G. H. Raisoni College of Engineering & Management, Wagholi, Pune

Department of Artificial Intelligence & Machine Learning

DATA STRUCTURES AND ALGORITHMS LAB (UCAIP204)	
Course Outcome	
CO1	Illustrate various technique to for searching, Sorting and hashing
CO2	Explain the significance of dynamic memory management Techniques
CO3	Design and analyze different linear data structure techniques to solve real world problem.
CO4	Implement non-linear data structure to find solution for given engineering applications.
CO5	Summarize different categories of data Structures

List of Experiment

Sr. No.	Name of Experiment	CO Mapped
1	Consider a student database of SY AI class (at least 10 records). Database contains different fields of every student like Roll No, Name and SGPA.(array of objects of class) 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.	CO1,CO3
2	Department of Artificial Intelligence has student's club named 'SAAI'. Students of Second, third and final year of department can be granted membership on request. Similarly one may cancel the membership of club. First node is reserved for president of club and last node is reserved for secretary of club. Write program to maintain club member's information using singly linked list. Store student MIS registration no. and Name. Write functions to a) Add and delete the members as well as president or even secretary. b) Compute total number of members of club c) Display members d) Display list in reverse order using recursion e) Two linked lists exists for two divisions. Concatenate two lists	CO1, CO3
3	Implement Stack using a linked list. Use this stack to perform evaluation of a postfix expression.	CO3,CO5
4	Queues are frequently used in computer programming, and a typical example is	CO3,CO5

	the creation of a job queue by an operating system. If the operating system does not use priorities, then the jobs are processed in the order they enter the system. Write C++ program for simulating job queue. Write functions to add job and delete job from queue.	
5	A double-ended queue (deque) is a linear list in which additions and deletions may be made at either end. Obtain a data representation mapping a deque into a one-dimensional array. Write C++ program to simulate deque with functions to add and delete elements from either end of the deque.	CO3,CO5
6	A book consists of chapters, chapters consist of sections and sections consist of subsections. Construct a tree and print the nodes. Find the time and space requirements of your method.	CO3,CO4
7	Implement binary tree using linked list and perform recursive traversals.	CO3,CO4
8	Beginning with an empty binary search tree, Construct binary search tree by inserting the values in the order given. After constructing a binary tree i. Insert new node ii. Find number of nodes in longest path iii. Minimum data value found in the tree iv. Change a tree so that the roles of the left and right pointers are swapped at every node v. Search a value	CO3,CO4
9	Implement graph using adjacency list or matrix and perform DFS or BFS.	CO3, CO4
10	You have a business with several offices; you want to lease phone lines to connect them up with each other; and the phone company charges different amounts of money to connect different pairs of cities. You want a set of lines that connects all your offices with a minimum total cost. Solve the problem by suggesting appropriate data structures.	CO3, CO4
Content Beyond Syllabus		
11	A classic problem that can be solved by backtracking is called the Eight Queens problem, which comes from the game of chess. The chess board consists of 64 square arranged in an 8 by 8 grid. The board normally alternates between black and white square, but this is not relevant for the present problem. The queen can move as far as she wants in any direction, as long as she follows a straight line, Vertically, horizontally, or diagonally. Write C++ program for generating all possible configurations for 4-queen's problem	CO1,CO5
12	A Dictionary stores keywords & its meanings. Provide facility for adding new keywords, deleting keywords, updating values of any entry. Provide facility to display whole data sorted in ascending/ Descending order. Also find how many maximum comparisons may require for finding any keyword. Use Binary tree and find the complexity for finding a keyword.	CO3,CO4

Experiment No. 1

Consider a student database of SY COMP class (at least 10 records). Database contains different fields of every student like Roll No, Name and SGPA.(array of objects of class)
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.

OBJECTIVE:
1. To study the concepts of array of structure.
2. To understand the concepts, algorithms and applications of Sorting.
3. To understand the concepts, algorithms and applications of Searching.

OUTCOME:
1. Understand linear data structure - 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:

1. Structure:

Structure is collection of heterogeneous types of data. In C++, a structure collects different data items in such a way that they can be referenced as a single unit. Data items in a structure are called fields or members of the structure.

Creating a Structures:

```
struct student
{
int roll_no;
char name[15];
float sgpa;
};
```

Here struct is keyword used to declare a structure. Student is the name of the structure. In this example, there
are three types of variables int, char and float. The variables have their own names, such as roll_no, name, sgpa.
Structure is represented as follows:

Int roll_no	char name[15]	float sgpa
2 bytes	15 bytes (1 byte for each char)	4 bytes

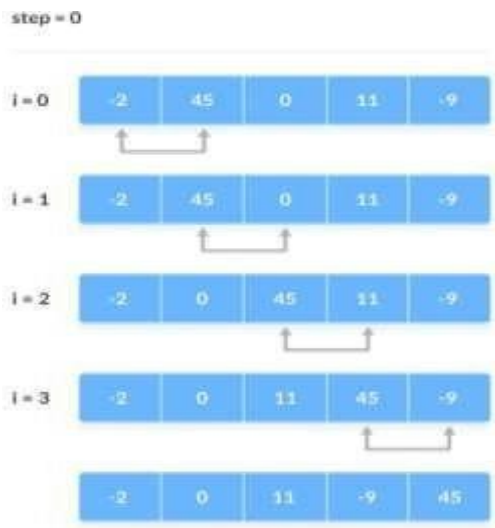
Declaring Structure Variables:
After declaring a structure, you can define the structure variables. For instance, the following structure variables are defined with the structure data type of automobile from the previous section:
struct student s1, s2, s3;
Here three structure variables, s1, s2, s3 are defined by the structure of student. All three structure variables contain the three members of the structure student.
Referencing Structure Members with the Dot Operator:
Given the structure student and s1 is variable of type struct student we can access the fields in following way:
s1.roll_no
s1.name
s1.sgpa
Here the structure name and its member's name are separated by the dot (.) operator.
Pointer to Structure:
We can declare a pointer variable which is capable of storing the address of a structure variable as follows:
struct student *ptr;
Suppose if we have structure variable declared as:
struct student s1;
Then ptr can store the address of s1 by:
ptr = &s1;
Referencing a Structure Member with -> (arrow operator)
Using ptr we can access the fields of s1 as follows:
ptr->name or (*ptr).name
Because of its clearness, the -> operator is more frequently used in programs than the dot operator.
2. Array of Structure:
12
To declare an array of a structures, we first define a structure and then declare an array variable of that type. To declare an 20 element array of structures of type student define earlier, we write,
struct student s[20];
This creates 20 sets of variables that are organized as defined in the structure student.
To access a specific structure, index the array name. For example, to print the name of 4th customer, we write
cout<< s[3].name;
3. Sorting:
It is the process of arranging or ordering information in the ascending or descending order of the key values.
➢ **Bubble Sort**
This is one of the simplest and most popular sorting methods. The basic idea is to pass through the array sequentially several times. In each pass we compare successive pairs of elements (A[i] with A[i+1]) and interchange the two if they are not in the required order. One element is placed in its correct position in each pass. In the first pass, the largest element will sink to the bottom, second largest in the second pass and so on. Thus, a total of n-1 passes are required to sort 'n' keys.

Analysis:

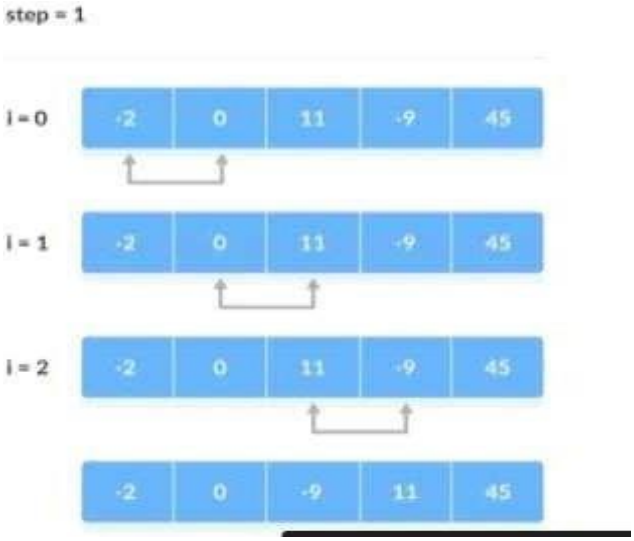
In this sort, n-1 comparisons takes place in 1st pass , n-2 in 2nd pass , n-3 in 3rd pass , and so on.

Therefore total number of comparisons will be
(n-1) + (n-2) + (n-3) + ……. + 3 + 2 + 1
This is an Arithmetic series in decreasing order
Sum = n/2 [2a + (n-1) d]
Where, n = no. of elements in series, a = first element in the series
d = difference between second element and first element.
Sum = (n-1) / 2 [2 * 1 + ((n-1) - 1) * 1]
= n(n-1) / 2
which is of order n^2 i.e $O(n^2)$
The main advantage is simplicity of algorithm and it behaves as O(n) for sorted array of element.
Additional space requirement is only one temporary variable.

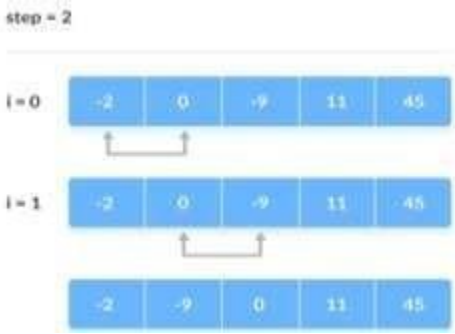
Example:
Compare two consecutive elements, if first element is greater than second then interchange the elements else no change.
Pass 1:



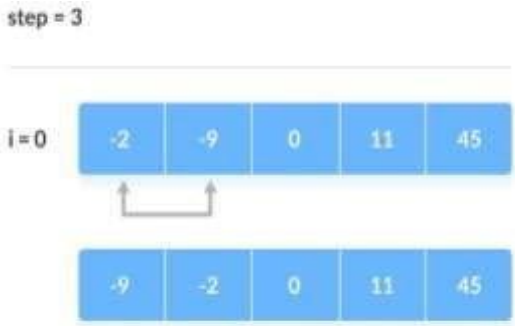
Pass 2:



Pass 3:



Pass 4:



➢ **Insertion Sort**

Insertion sort works similarly as we sort cards in our hand in a card game.

We assume that the first card is already sorted then, we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left. In the same way, other unsorted cards are taken and put at their right place. A similar approach is used by insertion sort.

Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in every iteration.

Analysis:

Worst Case Complexity: $O(n^2)$

Suppose, an array is in ascending order, and you want to sort it in descending order. In this case, worst case complexity occurs.

Each element has to be compared with each of the other elements so, for every nth element, (n-1) number of comparisons are made.

Thus, the total number of comparisons = $n*(n-1) \sim n^2$

Best Case Complexity: O(n)

When the array is already sorted, the outer loop runs for n number of times whereas the inner loop does not run at all. So, there are only n number of comparisons. Thus, complexity is linear.

Average Case Complexity: $O(n^2)$

It occurs when the elements of an array are in jumbled order (neither ascending nor descending).

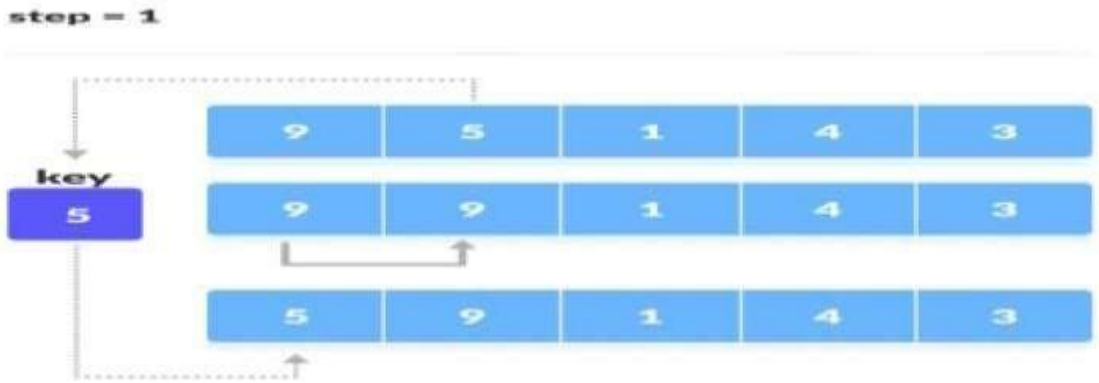
Example:

Suppose we need to sort the following array.

9	5	1	4	3

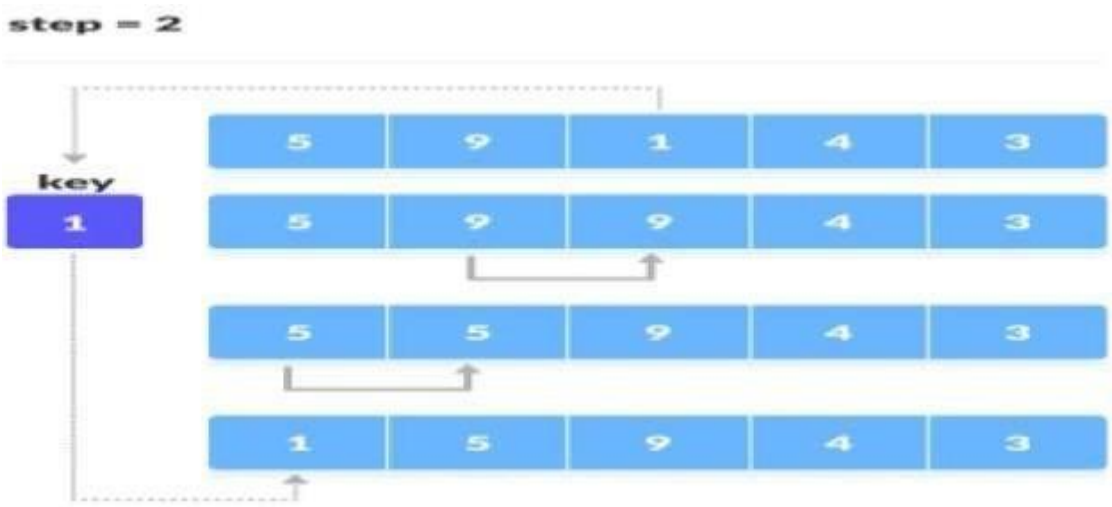
1. The first element in the array is assumed to be sorted. Take the second element and store it separately in key.

