Vision of Institute:

"Empowerment through Knowledge".

Mission of Institute:

To educate the students to transform them as professionally competent and quality conscious engineers by providing conducive environment for teaching, learning, and overall personality development, culminating the institute into an international seat of excellence.

Vision of Department

Developing globally competent IT professional for sustainable growth of humanity.

Mission of Department

1. To empower students for developing systems and innovative products for solving real life problems.
2. To build strong foundation in computing knowledge that enables self-development entrepreneurship and Intellectual property.
3. To develop passionate IT professional by imparting global competencies and skills for serving society.

Program Specific Outcomes:

1. ***Placement*:** Enhance employability of students through design methodologies, application development tools and building knowledge base in Networking, Database, Web applications, Security etc.

2. ***Higher Studies:*** To prepare students to excel in postgraduate programmers or to succeed in industry by using knowledge of basic programming languages: syntax and semantics, problem analysis etc.

3. ***Project Development*:** Acquire ability to identify and solve IT problems by undertaking a team project and to improve communication skill and ability to work in team. Develop confidence for establishing their own IT companies in future.

4. ***Life-long Learning*:** To groom the students to Design, implement, and assess an IT based or computer based system to meet desired needs of society.

Program Educational Objectives:

1) ***Core Competence****:* To provide students with a solid foundation in mathematical, scientific and engineering fundamentals required to solve engineering problems and also to pursue higher studies.

2) ***Breadth****:* To train students with good scientific and engineering breadth so as to comprehend, analyse, design, and create novel products and solutions for the real life problems.

3) ***Professionalism****:* To inculcate in students professional and ethical attitude, effective communication skills, teamwork skills, multidisciplinary approach, and an ability to relate engineering issues to broader social context.

4) ***Learning Environment****:* To provide student with an academic environment aware of excellence, leadership, written ethical codes and guidelines, and the life-long learning needed for a successful professional career.

5) ***Attainment***: To prepare students to excel in research or to succeed in industry/technical profession through global, rigorous education and research and to become future entrepreneurs.

1. Searching and Sorting

AIM:

Consider a student database of SEIT class (at least 15 records). Database contains different fields of every student like Roll No, Name and SGPA.(array of structure)

a) **Design a roll call list, arrange list of students according to roll numbers in ascending order (Use Bubble Sort)**

b) **Arrange list of students alphabetically. (Use Insertion sort)**

c) **Arrange list of students to find out first ten toppers from a class. (Use Quick sort)**

d) **Search students according to SGPA. If more than one student having same SGPA, then print list of all students having same SGPA.**

e) **Search a particular student according to name using binary search without recursion. (all the student records having the presence of search key should be displayed)**

THEORY:

Introduction Sorting:

Sorting means bringing some orderliness in the data, which are otherwise not ordered. For example, to print the list of students in ascending order of their birth dates, we need to sort the student information in ascending order of date of birth (student information stored in a file may not be in this order) before generating the print report. In most of the computer applications we need to sort the data either in ascending or descending order as per the requirements of the application. Sorting techniques are broadly classified as internal sort techniques and external sort techniques.

Internal sort: Internal sort is also known as Memory Sort, and it is used to sort the small volume of data in computer memory. Following are the some of internal or memory sort techniques.

1. Bubble sort
2. Selection sort
3. Insertion sort
4. Bucket sort
5. Radix sort
6. Quick sort
7. Merge sort

External sort:

The term External sort is referred to sorting of voluminous data. For example, to create a temporary file sorted on some key from the input file containing thousands of records, we need to divide the input file in small partitions, which can be sorted on the key by using any one of the known internal sort techniques and then merging those partitions on the sorted key to generate new partition of bigger size and this process is repeated until we get only one large sorted partition of size equal to the number of records in input file. In

practice most of the times it is not possible to bring the entire data from input file to the memory of computer for sorting and we have to use some secondary storage to store the intermediate results (partitions). Thus in external sort input and output sorted files are stored on secondary storage i.e. disk.

Bubble sort:

Basic step:

From the remaining unsorted list, compare the first element with other elements and interchange them if they are not in the required order.

For the list containing 'n' elements,

Number of passes : n - 1

Time complexity : $O(n^2)$

Space complexity : No extra space required

Worst case : List in reverse order

Comparisons : $O(n^2)$ Exchanges : $O(n^2)$

Best case : Already sorted list

Comparisons: $O(n^2)$, Exchanges: 0

Average case : List on random order

Comparisons : $O(n^2)$, Exchanges : $O(n^2)$

Comments : Good for small value of 'n'

ALGORITHM:

```
begin BubbleSort(list)
for all elements of list if
   list[i] > list[i+1]
   swap(list[i], list[i+1])
      end if end
   for
```

```
return list

end BubbleSort
```

Insertion sort:

Basic step:

To insert a record R into sequence of ordered records R1, R2, …, Ri, in such a way that resulting sequence of size (i+1) is also ordered.

For the list containing 'n' elements,

Number of passes : 1

Time complexity : O(n*n)

Space complexity : No extra space required

Worst case : List in reverse order

Comparisons : O(n*n), Push downs : O(n*n)

Best case : Already sorted list

Comparisons : O(n), Push downs : 0

Average case : List on random order

Comparisons : O(n*n), Push downs : O(n*n)

Comments : Good if the input list has very few entries Left Out of Order (LOO) **ALGORITHM:**

Step 1 – If it is the first element, it is already sorted. return 1;
Step 2 – Pick next element
Step 3 – Compare with all elements in the sorted sub-list **Step 4** – Shift all the elements in the sorted sub-list that is greater than the value to be sorted **Step 5** – Insert the value
Step 6 – Repeat until list is sorted

QUICK SORT : Quick sort scheme developed by C.A.R. Hoare, has the best average behaviour in terms of Time & Space Complexity among all the internal sorting methods.

BASIC STEP : In Quick Sort the key Ki (pivot – generally the first element in an array) controlling the process is placed at the right position with respect to the elements in an array i.e. if key Ki is placed at position s(i) then

$Kj < Ks(i)$ for $j < s(i)$ and

$Kj >= Ks(i)$ for $j > s(i)$

Thus after positioning of the element Ki has been made, the original array is partitioned into two sub arrays, one consisting of elements K1 to (Ks(i) – 1) and the other partition (Ks(i) + 1) to Kn. All the elements in first sub array are less than Ks(i) & those in the second sub array are greater or equal to Ks(i). These two sub arrays can be sorted independently

DIVIDE & CONQUER STRATEGY :

From the above description of Quick Sort, it is evident that Quick Sort is a good example of **Divide & Conquer** strategy for solving the problem. In Quick Sort, we divide the array of items to be sorted into two partitions and then call the same procedure recursively to sort the two partitions. Thus we **divide** the problem in two smaller problems and **conquer** by solving the smallest problem, in this case sorting of partition containing only one element at the end, which is trivial one. Thus Quick Sort is the best example of divide & conquer strategy, where the problem is solved (conquered) by dividing the original problem into two smaller problems and then applying the same strategy recursively until the problem is so small that it can be solved trivially. Thus the conquer part of the Quick Sort looks like this.

Initial Step

< Pivot1	**Pivot1**	**>= Pivot1**

Next Step

< Pivot2	**Pivot2**	**>= Pivot2**	**Pivot1**	**>= Pivot1**

EXAMPLE :

Pass No : **< -------- Input Array Elements -------- > Pivot m n i j**
Position : 0 1 2 3 4 5 6 7 8 9

Pass 1 :	**2**	**1**	**22**	**34**	**4 7**	**30 1**	**21 20**	2	0	9	1	9
Pass 2 :	**1**	**1**	2	34	4 7	30 22	21 20	1	0	1	1	1
Pass 3 :	1	1	2	**34**	**4 7**	**30 22**	**21 20**	34	3	9	4	9
Pass 4 :	1	1	2	**20**	**4 7**	**30 22**	**21** 34	20	3	8	4	8
Pass 5 :	1	1	2	**7**	**4** 20	30 22	21 34	7	3	4	4	4
Pass 6 :	1	1	2	4	7 20	**30 22**	**21** 34	30	6	8	7	8
Pass 7 :	1	1	2	4	7 20	**21 22**	30 34	30	6	7	7	7

Sorted Array : 1 1 2 4 7 20 21 22 30 34

1) Time Complexity :

a) Worst Case : Already sorted array is the worst case for Quick sort. Let us assume that there are 'n' elements in an array, then the number of elements in each partition for each pass will be,

Pass No	***Partitions***	***No. of comparisions***
Pass 1	: [0], [n - 1]	(n - 1)
Pass 2	: [0], [0], [n - 2]	(n - 2)
Pass 3	: [0],.[0], [0], [n - 3]	(n - 3)
.		.
.		.
.		.
Pass (n-1)	: [0],.[0], [0], …[0],[1]	1

Total number of comparisons would be = (n - 1) + (n - 2) + (n - 3) + …+ 1

= (n - 1) (n) / 2

= (n**2 / 2) – (n / 2)

= O (n**2) (ignoring lower order terms)

b) Best Case : Best case of Quick Sort would be the case where input elements in the array are such that in every pass **pivot** element is positioned at the middle such that the partition is divided in two partitions of equal size Then for a large value of n, the number of comparisons in each pass would be,

After 1st Pass : n comparisons + number of comparisons required to sort two Partitions of size n / 2.

= n + 2 * [n / 2]

After 2nd Pass = n + 2 * [n / 2 + 2 * (n / 4)]

= n + n + 4*[n / 4]

= 2n + 4*[n / 4]

After 3rd Pass = 2n + 4* [n / 4 + 2 * (n / 8)]

= 2n + n + 8*[n / 8]

= 3n + 8*[n / 8] = (log 8) * 8 + 8*[8/8] for 8 elements array

= **8*log8** since number of comparisons required to sort partition of size 1 would be zero.

. . .

. . .

. . .

After nth Pass = n*log(n)

c) Average case : Experimental results show that number of comparisons required to sort 'n' element array using Quick Sort are **O(nlogn)** to an average.

2) Space Complexity : In recursive quick sort, additional space for stack is required, which is of the order of **O(logn)** for **best and average case** and **O(n)** for **worst case.**

ALGORITHM:

```
void quicksort (int a[], int p, int q)
/* sorts the elements a[p]…,a[q] which reside in the global array a[1:n] into ascending
   order.*/
{
1. declare int j;
2. if(left >right) return;
3. j=partition(a, left ,right);
```

4. quicksort(a, left,j-1);
5. quicksort(a, j+1,right);

}

int part(int a[], int left ,int right)

/* within the array a[left: right] elements are arranged such that if pivot p is stored at location q then all elements less than p are stored from left to q-1 and all elements greater than p are stored from q+1 to right. */

{

1. declare int i,j,temp,v;
2. pivot=a[left]; i=left; j=right+1;
3. do
   {

   do

   {

   i++;

   } while(a[i]<pivot); do

   { j--;

   } while(a[j]>pivot);

   if(i<j)

   {

   temp=a[i]; a[i]=a[j];

   a[j]=temp;

   }

   }while(i<j);

4. a[left]=a[j];
5. a[j]=pivot;
6. // Display intermediate results
   Print partition position as j
   Print array
7. return(j);

}

Searching Techniques:

1. Sequential or Serial or Linear Search
2. Binary Search
3. Fibonacci Search

Sequential or Serial or Linear search:

In this method, entities are searched one by one starting from the first to the end. Searching stops as soon as we reach to the required entity or we reach to the end. For example, consider the array containing 'n' integers and to search a given number in array we start searching from the 1st element of the array to the end of array. Searching will stop as soon as the element of the array is equal to the given number (number found) or we reach the end of the array (number not found). This method is generally used when the **data stored is not in the order of the key** on which we want to search the table.

Binary Search:

If the **array is sorted on the key** & we want to search for a key having given value, then **Binary search** technique can be used which is more efficient as compared to serial search. Binary search uses the '**divide & conquer**' strategy as given below:

1) Divide the list into two equal halves.
2) Compare the given key value with the middle element of the array or list.
   There are three possibilities.
   a) middle element = key : key found and search terminates
   b) middle element > key : the value which we are searching is possibly in the first half of the list
   c) middle element < key : the value which we are searching is possibly in the second half of the list
3) Now search the key either in first half of second half of the list (b or c)
4) Repeat steps 2 and 3 till key is found or search fails in case key does not exist.

Binary Search:

a) Non recursive int bin_search_nonrec (char a[][20] , int

n, char key[20])

/* this procedure searches a record of given name from set of names and returns the record no to the calling program */

{

1) **Declare int** first, last, middle, passcnt=0;
2) **initialize** first=1, last =cnt; **3) while** (last >= first) {
   middle= (first + last)/2 **if** (

   strcmp(key, a[middle]) >0)

   first = middle + 1;

```
            else if (( strcmp(key, a[middle]) < 0) last
                    = middle – 1;
            else
                    return ( middle );
        }

    4) return( -1 ); //not found

}//end bin_search_nonrec
```

Conclusion:

Thus we have implemented bubble sort, insertion sort, quick sort and linear and binary search.

INPUT :

```
#include<iostream>
#include<string.h>
#include<conio.h>
using namespace std;

struct student{
   int roll_no;
   char name[15];
   float mark;
}s;

void create_database(student s[],int n);
void reply(student s[],int n);
void display_data(student s[],int n);
void bubble_sort(student s[],int n);
void insertion_sort(student s[],int n);
void quick_Sort(student s[],int low,int high);
void linearSearch(student s[],int n,int key);
void binarySearch(student s[],int n);
```

```
int main(){
   struct student s[20];
   cout<<"\t\t\tWELCOME"<<endl;

   int response,n;
   while(true){
      cout<<"\n1]Create a student database"<<endl;
      cout<<"2]Display the student database"<<endl;
      cout<<"3]Bubble sort the data by roll no"<<endl;
      cout<<"4]Sort names by insertion sort"<<endl;
                  cout<<"5]Quick sort the marks"<<endl;
      cout<<"6]Linear search on marks"<<endl;
      cout<<"7]Binary search on names"<<endl;
      cout<<"8]Exit"<<endl;
      cout<<"\nEnter the task = ";
      cin>>response;


      switch(response){

         case 1:
         cout<<endl;
         cout<<"Enter the number of students = ";
         cin>>n;
         create_database(s,n);
         break;

         case 2:
         cout<<endl;
         display_data(s,n);
         break;

         case 3:
```

```
        cout<<endl;
        bubble_sort(s,n);
        break;


        case 4:
        cout<<endl;
        insertion_sort(s,n);
        break;

                        case 5:
                        quick_Sort(s,0,n-1);
                        cout<<"\n** Information sorted succesfully  !! **"<<endl;
                        reply(s,n);
        break;

        case 6:
        int key;
        cout<<"Enter the marks to be found = ";
        cin>>key;
        linearSearch(s,n,key);
        break;

        case 7:
        binarySearch(s,n);
        break;

        case 8:
        return 0;

        default:
        cout<<"Enter the correct choice"<<endl;

    }
}
```

```
   return 0;
}


void create_database(student s[],int n){
      cout<<endl;

      for(int i=0;i<n;i++){
         cout<<"Student "<<i+1<<" :"<<endl;
                        cout<<"Enter the name    = ";
         cin>>s[i].name;

         cout<<"Enter the roll no = ";
         cin>>s[i].roll_no;

         cout<<"Enter the marks   = ";
         cin>>s[i].mark;
         cout<<endl;
         for(int j=i-1;j>=0;j--){
            if(s[i].roll_no==s[j].roll_no){
            i--;
            cout<<"enter the unique roll no."<<endl;
            break;
            }
         }
      }
}
void reply( student s[],int n){
         char re;
         cout<<"Want to display sorted data y or n = ";
         cin>>re;
         cout<<endl;
         if(re=='y'){
                  display_data(s,n);
                  cout<<endl;
```

```
        }
    else{
        return;
    }
}


void display_data(student s[],int n){



        cout<<"Roll no: "<<"\t\tName: "<<"\t\t\tMarks: "<<endl;

        for(int i=0;i<n;i++){
            cout<<s[i].roll_no<<"\t\t\t"<<s[i].name<<"\t\t\t"<<s[i].mark<<endl;
        }
        cout<<endl;
        cout<<"* Information displayed succesfully  !! *"<<endl;
        cout<<"\nPress any key to move further";
       getch();
       cout<<endl;
}
void bubble_sort(student s[],int n){

       student temp;
       int counter = 1;
       while (counter <n){
          for(int i=0;i<n;i++){
          if(s[i].roll_no>s[i+1].roll_no){
             temp = s[i];
             s[i] = s[i+1];
             s[i+1] = temp;
          }
       }
```

```
        counter ++;
        }
        cout<<endl;
        cout<<"* Information sorted succesfully  !! *"<<endl;

                    reply(s,n);
}
void insertion_sort(student s[],int n){

    student key;
    for(int i =1;i<n;i++){
        key=s[i];
        int j=i-1;

        while(j>=0 && strcmp(s[j].name,key.name)>0){
            s[j+1]=s[j];
            j=j-1;
        }
        s[j+1]=key;
    }
    cout<<"* Information sorted succesfully  !! *"<<endl;

          reply(s,n);
}


int partition (student s[], int low, int high) {

    int pivot = s[high].mark;
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++)
    {

        if (s[j].mark < pivot)
```

```
      {
         i++;
         ;
                        student temp;
                        temp = s[i];
                        s[i]=s[j];
                        s[j]=temp;
      }
   }

         student temp1;
         temp1=s[i+1];
         s[i+1]=s[high];
         s[high]=temp1;
   return (i + 1);
}


void quick_Sort(student s[], int low, int high) {

   if (low < high)
   {
      int pi = partition(s, low, high);
      quick_Sort(s, low, pi - 1);
      quick_Sort(s, pi + 1, high);
            }

}


void linearSearch(student s[],int n, int key){

   bool flag=true;

   cout<<"Roll no: "<<"\t\tName: "<<"\t\t\tMarks:"<<endl;
```

```
    for(int i =0;i<n;i++){

     if(s[i].mark==key){
        cout<<s[i].roll_no<<"\t\t\t"<<s[i].name<<"\t\t\t"<<s[i].mark<<endl;
         flag=false;
     }
    }
    if(flag){
         cout<<"NA"<<"\t\t\t"<<"NA"<<"\t\t\t"<<"NA"<<endl;
         cout<<"No data  found of entered marks"<<endl;

    }
    cout<<"/nPress any key to continue ";
    getch();

}

void binarySearch(student s[],int n){

    insertion_sort(s, n);
    cout<<endl;

    string key;
    cout<<"Enter the name to be found = ";
    cin>>key;
    cout<<endl;

    int start = 0;
    int end = n ;
    bool flag=true;
    cout<<"Roll no: "<<"\t\tName: "<<"\t\t\tMarks: "<<endl;
    while( start <= end){
      int mid = (start + end)/2;
      if(s[mid].name==key){
```

```
        flag=false;
        while(s[mid].name==key){

cout<<s[mid].roll_no<<"\t\t\t"<<s[mid].name<<"\t\t\t"<<s[mid].mark<<endl;
            mid--;
        }
        while(s[mid].name==key){

cout<<s[mid].roll_no<<"\t\t\t"<<s[mid].name<<"\t\t\t"<<s[mid].mark<<endl;
            mid++;
        }
        return;
     }
     else if(s[mid].name<key){
        start=mid+1;
     }
     else{
        end=mid-1;
     }
     cout<<"\nPress any key to continue "<<endl;
     getch();
   }
   if(flag){
      cout<<"NA"<<"\t\t\t"<<"NA"<<"\t\t\t"<<"NA"<<endl;
       cout<<"No data  found of entered marks"<<endl;
   }
   cout<<endl;
}
```

OUTPUT :

WELCOME

1]Create a student database
2]Display the student database
3]Bubble sort the data by roll no

4]Sort names by insertion sort
5]Quick sort the marks
6]Linear search on marks
7]Binary search on names
8]Exit

Enter the task = 1

Enter the number of students = 4

Student 1 :
Enter the name = g
Enter the roll no = 1
Enter the marks = 90

Student 2 :
Enter the name = s
Enter the roll no = 2
Enter the marks = 78

Student 3 :
Enter the name = a
Enter the roll no = 7
Enter the marks = 56

Student 4 :
Enter the name = r
Enter the roll no = 1
Enter the marks = 67

enter the unique roll no.
Student 4 :
Enter the name = r
Enter the roll no = 6
Enter the marks = 89

1]Create a student database
2]Display the student database
3]Bubble sort the data by roll no
4]Sort names by insertion sort
5]Quick sort the marks
6]Linear search on marks
7]Binary search on names
8]Exit

Enter the task = 2

Roll no:	Name:	Marks:
1	g	90
2	s	78
7	a	56
6	r	89

*** Information displayed succesfully !! ***

Press any key to move further

1]Create a student database
2]Display the student database
3]Bubble sort the data by roll no
4]Sort names by insertion sort
5]Quick sort the marks
6]Linear search on marks
7]Binary search on names
8]Exit

Enter the task = 3

*** Information sorted succesfully !! ***
Want to display sorted data y or n = y

Roll no:	Name:	Marks:
1	g	90
2	s	78
6	r	89
7	a	56

*** Information displayed succesfully !! ***

Press any key to move further

1]Create a student database
2]Display the student database
3]Bubble sort the data by roll no
4]Sort names by insertion sort
5]Quick sort the marks
6]Linear search on marks
7]Binary search on names
8]Exit

Enter the task = 4

*** Information sorted succesfully !! ***

Want to display sorted data y or n = y

Roll no:	Name:	Marks:
7	a	56
1	g	90
6	r	89
2	s	78

*** Information displayed succesfully !! ***

Press any key to move further

1]Create a student database
2]Display the student database
3]Bubble sort the data by roll no
4]Sort names by insertion sort
5]Quick sort the marks
6]Linear search on marks
7]Binary search on names
8]Exit

Enter the task = 5

**** Information sorted succesfully !! ****
Want to display sorted data y or n = y

Roll no:	Name:	Marks:
7	a	56
2	s	78
6	r	89
1	g	90

*** Information displayed succesfully !! ***

Press any key to move further

1]Create a student database
2]Display the student database
3]Bubble sort the data by roll no
4]Sort names by insertion sort
5]Quick sort the marks
6]Linear search on marks
7]Binary search on names
8]Exit

Enter the task = 6

```
Enter the marks to be found = 56
Roll no:        Name:        Marks:
7               a            56
/nPress any key to continue
1]Create a student database
2]Display the student database
3]Bubble sort the data by roll no
4]Sort names by insertion sort
5]Quick sort the marks
6]Linear search on marks
7]Binary search on names
8]Exit

Enter the task = 7
*** Information sorted succesfully  !! ***
Want to display sorted data y or n = y

Roll no:        Name:        Marks:
7               a            56
1               g            90
6               r            89
2               s            78

*** Information displayed succesfully  !! ***

Press any key to move further


Enter the name to be found = s

Roll no:        Name:        Marks:

Press any key to continue
2               s            78

1]Create a student database
2]Display the student database
3]Bubble sort the data by roll no
4]Sort names by insertion sort
5]Quick sort the marks
6]Linear search on marks
7]Binary search on names
8]Exit

Enter the task = 8
PS D:\C++ GB\Assignments>
```

Assignment 2 -Implementation of Stack AIM:

Implementation of Stack ADT and expression conversion

DETAILED PROBLEM STATEMENT:

Write a program to implement stack as an abstract data type using linked list and use this ADT for conversion of infix expression to postfix, prefix and evaluation of postfix/prefix expression.

OBJECTIVE:

1. To understand the concept and implementation of Stack
2. data structure using SLL.
3. To understand the concept of conversion of expression.
4. To understand the concept of evaluation of expression
5. To compare and analyze time and space complexity

OUTCOME:

1. Code linear data structure using array of structure.
2. Apply different sorting techniques on array of structure (Bubble, 3. Insertion and Quicksort) and display the output for every pass.
4. Apply different searching techniques on array of structure (Linear Search, Binary Search) and display the output for every pass..
5. Calculate time complexity.

THEORY:

What is an abstract datatype?

An Abstract Data type is defined as a mathematical model of the data objects that make up a data type as well as the functions that operate on these objects. There are no standard conventions for defining them. A broad division may be drawn between "imperative" and "functional" definition styles. In general terms, an abstract data type is a *specification* of the values and the operations that has two properties:

- It specifies everything you need to know in order to use the datatype
- It makes absolutely no reference to the manner in which the data type will be implemented.

When we use abstract data types, our programs divide into two pieces:

- The Application: The part that uses the abstract datatype.
- The Implementation: The part that implements the abstract datatype.

Concept of Linear data structure using linked organization

Linked lists provides a way to represent an ordered or linear list, in which each item in the list is a part of structure that also contains a "link" to the structure containing the next item. Each item has only one predecessor and only one successor.

Each structure of the list is called as a 'node' & it consists of two fields,

i) One containing the 'item' & ii)

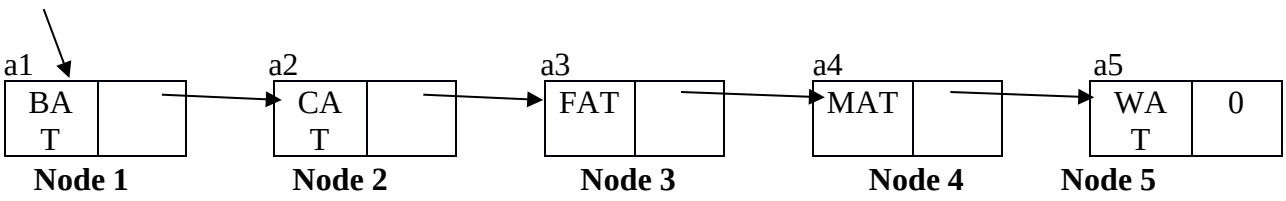
Second containing the 'address' of the next 'node'.

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

```
node
{
        <data    type>    item;
        struct node *next;
};
```

For example, consider the following ordered list stored as a linked list

L = {BAT, CAT, FAT, MAT, WAT}



Dynamic Memory Management:

1) Allocating a block of memory using new operator:

new operator allocates memory of specified size & returns a pointer of specified type . The format is,

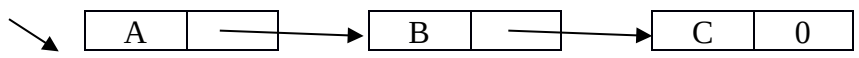
node *newp; newp = new node;

where 'newp' is a pointer of type node, pointing to a block of memory of size node and it is NULL if not enough space is available.

