MCA06

KRISHNA KANTA HANDIQUI STATE OPEN UNIVERSITY

Housefed Complex, Dispur, Guwahati - 781 006

Master of Computer Applications

DATA STRUCTURE THROUGH C LANGUAGE

CONTENTS

Subject Experts

1. Prof. Anjana Kakati Mahanta, Deptt. of Computer Science, Gauhati University
2. Prof. Jatindra Kr. Deka, Deptt. of Computer Science and Engineering, Indian Institute of Technology, Guwahati
3. Prof. Diganta Goswami, Deptt. of Computer Science and Engineering, Indian Institute of Technology, Guwahati

Course Coordinators

Tapashi Kashyap Das, Assistant Professor, Computer Science, KKHSOU
Arabinda Saikia, Assistant Professor, Computer Science, KKHSOU

SLM Preparation Team

UNITS	CONTRIBUTORS
1, 7	Arabinda Saikia, KKHSOU
2, 8, 9	Nabajyoti Sarma, Research Scholar, Deptt. of Computer Science, Gauhati University
3, 5	Swapnanil Gogoi, Assistant Professor, IDOL, Gauhati University
4, 6	Tapashi Kashyap Das, KKHSOU

December, 2011

The university acknowledges with thanks the financial support provided by the **Distance Education Council, New Delhi**, for the preparation of this study material.

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

Housefed Complex, Dispur, Guwahati- 781006; Web: www.kkhsou.org

COURSE INTRODUCTION

This is a course on "***Data Structure through C Language***". A data structure is a particular way of storing and organizing data in a computer's memory or even disk storage so that it can be used efficiently. Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks. The commonly used data structures in various programming languages, like C, are arrays, linked list, stack, queues, tree, graph etc. This course is designed to acquaint the learner such type of data structures.

This course comprises of nine units which are as follows:

Unit - 1 introduces you some elementary concepts like data, types of data, structure, pointer, array etc. as well as memory representation, address translation functions.

Unit - 2 concentrates on algorithms, complexity of algorithm in terms of time and space and their notations.

Unit - 3 deals with one of the most important linear data structure linked list. Representation of linked list, their types, operations associated with linked list like searching, insertion and deletion of element in a linked are described in this unit.

Unit - 4 focuses on stack data structure. In this unit various operations associted with stacks as well as their implementation using array and linked list are discussed.

Unit - 5 concentrates on queue data structure. This unit discusses array as well as linked implementation of queue, applications of queue etc. Concept of circular queue, priority queue are also covered in this unit.

Unit - 6 deals with the searching techniques. Linear and binary search techniques with their relative advantages and disadvantages are discussed in this unit.

Unit -7 discusses different sorting techniques, their implementations, complexity, advantages and disadvantages.

Unit - 8 concentrates on a new data structure called trees. This unit discusses binary tree, tree traversal methods, different notations like postfix, prefix etc. are discussed in this unit. Binary search tree, operations like searching, insertion and deletion on binary search tree are also discussed in this unit.

Unit - 9 is the last unit of this course. This unit focuses on an important data structure called graph. Graph representations as well as graph traversal techniques are illustrated in this unit.

Each unit of this course 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. You may find some boxes marked with: "LET US KNOW". These boxes will provide you with some additional interesting and relevant information. Again, you will get "CHECK YOUR PROGRESS" questions. These have been designed to self-check your progress of study. It will be helpful for you 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 PROGRESS" given at the end of each unit.

MASTER OF COMPUTER APPLICATIONS

Data Structure Through C Language

DETAILED SYLLABUS

UNIT 1 : INTRODUCTION TO DATA STRUCTURE

UNIT STRUCTURE

1.1 LEARNING OBJECTIVES

After going through this unit, you will able to :

- distinguish data and information
- learn about data structure
- define various types of data structures
- know different data structure operations
- describe about data types in C
- define abstract data types

1.2 INTRODUCTION

A **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. In this unit, we shall discuss about the data and information and overview of data structure. We will also discuss the types of data structure, data structure operations and basic concept of data types.

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 12^{th} 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.

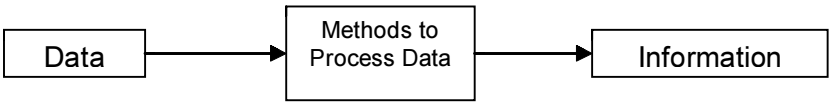


Fig 1.1 : Relation between data and information

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

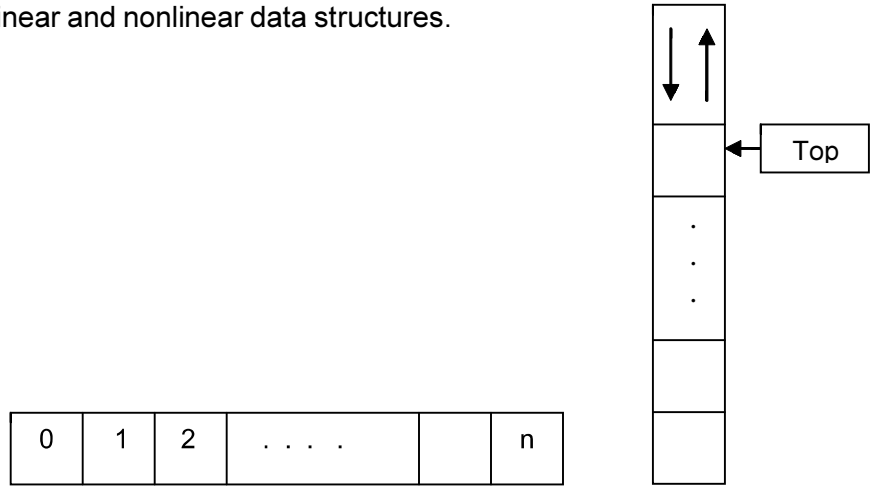
– creating standard, reusable code components

– 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.



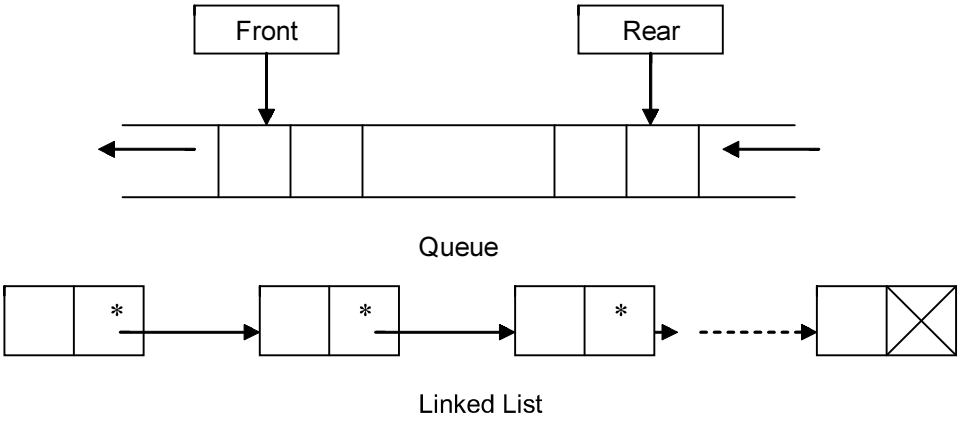


Fig 1.2 : Linear data structure

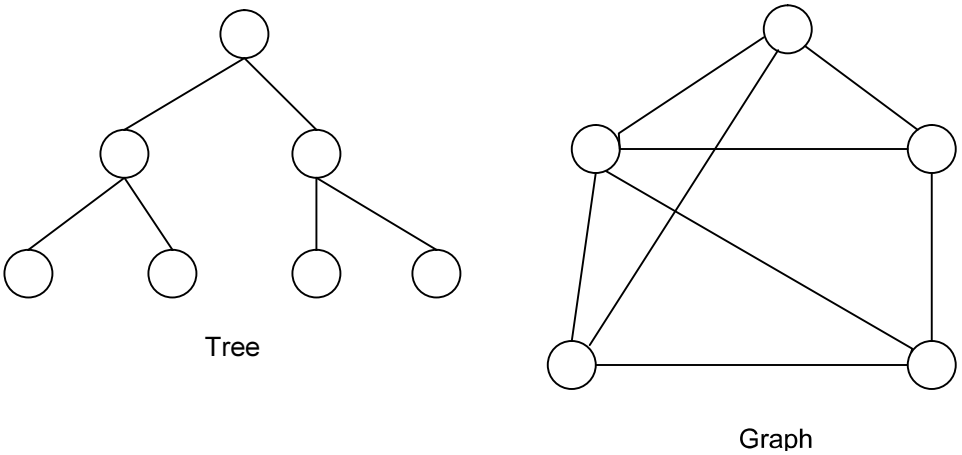


Fig 1.3 : Non linear data structure

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

We have already familiar with the term '*data type*'. 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 about the data types array, pointers and structures.

i) 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

Storage Class datatype arrayname [expression]

Here, *storage class* may be *auto, static or extern*, which you just remember, *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 fig.

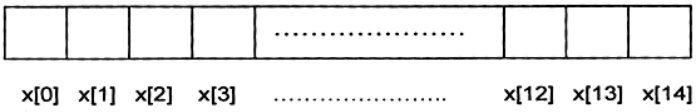


Fig 1.4 : An array data structure

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]

Fig 1.5 : the marks array after initialization

In the case of a character array *name* we get it 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]

Fig 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:

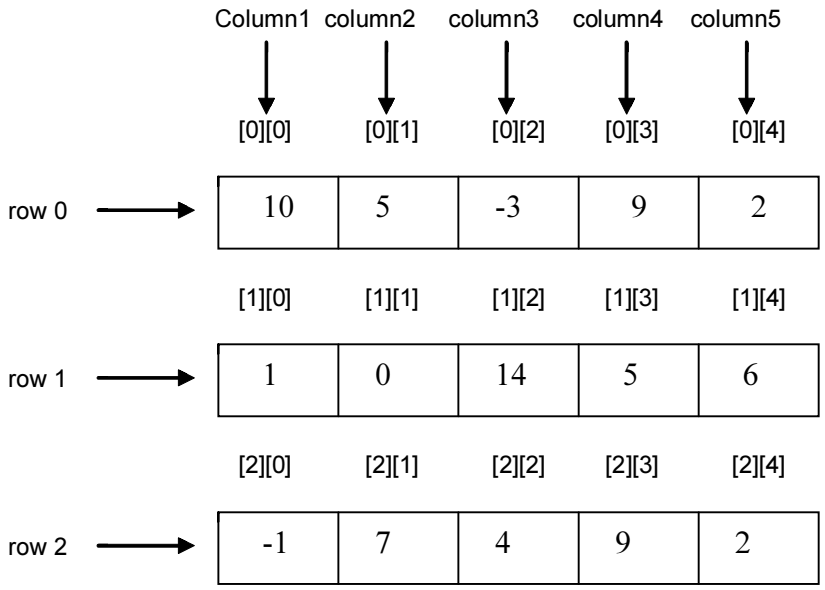


Fig. 1.7 : Matrix array after initialization

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.

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( );
}
```

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();
}
```

ii) Pointers : You have already introduced to the concept of pointer. At this moment 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 Fig.

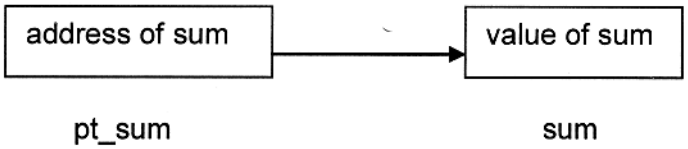


Fig. 1.8 : Relationship between pt_sum and sum

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.

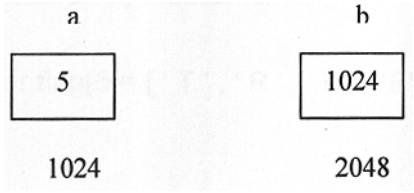
The declaration **float *s** does not mean that **s** is going to contain a floating-point value. What it means is, **s** is going to contain the address of a floating-point value. Similarly, **char *ch** means that **ch** is going to contain the address of a char value.

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

```
#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]



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 :

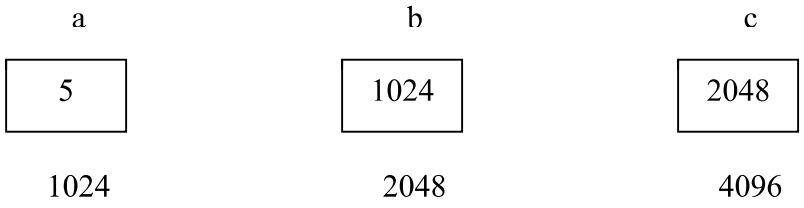
int **x;

Following program shows the use of pointer to pointer techniques :

```
#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]



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"};

iii) 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**. Recall that 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 :

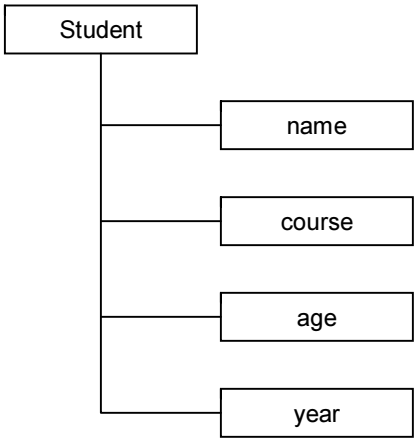


Fig. 1.9 : A structure named Student

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 };

A simple example of using of the structure data type is shown below :

```
#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 :

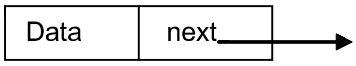


Fig. 1.10 : A node structure

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 :

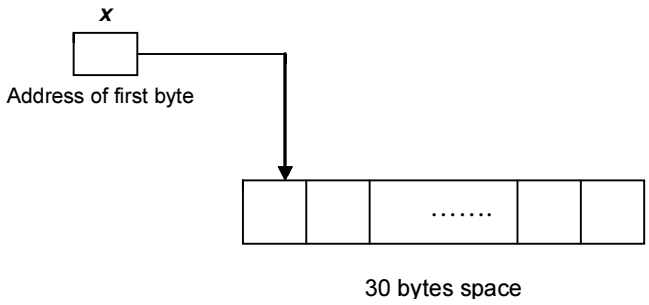


Fig. 1.11 : Representation of dynamic memory allocation

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 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

You are well acquainted with data types by now, 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 & -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

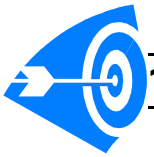
i) abstract data type ii) classic data type

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 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 & 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.

1.11 FURTHER READINGS

- *Data structures using C and C++*, Yedidyah Langsam, Moshe J. Augenstein, Aaron M.Tenenbaum, Prentice-Hall India.
- *Data Structures*, Seymour Lipschutz, Schaum's Outline Series in Computers,Tata Mc Graw Hill
- *Introduction to Data Structures in C*, Ashok N. Kamthane, Perason Education.

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?

UNIT 2 : ALGORITHM

UNIT STRUCTURE

2.1 LEARNING OBJECTIVES

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

- understand the concept of algorithm
- know the notations for defining the complexity of algorithm
- learn the method to calculate time complexity of algorithm

2.2 INTRODUCTION

The concept of an algorithm is the basic need of any programming development in computer science. 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. In this unit we will discuss about the algorithm and its complexity. Also we will discuss the asymptotic notation of algorithms.

2.3 DEFINITION OF ALGORITHM

Definition: 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 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 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 d" a'_2 d" a'_3 d" ……d" 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 place 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 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 broadly categorized in two classes

- Space complexity
- Time complexity

2.4.1 SPACE COMPLEXITY

Now a day's, memory is becoming more and more cheaper, even though it is very much important to analyze the amount of memory used by a program. Because, if the algorithm takes memory beyond the capacity of the machine, then the algorithm will not able to execute. So, it is very much important to analyze the space complexity before execute it on the computer.

Definition [Space Complexity]: 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 has 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 depend on the machine where it is executed.

Consider the following algorithms for exchange two numbers:

```
Algo1_exchange (a, b)
    Step 1: tmp = a;
    Step 2: a = b;
    Step 3: b = tmp;
```

```
Algo2_exchange (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 tmp and the second one take 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

Definition [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 has 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 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 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

```
Algo_add (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

```
Algo_addeven (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,

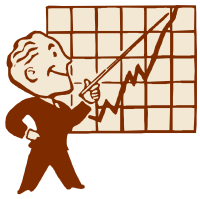
Step 1, Step 2 and Step 7 are simple statement and they will execute only once.

Step 3 is a loop statement and it will execute as many times the loop condition is true and once more time for check 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 indicate the end of while and no cost 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) Algorithm 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. For simplify this complex polynomial we use some notation to represent the complexity of an algorithm call Asymptotic Notation.

Θ (Theta) Notation

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

$$\Theta(g(n)) = \begin{cases} f(n) : \text{there exist constants } c_1 > 0,\ c_2 > 0 \text{ and } n_0 \text{ õ } N \\ \qquad \text{such that } 0 \text{ d" } c_1\, g(n) \text{ d" } f(n) \text{ d" } c_2\, g(n) \text{ for all } n \text{ e" } n_0 \end{cases}$$

In other words a function f(n) is said to belongs to Θ(g(n)), if there exists positive constants c_1 and c_2 such that 0 d" c_1 g(n) d" f(n) d" c_2g(n) for sufficiently large value of n. Fig 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_1g(n) and at or below c_2g(n). In other words, for all n e" n_0, the function f(n) is equal to g(n) to within a constant factor. So, g(n) is said an **asymptotically tight bound** for f(n).

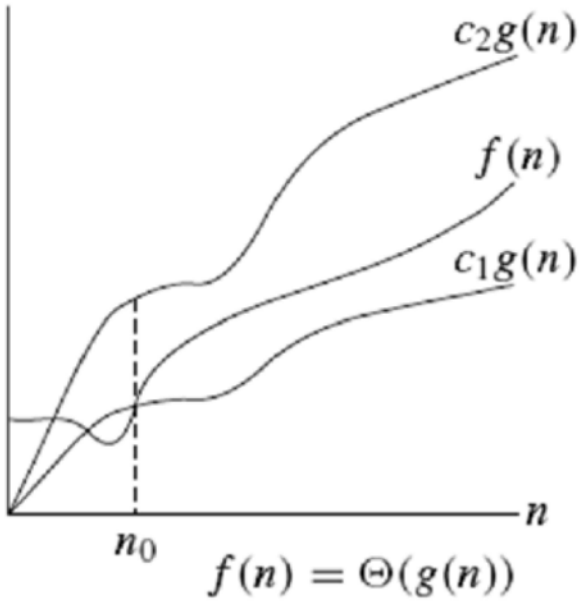


Fig 2.1 : Graphic Example of Θ notation.

For example

f(n) = ½ n^2 -3 n

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

$c_1 n^2$ d" ½ n^2 -3 n d" $c_2 n^2$

for all n e" n_0

dividing the whole equation by n^2, we get

c_1 d" ½ -3/n d" c_2

We can make the right hand inequality hold for any value of n e" 1 by choosing c_2 e" ½. Similarly we can make the left hand inequality hold for any value of n e" 7 by choosing c_1 d" $^1/_{14}$. 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 -3 n = Θ (n^2) .

O (Big O) Notation

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

$$O(g(n)) = \begin{cases} f(n) : \text{there exist constants } c > 0, \text{ and } n_0 \text{ õ } N \\ \qquad \text{such that } 0 \text{ d" } f(n) \text{ d" } c\, g(n) \text{ for all } n \text{ e" } n_0 \end{cases}$$

In other words a function f(n) is said to belongs to O(g(n)), if there exists positive constant c such that 0 d" f(n) d" c g(n) for sufficiently large value of n. Fig 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 an **asymptotically upper bound** for f(n).

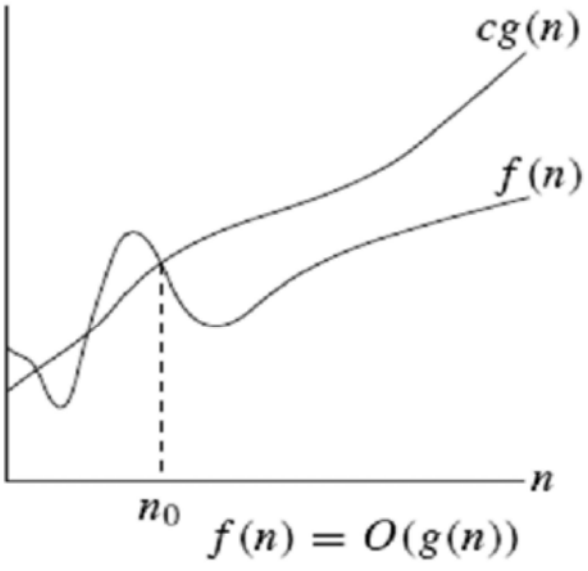


Fig 2.2 : Graphic Example of O notation.

&! (Big Omega) Notation

For a given function g(n), &! (g(n)) is defined as

$$\&!(g(n)) = \begin{cases} f(n) : \text{there exist constants } c > 0, \text{ and } n_0 \text{ õ } N \\ \qquad \text{such that } 0 \text{ d" } c\, g(n) \text{ d" } f(n) \text{ for all } n \text{ e" } n_0 \end{cases}$$

In other words, a function f(n) is said to belongs to &! (g(n)), if there exists positive constant c such that 0 d" c g(n) d" f(n) for sufficiently large value of n. Fig 2.3 gives a intuitive picture of functions f(n) and g(n), where f(n) = &! (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 g(n). So g(n) is said an **asymptotically lower bound** for f(n).

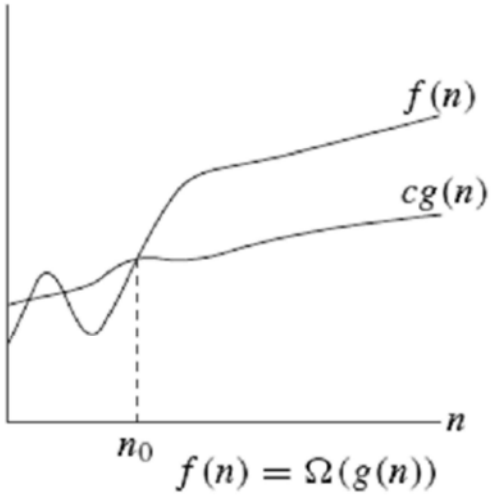


Fig 2.3 : Graphic Example of notation

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) … <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(n log(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 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 above asymptotic notations

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

   $3n^3 + 2n^2 + 4n + 3$ d" $c\,n^3$ for all n e" n_0

   dividing the whole equation by n^3, we get

   $3 + 2/n + 4/n^2 + 3/n^3$ d" c

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

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

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

2. **Prove that $3n^3 + 2n^2 + 4n + 3 = \&!(n^3)$**

   Here,

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

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

   to proof f(n) = &! (g(n)) we must determine the positive constants c and n_0 such that

   $c\,n^3$ d" $3n^3 + 2n^2 + 4n + 3$ for all n e" n_0

   dividing the whole equation by n^3, we get

   c d" $3 + 2/n + 4/n^2 + 3/n^3$

We can make the inequality hold for any value of n e" 1 by choosing c d" 3. Thus, by choosing c = 3 and n_0 = 1 we can have f(n) = &! (g(n)).

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

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

$c_1 n^3$ d" $7n^3 + 7$ d" $c_2 n^3$ for all n e" n_0

dividing the whole equation by n^3, we get

c_1 d" $7 + 7/n^3$ d" c_2

We can make the right hand inequality hold for any value of n e" 1 by choosing c_2 e" 14. Similarly we can make the left hand inequality hold for any value of n e" 1 by choosing c_1 d" 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 1. 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

So,

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 d" f(n) d" c g(n)

=> 0 d" 3n + 4 d" c n

Now if we take n = 1 and c = 7

=> 0 d" 3 x 1 + 4 d" 7 x 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. Consider the following algorithm to add two square matrix.

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 &! notation then we have to show that for some positive constant c and n_0

0 d" c g(n) d" f(n)

=> 0 d" c n^2 d" $3n^2 + 3n + 4$

Now if we take n = 1 and c = 3

=> 0 d" 3 x 1 d" 3 x 1^2 + 3 x 1+ 4

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

f(n) = &! (g(n)) that is

$3n^2 + 3n + 4 = O(n^2)$

In 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.

