# UNIT 4 STACKS

**Structure** **Page Nos.**



## 4.0 INTRODUCTION

One of the most useful concepts in computer science is stack. In this unit, we shall examine this simple data structure and see why it plays such a prominent role in the area of programming. There are certain situations when we can insert or remove an item only at the beginning or the end of the list.

A stack is a linear structure in which items may be inserted or removed only at one end called the *top of the stack*. A stack may be seen in our daily life, for example, *Figure 4.1* depicts a stack of dishes. We can observe that any dish may

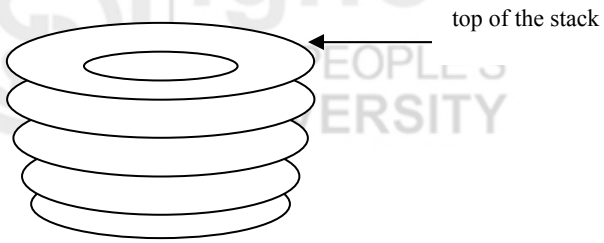



**Figure 4.1: A stack of dishes**

be added or removed only from the top of the stack. It concludes that the item added last will be the item removed first. Therefore, stacks are also called LIFO (Last In First Out) or FILO (First In Last Out) lists. We also call these lists as "piles" or "push-down list".

Generally, two operations are associated with the stacks named Push & Pop.

- *Push* is an operation used to insert an element at the top.
- *Pop* is an operation used to delete an element from the top

**Example 4.1**

Now we see the effects of push and pop operations on to an empty stack. *Figure 4.2(a)* shows (i) an empty stack; (ii) a list of the elements to be inserted on to stack;

and (iii) a variable top which helps us keep track of the location at which insertion or removal of the item would occur.

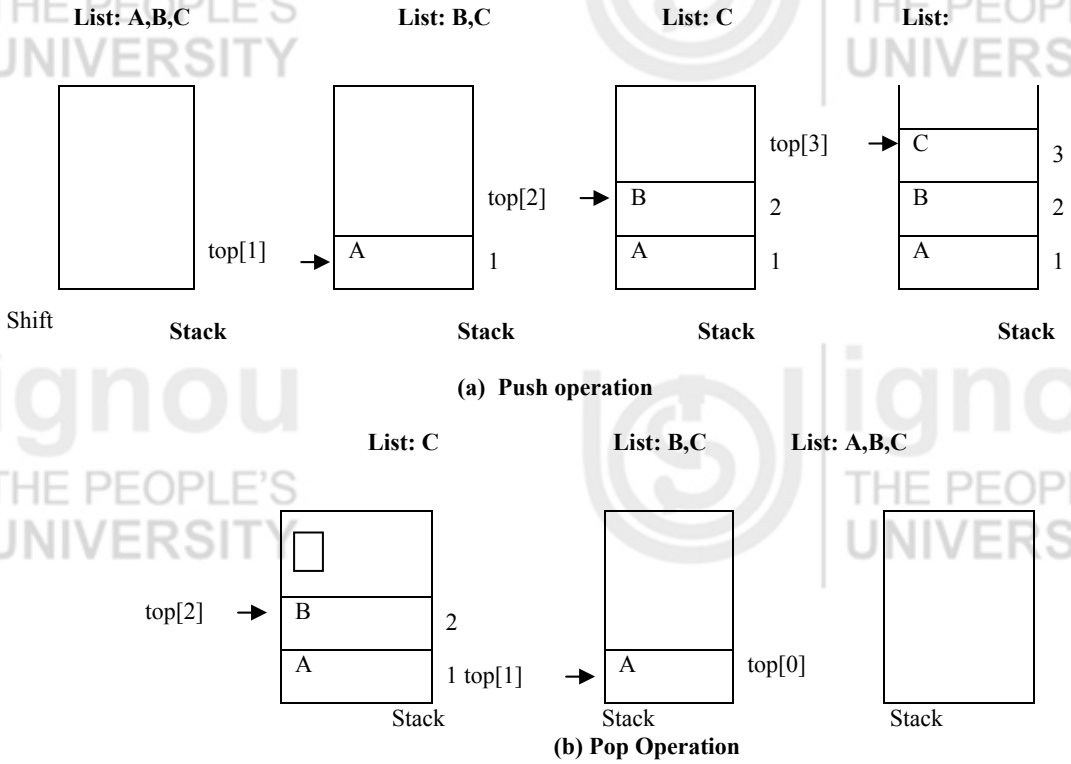



**Figure 4.2: Demonstration of (a) Push operation, (b) Pop operation**

Initially in *Figure 4.2(a)*, top contains 0, implies that the stack is empty. The list contains three elements, A, B &C. In *Figure 4.2(b),* we remove an element A from the list of elements, push it on to stack. The value of top becomes 1, pointing to the location of the stack at which A is stored.

Similarly, we remove the elements B & C from the list one by one and push them on to the stack. Accordingly, the value of the **top** is incremented. *Figure 4.2(a)* explains the pushing of B and C on to stack. The **top** now contains value 3 and pointing to the location of the last inserted element C.

On the other hand, *Figure 4.2(b)* explains the working of pop operation. Since, only the top element can be removed from the stack, in *Figure 4.2(b),* we remove the top element C (we have no other choice). C goes to the list of elements and the value of the top is decremented by 1. The top now contains value 2, pointing to B (the top element of the stack). Similarly, in *Figure 4.2(b),* we remove the elements B and A from the stack one by one and add them to the list of elements. The value of top is decremented accordingly.

There is no upper limit on the number of items that may be kept in a stack. However, if a stack contains a single item and the stack is popped, the resulting stack is called empty stack. The pop operation cannot be applied to such stacks as there is no element to pop, whereas the push operation can be applied to any stack.

## 4.1 OBJECTIVES

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

- understand the concept of stack;
- implement the stack using arrays;
- implement the stack using linked lists;
- implement multiple stacks, and
- give some applications of stack.

## 4.2 ABSTRACT DATA TYPE-STACK

Conceptually, the **stack** abstract data type mimics the information kept in a pile on a desk. Informally, we first consider materials on a desk, where we may keep separate stacks for bills that need paying, magazines that we plan to read, and notes we have taken. We can perform several operations that involve a stack:

- start a new stack;
- place new information on the top of a stack;
- take the top item off of the stack;
- read the item on the top; and
- determine whether a stack is empty. (There may be nothing at the spot where the stack should be).

When discussing these operations, it is conventional to call the addition of an item to the top of the stack as a **push operation** and the deletion of an item from the top as a **pop operation**. (These terms are derived from the working of a spring-loaded rack containing a stack of cafeteria trays. Such a rack is loaded by pushing the trays down on to the springs as each diner removes a tray, the lessened weight on the springs causes the stack to pop up slightly).

## 4.3 IMPLEMENTATION OF STACK

Before programming a problem solution that uses a stack, we must decide how to represent a stack using the data structures that exist in our programming language. Stacks may be represented in the computer in various ways, usually by means of a one-way list or a linear array. Each approach has its advantages and disadvantages. A stack is generally implemented with two basic operations – push and pop. **Push** means to insert an item on to stack. The push algorithm is illustrated in *Figure 4.3(a)*. Here, **tos** is a pointer which denotes the position of top most item in the stack. Stack is represented by the array **arr** and **MAXSTACK** represents the maximum possible number of elements in the stack. The pop algorithm is illustrated in *Figure 4.3(b).*

```
Step 1:  [Check for stack overflow]
                    if tos >=MAXSTACK
                    print "Stack overflow" and exit
Step 2:  [Increment the pointer value by one]
                    tos=tos+1
Step 3:  [Insert the item]
                    arr[tos]=value
Step 4:  Exit
```

**Figure 4.3(a): Algorithm to push an item onto the stack**

The pop operation removes the topmost item from the stack. After removal of top most value **tos** is decremented by 1.

```
Step 1:  [Check whether the stack is empty]
                    if tos = 0
                    print "Stack underflow" and exit

Step 2:  [Remove the top most item]
                    value=arr[tos]
                    tos=tos-1

Step 3:  [Return the item of the stack]
                    return(value)
```

**Figure 4.3(b): Algorithm to pop an element from the stack**

### 4.3.1 Implementation of Stack Using Arrays

A Stack contains an ordered list of elements and an array is also used to store ordered list of elements. Hence, it would be very easy to manage a stack using an array. However, the problem with an array is that we are required to declare the size of the array before using it in a program. Therefore, the size of stack would be fixed. Though an array and a stack are totally different data structures, an array can be used to store the elements of a stack. We can declare the array with a maximum size large enough to manage a stack. Program 4.1 implements a stack using an array.

```
#include<stdio.h>

int choice, stack[10], top, element;

void menu();
void push();
void pop();
void showelements();

void main()
{ choice=element=1;
 top=0;
 menu();
}

void menu()
{
        printf("Enter one of the following options:\n");
        printf("PUSH 1\n POP 2\n SHOW ELEMENTS 3\n  EXIT 4\n");
        scanf("%d", &choice);
        if (choice==1)
        {
         push(); menu();
        }
          if (choice==2)
        {
         pop();menu();
```

```
        }
          if (choice==3)
                                {
                                        showelements(); menu();
                                }
        }


void push()
{
        if (top<=9)
        {
                printf("Enter the element to be pushed to stack:\n");
                scanf("%d", &element);
                stack[top]=element;
                ++top;

        }
        else
        {
        printf("Stack is full\n");
        }
        return;
}

void pop()
{
        if (top>0)
        {
                --top;
                element = stack[top];
                printf("Popped element:%d\n", element);
         }
        else
        {
        printf("Stack is empty\n");
        }
        return;

}

void showelements()
{
        if (top<=0)
                printf("Stack is empty\n");

        else
                for(int i=0; i<top; ++i)
                        printf("%d\n", stack[i]);


}
```

**Program 4.1: Implementation of stack using arrays**

**Explanation**

The size of the stack was declared as 10. So, stack cannot hold more than 10 elements. The main operations that can be performed on a stack are push and pop. How ever, in a program, we need to provide two more options , namely, *showelements* and *exit*. *showelements* will display the elements of the stack. In case, the user is not interested to perform any operation on the stack and would like to get out of the program, then s/he will select *exit* option. It will log the user out of the program. *choice* is a variable which will enable the user to select the option from the push, pop, showelements and exit operations. *top* points to the index of the free location in the stack to where the next element can be pushed. *element* is the variable which accepts the integer that has to be pushed to the stack or will hold the top element of the stack that has to be popped from the stack. The array *stack* can hold at most 10 elements. *push* and *pop* will perform the operations of pushing the element to the stack and popping the element from the stack respectively.