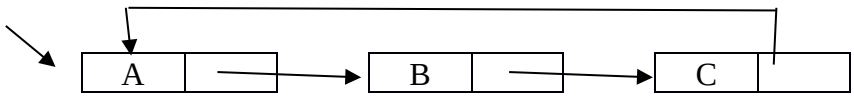
2) Releasing the used memory space using delete operator: The format is, delete newp; will release a memory block allocated by new operator pointed by pointer 'newp'.

Types of Linked Lists:

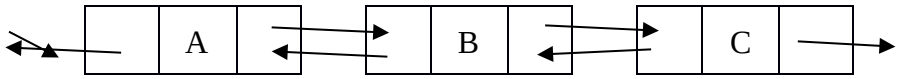
1. Singly Linked List (**SLL**)



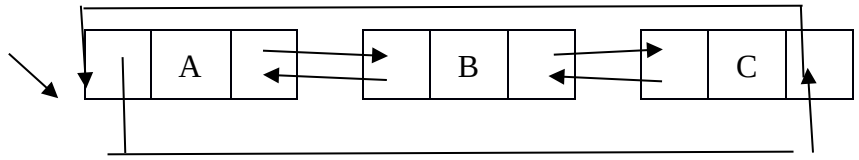
2. Circular Singly Linked List (**CSLL**)



3. Doubly Linked List (**DLL**) head



4. Circular Doubly Linked List



(**CDLL**) head

Example of Linear data structure

Data structure whose elements (objects) are sequential and ordered in a way so that:

- there is only one *first element* and has only one *next element*,
- there is only one *last element* and has only one *previous element*, while all other elements have a *next* and a *previous* element

Arrays, Strings, Stack, Queue, Lists

1. **Stack :**

i. **Concept**

Stacks are more common data objects found in computer algorithms. It is a special case of more general data object, an ordered or linear list. It can be defined as an ordered list, in which all insertions & deletions are made at one end, called as 'top' i.e. Last element inserted is outputted first (**L**ast **I**n **F**irst **O**ut or **LIFO**).

Stack can be represented mathematically as:

S = {a_1, a_2, …,a_n} where a_1 is the bottommost element & a_n is topmost element. & a_{i+1} is on the top of the element a_i , 1 < i <= n.

ii. **Definition of stack**

In computer science, a **stack** is a last in, first out (LIFO) abstract data type and data structure. A stack can have any abstract data type as an element, but is characterized by only two fundamental operations: push and pop. The push operation adds an item to the top of the stack, hiding any items already on the stack, or initializing the stack if it is empty. A pop either reveals previously concealed items, or results in an empty stack. A stack is a restricted data structure, because only a small number of operations are performed on it. The nature of the pop and push operations also mean that stack elements have a natural order. Elements are removed from the stack in the reverse order to the order of their addition: therefore, the lower elements are those that have been on the stack the longest. A collection of items in which only the most recently added item may be removed. The latest added item is at the top. Basic operations are push and pop. Often top and is Empty are available, too. Also known as "last-in, first-out" or LIFO.

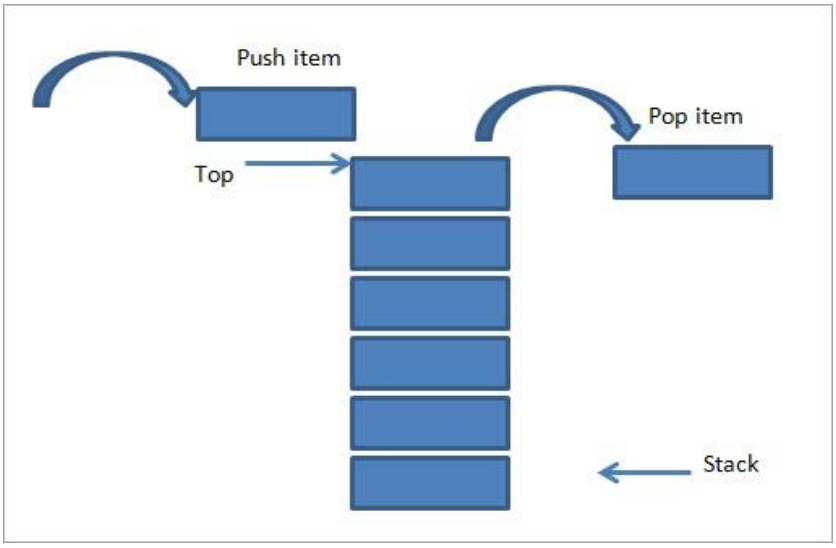
iii. **Terminology with stack**

An abstract data type (ADT) consists of a data structure and a set of **primitives**

a. **Initialize** creates/initializes the stack
b. **Push** adds a new element
c. **Pop** removes a element
d. **IsEmpty** reports whether the stack is empty
e. **IsFull** reports whether the stack is full
f. **Destroy** deletes the contents of the stack (may be implemented by re-initializing the stack)

iv. **Diagram (vertical view)**

Given below is a pictorial representation of Stack.



As shown above, there is a pile of plates stacked on top of each other. If we want to add another item to it, then we add it at the top of the stack as shown in the above figure (left-hand side). This operation of adding an item to stack is called "**Push**".

On the right side, we have shown an opposite operation i.e. we remove an item from the stack. This is also done from the same end i.e. the top of the stack. This operation is called "**Pop**".

As shown in the above figure, we see that push and pop are carried out from the same end. This makes the stack to follow LIFO order. The position or end from which the items are pushed in or popped out to/from the stack is called the "**Top of the stack**".

Initially, when there are no items in the stack, the top of the stack is set to -1. When we add an item to the stack, the top of the stack is incremented by 1 indicating that the item is added. As opposed to this, the top of the stack is decremented by 1 when an item is popped out of the stack

ADT of Stack

Define Structure for stack(Data, Next Pointer) ➢

Stack Empty:

Return True if Stack Empty else False.

Top is a pointer of type structure stack.

Empty(Top)

1. if Top == NULL
2. return 1
3. else
4. return 0

➢ **Push Operation:**

Top & Node pointer of structure Stack.

Push(element)

1. Node->data = element; //
2. Node->Next = Top;
3. Top = Node
4. Stop.

➢ **Pop Operation:**

Top & Temp pointer of structure Stack.

Pop()

1. if Top != NULL

i. Temp = Top; ii.

element=Temp->data; iii.

Top = (Top)->Next; iv.

delete temp;

2. Else Stack is Empty.
3. return element;

Realization of Stack ADT

i. Using Sequential organization (Array)

The three basic stack operations are push, pop, and stack top. Push is used to insert data into the stack. Pop removes data from a stack and returns the data to the calling module. Stack top returns the data at the top of t`he stack without deleting the data from the stack.

Push

Push adds an item at the top of the stack. After the push, the new item becomes the top. The only potential problem with this simple operation is that we must ensure that there is room for the new item. If there is not enough room, the stack is in an overflow state and the item cannot be added.

Following figure shows the push stack operation.

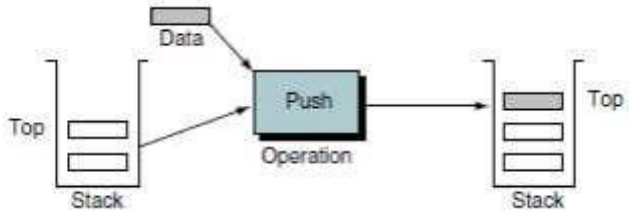


Fig.: Push Stack operation

Pop

When we pop a stack, we remove the item at the top of the stack and return it to the user. Because we have removed the top item, the next older item in the stack becomes the top. When the last item in the stack is deleted, the stack must be set to its empty state. If pop is called when the stack is empty, it is in an underflow state. The pop stack operation is shown in following figure

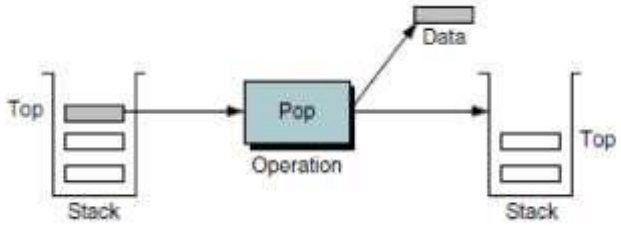


Fig.: Pop Stack operation

Stack top

The third stack operation is stack top. Stack top copies the item at the top of the stack; that is, it returns the data in the top element to the user but does not delete it. You might think of this operation as reading the

stack top. Stack top can also result in underflow if the stack is empty. The stack top operation is shown in following figure

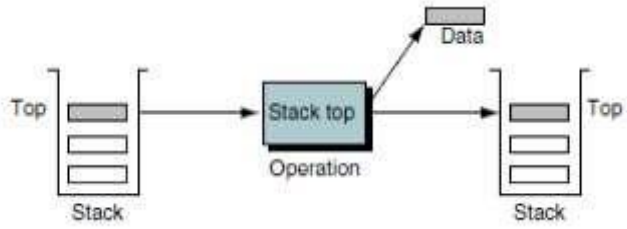


Fig.: Stack top operation

Following figure traces these three operations in an example. We start with an empty stack and push *green* and *blue* into the stack. At this point the stack contains two entries. We then pop *blue* from the top of the stack, leaving *green* as the only entry. After pushing *red*, the stack again contains two entries. At this point we retrieve the top entry, *red*, using stack top. Note that stack top does not remove *red* from the stack; it is still the top element. We then pop *red*, leaving *green* as the only entry. When *green* is popped, the stack is again empty. Note also how this example demonstrates the last in–first out operation of a stack.

Although *green* was pushed first, it is the last to be popped.

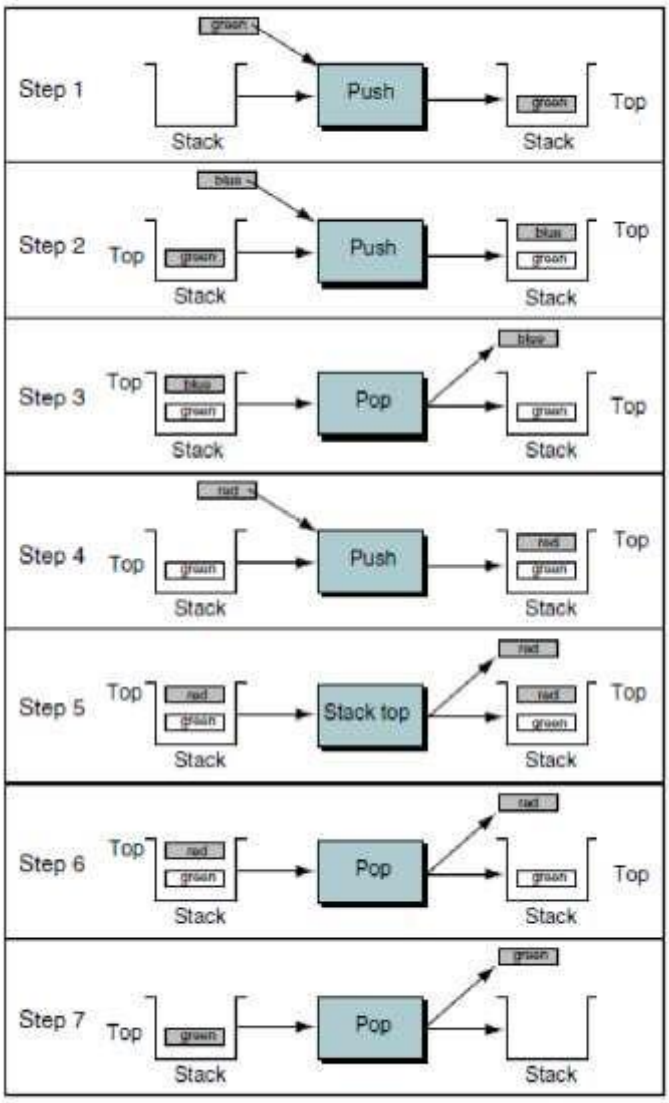


Fig.: Stack Example

ii. Using Linked organization (Linked list)

To implement the linked list stack, we need two different structures, a head and a data node. The head structure contains metadata—that is, data about data—and a pointer to the top of the stack. The data structure contains data and a link pointer to the next node in the stack. The conceptual and physical implementations of the stack are shown in following figure.

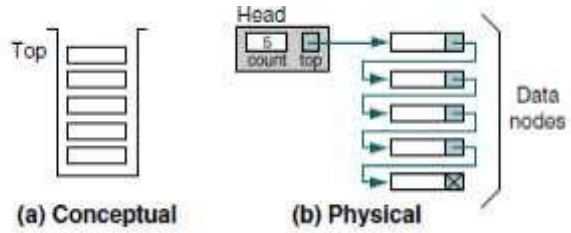


Fig.: Conceptual and Physical Stack Implementations

Stack Head

Generally, the head for a stack requires only two attributes: a top pointer and a count of the number of elements in the stack. These two elements are placed in a structure. Other stack attributes can be placed here also. For example, it is possible to record the time the stack was created and the total number of items that have ever been placed in the stack. These two metadata items allow the user to determine the average number of items processed through the stack in a given period. Of course, we would do this only if such a statistic were required for some reason. A basic head structure is shown in Figure.

Stack Data Node

The rest of the data structure is a typical linked list data node. Although the application determines the data that are stored in the stack, the stack data node looks like any linked list node. In addition to the data, it contains a link pointer to other data nodes, making it a self-referential data structure. In a self-referential structure, each instance of the structure contains a pointer to another instance of the same structure. The stack data node is also shown in following figure

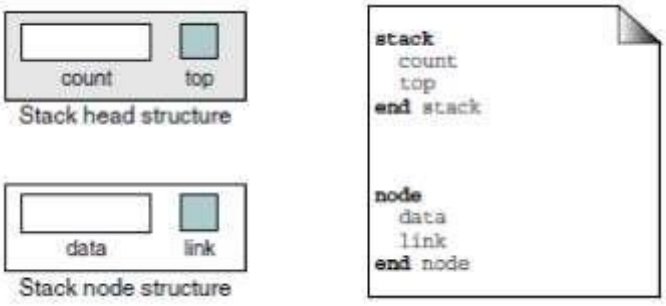


Fig.: Stack head, Stack Data node structure

We use the design shown in following figure, which demonstrates the four most common stack operations: create stack, push stack, pop stack, and destroy stack. Operations such as stacktop are not shown in the figure because they do not change the stack structure.

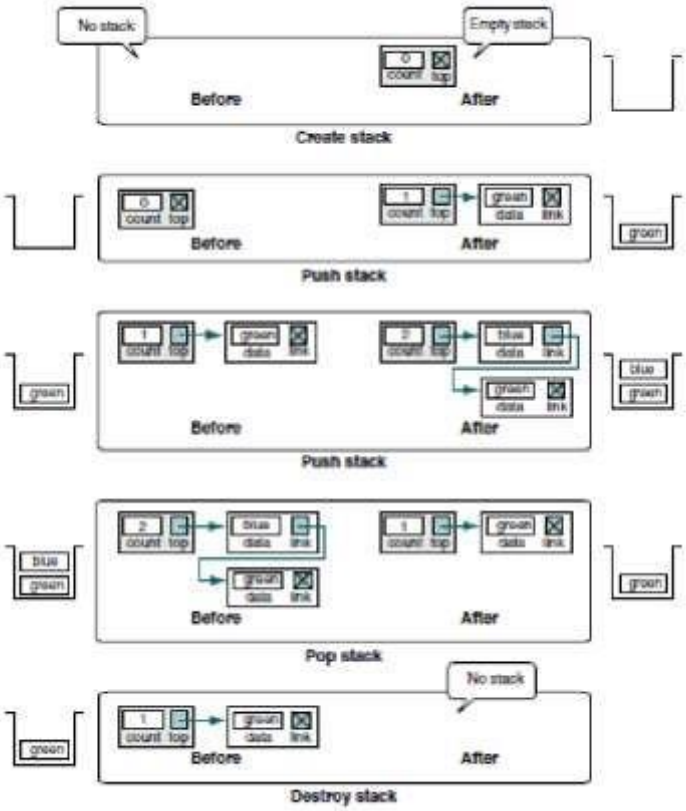


Fig.: Stack Operations

Push Stack

Push stack inserts an element into the stack. The first thing we need to do when we push data into a stack is find memory for the node. We must therefore allocate a node from dynamic memory. Once the memory is allocated, we simply assign the data to the stack node and then set the link pointer to point to the node currently indicated as the stack top. We also need to update the stack top pointer and add 1 to the stack count field. Figure traces a push stack operation in which a new pointer (pNew) is used to identify the data to be inserted into the stack.

To develop the insertion algorithm using a linked list, we need to analyze three different stack conditions: (1) insertion into an empty stack, (2) insertion into a stack with data, and (3) insertion into a stack when the available memory is exhausted. The first two of these situations are shown in Figure 3-8. The third is an error condition.

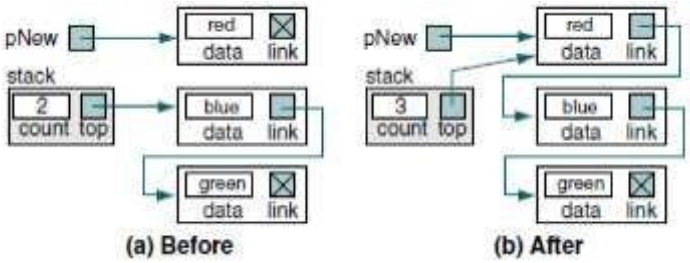


Fig.: Push Stack Example

When we insert into a stack that contains data, the new node's link pointer is set to point to the node currently at the top, and the stack's top pointer is set to point to the new node. When we insert into an empty stack, the new node's link pointer is set to null and the stack's top pointer is set to point to the new node. However, because the stack's top pointer is null, we can use it to set the new node's link pointer to null. Thus the logic for inserting into a stack with data and the logic for inserting into an empty stack are identical.

Pop Stack

Pop stack sends the data in the node at the top of the stack back to the calling algorithm. It then adjusts the pointers to logically delete the node. After the node has been logically deleted, it is physically deleted by recycling the memory that is, returning it to dynamic memory. After the count is adjusted by subtracting 1, the algorithm returns the status to the caller: if the pop was successful, it returns true; if the stack is empty when pop is called, it returns false. The operations for pop stack are traced in Figure.

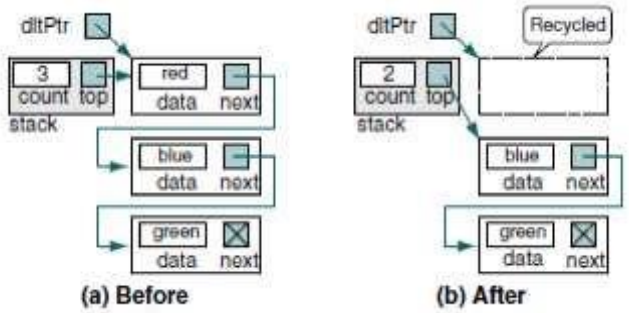


Fig.: Pop Stack Example

Application of Stack

Stack applications can be classified into four broad categories:

I. reversing data (Conversion of decimal number into binary number)

II. parsing data (Parenthesis matching)

III. postponing data usage (Expression conversion and evaluation)

IV. backtracking steps (Goal seeking and the eight queens problem)

Expression conversion and stack

I. Need for expression conversion

One of the disadvantages of the infix notation is that we need to use parentheses to control the evaluation of the operators. We thus have an evaluation method that includes parentheses and two operator priority classes. In the postfix and prefix notations, we do not need parentheses; each provides only one evaluation rule.

Although some high-level languages use infix notation, such expressions cannot be directly evaluated. Rather, they must be analyzed to determine the order in which the expressions are to be evaluated.

II. What is Polish Notation?

Conventionally, we use the operator symbol between its two operands in an arithmetic expression.

A+B C–D*E A*(B+C)

We can use parentheses to change the precedence of the operators. Operator precedence is pre-defined. This notation is called INFIX notation. Parentheses can change the precedence of evaluation. Multiple passes required for evaluation. Named after Polish mathematician Jan Named after Polish mathematician Jan Lukasiewicz

Reverse Polish (POSTFIX) notation refers to the notation in which the operator symbol is placed after its two operands.

AB+ CD* AB*CD+/

Polish PREFIX notation refers to the notation in which the operator symbol is placed before its two operands.

+AB *CD /*AB-CD

III. Advantages Polish notations over infix expression

a. No concept of operator priority.

b. Simplifies the expression evaluation rules.

c. No need of any parenthesis, Hence no ambiguity in the order of evaluation.

d. Evaluation can be carried out using a single scan over the expression string.

IV. Conversion of infix to postfix.

We can also use a stack to convert an expression in standard form (otherwise known as infix) into postfix. We will concentrate on a small version of the general problem by allowing only the operators +, *, (,), and insisting on the usual precedence rules. We will further assume that the expression is legal. Suppose we want to convert the infix expression a + b * c + (d * e + f) * g

into postfix. A correct answer is a b c * + d e * f + g * +.

When an operand is read, it is immediately placed onto the output. Operators are not immediately output, so they must be saved somewhere. The correct thing to do is to place operators that have been seen, but not placed on the output, onto the stack. We will also stack left parentheses when they are encountered. We start with an initially empty stack.

If we see a right parenthesis, then we pop the stack, writing symbols until we encounter a (corresponding) left parenthesis, which is popped but not output.

If we see any other symbol (+, *, (), then we pop entries from the stack until we find an entry of lower priority. One exception is that we never remove a (from the stack except when processing a). For the purposes of this operation, + has lowest priority and (highest.

When the popping is done, we push the operator onto the stack.

Finally, if we read the end of input, we pop the stack until it is empty, writing symbols onto the output.

The idea of this algorithm is that when an operator is seen, it is placed on the stack. The stack represents pending operators. However, some of the operators on the stack that have high precedence are now known to be completed and should be popped, as they will no longer be pending. Thus prior to placing the operator on the stack, operators that are on the stack, and which are to be completed prior to the current operator, are popped.

This is illustrated in the following table:

Expression	Stack When Third Operator Is Processed	Action
a*b-c+d	-	- is completed; + is pushed
a/b+c*d	+	Nothing is completed; * is pushed
a-b*c/d	- *	*is completed; / is pushed
a-b*c+d	- *	*and - are completed; + is pushed

Parentheses simply add an additional complication. We can view a left parenthesis as a high-precedence operator when it is an input symbol (so that pending operators remain pending) and a low-precedence operator when it is on the stack (so that it is not accidentally removed by an operator). Right parentheses are treated as the special case.

To see how this algorithm performs, we will convert the long infix expression above into its postfix form. First, the symbol a is read, so it is passed through to the output.

Then + is read and pushed onto the stack. Next b is read and passed through to the output.

The state of affairs at this juncture is as follows:

Next, a * is read. The top entry on the operator stack has lower precedence than *, so nothing is output and * is put on the stack. Next, c is read and output. Thus far, we have

The next symbol is a +. Checking the stack, we find that we will pop a * and place it on the output; pop the other +, which is not of lower but equal priority, on the stack; and then push the +.

The next symbol read is a (. Being of highest precedence, this is placed on the stack. Then d is read and output.

We continue by reading a *. Since open parentheses do not get removed except when a closed parenthesis is being processed, there is no output. Next, e is read and output.

The next symbol read is a +. We pop and output * and then push +. Then we read and output f.

Now we read a), so the stack is emptied back to the (. We output a +.

We read a * next; it is pushed onto the stack. Then g is read and output.

The input is now empty, so we pop and output symbols from the stack until it is empty.

As before, this conversion requires only O(N) time and works in one pass through the input. We can add subtraction and division to this repertoire by assigning subtraction and addition equal priority and multiplication and division equal priority. A subtle point is that the expression a - b - c will be converted to a b - c - and not a b c - -. Our algorithm does the right thing, because these operators associate from left to right. This is not necessarily the case in general, since exponentiation associates right to left: $2^{2^3} = 2^8 = 256$, not $4^3 = 64$. We leave as an exercise the problem of adding exponentiation to the repertoire of operators.

Let's work on one more example before we formally develop the algorithm.

A + B * C – D / E converts to A B C * + D E / -

The transformation of this expression is shown in following figure. Because it uses all of the basic arithmetic operators, it is a complete test.

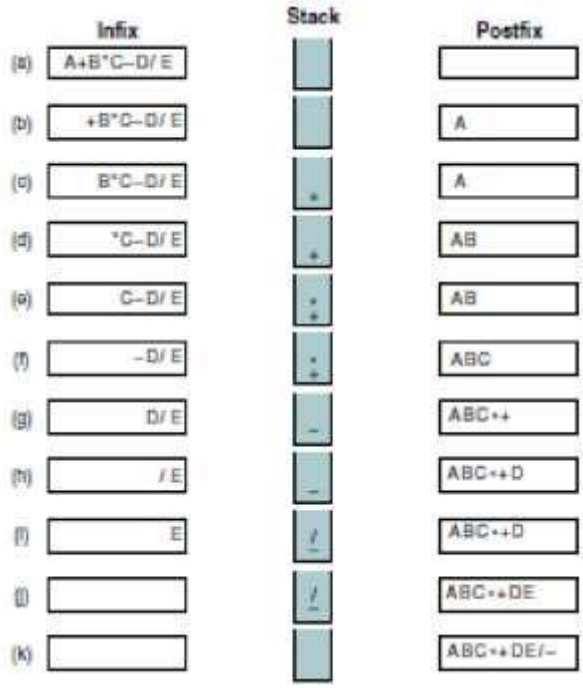


Fig.: Infix Transformation

We begin by copying the first operand, A, to the postfix expression. See Figure (b). The add operator is then pushed into the stack and the second operand is copied to the postfix expression. See Figure (d). At this point we are ready to insert the multiply operator into the stack. As we see in Figure (e), its priority is higher than that of the add operator at the top of the stack, so we simply push it into the stack. After copying the next operand, C, to the postfix expression, we need to push the minus operator into the stack. Because its priority is lower than that of the multiply operator, however, we must first pop the multiply and copy it to the postfix expression. The plus sign is now popped and appended to the postfix expression because the minus and plus have the same priority. The minus is then pushed into the stack. The result is shown in Figure (g). After copying the operand D to the postfix expression, we push the divide operator into the stack because it is of higher priority than the minus at the top of the stack in Figure (i).

