PG CA S2 - 02 (DSC)

TWO YEAR MASTER'S DEGREE COURSE

(CBCS in Accordance with NEP 2020)

SEMESTER - II

DATA STRUCTURE THROUGH C LANGUAGE

MSc IT

Hiranya Chandra Bhuyan School of Science and Technology

KRISHNA KANTA HANDIQUI STATE OPEN UNIVERSITY

Subject Experts

Prof. Anjana Kakoti Mahanta, Gauhati University

Prof. Hiren Kumar Deva Sarma,Gauhati University

Dr. Lipi B Mahanta, IASST, Guwahati

Dr. Sanjib Kr. Kalita, Gauhati University

Course Co-ordinators

Dr.Tapashi Kashyap Das, KKHSOU

Dr. Ridip Dev Choudhury, KKHSOU

Dr. Sruti Sruba Bharali, KKHSOU

SLM Preparation Team

UNITS	CONTRIBUTORS
1, 7	Mr. Arabinda Saikia, KKHSOU
2, 8, 15	Dr. Naba Jyoti Sarma, Nalbari Commerce College
3, 5	Dr. Swapnali Gogoi, IDOL,Gauhati University
4, 6	Dr. Tapashi Kashyap Das, KKHSOU
9	Mr. Aniruddha Deka,Royal Global University
10, 11, 14	Mr. Manabendra Nath, NERIM
12	Ms. Daisy Das, IASST
13	Mr. Atowaslam, Cotton University

Editorial Team

Content : Dr. Sruti Sruba Bharali, KKHSOU

Language : Prof. (Retd.) Robin Goswami, Cotton College

Structure, Format & Graphics : Dr. Sruti Sruba Bharali, KKHSOU

April, 2024

This Self Learning Material (SLM) of the Krishna Kanta Handiqui State University is made available under a Creative Commons Attribution-Non Commercial-ShareAlike4.0 License (International): http.//creativecommons.org/licenses/by-nc-sa/4.0.

Printed and published by Registrar on behalf of the Krishna Kanta Handiqui State Open University.

Headquarters: Patgaon, Rani Gate, Guwahati-781017
City Office: Resham Nagar, Khanapara, Guwahati-781022; Web: www.kkhsou.in

The University acknowledges with thanks the financial support provided by the Distance Education Bureau, UGC for preparation of this material.

MASTER OF SCIENCE IN INFORMATION TECHNOLOGY

DETAILED SYLLABUS

		Pages
UNIT 1:	**Introductionto Data Structure** Data and Information; Data Structure and ItsTypes; Data Structure Operations; Different Data Types : Arraya, Pointers, Structures; Dynamic Memory Allocation : Abstract Data Types	**7-31**
UNIT 2:	**Algorithms** Algorithm; Complexity : Space Complexity, Time Complexity; Asymptotic Notation : Theta È Notation, Big O (O) Notation, Big Omega (Ù) Notation	**32-49**
UNIT 3:	**Linked List** Linked List ; Types of Linked List : Singly Linked List, Doubly Linked List, Circular Linked List ;Traversinga Single Linked List; Searchinga Single Linked List; Insertionintoa Single Linked List; Deletion froma Single Linked List; Reversing a Single Linked List	**50-74**
UNIT 4:	**Stack** Stack; Operationson Stack; Implementation of Stack: Implementation of Stack using Arrays, Implementation of Stackusing Linked Lists; Applications of Stack	**75-90**
UNIT 5:	**Queue** Queue; Array Implementation of Queue; Circular Queue; Linked List Implementation of Queue: Using Singly Linked List, Using Doubly Linked List, Using Circular Linked List; Priority Queues; Application of Queues	**91-135**
UNIT 6:	**Searching** Searching; Types of Searching: Linear Search, Binary Search	**136-150**
UNIT 7:	**Sorting** Sorting; Insertion Sort; Selection Sort; Bubble Sort; Quick Sort	**151-172**
UNIT 8:	**Trees** Tree; Binary Tree; Representation of Binary Tree: Sequential Representation of Binary Tree, Link Representation of Binary Tree; Tree Traversal Algorithms: Preorder Traversal, Inorder Traversal, Postorder Traversal; Prefix and Postfix Notations; Binary Search Tree	**173-201**

UNIT 9: Search Trees **202-220**

Search Trees; Balancing of Search Trees; AVL Trees: AVL Rotations; Multi-way Search Trees; 2-3 Trees; Splay Trees; Hashing

UNIT 10: Red Black Tree **221-239**

Red Black Tree: Properties of Red Black Tree, Height of a Red Black Tree; Balancing of Red Black Tree; Red Black Tree Rotations: Left Rotation, Right Rotation; Insertion in Red Black Tree; Deletion in Red Black Tree

UNIT 11: B-Trees **240-255**

B-Tree: Properties of B-Tree; Insertion in B-Tree; Deletion from B-Tree; B+ Tree: Insertion in B+ Tree; External Sorting

UNIT 12: Advance Sorting and Selection Techniques **256-269**

Heap Sort; Shell Sort; Sorting in Linear Time; Counting Sort; Radix Sort; Medians and Other Statistics; Lower Bound on Sorting

UNIT 13: Priority Queue **270-288**

Priority Queue; Priority Queue Using Array; Multi-queue Implementation; Linked List Representation of Priority Queue; Binary Heap; Binomial Heap; Fibonacci Heap

UNIT 14: Partition ADT **289-297**

Partition; Disjoint Set Operations: Linked List representation of Disjoint Sets; Union: Weighted Merge

UNIT 15: Graph **298-316**

Graph; Representation of Graphs: Adjacency Matrix Representation of Graph, Adjacency List Representation of Graph; Traversal of Graphs: Breadth First Traversal, Depth First Traversal

COURSE INTRODUCTION

Welcome to the course "Data Structures Through C Language.” In this course, we will explore the basics of organizing and working with data efficiently, using the C programming language.

In the initial units, we will start by understanding what data and information are, the types of data structures, and the different things we can do with them. We'll also look at the various ways data can be stored, like arrays and pointers, and how we can create and manage them. We will also learn about the problem - solving tools called algorithms and learn to measure how much space and time an algorithm needs. Next, we will explore linked lists, and see how we can make lists that go one way (singly linked lists), lists that go both ways (doubly linked lists), and lists that connect in a circle (circular linked lists). We'll also learn how to do things like finding, adding, and removing data from these lists. Data structures like stacks and queues are also included in this course and we will see how we can create and use these data structures using arrays and linked lists. Different searching and sorting algorithms are also covered in this course. In the latter part of the course, we will look in detail into different types of tree data structures. Trees are a way to organize data like branches on a tree. We'll see how they work, learn how to go through them in different ways, and also learn about binary search trees. We'll also learn about search trees and how to keep them balanced and explore different types like AVL trees and 2-3 trees. We will also look into special trees like Red-Black Trees and B-Trees see what makes them unique, how to keep them balanced, and how to add and remove data from them. We will also dive into more sorting methods like heap sort, shell sort, counting sort, radix sort, and ways to sort data quickly. Furthermore, we will introduce you to priority queues and various ways to create them, including using arrays and linked lists, and explore some advanced options like binary heaps and fibonacci heaps. We will also discuss on partitions, disjoint sets and how to combine them, giving more weight to certain groups. Lastly, we will finish our course by looking at graphs, which are like networks of data We'll see how to represent them and how to move through them using different methods.

Each unit includes some along-side boxes to help you know some of the difficult, unseen terms . Some "EXERCISES" have been included to help you apply your own thoughts. Again, we have included some relevant concepts in "LET US KNOW" along with the text. And, at the end of each section, you will get "CHECK YOUR PROGRESS” questions. These have been designed to self-check your progress of study. It will be better if you solve the problems put in these boxes immediately after you go through the sections of the units and then match your answers with "ANSWERS TO CHECK YOUR PPROGRESS" given at the end of each unit.

UNIT 1: INTRODUCTION TO DATA STRUCTURE

UNIT STRUCTURE

1.1 Learning Objectives

1.2 Introduction

1.3 Data and Information

1.4 Data Structure and Its Types

1.5 Data Structure Operations

1.6 Different Data Types

1.6.1 Arrays

1.6.2 Pointers

1.6.3 Structures

1.7 Dynamic Memory Allocation

1.8 Abstract Data Types

1.9 Let Us Sum Up

1.10 Further Reading

1.11 Answers to Check Your Progress

1.12 Model Questions

1.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- distinguish data and information
- define data structure and describe various types of data structures
- describe different data structure operations
- define abstract data types.

1.2 INTRODUCTION

Data structure in computer science is a way of storing and organizing data in a computer's memory or even disk storage so that it can be used efficiently. It is an organization of mathematical and logical concepts of data. A well-designed data structure allows a variety of critical operations to be performed, using as few resources, both execution time and memory

space, as possible. Data structures are implemented by a programming language by the data types and the references and operations provide by that particular language.

Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to certain tasks. For example, B-trees are particularly well-suited for implementation of databases. In the design of many types of computer program, the choice of data structures is a primary design consideration. Experience in building large systems has shown that the difficulty of implementation and the quality and performance of the final result depends heavily on choosing the best data structure.

In this unit, we will introduce you to the fundamental concepts of data structure. We shall also discuss in the unit arrays, pointers and structures besides different data structure operations and abstract data types. In the next unit we will discuss about the time complexity and space complexity of algorithms.

1.3 DATA AND INFORMATION

The term **data** comes from its singular form **datum**, which means 'a fact'. The data is a fact about people, places or some entities. In computers, data is simply the value assigned to a variable. The term *variable* refers to the name of a memory location that can contain only one data at any point of time. For example, consider the following statements:

Vijay is 16 years old.

Vijay is in the 12th standard.

Vijay got 80% marks in Mathematics.

Let 'name', 'age', 'class', 'marks' and 'subject' be some defined variables. Now, let us assign a value to each of these variables from the above statements.

Name = Vijay

Class = 12

Age = 16

Marks = 80

Subject = Mathematics

In the above example the values assigned to the five different variables are called data.

We will now see what is **information** with respect to computers. Information is defined as a set of processed data that convey the relationship between data considered. *Processing* means to do some operations or computations on the data of different variables to relate them so that these data, when related, convey some meaning. Thus, *information* is as group of related data conveying some meaning. In the examples above, when the data of the variables 'name' and 'age' are related, we get the following information:

Vijay is 16 years old.

In the same example, when the data of the variables 'name' and 'class' are related we get another information as:

Vijay is in the 12th standard.

Further, when we relate the data of the variables, 'name', 'marks', and 'subject', we get more information that Vijay got 80% marks in Mathematics. The following figure shows how information can be drawn from data.

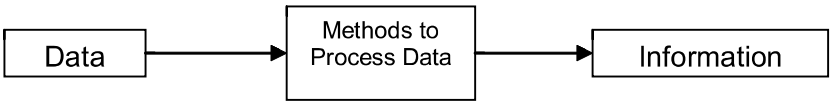


Figure 1.1: Relation between data and information

1.4 DATA STRUCTURE AND ITS TYPES

The way in which the various data elements are organized in memory with respect to each other is called a *data structure*. Data structures are the most convenient way to handle data of different types including *abstract data type* for a known problem. Again problem solving is an essential part of every scientific discipline. To solve a given problem by using a computer, you need to write a program for it. A program consists of two components: *algorithm* and d*ata structure*.

Many different algorithms can be used to solve the same problem. Similarly, various types of data structures can be used to represent a problem in a computer.

To solve the problem in an efficient manner, you need to select a combination of algorithms and data structures that provide maximum efficiency. Here, efficiency means that the algorithm should work in minimal time and use minimal memory. In addition to improving the efficiency of an algorithm, the use of appropriate data structures also allows you to overcome some other programming challenges, such as:

– simplifying complex problems

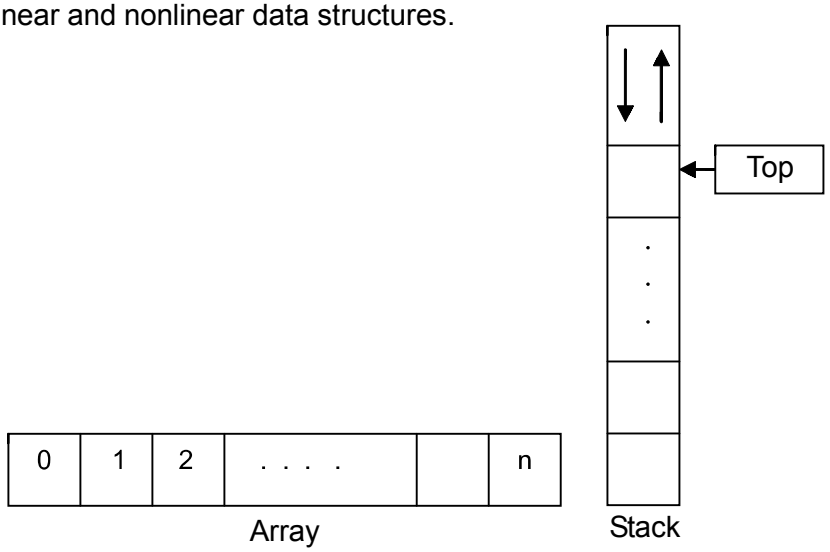
– creating standard, reusable code components

– creating programs that are easy to understand and maintain

Data can be organized in many different ways; therefore, you can create as many data structures as you want. However, data structures have been classified in several ways. Basically, data structures are of two types: **linear data structure** and **non linear data structure**.

Linear data structure: A data structure is said to be linear if the elements form a sequence i.e., while traversing sequentially, we can reach only one element directly from another. For example: Array, Linked list, Queue etc.

Non linear data structure: Elements in a nonlinear data structure do not form a sequence i.e. each item or element may be connected with two or more other items or elements in a non-linear arrangement. Moreover removing one of the links could divide the data structure into two disjoint pieces. For example: Trees and Graphs etc. The following figures shows the linear and nonlinear data structures.



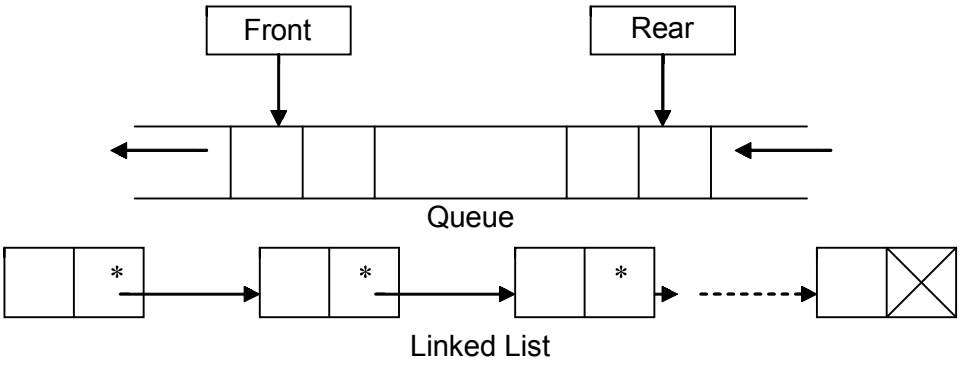


Figure 1.2: Linear data structure

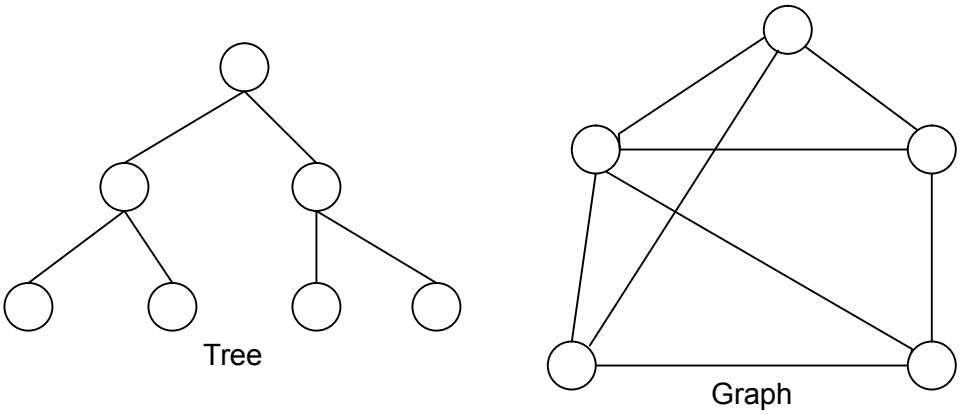


Figure 1.3: Non linear data structure

All these data structures are designed to hold a collection of data items.

1.5 DATA STRUCTURE OPERATIONS

We come to know that data structure is used for the storage of data in computer so that data can be used efficiently. The data manipulation within the data structures are performed by means of certain operations. In fact, the particular data structure that one chooses for a given situation depends largely on the frequency with which specific operations are performed. The following four operations play a major role on data structures.

a) *Traversing:* accessing each record exactly once so that certain items in the record may be processed. (This accessing and processing is sometimes called “visiting” the record.)

b) *Searching:* finding the location of the record with a given key value, or finding the locations of all records, which satisfy one or more conditions.

c) *Inserting:* adding a new record to the structure.

d) *Deleting:* removing a record from the structure.

Sometimes two or more of these operations may be used in a given situation. For example, if we want to delete a record with a given key value, at first we willl have need to search for the location of the record and then delete that record.

The following two operations are also used in some special situations:

i) *Sorting:* operation of arranging data in some given order, such as increasing or decreasing, with numerical data, or alphabatically, with character data.

ii) *Merging:* combining the records in two different sorted files into a single sorted file.

1.6 CONCEPT OF DATA TYPES

A *data type* is nothing but a term that refers to the type of data values that may be used for processing and computing. The fundamental data types in C are *char, int, float* and *double*. These data types are called built-in data types. There are three categories of data types in C, they are:

a) Built-in types, includes *char, int, float* and *double*

b) Derived data types, includes *array* and *pointers*

c) User defined types, includes *structure, union and enumeration.*

In this section, we will briefly review the data types *array*, *pointers* and *structures* that we have already learnt in the first semester course Programming in C Language.

1.6.1 Array

An array is a collection of two or more adjacent memory locations containing same types of data. These data are the array elements. The individual data items can be characters, integers, floating-point numbers, etc. However, they must all be of the same type and the same storage class.

Each array element (i.e., each individual data item) is referred to by specifying the array name followed by one or more **subscripts**,

with each subscript enclosed in square brackets. The syntax for declaration of an array is given below.

Storage Class datatype arrayname [expression]

Here, *storage class* may be *auto, static or extern*, *Storage class* refers to the permanence of a variable, and its scope within the program, i.e., the portion of the program over which the variable is recognized. If the storage class is not given then the compiler assumes it is an *auto* storage class.

The array can be declared as:

int x[15]; **x is an 15 element integer array**

char name[25]; **name is a 25 element character array**

In the array x, the array elements are x[0], x[1], ..., x[14] as illustrated in the figure 1.4.

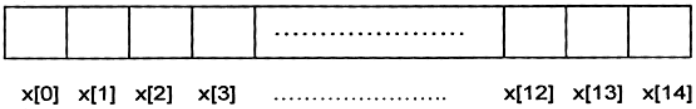


Figure 1.4: An array data structure

Array can be initialize at the time of the declaration of the array. For example,

```
int marks [5] ={85,79,60,87,70};
```

Then, the **marks** array can be represented as follows:

85	79	60	87	70

marks [0] marks [1] marks [2] marks [3] marks [4]

Figure 1.5: the marks array after initialization

In the case of a character array *"name"* it is represented as:

```
char name [20] = { "krishna" }
```

K	r	i	s	h	n	a	\0

name [0] name [1] name [2] name [3] name [4] name[5] name[6] name[7]

Figure 1.6: "Name" array after initialization

We know that every character string terminated by a null character (\0). Some more declarations of arrays with initial values are given below:

```
char vowels [ ] = { 'A', 'E', 'I', 'O', 'U' };
int age [ ] = { 16, 21, 19, 5, 25 }
```

In the above case, compiler assumes that the array size is equal to the number of elements enclosed in the curly braces. Thus, in the above statements, size of array would automatically assumed to be 5. If the number of elements in the initializer list is less than the size of the array, the rest of the elements of the array are initialized to zero. The number of the subscripts determines the dimensionality of the array. For example,

```
marks [i],
```

refers to an element in the one dimensional array. Similarly, **matrix [i] [j]** refers to an element in the two dimensional array.

Two dimensional arrays are declared the same way that one dimensional arrays. For example,

```
int matrix [ 3 ] [ 5 ]
```

is a two dimensional array consisting of 3 rows and 5 column for a total of 20 elements. Two dimensional array can be initialized in a manner analogous to the one dimensional array:

```
int matrix [ 3 ] [ 5 ] =   {
                    { 10, 5,  -3, 9, 2 },
                    {  1 , 0, 14, 5, 6 },
                    { -1, 7,  4,  9,  2 }

                };
```

The matrix array can be represented as follows:

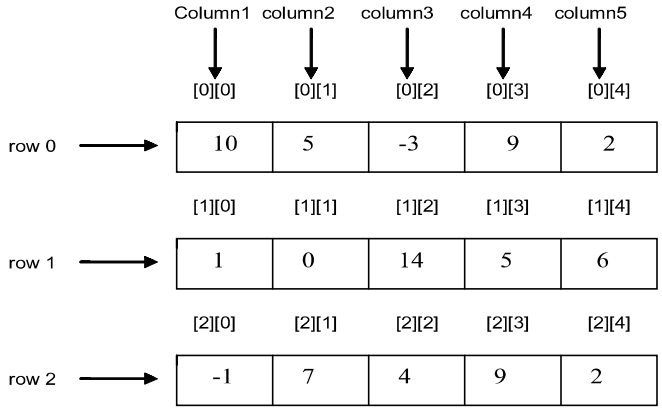


Figure 1.7: Matrix array after initialization

The above statement can be written as follows:

int matrix [3] [5] = {10, 5, –3, 9, 2, 1, 0, 14, 5, 6, –1, 7, 4, 9, 2}

A statement such as:

```
int matrix [ 3 ] [ 5 ] =   {
                             {10, 5, -3},
                             {1, 0, 14},
                             {-1, 7, 4}
                           };
```

only initializes the first three elements of each row of the two dimensional array *matrix*. The remaining values are set to 0.

Program 1.1: A Simple Program Using One-dimensional Array

A one dimensional array is used when it is necessary to keep a large number of items in memory and reference all the items in a uniform manner. Let us try to write a program to find average marks obtained by a class of 30 students in a test.

```
#include<stddio.h>
#include<conio.h>
void main( )
   {
      int avg, sum = 0;
      int i; int marks[30]; /* array declaration */
      clrscr( );
      for ( i = 0 ; i<=29 ; i++ )
      {
         printf ( ''\nEnter marks'' ) ;
         scanf ( ''%d", &marks[i] ) ; /* store
         data in array */
      }
   for ( i = 0 ; i <= 29 ; i++ )
      sum = sum + marks[i] ; /* read data
      from an array */
      avg = sum / 30 ;
```

```
        printf ( "\nAverage marks = %d", avg) ;
        getch( );
}
```

Program 1.2: A Simple Program Demonstrating the use of Two-Dimensional Array

A transpose of a matrix is obtained by interchanging its rows and columns. If A is a matrix of order m x n then its transpose A^T will be of order n x m. Let us implement this program using two dimensional array as follows:

```
#include<stdio.h>
#include<conio.h>
void main()
   {
      int matrix1 [20][20], matrix2 [20][20], i,
      j, m, n;
      clrscr();
      printf("Enter the Rows and Column of
      matrix\n");
      scanf("%d %d", &m, &n);
      printf("Enter the elements of the Matrix\n");
      for(i=1; i<m+1; i++)
      for(j=1; j<n+1; j++)
      scanf("%d", &matrix1 [i][j]);
      for(i=1; i<n+1; i++)
      for(j=1; j<m+1; j++) /*stores the elements
in matrix2*/
      matrix2 [i][j] = matrix1 [j][i];
      printf("\n\t Transpose of Matrix\n");
      for(i=1; i<n+1; i++)
   {
      for(j=1; j<m+1; j++)
      printf( "%3d", matrix2 [i][j]);
```

```
        printf("\n");
    }
    getch();
}
```

1.6.2 Pointers

You have already introduced to the concept of pointer. We will recall some of its properties and applications. A **pointer** is a variable that represents the **location** (rather than the **value**) of a data item, such as a variable or an array element.

Suppose we define a variable called **sum** as follows:

int sum = 25;

Let us define an another variable, called **pt_sum** like the following way,

int *pt_sum;

It means that **pt_sum** is a pointer variable pointing to an integer, where * is a unary operator, called the **indirection operator**, that operates only on a pointer variable.

We have already used the '**&**' *unary operator* as a part of a *scanf* statement in our C programs. This operator is known as the **address operator**, that evaluates the address of its operand.

Now, let us assign the address of sum to the variable **pt_sum** such as:

pt_sum = ∑

Now the variable **pt_sum** is called a **pointer** to **sum**, since it "points" to the location or address where **sum** is stored in memory. Remember, that **pt_sum** represents **sum's** address, not its value. Thus, **pt_sum** referred to as a **pointer variable**.

The relationship between pt_sum and sum is illustrated in Figure 1.8.

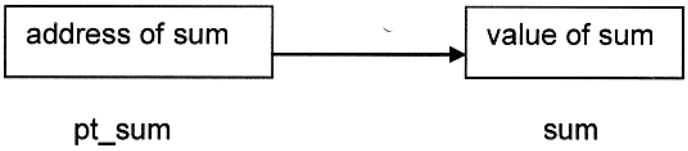


Figure 1.8: Relationship between pt_sum and sum

The data item represented by **sum** (i.e., the data item stored in sum's memory cells) can be accessed by the expression ***pt_sum**.

Therefore, ***pt_sum** and **sum** both represent the same data item i.e., 25.

Several typical pointer declarations in C program are shown below:

```
int *alpha;
char *ch;
float *s;
```

Here, alpha, ch and s are declared as pointer variables, i.e. variables capable of holding addresses. Remember that, addresses (location nos.) are always going to be whole numbers, therefore pointers always contain whole numbers.

The declaration **float *s** does not mean that **s** is going to contain a floating-point value. What it means is, **s** is going to contain the address of a floating-point value. Similarly, **char *ch** means that **ch** is going to contain the address of a char value.

Let us try to write a program that demonstrate the use of a pointer:

Program 1.3: Program to demostrate the use of a pointer

```
#include<stdio.h>
#include<conio.h>
void main( )
   {
      int a = 5;
      int *b;
      b = &a;
```

```
    clrscr();
    printf ("value of a = %d\n", a);
    printf ("value of a = %d\n", *(&a));
    printf ("value of a = %d\n", *b);
    printf ("address of a = %u\n", &a);
    printf ("address of a = %d\n", b);
    printf ("address of b = %u\n", &b);
    printf ("value of b = address of a = %u", b);
    getch();
}
```

[Suppose address of the variable a = 1024, b = 2048]

a	b
5	1024
1024	2048

OUTPUT: value of a = 5
value of a = 5
value of a = 5
address of a = 1024
address of a = 1024
value of b = address of a = 1024

Pointer to Pointer: A pointer to a pointer is a techniques used frequently in more complex programs. To declare a pointer to a pointer, place the variable name after two successive asterisks (*). In this case, one pointer variable holds the address of the another pointer variable. In the following shows a declaration of pointer to pointer:

```
int **x;
```

Following program shows the use of pointer to pointer techniques:

Program 1.4: Program showing use of a pointer

```
#include<stdio.h>
#include<conio.h>
void main( )
```

```
{
   int a = 5;
   int *b;
   int **c;
   b = &a;
   c = &b;
   printf ("value of a = %d\n", a);
   printf ("value of a = %d\n",*(&a));
   printf ("value of a = %d\n",*b);
   printf ("value of a = %d\n",**c);
   printf ("value of b = address of a = %u\n", b);
   printf ("value of c = address of b = %u\n", c);
   printf ("address of a = %u\n", &a);
   printf ("address of a = %u\n", b);
   printf ("address of a = %u\n", *c);
   printf ("address of b = %u\n", &b);
   printf ("address of b = %u\n", c);
   printf ("address of c = %u\n", &c);
   getch();
}
```

[Suppose address of the variable a = 1024, b = 2048]

a	b	c
5	1024	2048
1024	2048	4096

OUTPUT: value of a = 5
value of a = 5
value of a = 5
value of a = 5
value of b = address of a = 1024
value of c = address of b = 2048
address of a = 1024

address of a = 1024

address of a = 1024

address of b = 2048

address of b = 2048

address of c = 4096



CHECK YOUR PROGRSS

Q.1: Describe the array that is defined in each of the following statements. Indicate what values are assigned to the individual array elements.

a) char flag[5]= {'T', 'R', 'U', 'E'};

c) int p[2][4] = {
{1, 3, 5, 7},
{2, 4, 6, 8}
};

Q.2: Define a one-dimensional, six-element floating-point array called **consts.** Assign the following values to the array elements: 0.005, –0.032, 1e–6, 0.167, –0.3e8, 0.015

Q.3: Explain the meaning of each of the following declarations

a) float a, b;
float *pa, *pb;

b) float a = –0.167;
float *pa = &a;

c) char *d[4] = {"north", "south", "east", "west"};

1.6.3 Structure

Structure is the most important user defined data type in C. A structure is a collection of variables under a single name. These variables can be of different types, and each has a name which is used to select it from the structure. But in an array, all the data items are of same type. The individual variables ina structure are called *member variables*. A structure is a convenient way of grouping

several pieces of related information together.

Here is an example of a structure definition.

```
struct student
   {
      char name[25];
      char course[20];
      int age;
      int year;
   };
```

Declaration of a structure always begins with the key word **struct** followed by a user given name, here the **student**. After the opening brace there will be the member of the structure containing different names of the variables and their data types followed by the closing brace and the semicolon.

Graphical representation of the structure student is shown below:

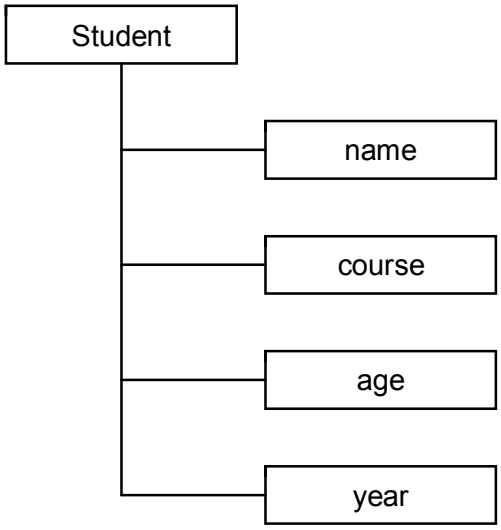


Figure 1.9: A structure named Student

Now let us see how to declare a structure variable. In the following we have declare a variable **st_rec** of type student:

student st_rec[100];

In this declaration **st_rec** is a 100 element array of structures.

Hence each element of **st_rec** is a separate structure of

type student (i.e. each element of **st_rec** represents an individual student record.)

Having declared the structure type and the structure variables, let us see how the elements of the structure can be accessed. In arrays we can access individual elements of an array using a subscript. Structures use a different scheme. They use a dot (.) operator. As an example if we want to access the name of the 10th student (i.e., **st_rec[9]** since the subscript begins with 0) from the above structure then we will have to write,

st_rec[9].name

Similarly, course and age of the 10th student can be accessed by writing,

st_rec[9].course and **st_rec[9].age**

The members of a structure variable can be assigned initial values in much the same manner as the elements of an array. Example of assigning the values for the 10th student record is shown in the following:

struct student **st_rec[9]** = {“Arup Deka”, “BCA”, 21, 2008};

Program 1.5: Program showing use of structure data type

```
#include<stdio.h>
#include<conio.h>
void main( )
   {
   struct book
      {
         char name;
         float price;
         int pages;
      };
   struct book b1, b2, b3;
   clrscr();
```

```
printf (''\nEnter names, prices & no. of pages
of 3 books\n'')
scanf (''%c %f %d'', &b1.name, &b1.price,
&b1.pages);
scanf (''%c %f %d'', &b2.name, &b2.price,
&b2.pages);
scanf (''%c %f %d'', &b3.name, &b3.price,
&b3.pages);
printf (''\nAnd this is what you entered'');
printf (''\n%c %f %d'', b1.name, b1.price,
b1.pages);
printf (''\n%c %f %d'', b2.name, b2.price,
b2.pages);
printf (''\n%c %f %d'', b3.name, b3.price,
b3.pages);
getch();
}
```

Self-referencial Structure: When a member of a structure is declared as a pointer that points to the same structure (parent structure), then it is called a *self-referential structure*. It is expressed as shown below:

```
struct node
   {
      int data;
      struct node *next;
   };
```

where '**node**' is a structure that consists of two members one is the data item and other is a pointer variable holding the memory address of the other node. The pointer variable **next** contains an address of either the location in memory of the successor node or the special value **NULL**.

Self-referential structures are very useful in applications that

involves linked data structures such as linked list and trees.

The basic idea of a linked data structure is that each component wiithin the structure includes a pointer indicating where the next component can be found. Therefore, the relative order of the components can easily be changed simply by altering the pointers. In addition, individual components can easily be added or deleted again by altering the pointers.

The diagramatic representation of a node is shown below:

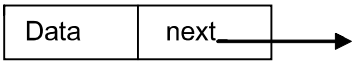


Figure 1.10: A node structure

1.7 DYNAMIC MEMORY ALLOCATION

The memory allocation process may be classified as static allocation and dynamic allocation. In static allocation, a fixed size of memory are reserved before loading and execution of a program. If that reserved memory is not sufficient or too large in amount then it may cause failure of the program or wastage of memory space. Therefore, C language provides a technique, in which a program can specify an array size at run time. The process of allocating memory at run time is known as *dynamic memory allocation*.

There are three dynamic momory allocation functions and one memory deallocation (releasing the memory) function. These are **malloc( ), calloc( ), realloc( )** and **free( )**.

malloc( ): The function *malloc( )* allocates a block of memory. The *malloc( )* function reserves a block of memory of specified size and returns a pointer of type void. The reserved block is not initialize to zero. The syntax for usinig *malloc( )* function is:

```
ptr = (cast-type *) malloc(byte-size);
```

where *ptr* is a pointer of type cast-type. The *malloc( )* returns a pointer (of cast-type) to an area of memory with size byte-size.

Suppose x is a one dimensional integer array having 15 elements.

It is possible to define *x* as a pointer variable rather than an array.

Thus, we write,

```
int *x;
```

instead of int x[15] or #define size 15
int x[size];

When *x* is declared as an array, a memory block having the capacity to store 15 elements will be reserved in advance. But in this case, when *x* is declared as a pointer variable, it will not assigned a memory block automatically.

To assign sufficient memory for *x*, we can make use of the library function *malloc*, as follows:

```
x = (int *)malloc(15 * sizeof(int));
```

This function reserves a block of memory whose size (in bytes) is equivalent to 15 times the size of an integer. The address of the first byte of the reserved memory block is assigned to the pointer *x* of type **int**. Diagammatic representation is shown below in figure 1.11.

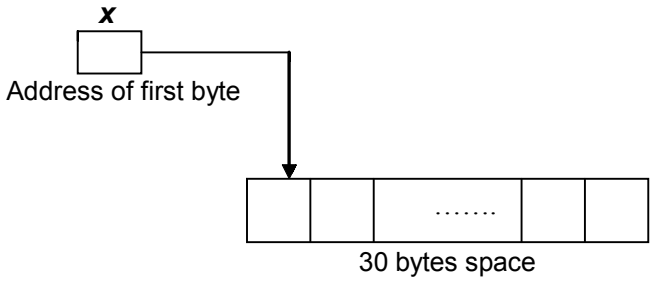


Figure 1.11: Representation of dynamic memory allocation

calloc( ): *calloc* is another memory allocation function that is normally used for requesting memory space at run time for storing derived data types such as arrays and structures. The main difference between the *calloc* and *malloc* function is that *malloc* function allocates a single block of storage space while the *calloc* function allocates a multiple blocks of storage space having the same size and intialize the allocated bytes to zero. The syntax for usinig *calloc( )* function is:

```
ptr = (cast-type *) calloc (n, element-size)
```

Where, n is the number of contiguous blocks to be allocate to each of having the size *element-size*.

realloc( ): *realloc( )* function is used to change the size of the

previouslly allocated memory blocks. If the previously allocated memory is not sufficient or much larger and we need more space for more elements or we need reduced space for less elements then by the using the realloc function block size can be maximize or minimize. The syntax for usinig *realloc( )* function is:

```
ptr = realloc(ptr, newsize)
```

where, *newsize* is the size of the memory space to be allocate.

free( ): It is necessary to free the memory allocated so that the memory can be reused. The free( ) function frees up (deallocates) memory that was previously allocated with malloc( ), calloc( ) or realloc( ). The syntax for usinig *free( )* function is:

```
free(ptr)
```

where, *ptr* is apointer to a memory block, which has already been created by *malloc* or *calloc*.

1.8 ABSTRACT DATA TYPES

By now, you are well acquainted with data types, like integers, arrays, and so on. To access the data, you have used operations defined in the programming language for the data type. For example, array elements are accessed by using the square bracket notation, or scalar values are accessed simply by using the name of the corresponding variables.

This approach doesn't always work on large and complex programs in the real world. A modification to a program commonly requires a change in one or more of its data structures. It is the programmers responsibility to create special kind of data types. The programmer needs to define everything related to the new data type such as:

- how the data values are stored,
- the possible operations that can be carried out with the custom data type and
- new data type should be free from any confusion and must behave like a built-in type

Such custom data types are called **Abstract Data Types**.

Thus, an abstract data type is a formal specification of the logical properties of a data type such as its values, operations that are to be defined for the data type etc. It hides the detailed implementation of the data type and provides an interface to manipulate them.

Examples of abstract data types are– stacks, queues etc. We will discuss on these abstract data types in the next units.



CHECK YOUR PROGRESS

Q.4: Define a structure named complex having two floating point members *real* and *imaginary*. Also declare a variable x of type complex and assign the initial vaules 3.25 and –2.25.

Q.5: Declare a one dimensional, 75 element array called *sum* whose elements are stucture of type complex (declared in the above question).

Q.6: Define a self referential structure named *team* with the following three members :

a) a character array of 30 elements called name

b) an integer called age

c) a pointer to another strucutre of this same type, called next.

Q.7: a) The free( ) function is used to:

i) release the memory ii) destroy the memory

iii) create a link iv) none of the above

b) The data type defined by the user is known as

i) abstract data type ii) classic data type

iii) built-in data type iv) all of the above



1.9 LET US SUM UP

- The *data* is a fact about people, places or some entities. In computers, data is simply the value assigned to a variable.
- *Information* is a group of related data conveying some meaning.
- Data structures are of two types: linera data structure and non linear data structure. For example, Array, Linked list, Queue etc. are linear datastructure and Trees and Graphs etc are non-linear data structure.
- The possible data structure operations are– traversing, searching, inserting, deleting, sorting and merging.
- An array is a collection of two or more adjacent memory locations containing same types of data.
- A pointer is a memory variable that stores a memory address of another variable. It can have any name that is valid for other variable and it is declared in the same way as any other variable. It is always denoted by '*'.
- A structure is a collection of variables under a single name. These variables can be of different types, and each has a name which is used to select it from the structure.
- A strcture which contains a member field that point to the same structure type are called a self-referential structure.
- There is a technique in C language, in which a program can specify an array size at run time. The process of allocating memory at run time is known as dynamic memory allocation. There are three dynamic momory allocation functions: **malloc( ), calloc( ), realloc( )** and one memory deallocation function which is **free( )**.
- an abstract data type is a formal specification of the logical properties of a data type such as its values, operations that are to be defined for the data type etc.



1.10 FURTHER READING

1) Kamthane, Ashok N. *Introduction to Data Structures in C*. Perason Education.
2) Langsam, Yedidyah; Augenstein, Moshe J. & Tenenbaum, Aaron M. *Data Structures Using C and C++*. Prentice-Hall India.
3) Lipschutz, Seymour. *Data Structures*. Schaum's Outline Series in Computers. Tata McGraw Hill.



1.11 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: a) flag[0] = 'T', flag[1] = 'R', flag[2] = 'U', flag[3] = 'E' and flag[4] is assigned to zero.

b) p[0][0] = 1, p[0][1] = 3, p[0][2] = 5, p[0][3] = 7, p[1][0] = 2, p[1][1] = 4, p[1][2] = 6, p[1][3] = 8

Ans. to Q. No. 2: float consts[6] = {0.005, –0.032, 1e-6, 0.167, –0.3e8, 0.015}

Ans. to Q. No. 3: a) a and b are floating point variables, pa and pb are pointers to floating point quantities (though not necessarily to a and b)

b) a is a floatinig point variable whose initial value is –0.167; pa is a pointer to a floating point quantity, the address of a is assigned to pa as an intial value.

c) d is a one dimensional array of pointers to the string 'north', 'south', 'east' and 'west'.

Ans. to Q. No. 4:

```
struct complex
    {
       float  real;
       float imaginary;
    };
struct complex x = { 3.25, –2.25 }
```

Ans. to Q. No. 5: struct complex sum[75];

Ans. to Q. No. 6:

```
struct team
{
   char name[30];
   int age;
   struct team *next;
};
```

Ans. to Q. No. 7: a) i.; b) i.



1.12 MODEL QUESTIONS

Q.1: What is information? Explain with few examples.

Q.2: What is data? Explain with few examples.

Q.3: Name and describe the four basic data types in C.

Q.4: What is a data structure? Why is an array called a data structure?

Q.5: How does a structure differ from an array? How is a structure member accessed?

Q.6: What is a pointer? How is a pointer initialized?

Q.7: What is dynamic memory allocation? How is it useful?

Q.8: What is the principal difference between the functions *malloc()* and *calloc().*

Q.9: Define and explain abstract data type?

Q.10: What is a self-referential structure? For what kind of applications are self referential structures useful?

Q.11: Write short notes on:

i) dynamic memory allocation ii) abstract data type

iii) pointers iv) structures

v) self-referential structure

*** ***** ***

UNIT 2: ALGORITHM

UNIT STRUCTURE

2.1 Learning Objectives

2.2 Introduction

2.3 Algorithm

2.4 Complexity

- 2.4.1 Space Complexity
- 2.4.2 Time Complexity

2.5 Asymptotic Notation

- 2.5.1 Theta Θ Notation
- 2.5.2 Big O (O) Notation
- 2.5.3 Big Omega (Ω) Notation

2.6 Let Us Sum Up

2.7 Further Reading

2.8 Answers to Check Your Progress

2.9 Model Questions

2.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- define algorithm
- describe the concept of time complexity and space complexity
- describe the notations for defining the complexity of algorithm
- calculate time complexity of algorithm.

2.2 INTRODUCTION

In the previous unit, we have learnt about the different types of data structures like array, pointers and structures. These data structures are used to solve various problems. Algorithm exists for many common problems, but designing an efficient algorithm is a challenge and it plays a crucial role in large scale computer system. The concept of an algorithm is the basic need of any programming development in computer science. In this unit we

will discuss about algorithms and its space and time complexity. We will also discuss the asymptotic notation of algorithms. In the next unit, we will explore the concept of linked list and also discuss different types of linked lists.

2.3 ALGORITHM

An algorithm is a well-defined computational method which takes some value(s) as input and produces some value(s) as output. In other words, an algorithm is a sequence of computational steps that transforms input(s) into output(s).

Each algorithm must have:

- **Specification:** Description of the computational procedure
- **Pre-conditions:** The condition(s) on input
- **Body of the Algorithm:** A sequence of clear and unambiguous instructions
- **Post-conditions**: The condition(s) on output

Consider a simple algorithm for finding the factorial of **n**.

Algorithm 2.1: Factorial (n)

```
Step 1:  FACT = 1
Step 2:  for i = 1 to n do
Step 3:  FACT = FACT*i
Step 4:  print FACT
```

In the above algorithm we have:

Specification: Computes *n*!

Pre-condition: *n*> = 0

Post-condition: FACT = *n*!

Now, let us take one more example.

Problem Definition: Sort given *n* numbers by non-descending order by using insertion sort.

Input: A sequence of n numbers $<a_1, a_2, a_3, \ldots, \ldots, a_n>$.

Output: A permutation (reordering) $<a'_1, a'_2, a'_3, \ldots, \ldots, a'_n>$ of input sequences such that $a'_1 = a'_2 = a'_3 = \ldots = a'_n$.

Insertion sort is an efficient algorithm for sorting a small number of elements. Insertion sort works the way many people sort a hand of playing

cards. We start with an empty left hand and the cards are face down on the table. Then we draw one card at a time from the table and place into correct position in the left hand.

Consider the following example of five integers:

79 43 39 58 13

Here we assume that the array has only one element that is 79 and it is sorted. So the array is:

79 43 39 58 13

Next we will take 43, since 43 is less than 79 so it will be placed before 79. After placing 43 into its place the array will be:

43 79 39 58 13

Next we will take 39, since 39 is less than 43 and 79 so it will be placed before 43. After placing 39 into its place the array will be:

39 43 79 58 13

Next we will take 58, since 58 is less than 79 but grater then 39 and 49 so it will be placed before 79 but after 43. After placing 58 into its place the array will be:

39 43 58 79 13

Finally 13 will be considered, since 13 is smaller than all other elements so it will be placed before 39. After placing 13 the sorted array will be:

13 39 43 58 79

Here in insertion sort, we consider that first (i–1) numbers are sorted then we try to insert the i^{th} number into its correct position. This can be done by shifting numbers right one number at a time until the position for i^{th} number is found.

The algorithmic description of insertion sort is given below.

Algorithm 2.2: Insertion_Sort (a[n])

```
Step 1:  for i = 2 to n do
Step 2:  current_num = a[i]
Step 3:  j = i
Step 4:  while ((j>1) and (a[j–1]>current_num)) do
Step 5:  a[j] = a[j–1]
Step 6:  j = j–1
Step 7:  a[j] = current_num
```

2.4 COMPLEXITY

Once we develop an algorithm, it is always better to check whether the algorithm is efficient or not. The efficiency of an algorithm depends on the following factors:

- Accuracy of the output
- Robustness of the algorithm
- User friendliness of the algorithm
- Time required to run the algorithm
- Space required to run the algorithm
- Reliability of the algorithm
- Extensibility of the algorithm

In case of complexity analysis, we mainly concentrate on the time and space required by a program to execute. So complexity analysis is broadly categorized in two classes

- Space complexity
- Time complexity

2.4.1 Space Complexity

Though computer memory is less costly nowadays, it is very much important to analyze the amount of memory used by a program. This is because if the algorithm takes memory beyond the capacity of the machine, then the algorithm will not be able, to successfully execute. So, it is very much important to analyze the space complexity before we execute it on the computer.

The space complexity of an algorithm is the amount of main memory needed to run the program till completion.

To measure the space complexity in absolute memory unit there are the following problems:

1) The space required for an algorithm depends on space required by the machine during execution, they are
   i) Programme space
   ii) Data space

2) The programme space is fixed and it is used to store the temporary data, object code, etc.
3) The data space is used to store the different variables, data structures defined in the program.

In case of analysis we consider only the data space, since programme space is fixed and depends on the machine where it is executed.

Consider the following algorithms for exchange of two numbers:

Algorithm 2.3: Exchange_1(a, b)

Step 1: temp = a;
Step 2: a = b;
Step 3: b = temp;

Algorithm 2.4: Exchange_2(a, b)

Step 1: a = a + b;
Step 2: b = a – b;
Step 3: a = a – b;

The first algorithm uses three variables a, b and temp and the second algorithm takes only two variables. So, if we look from the space complexity perspective the second algorithm is better than the first one.

2.4.2 Time Complexity

The time complexity of an algorithm is the amount of computer time it needs to run the program till completion. To measure the time complexity in absolute time unit there are the following problems:

1) The time required for an algorithm depends on number of instructions executed by the algorithm.
2) The execution time of an instruction depends on the computer's power. Since, different computers take different amount of time for the same instruction.

3) Different types of instructions take different amount of time on the same computer.

For time complexity analysis we design a machine by removing all the machine dependent factors called Random Access Machine (RAM). The random access machine model of computation was devised by John von Neumann to study algorithms. The design of RAM is as follows:

1) Each "simple" operation (+, –, =, if, call) takes exactly 1 step.
2) Loops and subroutine calls are not simple operations. They depend upon the size of the data and the contents of a subroutine.
3) Each memory access takes exactly 1 step.

Consider the following algorithm for add two number:

Algorithm 2.5: Add_Odd(a, b)

```
Step 1:  C = a + b;
Step 2:  return C;
```

Here this algorithm has only two simple statements so the complexity of this algorithm is 2.

Consider another algorithm for add n even number:

Algorithm 2.6: Add_Even(n)

```
Step 1:  i = 2;
Step 2:  sum = 0;
Step 3:  while i <= 2*n
Step 4:  sum = sum + i
Step 5:  i = i + 2;
Step 6:  end while;
Step 7:  return sum;
```

Here, in algorithm 2.6, Step 1, Step 2 and Step 7 are simple statement and they will execute only once. Step 3 is a loop statement and it will execute as many times as the loop condition is true and one more time for check if the condition is false. Step 5 and Step 6 are inside the loop so it will run as much as the loop condition is true. Step 6 just indicates the end of while and no cost is associated with it.

Statement	Cost	Frequency	Total cost
Step 1: i = 2;	1	1	1
Step 2: sum = 0;	1	1	1
Step 3: while i <= 2*n	1	n + 1	n + 1
Step 4: sum = sum + i	1	n	n
Step 5: i = i + 2;	1	n	n
Step 6: end while;	0	1	0
Step 7: return sum;	1	1	1
Total cost			**3n + 4**



CHECK YOUR PROGRESS

Q.1: State True or False:

a) Time complexity is the time taken to design an algorithm.

b) Space complexity is the amount of space required by a program during execution.

c) An algorithm may not produce any output.

d) Algorithms are computer programs which can be directly run into the computer.

e) If an algorithm is designed for a problem then it will work all the valid inputs for the problem.

2.5 ASYMPTOTIC NOTATION

When we calculate the complexity of an algorithm we often get a complex polynomial. To simplify this complex polynomial we use some notation to represent the complexity of an algorithm called asymptotic notation.

2.5.1 Theta (Θ) Notation

For a given function g(n), Θ(g(n)) is defined as:

$$\Theta(g(n)) = f(n): \begin{cases} \text{there exist constants } c_1 > 0,\ c_2 > 0 \text{ and } n_0 \in N \\ \text{such that } 0 \le c_1\, g(n) \le f(n) \le c_2\, g(n) \text{ for all } n \ge n_0 \end{cases}$$

In other words, a function f(n) is said to belong to Θ(g(n)), if there exists positive constants c_1 and c_2 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for sufficiently large value of n. Figure 2.1 gives a intuitive picture of functions f(n) and g(n), where f(n) = Θ(g(n)). For all the values of n at and to right of n_0, the values of f(n) lies at or above $c_1 g(n)$and at or below $c_2 g(n)$. In other words, for all $n \ge n_0$, the function f(n) is equal to g(n) to within a constant factor. So, g(n) is said to be **asymptotically tight bound** for f(n).

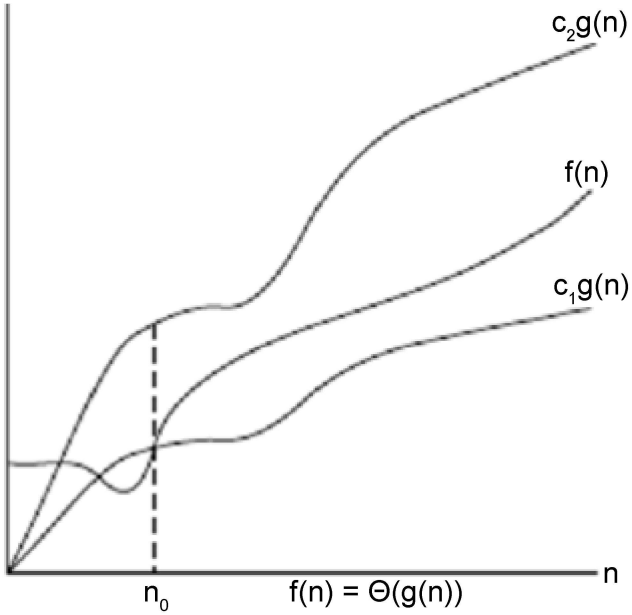


Figure 2.1: Graphic example of Θ notation

For example, $f(n) = ½n^2 – 3n$

Let $g(n) = n^2$

to proof f(n) = Θ(g(n)) we must determine the positive constants c_1, c_2 and n_0 such that,

$c_1 n^2 \le ½n^2 – 3n \le c_2 n^2$ for all $n = n_0$

Dividing the whole equation by n^2, we get,

$c_1 \le ½ – 3/n \le c_2$

We can make the right hand inequality hold for any value of $n \ge 1$ by choosing $c_2 \ge ½$. Similarly we can make the left hand inequality hold for any value of n = 7 by choosing $c_1 \le 1/14$. Thus, by choosing $c_1 = 1/14$, $c_2 = ½$. And $n_0 = 7$, we can have, f(n) = Θ(g(n)).

That is $½n^2 – 3n = Θ(n^2)$.

2.5.2 Big O (O) Notation

For a given function g(n), O(g(n)) is defined as:

$$O(g(n)) = f(n): \begin{cases} \text{there exist constants } c > 0, \text{ and } n_0 \in N \\ \text{such that } 0 \le f(n) \le c\, g(n) \text{ for all } n \ge n_0 \end{cases}$$

In other words a function f(n) is said to belong to O(g(n)), if there exists positive constant c such that $0 \le f(n) \le c\, g(n)$ for sufficiently large value of n. Figure 2.2 gives a intuitive picture of functions f(n) and g(n), where f(n) = O (g(n)). For all the values of n at and to right of n_0, the values of f(n) lies at or below cg(n). So, g(n) is said to be an **asymptotically upper bound** for f(n).

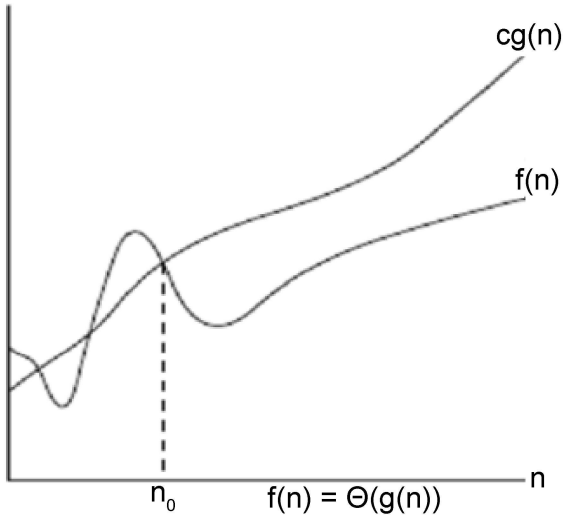


Figure 2.2: Graphic example of O notation

2.5.3 Big Omega (Ω) Notation

For a given function g(n), Ω(g(n)) is defined as:

$$\Omega(g(n)) = f(n): \begin{cases} \text{there exist constants } c > 0, \text{ and } n_0 \in N \\ \text{such that } 0 \le c\, g(n) \le f(n) \text{ for all } n \ge n_0 \end{cases}$$

In other words, a function f(n) is said to belong to Ω(g(n)), if there exists positive constant c such that $0 \le c\, g(n) \le f(n)$ for sufficiently large value of n. Figure 2.3 gives a intuitive picture of functions f(n) and g(n), where f(n) = Ω(g(n)). For all the values of n at and to right of n_0, the values of f(n) lies at or above cg(n). So, g(n) is said an asymptotically lower bound for f(n).

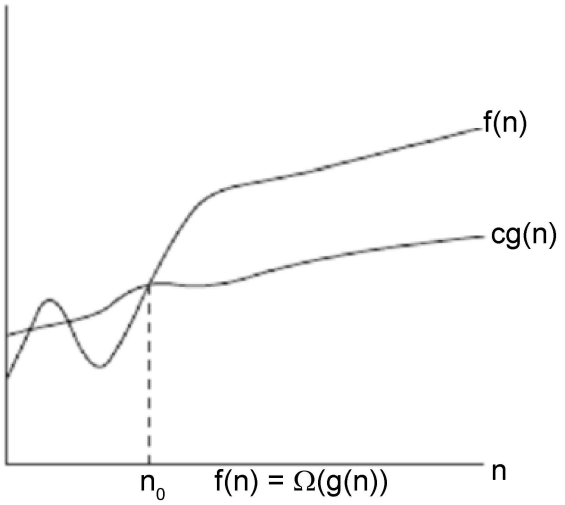


Figure 2.3: Graphic example of Ω notation

The growth patterns of order notations have been listed below:

$O(1) < O(\log(n)) < O(n) < O(n \log(n)) < O(n^2) < O(n^3) \ldots < O(2^n)$.

The common name of few order notations is listed below:

Notation	Name
O(1)	Constant
O(log(n))	Logarithmic
O(n)	Linear
O(nlog(n))	Linearithmic
$O(n^2)$	Quadratic
$O(c^n)$	Exponential
O(n!)	Factorial

A comparison of typical running time of different order notations for different input size is listed below:

$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1014	32768	4294967296

Now let us take few examples of the above asymptotic notations.

Example 2.1: Prove that $3n^3 + 2n^2 + 4n + 3 = O(n^3)$

Here, $f(n) = 3n^3 + 2n^2 + 4n + 3$

$g(n) = O(n^3)$

to proof $f(n) = O(g(n))$ we must determine the positive constants c and n_0 such that, $3n^3 + 2n^2 + 4n + 3 \leq cn^3$ for all $n \geq n_0$

dividing the whole equation by n^3, we get,

$3 + 2/n + 4/n^2 + 3/n^3 \leq c$

We can make the inequality hold for any value of n = 1 by choosing $c \geq 12$. Thus, by choosing $c \geq 12$ and $n_0 = 1$, we can have,

$f(n) = O(g(n))$.

Thus, $3n^3 + 2n^2 + 4n + 3 = O(n^3)$.

Example 2.2: Prove that, $3n^3 + 2n^2 + 4n + 3 = \Omega(n^3)$

Here, $f(n) = 3n^3 + 2n^2 + 4n + 3$

$g(n) = O(n^3)$

to proof $f(n) = \Omega(g(n))$ we must determine the positive constants c and n_0 such that,

$c\, n^3 \leq 3n^3 + 2n^2 + 4n + 3$ for all $n \geq n_0$

dividing the whole equation by n^3, we get,

$c \leq 3 + 2/n + 4/n^2 + 3/n^3$

We can make the inequality hold for any value of $n \geq 1$ by choosing $c \leq 3$. Thus, by choosing c = 3 and $n_0 = 1$ we can have,

$f(n) = \Omega(g(n))$.

Thus, $3n^3 + 2n^2 + 4n + 3 = \Omega(n^3)$.

