Master of Computer Application

(Open and Distance Learning Mode)

Semester – I

Data Structures using C++

Centre for Distance and Online Education (CDOE)

DEVI AHILYA VISHWAVIDYALAYA, INDORE

"A+" Grade Accredited by NAAC

IET Campus, Khandwa Road, Indore - 452001

www.cdoedavv.ac.in

www.dde.dauniv.ac.in

Program Coordinator

Dr. Anand More
School of Computer Science and IT
Devi Ahilya Vishwavidyalaya, Indore – 452001

Content Design Committee

Dr. Pratosh Bansal
Centre for Distance and Online Education
Devi Ahilya Vishwavidyalaya, Indore – 452001

Dr. C.P. Patidar
Institute of Engineering & Technology
Devi Ahilya Vishwavidyalaya, Indore – 452001

Dr. Shaligram Prajapat
International Institute of Professional Studies
Devi Ahilya Vishwavidyalaya, Indore – 452001

Language Editors

Dr. Arti Sharan
Institute of Engineering & Technology
Devi Ahilya Vishwavidyalaya, Indore – 452001

Dr. Ruchi Singh
Institute of Engineering & Technology
Devi Ahilya Vishwavidyalaya, Indore – 452001

SLM Author(s)

Mr. Manoj Kumar Pawaiya
B.E., M.E.
IET, Devi Ahilya Vishwavidyalaya, Indore – 452001

Mrs. Sunita Goud
M. Tech.
SCS, Devi Ahilya Vishwavidyalaya, Indore – 452001

Copyright : Centre for Distance and Online Education (CDOE), Devi Ahilya Vishwavidyalaya
Edition : 2022 (Restricted Circulation)
Published by : Centre for Distance and Online Education (CDOE), Devi Ahilya Vishwavidyalaya
Printed at : University Press, Devi Ahilya Vishwavidyalaya, Indore – 452001

Data Structures using C++

Table of Contents

Introduction

Module 1 - INTRODUCTION

Unit 1 – Introduction

1.0 Introduction

1.1 Unit Objectives

1.2 Concept of Data type

1.3 Data structures

1.4 Abstract Data types

1.5 Summary

1.6 Key Terms

1.7 Check Your Progress

Unit 2 – Primitive and Non- primitive Data Structures

2.0 Introduction

2.1 Unit Objectives

2.2 Primitive Data Structures

2.3 Operations on Primitive Data Structures

2.4 Non- primitive Data Structures

  2.4.1 Linear Data Structures

  2.4.2 Non- Linear Data Structures

2.5 Operations on Non- primitive Data Structures

2.6 Summary

2.7 Key Terms

2.8 Check Your Progress

Unit 3 – Linked Lists

3.0 Introduction

3.1 Unit Objectives

3.2 Singly-Linked Lists

  3.2.1 Memory Representation

  3.2.2 Operations

3.3 Circular Linked Lists

  3.3.1 Traversing

3.3.2 Insertion
3.3.3 Deletion
3.4 Doubly-Linked Lists
3.4.1 Insertion
3.4.2 Deletion
3.5 Dynamic Storage Management: Application of a Doubly-Linked List
3.6 Generalized Lists
3.7 Garbage Collection
3.8 Summary
3.9 Key Terms
3.10 Check Your Progress

Module 2 Other Data Structures

Unit 4 – Stacks
4.0 Introduction
4.1 Unit Objectives
4.2 Stacks
4.3 Representation of Stacks
4.4 Stack Operations
4.5 Stack Applications
4.6 Summary
4.7 Key Terms
4.8 Check Your Progress

Unit 5 – Queues
5.0 Introduction
5.1 Unit Objectives
5.2 Basic terminology of Queues
5.3 Queue Operations
5.4 Representation of a Queue
5.4.1 Using an array
5.4.2 Using a Linked List
5.5 Various Queue Structures
5.5.1 Circular Queue

5.5.2 Priority Queue
5.6 Summary
5.7 Key Terms
5.8 Check Your Progress
Unit 6 – Trees
6.0 Introduction
6.1 Unit Objectives
6.2 Basic terminology of Trees
6.3 Binary Trees
6.4 Representation of a Binary Tree
6.4.1 Array Representation
6.4.2 Linked Representation
6.5 Binary Tree Traversals
6.6 Binary Search Tree
6.7 Threaded Binary Tree
6.8 Summary
6.9 Key Terms
6.10 Check Your Progress
Unit 7 – Graphs
7.0 Introduction
7.1 Unit Objectives
7.2 Graph Terminologies
7.3 Types of Graphs
7.3.1 Classification on the basis of Edge Connectivity
7.3.2 Classification on the basis of Direction
7.3.3 Classification on the basis of Weight or Level
7.3.4 Classification on the basis of Connectivity
7.4 Representation of Graphs
7.4.1 Set Representation
7.4.2 Linked Representation
7.4.3 Matrix Representation
7.5 Graph Traversal Algorithms
7.5.1 Breadth-First Search Algorithm

7.5.2 Depth-first Search Algorithm
7.6 Shortest Path Algorithms
7.6.1 Minimum Spanning Trees
7.6.2 Prim's Algorithm
7.6.3 Kruskal's Algorithm
7.6.4 Dijkstra's Algorithm
7.7 Summary
7.8 Key Terms
7.9 Check Your Progress

Module 3 - Types of Trees

Unit 8 – Balanced Trees
8.0 Introduction
8.1 Unit Objectives
8.2 Basic Terminology
8.3 AVL Trees
8.4 Weight Balanced Trees
8.5 Summary
8.6 Key Terms
8.7 Check Your Progress
Unit 9 B- Trees
9.0 Introduction
9.1 Unit Objectives
9.2 B- Trees
9.2.1 Operations on a B- Tree
9.3 B+ Trees
9.3.1 Operations on a B+ Tree
9.4 Red-Black Trees
9.4.1 Inserting a Node in a Red-black Tree
9.4.2 Deleting a Node from a Red-black Tree
9.5 Splay Trees
9.6 Summary
9.7 Key Terms

9.8 Check Your Progress

Unit 10 – Advanced Trees

10.0 Introduction

10.1 Unit Objectives

10.2 Interval Trees

10.3 Segment Trees

10.4 KD-Trees

10.5 Quad Trees

10.6 Summary

10.7 Key Terms

10.8 Check Your Progress

Module 4 - Indexing, Searching & Sorting

Unit 11 – Indexing

11.0 Introduction

11.1 Unit Objectives

11.2 File Organization

11.3 Indexing

11.3.1 Ordered Indices

11.3.2 Dense and sparse indices

11.3.3 Cylinder surface indices

11.3.4 Multi-level indices

11.3.5 Inverted indices

11.3.6 B-Tree indices

11.3.7 Harshed indices

11.4 Summary

11.5 Key Terms

11.6 Check Your Progress

Unit 12- Searching

12.0 Introduction

12.1 Unit Objectives

12.2 Searching and its types

12.2.1 Linear Search
12.2.2 Binary Search
12.3 Interpolation Search
12.4 Jump Search
12.5 Comparison of different search algorithms
12.6 Summary
12.7 Key Terms
12.8 Check Your Progress
Unit 13- Sorting
13.0 Introduction
13.1 Unit Objectives
13.2 Sorting
13.3 Internal Sorting
13.3.1 Insertion Sort
13.3.2 Bubble Sort
13.3.3 Selection Sort
13.3.4 Heap Sort
13.3.5 Merge Sort
13.3.6 Quick Sort
13.3.7 Shell Sort
13.4 Comparison of different sorting algorithms
13.5 External Sorting
13.6 Summary
13.7 Key Terms
13.8 Check Your Progress

INTRODUCTION

A data structure is defined as a set of data elements that represents operations such as insertion, deletion, modification and traversal of the values present in the data elements. Data elements are the data entries that are stored in the memory for organizing and storing data in an ordered and controlled way. The commonly used data structures in a programming language as C, such as are arrays, linked lists, stacks and trees. Data structures are of two types, linear and nonlinear.

The study material is divided into four modules each containing units for the relevant topics.

Module-1 is further divided into three units. Unit-1 & 2 provide the basic introduction to data structures and its types. Unit-3 discusses linked lists. It explains the various types of linked lists such as singly linked, doubly linked and circular linked lists. The unit also discusses how different operations can be performed on different types of linked lists.

Module-2 describes the data structures like stacks, queues, Trees, and Graphs including their representations and types. The basic operations performed on these data structures are explained in detail.

Types of advanced trees, their terminology, basic concepts, and applications are discussed in Module-3. It includes AVL trees, balanced trees, B-Trees, B+ trees, Quad trees, and KD trees.

Module-4 depicts the concept of indexing in file organization in the computer's memory. Unit-12 & 13 also deals with searching and sorting, including the use of various data structures for searching and sorting.

This content is designed comprehensively and follows a simple approach, keeping in mind the syllabus of the program. It exhilarates interest and is sure to stimulate knowledge among the readers. Numerous figures and tables, key terms help in simplifying learning about the subject. The 'Check Your Progress' section intends the readers to test their knowledge. It is hoped that the language and the content demonstration is coherent to the readers and will enhance their learning in the best way possible.

All the Best!

Module: 1

Introduction

Unit 1 Introduction

Structure

1.0 Introduction

1.1 Unit Objectives

1.2 Concept of Data type

1.3 Data structures

1.4 Abstract Data types

1.5 Summary

1.6 Key Terms

1.7 Check Your Progress

1.0 Introduction

All computer programs involve operations on data. Data plays an important role in programming. The data may be defined as a value or a set of values, such as the name and age of a person, the grade of a student, the salary of an employee, and so on. It is just a collection of values and no conclusion can be drawn from it. However, after processing, it becomes information that can help in making decisions. In order to process data, it should be available in the main memory since the processor can only act on data in the main memory. In order to represent the data in the main memory, some model is needed to process it efficiently. This model is called a data structure. Various models are available, and this unit will introduce you to these various structures and the different operations that can be performed on them.

1.1 Unit Objectives

After going through this unit, the reader will be able to:

- Understand the basic concept of data type
- Explain the different types of data structures
- Describe the abstract data types

1.2 Concept of Data type

Data is a value or a collection of values that may be obtained from an experiment, survey, etc. The term which is used to refer to a single unit of values is called a *data item*. A data item may be a group item or an elementary item. A *group item* is the one

that can be further divided into sub-items, whereas an *elementary item* is the one that cannot be divided further. The address of a person, for example, is a group item because it is usually divided into sub-items like house number, street, city, state, PIN code, etc. On the other hand, the state, city, PIN code, etc., are elementary items.

A set of data items are used to represent a thing in the real world with the physical or the logical existence called an *entity*. In the context of entities, the data items are termed as *attributes* or *properties* of the entity. The attributes like name, roll number, marks, and so on, for example, can be used to represent an entity student. Generally, each attribute of an entity is assigned a particular value, such as the name is assigned a value 'James'. A set of entities having similar attributes is called an *entity set*, e.g., all the students of a class, all the employees of an organization, and so on.

Data type refers to the kind of data that may appear in the computation of any program. Some of the frequently used data types are Real, boolean, character, complex, numeric (integer), date, alphanumeric, graphics, string, Image, etc.

The syntax for declaring data type and the variable name is given below:

Syntax:

<(data type)><variable names>;

The data types can be broadly classified as Built-in data type and abstract data type. Every programming language contains a set of data types called ***built-in data types***. For example,

Data types in C: int, float, char, double, enum, etc.

Data types in FORTRAN: INTEGER, REAL, LOGICAL, COMPLEX, DOUBLE PRECISION, CHARACTER, etc.

Pascal: Integer, Real, Character, Boolean, etc.

The built-in data types are advantageous in processing various types of data. For instance, if a variable is declared of the type Real, then several things are automatically implied, such as how to store a value for that variable, what amount of memory is required to store, etc.

When a program requires a special type of data that is not available as a built-in data type, then it is implemented by the user on its own. This implemented special type of

data is termed as an ***abstract data type***. It is also known as a ***user-defined data type.*** In such a type of data, the user has to give more effort regarding how to store value for that data, operations to manipulate variables of that kind of data, etc. For example, to process a date (dd/mm/yy), no built-in data type is available in C, FORTRAN, and Pascal. This can be accomplished using an abstract data type.

1.3 Data Structures

The logical or mathematical model used to organize the data in main memory is called a *data structure*. Various data structures are available each with its special features. These features should be kept in mind while choosing a data structure for a particular situation. Generally, the choice of any data structure depends on its simplicity and effectiveness in processing data. In addition, we also consider how well it represents the actual relationship between the data in the real world.

Data structures are helpful for programmers to manipulate and store the data effectively. They help in establishing the relationship of one data element with other data elements. Various methods are provided by data structures to organize and represent the data in the computer's memory. The data structures also govern the dynamic behavior towards data handling.

Different ways of data organization require different kinds of data structures. Basically, two complementary goals are implemented by data structures. The first goal is to develop mathematical entities and operations to solve particular problems. The second goal is to search for suitable representations for these entities and then carry out the desired operations. The second goal requires high-level data types to solve the problems. These high-level data types use existing data types.

Generally, the following additional goals are involved in producing quality data structure to have quality software implementation:

- **Correctness:** The design of a data structure should be such that it can operate correctly for all kinds of input, based on the domain of interest. For example, a data structure designed to store a collection of numbers, in a specific order, must make sure that the numbers are not stored in an unorganized way.
- **Efficiency:** The data structure must be efficient to process the data at high

speed without utilizing much of the computer memory.

On the basis of their implementation, the data structures are divided into two categories, namely *Primitive data structure* and *Non-primitive data structure*. The primitive data structures further include *Integer, Real, Character, and Boolean*. The non-primitive data structures are further divided into two groups, *Linear* and *Non-linear data structures*. Arrays, linked lists, stacks, and queues are linear data structures while Trees and graphs are non-linear data structures. All these are discussed in the upcoming units in detail.

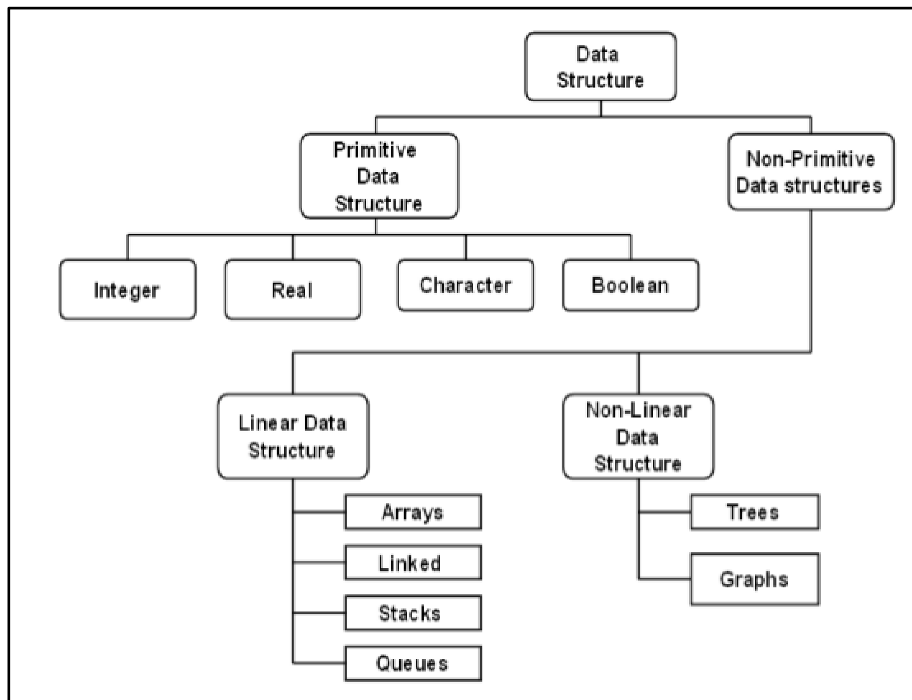


Figure 1.1 Classification of Data Structures

1.4 Abstract Data Types

The data type whose behavior is defined as a set of values and a set of operations is known to be Abstract data type (ADT). ADT only mentions what operations are to be performed but not about the process of the operations to be implemented. It is known as *"abstract"* as it provides an implementation-independent view. Hiding the main details and providing only the essentials is known as ***abstraction***.

An Abstract Data Type [ADT] consists of two parts, namely, a ***value definition*** and an ***operator definition***. A value definition consists of a definition clause and a

condition clause. The operator definition consists of three parts: a header, preconditions, and post-conditions. The preconditions and post-conditions are optional and can be used depending on the program requirement.

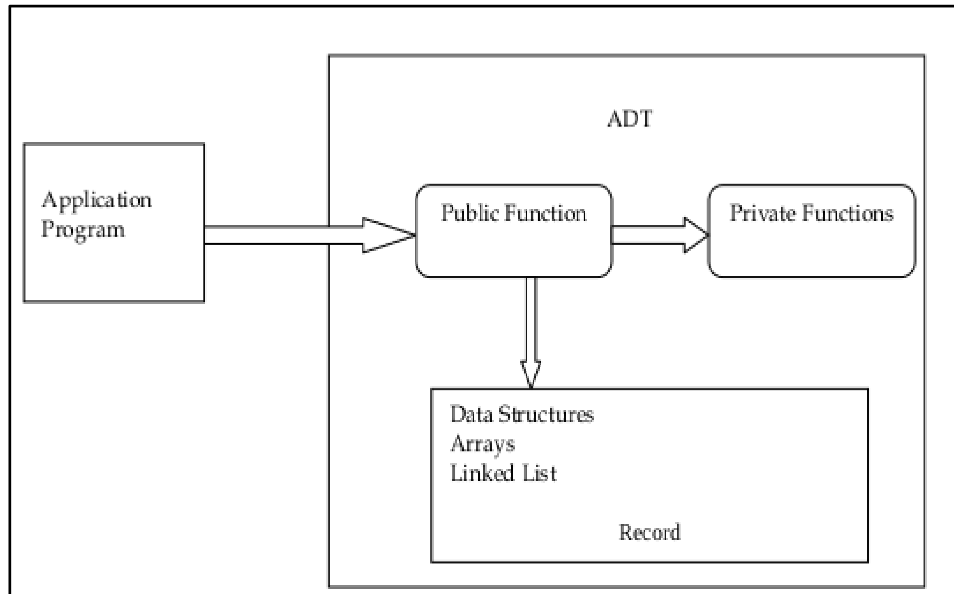


Figure 1.2 Abstract Data Type (ADT) Model

As shown in figure 1.2, the ADT model has two different parts- Functions (public and private) and Data structures. Both of these parts are confined to ADT and not a part of the application program. Data is entered, accessed, modified, and deleted through the external application programming interface. This interface can access only public functions. Every ADT operation has an algorithm to perform a specific task.

Generally, three types of ADTs are defined: List ADT, Stack ADT, and Queue ADT.

- **List ADT**

  The data is generally stored in a sequence in the form of a list that has a head structure consisting of *count*, *pointers*, and *address of compare function* needed to compare the data in the list. The data node contains the *pointer* to a data structure and another *pointer* that points to the next node in the list. Some of the operations performed on such list ADTs are:

  - ➔ get() – Return an element from the list at any given position.
  - ➔ insert() – Insert an element at any position of the list.
  - ➔ remove() – Remove the first occurrence of any element from a non-empty

list.

- ➔ removeAt() – Remove the element at a specified location from a non-empty list.
- ➔ replace() – Replace an element at any position by another element.
- ➔ size() – Return the number of elements in the list.
- ➔ isEmpty() – Return true if the list is empty, otherwise return false.
- ➔ isFull() – Return true if the list is full, otherwise return false.

- **Stack ADT**

  In stack ADTs, the pointer to data is stored in the nodes in place of the data itself. Memory is allocated for the data and the address is sent to the stack ADT. The head node and data nodes are encapsulated in ADT. Only a pointer to the stack is displayed to the calling function. The stack head contains a pointer to the *top* and also a *count* of the number of entries present in the stack. Conceptually, a stack is an arrangement of the same type of elements in sequential order. The operations take place only at the top end of the stack. Some of the operations performed on stack ADTs are:

  - ➔ push() – Insert an element at one end of the stack called top.
  - ➔ pop() – Remove and return the element at the top of the stack, if it is not empty.
  - ➔ peek() – Return the element at the top of the stack without removing it, if the stack is not empty.
  - ➔ size() – Return the number of elements in the stack.
  - ➔ isEmpty() – Return true if the stack is empty, otherwise return false.
  - ➔ isFull() – Return true if the stack is full, otherwise return false.

- **Queue ADT**

  In a queue ADT, each node contains a void pointer to the *data* and the *link pointer* to the next element in the queue. The same type of elements are arranged in sequential order and operations take place at both the ends of a queue. Insertion is done at the end while deletion is done at the front. Some of the operations performed on stack ADTs are:

  - ➔ enqueue() – Insert an element at the end of the queue.
  - ➔ dequeue() – Remove and return the first element of the queue, if the queue is not empty.

- ➔ peek() – Return the element of the queue without removing it, if the queue is not empty.
- ➔ size() – Return the number of elements in the queue.
- ➔ isEmpty() – Return true if the queue is empty, otherwise return false.
- ➔ isFull() – Return true if the queue is full, otherwise return false.

1.5 Summary

- Data is a value or a collection of values that may be obtained from an experiment, survey, etc.
- A set of data items are used to represent a thing in the real world with the physical or the logical existence called an *entity*.
- Data type refers to the kind of data that may appear in the computation of any program. Some of the frequently used data types are Real, boolean, character, complex, numeric (integer), date, alphanumeric, graphics, string, Image, etc. The data types can be broadly classified as Built-in data type and abstract data type.
- The logical or mathematical model used to organize the data in main memory is called a *data structure*. The data structures are divided into two categories, namely *primitive data structure* and *non- primitive data structure*.
- The data type whose behavior is defined as a set of values and a set of operations is known to be Abstract data type (ADT). ADT only mentions what operations are to be performed but not about the process of the operations to be implemented.

1.5 Key Terms

- **Data item:** A single unit of values.
- **Entity:** A set of data items used to represent a thing in the real world with the physical or the logical existence.
- **Data structure:** The logical or mathematical model used to organize the data in main memory.
- **Abstraction:** Hiding the main details and providing only the essentials is known as abstraction.
- **Application Program:** A program designed to perform a particular function

directly for the user or for another application program.

1.6 Check Your Progress

Short- Answer type

Q1) Abstract Data types provide an implementation-independent view. True/ False?

Q2) Define a data structure.

Q3) Which of the following is not a linear data structure?

(a) Stack (b) Queue (c) Tree (d) Linked List

Q4) _________ are also known as user-defined data types.

Q5) The two different parts of the ADT model are functions and __________.

Long- Answer type

Q1) Differentiate between Built-in and Abstract Data types

Q2) Explain the basic concept of data types.

Q3) Give the classification of Data structures.

Q4) What are Abstract Data Types? Explain its types.

Q5) Briefly explain the Abstract data type (ADT) model.

References

- *Data Structures with C*, Lipschutz, Seymour, Delhi: Tata McGraw Hill.
- *Classic Data Structures,* Debasis Samanta, PHI Learning Pvt. Ltd. 2nd Edition.

Unit 2 – Primitive and Non-Primitive Data Structures

Structure

2.0 Introduction
2.1 Unit Objectives
2.2 Primitive Data Structures
2.3 Operations on Primitive Data Structures
2.4 Non- primitive Data Structures
    2.4.1 Linear Data Structures
    2.4.2 Non- Linear Data Structures
2.5 Operations on Non- primitive Data Structures
2.6 Summary
2.7 Key Terms
2.8 Check Your Progress

2.0 Introduction

We are already familiar with the definition of a data structure. Data structure provides a set of variables being associated with each other in different ways. Various programming languages utilize these variables to represent relationships between data elements and help in the easy processing of data. According to their utilization in distinct programming languages, data structures are classified as Primitive and non-primitive data structures. This unit explains the use and representation of Primitive data structures in detail.

2.1 Unit Objectives

After going through this unit, the reader will be able to:

- Understand the basic concept of primitive data structures.
- Explain the use and representation of different primitive data structures.

2.2 Primitive Data Structures

The primitive data structures are used to represent numbers and characters that are included in the built-in programs. They are the basic data types of any programming language and are not composed of other data types. They can be manipulated or can

even be operated directly with the help of machine-level instructions. Primitive data structures include basic data types like Integer, Real, Character, and Boolean.

1. **Integer**

   The integral or fixed-precision values are represented by integers (denoted as ***int***). The type INTEGER includes a subset of the whole numbers with a variation in size for different computer systems. All the operations on this type of data are precise and also follow the basic laws of arithmetic. Though arithmetic operations yield accurate results if the result of such operations lies outside the subset, then the computation might fail.

2. **Real**

   The REAL primitive data type includes a subset of the real numbers. The arithmetic operations performed on the REAL data types can be incorrect within the limits of round-off errors while the arithmetics performed on INTEGER type yield accurate results. This is considered to be the main difference between the INTEGER and REAL types.

3. **Character**

   Denoted as ***Char***, the character primitive data type consists of a set of structured and printable characters with 26 upper case letters, 26 lower case letters, 10 decimal digits, and various other graphic characters like punctuation marks. Every computer system stores character data in a one-byte field as an integer value. One byte comprises 8 bits so it has 256 possibilities having positive values of 0 to 255.

4. **Boolean**

   This primitive data structure is used for Boolean values that are denoted by two identities TRUE and FALSE. The Boolean operators include logical conjunction (&), disjunction (OR), and negation (~).

2.3 Operations on Primitive Data Structures

Several operations can be performed on primitive data structures. Some of the general operations are:

1. ***Creation Operation:*** The creation operation creates a data structure. For example,

   int x;

Here, the above-declared statement will create 2 bytes of memory space for variable 'x'. This variable will be used to store only integer values.

2. ***Destroy Operation:*** The destroy operation destroys the data structure. In C language, 'free()' operation is used to destroy the data structure. This helps using the memory of the system efficiently.
3. ***Selection Operation:*** The selection operation is used to access data within a data structure. The significance of the selection operation is provided in a file data structure.
4. ***Update Operation:*** To modify data in the data structure, the update operation is used.

2.4 Non- primitive Data Structures

The data structures derived from primitive data structures are known as Non-primitive data structures. These data structures cannot be manipulated or operated directly by machine-level instructions. A set of either homogeneous (same data type) or heterogeneous (different data type) data elements are formed. These are further divided into linear and non-linear data structures based on the structure and arrangement of data.

2.4.1 Linear Data Structures

The linear data structure is the one in which its elements form a sequence. It means each element in the structure has a unique predecessor and a unique successor. An array is the simplest linear data structure. Various other linear data structures are linked lists, stacks, and queues.

- **Arrays**

  A finite collection of homogeneous elements is termed as an *array*. Here, the word 'homogenous' indicates that the data type of all the elements in the collection should be the same, that is, int or char or float or any other built-in or user-defined data type. However, an array cannot have elements of two or more data types together.

  Elements of an array are always stored in contiguous memory locations irrespective of the array size. The elements of an array can be referred to by using one or more *indices* or *subscripts*. An index or subscript is a positive

integer value that indicates the position of a particular element in the array. If the number of subscripts required to access any particular element of an array is one, then it is a *single-dimensional array*. Otherwise, it is a *multidimensional array*. *The multi-dimensional array* may be a two-dimensional *array* or even more.

Consider a single-dimensional array Arr with size n, where n is the maximum number of elements that Arr can store. Mathematically, the elements of Arr are denoted as Arr_1, Arr_2, Arr_3,..., Arr_n. In different programming languages, array elements are denoted by different notations, such as by parenthesis notation or by bracket notation. Table 1.1 shows the notation of elements of a single-dimensional array Arr with size n in different programming languages.

Table 1.1 Different Notations of a Single-dimensional Array

S.No.	Notation	Programming Language(s)
1.	Arr(1), Arr(2), Arr(3),.... Arr(n)	BASIC and FORTRAN
2.	Arr[1], Arr[2], Arr[3],...... Arr[n]	Pascal
3.	Arr[0], Arr[1], Arr[2],.... Arr[n-1]	C, C++, and Java

Note that in the languages like BASIC, PASCAL, and FORTRAN, the smallest subscript value is 1 and the largest subscript value is n. On the other hand, in languages like C, C++, and Java, the smallest subscript value is 0 and the largest subscript value is n–1. In general, the smallest subscript value used to access an array element is the lower bound (L_b) and the largest subscript value is the upper bound (U_b).

In two-dimensional arrays, the elements can be viewed as arranged in the form of rows and columns (matrix form). To access an element of a two-dimensional array, two subscripts are used—the first one represents the row number and the second one represents the column number. Consider a two-dimensional array Arr with size m*n, where m and n represent the number of rows and columns respectively. Mathematically, the array Arr is denoted as Arrij, where i and j indicate row number and column number with i< = m and j< = n. Table 1.2 shows the notation of elements of a two-dimensional array Arr in different

programming languages.

Table 1.2 Different Notations of a Two-dimensional Array

S.No.	Notation	Programming Language(s)
1.	Arr(i, j) with 0<i<=m and 0<j<=n	BASIC and FORTRAN
2.	Arr[i, j] with 0<i<=m and 0<j<=n	Pascal
3.	Arr [i] [j] with 0<=i<m and 0<=j<n	C, C++, and Java

- **Linked Lists**

  A linked list is a linear collection of similar data elements, called ***nodes***, with each node containing some data and pointer(s) pointing to other nodes (s) in the list. Nodes of a linked list are not constrained to be at contiguous memory locations; instead, they can be stored anywhere in the memory. The linear order of the list is maintained by the pointer field(s) in each node.

  Depending on the pointer field(s) in each node, linked lists can be of different types. If each node of a linked list contains only one pointer and it points to the next node, then it is called a ***linear linked list*** or ***singly-linked list***. In such types of lists, the pointer field in the last node contains **NULL**. However, if the pointer in the last node is modified to point to the first node of the list, then it is called a ***circular linked list***. In addition to the pointer to the next node, each node of a linked list can also contain a pointer to its previous node. Such a type of linked list is called a ***doubly-linked list***. Figure 1.2 shows a singly, circular, and doubly-linked list with five nodes each.

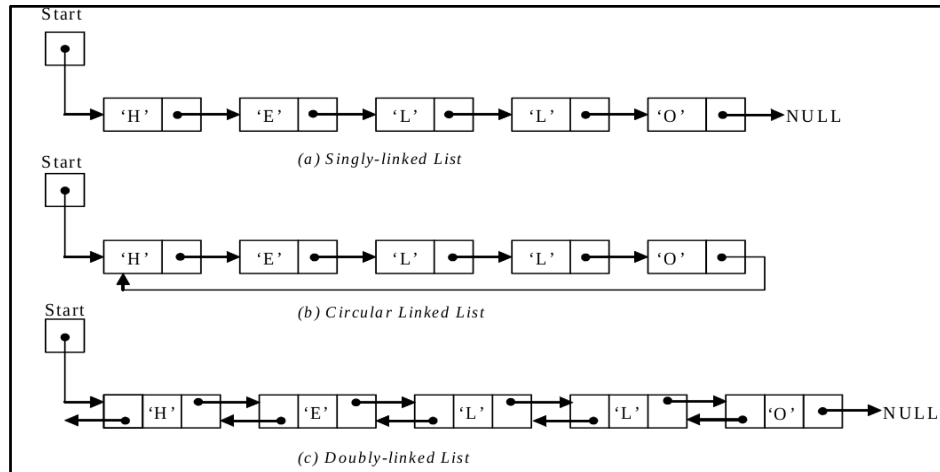


Figure 2.1 Various Types of Linked Lists

- **Stacks and Queues**

  A *stack* is a linear list of data elements in which the addition of a new element or deletion of an element occurs only at one end. This end is called ***Top*** of the stack. The operation of adding a new element in the stack and deleting an element from the stack is called ***push*** and ***pop*** respectively. Since the addition and deletion of elements always occur at one end of the stack, the last element that is pushed onto the stack is the first one to come out. Therefore, a stack is also called a ***Last-In-First-Out*** (LIFO) list.

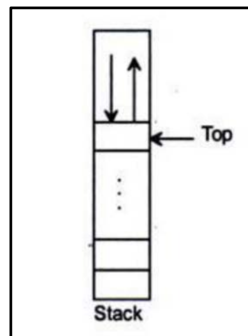


Figure 2.2 A stack

A *queue* is a linear data structure in which the addition or insertion of a new element occurs at one end, called ***Rear***, and deletion of an element occurs at the other ends, called ***Front***. Since insertion and deletion occur at opposite ends of the queue, the first element that is inserted in the queue is the first one to come out. Therefore, a queue is also called a ***First-In-First-Out*** (FIFO) list.

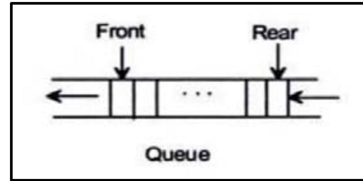


Figure 2.3 A queue

2.4.2 Non- Linear Data Structures

A non-linear data structure is one in which its elements do not form a sequence. It means, unlike linear data structure, each element is not constrained to have a unique predecessor and a unique successor. Trees and graphs are the two data structures that come under this category.

- **Trees**

  Usually, we observe a hierarchical relationship between various data elements. This hierarchical relationship between data elements can easily be represented using a non-linear data structure called *trees*. A tree consists of multiple nodes with each node containing zero, one or more pointers to other nodes called ***child nodes***. Each node of a tree has exactly one parent except a special node at the top of the tree called the ***root node***.

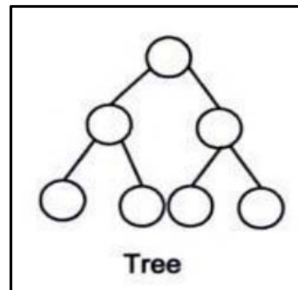


Figure 2.4 A Tree

- **Graphs**

  A graph consists of a finite set of nodes (or vertices) and a set of edges connecting them. The graph is used to represent the non-hierarchical relationship among pairs of data elements. Let's say, a graph G(V, E) consists of a pair of two non-empty sets V and E, where V is a set of vertices or nodes and E is a set of edges. The data elements become the vertices of the graph and the relationship is shown by edges between the two vertices. For example,

assume four places W, X, Y, and Z, such that:

- ➔ There exists some path from X to Y, X to W, Y to W, Y to Z, and Z to W.
- ➔ There is no direct path from X to Z.

We can simply represent this situation using a graph where the places W, X, Y, and Z are represented as the nodes of the graph, and a path from one place to another place is represented by an edge between them.

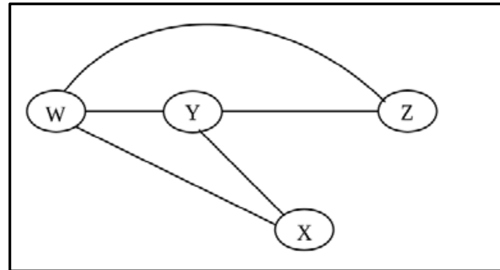


Figure 2.5 An example of a graph

It is clear from the figure that each node can have links with multiple other nodes. This analogy suggests that it is similar to a tree. However, unlike trees, there is no root node in a graph. Further, graphs show relationships that may be non-hierarchical in nature. It means there is no parent and child relationship. However, a tree can be considered as a variant or a special type of graph.

2.5 Operations on Non- primitive Data Structures

The logical organization of data and their storage structure govern the operations on the non-primitive data structure. A set of either homogeneous (same data type) or heterogeneous (different data type) data elements are formed in non-primitive data structures. Therefore, these data structures cannot be operated or manipulated directly by the machine-level instructions.

Some of the general operations on non-primitive data structures are:

1. ***Traversing:*** The method in which each element of the data structure is processed exactly once is known as Traversing. This method is used for checking the availability of data elements in an array. Traversing is even used to check if the element is successfully inserted or deleted.

2. ***Sorting:*** The method of arranging the data elements in a logical order, maybe ascending or descending order, is known as sorting. Sorted lists are required by some algorithms. Thus, efficient sorting becomes essential for optimizing these algorithms to ensure their accuracy.
3. ***Merging:*** Merging is a technique of combining the data elements of two different sorted lists into a single sorted list. The basic idea behind this method is based on the divide-and-conquer algorithm.
4. ***Searching:*** Searching is the method of finding the location of an element with a given key value, or finding the location of an element that satisfies a given condition. Searching helps in finding unambiguous items from the set of elements, just like a particular file from the memory of the system.
5. ***Insertion:*** Insertion operation is used to add a new element to the data structure. The insertion process may add a new element in the ith position of the data structure. In addition to insertion, if sorting also needs to be performed, first we need to assign an item to the given elements and compare it with the previous elements. If the assigned element is smaller than the previous element, we need to swap the positions of both these items. This process is repeated until the correct position of the item is Identified.
6. ***Deletion:*** Deletion means removing an item from the structure. When any node is not required in the data structure, it can be removed using the delete operation.

2.6 Summary

- The data structure can be classified into two categories: primitive data structure and non-primitive data structure.
- Basic data types such as integer, real, character, and Boolean are categorized under primitive data structures. These data types are also known as simple data types because they consist of characters that cannot be divided.
- Operations like creation, destroy, selection, and update are performed on the primitive data structures.
- Non-primitive data structures are further divided into linear and non-linear data structures based on the structure and arrangement of data. Arrays, linked lists, stacks, queues are examples of linear data structure. Trees and graphs

are examples of non-linear data structures.

- Some general operations that can be performed on non-primitive data structures are traversing, sorting, merging, searching, insertion, and deletion.

2.7 Key Terms

- **Array:** A finite collection of homogeneous elements.
- **Linked list:** A linear collection of similar data elements, called nodes, with each node containing some data and pointer(s) pointing to other nodes (s) in the list.
- **Stack:** A linear list of data elements in which the addition of a new element or deletion of an element occurs only at one end.
- **Queue:** A linear data structure in which the addition or insertion of a new element occurs at one end, called rear, and deletion of an element occurs at other ends, called the front.

2.8 Check Your Progress

Short- Answer type

Q1) Deletion operation occurs at the rear end of a queue. True/ False?

Q2) List four major operations on linear data structures.

Q3) A ______ is a linear list of data elements in which the addition of a new element or deletion of an element occurs only at one end.

Q4) If the number of subscripts required to access any particular element of an array is one, then it is a _________ array.

Q5) An array always contains similar data elements. True/ False?

Long- Answer type

Q1) Write a short note on:

a) Arrays
b) Trees

Q2) Why is the stack also called a Last-in First-out (LIFO) list?

Q3) Briefly explain the basic operations on Non- primitive data structures.

Q4) Differentiate between a linear data structure and a non-linear data structure.

Q5) Explain the different types of linked lists.

References

- *Data Structures with C*, Lipschutz, Seymour, Delhi: Tata McGraw Hill.
- *Data Structures Using C*, Reddy. A.M Padma (2006), Bangalore: Sri Nandi Publications.
- *Classic Data Structures,* Debasis Samanta, PHI Learning Pvt. Ltd. 2nd Edition.

Unit 3 – Linked Lists

Structure

3.0 Introduction
3.1 Unit Objectives
3.2 Singly-Linked Lists
3.2.1 Memory Representation 3.2.2 Operations
3.3 Circular Linked Lists
3.3.1 Traversing 3.3.2 Insertion
3.3.3 Deletion
3.4 Doubly-Linked Lists
3.4.1 Insertion 3.4.2 Deletion
3.5 Dynamic Storage Management: Application of a Doubly-Linked List
3.6 Generalized Lists
3.7 Garbage Collection
3.8 Summary
3.9 Key Terms
3.10 Check Your Progress

3.0 Introduction

In the simplest terms, a list refers to a collection of data items of similar type arranged in a sequence (that is, one after another), e.g., list of students' names, list of addresses, etc. One way to store such lists in memory is to use an array. However, arrays have certain problems associated with them. As array elements are stored in adjacent memory locations, a sufficient block of memory is allocated to an array at compilation time. Once the memory is allocated to an array, it cannot be expanded or contracted. This is why an array is called a static data structure. If the number of elements to be stored in an array increases or decreases significantly at run-time, it may require more memory space or may result in wastage of memory, both of which are unacceptable. Another problem is that the insertion and deletion of an element into an array are expensive operations, since they may require a number of elements to be shifted. As a result of these problems, arrays are not generally used to implement linear lists; instead, another data structure known as a linked list is used. A linked list is a linear collection of homogeneous elements called ***nodes***. The

successive nodes of a linked list need not occupy adjacent memory locations and the linear order between the nodes is maintained by means of pointers. This unit discusses different types of linked lists, such as singly-linked lists, circular linked lists, and doubly-linked lists, and the various operations, such as creation, traversal, search, insertion, and deletion, that can be performed on them. It also discusses how data structures, stacks, and queues are implemented using linked lists.

3.1 Unit Objectives

After going through this unit, the reader will be able to:

- Explain the singly linked list data structure.
- Understand the salient features and applications of circular linked lists.
- Understand the salient features and applications of doubly-linked lists, including dynamic storage management.
- Describe the applications of generalized lists.
- Explain the meaning and applications of garbage collection.

3.2 Singly-Linked Lists

In a singly-linked list (also called linear linked list), each node consists of two fields: ***info*** and ***next*** (Figure 3.1). The info field contains the data and the next field contains the address of the memory location where the next node is stored. The last node of the singly-linked list contains NULL in its next field that indicates the end of the list.

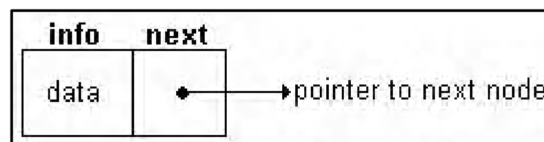


Figure 3.1 Node of a linked list

The data stored in the info field may be a single data item of any data type or a complete record representing a student or an employee or any other entity. In this unit, however, we assume the info field contains integer data. As each node of the list contains only a single pointer pointing to the next node (not to the previous node) thereby allowing traversing in only one direction, it is also referred to as a one-way list. Figure 3.2 shows a singly-linked list with four nodes.

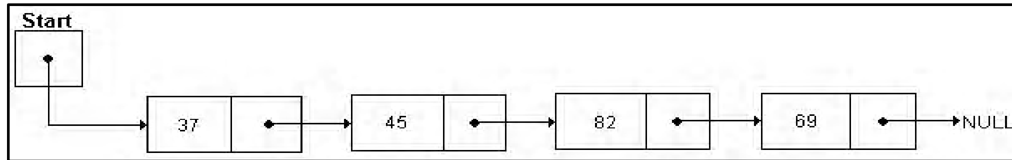


Figure 3.2 A Singly-linked List with Four Nodes

The linked list overcomes all the drawbacks of arrays. It is a dynamic data structure, which implies that the memory is allocated dynamically at run-time. Also, there is no upper limit on the size of the linked list and new nodes can be inserted into it as far as memory is available. Moreover, since the successive nodes of a linked list are stored in non-contiguous memory locations, nodes can be inserted or deleted without shifting of the existing nodes.

3.2.1 Memory Representation

To maintain a linked list in memory, two parallel arrays of equal size are used. One array (say, INFO) is used for the info field, and another array (say, NEXT) for the next field of the nodes of the list. The values in the arrays are stored such that the ith location in arrays INFO and NEXT contain, respectively, the info and next fields of a node of the list. In addition, a pointer variable Start is maintained in memory that stores the starting address of the list. Figure 3.3 shows the memory representation of a linked list where each node contains an integer.

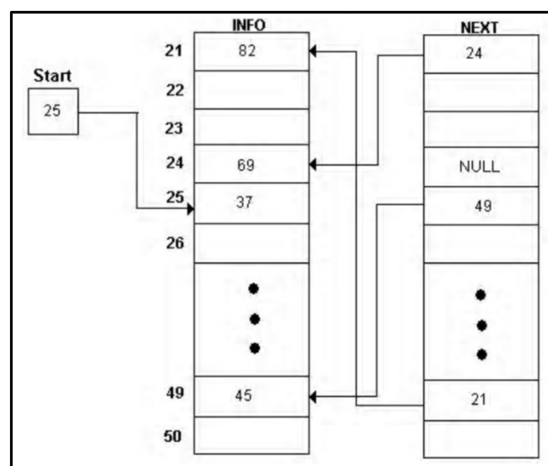


Figure 3.3 Memory Representation of a Singly- linked list

In this figure, the pointer variable Start contains 25, that is, the address of the first

node of the list, which stores the value 37 in array INFO and its corresponding element in array NEXT stores 49, that is, the address of the next node in the list, and so on. Finally, the node at address 24 stores value 69 in array INFO and NULL in array NEXT, thus, it is the last node of the list. Note that the values in array INFO are stored randomly and the array NEXT is used to keep track of the values in the list.

Memory allocation

As memory is allocated dynamically to the linked list, a new node can be inserted anytime in the list. For this, the memory manager maintains a special linked list known as a free-storage list or memory bank or free pool that consists of unused memory cells. This list keeps track of the free space available in the memory and a pointer to this list is stored in a pointer variable Avail (Figure 3.4). Note that the end of the free-storage list is also denoted by storing NULL in the last available block of memory.

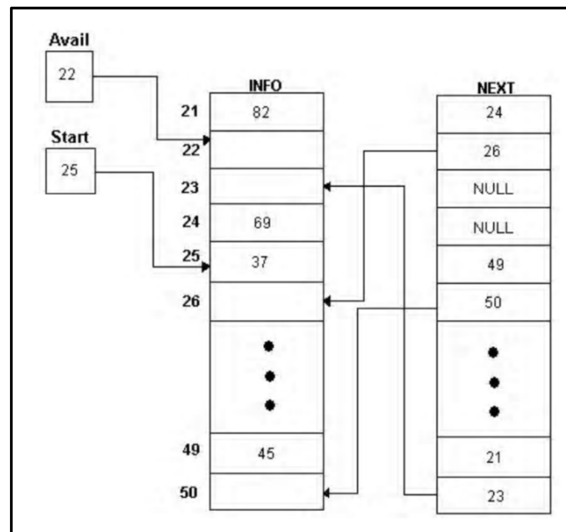


Figure 3.4 Free- storage List

In this figure, Avail contains 22, hence, INFO[22] is the starting point of the free-storage list. Since NEXT[22] contains 26, INFO[26] is the next free memory location. Similarly, other free spaces can be accessed and the NULL in NEXT[23] indicates the end of the free-storage list. While creating a linked list or inserting an element into a linked list, whenever a request for the new node arrives, the memory manager searches through the free- storage list for the block of the desired size. If the block of

the desired size is found, it returns a pointer to that block. However, sometimes there is no space available, that is, the free-storage list is empty. This situation is termed as overflow. In this situation, the memory manager replies accordingly.

3.2.2 Operations

A number of operations can be performed on the singly-linked lists. These operations include traversing, searching, inserting, and deleting nodes, and reversing, sorting, and merging linked lists. Before implementing these operations, first, we need to understand how a node of a linked list is created.

Creating a Node

Creating a node means defining its structure, allocating memory to it, and its initialization. As discussed earlier, the node of a linked list consists of data and a pointer to the next node. To define a node containing integer data and a pointer to the next node in C language, we can use a self-referential structure whose definition is shown here.

```
typedef struct node
{
    int info;                    /*to store integer type data*/
    struct node *next;           /*to store a pointer to next node*/
} Node;
    Node *nptr;                  /*nptr is a pointer to node*/
```

After declaring a pointer nptr to the new node, the memory needs to be allocated dynamically to it. If the memory is allocated successfully (that is, no overflow), the node is initialized. The info field is initialized with a valid value and the next field is initialized with NULL .

Algorithm 3.1 Creation of Node

create_node()
1. Allocate memory for nptr   //nptr is a pointer to the new node
2. If nptr = NULL
    Print “Overflow: Memory not allocated!” and go to step 7
  End If

```
3. Read item                         //item is the value to be inserted in the new node
4. Set nptr->info = item
5. Set nptr->next = NULL
6. Return nptr                       //returning pointer nptr
7. End
```

Now, the linked list can be formed by creating several nodes of type Node and inserting them either in the beginning or at the end or at a specified position in the list.

Traversing

Traversing a list means accessing its elements one by one to process all or some of the elements. For example, if we need to display values of the nodes, count the number of nodes, or search a particular item in the list, then traversing is required. We can traverse the list by using a temporary pointer variable (say, temp), which points to the node currently being processed. Initially, we make temp point to the first node, a process that element, then move temp to point to the next node using statement temp=temp->next, a process that element and move so on as long as the last node is not reached, that is, until temp becomes NULL.

Algorithm 3.2 Traversing a list

```
display(Start)
1. If Start = NULL                  //Start points to the first node of the list
     Print "List is empty!!" and go to step 4
   End If
2. Set temp = Start                 //initialising temp with Start
3. While temp != NULL
     Print temp->info               //displaying value of each node
     Set temp = temp->next          //moving temp to point to next node
   End While
```

Insertion

To insert a node in the linked list, a new node is created (as explained in Algorithm 3.1) and then placed at the desired position by adjusting the pointers. Nodes can be inserted either in the beginning or at the end or at any specified position in the list as discussed in this section.

a) Insertion at the beginning: To insert a node at the beginning of the list, the next field of the new node (pointed to by nptr) is made to point to the existing first node and the Start pointer is modified to point to the new node (Figure 3.5).

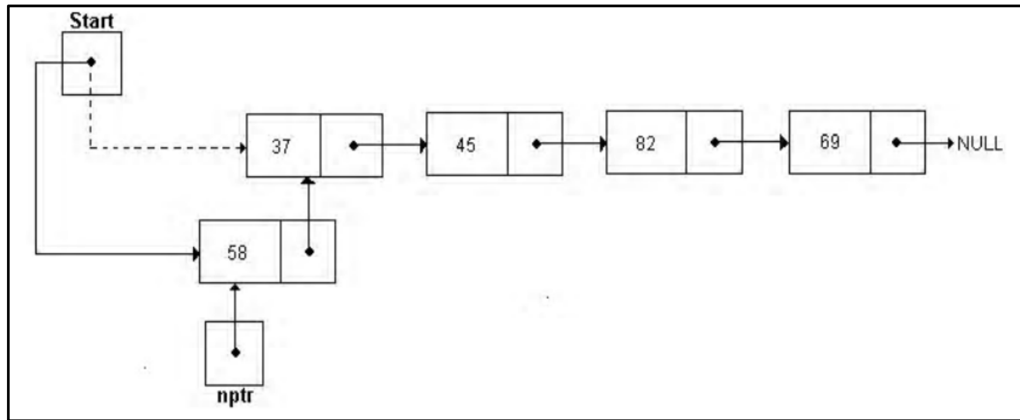


Figure 3.5 Insertion at the Beginning of a Linked List

Algorithm 3.4 Insertion in Beginning	
insert_beg(Start)	
1. Call create_node()	*//creating a new node pointed to by nptr*
2. Set nptr->next = Start	
3. Set Start = nptr	*//Start pointing to new node*
4. End	

b) Insertion at the end: To insert a node at the end of a linked list, the list is traversed up to the last node and the next field of this node is modified to point to the new node. However, if the linked list is initially empty, then the new node becomes the first node, and Start points to it. Figure 3.6(a) shows a linked list with a pointer variable temp pointing to its first node and Figure 3.6(b) shows temp pointing to the last node and the next field of last node pointing to the new node.

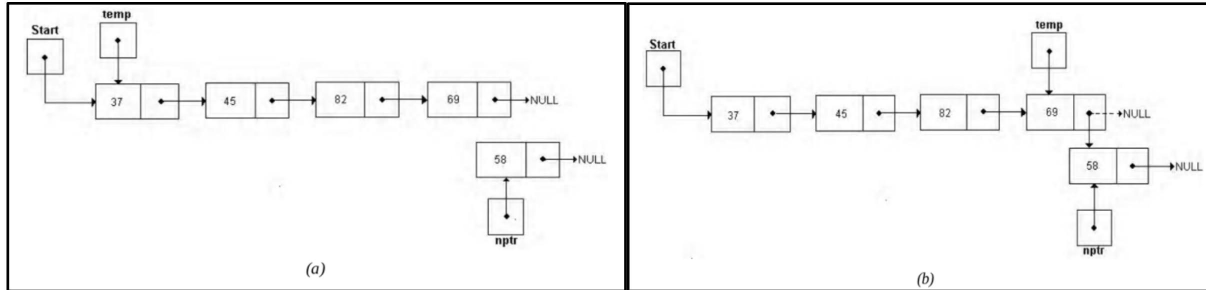


Figure 3.6 Insertion at the End of a Linked List

Algorithm 3.5 Insertion at End	
insert_end(Start)	
1. Call create_node()	*//creating a new node pointed to by nptr*
2. If Start = NULL	*//checking for empty list*
Set Start = nptr	*//inserting new node as the first node*
Else	
Set temp = Start	
While temp->next != NULL	*//traversing up to the last node*
Set temp = temp->next	
End While	
Set temp->next = nptr	*//appending new node at the end*
End If	
3. End	

c) Insertion at a specified position: To insert a node at a position (say, pos) specified by the user, the list is traversed up to the pos-1 position. Then the next field of the new node is made to point to the node that is already at the pos position and the next field of the node at the pos-1 position is made to point to the new node. Figure 3.7 shows the insertion of the new node pointed to by nptr at the third position.

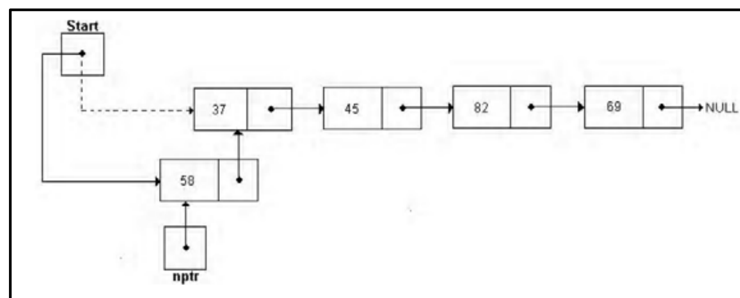


Figure 3.7 Insertion at a Specified Position in a Linked List

Algorithm 3.6 Insertion at a Specified Position

insert_pos(Start)
1. Call create_node()   //creating a new node pointed to by nptr
2. Set temp = Start
3. Read pos   //position at which the new node is to be inserted
4. Call count_node(temp)   //counting total number of nodes in count variable
5. If (pos > count + 1 OR pos = 0)
*   Print "Invalid position!" and go to step 7*
* End If*
6. If pos = 1
*   Set nptr->next = Start*
*   Set Start = nptr   //inserting new node as the first node*
* Else*
*   Set i = 1*
*   While i < pos - 1   //traversing up to the node at pos-1 position*
*   Set temp = temp->next*
*   Set i = i + 1*
* End While*
* Set nptr->next = temp->next   //inserting new node at pos position*
* Set temp->next = nptr*
* End If*
7. End

Deletion

Like insertion, nodes can be deleted from the linked list at any point of time and from any position. Whenever a node is deleted, the memory occupied by the node is de-allocated. Note that while performing deletions, we need to keep track of the node that is the immediate predecessor of the node to be deleted. Thus, two temporary pointer variables are used (except in case of deletion from the beginning) while traversing the list.

a) Deletion from the beginning: To delete a node from the beginning of a linked list, the address of the first node is stored in a temporary pointer variable temp and Start is modified to point to the second node in the linked list. After that, the memory occupied by the node pointed to by temp is de-allocated. Figure 3.8 shows the deletion of a node from the beginning of a linked list.

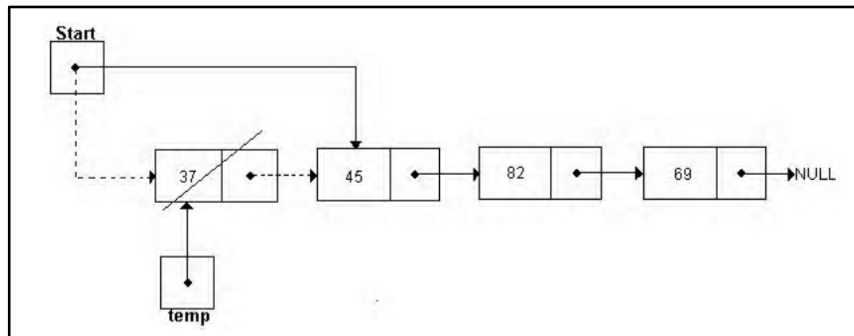


Figure 3.8 Deletion from the Beginning of a Linked List

Algorithm 3.7 Deletion from Beginning
delete_beg(Start) *1. If Start = NULL //checking for underflow* *Print "Underflow: List is empty!" and go to step 5* *End If* *2. Set temp = Start //temp pointing to the first node* *3. Set Start = Start->next //moving Start to point to the second node* *4. Deallocate temp //deallocating memory* *5. End*

b) Deletion from the end:_To delete a node from the end of a linked list, the list is traversed up to the last node. Two pointer variables save and the temp is used to traverse the list, where save points to the node previously pointed to by temp. At the end of traversing, temp points to the last node and save points to the second-last node. Then the next field of the node pointed to by save is made to point to NULL, and the memory occupied by the node pointed to by temp is de-allocated. Figure 3.9 shows the deletion of a node from the end of a linked list.

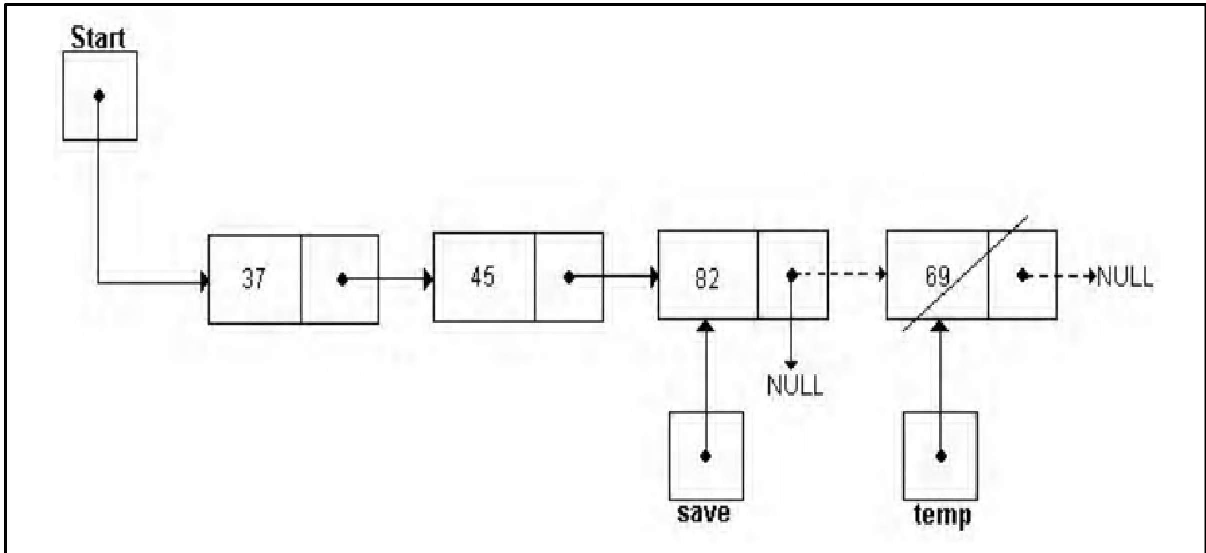


Figure 3.9 Deletion from the End of a Linked List

Algorithm 3.8 Deletion from the End

delete_end(Start)
1. If Start = NULL *//checking for underflow*
Print "Underflow: List is empty!" and go to step 6
End If
2. Set temp = Start *//temp pointing to the first node*
3. If temp->next = NULL *//deleting the only node of the list*
Set Start = NULL
Else
While (temp->next) != NULL *//traversing up to the last node*
Set save = temp *//save pointing to node previously*
//pointed to by temp
Set temp = temp->next *//moving temp to point to next node*
End While
End If
4. Set save->next = NULL *//making new last node to point to NULL*
5. Deallocate temp *//deallocating memory*
6. End

c) Deletion from a specified position: To delete a node from the position (say, pos) specified by the user, the list is traversed up to the pos position using pointer

variables temp and save. At the end of traversing, temp points to the node at the pos position and saves points to the node at the pos-1 position. Then the next field of the node pointed to by save is made to point to the node at pos+1 position, and the memory occupied by the node pointed to by temp is de-allocated. Figure 3.10 shows the deletion of a node in the third position.

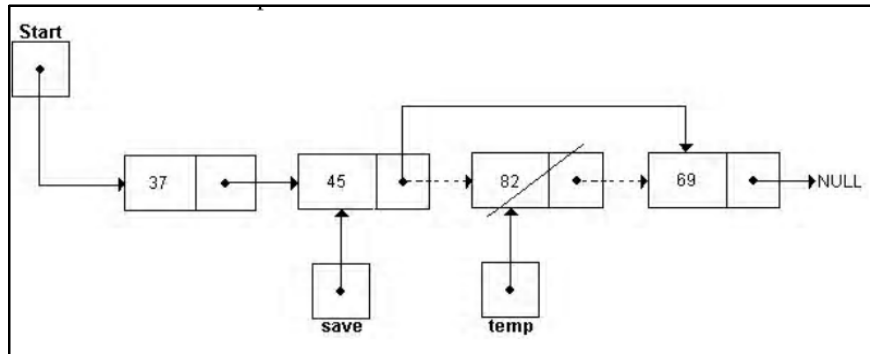


Figure 3.10 Deletion from a Specified Position in a Linked List

Algorithm 3.9 Deletion from a Specified Position

delete_pos(Start)
1. If Start = NULL //checking for underflow
Print "Underflow: List is empty!" and go to step 8
End If
2. Set temp = Start
3. Read pos //position of the node to be deleted
4. Call count_node(Start) //counting total number of nodes in the count variable
5. If pos > count OR pos = 0
Print "Invalid position!" and go to step 8
End If
6. If pos = 1
Set Start = temp->next //deleting the first node
Else
Set i = 1
While i < pos //traversing up to the node at position pos
Set save = temp
Set temp = temp->next
Set i = i + 1
End While
Set save->next = temp->next //deleting the node at position pos
End If
7. Deallocate temp //deallocating memory

8. End

Program 3.1: A program to illustrate the implementation of a singly-linked list.

```
#include<stdio.h>
#include<conio.h>
#define True 1
#define False 0
typedef struct node
{
        int info;
        struct node *next;
}Node;
/* Function prototypes */
Node * create_node();
int isempty(Node *);
void display(Node *);
int count_node(Node *);
void insert_beg(Node **);
void insert_end(Node **);
void insert_pos(Node **);
void delete_beg(Node **);
void delete_end(Node **);
void delete_pos(Node **);
void main()
{
        int item,ch,ch1;
        Node *Start=NULL;
        do
        {
                clrscr();
                printf("\n\n\tMain Menu");
                printf("\n1. Insert");
                printf("\n2. Delete");
                printf("\n3. Display");
                printf("\n4. Exit\n");
                printf("\nEnter your choice: ");
```

```
scanf("%d",&ch);
switch(ch)
{
case 1: printf("\n1. Insert in the beginning");
printf("\n2. Insert at the end");
printf("\n3. Insert at a specified
position");
printf("\n4. Back to main menu\n");
printf("\nEnter your choice: ");
scanf("%d",&ch1);
switch(ch1)
{
        case 1:    insert_beg(&Start);
                                break;
        case 2:    insert_end(&Start);
                                break;
        case 3:    insert_pos(&Start);
                                break;
        case 4: break;
        default: printf("\nInvalid choice!");
}
break;
case 2 : printf (" \ n1 . Delete from the beginning");
printf("\n2. Delete from the end");
printf("\n3. Delete from a specified
position");
printf("\n4.
Back to main menu\n");
printf("\nEnter your choice: ");
scanf("%d",&ch1);
switch(ch1)
{
        case 1:    delete_beg(&Start);
                                break;
        case 2:    delete_end(&Start);
                                break;
        case 3:    delete_pos(&Start);
```

```
                                        break;
                        case 4: break;
                        default: printf("\nInvalid
                        choice!");
                    }
                    break;
                    case 3: display(Start);
                         break;
                    case 4: exit();
                    default: printf("\nInvalid choice!");
                }
                getch();
            }while(1);
        }
        Node * create_node()
            {
                    Node *nptr;
                    int item;
                    nptr=(Node *)malloc(sizeof(Node));
                    if(nptr==NULL)
                    {
                        printf("\nOverflow: Memory not allocated!");
                        exit();
                    }
                    printf("\nEnter the value to be inserted: ");
                    scanf("%d",&item);
                    nptr->info=item;
                    nptr->next=NULL;
                    return(nptr);
            }
                    int isempty(Node *Start)
                    {
                        if(Start==NULL)
                            return True;
                        else
                            return False;
                    }
```

```
void display(Node *Start)
{
        Node *temp=Start;
        if(isempty(temp))
{
                printf("\nList is empty!!");
                return;
}
printf("\nThe linked list is: ");
while(temp != NULL)
{
                printf("%d ",temp->info);
                temp=temp->next;
}
}
int count_node(Node *Start)
{
        Node *temp=Start;
        int count=0;
        while(temp != NULL)
        {
                count++;
                temp=temp->next;
        }
        return(count);
}
void insert_beg(Node **Start)
{
        Node *nptr=create_node();
        nptr->next=*Start;
        *Start=nptr;
        printf("\nNode inserted.");
}
void insert_end(Node **Start)
{
        Node *temp=*Start;
        Node *nptr=create_node();
```

```
        if(isempty(temp))
            *Start=nptr;
    else
    {
        while(temp->next != NULL)
            temp=temp->next;
        temp->next=nptr;
    }
    printf("\nNode inserted.");
}
void insert_pos(Node **Start)
{
    int i,pos,count;
    Node *nptr=create_node();
    Node *temp=*Start;
    printf("\nEnter the position at which you want to insert:");
    scanf("%d",&pos);
    count=count_node(temp);
    if(pos>count+1 || pos==0)
    {
        printf("\nInvalid position!");
        return;
    }
    if(pos==1)
    {
        nptr->next=*Start;
        *Start=nptr;
    }
    else
    {
        for(i=1;i<pos-1;i++)
            temp=temp->next;
        nptr->next=temp->next;
        temp->next=nptr;
    }
    printf("\nNode inserted.");
    }
```

```
void delete_beg(Node **Start)
{
        Node *temp=*Start;
        if(isempty(temp))
{
                printf("\nUnderflow: List is empty!");
                return;
}
        *Start=temp->next;
        free(temp);
        printf("\nNode deleted.");
}
void delete_end(Node **Start)
{
        Node *temp=*Start;
        Node *save;
        if(isempty(temp))
{
        printf("\nUnderflow: List is empty!");
        return;
}
if(temp->next==NULL)
        *Start=NULL;
else
{
        while(temp->next != NULL)
        {
                save=temp;
                temp=temp->next;
        }
        save->next=NULL;
}
free(temp);
printf("\nNode deleted.");
}
void delete_pos(Node **Start)
{
```

```
        Node *temp=*Start,*save;
        int i,pos,count;
        if(isempty(temp))
        {
                printf("\nUnderflow: List is empty!");
                return;
        }
        printf("\nEnter the position of the node to be deleted:");
        scanf("%d",&pos);
        count=count_node(temp);
        if(pos>count || pos==0)
        {
                printf("\nInvalid position!");
                return;
        }
        if(pos==1)
                *Start=temp->next;
        else
        {
                for(i=1;i<pos;i++)
                {
                        save=temp;
                        temp=temp->next;
                }
                save->next=temp->next;
        }
        free(temp);
        printf("\nNode deleted.");
}
```

The output of the program is:

Main Menu

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 1

1. Insert in the beginning
2. Insert at the end
3. Insert at a specified position
4. Back to main menu

Enter your choice: 1

Enter the value to be inserted: 1

Node inserted.

Main Menu

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 1

1. Insert in the beginning
2. Insert at the end
3. Insert at a specified position
4. Back to main menu

Enter your choice: 2

Enter the value to be inserted: 3

Node inserted

Main Menu

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 1

1. Insert in the beginning
2. Insert at the end
3. Insert at a specified position
4. Back to main menu

Enter your choice: 3

Enter the value to be inserted: 2

Enter the position at which you want to insert: 2

Node inserted.

Main Menu

1. Insert

2. Delete

3. Display

4. Exit

Enter your choice: 3

The linked list is: 1 2 3

Main Menu

1. Insert

2. Delete

3. Display

4. Exit

Enter your choice: 2

1. Delete from the beginning

2. Delete from the end

3. Delete from a specified position

4. Back to main menu

Enter your choice: 1

Node deleted.

Main Menu

1. Insert

2. Delete

3. Display

4. Exit

Enter your choice: 3

The linked list is: 2 3

Main Menu

1. Insert

2. Delete

3. Display

4. Exit

Enter your choice: 2

1. Delete from the beginning
2. Delete from the end
3. Delete from a specified position
4. Back to main menu

Enter your choice: 3

Enter the position of the node to be deleted: 2

Node deleted.

Main Menu

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 2

1. Delete from the beginning
2. Delete from the end
3. Delete from a specified position
4. Back to main menu

Enter your choice: 2

Node deleted.

Main Menu

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 3

List is empty!!

Main Menu

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 4

Searching

Searching a value (say, item) in a linked list means finding the position of the node, which stores the item as its value. If an item is found in the list, the search is successful and the position of that node is displayed. However, if an item is not found till the end of the list, then the search is unsuccessful and an appropriate message is displayed. Note that the linked list may be in sorted or unsorted order. Therefore, we are discussing two searching algorithms: one for sorted and another for an unsorted linked list. Only linear search can be performed on linked lists.

a) Searching in an unsorted list: If the data in the linked list are not arranged in a specific order, the list is traversed completely starting from the first node towards the last node and the value of each node (that is, node–>info) is compared with the value to be searched.

b) Searching in a sorted list: The process of searching an item into a sorted (ascending order) linked list is similar to that of an unsorted linked list. However, while comparing, once the value of any node exceeds the item (the value to be searched), the search is stopped immediately. In that case, the list is not required to be traversed completely.

Algorithm 3.10 Searching in an Unsorted List

```
search_unsort(Start)
1. If Start = NULL
       Print "List is empty!!" and go to step 7
   End If
2. Set ptr = Start                    //ptr pointing to the first node
3. Set pos = 1
4. Read item                          //item is the value to be searched
5. While ptr != NULL                  //traversing up to the last node
    If item = ptr->info
           Print "Value found at position", pos and go to step 7
    Else
          Set ptr = ptr->next         //moving ptr to point to next node
          Set pos = pos + 1
    End If
    End While
6. Print "Value not found"            //search unsuccessful
```

7. End

Algorithm 3.11 Searching in a Sorted List

```
search_sort(Start)
1. If Start = NULL
     Print "List is empty!!" and go to step 7
   End If
2. Set ptr = Start                  //ptr pointing to the first node
3. Set pos = 1
4. Read item
5. While ptr->next != NULL         //traversing up to the last node
   If item < ptr->info             //comparing item with the value of current node
        Print "Value not found" and go to step 7
   Else If item = ptr->info
        Print "Value found at position", pos and go to step 7
   Else
        Set ptr = ptr->next        //moving ptr to point to next node
   Set pos = pos + 1
   End If
   End While
6. Print "Value not found"         //search unsuccessful
7. End
```

Reversing

To reverse a singly-linked list, the list is traversed up to the last node, and the links of the nodes are reversed such that the first node of the list becomes the last node and the last node becomes the first node. For this, three-pointer variables (say, save, ptr, and temp) are used. Initially, temp points to Start, and both ptr and save point to NULL. While traversing the list, temp points to the current node, ptr points to the node previously pointed to by temp, and save points to the node previously pointed to by ptr. The links between nodes are reversed by making the next field of the node pointed to by ptr to point to the node pointed to by save. At the end of traversing, temp points to NULL, ptr points to the last node, and save points to the second last node of the list. Then Start is made to point to the node pointed to by ptr in order to make the last node the first node of the list. Figure 3.11 (a–f) shows the process of reversing a linked list.

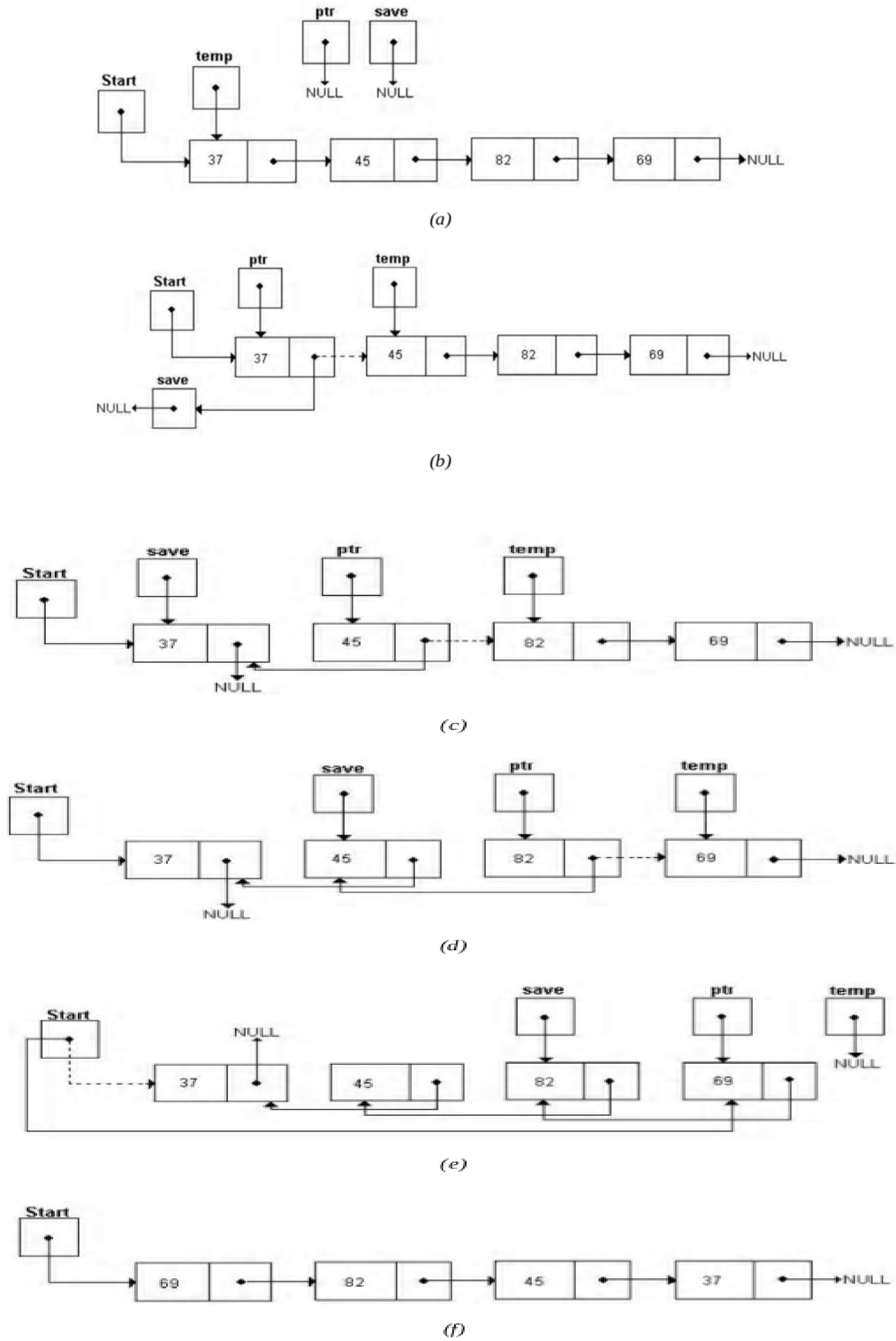


Figure 3.11 Reverse of a Linked List

3.3 Circular- Linked Lists

A linear linked list in which the next field of the last node points back to the first node instead of containing NULL is termed as a circular linked list. The main advantage of a circular linked list over a linear linked list is that in the former by starting with any node in the list, we can reach any of its predecessor nodes. This is because when we traverse a circular linked list starting with a particular node, we come back to the same node at the end. Figure 3.12 shows an example of a circular linked list.