After copying E to the postfix expression, we are left with an empty infix expression and two operators in the stack. See Figure (j). All that is left at this point is to pop the stack and copy each operator to the postfix expression. The final expression is shown in Figure (k).

ALGORRITHMS/PESUDOCODE :

We are now ready to develop the algorithm. We assume only the operators shown below. They have been adapted from the standard algebraic notation.

Priority 2: * /

Priority 1: + -

Priority 0: (

- **Convert Infix expression to Postfix expression**

Infix_TO_ Postfix (Input Expression) //Convert infix Expression to postfix.

//Pre Expression is infix notation that has been edited

//to ensure that there are no syntactical errors

//Post postfix Expression has been formatted as a string

Return postfix Expression

Step 1: create Stack (stack)

Step 2: loop (for each character in Expression)

i. if (character is open parenthesis)

i. pushStack (stack, character)

ii. elseif (character is close parenthesis)

a. popStack (stack, character)

b. loop (character not open parenthesis)

i. concatenate character to postFixExpr ii. popStack (stack, character)

c. end loop iii. elseif (character is operator) //Test priority of token to token at top of stack

a. stackTop (stack, topToken)

b. loop (not emptyStack (stack) AND priority(character) <= priority(topToken))

i. popStack (stack, tokenOut) ii. concatenate tokenOut to postFixExpr iii. stackTop (stack, topToken)

c. end loop

d. pushStack (stack, token)

v. else // Character is operand

a. Concatenate token to postFixExpr

vi. end if

Step 3: end loop //Input Expression empty.

//Pop stack to postFix

Step 4: loop (not emptyStack (stack))

i. popStack (stack, character) ii.

concatenate token to postFixExpr

Step 5: end loop

Step 6: return postFix

//end inToPostFix

- **Evaluation postfix and infix with example**

Now let's see how we can use stack postponement to evaluate the postfix expressions we developed earlier.

For example, given the expression shown below,

A B C + * and assuming that A is 2, B is 4, and C is 6, what is the

expression value?

The first thing you should notice is that the operands come before the operators. This means that we will have to postpone the use of the operands this time, not the operators. We therefore put them into the stack. When we find an operator, we pop the two operands at the top of the stack and perform the operation. We then push the value back into the stack to be used later.

Following figure traces the operation of our expression. (Note that we push the operand values into the stack, not the operand names. We therefore use the values in the figure.)

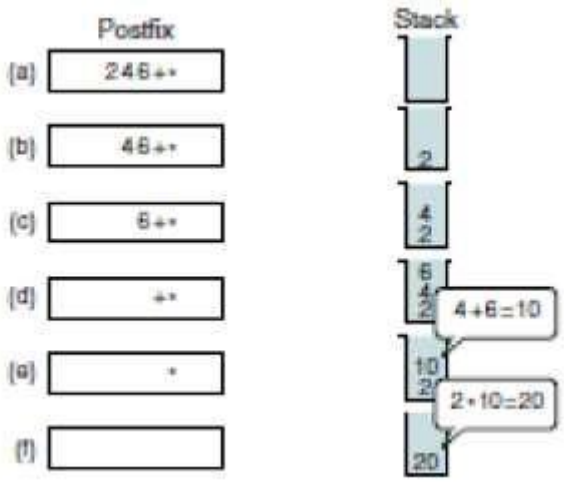


Fig.: Evaluation of Postfix expression

When the expression has been completely evaluated, its value is in the stack. The

algorithm is shown below

- **Evaluation of Postfix Expressions postFixEvaluate**

(input Postfix _expr)

//This algorithm evaluates a postfix expression and returns its value.

//Pre a valid expression

//Post postfix value computed

//Return value of expression

Step 1 :createStack (stack)

Step 2 : loop (for each character)

a. if (character is operand)

i. pushStack (stack, character)

b.2 else

i. popStack (stack, oper2) ii. popStack (stack, oper1) iii. operator = character iv. set value to calculate (oper1, operator, oper2) v. pushStack (stack, value)

c. end if

Step 3: end loop

Step 4 : popStack (stack, result)

Step 5: return (result)

//end postFixEvaluate

Test cases/validation

Validation

I. If Stack Empty	Display message “Stack Empty”
II. If memory not available	Display message “memory not available”
III Parenthesis matching	Display appropriate message open/close parenthesis missing

Infix to postfix /prefix Test Cases

INPUT:	**POSTFIX OUTPUT:**
(A+B) * (C-D)	AB+CD-*
A$B*C-D+E/F/(G+H)	AB$C*D-EF/GH+/+
((A+B)*C-(D-E))$(F+G)	AB+C*DE—FG+$
A-B/(C*D$E)	ABCDE$*/-
A^B^C	ABC^^
INPUT:	**PREFIX OUTPUT:**
(A+B) * (C-D)	*+AB-CD
A$B*C-D+E/F/(G+H)	+-*$ABCD//EF+GH
((A+B)*C-(D-E))$(F+G)	$-*+ABC-DE+FG
A-B/(C*D$E)	-A/B*C$DE
A^B^C	^A^BC

CONCLUSION:

Thus we have implemented Stack ADT and expression conversion .

INPUT:

```
  #include <iostream>
#define max 30
using namespace std;
 typedef struct node
{
   char data;
   struct node *next;
} node;
class Stack
{
   node *top;

public:
   Stack()
   {
      top = NULL;
   }
```

```
    int isempty()
    {
        if (top == NULL)
            return 1;
        return 0;
    }
    void push(char x)
    {
        node *n;
        n = new node();
        n->data = x;
        n->next = top;
        top = n;
    }
    char pop()
    {
        node *n;
        char x;
        n = top;
        x = n->data;
        top = top->next;
        delete (n);
        return x;
    }
    char topdata()
    {
        return top->data;
    }
};

void infix_postfix(char infix[], char postfix[]);
void reverse(char a[], char b[]);
void infix_prefix(char infix[], char prefix[]);
int evaluate(int op1, int op2, char op);
void evaluate_postfix(char postfix[]);
int precedence(char x);
void evaluate_prefix(char prefix[]);

int main()
```

```
{
   char infix[20], tok, postfix[20], prefix[20];
   int ch, result;
   do
   {
      cout <<endl<< "1. Infix to Postfix "<<endl;
      cout<< "2. Infix to Prefix "<<endl;
      cout<< "3. Evaluate Postfix "<<endl;
      cout<< "4. Evaluate Prefix "<<endl;
      cout<< "5. Exit"<<endl;
      cout<< "Enter your choice: ";
      cin >> ch;
      switch (ch)
      {
      case 1:
         cout<<endl<<"Enter Infix expression :";
         cin >> infix;
         infix_postfix(infix, postfix);
         cout<<endl<< "Postfix expression : " << postfix<<endl;
         break;
      case 2:
         cout<< "Enter infix expression :";
         cin >> infix;
         infix_prefix(infix, prefix);
         cout<<endl<<"Prefix expression : " << prefix<<endl;
         break;
      case 3:
         evaluate_postfix(postfix);
         break;
      case 4:
         evaluate_prefix(prefix);
         break;
      case 5: break;
      default: cout<<endl<<"Invalid Choice !!! "<<endl;
      break;
      }
   } while (ch != 5);
   return 0;
}
```

```
void infix_postfix(char infix[20], char postfix[20])
{
   Stack s;
   int i, j = 0;
   char tok, x;
   for (i = 0; infix[i] != '\0'; i++)
   {
      tok = infix[i];
      if (isalnum(tok))
      {
         postfix[j] = tok;
         j++;
      }
      else
      {
         if (tok == '('
            s.push(tok);
         else if (tok == ')')
         {
            while ((x = s.pop()) != '('
            {
               postfix[j] = x;
               j++;
            }
         }
         else
         {
            while (s.isempty() != 1 && precedence(tok) <= precedence(s.topdata()))
            {
               postfix[j] = s.pop();
               j++;
            }
            s.push(tok);
         }
      }
   }
   while (s.isempty() != 1)
   {
      postfix[j] = s.pop();
```

```
        j++;
    }
    postfix[j] = '\0';
}
void infix_postfix1(char infix[20], char postfix[20])
{
    Stack s;
    int i, j = 0;
    char tok, x;
    for (i = 0; infix[i] != '\0'; i++)
    {
        tok = infix[i];
        if (isalnum(tok))
        {
            postfix[j] = tok;
            j++;
        }
        else
        {
            if (tok == '(')
                s.push(tok);
            else if (tok == ')')
            {
                while ((x = s.pop()) != '(')
                {
                    postfix[j] = x;
                    j++;
                }
            }
            else
            {
                while (s.isempty() != 1 && precedence(tok) <precedence(s.topdata()))
                {
                    postfix[j] = s.pop();
                    j++;
                }
                s.push(tok);
            }
        }
```

```
    }
    while (s.isempty() != 1)
    {
        postfix[j] = s.pop();
        j++;
    }
    postfix[j] = '\0';
}
void reverse(char a[20], char b[20])
{
    int i, j = 0;
    for (i = 0; a[i] != '\0'; i++)
    {
    }
    i--;
    for (j = 0; i >= 0; j++, i--)
    {
        if (a[i] == '(')
            b[j] = ')';
        else if (a[i] == ')')
            b[j] = '(';
        else
            b[j] = a[i];
    }
    b[j] = '\0';
}

void infix_prefix(char infix[20], char prefix[20])
{
    char prefix1[20], infix1[20];
    reverse(infix, infix1);
    infix_postfix1(infix1, prefix1);
    reverse(prefix1, prefix);
}

int precedence(char x)
{
    if (x == '(')
        return 0;
```

```
    if (x == '+' || x == '-')
        return 1;
    if (x == '*' || x == '/')
        return 2;
    return 3;
}

int evaluate(int op1, int op2, char op)
{
    if (op == '+')
        return op1 + op2;
    if (op == '-')
        return op1 - op2;
    if (op == '*')
        return op1 * op2;
    if (op == '/')
        return op1 / op2;
    if (op == '%')
        return op1 % op2;

    return 0;
}

void evaluate_postfix(char postfix[20])
{
    Stack s;
    int i, op1, op2, result;
    char tok;
    int x;
    for (i = 0; postfix[i] != '\0'; i++)
    {
        tok = postfix[i];
        if (isalnum(tok))
        {
            cout <<"Enter "<< tok<<": ";
            cin >> x;
            s.push(char(x));
        }
        else
```

```
        {
            op2 = s.pop();
            op1 = s.pop();
            result = evaluate(op1, op2, tok);
            s.push(char(result));
        }
    }
    result = s.pop();
    cout <<endl<<"Simplified value of postfix expression is :  "<<result<<endl;
}

void evaluate_prefix(char prefix[20])
{
    Stack s;
    int i, op1, op2, result;
    char tok;
    int x;
    for (i = 0; prefix[i] != '\0'; i++)
    {
    }
    i--;

    for (; i >= 0; i--)
    {
        tok = prefix[i];

        if (isalnum(tok))
        {
            cout << "Enter " << tok<<" : ";
            cin >> x;
            s.push(char(x));
        }
        else
        {
            op1 = s.pop();
            op2 = s.pop();
            result = evaluate(op1, op2, tok);
            s.push(char(result));
        }
```

```
    }
    result = s.pop();
    cout << "Simplified value of prefix epression is : " << result<<endl;
}
```

OUTPUT:

```
1. Infix to Postfix
2. Infix to Prefix
3. Evaluate Postfix
4. Evaluate Prefix
5. Exit
Enter your choice: 1

Enter Infix expression :3+(5-2)*4-8/2+5

Postfix expression : 352-4*+82/-5+

1. Infix to Postfix
2. Infix to Prefix
3. Evaluate Postfix
4. Evaluate Prefix
5. Exit
Enter your choice: 2
Enter infix expression :3+(5-2)*4-8/2+5

Prefix expression : +-+3*-524/825

1. Infix to Postfix
2. Infix to Prefix
3. Evaluate Postfix
4. Evaluate Prefix
5. Exit
Enter your choice: 3
Enter 3: 3
Enter 5: 5
```

Enter 2: 2
Enter 4: 4
Enter 8: 8
Enter 2: 2
Enter 5: 5

Simplified value of postfix expression is : 16

1. Infix to Postfix
2. Infix to Prefix
3. Evaluate Postfix
4. Evaluate Prefix
5. Exit
Enter your choice: 4
Enter 5 : 5
Enter 2 : 2
Enter 8 : 8
Enter 4 : 4
Enter 2 : 2
Enter 5 : 5
Enter 3 : 3
Simplified value of prefix epression is : 16

1. Infix to Postfix
2. Infix to Prefix
3. Evaluate Postfix
4. Evaluate Prefix
5. Exit
Enter your choice: 5

3. Circular Queue

AIM : Implement Circular Queue using Array. Perform following operations on it.

a) Insertion (Enqueue)
b) Deletion (Dequeue)
c) Display

Objectives :

1. To understand the concept of Circular Queue using Array as a data structure.

2. Applications of Circular Queue.

Theory :

1) Definition of Circular Queue –

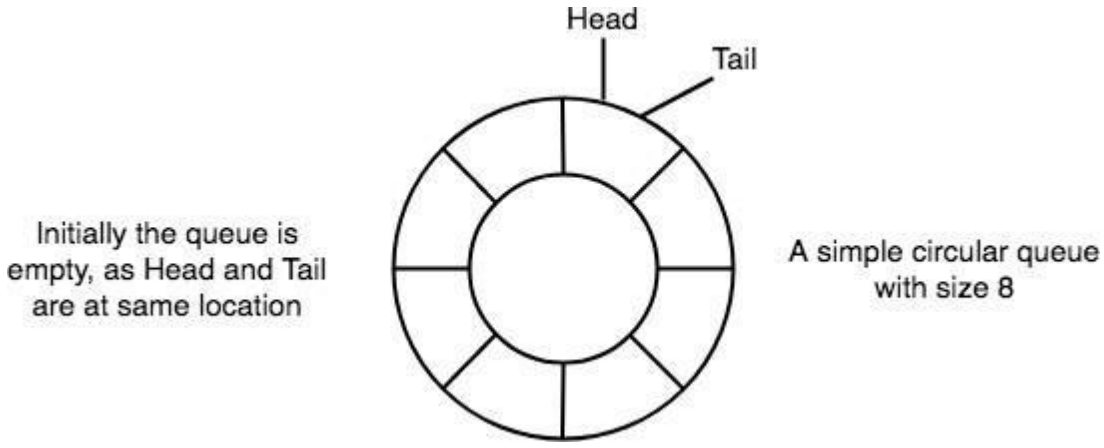
Circular Queue is a linear data structure, which follows the principle of **FIFO**(First In First Out), but instead of ending the queue at the last position, it again starts from the first position after the last, hence making the queue behave like a circular data structure.

2) Definition of Array –

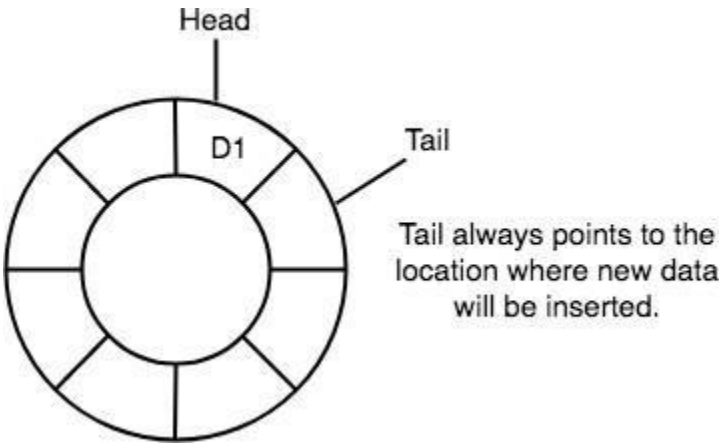
An **array**, is a data structure consisting of a collection of *elements* (values or variables), each identified by at least one *array index* or *key* .

Basic features of Circular Queue

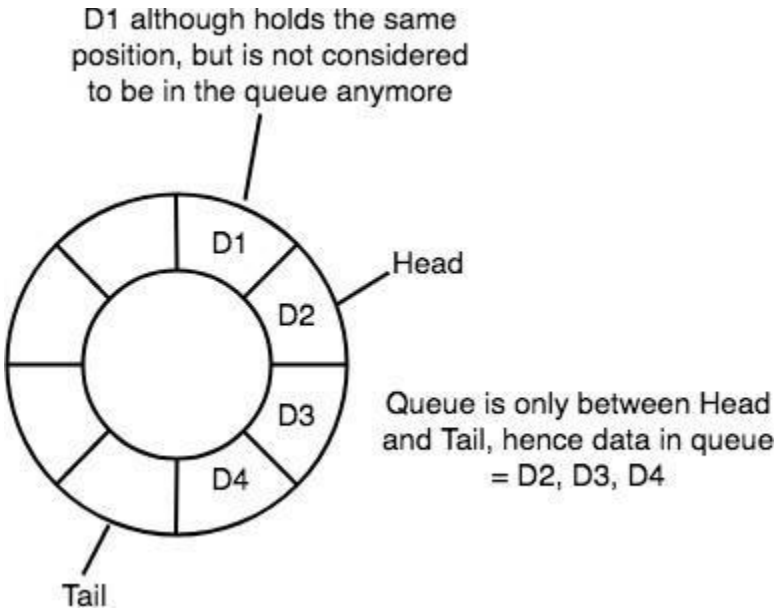
1. In case of a circular queue, `head` pointer will always point to the front of the queue, and `tail` pointer will always point to the end of the queue.
2. Initially, the head and the tail pointers will be pointing to the same location, this would mean that the queue is empty.



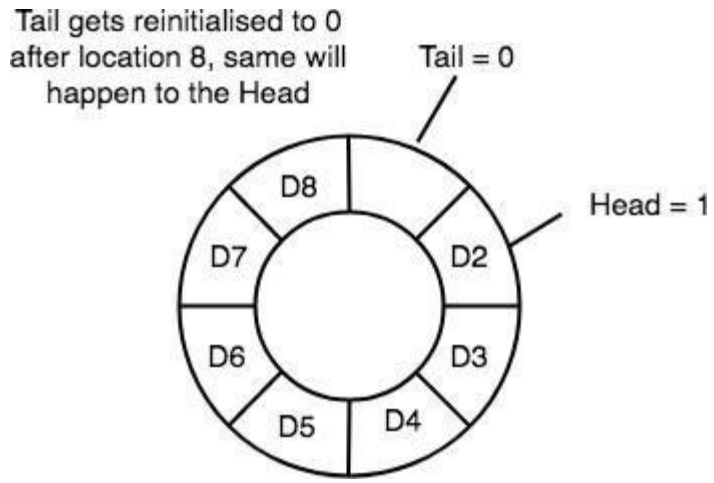
3. New data is always added to the location pointed by the `tail` pointer, and once the data is added, `tail` pointer is incremented to point to the next available location.



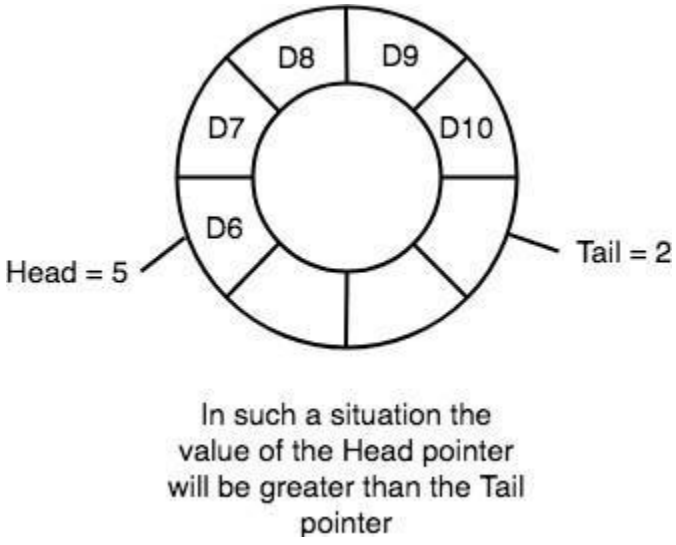
4. In a circular queue, data is not actually removed from the queue. Only the `head` pointer is incremented by one position when **dequeue** is executed. As the queue data is only the data between `head` and `tail`, hence the data left outside is not a part of the queue anymore, hence removed.



5. The `head` and the `tail` pointer will get reinitialised to **0** every time they reach the end of the queue.



6. Also, the `head` and the `tail` pointers can cross each other. In other words, `head` pointer can be greater than the `tail`. Sounds odd? This will happen when we dequeue the queue a couple of times and the `tail` pointer gets reinitialised upon reaching the end of the queue.



Insertion :

enQueue(value) This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at Rear position.

Steps:

1. Check whether queue is Full – Check ((rear == SIZE-1 && front == 0) || (rear == front-1)).

2. If it is full then display Queue is full. If queue is not full then, check if (rear == SIZE – 1 && front != 0) if it is true then set rear=0 and insert element.

Pseudo Code :

```
      void insertCQ(int val)

{ if ((front == 0 && rear == n - 1) || (front == rear + 1))
    { cout << "Queue Overflow \n";
        return;
    } if (front == -
    1)
    { front = 0;
        rear = 0;
    } else
    { if (rear == n - 1)
        rear = 0;
        else rear = rear +
            1;
    } cqueue[rear] =
    val;
}
```

Deletion :

deQueue() This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from front position.

Steps:

1. Check whether queue is Empty means check (front==-1).
2. If it is empty then display Queue is empty. If queue is not empty then step 3
3. Check if (front==rear) if it is true then set front=rear= -1 else check if (front==size-1), if it is true then set front=0 and return the element.

Pseudo Code :

```
      void deleteCQ()

{ if (front == -1)
    { cout << "Queue Underflow\n"; return; } cout << "Element deleted
    from queue is : " << cqueue[front] << endl; if (front == rear)
    { front = -1;
        rear = -1;
    } else
    { if (front == n - 1)
        front = 0;
        else front = front +
            1;
```

```
    }
}
```

Display Circular Queue :

We can use the following steps to display the elements of a circular queue...

- **Step 1 -** Check whether **queue** is **EMPTY**. (**front == -1**)
- **Step 2 -** If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.
- **Step 3 -** If it is **NOT EMPTY**, then define an integer variable '**i**' and set '**i** = **front**'.
- **Step 4 -** Check whether '**front <= rear**', if it is **TRUE**, then display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until '**i <= rear**' becomes **FALSE**.
- **Step 5 -** If '**front <= rear**' is **FALSE**, then display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until'**i <= SIZE - 1**' becomes **FALSE**.
- **Step 6 -** Set **i** to **0**.
- **Step 7 -** Again display '**cQueue[i]**' value and increment **i** value by one (**i++**). Repeat the same until '**i <= rear**' becomes **FALSE**.

Pseudo Code :

```
      void displayCQ_forward()


{ int f = front, r = rear;
    if (front == -1)
    { cout << "Queue is empty" << endl;
        return;
    } cout << "Queue elements are
    :\n"; if (f <= r)
    { while (f <= r)
        { cout << cqueue[f] << " ";
            f++;
        }
    } else
    { while (f <= n - 1)
        { cout << cqueue[f] << " ";
        f++; } f = 0; while (f <= r)
        { cout << cqueue[f] << " ";
            f++;
        } }
    cout << endl;
}
```

Conclusion:

Thus we had Implemented Circular Queue using Array and performed following operations :

1) Insertion (Enqueue)
2) Deletion (Dequeue)
3) Display

INPUT:

```
#include <iostream>
using namespace std;

int cqueue[5];
int front = -1 ,rear=-1,n=5;

void insert(int val){
    if((front == 0 && rear ==n-1) || (front == rear+1)){
        cout<<"\nQueue is filled "<<endl;
        return;
    }
    if(front== -1){
        front = 0;
        rear = 0;
    }
    else{
        if(rear==n-1){
            rear=0;
        }
        else{
            rear = rear+1;
        }

    }
    cqueue[rear] = val;
}

void deletion(){
    if(front == -1){
        cout<<"\nQueue is already empty "<<endl;
        return;
    }
```

```
    cout<<"Element deleted from queue is --> "<<cqueue[front]<<endl;

    if(front == rear){
        front = -1;
        rear = -1;
    }
    else{
        if(front == n-1){
            front = 0;
        }
        else{
            front = front +1;
        }
    }
}

void display_front(){
    int f= front ,r = rear;

    if(front == -1){
        cout<<"\nQueue is already empty "<<endl;
        return ;
    }
    cout<<"Queue elements in forward order -->"<<endl;

    if(f<=r){
        while(f<=r){
            cout<<cqueue[f]<<" ";
            f++;
        }
    }
    else{
        while(f<=n-1){
```

```
            cout<<cqueue[f]<<" ";
            f++;
        }
        f=0;
        while(f<=r){
            cout<<cqueue[f]<<" ";
            f++;
        }
    }
    cout<<endl;

}

void display_reverse(){
    int f= front ,r = rear;

    if(front == -1){
        cout<<"\nQueue is already empty "<<endl;
        return ;
    }
    cout<<"Queue elements in reverse order -->"<<endl;

    if(f<=r){
        while(f<=r){
            cout<<cqueue[r]<<" ";
            r--;
        }
    }
    else{
        while(r>=0){
            cout<<cqueue[r]<<" ";
            r--;
        }
```

```
        r=n-1;
        while(r>=f){
            cout<<cqueue[r]<<" ";
            r--;
        }
    }
    cout<<endl;

}


int main()
{
     int ch,val;
     cout<<"1]Insert"<<endl;
     cout<<"2)Delete"<<endl;
     cout<<"3)Display Forward"<<endl;
     cout<<"4)Display Reverse"<<endl;
     cout<<"5)Exit"<<endl;

   do {
      cout<<"Enter choice --> ";
      cin>>ch;
      switch(ch) {
         case 1:
         cout<<"Input for insertion--> ";
         cin>>val;
         cout<<endl;
         insert(val);
         break;

         case 2:
         deletion();
```

```
        cout<<endl;
        break;

        case 3:
        display_front();
        cout<<endl;
        break;

        case 4:
        display_reverse();
        cout<<endl;
        break;

        case 5:
        cout<<"Exit\n";
        break;

        default: cout<<"\nEnter correct choice !"<<endl;
      }
   } while(ch != 5);
   return 0;
}
```

OUTPUT:

```
1]Insert
2)Delete
3)Display Forward
4)Display Reverse
5)Exit
Enter choice --> 1
Input for insertion--> 10

Enter choice --> 1
Input for insertion--> 20

Enter choice --> 1
```

Input for insertion--> 30

Enter choice --> 1
Input for insertion--> 40

Enter choice --> 1
Input for insertion--> 50

Enter choice --> 1
Input for insertion--> 60

Queue is full you can't insert the element
Enter choice --> 2
Element deleted from queue is --> 10

Enter choice --> 2
Element deleted from queue is --> 20

Enter choice --> 2
Element deleted from queue is --> 30

Enter choice --> 2
Element deleted from queue is --> 40

Enter choice --> 2
Element deleted from queue is --> 50

Enter choice --> 2

Queue is already empty

Enter choice --> 1
Input for insertion--> 10

Enter choice --> 1
Input for insertion--> 20

Enter choice --> 1

Input for insertion--> 30

Enter choice --> 3
Queue elements in forward order -->
10 20 30

Enter choice --> 4
Queue elements in reverse order -->
30 20 10

Assignment 4 – Expression Tree and Tree Traversals

AIM : To construct an expression tree and perform recursive and non-recursive traversals.