### 4.3.2 Implementation of Stack Using Linked Lists

In the last subsection, we have implemented a stack using an array. When a stack is implemented using arrays, it suffers from the basic limitation of an array – that is, its size cannot be increased or decreased once it is declared. As a result, one ends up reserving either too much space or too less space for an array and in turn for a stack. This problem can be overcome if we implement a stack using a linked list. In the case of a linked stack, we shall push and pop nodes from one end of a linked list.
The stack, as linked list is represented as a singly connected list. Each node in the linked list contains the data and a pointer that gives location of the next node in the list. Program 4.2 implements a stack using linked lists.

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

/* Definition of the structure node */

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

/* Definition of push function */

void  push(node **tos,int item)

{

   node *temp;
   temp=(node*)malloc(sizeof(node));      /* create a new node dynamically */
   if(temp==NULL)                         /* If sufficient amount of memory is */
    {                                     /* not available, the function malloc will */
     printf("\nError: Insufficient Memory Space");      /* return NULL to temp */
     getch();
     return;
    }

   else                                   /* otherwise*/

   {

       temp->data=item;   /* put the item in the data portion of  node*/


        temp->next=*tos;                  /*insert this node at the front of the stack */
       *tos=temp;                         /* managed by linked list*/
```

```
        }
}                                   /*end of function push*/

        /* Definition of pop function */
int pop(node **tos)
{
     node *temp;
     temp=*tos;
     int item;
     if(*tos==NULL)
          return(NULL);
    else
          {
            *tos=(*tos)->next;              /* To pop an element from stack*/
             item=temp->data;                /* remove the front node of the */
             free(temp);                     /* stack managed by L.L*/
             return (item);
          }
}  /*end of function pop*/

/* Definition of display function */
void display(node *tos)

{
 node *temp=tos;
 if(temp==NULL)                 /* Check whether the stack is empty*/
   {
     printf("\nStack is empty");
     return;
   }
      else
          {
            while(temp!=NULL)
            {
               printf("\n%d",temp->data);    /* display all the values of the stack*/
               temp=temp->next;              /* from the front node to the last node*/
             }
          }

  }                                          /*end of function display*/


/* Definition of main function */
void main()

{
  int item, ch;
  char choice='y';
  node *p=NULL;
  do
    {
          clrscr();
          printf("\t\t\t\t*****MENU*****");
```

```
        printf("\n\t\t\t1. To PUSH an element");
        printf("\n\t\t\t2. To POP an element");
        printf("\n\t\t\t3. To DISPLAY the elements of stack");
        printf("\n\t\t\t4. Exit");
        printf("\n\n\n\t\t\tEnter your choice:-");
        scanf("%d",&ch);
        switch(ch)
                {
                case 1:
                        printf("\n Enter an element which you want to push ");
                        scanf("%d",&item);
                        push(&p,item);
                        break;
                case 2:
                        item=pop(&p);
                        if(item!=NULL);
                        printf("\n Detected item is%d",item);
                        break;
                case 3:
                        printf("\nThe elements of stack are");
                        display(p);
                        break;
                case 4:
                        exit(0);

                }       /*switch closed */
                printf("\n\n\t\t Do you want to run it again y/n");
                scanf("%c",&choice);
    } while(choice=='y');

}
                                        /*end of function main*/
```

**Program 4.2: Implementation of Stack using Linked Lists**

Similarly, as we did in the implementation of stack using arrays, to know the working of this program, we executed it thrice and pushed 3 elements (10, 20, 30). Then we call the function display in the next run to see the elements in the stack.

**Explanation**

Initially, we defined a structure called *node*. Each node contains two portions, data and a pointer that keeps the address of the next node in the list. The *Push* function will insert a node at the front of the linked list, whereas *pop* function will delete the node from the front of the linked list. There is no need to declare the size of the stack in advance as we have done in the program where in we implemented the stack using arrays since we create nodes dynamically as well as delete them dynamically. The function *display* will print the elements of the stack.

### ☞ Check Your Progress 1

1) State True or False.

   (a) Stacks are sometimes called FIFO lists.

   (b) Stack allows Push and Pop from both ends.

   (c) TOS (top of the stack) gives the bottom most element in the stack.

2) Comment on the following.

(a) Why is the linked list representation of the stack better than the array representation of the stack?

(b) Discuss the underflow and overflow problem in stacks.

## 4.4 ALGORITHMIC IMPLEMENTATION OF MULTIPLE STACKS

So far, now we have been concerned only with the representation of a single stack. What happens when a data representation is needed for several stacks? Let us see an array X whose dimension is m. For convenience, we shall assume that the indexes of the array commence from 1 and end at m. If we have only 2 stacks to implement in the same array X, then the solution is simple.

Suppose A and B are two stacks. We can define an array stack A with $n_1$ elements and an array stack B with $n_2$ elements. Overflow may occur when either stack A contains more than $n_1$ elements or stack B contains more than $n_2$ elements.

Suppose, instead of that, we define a single array stack with $n = n_1 + n_2$ elements for stack A and B together. See the *Figure 4.4* below. Let the stack A "grow" to the right, and stack B "grow" to the left. In this case, overflow will occur only when A and B together have more than $n = n_1 + n_2$ elements. It does not matter how many elements individually are there in each stack.

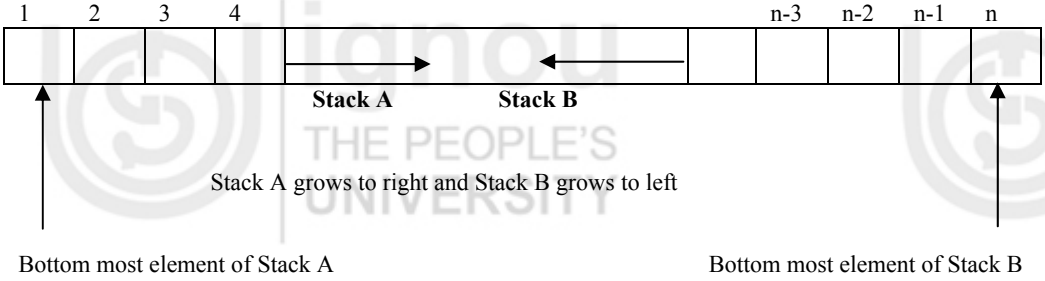
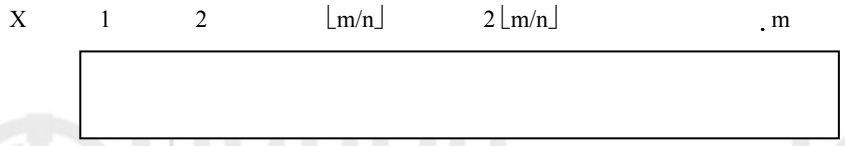



**Figure 4.4: Implementation of multiple stacks using arrays**

But, in the case of more than 2 stacks, we cannot represent these in the same way because a one-dimensional array has only two fixed points X(1) and X(m) and each stack requires a fixed point for its bottom most element. When more than two stacks, say **n**, are to be represented sequentially, we can initially divide the available memory X(1:m) into **n** segments. If the sizes of the stacks are known, then, we can allocate the segments to them in proportion to the expected sizes of the various stacks. If the sizes of the stacks are not known, then, X(1:m) may be divided into equal segments. For each stack i, we shall use BM (i) to represent a position one less than the position in X for the bottom most element of that stack. TM(i), $1 \leq i \leq n$ will point to the topmost element of stack i. We shall use the boundary condition BM (i) = TM (i) iff the $i^{th}$ stack is empty (refer to *Figure 4.5*). If we grow the $i^{th}$ stack in lower memory indexes than the $i+1^{st}$ stack, then, with roughly equal initial segments we have
BM (i) = TM (i) = $\lfloor m/n \rfloor (i - 1)$, $1 \leq i \leq n$, as the initial values of BM (i) and TM (i).



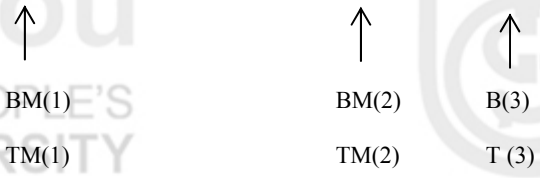



**Figure 4.5: Initial configuration for n stacks in X(1:m)**

All stacks are empty and memory is divided into roughly equal segments.

*Figure 4.6* depicts an algorithm to add an element to the $i^{th}$ stack. *Figure 4.7* depicts an algorithm to delete an element from the $i^{th}$ stack.

```
    ADD(i,e)
Step1: if TM (i)=BM (i+1)
        Print "Stack is full" and exit
Step2: [Increment the pointer value by one]
            TM (i)← TM (i)+1
             X(TM (i))← e
Step3: Exit
```

**Figure 4.6: Algorithm to add an element to $i^{th}$ stack**

//delete the topmost elements of stack i.

