GCA S2 03

Data Structure through C Language

SEMESTER - II

BACHELOR OF COMPUTER APPLICATION

BLOCK - 2



KRISHNA KANTA HANDIQUI STATE OPEN UNIVERSITY

Subject Experts

Prof. Anjana Kakati Mahanta,	Gauhati University
Prof. (Retd.) Pranhari Talukdar,	Gauhati University
Dr. Jyotiprokash Goswami,	Assam Engineering College

Course Coordinator

Dr. Sanjib Kr. Kalita,	KKHSOU
Dr. Tapashi Kashyap Das,	KKHSOU
Sruti Sruba Bharali,	KKHSOU

SLM Preparation Team

Units	Contributor
8, 10	Dr. Tapashi Kashyap Das, KKHSOU
9,12,13,14	Dr. Khirod Sarma, NERIM
11	Arabinda Saikia, KKHSOU

Editorial Team

Content	Dr. Jyotiprokash Goswami, Assam Engineering College
Language	Prof. Robin Goswami, Retd.Professor, Cotton College
Structure, Format & Graphics:	Dr. Tapashi Kashyap Das, Sruti Sruba Bharali, KKHSOU

Jan, 2018

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

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

Headquarters : Patgaon, Rani Gate, Guwahati - 781017
City Office : Housefed Complex, Dispur, Guwahati-781 006; Web: www.kkhsou.in

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

BACHELOR OF COMPUTER APPLICATION

DATA STRUCTURE THROUGH C LANGUAGE

DETAILED SYLLABUS

Block-2

Unit-8 : Stacks **Pages** 1-18

Definitions, LIFO, Representation using array and linked list structure, Applications of stack.

Unit-9 : Queues **Pages** 19-33

Definitions, FIFO, Representation using array, linked representation of queues, application of queue.

Unit-10 : Searching **Pages** 34-48

Linear and binary search algorithms, performance and complexity using big 'O' notation

Unit-11 : Sorting **Pages** 49-70

Sorting algorithms (Complexity, advantages and disadvantage, implementation), bubble sort, insertion sort, selection sort, quick sort

Unit-12 : Trees **Pages** 71-90

Definition, Types of trees, implementation: Binary Tree, Tree traversal algorithms (in-order, preorder, post-order), infix, postfix, prefix notations

Unit-13 : Binary Search Tree **Pages** 91-109

Searching in Binary Search Tree, insertion and deletion in Binary Search Trees, Height of Tree, B-Tree

Unit-14 : Graphs **Pages** 110-127

Introduction to Graphs, Edge, Vertex, Depth first search (DFS) and Breadth first search (BFS) technique

BLOCK INTRODUCTION

This is the *Second Block* of the course '***Data Structure Through C Language***'. After completing this block, you will be able to use different data structures for diffrent data efficiently.

This block comprises of the following seven units:

Unit - 8 introduces you to stack data structure. Representation of stack using array and linked list are included in this unit.

Unit - 9 introduces you to queue data structure. Representation of queues using array and linked list are included in this unit.

Unit- 10 concentrates on the different types of searching. Linear and Binary search techniques are included in this unit.

Unit - 11 concentrates on the different types of sorting. Sorting algorithms like bubble, insertion, quick sort and selection sort are covered in this unit.

Unit - 12 introduces us to the concepts of trees. Binary trees and the different types of tree traversals are includedi in this unit.

Unit - 13 introduces us to binary search trees. Insertion and deletion of binary search trees are also included in this unit.

Unit - 14 introduces us to the concepts of graphs. Graph traversal technique like depth first and breadth first are included in this unit.

The structure of Block 2 is as follows:

UNIT 8 : Stacks

UNIT 9 : Queues

UNIT 10 : Searching

UNIT 11 : Sorting

UNIT 12 : Trees

UNIT 13 : Binary Search Tree

UNIT 14 : Graphs

UNIT 8 : STACKS

UNIT STRUCTURE

8.1 Learning Objectives
8.2 Introduction
8.3 Stacks
8.4 Operations on Stack
8.5 Stack Implementation
    8.5.1 Stack implemented as an Array
    8.5.2 Stack implemented as Linked List
8.6 Applications of Stack
8.7 Let Us Sum Up
8.8 Answers to Check Your Progress
8.9 Further Reading
8.10 Model Questions

8.1 LEARNING OBJECTIVES

After going through this unit, you will able to :

- define a Stack
- learn about LIFO structure
- describe Push and Pop operation
- represent Stack using Array
- represent Stack using Linked list
- learn about the applications of Stack.

8.2 INTRODUCTION

In the earlier units we have seen the primitive data structures that are available in C. We have also looked through arrays and linked lists. While studying linked list we have seen that insertion and deletion of elements is possible at any place in the list i.e., at the beginning, at the end, or at the middle.

There are certain frequent situations in computer science when one

wants to restrict insertions and deletions so that they can take place only at the beginning or at the end of the list, not in the middle. Two of the data structures that are useful in such situations are ***stacks*** and ***queues***.

In this unit we will discuss the stack data structure.

8.3 STACKS

A stack is an **abstract data type**(ADT) commonly used in most programming language. It is named stack as it operates like a real-world stack, for example, pile of plates, deck of cards, stack of coins etc.

In general, a stack is a pile of objects, typically one that is neatly arranged. We can also define stack as a storage that handles data so the next item to be retrieved is the item most recently stored. Figure 8.1 shows two everyday examples of such a structure: *a stack of plates* and *a stack of coins* on a table.

A real-world stack allows operations at one end only. For example, we can place or remove a plate from top of the pile only. When the plates are being stacked, they are kept one on top the each other. It does not make much sense to put each plate on the bottom of the pile, as that would be far more work, and would accomplish nothing over stacking them on top of each other. Similarily, when a plate is taken, it is usually taken from the top of the stack. This means, in particular, that the last item to be added to a stack is the first item to be removed. This feature makes it **LIFO** data structure. LIFO stands for **Last-In First-Out**. Here, the element which is placed (inserted) last, is accessed first. Although the stack may seem to be a very restricted type of data structure, it has many important applications in computer science.

Stacks are linear data structure that represents a set of homogeneous (similar kind) collection of data elements. A stack allows us to perform the insertion and deletion of data elements from one end of the stack, called the ***top*** of the stack. This means that data can be added or removed from only the ***top***.

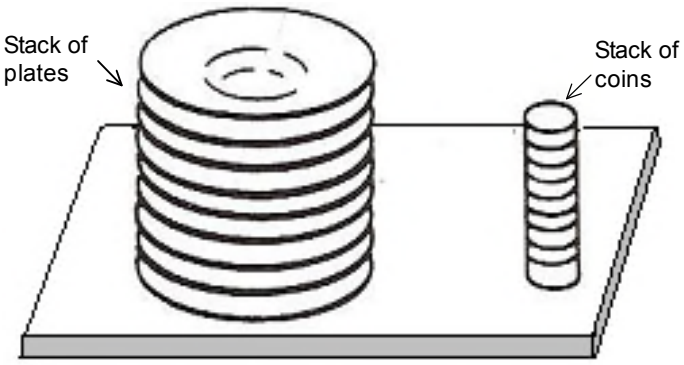


Fig. 8.1: Real life examples of Stack

8.4 OPERATIONS ON STACK

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

- **Push** means inserting (storing) an element on the stack.
- **Pop** means removing (accessing) an element from the stack.

Apart from these main functions, some basic operations associated with stacks are:

Creating a stack

Checking whether a stack is empty or full

Initializing a stack

Accessing the top element

Displaying elements in a stack.

Push operation : The process of inserting a new element onto stack is known as *push* operation. Suppose, Top is a pointer to the top element in a stack. After every push operation, the value of Top is incremented by one. When the stack is full, the status of the stack is full and this condition is called ***stack overflow***. In such a case, no further element can be inserted. The following picture Fig.(8.2) shows the push operation.

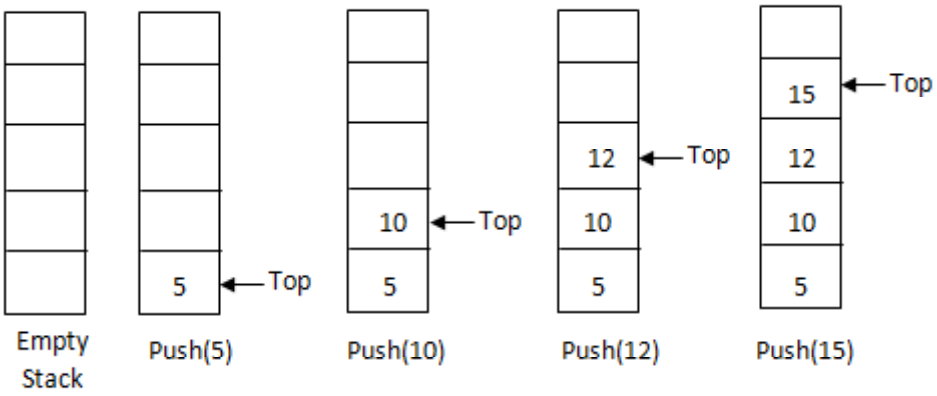


Fig. 8.2 : *Push* operation with stack

Push operation can be accomplished in the following steps:

Step 1 - Check if stack is full.

Step 2 - If stack is full, give an error message and exit.

Step 3 - If stack is not full, increment top to point next empty location.

Step 4 - Add data element to the stack location, where top is pointing.

Step 5 - return success.

The program segment for Push operation will be as follows:

```
void push(int data)
 {
      if(!isFull( ))     // if the stack is not full
     {
          top = top + 1;
         stack[top] = data;
      }
      else
      {
           printf("Stack is full, unable to insert element.\n");
      }
}
```

Pop operation : The procedure of removing element from the top of the stack is called *pop* operation. After every pop operation, the value of Top is decremented by one. If there is no element in the stack, then the stack is empty or ***stack underflow***. In such case, pop operation cannot be applied. Figure 8.3 illustrates the pop operation.

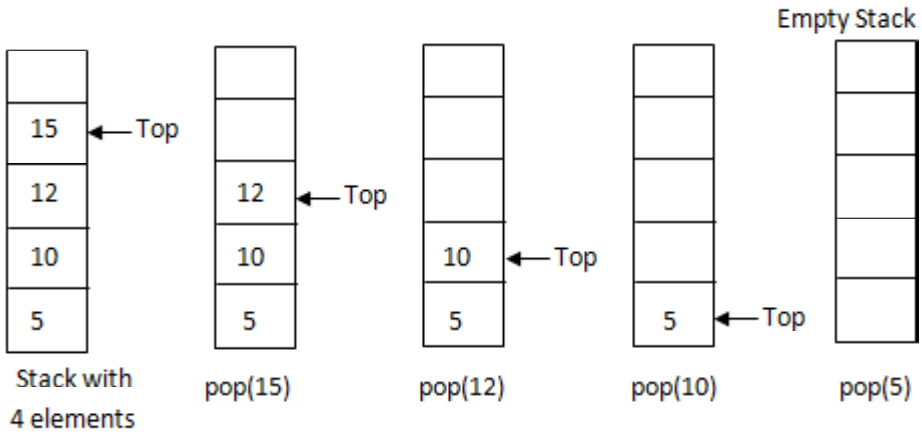


Fig. 8.3 : *Pop* operation with stack

In array implementation of pop operation, data element is not actually removed, instead Top is decremented to a lower position in stack to point to next value. But in linked-list implementation, pop operation actually removes data element and deallocates memory space.

Pop operation can be accomplished in the following steps :

Step 1 - Check if stack is empty

Step 2 - If stack is empty, display an error message and exit

Step 3 - If stack is not empty, access the data element at which top is pointing

Step 4 - Decrease the value of top by 1

Step 5 - return success

The program segment for Pop operation will be as follows:

```
int pop(int data)
{
    if(!isempty( ))
```

```
    {
        data = stack[top];
        top = top - 1;
        return data;
     }
    else
    {
        printf("Stack is empty\n");
    }
}
```

8.5 STACK IMPLEMENTATION

There are two methods of stack implementation. They are:

- *Static implementation*
- *Dynamic implementation*

Static implementation can be achieved by using arrays. Though it is a simple method, it has a few limitations. Once a size of an array is declared, its size cannot be modified during program execution. If we store less element than declared, 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.

The limitations of static implementation can be removed using dynamic implementation. ***Dynamic implementation*** can be achieved using linked-list. With the use of pointers in dynamic implementation, there is no restriction on the number of elements. The stack may be expandable. The memory is efficiently utilized with pointers. Memory is allocated only after element is inserted to the stack.

8.5.1 Stack Implemented as an Array

Here, we are going to implement stack using arrays which makes it a fixed size stack implementation. When implemented this way, the data is simply stored in the array. *Top* is an integer variable,

which contains the array index for the top 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. By convention, an empty stack is indicated by setting Top to be equal to -1.

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.

/*Program 8.1: Program of stack using array */

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define SIZE 10
void push();   //function prototype for push operation
void pop();    //function prototype for pop operation
void display();  //function prototype for displaying the stack
int top = -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()    //function definition of 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()  //function definition for 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]);
    }
}
```



CHECK YOUR PROGRESS

Q.1. 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.2. 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.

8.5.2 Stack Implemented as Link List

To solve the problems discussed in the above section, a stack can be implemented using pointers, as a form of a link list. When a stack is implemented this way, each node contains a data field for the information and a pointer to the next node on the list. Top is a pointer to the top of the list. When top is NULL, the stack is empty.

Implementing stacks as link lists provides a solution to the problem of dynamically growing stacks as a linked list. The stack can grow or shrink as the program demands it. 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.

To implement stack through linked list, we will take the structure as :

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

The ***data*** fields of the node hold the element of the stack and the ***next*** field hold pointers to the next element in the stack. Figure 8.4 illustrates the linked representation of a stack.

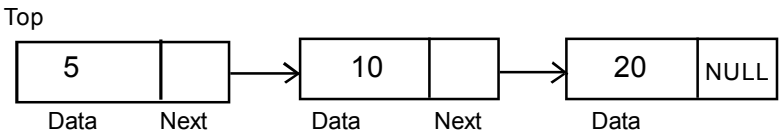


Fig. 8.4 : Link list representation of a stack

Here, first we check for empty condition and then we pop the first element of stack. Now top will point the second node of link list which will become the first node of link list.

*/*Program 8.2:Program of stack using linked list*

```
#include<stdio.h>
#include<conio.h>
#include<malloc.h>
#include<stdlib.h>
void push();     //push() function declaration
void pop();      //pop() function declaration
void display();  //display() function declaration
struct node
{
    int data;
    struct node *next;
};
struct node *top=NULL;
void main()
{
    int choice;
    clrscr();
    while(1)
    {
        printf("\n1.Push");
        printf("\n2.Pop");
        printf("\n3.Display");
        printf("\n4.Quit");
        printf("\nEnter your choice:");
        scanf("%d",&choice);
        switch(choice)
       {
        case 1: push();
                break;
        case 2:     pop();
                break;
        case 3: display();
                break;
```