DETAILED PROBLEM STATEMENT :

To construct an Expression Tree from postfix and prefix expression. Perform recursive and non- recursive in-order, pre-order and post-order traversals.

OBJECTIVE:

1. To understand non-linear data structure: Tree.
2. To implement non-linear data structure: Tree.
3. To understand applications of Tree.

OUTCOME:

1. To implement expression tree.
2. To perform recursive and non-recursive traversals.
3. To understand applications of tree

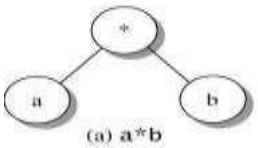
THEORY:

Definition of Expression Tree:
For an Algebraic expressions such as: 3 + (4 * 6) or a + b * c

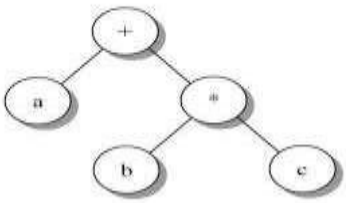
The terminal nodes (leaves) of an expression tree are the variables or constants in the expression (3,4, 6, or a, b, c). The non-terminal nodes of an expression tree are the operators (+, -, *, /).

Notice that the parentheses which appear in equation do not appear in the tree.

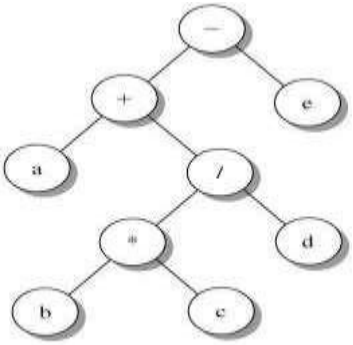
Examples of Expression Tree:



(a) a*b



(b) a + b* c



(c) a + b*c/d − e

Applications of Expression Tree:

1. Expression conversion i.e. to convert infix expression to postfix or prefix and vice a versa.
2. Evaluation of an infix, prefix and postfix expression.

Data Structures to Implement Expression Tree:

An arithmetic expression consists of operands (variable or constant) and operators. To construct an expression binary tree Stack is an appropriate data structure.
In construction of expression tree process two stack as an array are declared one as operand stack to store operand as node and other as operator stack to store operator as a node for an infix or prefix or postfix expression.

Different type of traversals with example:

To traverse a non-empty binary tree in **pre-order**:

1. Visit the root.
2. Traverse the left sub tree.
3. Traverse the right sub tree.

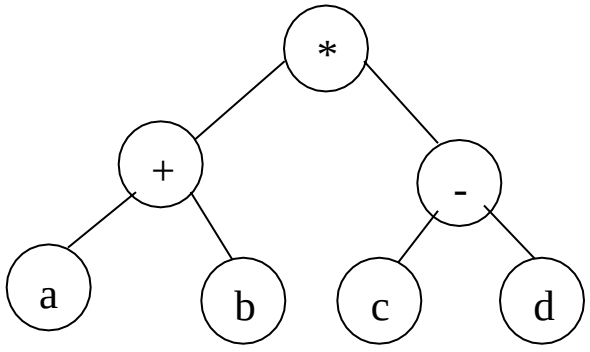
To traverse a non-empty binary tree in **in-order**:

1. Traverse the left sub tree.
2. Visit the root.
3. Traverse the right sub tree.

To traverse a non-empty binary tree in **post-order**,

1. Traverse the left sub tree.
2. Traverse the right sub tree.
3. Visit the root.

Consider an example of an expression tree:



1. Pre-order (Prefix) : * + a b – c d

2. In-order (Infix) : a + b * c - d

3. Post-order (postfix) : a b + c d - *

ADT of Expression Tree:

Structure: Binary_Tree ( ) is

Objects: a finite set of nodes either empty or consisting of a root node, left_Binary_Tree, and right_Binary_Tree.

Functions:
For all bt, bt1, bt2 ∈ Binary_Tree, item ∈ element

Binary_Tree Create( )::= creates an empty binary tree

Boolean IsEmpty(bt)::= if (bt==empty binary tree) return TRUE else return FALSE

Binary_Tree MakeBT(bt1, item, bt2)::= return a binary tree whose left sub tree is bt1, whose right sub tree is bt2, and whose root node contains the data item

Binary_Tree Lchild(bt)::= if (IsEmpty(bt)) return error else return the left sub tree of bt

Element Data(bt)::= if (IsEmpty(bt)) return error else return the data in the root node of bt

Binary_Tree Rchild(bt)::= if (IsEmpty(bt)) return error else return the right sub tree of bt

END

ALGORITHMS/PESUDOCODE :

- **Create Expression Tree:**

For constructing expression tree we use a stack. We loop through input expression and do following for every character.

Create_ET(input expression)

//For constructing expression tree we use a stack.
//We loop through input expression and do following for every character.
Step 1: Read an infix expression in an array and convert it to postfix/prefix
Step 2: While not of postfix expression

i. If character is operand push that into stack

ii. If character is operator

a. New BNode

b. BNode-> data = operator assign data part of the node

c. BNode- >Rchild =pop stack top // assign right child first top symbol of the stack

d. BNode ->Lchild = pop stack top2 // assign left child as second top symbol of the stack

e.. push current node again.

Step 3: end of while
Step 4: root of the tree = pop stack
Step 5: End of Create_ET

- **In-order Traversal (Recursive)**

1. Traverse the left sub tree, i.e. call In-order (left – sub-tree)
2. Visit the root.
3. Traverse the right sub tree, i.e. call In-order (right – sub-tree)

- **Pre-order Traversal (Recursive)**

1. Visit the root.
2. Traverse the left sub tree, i.e. call Pre-order (left – sub-tree)
3. Traverse the right sub tree, i.e. call Pre-order (right – sub-tree)

- **Post-order Traversal (Recursive)**

1. Traverse the left sub tree, i.e. call Post-order (left – sub-tree)
2. Traverse the right sub tree, i.e. call Post-order (right – sub-tree)
3. Visit the root.

- **In-order Traversal (Non- recursive)**

1. Create an empty stack S.
2. Initialize current node as root
3. Push the current node to S and set current = current->left until current is NULL
4. If current is NULL and stack is not empty then
    a. Pop the top item from stack.
    b. Print the popped item, set current = poped_item->right
    c. Go to step 3.
5. If current is NULL and stack is empty then stop.

- **Pre-order Traversal (Non – recursive)**

1 Create an empty stack Sand push root node to stack.

2. Do following while S is not empty.

a. Pop an item from stack and print it.

b. Push right child of popped item to stack

c. Push left child of popped item to stack

3. If stack is empty, then stop.

- **Post-order Traversal (Non – recursive)**

1. Create an empty stack S
2. Do following while root is not NULL
   a. Push root's right child and then root to stack.
   b. Set root as root's left child.
3. Pop an item from stack and set it as root.
   a. If the popped item has a right child and the right child is at top of stack, then remove the right child from stack, push the root back and set root as root's right child.
   b. Else print root's data and set root as NULL.
4. Repeat steps 3.a and 3.b while stack is not empty, then stop.

Test Cases / Validation:

Validations:

1. Identification of symbol as operator or operand.
2. Postfix and prefix expressions can be of char or int data type

Test Cases

1. Test for valid prefix or postfix expression
2. no of operator and operand relationship check

Sr. No.	Sample Infix Expression	Postfix	Prefix
1.	A+B*C	ABC*+	+A*CB
2.	A*B-C	AB*C-	-*ACB

3.	A^B-C	AB^C-	-^ABC
4.	A+B*C^E	ABCE^*+	+A*B^CE
5.	A-B*C+A	ABC*-A+	-+A*BCA
6.	(A+B)/(C+D)^E^F-D*F-D	AB+CD+EF^^/DF*-D-	/+AB+CD^EF*DFD
7.	A+B+C	AB+C+	++ABC
8.	A*B/C	AB*C/	/*ACB
9.	A^B^C	ABC^^	^A^BC

CONCLUSION:

Thus we have studied the concept of tree building from postfix and prefix expression.

INPUT:

```
#include <iostream>
using namespace std;
typedef struct node              //structure defined for node
{
   char data;
   struct node *left;
   struct node *right;
} node;

typedef struct stacknode              //structure defined for stack
{
   node *data;
   struct stacknode *next;
} stacknode;

class stack
{
   stacknode *top;                    //top node is introduced
public:
   stack()
   {
      top = NULL;
   }
   node *topp()                 //it will return the top element
   {
      return (top->data);
   }
```

```
    int isempty()                    //check if stack is empty(1)
    {
       if (top == NULL)
          return 1;
       return 0;
    }
    void push(node *a)               //push function
    {
       stacknode *p;
       p = new stacknode();
       p->data = a;
       p->next = top;
       top = p;
    }
    node *pop()                       //pop function
    {
       stacknode *p;
       node *x;
       x = top->data;
       p = top;
       top = top->next;
       return x;
    }
};

node *create_pre(char prefix[10]);
node *create_post(char postfix[10]);
void inorder(node *p);
void preorder(node *p);
void postorder(node *p);
void inorder_non_recursive(node *t);
void preorder_non_recursive(node *t);
void postorder_non_recursive(node *t);

node *create_post(char postfix[10])
{
    node *p;
    stack s;
    for (int i = 0; postfix[i] != '\0'; i++)
    {
       char token = postfix[i];              //token is the element in postfix
```

```
        if (isalnum(token))                    //check if token is alphanumeric (operand)
        {
            p = new node();                         //node creation
            p->data = token;
            p->left = NULL;
            p->right = NULL;
            s.push(p);
        }
        else //operator
        {
            p = new node();
            p->data = token;
            p->right = s.pop();
            p->left = s.pop();
            s.push(p);
        }
    }
    return s.pop();
}

node *create_pre(char prefix[10])
{
    node *p;
    stack s;
    int i;
    for (i = 0; prefix[i] != '\0'; i++)
    {
    }
    i = i - 1;
    for (; i >= 0; i--)
    {
        char token = prefix[i];            // prefix element
        if (isalnum(token))                // operand
        {                               //node creation
            p = new node();
            p->data = token;
            p->left = NULL;
            p->right = NULL;
            s.push(p);
        }
        else //operator
```

```
        {
            p = new node();
            p->data = token;
            p->left = s.pop();
            p->right = s.pop();
            s.push(p);
        }
    }
    return s.pop();
}
void inorder(node *p)                    //inorder traversal using recursion
{
    if (p == NULL)
    {
        return;
    }
    inorder(p->left);
    cout << p->data;
    inorder(p->right);
}
void preorder(node *p)                   //preorder traversal using recursion
{
    if (p == NULL)
    {
        return;
    }
    cout << p->data;
    preorder(p->left);
    preorder(p->right);
}
void postorder(node *p)                  //postorder traversal using recursion
{
    if (p == NULL)
    {
        return;
    }
    postorder(p->left);
    postorder(p->right);
    cout << p->data;
}
```

```
int main()
{
   node *r = NULL, *r1;
   char postfix[10], prefix[10];
   int x;
   int ch, choice;
   do
   {
      cout << "\n\t*********MENU**********\n\n1.Construct tree from postfix
    Expression/prefix Expression.\n2.Inorder traversal.\n3.Preorder traversal.\n4.Postorder
    Traversal.\n5.Exit\n\nEnter your choice: ";
      cin >> ch;
      switch (ch)
      {
      case 1:
         cout << "\nENTER CHOICE:\n\t1.Postfix expression\n\t2.Prefix
    expression\nChoice= ";
         cin >> choice;

         if (choice == 1)
         {
            cout << "\nEnter postfix expression= ";
            cin >> postfix;
            r = create_post(postfix);
         }
         else
         {
            cout << "\nEnter prefix expression= ";
            cin >> prefix;
            r = create_pre(prefix);
         }
         cout << "\n** Tree created successfully ** \n";
         break;
      case 2:
         cout << "\n**********************" << endl;
         cout << "\nInorder Traversal of tree\n\n";
         cout << "With recursion:\t";
         inorder(r);
         cout << "\n\nWithout recursion: ";
         inorder_non_recursive(r);
         cout << "\n\n**********************" << endl;
         break;
```

```
        case 3:
            cout << "**********************" << endl;
            cout << "\nPreorder Traversal of tree\n\n";
            cout << "With recursion:\t";
            preorder(r);
            cout << "\n\nWithout recursion: ";
            preorder_non_recursive(r);
            cout << "\n\n**********************" << endl;
            break;
        case 4:
            cout << "**********************" << endl;
            cout << "\nPostorder Traversal of tree\n\n";
            cout << "With recursion:\t";
            postorder(r);
            cout << "\n\nWithout recursion: ";
            postorder_non_recursive(r);
            cout << "\n\n**********************" << endl;
            break;
        }
    } while (ch != 5);
    return 0;
}

void inorder_non_recursive(node *t)
{
    stack s;
    while (t != NULL)
    {                       //data pushed in stack and moved to left till null(last)
        s.push(t);
        t = t->left;
    }
    while (s.isempty() != 1)
    {
        t = s.pop();              // topmost data of stack is printed and then moved to the right
        cout << t->data;
        t = t->right;
        while (t != NULL)
        {                         //if child is represent push it to the stack
            s.push(t);
            t = t->left;
        }
```

```
    }
}

void preorder_non_recursive(node *t)
{
    stack s; //stack
    while (t != NULL)
    {                          //it will start from the root and then move to left
        cout << t->data;
        s.push(t);
        t = t->left;
    }                          //once left side is traversed we will pop and move to right
    while (s.isempty() != 1)
    {
        t = s.pop();
        t = t->right;
        while (t != NULL)
        {                          //if child is represent we will push in stack
            cout << t->data;
            s.push(t);
            t = t->left;
        }
    }
}

void postorder_non_recursive(node *t)
{
    stack s, s1;                   //two stack maintained
    node *t1;                      //root
    while (t != NULL)
    {
        s.push(t);
        s1.push(NULL);
        t = t->left;
    }
    while (s.isempty() != 1)
    {
        t = s.pop();
        t1 = s1.pop();
        if (t1 == NULL)
        {
```

```
            s.push(t);
            s1.push((node *)1);
            t = t->right;
            while (t != NULL)
            {
               s.push(t);
               s1.push(NULL);
               t = t->left;
            }
         }
         else
            cout << t->data;
      }
       }
```

OUTPUT:

```
         **********MENU**********

1.Construct tree from postfix Expression/prefix Expression.
2.Inorder traversal.
3.Preorder traversal.
4.Postorder Traversal.
5.Exit

Enter your choice: 1

ENTER CHOICE:
         1.Postfix expression
         2.Prefix expression
Choice= 2

Enter prefix expression= *+23+45

** Tree created successfully **

         **********MENU**********
```

1.Construct tree from postfix Expression/prefix Expression.
2.Inorder traversal.
3.Preorder traversal.
4.Postorder Traversal.
5.Exit

Enter your choice: 2

****

Inorder Traversal of tree

With recursion: 2+3*4+5

Without recursion: 2+3*4+5

****

**********MENU**********

1.Construct tree from postfix Expression/prefix Expression.
2.Inorder traversal.
3.Preorder traversal.
4.Postorder Traversal.
5.Exit

Enter your choice: 3
****

Preorder Traversal of tree

With recursion: *+23+45

Without recursion: *+23+45

****

**********MENU**********

1.Construct tree from postfix Expression/prefix Expression.
2.Inorder traversal.

3.Preorder traversal.
4.Postorder Traversal.
5.Exit

Enter your choice: 4
****

Postorder Traversal of tree

With recursion: 23+45+*

Without recursion: 23+45+*

****

**********MENU**********

1.Construct tree from postfix Expression/prefix Expression.
2.Inorder traversal.
3.Preorder traversal.
4.Postorder Traversal.
5.Exit

Enter your choice: 5
PS D:\C++ GB\Assignments>

Assignment 5 - Binary search tree

AIM : **Implementation of binary search tree**

DETAILED PROBLEM STATEMENT :

Implement binary search tree and Perform following operations:
a) Insert,
b) delete,
c) search
d) display tree (traversal)
e) display – depth of tree
f) display - mirror image
e) create a copy
f) Display all parent nodes with their child nodes
g) display tree level wise
h) display leaf nodes.
(Note: Insertion, Deletion, Search and Traversal are compulsory, from rest of operations, perform Any three)

OBJECTIVE:

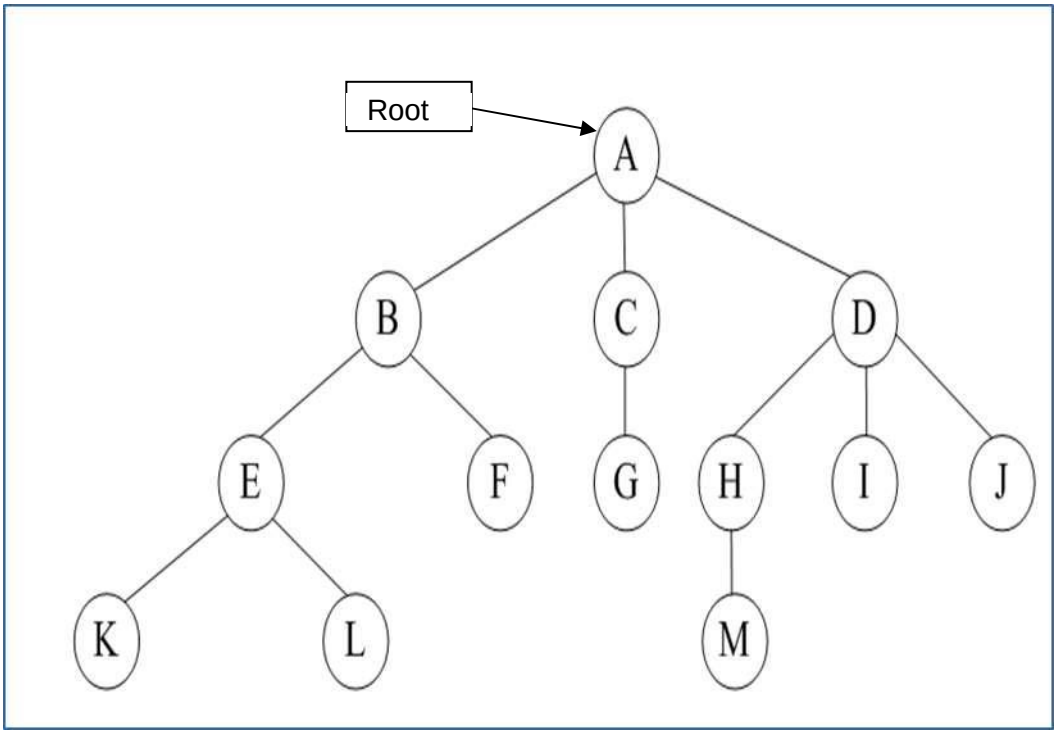
1. To understand difference between Binary Tree and Binary Search Tree.
2. To implement Binary Search Tree
3. To understand applications of Binary Search Tree

OUTCOME :

1. To construct binary search tree.
2. To perform different operations on it.
3. To do traversals on a tree.

THEORY:

Definition of binary search tree
A binary tree in which each internal node x stores an element such that the element 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. This is called binary-search-tree property.



Binary Search Tree :

- **Concept:**

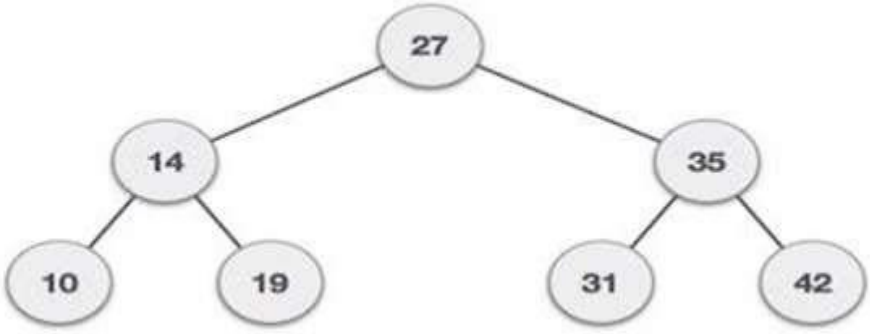
A Binary Search Tree is a type of binary tree data structure in which the nodes are arranged in order, hence also called as "ordered binary tree". It is a node-based data structure, which provides an efficient and fast way of sorting, retrieving, searching data.

- **Definition:**

Binary search tree is a node-based binary tree data structure which has the following properties:

1. The left subtree of a node contains only nodes with keys lesser than the node's key.
2. The right subtree of a node contains only nodes with keys greater than the node's key.
3. The left and right subtree each must also be a binary search tree.

- Type of data structure: It is a non-linear type of data structure.



- ADT:

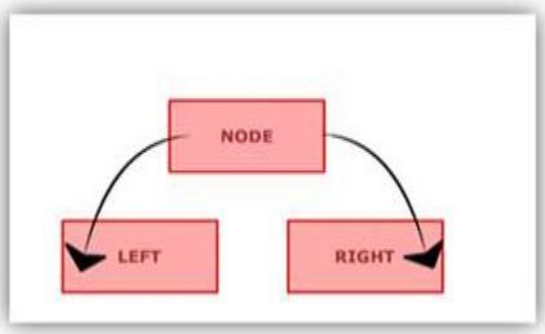
It is a special kind of binary tree which performs following operations:

1. Insert (X)::= depending on the root element X get inserted either to left or to right and new BST

will be formed.

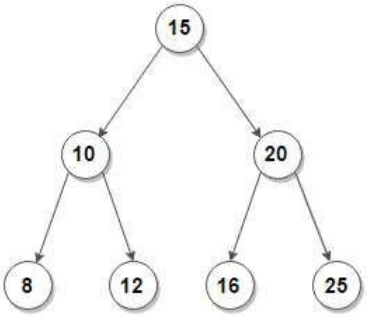
2. Search (X)::= the element is searched either to the left or to the right half of the tree.
3. Delete (X)::= the element is deleted and new BST will be formed.
4. Traversal (root)::= it will return all traversals of BST.

- Realization of ADT:



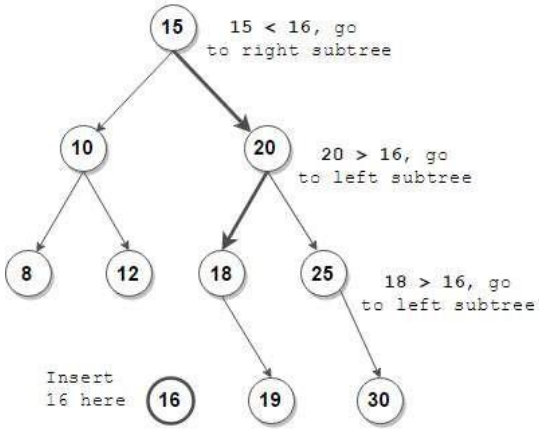
Data that represents value stored in the node.
Left that represents the pointer to the left child.
Right that represents the pointer to the right child.

- Example with basic operations:



Binary Search Tree

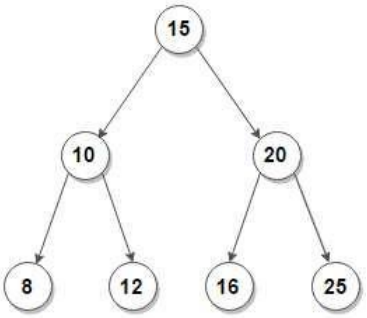
Insert: insert 16 in the above BST



Insert (root, 16)

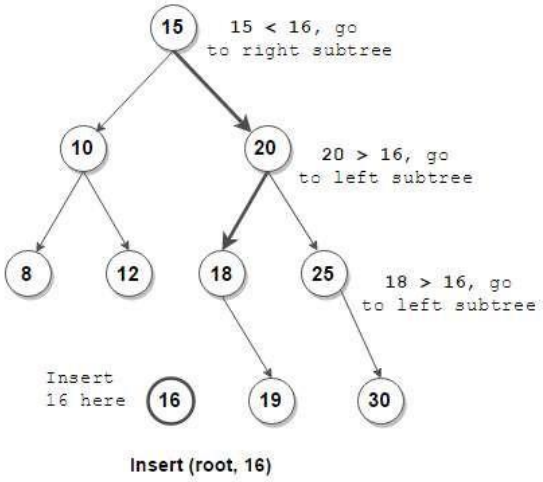
Right that represents the pointer to the right child.