```
DELETE(i,e)
Step1: if TM (i)=BM (i)
        Print "Stack is empty" and exit
Step2: [remove the topmost item]
            e← X(TM (i))
             TM (i)←TM(i)-1
Step3: Exit
```

**Figure 4.7: Algorithm to delete an element from $i^{th}$ stack**

## 4.5 APPLICATIONS

Stacks are frequently used in evaluation of arithmetic expressions. An arithmetic expression consists of operands and operators. Polish notations are evaluated by stacks. Conversions of different notations (Prefix, Postfix, Infix) into one another are performed using stacks. Stacks are widely used inside computer when recursive functions are called. The computer evaluates an arithmetic expression written in infix notation in two steps. First, it converts the infix expression to postfix expression and then it evaluates the postfix expression. In each step, stack is used to accomplish the task.

## 4.6 SUMMARY

In this unit, we have studied how the stacks are implemented using arrays and using liked list. Also, the advantages and disadvantages of using these two schemes were discussed. For example, when a stack is implemented using arrays, it suffers from the basic limitations of an array (fixed memory). To overcome this problem, stacks are implemented using linked lists. This unit also introduced learners to the concepts of multiple stacks. The problems associated with the implementation of multiple stacks are also covered.

**☞ Check Your Progress 2**

1) Multiple stacks can be implemented using _____.
2) _____ are evaluated by stacks.
3) Stack is used whenever a _____ function is called.

## 4.7 SOLUTIONS / ANSWERS

**Check Your Progress 1**

1) (a) False (b) False
(c) False

**Check Your Progress 2**

1) Arrays or Pointers
2) Postfix expressions
3) Recursive

## 4.8 FURTHER READINGS

1. *Data Structures Using C and C++*, Yedidyah Langsam,Moshe J. Augenstein,Aaron M Tenenbaum, Second Edition,PHI publications.

2. *Data Structures,* Seymour Lipschutz, Schaum's Outline series, Mc GrawHill.

**Reference Websites**

http://www.cs.queensu.ca

# UNIT 5 QUEUES

**Structure** **Page Nos.**



## 5.0 INTRODUCTION

Queue is a linear data structure used in various applications of computer science. Like people stand in a queue to get a particular service, various processes will wait in a queue for their turn to avail a service. In computer science, it is also called a FIFO (first in first out) list. In this chapter, we will study about various types of queues.

## 5.1 OBJECTIVES

After going through this unit, you should be able to

- define the queue as an abstract data type;
- understand the terminology of various types of queues such as simple queues, multiple queues, circular queues and dequeues, and
- get an idea about the implementation of different types of queues using arrays and linked lists.

## 5.2 ABSTRACT DATA TYPE-QUEUE

An important aspect of Abstract Data Types is that they describe the properties of a data structure without specifying the details of its implementation. The properties can be implemented independent of any implementation in any programming language.

*Queue* is a collection of elements, or items, for which the following operations are defined:

createQueue(Q) : creates an empty queue Q;
isEmpty(Q): is a boolean type predicate that returns ``true'' if Q exists and is empty, and returns ``false'' otherwise;
addQueue(Q,item) adds the given item to the queue Q; and
deleteQueue (Q, item) : delete an item from the queue Q;
next(Q) removes the least recently added item that remains in the queue Q, and returns it as the value of the function;

isEmpty (createQueue(Q)) : is always true, and
deleteQueue(createQueue(Q)) : error

The primitive isEmpty(Q) is required to know whether the queue is empty or not, because calling next on an empty queue should cause an error. Like stack, the situation may be such when the queue is "full" in the case of a finite queue. But we avoid defining this here as it would depend on the actual length of the Queue defined in a specific problem.

The word "queue" is like the queue of customers at a counter for any service, in which customers are dealt with in the order in which they arrive i.e. first in first out (FIFO) order. In most cases, the first customer in the queue is the first to be served.

As pointed out earlier, Abstract Data Types describe the properties of a structure without specifying an implementation in any way. Thus, an algorithm which works with a "queue" data structure will work wherever it is implemented. Different implementations are usually of different efficiencies.

## 5.3 IMPLEMENTATION OF QUEUE

A physical analogy for a queue is a line at a booking counter. At a booking counter, customers go to the *rear* (end) of the line and customers are attended to various services from the *front* of the line. Unlike stack, customers are added at the rear end and deleted from the front end in a queue (FIFO).

An example of the queue in computer science is print jobs scheduled for printers. These jobs are maintained in a queue. The job fired for the printer first gets printed first. Same is the scenario for job scheduling in the CPU of computer.

Like a stack, a queue also (usually) holds data elements of the same type. We usually graphically display a queue horizontally. *Figure 5.1* depicts a queue of 5 characters.

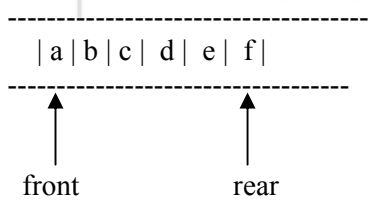



**Figure 5.1: A queue of characters**

The rule followed in a queue is that elements are added at the *rear* and come off of the *front* of the queue. After the addition of an element to the above queue, the position of rear pointer changes as shown below. Now the *rear* is pointing to the new element 'g' added at the rear of the queue(refer to *Figure 5.2*).

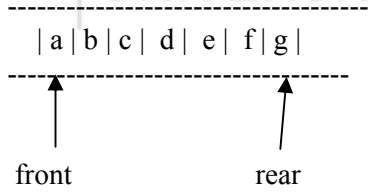



**Figure 5.2: Queue of figure 5.1 after addition of new element**

After the removal of element 'a' from the front, the queue changes to the following with the *front* pointer pointing to 'b' (refer to *Figure 5.3*).

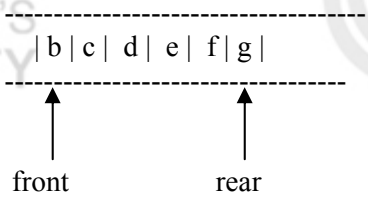



**Figure 5.3: Queue of figure 5.2 after deletion of an element**

***Algorithm for addition of an element to the queue***

Step 1: Create a new element to be added
Step 2: If the queue is empty, then go to step 3, else perform step 4
Step 3: Make the front and rear point this element
Step 4: Add the element at the end of the queue and shift the rear pointer to the newly added element.

***Algorithm for deletion of an element from the queue***

Step 1: Check for Queue empty condition. If empty, then go to step 2, else go to step 3
Step 2: Message "Queue Empty"
Step 3: Delete the element from the front of the queue. If it is the last element in the queue, then perform *step a* else *step b*
a) make front and rear point to null
b) shift the front pointer ahead to point to the next element in the queue

### 5.3 1 Array implementation of a queue

As the stack is a list of elements, the queue is also a list of elements. The stack and the queue differ only in the position where the elements can be added or deleted. Like other liner data structures, queues can also be implemented using arrays. Program 5.1 lists the implementation of a queue using arrays.

```
#include "stdio.h"
#define QUEUE_LENGTH 50
struct queue
{    int element[QUEUE_LENGTH];
     int front, rear, choice,x,y;
}

struct queue q;

main()
{
int choice,x;
printf ("enter 1 for add and 2 to remove element front the queue")
printf("Enter your choice")
scanf("%d",&choice);
switch (choice)

  {

case 1 :
 printf ("Enter element to be added :");
```

```
scanf("%d",&x);
add(&q,x);
break;

case 2 :
delete();
break;

  }

}
add(y)
{
++q.rear;
if (q.rear < QUEUE_LENGTH)
   q.element[q.rear] = y;
else
 printf("Queue overflow")
}

delete()
{

if q.front > q.rear printf("Queue empty");
else{
  x = q.element[q.front];
 q.front++;
 }
retrun x;
}
```

**Program 5.1: Array implementation of a Queue**

### 5.3.2 Linked List Implementation of a queue

The basic element of a linked list is a "record" structure of at least two fields. The object that holds the data and refers to the next element in the list is called a node (refer to *Figure 5.4*).

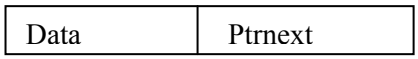



**Figure 5.4: Structure of a node**

The *data* component may contain data of any type. *Ptrnext* is a reference to the next element in the queue structure. *Figure 5.5* depicts the linked list representation of a queue.

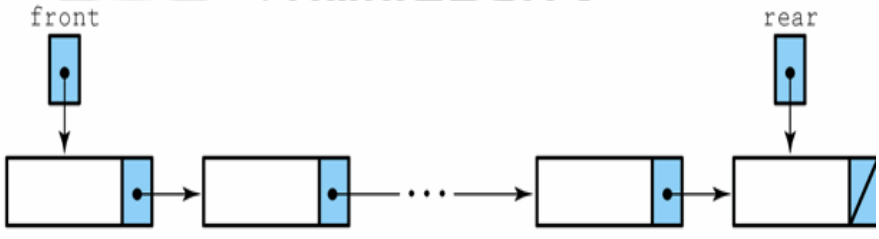



**Figure 5.5: A linked list representation of a Queue**

Program 5.2 gives the program segment for the addition of an element to the queue.

Program 5.3 gives the program segment for the deletion of an element from the queue.

```
add(int value)
{
struct queue *new;
new = (struct queue*)malloc(sizeof(queue));
new->value = value;
new->next = NULL;
if (front == NULL)
{
  queueptr = new;
  front = rear = queueptr
}
else
{
rear->next = new;
rear=new;
}
}
```

**Program 5.2: Program segment for addition of an element to the queue**

```
delete()
{
int delvalue = 0;
if (front == NULL) printf("Queue Empty");
{
  delvalue = front->value;
 if (front->next==NULL)
{
free(front);
queueptr=front=rear=NULL;
}
else
{
front=front->next;
free(queueptr);
queueptr=front;

}
}
}
```

**Program 5.3: Program segment for deletion of an element from the queue**

## ☞ Check Your Progress 1

1) The queue is a data structure where addition takes place at _____ and deletion takes place at _____.

2) The queue is also known as _____ list.

3) Compare the array and linked list representations of a queue. Explain your answer.

# 5.4 IMPLEMENTATION OF MULTIPLE QUEUES

So far, we have seen the representation of a single queue, but many practical applications in computer science require several queues. Multiqueue is a data structure where multiple queues are maintained. This type of data structures are used for process scheduling. We may use one dimensional array or multidimensional array to represent a multiple queue.

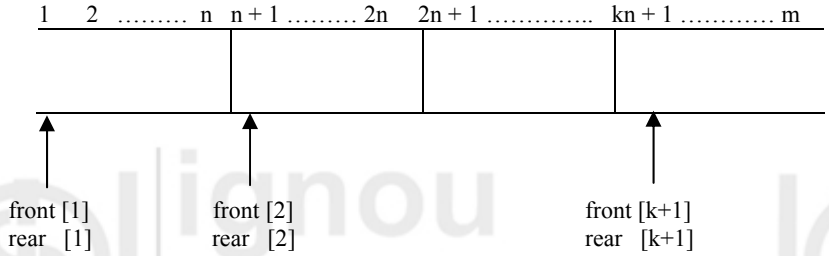



**Figure 5.6: Multiple queues in an array**

A multiqueue implementation using a single dimensional array with *m* elements is depicted in *Figure 5.6*. Each queue has *n* elements which are mapped to a liner array of *m* elements.

### Array Implementation of a multiqueue

Program 5.4 gives the program segment using arrays for the addition of an element to a queue in the multiqueue.

```
addmq(i,x)  /* Add x to queue i */
{
int i,x;
++rear[i];
if ( rear[i] == front[i+1])
 printf("Queue is full");
else
{
rear[i] = rear[i]+1;
mqueue[rear[i]] = x;
}
}
```

**Program 5.4: Program segment for the addition of an element to the queue**

Program 5.5 gives the program segment for the deletion of an element from the queue.