```
            case 4: exit(1);
                    default:printf("Invalid Choice\n");
            }//end of switch
        } //end of while loop
}//end of main fuction
void push()
{
    struct node *ptr;
    int item;
    ptr=(struct node *)malloc(sizeof(struct node));
    printf("Enter a value to be pushed onto the stack: ");
    scanf("%d",&item);
    ptr->data=item;
    ptr->next=top;
    top=ptr;
}//end of push fuction definition
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
```

8.6 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 loaded 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 LIFO 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.

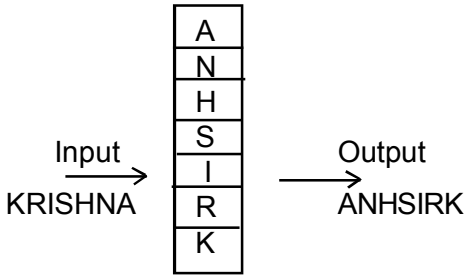


Fig. 8.5 : Stack

// Program 8.3: 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***.

Evaluation of arithmetic expressions : Usually, arithmetic expressions are written in *infix* notation, e.g., A+B*C

An expression can as well be written in *postfix* notation (also called *reverse polish notation*): A+B becomes AB+

A*C becomes AC*

A+B*C becomes ABC*+

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

Evaluating expressions : Given an expression in postfix notation. Using a stack they can be evaluated as follows :

- Scan the expression from left to right
- When a value(operand) is encountered, push it on the stack
- When an operator is encountered, the first and second element from the stack are popped and the operator is applied
- The result is pushed on the stack.



CHECK YOUR PROGRESS

Q.3. 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) d) 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

Q.4. Distinguish between static and dynamic implementation of stack.



8.7 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 the least reachable elements in a stack are respectively known as the *top* and *bottom*.
- The insertion and deletion operations are carried out at one end. 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*.
- If a pop operation is performed on an empty stack it is called *stack undeflow*.
- An operation to insert an element more than the capacity of stack is known as *stack overflow*.
- Static implementation can be implemented using arrays. However, it is a very simple method but it has some 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.



8.8 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1 : Stack will contain only one item and the item is A

Ans. to Q. No. 2 : i) False; ii) True; iii) True

Ans. to Q. No. 3 : 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

Ans. to Q. No. 4 : (See text content)



8.9 FURTHER READING

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

2. Kamthane, A. (2009). *Programming with ANSI and Turbo C*. Pearson Education India.

3. Lipschutz, S. (1987). *Schaum's Outline of Data Structure*. McGraw-Hill, Inc..

8.10 MODEL QUESTIONS



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

Q.2. What is *stack overflow* and *stack underflow*?

Q.3. What is linked implementation of stack?

Q.4. Write a C program to implement stack with array. Perform *push* and *pop* operation.

Q.5. Write a program to perform following operations:

a) push

b) pop

c) display all elements.

Q.6. Write short notes on static and dynamic implementation of stack.

Q.7. Describe two applications of stack.

*****

UNIT 9: QUEUES

UNIT STRUCTURE

9.1 Learning Objectives
9.2 Introduction
9.3 Definition of Queue
9.4 Implementation of Queue
9.4.1 Implementing Queue using Array
9.4.2 Implementing Queue using Link List
9.5 Applications of Queue
9.6 Let Us Sum Up
9.7 Answers to Check Your Progress
9.8 Further Reading
9.9 Model Questions

9.1 LEARNING OBJECTIVES

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

- define a queue
- learn about FIFO structure
- implement queue statically and dynamically using array and linked list
- learn various applications of queue.

9.2 INTRODUCTION

Queues in computer science are very similar to queues in real life. A queue in real life would be a line up at a railway ticket counter or a bank teller. The people are dealt with in the order that they arrived.



Fig. 9.1 : Queue of people waiting for bus in a bus stop

The theories of queue is common to all. In computer science this is called *FIFO (First-In First-Out)* queue. Items can only be removed from the queue in the order that they are inserted into it. Queues are very common in the everyday functions of computers. In a computer, there may be a queue of tasks waiting for execution, few others for printing, and some are for inputting the data and instructions through the keyboard. Unless the task turns and comes on front, it will not be executed. The most obvious is printing queues. When we send multiple print jobs to a printer, each printing job is inserted at the rear of the queue in the order it was sent. Each job is then printed in the order sent to the printer. Processes waiting to be executed by a CPU are usually in the form of a queue also.

This unit describes the queue data structure and its implementation.

9.3 DEFINITION OF QUEUE

A queue is an ordered homogeneous group of items in which the items are added at one end called the *rear end* (back) and are removed from the other end called *front end*. This is known as a ***First-In First-Out (FIFO)*** data structure. We usually draw queues horizontally. Basic operation of queue are:

- Creation of Queue
- Check for empty Queue
- Check for full Queue
- Delete element in Queue
- Diaplay Queue

The fundamental operations that can be performed on queue are **Insert** and **Delete**. When element is added in the rear end of the queue is called Insert operation. When data is removed from the front end of the queue, the operation is called delete. The insertion and deletion operation in a queue can be explained as follows:

Queue operation	Value of rear and front	0	1	2
Insert(5)	Front=0 Rear=0			5
Insert(10)	Front=0 Rear=1		5	10
Insert(20)	Front=0 Rear=2	5	10	20
Delete()	Front=1 Rear=2		10	20
Delete()	Front=2 Rear=2			20

Here is a queue of integers with 4 elements :

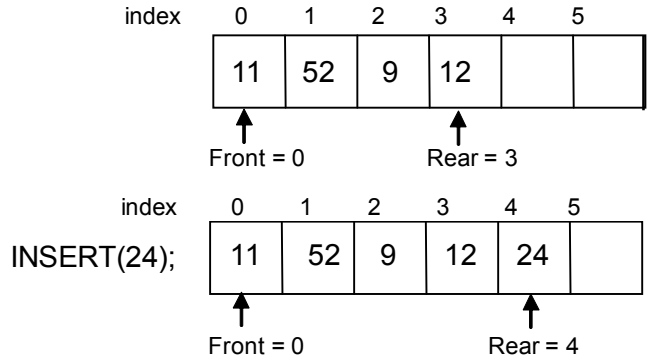


Fig. 9.2 : Addition of item at the rear end of the queue

In the figure 9.2, initially index of *Front* is 0 and index of *Rear* is 3 which contains element 12. While adding 24 we see that index of *front* remains the same i.e.,0 but index of *rear* becomes 4 and it contains element 24.

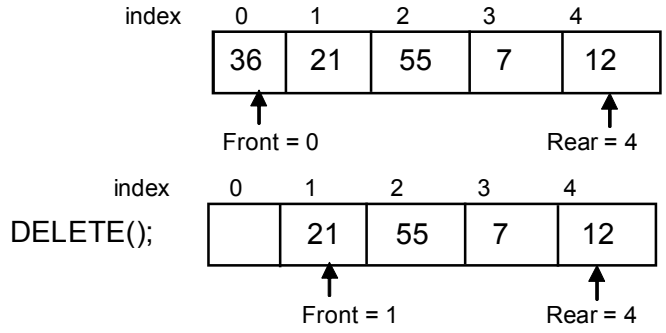


Fig. 9.3 : Deletion of item from the front end of the queue

If we remove an element from the queue, then it will be removed from the *Front* end. Here in the figure 9.3, we see that index of *Front* becomes 1 and it contains 21.

9.4 IMPLEMENTATION OF QUEUE

9.4.1 Implementing Queue with Array

Since a queue usually holds same type of elements, we could implement a queue with an array. As addition of element takes place at rear end and deletion takes place at front end, so we take two variables *rear* and *front* which keeps the status of last added item and first item of the queue. Two possibility may arise while adding and deleting elements. These are: ***Overflow*** and ***Underflow***.

Overflow arises when there is no place for adding elements in the queue. For that first we check the value of *rear* with size of array. *Underflow* arises when there is no element for deleting from the queue.

Let us simplify our array implementation of a queue by using an array of a fixed size, ***MAX***. We take ***qarray*** as the name of the queue of length 5.

Insertion of Element in a Queue : Here, everytime we insert a new element we increment rear by 1 and will check for the queue 'overflow' condition as we are using a fixed size array. We will take rear as -1 and front as 0 when the queue is empty.

Algorithm for inserting element in a queue

Step 1 : Increment the 'rear' by 1

Step 2 : Check for the 'queue overflow' condition. If queue not overflowing then go to step-3 else display "Queue Overflow" and quit.

Step 3 : Put the new element at the position pointed by 'rear'.

Function for insert operation in a queue :

```
int qarray[MAX];
int front = 0, rear = -1;
insert(int new_element)
{
    rear = rear + 1;
```

```
    if(rear < MAX)
        qarray[rear] = new_element;
    else
        printf("Queue overflow");
}
```

Program 9.1 : Program for inserting element in a queue of fixed length

```
#include<stdio.h>
#include<conio.h>
#define MAX 5
void main()
{
    int qarray[MAX];     //queue of maximum length MAX
    int rear;  //to hold the address of location where data is
         //inserted
    rear=0;
    clrscr();
    while(1)
    {
        if(rear>=MAX)  //checking overflow condition
        {
            printf("\nQueue is full");
            break;
        }
        else
        {
            printf("\nEnter elements in the queue:");
            scanf("%d",&qarray[rear]);
            rear++;
        }
    }
    printf("\nElements of the queue are:");
    rear=0;
```

```
    while(1)    {
            printf("\n%d",qarray[rear]);
            rear++;
            if(rear==MAX)
                break;
    }
    getch();
}
```

Output of the above program will be :

Enter elements in the queue : 2

Enter elements in the queue : 4

Enter elements in the queue : 5

Enter elements in the queue : 6

Enter elements in the queue : 8

Queue is full

Elements of the queue are : 2 4 5 6 8

Deletion of Element in a Queue : When an element leaves the queue the 'front' gets incremented by one. Every time before deletion we will check for the 'queue empty' condition. If the queue is not empty, put a NULL value at the position pointed by 'front' and increment the value of 'front' by 1.

Algorithm for deleting element in a queue :

Step 1 : If the queue is empty then display "Underflow" and quit else go to *Step 2*

Step 2 : Put a NULL value at the position pointed by 'front'. Increment the 'front' by 1.

Delete function for delete operation in a queue :

```
int qarray[MAX]; // Queue of maximum size MAX
int front = 0, rear = -1;       // when the queue is empty
int delete( )
{
    int del_val =0;             // holds the deleted value
    if(front > rear)
```

```
        printf("Queue empty");
    else
    {
        del_val = qarray[front];
        qarray[front] = NULL;  /* this represents that the
            element has been deleted */
        front = front + 1;
    }
    return(del_val);
}
```

9.4.2 Implementing Queue with Link List

We have already learnt the concept of link list in our previous unit. Link list does not have any restrictions on the number of elements it can hold. Memory space for the elements in a link list is allocated dynamically, hence link list can grow as long as there is enough memory available for dynamic allocation. Linked lists are made up of things called *nodes*. A node contains two main parts: ***data*** and a ***next,*** a pointer to the node after it. Queue can also be implemented through linked list. The structure of the node will be as:

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

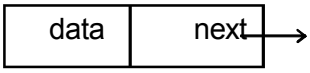


Fig. 9.4 : Structure of a node

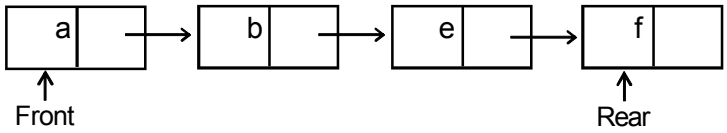


Fig. 9.5 : Link list implementation of a queue

Insert operation in Queue using link list : To indicate that the queue is empty, we assign NULL to front and rear pointer. If the element added is the first element, then both front and rear are made to point to the new node(or new_element). For adding element in a queue we add the element at the end of link list. Here front will point to the first node of link list and rear will point to the last node of link list. Thus, we have in terms of pointers:

```
temp -> data = new_element;
temp ->next = NULL;
if(front == NULL)        //if the queue is empty
    front = temp;
else
    rear->next = temp;
rear = temp;
```

In the above code, *temp* is a temporary node which has two parts: *data* and *next*. We have assigned *new_element* in to the *data* part of temp and NULL is assigned to *next* part. If the queue is empty then *new_element* will be the first element of the queue and front and rear will point that element. If the queue have already some elements then the *new_element* added will be the last element of the queue and rear will point that element.

Delete operation in Queue using link list : As queue is a *First-In_First_Out (FIFO)* structure, so the first node of the link list will be the first element of the queue. For removing an element in a queue, we delete the first element of the link list.

Thus, we have in terms of pointer:

```
if(front == NULL)
    printf("\nUnderflow Occurs");
else
{
    temp = front;
    printf("\nDeleted element is : %d", temp->data);
    front = front ->next;
    free(temp);
}
```

Removal of element from the list actually deletes the node to which front is pointing. After deletion of a node, front is made to point the next node that comes in the list, whereas rear continues to point to the last node in the list.

Program 9.2 : Insert and delete operation in a queue using linked list

```
//(Implementation of queue usning linked list)
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct list
{
    char data;
    struct list *next;
};
typedef struct list node;
void qinsert(node**,node**);      //function declaration
                    //for insert operation
void qdelete(node**,node**); //function declaration for delete
                                    //  operation
void disp(node*);  //function declation for dispalying the queue
void main()
```

```
{
    int opt;      //option inputing variable
    char ch;    //choice inputing variable
    clrscr();
    node *front;
    node *rear;
    rear=front=NULL;  //initially front and rear are assigned NULL
    do
    {   printf("\nEnter your option to perform Queue operation:::\n");
        printf("1. Insert\n");
        printf("2. Delete\n");
        printf("3. Display the Queue\n");
        scanf("%d",&opt);
        switch(opt)
        {
            case 1 :  qinsert(&front,&rear);  //qinsert() function call
                      break;
            case 2 :  qdelete(&front,&rear);//qdelete() function call
                      break;
            case 3 :  disp(front); //disp() function call
                      break;
        }
        printf("\nDo you wish to continue[y/n]:");
        ch=(char)getche();
    }while(ch=='Y' || ch=='y');
    printf("\nPress any key to exit from the program");
    getch();
}
```

//Insert operation

```
void qinsert(node **front,node **rear)
{
```

```
    node *new_element;   /* New element to be inserted */
    new_element=(node*)malloc(sizeof(node));
    //memory allocated
    new_element->next=NULL;
printf("\nEnter the character to be inserted in the queue:");
fflush(stdin);
scanf("%c",&(new_element->data));
if(*front==NULL && *rear==NULL)
    {
        *front=new_element;
        *rear=new_element;
    }
    else
    {
        (*rear)->next=new_element;
        *rear=new_element;
    }
}
void qdelete(node **front, node **rear)  //Delete operation
{
    node *delnode;      // Node to be deleted
    if((*front)==NULL || (*rear)==NULL)
        printf("\nQueue is empty\n");
    else
    {
        delnode=*front;
        (*front)=(*front)->next;
        free(delnode);
    }
}
void disp(node *f)  //Displaying the queue
{
```