- **Example with basic operations:**



Binary Search Tree

Insert: insert 16 in the above BST



Insert (root, 16)

Traversals:

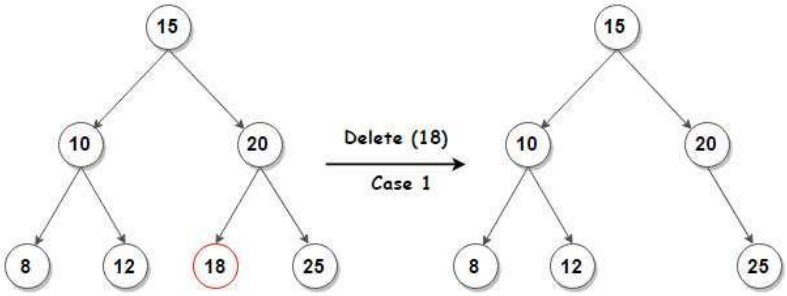
Inorder Traversal: 8 10 12 15 16 18 19 20 25 26

Preorder Traversal: 15 10 8 12 20 18 16 19 25 30

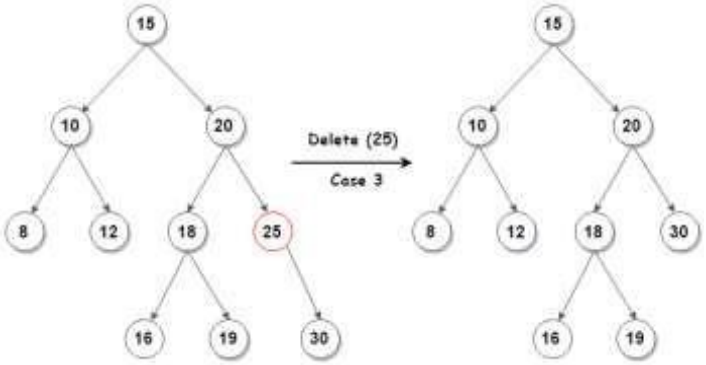
Postorder Traversal: 8 12 10 16 19 18 30 25 20 15

Delete:

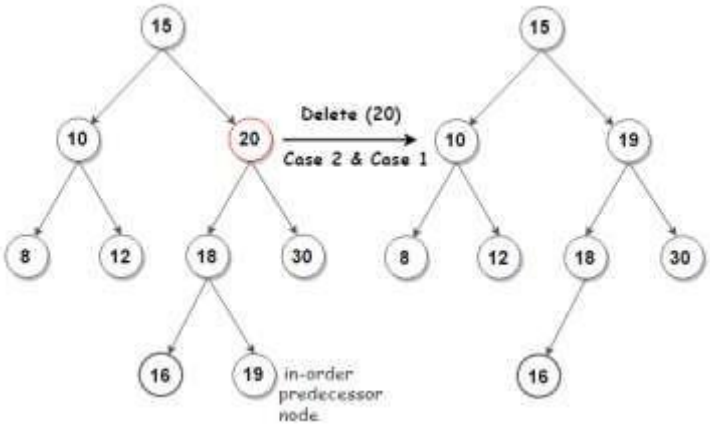
1) **Node to be deleted is leaf:** Simply remove from the tree.



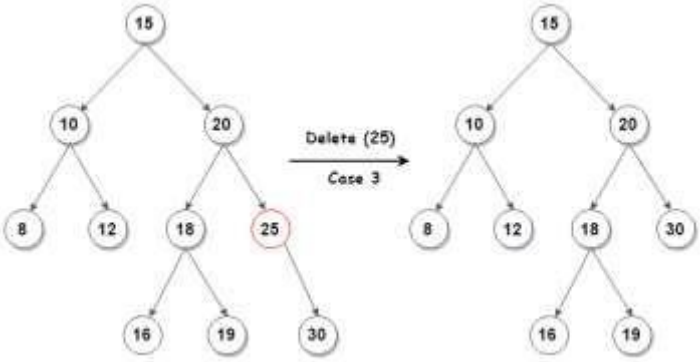
2) **Node to be deleted has only one child:** Copy the child to the node and delete the child



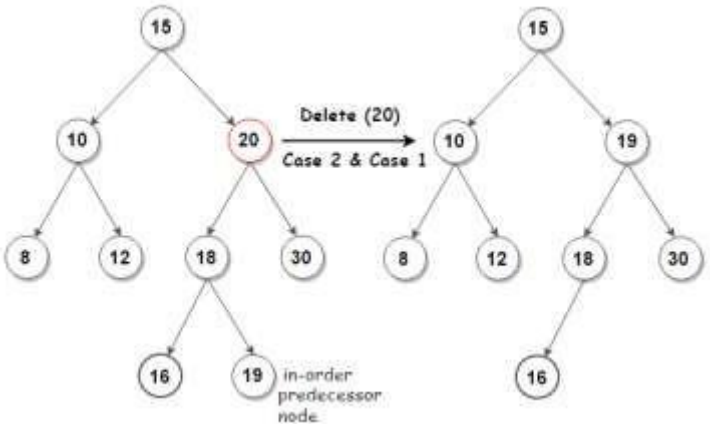
3) **Node to be deleted has two children:** Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.



4) **Node to be deleted has only one child:** Copy the child to the node and delete the child



5) **Node to be deleted has two children:** Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.



- **Insert Node:**

Insert(Root, Node)

Root is a variable of type structure represent Root of the Tree. Node is a variable of type structure represent new Node to insert in a tree.
Step 1: Repeat Steps 2, 3 & 4 Until Node do not insert at appropriate position.
Step 2: If Node Data is Less that Root Data & Root Left Tree is NULL
Then insert Node to Left.
Else Move Root to Left
Step 3 : Else If Node Data is Greater that Equal that Root Data & Root Right Tree is NULL
Then insert Node to Right.
Else Move Root to Right.
Step 4: Stop.

- **Search Node:**

Search (Root, Item)
Root is a variable of type structure represent Root of the Tree. Item is the element to search. This function search an element in a Tree.
Step 1: Repeat Steps 2,3 & 4 Until element Not find && Root != NULL
Step 2: If item Equal to Root Data
Then print message item present.
Step 3 : Else If item Greater that Equal that Root Data
Then Move Root to Right.
Step 4 : Else Move Root to Left.
Step 5: Stop.

- **Delete Node:**

Dsearch(Root, Item)

Root is a variable of type structure represent Root of the Tree. Item is the element to delete. Stack is a pointer array of type structure. PTree(Parent of Searched Node),Tree(Node to be deleted), RTree(Pointg to Right Tree),Temp are pointer variable of type structure;

Step 1: Search an Item in a Binary Search Tree
Step 2: If Root = = NULL Then Tree is NULL
Step 3: Else //Delete Leaf Node
If Tree->Left = = NULL && Tree->Right = = NULL
Then a) If Root = = Tree Then Root = NULL;
b) If Tree is a Right Child PTree->Right=NULL;
Else PTree->Left=NULL;
Step 4: Else // delete Node with Left and Right children
If Tree->Left != NULL && Tree->Right != NULL
Then a) RTree=Temp=Tree->Right;
b) Do steps i && ii while Temp->Left !=NULL
i) RTree=Temp;
ii) Temp=Temp->Left;
c) RTree->Left=Temp->Right;
d) If Root == Tree//Delete Root Node
Root=Temp;
e) If Tree is a Right Child PTree->Right=Temp;
Else PTree->Left=Temp;
f) Temp->Left=Tree->Left;
g) If RTree!=Temp
Then Temp->Right = Tree->Right;
Step 5: Else //with Right child
If Tree->Right!= NULL
Then a) If Root==Tree Root = Root->Right;
b) If Tree is a Right Child PTree->Right=Tree->Right;
Else PTree->Left=Tree->Left;
Step 6: Else //with Left child
If Tree->Left != NULL
Then a) If Root==Tree Root = Root->Left;
b) If Tree is a Right Child PTree->Right=Tree->Left;
Else PTree->Left=Tree->Left;
Step 7: Stop.

- **Inorder Traversal Recursive :**

Tree is pointer of type structure.

InorderR(Tree)
Step 1: If Tree != NULL
Step 2: InorderR(Tree->Left);
Step 3: Print Tree->Data
Step 4: InorderR(Tree->Right);

- **Postorder Traversal Recursive:**

Tree is pointer of type structure.

PostorderR(Tree)
Step 1: If Tree != NULL
Step 2: PostorderR(Tree->Left);
Step 3: PostorderR(Tree->Right);
Step 4: Print Tree->Data;

- **Preorder Traversal Recursive:**

Tree is pointer of type structure.

PreorderR(Tree)
Step 1: If Tree != NULL
Step 2: Print Tree->Data;
Step 3: PreorderR(Tree->Left);
Step 4: PreorderR(Tree->Right);

- **Mirror of thetree Recursive**

ptr Mirror_BST(BST root)

Step 1: If treet != NULL
Step 2: temp =tree->Right
tree-> Right= tree->Left
tree->Left=temp
Step 3: Mirror_BST(tree->Left)
Step 4: Mirror_BST(tree->Right)

- **Height of the tree recursive :**

int height (root)
Step1 : If (root=NULL)
Display " tree is empty"
stop
Step 2: Else
Step 3: return 1 + max(height(root->left),height(root->right))

- **Level Order Traversal of a tree :**

levelorder(root)

```
//  q is QUEUE of size(1:n)
Step 1: If(T=NULL ) //empty tree
              then  write('Tree is Emty)
               return
Step 2: else          //create empty Queue
Step 3:         q = empty queue
Step 4:         enqueue(T)      //Intially root

Step 5:    while(!(isempty(q))
Step 6:    node = dequeue(q)  //front
Step 7:    write /visit (node)
Step  8:              if (node->left != NULL )
           //insert left node next level
Step 9:                enqueue(node->left)
           //insert right node of next levle
Step 10:              if (node->right != NULL)
              enqueue(node->right)
Step 11:nd Level Order
```

Validations :

1. No duplicate key is allowed in binary search tree while insertion.

Test Cases

1. Try to insert duplicate key: Message should be display not allowed.
2. In mirror image, inorder traversal should be in descending order.
3. Try to delete leaf node: it should return proper binary search tree.
4. Try to delete a node with one child: it should return proper binary search tree.
5. Try to delete a node with two child: it should return proper binary search tree.

FAQS:

1. What is Binary search tree?
2. What are the members of structure of tree & what is the size of structure?
3. What are rules to construct binary search tree?
4. How general tree is converted into binary tree?
5. What is binary threaded tree?
6. What is use of thread in traversal?

CONCLUSION:

Hence we have studied the concept of Binary Search Tree.

INPUT:

```
#include <iostream>
#include<stdlib.h>
using namespace std;
struct node
{
   int data;
   struct node *left;
   struct node *right;
};
node *insert(node *root, int val){
   if (root == NULL){
      node *temp;            //new node temp
      temp=new node;
      temp->data=val;
      temp->left=temp->right=NULL;   //left and right is NULL bcz only one
node create
      return temp;               // return single node
   }
   if (val < root->data){
      root->left = insert(root->left, val);
   }
   else{
      //val>root->data
      root->right = insert(root->right, val);
   }
   return root;
}
void inorder(node *root)
{
   if (root == NULL){
      return;
   }
   inorder(root->left);
   cout << root->data << " ";
   inorder(root->right);
```

```
}
node* inorderSucc(node* root){
   node* curr=root;
   while(curr && curr->left!=NULL){
      curr=curr->left;
   }
   return curr;
}
node *delet(node *root, int key){
   if(key<root->data){
      root->left=delet(root->left, key);
   }
   else if(key>root->data){
      root->right=delet(root->right,key);
   }
   //if key==root->data
   else{
      if(root->left==NULL){
         node* temp=root->right;
         free(root);
         return temp;
      }
      else if(root->right==NULL){
         node* temp=root->left;
         free(root);
         return temp;
      }
      node* temp=inorderSucc(root->right);
      root->data=temp->data;
      root->right=delet(root->right, temp->data);
   }
   return root;
}
node *search(node* root, int val){
   if(root==NULL){
      return NULL;
   }
```

```
    if(val>root->data){
        return search(root->right, val);
    }
    else if(val<root->data){
        return search(root->left, val);
    }
    else{
        return root;
    }
}
void mirrorImg(node* root){
    if(root==NULL){
        return;
    }
    else{
        struct node *temp;
        mirrorImg(root->left);
        mirrorImg(root->right);
        swap(root->left,root->right);
    }
}
node *copy(node *root){
    node *temp=NULL;
    if(root!=NULL){
        temp=new node();
        temp->data=root->data;
        temp->left=copy(root->left);
        temp->right=copy(root->right);
    }
    return temp;
}
void leafNodes(node* root){
    if(root==NULL){
        return;
    }
    if(!root->left && !root->right){
        cout<<root->data<<" ";
```

```
        return;
    }
    if(root->right)
      leafNodes(root->right);
    if(root->left)
      leafNodes(root->left);
}
int calHeight(node* root){
    if(root==NULL){
        return 0;
    }
    int lheight=calHeight(root->left);
    int rheight=calHeight(root->right);
    return max(lheight,rheight)+1;
}
node *findMin(node *root){
    if(root==NULL){
        return NULL;
    }
    if(root->left)
        return findMin(root->left);
    else
        return root;
}
node *findMax(node *root){
    if(root==NULL){
        return NULL;
    }
    if(root->right)
      return findMax(root->right);
    else
        return root;
}
int main()
{
    node *root=NULL, *temp;              //initially tree is NULL
    int ch;
```

```
while (1){
    cout<<"\n\n\t1)Insert" << endl;
    cout<<"\t2)Delete" << endl;
    cout<<"\t3)Search" << endl;
    cout<<"\t4)Create the copy "<<endl;
    cout<<"\t5)Display leaf nodes "<<endl;
    cout<<"\t6)Height of the tree"<<endl;
    cout<<"\t7)Find the minimum"<<endl;
    cout<<"\t8)Find the maximum"<<endl;
    cout<<"\t9)Mirror image"<<endl;
    cout<<"\t10)Exit"<<endl;
    cout<<"\nEnter your choice: ";
    cin>>ch;
    switch (ch){
    case 1:
        cout << "Enter the element to be insert:  ";
        cin >> ch;
        root= insert(root, ch);
        cout << "*****Elements in BST are*****:  ";
        inorder(root);
        break;
    case 2:
        cout<<"Enter the element to be deleted:  ";
        cin>>ch;
        root=delet(root, ch);
        cout<<"Element deleted successfully !!";
        cout<<"\n*****After deletion the elements in the BST are*****:  ";
        inorder(root);
        break;
    case 3:
        cout<<"Enter the element to be searched:   ";
        cin>>ch;
        temp=search(root, ch);
        if(temp==NULL){
            cout<<"*****Element is not found*****";
        }
        else{
```

```
            cout<<"*****Element is found*****";
        }
        break;
    case 4:
        cout<<"The copy of the tree is:   ";
        root=copy(root);
        inorder(root);
        break;
    case 5:
        cout<<"The leaf nodes are:  ";
        leafNodes(root);
        break;
    case 6:
        cout<<"Height of the binary search tree is:  "<<calHeight(root);
        break;
    case 7:
        temp=findMin(root);
        cout<<"\nMinimum element is :  "<<temp->data;
        break;
    case 8:
        temp=findMax(root);
        cout<<"\nMaximum element is:   "<<temp->data;
        break;
    case 9:
        cout<<" inorder tree:  ";
        inorder(root);
        cout<<endl;
        mirrorImg(root);
        cout<<"mirror image is:  ";
        inorder(root);
        break;
    case 10:
        return 0;
    default:
        cout<<"\nInvalid choice !! Please enter your choice again";
    }
}
```

```
    return 0;
}
```

OUTPUT:

```
      1)Insert
       2)Delete
       3)Search
       4)Create the copy
       5)Display leaf nodes
       6)Height of the tree
       7)Find the minimum
       8)Find the maximum
       9)Mirror image
       10)Exit

Enter your choice: 1
Enter the element to be insert:  2
*****Elements in BST are*****:  2

       1)Insert
       2)Delete
       3)Search
       4)Create the copy
       5)Display leaf nodes
       6)Height of the tree
       7)Find the minimum
       8)Find the maximum
       9)Mirror image
       10)Exit

Enter your choice: 1
Enter the element to be insert:  6
*****Elements in BST are*****:  2 6

       1)Insert
```

2)Delete
3)Search
4)Create the copy
5)Display leaf nodes
6)Height of the tree
7)Find the minimum
8)Find the maximum
9)Mirror image
10)Exit

Enter your choice: 1
Enter the element to be insert: 3
*****Elements in BST are*****: 2 3 6

1)Insert
2)Delete
3)Search
4)Create the copy
5)Display leaf nodes
6)Height of the tree
7)Find the minimum
8)Find the maximum
9)Mirror image
10)Exit

Enter your choice: 1
Enter the element to be insert: 7
*****Elements in BST are*****: 2 3 6 7

1)Insert
2)Delete
3)Search
4)Create the copy
5)Display leaf nodes
6)Height of the tree
7)Find the minimum
8)Find the maximum

```
        9)Mirror image
        10)Exit

Enter your choice: 1
Enter the element to be insert:  8
*****Elements in BST are*****:  2 3 6 7 8

        1)Insert
        2)Delete
        3)Search
        4)Create the copy
        5)Display leaf nodes
        6)Height of the tree
        7)Find the minimum
        8)Find the maximum
        9)Mirror image
        10)Exit

Enter your choice: 1
Enter the element to be insert:  4
*****Elements in BST are*****:  2 3 4 6 7 8

        1)Insert
        2)Delete
        3)Search
        4)Create the copy
        5)Display leaf nodes
        6)Height of the tree
        7)Find the minimum
        8)Find the maximum
        9)Mirror image
        10)Exit

Enter your choice: 2
Enter the element to be deleted:  2
Element deleted successfully !!
*****After deletion the elements in the BST are*****:  3 4 6 7 8
```

1)Insert
2)Delete
3)Search
4)Create the copy
5)Display leaf nodes
6)Height of the tree
7)Find the minimum
8)Find the maximum
9)Mirror image
10)Exit

Enter your choice: 3
Enter the element to be searched: 4
*****Element is found*****

1)Insert
2)Delete
3)Search
4)Create the copy
5)Display leaf nodes
6)Height of the tree
7)Find the minimum
8)Find the maximum
9)Mirror image
10)Exit

Enter your choice: 4
The copy of the tree is: 3 4 6 7 8

1)Insert
2)Delete
3)Search
4)Create the copy
5)Display leaf nodes
6)Height of the tree
7)Find the minimum

8)Find the maximum
9)Mirror image
10)Exit

Enter your choice: 4
The copy of the tree is: 3 4 6 7 8

1)Insert
2)Delete
3)Search
4)Create the copy
5)Display leaf nodes
6)Height of the tree
7)Find the minimum
8)Find the maximum
9)Mirror image
10)Exit

Enter your choice: 5
The leaf nodes are: 8 4

1)Insert
2)Delete
3)Search
4)Create the copy
5)Display leaf nodes
6)Height of the tree
7)Find the minimum
8)Find the maximum
9)Mirror image
10)Exit

Enter your choice: 6
Height of the binary search tree is: 3

1)Insert
2)Delete

3)Search
4)Create the copy
5)Display leaf nodes
6)Height of the tree
7)Find the minimum
8)Find the maximum
9)Mirror image
10)Exit

Enter your choice: 7

Minimum element is : 3

1)Insert
2)Delete
3)Search
4)Create the copy
5)Display leaf nodes
6)Height of the tree
7)Find the minimum
8)Find the maximum
9)Mirror image
10)Exit

Enter your choice: 8

Maximum element is: 8

1)Insert
2)Delete
3)Search
4)Create the copy
5)Display leaf nodes
6)Height of the tree
7)Find the minimum
8)Find the maximum
9)Mirror image

10)Exit

Enter your choice: 9
inorder tree: 3 4 6 7 8
mirror image is: 8 7 6 4 3

1)Insert
2)Delete
3)Search
4)Create the copy
5)Display leaf nodes
6)Height of the tree
7)Find the minimum
8)Find the maximum
9)Mirror image
10)Exit

Enter your choice: 10

Assignment 6- Threaded Binary Tree

AIM : Implementation In-order Threaded Binary Tree (TBT)

DETAILED PROBLEM STATEMENT :

To implement In-order TBT and to perform In-order, and Pre-order traversals.

OBJECTIVE :

1. To understand construction of TBT.
2. To implement In-order TBT.
3. To understand In-order, and Pre-order traversals of TBT.
4. To understand pros/cons of TBT over Binary Tress.
5.

OUTECOME :

1. To implement In-order TBT
2. To Perform In-Order and Preorder Traversal of TBT

THEORY :

Issues with regular Binary Tree Traversals:

1. The storage space required for stack and queue is large.
2. The majority of pointers in any binary tree are NULL. For example, a binary tree with n nodes has n+1 NULL pointers and these were wasted.
3. It is difficult to find successor node (Pre-order, In-order and Post-order successors) for a given node.

Motivation for Threaded Binary Trees:

To solve these problems, one idea is to store some useful information in NULL pointers. In the traversals of regular binary tree, stack/queue is required to record the current position in order to move to right subtree after processing the left subtree. If the null links are replaced with useful information, then storing current position in order to move to right subtree after processing the left subtree on the stack / queue is not required. The binary trees which store such information in NULL pointers are called Threaded Binary Trees.

What to store in the null links?

The common convention is put predecessor/successor information. That means, if TBT to be constructed is an In-order-TBT then for a given node left pointer will contain in-order predecessor information and right pointer will contain In-order successor information. These special Pointers are called Threads.

Types of Threaded Binary Trees:

Based on above forms we get three representations for threaded binary trees .

Pre-order Threaded Binary Trees: NULL left pointer will contain Pre-order predecessor information and NULL right pointer will contain Pre-order

1. successor information.

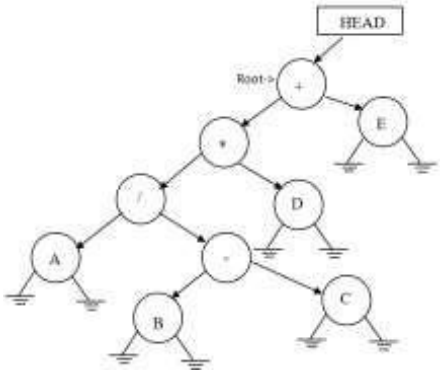
2. **In-order Threaded Binary Trees:** NULL left pointer will contain In-order predecessor information and NULL right pointer will contain In-order successor information.
3. **Post-order Threaded Binary Trees:** NULL left pointer will contain Post-order predecessor information and NULL right pointer will contain Post-order successor information.

In-Order Threaded Binary Tree

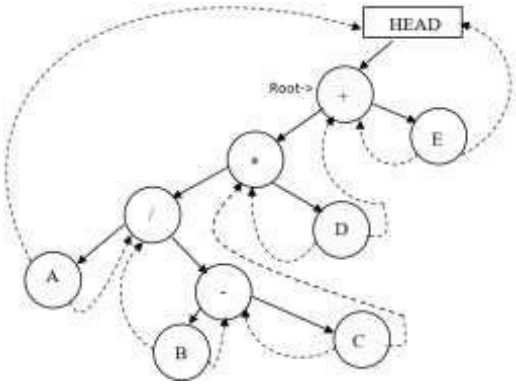
Using above concept, if right link of node p (i.e. RCHILD (p)) is null, then it can be replaced with a pointer (thread) to a node which would immediately succeed the node p (i.e. successor of p) while traversing the binary tree in In-order manner. Similarly, if left link of node p (i.e. LCHILD(p)) is null, then it can be replaced with a pointer (thread) to a node which would immediately precede the node p (i.e. predecessor of p)while traversing the binary tree in In-order manner.

For example, (Ref. figure below), let us consider a binary tree for the prefix arithmetic expression +*/A-BCDE

(Note: It is not mandatory that TBT should be constructed only by using either prefix / postfix expression.)



The tree has n=9 nodes, n-1 (i.e. 8) normal (constructional) pointers (**represented by solid directional lines**), and n+1 (i.e. 10) null links. In TBT, the n+1 null links are replaced by Threads (**represented by dotted lines**) (Ref. figure below).



If the above tree is traversed in the In-order manner, then the output would be

A / B – C * D + E----------(1)

For example, node D's left link is replaced with a pointer (Thread) to a node '*' which is an immediate predecessor of node D (Ref. In-order Traversal in 1). Node D's right link is replaced with a pointer (Thread) to a node '+' which is an immediate successor of node D (Ref. In-order Traversal in 1).

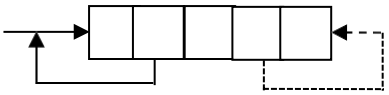
To distinguish between normal (constructional) pointers and a thread two extra one bit fields i.e. LBIT and RBIT are required in the node structure. Each node has five fields namely LBIT, LCHILD, DATA, RCHILD, RBIT and as shown below

LBIT	LCHILD	DAT A	RCHILD	RBIT

LBIT: '0' if LCHILD is a thread i.e. pointer to predecessor; '1' if LCHILD is pointer to root of LEFT binary tree
LCHILD: A pointer (address) to root node of LEFT binary tree or left thread
DATA: Atom or data item or information at node
RCHILD: A pointer (address) to root node of RIGHT binary tree or right thread
RBIT: '0' if RCHILD is a thread i.e. pointer to successor; '1' if RCHILD is pointer to root of RIGHT binary tree

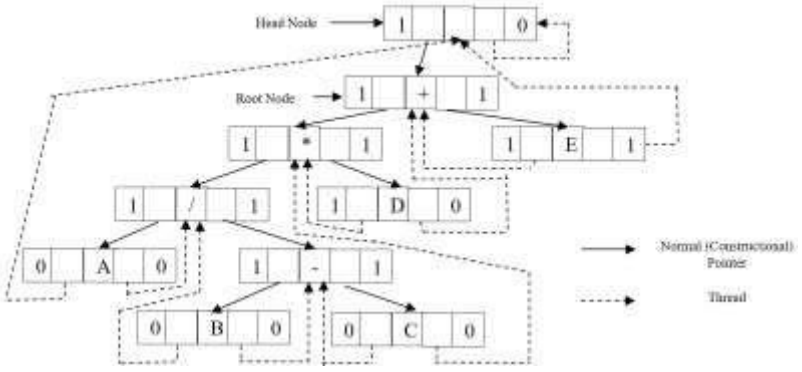
In the above figure node 'A' s left pointer and node 'E' s right pointer have been left dangling as node 'A' doesn't have any In-order predecessor and node 'E' doesn't have any In-order successor (Ref. 1). In order to have no loose Threads, all dangling pointers are connected to a node called as Head node in all the TBT's.