Compare key with the first element. If the first element is greater than key, then key is placed in front of the first element.

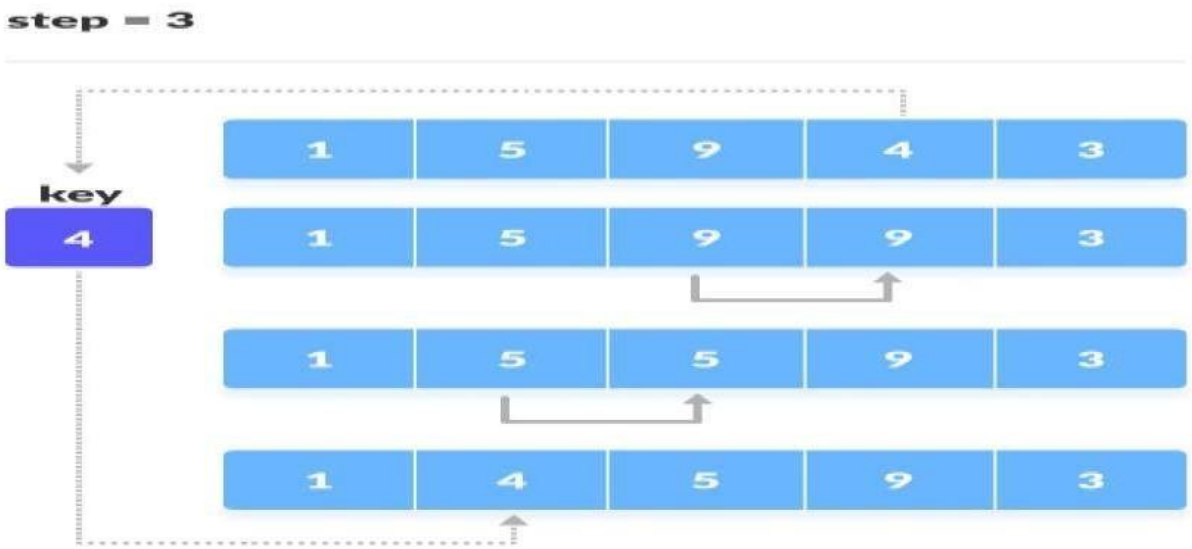


2. Now, the first two elements are sorted.

Take the third element and compare it with the elements on the left of it. Placed it just behind the element smaller than it. If there is no element smaller than it, then place it at the beginning of the array.



3. Similarly, place every unsorted element at its correct position.
Place 4 behind 1



Place 3 behind 1 and the array is sorted:

➢ **Quick Sort**

Quick sort is an algorithm based on divide and conquers approach in which the array is split into sub arrays and these sub-arrays are recursively called to sort the elements.

Analysis:

Worst Case Complexity [Big-O]: $O(n^2)$

It occurs when the pivot element picked is either the greatest or the smallest element. This condition leads to the case in which the pivot element lies in an extreme end of the sorted array. One sub-array is always empty and another sub-array contains n - 1 elements. Thus, quick sort is called only on this sub-array. However, the quick sort algorithm has better performance for scattered pivots.

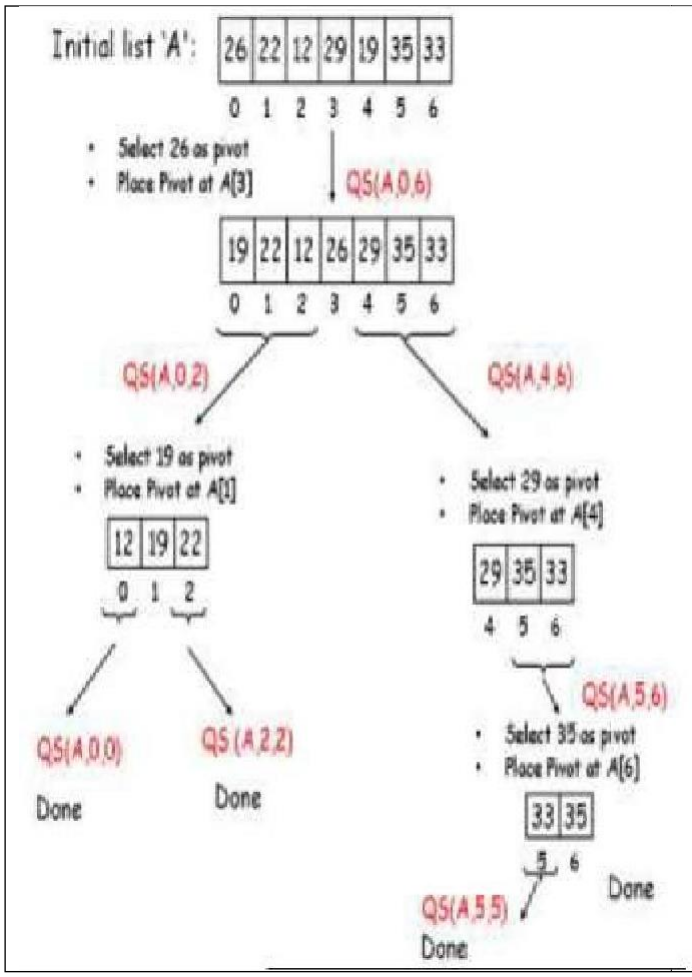
Best Case Complexity [Big-omega]: O(n*log n)

It occurs when the pivot element is always the middle element or near to the middle element.

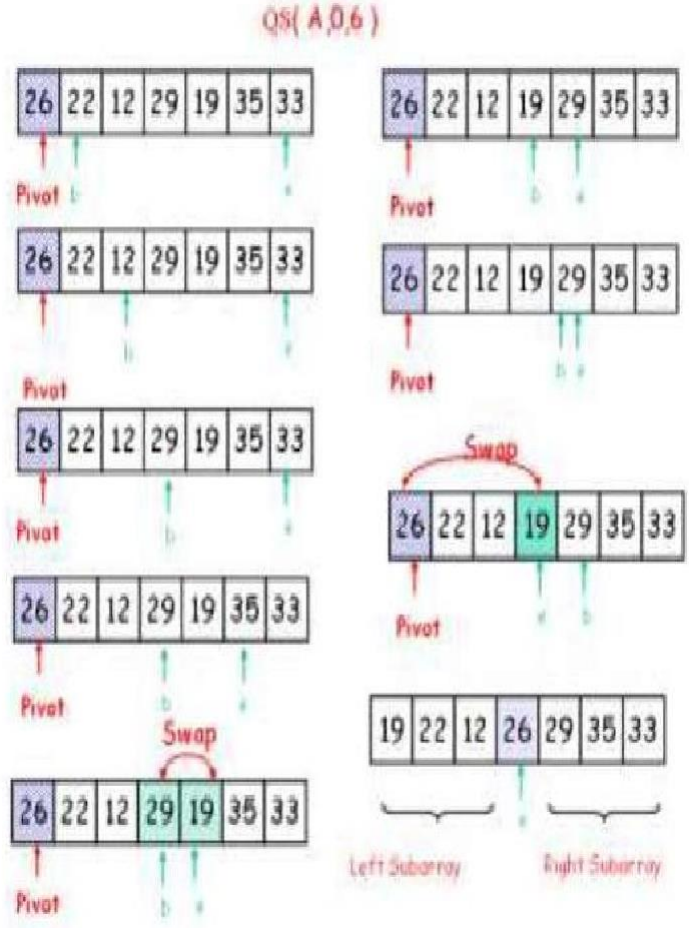
Average Case Complexity [Big-theta]: O(n*log n)

It occurs when the above conditions do not occur.

Example:



Step by Step Process for QS (A, 0, 6)



Searching:

Searching is a process of retrieving or locating a record with a particular key value.

- **Linear Search**

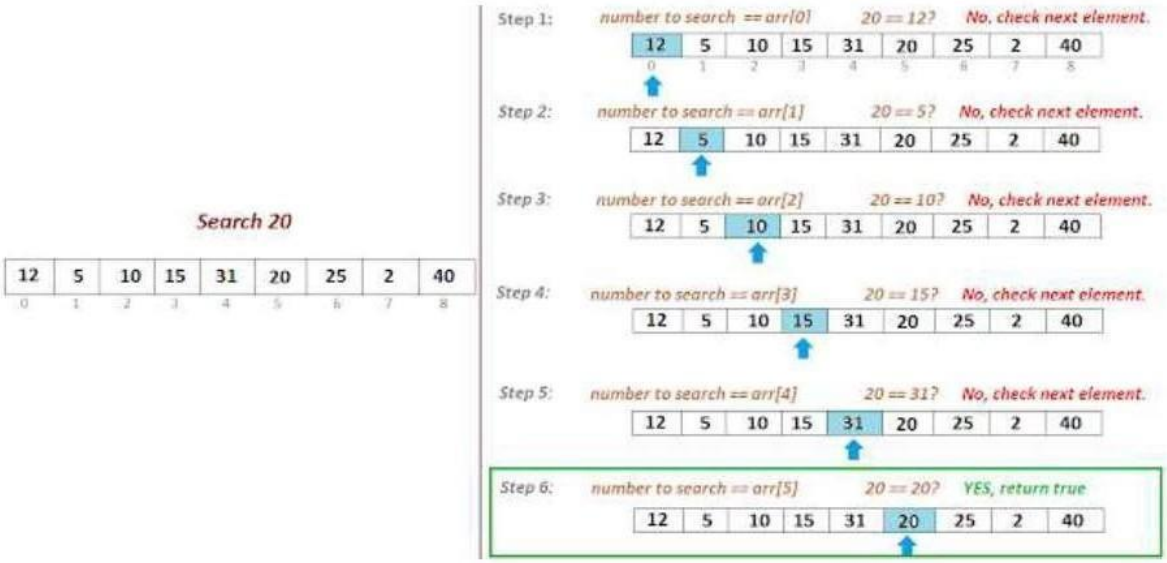
Linear search is the simplest searching algorithm that searches for an element in a list in sequential order. We start at one end and check every element until the desired element is not found.

Analysis:

Time Complexity: O(n)

Space Complexity: O(1)

Example:-



Binary Search

This is a very efficient searching method used for linear / sequential data. In this search, Data has to be in the sorted order either ascending or descending. Sorting has to be done based on the key values.

In this method, the required key is compared with the key of the middle record. If a match is found, the search terminates. If the key is less than the record key, the search proceeds in the left half of the table. If the key is greater than record key, search proceeds in the same way in the right half of the table. The process continues till no more partitions are possible. Thus every time a match is not found, the remaining table size to be searched reduces to half.

If we compare the target with the element that is in the middle of a sorted list, we have three possible results: the target matches, the target is less than the element, or the target is greater than the element. In the first and best case, we are done. In the other two cases, we learn that half of the list can be eliminated from consideration.

When the target is less than the middle element, we know that if the target is in this ordered list, it must be in the list before the middle element. When the target is greater than the middle element, we know that if the target is in this ordered list, it must be in the list after the middle element. These facts allow this one comparison to eliminate one-half of the list from consideration. As the process continues, we

will eliminate from consideration, one-half of what is left of the list with each comparison. This technique is called as binary search.

Analysis:

After 0 comparisons Remaining file size = n

After 1 comparisons Remaining file size = $n / 2 = n / 2^1$ After 2 comparisons Remaining file size = $n / 4 = n / 2^2$ After 3 comparisons Remaining file size = $n / 8 = n / 2^3$

……

After k comparisons Remaining file size = $n / 2^k$

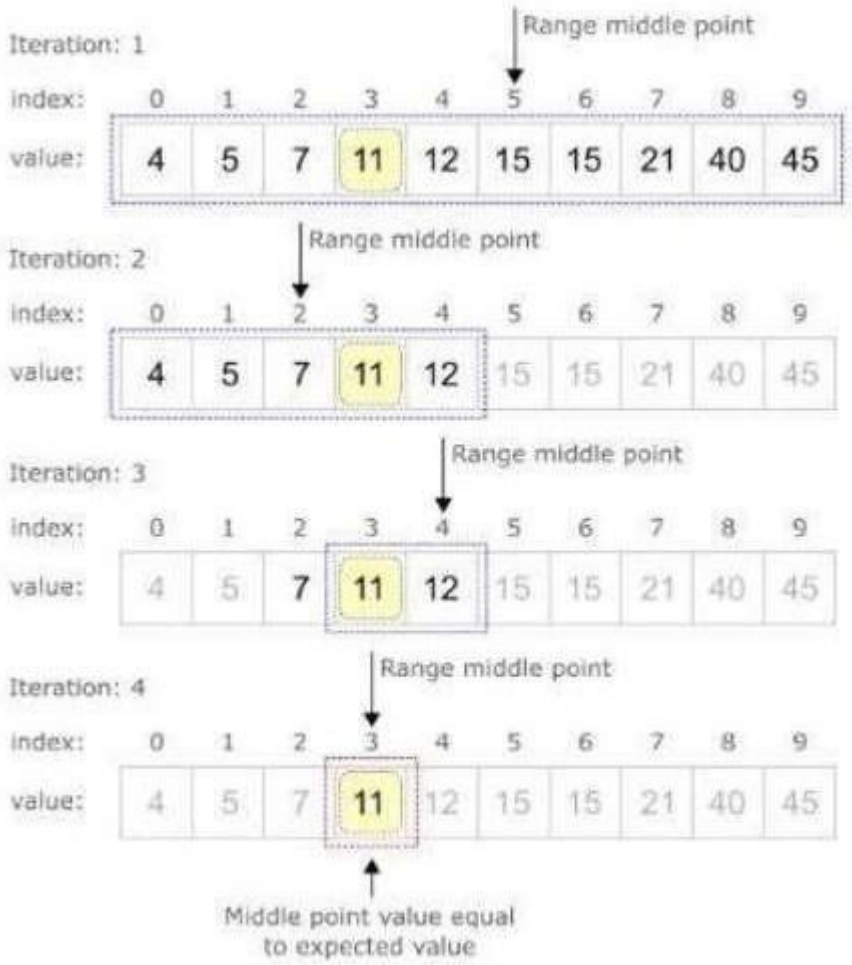
The process terminates when no more partitions are possible i.e. remaining file size = 1

$n / 2^k = 1$

$k = \log_2 n$

Thus, the time complexity is $O(\log_2 n)$. Which is very efficient.