```
        printf("\nElements in the Queue are:::");
        while(f!=NULL)
        {
            printf(" %c ",f->data);
            f=f->next;
        }
}
```

The linked implementation of queue is more efficient than the array implementation (i.e., contiguous implementation) because it uses all its allocated storage. Moreover, the linked implementation releases memory. So it never occupies more memory than it needs.

9.5 APPLICATIONS OF QUEUE

Queues are widely used in simulations. In any situation, where the rate at which users' demand for services can exceed the rate at which those services can be supplied, queues are normally used.

Queues are useful at the time of sharing computer system where many users share the system simultaneously. For e.g., printer maintains a queue of jobs waiting to get the printer. A queue also stores the keystroke of data as you type on the keyboard. Queues are frequently used in computer programming and a typical example is the creation of job queue by an operating system. If the operating system does not use priorities, then the jobs are processed in the order they enter the system, i.e., first come first served basis. Queues are also used in Internet message packets waiting to be routed.



LET US SUM UP

Let us consider the queue as shown in figure 9.3. After delete operation, we see that, front is 1 and rear is 4 which is the last possible index of the rear, as the maximum length of the queue is 5. We cannot add an element although the first space of the

queue is empty. In case of a ordinary queue, the insertion takes place from the rear end and the deletion takes place from the front end. In the case of deletion from the front end the datas are deleted but the space of the queue is not utilized for the further storage. So this problem is solved in case of a ***circular queue.*** Even if the rear is full but there is space at the front end then the data can be stored in the front end until the queue overflows. In a circular queue, after last element, first element occurs i.e the next part of the last node points to the first node of the queue as shown in the following figure 9.6.

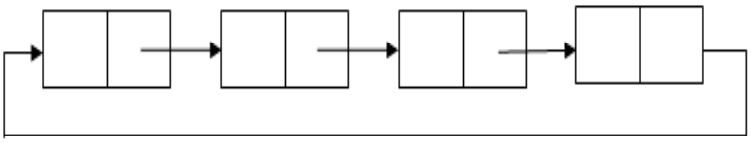


Fig 9.6: Circular queue



CHECK YOUR PROGRESS

Q.1. State whether the following statements are true(T) or false(F):

i) When an element is inserted in a queue, the position of rear decreases.

ii) When an element is deleted from a queue the value of front increases by one.

iii) Queue is a FIFO structure.

iv) Insertion takes place at front end in queue.

v) In queue, deletion takes place at front end.

Q.2. Why are queues called “FIFO” structures?

.....



9.6 LET US SUM UP

- Queue is a data structure in which objects are removed from one end called the *front(head),* and added to the other, called the *rear(tail or back)*.
- A physical analogy for a queue is a line of people at a bank. When you go to the bank, customers go to the *rear* (back) of the line and customers come off the line (i.e., are serviced) from the *front* of the line. In computer science examples of queue are: a sequence of stored data or programs awaiting for processing, queues of task waiting for printing etc.
- Queue is a *First-In First-Out (FIFO)* data structure
- Two operations can be carried out on queue:
  Insertion of element and
  Deletion of element
- Queue can be inplemented two ways:
  Using array and
  Using linked list.



9.7 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1 : i) False; ii) True; iii) True; iv) False; v) True

Ans. to Q. No. 2 : "FIFO" is an acronym for "First-In_First_Out". Queues are called "FIFO" structures because the first element that is inserted into a queue is always the first element to be removed.



9.8 FURTHER READING

1. Seymour, L. (2010). Data Structures with C. Tata McGraw-Hill publication.
2. Kamthane, A. N. (2012). Introduction to Data Structures in C. Dorling Kindersley.



9.9 MODEL QUESTIONS

Q.1. What is queue? What are its properties?

Q.2. What are the applications of queue?

Q.3. What is overflow and underflow in a queue?

Q.4. Write down the steps of insertion algorithm of queue implemented with array.

Q.5. Write a program for inserting element in a queue of fixed length.

Q.6. Write a function for queue insert operation using array implementation of queue.

Q.7. Write an algorithm for insertion of element in a queue implementaed as an array. Write a C function for the insert operation.

Q.8. Write an algorithm for deletion of element in a queue implementaed as an array. Write a C function for the delete operation.

UNIT 10 : SEARCHING

UNIT STRUCTURE

10.1 Learning Objectives
10.2 Introduction
10.3 Searching
    10.3.1 Linear Ssearch
    10.3.2 Binary Search
10.4 Performance and Complexity using Big 'O' notation
10.5 Let Us Sum Up
10.6 Answers to Check Your Progress
10.7 Further Reading
10.8 Model Questions

10.1 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 performance and complexity of search technique using Big 'O' notation.

10.2 INTRODUCTION

In our day-to-day life there are various applications, in which the process of searching is to be carried. Searching the name of a person from the given list, searching a specific card from the set of cards, etc., are some examples of searching. Searching is an important function in computer science. 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. Searching methods are designed to take advantage of the file

(that consits of many records/data) organisation and optimise the search for a particular record or to establish its absence.

This unit will focus on searching for data stored in a linear data structure such as an array or linked list.

10.3 SEARCHING

Searching is a technique of finding an element from the given data list or set of the elements like an array etc. For example, let us consider an array of 20 elements. These data elements are stored in successive memory locations. We need to search an element from the array. In the searching operation, assume a particular element ***e*** to be searched. The element ***e*** is compared with all the elements in a list starting from the first element of an array till the last element. In other words, the process of searching is continued till the element is found or list is completely exhausted. When the exact match is found then the search process is successfully terminated. In case, no such element exists in the array, the process of searching should be abandoned.

Suppose, our element to be searched is 8. In that case if the given element is exist in the set of elements or array, then the search process is said to be successful as per Fig.10.1(a). It indicates that the given element belongs to the array.

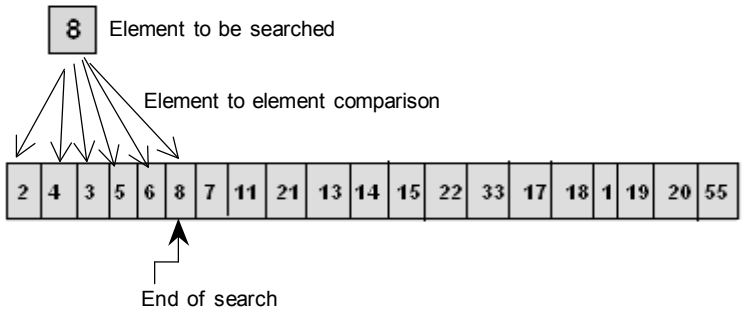


Fig. 10.1 (a) : Successful search

Again, let us consider our element to be searched is 66. The search is said to be unsuccessful if the given element does not exist in the array as per Fig. 10.1(b). It means that the given element does not belong to the array or

collection of the items.

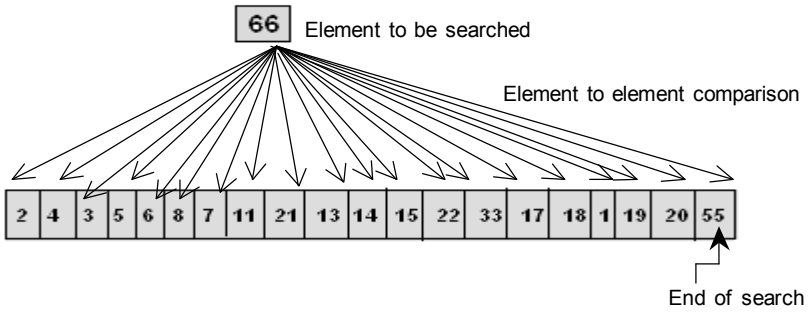


Fig. 10.1 (b) : Unuccessful search

We will now discuss two searching techniques :

- Linear or Sequential search
- Binary search

10.3.1 Linear Search

Linear search, also known as ***sequential search***, means starting at the beginning of the data and checking each item in turn until either the desired item is found or the end of the data is reached. This is the most natural searching method. 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. The linear search 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 the next element.

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

```
/* Program 10.1 :Program to search an element from the entered numbers
(a program for linear searching) */
    # include<stdio.h>
    #include<conio.h>
    void main()
    {
        int arr[20],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++)
        {
            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 program above, suppose the user enters 10 numbers i.e., *n=10* using the first *for* loop. 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 elements in the array then the

location is displayed. Otherwise the 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 lowest possible comparison is equal to **1** when the required item is the first item in the list. 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. Thus, if the required item is in position **i** in the list, then only **i** comparisons are required. Hence the average number of comparisons done by linear (or sequential) search will be :

$$\frac{1+2+3+.....+i+.....+N}{N}$$

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

$$= \frac{N+1}{2}$$

Advantages and Disadvantages of Linear Search : 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 the worst case if the element is at the end of the list.



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

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

.....

.....

10.3.2 Binary Search

The binary search approach is different from the linear search. Let us consider a list in ascending sorted order. It would work to search from the beginning until an 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 in which half the item is in (because the list is sorted). This effectively reduces the working range into half of the original list 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. We compare the element with middle element of the array or list. If it is less than the middle element then we search it in the left portion of the array and if it is greater than the middle element then search will be in the right portion of the list. Now, we will take that portion only for search and compare it with the middle element of that portion. This process will be in iteration until we find the element or the middle element has no left or right portion to search.

Let us take a sorted array of 11 elements and they are shown in Fig 10.2(a). Let us assume that we want to search the element 56 from the list of elements.

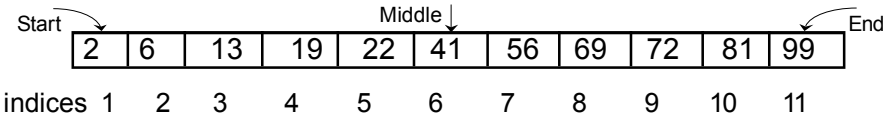


Fig. 10.2 (a) Sorted array

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 :

$$\text{middle} = \frac{\text{start} + \text{end}}{2}$$

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

Step 1 : The element 2 and 99 are at start and end positions respectively.

Step 2 : Calculate the middle index which is as

middle = (start + end)/2

middle = (1+11)/2

middle = 6

Step 3 : The key element 56 is to be comapred 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 10.2(b).

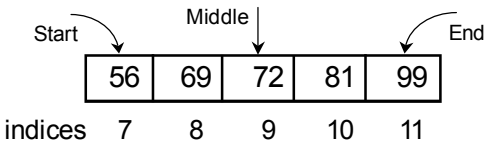


Fig. 10.2 (b)

Step 6 : The key element 56 is less 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

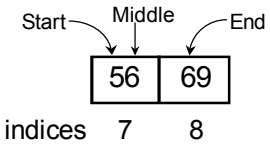


Fig. 10.2 (c)

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.

```
//Program 10.2: 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);
        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();
    }
```

In Binary search, by dividing the working data set in half with each comparison, logarithmic performance, O(log n), is achieved.

/* Program 10.3 :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();
}
```

10.4 PERFORMANCE AND COMPLEXITY

Searching for data is one of the fundamental fields of computing. Often, the difference between a fast program and a slow one is the use of a good algorithm for the data set. The ability to analyze a piece of code or an algorithm and understand its efficiency is vital for understanding computer science.

Suppose our problem is to input a name (key) and a list of names, and to determine whether the name occurs in the list and if so where does it occur.

For example, if a list contains 6 names i.e., N=6 and we are using ***linear search*** technique, then

	Compares to find
Rahul	*1*
Joy	*2*
Rita	*3*
John	*4*
Kiran	*5*
Krishna	*6*

Worst case : 6 compares.

Best case : 1 compare.

Average case : Expect to compare key with half of list

In linear (sequential) search ***Comparison*** is the key operation. Number of compares depends on data (position of the element to be searched and the number of elements). If the list or array size is N:

Best case : 1 compare (if the sequential search finds the name at the first position of the list then it will behave like best case).

Worst case : Compare key with all names in list. The sequential search situation will be in the worst case if the name is at the end of the list, i.e., the number of comparison needed is N.

Average case : Key may match with name in any position. Expect to compare with half of list.

Again, for example, if we are to determine a target value (key) that

occurs in a list using binary search, then the algorithm will be as follows

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")

Binary Search required a sorted array. Given that the array is of size n, split the array into half — test the middle element, originally n/2; if it is the value you are searching for, return the current index. Otherwise, if you have only one element in your subarray, return a sentinel indicating that you cannot find the value. If that is not the case, if your search key is less than the middle element, perform the same operation on the elements from the start to n/2 - 1; if the middle element is less than your search key, perform the search on the subarray of n/2 + 1 to n. Every time you do this, you cut the size of the array to search in half — so it takes **O(log n)** time with base 2. Binary search is worst case **O(log n)** where n is the size of the list and log is with base 2. Thus if n = 8, then Log 8 = Log $_2 2^3$ = 3 Log $_2 2$ = 3.



CHECK YOUR PROGRESS

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.



10.5 LET US SUM UP

- Searching is a technique of finding accurate location of an element in the given list of item or set of elements of an array.
- 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 is easy to implement and relatively efficient to use for small lists. But it is very time consuming for long unsorted lists.
- Binary Search works only on sorted lists (or arrays). It finds the median, 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. It has checks to be incorporated to ensure speedy termination under all possible conditions and hence it is very efficient. The prerequisite for it is that the list should be sorted order.



10.6 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;
}
```



10.7 FURTHER READING

1. Seymour, L. (2010). *Data Structures with C*. Tata McGraw-Hill publication.
2. Kruse, R., & Tondo, C. L. (2007). *Data structures and program design in C.* Pearson Education India.



10.8 MODEL QUESTIONS

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

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

Q.3. What are the precondition of binary search technique?

Q.4. Differentiate between linear and binary search.

Q.5. Explain linear and binary search.

Q.6. Write a program to demonstrate binary search. Use integer array and store 10 elements. Find the given element.

Q.7. Write down the best, worst and average case time complexity of sequential search.

UNIT 11 : SORTING

UNIT STRUCTURE

11.1 Learning Objectives
11.2 Introduction
11.3 Sorting
11.4 Insertion Sort
11.5 Selection Sort
11.6 Bubble Sort
11.7 Quick Sort
11.8 Let Us Sum Up
11.9 Answers to Check Your Progress
11.10 Further Reading
11.11 Model Questions

11.1 LEARNING OBJECTIVES

After going through this unit, you will able to

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

11.2 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 for 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. Also, we shall discuss the various sorting algorithms i.e. insertion sort, selection sort, bubble sort and quick sort including their complexity.

11.3 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.

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

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

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, in ascending order 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.

11.4 INSERTION SORT

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

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

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.

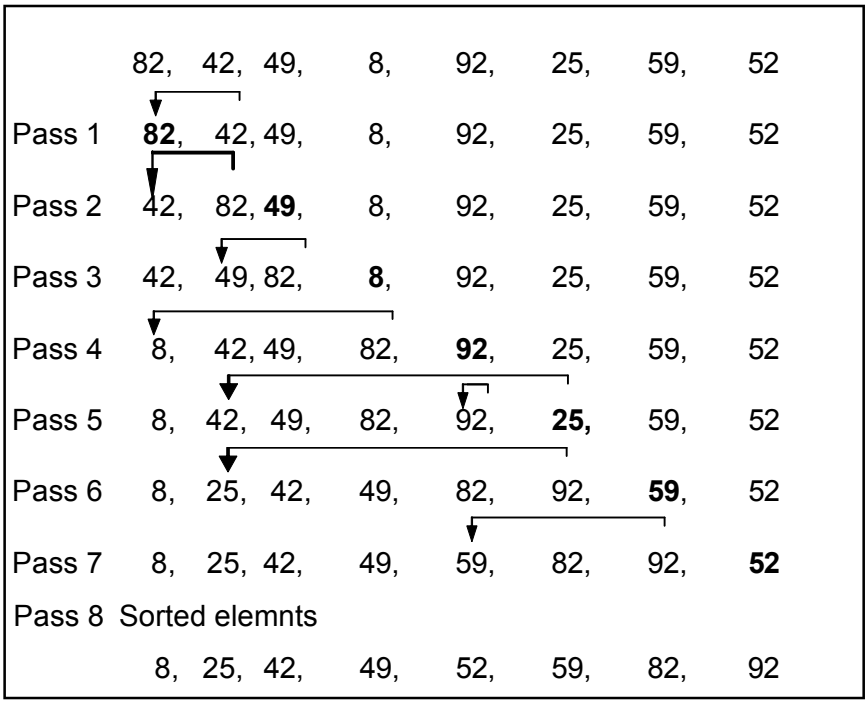
Insertion sort algorithm is frequently used when *n* is small.

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 :



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 element *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 comparison. Second element requires 1 comparison, third element requires 2 comparisons, fourth element requires 3 comparisons and so on. The last element requires n–1 comparisons. So the total number of comparisons will be

1 + 2 + 3 + ... + (n–2) + (n–1)

It is 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 = $\frac{(n-1)}{2}$ {2×1 + (n–1)–1} × 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 best case of the above algorithm then it will be of O(n). The insertion sort technique is very efficient if the number of element to be sorted are very less.

11.5 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 the smallest element is picked out 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 follows –

Pass 1: Search the smallest element from A[0], A[1], ..., ..., A[N–1].
Interchange A[0] with the smallest element
Result : A[0] is sorted.

Pass 2: Search the smallest element from A[1], A[2], ..., ..., A[N–1].
Interchange A[1] with the smallest element
Result : A[0], A[1] is sorted.

.....

.....

Pass N–1: Search the smallest element from A[N–2], A[N–1]

Interchange A[N -2] with the smallest element

Result : A[0], A[1], ..., ..., ..., A[N–2], A[N–1] is sorted.

Thus A is sorted after N -1 passes.

Let us take an example of the following elements :

Pass 1	**75**	35	42	13	87	24	64	57
Pass 2	13	**35**	42	75	87	**24**	64	57
Pass 3	13	24	**42**	75	87	**35**	64	57
Pass 4	13	24	35	**75**	87	**42**	64	57
Pass 5	13	24	35	42	**87**	75	64	57
Pass 6	13	24	35	42	57	**75**	**64**	87
Pass 7	13	24	35	42	57	64	**75**	87
Sorted Elements	13	24	35	42	57	64	**75**	87

```
/*Program of sorting using selection sort*/
#include <stdio.h>
#include<conio.h>
void main()
{
    int a[25], i, ,j, k, n, temp, smallest;
    clrscr();
    printf("Enter the number of elements : ");
    scanf("%d",&n);
    for (i = 0; i < n; i++)
    {
        printf("Enter element %d : ",i+1);
        scanf("%d", &a[i]);
```

```
}
        /* Display the unsorted list */
printf("Unsorted list is : \n");
for (i = 0; i < n; i++)
    printf("%d ", a[i]);
printf("\n");
        /*Selection sort*/
for(i = 0; i< n - 1 ; i++)
{
        /*Find the smallest element*/
    smallest = i;
    for(k = i + 1; k < n ; k++)
    {
        if(a[smallest] > a[k])
            smallest = k ;
    }
    if( i != smallest )
        {
            temp = a [i];
            a[i] = a[smallest];
            arr[smallest] = temp ;
        }
        printf("After Pass %d elements are :  ", i+1);
        for (j = 0; j < n; j++)
            printf("%d ", a[j]);
        printf("\n");
} /*End of for*/
    printf("Sorted list is : \n");
    for (i = 0; i < n; i++)
        printf("%d ", a[i]);
    printf("\n");
    getch();
} /*End of main()*/
```

Analysis : As we have seen, selection sort algorithm will search the smallest element in the array and then that element will be at proper position. So, in Pass 1 it will compare n–1 elements, in Pass 2 comparison will be n–2 because the first element is already at proper position. Thus, we can write the function for comparison as

F(n) = (n–1) + (n–2) + (n–3) + ... + 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 few temporary variables will be required for interchanging the elements and it is simple to implement.

11.6 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.

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



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*.

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

	0	1	2	3	4	5	6	7
A	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, 32 < 62, no change

d) Compare A3 and A4, 62< 68, no change

e) Compare A4 and A5, 68 > 52, interchange

13 20 32 62 52 68 38 46

f) Compare A5 and A6, 68>38, interchange

13 20 32 62 52 38 68 46

g) Compare A6 and A7, 68 > 46, 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 place it at proper position in each Pass. So in Pass 1 it will compare n -1 elements, in Pass 2 comparison will be n -2 because the first element is already at proper position. Thus, we can write the function for comparison as

F(n) = (n–1) + (n–2) + (n–3) + ... + 3 + 2 + 1

This is an arithmetic series, solving the series we will get

$$F(n) = \frac{n}{2}\{2a + (n-1)d\}$$

$$= \frac{(n-1)}{2}\{2 \times 1 + (n-1)-1) \times 1\}$$

$$= \frac{(n-1)}{2}(2 + n-2)$$

$$= \frac{n(n-1)}{2}$$

$$= O(n^2)$$

Thus, the time required to execute the bubble sort algorithm is proportional to (n^2), where *n* is the number of input items.