Example 2.3: Prove that $7n^3 + 7 = \Theta(n^3)$

Here, $f(n) = 7n^3 + 7$

$g(n) = O(n^3)$

to proof $f(n) = \Theta(g(n))$ we must determine the positive constants c_1, c_2 and n_0 such that,

$c_1 n^3 \leq 7n^3 + 7 \leq c_2\, n^3$ for all $n \geq n_0$

dividing the whole equation by n^3, we get,

$c_1 \leq 7 + 7/n^3 \leq c_2$

We can make the right hand inequality hold for any value of $n \geq 1$ by choosing $c_2 \geq 14$. Similarly we can make the left hand inequality hold

for any value of $n \geq 1$ by choosing $c_1 \leq 7$. Thus, by choosing $c_1 = 7$, $c_2 = 14$. And $n_0 = 1$ we have $f(n) = \Theta(g(n))$.

Thus, $7n^3 + 7 = \Theta(n^3)$.

Now let us take few examples of algorithms and represent their complexity in asymptotic notations.

Example 2.4: Consider the following algorithm to find out the sum of all the elements in an array:

Statement	**Cost**	**Frequency**	**Total cost**
Sum_Array(arr[], n)			
Step 1: i = 0;	1	1	1
Step 2: s = 0;	1	1	1
Step 3: while i < n	1	n + 1	n + 1
Step 4: s = s + arr [i]	1	n	n
Step 5: i = i + 1;	1	n	n
Step 6: end while;	0	1	0
Step 7: return s;	1	1	1
Total Cost:			**3n + 4**

Here, $f(n) = 3n + 4$

Let, $g(n) = n$

If we want to represent it in O notation then we have to show that for some positive constant c and n_0

$0 \leq f(n) \leq cg(n)$

$\Rightarrow 0 \leq 3n + 4 \leq cn$

Now if we take n = 1 and c = 7

$\Rightarrow 0 \leq 3 \times 1 + 4 \leq 7 \times 1$

Which is true, so we can say that for $n_0 = 1$ and c = 7

$f(n) = O(g(n))$ that is $3n + 4 = O(n)$.

Example 2.5: Consider the following algorithm to add two square matrixes.

Statement	Cost	Frequency	Total cost
Mat_add(a[], n, b[])			
Step 1: i = 0	1	1	1
Step 2: j = 0;	1	1	1
Step 3: while i < n	1	n + 1	n + 1
Step 4: while j < n	1	n(n + 1)	n(n + 1)
Step 5: c[i][j] = a[i][j] + b[i][j]	1	n*n	n*n
Step 6: j = j + 1	1	n*n	n*n
Step 7: end inner while;	0	n	0
Step 8: i = I + 1	1	n	n
Step 9: end outer while	0	1	0
Step 10: return c;	1	1	1
Total Cost:			**$3n^2 + 3n + 4$**

Here, $f(n) = 3n^2 + 3n + 4$

Let, $g(n) = n^2$

If we want to represent it in Ω notation then we have to show that for some positive constant c and n_0

$0 \leq cg(n) \leq f(n)$

$\Rightarrow 0 \leq cn^2 \leq 3n^2 + 3n + 4$

Now if we take n = 1 and c = 3

$\Rightarrow 0 \leq 3 \times 1 \leq 3 \times 1^2 + 3 \times 1 + 4$

Which is true, so we can say that for $n_0 = 1$ and c = 3

$f(n) = \Omega(g(n))$ that is $3n^2 + 3n + 4 = O(n^2)$.

In the analysis of algorithm we may have three different cases depending on the input to the algorithm. They are:

Worst Case: Worst case execution time is an upper bound for execution time with any input. It guarantees that, irrespective of the type of input, the algorithm will not take any longer than the worst case time.

Average case: This gives the average execution time of algorithm. Average case execution time is the execution time taken by an algorithm in average for any random input to the algorithm.

Best Case: Best case execution time is the lower bound for execution time with any input. It guarantees that under any circumstances the execution time of the algorithms will be at best case execution time.

Example 2.6: Consider the following Insertion sort algorithm:

Insertion_Sort(a[n]) Algorithm

```
Step 1:  i = 2
Step 2:  while i < n
Step 3:  num = a[i]
Step 4:  j = i
Step 5:  while ((j >1) && (a[j–1] > num))
Step 6:  a[j] = a[j–1]
Step 7:  j = j – 1
Step 8:  end while (inner)
Step 9:  a[j] = num
Step 10: i = i + 1
Step 11: end while (outer)
```

Worst case Analysis of Insertion Sort: In the *worst case* inputs to the algorithm will be reversely sorted. So the loop statement will run for maximum time. In worst case in every time we will find a[j – 1] > num in line 5 as true. So this statement will run for 2 + 3 + 4 + … + n times total n(n + 1) – 1 times. Statement 6 will run for 1 + 2 + 3 + … + n – 1 times total n(n – 1) times. It is the same for statement 7.

Statement	Cost	Frequency	Total cost
Step 1	1	1	1
Step 2	1	n	n
Step 3	1	n – 1	n – 1
Step 4	1	n – 1	n – 1
Step 5	1	n(n + 1) –1	n(n + 1) – 1
Step 6	1	n(n – 1)	n(n – 1)
Step 7	1	n(n – 1)	n(n – 1)
Step 8	0	n – 1	0
Step 9	1	n – 1	n – 1

Step 10	1	n – 1	n – 1
Step 11	0	1	0
Total Cost			**$3n^2 + 4n – 4$**

Here, $f(n) = 3n^2 + 4n – 4$

Let, $g(n) = n^2$

If we want to represent it in O notation then we have to show that for some positive constant c and n_0

$0 \leq f(n) \leq cg(n)$

$\Rightarrow 0 \leq 3n^2 + 4n – 4 \leq cn^2$

Now if we take n = 1 and c = 7

$\Rightarrow 0 \leq 3 \times 1^2 + 4 \times 1 – 4 \leq 7 \times 1^2$

Which is true, so we can say that for $n_0 = 1$ and c = 7

$f(n) = O(g(n))$ that is $3n^2 + 4n – 4 = O(n^2)$.

So worst case time complexity of insertion sort is $O(n^2)$.

Average case Analysis of Insertion Sort: In *average case,* inputs to the algorithm will be random. In average case, half of the time we will find a[j - 1] > num is true and false in other half. So, statement 5 will run for (2 + 3 + 4 + … + n)/2 times total (n(n + 1) – 1)/2 times. Statement 6 will run for (1 + 2 + 3 + … + n – 1)/2 times total (n(n – 1))/2 times. Similarly, for statement 7.

Statement	Cost	Frequency	Total cost
Step 1	1	1	1
Step 2	1	n	n
Step 3	1	n – 1	n – 1
Step 4	1	n – 1	n – 1
Step 5	1	(n(n + 1) – 1)/2	(n(n + 1) – 1)/2
Step 6	1	(n(n – 1))/2	(n(n – 1))/2
Step 7	1	(n(n – 1)/2	(n(n – 1))/2
Step 8	0	n – 1	0
Step 9	1	n – 1	n – 1
Step 10	1	n – 1	n – 1
Step 11	0	1	0
Total Cost			**$^3/_2\, n^2 + {}^7/_2\, n – 4$**

Similarly we can show that average case time complexity of insertion sort is $O(n^2)$.

Best case Analysis of Insertion Sort: In the *best case* inputs will be already sorted. So a[j–1] > num will be false always. So statement 5 will run for n times (only to check the condition is false). Statement 6 will run for 0 times since while loop will be false always. It is the same for statement 7.

Statement	Cost	Frequency	Total cost
Step 1	1	1	1
Step 2	1	n	n
Step 3	1	n – 1	n – 1
Step 4	1	n – 1	n – 1
Step 5	1	n	n
Step 6	1	0	0
Step 7	1	0	0
Step 8	0	n – 1	0
Step 9	1	n – 1	n – 1
Step 10	1	n – 1	n – 1
Step 11	0	1	0
Total Cost			**5n – 3**

Similarly we can show that best case time complexity of insertion sort is O(n).



CHECK YOUR PROGRESS

Q.2: State True or False:

a) $7n^3 + 4n + 27 = O(n^3)$ b) $2n^2 + 34 = \Omega(n^3)$

c) $2n^2 + 34 = O(n^3)$ d) $2n^2 + 34 = \Theta(n^3)$

e) $2n^2 + 34 = \Omega(n)$ f) $2n^2 + 34 = \Theta(n^2)$

g) $2n^7 + 4n^3 + 2n = \Omega(n^3)$ h) $2n^4 + 3n^3 + 17n^2 = O(n^3)$

Q.3: Define time complexity.

Q.4: Define space complexity.



2.6 LET US SUM UP

- An algorithm is a sequence of computational steps that start with a set of input(s) and finish with valid output(s)
- An algorithm is correct if for every input(s), it halts with correct output(s).
- Computational complexity of algorithms are generally referred to space complexity and time complexity of the program
- The space complexity of an algorithm is the amount of main memory needed to run the program till completion.
- The time complexity of an algorithm is the amount of computer time it needs to run the program till completion.
- $O(1) < O(\log(n)) < O(n) < O(n \log(n)) < O(n^2) < O(n^3) \ldots < O(2^n)$.



2.7 FURTHER READING

1) Aho, A. V. & Hopcroft, J. E. (1999). *The Design and Analysis of Computer Algorithms*. Pearson Education India.
2) Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. (2009). *Introduction to Algorithms*, MIT Press.
3) Horowitz, E., Sahni, S. & Rajasekaran, S. (2009). *Fundamental of Data Structure in C*.



2.8 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: a) False, b) True, c) False, d) False, e) True

Ans. to Q. No. 2: a) True, b) False, c) True, d) False, e) True, f) True, g) True, h) False

Ans. to Q. No. 3: The time complexity of an algorithm is the amount of computer time it needs to run the program till completion.

Ans. to Q. No. 4: The space complexity of an algorithm is the amount of main memory is needed to run the program till completion.



2.9 MODEL QUESTIONS

Q.1: Given an array of n integers, write an algorithm to find the smallest element. Find the number of instruction executed by your algorithm. What are the time and space complexities?

Q.2: Write an algorithm to find the median of n numbers. Find number of instruction executed by your algorithm. What are the time and space complexities?

Q.3: Write an algorithm to sort elements by bubble sort algorithm. What are the time and space complexities?

Q.4: Explain the need of analysis of algorithm.

Q.5: Prove the following:

i) $3n^5 – 7n + 4 = \Theta(n^5)$ ii) $^1/_3n^4 – 7n^2 + 3n = \Theta(n^4)$

iii) $2n^2 + n + 4 = \Theta(n^2)$ iv) $3n^5 – 7n + 4 = O(n^5)$

v) $3n^5 – 7n + 4 = \Omega(n^5)$

*** ***** ***

UNIT 3: LINKED LIST

UNIT STRUCTURE

3.1 Learning Objectives

3.2 Introduction

3.3 Linked List

3.4 Types of Linked List

- 3.4.1 Singly Linked List
- 3.4.2 Doubly Linked List
- 3.4.3 Circular Linked List

3.5 Traversing a Single Linked List

3.6 Searching a Single Linked List

3.7 Insertion into a Single Linked List

3.8 Deletion from a Single Linked List

3.9 Reversing a Single Linked List

3.10 Let Us Sum Up

3.11 Further Reading

3.12 Answers to Check Your Progress

3.13 Model Questions

3.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- define linked list
- describe the different types of linked list
- describe and write programs on the different operations on linked list like traversal, searching, insertion and deletion
- write programs for reversing a linked list.

3.2 INTRODUCTION

We are already familiar with the array data structure. Array is a linear and homogenous data structure. It is very simple to represent a list of elements by using the array but it has some limitations. Once the size of the array is

declared, it cannot be changed during program execution. The unused portion of an array also occupies the memory space. Insertion and deletion operation in an array is time consuming and it needs shifting of elements to rearrange the numbers in order.

We will learn about *linked list* in this unit. The different types of linked lists are also discussed in this unit. In this unit, we will concentrate on the basic operations that are performed on a linked list. We shall discuss the traversing and searching procedure in a single linked list. In addition, insertion and deletion techniques will also be discussed along with reversing a linked list. In the next unit, we will discuss about stacks and applications of stacks.

3.3 LINKED LIST

In general, a series of linearly arranged finite elements having the same type is called a List. The data elements are called nodes. An array is an example of List. In array, by the index the sequential pattern is maintained. We just use the index for accessing and manipulating array elements.

Like the arrays, a linked list is also a linear data structure consisting of nodes where each node has two parts:

i) data part

ii) next part (link part)

The data part can hold data of any type (i.e. int, char, float. ... etc.) and the next part holds the address of the next node. This type of list is called a Linked list because it is a list whose order is given by links from one item to the next. An individual node is shown in the following figure.

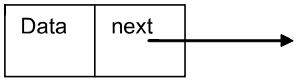


Figure 3.1: A node strucutre

A linked list is represented by a pointer (named 'START') to the first node and the *next* part of the last node is assigned the value 'NULL'. Diagrammatic representation of a linked list is shown below:

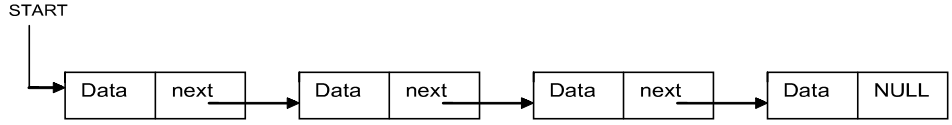


Figure 3.2: A linked list

In the above figure, the linked list contains a pointer variable called START, which contains the address of the first node in the list. Each node has two parts, a *data part* and a *next pointer*. The data part contains information which may be an integer, a character, a string etc., and *next* contains the information of pointers which holds the address of the next node. If the *next* has the value NULL, then it is the last node of the list. So, the last node can be identified by the presence of NULL.

A linked list is therefore a collection of structures ordered not by their physical placement in memory (like an array) but by logical links that are stored as part of the data in the structure itself. The link is in the form of a pointer to another structure of the same type. Such a structure is represented as follows:

```
struct node
   {
      int data;
      struct node *next;
   };
```

The first member is an integer data (it can be of any type) and the second member is a pointer to the next node. The next part or link part of a node actually contains memory address, where the data next to the current one is stored. It is also called a pointer to the next data. The structure, which contains a member field that points to the same structure type, is called a *self-referential structure*.

3.4 TYPES OF LINKED LIST

There are three types of linked list. These are:

- Singly linked list
- Doubly linked list
- Circular linked list

3.4.1 Singly Linked List

Singly linked list is a linked list in which two successive nodes of the list are linked with each other in sequential linear manner. The following figure depicts a singly linked list.

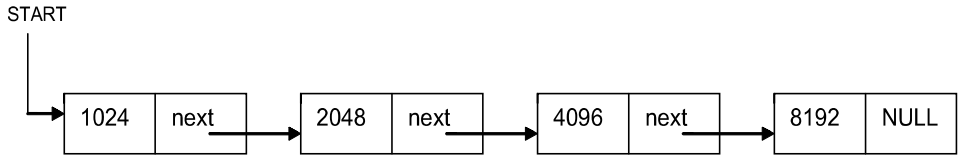


Figure 3.3: A singly linked list

In the figure, the first node is pointed by a pointer named 'START'. The NULL in the last node denotes the end of the list. 1024, 2048, 4096, 8192 are the data items of the list.

3.4.2 Doubly Linked List

Doubly linked list is another type of linked list, in which each node has double set of pointers along with the data part. One pointer is pointing to the next node (successor node) and the other pointer points to the preceding node (predecessor node). It means, in a doubly linked list addresses of next as well as preceding elements are linked with current node. This allows us to traverse the list either in forward direction or in backward direction. But in a singly linked list traverse is possible only in one direction. The following figure depicts a doubly linked list.

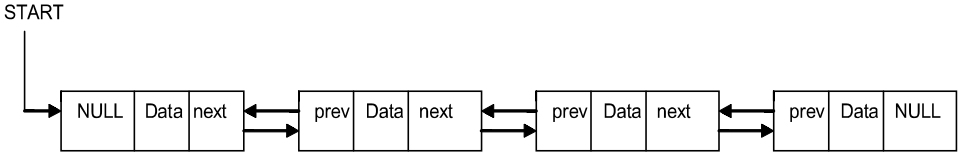


Figure 3.4: A doubly linked list

In the figure, each node has two pointers one representing the address of the successor node and the other representing the address of the predecessor node. Here, in the we have seen that there are two NULL pointers, one for the previous node and the other for the next node are being maintained. When traversing the list from left to right, it identifies the end of the list when *node* $\rightarrow$ *next* is NULL. Similarly, when traversing the list from right to left, the end of the list is indicated when *node* $\rightarrow$ *prev* is NULL. The declaration of a doubly linked list is shown below:

```
struct node
   {  struct node *prev;
      int data;
      struct node *next;
   };
```

3.4.3 Circular Linked List

Circular linked lists are the linked lists which are obtained by linking the last node of the linked list to the first node of the list. So, in a circular linked list, the last node does not contain the NULL pointer; instead it contains the address of the first node. The following figure depicts a circular linked list.

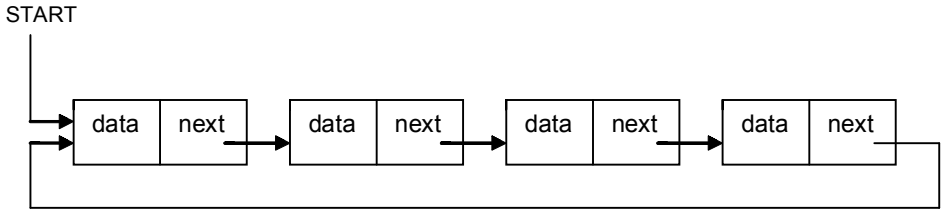


Figure 3.5: A circular linked list

One of the shortcomings of a singly linked list is that after reaching the last node in the list, we cannot go back to the previous node unless the address is available. Also, to reach the first node after reaching the end, there will be a need to traverse the whole list backward if pointers(i.e. addresses) are available. But in a circular linked list, each node can be accessed from any node. It means that the circular list can be traversed starting from any node.

3.5 TRAVERSING A SINGLE LINKED LIST

We have already learnt that linked list is a collection of elements called *node*. Each node has two components: the first component contains the *data* (or *information)* of the element and the second component called the *next field* (or *linked field*) which holds the address of the next node. To indicate the end of the list, we generally assign a NULL value. A node in a single linked list has the following structure:

```
struct node
   {
      int data;
      struct node *next;
   };
```

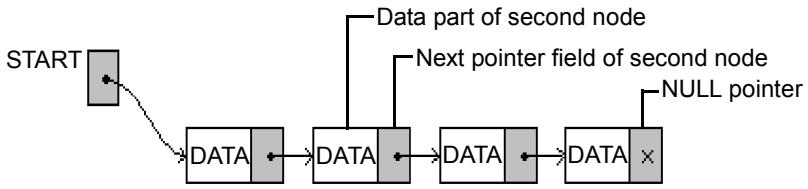


Figure 3.6: Linked List with four nodes

Here member of the structure struct node *link points to the structure itself.

Traversing a linked list: Traversing a linked list means to visit every node or element in the list starting from the first to the last element. For traversing a linked list we consider **ptr** as a pointer variable. The linked list also contains a pointer variable called **start** which contains the address of the first node in the list. **NULL** pointer denoted by x in the figure indicates the end of the list. Initially we assign the value of **start** to **ptr**. Thus, **ptr** will point to the first element of the list. For processing the next element we assign the address of the next element to the **ptr** as

```
ptr = ptr->next;
```

Now, **ptr** holds the address of the next node of the list. We can traverse each element of the linked list through this assignment untill ptr has **NULL** address (which denotes the last node of the list). Thus, the the traversal of a linked list can be implemented with the following statements:

```
while(ptr!=NULL)
   ptr=ptr->next;
```

3.6 SEARCHING A SINGLE LINKED LIST

Searching refers to the finding of an item or data from a given list of elements. So in case of a linked list searching means finding a specific node from a specific list. In searching operation we compare the KEY (item to be searched) value with each node value in the list. It can be implemented as:

```
while(ptr != NULL)
{
if(ptr -> data == key)//Key is the data be searched
        printf ("Item found in the list");
    else
        ptr =ptr -> next;
}
```

3.7 INSERTION INTO A SINGLE LINKED LIST

There may be three possibilities for inserting a node depending upon the position of insertion. The three ways are:

- Insertion at the beginning of the list
- Insertion at the end of the list
- Insertion in between two nodes

Insertion at the beginning: Let us consider a pointer **temp** which points to the node that has to be inserted. The data part of the new node is assigned to the data (i.e., d) through the temp pointer as:

```
temp->data=d;
```

Again, we know that the start pointer points to the first node of the linked list. Therefore, for inserting at the beginning we will have to assign start to the next part of temp i.e.,

```
temp->next=start;
```

So that next part of the inserted node will point to the node which was originally in first position.

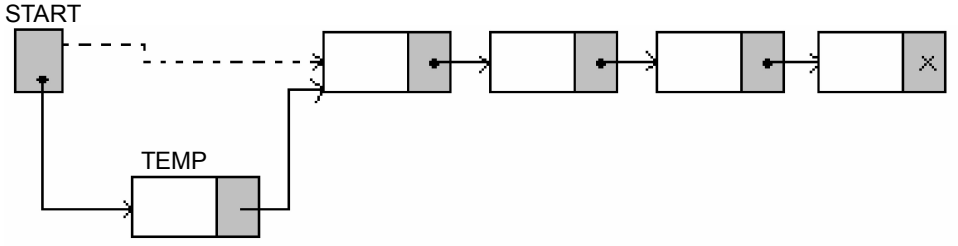


Figure 3.7: Insertion of node at the beginning

Now, inserted node points to the next node which was the first node of the linked list. So the inserted node is the first node of the linked list and start pointer is reassigned as :

```
start = temp;
```

So that start points to the newly inserted node.

Insertion at the end: Let us again consider a pointer **temp** which points to the node that has to be inserted. The data part of the new node is assigned to the data (i.e., d) through the temp pointer:

```
temp->data=d;
```

We know that to add the new node at the end position we would need to traverse the whole list to reach the last node. Once we traverse the entire linked list and reach the last node the null value of the last node will be replaced by the temp so that it points to the node to be inserted.

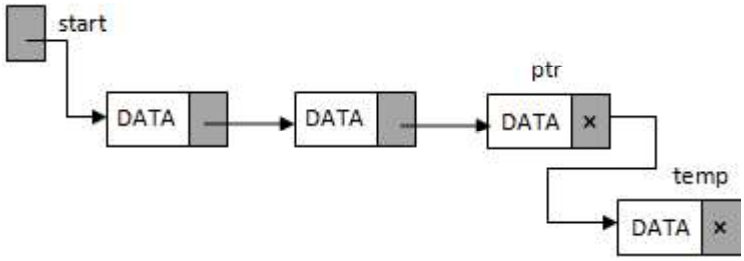


Figure 3.8: Insertion of node at the end

```
while(ptr!=NULL)
   ptr = ptr -> next;
   ptr->next = temp;
```

Insertion in between: For inserting a node at any position of the linked list we will have to first traverse the linked list for obtaining the node after which we want to insert the new node. Let us consider a pointer **temp** which points to the node that has to be inserted. The data part of the new node is assigned to the data (i.e., d) through the temp pointer as:

```
temp->data=d;
```

Again, we consider a pointer **q** which points to the node after which we have to insert the new node. For inserting the element after the node, we assign the next part of the node to the next part of new inserted node as:

```
temp->next=q->next;
```

Then the address part of the new inserted node is assigned to the next part of the previous node.

```
q->next=temp;
```

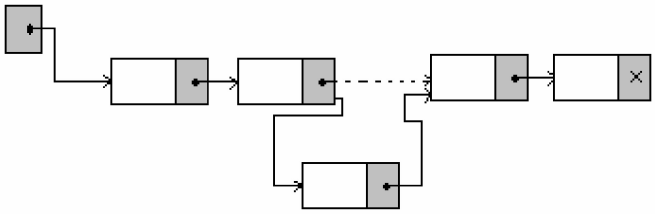


Figure 3.9: Insertion of node in between the linked list

3.8 DELETION FROM A SINGLE LINKED LIST

An element that is to be deleted from a linked list is to be searched first. For this, the traversing operation must be carried out thoroughly on the list. After finding the element there may be three cases for deletion:

- Deletion at beginning
- Deletion at end
- Deletion in between

Deletion at beginning: We know that **start** pointer points to the first element of a linked list. If the element to be deleted is the first element of linked list then we assign the value of start to temp as:

```
temp=start;
```

So now temp points to first node which has to be deleted. Now we assign the next part of the deleted node to start as:

```
start=start->next;
```

Since start points to the first element of the linked list, so start-> next will point to the second element of the linked list. Now, we should free the element to be deleted which is pointed by temp with the following statement:

```
free(temp);
```

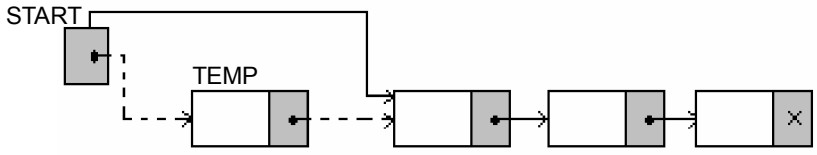


Figure 3.10: Deletion of node at the beginning

Deletion at end: If the element to be deleted is the last element of the linked list then we need to make the second last node as the last node of the linked list. We traverse the linked list using a pointer "temp" that will

point to the last node and another pointer “prev” that points to the second last node. To make the second last node, the last node of the linked list we assigh a value of NULL to the next part of prev and delete the element pointed by temp using the following statements:

```
temp=start;
   while (temp!=NULL)
   {
      prev = temp;
      temp=temp->next;
   }
   prev->next = NULL;
free(temp);
```

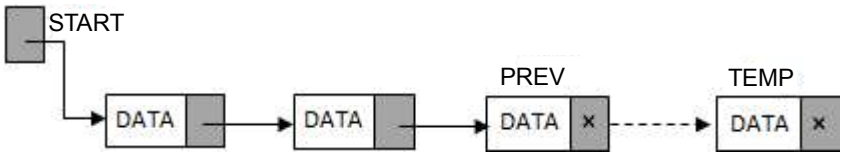


Figure 3.11: Deletion of node at the end of the linked list

Deletion in between: If the element is other than the first element of the linked list then we assign the next part of the deleted node to the next part of the previous node. This can be written as:

```
temp=q->next;
q->next=temp->next;
free(temp);
```

Here, the pointer **q** is pointing to the previous node of the node to be deleted. After the first statement **temp** will point to the node to be deleted, after second statement next of previous node will point to the next node of the node to be deleted.

If the node to be deleted is the last node of the linked list then the second statement will be as:

```
q->next=NULL;
```

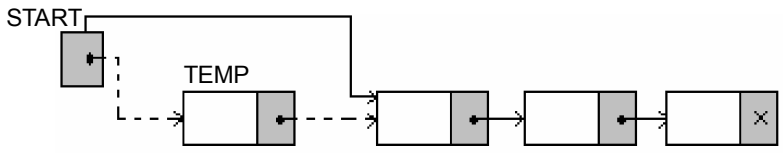


Figure 3.12: Deletion of node in between the linked list

The basic operation performed on a single linked list is implemented in the following C program.

Program 3.1: Program for operation of single linked list:

```
# include<stdio.h>
# include<conio.h>
# include<malloc.h>
struct node
{
int data;
struct node *next;
};
struct node *start;
struct node* create(int ele);
void display(struct node *);
struct node* add_beg(int ele, struct node* start);
struct node* add_end(int ele, struct node* start);
struct node* add_mid(int ele, int pos, struct node*
start);
struct node* del_beg(struct node* start);
struct node* del_end(struct node* start);
struct node*del_mid(int pos, struct node* start);
void search(int d);

int main()
{
int ch,num,pos;
clrscr();
do
{
printf("Enter\n 1 to create\n 2 to display\n 3 to
insert at beginning\n 4 to insert at end\n 5 to
insert after a postion\n 6 to delete at beginning\n
7 to delete at end\n 8 to delete at position\n 9 to
search\n 10 to exit\n");
```

```
scanf(''%d'',&ch);
    switch(ch)
    {
    case 1: printf(''Creation\n Enter number: '');
                scanf(''%d'',&num);
                start = create(num);
                break;
    case 2: printf(''Displaying list\n'');
                display(start);
                break;
    case 3: printf(''Insertion at the
            beginning.\nEnter the
            element:'');
                scanf(''%d'',&num);
                start = add_beg(num,start);
                break;
    case 4: printf(''Insertion at the
                end.\nEnter the element:'');
                scanf(''%d'',&num);
                start = add_end(num,start);
                break;
    case 5: printf(''Insertion at the
            middle.\nEnter the element:'');
                scanf(''%d'',&num);
                printf(''Enter the position of the
                element after
                which element will be added:\n'');
                scanf(''%d'',&pos);
                start = add_mid(num, pos, start);
                break;
    case 6: printf(''Delete at beginning\n'');
                start = del_beg(start);
                break;
    case 7: printf(''Delete at end\n'');
```

```
                    start=del_end(start);
                    break;
        case 8: printf(''Enter the position at which
                to delete:\n'');
                    scanf(''%d'',&pos);
                    start = del_mid(pos,start);
                    break;
        case 9: printf(''Enter the element you want
                to search:'');
                    scanf(''%d'',&num);
                    search(num);
                    break;
        case 10:exit();
        }
    } while(ch != 10);
    getch();
    return 0;
    }
    struct node* create(int ele)
    {
    struct node *cont;
    cont = (struct node *)malloc(sizeof(struct node));

    cont->data = ele;
    cont->next = NULL;

    start= cont;
    return start;
    }

    void display(struct node *temp)
    {
    temp = start;
```

```
if(temp!=NULL)
{
    while(temp->next != NULL)
    {
           printf(''%d\n'',temp->data);
           temp = temp->next;
    }
    printf(''%d\n'',temp->data);
    }
else
{
    printf(''\nThe linked list is empty\n'');
}
}

struct node * add_beg(int ele,struct node *start)
{
struct node* cont;

cont = (struct node *)malloc(sizeof(struct node));
cont->data = ele;
cont->next = start;

start=cont;
return start;
}

struct node * add_end(int ele,struct node *start)
{
struct node* temp;
struct node* cont;

cont = (struct node *)malloc(sizeof(struct node));
cont->data = ele;
```

```
cont->next = NULL;
temp = start;

while(temp->next != NULL)
{
    temp = temp->next;
}
temp->next = cont;

return start;
}

struct node * add_mid(int ele,int pos, struct node
*start)
{
struct node* temp1;
struct node* temp2;
struct node* cont;
int count=1;

cont = (struct node *)malloc(sizeof(struct node));
cont->data = ele;
temp1 = start;

while(count < pos)
{
    temp1 = temp1->next;
    count = count+1;
}

temp2=temp1->next;
temp1->next=cont;
cont->next=temp2;
```

```
return start;
}

struct node* del_beg(struct node* start)
{
struct node *temp=start;
if(temp->next==NULL)
    {
           start = NULL;
           free(temp);
    }
else
    {
           start = start->next;
           free(temp);
    }
    return start;
}

struct node* del_end(struct node* start)
{
struct node *temp=start;
struct node *prev=NULL;

           if(start->next == NULL)
           {
               start = NULL;
           }
           else
           {
               while (temp->next!=NULL)
                 {
```

```
                        prev = temp;
                        temp=temp->next;
                    }
                prev->next = NULL;
                free(temp);
                }
                return start;
}
struct node* del_mid(int pos, struct node* start)
{
struct node *temp1=start,*temp2,*prev;
int count=1;

if(pos==1)
{
start=start->next;
}
else
{
    while (count < pos)
    {
            prev=temp1;
            temp1=temp1->next;
            count = count + 1;
    }

    temp2=temp1->next;
    prev->next = temp2;
    free(temp1);
}
return start;
}
void search(int d)
```

```
{
    struct node *ptr = start;
    int pos = 1;
    while(ptr!=NULL)
    {
        if(ptr->data==d)
        {
            printf("Item %d found at position
            %d\n",d,pos);
            return;
        }
        ptr = ptr->next;
        pos++;
    }
    if(ptr == NULL)
        printf(''Item %d not found in
        list\n'',d);
}
```



CHECK YOUR PROGRESS

Q.1: Define linked list. Name the three different types of linked list.

Q.2: Visiting all the nodes in the linked list is known as:

a) Searching b) Traversing

c) Insertion d) none of these

Q.3: A ..... identifies the last logical node in a linked list.

Q.4: Write the functions in C language for the following:

a) Creating a linked list.

b) Inserting a node at the beginning of a list.

3.9 REVERSING A SINGLE LINKED LIST

Reversing a linked list is to simply change the order of the elements in the linked list. The first element in the linked list becames the last element, the second elemnt becomes the second last element and so on till the last element of the previous linked list becomes the first element of the new linked list.

Program 3.2: Program to reverse a linked list

```
#include<stdio.h>
#include<conio.h>
#include<malloc.h>

struct node
{
    int data;
    struct node *next;
};

struct node *start;
struct node* create(int ele);
void display(struct node *);
struct node* add_end(int ele, struct node* start);
struct node *reverse(struct node* start);

void main()
{
    int num;
    clrscr();
    start=NULL;

    printf(''Creation\n Enter first elemnt:'');
    scanf(''%d'',&num);
    start = create(num);
```

```
        printf(''Insertion at the end.\nEnter the 2nd
        element:'');
        scanf(''%d'',&num);
        start = add_end(num,start);

        printf(''Insertion at the end.\nEnter the 3rd
        element:'');
        scanf(''%d'',&num);
        start = add_end(num,start);

        printf(''Insertion at the end.\nEnter the 4th
        element:'');
        scanf(''%d'',&num);
        start = add_end(num,start);

        printf(''Displaying linked list\n'');
        display(start);

        printf(''Reversing linked list\n'');
        start = reverse(start);
        printf(''Displaying reversed list\n'');
        display(start);
    getch();
    }

    struct node * create(int d)
    {
        struct node *temp;
        temp= malloc(sizeof(struct node));
        temp->data=d;
        temp->next=NULL;

        if(start==NULL)/*If list is empty */
```

```
            start=temp;
        else
        printf(''Linked list already created'');

        return start;
}

void display(struct node *temp)
{
temp = start;
if(temp!=NULL)
{
        while(temp->next != NULL)
        {
            printf(''%d\n'',temp->data);
            temp = temp->next;
        }
        printf(''%d\n'',temp->data);
}
else
        {
            printf(''\nThe linked list is empty\n'');
        }
}
        struct node * add_end(int ele,struct node *start)
{
        struct node* temp;
        struct node* cont;

        cont=(struct node *)malloc(sizeof(struct node));
        cont->data=ele;
        cont->next=NULL;
        temp = start;
```

```
        while(temp->next != NULL)
        {
            temp = temp->next;
        }
            temp->next = cont;
            return start;
        }

            struct node* reverse(struct node* root)
        {
            struct node* new_root = NULL;

            while (root)
        {
            struct node* temp = root->next;
            root->next = new_root;
            new_root = root;
            root = temp;
        }
        return new_root;
    }
```



3.10 LET US SUM UP

- A **linked list** is a dynamic data structure that consists of a sequence of nodes such that in each node there are two fields: one field holds data items and the other field contains a reference (i.e., a *link*) to the next record in the sequence.
- There are three types of linked list. These are: Singly linked list, Doubly linked list and Circular linked list.
- The following are the primitive operations that can be performed with linked list: traversing, searching, insertion, and deletion

- Traversing is the procedure of passing through (visiting) all the nodes of the linked list from starting node to the ending node. When any given record is to be searched in the linked list, traversing is appied. When the given element is found, the operation can be terminated or continued for next search.
- Searching is a process in which a given element is compared with all the linked list elements.
- Insertion is the procedure through which we can add new node into a linked list at any position i.e., at the begining of the list or at any position of the list.
- Deletion is a process in which we can delete or remove elements from a linked list i.e. deleting the first node, deleting the last node as well as deleting a particular node from the linked list.
- Reversing a linked list is to simply change the order the order of elements in the linked list.



3.11 FURTHER READING

1) Kamthane, A. (2009). *Programming with ANSI and Turbo C*. Pearson Education India.
2) Langsam, Yedidyah; Augenstein, Moshe J. & Tenenbaum, Aaron M. *Data Structures Using C and C++*. Prentice-Hall India.
3) Lipschutz, S. (1987). *Schaum's Outline of Data Structure*. McGraw-Hill, Inc.



3.12 ANSWER TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: A **linked list** is a dynamic data structure that consists of a sequence of nodes such that in each node there are two fields: one field holds data items and the other field contains a reference (i.e., a *link*) to the next record in the sequence.

There are three types of linked list: single linked list, double linked list and circular linked list.

Ans. to Q. No. 2: b) Traversing

Ans. to Q. No. 3: NULL,

Ans. to Q. No. 4: a)

```
create(int d)
{
    struct node *q,*temp;
    temp= malloc(sizeof(struct node));
    temp->data=d;
    temp->next=NULL;
    if(start==NULL)/*If list is empty*/
        start=temp;
    else
    {   /*Element inserted at the end*/
        q=start;
        while(q->next!=NULL)
            q=q->next;
        q->next=temp;
    }
}
```

b)

```
add_beg(int d)
    {
        struct node *temp;
        temp=malloc(sizeof(struct node));
        temp->data=d;
        temp->next=start;
        start=temp;
    }   /*End of add_beg()*/
```



3.13 MODEL QUESTIONS

Q.1: Write a C program to remove the first and the last node of the linked list.

Q.2: Write a C program to create and display the elements of a single linked list.

Q.3: Write a C program to insert a node at the first location of a linked list.

Q.4: Describe the insertion procedure in a single linked list.

Q.5: Describe the delete operation in a single linked list.

*** ***** ***

UNIT 4: STACKS

UNIT STRUCTURE

4.1 Learning Objectives

4.2 Introduction

4.3 Stack

4.4 Operations on Stack

4.5 Implementation of Stack

- 4.5.1 Implementation of Stack using Arrays
- 4.5.2 Implementation of Stack using Linked Lists

4.6 Applications of Stack

4.7 Let Us Sum Up

4.8 Further Reading

4.9 Answers to Check Your Progress

4.10 Model Questions

4.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- describe the concept of stack
- explain LIFO structure
- describe the operations associated with a stack
- implement stack using arrays
- implement stack using linked lists
- list applications of stack.

4.2 INTRODUCTION

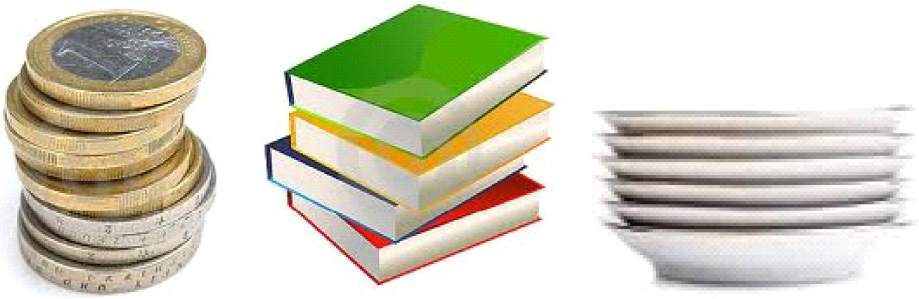
In the previous units, we have learnt about primitive data structures that are available in C. In unit 3, while studying linked list we have seen that insertion and deletion of element is possible at any place in the list. There may arise certain situations in which items may need to be inserted or removed only at one end. Stacks and queues are such data structures where insertion and deletion are done at only one end. In this unit, we will learn

about the stack data structure and its applications. In the next unit, we will explore the concept of queues.

4.3 STACK

Stack is a special type of data structure where elements are inserted from one end and elements are deleted from the same end. The position from where elements are inserted and from where elements are deleted is termed as **top** of the stack. Thus, stack is a homogeneous collection of elements of any one type, arranged linearly with access at one end only.

In order to clarify the idea of a stack let us consider some real life examples of stack. Figure 4.1 depicts three everyday examples of such a data structure. A stack of book is a good analogy: we can't read any book in the stack without first removing the books that are stacked on top of it. In the same way, plates in a hotel are kept on top of each other. When a plate is taken it is usually taken from the top of the stack. Using this approach, the last element inserted is the first element to be deleted out, and hence, stack is also called **Last In First Out** (LIFO) data structure. Although the stack may seem to be a very restricted type of data structure, it has many important applications in computer science.



Stack of coins Stack of books Stack of plates

Figure 4.1: Real life examples of Stack

4.4 OPERATIONS ON STACK

The two basic operations associated with stacks are *Push* and *Pop*.

Push: Data is added to the stack using the *Push* operation.

Pop: Data is removed using the *Pop* operation

Push operation: The procedure of inserting a new element to the top of the stack is known as *push* operation. For example, let us consider the stack shown in figure 4.2 with STACK_SIZE = 4 where we can insert atmost four elements. If we consider TOP as a pointer to the top element in a stack then after every push operation, the value of TOP is incremented by one. After inserting 11, 9, 7 and 5 there is no space to insert any element. Then we say that stack is full. If the stack is full and does not contain enough space to accept the given element or data, the stack is then considered to be in an ***overflow*** state.

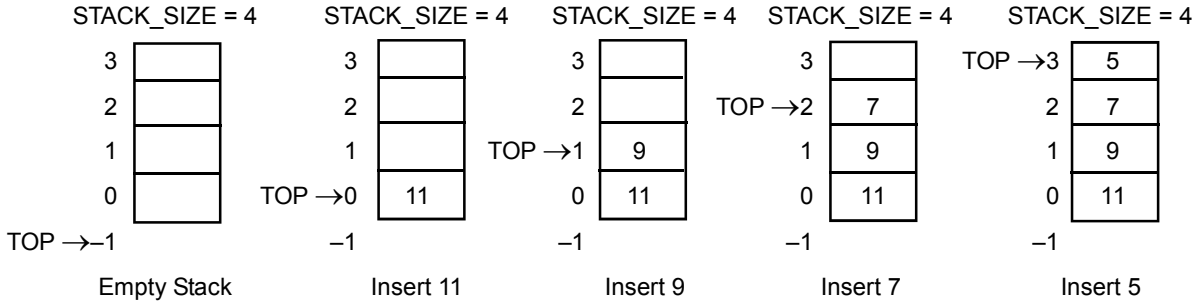


Figure 4.2: Sequence of *Push* operation

Pop operation: The procedure of removing element from the top of the stack is called *pop* operation. Only one element can be deleted from at a time and the element has to be deleted only from the top of the stack. When elements are being deleted, there is a possibility of stack being empty. When stack is empty, it is not possible to delete any more element. Trying to delete an element from an empty stack results in stack **underflow**. Figure 4.3 illustrates the pop operation. After every pop operation, the value of TOP is decremented by one.

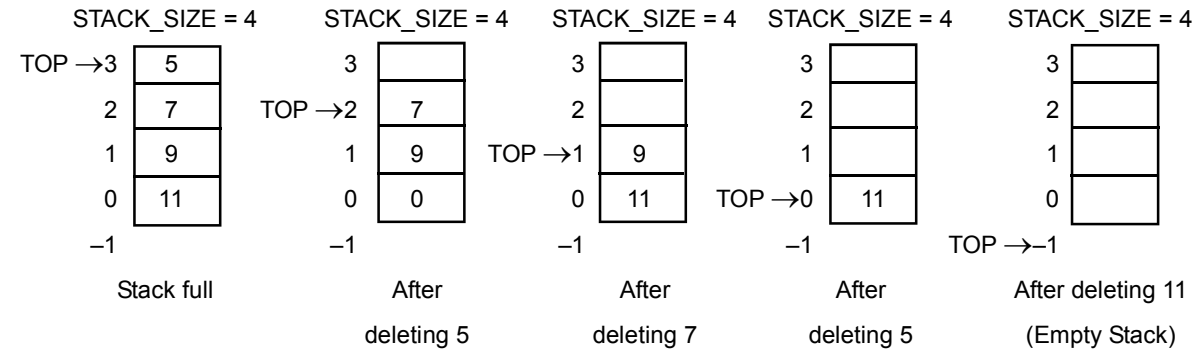


Figure 4.3: Sequence of Pop operation

In the above figure, after deleting 5, 7, 9 and 11, there are no elements in the stack and the stack becomes empty.

4.5 STACK IMPLEMENTATION

A stack can be implemented using either an array or a singly linked list. Thus there are two methods of stack implementation. They are: *Static implementation* and *Dynamic implementation*.

Static implementation can be achieved using arrays. Though array implementation is a simple technique, it provides less flexibility and is not very efficient with respect to memory organization. This is because once a size of an array is declared, its size cannot be modified during program execution. If the number of elements to be stored in a stack is less than the allocated memory, then memory is wasted and if we want to store more elements than declared, array cannot be expanded. It is suitable only when we exactly know the number of elements to be stored.

A stack can be implemented using pointers, as a form of a linked list. **Dynamic implementation** can be achieved by using linked list as it is a dynamic data structure. The limitations of static implementation can be removed using dynamic implementation. The memory is efficiently utilized with pointers. Memory is allocated only after element is inserted to the stack. The stack can grow or shrink as the program demands it to. However, if a small and/or fixed amount of data is being dealt with, it is often simpler to implement the stack as an array.

4.5.1 Implemenation of Stack using Arrays

One of the two ways to implement a stack is by using a one dimensional array. When implemented this way, the data is simply stored in the array. A variable named *"Top"* is used to point to the top element of the stack. Each time data is added or removed, *Top* is incremented or decremented accordingly, to keep track of the current *Top* of the stack. Initially, the value of *Top* is set to –1 to indicate an empty stack.

To push (or insert) an element onto the stack, *Top* is incremented by one, and the element is pushed at that position. When *Top* reaches SIZE-1 and an attempt is made to push a new element, then the stack overflows. Here, SIZE 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 displayed, 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.

Program 4.1: Stack implementation using arrays (static implementation of stacks)

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define SIZE 10
void push();
void pop();
void display();
int top= –1;// value of top is initialised to –1
int stack[SIZE];
void main()
{       int choice;
         while(1)
        {
         printf(''\n1.Push\n'');
         printf(''\n2.Pop\n'');
         printf(''\n3.Display\n'');
         printf(''\n4.Quit\n'');
         printf(''\nEnter your choice:'');
         scanf(''%d'',&choice);
         switch(choice)
         {
         case 1: push();
                 break;
```

```
            case 2: pop();
                    break;
            case 3: display();
                    break;
            case 4: exit(1);
                    default:printf(``Invalid
                    Choice\n'');
            }
        }
}
void push()
{   int item;
    if(top==(SIZE-1))
        printf(``\nStack Overflow'');
    else
    {
        printf(``\nEnter the item to be pushed in
        stack:'');
        scanf(``%d'',&item);
        top=top+1;
        stack[top]=item;
    }
}
void pop()
{
    if(top==-1)
    printf(``Stack Underflow\n'');
    else
    {
        printf(``\nPopped element is:
        %d\n'',stack[top]);
        top=top-1;
    }
```

```
}
void display()
{
    int i;
    if(top==-1)
        printf(''\nStack is empty\n'');
    else
    {
        printf(''\nStack elements:\n'');
        for(i=top;i>=0;i--)
        printf(''%d\n'',stack[i]);
    }
}
```

Stacks implemented as arrays are useful if a fixed amount of data is to be used. However, if the amount of data is not a fixed size or the amount of the data fluctuates widely during the stack's life time, then an array is a poor choice for implementing a stack. A much more elegant solution to this problem will be covered in the next section.



CHECK YOUR PROGRESS

Q.1: Fill in the blanks:

i) The insertion of an element in a stack is known as the ..... operation.

ii) The removing an element from the stack is known as ..... operation.

Q.2: What would the state of a stack be after the following operations:

create stack

push A onto stack

push F onto stack

push X onto stack

pop item from stack

push B onto stack

pop item from stack

pop item from stack

Q.3: State whether the following statements are true or false:

i) Stack follows a first-in-first-out technique.

ii) Insertion of data into the stack is called the push operation.

iii) Removal of element is termed as pop operation.

4.5.2 Implementation of Stack using Linked List

When a stack is implemented as a linked list, each node in the linked list contains the data and a pointer that gives location of the next node in the list. In this implementation. there is no need to declare the size of the stack in advance since we create nodes dynamically as well as delete them dynamically.

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, **Top** is modified to point to this new node.

Program 4.2: Program implementing stack using linked list

```
#include<stdio.h>
#include<conio.h>
#include<malloc.h>
#include<stdlib.h>
void push();     //push() function declaration
void pop();      //pop() function declaration
void display(); //display() function declaration

struct node
{
    int data;
```

```
        struct node *next;
};
struct node *top=NULL;
void main()
{
    int choice;
    clrscr();
    while(1)
    {
    printf(''\n1.Push'');
    printf(''\n2.Pop'');
    printf(''\n3.Display'');
    printf(''\n4.Quit'');
    printf(''\nEnter your choice:'');
    scanf(''%d'',&choice);
    switch(choice)
        {
        case 1: push();
              break;
        case 2: pop();
              break;
        case 3: display();
              break;
        case 4: exit(1);
        default: printf(''Invalid Choice\n'');
        } //end of switch
    } //end of while loop
} //end of main fuction
void push()
{
    struct node *ptr;
    int item;
    ptr=(struct node *)malloc(sizeof(struct node));
```

```
        printf(''Enter a value to be pushed onto the
        stack:'');
        scanf(''%d'',&item);
        ptr->data=item;
        ptr->next=top;
        top=ptr;
    } //end of push fuction
    void pop()
    {
        struct node *ptr,*next;
        if(top==NULL)
            printf(''\nStack is Empty'');
        else
        {
            ptr=top;
            printf(''\nPopped element is: %d\n'',
            ptr->data);
            top=top->next;
            free(ptr);
        }
    } //end of pop function definition
    void display()
    {
        struct node *p;
        p=top;
        if(top==NULL)
            printf(''\nStack is empty\n'');
        else
        {
            printf(''\nStack elements:\n'');
            while(p!=NULL)
            {
                  printf(''%d\n'',p->data);
```

```
                p=p->next;
            }
        }
    } //end of display function
```

4.6 APPLICATIONS OF STACKS

Stacks have many applications. For example, as processor executes a program, when a function call is made, the called function must know where to return back to the program, so the current address of program execution is pushed onto a stack. Once the function is finished, the address that was saved is removed from the stack, and execution of the program resumes.

If a series of function calls occur, the successive return values are pushed onto the stack in Last-In-First-Out order so that each function can return back to calling program. Stacks support recursive function calls in the same manner as conventional nonrecursive calls.

Stacks are also used by compilers in the process of evaluating expressions and generating machine language code. Two simple applications of stack are described below:

Reversal of string: We can reverse a string by pushing each character of the string on the stack. When the whole string is pushed on the stack we will pop the characters from the stack and we will get the reversed string.

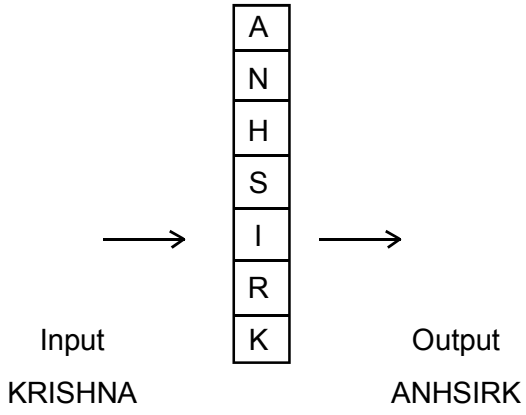


Figure 4.5: Stack

Program 4.3: Program for reversing a string using stack.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#define SIZE 25
int top=-1;
char stack[SIZE];
char pop();
void push(char);
void main()
{
    char str[20];
    int i;
    clrscr();
    gets(str);
    printf(''\nEnter a string :'');
 //Push characters of the string str on the stack
    for(i=0; i < strlen(str); i++)
        push(str[i]);
//Pop characters from the stack and store in string
//str
    for(i=0; i<strlen(str); i++)
        str[i]=pop();
    printf(''\nString in reversed order is: '');
    puts(str);
    getch();
 } //end of main
void push(char item)
{
    if(top==(SIZE-1))
        printf(''\nStack Overflow'');
    else
        stack[++top]=item;
```

```
}   //end of push function
char pop()
{
    if(top == -1)
        printf(''\nStack Underflow'');
    else
        return stack[top--];
}   //end of pop function
```

If we enter a string *State University* then the output of the above program will be *ytisrevinU etatS.*

Evaluation of arithmetic expressions: Stacks can also be useful in the evaluation of arithmetic expression. Given an expression in postfix notation. Using a stack they can be evaluated as follows:

- Scan the symbol from left to right
- If the scanned symbol is an operand, push it on to the stack.
- If the scanned symbol is an operator, pop two elements from the stack. The first popped element is op2 and the second popped element is op1. This can be achieved using the statements:
  op2=s[top--]; //First popped element is operand 2
  op1=s[top--]; //second popped element is operand 1
- Perform the indicated operation
  result=op1 op op2 //op is the operator such as +, –, /, *
- Push the result on to the stack.
- Repeat the above procedure till the end of input is encountered.



CHECK YOUR PROGRESS

Q.4: Select the appropriate option for each of the following questions:

i) The push() operation is used

a) to move an element b) to remove an element

c) to insert an element d) none of these

ii) The stack is based on the rule

a) first-in-first-out b) last-in-first-out

c) both (a) and (b) c) none of these

iii) A stack holding elements equal to its capacity and if push is performed then the situation is called

a) Stack overflow b) Stack underflow

c) Pop d) illegal operation

iv) The top pointer is increased

a) when push() operation is done

b) when pop() operation is done

c) both (a) and (b)

d) none of the above

v) The pop operation removes

a) the element lastly inserted

b) first element of the stack

c) any element randomly

d) none of the above



4.7 LET US SUM UP

- Stack is one of the most essential linear data structures and an abstract data type.
- The insertion of element onto a stack is called *Push* and deletion operation is called *pop*.
- The most and the least reachable elements in a stack are respectively known as the *top* and *bottom*.
- The insertion and deletion operations are carried out at one end.
- The recent element inserted is deleted first. If we want to delete a particular element of a stack, it is necessary to delete all the elements appearing before the element. This means that the last item to be added to a stack is the first item to be removed. Accordingly, stack is a set of elements in a *Last-In-First-Out* (**LIFO**) technique.

- When stack has no element, it is called *empty stack*.
- *Overflow* occurs when the stack is full and there is no space for a new element, and an attempt is made to push a new element.
- *Underflow* occurs when the stack is empty, and an attempt is made to pop an element.
- Static can be implemented using arrays and linked lists.



4.8 FURTHER READING

1) Kamthane, A. (2009). *Programming with ANSI and Turbo C*. Pearson Education India.
2) Lipschutz, S. (1987). *Schaum's Outline of Data Structure*. McGraw-Hill, Inc.
3) Srivastava, S. K. & Srivastava, D. (2003). *Data Structures Through C in Depth*. BPB Publications.



4.9 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: i) Push, ii) Pop

Ans. to Q. No. 2: Stack will contain only one item and the item is A.

Ans. to Q. No. 3: i) False, ii) True, iii) True

Ans. to Q. No. 4: i) (c) insert an element;
ii) (b) last-in-first-out
iii) (a) Stack overflow
iv) (a) when push() operation is done
v) (a) the element lastly inserted



4.10 MODEL QUESTIONS

Q.1: What is a stack? What different operations can be performed on stacks?

Q.2: Why are stacks called “LIFO” structures?

Q.3: Explain the *push* and *pop* operations.

Q.4: What are *stack overflow* and *stack underflow*?

Q.5: Define linked implementation of stack.

Q.6: Write a C program to implement stack with array. Perform *push* and *pop* operation.

Q.7: Write a program to perform following operations:
(a) push (b) pop (c) display all elements.

Q.8: Distinguish between static and dynamic implementation of stack.

Q.9: Describe two applications of stack.

Q.10: What are the advantages and disadvantages of linked implementation of a stack relative to the contiguous implementation?

Q.11: Write a program of stack where elements will be pushed from last position of array.

Q.12: Write a program to implement a stack which contains the address of the pointers for allocation of memory. Do the pop operation on stack and free the popped pointers.

Q.13: Write a recursive function in C to compute the sum of n natural numbers.

Q.14: What are the two ways of implementing stacks. Which one is preferred over the other and why?

Q.15: What are the various applications of stacks? Write a C program to implement any one of them.

*** ***** ***

UNIT 5: QUEUE

UNIT STRUCTURE

5.1 Learning Objectives

5.2 Introduction

5.3 Queue

5.4 Array Implementation of Queue

5.5 Circular Queue

5.6 Linked List Implementation of Queue

5.6.1 Using Singly Linked List

5.6.2 Using Doubly Linked List

5.6.3 Using Circular Linked List

5.7 Priority Queues

5.8 Application of Queues

5.9 Let Us Sum Up

5.10 Further Reading

5.11 Answers to Check Your Progress

5.12 Model Questions

5.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- describe queue data structure
- explain the usefulness of circular queue and implement it
- describe array implementation of queue
- describe list implementations of queue
- list different applications of queue.

5.2 INTRODUCTION

In the previous units, we have learnt about arrays and linked list as well as stack data structure. Now, in this unit we will learn about another linear data structure known as queue. Different implementation of queue using array and linked list are also described in detail in this unit. We will

also explore the different applications of queue in this unit. In the next unit, we will cover the different searching techniques.

5.3 QUEUE

Queue is a linear data structure in which removal of elements are done in the same order they were inserted i.e., the element will be removed first which is inserted first. A queue has two ends which are front end and rear end. Insertion is to take place at rear end and deletion is to take place at front end. Because of this, queue is also known as First In First Out (FIFO) data structure.

5.4 ARRAY IMPLEMENTATION OF QUEUE

Array implementation of queue can be done in many ways. In this array implementation of queue, a single array and two variables are used. The two variables named 'front' and 'rear' are initialized with a value of '–1'. When we want to insert an element, at first the value of 'rear' is incremented if it is within the range and then the element is inputted into the array at the subscript value equal to the value of 'rear'. The overflow condition for array implementation in C language is **"front' is equal to 0 AND 'rear' is equal to (size of the array) –1".** Now, if in any case if "'rear' is equal to (size of the array) –1 but 'front' is not equal to 0 then to insert a new element, the elements in the queue must be shifted towards the left most direction of the array, so that at the rear end one or more free spaces become available to insert new elements. Here, the underflow condition is "'front' is equal to '–1'". In case of deletion operation 'front' variable is incremented by one if it is not equal to '–1'.