Example:-



ALGORITHM / PSEUDOCODE :

- **Create a structure**

Create_database(struct student s[])
Step 1: Accept how many records user need to add, say, no of records as n
Step 2: For i = 0 to n - 1
i. Accept the record and store it in s[i]
Step 3: End For
Step 4: stop
Display_database(struct student s[] ,int n)
Step 1: For i = 0 to n -1
i. Display the fields s.roll_no, s.name, s.sgpa
Step 2: End For
Step 3: Stop
Bubble Sort student according to sort to roll numbers
BubbleSort(Student s[], n)
Step 1: For Pass = 1 to n-1
Step 2: For i = 0 to (n - pass - 1)
i. If s[i].roll_no < s[i+1].roll_no
20
a. Swap (s[i]. s[i+1])
iii. End if
Step 3: End for
Step 4: End For
Step 5 : Stop

- **Insertion Sort to sort student on the basis of names**

insertion_Sort (Struct student S[], int n)
Step 1: For i = 1 to n-1
i. Set key to s[i]
ii. Set j to i-1
iii. While j>=0 AND strcmp(s[i].name,key.name)>0
a. Assign s[j] to s[j+1]
b. Decrement j
iv. End While
Step 2: Assign key to s[j+1]
Step 3: End for
Step 4: end of insertion sort

- **Quick Sort to sort students on the basis of their sgpa.**

partition (struct student s[], int l, int h)
// where s is the array of structure , l is the index of starting element
// and h is the index of last element.
Step 1: Select s[l].sgpa as the pivot element
Step 2: Set i = l
Step 3: Set j = h-1
Step 4: While i ≤ j
i. Increment i till s[i].sgpa ≤ pivot element
ii. Decrement j till s[j].sgpa > pivot element

iii. If i < j
iv. Swap(s[i], s[j])
v. End if
Step 5: End while
Step 6: Swap(s[j],s[l])
Step 7: return j
Step 8 :end of Partition
quicksort(struct student s[], int l, int h)
//where s is the array of structure , l is the index of starting element
//and h is the index of last element.
Step 1: If l<h
i. P=partition(s,l,h)
ii. quicksort (s,l,p-1)
iii. quicksort (s,p+1,h)
Step 2: End if
Step 3: end of quicksort
21

- **Algorithm for Line ar Search to search students with sgpa given and display all of them**

Linear_search (struct student s[], float key, int n)
//Here s is array of structure student, key is sgpa of student to be searched and
// displayed, n is total number of students in record
Step 1: Set i to 0 and flag to 0
Step 2: While i<n
i. If s[i].sgpa==key
a. Print s[i].roll_no, s[i].name
b. Set flag to 1
c. i++
Step 3: End while
Step 4: If flag==0
i. Print No student found with sgpa=value of key
Step 5: End if
Step 6: End of linear_ sear

- **Algorithm for Binary Search to search students having given string in their names**

Binary_Search (s, n , Key)
// Where s is an array of structure , n is the no of records, and key is element to be searched
Step 1: Set l = 0 & h = n-1
Step 2: While l ≤ h
i. mid = (l + h) / 2
ii. If strcmp (s[mid].name, key)==0)
a. foun
b. stop
iii. Else
a. if (strcmp (key, s[mid].name)<0
i. h = mid - 1
b. Else
ii. l = mid + 1

c. End if
iv. End if
Step 4: End while
Step 5: not found // search is unsuccessful

Validation:

- Limit of the array should not be -ve, and should not cross the lower and upper bound.
- Roll numbers should not repeat, should not -ve
- Name should only contain alphabets, space and .
- Should not allowed any other operations before the input list is entered by the user .
- Before going to (binary search) records should be sorted according a names.

Test Cases :

- **Sorting :**

Four test cases :
22
i. Already Sorted according to the requirement
ii. Sorted in reverse order
iii. Partially sorted
iv. Completely Random List

Expected output /analysis is :
i. Test algorithm for above four test cases
ii. Analyze the algorithms based on no of comparisons and swapping/shifting required
iii. Check for Sort Stability factor
iv. No of passes needed
v. Best /average/ worst case of the each algorithm based on above test case
vi. Memory space required to sort

Searching :

- Find the max and minimum comparison required to search element
- Calculate how many comparisons are required for unsuccessful search

Application:
Useful in managing large amount of data.

FAQ:

- What is the need of sorting
- What is searching?
- How to get output after each pass?
- What do you mean bypass?
- What is recursion?
- Where the values are stored in recursion?
- Explain the notation Ω, θ, O for time analysis.

- Explain the time complexity of bubble sort and linear search, binary search.
- What is need of structure?
- What are the differences between a union and a structure?
- Which operators are used to refer structure member with and without pointer?
- List the advantages and disadvantages of Sequential / Linear Search?
- List the advantages and disadvantages of binary Search?
- What you mean by internal & External Sorting?
- Analyze Quick sort with respect to Time complexity
- In selecting the pivot for QuickSort, which is the best choice for optimal partitioning:

a) The first element of the array
b) The last element of the array
c) The middle element of the array
d) The largest element of the array
e) The median of the array
f) Any of the above?

- Explain the importance of sorting and searching in computer applications?
- Analyze bubble sort with respect to time complexity.
- What is Divide and conquer methodology?
- How the pivot is selected and partition is done in quick sort?
- What is the complexity of quick sort?

Program Code:

```
#include<iostream>
#include<string.h>
using namespace std;
typedef struct student
{
int roll_num;
char name [20];
float marks;
}stud;
void create(stud s[20],int n);
void display(stud s[20],int n);
void bubble_sort(stud s[20],int n);
void insertionSort(stud s[20],int n);
void quick_sort(stud s[20],int,int);
int partition(stud s[20],int,int);
void search(stud s[20],int n,int key);
int bsearch(stud s[20], char x[20],int low,int high);
int main()
{
stud s[20];
int ch,n,key,result;
char x[20];
do
{
cout<<"\n 1) Create Student Database ";
cout<<"\n 2) Display Student Records ";
```

```
cout<<"\n 3) Bubble Sort ";
cout<<"\n 4) Insertion Sort ";
cout<<"\n 5) Quick Sort ";
cout<<"\n 6) Linear search ";
cout<<"\n 7) Binary search ";
cout<<"\n 8) Exit ";
cout<<"\n Enetr Your Choice:=";
cin>>ch;
switch(ch)
{case 1:
cout<<"\n Enter The Number Of Records:=";
cin>>n;
create(s,n);
break;
case 2:
display(s,n);
break;
case 3:
bubble_sort(s,n);
break;
case 4:
insertionSort(s,n);
break;
case 5:
quick_sort(s,0,n-1);
cout<<"\n"<< "\t"<< "Roll No"<< "\t"<<" Name" <<"\t"<< "Marks";
for(int i=n-1; i>=n-10; i--)
```

```
{          cout<<"\n";
cout<<"\t "<< s[i].roll_num<<"\t "<<s[i].name<<"\t "<<s[i].marks;}
break;
case 6:
cout<<"\n Enter the marks which u want to search:=";
cin>>key;
search(s,n,key);
break;
case 7:
cout<<"\n Enter the name of student which u want to search:=";
cin>>x;
      insertionSort(s,n);
      result=bsearch(s,x,0,(n-1));
      if(result==-1)
      {cout<<" \n Student name you want to search for is not present ! \n";}
      else {cout<<" \n The student  is present :\t" << s[result].name;}
      break;
case 8:return 0;
     default:cout<<"\n Invalid choice !! Please enter your choice again."<<endl;}
}while(ch!=8);}
void create(stud s[20],int n)
{int i;
for(i=0;i<n;i++)
{cout<<"\n Enter the roll number:=";
cin>>s[i].roll_num;
cout<<"\n Enter the Name:=";
cin>>s[i].name;
```

```
cout<<"\n Enter the marks:=";
cin>>s[i].marks;
}}
void display(stud s[20],int n)
{int i;
cout<<"\n"<< "\t"<< "Roll No"<< "\t"<<" Name" <<"\t"<< "Marks";
for(i=0;i<n;i++)
{cout<<"\n";
cout<<"\t "<< s[i].roll_num<<"\t "<<s[i].name<<"\t "<<s[i].marks;
}}
//bubble sort to sort in ascending order on roll number
void bubble_sort(stud s[20],int n)
{
int  i,j;
stud temp;
for(i=1;i<n;i++)
{for(j=0;j<n-i;j++)
{if(s[j].roll_num>s[j+1].roll_num)
temp=s[j];
s[j]=s[j+1];
s[j+1]=temp;
}}}}
// insertion sort to sort on names in ascending order
void insertionSort(stud s[20], int n)
{      int i, j;
    stud key;
    for (i = 1; i < n; i++){
```

```
        key= s[i];          j = i - 1;
        /* Move elements of arr[0..i-1], that are
        greater than key, to one position ahead
        of their current position */
        while (j >= 0 && strcmp(s[j].name, key.name) >0)
        {
           s[j + 1]= s[j];
           j = j - 1;
        }
        s[j + 1]= key;
    }
}
 //Quick sort to sort on marks
void quick_sort(stud s[20], int l,int u)
{
    int j;
    if(l<u)
    {
        j=partition(s,l,u);
        quick_sort(s,l,j-1);
        quick_sort(s,j+1,u);
    }
}
 int partition(stud s[20], int l,int u)
{
    int i,j;
            stud temp, v;    v=s[l];    i=l;    j=u+1;
```

```
  do
   {
      do
         i++;

      while(s[i].marks<v.marks&&i<=u);

      do
         j--;
      while(v.marks<s[j].marks);

      if(i<j)
      {
         temp=s[i];
         s[i]=s[j];
         s[j]=temp;
      }
   }while(i<j);

   s[l]=s[j];
   s[j]=v;

   return(j);
}
```

```
// linear search for marks if more than one student having same marks print all of them

void search(stud s[20],int n,int key)
{
int i;

cout<<"\n"<< "\t"<< "Roll No"<< "\t"<<" Name" <<"\t"<< "Marks";
for(i=0;i<n;i++)
{
if(key==s[i].marks)
{
cout<<"\n\t "<< s[i].roll_num<<"\t "<<s[i].name<<"\t "<<s[i].marks;
}}}
int bsearch(stud s[20], char x[20],int low,int high)
{
int mid;
while(low<=high)
{
mid=(low+high)/2;
if(strcmp(x,s[mid].name)==0)
{
return mid;
}
else if(strcmp(x,s[mid].name)<0)
{
high=mid-1;
}
```

```
else
{
low=mid+1;
}}
return -1;}
```

OUTPUT:

```
 1) Create Student Database
 2) Display Student Records
 3) Bubble Sort
 4) Insertion Sort
 5) Quick Sort
 6) Linear search
 7) Binary search
 8) Exit
 Enetr Your Choice:=1
Enter The Number Of Records:=3
 Enter the roll number:=4
 Enter the Name:=dhiraj
 Enter the marks:=78
 Enter the roll number:=1
 Enter the Name:=dhanush
 Enter the marks:=90
 Enter the roll number:=10
 Enter the Name:=samu
 Enter the marks:=99
 1) Create Student Database
 2) Display Student Records
```

3) Bubble Sort

4) Insertion Sort

5) Quick Sort

6) Linear search

7) Binary search

8) Exit

Enter Your Choice:=2

Roll No	Name	Marks
4	dhiraj	78
1	dhanush	90
10	samu	99

1) Create Student Database

2) Display Student Records

3) Bubble Sort

4) Insertion Sort

5) Quick Sort

6) Linear search

7) Binary search

8) Exit

Enter Your Choice:=3

1) Create Student Database

2) Display Student Records

3) Bubble Sort

4) Insertion Sort

5) Quick Sort

6) Linear search

7) Binary search

8) Exit

Enter Your Choice:=2

Roll No Name Marks

1 dhanush 90

4 dhiraj 78

10 samu 99

1) Create Student Database

2) Display Student Records

3) Bubble Sort

4) Insertion Sort

5) Quick Sort

6) Linear search

7) Binary search

8) Exit

Enter Your Choice:=4

1) Create Student Database

2) Display Student Records

3) Bubble Sort

4) Insertion Sort

5) Quick Sort

6) Linear search

7) Binary search

8) Exit

Enter Your Choice:=2

Roll No Name Marks

1 dhanush 90

4 dhiraj 78

10 samu 99

1) Create Student Database

2) Display Student Records

3) Bubble Sort

4) Insertion Sort

5) Quick Sort

6) Linear search

7) Binary search

8) Exit

Enter Your Choice:=2

Roll No Name Marks

1 dhanush 90

4 dhiraj 78

10 samu 99

1) Create Student Database

2) Display Student Records

3) Bubble Sort

4) Insertion Sort

5) Quick Sort

6) Linear search

7) Binary search

8) Exit

Enter Your Choice:=8

Experiment No. 2

Aim: Department of Computer Engineering has student's club named 'COMET'. Students of Second, third and final year of department can be granted membership on request. Similarly one may cancel the membership of club. First node is reserved for president of club and last node is reserved for secretary of club. Write program to maintain club member's information using singly linked list. Store student MIS Registration No. and Name. Write functions to a) Add and delete the members as well as president or even secretary. b) Compute total number of members of club c) Display members d) Display list in reverse order using recursion e) Two linked lists exists for two divisions. Concatenate two lists.