11.7 QUICK SORT

This sorting algorithm was 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 list of elements are divided into two sub-lists. For example, a list of *n* elements are to be sorted. The quick sort marks the first element in the list called as ***pivot*** or ***key***. Consider the first element as *P*. 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.

Now, the process for sorting the elements through quick sort is as follows:

1. Take the first element of list as pivot.
2. Place pivot at the proper place in list. So at least one element of the list i.e. pivot will be at its proper place.
3. Create two sublists in 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 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 comparison will start from the interchanged element position from left to right for getting the element which has higher value than pivot.
4. Repeat the same process untill pivot is at its proper position.

Let us take a list of elements and assume that 48 is the pivot element.

48 29 8 59 72 88 42 65 95 19 82 68

We have to start comparison 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 comparison 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 comparison will start from 59 and will be from right to left. The first element less than 48 is 42. So interchange it with pivot.

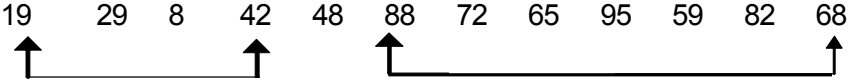
19 29 8 42 72 88 48 65 95 59 82 68

Now, the comparison 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 comparison will start from 72 and will be from right to left. There is no element less than 48. So, now 48 is at its proper position in the list. So we can divide the list into two sublist, left and right side of pivot.

19 29 8 42 48 88 72 65 95 59 82 68



Sublist 1 Sublist 2

Now, the same procedure will be followed for the 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 chosen in the middle position of the list so that it divides the list into two sublists of equal size. Now, repeatedly applying the quick sort algoritm on both the sublists we will finally have the sorted list of the elements.

Now, after 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 *I* is the number of steps. If *n* is comparison per step then for $\log_2 n$ steps we get = $n\log_2 n$ comparisons. 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 is that, the given list of the elements is initially sorted. We consider the first element of the list as the pivot element.

In this case, the number of steps needed for obtaining the finally sorted list is = (n–1).

Again, the number of comparisions in each step will be almost *n i .e. 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. Selection sort and quick sort both fall into the same category of sorting algorithms. What is this category?

A. O(n log n) sorts B. Divide-and-conquer sorts

C. Interchange sorts D. Average time is quadratic.

Q.2. What is the worst-case time for quick sort to sort an array of n elements?

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

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

Q.3. When is insertion sort a good choice for sorting an array?

A. Each component of the array requires a large amount of memory.

B. Each component of the array requires a small amount of memory.

C. The array has only a few items out of place.

D. The processor speed is fast.

Q.4. How does a quick sort perform on :

A. An array that is already sorted

.....

B. An array that is sorted in reversed order

.....

Q.5 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?

.....



11.8 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 it inserts the next element in its correct position relative to the subarray that has already been processed. The worst case 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 the smallest element is picked out from the list and placed in the second location and so on until the largest element of the list. The worst case 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 worst case time 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. It guarantees that the single element, called the pivot element, is in its correct position. Then the algorithm proceeds, applying the same method to the two pieces separately. This is naturally recursive. The best case complexity of Quick Sort is O(*nlogn*).



11.9 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1 : C

Ans. to Q. No. 2 : D

Ans. to Q. No. 3 : C

Ans. to Q. No. 4 : The quick sort is quite sensitive to input. The Quick Sort degrades 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 that the pivot element will be the first element. So, here it produces only 1 sublist which is on the right side of first element and starts from second element. Similarly, other sublists will be created only at right side. The number of comparison for first element is n; second element requires n–1 comparison and so on. Thus, we will get the complexity of order n^2 instead *logn.*

Ans. to Q. No. 5 : The Bubble Sort is slower than the Selection Sort, and the Insertion Sort (in most cases) is a little faster.



11.10 FURTHER READING

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

2. Seymour, L. (2010). *Data Structures with C*. Tata McGraw-Hill publication.

3. Kamthane, A. N. (2012). Introduction to Data Structures in C. Dorling Kindersley.



11.11 MODEL QUESTIONS

Q.1. Explain the different types of sorting memthods.

Q.2. What is sorting? Write a function for Bubble Sort.

Q.3. Write a program for sorting a given list by using Insertion Sort :

2, 32, 45, 67, 89, 4, 3, 8, 10

Q.4. Write a program to sort the following members in ascending order using Selection Sort :

12, 34, 5, 78, 4, 56, 10, 23, 1, 45, 65

UNIT 12 : TREES

UNIT STRUCTURE

12.1 Learning Objectives
12.2 Introduction
12.3 Definition of Tree
12.4 Binary Tree
12.5 Binary Tree Representation
12.6 Tree Traversal Algorithms
12.6.1 Preorder Traversal
12.6.2 Inorder Traversal
12.6.3 Postorder Traversal
12.7 Prefix, Postfix and Infix Notations
12.8 Let Us Sum Up
12.9 Answers to Check Your Progress
12.10 Further Reading
12.11 Model Questions

12.1 LEARNING OBJECTIVES

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

- learn about the basic definition of Tree, Binary Tree
- describe the different implemetations of Tree
- describe the different traversal algorithms of Binary tree
- distinguish between Prefix, Infix, Postfix notations

12.2 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 as elements are stored in a linear or sequential order. A tree is a nonlinear data structure that is, the elements are not stored in sequential order but it 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 basic terminology and its different operations as well as implementations of the trees and binary trees which are the most important nonlinear data structures. We will also discuss about the different tree traversal techniques in this unit.

12.3 DEFINITION OF TREE

The tree data structure is mainly used to represent the data containing a hierarchical relationship between elements, e.g. records, family trees and tables of elements.

Each tree must have atleast one node, called the **root**, from which all nodes of the tree extend (and which has no parent of its own). The other end of the tree – the last level down — contains the leaf nodes of the tree.

A tree can be recursively defined as a set of one or more nodes where one nodes is designated as the root of the tree and all the remaining nodes can be partitioned into non-empty sets each of which is a sub-tree of the root. The number of lines you pass through when you travel from the root until you reach a particular node is the **depth** of that node in the tree (node G in the figure 12.1 has a depth of 2 and node A has 0 depth). The **height** of the tree is the maximum depth of any node in the tree (the tree in Figure 12.1 has a height of 3). The number of children emanating from a given node is referred to as its **degree** — for example, node A in the figure has a degree of 3 and node H has a degree of 1.Remember that the degree of a leaf node is zero. **In-degree** of a node is the number of edges arriving at that node and **out-degree** of a node is the number of edges leaving that node. Every node in the tree is assigned a **level number** in such a way that the root node is at level 0,children of the root node are at level 1. Therefore, every node is at a level higher than its parent.

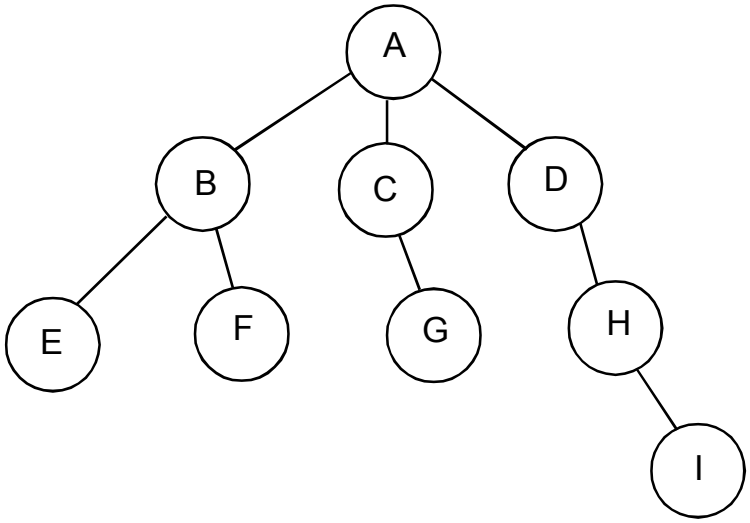


Fig. 12.1 : Tree

A tree is also defined as a finite nonempty set of elements in which one element is called the root and the remaining elements are partitioned into m>=0 disjoint subsets, each of which is itself a tree. Each element in a tree is called a node of the tree. A node with no subtree is a leaf.

An ordered tree is defined as a tree in which the subtrees of each node form an ordered set. A forest is an ordered set of ordered tree.

A general tree is defined to be a nonempty finite set T of elements, called nodes, such that :

1) T contains a distinguished element R , called the root of T.
2) The remaining elements of T form an ordered collection of zero or more disjoint trees T_1, T_2,.....T_n

The trees T_1, T_2,.....T_n are called subtrees of the root R, and the roots of T_1,T_2,.....T_n are called successors of R.

12.4 BINARY TREE

A binary tree T is defined as a finite set of elements, called nodes, such that :

a) T is empty (called the null tree or empty tree), or
b) T contains a distinguished node R, called the root of T, and the remaining nodes of T form an ordered pair of disjoint binary trees T1 and T2.

If T does contain a root R , then the two trees T1 and T2 are called, respectively, the left and right subtrees of R. If T1 is nonempty, then its root is called the left successor of R. Simillarly, if T2 is nonempty, then its root is called the right successorof R. In a binary tree, the top most element is called the root node and each node has 0,1 or at most 2 children.Every node contains a data element, a left pointer which points to the left child and a right pointer which points to the right child.

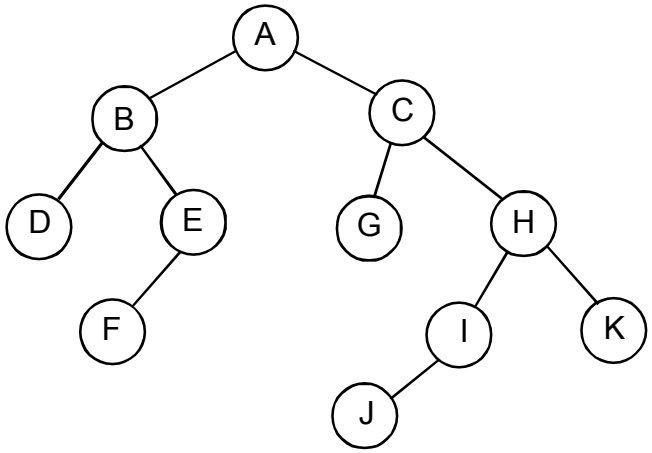


Fig. 12.2 : Binary Tree

In the above figure 12.2 a binary tree T is presented. Here in the binary tree T we have

i) T consists of 11 nodes, represented by the letter A through K
ii) The root of T is the node A at the top of the diagram.
iii) B is a left successor and C is the right successor of the node A.
iv) The left subtree of the root A consists of the nodes B, D, E, and F and the right subtree of A consists of the nodes C, G, H, I, J and K.

Any node N in a binary tree T has either 0, 1, or 2 successors. Here the nodes A, B, C and H have two successors. The nodes E and I have only one successor, and the nodes D, F, G, J and K have no successors. The nodes with no successors are called **terminal nodes**. A terminal node is also called a leaf, and a path ending in a leaf is called branch.

Each node in a binary tree T is assigned a level number. The root A of the tree T is assigned the level number 0, and and every other node is assigned a level number which is 1 more than the level number of its parent.

The ***depth*** (or ***height***) of a tree T is the maximum number of nodes in a branch of T. This is 1 more than the largest level number of T. A binary tree of height h has at least h nodes and at most 2^h-1 nodes. A binary tree of n nodes has exactly n-1 edges.The tree T in the figure 12.2 has depth 5.

- **Complete Binary Tree :** For any binary tree T each node of T can have at most two children. Accordingly, we can show that level r of binary tree T has at the most 2^r nodes. The tree T is said to be a complete binary tree 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.

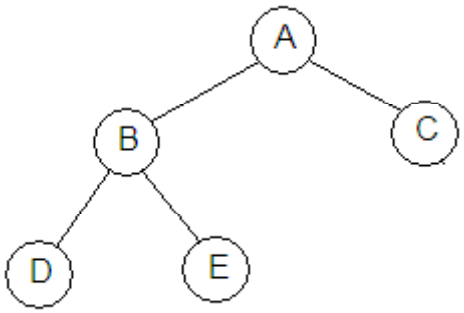


Fig. 12.3 : A complete Binary Tree

The height of a tree having exactly n nodes is given by $\log_2(n+1)$.

- **Strictly Binary Tree :** When every non-leaf node in binary tree is filled with left and right sub-trees, the tree is called strictly binary tree. It is shown in the figure 12.4.

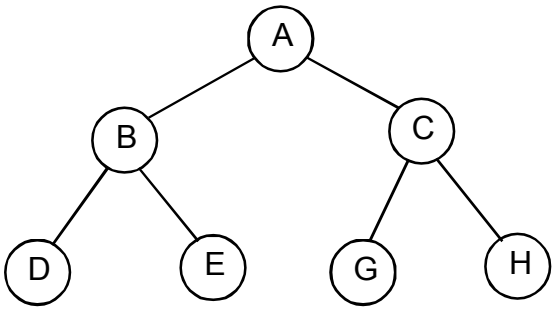


Fig.12.4 : A strictly Binary Tree

In the above strictly binary tree A, B and C are non-terminal nodes with non-empty left and right sub trees.

- **Extended Binary Tree :** When every node of a tree has either 0 or 2 children then such a tree is called extended binary tree or 2-tree. The nodes with two children are called internal nodes. The nodes without children are known as external nodes. Sometimes the nodes are distinguished in diagram by using circles for internal nodes and squares for external nodes.

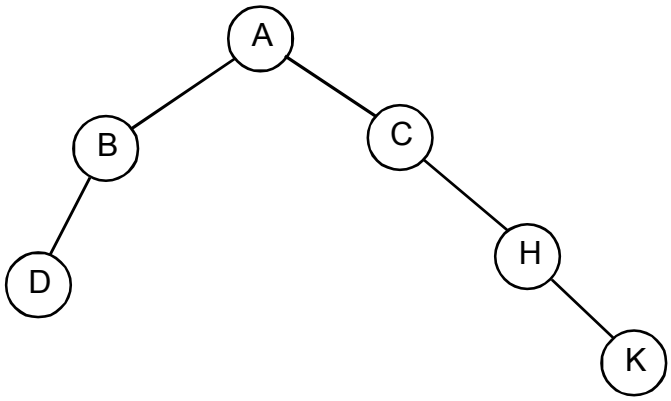


Fig. 12.5 (a) : Binary Tree

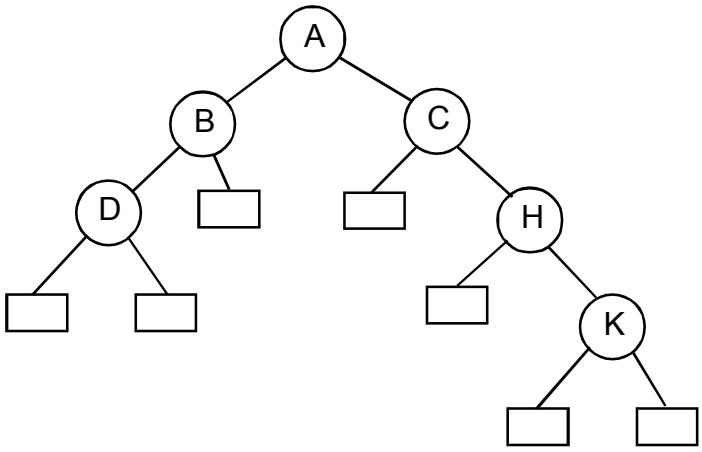


Fig. 12.5 (b) : Extended Binary Tree

12.5 BINARY TREE REPRESENTATION

Binary tree can be represented by two ways :

i) Array representation

ii) Linked representation

i) Array representation : In any type of data structure array representation plays an important role. The nodes of trees can be stored in an array. The nodes can be accessed in sequence one

after another.

In array, element counting starts from zero (0) to n–1 where n is the maximum number of nodes.

For example, for an integer array declaration :

int tree [n] ;

The root node of the tree always starts at index zero. Then successive memory locations are used for storing left and right child nodes. Consider the following :

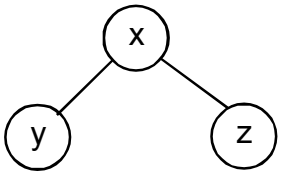


Fig. 12.6 : Array representation of tree

Note : Array representation of a binary tree with depth d will require an array with approximately 2^{d+1} elements.

ii) Linked representation : Let us consider a binary tree T. T can be maintained in memory by means of a linked representation which uses three parallel arrays INFO, LEFT and RIGHT, and a pointer variable ROOT as follows :

1) INFO[K] contains the data at the node N.
2) LEFT[K] contains the location of the left child of node N.
3) RIGHT[K] contains the location of the right child of node N.

ROOT will contain the location of the root R of T. If any subtree is empty, then the corresponding pointer will contain the null value. If the tree T itself is empty, then ROOT will conatin the null value.