```
delmq(i)   /* Delete an element from queue i */
{
int i,x;
if ( front[i] == rear[i])
 printf("Queue is empty");
{
x = mqueue[front[i]];
front[i] = front[i]-1 ;
return x;
}
}
```

**Program 5.5: Program segment for the deletion of an element from the queue**

## 5.5 IMPLEMENTATION OF CIRCULAR QUEUES

One of the major problems with the linear queue is the lack of proper utilisation of space. Suppose that the queue can store 100 elements and the entire queue is full. So, it means that the queue is holding 100 elements. In case, some of the elements at the front are deleted, the element at the last position in the queue continues to be at the same position and there is no efficient way to find out that the queue is not full. In this way, space utilisation in the case of linear queues is not efficient. This problem is arising due to the representation of the queue.

The alternative representation is to depict the queue as circular. In case, we are representing the queue using arrays, then, a queue with *n* elements starts from index *0* and ends at *n-1*.So, clearly , the first element in the queue will be at index 0 and the last element will be at n-1 when all the positions between index 0 and n-1(both inclusive) are filled. Under such circumstances, front will point to 0 and rear will point to n-1. However, when a new element is to be added and if the rear is pointing to n-1, then, it needs to be checked if the position at index 0 is free. If yes, then the element can be added to that position and rear can be adjusted accordingly. In this way, the utilisation of space is increased in the case of a circular queue.

In a circular queue, front will point to one position less to the first element anti-clock wise. So, if the first element is at position 4 in the array, then the front will point to position 3. When the circular queue is created, then both front and rear point to index 1. Also, we can conclude that the circular queue is empty in case both front and rear point to the same index. *Figure 5.7* depicts a circular queue.

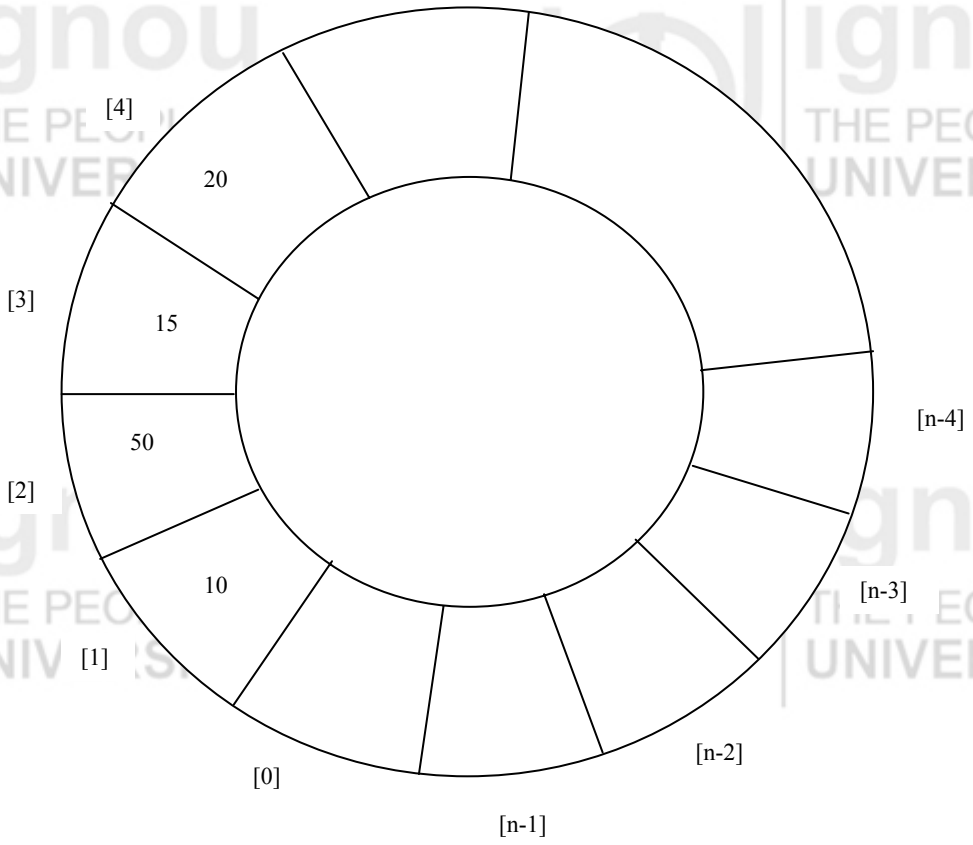



**Figure 5.7 : A circular queue (Front = 0, Rear = 4)**

**Algorithm for Addition of an element to the circular queue:**

**Step-1:** If "rear" of the queue is pointing to the last position then go to step-2 or else Step-3
**Step-2:** make the "rear" value as 0
**Step-3:** increment the "rear" value by one
**Step-4:** a. if the "front" points where "rear" is pointing and the queue holds a not NULL value for it, then its a "queue overflow" state, so quit; else go to step-b
b. add the new value for the queue position pointed by the "rear"

**Algorithm for deletion of an element from the circular queue:**

**Step-1:** If the queue is empty then say "queue is empty" and quit; else continue
**Step-2:** Delete the "front" element
**Step-3:** If the "front" is pointing to the last position of the queue then go to step-4 else go to step-5
**Step-4:** Make the "front" point to the first position in the queue and quit
**Step-5:** Increment the "front" position by one

### 5.5.1 Array implementation of a circular queue

A circular queue can be implemented using arrays or linked lists. Program 5.6 gives the array implementation of a circular queue.

```
#include "stdio.h"
void add(int);
void deleteelement(void);
int max=10;          /*the maximum limit for queue has been set*/
static int queue[10];
int front=0, rear=-1;  /*queue is initially empty*/
void main()
{
int choice,x;
printf ("enter 1 for addition and 2 to remove element front the queue and 3 for exit");
printf("Enter your choice");
scanf("%d",&choice);
switch (choice)
{
case 1 :
printf ("Enter the element to be added :");
scanf("%d",&x);
add(x);
break;
case 2 :
deleteelement();
break;
 }
}
void add(int y)
{
 if(rear == max-1)
   rear = 0;
   else
   rear = rear + 1;
   if( front == rear && queue[front] != NULL)
   printf("Queue Overflow");
   else
```

```
  queue[rear] = y;
}

void deleteelement()
{
int deleted_front = 0;
 if (front == NULL)
     printf("Error - Queue empty");
 else
    {
     deleted_front = queue[front];
     queue[front] = NULL;
      if (front == max-1)
      front = 0;
     else
     front = front + 1;
  }
}
```

**Program 5.6: Array implementation of a Circular queue**

### 5.5.2 Linked list implementation of a circular queue

Link list representation of a circular queue is more efficient as it uses space more efficiently, of course with the extra cost of storing the pointers. Program 5.7 gives the linked list representation of a circular queue.

```
#include “stdio.h”
 struct cq
{     int value;
      int *next;
};

typedef struct  cq *cqptr
cqptr p, *front, *rear;

main()
{
int choice,x;

/*  Initialise the circular queue */
cqptr =  front = rear =  NULL;

printf (“Enter 1 for addition and 2 to delete element from the queue”)
printf(“Enter your choice”)
scanf(“%d”,&choice);
switch (choice)
{

case 1 :
            printf (“Enter the element to be added :”);
            scanf(“%d”,&x);
            add(&q,x);
            break;

case 2 :
            delete();
```

```
            break;
    }
}

/************ Add element ****************/

add(int value)
{
   struct cq *new;
   new = (struct cq*)malloc(sizeof(queue));
   new->value = value
   new->next = NULL;

   if (front == NULL)
   {
    cqptr = new;
    front = rear = queueptr;
   }
  else
       {
     rear->next = new;
     rear=new;
    }
}

/* *************** delete element ***********/
delete()
{
int delvalue = 0;
if (front == NULL)
{ printf("Queue is empty");
  delvalue = front->value;
  if (front->next==NULL)
 {
   free(front);
   queueptr = front = rear = NULL;
}
else
{
front=front->next;
free(queueptr);
queueptr = front;

}
}
}
```

**Program 5.7 : Linked list implementation of a Circular queue**

## 5.6 IMPLEMENTATION OF DEQUEUE

Dequeue (a double ended queue) is an abstract data type similar to queue, where addition and deletion of elements are allowed at both the ends. Like a linear queue and a circular queue, a dequeue can also be implemented using arrays or linked lists.

### 5.6.1 Array implementation of a dequeue

If a Dequeue is implemented using arrays, then it will suffer with the same problems that a linear queue had suffered. Program 5.8 gives the array implementation of a Dequeue.

```
#include “stdio.h”
#define QUEUE_LENGTH 10;
int dq[QUEUE_LENGTH];
int front, rear, choice,x,y;
main()
{
  int choice,x;
  front = rear = -1; /* initialize the front and rear to null i.e empty queue */

 printf (“enter 1 for addition and  2 to remove element from the front of the queue");
 printf (“enter 3 for addition and  4 to remove element from the rear of the queue");
 printf(“Enter your choice”);
 scanf(“%d”,&choice);
 switch (choice)
 {
  case 1:
            printf (“Enter element to be added :”);
            scanf(“%d”,&x);
            add_front(x);
            break;
  case 2:
           delete_front();
           break;
  case 3:
           printf (“Enter the element to be added :”);
           scanf(“%d ”,&x);
           add_rear(x);
           break;
  case 4:
           delete_rear();
           break;
 }
}
/*************** Add at the front **************/
add_front(int y)
{
if (front == 0)
{
   printf(“Element can not be added at the front“);
   return;
else
    {
     front = front - 1;
     dq[front] = y;
      if (front == -1 ) front = 0;
    }
 }
/*************** Delete from the front **************/
delete_front()
{
 if front == -1
 printf(“Queue empty”);
```

```
 else
        return dq[front];
        if (front = = rear)
        front = rear = -1
        else
             front = front + 1;
}
/*************** Add at the rear **************/
add_rear(int y)
if (front == QUEUE_LENGTH -1 )
{
 printf("Element can not be added at the rear ")
 return;
else
{
 rear  = rear + 1;
 dq[rear] = y;
 if (rear = = -1 )
 rear = 0;
}
}

/*************** Delete at the rear **************/
delete_rear()
{
 if rear == -1
 printf("deletion is not possible from rear");
 else
 {
        if (front = = rear)
        front = rear = -1
        else
             { rear = rear – 1;
               return dq[rear];
             }
}
}
```

**Program 5.8: Array implementation of a Dequeue**

### 5.6.2 Linked list implementation of a dequeue

Double ended queues are implemented with doubly linked lists.

A doubly link list can traverse in both the directions as it has two pointers namely left and right. The right pointer points to the next node on the right where as the left pointer points to the previous node on the left. Program 5.9 gives the linked list implementation of a Dequeue.