Theory:-

A linked list is a sequence of data structures, which are connected together via links. Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

- **Link** − Each link of a linked list can store a data called an element.
- **Next** − Each link of a linked list contains a link to the next link called Next.
- **LinkedList** − A Linked List contains the connection link to the first link called First.

Linked ListRepresentation

Linked list can be visualized as a chain of nodes, where every node points to the next node.

As per the above illustration, following are the important points to be considered.

- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.

TypesofLinkedList

Following are the various types of linked list.

- **Simple Linked List** − Item navigation is forward only.
- **Doubly Linked List** − Items can be navigated forward and backward.

- **Circular Linked List** − Last item contains link of the first element as next and the first element has a link to the last element as previous.

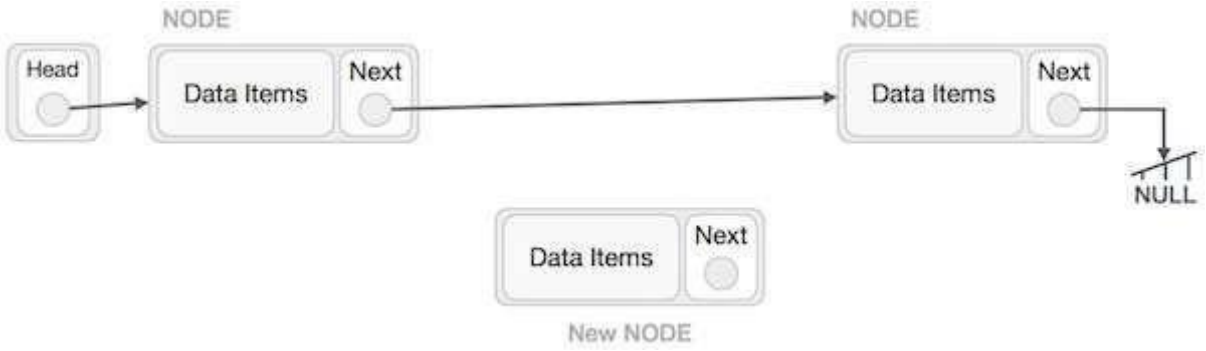
Basic Operations

Following are the basic operations supported by a list.

- **Insertion** − Adds an element at the beginning of the list.
- **Deletion** − Deletes an element at the beginning of the list.
- **Display** − Displays the complete list.
- **Search** − Searches an element using the given key.
- **Delete** − Deletes an element using the given key.

InsertionOperation

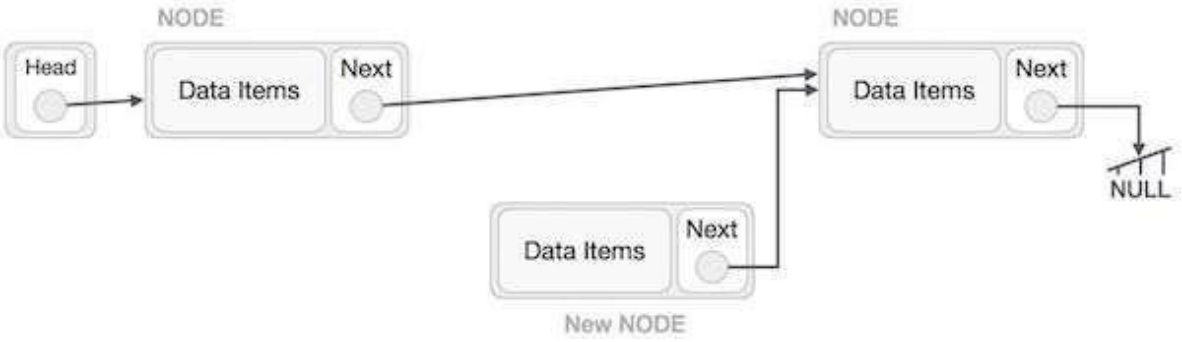
Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.



Imagine that we are inserting a node **B** (NewNode), between **A** (LeftNode) and **C** (RightNode). Then point B.next to C −

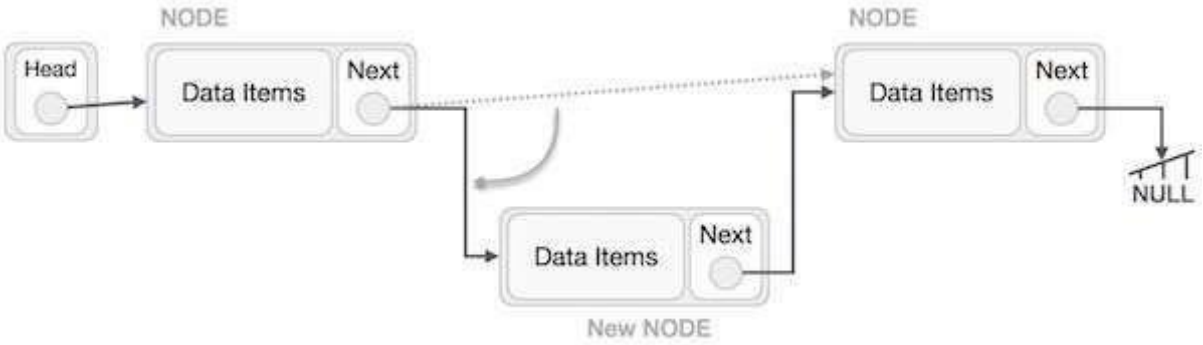
```
NewNode.next −> RightNode;
```

It should look like this −



Now, the next node at the left should point to the new node.

```
LeftNode.next −> NewNode;
```



This will put the new node in the middle of the two. The new list should look like this −

Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.

Deletion Operation

Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.

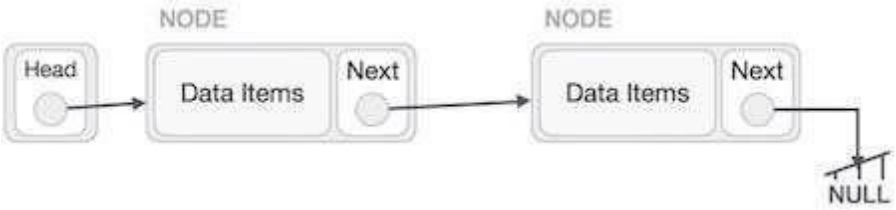
The left (previous) node of the target node now should point to the next node of the target node −

```
LeftNode.next −> TargetNode.next;
```

This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.

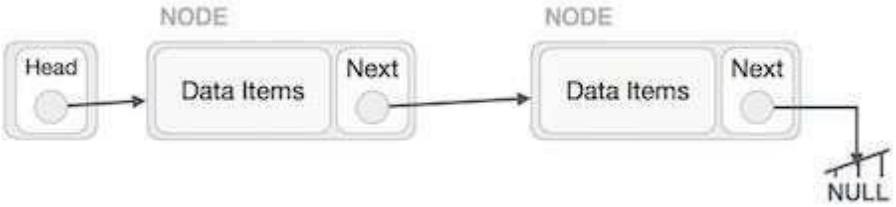
```
TargetNode.next −> NULL;
```

We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.

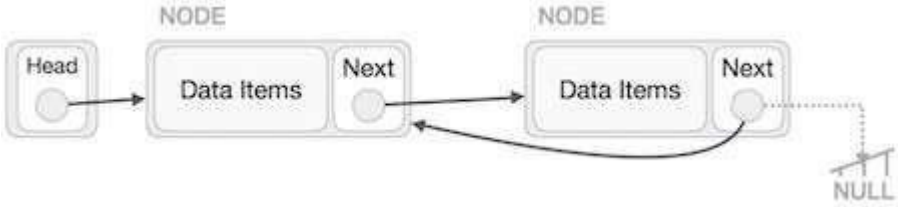


Reverse Operation

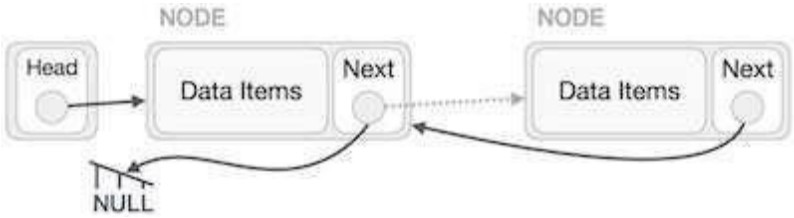
This operation is a thorough one. We need to make the last node to be pointed by the head node and reverse the whole linked list.



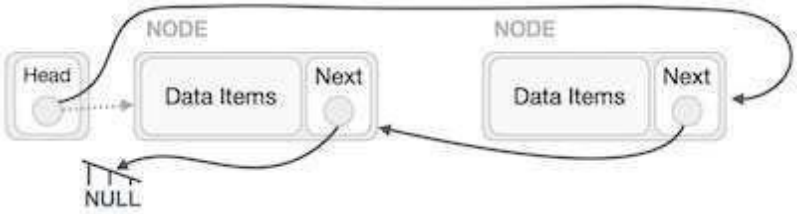
First, we traverse to the end of the list. It should be pointing to NULL. Now, we shall make it point to its previous node −



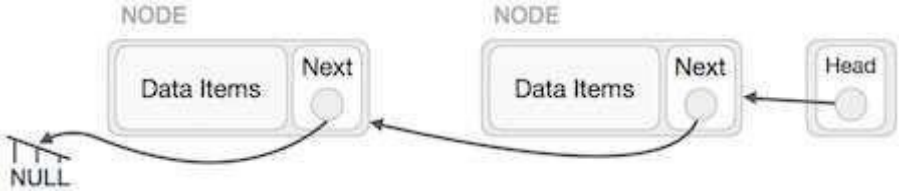
We have to make sure that the last node is not the lost node. So we'll have some temp node, which looks like the head node pointing to the last node. Now, we shall make all left side nodes point to their previous nodes one by one.



Except the node (first node) pointed by the head node, all nodes should point to their predecessor, making them their new successor. The first node will point to NULL.



We'll make the head node point to the new first node by using the temp node.



The linked list is now reversed.

Conclusion- This way we implemented, operations on Singly linked list.

PROGRAM

```
#include<iostream>
#include<string.h>
using namespace std;
struct node
{    int prn,rollno;
char name[50];
struct node *next;
};
class info
{
node
*s=NULL,*head1=NULL,*temp1=NULL,*head2=NULL,*temp2=NULL,*head=NULL,*temp=NULL;
int b,c,i,j,ct;
```

```
char a[20];
  public:
node *create()
void insertp();
void insertm()
 void delm();
  void delp();

void dels();

        void display();

        void count();

        void reverse();

        void rev(node *p);

        void concat();
        } ;
```

```
node *info::create()
{ node *p=new(struct node);
    cout<<"enter name of student ";
    cin>>a;
    strcpy(p->name,a);
      cout<<"\n enter prn no. of student \n";
    cin>>b;
    p->prn=b;
  cout<<"enter student rollno";
cin>>c;
    p->rollno=c;
    p->next=NULL;
    return  p;
  }
  void info::insertm()
  {
      node *p=create();
```

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

}
  void info::insertp()
{
    node *p=create();

  if(head==NULL)
```

```
 {    head=p;
 }
else
{      temp=head;
       head=p;
         head->next=temp->next;

 }

}

void info::display()
{         if(head==NULL)
          {    cout<<"linklist is empty";
           }
         else
         {
```

```
            temp=head;
                    cout<<"     prn     rolln0    NAME \n";
                    while(temp->next!=NULL)
                    {     cout<<"    \n"<<temp->prn<<"    "<<temp->rollno<<"    "<<temp->name;
                         temp=temp->next;
                    }
                   cout<<"    "<<temp->prn<<"    "<<temp->rollno<<"    "<<temp->name;
            }
}
void info::delm()
{ int m,f=0;
   cout<<"\n enter the prn no. of student whose data you want to delete";
    cin>>m;
    temp=head;
    while(temp->next!=NULL)
    {
        if(temp->prn==m)
```

```
        {           s->next=temp->next;
                    delete(temp);       f=1;
        }
        s=temp;
        temp=temp->next;
    }     if(f==0)
         { cout<<"\n sorry memeber not deleted ";  }
}
void info::delp()
{    temp=head;
   head=head->next;
      delete(temp);
   }
   void info::dels()
{
   temp=head;
   while(temp->next!=NULL)
```

```
  { s=temp;
  temp=temp->next;
  }     s->next=NULL;
     delete(temp);


}
void info::count()
{      temp=head;    ct=0;
       while(temp->next!=NULL)
       {    temp=temp->next; ct++; }
         ct++;
         cout<<" Count of members is:"<<ct;