The empty in order threaded tree binary tree will have only one HEAD node as shown in the following diagram



Representation of the TBT in Computer's Memory by Using the Node Structure

The complete memory representation for the tree is as shown below



ALGORITHMS/ PESUDOCODE:

- **Creation of TBT**

Create_root()

```
  ///1.  Creat a Head node
    Step 1 :  head= getnode('$')
    Step 2 :  head->lthread=head-rthread=1
    Step 3 :  head->lptr=head->rptr= head
    //2. Create root node :
    Step 4 :  root =getnode()
    Step 5 :  root->lthread =rthread=1
    Step 6 : root->rptr=lptr= head
    Step 7 :  head->lptr=root
    Step 8 :  head->lthread= 0
//End of root create()
```

- **Insert node**

Insert(head , X)

```
Step 1: if(head= Null)
        i.   THEN  error message ('CREATE ROOT FIRST')
        ii.   (create_ root(head))
        iii.  return head
Step 2:  Parent =head ->lptr

Step 3:  Repeat through  step  4  till the successful insertion is not happens

Step 4:  Write ('ROOT IS' , parent->data)
     //take user choice were to insert
      '1.INSERT AT LEFT , 2.INSERT AT RIGHT'
     //based on the choice insert at left or right.
Step 5:  if choice is 1
           //insert  as a left chiled
          I. if (parent->lthread=1)//if no lchild
                i.     new = getnode (X)
                ii.    new->lptr= parent->lptr
                iii.    new->rptr= parent
                iv.     new->lthread=new->rthread= 1
                v.     parent-lptr=new
                vi.    parent->lthread=0
          II  Else
              i. parent= parent->lpter
 Step 7:  if  choice is 2
             //insert at right side
            I. if (parent->rthread=1)//if no right child
```

i. new = getnode (X)
ii. new-rptr= parent->rptr
iii. new->lptr= parent
iv. new->rthread=new->lthread= 1
v. parent->rpr=new
vi. parent->rthread=0

II. Else

i. parent= parent->rpter

tep 5 : end of insert

- **Non Recursive Preorder Traversal**

Non_preorder(head)

Step 1: current = head->lptr
Step 2: if current =head
i. THEN WRITE('EMPTY TREE')
ii. Exit
Step 3: while current ! =head
i. display(current->data) //process data in preorder
ii. if(current->lthread =0) //if having leftchiled process left subtree a. current=current->lptr
iii. else
//if no left subtree , go to right subtree
a. while (current->rthread=1)
i. current=current->rptr
b. current=current->rptr// if left subtree process it first
c. end while
Step 4: end while
Step 5: end of preorder

- **Non Recursive Inorder Traversal**

Non_Inroder(head)

Step 1: current =head->lptr
Step 2: if (current->lptr=head)
i. THEN WRITE('EMPTY TREE')
ii. Exit
Step 3: while (current ->lthread=0) //go to leftmost chiled of the left subtree
i. current=current->lptr
ii. while(current !=head)
a. display(current ->data)
. b. If current->rthread= 1
 current =current->rptr
c. else
i. current= current->rptr

ii. repeat while (current->lthread =0)
a. current =current->lptr
iii. end while
d. end if
iii. end while
Step 4: end While
Step 5: end of inorder

Analysis of Algorithm:

1. Time Complexity:

Time complexity for normal binary tree traversal whether recursive or non-recursive is also O(n), where 'n' is equal to number of nodes in binary tree, since every node is visited only once in both. For the TBT, it is still O(n), but it is constant times less than recursive and non-recursive using stack.

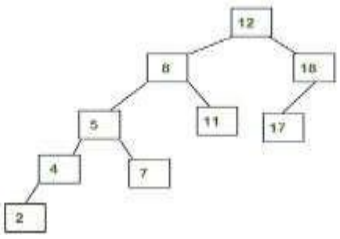
2. Space Complexity:
In recursive and non-recursive traversals of normal binary tree. We need a stack of depth 'k' which is equal to height or depth of a binary tree. But in TBT stack is not required.

Test cases / validation:

Test Cases
Test your program for following cases
For each test cases
1. tree with two nodes
2. tree with 10 nodes

INPUT :

Enter node in the following order



OUTPUT :

Show non recursive preorder and inorder traversal

Preorder : 12,8,5,4,2,7,11,18,17
Inorder : 2,4,5,7,8,11,12,17,18

CONCLUSION:

Hence we have studied the concept of threaded binary tree.

INPUT:

```
#include<bits/stdc++.h>
using namespace std;
class Node{
public:
    int data;
    Node* left;
    Node* right;
    int leftThread; // leftThread=0 -> left pointer points to the inorder predecessor
    int rightThread; // rightThread=0 -> right pointer points to the inorder successor
    Node(int val){
        this->data = val;
    }
};
class DoubleThreadedBinaryTree{
private:
    Node* root;
public:
    DoubleThreadedBinaryTree(){
        // dummy Node with value as INT_MAX
        root = new Node(INT_MAX);
        root->left = root->right = root;
        root->leftThread = 0;
        root->rightThread = 1;
    }
    void insert(int data){
        Node* new_node = new Node(data);
        if(root->left == root && root->right == root){
            //Empty Tree
            new_node->left = root;
            root->left = new_node;
            new_node->leftThread = 0;
            new_node->rightThread = 0;
            root->leftThread = 1;
            new_node->right = root;
            return;
        }
        else{
            Node* current = root->left;
            while(true){
                if(current->data > data){
                    if(current->leftThread == 0 ){
                     // this is the last Node
                     new_node->left = current->left;
                     current->left = new_node;
                     new_node->leftThread = current->leftThread;
                     new_node->rightThread = 0;
                     current->leftThread = 1;
                     new_node->right = current;
                     break;
```

```
            }
            else{
                current = current->left;
            }
        }
        else{
            if(current->rightThread == 0){
                // this is the last Node
                new_node->right = current->right;
                current->right = new_node;
                new_node->rightThread = current->rightThread;
                new_node->leftThread = 0;
                current->rightThread=1;
                new_node->left = current;
                break;
            }
            else{
                current = current->right;
            }
        }
    }
}
Node* findNextInorder(Node* current){
    if(current->rightThread == 0){
        return current->right;
    }
    current = current->right;
    while (current->leftThread != 0)
    {
        current = current->left;
    }
    return current;
}
void inorder(){
    Node* current = root->left;
    while(current->leftThread == 1){
        current = current->left;
    }
    while(current != root){
        cout<<current->data<<" ";
        current = findNextInorder(current);
    }
    cout<<"\n";
}
void preorder(){
    Node* current = root->left;
    while(current != root){
        cout<<current->data<<" ";
        if(current->left != root && current->leftThread != 0)
            current= current->left;
        else if(current->rightThread == 1){
            current = current->right;
        }
        else{
            while (current->right != root && current->rightThread == 0)
            {
                current = current->right;
            }
            if(current->right == root)
                break;
```

```
            else
            {
                current=current->right;
            }
        }
    }
    cout<<"\n";
}


};

int main(){
    DoubleThreadedBinaryTree dtbt;
    dtbt.insert(10);
    dtbt.insert(1);
    dtbt.insert(11);
    dtbt.insert(5);
    dtbt.insert(21);
    dtbt.insert(17);
    dtbt.insert(31);
    dtbt.insert(100);
    dtbt.inorder();
    dtbt.preorder();
    return 0;
}
```

OUTPUT:

```
1 5 10 11 17 21 31 100
10 1 5 11 21 17 31 100
```

Assignment 7- Minimum Spanning tree

AIM : Implementation of Minimum Spanning tree using Prims and Kruskals Algorithm

DETAILED PROBLEM STATEMENT:

Represent a graph of your college campus using adjacency list /adjacency matrix. Nodes should represent the various departments/institutes and links should represent the distance between them. Find minimum spanning tree

a) Using Kruskal's algorithm.
b) Using Prim's algorithm.

OBJECTIVE

1. To study Graph theory.
2. To study different graph traversal methods.
3. To understand the real time applications of graph theory.
4. To Implement a MST using Prims and Kruskals algorithm

OUTCOME :

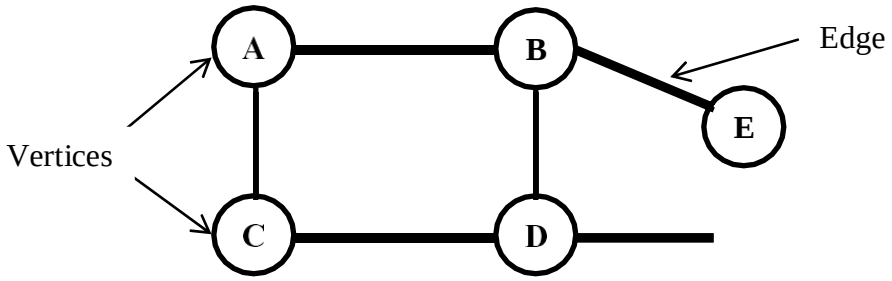
1. Understand Non-linear data structure - graph.
2. Represent a graph using adjacency list / adjacency matrix.
3. Implement Prim's and Kruskal's Algorithm
4. Identify applications of Minimum Spanning Trees.
5. Analyze the Time and Space complexity.

THEORY:

Introduction Graph :

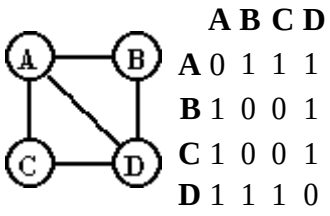
- **Definition :** A graph is a triple G = (V,E, φ)where • V is a finite set, called the vertices of G, E is a finite set, called the edges of G, and,φ is a function with domain E and codomain P2(V).
- **Loops:** A loop is an edge that connects a vertex to itself.
- **Degrees of vertices**: Let G = (V,E, φ) be a graph and v ∈ V a vertex. Define the degree of v, d(v) to be the number of e ∈ E such that v ∈ φ(e); i.e., e is Incident on v.
- **Directed graph:** A directed graph (or digraph) is a triple D = (V,E, φ) where V and E are finite sets and φ is a function with domain E and codomain V × V . We call E the set of edges of the digraph D and call V the set of vertices of D.
- **Path:** Let G = (V,E, φ) be a graph.
  Let e1, e2, . . . , en−1 be a sequence of elements of E (edges of G) for which there is a sequence a1, a2, . . . , an of distinct elements of V (vertices of G) such that φ(ei) = {ai, ai+1} for i = 1, 2, . . . , n − 1. The sequence of edges e1, e2, . . . , en−1 is called a path in G. The sequence of vertices a1, a2, . . . , an is called the vertex sequence of the path.
- **Circuit and Cycle:** Let G = (V, E, φ) be a graph and let e1, . . . , en be a trail with vertex sequence a1, . . . , an, a1. (It returns to its starting point.) The subgraph G′of Ginduced by the set of edges{e1,...,en}is called a circuit of
  G. The length of the circuit is n.
  e.g.:
  This graph G can be defined as G = (V, E) Where V= {A,B,C,D,E} and E=

{(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.



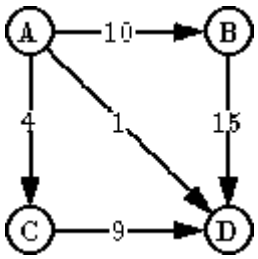
Different representations of graph:

1. **Adjacency matrix:** Graphs $G = (V, E)$ can be represented by adjacency matrices $G [v_1..v_{|V|}, v_1..v_{|V|}]$, where the rows and columns are indexed by the nodes, and the entries $G [v_i, v_j]$ represent the edges. In the case of unlabeled graphs, the entries are just Boolean values.



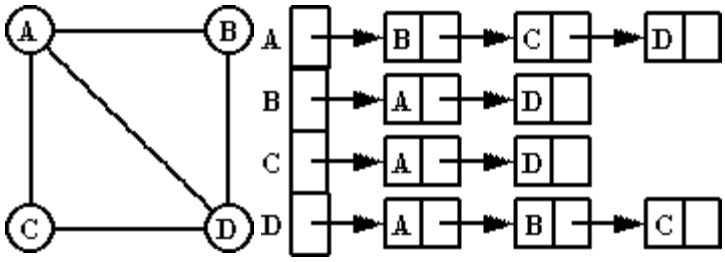
	A	**B**	**C**	**D**
A	0	1	1	1
B	1	0	0	1
C	1	0	0	1
D	1	1	1	0

In case of labeled graphs, the labels themselves may be introduced into the entries.



	A	**B**	**C**	**D**
A	∞	10	4	1
B	∞	∞	∞	15
C	∞	∞	∞	9
D	∞	∞	∞	∞

2. **Adjacency List:** A representation of the graph consisting of a list of nodes, with each node containing a list of its neighboring nodes.



- **Spanning Tree:**

A Spanning Tree of a graph G = (V, E) is a sub graph of G having all vertices of G and no cycles in it.

Minimal Spanning Tree: The cost of a graph is the sum of the costs of the edges in the weighted graph. A spanning tree of a graph G= (V, E) is called minimal cost spanning tree or simply the minimal spanning tree of G if its cost is minimum.

- When a graph G is connected, depth first or breadth first search starting at any vertex visits all the vertices in G.
- The edges of G are partitioned into two sets i.e. T for the tree edges & B for back edges. T is the set of tree edges and B for back edges. T is the set of edges used or traversed during the search & B is the set of remaining edges.
- The edges of G in T form a tree which includes all the vertices of graph G and this tree is called as spanning tree.

Definition: Any tree, which consists solely of edges in graph G and includes all the vertices in G, is called as spanning tree. Thus for a given connected graph there are multiple spanning trees possible. For maximal connected graph having ‗n' vertices the number of different possible spanning trees is equal to n.
Cycle: If any edge from set B of graph G is introduced into the corresponding spanning tree T of graph G then cycle is formed. This cycle consists of edge (v, w) from the set B and all edges on the path from w to v in T.

There are many approaches to computing a minimum spanning tree. We could try to detect cycles and remove edges, but the two algorithms we will study build them from the bottom-up in a greedy fashion.

Prim's Algorithm – starts with a single vertex and then adds the minimum edge to extend the spanning tree.

Kruskal's Algorithm – starts with a forest of single node trees and then adds the edge with the minimum weight to connect two components.

- **Prim's algorithm:** Prim's algorithm is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm was discovered in 1930 by mathematician Vojtech Jarník and later independently by computer scientist Robert C. Prim in 1957 and rediscovered by Edsger Dijkstra in 1959. Therefore it is sometimes called the DJP algorithm, the Jarník algorithm, or the Prim-Jarník algorithm.

Applications of spanning Trees:

- To find independent set of circuit equations for an electrical network. By adding an edge from set B to spanning tree we get a cycle and then Kirch off's second law is used on the resulting cycle to obtain a circuit equation.
- Using the property of spanning trees we can select the spanning tree with (n-1) edges such that total cost is minimum if each edge in a graph represents cost.
- Analysis of project planning
- Identification of chemical compounds

- Statistical mechanics, genetics, cybernetics, linguistics, social sciences

ALGORITHMS/ PESUDOCODE :

- **Prims algorithm :**

Data structure used:
Array: Two dimensional array (adjacency matrix) to store the adjacent vertices & the weights associated edges
One dimensional array to store an indicator for each vertex whether visited or not. #define max 20
int adj_ver[max][max], int edge_wt[max][max], int ind[max]

- **Algorithm to generate spanning tree by Prim's**

Prims(Weight ,Vertex)
//Weight is a two dimensional array having V no of rows and columns. KOWN ,cost , PRIEV are the l vectors.
Step 1: Repeat for I = 1 to V
i. KNOWN[I] = 0
ii. PREV[I] = 0
iii. cost[I] = 32767
Step 2: current= 1 //starting vertex of prims
Step 3: Total_V =0 //total vertex considered till the time
Step 4: KNOWN[current] =1 // Start is known now

Step 5: Repeat thru step 6 while Total_v != Vertex
i. mincost= 32767
ii. Repeat for I =1 to V
iii. If (weight[current][I] != 0 AND KNOWN[I] =0)
i. If (cost[I] >= weight[current][I])
a. cost[I] = weight[current][I]
ii. end if
iv. Repeat for I =1 to V //finding min cost edge from current vertices
i. If (KNOWN[I] = 0 AND cost[I] <= mincost)
a. mincost =cost[I] //if min is Cost[i]
b. current = I //next node visited is I
ii. end if
v. end if
Step 6: KNOWN[current] = 1
Step 7: Toatal_v= Total_v + 1
Step 8: mincost = 0
Step 9: Repeat for I = 1 to V
i. WRITE(I , PREV[I]) //display mst edges
ii. If cost[I] != 32767
a. mincost =mincost + cost[I]
Step 10 : display final mincost
Step 11: end of prims

Trace of Prim‘s algorithm for graph G1 starting from vertex 1

Step No.	**Set A**	**Set (V-A)**	**Min cost Edge (u, v)**	**Cost**	**Set B**

Initial	{1}	{2,3,4,5,6,7}	--	--	{}
1	{1,2}	{3,4,5,6,7}	(1, 2)	1	{(1, 2)}
2	{1, 2, 3}	{4, 5, 6, 7}	(2, 3)	2	{(1,2},(2,3)}
3	{1,2,3,5}	{4, 6, 7}	(2, 5)	4	{(1,2),(2,3),(2,5)}
4	{1,2,3,5,4}	{6, 7}	(1,4)	4	{(1,2),(2,3),(2,5),(1,4)}
5	{1,2,3,5,4,7}	{6}	(4,7)	4	{(1,2),(2,3),(2,5),(1,4),(4,7)}
6	{1,2,3,5,4,7.6}	{ }	(7,6)	3	{(1,2),(2,3),(2,5),(1,4),(4,7),(7,6)}
		Total Cost		17	

Thus the minimum spanning tree for graph G1 is : A = { 1,2,3,4,5,7,6} B ={(1,2),(2,3),(2,5),(1,4),(4,7),(7,6)}, total Weight: 1+2+4+3+4+3=17

- **Kruskal's Algorithm:**

- **Prerequisite for Kruskals**

```
struct edge
{
    Number v1,v2,wt
} edge
```

- **Create edge Matrix**

AdjToEdges(int weight[][MAX], int n, edge E[])

Step 1: for i=0 to n do
Step 2: for j=i+1 to n do
i.i f(weight[i][j])
a. edge_matrix[k]. start=i
b. edge_matrix[k].end =j
c. edge_matrix[k++].Value=G[i][j]
ii. end if
Step 3:end for
Step 4: end for return k;
Step 5: end AdjToEdges

- **Function to sort the edges according to weights Algorithm**

SortEdges(edge edge_matrix[], no_edge)

Step 1:for i=0 to no_edge
Step 2: for j=i+1 to no_edge
i. if(edge_matrix[i].value> edge_matrix[j].value)
a. t=edge_matrix[i]
b. edge_matrix[i]= edge_matrix[j]
c. edge_matrix[j]=t
ii. end if
Step 3 : end for j
Step 4: end for i
Step 5: end sort edge

Kruskals (G, N)

//G is a pointer to head of the adjacency List. N is max. number of vertices.

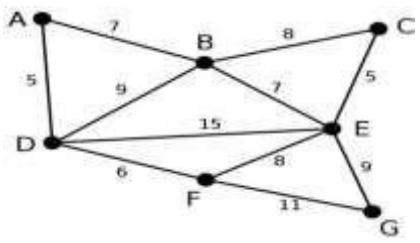
// L and K =0
Step 1: mincost = 0

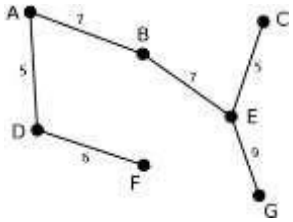
Step 2: EARRAY(G)

Step 3:Repeat for I=1 to N
i. set[I] = I

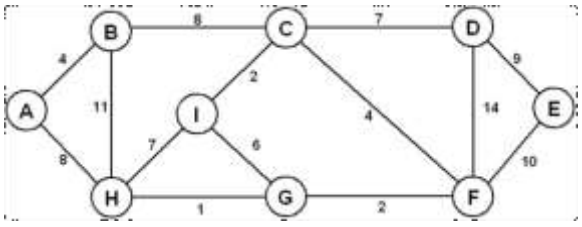
Step 4: Repeat Thru step 6 while L < Vertex-1 //select the min cost edge tillno of edges =Vertex-1
i. T = Edge_mat[K] // select min cost edge
ii. K = K+1 // to select next edge
iii. Repeat for I = 1 to N
iv. PV1=FIND(set,T.V1) //check the set Membership of V1
v.PV2=FIND(set,T.V2) // check the Set membership of V2
vi. if (PV1 != PV2) //if both v1 and v2 are belongs to different set
//they are not forming cycle and can be added to fina
a.WRITE(V1(T),V2(T) , D(T))
b. mincost = micost + Dist[T]
c. L=L+1
d. for J=1 to N
1. If (C[J] = PV2)
i. C[J] □ PV1
2. end if
e. end for
vii. end if

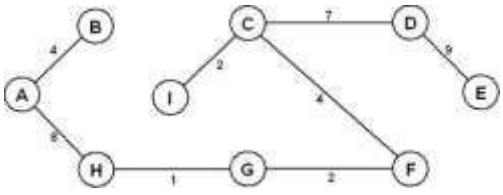
Step 5: display mincost
Step 6: end Kruskals





Cost of MST -39





Cost of MST -37

- **Sample Input/Output:**

Input: Enter vertices of a graph: 1 2 3 4 5 6 7
Enter vertex wise adjacent vertices & cost of edges.

Vertex	**Adjacent Vertex**	**Cost**
Pune (1)	2	1
	4	4
	0	
Mumbai (2)	**1**	**1**
	4	**6**
	5	**4**
	3	**2**
	0	
Bangalore (3)	**5**	**2**
	6	**8**
	0	
Hyderabad (4)	**1**	**4**
	2	**6**
	5	3
	5	3
	7	4
	0	
Chennai (5)	**2**	**4**
	3	**5**
	6	**8**
	7	**7**
	4	**3**

	0	
Delhi (6)	**3**	**8**
	5	**8**
	7	**3**
	0	
Ahmadabad (7)	**4**	**4**
	5	**7**
	6	**3**
	0	

OUTPUT : Display of adjacent matrix_adj vertices & cost of associated edges.
Pune (1): (0, 0) (2, 1) (0,0) (4, 4) (0, 0) (0, 0) (0, 0)
Mumbai (2): (1, 1) (0, 0) (3, 2) (4, 6) (5, 4) (0, 0) (0, 0)
Bangalore (3): (0, 0) (2, 2) (0, 0) (0, 0) (5, 5) (6, 8) (0, 0)
Hyderabad (4): (1, 4) (2, 6) (0, 0) (0, 0) (5, 3) (0, 0) (7, 4)
Chennai (5): (0, 0) (2, 4) (3, 5) (4, 3) (0, 0) (6, 8) (7, 7)
Delhi (6): (0, 0) (0, 0) (3, 8) (0, 0) (5, 8) (0, 0) (7, 3)
Ahmadabad (7): (0, 0) (0, 0) (0, 0) (4, 4) (5, 7) (6, 3) (0, 0)

Total cost: 17

Testcases :

a) Display the total number of comparisons required to construct the graph in computer memory.
b) Display the results as given in the sample o/p above.
c) Finally conclude on time & time space complexity for the construction of the graph and for generation of minimum spanning tree using Prim's algorithm.

Time Complexity:

Fortheconstructionofanundirectedgraphwith_n'verticesand_e'edgesusing adjacency list is O(n+e),since fore very vertex _v'in Gweneed to store all adjacent edges to vertex v.

- ✓ In Prim's algorithm to get minimum spanning tree from an undirected graph with_n' vertices using adjacency matrix is $O(n^2)$.
- ✓ Using Kruskal'salgorithm
  using adjacency matrix = $O(n^2)$.
  using adjacency list=O(eloge)

FAQS:

1. Explain the PRIM's algorithm for minimum spanning tree.
2. What are the traversal techniques?
3. What are the graph representation techniques?

CONCLUSION:

Hence we have studied the concept of prims algorithm using adjacency matrix and Kruskal's algorithm by using adjacency list.