```
# include "stdio.h"
#define NULL 0
struct dq {
              int info;
              int *left;
              int *right;
            };
typedef struct dq *dqptr;
dqptr p, tp;
```

```
dqptr head;
dqptr tail;
main()
{
        int choice, I, x;
        dqptr n;
       dqptr getnode();
        printf("\n Enter 1: Start  2 : Add at Front 3 : Add at Rear 4: Delete at Front 5:
Delete at Back");
while (1)
{
        printf("\n 1: Start  2 : Add at Front 3 : Add at Back 4: Delete at Front 5: Delete
at Back 6 : exit");
        scanf("%d", &choice);
        switch (choice)
        {
         case 1:
                  create_list();
                  break;
         case 2:
                   eq_front();
                   break;
         case 3:
                  eq_back();
                  break;
        case 4:
                 dq_front();
                 break;
        case 5:
                 dq_back();
                 break;
        case 6 :
                 exit(6);
  }
 }
}

create_list()
{
  int I, x;
  dqptr t;
  p = getnode();
  tp = p;
  p->left = getnode();
  p->info = 10;
  p_right = getnode();
  return;
}

dqptr getnode()
{
   p = (dqptr) malloc(sizeof(struct dq));
   return p;
}

dq_empty(dq q)
{
  return q->head = = NULL;
```

```
}
eq_front(dq q, void *info)
{
 if (dq_empty(q))
    q->head = q->tail = dcons(info, NULL, NULL);
else
{
   q-> head -> left =dcons(info, NULL, NULL);
   q->head -> left ->right = q->head;
   q ->head  = q->head ->left;
}
}

eq_back(dq q, void *info)
{
 if (dq_empty(q))
    q->head = q->tail = dcons(info, NULL, NULL)
 else
{
 q-> tail -> right =dcons(info, NULL, NULL);
 q->tail -> right -> left = q->tail;
 q ->tail  = q->tail ->right;
}
}
 dq_front(dq q)
{
  if dq is not empty
{
dq tp = q-> head;
void *info = tp -> info;
q ->head = q->head-> right;
free(tp);
if (q->head = = NULL)
  q -> tail = NULL;
else
 q -> head -> left = NULL;
return info;
}
}