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 least best case execution 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

Example 2. Consider the following Insertion sort algorithm

```
Algorithm Insertion_Sort (a[n])
    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 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. 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	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 d" f(n) d" c g(n)

=> 0 d" $3n^2 + 4n - 4$ d" c n^2

Now if we take n = 1 and c = 7

=> 0 d" $3x1^2 + 4x1 - 4$ d" 7 x 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 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. 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
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. Sate True or False.

a) $7\ n^3 + 4n + 27 = O\ (n^3)$

b) $2n^2 + 34 = \&!\ (n^3)$

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

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

e) $2n^2 + 34 = \&!\ (n)$

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

g) $2n^7 + 4n^3 + 2n = \&!\ (n^3)$

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

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 by space complexity and time complexity of the program
- The Space complexity of an algorithm is the amount of main memory is 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 READINGS

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, Second Edition, Prentice Hall of India Pvt. Ltd, 2006.
- Ellis Horowitz, Sartaj Sahni and Sanguthevar Rajasekaran, *Fundamental of data structure in C*, Second Edition, Universities Press, 2009.
- Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman, *The Design and Analysis of Computer Algorithms*, Pearson Education, 1999.
- Ellis Horowitz, Sartaj Sahni and Sanguthevar Rajasekaran, *Computer Algorithms/ C++*, Second Edition, Universities Press, 2007.

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

2.9 MODEL QUESTIONS

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

Q.2. Write a 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) $^1/_3 n^4 - 7n^2 + 3n = \Theta (n^4)$

iii) $2n^2 + n + 4 = \Theta (n^2)$

iv) $3n^5 - 7n + 4 = O (n^5)$

v) $3n^5 - 7n + 4 = \&! (n^5)$

UNIT 3 : LINKED LIST

UNIT STRUCTURE

3.1 LEARNING OBJECTIVES

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

- learn about the linked list
- describe different types of linked lists

- implement different operations on singly, doubly and circular linked list
- learn about advantages and disadvantages of linked list over array

3.2 INTRODUCTION

We have already learned about array and its limitations. In this unit, we will learn about the linear and dynamic data structure called linked list .There are three types of linked list available which are singly linked list, doubly linked list and circular linked list. The operations on these linked lists will be discussed in the following sections. The differences between linked list and array will also be discussed in the following sections.

3.3 INTRODUCTION TO LINKED LIST

Linked list is a linear dynamic data structure. It is a collection of some nodes containing homogeneous elements. Each node consists of a data part and one or more address part depending upon the types of the linked list. There three different types of linked list available which are singly linked list, doubly linked list and circular linked list.

3.4 SINGLY LINKED LIST

Singly linked list is a linked list which is a linear list of some nodes containing homogeneous elements .Each node in a singly linked list consists of two parts, one is data part and another is address part. The data part contains the data or information and except the last node, the address part contains the address of the next node in the list. The address part of the last node in the list contains NULL. Here one pointer is used to point the first node in the list. Now in the following sections, we are going to discussed three basic operations on singly linked list which are insertion of a new node, deletion of a node and traversing the linked list.

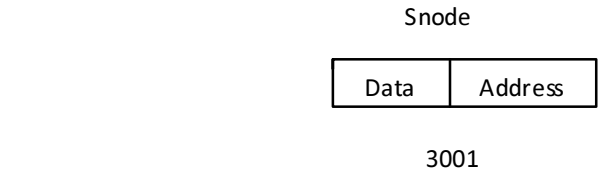


Fig. 3.1(a) : Node of singly linked list

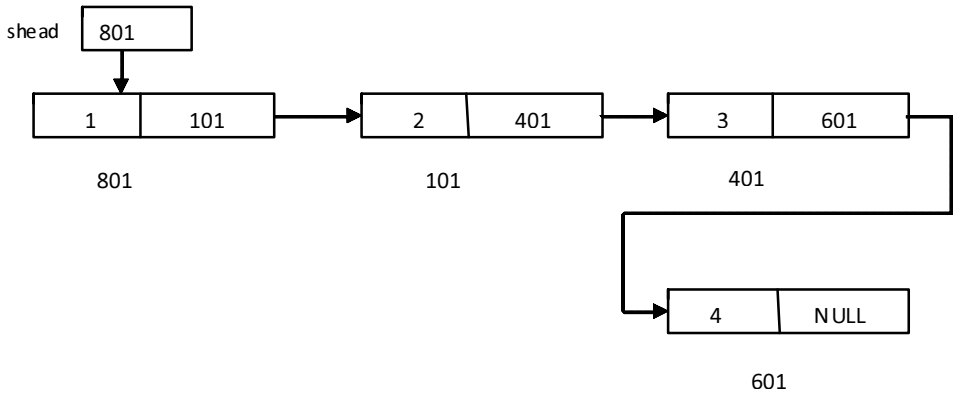


Fig. 3.1(b) : Example of a singly linked list

The structure of a node of singly linked list is shown diagrammatically in fig. 3.1(a). Here "3001" is the memory address of the node and "snode" is the name of the memory location. A diagrammatic representation of a singly linked list is given in fig. 3.1(b). Here "shead" is the pointer which points the first node of the linked list. So here "shead" contains "801" that is address of the first node. The address part of the last node whose memory address is 601 contains NULL.

3.4.1 Insertion of a New Node into a Singly Linked List

Here we will discuss insertion of a new node into a singly linked list at the first position, at the last position and at the position which is inputed by the user. In the following algorithms two parameters are used. "shead" is used to point the first node and element is used to store the data of the new node to be inserted in to the singly linked list.

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

DATA(snode) means data part of the node pointed by the pointer “snode”.

“newnode” is the pointer which points the node to be inserted into the linked list.

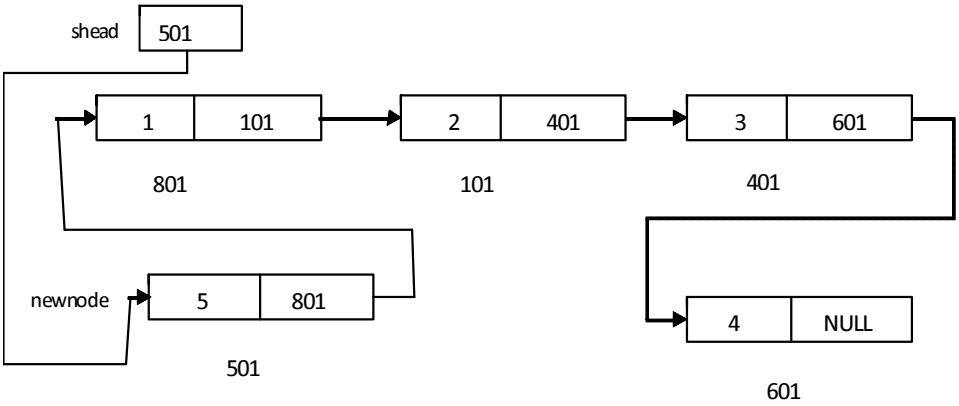


Fig. 3.2 : Example for insertion of a new node into the first position in a singly linked list

Algorithm for inserting new node at the first position into a singly linked list:

```
insert_first(shead,element)
Step 1.   ALLOCATE MEMORY FOR newnode
Step 2.   ADDRESS(newnode) = NULL
Step 3.   DATA(newnode) = element
Step 4.   IF shead == NULL
Step 5.       shead = newnode
Step 6.   END OF IF
Step 7.   ELSE
Step 8.       ADDRESS(newnode) = shead
Step 9.       shead = newnode
Step 10.  END OF ELSE
```

In fig. 3.2 , a node with memory address “501” is inserted in the first position of a singly linked list.

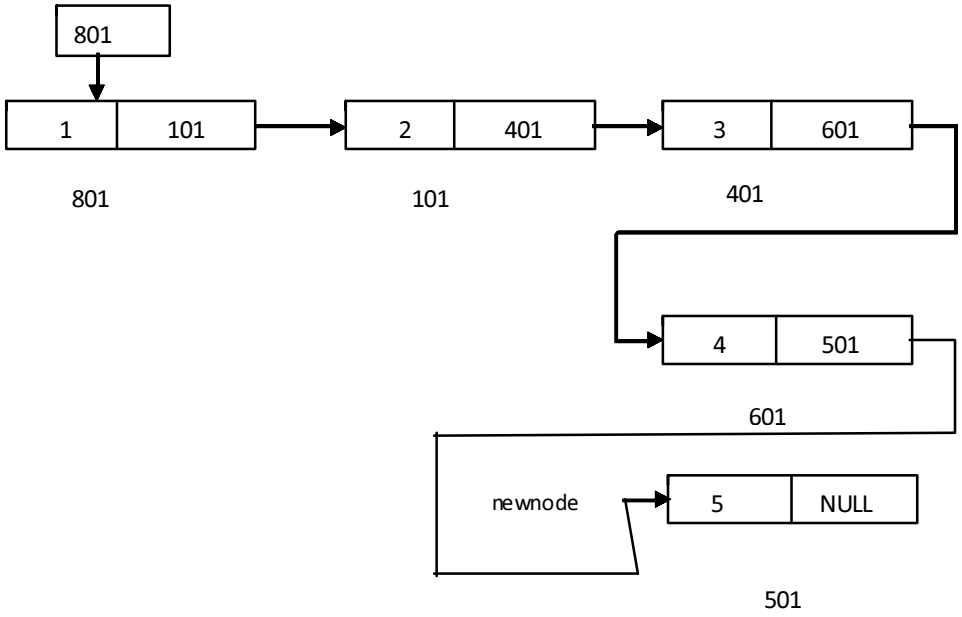


Fig. 3.3 : Example for insertion of a new node into the last position in a singly linked list

Algorithm for inserting new node at the last position into a singly linked list:

insert_last(shead,element)

Step 1. ALLOCATE MEMORY FOR newnode

Step 2. ADDRESS(newnode) = NULL

Step 3. DATA(newnode) = element

Step 4. IF shead == NULL

Step 5. shead = newnode

Step 6. END OF IF

Step 7. ELSE

Step 8. temp = shead

Step 9. WHILE ADDRESS(temp) ! = NULL

Step 10. temp = ADDRESS(temp)

Step 11. END OF WHILE

Step 12. ADDRESS(temp) = newnode

Step 13. END OF ELSE

In fig. 3.3, a node with memory address “501” is inserted in the last position of a singly linked list.

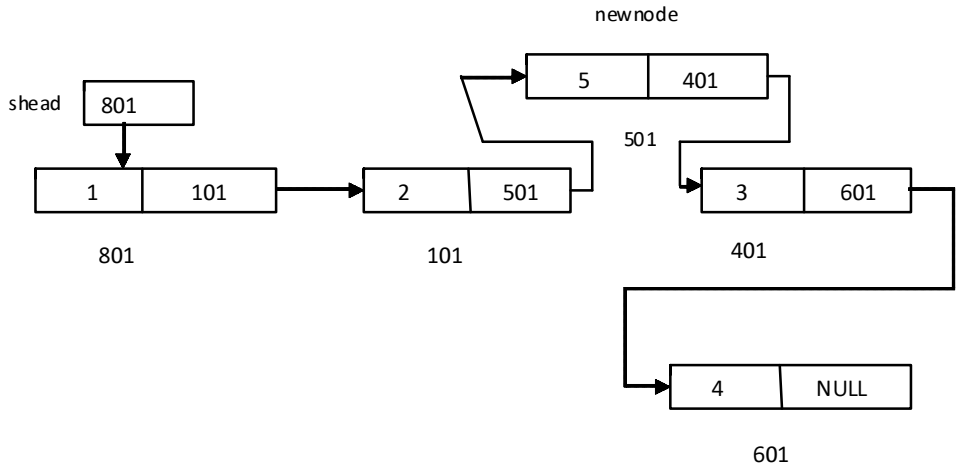


Fig. 3.4 : Example for insertion of a new node into the 3rd position in a singly linked list

Algorithm for inserting new node at a position which is inputed by the user into a singly linked list:

Here "pos" is used to store the position inputted by the user in which the new node is to be inserted in to the singly linked list.

```
insert_at_p(shead,element,pos)
Step 1.   count = 1
Step 2.   ALLOCATE MEMORY FOR  newnode
Step 3.   DATA(newnode) = element
Step 4.   ADDRESS(newnode) = NULL
Step 5.   IF  pos <=0 OR (pos > 1 AND shead=NULL)
Step 6.       PRINT  "Wrong input for position"
Step 7.   END OF IF
Step 8.   ELSE
Step 9.       IF  pos == 1
Step 10.      ADDRESS(newnode) = shead
Step 11.          shead = newnode
Step 12.  END OF IF
Step 13.  temp1 = shead
Step 14.  WHILE  count < pos AND ADDRESS(temp1) != NULL
Step 15.      temp2 = temp1
Step 16.      temp1 = ADDRESS(temp1)
Step 17.      count = count+1
```

Step 18. END OF WHILE

Step 19. IF count == pos

Step 20. ADDRESS(newnode) = temp1

Step 21. ADDRESS(temp2) = newnode

Step 22. END OF IF

Step 23. ELSE IF count == pos-1

Step 24. ADDRESS(temp1) = newnode

Step 25. END OF ELSE IF

Step 26. ELSE

Step 27. PRINT "Wrong input for position"

Step 28. END OF ELSE

Step 29. END OF ELSE

In fig. 3.4, a node with memory address "501" is inserted in the 3rd position of a singly linked list.

3.4.2 Deletion of a Node from a Singly Linked List

Here we will discuss deletion of the first node, the last node and the node whose position is inputed by the user from a singly linked list. In the following algorithms, one parameters are used. "shead" is used to point the first node of a singly linked list.

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

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

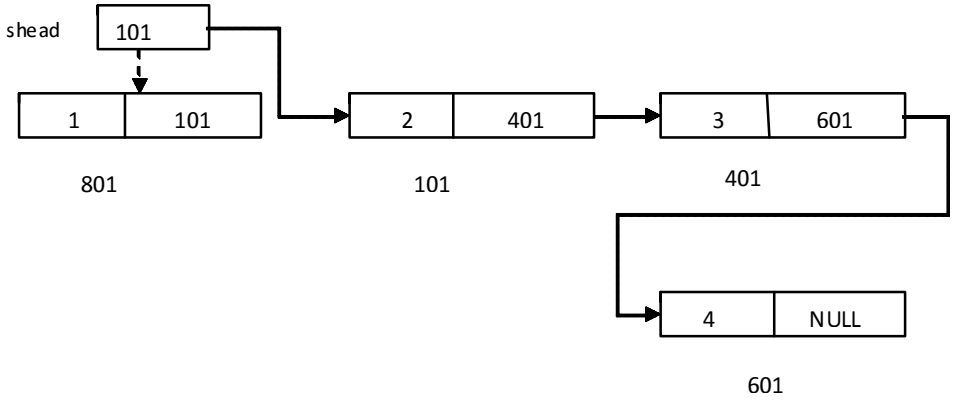


Fig. 35 : Example for deletion of the first node from a singly linked list

Algorithm for deletion of the fist node:

delet_first(shead)

Step 1. IF shead == NULL

Step 2. PRINT "The linked list is empty"

Step 3. END OF IF

Step 4. ELSE

Step 5. temp = shead

Step 6. shead = ADDRESS(shead)

Step 7. DEALLOCATE MEMORY FOR temp

Step 8. END OF ELSE

In fig. 3.5, the first node with memory address "801" is deleted from the singly linked list.

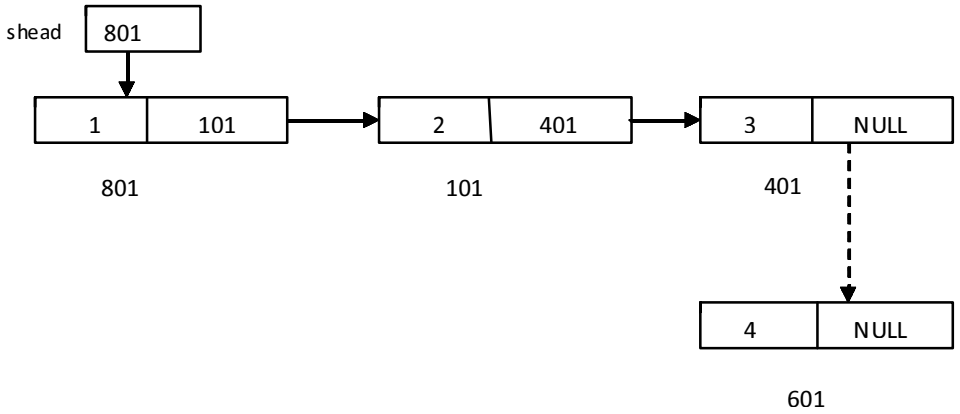


Fig. 3.6 : Example for deletion of the last node from a singly linked list

Algorithm for deletion of the last node:

delet_last(shead)

Step 1. IF shead==NULL

Step 2. PRINT "The linked list is empty"

Step 3. END OF IF

Step 4. ELSE

Step 5. temp1 = shead;

Step 6. WHILE ADDRESS(temp1) != NULL

Step 7. temp2 = temp1

Step 8. temp1 = ADDRESS(temp1)

Step 9. END OF WHILE

Step 10. IF temp1 == shead

Step 11. shead = NULL

Step 12. END OF IF

Step 13. ADDRESS(temp2) = NULL

Step 14. DEALLOCATE MEMORY FOR temp1

Step 15. END OF ELSE

In fig. 3.6, the last node with memory address "601" is deleted from the singly linked list.

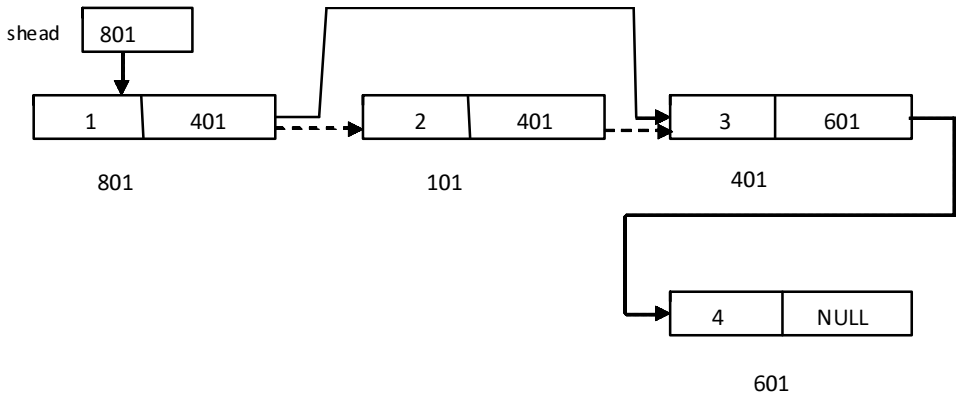


Fig. 3.7 : Example for deletion of the 2nd node from a singly linked list

Algorithm for deletion of the node whose position is inputed by the user:

delet_p(shead,pos)

Step 1. IF shead == NULL

Step 2. PRINT "The linked list is empty "

Step 3. END OF IF

Step 4. ELSE

Step 5. temp1 = shead;

Step 6. IF pos == 1

Step 7. shead = ADDRESS(shead)

Step 8. DEALLOCATE MEMORY FOR temp1

Step 9. END OF IF

Step 10. ELSE

Step 11. WHILE count < pos AND ADDRESS(temp1)! = NULL

Step 12. temp2 = temp1

Step 13. temp1 = ADDRESS(temp1)

Step 14. count = count + 1
Step 15. END OF WHILE
Step 16. IF pos == count
Step 17. ADDRESS(temp2) = ADDRESS(temp1)
Step 18. DEALLOCATE MEMORY FOR temp1
Step 19. END OF IF
Step 20. ELSE
Step 21. PRINT "Wrong input for the position"
Step 22. END OF ELSE
Step 23. END OF ELSE
Step 24. END OF ELSE

In fig. 3.7, the 2nd node with memory address "101" is deleted from the singly linked list.

3.4.2 Traversal of Nodes in Singly Linked List

In a singly linked list the traversal of nodes is done sequentially from the first node to the last node.

Algorithm for traversing nodes in a singly linked list:

traverse_slist(shead)
Step 1. temp = shead
Step 2. IF shead == NULL
Step 3. PRINT "The linked list is empty"
Step 4. END OF IF
Step 5. ELSE
Step 6. WHILE temp != NULL
Step 7. temp = ADDRESS(temp)
Step 8. END OF WHILE
Step 9. END OF ELSE

3.4.3 C Program to Implement Singly Linked List

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

```
// Structure to create a node for singly linked list
    struct snode
    {
        int data;
        struct snode *next;
    };
    typedef struct snode snode;
    //Function prototypes
    void insert(snode **,int );
    int insert_first(snode **,int);
    int insert_last(snode **,int);
    int insert_at_p(snode **,int,int);
    void delet(snode **);
    int delet_first(snode **);
    int delet_last(snode **);
    int delet_p(snode **,int);
    int isempty(snode *);
    void init(snode **);
    void creat(snode **);
    void display(snode *);
    void main()
    {
        snode *shead;
        int option,elem,flag;
        char cont;
        clrscr();
        init(&shead);
        creat(&shead);
        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 linked list::");
                        scanf("%d",&elem);
                        insert(&shead,elem);
                        break;
                    case 2:delet(&shead);
                        break;
                    case 3:display(shead);
                        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 the pointer which points the starting
    node of the singly linked list
        void init(snode **shead)
        {
            *shead = NULL;
        }
    //Function to check underflow condition
        int isempty(snode *shead)
        {
            if(shead == NULL)
                return(1);
```

```
        else
            return(0);
    }
//Function for insertion of a new node into a singly linked list
    void insert(snode **shead,int element)
    {
        int opt,pos;
        printf("\n*******INSERT MENU*********");
        printf("\n1.Insert at first position*");
        printf("\n2.Insert at last position *");
        printf("\n3.Insert at pth position  *");
        printf("\n**************************");
        printf("\nEnter your option::");
        scanf("%d",&opt);
        switch(opt)
        {
            case 1: if(insert_first(shead,element))
                    {
                    printf("\n%d is succesfully inserted at the first
                    position",element);
                    printf("\nAfter insertion the linked list is::\n");
                    display(*shead);
                    }
                    else
                    {
                    printf("\nInsertion isnot successfull");
                    }
                    break;
            case 2: if(insert_last(shead,element))
                    {
                    printf("\n%d is succesfully inserted at the last
                    position",element);
                    printf("\nAfter insertion the linked list is::\n");
```

```
                    display(*shead);
                    }
                    else
                    {
                    printf("\nInsertion isnot successfull");
                    }
                    break;
            case 3: printf("\nEnter the position::");
                    scanf("%d",&pos);
                    if(insert_at_p(shead,element,pos))
                    {
                    printf("\n%d is succesfully inserted at %d
                    position",element,pos);
                    printf("\nAfter insertion the linked list is::\n");
                    display(*shead);
                   }
                   else
                   {
                   printf("\nInsertion isnot successfull");
                   }
                   break;
            default:printf("\nWrong input..Try agail");
        }
    }
//Function for deletion of a node from a singly linked list
    void delet(snode **shead)
    {
        int opt,pos,elem;
        printf("\n*******DELETE MENU*********");
        printf("\n1.Delete the first node   *");
        printf("\n2.Delete the last node    *");
        printf("\n3.Delete the pth node     *");
        printf("\n***************************");
```

```
printf("\nEnter your option::");
scanf("%d",&opt);
switch(opt)
{
    case 1: elem = delet_first(shead);
            if(elem == -99)
            {
            printf("\nDeletion isnot possible as the linked
            list is empty");
            }
            else
            {
            printf("\n%d is succesfully deleted",elem);
            printf("\nAfter deletion the linked list is::\n");
            display(*shead);
            }
            break;
    case 2: elem = delet_last(shead);
            if(elem == -99)
            {
            printf("\nDeletion isnot possible as the linked
            list is empty");
            }
            else
            {
            printf("\n%d is succesfully deleted",elem);
            printf("\nAfter deletion the linked list is::\n");
            display(*shead);
            }
            break;
    case 3: printf("\nEnter the position::");
            scanf("%d",&pos);
            elem = delet_p(shead,pos);
```

```
                    if(elem == -99)
                    {
                    printf("\nDeletion isnot possible as the linked
                    list is empty");
                    }
                    else if(elem == -98)
                    printf("\nWrong input for position");
                    else
                    {
                    printf("\n%d is succesfully deleted",elem);
                    printf("\nAfter deletion the linked list is::\n");
                    display(*shead);
                    }
                    break;
            default:printf("\nWrong input..Try agail");
        }
    }
// Function to display the elements in s singly linked list
    void display(snode *shead)
    {
        snode *temp;
        temp = shead;
        if(isempty(shead))
            printf("\nThe linked list is empty");
        else
        {
        printf("\nthe elements in the singly linked list are:\n");
        76mn while(temp != NULL)
        {
        printf("%5d",temp->data);
        temp = temp->next;
        }
    }
```