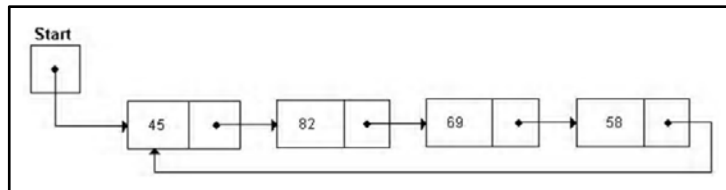


Figure 3.12 An example of a Circular Linked List

All the operations that can be performed on linear linked lists can easily be performed on circular linked lists but with some modifications. Some of these operations are discussed as follows.

Note: The process of creating a node of a circular linked list is the same as that of a linear linked list.

3.3.1 Traversing

We can traverse a circular linked list in the same way as a linear linked list except for the condition for checking the end of the list. Here, the list is traversed until we reach a node in the list that contains the address of the first node in its next field rather than NULL as in the case of a linear linked list.

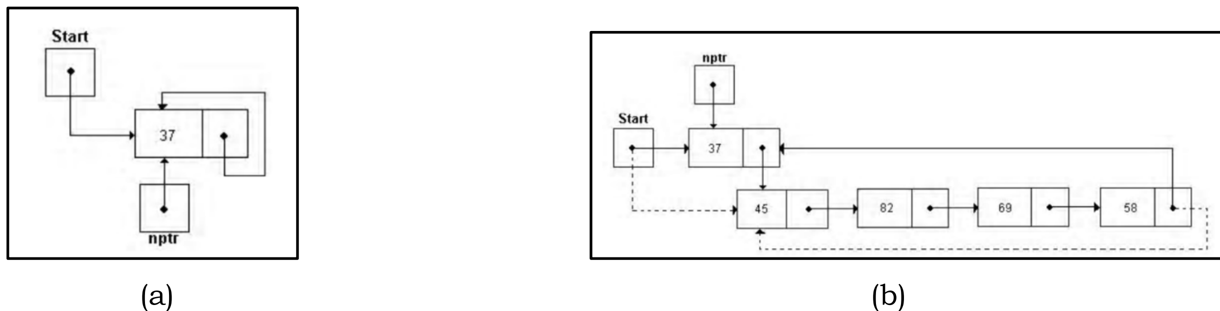
Algorithm 3.13 Traversing a Circular Linked List

display(Start)
1. If Start = NULL
    Print "List is empty!!" and go to step 4
  End If
2. Set temp = Start   *//initialising temp with Start*
3. Do

```
        Print temp->info              //displaying value of each node
        Set temp = temp->next
    While temp != Start
4. End
```

3.3.2 Insertion

Like linear linked lists, nodes can be inserted at any position in a circular linked list. To insert a new node (pointed to by nptr) at the beginning of a circular linked list [Figure 3.13(b)], the next field of the new node is made to point to the existing first node and the Start pointer is modified to point to the new node. Now, since the first node of the list is changed, the next field of the last node also needs to be modified to point to the new node. However, if initially, the list is empty, a new node is inserted as the first node and its next field is made to point to itself [Figure 3.13(a)].



(a) (b)

Figure 3.13 Insertion in the beginning of a circular linked list

Algorithm 3.14 Insertion in the Beginning

```
insert_beg(Start)
1. Call create_node()                //creating a new node pointed to by nptr
2. If Start = NULL                   //checking for empty list
        Set Start = nptr             //inserting new node as the
   first node
        Set Start->next = Start
        Else
        Set temp = Start
        While temp->next != Start    //traversing up to the last node
            Set temp = temp->next
        End While
        Set nptr->next = Start       //inserting new node in the beginning
        Set Start = nptr             //Start pointing to new node
```

Set temp->next = Start //next field of last node pointing to new node *End If* *3. End*

While inserting a new node (pointed to by nptr) at the end of a circular linked list, the list is traversed up to the last node. The next field of the last node is made to point to the new node and the next field of the new node is made to point to Start. However, if the circular linked list is empty, a new node becomes the first node, and Start points to it. In addition, the next field of the new node points to itself as it is the single node in the list. Figure 3.14 shows the insertion of a new node pointed to by nptr at the end of a circular linked list.

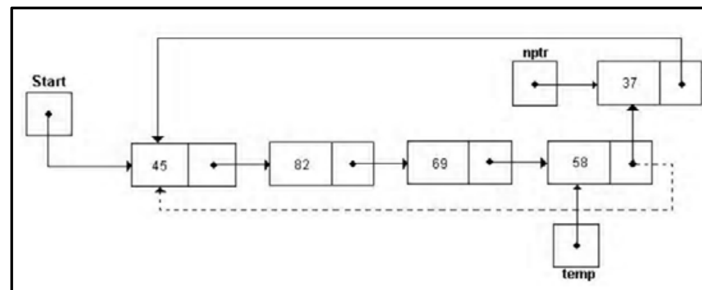


Figure 3.14 Insertion at the end of a circular linked list

Algorithm 3.15 Insertion at the End

```
insert_end(Start)
1. Call create_node()                  //creating a new node pointed to by nptr
2. If Start = NULL                      //checking for empty list
        Set Start = nptr                //inserting new node in the empty linked list
        Set Start->next = Start         //next field of first node pointing to itself
        Else
        Set temp = Start
        While temp->next != Start       //traversing up to the last node
             Set temp = temp->next
        End While
        Set temp->next = nptr           //next field of last node pointing to new node
        Set nptr->next = Start          //next field of new node pointing to Start
   End If
3. End
```

3.3.3 Deletion

To delete a node from the beginning of a circular linked list, Start is modified to point to the second node, and the next field of the last node is made to point to the new first node. For this, two-pointer variables temp and ptr are used. The pointer temp stores the address of the node to be deleted (that is, the address of the first node), and Start is modified to point to the second node. The pointer ptr is used for traversing the list and at the end of traversing, it stores the address of the last node. Then the next field of the last node is made to point to the new first node. Also, the memory occupied by the node pointed to by temp is de-allocated. Figure 3.15 shows the deletion of a node from the beginning of a circular linked list.

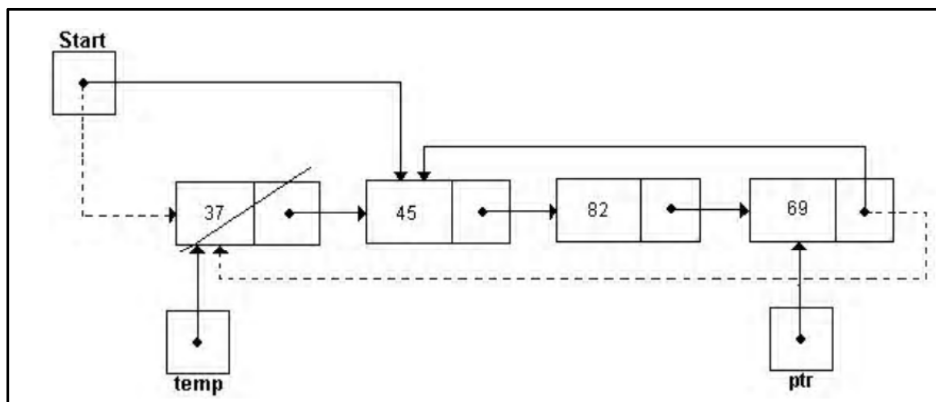


Figure 3.15 Deletion from the beginning of a circular linked list

Algorithm 3.16 Deletion from Beginning

delete_beg(Start)
1. If Start = NULL
Print "Underflow: List is empty!" and go to step 8
End If
2. Set temp = Start
3. Set ptr = temp
4. While ptr->next != Start //traversing up to the last node
Set ptr = ptr->next
End While
5. Set Start = Start->next //Start pointing to the next node
6. Set ptr->next = Start //last node pointing to new first node
7. Deallocate temp //deallocating memory
8. End

To delete a node from the end of a circular linked list, two-pointer variables save and the temp is used. The pointer variable temp is used to traverse the list and save points to the node previously pointed to by temp. At the end of traversing, temp points to the last node and save points to the second-last node. Then the next field of save is made to point to Start, and the memory occupied by the last node (that is, pointed to by temp) is de-allocated. Figure 3.16 shows the deletion of a node from the end of a circular linked list.

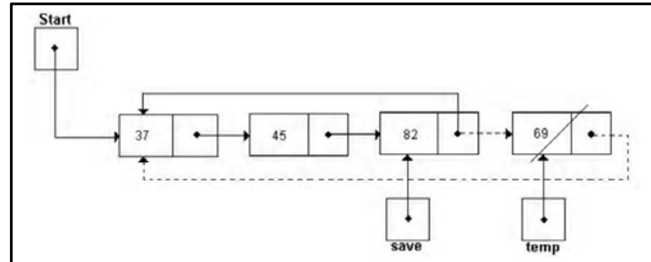


Figure 3.16 Deletion from the end of a circular linked list

Algorithm 3.17 Deletion from the End

delete_end(Start)
1. If Start = NULL //checking for underflow
Print "Underflow: List is empty!" and go to step 5
End If
2. Set temp = Start
3. If temp->next = Start //deleting the only node of the list
Set Start = NULL
Else
While temp->next != Start//traversing up to the last node
Set save = temp
Set temp = temp->next
End While
Set save->next = Start //second last node becomes the last node
End If
4. Deallocate temp //deallocating memory
5. End

Note: The process of deleting a node from a specified position in a circular linked list is the same as that of a singly-linked list.

Program 3.2: A program to illustrate the implementation of a circular linked list.

```
#include<stdio.h>
#include<conio.h>
#define True 1
#define False 0
typedef struct node
{
	int info;
	struct node *next;
}Node;
/* Function prototypes */
Node * create_node();
int isempty(Node *);
void display(Node *);
void insert_beg(Node **);
void insert_end(Node **);
void delete_beg(Node **);
void delete_end(Node **);
void main()
{
	int item,ch,ch1;
	Node *Start=NULL;
	do
	{
		clrscr();
		printf("\n\n\tMain Menu");
		printf("\n1. Insert");
		printf("\n2. Delete");
		printf("\n3. Display");
		printf("\n4. Exit\n");
		printf("\nEnter your choice: ");
		scanf("%d",&ch);
		switch(ch)
		{
			case 1: printf("\n1. Insert in the beginning");
			printf("\n2. Insert at the end");
			printf("\n3. Back to main menu\n");
```

```
            printf("\nEnter your choice: ");
            scanf("%d",&ch1);
            switch(ch1)
            {
                        case 1: insert_beg(&Start);
                                                            break;
                        case 2: insert_end(&Start);
                                                            break;
                        case 3: break;
                        default: printf("\nInvalid choice!");
            }
break;
            case 2: printf("\n1. Delete from the beginning");
            printf("\n2. Delete from the end");
            printf("\n3. Back to main menu\n");
            printf("\nEnter your choice: ");
            scanf("%d",&ch1);
            switch(ch1)
            {
                        case 1: delete_beg(&Start);
                                                            break;
                        case 2: delete_end(&Start);
                                                            break;
                        case 3: break;
                        default: printf("\nInvalid choice!");
                        }
                        break;
            case 3: display(Start);
            break;
            case 4: exit();
            default: printf("\nInvalid choice!");
            }
      getch();
      }while(1);
}
Node * create_node()
{
```

```
        Node *nptr;
        int item;
        nptr=(Node *)malloc(sizeof(Node));
        if(nptr==NULL)
        {
                printf("\nOverflow: Memory not allocated!");
                exit();
        }
        printf("\nEnter the value to be inserted: ");
        scanf("%d",&item);
        nptr->info=item;
        nptr->next=NULL;
        return(nptr);
}
int isempty(Node *Start)
{
        if(Start==NULL)
                return True;
        else
                return False;
}
void display(Node *Start)
{
        Node *temp=Start;
        if(isempty(temp))
                printf("\nList is empty!!");
        else
        {
                printf("\nThe linked list is: ");
                do
                {
                        printf("%d ",temp->info);
                        temp=temp->next;
                }while(temp != Start);
        }
}
void insert_beg(Node **Start)
```

```
{
        Node *nptr=create_node();
        Node *temp=*Start;
        if(isempty(temp))
        {
                *Start=nptr;
                (*Start)->next=*Start;
        }
        else
                {
                        while(temp->next != *Start)
                                temp=temp->next;
                        nptr->next=*Start;
                        *Start=nptr;
                        temp->next=*Start;
                }
                printf("\nNode inserted.");
        }
        void insert_end(Node **Start)
        {
                Node *temp=*Start;
                Node *nptr=create_node();
                if(isempty(temp))
                {
                        *Start=nptr;
                        (*Start)->next=*Start;
                }
                else
                {
                        while(temp->next != *Start)
                                temp=temp->next;
                        temp->next=nptr;
                        nptr->next=*Start;
                }
                printf("\nNode inserted.");
        }
        void delete_beg(Node **Start)
```

```
{
        Node *temp=*Start;
        Node *ptr=temp;
        if(isempty(temp))
        {
                printf("\nUnderflow: List is empty!");
                return;
        }
        while(ptr->next != *Start)
                ptr=ptr->next;
        *Start=(*Start)->next;
        ptr->next=*Start;
        free(temp);
        printf("\nNode deleted.");
}
void delete_end(Node **Start)
{
        Node *temp=*Start;
        Node *save;
        if(isempty(temp))
        {
                printf("\nUnderflow: List is empty!");
                return;
        }
        if(temp->next==*Start)
                *Start=NULL;
        else
        {
                while(temp->next != *Start)
                {
                        save=temp;
                        temp=temp->next;
                }
                save->next=*Start;
        }
        free(temp);
        printf("\nNode deleted.");
```

}

The output of the program is:

Main Menu
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1
1. Insert in the beginning
2. Insert at the end
3. Back to main menu
Enter your choice: 1
Enter the value to be inserted: 5
Node inserted.
Main Menu
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1
1. Insert in the beginning
2. Insert at the end
3. Back to main menu
Enter your choice: 1
Enter the value to be inserted: 4
Node inserted.
Main Menu
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1

1. Insert in the beginning
2. Insert at the end
3. Back to main menu

Enter your choice: 2

Enter the value to be inserted: 6

Node inserted.

Main Menu

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 3

The linked list is: 4 5 6

Main Menu

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 2

1. Delete from the beginning
2. Delete from the end
3. Back to main menu

Enter your choice: 1

Node deleted.

Main Menu

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 3

The linked list is: 5 6

Main Menu

1. Insert

2. Delete
3. Display
4. Exit
Enter your choice: 2
1. Delete from the beginning
2. Delete from the end
3. Back to main menu
Enter your choice: 2
Node deleted.
Main Menu
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3
The linked list is: 5
Main Menu
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 4

3.4 Doubly- Linked Lists

In a singly-linked list, each node contains a pointer to the next node and it has no information about its previous node. Thus, we can traverse only in one direction, that is, from beginning to end. However, sometimes it is required to traverse in the backward direction, that is, from end to the beginning. This can be implemented by maintaining an additional pointer in each node of the list that points to the previous node. Such a type of linked list is called the doubly-linked list.

Each node of a doubly-linked list consists of three fields: prev, info, and next (Figure 3.17). The info field contains the data, the prev field contains the address of the previous node and the next field contains the address of the next node.

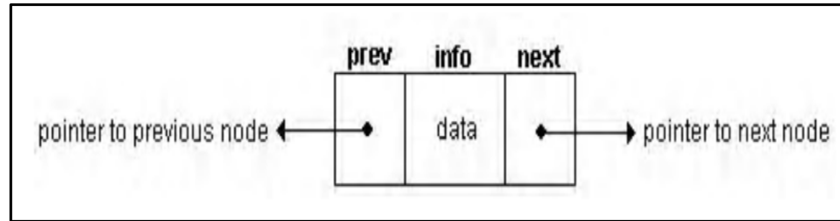


Figure 3.17 Node of a Doubly-linked List

Since a doubly-linked list allows traversing in both forward and backward directions, it is also referred to as a two-way list. Figure 3.18 shows an example of a doubly-linked list having four nodes. Note that the prev field of the first node and the next field of the last node in a doubly-linked list points to NULL.

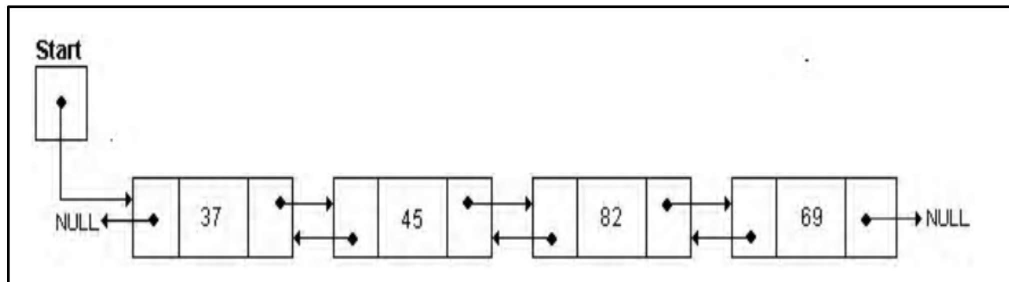


Figure 3.18 An Example of a Doubly-linked List with Four Nodes

To define the node of a doubly-linked list in the 'C' language, the structure used to represent the node of the singly-linked list is extended to have an extra pointer, which points to the previous node.

The structure of a node of a doubly-linked list is shown here.

```
typedef struct node
{
        int info;                               /*to store integer type data*/
        struct node *next;              /*to store a pointer to next node*/
        struct node *prev;              /*to store a pointer to previous node*/
}Node;
        Node *nptr;                     /*nptr is a pointer to node*/
```

When memory is allocated successfully to a node, that is, there is no overflow, the node is initialized. The info field is initialized with a valid value and the prev and next

fields are initialized with NULL.

Algorithm 3.18 Creating a Node of Doubly Linked List
create_node() *1. Allocate memory for nptr //nptr is a pointer to a new node* *2. If nptr = NULL* *Print "Overflow: Memory not allocated!" and go to step 8* *3. Read item //item is the value stored in the node* *4. Set nptr->info = item* *5. Set nptr->next = NULL* *6. Set nptr->prev = NULL* *7. Return nptr* *8. End*

Note that all the operations that are performed on singly-linked lists can also be performed on doubly-linked lists. In this section, we will discuss only insertion and deletion operations on doubly-linked lists.

3.4.1 Insertion

To insert a new node at the beginning of a doubly-linked list, a pointer (say, nptr) to a new node is created. The next field of the new node is made to point to the existing first node and the prev field of the existing first node (that has become the second node now) is made to point to the new node. After that, Start is modified to point to a new node. Figure 3.19 shows the insertion of a node at the beginning of a doubly-linked list.

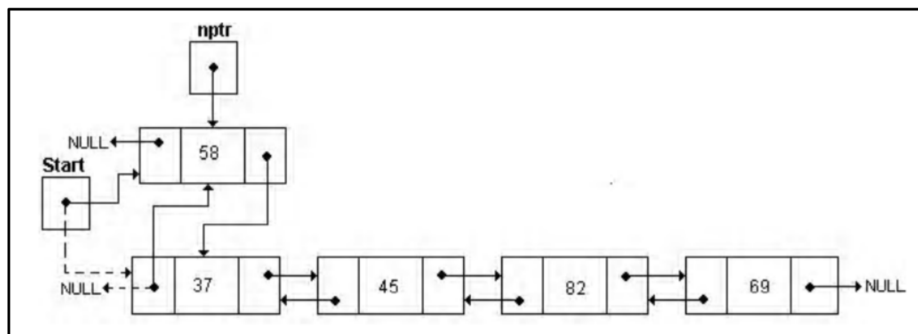


Figure 3.19 Insertion at the beginning of a doubly-linked list

Algorithm 3.19 Insertion in the Beginning
insert_beg(Start)
1. Call create_node()   //creating a new node pointed to by nptr
2. If Start != NULL
Set nptr->next = Start   //inserting node in the beginning
Set Start->prev = nptr
End If
3. Set Start = nptr   //making Start point to the new node
4. End

To insert a new node at the end of a doubly-linked list, the list is traversed up to the last node using some pointer variable (say, temp). At the end of traversing, temp points to the last node. Then, the next field of the last node (pointed to by temp) is made to point to the new node and the prev field of the new node is made to point to the node pointed to by temp. However, if the list is empty, the new node is inserted as the first node in the list.

Figure 3.20 shows the insertion of a new node at the end of a doubly-linked list.

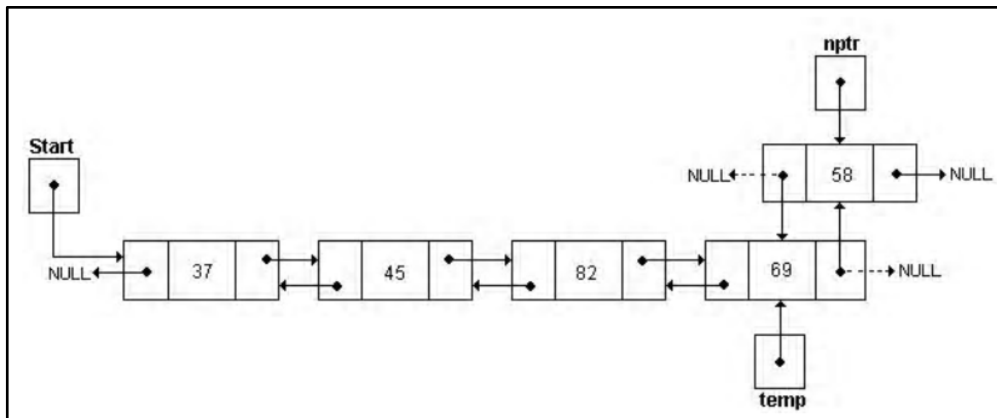


Figure 3.20 Insertion at the end of a doubly linked list

Algorithm 3.20 Insertion at the end
insert_end(Start)
1. Call create_node()   //creating a new node pointed to by nptr
2. If Start = NULL
Set Start = nptr   //inserting new node as the first node
Else
Set temp = Start   //pointer temp used for traversing
While temp->next != NULL
Set temp = temp->next

End While
Set temp->next = nptr
Set nptr->prev = temp
End If
3. End

To insert a new node (pointed to by nptr) at a specified position (say, pos) in a doubly-linked list, the list is traversed up to the pos-1 position. At the end of traversing, temp points to the node at the pos-1 position. For simplicity, we use another pointer variable (say, ptr) to point to the node that is already at the pos position. Then, the prev field of the node pointed to by ptr is made to point to the new node and the next field of the new node is made to point to the node pointed to by ptr. Also, the prev field of the new node is made to point to the node pointed to by temp and the next field of the node pointed to by temp is made to point to the new node.

Figure 3.21 shows the insertion of a new node at the third position in a doubly-linked list.

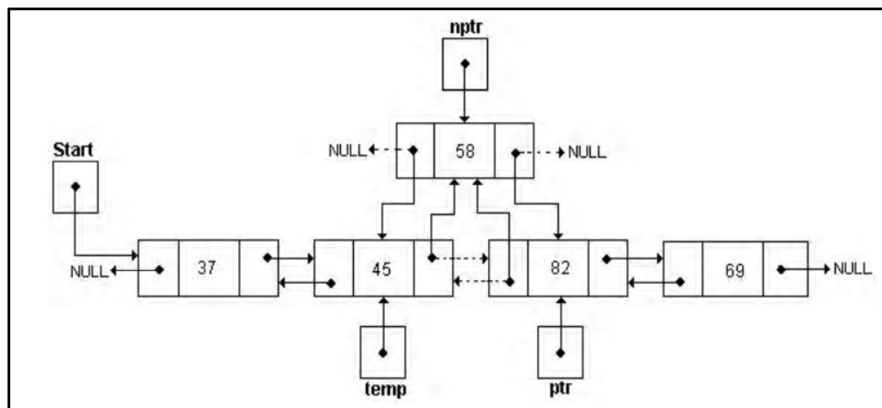


Figure 3.21 Insertion at a specified position of a doubly linked list

Algorithm 3.21 Insertion at a Specified Position
insert_pos (Start)
1. Call create_node() *//creating a new node pointed to by nptr*
2. Set temp = Start
3. Read pos
4. Call count_node(temp) *//counting number of nodes in count variable*
5. If pos = 0 OR pos > count + 1
Print "Invalid position!" and go to step 7

```
   End If
6. If pos = 1
            Set nptr->next = Start      //inserting node at the beginning
        Set Start = nptr                //Start pointing to new node
  Else
        Set i = 1
        While i < pos-1                 //traversing up to the node at pos-1 position
              Set temp = temp->next
              Set i = i + 1
        End While
        Set ptr = temp->next
        Set ptr->prev = nptr
        Set nptr->next = ptr
        Set nptr->prev = temp
        Set temp->next = nptr
   End If
7. End
```

3.4.2 Deletion

To delete a node from the beginning of a doubly-linked list, a pointer variable (say, temp) is used to point to the first node. Then Start is modified to point to the next node and the prev field of this node is made to point to NULL. After that, the memory occupied by the node pointed to by temp is de-allocated. Figure 3.22 shows the deletion of a node from the beginning of a doubly-linked list.

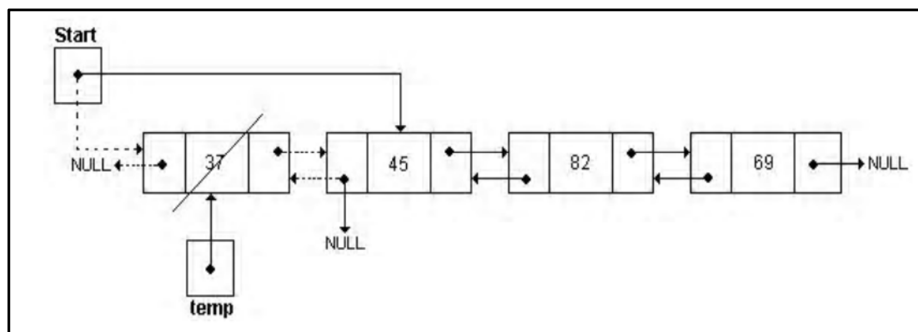


Figure 3.22 Deletion from the beginning of a doubly-linked list

Algorithm 3.22 Deletion from the beginning
delete_beg(Start)
1. If Start = NULL

Print "Underflow: List is empty!" and go to step 6 *End If* *2. Set temp = Start //temp points to the node to be deleted* *3. Set Start = Start->next //making Start to point to next node* *4. Set Start->prev = NULL* *5. Deallocate temp //de-allocating memory* *6. End*

Note: The process of deleting a node from the end of a doubly-linked list is the same as that of the singly-linked list.

To delete a node from a position (say, pos) specified by the user, the list is traversed up to the pos position using pointer variables temp and save. At the end of traversing, temp points to the node at the pos position and save points to the node at the pos-1 position. Here, for simplicity, we use another pointer variable ptr to point to the node at the pos+1 position. Then, the next field of the node at pos-1 position (pointed to by save) is made to point to the node at pos+1 position (pointed to by ptr). In addition, the prev field of the node at pos+1 position (pointed to by ptr) is made to point to the node at pos-1 position (pointed to by save). After that, the memory occupied by the node pointed to by temp is de-allocated. Figure 3.23 shows the deletion of a node at the third position from a doubly-linked list.

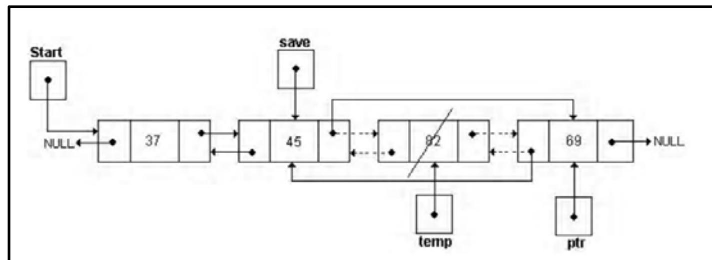


Figure 3.23 Deletion from a Specified Position of a doubly-linked list

Algorithm 3.23 Deletion from a Specified Position

delete_pos(Start)
1. If Start = NULL
Print "Underflow: List is empty!" and go to step 8
End If
2. Set temp = Start
3. Read pos
4. Call count_node(temp) //counting total number of nodes in count variable

```
5. If pos > count OR pos = 0
      Print "Invalid position!" and go to step 6
   End If
6. If pos = 1
      Set Start = Start->next                //deleting the first node
      Start->prev = NULL
   Else
      Set i = 1
      While i < pos                          //traversing up to the node at pos position
            Set save = temp                  //save pointing to the node at pos-1 position
            Set temp = temp->next            //making temp to point to next node
            Set i = i + 1
      End While
      Set ptr = temp->next
      Set save->next = ptr
      Set ptr->prev = save
   End If
7. Deallocate temp                           //deallocating memory
8. End
```

Note: A doubly-linked list in which the next field of the last node points to the first node instead of NULL is termed as a doubly-circular linked list.

Program 4.4: A program to illustrate the implementation of a doubly-linked list.

```
#include<stdio.h>
#include<conio.h>
#define True 1
#define False 0
typedef struct node
{
        int info;
        struct node *next;
        struct node *prev;
}Node;                                             /* node of a doubly linked list */
/* Function prototypes */
Node * create_node();
int isempty(Node *);
void display(Node *);
```

```
int count_node(Node *);
void insert_beg(Node **);
void insert_end(Node **);
void insert_pos(Node **);
void delete_beg(Node **);
void delete_end(Node **);
void delete_pos(Node **);
/*Main Function*/
void main()
{
        int item,ch,ch1;
        Node *Start=NULL;
        do
        {
                clrscr();
                printf("\n\n\tMain Menu");
                printf("\n1. Insert");
                printf("\n2. Delete");
                printf("\n3. Display");
                printf("\n4. Exit\n");
                printf("\nEnter your choice: ");
                scanf("%d",&ch);
                switch(ch)
                {
                        case 1: printf("\n1. Insert in the beginning");
                        printf("\n2. Insert at the end");
                        printf("\n3. Insert at a specified position");
                        printf("\n4. Back to main menu\n");
                        printf("\nEnter your choice:");
                        scanf("%d",&ch1);
                        switch(ch1)
                        {
                                case 1:insert_beg(&Start);
                                            break;
                                case 2:insert_end(&Start);
                                            break;
                                case 3:insert_pos(&Start);
```

```
                                    break;
                    case 4:break;
                        default:printf("\nInvalid choice!");
                    }
                    break;
                case 2 : printf ("\n1. Delete from the beginning");
                printf("\n2. Delete from the end");
                printf("\n3. Delete from a specified position");
                printf("\n4. Back to main menu\n");
                printf("\nEnter your choice: ");
                scanf("%d",&ch1);
                switch(ch1)
                {
                    case 1:delete_beg(&Start);
                                    break;
                    case 2:delete_end(&Start);
                                    break;
                    case 3:delete_pos(&Start);
                                    break;
                    case 4:break;
                        default: printf("\nInvalid choice!");
                }
                break;
                case 3:display(Start);
                break;
                case 4: exit();
                default: printf("\nInvalid choice!");
                }
                getch();
            }while(1);
}
Node * create_node()
{
            Node *nptr;
            int item;
            nptr=(Node *)malloc(sizeof(Node));
            if(nptr==NULL)
```

```
        {
            printf("\nOverflow: Memory not allocated!");
            exit();
        }
        printf("\nEnter the value to be inserted: ");
        scanf("%d",&item);
        nptr->info=item;
        nptr->next=NULL;
        nptr->prev=NULL;
        return(nptr);
    }
    int isempty(Node *Start)
    {
        if(Start==NULL)
            return True;
        else
            return False;
    }
void display(Node *Start)
{
    Node *temp=Start;
    if(temp==NULL)
        printf("\nList is empty!!");
    else
    {
        printf("\nThe linked list is: ");
        while(temp != NULL)
        {
            printf("%d ",temp->info);
            temp=temp->next;
        }
    }
}
int count_node(Node *Start)
{
    Node *temp=Start;
    int count=0;
```

```
        while(temp != NULL)
        {
                count++;
                temp=temp->next;
        }
        return(count);
}
void insert_beg(Node **Start)
{
        Node *nptr=create_node();
        if (*Start != NULL)
        {
                nptr->next=*Start;
                (*Start)->prev=nptr;
        }
        *Start=nptr;
        printf("\nNode inserted.");
}
void insert_end(Node **Start)
{
        Node *temp;
        Node *nptr=create_node();
        if(*Start==NULL)
                *Start=nptr;
        else
        {
                temp=*Start;
                while(temp->next != NULL)
                temp=temp->next;
                temp->next=nptr;
                nptr->prev=temp;
        }
        printf("\nNode inserted.");
}
void insert_pos(Node **Start)
{
        int i,pos,count;
```

```
        Node *nptr=create_node();
        Node *temp=*Start,*ptr;
        printf("\nEnter the position at which you want to insert:");
        scanf("%d",&pos);
        count=count_node(temp);
        if(pos==0 || pos>count+1)
        {
                printf("\nInvalid position!");
                return;
        }
        if(pos==1)
        {
                nptr->next=*Start;
                *Start=nptr;
        }
        else
        {
                for(i=1;i<pos-1;i++)
                temp=temp->next;
                ptr=temp->next;
                ptr->prev=nptr;
                nptr->next=ptr;
                nptr->prev=temp;
                temp->next=nptr;
        }
        printf("\nNode inserted.");
}
void delete_beg(Node **Start)
{
        Node *temp=*Start;
        *Start=(*Start)->next;
        (*Start)->prev=NULL;
        free(temp);
        printf("\nNode deleted.");
}
void delete_end(Node **Start)
{
```

```
	Node *temp=*Start;
	Node *save;
	if(isempty(temp))
	{
	printf("\nUnderflow: List is empty!");
	return;
	}
	if(temp->next==NULL)
		*Start=NULL;
	else
	{
		while(temp->next != NULL)
	{
		save=temp;
		temp=temp->next;
	}
		save->next=NULL;
	}
	free(temp);
	printf("\nNode deleted.");
}
void delete_pos(Node **Start)
{
	Node *temp=*Start,*save,*ptr;
	int i,pos,count;
	printf("\nEnter the position of the node to be deleted:");
	scanf("%d",&pos);
	count=count_node(temp);
	if(pos>count)
	{
	printf("\nInvalid position!\n");
	return;
	}
	if(pos==1)
	{
	*Start=temp->next;
	(*Start)->prev=NULL;
```

```
}
else
{
for(i=1;i<pos;i++)
{
save=temp;
temp=temp->next;
}
ptr=temp->next;
save->next=ptr;
ptr->prev=save;
}
free(temp);
printf("\nNode deleted.\n");
}
```

The output of the program is:

Main Menu

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 1

1. Insert in the beginning
2. Insert at the end
3. Insert at a specified position
4. Back to main menu

Enter your choice: 1

Enter the value to be inserted: 6

Node inserted.

Main Menu

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 1

1. Insert in the beginning
2. Insert at the end
3. Insert at a specified position
4. Back to main menu

Enter your choice: 2

Enter the value to be inserted: 5

Node inserted.

Main Menu

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 1

1. Insert in the beginning
2. Insert at the end
3. Insert at a specified position
4. Back to main menu

Enter your choice: 3

Enter the value to be inserted: 8

Enter the position at which you want to insert: 2

Node inserted.

Main Menu

1. Insert
2. Delete
3. Display
4. Exit

Enter your choice: 3

The linked list is: 6 8 5

Main Menu

1. Insert
2. Delete
3. Display

4. Exit

Enter your choice: 2

1. Delete from the beginning

2. Delete from the end

3. Delete from a specified position

4. Back to main menu

Enter your choice: 3

Enter the position of the node to be deleted: 4

Invalid position!

Main Menu

1. Insert

2. Delete

3. Display

4. Exit

Enter your choice: 2

1. Delete from the beginning

2. Delete from the end

3. Delete from a specified position

4. Back to main menu

Enter your choice: 1

Node deleted.

Main Menu

1. Insert

2. Delete

3. Display

4. Exit

Enter your choice: 3

The linked list is: 8 5

Main Menu

1. Insert

2. Delete

3. Display

4. Exit

Enter your choice: 2
1. Delete from the beginning
2. Delete from the end
3. Delete from a specified position
4. Back to main menu
Enter your choice: 2
Node deleted.
Main Menu
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3
The linked list is: 8
Main Menu
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 4

3.5 Dynamic Storage Management: Application of a Doubly-Linked List

As we know in a multiprogramming environment, multiple programs reside in the main memory at one time in order to efficiently utilize the memory. Whenever a program requests a memory block of some specific size, the memory manager allocates memory to it, if available. Once a program completes its execution, it releases the memory allocated to it so that other programs may use it. In a dynamic storage management scheme, the memory requirements of the programs are not known in advance. In addition, the order in which the memory is de-allocated by the programs may be different from that of memory allocation.

Initially, when there are no jobs in the memory, the whole memory is available for allocation and is considered as a single large block of available memory (a hole). As programs enter the system and request variable-size blocks, the memory manager

allocates them the memory blocks that are large enough to accommodate the programs. As soon as any program terminates, the memory occupied by it is de-allocated. Thus, at a given point of time, some blocks of memory may be in use while others may be free [Figure 3.24 (a)]. Now to make further allocations, the memory manager must keep track of the free space in memory. For this, the memory manager maintains a free-storage list that keeps track of the unused blocks (holes of variable sizes) of memory. The free-storage list is implemented as a linked list where each node contains the size of the block and the address of the next available block [Figure 3.24 (b)].

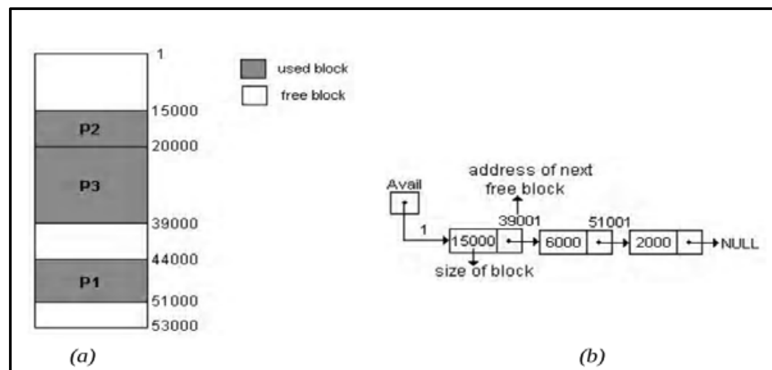


Figure 3.24 Memory Status and Free- storage List

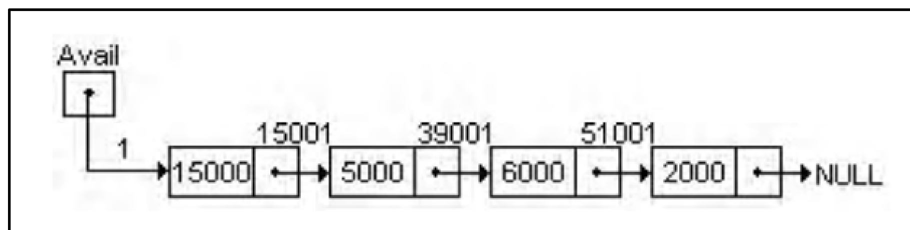
If any program requests for a block of size n, the memory manager may adopt one of the following strategies to select a hole from the free-storage list.

- **First-fit:** In this technique, the memory manager searches the free-storage list for the first hole of size >= n and allocates its n bytes to the program. Searching can start either from the beginning of the list or where the previous first-fit search ended in case the free-storage list has been implemented as a circular linked list.
- **Best-fit:** In this technique, the memory manager searches the free-storage list for the smallest hole whose size is more than or equal to n and allocates its n bytes to the program. Searching always starts from the beginning of the list unless the list is sorted by size.
- **Worst-fit:** In this technique, the memory manager searches the free-storage list for the largest hole and allocates its n bytes to the program. Searching always starts from the beginning of the list unless the list is sorted by size.

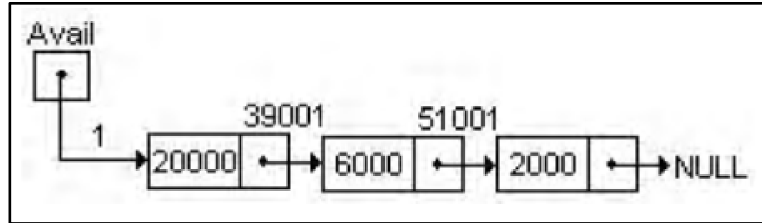
Note: If only a portion of a free block and not the entire block is to be allocated, the allocation is made from the bottom of the block to avoid changing links in the free-storage list.

Out of the above-mentioned strategies, the first-fit is generally faster and simple enough to understand. However, both first-fit and best-fit suffer from external fragmentation which leads to the fragmentation of memory into many small holes of variable sizes that individually are not large enough to satisfy a request. This happens when allocating a portion of a free block to a program leaves behind a hole that is smaller than any requests made to the system. If a program, for example, requests for a block of size 4400 bytes and a free block of size 4500 bytes is available, then allocating a portion of this block would result in a hole of 100 bytes that is too small to satisfy any other request made to the system. Moreover, the overhead to keep track of this small hole is larger than the hole itself. To get rid of this problem, some suitable constant (say, α) is chosen such that if the allocation of a portion of the free block to any program results in a hole of size less than α, then the entire block is allocated to the program.

Once a program terminates, the block of memory allocated to it gets released and is returned to the free storage list. Along with the returning of blocks, the operating system must also check whether its adjacent blocks in the free-storage list can be combined with it to form a single block. If program P2, for example, terminates [consider Figure 3.24(a)], then instead of adding this block as a node to the free-storage list [Figure 3.25(a)], it should coalesce with its left neighbor [first node of the free-storage list shown in Figure 3.24(b)] as shown in Figure 3.25(b). The coalescing of free blocks is necessary because the allocation algorithm makes the memory fragmented into small holes. With small holes, it becomes almost impossible to satisfy the larger requests even if the total free space is available.



(a)



(b)

Figure 3.25 Free- storage List

Therefore, it is necessary to check whether the adjacent blocks of the block being returned are free and if they are so, they should coalesce. To examine the adjacent blocks of the block is returned, its left and right neighbors in the free-storage list need to be accessed. Since a singly-linked list is being used, accessing the right neighbor is easy. However, accessing the left neighbor requires traversing the list from the first node. This process has to be repeated every time a block is returned to the free-storage list, which becomes very time-consuming.

Therefore, the memory manager uses a doubly-linked list to manage the memory in order to facilitate efficient memory de-allocation. Each node of this list consists of four fields: prev, size, status, and next. The prev field contains the address of the previous block, the size field contains the size of the block, the status field contains either 0 or 1 to indicate whether it is a free or allocated block, respectively, and the next field contains the address of the next block. The use of a doubly-linked list makes the traversal of the list in both forward and backward directions more convenient. Therefore, accessing the adjacent nodes of a given node becomes easier. Moreover, the inclusion of the status field in the node structure helps in determining whether the adjacent block is free or allocated.

3.6 Generalized Lists

The linked lists that we have discussed so far can contain only atomic values, such as integers, floating-point numbers, characters, etc. On the other hand, a generalized list is a general form of a linked list whose elements are either atoms or generalized lists in themselves (also called sublists). Formally, it is defined as a finite sequence of zero or more elements $\{a_0, a_1, \ldots\ldots a_k\}$ such that ai is either an atom or a sublist. It can be represented as a simple linked list; however, to indicate whether the element is an atom or a sublist, an extra field (say, tag) that takes either 0 or 1 is included in each

node (Figure 3.26). If the value of the tag is 1, then the element represented by the node is a sublist, otherwise, it is an atom. In case a node represents a sublist, it stores the address of the first node of the sub list.

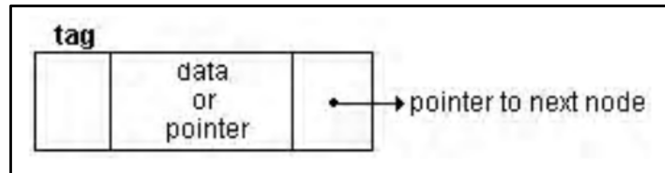


Figure 3.26 Node of a Generalized List

The structure definition of the node of a generalized list in 'C' language is as follows:

```
typedef struct node
{
        int tag;                         /*to indicate atom or sublist*/
        struct node *next;               /*to store pointer to next node*/
        union                            /*to store either data or pointer*/
        {
                int info;                /*to store data*/
                struct node *downptr;    /*to store address of the first node of sub list*/
                }data_ptr;
}GNode;                              /*node of a generalized list*/
```

For example, a generalized list {3, {4,5}, 6, {{7,8,9}, 10}}, whose first and third elements are atoms while others are sublists, can be represented as shown in Figure 3.27.

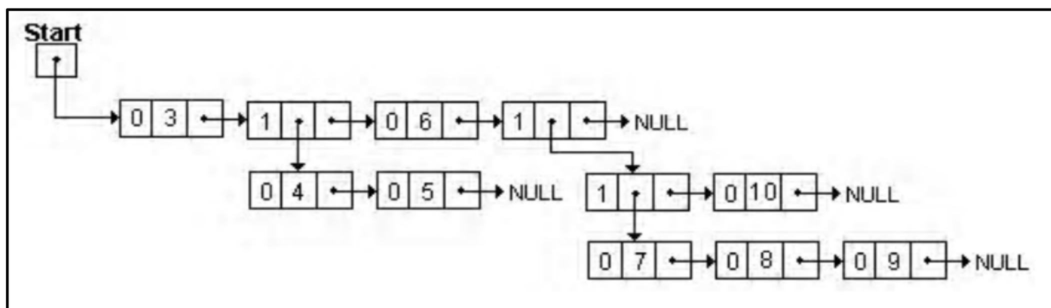


Figure 3.27 Representation of Generalized List

An important application of generalized lists is to represent polynomials in multiple

variables. To understand this, consider a polynomial in two variables
$f(x,y)=2x^5 y^3 + 4x^5 -3x^3 y^2+7x-9y^3 +7y$.
The polynomial f(x,y) can also be written as
$(2y^3 +4)x^5 +(-3y^2)x^3 +7x+(-9y^3 +7y)x^0$ by factoring out the variable x.

Now, f(x,y) may be viewed as $Ax^5 +Bx^3 +Cx+Dx^0$, where A, B, and D are polynomials (or sub-lists) in single variable y and C is a constant (or atom). Thus, this polynomial can be represented using a generalized list (Figure 3.28), where each node consists of four fields: tag, coeff, expo, and next.

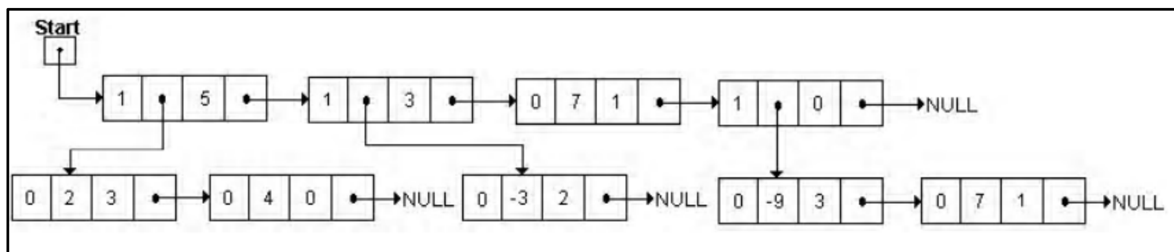


Figure 3.28 Representation of f(x,y) using Generalized List

3.6 Garbage Collection

Whenever a node of a linked list or the entire list is deleted, some memory space becomes free which can be reused by adding it to the free-storage list. To do this, there exists a program in memory called *garbage collector,* which returns unused space to the free-storage list so that it can be reused in the future. This process of collecting unused space and returning it to the free-storage list is called *garbage collection*. It may take place at the moment a node releases the memory allocated to it. Alternatively, the operating system may periodically search the memory to collect and return the unused memory space to the free-storage list, e.g., whenever there is some or no space available in the free-storage list or whenever the CPU is idle.
The former method of accomplishing garbage collection is very time- consuming. So, the latter method can be chosen, which requires two phases to carry out the garbage collection: in the first phase, the operating system goes through the memory to mark all those blocks that are still in use; and in the second phase, the operating system collects all unmarked memory blocks and returns them to the free-storage list.

3.7 Summary

- A linked list is a linear collection of homogeneous elements called nodes. The successive nodes of a linked list need not occupy adjacent memory locations, and the linear order between nodes is maintained by means of pointers.
- In a singly-linked list (also called linear linked list), each node consists of two fields: info and next. The info field contains the data and the next field contains the address of the memory location where the next node is stored.
- A linear linked list in which the next field of the last node points back to the first node instead of containing NULL is termed as a circular linked list. The main advantage of a circular linked list over a linear linked list is that by starting with any node in the circular list, we can reach any of its predecessor nodes.
- A linked list that maintains an additional pointer pointing to the previous node in each node of the list is termed as a doubly-linked list. Each node of a doubly-linked list consists of three fields: prev, info, and next. The info field contains the data, the prev field contains the address of the previous node and the next field contains the address of the next node.
- A generalized list is a general form of a linked list whose elements are either atoms or generalized lists in themselves (also called sub-lists).

3.8 Key Terms

- **Free-storage list or memory bank or free pool:** A special list consisting of unused memory cells.
- **Overflow:** A situation when there is no space available, that is, the free-storage list is empty.
- **Underflow:** A situation where the user tries to delete a node from an empty linked list.
- **Garbage collection:** The process of collecting unused space and returning it to the free-storage list.

3.9 Check Your Progress

Short- Answer type

Q1) A linked list is a linear collection of homogeneous elements called ______.

Q2) What is the structure of the node of a singly-linked list?

Q3) A new node can be inserted only at the beginning or at the end of a linked list. (True/ False?)

Q4) When a new node is inserted in between a linked list, which of these is true?

(a) Only the nodes that appear after the new node need to be moved

(b) Only the nodes that appear before the new node need to be moved

(c) The nodes that appear before and after the new node need to be moved

(d) None of the above

Q5) The process of collecting unused space and returning it to the free-storage list is called ______________.

Long- Answer type

Q1) Write an algorithm to insert a new node in a singly-linked list.

Q2) Write a brief note on

(a) Garbage collection

(b) Generalized Lists

Q3) Differentiate between a singly linked list and a circular linked list.

Q4) Describe Traversing Operation in singly linked list and circular linked list.

Q5) Write an algorithm to delete a node from a specified position in a doubly-linked list.

References

- *Classic Data Structures,* Debasis Samanta, PHI Learning Pvt. Ltd. 2nd Edition.
- *Advanced Data Structures*, Peter Brass, Cambridge University Press, New York, 2008.
- *Data Structures and Algorithms,* Aho, Ullman and Hopcroft, Addison Wesley

Module: 2

Other Data Structures

Unit 4 Stacks

Structure

4.0 Introduction
4.1 Unit Objectives
4.2 Stacks
4.3 Memory Representation of Stacks
4.4 Stack Applications
4.4.1 Recursion
4.4.2 Reversing Strings
4.4.3 Polish Notation
4.5 Summary
4.6 Key Terms
4.7 Check Your Progress

4.0 Introduction

A stack is a linear data structure in which an element can be added or removed only at one end, called the top of the stack. This unit discusses the memory representation of stacks. Various applications of stacks, such as recursion, string reversal, and Polish notation are also introduced in this unit. If a function definition includes a call to itself, it is referred to as a recursive function and the process is known as recursion or circular definition. Reversing strings is a simple application of stacks. To reverse a string, the characters of the string are pushed onto the stack one by one as the string is read from left to right. The evaluation of arithmetic expressions is another important application of stacks. The general way of writing arithmetic expressions is known as the infix notation in which the binary operator is placed between two operands on which it operates.

4.1 Unit Objectives

After going through this unit, the reader will be able to:

- Explain the memory representation of stacks.
- Explain the push and pop operations performed on stacks.
- Describe different applications of stacks, including recursion, string reversal, and evaluation of arithmetic expressions.

4.2 Stacks

A stack is a linear list of data elements in which the addition of a new element or deletion of an element occurs only at one end. This end is called Top of the stack. The operation of adding a new element in the stack and deleting an element from the stack is called ***push*** and ***pop*** respectively. Since the addition and deletion of elements always occur at one end of the stack, the last element that is pushed onto the stack is the first one to come out. Therefore, a stack is also called a Last-In-First-Out (LIFO) list.

A pile of books is one of the common examples of a stack. A new book to be added to the pile is placed at the top and a book to be removed is also taken off from the top. The book that is put most recently on the pile is the first one to be taken off. Similarly, the book at the bottom is the last one to be removed. Therefore, in order to take out the book at the bottom, all the books above it need to be removed from the pile.

Although arrays, linked lists, and stacks are linear data structures, the difference lies in insertion and deletion operations. In arrays and linked lists, insertion and deletion can take place at any place while in the case of stacks, these operations are limited to the top of the stack.

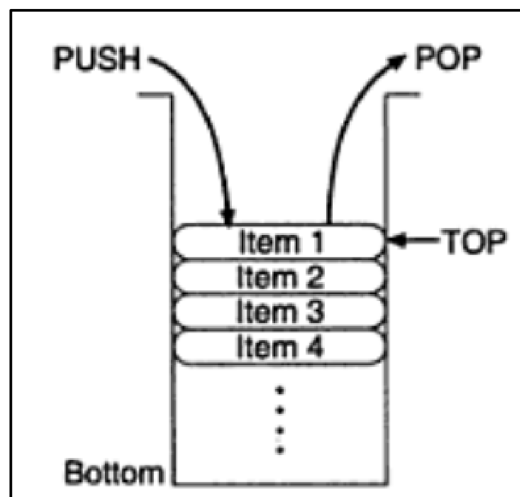


Figure 4.1 Schematic representation of a stack

(**Source-** Classic Data Structures, Debasis Samanta, PHI Learning Pvt. Ltd. 2nd Edition, Chapter- 4, Page No.- 106)

4.3 Memory Representation of Stacks

A stack can be represented in memory either as an *array* or as a *singly-linked list.* In both cases, insertion and deletion of elements are allowed at one end only. Insertion

and deletion in the middle of the array or the linked list are not allowed. An array representation of a stack is static. However, the linked list representation is dynamic in nature. Though array representation is a simple technique, it provides less flexibility and is not very efficient with respect to memory utilization. This is because if the number of elements to be stored in the stack is less than the allocated memory, then the memory space will be wasted. Conversely, if the number of elements to be handled by the stack is more than the size of the stack, then it will not be possible to increase the size of the stack to store these elements.

Array Representation of Stacks

When stacks are represented as arrays, a variable named Top is used to point to the top element of the stack. Initially, the value of Top is set to –1 to indicate an empty stack. To push an element onto the stack, Top is incremented by one, and the element is pushed at that position. When Top reaches MAX-1 and an attempt is made to push a new element, then stack overflows. Here, MAX is the maximum size of the stack. Similarly, to pop (or remove) an element from the stack, the element on the top of the stack is assigned to a local variable, and then Top is decremented by one. When the value of Top is equal to –1 and an attempt is made to pop an element, the stack underflows.

Therefore, before inserting a new element onto the stack, it is necessary to test the condition of overflow. Similarly, before removing the top element from the stack, it is necessary to check the condition of the underflow. The total number of elements in a stack at a given point of time can be calculated from the value of Top as follows:

Number of elements = Top + 1

Figure 4.2 shows an empty STACK with size 3 and Top = –1.

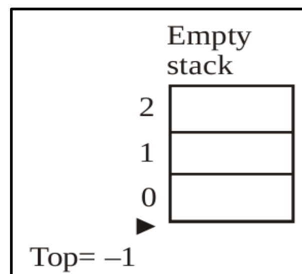


Figure 4.2 Initial State of STACK

To insert an element 1 in the Stack, Top is incremented by one and the element 1 is stored at Stack [Top]. Similarly, other elements can be added to the Stack until Top reaches 2 (Figure 4.3). To pop an element from the Stack (data element 3), Top is decremented by one, which removes element 3 from the Stack. Similarly, other elements can be removed from the Stack until Top reaches –1. Figure 4.3 shows different states of Stack after performing push and pop operations on it.

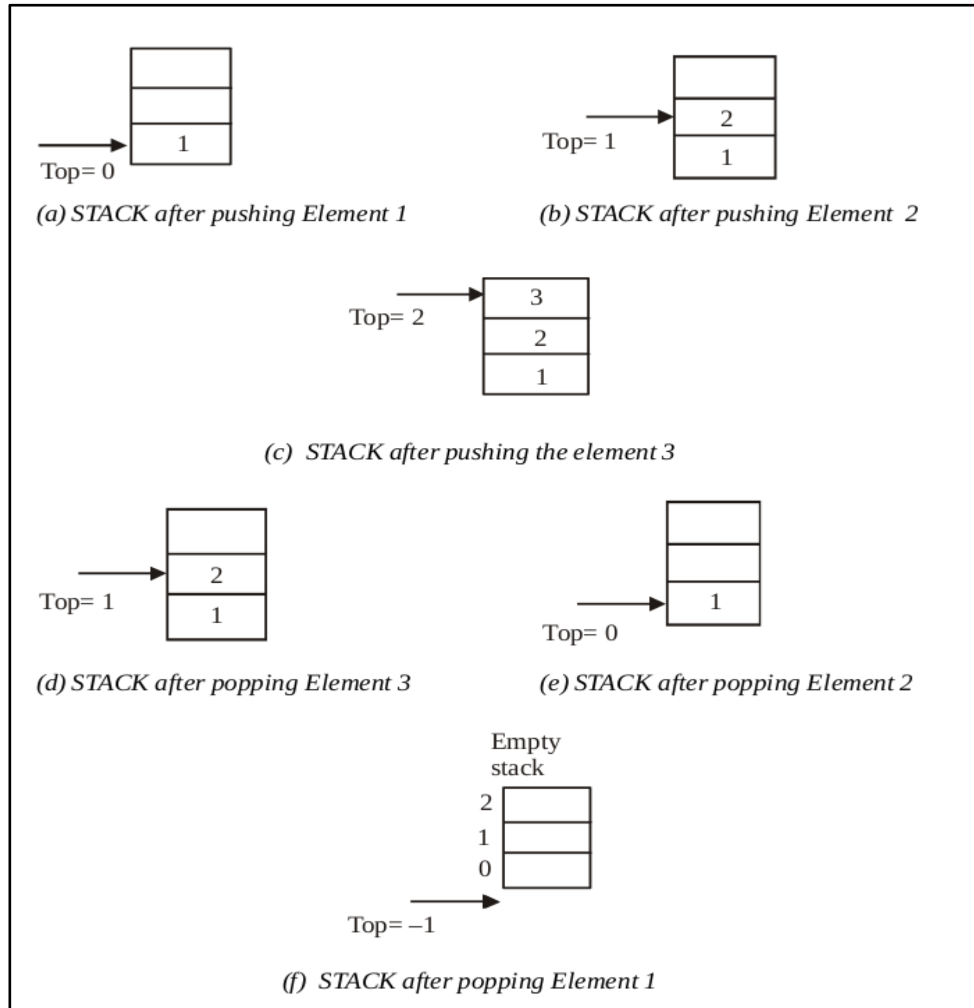


Figure 4.3 Various States of STACK after Push and Pop Operations

To implement a stack as an array in C language, the following structure named stack needs to be defined.

```
struct stack
{
	int item[MAX];
```

```
            int Top;
};
```

Algorithm 4.1 Push Operation on Stack	
push(s, element)	*//s is a pointer to stack*
1. If s->Top = MAX-1	*//checking for stack overflow*
Print "Overflow: Stack is full!" and go to step 5	
End If	
2. Set s->Top = s->Top + 1	*//incrementing Top by 1*
3. Set s->item[s->Top] = element	*//inserting element in the stack*
4. Print "Value is pushed onto the stack..."	
5.End	

Algorithm 4.2 Pop Operation on Stack	
pop(s)	
1. If s->Top = -1	*//checking for stack underflow*
Print "Underflow: Stack is empty!"	
Return 0 and go to step 5	
End If	
2. Set popped = s->item[s->Top]	*//taking off the top element from the stack*
3. Set s->Top = s->Top - 1	*//decrementing Top by 1*
4. Return popped	
5. End	

Program 4.1: A program to implement a stack as an array.

```
#include<stdio.h>
#include<conio.h>
#define MAX 10
#define True 1
#define False 0
typedef struct stack
{
            int item[MAX];
            int Top;
}stk;
/*Function prototypes*/
```

```
void createstack(stk *);            /*to create an empty stack*/
void push(stk *, int);              /*to push an element onto the stack*/
int pop(stk *);                     /*to pop the top element from the stack*/
int isempty(stk *);                 /*to check for the underflow condition*/
int isfull(stk *);                          /*to check for the overflow condition*/
void main()
{
        int choice;
        int value;
        stk s;
        createstack(&s);
        do{
                clrscr();
                printf("\n\tMain Menu");
                printf("\n1. Push");
                printf("\n2. Pop");
                printf("\n3. Exit\n");
                printf("\nEnter your choice: ");
                scanf("%d", &choice);
                switch(choice)
                {
                        case 1: printf("\nEnter the value to be inserted: ");
                                scanf("%d", &value);
                                push(&s, value);
                                getch();
                                break;
                        case 2: value=pop(&s);
                                if (value==0)
                                        printf("\nUnderflow: Stack is empty!");
                                else
                                        printf("\nPopped element is: %d", value);
                                getch();
                                break;
                        case 3: exit();
                                default: printf("\nInvalid choice!");
                }
        }while(1);
```

```
}
void createstack(stk *s)
{
        s->Top=-1;
}
void push(stk *s, int element)
{
        if (isfull(s))
        {
                printf("\nOverflow: Stack is full!");
                return;
        }
        s->Top++;
        s->item[s->Top]=element;
        printf("\nValue is pushed onto the stack…");
}
int pop(stk *s)
{
        int popped;
        if (isempty(s))
                return 0;
        popped=s->item[s->Top];
        s->Top—;
        return popped;
}
int isempty(stk *s)
{
        if (s->Top==-1)
                return True;
        else
                return False;
}
int isfull(stk *s)
{
        if (s->Top==MAX-1)
                return True;
        else
```

```
        return False;
}
```

The output of the program is:

```
          Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 1
Enter the value to be inserted: 23
Value is pushed onto the stack...
Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 1
Enter the value to be inserted: 35
Value is pushed onto the stack...
Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 1
Enter the value to be inserted: 40
Value is pushed onto the stack...
          Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 2
Popped element is: 40
          Main Menu
1. Push
```

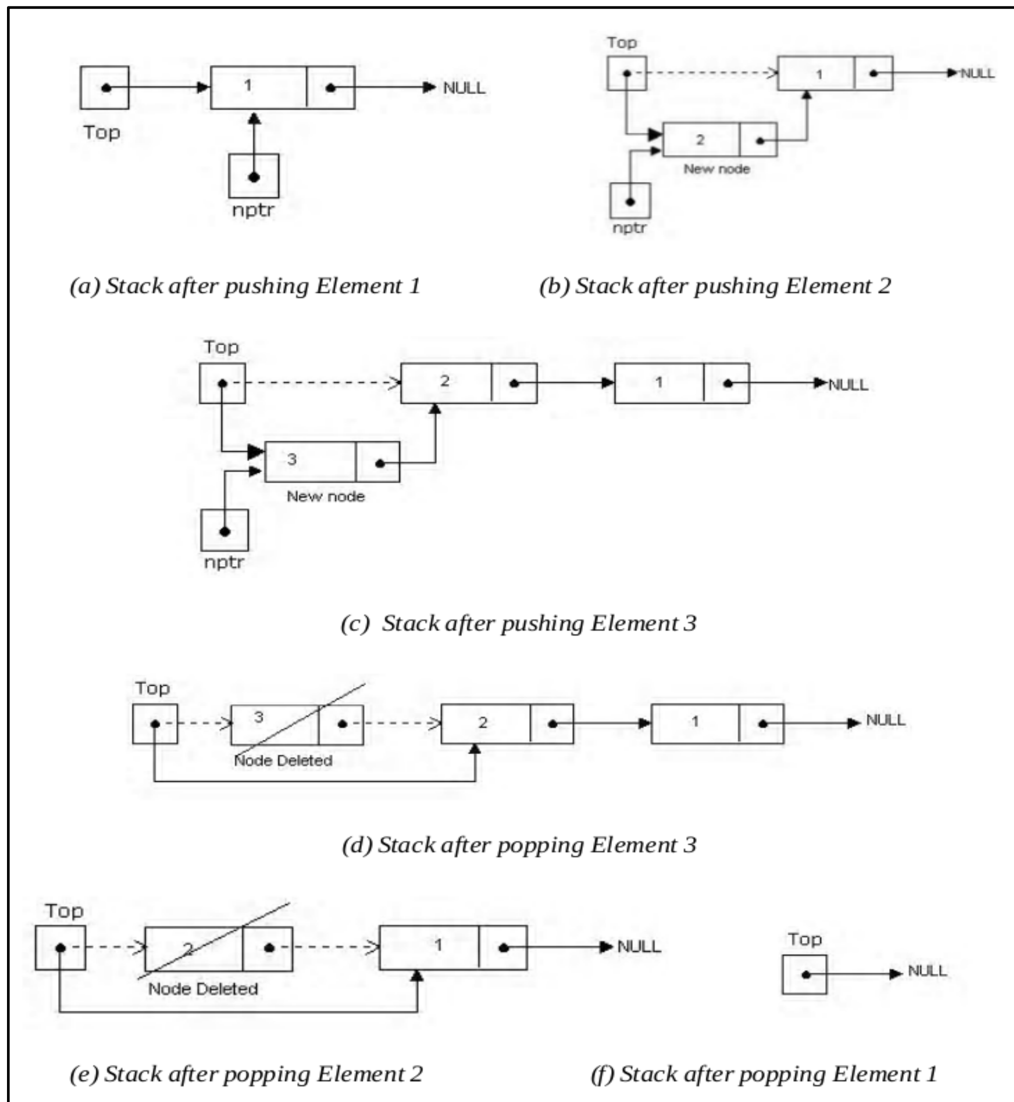
2. Pop
3. Exit
Enter your choice: 2
Popped element is: 35
Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 2
Popped element is: 23
Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 2
Underflow: Stack is empty!
Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 3

Linked List Representation of Stacks

A stack implemented as a singly-linked list is commonly known as a linked stack. When a stack is implemented as a linked list, a pointer variable Top is used to point to the top element of the stack. Initially, Top is set to NULL to indicate an empty stack. Whenever a new element is to be inserted in the stack, a new node is created, and the element is inserted into the node. Then the Top is modified to point to this new node. Since the memory is allocated dynamically, a linked stack reaches the overflow condition when no more memory space is available to be allocated dynamically.

Consider a linked stack, say, Stack. Each node of Stack has two members, info and next that represent the element in a stack and a pointer to the next node,

respectively. The pointer Top points to the top node of Stack and is initially set to NULL for an empty stack. To push an element onto the Stack, a new node nptr is created and the element is inserted into it. Then, the Top is modified to point to nptr. On the other hand, to pop an element from the Stack, a temporary pointer is created which is made to point to the node pointed to by Top. Then Top is modified to point to the next node in the Stack, and the temporary node is deleted from the memory. The different states of Stack after performing push and pop operations on it are shown in Figure 4.4.



(a) Stack after pushing Element 1 *(b) Stack after pushing Element 2*

(c) Stack after pushing Element 3

(d) Stack after popping Element 3

(e) Stack after popping Element 2 *(f) Stack after popping Element 1*

Figure 4.4 Various States of Linked Stack after Push and Pop Operations

Algorithm 4.3 Push Operation on Linked Stack
push(Top,element) *1. Allocate memory for nptr //nptr is a pointer to the new node* *2. If nptr = NULL //checking for stack overflow* *Print "Overflow: Memory not allocated!" and go to step 6* *End If* *3. Set nptr->info = element* *4. Set nptr->next = Top* *5. Set Top = nptr* *6. End*

Algorithm 4.4 Pop Operation on Linked Stack
pop(Top) *1. If Top = NULL //checking for stack underflow* *Print "Underflow: Stack is empty!"* *Return 0 and go to step 7* *End If* *2. Set popped = Top->info //popped is a data item at the top of the stack* *3. Set temp = Top //temp is a temporary pointer, initialized with Top* *4. Set Top = Top->next //making Top point to the next node in the linked stack* *5. Deallocate temp //de-allocating memory* *6. Return popped* *7. End*

Program 4.2: A program to illustrate the implementation of a stack as linked list.

```
#include<stdio.h>
#include<conio.h>
#define True 1
#define False 0
typedef struct node
{
	int info;
	struct node *next;
}Node;
void createstack(Node **);
```

```
int isempty(Node *);
void push(Node **,int);
int pop(Node **);
void main()
{
	int choice, value;
	Node *Top;
	createstack(&Top);
	do
	{
		clrscr();
		printf("\n\tMain Menu");
		printf("\n1. Push ");
		printf("\n2. Pop ");
		printf("\n3. Exit\n");
		printf("\nEnter your choice: ");
		scanf("%d", &choice);
		switch(choice)
		{
			case 1: printf("\nEnter the value to be inserted: ");
				scanf("%d",&value);
				push(&Top,value);
				getch();
				break;
			case 2: value=pop(&Top);
				if(value==0)
					printf("\nUnderflow: Stack is empty! ");
				else
					printf("\nPopped item is: %d",value);
				getch();
				break;
			case 3: exit();
				default: printf("\nInvalid choice!");
		}
	}while(1);
}
void createstack(Node **Top)
```

```
{
        *Top=NULL;
}
int isempty(Node *Top)
{
        if(Top==NULL)
                return True;
        else
                return False;
}
void push(Node **Top, int element)
{
        Node *nptr;
        nptr=(Node*)malloc(sizeof(Node));
        if (nptr==NULL)
        {
                printf("\nOverflow: Memory not allocated!");
                return;
        }
        nptr->info=element;
        nptr->next=*Top;
        *Top=nptr;
        printf("\nValue is pushed onto the stack...");
}
int pop(Node **Top)
{
        int popped;
        Node *temp;
        if(isempty(*Top))
                return 0;
        popped=(*Top)->info;
        temp=*Top;
        *Top=(*Top)->next;
        free(temp);
        return popped;
}
```

The output of the program is:

```
          Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 1
Enter the value to be inserted: 23
Value is pushed onto the stack...
          Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 1
Enter the value to be inserted: 34
Value is pushed onto the stack...
Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 2
Popped item is: 34
          Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 2
Popped item is: 23
          Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 2
Underflow: Stack is empty!
```

Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 3

4.4 Stack Applications

Stacks are used where the last-in-first-out principle is required, such as implementing recursion, string reversal, evaluation of arithmetic expressions, implementing function calls, etc. Some of the applications are discussed below.

4.4.1 Recursion

When a function definition includes a call to itself, it is referred to as a recursive function and the process is known as recursion or circular definition. A recursive function is said to be well-defined if it satisfies the following two properties:

- The arguments passed to the recursive function must have certain base values for which the function does not call itself. In simple words, a recursive function must include a condition or a statement to terminate the function.
- Each time the function calls itself (directly or indirectly), the argument of the function must get closer to the base value.

In each recursive call, the current values of the parameters, local variables, and the return address where the control has to return from the call are required to be stored. For storing all these values, a stack is maintained. When a recursive function is called for the first time, space is set aside in the memory to execute this call and the function body is executed. Then a second call to the function is made; again space is set for this call, and so on. These memory areas for each function call are arranged in the stack. Each time the function is called, its memory area is placed on the top of the stack and is removed when the execution of the call is completed (Figure 4.5).

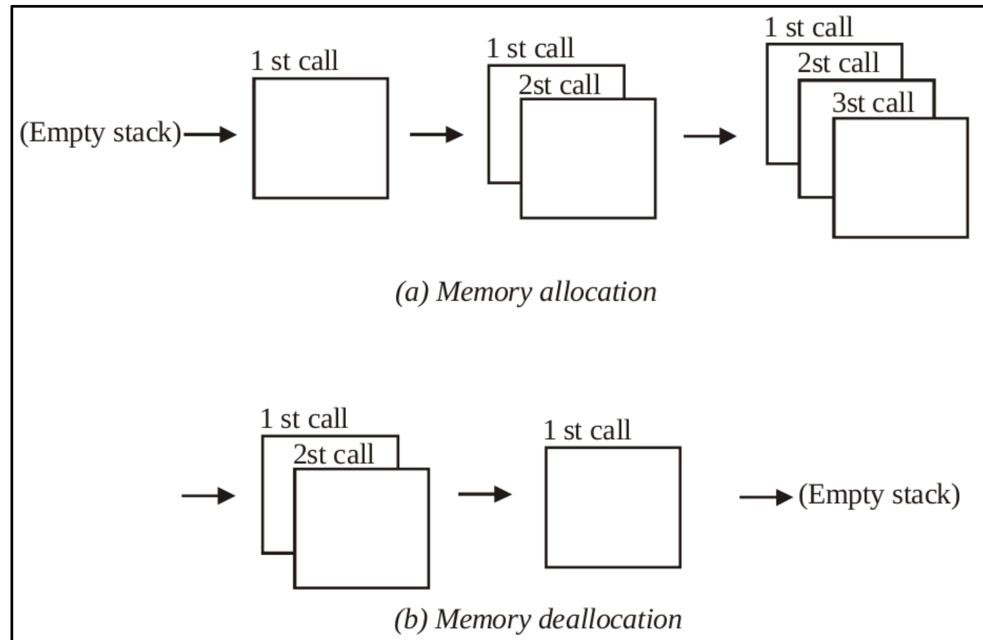


Figure 4.5 Calling Recursive Functions

Recursion is one of the most powerful concepts in computer science. Many mathematical problems, such as computing factorial of a given number, finding the Fibonacci series, determining the Greatest Common Divisor (GCD) of two positive numbers, computing binary equivalent of a decimal number, computing binomial coefficient, etc., can be solved efficiently using recursion. In this section, we are discussing some of these problems.

Factorial of a given number

The factorial of a given positive number n is defined as the product of all the numbers from 1 to n (both inclusive). It is usually denoted by n! that is:

n!=1*2*3*4*...*(n-2)*(n-1)*n

For example, if n=6, then 6! = 1*2*3*4*5*6 = 720

Observe that 6! = 6*5!

Similarly, 5! = 5*4!

In general, for every positive number n, n! = n*(n-1)!. Note that for n=0, we can define 0! = 1. Therefore, when n becomes zero, the recursive function terminates (thus, 0 is the base value in this case).

Algorithm 4.5 Factorial of a Number

fact(n)
1. If (n == 0)
        return 1 and go to step 2
    Else
        *return (n * fact(n-1)) //recursive call to fact()*
    End If
2. End

Program 4.3: A program to implement the recursive function for computing factorial of a given number.

```
#include<stdio.h>
#include<conio.h>
/*Function prototype*/
int fact(int);
void main()
{
        int n;
        clrscr();
        printf("Enter a number: ");
        scanf("%d", &n);
        printf("\nFactorial of %d is %d", n, fact(n));
        getch();
}
int fact(int i)
{
        if(i==0)
                return 1;
        else
                return (i*fact(i-1));                    /*recursive call to fact()*/
}
```

The output of the program is:

Enter a number: 7

Factorial of 7 is 5040

Fibonacci series

The Fibonacci series up to n terms (generally denoted by F_0, F_1, F_3, ..., F_n), is generated as follows:

0, 1, 1, 2, 3, 5, 8, 13, 21...