dq_back(dq q)
{
  if (q!=NULL)
  {
  dq tp = q-> tail;
  *info = tp -> info;
  q ->tail = q->tail-> left;
  free(tp);
  if (q->tail = = NULL)
               q -> head = NULL;
 else
              q -> tail -> right = NULL;
  return info;
  }
}
```

**Program 5.9 : Linked list implementation of a Dequeue**

☞ **Check Your Progress 2**

1) _____ allows elements to be added and deleted at the front as well as at the rear.
2) It is not possible to implement multiple queues in an Array. (True/False)
3) The index of a circular queue starts at _____.

## 5.7 SUMMARY

In this unit, we discussed the data structure *Queue***.** It had two ends. One is front from where the elements can be deleted and the other if rear to where the elements can be added. A queue can be implemented using Arrays or Linked lists. Each representation is having it's own advantages and disadvantages. The problems with arrays are that they are limited in space. Hence, the queue is having a limited capacity. If queues are implemented using linked lists, then this problem is solved. Now, there is no limit on the capacity of the queue. The only overhead is the memory occupied by the pointers.

There are a number of variants of the queues. Normally, queues mean circular queues. Apart from linear queues, we also discussed circular queues in this unit. A special type of queue called Dequeue was also discussed in this unit. Dequeues permit elements to be added or deleted at either of the rear or front. We also discussed the array and linked list implementations of Dequeue.

## 5.8 SOLUTIONS/ANSWERS

**Check Your Progress 1**

1. rear , front
2. First in First out (FIFO) list

**Check Your Progress 2**

1. Dequeue
2. False
3. 0

## 5.9 FURTHER READINGS

**Reference Books**

1. *Data Structures using C* by Aaron M.Tanenbaum, Yedidyah Langsam, Moshe J.Augenstein , PHI publications
2. *Algorithms+Data Structures = Programs* by Niklaus Wirth, PHI publications

**Reference Websites**

http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/queues.html
http://www.cs.toronto.edu/~wayne/libwayne/libwayne.html

# UNIT 6 TREES

**Structure** **Page Nos.**



## 6.0 INTRODUCTION

Have you ever thought how does the operating system manage our files? Why do we have a hierarchical file system? How do files get saved and deleted under hierarchical directories? Well, we have answers to all these questions in this section through a hierarchical data structure called Trees! Although most general form of a tree can be defined as an **acyclic graph**, we will consider in this section only rooted tree as general tree does not have a parent-child relationship.

Tree is a data structure which allows you to associate a parent-child relationship between various pieces of data and thus allows us to arrange our records, data and files in a hierarchical fashion. Consider a Tree representing your family structure. Let us say that we start with your grand parent; then come to your parent and finally, you and your brothers and sisters. In this unit, we will go through the basic tree structures first (general trees), and then go into the specific and more popular tree called binary-trees.

## 6.1 OBJECTIVES

After going through this unit, you should be able

- to define a tree as abstract data type (ADT);
- learn the different properties of a Tree and a Binary tree;
- to implement the Tree and Binary tree, and
- give some applications of Tree.

## 6.2 ABSTRACT DATA TYPE-TREE

**Definition:** A set of data values and associated operations that are precisely specified independent of any particular implementation.

Since the data values and operations are defined with mathematical precision, rather than as an implementation in a computer language, we may reason about effects of the

operations, relationship to other abstract data types, whether a programming language implements the particular data type, etc.

Consider the following abstract data type:

**Structure** Tree

type Tree = nil | fork (Element , Tree , Tree)

**Operations:**

null : Tree -> Boolean
leaf : Tree -> Boolean
fork : (Element , Tree , Tree) -> Tree
left : Tree -> Tree // It depicts the properties of tree that left of a tree is also a tree.
right: Tree -> Tree
contents: Tree -> Element
height (nil) = 0 |
height (fork(e,T,T')) = 1+max(height(T), height(T'))
weight (nil) = 0 |
weight (fork(e,T,T')) = 1+weight(T)+weight(T')

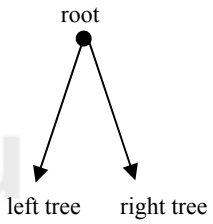



**Figure 6.1: A binary tree**

**Rules:**

null(nil) = true // nil is an empty tree

null(fork(e, T, T'))= false // e : element , T and T are two sub tree

leaf(fork(e, nil, nil)) = true
leaf(fork(e, T, T')) = false if not null(T) or not null(T')
leaf(nil) = error

left(fork(e, T, T')) = T
left(nil) = error

right(fork(e, T, T')) = T'
right(nil) = error

contents(fork(e, T, T')) = e
contents(nil) = error

Look at the definition of Tree (ADT). A way to think of a *binary tree* is that it is either empty (nil) or contains an element and two sub trees which are themselves binary trees (Refer to *Figure 6.1*). Fork operation joins two sub tree with a parent node and

produces another Binary tree. It may be noted that a tree consisting of a single leaf is defined to be of height 1.

**Definition :** A tree is a connected, acyclic graph (Refer to *Figure 6.2*).

It is so connected that any node in the graph can be reached from any other node by exactly one path.

It does not contain any cycles (circuits, or closed paths), which would imply the existence of more than one path between two nodes. This is the most general kind of tree, and may be converted into the more familiar form by designating a node as the root. We can represent a tree as a construction consisting of nodes, and edges which represent a relationship between two nodes. In *Figure 6.3*, we will consider most common tree called **rooted tree**. A rooted tress has a single root node which has no parents.

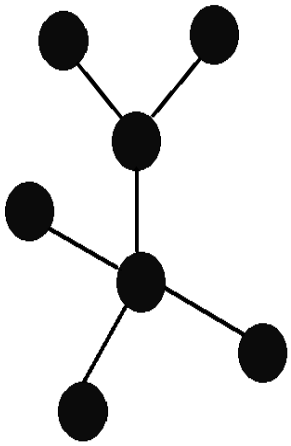


**Figure 6.2 : Tree as a connected acyclic graph**

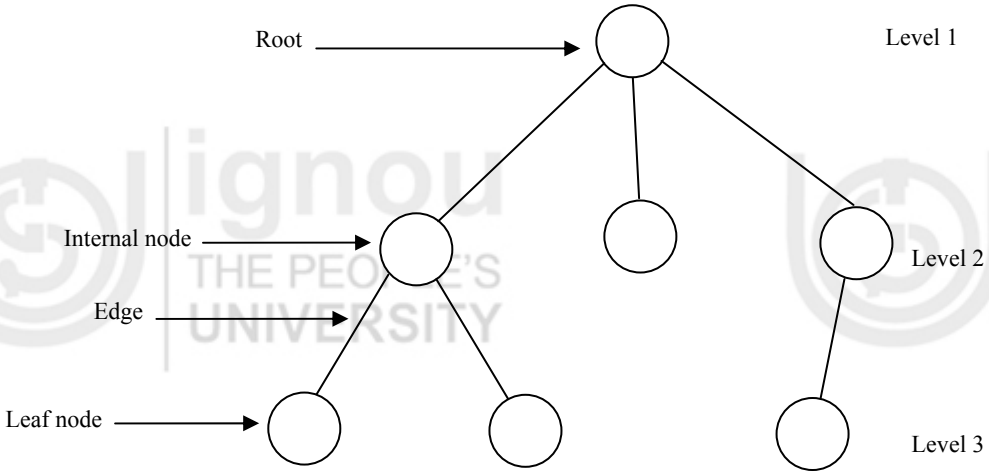



**Figure 6.3 : A rooted tree**

In a more formal way, we can define a tree T as a finite set of one or more nodes such that there is one designated node r called the root of T, and the remaining nodes in (T – { r } ) are partitioned into n > 0 disjoint subsets $T_1$, $T_2$, ..., $T_k$ each of which is a tree, and whose roots r1 , r2 , ..., rk , respectively, are children of r. The general tree is a generic tree that has one root node, and every node in the tree can have an unlimited number of child nodes. One popular use of this kind of tree is a Family Tree.
A tree is an instance of a more general category called graph.

- A tree consists of nodes connected by edges.
- A root is a node without parent.
- Leaves are nodes with no children.
- The root is at level 1. The child nodes of root are at level 2. The child nodes of nodes at level 2 are at level 3 and so on.
- The depth (height) of a Binary tree is equal to the number of levels in it.
- Branching factor defines the maximum number of children to any node. So, a branching factor of 2 means a binary tree.

- Breadth defines the number of nodes at a level.
- The depth of a node M in a tree is the length of the path from the root of the tree to M.
- A node in a Binary tree has at most 2 children.

The following are the properties of a Tree.

*Full Tree :* A tree with all the leaves at the same level, and all the non-leaves having the same degree

- Level h of a full tree has $d^{h-1}$ nodes.
- The first h levels of a full tree have $1 + d + d^2 + d^3 + d^4 + \ldots\ldots + d^{h-1} = (d^h - 1)/(d - 1)$ nodes where d is the degree of nodes.
- The number of edges = the number of nodes – 1 (Why? Because, an edge represents the relationship between a child and a parent, and every node has a parent except the root.
- A tree of height *h* and degree *d* has at most $d^h - 1$ elements.

*Complete Trees*

*A complete tree is a k-ary position tree in which all levels are filled from left to right. There are a number of specialized trees.*

They are binary trees, binary search trees, AVL-trees, red-black trees, 2-3 trees.

*Data structure- Tree*

Tree is a dynamic data structures. Trees can expand and contract as the program executes and are implemented through pointers. A tree deallocates memory when an element is deleted.

Non-linear data structures: Linear data structures have properties of ordering relationship (can the elements/nodes of tree be sorted?). There is no first node or last node. There is no ordering relationship among elements of tree.

Items of a tree can be partially ordered into a hierarchy via parent-child relationship. Root node is at the top of the hierarchy and leafs are at the bottom layer of the hierarchy. Hence, trees can be termed as hierarchical data structures.

## 6.3 IMPLEMENTATION OF TREE

The most common way to add nodes to a general tree is to first find the desired parent of the node you want to insert, then add the node to the parent's child list. The most common implementations insert the nodes one at a time, but since each node can be considered a tree on its own, other implementations build up an entire sub-tree before adding it to a larger tree**.** As the nodes are added and deleted dynamically from a tree, tree are often implemented by link lists. However, it is simpler to write algorithms for a data representation where the numbers of nodes are fixed. *Figure* 6.4 depicts the structure of the node of a general k-ary tree.

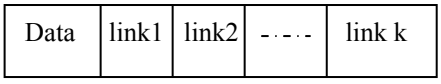



**Figure 6.4 : Node structure of a general k-ary tree**

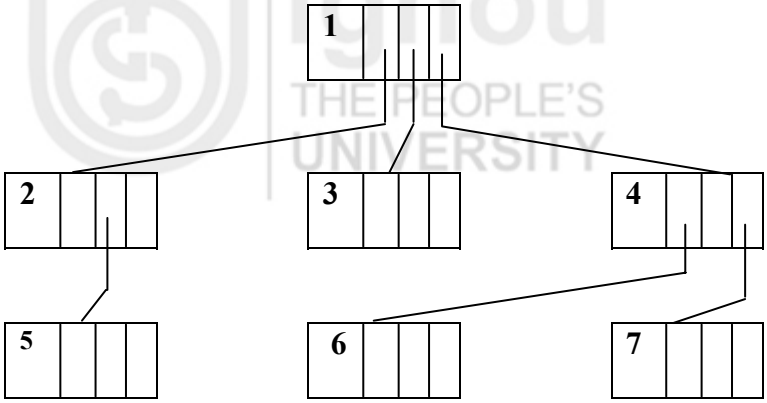



**Figure 6.5: A linked list representation of tree (3-ary tree)**

*Figure 6.5* depicts a tree with one data element and three pointers. The number of pointers required to implement a general tree depend of the maximum degree of nodes in the tree.

## 6.4 TREE TRAVERSALS

There are three types of tree traversals, namely, Preorder, Postorder and Inorder.

*Preorder traversal*: Each node is visited before its children are visited; the root is visited first.

**Algorithm for pre order traversal:**

1. visit root node
2. traverse left sub-tree in preorder
3. traverse right sub-tree in preorder

*Example of pre order traversal:* Reading of a book, as we do not read next chapter unless we complete all sections of previous chapter and all it's sections (refer to *Figure 6.6*).

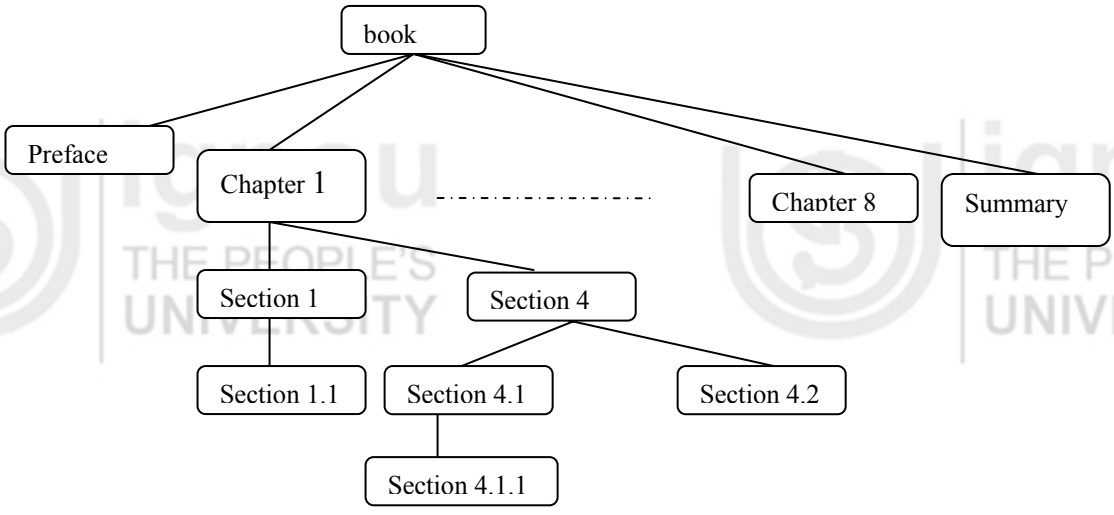



**Figure 6.6 : Reading a book : A preorder tree traversal**

As each node is traversed only once, the time complexity of preorder traversal is T(n) = O(n), where n is number of nodes in the tree.

*Postorder traversal:* The children of a node are visited before the node itself; the root is visited last. Every node is visited after its descendents are visited.

**Algorithm for postorder traversal:**

1. traverse left sub-tree in post order
2. traverse right sub-tree in post order
3. visit root node.

Finding the space occupied by files and directories in a file system requires a postorder traversal as the space occupied by directory requires calculation of space required by all files in the directory (children in tree structure) (refer to *Figure 6.7*)

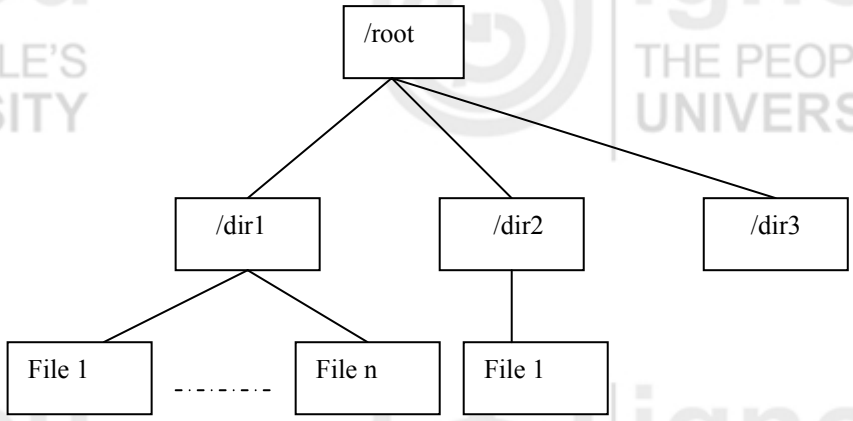



**Figure 6.7 : Calculation of space occupied by a file system : A post order traversal**

As each node is traversed only once, the time complexity of post order traversal is T(n) = O(n), where n is number of nodes in the tree.

Inorder traversal: The left sub tree is visited, then the node and then right sub-tree.

**Algorithm for inorder traversal:**

1. traverse left sub-tree
2. visit node
3. traverse right sub-tree

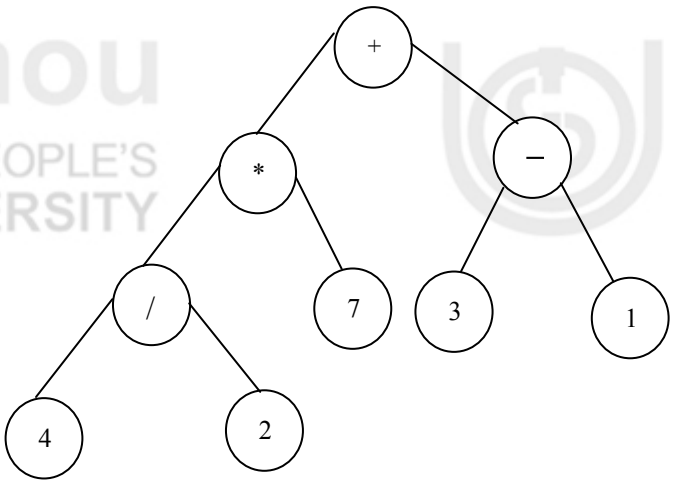



**Figure 6.8 : An expression tree : An inorder traversal**

Inorder traversal can be best described by an expression tree, where the operators are at parent node and operands are at leaf nodes.

Let us consider the above expression tree (refer to *Figure 6.8*). The preorder, postorder and inorder traversal are given below:

preorder Traversal : + * / 4 2 7 - 3 1
postorder traversal : 4 2 / 7 * 3 1 - +
inorder traversal : - ((((4 / 2) * 7) + (3 - 1))

There is another tree traversal (of course, not very common) is called level order, where all the nodes of the same level are travelled first starting from the root (refer to *Figure 6.9*).

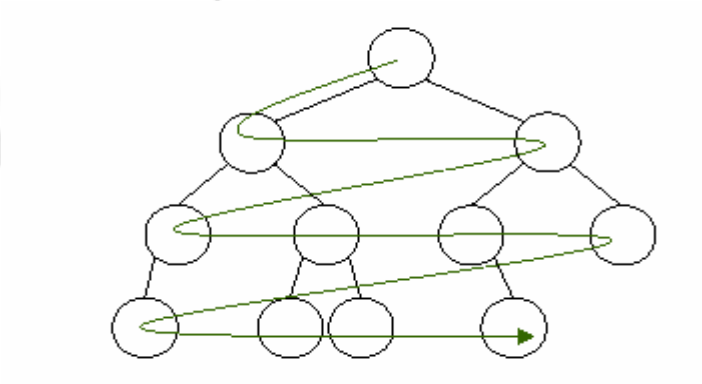


**Figure 6.9: Tree Traversal: Level Order**

### ☞ Check Your Progress 1

1) If a tree has 45 edges, how many vertices does it have?
2) Suppose a full 4-ary tree has 100 leaves. How many internal vertices does it have?
3) Suppose a full 3-ary tree has 100 internal vertices. How many leaves does it have?
4) Prove that if T is a full m-ary tree with v vertices, then T has ((m-1)v+1)/m leaves.

## 6.5 BINARY TREES

A binary tree is a special tree where each non-leaf node can have atmost two child nodes. Most important types of trees which are used to model yes/no, on/off, higher/lower, i.e., binary decisions are binary trees.

Recursive Definition: A binary tree is either empty or a node that has left and right sub-trees that are binary trees. Empty trees are represented as boxes (but we will almost always omit the boxes).

In a formal way, we can define a binary tree as a finite set of nodes which is either empty or partitioned in to sets of $T_0$, $T_l$, $T_r$, where $T_0$ is the root and $T_l$ and Tr are left and right binary trees, respectively.

*Properties of a binary tree*

- If a binary tree contains ***n*** nodes, then it contains exactly $n - 1$ edges;
- A Binary tree of height $h$ has $2^h - 1$nodes or less.
- If we have a binary tree containing $n$ nodes, then the height of the tree is at most $n$ and at least ceiling $\log_2(n + 1)$.
- If a binary tree has n nodes at a level l then, it has at most 2n nodes at a level l+1
- The total number of nodes in a binary tree with depth d (root has depth zero) is

$$N = 2^0 + 2^1 + 2^2 + \ldots\ldots + 2^d = 2^{d+1} - 1$$

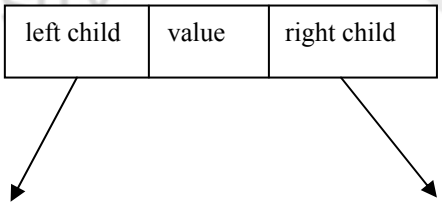
*Full Binary Trees:* A binary tree of height h which had $2^h - 1$ elements is called a Full Binary Tree.

*Complete Binary Trees:* A binary tree whereby if the height is d, and all levels, except possibly level d, are completely full. If the bottom level is incomplete, then it has all nodes to the left side. That is the tree has been filled in the level order from left to right.

## 6.6 IMPLEMENTATION OF A BINARY TREE

Like general tree, binary trees are implemented through linked lists. A typical node in a Binary tree has a structure as follows (refer to *Figure 6.10*):