Example 12.1: Write a C program to implement binary tree and display it.

```
#include <stdlib.h>
#include <stdio.h>
struct tree
 {
```

```
                char info;
                struct tree *left;
                 struct tree *right;
};
struct tree *root;                    /* first node in tree */
struct tree *stree(struct tree *root,
struct tree *r, char info);
void print_tree(struct tree *root, int l);
void main()
{
        char s[80];
        root = NULL;                  /*initialize the root */
         do
         {
                printf(Enter a letter: “);
                 gets(s);
                 root = stree(root, root, *s);
           }while(*s);
           print_tree(root,0);
          getch();
}  //end of main()
struct tree *stree( struct tree *root, struct tree *r, char info)
{
      if(!r)
      {
             r = (struct tree *) malloc(sizeof(struct tree));
             if(!r)
             {
                 printf(“Out of Memory\n”);
                 exit(0);
              }
              r->left = NULL;
              r->right = NULL;
```

```
            r->info = info;
            if(!root)
                return r ;
            if(info < root->info)
                root->left = r;
            else
                root->right = r;
            return r;
        }
        if(info < r->info)
            stree(r, r->left, info);
        else
            stree(r, r->right, info);
       return root;
}
void print_tree(struct tree *r, int l)
{
        int  i;
        if(!r)
            return ;
        print_tree(r->right, l+1);
        for (i=0; i<l; ++i) printf(" ");
                printf ( "%c\n", r->info);
        print_tree(r->left, l+1);
}
```

12.6 TREE TRAVERSAL ALGORITHMS

To traverse a binary tree means to pass through the tree, enumerating each of its nodes once. There are *three* standard methods of traversing a binary tree T with root R. These methods are all defined recursively, so that traversing a binary tree involves visiting the root and traversing its left and right subtree. The only difference among the methods is the order in which these three operations are performed. These traversal

methods are :

a) Preorder traversal

b) Inorder traversal

c) Postorder traversal

12.6.1 Preorder Traversal

To traverse a binary tree in ***preorder*** (also called ***depth-first order***), we perform the following three operations :

1) Visit the root
2) Traverse the left sub-tree in preorder
3) Traverse the right sub-tree in preorder

Function for preorder traversal is as follows:

```
void preorder(struct node *r)
    {
        if(r!=NULL)
        {
            printf("\t %d",r->data);
            preorder(r->left);
            preorder(r->right);
        }
    }
```

12.6.2 Inorder Traversal

To traverse a binary tree in ***inorder*** (also called ***symmetric order***), we perform the following three operations :

1) Traverse the left sub-tree in inorder
2) Visit the root
3) Traverse the right subtree in inorder

Function for inorder traversal is as follows:

```
void inorder(struct node *r)
    {
        if(r!=NULL)
        {
            inorder(r->left);
            printf("\t %d",r->data);
            inorder(r->right);
        }

    }
```

12.6.3 Postorder Traversal

To traverse a binary tree in ***postorder*** , we perform the following three operations :

1) Traverse the left subtree in postorder
2) Traverse the right subtree in postorder
3) Visit the root

Function for postorder traversal is as follows:

```
void postorder(struct node *r)
    {
        if(r!=NULL)
        {
            postorder(r->left);
            postorder(r->right);
            printf("\t %d",r->data);
        }
    }
```

Example 12.2: Write a C program to implement traversals (preorder, inorder, postorder) of binary tree.

```
#include<stdio.h>
#include<conio.h>
struct node
{
int data;
struct node *right, *left;
}*root,*p,*q;
struct node *make(int y)
{
        struct node *newnode;
        newnode=(struct node *)malloc(sizeof(struct node));
        newnode->data=y;
        newnode->right=newnode->left=NULL;
        return(newnode);
}
void left(struct node *r,int x)
{
        if(r->left!=NULL)
        printf("\n Invalid !");
        else
        r->left=make(x);
}
void right(struct node *r,int x)
{
        if(r->right!=NULL)
        printf("\n Invalid !");
        else
        r->right=make(x);
}
void inorder(struct node *r)
{
        if(r!=NULL)
```

```
        {
                inorder(r->left);
                printf("\t %d",r->data);
                inorder(r->right);
        }
}
void preorder(struct node *r)
{
        if(r!=NULL)
        {
                printf("\t %d",r->data);
                preorder(r->left);
                preorder(r->right);
        }
}
void postorder(struct node *r)
{
        if(r!=NULL)
        {
                postorder(r->left);
                postorder(r->right);
                printf("\t %d",r->data);
        }
}
void main()
{
        int no;
        int choice;
        clrscr();
        printf("\n Enter the root:");
        scanf("%d",& no);
        root=make(no);
        p=root;
```

```
while(1)
{
    printf("\n Enter another number:");
    scanf("%d",& no);
    if(no==-1)
    break;
    p=root;
    q=root;
    while(no!=p->data && q!=NULL)
    {
        p=q;
        if(no<p->data)
        q=p->left;
        else
        q=p->right;
    }
    else
    {
        right(p,no);
        printf("\n Right Branch of %d is %d",p->data,no);
    }
}
while(1)
{
    printf("\n 1.Inorder Traversal \n 2.Preorder Traversal
    \n 3.Postorder Traversal \n 4.Exit");
    printf("\n Enter choice:");
    scanf("%d",&choice);
    switch(choice)
    {
    case 1 :    inorder(root);
                break;
    case 2 :    preorder(root);
```

```
                    break;
            case 3 :    postorder(root);
                    break;
            case 4 :    exit(0);
            default:printf("Error ! Invalid Choice ");
                    break;
            }
            getch();
        }
    }
```

For example, let us look at the following traversals for the binary tree T given below :

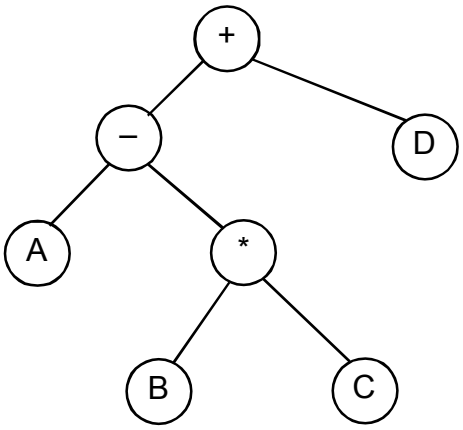


Fig. 12.7 Binary Tree

Inorder traversal : A – B * C + D

Preorder traversal : + – A * B C D

Postorder traversal : A B C * – D +

Inorder traversal of a binary tree visits the nodes in ascending order.

12.7 PREFIX, POSTFIX AND INFIX NOTATIONS

Let us consider the sum of two variables A and B. We think of applying the operator "+" to the operands A and B and write the sum as A+B. This particular representation is called *infix notation*. There are two alternate

notations for expressing the sum of A and B using the symbol A, B and +. These are:

+ A B (prefix notation)

A B + (postfix notation)

These prefixes *"pre-", "post-" and "in-"* refer to the relative position of the operators with respect to the two operands.These techniques of representing arithmetic expressions where operators will be before or after operands were presented by Polish (*of Poland*) mathematician Jan Lukasiewich. So, these notations are also called ***Polish notation***. According to this:

Prefix Notation : In prefix notation, the operator precedes the operands. For example : – + A B C

Postfix Notation : In postfix notation, the operator follows the operands. For example : A B + C –

Infix Notation : In infix notation, the operator is between the operands. For example : A + B – C

For example, let us write these three notations for the expression

A – B * C + D

Infix notation is : A – B * C + D

Prefix notation is : + – A*BCD

Postfix notation is : ABC* – D +

We can also express these three notations infix, prefix and postfix from the binary tree traversal point of view.

For example, the expression A – B * C + D can be represented by the binary tree as follows :

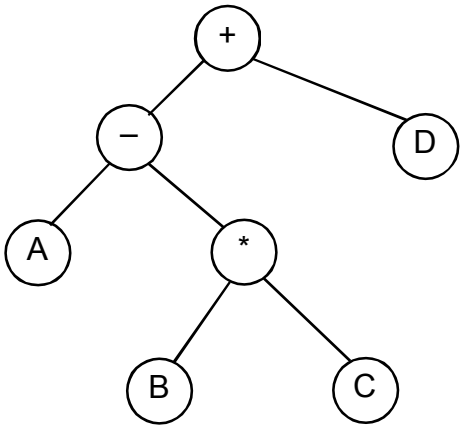


Fig. 12.8 : Binary Tree for A – B * C + D

If we traverse the above binary tree we see that :

Inorder traversal equvalents to infix expression : A – B * C + D

Preorder traversal equvalents to prefix expression : + – A * B C D

Postorder traversal equvalents to postfix expression : A B C * – D +



CHECK YOUR PROGRESS

Q.1. If a binary tree is traversed in inorder then numbers of the nodes are printed

a) in ascending order b) in descending order

c) randomly d) none of these.

Q.2. Binary expression tree is traversed in ___ traversal

a) preorder b) postorder

c)inorder d) both (a) & (b)

Q.3. The number of nodes in a full binary tree of depth 4 is

a) 15 b) 16 c) 14 d)13

Q.4. How many nodes does a complete binary tree of level 5 have ?

a) 15 b) 25 c) 63 d) 33

Q.5. At level r, binary tree T have at the most ___ nodes.

a) $2^r + 1$ b) 2^r c) $2^r - 1$ d) none of these

Q.6. Postorder traversal is

a) root,left subtree,right subtree

b) left subtree, root ,right subtree

c) left subtree, right subtree, root

d) root, right subtree, left subtree

Q.7. Prefix notation representation of : A + B – C is

a) A B + C – b) – + A B C

c) ABC + – d) none of these.

Q.8. Traversing a binary tree, first root and then left and right subtree is called

a) postorder b) inorder

c) perorder d) none of these

Q.9. The postfix equivalent of the prefix * + a b – c d is

a) a b + c d – * b) a b c d + – *

c) a b + c d * – d) a b + – c d *



12.8 LET US SUM UP

- A tree is a nonlinear data structure that represents data in a hierarchical manner. A general tree is defined to be a nonempty finite set T of elements, called nodes, such that :
  1) T contains a distinguished element R , called the root of T.
  2) The remaining elements of T form an ordered collection of zero or more disjoint trees T1,T2,.....Tn.

  The trees T1,T2,.....Tn are called subtrees of the root R, and the roots of T1,T2,.....Tn are called successors of R.
- A binary tree T is defined as a finite set of elements, called nodes, such that :
  a) T is empty (called the null tree or empty tree), or
  b) T contains a distinguished node R, called the root of T, and the

remaining nodes of T form an ordered pair of disjoint binary trees T1 and T2.

- The tree T is said to be a complete binary tree 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.
- When every non-leaf node in binary tree is filled with left and right sub-trees, the tree is called strictly binary tree.
- The degree of a node is equal to the number of children that a node has. The degree of a leaf node is zero.
- When every node of a tree has either 0 or 2 children then such a tree is called extended binary tree or 2-tree.
- Binary tree can be represented in two ways :
  1. Array representation
  2. Linked representation
- The height of a binary tree with n nodes is at least $\log_2(n+1)$ and at most n. A binary tree of height h has at least h nodes and at most 2^h-1 nodes.
- Traversal means passing through every node of the tree one by one. Every node is traversed only once. There are three tree traversing techniques :
  a) Preorder traversal
  b) Inorder traversal
  c) Postorder traversal
- Arithmetic expressions can be defined in three kinds of notation : infix, prefix and postfix.

12.9 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1 : a) ascending order
Ans. to Q. No. 2 : c) inordered
Ans. to Q. No. 3 : a) 15
Ans. to Q. No. 4 : c) 63
Ans. to Q. No. 5 : b) 2^r

Ans. to Q. No. 6 : c) left subtree, right subtree, root
Ans. to Q. No. 7 : b) – + A B C
Ans. to Q. No. 8 : c) perorder
Ans. to Q. No. 9 : a) a b + c d – *



12.10 FURTHER READING

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

2. Seymour, L. (2010). *Data Structures with C,* Tata McGraw-Hill publication.

3. Kamthane, A. N. (2012). *Introduction to Data Structures in C.* Dorling Kindersley.



12.11 MODEL QUESTIONS

Q.1. Explain trees and binary trees.
Q.2. Give the properties of binary tree.
Q.3. Explain the various types of binary trees.
Q.4. What do you mean by traversing? Explain different traversal methods of binary tree.
Q.5. How are binary trees represented ?

UNIT 13: BINARY SEARCH TREE

UNIT STRUCTURE

13.1 Learning Objectives
13.2 Introduction
13.3 Binary Search Tree
13.4 Searching in Binary Search Tree
13.5 Insertion into Binary Search Tree
13.6 Deletion from Binary Search Tree
13.7 Height of a Binary Search Tree
13.8 B-Tree
13.9 Let Us Sum Up
13.10 Answers to Check Your Progress
13.11 Further Reading
13.12 Model Questions

13.1 LEARNING OBJECTIVES

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

- learn the basic concept of Binary Search Tree (BST)
- describe the searching operation on BST
- explain the insertion and deletion operation on BST
- learn basic definition of B-Tree

13.2 INTRODUCTION

In the previous unit, we have already discussed about trees and its various types along with different tree traversal procedures. A tree is either empty (that cannot be drawn) or it consists of a ***node*** called the ***root*** of the tree and a number of disjoint trees called the ***subtrees*** of the original tree.

A node may contain a value or a condition or represents a separate data structure or a tree of its own. Each node in a tree has zero or more child nodes, which are below it in the tree (by convention, trees grow down,

not up as they do in nature).

A node that has a child is called the parent node (or ancestor node, or superior). The **height** of a node is the length of the longest downward path to a leaf from that node. The height of the root is the height of the tree. The **depth** of a node is the length of the path to its root (i.e., its root path).

The topmost node in a tree is called the **root node**. All other nodes can be reached from the root node by following edges or links. In some trees, such as heaps, the root node has special properties. Every node in a tree can be seen as the root node of the subtree rooted at that node. The nodes which does not contain any child nodes are called leaf nodes.

An **internal node** or **inner node** is any node of a tree that has child nodes and is thus not a leaf node.

A **subtree** is a portion of a tree data structure that can be viewed as a complete tree in itself. Any node in a tree T, together with all the nodes below it, comprise a subtree of T.

In this unit, we will introduce a binary tree data structure namely ***binary search tree***. We will also discuss about the different operations (i.e. insertion, deletion and searching) on binary search tree. At the end we will define B-tree.

13.3 BINARY SEARCH TREE

A binary tree is called a ***binary search tree (BST)*** if for every node in the tree the data elements in the node is greater than or equal to the data elements in all the nodes in the *left subtree* and is less than the data elements in all the nodes in the *right subtree*. It is also known as an ordered binary tree in which the nodes are arranged in an order.Since the nodes in a BST are ordered, the time required to search an element in the tree is greatly reduced as it does not need to traverse the entire tree. The worst case time complexity of BST is O(n) for searching an element of n elements.

BST also speeds up the insertion and deletion operations. BST can be considered as one of the efficient data structure especially when compared with sorted linear arrays and linked lists.

Consider a binary tree T. T is a binary seach tree, that is, the value of the data elements in every node N in T greater than or equal to the data elements in every node in left subtree and is less than the data elements in every node in right subtree as shown in Fig. 13.1. Here, suppose 23 is replaced by 32. Then T would still be a binary search tree. On the other hand, suppose, 23 is replaced by 42, then T would not be a binary search tree, since the 38 would not be greater than 42 in its left subtree.

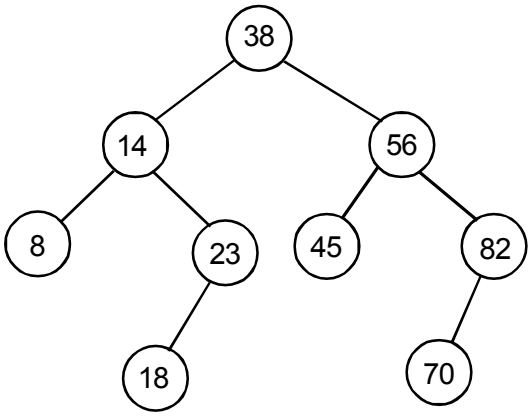


Fig. 13.1 : A Binary Seach Tree

The inorder traversal of such a binary search tree yields a list in sorted order. Now, let us discuss the various operations performed on BST i.e. searching for a node, inserting a node and deleting a node.

13.4 SEARCHING IN BINARY SEARCH TREE

Suppose T is a binary search tree. We are going to discuss the basic operation of searching with respect to the binary search tree T. The following algorithm finds the location of ITEM in the binary search tree T.

Step 1 : Compare ITEM with the root node N of the tree.

If ITEM = N then successful Else

i) If ITEM < N, proceed to the left child of N.

ii) if ITEM > N, proceed to the right child of N.

Step 2 : Repeat Step1 untill the following occurs :

i) We meet a node N such that ITEM = N. In this case the

search is successful.