That is, each successive term is the sum of its two preceding terms. That is,

If n = 0 or n=1 (0 and 1 are the base values in this case)

Then

$F_n = n$,

Else

$F_n = F_{n-2} + F_{n-1}$

Algorithm 4.6 Fibonacci Series up to n terms

fib(n)
1. If (n == 0 OR n == 1)
return n and go to step 2
Else
return (fib(n-1) + fib(n-2))
fib()
End If
2. End

Program 4.4: A program to implement the recursive function for finding the Fibonacci series up to n terms.

```
#include<stdio.h>
#include<conio.h>
/*Function prototype*/
unsigned fib(unsigned int);
void main()
{
        unsigned int n;
        int i;
        clrscr();
        printf("Enter the number of terms to be generated: ");
        scanf("%d", &n);
```

```
    printf("\nFibonacci series up to %d terms is: \n\n", n);
    for(i=0;i<n;i++)
        printf("%u ", fib(i));
    getch();
}
unsigned fib(unsigned int i)
{
    if(i==0 || i==1)
        return i;
    else
        return (fib(i-1)+fib(i-2));        /*recursive call to fib()*/
}
```

The output of the program is:

Enter the number of terms to be generated: 15

Fibonacci series up to 15 terms is:

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377

Greatest common divisor

Another example of recursive function is to determine the greatest common divisor (GCD) of two positive numbers using Euclid's algorithm, which is given below:

$$GCD(n1, n2) = \begin{cases} GCD(n2, n1), & \text{if } (n1<n2) \\ n2, & \text{if } (n1>=n2) \text{ and } n1 \bmod n2=0 \\ GCD(n2, n1 \bmod n2), & \text{otherwise} \end{cases}$$

Algorithm 4.7 GCD of Two Numbers

GCD(p, q)
1. Set rem = p % q
2. If (p >= q AND rem == 0)
    return q and go to step 3
  Else
    Call GCD(q, rem) //recursive call to GCD()
  End If
3. End

Program 4.5: A program to implement the recursive function for finding the GCD of two positive numbers.

```
#include<stdio.h>
#include<conio.h>
/*Function prototype*/
int GCD(int, int);
void main()
{
	int num1, num2;
	clrscr();
	printf("Enter first number: ");
	scanf("%d", &num1);
	printf("\nEnter second number: ");
	scanf("%d", &num2);
	printf("\nGCD of %d and %d is: %d", num1, num2, GCD(num1, num2));
	getch();
}
int GCD(int p, int q)
{
	int rem=p%q;
	if((p>=q) && (rem==0))
		return q;
	else
		GCD(q, rem);			/*recursive call to GCD()*/
}
```

The output of the program is:

Enter the first number: 49

Enter second number: 63

GCD of 49 and 63 is: 7

Decimal to binary conversion

The conversion of decimal number n to its equivalent binary number can also be performed in a recursive manner. The recursive function, in this case, terminates when n becomes 0 or 1. Therefore, 0 and 1 are the base values in this case.

Algorithm 4.8 Decimal to Binary Conversion

binary(n)
1. If (n == 1 OR n == 0)
Print n and go to step 2
Else
Set rem = n % 2
Set n = n / 2
Call binary(n)   //recursive call to binary()
Print rem
End If
2. End

Program 4.6: *A program to implement the recursive function for finding the binary equivalent of a decimal number.*

```
#include<stdio.h>
#include<conio.h>
/*Function prototype*/
void binary(int);
void main()
{
        int num;
        clrscr();
        printf("Enter a number: ");
        scanf("%d", &num);
        printf("\nBinary equivalent of this number is ");
        binary(num);
        getch();
}
void binary(int n)
{
        int rem;
        if (n==1||n==0)
                printf("%d", n);
        else
```

```
{
        rem=n%2;
        n=n/2;
        binary(n);              /*recursive call to binary()*/
        printf("%d", rem);
        }
}
```

The output of the program is:

Enter a number: 89

Binary equivalent of this number is 1011001

4.4.2 Reversing Strings

Another simple application of stacks is reversing strings. To reverse a string, the characters of the string are pushed onto the stack one by one as the string is read from left to right. Once all the characters of the string are pushed onto the stack, they are popped one by one. Since the character last pushed in comes out first, subsequent pop operations result in the reversal of the string.

For example, to reverse the string "REVERSE", the string is read from left to right and its characters are pushed onto a stack, starting from the letter R, then E, V, E, and so on as shown in Figure 4.6.

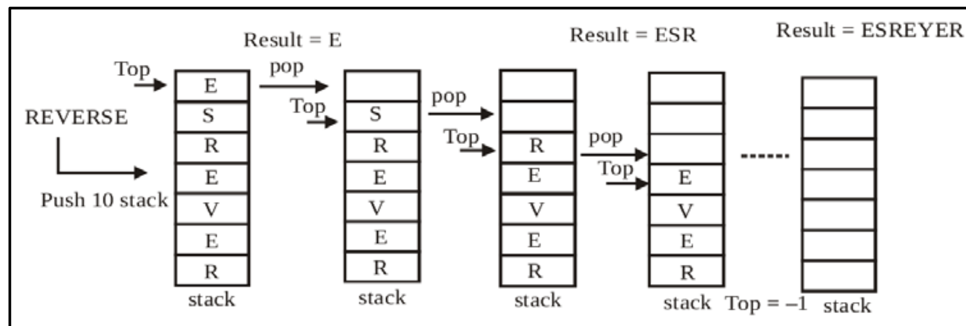


Figure 4.6 Reversing a String using a Stack

Once all the letters are stored in the stack, they are popped one by one. Since the letter at the top of the stack is E, it is the first letter to be popped. The subsequent pop operations take out the letters S, R, E, and so on. Thus, the resultant string is the reverse of the original one as shown in the above figure.

Algorithm 4.9 String Reversal using Stack

reversal(s, str)
1. Set i = 0
2. While(i < length_of_str)
    Push str[i] onto the stack
    Set i = i + 1
  End While
3. Set i = 0
4. While(i < length_of_str)
    Pop the top element of the stack and store it in str[i]
    Set i = i + 1
  End While
5. Print "The reversed string is: ", str
6. End

Program 4.7: A program to reverse a given string using stacks.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#define MAX 101
typedef struct stack
{
	char item[MAX];
	int Top;
}stk;
/*Function prototypes*/
void createstack(stk *);
void reversal(stk *, char *);
void push(stk *, char);
char pop(stk *);
void main()
{
	stk s;
	char str[MAX];
	int i;
```

```
        createstack(&s);
        clrscr();
        do
        {
        printf("Enter any string (max %d characters): ", MAX-1);
        for(i=0;i<MAX;i++)
        {
                scanf("%c", &str[i]);
                if(str[i]=='\n')
                break;
        }
        str[i]='\0';
}while(strlen(str)==0);
reversal(&s, str);
getch();
}
void createstack(stk *s)
{
        s->Top=-1;
}
void reversal(stk *s, char *str)
{
        int i;
        for (i=0;i<strlen(str);i++)
        push(s, str[i]);
        for(i=0;i<strlen(str);i++)
        str[i]=pop(s);
        printf("\nThe reversed string is: %s", str);
}
void push(stk *s, char item)
{
        s->Top++;
        s->item[s->Top]=item;
}
char pop(stk *s)
{
        char popped;
```

```
        popped=s->item[s->Top];
        s->Top—;
        return popped;
}
```

The output of the program is:

Enter any string (max 100 characters): Hello World

The reversed string is: dlroW olleH

4.4.3 Polish Notation

Another important application of stacks is the evaluation of arithmetic expressions. The general way of writing arithmetic expressions is known as the infix notation where the binary operator is placed between two operands on which it operates. (For simplicity, we have ignored expressions containing unary operators). The expressions a+b and (a-c)*d, ((a+b)*(d/f)-f) , for example, are in infix notation. The order of evaluation in these expressions depends on the parentheses and the precedence of operators. The order of evaluation of the expression (a+b)*c, for example, is different from that of a+(b*c). As a result, it is difficult to evaluate an expression in infix notation. Thus, the arithmetic expressions in the infix notation are converted to another notation, which can be easily evaluated by a computer system to produce correct results.

A Polish mathematician, ***Jan Lukasiewicz***, suggested two alternative notations to represent an arithmetic expression. In these notations, the operators can be written either before or after the operands on which they operate.

The notation in which an operator occurs before its operands are known as the *prefix notation* (also known as the *Polish notation*). For example, the expressions +ab and *-acd are in prefix notation. On the other hand, the notation in which an operator occurs after its operands is known as the *postfix notation* (also known as the *Reverse Polish or suffix notation*). The expressions ab+ and ac-d*, for example, are in postfix notation.

A characteristic feature of prefix and postfix notations is that the order of evaluation of an expression is determined by the position of the operator and operands in the expression. In other words, the operations are performed in the order in which the operators are encountered in the expression. Hence, parentheses are not required for

the prefix and postfix notations. Moreover, while evaluating the expression, the precedence of operators is insignificant. As a result, they are compiled faster than the expressions in infix notation. Note that the expressions in infix notation can be converted to both prefix and postfix notations. Here we will discuss infix to postfix conversion only and the evaluation of postfix expressions.

Conversion of infix to postfix notation

To convert an arithmetic expression from infix notation to postfix notation, the precedence and associativity rules of operators are always kept in mind. The operators of the same precedence are evaluated from left to right. This conversion can be performed either manually (without using stacks) or by using stacks. The three steps for converting the expression manually are given here.

Step1: The actual order of evaluation of the expression in infix notation is determined by inserting parentheses in the expression according to the precedence and associativity of operators.

Step 2: The expression in the innermost parentheses is converted into postfix notation by placing the operator after the operands on which it operates.

Step 3: Step 2 is repeated until the entire expression is converted into a postfix notation.

For example, to convert the expression a+b*c into equivalent postfix notation, the following steps are performed:

1. Since the precedence of * is higher than +, the expression b*c has to be evaluated first. Hence, the expression is written as:

   (a+(b*c))

2. The expression in the innermost parentheses, that is, b*c is converted into its postfix notation. Hence, it is written as bc*. The expression now becomes:

   (a+bc*)

3. Now the operator + has to be placed after its operands. The two operands for the + operator are a and the expression bc*. The expression now becomes:

   (abc*+)

   Hence, the equivalent postfix expression is:

   abc*+

When expressions are complex, manual conversion becomes difficult. On the other

hand, the conversion of an infix expression into a postfix expression is simple when it is implemented through stacks. In this method, the infix expression is read from left to right, and a stack is used to temporarily store the operators and the left parenthesis. The order in which the operators are pushed onto and popped from the stack depends on the precedence of operators and the occurrence of parenthesis in the infix expression. The operands in the infix expression are not pushed onto the stack; rather they are directly placed in the postfix expression. Note that the operands maintain the same order as in the original infix notation.

Algorithm 4.10 Infix to Postfix Conversion

```
infixtopostfix(s, infix, postfix)
1. Set i = 0
2. While (i < number_of_symbols_in_infix)
        If infix[i] is a whitespace or comma
            Set i = i + 1 and continue
       If infix[i] is an operand, add it to postfix
       Else If infix[i] = '(', push it onto the stack
       Else If infix[i] is an operator, follow these steps:
            i. For each operator on the top of the stack whose precedence is greater than or
equal
              to the precedence of the current operator, pop the operator from the stack and add
              it to the postfix
            ii. Push the current operator to the stack
       Else If infix[i] = ')', follow these steps:
            i. Pop each operator from the top of the stack and add it to postfix until '(' is
encountered in the stack
            ii. Remove '(' from the stack and do not add it to the postfix
          End If
          Set i = i + 1
   End While
3. End
```

For example, consider the conversion of the following infix expression to postfix expression:

a-(b+c)*d/f

Initially, a left parenthesis '(' is pushed onto the stack, and the infix expression is appended with a right parenthesis ')'. The initial state of the stack, infix expression, and postfix expression is shown in Figure 4.7.

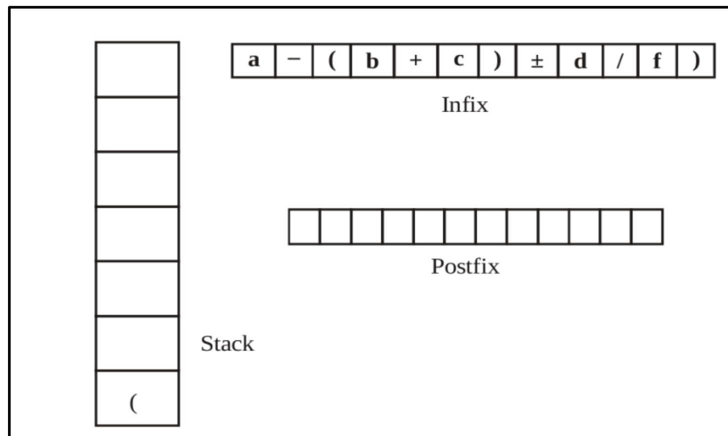


Figure 4.7 Initial State of the Stack, Infix Expression, and Postfix Expression

infix is read from left to right and the following steps are performed.

1. The operand 'a' is encountered, which is directly put to postfix.
2. The operator '–' is pushed onto the stack.
3. The left parenthesis '(' is pushed onto the stack.
4. The next element is b which being an operand is directly put to postfix.
5. '+' being an operator is pushed onto the stack.
6. Next, 'c' is put to postfix.
7. The next element is the right parenthesis ')' and, hence, the operators on the top of the stack are popped until '(' is encountered in the stack. Till now, the only operator in the stack above the '(' is '+', which is popped and put to postfix. '(' is popped and removed from the stack (Figure 4.8 (a). Figure 4.8 (b) shows the current position of the stack.

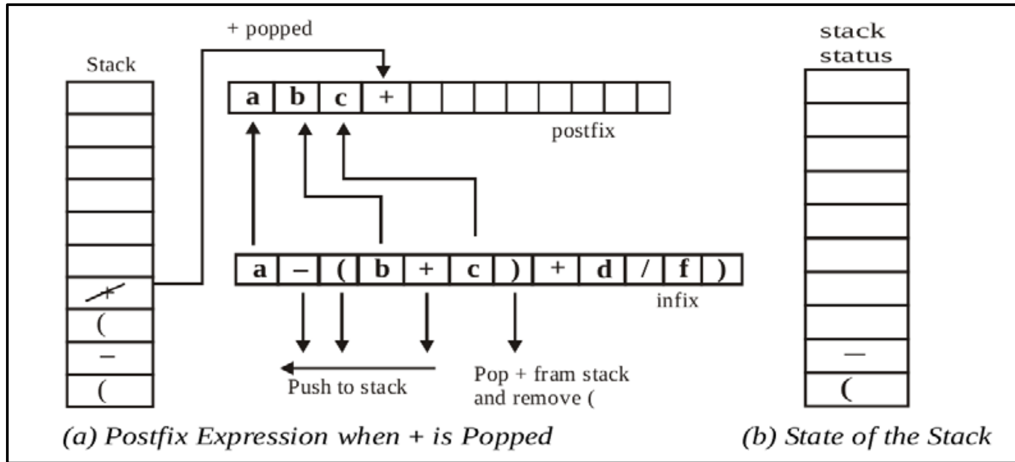


Figure 4.8 Intermediate States of Postfix and Infix Expressions and the Stack

8. After this, the next element '*' is an operator and, hence, it is pushed onto the stack.
9. Then, 'd' is put to postfix.
10. The next element is '/'. Since the precedence of / is the same as the precedence of *, the operator * is popped from the stack and / is pushed onto the stack (Figure 4.9).
11. The operand 'f' is directly put to postfix after which ')' is encountered.
12. On reaching ')', the operators in the stack before the next '(' is reached are popped. Hence, / and – are popped and put to postfix as shown in Figure 4.9.

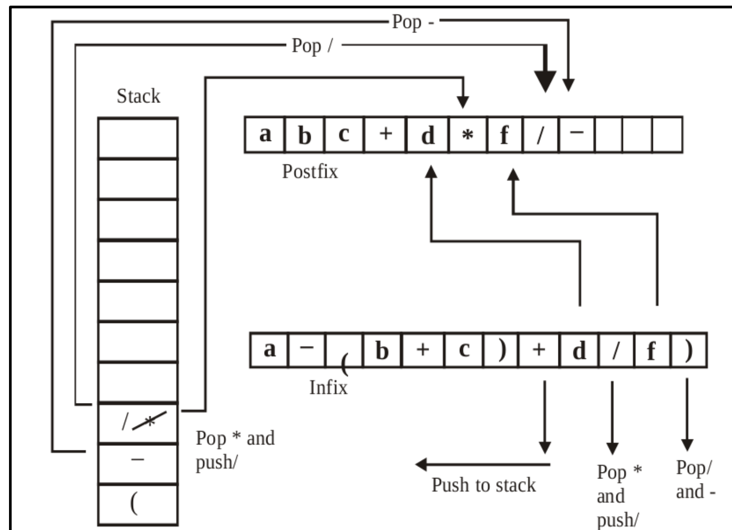


Figure 4.9 The State when – and / are Popped

13. '(' is removed from the stack. Since the stack is empty, the algorithm is terminated and a postfix is printed.

The stepwise conversion of expression a-(b+c)*d/f into its equivalent postfix expression is shown in Table 2.1.

Table 2.1 Conversion of Infix Expression into Postfix

Element	Action performed	Stack status	Postfix expression
a	Put to postfix	(	a
-	Push	(-	a
(	Push	(-(	a
b	Put to postfix	(-(	ab
+	Push	(-(+	ab
c	Put to postfix	(-(+	abc
)	Pop +, put to postfix, pop (	(-	abc+
*	Push	(-*	abc+
d	Put to postfix	(-*	abc+d
/	Pop *, put to postfix, push /	(-/	abc+d*
f	Put to postfix	(-/	abc+d*f
)	Pop / and -	Empty	abc+d*f/-

Program 4.8: A program to convert an expression from infix notation to postfix notation.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#define MAX 102
typedef struct stack
{
        char item[MAX];
        int Top;
}stk;
/*Function prototypes*/
void createstack(stk *);
void infixtopostfix(stk *, char *, char *);
int precedence(char);
int isOperator(char);
void push(stk *, char);
char pop(stk *);
void main()
{
```

```
        stk s;
        char infix[MAX], *postfix;
        int i, len;
        clrscr();
        createstack(&s);
        do
        {
                printf("\nEnter expression in infix notation (max %d characters): ", MAX-2);
                for(i=0;i<MAX-1;i++)
                {
                        scanf("%c", &infix[i]);
                        if(infix[i]=='\n')
                        break;
                }
                infix[i]=')';
                infix[i+1]='\0';
        }while(strlen(infix)==0);
        push(&s, '(');
        len=strlen(infix);
        postfix=(char*)malloc(len+1);
        infixtopostfix(&s, infix, postfix);
        printf("\nThe equivalent postfix expression is %s", postfix);
        getch();
}
void createstack(stk *s)
{
        s->Top=-1;
}
void infixtopostfix(stk *s, char *in, char *po)
{
        int i,j,len, preStack, preOp;
        char popped;
        len=strlen(in);
        i=j=0;
        while(i<len)
        {
                if (in[i]==' '||in[i]=='\t' || in[i]==',')
```

```
                {
                        i++;
                        continue;
                }
                if(in[i]=='(')
                        push(s, in[i]);
                else if(isOperator(in[i]))
                {
                        preStack=precedence(s->item[s->Top]);
                        preOp=precedence(in[i]);
                        while(preStack>=preOp)
                        {
                                po[j++]=pop(s);
                                preStack=precedence(s->item[s->Top]);
                        }
                        push(s, in[i]);
                }
                else if(in[i]==')')
                {
                        while((popped=pop(s))!='(')
                        {
                                po[j++]=popped;
                        }
                }
                else
                        po[j++]=in[i];
                i++;
                }
                po[j]='\0';
}
void push(stk *s, char item)
{
        s->Top++;
        s->item[s->Top]=item;
}
char pop(stk *s)
{
```

```
        char popped;
        popped=s->item[s->Top];
        s->Top—;
        return popped;
}
int isOperator(char op)
{
        switch(op)
        {
                case '^':
                case '+':
                case '-':
                case '*':
                case '/': return 1;
        }
        return 0;
}
int precedence(char op)
{
switch(op)
{
        case '^': return 3;
        case '/':
        case '*':
        case '%': return 2;
        case '+':
        case '-': return 1;
        }
        return 0;
}
```

The output of the program is:

Enter expression in infix notation (max 100 characters): A+(B*C- (D/E^F)*)*H

The equivalent postfix expression is ABC*DEF^/*-H*+

Evaluation of postfix expression

In a computer system when an arithmetic expression in an infix notation needs to be evaluated, it is first converted into its postfix notation. The equivalent postfix expression is then evaluated. The evaluation of postfix expressions is also implemented through stacks. Since the postfix expression is evaluated in the order of appearance of operators, parentheses are not required in the postfix expression.

During the evaluation, a stack is used to store the intermediate results of the evaluation. Since an operator appears after its operands in a postfix expression, the expression is evaluated from left to right. Each element in the expression is checked whether it is an operator or an operand. If the element is an operand, it is pushed onto the stack. On the other hand, if the element is an operator, the first two operands are popped from the stack and the operation is performed on them. The result of the operation is then pushed back to the stack. This process is repeated until the entire expression is evaluated.

Algorithm 4.11 Evaluation of a Postfix Expression

```
evaluationofpostfix(s, postfix)
1. Set i = 0, RES=0.0
2. While (i < number_of_characters_in_postfix)
           If postfix[i] is a whitespace or comma
                 Set i = i + 1 and continue
           If postfix[i] is an operand, push it onto the stack
           If postfix[i] is an operator, follow these steps:
              I.  Pop the top element from stack and store it in operand2
             II.  Pop the next top element from stack and store it in operand1
            III.  Evaluate operand2 op operand1, and store the result in RES (op is the current operator)
             IV.  Push RES back to stack
        End If
        Set i = i + 1
   End While
3. Pop the top element and store it in RES
4. Return RES
5. End
```

For example, consider the evaluation of the following postfix expression using stacks:

abc+d*f/-

where, a=6, b=3, c=6, d=5, f=9

After substituting the values of a, b, c, d, and f, the postfix expression becomes:

636+5*9/-

The expression is evaluated as follows:

1. The expression is read from left to right and each element is checked whether it is an operand or an operator.
2. The first element is '6', which being an operand is pushed onto the stack.
3. Similarly, operands '3' and '6' are pushed onto the stack.
4. The next element is '+', which is an operator. Hence, the element at the top of stack '6' and the next top element '3' are popped from the stack as shown in Figure 4.10.

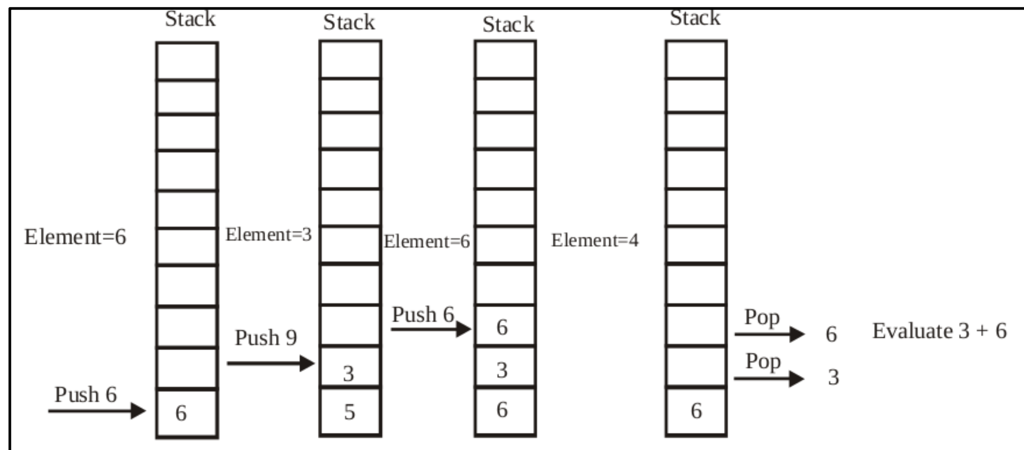


Figure 4.10 Evaluation of the Expression using Stacks

5. The expression 3+6 is evaluated and the result (that is, 9) is pushed back to stack as shown in Figure 4.11.
6. The next element in the expression, that is 5, is pushed to the stack.
7. The next element is '*', which is a binary operator. Hence, the stack is popped twice and elements 5 and 9 are taken off from the stack as shown in Figure 4.11.

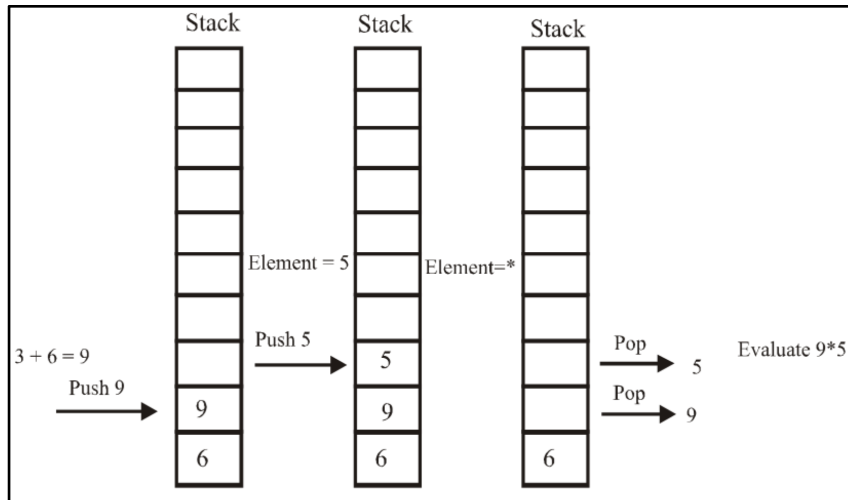


Figure 4.11 Popping 9 and 5 from Stack

8. The expression '9*5' is evaluated and the result, that is '45', is pushed back to the stack.
9. The next element in the postfix expression is '9', which is pushed onto the stack.
10. The next element is the operator '/'. Therefore, the two operands from the top of the stack, that is '9' and '45', are popped from the stack, and operation '45/9' is performed. Result '5' is again pushed to the stack.
11. The next element in the expression is '–'. Hence, '5' and '6' are popped from the stack, and operation '6-5' is performed. The resulting value, that is '1', is pushed to the stack (Figure 4.12).

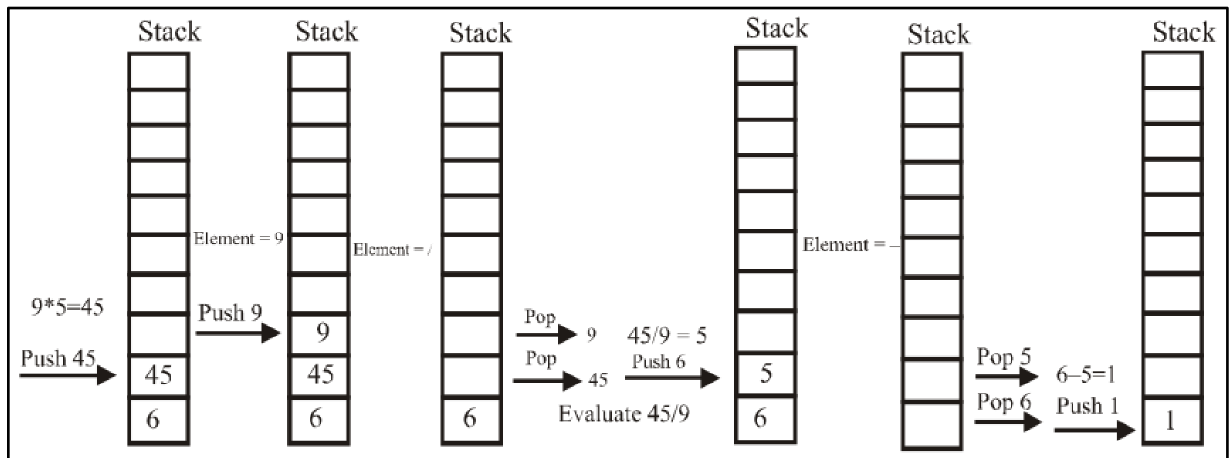


Figure 4.12 Final State of Stack with the Result

12. There are no more elements to be processed in the expression. The element on top of the stack is popped, which is the result of the evaluation of the postfix

expression. Thus, the result of the expression is ‘1’.

The step-wise evaluation of the expression 636+5*9/- is shown in Table 2.2.

Table 2.2 Evaluation of the Postfix Expression

Element	Action performed	Stack Status
6	Push to stack	6
3	Push to stack	6 3
6	Push to stack	6 3 6
+	Pop 6	6 3
	Pop 3	6
	Evaluate 3+6=9	6
	Push 9 to stack	6 9
5	Push to stack	6 9 5
*	Pop 5	6 9
	Pop 9	6
	Evaluate 9*5=45	6
	Push 45 to stack	6 45
9	Push to stack	6 45 9
/	Pop 9	6 45
	Pop 45	6
	Evaluate 45/9=5	6
	Push 5 to stack	6 5
-	Pop 5	6
	Pop 6	EMPTY
	Evaluate 6-5=1	EMPTY
	Push 1 to stack	1
	Pop VALUE=1	EMPTY

Program 4.9: A program to evaluate a postfix expression.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<math.h>
#define MAX 102
typedef struct stack
{
	float item[MAX];
	int Top;
}stk;
/*Function prototypes*/
void createstack(stk *);
float evaluationofpostfix(stk *, char *);
void push(stk *, float);
```

```
float pop(stk *);
void main()
{
        stk s;
        char postfix[MAX];
        int i;
        float result;
        clrscr();
        createstack(&s);
        do
        {
                printf("\nEnter expression in postfix notation (max %d characters): ", MAX-2);
                for(i=0;i<MAX-1;i++)
                {
                        scanf("%c", &postfix[i]);
                        if(postfix[i]=='\n')
                        break;
                }
        postfix[i]='\0';
}while(strlen(postfix)==0);
result=evaluationofpostfix(&s, postfix);
printf("\nThe result of postfix expression is %7.2f", result);
getch();
}
void createstack(stk *s)
{
        s->Top=-1;
}
float evaluationofpostfix(stk *s, char *po)
{
        int i, len;
        int number;
        float operand1, operand2;
        float res=0.0;
        len=strlen(po);
        i=0;
        while(i<len)
```

```
{
        if (po[i]==' '||po[i]=='\t' || po[i]==',')
        {
                i++;
                continue;
        }
        if(isdigit(po[i]))
        {
                number=(int)(po[i]-'0');
                i++;
                while (isdigit(po[i]))
                {
                        po[i]=(int)(po[i]-'0');            / * converting char to int*/
                        number=number*10;
                        number+=po[i];
                        i++;
                }
                push(s, number);
        }
        else
        {
                operand2=pop(s);
                operand1=pop(s);
                switch(po[i])
                {
                        case '+': res=operand1+operand2;
                                break;
                        case '-': res=operand1-operand2;
                                break;
                        case '*': res=operand1*operand2;
                                break;
                        case '/': res=(float)operand1/operand2;
                                break;
                        case'%': res=(int)operand1%(int)operand2;
                                break;
                        case '^': res=pow(operand1, operand2);
                                break;
```

```
                                        default: printf("\nIllegal expression...");
                                        getch();
                                        exit();
                              }
                              push(s, res);
                    }
                    i++;
          }
          res=pop(s);
          return res;
}
void push(stk *s, float item)
{
          s->Top++;
          s->item[s->Top]=item;
}
float pop(stk *s)
{
          float popped;
          popped=s->item[s->Top];
          s->Top—;
          return popped;
}
```

The output of the program is:

Enter expression in postfix notation (max 100 characters): 7 5 - 9 2 / *

The result of postfix expression is 9.00

4.5 Summary

- A stack is a linear data structure in which an element can be added or removed only at one end called the top of the stack.
- In stack terminology, the insert and delete operations are known as push and pop operations respectively. A stack works on the principle of 'last-in-first-out' and is also known as a Last-In-First-Out (LIFO) list.
- A stack can be represented in memory either as an array or as a singly linked

list. An array representation of a stack is static and linked list representation is dynamic in nature.

- When a function definition includes a call to itself, it is referred to as a recursive function and the process is known as recursion or circular definition.
- To reverse a string, the characters of the string are pushed onto the stack one by one as the string is read from left to right. Once all the characters of the string are pushed onto the stack, they are popped one by one.
- The notation in which an operator occurs before its operands are known as the prefix notation (also known as the Polish notation). On the other hand, the notation in which an operator occurs after its operands is known as the postfix notation (also known as the Reverse Polish or suffix notation).

4.6 Key Terms

- **Recursive function:** A function whose definition includes a call to itself.
- **Infix notation:** The general way of writing arithmetic expressions in which the binary operator is placed between two operands on which it operates.
- **Prefix notation:** The notation in which an operator occurs before its operands (also known as the Polish notation).
- **Postfix notation:** The notation in which an operator occurs after its operands (also known as the Reverse Polish or suffix notation).
- **Dynamic Allocation:** Automatic memory allocation where memory is allocated as required at run-time.

4.7 Check Your Progress

Short- Answer type

Q1) A stack can be represented as an array as well as a linked list. (True/False?)

Q2) When a function definition includes a call to itself, it is referred to as a ________.

Q3) The condition Top= –1 indicates that:

(a) Stack is empty (b) Stack is full (c) Stack has only one element

(d) None of the above

Q4) An infix expression can be converted into a postfix expression with the help of stacks. (True/False?)

Q5) Define String Reversal.

Long- Answer type

Q1) Write a C program to convert an infix expression into a postfix notation.

Q2) What are the two ways of implementing stacks? Which one is preferred over the other and why?

Q3) Write the algorithm to implement push and pop operations on a stack.

Q4) What are the various applications of stacks? Write a C program to implement any one of them.

Q5) Differentiate between infix, postfix, and prefix expressions.

References

- *Classic Data Structures,* Debasis Samanta, PHI Learning Pvt. Ltd. 2nd Edition.
- *Data Structures with C,* Lipschutz, S. (2011), Delhi: Tata McGraw-Hill.
- *Data Structures and Algorithms,* Aho, Ullman and Hopcroft, Addison Wesley (January 1983).

Unit 5 Queues

Structure

5.0 Introduction
5.1 Unit Objectives
5.2 Basic terminology of Queues
5.3 Queue Operations
5.4 Representation of a Queue
5.4.1 Using an array 5.4.2 Using a Linked List
5.5 Various Queue Structures
5.5.1 Circular Queue 5.5.2 Priority Queue
5.6 Summary
5.7 Key Terms
5.8 Check Your Progress

5.0 Introduction

A queue refers to a linear data structure in which a new element is inserted at one end and another element is deleted from the other end. The first element added to the queue is to be removed first. It means that the queue works on the principle of 'first-in-first-out'. This is why it is also known as a First-In-First-Out (FIFO) list. Queues, such as stacks, can be represented in memory by using an array or a singly linked list. You will learn about two types of queues: circular queue and priority queue. In a circular queue, as soon as the rear index of the queue reaches the maximum size of the array, the rear is reset to the beginning of the queue, provided it is free. A priority queue refers to a type of queue in which each element is assigned a priority and the elements are added or removed according to that priority.

A common example of a queue is people waiting in line at a bus stop. The first person in the queue enters the bus first. Any new person has to join at the end of the queue. In other words, the order in which people take the bus in the order in which they have joined the queue. The size of the queue is not fixed and it keeps varying as per the number of people joining and leaving the queue.

5.1 Unit Objectives

After going through this unit, the reader will be able to:

- Discuss the basic terminology of Queues.
- Representation of a Queue using an array and Linked List.
- Explain different types of Queue structures.

5.2 Basic terminology of Queues

An ordered collection of homogeneous data elements in which insertion and deletion operations take place at two extreme ends is called a queue. A queue is also a linear structure like an array, a linked list, and a stack but a queue is a first-in-first-out (FIFO) list. It means that the data in the queue is processed in the same order as it had entered. The process of inserting a data element into a queue is termed ENQUEUE and deletion is termed as DEQUEUE. Both the operations take place at the two ends of the queue called REAR and FRONT respectively. An element in the queue is termed as an ITEM. The number of elements a queue can hold is termed as the LENGTH of the queue. Figure 5.1 shows the schematic of a queue with REAR and FRONT ends.

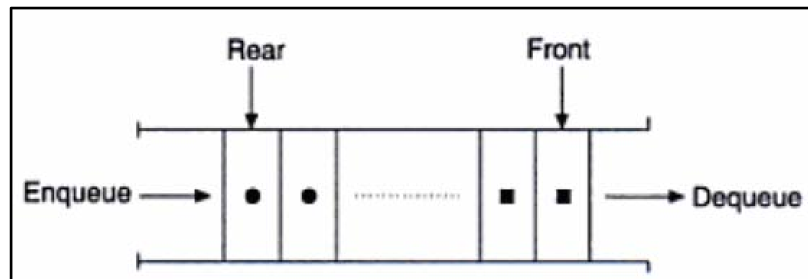


Figure 5.1 Schematic representation of a queue

Source- Classic Data Structures, Debasis Samanta, PHI Learning Pvt. Ltd. 2nd Edition

5.3 Queue Operations

Basically, there are two operations performed on a queue, insertion, and deletion. The insertion of any item at the **REAR** end of the queue is referred to as **ENQUEUE** and the deletion of any item at the **FRONT** end of the queue is referred to as

DEQUEUE.

The queue supports the following operations:

- *enqueue(obj):* Insert obj at the end of the queue, making it the last item.
- *dequeue():* Return the first object from the queue and remove it from the queue.

- *queue empty():* Test whether the queue is empty.

Insert at Rear-End (ENQUEUE)

For inserting an item into a queue, firstly we have to verify whether the queue is full or not. If the queue is full, then a new item can not be inserted. After verifying this condition, items can be inserted at the rear end of the queue. After insertion, the value of the rear is incremented by 1.

Delete from the Front End (DEQUEUE)

To delete an item from the queue, firstly we have to verify that the queue should not be empty. After verifying this condition that the queue is not empty, the items are deleted from the front end of the queue. After deleting an item, the value of the front is incremented by 1.

For example, figure 5.2 represents the basic operations of a queue. The first element to be inserted into the queue is 10, the second element to be inserted is 15 and so on. 30 is the last inserted element. Accordingly, the first element to be deleted from the queue is 10. If we have to add a new element, then it can be inserted after 30 and it will be at the last place in the queue. We can not insert a new element in the queue when the queue is full.

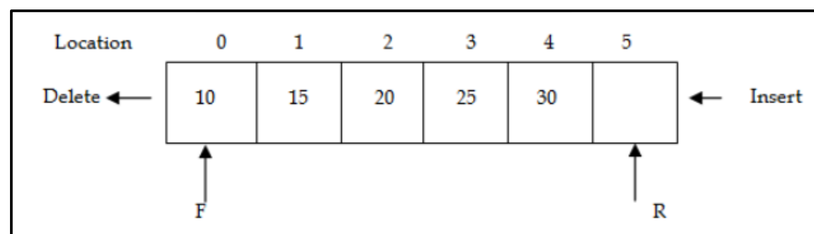


Figure 5.2 Example of basic operations in a queue

5.4 Representation of a Queue

Like stacks, queues can also be represented in memory by using an array or a singly linked list. An array representation of a queue is static. However, the linked list representation is dynamic in nature. Though array representation is a simple technique, it provides less flexibility and is not very efficient with respect to memory utilization. This is because an array reserves a fixed memory for the number of

elements to be stored in the queue. So, it could result in generating possible overflow errors. Let's discuss both the methods of representing a queue in detail.

5.4.1 Using an array

When a queue is implemented as an array, all the characteristics of an array are applicable to the queue. Since an array is a static data structure, the array representation of a queue requires the maximum size of the queue to be predetermined and fixed. As we know that the queue keeps on changing as the elements are inserted or deleted, the maximum size should be large enough for a queue to expand or shrink.

The representation of a queue as an array needs an array to hold the elements of the queue and two variables, Rear and Front, to keep track of the rear and the front ends of the queue respectively. Initially, the value of the Rear and Front is set to –1 to indicate an empty queue. Before we insert a new element in the queue, it is necessary to test the condition of overflow. The queue is in a condition of overflow (full) when Rear is equal to the MAX -1, where MAX is the maximum size of the array. If the queue is not full, the insert operation can be performed. To insert an element in the queue, the Rear is incremented by one, and the element is inserted at that position.

Similarly, before we delete an element from the queue, it is necessary to test the condition of the underflow. The queue is in the condition of underflow (empty) when the value of Front is –1. If the queue is not empty, a delete operation can be performed. To delete an element from the queue, the element referred to by the Front is assigned to a local variable, and then the Front is incremented by one.

The total number of elements in a queue at a given point of time can be calculated from the values of the Rear and Front as given here.

Number of elements = Rear – Front + 1

To understand the implementation of the queue as an array in detail, consider a queue stored in the memory as an array named Queue that has MAX as its maximum number of elements. Rear and Front store the indices of the rear and front elements of the Queue. Initially, the Rear and Front are set to –1 to indicate an empty queue (Figure 5.3 (a)).

Whenever a new element has to be inserted in the queue, the Rear is incremented by

one and the element is stored at Queue [Rear]. Suppose element 9 is to be inserted in the queue. In this case, the rear is incremented from –1 to 0, and the element is stored at Queue [0]. Since it is the first element to be inserted, the Front is also incremented by one to make it refer to the first element of the queue (Figure 5.3 (b)). For subsequent insertions, the value of Rear is incremented by one, and the element is stored at Queue [Rear]. However, the Front remains unchanged (Figure 5.3 (c)). Observe that the Front and Rear elements of the Queue are the first and the last elements of the list, respectively.

Whenever an element is to be deleted from the queue, the Front is incremented by one. Suppose that an element is to be deleted from the Queue. Then, here, it must be 9. It is because the deletion is always made at the front end of a queue. The deletion of the first element results in the queue as shown in Figure 5.3(d). Similarly, deletion of the second element results in a queue as shown in Figure 5.3(e). Observe that after deleting the second element from the queue, values of Front and Rear are equal. Here, it is apparent that when values of Front and Rear are equal other than –1, there is only one element in the queue. When this only element of the queue is deleted, both Rear and Front are again made equal to –1 to indicate an empty queue.

Further, suppose that some more elements are inserted and the Rear reaches the maximum size of the array (Figure 5.3 (f)). That means Queue is full and no more elements can be inserted in Queue even though the space is vacant on the left of the Front. This problem can be resolved using circular queues.

To implement a queue as an array in C language, the following structure named queue is used.

```
struct queue
{
        int item[MAX];
        int Front;
        int Rear;
};
```

Algorithm 5.1 Insert Operation on Queue

qinsert(q, val))   //q is a pointer to structure type queue and val is the value to be inserted
1. If q->Rear = MAX-1   //check if queue is full
Print “Overflow: Queue is full!” and go to step 5
End If
2. If q->Front = -1   //check if queue is empty

```
    Set q->Front = 0          //make front to refer to first element
  End If
3. Set q->Rear = q->Rear + 1       //increment Rear by one
4. Set q->item[q->Rear] = val        //insert val
5. End
```

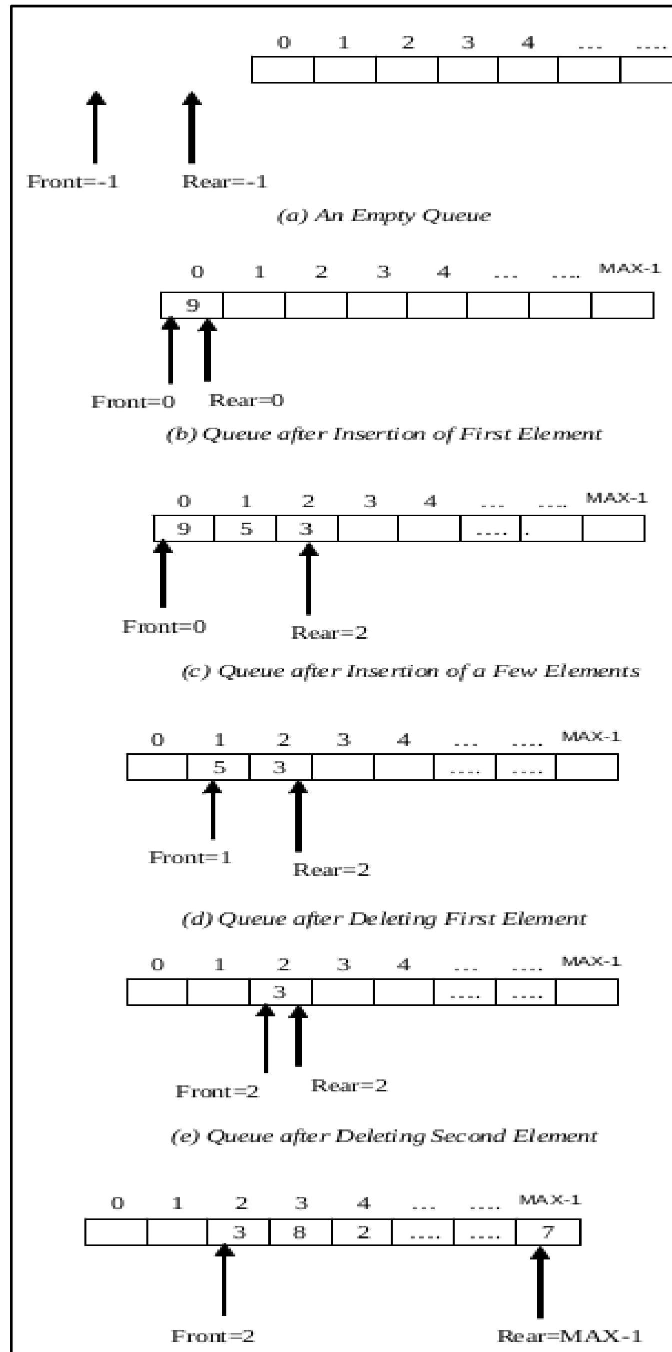


Figure 5.3 Various States of a Queue after Insert and Delete Operations

Algorithm 5.2 Delete Operation on Queue
qdelete(q)
1. If q->Front = -1 //check if queue is empty
Print "Underflow: Queue is empty!"
Return 0 and go to step 5
End If
2. Set del_val = q->item[q->Front] //del_val is the value to be deleted
3. If q->Front = q->Rear //check if there is only one element
Set q->Front = q->Rear = -1
Else
Set q->Front = q->Front + 1 //increment Front by one
End If
4. Return del_val
5. End

Program 5.1: A program to implement a queue as an array.

```
/* Function prototypes */
void createqueue(que *);
void qinsert(que *,int);
int qdelete(que *);
int isempty(que);
int isfull(que);
void main()
{
        int choice,val,element;
        que q;
        createqueue(&q);
        do
        {
                clrscr();
                printf("\n\t Main Menu");
                printf("\n1. Insert");
                printf("\n2. Delete");
                printf("\n3. Exit\n");
                printf("\nEnter your choice: ");
                scanf("%d", &choice);
                switch(choice)
```

```
        {
            case 1: printf("\nEnter the value to be inserted: ");
                scanf("%d", &element);
                qinsert(&q, element);
                getch();
                break;
            case 2: val=qdelete(&q);
                if(val==0) #include<stdio.h>
    #include<conio.h>
    #define MAX 4
    #define True 1
    #define False 0
    typedef struct queue
    {
        int item[MAX];
        int Front;
        int Rear;
        }que;
            printf("\nUnderflow: Queue is empty!");
        else
            printf("\nDeleted item is: %d\n", val);
        getch();
        break;
    case 3: exit();
        default: printf("Invalid choice");
    }
    } while(1);
}
void qinsert(que *q, int val)
{
    if(isfull(*q))
    {
        printf("\nOverflow: Queue is full!");
        return;
    }
    if(isempty(*q))
    q->Front=0;
```

```
(q->Rear)++;
q->item[q->Rear]=val;
printf("\nValue is inserted in queue...");
}
int qdelete(que *q)
{
        int del_val;
        if(isempty(*q))
                return 0;
                del_val=q->item[q->Front];
        if(q->Front==q->Rear)
                q->Front=q->Rear=-1;
        else
                (q->Front)++;
                return del_val;
}
void createqueue(que *q)
{
        q->Front=q->Rear=-1;
        return;
}
int isempty(que q)
{
        if(q.Front==-1)
                return True;
        else
                return False;
}
int isfull(que q)
{
        if(q.Rear==MAX-1)
                return True;
        else
                return False;
}
```

The output of the program is:

```
          Main Menu
1. Insert
2. Delete
3. Exit
Enter your choice: 2
Underflow: Queue is empty!
          Main Menu
1. Insert
2. Delete
3. Exit
Enter your choice: 1
Enter the value to be inserted: 3
Value is inserted in queue...
          Main Menu
1. Insert
2. Delete
3. Exit
Enter your choice: 1
Enter the value to be inserted: 5
Value is inserted in queue...
          Main Menu
1. Insert
2. Delete
3. Exit
Enter your choice: 2
Deleted item is: 3
          Main Menu
1. Insert
2. Delete
3. Exit
Enter your choice: 3
```

5.4.2 Using a Linked List

A queue implemented as a linked list is known as a linked queue. A linked queue is represented using two pointer variables Front and Rear that point to the first and the last node of the queue, respectively. Initially, the Rear and Front are set to NULL to indicate an empty queue.

To understand the implementation of a linked queue, consider a linked queue, say, Queue. The info and next field of each node represent the element of the queue and a pointer to the next element in the queue, respectively. Whenever a new element is to be inserted in the queue, a new node nptr is created and the element is inserted into the node. If it is the first element being inserted in the queue, both Front and Rear are modified to point to this new node. On the other hand, in subsequent insertions, only the Rear is modified to point to the new node, and the Front remains unchanged.

Whenever an element is deleted from the Queue, a temporary pointer is created which is made to point to the node pointed to by Front. Then Front is modified to point to the next node in the Queue, and the temporary node is deleted from the memory. Figure 5.4 shows various states of the queue after insert and delete operations.

Note: Since the memory is allocated dynamically, a linked queue reaches the overflow condition when no more memory space is available to be dynamically allocated.

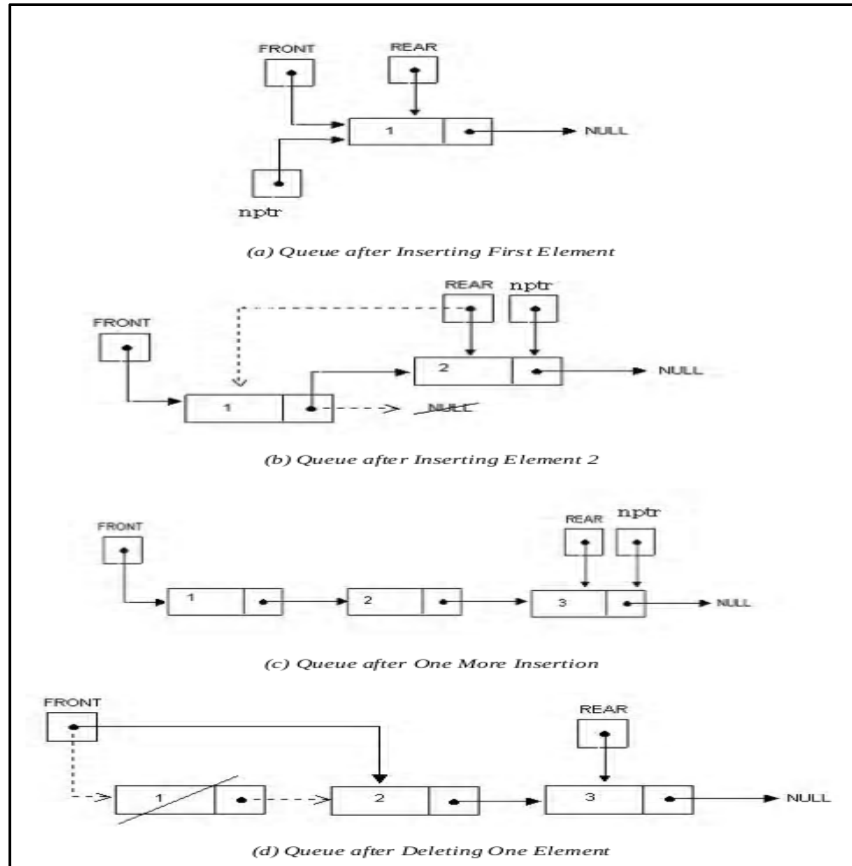


Figure 5.3 Various States of Linked queue after Insert and Delete Operations

Algorithm 5.3 Insert Operation on Linked Queue

qinsert(q, element)
1. Allocate memory for nptr //nptr is a pointer to the new node to be inserted
2. If nptr = NULL //checking for queue overflow
Print “Overflow: Memory not allocated!” and go to step 6
End If
3. Set nptr->info = element
4. Set nptr->next = NULL
5. If Front = NULL //check if queue is empty
*Set (*q)->Rear = (*q)->Front = nptr //rear and front are made to point to new node*
Else
*Set (*q)->Rear->next = nptr*
*Set (*q)->Rear = nptr //rear is made to point to new node*
End If
6. End

Algorithm 5.4 Delete Operation on Linked Queue

qdelete(q)
1. If Front = NULL
    Print "Underflow: Queue is empty!"
    Return 0 and go to step 7
  End if
*2. Set del_val = (*q)->Front->info   //del_val is the element pointed by the Front*
*3. Set temp = (*q)->Front   //temp is the temporary pointer to Front*
*4. If (*q)->Front = (*q)->Rear   //checking if there is one element in the queue*
    *Set (*q)->Front = (*q)->Rear = NULL*
  Else
    *Set (*q)->Front = ((*q)->Front)->next   //making Front point to next node*
    End If
5. Deallocate temp   //deallocating memory
6. Return del_val
7. End

Program 5.2: A program to illustrate the implementation of a queue as linked list.

```
#include<stdio.h>
#include<conio.h>
#define True 1
#define False 0
typedef struct node
{
        int info;
        struct node *next;
}Node;
typedef struct queue
{
        Node *Front;
        Node *Rear;
}que;
void createqueue(que **);
int isempty(que *);
void qinsert(que **, int);
```

```
int qdelete(que **);
void main()
{
	que q;
	int choice,val,element;
	createqueue(&q);
	do
	{
		clrscr();
		printf("\n\tMain Menu");
		printf("\n1. Insert");
		printf("\n2. Delete");
		printf("\n3. Exit\n");
		printf("\nEnter your choice: ");
		scanf("%d", &choice);
		switch(choice)
		{
			case 1: printf("\nEnter the value to be inserted: ");
				scanf("%d", &element);
				qinsert(&q,element);
				getch();
				break;
			case 2: val=qdelete(&q);
				if(val==0)
					printf("\nUnderflow: Queue is empty!");
				else
					printf("\nDeleted item is: %d", val);
				getch();
				break;
			case 3: exit();
			default: printf("Invalid choice!");
		}
	}while(1);
}
void createqueue(que **q)
{
	(*q)->Front=NULL;
```

```
        (*q)->Rear=NULL;
}
int isempty(que *q)
{
        if(q->Front==NULL)
                return True;
        else
                return False;
}
void qinsert(que **q,int element)
{
        Node *nptr;
        nptr=(Node*)malloc(sizeof(Node));
        if(nptr==NULL)
        {
                printf("\nOverflow: Memory not allocated!");
                return;
        }
nptr->info=element;
nptr->next=NULL;
if((*q)->Front==NULL)
        (*q)->Rear=(*q)->Front=nptr;
else
        {
                ((*q)->Rear)->next=nptr;
                (*q)->Rear=nptr;
        }
        printf("\nValue is inserted in the queue... ");
}
int qdelete(que **q)
{
        int del_val;
        Node *temp;
        if(isempty((*q)->Front))
                return 0;
        del_val=((*q)->Front)->info;
        temp=(*q)->Front;
```

```
        if((*q)->Front==(*q)->Rear)
                (*q)->Front=(*q)->Rear=NULL;
        else
                (*q)->Front=((*q)->Front)->next;
free(temp);
return del_val;
}
```

The output of the program is:

```
        Main Menu
1. Insert
2. Delete
3. Exit
Enter your choice: 1
Enter the value to be inserted: 3
Value is inserted in the queue…
        Main Menu
1. Insert
2. Delete
3. Exit
Enter your choice: 1
Enter the value to be inserted: 5
Value is inserted in the queue…
        Main Menu
1. Insert
2. Delete
3. Exit
Enter your choice: 2
Deleted item is: 3
        Main Menu
1. Insert
2. Delete
3. Exit
Enter your choice: 2
```

Deleted item is: 5

Main Menu

1. Insert
2. Delete
3. Exit

Enter your choice: 2

Underflow: Queue is empty!

Main Menu

1. Insert
2. Delete
3. Exit

Enter your choice: 3

5.5 Various Queue Structures

Apart from representing a queue using an array and a linked list, there are various queue structures used as per their requirement. These structures are Circular and priority queues. Let's discuss these structures in detail.

5.5.1 Circular Queue

As discussed earlier, in the case of a queue represented as an array, once the value of the rear reaches the maximum size of the queue, no more elements can be inserted. However, there may be the possibility that space on the left of the front index is vacant. Hence, in spite of the space on the left of the front is empty, the queue is considered to be full. This wastage of space in the array implementation of a queue can be avoided by shifting the elements to the beginning of the array if the space is available. In order to do this, the values of Rear and Front indices have to be changed accordingly. However, this is a complex process and is difficult to be implemented. An alternative solution to this problem is to implement a queue as a circular queue.

The array implementation of a circular queue is similar to the array implementation of a general queue. The only difference is that in a circular queue, as soon as the rear index of the queue reaches the maximum size of the array, the Rear is reset to the beginning of the queue provided it is free. The circular queue is full only when all the

locations in the array are occupied. The circular queue is shown in Figure 5.4.

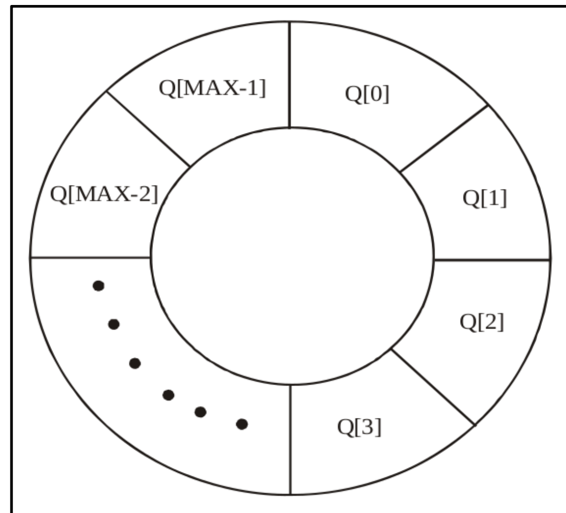


Figure 5.4 A circular queue

Note: A circular queue is generally implemented as an array, though it can also be implemented as a circular linked list.

To understand the operations on a circular queue, consider a circular queue represented in the memory by the array **CQueue[MAX]**. Rear and Front are used to store the indices of the rear and front elements of CQueue, respectively. Initially, both Rear and Front are set to –1 to indicate an empty queue.

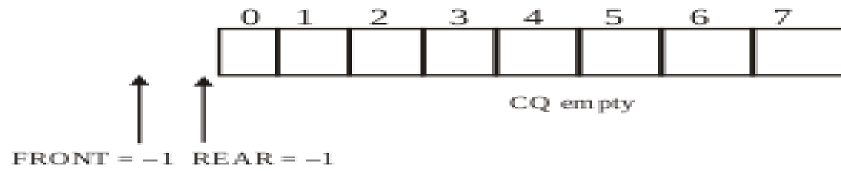
Whenever an element is to be inserted in the circular queue, the Rear is incremented by one. However, if the value of the Rear index is MAX-1, instead of incrementing Rear, it is reset to the first index of the array if space is available in the beginning. Hence, if any locations to the left of the Front index are empty, the elements can be added to the queue at an index starting from 0.

The queue is considered full in the following cases.

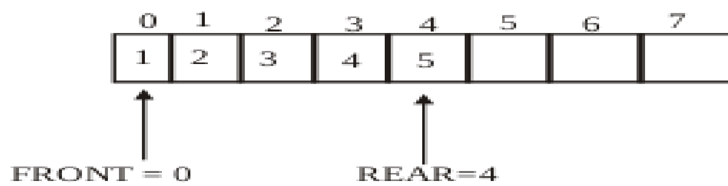
- When the value of Rear equals the maximum size of the array and Front is at the beginning of the array.
- When the value of the Front is one more than the value of the Rear.

Whenever an element is to be deleted from the queue, the Front is incremented by one. However, if the value of Front is MAX-1, it is reset to the 0th position in the array. When the value of the Front is equal to the value of the Rear (other than –1), it

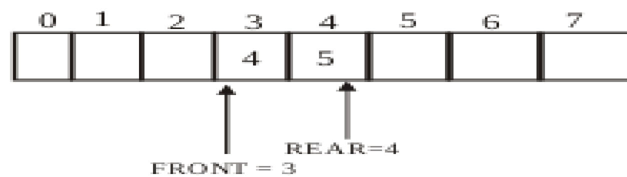
indicates that there is only one element in the queue. On deleting the last element, both Rear and Front are reset to –1 to indicate an empty queue. Figure 5.5 shows various states of the queue after some insert and delete operations.



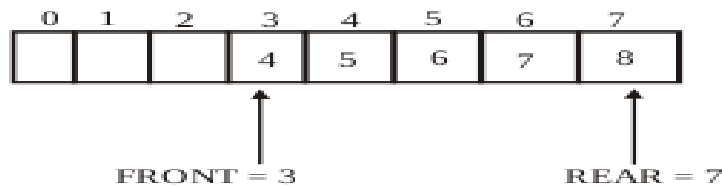
(a) Empty Queue



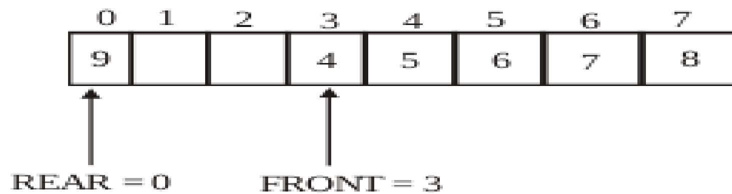
(b) Queue after inserting Few Elements



(c) Queue after deleting Few Elements



(d) Queue when Rear=MAX-1



(e) Rear is Reset to Zero

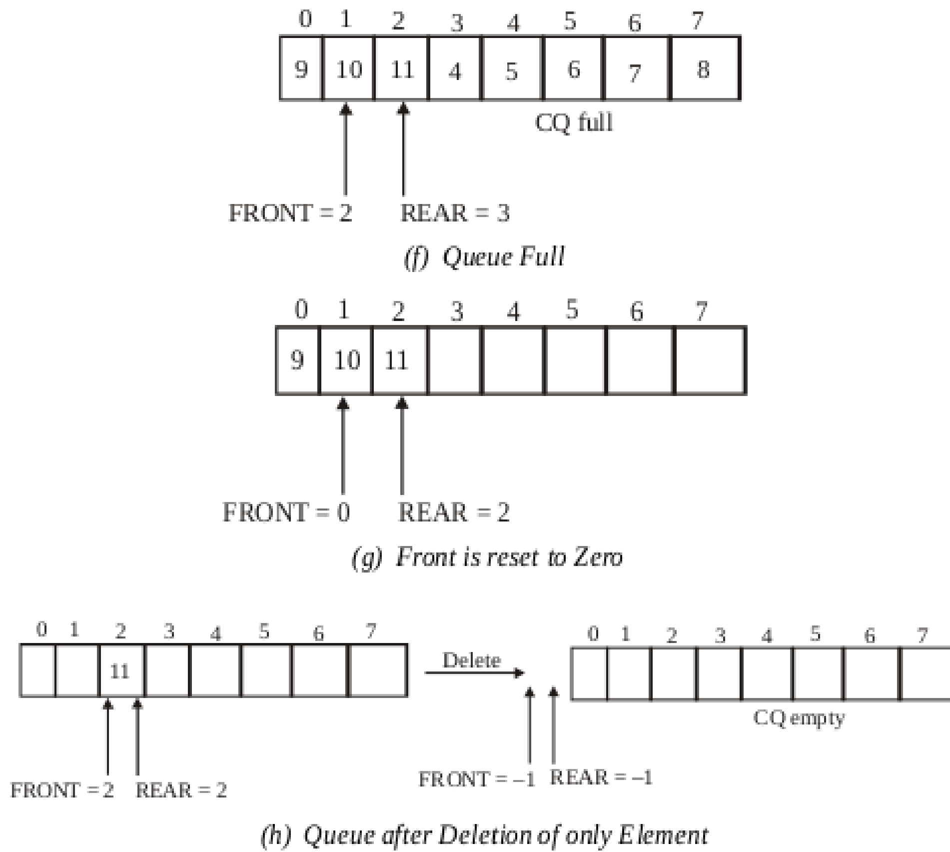


Figure 5.5 Various States of a Circular Queue after Insert and Delete Operations

The total number of elements in a circular queue at any point in time can be calculated from the current values of the Rear and the Front indices of the queue.

- In case, **Front<Rear**, the **total number of elements = Rear-Front+1**. For instance, in Figure 5.6(a), Front=3 and Rear=7. Hence, the total number of elements in CQueue at this point in time is 7–3+1=5.
- In case, **Front>Rear**, the **total number of elements = Max +(Rear-Front)+1**. For instance, in Figure 5.6(b), Front=3 and Rear=0. Hence, the total number of elements in CQueue is 8+(0–3)+1.

Figure 5.6 Number of Elements in a Circular Queue

Algorithm 5.5 Insert Operation on Circular Queue

```
qinsert(q, val)
1. If ((q->Rear = MAX-1 AND q->Front = 0) OR (q->Rear + 1 = q->Front))
       Print "Overflow: Queue is full!" and go to step 5
   End If                                    //check if circular queue is full
2. If q->Rear = MAX-1                        //check if rear is MAX-1
       Set q->Rear = 0
   Else
       Set q->Rear = q->Rear + 1             //increment rear by one
End If
3. Set q->CQueue[q->Rear] = val              //val is the value to be inserted in the queue
4. If q->Front = -1                          //check if queue is empty
       Set q->Front = 0
   End If
5. End
```

Algorithm 5.6 Delete Operation on Circular Queue

```
qdelete(q)
1. If q->Front = -1
      Print "Underflow: Queue is empty!"
      Return 0 and go to step 5
   End If
2. Set del_val = q->CQueue[q->Front]        //del_val is the value to be deleted
3. If q->Front = q->Rear                    //check if there is one element in the queue
      Set q->Front = q->Rear = -1
   Else
      If q->Front = MAX-1
         Set q->Front = 0
      Else
         Set q->Front = q->Front +1
      End If
   End If
4. Return del_val
5. End
```

Program 5.3: A program to implement a circular queue.

```
#include<stdio.h>
#include<conio.h>
#define MAX 4
#define True 1
#define False 0
typedef struct queue
{
        int CQueue[MAX];
        int Front;
        int Rear;
}que;
/* Function prototypes */
void createqueue(que *);
void qinsert(que *,int);
int qdelete(que *);
void qdisplay(que);
```

```
int isempty(que);
int isfull(que);
void main()
{
	que q;
	int choice,element,val;
	createqueue(&q);
	do
	{
		clrscr();
		printf("\n\n\tMain Menu");
		printf("\n1. Insert");
		printf("\n2. Delete");
		printf("\n3. Exit\n");
		printf("\nEnter your choice: ");
		scanf("%d", &choice);
		switch(choice)
		{
			case 1: printf("\nEnter the value to be inserted: ");
				scanf("%d", &element);
				qinsert(&q, element);
				getch();
				break;
			case 2: val=qdelete(&q);
				if(val==0)
					printf("\nUnderflow: Queue is empty!");
				else
					printf("\nThe value of deleted item is: %d\n", val);
				getch();
				break;
			case 3: exit();
			default: printf("Invalid choice");
		}
	} while(1);
}
void createqueue(que *q)
{
```

```
        q->Front=q->Rear=-1;
}
void qinsert(que *q, int val)
{
        if(isfull(*q))
        {
                printf("\nOverflow: Queue is full!");
                return;
        }
        if(q->Rear==MAX-1)
                q->Rear=0;
        else
                (q->Rear)++;
        q->CQueue[q->Rear]=val;
        if(isempty(*q))
                q->Front=0;
        qdisplay(*q);
}
int qdelete(que *q)
{
        int del_val;
        if(isempty(*q))
        return 0;
del_val=q->CQueue[q->Front];
if(q->Front==q->Rear)
        q->Front=q->Rear=-1;
else
{
        if(q->Front==MAX-1)
                q->Front=0;
        else
                (q->Front)++;
}
        return del_val;
}
void qdisplay(que q)
{
```

```
        int i;
        printf("\nFront: %d, Rear: %d", q.Front, q.Rear);
        printf("\n\nQueue is: ");
        if(q.Front<=q.Rear)
                for(i=q.Front; i<=q.Rear; i++)
                        printf("%d ", q.CQueue[i]);
        else
        {
                for(i=0; i<=q.Rear; i++)
                        printf("%d ", q.CQueue[i]);
                for(i=q.Front; i<MAX; i++)
                        printf("%d ", q.CQueue[i]);
        }
}
int isempty(que q)
{
        if(q.Front==-1)
                return True;
        else
                return False;
}
int isfull(que q)
{
        if((q.Rear==MAX-1 && q.Front==0)||(q.Rear+1==q.Front))
                return True;
        else
                return False;
}
```

The output of the program is:

```
        Main Menu
1. Insert
2. Delete
3. Exit
Enter your choice: 1
Enter the value to be inserted: 1
```

Front: 0, Rear: 0

Queue is: 1

Main Menu

1. Insert

2. Delete

3. Exit

Enter your choice: 1

Enter the value to be inserted: 2

Front: 0, Rear: 1

Queue is: 1 2

Main Menu

1. Insert

2. Delete

3. Exit

Enter your choice: 1

Enter the value to be inserted: 3

Front: 0, Rear: 2

Queue is: 1 2 3

Main Menu

1. Insert

2. Delete

3. Exit

Enter your choice: 2

Deleted item is: 1

Main Menu

1. Insert

2. Delete

3. Exit

Enter your choice: 1

Enter the value to be inserted: 7

Front: 1, Rear: 3

Queue is: 2 3 7

Main Menu

1. Insert
2. Delete
3. Exit

Enter your choice: 1

Enter the value to be inserted: 4

Front: 1, Rear: 0

Queue is: 4 2 3 7

5.5.2 Priority Queue

A priority queue is a type of queue in which each element is assigned a priority and the elements are added or removed according to that priority. While implementing a priority queue, the following two rules are applied.

- The element with higher priority is processed before any element of lower priority.
- The elements with the same priority are processed according to the order in which they were added to the queue.

A priority queue can be represented in many ways. Here, we are discussing the implementation of the priority queue using multiple queues.

Multiple Queue Implementation

In multiple queue representation of the priority queue, a separate queue for each priority is maintained. Each queue is implemented as a circular array and has its own two variables, Front and Rear (Figure 5.7). The element with the given priority number is inserted in the corresponding queue. Similarly, whenever an element is to be deleted from the queue, it must be the element from the highest priority queue. Note that the lower priority number indicates higher priority.

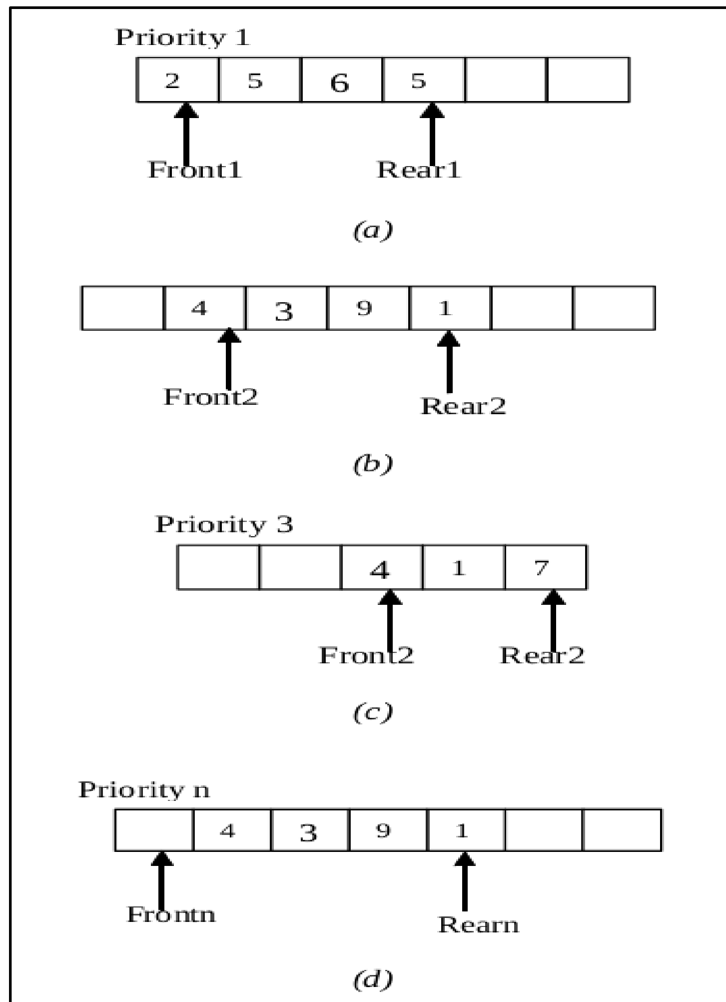


Figure 5.7 Queue according to Priority

If the size of each queue is the same, then instead of multiple one-dimensional arrays, a single two-dimensional array can be used where the row number shows the priority and the column number shows the position of the element within the queue. In addition, two arrays to keep track of the front and rear positions of each queue corresponding to each row are maintained (Figure 5.8).

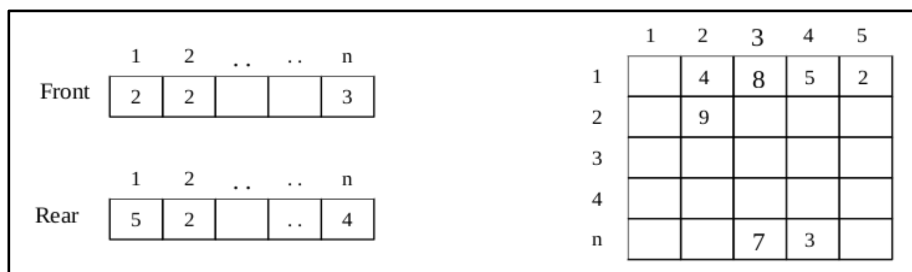


Figure 5.8 Priority Queue as a Two-Dimensional Array

Algorithm 5.7 Insert Operation in the Priority Queue

qinsert(q, val, prno)   //prno is the priority of val
1. If (q->Rear[prno] = MAX-1 AND q->Front[prno] = 0) OR (q->Rear[prno]+1 = q->Front[prno])   Print "Overflow: Queue full!" and go to step 5
End If
2. If q->Rear[prno-1] = MAX-1
Set q->Rear[prno-1] = 0
Else
Set q->Rear[prno-1] = q->Rear[prno-1] + 1
End If
3. Set q->CQueue[prno-1][q->Rear[prno-1]] = val
4. If q->Front[prno-1] = -1
Set q->Front[prno-1] = 0
End If
5. End

Algorithm 5.8 Delete Operation in the Priority Queue

qdelete(q)
1. Set flag = 0, i = 0
2. While i <= MAX-1
If NOT (q->Front[prno]) = -1   //check if not empty
Set flag = 1
Set del_val = q->CQueue[i][q->Front[i]]
If q->Front[i] = q->Rear[i]
Set q->Front[i] = q->Rear[i] = -1
Else If q->Front[i] = MAX-1
Set q->Front[i] = 0
Else
Set q->Front[i] = q->Front[i] + 1
End If
End If
break
End If
Set i = i +1
End While
3. If flag = 0

```
      Return 0 and go to step 4
   Else
      Return del_val
   End If
4. End
```

Program 5.4: A program to implement priority queue using multiple queues.