```
struct NODE
 {
 struct NODE *leftchild;
 int nodevalue;              /* this can be of any data type */
 struct NODE *rightchild;
};
```




The 'left child'and 'right child' are pointers to another tree-node. The "leaf node" (not shown) here will have NULL values for these pointers.

**Figure 6.10 : Node structure of a binary tree**

The binary tree creation follows a very simple principle. For the new element to be added, compare it with the current element in the tree. If its value is less than the current element in the tree, then move towards the left side of that element or else to its right. If there is no sub tree on the left, then make your new element as the left child of that current element or else compare it with the existing left child and follow the same rule. Exactly, the same has to done for the case when your new element is greater than the current element in the tree but this time with the right child. Though this logic is followed for the creation of a Binary tree, this logic is often suitable to search for a key value in the binary tree.

**Algorithm for the implementation of a Binary tree:**

Step-1: If value of new element < current element, then go to step-2 or else step -3
Step-2: If the current element does not have a left sub-tree, then make your new

element the left child of the current element; else make the existing left child as your current element and go to step-1

Step-3: If the current element does not have a right sub-tree, then make your new element the right child of the current element; else make the existing right child as your current element and go to step-1

Program 6.1 depicts the segment of code for the creation of a binary tree.

```
struct NODE
{
 struct NODE *left;
 int value;
 struct NODE *right;
};

create_tree( struct NODE  *curr, struct NODE  *new )
{
  if(new->value <= curr->value)
     {
       if(curr->left != NULL)
       create_tree(curr->left, new);
       else
       curr->left = new;
     }
 else
  {
      if(curr->right != NULL)
     create_tree(curr->right, new);
     else
     curr->right = new;
  }
}
```

**Program 6.1 : Binary tree creation**

***Array-based representation of a Binary Tree***

Consider a complete binary tree T having n nodes where each node contains an item (value). Label the nodes of the complete binary tree T from top to bottom and from left to right 0, 1, ..., n-1. Associate with T the array A where the $i^{th}$ entry of A is the item in the node labelled i of T, i = 0, 1, ..., n-1. Figure 6.11 depicts the array representation of a Binary tree of Figure 6.16.

Given the index **i** of a node, we can easily and efficiently compute the index of its parent and left and right children:

Index of Parent: (i – 1)/2, Index of Left Child: 2i + 1,  Index of Right Child: 2i + 2.

| Node # | Item | Left child | Right child |
|---|---|---|---|
| 0 | A | 1 | 2 |
| 1 | B | 3 | 4 |
| 2 | C | -1 | -1 |
| 3 | D | 5 | 6 |
| 4 | E | 7 | 8 |
| 5 | G | -1 | -1 |
| 6 | H | -1 | -1 |
| 7 | I | -1 | -1 |
| 8 | J | -1 | -1 |
| 9 | ? | ? | ? |

**Figure 6.11 : Array Representation of  a Binary Tree**

First column represents index of node, second column consist of the item stored in the node and third and fourth columns indicate the positions of left and right children (–1 indicates that there is no child to that particular node.)

## 6.7 BINARY TREE TRAVERSALS

We have already discussed about three tree traversal methods in the previous section on general tree. The same three different ways to do the traversal – preorder, inorder and postorder are applicable to binary tree also.

Let us discuss the inorder binary tree traversal for following binary tree (refer to *Figure 6.12*):

We start from the root i.e. * We are supposed to visit its left sub-tree then visit the node itself and its right sub-tree. Here, root has a left sub-tree rooted at +. So, we move to + and check for its left sub-tree (we are suppose repaeat this for every node). Again, + has a left sub-tree rooted at 4. So, we have to check for 4's left sub-tree now, but 4 doesn't have any left sub-tree and thus we will visit node 4 first (print in our case) and check for its right sub-tree. As 4 doesn't have any right sub-tree, we'll go back and visit node +; and check for the right sub-tree of +. It has a right sub-tree rooted at 5 and so we move to 5. Well, 5 doesn't have any left or right sub-tree. So, we just visit 5 (print 5)and track back to +. As we have already visited + so we track back to * . As we are yet to visit the node itself and so we visit * before checking for the right sub-tree of *, which is 3. As 3 does not have any left or right sub-trees, we visit 3.
So, the inorder traversal results in 4 + 5 * 3

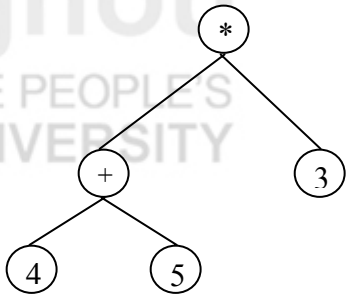



**Figure 6.12 : A binary tree**

**Algorithm: Inorder**
Step-1: For the current node, check whether it has a left child. If it has, then go to step-2 or else go to step-3
Step-2: Repeat step-1 for this left child
Step-3: Visit (i.e. printing the node in our case) the current node
Step-4: For the current node check whether it has a right child. If it has, then go to step-5
Step-5: Repeat step-1 for this right child

The preoreder and postorder traversals are similar to that of a general binary tree. The general thing we have seen in all these tree traversals is that the traversal mechanism is inherently recursive in nature.

### 6.7.1 Recursive Implementation of Binary Tree Traversals

There are three classic ways of recursively traversing a binary tree. In each of these, the left and right sub-trees are visited recursively and the distinguishing feature is **when the element in the root is visited or processed**.

Program 6.2, Program 6.3 and Program 6.4 depict the inorder, preorder and postorder traversals of a Binary tree.

```
struct NODE
{
```

```
 struct NODE *left;
 int value;      /* can be of any type */
 struct NODE *right;
};

inorder(struct NODE *curr)
{
 if(curr->left != NULL) inorder(curr->left);
 printf("%d", curr->value);

 if(curr->right != NULL) inorder(curr->right);

}
```

**Program 6.2 : Inorder traversal of a binary tree**

```
struct NODE
{
 struct NODE *left;
 int value;      /* can be of any type */
 struct NODE *right;
};

preorder(struct NODE *curr)
{
 printf("%d", curr->value);
 if(curr->left != NULL) preorder(curr->left);
 if(curr->right != NULL) preorder(curr->right);

}
```

**Program 6.3 : Preorder traversal of a binary tree**

```
struct NODE
{
 struct NODE *left;
 int value;      /* can be of any type */
 struct NODE *right;
};

postorder(struct NODE *curr)
{
  if(curr->left != NULL) postorder(curr->left);
 if(curr->right != NULL) postorder(curr->right);

printf("%d", curr->value);

}
```

**Program 6.4 : Postorder traversal of a binary tree**

In a preorder traversal, the root is visited first (pre) and then the left and right sub-trees are traversed. In a postorder traversal, the left sub-tree is visited first, followed by right sub-tree which is then followed by root. In an inorder traversal, the left sub-tree is visited first, followed by root, followed by right sub-tree.

### 6.7.2 Non-recursive implementation of binary tree traversals

As we have seen, as the traversal mechanisms were inherently recursive, the implementation was also simple through a recursive procedure. However, in the case of a non-recursive method for traversal, it has to be an iterative procedure; meaning, all the steps for the traversal of a node have to be under a loop so that the same can be applied to all the nodes in the tree.

**Algorithm : Non-recursive preorder binary tree traversal**

```
Stack S
push root onto S
repeat until S is empty
{
  v = pop S
  if v is not NULL
  visit v
  push v's right child onto S
  push v's left child onto S
}
```

Program 6.5 depicts the program segment for the implementation of non-recursive preorder traversal.