  }
  void info::reverse()
  {     rev(head);  }
  void info::rev(node *temp)
```

```
{     if(temp==NULL)
      { return; }
      else
      { rev(temp->next); }
      cout<<"    "<<temp->prn<<"    "<<temp->rollno<<"    "<<temp->name;
}
void info::concat()
{  int k,j;
   cout<<"enter no. of members in list1";
   cin>>k;
    head=NULL;
   for(i=0;i<k;i++)
   { insertm();
     head1=head;

   } head=NULL;
  cout<<"enter no. of members in list2";
```

```
cin>>j;
for(i=0;i<j;i++)
{ insertm();
  head2=head;

} head=NULL;

 temp1=head1;
while(temp1->next!=NULL)
{ temp1=temp1->next; }
    temp1->next=head2;

                temp2=head1;
               cout<<"     prn     rolln0    NAME \n";
              while(temp2->next!=NULL)
              {
```

```
                cout<<"\n    "<<temp2->prn<<"    "<<temp2->rollno<<"      "<<temp2->name<<"\n";;
                temp2=temp2->next;
              }
                cout<<"\n    "<<temp2->prn<<"    "<<temp2->rollno<<"      "<<temp2->name<<"\n";
  }
int main()
{ info s;
int i;

    char ch;
   do{
    cout<<"\n choice the options";
    cout<<"\n 1. To insert president ";
    cout<<"\n 2. To insert member ";
    cout<<"\n 3. To insert secretary ";
    cout<<"\n 4. To delete president ";
    cout<<"\n 5. To delete member  ";
```

```
 cout<<"\n 6. To delete secretary ";
 cout<<"\n 7. To display data ";
 cout<<"\n 8. Count of members";
 cout<<"\n 9. To display reverse of string ";
 cout<<"\n 10.To concatenate two strings ";
 cin>>i;
switch(i)
{       case 1:   s.insertp();
                  break;
        case 2:   s.insertm();
                  break;
        case 3:   s.insertm();
                  break;
        case 4:   s.delp();
                  break;
        case 5:   s.delm();
                  break;
```

```
            case 6:  s.dels();
                        break;
            case 7: s.display();
                        break;
            case 8: s.count();
                        break;
            case 9:   s.reverse();
                        break;
            case 10:  s.concat();


                        break;
            default: cout<<"\n unknown choice";
      }
        cout<<"\n do you want to continue enter y/Y \n";
        cin>>ch;

   }while(ch=='y'||ch=='Y');
```

```
        return 0;

}
```

Output:

```
 choice the options
  1. To insert president
  2. To insert member
  3. To insert secretary
  4. To delete president
  5. To delete member
  6. To delete secretary
  7. To display data
  8. Count of members
  9. To display reverse of string
  10.To concatenate two strings 9

 do you want to continue enter y/Y
y

 choice the options
  1. To insert president
  2. To insert member
  3. To insert secretary
  4. To delete president
  5. To delete member
  6. To delete secretary
  7. To display data
  8. Count of members
  9. To display reverse of string
10.To concatenate two strings 1
enter name of student asha

  enter prn no. of student
1234
enter student rollno43

 do you want to continue enter y/Y
y

 choice the options
  1. To insert president
```

2. To insert member
3. To insert secretary
4. To delete president
5. To delete member
6. To delete secretary
7. To display data
8. Count of members
9. To display reverse of string
10.To concatenate two strings 2
enter name of student hina

enter prn no. of student
234
enter student rollno8

do you want to continue enter y/Y
y

choice the options
1. To insert president
2. To insert member
3. To insert secretary
4. To delete president
5. To delete member
6. To delete secretary
7. To display data
8. Count of members
9. To display reverse of string
10.To concatenate two strings 3
enter name of student hari

enter prn no. of student
4444
enter student rollno23

do you want to continue enter y/Y
y

choice the options
1. To insert president
2. To insert member
3. To insert secretary
4. To delete president
5. To delete member
6. To delete secretary
7. To display data

8. Count of members
9. To display reverse of string
10.To concatenate two strings 7
prn rolln0 NAME

1234 43 asha
234 8 hina 4444 23 hari
do you want to continue enter y/Y
y

choice the options
1. To insert president
2. To insert member
3. To insert secretary
4. To delete president
5. To delete member
6. To delete secretary
7. To display data
8. Count of members
9. To display reverse of string
10.To concatenate two strings 8
Count of members is:3
do you want to continue enter y/Y
y

choice the options
1. To insert president
2. To insert member
3. To insert secretary
4. To delete president
5. To delete member
6. To delete secretary
7. To display data
8. Count of members
9. To display reverse of string
10.To concatenate two strings 10
enter no. of members in list11
enter name of student wama

enter prn no. of student
23456
enter student rollno22
enter no. of members in list21
enter name of student qqq

enter prn no. of student

```
2223
enter student rollno34
     prn      rolln0    NAME

    23456    22      wama

    2223    34      qqq

 do you want to continue enter y/Y
y

 choice the options
  1. To insert president
  2. To insert member
  3. To insert secretary
  4. To delete president
  5. To delete member
  6. To delete secretary
  7. To display data
  8. Count of members
  9. To display reverse of string
  10.To concatenate two strings 9

 do you want to continue enter y/Y
y

 choice the options
  1. To insert president
  2. To insert member
  3. To insert secretary
  4. To delete president
  5. To delete member
  6. To delete secretary
  7. To display data
  8. Count of members
  9. To display reverse of string
10.To concatenate two strings 7
linklist is empty
 do you want to continue enter y/Y
y
```

Experiment No. 3

Aim: Implement Stack using a linked list. Use this stack to perform evaluation of a postfix expression

OBJECTIVE:

1) To understand the concept of abstract data type.
2) How different data structures such as arrays and a stacks are represented as an ADT.

Theory:-

A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named

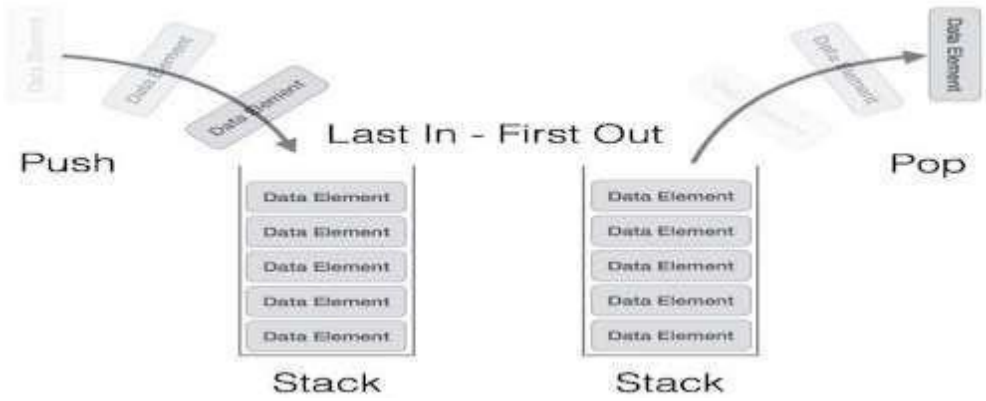
stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation iscalled **PUSH** operation and removal operation is called **POP** operation.

Stack Representation

The following diagram depicts a stack and its operations −



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

- **push()** − Pushing (storing) an element on the stack.
- **pop()** − Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

- **peek()** − get the top data element of the stack, without removing it.
- **isFull()** − check if stack is full.
- **isEmpty()** − check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top**pointer provides top value of the stack without actually removing it.

First we should learn about procedures to support stack functions −

```
bool isfull() {
   if(top == MAXSIZE)
      return true;
   else
      return false;
}
```

Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty. Here's the code −

```
bool isempty() {
   if(top == -1)
      return true;
   else
      return false;
}
```

Push Operation

The process of putting a new data element onto stack is known as a Push Operation.

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

```
}
```

Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

```
int pop(int data) {

  if(!isempty()) {

    data = stack[top];

    top = top - 1;

    return data;

  } else {

    printf("Could not retrieve data, Stack is empty.\n");

  }

}
```

Program :

```
#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include<ctype.h>

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

struct node *top = NULL;

/* create a new node with the given data */
struct node* createNode(int data) {
    struct node *ptr = (struct node *) malloc(sizeof (struct node));
    ptr->data = data;
    ptr->next = NULL;
}
```

```
/* push the input data into the stack */
void push (int data) {
    struct node *ptr = createNode(data);
    if (top == NULL) {
        top = ptr;
        return;
    }
    ptr->next = top;
    top = ptr;
}

/* pop the top element from the stack */
int pop () {
    int data;
    struct node *temp;
    if (top == NULL)
        return -1;
    data = top->data;
    temp = top;
    top = top->next;
    free(temp);
    return (data);
}

int main() {
    char str[100];
    50nti , data = -1, operand1, operand2, result;
    /* i/p postfix expr from the user */
    cout<<"Enter ur postfix expression:";
    fgets(str, 100, stdin);
    for (i = 0; i < strlen(str); i++) {
        if (isdigit(str[i])) {
            /*
             * if the i/p char is digit, parse
             * character by character to get
             * complete operand
             */
            data = (data == -1) ? 0 : data;
```

```
        data = (data * 10) + (str[i] – 48);
        continue;
}

/* push the operator into the stack */
if (data != -1) {
        push(data);
}

if (str[i] == '+' || str[i] == '-'
        || str[i] == '*' || str[i] == '/') {
        /*
         * if the i/p character is an operator,
         * then pop two elements from the stack,
         * apply operator and push the result into
         * the stack
         */
        operand2 = pop();
        operand1 = pop();
        if (operand1 == -1 || operand2 == -1)
                break;
        switch (str[i]) {
                case '+':
                        result = operand1 + operand2;
                        /* pushing result into the stack */
                        push(result);
                        break;
                case '-':
                        result = operand1 – operand2;
                        push(result);
                        break;
                case '*':
                        result = operand1 * operand2;
                        push(result);
                        break;
                case '/':
                        result = operand1 / operand2;
                        push(result);
```

```
                        break;
                    }
                }
                data = -1;
            }
            if (top != NULL && top->next == NULL)
                cout<<“Output:%d\n”, top->data;
            else
                cout<<“u ve entered wrong expression\n”;
            return 0;
}
```

OUTPUT

```
Enter ur postfix expression:10 20 * 30 40 10 / - +
Output:226
```

Experiment No. 4

Aim: Queues are frequently used in computer programming, and a typical example is the creation of a job queue by an operating system. If the operating system does not use priorities, then the jobs are processed in the order they enter the system. Write C++ program for simulating job queue. Write functions to add job and delete job from queue.

Pre-requisite:

- Basics of Queue
- Different operations that can be performed on queue

Objective:

- To perform addition and deletion operations on queue.

Input:

Size of queue Elements in queue

Outcome:

- Result of addition of job operation on queue.
- Result of deletion of job operation on queue.

Theory :

- Write theory of queue (definition, concepts, types, advantages, disadvantages)
- Explain queue as an ADT. (write pseudo code)

Algorithms :

Write your own algorithms

Flowchart :

Draw flowchart for above algorithms

Conclusion:

By this way, we can perform different operations on queue

Question Bank:

1. What is Queue?

2. What are the different operations that can be performed on queue?
3. Explain all the operations on queue
4. Which are different types of queues , Explain.

Program :

```
#include <iostream>
#define MAX 10
using namespace std;
struct queue
{       int data[MAX];
 int front,rear;
};
class Queue
{    struct queue q;
  public:
     Queue(){q.front=q.rear=-1;}
     int isempty();
     int isfull();
     void enqueue(int);
     int delqueue();
     void display();
};
int Queue::isempty()
{
 return(q.front==q.rear)?1:0;
}
int Queue::isfull()
{    return(q.rear==MAX-1)?1:0;}
void Queue::enqueue(int x)
{q.data[++q.rear]=x;}
int Queue::delqueue()
{return q.data[++q.front];}
void Queue::display()
{ int i;
   cout<<"\n";
   for(i=q.front+1;i<=q.rear;i++)
     cout<<q.data[i]<<" ";
```

```
}
int main()
{      Queue obj;
 int ch,x;
 do{    cout<<"\n 1. insert job\n 2.delete job\n 3.display\n 4.Exit\n Enter your choice:";
        cin>>ch;
 switch(ch)
 { case 1: if (!obj.isfull())
              { cout<<"\n Enter data:";
                    cin>>x;
                    obj.enqueue(x);
              }
           else
                cout<< "Queue is overflow";
            break;
   case 2: if(!obj.isempty())
                      cout<<"\n Deleted Element="<<obj.delqueue();
              else
                    { cout<<"\n Queue is underflow";  }
              cout<<"\nremaining jobs :";
              obj.display();
            break;
   case 3: if (!obj.isempty())
        { cout<<"\n Queue contains:";
                obj.display();
        }
        else
                  cout<<"\n Queue is empty";
        break;

   case 4: cout<<"\n Exit";
        }
      }while(ch!=4);
return 0;
}
```

OUTPUT

```
1. insert job
2.delete job
3.display
```

```
4.Exit
Enter your choice:1
Enter data:34

1. insert job
2.delete job
3.display
4.Exit
Enter your choice:1
Enter data:64

1. insert job
2.delete job
3.display
4.Exit
Enter your choice:1
Enter data:84

1. insert job
2.delete job
3.display
4.Exit
Enter your choice:1
Enter data:93

1. insert job
2.delete job
3.display
4.Exit
Enter your choice:3
Queue contains:
34 64 84 93

1. insert job
2.delete job
3.display
4.Exit
Enter your choice:2
Deleted Element=34
remaining jobs :
64 84 93