```
#include<stdio.h>
#include<conio.h>
#define MAX 5
#define True 1
#define False 0
typedef struct queue
{
        int CQueue[MAX][MAX];
        int Front[MAX];
        int Rear[MAX];
}que;
void createqueue(que *);
void qinsert(que *, int, int);
int qdelete(que *);
void qdisplay(que, int);
int isempty(que, int);
int isfull(que, int);
void main()
{
        que q;
        int choice,element,pno,val;
        createqueue(&q);
        do
        {
                clrscr();
                printf("\n\n\tMain Menu");
                printf("\n1. Insert");
                printf("\n2. Delete");
                printf("\n3. Exit\n");
                printf("\nEnter your choice: ");
```

```
                scanf("%d", &choice);
                switch(choice)
                {
                        case 1: printf("\nEnter the value and its priority: ");
                                scanf("%d%d", &element, &pno);
                                qinsert(&q, element, pno);
                                getch();
                                break;
                        case 2: val=qdelete(&q);
                                if(val==0)
                                        printf("\nUnderflow: Queue is empty!");
                                else
                                        printf("\nThe Deleted item is: %d\n",val);
                                getch();
                                break;
                        case 3: exit();
                        default: printf("Invalid choice");
                }
        } while(1);
}
void createqueue(que *q)
{
        int i;
        for(i=0;i<MAX;i++)
        q->Front[i]=q->Rear[i]=-1;
}
int isempty(que q, int prno)
{
        if(q.Front[prno]==-1)
                return True;
        else
                return False;
}
int isfull(que q, int prno)
{
        if((q.Rear[prno]==MAX-1 &&
        q.Front[prno]==0) || (q.Rear[prno]+1==q.Front[prno]))
```

```
                return True;
        else
                return False;
}
void qinsert(que *q,int val,int prno)
{
        int j;
        if(isfull(*q, prno))
        {
                printf("\nOverflow: Queue is full!");
                return;
        }
if(q->Rear[prno-1]==MAX-1)
        q->Rear[prno-1]=0;
else
        (q->Rear[prno-1])++;
q->CQueue[prno-1][q->Rear[prno-1]]=val;
if(isempty(*q, prno))
        q->Front[prno-1]=0;
qdisplay(*q, prno);
}
int qdelete(que *q)
{
        int del_val, i, prno,flag=0;
        for(i=0;i<= MAX-1;i++)
        {
                if(!isempty(*q,i))
                {
                        flag=1;
                        del_val=q->CQueue[i][q->Front[i]];
                        if(q->Front[i]==q->Rear[i])
                                q->Front[i]=q->Rear[i]=-1;
                        else if(q->Front[i]==MAX-1)
                                q->Front[i]=0;
                        else
                                q->Front[i]++;
                        prno =i+1;
```

```
                break;
            }
        }
        if(flag==0)
            return 0;
        else
        {
            printf("\nPriority of deleted item is: %d\n",prno);
            return del_val;
        }
}
void qdisplay(que q, int prno)
{
        int i;
        printf("\nFront: %d, Rear: %d", q.Front[prno-1], q.Rear[prno-1]);
        printf("\n\nQueue for prno %d is: ", prno);
        if(q.Front[prno-1]<=q.Rear[prno-1])
        {
            for(i=q.Front[prno-1];
                i<=q.Rear[prno-1]; i++)
            printf("%d ",q.CQueue[prno-1][i]);
        }
else
{
        for(i=0; i<=q.Rear[prno-1]; i++)
            printf("%d ", q.CQueue[prno-1][i]);
        for(i=q.Front[prno-1]; i<MAX; i++)
            printf("%d ", q.CQueue[prno-1][i]);
        }
}
```

The output of the program is:

Main Menu

1. Insert
2. Delete
3. Exit

Enter your choice: 2

Underflow: Queue is empty!

Main Menu

1. Insert

2. Delete

3. Exit

Enter your choice: 1

Enter the value and its priority: 8 3

Front: 0, Rear: 0

Queue for prno 3 is: 8

Main Menu

1. Insert

2. Delete

3. Exit

Enter your choice: 1

Enter the value and its priority: 9 4

Front: 0, Rear: 0

Queue for prno 4 is: 9

Main Menu

1. Insert

2. Delete

3. Exit

Enter your choice: 2

The priority of deleted item is: 3

The Deleted item is: 8

Main Menu

1. Insert

2. Delete

3. Exit

Enter your choice: 3

5.6 Summary

- A queue refers to a linear data structure in which a new element is inserted at

one end and the other element is deleted from the other end. It works on the principle of 'first-in-first-out' (FIFO).

- Like stacks, queues can also be represented in memory by using an array or a singly-linked list.
- Before we insert a new element in the queue, it is necessary to test the condition of overflow. Similarly, before we remove an item from the queue, it is necessary to test the condition of the underflow.
- There are two types of queue structures: Circular queue and priority queue.
- In a circular queue, as we go on adding elements to the queue and reach the end of the array, the next element is stored in the first position of the array (if it is free).
- A priority queue is a data structure in which each element is assigned a priority and the elements are added or removed according to that priority.

5.7 Key Terms

- **Queue:** A linear data structure in which a new element is inserted at one end and the other element is deleted from the other end.
- **Circular queue:** A data structure in which on adding elements to the queue and reaching the end of the array, the next element is stored in the first position of the array, if it is free.
- **Priority queue:** A data structure in which each element is assigned a priority and the elements are added or removed according to that priority.
- **Dequeue:** Process of deleting elements from the queue.
- **Enqueue:** Process of inserting elements into a queue.

5.8 Check Your Progress

Short- Answer type

Q1) A queue can be represented as an array as well as a linked list. (True/False?)

Q2) For a queue implemented as an array, the initial values of the front and rear set to _______.

Q3) A queue is a:

(a) linear data structure (b) non-linear data structure (c) Both (a) and (b)

(d) None of the above

Q4) Which two rules are followed while implementing a priority queue?

Q5) A ______ is a data structure in which each element is assigned a priority and the elements are added or removed according to that priority.

Long- Answer type

Q1) Write a short note on multiple queues implementation.

Q2) Write an algorithm to insert an element in a circular queue.

Q3) Differentiate between a circular queue and a priority queue.

Q4) How can a queue be implemented using an array and a linked list? Explain.

Q5) Write an algorithm to delete an element from a priority queue.

References

- *Classic Data Structures,* Debasis Samanta, PHI Learning Pvt. Ltd. 2nd Edition.
- *Data Structures with C,* Lipschutz, S. (2011), Delhi: Tata McGraw-Hill.

Unit 6 Trees

Structure

6.0 Introduction
6.1 Unit Objectives
6.2 Basic terminology of Trees
6.3 Binary Trees
6.4 Representation of a Binary Tree
    6.4.1 Array Representation
    6.4.2 Linked Representation
6.5 Binary Tree Traversals
6.6 Binary Search Tree
6.7 Threaded Binary Tree
6.8 Summary
6.9 Key Terms
6.10 Check Your Progress

6.0 Introduction

A tree is a widely used non-linear data structure. It resembles a hierarchical tree structure possessing a set of nodes that are linked to one another. Each node of a tree store a data value and has zero or more pointers pointing to the other nodes of the tree which are also known as its child nodes. A binary tree refers to a special type of tree that can be either empty or has a finite set of nodes. Binary trees are primarily of two types: complete binary tree and extended binary tree. This unit explains the various modes of binary tree representation, such as array representation and linked representation.

This unit will also introduce you to the binary search tree and threaded binary tree. A binary search tree, also known as a binary sorted tree, is a kind of a binary tree in which the data value in each node is a key (unique) value, i.e., no two nodes can have identical values. The structure of the node of a threaded binary tree is similar to the node of a binary tree, with some additional variables indicating whether the left or right pointers are normal pointers or threads.

6.1 Unit Objectives

After going through this unit, the reader will be able to:

- Understand the definition and basic concepts of trees.
- Explain a binary tree and the various terminologies associated with it.
- Discuss the various forms and representations of binary trees.
- Discuss the various operations on a binary tree.

6.2 Basic Terminology of Trees

A tree is a non-linear data structure representing a hierarchical structure of one or more elements known as ***nodes***. Each node of a tree store a data value and has zero or more pointers pointing to the other nodes of the tree, which are also known as its ***child nodes***. Each node in a tree can have zero or more child nodes located at one level below it. However, each child node can have only one parent node which is at one level above it. The node at the top of the tree is known as the ***root*** of the tree and the nodes at the lowest level are known as the ***leaf nodes***. The root node is a special node having no parent node and leaf nodes are nodes having no child nodes. Any node having a child node as well as parent node is known as an ***internal node***.

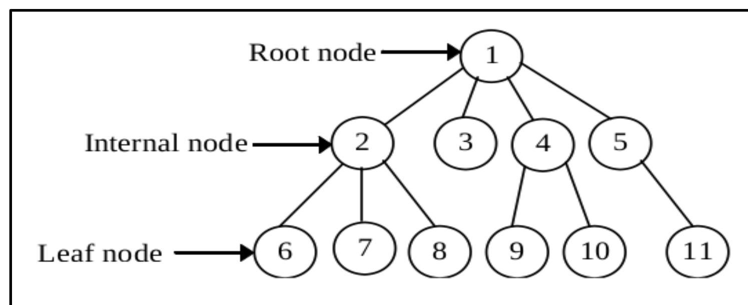


Figure 6.1 Structure of a Tree

Trees have the advantage of handling a lot of data together. The operations of insertion, deletion, sorting, etc. are more efficient in trees than in linear data structures like stacks, queues, and linked lists.

Some of the important terms regarding trees are discussed below.

- **Node:** A node is the main component of a tree. It stores actual data and links it to the other node.
- **Parent:** The parent of a node is the immediate predecessor of that node. Here in

figure 6.1, 1 is the parent of 2, 3, 4, and 5.

- **Child:** All the immediate successors of the parent node are known as Child. The child on the left side is called the *left child* and that on the right side is known as the *right child*.
- **Link:** A pointer to a node in a tree is called a link. There may be more than two links of a node.
- **Root:** The first node of a tree is called the root. A root does not have any parent.
- **Leaf:** The end node which does not have any child is known as a leaf. It is also termed a terminal node.
- **Level:** The ranking of the hierarchy of the tree is known as level. The root level is marked as 0. If a node is at level l, then its child is at level l+1 and its parent is at level l-1.
- **Height:** The maximum number of nodes possible from the root node to a leaf node is termed as the height of a tree.
- **Degree:** The maximum number of children that is possible for a node is known as the degree of a node.
- **Sibling:** The nodes with the same parent nodes are called siblings.

6.3 Binary Tree and its properties

A binary tree is a special type of tree, which can be either empty or has a finite set of nodes, such that one of the nodes is designated as the root node and the remaining nodes are partitioned into two subtrees of root node known as left subtree and right subtree. The non-empty left subtree and the right subtree are also binary trees. Unlike a general tree, each node in a binary tree is restricted to have at most two child nodes. Consider a sample binary tree T shown in Figure 6.2.

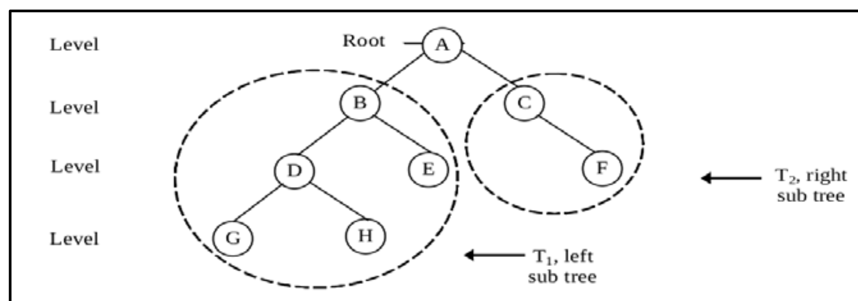


Figure 6.2 A Binary Tree

In this figure, the topmost node A is the root node of the tree T. Each node in this tree has zero or utmost two child nodes. The nodes A, B, and D have two child nodes, node C has only one child node, and nodes G, H, E and F are leaf nodes having no child nodes. The nodes B, C, D are internal nodes having the child as well as parent nodes. Before discussing binary trees in detail, let us discuss some basic terminologies that are used in association with binary trees (refer to Figure 6.2).

- **Ancestor and descendant:** A node N1 is said to be an ancestor of node N2 if N1 is the parent node of N2 or parent of the parent node of N1, and so on, whereas node N2 is said to be a descendant of node N1. The node N2 is said to be the left descendant of node N1 if it belongs to the left subtree of N1 and is said to be the right descendant of N1 if it belongs to the right subtree of N1. In the binary tree, as shown in Figure 6.2, node A is the ancestor of node H, and node H is the left descendent of node A.
- **Degree of a node:** The degree of a node is equal to the number of its child nodes. In the binary tree shown in Figure 6.2, the nodes A, B, and D have degree 2; node C has degree 1; and nodes G, H, E, and F have degree 0.
- **Level:** Since the binary tree is a multilevel data structure, each node belongs to a particular level number. In the binary tree shown in Figure 6.2, the root node A belongs to level 0, its child nodes belong to level 1, child nodes of nodes B and C belong to level 2, and so on.
- **Depth (or height):** Depth of the binary tree is the highest level number of any node in a binary tree. In the binary tree shown in Figure 6.2, the nodes G and H are nodes with the highest level number 3. Hence, the depth of the binary tree is 3.
- **Siblings:** The nodes belonging to the same parent node are known as sibling nodes. In the binary tree shown in Figure 6.2, nodes B and C are sibling nodes as they have the same parent node, that is, A. Similarly, nodes D and E are also sibling nodes.
- **Edge:** Edge is a line connecting any two nodes. In the binary tree shown in Figure 6.2, there exists an edge between nodes A and B, whereas there is no edge between nodes B and C.
- **Path:** Path between the two nodes x and y is a sequence of consecutive edges being followed from node x to y. In the binary tree shown in Figure 6.2, the path

between the nodes A and H is A->B->D->H. Similarly, the path from A to F is A->C->F.

There are various forms of binary trees that are formed by imposing certain restrictions on them. Some of the variations of binary trees are—complete binary tree and extended binary tree.

Complete binary tree

A binary tree is said to be a complete binary tree if all the leaf nodes of the tree are at the same level. Thus, the tree has a maximum number of nodes at all levels (see Figure 6.3). At any level n of a binary tree, there can be at the most 2 n nodes. That is,

At n = 0, there can be at most 2 0 = 1 node.

At n =1, there can be at most 2 1 = 2 nodes.

At n = 2, there can be at most 2 2 = 4 nodes.

:

At level n, there can be at most 2 n nodes.

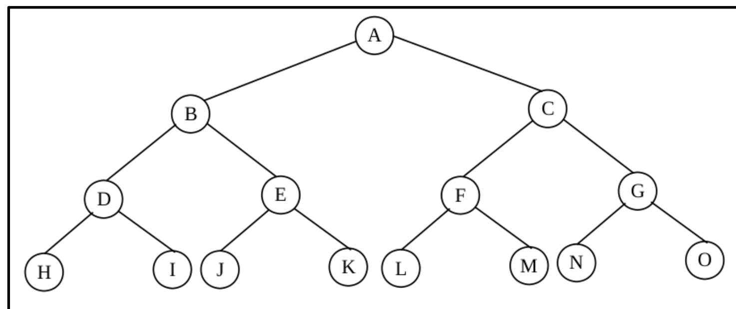


Figure 6.3 Complete Binary Tree

Extended binary tree

A binary tree is said to be an extended binary tree (also known as 2-tree) if all of its nodes are of either a zero degree or two degrees. In this type of binary tree, the nodes with degree two (also known as internal nodes) are represented as circles, and nodes with degree zero (also known as external nodes) are represented as squares (Figure 6.4).

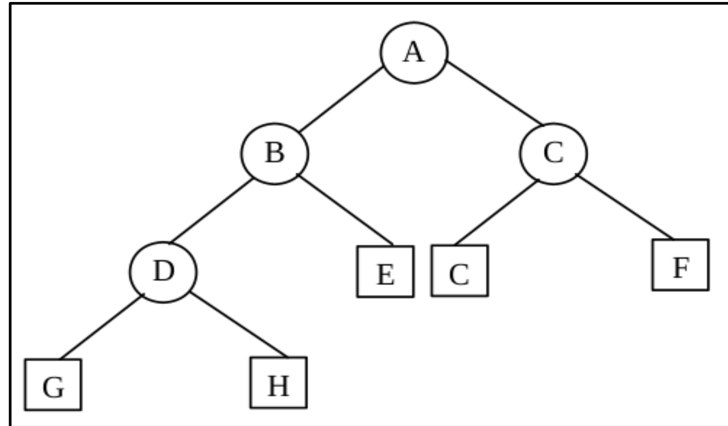


Figure 6.4 Extended Binary Tree

6.4 Representation of a Binary Tree

Like stacks and queues, binary trees can also be represented in the memory in two ways: memory-array (sequential) representation and linked representation. In an array representation, memory is allocated at compile-time, while in linked representation, memory is allocated dynamically.

6.4.1 Array Representation

In an array representation, a binary tree is represented sequentially in memory by using a single one-dimensional array. A binary tree of height n may comprise utmost 2 (n+1) -1 nodes, hence an array of maximum size 2 (n+1) -1 is used for representing such a tree. All the nodes of the tree are assigned a sequence number [from 0 to (2 (n+1) -1)-1] level by level. In other words, the root node at level 0 is assigned a sequence number 0, then nodes at level 1 are assigned sequence number in ascending order from left to right, and so on. For example, the nodes of a binary tree of height 2, having 7 (2 (n+1) -1) nodes can be numbered as shown in Figure 6.5 (a).

The numbers assigned to the nodes indicate the position (index value) of an array at which that particular node is stored. The array representation of this tree is shown in Figure 6.5 (b). It can be observed that if any node is stored at position p, then its left child node is stored at 2*p+1 position, and its right child node is stored at 2*p+2 position. In Figure 6.5(b), for example, the node G is stored at position 1, its left child node D is stored at position 3 (2*1+1) and its right child node is stored at position 4 (2*1+2). Note that if any of the nodes in the tree have empty subtrees (except the leaf nodes), the nodes forming the part of these empty subtrees are also numbered and

their values in the corresponding position in the array are NULL.

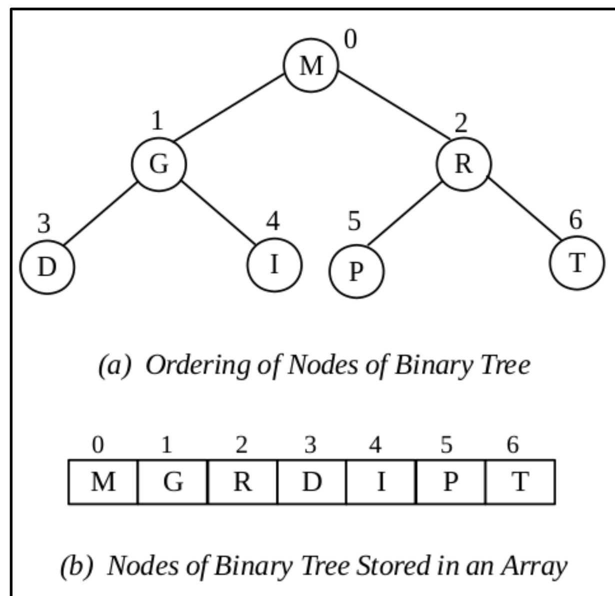


Figure 6.5 Array Representation of a Binary Tree

Consider an example, a binary tree is shown in Figure 6.6 (a). Its array representation is shown in Figure 6.6 (b). In this representation, an array of maximum size is declared (to accommodate the maximum number of nodes for a binary tree of a given height) before run-time which leads to a wastage of a lot of memory space in the case of unbalanced trees.

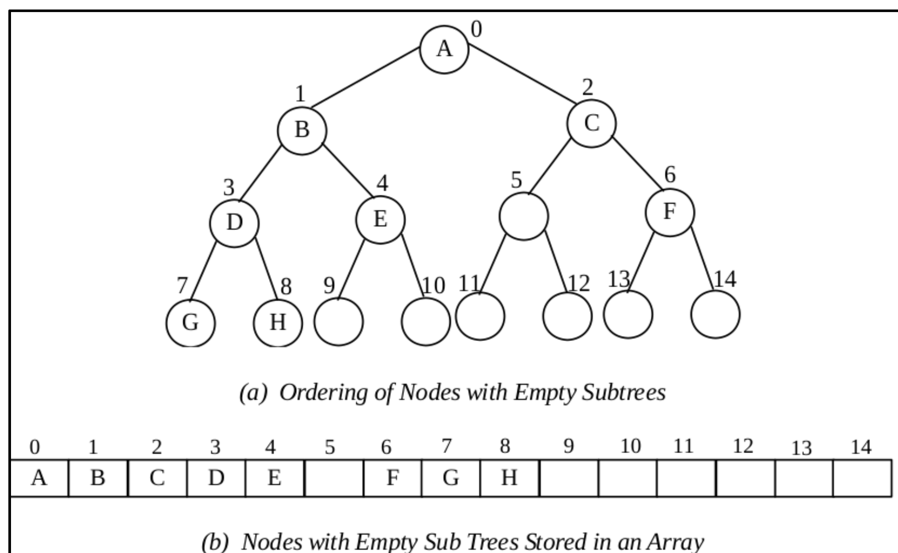
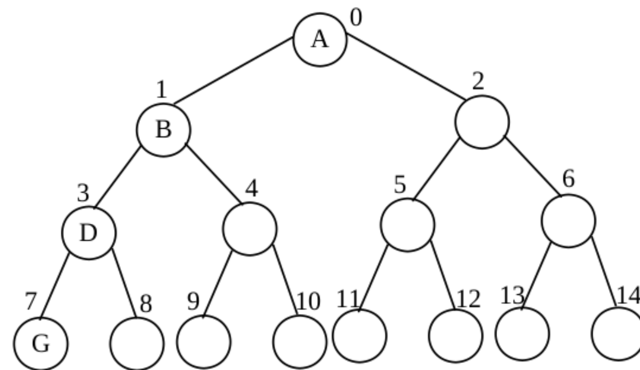


Figure 6.6 Array Representation of Binary Tree with Empty subtrees

In unbalanced trees, the number of nodes is very small as compared to the maximum number of nodes for a given height. Consider, for example, an unbalanced tree is shown in Figure 6.7 (a). Since, this tree is of height 3, an array of size 14 (2 (3+1 -1) will be declared to store nodes of this tree. The array representation of this tree is shown in Figure 6.7 (b).



(a) An Unbalanced Binary Tree

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	B		D				G							

(b) Nodes of an Unbalanced Binary Tree Stored in an Array

Figure 6.7 Array Representation of an Unbalanced Binary Tree

It can be observed from this array representation that most of the array positions are NULL, leading to wastage of memory space. Due to this disadvantage of array representation of binary trees, the linked representation of binary trees is preferred.

6.4.2 Linked Representation

Linked representation is one of the most common and important ways of representing a binary tree in memory. The linked representation of a binary tree is implemented by using a linked list having an info part and two pointers. The info part contains the data value and two pointers, left and right, are used to point to the left and right subtree of a node, respectively. The structure of such a node is shown in Figure 6.8.

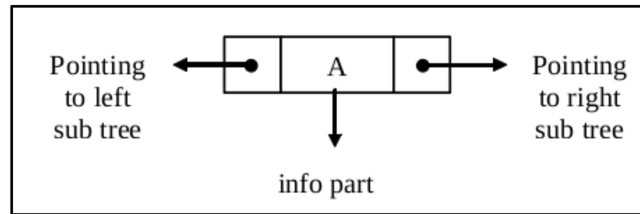


Figure 6.8 Structure of a Node of a Binary Tree

To define a node of a binary tree in 'C' language, a self-referential structure can be used whose definition is as follows.

```
typedef struct node
{
        int info;
        struct node *left;
        struct node *right;
}Node;
```

In linked representation, a pointer variable Root of Node type is used to point to the root node of a tree. The root variable is used for accessing the root and the subsequent nodes of a binary tree. Since the binary tree is empty in the beginning, the pointer variable Root is initialized with NULL. The linked representation of a sample binary tree is shown in Figure 6.9.

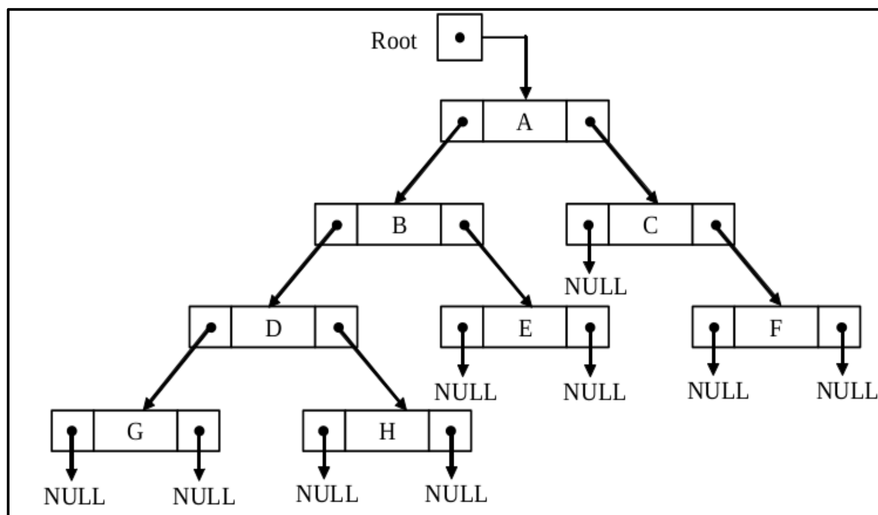


Figure 6.9 Linked Representation of a Binary Tree

6.5 Binary Tree Traversals

Traversing a binary tree refers to the process of visiting each and every node of the tree exactly once. The three different ways in which a tree can be traversed are—in-order, pre-order, and post-order traversals. The main difference in these traversal methods is based on the order in which the root node is visited. Note that in all the traversals the left subtree is always traversed before the traversal of the right subtree. To understand these traversal methods, consider a simple binary tree T, shown in Figure 6.10.

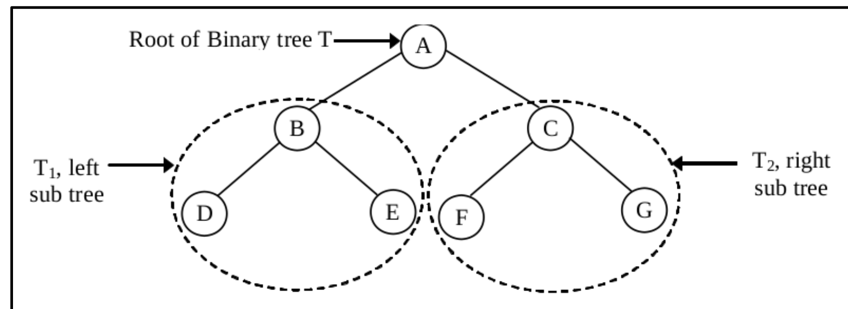


Figure 6.10 A simple Binary Tree T

Pre-order

In pre-order traversal, the root node is visited before traversing its left and right subtrees. Steps for traversing a non-empty binary tree in pre-order are:

1. Visit the root node R.
2. Traverse the left subtree of root node R in pre-order.
3. Traverse the right subtree of root node R in pre-order.

In the binary tree T (shown in Figure 6.10), for example, the root node A is traversed before traversing its left subtree and right subtree. In the left subtree T_1, the root node B (of left subtree T_1) is traversed before traversing the nodes D and E. After traversing the root node of binary tree T and traversing the left subtree T_1, the right subtree T_2 is also traversed following the same procedure. Hence, the resultant pre-order traversal of the binary tree T is A, B, D, E, C, F, G.

In-order

In in-order traversal, the root node is visited after the traversal of its left subtree and before the traversal of its right subtree. Steps for traversing a non-empty binary tree in in-order are:

1. Traverse the left subtree of root node R in in-order.

2. Visit the root node R.
3. Traverse the right subtree of root node R in in-order.

In the binary tree T (shown in Figure 6.10), for example, the left subtree T_1 is traversed before traversing the root node A. In the left subtree T_1, the node D is traversed before traversing its root node B (of left subtree T_1). After traversing the node D and B, node E is traversed. Once the traversals of left subtree T_1 and the root node A of binary tree T are complete, the right subtree T_2 is traversed following the same procedure. Hence, the resultant in-order traversal of the binary tree T is D, B, E, A, F, C, G.

Post-order

In post-order traversal, the root node is visited after traversing its left and right subtrees. Steps for traversing a non-empty binary tree in post-order are:

1. Traverse the left subtree of root node R in post-order.
2. Traverse the right subtree of root node R in post-order.
3. Visit the root node R.

In binary tree T (shown in Figure 6.10), for example, the root node A is traversed after traversing its left subtree and right subtree. In the left subtree T_1, the root node B (of left subtree T_1) is traversed after traversing the nodes D and E. Similarly, the nodes of right subtree T_2 are traversed following the same procedure. After traversing the left subtree (T_1) and right subtree (T_2), the root node A of binary tree T is traversed. Hence, the resultant post-order traversal of the binary tree T is D, E, B, F, G, C, A.

In addition to these traversals, there is another way of traversing a tree known as *level-order traversal*. In this traversal, every node at one level is visited before moving onto the next level.

6.6 Binary Search Tree

A binary search tree, also known as a *binary sorted tree*, is a kind of a binary tree that satisfies the following conditions (Figure 6.11):

1. The data value in each node is a key (unique) value, that is, no two nodes can have identical values.
2. The data values in the nodes of the left subtree, if exists, are smaller than the value in the root node.
3. The data values in the nodes of the right subtree, if exists, are greater than or

equal to the value in the root node.

4. The left and right subtrees, if exist, are also binary search trees.

In other words, values in the left subtree of a root node are smaller than the value of the root node, and values in the right subtree are greater than or equal to the value of the root node. This rule is applicable to all the subsequent subtrees in a binary search tree. In addition, each and every value in a binary search tree is unique, that is, no two nodes in it can have identical values.

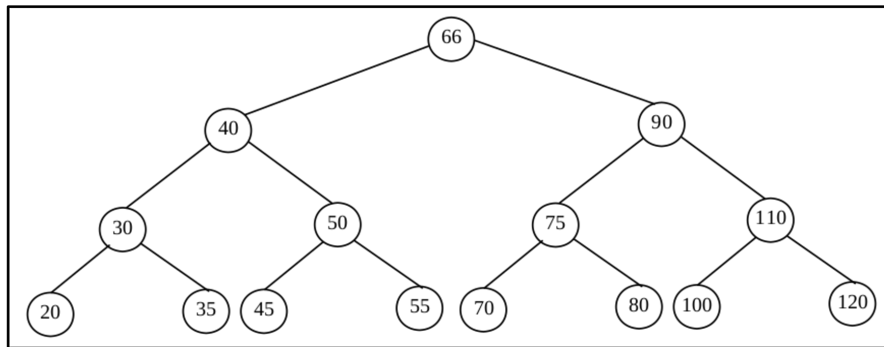


Figure 6.11 Binary Search Tree

There are various operations that can be performed on the binary search trees. Some of these are search of a node, insertion of a new node, deletion of a node, and traversal of a tree.

Searching a Node in Binary Search Tree

Searching an element in a binary search tree is easy since the elements in this tree are arranged in sorted order. The element to be searched is compared with the value in the root node. If the element is smaller than the value in the root node, then the searching will proceed to the left subtree, and if the element is greater than the value in the root node, then the searching will proceed to the right subtree. This process is repeated until either the element to be searched is found or NULL value is encountered.

Consider, for example, a sample binary search tree given in Figure 6.11. The steps to search element 45 are given here.

1. Compare element 45 with the value in the root node (66). Since 45 is smaller

than 66, move to its left subtree.

2. Compare element 45 with the value (40) appearing in the left subtree. Since 45 is greater than 40, move to its right subtree.
3. Now, compare element 45 with the value (50) appearing in the right subtree. Since 45 is smaller than 50, move to its left subtree.
4. In the next step, compare element 45 with the value (45) appearing in the left subtree. Since 45 is equal to the value (45) stored in this node, the required element is found. Therefore, terminate the procedure.

In case the value 48 is to be searched, the first four steps are the same. After step 4, the right subtree of 45 will be accessed. This is NULL indicating the end of the tree. Therefore, the element is not found in the tree and the search is unsuccessful.

Algorithm 6.1 Searching in a Binary Search Tree

```
search(item, ptr)
1. If !(ptr)
       Print "Element not found!" and go to step 3
   End If
2. If item < ptr->info
      Call search(item, ptr->left)
   Else If item > ptr->info
      Call search(item, ptr->right)
   Else
      Print "Element found."
   End If
3. End
```

Inserting a Node

Insertion in a binary search tree is similar to the procedure for searching an element in a binary search tree. The difference is that in the case of insertion, an appropriate null pointer is searched where a new node can be inserted. The process of inserting a node in a binary search tree can be divided into two steps-in the first step, the tree is searched to determine the appropriate position where the node is to be inserted and in the second step, the node is inserted at this searched position.

There are two cases of insertion in a tree-first, insertion into an empty tree, and second insertion into a non-empty tree. In case the tree is initially empty, the new node to be inserted becomes its root node. In case the tree is non-empty, an appropriate position is determined for insertion. For this, first of all, the value in the new node is compared with the root node of the tree. If the value in the new node is less than the value in the root node, the new node is added as the left leaf if the left subtree is empty, otherwise, the search continues in the left subtree. On the other hand, if the value in the new node is greater than the value in the root node, the new node is added as the right leaf if the right subtree is empty, otherwise, the search continues in the right subtree.

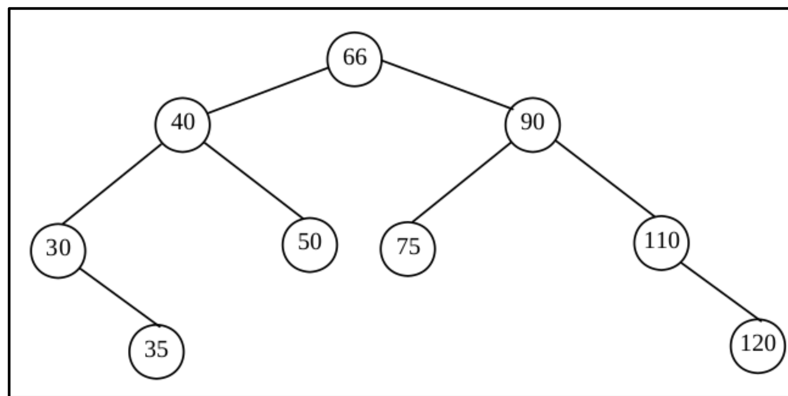


Figure 6.12 (a) A sample Binary Search Tree

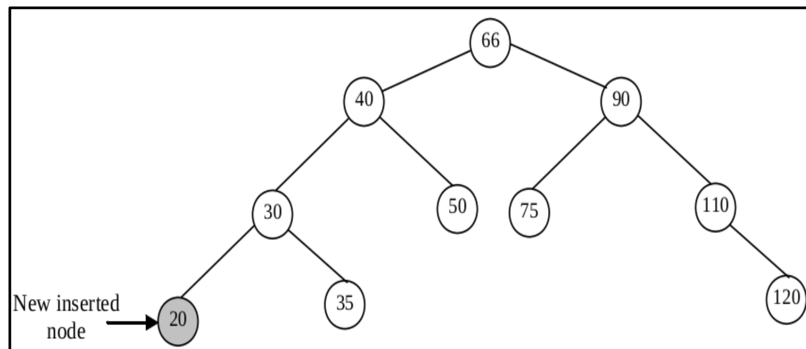


Figure 6.12 (b) Insertion of a node with value 20

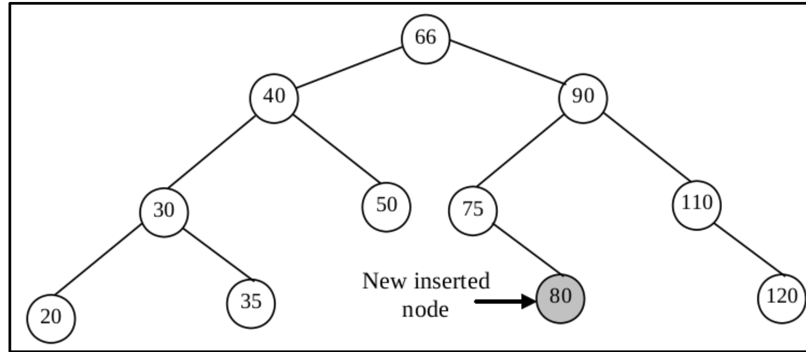


Figure 6.12 (c) Insertion of a node with value 80

Consider, for example, a sample binary search tree is shown in Figure 6.12 (a). For inserting elements 20 and 80, follow the steps given here.

Steps for inserting element 20 are as follows:

1. Compare 20 with the value in the root node, that is, 66. Since 20 is smaller than 66, move to the left subtree.
2. Finding that the left pointer of the root node is non-null, compare 20 with the value (40) in this node. Since 20 is smaller than 40, move to the left subtree.
3. Again, as the left pointer of the current node is non-null, compare 20 with the value (30) in this node. Since 20 is smaller than 30, move to the left subtree.
4. Now, the left pointer is null, thus 20 will be inserted at this position. After insertion, the tree will appear as shown in Figure 6.12 (b).

Steps for inserting element 80 are as follows:

1. Compare 80 with the value in root node 66. Since 80 is greater than 66, move to the right subtree.
2. Finding that the right pointer of the root node is non-null, compare 80 with the value (90) in this node. Since 80 is smaller than 90, move to the left subtree.
3. Again, as the left pointer of the current node is non-null, compare 80 with the value (75) in this node. Since 80 is greater than 75, move to the right subtree.
4. Now, the right pointer is null, thus 80 will be inserted at this position. After insertion, the tree will appear as shown in Figure 6.12 (c).

Algorithm 6.2 Insertion into a Binary Search Tree

```
insert_node(item, ptr)
1. If !(ptr)
       Allocate memory for ptr
       Set ptr->info = item
       Set ptr->left = NULL
       Set ptr->right = NULL
   Else
       If item < ptr->info
              Call insert_node(item, ptr->left)
       Else
              Call insert_node(item, ptr->right)
       End If
   End If
2. End
```

Deleting a Node in Binary Search Tree

Deletion of a node from a binary search tree involves two steps—first, searching the desired node, and second, deleting the node. Whenever a node is deleted from a tree, it must be ensured that the tree remains a binary search tree, that is, the sorted order of the tree must not be disturbed. The node being deleted may have zero, one, or two child nodes. On the basis of the number of child nodes of the node to be deleted, there are three cases of deletion which are discussed here.

Case 1: If the node to be deleted has no child node, it is deleted by making its parent's pointer pointing to NULL and de-allocating memory allocated to it. The node with value 75, for example, is to be deleted from the tree shown in Figure 6.13 (a). Since this node has no child node, its parent's (90) left pointer will be made to point to NULL and the memory space of the node (75) is de-allocated.

Case 2: If the node to be deleted has only one child node, it is deleted by adjusting its parent's pointer pointing to its only child and deallocating memory allocated to it. The node, for example, with value 110 is to be deleted from the tree shown in Figure 6.13 (b). Since this node has one child node, its parent's (90) right pointer will be made to point to its child node (120) and the memory space of the node (110) is deallocated.

Case 3: If the node to be deleted has two child nodes, it is deleted by replacing its value by the largest value in the left subtree (in-order predecessor) or by the smallest value in the right subtree (in-order successor). The node whose value is used for replacement is then deleted using case 1 or case 2.

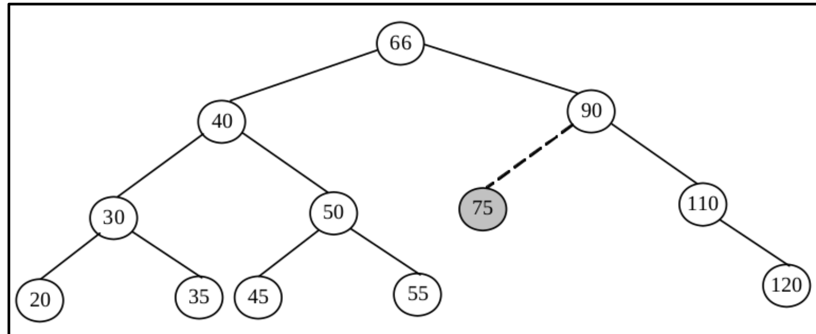


Figure 6.13 (a) Deletion of a Node with No Child Node

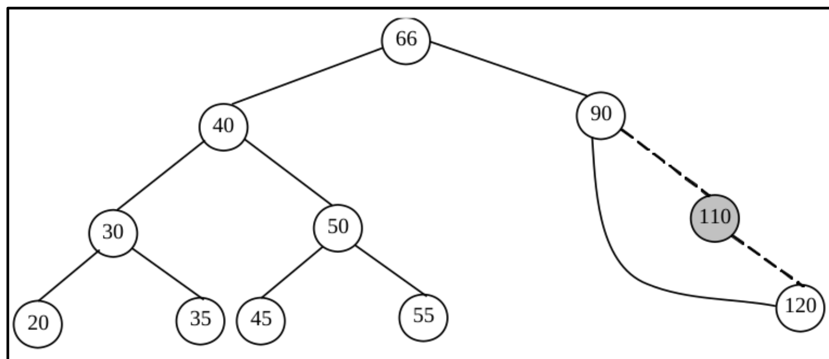


Figure 6.13 (b) Deletion of a Node with Only One Child

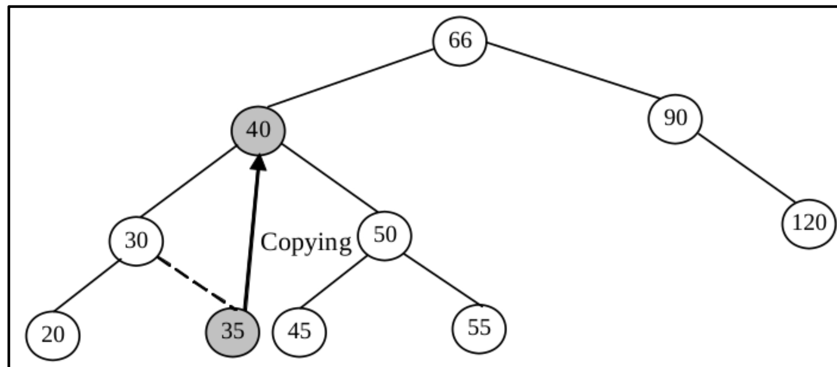


Figure 6.13 (c) Deletion of a Node with Two Child Nodes

The node, for example, with the value 40 is to be deleted from the tree shown in Figure 6.13 (c). Since this node has two subtrees or child nodes, a value has to be searched from its subtrees which can be used for its replacement. The value that will

be used for replacement can either be the largest value from its left subtree (35) or the smallest value from its right subtree (45). Suppose the value 35 is selected for this purpose, then the value 35 is copied in the node with the value 40. After this, the right pointer of the parent node (30) of the node used for replacement (35) is made to point to NULL, and memory allocated to the node with value 35 is de-allocated. As a result of the deletion of this node, the order of the tree is maintained. The final structure of the tree after the deletion of node 40 will be as shown in Figure 6.14.

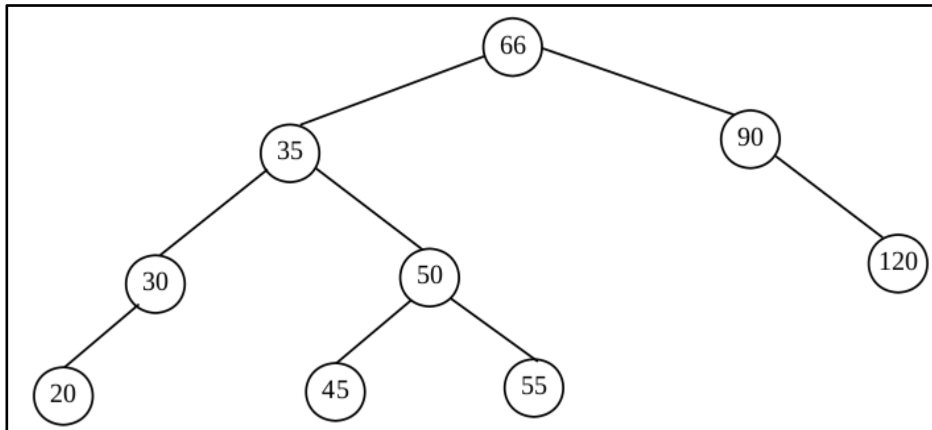


Figure 6.14 Binary Search Tree after Deletion

Algorithm 6.3 Deletion from Binary Search Tree

```
del_node(item, ptr)
1. If !(ptr)
        Print "Item does not exist." and go to step 3
2. If item < ptr->info
        Call del_node(item,&(ptr->left))
    Else
        If item > ptr->info
            Call del_node(item,&(ptr->right))
        Else
            If item = ptr->info
                Set save = ptr
            If save->right = NULL
                Set ptr = save->left
                Deallocate save
            Else
                If save->left = NULL
                Set ptr = save->right
```

```
                Deallocate save
            Else
                Call del(&(save->left),save)
            End If
          End If
       End If
   End If
   End If
3. End
```

```
del(p, q)        //q is the node to be deleted, p is the node whose value is used        for
replacing the value in q and p is de-allocated
1. If p->right != NULL
        Call del(&(p->right),q)
   Else
        Set delnode = p
        Set q->info = p->info
        Set p = p->left
        Deallocate delnode
   End If
2. End
```

Traversals in Binary Search Tree

Traversing a binary search tree is the same as traversing a binary tree. In other words, binary search trees can also be traversed in three different ways—pre-order, in-order, and post-order. It can be observed that when a binary search tree is traversed in-order, it results in the sequence of elements in ascending order. The algorithms for traversing trees in pre-order, in-order, and post-order are recursive in nature, which are given below.

Algorithm 6.4 Pre-order Traversal in Binary Search Tree

```
preorder(ptr)
1. If ptr != NULL
        Print ptr->info        //ptr is temporary pointer initialised with Root
        Call preorder(ptr->left)
        Call preorder(ptr->right)
   End If
2. End
```

Algorithm 6.5 In-order Traversal in Binary Search Tree

inorder(ptr)
1. If ptr != NULL
        Call inorder(ptr->left) //ptr is temporary pointer initialised with Root
        Print ptr->info
        Call inorder(ptr->right)
   End If
2. End

Algorithm 6.6 Post-order Traversal in Binary Search Tree

postorder(ptr)
1. If ptr != NULL
        Call postorder(ptr->left) //ptr is temporary pointer initialised with Root
        Call postorder(ptr->right)
        Print ptr->info
   End If
2. End

Program 6.1: A program to illustrate various operations performed on binary search tree [In case of deletion of a node with two child nodes, the largest value from left subtree (in-order predecessor) is used for replacement].

```
#include<stdio.h>
#include<conio.h>
typedef struct node {
                int info;
                struct node *left;
                struct node *right;
        }Node;
int nodes, leaves;
/*Function prototypes*/
void insert_node(int, Node **);
void search(int, Node *);
```

```
void print_treeform(Node *, int);
void preorder(Node *);
void inorder(Node *);
void postorder(Node *);
void count_nodes(Node *);
void count_leaves(Node *);
void del(Node **, Node *);
void del_node(int, Node **);
void main()
{
        int choice, n;
        Node *root=NULL;
        do
        {
                clrscr();
                printf("\nMain Menu");
                printf("\n1. Insert");
                printf("\n2. Display in tree form");
                printf("\n3. Pre-order traversal of tree");
                printf("\n4. In-order traversal of tree");
                printf("\n5. Post-order traversal of tree");
                printf("\n6. Number of nodes");
                printf("\n7. Number of leaves");
                printf("\n8. Searching");
                printf("\n9. Delete");
                printf("\n10.Exit");
                printf("\nEnter your choice . . . ");
                scanf("%d", &choice);
                switch(choice)
                {
                        case 1 : printf("\nEnter data for new node:");
                                scanf("%d", &n);
                                insert_node(n, &root);
                                Break;
                        case 2 : printf("\nTree in tree form —>\n");
                                if(!root)
                                        print_treeform(root, 1);
```

```
        else
                printf("Tree is empty!!");
        break;
case 3 : printf("\nPre-order traversal of tree —>\n\n");
        if(!root)
                preorder(root);
        else
                printf("Tree is empty!!");
        break;
case 4 : printf("\nIn-order traversal of tree —>\n\n");
        if(!root)
                inorder(root);
        else
                printf("Tree is empty!!");
        break;
case 5 : printf("\nPost-order traversal of tree —>\n\n");
        if(!root)
                postorder(root);
        else
                printf("Tree is empty!!");
        break;
case 6 : if(root==NULL)
                nodes=0;
        else
                nodes=1;
                count_nodes(root);
                printf("\nNumber of nodes are : %d", nodes);
        break;
case 7 : leaves=0;
        count_leaves(root);
        printf("\nNumber of leaves are : %d", leaves);
        break;
case 8 : printf("\nEnter value of node to be searched : ");
        scanf("%d", &n);
        search(n, root);
        break;
case 9 : printf("\nEnter value of node to be deleted : ");
```

```
                    scanf("%d", &n);
                    del_node(n, &root);
                    break;
                case 10 : printf("\nNormal termination of program.");
                    break;
                    default : printf("\nWrong Choice !!");
            }
            getch();
    }while(choice!=10);
}
/*Function to insert node in a tree*/
void insert_node(int item, Node **ptr)
{
    if(!(*ptr))
    {
        (*ptr)=(Node*) malloc(sizeof(Node));
        (*ptr)->info=item;
        (*ptr)->left=NULL;
        (*ptr)->right=NULL;
    }
    if(item<(*ptr)->info)
        insert_node(item,&((*ptr)->left));
    else if(item>(*ptr)->info)
        insert_node(item,&((*ptr)->right));
    }
/*Function to print tree in tree format*/
void print_treeform(Node *ptr, int level)
{
    int i;
    if(ptr)
    {
        print_treeform(ptr->right, level+1);
        printf("\n");
        for(i=0;i<level;i++)
            printf(" ");
        printf("%d", ptr->info);
        print_treeform(ptr->left, level+1);
```

```
        }
}
/*Function to print tree in pre-order*/
void preorder(Node *ptr)
{
        if(ptr)
        {
                printf("%d ", ptr->info);
                preorder(ptr->left);
                preorder(ptr->right);
        }
}
/*Function to print tree in in-order*/
void inorder(Node *ptr)
{
        if(ptr)
        {
                inorder(ptr->left);
                printf("%d ", ptr->info);
                inorder(ptr->right);
        }
}
/*Function to print tree in post-order*/
void postorder(Node *ptr)
{
        if(ptr)
        {
                postorder(ptr->left);
                postorder(ptr->right);
                printf("%d ", ptr->info);
        }
}
/*Function to count number of nodes in a tree*/
void count_nodes(Node *ptr)
{
        if(ptr != NULL)
        {
```

```
                if(ptr->left != NULL)
                {
                        nodes++;
                        count_nodes(ptr->left);
                }
                if(ptr->right != NULL)
                {
                        nodes++;
                        count_nodes(ptr->right);
                }
        }
}
/*Function to count number of leaves in a tree*/
void count_leaves(Node *ptr)
{
        if(ptr != NULL)
        {
                if((ptr->left==NULL) && (ptr->right==NULL))
                        leaves++;
                else
                        count_leaves(ptr->left);
                        count_leaves(ptr->right);
        }
}
/*Function to search a node in a tree*/
void search(int item, Node *ptr)
{
        if(!ptr)
        {
                printf("Element not found.");
                return;
        }
        else if(item<ptr->info)
                search(item, ptr->left);
        else if(item>ptr->info)
                search(item, ptr->right);
        else
```

```
        {
                printf("Element found.");
        }
}
/*Function to delete a node from tree*/
void del_node(int item, Node **ptr)
{
        Node *save;
        if(!(*ptr))
        {
                printf("\nItem does not exist.");
                return;
        }
        else
        {
                if(item<(*ptr)->info)
                        del_node(item, &((*ptr)->left));
                else
                        if(item>(*ptr)->info)
                                del_node(item, &((*ptr)->right));
                        else if(item==(*ptr)->info)
                        {
                                save=*ptr;
                                if(save->right==NULL)
                                {
                                        *ptr=save->left;
                                        free(save);
                                }
                                else
                                        if(save->left==NULL)
                                        {
                                                *ptr=save->right;
                                                free(save);
                                        }
                                        else
                                                del(&(save->left), save);
                        }
```

```
    }
    return;
}
/*Called from Del_node() function to delete nodes with child nodes*/
void del(Node **p, Node *q)
{
    Node *delnode;
    if((*p)->right != NULL)
        del(&((*p)->right), q);
    else
    {
        delnode=*p;
        q->info=(*p)->info;
        *p=(*p)->left;
        free(delnode);
    }
    return;
}
```

The output of the program is:

Main Menu

1. Insert
2. Display in tree form
3. Pre-order traversal of tree
4. In-order traversal of tree
5. Post-order traversal of tree
6. Number of nodes
7. Number of leaves
8. Searching
9. Delete
10. Exit

Enter your choice . . . 1

Enter data for new node: 66

: / Similarly insert values 40 90 30 50 75 110 20 35 45 55 70 80 100 120 in this: order*/*

Main Menu

```
1. Insert
2. Display in tree form
3. Pre-order traversal of tree
4. In-order traversal of tree
5. Post-order traversal of tree
6. Number of nodes
7. Number of leaves
8. Searching
9. Delete
10. Exit
Enter your choice . . . 2
Tree in tree form —>
                120
        110
                100
        90
                80
        75
                70
        66
                55
        50
                45
        40
                35
        30
                20
        Main Menu
1. Insert
2. Display in tree form
3. Pre-order traversal of tree
4. In-order traversal of tree
5. Post-order traversal of tree
```

6. Number of nodes
7. Number of leaves
8. Searching
9. Delete
10. Exit

Enter your choice . . . 3

Pre-order traversal of tree —>

66 40 30 20 35 50 45 55 90 75 70 80 110 100 120

Main Menu

1. Insert
2. Display in tree form
3. Pre-order traversal of tree
4. In-order traversal of tree
5. Post-order traversal of tree
6. Number of nodes
7. Number of leaves
8. Searching
9. Delete
10. Exit

Enter your choice . . . 4

In-order traversal of tree —>

20 30 35 40 45 50 55 66 70 75 80 90 100 110 120

Main Menu

1. Insert
2. Display in tree form
3. Pre-order traversal of tree
4. In-order traversal of tree
5. Post-order traversal of tree
6. Number of nodes
7. Number of leaves
8. Searching
9. Delete
10. Exit

Enter your choice . . . 5

Post-order traversal of tree —>

20 35 30 45 55 50 40 70 80 75 100 120 110 90 66

Main Menu

1. Insert
2. Display in tree form
3. Pre-order traversal of tree
4. In-order traversal of tree
5. Post-order traversal of tree
6. Number of nodes
7. Number of leaves
8. Searching
9. Delete
10. Exit

Enter your choice . . . 6

Number of nodes are: 15

Main Menu

1. Insert
2. Display in tree form
3. Pre-order traversal of tree
4. In-order traversal of tree
5. Post-order traversal of tree
6. Number of nodes
7. Number of leaves
8. Searching
9. Delete
10. Exit

Enter your choice . . . 7

Number of leaves are: 8

Main Menu

1. Insert
2. Display in tree form
3. Pre-order traversal of tree

4. In-order traversal of tree

5. Post-order traversal of tree

6. Number of nodes

7. Number of leaves

8. Searching

9. Delete

10. Exit

Enter your choice . . . 8

Enter value of node to be searched: 120

Element found.

Main Menu

1. Insert

2. Display in tree form

3. Pre-order traversal of tree

4. In-order traversal of tree

5. Post-order traversal of tree

6. Number of nodes

7. Number of leaves

8. Searching

9. Delete

10. Exit

Enter your choice . . . 9

Enter value of node to be deleted: 70

Main Menu

1. Insert

2. Display in tree form

3. Pre-order traversal of tree

4. In-order traversal of tree

5. Post-order traversal of tree

6. Number of nodes

7. Number of leaves

8. Searching

9. Delete

```
10. Exit
Enter your choice . . . 9
Enter value of node to be deleted: 75
        Main Menu
1. Insert
2. Display in tree form
3. Pre-order traversal of tree
4. In-order traversal of tree
5. Post-order traversal of tree
6. Number of nodes
7. Number of leaves
8. Searching
9. Delete
10. Exit
Enter your choice . . . 2
Tree in tree form —>
                120
        110
                100
        90
                80
        66
                55
        50
                45
        40
                35
        30
                20
        Main Menu
1. Insert
2. Display in tree form
3. Pre-order traversal of tree
```

```
4. In-order traversal of tree
5. Post-order traversal of tree
6. Number of nodes
7. Number of leaves
8. Searching
9. Delete
10. Exit
Enter your choice . . . 9
Enter value of node to be deleted: 40
          Main Menu
1. Insert
2. Display in tree form
3. Pre-order traversal of tree
4. In-order traversal of tree
5. Post-order traversal of tree
6. Number of nodes
7. Number of leaves
8. Searching
9. Delete
10. Exit
Enter your choice . . . 2
Tree in tree form —>
                    120
          110
                    100
          90
                    80
          66
                    55
          50
                    45
          35
                    30
```

20

Main Menu

1. Insert
2. Display in tree form
3. Pre-order traversal of tree
4. In-order traversal of tree
5. Post-order traversal of tree
6. Number of nodes
7. Number of leaves
8. Searching
9. Delete
10. Exit

Enter your choice . . . 10

Normal termination of program.

6.7 Threaded Binary Tree

One of the most common operations that are performed on the trees is the traversal of nodes. Hence, it is required to make this operation more efficient. This can be achieved by utilizing space occupied by the NULL pointers in the leaf nodes and internal nodes having only one child node. These pointers can be modified to point to their corresponding in-order successor, in-order predecessor, or both. These modified pointers are known as threads and binary trees having such types of pointers are known as *threaded binary trees.*

The different types of threaded binary trees are as follows (see Figure 6.15, threads denoted by dotted lines):

- **Right-threaded binary tree:** In this tree, the right NULL pointer of each node (not having the right child node) points to its in-order successor. Such a right NULL pointer is known as the right thread. In this tree, only the right pointer of the rightmost node [F, see Figure 6.15 (a)] will be a NULL pointer and all the left NULL pointers will remain NULL.
- **Left-threaded binary tree:** In this tree, the left NULL pointer of each node (not

having a left child node) points to its in-order predecessor. Such a left NULL pointer is known as a left thread. In this tree, only the left pointer of the leftmost node [G, see Figure 6.15(b)] will be a NULL pointer, and all right NULL pointers will remain NULL.

- **Full-threaded binary tree:** In this tree, both right and left NULL pointers, point to their in-order successor and in-order predecessor, respectively. In this tree, both the right pointer of the rightmost node (F) and the left pointer of the leftmost node (G) will be a NULL pointer.

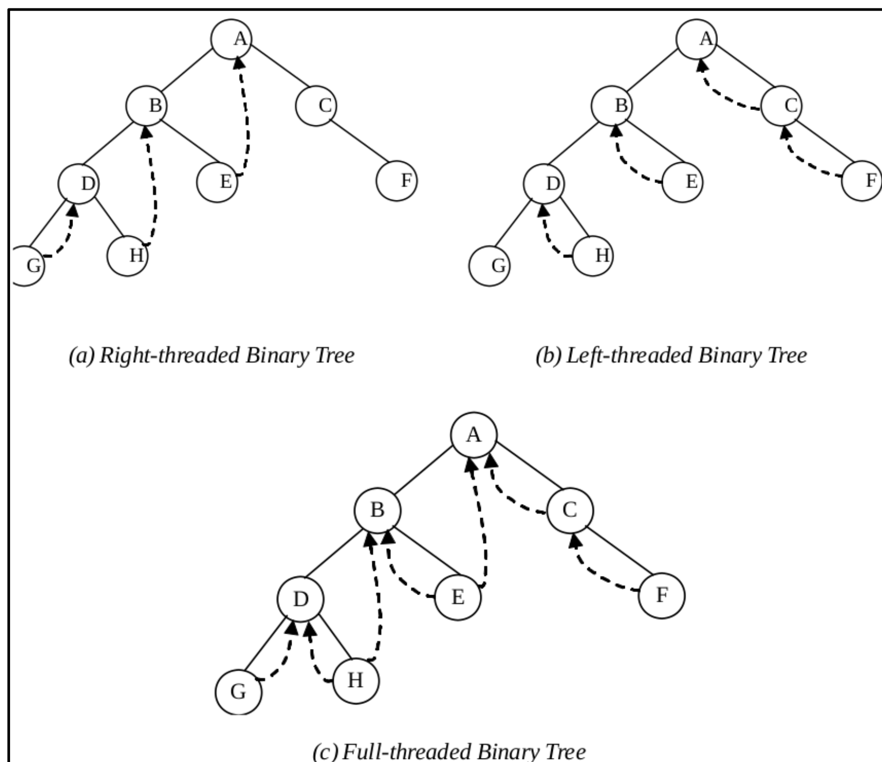


Figure 6.15 Types of Threaded Binary Tree

This way of threading the binary trees corresponds to the in-order traversal of the tree. Similarly, there can be threaded binary trees corresponding to the pre-order traversal of trees. However, there is no threaded binary tree corresponding to the post-order traversal of the tree. Threaded binary trees can also be categorized on the basis of the number of threads being used. A threaded binary tree in which only one thread is used is known as a *one-way threaded binary tree*, whereas a threaded binary tree in which two threads are used is known as *a two-way threaded binary tree.*

The structure of the node of a threaded binary tree is similar to the node of a binary

tree, with some additional variables, indicating whether the left or right pointers are normal pointers or threads. To define a node of a full-threaded binary tree in 'C' language, a self-referential structure can be used whose definition is given here.