```
/* preorder traversal of a binary tree, implemented using a stack */
void preorder(binary_tree_type *tree)
{
 stack_type *stack;
 stack = create_stack();
 push(tree, stack);          /* push the first element of the tree to the stack */
 while (!empty(stack))
  {
  tree = pop(stack);
  visit(tree);
  push(tree->right, stack);  /* push right child to the stack */
  push(tree->left, stack);    /* push left child to the stack */
  }
 }
```

**Program 6.5: Non-recursive implementation of preorder traversal**

In the worst case, for preorder traversal, the stack will grow to size n/2, where n is number of nodes in the tree. Another method of traversing binary tree non-recursively which does not use stack requires pointers to the parent node (called threaded binary tree).

A threaded binary tree is a binary tree in which every node that does not have a right child has a THREAD (a third link) to its INORDER successor. By doing this threading we avoid the recursive method of traversing a tree and use of stack, which makes use of a lot of memory and time.

A node structure of threaded binary is :

The node structure for a threaded binary tree varies a bit and its like this –

```
struct NODE
{
```

```
 struct NODE *leftchild;
 int node_value;
 struct NODE *rightchild;
 struct NODE *thread;   /* third pointer to it's inorder successor */
}
```

## 6.8 APPLICATIONS

Trees are used enormously in computer programming. These can be used for improving database search times (binary search trees, 2-3 trees, AVL trees, red-black trees), Game programming (minimax trees, decision trees, pathfinding trees), 3D graphics programming (quadtrees, octrees), Arithmetic Scripting languages (arithmetic precedence trees), Data compression (Huffman trees), and file systems (B-trees, sparse indexed trees, tries ). Figure 6.13 depicts a tic-tac-toe game tree showing various stages of game.

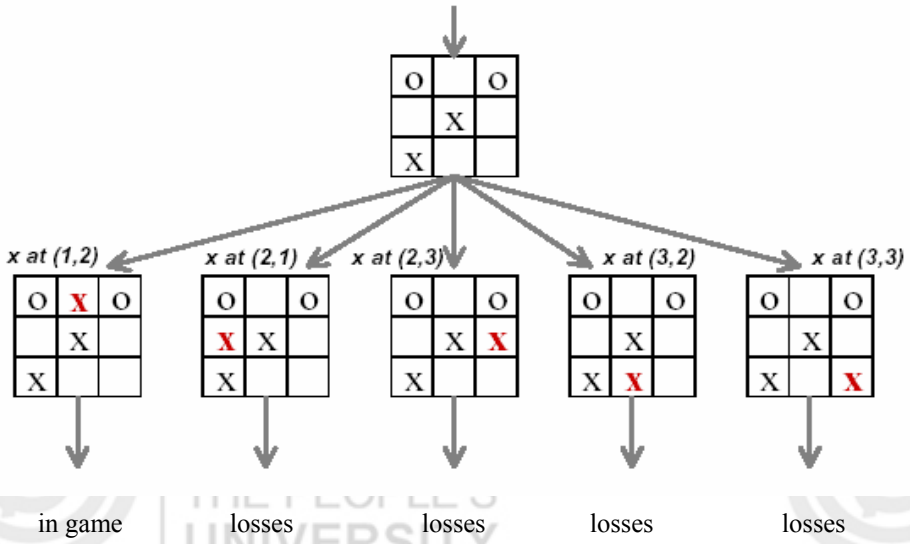



**Figure 6.13 : A tic-tac-toe game tree showing various stages of game**

In all of the above scenario except the first one, the player (playing with X) ultimately looses in subsequent moves.

The General tree (also known as Linked Trees) is a generic tree that has one root node, and every node in the tree can have an unlimited number of child nodes. One popular use of this kind of tree is in Family Tree programs. In game programming, many games use these types of trees for decision-making processes as shown above for tic-tac-toe. A computer program might need to make a decision based on an event that happened.

But this is just a simple tree for demonstration. A more complex AI decision tree would definitely have a lot more options. The interesting thing about using a tree for decision-making is that the options are cut down for every level of the tree as we go down, greatly simplifying the subsequent moves and improving the speed at which the AI program makes a decision.

The big problem with tree based level progressions, however, is that sometimes the tree can get too large and complex as the number of moves (level in a tree) increases. Imagine a game offering just two choices for every move to the next level at the end of each level in a ten level game. This would require a tree of 1023 nodes to be created.

Binary trees are used for searching keys. Such trees are called Binary Search trees(refer to *Figure 6.14*).

A Binary Search Tree (BST) is a binary tree with the following properties:

1. The key of a node is always greater than the keys of the nodes in its left sub-tree
2. The key of a node is always smaller than the keys of the nodes in its right sub-tree

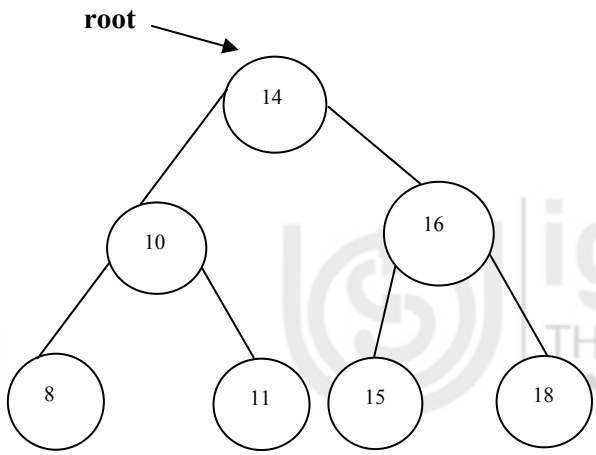



**Figure 6.14 : A binary search tree (BST)**

It may be seen that when nodes of a BST are traversed by inorder traversal, the keys appear in sorted order:

```
inorder(root)
{
inorder(root.left)
print(root.key)
inorder(root.right)
}
```

Binary Trees are also used for evaluating expressions.
A binary tree can be used to represent and evaluate arithmetic expressions.

1. If a node is a leaf, then the element in it specifies the value.
2. If it is not a leaf, then evaluate the children and combine them according to the operation specified by the element.

*Figure 6.15* depicts a tree which is used to evaluate expressions.

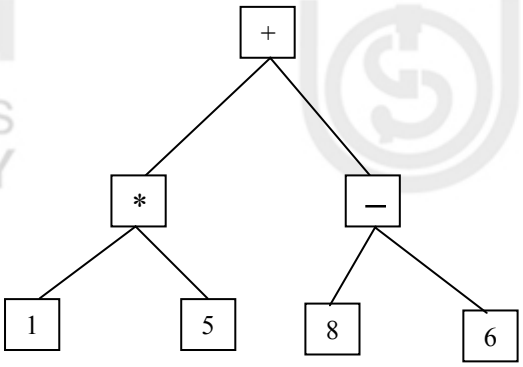



**Figure 6.15 : Expression tree for 1 * 5 + 8 - 6**

☞ **Check Your Progress 2**

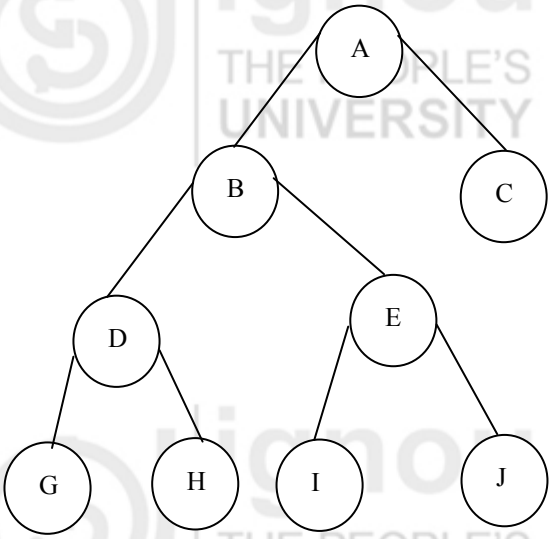



**Figure 6.16 : A binary tree**

1) With reference to *Figure 6.16*, find
   a) the leave nodes in the binary tree
   b) sibling of J
   c) Parent node of G
   d) depth of the binary tree
   e) level of node J
2) Give preorder, post order, inorder and level order traversal of above binary tree
3) Give array representation of the binary tree of *Figure 6.12*
4) Show that in a binary tree of N nodes, there are N+1 children with both the links as null (leaf node).

## 6.9 SUMMARY

Tree is one of the most widely used data structure employed for representing various problems. We studied tree as a special case of an acyclic graph. However, rooted trees are most prominent of all trees. We discussed definition and properties of general trees with their applications. Various tree traversal methods are also discussed.

Binary tree are the special case of trees which have at most two children. Binary trees are mostly implemented using link lists. Various tree traversal mechanisms include inorder, preorder and post order. These tree traversals can be implemented using recursive procedures and non-recursive procedures. Binary trees have wider applications in two way decision making problems which use yes/no, true/false etc.

## 6.10 SOLUTIONS / ANSWERS

**Check Your Progress 1**

1) If a tree has e edges and n vertices, then e=n – 1. Hence, if a tree has 45 edges, then it has 46 vertices.

2) A full 4-ary tree with 100 leaves has i=(100 – 1)/(4 – 1)=33 internal vertices.

3) A full 3-ary tree with 100 internal vertices has l = (3 – 1)*100+ 1=201 leaves

### Check Your Progress 2

1) Answers
   a. G,H,I and J
   b. I
   c. D
   d. 4
   e. 4
2) Preorder : ABDCHEIJC Postorder : GHDIJEBCA Inorder : GDHBIEFAC level-order: ABCDEGHIJ
3) Array representation of the tree in *Figure 6.12*

| Index of Node | Item | Left child | Right child |
|---|---|---|---|
| 0 | * | 1 | 2 |
| 1 | + | 3 | 4 |
| 2 | 3 | -1 | -1 |
| 3 | 4 | -1 | -1 |
| 4 | 5 | -1 | -1 |
| 5 | ? | ? | ? |

## 6.11 FURTHER READINGS

1. *Fundamentals of Data Structures in C++* by E.Horowitz, Sahni and D.Mehta; Galgotia Publications.
2. *Data Structures and Program Design in C* by Kruse, C.L.Tonodo and B.Leung; Pearson Education.

**Reference websites**

**http://www.csee.umbc.edu**
**http://www.cse.ucsc.edu**
**http://www.webopedia.com**