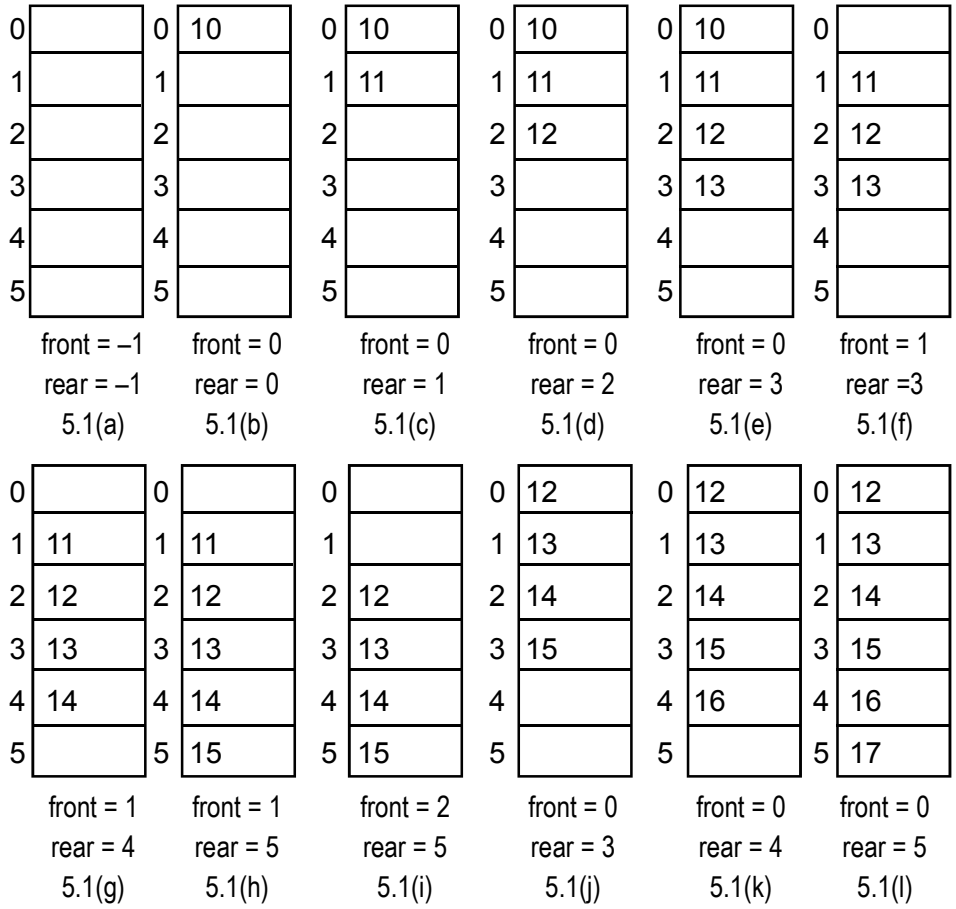


Figure 5.1: (a) Empty queue, (b) Insertion of "10", (c) Insertion of "11", (d) Insertion of "12", (e) Insertion of "13", (f) Deletion of first element, (g) Insertion of "14", (h) Insertion of "15", (i) Deletion of next element, (j) Shifting of elements, (k) Insertion of "16", (l) Insertion of "17"

In figure 5.1(a), an empty queue is represented with front = –1 and rear = –1. In figure 5.1(b), 10 is inserted into the queue and it is the first element in the queue with front = 0 and rear = 0. In figure 5.1(c), 11 is inserted into the queue with front = 0 and rear = 1. Similarly to this, elements 12, 13, 14, 15, 16 and 17 are inserted into the queue in figures 5.1(d), (e), (g), (h), (k) and (l) respectively. In figure 5.1(j), the existing elements in the queue are shifted towards the beginning of the queue to make free space for the insertion of 16 and 17. In figure 5.1(f), 10 is deleted with front = 1 and rear = 3 and in 1(i), 11 is deleted with front = 2 and rear = 5.

Algorithm 5.1: Algorithm for inserting new information into a queue

Here, queue[ ] is the array to store the array elements
'front' is a variable used to represent the front end of the queue
'rear' is a variable to represent the rear end of the queue
'element' is a variable to store the new element to be inputted into the queue.

Insert (queue [ ], front, rear, element)

Step 1: IF front == 0 AND rear == (size of queue)–1 THEN
Step 2: PRINT "Queue is overflow"
Step 3: END OF IF
Step 4: ELSE
Step 5: IF rear == (size of queue)–1 THEN
Step 6: FOR I = 0 TO i == rear – front
Step 7: queue[i] = queue[front +i]
Step 8: END OF FOR
Step 9: rear = rear – front
Step 10: front = 0
Step 11: END OF IF
Step 12: IF front == –1
Step 13: front = 0
Step 14: END OF IF
Step 15: rear = rear+1
Step 16: queue [rear] = element
Step 17: END OF ELSE

Algorithm 5.2: Algorithm for deleting information from a queue

Here, queue [ ] is the array to store the array elements
'front' is a variable used to represent the front end of the queue
'rear' is a variable to represent the rear end of the queue

Delete (queue[ ], front, rear)

Step 1: IF front == –1
Step 2: PRINT "Queue is underflow"
Step 3: END OF IF

Step 4: ELSE

Step 5: front = front + 1

Step 6: IF front > rear

Step 7: front = –1

Step 8: rear = –1

Step 9: END IF

Step 10: END OF ELSE

Program 5.1: C program to implement queue using array

```
#include<stdio.h>
#include<conio.h>
#define max 40

//Structure for creating a queue

struct queue
{
    int front,rear;
    int data[max];
};

//Function prototypes

int insert(struct queue *,int );
int delet(struct queue *);
int isfull(struct queue *);
int isempty(struct queue *);
void init(struct queue *);
void display(struct queue *);

void main()
{
struct queue *q;
int option,elem,flag;
```

```
char cont;
clrscr();
q = (struct queue *)malloc(sizeof(struct queue));
init(q);
do
    {
        printf("\n*****************");
        printf("\n1.Insertion     *");
        printf("\n2.Deletion      *");
        printf("\n3.Display       *");
        printf("\n*****************");
        printf("\nEnter your option::");
        scanf("%d",&option);
        switch(option)
    {
        case 1: printf("\nEnter the element to
                be inserted into the
                queue::");
                scanf("%d",&elem);
                flag = insert(q,elem);
                if(flag == 1)
                {
                    printf("\n%d is succesfully
                    inserted into the
                    queue",elem);
                    printf("\nAfter insertion ");
                    display(q);
                }
                else
                {
                    printf("\nInsertion is not
                    possible as the
                    queue is full");
```

```
                }
            break;

            case 2: elem = delet(q);
                    if(elem == -99)
                    printf("\nDeletion is not possible
                    as the queue is
                    empty");
                    else
                    {
                        printf("\n%d is deleted from
                        the queue");
                        printf("\nAfter deletion'');
                        display(q);
                    }
            break;

            case 3: display(q);
            break;

        default: printf("\nWrong input...try again");
        }
        printf("\nDo you want to continue...Press 'y'
        or 'Y' to continue");
            cont = getch();
        } while(cont == 'y'||cont == 'Y');
    }

    //Function to initialize a queue
    void init(struct queue *q)
    {
        q->front = -1;
        q->rear = -1;
    }
```

```
//Function to check a queue is overflow or not
int isfull(struct queue *q)
{
    if(q->front == 0 && q->rear == max-1)
        return(1);
    else
        return(0);
}
//Function to check a queue is underflow or not
int isempty(struct queue *q)
{
    if(q->front == -1)
        return(1);
    else
        return(0);
}

//Function to insert a new element into a queue
int insert(struct queue *q,int element)
{
    int i;
    if(isfull(q))
        return(0);
    else
    {
        if(q->rear == max -1)
    {
        for(i = 0; i <= (q->rear)-(q->front); i++)
        q->data[i] = q->data[qv>front+i];

    q->rear = q->rear - q->front;
    q->front = 0;
    }
```

```
            if(q->front == -1)
                q->front = 0;
            q->data[++q->rear] = element;
            return(1);
        }
    }
    //Function to delete a element from a queue
    int delet(struct queue *q)
    {
        int delement;
        if(isempty(q))
        return(-99);
            else
            {
            delement = q->data[q->front];
        q->front++;
        if(q->front > q->rear)
            {
            q->front = -1;
            q->rear = -1;
            }
            return(delement);
        }
    }

    //Function to display the elements available in
    the queue
    void display(struct queue *q)
    {
        int i;
        if(isempty(q))
            printf("\nThe queue is empty");
        else
```

```
        {
            printf("\nthe elements in the queue
            are:\n");
            for(i = q->front; i <=q->rear; i++)
            printf("%5d",q->data[i]);
        }
    }
```

5.5 CIRCULAR QUEUE

In the earlier implementation of queue, we have learnt that shifting of queue elements must be done to the left most direction of the array to make free space available at the rear end in case of insertion of new elements when 'rear' is equal to 'size of the array' –1 and 'front' is not equal to 0. So, in case of a queue with a large number of elements, the shifting of queue elements is a huge wastage of time. This is a drawback of the queue. So to remove this drawback, circular queue implementation is used. In case of circular queue implementation, if 'rear' is equal to (size of the queue) –1 and 'front' is not equal to 0 then a new element is inserted into the array at the subscript value 0^{th} position.

Let us imagine the array as a circular list. Here, if the value of 'rear' or 'front' is {(size of queue) –1} then next value of 'rear' or 'front' is 0, otherwise the next value of 'rear' or 'front' will be one more than the earlier value. In this implementation, the 'front' and the 'rear' are initialized with the value '–1'. The overflow condition is "'front' is equal to the next value of the rear's recent value" and underflow condition is "'front' is equal to '–1'". In this implementation, if 'front' is not equal to –1 and 'front' is equal to 'rear' then it means that there is only one element is available in the circular queue. But in some other implementation, if 'front' is equal to 'rear' then it means that the circular queue is empty.

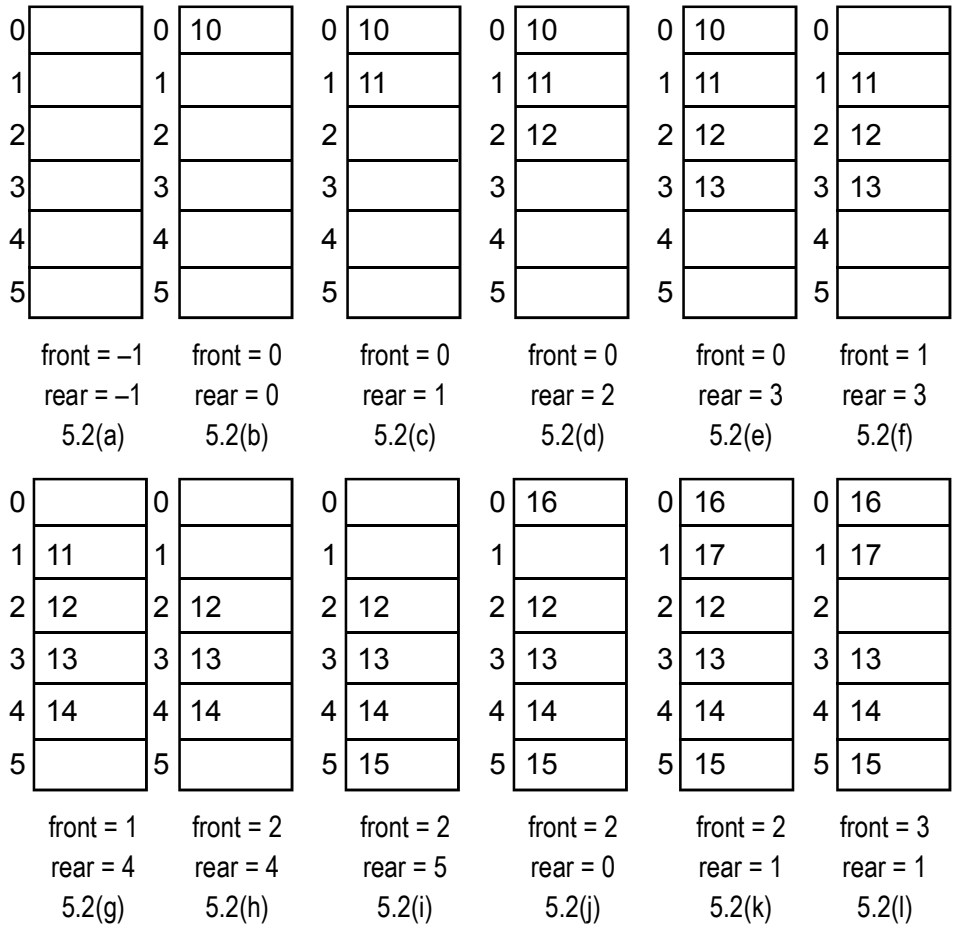


Figure 5.2: (a) Empty queue, (b) Insertion of "10", (c) Insertion of "11", (d) Insertion of "12", (e) Insertion of "13", (f) Deletion of first element, (g) Insertion of "14", (h) Deletion of next element, (i) Insertion of "15", (j) Insertion of "16", (k) Insertion of "17", (l) Deletion of next element

In figure 5.2(a), an empty circular queue is represented with front = –1 and rear = –1. In figure 5.2(b), 10 is inserted into the queue and it is the first element in the queue with front = 0 and rear = 0. Here 'front' is equal to 'rear' which means only one element is available in the circular queue. In figure 5.2(c), 11 is inserted into the queue with front = 0 and rear = 1. So like this, 12, 13, 14, 15, 16 and 17 are inserted into the circular queue in figures 5.2(d), (e), (g), (i), (j) and (k) respectively. In figure 5.2(f), 10 is deleted from the circular queue with front = 1 and rear = 3. In figure 5.2(h), 11 is deleted from the circular queue with front = 2 and rear = 4. In figure 5.2(l), 12 is deleted from the circular queue with front = 3 and rear = 1.

Algorithm 5.3: Algorithm for inserting new information into a circular queue

Here, cqueue[ ] is the array to store the queue elements.

'front' is a variable used to represent the front end of the circular queue.

'rear' is a variable to represent the rear end of the circular queue.

'element' is a variable to store the new element into the circular queue.

Insertcq (cqueue[ ], front, rear, element)

Step 1: IF front == (rear+1)% (size of cqueue)

Step 2: PRINT "Queue overflow"

Step 3: END OF IF

Step 4: ELSE

Step 5: rear = (rear+1)%(size of cqueue)

Step 6: IF front == –1

Step 7: front = 0

Step 8: END OF IF

Step 9: cqueue[rear] = element

Step 10: END OF ELSE

In this algorithm, the next value for 'front' and 'rear' can be calculated using modulus (%) operator. Modulus (%) operator gives the reminder of any division operation. When the value of 'rear' or 'front' is equal to (size of the queue) –1 then the next value of 'rear' or 'front' is equal to ((rear or front) +1)% (size of the queue) i.e., 0 otherwise it will be one more than the recent value of 'rear' or 'front'.

Algorithm 5.4: Algorithm for deleting information from a circular queue

Here, cqueue[ ] is the array to store the array elements

'front' is a variable used to represent the front end of the queue

'rear' is a variable to represent the rear end of the queue

deletecq(cqueue[ ],front,rear)

Step 1: IF front == –1

Step 2: PRINT “Queue is underflow”

Step 3: END OF IF

Step 4: ELSE

Step 5: IF front == rear)

Step 6: front = –1

Step 7: rear = –1

Step 8: END OF IF

Step 9: ELSE

Step 10: front = (front+1)%(size of cqueue)

Step 11: END OF ELSE

Step 12: END OF ELSE

Program 5.2: Program to implement circular queue

```
#include<stdio.h>
#include<conio.h>
#define max5

//Structure to create a circular queue

struct cirqueue
    {
    int front,rear;
    int data[max];
    };
typedef struct cirqueue cirqueue;

// Function prototypes

int insert(cirqueue **,int );
int delet(cirqueue **);
int isfull(cirqueue **);
int isempty(cirqueue **);
void init(cirqueue **);
void display(cirqueue *);
```

```
void main()
{
    cirqueue *q;
    int option,elem,flag;
    char cont;
    clrscr();
    q = (cirqueue *)malloc(sizeof(cirqueue));
    init(&q);
    do
    {
        printf("\n*****************");
        printf("\n1.Insertion     *");
        printf("\n2.Deletion      *");
        printf("\n3.Display       *");
        printf("\n*****************");
        printf("\nEnter your option::");
        scanf("%d",&option);
        switch(option)
        {

           case 1: printf("\nEnter the element to
                   be inserted into the queue::");
                   scanf("%d",&elem);
                   flag = insert(&q,elem);
                   if(flag == 1)
                   {
                   printf("\n%d is succesfully
                   inserted into the queue",elem);
                   printf("\nAfter insertion ");
                   display(q);
                   }
                   else
                   {
```

```
                    printf("\nInsertion isnot
                    possible as the queue is full");
                    }
                    break;
            case 2: elem = delet(&q);
                    if(elem == -99)
                    printf("\nDeletion is not
                    possible as the queue is empty");
                    else
                    {
                    printf("\n%d is deleted from the
                    circular queue",elem);
                    printf("\nAfter deletion ");
                    display(q);
                    }
                    break;
            case 3: display(q);
                    break;
        default:printf("\nWrong input...try
        again");
        }
        printf("\nDo you want to continue..Press
        'y' or 'Y' to continue");
        cont = getch();
    } while(cont == 'y' || cont == 'Y');
}

//Function to initialize a circular queue
void init(cirqueue **q)
{
    (*q)->front = -1;
    (*q)->rear = -1;
}
```

```
//Function to check a circular queue is overflow
or not
int isfull(cirqueue **q)
{
    if((*q)->front == ((*q)->rear+1)%max)
        return(1);
    else
        return(0);
}

//Function to check a circular queue is underflow
or not
int isempty(cirqueue **q)
{
    if((*q)->front == -1)
        return(1);
    else
        return(0);
}

//Function to insert a new element into the circular
queue
int insert(cirqueue **q,int element)
{
    int i;
    if(isfull(q))
    return(0);
    else
    {
        (*q)->rear = ((*q)->rear+1)%max;
        if((*q)->front == -1)
        (*q)->front = 0;
        (*q)->data[(*q)->rear] = element;
```

```
            return(1);
        }
    }

    //Function to delete an element from the circular
    queue
    int delet(cirqueue **q)
    {
        int delement;
        if(isempty(q))
            return(-99);
        else
        {
            delement = (*q)->data[(*q)->front];
            if((*q)->front == (*q)->rear)
        {
            (*q)->front = -1;
            (*q)->rear = -1;
        }
        else
            (*q)->front = ((*q)->front+1)%max;
        return(delement);
        }
    }

    //Function to display elements available in the
    circular queue
    void display(cirqueue *q)
    {
        int i;
        if(isempty(&q))
        printf("\nThe queue is empty");
        else
```

```
    {
        printf("\nthe elements in the queue
        are:\n");
        I = q->front;
        while(i != q->rear)
        {
            printf("%5d", q->data[i]);
            i = (i+1)%max;
        }
        printf("%5d", q->data[i]);
    }
}
```

5.6 LINKED LIST IMPLEMENTATION OF QUEUE

In the following section, queue is implemented with the three types of linked list.

5.6.1 Using Singly Linked List

In this implementation, queue is a singly linked list with two pointers ‘rear’ and ‘front’. The pointer ’front’ points the first element in the list and the pointer ‘rear’ points the last elements in the list. Both ‘front’ and ‘rear’ are initialized to NULL. Here, the underflow condition is ‘front’ is equal to NULL. In deletion operation, the node pointed by ‘front’ is deleted and then ‘front’ will point to the next node of the deleted node if available. In insertion operation the new node is inserted after the node pointed by rear i.e. at the last position and ‘rear’ will point the new last node in the list.

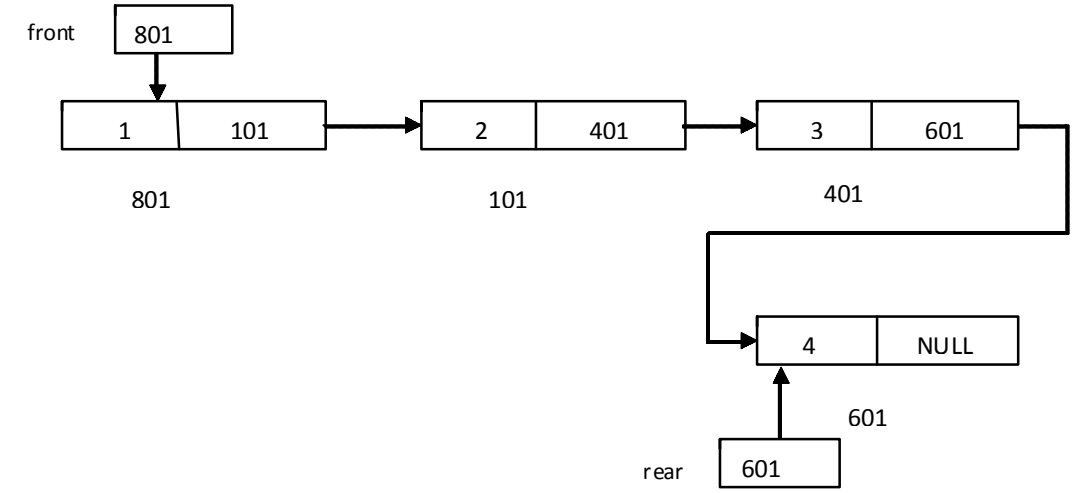


Figure 5.3: (a) Example of a queue implemented using singly linked list

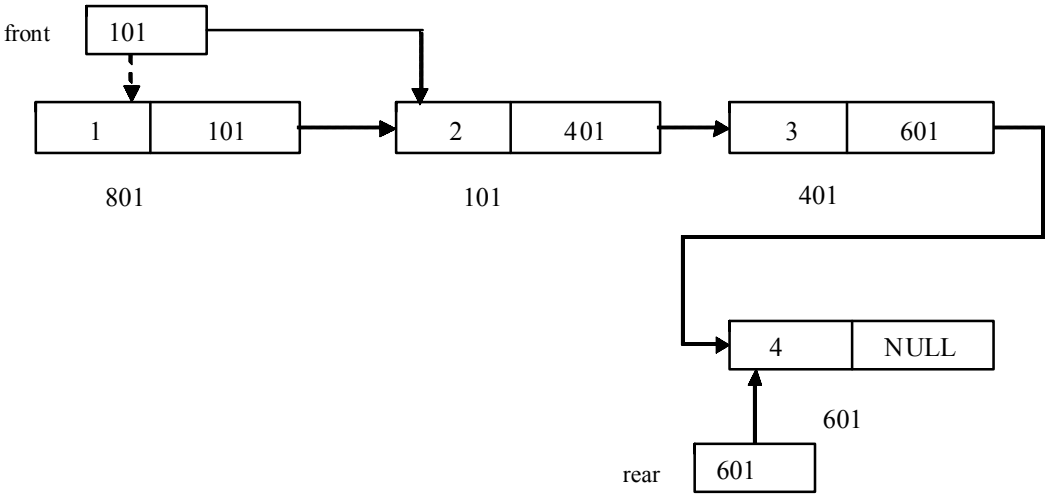


Figure 5.3: (b) Deletion of queue element

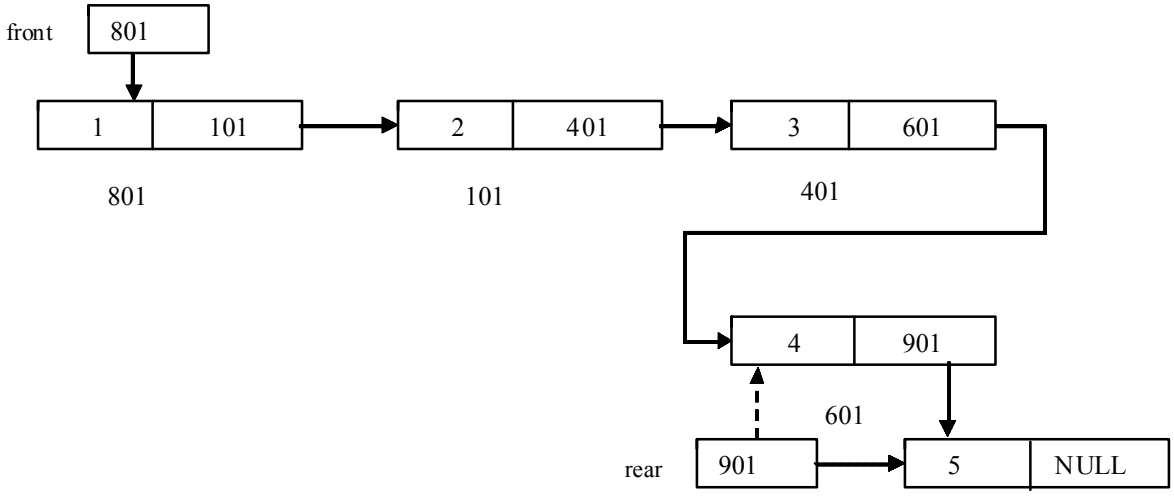


Figure 5.3: (c) Insertion of new queue element

Figure 5.3 (a) shows an example of queue. In figure 5.3(b), the node pointed by the pointer 'front' with data "1" is deleted from the queue which is implemented using singly linked list. In figure 5.3(c), a new node with data "5" is inserted into the queue which is implemented using singly linked list.

Algorithm 5.5: Algorithm for inserting new information to a queue:

ADDRESS(ptr) means address part of the node pointed by the pointer "ptr" which points the next node in the queue implemented using singly linked list.

DATA(ptr) means data part of the node pointed by the pointer "ptr".

"newnode" is the pointer which points the node to be inserted into the queue implemented using singly linked list.

'element' is a variable to store the new element to be inputted into the queue implemented using singly linked list.

Insert (front, rear, element)

Step 1: ALLOCATE MEMORY FOR newnode
Step 2: DATA(newnode) = element
Step 3: ADDRESS(newnode) = NULL
Step 4: IF front == NULL
Step 5: front = newnode
Step 6: rear=newnode
Step 7: END OF IF
Step 8: ELSE
Step 9: ADDRESS (rear) = newnode
Step 10: rear = ADDRESS (rear)
Step 11: END OF ELSE

Algorithm 5.6: Algorithm for deleting information from a queue:

ADDRESS (ptr) means address part of the node pointed by the pointer "ptr" which points the next node in the queue implemented using singly linked list.

"temp" is a pointer to point any node of a queue implemented using singly linked list.

delete(front, rear)

Step 1: IF front == NULL

Step 2: PRINT “Queue is underflow”

Step 3: END OF IF

Step 4: ELSE

Step 6: temp = front

Step 7: front = ADDRESS(front)

Step 8: IF front == NULL THEN

Step 9: rear = NULL

Step 10: END OF IF

Step 11: DEALLOCATE MEMORY FOR temp

Step 12: END OF ELSE

Program 5.3: C program to implement queue using singly linked list

```
#include<stdio.h>
#include<conio.h>

//Structure to create a node of a queue

struct lqueue
    {
    int data;
    struct lqueue*next;
    };
typedef struct lqueue lqueue;

//Function prototypes
int insert(lqueue **,lqueue **,lqueue **,int );
int delet(lqueue **,lqueue **,lqueue **);
int isempty(lqueue *);
void init(lqueue **,lqueue **,lqueue **);
void display(lqueue *);
```

```
void main()
{
    lqueue *qhead, *front, *rear;
    int option,elem,flag;
    char cont;
    clrscr();
    init(&qhead,&front,&rear);
    do
    {
        printf("\n*****************");
        printf("\n1.Insertion     *");
        printf("\n2.Deletion      *");
        printf("\n3.Display       *");
        printf("\n*****************");
        printf("\nEnter your option::");
        scanf("%d",&option);
        switch(option)
        {

        Case 1: printf("\nEnter the element to be
                inserted into
                the queue::");
                scanf("%d",&elem);
    flag = insert(&qhead,&front,&rear,elem);
        if(flag == 1)
                {
                printf("\n%d is succesfully
                inserted into the queue",elem);
                printf("\nAfter insertion'');
                display(qhead);
                }
                else
                {
```

```
                printf("\nInsertion isnot
                successfull");
                }
                break;
        Case 2: elem = delet(&qhead, &front,&rear);
                if(elem == -99)
                printf("\nDeletion is not possible
                as the queue is empty");
                else
                {
                printf("\n%d is deleted from the
                queue",elem);
                printf("\nAfter deletion'');
                display(qhead);
                }
                break;
        Case 3: display(qhead);
                break;
        default: printf("\nWrong input...try
        again");
        }
            printf("\nDo you want to continue..Press
            'y' or 'Y' to continue");
            cont = getch();
    } while(cont == 'y' || cont == 'Y');
}
//Function to initialize a singly linked list
void init(lqueue **qhead,lqueue **f,lqueue **r)
{
    *qhead = NULL;
    *f = NULL;
    *r = NULL;
}
```

```
//Function to check a queue is underflow or not
int isempty(lqueue *qhead)
{
    if(qhead == NULL)
        return(1);
    else
        return(0);
}

//Function to insert a new node into a queue
int insert(lqueue **qhead,lqueue **f,lqueue **r,int
element)
{
    lqueue *newnode;
    newnode = (lqueue *)malloc(sizeof(lqueue));
    if(newnode == NULL)
        return(0);
    newnode->next = NULL;
    newnode->data = element;
    if(*qhead == NULL)
    {
        *f = newnode;
        *r = newnode;
        *qhead = newnode;
    }
    else
    {
        (*r)->next = newnode;
        (*r) = (*r)->next;
    }
    return(1);
}
//Function to delete a node from a queue
```

```
int delet(lqueue **qhead,lqueue **f,lqueue **r)
{
    int delement;
    lqueue *temp;
    if(isempty(*qhead))
        return(-99);
    else
    {
        temp = *f;
        delement = temp->data;
        (*f) = (*f)->next;
        if(*f == NULL)
        *r = NULL;
        *qhead = *f;
        free(temp);
        return(delement);
    }
}

//Function to display all the elements available
in the queue
void display(lqueue *qhead)
{
    lqueue *temp;
    int i;
    temp = qhead;
    if(isempty(qhead))
        printf("\nThe queue is empty");
    else
    {
        printf("\nthe elements in the queue
        are:\n");
        while(temp != NULL)
        {
```

```
            printf("%5d",temp->data);
            temp = temp->next;
            }
        }
}
```

5.6.2 Using Doubly Linked List

In this implementation queue is a doubly linked list with two pointers 'rear' and 'front'. The pointer 'front' points the first element in the list and the pointer 'rear' points the last elements in the list. Both 'front' and 'rear' are initialized to NULL. Here, the underflow condition is 'front' is equal to NULL. In deletion operation, the node pointed by 'front' is deleted and then 'front' will point to the next node of the deleted node if available. In insertion operation the new node is inserted after the node pointed by rear i.e. at the last position and 'rear' will point the new last node in the list.

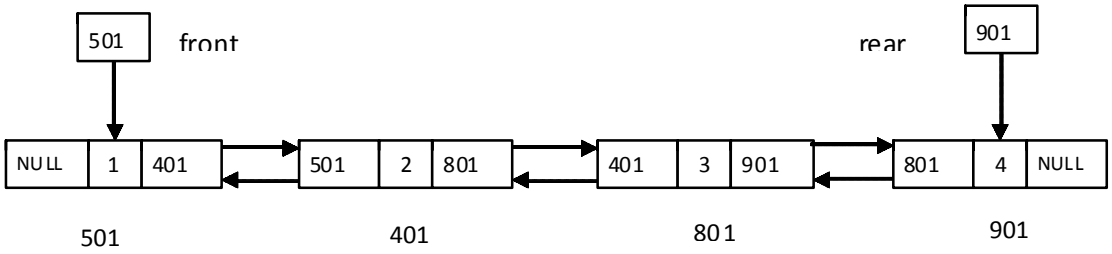


Figure 5.4: (a) Example of a queue implemented using doubly linked list

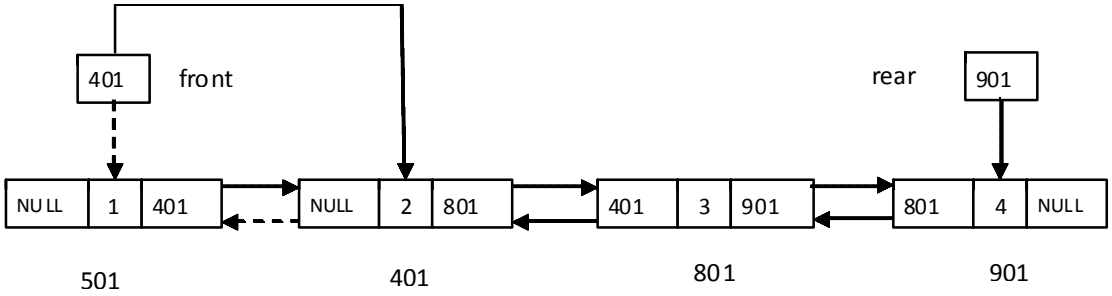


Figure 5.4: (b) Deletion of queue element

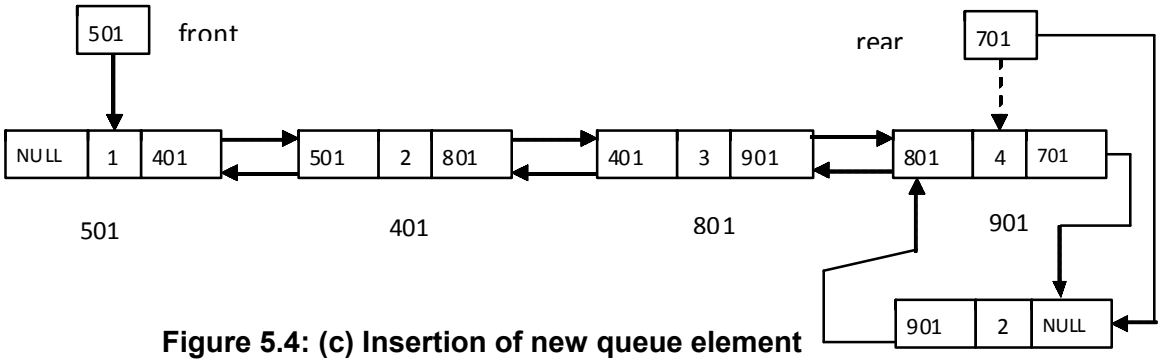


Figure 5.4: (c) Insertion of new queue element

Figure 5.4 (a) shows an example of queue. In figure 5.4(b) 701 the node pointed by the pointer 'front' with data "1" is deleted from the queue which is implemented using doubly linked list. In figure 5.4(c), a new node with data "2" is inserted into the queue which is implemented using doubly linked list.

Algorithm 5.7: Algorithm for inserting new information from a queue

ADDRESSNEXT(ptr) means the address part of a node pointed by the pointer "ptr" which points the next node in the queue implemented using doubly linked list.

ADDRESSPREVIOUS(ptr) means the address part of a node pointed by the pointer "ptr" which points the previous node in the queue implemented using doubly linked list.

DATA(ptr) means the data part of a node pointed by the pointer "ptr" of a queue implemented using doubly linked list.

'element' is a variable to store the new element to be inputted into the queue implemented using doubly linked list.

insert(front, rear, element)

Step 1: ALLOCATE MEMORY FOR newnode

Step 2: DATA(newnode) = element

Step 3: ADDRESSNEXT(newnode) = NULL

Step 4: ADDRESSPREVIOUS(newnode) = NULL

Step 5: IF front == NULL THEN

Step 6: front = newnode

Step 7: rear = newnode

Step 8: END OF IF

Step 9: ELSE

Step 10: ADDRESSPREVIOUS(newnode) = rear

Step 11: ADDRESSNEXT(rear) = newnode

Step 12: rear = ADDRESSNEXT(rear)

Step 13: END OF ELSE

Algorithm 5.8: Algorithm for deleting information from a queue

ADDRESSNEXT(ptr) means the address part of a node pointed by the pointer "ptr" which points the next node in the queue implemented using doubly linked list.

ADDRESSPREVIOUS(ptr) means the address part of a node pointed by the pointer "ptr" which points the previous node in the queue implemented using doubly linked list.

"temp" is a pointer to point any node of a queue implemented using doubly linked list.

delete(front, rear)

Step 1: IF front == NULL THEN

Step 2: PRINT "Queue is underflow"

Step 3: END OF IF

Step 4: ELSE

Step 5: temp = front

Step 6: front = ADDRESSNEXT(front)

Step 7: IF front == NULL

Step 8: drear = NULL

Step 9: END OF IF

Step 10: ELSE

Step 11: ADDRESSPREVIOUS(front) = NULL

Step 12: END OF ELSE

Step 13: DEALLOCATE MEMORY FOR temp

Step 14: END OF ELSE

Program 5.4: C program to implement queue using doubly linked list

```
#include<stdio.h>
#include<conio.h>

//Structure to create a node of a doubly linked
list
struct dlqueue
    {
        int data;
        struct dlqueue *prev;
        struct dlqueue *next;
    };
typedef struct dlqueue dlqueue;

//Function prototypes
int insert(dlqueue **,dlqueue **,dlqueue **,int);
int delet(dlqueue **,dlqueue **,dlqueue **);
int isempty(dlqueue *);
void init(dlqueue **,dlqueue **,dlqueue **);
void display(dlqueue *);

void main()
{
    dlqueue *qhead, *front, *rear;
    int option,elem,flag;
    char cont;
    clrscr();
    init(&qhead,&front,&rear);
    do
    {
        printf("\n*****************");
        printf("\n1.Insertion     *");
```

```
        printf("\n2.Deletion      *");
        printf("\n3.Display       *");
        printf("\n*****************");
        printf("\nEnter your option::");
        scanf("%d",&option);
        switch(option)
        {
        Case 1: printf("\nEnter the element to be
        inserted into the queue::");
                scanf("%d",&elem);
    flag = insert(&qhead,&front,&rear,elem);
        if(flag == 1)
                {
                printf("\n%d is succesfully
                inserted into the queue",elem);
                printf("\nAfter insertion'');
                display(qhead);
                }
                else
                {
                printf("\nInsertion isnot
                successfull");
                }
                break;
        Case 2: elem = delet(&qhead,&front,&rear);
                if(elem == -99)
                printf("\nDeletion is not possible
                as the queue is empty");
                else
                {
                printf("\n%d is deleted from the
                queue",elem);
                printf("\nAfter deletion ");
```

```
                display(qhead);
                }
                break;
        Case 3: display(qhead);
                break;
        default:printf("\nWrong input...try
        again");
        }
        printf("\nDo you want to continue...Press
        'y' or 'Y' to continue");
        cont = getch();
    } while(cont == 'y' || cont == 'Y');
}

//Function to initialize a doubly linked list
void init(dlqueue **qhead,dlqueue **f,dlqueue **r)
{
    *qhead = NULL;
    *f = NULL;
    *r = NULL;
}

//Function to check a queue is underflow or not
int isempty(dlqueue *qhead)
{
    if(qhead == NULL)
        return(1);
    else
        return(0);
}

//Function to insert a new node into a queue
int insert(dlqueue **qhead,dlqueue **f,dlqueue **r,
int element)
```

```
{
    dlqueue *newnode;
    newnode = (dlqueue *)malloc(sizeof(dlqueue));
    if(newnode == NULL)
        return(0);
    newnode->next = NULL;
    newnode->prev = NULL;
    newnode->data = element;
    if(*qhead == NULL)
    {
        *f = newnode;
        *r = newnode;
        *qhead = newnode;
    }
    else
    {
        newnode->prev = *r;
        (*r)->next = newnode;
        (*r) = (*r)->next;
    }
    return(1);
}

//Function to delete a node from a queue
int delet(dlqueue **qhead,dlqueue **f,dlqueue **r)
{
    int delement;
    dlqueue *temp;
    if(isempty(*qhead))
        return(-99);
    else
    {
        temp = *f;
```

```
            delement = temp->data;
            (*f) = (*f)->next;
            if(*f == NULL)
              *r = NULL;
        else
            (*f)->prev = NULL;

            *qhead = *f;
            free(temp);
            return(delement);
        }
    }

    //Function to display all the elements available
    in a queue
    void display(dlqueue*qhead)
    {
        dlqueue *temp;
        int i;
        temp = qhead;
        if(isempty(qhead))
            printf("\nThe queue is empty");
        else
        {
            printf("\nthe elements in the queue
            are:\n");
            while(temp != NULL)
            {
                printf("%5d",temp->data);
                temp = temp->next;
            }
        }
    }
```

5.6.3 Using Circular Linked List

In this implementation, queue is a circular linked list with two pointers 'rear' and 'front'. The pointer 'front' points the first element in the list and the pointer 'rear' points the last elements in the list. Both 'front' and 'rear' are initialized to NULL. The underflow condition is same with the singly linked list implementation. Here, the address field of the node which is pointed by 'rear' points the node which is pointed by 'front'.

In deletion operation, the node pointed by 'front' is deleted and then 'front' will point to the next node of the deleted node if available otherwise 'front' and 'rear' will be NULL. The address field of the node pointed by 'rear' will point the node which is pointed by 'front' after deletion operation. In insertion operation the new node is inserted after the node pointed by 'rear' i.e. at the last position and then 'rear' will point the new last node in the list. The address part of the new node will point the node pointed by front.

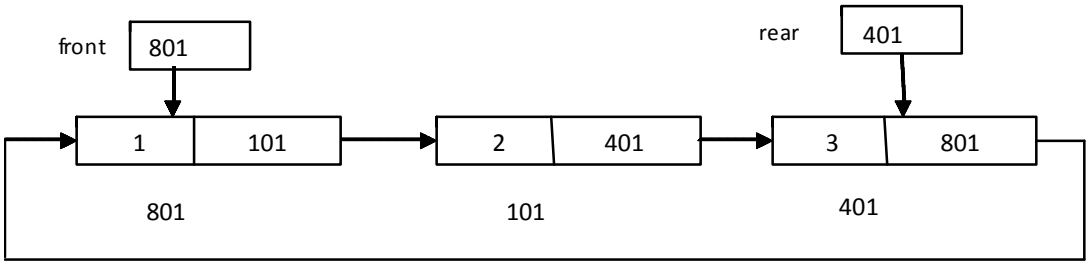


Figure 5.5: (a) Example of a queue implemented using circular linked list

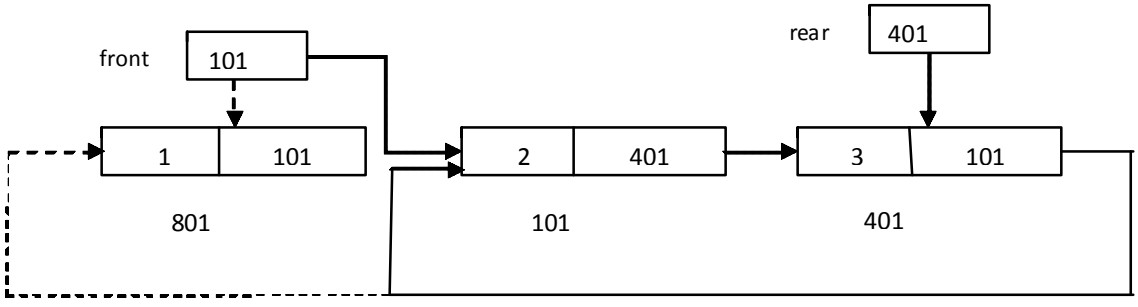


Figure 5.5: (b) Deletion of queue element

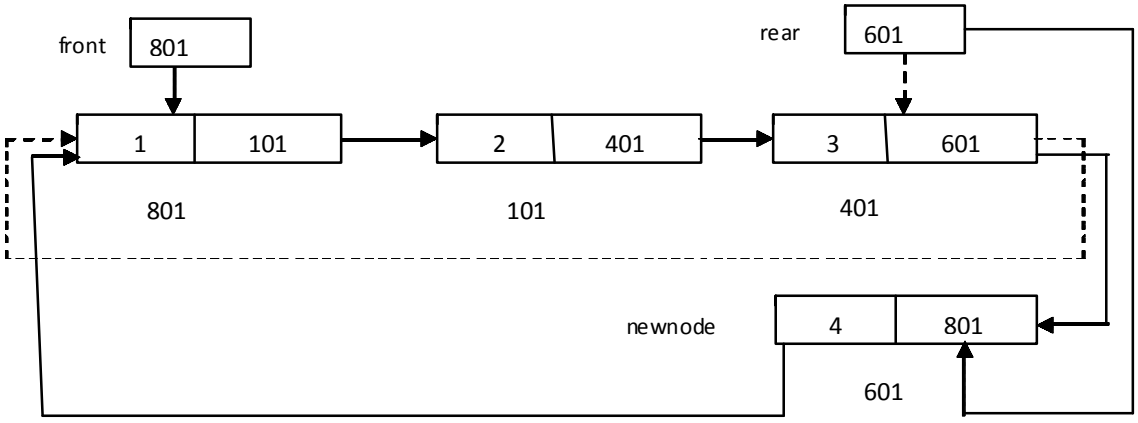


Figure 5.5: (c) Insertion of new queue element

Figure 5.5 (a) shows an example of queue. In figure 5.5(b), the node pointed by the pointer 'front' with data "1" is deleted from the queue which is implemented using circular linked list. In figure 5.5(c), a new node with data "4" is inserted into the queue which is implemented using circular linked list.

Algorithm 5.9: Algorithm for inserting new information from a queue

ADDRESS(ptr) means address part of the node pointed by the pointer "ptr" which points the next node in the queue implemented using circular linked list.

DATA(ptr) means data part of the node pointed by the pointer "ptr".

"newnode" is the pointer which points the node to be inserted into the queue implemented using circular linked list.

'element' is a variable to store the new element to be inputted into the queue implemented using circular linked list.

insert(front, rear, element)

Step 1: ALLOCATE MEMORY FOR newnode

Step 2: DATA(newnode) = element

Step 3: ADDRESS(newnode) = NULL

Step 4: IF front == NULL THEN

Step 5: front = newnode

Step 6: rear = newnode

Step 7: ADDRESS(rear) = front

Step 8: END OF IF

Step 9: ELSE

Step 10: ADDRESS(newnode) = front

Step 11: ADDRESS(rear) = newnode

Step 12: rear = ADDRESS(rear)

Step 13: END OF ELSE

Algorithm 5.10: Algorithm for deleting information from a queue

ADDRESS(ptr) means address part of the node pointed by the pointer "ptr" which points the next node in the queue implemented using circular linked list.

"temp" is a pointer to point any node of a queue implemented using circular linked list.

delete(front, rear)

int delement;

Step 1: IF front == NULL THEN

Step 2: PRINT"Queue underflow"

Step 3: END OF IF

Step 4: ELSE

Step 5: temp = front

Step 6: IF ADDRESS(front) == front THEN

Step 7: rear = NULL

Step 8: front = NULL

Step 9: END OF IF

Step 10: ELSE

Step 11: front = ADDRESS(front)

Step 12: ADDRESS(rear) = front

Step 13: END OF ELSE

Step 14: DEALLOCATE MEMORY FOR temp

Step 15: END OF ELSE

Program 5.5: C program to implement queue using circular linked list

```
#include<stdio.h>
#include<conio.h>

//Structure to create a node of a queue
struct crqueue
    {
    int data;
    struct crqueue*next;
    };
typedef struct crqueue crqueue;

//Function prototypes
int insert(crqueue **,crqueue **,crqueue **,int);
int delet(crqueue **,crqueue **,crqueue **);
int isempty(crqueue *);
void init(crqueue **,crqueue **,crqueue **);
void display(crqueue *);

void main()
{
    crqueue*qhead,*front,*rear;
    int option,elem,flag;
    char cont;
    clrscr();
    init(&qhead,&front,&rear);
    do
    {
        printf("\n*****************");
        printf("\n1.Insertion     *");
        printf("\n2.Deletion      *");
        printf("\n3.Display       *");
```

```
            printf("\n*****************");
            printf("\nEnter your option::");
            scanf("%d",&option);
            switch(option)
            {
                case 1: printf("\nEnter the element to
                        be inserted into the queue::");
                        scanf("%d",&elem);
        flag = insert(&qhead,&front,&rear,elem);
                        if(flag == 1)
                        {
                        printf("\n%d is succesfully
                        inserted into the queue",elem);
                        printf("\nAfter insertion'');
                        display(qhead);
                        }
                        else
                        {
                        printf("\nInsertion isnot
                        successfull");
                        }
                        break;
                case 2: elem = delet(&qhead,&front,
                        &rear);
                        if(elem == -99)
                        printf("\nDeletion is not
                        possible as the queue is empty");
                        else
                        {
                        printf("\n%d is deleted from
                        the queue",elem);
                        printf("\nAfter deletion'');
                            display(qhead);
```

```
                    }
                    break;
            case 3: display(qhead);
                    break;
        default:printf("\nWrong input...try
        again");
    }
        printf("\nDo you want to continue.
        Press 'y' or 'Y' to continue");
        cont = getch();
    } while(cont == 'y' || cont == 'Y');
}

//Function to initialize a circular linked list
void init(crqueue **qhead,crqueue **f,crqueue **r)
{
    *qhead = NULL;
    *f = NULL;
    *r = NULL;
}

//Function to check a queue is underflow or not
int isempty(crqueue *qhead)
{
    if(qhead == NULL)
        return(1);
    else
        return(0);
}

//Function to insert a new node into the queue
int insert(crqueue **qhead,crqueue **f,crqueue **r,
int element)
```

```
{
    crqueue *newnode;
    newnode = (crqueue *)malloc(sizeof(crqueue));
    if(newnode == NULL)
        return(0);
    newnode->data = element;
    if(*qhead == NULL)
    {
        *f = newnode;
        *r = newnode;
        (*r)->next = *f;
        *qhead = newnode;
    }
    else
    {
        newnode->next = *f;
        (*r)->next = newnode;
        (*r) = (*r)->next;
    }
        return(1);
}

//Function to delete a node from the queue
int delet(crqueue **qhead,crqueue **f,crqueue **r)
{
    int delement;
    crqueue *temp;
    if(isempty(*qhead))
        return(-99);
    else
    {
        temp = *f;
        delement = temp->data;
```

```
            (*f) = (*f)->next;

                if((*f)->next == *f)
                {
                *r = NULL;
                *f = NULL;
                }
            else
                (*r)->next = *f;

            *qhead = *f;
            free(temp);
            return(delement);
        }
    }

    //Function to display all the elements available in
    a queue
    void display(crqueue *qhead)
    {
        crqueue *temp;
        int i;
        temp = qhead;
        if(isempty(qhead))
            printf("\nThe queue is empty");
        else
        {
            printf("\nthe elements in the queue
            are:\n");
            printf("%5d",temp->data);
            temp = temp->next;
            while(temp != qhead)
                {
```

```
                printf("%5d",temp->data);
                temp = temp->next;
            }
        }
    }
```

5.7 PRIORITY QUEUES

Priority queue is a type of queue where each element has a priority value and the deletion of the elements is dependent upon the priority value. In case of max-priority queue, the highest priority value will be deleted first and in case of min-priority queue the element which has the minimum priority value will be deleted first.

One application of max-priority queue is to schedule jobs on a shared computer. A min-priority queue can be used in an even-driven simulator. We will learn about priority quenes in detail in Unit 13.

5.8 APPLICATION OF QUEUE

There are several applications of queue available in computer system. Some of these are given as follows:

- In printers, queue is used to print the different files.
- Queue is used to access files from a disk system.
- In a multiprogramming environment, queue is used for CPU scheduling or job scheduling of operating system.
- In any type of ticket reservation system, queue can be used for issuing tickets to the customers.
- Queue is used in the implementation of breadth first traversal of graph.

Queue is used in many other real world systems which are used in some scientific research, military operations etc.



CHECK YOUR PROGRESS

Q.1: Multiple choice questions:

I) "FRONT==REAR" pointer refers to–

A) Empty stack B) Empty queue

C) Full stack D) Full queue

II) Last in last out–

A) Array B) Stack

C) Queue D) None of the above

III) Queue cannot be used for–

A) the line printer B) access to disk storage

C) function call D) Both A and B

IV) Insertion into an queue is done at the–

A) Front end B) Rear end

C) Top end D) Both A and C

V) In case of a circular queue, if rear is 10 and front is 20 and the maximum number of element can be stored is N then at the current situation the total number of element stored in the queue is–

A) N–9 B) N–19

C) N–20 D) None of the above

Q.2: Fill in the blanks:

I) ..... is used to implement Breadth first search algorithm.

II) Deletion is done at ..... end in a queue.

III) At rear end ..... is done in case of a queue.

IV) If in a circular queue, front==(rear+1)%(size of the queue) then it means that .....

V) Printing files is an application of .....

Q.3: State whether the following statements are true or false:

I) Reversing strings can be done using queue

II) Conversion of infix notation to postfix notation can be done using queue.

III) Circular Queue can be implemented using circular linked list.

IV) Queue is also known as FIFO.

V) Queue is used in job scheduling of operating system.



5.9 LET US SUM UP

- Queue is a linear data structure in which the element will be removed first which is inserted first. Queue is also known as first in first out (FIFO) data structure.
- A queue has two ends which are front end and rear end. Insertion is to take place at rear end and deletion is to take place at front end.
- Queue can be implemented using both array and linked list.
- Circular queue is a type of array implementation of queue where the array is imagined as a circular data structure i.e. the next subscript value of the last subscript value of the array is '0'.
- Printing files, accessing files from a disk system, job scheduling by operating system etc. are some of the applications of queue.
- Priority queue is a type of queue where the deletion of elements depends upon some priority value associated with each element in the queue.
- In case of max-priority queue, the element that has the highest priority value will be deleted first and in case of min-priority queue the element that has the lowest priority value will be deleted first.



5.10 FURTHER READING

1) Horowitz, E., Sahni, S. & Anderson-Freed, S. (1983). *Fundamentals of Data Structures*. (Vol. 20). Rockville, MD: Computer Science Press.

2) Langsam, Y., Augenstein, M. & Tenenbaum, A. M. (1996). *Data Structures Using C and C++*. (Vol. 2). New Jersey: Prentice Hall.



5.11 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: I) (B), II) (C), III) (C), IV) (B), V) (A)

Ans. to Q. No. 2: I) Queue, II) Front, III) Insertion,
IV) The queue is overflow, V) queue

Ans. to Q. No. 3: I) False, II) False, III) False, IV) True, V) True.



5.12 MODEL QUESTIONS

Q.1: Define queue.

Q.2: What are the different types of queue?

Q.3: Why is circular queue needed? Implement circular queue in C.

Q.4: Write a C program to implement queue using circular linked list.

Q.5: Write down algorithms for insert and delete operation on a queue.

Q.6: Give two applications of queue.

Q.7: Where are priority queues used in real life situations?

*** ***** ***

UNIT 6: SEARCHING

UNIT STRUCTURE

6.1 Learning Objectives

6.2 Introduction

6.3 Searching

6.4 Types of Searching

6.4.1 Linear Search

6.4.2 Binary Search

6.5 Let Us Sum Up

6.6 Further Reading

6.7 Answers to Check Your Progress

6.8 Model Questions

6.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- describe searching techniques
- explain linear seach and binary search
- search an element in both sorted and unsorted list
- compare the advantages and disadvantages of linear and binary search techniques.

6.2 INTRODUCTION

In our day-to-day life there are various applications in which the process of searching is to be carried out. Searching a name of a person from the given list, searching a specific card from the set of cards, etc., are a few examples of searching.

In computer science, there are many applications of searching process. Many advanced algorithms and data structures have been devised for the sole purpose of making searches more efficient. As the data sets become larger and larger, good search algorithms will become more important. While solving a problem, a programmer may need to search a

value in an array. This unit will focus on different searching tehniques like linear searching and binary searching. In the next unit, we will focus on different sorting techniques.

6.3 SEARCHING

Searching is a technique of finding an element from a given data list or set of elements. To illustrate the search process, let us consider an array of 20 elements. These data elements are stored in successive memory locations. We need to search a particular element from the array. Let **e** be the element to be searched. The element **e** is compared with all the elements in the array starting from the first element till the last element. When the exact match is found then the search process is terminated. In case no such element exists in the array, the process of searching should be abandoned. Suppose, we are to search the element 8 in the array. In that case the given element is present in the array and thus the search process is said to be successful as per Figure 6.1(a).

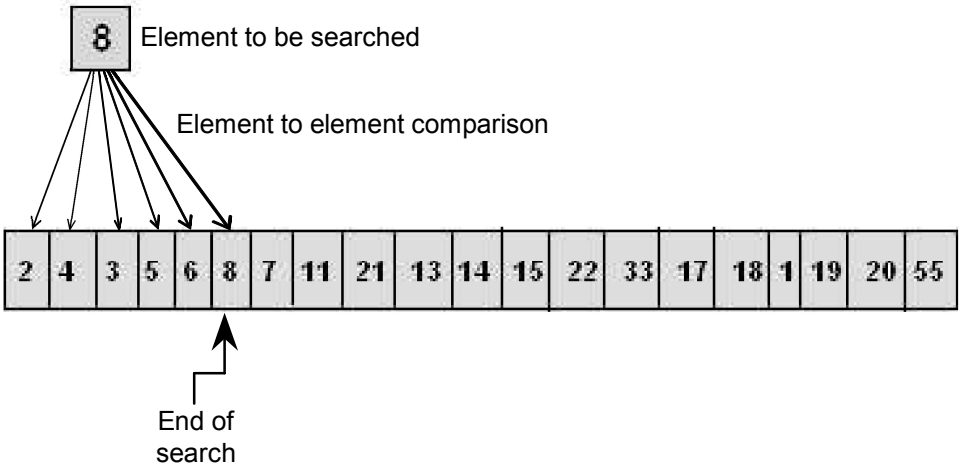


Figure 6.1(a): Successful search

Again let us consider our element to be searched is 66. The search is said to be unsuccessful as the given element does not exist in the array as per Figure 6.1(b).

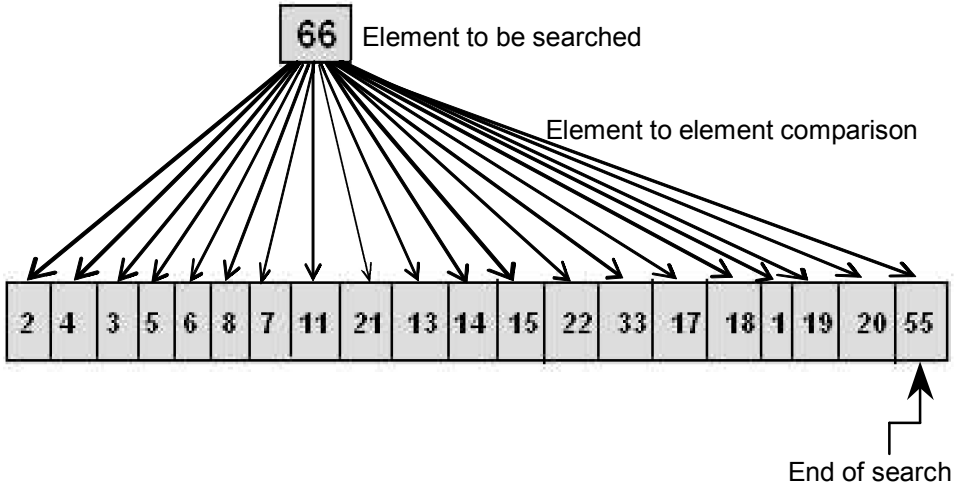


Figure 6.1(b): Unuccessful search

6.4 TYPES OF SEARCHING

The two most important searching techniques are:

- Linear or Sequential search
- Binary search

6.4.1 Linear Search

Linear search, also known as *sequential search*, is a technique in which the array 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.