```
typedef struct node
{
        int info;
        struct node *left;
        char lthread;
        struct node *right;
        char rthread;
};
```

The variables lthread and rthread are used to indicate whether the left and right pointers are normal pointers or threads. The value '1' is stored in these variables to indicate that the corresponding left and right pointers are normal pointers and the value '0' indicates that the corresponding variables will be used as threads. In the case of the right-threaded binary tree and left-threaded binary tree, only corresponding variables are included in the structure of the node. The linked representation of a full-threaded binary tree is shown in Figure 6.16.

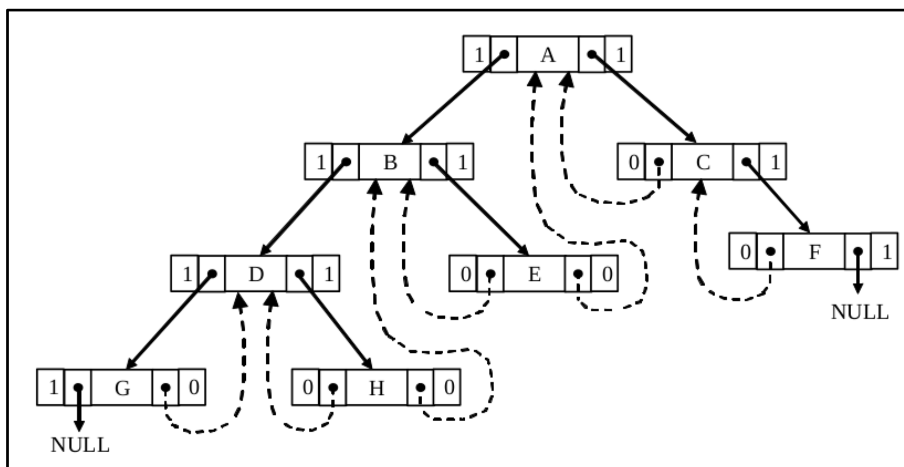


Figure 6.16 Storage Representation of Full-threaded Binary Tree

6.8 Summary

- A tree is a non-linear data structure representing a hierarchical structure of one or more elements known as nodes.
- The node at the top of the tree is known as the root of the tree and the nodes at

the lowest level are known as the leaf nodes.

- The binary tree is a special type of tree that can be either empty or has a finite set of nodes, such that one of the nodes is designated as the root node and the remaining nodes are partitioned into two subtrees of the root node, known as left subtree and right subtree.
- A binary tree is said to be an extended binary tree (also known as 2-tree) if all of its nodes are of either zero or two degrees.
- Trees can be represented using an array or a linked list. An array representation binary trees are represented sequentially in memory, by using a single one-dimensional array. The linked representation of a binary tree is implemented by using a linked list having an info part and two pointers. The info part contains the data value and two pointers, left and right, are used to point to the left and right subtree of a node, respectively.
- A binary search tree, also known as a binary sorted tree, is a kind of a binary tree in which values in the left subtree of a root node are smaller than the value of the root node, and values in the right subtree are greater than the value of the root node.
- In a threaded binary tree, the NULL pointers are modified to point to their in-order successor or in-order predecessor or both. It can be of three types, right-threaded binary tree, left-threaded binary tree, and full-threaded binary tree.

6.9 Key Terms

- **Complete binary tree:** A binary tree in which all the leaf nodes of the tree are at the same level.
- **Traversing in a binary tree:** The process of visiting each and every node of the tree exactly once.
- **Depth (or height):** Depth of the binary tree is the highest level number of any node in a binary tree.
- **Degree of a node:** The degree of a node is equal to the number of its child nodes.
- **One-way threaded binary tree:** Threaded binary tree in which only one thread is used.

6.10 Check Your Progress

Short- Answer type

Q1) A binary tree could either have only a root node or have two disjoint binary trees called the left subtree or the right subtree. (True/ False?)

Q2) The nodes which have the same parent node are known as ………………

(i) Root (ii) Siblings (iii) Left nodes (iv) Right nodes

Q3) Differentiate between a one- way threaded binary tree and a two-way threaded binary tree.

Q4) When a binary search tree is traversed in _________, it results in a sequence of the elements in ascending order.

Q5) The node at the top of the tree is known as the _________ of the tree and the nodes at the lowest level are known as the ________.

Long- Answer type

Q1) Differentiate between an extended binary tree and a complete binary tree.

Q2) Write an algorithm for inserting an element into a binary search tree.

Q3) What are the different types of threaded binary trees? Explain in detail.

Q4) Define the following:

(i) Root node (ii) Leaf node (iii) Internal node (iv) Edge

Q5) Write short notes on the array and linked representations of a binary tree.

References

- *Classic Data Structures,* Debasis Samanta, PHI Learning Pvt. Ltd. 2nd Edition.
- *Data Structures with C,* Lipschutz, S. (2011), Delhi: Tata McGraw-Hill.
- *Data Structures and Algorithm Analysis in C*, Weiss. M. (1996), Addison Wesley Publications

Unit: 7 Graphs

Structure

7.0 Introduction
7.1 Unit Objectives
7.2 Graph Terminologies
7.3 Types of Graphs
    7.3.1 Classification on the basis of Edge Connectivity
    7.3.2 Classification on the basis of Direction
    7.3.3 Classification on the basis of Weight or Level
    7.3.4 Classification on the basis of Connectivity
7.4 Representation of Graphs
    7.4.1 Set Representation
    7.4.2 Linked Representation
    7.4.3 Matrix Representation
7.5 Graph Traversal Algorithms
    7.5.1 Breadth-First Search Algorithm
    7.5.2 Depth-first Search Algorithm
7.6 Shortest Path Algorithms
    7.6.1 Minimum Spanning Trees
    7.6.2 Prim's Algorithm
    7.6.3 Kruskal's Algorithm
    7.6.4 Dijkstra's Algorithm
7.7 Summary
7.8 Key Terms
7.9 Check Your Progress

7.0 Introduction

Another important non-linear data structure is Graphs. Graph structures can be easily related to the real world. For example, the airlines or railways network of the different cities can be represented using graph structures. The relation between the two variables can be depicted using different types of graph structures. Even the maps are also special kinds of graph structures that are very commonly used in day to day life.

A graph is a non-linear abstract data structure that is used to implement any relational or mathematical concepts of the entities. Simply, a graph is defined as a collection of vertices or nodes and edges. It is interesting to note that Trees are also a special kind of graph data structure. The relationship between the two nodes in a graph is less restricted than in trees. The nodes in trees follow one parent to many children relationship while in the case of graphs, the relationship of nodes is from many parents to many children.

This unit explains the fundamentals of graph data structures, different graph terminologies, representations, operations, and applications of graphs.

7.1 Unit Objectives

After going through this unit, the reader will be able to:

- Explain the fundamentals of Graphs.
- Understand its terminologies and different representations.
- Learn about the distinct operations of graph structures.
- Discuss the various applications of graphs.

7.2 Graph Terminologies

To learn about various terminologies of a graph structure, let us consider a graph G with an ordered set (V, E), where V(G) depicts the set of vertices and E(G) depicts the edges that connect the vertices. Figure 7.1 represents graph G with V(G) = {A, B, C, D and E} and E(G) = {(A, B), (B, C), (A, D), (B, D), (D, E), (C, E)}. It should be noted that there are five vertices or nodes and six edges in the graph.

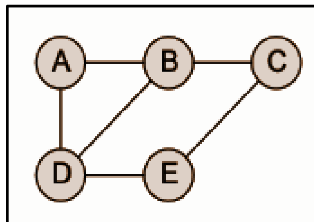


Figure 7.1 An example of a graph

Some of the important terms regarding trees are discussed below.

- **Adjacent nodes or neighbors:** For every edge, E = (p, q) that connects nodes p and q, the nodes p and q are said to be the adjacent nodes or neighbors.
- **Degree of a node:** Degree of a node p, deg(p), is the total number of edges

containing the node p. If deg(p) = 0, it means that p does not belong to any edge and such a node is known as an *isolated node*.

- **Regular graph:** A graph where each vertex has the same number of neighbors i.e., every node has the same degree. A regular graph with vertices of degree k is called a *k–regular graph* or *a regular graph of degree k*. Figure 7.2 shows the regular graphs.

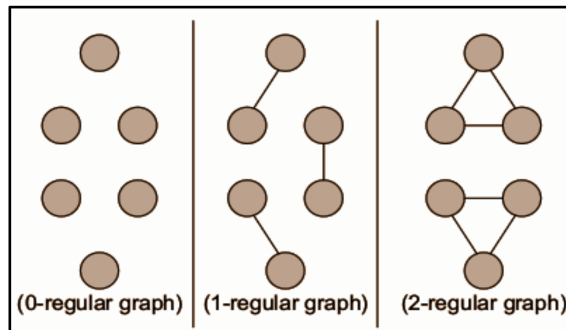


Figure 7.2 Regular Graphs

- **Path:** A path P = {v_0 , v_1 , v_2 , ..., v_n), of length n from a node u to v is defined as a sequence of (n+1) nodes. Here, u = v_0, v = v_n and v_{i+1} is adjacent to v_i for i = 1, 2, 3,..,n.
- **Closed path:** A path P is termed as a closed path if the edge has the same end-points, i.e. if v_0 = v_n.
- **Simple path:** A path P is termed as a simple path if all the nodes in the path are distinct with the exception that v_0 may be equal to v_n. If v_0 = v_n, then the path is called a *closed simple path.*
- **Cycle:** A path in which the first and the last vertices are the same, forms a cycle. There are no repeated edges or other vertices (except the first and last vertices) in a simple cycle.
- **Connected graph:** If any two vertices (u, v) in V are connected through a path from u to v, it is known as a connected graph. There are no isolated nodes in a connected graph. A connected graph with no cycle is called a tree.
- **Complete graph:** If all the vertices or nodes of a graph are fully connected, then it is said to be complete. It should be noted that in a complete graph, there is a path from one node to every other node. A complete graph has n(n–1)/2 edges, where n is the number of nodes in G.
- **Size of a graph:** The size of a graph is the total number of edges in it.
- **Multiple edges:** Distinct edges that connect the same end-points are called

multiple edges.

7.3 Types of Graphs

Graphs provide an advantage to represent complex data in the simplest forms. Accordingly, there are distinct graphs and their components in graph theory. There are different criteria to classify graphs.

7.3.1 Classification on the basis of Edge Connectivity

On the basis of edge connectivity, there are four types of graphs.

1. **Simple Graphs:** A simple graph connects only a pair of vertices or nodes in a graph containing vertices and edges. A simple graph is specified by its set of vertices and a set of edges. The set of edges is treated as a set of unordered pairs of vertices, like e=(u,v) (or e=(v,u)). Here, u & v are the endpoints of an edge, also known as adjacent nodes or neighbors.

   Simple graphs are most widely used in graph theory. Most of the algorithms and applications are developed using simple graphs only.
2. **Multi- Graphs:** A graph that contains multiple edges between a pair of vertices is called a multi- graph. It can be represented in figure 7.3.

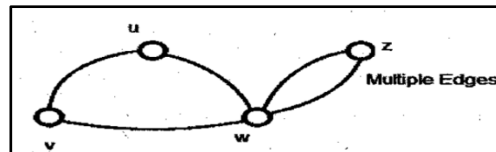


Figure 7.3 An example of Multigraph

3. **Graph with Loops:** A graph that permits loops that starts and ends at the same vertex is called a graph with loops. It can even contain self-loop, then it is known as a graph with self-loop. Figure 7.4 shows a graph with a self-loop at vertex v.

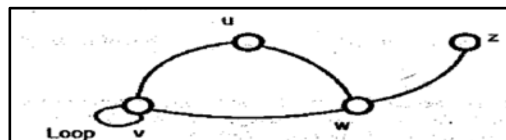


Figure 7.4 A graph with self-loop

4. **Hypergraph:** In a hypergraph, an edge can connect one or more vertices (even more than two). This edge is called hyperedge. Figure 7.4 shows a hypergraph with 5 vertices and 4 hyperedges.

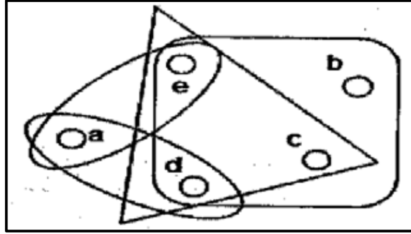


Figure 7.5 A hypergraph with 4 hyperedges

7.3.2 Classification on the basis of Direction

The graphs can be classified on the basis of the direction of the edges. The origin of an edge is from one vertex and ends at another vertex. An arrow indicates the direction of an edge. According to the direction, graphs can be directed or undirected.

1. **Directed Graphs:** In a directed graph, an arrow at the end vertex of an edge indicates its direction. The edge is considered as in-degree of the vertex where the arrow of the edge ends and the edge is considered as out-degree where the edge starts. Figure 7.6 shows a directed graph.

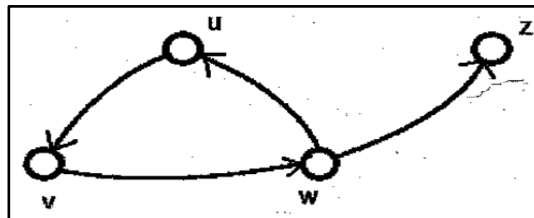


Figure 7.6 A directed graph

2. **Undirected Graphs:** In an undirected graph, there is no direction of the edge. The edge is considered in the degree of both the vertices. When not specified, the graph is considered to be an undirected graph. Figure 7.3 is an undirected graph where the direction of the edge is not specified.

7.3.3 Classification on the basis of Weight or Label

According to the requirement of any problem, the edges of the graphs are assigned a certain value or number that is termed as the weight or label of that edge. The total weight of the graph can also be calculated when required. On the basis of weight, graphs can be classified into four types.

1. **Unlabelled Graphs:** In an unlabelled graph, the vertices are not assigned any names rather each vertex is treated equal.
2. **Labelled Graphs:** In a labelled graph, a unique name is assigned to every

vertex of the graph.

3. **Unweighted Graphs:** In an unweighted graph, the edges are connected to the adjacent vertices only. The edges or vertices are not assigned any number.
4. **Weighted Graphs:** In a weighted graph, each vertex or edge is assigned a number. This number represents a parameter of interest. Depending on the problem, this number could be distance, cost, capacity, etc. The weighted graphs can be further classified into six types, as discussed below:
   - Edge weighted graph: When the weight is associated with the edges of the graph, then it is an edge-weighted graph. It is shown in figure 7.7(a).
   - Vertex weighted graph: When the weight is associated with the vertices of the graph, then it is a vertex weighted graph. It is shown in figure 7.7(b).
   - Positive weights: The weights associated with edges or vertices are positive integers then they are positive weights. In figure 7.7(a), all edges except edge(w,z) represent positive weights.
   - Negative weights: The weights associated with edges or vertices are negative integers then they are negative weights. In figure 7.7(a), edge(w,z) represents negative weights.
   - Additive weights: There is a need to compute the total weight of the graph while traversing the edges or vertices of the graph. For additive computation, the weights associated with the edges or vertices are added. For example, in figure 7.7(a), traversing u to v to w costs 12 if the weights are additive.
   - Multiplicative weights: The weights of the edges or vertices of a graph can be multiplicative too. In other words, the total cost of traversal can be computed by multiplying weights of edges or vertices. For example, in figure 7.7(a), traversing u to v to w costs 20 if the weights are multiplicative.

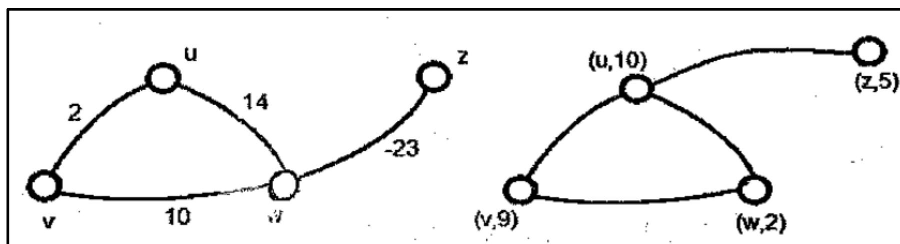


Figure 7.7 (a) Edge weighted graph (b) Vertex weighted graph

7.3.4 Classification on the basis of Connectivity

On the basis of connectivity, the graphs can be classified as connected and disconnected graphs.

1. **Connected Graphs:** A graph is said to be a connected graph when every vertex is reachable from any other vertex by traversing the edges. A connected graph is shown in figure 7.3 in which every vertex is reachable from every other vertex through a set of edges. For example, vertex z is reachable from vertex v through edges (v,w) and (w,z).
2. **Disconnected Graphs:** A graph in which certain vertices are not reachable from other vertices through any set of edges, is known as a disconnected graph. Figure 7.8 shows a disconnected graph in which vertex b and d are not connected through any set of edges of the graph.

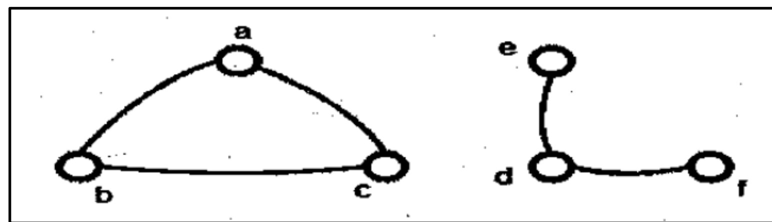


Figure 7.8 A disconnected graph

7.4 Representation of Graphs

The graphs can be represented in various ways in the computer's memory. Some of these graph representations are discussed below.

7.4.1 Set Representation

It is one of the simplest and straightforward methods of representation of graphs. In this method, two sets are maintained for edges and vertices. V is the set of vertices and E is the set of edges. Both edges and vertices are subsets of $V \, x \, V$. If the graph is weighted, the set E will be represented as $E = W \, x \, V \, x \, V$, where W is the set of weights. Consider an example in figure 7.9 for different types of graphs.

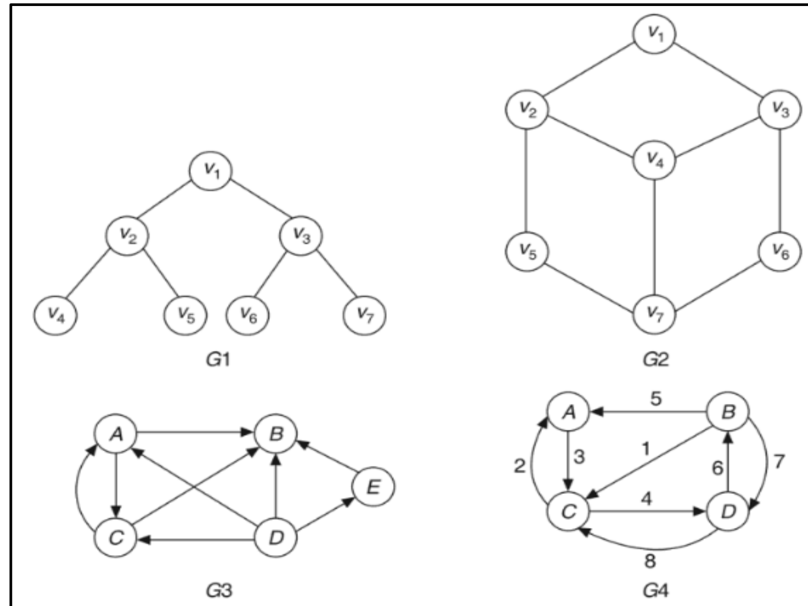


Figure 7.8 An example of different types of graphs

(Source- Classic Data Structures, Debasis Samanta, PHI Learning Pvt. Ltd. 2nd Edition, Chapter- 8, Page No.- 423**)**

These graphs in figure 7.8 can be represented using the set representation method, as below:

Graph G1

V(G1) = $\{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$

E(G1) = $\{(v_1, v_2), (v_1, v_3), (v_2, v_4), (v_2, v_5), (v_3, v_6), (v_3, v_7)\}$

Graph G2

V(G2) = $\{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$

E(G2) = $\{(v_1, v_2), (v_1, v_3), (v_2, v_4), (v_2, v_5), (v_3, v_4), (v_3, v_6), (v_4, v_7), (v_5, v_7), (v_6, v_7)\}$

Graph G3

V(G3) = {A, B, C, D, E}

E(G3) = {(A, B), (A, C), (C, B), (C, A), (D, A), (D, B), (D, C), (D, E), (E, B)}

Graph G4

V(G4) = {A, B, C, D}

E(G4) = {(3, A, C), (5, B, A), (1, B, C), (7, B, D), (2, C, A), (4, C, D), (6, D, B), (8, D, C)}

Note: The set representation method does not allow to store parallel edges if the graph is a multigraph and undirected graph. It is so as in a set two identical elements are not allowed. Though it is a straightforward approach, it is not useful for

manipulation of the graph.

7.4.2 Linked Representation

Another space-saving approach to graph representation is Linked representation. In this method, two types of node structures are assumed, as shown in figure 7.9 for unweighted and weighted graphs.

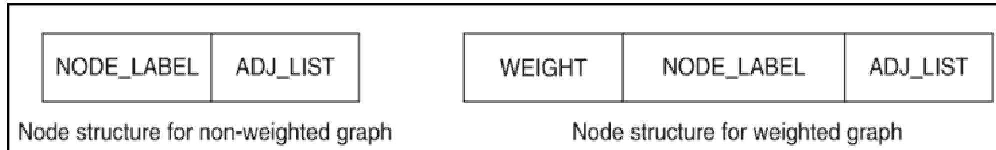


Figure 7.9 Node structures in Linked representation

Source: Classic Data Structures, Debasis Samanta, PHI Learning Pvt. Ltd. 2nd Edition

This method uses an ***adjacency list*** that contains the list of all the vertices/ nodes. ADJ_LIST in figure 7.9 represents an adjacency list. Every vertex is in turn linked to its own list that includes the names of all the vertices that are adjacent to it.

An adjacency list has the following advantages:

- It is easy to follow an adjacency list. It clearly displays the adjacent vertices of a particular vertex.
- Adding new vertices in an adjacency list is easier than in an adjacency matrix.
- An adjacency list is generally preferred for sparse graphs that have a small-to-moderate number of edges. For a large number of edges, the adjacency matrix is preferred.

Consider the types of graphs shown in figure 7.8. The linked representation of these graphs is shown in figure 7.10.

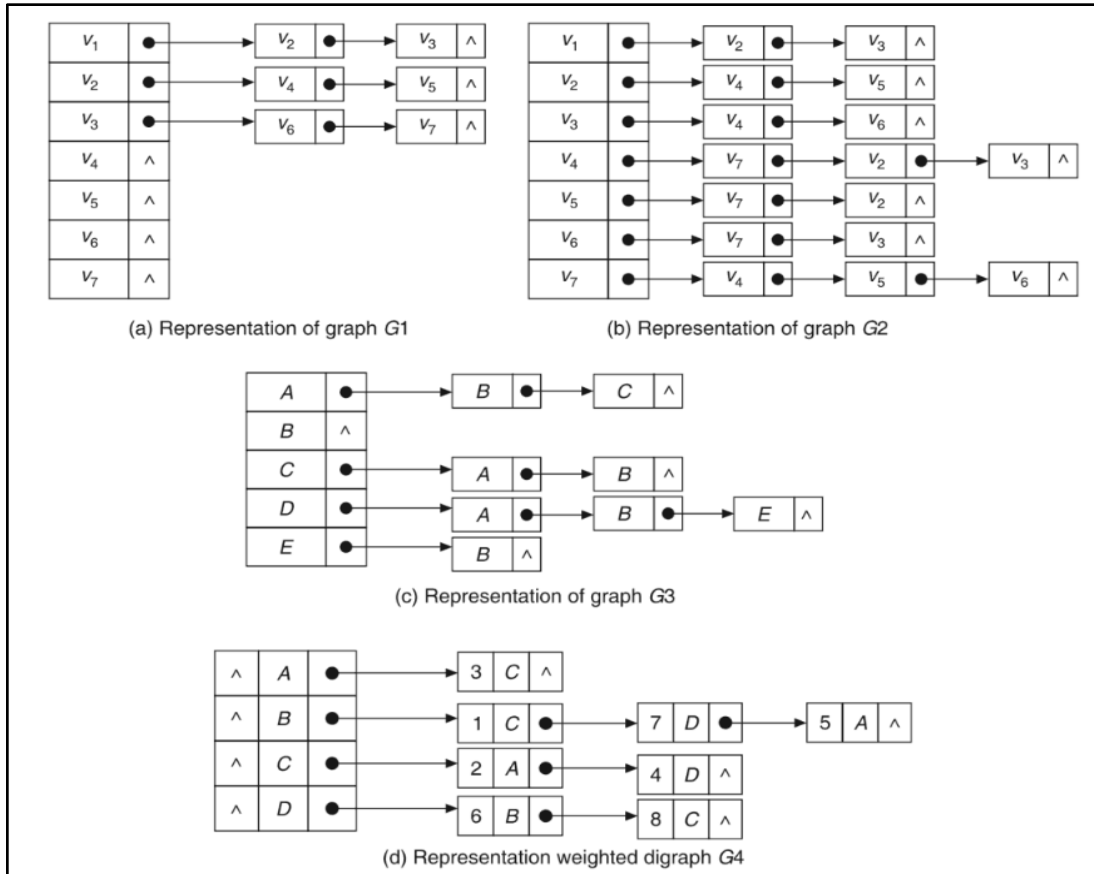


Figure 7.10 Linked representation of graphs of figure 7.8

(**Source**- Classic Data Structures, Debasis Samanta, PHI Learning Pvt. Ltd. 2nd Edition, Chapter- 8, Page No.- 424)

Program 7.1 Write a program to create a graph of n vertices using an adjacency list. Also write the code to read and print its information and finally to delete the graph.

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
struct node
{

        char vertex;
        struct node *next;
};
struct node *gnode;
void displayGraph(struct node *adj[], int no_of_nodes);
```

```
void deleteGraph(struct node *adj[], int no_of_nodes);
void createGraph(struct node *adj[], int no_of_nodes);
int main()
{

        struct node *Adj[10];
        int i, no_of_nodes;
        clrscr();
        printf("\n Enter the number of nodes in G: ");
        scanf("%d", &no_of_nodes);
        for(i = 0; i < no_of_nodes; i++)
                Adj[i] = NULL;
        createGraph(Adj, no_of_nodes);
        printf("\n The graph is: ");
        displayGraph(Adj, no_of_nodes);
        deleteGraph(Adj, no_of_nodes);
        getch();
        return 0;
}
void createGraph(struct node *Adj[], int no_of_nodes)
{

        struct node *new_node, *last;
        int i, j, n, val;
        for(i = 0; i < no_of_nodes; i++)
        {
                last = NULL;
                printf("\n Enter the number of neighbours of %d: ", i);
                scanf("%d", &n);
                for(j = 1; j <= n; j++)
                {
                        printf("\n Enter the neighbour %d of %d: ", j, i);
                        scanf("%d", &val);
                        new_node = (struct node *) malloc(sizeof(struct node));
                        new_node –> vertex = val;
                        new_node –> next = NULL;
                        if (Adj[i] == NULL)
```

```
                    Adj[i] = new_node;
                else
                    last –> next = new_node;
                last = new_node
            }
        }
}
void displayGraph (struct node *Adj[], int no_of_nodes)
{

        struct node *ptr;
        int i;
        for(i = 0; i < no_of_nodes; i++)
        {
            ptr = Adj[i];

            printf("\n The neighbours of node %d are:", i);
            while(ptr != NULL)
            {
                printf("\t%d", ptr –> vertex);
                ptr = ptr –> next;
            }
        }
}
void deleteGraph (struct node *Adj[], int no_of_nodes)
{

        int i;
        struct node *temp, *ptr;
        for(i = 0; i <= no_of_nodes; i++)
        {
            ptr = Adj[i];
            while(ptr ! = NULL)
            {
                temp = ptr;
                ptr = ptr –> next;
                free(temp);
```

```
                }

                Adj[i] = NULL;
        }
}
```

The Output of the program is:

Enter the number of nodes in G: 3
Enter the number of neighbours of 0: 1
Enter the neighbour 1 of 0: 2
Enter the number of neighbours of 1: 2
Enter the neighbour 1 of 1: 0
Enter the neighbour 2 of 1: 2
Enter the number of neighbours of 2: 1
Enter the neighbour 1 of 2: 1
The neighbours of node 0 are: 1
The neighbours of node 1 are: 0 2
The neighbours of node 2 are: 0

Note: If the graph in the above program had been a weighted graph, then the structure of the node would have been:

```
typedef struct node
{
        int vertex;
        int weight;
        struct node *next;
};
```

7.4.3 Matrix Representation

The most useful way of representing any graph is Matrix representation. In this method, a square matrix of order *n x n* is used, where n represents the number of vertices in the graph. This matrix is known as an *adjacency matrix* as an entry in the matrix stores the information whether two vertices are adjacent or not. As we know,

the adjacent nodes are those that have a common edge connecting them. This matrix is also known as a bit *matrix* or *Boolean matrix* as the entered values are either 0 or 1.

The entries in the matrix can be decided as per the following conditions:

a_{ij} = 1, if there is an edge from v_i to v_j

= 0, otherwise

The adjacency matrix is preferred for storing multigraphs and weighted graphs. For multigraphs, the entry will be according to the number of edges between two vertices, instead of entry 1. For weighted graphs, the entries in the matrix will be according to the weights of the edges, instead of 0 or 1.

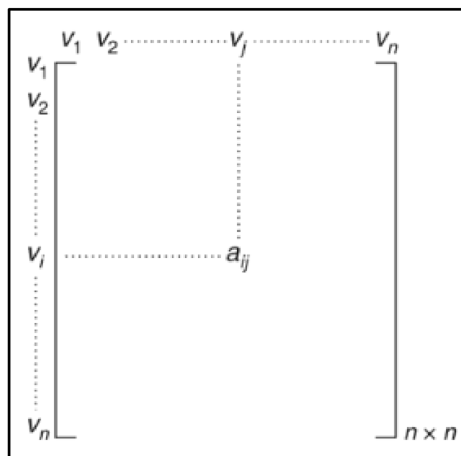
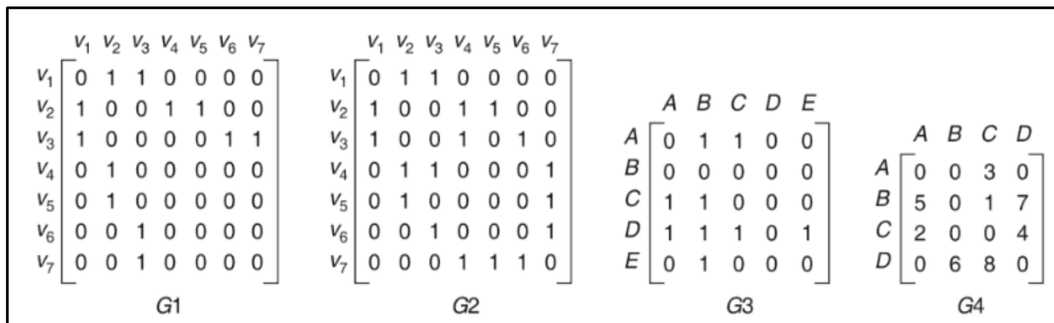


Figure 7.11 Matrix representation of graph

Source: Classic Data Structures, Debasis Samanta, PHI 2nd Edition, Chapter- 8, Page No.- 425

Consider the types of graphs shown in figure 7.8. The matrix representation of these graphs is shown in figure 7.12.



G1:

	v_1	v_2	v_3	v_4	v_5	v_6	v_7
v_1	0	1	1	0	0	0	0
v_2	1	0	0	1	1	0	0
v_3	1	0	0	0	0	1	1
v_4	0	1	0	0	0	0	0
v_5	0	1	0	0	0	0	0
v_6	0	0	1	0	0	0	0
v_7	0	0	1	0	0	0	0

G2:

	v_1	v_2	v_3	v_4	v_5	v_6	v_7
v_1	0	1	1	0	0	0	0
v_2	1	0	0	1	1	0	0
v_3	1	0	0	1	0	1	0
v_4	0	1	1	0	0	0	1
v_5	0	1	0	0	0	0	1
v_6	0	0	1	0	0	0	1
v_7	0	0	0	1	1	1	0

G3:

	A	B	C	D	E
A	0	1	1	0	0
B	0	0	0	0	0
C	1	1	0	0	0
D	1	1	1	0	1
E	0	1	0	0	0

G4:

	A	B	C	D
A	0	0	3	0
B	5	0	1	7
C	2	0	0	4
D	0	6	8	0

Figure 7.12 Matrix representation of graphs of figure 7.8

Source: Classic Data Structures, Debasis Samanta, PHI 2nd Edition, Chapter- 8, Page No.- 425

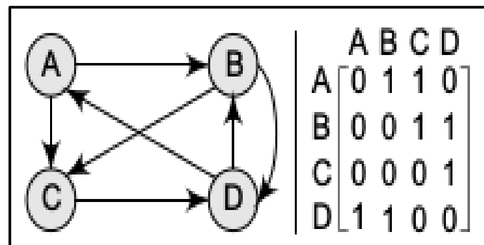
Following conclusions can be drawn from figure 7.12:

- For a simple graph (that has no loops), the adjacency matrix has 0s on the diagonal.
- The adjacency matrix of an undirected graph is symmetric.
- Number of 1s (or non-zero entries) in an adjacency matrix is equal to the number of edges in the graph.
- The adjacency matrix for a weighted graph contains the weights of the edges connecting the nodes.

Power of adjacency matrix: We already know that the adjacency matrix A^1 means that an entry 1 in the ith row and j^{th} column is due to an edge of length 1 from v_i to v_j. Now consider, A^2, A^3, and A^4.

$$(a_{ij})^2 = \sum a_{ik}\, a_{kj}$$

Any entry $a_{ij} = 1$ if $a_{ik} = a_{kj} = 1$. It may be concluded that if there are two edges (v_i, v_k) and (v_k, v_j) then the length is 2. Similarly the power of the adjacency matrix will be defined according to the number of edges between two adjacent nodes. Generally, every entry in the ith row and j^{th} column of A^n (where n is the number of nodes in the graph) gives the number of edges of length n from node v_i to v_j. Figure 7.13 shows a directed graph with its adjacency matrix and computation of adjacent matrices for different powers.



(a)

$$A^2 = A^1 \times A^1$$

$$A^2 = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 2 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 2 & 1 \end{bmatrix}$$

$$A^3 = \begin{bmatrix} 0 & 0 & 1 & 2 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 2 & 1 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 2 & 0 & 1 \\ 1 & 2 & 2 & 1 \\ 0 & 1 & 2 & 1 \\ 1 & 2 & 2 & 3 \end{bmatrix}$$

$$A^4 = \begin{bmatrix} 2 & 2 & 0 & 1 \\ 1 & 2 & 2 & 1 \\ 0 & 1 & 2 & 1 \\ 1 & 2 & 2 & 3 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 3 & 4 & 2 \\ 1 & 2 & 3 & 4 \\ 1 & 1 & 1 & 3 \\ 3 & 4 & 3 & 4 \end{bmatrix}$$

(b)

Figure 7.13 (a) Directed graph with its adjacency matrix (b) Adjacency matrices for A^2, A^3, and A^4

Source: Data Structures using C, Reema Thareja, Oxford University Press, 2nd Edition, Chapter- 13, Page No.- 389

If we define matrix B as:

$$B_r = A_1 + A_2 + A_3 + \ldots + A_r$$

An entry in the ith row and j^{th} column of matrix B_r gives the number of edges of length r from vertex v_i to v_j. This matrix B is used to obtain the ***path matrix (P)***. The path matrix P can be calculated from B by setting an entry $P_{ij} = 1$, if B_{ij} is non-zero and $P_{ij} = 0$, otherwise.

$$P_{ij} = 1, \text{ if } B_{ij} \text{ is non-zero}$$
$$= 0, \text{ otherwise}$$

The path matrix is used to show whether there is an edge from node v_i to v_j or not.

7.5 Graph Traversal Algorithms

The method of visiting each vertex and edge of a graph for at least once is known as traversing a graph. Following points should be noted for a graph:

- There is no first node or root in a graph. So, the graph can be started at any node.
- A particular node can be visited repeatedly. Hence, it becomes essential to note the status of the nodes that are traversed or not.
- Only those nodes can be traversed that are accessible from the current node.

The path of the traversing graph can be determined stepwise.

- In a graph, more than one edge is available, to reach a particular node.

Two standard methods are used for graph traversal:

1. Breadth-first search
2. Depth-first search

In breadth-first search method, a queue is used to store the vertices for further processing while in depth-first search method, a stack is used for this purpose. Both the algorithms use a variable STATUS that is set to 1 or 2 for every node, depending on its current state, during the execution. Table 7.1 describes the value and significance of the STATUS variable.

Table 7.1 Value and Significance of the STATUS variable

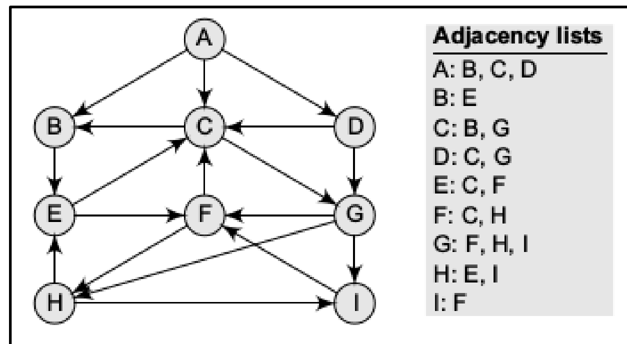
Status	State of the node	Description
1	Ready	The initial state of the node N
2	Waiting	Node N is placed on the queue or stack and waiting to be processed
3	Processed	Node N has been completely processed

7.5.1 Breadth-First Search Algorithm

The breadth-first search algorithm is accomplished by using a queue. The algorithm begins at a particular node that is considered as a root node and all the neighboring nodes are explored. Then for each of those nearest nodes, all other unexplored neighboring nodes are explored, and so on. For instance, if we start with node P, all neighbors of P are explored and then the neighbors of the neighbors of P are examined. The algorithm moves on until all the nodes are explored only once and not repeated. The queue helps in tracking and holding the waiting nodes for processing. The variable STATUS is used to represent the current state of the particular node.

Algorithm 7.1 Breadth-first search Algorithm
Step-1: Set STATUS = 1 (ready state) for every node in G. *Step-2: Enqueue the starting node A and set its STATUS = 2 (waiting state).* *Step-3: Repeat Steps 4 and 5 until QUEUE is empty.* *Step-4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).* *Step-5: Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)* *[END OF LOOP]* *Step 6: EXIT*

Example 7.1 Consider the graph G given below. Find the minimum path *P* from *A* to *I* given that every edge has a length of 1.



Solution: In the given figure, the minimum path *P* can be calculated with the help of the breadth-first search algorithm beginning at node A till node I. Here, we are using two arrays: QUEUE and ORIG. QUEUE is used to hold the nodes to be processed while ORIG is used to track the origin of each edge. Initially, FRONT = REAR = -1. The algorithm is as follows:

Step-1: Add *A* to QUEUE and NULL to ORIG.

FRONT = 0 QUEUE = A

REAR = 0 ORIG = \0

Step-2: Set FRONT = FRONT + 1 and remove the FRONT element of QUEUE (*Dequeue*) and add the neighbors of *A* (*Enqueue*). Also, add *A* to ORIG for the origin of the path of its neighbors.

FRONT = 1 QUEUE = A B C D

REAR = 3 ORIG = 0 A A A

Step-3: Set FRONT = FRONT + 1 and enqueue neighbors of *B*. Add *B* to ORIG of its neighbors.

FRONT = 2 QUEUE = A B C D E

REAR = 4 ORIG = 0 A A A B

Step-4: Again set FRONT = FRONT + 1 and enqueue the neighbours of *C*. Add *C* as the ORIG of its neighbours. Please note that *B* and *G* are two neighbours of *C*. Since *B* has already been added to the queue, we will not add *B* again and only add *G*.

FRONT = 3 QUEUE = A B C D E G

REAR = 5 ORIG = 0 A A A B C

Step-5: Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of *D*. Add *D* as the ORIG of its neighbours. Please note that *C* and *G* are two neighbours of *D*. Since both of them are already added to the queue, we will not add them again.

FRONT = 4 QUEUE = A B C D E G

REAR = 5 ORIG = 0 A A A B C

Step-6: Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of *E*. Add *E* as the ORIG of its neighbours. Please note that *C* and *F* are two neighbours of *E*. Since *C* has already been added to the queue, we will not add *C* again and only add *F*.

FRONT = 5 QUEUE = A B C D E G F

REAR = 6 ORIG = 0 A A A B C E

Step-7: Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of *G*. Add *G* as the ORIG of its neighbours. Please note that *F, H* and *I* are three neighbours of *G*. Since *F* has already been added to the queue, we will only add *H* and *I*. As our final goal was to reach *I,* the algorithm will be stopped here.

FRONT = 6 QUEUE = A B C D E G F H I

REAR = 9 ORIG = 0 A A A B C E G G

Now, to find the minimum path *P*, we have to backtrack using ORIG from *I*. The result obtained is ***A→ C→ G→ I***.

Program 7.2: Write a program to implement the breadth-first search algorithm.

```
#include <stdio.h>
#define MAX 10
void breadth_first_search(int adj[ ][MAX],int visited[ ],int start)
{
        int queue[MAX],rear = –1,front =– 1, i;
        queue[++rear] = start;
        visited[start] = 1;
        while(rear != front)
        {
                start = queue[++front];
                if(start == 4)
                        printf("5\t");
                else
                        printf("%c \t",start + 65);
        for(i = 0; i < MAX; i++)
                {
                        if(adj[start][i] == 1 && visited[i] == 0)
                        {
                                queue[++rear] = i;
                                visited[i] = 1;
                        }
                }
        }
}
int main()
{
        int visited[MAX] = {0};
        int adj[MAX][MAX], i, j;
        printf("\n Enter the adjacency matrix: ");
        for(i = 0; i < MAX; i++)
                for(j = 0; j < MAX; j++)
                        scanf("%d", &adj[i][j]);
```

```
breadth_first_search(adj,visited,0);
        return 0;
}
```

The output of the program is:

Enter the adjacency matrix:

0 1 0 1 0

1 0 1 1 0

0 1 0 0 1

1 1 0 0 1

0 0 1 1 0

A B C D E

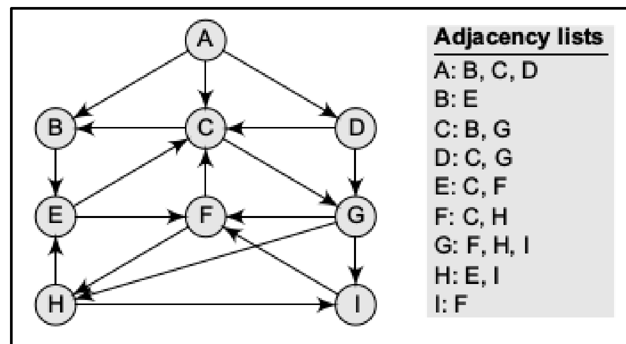
7.5.2 Depth-First Search Algorithm

The depth-first search algorithm is similar to the in-order traversal of a binary tree. The implementation of this algorithm is similar to that of the breadth-first search algorithm but a stack is used in place of a queue. Likewise, the STATUS variable is used to represent the current state of the node. In this method, the starting node of the graph G is expanded and then the process goes deeper and deeper until the goal node is achieved. The goal node can also be found when there are no children nodes for that node. On achieving the goal node, the algorithm backtracks and returns to the recent node that has not been explored completely. For instance, when the algorithm starts at node A, it becomes the current node. Then each node N is examined along a path P, beginning at A. That means, first we process node A, then its neighbors, and then the neighbors of the neighbors of A, and so on. If we reach a path that is associated with an already processed node N, then we backtrack to the current node. Otherwise, the unvisited node becomes the current node.

The algorithm continues to the dead-end, i.e. end of path *P* and after that we backtrack to find another path *P'*. When the backtracking leads back to the starting node A, the algorithm terminates. The edges that lead to new nodes are called *discovery edges* and the edges that lead to an already processed node are known as *back edges*.

Algorithm 7.2 Depth-first search Algorithm
Step-1: SET STATUS = 1 (ready state) for each node in G. *Step-2: Push the starting node A on the stack and set its STATUS = 2 (waiting state).* *Step-3: Repeat Steps 4 and 5 until STACK is empty.* *Step-4: Pop the top node N. Process it and set its STATUS = 3 (processed state).* *Step-5: Push on the stack all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state).* *[END OF LOOP]* *Step-6: EXIT*

Example 7.2: Consider the graph G of example 7.1, If we need to print all the nodes that can be reached from node H (including H) using the depth-first search algorithm starting at node H. The solution will be as follows.



Solution: In depth-first search algorithm, we use a STACK. The algorithm is as follows:

Step-1: Push *H* onto the STACK.

STACK: H

Step-2: Pop and print the top element of the STACK, i.e. *H.* Push all the neighbors of *H* on the STACK.

PRINT : H STACK: E, I

Step-3: Pop and print the top element of the STACK, i.e. *I.* Push all the neighbors of *I* on the STACK.

PRINT : I STACK: E, F

Step-4: Pop and print the top element of the STACK, i.e. *F*. Push all the neighbors of *F* on the STACK. It should be noted that *C* and *H* are two neighbors of but as *H* is already processed, only *C* will be added.

PRINT : F STACK: E, C

Step-5: Pop and print the top element of the STACK, i.e. *C*. Push all the neighbors of *C* on the STACK.

PRINT : C STACK: E, B, G

Step-6: Pop and print the top element of the STACK, i.e. *G*. Push all the neighbors of *G* on the STACK. Since none of the neighbors of *G* are in ready state, so no push operation is performed.

PRINT : G STACK: E, B

Step-7: Pop and print the top element of the STACK, i.e. *B*. Push all the neighbors of *B* on the STACK. Since none of the neighbors of *B* are in ready state, so no push operation is performed.

PRINT : B STACK: E

Step-8: Pop and print the top element of the STACK, i.e. *E*. Push all the neighbors of *E* on the STACK. Since none of the neighbors of *E* are in ready state, so no push operation is performed.

PRINT : E STACK:

As the STACK is now empty, the algorithm is terminated here and the nodes that were printed are:

H, I, F, C, G, B, E

The above printed nodes are reachable for H.

Program 7.3: Write a program to implement the depth-first search algorithm.

```
#include <stdio.h>
#define MAX 5
void depth_first_search(int adj[][MAX],int visited[],int start)
{
	int stack[MAX];
```

```
    int top = –1, i;
    printf("%c–",start + 65);
    visited[start] = 1;
    stack[++top] = start;
    while(top ! = –1)
    {
        start = stack[top];
        for(i = 0; i < MAX; i++)
        {
            if(adj[start][i] && visited[i] == 0)
            {
                stack[++top] = i;
                printf("%c–", i + 65);
                visited[i] = 1;
                break;
            }
        }
        if(i == MAX)
        top--;
    }
}
int main()
{
    int adj[MAX][MAX];
    int visited[MAX] = {0}, i, j;
    printf("\n Enter the adjacency matrix: ");
    for(i = 0; i < MAX; i++)
        for(j = 0; j < MAX; j++)
            scanf("%d", &adj[i][j]);
    printf("DFS Traversal: ");
    depth_first_search(adj,visited,0);
    printf("\n");
    return 0;
}
```

The output of the program is:

Enter the adjacency matrix:

0 1 0 1 0

1 0 1 1 0

0 1 0 0 1

1 1 0 0 1

0 0 1 1 0

DFS Traversal: A→ C→ E→

7.6 Shortest Path Algorithms

There are various algorithms that help in computing the shortest path between the vertices of a graph. Some of the important algorithms are discussed below are:

- Minimum Spanning Trees
- Prim’s Algorithm
- Kruskal’s Algorithm
- Dijkstra’s Algorithm

Let’s discuss each one of them in detail.

7.6.1 Minimum Spanning Trees

A spanning tree is a subgraph of a connected and undirected graph that connects all the vertices together. A single graph can have many spanning trees. We can even assign weights to each edge of the graph and ultimately assign weight to the spanning tree. It can be done by calculating the sum of the weights of the edges in that spanning tree.

A spanning tree with weight less than or equal to the weight of every other spanning tree, is known as a *minimum spanning tree (MST).*

Some important properties of spanning trees are:

- It is possible that there exist multiple minimum spanning trees of the same weight in a graph. It should be noted that every spanning tree will be considered minimum, if all the weights are the same.
- To obtain a unique minimum spanning tree, each edge of the graph is assigned a different weight.
- If the weights of the edges are non-negative, then the minimum spanning trees

is treated as the minimum-cost subgraph.

- If there is a cycle C in the graph G that has a larger weight than that of the other edges of C, then this edge is not included in the minimum spanning tree.
- Minimum spanning trees create a sparse subgraph that displays a lot about the original graph. It is easy and quick to compute MSTs and provide optimal solutions.
- The minimum spanning tree of a weighted graph consists of n-1 edges of minimum total weight of the graph. It should be noted that any spanning tree for an unweighted graph is a minimum spanning tree.

For example, figure 7.14 shows the eight spanning trees drawn from a unweighted graph G. There can be even more spanning trees. For an unweighted graph, every spanning tree is considered as a minimum spanning tree.

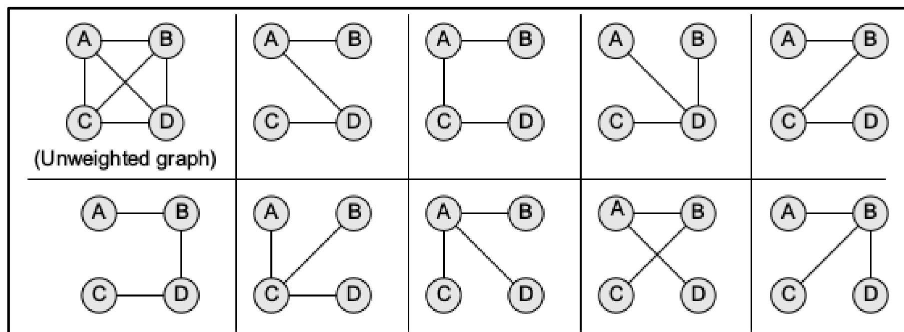


Figure 7.14 Unweighted graph and its spanning trees

Source: *Data Structures using C, Reema Thareja, Oxford University Press, 2nd Edition, Chapter- 13, Page No.- 406*

Another example can be considered for a weighted graph, as shown in figure 7.15. Distinct spanning trees can be drawn from the graph G. But, it should be noted that only one minimum spanning tree can be obtained. Here, the spanning tree with total cost = 9 is said to be the minimum spanning tree of weighted graph G..

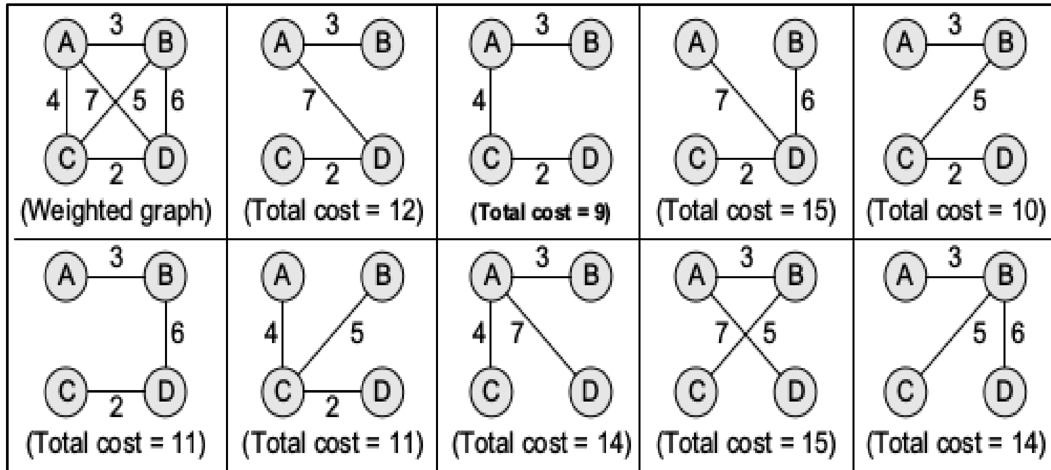


Figure 7.15 Weighted graph and its spanning trees

Source- Data Structures using C, Reema Thareja, Oxford University Press, 2nd Edition, Chapter- 13, Page No.- 407

Applications of Minimum Spanning Trees

- Minimum spanning trees are widely used for designing networks according to the requirement. For example, MSTs can be used to determine the least costly paths to deploy cable in a telephone network.
- MSTs can be used to optimize the cheapest airline routes, connect terminals for roads, railways, wires, etc.
- MSTs can be applied in routing algorithms to find the most effective path.

7.6.2 Prim's Algorithm

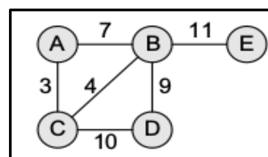
Prim's algorithm is used to draw a minimum spanning tree for a weighted- undirected graph. This algorithm forms a tree that includes every node and a subset of the edges, such that the total weight of all the edges of the tree is minimum. To accomplish this, three sets of vertices are maintained:

- ***Tree Vertices:*** The vertices that are a part of the minimum spanning tree.
- ***Fringe Vertices:*** The vertices that are adjacent but currently are not a part of tree vertices.
- ***Unseen Vertices:*** The vertices other than tree and fringe vertices are termed as unseen vertices.

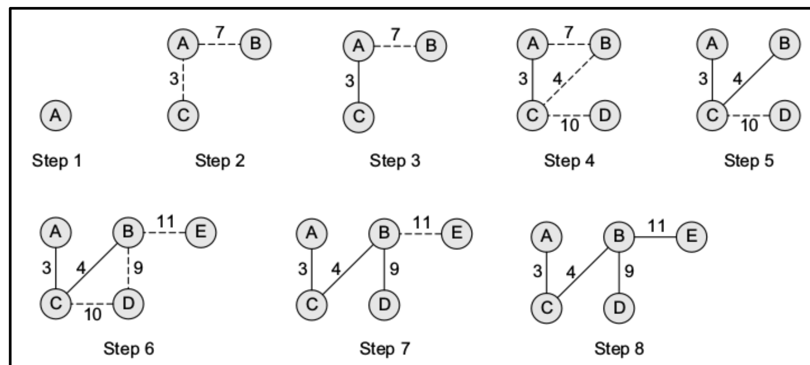
Algorithm 7.3 Prim's Algorithm
Step-1: Select a starting vertex *Step-2: Repeat Steps 3 and 4 until there are fringe vertices* *Step-3: Select an edge e connecting the tree vertex and fringe vertex that has minimum weight* *Step-4: Add the selected edge and the vertex to the minimum spanning tree T* *[END OF LOOP]* *Step-5: EXIT*

The method starts with choosing a starting vertex. The starting vertex is branched out and during each iteration, a new vertex and edge is selected. The vertex is selected from the fringe vertices such that minimum weight is assigned to the edge connecting the tree and new vertex. The running time of this algorithm can be computed from ***O (E log V)***, where V is the number of vertices and E is the number of edges in the graph.

For example, if we have to construct a minimum spanning tree of the graph given in figure 7.16 (a) using Prim's algorithm. Figure 7.16 (b) depicts the step by step process of forming a minimum spanning tree for the given graph.



(a)



(b)

Figure 7.16(a) Graph G (b) Minimum spanning tree of graph G using Prim's algorithm

Source: Data Structures using C, Reema Thareja, Oxford University Press, 2nd Edition, Chapter- 13, Page No.- 408

The steps for Prim's algorithm for given graph G in figure 7.16 (a) are as follows:

1. Choose a starting vertex *A*.
2. Add the fringe vertices that are adjacent to *A*. The edges connecting the starting vertex and fringe vertices are represented in figure 7.16 (b) with dotted lines.
3. Now, the edge connecting tree vertex and fringe vertex with minimum weight is selected. This edge and vertex is added to the minimum spanning tree T. Here, as shown, the edge connecting *A* & *C* has less weight, so *C* is added to the tree. Now, *C* is treated as a tree vertex not a fringe vertex.
4. Add the fringe vertices adjacent to *C*.
5. Repeat step-3. As the edge connecting *C* & *B* has less weight, so *B* is added to the tree and now *B* becomes a tree vertex and is no longer a fringe vertex.
6. Add the fringe vertices adjacent to *B*.
7. Repeat step-3. As the edge connecting *B* & *D* has less weight, so *D* is added to the tree and now *D* is now a tree vertex and is no longer a fringe vertex.
8. Note, now node *E* remains unconnected, so we will add it in the tree as a minimum spanning tree is the one in which all the n nodes are connected with n–1 edges that have minimum weight.

7.6.3 Kruskal's Algorithm

Kruskal's algorithm was first proposed by Joseph Kruskal in 1956. It is used to form a minimum spanning tree for a connected- weighted graph. This algorithm finds a subset of the edges forming a tree including every vertex, such that the total weight of all the edges of the tree is minimum. If the graph is an unconnected- weighted graph, then a *minimum spanning forest* is obtained. Minimum spanning forest is a collection of minimum spanning trees.

Algorithm 7.4 Kruskal's Algorithm
Step-1: Create a forest in such a way that each graph is a separate tree. *Step-2: Create a priority queue Q that contains all the edges of the graph.* *Step-3: Repeat Steps 4 and 5 while Q is NOT EMPTY.*

Step-4: Remove an edge from Q.

Step-5: IF the edge obtained in Step 4 connects two different trees, then add it to the forest (for combining two trees into one tree).

ELSE

Discard the edge

Step-6: END

The algorithm uses a priority queue, Q. In this queue, the edges with minimum weight have priority over other edges in the graph. On termination of the algorithm, the forest contains only one component that forms a minimum spanning tree of the graph. The running time of this algorithm can be computed from ***O (E log V)***, where V is the number of vertices and E is the number of edges in the graph.

For example, consider a graph given in figure 7.17 and apply Kruskal's algorithm on the graph.

Initially, we have F = {{A}, {B}, {C}, {D}, {E}, {F}}

MST = {}

Q = {(A, D), (E, F), (C, E), (E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}

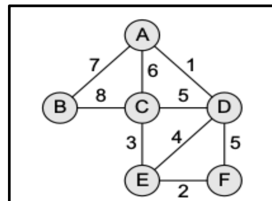


Figure 7.17 Graph G for Kruskal's algorithm application

Source: Data Structures using C, Reema Thareja, Oxford University Press, 2nd Edition, Chapter- 13, Page No.- 410

The steps for Kruskal's Algorithm for given graph G are:

1. Remove the edge (A, D) from Q and make the following changes:

   F = {{A, D}, {B}, {C}, {E}, {F}}

   MST = {A, D}

   Q = {(E, F), (C, E), (E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}

2. Remove the edge (E, F) from Q and make the following changes:

   F = {{A, D}, {B}, {C}, {E, F}}

   MST = {(A, D), (E, F)}

   Q = {(C, E), (E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}

3. Remove the edge (C, E) from Q and make the following changes:

   F = {{A, D}, {B}, {C, E, F}}

   MST = {(A, D), (C, E), (E, F)}

   Q = {(E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}

4. Remove the edge (E, D) from Q and make the following changes:

   F = {{A, C, D, E, F}, {B}}

   MST = {(A, D), (C, E), (E, F), (E, D)}

   Q = {(C, D), (D, F), (A, C), (A, B), (B, C)}

5. Remove the edge (C, D) from Q. It should be noted that this edge does not connect different trees, so it is simply discarded. Only an edge connecting (A, D, C, E, F) to B will be added to the MST. Therefore,

   F = {{A, C, D, E, F}, {B}}

   MST = {(A, D), (C, E), (E, F), (E, D)}

   Q = {(D, F), (A, C), (A, B), (B, C)}

6. Remove the edge (D, F) from Q. It should be noted that this edge does not connect different trees, so it is simply discarded. Only an edge connecting (A, D, C, E, F) to B will be added to the MST.

   F = {{A, C, D, E, F}, {B}}

   MST = {(A, D), (C, E), (E, F), (E, D)}

   Q = {(A, C), (A, B), (B, C)}

7. Remove the edge (A, C) from Q . Note that this edge does not connect different trees, so simply discard this edge. Only an edge connecting (A, D, C, E, F) to B will be added to the MST.

   F = {{A, C, D, E, F}, {B}}

   MST = {(A, D), (C, E), (E, F), (E, D)}

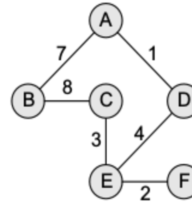
   Q = {(A, B), (B, C)}

8. Remove the edge (A, B) from Q and make the following changes:

   F = {A, B, C, D, E, F}

   MST = {(A, D), (C, E), (E, F), (E, D), (A, B)}

   Q = {(B, C)}

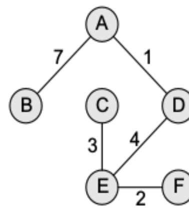


9. The algorithm continues until Q is empty. Since the entire forest has become one tree, all the remaining edges will simply be discarded.

   F = {A, B, C, D, E, F}

   MST = {(A, D), (C, E), (E, F), (E, D), (A, B)}

   Q = {}



Program 7.5: Write a program which finds the cost of a minimum spanning tree.

```
#include<stdio.h>
#include<conio.h>
#define MAX 10
int adj[MAX][MAX], tree[MAX][MAX], n;
void readmatrix()
{
	int i, j;
	printf("\n Enter the number of nodes in the Graph : ");
	scanf("%d", &n);
	printf("\n Enter the adjacency matrix of the Graph");
	for (i = 1; i <= n; i++)
		for (j = 1; j <= n; j++)
			scanf("%d", &adj[i][j]);
}
int spanningtree(int src)
{
	int visited[MAX], d[MAX], parent[MAX];
	int i, j, k, min, u, v, cost;
	for (i = 1; i <= n; i++)
	{
```

```
                d[i] = adj[src][i];
                visited[i] = 0;
                parent[i] = src;
        }
        visited[src] = 1;
        cost = 0;
        k = 1;
        for (i = 1; i < n; i++)
        {
                min = 9999;
                for (j = 1; j <= n; j++)
                {
                        if (visited[j]==0 && d[j] < min)
                        {
                                min = d[j];
                                u = j;
                                cost += d[u];
                        }
                }
                visited[u] = 1;
                //cost = cost + d[u];
                tree[k][1] = parent[u];
                tree[k++][2] = u;
                for (v = 1; v <= n; v++)
                        if (visited[v]==0 && (adj[u][v] < d[v]))
                        {
                                d[v] = adj[u][v];
                                parent[v] = u;
                        }
        }
        return cost;
}
void display(int cost)
{
        int i;
        printf("\n The Edges of the Minimum Spanning Tree are");
        for (i = 1; i < n; i++)
```

```
                printf(" %d %d \n", tree[i][1], tree[i][2]);
        printf("\n The Total cost of the Minimum Spanning Tree is : %d", cost);
}
main()
{
        int source, treecost;
        readmatrix();
        printf("\n Enter the Source : ");
        scanf("%d", &source);
        treecost = spanningtree(source);
        display(treecost);
        return 0;
}
```

The output of the program is:

```
Enter the number of nodes in the Graph : 4
Enter the adjacency matrix : 0   1   1   0
                             0   0   0   1
                             0   1   0   0
                             1   0   1   0
Enter the source: 1

The edges of the Minimum Spanning Tree are: 1   4
                                            4   2
                                            2   3
The total cost of the Minimum Spanning Tree is: 1
```

7.6.4 Dijkstra's Algorithm

A Dutch scientist Edsger Dijkstra in 1959 introduced Dijkstra's algorithm to find the shortest path tree. It is widely used in network routing protocols. This algorithm can be used to find the shortest path having the lowest cost between source node and destination node of graph G.

In Dijkstra's algorithm, the length of an optimal path between two nodes of a graph is computed. The term optimal here can refer to shortest, fastest or cheapest. The

algorithm starts with an initial/ source node and accordingly distance from initial node to any other node is calculated.

Algorithm 7.5 Dijkstra's Algorithm
Step-1: Select the source node also called the initial node. *Step-2: Define an empty set N that will be used to hold nodes to which a shortest path has been found.* *Step-3: Label the initial node with , and insert it into N.* *Step-4: Repeat Steps 5 to 7 until the destination node is in N or there are no more labelled nodes in N.* *Step-5: Consider each node that is not in N and is connected by an edge from the newly inserted node.* *Step-6: (a) If the node that is not in N has no label then SET the label of the node = the label of the newly inserted node + the length of the edge.* *(b) Else if the node that is not in N was already labelled, then SET its new label = minimum (label of newly inserted vertex + length of edge, old label)* *Step-7: Pick a node not in N that has the smallest label assigned to it and add it to N.*

In this algorithm, every node in the graph is labelled as the distance (cost) from the source node to that node. Labels can be of two types: *Temporary and Permanent.* The nodes that have not been reached, are assigned as Temporary labels, while permanent labels are for those nodes that have been reached and their distance to the source node is also known. A node can be labelled either temporary or permanent but not both.

Executing this algorithm can produce either of the following two results:

1. A labelled destination node will in turn represent the distance from the source node to the destination node.
2. A non- labelled destination node specifies that there is no path from the source to the destination node.

For example, consider a graph G in figure 7.18. The initial node is taken as D and Dijkstra's algorithm is applied to graph G. The steps for the same are given below:

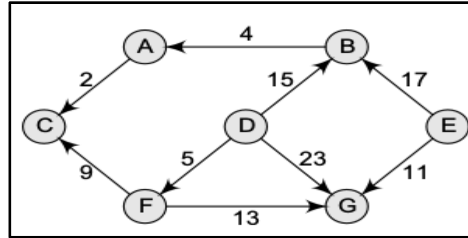


Figure 7.18 Graph G for Dijkstra's algorithm

Source: Data Structures using C, Reema Thareja, Oxford University Press, 2nd Edition, Chapter- 13, Page No.- 414

1. Set the label of D = 0 and N = {D}.
2. Label of D = 0, B = 15, G = 23 , and F = 5 . Therefore, N = {D, F}.
3. Label of D = 0, B = 15, G has been re-labelled 18 because minimum (5 + 13, 23) = 18, C has been re-labelled 14 (5 + 9). Therefore, N = {D, F, C}.
4. Label of D = 0, B = 15, G = 18 . Therefore, N = {D, F, C, B}.
5. Label of D = 0 , B = 15, G = 18 and A = 19 (15 + 4) . Therefore, N = {D, F, C, B, G}.
6. Label of D = 0 and A = 19 . Therefore, N = {D, F, C, B, G, A}.

Note that we have no labels for node E; this means that E is not reachable from D. Only the nodes that are in N are reachable from D. The running time of Dijkstra's algorithm can be given as $O(|V|^2+|E|)=O(|V|^2)$ where V is the set of vertices and E is the number of edges in the graph.

7.7 Summary

- A graph is a collection of vertices (or nodes) and edges that connect these vertices. Degree of a node p, deg(p), is the total number of edges containing the node p.
- Graphs can be classified on the basis of edge connectivity, direction, weight or label, and connectivity. The graphs can be represented using three methods; set representation, Linked representation using an adjacency list, and Matrix representation using an adjacency matrix.
- Two standard methods for graph traversal are Breadth-first search algorithm (BFS) and Depth-first search algorithm (DFS). The BFS algorithm is accomplished by using a queue while the DFS algorithm is accomplished using a stack.

- A spanning tree of a connected, undirected graph G is a sub-graph of G which is a tree that connects all the vertices together.
- Kruskal's algorithm is an example of a greedy algorithm, as it makes the locally optimal choice at each stage with the hope of finding the global optimum.
- Dijkstra's algorithm is used to find the length of an optimal path between two nodes in a graph.

7.8 Key Terms

- **Isolated Node:** A node with degree zero is known as an isolated node.
- **Graph with Loops:** A graph that permits loops that starts and ends at the same vertex is called a graph with loops.
- **Discovery edges:** In a DFS algorithm, the edges that lead to new nodes are called discovery edges.
- **Minimum Spanning Tree:** A spanning tree with weight less than or equal to the weight of every other spanning tree.
- **Minimum Spanning Forest:** It is a collection of minimum spanning trees.

7.9 Check Your Progress

Short- Answer type

Q1) Adjacency matrix is also known as a ______.

Q2) Graph is a linear data structure. (True/ False?)

Q3) The term optimal can mean-

(a) Shortest (b) Cheapest (c) Fastest (d) All of these

Q4) What is an adjacency matrix?

Q5) Define Minimum Spanning Tree.

Long- Answer type

Q1) Explain the graph traversal algorithms in detail with example.

Q2) Describe Prim's algorithm.

Q3) Write a brief note on:

(a) Kruskal's algorithm (b) Dijkstra's algorithm

Q4) Discuss the types of graphs in detail.

Q5) Given the adjacency matrix of a graph, write a program to calculate the degree of

a node N in the graph.

References

- *Data Structures using C,* Reema Thareja, Oxford University Press, 2nd Edition
- *Classic Data Structures,* Debasis Samanta, PHI Learning Pvt. Ltd. 2nd Edition.

Module: 3

Types of Trees

Unit 8 Balanced Trees

Structure

8.0 Introduction
8.1 Unit Objectives
8.2 Basic Terminology
8.3 AVL Trees
8.4 Weight Balanced Trees
8.5 Summary
8.6 Key Terms
8.7 Check Your Progress

8.0 Introduction

In the previous units, we have already discussed the binary trees and their terminologies in detail. A binary tree is a special type of tree, which can be either empty or has a finite set of nodes, such that one of the nodes is designated as the root node and the remaining nodes are partitioned into two subtrees of root node known as left subtree and right subtree. These left and right subtrees should not be empty and should be binary trees. Binary trees can be represented in the computer's memory using an array or a linked list.

Binary search trees are a kind of binary tree having values in the left subtree of a root node smaller than the value of the root node, and values in the right subtree greater than or equal to the value of the root node. Now, let's learn about balanced binary trees.

Balanced trees are a versatile set of data structures in which every leaf is "at a certain distance" from the root than any other leaf. As we already know that the maximum number of nodes possible from the root node to a leaf node is termed as the height of a tree. So, a binary tree is said to be balanced if the height of the tree is O(log n), where n is the number of nodes of the tree. This unit describes the basic terminology of balanced trees and also discusses different types of balanced trees in detail.

8.1 Unit Objectives

After going through this unit, the reader will be able to:

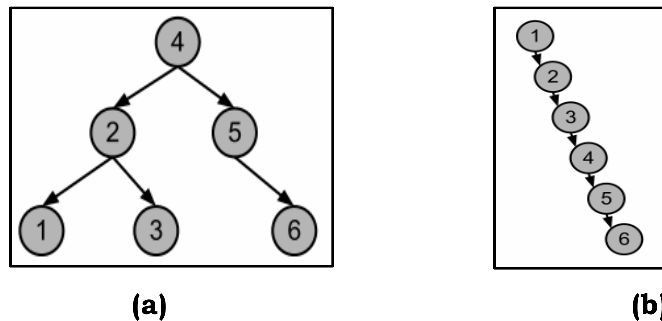
- Understand the fundamentals of Balanced binary search trees.

- Explain the basic terminology of Height balanced trees specifically AVL Trees.
- Learn about the properties of Weight-balanced Trees.

8.2 Basic Terminology

Balancing a binary search tree is beneficial as a balanced tree provides O(log n) time for all the operations like searching, inserting, and deleting. An unbalanced tree consumes more running time i.e. O(n) as the shape becomes distorted. It means that if one branch of the tree is much longer than the other, the operations take more running time. Figure 8.1 represents balanced and unbalanced binary trees.

The height of a binary tree is an important parameter in relation to the efficient operations on the tree. Searching, inserting, and deleting operations in a binary search tree are all O(Height). Generally, the height (H) and the number of nodes (n) are related to H = (log n). So, the effective operations become O(H) = O(log n).



(a) **(b)**

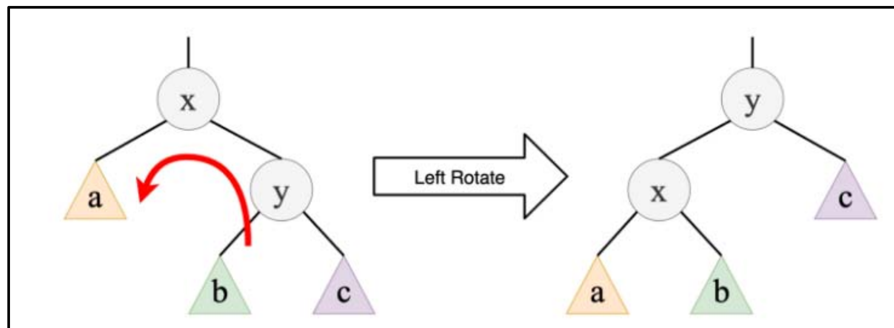
Figure 8.1 (a) Balanced Tree (b) Unbalanced Tree

Balancing of the binary trees can be achieved through different approaches. Either height or weight of the tree is balanced such that the average running time of the operations is maintained to O(log n). The balancing approach can be partial or complete as per requirement. Though it is difficult to attain a perfectly balanced binary tree.

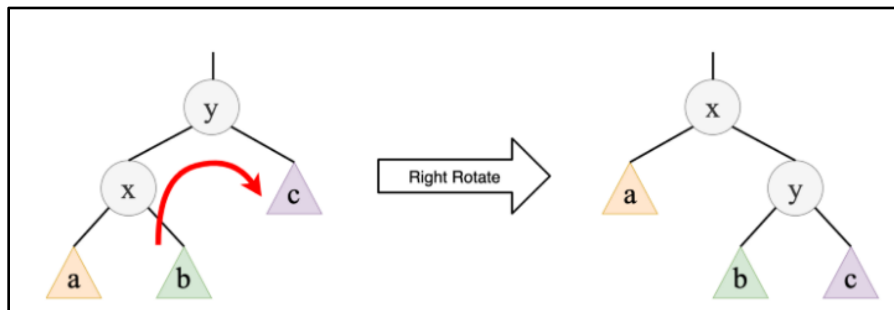
One of the most commonly used balancing approaches is ***self-balancing trees***. Such trees maintain balance automatically by keeping the height as small as possible during the insertion and deletion operation on the binary tree. The self-balancing binary search trees perform ***rotations*** to maintain the balance in the tree, even after the insert and delete operations. Two types of rotations are possible in a binary search tree without violating its in-order traversal property. Consider figure 8.1 (a) and (b) representing the left and right rotation of a binary tree.

a) **Left Rotation:** According to figure 8.2 (a), during the left rotation about node X, the new root of the subtree is now node Y. Node X becomes the left child of node Y while subtree B is now the right child of node X.

b) **Right Rotation:** In figure 8.2 (b), during the right rotation about node Y, the new root of the subtree is now node X. Node Y becomes the right child of node X while subtree B is now the left child of node Y.



(a)



(b)

Figure 8.2 (a) Left Rotation (b) Right Rotation of a binary search tree

Source: https://towardsdatascience.com/self-balancing-binary-search-trees-101-fc4f51199e1d

The balanced binary search trees are classified into two groups, i.e. Height balanced trees and Weight balanced trees. In height-balanced trees, the height of the siblings of a node is "approximately the same". In weight balanced trees, the number of descendants of sibling nodes is "approximately the same". Different types of height-balanced binary search trees are there. Some of them like AVL trees, Red-black Trees, Splay Trees will be discussed in this course.

8.3 AVL Trees

AVL Trees are one of the most commonly known self-balancing binary search trees. These trees were first introduced by two mathematicians *G.M.* ***A****delson-****V****elsky and*

E.M. ***L****andis* in 1962. AVL tree is named so in honor of its inventors. Being a self-balanced binary search tree, an AVL tree takes O(log n) time to perform the search, insert, and delete operations. That means the height of the AVL tree is limited to O(log n). The key property for AVL trees is that the heights of the two subtrees of one node may differ by at most one. Due to this, the AVL tree is also termed a ***height-balanced tree***.

There is a little difference between the structure of an AVL tree and a simple binary search tree. In an AVL tree, the additional variable ***"balance factor"*** is associated with each node. The balance factor of a node is the difference between the height of the right subtree and the height of the left subtree.

Balance factor = Height (left subtree) – Height (right subtree)

In a height-balanced tree, every node has a balance factor of -1, 0, or 1. Any other value of the balance factor makes the binary search tree unbalanced. Some important key points about balance factor are:

- If a node has a balance factor of 1, it means that the right subtree of the node is one level lower than the left subtree. Such a tree is called a *lefty-heavy tree* as shown in figure 8.3 (a).
- If a node has a balance factor of 0, it means that the height of the left subtree is equal to the height of the right subtree.
- If a node has a balance factor -1, it means that the right subtree of the node is one level higher than the left subtree. Such a tree is called a *right-heavy tree* as shown in figure 8.3 (b).

In figure 8.3, it should be noted that the balance factor of nodes 18, 39, 54, 63, and 72 is 0; the balance factor for nodes 27, 36, and 45 is 1.

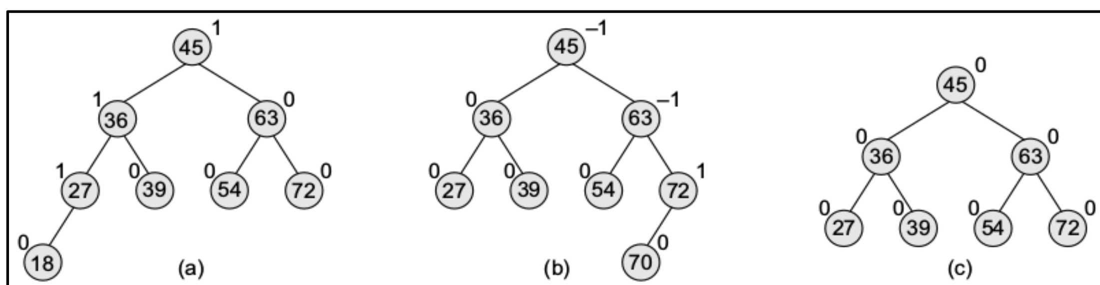


Figure 8.3 (a) Left-heavy AVL Tree (b) Right-heavy AVL Tree (c) Balanced Tree

Source: *Data Structures using C, Reema Thareja, Oxford University Press, 2nd Edition, Chapter- 10, Page No.- 317*

The insertion and deletion operations on the AVL tree may imbalance the tree, resulting in disturbing the balance factor of the nodes. In such cases, the tree is rebalanced using rotation operation at the critical node.

Searching for a Node in an AVL Tree

The search operation is the same for both AVL trees and binary search trees. The structure of the tree does not modify, so no special provisions are required. According to the property, the running time taken by the search operation to be completed is O(log n).

Inserting a New Node in an AVL Tree

The insert operation in an AVL tree is similar to the one in binary search trees. In an AVL tree, the new node is always inserted as the leaf node and the insertion step is generally followed by an additional step, i.e. rotation. Rotation is helpful in rebalancing the tree. Though, if the balance factor does not get disturbed due to insert operation, i.e. it is still -1, 0, or 1, then there is no need for the rotation. In AVL trees, the new node is always inserted as a leaf node, so the balance factor will always be 0. Change in balance factor can be observed only for those nodes that are in the path of the root node and newly inserted node. Some possible changes in the path for any node are discussed below:

- Initially, the node of an AVL tree was either left-heavy or right-heavy. After the insertion of a new node, it becomes balanced.
- The node that was balanced initially, becomes either left-heavy or right-heavy after inserting a new node.
- The node that was either left-heavy or right-heavy initially, becomes unbalanced due to a new node insertion. Such a node is termed a *critical node.* The nearest ancestor node on the path between the inserted node and the root with balance factor neither -1, 0, nor 1 is the critical node.

Four types of rotations are generally used after insert operation in an AVL tree, they are LL rotation, RR rotation, LR rotation, RL rotation.

1. **LL rotation:** The new node is inserted in the left subtree of the left subtree of the critical node. LL rotation in an AVL tree is shown in figure 8.4. In this

figure, node A is considered as a critical node, as it is the nearest ancestor whose balance factor is not -1, 0, or 1. After insert operation, the new node has now become the part of tree T_1. Now, during LL rotation, node B becomes the root; T_1 and A are its left and right child respectively; T_2 and T_3 are now left and right subtrees of A.

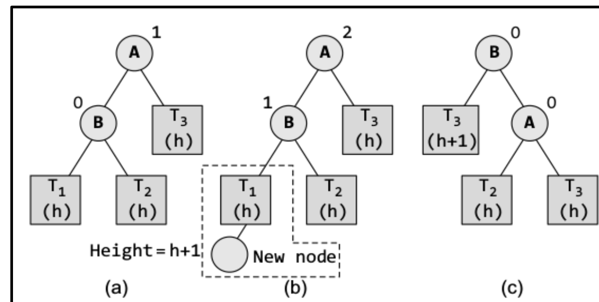


Figure 8.4 (a) Given AVL tree (b) Inserting a new node in the left subtree of left subtree of critical node (c) LL rotation for given AVL Tree

2. **RR rotation:** The new node is inserted in the right subtree of the right subtree of the critical node. RR rotation in an AVL tree is shown in figure 8.5. In this figure, node A is considered as a critical node, as it is the nearest ancestor whose balance factor is not -1, 0, or 1. After insert operation, the new node has now become the part of tree T_3. Now, during RR rotation, node B becomes the root; A and T_3 are its left and right child respectively; T_1 and T_2 are now left and right subtrees of A.

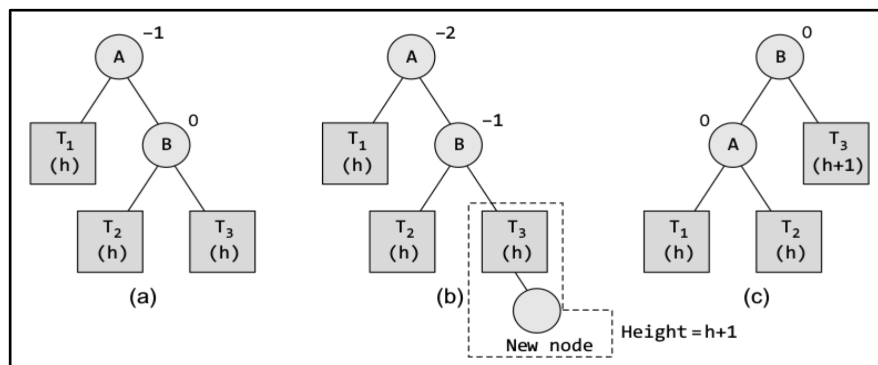


Figure 8.5 (a) Given AVL tree (b) Inserting a new node in the right subtree of right subtree of critical node (c) RR rotation for given AVL Tree

3. **LR rotation:** The new node is inserted in the right subtree of the left subtree of the critical node. LR rotation in an AVL tree is shown in figure 8.6. In this

figure, node A is considered as a critical node, as it is the nearest ancestor whose balance factor is not -1, 0, or 1. After insert operation, the new node has now become the part of tree T_2. Now, during LR rotation, node C becomes the root; B and A are its left and right child respectively; T_1 and T_2 are now left subtrees and right subtrees of B and T_3 and T_4 are now left subtrees and right subtrees of A.

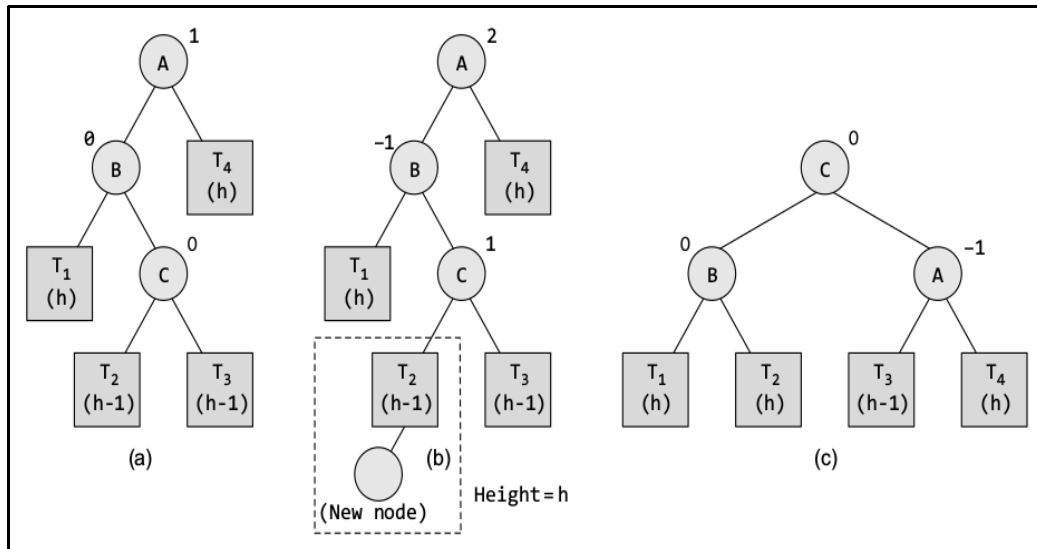


Figure 8.6 (a) Given AVL tree (b) Inserting a new node in the right subtree of left subtree of critical node (c) LR rotation for given AVL Tree

4. **RL rotation:** The new node is inserted in the left subtree of the right subtree of the critical node. RL rotation in an AVL tree is shown in figure 8.7. In this figure, node A is considered as a critical node, as it is the nearest ancestor whose balance factor is not -1, 0, or 1. After insert operation, the new node has now become the part of tree T_2. Now, during RL rotation, node C becomes the root; A and B are its left and right children, respectively; T_1 and T_2 are now left subtrees and right subtrees of A and T_3 and T_4 are now left subtrees and right subtrees of B.

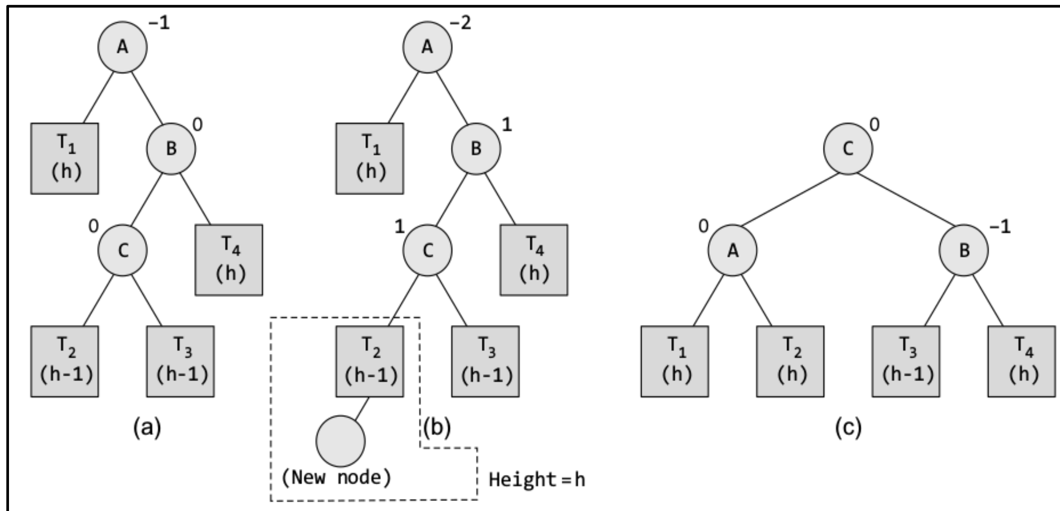


Figure 8.7 (a) Given AVL tree (b) Inserting a new node in the left subtree of right subtree of critical node (c) RL rotation for given AVL Tree

Deleting a node from an AVL Tree

The delete operation in an AVL tree is similar to that of a binary search tree. The only difference is in terms of maintaining the balance in the tree. After the delete operation, there may be a need to rebalance the AVL tree for which it is required to perform rotations. There are two types of rotations that can be performed on an AVL tree after deleting a given node. These rotations are *R rotation* and *L rotation*.

If node A becomes the critical node while deleting node X from the AVL tree, then the type of rotation depends on whether X is in the left subtree or in the right subtree of node A. If the node X is in the left subtree of A, then L rotation is applied. If X is in the right subtree, R rotation is performed.