1. insert job
2.delete job
3.display
4.Exit
```

```
Enter your choice:3
Queue contains:
64 84 93

1. insert job
2.delete job
3.display
4.Exit
Enter your choice:4
Exit
```

Experiment No.5

Aim: A double-ended queue(deque) is a linear list in which additions and deletions may be made at either end. Obtain a data representation mapping a deque into a one-dimensional array. Write C++ program to simulate deque with functions to add and delete elements from either end of the deque.

Pre-requisite:

- **Knowledge of Queue**
- **Types of queue**
- **Knowledge of double ended queue and different operations that can be performed on it**

Objective:

- To simulate deque with functions to add and delete elements from either end of the deque.

Input:

Size of array Elements in the queue

Outcome:

- Result of deque with functions to add and delete elements from either end of the deque.

Theory :

Double-Ended Queue

A double-ended queue is an abstract data type similar to an simple queue, it allows you to insert and delete from both sides means items can be added or deleted from the front or rear end.

Algorithm for Insertion at rear end

Step -1: [Check for overflow] if(rear==MAX) Print("Queue is Overflow"); return;
Step-2: [Insert element] else
rear=rear+1;
q[rear]=no;
[Set rear and front pointer]

if rear=0
rear=1; if front=0
front=1; Step-3: return

Implementation of Insertion at rear end

```
void add_item_rear()
{
int num;
printf("\n Enter Item to insert : "); scanf("%d",&num); if(rear==MAX)
{
printf("\n Queue is Overflow"); return;
}
else
{
rear++; q[rear]=num; if(rear==0) rear=1; if(front==0) front=1;
}
}
```

Algorithm for Insertion at font end

Step-1 : [Check for the front position] if(front<=1)
Print (“Cannot add item at front end”); return;
Step-2 : [Insert at front] else
front=front-1; q[front]=no; Step-3 : Return

Implementation of Insertion at font end

```
void add_item_front()
{
int num;
printf("\n Enter item to insert:"); scanf("%d",&num);

if(front<=1)

{
printf("\n Cannot add item at front end"); return;
}
else
{
front--; q[front]=num;
}
```

}

Algorithm for Deletion from front end

Step-1 [Check for front pointer] if front=0
print(" Queue is Underflow"); return;
Step-2 [Perform deletion] else
no=q[front];
print("Deleted element is",no); [Set front and rear pointer]
if front=rear front=0; rear=0;
else front=front+1; Step-3 : Return

Implementation of Deletion from front end

```
void delete_item_front()
{
int num; if(front==0)
{
printf("\n Queue is Underflow\n"); return;
}
else
{
num=q[front];
printf("\n Deleted item is %d\n",num); if(front==rear)
{
front=0; rear=0;
}
else
{
front++;

}

}
}
```

Algorithm for Deletion from rear end

Step-1 : [Check for the rear pointer] if rear=0
print("Cannot delete value at rear end"); return;

Step-2: [perform deletion] else
no=q[rear];
[Check for the front and rear pointer] if front= rear
front=0; rear=0; else
rear=rear-1;
print(“Deleted element is”,no); Step-3 : Return

Implementation of Deletion from rear end

```
void delete_item_rear()
{
int num; if(rear==0)
{
printf("\n Cannot delete item at rear end\n"); return;
}
else
{
num=q[rear]; if(front==rear)
{
front=0; rear=0;
}
else
{
rear--;
printf("\n Deleted item is %d\n",num);
}
}
}
```

Conclusion:
By this way, we can perform operations on double ended queue

Program :

```
#include<iostream>
//#include
//#include
using namespace std;
#define SIZE 5

class dequeue
{

 int a[10],front,rear,count;

public:
 dequeue();
 void add_at_beg(int);
 void add_at_end(int);
 void delete_fr_front();
 void delete_fr_rear();
 void display();
};
dequeue::dequeue()
{
 front=-1;
 rear=-1;
 count=0;
}
void dequeue::add_at_beg(int item)
{
 int i;
 if(front==-1)
 {
        front++;
        rear++;
        a[rear]=item;
        count++;
 }
 else if(rear>=SIZE-1)
 {
```

```
        cout<<"\nInsertion is not possible,overflow!!!!";
 }
 else
 {
        for(i=count;i>=0;i--)
        {
                a[i]=a[i-1];
        }
        a[i]=item;
        count++;
        rear++;
 }
}
void dequeue::add_at_end(int item)
{

 if(front==-1)
 {
        front++;
        rear++;
        a[rear]=item;
        count++;
 }
 else if(rear>=SIZE-1)
 {
        cout<<"\nInsertion is not possible,overflow!!!";
        return;
 }
 else
 {
        a[++rear]=item;
 }}


void dequeue::display()
{

 for(int i=front;i<=rear;i++)
 {
```

```
        cout<<a[i]<<" ";        }
}

void dequeue::delete_fr_front()
{
 if(front==-1)
 {
        cout<<"Deletion is not possible:: Dequeue is empty";
        return;
 }
 else
 {
        if(front==rear)
        {
                front=rear=-1;
                return;
        }
        cout<<"The deleted element is "<<a[front];
        front=front+1;
 }}

void dequeue::delete_fr_rear()
{
 if(front==-1)
 {
        cout<<"Deletion is not possible:Dequeue is empty";
        return;
 }
 else
 {
        if(front==rear)
        {
                front=rear=-1;
        }
        cout<<"The deleted element is "<< a[rear];
        rear=rear-1;
 }}
```

```
int main()
{
 int c,item;
 dequeue d1;

 do
 {
        cout<<"\n\n****DEQUEUE OPERATION****\n";
        cout<<"\n1-Insert at beginning";
        cout<<"\n2-Insert at end";
        cout<<"\n3_Display";
        cout<<"\n4_Deletion from front";
        cout<<"\n5-Deletion from rear";
        cout<<"\n6_Exit";
        cout<<"\nEnter your choice<1-4>:";
        cin>>c;

        switch(c)
        {
        case 1:
                cout<<"Enter the element to be inserted:";
                cin>>item;
                d1.add_at_beg(item);
                break;

        case 2:
                cout<<"Enter the element to be inserted:";
                cin>>item;
                d1.add_at_end(item);
                break;

        case 3:
                d1.display();
                break;

        case 4:
                d1.delete_fr_front();
                break;
```

```
            case 5:
                    d1.delete_fr_rear();
                    break;

            case 6:
                    exit(1);
                    break;

            default:
                    cout<<"Invalid choice";
                    break;
            }

 }while(c!=7);
 return 0;}
```

OUTPUT

```
****DEQUEUE OPERATION****

1-Insert at beginning
2-Insert at end
3_Display
4_Deletion from front
5-Deletion from rear
6_Exit
Enter your choice<1-4>:1
Enter the element to be inserted:45


****DEQUEUE OPERATION****

1-Insert at beginning
2-Insert at end
3_Display
4_Deletion from front
5-Deletion from rear
6_Exit
Enter your choice<1-4>:2
```

Enter the element to be inserted:46

****DEQUEUE OPERATION****

1-Insert at beginning
2-Insert at end
3_Display
4_Deletion from front
5-Deletion from rear
6_Exit
Enter your choice<1-4>:3
45 46

****DEQUEUE OPERATION****

1-Insert at beginning
2-Insert at end
3_Display
4_Deletion from front
5-Deletion from rear
6_Exit
Enter your choice<1-4>:4
The deleted element is 45

****DEQUEUE OPERATION****

1-Insert at beginning
2-Insert at end
3_Display
4_Deletion from front
5-Deletion from rear
6_Exit
Enter your choice<1-4>:3
46

****DEQUEUE OPERATION****

1-Insert at beginning
2-Insert at end
3_Display
4_Deletion from front
5-Deletion from rear
6_Exit
Enter your choice<1-4>:5
The deleted element is 0

****DEQUEUE OPERATION****

1-Insert at beginning
2-Insert at end
3_Display
4_Deletion from front
5-Deletion from rear
6_Exit
Enter your choice<1-4>:3

****DEQUEUE OPERATION****

1-Insert at beginning
2-Insert at end
3_Display
4_Deletion from front
5-Deletion from rear
6_Exit
Enter your choice<1-4>:

Experiment No 6

Aim: A book consists of chapters, chapters consist of sections and sections consist of subsections. Construct a tree and print the nodes. Find the time and space requirements of your method.

Theory:

Introduction to Tree:

Definition:

A tree T is a set of nodes storing elements such that the nodes have a parent-child relationship that satisfies the following

• if T is not empty, T has a special tree called the root that has no parent

• each node v of T different than the root has a unique parent node w; each node with parent w is a child of w

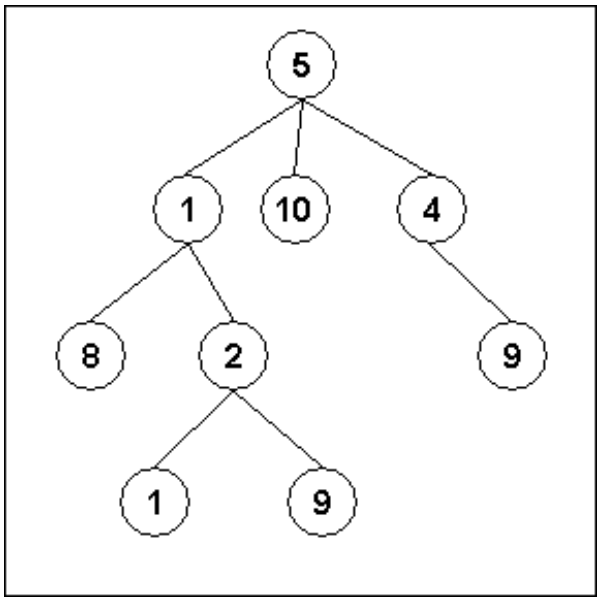


Fig1. An example of a tree

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

There are two basic types of trees. In an unordered tree, a tree is a tree in a purely structural sense — that is to say, given a node, there is no order for the children of that node. A tree on which an order is imposed — for example, by assigning different natural numbers to eachchild of each node — is called an ordered tree, and data structures built on them are called ordered tree data structures. Ordered trees are by far the most common form of tree data

structure. Binary search trees are one kind of ordered tree.

Important Terms

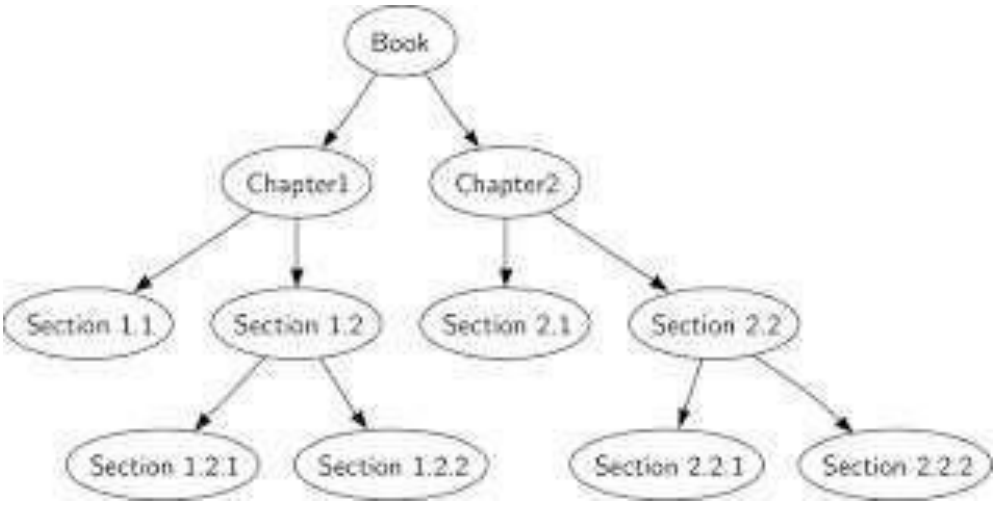
Following are the important terms with respect to tree.

- **Path** − Path refers to the sequence of nodes along the edges of a tree.
- **Root** − The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent** − Any node except the root node has one edge upward to a node called parent.
- **Child** − The node below a given node connected by its edge downward is called its child node.
- **Leaf** − The node which does not have any child node is called the leaf node.
- **Subtree** − Subtree represents the descendants of a node.
- **Visiting** − Visiting refers to checking the value of a node when control is on the node.
- **Traversing** − Traversing means passing through nodes in a specific order.
- **Levels** − Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- **keys** − Key represents a value of a node based on which a search operation is to be carried out for a node.

Advantages of trees

Trees are so useful and frequently used, because they have some very serious advantages:

- Trees reflect structural relationships in the data
- Trees are used to represent hierarchies
- Trees provide an efficient insertion and searching
- Trees are very flexible data, allowing to move subtrees around with minumum effort
  For this assignment we are considering the tree as follows.



Recursive definition

• T is either empty

• or consists of a node r (the root) and a possibly empty set of trees whose roots are the children of r

Tree is a widely-used data structure that emulates a tree structure with a set of linked nodes. The tree graphically is represented most commonly as on *Picture 1*. The circles are the nodes and the edges are the links between the trees are usually used to store and represent data in some hierarchical order. The data are stored in the nodes, from which the tree is consisted of.

A node may contain a value or a condition or represent a separate data structure or a tree of its own. Each node in a tree has zero or more child nodes, which are one level lower in the tree hierarchy (by convention, trees grow down, not up as they do in nature). A node that has a child is called the child's parent node (or ancestor node, or superior). A node has at most one parent. A node that has no child's is called a leaf, and that node is of course at the bottommost level of the tree. The height of a node is the length of the longest path to a leaf from that node. The height of the root is the height of the tree. In other words, the "height" of tree is the "number of levels" in the tree. or more formerly, the height of a tree is defined as follows:

1. The height of a tree with no elements is 0
2. The height of a tree with 1 element is 1
3. The height of a tree with > 1 element is equal to 1 + the height of its tallest subtree.

The depth of a node is the length of the path to its root (i.e., its root path). Every child node is always one level lower than his parent.

The topmost node in a tree is called the root node. Being the topmost node, the root node will not have parents. It is the node at which operations on the tree commonly begin (although some algorithms begin with the leaf nodes and work up ending at the root). All other nodes can

be reached from it by following edges or links. (In the formal definition, a path from a root to a node, for each different node is always unique). In diagrams, it is typically drawn at the top.

In some trees, such as heaps, the root node has special properties.

A subtree is a portion of a tree data structure that can be viewed as a complete tree in itself. Any node in a tree T, together with all the nodes below his height, that are reachable from the node, comprise a subtree of T. The subtree corresponding to the root node is the entire tree; the subtree corresponding to any other node is called a proper subtree (in analogy to the term proper subset).

Every node in a tree can be seen as the root node of the subtree rooted at that node.

Questions:

1. What is class, object and data structure?
2. What is tree data structure?
3. Explain different types of tree?

Input: Book name & its number of sections and subsections along with name.

Output: Formation of tree structure for book and its sections.

Conclusion: This program gives us the knowledge tree data structure.

Program:

```
#include <iostream>
# include <cstdlib>
# include <string.h>
using namespace std;
/*
 * Node Declaration
 */
struct node
{
   char label[10];
   int ch_count;
   struct node *child[10];
}*root;

/*
 * Class Declaration
 */
class BST
{
```

```
   public:
       void create_tree();
 void display(node * r1);

       BST()
       {
          root = NULL;
       }
};

void BST::create_tree()
{
 int tbooks,tchapters,i,j,k;
 root = new node();
 cout<<"Enter name of book";
 cin>>root->label;
 cout<<"Enter no. of chapters in book";
 cin>>tchapters;
 root->ch_count = tchapters;
 for(i=0;i<tchapters;i++)
 {
  root->child[i] = new node;
  cout<<"Enter Chapter name\n";
  cin>>root->child[i]->label;
  cout<<"Enter no. of sections in Chapter: "<<root->child[i]->label;
  cin>>root->child[i]->ch_count;
  for(j=0;j<root->child[i]->ch_count;j++)
  {
   root->child[i]->child[j] = new node;
   cout<<"Enter Section "<<j+1<<"name\n";
   cin>>root->child[i]->child[j]->label;
   //cout<<"Enter no. of subsections in "<<r1->child[i]->child[j]->label;
   //cin>>r1->child[i]->ch_count;
  }

 }

}

void BST::display(node * r1)
{
 int i,j,k,tchapters;
 if(r1 != NULL)
 {
  cout<<"\n-----Book Hierarchy---";
```

```
  cout<<"\n Book title : "<<r1->label;
  tchapters = r1->ch_count;
  for(i=0;i<tchapters;i++)
  {

   cout<<"\n Chapter "<<i+1;
   cout<<" "<<r1->child[i]->label;
   cout<<"\n Sections";
   for(j=0;j<r1->child[i]->ch_count;j++)
   {
    //cin>>r1->child[i]->child[j]->label;
    cout<<"\n  "<<r1->child[i]->child[j]->label;
   }

  }
 }
}


/*
 * Main Contains Menu
 */
int main()
{
    int choice;
    BST bst;
    while (1)
    {
        cout<<" ----------------"<<endl;
        cout<<"Book Tree Creation"<<endl;
        cout<<" ----------------"<<endl;
        cout<<"1.Create"<<endl;
        cout<<"2.Display"<<endl;
        cout<<"3.Quit"<<endl;
        cout<<"Enter your choice : ";
        cin>>choice;
        switch(choice)
        {
        case 1:
             bst.create_tree();
        case 2:
             bst.display(root);
           break;
        case 3:
           exit(1);
        default:
```

```
            cout<<"Wrong choice"<<endl;
        }
    }
}
```

OUTPUT

```
1.Create
2.Display
3.Quit
Enter your choice : 1
Enter name of bookC++
Enter no. of chapters in book2
Enter  Chapter name
Operators
Enter no. of sections in Chapter: Operators1
Enter Section 1name
Arithmetic
Enter Chapter name
Functions
Enter no. of sections in Chapter: Functions1
Enter Section 1name
FunctionDefine

-----Book Hierarchy---
 Book title : C++
  Chapter 1 Operators
 Sections
  Arithmetic
  Chapter 2 Functions
 Sections
  Function Define
                  Book Tree Creation
1. Create
2. Display
3.Quit
```

Experiment No: 7

Aim: Implement binary tree using linked list and perform recursive traversals.

Theory:

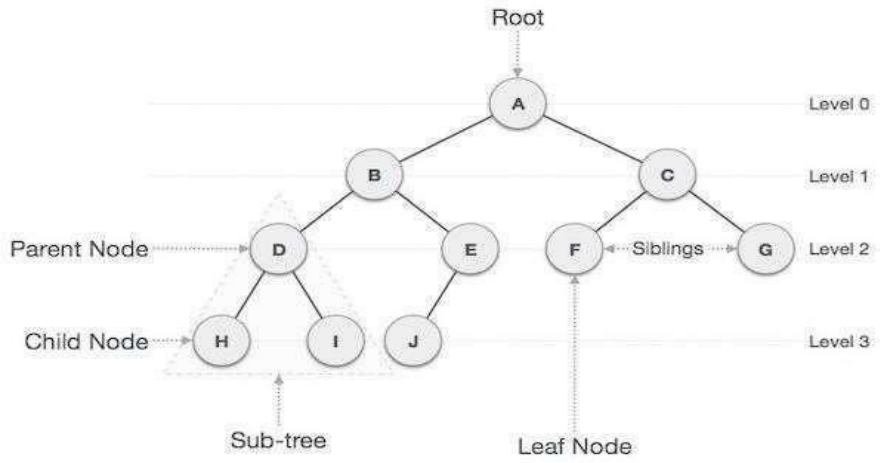
Tree

Tree represents the nodes connected by edges also a class of graphs that is acyclic is termed as trees. Let us now discuss an important class of graphs called trees and its associated terminology. Trees are useful in describing any structure that involves hierarchy. Familiar examples of suchstructures are family trees, the hierarchy of positions in an organization, and so on.

Binary Tree

A binary tree is made of nodes, where each node contains a "left" reference, a "right" reference, and a data element. The topmost node in the tree is called the root.

Every node (excluding a root) in a tree is connected by a directed edge from exactly one other node. This node is called a parent. On the other hand, each node can be connected to arbitrary number of nodes, called children. Nodes with no children are called leaves, or external nodes. Nodes which are not leaves are called internal nodes. Nodes with the same parent are called siblings.



Insert Operation

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

Traversals

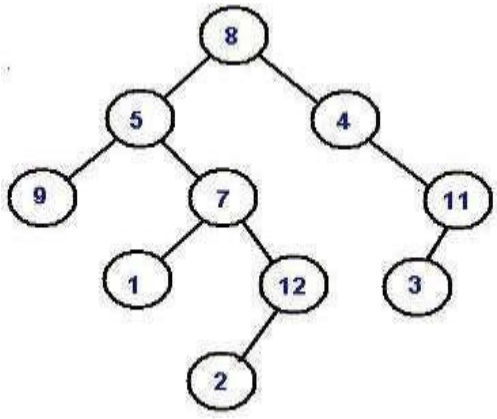
A traversal is a process that visits all the nodes in the tree. Since a tree is a nonlinear data structure, there is no unique traversal. We will consider several traversal algorithms with we group in the following two kinds

- depth-first traversal
- breadth-first traversal

There are three different types of depth-first traversals, :

- PreOrder traversal - visit the parent first and then left and right children;
- InOrder traversal - visit the left child, then the parent and the right child;
- PostOrder traversal - visit left child, then the right child and then the parent;

There is only one kind of breadth-first traversal--the level order traversal. This traversal visits nodes by levels from top to bottom and from left to right. As an example consider the following tree and its four traversals:



PreOrder - 8, 5, 9, 7, 1, 12, 2, 4, 11, 3
InOrder - 9, 5, 1, 7, 2, 12, 8, 4, 3, 11
PostOrder - 9, 1, 2, 12, 7, 5, 3, 11, 4, 8
LevelOrder - 8, 5, 4, 9, 7, 11, 1, 12, 3, 2

Algorithm:

Algorithm to insert a node :

Step 1 - Search for the node whose child node is to be inserted. This is a node at some level i, and a node is to be inserted at the level i +1 as either its left child or right child. This is the node after which the insertion is to be made.

Step 2 - Link a new node to the node that becomes its parent node, that is, either the Lchild or the Rchild.

Algorithm to traverse a tree :

- **Inorder traversal**

Until all nodes are traversed −

Step 1 − Recursively traverse left subtree.
Step 2 − Visit root node.

Step 3 − Recursively traverse right subtree.

- **Preorder**

Until all nodes are traversed

– Step 1 − Visit root node.

Step 2 − Recursively traverse left subtree.

Step 3 − Recursively traverse right subtree.

- **Postorder**

Until all nodes are traversed −

Step 1 − Recursively traverse left subtree.
Step 2 − Recursively traverse right subtree.
Step 3 − Visit root node.

Algorithm to copy one tree into another tree :

Step 1 – if (Root == Null)
Then return Null
Step 2 −Tmp = new TreeNode

Step 3 – Tmp->Lchild = TreeCopy(Root->Lchild);

Step 4 – Tmp->Rchild = TreeCopy(Root->Rchild);

Step 5 – Tmp-Data = Then return

Outcome

Learn object oriented programming features.

Understand & implement different operations on tree & binary tree.

Conclusion : Thus we have studied the implementation of various Binary tree operations.

Program:

```
#include <iostream>
using namespace std;
#include <conio.h>
struct tree
{
   tree *l, *r;
   int data;
}*root = NULL, *p = NULL, *np = NULL, *q;

void create()
{
   int value,c = 0;
   while (c < 7)
   {
      if (root == NULL)
      {
         root = new tree;
         cout<<"enter value of root node\n";
         cin>>root->data;
         root->r=NULL;
         root->l=NULL;
```

```
}
else
{
    p = root;
    cout<<"enter value of node\n";
    cin>>value;
    while(true)
    {
        if (value < p->data)
        {
            if (p->l == NULL)
            {
                p->l = new tree;
                p = p->l;
                p->data = value;
                p->l = NULL;
                p->r = NULL;
                cout<<"value entered in left\n";
                break;
            }
            else if (p->l != NULL)
            {
                p = p->l;
            }
        }
        else if (value > p->data)
        {
            if (p->r == NULL)
            {
                p->r = new tree;
```

```
                    p = p->r;
                    p->data = value;
                    p->l = NULL;
                    p->r = NULL;
                    cout<<"value entered in right\n";
                        break;
                     }
                else if (p->r != NULL)
                {
                    p = p->r;
                }
              }
           }
       }
       c++;
   }
}
void inorder(tree *p)
{
   if (p != NULL)
   {
       inorder(p->l);
       cout<<p->data<<endl;
       inorder(p->r);
   }
}
void preorder(tree *p)
{
   if (p != NULL)
   {
```

```
        cout<<p->data<<endl;
        preorder(p->l);
        preorder(p->r);
    }
}
void postorder(tree *p)
{
    if (p != NULL)
    {
        postorder(p->l);
        postorder(p->r);
        cout<<p->data<<endl;
    }
}
int main()
{
    create();
    cout<<"printing traversal in inorder\n";
    inorder(root);
    cout<<"printing traversal in preorder\n";
    preorder(root);
    cout<<"printing traversal in postorder\n";
    postorder(root);
    getch();
}
```

OUTPUT