Linear search is one of the simplest searching techniques. Though, it is simple and straightforward, it has some limitations. It consumes more time and reduces the retrieval rate of the system. The linear or sequential name implies that the items are stored in systematic manner. It can be applied on sorted or unsorted linear data structure.

Algorithm 6.1: Algorithm of Linear Search

Let us start with an array or list, L which may have the item in question.

Step 1: If the list L is empty, then the list has nothing. The list does not have the item in question. Stop here.

Step 2: Otherwise, look at all the elements in the list L.

Step 3: For each element:

If the element equals the item in question, the list conains the item in question. Stop here. Otherwise, go onto next element.

Step 4: The list does not have the item in question.

Program 6.1: Program to search an element in an array applying linear search

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int arr[50],n,i,item;
    printf(''How many elements you want to enter
    in the array :'');
    scanf(''%d'',&n);
        for(i=0; i<n; i++)
        {
        printf(''\nEnter element%d :'', i+1);
        scanf(''%d'',&arr[i]);
        }
    printf(''\nEnter the element to be searched :'');
    scanf(''%d'',&item);
    for(i=0; i<n; i++)
    {
        // searched item is compared with array element
        if(item == arr[i])
        {
            printf(''\n%d found at position%d\n'',
            item, i+1);
```

```
                break;
            }
        }
        if(i == n)
            printf(''\nItem%d not found in array\n'',
            item);
        getch();
}
```

In the above program suppose the user enters few numbers using the first *for* loop (Number of elements should be less or equal to 50 as the array size is 50). The element which is to be searched is stored in the variable *item*. By using a second *for* loop the element is compared with each element of the array. If the element in *item* variable is matched with any of the element in the array then the location is displayed. Otherwise *item* is not present in the array.

Analysis of Linear Search: We have carried out linear search on lists implemented as arrays. Whether the linear search is carried out on lists implemented as arrays or linked list or on files, the criteria part in performance is the comparison loops (i.e., **Step 3** of the algorithm). Obviously the fewer the number of comparisons, the sooner the algorithm will terminate.

The fewest possible comparisons is equal to **1** when the required item is the first item in the list and which will be the *best case*. Thus, in this case, the complexity of the algorithm is **O(1)**. The maximum comparisons will be equal to **n** (total numbers of elements in the list) when the required item is the last in the list or not present in the list. This will be the worst case. In both cases, the average complexity of linear search is **O(n)**.

If the required item is in position **i** in the list, then only **i** comparisons are required. Hence, in average case, the number of comparisons done by linear search will be:

$$\frac{1+2+3+.....+i+.....+N}{N}$$
$$=\frac{N(N+1)}{2\times N}$$
$$=\frac{N+1}{2}$$



CHECK YOUR PROGRESS

Q1: Select the appropriate option for each of the following questions:

i) Linear search is efficient in case of–

a) short list of data b) long list of data

c) both (a) and (b) d) none of these

ii) The process of finding a particular record is called–

a) indexing b) searching

c) sorting d) none of these

6.4.2 Binary Search

The binary search approach is different from the linear search. The binary search technique is used to search for a particular element in a sorted array or list. In this technique, two partitions of lists are made and then the given element is searched and hence, it is known as binary search.

Let us consider a list which is sorted in ascending order. It would work to search from the beginning until the *item* is found or the end is reached, but it makes more sense to remove as much of the working data set as possible so that the item is found more quickly. If we started at the middle of the list we could determine which half the *item* is in (because the list is sorted). This effectively divides the working range in half with a single test. By repeating the procedure, the result is a highly efficient search algorithm called *binary search*.

The binary search is based on the **divide-and-conquer** approach. In this technique, the element to be searched (say, *item*)

is compared with the middle element of the array. If *item* is equal to the middle element, then search is successful. If the *item* is smaller than the middle element, then *item* is searched in the segment of the array before the the middle element. However, if the *item* is greater than the middle element, *item* is searched in the array segment after the middle element. This process will be in iteration until the element is found or the array segment is reduced to a single element that is not equal to *item*.

To illustrate the process of binary search, let us assume a sorted array of 11 elements as shown in figure 6.2(a). Suppose we want to search the element 56 from the array of elements.

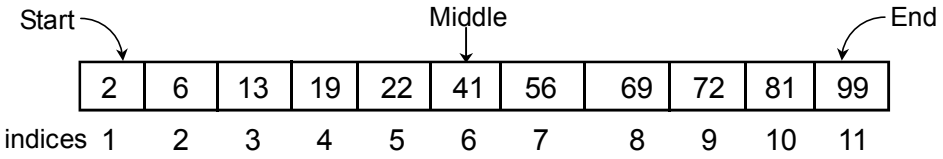


Figure 6.2(a): Sorted array of elements

For this we will take 3 variables *Start*, *End* and *Middle*, which will keep track of the status of start, end and middle value of the portion of the array, in which we will search the element. The value of the middle will be as :

$$\text{Middle} = \frac{\text{Start} + \text{End}}{2}$$

Initially, *Start* =1, *End* = 11 and the *Middle* = *(1 + 11)/2 = 6*. The value at index 6 is 41 and it is smaller than the target value i.e., (56). The steps are as follows:

Step 1: The element 2 and 99 are at Start and End positions respectively.

Step 2: Calculate the middle index which is as:

Middle = (Start + End)/2

Middle = (1+11)/2

Middle = 6

Step 3: The key element 56 is to be compared with the *Middle* value. If the key is less than the value at *Middle* then the

key element is present in the first half else in other half; in our case, the key is on the right half. Now we have to search only on right half.

Hence, Now *Start = Middle + 1 = 6 + 1 = 7.*

End will be same as earlier.

Step 4: Calculate the middle index of the second half

Middle = (Start + End)/2

Middle = (7 + 11)/2

Middle = 9

Step 5: Again, the *Middle* divides the second half into the two parts, which is shown in figure 6.2(b).

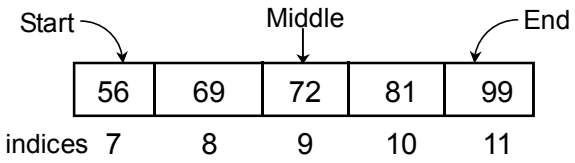


Figure 6.2(b): After divining array in middle

Step 6: The key element 56 is lesser than the value at *Middle* which is 72, hence it is present in the left half i.e., towards the left of 72. Hence now, *End = Middle – 1 = 8. Start* will remain 7.

Step 7: At last, the *Middle* is calculated as:

Middle = (Start + End)/2

Middle = (7 + 8)/2

Middle = 7

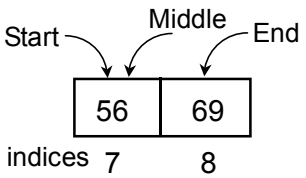


Figure 6.2(c): After further division of array in middle

Step 8: Now if we compare our key element 56 with the value at *Middle*, then we see that they are equal. The key element is searched successfully and it is in 7th location.

Algorithm 6.2: Binary Search Algorithm

Input: (List, Key)

Step 1: WHILE (List is not empty) DO
(Select "middle" entry in list as test entry

Step 2: IF (Key = test entry) THEN (Output ("Found it") Stop)

Step 3: IF (Key < test entry) THEN (apply BinarySearch to part of list preceding test entry)

Step 4: IF (Key > test entry) THEN (apply BinarySearch to part of list following test entry)

Output: ("Not Found")

Analysis of Binary Search: The binary search algorithm reduces the array to one-half in each iteration. Therefore, for an array containing **n** elements, there will be $\log_2 n$ iterations. Thus, the complexity of binary search algorithm 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.

Program 6.2: Program to search an element using binary search

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int arr[20],start,end,middle,n,i,item;
    clrscr();
    printf(''How many elements you want to enter
    in the array:'');
    scanf(''%d'',&n);
    for(i=1;i<=n;i++)  {
        printf(''Enter element%d:'', i);
        scanf(''%d'',&arr[i]);
    }
    printf(''\nEnter the element to be searched:'');
    scanf(''%d'',&item);//element to be searched
    is stored in variable item
```

```
    start = 1;
    end = n;
    middle = (start + end)/2;
    while(item!=arr[middle] && start<=end)
    {
        if(item>arr[middle])
        start = middle+1;
        else
        end = middle-1;
        middle = (start+end)/2;
    }
    if(item==arr[middle])
        printf(''\n%d is found at position%d\n'',
        item,middle);
    if(start>end)
        printf(''\n%d is not found in array\n'',
        item);
    getch();
}
```

Program 6.3: Program to search a name from a list of 10 names using binary search

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
    int start=1, end=10, mid, i, flag=0, value;
    char name[15][10],nm[15];
    clrscr();
    printf(''\nEnter 10 names:\n'');
    for(i=1; i<11; i++)
        scanf(''%s'', & name[i]);
```

```
    printf(''\nEnter the name to search:'');
    scanf(''%s'', & nm);
    mid = (start+end)/2;
    while(strcmp(nm,name[mid])!=0 && start<=end)
    {
        value=strcmp(nm,name[mid]);
        if(value>0)
        {
            start=mid+1;
            mid=(start+end)/2;
        }
        else
        {
            end=mid-1;
            mid=(start+end)/2;
        }
    }
        if(strcmp(nm,name[mid])==0)
    {
        flag=1;
    }
    if(flag==1)
        printf(''\nThe name %s is found
        successfully'',nm);
    else
        printf(''\nThe name%s is not found'',nm);
    getch();
}
```

Advantages and Disadvantages: Linear search algorithm is easy to write and efficient for short lists. It does not require sorted data. However, it is lengthy and time consuming for long lists. There is no way of quickly establishing that the required item is not in the list or of finding all occurances of a required item at one place. The linear

search situation will be in worst case if the element is at the end of the list.

A **binary search** halves the number of items to check with each iteration, so locating an item (or determining its absence) takes logarithmic time. The advantage of a binary search over a linear search is astounding for large numbers. In binary search, array/list of elements should be in shorted order.

For an array of a million elements, binary search, O(logn), will find the target element with a worst case of only 20 comparisons. Linear search, O(n), on average will take 5,00,000 comparisons to find the element.



CHECK YOUR PROGRESS

Q.2: What are the advantages of sequential search?

Q.3: State whether the following statements are true (T) or false (F):

i) Element should be in sorted order in case of binary search.

ii) Binary search cannot be applied in character array.

iii) Sequential search is worst case and average case O(n).

iv) Binary search is worst case O(logn).

Q.4: In the following function code which type of search algorithm is applied?

```
int find (int a[ ], int n, int x)
{
    int i;
    for (i = 0; i<n; i++)
    {
        if (a[i] == x )
            return i;
    }
    return 0;
}
```

Q.5: Write a function in C to find an element **x** in an array of **n** elements where the array, size of the array and the element is passed as arguments.



6.5 LET US SUM UP

- The process of finding the occurance of a particular data item in a list is known as **searching**.
- If the search element is present in the collected elements or array then the search process is said to be successful. The search is said to be unsuccessful if the given element does not exist in the array.
- Linear search is a search algorithm that tries to find a certain value in a set of data. It operates by checking every element of a list (or array) one at a time in sequence until a match is found.
- The linear search was seen to be easy to implement and relatively efficient to use for small lists. But very time consuming for long unsorted lists.
- Binary search only works on sorted lists (or arrays). It finds the middle element, makes a comparison to determine whether the desired value comes before or after it and then searches the remaining half in the same manner.
- The binary search is an improvement, in that it eliminates half the list from consideration at each iteration. The prerequisite for it is that the list should be sorted order.



6.6 FURTHER READING

1) Lipschutz, S. (1987). *Schaum's Outline of Data Structure*. McGraw-Hill, Inc..
2) Robert, L. K., Tondo, C. L., & Leung, B. P. (2002). *Data Structures And Program Design*.



6.7 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: i) (a), ii) (b)

Ans. to Q. No. 2: Sequential search is easy to implement and relatively efficient to use for small lists. It does not require a sorted list of elements or data.

Ans. to Q. No. 3: i) True, ii) False, iii) True, iv) True

Ans. to Q. No. 4: Sequential (Linear) search is applied.

Ans. to Q. No. 5:

```
int find (int a[ ], int n, int x )
{
    int i = 0;
    while(i < n)
    {
        int mid = (n+i) / 2;
        if (a[mid] < x)
            n = mid;
        else if (a[mid] > x )
            i = mid+1;
        else
            return mid;
        }
        return 0;
}
```



6.8 MODEL QUESTIONS

Q.1: Write a program to find the given number in an array of 20 elements. Also display how many times a given number exists in the list.

Q.2: What is searching? What are the advantages and disadvantages of sequential search technique?

Q.3: What are the advantages of binary search technique?

Q.4: Differentiate between linear and binary search.

Q.5: Write a program to demonstrate binary search. Use integer array and store 10 elements. Find the given element.

Q.6: Write down the best, worst and average case time complexity of Sequential search.

Q.7: Write a program to demonstrate successful and unsuccessful search. Display appropriate messages.

*** ***** ***

UNIT 7: SORTING

UNIT STRUCTURE

7.1 Learning Objectives

7.2 Introduction

7.3 Sorting

7.4 Insertion Sort

7.5 Selection Sort

7.6 Bubble Sort

7.7 Quick Sort

7.8 Let Us Sum Up

7.9 Further Reading

7.10 Answers to Check Your Progress

7.11 Model Questions

7.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- define sorting
- explain insertion sort algorithm
- describe selection sort algorithm
- explain bubble sort algorithm
- describe quick sort algorithm
- compute the complexity of the above sorting algorithms.

7.2 INTRODUCTION

In computer science and mathematics, a sorting algorithm is an algorithm that puts elements of a list in a certain order. The most used orders are numerical order and lexicographical order. Since the dawn of computing, the sorting problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement. Efficient sorting is important to optimizing the use of other algorithms that require sorted lists to work correctly; it is also often useful for canonicalizing data and for producing human-readable output.

We have leaned about binary search and linear searching techniques in the previous unit. This unit, we will introduce you to the fundamental concepts of sorting and we shall discuss about the various sorting algorithms i.e., insertion sort, selection sort, bubble sort and quick sort including their complexity. The next block of units will deal with the concept of different types of trees and graphs.

7.3 SORTING

Sorting refers to the operation of arranging data in some given order, such as increasing or decreasing, with numerical data, or alphabatically, with character data. In real life we come accross several examples of sorted information. For example, in telephone directory the names of the subscribers and their phone numbers are written in alphabetical order. The records of the list of these telephone holders are to be sorted by their names. By using the directory, we can access the telephone number and the address of the subscriber very easily. In the same way in dictionary the words are placed in lexicographical order which is nothing but a sorting.

Let L be a list of *n* elements L_1, L_2, ..., ..., ..., L_n in memory. *Sorting* a list refers to the operation of rearranging the contents of L so that they are increasing in order (numerically or lexicographically), that is,

$$L_1 \leq L_2 \leq L_3 \leq \ldots \ldots \ldots L_n$$

Since L has *n* elements, there are *n!* ways that the contents can appear in L. These ways correnspond precisely to the n! permutations of 1, 2, ..., ..., ...n.

Let us assume that an array ELEMENT contains 10 elements as follows:

ELEMENT: 56, 37, 43, 21, 34, 16, 59, 25, 90, 64

After sorting, ELEMENT must appear in memory as follows:

ELEMENT: 16, 21, 25, 34, 37, 43, 56, 59, 64, 90

Since ELEMENT consists of 10 elements, there are 10!=3628800 ways that the numbers 16, 21, ..., ..., ..., 90 can appear in ELEMENT.

7.4 INSERTION SORT

Insertion sort is a simple sorting algorithm that is relatively efficient for small lists and mostly-sorted lists, and it is often used as part of more sophisticated algorithms. It works by taking elements from the list one by one and inserting them in their correct position into a new sorted list. In arrays, the new list and the remaining elements can share the array's space, but insertion is expensive, requiring shifting all following elements over by one.

Let A be an array with *n* elements A[0], A[1], ..., ..., ..., A[N – 1] in memory. The insetrtion sort algorithm scans the the array from A[0] to A[N –1], and the process of inserting each element in proper place is as follows:

Pass 1: A[0] by itself is sorted because of one element.

Pass 2: A[1] is inserted either before or after A[0], so that A[0], A[1] are sorted.

Pass 3: A[2] is inserted into its proper place in A[0], A[1], i.e., before A[0], between A[0] and A[1], or after A[1], so that: A[0], A[1], A[2] are sorted.

Pass 4: A[3] is inserted into its proper place in A[0], A[1], A[2] so that: A[0], A[1], A[2], A[3] are sorted.

.....

.....

Pass N: A[N – 1] is inserted into its proper place in A[0], A[1], ..., ..., ..., A[N – 2] so that: A[0], A[1], ..., ..., ..., A[N – 1] is sorted.

Insertion sort algorithm is frequently used when *n* is small.

The element inserted in the proper place is compared with the previous elements and placed in between the i^{th} element and $(i + 1)^{th}$ element if,

element $\leq i^{th}$ element

element $\geq (i+1)^{th}$ element

Let us take an example of the following elements:

	82	42	49	8	92	25	59	52
Pass 1	**82**	42	49	8	92	25	59	52
Pass 2	82	**42**	49	8	92	25	59	52
Pass 3	42	82	**49**	8	92	25	59	52
Pass 4	42	49	82	**8**	92	25	59	52
Pass 5	8	42	49	82	**92**	25	59	52
Pass 6	8	42	49	82	**92**	25	59	52
Pass 7	8	25	42	49	82	92	**59**	52
Pass 8	8	25	42	49	59	82	92	**52**
Sorted elements	8	25	42	49	59	82	92	52

Finally, we get the sorted array.The following program uses the insertion sort technique to sort a list of numbers.

Program 7.1: Program of sorting using insertion sort

```
#include<stdio.h>
#include<conio.h>

void main()
{
    int a[25], i, j, k, n;
    clrscr();
    printf("Enter the number of elements :'');
    scanf("%d",&n);
    for (i=0; i<n; i++)
    {
        printf("Enter element %d :'',i+1);
        scanf("%d", &a[i]);
    }
    printf("Unsorted list is:\n");
    for (i = 0; i < n; i++)
        printf("%d'', a[i]);
    printf("\n");
        /*Insertion sort*/
```

```
for(j=1; j<n; j++)
{
    k=a[j]; /*k is to be inserted at proper
    place*/
    for(i=j-1; i>=0 && k<a[i]; i--)
    a[i+1]=a[i];
    a[i+1]=k;
    printf("Pass %d, Element %d inserted in
    proper place\n", j,k);
for (i = 0; i < n; i++)
    printf("%d'', a[i]);
    printf("\n");
}
    printf("Sorted list is :\n");
    for (i = 0; i < n; i++)
        printf("%d '', a[i]);
    printf("\n");
    getch();
}   /*End of main()*/
```

Analysis: In insertion sort we insert the elements *i* before or after and we start comparison from the first element. Since the first element has no other elements before it, so it does not require any comparision. Second element requires 1 comparision, third element requires 2 comparisions, fourth element requires 3 comparisions and so on. The last element requires n – 1 comparisions. So the total number of comparisions will be–

$$1 + 2 + 3 + \ldots \ldots \ldots + (n - 2) + (n - 1)$$

It's a form of arithmetic progression series, so we can apply the formula sum = $\frac{n}{2}\{2a + (n - 1)d\}$.

Where, d = common difference i.e. first term – second term, a = first term in the series, n = total term.

Thus, sum = $\frac{(n-1)}{2}\{2 \times 1 + (n - 1) -1) \times 1\}$

$$= \frac{(n-1)}{2}\{2 + n - 2)$$

$$= n^2 \frac{(n-1)}{2}$$

Which is of $O(n^2)$.

It is the worst case behaviour of insertion sort where all the elements are in reverse order. If we compute the average case of the above algorithm then it will be of $O(n^2)$. The insertion sort technique is very efficient if the number of elements to be sorted are very less.

7.5 SELECTION SORT

Let us have a list containing *n* elements in unsorted order and we want to sort the list by applying the selection sort algorithm. In selection sort technique, the first element is compared with all the remaining (n – 1) elements. The smallest element is placed at the first location. Again, the second element is compared with the remaining (n – 2) elements and pick out the smallest element from the list and placed in the second location and so on until the largest element of the list.

Let A is an array with *n* elements A[0], A[1], ..., ..., A[N – 1]. First we will search the position of smallest element from A[0], ..., ..., ..., A[N – 1]. Then we will interchange that smallest element with A[0]. Now you will search the position of the second smallest element (because the A[0] is the first smallest element) from A[1], ..., ..., ..., A[N – 1], then interchange that smallest element with A[1]. Similarly the process will be for A[2], ..., ..., ..., A[N – 1]. The whole process will be as–

Pass 1: Search the smallest element from A[0], A[1], ..., ..., A[N – 1]
Interchange A[0] with the smallest element
Result: A[0] is sorted.

Pass 2: Search the smallest element from A[1], A[2], ..., ..., A[N – 1]
Interchange A[1] with the smallest element
Result: A[0], A[1] is sorted.

.....

.....

Pass N – 1: Search the smallest element from A[N – 2], A[N – 1]

Interchange A[N -2] with the smallest element

Result: A[0], A[1], ..., ..., A[N – 2], A[N – 1] is sorted.

Thus A is sorted after N – 1 passes.

Let us take an example of the following elements:

Pass 1	**75**	35	42	13	87	24	64	57
Pass 2	13	**35**	42	75	87	**24**	64	57
Pass 3	13	24	**42**	75	87	**35**	64	57
Pass 4	13	24	35	**75**	87	**42**	64	57
Pass 5	13	24	35	42	**87**	75	64	57
Pass 6	13	24	35	42	57	**75**	**64**	87
Pass 7	13	24	35	42	57	64	**75**	87
Sorted elements	13	24	35	42	57	64	**75**	87

Program 7.2: Program of sorting using selection sort

```
#include<stdio.h>
#include<conio.h>

void main()
    {
        int a[25], i, j, k, n, temp, smallest;
        clrscr();
        printf("Enter the number of elements :'');
        scanf("%d", &n);
        for (i = 0; i < n; i++)
        {
            printf("Enter element %d:'', i+1);
            scanf("%d", &a[i]);
        }
            /*Display the unsorted list*/
            printf("Unsorted list is : \n");
            for (i = 0; i < n; i++)
                printf("%d '', a[i]);
            printf("\n");
```

```
            /*Selection sort*/
        for(i = 0; i < n-1 ; i++)
        {
            /*Find the smallest element*/
                smallest = i;
            for(k = i+1; k < n ; k++)
            {
                if(a[smallest] > a[k])
                    smallest = k ;
            }
            if(i ! = smallest)
            {
                temp = a [i];
                a[i] = a[smallest];
                arr[smallest] = temp ;
            }
            printf(''After Pass %d elements are
            : '', i+1);
            for (j = 0; j < n; j++)
                printf(''%d '', a[j]);
                printf(''\n'');
        } /*End of for*/
        printf("Sorted list is : \n");
        for (i = 0; i < n; i++)
            printf(''%d '', a[i]);
        printf(''\n'');
    getch();
} /*End of main()*/
```

Analysis: As we have seen selection sort algorithm will search the smallest element in the array and then that element will be at proper position. So in Pass 1 it will compare n – 1 elements, in Pass 2 comparison will be n – 2 because the first element is already at proper position. Thus we can write the function for comparison as:

$$F(n) = (n-1) + (n-2) + (n-3) + \ldots\ \ldots\ \ldots + 3 + 2 + 1$$

This is an arithmetic series, solving the series we will get,

$$F(n) = \frac{(n-1)}{2}\,[2(n-1) + \{(n-1)-1\}\{(n-2)-(n-1)\}]$$

$$= \frac{(n-1)}{2}\,[2n - 2 + (n-1-1)(n-2-n+1)]$$

$$= \frac{(n-1)}{2}\,[2n - 2 + (n-2)(-1)]$$

$$= \frac{(n-1)}{2}\,[2n - 2 - n + 2]$$

$$= \frac{n(n-1)}{2}$$

$$= O(n^2)$$

Thus, the number of comparisons is proportional to (n^2). It is the worst case behaviour of selection sort. If we compute the average case of the above algorithm then it will be of $O(n^2)$. The best thing with the selection sort is that in every pass one element will be at correct position, very less temporary variables will be required for interchanging the elements and it is simple to implement.

7.6 BUBBLE SORT

Bubble sort is a commonly used sorting algorithm. In this algorithm, two successive elements are compared and interchanging is done if the first element is grearter than the second one. The elements are sorted in ascending order.

Let A is an array with *n* elements A[0], A[1],, ..., ..., ..., A[N – 1]. The bubble sort algorithm works as follows:

Step 1: First compare A[0] and A[1] and arrange them so that A[0]<A[1]. Then compare A[1] and A[2] and arrange them so that A[1]<A[2]. Then compare A[2] and A[3] and arrange them so that A[2]<A[3]. Continue until we compare A[N – 2] and A[N – 1] and arrange them int the desired order so that A[N – 2]<A[N – 1].



The process of sequentially traversing through all or part of a list is frequently called a *pass*, so each of the above steps is called a *pass*.

Observe that Step 1 involves *n – 1* comparisions. During the Step 1, the largest element is "bubbled up" to the *n*th position or "sinks" to the *n*th position. When Step 1 is completed, A[N – 1] will contain the largest element.

Step 2: Repeat Step 1 and finally the second largest element will occupy A[N -2]. In this step there will be *n – 2* comparisons.

Step 3: Repeat Step 1 and finally the third largest element will occupy A[N -3]. In this step there will be *n – 3* comparisons.

.....

.....

Step N–1: Compare A[1] and A[2] and arrange them so that A[1]<A[2]

After *n – 1* steps, the list will be sorted in ascending order.

Let us take an example of the following elements:

13 32 20 62 68 52 38 46

We will apply the bubble sort algorithm to sort the elements.

Pass 1: We have the following comparisions:

a) Compare A0 and A1, 13<32, no change

b) Compare A1 and A2, 32>20, interchange

13 20 32 62 68 52 38 46

c) Compare A2 and A3, A2<A3, no change

d) Compare A3 and A4, A3<A4, no change

e) Compare A4 and A5, A4>A5, interchange

13 20 32 62 52 68 38 46

f) Compare A5 and A6, A5>A6, interchange

13 20 32 62 52 38 68 46

g) Compare A6 and A7, A6>A7, interchange

13 20 32 62 52 38 46 68

Pass 2: 13 20 32 62 52 38 46 68

a) Compare A0 and A1, 13<20, no change

b) Compare A1 and A2, 20<32, no change

c) Compare A2 and A3, 32<62, no change

d) Compare A3 and A4, 62>52, interchange

13 20 32 52 62 38 46 68

e) Compare A4 and A5, 62>38, interchange

13 20 32 52 38 62 46 68

f) Compare A5 and A6, 46>62, interchange

13 20 32 52 38 46 62 68

At the end of the Pass 2 the second largest element 62 has moved to its proper place.

Pass 3: 13 20 32 52 38 46 62 68

a) Compare A0 and A1, 13<20, no change

b) Compare A1 and A2, 20<32, no change

c) Compare A2 and A3, 32<52, no change

d) Compare A3 and A4, 52>38, interchange

13 20 32 38 52 46 62 68

e) Compare A4 and A5, 52>46, interchange

13 20 32 38 46 52 62 68

Pass 4: 13 20 32 38 46 52 62 68

a) Compare A0 and A1, 13<20, no change

b) Compare A1 and A2, 20<32, no change

c) Compare A2 and A3, 32<38, no change

d) Compare A3 and A4, 38<46, no change

Pass 5: 13 20 32 38 46 52 62 68

a) Compare A0 and A1, 13<20, no change

b) Compare A1 and A2, 20<32, no change

c) Compare A2 and A3, 32<38, no change

Pass 6: 13 20 32 38 46 52 62 68

a) Compare A0 and A1, 13<20, no change

b) Compare A1 and A2, 20<32, no change

Pass 7: 13 20 32 38 46 52 62 68

a) Compare A0 and A1, 13<20, no change

Since the list has 8 elements, it is sorted after the 7th Pass. The list was actually sorted after the 4th Pass.

Program 7.3: Program of sorting using bubble sort

```
#include<stdio.h>
#include<conio.h>

void main()
    {
        int a[25], i, j, k, temp, n, xchanges;
        clrscr();
        printf(''Enter the number of elements :'');
        scanf(''%d'',&n);
        for (i = 0; i < n; i++)
        {
            printf(''Enter element%d :'', i+1);
            scanf(''%d'',&a[i]);
        }
        printf(''Unsorted list is:\n'');
        for (i = 0; i < n ; i++)
            printf(''%d '', a[i]);
        printf(''\n'');

        /* Bubble sort*/
        for (i = 0; i < n-1 ; i++)
        {
            xchanges=0;
        /* Bubble sort*/
        for (i = 0; i < n-1 ; i++)
        {
            xchanges=0;
            for (j = 0; j <n-1-i; j++)
            {
                if (a[j] > a[j+1])
                {
                    temp = a[j];
```

```
                    a[j] = a[j+1];
                    a[j+1] = temp;
                    xchanges++;
                } /*End of if*/
            } /*End of inner for loop*/
            if(xchanges==0)    /*If list is sorted*/
                break;
            printf(``After Pass %d elements are :
            '', i+1);
            for (k = 0; k < n; k++)
                printf(``%d '', a[k]);
            printf(``\n'');
        } /*End of outer for loop*/

        printf(``Sorted list is :\n'');
        for (i = 0; i < n; i++)
            printf(``%d '', a[i]);
        getch();
        printf(``\n'');
    } /*End of main()*/
```

Analysis: As we have seen bubble sort algorithm will search the largest element in the array and placed it at proper position in each Pass. So in Pass 1 it will compare n – 1 elements, in Pass 2 comparison will be n – 2 because the first element is already at proper position. Thus, we can write the function for comparison as

F(n) = (n – 1) + (n – 2) + (n – 3) + ... + 3 + 2 + 1

This is an arithmetic series, solving the series we will get ,

$$F(n) = \frac{n}{2}\{2a + (n-1)d\}$$

$$= \frac{(n-1)}{2}\{2 \times 1 + (n-1) - 1) \times 1\}$$

$$= \frac{(n-1)}{2}\{2 + n - 2)$$

$$= \frac{n(n-1)}{2}$$

$$= O(n^2)$$

Thus, the time required to execute the bubble sort algorithm is proportional to (n^2), where *n* is the number of input items.

7.7 QUICK SORT

This is the most widely used sorting algorithm, invented by C.A.R. Hoare in 1960. This algorithm is based on partition. Hence it falls under the divide and conquer technique. In this algorithm, the main list of elements is divided into two sub-lists. For example, a list of *n* elements are to be sorted. The quick sort marks an element in the list called **pivot** or **key**. Consider the first element *P* as a pivot. Shift all the elements whose value is less than *P* towards the left and elements whose value is greater than *P* to the right of *P*. Now, the pivot element divides the main list into two parts. It is not necessary that the selected key element must be in the middle. Any element from the list can act as key or pivot element.

Now, the process for sorting the elements through quick sort is as:

1) Take the first element of list as pivot.
2) Place pivot at the proper place in list. So one element of the list i.e., pivot will be at its proper place.
3) Create two sublists left and right side of pivot.
4) Repeat the same process untill all elements of the list are at proper position in the list.

For placing the pivot element at proper place we have to do the following process:

1) Compare the pivot element one by one from right to left for getting the element which has value less than the pivot element.
2) Interchange the element with pivot element.
3) Now the comparision will start from the interchanged element position from left to right for getting the element which has higher value than pivot.
4) Repeat the same process untill pivot is at its proper position.

Let us take a list of element and assume that 48 is the pivot element.

48 29 8 59 72 88 42 65 95 19 82 68

We have to start comparision from right to left. Now the first element less than 48 is 19. So interchange it with pivot i.e., 48.

19 29 8 59 72 88 42 65 95 48 82 68

Now the comparision will start from 19 and will be from left to right. The first element greater than 48 is 59. So interchange it with pivot.

19 29 8 48 72 88 42 65 95 59 82 68

Now the comparision will start from 59 and will be from right to left. The first element less than 48 is 42. So interchange it with pivot.

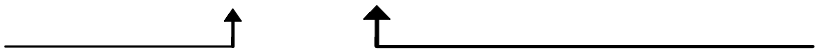
19 29 8 42 72 88 48 65 95 59 82 68

Now the comparision will start from 42 and will be from left to right. The first element greater than 48 is 72. So interchange it with pivot.

19 29 8 42 48 88 72 65 95 59 82 68

Now the comparision will start from 72 and will be from right to left. There is no element less than 48. So, now 48 is at its proper position in the list. So we can divide the list into two sublists, left and right side of pivot.

19 29 8 42 48 88 72 65 95 59 82 68



Sublist 1 Sublist 2

Now, the same procedure will be followed for the sublist 1 and sublist 2 and finally you will get the sorted list.

Program 7.4: Program to demonstrate the quick sort allgorithm

```
#include<stdio.h>
#include<conio.h>

enum bool { FALSE,TRUE };
void main()
    {
        int elem[20],n,i;
        clrscr();
        printf(''Enter the number of elements :'');
```

```
        scanf(''%d'',&n);
        for(i=0; i<n; i++)
        {
            printf(''Enter element %d : '',i+1);
            scanf(''%d'',&elem[i]);
        }

        printf(''Unsorted list is :\n'');
        display(elem,0,n-1);
        printf(''\n'');

        quick(elem,0,n-1);

        printf(''Sorted list is :\n'');
        show(elem,0,n-1);
        getch();
        printf(''\n'');

    }/*End of main()*/

quick(int a[ ],int low, int up)
    {
        int piv, temp, left, right;
        enum bool pivot_placed=FALSE;
        left=low;
        right=up;
        piv=low;/*Take the first element of
        sublist as piv */

        if(low>=up)
            return;
        printf(''Sublist : '');
        show(a,low,up);
```

```
/*Loop till pivot is placed at proper place in
the sublist*/

while(pivot_placed==FALSE)
    {
        /*Compare from right to left */
    while( a[piv]<=a[right] && piv!=right )
            right=right-1;
        if( piv==right )
            pivot_placed=TRUE;
/* Bubble sort*/
for (i = 0; i < n-1 ; i++)
{
    xchanges=0;
if( a[piv] > a[right] )
    {
        temp=a[piv];
        a[piv]=a[right];
        a[right]=temp;
        piv=right;
    }
/*Compare from left to right */
    while( a[piv]>=a[left] && left!=piv )
        left=left+1;
    if(piv==left)
        pivot_placed=TRUE;
    if(a[piv] < a[left])
    {
        temp=a[piv];
        a[piv]=a[left];
        a[left]=temp;
        piv=left;
    }
} /*End of while*/
```

```
            printf(``->Pivot Placed is%d -> '', a[piv]);
            show(a,low,up);
            printf(``\n'');

                quick(a, low, piv-1);
                quick(a, piv+1, up);
        }/*End of quick()*/

    show(int a[ ], int low, int up)
    {
        int i;
        for(i=low; i<=up; i++)
            printf(``%d '',a[i]);
    }
```

Analysis: The time required by the quick sorting method (i.e., the efficiency of the algorithm) depends on the selection of the pivot element. Suppose the pivot element is chosen in the middle position of the list so that it divides the list into two sublists of equal size. Now, repeatedly applying the quick sort algoritm on both the sublists we will finally have the sorted list of the elements.

Now, after 1^{st} step total elements in correct position is = $1 = 2^1 - 1$

after 2^{nd} step total elements in correct position is = $3 = 2^2 - 1$

Now, after 1^{st} step total elements in correct position is = $1 = 2^1 - 1$

after 3^{rd} step total elements in correct position is = $7 = 2^3 - 1$

.....

.....

after l^{th} step total elements in correct position is = $2^l - 1$

Therefore, $2^l - 1 = n - 1$

or $2^l = n$

or $l = \log_2 n$

Here, the value of *l* is the number of steps. If *n* is comparision per step then for $\log_2 n$ steps we get = $n\log_2 n$ comparisions. Thus, **the overall time complexity of quick sort is = $n\log_2 n$.**

The above calculated complexity is called the **average case complexity**. *So, the condition for getting the average case complexity is choosing the pivot element at the middle of the list*.

Now, let us suppose we have one condition that the given list of the elements is initially sorted. We consider the first element of the list as the pivot element. In this case, the number of steps needed for obtaining the finally sorted list is = (n – 1).

Again, the number of comparisions in each step will be almost *n* i.e., it will be of *O(n).*

So, the total number of comparisions for (n – 1)steps is = (n – 1)O(n)

$= O(n^2)$

Thus, the time complexity in this case will be = $O(n^2)$. This is called the **worst case complexity** of the quick sort algorithm. *So, the condition for worst case is if the list is initially sorted*.



CHECK YOUR PROGRESS

Q.1: Selection sort and quick sort both fall into the same category of sorting algorithms. What is this category?

A) O(n log n) sorts B) Divide-and-conquer sorts

C) Interchange sorts D) Average time is quadratic.

Q.2: What is the worst-case time for quick sort to sort an array of n elements?

A) O(logn) B) O(n)

C) O(nlogn) D) $O(n^2)$

Q.3) When is insertion sort a good choice for sorting an array?

A) Each component of the array requires a large amount of memory.

B) Each component of the array requires a small amount of memory.

C) The array has only a few items out of place.

D) The processor speed is fast.

Q.4: Give the time complexities of Insertion Sort, Selection Sort, Bubble Sort and Quick Sort.



7.8 LET US SUM UP

- Sorting is a method of arranging data in ascending or descending order.
- The Insertion Sort is so named because on each iteration of its main loop it inserts the next element in its correct position relative to the subarray that has already been processed.
- The complexity of Insertion Sort is $O(n^2)$.
- The complexity of Selection Sort is $O(n^2)$.
- The Bubble Sort name comes from the idea that the larger elements 'bubble up' to the top (the high end) of the array like the bubbles in waters.
- The Bubble Sort is slower than the Selection Sort, and the Insertion Sort (in most cases) is a little faster.
- The complexity of Bubble Sort is $O(n^2)$.
- The Quick Sort works by partitioning the array into two pieces separated by a single element that is greater than all the elements in the left piece and smaller than all the elements in the right piece.
- The complexity of Quick Sort is $O(nlogn)$.



7.9 FURTHER READING

1) Kamthane, A. N. (2004). *Introduction to Data Structures in C*. Pearson Education India.
2) Langsam, Y., Augenstein, M. & Tenenbaum, A. M. (1996). *Data Structures using C and C++* (Vol. 2). New Jersey: Prentice Hall.
3) Seymour, L. (2010). *Data Structures with C*. Tata McGraw Hill.



7.10 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: C

Ans. to Q. No. 2: D

Ans. to Q. No. 3: C

Ans. to Q. No. 4: The complexity of Insertion Sort is $O(n^2)$. The complexity of Selection Sort is $O(n^2)$. The complexity of Bubble Sort is $O(n^2)$. The complexity of Quick Sort is $O(nlogn)$.



7.11 MODEL QUESTIONS

Q.1: Explain the different types of sorting memthods.

Q.2: What is sorting? Write a function for Bubble Sort.

Q.3: Write a program for sorting a given list by using Insertion Sort:

2, 32, 45, 67, 89, 4, 3, 8, 10

Q.4: Write a program to sort the following members in ascending order using Selection Sort:

12, 34, 5, 78, 4, 56, 10, 23, 1, 45, 65

Q.5: Write a program to sort the following list using Quick Sort method:

4, 3, 1, 6, 7, 2, 5, 8

Q.6: Show all the passes using the Quick Sorting technique with the following list–

a) 26, 5, 37, 1, 61, 11, 15, 48, 19

b) 4, 3, 1, 6, 7, 2, 5, 8

Q.7: Show all the passes using the Bubble Sorting technique with the following list–

234, 54, 12, 76, 11, 87, 32, 12, 45, 67, 76

Q.8: Show all the passes using the Selection Sorting technique with the following list–

12, 34, 5, 78, 4, 56, 10, 23, 1, 45, 65

Q.9: Show all the passes using the Insertion Sorting technique with the following list–

13, 33, 27, 77, 12, 43, 10, 432, 112, 90

Q.10: Compare the performance of quick sort and selection sort.

Q.11: How does a quick sort performs on:

a) An array that is already sorted

b) An array that is sorted in reversed order

Q.12: The Bubble sort, the Selection sort, and the Insertion sort are all $O(n^2)$ algorithms. Which is the fastest and which is the slowest among them?

*** ***** ***

UNIT 8: TREES

UNIT STRUCTURE

8.1 Learning Objectives

8.2 Introduction

8.3 Tree

8.4 BinaryTree

8.5 Representation of Binary Tree

- 8.5.1 Sequential Representation of Binary Tree
- 8.5.2 Link Representation of Binary Tree

8.6 Tree Traversal Algorithms

- 8.6.1 Preorder Traversal
- 8.6.2 Inorder Traversal
- 8.6.3 Postorder Traversal

8.7 Prefix and Postfix Notations

8.8 Binary Search Tree

8.9 Let Us Sum Up

8.10 Further Reading

8.11 Answers to Check Your Progress

8.12 Model Questions

8.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- define tree as abstract data type (ADT)
- describe different properties of both tree and binary tree
- implement both trees and binary trees
- explain the different traversal technique of a binary tree
- explain binary search tree and operations on binary search tree.

8.2 INTRODUCTION

In the previous units, we have studied the various data structures such as arrays, stacks, queues and linked list. All these are linear data

structures. A tree is a nonlinear data structure that represents data in a hierarchical manner. It associates every object to a node in the tree and maintains the parent/child relationships between those nodes. Here, in this unit, we will introduce you to the trees, binary trees and binary search trees. We will also discuss the different tree traversal techniques in this unit. In the next unit, we will explore the concept of AVL trees, M-way trees and hashing.

8.3 TREE

A tree is a nonlinear data structure, where there is a specially designed node called root and rest of the node are divided into n subsets. Each subsets follow the properties of tree and the value of n is greater than equal to zero.

Definition [Tree]: A tree is a finite set of one or more nodes such that:

- There is a specially designated node called the root.
- The remaining nodes are partitioned into n (n>=0) disjoint sets T_1, ..., ..., ..., T_n, where each of these sets is a tree.

  We call T_1, ..., ..., ..., T_n the subtrees of the root.

  In figure 8.1 a general tree is given.

  Let us look at some more terminologies related to trees in general.

Definition [Subtree]: Any part of the tree which follows the properties of the tree is known as subtree of the tree.

Definition [Root Node]: A node is said to be a root node only if it does not have any parent. In figure 8.1 'A' is the root node of the tree.

Definition [Leaf Node]: A node is said to be a leaf node if it does not have any child. In figure 8.1 K, H, I, J and C are the leaf node.

Definition [Internal Node]: A node is said to be a internal node if the node has one or more children. In figure 8.1 A, B, D, E, F and G are the internal node.

Definition [Sibling]: A node is said to be sibling of another node only if both the nodes have the same parent. In figure 8.1 B and C are siblings, D, E, F are siblings and H and I are also siblings.

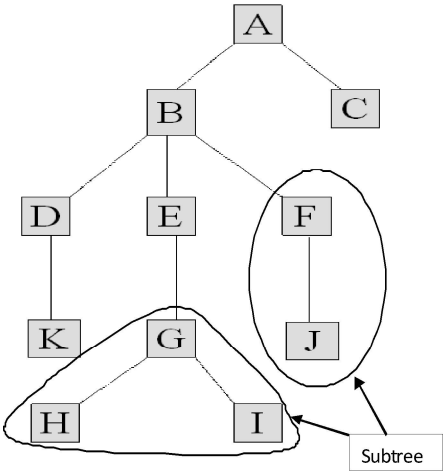


Figure 8.1: Tree

Definition [Level/Depth]: Level/Depth of a node is the distance from the root of the tree. In a tree root is at level 0, children of the root are at level 1, children of those nodes which are at level 1 are in level 2 and so on. In figure 8.1 D is at level 2, G is at level 3 etc.

Definition [Height of a tree]: Height of a tree is the maximum level of any node in the tree. In figure 8.1 Height of the tree is 4.

Definition [Degree of a Node]: The degree of a node is the number of children of that node. For example in Figure 8.1 Degree of node B is 3, Degree of node C is 0 etc. A node with degree 0 is called a leaf node.

In the following figure 8.2, UNIX file system is shown which is a example of tree structure.

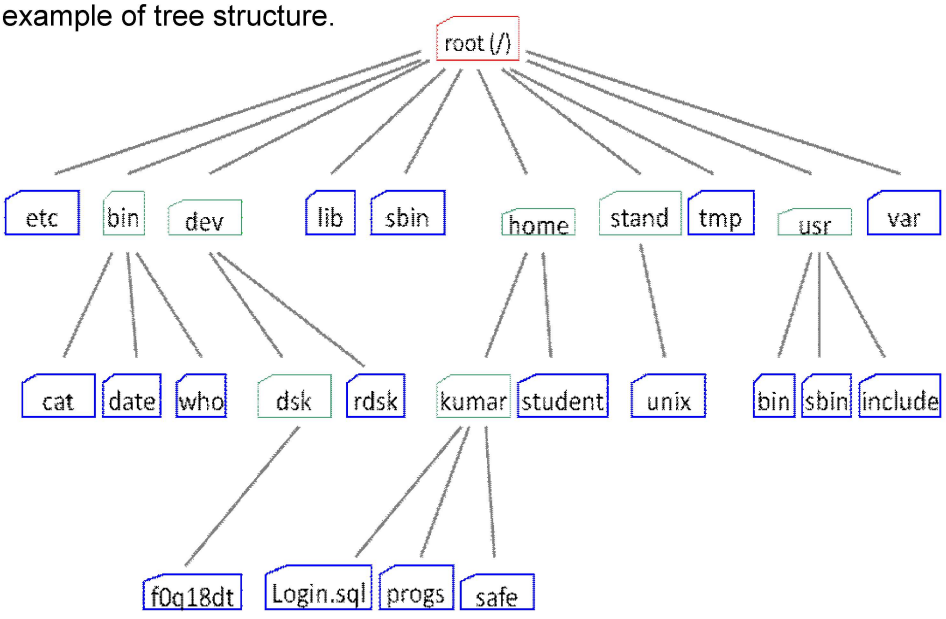


Figure 8.2: Example of Tree-UNIX File System

8.4 BINARY TREE

A binary tree is a tree which is either empty or has at most two subtrees, each of the subtrees also being a binary tree. It means each node in a binary tree can have 0, 1 or 2 subtrees.

Definition [Binary tree]: A Binary Tree is a *structure* defined on a finite set of nodes that either–

– contains no nodes, (*Null tree* or *empty tree*), or

– is composed of three *disjoint* sets of nodes: a root node, a binary tree called its left subtree, and a binary tree called its right subtree.

Arithmetic expressions involving only binary operations can be represented using binary tree. For example E = (a – b)/((c*d) + e) can be represented as shown in figure 8.3.

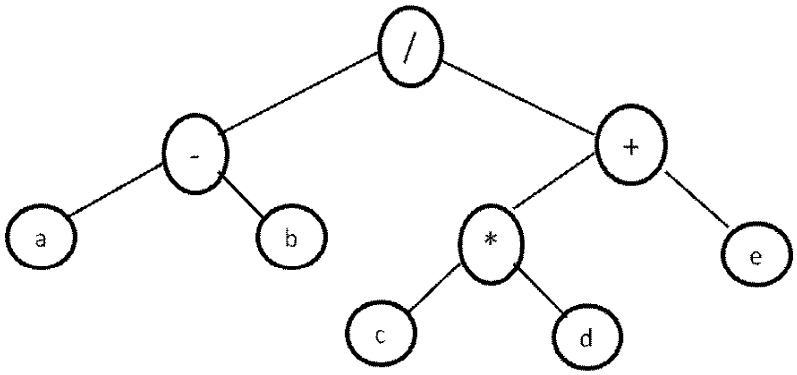


Figure 8.3: Representation of Arithmetic Expression

Definition [Complete Binary tree]: A Binary tree T is said to be *complete* if all its levels, except possibly the last, have the maximum number of possible nodes, and if all the nodes at the last level appear as far left as possible.

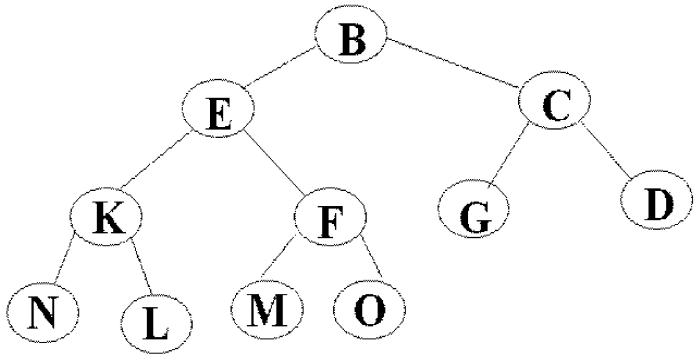


Figure 8.4: A Complete Binary Tree

Definition [Full Binary tree]: A binary tree is said to be a full binary tree if all the levels contain maximum possible node.

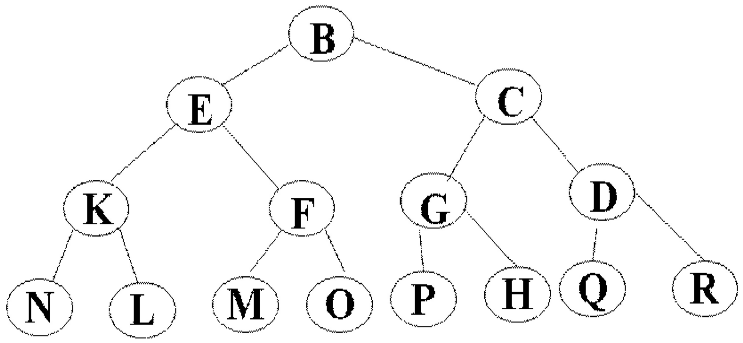


Figure 8.5: Full Binary Tree

Let us now, look at some mathematical facts related to trees.

- In a full binary tree i^{th} level will have 2^i element / node. If h is the height of a full binary tree. Then,

  Number of leaf node of the tree will be: 2^h

  Number of internal node of the tree will be:

  $$1 + 2 + 2^2 + \ldots \ldots ... + 2^{h-1} = 2^h - 1$$

  Total number of node will be = Number of internal node + Number of leaf node $= 2^h + 2^h - 1$

  $= 2.2^h - 1$

  $= 2^{h+1} - 1$

- For a full binary tree, number of internal node = number of leaf node –1.

- For a full binary tree having n nodes,

  n = number of internal node + number of leaf node

  Number of leaf node = n – number of internal node

  = n – (number of leaf node –1)

  Number of leaf node = (n+1)/2

  Height of a full binary tree is $\log_2$ (number of leaf) = $\log_2^{(n+1)/2}$

- For a binary tree of height h can

  At level i it can have maximum 2^i nodes

  Maximum number of node in the tree can be:

  $1 + 2^2 + 2^3 + \ldots \ldots ... + 2^h = 2^{h+1} - 1$

  Minimum Number of nodes in the tree can be = h + 1

- For a binary tree having n nodes

  Maximum height of the tree can be = n – 1

  Minimum height of the tree can be = $\log_2^{(n+1)/2}$

8.5 REPRESENTATION OF BINARY TREE

A binary tree can be represented in the computer memory in two ways:

- Sequential representation (by using Array)
- Link representation (by using node structure)

Let us look at both of these representations in detail in the subsections below.

8.5.1 Sequential Representation of Binary Tree

In sequential representation of binary tree, a single array is used to represent a binary tree. For this, nodes are indexed according to a scheme giving the value '0' to root. Then all the nodes are numbered from left to right level by level from top to bottom,empty nodes are also numbered. Then each node having an index i is put into the array as its i^{th} element.

In the figure 8.6 shown below, the nodes of binary tree are numbered according to the given scheme.

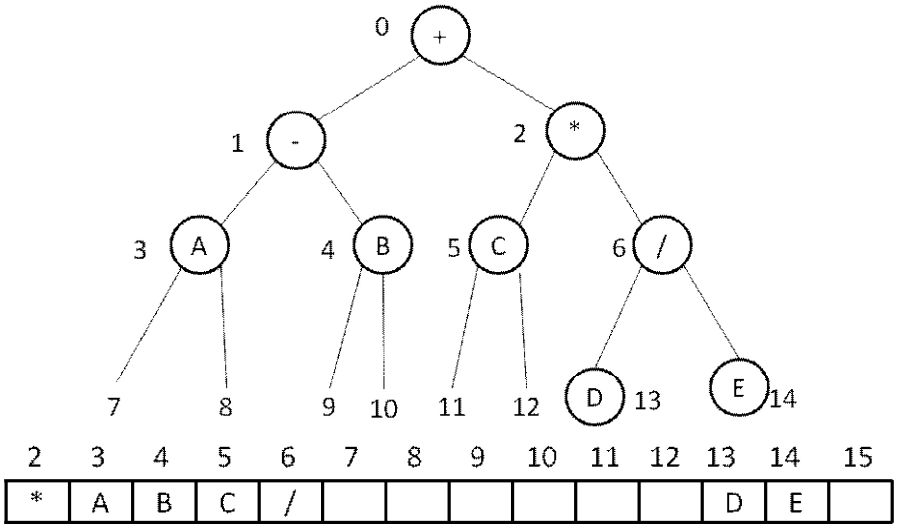


Figure 8.6: Array Representation of Binary Tree

Figure 8.6 shows how a binary tree is represented as an array. The root '+' is the 0^{th} element while its leftchild '–' is the 1^{st} element of the array. Node 'A' does not have any child so its children that are 7^{th} & 8^{th} element of the array are shown as a NULL value.

It is found that if n is the number or index of a node, then its left child occurs at $(2n + 1)^{th}$ position & right child at $(2n + 2)^{th}$ position of the array. If any node does not have any child, then null value is stored at the corresponding index of the array.

Let us look at some mathematical facts about binary tree now. In general, in array representation of binary tree:

Root is stored at position 0

Left child of the root is at position 1

Right child of the root is at position 2

For a node which array index is N

Left child of the node is at position 2xN+1

Right child of the node is at position 2xN+2

Parent of the node is at position (N–1)/2 if N is odd and (N–2)/2 if N is even.

Advantage of array representation of binary tree:

- Data is stored without any pointer to its successor or parent
- Any node can be accessed from any node by calculating the index. Efficient from execution point of view.

Disadvantage of array representation of binary tree:

- Most of the array entries are empty
- No possible way to enhance tree structure (Limited array size).
- Do not support dynamic memory allocation.
- Insertion, deletion etc. are inefficient (costly) as it requires considerable data movement, thus more processing time.

8.5.2 Link Representation of Binary Tree

In link representation, for representing each data one structure is used called "node", each node contains tree field's data part and two pointers to contain the address of the left child and the

right child. If any node has its left or right child empty then it will have in its respective link field, a null value. A leaf node has null value in both of its links.

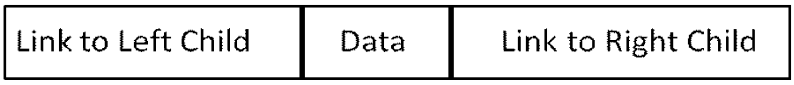


Figure 8.7: A Node Structure

The structure defining a node of binary tree in C is as follows.