Program 8.1: Write a C program that shows insertion operation in an AVL tree.

```
#include <stdio.h>
typedef enum { FALSE ,TRUE } bool;
struct node
{
	int val;
	int balance;
	struct node *left_child;
	struct node *right_child;
};
```

```
struct node* search(struct node *ptr, int data)
{
        if(ptr!=NULL)
                if(data < ptr -> val)
                        ptr = search(ptr -> left_child,data);
                 else if( data > ptr -> val)
                        ptr = search(ptr -> right_child, data);
        return(ptr);
}
struct node *insert (int data, struct node *ptr, int *ht_inc)
{
        struct node *aptr;
        struct node *bptr;
        if(ptr==NULL)
        {
                ptr = (struct node *) malloc(sizeof(struct node));
                ptr -> val = data;
                ptr -> left_child = NULL;
                ptr -> right_child = NULL;
                ptr -> balance = 0;
                *ht_inc = TRUE;
                return (ptr);
        }
        if(data < ptr -> val)
        {
                ptr -> left_child = insert(data, ptr -> left_child, ht_inc);
                if(*ht_inc==TRUE)
                {
                         switch(ptr -> balance)
                         {
                                case -1: /* Right heavy */
                                ptr -> balance = 0;
                                *ht_inc = FALSE;
                                break;
                                case 0: /* Balanced */
                                        ptr -> balance = 1;
```

```
                        break;
                case 1: /* Left heavy */
                        aptr = ptr -> left_child;
                        if(aptr -> balance == 1)
                        {
                                printf("Left to Left Rotation\n");
                                ptr -> left_child= aptr -> right_child;
                                aptr -> right_child = ptr;
                                ptr -> balance = 0;
                                aptr -> balance=0;
                                ptr = aptr;
                        }
                        else
                        {
                                printf("Left to right rotation\n");
                                bptr = aptr -> right_child;
                                aptr -> right_child = bptr -> left_child;
                                bptr -> left_child = aptr;
                                ptr -> left_child = bptr -> right_child;
                                bptr -> right_child = ptr;
                                if(bptr -> balance == 1 )
                                        ptr -> balance = -1;
                                else
                                        ptr -> balance = 0;
                                if(bptr -> balance == -1)
                                        aptr -> balance = 1;
                                else
                                        aptr -> balance = 0;
                                bptr -> balance=0;
                                ptr = bptr;
                        }
                        *ht_inc = FALSE;
            }
        }
    }
    if(data > ptr -> val)
```

```
{
          ptr -> right_child = insert(info, ptr -> right_child, ht_inc);
          if(*ht_inc==TRUE)
          {
                    switch(ptr -> balance)
                    {
                              case 1: /* Left heavy */
                                        ptr -> balance = 0;
                                        *ht_inc = FALSE;
                                        break;
                              case 0: /* Balanced */
                                        ptr -> balance = -1;
                                        break;
                              case -1: /* Right heavy */
                                        aptr = ptr -> right_child;
                                        if(aptr -> balance == -1)
                                        {
                                                  printf("Right to Right Rotation\n");
                                                  ptr -> right_child= aptr -> left_child;
                                                  aptr -> left_child = ptr;
                                                  ptr -> balance = 0;
                                                  aptr -> balance=0;
                                                  ptr = aptr;
                                        }
                                        else
                                        {
                                                  printf("Right to Left Rotation\n");
                                                  bptr = aptr -> left_child;
                                                  aptr -> left_child = bptr -> right_child;
                                                  bptr -> right_child = aptr;
                                                  ptr -> right_child = bptr -> left_child;
                                                  bptr -> left_child = pptr;
                                                  if(bptr -> balance == -1)
                                                            ptr -> balance = 1;
                                                  else
                                                            ptr -> balance = 0;
```

```
                                        if(bptr -> balance == 1)
                                                aptr -> balance = -1;
                                        else
                                                aptr -> balance = 0;
                                        bptr -> balance=0;
                                        ptr = bptr;
                                }/*End of else*/
                                *ht_inc = FALSE;
                        }
                }
        }
        return(ptr);
}
void display(struct node *ptr, int level)
{
        int i;
        if ( ptr!=NULL )
        {
                display(ptr -> right_child, level+1);
                printf("\n");
                for (i = 0; i < level; i++)
                        printf(" ");
                printf("%d", ptr -> val);
                display(ptr -> left_child, level+1);
        }
}
void inorder(struct node *ptr)
{       if(ptr!=NULL)
        {
                inorder(ptr -> left_child);
                printf("%d ",ptr -> val);
                inorder(ptr -> right_child);
        }
}
main()
{
```

```
bool ht_inc;
int data ;
int option;
struct node *root = (struct node *)malloc(sizeof(struct node));
root = NULL;
while(1)
{
        printf("1.Insert\n");
        printf("2.Display\n");
printf("3.Quit\n");
        printf("Enter your option : ");
        scanf("%d",&option);
        switch(choice)
        {
                case 1:
                        printf("Enter the value to be inserted : ");
                        scanf("%d", &data);
                                if( search(root,data) == NULL )
                                        root = insert(data, root, &ht_inc);
                                        else
                                        printf("Duplicate value ignored\n");
                                break;
                case 2:
                        if(root==NULL)
                        {
                                printf("Tree is empty\n");
                                continue;
                        }
                        printf("Tree is :\n");
                        display(root, 1);
                        printf("\n\n");
                        printf("Inorder Traversal is: ");
                        inorder(root);
                        printf("\n");
                        break;
                case 3:
```

```
                    exit(1);
                    default:
                            printf("Wrong option\n");
                    }
            }
}
```

8.4 Weight Balanced Trees

As we already know that the weight balanced trees are the type of self-balancing trees that are dependent on the number of leaves in the subtrees of a node. A binary search tree is said to be weight-balanced if the weight of the left and right subtree in each node differ by at most one. Weight balanced binary search trees were introduced by *Nievergelt and Reingold* in the 1970s, in the name "*trees of bounded balance*". Later, they were modified as weight balanced trees by *Kruth*. These trees are generally used to implement dynamic sets, maps, and sequences.

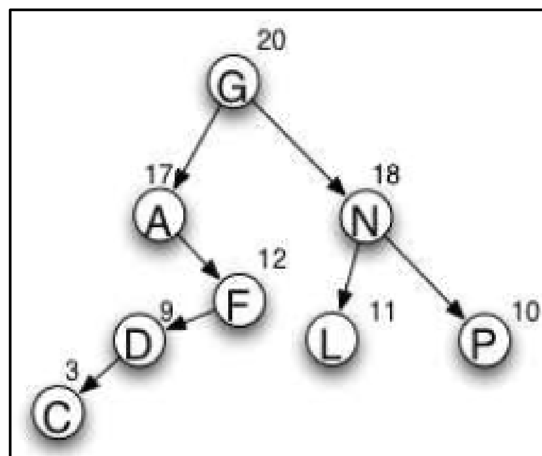


Figure 8.8 An example of Weight Balanced Tree

Source- https://www.wisdomjobs.com/e-university/data-structures-tutorial-290/weight-balanced-tree-7211.html

Like other self-balancing trees, weight-balanced trees also perform rotations to restore the balance between the nodes, when it becomes unbalanced by search, insert, and delete operations. The size of the subtree rooted at the node is stored by each node of the tree. The sizes of left and right subtrees are kept approximately the same by some factor. Let this factor be ***a.*** The types of rotations used to rebalance the binary trees are the same as those used to rebalance AVL trees.

According to the definition, the size of a leaf is zero and the size of an internal node is

calculated as adding one to the sum of sizes of its two children. The weight can be defined as adding one to the size of that internal node.

$$size[n] = size[n.left] + size[n.right] + 1$$

$$weight[n] = size[n] + 1$$

A node is said to be an α-weight balanced tree if it satisfies the following condition,

$$weight[n.left] \geq \alpha \cdot weight[n] \text{ and } weight[n.right] \geq \alpha \cdot weight[n]$$

Program 8.2: Write a C program to check whether the given tree is balanced or not.

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
        int data;
        struct node *left;
        struct node *right;
};
bool isBalanced(struct node *root);
int findheight(struct node *root)
{
   int lefth=0,righth=0;
   if(root==NULL)
   {
      return 0;
   }
   lefth=findheight(root->left);
   righth=findheight(root->right);
   if(lefth>righth)
   {
       return lefth+1;
   }
   else
   {
       return righth+1;
   }
}
```

```
bool isBalanced(struct node *root)
{
    int left_height,right_height;
    if(root==NULL)
    {
        return true;
    }
    left_height=findheight(root->left);
    right_height=findheight(root->right);
    if(abs(left_height-right_height)<=1 && isBalanced(root->left) && isBalanced(root->right))
    {
          return true;
    }
    return false;
}
int main()
{
struct node *root;
    root=(struct node*)malloc(sizeof(struct node));
    root->data=5;
    root->left=(struct node*)malloc(sizeof(struct node));
    root->left->data=8;
    root->left->left=(struct node*)malloc(sizeof(struct node));
    root->left->left->data=10;
    root->left->left->left=root->left->left->right=NULL;
    root->left->right=(struct node*)malloc(sizeof(struct node));
    root->left->right->data=15;
    root->left->right->left=root->left->right->right=NULL;
    root->right=(struct node*)malloc(sizeof(struct node));
    root->right->data=34;
    root->right->left=root->right->right=NULL;
          if(isBalanced(root))
    {
         printf("\n\n\nThe above given tree is a Balanced Tree\n\n\n");
    }
    else
```

```
    {
        printf("\n\n\nThe above given tree is not a Balanced Tree\n\n\n");
    }
    return 0;
}
```

The output of the program is:

The above-given tree is a Balanced Tree.

Or The above-given tree is not a Balanced Tree.

8.5 Summary

- Balanced trees are a versatile set of data structures in which every leaf is "at a certain distance" from the root than any other leaf.
- A binary tree is said to be balanced if the height of the tree is O(log n), where n is the number of nodes of the tree.
- Self- balancing trees maintain balance automatically by keeping the height as small as possible during the insertion and deletion operation on the binary tree.
- The balance factor of a node is the difference between the height of the right subtree and the height of the left subtree. In a height-balanced tree, every node has a balance factor of -1, 0, or 1.
- Rotations are used to retain the balance in a binary search tree. There are four types of rotations: LL rotation, RR rotation, LR rotation, and RL rotation.
- A binary search tree is said to be weight-balanced if the weight of the left and right subtree in each node differ by at most one.

8.6 Key Terms

- **Height- balanced Trees:** In height-balanced trees, the height of the siblings of a node is "approximately the same".
- **Weight- balanced Trees:** In weight balanced trees, the number of descendants of sibling nodes is "approximately the same".
- **Balance Factor:** It is the difference between the height of the right subtree and the height of the left subtree.
- **Left- heavy Tree:** If a node has a balance factor of 1, it means that the right subtree of the node is one level lower than the left subtree. Such a tree is called

a lefty-heavy tree.

- **Right- heavy Tree:** If a node has a balance factor -1, it means that the right subtree of the node is one level higher than the left subtree. Such a tree is called a right-heavy tree.

8.7 Check Your Progress

Short- Answer type

Q1) Time taken by an AVL tree to perform the search, insert, and delete operations in average as well as worst case is:

(a) O(n) (b) O(log n) (c) $O(n^2)$ (d) O(n log n)

Q2) In an AVL tree, searching operation takes ______ time.

Q3) When the new node is inserted in the right subtree of the right subtree of the critical node, then it is called RL rotation. (True/ False?)

Q4) When the right subtree of a node is one level lower than the left subtree, then the balance factor is

(a) 0 (b) 1 (c) –1 (d) 2

Q5) A new node inserted in a binary search tree, will be added as an internal node. (True/ False?)

Long- Answer type

Q1) The height of a binary search tree affects its performance. Explain.

Q2) State the advantages of AVL trees.

Q3) Differentiate between Height balanced and Weight balanced Trees.

Q4) Create an AVL tree using the following sequence of data: 16, 27, 9, 11, 36, 54, 81, 63, 72.

Q5) Explain the rotation process in Balanced trees in detail. Also, discuss the types of rotations.

References

- *Data Structures using C,* Reema Thareja, Oxford University Press, 2nd Edition
- *Classic Data Structures,* Debasis Samanta, PHI Learning Pvt. Ltd. 2nd Edition.
- https://epgp.inflibnet.ac.in/Home/ViewSubject?catid=7

Unit 9 B-Trees

Structure

9.0 Introduction
9.1 Unit Objectives
9.2 B- Trees
    9.2.1 Operations on a B- Tree
9.3 B+ Trees
    9.3.1 Operations on a B+ Tree
9.4 Red-Black Trees
    9.4.1 Inserting a Node in a Red-black Tree
    9.4.2 Deleting a Node from a Red-black Tree
9.5 Splay Trees
9.6 Summary
9.7 Key Terms
9.8 Check Your Progress

9.0 Introduction

We have discussed that in a binary search tree every node has one value and two pointers, that point to the left and right subtrees of the node, respectively. B-trees are generally used in file systems and databases. A tree data structure that sorts the data and then performs the insertion and deletion operations, is referred to as B-Tree. The internal nodes of a B-Tree may have a variable number of child nodes in some predefined range. The number of child nodes varies with the insertion or deletion of any data from the node. To maintain the predefined range, the internal nodes can be merged or splitted. As B-trees permit the maintenance of these child nodes, rebalancing is not frequently required in B-trees as other self-balancing trees require. But, this leads to the wastage of some space in the memory as nodes are not completely full.

This unit deals with the basics of B-Trees, its operations, and its applications. Apart from AVL trees, the fundamentals of other balanced trees like Red-Black Trees, Splay Trees, and B+ Trees are also discussed.

9.1 Unit Objectives

After going through this unit, the reader will be able to:

- Explain the basics of B-Trees.
- Understand the operations and applications of B-Trees.
- Learn about the distinct balanced trees like Red-black trees and Splay trees.
- Discuss the fundamentals of B+ Trees.

9.2 B- Trees

B-trees were developed by *Rudolf Bayer and Ed McCreight* in 1970. They are widely used for accessing the disk of computer systems. A B-tree having an order of *m* consists of *m-1* keys and *m* pointers to the subtrees. The purpose of using B-trees is to store a large number of keys in a single node to keep the height of the tree relatively small. The small height of the tree will take less processing time as compared to the tree with more height.

In B- trees the number of child nodes are in a predefined range and can vary with the insert or delete operations. There is a need to maintain this predefined range by merging or splitting these internal nodes. Unlike other self-balancing trees, B- trees do not require rebalancing frequently as they focus on maintaining the predefined range of internal nodes. B- trees consist of two limits i.e. upper bound and lower bound. These two bounds are fixed for the number of child nodes for a particular implementation.

As we already know that the height of all the leaf nodes has to be maintained to keep the tree balanced. Similarly, a B-tree is also balanced by keeping all the leaf nodes at the same depth/ height. The height of the B- Tree will increase with the addition of elements to the tree, but the overall height of the tree will not increase frequently.

A B- tree may have a variable number of keys and children, unlike a binary- tree. These keys are arranged in non-decreasing order. Each of these keys is associated with a child. This child behaves as the root of a subtree having all the nodes with keys less than or equal to the key but greater than the preceding key. An additional

rightmost child is also associated with the node. This rightmost child behaves as the root for a subtree that has all keys greater than any keys in the node.

A B-tree should possess the following properties:

- In a B-tree with order *m*, every node should have a maximum of *m* children.
- Every node except the root node and leaf nodes should have minimum *m/2* children.
- The root node should have at least two child nodes if it is not a leaf node.
- All leaf nodes should be at the same level.

Figure 9.1 shows a B-tree of order 4. It should be noted that the B-tree shown in the figure fulfills all the properties that are mentioned above.

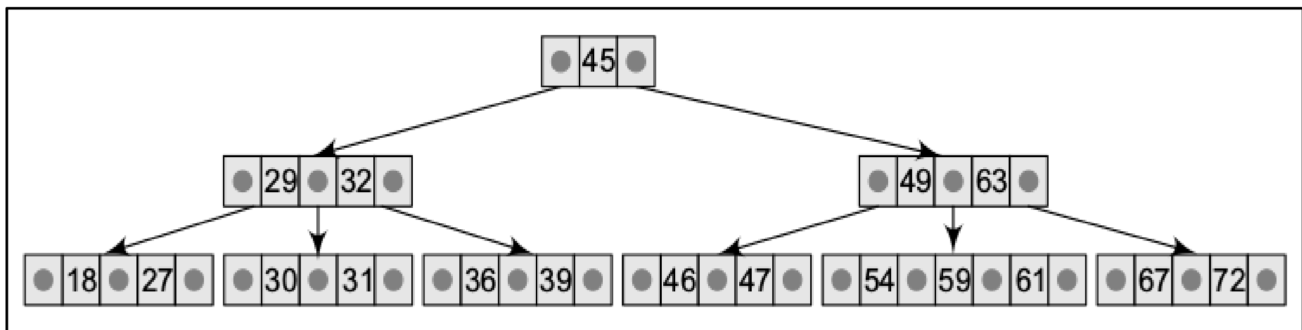


Figure 9.1 A B-tree of order 4.

Source- Data Structures using C, Reema Thareja, Oxford University Press, 2nd Edition, Chapter- 11, Page No.- 345

B-trees are balanced trees that can minimize the number of disk access for the computer system. Certain data is stored in secondary storage such as magnetic disks and disk access is expensive and time-consuming in such cases. So, B-trees help in minimizing the number of disk access attempts.

9.2.1 Operations on a B-Tree

Like other binary trees, searching, inserting, and deleting operations are also supported by B-trees. All these operations follow single-pass algorithms as they do not traverse back. As the basic motive of the B-tree is to minimize the disk access, these single pass approaches will support this motive. It is assumed that all the nodes are stored in secondary storage instead of primary storage. Disk-Read operation is used to read all the given nodes. Similarly, the write operation is denoted by Disk- Write.

Allocate- Node call is used to create new nodes and assign them storage.

- **Searching in a B-tree:** The search operation of a B-tree is similar to that of a binary tree. Unlike a binary tree, B-tree marks an n-way search instead of choosing between the left and right child of a node. The correct choice is made by performing a linear search for the values in the node. After obtaining the value greater than or equal to the required value, the search follows the child pointer to the immediate left of the value. On the other hand, if all the values are less than the required value, it follows the rightmost child pointer. The search operation is terminated as soon as the required node is found. The running time of the operation is decided by the height of the tree, i.e. O(log n). The search algorithm is given below:

```
B-Tree-Search(x, k)
i <- 1
while i <= n[x] and k > keyi[x]
        do i <- i + 1
if i <= n[x] and k = keyi[x]
        then return (x, i)
if leaf[x]
        then return NIL
else Disk-Read(ci[x])
return B-Tree-Search(ci[x], k)
```

- **Insertion in a B-Tree:** Before inserting any element in a B-tree, we must locate the appropriate node for the key, using certain algorithms such as B-tree search. Next, the key is inserted into the node. If the node is not full, no special action is required while if the node is full, then the node should be split and then the new key is loaded. The splitting operation moves one key to the parent node. Also, this parent node must not be full otherwise another split operation will be required. This process may repeat up to the root node.

```
B-Tree-Insert(T, k)
r <- root[T]
if n[r] = 2t - 1
        then s <- Allocate-Node()
                root[T] <- s
        leaf[s] <- FALSE
```

```
        n[s] <- 0
        c1 <- r
        B-Tree-Split-Child(s, 1, r)
        B-Tree-Insert-Nonfull(s, k)
        else B-Tree-Insert-Nonfull(r, k)
B-Tree-Insert-Nonfull(x, k)
i <- n[x]
if leaf[x]
        then while i >= 1 and k < keyi[x]
                do keyi+1[x] <- keyi[x]
        i <- i - 1
        keyi+1[x] <- k
n[x] <- n[x] + 1
Disk-Write(x)
else while i >= and k < keyi[x]
        do i <- i - 1
i <- i + 1
Disk-Read(ci[x])
if n[ci[x]] = 2t - 1
        then B-Tree-Split-Child(x, i, ci[x])
                if k > keyi[x]
                then i <- i + 1
B-Tree-Insert-Nonfull(ci[x], k)
```

- **Deletion in a B-Tree:** The delete operation for a B-Tree is carried out from the leaf node, like in the insert operation. A leaf node and an internal node can be deleted from a B-Tree.

  In the case of a leaf node, the following steps are involved:

  a) First, locate the leaf node that has to be deleted.
  b) If the leaf node has more than m/2 elements (more than a minimum number of key values), then delete the value.
  c) Else, if the leaf node does not have m/2 elements, then first fill the node either from the left or from the right sibling.
     - ➔ If there are more than m/2 elements in the ***left*** sibling, then its ***largest*** key is pushed into its parent's node. Also, the intermediate element of the parent and leaf node is taken down where the key is deleted.
     - ➔ Else, if there are more than m/2 elements in the ***right*** sibling, then

its ***smallest*** key is pushed into its parent's node. Also, the intermediate element of the parent and leaf node is taken down where the key is deleted.

d) Else, if there are m/2 elements in both left and right siblings, then a new leaf node is created by combining the two leaf nodes and the intermediate element of the parent node. It should be ensured that the number of elements should not exceed the maximum number of elements a node can have, i.e. m. If after pulling down the intermediate element of the parent node, it has less than m/2 elements, then the process is propagated upwards and the height of the B-Tree gets reduced.

In the case of an internal node, the successor or predecessor of the key to be deleted is promoted to occupy the position of the deleted key. The predecessor or successor keys are always in the leaf node, so the operation is processed according to the deletion in a leaf node.

Example 9.1: Consider the B-Tree of order 3 given below and perform the following operations: (a) insert 121, 87 and then (b) delete 36, 109.

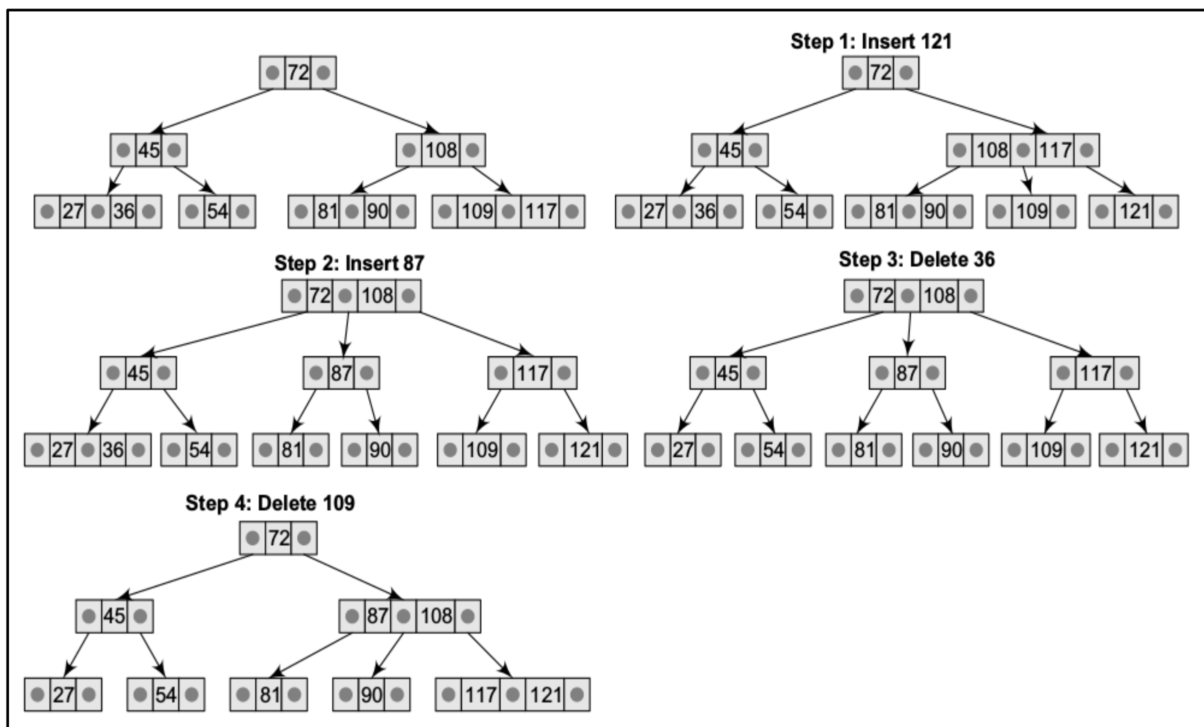


Figure 9.2 A B-tree of order 3 performing insert and delete operation.

Source- Data Structures using C, Reema Thareja, Oxford University Press, 2nd Edition, Chapter- 11, Page No.- 349

An underflow condition may occur during the delete operation in a B-Tree. A leaf node underflows if it contains (m/2 - 1) keys after deleting a key from it. On the other hand, an internal node (excluding the root node) underflows if there are (m/2 - 2) keys in the deletion process. While deleting any element from the B-Tree, either leaf node or internal node, underflow condition is checked every time.

9.3 B+ Trees

B+ trees are a variant of B-Trees that also store sorted data only in the leaf nodes. In a B-Tree both keys and records are stored in its internal nodes. In contrast, the B+ tree stores all the records at its leaf node, and internal nodes contain only the keys. An added advantage of using a B+ tree is that the leaf nodes are often linked to each other in a linked list. This makes the queries simpler and more efficient. B+ trees allow efficient insertion, retrieval, and deletion of records. Generally, B+ trees are used to store large data. The leaf nodes of the B+ tree are stored in the secondary storage while the internal nodes of the tree are stored in the main memory. The internal nodes of a B+ tree are called ***index nodes*** or ***i-nodes***.

B+ Trees are simple and used to implement many database systems. B+ trees are always balanced as all the data appear in the leaf nodes and are sorted. B+ trees also make searching for data-efficient. A B+ tree of order is shown in figure 9.3. Also, a comparison between B- Tree and B+ Tree is depicted in Table 9.1.

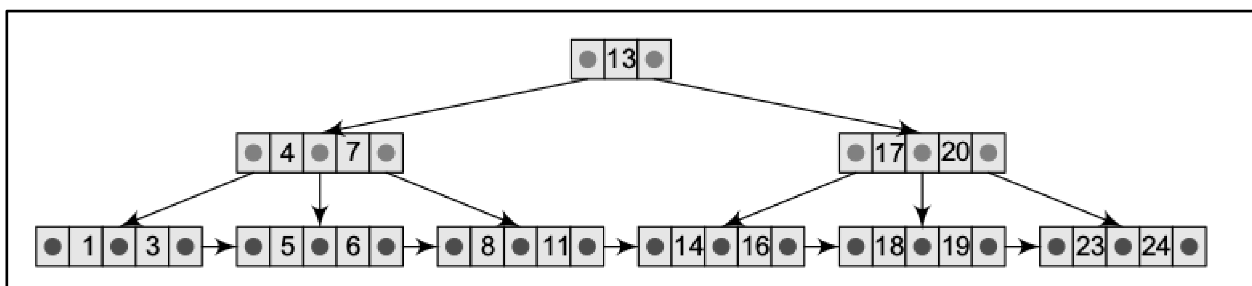


Figure 9.3 A B+tree of order 3

(Source- Data Structures using C, Reema Thareja, Oxford University Press, 2nd Edition, Chapter- 11, Page No.- 351**)**

Table 9.1 Comparison between B-trees and B+ trees

B Tree	B+ Tree
1. Search keys are not repeated	1. Stores redundant search key
2. Data is stored in internal or leaf nodes	2. Data is stored only in leaf nodes
3. Searching takes more time as data may be found in a leaf or non-leaf node	3. Searching data is very easy as the data can be found in leaf nodes only
4. Deletion of non-leaf nodes is very complicated	4. Deletion is very simple because data will be in the leaf node
5. Leaf nodes cannot be stored using linked lists	5. Leaf node data are ordered using sequential linked lists
6. The structure and operations are complicated	6. The structure and operations are simple

9.3.1 Operations on a B+ Tree

Like B- Tree, insert and delete operations can be performed on B+ trees too. Let's discuss these operations in detail.

- **Inserting a new element in a B+ Tree:** As we know that B+ trees are relevant to leaf nodes, a new element in a leaf node can be simply added if there is space for it. But, if there is no space for a new element, then the node splits into two nodes. A new index value is added to the parent node so that future queries can arbitrate between the two nodes. In fact, when a new element is added to a leaf node, it may be possible that all the nodes on the path from a leaf to the root may split. If a root node splits, a new leaf node gets created and the level of the tree increases by one. B+ trees follow the given algorithm to insert a new node:
    a) Insert the new node as the leaf node.
    b) If the leaf node overflows, split the node and copy the middle element to the next index node.
    c) If the index node overflows, split that node and move the middle element to the next index page.

Example 9.2: Consider the B+ tree of order 4 given and insert 33 in it.

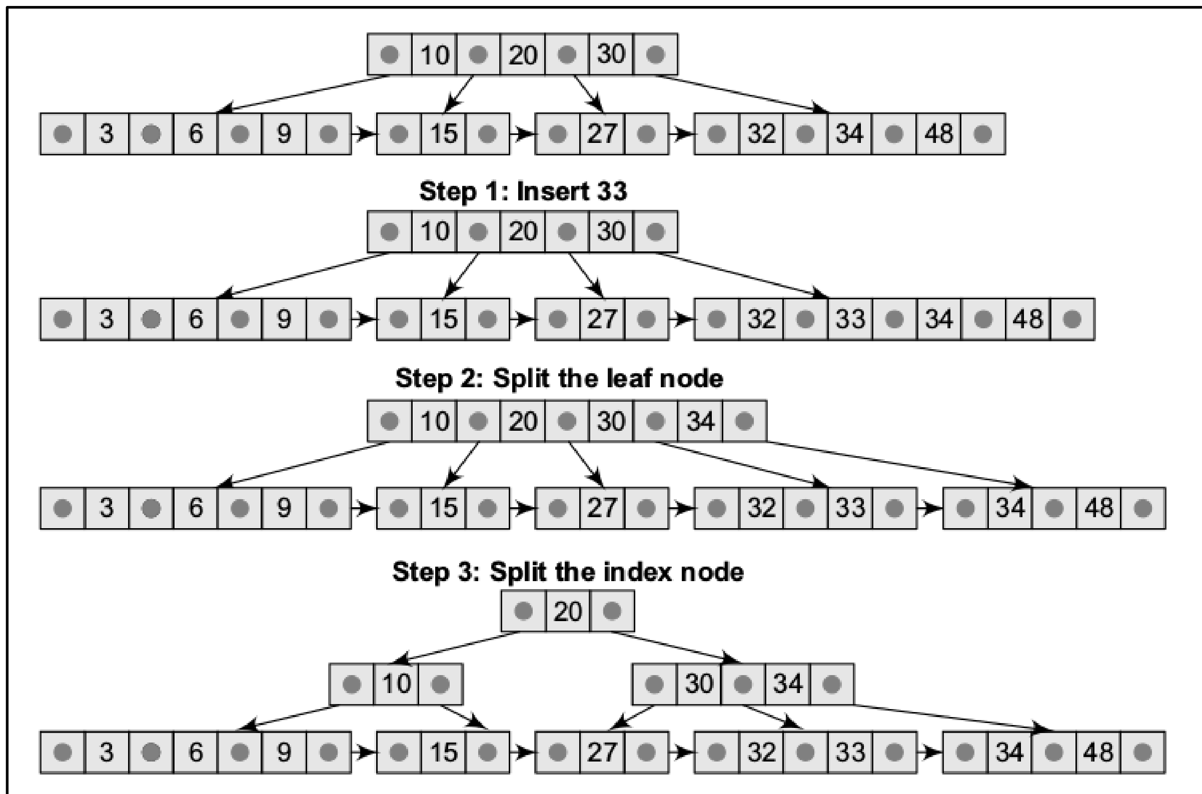


Figure 9.4 A B+tree of order 4 performing an insert operation

Source- Data Structures using C, Reema Thareja, Oxford University Press, 2nd Edition, Chapter- 11, Page No.- 352

- **Deleting an element from a B+ Tree:** As already discussed, in a B- Tree, deletion is done from a leaf node and an internal node. In B+ trees, deletion is always done from a leaf node. If the delete operation leaves that node empty, then the adjacent nodes are merged with that empty node. Due to this, an index value is deleted from the parent index node that in turn, may become empty. The process of merging and deleting may proceed from a leaf node to the root node. As a result, the level of the tree may decrease by one. B+ trees follow the given algorithm to delete a node from the tree:
    a) Delete the key and data from the leaves.

b) If the leaf node underflows, merge that node with the sibling and delete the key in between them.

c) If the index node underflows, merge that node with the sibling and move down the key in between them.

Example 9.3: Consider the B+ tree of order 4 given below and delete node 15 from it.

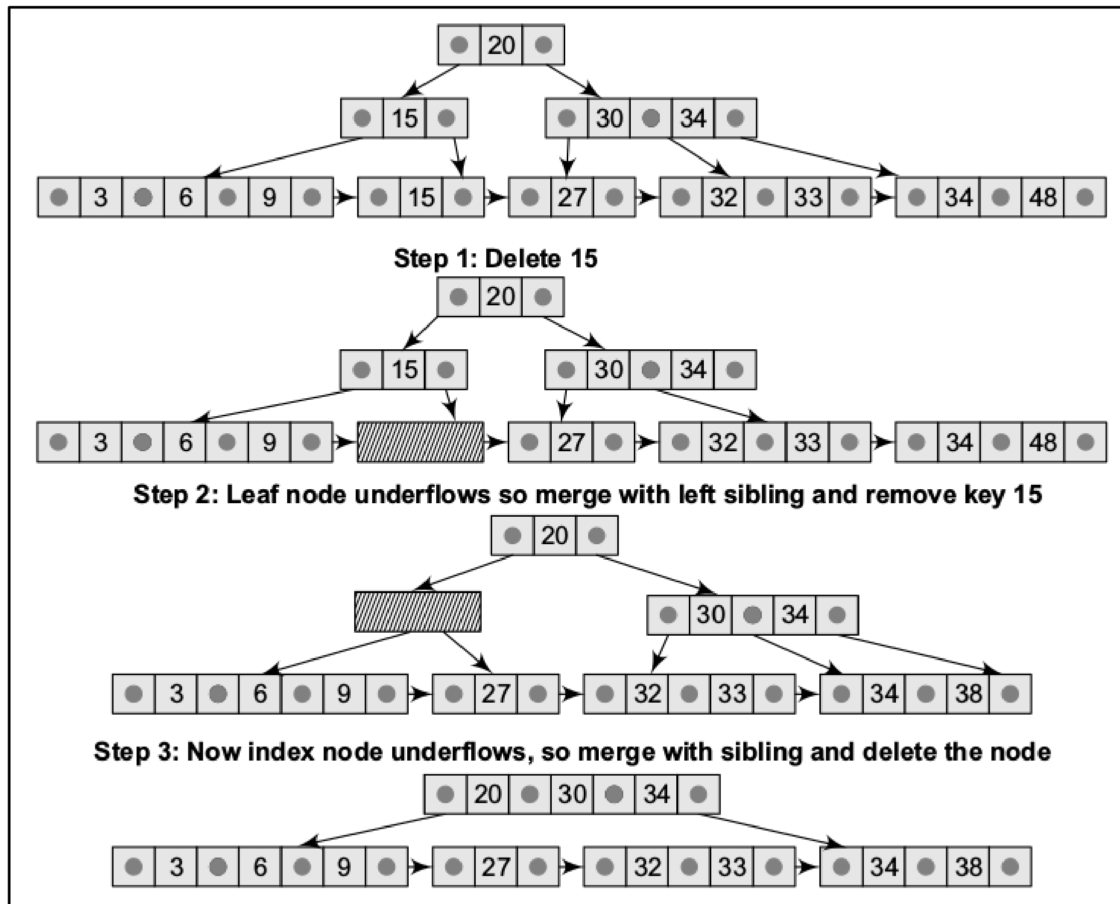


Figure 9.5 A B+tree of order 4 performing an insert operation

Source- Data Structures using C, Reema Thareja, Oxford University Press, 2nd Edition, Chapter- 11, Page No.- 353

9.4 Red-Black Trees

Another self-balancing tree in the series is Red-black Trees. They were first introduced by *Rudolf Bayer* in 1972 as 'symmetric binary B-Tree'. Although being complex trees, red-black trees are efficient in search, insert, and delete operations. Red-black trees have a good worst-case running time and all the operations get completed in O(log n) time. The red-black tree maintains the balance by intelligently inserting and deleting elements. It is interesting to note that no data is stored in the leaf nodes of a Red-

black tree. In a red-black tree, every node is labeled with either red or black color. Red-black trees follow the below rules:

a) Every node is labeled as either red or black in color.
b) The root node is always colored black.
c) No two adjacent nodes are red in color. That means a red node cannot have a red parent or a red child. Every red node should have both the children in black.
d) Every simple path from a given node to any of its leaf nodes has an equal number of black nodes.

Figure 9.6 shows an example of a red-black tree showing the color coding according to the above rules.

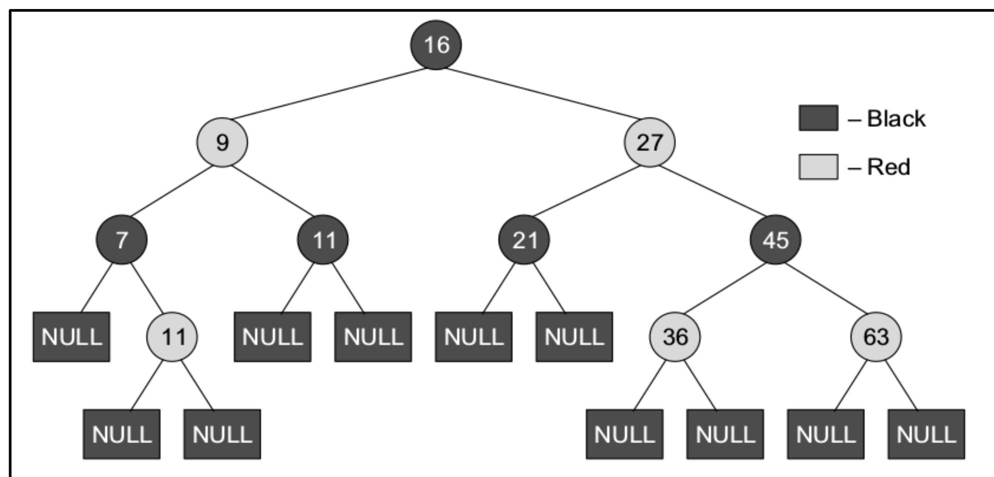


Figure 9.6 An example of a Red-black tree

Source: Data Structures using C, Reema Thareja, Oxford University Press, 2nd Edition, Chapter- 10, Page No.- 328

In a red-black tree, the longest path from the root node to any leaf node should not be more than twice as long as the shortest path from the root to any other leaf in that tree. An almost balanced tree is obtained from this. As the insert, search and delete operation require worst-case time proportional to the height of the tree. The red-black tree proves to be more efficient than any other ordinary binary search tree as it allows a certain upper bound on the height of the tree.

There is no modification in the read-only operations like traversing the nodes in a red-black tree, they are similar to that of the binary search trees. However, the insertion and deletion operation may affect the properties of a red-black tree. These operations may require a change in the color code of the tree. Let's discuss these operations for a

red-black tree.

9.4.1 Inserting a Node in a Red-black Tree

The insertion operation is the same as that in the binary search tree. Although, in a binary search tree, a new node is always added as a leaf, while in a red-black tree, there is no data in the leaf node. So, instead of a new leaf node, a red interior node having two black leaf nodes is added to the red-black tree. This follows the property of a red-black tree, having a red node as a root and black nodes as its children. This addition may violate the other properties of a red-black tree. So, to restore its properties, certain cases are checked and the related property is restored accordingly. Some important terms used in red-black tree insertion are discussed below.

- ***Grandparent node (G):*** It refers to the parent (P) of a node N, just like in a human family tree. Code (C) to find a node's grandparent can be given as follows:

```
struct node * grand_parent(struct node *n)
{
// No parent means no grandparent
if ((n != NULL) && (n -> parent != NULL))
        return n -> parent -> parent;
else
        return NULL;
}
```

- ***Uncle node (U):*** It refers to the sibling of a node N's parents (P), just like in a human family tree. The C code to find a node's uncle can be given as follows:

```
struct node *uncle(struct node *n)
{
struct node *g;
g = grand_parent(n);
//With no grandparent, there cannot be any uncle
if (g == NULL)
        return NULL;
if (n -> parent == g -> left)
        return g -> right;
else
```

```
        return g -> left;
    }
```

While inserting a new node in a red-black tree, the following points should be noted:

1. All leaf nodes are always black.
2. The property of a red-black tree that both children of every red node are black is threatened only by adding a red node, repainting a black node-red, or a rotation.
3. The property of a red-black tree that all paths from any given node to its leaf nodes has an equal number of black nodes is threatened only by adding a black node, repainting a red node black, or a rotation.

Case 1: The New Node N is Added as the Root of the Tree

In this case, N is repainted black, as the root of the tree is always black. Since N adds one black node to every path at once, Property 5 is not violated. The C code for case 1 can be given as follows:

```
void case1(struct node *n)
{
    if (n -> parent == NULL) // Root node
        n -> colour = BLACK;
    else
        case2(n);
}
```

Case 2: The New Node's Parent P is Black

In this case, both children of every red node are black. The new node N has two black leaf children, but because N is red, the paths through each of its children have the same number of black nodes. So, no property of a red-black tree is violated. The C code to check for case 2 can be given as follows:

```
void case2(struct node *n)
{
    if (n -> parent -> colour == BLACK)
        return; /* Red black tree property is not violated*/
    else
```

```
        case3(n);
}
```

Before proceeding to case 3, it is assumed that N has a grandparent node G, because its parent P is red, and if it were the root, it would be black. Thus, N also has an uncle node U (irrespective of whether U is a leaf node or an internal node).

Case 3: If Both the Parent (P) and the Uncle (U) are Red

In this case, the property that all paths from any given node to its leaf nodes have an equal number of black nodes is violated. Insertion in the third case is illustrated in figure 9.7.

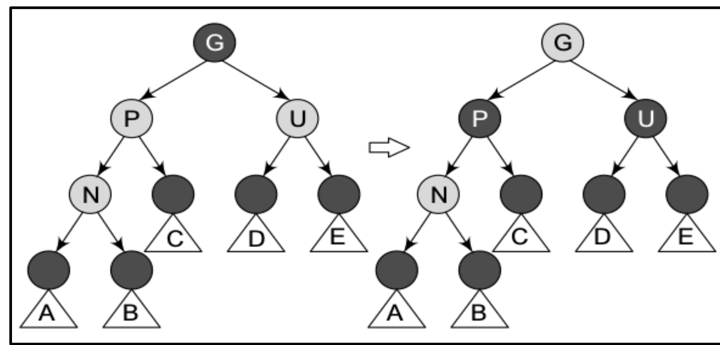


Figure 9.7 Insertion in a red-black tree (Case 3)

Source- Data Structures using C, Reema Thareja, Oxford University Press, 2nd Edition, Chapter- 10, Page No.- 331

In order to restore the above property, both nodes (P and U) are repainted black and the grandparent G is repainted red. Now, the new red node N has a black parent. Since any path through the parent or uncle must pass through the grandparent, the number of black nodes on these paths has not changed.

However, the grandparent G may now violate the property which states that the root node is always black or another property which states that both children of every red node are black. The latter property will be violated when G has a red parent. In order to fix this problem, this entire procedure is recursively performed on G from Case 1.

The C code to deal with Case 3 insertion is as follows:

```
void case3(struct node *n)
{
        struct node *u, *g;
        u = uncle (n);
```

```
        g = grand_parent(n);
        if ((u != NULL) && (u -> colour == RED)) {
                n -> parent -> colour = BLACK;
                u -> colour = BLACK;
                g -> colour = RED;
case1(g);
        }
else {
                insert_case4(n);
        }
}
```

Please note that in the remaining cases, it is assumed that the parent node P is the left child of its parent. If it is the right child, then interchange left and right in cases 4 and 5.

Case 4: The Parent P is Red but the Uncle U is Black and N is the Right Child of P and P is the Left Child of G

In order to fix this problem, a left rotation is done to switch the roles of the new node N and its parent P. After the rotation, note that in the C code, we have re-labeled N and P and then, case 5 is called to deal with the new node's parent. This is done because the property which says both children of every red node should be black is still violated. Figure 9.8 illustrates Case 4 insertion.

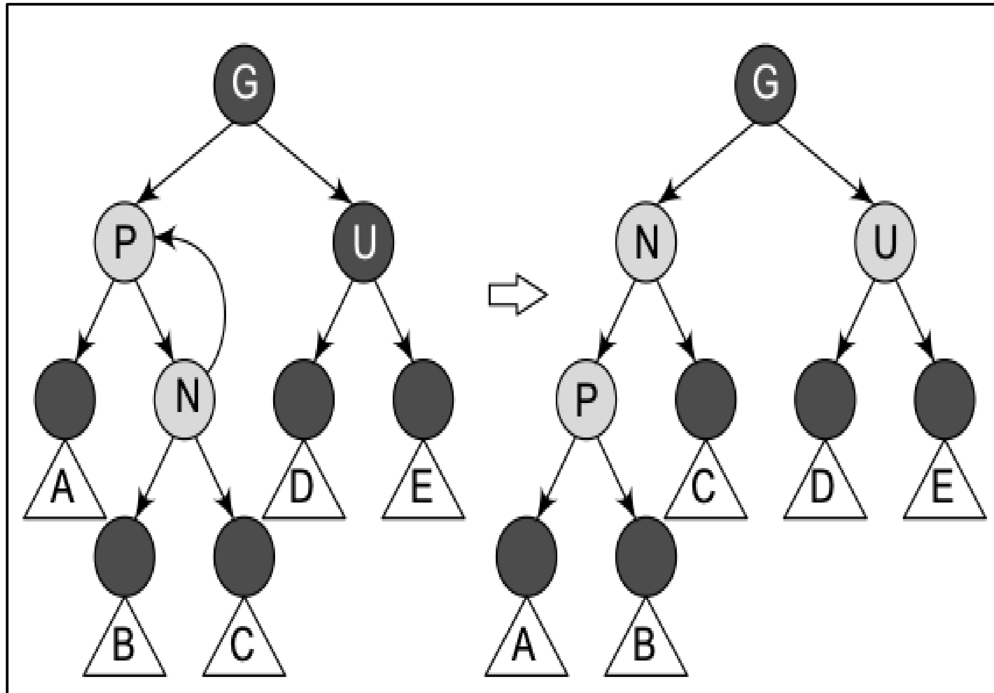


Figure 9.8 Insertion in a red-black tree (Case 4)

Source- Data Structures using C, Reema Thareja, Oxford University Press, 2nd Edition, Chapter- 10, Page No.- 332

It should be noted that in this case, N is the left child of P and P is the right child of G, we have to perform a right rotation. In the C code that handles Case 4, we check for P and N and then, perform either a left or a right rotation.

```
void case4(struct node *n)
{
	struct node *g = grand_
	parent(n);
	if ((n == n -> parent -> right)&& (n -> parent == g -> left))
	{
		rotate_left(n -> parent);
		n = n -> left;
	}
	else if ((n == n -> parent -> left) && (n -> parent == g -> right))
	{
		rotate_right(n -> parent);
		n = n -> right;
	}
	case5(n);
```

```
}
```

Case 5: The Parent P is Red but the Uncle U is Black and the New Node N is the Left Child of P, and P is the Left Child of its Parent G.

In order to fix this problem, a right rotation on G (the grandparent of N) is performed. After this rotation, the former parent P is now the parent of both the new node N and the former grandparent G.

We know that the color of G is black (because otherwise, its former child P could not have been red), so now switch the colors of P and G so that the resulting tree satisfies the property stating that both children of a red node are black. Case 5 insertion is illustrated in figure 9.9.

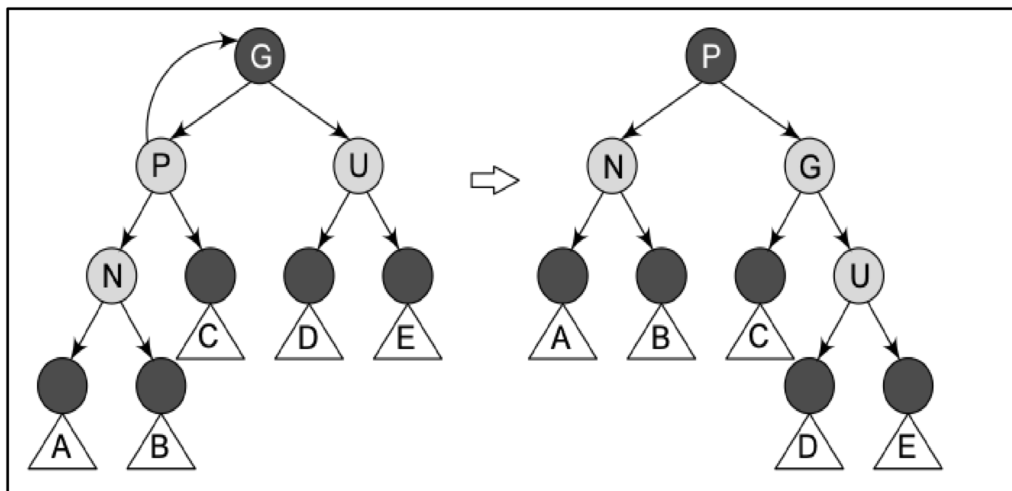


Figure 9.9 Insertion in a red-black tree (Case 5)

Source: Data Structures using C, Reema Thareja, Oxford University Press, 2nd Edition

It should be noted that in case, N is the right child of P and P is the right child of G, we perform a left rotation. In the C code that handles Case 5, we check for P and N and then, perform either a left or a right rotation.

```
void case5(struct node *n)
{
        struct node *g;
        g = grandparent(n);
        if ((n == n -> parent -> left) && (n -> parent == g -> left))
                rotate_right(g);
        else if ((n == n -> parent -> right) && (n -> parent == g -> right))
```

```
        rotate_left(g);
    n -> parent -> colour = BLACK;
    g -> colour = RED;
}
```

9.4.2 Deleting a Node from a Red-black Tree

Deleting a node from a red-black tree is the same as that in the binary search tree. In a binary search tree, when we delete a node with two non-leaf children, we find either the maximum element in its left subtree of the node or the minimum element in its right subtree, and move its value into the node being deleted. After that, we delete the node from which we had copied the value. It should be noted that this node must have less than two non-leaf children. Therefore, merely copying a value does not violate any red-black properties, but it just reduces the problem
of deleting to the problem of deleting a node with at most one non-leaf child. It will be assumed that we are deleting a node with at most one non-leaf child, which we will call its child. In case this node has both leaf children, then let one of them be its child.
While deleting a node, if its color is red, then we can simply replace it with its child, which must be black. All paths through the deleted node will simply pass through one less red node, and both the deleted node's parent and the child must be black, so none of the properties will be violated.
However, a complex situation arises when both the node to be deleted as well as its child is black. In this case, we begin by replacing the node to be deleted with its child. In the C code, we label the child node as (in its new position) N, and its sibling (its new parent's other child) as S.

The C code to find the sibling of a node can be given as follows:

```
struct node *sibling(struct node *n)
{
    if (n == n -> parent -> left)
        return n -> parent -> right;
    else
        return n -> parent -> left;
}
```

We can start the deletion process by using the following code, where the function replace_node substitutes the child into N's place in the tree. For convenience, we assume that null leaves are represented by actual node objects, rather than NULL.

```
void delete_child(struct node *n)
{
        /* If N has at most one non–null child */
        struct node *child;
        if (is_leaf(n -> right))
                child = n -> left;
        else
                child = n -> right;
        replace_node(n, child);
        if (n -> colour == BLACK) {
                if (child -> colour == RED)
                        child -> colour = BLACK;
                else
                        del_case1(child);
        }
        free(n);
}
```

When both N and its parent P are black, then deleting P will cause paths that precede through
N to have one fewer black nodes than the other paths. This will violate the property stating that every simple path from a given node to any of its leaf nodes has an equal number of black nodes. Therefore, the tree needs to be rebalanced. There are several cases to consider, which are discussed below.

Case 1: N is the New Root

In this case, we have removed one black node from every path, and the new root is black, so none of the properties are violated.

```
void del_case1(struct node *n)
{
        if (n -> parent != NULL)
```

```
        del_case2(n);
}
```

In the upcoming cases 2, 5, and 6, we assume N is the left child of its parent P. If it is the right child, left and right should be interchanged throughout these three cases.

Case 2: Sibling S is Red

In this case, interchange the colors of P and S, and then rotate left at P. In the resultant tree, S will become N's grandparent. Figure 9.10 illustrates Case 2 deletion.

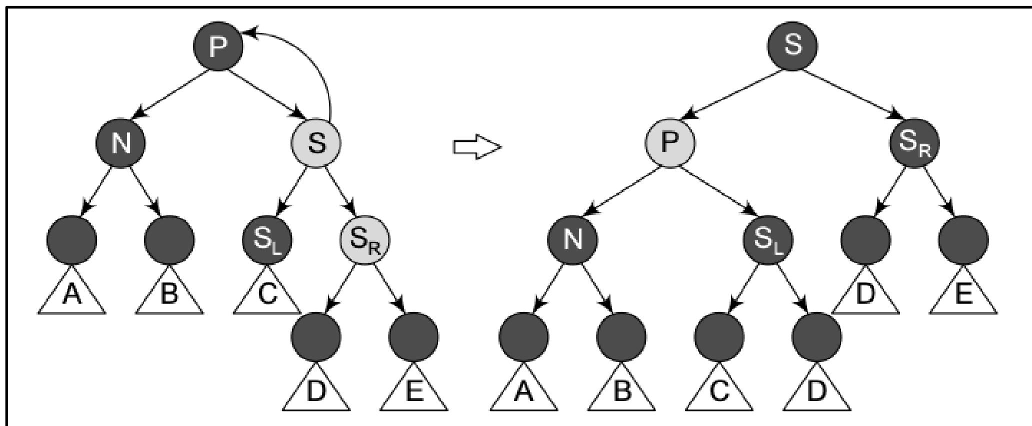


Figure 9.10 Deletion in a red-black tree (Case 2)

Source- Data Structures using C, Reema Thareja, Oxford University Press, 2nd Edition, Chapter- 10, Page No.- 334

The C code that handles case 2 deletions can be given as follows:

```
void del_case2(struct node *n)
{
    struct node *s;
    s = sibling(n);
    if (s -> colour == RED)
    {
        if (n == n -> parent -> left)
            rotate_left(n -> parent);
        else
            rotate_right(n -> parent);
        n -> parent -> colour = RED;
        s -> colour = BLACK;
    }
```

```
        del_case3(n);
}
```

Case 3: P, S, and S's Children are Black

In this case, simply repaint S with red. In the resultant tree, all the paths passing through S will have one less black node. Therefore, all the paths that pass through P now have one fewer black node than the paths that do not pass through P, so one of the properties is still violated. To fix this problem, we perform the rebalancing procedure on P, starting at Case 1. Case 3 is illustrated in figure 9.11.

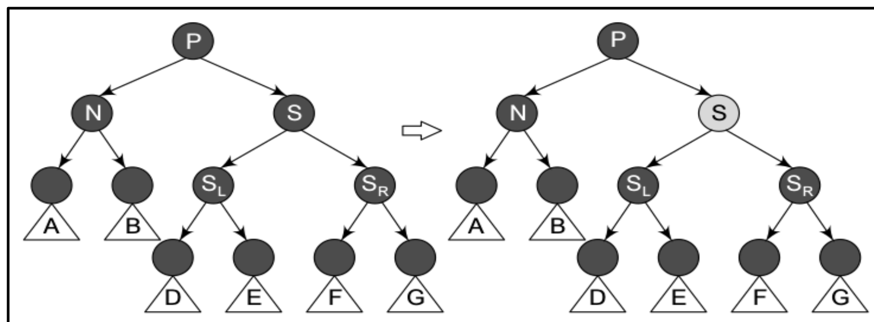


Figure 9.11 Deletion in a red-black tree (Case 3)

(**Source-** Data Structures using C, Reema Thareja, Oxford University Press, 2nd Edition, Chapter- 10, Page No.- 335)

The C code for Case 3 can be given as follows:

```
void del_case3(struct node *n)
{
        struct node *s;
        s = sibling(n);
        if ((n -> parent -> colour == BLACK) && (s -> colour == BLACK) && (s -> left -> colour ==
BLACK) && (s -> right -> colour == BLACK))
        {
                s -> colour = RED;
                del_case1(n -> parent);
        } else
                del_case4(n);
}
```

Case 4: S and S's children are Black, but P is Red

In this case, we interchange the colors of S and P. Although this will not affect the

number of black nodes on the paths going through S, it will add one black node to the paths going through N, making up for the deleted black node on those paths. The C code to handle Case 4 is as follows:

```
void del_case4(struct node *n)
{
        struct node *s;
        s = sibling(n);

if ((n -> parent -> colour == RED) && (s -> colour == BLACK) && (s -> left -> colour == BLACK) &&
(s -> right -> colour == BLACK))
        {
                s -> colour = RED;
                n -> parent -> colour = BLACK;
        } else
                del_case5(n);
}
```

Case 5: N is the Left Child of P and S is Black, S's Left Child is Red, S's Right Child is Black.

In this case, perform a right rotation at S. After the rotation, S's left child becomes S's parent and N's new sibling. Also, interchange the colors of S and its new parent.

It should be noted that now all paths still have an equal number of black nodes, but N has a black sibling whose right child is red, so we fall into Case 6. Deletion in Case 5 is illustrated in figure 9.12.

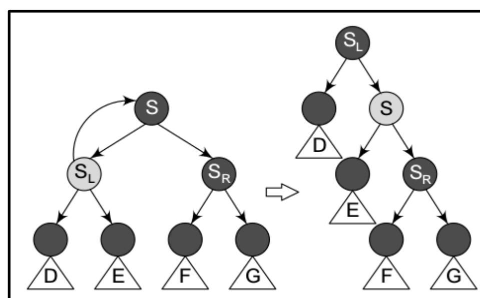


Figure 9.12 Deletion in a red-black tree (Case 5)

Source- Data Structures using C, Reema Thareja, Oxford University Press, 2nd Edition, Chapter- 10, Page No.- 336

The C code to handle case 5 is given as follows:

```
void del_case5(struct node *n)
```

```
{
        struct node *s;
        s = sibling(n);
        if (s -> colour == BLACK)
        {
        /* the following code forces the red to be on the left of the left of the parent, or right of the
right, to be correctly operated in case 6. */
                if ((n == n -> parent -> left) && (s -> right -> colour == BLACK) && (s -> left-> colour
== RED))
                                rotate_right(s);
                else if ((n == n -> parent -> right) && (s -> left -> colour == BLACK) && (s -> right->
colour == RED))
                        rotate_left(s);
                s -> colour = RED;
                s -> right -> colour = BLACK;
        }
        del_case6(n);
}
```

Case 6: S is Black, S's Right Child is Red, and N is the Left Child of its Parent P

In this case, a left rotation is done at P to make S the parent of P and S's right child. After the rotation, the colors of P and S are interchanged and S's right child is colored black. Once these steps are followed, you will observe that property 4 and property 5 remain valid. Case 6 is illustrated in figure 9.13.

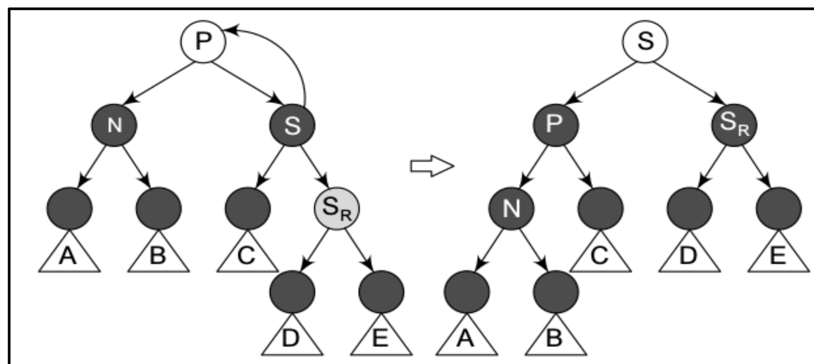


Figure 9.13 Deletion in a red-black tree (Case 6)

Source: Data Structures using C, Reema Thareja, Oxford University Press, 2nd Edition, Chapter- 10, Page No.- 337

The C code to fix Case 6 can be given as follows:

```
Void del_case6(struct node *n)
{
        struct node *s;
        s = sibling(n);
        s -> colour = n -> parent -> colour;
        n -> parent -> colour = BLACK;
        if (n == n -> parent -> left)
        {
                s -> right -> colour = BLACK;
                rotate_left(n -> parent);
        }
        else {
                s -> left -> colour = BLACK;
                rotate_right(n -> parent);
        }
}
```

The red-black trees are efficient variants of binary search trees, as they offer a worst-case time guarantee for insertion, deletion, and search operations. These trees are valuable in time-sensitive applications such as real-time applications. Red-black trees are also preferred to be used as a building block in other data structures that provide a worst-case guarantee.

9.5 Splay Trees

Splay trees are self-balancing binary search trees that were invented by *Daniel Sleator and Robert Tarjan*. Splay trees have an additional property of re-accessing the recently accessed elements fastly. Splay trees are efficient binary search trees as they can perform basic operations like search, insertion, and deletion in O(log n) time. They are advantageous for various non-uniform or even unknown series of operations.

A splay tree is a binary tree with no additional fields. While accessing any node in a splay tree, it is rotated or splayed to the root, which ultimately changes the structure of the tree. As the most frequently accessed nodes are always closer to the root node, we can locate these nodes faster. Thus, it can be interpreted that if any node is accessed once, then it can likely be accessed again.

Unlike other binary search trees, the basic operations in a splay tree are combined with a "*splaying*" operation. This additional operation for a particular node rearranges

that node at the root. In the splaying process, a standard binary search operation is performed for the desired node and then rotations are used in specific order to bring that node on the top.

The advantages of using a splay tree are:

- A splay tree gives good performance for search, insertion, and deletion operations. This advantage centers on the fact that the splay tree is a self-balancing and self-optimizing data structure in which the frequently accessed nodes are moved closer to the root so that they can be accessed quickly. This advantage is particularly useful for implementing caches and garbage collection algorithms.
- Splay trees are considerably simpler to implement than the other self-balancing binary search trees, such as red-black trees or AVL trees, while their average-case performance is just as efficient.
- Splay trees minimize memory requirements as they do not store any book-keeping data.
- Unlike other types of self-balancing trees, splay trees provide good performance (amortized O(log n)) with nodes containing identical keys.

However, the demerits of splay trees include:

- While sequentially accessing all the nodes of a tree in sorted order, the resultant tree becomes completely unbalanced. This takes n accesses of the tree in which each access takes O(log n) time. For example, re-accessing the first node triggers an operation that in turn takes O(n) operations to rebalance the tree before returning the first node. Although this creates a significant delay for the final operation, the amortized performance over the entire sequence is still O(log n).
- For uniform access, the performance of a splay tree will be considerably worse than a somewhat balanced simple binary search tree. For uniform access, unlike splay trees, these other data structures provide worst-case time guarantees and can be more efficient to use.

Let’s discuss the basic operations of splay trees in detail.

A. Splaying

When we access a node N, splaying is performed on N to move it to the root. To perform a splay operation, certain splay steps are performed where each step moves N closer to the root. Splaying a particular node of interest after every access ensures that the recently accessed nodes are kept closer to the root and the tree remains roughly balanced so that the desired amortized time bounds can be achieved. Each splay step depends on three factors:

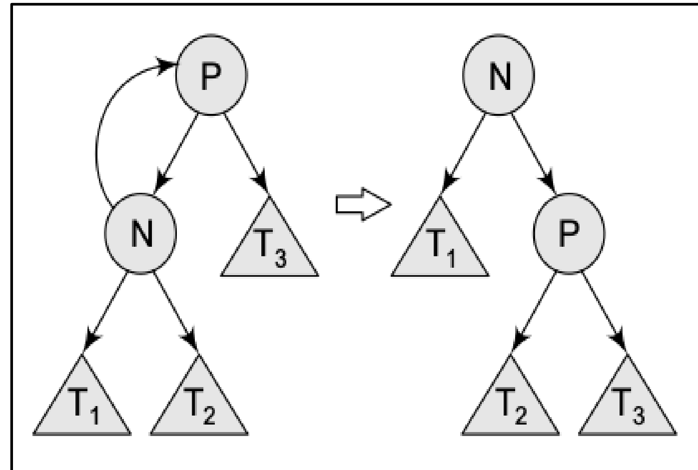
- Whether N is the left or right child of its parent P,
- Whether P is the root or not, and if not,
- Whether P is the left or right child of its parent, G (N's grandparent).

Depending on these three factors, we have one splay step based on each

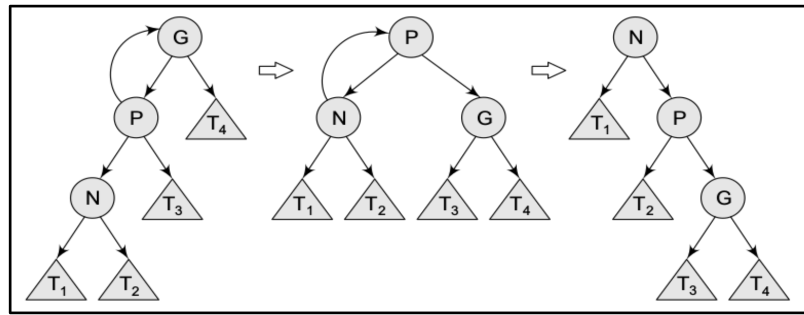
Zig step: The *zig* operation is performed when P (the parent of N) is the root of the splay-tree. In the zig step, the tree is rotated on the edge between N and P. Zig step is usually performed as the last step in a splay operation and only when N has an odd depth at the beginning of the operation. The zig step is shown in figure 9.14 (a).

Zig-zig step: The zig–zig operation is performed when P is not the root. In addition to this, N and P are either both right or left children of their parents. Figure 9.14 (b) shows the case where N and P are the left children. During the zig–zig step, first the tree is rotated on the edge joining P and its parent G, and then again rotated on the edge joining N and P.

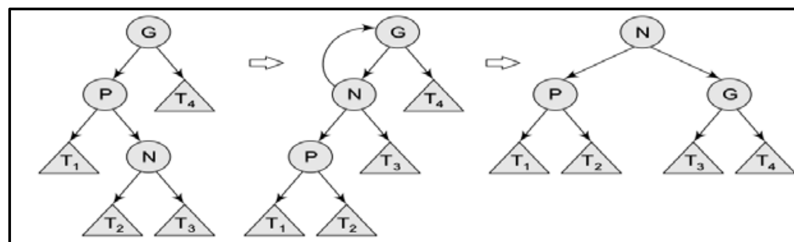
Zig-zag step: The zig-zag operation is performed when P is not the root. In addition to this, N is the right child of P and P is the left child of G or vice versa. In the zig-zag step, the tree is first rotated on the edge between N and P and then rotated on the edge between N and G. The zig-zag step is shown in figure 9.14 (c).



(a) The zig step



(b) The zig-zig step



(c) The zig-zag step

Figure 9.14 Splaying operation in splay trees

Source- Data Structures using C, Reema Thareja, Oxford University Press, 2nd Edition, Chapter- 10, Page No.- 338

B. Inserting a Node in a Splay Tree

Although the process of inserting a new node N into a splay tree begins in the same way as we insert a node in a binary search tree, after the insertion, N is made the new root of the splay-tree. The steps performed to insert a new node N in a splay tree can be given as follows:

1. Search N in the splay-tree. If the search is successful, splay at the node N.

2. If the search is unsuccessful, add the new node N in such a way that it replaces the NULL pointer reached during the search by a pointer to a new node N. Splay the tree at N.

C. Searching for a Node in a Splay Tree

If a particular node N is present in the splay tree, then a pointer to N is returned; otherwise, a pointer to the null node is returned. The steps performed to search a node N in a splay tree include:

1. Search down the root of the splay tree looking for N.
2. If the search is successful, and we reach N, then splay the tree at N and return a pointer to N.
3. If the search is unsuccessful, i.e., the splay tree does not contain N, then we reach a null node. Splay the tree at the last non-null node reached during the search and return a pointer to null.

D. Deleting a Node from a Splay Tree

To delete a node N from a splay tree, we perform the following steps:

1. Search for N that has to be deleted. If the search is unsuccessful, splay the tree at the last non-null node encountered during the search.
2. If the search is successful and N is not the root node, then let P be the parent of N. Replace N by an appropriate descendent of P (as we do in binary search tree). Finally, splay the tree at P.

9.6 Summary

- A B-tree having an order of *m* consists of *m-1* keys and *m* pointers to the subtrees. The purpose of using B-trees is to store a large number of keys in a single node to keep the height of the tree relatively small.
- B+ trees are a variant of B-Trees that also store sorted data only in the leaf nodes.
- In a B-Tree both keys and records are stored in its internal nodes. In contrast, the B+ tree stores all the records at its leaf node, and internal nodes contain only the keys.
- A red-black tree is a self-balancing binary search tree which is also known as a

'symmetric binary B-tree'. Although a red-black tree is complex, it has a good worst-case running time for its operations and is efficient to use, as searching, insertion, and deletion can all be done in O(log n) time.

- A splay tree is a self-balancing binary search tree with an additional property that recently accessed elements can be re-accessed fast.

9.7 Key Terms

- **Amortized Analysis:** The time complexity of maintaining a splay tree is analyzed using an Amortized Analysis.
- **B-trees:** B-trees are balanced trees that are optimized for situations when part or the entire tree must be maintained in secondary storage such as a magnetic disk.
- **Minimization factor:** A b-tree has a minimum number of allowable children for each node known as the Minimization factor.
- **Splaying:** Splaying a particular node of interest after every access ensures that the recently accessed nodes are kept closer to the root and the tree remains roughly balanced.

9.8 Check Your Progress

Short- Answer type

Q1) Every node in a B tree has at most ______ children.

(a) M (b) M–1 (c)2 (d)M+1

Q2) In _______ data is stored in internal or leaf nodes.

Q3) A B+ tree stores data only in the i-nodes. True/ False?

Q4) Splay Trees were invented by

(a) Sleator (b) Tarjan (c) Newton (d) Both (a) and (b)

Q5) The lower and upper bounds on the number of child nodes are typically fixed for a particular implementation.

Long- Answer type

Q1) Explain splay operation in splay trees.

Q2) Differentiate between B-Trees and B+ Trees.

Q3) Consider the B-tree given below:

(a) Insert 1, 5, 7, 11, 13, 15, 17, and 19 in the tree.

(b) Delete 30, 59, and 67 from the tree.

Q4) List the merits and demerits of a splay-tree.

Q5) Discuss the properties of a red-black tree. Explain the insertion cases.

References

- *Data Structures using C,* Reema Thareja, Oxford University Press, 2nd Edition
- *Classic Data Structures,* Debasis Samanta, PHI Learning Pvt. Ltd. 2nd Edition.
- *Data Structures and Efficient Algorithms*, Burkhard Monien, Thomas Ottmann, Springer.

Unit 10 Advanced Trees

Structure

10.0 Introduction
10.1 Unit Objectives
10.2 Interval Trees
10.3 Segment Trees
10.4 KD-Trees
10.5 Quad Trees
10.6 Summary
10.7 Key Terms
10.8 Check Your Progress

10.0 Introduction

We are now already aware of the fundamentals of Balanced trees. The importance of balanced search trees does not come primarily from the importance of dictionary structures; they are just the most basic applications. Balanced search trees provide a frame on which many other useful structures can be built. These other structures can then take advantage of the logarithmic depth and the mechanisms that preserve it, without going into the details of studying the underlying search-tree balancing methods. In this chapter, we describe several structures that are built on top of a balanced search tree and that implement different queries or even an entirely different abstract structure.

10.1 Unit Objectives

After going through this unit, the reader will be able to:

- Explain the different abstract tree structures.
- Describe the interval trees and segment trees.
- Define KD- trees and Quadtrees.

10.2 Interval Trees

Interval trees were invented by *Edelsbrunner* and *McCreight*. The interval tree structure stores a set of intervals and returns for any query key all the intervals that contain this query value. The structure is in a way dual to the one-dimensional range

queries such as they keep track of a set of values and return for a given query interval all key values in that interval, whereas we now have a set of intervals as data and a key-value as a query. In both cases the answer can be potentially large, so we have to aim for an output-sensitive complexity bound.
The idea of the interval tree structure is simple. Suppose the underlying set of intervals is the set $\{[a_1, b_1], [a_2, b_2], . . . , [a_n, b_n]\}$. Let T be any balanced search tree for the set of interval endpoints $\{a_1, a_2, . . . , a_n, b_1, . . . , b_n\}$. With each interior node of this search tree, we associate the interval of possible key values that can reach this node.

Each interval $[a_i, b_i]$ of our set is now stored in a node that satisfies the following conditions:

1. The key of the node is contained in $[a_i, b_i]$, and
2. The interval $[a_i, b_i]$ is contained in the interval associated with the node.

Such a node is easy to find: given $[a_i, b_i]$ and T, we start with the root as the current node. The interval associated with the root is $]-\infty, \infty[$, so property 2 is initially satisfied by the current node. If the key in the current node is contained in $[a_i, b_i]$, then this node satisfies both properties and we choose it; otherwise, $[a_i, b_i]$ is either entirely to the left or entirely to the right of the key of the current node, so it is contained in the interval associated with the left or right lower neighbor, which we choose as the new current node. Thus, each interval moves down in the search tree till we find a node for which properties 1 and 2 are satisfied. This node might not be unique; if during this descent the key of the current node occurs as an endpoint of the interval, then some node below the current node will also satisfy both properties. For the interval tree structure, it makes no difference which node we choose.
Within the node, there might be multiple intervals that should be stored in that node. We keep the intervals in two lists– one list of the left endpoints in increasing order and one list of the right endpoints in decreasing order. Each interval stored in that node appears on both lists. All left endpoints are smaller than or equal to the key in the node, and all right endpoints are larger than or equal to the key in the node.

Implementation of Interval Trees

By this, we have specified the abstract structure of an interval tree. To implement it, we need two different types of nodes: the search-tree nodes augmented by the left and right list pointers, and the list nodes. The list nodes contain, in addition to the interval endpoint, a pointer to the object associated with the interval. The nodes have the following structure:

```
typedef struct ls_n_t { key_t key;
                        struct ls_n_t *next;
                        object_t *object;
                        } list_node_t;

typedef struct tr_n_t { key_t key;
                        struct tr_n_t *left;
                        struct tr_n_t *right;
                        list_node_t *left_list;
                        list_node_t *right_list;
                        /* balancing information */
                        } tree_node_t;
```

Given the interval tree, we can now describe the query algorithm. For a given value query key, we follow the underlying search-tree structure with its usual find algorithm. In each tree node *n we visit, we output intervals as follows:

1. If query key < n->key
   we setlist to n->left list,
   while list = NULL and list->key ≤ query key.
   1.1 We output list->object and setlist to list->next.
2. Else query key ≥ n->key
   we setlist to n->right list,
   while list = NULL and list->key ≥ query key.
   2.1 We output list->object and setlist to list->next.

In each tree node, we perform O(1) work for each object we list, so the total time is O(h + k), where h is the height of the tree and k is the number of objects listed, so using any balanced search tree as the underlying structure, we get an output-sensitive complexity of O(log n + k).

We still have to show that the output given by this method is correct. For this, we observe that if an interval $[a_i, b_i]$ contains the query key, then it will be stored in one of the tree nodes along the path followed by the query key. On each level, there is at most one node whose associated interval contains $[a_i, b_i]$, and if the query key is in that interval, this path will pass through that node. But for each node, we need to consider only those intervals for which the query key is between the interval endpoint and the node key. Because the node key is contained in all intervals stored in that node, we do not need to check the other interval endpoint.
Thus,

1. If the query key is less than the node key, and the list item key is less than the query key, we have left endpoint ≤ query key < node key ≤ right endpoint. While if the list item key is larger than the node key, this holds by the increasing order of the left list also for all following keys, so none of the remaining intervals contains the query key.
2. If the query key is larger than the node key, and the list item key is larger than the query key, we have left endpoint≤ node key ≤ query key ≤ right endpoint. While if the list item key is smaller than the node key, this holds by the decreasing order of the right list also for all following keys, so none of the remaining intervals contains the query key.

So this algorithm lists exactly the intervals (or associated objects) that contain the query key.
The interval tree is a static data structure, we can build it once, but there is no update operation; insertion and deletion of intervals are not possible. To build it from a given list of n intervals, we first build the search tree for the interval endpoints in O(n log n) time. Next, we construct a list of the intervals sorted in decreasing order of their left interval endpoints, in O(n log n), and find for each interval the node where it should be stored, and insert it there in front of the left list, in O(log n) per interval. Finally, we construct a list of the intervals sorted in increasing order of their right interval endpoints, in O(n log n), and find for each interval the node where it should be stored, and insert it there in front of the right list, in O(log n) per interval. By this initial sorting and inserting in that order, all node lists are in the correct order. The total work needed to construct the interval tree structure is O(n log n). The total

space needed by the interval tree is O(n) because the search tree needs O(n) space and each interval occur only on two lists. This completes the analysis of the interval tree structure.

Theorem: The interval tree structure is a static data structure that can be built in time O(n log n) and needs space O(n). It lists all intervals containing a given query key in output-sensitive time O(log n + k) if there are k such intervals.

Before we now give the code for the query function find intervals, we need to decide how to return multiple results – a question that occurs whenever our query operation has potentially many results. Our preferred solution is to construct a list of all results and return that list as an answer. This has the advantage of conceptual clarity, but it depends on the list nodes being correctly returned by the program that gets this list to avoid a memory leak. The alternative would be to divide the query function in two: one to start the query and one to get the next result.