INPUT:

A] Prims algorithm:

```
        /*primes*/
#include <iostream>
using namespace std;

class graph
{
   int G[20][20], n;

public:
   void accept()
   {
      int i, j, e;
      int src, dest, cost;
      cout << "\nEnter the no. of vertices: ";
      cin >> n;
      for (i = 0; i < n; i++) {
         for (j = 0; j < n; j++){
            G[i][j] = 0;
         }
      }

      cout << "\nEnter the no. of  Edges: ";
      cin >> e;
      for (i = 0; i < e; i++){
         cout << "\nEnter Source: ";
         cin >> src;
         cout << "\nDestination: ";
         cin >> dest;
         cout << "\nCost: ";
         cin >> cost;
```

```
        G[src][dest] = cost;
        G[dest][src] = cost;
    }
}

void display(){
    int i, j;
    for (i = 0; i < n; i++){
        cout << "\n";
        for (j = 0; j < n; j++){
            cout << "\t" << G[i][j];
        }
    }
}

void prims()
{
    int i, j, R[20][20];
    int src, dest, cost, count, min;
    int total = 0;
    int visited[20];

    for (i = 0; i < n; i++){
        for (j = 0; j < n; j++){
            if (G[i][j] == 0){
                R[i][j] = 999;
            }
            else
                R[i][j] = G[i][j];
        }
    }

    for (i = 0; i < n; i++) {
        visited[i] = 0;
    }
    cout << "\nEnter start vertex: ";
```

```
        cin >> src;
        visited[src] = 1;
        count = 0;
        while (count < n - 1) {
            min = 999;
            for (i = 0; i < n; i++){
                if (visited[i] == 1)
                    for (j = 0; j < n; j++){
                        if (visited[j] != 1){
                            if (min > R[i][j]){
                                min = R[i][j];
                                src = i;
                                dest = j;
                            }
                        }
                    }
            }
            cout << "\nEdge from " << src << " to " << dest << " \twith cost:  " <<
min;
            total = total + min;
            visited[dest] = 1;
            count++;
        }
        cout << "\nTotal Cost: " << total << "\n";
    }
};
int main()
{
    graph g;
    g.accept();
    g.display();
    g.prims();
}
```

OUTPUT OF PRIMS ALGORITHM:

Enter the no. of vertices: 4

Enter the no. of Edges: 5

Enter Source: 0

Destination: 1

Cost: 15

Enter Source: 1

Destination: 2

Cost: 15

Enter Source: 0

Destination: 2

Cost: 5

Enter Source: 3

Destination: 1

Cost: 2

Enter Source: 3

Destination: 2

Cost: 40

```
        0       15      5       0
        15      0       15      2
        5       15      0       40
        0       2       40      0
Enter start vertex: 0

Edge from 0 to 2        with cost:  5
Edge from 0 to 1        with cost:  15
Edge from 1 to 3        with cost:  2
Total Cost: 22
```

B] Kruskal's Algorithm:

```
#include <iostream>
#define INFINITY 999
using namespace std;
class kruskal{
    typedef struct graph{
        int v1, v2, cost;
    }GR;
    GR G[20];
    public:
    int tot_edges , tot_nodes;
    void create();
    void Spanning_tree();
    void get_input();
    int minimum(int);
};
void kruskal::get_input(){
    cout<<"\nEnter total number of nodes: ";
    cin>>tot_nodes;
    cout<<"\nEnter total number of edges: ";
    cin>>tot_edges;
}
void kruskal::create(){
```

```
    for(int k=0; k<tot_edges; k++){
        cout<<"\nEnter edge v1:  ";
        cin>>G[k].v1;
        cout<<"\nEnter edge v2:  ";
        cin>>G[k].v2;
        cout<<"\nEnter corresponding cost: ";
        cin>>G[k].cost;
    }
}
int kruskal::minimum(int n){
    int i,small, pos;
    small=INFINITY;
    pos=-1;
    for(i=0; i<n; i++){
        if(G[i].cost<small){
            small=G[i].cost;
            pos=i;
        }
    }
    return pos;
}
void un(int i, int j, int parent[]){
    if(i<j)
    parent[j]=i;
    else
    parent[i]=j;
}
int find(int v2, int parent[]){
    while(parent[v2]!=v2){
        v2=parent[v2];
    }
    return v2;
}
void kruskal:: Spanning_tree(){
    int count, k, v1, v2, i, j, tree[10][10], pos, parent[10];
    int sum;
```

```
    count=0;
    k=0;
    sum=0;
    for(i=0; i<tot_nodes; i++)
    parent[i]=i;
    while(count!=tot_nodes-1){
        pos=minimum(tot_edges);
        if(pos==-1)
        break;
        v1=G[pos].v1;
        v2=G[pos].v2;
        i=find(v1,parent);
        j=find(v2,parent);
        if(i!=j){
            tree[k][0]=v1;
            tree[k][1]=v2;
            k++;
            count++;
            sum+=G[pos].cost;
            un(i,j,parent);
        }
        G[pos].cost=INFINITY;
    }
    if(count==tot_nodes-1){
        cout<<"\nSpanning tree is: "<<endl;
        for(i=0; i<tot_nodes-1; i++){
            cout<<"|"<<tree[i][0];
            cout<<" ";
            cout<<tree[i][1]<<"|"<<endl;
        }
        cout<<"cost of spanning tree is: "<<sum<<endl;
    }
    else{
        cout<<"There is no spannin tree"<<endl;
    }
}
```

```
int main()
{
   kruskal obj;
   obj.get_input();
   obj.create();
   obj.Spanning_tree();
return 0;
}
```

OUTPUT OF KRUSKAL'S ALGORITHM:

```
Enter total number of nodes: 8

Enter total number of edges: 9

Enter edge v1:  1

Enter edge v2:  2

Enter corresponding cost: 5

Enter edge v1:  2

Enter edge v2:  3

Enter corresponding cost: 6

Enter edge v1:  4

Enter edge v2:  3

Enter corresponding cost: 2

Enter edge v1:  1
```

Enter edge v2: 4

Enter corresponding cost: 9

Enter edge v1: 3

Enter edge v2: 5

Enter corresponding cost: 5

Enter edge v1: 5

Enter edge v2: 6

Enter corresponding cost: 10

Enter edge v1: 6

Enter edge v2: 7

Enter corresponding cost: 7

Enter edge v1: 7

Enter edge v2: 8

Enter corresponding cost: 1

Enter edge v1: 8

Enter edge v2: 5

Enter corresponding cost: 1

Spanning tree is:
|7 8|

|8 5|
|4 3|
|1 2|
|3 5|
|2 3|
|6 7|
cost of spanning tree is: 27

Assignment 8

Title: Implementation of Dijikstra's algorithm

AIM: Represent a given graph using adjacency matrix /adjacency list and find the shortest path using Dijkstra's algorithm (single source all destination).

OBJECTIVE:

1. To understand the application of Dijkstra's algorithm

THEORY:

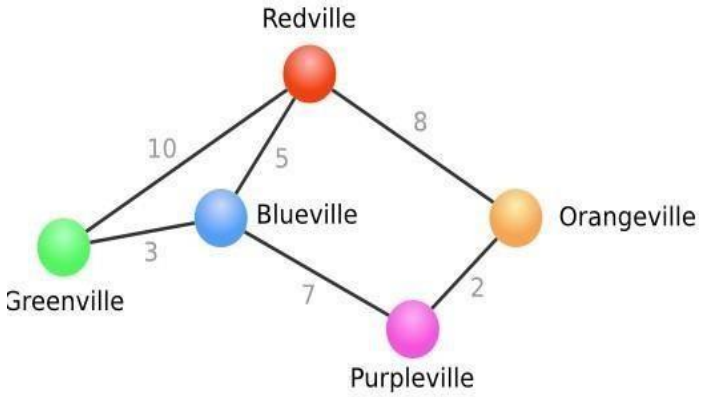
1. Explain in brief with examples how to find the shortest path using Dijkstra's algorithm.

Definition of Dijkstra's Shortest Path

1. To find the shortest path between points, the weight or length of a path is calculated as the sum of the weights of the edges in the path.
2. A path is a shortest if there is no path from x to y with lower weight.
3. Dijkstra's algorithm finds the shortest path from x to y in order of increasing distance from x. That is, it chooses the first minimum edge, stores this value and adds the next minimum value from the next edge it selects.
4. It starts out at one vertex and branches out by selecting certain edges that lead to new vertices.
5. It is similar to the minimum spanning tree algorithm, in that it is "greedy", always choosing the closest edge in hopes of an optimal solution.

Example:

It is easiest to think of the geographical distances, with the vertices being places, such as cities.



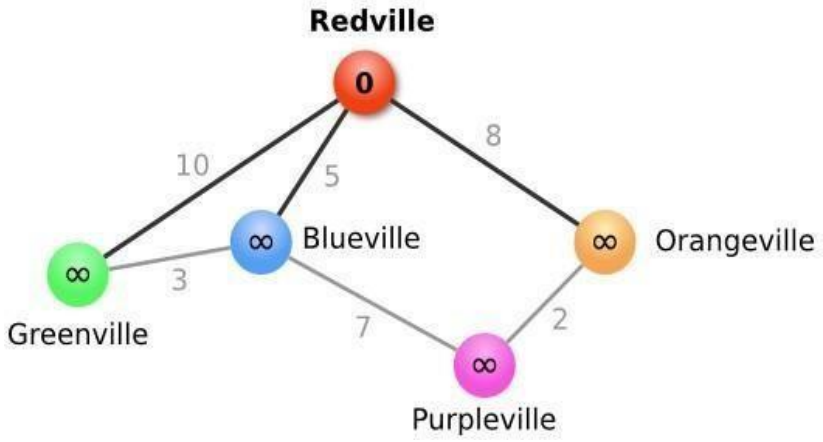
Imagine you live in Redville, and would like to know the shortest way to get to the surrounding towns: Greenville, Blueville, Orangeville, and Purpleville. You would be confronted with problems like: Is it faster to go through Orangeville or Blueville to get to Purpleville? Is it faster to take a direct route to Greenville, or to take the route that goes through Blueville? As long as you knew the distances of roads going directly from one city to another, Dijkstra's algorithm would be able to tell you what the best route for each of the nearby towns would be.

- Begin with the source node (city), and call this the current node. Set its value to 0. Set the value of all other nodes to infinity. Mark all nodes as unvisited.
- For each unvisited node that is adjacent to the current node (i.e. a city there is a direct route to from the present city), do the following. If the value of the current node plus the value of the edge is less than the value of the adjacent node, change the value of the adjacent node to this value. Otherwise leave the value as is.
- Set the current node to visited. If there are still some unvisited nodes, set the unvisited node with the smallest value as the new current node, and go to step 2. If there are no unvisited nodes, then we are done.

In other words, we start by figuring out the distance from our hometown to all of the towns we have a direct route to. Then we go through each town, and see if there is a quicker route through it to any of the towns it has a direct route to. If so, we remember this as our current best route.

Step I:

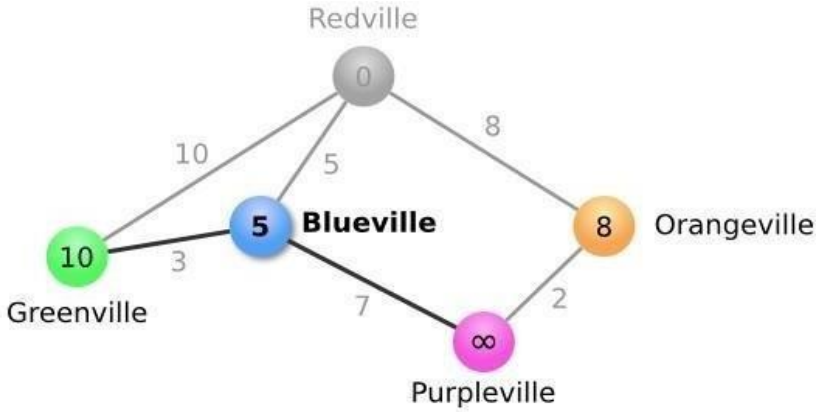
We set Redville as our current node. We give it a value of 0, since it doesn't cost anything to get to it from our starting point. We assign everything else a value of infinity, since we don't yet know of a way to get to them.



Step II:

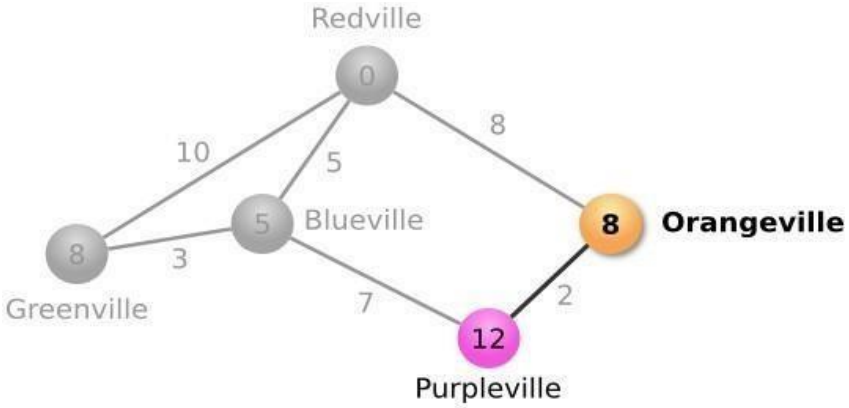
Next, we look at the unvisited cities our current node is adjacent to. This means Greenville, Blueville and Orangeville. We check whether the value of the connecting edge, plus the value of our current node, is less than the value of the adjacent node, and if so we change the value. In this case, for all three of the adjacent nodes we should be changing the value, since all of the adjacent nodes have the value infinity. We change the value to the value of the current node (zero) plus the value of the connecting edge (10 for Greenville, 5 for

Blueville, 8 for Orangeville). We now mark Redville as visited, and set Blueville as our current node since it has the lowest value of all unvisited nodes.



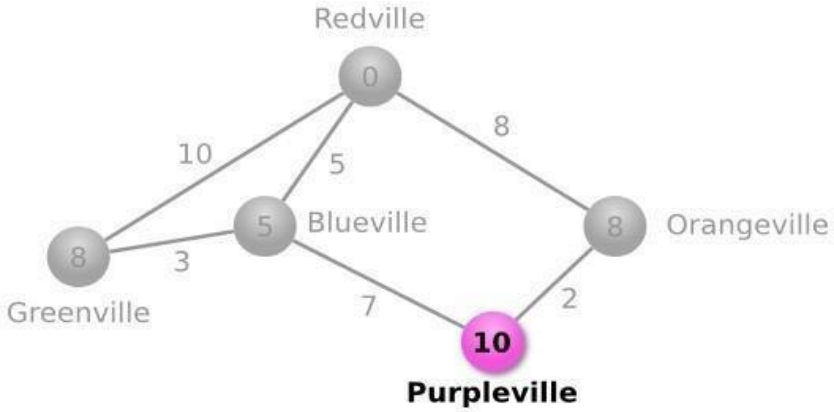
Step III:

The unvisited nodes adjacent to Blueville, our current node, are Purpleville and Greenville. So we want to see if the value of either of those cities is less than the value of Blueville plus the value of the connecting edge. The value of Blueville plus the value of the road to Greenvile is 5 + 3 = 8. This is less than the current value of Greenville (10), so it is shorter to go through Blueville to get to Greenville. We change the value of Greenville to 8, showing we can get there with a cost of 8. For Purpleville, 5 + 7 = 12, which is less than Purpleville‘s current value of infinity, so we change its value as well? We mark Blueville as visited. There are now two unvisited nodes with the lowest value (both Orangeville and Greenville have value 8). We can arbitrarily choose Greenville to be our next current node. However, there are no unvisited nodes adjacent to Greenville! We can mark it as visited without making any other changes, and make Orangeville our next current node.



Step IV:

There is only one unvisited node adjacent to Orangeville. If we check the values, Orangeville plus the connecting road is 8 + 2 = 10, Purpleville's value is 12, and so we change Purpleville's value to 10. We mark Orangeville as visited, and Purpleville is our last unvisited node, so we make it our current node. There are no unvisited nodes adjacent to Purpleville, so we're done!



All above steps can be simply put in a tabular form like this:

Current	Visited	Red	Green	Blue	Orange	Purple	Description
Red		0	Infinity	Infinity	Infinity	Infinity	Initialize Red as current, set initial values
Red		0	10	5	8	Infinity	Change values for Green, Blue, Orange
Blue	Red	0	10	5	8	Infinity	Set Red as visited, Blue as current
Blue	Red	0	8	5	8	12	Change value for Purple
Green	Red, Blue	0	8	5	8	12	Set Blue as visited, Green as current
Orange	Red, Blue, Green	0	8	5	8	12	Set Green as visited, Orange as current
Orange	Red, Blue, Green	0	8	5	8	10	Change value for Purple
Purple	Red, Blue, Green, Orange	0	8	5	8	10	Set Orange as visited, Purple as current
	Red, Blue, Green, Orange, Purple	0	8	5	8	10	Set Purple as visited

ALGORITHM:

College Area represented by Graph.
A graph G with N nodes is maintained by its adjacency matrix Cost. Dijkstra‘s algorithm find shortest path matrix D of Graph G.
Staring Node is 1.

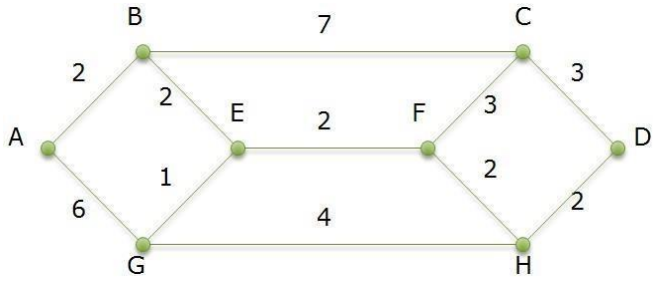
Step 1: Repeat Step 2 for I = 1 to N
D[I] = Cost[1][I].
Step 2: Repeat Steps 3 & 4 for I = 1 to N
Step 3: Repeat Steps 4 for J = 1 to N
Step 4: If D[J] > D[I] + D[I][J]
Then D[J] = D[I] + D[I][J] Step 5:
Stop.

INPUT:

The graph in the form of edges, nodes and corresponding weights, the source node and destination node.

OUTPUT:

The shortest path and length of the shortest path

INPUT	OUTPUT	Remark
	Shortest Path is A-B-E-F-H-D Cost - 10	Consider Source Vertex as node A and Destination Vertex as node D

FAQs:

1. What is shortest path?
2. What are the graph representation techniques?
3. What is adjacency Matrix?
4. What is adjacency list?
5. What is adjacency Multi-list?

CONCLUSION:

Hence, we can find the shortest path using Dijkstra's algorithm from single source to all destination.

INPUT:

```
#include <iostream>
using namespace
std; class graph{
int g[20][20];
int e,v;
public:
    void accept();     void
display();     void
dijkstra(int start);
}; void graph:: accept(){     int src, dest,
cost, i,j;     cout<<"Enter the number of
vertices:  ";     cin>>v;
    cout<<"Enter the number of edges:  ";
cin>>e;     for(i=0; i<v; i++){
for(j=0; j<v;j++){
g[i][j]=0;
        }     }     for(i=0; i<e; i++){
cout<<"\nEnter source vertex:  ";
cin>>src;
        cout<<"Enter destination vertex:  ";
cin>>dest;
        cout<<"Enter the cost of the edge:  ";
cin>>cost;         g[src][dest]=cost;
g[dest][src]=cost;
    } } void
graph::display(){
int i,j;     for(i=0;
i<v; i++){
cout<<endl;
for(j=0; j<v; j++){
cout<<g[i][j]<<"\t";
        }
    } }
void graph::dijkstra(int start){
int r[20][20],
visited[20],distance[20],from[20],i,j,cnt,mindst,next;
for(i=0; i<v; i++){         for(j=0; j<v; j++){
if(g[i][j]==0){                 r[i][j]=999;
            }
else{
                r[i][j]=g[i][j];
```

```
            }
        }     }     for(i=0;
i<v; i++){          visited[i]=0;
from[i]=start;
distance[i]=r[start][i];
    }
    distance[start]=0;
visited[start]=1;
cnt=v;     while(cnt>0){
mindst=999;
for(i=0; i<v; i++){
            if((mindst>distance[i]) && visited[i]==0){
mindst=distance[i];                 next=i;
            }         }
visited[next]=1;          for(i=0; i<v; i++){
if(visited[i]==0 &&
distance[i]>(mindst+r[next][i])){
distance[i]=mindst+r[next][i];
from[i]=next;
            }
}         cnt--
;     }
    for(i=0; i<v; i++){
        cout<<"\nDistance of "<<i<< " from "<<start<<" is
"<<distance[i]<<endl<<"Path "<<i;
j=i;         do{
j=from[j];             cout<<"<-
"<<j;
        }
        while(j!=start);
    } } int
main() {
graph g;
int s;
    g.accept();
    g.display();
    cout<<"\nEnter the starting vertex: ";
cin>>s;
    g.dijkstra(s);    return
0;
}
```

OUTPUT:

```
Enter the number of vertices:  6
Enter the number of edges:  8

Enter source vertex:  0
Enter destination vertex:  1
Enter the cost of the edge:  7

Enter source vertex:  1
Enter destination vertex:  2
Enter the cost of the edge:  9

Enter source vertex:  2
Enter destination vertex:  3
Enter the cost of the edge:  1

Enter source vertex:  4
Enter destination vertex:  3
Enter the cost of the edge:  5

Enter source vertex:  5
Enter destination vertex:  4
Enter the cost of the edge:  10

Enter source vertex:  0
Enter destination vertex:  5
Enter the cost of the edge:  12

Enter source vertex:  1
Enter destination vertex:  5
Enter the cost of the edge:  2

Enter source vertex:  4
Enter destination vertex:  2
Enter the cost of the edge:  4

0     7     0     0     0     12
7     0     9     0     0     2
0     9     0     1     4     0
0     0     1     0     5     0
0     0     4     5     0     10
12    2     0     0     10    0
```

```
Enter the starting vertex: 0

Distance of 0 from 0 is 0
Path 0<-0
Distance of 1 from 0 is 7
Path 1<-0
Distance of 2 from 0 is 16
Path 2<-1<-0
Distance of 3 from 0 is 17
Path 3<-2<-1<-0
Distance of 4 from 0 is 19
Path 4<-5<-1<-0
Distance of 5 from 0 is 9
Path 5<-1<-0
```

Assignment 9 - Heap sort

AIM: **Implementation of Heap sort**

DETAILED PROBLEM STATEMENT :

Implement Heap sort to sort given set of values using max or min heap.

OBJECTIVE:

1. To know the concept of a Heap data structure
2. To learn heap sort
3. To analyze heap sort time and space complexity

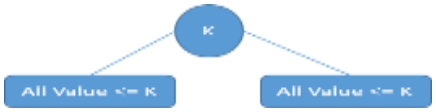
OUTCOME:

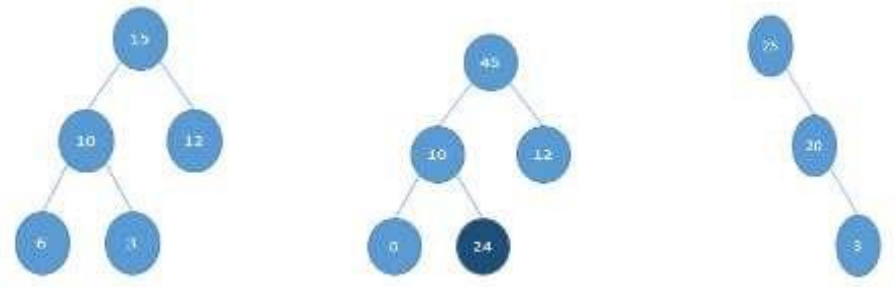
1. Understand the properties of Heap data structure.
2. Implement Heap sort.
3. Analysis of heap sort

THEORY:

Heap Data Structure :

i) Heap Tree:
Heap is a binary tree structure with the following properties:
1. The tree is complete or almost complete
2. The key value of each node is greater than or equal to the key value of its descendants.





(A) Valid Heap Tree

(B) Invalid Heap Tree due to sub-tree 10

(C) Invalid Heap Tree not nearly complete

ii) Max-Heap

If the key value of each node is greater than or equal to the key value of its descendants, this heap structure is called Max-Heap.

iii) Min-Heap

If the key value of each node is less than or equal to the key value of its descendants, this heap structure is called Min-Heap.

b)Maintenance Operations on Heap

To perform insert and delete a node in heap structure, it needs two basic algorithms :

1.**Reheap up**

This operation reorders a "broken" heap by floating the last element up the tree until it is at its correct position in the heap.

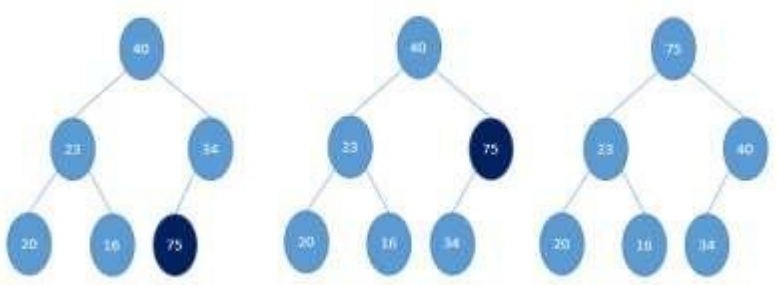


Fig. a
Not a heap tree

Fig. b
node 75 moved up
at correct position,

Fig. c
node 75 moved
it is heap tree

2.Reheap down

This operation reorders a "broken" heap by pushing the root down the tree until it is at its correct position in the heap.