```
    }
//Function to create a singly linked list
    void creat(snode **shead)
    {
        snode *newnode,*last;
        char cont;
        do
        {
        newnode = (snode *)malloc(sizeof(snode));
        newnode->next = NULL;
        printf("\nEnter data::");
        scanf("%d",&newnode->data);
        if(*shead == NULL)
        {
            *shead = newnode;
            last = newnode;
        }
        else
        {
            last->next = newnode;
            last = newnode;
        }
            printf("\nDo you want to add more node..press 'y' or
            'Y' to continue::");
            cont = getch();
        }while(cont == 'y' || cont == 'Y');
    }
// Function to insert a new node into the first position
    int insert_first(snode **shead,int element)
    {
    snode *newnode;
    newnode = (snode *)malloc(sizeof(snode));
    newnode->next = NULL;
```

```
        newnode->data = element;
        if(*shead == NULL)
        {
            *shead = newnode;
            return(1);
        }
        else
        {
            newnode->next = *shead;
            *shead = newnode;
            return(1);
        }
    }
    //Function to insert a new node at the last position
        int insert_last(snode **shead,int element)
        {
        snode *temp,*newnode;
        newnode = (snode *)malloc(sizeof(snode));
        newnode->next = NULL;
        newnode->data = element;
        if(*shead == NULL)
        {
            *shead = newnode;
            return(1);
        }
        else
        {
            temp = *shead;
            while(temp->next != NULL)
            temp = temp->next;
            temp->next = newnode;
            return(1);
        }
```

```
    }
//Function to insert a node at pth position
    int insert_at_p(snode **shead,int element,int pos)
        {
            snode *temp1,*temp2,*newnode;
            int count = 1;
            newnode = (snode *)malloc(sizeof(snode));
            newnode->data = element;
            newnode->next = NULL;
            if(pos <= 0 || (pos > 1 && *shead == NULL))
            {
            printf("\nWrong input for position");
            return(0);
            }
            else
            {
            if(pos == 1)
            {
            newnode->next = *shead;
            *shead = newnode;
            return(1);
            }
            temp1 = *shead;
            while(count < pos && temp1->next != NULL)
            {
            temp2 = temp1;
            temp1 = temp1->next;
            count++;
            }
            if(count == pos)
            {
            newnode->next = temp1;
            temp2->next = newnode;
```

```
                return(1);
                }
                else if(count == pos-1)
                {
                temp1->next = newnode;
                return(1);
                }
                else
                {
                printf("\nWrong input for position");
                return(0);
                }
            }
        }
    //Function to delete the first node
        int delet_first(snode **shead)
        {
            snode *temp;
            int delem;
            if(*shead == NULL)
            return(-99);
            else
            {
            temp = *shead;
            delem = temp->data;
            *shead = (*shead)->next;
            free(temp);
            return(delem);
            }
        }
    //Function to delete the last node
        int delet_last(snode **shead)
        {
```

```
        snode *temp1,*temp2;
        int delem;
        if(*shead == NULL)
        return(-99);
        else
        {
        temp1 = *shead;
        while(temp1->next != NULL)
        {
        temp2 = temp1;
        temp1 = temp1->next;
        }
        delem = temp1->data;
        if(temp1 == *shead)
        *shead = NULL;
        else
        temp2->next = NULL;
        free(temp1);
        return(delem);
        }
    }
//Function to delete the pth node
    int delet_p(snode **shead,int pos)
    {
        snode *temp1,*temp2;
        int delem,count = 1;
        if(*shead == NULL)
        return(-99);
        else
        {
        temp1 = *shead;
        if(pos == 1)
        {
```

```
            delem = (*shead)->data;
            *shead = (*shead)->next;
            free(temp1);
            return(delem);
            }
            while(count < pos && temp1->next != NULL)
            {
            temp2 = temp1;
            temp1 = temp1->next;
            count++;
            }
            if(pos == count)
            {
            delem = temp1->data;
            temp2->next = temp1->next;
            free(temp1);
            return(delem);
            }
            else
            {
            return(-98);
            }
        }
    }
```

3.5 DOUBLY LINKED LIST

Doubly linked list is a linked list which is a linear list of some nodes containing homogeneous elements .Each node in a doubly linked list consists of three parts, one is data part and other two are address parts. The data part contains the data or information. Except the first and the last node, one address part contains the address of the next node in the list and other address part contains the address of the previous node in the list. In

case of the first node, one address part contains the address of the next node and other address part contains NULL. On the other hand in case of the last node, one address part contains the address of the previous node and the other address part contains NULL. Here one pointer is used to point the first node in the list.

Doubly linked lists require more memory space than the singly linked list because of the one extra address part. But in doubly linked list the traversal can be done in both direction .So here movement of any node to any node is possible.

Now in the following sections, we are going to discussed three basic operations on doubly linked list which are insertion of a new node, deletion of a node and traversing the linked list.

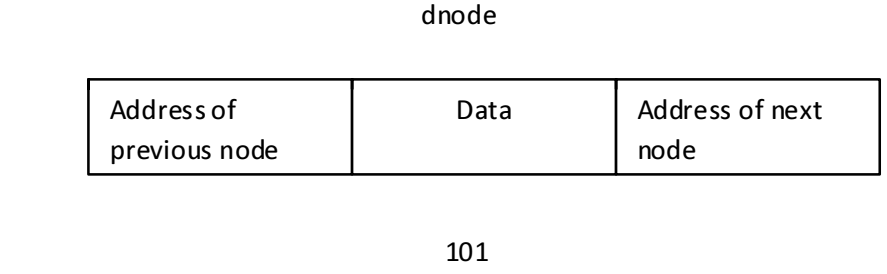


Fig. 3.8(a) : Node of Doubly linked list

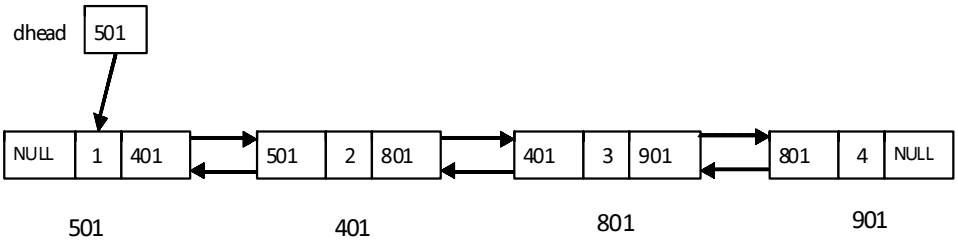


Fig. 3.8(b) : Example of a doubly linked list

The structure of a node of doubly linked list is shown diagrammatically in fig 8(a). Here "101" is the memory address of the node and "dnode" is the name of the memory location. A diagrammatic representation of a doubly linked list is given in fig 8(b). Here "dhead" is the pointer which points the first node of the linked list. So here "dhead" contains "501" that is address of the first node. The address part of the last node whose memory address is 901 and the address part of the first node whose memory address is "501" contain NULL.

3.5.1 Insertion of a New Node into a Doubly Linked List

Here we will discuss insertion of a new node into a doubly linked list at the first position, at the last position and at the position which is inputed by the user. In the following algorithms two parameters are used. “dhead” is used to point the first node and element is used to store the data of the new node to be inserted in to the doubly linked list.

ADDRESSNEXT(dnode) means the address part of a node pointed by the pointer “dnode” which points the next node in the doubly linked list.

ADDRESSPREVIOUS(dnode) means the address part of a node pointed by the pointer “dnode” which points the previous node in the doubly linked list.

DATA(dnode) means the data part of a node pointed by the pointer “dnode” of a doubly linked list.

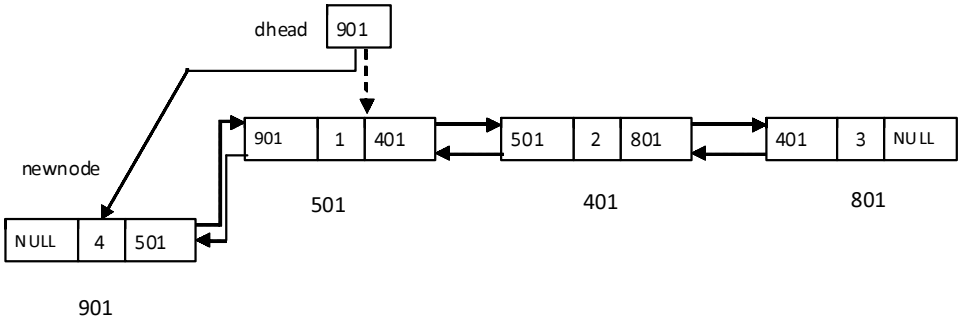


Fig. 3.9 : Example for insertion of a new node into the first position in a doubly linked list

Algorithm for inserting new node at the first position into a doubly linked list:

insert_first(dhead, element)

Step 1. ALLOCATE MEMORY FOR newnode

Step 2. ADDRESSNEXT(newnode) = NULL

Step 3. ADDRESSPREVIOUS(newnode) = NULL

Step 4. DATA(newnode) = element

Step 5. IF dhead == NULL

Step 6. dhead = newnode

Step 7. END OF IF

Step 8. ELSE

Step 9. ADDRESSNEXT(newnode) = dhead

Step 10. ADDRESSPREVIOUS(dhead) = newnode

Step 11. dhead = newnode

Step 12. END OF ELSE

In fig. 3.9, a node with memory address “901” is inserted into the first position of a doubly linked list.

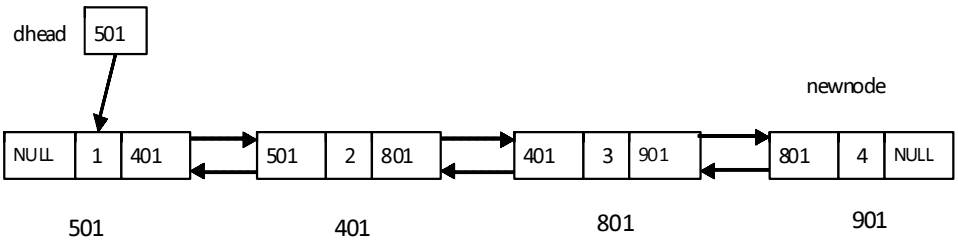


Fig. 3.10 : Example for insertion of a new node into the last position in a doubly linked list

Algorithm for inserting new node at the last position into a doubly linked list:

insert_last(dhead, element)

Step 1. ALLOCATE MEMORY FOR newnode

Step 2. ADDRESSPREVIOUS(newnode) = NULL

Step 3. ADDRESSNEXT(newnode) = NULL

Step 4. DATA(newnode) = element

Step 5. IF dhead == NULL

Step 6. dhead = newnode

Step 7. END OF IF

Step 8. ELSE

Step 9. temp = dhead

Step 10. WHILE ADDRESSNEXT(temp) != NULL

Step 11. temp = ADDRESSNEXT(temp)

Step 12. ADDRESSPERVIOUS(newnode) = temp

Step 13. ADDRESSNEXT(temp) = newnode

Step 14. END OF WHILE

Step 15. END OF ELSE

In fig. 3.10, a node with memory address "901" is inserted into the last position of a doubly linked list.

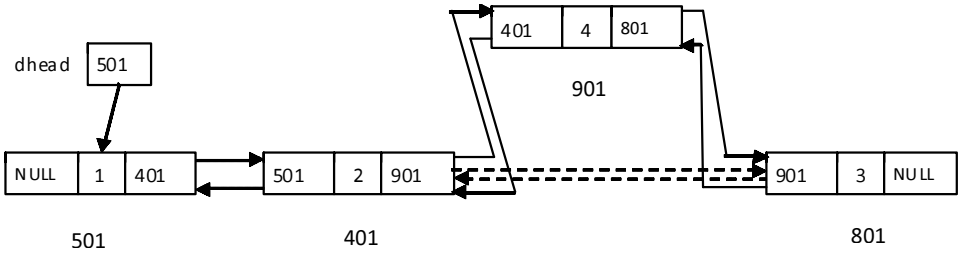


Fig. 3.11 : Example for insertion of a new node into the 3rd position in a doubly linked list

Algorithm for inserting new node at a position which is inputed by the user into a doubly linked list:

insert_at_p(dhead,element,pos)

Step 1. count = 1

Step 2. ALLOCATE MEMORY FOR newnode

Step 3. DATA(newnode) = element

Step 4. ADDRESSNEXT(newnode) = NULL

Step 5: ADDRESSPREVIOUS(newnode) = NULL

Step 6. IF pos<=0 OR (pos >1 AND dhead=NULL)

Step 7. PRINT "Wrong input for position"

Step 8. END OF IF

Step 9. ELSE

Step 10. IF pos == 1

Step 11. ADDRESSNEXT(newnode) = dhead

Step 12. ADDRESSPREVIOUS(dhead) = newnode

Step 13. dhead = newnode

Step 14. END OF IF

Step 15. temp1 = dhead

Step 16. WHILE count<pos AND ADDRESSNEXT(temp1) = NULL

Step 17. temp2 = temp1

Step 18. temp1 = ADDRESSNEXT(temp1)

Step 19. count = count + 1

Step 20. END OF WHILE

Step 21. IF count == pos

Step 22. ADDRESSNEXT(newnode) = temp1

Step 23. ADDRESSPREVIOUS(newnode) = temp2

Step 24. ADDRESSPREVIOUS(temp1) = newnode

Step 25. ADDRESSNEXT(temp2) = newnode

Step 26. END OF IF

Step 27. ELSE IF count == pos-1

Step 28. ADDRESSNEXT(temp1) = newnode

Step 29. ADDRESSPREVIOUS(newnode) = temp1

Step 30. END OF ELSE IF

Step 31. ELSE

Step 32. PRINT "Wrong input for position"

Step 33. END OF ELSE

Step 34. END OF ELSE

In fig. 3.11, a node with memory address "901" is inserted into the 3rd position of a doubly linked list.

3.5.2 Deletion of a Node from a Doubly Linked List

Here we will discuss deletion of the first node, the last node and the node whose position is inputed by the user from a doubly linked list.

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

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

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

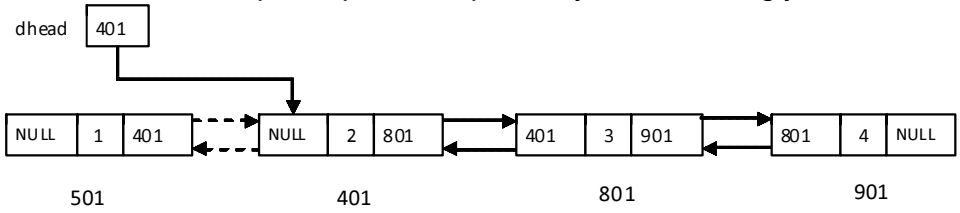


Fig. 3.12 : Example for deletion of the first node from a doubly linked list

Algorithm for deletion of the fist node:

```
delet_first(dhead)
Step 1.   IF dhead == NULL
Step 2.       PRINT " The linked list is empty"
Step 3.   END OF IF
Step 4.   ELSE
Step 5.       temp = dhead
Step 6.       dhead = ADDRESSNEXT(dhead)
Step 7.       IF  dhead != NULL
Step 8.           ADDRESSPREVIOUS(dhead) = NULL
Step 9.       END OF IF
Step 10.      DEALLOCATE MEMORY FOR temp
Step 11.  END OF ELSE
```

In fig. 3.12, the first node with memory address "501" is deleted from the doubly linked list.

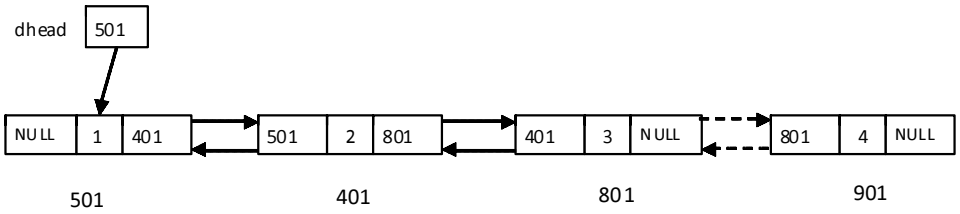


Fig. 3.13 : Example for deletion of the last node from a doubly linked list

Algorithm for deletion of the last node:

```
delet_last(dhead)
Step 1.   IF  dhead == NULL
Step 2.       PRINT " The linked is empty"
Step 3.   END OF IF
Step 4.   ELSE
Step 5.       temp = dhead
Step 6.       WHILE  ADDRESSNEXT(temp) != NULL
Step 7.           temp = ADDRESSNEXT(temp)
Step 8.       END OF WHILE
Step 9.       IF temp == dhead
Step 10.          dhead = NULL
Step 11.      END OF IF
```

Step 12. ELSE

Step 13. ADDRESSNEXT(ADDRESSPREVIOUS(temp)) = NULL

Step 14. END OF ELSE

Step 15. DEALLOCATE MEMORY FOR temp

Step 16. END OF ELSE

In fig. 3.13, the last node with memory address "901" is deleted from the linked list.

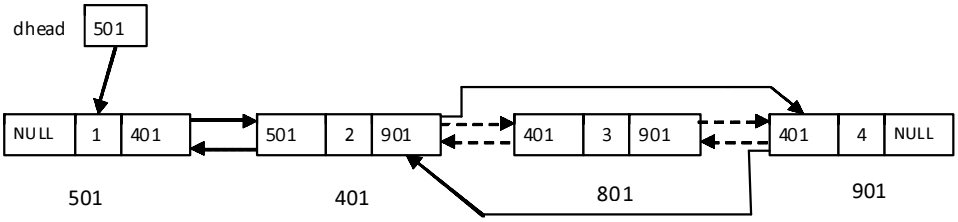


Fig. 3.14 : Example for deletion of the 3rd node from a doubly linked list

Algorithm for deletion of the node whose position is inputed by the user:

delet_p(dhead,pos)

Step 1. count = 1

Step 2. IF dhead == NULL

Step 3. PRINT " The linked list is empty"

Step 4. END OF IF

Step 5. ELSE

Step 6. temp = dhead

Step 7. IF pos == 1

Step 8. dhead = ADDRESSNEXT(dhead)

Step 9. IF dhead != NULL

Step 10. ADDRESSPREVIOUS(dhead) = NULL

Step 11. END OF IF

Step 12. DEALLOCATE MEMORY FOR temp

Step 13. END OF IF

Step 14. WHILE count < pos AND ADDRESSNEXT(temp)! = NULL

Step 15. temp = ADDRESSNEXT(temp)

Step 16. count = count+1

Step 17. END OF WHILE
Step 18. IF pos == count
Step 19. ADDRESSNEXT(ADDRESSPREVIOUS(temp))
= ADDRESSNEXT(temp)
Step 20. ADDRESSPREVIOUS(ADDRESSNEXT(temp))
= ADDRESSPREVIOUS(temp)
Step 21. DEALLOCATE MEMORY FOR temp
Step 22. END OF IF
Step 23. ELSE
Step 24. PRINT " Wrong input for position"
Step 25. END OF ELSE
Step 26. END OF ELSE

In fig. 3.14, the 3rd node is deleted from the doubly linked list.

3.5.3 Traversal of Nodes in Doubly Linked List

In a doubly linked list the traversal of nodes can be done sequentially in both directions, from the first node to the last node and from the last node to first node.

Algorithm for traversing nodes in a doubly linked list:

traverse(dhead)
Step 1. temp = dhead
Step 2. IF dhead == NULL
Step 3. PRINT "The linked list is empty"
Step 4. ELSE
Step 5. /* Traversal in the forward direction */
Step 6. WHILE temp != NULL
Step 7. temp = ADDRESSNEXT(temp)
Step 8. END OF WHILE
Step 9. /* Traversal in the backward direction */
Step 10. WHILE temp !=NULL
Step 11. temp = ADDRESSPREVIOUS(temp)
Step 12. END OF WHILE
Step 13. END OF ELSE

3.5.4 C Program to Implement Doubly Linked List

```
#include<stdio.h>
#include<conio.h>
#define max 40
//STRUCTURE TO CREATE NODE OF DOUBLY LINKED LIST
struct dnode
{
    int data;
    struct dnode *next;
    struct dnode *prev;
};
typedef struct dnode dnode;
//FUNCTION PROTOTYPES
void insert(dnode **,int );
int insert_first(dnode **,int);
int insert_last(dnode **,int);
int insert_at_p(dnode **,int,int);
void delet(dnode **);
int delet_first(dnode **);
int delet_last(dnode **);
int delet_p(dnode **,int);
int isempty(dnode *);
void init(dnode **);
void creat(dnode **);
void display(dnode *);
void main()
{
    dnode *dhead;
    int option,elem,flag;
    char cont;
    clrscr();
    init(&dhead);
    creat(&dhead);
```

```
        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 linked list::");
                     scanf("%d",&elem);
                     insert(&dhead,elem);
                     break;
            case 2:  delet(&dhead);
                     break;
            case 3:  display(dhead);
                     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 THE POINTER WHICH POINTS
      THE FIRST NODE OF THE DOUBLY LINKED LIST
        void init(dnode **dhead)
        {
            *dhead = NULL;
        }
```

```
//FUNCTION TO CHECK THE UNDERFLOW CONDITION
    int isempty(dnode *dhead)
    {
        if(dhead == NULL)
        return(1);
        else
        return(0);
    }
//FUNCTION TO INSERT NEW NODE INTO A DOUBLY LINKED
  LIST
    void insert(dnode **dhead,int element)
    {
        int opt,pos;
        printf("\n*******INSERT MENU*********");
        printf("\n1.Insert at first position*");
        printf("\n2.Insert at last position *");
        printf("\n3.Insert at pth position  *");
        printf("\n**************************");
        printf("\nEnter your option::");
        scanf("%d",&opt);
        switch(opt)
        {
            case 1:  if(insert_first(dhead,element))
                     {
                     printf("\n%d is succesfully inserted at the
                     first position",element);
                     printf("\nAfter insertion the linked list is::\n");
                     display(*dhead);
                     }
                     else
                     {
                     printf("\nInsertion isnot successfull");
                     }
```

```
                    break;
            case 2: if(insert_last(dhead,element))
                    {
                    printf("\n%d is succesfully inserted at the
                    last position",element);
                    printf("\nAfter insertion the linked list is::\n");
                    display(*dhead);
                    }
                    else
                    {
                    printf("\nInsertion isnot successfull");
                    }
                    break;
            case 3: printf("\nEnter the position::");
                    scanf("%d",&pos);
                    if(insert_at_p(dhead,element,pos))
                    {
                    printf("\n%d is succesfully inserted at %d
                    position",element,pos);
                    printf("\nAfter insertion the linked list is::\n");
                    display(*dhead);
                    }
                    else
                    {
                    printf("\nInsertion isnot successfull");
                    }
                    break;
            default:printf("\nWrong input..Try agail");
        }
}
//FUNCTION TO DELETE A NODE FROM THE DOUBLY
  LINKED LIST
    void delet(dnode **dhead)
```

```
{
    int opt,pos,elem;
    printf("\n*******DELETE MENU*********");
    printf("\n1.Delete the first node   *");
    printf("\n2.Delete the last node    *");
    printf("\n3.Delete the pth node     *");
    printf("\n***************************");
    printf("\nEnter your option::");
    scanf("%d",&opt);
    switch(opt)
    {
        case 1:  elem = delet_first(dhead);
                 if(elem == -99)
                 {
                 printf("\nDeletion isnot possible as the linked
                 list is empty");
                 }
                 else
                 {
                 printf("\n%d is succesfully deleted",elem);
                 printf("\nAfter deletion the linked list is::\n");
                 display(*dhead);
                 }
                 break;
        case 2:  elem = delet_last(dhead);
                 if(elem == -99)
                 {
                 printf("\nDeletion isnot possible as the linked
                 list is empty");
                 }
                 else
                 {
                 printf("\n%d is succesfully deleted",elem);
```

```
                printf("\nAfter deletion the linked list is::\n");
                display(*dhead);
                }
                break;
        case 3: printf("\nEnter the position::");
                scanf("%d",&pos);
                elem = delet_p(dhead,pos);
                if(elem == -99)
                {
                printf("\nDeletion isnot possible as the linked
                list is empty");
                }
                else if(elem == -98)
                printf("\nWrong input for position");
                else
                {
                printf("\n%d is succesfully deleted",elem);
                printf("\nAfter deletion the linked list is::\n");
                display(*dhead);
                }
                break;
        default:printf("\nWrong input..Try agail");
      }
   }
//FUNCTION TO DISPLAY THE ELEMENTS IN A DOUBLY
  LINKED LIST
   void display(dnode *dhead)
   {
      dnode *temp;
      temp = dhead;
      if(isempty(dhead))
      printf("\nThe linked list is empty");
      else
```