```
list_node_t *find_intervals(tree_node_t *tree, key_t query_key)
{ tree_node_t *current_tree_node;
  list_node_t *current_list, *result_list, *new_result;
  if( tree->left == NULL )
          return(NULL);
else
{         current_tree_node = tree;
          result_list = NULL;
          while( current_tree_node->right != NULL )
          { if( query_key < current_tree_node->key )
                    { current_list = current_tree_node->left_list;
          while( current_list != NULL && current_list->key <= query_key )
          { new_result = get_list_node();
          new_result->next = result_list;
          new_result->object = current_list->object;
          result_list = new_result;
          current_list = current_list->next;
          }
          current_tree_node =
          current_tree_node->left;
}
```

```
else
{
          current_list = current_tree_node->right_list;
          while( current_list != NULL && current_list->key >= query_key )
          { new_result = get_list_node();
          new_result->next = result_list;
          new_result->object = current_list->object;
          result_list = new_result;
          current_list = current_list->next;
          }
                    current_tree_node =
                    current_tree_node->right;
                    }
          }
          return( result_list );
          }
}
```

There are several problems in making this static data structure dynamic. The simpler problem is that to insert a new interval at the correct node, we need to insert it in the two ordered lists of left and right endpoints. The length of this ordered list can be anything up to n and inserting in an ordered list of length l takes up to (l) time. This could be reduced to O(log l) if we represent the left and right endpoints in a balanced search tree with a doubly connected list of leaves and a pointer to the first and last leaf: then we still have O(k) time to list the first k elements of the list and insertion or deletion time of O(log l) = O(log n).

The other, essentially unsolved, the problem consists of the restructuring of the underlying tree. The interval tree structure depends on each interval containing some key of a tree node. So although not every interval endpoint needs to be a key of the underlying search tree because many tree nodes will not store any intervals, we can be forced to add keys to the underlying search tree. And the tree can become unbalanced by this. But if we wish to rebalance the tree, for example, by rotations, we have to correct the associated lists and this requires that we join two ordered lists that are not separated and that we take apart an ordered list in two, depending on

whether the intervals associated with the list items contain some key value. There is no known way to do this in sublinear time.

If we know in advance some superset of all the interval endpoints that might occur during our use of the structure, we can, of course, build the underlying tree for that superset and that tree will never need to be restructured. This can be a quite efficient solution if that superset is not too large. For the left and right lists in each node, we still need search trees to efficiently insert and delete new intervals.

10.3 Segment Trees

Segment trees were invented by *Bentley*. The primary task performed by a segment tree is the same as that done by an interval tree: keeping track of a set of n intervals, here assumed to be half-open, and listing for a given query key all the intervals that contain that key in output-sensitive time O(log n + k) if the output consisted of k intervals. It is slightly worse at this task than the interval tree having a space requirement of O(n log n) instead of O(n). But the segment tree, or the idea of the canonical interval decomposition on which it is based, is really a framework on which a number of more general tasks can be performed. Again it is a static data structure. Assume a set $X = \{x_1, . . . , x_n\}$ of key values and a search tree T for $\{-\infty\} \cup X$. As usual, with each node of T we associate the interval of all key values for which the query path would go through that node. Any interval $[x_i, x_j[$ can be expressed in many ways as the union of node intervals (Here we need the key$-\infty$ as a leaf of the search tree; otherwise there would be no node interval starting at x_1) so it can be represented by subsets of the tree nodes. In any such representation, a node that is in the tree below some other node is redundant because its node interval is contained in that higher-up node. Among all such representations there is one that is highest: just take all nodes whose intervals are contained in the interval $[x_i, x_j[$ we want to represent and eliminate the redundant nodes. This representation consists of all those nodes whose node interval is contained in $[x_i, x_j[$, but the node interval of their upper neighbor is not contained in $[x_i, x_j[$. This is the canonical interval decomposition of the interval $[x_i, x_j[$ relative to that search tree T.

Theorem: The canonical interval decomposition is a representation of the interval as

the union of disjoint node intervals. Any search path for a value in the interval will go through exactly one node that belongs to the canonical interval decomposition.

The canonical interval decomposition is easy to construct. We start with the interval $[x_i, x_j[$ at the root:

1. Each time the node interval of the current node is entirely contained in $[x_i, x_j[$, we take that node into our representation and stop following that path down because all nodes below are redundant.
2. Each time the node interval of the current node partially overlaps $[x_i, x_j[$, we follow both paths down.
3. Each time the node interval of the current node is disjoint from $[x_i, x_j[$, we stop following that path down.

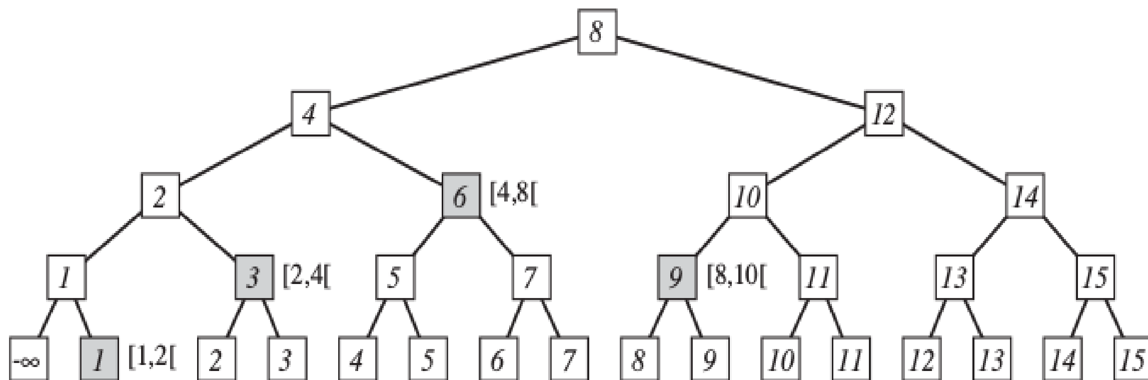


Figure 10.1 Canonical Interval Decomposition for Interval [1, 10]

Source: Advanced Data Structures, Peter Brass, Cambridge University Press, New York, 2008

It is easy to see that this operation selects exactly the nodes of the canonical interval decomposition. It remains to bound the size of the decomposition and the time necessary to construct it. For this, we look at case 2 because it is the only case that does not immediately terminate. Case 2 happens only for those nodes whose node interval contains an endpoint of the interval $[x_i, x_j[$ that we wish to represent, so the nodes for which case 2 is followed are the nodes along the search paths of x_i and x_j. Each of these nodes causes both its lower neighbors to be visited. Because the only way a node that belongs to case 1 or case 3 can be visited is by being a lower neighbor of a node of case 2, the total number of visited nodes is less than 4 height(T) and the total number of selected nodes is less than 2 height(T).

Theorem: Let $X = \{x_1, . . . , x_n\}$ be a set of key values and T a search tree for $\{-\infty\} \cup X$. Then for any interval bounded by values from X, the canonical decomposition has size at most 2 height(T) and can be constructed in time O (height(T)). If T is of height O(log n), the canonical interval decomposition has the size O(log n) and can be found in time O(log n).

Now we have the canonical interval decomposition; the segment tree structure that represents a set of intervals $\{[a_1, b_1[, [a_2, b_2[, . . . , [a_n, b_n[\}$ is easy to describe. It consists of some balanced search tree T for the extended set of interval endpoints $\{-\infty, a_1, a_2, . . . , a_n, b_1, . . . ,b_n\}$ in which each node carries a list of all those intervals $[a_i, b_j[$ for which this node is part of the canonical interval decomposition.

With this structure, the interval containment queries are very easy: given a query key, we follow the search-tree structure down and for each node on the search path, we output all intervals on its list. All these intervals contain the query key, and each interval that contains the query key is met in exactly one node. Thus, the output does not contain any duplicates and the query time is O(log n + k) to follow the search path down and list k intervals. This would work just the same for any other interval decomposition that does not contain redundant elements, but we need the canonical interval decomposition because it is small and easy to build. Unlike the interval tree, each interval is stored in the segment tree many times, so the required space is not only O(n). Each interval generates at most O(log n) parts in its canonical interval decomposition, so the total required space is O(n log n). And the segment tree structure can be built in O(n log n) time, first building the balanced search tree and then inserting the n intervals, constructing the canonical interval decomposition of each in O(log n).

Theorem: The segment tree structure is a static data structure that can be built in time O(n log n) and needs space O(n log n). It lists all intervals containing a given query key in output-sensitive time O(log n + k) if there are k such intervals.

Implementation of Segment Tree Structure

To implement the segment tree structure, we again need two types of nodes – the tree nodes and the interval lists attached to each tree node.

```
typedef struct ls_n_t { key_t key_a, key_b;
        /* interval [a,b[ */
```

```
            struct ls_n_t *next;
            object_t *object;
            } list_node_t;
typedef struct tr_n_t { key_t key;
            struct tr_n_t *left;
            struct tr_n_t *right;
            list_node_t *interval_list;
            /* balancing information */
            } tree_node_t;
```

Then the query algorithm is as follows:

```
list_item_t *find_intervals(tree_node_t *tree, key_t query_key)
            { tree_node_t *current_tree_node;
              list_node_t *current_list, *result_list,
              *new_result;
if( tree->left == NULL )  /* tree empty */
            return(NULL);
else                                    /* tree nonempty, follow search path */
{ current_tree_node = tree;
  result_list = NULL;
  while( current_tree_node->right != NULL )
            { if( query_key < current_tree_node->key )
                        current_tree_node = current_tree_node->left;
            else
            current_tree_node = current_tree_node->right;
            current_list = current_tree_node->interval_list;
  while( current_list != NULL )
   { /* copy entry from node list to result list */
            new_result = get_list_node();
            new_result->next = result_list;
            new_result->key_a = current_list->key_a;
            new_result->key_b = current_list->key_b;
            new_result->object = current_list->object;
            result_list = new_result;
            current_list = current_list->next;
            }
            }
```

```
        return( result_list );
        }
}
```

Notice that neither the root nor any node on the left or right boundary path of the tree can have any intervals of the canonical interval decomposition attached to it because their node intervals are unbounded and we are representing only finite intervals. Typically, nodes near the leaf level will have non-empty lists, whereas, in the interval tree, the intervals tend to be stored in higher-up nodes.

The construction of the segment tree structure has two phases. First, the underlying balanced search tree is built using any method. We assume that initially, all the interval list fields of the tree nodes are NULL. Then the intervals are inserted one after another. Next is the code for the insertion of an interval [a, b[in the tree; the insertion of an interval into the interval list of a node is written as a separate function.

```
void attach_intv_node(tree_node_t *tree_node, key_t a, key_t b, object_t *object)
{ list_node_t *new_node;
        new_node = get_list_node();
        new_node->next = tree_node->interval_list;
        new_node->key_a = a; new_node->key_b = b;
        new_node->object = object;
        tree_node->interval_list = new_node;
}
void insert_interval(tree_node_t *tree, key_t a, key_t b, object_t *object)
{ tree_node_t *current_node, *right_path, *left_path;
list_node_t *current_list, *new_node;
if( tree->left == NULL )
        exit(-1);                       /* tree incorrect */
else
        { current_node = tree;
        right_path = left_path = NULL;
        while( current_node->right != NULL )
                                        /* not at leaf */
        {
        if( b < current_node->key )
```

```
                                        /* go left: a < b < key */
            current_node = current_node->left;
      else if( current_node->key < a)
                                        /* go right: key < b < a */
            current_node = current_node->right;
      else if( a < current_node->key && current_node->key < b )
                                        /* split: a < key < b */
      {
            right_path = current_node->right;
                                        /* both right */
            left_path = current_node->left;
                                        /* and left */
            break;
      }
      else if( a == current_node->key )
                                        /* a = key < b */
      {
            right_path = current_node->right;
                                        /* no left */
            break;
            }
            else                        /*current_node->key == b, so a< key = b */
            {
                  left_path = current_node->left;
                                        /* no right */
                  break;
            }
      }
if( left_path != NULL )
{                                       /* now follow the path of the left endpoint a*/
while( left_path->right != NULL )
{
      if( a < left_path->key )
      {
                                        /* right node must be selected */
            attach_intv_node(left_path-> right, a,b,object);
```

```
            left_path = left_path->left;
        }
        else if ( a == left_path->key )
        {
            attach_intv_node(left_path ->right, a,b,object);
            break;                  /* no further descent necessary */
        }
else
                                    /* go right, no node selected */
        left_path = left_path->right;
        }
                                    /* left leaf of a needs to be selected if reached */
        if( left_path->right == NULL && left_path->key == a )
            attach_intv_node(left_path, a,b,object);
}                                   /* end left path */
        if( right_path != NULL )
            {                       /* and now follow the path of the right endpoint b */
                while( right_path->right != NULL )
                {
                    if( right_path->key < b )
                    {
                        /* left node must be selected */
                        attach_intv_node(right_path->left, a,b, object);
                        right_path = right_path->right;
                    }
                    else if ( right_path->key == b)
                    {
                        attach_intv_node(right_path-> left, a,b, object);
                        break;              /* no further descent necessary */
                    }
                    else                    /* go left, no node selected */
                    right_path = right_path->left;
                }
                        /* on the right side, the leaf of b is never attached */
            }                       /* end right path */
        }
```

}

Again, like the interval tree, the segment tree is a static structure, and we face the same problems in making it dynamic: we have to allow insertion and deletion in each node, and we have to support the restructuring of the underlying tree. For the insertion and deletion in the nodes, we can again use a search tree. But we have to insert or delete the O(log n) fragments of the canonical interval decomposition for a single insert or delete; so it would be efficient to use a search tree only for the first fragment and then have the remaining fragments on a linked list from the first fragment. Then each tree node would need two structures: a search tree for all those intervals whose canonical interval decomposition has its first fragment in that node and a doubly-linked list, allowing O(1) insertion and deletion, for those intervals that started somewhere else. This shows that we can perform O(log n) insertion and deletion of intervals as long as the underlying tree does not change. A rebalancing of the underlying tree by rotations again causes changes in the lists attached to the tree nodes that can be resolved only by looking at the entire list and so this is no efficient solution. The situation here is better than that for interval trees because the sequence of the intervals attached to a tree node does not matter.

10.4 KD- Trees

The kd-tree was invented by *Jon Bentley* (1975) as a direct analog of the normal balanced search tree, which is viewed as a one-dimensional tree. The name kd-tree was originally meant as a k-dimensional tree, where k represented the number of dimensional structures like 3- dimensional or 4- dimensional, etc. The kd-tree is the structure that supports orthogonal range searching. It is quite popular in practical applications and conceptually easy to understand and implement, but it is unsatisfactory because its worst-case performance is much worse than orthogonal range trees. In the two-dimensional version, the worst-case query time is $O(\sqrt{n} + k)$ instead of $O((\log n)^2 + k)$, and the d-dimensional analog is even worse, with $O(n^{(1-1/d)} + k)$ instead of $O((\log n)^d + k)$. The empirical performance in database examples seems better than this worst-case complexity, so in database literature, this and related structures have been widely studied and used.

The lower bound for the query time was given by *Lee* and *Wong* (1977), and a first comparative analysis of several range-searching structures, among them the kd-tree, the orthogonal range tree, and the Bentley–Maurer structures appear in *Bentley* and *Friedman* (1979). The bad worst-case query time places the kd-tree in any comparison far behind these structures, only under strong assumptions like uniformly distributed data points and small, "*relatively square*" query rectangles; its performance becomes comparable to them. Square query rectangles occur when we really aim at a nearest-neighbor query or at least some filter for the neighborhood of the query point. Variants of the kd-tree structure are analyzed in numerous papers under input and query distribution assumptions. Much work went into making kd-trees a dynamic structure, allowing insertions and deletions of points starting with kd-trees, semi-dynamic kd-trees, and divided kd trees.

External memory efficiency has also been a major consideration in these structures; further related structures supporting various types of range-restricted queries have been developed in the database community.

The idea of the kd-tree is that we have a search tree, wherein each node we make a comparison and enter the left or right subtree, but unlike the normal search trees, we can compare in different nodes against different coordinates. The simplest choice is to cycle through the coordinates; in the root, we compare against the first coordinate, in the nodes below, we compare against the second coordinate, and so on. In each node, we choose as a comparison key a value that divides the set of points below that node in a balanced way. As in the normal search trees, this defines a node interval for each node, which is now a d-dimensional half-open box the set of all possible query points whose search path would go through that node.

The comparison with the node key then divides the box by a hyperplane in the direction of that coordinate which we used in the comparison. So we get a hierarchy of possibly unbounded orthogonal boxes. In the two-dimensional version, these are rectangles alternatingly divided in the horizontal and vertical directions.

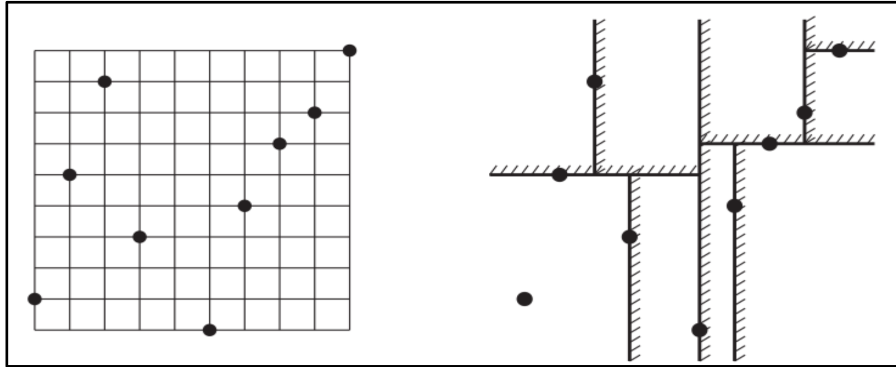


Figure 10.2 Set of Nine Points with kd-Tree Structure: All Rectangles Are Half-Open to the Right and the Top

Source: Advanced Data Structures, Peter Brass, Cambridge University Press, New York, 2008

If we have this structure, a range query can be answered just as in the one-dimensional case: starting in the root, we descend into each node whose node interval has a non-empty intersection with the query region and stop following any branch when that intersection becomes empty. This is a very natural and generic query algorithm that can be applied for any type of query range, not only for rectangles. This is a great strength of this type of structure, but it is not very efficient, for the number of leaves we visit without actually finding a point that should belong to the answer can be as large as $\Omega(\sqrt{n})$. And this is not only for specific bad point sets, or bad subdivision structures; it is a problem that always occurs: there is always a query rectangle that intersects $\Omega(\sqrt{n})$ of the cells without containing any point of the underlying set.

Theorem: kd-trees are a static structure that supports d-dimensional orthogonal range queries in a set of d-dimensional points in output-sensitive time $O(n^{1-1/d} + k)$ if the output consists of k points. They can be built in O(n(log n)) time using O(n) space.

10.5 Quad Trees

Quadtrees were introduced by *Raphael Finkel* and *J.L. Bentley* in 1974. Quadtrees are hierarchical spatial tree data structures that are based on the principle of recursive decomposition of space. The term *quadtree* originated from the representation of two-dimensional data by recursive decomposition of space using

separators parallel to the co-ordinate axis. The resulting split of a region into four regions corresponding to southwest, northwest, southeast, and northeast quadrants is represented as four children of the node corresponding to the region, hence the term"quad" tree. In a three dimensional analog, a region is split into eight regions using planes parallel to the coordinate planes. As each internal node can have eight children corresponding to the 8-way split of the region associated with it, the term octree is used to describe the resulting tree structure. Analogous data structures for representing spatial data in higher than three dimensions are called *hyper octrees*. It is also common practice to use the term quadtrees in a generic way irrespective of the dimensionality of the spatial data. This is especially useful when describing algorithms that are applicable regardless of the specific dimensionality of the underlying data.

In constructing a quadtree, one starts with a square, cubic, or hypercubic region (depending on the dimensionality) that encloses the spatial data under consideration. The different variants of the quadtree data structure are differentiated by the principle used in the recursive decomposition process. One important aspect of the decomposition process is if the decomposition is guided by input data or is based on the principle of equal subdivision of the space itself. The former results in a tree size proportional to the size of the input. If all the input data is available as prior, it is possible to make the data structure height-balanced.

These attractive properties come at the expense of difficulty in making the data structure dynamic, typically in accommodating deletion of data. If the decomposition is based on an equal subdivision of space, the resulting tree depends on the distribution of spatial data. As a result, the tree is height-balanced and is linear in the size of input only when the distribution of the spatial data is uniform, and the height and size properties deteriorate with an increase in non-uniformity of the distribution. The beneficial aspect is that the tree structure facilitates easy update operations and the regularity in the hierarchical representation of the regions facilitates geometric arguments helpful in designing algorithms.

Quadtrees have been used as fixed resolution data structures, where the decomposition stops when a preset resolution is reached, or as variable resolution data structures, where the decomposition stops when a property based on input data present in the region is satisfied. They are also used in a hybrid manner, where the

decomposition is stopped when either a resolution level is reached or when a property is satisfied.

Quadtrees are used to represent many types of spatial data including points, line segments, rectangles, polygons, curvilinear objects, surfaces, volumes, and cartographic data. Their use is pervasive spanning many application areas including computational geometry, computer-aided design, computer graphics, databases, geographic information systems, image processing, pattern recognition, robotics, and scientific computing.

We first explore quadtrees in the context of the simplest type of spatial data multidimensional points. Consider a set of n points in d dimensional space. The principal reason a spatial data structure is used to organize multidimensional data is to facilitate queries requiring spatial information. A number of such queries can be identified for point data.

For example:

1. *Range query:* Given a range of values for each dimension, find all the points that lie within the range. This is equivalent to retrieving the input points that lie within a specified hyper rectangular region. Such a query is often useful in database information retrieval.
2. *Spherical region query:* Given a query point p and a radius r, find all the points that lie within a distance of r from p. In a typical molecular dynamics application, spherical region queries centered around each of the input points is required.
3. *All nearest neighbor query:* Given n points, find the nearest neighbor of each point within the input set.

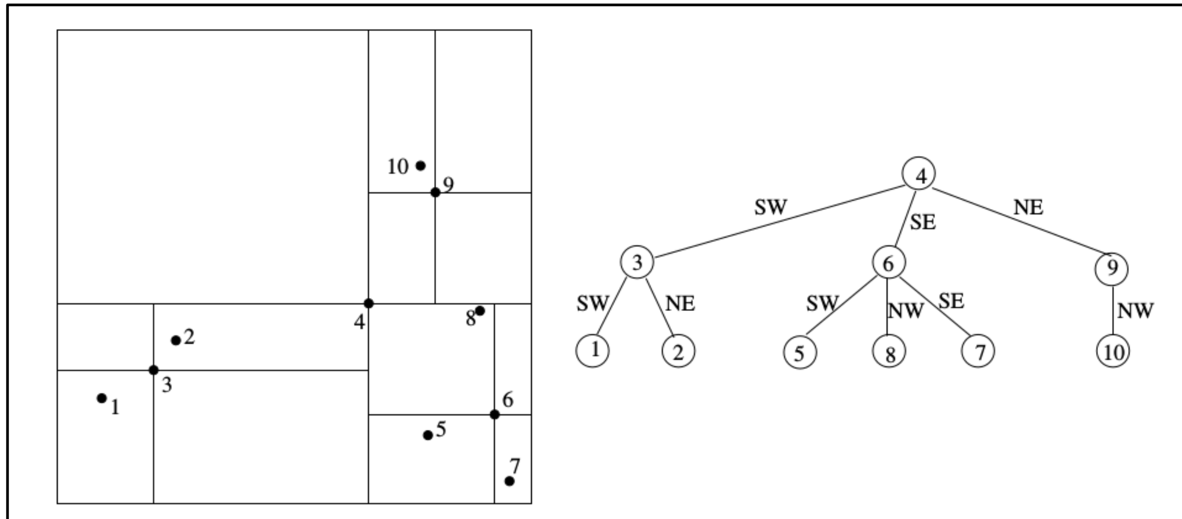


Figure 10.3 A two dimensional set of points and a corresponding point quadtree

Source: Handbook of Data Structures and Applications, Chapman & Hall/CRC, 2005

Point Quadtrees

The point quadtree is a natural generalization of the binary search tree data structure to multiple dimensions. For convenience, first, consider the two-dimensional case. Start with a square region that contains all of the input points. Each node in the point quadtree corresponds to an input point. To construct the tree, pick an arbitrary point and make it the root of the tree. Using lines parallel to the coordinate axis that intersect at the selected point (see figure 10.3), divide the region into four subregions corresponding to the southwest, northwest, southeast, and northeast quadrants, respectively. Each of the subregions is recursively decomposed in a similar manner to yield the point quadtree. For points that lie at the boundary of two adjacent regions, a convention can be adopted to treat the points as belonging to one of the regions. For instance, points lying on the left and bottom edges of a region may be considered included in the region, while points lying on the top and right edges are not. When a region corresponding to a node in the tree contains a single point, it is considered a leaf node. Note that point quadtrees are not unique and their structure depends on the selection of points used in region subdivisions. Irrespective of the choices made, the resulting tree will have n nodes, one corresponding to each input point.

Region Quadtrees

The region quadtree for n points in d dimensions is defined as follows: Consider a hypercube large enough to enclose all the points. This region is represented by the root of the d-dimensional quadtree. The region is subdivided into 2 d subregions of equal size by bisecting along each dimension. Each of these regions containing at least one point is represented as a child of the root node. The same procedure is recursively applied to each child of the root node. The process is terminated when a region contains only a single point. This data structure is also known as the *point region quadtree*. At times, we will simply use the term quadtree when the tree implied is clear from the context. The region quadtree corresponding to a two dimensional set of points is shown in figure 10.4. Once the enclosing cube is specified, the region quadtree is unique. The manner in which a region is subdivided is independent of the specific location of the points within the region.

This makes the size of the quadtree sensitive to the spatial distribution of the points.

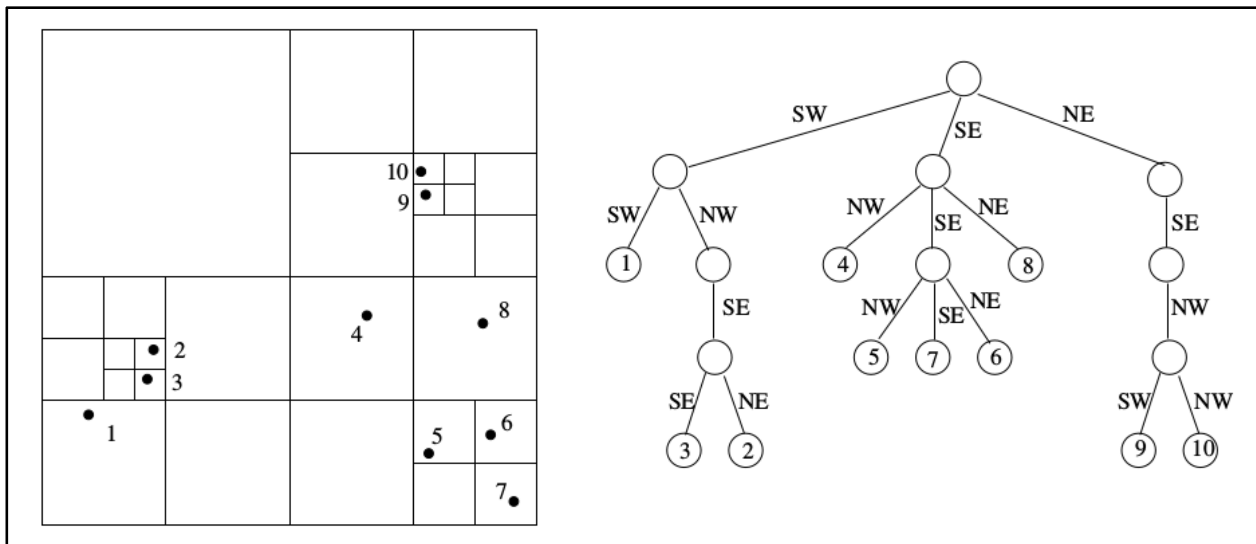


Figure 10.4 A two dimensional set of points and a corresponding region quadtree

Source: Handbook of Data Structures and Applications, Chapman & Hall/CRC, 2005

Compressed Quadtrees and Octrees

In an n-leaf tree where each internal node has at least two children, the number of nodes is bounded by 2n − 1. The size of quadtrees is distribution dependent because there can be internal nodes with only one child. In terms of the cell hierarchy, a cell

may contain all its points in a small volume so that, recursively subdividing it may result in just one of the immediate subcells containing the points for an arbitrarily large number of steps. Note that the cells represented by nodes along such a path have different sizes but they all enclose the same points. In many applications, all these nodes essentially contain the same information as the information depends only on the points the cell contains. This prompted the development of compressed quadtrees, which are obtained by compressing each such path into a single node. Therefore, each node in a compressed quadtree is either a leaf or has at least two children.

The compressed quadtree corresponding to the quadtree is depicted in figure 10.5.

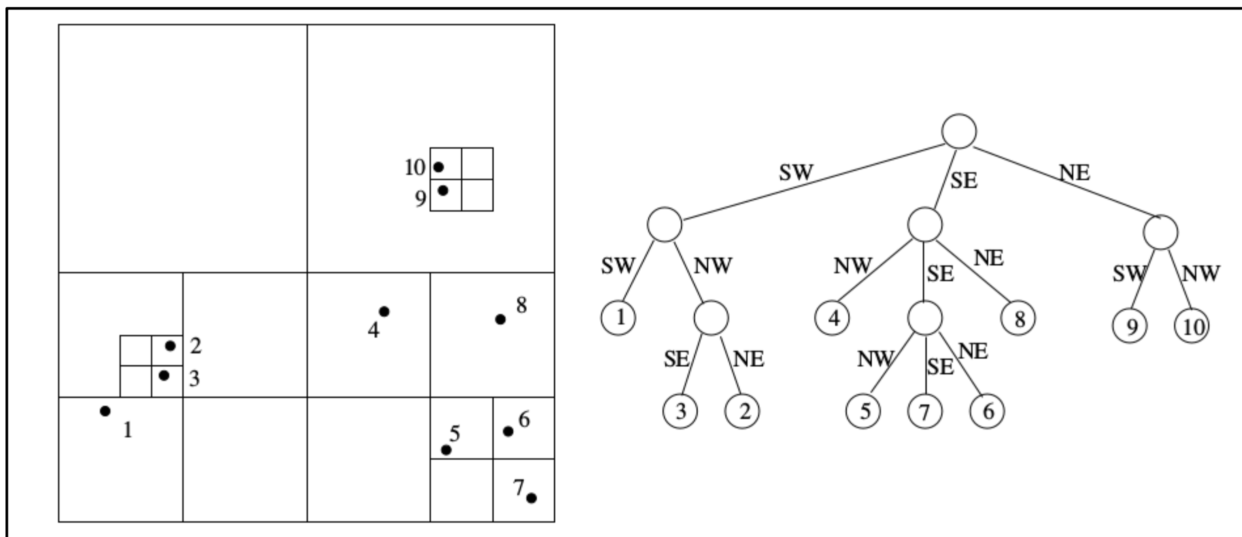


Figure 10.5 A two-dimensional set of points and the corresponding compressed quadtree

Source: Handbook of Data Structures and Applications, Chapman & Hall/CRC, 2005

Fast algorithms for operations on quadtrees can be designed by simultaneously keeping track of spatial ordering and one-dimensional ordering of cells in the compressed quadtree. The spatial ordering is given by the compressed quadtree itself. In addition, a balanced binary search tree (BBST) is maintained on the large cells of the nodes to enable fast cell searches. Both the trees consist of the same nodes and this can be achieved by allowing each node to have pointers corresponding to compressed quadtree structure and pointers corresponding to BBST structure.

Point and Cell Queries

Point and cell queries are similar since a point can be considered to be a zero-length cell. A node v is considered to represent cell C if $S(v) \subseteq C \subseteq L(v)$. The node in the compressed quadtree representing the given cell is located using the BBST. Traverse the path in the BBST from the root to the node that is being searched in the following manner: To decide which child to visit next on the path, compare the query cell with the large and small cells at the node. If the query cell precedes the small cell in cell ordering, continue the search with the left child. If it succeeds the large cell in cell ordering, continue with the right child. If it lies between the small cell and large cell in cell ordering, the node represents the query cell. As the height of a BBST is O(log n), the time is taken for a point or cell query is O(d log n).

Insertions and Deletions

As points can be treated as cells of zero length, insertion and deletion algorithms will be discussed in the context of cells. These operations are meaningful only if a cell is inserted as a leaf node or deleted if it is a leaf node. Note that a cell cannot be deleted unless all its subcells are previously deleted from the compressed quadtree.

Cell Insertion

To insert a given cell C, first check whether it is represented in the compressed quadtree. If not, it should be inserted as a leaf node. Create a node v with S(v) = C and first insert v in the BBST using a standard binary search tree insertion algorithm. To insert v in the compressed quadtree, first find the BBST successor of v, say u. Find the smallest cell D containing C and the S(u). Search for cell D in the BBST and identify the corresponding node w. If w is not a leaf, insert v as a child of w in a compressed quadtree. If w is a leaf, create a new node w' such that S(w') = D. Nodes w and v become the children of w in the compressed quadtree. The new node w' should be inserted in the BBST. The overall algorithm requires a constant number of insertions and searches in the BBST and takes O(d log n) time.

Cell Deletion

As in insertion, the cell should be deleted from the BBST and the compressed quadtree. To delete the cell from BBST, the standard deletion algorithm is used. During the execution of this algorithm, the node representing the cell is found. The node is deleted from the BBST only if it is present as a leaf node in the compressed quadtree. If the removal of this node from the compressed quadtree leaves its parent

with only one child, the parent is deleted as well. Since each internal node has at least two children, the delete operation cannot propagate to higher levels in the compressed quadtree.

10.6 Summary

- The interval tree structure stores a set of intervals and returns for any query key all the intervals that contain this query value.
- The canonical interval decomposition is a representation of the interval as a union of disjoint node intervals. Any search path for a value in the interval will go through exactly one node that belongs to the canonical interval decomposition.
- The kd-tree is the structure that supports orthogonal range searching.
- Quadtrees are hierarchical spatial tree data structures that are based on the principle of recursive decomposition of space.
- Quadtrees are used to represent many types of spatial data including points, line segments, rectangles, polygons, curvilinear objects, surfaces, volumes, and cartographic data.

10.7 Key Terms

- **Octree:** When each internal node can have eight children corresponding to the 8-way split of the region associated with it, it is termed as an octree.
- **Hyper Octrees:** Analogous data structures for representing spatial data in higher than three dimensions are called hyper octrees.
- **Spherical Region Query:** The problem of finding all points in a data set that lie within a given distance from a query point, commonly known as the spherical region query.
- **Supercell:** A cell containing the subcell is called a supercell.

10.8 Check Your Progress

Short- Answer type

Q1) Insertion into a 2-d tree is a trivial extension of insertion into a binary search tree. True/ False?

Q2) In what time can a kd tree be constructed?

(a) O(N) (b) O(N log N) (c) O(N2) (d) O(M log N)

Q3) __________ is the simplest data structure that supports range searching.

Q4) In which of the following data structures does every internal node have at most four children?

(a) Point quadtree (b) Edge quadtree (c) Quadtree (d) None of these

Q5) Point quadtree defines a partition of space in two dimensions by dividing the region into four equal quadrants. True/ False?

Long- Answer type

Q1) Write a short note on interval trees.

Q2) Discuss the types of Quadtrees and their relevant applications.

Q3) What is the basic terminology of KD trees?

Q4) Differentiate between balanced binary trees and Quadtrees.

Q5) The segment tree structure is a static data structure. Explain.

References

- *Handbook of Data Structures and Applications*, Chapman & Hall/CRC, 2005.
- *Advanced Data Structures*, Peter Brass, Cambridge University Press, New York, 2008.

Module: 4

Indexing, Searching & Sorting

Unit: 11 Indexing

Structure

11.0 Introduction

11.1 Unit Objectives

11.2 File Organization

11.3 Indexing

- 11.3.1 Ordered Indices
- 11.3.2 Dense and sparse indices
- 11.3.3 Cylinder surface indices
- 11.3.4 Multi-level indices
- 11.3.5 Inverted indices
- 11.3.6 B-Tree indices
- 11.3.7 Hashed indices

11.4 Summary

11.5 Key Terms

11.6 Check Your Progress

11.0 Introduction

In computer terminology, a file is a block of useful data that is available to a computer program and is usually stored on a persistent storage medium. Storing a file on a persistent storage medium like a hard disk ensures the availability of the file for future use. Every file contains data that can be organized in a hierarchy to present a systematic organization. The data hierarchy includes data items such as fields, records, files, and databases. These terms are defined below.

- A *data field* is an elementary unit that stores a single fact. A data field is usually characterized by its type and size.
- A *record* is a collection of related data fields that is seen as a single unit from the application point of view.
- A *file* is a collection of related records. For example, if there are 60 students in a class, then there are 60 records. All these related records are stored in a file. Similarly, we can have a file of all the employees working in an organization, a file of all the customers of a company, a file of all the suppliers, so on and so

forth.

- A *directory* stores information about related files. A directory organizes information so that users can find it easily.

Every file in a computer system is stored in a directory. Each file has a list of attributes associated with it that gives the operating system and the application software information about the file and how it is intended to be used. A software program that needs to access a file looks up the directory entry to discern the attributes of that file. For example, if a user attempts to write to a file that has been marked as a read-only file, then the program prints an appropriate message to notify the user that he is trying to write to a file that is meant only for reading.
Similarly, there is an attribute called *hidden*. When you execute the DIR command in DOS, then the files whose hidden attribute is set will not be displayed. These attributes are explained here.

- **Filename:** It is a string of characters that stores the name of a file. File naming conventions vary from one operating system to the other.
- **File position:** It is a pointer that points to the position at which the next read/write operation will be performed.
- **File structure:** It indicates whether the file is a text file or a binary file. In the text file, the numbers (integer or floating-point) are stored as a string of characters. A binary file, on the other hand, stores numbers in the same way as they are represented in the main memory.
- **File Access Method:** It indicates whether the records in a file can be accessed sequentially or randomly. In sequential access mode, records are read one by one. That is, if 60 records of students are stored in the STUDENT file, then to read the record of the 39th student, you have to go through the record of the first 38 students. However, in random access, records can be accessed in any order.
- **Attributes Flag:** A file can have six additional attributes attached to it. These attributes are usually stored in a single byte, with each bit representing a specific attribute. If a particular bit is set to '1' then this means that the corresponding attribute is turned on. These attributes are Read-only, Hidden,

System, Volume Label, Directory, Archive.

11.1 Unit Objectives

After going through this unit, the reader will be able to:

- Discuss the basics of file organization and its different methods.
- Learn about various indexing strategies for faster access to data.

11.2 File Organization

The main issue in file management is the way in which the records are organized inside the file because it has a significant effect on the system performance. Organization of records means the logical arrangement of records in the file and not the physical layout of the file as stored on a storage media. Since choosing an appropriate file organization is a design decision, it must be done keeping the priority of achieving good performance with respect to the most likely usage of the file. Therefore, the following considerations should be kept in mind before selecting an appropriate file organization method:

- ➔ Rapid access to one or more records.
- ➔ Ease of inserting/updating/deleting one or more records without disrupting the speed of accessing record(s).
- ➔ Efficient storage of records.
- ➔ Using redundancy to ensure data integrity.

Different file organization methods are available like sequential organization, relative file organization, and indexed sequential file organization.

Sequential Organization

A sequentially organized file stores the records in the order in which they were entered. That is, the first record that was entered is written as the first record in the file, the second record entered is written as the second record in the file, and so on. As a result, new records are added only at the end of the file. Figure 11.1 shows the features, advantages, and disadvantages of sequential organization.

Sequential files can be read-only sequentially, starting with the first record in the file. Sequential file organization is the most basic way to organize a large collection of

records in a file. Once we store the records in a file, we cannot make any changes to the records. We cannot even delete the records from a sequential file. In case we need to delete or update one or more records, we have to replace the records by creating a new file.

In sequential file organization, all the records have the same size and the same field format, and every field has a fixed size. The records are sorted based on the value of one field or a combination of two or more fields. This field is known as the *key*. Each key uniquely identifies a record in a file. Thus, every record has a different value for the key field. Records can be sorted in either ascending or descending order. Sequential files are generally used to generate reports or to perform a sequential reading of large amounts of data which some programs need to do such as payroll processing of all the employees of an organization. Sequential files can be easily stored on both disks and tapes.

Features	Advantages	Disadvantages
• Records are written in the order in which they are entered • Records are read and written sequentially • Deletion or updation of one or more records calls for replacing the original file with a new file that contains the desired changes • Records have the same size and the same field format • Records are sorted on a key value • Generally used for report generation or sequential reading	• Simple and easy to handle • No extra overheads involved • Sequential files can be stored on magnetic disks as well as magnetic tapes • Well suited for batch-oriented applications	• Records can be read only sequentially. If i^{th} record has to be read, then all the i–1 records must be read • Does not support update operation. A new file has to be created and the original file has to be replaced with the new file that contains the desired changes • Cannot be used for interactive applications

Figure 11.1 Sequential Organization

Source: Data Structures using C, Reema Thareja, Oxford University Press, 2nd Edition

Relative File Organization

Relative file organization provides an effective way to access individual records directly. In a relative file organization, records are ordered by their *relative key*. It means the record number represents the location of the record relative to the beginning of the file. The record numbers range from 0 to n–1, where n is the number of records in the file. For example, the record with record number 0 is the first record

in the file. The records in a relative file are of fixed length. Figure 11.2 shows the features, advantages, and disadvantages of Relative file organization.

Therefore, in relative files, records are organized in ascending *relative record numbers*. A relative file can be thought of as a single dimension table stored on a disk, in which the relative record number is the index into the table. Relative files can be used for both random as well as sequential access. For sequential access, records are simply read one after another.

Relative files provide support for only one key, that is, the relative record number. This key must be numeric and must take a value between 0 and the current highest relative record number –1. This means that enough space must be allocated for the file to contain the records with relative record numbers between 0 and the highest record number –1.

Relative file organization provides random access by directly jumping to the record which has to be accessed. If the records are of fixed length and we know the base address of the file and the length of the record, then any record i can be accessed using the following formula:

$$\text{Address of } i^{th} \text{ record} = \text{base_address} + (i-1) * \text{record_length}$$

Note that the base address of the file refers to the starting address of the file.

We took i–1 in the formula because record numbers start from 0 rather than 1.

Features	Advantages	Disadvantages
• Provides an effective way to access individual records • The record number represents the location of the record relative to the beginning of the file • Records in a relative file are of fixed length • Relative files can be used for both random as well as sequential access • Every location in the table either stores a record or is marked as FREE	• Ease of processing • If the relative record number of the record that has to be accessed is known, then the record can be accessed instantaneously • Random access of records makes access to relative files fast • Allows deletions and updations in the same file • Provides random as well as sequential access of records with low overhead • New records can be easily added in the free locations based on the relative record number of the record to be inserted • Well suited for interactive applications	• Use of relative files is restricted to disk devices • Records can be of fixed length only • For random access of records, the relative record number must be known in advance

Figure 11.2 Relative File Organization

(**Source-** Data Structures using C, Reema Thareja, Oxford University Press, 2nd Edition, Chapter- 16, Page No.- 495)

Indexed Sequential File Organization

Indexed sequential file organization stores data for fast retrieval. The records in an indexed sequential file are of fixed length and every record is uniquely identified by a key field. We maintain a table known as the *index table* which stores the record number and the address of all the records. That is for every file, we have an index table. This type of file organization is called an indexed sequential file organization because physically the records may be stored anywhere, but the index table stores the address of those records. The ith entry in the index table points to the ith record of the file. Initially, when the file is created, each entry in the index table contains NULL. When the ith record of the file is written, free space is obtained from the free space manager and its address is stored in the ith location of the index table. Figure 11.3 shows the features, advantages, and disadvantages of Indexed sequential file organization.

An indexed sequential file uses the concept of both sequential as well as relative files. While the index table is read sequentially to find the address of the desired record, direct access is made to the address of the specified record in order to access it randomly.

Features	Advantages	Disadvantages
• Provides fast data retrieval • Records are of fixed length • Index table stores the address of the records in the file • The ith entry in the index table points to the ith record of the file • While the index table is read sequentially to find the address of the desired record, a direct access is made to the address of the specified record in order to access it randomly • Indexed sequential files perform well in situations where sequential access as well as random access is made to the data	• The key improvement is that the indices are small and can be searched quickly, allowing the database to access only the records it needs • Supports applications that require both batch and interactive processing • Records can be accessed sequentially as well as randomly • Updates the records in the same file	• Indexed sequential files can be stored only on disks • Needs extra space and overhead to store indices • Handling these files is more complicated than handling sequential files • Supports only fixed length records

Figure 11.3 Indexed Sequential File Organization

(**Source-** Data Structures using C, Reema Thareja, Oxford University Press, 2nd Edition, Chapter- 16, Page No.- 496)

11.3 Indexing

An index for a file can be compared with a catalog in a library. Like a library has card catalogs based on authors, subjects, or titles, a file can also have one or more indices.

Indexed sequential files are very efficient to use, but in real-world applications, these files are very large and a single file may contain millions of records. Therefore, in such situations, we require a more sophisticated indexing technique. There are several indexing techniques and each technique works well for a particular application. For a particular situation at hand, we analyze the indexing technique based on factors such as access type, access time, insertion time, deletion time, and space overhead involved. Let's discuss the different types of indices.

11.3.1 Ordered Indices

Indices are used to provide fast random access to records. As stated above, a file may have multiple indices based on different key fields. An index of a file may be a primary index or a secondary index.

Primary Index

In a sequentially ordered file, the index whose search key specifies the sequential order of the file is defined as the primary index. For example, suppose records of students are stored in a STUDENT file in a sequential order starting from roll number 1 to roll number 60. Now, if we want to search a record for, say, roll number 10, then the student's roll number is the primary index. Indexed sequential files are a common example where a primary index is associated with the file.

Secondary Index

An index whose search key specifies an order different from the sequential order of the file is called the secondary index. For example, if the record of a student is searched by his name, then the name is a secondary index. Secondary indices are used to improve the performance of queries on non-primary keys.

11.3.2 Dense and Sparse Indices

In a dense index, the index table stores the address of every record in the file. However, in a sparse index, the index table stores the address of only some of the records in the file. Although sparse indices are easy to fit in the main memory, a dense index would be more efficient to use than a sparse index if it fits in the memory. Figure 11.4 shows a dense index and a sparse index for an indexed sequential file.

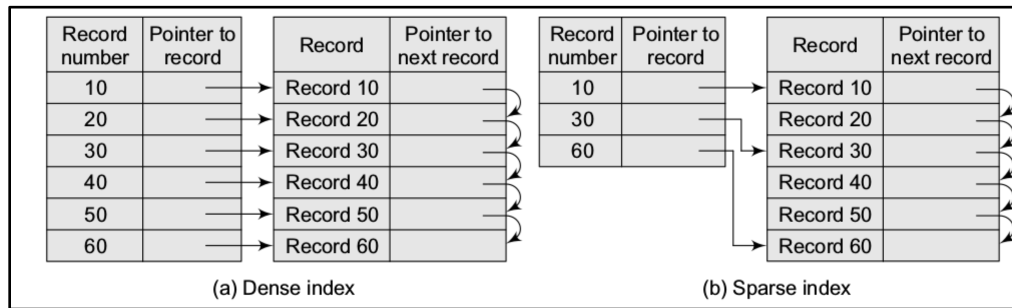


Figure 11.4 Dense and Sparse index

Source: Data Structures using C, Reema Thareja, Oxford University Press, 2nd Edition

Note that the records need not be stored in consecutive memory locations. The pointer to the next record stores the address of the next record.

By looking at the dense index, it can be concluded directly whether the record exists in the file or not. This is not the case in a sparse index. In a sparse index, to locate a record, we first find an entry in the index table with the largest search key value that is either less than or equal to the search key value of the desired record. Then, we start at that record pointed to by that entry in the index table and then proceed to search the record using the sequential pointers in the file, until the desired record is obtained. For example, if we need to access record number 40, then record number 30 is the largest key value that is less than 40. So jump to the record pointed by record number 30 and move along the sequential pointer to reach record number 40.

Thus we see that sparse index takes more time to find a record with the given key. Dense indices are faster to use, while sparse indices require less space and impose less maintenance for insertions and deletions.

11.3.3 Cylinder Surface Indexing

Cylinder surface indexing is a very simple technique used only for the primary key index of a sequentially ordered file. In a sequentially ordered file, the records are stored sequentially in the increasing order of the primary key. The index file will contain two fields- cylinder index and several surface indices. Generally, there are multiple cylinders, and each cylinder has multiple surfaces. If the file needs m cylinders for storage then the cylinder index will contain m entries.

Each cylinder will have an entry corresponding to the largest key value into that

cylinder. If the disk has n usable surfaces, then each of the surface indices will have n entries. Therefore, the ith entry in the surface index for cylinder *j* is the largest key value on the jth track of the ith surface. Hence, the total number of surface index entries is m.n. The physical and logical organization of the disk is shown in figure 11.5. It should be noted that the number of cylinders in a disk is only a few hundred and the cylinder index occupies only one track.

When a record with a particular key value has to be searched, then the following steps are performed:

- First, the cylinder index of the file is read into memory.
- Second, the cylinder index is searched to determine which cylinder holds the desired record. For this, either the binary search technique can be used or the cylinder index can be made to store an array of pointers to the starting of individual key values. In either case, the search will take O (log m) time.
- After the cylinder index is searched, the appropriate cylinder is determined.
- Depending on the cylinder, the surface index corresponding to the cylinder is then retrieved from the disk.
- Since the number of surfaces on a disk is very small, the linear search can be used to determine the surface index of the record.
- Once the cylinder and the surface are determined, the corresponding track is read and searched for the record with the desired key.

Hence, the total number of disk accesses is three—first, for accessing the cylinder index, second for accessing the surface index, and third for getting the track address. However, if track sizes are very large then it may not be a good idea to read the whole track at once. In such situations, we can also include sector addresses. But this would add an extra level of indexing and, therefore, the number of accesses needed to retrieve a record will then become four. In addition to this, when the file extends over several disks, a disk index will also be added.

The cylinder surface indexing method of maintaining a file and index is referred to as Indexed Sequential Access Method Sectors (ISAM). This technique is the most popular and simplest file organization in use for figure 11.5. Physical and logical organization of disk single key values. But with files that contain multiple keys, it is not possible to

use this index organization for the remaining keys.

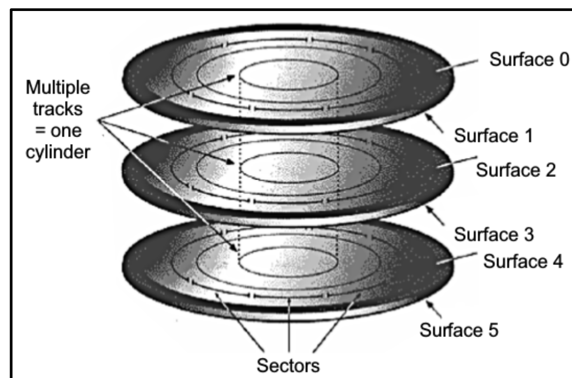


Figure 11.5 Physical and logical organization of disk

Source- Data Structures using C, Reema Thareja, Oxford University Press, 2nd Edition

11.3.4 Multi-level Indices

In real-world applications, we have very large files that may contain millions of records. For such files, a simple indexing technique will not suffice. In such a situation, we use multi-level indices. To understand this concept, consider a file that has 10,000 records. If we use simple indexing, then we need an index table that can contain at least 10,000 entries to point to 10,000 records. If each entry in the index table occupies 4 bytes, then we need an index table of 4 x 10000 bytes = 40000 bytes. Finding such a big space consecutively is not always easy. So, a better scheme is to index the index table. We can continue further by having three-level indexing and so on. But practically, we use two-level indexing. Note that two and higher-level indexing must always be sparse, otherwise multi-level indexing will lose its effectiveness.

11.3.5 Inverted Indices

Inverted files are commonly used in document retrieval systems for large textual databases. An inverted file reorganizes the structure of an existing data file in order to provide fast access to all records having one field falling within the set limits. For example, inverted files are widely used by bibliographic databases that may store author names, title words, journal names, etc. When a term or keyword specified in the inverted file is identified, the record number is given and a set of records corresponding to the search criteria are created. Thus, for each keyword, an inverted file contains an inverted list that stores a list of pointers to all occurrences of that term

in the main text. Therefore, given a keyword, the addresses of all the documents containing that keyword can easily be located. There are two main variants of inverted indices:

- A record-level inverted index (also known as *inverted file index* or *inverted file*) stores a list of references to documents for each word.
- A word-level inverted index (also known as *full inverted index* or *inverted list*) in addition to a list of references to documents for each word also contains the positions of each word within a document. Although this technique needs more time and space, it offers more functionality (like phrase searches).

Therefore, the inverted file system consists of an index file in addition to a document file (also known as a *text file*). It is this index file that contains all the keywords which may be used as search terms. For each keyword, an address or reference to each location in the document where that word occurs is stored. There is no restriction on the number of pointers associated with each word.

For efficiently retrieving a word from the index file, the keywords are sorted in a specific order (usually alphabetically). However, the main drawback of this structure is that when new words are added to the documents or text files, the whole file must be reorganized. Therefore, a better alternative is to use B-trees.

11.3.6 B-Tree Indices

A database is defined as a collection of data organized in a fashion that facilitates updating, retrieving, and managing the data (that may include any item, such as names, addresses, pictures, and numbers). Most organizations maintain databases for their business operations. For example, an airline reservation system maintains a database of flights, customers, and tickets issued. A university maintains a database of all its students. These real-world databases may contain millions of records that may occupy gigabytes of storage space.

For a database to be useful, it must support fast retrieval and storage of data. Since it is impractical to maintain the entire database in the memory, B-trees are used to index the data in order to provide fast access.

For example, searching a value in an un-indexed and unsorted database containing n

key values may take a running time of *O(n)* in the worst case, but if the same database is indexed with a B-tree, the search operation will run in *O(log n)* time. The majority of the database management systems use the B-tree index technique as the default indexing method. This technique supersedes other techniques of creating indices, mainly due to its data retrieval speed, ease of maintenance, and simplicity. Figure 11.6 shows a B-tree index.

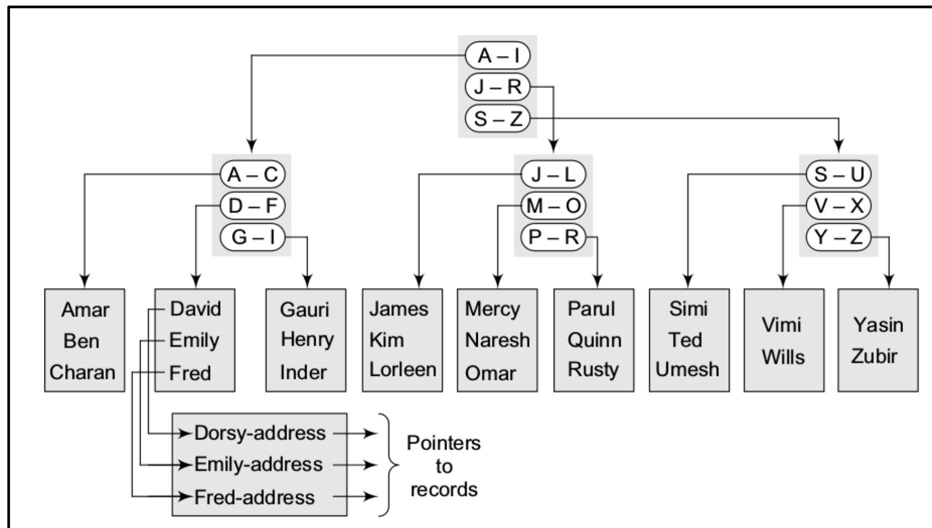


Figure 11.6 B-tree index

Source- Data Structures using C, Reema Thareja, Oxford University Press, 2nd Edition

It forms a tree structure with the root at the top. The index consists of a B-tree (balanced tree) structure based on the values of the indexed column. In this example, the indexed column is the *name* and the B-tree is created using all the existing names that are the values of the indexed column. The upper blocks of the tree contain index data pointing to the next lower block, thus forming a hierarchical structure. The lowest level blocks, also known as leaf blocks, contain pointers to the data rows stored in the table.

If a table has a column that has many unique values, then the selectivity of that column is said to be high. B-tree indices are most suitable for highly selective columns, but it causes a sharp increase in the size when the indices contain a concatenation of multiple columns. The B-tree structure has the following advantages:

- Since the leaf nodes of a B-tree are at the same depth, retrieval of any record from anywhere in the index takes approximately the same time.
- B-trees improve the performance of a wide range of queries that either search a value having an exact match or for a value within a specified range.
- B-trees provide fast and efficient algorithms to insert, update, and delete records that maintain the key order.
- B-trees perform well for small as well as large tables. Their performance does not degrade as the size of a table grows.
- B-trees optimize costly disk access.

11.3.7 Hashed Indices

Hashing is used to compute the address of a record by using a hash function on the search key value. If at any point of time, the hashed values map to the same address, then a collision occurs, and schemes to resolve these collisions are applied to generate a new address. Choosing a good hash function is critical to the success of this technique. By a good hash function, we mean two things. First, a good hash function, irrespective of the number of search keys, gives an average-case lookup that is a small constant. Second, the function distributes records uniformly and randomly among the buckets, where a bucket is defined as a unit of one or more
records (typically a disk block). Correspondingly, the worst hash function is one that maps all the keys to the same bucket. However, the drawback of using hashed indices includes:

- Though the number of buckets is fixed, the number of files may grow with time.
- If the number of buckets is too large, storage space is wasted.
- If the number of buckets is too small, there may be too many collisions.

It is recommended to set the number of buckets to twice the number of the search key values in the file. This gives a good space–performance tradeoff.

A hashed file organization uses hashed indices. Hashing is used to calculate the address of the disk block where the desired record is stored. If K is the set of all search key values and B is the set of all bucket addresses, then a hash function H maps K to B.

11.4 Summary

- File organization means the logical arrangement of records in the file. Files can be organized as sequential, relative, or index sequential.
- A sequentially organized file stores record in the order in which they were entered.
- In relative file organization, records in a file are ordered by their relative key. Relative files can be used for both random access as well as sequential access of data.
- In an indexed sequential file, every record is uniquely identified by a key field. We maintain a table known as the index table that stores the record number and the address of the record in the file.
- In a dense index, the index table stores the address of every record in the file. However, in a sparse index, the index table stores the address of only some of the records in the file.
- Cylinder surface indexing is a very simple technique that is used only for the primary key index of a sequentially ordered file.
- The majority of the database management systems use the B-tree indexing technique. The index consists of a hierarchical structure with upper blocks containing indices pointing to the lower blocks and lowest level blocks containing pointers to the data records.
- Hashed file organization uses hashed indices. Hashing is used to calculate the address of the disk block where the desired record is stored. If K is the set of all search key values and B is the set of bucket addresses, then a hash function H maps K to B.

11.5 Key Terms

- **Primary Index:** The index whose search key specifies the sequential order of the file is defined as the primary index.
- **Secondary Index:** An index whose search key specifies an order different from the sequential order of the file is called the secondary index.
- **Record-level inverted index:** It stores a list of references to documents for each word.
- **Word-level inverted index:** In addition to a list of references to documents for

each word also contains the positions of each word within a document.

- **Index Table:** It stores the record number and the address of all the records.

11.6 Check Your Progress

Short- Answer type

Q1) ______ files are frequently used in indexing techniques in document retrieval systems for large textual databases.

Q2) Which of the following indexing techniques is used in document retrieval systems for large databases?

(a) Inverted index (b) Multi-level indices (c) Hashed indices (d) B-tree index

Q3) B-tree indices are most suitable for highly selective columns. True/ False?

Q4) Index table stores ______ and ______ of the record in the file.

Q5) Relative files can be used for both random access of data as well as sequential access. True/ False?

Long- Answer type

Q1) Explain the terms field, record, file organization, key, and index.

Q2) Differentiate between the sparse index and dense index.

Q3) Give the merits and demerits of a B-tree index.

Q4) Explain the features of Indexed sequential file organization.

Q5) Briefly explain the different types of indices.

References

- *Data Structures using C,* Reema Thareja, Oxford University Press, 2nd Edition

Unit: 12 Searching

Structure

12.0 Introduction
12.1 Unit Objectives
12.2 Searching and its types
    12.2.1 Linear Search
    12.2.2 Binary Search
12.3 Interpolation Search
12.4 Jump Search
12.5 Comparison of different search algorithms
12.6 Summary
12.7 Key Terms
12.8 Check Your Progress

12.0 Introduction

In real life, we generally arrange our things in a particular order so that we can refer to them quickly and easily. Words in the dictionary, for example, are arranged in alphabetical order to facilitate easy and fast searching. In the same way, the large amounts of data stored in computer systems also need to be organized (sorted) in some logical manner so that individual records can be searched easily and efficiently. If the data is kept in a non-orderly manner, then the searching becomes a tedious task. Therefore, sorting and searching are the most basic and commonly performed operations in computer systems. This unit will discuss how various structures are used for searching the data.

As mentioned, searching and sorting are the two most useful operations that are to be performed on a list, which is maintained either in an array or in the linked list. To perform search operations on a given list, various techniques are available, namely linear search, binary search, and hashing.

The linear search algorithm can be applied on an array or on a linked list. It does not require any additional data structure to perform the search operation. The other search algorithm, i.e, a binary search algorithm can be applied on a list maintained in an array only. However, like linear search, it also does not require any additional data

structure for performing search operations. Hashing uses a data structure called *hash table* which is merely an array of fixed size and elements in it are inserted and searched using a function called a hash function. In general, linear search algorithms are slower than binary search, which, in turn, is slower than hashing. A comparison of the linear and binary search algorithms is provided in this unit.

12.1 Unit Objectives

After going through this unit, the reader will be able to:

- Explain the use of various data structures for searching.
- Describe the basics of linear search and binary search.
- Discuss the comparison between various search algorithms.

12.2 Searching and its types

While solving a problem, a programmer may need to search for a value in an array. The process of finding the occurrence of a particular data item in a list is known as *searching*. Search is said to be successful or unsuccessful depending on whether the data item is found or not. Searching means to find whether a particular value is present in an array or not. If the value is present in the array, then searching is said to be successful and the searching process gives the location of that value in the array. However, if the value is not present in the array, the searching process displays an appropriate message and in this case, searching is said to be unsuccessful. The two main search techniques are *linear search* and *binary search*. The algorithm that should be used depends entirely on how the values are organized in the array. For example, if the elements of the array are arranged in ascending order, then the binary search should be used, as it is more efficient for sorted lists in terms of complexity. Let's discuss all these methods in detail.

12.2.1 Linear Search

The linear search is one of the simplest searching techniques. In this technique, the array is traversed sequentially from the first element until the value is found or the end of the array is reached. While traversing, each element of the array is compared with the value to be searched, and if the value is found, the search is said to be

successful. This technique is suitable for performing a search in a small array or in an unsorted array.

Algorithm 12.1 Linear Search

linear_search(ARR, size, item)
1. Set i = 0
2. While i < size
    If ARR[i] = item   //item is the value to be searched
        Return i and go to step 4
    End If
    Set i = i + 1
  End While
3. Return -1   //search unsuccessful
4. End

Program 12.1: Write a program to perform linear search.

```
#include<stdio.h>
#include<conio.h>
#define MAX 10
/*Function prototype*/
int linear_search(int [], int, int);
void main()
{
        int ARR[MAX];
        int item, size, pos, i;
        do
        {
                clrscr();
        printf("\nEnter the size of the array (max %d): ",MAX);
        scanf("%d", &size);
}while(size>MAX);
printf("\nEnter elements of the array:\n");
for(i=0;i<size;i++)
        scanf("%d", &ARR[i]);
printf("\nEnter the element to be searched: ");
```

```
scanf("%d", &item);
pos=linear_search(ARR, size, item);
if (pos==-1)
        printf("\nElement not found");
else
        printf("\nElement found at location: %d", pos+1);
getch();
}
int linear_search(int ARR[], int size, int item)
{
        int i;
        for (i=0;i<size;i++)
        {
                if(ARR[i]==item)
                        return i;
        }
        return -1;
}
```

The output of the program is:

Enter the size of the array (max 10): 5

Enter elements of the array:

1

4

3

6

7

Enter the element to be searched: 4

Element found at location: 2

Analysis of linear search

In the best case, when the element is found at the first position, the search operation terminates successfully with only one comparison. Thus, in this case, the complexity of the algorithm is O(1). In the worst case, when the element to be searched appears at the end of the list or is not present in the list, linear search requires n

comparisons. In both cases, the average complexity of the linear search is O(n).

12.2.2 Binary Search

The binary search technique is used to search a particular element in a sorted (in ascending or descending order) array. In this technique, the element to be searched (say, item) is compared with the middle element of the array. If an item is equal to the middle element, then the search is successful. If an item is smaller than the middle element, the item is searched in the segment of the array before the middle element. However, if the item is greater than the middle element, the item is searched in the array segment after the middle element. This process is repeated until the element is found or the array segment is reduced to a single element that is not equal to the item. At every stage of the binary search technique, the array is reduced to a smaller segment. It searches a particular element in the lowest possible number of comparisons. Hence, the binary search technique is used for larger and sorted arrays, as it is faster as compared to linear search. Consider, for example, an array ARR shown in figure 12.1.

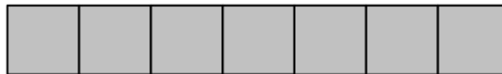
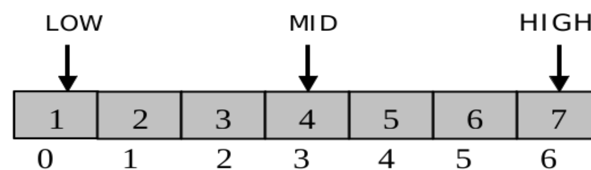


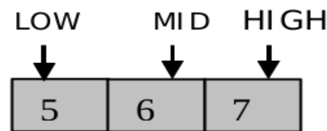
Figure 12.1 The Array ARR

To search an item (say,7) using binary search in the array ARR with size=7, these steps are performed.

1. Initially, set LOW=0 and HIGH= size–1. The middle of the array is determined using the formula MID=(LOW+HIGH)/2, that is, MID=(0+6)/2, which is equal to 3. Thus, ARR[MID]=4.



2. Since the value stored at ARR[3] is less than the value to be searched, that is, 7, the search process is now restricted from ARR[4] to ARR[6]. Now LOW is 4 and HIGH is 6. The middle element of this segment of the array is calculated as MID = (4+6)/2, that is, 5. Thus, ARR[MID]=6.



3. The value stored at ARR[5] is less than the value to be searched, hence the search process begins from the subscript 6. As ARR[6] is the last element, the item to be searched is compared with this value. Since ARR[6] is the value to be searched, the search is successful.

4.

Algorithm 12.2 Binary Search

binary_search(ARR, size, item)
1. Set LOW = 0
2. Set HIGH = size - 1
3. While LOW <= HIGH
        Set MID = (LOW + HIGH) / 2
        If ITEM = ARR [MID]
                Return MID and go to step 5
        Else If item < ARR [MID]
                Set HIGH = MID – 1
        Else
            Set LOW = MID + 1
       End If
     End While
4. Return -1
5. End

Program 12.2: Write a program to perform binary search.

```
#include<stdio.h>
#include<conio.h>
#define MAX 10
/*Function prototype*/
int binary_search(int [], int, int);
void main()
{
```

```
        int ARR[MAX], size, item, pos, i;
        do
        {
                clrscr();
                printf ("\nEnter the size of the array (max %d): ", MAX);
                scanf("%d", &size);
        }while(size>MAX);
        printf("\nEnter elements in sorted order: ");
        for(i=0;i<size;i++)
                scanf("%d", &ARR[i]);
        printf("\nEnter the element to be searched:");
        scanf("%d", &item);
        pos=binary_search(ARR, size, item);
        if (pos==-1)
                printf("\nElement not found");
        else
                printf("\nElement found at location: %d ", pos+1);
        getch();
}
int binary_search(int ARR[], int size, int item)
{
        int LOW = 0;
        int HIGH = size - 1;
        int MID;
        while(LOW<=HIGH)
        {
                MID=(HIGH+LOW)/2;
                if(ARR[MID]==item)
                        return MID;
                else
                        if(item<ARR[MID])
                                HIGH=MID-1;
                        else
                                LOW=MID+1;
        }
        return -1;
```

}

The output of the program is:

Enter no of elements (max 10): 5

Enter elements in sorted order: 11

22

33

44

55

Enter the element to be searched: 33

Element found at location: 3

Analysis of binary search

In each iteration, the binary search algorithm reduces the array to one half. Therefore, for an array containing n elements, there will be $\log_2 n$ iterations. Thus, the complexity of binary search algorithms is $O(\log_2 n)$. This complexity will be the same irrespective of the position of the element, even if the element is not present in the list.

12.3 Interpolation Search

Interpolation search, also known as extrapolation search, is a searching technique that finds a specified value in a sorted array. The concept of interpolation search is similar to how we search for names in a telephone book or for keys by which a book's entries are ordered. For example, when looking for the name "James" in a telephone directory, we know that it will be near the extreme left, so applying a binary search technique by dividing the list into two halves each time is not a good idea. We must start scanning the extreme left in the first pass itself.

In each step of interpolation search, the remaining search space for the value to be found is calculated. The calculation is done based on the values at the bounds of the search space and the value to be searched. The value found at this estimated position is then compared with the value being searched for. If the two values are equal, then the search is complete.

However, in case the values are not equal then depending on the comparison, the

remaining search space is reduced to the part before or after the estimated position. Thus, we see that the interpolation search is similar to the binary search technique. However, the important difference between the two techniques is that binary search always selects the middle value of the remaining search space. It discards half of the values based on the comparison between the value found at the estimated position and the value to be searched. But in interpolation search, interpolation is used to find an item near the one being searched for, and then the linear search is used to find the exact item.

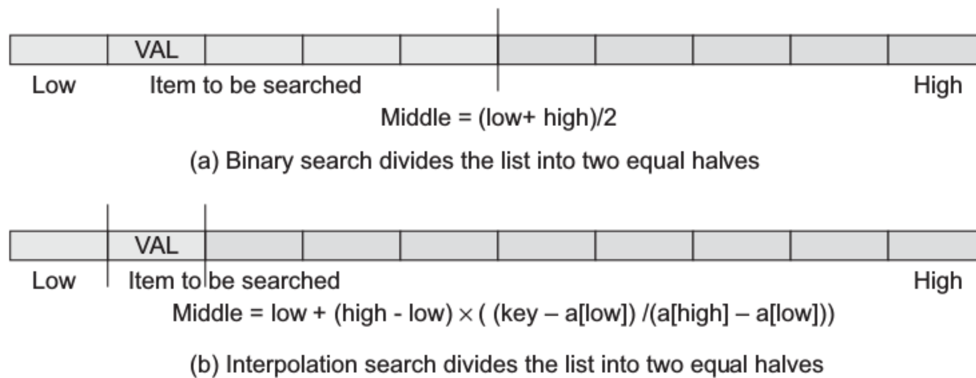


Figure 12.2 Difference between binary search and interpolation search

Algorithm 12.3 Interpolation Search

Interpolation_Search (A, lower_bound, upper_bound, VAL)
1. SET LOW = lower_bound, HIGH = upper_bound, POS = –1
2. Repeat Steps 3 to 4 while LOW <= HIGH
3. SET MID = LOW + (HIGH – LOW) × ((VAL – A[LOW]) / (A[HIGH] – A[LOW]))
4. IF VAL = A[MID]
  POS = MID
  PRINT POS
  Go to Step 6
  ELSE IF VAL < A[MID]
      SET HIGH = MID – 1
   ELSE
      SET LOW = MID + 1
   IF POS = –1
      PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
5. End

Analysis of Interpolation Search

When n elements of a list to be sorted are uniformly distributed (average case), interpolation search makes about log(log n) comparisons. However, in the worst case, that is when the elements increase exponentially, the algorithm can make up to O(n) comparisons.

Program 12.3: Write a program to search an element in an array using interpolation search.

```
#include <stdio.h>
#include <conio.h>
#define MAX 20
int interpolation_search(int a[], int low, int high, int val)
{
	int mid;
	while(low <= high)
	{
		mid = low + (high – low)*((val – a[low]) / (a[high] – a[low]));
		if(val == a[mid])
			return mid;
		if(val < a[mid])
			high = mid – 1;
		else
			low = mid + 1;
	}
	return –1;
}
int main()
{
	int arr[MAX], i, n, val, pos;
	clrscr();
	printf("\n Enter the number of elements in the array : ");
	scanf("%d", &n);
	printf("\n Enter the elements : ");
	for(i = 0; i <n; i++)
```

```
        scanf("%d", &arr[i]);
    printf("\n Enter the value to be searched : ");
    scanf("%d", &val);
    pos = interpolation_search(arr, 0, n–1, val);
    if(pos == –1)
        printf("\n %d is not found in the array", val);
    else
        printf("\n %d is found at position %d", val, pos);
    getch();
    return 0;
}
```

12.4 Jump Search

When we have an already sorted list, then the other efficient algorithm to search for a value is jump search or block search. In jump search, it is not necessary to scan all the elements in the list to find the desired value. We just check an element and if it is less than the desired value, then some of the elements following it are skipped by jumping ahead. After moving a little forward again, the element is checked. If the checked element is greater than the desired value, then we have a boundary and we are sure that the desired value lies between the previously checked element and the currently checked element. However, if the checked element is less than the value being searched for, then we again make a small jump and repeat the process.

Once the boundary of the value is determined, a linear search is done to find the value and its position in the array. For example, consider an array a[ ] = {1,2,3,4,5,6,7,8,9} . The length of the array is 9. If we have to find value 8 then the following steps are performed using the jump search technique.

Step 1: First three elements are checked. Since 3 is smaller than 8, we will have to make a jump ahead

1	2	3	4	5	6	7	8	9

Step 2: Next three elements are checked. Since 6 is smaller than 8, we will have to make a jump ahead

1	2	3	4	5	6	7	8	9

Step 3: Next three elements are checked. Since 9 is greater than 8, the desired value lies within the current boundary

1	2	3	4	5	6	7	8	9

Step 4: A linear search is now done to find the value in the array.

Figure 12.3 An example of Jump search

Algorithm 12.4 Jump Search

```
Jump_Search (A, lower_bound, upper_bound, VAL, N)
1. Set STEP = sqrt(N), I = 0, LOW = lower_bound, HIGH = upper_bound, POS = –1
2. Repeat Step 3 while I < STEP
3. If VAL < A
                Set HIGH = STEP – 1
   else
                Set LOW = STEP + 1
        Set I = I + 1
4. Set I = LOW
5. Repeat Step 6 while I <= HIGH
6. IF A[I] = Val
        POS = I
        PRINT POS
        Go to Step 8
        Set I = I + 1
7. IF POS = –1
        PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
8. End
```

Analysis of Jump Search

Jump search works by jumping through the array with a step size (optimally chosen to be $\sqrt{n}$) to find the interval of the value. Once this interval is identified, the value is searched using the linear search technique. Therefore, the complexity of the jump search algorithm can be given as $O(\sqrt{n})$.

Program 12.4: Write a program to search an element in an array using jump search.