```
struct node
          {
struct node*lc;/*points to the left child*/ int data;/*data field*/
struct node*rc;/*points to the right child*/
          }
```

The binary tree of arithmetic expression on figure 8.3 can be represented in link representation as shown in figure 8.7.

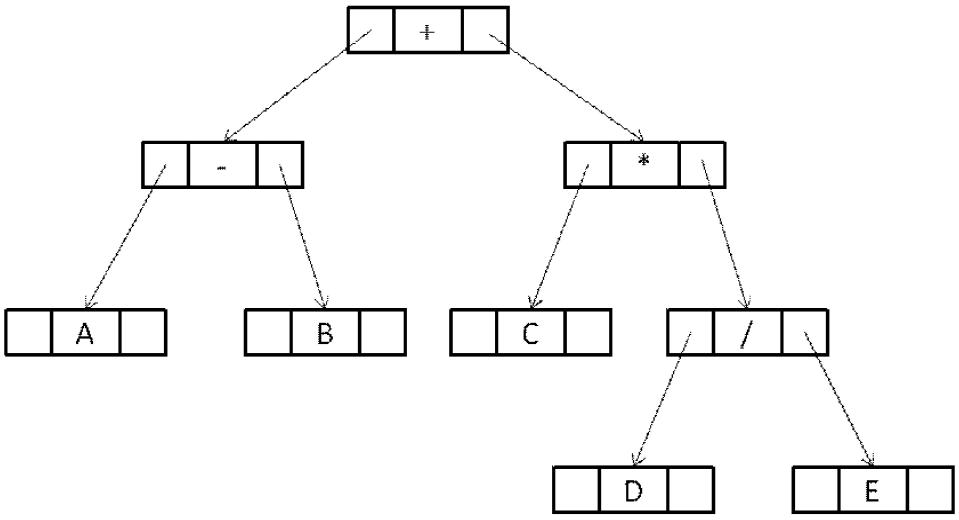


Figure 8.7: Link Representation of Binary Tree

Advantage of link representation of binary tree:

- No wastage of memory
- Enhancement of the tree is possible
- Insertions and deletions involve no data movement, only rearrangement of pointers.

Disadvantage of link representation of binary tree:

- Pointer fields are involved which occupy more space than just data fields.

8.6 TREE TRAVERSAL ALGORITHMS

Traversal of a binary tree means to visit each node in the tree exactly once. In a linear list, nodes are visited from first to last, but since trees are nonlinear; we need to definite rules to visit the tree. There are a number of ways to traverse a tree. All of them differ only in the order in which they visit the nodes.

The three main methods of traversing a tree are:

- Preorder Traversal
- Inorder Traversal
- Postorder Traversal

8.6.1 Preorder Traversal

In this traversal process, we visit every node as it moves left until it can move no further. Now it turns right to begin again or if there is no node in the right, retracts until it can move right to continue its traversal.

- Visit the Root
- Traverse the left subtree
- Traverse the right subtree

Algorithm 8.1: Preorder(node)

```
Step 1:  if (node==null) then return
           else
           visit(node)
           preorder(node–>leftchild)
           preorder(node–>rightchild)
```

The preorder traversal of the tree shown below is figure 8.8.

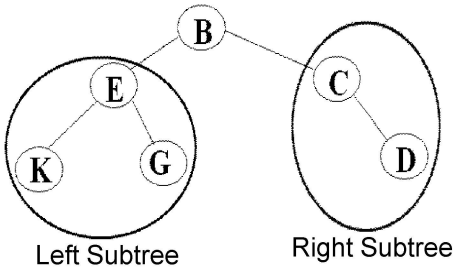


Figure 8.8: A Binary Tree

In preorder traversal of the tree process, B (root of the tree) traverses the left subtree and then traverses the right subtree. However the preorder traversal of left subtree process is the root E and then K and G. In the traversal of right subtree process the root C and then D. Hence, B E K G C D is the preorder traversal of the tree.

8.6.2 Inorder Traversal

The traversal keeps moving left in the binary tree until one can move no further to process node and moves to the right to continue its traversal again. In the absence of any node to the right, it retracts backward by a node and continues the traversal.

- Traverse the left subtree
- Visit the Root
- Traverse the right subtree

Algorithm 8.2: Inorder(node)

```
Step 1: if (node==null) then return
        else
        inorder(node–>leftchild)
        visit(node)
        inorder(node–>rightchild)
```

The inorder traversal of the tree (figure 8.8) traverses the left subtree, process B(root of the tree) and traverse right subtree. However, the inorder traversal of left subtree processes K, E and then G and the inorder traversal of right subtree processes C and then D. Hence, K E G B C D is the inorder traversal of the tree.

8.6.3 Postorder Traversal

The traversal proceeds by keeping to the left until it is no further possible, turns right to begin again or if there is no node to the right, processes the node and retraces its direction by one node to continue its traversal

- Traverse the left subtree
- Traverse the right subtree
- Visit the Root

Algorithm 8.3: Postorder(node)

```
Step 1:  if (node==null) then return
         else
         postorder(node–>leftchild)
         postorder(node–>rightchild)
         visit(node)
```

The postorder traversal of the tree (figure 8.8) traverse the left subtree, traverse right subtree and process B(root of the tree). However, the postorder traversal of left subtree processes K, G and then E and the postorder traversal of right subtree processes D and then C. Hence, K G E D C B is the postorder traversal of the tree.



CHECK YOUR PROGRESS

Q.1: What is the maximum height of a binary tree having n nodes?

Q.2: What is the maximum number of node in a binary tree at level i?

Q.3: What is the minimum number of node in a binary tree at level i?

Q.4: What is the minimum height of a binary tree having n node?

Q.5: What is the minimum number of node in a full binary tree at level i?

8.7 PREFIX AND POSTFIX NOTATION

The general way of writing arithmetic expression is known as the infix notation, where the binary operators are placed between the two operands on which it operates and parenthesis are used to distinguish the operations. For example, following expressions are in infix notation:

A+B,

(A+B)*(B–C),

A+((B/C)*D) – (F+G)

Here, the order of evaluation depends on the parenthesis and the precedence of operator.

Prefix Notation (Polish Notation): Polish notation, also known as prefix notation, is a symbolic logic invented by Polish mathematician Jan Lukasiewicz in the 1920's. When using Polish notation, the instruction (operation) precedes the data (operands).

For example, the prefix notation for the expression:

3*(4+5) is *3+4 5

(A+B)*(B–C) is *+AB–BC

The fundamental property of prefix notation is that the order in which the operations are performed is completely determined by the positions of the operators and operands in the expression.

Postfix Notation (Reverse Polish Notation): Postfix notation or reverse polish notation also a symbolic logic where the operator symbol is place after the operation. Later we will discuss the rules for converting infix expression to postfix expression.

For example, the postfix notation for the expression:

3*(4+5) is 3 4 5+*

(A+B)*(B–C) is AB+BC–*

Like prefix notation here also the order of the operations to be performed is completely determined by the positions of the operators and operands in the expression.

Expression Trees: Arithmetic expressions can be represented by using binary tree, for example E = (a–b)/((c*d)+e) can be represented as shown in figure 8.9.

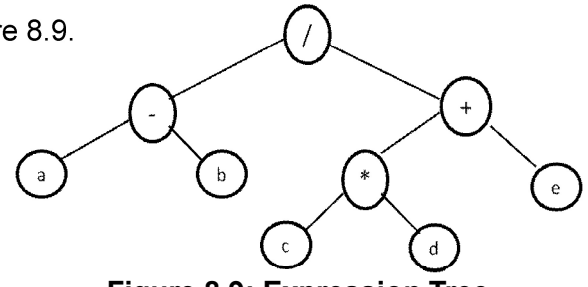


Figure 8.9: Expression Tree

This kind of tree is called an **expression tree.**

Here the terminal nodes (leaves) are the variables or constants in the expression (*a*, *b*, *c*, *d*, and *e*) and the non-terminal nodes are the operator in the expression (+, –, *, and /).

If we traverse the tree in preorder fashion we get, **/ – a b + * c d e** which is the prefix notation of the expression. If we traverse the tree in postorder fashion we get, **a b – c d * e + /,** which is the postfix notation of the expression.

The computer usually evaluates an arithmetic expression written infix notation in two steps. First it converts the expression in infix notation into postfix notation and then evaluates the postfix expression.

Conversion of infix to postfix expression: The conversion of an infix expression to a postfix expression takes account of precedence and associatively the rules of operator. This conversion uses stack. The steps for converting an infix expression to a postfix expression is given in algorithm 8.4.

Algorithm 8.4: Infix_to_postfix (E, P)

//Here E is the infix expression to be converted and P is the resulting postfix expression

Step 1: PUSH '('into the STACK and add')' to the end of E

Step 2: Scan E from left to right and repeat step 3 to 6 for each element of E until the STACK is not EMPTY

Step 3: If an operand is encountered, add it to P

Step 4: If a left parenthesis is encountered, PUSH it onto STACK

Step 5: If an operator is encountered, recursively POP all the element that has same or higher precedence and put it to P. PUSH the operator onto the STACK

Step 6: If a right parenthesis is encountered, POP all the elements and put it to P until a left parenthesis encountered and POP the left parenthesis from the STACK.

Step 7: STOP

Example 8.1: Consider the following arithmetic expression

A/(B+C*D) + E*(F/G)

Now according to the algorithm:

E = A/(B+C*D) + E*(F/G)

And P is empty

We will add '(' to the stack and ')' to the expression so

E = A/(B+C*D) + E*(F/G)

Symbol Scanned	STACK	Expression P
1. A	(	A
2. /	(/	A
3. (	(/ (	A
4. B	(/ (	A B
5. +	(/ (+	A B
6. C	(/ (+	A B C
7. *	(/ (+ *	A B C
8. D	(/ (+ *	A B C D
9.)	(/	A B C D * +
10. +	(+	A B C D * + /
11. E	(+	A B C D * + / E
12. *	(+ *	A B C D * + / E
13. (	(+ * (	A B C D * + / E F
14. F	(+ * (	A B C D * + / E F
15. /	(+ * (/	A B C D * + / E F
16. G	(+ * (/	A B C D * + / E F G
17.)	(+ *	A B C D * + / E F G /
18.)	EMPTY	A B C D * + / E F G / * +

When we encounter '+' in line 10 we send '/' to P from STACK since precedence of '/' is grater then the '+'

Evaluation of postfix expression: The evaluation of postfix expression is simple. We push all the operand onto the stack, whenever we encounter an operator we pop top two elements from stack perform the operation and push back the result on to the stack until we have any element left in the postfix expression.

Algorithm 8.5: Postfix_evaluation (P)

//Here P is the postfix expression to be evaluated

Step 1: Scan E from left to right and repeat step 3 to 6 for each element of E until end of the postfix expression

Step 2: If an operand is encountered, PUSH it to STACK

Step 3: If an operator is encountered, POP two elements from the STACK, Perform the operation. PUSH the result onto the STACK
(hint: if A is the top element and B is the next top element perform the operation as B operator A)

Step 4: STOP

Example 8.2: Consider the following postfix expression:

7 6 4 2 * + / 5 8 2 / * +

Answer: The postfix evaluation will be as follows:

Symbol Scanned	STACK	Operation performed
1. 7	7	
2. 6	7 6	
3. 4	7 6 4	
4. 2	7 6 4 2	
5. *	7 6 8	4 * 2 = 8
6. +	7 14	6 + 8 = 14
7. /	0.5	7 / 14 = 0.5
8. 5	0.5 5	
9. 8	0.5 5 8	
10. 2	0.5 5 8 2	
11. /	0.5 5 4	8 / 2 = 4
12. *	0.5 20	5 * 4 = 20
13. +	20.5	0.5 + 20 = 20.5 Result is 20.5

8.8 BINARY SEARCH TREE

A binary search tree is a binary tree that may be empty, and every node must contain an identifier. An identifier of any node in the left subtree is less than the identifier of the root. An identifier of any node in the right subtree is greater than the identifier of the root. Both the left subtree and right subtree are binary search trees.

Definition (recursive): A binary tree is said to be a binary search tree if it is the empty tree or

1) if there is a left-child, then the data in the left-child is less than the data in the root,
2) if there is a right-child, then the data in the right-child is no less than the data in the root, and
3) every sub-tree is a binary search tree.

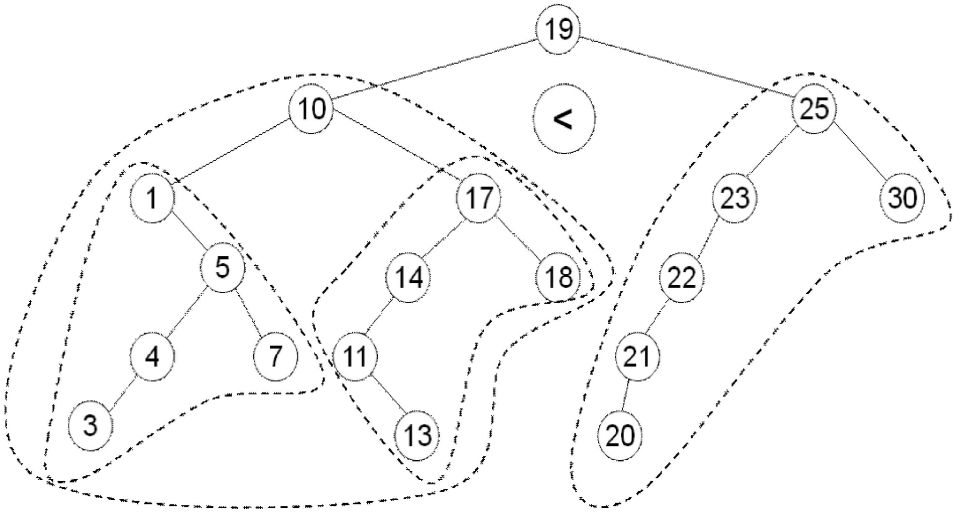


Figure 8.10 : Binary Search Tree

The binary search tree is basically a binary tree, and therefore it can be traversed in inorder, preorder, and postorder. If we traverse a binary search tree in inorder and print the identifiers contained in the nodes of the tree, we get a sorted list of identifiers in ascending order.

A binary search tree is an important search structure. For example, consider the problem of searching a list. If a list is ordered, searching becomes faster if we use a contiguous list and perform a binary search. But if we need to make changes in the list, such as inserting new entries and

deleting old entries, using a contiguous list would be much slower, because insertion and deletion in a contiguous list requires moving many of the entries every time. So we may think of using a linked list because it permits insertions and deletions to be carried out by adjusting only a few pointers. But in an n-linked list, there is no way to move through the list other than one node at a time, permitting only sequential access. Binary trees provide an excellent solution to this problem. By making the entries of an ordered list into the nodes of a binary search tree, we find that we can search for a key in O(nlogn) steps.

We assume that every node of a binary search tree is capable of holding an integer data item and that the links can be made to point to the root of the left subtree and the right subtree, respectively. Therefore, the structure of the node can be defined using the following declaration:

```
struct tnode
    {
structtnode *lchild;

int data;
structtnode *rchild;

    };
```

Program 8.1: Program for creating a Binary Search Tree

```
#include <stdio.h>
#include <stdlib.h>

struct tnode
{
structtnode *lchild;
int data;
structtnode*rchild;
};
```

```
struct tnode *insert(struct tnode *p,intval)
{
struct tnode *temp1,*temp2;
if(p == NULL)
    {
        p = (struct tnode *) malloc(sizeof(struct
        tnode));
        /*insert the new node as root node*/
        if(p == NULL)
            {
            printf("Cannot allocate\n");
            exit(0);
            }
            p->data = val;
            p->lchild=p->rchild=NULL;
                    }
                else
                    {
                temp1 = p;
    /* traverse the tree to get a pointer to that
    node whose child will be the newly created
    node*/
    while(temp1 != NULL)
        {
            temp2 = temp1;
            if( temp1 ->data >val)
            temp1 = temp1->lchild;
        else
            temp1 = temp1->rchild;
        }
        if( temp2->data >val)
{
temp2->lchild = (structtnode*)malloc(sizeof(struct
tnode));
```

```
        /*inserts the newly created node as left child*/
            temp2 = temp2->lchild;
        if(temp2 == NULL)
                {
            printf("Cannot allocate\n");
            exit(0);
                }
            temp2->data = val;
            temp2->lchild=temp2->rchild = NULL;
    }
    else
    {
    temp2->rchild = (structtnode*)malloc(sizeof(struct
    tnode));
        / *inserts the newly created node as left child*/
        temp2 = temp2->rchild;
    if(temp2 == NULL)
            {
        printf("Cannot allocate\n");
        exit(0);
            }
        temp2->data = val;
        temp2->lchild=temp2->rchild = NULL;
    }
    return(p);
    {
    /* a function to binary tree in inorder */
    void inorder(struct tnode *p)
        {
        if(p != NULL)
            {
        inorder(p->lchild);
```

```
    printf("%d\t",p->data);
    inorder(p->rchild);
        }
    }

void main()
{
struct tnode*root = NULL;
int n,x;
printf("Enter the number of nodes\n");
scanf("%d",&n);

while( n->0)
    {
printf("Enter the data value\n");
scanf("%d",&x);
root = insert(root,x);
    }
inorder(root);
}
```

Let us now look at the explanation of the above program.

- To create a binary search tree, we use a function called insert, which creates a new node with the data value supplied as a parameter to it, and inserts it into an already existing tree whose root pointer is also passed as a parameter.
- The function accomplishes this by checking whether the tree whose root pointer is passed as a parameter is empty. If it is empty, then the newly created node is inserted as a root node. If it is not empty, then it copies the root pointer into a variable temp1. It then stores the value of temp1 in another variable, temp2, and compares the data value of the node pointed to by temp1 with the data value supplied as a parameter. If the data value supplied as a parameter is smaller than the data value of the node pointed to by temp1, it

copies the left link of the node pointed to by temp1 into temp1 (goes to the left); otherwise it copies the right link of the node pointed to by temp1 into temp1 (goes to the right).

- It repeats this process until temp1 reaches NULL. When temp1 becomes NULL, the new node is inserted as a left child of the node pointed to by temp2, if the data value of the node pointed to by temp2 is greater than the data value supplied as a parameter. Otherwise, the new node is inserted as a right child of the node pointed to by temp2. Therefore the insert procedure is:
  - Input: The number of nodes that the tree to be created should have and the data values of each node in the tree to be created
  - Output: The data value of the nodes of the tree in inorder.

Example 8.3: Consider the following input for the binary search tree.

Input: The number of nodes that the created tree should have = 5
The data values of the nodes in the tree to be created are: 10, 20, 5, 9, 8

Output: 5 8 9 10 20

A function for preorder traversal of a binary tree is given in function 8.1.

Function 8.1: Preorder (struct tnode *p)

```
void preorder(struct tnode *p)
    {
        if(p != NULL)
            {
        printf("%d\t",p->data);
        preorder(p->lchild);
        preorder(p->rchild);
            }
    }
```

A function for postorder traversal of a binary tree is given in function 8.2.

Function 8.1: Postorder (struct tnode*p)

```
void postorder(struct node*p)
{
if(p!=NULL)
        {
    postorder(p->lchild);
    postorder(p->rchild);
    printf("%d\t",p->data);
        }
}
```

Searching for a target key in a binary search tree: Data values are given which we call a key in a binary search tree. To search for the key in the given binary search tree, start with the root node and compare the key with the data value of the root node. If they match, return the root pointer. If the key is less than the data value of the root node, repeat the process by using the left subtree. Otherwise, repeat the same process with the right subtree until either a match is found or the subtree under consideration becomes an empty tree.

If we search 2, 18 and 26 in the BST then the searching is shown in figure is 8.11.

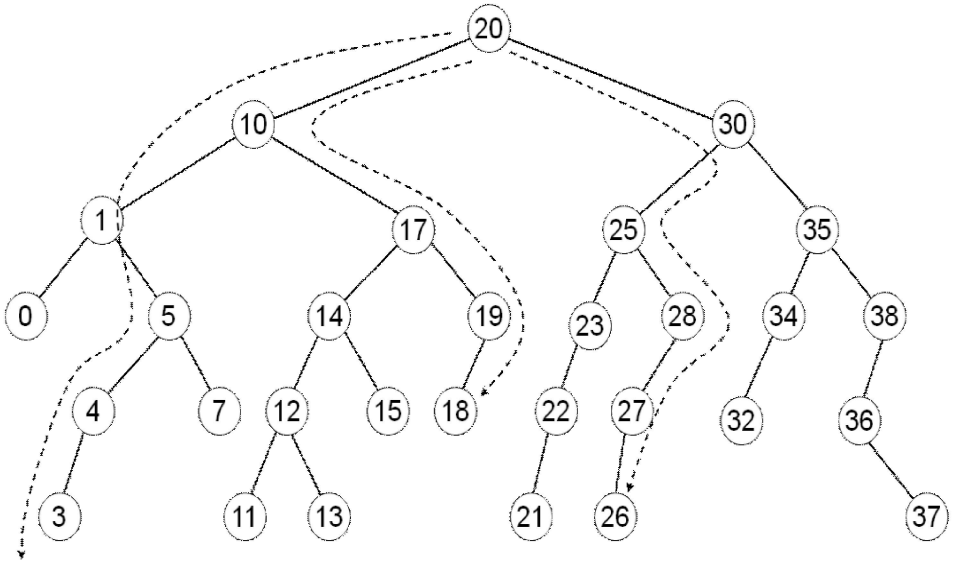


Figure 8.11: Searching in BST

A function to search for a given data value in a binary search tree is given in function 8.3.

Function 8.3: Search (struct tnode *p, int key)