```
        {
        printf("\nthe elements in the doubly linked list are:\n");
        while(temp != NULL)
        {
        printf("%5d",temp->data);
        temp = temp->next;
        }
    }
}
//FUNCTION TO CREATE A DOUBLY LINKED LIST
    void creat(dnode **dhead)
    {
        dnode *newnode,*last;
        char cont;
        do
        {
            newnode = (dnode *)malloc(sizeof(dnode));
            newnode->next = NULL;
            newnode->prev = NULL;
            printf("\nEnter data::");
            scanf("%d",&newnode->data);
            if(*dhead == NULL)
            {
                *dhead = newnode;
                last = newnode;
            }
        else
        {
            newnode->prev = last;
            last->next = newnode;
            last = newnode;
        }
        printf("\nDo you want to add more node..press 'y' or 'Y'
```

```
            to continue::");
            cont = getch();
        }while(cont == 'y' || cont == 'Y');
    }
    //FUNCTION TO INSERT A NEW NODE AT THE FIRST
     POSITION
        int insert_first(dnode **dhead,int element)
        {
            dnode *newnode;
            newnode = (dnode *)malloc(sizeof(dnode));
            newnode->next = NULL;
            newnode->prev = NULL;
            newnode->data = element;
            if(*dhead == NULL)
            {
                *dhead = newnode;
                return(1);
            }
            else
            {
                newnode->next = *dhead;
                (*dhead)->prev = newnode;
                *dhead = newnode;
                return(1);
            }
    }
    //FUNCTION TO INSERT A NEW NODE AT THE LAST
     POSITION
        int insert_last(dnode **dhead,int element)
        {
            dnode *temp,*newnode;
            newnode = (dnode *)malloc(sizeof(dnode));
            newnode->prev = NULL;
```

```
        newnode->next = NULL;
        newnode->data = element;
        if(*dhead == NULL)
        {
            *dhead = newnode;
            return(1);
        }
        else
        {
            temp = *dhead;
            while(temp->next != NULL)
            temp = temp->next;
            newnode->prev = temp;
            temp->next = newnode;
            return(1);
        }
    }
//FUNCTION TO INSERT A NEW NODE AT PTH POSITION
    int insert_at_p(dnode **dhead,int element,int pos)
    {
        dnode *temp1,*temp2,*newnode;
        int count = 1;
        newnode = (dnode *)malloc(sizeof(dnode));
        newnode->data = element;
        newnode->next = NULL;
        if(pos <=0 || (pos > 1 && *dhead == NULL))
        {
        printf("\nWrong input for position");
        return(0);
        }
        else
        {
        if(pos == 1)
```

```
        {
            newnode->next = *dhead;
            (*dhead)->prev = newnode;
            *dhead = newnode;
            return(1);
        }
        temp1 = *dhead;
        while(count < pos && temp1->next != NULL)
        {
        temp2 = temp1;
        temp1 = temp1->next;
        count++;
        }
        if(count == pos)
        {
            newnode->next = temp1;
            newnode->prev = temp2;
            temp1->prev = newnode;
            temp2->next = newnode;
            return(1);
        }
        else if(count == pos-1)
        {
            temp1->next = newnode;
            newnode->prev = temp1;
            return(1);
        }
        else
        {
            printf("\nWrong input for position");
            return(0);
        }
    }
```

```
//FUNCTION TO DELETE THE FIRST NODE
    int delet_first(dnode **dhead)
    {
        dnode *temp;
        int delem;
        if(*dhead == NULL)
        return(-99);
        else
        {
        temp = *dhead;
        delem = temp->data;
        *dhead = (*dhead)->next;
        if(*dhead != NULL)
        (*dhead)->prev = NULL;
        free(temp);
        return(delem);
        }
    }
//FUNCTION TO DELETE THE LAST NODE
    int delet_last(dnode **dhead)
    {
        dnode *temp;
        int delem;
        if(*dhead == NULL)
        return(-99);
        else
        {
        temp = *dhead;
        while(temp->next != NULL)
        temp = temp->next;
        delem = temp->data;
         if(temp == *dhead)
        *dhead = NULL;
```

```
            else
            (temp->prev)->next = NULL;
            free(temp);
            return(delem);
            }
        }
    //FUNCTION TO DELETE THE PTH NODE
        int delet_p(dnode **dhead,int pos)
        {
            dnode *temp;
            int delem,count = 1;
            if(*dhead == NULL)
            return(-99);
            else
            {
            temp = *dhead;
            if(pos == 1)
            {
            delem = (*dhead)->data;
            *dhead = (*dhead)->next;
            if(*dhead != NULL)
            (*dhead)->prev = NULL;
            free(temp);
            return(delem);
            }
            while(count < pos && temp->next != NULL)
            {
            temp = temp->next;
            count++;
            }
            if(pos == count)
            {
                delem = temp->data;
```

```
                (temp->prev)->next = temp->next;
                (temp->next)->prev = temp->prev;
                free(temp);
                return(delem);
            }
            else
            {
                return(-98);
            }
        }
    }
```

3.6 CIRCULAR LINKED LIST

Circular linked list is a linked list similar with the singly linked list. The only difference between the circular linked list and the singly linked list is the address part of the last node in circular linked list contains the address of the first node in the list. So here no address part of any node contains NULL. Here also one pointer is used to point the first node in the list.

Now in the following sections, we are going to discussed three basic operations on singly linked list which are insertion of a new node, deletion of a node and traversing the linked list.

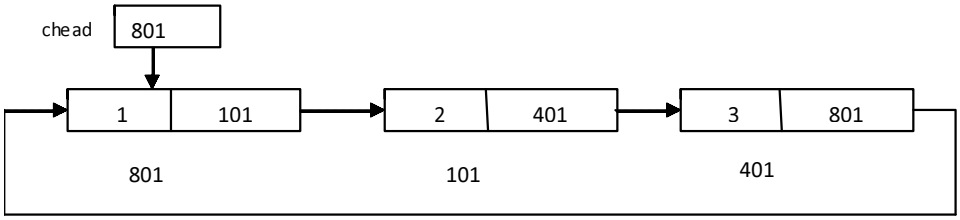


Fig. 3.15 : Example of circular linked list

The structure of a node of circular linked list is same with singly linked list (fig 1(a)). A diagrammatic representation of a circular linked list is given in fig 15. Here “chead” is the pointer which points the first node of the linked list. So here “chead” contains “801” that is address of the first node. The address part of the last node whose memory address is “401” contains the address of the first node that is “801”.

3.6.1 Insertion of a New Node into a Circular Linked List

Here we will discuss insertion of a new node into a circular linked list at the first position, at the last position and at the position which is inputed by the user.

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

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

"newnode" is the pointer which points the node to be inserted into the linked list.

"clast" is a pointer which points the last node of the circular linked list.

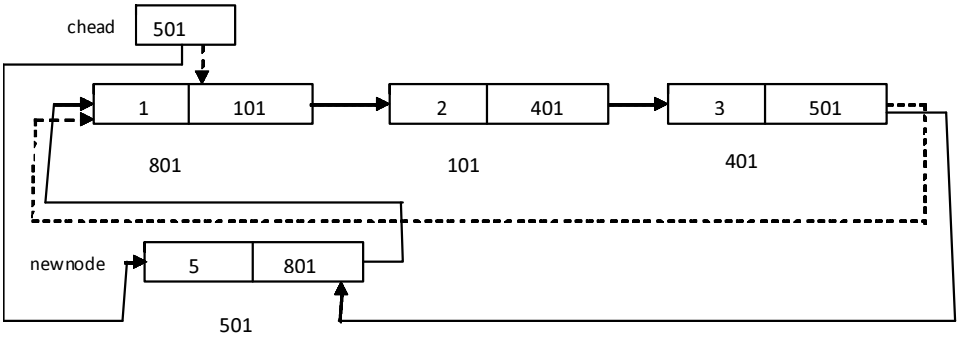


Fig. 3.16 : Example for insertion of a new node into the first position in a cricular linked list

Algorithm for inserting new node at the first position into a circular linked list:

```
insert_first(chead,clast,element)
Step 1.   ALLOCATE MEMORY FOR   newnode
Step 2.   DATA(newnode) = element
Step 3.   IF  chead == NULL
Step 4.        chead = newnode
Step 5.             clast= newnode
Step 6.             ADDRESS(newnode) = chead
Step 7.   END OF IF
```

Step 8. ELSE

Step 9. ADDRESS(newnode) = chead

Step 10. chead = newnode

Step 11. ADDRESS(clast) = chead

Step 12. END OF ELSE

In fig. 3.16, a node with memory address “501” is inserted in to the first position of a circular linked list.

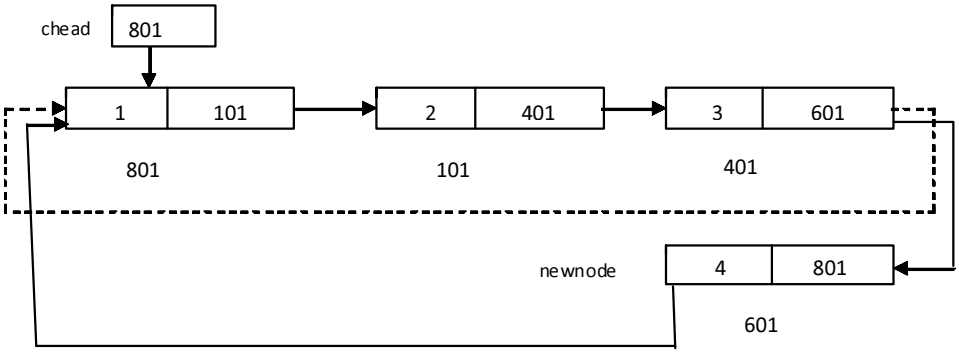


Fig. 3.17 : Example for insertion of a new node into the last position in a circular linked list

Algorithm for inserting new node at the last position into a circular linked list:

insert_last(chead,clast,element)

Step 1. ALLOCATE MEMORY FOR newnode

Step 2. DATA(newnode) = element

Step 3. IF chead == NULL

Step 4. chead = newnode

Step 5. clast = newnode

Step 6. ADDRESS(clast) = chead

Step 7. END OF IF

Step 8. ELSE

Step 9. ADDRESS(clast) = newnode

Step 10. clast = ADDRESS(clast)

Step 11. ADDRESS(clast) = chead

Step 12. END OF ELSE

In fig. 3.17, a node with memory address “601” is inserted in to the last position of a circular linked list.

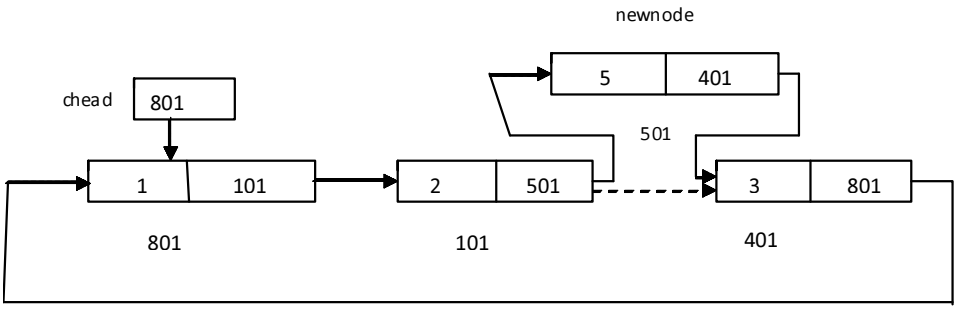


Fig. 3.18 : Example for insertion of a new node into the 3rd position in a circular linked list

Algorithm for inserting new node at a position which is inputed by the user into a circular linked list:

```
insert_at_p(chead,clast,element,pos)
Step 1.   count = 1
Step 2.   ALLOCATE MEMORY FOR newnode
Step 3.   DATA(newnode) = element
Step 4.   IF  pos<=0  OR  ( pos > 1 AND chead = NULL)
Step 5.       PRINT  "Wrong input for position "
Step 6.   END OF IF
Step 7.   ELSE
Step 8.       IF  pos == 1
Step 9.           ADDRESS(newnode) = chead
Step 10.          IF  chead == NULL
Step 11.              clast = newnode
Step 12.          END OF IF
Step 13.          chead = newnode
Step 14.          ADDRESS(clast) = chead
Step 15.      END OF IF
Step 16.      temp1 = chead
Step 17.      WHILE count<pos AND ADDRESS(temp1)!= chead
Step 18.          temp2 = temp1
Step 19.          temp1 = ADDRESS(temp1)
Step 20.          count = count + 1
Step 21.      END OF WHILE
Step 22.      IF count == pos
```

Step 23. ADDRESS(newnode) = temp1
Step 24. ADDRESS(temp2) = newnode
Step 25. END OF IF
Step 26. ELSE IF count == pos-1
Step 27. ADDRESS(temp1) = newnode
Step 28. clast = newnode
Step 29. ADDRESS(clast) = chead
Step 30. END OF ELSE IF
Step 31. ELSE
Step 32. PRINT "Wrong input for position "
Step 33. END OF ELSE
Step 34. END OF ELSE

In fig. 3.18, a node with memory address "501" is inserted in to the 3rd position of a circular linked list.

3.6.2 Deletion of a Node from a Circular Linked List

Here we will discuss deletion of the first node, the last node and the node whose position is inputed by the user from a circular linked list.

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

"clast" is a pointer which points the last node of the circular linked list.

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

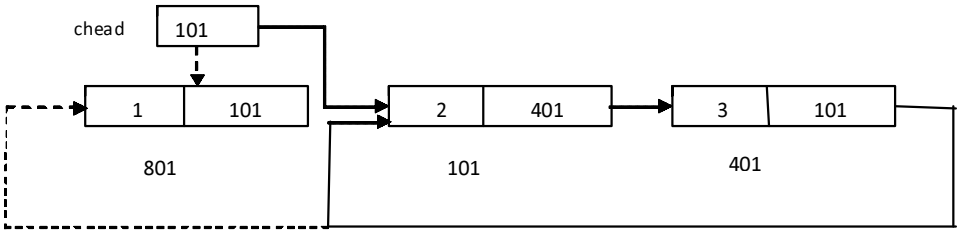


Fig. 3.19 : Example for deletion the first node from a circular linked list

Algorithm for deletion of the fist node:

delet_first(chead,clast)

Step 1. IF chead == NULL

Step 2. PRINT “Linked list is empty”

Step 3. END OF IF

Step 4. ELSE

Step 5. temp = chead

Step 6. IF ADDRESS(chead) == chead

Step 7. chead = NULL

Step 8. clast = NULL

Step 9. END OF IF

Step 10. ELSE

Step 11. chead = ADDRESS(chead)

Step 12. ADDRESS(clast) = chead

Step 13. END OF ELSE

Step 14. DEALLOCATE MEMORY FOR temp

Step 15. END OF ELSE

In fig. 3.19, the first node with memory address “801” is deleted from a circular linked list.

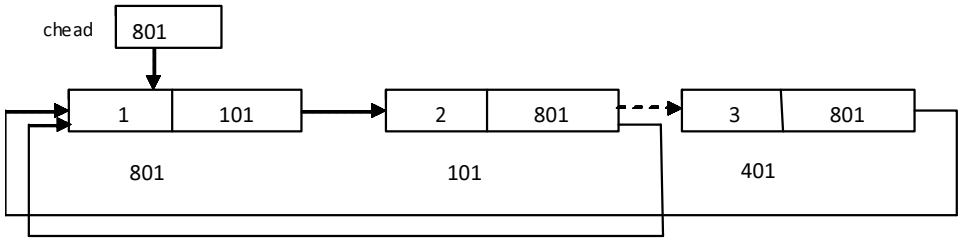


Fig. 3.20 : Example for deletion of the last node from a circular linked list

Algorithm for deletion of the last node:

delet_last(chead,clast)

Step 1. IF chead == NULL

Step 2. PRINT “ Linked list is empty”

Step 3. END OF IF

Step 4. ELSE

Step 5. temp1 = chead

Step 6. WHILE ADDRESS(temp1) != chead

Step 7. temp2 = temp1

Step 8. temp1 = ADDRESS(temp1)

Step 9. END OF WHILE

Step 10. IF ADDRESS(chead) == chead

Step 11. chead = NULL

Step 12. clast = NULL

Step 13. END OF IF

Step 14. ELSE

Step 15. ADDRESS(temp2) = chead

Step 16. clast = temp2

Step 17. END OF ELSE

Step 18. DEALLOCATE MEMORY FOR temp1

Step 19. END OF ELSE

In fig. 3.20, the last node with memory address "401" is deleted from a circular linked list.

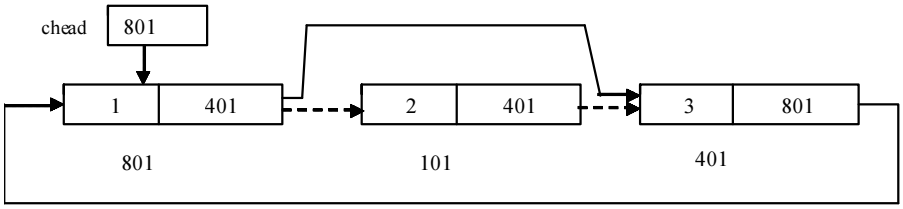


Fig. 3.21 : Example for deletion of the 2nd node from a circular linked list

Algorithm for deletion of the node whose position is inputed by the user:

delet_p(chead,clast,pos)

Step 1. count = 1

Step 2. IF chead == NULL

Step 3. PRINT " Linked list is empty "

Step 4. END OF IF

Step 5. ELSE

Step 6. temp1 = chead

Step 7. IF pos == 1

Step 8. IF ADDRESS(chead) == chead

```
Step 9.              chead = NULL
Step 10.             clast = NULL
Step 11.         END OF IF
Step 12.         ELSE
Step 13.             chead = ADDRESS(chead)
Step 14.             ADDRESS(clast) = chead
Step 15.         END OF ELSE
Step 16.         DEALLOCATE MEMORY FOR  temp1
Step 17.     END OF IF
Step 18.     WHILE count < pos AND ADDRESS(temp1)!= chead
Step 19.         temp2 = temp1
Step 20.         temp1 = ADDRESS(temp1)
Step 21.         count = count + 1
Step 22.     END OF WHILE
Step 23.     IF pos == count
Step 24.         ADDRESS(temp2) = ADDRESS(temp1)
Step 25.         IF  temp1 == clast
Step 26.             clast = temp2
Step 27.         END OF IF
Step 28.         DEALLOCATE MEMORY FOR  temp1
Step 29.     END OF IF
Step 30.     ELSE
Step 31.         PRINT " Wrong input for position "
Step 32.     END OF ELSE
Step 33. END OF ELSE
```

In fig. 3.21, the 2nd node with memory address "101" is deleted from a circular linked list.

3.6.3 Traversal of Nodes in Circular Linked List

Traversal of nodes in circular linked list is same with the singly linked list. Here any node to any node movement is possible.

Algorithm for traversing nodes in a circular linked list:

```
traversal (chead)
Step 1.   IF chead == NULL
Step 2.       PRINT “The linked list is empty “
Step 3.   END OF IF
Step 4.   ELSE
Step 5.       temp = chead
Step 6.       temp = ADDRESS(temp)
Step 7.       WHILE  temp != chead
Step 8.           temp = ADDRESS(temp)
Step 9.       END OF WHILE
Step 10.  END OF ELSE
```

3.6.4 C Program to Implement Circular Linked Llist

```
#include<stdio.h>
#include<conio.h>
#define max 40
//STRUCTURE TO CREATE NODE OF CIRCULAR LINKED
  LIST
struct cnode
{
    int data;
    struct cnode *next;
};
typedef struct cnode cnode;
//FUNCTION PROTOTYPES
void insert(cnode **,cnode **,int );
int insert_first(cnode **,cnode **,int);
int insert_last(cnode **,cnode **,int);
int insert_at_p(cnode **,cnode **,int,int);
void delet(cnode **,cnode **);
int delet_first(cnode **,cnode **);
int delet_last(cnode **,cnode **);
```

```
int delet_p(cnode **,cnode **,int);
int isempty(cnode *);
void init(cnode **,cnode **);
void creat(cnode **,cnode **);
void display(cnode *);
void main()
{
    cnode *chead,*clast;
    int option,elem,flag;
    char cont;
    clrscr();
    init(&chead,&clast);
    creat(&chead,&clast);
    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 linked list::");
                     scanf("%d",&elem);
                     insert(&chead,&clast,elem);
                     break;
            case 2:  delet(&chead,&clast);
                     break;
            case 3:  display(chead);
                     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 THE POINTER WHICH POINTS THE
 FIRST NODE OF THE CIRCULAR LINKED LIST
    void init(cnode **chead,cnode **clast)
    {
        *chead = NULL;
        *clast = NULL;
    }
//FUNCTION TO CHECK THE UNDERFLOW CONDITION
    int isempty(cnode *chead)
    {
        if(chead == NULL)
        return(1);
        else
        return(0);
    }
//FUNCTION TO INSERT NEW NODE INTO A CIRCULAR
 LINKED LIST
    void insert(cnode **chead,cnode **clast,int element)
    {
        int opt,pos;
        printf("\n*******INSERT MENU*********");
        printf("\n1.Insert at first position*");
        printf("\n2.Insert at last position *");
        printf("\n3.Insert at pth position  *");
        printf("\n**************************");
        printf("\nEnter your option::");
```

```
scanf("%d",&opt);
switch(opt)
{
    case 1: if(insert_first(chead,clast,element))
            {
            printf("\n%d is succesfully inserted at the
            first position",element);
            printf("\nAfter insertion the linked list is::\n");
            display(*chead);
            }
            else
            {
            printf("\nInsertion isnot successfull");
            }
            break;
    case 2: if(insert_last(chead,clast,element))
            {
            printf("\n%d is succesfully inserted at the
            last position",element);
            printf("\nAfter insertion the linked list is::\n");
            display(*chead);
            }
            else
            {
            printf("\nInsertion isnot successfull");
            }
            break;
    case 3: printf("\nEnter the position::");
            scanf("%d",&pos);
            if(insert_at_p(chead,clast,element,pos))
            {
            printf("\n%d is succesfully inserted at %d
            position",element,pos);
```

```
                    printf("\nAfter insertion the linked list is::\n");
                    display(*chead);
                    }
                    else
                    {
                    printf("\nInsertion isnot successfull");
                    }
                    break;
            default:printf("\nWrong input..Try agail");
        }
    }
//FUNCTION TO DELETE A NODE FROM THE CIRCULAR
 LINKED LIST
    void delet(cnode **chead,cnode **clast)
    {
        int opt,pos,elem;
        printf("\n*******DELETE MENU*********");
        printf("\n1.Delete the first node   *");
        printf("\n2.Delete the last node    *");
        printf("\n3.Delete the pth node     *");
        printf("\n**************************");
        printf("\nEnter your option::");
        scanf("%d",&opt);
        switch(opt)
        {
            case 1:  elem = delet_first(chead,clast);
                    if(elem == -99)
                    {
                    printf("\nDeletion isnot possible as the linked
                    list is empty");
                    }
                    else
                    {
```

```
                printf("\n%d is succesfully deleted",elem);
                printf("\nAfter deletion the linked list is::\n");
                display(*chead);
                }
        break;
case 2: elem = delet_last(chead,clast);
                if(elem == -99)
                {
                printf("\nDeletion isnot possible as the linked
                list is empty");
                }
                else
                {
                printf("\n%d is succesfully deleted",elem);
                printf("\nAfter deletion the linked list is::\n");
                display(*chead);
                }
                break;
case 3: printf("\nEnter the position::");
                scanf("%d",&pos);
                elem = delet_p(chead,clast,pos);
                if(elem == -99)
                {
                printf("\nDeletion isnot possible as the linked
                list is empty");
                }
                else if(elem == -98)
                printf("\nWrong input for position");
                else
                {
                printf("\n%d is succesfully deleted",elem);
                printf("\nAfter deletion the linked list is::\n");
                display(*chead);
```

```
                    }
                    break;
                default:printf("\nWrong input..Try agail");
            }
        }
//FUNCTION TO DISPLAY THE ELEMENTS IN A CIRCULAR
 LINKED LIST
    void display(cnode *chead)
    {
        cnode *temp;
        temp = chead;
        if(isempty(chead))
        printf("\nThe linked list is empty");
        else
        {
        printf("\nthe elements in the circular linked list are:\n");
        printf("%5d",temp->data);
        temp = temp->next;
        while(temp != chead)
        {
        printf("%5d",temp->data);
        temp = temp->next;
        }
    }
//FUNCTION TO CREATE A CIRCULAR LINKED LIST
    void creat(cnode **chead,cnode **clast)
    {
        cnode *newnode;
        char cont;
        printf("\nCreation of Circular linked list");
        do
        {
        newnode = (cnode *)malloc(sizeof(cnode));
```