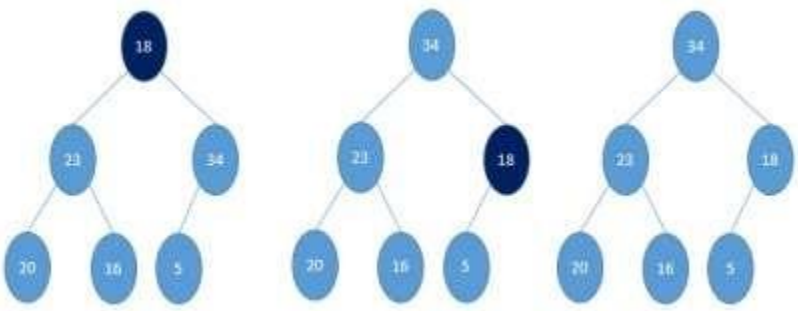


Fig. a
Not a heap tree

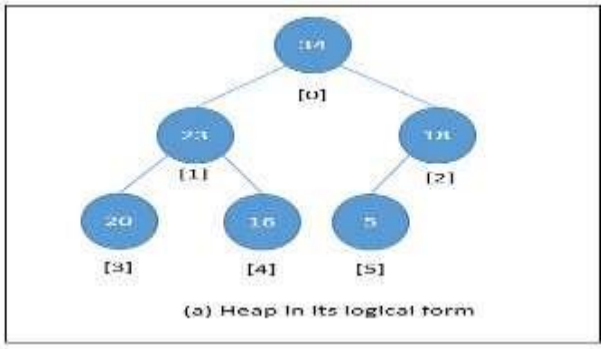
Fig. b
Moved down node
18 at its correct place

Fig. c
It is heap tree
(Max heap)

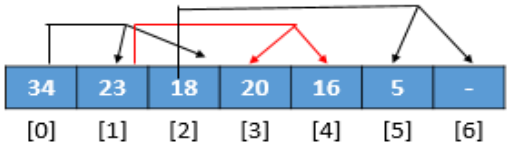
Heap Implementation

Heap implementation is possible using an array because it must be a complete or almost complete binary tree, which allows a fixed relationship between each node and its children and it can be calculated as :

i. For node located at ith index, its children are :
   Left child : 2i + 1
   Right child : 2i + 2
ii. Parent of node located at ith index : (i-1)/2
iii. Given the index for a left child, j,its right sibling at : j+1
iv. Given the index for a right child, k, its left sibling at :k-1



(a) Heap in its logical form



(b) Heap representation in an array

Applications of Heap

i. Selection algorithm
ii. Priority Queue
iii. Sorting (Heap Sort)

Heap Sort Complexity

The Heap sort efficiency is O(n log n)

ALGORITHMS/PESUDOCODE:

Heapify_asc(data,i)

Step 1 : Set Lt = Left(i)
Step 2 : Set Rt = Right(i)
Step 3 : if Lt <= heap_size[data] - 1 and data[Lt] > data[i]
i. then Set Max = Lt;
else
ii. Set Max = i
Step 4 : if Rt <= heap_size[data] - 1 and data[Rt] > data[Max]
i. then Set Max = Rt
Step 5 : if Max != i
i. then Swap(data[i], data[Max])
Step 6 : Heapify_asc(data, Max)
Step 7: end of Heapify_asc

BuildHeap(data)

Step 1 : Set heap_size[data] = length[data]
Step 2 : for i= (length[data]-1)/2 to 0
i. Heapify(data,i)
Step 3: End of BuildHEap

Heapsort(data)

Step 1 : BuildHeap(data)
Step 2 : for i = length[data] - 1 to 1
i. Swap(data[0],data[i])
ii. Set heap_size[data] = heap_size[data] - 1
iii. Heapify(data,0)
Step 3: End of heapsort

INPUT:

1. 45, 78, 24, 36, 12
2. 10, 28, 56, 68, 75
3. 89, 70, 64, 52,

OUTPUT:

Enter how many elements you want to sort? :45 5

Enter elements :45 78 36 24 12
1. Ascending order
2. Descending order
3. Exit
Enter your choice :1

size = 5 78 45 36 24 12
After building the heap…
Sorted Elements are: 12 24 36 45 78
1. Ascending order
2. Descending order
3. Exit
Enter your choice :2

size = 5 12 24 36 45 78
After building the heap...
Sorted Elements are: 78 45 36 24 12
1. Ascending order
2. Descending order
3. Exit
Enter your choice :3

FAQS:

1. How do you sort an array using heap sort?
2. What are the heap properties ?
3. What is the space complexity of heap Sort?
4. Heap sort time complexity ?
5. Compare heap sort with quick and merge sort?

CONCLUSION:

Hence, we have studied the concept of heap sort .

INPUT:

```
#include <iostream>
using namespace std;

void maxHeapify(int a[], int i, int n){
   int j, temp;
   temp=a[i];
   j=2*i;
   while(j<=n){
      if(j<n && a[j+1]>a[j])
      j=j+1;
      if(temp>a[j])
      break;
```

```
        else if(temp<=a[j]){
            a[j/2]=a[j];
            j=2*j;
        }
    }
    a[j/2]=temp;
    return;
}
void build_maxheap(int a[], int n){
    int i;
    for(i=n/2 ; i>=1; i--){
    maxHeapify(a,i,n);
    }
}
void max_HeapSort(int a[], int n){
    int i, temp;
    for(i=n; i>=2; i--){
        temp = a[i];
        a[i] = a[1];
        a[1] = temp;
        maxHeapify(a, 1, i-1);
    }
}

void min_heapify(int a[], int i, int n){
    int j, temp;
    temp = a[i];
    j = 2*i;
    while(j<=n){
        if(j<n && a[j+1]<a[j])
        j=j+1;
        if(temp<a[j])
        break;
        else if(temp>=a[j]){
            a[j/2] = a[j];
            j= 2*j;
        }
    }
    a[j/2] = temp;
    return;
}
void build_minheap(int a[], int n){
```

```
    int i;
    for(i=n/2; i>=1; i--){
        min_heapify(a,i,n);
    }
}
void min_HeapSort( int a[], int n){
    int i, temp;
    for(i=n; i>=2; i--){
        temp = a[i];
        a[i] = a[1];
        a[1] = temp;
        min_heapify(a, 1, i-1);
    }
}

void print(int arr[], int n){
    cout<<"\nsorted data: ";
    for(int i=1; i<=n; i++){
    cout<<"->"<<arr[i];
    }
    return;
}
int main()
{
    int n, i, ch;
    cout<<"Enter the number of elements to be sorted:  " ;
    cin>>n;
    int arr[n];
    for(i=1; i<=n; i++) {
        cout<<"Enter element "<<i<<": ";
        cin>>arr[i];
    }

    do{
        cout<<"\n\n1]Heap sort using max heap";
        cout<<"\n2]Heap sort using min heap";
        cout<<"\n3]Exit";

        cout<<"\nEnter your choice: ";
        cin>>ch;
        switch(ch){
            case 1:
```

```
            build_maxheap(arr, n);
            max_HeapSort(arr, n);
            print(arr, n);
            break;
          case 2:
            build_minheap(arr, n);
            min_HeapSort(arr, n);
            print(arr, n);
            break;
       }
   }while(ch!=3);
return 0;
}
```

OUPUT:

```
Enter the number of elements to be sorted:  5
Enter element 1: 3
Enter element 2: 2
Enter element 3: 4
Enter element 4: 1
Enter element 5: 5

1]Heap sort using max heap
2]Heap sort using min heap
3]Exit
Enter your choice: 1

sorted data: ->1->2->3->4->5

1]Heap sort using max heap
2]Heap sort using min heap
3]Exit
Enter your choice: 2

sorted data: ->5->4->3->2->1

1]Heap sort using max heap
2]Heap sort using min heap
```

3]Exit
Enter your choice: 3

Assignment 10 - Sequential file Organization

AIM: To implement program for Sequential Access File

DETAILED PROBLEM STATEMENT:

Department maintains student's database. The file contains roll number, name, division and address. Write a program to

i. create a sequential file to store and maintain student data.
ii. It should allow the user to add, delete information of student.
iii. Display information of particular student.
-> If record of student does not exist an appropriate message is displayed.
->If student record is found it should display the student details.

OBJECTIVE:

1. Understand the concept of Permeant Data structure
2. To learn Sequential file organization.
3. Create a sequential file and various operations on file .
4. Design the application using file data structure for database management .

OUTCOME:

1. Understand file as a permanent data structure.
2. Implement a file and preform various operations on it.
3. To implement the Database application using file data structure

THEORY :

File :
A file is a collection on information, usually stored on a computer's disk. Information can be saved to files and then later reused.

Type of File

- **Binary File**

- The binary file consists of binary data
- It can store text, graphics, sound data in binary format
- The binary files cannot be read directly o Numbers stored efficiently

- **Text File**

- The text file contains the plain ASCII characters
- It contains text data which is marked by _end of line' at the end of each record
- This end of record marks help easily to perform operations such as read and write o Text file cannot store graphical data.

- **File Organization :**

The proper arrangement of records within a file is called as file organization. The factors that affect file organization are mainly the following:

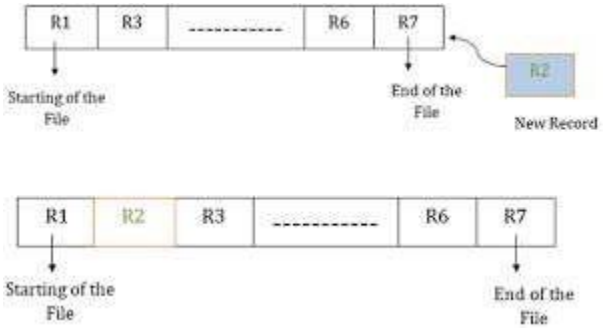
- Storage device
- Type of query
- Number of keys
- Mode of retrieval/update of record

- **Different types of File Organizations are as :**

- Sequential file
- Direct or random access file
- Indexed sequential file
- Multi-Indexed file

- **Sequential file :**

In sequential file, records are stored in the sequential order of their entry.
This is the simplest kind of data organization. The order of the records is fixed. Within each block, the records are in sequence . A sequential file stores records in the order they are entered. New records always appear at the end of the file.



Features of Sequential files :

- Records stored in pre-defined order.
- Sequential access to successive records.
- Suited to magnetic tape.
- To maintain the sequential order updating becomes a more complicated and difficult task**.** Records will usually need to be moved by one place in order to add (slot in) a record in the proper sequential order. Deleting records will usually require that records be shifted back one place to avoid gaps in the sequence.
- Very useful for transaction processing where the hit rate is very high e.g. payroll systems, when the whole file is processed as this is quick and efficient.
- Access times are still too slow (no better on average than serial) to be useful in online applications.

Drawbacks of Sequential File Organization

- Insertion and deletion of records in in-between positions huge data
- Movement Accessing any record requires a pass through all the preceding records,

which is time consuming. Therefore, searching a record also takes more time.
- Needs reorganization of file from time to time.
- If too many records are deleted logically, then the file must be reorganized to free the space occupied by unwanted records

Primitive Operations on Sequential files

- **Open—**This opens the file and sets the file pointer to immediately before the first record
- **Read-next—**This returns the next record to the user. If no record is present, then EOF condition will be set.
- **Close—**This closes the file and terminates the access to the file
- **Write-next—**File pointers are set to next of last record and write the record to the file
- **EOF—**If EOF condition occurs, it returns true, otherwise it returns false
- **Search—**Search for the record with a given key
- **Update—**Current record is written at the same position with updated values

➢ **Direct or random access file :**

Files that have been designed to make direct record retrieval as easy and efficiently as possible is known as directly organized files. Though we search records using key, we still need to know the address of the record to retrieve it directly. The file organization that supports Files such access is called as direct or random file organization. Direct access files are of great use for immediate access to large amounts of information. They are often used in accessing large databases.

Advantages of Direct Access Files :

- Rapid access to records in a direct fashion.
- It doesn't make use of large index tables and dictionaries and therefore response times are very fast.

➢ **Indexed sequential file :**

- Records are stored sequentially but the index file is prepared for accessing the record directly. An index file contains records ordered by a record key. The record key uniquely identifies the record and determines the sequence in which it is accessed with respect to other records.

- A file that is loaded in key sequence but can be accessed directly by use of one or more indices is known as an indexed sequential file. A sequential data file that is indexed is called as indexed sequential file. A solution to improve speed of retrieving target is index sequential file. An indexed file contains records ordered by a record key. Each record contains a field that contains the record key.

- This system organizes the file into sequential order, usually based on a key field, similar in principle to the sequential access file. However, it is also possible to directly access records by using a separate index file. An indexed file system consists of a pair of files: one holding the data and one storing an index to that data.The index file will store the addresses of the records stored on the main file. There may be more than one index created for a data file e.g. a library may have itsbooks stored on computer with indices on author, subject and class mark.

Characteristics of Indexed Sequential File :

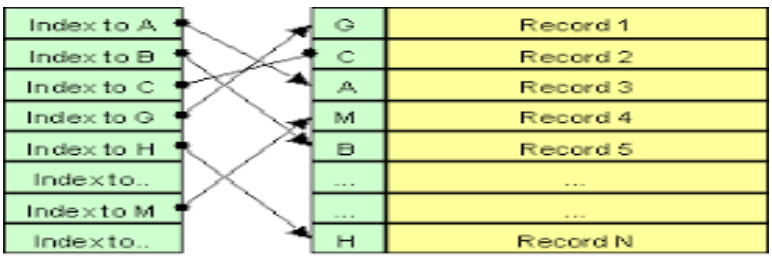
- Records are stored sequentially but the index file is prepared for accessing the
- record directly
- Records can be accessed randomly
- File has records and also the index
- Magnetic tape is not suitable for index sequential storage
- Index is the address of physical storage of a record
- When randomly very few are required/accessed, then index sequential is better
- Faster access method
- Addition overhead is to maintain index
- Index sequential files are popularly used in many applications like digital library

Primitive operations on Index Sequential files (IS):

- **Write (add, store) :** User provides a new key and record, IS file inserts the new
- record and key.
- **Sequential Access (read next) :** IS file returns the next record (in key order)
- **Random access (random read, fetch) :** User provides key, IS file returns the record or "not there"
- **Rewrite (replace) :** User provides an existing key and a new record, IS file replaces existing record with new.
- **Delete :** User provides an existing key, IS file deletes existing record

Indexed Sequential Files :

This is basically a mixture of sequential and indexed file organization techniques. Records are held in sequential order and can be accessed randomly through an index. Thus, these files share the merits of both systems enabling sequential or direct access to the data. The index to these files operates by storing the highest record key in given cylinders and tracks. Note how this organization gives the index a tree structure. Obviously this type of file organization will require a direct access device, such as a hard disk. Indexed sequential file organization is very useful where records are often retrieved randomly and are also processed in (sequential) key order. Banks may use this organization for their auto-bank machines i.e. customers randomly access their accounts throughout the day and at the end of the day the banks can update the whole file sequentially.



Indexed File Organisation

Advantages of Indexed Sequential Files:

1. Allows records to be accessed directly or sequentially.
2. Direct access ability provides vastly superior (average) access times.

Disadvantages of Indexed Sequential Files:

1. The fact that several tables must be stored for the index makes for a considerable storage overhead.
2. As the items are stored in a sequential fashion this adds complexity to the addition/deletion of records. Because frequent updating can be very inefficient, especially for large files, batch updates are often performed.

Application of files :
Database applications :

- ticket reservation system
- hotel management system
- online examinations
- student admission process etc.

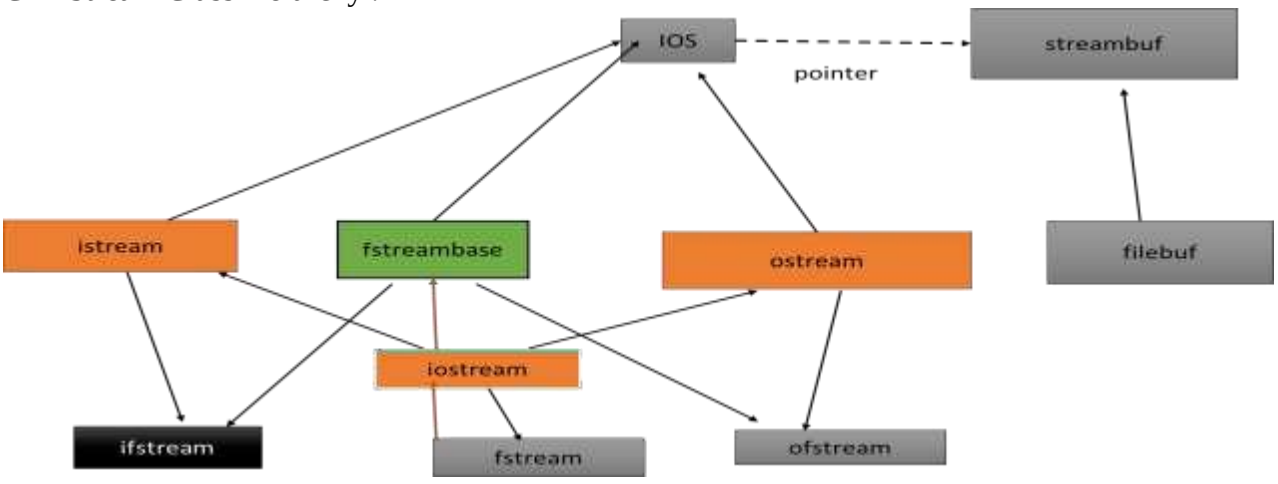
Software System applications

- operating system,
- language processors
- graphics systems etc.

To Build a vital software system of the future,

- We need to be able to structure and manipulate information effectively and efficiently
- To fulfill this we need a information management system .
- File organization is one of its component**.**

C++ Stream Class hierarchy :



Class	Functions	Meaning
Ifstream	open() get() getline() read()	Open the file in default input mode(read from file) ios::in Read one character from the file Read one complete line from the file Read any type of data from the file

	tellg(), seekg()	Direct access file functions(not required for this assignment)
Ofstream	open() put() write() tellp(), seekp()	Open the file in default output mode(write into file)ios::out Write one character into the file Write any type of data into the file Direct access file functions(not required for this assignment)
Fstream	Inherits all above functions through iostream	Provides support for simultaneous input and output operations. Ios::in|ios::out

Outline of implementation

1. 2. 3.

```
struct student                  struct DoB                  class seqfile
{                               {                           {
Int stud_rollno;                        int stud_day;               student stud_rec;
char stud_name[30];              int stud_month;           ostream outfile;
char stud_division;             int stud_year;              istream infile;
char stud_address[30];                  }
DoB stud_DoB;                                                     public:
float stud_percent;                                          void create();
char stud_grade;                                             void display(key);
}                                                            void add();
                                                        void search(key);
                                                             Void modify(key);
                                                             void delete(key);
                                                        }
```

ALGORITHMS :

Note : StudentData is file a name

- **Create a file**

CreateAFile()

Step 1 : Open StudentData for output
Step 2 : If(file opened) then
Step 3: scan noofReords
Step 4: For I =1 to noofRecords
  i. Display "Please enter the information of student: "
  ii. Get rollno, name, divison, address, date of birth, percentage, grade
  iii. Write rollno, name, divison, address, date of birth, percentage, grade into StudentData
Step 5: end for
Step 6 : Close file
Step 7: END CreateAFile

- **Display a file**

DisplayFileContents()

Step 1: Open StudentData for input
Step 2: If FileNotPresent
  i. Display error message
  ii. Exit
Step 3: end if
Step 4: while Not EndofFile StudentData
  i. Read Student Information from StudentData
  ii. Display Student Information
Step 5: end while
Step 6: Close StudentData
Step 7 : endDisplayFileContents

- **Add a record**

AddNewRecords()
Step 1: Open StudentData for append // file pointer will automatically moved to the end of file
Step2 : If FileNotPresent
  i. Display error message
  ii. Exit
Step 3: end if
Step 4: Display "Please enter the information of student: "
  i. Get rollno, name, divison, address, date of birth, percentage, grade
  ii. Write rollno, name, divison, address, date of birth, percentage, grade into StudentData

Step 5: Close StudentData
Step 6: end AddNewRecords

- **Search a record**

SearchRecord(key)
Step 1: Open StudentData for input
Step 2: if FileNotPresent
  i. Display error message
  ii. Exit
Step 3: end if
Step 4: while Not EndofFile StudentData
  i. Read Student Information from StudentData
  ii. if StudentRecord contains key //key can be unique roll no or if 'name' there can be multiple records displayed
    a. Display Student Information
  iii. ENDIF
Step 5 : end while
Step 6: Close StudentData
Step 7: end SearchRecord

- **Modify a Record**

ModifyRecord(key)
Step 1: Open StudentData for input
Step 2: if FileNotPresent
    i. Display error message
    ii. Exit
Step 3: end if
Step 4: Open Temporary file for output
Step 5: while Not EndofFile StudentData
    i. Read Student Information from StudentData
    ii. if StudentRecord contains key //key should be unique as 'roll no'
        a. Display Student Information
        b. Get new information for modification
        c. Write information into Temporary file
    iii. else
        a. Write information into Temporary file
    iv. end if
Step 6: end while
Step 7: Delete StudentData
Step 8: Rename Temporary File as StudentData
Step 9: Close StudentData
Step 10:end ModifyRecord

- **Delete a Record**

Delete Record(key)
Step 1: Open StudentData for input
Step 2: if FileNotPresent
    i. Display error message
    ii. Exit
Step 3: end if
Step 4: Open Temporary file for output
Step 5: while Not EndofFile StudentData
    i. Read Student Information from StudentData
    ii. if StudentRecord contains key //key should be unique as 'roll no'
        a. Continue // read next record
    iii. else
        a. Write information into Temporary file
    iv. endif
Step 6: end whil
Step 7: Delete StudentData
Step 8: Rename Temporary File as StudentData
Step 9: Close StudentData
Step 10: end ModifyRecord

Test Cases:

1. Open a file in reading mode which does not exist.
2. Open a file in writing mode which is already exist.
3. Search /Modify/Delete record with no key present.
4. Not able to open a new file.
5. We may have invalid file name.

CONCLUSION:

Hence, we have studied the concept of file handling in c++.

INPUT:

```
#include<iostream>
#include<fstream>
using namespace std;

void create();
void display();
int search(int roll);
void add();
void del();
void modify();

int rollno;
char name[10];
char dept[10];

fstream f,r,temp;

int main()
{
   int x=1,ch;

   while(x)
      {
         cout<<"\nEnter your choice what you want to perform:-\n1. Create\n2.
    Display\n3. Add\n4. Modify\n5. Delete\n6. Exit\n";
         cout<<"Enter the choice: ";
         cin>>ch;

      switch(ch)
      {case 1:
         create();
```

```
            break;
        case 2:
           display();
            break;
        case 3:
           add();
            break;
        case 4:
           modify();
            break;
        case 5:
           del();
            break;
        case 6:
        x=0;
         break;
        default:
           cout<<"\nPLease enter correct choice!\n";
            break;
        }

    }

}
void modify() {
     int x,roll;
    cout<<"Enter roll no: ";
    cin>>roll;
    x = search(roll);
    if(x==1) {
         f.open("data.txt",ios::in);
        temp.open("temp.txt",ios::out);
         f>>rollno>>name>>dept;
         while(!f.eof()) {
                if(roll == rollno) {
                        cout<<"Enter name : ";
                        cin>>name;
              cout<<"Enter the department: ";
              cin>>dept;
                        temp<<rollno<<"\t"<<name<<"\t"<<dept<<"\n";
              f>>rollno>>name>>dept;
                        continue;
             }
```

```
            temp<<rollno<<"\t"<<name<<"\t"<<dept<<"\n";
        f>>rollno>>name>>dept;
     }
    f.close(); temp.close();
    f.open("data.txt",ios::out);
    temp.open("temp.txt",ios::in);
    temp>>rollno>>name>>dept;
    while(!temp.eof()) {
            f<<rollno<<"\t"<<name<<"\t"<<dept<<"\n";
            temp>>rollno>>name>>dept;
     }
     f.close();
    temp.close();
     }
     else{
            cout<<"\nRecord not exist!\n";
     }
}

void del()
{
     int x,roll;
     cout<<"Enter roll no to delete: ";
    cin>>roll;
    x = search(roll);
    if(x==1) {
        f.open("data.txt",ios::in);
        temp.open("temp.txt",ios::out);
        f>>rollno>>name>>dept;
        while(!f.eof()){
                if(roll != rollno) {
                        temp<<rollno<<"\t"<<name<<"\t"<<dept<<"\n";
            }
            f>>rollno>>name>>dept;
     }
     f.close();
    temp.close();
     f.open("data.txt",ios::out);
     temp.open("temp.txt",ios::in);
     temp>>rollno>>name>>dept;
     while(!temp.eof()) {
            f<<rollno<<"\t"<<name<<"\t"<<dept<<"\n";
        temp>>rollno>>name>>dept;
```

```
        }

f.close();
temp.close();
}

else {
            cout<<"\nRecord not exist!\n";
    }
}
void add()
{
    int x,i,roll;
    cout<<"Enter roll no: ";
   cin>>roll;
   x = search(roll);
    if(x==1) {
            cout<<"\nRecord already exist!\n";
    }
    else {
    f.open("data.txt",ios::app);
   cout<<"Enter name: ";
   cin>>name;
   cout<<"and department: ";
   cin>>dept;
   rollno = roll;
   f<<rollno<<"\t"<<name<<"\t"<<dept<<"\n";
    f.close();
   }
}
int search(int roll) {
    r.open("data.txt",ios::in);
    while(!r.eof()) {
            r>>rollno>>name>>dept;
      if(roll == rollno) {
                    r.close();
                    return 1;
            }
    }
    r.close();
    return 0;
}
void create()
```

```
{
    int n,i;
    f.open("data.txt",ios::out);
    cout<<"How many records you want to enter:-";
    cin>>n;
    for(i=0;i<n;i++) {
        cout<<"\nEnter roll no: ";
      cin>>rollno;
      cout<<"Enter the name: ";
      cin>>name;
      cout<<"Enter the department: ";
      cin>>dept;
      f<<rollno<<"\t"<<name<<"\t"<<dept<<"\n";
    }
    f.close();
}

void display()
{
    f.open("data.txt",ios::in);
    cout<<"Roll No"<<"\t"<<"Name"<<"\t"<<"Department"<<"\n";
    f>>rollno>>name>>dept;
   while(!f.eof()) {
          cout<<rollno<<"\t"<<name<<"\t"<<dept<<"\n";
      f>>rollno>>name>>dept;
    }
    f.close();
}
```

OUTPUT:

```
Enter your choice what you want to perform:-
1. Create
2. Display
3. Add
4. Modify
5. Delete
6. Exit
Enter the choice: 1
How many records you want to enter:-3
```

Enter roll no: 1
Enter the name: F
Enter the department: IT

Enter roll no: 2
Enter the name: G
Enter the department: COMP

Enter roll no: 3
Enter the name: J
Enter the department: AI

Enter your choice what you want to perform:-
1. Create
2. Display
3. Add
4. Modify
5. Delete
6. Exit
Enter the choice: 2
Roll No Name Department
1 F IT
2 G COMP
3 J AI

Enter your choice what you want to perform:-
1. Create
2. Display
3. Add
4. Modify
5. Delete
6. Exit
Enter the choice: 3
Enter roll no: 6
Enter name: K
and department: MECHANICAL

Enter your choice what you want to perform:-
1. Create
2. Display
3. Add
4. Modify
5. Delete

6. Exit
Enter the choice: 4
Enter roll no: 4

Record not exist!

Enter your choice what you want to perform:-
1. Create
2. Display
3. Add
4. Modify
5. Delete
6. Exit
Enter the choice: 4
Enter roll no: 2
Enter name : D
Enter the department: SCIENCE

Enter your choice what you want to perform:-
1. Create
2. Display
3. Add
4. Modify
5. Delete
6. Exit
Enter the choice: 5
Enter roll no to delete: 1

Enter your choice what you want to perform:-
1. Create
2. Display
3. Add
4. Modify
5. Delete
6. Exit
Enter the choice: 2
Roll No Name Department
2 D SCIENCE
3 J AI
6 K MECHANICAL

Enter your choice what you want to perform:-
1. Create

2. Display
3. Add
4. Modify
5. Delete
6. Exit
Enter the choice: 6
PS D:\C++ GB\Assignments>