```
struct tnode*search( struct tnode *p,int key)
    {
/* Here p initially is the root of the tree and key
is the data to be search*/
struct tnode*temp;
temp = p;
while(temp != NULL)
    {
        if(temp->data == key)
            return(temp);
        else
            if(temp->data > key)
            temp = temp->lchild;
        else
            temp=temp->rchild;
        }
        return(NULL);
    }
```

This function will return the node on success, otherwise the function will return NULL.

Deletion of a node from binary search tree: To delete a node from a binary search tree, the method to be used depends on whether a node to be deleted has one child, two children, or no children.

Deletion of a Node with No Child: If we want to delete a leaf node, we only need to set the pointer of its parent pointing to the node to NULL and delete the node.

Consider the binary search tree shown in figure 8.12.

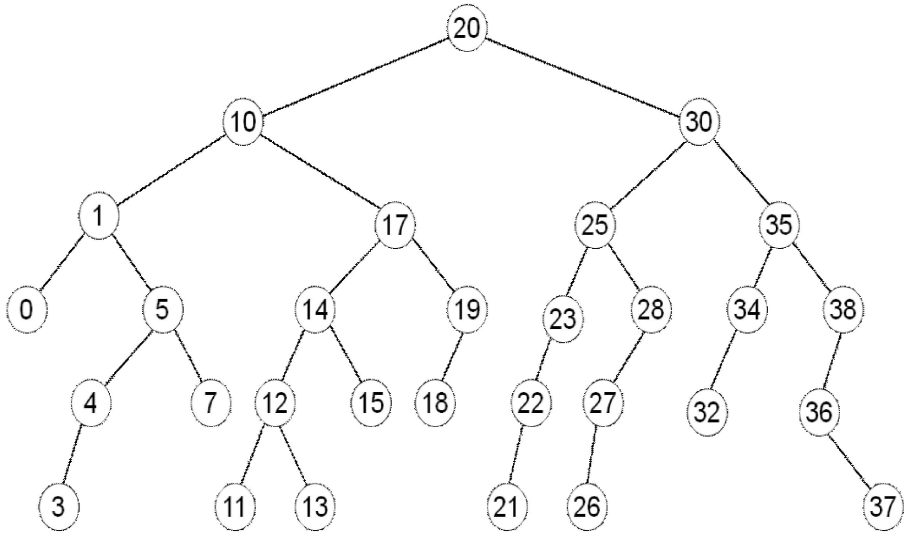


Figure 8.12: A Binary Tree

If we want to delete 13 from the above tree (figure 8.12), where 13 is a leaf node we only need to set the right pointer of 12 to NULL and then delete 13. After deleting the node 13 tree will be as shown below.

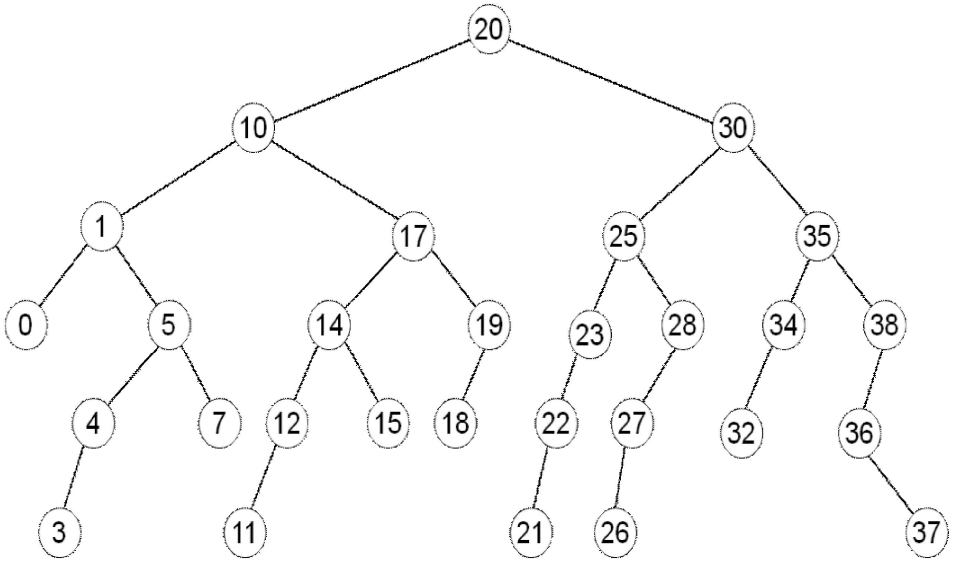


Figure 8.13: BST After Deleting the Node 13

Deletion of a Node with One Child: If we want to delete a node with one child then set the pointer of its parent pointing to this node to point the child of the node and delete the node.

If we want to delete 12 from the tree in Figure 8.13, where 12 is a node having only one child that is 11 we only need to set the left pointer of 14 (parent of 12) to point 11 and then delete 12.

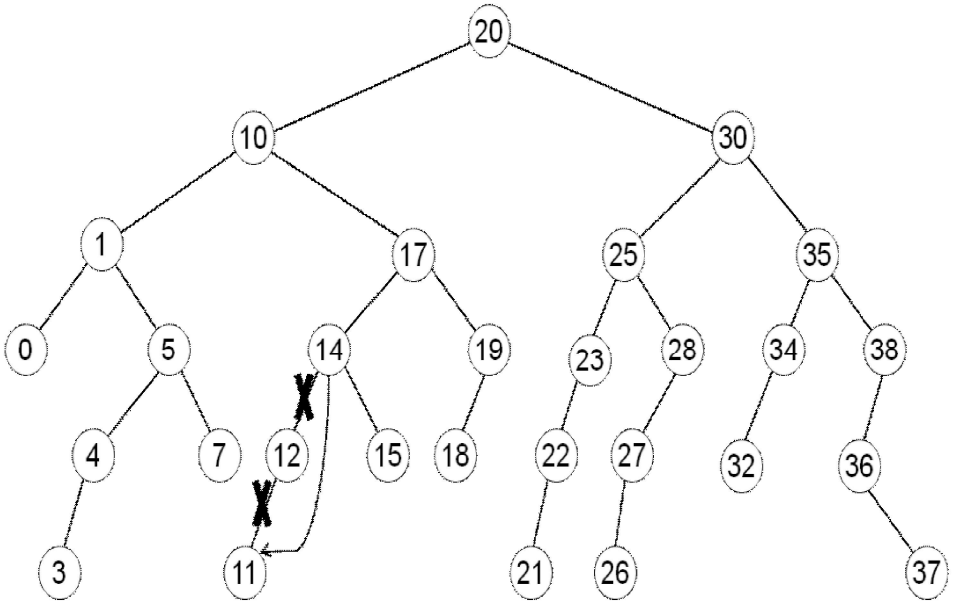


Figure 8.14: Intermediate Step for Delete the Node 12

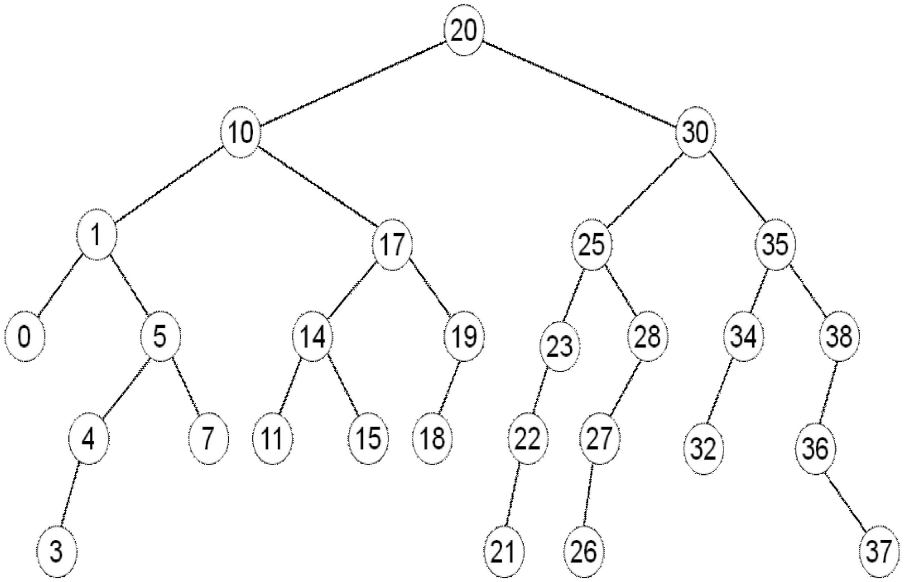


Figure 8.15: BST After Deleting the Node 12

Similarly the tree after deleting node 28 from the tree in figure 8.15 is shown below.

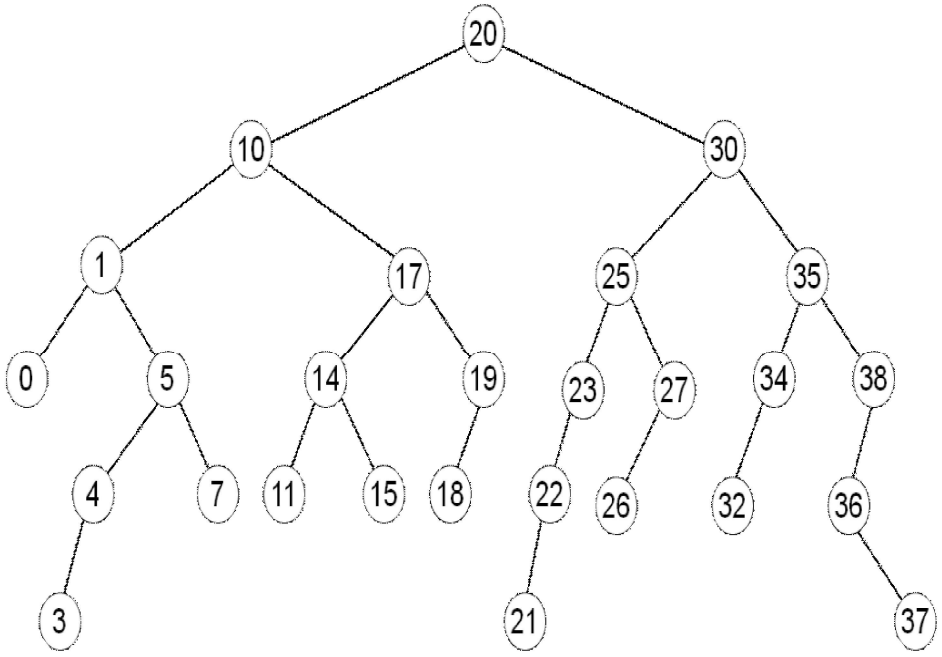


Figure 8.16: BST After Deleting the Node 28

Deletion of a node with two children: If the node to be deleted has two children then it is replaced by the inorder successor of the node and the inorder successor is deleted according to the rule.

If we want to delete 14 from the tree in figure 8.16, where 15 is the inorder successor of 14, first we need to delete 15 and then replace 14 by 15.

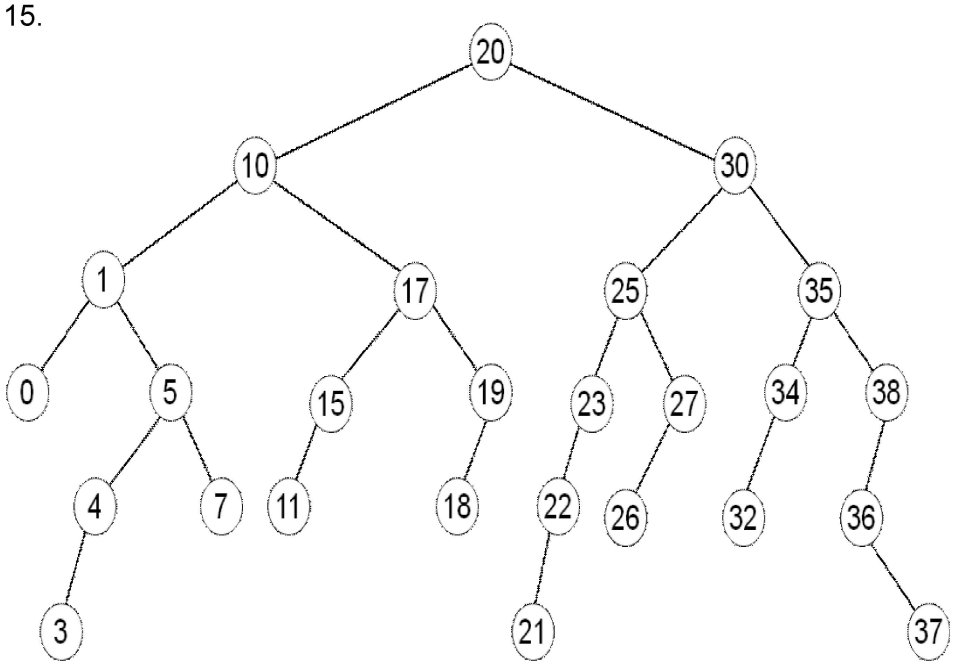


Figure 8.17: BST After Deleting the Node 14

If we want to delete 35 from the tree in figure 8.17, where 36 is the inorder successor of 35, first we need to delete 36 and then replace 35 by 36.

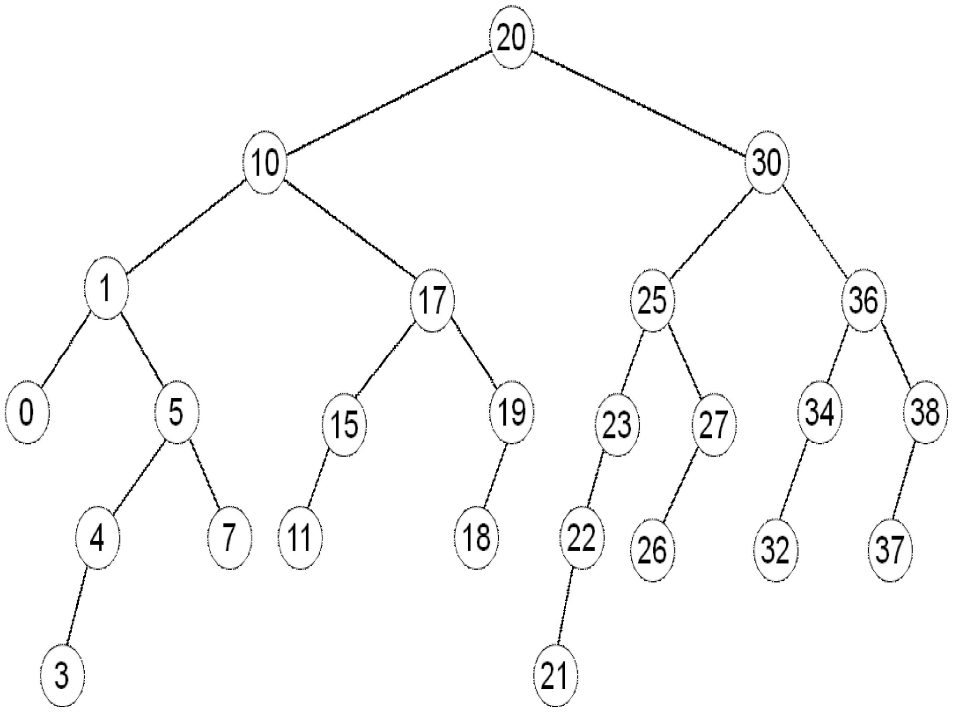


Figure 8.18: BST After Deleting the Node 35



CHECK YOUT PROGRESS

Q.6: State True or False:

a) Postfix notation can be evaluated with the help of stack.

b) In prefix notation parenthesis are used to maintain the ordering of operator.

c) A binary search tree may not be a binary tree.

d) We cannot delete the root node of binary search tree.

e) The value of left child of any node in a binary search tree is less than the value of the node.



8.9 LET US SUM UP

- Trees are used to organize a collection of data items into a hierarchical structure.
- A tree is a collection of elements called nodes, one of which is distinguished as the root, along with a relation that places a hierarchical structure on the node.
- The degree of a node of a tree is the number of descendants that node has.
- A leaf node of a tree is a node with a degree equal to 0.
- The level of the root node is 0, and as we descend the tree, we increment the level of each node by 1.
- Depth of a tree is the maximum value of the level for the nodes in the tree.
- The maximum number of nodes at level i in a binary tree is 2^i.
- Inorder, preorder, and postorder are the three commonly used traversals that are used to traverse a binary tree.
- In inorder traversal, we start with the root node, visit the left subtree first, then process the data of the root node, followed by that of the right subtree.
- In preorder traversal, we start with the root node. First we process the data of the root node, then visit the left subtree, then the right subtree.
- In postorder traversal, we start with the root node, visit the left subtree first, then visit the right subtree, and then process the data of the root node.



8.10 FURTHER READING

1) Aho, A. V. & Hopcroft, J. E. (1999). *The Design and Analysis of Computer Algorithms*. Pearson Education India.
2) Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. (2009). *Introduction to Algorithms*. MIT press.
3) Horowitz, E. (2006). *Fundamentals of Data Structures in C++*. Galgotia Publications.

4) Horowitz, E., Sahni, S. & Rajasekaran, S. (1997). *Computer Algorithms C++: C++ and Pseudocode Versions*. Macmillan.



8.11 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: n–1

Ans. to Q. No. 2: 2^i

Ans. to Q. No. 3: 1

Ans. to Q. No. 4: $\log_2^n$

Ans. to Q. No. 5: 2^i.

Ans. to Q. No. 6: a) True, b) False, c) False, d) False, e) True



8.12 MODEL QUESTIONS

Q.1: Write a C program to convert an infix expression to a postfix expression.

Q.2: Write a C program to evaluate a postfix expression.

Q.3: Write a C program to count the number of non-leaf nodes of a binary tree.

Q.4: Write a C program to delete all the leaf nodes of a binary tree.

Q.5: How many binary trees are possible with three nodes?

Q.6: Construct a binary search tree with following data point and find out the inorder preorder and postorder traversal of the tree:

i) 5, 1, 3, 11, 6, 8, 4, 2, 7

ii) 6, 1, 5, 11, 3, 4, 8, 7, 2

Q.7: For the following expression find out the postfix notation and evaluate the postfix notation:

i) (2+7*3)/(4*8–2) + 7

ii) 4*5 + (2–3*7) + 2/8

*** ***** ***

UNIT 9: SEARCH TREES

UNIT STRUCTURE

9.1 Learning Objectives

9.2 Introduction

9.3 Search Trees

9.4 Balancing of Search Trees

9.5 AVL Trees

9.5.1 AVL Rotations

9.6 Multi-way Search Trees

9.7 2-3 Trees

9.8 Splay Trees

9.9 Hashing

9.10 Let Us Sum Up

9.11 Further Reading

9.12 Answers to Check Your Progress

9.13 Model Questions

9.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- define search trees
- describe AVL tree and its different rotations,
- describe 2-3 tree
- explain splay tree
- describe hashing

9.2 INTRODUCTION

In the previous unit, we have learnt about the definition of trees and the different types of trees. We have learnt about binary tree and binary search tree in detail. In this unit, we will learn about search trees which are used for searching of elements in real life situations.

The search tree is a fundamental data structure in computer science. Just as with all other data structures and algorithms, the search tree is built

on specific rules and assumptions. These assumptions enable us to examine the shortcomings of previous data structures such as the linked list, and improve the search time. In this chapter, we will explore the search tree data structure and some of the searching algorithms used. Well, it is because of these assumptions that we are able to get such high performance from certain data structures. Assumptions enable us to code specifically. The search tree enables us to search for items much more quickly. In the next unit, we will explore the concept of red black trees in detail.

9.3 SEARCH TREES

A search tree is a tree data structure used for locating specific keys from within a set. In order for a tree to function as a search tree, the key for each node must be greater than any keys in subtrees on the left and less than any keys in subtrees on the right.

The advantage of search trees is their efficient search time given the tree is reasonably balanced. This is to say that the leaves at either end are of comparable depths. Various search-tree data structures exist, several of which also allow efficient insertion and deletion of elements, and operations then have to maintain tree balance.

Search trees are often used to implement an associative array. The search tree algorithm uses the key from the key-value pair to find a location, and then the application stores the entire key-value pair at that location.

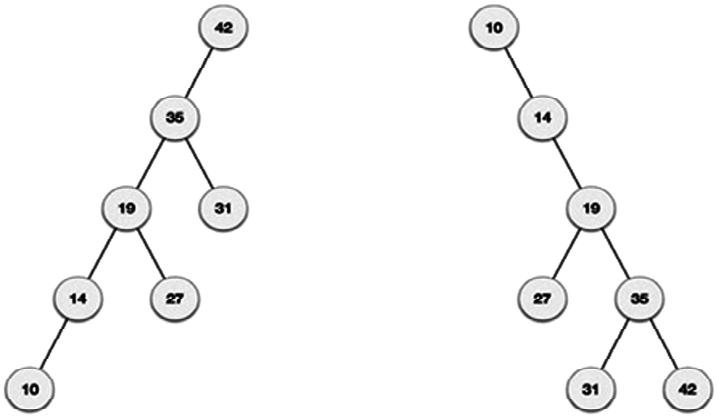
9.4 BALANCING OF SEARCH TREE

Most operations on a binary search tree take time directly proportional to the height of the tree, so it is desirable to keep the height small. A balanced binary search tree is a tree that automatically keeps its height small (guaranteed to be logarithmic) for a sequence of insertions and deletions. This structure provides efficient implementations for abstract data structures such as associative arrays. The primary step to get the flexibility that we need to guarantee balance in binary search trees is to allow the nodes in our trees to hold more than one.

Let us look at some of the search trees in detail now.

9.5 AVL TREE

AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right sub trees cannot be more than one for all nodes. What if the input to binary search tree comes in a sorted (ascending or descending) manner? It will then look like this:



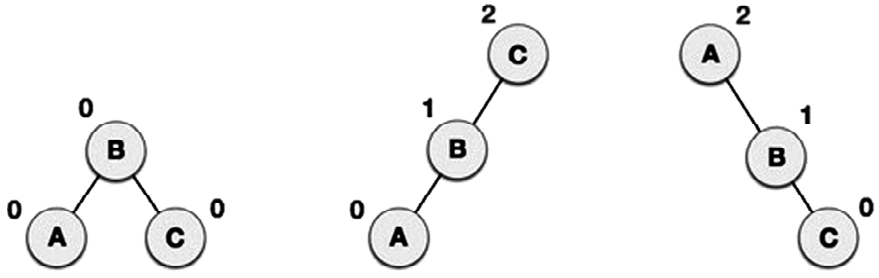
If input 'appears' non-increasing manner If input 'appears' non-decreasing manner

Figure 9.1: Example of Two Binary Search Tree

It is observed that BST's worst-case performance is closest to linear search algorithms, that is O(n). In real-time data, we cannot predict data pattern and their frequencies. So, a need arises to balance out the existing BST.

Named after their inventor **Adelson**, **Velski** & **Landis**, **AVL trees** are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the **Balance Factor**.

In figure 9.2, we can see that the first tree is balanced and the next two trees are not balanced:



Balanced Not balanced Not balanced

Figure 9.2: Balanced and Not Balanced Trees

In the second tree, the left subtree of **C** has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of **A** has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

Balance Factor = height(left-sutree) – height(right-sutree)

If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

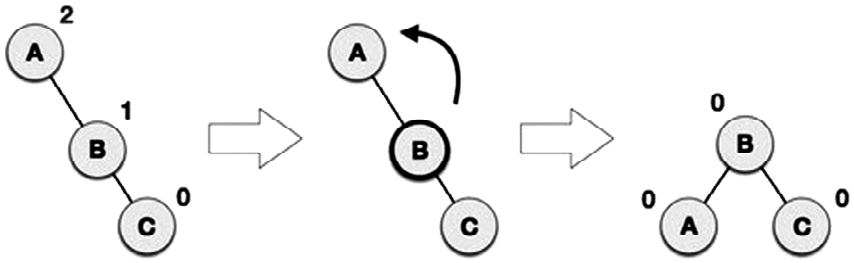
9.5.1 AVL Rotations

To balance itself, an AVL tree may perform the following four kinds of rotations:

- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation

The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. Let us understand these rotations one by one.

- **Left Rotation:** If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation:

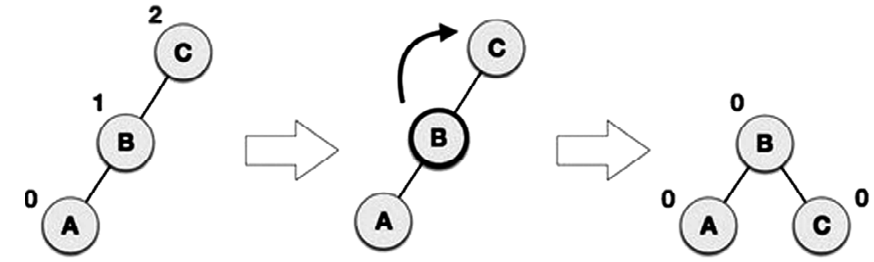


Right unbalanced Tree Left Rotation Balanced

Figure 9.3: Example Showing Left Rotation

In figure 9.3, node **A** has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making **A** the left-subtree of B.

- **Right Rotation:** AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.

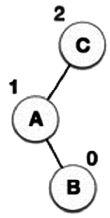
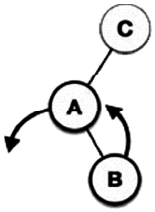
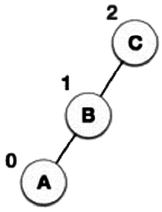


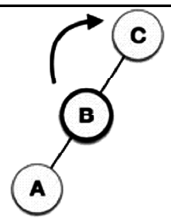
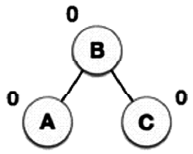
Left unbalanced Tree Right Rotation Balanced Tree

Figure 9.4: Example Showing Right Rotation

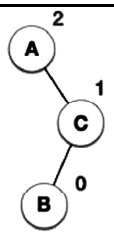
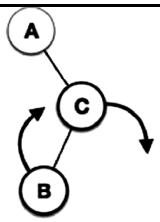
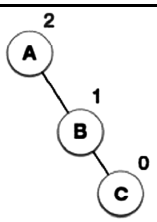
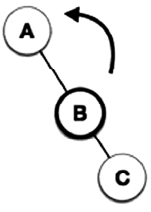
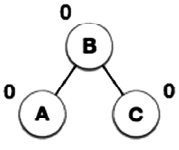
As depicted in figure 9.4, the unbalanced node becomes the right child of its left child by performing a right rotation.

- **Left-Right Rotation:** Double rotations are slightly complex version of already explained versions of rotations. A left-right rotation is a combination of left rotation followed by right rotation. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation.

State	Action
	A node has been inserted into the right subtree of the left subtree. This makes C, an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.
	We first perform the left rotation on the left subtree of C. This makes A, the left subtree of B.
	Node C is still unbalanced, however now, it is because of the left-subtree of the left-subtree.

	We shall now right-rotate the tree, making B the new root node of this subtree. C now becomes the right subtree of its own left subtree.
	The tree is now balanced.

- **Right-Left Rotation:** The second type of double rotation is Right-Left Rotation. This rotation can be define as a combination of right rotation followed by left rotation.

State	Action
	A node has been inserted into the left subtree of the right subtree. This makes A, an unbalanced node with balance factor 2.
	First, we first perform the right rotation along C node, making C the right subtree of its own left subtree B. Now B becomes the right subtree of A.
	Node A is still unbalanced because of the right subtree of its subtree and requires a left rotation.
	A left rotation is performed by making B the new root node of the subtree. A becomes the left subtree of its right subtree B.
	The tree is now balanced.



CHECK YOUR PROGRESS

Q.1: What are the different kinds of rotations performed in AVL tree?

9.6 M-WAY SEARCH TREES

A binary search tree has *one* value in each node and *two* subtrees. This notion easily generalizes to an M-way search tree, which has (M–1) values per node and M subtrees. M is called the *degree* of the tree. A binary search tree, therefore, has degree 2.

In fact, it is not necessary for every node to contain exactly (M–1) values and have exactly M subtrees. In an M-way subtree a node can have anywhere from 1 to (M–1) values, and the number of (non-empty) subtrees can range from 0 (for a leaf) to 1+(the number of values). M is thus a *fixed upper limit* on how much data can be stored in a node.

The values in a node are stored in ascending order, V**1** < V**2** < ... V**k** (k <= M–1) and the subtrees are placed between adjacent values, with one additional subtree at each end. We can thus associate with each value a 'left' and 'right' subtree, with the right subtree of V_2 being the same as the left subtree of V(i+1). All the values in V**1**'s left subtree are less than V**1**; all the values in Vk's subtree are greater than Vk; and all the values in the subtree between V(i) and V(i+1) are greater than V(i) and less than V(i+1).

For example, a 3-way search tree is shown in figure 9.5.

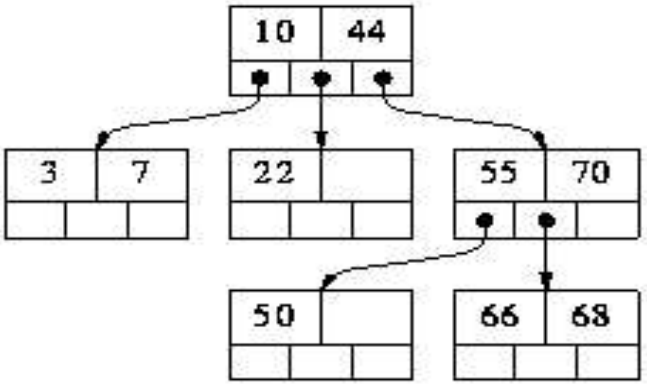


Figure 9.5: 3-Way Search Tree

For examples, it will be convenient to illustrate M-way trees using a small value of M. But bear in mind that, in practice, M is usually very large. Each node corresponds to a physical block on disk, and M represents the maximum number of data items that can be stored in a single block. M is maximized in order to speedup processing: to move from one node to another involves reading a block from disk – a very slow operation compared to moving around a data structure stored in memory.

The algorithm for searching for a value in an M-way search tree is the obvious generalization of the algorithm for searching in a binary search tree. If we are searching for value X and are currently at node consisting of values V1...Vk, there are four possible cases that can arise:

1) If X < V1, recursively search for X in V1's left subtree.
2) If X > Vk, recursively search for X in Vk's right subtree.
3) If X = Vi, for some i, then we are done (X has been found).

The only remaining possibility is that, for some i, Vi < X < V(i+1). In this case recursively search for X in the subtree that is in between Vi and V(i+1).

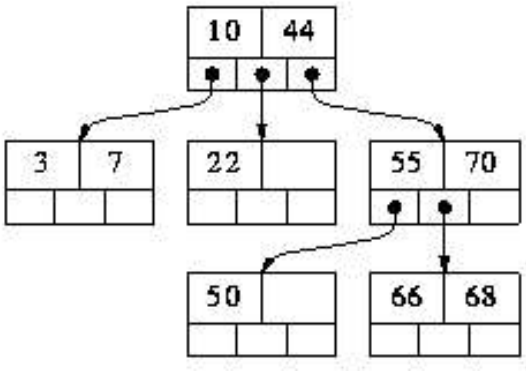


Figure 9.6: Example of M-way Tree

For example, suppose we were searching for 68 in the tree above. At the root, case (2) would apply, so we would continue the search in V2's right subtree. At the root of this subtree, case (4) applies, 68 is between V1=55 and V2=70, so we would continue to search in the subtree between them. Now case (3) applies, 68=V2, so we are done. If we had been searching for 69, exactly the same processing would have occurred down to the last node. At that point, case (2) would apply, but the subtree we want to search in is empty. Therefore, we conclude that 69 is not in the tree.

Similar to the algorithms for binary search trees - insertion and deletion - generalize in a similar way. As with binary search trees, inserting values in ascending order will result in a degenerate M-way search tree; i.e. a tree whose height is O(N) instead of O(logN). This is a problem because all the important operations are O(height), and it is our aim to make them O(logN). One solution to this problem is to *force* the tree to be height-balanced; we sketched this last lecture for binary search trees and now we will examine it in detail for M-way search trees.

9.7 2-3 TREES

2-3 trees are tree data structures in which every internal node (non-leaf node) has either one data element and two children or two data elements and three children. If a node contains one data element leftVal, it has two subtrees (children) namely left and middle. But if a node contains two data elements leftVal and rightVal, it has three subtrees namely left, middle and right.

The main advantage with 2-3 trees is that it is balanced in nature as opposed to a binary search tree whose height in the worst case can be O(n). Due to this, the worst case time-complexity of operations such as search, insertion and deletion is O(log(n)) as the height of a 2-3 tree is O(log(n)).

Search: To search a key K in given 2-3 tree T, we follow the following procedure:

Base cases:

1) If T is empty, return False (key cannot be found in the tree)
2) If current node contains data value which is equal to K, return True.
3) If we reach the leaf-node and it doesn't contain the required key value K, return False.

Recursive Calls:

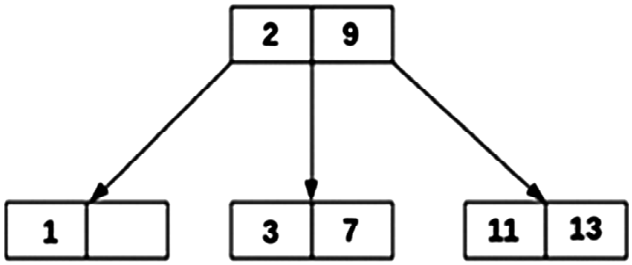
1) If K < currentNode.leftVal, we explore the left subtree of the current node.
2) Else if currentNode.leftVal < K currentNode.rightVal, we explore the right subtree of the current node.

3) Else if K > currentNode.rightVal, we explore the right subtree of the current node.

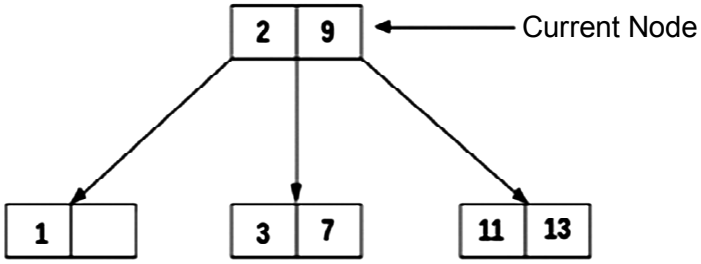
Let us consider the following example.

Example 9.1: Search 5 in the following 2-3 tree.

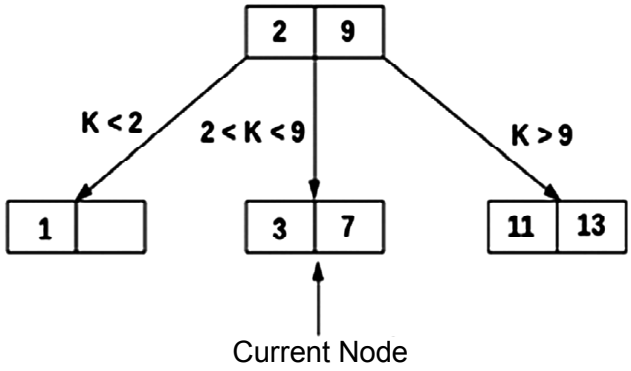
Step 1: Initial condition



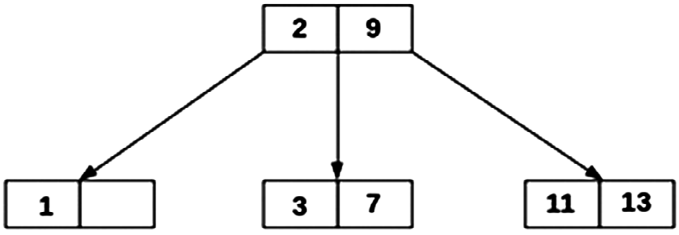
Step 2:



Step 3:



Step 4:



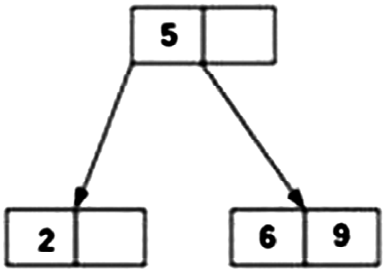
5 Not Found, Return False

Insertion: There are three possible cases in insertion which have been discussed below:

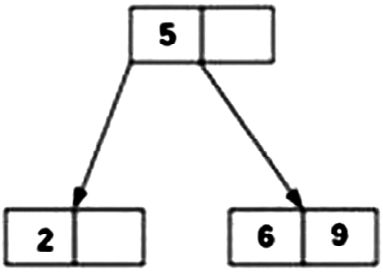
Case 1: Insert in a node with only one data element.

Example 9.2: Insert 5 in the following 2-3 tree.

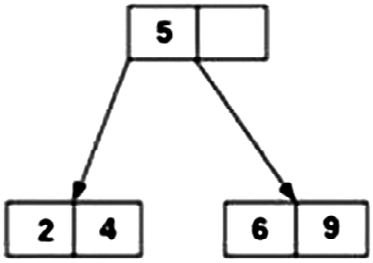
Step 1: Initial condition



Step 2:



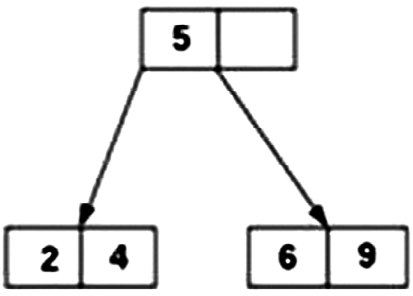
Step 3: After insertion



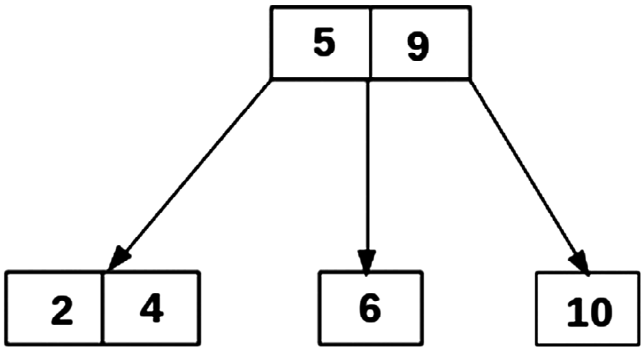
Case 2: Insert in a node with two data elements whose parent contains only one data element.

Example 9.3: Insert 10 in the following 2-3 tree.

Step 1: Initial condition



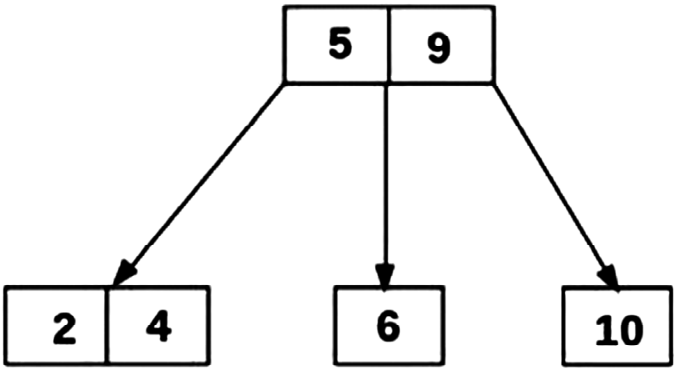
Step 2: Move the middle element to parent and split the current node.



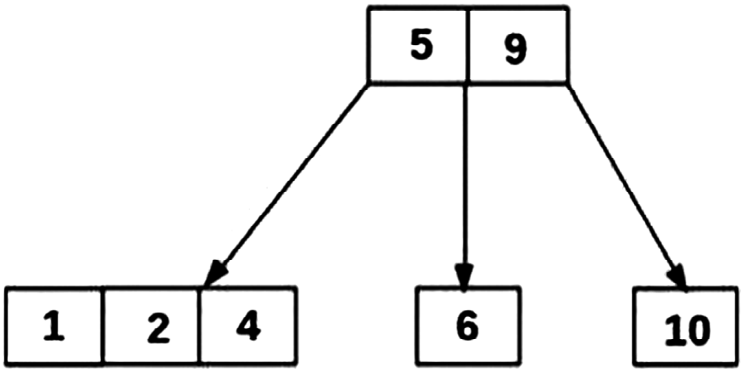
Case 3: Insert in a node with two data elements whose parent also contains two data elements.

Example 9.4: Insert 1 in the following 2-3 tree.

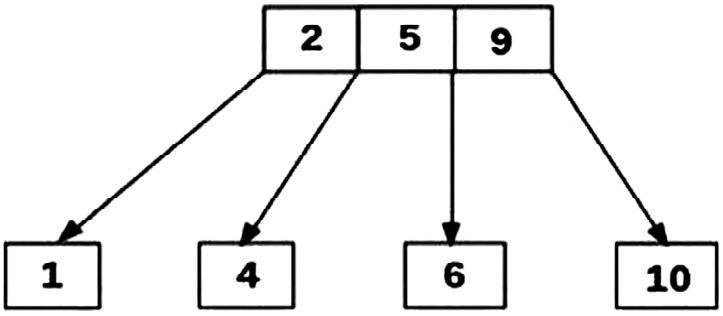
Step 1: Initial condition



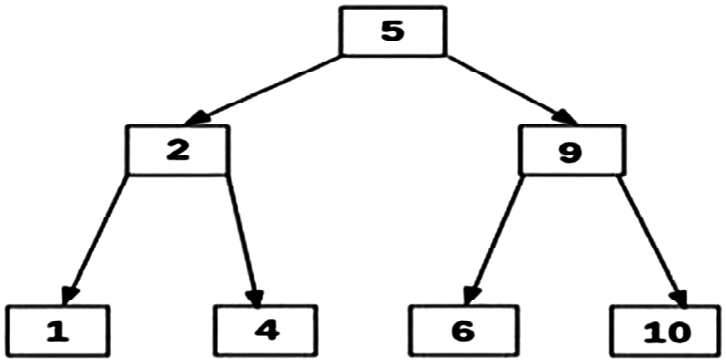
Step 2: Temporary node with 3 data element



Step 3: Move the middle element to the parent and split the current node



Step 4: Move the middle element to the parent and split the current node.



9.8 SPLAY TREES

A splay tree is an efficient implementation of a balanced binary search tree that takes advantage of locality in the keys used in incoming lookup requests. For many applications, there is excellent key locality. A good example is a network router. A network router receives network packets at a high rate from incoming connections and must quickly decide on which outgoing wire to send each packet, based on the IP address in the packet. The router needs a big table (a map) that can be used to look up an IP address and find out which outgoing connection to use. If an IP address has been used once, it is likely to be used again, perhaps many times. Splay trees can provide good performance in this situation. Importantly, splay trees offer amortized O(lg n) performance; a sequence of M operations on an n-node splay tree takes O(M lg n) time.

A splay tree is a binary search tree. It has one interesting difference, however: whenever an element is looked up in the tree, the splay tree

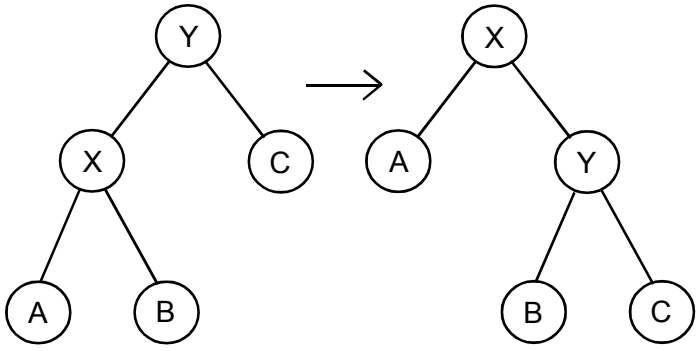
reorganizes to move that element to the root of the tree, without breaking the binary search tree invariant. If the next lookup request is for the same element, it can be returned immediately. In general, if a small number of elements are being heavily used, they will tend to be found near the top of the tree and are thus found quickly.

We have already seen a way to move an element upward in a binary search tree: tree rotation. When an element is accessed in a splay tree, tree rotations are used to move it to the top of the tree. This simple algorithm can result in extremely good performance in practice. Notice that the algorithm requires that we be able to update the tree in place, but the abstract view of the set of elements represented by the tree does not change. This is an example of a benign side effect, because it does not change the value represented by the data structure.

There are three kinds of tree rotations that are used to move elements upward in the tree. These rotations have two important effects: they move the node being splayed upward in the tree, and they also shorten the path to any nodes along the path to the splayed node. This latter effect means that splaying operations tend to make the tree more balanced.

Rotation 1: Simple rotation

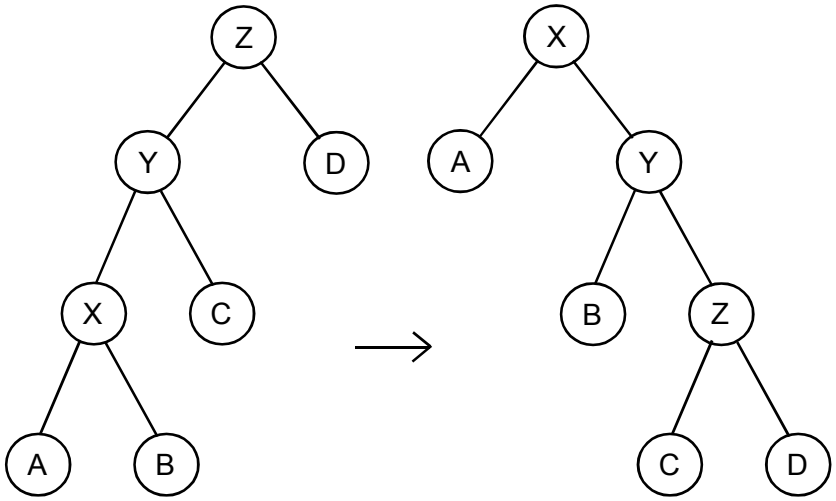
The simple tree rotation used in AVL trees is also applied at the root of the splay tree, moving the splayed node x up to become the new tree root. Here, we have A < x < B < y < C, and the splayed node is either x or y depending on which direction the rotation is:



Rotation 2: Zig-Zig and Zag-Zag

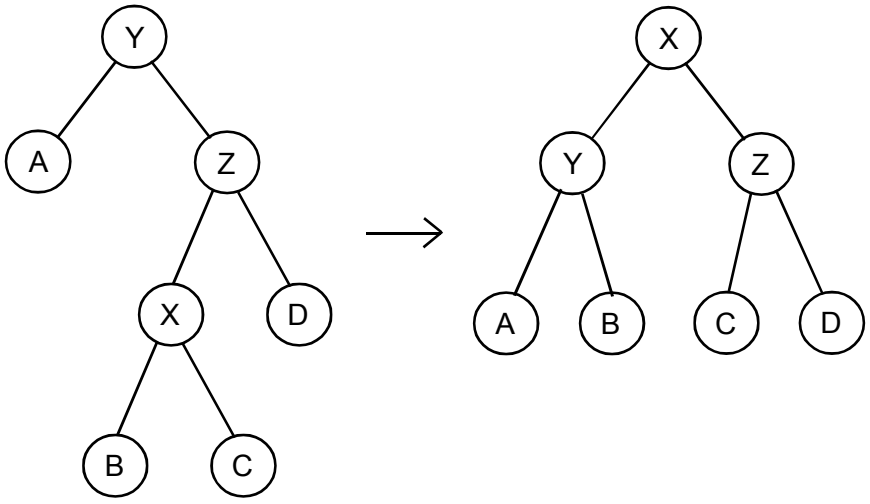
Lower down tree rotations are performed in pairs so that nodes on the path from the splayed node to the root move closer to the root on average.

In the “zig-zig” case, the splayed node is the left child of a left child or the right child of a right child (“zag-zag”).



Rotation 3: Zig-Zag

In the “zig-zag” case, the splayed node is the left child of a right child or vice-versa. The rotations produce a subtree whose height is less than that of the original tree. Thus, this rotation improves the balance of the tree. In each of the two cases shown, y is the splayed node:



The classic version of splay trees is an imperative data structure in which tree rotations are done by imperatively updating pointers in the nodes. This data structure can only implement a mutable set/map abstraction because it modifies the tree destructively.

9.9 HASHING

Suppose we want to design a system for storing employee records keyed using phone numbers. And we also want the following queries to be performed efficiently:

1) Insert a phone number and corresponding information.
2) Search a phone number and fetch the information.
3) Delete a phone number and related information.

We can think of using the following data structures to maintain information about different phone numbers.

1) Array of phone numbers and records.
2) Linked List of phone numbers and records.
3) Balanced binary search tree with phone numbers as keys.
4) Direct Access Table.

For arrays and linked lists, we need to search in a linear fashion, which can be costly in practice. If we use arrays and keep the data sorted, then a phone number can be searched in O(log n) time using Binary Search, but insert and delete operations become costly as we have to maintain sorted order. With balanced binary search tree, we get moderate search, insert and delete times. All of these operations can be guaranteed to be in O(logn) time.

Another solution that one can think of is to use a direct access table where we make a big array and use phone numbers as index in the array. An entry in array is NIL if phone number is not present, else the array entry stores pointer to records corresponding to phone number. Time complexity wise this solution is the best among all because we can do all operations in O(1) time. For example to insert a phone number, we create a record with details of the given phone number, use phone number as index and store the pointer to the created record in table.

This solution has many practical limitations. First problem with this solution is extra space required is huge. For example, if phone number is n digits, we need O(m*10n) space for table where m is size of a pointer to

record. Another problem is an integer in a programming language may not store n digits.

Due to above limitations Direct Access Table cannot always be used. **Hashing** is the solution that can be used in almost all such situations and if performs extremely well compared to above data structures like Array, Linked List, Balanced BST in practice. With hashing we get O(1) search time on average (under reasonable assumptions) and O(n) in worst case.

Hashing is an improvement over Direct Access Table. The idea is to use hash function that converts a given phone number or any other key to a smaller number and uses the small number as index in a table called hash table.

Hash Function: A function that converts a given big phone number to a small practical integer value. The mapped integer value is used as an index in hash table. In simple terms, a hash function maps a big number or string to a small integer that can be used as index in hash table.

A good hash function should have the following properties:

1) Efficiently computable.
2) Should uniformly distribute the keys (Each table position equally likely for each key)

For example, for phone numbers a bad hash function is to take the first three digits. A better function is to consider the last three digits. Please note that this may not be the best hash function. There may be better ways.

Hash Table: An array that stores pointers to records corresponding to a given phone number. An entry in hash table is NIL if no existing phone number has hash function value equal to the index for the entry.

Collision Handling: Since a hash function gets us a small number for a big key, there is a possibility that two keys may result in the same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some collision handling technique. Following are the ways to handle collisions:

Chaining: The idea is to make each cell of hash table point to a linked list of records that have the same hash function value. Chaining is simple, but it requires additional memory outside the table.

Open Addressing: In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we examine table slots one by one until the desired element is found or it is clear that the element is not in the table.



CHECK YOUR PROGRESS

Q.2: What are 2-3 trees?

Q.3: Define splay trees.



9.10 LET US SUM UP

- AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right sub trees cannot be more than one for all nodes.
- 2-3 trees are tree data structures in which every internal node (non-leaf node) has either one data element and two children or two data elements and three children.
- A splay tree is a binary search tree.
- In open addressing, all elements are stored in the hash table itself.
- Each table entry contains either a record or NIL.



9.11 FURTHER READING

1) Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. (2009). *Introduction to Algorithms*. MIT press.



9.12 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: The different kinds of rotation performed in AVL tree are: Left rotation, Right rotation, Left-Right rotation and Right-Left rotation.

Ans. to Q. No. 2: 2-3 trees are tree data structures in which every internal node (non-leaf node) has either one data element and two children or two data elements and three children. If a node contains one data element leftVal, it has two subtrees (children) namely left and middle. Whereas if a node contains two data elements leftVal and rightVal, it has three subtrees namely left, middle and right.

Ans. to Q. No. 3: A splay tree is a binary search tree. It has one interesting difference, however: whenever an element is looked up in the tree, the splay tree reorganizes to move that element to the root of the tree, without breaking the binary search tree invariant. If the next lookup request is for the same element, it can be returned immediately. In general, if a small number of elements are being heavily used, they will tend to be found near the top of the tree and are thus found quickly.



9.13 MODEL QUESTIONS

Q.1: Describe how rotations are done in AVL trees.

Q.2: Describe M-way trees.

Q.3: Explain splay trees using example.

Q.4: Define 2-3 trees.

Q.5: Describe the concept of hashing.

*** ***** ***

UNIT 10: RED BLACK TREE

UNIT STRUCTURE

10.1 Learning Objectives

10.2 Introduction

10.3 Red Black Tree

10.3.1 Properties of Red Black Tree

10.3.2 Height of a Red Black Tree

10.4 Balancing of Red Black Tree

10.5 Red Black Tree Rotations

10.5.1 Left Rotation

10.5.2 Right Rotation

10.6 Insertion in Red Black Tree

10.7 Deletion in Red Black Tree

10. 8 Let Us Sum Up

10.9 Further Reading

10.10 Answers to Check Your Progress

10.11 Model Questions

10.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- define red black tree
- explain the properties of red black tree
- define and calculate the height of a red black tree
- describe how left rotation and right rotation work in red black tree
- explain the insertion of a node in a red black tree
- explain the deletion of a node in a red black tree.

10.2 INTRODUCTION

In the previous units we have learnt about different types of trees like binary tree, binary search tree, AVL trees and multi-way trees. In this unit we will help you to learn about the basic concepts of red black tree.

This unit will give you an idea about the main properties of red black tree. We will also learn about the rotations of a red black tree and why these rotations are necessary. It will also cover the insertion procedure of a node in a red black tree and how to handle various situations while inserting a node besides explaining how to maintain the properties of a red black tree during insertion operation. This unit also contains different situations of deletion process for red black trees. In the next unit, we will explore B-tree in detail.

10.3 RED BLACK TREE

A **red–black tree** is a self balancing binary search tree. Each node of the binary tree has an extra bit, and that bit is often referred to as the color (red or black) of the node. A red black tree is a binary search tree having an additional property of color in every node, which can be either RED or BLACK. If a node of the binary search tree does not have a left and/or right child then we add an external node. External nodes are marked as Null. One key concept of red black tree is black depth of an external node which can be defined as the number of black ancestors it has. Every external node has the same black depth which is also referred to as black height of the tree.

10.3.1 Properties of Red Black Tree

A red black tree is a binary search tree having the following properties:

1) Every node is either red or black
2) The root is black
3) Every leaf (NULL) is black
4) A red node must have black children.
5) For each node, all paths from the node to descendant leave contain the same number of black nodes.

10.3.2 Height of a Red Black Tree

1) Let **h** be the black height of red black tree on **n** nodes.
2) **n** is smallest when all nodes are black. In this case tree is a complete binary tree of height **h** and $\mathbf{n = 2^h - 1}$
3) **n** is largest when alternate levels of the tree are red. In this case the height of the tree is **2h** and $\mathbf{n= 2^{2h} - 1}$
4) Hence $\mathbf{\log_4 n < h < 1 + \log_2 n}$

The black height of a red black tree can be defined as the number of black nodes on any path from any node x to a leaf. The black height of all external nodes is same, since all descending paths from the node have the same number of black nodes. It can be defined that the black height of a red black tree to be the black height of its root.

10.4 BALANCING OF RED BLACK TREE

Red black trees are self balancing trees. But AVL trees are more balanced than red black trees. And AVL trees need more rotations to balance the tree when nodes are inserted or deleted. For lookup intensive tasks, AVL tree is preferable. But, if addition and deletion of nodes occur very frequently, then a Red Black Tree's performance is better. It requires less rotation to balance the overall height of the tree.

In red black tree, there are two methods to balance the tree when imbalances occur due to insertion of new nodes. These two methods are-

1) Recoloring
2) Rotations

Firstly, recoloring is tried to balance the tree, but when it can't be balanced with this method then rotations are done to balance the tree. When any new node is inserted in a red black tree then the properties of a red black tree get violated and the need arises to balance the tree by maintaining the properties of red black tree. The balancing of red black tree is explained in insertion and deletion part of this unit.



CHECK YOUR PROGRESS

Q.1: Define red black tree.

Q.2: In a red black tree, if a node is red, then both its children are .....

Q.3: The root of a red black tree is always ..... (black/red)

10.5 ROTATIONS

The two basic operations of red black tree are Tree-Insert and Tree-Delete. When these two operations are performed on red black tree then it modifies the tree and as result it leads to the violation of the properties of the red black tree. To restore these properties, there is a need of changing the colors of some nodes in the tree and also pointer structures. The change in the pointer structure is made through rotations.

Rotation is an operation in a search tree which preserves the binary search tree property. There are two main rotations, namely– Left rotation and Right rotation. Rotations alter the structure of a tree by rearranging the sub-trees. The objective of rotation is to decrease the height of the tree. It does not affect the order of elements.

10.5.1 Left Rotation

While doing a left rotation on a node x, it is considered that its right child y is not NULL. Let x is a node in the tree and its right child is not NULL. The left rotation pivots around the link from x to y. The left rotation is performed with x as the root. After this operation y will become the new root of the sub-tree and x will be shifted to y's left and y's left child will become x's right child. This operation results in a rotation of the tree in the anti-clockwise direction. The left rotation of a red black tree is discussed with the help of the following procedure. The pseudocode for LEFT-ROTATE assumes that right [x] $\neq$ NULL and the root's parent is NULL.

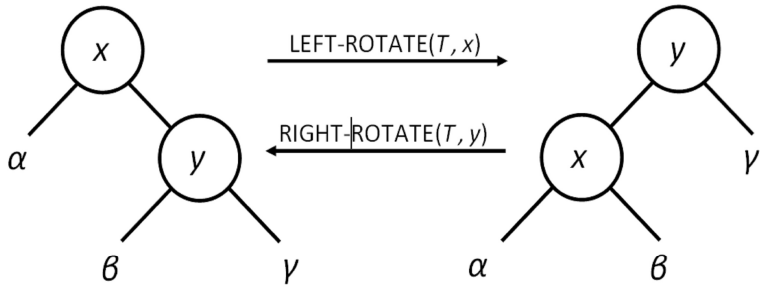


Figure 10.1: Rotations in a Red Black Tree

Algorithm 10.1: LEFT-ROTATE (T, x)

1) y ← right [x]
2) right [x] ← left [y]
3) p [left [y]] ← x
4) p [y] ← p [x]
5) if p [x] = Null
6) then root [T] ← y
7) else if x = left [p [x]]
8) then left [p [x]] ← y
9) else right [p [x]] ← y
10) left [y] ← x
11) p [x] ← y

An example of Left rotation is explained in the figure 10.2.

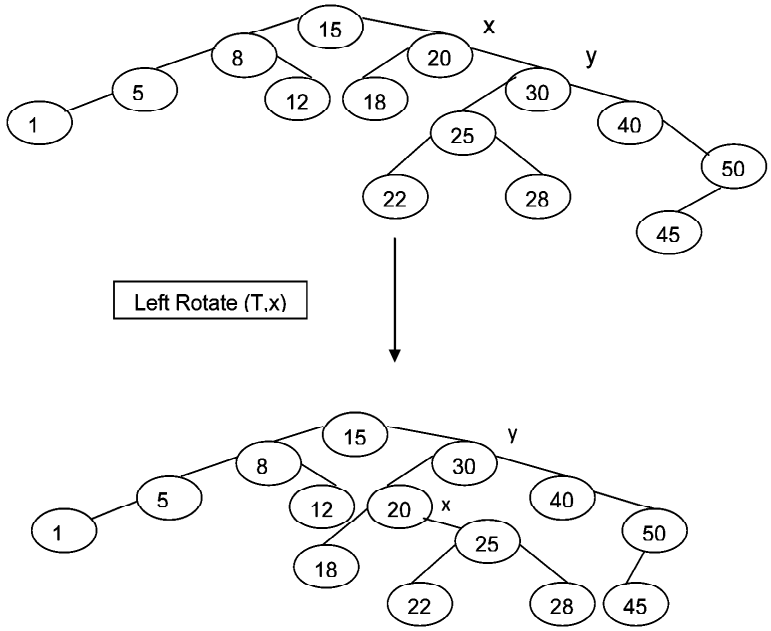


Figure 10.2: Example to Explain the Procedure LEFT-ROTATE

10.5.2 Right Rotation

The inverse operation to the left rotation is the right rotation, which results in a movement in a clockwise direction. The right rotation operation is performed with y as the root. The right rotation of a red black tree is discussed with the help of the following procedure.

Algorithm 10.2: RIGHT-ROTATE(T,y)

1) x ← left [y]
2) left [y] ← right [x]
3) p [right [x]] ← y
4) p [x] ← p [y]
5) p [y] = Null
6) then root [T] ← x
7) else if y = right [p [y]]
8) then right [p [y]] ← x
9) else left [p [y]] ← x
10) right [x] ← y
11) p [y] ← x

10.6 INSERTION IN RED BLACK TREE

A new node in red black tree is inserted as it is done in binary search tree insertion but with a difference that the node is colored red. The time required for inserting a node into an n-node red black tree is O(lg n).

The insertion procedure INSERT(T, z) is used to insert node z into the tree T similar to binary search tree, and then it is colored red. In binary search tree a new node is inserted at leaf. Therefore firstly the leaf node is found and then the new node is inserted as a child of the leaf node. After that another method INSERT-FIXUP() is called for preservation of the red-black tree properties, which will recolor nodes and perform rotations. The procedure INSERT (T, z) inserts node z, whose key is assumed to have already been filled in, into the red-black tree T. Following is the code for inserting a node:

Algorithm 10.3: INSERT(T, z)

1) y ← null
2) x ← root[T]
3) while x ≠ null
4) do y ← x
5) if key[z] < key[x]
6) then x ← left[x]
7) else x ← right[x]
8) p[z] ← y
9) if y = null
10) then root[T] ← z
11) else if key[z] < key[y]
12) then left[y] ← z
13) else right[y] ← z
14) left[z] ← null
15) right[z] ← null
16) color[z] ← RED
17) INSERT-FIXUP(T,z)

In the red black tree insertion procedure, the left[z] and right[z] is set to null. The color of z node is set to red. When the newly added node z is colored red, it will lead to violation of one of the red black properties, therefore to restore the red-black properties, INSERT-FIXUP(T,z) is invoked.

Algorithm 10.4: INSERT-FIXUP(T, z)

1) while color[p[z]] = RED
2) do if p[z] = left[p[p[z]]]
3) then y ← right[p[p[z]]]
4) if color[y] = RED
5) then color[p[z]] ← BLACK Case 1
6) color [y] ← BLACK Case 1
7) color [p[p[z]]] ← RED Case 1
8) z ← p[p[z]] Case 1
9) else if z = right[p[z]]
10) then z ← p[z] Case 2

11) LEFT-ROTATE(T, z) Case 2
12) color[p[z]] ← BLACK Case 3
13) color[p[p[z]]] ← RED Case 3
14) RIGHT-ROTATE(T, p[p[z]]) Case 3
15) else (same as then clause with “right” and “left” exchanged) color[root[T]] ← BLACK

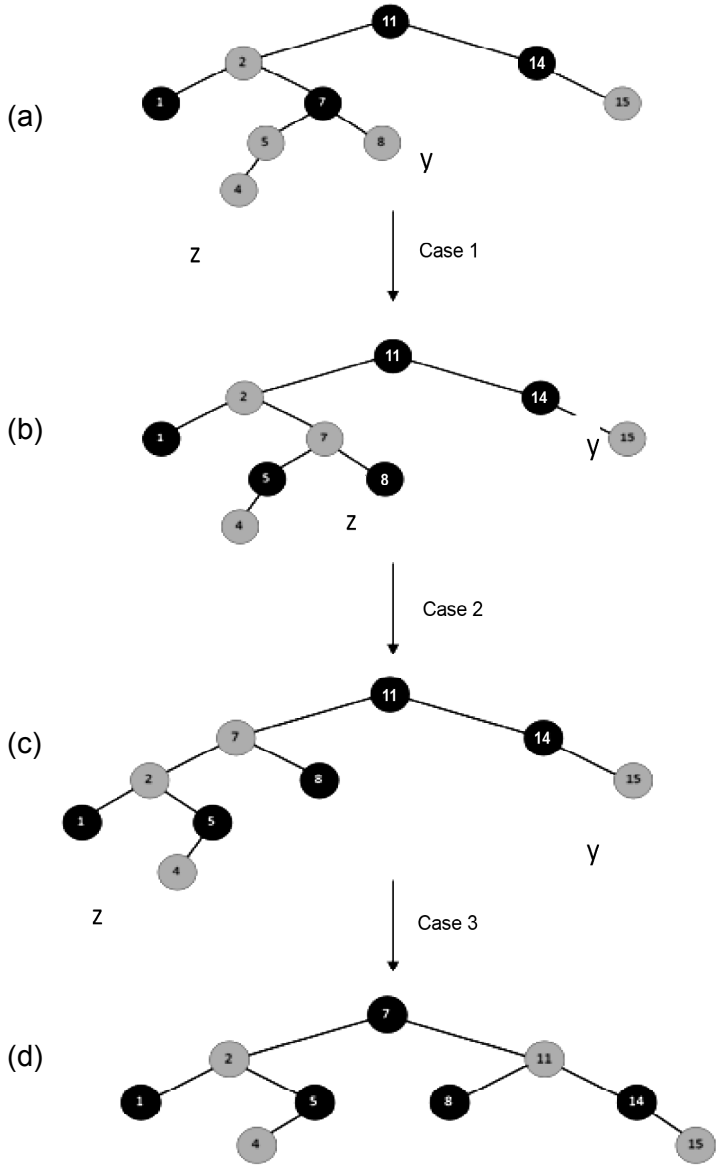


Figure 10.3: The operation of INSERT-FIXUP()

(a) Case1 (b) Case 2 (c) Case 3 (d) Final Red Black tree

The objective of the procedure INSERT-FIXUP() are as follows:

i) To determine what violations of the red-black properties are introduced in INSERT() when the node z is inserted and colored red.

ii) To examine the overall objective of the while loop.

iii) To explore all the three cases of the while loop.

With reference to the figure 10.3.a, If a node z (node with key 4) is inserted in the left of the node with key 5, then the property 4 is violated, which says a red node cannot have a red child. Since z's uncle y (node with key 8 in Figure 10.3.a) is red, therefore case 1 in the code is applicable here. To resolve this issue, nodes are recolored and z is moved up the tree, resulting in the tree shown in Figure 10.3.b. Now z (node with key 7 in Figure 10.3.b) and its parent (node with key 2 in Figure 10.3.b) are both red, but z's uncle y (node with key 14 in Figure 10.3.b) is black. Since z is the right child of p[z], case 2 is applicable. A left rotation is performed, and the tree that results is shown in Figure 10.3.c. After this, z has become the left child of its parent, and case 3 is applicable. A right rotation is performed to produce the tree in Figure 10.3.d, which is the final valid red-black tree.

At the beginning of each iteration of the while loop in the procedure INSERT-FIXUP(), following conditions remain true:

a) The newly inserted node z is red,

b) If the parent of z, p[z] is root, then p[z] is black.

c) At this stage, there is a possibility that there may be a violation of either property 2 or property 4. Property 2 will violate if z becomes root and it is red. Property 4 will violate when z and the parent of z, p[z] are red.

The three steps of the loop are discussed below:

1) Initialization: Before addition of the red node z, all the properties of the red black tree were preserved. After adding the z node, the method INSERT-FIXUP() is called–

a) When INSERT-FIXUP is called, the red node z was added.

b) If p[z] is the root, then p[z] is black and did not change before calling the method INSERT- FIXUP().

c) Properties 1, 3, and 5 were maintained when INSERT-FIXUP () method is called.

If the property 2 doesn't hold, then it means that the newly added red node z is the only internal node in the tree. If there is a violation of property 4, then because the children of node z are null and black; the tree had no other violations prior to z being added, the violations must be because both z and p[z] are red.

2) **Termination:** The loop terminates when p[z] is black. Thus, there is no violation of property 4 at loop termination. The other properties except property 2 will hold by the loop invariant; only property 2 might get violated. When INSERT-FIXUP() is called and its execution is over then all the properties of red-black hold.

3) **Maintenance:** In the while loop 6 cases can be considered, but 3 of these cases are symmetric to the other 3, which depends on the position of z in the tree, whether parent of z, p[z] is a left child or a right child of z's grandparent p[p[z]], which is determined in line 2 of the code. The code is applicable only for the situation in which p[z] is a left child. The node p[p[z]] exists, if p[z] is the root and is black. Since we enter loop iteration only if p[z] is red, we know that p[z] cannot be the root. Hence, p[p[z]] exists. The following cases are discussed in detail.

Case 1: z's uncle is red

Case 1 is applicable when both p[z] and y are red. Since p[p[z]] is black, both p[z] and y is colored black, thereby fixing the problem of property 4 and p[p[z]] is colored red, to maintain the property 5. Then while loop is repeated considering grandparent of z, p[p[z]] as the new node z. The z will move up by 2 levels in the tree. The same action will be taken whether z is a right child or z is a left child.

Case 2: z is a right child and z's uncle y is black

In both the cases (Case 2 and Case 3), the color of z's uncle y is black. Both the cases are similar but with a difference whether z is a right child or left child of its parent p[z]. In case 2, a left rotation is performed to transform the situation into case 3, in which node z is

a left child. Since both z and the parent of z, p[z] are red, the rotation doesn't affect the black-height of nodes and property 5. Since z's uncle y is black, otherwise case 1 would have been executed. Additionally, it is considered that the node p[p[z]] exists and after moving z, 1 level up in line 10 of the code and again one level down in the line 11, doesn't alter the identity of p[p[z]].

Case 3: z is a left child and z's uncle y is black

In the case 3, a right rotation has been performed and some color changes are made, which preserve the property 5, and it doesn't have two red nodes in a row. The body of the while loop is not executed another time, since p[z] is black. In the following diagrams these cases are explained:

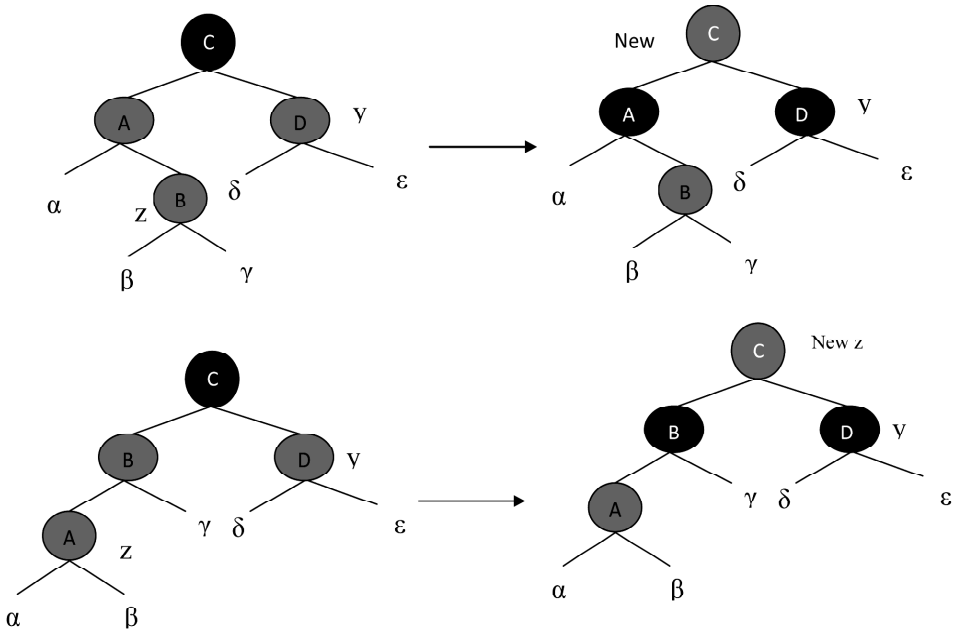


Figure 10.4: Case 1 of the procedure INSERT-FIXUP()

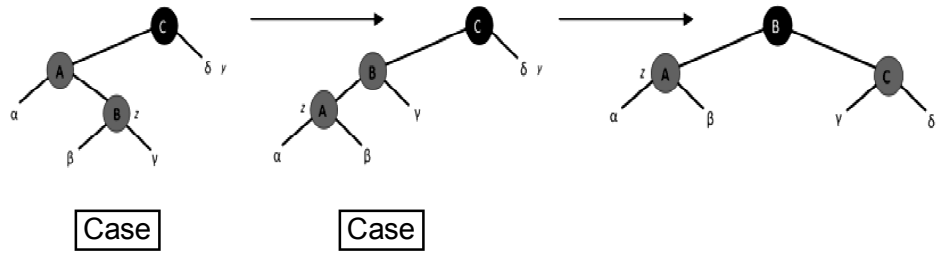


Figure 10.5: Case 2 and Case 3 of the Procedure INSERT-FIXUP()



CHECK YOUR PROGRESS

Q.4: The color of a new node to be inserted in a red black tree is .....

Q.5: What is the need of the procedure INSERT-FIXUP()?

Q.6: What are the properties of a red black tree?

10.7 DELETION IN RED BLACK TREE

Delete operation in red black tree is similar to the delete operation of standard binary search tree. In binary search tree deletion process, if the node to be deleted is a leaf node then it is directly removed from tree. If the node to be deleted has one child then we remove the node and replace it with its child. When a node to be deleted has two children, then we need to find it's either inorder successor or inorder predecessor node and copy it's value to the node. After that delete the inorder successor or inorder predecessor node. The deletion process of a node takes time O (lg n). The procedure DELETE() is used for deleting a node from a red black tree. After deleting a node from the tree, the procedure DELETE() calls a procedure DELETE-FIXUP() which changes colors and performs rotations to restore the red-black properties. The code for deleting a node from a red black tree T is given in algorithm 10.5.

Algorithm 10.5: DELETE (T, z)

1) if left[z] = null or right[z] = null
2) then y ← z
3) else y ← TREE-SUCCESSOR(z)
4) if left[y] ≠ null
5) then x ← left[y]
6) else x ← right[y]
7) p[x] ← p[y]
8) if p[y] = null
9) then root[T] ← x
10) else if y = left[p[y]]

11) then left[p[y]] ← x
12) else right[p[y]] ← x
13) if y ≠ z
14) then key[z] ← key[y]
15) copy y' s satellite data into z
16) if color[y] = BLACK
17) then DELETE-FIXUP(T, x)
18) return y

If y is red, the red-black properties still hold when y is deleted, for the following reasons:

1) No black-heights in the tree have changed.
2) No red nodes have been made adjacent.
3) Y could not have been the root if it was red, so the root remains black.

The DELETE-FIXUP() method is provided with node x as the parameter which may be either the node that was y's sole child before y was deleted if y had a child that was not the external node, or if y had no children, x is the external node. If x is the external node then the parent of y becomes the parent of x. The procedure DELETE-FIXUP() restores the red-black properties.

Algorithm 10.6: DELETE-FIXUP(T, x)

1) while x ≠ root[T] and color[x] = BLACK
2) do if x = left[p[x]]
3) then w ← right[p[x]]
4) if color[w] = RED
5) then color [w] ← BLACK Case 1
6) color[p[x]] ← RED Case 1
7) LEFT-ROTATE(T, p[x]) Case 1
8) w ← right[p[x]] Case 1
9) if color[left[w]] = BLACK and color[right[w]] = BLACK
10) then color[w] ← RED Case 2
11) x ← p[x] Case 2
12) else if color[right[w]] = BLACK
13) then color[left[w]] ← BLACK Case 3

14) color[w] ← RED Case 3
15) RIGHT-ROTATE(T, w) Case 3
16) w ← right [p[x]] Case 3
17) color[w] ← color[p[x]] Case 4
18) color[p[x]] ← BLACK Case 4
19) color[right[w]] ← BLACK Case 4
20) LEFT-ROTATE(T, p[x]) Case 4
21) x ← root[T] Case 4
22) else (same as then clause with “right” and “left” exchanged)
23) color[x] ← BLACK

If the deleted node y in DELETE() method is black, three problems may arise.

i) if y had been the root and a red child of y becomes a new root, then property 2 is violated.

ii) if both x and p[y] were red, then property 4 is violated.

iii) If y is removed then it will cause to have one fewer black node in any path that previously contained y. Thus, property 5 is now violated by any ancestor of y in the tree. It can be corrected by mentioning that node x has an “extra” black. To preserve the property 5, 1 needs to be added to the count of black nodes on any path that contains x. When black node y is deleted, its blackness is pushed onto its child. After this property 1 is violated because node x is neither red nor black. Now node x is either “double black” or “red-and-black” and the count of black nodes on paths containing x increases by one or two respectively. The color attribute of x will be either RED (if x is red-and-black) or BLACK (if x is double black).

The while loop of the procedure DELETE-FIXUP() targets to move the extra black, up the tree until the following conditions:

1) x points to a red and black node
2) x points to the root
3) Suitable rotations and recoloring can be performed.

Within the while loop, x always points to a non root double black node. Line 2 of the code determines whether x is a left child or a right child

of its parent p[x]. A pointer w is maintained to the sibling of x. Since node x is double black, node w cannot be NULL [T].

The four cases in the code are explained below. In all the cases, to preserve the property 5 i.e. from the root to each of the subtrees α, β, ... the same number of black nodes is maintained by the transformation. Thus, if property 5 holds before the transformation, it continues to hold afterward.

Case 1: x's sibling is red

With reference to the figure 10.6.a, this situation arises when node w and the sibling of node x, is red. Since w must have black children, the colors of w and p[x] are interchanged and then a left rotation on p[x] is performed without violating any of the red-black properties. The new sibling of x, which is one of w's children prior to the rotation, is now black, and thus case 1 has been converted into case 2, 3, or 4. Cases 2, 3 and 4 is applicable when node w is black; they are different from each other by the colors of w's children.

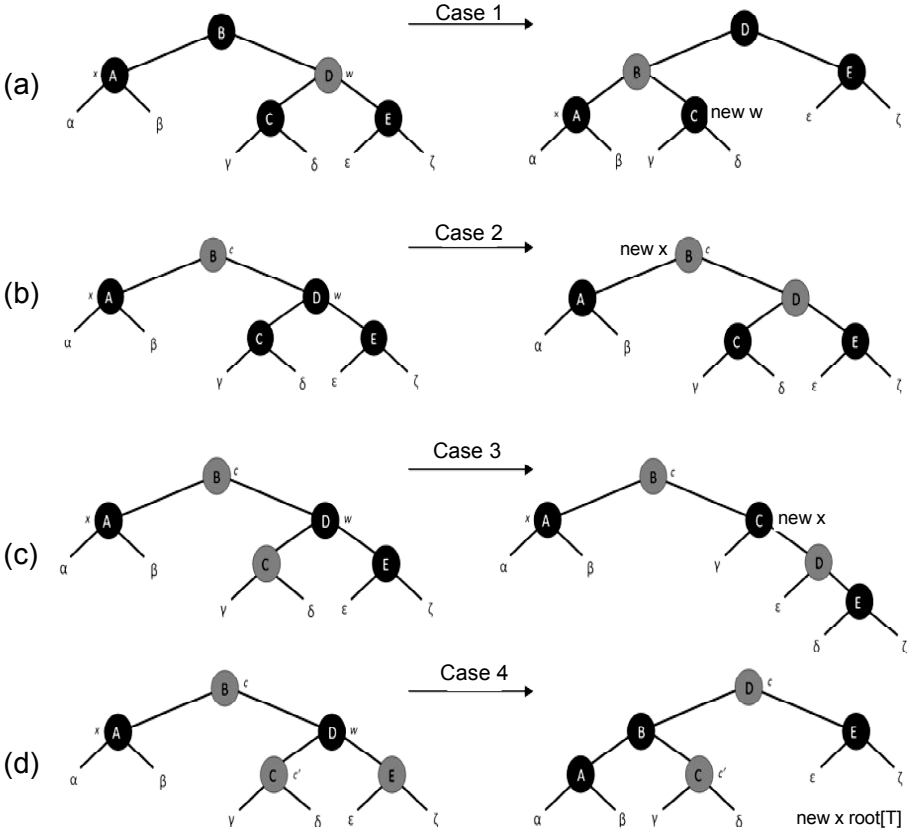


Figure 10.6: Example Showing Cases in the While Loop of the Procedure DELETE-FIXUP()

Case 2: x's sibling w is black, and both w's children are black

Here, both of w's children are black and w is also black, one black is removed from both x and w and as a result x will have one black and w will be red. To compensate for removing one black from x and w, an extra black is added to p[x], which was originally either red or black. It is done by repeating the while loop with p[x] as the new node x. When case 2 is entered through case 1, the new node x is red-and-black, since the original p[x] was red. The color of the new node x is RED, and the loop terminates when it checks the loop condition. The new node x is then colored (single) black.

Case 3: x's sibling w and w's right child is black, w's left child is red

This situation arises when w and its right child is black, its left child is red. The colors of w and its left child left[w] are interchanged and a right rotation on w is performed without violating red black properties. The new sibling w of x is now a black node with a red right child, and thus transformed case 3 into case 4.

Case 4: x's sibling w is black, and w's right child is red

This situation arises when node x's sibling w is black and w's right child is red. The extra black on x is removed by making some changes in the color and performing a left rotation on p[x], making it single black and red-black properties are maintained. When x is set as root then the while loop terminates after testing the loop condition.



CHECK YOUR PROGRESS

Q.7: The time duration of deletion process in a red black tree is .....

Q.8: If a deleted node y is black and y had been the root and a red child of y becomes a new root, then which property of red black tree will be violated?

Q.9: When a black node y is deleted, its blackness is pushed onto its ..... (child/parent)



10.8 LET US SUM UP

- A red–black tree is a kind of self balancing binary search tree. Each node of the binary tree has an extra bit, and that bit is often interpreted as the color (red or black) of the node
- A red black tree is a binary search tree having the following properties:
  1) Every node is either red or black
  2) The root is black
  3) Every leaf (NULL) is black
  4) If a node is red, then both its children are black.
  5) For each node, all paths from the node to descendant leaves contain the same number of black nodes.
- Rotation is an operation in a search tree which preserves the binary search tree property. There are two main rotations namely- Left rotations and Right rotations.
- The time required for inserting a node into an n-node red black tree is O(lg n).
- A new node is inserted in red black tree as it is done in binary search tree insertion and but it is colored red.
- Deletion of a node take time O(lg n). After deleting a node the procedure DELETE() calls a procedure DELETE-FIXUP() which changes colors and performs rotations to restore the red-black properties.



10.9 FURTHER READING

1) Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. (2009). *Introduction to Algorithms*. MIT Press.



10.10 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: A red black tree is a binary search tree having an additional property of color in every node, which can be either RED or BLACK.

Ans. to Q. No. 2: In a red black tree, If a node is red, then both its children are black.

Ans. to Q. No. 3: The root of a red black tree is always black.

Ans to Q No 4: The color of a new node to be inserted in a red black tree is red.

Ans. to Q. No. 5: After deleting a node the procedure DELETE( ) calls a procedure DELETE-FIXUP( ) which changes colors and performs rotations to restore the red-black properties.

Ans. to Q. No. 6: A red black tree is a binary search tree having the following properties:

a) Every node is either red or black
b) The root is black
c) Every leaf (NULL) is black
d) If a node is red, then both its children are black.
e) For each node, all paths from the node to descendant leaves contain the same number of black nodes

Ans. to Q. No. 7: The time duration of deletion process in a red black tree is O(lg n).

Ans. to Q. No. 8: If a deleted node y is black and y had been the root and a red child of y becomes a new root, then property 2 of red black tree will be violated.

Ans. to Q. No. 9: When a black node y is deleted, its blackness is pushed onto its Child.



10.11 MODEL QUESTIONS

Q.1: What is a red black tree? Explain its properties.

Q.2: Write the pseudocode for right rotation in a red black tree.

Q.3: What is the use of recoloring of nodes in a red black tree?

Q.4: Show the insertion procedure of a new node in a red black tree with the help of an example.

Q.5: Write the psedocode for deleting a node from a red black tree.

Q.6: Explain the insertion procedure when the node z is to be inserted and z's uncle y is black and z is a left child.

Q.7: With the help of an example discuss the left rotation process of a red black tree.

Q.8: Construct a red black tree by inserting the keys in the following sequence into an initially empty red black tree: 13, 10, 8, 3, 4 and 9. Show each step.

Q.9: What are three issues that may arise when a black node is deleted from a red black tree?

Q.10: Write the pseudocode for RIGHT-ROTATE().

*** ***** ***

UNIT 11: B-TREE

UNIT STRUCTURE

11.1 Learning Objectives

11.2 Introduction

11.3 B-Tree

11.3.1 Properties of B-Tree

11.4 Insertion in B-Tree

11.5 Deletion from B-Tree

11.6 B+ Tree

11.6.1 Insertion in B+ Tree

11.7 External Sorting

11.8 Let Us Sum Up

11.9 Further Reading

11.10 Answers to Check Your Progress

11.11 Model Questions

11.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- define B-Tree and B+ Tree
- describe properties of B-Tree
- explain the insertion technique of keys in the B-tree
- explain the techniques to delete the keys from a B-Tree
- describe the insertion algorithm into the B+ Tree
- describe the concept of external sorting.

11.2 INTRODUCTION

In the previous unit, we have learned about red black trees. We have also learned how to insert and delete elements in a red black tree. In this unit, we will explore the basic knowledge of a B-tree as well as its Its definition and properties. We will also learn how to insert new keys into a B-tree and also deletion of keys from a B-tree. Besides, in this unit different

cases of the deletion process in a B-tree are discussed in detaild with the help of examples. On the whole, the unit will give you the basic concept of B+ tree and its insertion algorithms based on different conditions. You will also learn about the concept and applications of external sorting. In the next unit, we will explore different advance sorting and selection techniques.

11.3 B-TREE

B-tree is a balanced search tree which is designed to work on magnetic disks and other direct access secondary storage devices. B-tree is similar to red black trees, but it is better at minimizing the disk input output operations. B-tree is different from red black tree in the manner that B-tree has many children, which means the branching factor of a B-tree is very large. The height of a B-tree may be less than that of a red black tree since the branching factor of B-tree can be much larger.

B-tree of order m is an ordered tree where each node has at most m children. If k is the actual number of children in the node then there must be k–1 number of keys in the node. In other words, if an internal node x contains n[x] keys then x has n[x] + 1 children. Order of B-tree defines (maximum and minimum) the number of children. Degree of B-tree defines (maximum and minimum) number of keys. Degree is defined as the minimum degree of B-tree.

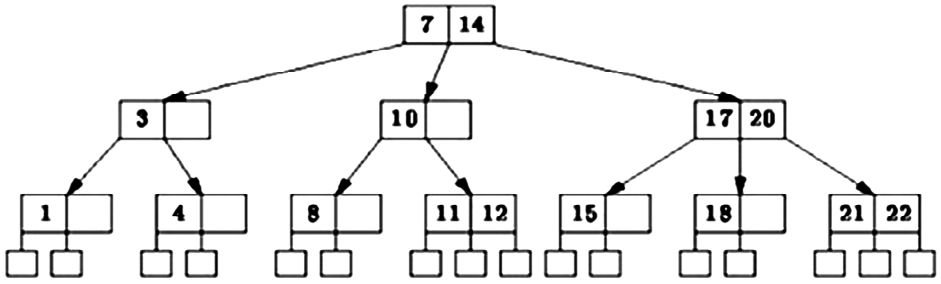


Figure 11.1: Example of a B-Tree

11.3.1 Properties of B-Tree

A **B-tree (T)** has the following properties:

1) If a node x has n[x] number of keys which are stored in increasing order, then it will have n[x] + 1 number of children.

2) All leaves have the same depth, which is the tree's height h.
3) The keys $key_i[x]$ separate the ranges of keys stored in each sub-tree: if k_i is any key stored in the subtree with root $c_i[x]$, then $K_1 = key_1[x] = K_2 = key_1[x]$ ..... $= key_{n[x]}[x] = K_{n[x]+1}$
4) The degree of a B-tree is the upper and lower bounds on the number of keys of a node.
   a) Minimum degree 't' of a B-tree is equal or greater than 2.
   b) Except root other nodes must have at least t–1 keys.
   c) The root must contain at least one key.
   d) Every node can have at most 2t–1 keys.



CHECK YOUR PROGRESS

Q.1: Define B-tree.

Q.2: The height of all the leaves should have ..... depth.

Q.3: How are keys stored in a B-tree?

Q.4: How is B-tree different from red black tree?

11.4 B-TREE INSERTION

In a B-tree, the new element must be added only at leaf node. The operation of insertion works as follows.

1) Check whether tree is Empty.
2) If tree is Empty, then create a new node with new key value and insert into the tree as a root node.
3) If tree is Not Empty, then find a leaf node to which the new key value can be added using Binary Search Tree logic.
   a) If there is an empty position available in the leaf node, then new key value is added to that leaf node without violating the ascending order of key value within the node.
   b) If that leaf node is already full, then split that leaf node by sending middle value to its parent node. Repeat the same until the sending value is fixed into a node.

c) If the splitting is occurring to the root node, then the middle value becomes new root node for the tree and the height of the tree is increased by one.

Example 11.1: Let us insert keys into a B-tree where the minimum degree (t) for the following B-tree is 3. So a node can contain at most 5 keys.

a) Initial B-Tree: Let us consider the following B-tree for showing the Insertion operations of keys into a B-tree.

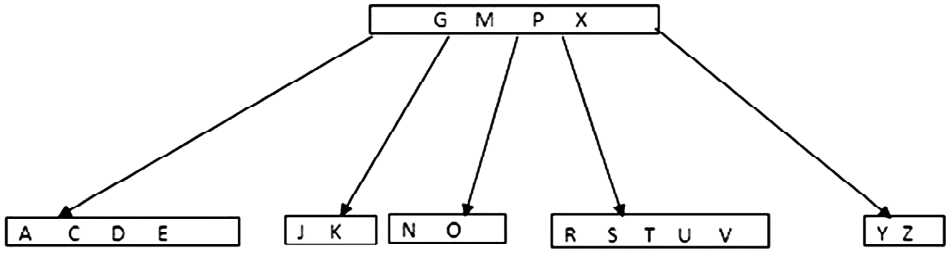


Figure 11.2: Initial B-Tree

b) B is inserted: This is a very simple insertion operation into a leaf node. The new key B is inserted into the leaf node of the B-tree without violating the ascending order of key value within the node.

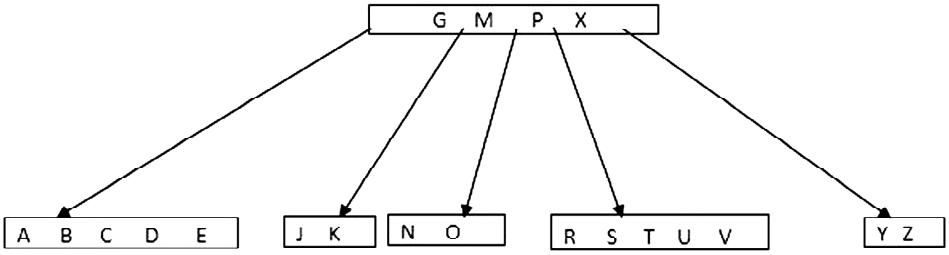


Figure 11.3: The result of inserting B into the Initial Tree

c) Q is inserted: In this step the node R S T U V is split into two nodes containing R S and U V. The key T is moved up to the root and after that Q is inserted in the leftmost of the two halves i.e. R S node.

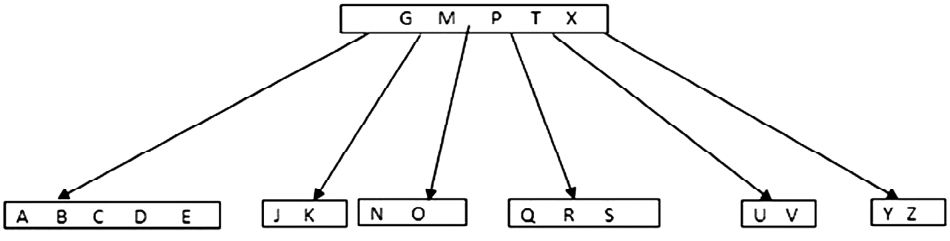


Figure 11.4: The result of inserting Q into the Previous Tree

d) L is inserted: In this case the root needs to be split because it is full and as a result the height of the tree increases by 1. L is inserted into the leaf which contains J K.

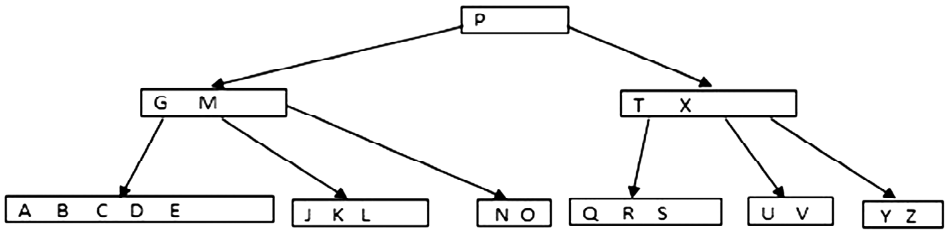


Figure 11.5: The result after inserting L into the Previous Tree

e) F is inserted: The node with A B C D E is split before F is inserted into the rightmost of the two haves i.e. the node with D E. The node needs to be split because more than 5 keys can't be accommodated in a node.

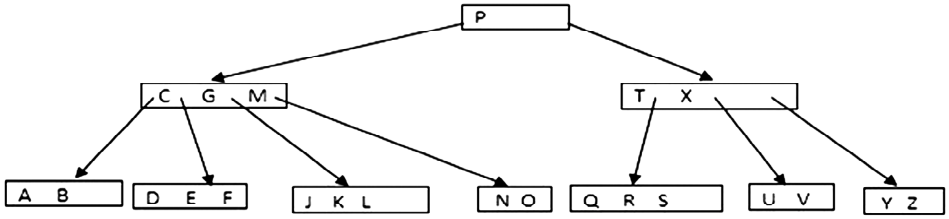


Figure 11.6: The result after inserting F into the Previous Tree

11.5 B-TREE DELETION

Deletion process of B-tree involves deleting a key from any node and deletion from an internal node requires the rearrangement of the node's children. It is assumed that the key k is to be deleted from the sub-tree whose root is x. If the root node x ever becomes an internal node having no keys then we delete x, and x's only child c1 becomes the new root of the tree, as a result it will decrease the height of the tree by 1 and it will hold the property that the root of the tree contains at least 1 key (unless the tree is empty).

Various cases of deleting keys from a B-tree are discussed below:

1) Delete the key k from x, If the key k is in node x and x is a leaf.
2) Following steps need to follow when the key k is in node x and x is an internal node:

a) If the child y that precedes k in the node x contains at least t keys, then it will find the predecessor of k in the sub-tree rooted at y. After that the value of k will be replaced by its predecessor and recursively the value of the predecessor will be deleted from its own place.

b) Similarly, If the child z that follows k in the node x has at least t keys then it will find the successor of k in the sub-tree rooted at z. After that the value of k will be replaced by its successor and recursively the value of the successor will be deleted from its own.

c) Otherwise, if both the child y and z have only t–1 keys, then all of z and k will be merged together into y, thereby x will lose both k and the pointer to z, and y now has 2t–1 keys. Then z is made free and k will be deleted from y.

3) If the key k is not present in internal node x but present in the tree, then it is required to determine the root ci[x] of the appropriate sub-tree that must contain k. If ci[x] has only t–1 keys, then the step 3a or 3b has to be executed as necessary to ensure that we go down to a node which contains at least t keys. Then, it is finished by doing a recursion on the appropriate child of x.

a) If the root ci[x] has only t–1 keys and has an immediate sibling with at least t keys, then an extra key is given to ci[x] by moving a key from x down into ci[x], moving a key from ci[x]'s immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into ci[x].

b) If ci[x] and both of ci[x]'s immediate siblings have t–1 keys, then we need to merge root ci[x] with one sibling, which will involve moving a key from x down into the newly merged node which will become the median key for that node.

Example 11.2: Consider that the minimum degree of the B-tree is 3. Therefore, a node can't have less than 2 keys except the root.

a) Initial B-tree having P as the root:

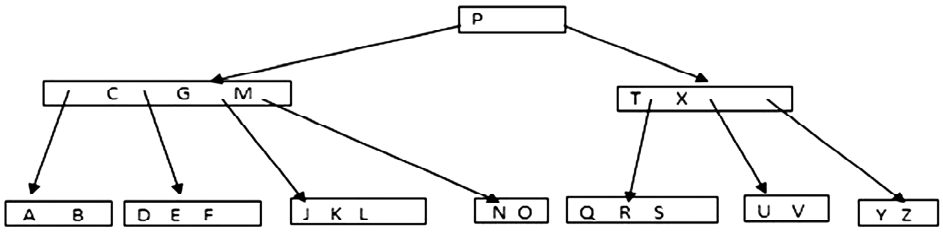


Figure 11.7: Initial B-Tree

b) Case 1: F Deleted: In this case F can be deleted directly from the leaf node because the key F is present in the leaf node. From the leaf node any key can be deleted directly.

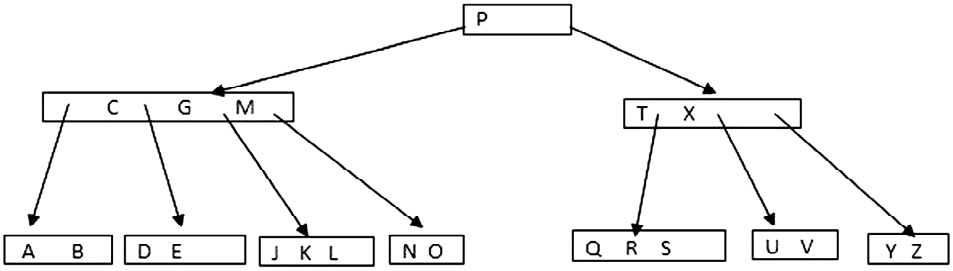


Figure 11.8: Case 1

c) Case 2.a: M deleted:

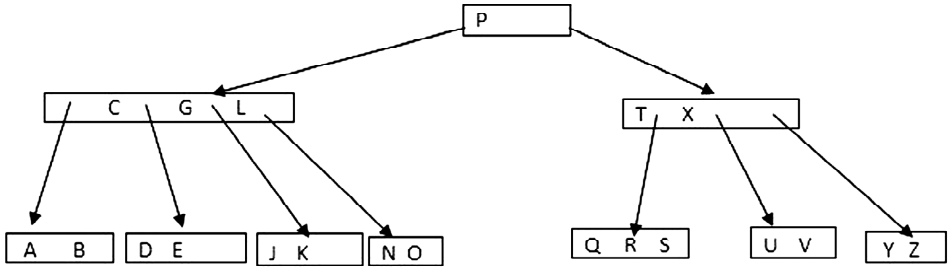


Figure 11.9: Case 2.a

In this case, the first step is to find out the predecessor of M and the predecessor value will be replaced with M. Therefore the predecessor of M is traced out which is L and the value L is moved up to replaced with M.

d) Case 2.c: G deleted: In this case G is pushed down to make the node D E G J K and G is deleted from the node.

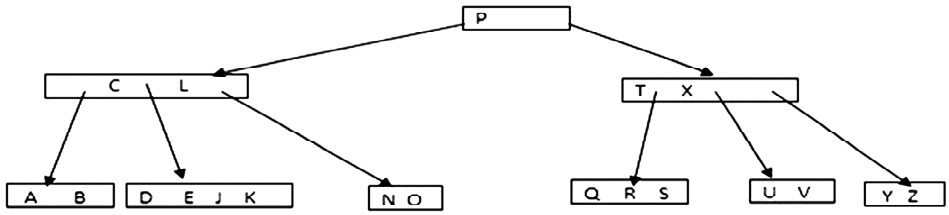


Figure 11.10: Case 2.c

e) Case 3.b: D is deleted: In this case the recursion can't descend to node C L because it has only 2 keys, So P is pushed down and merged with C L and T X to form C L P T X; after that key D is deleted from the leaf node.

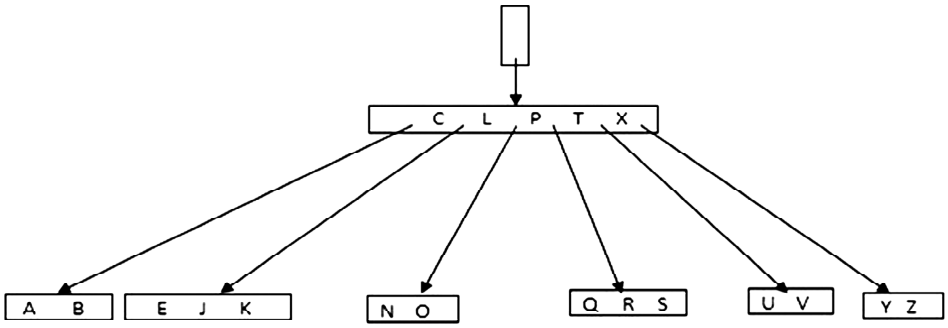


Figure 11.11: Case 3.b

f) Tree Shrinks in height: Here the root is deleted and the tree shrinks in height by 1.

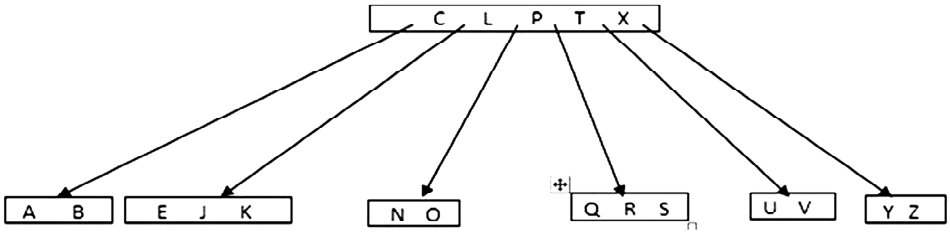


Figure 11.12: Case 3.b

g) Case 3.a: B is deleted: In this case C is moved downward to fill B's position and E is moved upward to fill C's position.

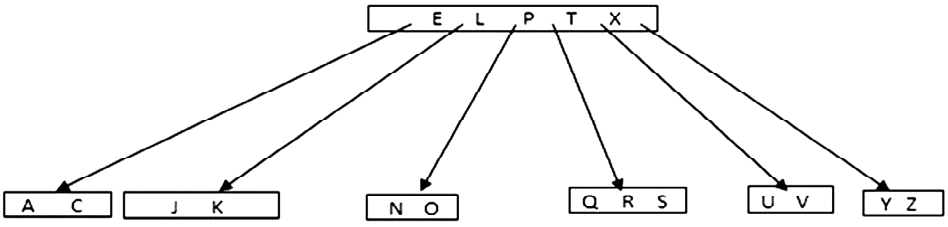


Figure 11.13: Case 3



CHECK YOUR PROGRESS

Q.4: How to insert a new key into the leaf node of a B-Tree?

Q.5: How to delete a leaf node from a B-Tree?

Q.6: If that leaf node is already full, then split that leaf node by sending ..... value to its parent node.

11.6 B+ TREE

B+ tree is widely used in the implementation of database indexes. B+ tree is a balanced tree in which each path from the root of the tree to a leaf of the tree is of the same length. B+Tree is a balanced tree which contains up to n–1 search key values $K_1, K_2, \ldots, \ldots, \ldots, K_{n-1}$, and n pointers $P_1, P_2, \ldots, \ldots, \ldots, P_n$, where n is fixed for a particular tree. The search key values within a node are in sorted order; if i < j, then $K_i < K_j$. Each leaf can contain up to n–1 values and can contain at least $\lceil (n-1)/2 \rceil$ values. If L_i and L_j are leaf nodes and i < j, then every search key value in L_i is less than every search key value in L_j. A non leaf node can contain up to n pointers and at least $\lceil n/2 \rceil$ pointers. The root node of a B+tree can have fewer than $\lceil n/2 \rceil$ pointers but it must have at least two pointers. Each node of the B+tree has an ordered list of keys and pointers which point to the lower level nodes in the tree.

11.6.1 Insertion in B+ Tree

The insertion algorithm is based on the following three cases:

1) When the leaf page and the index page are not full, place the record in the sorted position in the appropriate leaf page.
2) When leaf page is full and index page is not full then do the following:
   i) Split the leaf page
   ii) Place middle key in the index page in sorted order.
   iii) The records with keys below the middle key will be placed in Left Leaf page.

iv) The records with keys equal to or greater than the middle key will be placed in Right Leaf page.

3) When both leaf page and index page are full then do the following:

i) Split the leaf page

ii) Records with keys < middle key, go to the left leaf page.

iii) Records with keys = middle key, go to the right leaf page.

iv) Split the index page

v) Keys < middle key, move to the left index page

vi) Keys > middle key, move to the right index page

vii) The middle key goes to the next (higher level) index.

viii) If the next level index page is full, continue splitting the index page.

Example 11.3: Let us take the example of the following B+ tree.

Case 1: In this case leaf page is not full and index page is not null.

- We need to insert 28 in the tree.
- Leaf page is not full and index page is not null.
- The value 28 is placed in the sorted position in the appropriate leaf page.
- From the figure it is clear that the key 28 needs to be placed in the node containing the keys 25 and 30.
- 28 has been inserted into the node by maintaining the ascending order.



Figure 11.14: Insertion Operation when Leaf Page is not Full and Index Page is not Null

Case 2: Leaf page is full but index page is not full

- Now we have to insert 70 into the tree.
- Leaf page is full but index page is not full.
- The leaf page containing values 50, 55, 60, 65 is split
- The middle key 60 is placed in the index page in sorted order.
- The records with keys below the middle key i.e. 50 and 55 will be placed in Left Leaf page.
- The records with keys equal to or greater than the middle key i.e., 60, 65, 70 will be placed in Right Leaf page.

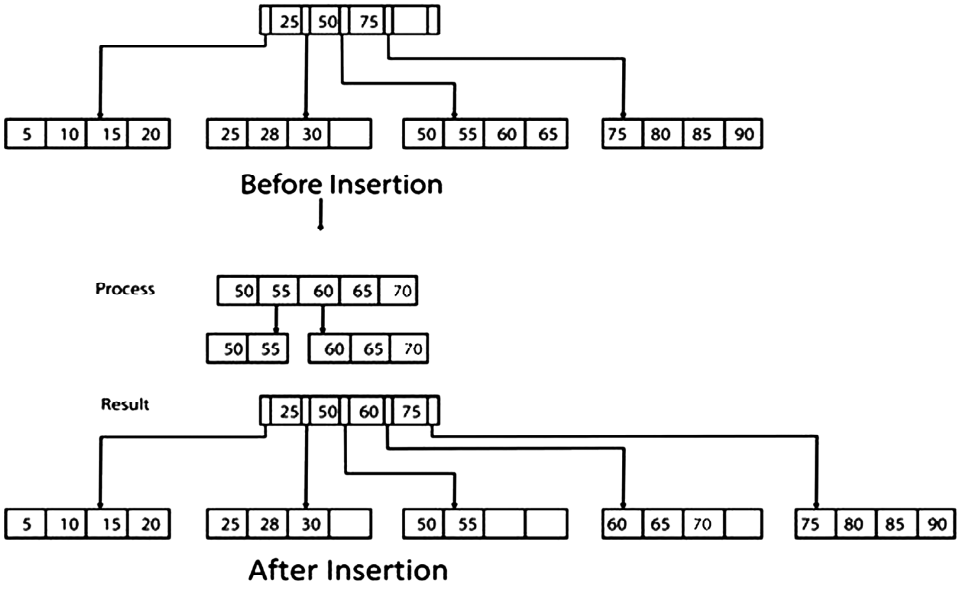


Figure 11.15: Insertion Operation when Leaf Page is Full but Index Page is not Full

Case 3: Leaf page is full and index page is also full

- The value 95 will be inserted in the Tree.
- Here both leaf page or data page and index page are full.
- Split the leaf page or data page having keys 75, 80, 85, 90, 95
- Records with keys < middle key, will go to the left leaf page i.e., 75 and 80 will be in the left leaf page.
- Records with keys = middle key, go to the right leaf page i.e., 85, 90, 95 will go to the right leaf page.
- Since index page is full, therefore split the index page.
- Keys (25, 50) < middle key (60), so moved to the left index page.
- Keys (75, 80) > middle key (60), so moved to the right index page.

- The middle key (60) goes to the next (higher level) index.
- If the next level index page is full, it will continue splitting the index page.

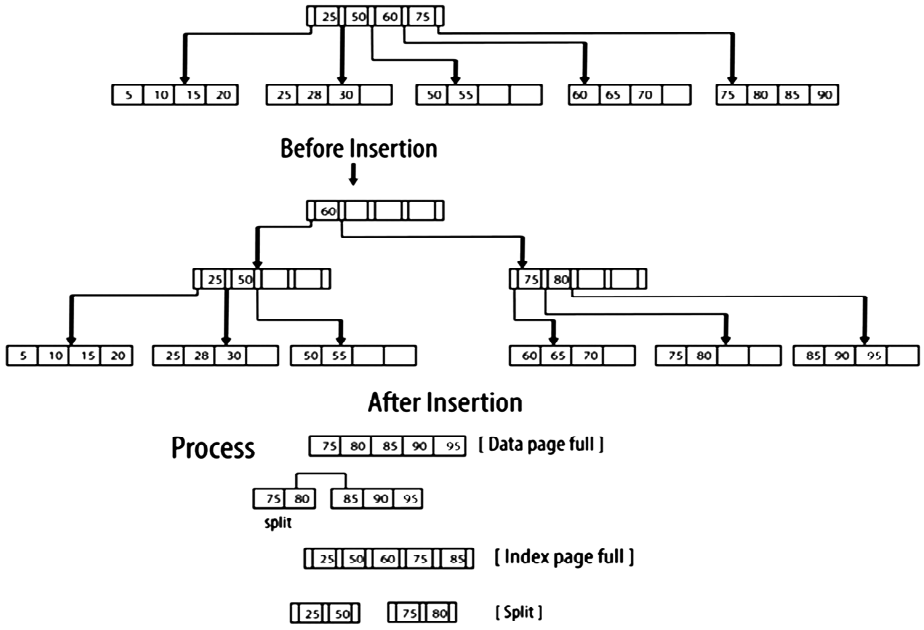


Figure 11.16: Insertion Operation when Leaf Page is Full and Index Page is also Full

11.7 EXTERNAL SORTING

External sorting is a technique to sort records which reside in external memory. There is a need of external sorting when main memory is not sufficient to accommodate for sorting of large collection of records. Here some records are read from disk; these are sorted and then again stored in the disk. This process is continued till the entire file is sorted. The main goal of external sorting is to minimize the number of times information must be read from or written to disk. One of the external sort techniques is reading an entire file into virtual memory of the computer and an internal sorting method such as quick-sort is applied to sort the file. In this process virtual memory manager controls disk accesses by using its normal buffer pool mechanism. But this method is not always feasible due to the limited size of virtual memory compared to that of disk space. When a large file can't be accommodated into virtual memory then an internal sorting method uses their own buffer pool.

Merge sort algorithm can be used to implement external sorting. In this approach a series of sequential passes are performed over the records, and in every pass sublists of different sizes are merged together. During this process, sublists of size 1 is merged into the sublists of size 2 in the first pass; in the second pass the sublists of size 2 is merged into sublists of size 4; and so on. A sorted sublist is termed as a run. After every pass gradually longer runs are produced by merging pairs of runs. The contents of the file are copied to another file after each pass. The algorithm works as follows:

1) The original file is split into two equal-sized run files.
2) From each run file one block is read into input buffers.
3) The first record is taken from each input buffer, and is written a run of length 2 to an output buffer in sorted order.
4) The next record is taken from each input buffer, and is written a run of length two to a second output buffer in sorted order.
5) It is repeated until it is finished, alternating output between the two output run buffers. After the input block's end is reached, the next block from the appropriate input file is read. After an output buffer becomes full, it is written into the appropriate output file.
6) Steps 2 through 5 are repeated. In the second pass, the first two records of each input run file remains in sorted order. A single run of four elements is produced as an output by merging two runs from second pass.
7) Each pass through the run files produces larger runs until only one run remains.



CHECK YOUR PROGRESS

Q.8: Why external sorting is required?

Q.9: Define B+ Tree.

Q.10: In a B+ Tree, when the leaf page is full and index page is not full then for inserting a new key, first step to be done is .....



11.8 LET US SUM UP

- B-tree is a balanced search tree which is designed to work on magnetic disks and other direct access secondary storage devices.
- B-tree of order m is an ordered tree where each node has at most m children.
- If an internal node x contains n[x] keys then x has n[x] + 1 children.
- In a B-tree, the new element must be added only at leaf node.
- Deletion process of B-tree involves deleting a key from any node and deletion from an internal requires the rearrangement of the node's children.
- B+ tree is a balanced tree in which each path from the root of the tree to a leaf of the tree is of the same length.
- B+ tree is a balanced tree which contains up to n–1 search key values K1, K2, …, ..., ..., Kn–1, and n pointers P1, P2, …, ..., ..., Pn. The search key values within a node are in sorted order; if i < j, then Ki < Kj.
- External sorting is a technique to sort records which reside in external memory.



11.9 FURTHER READING

1) Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. (2009). *Introduction to Algorithms*. MIT Press.
2) Silberschatz, A., Korth, H. F. & Sudarshan, S. (1997). *Database System Concepts* (Vol. 4). New York: McGraw-Hill.



11.10 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: B-tree is a balanced search tree which is designed to work on magnetic disks and other direct access secondary storage devices. B-tree of order m is an ordered tree where each node has

at most m children. If k is the actual number of children in the node then there must be k–1 number of keys in the node. Order of B-tree defines (maximum and minimum) number of children. Degree of B-tree defines (maximum and minimum) number of keys.

Ans. to Q. No. 2: The height of all the leaves should have same depth.

Ans. to Q. No. 3: If a node x has n[x] number of keys then these will be stored in increasing order.

Ans. to Q. No. 4: B-tree is different from red black tree in the manner that B-tree has many children, means branching factor of a B-tree is very large. The height of a B-tree may be less than that of a red black tree since the branching factor of B-tree can be much larger.

Ans. to Q. No. 5: If there is an empty position available in the leaf node, then new key value is added to that leaf node without violating the ascending order of key value within the node.

Ans. to Q. No. 6: Delete the key directly from the leaf node without making any other changes.

Ans. to Q. No. 7: If that leaf node is already full, then split that leaf node by sending middle value to its parent node.

Ans. to Q. No. 8: External sorting is a technique to sort records which reside in external memory. There is a need of external sorting when main memory is not sufficient to accommodate for sorting of large collection of records.

Ans. to Q. No. 9: A B+ tree is a balanced tree which contains up to n–1 search key values K1, K2, …, ..., ..., Kn-1, and n pointers P1, P2, …, ..., ..., Pn. The search key values within a node are in sorted order; if i < j, then Ki < Kj.

Ans. to Q. No. 10: In a B+ tree, when the leaf page is full and Index page is not full then for inserting a new key, first step to be done is Split the leaf page.



11.11 MODEL QUESTIONS

Q.1: Show the results of inserting the following keys in order into an empty B-tree with minimum degree 2.
F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E.

Q.2: Discuss the properties of a B-tree.

Q.3: Explain the condition of splitting a node in a B-tree during an insertion operation.

Q.4: Write down the steps involved during a delete operation in B-tree.

Q.5: Define B+ tree with an example.

Q.6: Why is external sorting required? Discuss.

Q.7: Discuss the major steps involved in external sorting.

Q.8: Explain the insertion operation in a B+ tree.

Q.9: Write short notes on:
a) External Sorting, b) B+ tree

Q.10: Write down the steps of insertion algorithm when both leaf page and Index page are full.

*** ***** ***

UNIT 12: ADVANCE SORTING AND SELECTION TECHNIQUES

UNIT STRUTURE

12.1 Learning Objectives

12.2 Introduction

12.3 Heap Sort

12.4 Shell Sort

12.5 Sorting in Linear Time

12.6 Counting Sort

12.7 Radix Sort

12.8 Medians and Other Statistics

12.9 Lower Bound on Sorting

12.10 Let Us Sum Up

12.11 Further Reading

12.12 Answers to Check Your Progress

12.13 Model Questions

12.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- describe heap sort, its application and implementation
- describe shell sort, its application and implementation
- explain counting sort, its application and implementation
- describe radix sort, its application and implementation
- define linear and comparison sort
- define medians
- describe lower bound on sorting algorithms.

12.2 INTRODUCTION

In previous units, we have learned about primitive sorting and searching algorithms like bubble sort, selection sort, quick sort and linear and binary searching algorithms. These are some popular and rapidly used

algorithms. Here, we shall discuss Heap Sort, Shell sort, Counting sort, Radix sort, sorting in linear time algorithms and its applications. These are known as some of the advanced sorting algorithm in data structure. Using these algorithms under special cases helps us to achieve a better space and run time analysis for our program. The efficiency of algorithms depends on its time and space complexities else using any sorting algorithm will sort your data. So for choosing the right algorithm for specific task you will require a good understanding of the available options. In some cases, the advanced algorithms would be beneficial for us to use. So, let us understand all the algorithms in this unit one by one.

12.3 HEAP SORT

Before starting heap sort we should know what a heap is. A heap is a complete binary tree i.e all of its levels except the last have completely filled nodes. The nodes are filled from left to right. Heaps are of two types- either it will be a min heap or a max heap. If the value of the parent node is always lesser than its child node than it is called a min heap while in max heap the value of parent nodes are always greater than its children. Therefore, in max heap the largest element will be the root node while in min heap the root will have the smallest element in the dataset. Heap sort is a sorting algorithm that uses the heap property. A heap sort can be used when you have to find the maximum or minimum element very quickly. A heap sort is mostly used in the implementation of a priority queue. Though it is similar to selection sort yet performancewise it is much better than the selection sort algorithm.

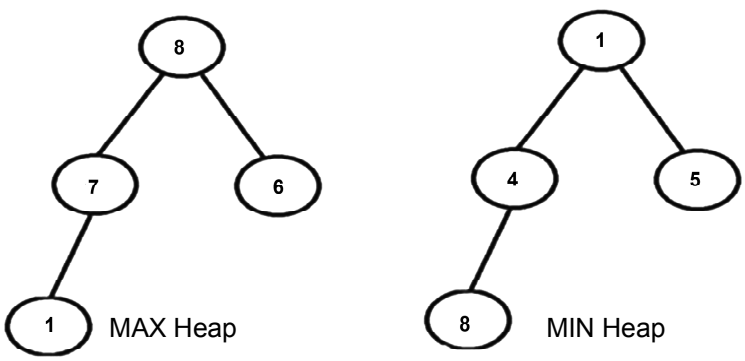


Figure 12.1: MAX Heap and MIN Heap

A heap is represented in an array structure. The root is represented by the array first element. The left child is denoted by 2*i and right child as 2*i+1 for node i. The best, average and worst case time complexity of heap sort is O(n*logn).

To perform heap sort on an unsorted array we follow the algorithm 12.1. First, we transform the unsorted array into a max heap or min heap. The root now contains the maximum or minimum element according to creation of a max heap or min heap step 1. Then, we replace the root element with the last element of the heap which reduces the size of heap by 1. Finally, we heapify the root of tree. We repeat these steps till heap size is greater than 1.

Algorithm 12.1: Heapify(array,pos,n)

Step 1: leftchild = 2pos

Step 2: rightchild = 2pos + 1

Step 3: if (leftchild < n) and (array[leftchild] > a[i]) then

max = leftchild

else

max = pos

Step 4: if (rightchild < n) and (array[rightchild] > a[max]) then

max = rightchild

Step 5: if (max!=pos) then

swap(array[pos],array[max])

Step 6: Heapify(array, max, n)

Let us now look at an example for the heapify algorithm given in algorithm 12.1.

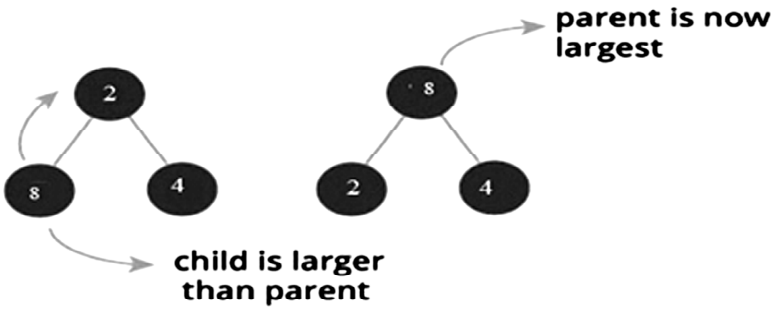


Figure 12.2: Heapify Algorithm

Example 12.1: Sort the array 2, 8, 4, 6, 1 using heap sort.

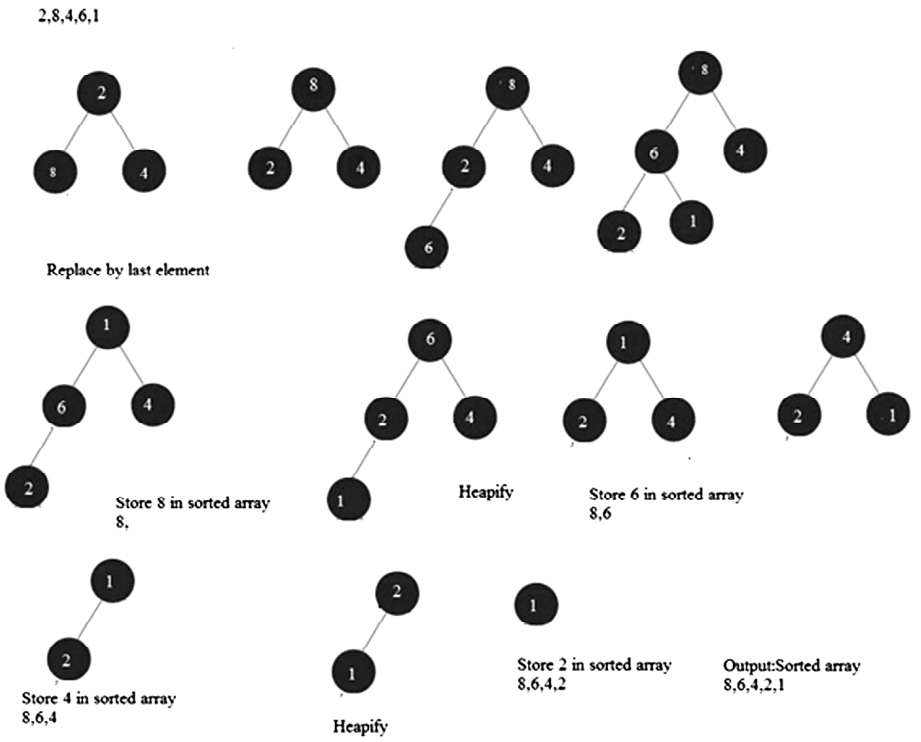


Figure 12.3: Example of Heap Sort



EXERCISE 12.1

Sort the above array using in ascending order.

12.4 SHELL SORT

The Shell sort was introduced to counter the demerit of insertion sort. Donald Shell introduced the Shell sort. In insertion sort if we need to move an element from far away it requires many steps to reach at the designated point since in insertion sort we move by unit element. The shell sort removes this disadvantage by breaking the unsorted array into smaller sub arrays. Each sub array is solved using insertion sort. The distance between each successive element in the sub array is called **gap**. At first it sorts the widely spread element and next it sorts the less spread elements.

There are many ways to calculate the gap given by:

Shell's original sequence: n/2, n/4 , …, …, …,1 (divide by 2 repeatedly);

Hibbard's increments: 1, 3, 7, …, …, …,2^n – 1;

Knuth's increments: 1, 4, 13, …, …, …,$(3^n - 1)/2$;

Sedgewick's increments: 1, 5, 19, 41, 109, …, ..., ...,

The best case complexity of shell sort is O(n*logn).The worst case time complexity is $O(n^r)$,$1 < r < 2$.

Algorithm 12.2: Shell_Sort

Step 1: Calculate and initalize the value of h.

Step 2: Divide the array into sub-arrays of h equal interval.

Step 3: Use insertion sort to solve the sub array.

Step 4: Repeat the steps until the interval is equal to 1.

Example 12.2: Shell Sort Example

Let interval = n/2 = 6

{60, 107, 153, 109, 33, 2, 21, 117, 122, 126, 27, 140}

The following are the six sub list of the input array,

{60, 21}, {107, 117}, {153, 122}, {109, 126}, {33, 27}, {2, 140}

We get the sorted list as,

{21, 107, 122, 109, 27, 2, 60, 117, 153, 126, 33, 140}.

Calculating new interval (k/2 = 6/2 = 3),

The following are the new 3 sub list,

{21, 109, 60, 126}, {107, 27, 117, 33}, {122, 2, 153, 140}.

We get the sorted list as,

{21, 27, 2, 60, 33, 122, 109, 107, 140, 126, 117, 153}

We continue to find the interval unless it decreased to1.

Next (k/2 = |3/2| = 1)

The required sub list on implementing insertion sort is,

{2, 21, 27, 33, 60, 107, 109, 117, 122, 126, 140, 153}.



CHECK YOUR PROGRESS

Q.1: The value of a parent node in max heap is ..... than its children.

Q.2: The value of a parent node in min heap is ..... than its children.

Q.3: Shell sort uses ..... sort.

12.5 SORTING IN LINEAR TIME

So far, we have seen insertion sort, selection sort, merge sort, quick sort, heap sort, shell sort, and bubble sort. All these algorithms are comparison based sorting algorithms. The only way we could use to achieve the ordering information was a pair wise comparison between the elements. The best time complexity of the sorting algorithms until now is O(n*logn). The question arises is can we obtain a better lower bound on this sorting algorithms? There exist sorting algorithms that have a faster time complexities that are better than O(n*logn) times but requires the input sequence to follow special assumptions. These sorting methods have linear running time. Counting sort, radix sort and bucket sort are examples of linear sorting algorithms. These sorting algorithms use different operations to sort other than comparison. Though these sorts are better than comparison sorts they come with their own limitations. Counting sort and radix has the limitation that they assume that the input elements consist of integer values in a smaller range, but, the bucket sort assumes the input to be distributed uniformly over an interval. Though these linear algorithms can solve sorting in linear time yet is not popularly used due to its limitations. First its efficiency degrades if the input is not randomly ordered. Secondly, these methods have the limitation of requiring an extra space proportional to the size of the original array. Therefore, if the input is big in size the extra space is a real liability.

12.6 COUNTING SORT

Counting sort is an efficient sorting algorithm for sorting of array elements that use integer keys to sort the values. Counting sort is used as subroutine for other sorting algorithms like the Radix sort. In integer array the key is simply the list of integer value while a list of alphabet array could have assigned keys by means of some mappings and then sorting the alphabet array by using these keys and finally getting the sorted array. Counting sort is not a comparison based sorting algorithm unlike the other sorts. It is a linear sort algorithm that uses other operations other than comparison of element pairs. It has a running time much faster than

comparison based sorting algorithms. The occurrence of the distinct elements of the input array to be sorted is counted in an auxiliary array and then its index is mapped in the sorted array according to the value of the array element. The run time complexity of counting sort is O(n) with space proportional to the range of data.

Example 12.3: Perform counting sort on the array {4, 2, 2, 1, 3, 0}.

Answer: Let the input data be:

4	2	2	1	3	0

The count array will be:

Array index	0	1	2	3	4
Array elements	1	1	2	1	1

The sorted array will be:

0	1	2	2	3	4

Algorithm 12.3: Counting_Sort(Array, B, k)

Step 1: for i ← 1 to k

Step 2: Count[i] ← 0

Step 3: for j ← 1 to n

Count[Array[j]] ← Count[Array[j]] + 1 // Count[i] = the number of appearances of i in Array.

Step 4: for i ← 2 to k

Count[i] ← Count[i] + Count[i-1] // Count[i] = the number of elements in Array that are = i

Step 5: for j ← n down to 1

B[Count[Array[j]]] ← Array[j]

Count[Array[j]] ← Count[Array[j]] – 1

return B



EXERCISE 12.2

Sort the array A = { 5, 6, 6, 7, 1, 1, 1, 8, 3, 6, 0, 0, 1, 4} using counting sort. Show the contents of the counter array.

12.7 RADIX SORT

Radix sort is a sorting algorithm used to sort 'd' digits where digits are base 'b' i.e., the digits have a range of [0, b–1]. Radix sort is more convenient to be used while sorting large numbers than counting sort. The algorithm sorts the elements of array by each digits starting with the least significant digit (the rightmost last one) moving to the most significant digit (the leftmost digit).

Algorithm 12.4: Radix(Array[1...n], d)

Step 1: for i ← 1 to d

Step 2: Use stable sort(eg: counting sort) to sort Array on digit i

Assuming the time taken to sort digitwise for inner loop is O(n+b) (counting sort) the running time of radix sort is O(d(n+b)) = O(dn+db).

Example 12.4: Let us take the following array with 6 numbers with 3 digits. The array is {145, 123, 678, 987, 200, 345}

Answer: Looking at the array, we know that

n = 6, d = 3, b = 10 (decimal)

The array will look like this after first digit sorting:

20**0**, 12**3**, 14**5**, 34**5**, 98**7**, 67**8** (sorted by 1st digit)

The array will look like this after second digit sorting:

2**0**0, 1**2**3, 1**4**5, 3**4**5, 6**7**8, 9**8**7 (sorted by 2nd digit)

The array will look like this after third digit sorting:

123, **1**45, **2**00, **3**45, **6**78, **9**87 (sorted by 3rd digit)



CHECK YOUR PROGRESS

Q.4: Counting sort and radix sort are ..... sorting algorithms.

Q.5: Radix sort is used to sort ..... values.

12.8 MEDIANS AND OTHER STATISTICS

The i^{th} order static of an input array A of n elements where 1<i<n is the i^{th} smallest element of A. For minimum it is the 1^{st} element and for maximum it is the n^{th} element. The median of the array elements is calculated

as (n+1)/2 .If the array has odd number of elements we have one median and for even elements we have two medians. Now the problem is given an array how to find the i^{th} element. Use of a sorting algorithm for finding a single element is not a good choice and can have a faster linear time algorithm. We shall learn selecting the i^{th} element from a group of n elements now.

Finding the Minimum and Maximum Elements: How many comparisons are needed to find the minimum and maximum elements? There will be n–1 comparisons and for both minimum and maximum element and in total we would have 2n–2 comparison. Can we do improve on this comparison? The answer is yes, we can attempt to have a better algorithm of (3n–3)/2 steps.

The idea is to have two variables corresponding to min and max and process the n numbers in pairs. Take the first pair first element as minimum and second element as maximum. Next for every pair compare the smaller value with minimum and the larger with maximum to obtain the minimum and maximum value of n elements.

Algorithm 12.5: MAX_MIN(Array, n)