```
            printf("\nEnter data::");
            scanf("%d",&newnode->data);
            if(*chead == NULL)
            {
            *chead = newnode;
            *clast = newnode;
            (*clast)->next = *chead;
            }
            else
            {
            (*clast)->next = newnode;
            *clast = newnode;
            (*clast)->next = *chead;
            }
            printf("\nDo you want to add more node..press 'y' or 'Y' to
            continue::");
            cont = getch();
        }while(cont == 'y' || cont == 'Y');
        printf("\nCircular linked list is created");
}
//FUNCTION TO INSERT A NEW NODE AT THE FIRST
 POSITION
        int insert_first(cnode **chead,cnode **clast,int element)
        {
            cnode *newnode;
            newnode = (cnode *)malloc(sizeof(cnode));
            newnode->data = element;
            if(*chead == NULL)
            {
                *chead = newnode;
                *clast = newnode;
                newnode->next = *chead;
                return(1);
```

```
        }
        else
        {
            newnode->next = *chead;
            *chead = newnode;
            (*clast)->next = *chead;
            return(1);
        }
    }
//FUNCTION TO INSERT A NEW NODE AT THE LAST
 POSITION
    int insert_last(cnode **chead,cnode **clast,int element)
    {
        cnode *newnode;
        newnode = (cnode *)malloc(sizeof(cnode));
        newnode->data = element;
        if(*chead == NULL)
        {
            *chead = newnode;
            *clast = newnode;
            (*clast)->next = *chead;
            return(1);
        }
        else
        {
            (*clast)->next = newnode;
            *clast = (*clast)->next;
            (*clast)->next = *chead;
            return(1);
        }
    }
//FUNCTION TO INSERT A NEW NODE AT PTH POSITION
    int insert_at_p(cnode **chead,cnode **clast,int element,int pos)
```

```
{
    cnode *temp1,*temp2,*newnode;
    int count = 1;
    newnode = (cnode *)malloc(sizeof(cnode));
    newnode->data = element;
    if(pos <=0 || (pos > 1 && *chead == NULL))
    {
    printf("\nWrong input for position");
    return(0);
    }
    else
    {
    if(pos == 1)
    {
    newnode->next = *chead;
    if(*chead == NULL)
    *clast = newnode;
    *chead = newnode;
    (*clast)->next = *chead;
    return(1);
    }
    temp1 = *chead;
    while(count < pos && temp1->next != *chead)
    {
    temp2 = temp1;
    temp1 = temp1->next;
    count++;
    }
    if(count == pos)
    {
    newnode->next = temp1;
    temp2->next = newnode;
    return(1);
```

```
        }
        else if(count == pos-1)
        {
        temp1->next = newnode;
        *clast = newnode;
        (*clast)->next = *chead;
        return(1);
        }
        else
        {
        printf("\nWrong input for position");
        return(0);
        }
    }
//FUNCTION TO DELETE THE FIRST NODE
    int delet_first(cnode **chead,cnode **clast)
    {
        cnode *temp;
        int delem;
        if(*chead == NULL)
        return(-99);
        else
        {
        temp = *chead;
        delem = temp->data;
        if((*chead)->next == *chead)
        {
        *chead = NULL;
        *clast = NULL;
        }
        else
        {
        *chead = (*chead)->next;
```

```
            (*clast)->next = *chead;
            }
            free(temp);
            return(delem);
            }
        }
    //FUNCTION TO DELETE THE LAST NODE
        int delet_last(cnode **chead,cnode **clast)
        {
            cnode *temp1,*temp2;
            int delem;
            if(*chead == NULL)
            return(-99);
            else
            {
            temp1 = *chead;
            while(temp1->next != *chead)
            {
            temp2 = temp1;
            temp1 = temp1->next;
            }
            delem = temp1->data;
            if((*chead)->next == *chead)
            {
            *chead = NULL;
            *clast = NULL;
            }
            else
            {
            temp2->next = *chead;
            *clast = temp2;
            }
            free(temp1);
```

```
            return(delem);
            }
        }
    //FUNCTION TO DELETE THE PTH NODE
        int delet_p(cnode **chead,cnode **clast,int pos)
        {
            cnode *temp1,*temp2;
            int delem,count = 1;
            if(*chead == NULL)
            return(-99);
            else
            {
            temp1 = *chead;
            if(pos == 1)
            {
            delem = (*chead)->data;
            if((*chead)->next == *chead)
            {
            *chead = NULL;
            *clast = NULL;
            }
            else
            {
            *chead = (*chead)->next;
            (*clast)->next = *chead;
            free(temp1);
            return(delem);
            }
            }
            while(count < pos && temp1->next != *chead)
            {
            temp2 = temp1;
            temp1 = temp1->next;
```

```
            count++;
            }
            if(pos == count)
            {
                delem = temp1->data;
                temp2->next = temp1->next;
                if(temp1 == *clast)
                *clast = temp2;
                free(temp1);
                return(delem);
                }
                else
                {
                return(-98);
                }
            }
        }
```

3.7 COMPARATIVE STUDIES WITH IMPLEMENTATIONS USING ARRAY STRUCTURE

- Array is a static data structure. So in case of an array the amount of data can be stored is fixed at compile time. On the other hand in case of linked list the data can be stored and removed dynamically at runtime, so if memory is available, data can de inserted into a linked list.
- In case of array, direct access of data can be possible using the subscript value of the array as the element in an array are stored in contiguous memory locations. But on the other hand, in case of linked list the elements may not be stored in contiguous memory locations, so direct access is not possible. In a linked list, the elements are accessed by traversing from the first node to the required node using a pointer.

- In case of array, in some cases, to insert or to delete data, we require shifting of data. But in case of linked list shifting of data is not required. Here, only the links of the nodes are to be managed carefully.

CHECK YOUR PROGRESS

Q.1. Multiple choice question:

A. In linked list, a node contains at least

i) node address field, data field, node number

ii) node number, data field

iii) next address field, information field

iv) none of these

B. The nth node, in singly linked list is accessed via

i) the head node ii) the tail node

iii) (n-1) nodes iv) None of these

C. In linked list, the successive elements

i) must occupy contiguous space in memory

ii) need not occupy contiguous space in memory

iii) must not occupy contiguous space in memory

iv) None of these

D. Under flow condition in linked list may occur when attempting to

i) insert a new node when there is no free space for it

ii) delete a non existent node in the list

iii) delete a node in empty list

iv) none of these

E. Overflow condition in linked list may occur when attempting to

i) insert a node into a linked list when free space pool is empty

ii) traverse the nodes when free space pool is empty

iii) insert a node into a linked list when linked list is empty

iv) None of these

F. Which is not a type of linked list

i) Singly linked list ii) Doubly linked list

iii) Sequential linked list iv) Circular linked list

G. Circular linked list can be used to implement

i) Circular queue ii) Priority queue

iii) Deque iv) Both (ii) and (iii)

H. Less memory required in case of

i) Singly linked list ii) Doubly linked list

iii) Circular linked list iv) Both (i) and (iii)

I. Inserting a node in a doubly linked list after a given node requires

i) One pointer change ii) Four pointer change

iii) Two pointer change iv) None of the above

J. Traversing can be done in both directions in case of

i) singly linked list ii) circular linked list

iii) doubly linked list iv) Both B and C

Q.2. Fill in the blanks:

A. _____ linked list does not have any NULL links.

B. In a circular linked list if the address field of a node point itself then the total number of nodes in the circular linked list is _____.

C. _____ access of element in linked list is not possible.

D. The address field of the first node in a singly linked list contains _____ if the number of nodes is one.

E. In case of singly linked list, if a node contains NULL then it means _____.

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

A. Circular linked list dose not have any NULL links

B. Traversal from any node to any node is possible in case of doubly linked list and circular linked list.

C. In singly linked list the Pth node can be deleted from the (P+1)th node

D. In a circular linked list with four nodes, the address field of the third node contains the address of the first node.

E. In doubly linked list, the Pth node can be deleted from (P+1)th and (P-1)th node.

3.8 LET US SUM UP

- Linked list is a linear dynamic data structure. It is a collection of some nodes containing homogeneous elements. Each node consists of a data part and one or more address part depending upon the types of the linked list.
- In case of linked list, the amount of data will be stored is not fixed at compile time like array. Here data can be stored and removed dynamically at run time.
- In case of linked list, data need not to be stored in contiguous memory locations like array. So direct access of any node is not possible.
- There three different types of linked list available which are singly linked list, doubly linked list and circular linked list.
- A node of singly linked list has two fields, one field contains information and other field contain address of the next node. The address field of the last node contain NULL.
- The structure of a node of circular linked list is same with singly linked list. The address field of the last node of a circular linked list contains the address of the first node.
- A node of doubly linked list has three fields, one field contain information, one field contain the address of the next node and the third field contains the address of the previous node.

3.9 FURTHER READINGS

- Ellis Horowitz, Sartaj Sahni : *Fundamentals of data structures*, Computer Science Press.
- Yedidyah Langsam,Moshe J. Augenstein, Aaron M.Tenenbaum : *Data structures using C and C*++, Prentice-Hall India.

3.10 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1. : A. (iii), B. (iii), C. (ii), D.(iii), E.(i), F. (iii), G. (iv), H. (iv), I.(ii), J. (iii)

Ans. to Q. No. 2. : A. circular, B. one, C. direct, D. NULL,
E. end of the list or last node of the list

Ans. to Q. No. 3. : A. true, B. true, C. false, D. false, E. true

3.11 MODEL QUESTIONS

Q.1. Implement stack and queue using circular linked list.

Q.2. Write a function to delete the previous node of the last node from a doubly linked list.

Q.3. Write a function to insert a new node before a given node into a singly linked list.

Q.4. Write down a comparison about the three types of linked list.

UNIT 4 : STACKS

UNIT STRUCTURE

4.1 LEARNING OBJECTIVES

After going through this unit, you will able to :

- learn the concept of stack
- learn about LIFO structure
- describe the operation associated with a stack
- implement stack using arrays
- implement stack using linked lists
- learn about some applications of stack

4.2 INTRODUCTION

So far we have been looking at the primitive data structures that are available in C. While studying linked list we have seen insertion and deletion of element is possible at any place in the list. There are certain situations in which items may be inserted or removed only at one end. Stacks and queues are data structures. In this unit, we'll be able to learn about the stack data structure.

4.3 DEFINITION OF 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 a 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. Fig.(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 one 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.

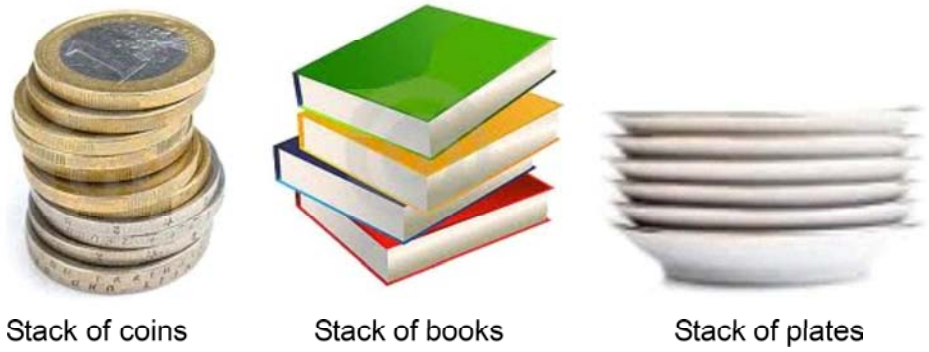


Fig. 4.1 : Real life examples of Stack

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 Fig.(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.

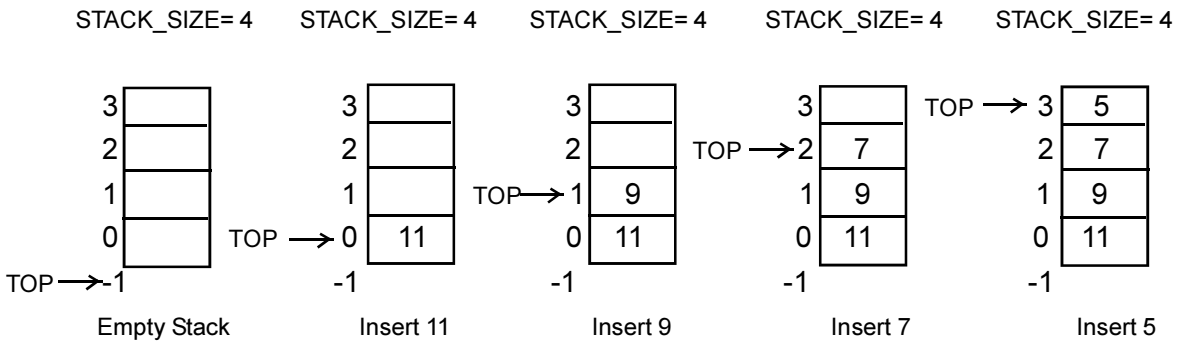


Fig. 4.2 : Sequence of *Push* operation

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 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 element. Trying to delete an element from an empty stack results in stack ***underflow***. Fig.(4.3) illustrates the pop operation. After every pop operation, the value of TOP is decremented by one.

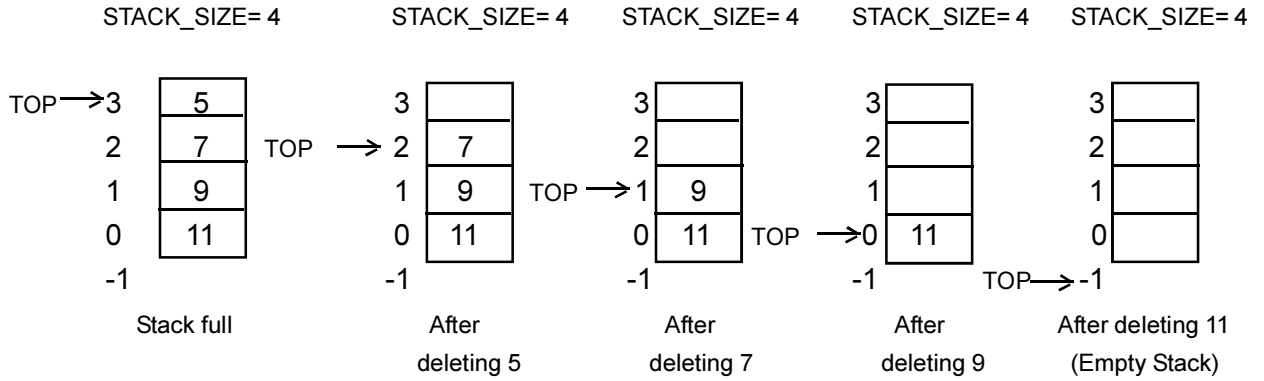


Fig. 4.3 : Sequence of Pop operation

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 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.

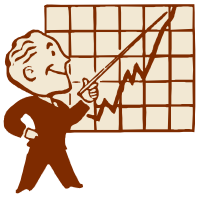
To push (or insert) an element onto the stack, *Top* is incremented by one, and the element is pushed at that position. When *Top* reaches SIZE-1 and an attempt is made to push a new element, then the stack overflows. Here, SIZE is the maximum size of the stack. Similarly, to pop (or remove) an element 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(T) or false(F):

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 of stack using linked list (i.e., Linked list Implementation */*

```
#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.

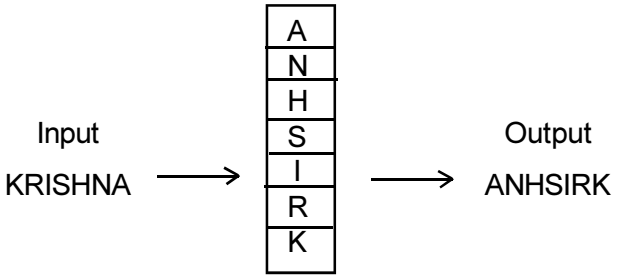


Fig. 4.5 : Stack

```
// Program 4.3: Program of 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.*

LET US KNOW

Infix, Prefix and Postfix Notations : An arithmetic expression consists of operators and operands. In solving an expression, the precedence and associativity of operator plays an important role. Arithmetic expressions are defined in three kinds of notation: *infix*, *prefix* and *postfix*.

Usually, arithmetic expressions are written in ***infix*** notation with binary operator between the operands. If A and B are two valid expressions then (A+B), (A - B), (A / B) are all legal infix expressions. Following are the examples of infix notation:

A+B*C

A+B

(A+B)*C

(X+Y)*(L+M)

In the ***prefix*** notation, the operator precedes the operands. The prefix notation is also termed as *polish notation*. Following are some examples of prefix notation:

+AB

+A*BC

*+ABC

*+AB+PQ

An expression can as well be written in ***postfix*** notation (also called *reverse polish notation*). Here, the operator trails the operands. Following are the examples of postfix expressions:

A+B becomes AB+

A*C becomes AC*

A+(B*C) becomes ABC*+

(A+B)*C becomes AB+C*

Let us explain how the infix expression A+(B*C) can be converted to postfix form ABC*+.

We know that multiplication has higher precedence than addition. By applying the rules of priority, we can write

A+(B*C) as A+(BC)*

where multiplication is converted to postfix. Then we can write

A+(BC)* as A(BC)*+

where addition is converted to postfix. The postfix expression A(BC)* is same as ABC*+ (i.e., parentheses may or may not be applied).

Table : Arithmetic operators along with priority values

Arithmetic Operators	Priority	Associativity
Exponentiation (^)	6	Right to Left
Multiplication (*)	4	Left to Right
Division (/)	4	Left to Right
Mod (%)	4	Left to Right
Addition (+)	2	Left to Right
Subtraction (–)	2	Left to Right

Evaluation of arithmetic expressions : Stacks can also be useful in 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 operand2

  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

vi) The postfix notation is also called as

a) prefix notation b) polish notation

c) reverse polish notation d) infix notation

Q.5. Distinguish between static and dynamic implementation of stack.

4.7 LET US SUM UP

- In Computer Science, 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 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. Hence, 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 implementation can be implemented using arrays. However, it is a very simple method but it has few limitations.
- Pointer can also be used for implementation of stack. Linked list implementation is an example of this type of dynamic implementation. The limitations noticed in static implementation can be removed by using dynamic implementation.

4.8 FURTHER READINGS

- *Data Structures* by Seymour Lipschutz, Tata McGraw-Hill publication.
- *Introduction to Data Structure in C* by Kamthane, Pearson Education publication

- *Data Structures through C in Depth* by S.K. Srivastava, Deepali Srivastava, 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,
vi) (c) reverse polish notation

Ans. to Q. No. 5. : (See text content)

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 do you mean by *stack overflow* and *stack underflow* ?

Q.5. What do you mean by 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

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

- learn about queue data structure
- describe array implementation of queue
- learn about the usefulness of circular queue and implement it
- describe list implementations of queue
- illustrate different applications of queue

5.2 INTRODUCTION

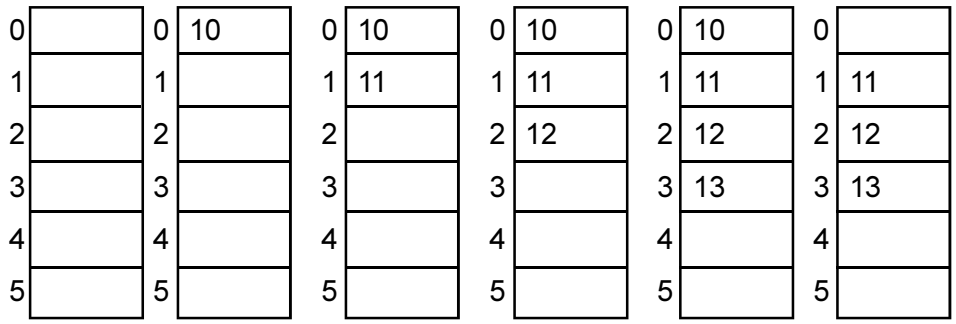
In unit 4 we have learnt about stack. Now there is another linear data structure available known as queue.

5.3 DEFINITION OF 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 take place at rear end and deletion is take place at front end. Queue is also known as first in first out (FIFO) data structure.

5.4 ARRAY IMPLEMENTATION OF QUEUE

In array implementation, an array and two variables, 'front' and 'rear' is used. 'front' and 'rear' are initialized with value '–1'. Here to insert an element, at first the value of 'rear' is incremented if it is possible 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 using 'C' 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 an 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'.



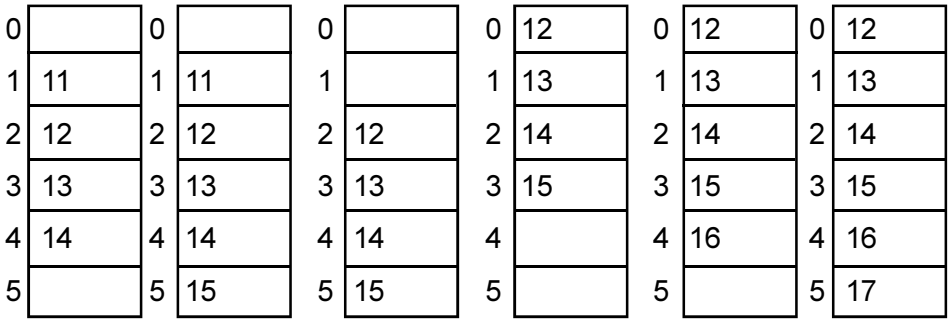


Fig. 5.1 : Example of insertion and deletion operation on a queue

Algorithm for inserting a 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

In fig. 5.1(a), an empty queue is represented with front = –1 and rear = –1.

In fig. 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 fig. 5.1(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 queue in fig. 5.1(d), 5.1(e), 5.1(g), 5.1(h), 5.1(k) and 5.1(l) respectively.

In fig. 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.

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

In fig. 5.1(f) , 10 is deleted with front = 1 and rear = 3.

In fig. 5.1(i), 11 is deleted with front = 2 and rear = 5.

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[q->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 has make wastage of time which 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. So here the array is imagined 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.

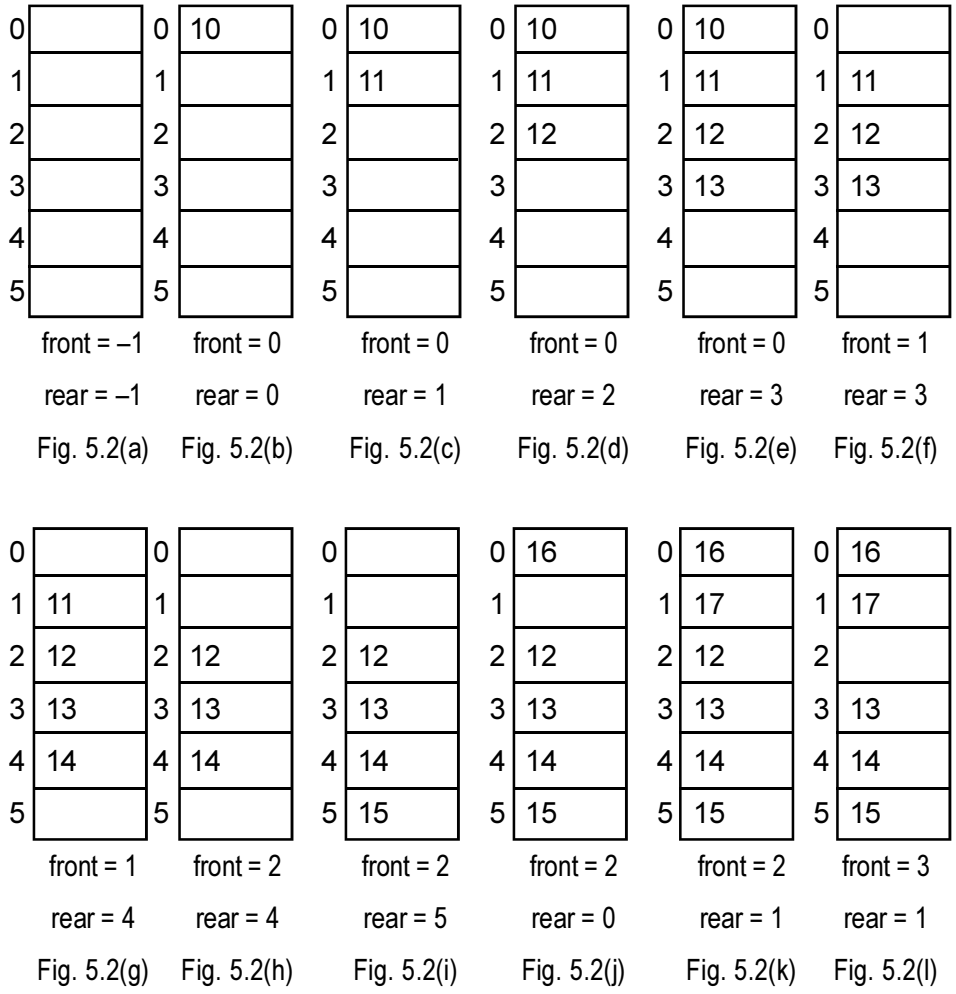


Fig. 5.2. Example of insertion and deletion operation on a circular queue

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 to be inputted into the circular queue.

Here 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 other wise it will be one more than the recent value of 'rear' or 'front'.