The most common operation performed on a BST is searching for a key stored in the tree. These operations run in *O(h)* time where *h* is the height of the tree i.e. *h* is the number of links root node to the deepest node.

The TREE-SEARCH (*x, k*) algorithm searches the tree (with root at *x)* for a node whose key value equals *k*. It returns a pointer to the node if it exists, otherwise NIL

BSTREE_SEARCH(*x, k*)

if *x* = NIL OR *k* = key[*x*]

then return *x*

if *k* < key[*x*]

then

return **BSTREE_SEARCH** (left[*x*], *k*)

else

return **BSTREE_SEARCH** (right[*x*], *k*)

Clearly, this algorithm runs in *O*(*h*) time where *h* is the height of the tree.

Algorithm : Searching in Binary Search Tree

FINDBST(INFO, LEFT,RIGHT,ROOT,ITEM,LOC,PAR)

A binary search tree T is in memory and an ITEM of information is given. This procedure finds the location LOC of ITEM in T and also the location PAR of the parent of ITEM. There are three special cases:

1. LOC=NULL and PAR=NULL will indicate that the tree is empty.
2. LOC=!NULL and PAR=NULL indicates that ITEM is the root of T.
3. LOC=NULL and PAR=!NULL indicates that ITEM is not in T and can be added to T as a child of the node N with location PAR.

Step 1 : If(ROOT=NULL) then // Tree Empty//
Set LOC:=NULL and PAR:=NULL and Return.

Step 2 : If (ITEM=INFO[ROOT] then // ITEM at root//
Set LOC:= ROOT and PAR:=NULL and Return.

Step 3 : If (ITEM<INFO[ROOT] then //Initialize pointers PTR and SAVE/
Set PTR:=LEFT[ROOT] and SAVE:=ROOT

Else Set PTR:=RIGHT[ROOT] and SAVE:=ROOT

Step 4 : Repeat Step5 and Step6 while PTR=!NULL

Step 5 : If(ITEM=INFO[PTR]) then

Set LOC:=PTR and PAR:=SAVE and Return

Step 6 : If(ITEM<INFO[PTR]) then

Set SAVE:=PTR and PTR:=LEFT[PTR]

Else Set SAVE:=PTR and PTR:=RIGHT[PTR]

End of the step4 loop.

Step 7 : Set LOC:=NULL and PAR:=SAVE

Step 8 : Exit.

Example 13.1: Write a C program to search an element from the binary search tree.

Solution :

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

struct tree
{
    int data;
    struct tree *lft;
    struct tree *rt;
};

int sn;
struct tree *bt=NULL;
struct tree *insert(struct tree *bt,int no);
void search(struct tree *bt, int sn);

void main()
{
    int no;
    clrscr();
```

```
        printf("\nEnter the nodes of tree in preorder and O to quit");
        scanf("%d",$no);
        while(no!=0)
        {
            bt=insert(bt,no);
            scanf("%d",&no);
        }
        printf("\nEnter the number to be searched=");
        scanf("%d",&ns);
        search(bt,sn);
        }
struct tree *insert(struct tree *bt,int no)
{
        if(bt==NULL)
        {
            bt=(struct tree *)malloc(sizeof(struct tree));
            bt->lft=bt->rt=NULL;
            bt->data=no;
        }
       else
        {
            if(no<bt->data)
                bt->lft=insert(bt->lft,no);
            else if(no>bt->data)
                bt->rt=insert(bt->rt,no);
            else if(no==bt->data)
            {
                printf("\nDuplicate nodes:Program exited");
                exit(0);
            }
        }
        return(bt);
        }
```

```
void search(struct tree *bt,int fn)
{
    if(bt==NULL)
    printf("\nThe number doesnot exit");
    else if( fn==bt->data)
    printf("\nTHe numbsr %d is present in tree",fn);
    else if(fn<bt->data)
    search(bt->lft,fn);
    else
    search(bt->rt,fn);
}
```

13.5 INSERTION INTO BINARY SEARCH TREE

An element can be inserted into an appropriate position in a BST. For inserting an element, we will have to locate the parent node. Therefore, the node's value is first compared with the value of the root. If its value is less than the root, it is then compared with the value of the root's left child and recursively called the left sub-tree. If its value is greater than the root, it is compared with the root's right child and recursively called the right sub-tree. This process continues, until the new node is compared with a leaf node, and then it is added as this node's right or left child, depending on its value.

In the following we represent the function to implement insertion operation into a binary search tree.

```
void InsertNode(struct node *treeNode, struct node *newNode)
{
    if (treeNode == NULL)
        treeNode = newNode;
        else if (newNode->value < treeNode->value)
        InsertNode(treeNode->left, newNode);
        else
            InsertNode(treeNode->right, newNode);
}
```

This operation requires time proportional to the height of the tree in the worst case, which is O(log n) time in the average case over all trees. Consider the following binary search tree T. Suppose we want to insert an ITEM=20. Then we proceed in the following ways :

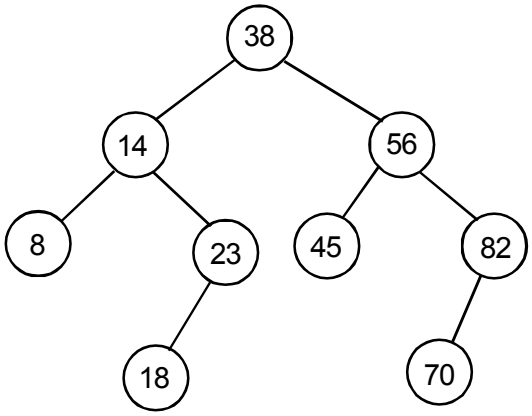


Fig. 13.2 : Binary Search Tree T

Step 1 : Compare ITEM=20 with the root, 38 of the tree T. Since 20<38,proceed to the left child of 38, which is 14.

Step 2 : Compare ITEM =20 with 14, Since 20>14, proceed to the right child of 23,

Step 3 : Compare ITEM =20 with 23, Since 20<23, proceed to the left child of 23, which is 18.

Step 4 : Compare ITEM =20 with 18. Since 20>18 and 18 does not have a right child, insert 20 as the right child of 18.

Hence we will get the following Binary search Tree after inserting the ITEM=20.

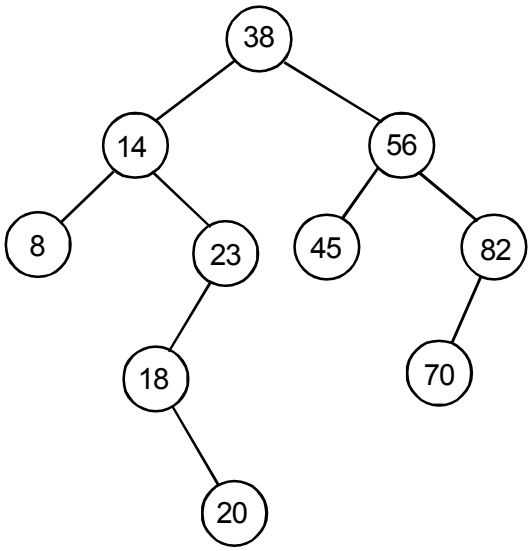


Fig. 13.3 : BST after inserting ITEM 20

Example 13.2: Write a C program to insert an element into the binary search tree.

Solution :

```
#include<stdio.h>
#include<conio.h>
struct tree
{
        int data;
        struct tree *lft ,*rt;
}
int in;
struct tree *bt=NULL;
struct tree * insert(struct tree *bt, int no);
void inorder(struct tree *bt);
void main()
{
        clrscr();
        printf("\nEnter the nodes of the tree in preorder: and 0 to quit");
        scanf("%d",&no);
        while(no!=0)
```

```
        {
                bt=insert(bt,no);
                scanf("%d",&no);
        }
        printf("\nEnter the number to be inserted=");
        scanf("%d",&in);
        bt=insert(bt,in);
        printf("\nThe inorder of tree after insertion of an element");
        inorder(bt);
}
struct tree *insert(struct tree *bt, int no)
{
        if (bt==NULL)
        {
                bt=(struct tree *)malloc(sizeof(struct tree));
                bt->lft=bt->rt=NULL;
                bt->data=no;
        }
        else
        {
                if(no<bt->data)
                bt->lft=insert(bt->lft,no);
        else
                if(no>bt->data)
                bt->rt=insert(bt->rt,no);
        else
                if(no==bt->data)
                {
                        printf("\nDuplicate nodes :Program exited");
                        exit(0);
                }
        }
        return(bt);
```