```
Step 1:  maximum ← Array[n]
Step 2:  minimum ← Array[n]
Step 3:  for i = 1 to n/2 do
                 if Array[2i – 1] ≥ Array[2i]
         then
                 if Array[2i – 1] > maximum
         then
                 maximum ← Array[2i – 1]
Step 4:  if Array[2i] < minimum
         then
                 minimum ← Array[2i]
         else if Array[2i] > maximum then
                 maximum ← Array[2i]
Step 5:  if Array[2i – 1] < minimum then
                 minimum ← Array[2i – 1]
Step 6:  return maximum and minimum
```



EXERCISE 12.3

Write a C program to implement the MAXMIN algorithm for finding the order statics.

Adversary Arguments: To prove that the above algorithm takes (3n–3)/2 time and any algorithm that takes less than (3n–3)/2 amount of time fails for the output can be shown using the concept of adversary arguments. An adversary is referred to as an opponent that plays against the key algorithm. The goal of an adversary algorithm is to maximize the number of steps of the key algorithm needed to solve a problem.

Selection Algorithm: Selection is of little importance if we have sorted values but if we use a sorting algorithm to find just one value it would be like firing bullet for killing a mosquito. Using a sorting algorithm of O(nlogn) gives us a selection algorithm of running time O(nlogn). Can we solve our problem faster?? We can have O(n) selection algorithm by using randomized selection method.

Randomized Selection Method: For finding the k^{th} order static from n elements we use randomized partition and split the given array into two sub arrays say s–1 and n–s be the left sub array and right sub array respectively. If k=s the pivot is the element we are looking for. If k<=s then we search the left search array, else we search the right sub array.

Algorithm 12.6: Random_Select(Array[a..l], i)

```
Step 1:  if a = l      return Array[a]
Step 2:  q<–Randomized–Partition (Array,a,l)
Step 3:  k<–q–a+1
Step 4:  if i=k then
              return Array[q]
              else if i < k then
              return RandomSelect (Array, a, q–1, i)
              else
              return RandomSelect (A, q+1, l, i–k)
```

Worst case running time: $O(n^2)$

Best case running time: O(n)



EXERCISE 12.4

Find the 8th order statics from the array A = {4, 6, 2, 3, 8, 9, 11, 25, 34, 11} using random select algorithm.

The algorithm for worst case linear time selection method is given below.

Algorithm 12.7: LSelect(Array, n, i)

//finding i^{th} order statics

Step 1: Partition the input array into sub group arrays of size 5.

/* Partition on median-of-medians

Step 2: Find the median of each subgroup

// Let M be the array of each group medians

Step 3: pivot = LSelect (M, ,)

Step 4: [L, R] = partition (A, pivot); // L = left array, R = right array

Step 5: k = |L| + 1

Step 6: If i = k, return pivot

Step 7: If (i > k) return LSelect (R, n–k, i–k)

Step 8: If (i < k) return LSelect (L, k–1, i)

12.9 LOWER BOUND ON SORTING

Till now the running time of sorting algorithms that we came across are $\Theta(n \log n)$ and $\Theta(n^2)$ for heap sort, merge sort, quick sort, bubble sort, selection sort and insertion sort. All of these methods have a worst time complexity of $\Omega(n \lg n)$. The question is can we find an algorithm faster than this? If not we call this as the lower bound for this algorithms. A lower bound can be termed as the best running time in worst case for solving a particular problem. To prove that the lower bound for sorting is $\Omega(n \lg n)$ we have to prove that all algorithms for sorting however fast and smart, do not perform better than nlogn in worst case scenario. But how do we prove without even knowing all sorting algorithms? This can be achieved by understanding the model of the algorithms used to perform the specific task. All the sorting algorithms are based on comparison based model. We

will prove that all comparison based models have to perform Ω(n lg n) comparisons. Therefore, we create a decision tree model as decision tree is the only model that uses comparison based method to gain knowledge on inputs. In such cases, the worst case on the tree would be the maximum height of the tree.

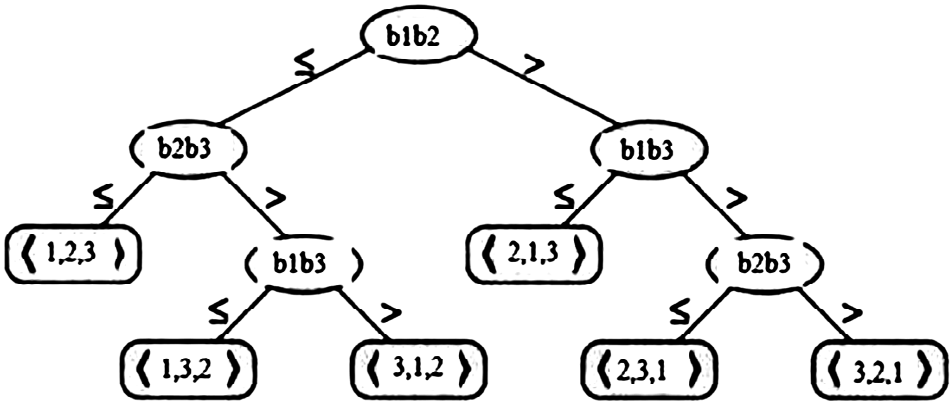


Figure 12.4: Binary Tree for Comparison Sort

Assuming that the elements consist of distinct numbers from 1 to n, the number of leaves must be n! (for n elements must appear as leaf of the decision tree). A tree of height l has 2*l leaves.

$2^*l \geq n! \Rightarrow l \geq \log(n!)$

$= \log(n^*(n-1)^*(n-2)\cdot\cdot\cdot(2))$

$= \log n + \log(n-1) + \log(n-2) + \cdot\cdot\cdot + \log 2$

$= \sum_{i=2}^{n} \log i$

$= \sum_{i=2}^{n/2-1} \log i + \sum_{i=n/2}^{n} \log i$

$= 0 + \sum_{i=n/2}^{n} \log n/2$

$= n/2^*Log^*n/2$

$= \Omega(n \log n)$



12.10 LET US SUM UP

- Heap sort is used when we need to find the maximum and minimum element from an array.
- Heap sort is used to implement a priority queue.

- Insertion sort, selection sort, bubble sort, merge sort, quick sort are examples of comparison based sorting methods.
- Radix sort and counting sort are examples of linear sorts.
- Linear sort is better than comparison sort but has its own limitations.
- The lower bound of comparison sort is $\Omega(n \log n)$.
- Radix sort and counting sort sorts integer values.
- Shell sort is used to counter the demerit of insertion sort.
- Shell sort uses insertion sort.
- The gap in insertion sort can be calculated using Shell's original sequence, Hibbard's increments, Kruth's increments, Sedgewick's increments.
- The parent of a max heap is always greater than its children.
- The parent of a min heap is always less than its children.



12.11 FURTHER READING

1) Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. (2009). *Introduction to Algorithms*. MIT Press.

2) Silberschatz, A., Korth, H. F. & Sudarshan, S. (1997). *Database System Concepts* (Vol. 4). New York: McGraw-Hill.



12.12 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: Greater

Ans. to Q. No. 2: Lesser

Ans. to Q. No. 3: Insertion

Ans. to Q. No. 4: Linear

Ans. to Q. No. 5: Integer



12.13 MODEL QUESTIONS

Q.1: Write two applications of heap sort.

Q.2: When do you use a heap sort?

Q.3: What are max heap and min heap? Explain with a suitable example.

Q.4: Sort A = {44, 87, 99, 10, 30, 45, 77, 99, 135, 56} using heap sort. Also find the maximum element from the array.

Q.5: Why is Radix sort better than Counting sort?

Q.6: Why Shell sort is better that Insertion sort?

Q.7: Sort the array B = {44, 78, 56, 33, 90, 88, 1, 3, 4, 99, 63, 27} using shell sort.

Q.8: What is the difference between a comparison sort and a linear sort? Give examples of both sorting method.

Q.9: Prove that the lower bound of comparison sort is ?(n log n).

Q.10: Use radix sort to sort the array
C = {4567, 1234, 8967, 4356, 9987, 3475}.

Q.11: What are adversary arguments?

Q.12: Complete the table below:

Algorithm	**When it is used**	**Time complexity**		
		Best case	**Average case**	**Worst case**
Selection sort				
Merge sort				
Insertion sort				
Quick sort				
Bubble sort				
Heap sort				
Shell sort				
Radix sort				
Counting sort				

*** ***** ***

UNIT 13: PRIORITY QUEUE

UNIT STRUCTURE

13.1 Learning Objectives

13.2 Introduction

13.3 Priority Queue

13.4 Priority Queue Using Array

13.5 Multi-queue Implementation

13.6 Linked List Representation of Priority Queue

13.7 Binary Heap

13.8 Binomial Heap

13.9 Fibonacci Heap

13.10 Let Us Sum Up

13.11 Further Reading

13.12 Answers to Check Your Progress

13.13 Model Questions

13.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- define priority queue
- describe different representations of prior ty queue
- explain applications of priority queue
- describe binary heap
- describe binomial heap
- explain Fibonacci heap.

13.2 INTRODUCTION

In the few units, we have learnt about different types of trees. In the previous unit, we have explored different sorting techniques like counting sort, radix sort etc. In this unit, we will learn about priority queue's and its applications. We will also explore the different ways priority queue can be implemented. We will also learn about binary heap, binomial heap and

Fibonacci heaps. In the next unit, we will explore the concept of partitions and disjoint sets.

13.3 PRIORITY QUEUE

A priority queue is another variation of queue structure. Here, each element has assigned a value, called priority of the element, and an element can be inserted or deleted not only at the ends but at any position of the queue. Figure 13.1 shows a priority queue.

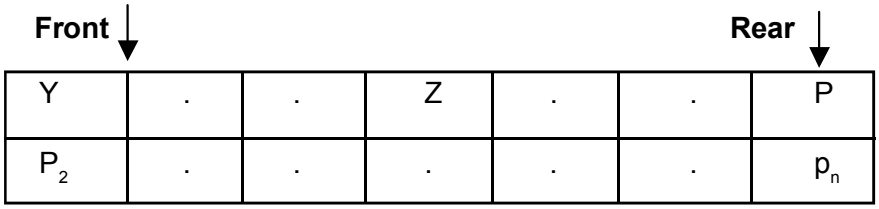


Figure 13.1: View of Priority Queue

With this structure, an element X of priority p_i may be deleted before an element which is at FRONT. Similarly, insertion of an element is based on its priority, that is, instead of adding it after the REAR we may insert it at an intermediate position dictated by its priority value.

Note that, the name priority queue is a misnomer in the sense that the data structure is not a queue as per the definition; priority queue does not strictly follow first-in first out(FIFO) principle which is the basic principle of a queue. Nevertheless, the name now firmly associated with this particular data type. However, there are various models of priority queue known in different applications. Let us consider a particular model of priority queue.

a) An element of higher priority processed before any element of lower priority.
b) Two elements with the same priority are processed according to the order in which they were added to the queue.

Here, 'process' means two basic operations, namely insertion or deletion. There are various ways of implementing the structure of priority queue. They are–

a) Using a simple/circular array
b) Multi-queue Implementation

c) Using a double linked list, and

d) Using heap tree

13.4 PRIORITY QUEUE USING ARRAY

With this representation, an array can be maintained to hold the item and its priority value. The element will be inserted at the REAR end as usual. The deletion operation will then be performed in either of the two following ways:

a) Starting from the FRONT pointer, traverse the array for an element of the highest priority. Delete this element from the queue. If this is not the front most element shift all its trailing elements after the deleted element one stroke each to fill up the vacant position.

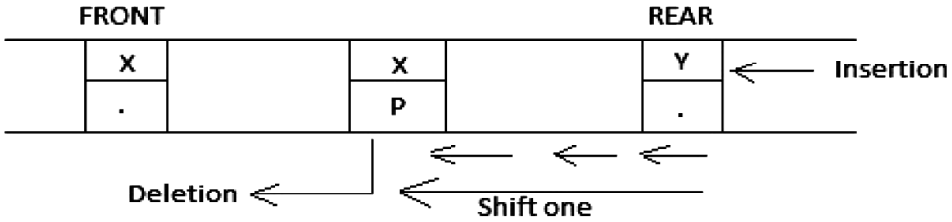


Figure 13.2: Deletion Operations in an Array Representation of a Priority Queue

This implementation, however, is very inefficient as it involves searching the queue for the highest priority element and shifting the trailing elements after the deletion. A better implementation is as follows:

b) Add the elements of the REAR end as earlier. Using a stable sorting algorithm, sort the elements of the queue so that the highest priority elements are at the FRONT end. When a deletion is required, delete it from the FRONT end only.

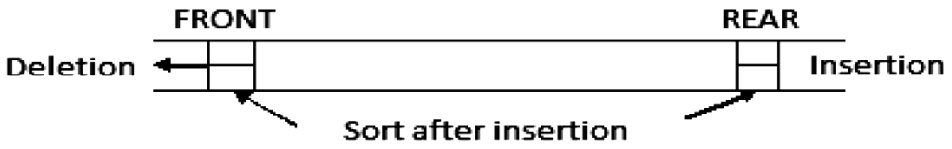


Figure 13.3: Another Array Implementation of a Priority Queue

The second implementation is comparatively better than the first one, here the only burden is to sort the elements. The algorithms of the two implementations are left as assignments to the reader.

13.5 MULTI-QUEUE IMPLEMENTATION

This implementation assumes N different priority values. For each priority Pi, there are two pointers Fi and Ri corresponding to FRONT and REAR pointer respectively. The elements between Fi and Ri are all equal priority value Pi. Figure 13.4 represents a view of such structure.

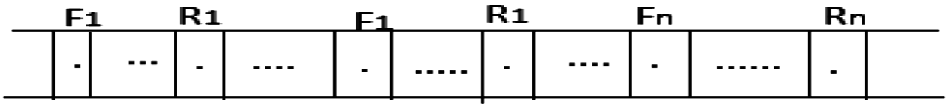


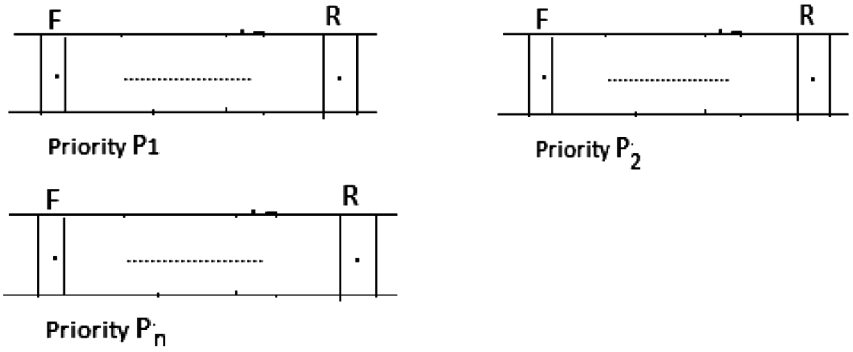
Figure 13.4: Multi-queue Representation of a Priority Queue

With this representation, an element with priority value Pi will consult Fi for the deletion and Ri for the insertion. But this implementation is associated with a number of difficulties.

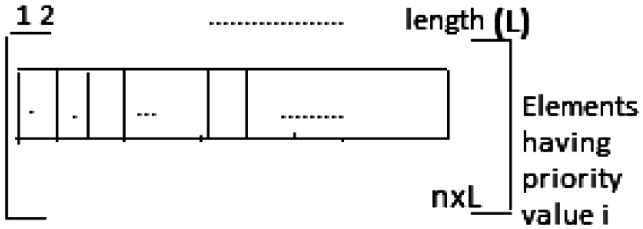
a) It may lead to a huge shifting in order to make a room for an item to be inserted.

b) Large number of pointers are involved when the range of priority values are large.

In addition to above, there are two other techniques to represent multi-queue which are shown in figure 13.5(a) and 13.5(b).



a) Multiples queues with simple queues



b) Matrix Representation

Figure 13.5: Multi-queue Implementation with Multiple Simple Queue and Matrix.

It is clear that in figure 13.5(a) that for each priority value a simple queue is to be maintained. An element will be added into a particular queue depending on its priority value.

The priority queue as shown in figure 13.5(b) is in some way better than the multi-queue with multiple queues. Here, one can get rid of maintaining several pointers for FRONT and REAR in several queues. Multi-queue with multiple queues has one advantage that one can have different queues of arbitrary length. In some application, it is seen that, the number of occurrence of elements with some priority value is much larger than other value, thus demanding a queue of larger size.

Both the above representations are not economic from the memory utilization point of view, majority of the memory space remains vacant.

13.6 LINK LIST REPRESENTATION OF PRIORITY QUEUE

This representation assumes the structure as shown in figure 13.6. LLINK and RLINK are two usual link fields, DATA is to store the actual content and priority is to store the priority value of the item. We will consider FRONT and REAR as two pointers pointing the first and last node in the queue, respectively. Here all the nodes are in stored order according to the priority values of the items in the nodes. Following is an instance of priority queue.

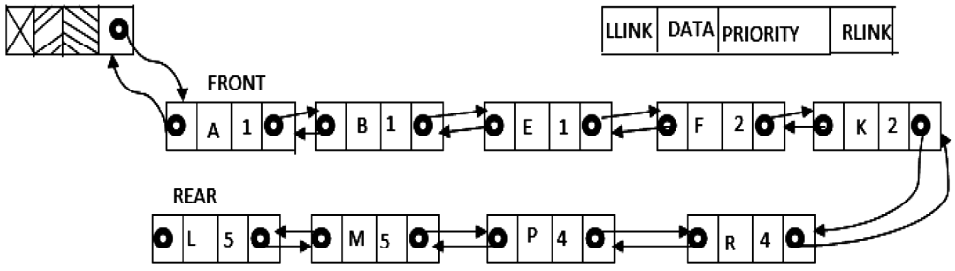


Figure 13.6: Link List Representation of a Priority Queue

With this structure, to delete an item having priority p, the list will be searched starting from the node under pointer REAR and the first occurring node with PRIORITY = p will be deleted. Similarly, to insert a node containing an item with priority p, the search will begin from node under pointer FRONT

and node will be inserted before a node found first with priority value p, or if not found then before the node with the next priority value.

The following two insertion and deletion operation of priority queue are:

a) Insert/Enqueue Operation: Whenever an element is inserted into queue, priority queue inserts the item according to its order. Here we're assuming that data with high value has low priority.

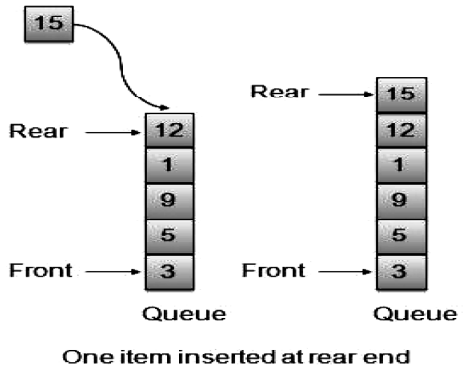


Figure 13.7: Insertion of Element in Queue

Function 13.1: Function to show Insertion and Remove operation of Priority Queue

```
void insertpqueue(int data)
{
    int i = 0;

    if(!isFull()){
        // if queue is empty, insert the data
        if(itmCount == 0)
        {
            intArr[itmCount++] = data;
        }
        else
            {
            //start from the right end of the queue
            for(i = itmCount - 1; i >= 0; i-- )
            {
```

```
                // if data is larger, shift existing item
                to right end
                    if(data > intArr[i]){
                        intArr[i+1] = intArr[i];
                    }
                    else
                    {
                        break;
                    }
                }
                    // insert the data
                    intArr[i+1] = data;
                    itmCount++;
                }
            }
        }
```

b) Remove/Dequeue Operation: Whenever an element is to be removed from queue, the queue gets the element using item count. Once an element is removed the item count is reduced by one.

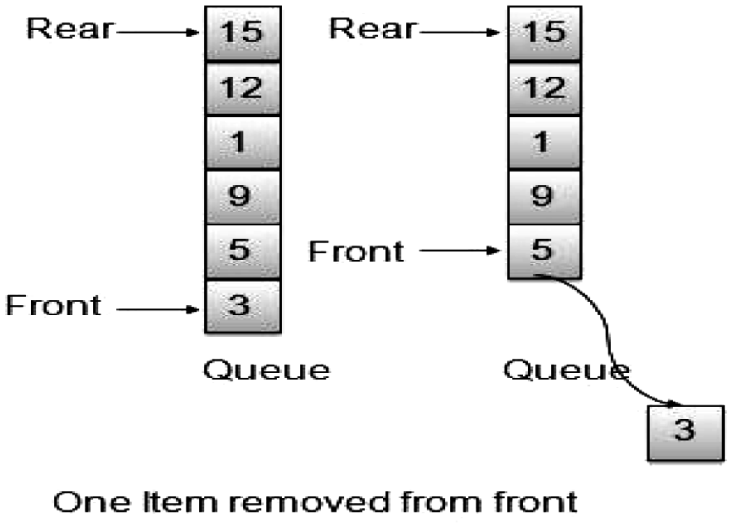


Figure 13.8: Removal of element from queue

Program 13.1: C program to Implementation priority queue

```
# include<stdio.h>
# include<malloc.h>

typedef struct node
{
    int priority;
    int info;
    struct node *link;
}NODE;
NODE *front = NULL;

/*Begin of insert*/
void insert(int item,int priority)
    {
        NODE *tmp,*q;

        tmp = (NODE *)malloc(sizeof(NODE));
        tmp->info = item;
        tmp->priority = priority;
        /*Queue is empty or item to be added has
        priority more than first item*/
        if(front == NULL || priority<front->priority)
        }
    {
        tmp->link = front;
        front = tmp;
    }
    else
    {
        q = front;
        while(q->link != NULL && q->link->priority
        <= priority )
            q=q->link;
```

```
        tmp->link = q->link;
        q->link = tmp;
        }
    }
    /*End of insert*/

    /*Begin of del*/
    void del()
    {
        NODE *tmp;
        if(front == NULL)
            printf(''Queue Underflow\n'');
        else
        {
            tmp = front;
            printf(''Deleted item is%d\n'',
            tmp->info);
            front = front->link;
            free(tmp);
        }
    }
    /*End of del*/

    /*Begin of display*/
    void display()
    {
        NODE *ptr;
        ptr = front;
        if(front == NULL)
            printf(''Queue is empty\n'');
        else
        {
            printf(''Queue is :\n'');
```

```
                printf(''Priority Item\n'');
                while(ptr != NULL)
                {
                    printf(''%5d%5d\n'',ptr
                    ->priority,ptr->info);
                    ptr = ptr->link;
                }
            }
        }
        /*End of display*/

        /*Begin of main*/
        int main()
    {
        int choice,item,priority;
        do
        {
            printf(''1.Insert\n'');
            printf(''2.Delete\n'');
            printf(''3.Display\n'');
            printf(''4.Quit\n'');
            printf(''Enter your choice :'');
            scanf(''%d'', &choice);
            switch(choice)
            {
            case 1:
                printf(''Input the item value to be
                added in the queue : '');
                scanf(''%d'',&item);
                printf(''Enter its priority : '');
                scanf(''%d'',&priority);
                insert(item,priority);
                break;
```