```
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 fig. 5.2(a), an empty circular queue is represented with front = –1 and rear = –1.

In fig. 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 fig. 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 2(d),2(e),2(g),2(i),2(j) and 2(k) respectively.

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

In fig. 5.2(f),10 is deleted from the circular queue with front = 1 and rear = 3.

In fig. 5.2(h),11 is deleted from the circular queue with front = 2 and rear = 4.

In fig. 5.2(l), 12 is deleted from the circular queue with front = 3 and rear = 1.

C program to implement circular queue:

```
#include<stdio.h>
#include<conio.h>
#define max 5
//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 sections 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'. 'front' points the first element in the list and 'rear' points the last elements in the list. '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.

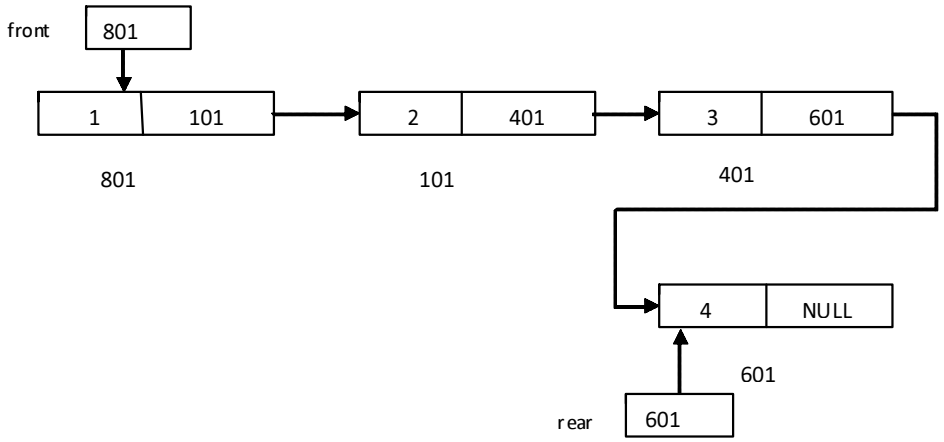


Fig. 5.3(a) : Example of a queue implemented using singly linked list

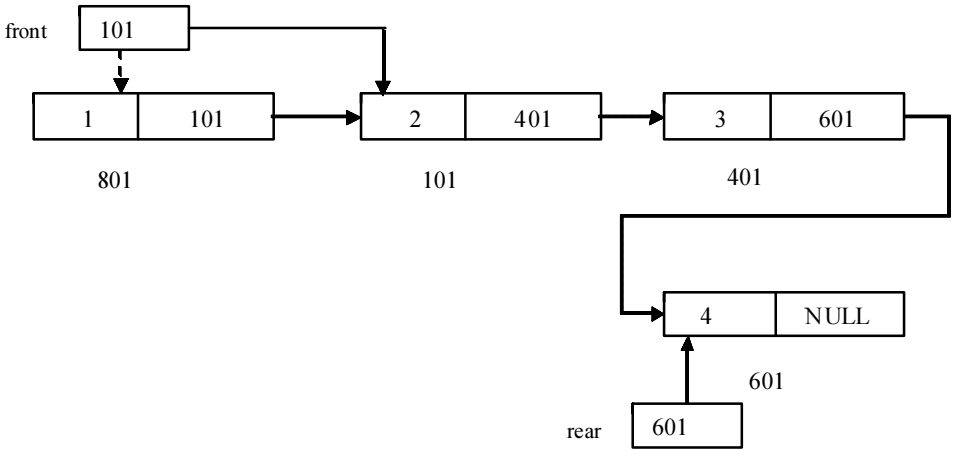


Fig. 5.3(b) : Deletion of queue element

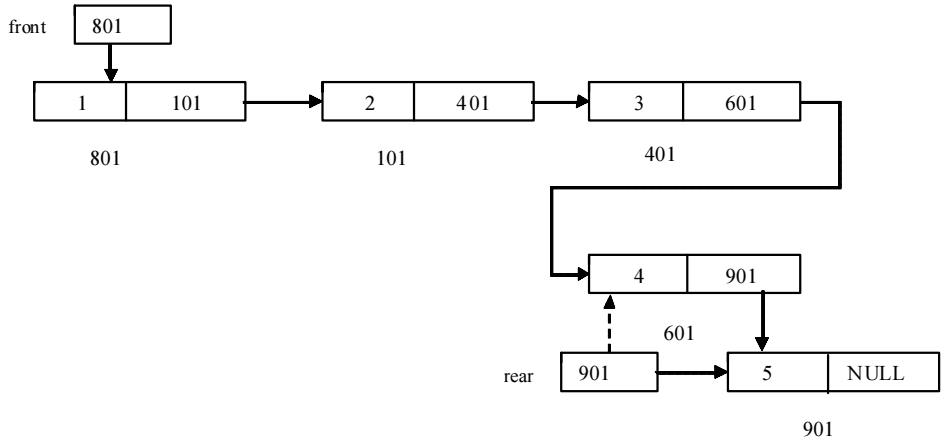


Fig. 5.3(c) : Insertion of new queue element

Algorithm for inserting a 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 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
```

In fig. 5.3(c), a new node with data “5” is inserted into the queue which is implemented using doubly linked list.

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
```

In fig. 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.

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'. 'front' points the first element in the list and 'rear' points the last elements in the list. '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.

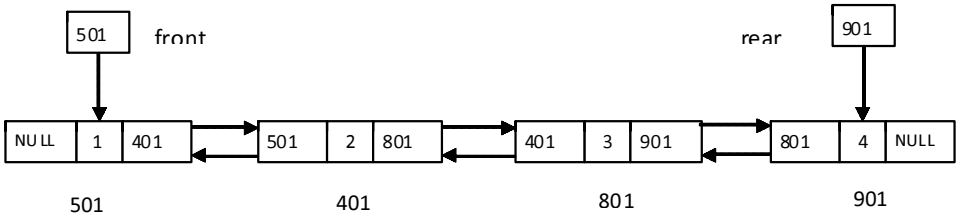


Fig. 5.4(a) : Example of a queue implemented using doubly linked list

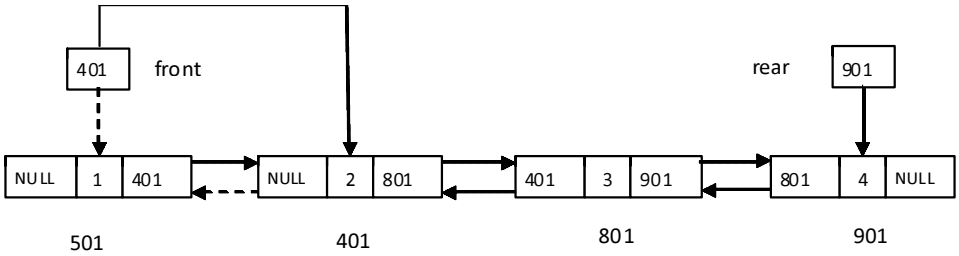


Fig. 5.4(b) : Deletion of queue element

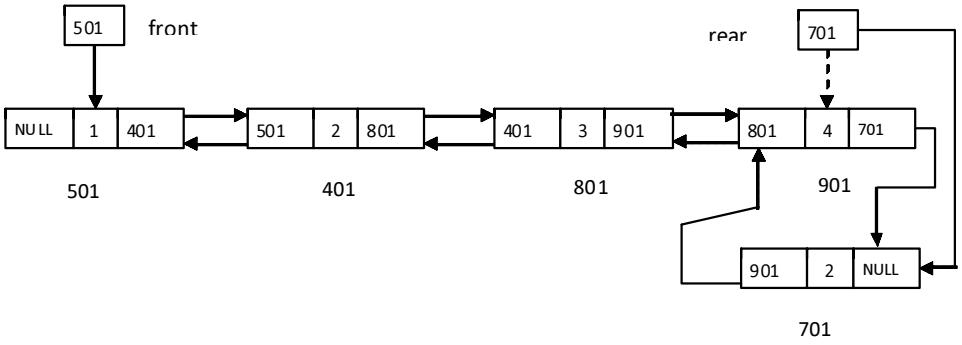


Fig. 5.4(c) : Insertion of new queue element

Algorithm for inserting a 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
```

In fig. 5.4(c), a new node with data "2" is inserted into the queue which is implemented using doubly linked list.

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
```

In fig. 5.4(b), the node pointed by the pointer 'front' with data "1" is deleted from the queue which is implemented using doubly linked list.

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'. 'front' points the first element in the list and 'rear' points the last elements in the list. '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.

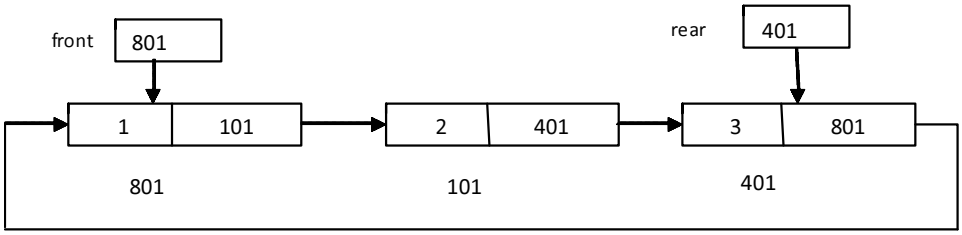


Fig. 5.5(a) : Example of a queue implemented using circular linked list

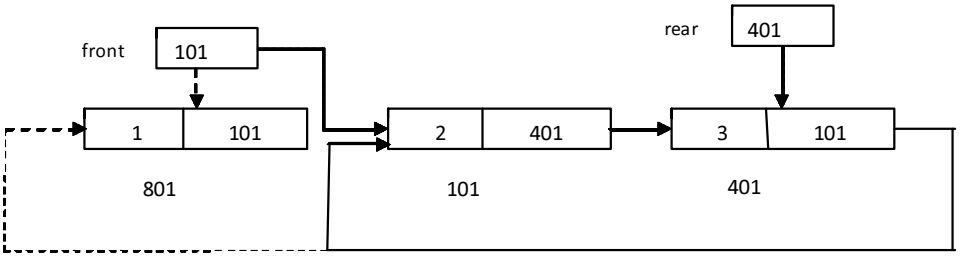


Fig. 5.5(b) : Deletion of queue element

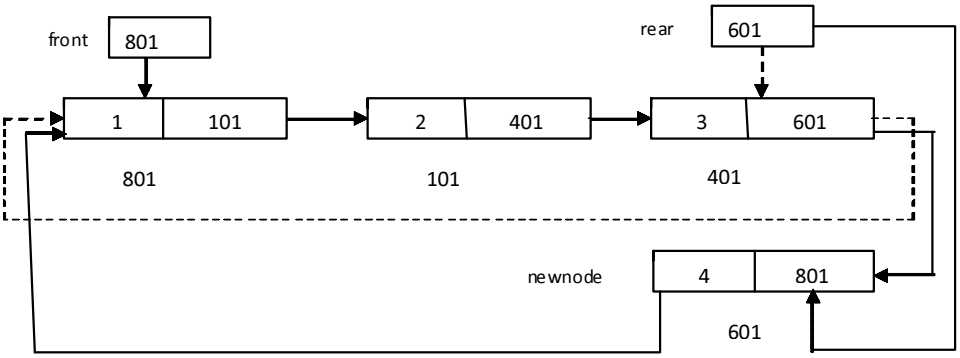


Fig. 5.5(c) : Insertion of new queue element

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

In fig. 5.5(c), a new node with data "4" is inserted into the queue which is implemented using circular linked list.

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
```

In fig. 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.

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 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.

5.8 PRIORITY QUEUES

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

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.

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 take place at rear end and deletion is 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 is depended upon some priority value associated with each element

in the queue. In case of max-priority queue, the element has highest priority value will be deleted first and in case of min-priority queue the element has lowest priority value will be deleted first.

5.10 FURTHER READINGS

- Yedidyah Langsam,Moshe J. Augenstein, Aaron M.Tenenbaum: *Data structures using C and C++*, Prentice-Hall India.
- Ellis Horowitz, Sartaj Sahni: *Fundamentals of Data Structures*, Galgotia Publications.

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. What is queue? Why circular queue is needed? Implement circular queue in C.

Q.2. Write a C program to implement queue using circular linked list.

Q.3. Write down algorithms for insert and delete operation on a queue.

UNIT 6 : SEARCHING

UNIT STRUCTURE

6.1 LEARNING OBJECTIVES

After going through this unit, you will able to :

- learn about searching techniques
- describe linear seach and binary search
- search an element in sorted and unsorted list of elements
- learn about the advantages and disadvantages of linear and binary search techniques.
- analyse linear and binary search

6.2 INTRODUCTION

In our day-to-day life there are various applications, in which the process of searching is to be carried. Searching a name of a person from the given list, searching a specific card from the set of cards, etc., are 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 searching for data stored in a linear data structure such as an array or linked list.

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 Fig. 6.1(a).

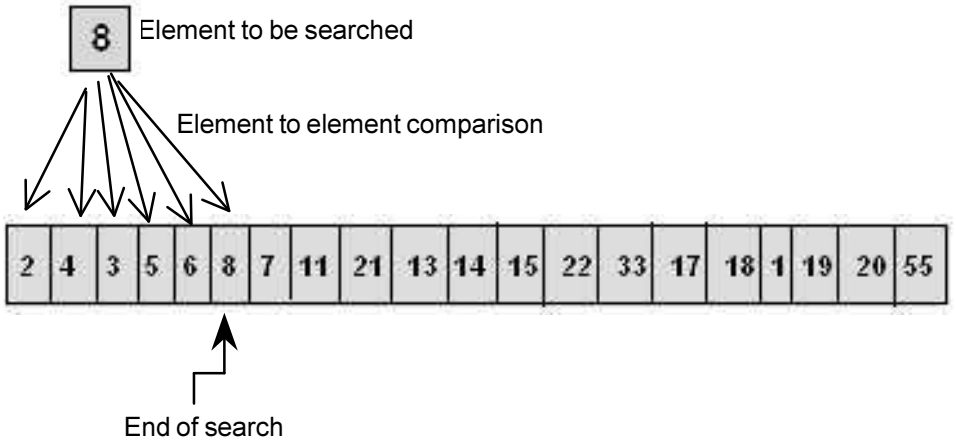


Fig. 6.1(a) : Successful search

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 Fig. 6.1(b).

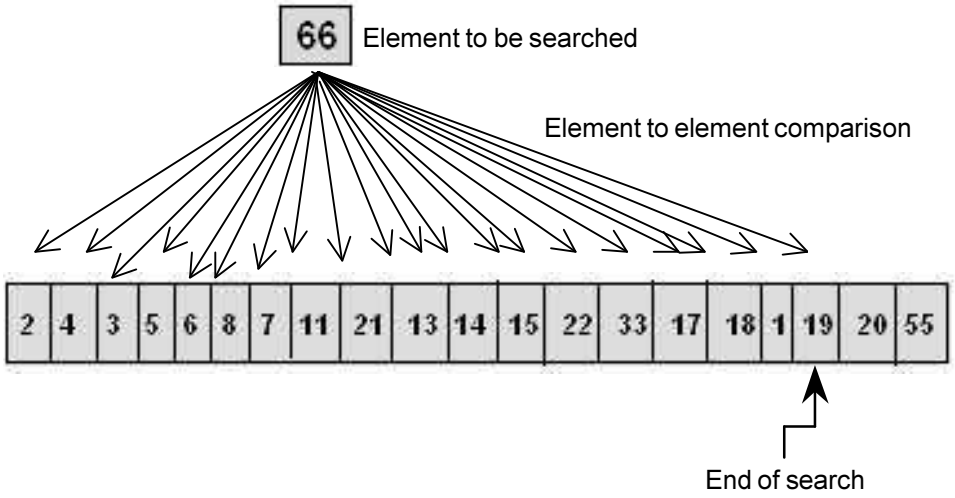


Fig. 6.1(b) : Unuccessful search

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 seacrch 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 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();
}
```

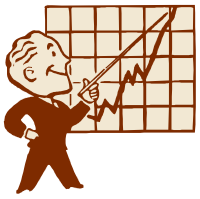
In the above program suppose user enters few numbers using the first *for* loop (Number of elements should be less or equal to 50 as the array size is 50). The element which is to be searched is stored in the variable *item*. By using a second *for* loop the element is compared with each element of the 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

Q.1. 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 Fig 6.2(a). Suppose we want to search the element 56 from the array of elements.

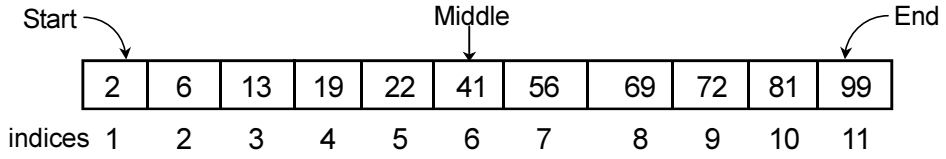


Fig. 6.2 (a)

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 :

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

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 Fig. 6.2(b).

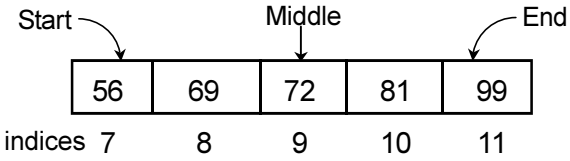


Fig. 6.2 (b)

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

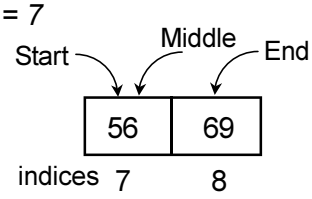


Fig. 6.2(c)

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.

Binary Search Algorithm :

Input: (List, Key)

WHILE (List is not empty) DO

(Select “middle” entry in list as test entry

IF (Key = test entry) THEN (Output(“Found it”) Stop)

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

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: Write a 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();
    }
```

6.5 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(log n), will find the target element with a worst case of only 20 comparisons. Linear search, O(n), on average will take 500,000 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(log n).

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.

6.6 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.

6.7 FURTHER READINGS

- *Data Structures :* Seymour Lipschutz, Tata McGraw-Hill.
- *Data Structure's and Program Design :* Robert. L. Kruse, PHI.

6.8 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;
}
```

6.9 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

After going through this unit, you will able to

- define sorting
- describe insertion sort algorithm
- trace 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.

In this unit, we will introduce you to the fundamental concepts of sorting. In this unit, we shall discuss about the various sorting algorithms i.e. insertion sort, selection sort, bubble sort and quick sort including their complexity.

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 address of the subscriber very easily. Like the same way in dictionary the words are placed in lexicographical order which nothing but a sorting.

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

$$L_1 \leq L_2 \leq L_3 \leq \quad ..... \leq L_n$$

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 often is 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 is an array with *n* elements A[0], A[1], ... A[N -1] is 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 -

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 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 + ... + (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 X 1 + (n–1) -1) X 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 element 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 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 you will search the position of smallest element from A[0].....A[N–1]. Then you 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–

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*.

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

```
#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) + ... + 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]

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 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;
        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

$F(n) = (n–1) + (n–2) + (n–3) + \ldots\ldots\ldots\ldots + 3 + 2 + 1$

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 as ***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 it's proper place.
3. Create two sublists left and right side of pivot.
4. Repeat the same process untill all elements of list are at proper position in list.

For placing the pivot element at proper place we have a need 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 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 it's 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

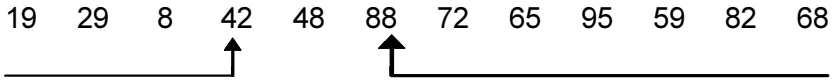
Now the comparision will start from 59 and will be from right to left. The first element less than 48 is 42. So interchange it with pivot.

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 it's proper position in the list. So we can divide the list into two sublist, left and right side of pivot.



Sublist 1 Sublist 2

Now the same procedure will be followed for the sublist1 and sublist2 and finally you will get the sorted list.

The following program will 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;
    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 that, the pivot element is choosen in the middle position of the list so that it divides the list into two sublist of equal size. Now, repeatedly applying the quick sort algoritm on both the sublist we will finally have the sorted list of the elements.

Now, after I^{st} step total elements in correct position is $= 1 = 2^1 - 1$

after 2^{nd} step total elements in correct position is $= 3 = 2^2 - 1$

after 3^{rd} step total elements in correct position is $= 7 = 2^3 - 1$

.....

.....

after I^{th} step total elements in correct position is $= 2^I - 1$

Therefore, $2^I - 1 = n - 1$

or $2^I = n$

or $I = \log_2 n$

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, suppose one condition that , the given list of the elements are initially sorted. We consider the first element of the list is as the pivot element. In this case, the number of steps needed for obtain the finally sorted list is = (n - 1).

Again, the number of comparisions in each step will be almost *n i .e. i*t 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. a) 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.

b) What is the worst-case time for quick sort to sort an array of n elements?

A. O(log n) B. O(n)

C. O(n log n) D. $O(n^2)$

c) 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.2. How does a quick sort performs on :

a) An array that is already sorted

b) An array that is sorted in reversed order

Q.3. The Bubble sort, the Selection sort, and the Insertion sort are all O(n2) algorithms. Which is the fastest and which is the slowest among them?

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 $O(n^2)$.
- In Selection Sort, the first element is compared with remaining (n–1) elements and 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. The complexity of Selection Sort is $O(n^2)$.
- The Bubble Sort is probably the simplest of the sorting algorithms. Its 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 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 guarantees that the single element, called the pivot element, is

in its correct polsition. Then the algorithm proceeds, applying the same method to the two pieces separately. This is naturally recursive, and very quick. Time consumption of the quick sort depends on the location of the pivot element inthe list. The complexity of Quick Sort is O(*nlogn*).

7.9 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1 : a) C, b) D, c) C,

Ans. to Q. No. 2 : The quick sort is quite sensitive to input. The Quick Sort degrade into an $O(n^2)$ algorithm in the special cases where the array is initially sorted in ascending or descending order. This is because if we consider the pivot element will be the first element. So here it produces only 1 sublist which is on right side of first element start from second element. Similarly other sublists will be created only at right side. The number of comparision. The number of comparision for first element is n, second element requires n - 1 comparision and so on. Thus, we will get the complexity of order n^2 instead *logn.*

Ans. to Q. No. 3 : The Bubble Sort is slower than the Selection Sort, and the Insertion Sort (in most cases) is a little faster.

7.10 FURTHER READINGS

- *Data structures using C and C++*, Yedidyah Langsam, Moshe J. Augenstein, Aaron M.Tenenbaum, Prentice-Hall India.
- *Data Structures*, Seymour Lipschutz, Schaum's Outline Series in Computers,Tata Mc Graw Hill.
- *Introduction to Data Structures in C*, Ashok N. Kamthane, Perason Education.

7.11 MODEL QUESTIONS

Q.1. Explain 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.

UNIT 8 : TREES

UNIT STRUCTURE

8.1 LEARNING OBJECTIVES

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

- define tree as abstract data type (ADT)
- learn the different properties of a Tree and Binary tree
- implement the Tree and Binary tree
- explain 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 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 and binary trees and binary search tree. We will also discuss about the different tree traversal techniques in this unit.

8.3 DEFINITION OF TREE

A tree is a nonlinear data structure, where a specially designed node called root and rest of the data/node are divided into n subsets. Where each subsets follow the properties of tree and the value of n is grater then 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 fig 8.1 a general tree is given and in fig 8.2 the structure of unix file system is given.

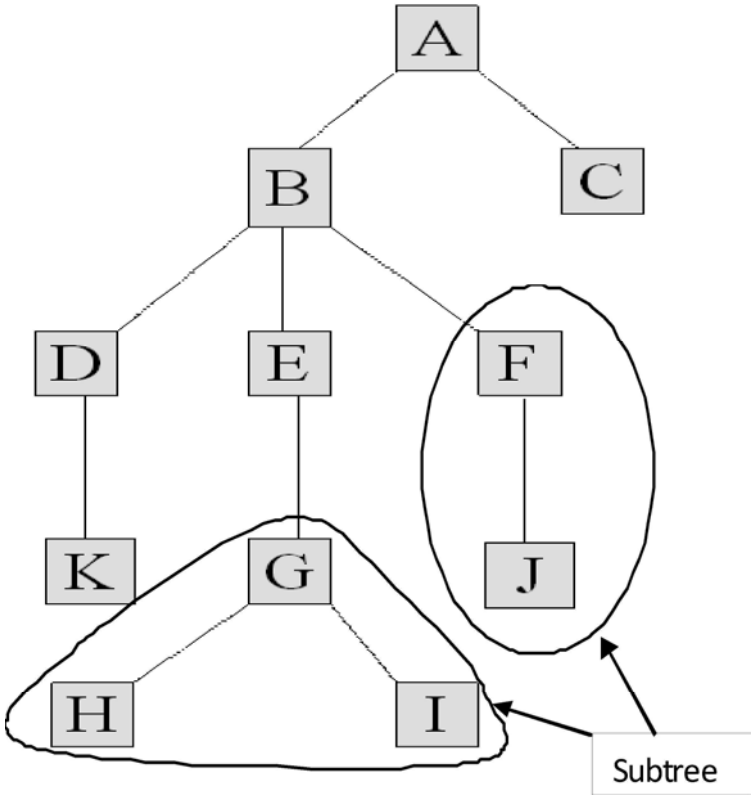


Fig. 8.1 : Tree

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 fig 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 fig 8.1 K, H, I, J and C are the leaf node.