```
}
void inorder(struct tree *bt)
{
        if(bt!=NULL)
        {
                inorder(bt->lft);
                printf("%d",bt->data);
                inorder(bt->rt);
        }
}
```

13.6 DELETION FROM BINARY SEARCH TREE

Suppose T is a binary search tree, and an ITEM of information is given which needs to be deleted from the tree T.

There are several cases to be considered :

1. Deleting a leaf : deleting a node with no children is easy, as we can simply remove it from the tree.
2. Deleting a node with one child : delete it and replace it with its child.
3. Deleting a node with two children : suppose the node to be deleted is called N. We replace the value of N with either its in-order successor (the left-most child of the right subtree) or the in-order predecessor (the right-most child of the left subtree).

Once we find either the in-order successor or predecessor, swap it with N, and then delete it. Since both the successor and the predecessor must have fewer than two children, either one can be deleted using the previous two cases.

Let us take an example. From the following BST, if we try to delete a leaf, there is no problem. We just delete it and the rest of the tree is exactly as it was, so it is still a BST.

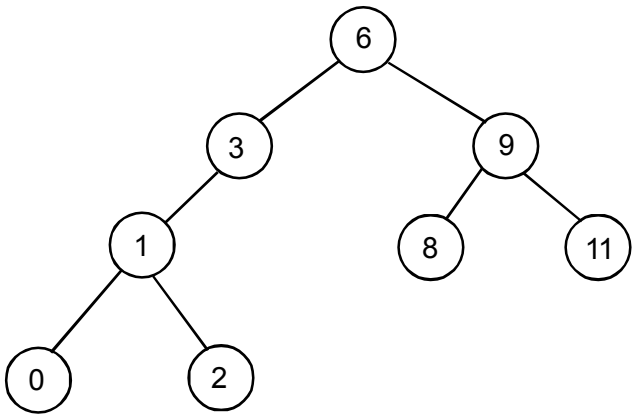


Fig. 13.4 Example of BST

There is another situation: suppose the node we're deleting has only one subtree. In the BST of Fig. 13.4, '3' has only 1 subtree.

To delete a node with 1 subtree, we just 'link past' the node, i.e. connect the parent of the node directly to the node's only subtree. This always works, whether the one subtree is on the left or on the right. Deleting '3' gives us:

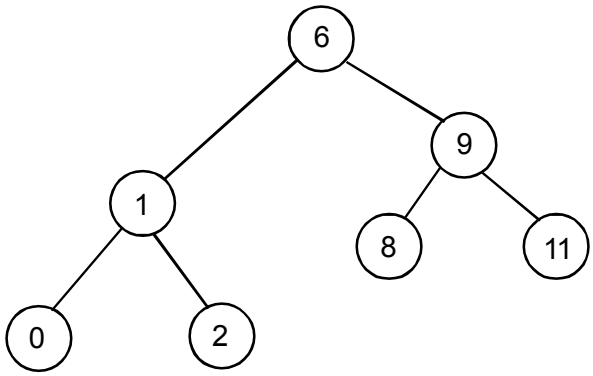


Fig. 13.5 After deleting '3'

which we normally draw as :

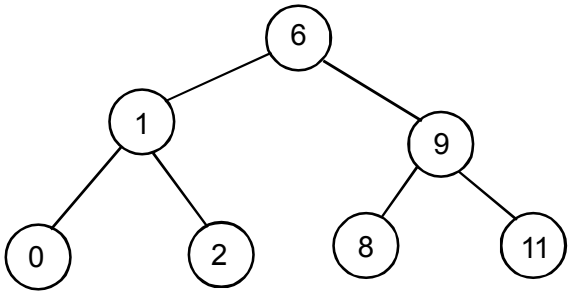


Fig. 13.6 Normal adjusting after deleting '3'

Finally, let us consider the only remaining case : how to delete a node having two subtrees. For example, how to delete '6'? We'd like to do this with minimum amount of work and disruption to the structure of the tree.

The standard solution is based on this idea : we leave the node containing '6' exactly where it is, but we erase the value 6 and find another value to store in the '6' node. This value is taken from a node below the '6's node, and it is *that* node that is actually removed from the tree.

Let us take a BST structure from which we want to remove '6'. This structure is shown in the in figure 13.7.

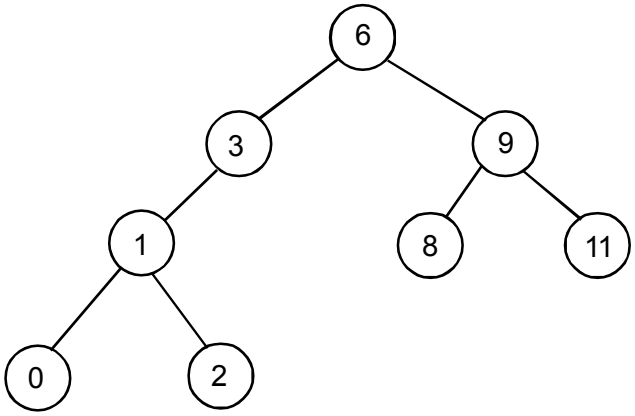


Fig. 13.7 Example of BST

Now we erase 6, but keep its node.

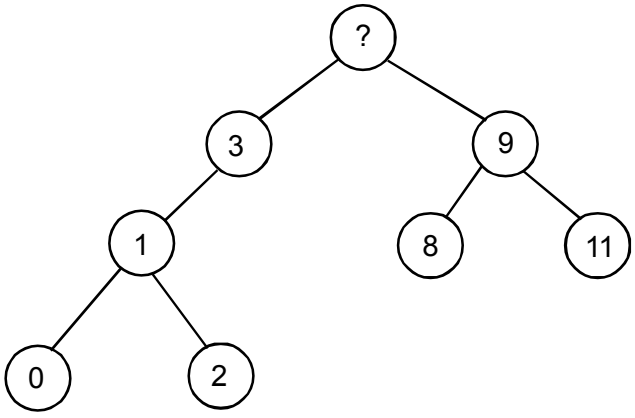


Fig. 13.8 After erasing '6'

Now, what value can we move into the vacated node and have a binary search tree? Let us try. If we choose value X, then : everything in the left subtree must be smaller than X and everything in the right subtree

must be bigger than X.

Let us suppose we're going to get X from the left subtree. In our example, 3 is the largest value in the left subtree. So if we put 3 in the vacated node and delete it from its current position we will have a BST with 6 deleted.

Here it is :

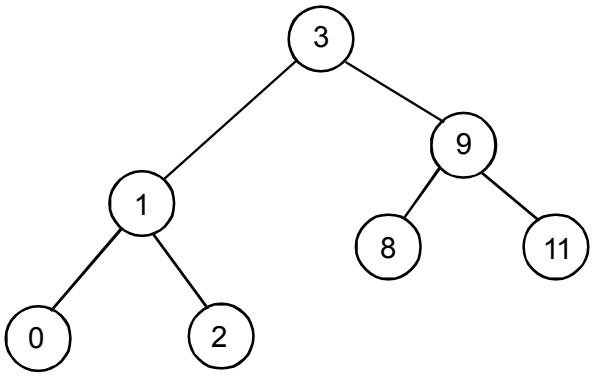


Fig. 13.9 BST after deleting '6'

So our general algorithm is : to delete N, if it has two subtrees, replace the value in N with the largest value in its left subtree and then delete the node with the largest value from its left subtree.

13.7 HEIGHT OF A BINARY SEARCH TREE

In order to determine the height of a binary search tree (BST) we can calculate the height of the left sub-tree and the right sub-tree. Whichever sub-tree's height is greater, 1 is added to it.

For example , if the height of the right sub-tree is greater than that of the left sub-tree then 1 is just added to the right sub-tree ,otherwise add 1 to the left sub-tree. Simillarly, to determine the number of nodes in a BST we calculate the total number of nodes in the both left sub-tree and right sub-tree and add 1 to it. That means,

Number of nodes in BST=

total nodes (left sub-tree) + total nodes (right sub-tree)+1

If the BST is empty then the number of nodes will be zero.

13.8 B-TREE

A *B-tree* is a specialized M-way tree developed by Rudolf Bayer and Ed McCreight in 1970 which is widely used for disk access.It is a *m-way* search tree, that have the advantage of minimizing file access due to their restricted height. However, it is essential that the height of the tree be kept as low as possible and therefore there arises the need to maintain balanced m-way search trees. Such a balanced m-way search tree is defined as a ***B-tree***. A *B-tree* of order m can have a maximum of m-1 keys and m pointers to its sub-trees.It may contain a large number of key values and pointers to its sub-trees.

A *B-tree* is designed to store sorted data and allows search,insertion and deletion operations to be performed in logarithmic time.

Each piece of data stored in a B-tree is called a “key”. Each key is unique and can occur in the B-tree in only one location. A B-tree consists of “node” records containing the keys and pointers that link the nodes of the B-tree together. Every B-tree is of some “order n”, meaning that nodes contain from n to 2n keys, and nodes are thereby always at least half full of keys. Keys are kept in sorted order within each node. A corresponding list of pointers is placed between keys to indicate where to search for a key if it is not in the current node. A node containing k keys always also contains k+1 pointers.

A B-tree is a rooted tree having the following properties:

1. Every node x has the following fields:

a) n[x], the number of keys currently stored in node x.

b) The n[x] keys themselves, stored in nondecreasing order, so that $key_1[x] <= key_2[x]] <= ..... <= key_{n[x]}[x]$.

c) leaf[x], a boolean values that is TRUE if x is a leaf and FALSE if x is an internal node.

2. Each internal node x also contains n[x]+1 pointers $c_1[x]$, $c_2[x]$, ..., ..., $c_n[x]+1$ to its children. Leaf nodes have no children, so their c_i fields are undefined.
3. The keys $key_i[x]$ separate the ranges of keys stored in each subtree: If k_i any key stored in the subtree with root $c_i[x]$, then
   $k_1 <= key_1[x] <= k_2 <= key_2[x] <= ..... << key_{n[x]}[x] << k_{n[x]+1}$
4. All leaves have the same depth, which is the tree's height h.
5. There are lower and upper bounds on the number of keys a node can contain. These bounds can be expressed in terms of a fixed integer t>=2 called the *minimum degree* of the B-tree.
   a) Every node other than the root must have at least t-1 keys. Every internal node other than the root thus has at least t children. If the tree is nonempty, the root must have at least one key.
   b) Every node can contain at most 2t-1 keys. Therefore, an internal node can have at most 2t children. We say that a node is full if it contains exactly 2t-1 keys.



CHECK YOUR PROGRESS

Q.1. A binary search tree (BST) is a binary tree data structure which has the following properties:
   i) Each node has a value.
   ii) A total order is defined on these values.
   iii) The left subtree of a node contains only values less than the node's value.
   iv) The right subtree of a node contains only values greater than or equal to the node's value. Which properties are true? Select the most suitable option.
   a) 1, & 2 only b) 1, 2 & 3 only
   c) 2 & 3 only d) all above

Q.2. Which of the following traversal techniques lists the nodes of a binary search tree in ascending order?

a) Post-order b) In-order

c) Pre-order d) none of these.

Q.3.Binary expression tree is traversed in ..... traversal.

a) Pre-order b) Post-order.

c) In-order d) Both (a) & (b).

Q.4. The most common operation performed on a BST is searching for a key stored in the tree. If h is the height of the tree then this operation runs in the time

a) O(h) b) O(2h)

c) O(1) d) none of above

Q.5. To traverse a non-empty binary search tree in inorder, perform the following operations :

a) i) Traverse the left subtree in inorder

ii) Visit the root

iii) Traverse the right subtree in inorder

b) i) Visit the root

ii) Traverse the left subtree in preorder

iii) Traverse the right subtree in preorder

c) i) Traverse the left subtree in postorder

ii) Traverse the right subtree in postorder

iii) Visit the root

d) none of these



13.9 LET US SUM UP

- Binary Search Tree is a binary tree in which each internal node *x* stores an element so that the elements stored in the left subtree of *x* are less than or equal to *x* and elements stored in the right subtree of *x* are greater than or equal to *x*.
- In-order traversal of the binary search tree will produce the elements in ascending order.
- Basic operations performed on BST are searching, insertion and

deletion.

- Searching a binary tree for a specific value can be a recursive or iterative process.
- Insertion begins as a search would begin; if the root is not equal to the value, we search the left or right subtrees to findout the appropriate position for inserting elements.
- For deleting a particular node from a BST, there are several cases to be considered :
  - i) Deleting a leaf : deleting a node with no children is easy, as we can simply remove it from the tree.
  - ii) Deleting a node with one child : delete it and replace it with its child.
  - iii) Deleting a node with two children : suppose the node to be deleted is called N. We replace the value of N with either its in-order successor (the left-most child of the right subtree) or the in-order predecessor (the right-most child of the left subtree).



13.10 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1 : d) all above

Ans. to Q. No. 2 : c) Pre-order

Ans. to Q. No. 3 : d) Both (a) & (b).

Ans. to Q. No. 4 : a) O(h)

Ans. to Q. No. 5 : a) i)Traverse the left subtree in inorder
ii) Visit the root
iii) Traverse the right subtree in inorder



13.11 FURTHER READING

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

2. Seymour, L. (2010). *Data Structures with C.*

3. Kamthane, A. N. (2012). I*ntroduction to Data Structures in C*. Dorling Kindersley.



13.12 MODEL QUESTIONS

Q.1. What is binary search tree ? How is an element searched ?

Q.2. Explain the procedure of insertion and deletion of data in binary search tree.

Q.3. Write a C program to search an element from the binary search tree.

Q.4. Write a program to display all nodes of the binary search tree and then insert data and display it along with the previous data of the tree.

Q.5. Explain the various traversal techniques in a binary search tree.

Q.6. What is B-tree? What are the properties of a B-tree?

*****

UNIT 14 : INTRODUCTION TO GRAPH

UNIT STRUCTURE

14.1 Learning Objectives
14.2 Introduction
14.3 Graph
14.4 Terminologies of Graph
14.5 Different Types of Graph
14.6 Representation of Graph
14.6.1 Sequential Representation of Graph
14.6.2 Linked Representation of Graph
14.7 Traversal in Graphs
14.7.1 Depth First Search
14.7.2 Breadth First Search
14.8 Let Us Sum Up
14.9 Answers to Check Your Progress
14.10 Further Reading
14.11 Model Questions

14.1 LEARNING OBJECTIVES

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

- learn about the basic definition of graph
- describe the different types of graph
- describe the various representations of graph
- distinguish between depth first search and breadth first search

14.2 INTRODUCTION

Like tree, graphs are also nonlinear data structure. Tree has some specific structure whereas graph does not have a specific structure. It varies from application to application in our daily life. Graphs are frequently used in every walk of life. A map is a well-known example of a graph. In a map various connections are made between the cities. The cities are connected

via roads, railway lines and aerial network. For example, a graph that contains the cities of India is 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 graph traversal techniques like depth first search and breadth first search.

14.3 GRAPH

A graph G is a collection of two sets V and E where V is the collection of vertices v_0, v_1, ..., v_{n-1} and E is the collection of edges e_1, e_2, ..., e_n where an edge is an arc which connects two vertices. The vertices are also termed as *nodes* and edges are termed as *arcs*. This can be represented as

$$G = (V, E)$$

where, $V(G) = \{v_0, v_1, v_2, ..., v_{n-1}\}$ and $E(G) = \{e_1, e_2, e_3, ...e_n\}$.

For example let us consider the graph G = (V, E) shown in Fig.14.1

where V(G) = {A, B, C, D, E} and

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

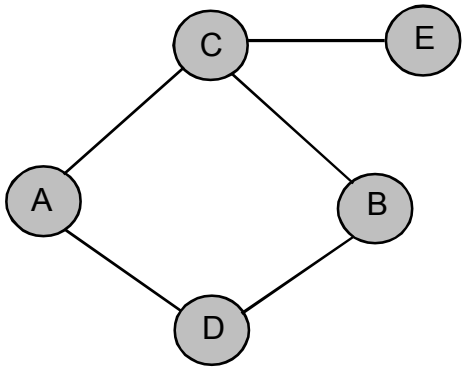


Fig. 14.1 : A Graph

14.4 TERMINOLOGIES OF GRAPH

Weighted Graph : A graph is said to be weighted if there are some non negative values assigned to each edge of the graph. The value is equal to the length between two vertices. Weighted graph is also called a network. A weighted graph is shown in figure 14.2.

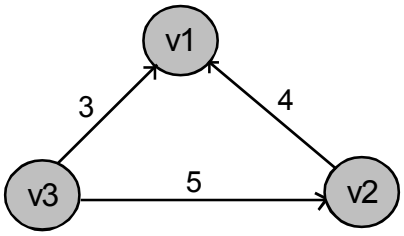


Fig. 14.2 : A weighted graph

Adjacent nodes : When there is an edge from one node to another then these nodes are called adjacent nodes.

Incidence : In an undirected graph the edge between v1 and v2 is incident on node v1 and v2.

Walk : A walk is defined as a finite alternating sequence of vertices and edges, beginning and ending with vertices, such that each edge is incident with the vertices preceding and following it.

Closed walk : A walk which is to begin and end at the same vertex is called close walk. Otherwise it is an **open walk.**

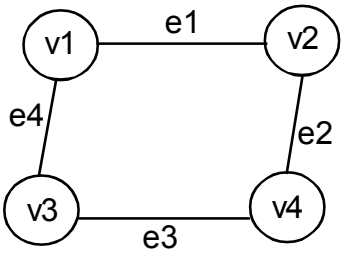


Fig. 14.3 Closed walk present in Graph

If e1, e2, e3 and e4 be the edges of pair of vertices (v1, v2), (v2, v4), (v4, v3) and (v3, v1) respectively, then v1 e1 v2 e2 v4 e3 v3 e4 v1 can be its closed walk or circuit.

Path : An open walk in which no vertex appears more than once is called a path.

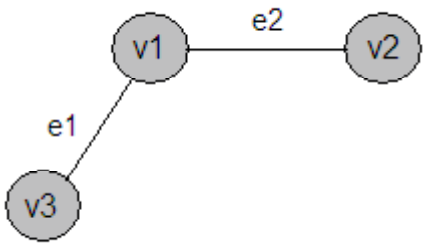


Fig. 14.4 A path

If e1 and e2 be the two edges between the pair of vertices (v1, v3) and (v1, v2) respectively, then v3 e1 v1 e2 v2 is its path.

Length of a path : The number of edges in a path is called the length of that path. In the following diagram the length of the path is 3.

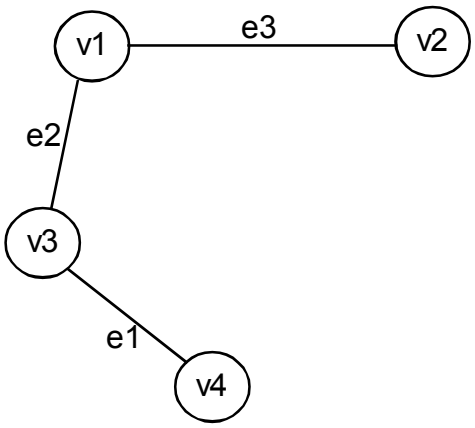


Fig. 14.5 : An open walk graph

Circuit : A closed walk in which no vertex (except the initial and the final vertex) appears more than once is called a circuit.

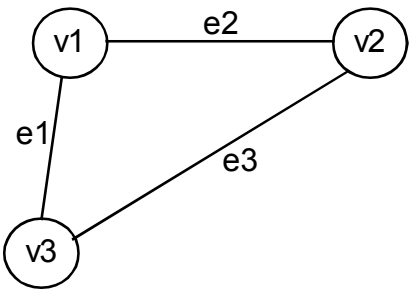


Fig. 14.6 : A circuit having three vertices and three edges

Sub Graph : A graph S is said to be a subgraph of a graph G if all

the vertices and all the edges of S are in G, and each edge of S has the same end vertices in S as in G.

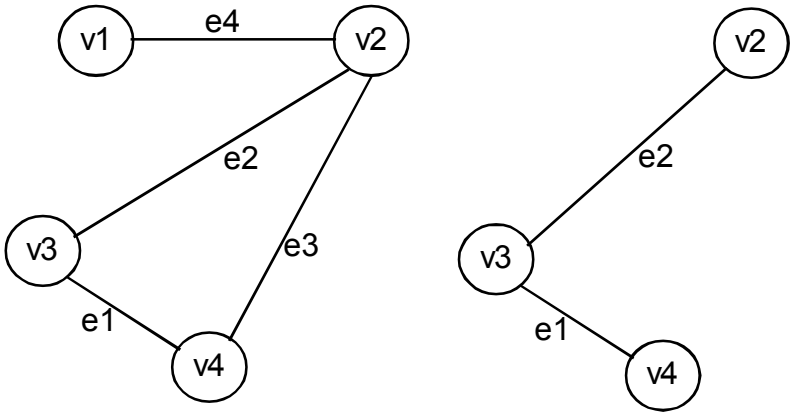


Fig. 14.7 : Graph G **Graph S**

In the above figure, S is a subgraph of the graph G, since all the vertices and edges in the graph S are also the vertcies and edges of G.

Connected Graph : A graph G is said to be connected if there is at least one path between every pair of vertices in G (Fig 14.8). Otherwise, the graph is disconnected.

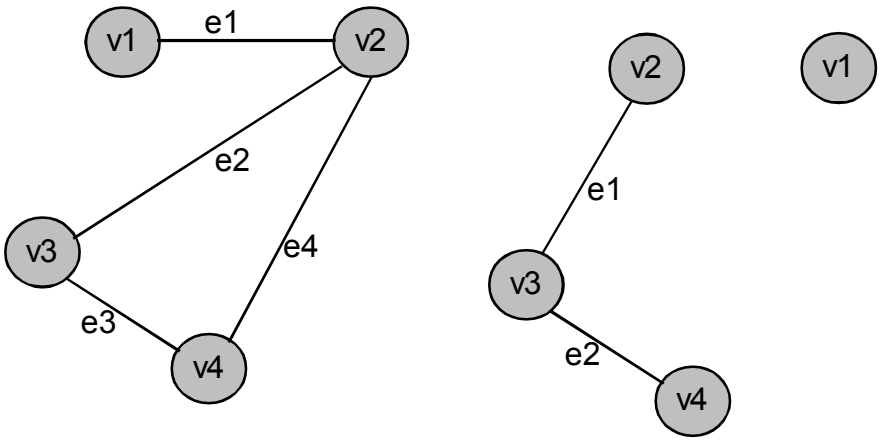


Fig.14.8 : A connected graph G **Fig.14.9 : A disconnected graph G**