```
            case 2:
                del();
                break;
            case 3:
                display();
                break;
            case 4:
                break;
                    default:
                    printf(''Wrong choice\n'');
            }
        }while(choice!=4);

        return 0;
    }
    /*End of main*/
```

Applications of priority queue: A priority queue is typically implemented using heap data structure. In Dijkstra's Shortest Path Algorithm priority queue is used when the graph is stored in the form of adjacency list or matrix. Priority queue can be used to extract minimum efficiently when implementing Dijkstra's algorithm. It is also used in Prim's algorithm to store the keys of nodes and extract minimum key node at every step. Priority queues are often useful in scheduling. For example, in discrete event situation, we simulate real-world systems as a series of events that are scheduled to take place at some future time. For example, if we wanted to simulate automobile traffic on downtown streets, we might create a whole collection of events, some of which spawn other future events:

- Each traffic light in our simulation would have an event scheduled for its next change. When a light changes to red, we schedule its next change to green some number of seconds later. When a light changes to green, we schedule a change-to-yellow event some time a little later, and so on.

- Parking lots and garages might be simulated as objects that, at random time intervals, toss a new car out on to the street in front of the lot or garage. The average size of the car-generation interval would vary with the time of day (e.g., at 5:00PM, when everyone is leaving work, the interval between cars leaving each garage would be reduced).
- Each moving car on the street would have an event associated with the time it needs to reach the next intersection along its path. When it reaches that intersection, we schedule a new event for the intersection after that, and so on.



CHECK YOUR PROGRESS

Q.1: Define priority queue.

Q.2: What are the different ways in which priority queue can be implemented?

13.7 BINARY HEAP

Let us remember the definition of binary leap. A binary heap is a complete binary tree which satisfies the heap ordering property. The ordering can be one of two types:

- the min-heap property: the value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root.

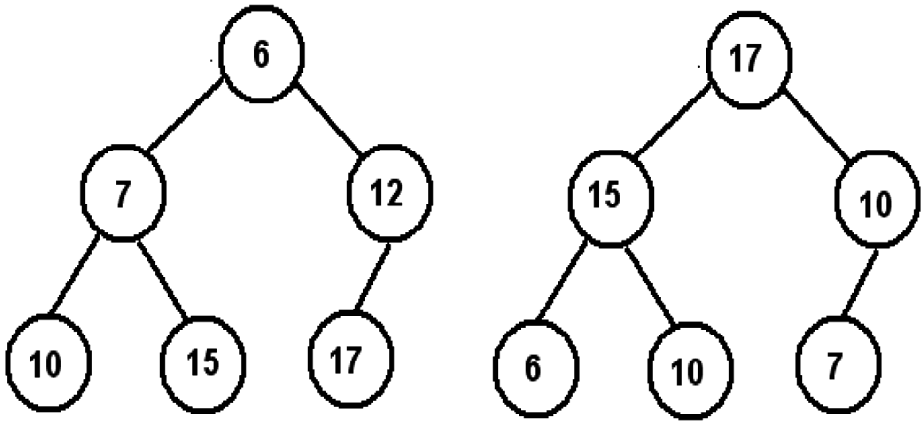


Figure 13.9: Min Heap and Max Heap

- the max-heap property: the value of each node is less than or equal to the value of its parent, with the maximum-value element at the root. As an example is given bellow figure 13.9.

13.8 BINOMIAL HEAPS

A binomial heap (BH) is a set of binomial trees that satisfy the binomial heap properties. First, let us discuss what a binomial tree is. A binomial tree is an ordered tree that can be recursively defined as follows:

a) A binomial tree is ordered 0 has a single node.

b) A binomial tree of order 1 has a root node whose children are the root nodes of binomial trees of order n–1, n–2, …, …, …, 2, 1 and 0.

c) A binomial tree BT has 2n nodes.

d) The height of a binomial tree BTn is n.

Look at the figure 13.10 which shows a few binomial trees of different orders. We can construct a binomial tree BTi from two binomial trees of order BHn–1 by linking them together in such a way that the root of one is the leftr

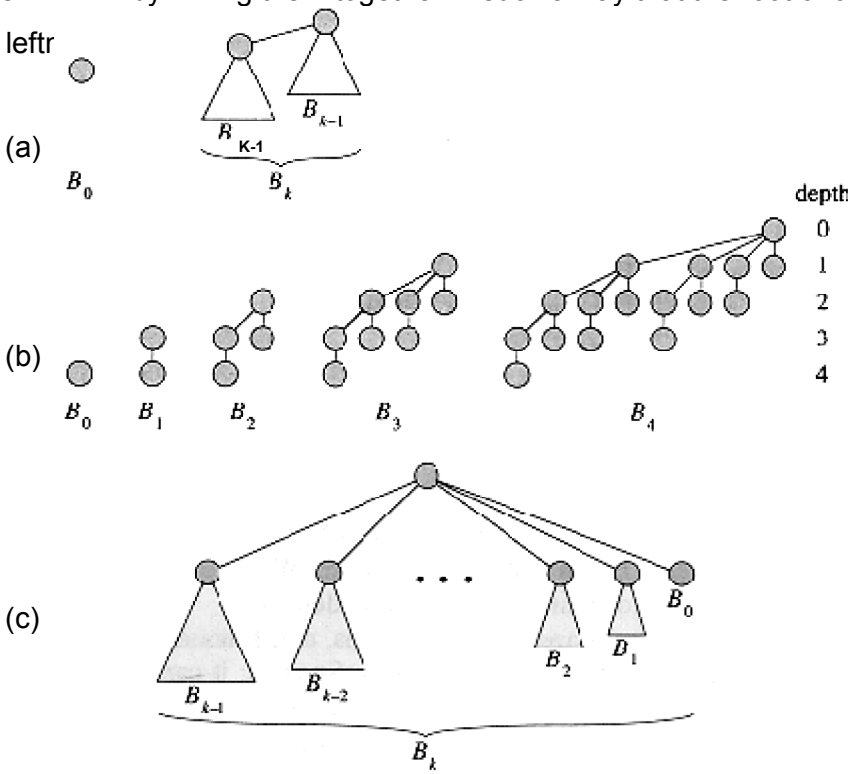


Figure 13.10: Binomial Trees

A binomial heap tree is a collection of binomial trees that satisfy the following properties.

a) Every binomial tree in binomial heap satisfies the minimum heap property.

b) There can be one or zero binomial trees for each order including zero order.

Link Representation of Binomial Heaps: Each node in a binomial heap BH has a value field that stores its value. In addition each node n has the following pointers:

a) P[X] that points to the parent of X

b) Child[X] that points to the leftmost child

c) Sibling [X] that points to the sibling of X which is immediately to its right.

If X is root node, then P[X]=NULL.

If X has no children, then Child[X]=NULL, and if X is the rightmost child of its parent, then Sibling[X]=NIL.

In addition to this, every node X has a degree field which stores the number of children of X. Figure 13.11 shows the binomial heap and Binomial heap and Link representation of Binomial Heap

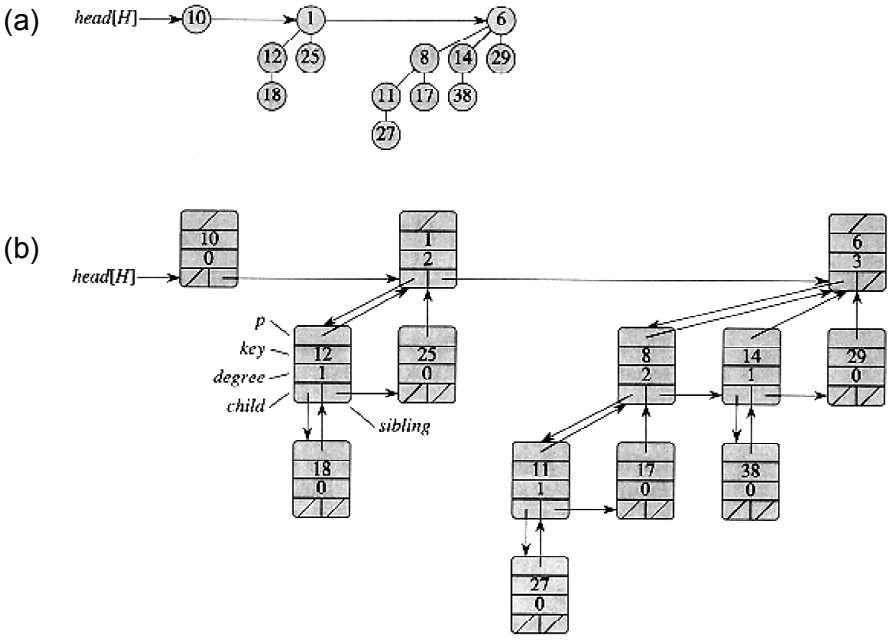


Figure 13.11: Binomial Heap and Link Representation of Binomial Heap

13.9 FIBONACCI HEAPS

A fibonacci heap is a collection of trees. It is loosely based on binomial heaps. If neither the decrease-value nor the delete operation is performed, each tree in the heap is like a binomial tree. Fibonacci heaps differ from binomial heaps as they have a more relaxed structure, allowing improved asymptomatic time bounds.

Like a binomial heap, a fibonacci heap is a collection of heap-ordered trees. The trees in a fibonacci heap are not constrained to be binomial trees, however. Figure 13.12 (a) shows an example of a Fibonacci heap. Unlike trees within binomial heaps, which are ordered, trees within fibonacci heaps are rooted but unordered. As figure 13.12(b) shows, each node x contains a pointer p[x] to its parent and a pointer child[x] to any one of its children. The children of x are linked together in a circular, doubly linked list , which we call the child list of x. Each child y in a child list has pointers left[y] and right[y] that point to y's left and right siblings, respectively. If node y is an only child, then left[y] = right[y] = y. The order in which siblings appear in a child list is arbitrary.

Circular, doubly linked lists have two advantages for use in Fibonacci heaps. First, we can remove a node from a circular, doubly linked list in O(1) time. Second, given two such lists, we can concatenate them (or "splice" them together) into one circular, doubly linked list in O(1) time. In the descriptions of Fibonacci heap operations, we shall refer to these operations informally, letting the reader fill in the details of their implementations.

Two other fields in each node will be of use. The number of children in the child list of node x is stored in degree[x]. The boolean-valued field mark[x] indicates whether node x has lost a child since the last time x was made the child of another node. Newly created nodes are unmarked, and a node x becomes unmarked whenever it is made the child of another node.

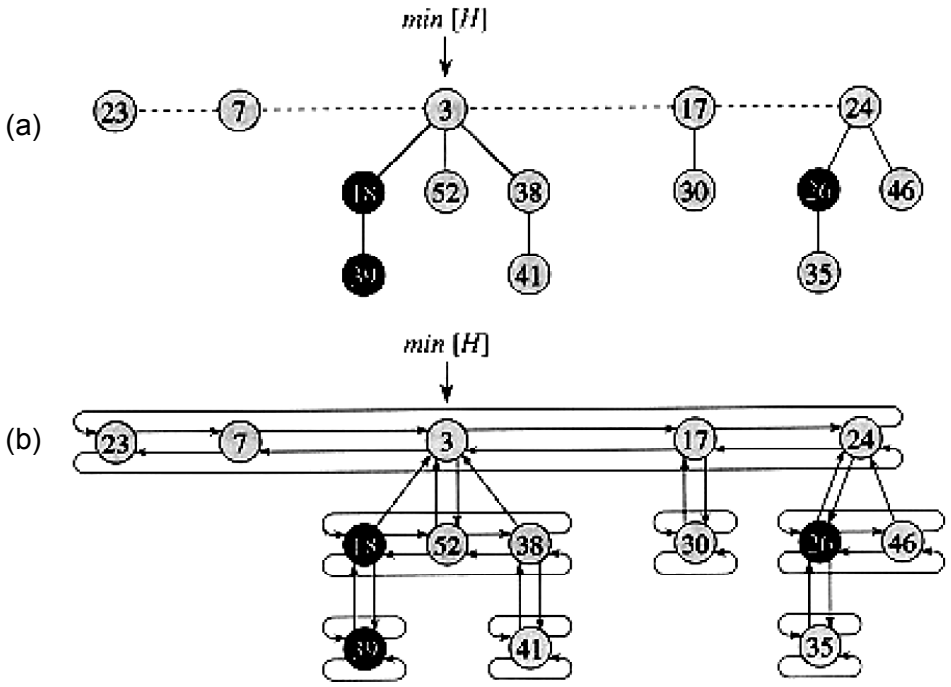


Figure 13.12 (a): A Fibonacci heap consisting of five heap-ordered trees and 14 nodes. The dashed line indicates the root list. The minimum node of the heap is the node containing the key 3. The three marked nodes are blackened. The potential of this particular Fibonacci heap is 5 + 2 3 = 11. (b) A more complete representation showing pointers p (up arrows), child (down arrows), and left and right (sideways arrows). These details are omitted in the remaining figures in this chapter, since all the information shown here can be determined from what appears in part (a).

A given Fibonacci heap H is accessed by a pointer min[H] to the root of the tree containing a minimum key; this node is called the minimum node of the Fibonacci heap. If a Fibonacci heap H his empty, then min[H] = NIL.

The roots of all the trees in a Fibonacci heap are linked together using their left and right pointers into a circular, doubly linked list called the root list of the Fibonacci heap. The pointer min[H] thus points to the node in the root list whose key is minimum. The order of the trees within a root list is arbitrary. We rely on one other attribute for a Fibonacci heap H: the number of nodes currently in H is kept in n[H].



CHECK YOUR PROGRESS

Q.3: What are the properties of binomial heap tree?

Q.4: How does Fibonacci tree differ from Binomial tree?



13.10 LET US SUM UP

- A binary heap is defined as a complete binary tree in which every node satisfies the heap property.
- There are two types of binary heaps: max heap and min heap.
- In a min heap, elements at every node will either be less than or equal to the elements at every node will either be greater than or equal to element at its left and right child.
- A binomial tree of order i has a root node whose children are the root nodes of binomial trees of order i–1, i–2, …, …, …, 2, 1, 0
- A binomial tree Bi of height i has 2i nodes.
- A binomial heap H is a collection of binomial trees that satisfy the following properties.
  - Every binomial tree in H satisfy the minimum heap property
  - There can be one or zero binomial trees for each order including zero order.
- A fibonacci heap is a collection of trees.
- Fibonacci heaps differ from binomial heaps, as they have a more relaxed structure, allowing improved asymptotic time bounds.



13.11 FURTHER READING

1) Samanta, D. (2004). Classic Data Structures. PHI Learning Pvt. Ltd.
2) Thareja, R. (2011). *Data Structures Using C*. Oxford University Press, Inc.
3) http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6.



13.12 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: A priority queue is another variation of queue structure. Here, each element has assigned a value, called priority of the element, and an element can be inserted or deleted not only at the ends but at any position of the queue.

Ans. to Q. No. 2: The different ways in which priority queue can be implemented are:

- Using a simple/circular array
- Multi-queue Implementation
- Using a double linked list, and
- Using heap tree

Ans. to Q. No. 3: A binomial heap tree is a collection of binomial trees that satisfy the following properties.

a) Every binomial tree in binomial heap satisfies the minimum heap property.

b) There can be one or zero binomial trees for each order including zero order.

Ans. to Q. No. 4: Fibonacci heaps differ from Binomial heaps as they have a more relaxed structure, allowing improved asymptomatic time bounds. Unlike trees within binomial heaps, which are ordered, trees within Fibonacci heaps are rooted but unordered.



13.13 MODEL QUESTIONS

Q.1: Answer the multiple choice questions:

I) The height of a binary heap with n nodes is equal to:

a) O(n) b) O(log n) c) O(n log n) d) O(n2)

II) The height of a binomial tree Bi is:

a) 2i b) 2i +1 c) i/2 d) i

III) In Fibonacci heap, how much time does it take to find the minimum node?

a) O(n) b) O(log n) c) O(n log n) d) O(1)

IV) An element at position i in an array has its left child stored at position:

a) 2i b) 2i+1 c) i/2 d) i/2 +1

V) How many nodes does a binomial tree of order 0 have?

a) 0 b) 1 c) 2 d) 3

Q.2: State True or False:

a) A binary heap is a complete binary tree.

b) An element at position 1 has its parent stored at position i/2.

c) Binomial heap are ordered

d) A binomial tree Bi has 2i nodes.

e) Fibonacci heaps are preferred over binomial heaps.

Q.3: Define binary heap.

Q.4: Compare binary tree and binary heaps.

Q.5: Explain the steps involved in inserting a new value in a binary heap with the help of suitable example.

Q.6: Discuss the applications of binary heap

Q.7: Differentiate among binary, binomial and Fibonacci heaps.

Q.8: Explain the operations performed on Binomial heap.

Q.9: Explain the operations performed on Fibonacci heap.

Q.10: Why are Fibonacci heaps preferred over binary and binomial heaps?

*** ***** ***

UNIT 14: PARTITION ADT

UNIT STRUCTURE

14.1 Learning Objectives

14.2 Introduction

14.3 Partition

14.4 Disjoint Set Operations

14.4.1 Linked List representation of Disjoint Sets

14.5 Union

14.5.1 Weighted Merge

14.6 Let Us Sum Up

14.7 Further Reading

14.8 Answers to Check Your Progress

14.9 Model Questions

14.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- define partition
- describe the different Disjoint Set Operations
- explain the representation of disjoint sets through linked list
- describe the basic idea of Union operation
- implement Union and Weighted Union operation.

14.2 INTRODUCTION

This unit gives an introduction of partition of sets. You will be able to understand the operations of disjoint sets. The unit has discussed the linked list representation of disjoint sets. The Union operation has a significant effectiveness which is discussed here besides the implementation of Union operation with the help of an example. The meaning of weighted union is explained and how it is to be implemented in data structure is also discussed. In the next unit, we will explore the concept of graphs in detail.

14.3 PARTITION

Partition is dividing a group of objects into disjoint sets. A partition is a collection of disjoint components and its union will be the total collection. The actual data type of partition is not the partition as a whole but rather the individual components of the partition. Each component of the partition is defined by an arbitrary element of the component referred to as representative element. To maintain a partition, a list of elements is maintained for each component. In this approach, each partition could have a reference to its list. This will require three references per element– (i) one for the next element in the component, (ii) one for the associated application data and (iii) one for the representative element. The union and find data structure provides a better solution in which each element requires two references and for which all methods run in constant time and efficient space utilization. In this approach the partition is represented as a forest of trees where each tree corresponds to a component of the partition. An advantage of this representation is that the tree structure is kept in a distributed fashion with each partition element only maintaining its parent. A partition is a collection of disjoint sets. Two sets are disjoint if their intersection is the empty set.

A **partition** of a set U is defined to be a collection P of subsets of U such that:

1) Every element of U is in one of the sets in P
2) No two sets in P have elements in common.

In other words, U is the union of all sets in P and P is pairwise disjoint. U is called the universe of P and P is a partition of U. Two sets are disjoint if their intersection is the empty set.

Partitions and equivalence relations are reflections of one another. A partition P determines an equivalence relation R on U by defining:

xRy iff x and y belong to the same set in P

Similarly, if R is an equivalence relation on U, then the set of equivalence classes of R is a partition:

$R[x] = \{y \in U : xRy\}$

$P = \{R[x] : x \in U\}$

The union-find algorithms take elements of a universe as arguments and operate on partitions of that universe.

14.4 DISJOINT SET OPERATIONS

A disjoint set data structure maintains a collection S = { S_1, S_2, …, …, …, S_n} of disjoint dynamic sets. Each set is identified by a representative, which is some member of the set. Each element of a set is represented by an object x to support the following operations.

MakeSet(x): It is an operation to create a new set whose only member is x.

Union(): It is an operation to merge two different sets into one set.

FindSet(): It is an operation to return the identity of a set which is usually an element in a set which acts as a representative of that set.

14.4.1 Linked List Representation of Disjoint Sets

A simple way to implement a disjoint set data structure is to represent each set by a linked list. The first object in each linked list serves as its set's representative. Each object in the linked list contains a set member, a pointer to the object containing the next set member, and a pointer back to the representative. Each maintains pointers head to the representative and tail to the last object in the list. Within each linked list, the objects may appear in any order.

MakeSet() and FindSet() will take O(1) time. To carry out MakeSet(x), a linked list is created whose only object is x. For FindSet(x), the pointer is returned, from x back to the representative.

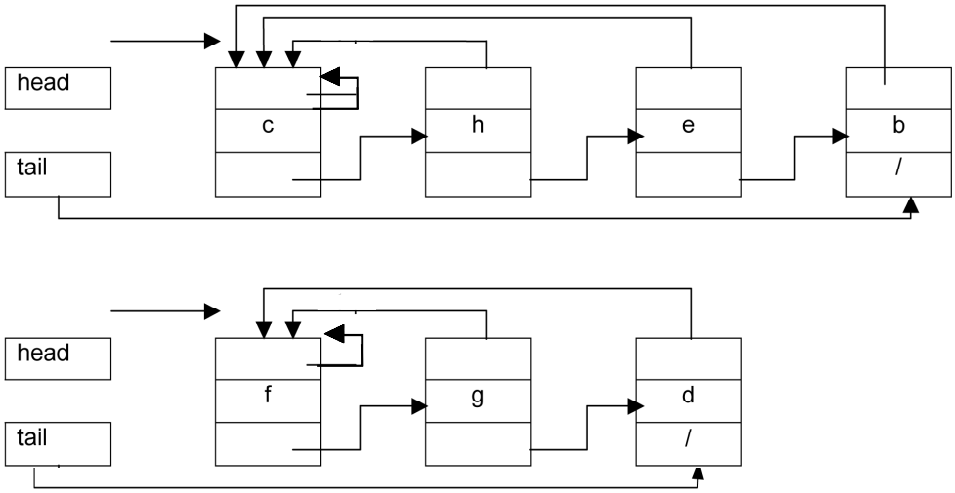


Figure 14.1: Linked list Tepresentation of Two Sets

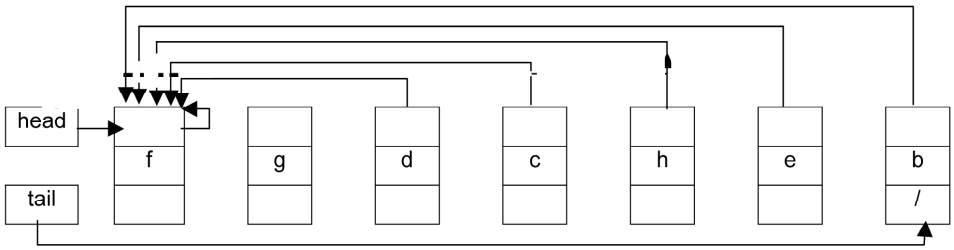


Figure 14.2: The Result of Union (e, g).



CHECK YOUR PROGRESS

Q.1: What is a partition?

Q.2: Two sets are disjoint if their intersection is the .....

Q.3: Define MakeSet() Operation.

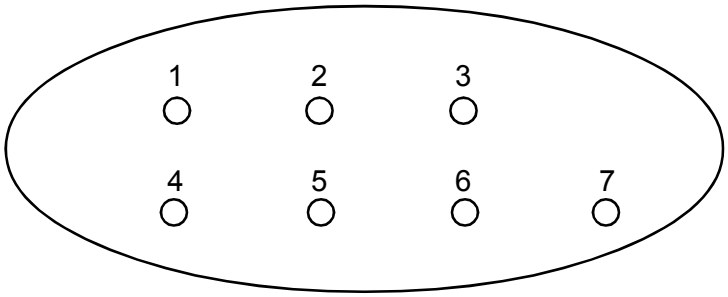
14.5 UNION

Union(x, y) operation is performed by appending x's list onto the end of y's list. The representative of the new set is the element that was originally the representative of the set containing y.

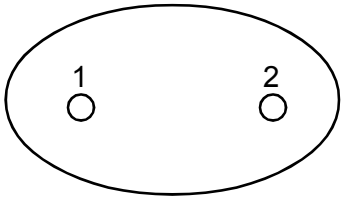
It is easy to implement with a sequence of m operations on n objects that requires $O(n^2)$ time. Suppose we have x1, x2, …, ..., ..., xn objects, then execution needs a sequence of n MakeSet operations followed by n-1 Union operations, so that m = 2n–1. Time required for performing n

MakeSet operations is O(n). Because ***i th*** Union operation updates i objects, the total number of objects updated by all n–1 operations in $O(n^2)$ time.

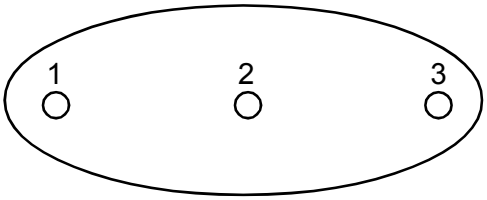
Let's discuss the disjoint operations with an example here. Suppose seven different elements are considered and all of them are in their own set, so it is called MakeSet() on each individual elements.



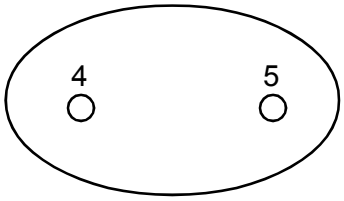
1) **Union (1, 2):** So it means that 1 and 2 will be in a set of its own and let's say 1 is a representative of this set. So findSet(1) will return 1 and findSet(2) will also return 1.



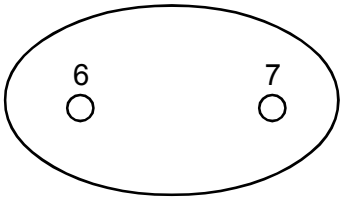
2) **Union (2, 3):** Here we need to merge the two sets together and let's consider 1 to be the representative of the set. Therefore findSet(1), findSet(2), and findSet(3) will return 1.



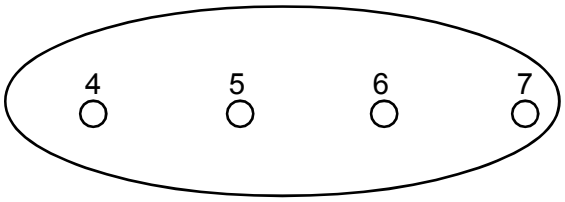
3) **Union (4, 5):** Here we need to merge 4 and 5 into one set and let's consider 4 is the representative of the set. Then findSet(4) and findSet(5) will return 4.



4) **Union (6, 7):** Here union of 6 and 7 will make a set of its own containing 6 and 7. Lets consider 6 is the representative of the set, so findSet(6) and findSet(7) both will return 6.

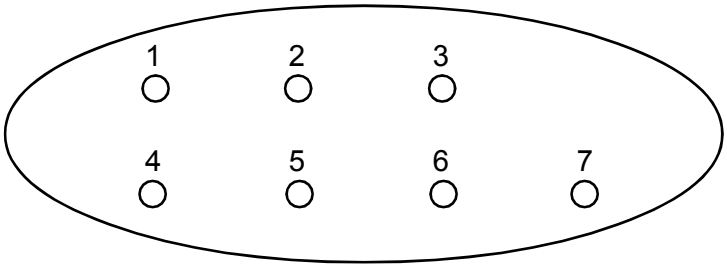


5) **Union (5, 6):** Next operation is union of 5 and 6. Here merge the two sets together in which one of which is 5 and one of which is 6. Let's consider 4 continue to be the representative of the set. After his operation the result will be as follows:



Here findSet(4), findSet(5), findSet(6), findSet(6) will return 4 because 4 is the representative of the combined set.

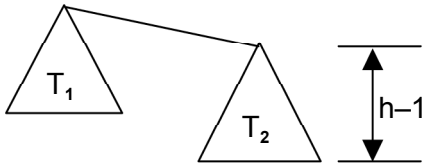
6) **Union (3, 7):** Here, we merge the two sets together in which one of sets is 3 and the other is 7. After this we will get a resultant set containing all the elements in the combined set. Let's consider 4 is the representative of the set. So findSet(1), findSet(2), findSet(3), findSet(4), findSet(5), findSet(6), findSet(7), all of these will return 4.



14.5.1 Weighted Merge

In weighted merge or weighted union the weight is equal to the number of nodes. When weighted merge or union operation is performed, then the smaller tree always points to the root of the

larger tree. With weighted union, an up-tree of height h has weight at least 2^h. Therefore, if h = 0, then the up-tree has one node, $2^0 = 1$.



Here, it is assumed that the weight of the tree T_1 is more than the tree T_2. It is also assumed that it is true for all h < h.

$W(T_1) \geq W(T_2) \geq 2^{h-1}$.

$W(T) \geq 2^{h-1} + 2^{h-1} + 2^{h-1} = 2^h$

Let T be an up-tree of weight n formed by weighted union. Let h be its height.

Then, $n > 2^h$

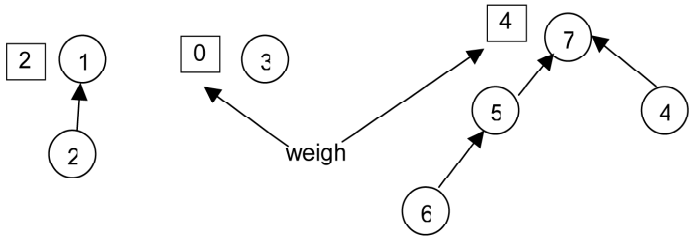
$\log_2 n > h$

Find(x) in tree T, takes O(log n) time.

Function 14.1: W-Union()

```
W_Union(I,j)
{
    //i and j are roots//
    wi := weight[i];
    wj := weight[j];
    if wi < wj then
        up[i] := j;
        weight[j] := wi + wj;
    else
        up[j] :=i;
        weight[i] := wi +wj;
}
```

The following figure shows that there are three different sets of nodes represented using trees. First tree consists of two nodes 1 and 2; second one has only one node 3; third one has 4 nodes 7, 5, 4 and 6. Array implementation is shown in the following diagram. Weight of root 1 is 2, weight of the root 3 is 1 and weight of the root 7 is 4.



Nodes	1	2	3	4	5	6	7
Up	0	1	0	7	7	5	0
Weight	2		1				4

Figure 14.3: Array Implementation of Weighted Union



CHECK YOUR PROGRESS

Q.4: What is an union() operation?

Q.5: What is weighted merge or union?

Q.6: With weighted union, an up-tree of height h has weight at least .....



14.6 LET US SUM UP

- A partition is a collection of disjoint components and its union will be the total collection.
- A disjoint set data structure maintains a collection s= { S1, S2, …, ..., ..., Sn} of disjoint dynamic sets.
- MakeSet(x): It is an operation to create a new set whose only member is x.
- Union(): It is an operation to merge two different sets into one set.
- FindSet(): It is an operation to return the identity of a set which is usually an element in a set which acts as a representative of that set.
- With weighted union, an up-tree of height h has weight at least 2h. Therefore, If h = 0, then the up-tree has one node, $2^0 = 1$.



14.7 FURTHER READING

1) Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. (2009). *Introduction to Algorithms*. MIT press.



14.8 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: Partition is dividing a group of objects into disjoint sets. A partition is a collection of disjoint components and its union will be the total collection.

Ans. to Q. No. 2: Two sets are disjoint if their intersection is the empty set.

Ans. to Q. No. 3: MakeSet(x): It is an operation to create a new set whose only member is x.

Ans. to Q. No. 4: Union():It is an operation to merge two different sets into one set.

Ans. to Q. No. 5: Weighted Union (weight = number of nodes). When weighted union operation is performed, then the smaller tree always points to the root of the larger tree.

Ans. to Q. No. 6: With weighted union, an up-tree of height h has weight at least 2^h.



14.9 MODEL QUESTIONS

Q.1: What is a partition?

Q.2: What are different Disjoint Set Operations? Explain with an example.

Q.3: What is a Union operation? Discuss the weighted union operation.

Q.4: Show the Linked List representation of disjoint sets with example.

Q.5: Explain partition with example.

*** ***** ***

UNIT 15: GRAPH

UNIT STRUCTURE

15.1 Learning Objectives

15.2 Introduction

15.3 Graph

15.4 Representation of Graphs

- 15.4.1 Adjacency Matrix Representation of Graph
- 15.4.2 Adjacency List Representation of Graph

15.5 Traversal of Graphs

- 15.5.1 Breadth First Traversal
- 15.5.2 Depth First Traversal

15.6 Let Us Sum Up

15.7 Further Reading

15.8 Answer to Check Your Progress

15.9 Model Questions

15.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- define graph and the terms associated with it
- explain the different representations of graphs
- describe breadth first and depth first traversal techniques.

15.2 INTRODUCTION

Till now, we have gone through many different data structures like linked list, stack, queue, and different types of trees. In this unit we are going to discuss the representation of graph in memory and present various traversal techniques of graph like depth first search and breadth first search.

15.3 GRAPH

We are already familiar with different types of tree data structure. Like trees, graphs are also nonlinear data structure. Tree has some specific

structure whereas graph does not have a specific structure. It varies from application to application in our daily life. Graphs are frequently used in every walk of life. A map is a well-known example of a graph. In a map various connections are made between the cities. The cities are connected via roads, railway lines and aerial network. For example, a graph that contains the cities of India connected by means of roads. We can assume that the graph is the interconnection of cities by roads. If we provide our office or home address to someone by drawing the roads, shops etc. in a piece of paper for easy representation, then that will also be a good example of graph.

Graph consists of a non empty set of points called *vertices* and a set of edges that link the vertices. Graph problem is originated from Konigsberg Bridge problem, Euler (1707-1783) in 1736 formulated the Konigsberg bridge problem as a graph problem.

Konigsberg Bridge Problem: Two river islands B and C are formed by the Parallel river in **Konigsberg** (then the capital of East Prussia, Now renamed Kaliningrad and in west Soviet Russia) were connected by seven bridges as shown in the figure below. The township people wondered whether they could start from any land areas walk over each bridge exactly once and return to the starting point.

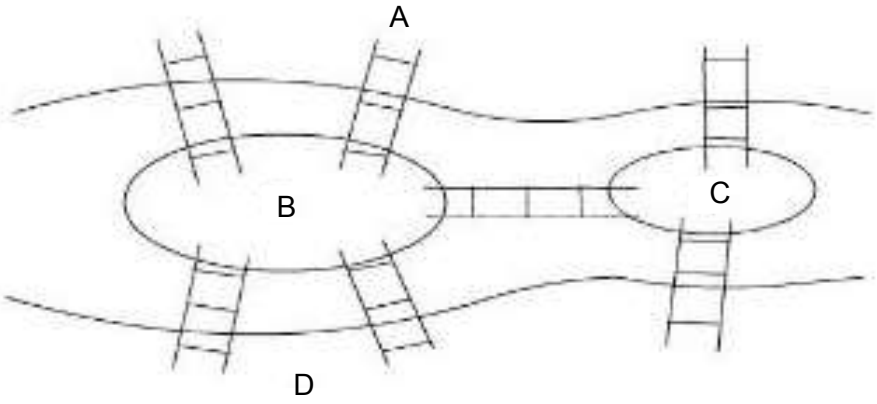


Figure 15.1: Konigsberg Bridge Problem

Definition: A graph G = (V, E) consists of

- a set V = {v1, v2, ..., ..., ..., vn} of n >1 vertices and
- a set of E = {e1, e2, ..., ..., ..., em} of m > 0 edges
- such that each edge ek is corresponds to an unordered pair of vertices (vi, vj) where 0 < i, j <= n and 0 < k <= m.

A road network is a simple example of a graph, in which vertices represents cities and road connecting them are correspond to edges.

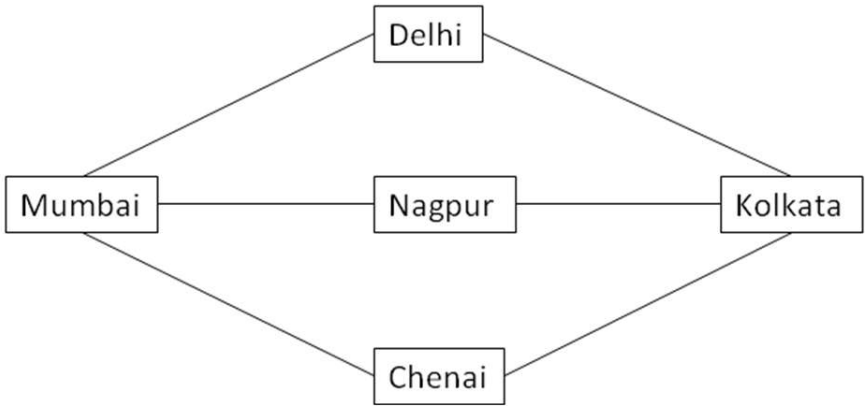


Figure 15.2: A Road Network

Here, V = {Delhi, Chenai, Kolkata, Mumbai, Nagpur}

E = {(Delhi, Kolkata), (Delhi, Mumbai}, (Delhi, Nagpur), (Chenai, Kolkata), (Chenai, Mumbai), (Chenai, Nagpur), (Kolkata, Nagpur), (Mumbai, Nagpur)}

Some of the terminology relating to a graph is given below:

Self Loop: An Edge having same vertices as its end vertices is called self loop. In figure 15.3 e3 is a self loop.

Parallel Edge: If more than one edges have the same pair of end vertices then the edges are called parallel edges. In figure 15.3 e6 and e7 are parallel edges.

Adjacent vertices: Two vertices x and y are said to be adjacent, if there is an edge from x to y or y to x. For example in figure 15.3 vertex A and B are adjacent but A and D are not adjacent.

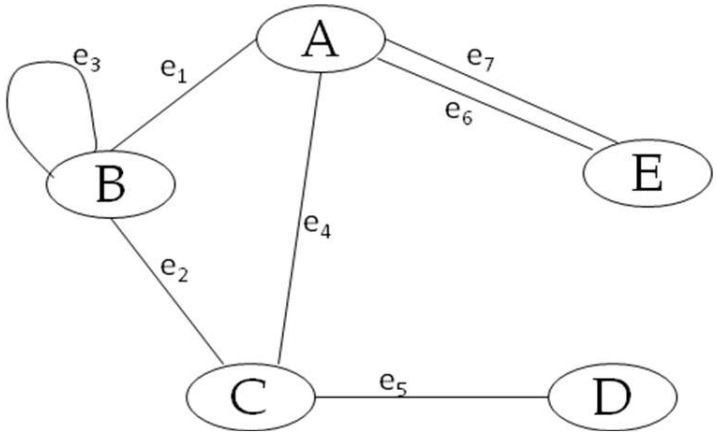


Figure 15.3: Example of Graph (Self Loop Parallel Edges)

Incidence: If an vertex vi is an end vertex of an edge ek , we say vertex vi is incident on ek and ek is incident on vi.

Simple Graph: A graph with no self loop and parallel edges is called a simple graph. For example figure 15.2 is a simple graph.

Degree of a vertex: The degree of a vertex x, denoted as degree(x), is the number of edges containing x. In figure 15.3 vertex A has degree 4.

Path: A path of length k from vertex x to a vertex y is defined as a sequence of K+1 vertices V1, V2, ..., ..., ..., Vk+1 such that V1 = x and Vk+1 = y and Vi is adjacent to Vi+1. For example A, B, C, D is a path of length 3 where A is the starting vertex of the path and D is the end vertex of the path. If the starting and end vertex of a path is same then it is said to be a closed path. If all the vertices of the path except the starting and ending vertex are distinct then it is said to be a simple path.

Cycle: A cycle is a closed simple path. If a graph contains a cycle, it is called cyclic graph. In figure 15.3, B, A, C, B is a cycle.

Walk: A walk is an alternative sequence of vertices and edge, starting and ending with vertices in such a way that no vertices in the sequence are repeated. In a walk starting and ending vertices are called terminal vertices of the walk.

Connected Graph: A graph is said to be connected, if there exist a path between every pair of vertices. For example, graph in figure 15.4 is not connected but the graph in figure 15.3 is connected.

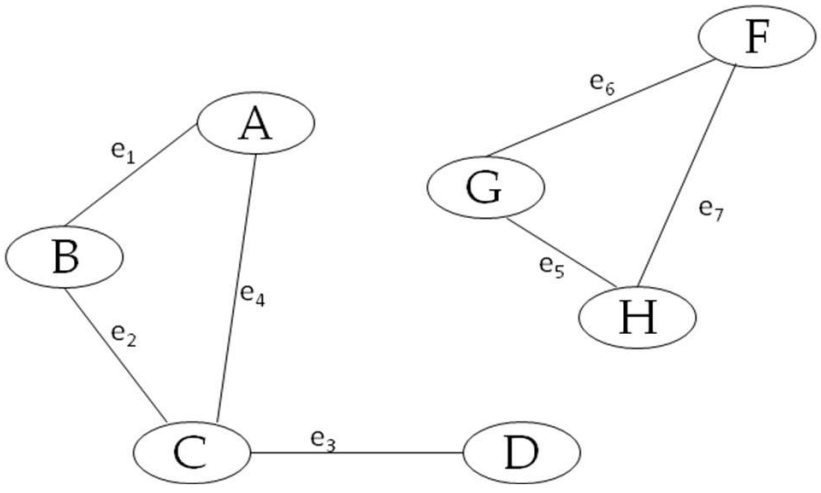


Figure 15.4 Example of Disconnected Graph

Weighted Graph: A graph is said to be a weighted graph, if there exist a nonnegative value (called weight) is associated with every edge in the graph. Figure 15.5 is an example of weighted graph.

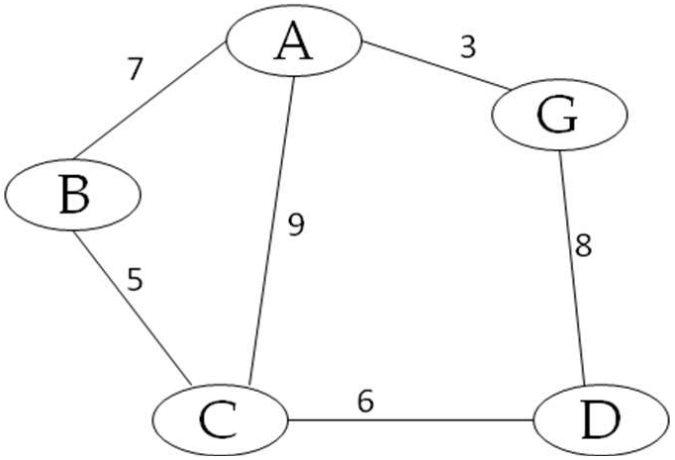


Figure 15.5: A Weighted Graph

Directed Graph: A graph is said to be a directed graph, if the edges of the graph has a direction. In case of directed graph edge (A, B) is not same with edge (B, A). Figure 15.6 is an example of directed graph.

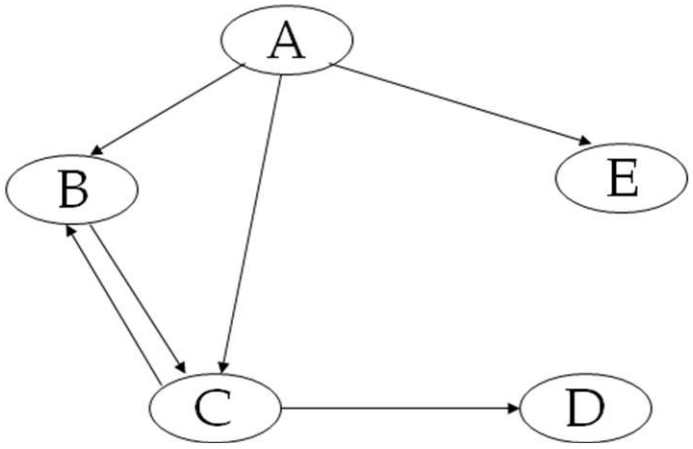


Figure 15.6 Directed Graph

Indegree of a Vertex: Indegree of a vertex x, denoted by indeg(x) refers to the number of edges terminated at x. For example in figure 15.6 indegree of vertex c is 2.

Outdegree of a Vertex: Outdegree of a vertex x, denoted by outdeg(x) refers to the number of edges originated from x. For example, in figure 15.6 outdegree of vertex c is 1.

Strongly connected: A directed graph is said to be connected or strongly connected if for every pair of vertices <x, y>, there exist a directed path from x to y and y to x. However, if there exist any pair <x, y> such that there is a path either form x to y or from y to x but not the both, then the graph is said to be weakly connected.

15.4 REPRESENTATION OF GRAPHS

There are several different ways to represent a graph in computer memory. Two main representations are:

- Adjacency Matrix
- Adjacency list.

Let us understand both these representations of graph in detail now.

15.4.1 Adjacency Matrix Representation of Graph

Suppose G(V, E) is simple graph (directed / undirected) with n vertices, and suppose G have been ordered and are called v1, v2, ..., ..., ..., vn. Then the adjacency matrix A = (aij) of the graph G is a n x n matrix defined as follows:

$$a_{ij} = \begin{cases} 1 & \text{if } v_i \text{ is adjacent to } v_j, \text{ that is if there exist an edge } (v_i, v_j) \\ 0 & \text{otherwise} \end{cases}$$

Such a matrix contain only 0 and 1 is called a Boolean Matrix.

For example, the following graph in figure 15.7 can be represented as follows:

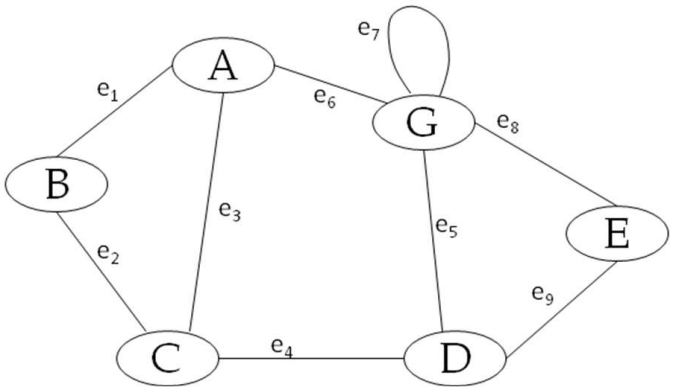


Figure 15.7 Graph Example

$$A = \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \\ G \end{array} \begin{array}{c} \begin{array}{cccccc} A & B & C & D & E & G \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 \end{pmatrix} \end{array}$$

The directed graph in figure 15.6 can be represented as follows:

$$A = \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

For Weighted Graph: Suppose G(V,E) is a weighted graph with n vertices v1, v2, ..., ..., ..., vn. Suppose each edge e in G is assigned a non negative number w(e) called the weight or length of the edge e. Then G can be represented in memory by its weight matrix W = (Wij) as follows:

$$w_{ij} = \begin{cases} w(e) & \text{if there exist an edge e from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

For example, the weighted graph in figure 15.5 can be represented as follows.

$$A = \begin{array}{c} \\ A \\ B \\ C \\ D \\ G \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & G \end{array} \\ \begin{pmatrix} 0 & 7 & 9 & 0 & 3 \\ 7 & 0 & 5 & 0 & 0 \\ 9 & 5 & 0 & 6 & 0 \\ 0 & 0 & 6 & 0 & 8 \\ 3 & 0 & 0 & 8 & 0 \end{pmatrix} \end{array}$$

Although an adjacency matrix is a simple way to represent a graph it needs n^2 bit to represent a graph with n vertices. In case of undirected graph storing only upper or lower triangle of the matrix,

this space can be reduced to half. However in case of a directed graph this is not possible. Moreover in case of undirected graph parallel edges cannot be represented by this method.

15.4.2 Adjacency List Representation of Graph

Suppose, G(V, E) is a simple graph (directed/undirected) with n vertices and e edges, the adjacency list have n head nodes corresponding to the n vertices of graph G, each of which point to a singly link list of nodes adjacent to the vertex representing to the head node.

In contrast to adjacency matrix representations, graph algorithm which makes use of an adjacency list representation would generally report a complexity of O(n+e) or O(n+2e) based on whether graph is directed or undirected respectively, thereby rendering them efficient.

For example, the directed graph in figure 15.6 can be represented in adjacency list representation as follows.

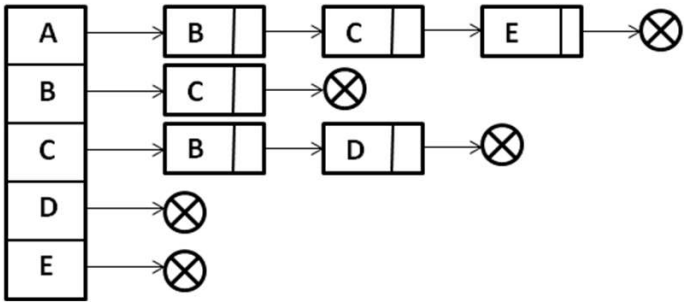


Figure 15.6: Directed Graph

For example, the undirected graph in figure 15.7 can be represented in adjacency list representation as follows.

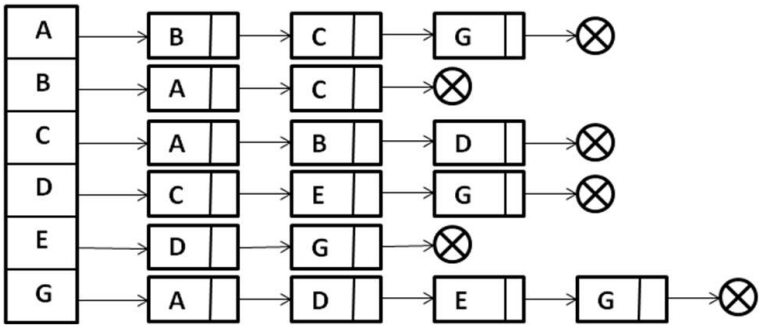


Figure 15.7: Undirected Graph



CHECK YOUR PROGRESS

Q.1: State True or False:

a) A simple graph can have parallel edges.

b) In a directed graph indegree of a vertex is the number of edge incident on to the vertex.

c) If two vertices connected by a path then they are called adjacent vertices.

d) For an undirected graph parallel edges can be represented in the adjacency matrix representation of graph.

15.5 TRAVERSAL OF GRAPHS

Traversal of a graph is a systematic walk which visits all the vertices in a specific order. There are mainly two different ways of traversing a graph:

- Breadth first traversal
- Depth first traversal

Let us understand both these traversals in detail now.

15.5.1 Breadth First Traversal

The general idea behind breadth first traversal is that, when we start at a random vertex, then visit all of its neighbors, the fist vertex that we visit, say is at level '0' and the neighbors are at level '1'. After visiting all the vertices at level '1' we then pick one of these vertex at level '1' and visit all its unvisited neighbors, we repeat this procedure for all other vertices at level '1'. Say neighbors of level 1 are in level 2, now we will visit the neighbors of all the vertices at level 2, and this procedure will continue.

Algorithm 15.1: Breadth First Traversal BFS()

Step1: Initialize all the vertices to ready state (STATUS = 1)

Step2: Put the starting vertex into QUEUE and change its status to waiting (STATUS = 2)

Step 3: Repeat Step 4 and 5 until QUEUE is EMPTY

Step 4: Remove the front vertex from QUEUE, Process the vertex, Change its status to processed state (STATUS = 3)
Step 5: ADD all the neighbors in the ready state (STATUS = 1) to the RARE of the QUEUE and change their status to waiting state (STATUS = 2)
Step 6: Exit.

The C program for breadth-first traversal of a graph is given in program 15.1. The program makes use of an array of *n* visited elements where *n* is the number of vertices of the graph. If visited[i] = 1, it means that the i^{th} vertex is visited. The program also makes use of a queue and the procedures addqueue and deletequeue for adding a vertex to the queue and for deleting the vertex from the queue, respectively. Initially, we set visited[i] = 0.

Program 15.1: Program for breadth-first traversal of a graph

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 10
struct node
    {
        int data;
        struct node *link;
    };

void buildadjm(int adj[][MAX], int n)
    {
        int i,j;
        printf("enter adjacency matrix \n",i,j);
        for(i=0; i<n; i++)
            for(j=0; j<n; j++)
                scanf("%d",&adj[i][j]);
    }
```

```
/* A function to insert a new node in queue*/
struct node *addqueue(struct node *p,int val)
{
    struct node *temp;
    if(p == NULL)
{
    p = (struct node *) malloc(sizeof(struct node));
        /*insert the new node first node*/
    if(p == NULL)
    {
        printf("Cannot allocate\n");
        exit(0);
    }
    p->data = val;
    p->link=NULL;
}
else
{
    temp= p;
    while(temp->link != NULL)
        {
            temp = temp->link;
        }
temp->link= (struct node*)malloc(sizeof(struct
node));
    temp = temp->link;
    if(temp == NULL)
        {
            printf("Cannot allocate\n");
            exit(0);
        }
    temp->data = val;
    temp->link = NULL;
    }
```

```
        return(p);
}

struct node *deleteq(struct node *p,int *val)
{
    struct node *temp;
    if(p == NULL)
    {
        printf("queue is empty\n");
            return(NULL);
    }
    *val = p->data;
    temp = p;
    p = p->link;
    free(temp);
    return(p);
}

void bfs(int adj[][MAX], int x,int visited[], int
n, struct node **p)
{
    int y, j, k;
    *p = addqueue(*p,x);
    do{

        *p = deleteq(*p,&y);
        if(visited[y] == 0)
        {
            printf("\nnode visited = %d\t",y);
            visited[y] = 1;
            for(j=0;j<n;j++)
            if((adj[y][j] ==1) && (visited[j] == 0))
                *p = addqueue(*p,j);
        }
```

```
    }while((*p) != NULL);
}

void main()
    {
        int adj[MAX][MAX];
        int n;
        struct node*start=NULL;
        int i, visited[MAX];

        printf("enter the number of nodes in graph
        maximum = %d\n",MAX);
        scanf("%d",&n);
        buildadjm(adj,n);
        for(i=0; i<n; i++)
            visited[i] =0;
        for(i=0; i<n; i++)
        if(visited[i] ==0)
        bfs(adj,i,visited,n,&start);
    }
```

Output:

Enter the number of nodes in graph maximum = 10 9

Enter adjacency matrix

0 1 0 0 1 0 0 0 0

1 0 1 1 0 0 0 0 0

0 1 0 0 0 0 1 0 0

0 1 0 0 1 1 0 0 0

1 0 0 1 0 0 0 0 0

0 0 0 1 0 0 0 0 1

0 0 1 0 0 0 0 1 1

0 0 0 0 0 0 1 0 0

0 0 0 0 0 1 1 0 0

node visited = 0
node visited = 1
node visited = 4
node visited = 2
node visited = 3
node visited = 6
node visited = 5
node visited = 7
node visited = 8

15.5.2 Depth First Traversal

The general idea behind depth first traversal is that, starting from any random vertex, single path is traversed until a vertex is found whose all the neighbors are already been visited. The search then backtracks on the path until a vertex with unvisited adjacent vertices is found and then begin traversing a new path starting from that vertex, and so on. This process will continue until all the vertices of the graph are visited.

Algorithm 15.2: Depth First Traversal DFS()

Step1:	Initialize all the vertices to ready state (STATUS = 1)
Step2:	Put the starting vertex into STACK and change its status to waiting (STATUS = 2)
Step 3:	Repeat Step 4 and 5 until STACK is EMPTY
Step 4:	POP the top vertex from STACK, Process the vertex, Change its status to processed state (STATUS = 3)
Step 5:	PUSH all the neighbors in the ready state (STATUS = 1) to the STACK and change their status to waiting state (STATUS = 2)
Step 6:	Exit.

The C program for depth-first traversal of a graph is provided in program 15.2. It makes use of an array visited of n elements where n is the number of vertices of the graph, and the elements are Boolean. If visited[i] = 1 then it means that the i^{th} vertex is visited. Initially we set visited[i] = 0.

Program 15.2: Program for depth-first traversal of a graph

```
#include <stdio.h>
#define max 10
/* a function to build adjacency matrix of a graph */
void buildadjm(int adj[][max], int n)
    {
        int i,j;
        for(i=0;i<n;i++)
            for(j=0; j<n; j++)
            {
        printf("enter 1 if there is an edge from %d
        to %d, otherwise enter0\n", i, j);

            scanf("%d",&adj[i][j]);
            }
        }

/ *a function to visit the nodes in a depth-first
order */
void dfs(int x,int visited[],int adj[][max],int n)
{
    int j;
    visited[x] = 1;
    printf("The node visited id %d\n",x);
    for(j=0;j<n; j++)
        if(adj[x][j] ==1 && visited[j] ==0)
            dfs(j,visited,adj,n);
}
void main()
{
    int adj[max][max],node,n;
    int i, visited[max];
```

```
printf("enter the number of nodes in graph
maximum = %d\n",max);
scanf("%d",&n);
buildadjm(adj,n);
for(i=0; i<n; i++)
    visited[i] =0;
for(i=0; i<n; i++)
    if(visited[i] ==0)
        dfs(i,visited,adj,n);
}
```

The explanation for the program given in 15.2 is provided below:

1) Initially, all the elements of an array named visited are set to 0 to indicate that all the vertices are unvisited.
2) The traversal starts with the first vertex (that is, vertex 0), and marks it visited by setting visited[0] to 1. It then considers one of the unvisited vertices adjacent to it and marks it visited, then repeats the process by considering one of its unvisited adjacent vertices.
3) Therefore, if the following adjacency matrix that represents the graph of Figure 22.9 is given as input, the order in which the nodes are visited is given here:

Input:1) The number of nodes in a graph

2) Information about edges, in the form of values to be stored in adjacency matrix 1 if there is an edge from node i to node j, 0 otherwise

Output: Depth-first ordering of the nodes of the graph starting from the initial vertex, which is vertex 0, in our case.



CHECK YOUR PROGRESS

Q.5: The most commonly used methods for traversing a graph are ..... and .....

Q.6: In depth first traversal we use ..... (Stack/Queue)



15.6 LET US SUM UP

- A graph is a structure that is often used to model the arbitrary relationships among the data objects while solving many problems.
- A graph is a structure made of two components: a set of vertices V, and the set of edges E. Therefore, a graph is G = (V, E), where G is a graph.
- The graph may be directed or undirected. When the graph is directed, every edge of the graph is an ordered pair of vertices connected by the edge.
- When the graph is undirected, every edge of the graph is an unordered pair of vertices connected by the edge.
- Adjacency matrices and adjacency lists are used to represent graphs.
- A graph can be traversed by using either the depth first traversal or the breadth first traversal.
- When a graph is traversed by visiting in the forward (deeper) direction as long as possible, the traversal is called depth first traversal.
- When a graph is traversed by visiting all the adjacencies of a node/ vertex first, the traversal is called breadth first traversal.



15.7 FURTHER READING

1) Aho, A. V. & Hopcroft, J. E. (1974). *The Design and Analysis of Computer Algorithms*. Pearson Education India.
2) Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. (2009). *Introduction to Algorithms*. MIT press.
3) Horowitz, E., Sahni, S. & Rajasekaran, S. (1997). *Computer Algorithms C++: C++ and Pseudocode Versions*. Macmillan.
4) Horowitz, E., Sahni, S. & Anderson-Freed, S. (1983). *Fundamentals of Data Structures* (Vol. 20). Rockville, MD: Computer Science Press.

15.8 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: a) False, b) True, c) False, d) False

Ans. to Q. No. 5: Depth first traversal and breadth first traversal.

Ans. to Q. No. 6: Stack

15.9 MODEL QUESTIONS

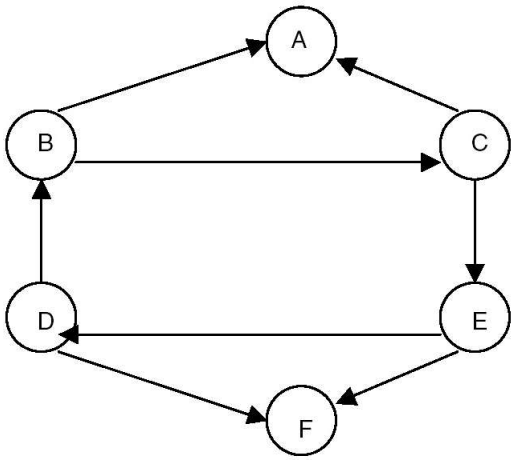
Q.1: Define the following terms:

i) Graph ii) Degree of a vertex

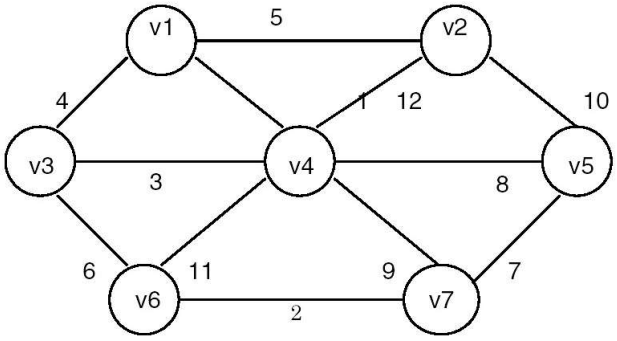
iii) Weighted Graph iv) Path

v) Strongly connected Graph

Q.2: For the following graph give the adjacency list and adjacency matrix representation. And find out the order in which graph will be traversed in depth first traversal, by taking B as the starting vertex.



Q.3: For the following graph give the adjacency matrix representation. And find out the order in which graph will be traversed in breadth first traversal, by taking v3 as the starting vertex.



Q.4: Describe breadth first traversal technique using example.

Q.5: Explain depth first traversal technique using example.

*** ***** ***



REFERENCES

1) Aho, A. V. & Hopcroft, J. E. (1999). *The Design and Analysis of Computer Algorithms*. Pearson Education India.
2) Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. (2009). *Introduction to Algorithms*, MIT Press.
3) Horowitz, E., Sahni, S. & Rajasekaran, S. (2009). *Fundamental of Data Structure in C*.
4) Horowitz, E. (2006). *Fundamentals of Data Structures in C++*. Galgotia Publications.
5) Horowitz, E., Sahni, S. & Anderson-Freed, S. (1983). *Fundamentals of Data Structures*. (Vol. 20). Rockville, MD: Computer Science Press.
6) Horowitz, E., Sahni, S. & Rajasekaran, S. (1997). *Computer Algorithms C++: C++ and Pseudocode Versions*. Macmillan.
7) Horowitz, E., Sahni, S. & Anderson-Freed, S. (1983). *Fundamentals of Data Structures* (Vol. 20). Rockville, MD: Computer Science Press.
8) Kamthane, Ashok N. (2004). *Introduction to Data Structures in C*. Perason Education.
9) Kamthane, A. (2009). *Programming with ANSI and Turbo C*. Pearson Education India.
10) Langsam, Y., Augenstein, M. & Tenenbaum, A. M. (1996). *Data Structures Using C and C++*. (Vol. 2). New Jersey: Prentice Hall.
11) Lipschutz, Seymour. *Data Structures*. Schaum's Outline Series in Computers. Tata McGraw Hill.
12) Lipschutz, S. (1987). *Schaum's Outline of Data Structure*. McGraw-Hill, Inc.
13) Robert, L. K., Tondo, C. L., & Leung, B. P. (2002). *Data Structures And Program Design*.
14) Samanta, D. (2004). Classic Data Structures. PHI Learning Pvt. Ltd.
15) Seymour, L. (2010). *Data Structures with C*. Tata McGraw Hill.

16) Silberschatz, A., Korth, H. F. & Sudarshan, S. (1997). *Database System Concepts* (Vol. 4). New York: McGraw-Hill.
17) Srivastava, S. K. & Srivastava, D. (2003). *Data Structures Through C in Depth*. BPB Publications.
18) Thareja, R. (2011). *Data Structures Using C*. Oxford University Press, Inc.
19) http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6.

Name of the Paper: .....
Course Code: .....

Centre for Internal Quality Assurance (CIQA)
Krishna Kanta Handiqui State Open University
City Office: Housefed Complex, Guwahati-781006

Learner's Feedback on Course

Dear Learner,

Regarding the course as mentioned above, we would like to know your opinions and comments so as to improve the quality of self learning materials in future. Please respond to the following statements by ticking the number you feel most reflect your opinion. After completion of the additional comments, please detach the page and send/mail the same to us at the address given below.

The Director, Centre for Internal Quality Assurance, KKHandiqui State Open University
Housefed Complex, Dispur, Guwahati-781006
(E-mail id: ciqa@kkhsou.in)

1) Approximately how many hours did you spend for studying the units in the course? ☐☐☐

2) Please give your opinions (by ✓ mark) to the following items based on your reading of the block:

Sl. No.	Statements	Strongly Agree	Agree	Neither Agree nor Disagree	Disagree	Strongly Disagree
I)	The SLMs of the course fulfil the learning objectives set out in the beginning of each unit					
II)	The units of the SLM could satisfy my academic needs and expectations					
III)	The Unit writers have excellent knowledge about the course contents					
IV)	Language and contents of the units were presented at a level which I could readily understand					
V)	Ample opportunity for participation in the activities provided in the units					
VI)	Used enough Illustrations (Diagrams, tables etc.) for conceptual clarity					
VII)	Quality of content is engaging, relevant, and up-to-date					
VIII)	The self check questions are very helpful					
IX)	The Possible/model questions and the answers to check my progress have benefited me a lot					

Additional Comments: *(Please feel free to provide your open comments)*

1) Which aspects of the SLM, according to you, worked well?

.....

.....

2) What sort of changes/improvements do you feel KKHSOU could implement to improve the overall quality of the SLM?

.....

.....

Thank you for taking the time to complete this form.