Definition [Internal Node]: Anode is said to be a internal node if the node have one or more children. In fig 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 node have same parent. In fig 8.1 B and C are sibling, D, E, F are sibling and H and I are sibling.

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 fig 8.1 D id 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 fig 8.1 Height of the tree is 4.

Definition [Degree of a Node]: The degree of a node is the no of children of that node. For example in Fig 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 Unix file system is shown which is nothing but a tree structure.

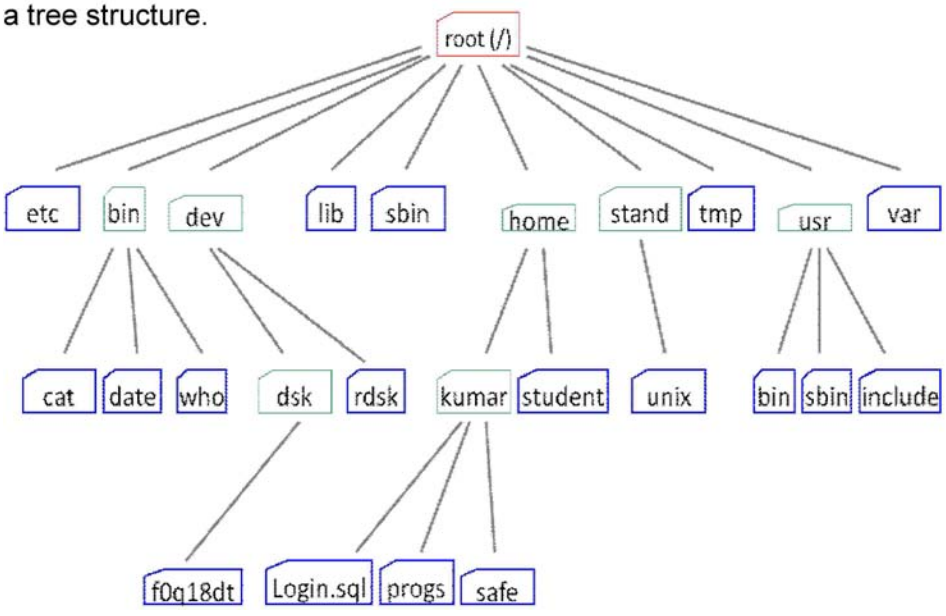


Fig. 8.2 : Unix File System

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

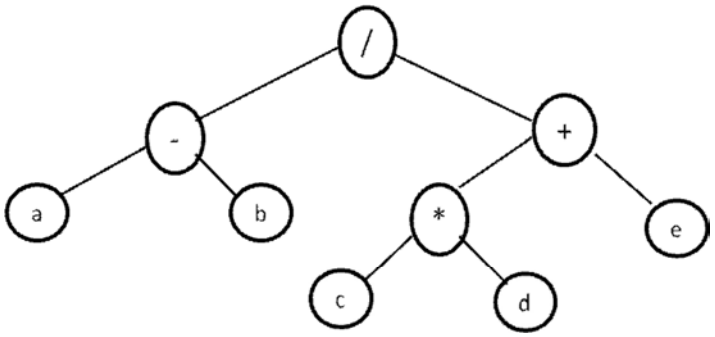


Fig. 8.3 : Representation of Arithmetic expression

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.

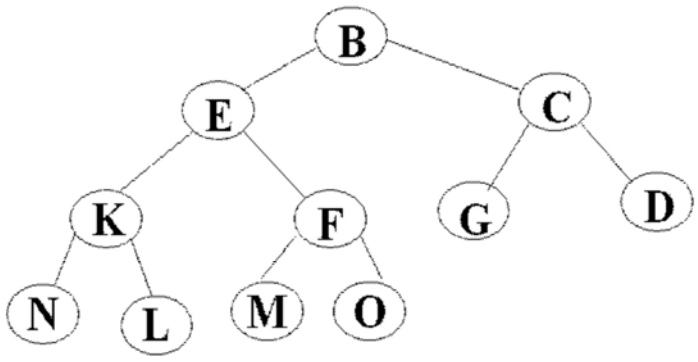


Fig. 8.4 : A complete binary tree

Definition [Full Binary tree]: A binary tree is said to be a full binary tree if all the levels contains maximum possible node.

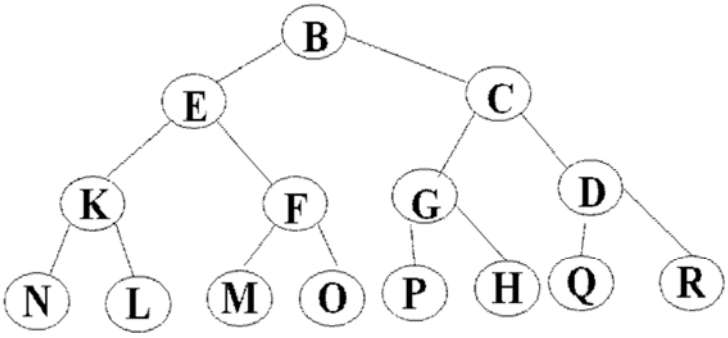


Fig. 8.5 : Full binary tree

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

Binary tree can be represented into the memory in two ways-

- Sequential Representation (by using Array)
- Link representation (by using node structure)

1. **Sequential Representation of Binary Tree:** In sequential representation of binary tree, a single array is used to represent a binary tree. For these, nodes are numbered/indexed according to a scheme giving 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, shown below the nodes of binary tree are numbered according to the given scheme

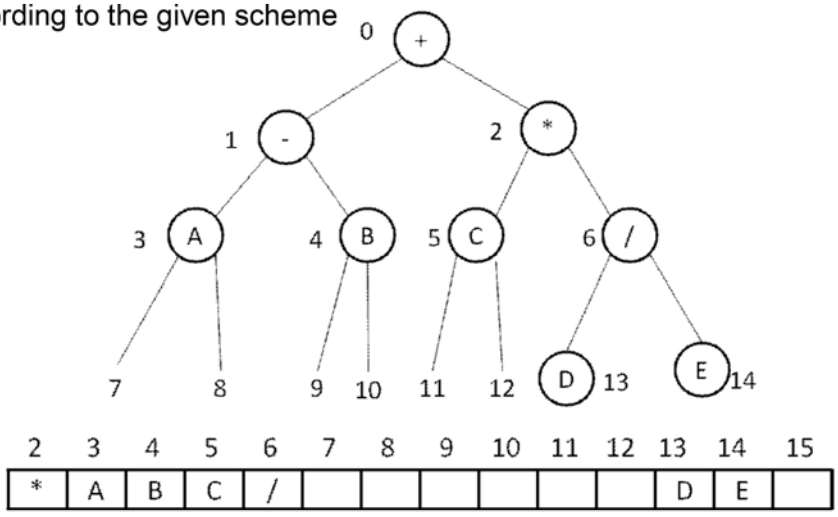


Fig. 8.6 : Array representation of binary tree

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 is 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 of its child, then null value is stored at the corresponding index of the array.

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

(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.

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.

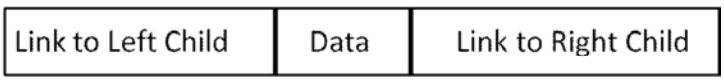


Fig. 8.7 : A node structure

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 fig 8.3 can be represented in link representation as follows

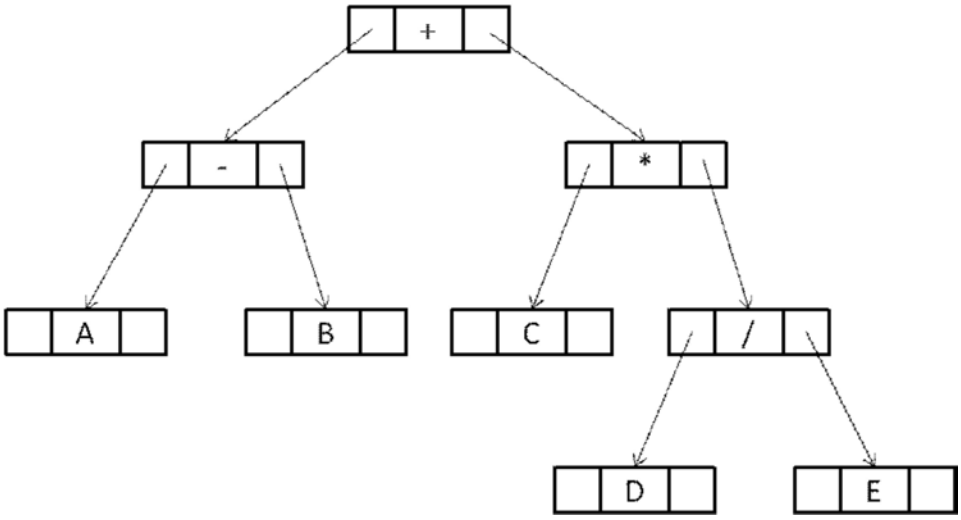


Fig. 8.7 : Link representation of binary tree

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

1. **Preorder Traversal :** In this traversal process, it 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 preorder(node)

```
if (node==null) then return
else
        visit(node)
        preorder(node->leftchild)
        preorder(node->rightchild)
```

The preorder traversal of the tree shown below is as follows.

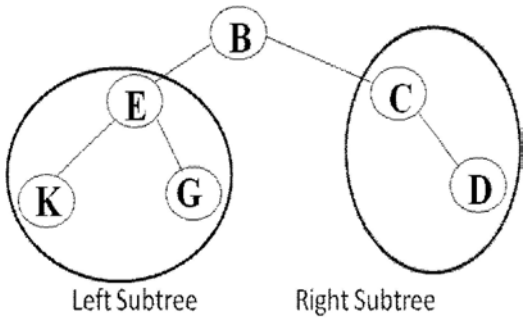


Fig. 8.8 : A binary tree

In preorder traversal of the tree process B (root of the tree), traverse the left subtree and traverse the right subtree. However the preorder traversal of left subtree process 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.

Inorder Traversal : The traversal keeps moving left in the binary tree until one can move no further, 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 inorder(node)

```
if (node==null) then return
else
        inorder(node->leftchild)
        visit(node)
        inorder(node->rightchild)
```

The inorder traversal of the tree (fig 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.

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 postorder(node)

```
if (node==null) then return
else
        inorder(node->leftchild)
        inorder(node->rightchild)
        visit(node)
```

The postorder traversal of the tree (fig 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 the tree. K G E D C B is the postorder traversal of the tree.

CHECK YOUR PROGRESS

Q.1. Answer the following:

a) What is the maximum height of a binary tree having n nodes?

b) What is the maximum number of node in a binary tree at level i?

c) What is the minimum number ofnode in a binary tree at level i?

d) What is the minimum height of a binary tree having n node?

d) 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 performed 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

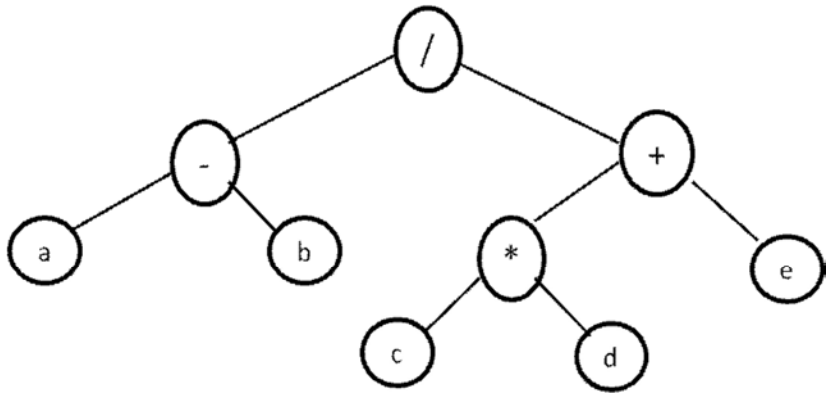


Fig. 8.9 : Expression tree

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 rules of operator. This conversion uses stack. The steps for converting an infix expression to a postfix expression is as follows

Algorithm : Infix_to_postfix (E, P)

// Here E is the infix expression to be converted and P is the resulting postfix expression

1. PUSH '(' into the STACK and add ')' to the end of E
2. Scan E from left to right and repeat step 3 to 6 for each element of E until the STACK is not EMPTY
3. If an operand is encountered, add it to P
4. If a left parenthesis is encountered, PUSH it onto STACK
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
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.
7. STOP

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 : postfix_evaluation (P)

// Here P is the postfix expression to be evaluated

1. Scan E from left to right and repeat step 3 to 6 for each element of E until end of the postfix expression
2. If an operand is encountered, PUSH it to STACK
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)
4. STOP

Consider the following postfix expression

7 6 4 2 * + / 5 8 2 / * +

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 every sub-tree is a binary search tree.

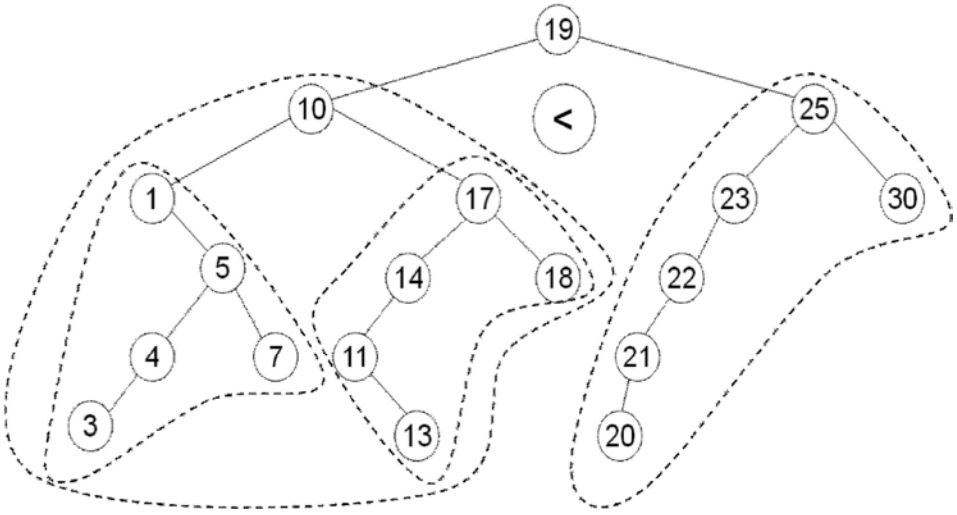


Fig. 8.10 : Binary Search Tree

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(n logn) steps.

Program : Creating a Binary Search Tree

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
    {
        struct tnode *lchild;
        int data;
        struct tnode *rchild;
    };
```

A complete C program to create a binary search tree follows:

```
#include <stdio.h>
#include <stdlib.h>
struct tnode
    {
        struct tnode *lchild;
        int data;
        struct tnode *rchild;
    };
struct tnode *insert(struct tnode *p, int val)
    {
        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 = (struct tnode*)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 = (struct tnode*)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);
}
```

Explanation :

1. 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.
2. 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).
3. 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:
  1. The number of nodes that the tree to be created should have
  2. 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 :

Input:

1. The number of nodes that the created tree should have = 5
2. 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:

```
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:

```
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

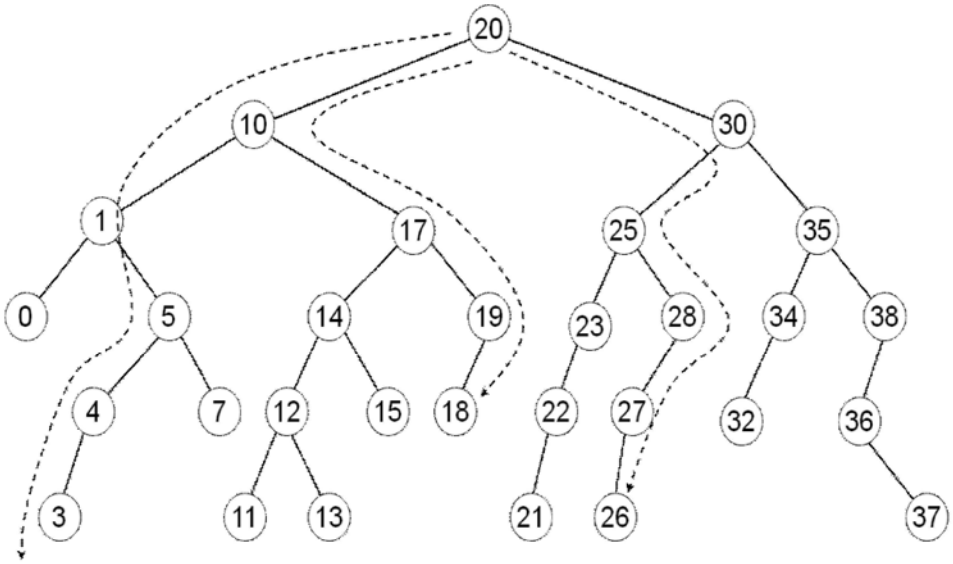


Fig. 8.11 : Searching in BST

A function to search for a given data value in a binary search tree

```
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 return the node on success, otherwise 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 bellow

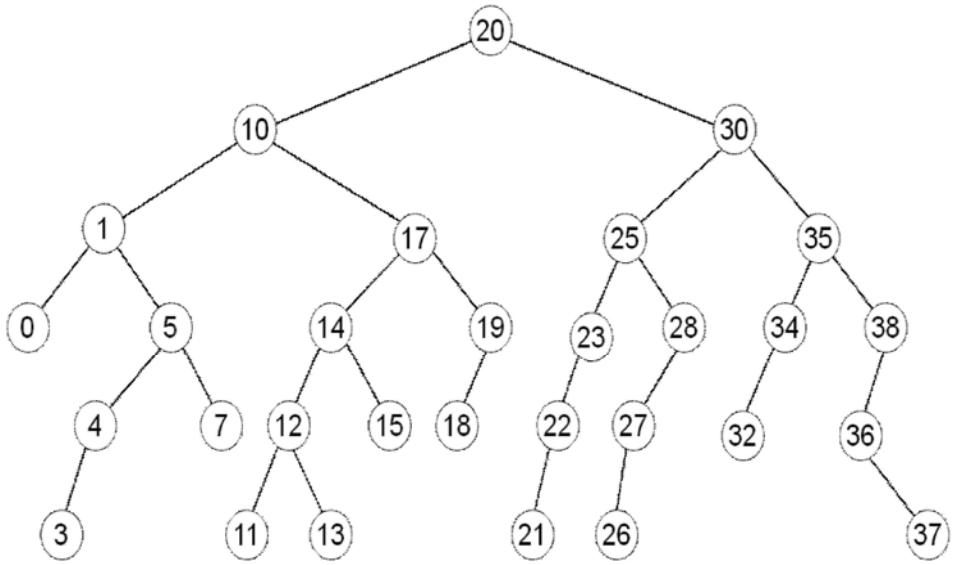


Fig. 8.12 : A binary tree

If we want to delete 13 from the above tree (fig 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 bellow.

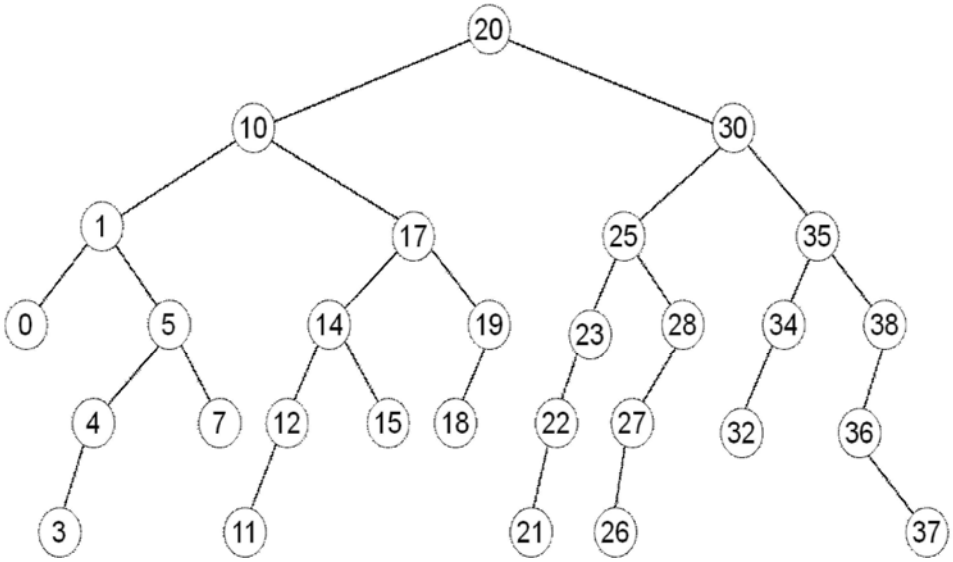


Fig. 8.13 : BST after deleting the node 13

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 Fig 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.

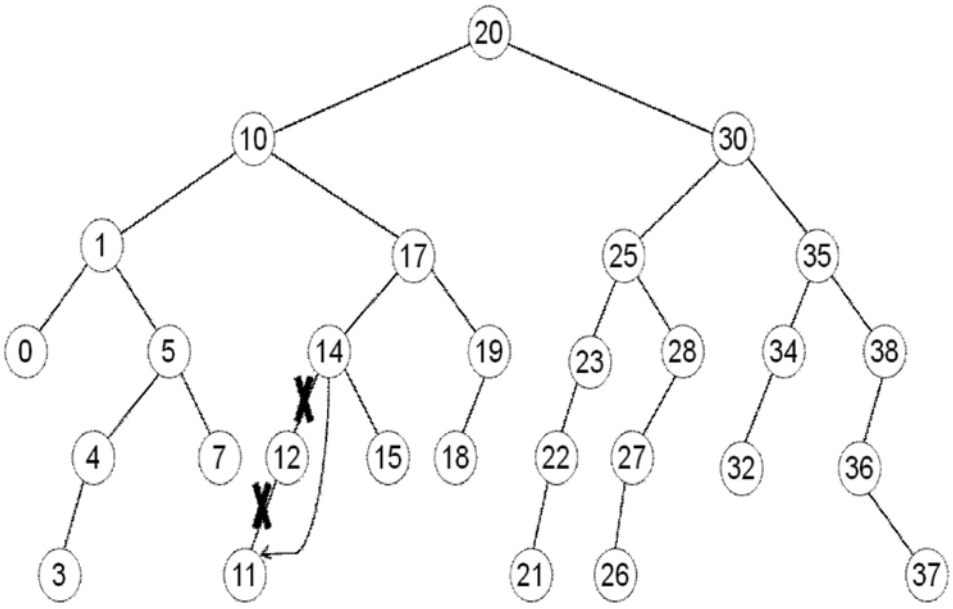


Fig. 8.14 : intermediate step for delete the node 12

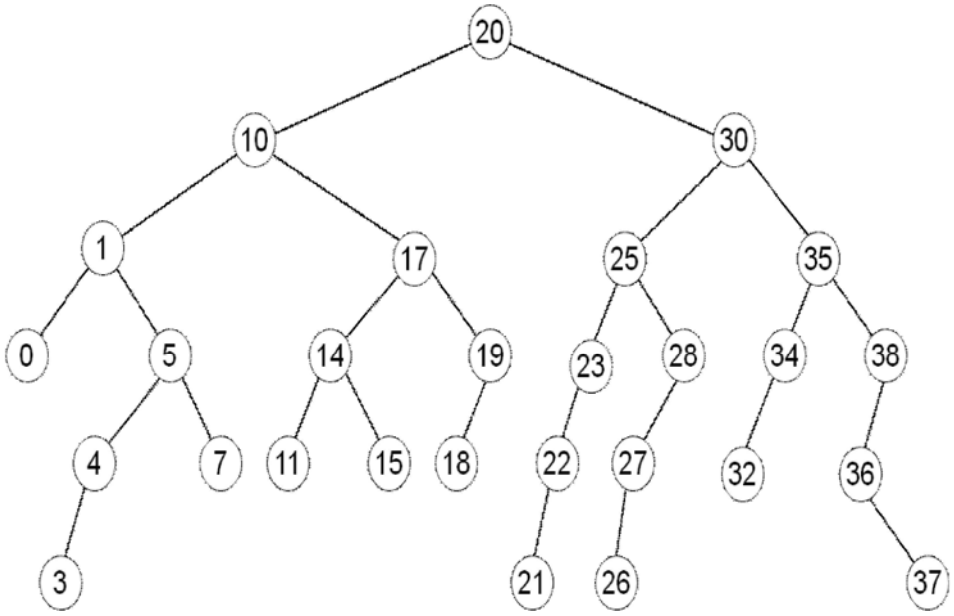


Fig. 8.15 : BST after deleting the node 12

Similarly the tree after deleting node 28 from the tree in fig 8.15 is shown bellow

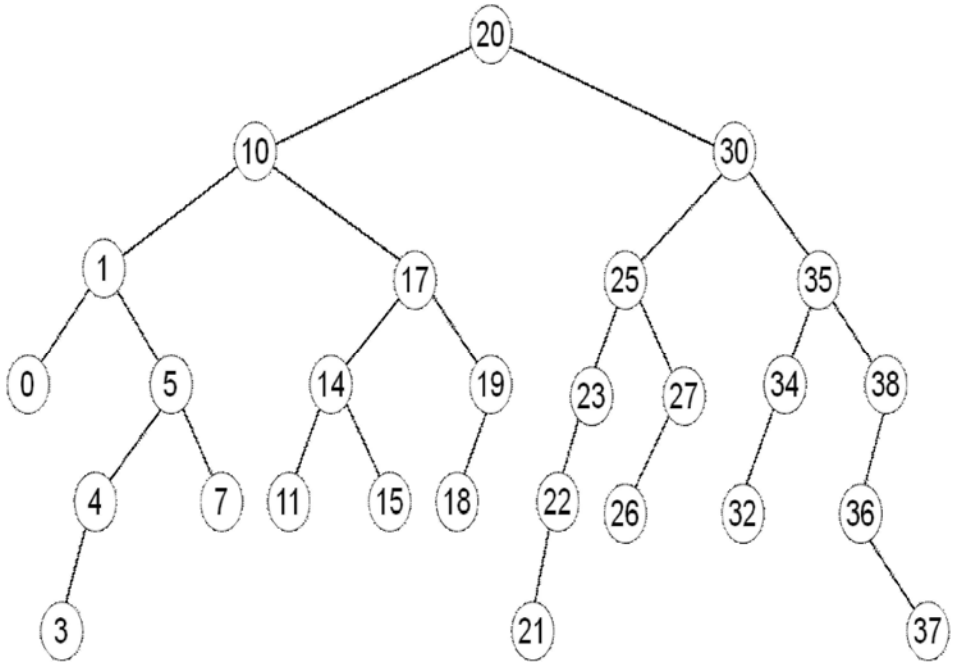


Fig. 8.16 : BST after deleting the node 28

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 fig 8.16, where 15 is the inorder successor of 14, first we need to delete 15 and then replace 14 by 15.