The graph in fig 14.9 is disconnected because the vertex v1 is not connected with the other vertices of the graph.

Degree : In an undirected graph, the number of edges connected to a node is called the degree of that node or we can say that the degree of a

node is the number of edges incident on it. In the graph shown in fig. 14.8, degree of vertex v1 is 1, degree of vertex v2 is 3, degree of v3 and v4 is 2. Degree of the vertex v1 of the graph shown in fig. 14.9 is zero.

Indegree : The indegree of a node is the number of edges connected to that node or, in other words, edges incident to it.

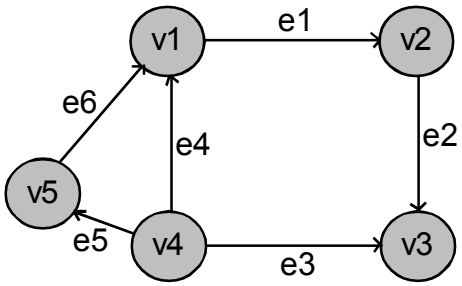


Fig. 14.10 : A graph

In the figure 14.10, indegree of vertices v1, v3 is 2, indegree of vertices v2, v5 is 1 and indegree of v4 is zero.

Outdegree : The outdegree of a node (or vertex) is the number of edges going outside from that node or in other words the edges incident from it. In the graph shown in figure 14.10, the outdegree of vertex v4 is 3, outdegree of vertex v1 is 1.

14.5 DIFFERENT TYPES OF GRAPH

There are different types of graphs. Some of them are :

1. Digraph
2. Undirected graph
3. Complete graph
4. Regular graph
5. Cyce graph
6. Acyclic graph
7. Multigraph
8. Euler graph

Let us look at some of the different types of graph is detail.

Directed Graph (Digraph) : A directed graph or digraph G consists

of a vertex set V(G) and an edge set E(G), where each edge is an ordered pair of vertices. A simple digraph is a digraph in which each ordered pair of vertices occurs at the most once as an edge. In this type of graph each edge has direction, meaning, (v1, v2) and (v2, v1) will represent different edges.

In the figure 14.11, V(G) = {A, B, C, D}, E(G) = {(A, B), (B, D), (D, A), (A, C), (C, D)}. The graph has 4 vertices and 6 edges.

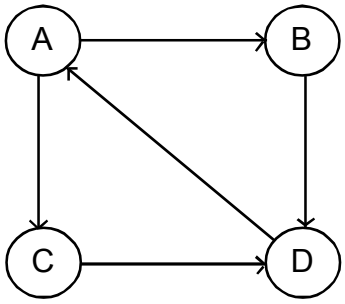


Fig. 14.11 : Directed Graph

Undirected Graph : A graph containing unodered pair of nodes is termed as undirected graph. If there is an edge between vertex v0 and v1 then it can be represented as (v0, v1) or (v1, v0).

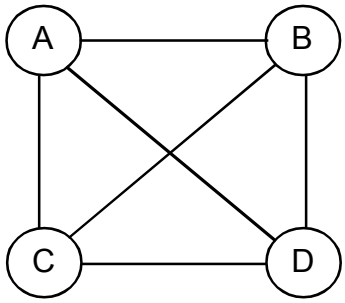


Fig. 14.12 : Undirected Graph

In the graph shown in fig.14.12, V(G) = {A, B, C, D} and E(G) = {(A, B), (A, C), (A, D), (B, C), (B, D), (D, C)}. The graph has 4 vertices and 6 edges.

Complete Graph : A graph in which any node is adjacent to all other nodes present in the graph is known as a complete graph. An undirected graph contains total edges equal to = n(n–1)/2 where n is the number of vertices present in the graph. The figure 14.13 shows a complete graph.

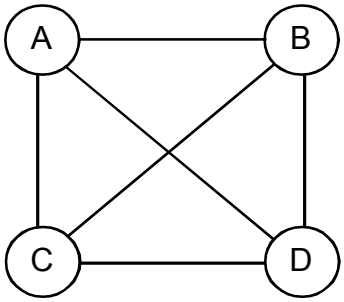


Fig. 14.13 : A complete graph

Regular Graph : Regular graph is the graph in which nodes are adjacent to each other, i.e., each node is accessible from any other node.

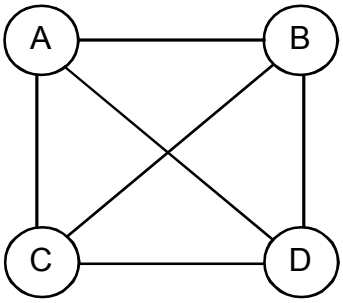


Fig.14.14 : A regular graph

Cycle Graph : A graph having cycle is called cycle graph. In this case the first and last nodes are the same. A closed simple path is a cycle.

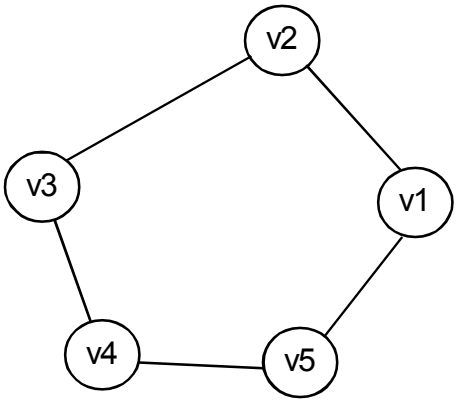


Fig. 14.15 : A cycle graph

Acyclic Graph : A graph without cycle is called acyclic graph.

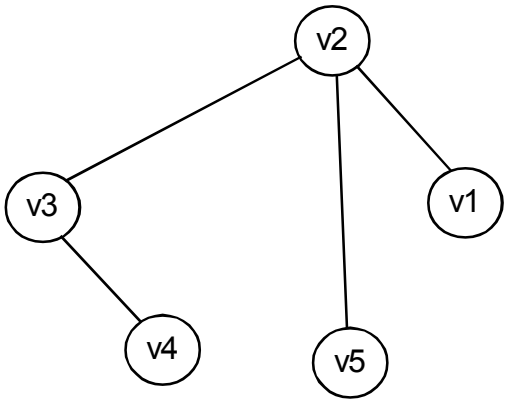


Fig. 14.16 : A acycle graph

Multigraph : A graph in which more than one edge is used to join the vertices is called multigraph.

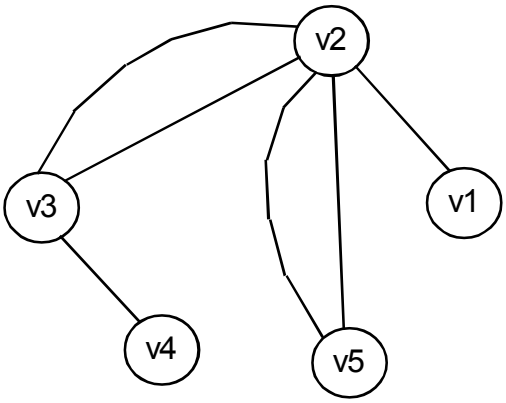


Fig. 14.17 : A multigraph

Euler Graph : If some closed walk in a graph contains all the edges of the graph,then the walk is called an Euler line and the graph is called an Euler graph.

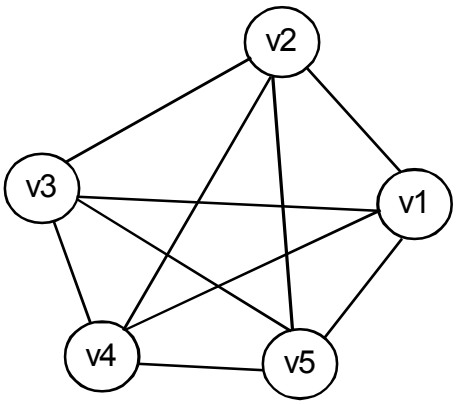


Fig. 14.18 : An Euler graph

14.6 REPRESENTATION OF GRAPH

There are two standard ways of storing graph G in the memory of a computer. One way, called the sequential representation of G, by means of its adjacency matrix A. The other way called the linked representation of G, is by means of linked list.

14.6.1 Sequential Representation of Graph

Adjacency Matrix : Suppose G is a simple directed graph with m nodes, and suppose the nodes of G have been ordered and are called $v_1, v_2, ..., ..., v_m$. Then the adjacency matrix $A = (a_{ij})$ of the graph G is the m x m matrix defined as follows:

$$a_{ij} = \begin{cases} 1, \text{if } v_i \text{ is adjacent to } v_j, \text{that is, if there is an edge } (v_i, v_j) \\ 0, \text{otherwise.} \end{cases}$$

Such a matrix A , which contains entries of only 0 and 1, is called a *bit matrix* or a *Boolean matrix*. For example: for the graph G in the following figure 14.19, we have the adjacency matrix A.

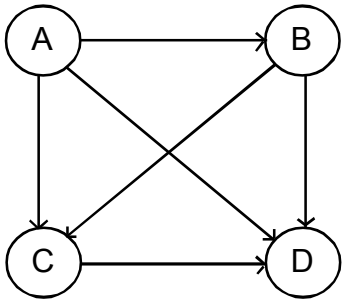


Fig. 14.19 : Directed Graph G

The equivalent adjacency matrix A is :

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

14.6.2 Linked Representation of Graph

In this type of representation a graph G is usually represented in memeory by a linked representation. It is also called an adjacency list. Let us consider a graph G as shown below :

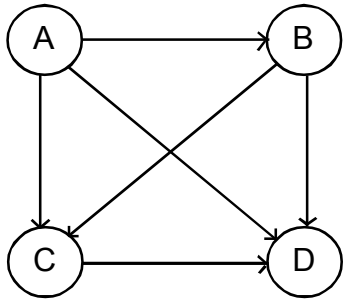


Fig. 14.20 : Directed Graph G

The graph G can be represented by its adjacency list, which is its list of adjacent nodes, also called its successors or neighbours.

Nodes	Adjacency List
A	B, C, D
B	C, D
C	D
D	

Table 14.1 : Adjacency list of graph G.



CHECK YOUR PROGRESS

Q.A. Select the appropriate option :

1. Graph can be implemented using :

i) arrays; ii) linked list; iii) stack; iv) queue

a) i), ii) and iv) b) i), ii) and iii)

c) ii) and iii) d) i) and ii)

2. Adjacency matrix for a diagraph is
   a) unit matrix b) anti symmetric
   c) symmetrix matrix d) none of these
3. A walk which is to begin and end at the same vertex is called
   a) close walk b) open walk
   c) path d) none of these

14.7 TRAVERSAL IN GRAPHS

When dealing with graphs, one fundamental issue is to traverse the graph. In other words, to visit each vertex. To solve this problem, many algorithms exist. Two of them will be presented here : ***Depth First Search (DFS)*** and ***Breadth First Search (BFS)***.

14.7.1 Depth First Search

In this method, a node from graph is taken as a starting node. Traverse through the possible paths of the starting node. When the last node of the graph is obtained and path is not available from the node, then control returns to previous node. This process is implemented using stack data structure. Let us consider the following graph G :

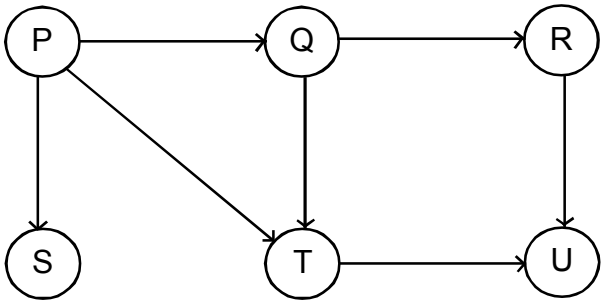


Fig. 14.21 : A model graph G

Let us apply Depth First Search traversal method, and consider P as starting node. Then, we traverse the node adjacent to P and we will get Q and then R (adjacent to Q) and U (adjacent to R).The traversal will be P-> Q-> R-> U.

The search is always carried in forward direction. After reaching to U, we reach the end of the path and further movement in forward direction is not possible. Hence, the control goes to the previous node and again traverses through the available paths for non-traversed nodes. We get the node R and it has no nontraversed node. Hence we go back to Q and it gives T. The node T gives U, but it is already visited. Therefore, control in reverse direction checks all the nodes. Then it goes back to P which gives node S. Hence the sequence of traversal will be P->Q->R->U->T->S.

The general idea behind a depth-first search beginning at a starting node A is as follows.

First we take the starting node A. Then we examine each node N along a path P which begins at the node A. That means we process a neighbor node of the node A. Then again a neighbor of a neighbor and proceed and so on. After coming to a “dead end” which is to the end of the path P, we backtrack on P until we can continue along another path P and so on. The algorithm is similar to the Breadth-First Search except that here we take a stack instead of queue. In the algorithm given below, a field STAT is used to tell the current status of a node.

Algorithm of DFS : Algorithm for Depth-First Search on a graph G beginning at a starting node A

Step 1. Initialize all nodes to the ready state [STAT=1]

Step 2. Push the starting node A onto STACK and change its status to the waiting state [STAT=2]

Step 3. Repeat Step 4 and Step 5 until STACK is empty.

Step 4. Pop the top node N of STACK. Process N and change its status to the processed state [STAT=3]

Step 5. Push onto STACK all the neighbors of N that are still in the ready state [STAT=1] and change their status to the waiting

state [STAT=2] [End of Step 3 loop]

Step 6. Exit.

The above algorithm will process only those nodes which are reachable from the starting node A.

14.7.2 Breadth First Search

This is one of the popular methods of traversing graph. This method uses the queue data structure for traversing nodes of the graph. Any node of the graph can act as a beginning node. Using any node as starting node, all other nodes of the graph are traversed. This process is repeated till unvisited nodes are available.

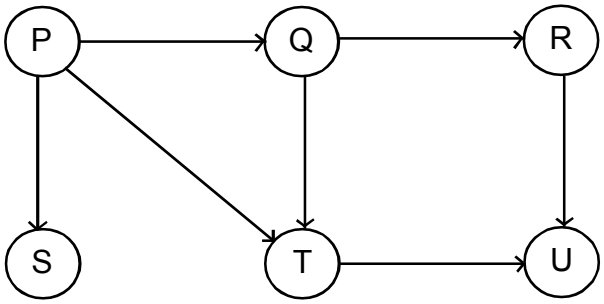


Fig. 14.22 : A model graph G

Let us apply the Breadth First Search traversal method in the above graph G. Take the P as a starting node and all the adjacent nodes of P are traversed, that is Q, T and S. The traversal of nodes can be carried in any sequence. For example, the sequence of traverse of nodes is Q,S and T. The traversal will be P->Q->S->T.

First, all the nodes neighbouring Q are traversed, then neighbouring nodes of S and finally T are taken into account. The adjacent node of Q is R and T is U. Simillarly, the adjacent node of T is U and S does not have any adjacent node. Hence, in this step the traversal now should be in the following way: P->Q->S->T->R->U.

The general idea behind a breadth-first search beginning at a starting node A is as follows :

First, we examine the starting node A. Then we verify all the neighbors of A. Then we examine all the neighbors of the neighbors

of A and proceed so on. Here actually we need to keep track of the neighbors of a node, and we need to guarantee that no node is processed more than once. This is done by using queue to hold the nodes that are waiting to be processed, and by using a field STAT which keep track of the current status of any node. The algorithm is as given below:

Algorithm of Breadth First Search : Algorithm for a breadth-first search on a graph G beginning at a starting node A

Step 1. Initialize all the nodes to the ready state [STAT=1]
Step 2. Put the starting node A in QUEUE and change its status to the waiting state [STAT=2]
Step 3. Repeat Step 4 and Step 5 until QUEUE is empty.
Step 4. Remove the front node N of QUEUE. Process N and change the status of N to the processed state [STAT=3]
Step 5. Add to the rear of QUEUE all the neighbors of N that are in the steady state [STAT=1] and change their status to the waiting state [STAT=2] [End of Step3 loop]
Step 6. Exit.

The above algorithm will process only those nodes which are reachable from the starting node A.

In an unweighted graph, you can consider all edges to have an equal weight of 1. BFS visits all the vertices on the same level before proceeding to a “deeper” level, so all vertices newly visited on a level will be of the same distance from the starting vertex. Further, since BFS proceeds level by level with distances of 0, 1, 2 etc, it will reach a vertex via the shortest path to that vertex.



CHECK YOUR PROGRESS

Q.B. Select the appropriate option :

1. Breadth First Search is a method to traverse
   a) all successors of a visited node before any successors of any of those succssors
   b) a single path of the graph as far as it can go.
   c) the graph using shorted path
   d) none of these.
2. Identify the correct statements about DFS traversal
   i) It can be used to determine whether a graph is acyclic or not.
   ii) It identifies the connected component of an undirected graph.
   iii) Traverses a single path of the graph until it visits a node with no successor.
   a) i) and iii) b) ii) and iii)
   c) i) and ii) d) i), ii) and iii)
3. A vertex of degree one is called _____
   a) pendant b) isolated vertex
   c) null vertex d)coloured vertex
4. A graph in which all nodes are of equal degree is known as
   a) multigraph b) non regular graph
   c) regular graph d) complete graph
5. A simple graph with n vertices and k components can have at the most _____
   a) n edges
   b) n-k edges
   c) (n–k)(n–k–1)/2 edges
   d) (n–k)(n–k+1)/2 edges
6. If there exists at least one path between every pair of vertices in a graph, the graph is known as.

a) complete graph	b) disconnected graph
c) connected graph	d) euler graph.



14.8 LET US SUM UP

- A graph is defined as a set of vertices and edges.
- *Directed Grpah :* A directed graph or digraph G consists of a vertex set V(G) and an edge set E(G), where each edge is an ordered pair of vertices.
- *Undirected Graph :* Graph containing unordered pair of nodes is termed as undirected graph.
- *Complete Graph :* A graph in which any V node is adjacent to all other nodes present in the graph is known as a complete graph.
- *Adjacency Matrix :* Suppose G is a simple directed graph with m nodes, and suppose the nodes of G have been ordered and are called $v_1, v_2, ..., ..., v_m$, then the adjacency matrix A = (a_{ij}) of the graph G is the m x m matrix defined as follows:

$$a_{ij} = \begin{cases} 1, \text{if } v_i \text{ is adjacent to } v_j, \text{that is, if there is an edge } (v_i, v_j) \\ 0, \text{otherwise.} \end{cases}$$

- *Linked Representation of Graph:* In this type of representation a graph G is usually represented in memeory by a linked representation. It is also called an adjacent list.
- *Traversal in Graphs:* Two type of traversal methods. They are :
  i) Depth-First Search (abbr. DFS) and
  ii) Breadth-First Search (BFS).
- *Depth First Search :* In this method, a node from graph is taken as a starting node. Traverse through all the possible paths of the starting node. When the last node of the graph is obtained and path is not available from the node; then control returns to previous node. This process is implemented using stack data structure.

- *Breadth First Search :* This method uses the queue data structure for traversing nodes of the graph. Any node of the graph can act as a beginning node. Using any node as starting node, all other nodes of the graph are traversed. We repeat the process till unvisited nodes are available.



14.9 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. A : 1. d) i) and ii); 2. c) symmetrix matrix; 3. b) open walk.

Ans. to Q. No. B : 1. a); 2. d); 3. a); 4. c); 5. d); 6. c).



14.10 FURTHER READING

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

2. Seymour, L. (2010). *Data Structures with C*. Tata Mc Graw- Hill.



14.11 MODEL QUESTIONS

Q.1. Distinguish between the directed and undirected graphs.

Q.2. What are complete graphs ?

Q.3. Distinguish between Breadth First and Depth First search traversals in a graph.

Q.4. How are graphs implemented in computer memory ?

Q.5. Distingish between tree and graph.

Q.6. Write short notes on: (a) Breadth First Search (BFS) and (b) Depth First Search (DFS).

Q.7. Define degree, indegree and outdegree of graph. Explain with examples.