```
#include <stdio.h>
#include <math.h>
#include <conio.h>
#define MAX 20
int jump_search(int a[], int low, int high, int val, int n)
{
        int step, i;
        step = sqrt(n);
        for(i=0;i<step;i++)
        {
```

```
                if(val < a[step])
                        high = step – 1;
                else
                        low = step + 1;
        }
        for(i=low;i<=high;i++)
        {
           if(a[i]==val)
                   return i;
        }

return –1;
}
int main()
{
        int arr[MAX], i, n, val, pos;
        clrscr();
        printf("\n Enter the number of elements in the array : ");
        scanf("%d", &n);
        printf("\n Enter the elements : ");
        for(i = 0; i <n; i++)
                scanf("%d", &arr[i]);
        printf("\n Enter the value to be searched : ");
        scanf("%d", &val);
        pos = jump_search(arr, 0, n–1, val, n);
        if(pos == –1)
                printf("\n %d is not found in the array", val);
        else
                printf("\n %d is found at position %d", val, pos);
        getch();
        return 0;
}
```

12.5 Comparison of Different Search Algorithms

To compare the linear and binary search algorithms, consider an array ARR of ten

elements shown in Figure 12.4, and we have to find element 18 in this array. Using linear search, element 18 is compared with each element of ARR sequentially from the first element, that is, 11 until either 18 is found or the end of the array is encountered. Hence, it makes eight comparisons as 18 is present at 8th position in ARR. On the other hand, using binary search, element 18 is compared with the middle element of ARR (that is, 15). Since 18 is not equal to 15, the search list is reduced into two halves, and the search proceeds in the second half (6th to 10th element). Now, 18 is compared with the middle element of the second half, that is, 18. This makes the search successful in just two comparisons.

11	12	13	14	15	16	17	18	19	20
0	1	2	3	4	5	6	7	8	9

Figure 12.4 An Array ARR of Ten Elements

It is clear from the above example that the performance of binary search is better than the linear search. This is because the binary search reduces the search list to its half in each iteration, thus, requiring less number of comparisons. However, it only works on the sorted lists which is the main disadvantage of the binary search. Moreover, implementing the binary search algorithm is more complex than linear search.

12.6 Summary

- The process of finding the occurrence of a particular data item in a list is known as searching.
- The two main search techniques are *linear search* and *binary search*.
- In linear search, the array is traversed sequentially from the first element until the value is found or the end of the array is reached. This technique is suitable for performing a search in a small array or in an unsorted array.
- The binary search technique is used to search a particular element in a sorted (in ascending or descending order) array. In this technique, the element to be searched (say, item) is compared with the middle element of the array.
- Interpolation search, also known as extrapolation search, is a searching technique that finds a specified value in a sorted array.

12.7 Key Terms

- **Hash Table:** It is merely an array of fixed size and elements in it are inserted and searched using a function called a hash function.
- **Interpolation Search:** The searching technique that finds a specified value in a sorted array.
- **Hashing:** The type of data structure used to map a given value with a particular key for faster access of elements.
- **Jump Search:** The searching technique in which we can jump to the desired element and there is no need to scan all the elements in the array.

12.8 Check Your Progress

Short- Answer type

Q1) Hashing uses a data structure called ________ which is merely an array of fixed size.

Q2) For a binary search, the array should be already sorted. True/False?

Q3) Which of these searching techniques is suitable for unsorted arrays?

(a) Binary search (b) Linear search (c) None of these (d) Any of these

Q4) Interpolation search is also known as _________ search.

Q5) A linear search is more efficient than a binary search. True/False?

Long- Answer type

Q1) What is searching? Name the various searching techniques.

Q2) Give the comparison for various search algorithms with the help of an example.

Q3) Write a short note on the linear search algorithm.

Q4) Write an algorithm for binary search.

Q5) Write a short note on:

(a) Interpolation Search (b) Jump Search

References

- *Data Structures using C,* Reema Thareja, Oxford University Press, 2nd Edition.
- *Classic Data Structures,* Debasis Samanta, PHI Learning Pvt. Ltd. 2nd Edition.

Unit- 13 Sorting

Structure

13.0 Introduction
13.1 Unit Objectives
13.2 Sorting
13.3 Internal Sorting
    13.3.1 Insertion Sort
    13.3.2 Bubble Sort
    13.3.3 Selection Sort
    13.3.4 Heap Sort
    13.3.5 Merge Sort
    13.3.6 Quick Sort
    13.3.7 Shell Sort
13.4 Comparison of different sorting algorithms
13.5 External Sorting
13.6 Summary
13.7 Key Terms
13.8 Check Your Progress

13.0 Introduction

Searching and sorting are the two most useful operations that are to be performed on a list, which is maintained either in an array or in the linked list. We have already studied various searching algorithms. This unit focuses on Sorting techniques and algorithms.

Various algorithms are available to sort a given list. All the sort algorithms take a list as input and produce a sorted list as output. Some algorithms can be applied on both arrays and linked lists, while some can be applied only on arrays. The simple sort algorithms are bubble, selection, and insertion, and none of them requires any additional data structure to sort a given list. On the other hand, sort algorithms such as merge, quick, and heap use additional data structures. The merge and quicksort algorithms use a stack, whereas heap sort makes use of the heap data structure. The heap data structure can be viewed as a complete binary tree in which the value in the

parent node is greater than the value in each of its child nodes. In general, the bubble, selection, and insertion sort algorithms are slower than the merge, quick, and heap sort algorithms.

13.1 Unit Objectives

After going through this unit, the reader will be able to:

- Explain the basics and need for Sorting.
- Describe the fundamentals of internal sorting.
- Discuss the insertion, bubble, selection, merge, quick, and heap sort techniques.
- Analyze the various sorting algorithms in terms of their time complexities.

13.2 Sorting

The process of arranging data in a particular logical order is known as sorting. The order can be ascending or descending for numeric data and alphabetical for character data. There are two types of sorting, namely internal sorting, and external sorting. If all the data that are to be sorted fit entirely in the main memory, then internal (in-memory) sorting is used. On the other hand, if all the data that are to be sorted do not fit entirely in the main memory, external sorting is required. An external sort requires the use of external memory, such as disks or tapes, during sorting. In external sorting, some part of the data is loaded into the main memory, sorted using any internal sorting technique, and written back to the disk in some intermediate file. This process continues until all the data are sorted. This section covers only some of the internal sorting algorithms. It also gives a brief comparison of various algorithms in terms of their time complexities.

13.3 Internal Sorting

There are different internal sorting algorithms, such as insertion sort, bubble sort, selection sort, heap sort, merge sort, and quicksort. The choice of a particular algorithm depends on the properties of the data and the operations to be performed on the data. For all these algorithms, we will consider an array ARR containing n elements, which are to be sorted in ascending order.

13.3.1 Insertion Sort

The insertion sort algorithm selects each element and inserts it at its proper position in the earlier sorted sublist. In the first pass, the element ARR[1] is compared with ARR[0], and if ARR[1] and ARR[0]are not sorted, they are swapped. In the second pass, the element ARR[2] is compared with ARR[0]and ARR[1], and it is inserted at its proper position in the sorted sublist containing the elements ARR[0]and ARR[1]. Similarly, during i^{th} iteration, the element ARR[i] is placed at its proper position in the sorted sublist containing the elements ARR[0], ARR[1], ARR[1],..., ARR[i–1].

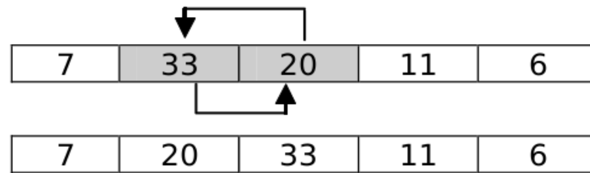
In order to determine the actual position of the element (say, ARR[i]) in the sorted sublist containing the elements ARR[0], ARR[1], ..., ARR[i-1], the element ARR[i] is compared with all other elements to its left, until an element ARR[j] is found such that ARR[j]<=ARR[i]. Now, to insert the element at its actual position, all the elements ARR[i-1], ARR[i-2], ARR[i-3],..., ARR[j+1] are shifted one position towards the right to create space for ARR[i], and then ARR[i] is inserted at (j+1) st position.

To understand the insertion sort algorithm, consider the following unsorted array. The steps to sort the values stored in the array in ascending order using insertion sort are given here.

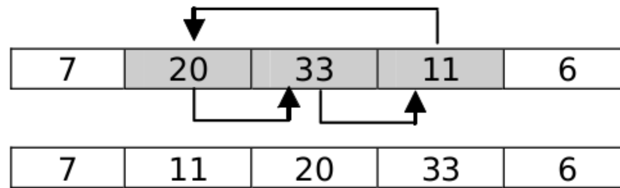
7	33	20	11	6

Unsorted Array

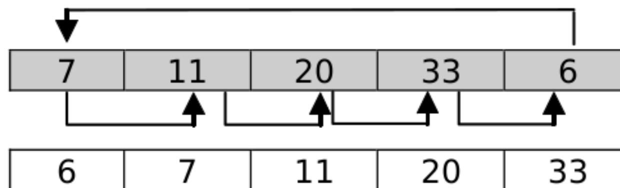
1. The first value, that is 7, is trivially sorted by itself.
2. Then the second value 33 is compared with the first value 7. Since 33 is greater than 7, no changes are made.
3. Next, the third element 20 is compared with its previous elements (elements towards its left). Since 20 is smaller than 33 but greater than 7, it is inserted at the second position. For this, element 33 is shifted one position towards the right, and 20 is inserted at its appropriate (second) position.



4. Then the fourth element 11 is compared with its previous elements. Since 11 is greater than 7 and less than 20 and 33, it is placed between 7 and 20. For this, elements 20 and 33 need to be shifted one position towards the right.



5. Finally, the last element 6 is compared with all the elements preceding it. Since it is smaller than all the other elements, preceding elements are shifted one position towards the right and 6 is inserted at the first position in the array. After this pass, the array is sorted.



Final Sorted Array

Algorithm 13.1 Insertion Sort

```
insertion_sort(ARR, size)
1. Set i = 1
2. While (i < size)
        Set temp = ARR[i]
        j = i - 1
        While (temp < ARR[j] AND j >= 0)
                Set ARR[j+1] = ARR[j]
                Set j = j - 1
   End While
   Set ARR[j+1] = temp
   Print ARR after i th pass
   Set i = i + 1
End While
3. Print "No. of passes: ", i-1
4. End
```

Program 13.1: Write a program to show sorting of an array using insertion sort.

```
#include<stdio.h>
#include<conio.h>
#define MAX 10
/*Function prototype*/
void insertion_sort(int [], int);
void main()
{
        int ARR[MAX], i, size;
        do
        {
                clrscr();
                printf("\nEnter the size of the array (max %d): ",MAX);
                scanf("%d", &size);
                }while(size>MAX);
        printf("\nEnter the elements of the array:\n");
        for(i=0;i<size;i++)
                scanf("%d", &ARR[i]);
        insertion_sort(ARR, size);
        printf("\nThe sorted array is: ");
```

```
        for(i=0;i<size;i++)
        {
                printf("%d ", ARR[i]);
        }
        getch();
}
void insertion_sort(int ARR[], int size)
{
int i, j, k, temp, count=0;;
for (i=1;i<size;i++)
{
temp=ARR[i];
j=i-1;
if(temp<ARR[j])
{
while(temp<ARR[j] && j>=0)
{
ARR[j+1]=ARR[j];
j—;
}
}
ARR[j+1]=temp;
printf("\nArray after pass %d: ", i);
for(k=0;k<size;k++)
{
printf("%d ", ARR[k]);
}
}
printf("\nNo. of passes: %d", i-1);
}
```

The output of the program is:

Enter the size of the array (max 10): 5

Enter the elements of the array:

35

20

4

10

5

Array after pass 1: 20 35 4 10 5

Array after pass 2: 4 20 35 10 5

Array after pass 3: 4 10 20 35 5

Array after pass 4: 4 5 10 20 35

No. of passes: 4

The sorted array is: 4 5 10 20 35

Analysis of insertion sort

In the worst case, when the input list is in descending order, the first pass of insertion sort requires one comparison, second pass requires two comparisons,..., i^{th} pass requires i comparisons, and the last pass requires (n-1) comparisons. Therefore, complexity of insertion sort algorithm is:

$f(n) = 1 + 2 + 3 + \dots + (n-i) + \dots + (n-3) + (n-2) + (n-1)$

$= n(n-1)/2$

$= (n^2 - n)/2$

Since for all n, $(n^2 - n)/2$ is always less than n^2, the time complexity of insertion sort algorithm is $O(n^2)$.

13.3.2 Bubble Sort

The bubble sort algorithm requires n-1 passes to sort an array. In the first pass, each element (except the last) in the list is compared with the element next to it, and if it is greater, then both the elements are swapped. After the first pass, the largest element in the list is placed at the last position. Similarly, in the second pass, the second largest element is placed at its appropriate position. Thus, in each subsequent pass, one more element is placed at its appropriate position. Since this algorithm makes the larger values to 'bubble up' to the end of the list, it is named bubble sort.

The bubble sort algorithm possesses an important property that if a particular pass is made through the list without swapping any items, then there will be no further

swapping of elements in the subsequent passes. This property can be used to eliminate the unnecessary passes once the list is sorted in the desired order. For this, a flag variable can be used to detect if any interchange has been made during the pass. We use flag=0 to indicate that no swaps have occurred in a particular pass, therefore, no further passes are required.

To understand the bubble sort technique, consider the following unsorted array.

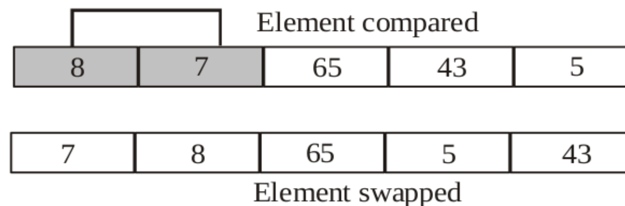
8	7	65	5	43

Unsorted Array

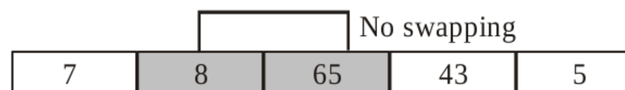
The steps to sort the values stored in the array in ascending order using bubble sort are given here.

First pass:

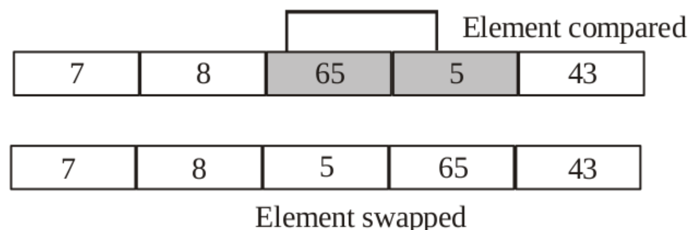
1. The values 8 and 7 are compared with each other. Since 7 is smaller than 8, both the values are swapped with each other.



2. Next, the values 8 and 65 are compared with each other. Since 8 is less than 65, that means they are in the proper order and hence, no swapping is required. The list remains unchanged.

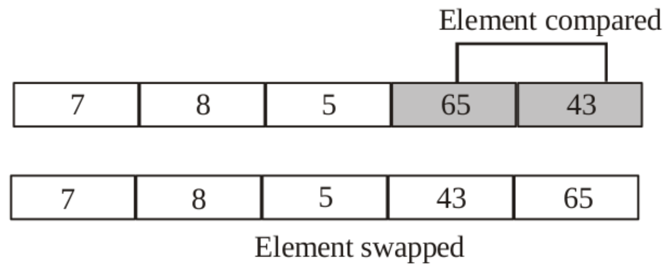


3. Then the values 65 and 5 are compared with each other. Since 5 is less than 65, both the values are swapped.



4. Next, the values 65 and 43 are compared with each other. Since 43 is less

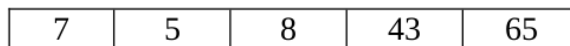
than 65, both the values are swapped.



After the first pass, the largest value of the array (here, 65) is placed at the last position.

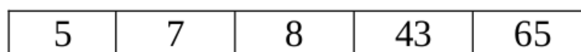
Second pass:

1. The values 7 and 8 are compared with each other. Since 7 is smaller than 8, no swapping is required.
2. Then the values 8 and 5 are compared. Since 8 is greater than 5, both are swapped.
3. Next, the elements 8 and 43, and 43 and 65 are compared. Since they are already in ascending order, they need not be swapped.



Third pass:

1. The values 7 and 5 are compared with each other. Since 7 is greater than 5, both are swapped.
2. Since the remaining elements are already in ascending order, they are not swapped.



Fourth pass:

In the fourth pass, no swapping is required as all the elements are already in ascending order. Thus, at the end of this pass, the list is sorted in ascending order.

Algorithm 13.2 Bubble Sort

```
bubble_sort(ARR, size)
1. Set i = 0, flag = 1
2. While (i < size-1 AND flag = 1)
        Set j = 0
        Set flag = 0
        While (j < size-i-1)
                If (ARR[j] > ARR[j+1])
                        Set flag = 1             //swap will occur, hence set flag = 1
                        Set temp = ARR[j]        //temp is temporary variable used to swap two values
                        Set ARR[j] = ARR[j+1]
                        Set ARR[j+1] = temp
                End If
                Set j = j + 1
        End While
    Print ARR after (i+1) th pass
    Set i = i + 1
    End While
3. Print "No. of passes: ", i
4. End
```

Program 13.2: Write a program to show sorting of an array using bubble sort.

```
#include<stdio.h>
#include<conio.h>
#define MAX 10
/*Function prototype*/
void bubble_sort(int [], int);
void main()
{
        int ARR[MAX],i, size;
        do
        {
                clrscr();
                printf("\nEnter the size of the array (max %d): ", MAX);
                scanf("%d", &size);
        }while(size>MAX);
        printf("\nEnter the elements of the array:\n");
        for(i=0;i<size;i++)
```

```
            scanf("%d", &ARR[i]);
        bubble_sort(ARR, size);
        printf("\nThe sorted array is: ");
        for(i=0;i<size;i++)
            printf("%d ", ARR[i]);
        getch();
}
void bubble_sort(int ARR[], int size)
{
        int i, j, k, temp, flag=1;
        i=0;
        while (i<size-1 && flag==1)
        {
            flag=0;
            for(j=0;j<size-i-1; j++)
            {
                if (ARR[j]>ARR[j+1])
                {
                    flag=1;
                    temp=ARR[j];
                    ARR[j]=ARR[j+1];
                    ARR[j+1]=temp;
                }
            }
            printf("\nArray after pass %d: ", i+1);
            for(k=0;k<size;k++)
            {
                printf("%d ", ARR[k]);
            }
            i++;
        }
        printf("\nNo. of passes: %d", i);
}
```

The output of the program is:

Enter the size of the array (max 10): 5
Enter the elements of the array:
8
7
65
5
43
Array after pass 1: 7 8 5 43 65
Array after pass 2: 7 5 8 43 65
Array after pass 3: 5 7 8 43 65
Array after pass 4: 5 7 8 43 65
No. of passes: 4
The sorted array is: 5 7 8 43 65

Analysis of bubble sort

To sort a list containing n elements, at most n-1 passes are required. The first pass requires n-1 comparisons, second pass requires n-2 comparisons, ..., i^{th} pass requires n-i comparisons. Therefore, average complexity of bubble sort algorithm is:

$$f(n) = (n-1) + (n-2) + (n-3) + \ldots + (n-i) + \ldots + 3 + 2 + 1$$
$$= n(n-1)/2$$
$$= (n^2 - n)/2$$
$$= O(n^2)$$

Note that under best-case conditions (when the list is almost or completely sorted), the bubble sort can approach the O(n) level of complexity. In other cases, the complexity level is $O(n^2)$.

13.3.3 Selection Sort

In selection sort, first, the smallest element in the list is searched and is swapped with the first element in the list (that is, it is placed at the first position). Then, the second smallest element is searched and swapped with the second element in the list (that is, it is placed at the second position), and so on.

Like the bubble sort algorithm, the selection sort also requires n-1 passes to sort an array containing n elements. However, there is a slight difference between the selection sort and bubble sort algorithm. In selection sort, the smallest element is the first one to be placed at its correct position, then the second smallest element comes at its position, and so on. In bubble
sort, on the other hand, the largest element is the first one to be placed at its appropriate position, then the second-largest element, and so on.
To understand the selection sort algorithm, consider the following unsorted array.

8	33	6	21	4

The steps to sort the values stored in the array in ascending order using selection sort are given here.

1. In the first pass, the entire array is scanned for the smallest element, which is 4 in this list. It is swapped with the first element, that is, 8. Thus, 4 is placed at its correct position and is not used for any further comparisons.

8	33	6	21	4

4	33	6	21	8

2. In the second pass, the smallest element is searched from the last four elements, which is 6. It is swapped with the second element, that is, 33.

4	33	6	21	8

4	6	33	21	8

3. In the third pass, the smallest element is searched from the last three elements, which is 8. This value is swapped with the third element, that is, 33.

4	6	33	21	8

4	6	8	21	33

4. In the fourth pass, the smallest element is searched from the last two elements. Since 21 is smaller than 33, no changes are made in the list obtained after the third pass, and the list is sorted in ascending order. The

sorted list is shown below.

4	6	8	21	33

Algorithm 13.3 Selection Sort

1. Set i = 0
2. While (i < size-1)
    Set small = ARR[i]
    Set pos = i
    Set j = i + 1
    While (j < size) //searching the smallest element in unsorted list
        If (ARR[j]<small)
            Set small = ARR[j]
            Set pos = j
        End If
        Set j = j + 1
    End While
    Set ARR[pos] = ARR[i] //placing the smallest element at its correct position
    Set ARR[i] = small
3. Print ARR after (i+1)th pass
    Set i = i + 1
    End While
    Print “No. of passes: ”, i selection_sort(ARR, size)
4. End

Program 13.3: Write a program to show sorting of an array using selection sort.

```
#include<stdio.h>
#include<conio.h>
#define MAX 10
/*Function prototype*/
void selection_sort(int [], int);
void main()
{
        int ARR[MAX], i, size;
        do
        {
                clrscr();
```

```
        printf("\nEnter the size of the array (max %d): ", MAX);
        scanf("%d", &size);
    }while(size>MAX);
    printf("\nEnter the elements of the array:\n");
    for(i=0;i<size;i++)
        scanf("%d", &ARR[i]);
    selection_sort(ARR, size);
    printf("\nThe sorted array is: ");
    for(i=0;i<size;i++)
        printf("%d ", ARR[i]);
    getch();
}
void selection_sort(int ARR[], int size)
{
    int i, j, k, small, pos;
    for (i=0;i<(size-1);i++)
    {
        small=ARR[i];
        pos=i;
        for (j=i+1;j<size;j++)
        {
            if (ARR[j]<small)
            {
                small=ARR[j];
                pos=j;
        }
    }
    ARR[pos]=ARR[i];
    ARR[i]=small;
    printf("\nArray after pass %d: ", i+1);
    for(k=0;k<size;k++)
        printf("%d ", ARR[k]);
    }
    printf("\nNo. of passes: %d", i);
}
```

The output of the program is:

Enter the size of the array (max 10): 5

Enter the elements of the array:

8

6

33

21

5

Array after pass 1: 5 6 33 21 8

Array after pass 2: 5 6 33 21 8

Array after pass 3: 5 6 8 21 33

Array after pass 4: 5 6 8 21 33

No. of passes: 4

The sorted array is: 5 6 8 21 33

Analysis of selection sort

Selection sort also requires n-1 passes to sort an array of n elements. The first pass requires n-1 comparisons, second pass requires n-2 comparisons,..., i^{th} pass requires n-i comparisons and the last pass requires only one comparison. Therefore, average complexity of selection sort algorithm is:

$$f(n) = (n-1) + (n-2) + (n-3) + \ldots + (n-i) + \ldots + 3 + 2 + 1$$
$$= n(n-1)/2$$
$$= (n^2 - n)/2$$
$$= O(n^2)$$

13.3.4 Heap Sort

Heapsort is a more efficient version of the selection sort. Like selection sort, it also first determines the largest (or smallest) element of the list, places it at the end (or beginning) of the list, and then continues with the rest of the list. However, it accomplishes this task efficiently by using a different data structure called a *heap*, which can be visualized as a complete binary tree. Recall that a complete binary tree is completely filled, with the possible exception of the last level which is filled from left to right (Figure 13.1).

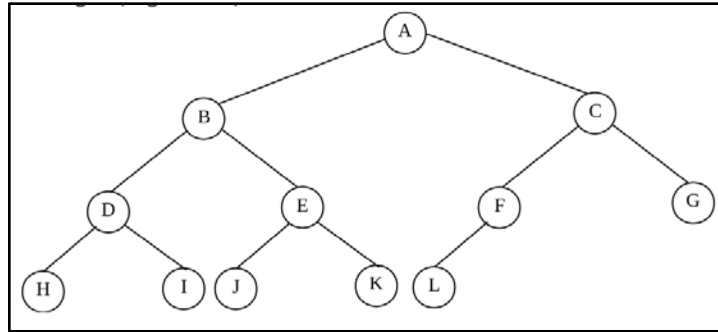
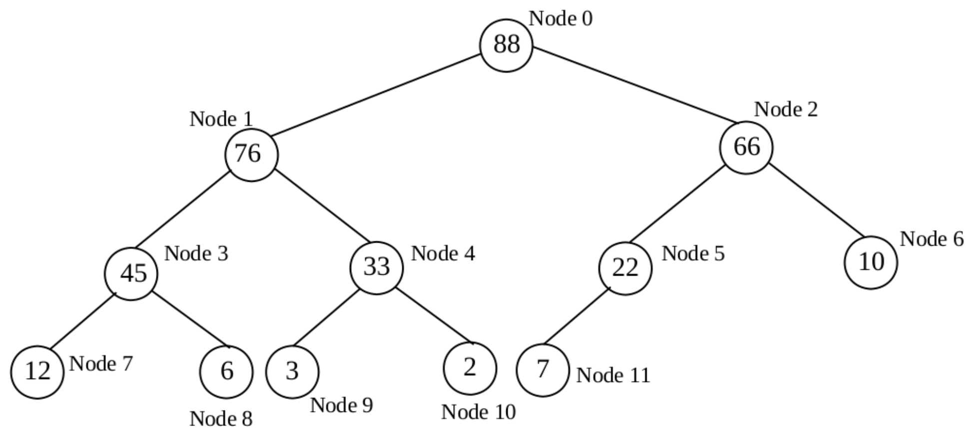
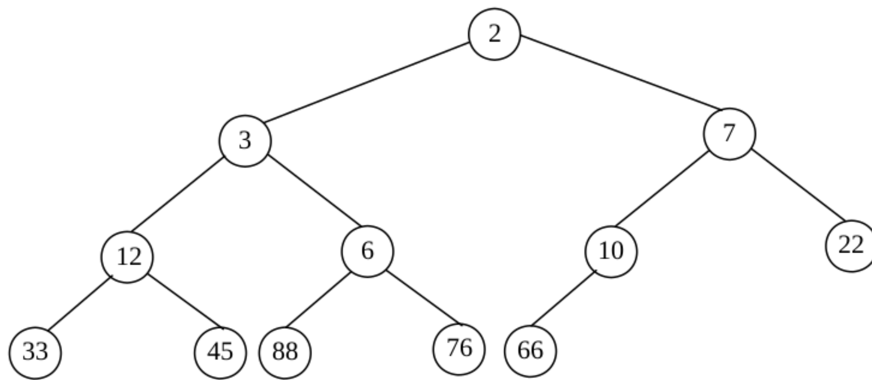


Figure 13.1 A complete binary tree

Heaps can be of two types, namely max-heap and min-heap. A max-heap (or descending heap) is a kind of heap in which the value present at any node is greater than or equal to the value of each of its child nodes. On the other hand, a min-heap (or ascending heap) is a kind of heap in which the value present at any node is smaller than or equal to the value of each of its child nodes. The max-heap and min-heap with 12 nodes are shown in figure 13.2. Note that the values in the child nodes of a node may not be in order, that is, sometimes the value in the right child may be more than that in the left child and at some other times, it may be less than the value in the left child.



(a) Max- Heap



(b) Min- Heap

Figure 13.2 Types of Heap

A complete binary tree can be stored most efficiently as a single-dimensional array in which the root node is stored at 0th position and its left and right child nodes are stored at 1st and 2nd position respectively. For each ith node, the left and right child exist at (2i+1)th and (2i+2)th position respectively. The parent node of ith node is stored at [(i-1)/2]th node. For example, in Figure 13.2(a) the left and right child nodes of 4th node are stored at 9th (2*4+1) and 10th (2*4+2) positions respectively. The parent node of the 4th node is stored at 1st ((4-1)/2) position.

The array representation of the heap shown in Figure 13.2(a) is shown below.

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]	A[11]
88	76	66	45	33	22	10	12	6	3	2	7

To sort an array of size n in ascending order using heap sort, the following steps are performed:

1. The initial max-heap is built from the given array.
2. The root element is swapped with the last element in the array.
3. The heap of remaining elements is restored.
4. Steps 2 and 3 are repeated until there are no more elements.

To understand the heap sort, consider an unsorted array ARR.

9	11	6	45	22	10	12	90	67	17

Let us first discuss the steps to build a heap out of the given array. The elements in the given array are considered one by one. If an element ARR[i] is greater than its

parent which is stored at location (i-1)/2, then the element is swapped with its parent. The element is then compared with its new parent and a swap occurs if it is greater than its parent. This process continues until no more swapping is needed, or we are at the root node. For example, the steps to construct a heap out of the array

ARR shown above are as follows:

1. The first element, that is, 9, is stored at position ARR[0].
2. The second element 11 is compared with its parent node which is at the location (i-1)/2, that is, (1-1)/2=0. Since ARR[0]<ARR[1], they are swapped. Now ARR is as follows:

11	9	6	45	22	10	12	90	67	17

3. The third element 6 is compared with its parent node which is at the location (2-1)/2=0. Since ARR[0]>ARR[2], they are not swapped. ARR remains the same.
4. The fourth element 45 is compared with its parent node which is at the location (3-1)/2=1. Since ARR[1]<ARR[3], they are swapped. It is again compared with its parent node which is at the location (1–1)/2=0. Since ARR[0]>ARR[1], they are swapped. Now ARR is as follows:

45	11	6	9	22	10	12	90	67	17

5. The next element 22 is compared with its parent node which is at the location (4-1)/2=1. Since ARR[1]<ARR[4], they are swapped. It is again compared with its parent node which is at the location (1-1)/2=0. Since ARR[0]>ARR[1], they are not swapped. ARR at this point is as follows:

45	22	6	9	11	10	12	90	67	17

6. The next element 10 is compared with its parent node which is at the location (5-1)/2=2. Since ARR[2]<ARR[5], they are swapped. It is again compared with its parent node which is at the location (2-1)/2=0. Since ARR[0]>ARR[1], they are not swapped. ARR at this point is as follows:

45	22	10	9	11	6	12	90	67	17

7. The seventh element 12 is compared with its parent node which is at the location (6-1)/2=2. Since ARR[2]<ARR[6], they are swapped. It is again compared with its parent node which is at the location (2-1)/2=0. Since ARR[0]>ARR[1], they are not swapped. ARR at this point is as follows:

45	22	12	9	11	6	10	90	67	17

8. The next element 90 is compared with its parent node which is at the location (7-1)/2=3. Since ARR[3]<ARR[7], they are swapped. It is again compared with its parent node which is at the location (3-1)/2=1. Since ARR[1]<ARR[3], they are swapped. It is again compared with its parent node which is at the location (1-1)/2=0. Since ARR[0]<ARR[1], they are swapped. ARR at this point is as follows:

90	45	12	22	11	6	10	9	67	17

9. The next element 67 is compared with its parent node which is at the location (8-1)/2=3. Since ARR[3]<ARR[8], they are swapped. It is again compared with its parent node which is at the location (3-1)/2=1. Since ARR[1]<ARR[3], they are swapped. It is again compared with its parent node which is at the location (1-1)/2=0. Since ARR[0]>ARR[1], they are not swapped. ARR at this point is as follows:

90	67	12	45	11	6	10	9	22	17

10. The last element 17 is compared with its parent node which is at the location (9-1)/2=4. Since ARR[4]<ARR[9], they are swapped. It is again compared with its parent node which is at the location (4-1)/2=1. Since ARR[1]>ARR[4], they are not swapped. ARR at this point is as follows:

90	67	12	45	17	6	10	9	22	11

The initial max-heap for the array ARR is shown in figure 13.3.

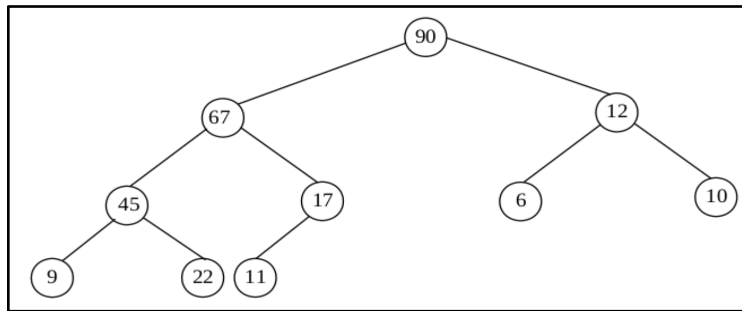


Figure 13.3 Initial Max-heap for ARR

Once the max-heap is created, the root node is guaranteed to contain the largest element of the list. This element is swapped with the last element in the list. It means that the largest element of the list is placed at its proper position. Now, the maximum index value of the array is reduced by one. The array at this point may not be satisfying the properties of the heap. Therefore, the heap needs to be restored with the remaining elements.

During the restoration, the element at the root node is compared with its child node(s), and if it is smaller than its child nodes, then it is swapped with the greatest of the two child nodes. Now, this element is compared with its current child nodes, and again it is swapped with the greatest of the two-child nodes if it is smaller than its child nodes. This process is repeated until this element is placed at its proper position. At this time, the second-largest element is placed at the root node. Now, this element is swapped with the second-last element in the list. It means that the second largest element is placed at its proper position. The maximum index value of the array is again reduced by one and the process continues until no more elements remain in the heap.

For example, to sort the given array using the heap sort, first, the root element (which is 90) is swapped with the last element (which is 11). This moves the largest element to the end of the list. Now the heap is restored with the remaining elements. Since the element at the root node, that is, 11 is smaller than its child nodes, it is swapped with the greatest of the two child nodes. Here, it is swapped with 67. Since element 11 is still smaller than its two child nodes, it is again swapped with the largest of the two child nodes, which is 45. Finally, element 11 is swapped with 22. The heap after this point is shown in Figure 13.4.

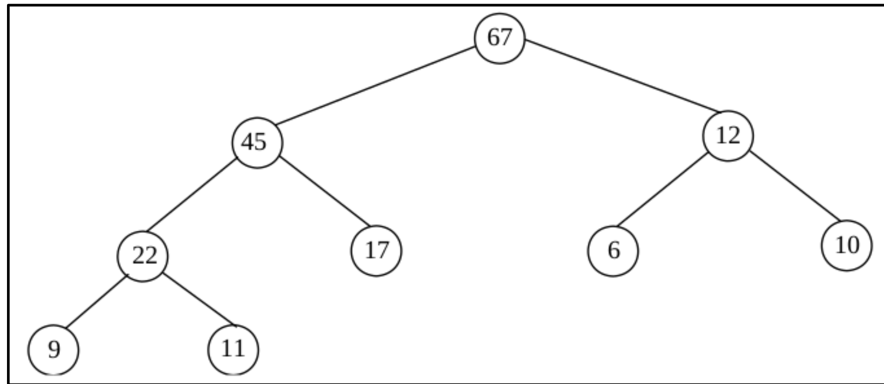


Figure 13.4 Heap after Eliminating 90

At this point, the second-largest element (that is, 67) is at the root node. Again 67 is swapped with the second-last element in the array (that is, 11). The heap is again restored with the remaining elements. This process is repeated until the array is sorted.

Algorithm 13.4 Heap Sort

heap_sort(ARR, size)
1. Set i = size-1, j = 1
2. Call make_heap(ARR, size)
3. Print the initial heap
4. While (i > 0)
        Set temp = ARR[i]
        Set ARR[i] = ARR[0]
        Set ARR[0] = temp
    Call restore(ARR, i) //calling restore function to restore the heap with remaining elements
    Print Heap after j th pass
    Set j = j + 1
    Set i = i - 1
  End While
5. Print "No. of passes: ", j-1
6. End

make_heap(ARR, size) //make_heap builds the initial heap of array ARR
1. Set i = 1
2. While (i < size)
        Set child = ARR[i]
        Set k = i

```
        Set parent = (k-1) / 2;
        While (k>0 AND ARR[parent]<child)
            Set ARR[k] = ARR[parent];
            Set k = parent;
            Set parent = (k-1) / 2;
        End While
        Set ARR[k] = child;
        Set i = i + 1
    End While
3. End
```

```
restore(ARR, size)              //restoring the heap with remaining elements
1. Set i = 0
2. Do
        Set lchild = (2*i+1)
        Set rchild = (2*i+2)
        If (rchild >= size)
              If (lchild < size AND ARR[i] < ARR[lchild])
                  Set temp = ARR[i]
                  Set ARR[i] = ARR[lchild]
                  Set ARR[lchild] = temp
           End If
           go to step 3
    Else If (ARR[i] < ARR[lchild] OR ARR[i] < ARR[rchild])
           If (ARR[lchild] > ARR[rchild])
                  Set temp = ARR[i]
                  Set ARR[i] = ARR[lchild]
                  Set ARR[lchild] = temp
                  Set i = lchild
           Else
                  Set temp = ARR[i]
                  Set ARR[i] = ARR[rchild]
                  Set ARR[rchild] = temp
                  Set i = rchild
           End If
      Else
           go to step 3
      End If
    While(1)
3. End
```

Program 13.4: Write a program to show sorting of an array using heap sort.

```
#include<stdio.h>
#include<conio.h>
#define MAX 20
/*Function prototypes*/
void make_heap(int [], int);
void heap_sort(int [], int);
void restore(int [], int);
void main()
{
	int ARR[MAX], i, size;
	do
	{
		clrscr();
		printf("\nEnter the size of the array (max %d): ", MAX);
		scanf("%d", &size);
	}while(size>MAX);
	printf("\nEnter the elements of the array:\n");
	for(i=0;i<size;i++)
		scanf("%d", &ARR[i]);
	heap_sort(ARR, size);
	printf("\nThe sorted array is: ");
	for(i=0;i<size;i++)
		printf("%d ", ARR[i]);
	getch();
}
void make_heap(int ARR[], int size)
{
	int i, k, parent, child;
	for(i=1;i<size;i++)
	{
		child=ARR[i];
		k=i;
		parent=(k-1)/2;
```

```
                while(k>0 && ARR[parent]<child)
                {
                        ARR[k]=ARR[parent];
                        k=parent;
                        parent=(k-1)/2;
                }
                ARR[k]=child;
        }
}
void restore(int ARR[], int size)
{
        int i=0, lchild, rchild, temp;
        do
        {
                lchild=(2*i+1);
                rchild=(2*i+2);
                if(rchild>=size)
                {
                        if(lchild<size && ARR[i]<ARR[lchild])
                        {
                                temp=ARR[i];
                                ARR[i]=ARR[lchild];
                                ARR[lchild]=temp;
                        }
                        break;
                }
                else if(ARR[i]<ARR[lchild] || ARR[i]<ARR[rchild])
                {
                        if(ARR[lchild]>ARR[rchild])
                        {
                                temp=ARR[i];
                                ARR[i]=ARR[lchild];
                                ARR[lchild]=temp;
                                i=lchild;
                        }
                        else
```

```
                {
                        temp=ARR[i];
                        ARR[i]=ARR[rchild];
                        ARR[rchild]=temp;
                        i=rchild;
                }
            }
            else
            break;
        }while(1);
}
void heap_sort(int ARR[], int size)
{
        int i, j=1, k, temp;
        make_heap(ARR, size);
        printf("\nInitial heap: ");
        for(i=0;i<size;i++)
            printf("%d ", ARR[i]);
        for(i=size-1;i>0;i—)
        {
            temp=ARR[i];
            ARR[i]=ARR[0];
            ARR[0]=temp;
            restore(ARR, i);/*rebuilding heap with remaining elements*/
            printf("\nHeap after %d pass: ",j);
            for(k=0;k<size;k++)
                printf("%d ", ARR[k]);
            j++;
        }
        printf("\nNo. of passes: %d", j-1);
}
```

The output of the program is:

Enter the size of the array (max 20): 10

Enter the elements of the array:

9

11

6

45

22

10

12

90

67

17

Initial heap: 90 67 12 45 17 6 10 9 22 11

Heap after 1 pass: 67 45 12 22 17 6 10 9 11 90

Heap after 2 pass: 45 22 12 11 17 6 10 9 67 90

Heap after 3 pass: 22 17 12 11 9 6 10 45 67 90

Heap after 4 pass: 17 11 12 10 9 6 22 45 67 90

Heap after 5 pass: 12 11 6 10 9 17 22 45 67 90

Heap after 6 pass: 11 10 6 9 12 17 22 45 67 90

Heap after 7 pass: 10 9 6 11 12 17 22 45 67 90

Heap after 8 pass: 9 6 10 11 12 17 22 45 67 90

Heap after 9 pass: 6 9 10 11 12 17 22 45 67 90

No of passes: 9

The sorted array is: 6 9 10 11 12 17 22 45 67 90

Analysis of heapsort

A complete binary tree with n nodes has a depth of log n. Therefore, building the initial heap of n elements requires n* log n comparisons, since inserting each element requires at most log n comparisons. After the creation of the initial heap, the element at the root node is swapped with the last element, and the heap is restored.

13.3.5 Merge Sort

Merge sort algorithm is based on the fact that it is easier and faster to sort two smaller arrays than one larger array. Therefore, it follows the principle of *divide-and-*

conquer. In this sorting, the list is first divided into two halves. The left and right sublists obtained are recursively divided into two sublists until each sublist contains not more than one element. The sublists containing only one element do not require any sorting. Therefore, we start merging the sublists of size one to obtain the sorted sub-list of size two. Similarly, the sublists of size two are then merged to obtain the sorted sub-list of size four. This process is repeated until we get the final sorted array.

To understand the merge sort algorithm, consider the following unsorted array. The steps to sort the values stored in the array in ascending order using merge sort are given here.

18	13	5	20	55	89	4	14

1. Initially, low=0 and high=7, therefore, mid=(0+7)/2=3. Thus, the given list is divided into two halves from the 4th element. The sub-lists are as follows:

18	13	5	20		55	89	4	14

2. The left sub-list is considered first, and it is again divided into two sublists. Now, low=0 and high=3, therefore, mid=(0+3)/2=1. Thus, the left sublist is divided into two halves from the 2nd element. The sub-lists are as follows:

18	13		5	20

3. These two sublists are again divided into sub-lists such that all of them contain one element.
4. Since each sub-list now contains one element, all sub-lists are first merged to produce the two arrays of size 2. First, the sublists containing the elements 18 and 13 are merged to give one sorted sub-array, and then sub-lists containing the elements 5 and 20 are merged to give another sorted sub-array. The two sorted subarrays are as follows:

13	18		5	20

5. Now, these two sub-arrays are again merged to give the following sorted subarray of size 4.

5	13	18	20

6. After sorting the left half of the array, we perform the same steps for the right half. The sorted right half of the array is given below:

4	14	55	89

7. Finally, the left and right halves of the array are merged to give the sorted array.

4	5	13	14	18	20	55	89

Algorithm 13.5 Merge Sort

```
merge_sort(ARR, low, high)
1. If (low < high)
      Set mid = (low + high) / 2
      Call merge_sort(ARR, low, mid)    //calling merge_sort recursively for left sub list
      Call merge_sort(ARR, mid+1, high)   //calling merge_sort for right sub list
      Call merging(ARR, low, mid, mid+1, high)
   End If
2. End
```

```
merging(ARR, ll, lr, ul, ur)
//merging() merges the two sub-arrays to produce a sorted array named merged ll and ul are
the lower bounds of the left and right sub-list respectively.
//ul and ur the upper bounds of the left and right sub-list respectively.
1. Set i = ll, j = ul, k = ll
2. While(i <= lr AND j <= ur)
        If(ARR[i] <= ARR[j])
              Set merged[k] = ARR[i]
              Set i = i + 1
      Else
         Set merged[k] = ARR[j]
         Set j = j + 1
      End If
      Set k= k + 1
    End While
    If(i <= lr)
      While(i <= lr)
         Set merged[k] = ARR[i]
         Set i = i + 1
         Set k = k + 1
```

```
        End While
    End If
    If(j <= ur)
        While(j <= ur)
            Set merged[k] = ARR[j]
            Set j = j + 1
            Set k = k+ 1
        End While
    End If
    Set k = ll
    While (k <= ur)
        Set ARR[k] = merged[k]
        Set k = k + 1
    End While
3. End
```

Program 13.5: Write a program to show sorting of an array using merge sort.

```
#include<stdio.h>
#include<conio.h>
#define MAX 20
/*Function prototypes*/
void merging(int [], int, int, int, int);
void merge_sort(int [], int, int);
void main()
{
        int ARR[MAX], i, size;
        do
        {
                clrscr();
                printf("\nEnter the size of the array (max %d): ", MAX);
                scanf("%d", &size);
        }while(size>MAX);
        printf("\nEnter the elements of the array:\n");
        for(i=0;i<size;i++)
                scanf("%d", &ARR[i]);
        merge_sort(ARR, 0, size-1);
        printf("\nThe sorted array is: ");
```

```
        for(i=0;i<size;i++)
                printf("%d ", ARR[i]);
        getch();
}
void merge_sort(int ARR[], int low, int high)
{
        int mid;
        if(low<high)
{
        mid=(low+high)/2;
        merge_sort(ARR, low, mid);      /*calling merge_sort recursively for left sub list*/
        merge_sort(ARR, mid+1, high); /*calling merge_sort recursively for right*/ /*sub list*/
        merging(ARR, low, mid, mid+1, high);
        }
}
void merging(int ARR[], int ll, int lr, int ul, int ur)
{
        int i, j, k, merged[MAX];
        i=ll;
        j=ul;
        k=ll;
        while(i<=lr && j<=ur)
        {
                if(ARR[i]<=ARR[j])
                {
                        merged[k]=ARR[i];
                        i++;
                }
                else
                {
                        merged[k]=ARR[j];
                        j++;
                }
                k++;
        }
        if(i<=lr)
        while(i<=lr)
        {
                merged[k]=ARR[i];
                i++;
```

```
                k++;
        }
if(j<=ur)
        while(j<=ur)
        {
                merged[k]=ARR[j];
                j++;
                k++;
        }
for(k=ll;k<=ur;k++)
        ARR[k]=merged[k];
}
```

The output of the program is:

Enter the size of the array (max 20): 10

Enter the elements of the array:

65

12

45

78

96

32

56

44

25

11

The sorted array is: 11 12 25 32 44 45 56 65 78 96

Analysis of merge sort

In the first pass of the merge sort algorithm, the given array is divided into two halves and each half is sorted separately. In each of the recursive calls to the merge_sort(), one for the left half and one for the right half, the array is further divided into two halves, thereby resulting in four segments of the array. Thus, in each pass, the number of segments of the array gets doubled until each segment

contains not more than one element. Therefore, the total number of divisions is log n. Moreover, in any pass, at most n comparisons are required. Hence, the complexity of the merge sort algorithm is O(n log n).

13.3.6 Quick Sort

Quicksort algorithm also follows the principle of *divide-and-conquer*. However, it does not simply divide the list into halves. Rather, it first picks up a partitioning element, called the *pivot*, that divides the list into two sublists such that all the elements in the left sub-list are smaller than the pivot, and all the elements in the right sublist are greater than the pivot. The same process is applied on the left and right sublists separately. This process is repeated recursively until each sublist contains not more than one element.

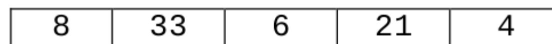
The main task in quicksort is to find the pivot that partitions the given list into two halves so that the pivot is placed at its appropriate location in the array. The choice of the pivot has a significant effect on the efficiency of the quick sort algorithm. The simplest way is to choose the first element as a pivot. However, the first element is not a good choice, especially if the given list is already or nearly ordered. For better efficiency, the middle element can be chosen as a pivot. Note that we will take the first element as a pivot for simplicity.

The steps involved in the quick sort algorithm are as follows:

1. Initially, three variables pivot, beg, and end are taken, such that both pivot and beg refer to the 0th position, and end refers to $(n-1)^{th}$ position in the list.
2. Starting with the element referred to by the end, the array is scanned from right to left, and each element on the way is compared with the element referred to by pivot. If the element referred to by pivot is greater than the element referred to by the end, both types of elements are swapped and step 3 is performed. Otherwise, the end is decremented by 1, and step 2 is continued.
3. Starting with the element referred to by beg, the array is scanned from left to right, and each element on the way is compared with the element referred to by pivot. If the element referred to by pivot is smaller than the element referred to by the end, both types of elements are swapped and step 2 is performed.

Otherwise, beg is incremented by 1, and step 3 is continued.

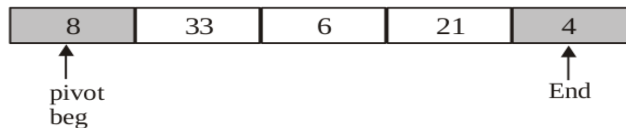
The first pass terminates when pivot, beg, and end all refer to the same array element. This indicates that the pivot element is placed at its final position. The elements to the left of this element are smaller than this element, and elements to its right are greater.
To understand the quick sort algorithm, consider the following unsorted array. The steps to sort the values stored in the array in ascending order using quick sort are given here.



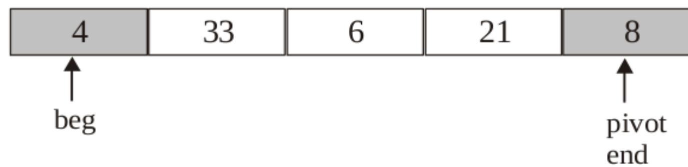
Unsorted Array

First pass:

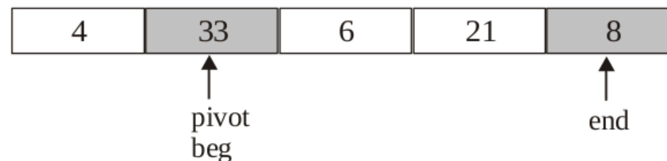
1. Initially, the index 0 in the list is chosen as the pivot, and the index variables beg and end are initialized with index 0 and n-1 respectively.



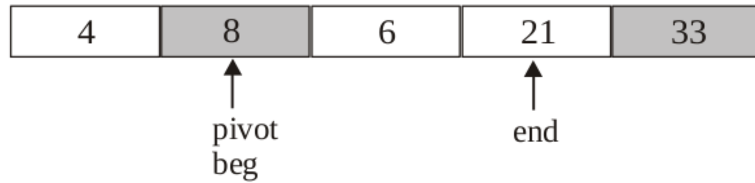
2. The scanning of elements is started from the end of the list. ARR[pivot] (that is, 8) is greater than ARR[end] (that is, 4). Therefore, they are swapped.



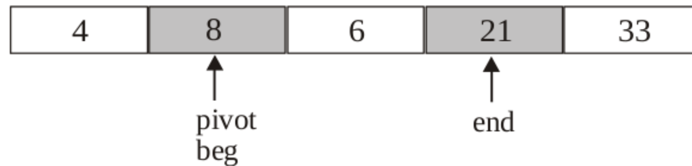
3. Now, the scanning of elements is started from the beginning of the list. Since ARR[pivot] (that is, 8) is greater than ARR[beg] (that is 33), therefore beg is incremented by 1, and the list remains unchanged.



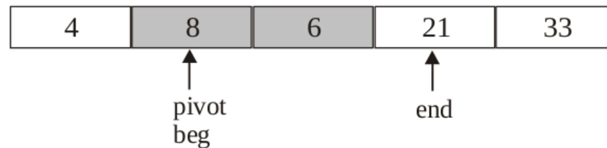
4. Next, the element ARR[pivot] is smaller than ARR[beg], they are swapped.



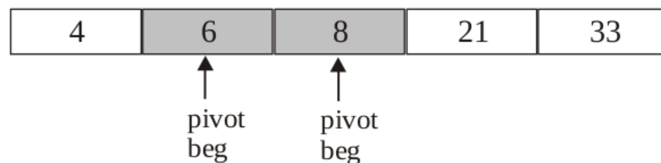
5. Again, the list is scanned from right to left. Since ARR[pivot] is smaller than ARR[end], therefore the value of end is decremented by 1, and the list remains unchanged.



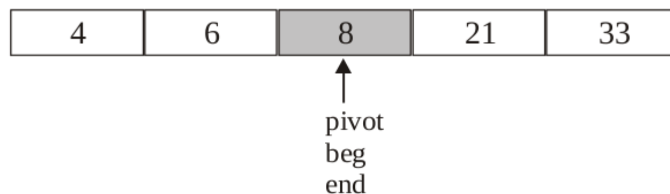
6. Next, the element ARR[pivot] is smaller than ARR[end], the value of the end is decremented by 1, and the list remains unchanged.



7. Now, ARR[pivot] is greater than ARR[end], they are swapped.



8. Now, the list is scanned from left to right. Since ARR[pivot] is greater than ARR[beg], the value of beg is incremented by 1, and the list remains unchanged.



At this point, since the variables pivot, beg, and end all refer to the same element, the first pass is terminated and the value 8 is placed at its appropriate position. The elements to its left are smaller than 8, and elements to its right are greater than 8.

The same process is applied to the left and right sublists.

Algorithm 13.6 Quick Sort

```
quick_sort(ARR, size, lb, ub)
1. Set i = 1                    //i is a static integer variable
2. If lb < ub
        Call splitarray(ARR, lb, ub)          //returning an integer value pivot
        Print ARR after ith pass
        Set i = i + 1
     Call quick_sort(ARR, size, lb, pivot – 1)    //recursive call to quick_sort() to sort left sub list
     Call quick_sort(ARR, size, pivot + 1, ub); //recursive call to quick_sort() to sort right sub list
   Else if (ub=size-1)
   Print "No. of passes: ", i
   End If
3. End
```

```
splitarray(ARR, lb, ub)  //split array partitions the list into two sublists such that the elements in the left sub list are smaller than ARR[pivot], and elements in the right sub-list are greater than ARR[pivot]
1. Set flag = 0, beg = pivot = lb, end = ub
2. While (flag != 1)
         While (ARR[pivot] <= ARR[end] AND pivot != end)
                 Set end = end – 1
        End While
        If pivot = end
               Set flag = 1
        Else
               Set temp = ARR[pivot]
               Set ARR[pivot] = ARR[end]
               Set ARR[end] = temp
               Set pivot = end
        End If
        If flag != 1
               While (ARR[pivot] >= ARR[beg] AND pivot != beg)
                       Set beg = beg + 1
               End While
               If pivot = beg
                       Set flag = 1
               Else
                       Set temp = ARR[pivot]
                       Set ARR[pivot] = ARR[beg]
```

```
                    Set ARR[beg] = temp
                    Set pivot = beg
               End If
          End If
     End While
3. Return pivot
4. End
```

Program 13.6: Write a program to show sorting of an array using quick sort.

```
#include<stdio.h>
#include<conio.h>
#define MAX 20
/*Function prototypes*/
void quick_sort(int [], int, int, int);
int splitarray(int [], int, int);
void main()
{
        int ARR[MAX], i, size;
        do
        {
                clrscr();
                printf("\nEnter the size of the array (max %d): ", MAX);
                scanf("%d", &size);
        }while(size>MAX);
        printf("\nEnter the elements of the array:\n");
        for(i=0;i<size;i++)
                scanf("%d", &ARR[i]);
        quick_sort(ARR, size, 0, size-1);
        printf("\nThe sorted array is: ");
        for(i=0;i<size;i++)
                printf("%d ", ARR[i]);
        getch();
}
void quick_sort(int ARR[], int size, int lb, int ub)
{
        int pivot, k;
```

```
        static int i=0;
        if (lb<ub)
        {
        pivot=splitarray(ARR, lb, ub);
        printf("\nArray after pass %d: ", i+1);
        for (k=0;k<size;k++)
                printf("%d ", ARR[k]);
        i++;
        quick_sort(ARR, size, lb, pivot-1);      //recursive call to function to sort left sub list
        quick_sort(ARR, size, pivot+1, ub);    //recursive call to function to sort right sub list
        }
        else if (ub==(size-1))
                printf("No. of passes: %d", i);
        }
int splitarray(int ARR[], int lb, int ub)
{
        int pivot, beg, end, temp, flag=0;
        beg=pivot=lb;
        end=ub;
        while(!flag)
        {
                while ((ARR[pivot]<=ARR[end]) && (pivot!=end))
                        end—;
                if (pivot==end)
                        flag=1;
                else
                {
                        temp=ARR[pivot];
                        ARR[pivot]=ARR[end];
                        ARR[end]=temp;
                        pivot=end;
                }
                if (!flag)
                {
                        while ((ARR[pivot]>=ARR[beg]) && (pivot!=beg))
                        beg++;
```

```
                if (pivot==beg)
                        flag=1;
                else
                {
                        temp=ARR[pivot];
                        ARR[pivot]=ARR[beg];
                        ARR[beg]=temp;
                        pivot=beg;
                }
            }
        }
        return pivot;
}
```

The output of the program is:

Enter the size of the array (max 20): 5

Enter the elements of the array:

6

5

4

3

2

Array after pass 1: 2 5 4 3 6

Array after pass 2: 2 5 4 3 6

Array after pass 3: 2 3 4 5 6

Array after pass 4: 2 3 4 5 6

No. of passes: 4

The sorted array is: 2 3 4 5 6

Analysis of quicksort

The quicksort algorithm gives the worst-case performance when the list is already sorted. In this case, the first element requires n comparisons to determine that it remains in the first position; the second element requires n-1 comparisons to

determine that it remains in the second position, and so on. Therefore, the total number of comparisons in this case is:

$f(n) = n + (n-1) + ... + 3 + 2 +1$

$= n(n+1)/2$

$= O(n^2)$

Note that in the worst case, the complexity of the quick sort algorithm is equal to that of the bubble sort algorithm. In the best case, when a pivot is chosen in such a way that it partitions the list approximately in half, then there will be log n partitions. Each pass does at most n comparisons. Therefore, the complexity of the quick sort algorithm in this case is:

$f(n) = n * \log n$

$= O(n \log n)$

13.3.7 Shell Sort

The shell sort algorithm was invented by Donald Shell in 1959. It is the most efficient sorting algorithm among all the algorithms with $O(n^2)$ complexity. Note that the shell sort algorithm does not actually sort the data itself; rather it increases the efficiency of other sorting algorithms. Usually, an insertion or bubble sort is used to arrange the data at each step, but other algorithms can be used. The algorithm performs several passes through the list and in each pass, the elements separated by a specific distance, say d, are arranged in order. Once all the elements with the current d are in order, the value of d is reduced by some factor and the process continues in the next pass.

Choosing the initial value of d is the most important task of the shell sort algorithm. Originally Donald Shell suggested size/2 as the initial value for d, where size is the number of elements in the array. However, d can be any number less than half the number of elements in the array. Further, in each subsequent pass, the value of d is reduced to half until it reaches 1.

To understand the shell sort algorithm, consider the following unsorted array with size=8.

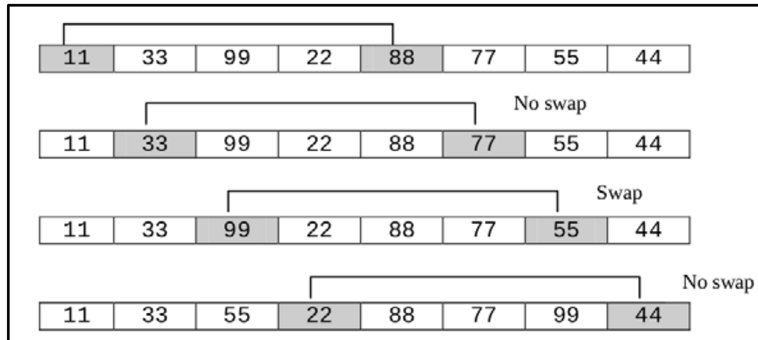
11	33	99	22	88	77	55	44

The steps to sort the values stored in the array in ascending order using shell sort

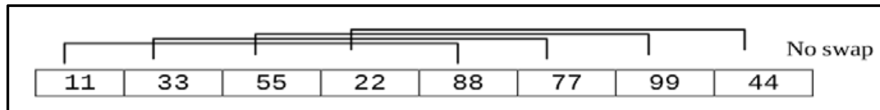
are given here.

First pass

The initial value of d=size/2, that is, 8/2=4. Therefore, the elements that are separated with distance 4 are arranged in order.



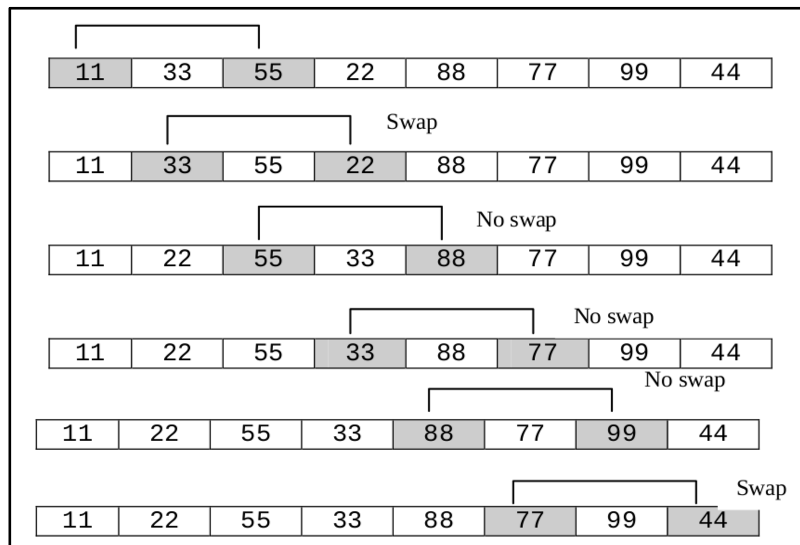
In this iteration, a swap has occurred. Thus, one more iteration is required with d=4.



This time, no swap occurs throughout the iteration. It means that the elements separated with distance 4 are in order. Hence, the first pass is completed.

Second pass

Now the value of d=d/2, that is, 4/2=2. Therefore, the elements that are separated with distance 2 are arranged in order.



In this iteration, two swaps have occurred. Thus, one more iteration is required with d=2.

No swap

11	22	55	33	88	44	99	77

This time, no swap occurs throughout the iteration. It means that the elements separated with distance 2 are in order. Hence, the second pass is completed.

Third pass

Now the value of d=d/2, that is, 2/2=1. Therefore, the elements that are separated with distance 1 are arranged in order.

No swap

11	22	55	33	88	44	99	77

No swap

11	22	55	33	88	44	99	77

Swap

11	22	55	33	88	44	99	77

No swap

11	22	33	55	88	44	99	77

Swap

11	22	33	55	88	44	99	77

No swap

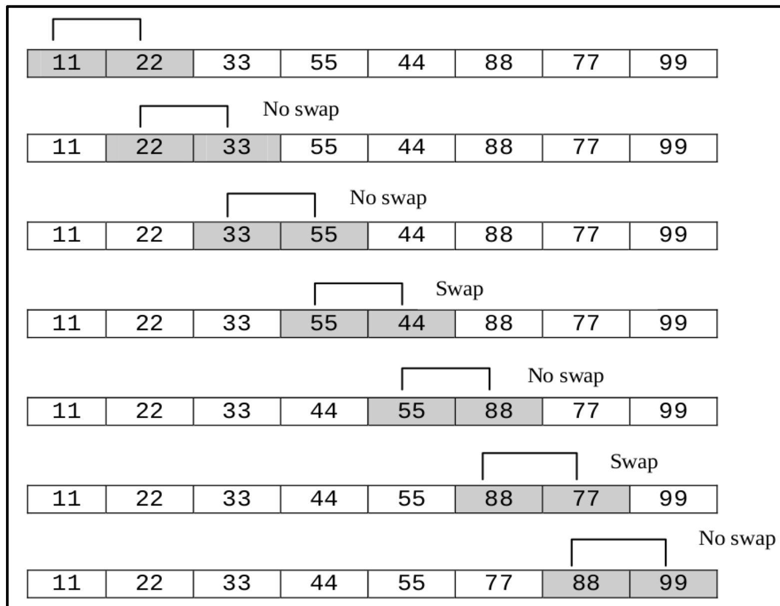
11	22	33	55	44	88	99	77

Swap

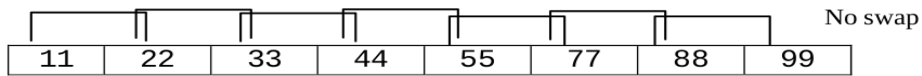
11	22	33	55	44	88	99	77

11	22	33	55	44	88	77	99

In this iteration, three swaps have occurred. Thus, one more iteration is required with d=1.



In this iteration, a swap has occurred. Thus, one more iteration is required with d=1.



Since no swap has occurred in this iteration. Therefore, the algorithm terminates after this pass.

Algorithm 13.7 Shell Sort

```
shell_sort(ARR, size)
1. Set count = 0
2. Set d = size / 2
3. Do
        Do
            Set swap = 0, i = 0
            While (i < size-d)
            If(ARR[i] >ARR[i+d])
            Set temp = ARR[i];
            Set ARR[i] = ARR[i+d];
            Set ARR[i+d] = temp;
            Set swap = 1
       End If
          Set i = i + 1
     End While
     While (swap)
     Set count = count + 1
     Print ARR after count th pass
     While (d = d / 2)
4. Print "No. of passes: ", count
5. End
```

Program 13.7: Write a program to show sorting of an array using shell sort.

```
#include<stdio.h>
#include<conio.h>
#define MAX 20
/*Function prototype*/
void shell_sort(int [], int);
void main()
{
        int ARR[MAX],i, size;
        do
        {
                clrscr();
                printf("\nEnter the size of the array: ");
                scanf("%d", &size);
                if (size>MAX)
```

```
                {
                        printf("\nInput size is greater than the maximum size.");
                getch();
        }
}while(size>MAX);
printf("\nEnter the elements of the array:\n");
for(i=0;i<size;i++)
        scanf("%d", &ARR[i]);
shell_sort(ARR, size);
printf("\nThe sorted array is: ");
for(i=0;i<size;i++)
        printf("%d ", ARR[i]);
        getch();
}
void shell_sort(int ARR[], int size)
{
        int swap, temp, i, count=0, k;
        int d=size/2;
        do
        {
                do
                {
                        swap=0;
                        for(i=0;i<size-d;i++)
                        {
                                if(ARR[i]>ARR[i+d])
                                {
                                        temp=ARR[i];
                                        ARR[i]=ARR[i+d];
                                        ARR[i+d]=temp;
                                        swap=1;/*swap=1 /*indicates that further execution with
                                current value of d is required*/
                                }
                        }
                }while(swap);
                count++;
```

```
                printf("\nArray after pass %d: ", count);
                for(k=0;k<size;k++)
                        printf("%d ", ARR[k]);
        }while(d=d/2);
        printf("\nNo. of passes: %d", count);
}
```

The output of the program is:

Enter the size of the array: 8

Enter the elements of the array:

11

33

99

22

88

77

55

44

Array after pass 1: 11 33 55 22 88 77 99 44

Array after pass 2: 11 22 55 33 88 44 99 77

Array after pass 3: 11 22 33 44 55 77 88 99

No. of passes: 3

The sorted array is: 11 22 33 44 55 77 88 99

Analysis of shell sort

It is very difficult to analyze the shell sort algorithm. This is because it is almost impossible to show the effect of one pass on subsequent passes. However, one thing is clear that if distance d is reduced each time to its half, then a total of log d passes are required since d=1 will complete the algorithm. In each pass, either bubble or insertion sort is used, each with the complexity of $O(n^2)$. Since shell sort moves the values with giant steps towards their final position in initial passes, it may require less number of iterations within each pass. However, the worst-case complexity of shell sort is $O(n^2)$.

13.4 Comparison of Various Sorting Algorithms

We have discussed several algorithms that can be used to sort a given set of elements. However, in order to choose an appropriate algorithm that suits a particular problem, the analysis of algorithms is necessary. For this, a number of efficiency parameters are considered. The two most important efficiency parameters are the time required to execute the algorithm and the amount of memory space it requires.

Now we will discuss the efficiency of an algorithm in terms of the amount of time required in its execution. The estimated amount of time required in executing an algorithm is referred to as the *time complexity* (or time efficiency) of the algorithm.

In sorting algorithms, several operations are performed, such as comparison operation (that compares two values to determine which is smaller or larger), interchange (swap) operation, an increment of an index variable in a loop, etc. However, comparison and interchange operations are the most significant operations as they require much more time than any other simple operation. Moreover, the number of interchanges cannot be greater than the number of comparisons. Therefore, we consider the number of comparisons as a useful measure of a sort's time efficiency.

In determining the time complexity of an algorithm, the size of an instance (input) also plays an important role. If the size of an instance is n, then the time complexity of the algorithm is some function of n. Thus, we need to determine a function f(n) that relates the number of operations to be performed to the size of the input. While comparing any two sorting algorithms, the algorithm whose function grows slower than the other is considered to be better. In mathematical terms, this relation is represented in *Big Oh* notation. In this notation, a function f(n) is O[g(n)], if there exist positive integers k and c such that $f(n) \le c*g(n)$, for all $n \ge k$ [where, f(n) and g(n) are the functions of two different algorithms]. The expression O is also called *Landau's symbol.*

Using the concept of *Big Oh* notation, we can thus compare various sorting algorithms and classify them as good or bad in general terms. If the algorithm has O(n) complexity, it implies that it has linear complexity and it grows linearly with the size of the input. For example, consider an algorithm that takes t time units to sort an array of 10 elements. In this case, the algorithm will take 10*t time units for 100 elements

(10*10=100). Unfortunately, no such sorting algorithm exists. Generally, the complexities of most of the sorting algorithms range between $O(n^2)$ and O(n log n). $O(n^2)$ is known as the *polynomial complexity* and O(n log n) is known as the *logarithmic complexity.* The insertion, bubble, selection, and shell sort algorithms have the complexity of $O(n^2)$, while heap, merge and quick sort algorithms have the complexity of O(n log n). If the size of the input is n, then O(n log n) is significantly faster than $O(n^2)$ as shown in Table 13.1.

Table 13.1 Comparing O(n log n) with $O(n^2)$

n	n^2	n log n
10	100	33
100	10000	665
1000	10^6	10^4
10^6	10^{12}	$2*10^7$
10^9	10^{18}	$3*10^{10}$

13.5 External Sorting

External sorting refers to the sorting of a file that is on a disk (or tape). Internal sorting refers to the sorting of an array of data that is in RAM. The main concern with external sorting is to minimize disk access since reading a disk block takes about a million times longer than accessing an item in RAM.

Perhaps the simplest form of external sorting is to use a fast internal sort with a good locality of reference (which means that it tends to reference nearby items, not widely scattered items) and hope that your operating system's virtual memory can handle it. (Quicksort is a one sort algorithm that is generally very fast and has a good locality of reference.) If the file is too huge, however, even virtual memory might be unable to fit it. Also, the performance may not be too great due to a large amount of time it takes to access data on disk.

Most external sort routines are based on mergesort. They typically break a large data file into a number of shorter, sorted "runs". These can be produced by repeatedly reading a section of the data file into RAM, sorting it with ordinary quicksort, and writing the sorted data to disk. After the sorted runs have been generated, a merge algorithm is used to combine sorted files into longer sorted files. The simplest scheme is to use a 2-way merge: merge 2 sorted files into one sorted file, then merge 2 more,

and so on until there is just one large sorted file. A better scheme is a multiway merge algorithm: it might merge perhaps 128 shorter runs together.

13.6 Summary

- The process of arranging data in some logical order is known as sorting.
- The insertion sort algorithm selects each element and inserts it at its proper position in the earlier sorted sub-list.
- The bubble sort algorithm requires n-1 passes to sort an array. In the first pass, the largest element in the list is placed at the last position. Similarly, after the second pass, the second largest element is placed at its appropriate position. Thus, in each subsequent pass, one more element is placed at its appropriate position.
- In selection sort, first, the smallest element in the list is searched and placed at the first position by swapping it with the first element. Then, the second smallest element is searched and placed at the second position, and so on.
- Heapsort uses a special data structure known as the heap, which is a complete binary tree.
- The merge sort algorithm is based on the fact that it is easier and faster to sort two smaller arrays than one larger array. Therefore, it follows the principle of divide-and-conquer.
- The quicksort algorithm also follows the principle of divide-and-conquer. It first picks up a partitioning element, called a pivot, that divides the list into two sublists such that all the elements in the left sub-list are smaller than the pivot, and all the elements in the right sub-list are greater than the pivot.
- The shell sort algorithm performs several passes through the list, and in each pass, the elements that are separated by a specific distance, say d, are arranged in order.

13.7 Key Terms

- **Max-heap (or descending heap):** A kind of heap in which the value present at any node is greater than or equal to the value of each of its child nodes.
- **Min-heap (or ascending heap):** A kind of heap in which the value present at

any node is smaller than or equal to the value of each of its child nodes.

- **Time complexity (or time efficiency) of the algorithm:** The estimated amount of time required in executing an algorithm.
- **Pivot:** It is a partitioning element, used in quicksort, that divides the list into two sublists such that all the elements in the left sub-list are smaller than the pivot, and all the elements in the right sublist are greater than the pivot.

13.8 Check Your Progress

Short- Answer type

Q1) Using the concept of ___________ notation, we can compare various sorting algorithms and classify them as good or bad in general terms.

Q2) Heap sort makes use of a data structure called a ___________.

Q3) Which of these is an internal sorting technique?

(a) Heap sort (b) Quick sort (c) Merge sort (d) All of these

Q4) Max-heap is also known as ascending heap. True/ False?

Q5) The quicksort algorithm is based on the principle of divide-and-conquer. True/ False?

Long- Answer type

Q1) What are the steps for sorting the values stored in an array in ascending order using selection sort?

Q2) Define Merge sort. Differentiate between Merge sort and Quicksort.

Q3) Write a brief analysis of the insertion sort.

Q4) Write a program to sort an array of strings using bubble sort.

Q5) Sort the following array in descending order using heap sort.

1, 3, 24, 17, 5, 32, 6, 99

References

- *Classic Data Structures,* Debasis Samanta, PHI Learning Pvt. Ltd. 2nd Edition.
- *Advanced Data Structures*, Peter Brass, Cambridge University Press, New York, 2008.
- *Data Structures and Algorithms,* Aho, Ullman and Hopcroft, Addison Wesley