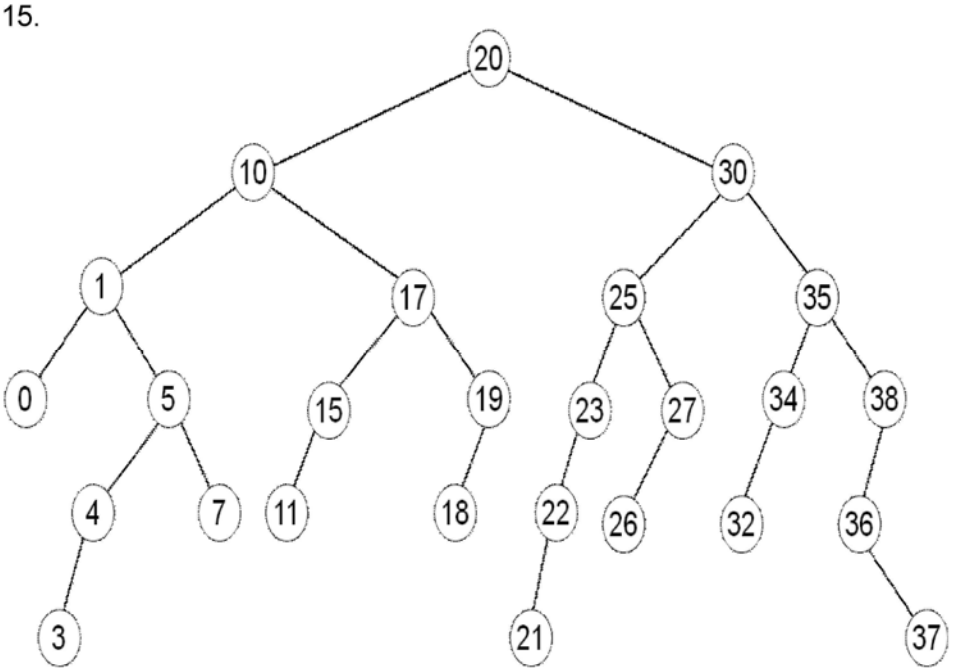


Fig. 8.17 : BST after deleting the node 14

If we want to delete 35 from the tree in fig 8.17, where 36 is the inorder successor of 35, first we need to delete 36 and then replace 35 by 36.

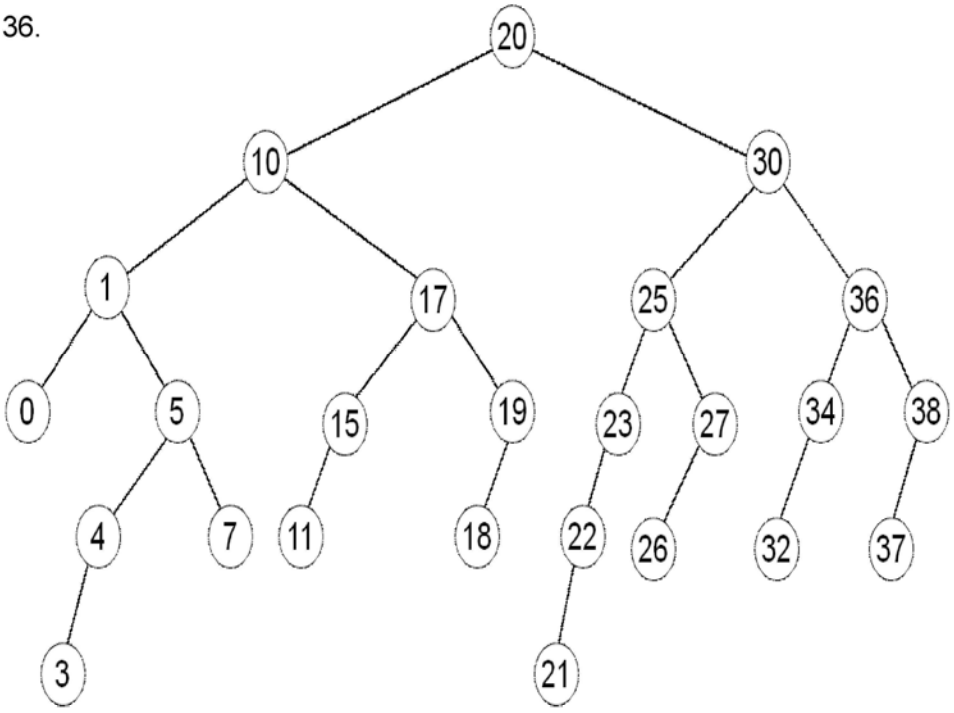
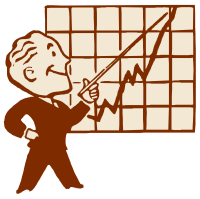


Fig. 8.18 : BST after deleting the node 35



CHECK YOUR PROGRESS

Q.2. Sate 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 READINGS

- T. H. Cormen, C. E. Leiserson, R.L.Rivest, and C. Stein, *Introduction to Algorithms*, Second Edition, Prentice Hall of India Pvt. Ltd, 2006.
- Ellis Horowitz, Sartaj Sahni and Sanguthevar Rajasekaran, *Fundamental of data structure in C*, Second Edition, Universities Press, 2009.
- Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman, *The Design and Analysis of Computer Algorithms*, Pearson Education, 1999.
- Ellis Horowitz, Sartaj Sahni and Sanguthevar Rajasekaran, *Computer Algorithms/ C++*, Second Edition, Universities Press, 2007.

8.11 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1 : a) n–1, b) 2^i, c) 1, d) $\log_2^n$, e) 2^i.

Ans. to Q. No. 2 : 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 : GRAPH

UNIT STRUCTURE

9.1 LEARNING OBJECTIVES

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

- Define Graph and terms associated with it.
- Representation of graph
- Explain different traversal technique on graphs.

9.2 INTRODUCTION

Like tree, graphs are also nonlinear data structure. Tree has some specific structure whereas graph does not have a specific structure. It varies 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.

In this unit we are going to discuss the representation of graph in memory and present various traversal technique of graph like depth first search and breadth first search.

9.3 BASIC TERMINOLOGY 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.

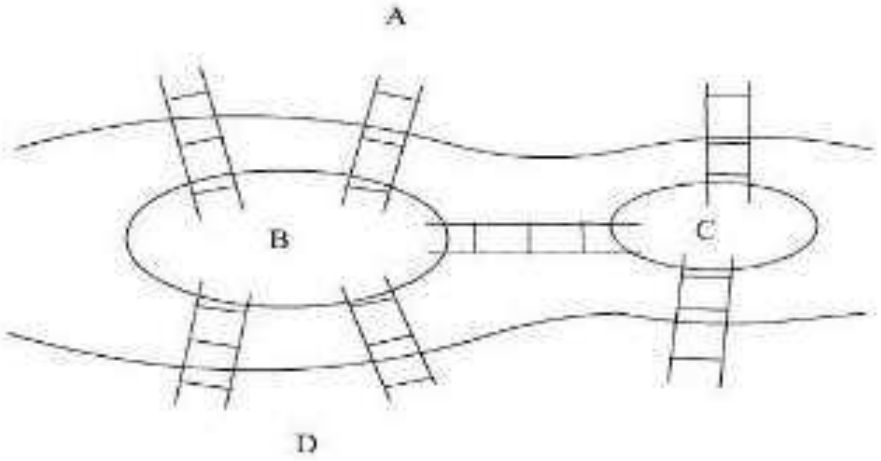


Fig. 9.1 : Konigsberg Bridge Problem

Definition: A graph G= (V, E) consists of

- a set V={v_1 , v_2 .....,v_n } of n >1 vertices and
- a set of E={e_1 ,e_2 ,.....,e_m } of m>0 edges
- such that each edge e_k is corresponds to an unordered pair of vertices (v_i ,v_j) 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.

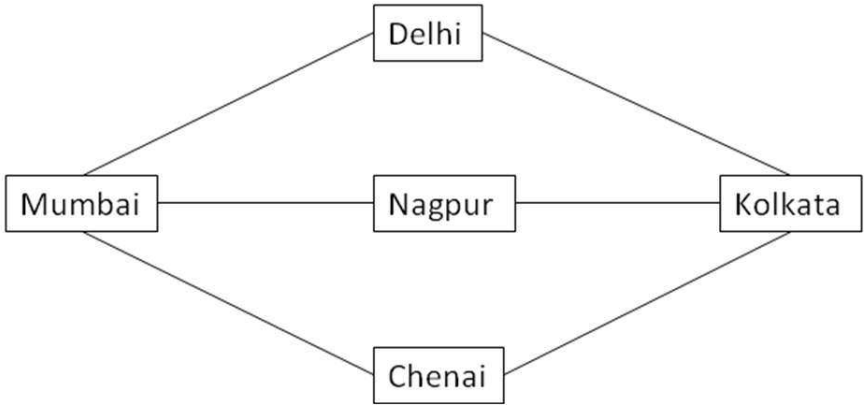


Fig. 9.2 : A road network

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 terminology relating to a graph is given below :

Self Loop: An Edge having same vertices as its end vertices is called self loop. In fig 9.3 e_3 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 fig 9.3 e_6 and e_7 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 fig 9.3 vertex A and B are adjacent but A and D are not adjacent.

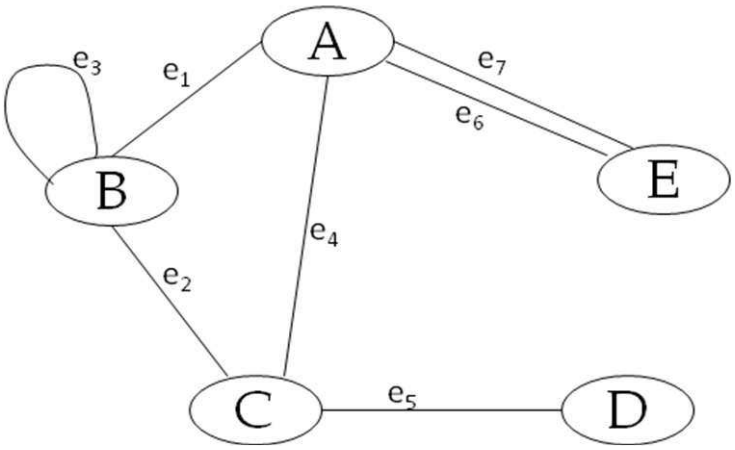


Fig. 9.3 : Example of Graph (self loop parallel edges)

Incidence: if an vertex v_i is an end vertex of an edge e_k , we say vertex v_i is incident on e_k and e_k is incident on v_i.

Simple Graph: A graph with no self loop and parallel edges is called a simple graph. For example Fig 9.2 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 fig 9.3 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 V_1, V_2 … V_{k+1} such that V_1 = x and V_{k+1}=y and V_i is adjacent to V_{i+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 the fig 9.3 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 in Fig 9.4 is not connected but the graph in fig 9.3 is connected.

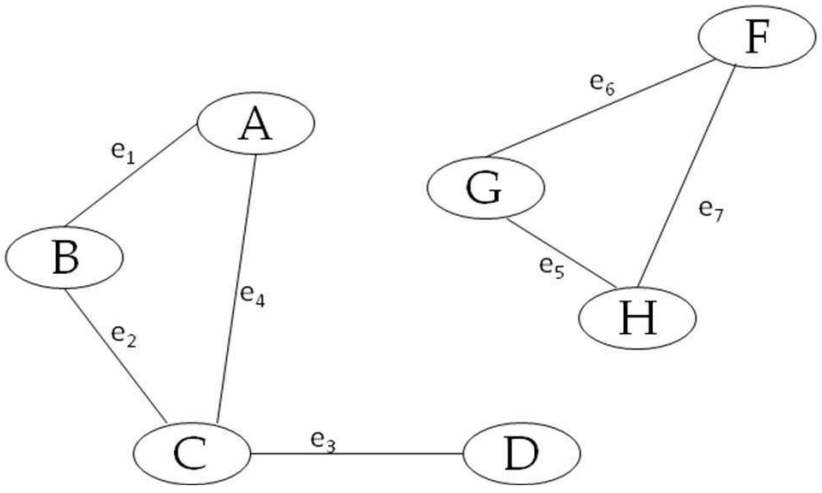


Fig. 9.4 : Example of disconnected graph

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. Fig 9.5 is an example of weighted graph.

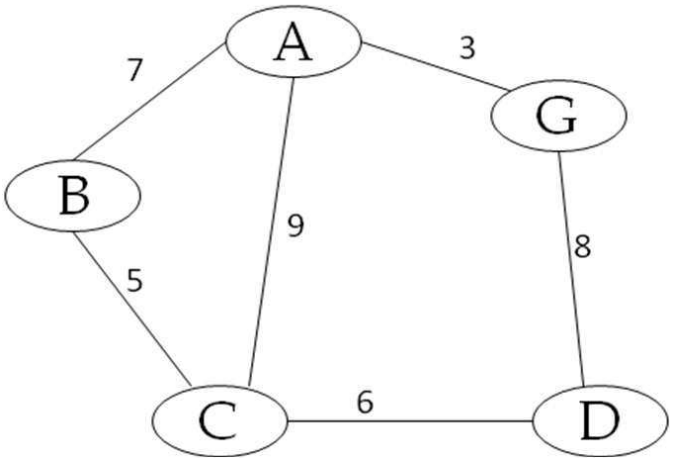


Fig. 9.5 : A weighted Graph

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). Fig 9.6 is an example of directed graph.

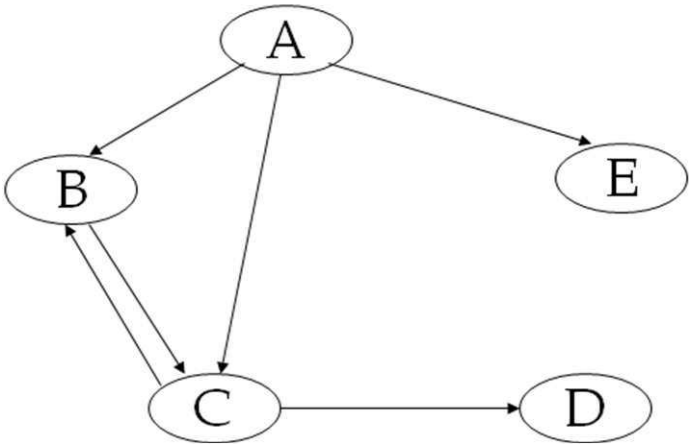


Fig. 9.6 : Directed Graph

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 fig 9.6 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 fig 9.6 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.

9.4 REPRESENTATION OF GRAPHS

There are several different ways to represent a graph in computer memory. Two main representations are:

- Adjacency Matrix
- Adjacency list.

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 v_1, v_2, …. v_n. Then the adjacency matrix A = (a_{ij}) 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 following graph can be represented as follows :

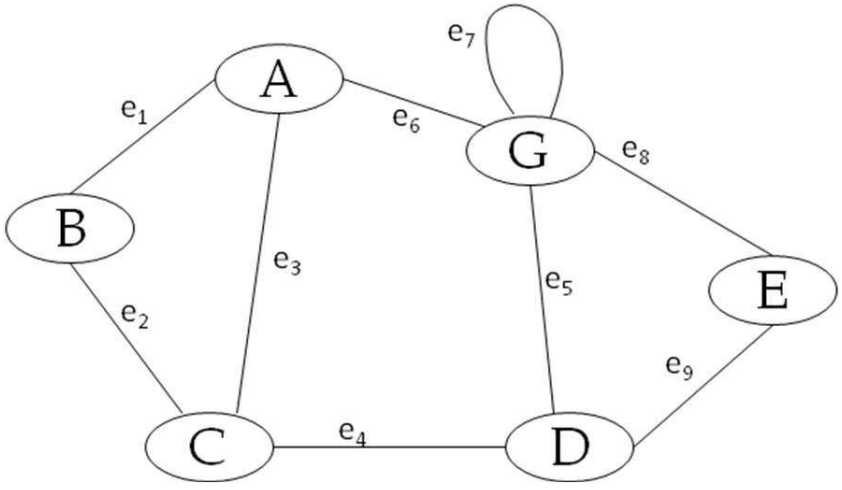


Fig. 9.7 : Graph Example

$$A = \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \\ G \end{array} \begin{array}{c} \begin{array}{cccccc} A & B & C & D & E & G \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 \end{pmatrix} \end{array}$$

The directed graph in fig 9.6 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 v_1, v_2, …. v_n. 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 = (w_{ij}) 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 fig 9.5 can be represented as follows

$$A = \begin{array}{c} \\ A \\ B \\ C \\ D \\ G \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & G \end{array} \\ \begin{pmatrix} 0 & 7 & 9 & 0 & 3 \\ 7 & 0 & 5 & 0 & 0 \\ 9 & 5 & 0 & 6 & 0 \\ 0 & 0 & 6 & 0 & 8 \\ 3 & 0 & 0 & 8 & 0 \end{pmatrix} \end{array}$$

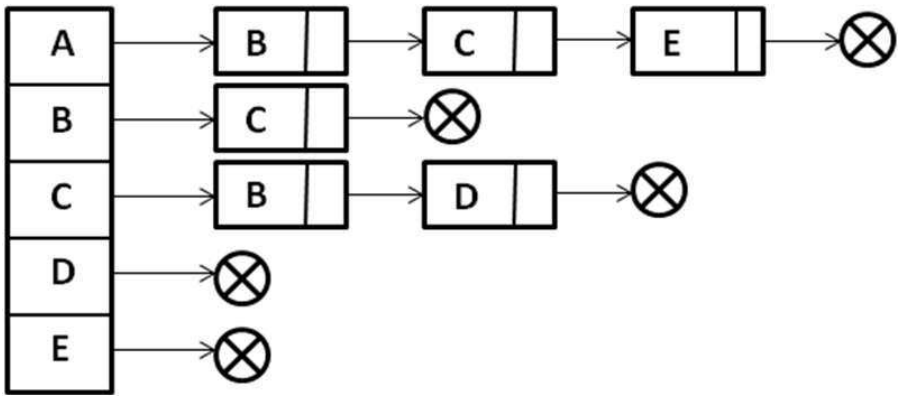
Although an adjacency matrix is a simple way to represent a graph it needs n^2 bit to represent a graph with n vertices. In case of undirected graph storing only upper or lower triangle of the matrix, this space can be

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.

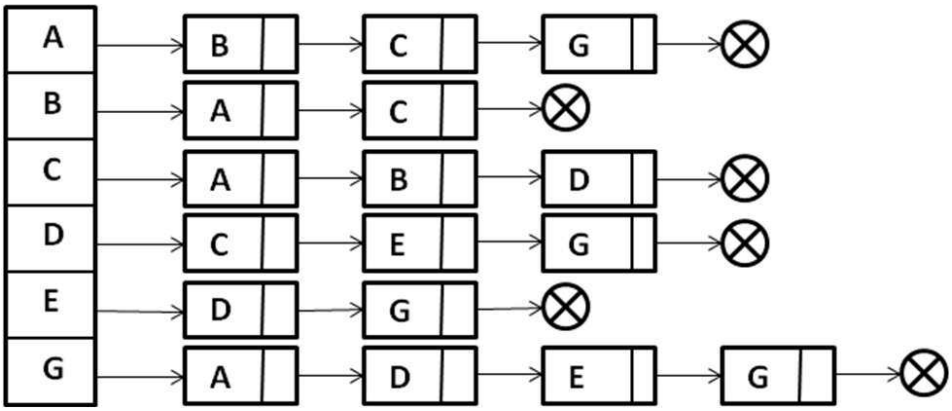
1. **Adjacency List Representation of Graph:** Suppose G(V, E) is simple graph (directed / undirected) with n vertices and e edges, the adjacency list have n head nodes corresponding to the n vertices of graph G, each of which point to a singly link list of nodes adjacent to the vertex representing to the head node.

In contrast to adjacency matrix representations, graph algorithm which make 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 fig 9.6 can be represented in adjacency list representation as follows



For example the undirected graph in fig 9.7 can be represented in adjacency list representation as follows



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 as adjacent vertices.

d) For an undirected graph parallel edges can be represented in the adjacency matrix representation of graph.

9.5 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

1. **Breadth First Traversal:** The general idea behind breadth first traversal is that, 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 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 .

A complete C program for breadth-first traversal of a graph appears next. 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.

```
#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);
    }
```

Example :

Input and 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

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 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 .

A complete C program for depth-first traversal of a graph follows. 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

```
#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);
        }
```

Explanation :

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.2. Fill in the banks :

a) The most commonly used methods for traversing a graph are _____ and _____.

b) In depth first traversal we use _____. (Stack/ Queue)

9.6 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.

9.7 FURTHER READINGS

- T. H. Cormen, C. E. Leiserson, R.L.Rivest, and C. Stein, *Introduction to Algorithms*, Second Edition, Prentice Hall of India Pvt. Ltd, 2006.
- Ellis Horowitz, Sartaj Sahni and Sanguthevar Rajasekaran, *Fundamental of data structure in C*, Second Edition, Universities Press, 2009.
- Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman, *The Design and Analysis of Computer Algorithms*, Pearson Education, 1999.
- Ellis Horowitz, Sartaj Sahni and Sanguthevar Rajasekaran, *Computer Algorithms/ C++*, Second Edition, Universities Press, 2007.

9.8 ANSWERS TO CHECK YOUR PROGRESS

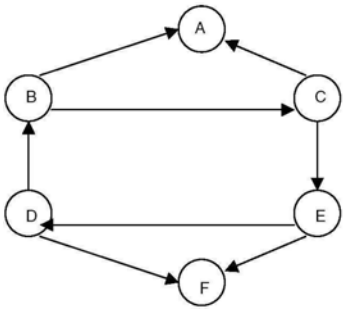
Ans. to Q. No. 1 : a) False, b) True, c) False, d) False

Ans. to Q. No. 2 : a) Depth first traversal and breadth first traversal,
b) Stack

9.9 MODEL QUESTIONS

Q.1. Define : 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.