```
enter value of root node
7
enter value of node
8
```

value entered in right

enter value of node

4

value entered in left

enter value of node

6

value entered in right

enter value of node

3

value entered in left

enter value of node

5

value entered in left

enter value of node

2

value entered in left

printing traversal in inorder

2

3

4

5

6

7

8

printing traversal in preorder

7

4

3

2

6

5

8

printing traversal in postorder

2

3

5

6

4

8

7

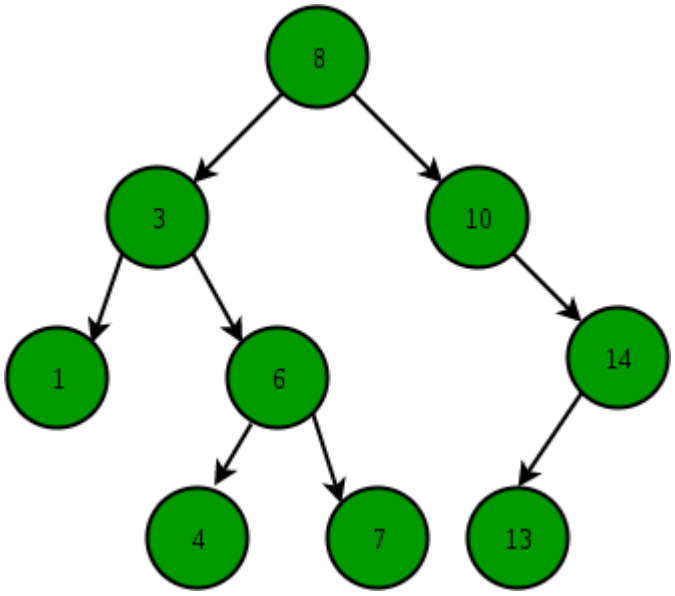
Experiment No-8

Aim: Beginning with an empty binary search tree, Construct binary search tree by inserting the values in the order given. After constructing a binary tree -
i. Insert new node
ii. Find number of nodes in longest path
iii. Minimum data value found in the tree
iv. Change a tree so that the roles of the left and right pointers are swapped at every node
v. Search a value

Theory:

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

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



ALGORITHMS:

Create and Insert:

1. Accept a random order of numbers from the user. The first number would be inserted at the root node.
2. Thereafter, every number is compared with the root node. If less than or equal to the data in the root node, proceed left of the BST, else proceed right.
3. Perform this till you reach a null pointer, the place where the present data is to be inserted.
4. Allocate a new node and assign the data in this node and allocate pointers appropriately.

Height of BST :

1. This algorithm is based on the idea of storing the nodes of every level of the BST in a dynamic queue (link list). It is also simultaneously useful to print the tree level wise. The total number of levels accessed would be the height of the tree.
2. Initialize the contents of the list with the root of the BST. The counter no_of_nodes_in_current_level =1 and the level_accessed =1.
3. Access no_of_nodes_in_current_level from the link list and add all their children to the list at the end and simultaneously keep track of the number of nodes accessed in the next level in a variable which at the end is assigned back to no_of_nodes_in_current_level. Also increment level_accessed, indicating one more level accessed in the BST.
4. Continue step 3 repeatedly till no_of_nodes_in_current_level is 0, which means no more nodes in the next level. The value stored in the variable level_accessed is the height of the BST.

(The above is a non recursive implementation to find the height of the BST. One could also write a recursive algorithm to do the same.)

Leaf Nodes of BST :

1. There are many algorithms to find the leaf nodes of a BST. The one considered here is based on the idea that one could do a simple inorder traversal of the BST and just before printing the data as one normally does in an inorder traversal, check if both the left and right nodes are NULL. If so, it means the node under consideration is a leaf node and must be printed.
2. Inorder: The recursive function will receive the root of the tree (or subtree) from where inorder traversal is to be initiated. Algorithm is to proceed left, which in this case is to call the same function with the left child of the node, print the data if both left and right pointers are NULL and then proceed right, which in this case is to call the same function with the right child of the node.
3. Thus all the leaf nodes of the BST are printed.

Mirror of Tree :

1. Following is a algorithm of a recursive function to find mirror of the tree. The function mirror_Tree accepts a pointer to a tree as the parameter. Initially the root node is passed later the roots of the subsequent subtrees are passed as the parameter.
2. The function begins by checking if the pointer passed is not NULL. If not, allocates a new node. Assigns the data of the original node to the copied node. Assigns the left child of the new node by calling the function mirror_Tree with the right child of the original node and assigns the right child of the new node by calling the function mirror_Tree with the left child of the original node. If the pointer passed is NULL, NULL is returned by the function else the new node created is returned.

Level wise printing :

1. This algorithm is based on the idea of storing the nodes of every level of the BST in a dynamic queue (link list).
2. Initialize the contents of the list with the root of the BST. The counter no_of_nodes_in_current_level =1 and the level_accessed =1.
3. Access no_of_nodes_in_current_level from the link list. Print the Level Number and all the data of all the nodes of the current level. Simultaneously add all their children to the list at the end and keep track of the number of nodes accessed in the next level in a variable which at the end is assigned back to no_of_nodes_in_current_level. Also increment level_accessed, indicating one more level accessed in the BST.
4. Continue step 3 repeatedly till no_of_nodes_in_current_level is 0, which means no more nodes in the next level.

(The above is a non recursive implementation to do level wise printing of the BST. One could also write a recursive algorithm to do the same.)

TEST CONDITIONS:

Simple input of random numbers. Display the height of the tree and the leaf nodes. The BST entered could be drawn on a rough page and one could check if the height calculated and the leaf nodes printed are correct.

For eg, Enter : 34, 12, 56, 6, 14, 40, 70.
The height of the BST is 3.
The leaf nodes are 6, 14, 40 and 70.

For Mirror Image:

Enter : 34, 12, 56, 6, 14, 40, 70.

Level wise printing of original tree

Level 1 : 34
Level 2 : 12, 56,
Level 3 : 6, 14, 40, 70

Level wise printing of the mirror tree

Level 1 : 34
Level 2 : 56, 12
Level 3 : 70,40, 14,6

INPUT :
Enter data (numbers) to be stored in the binary search tree. Every node in the BST would contain 3 fields: data, left child pointer and right child pointer

OUTPUT :

The height of the tree and the list of its leaf nodes.
The original and mirror image printed levelwise.

Program:

```
#include<iostream>

using namespace std;

struct node

{

int data;

node *L;

node *R;

};

node *root,*temp;

int count,key;


class bst

{

public:

void create();

void insert(node*,node*);

void disin(node*);

void dispre(node*);

void dispost(node*);

void search(node*,int);
```

```
int height(node*);
void mirror(node*);
void min(node*);
bst()
{
root=NULL;
count=0;
}
};

void bst::create()
{
char ans;
do
{
temp=new node;
cout<<"Enter the data : ";
cin>>temp->data;
temp->L=NULL;
temp->R=NULL;
if(root==NULL)
{
root=temp;
}
else
insert(root,temp);
cout<<"Do you want to insert more value : "<<endl;
```

```
 cin>>ans;
count++;
cout<<endl;
}while(ans=='y');
cout<<"The Total no.of nodes are:" <<count;
 }
void bst::insert(node *root,node* temp)
{
if(temp->data>root->data)
{
if(root->R==NULL)
{
root->R=temp;
}
else
insert(root->R,temp);
}
else
{
if(root->L==NULL)
{
root->L=temp;
}
else
insert(root->L,temp);
}
}
```

```
void bst::disin(node *root)
{
if(root!=NULL)
{
disin(root->L);
cout<<root->data<<"\t";
disin(root->R);
count++;
}
}
void bst::dispre(node *root)
{
if(root!=NULL)
{
cout<<root->data<<"\t";
dispre(root->L);
dispre(root->R);
}
}
void bst::dispost(node *root)
{
if(root!=NULL)
{
dispost(root->L);
dispost(root->R);
cout<<root->data<<"\t";
```

```
}
}
void bst::search(node * root,int key)
{
int flag=0;
cout<<"\nEnter your key : "<<endl;
cin>>key;
temp=root;
while(temp!=NULL)
{
if(key==temp->data)
{
cout<<"KEY FOUND\n";
 flag=1;
  break;
 } node *parent=temp; if(key>parent->data)
{
temp=temp->R;
}
else
{
temp=temp->L;
}

}
if(flag==0)
{
```

```
cout<<  "KEY NOT FOUND "          <<endl;
 }
 } int bst::height(node *root)
 { int hl,hr;
 if(root==NULL)
 { return 0; }
 else if(root->L==NULL && root->R==NULL)
{
return 0;
}
cout<<endl; hr=height(root->R);
hl=height(root->L);
if(hr>hl)
{
return(1+hr);
}
else
{
return(1+hl);
}
}

void bst::min(node *root)
{
temp=root;
cout<<endl; while(temp->L!=NULL)
{
```

```
temp=temp->L;
}
cout<<root->data;
}

void bst::mirror(node *root)
{
temp=root;
if(root!=NULL)
{
mirror(root->L);
mirror(root->R);
temp=root->L;
root->L=root->R;
root->R=temp;
}
}
int main()
{
bst t;
int ch;
char ans;
do
{
cout<<"\n1) Insert new node 2)number of nodes in longest path 3) minimum 4) mirror 5) search 6) inorder 7) preorder 8) postorder" <<endl;
cin>>ch;
```

```
switch(ch)
{
case 1:
t.create();
break;
case 2:
cout<<"\"n Number of nodes in longest path:"<<(1+(t.height(root)));
break;
case 3:
cout<<"\nThe min element is:";
t.min(root);
break;
case 4:
t.mirror(root);
cout<<"\nThe mirror of tree is: ";
t.disin(root);
break;
case 5:
t.search(root,key);
break;
case 6:
cout<<"\n***************INORDER**************" <<endl;
t.disin(root);
break;
case 7:
cout<<"\n***************PREORDER**************"<<endl;
t.dispre(root);
```

```
break;
case 8:
cout<<"\n**********POSTORDER**********"<<endl;
t.dispost(root);
break;
}
cout<<"\nDo you want to continue :"; cin>>ans;
}while(ans=='y');
return 0;
}
```

Output:

```
1) Insert new node 2)number of nodes in longest path 3) minimum 4) mirror 5) search 6) inorder 7) preorder 8) postorder
1
Enter the data : 3
Do you want to insert more value :
y

Enter the data : 4
Do you want to insert more value :
y

Enter the data : 1
Do you want to insert more value :
y
```

Enter the data : 6

Do you want to insert more value :

y

Enter the data : 9

Do you want to insert more value :

y

Enter the data : 7

Do you want to insert more value :

n

The Total no.of nodes are:6

Do you want to continue :y

1) Insert new node 2)number of nodes in longest path 3) minimum 4) mirror 5) search 6) inorder 7) preorder 8) postorder

2

"n Number of nodes in longest path:5

Do you want to continue :y

1) Insert new node 2)number of nodes in longest path 3) minimum 4) mirror 5) search 6) inorder 7) preorder 8) postorder

3

The min element is:

3

Do you want to continue :y

1) Insert new node 2)number of nodes in longest path 3) minimum 4) mirror 5) search 6) inorder 7) preorder 8) postorder

4

The mirror of tree is: 9 7 6 4 3 1

Do you want to continue :y

1) Insert new node 2)number of nodes in longest path 3) minimum 4) mirror 5) search 6) inorder 7) preorder 8) postorder

5

Enter your key :

1

KEY NOT FOUND

Do you want to continue :y

1) Insert new node 2)number of nodes in longest path 3) minimum 4) mirror 5) search 6) inorder 7) preorder 8) postorder

6

**********INORDER**********

9 7 6 4 3 1

Do you want to continue :y

1) Insert new node 2)number of nodes in longest path 3) minimum 4) mirror 5) search 6) inorder 7) preorder 8) postorder

7

**********PREORDER**********

3 4 6 9 7 1

Do you want to continue :y

1) Insert new node 2)number of nodes in longest path 3) minimum 4) mirror 5) search 6) inorder 7) preorder 8) postorder

8

**********POSTORDER**********

7 9 6 4 1 3

Do you want to continue :y

1) Insert new node 2)number of nodes in longest path 3) minimum 4) mirror 5) search 6) inorder 7) preorder 8) postorder

Experiment No. 9

Aim: Implement graph using adjacency list or matrix and perform DFS or BFS.

Theory:

ALGORITHMS:

Creation of Adjacency list :

1. Declare an array of pointers to a link list having a data field (to store vertex number) and a forward pointer. The number of array of pointers would equal the total number of vertices in the graph.
2. Take the edge set from the user. If for eg, vertex 1 is connected to vertex 2 and 3 in the graph, the 1st location of the array of pointers (corresponding to vertex 1) would point to 2 nodes, one having the data 2 (corresponding to vertex 2) and the other having data 3.
3. In this way construct the entire adjacency list.

DFS (Depth First Search).

1. The start vertex is visited. Next an unvisited vertex w adjacent to v is selected and a DFS from w initiated.
2. When a vertex u is reached such that all its adjacent vertices have been visited, we back up to the last vertex visited which has an unvisited vertex w adjacent to it and initiate a DFS search from w.
3. The search terminates when no unvisited vertex can be reached from any of the visited ones.

BFS(Breadth First Search).

1. Starting at vertex v and marking it as visited, BFS differs from DFS in that all unvisited vertices adjacent to v are visited next.
2. Then unvisited vertices adjacent to these vertices are visited and so on.
3. A queue is used to store vertices as they are visited so that later search can be initiated from those vertices.

TEST CONDITIONS:

Enter the graph with 8 vertices and 10 edges (1,2), (1,3), (2,4), (2,5), (3,6), (3,7), (4,8), (5,8), (6,8),(7,8).

The order of the vertices visited by DFS is : 1, 2, 4, 8, 5, 6, 3, 7.

The order of the vertices visited by BFS is : 1, 2, 3, 4, 5, 6, 7, 8.

INPUT :

The number of vertices and the edge set of the graph

OUTPUT :

The order of vertices visited in both DFS and BFS.

```
// C++ implementation of the approach
#include <bits/stdc++.h>
using namespace std;

class Graph {

    // Number of vertex
    int v;

    // Number of edges
    int e;

    // Adjacency matrix
    int** adj;

public:
    // To create the initial adjacency matrix
    Graph(int v, int e);

    // Function to insert a new edge
    void addEdge(int start, int e);

    // Function to display the BFS traversal
    void BFS(int start);
```

```
};

// Function to fill the empty adjacency matrix
Graph::Graph(int v, int e)
{
	this->v = v;
	this->e = e;
	adj = new int*[v];
	for (int row = 0; row < v; row++) {
		adj[row] = new int[v];
		for (int column = 0; column < v; column++) {
			adj[row][column] = 0;
		}
	}
}

// Function to add an edge to the graph
void Graph::addEdge(int start, int e)
{

	// Considering a bidirectional edge
	adj[start][e] = 1;
	adj[e][start] = 1;
}

// Function to perform BFS on the graph
void Graph::BFS(int start)
```

```
{
        // Visited vector to so that
        // a vertex is not visited more than once
        // Initializing the vector to false as no
        // vertex is visited at the beginning
        vector<bool> visited(v, false);
        vector<int> q;
        q.push_back(start);

        // Set source as visited
        visited[start] = true;

        int vis;
        while (!q.empty()) {
                vis = q[0];

                // Print the current node
                cout << vis << " ";
                q.erase(q.begin());

                // For every adjacent vertex to the current vertex
                for (int i = 0; i < v; i++) {
                        if (adj[vis][i] == 1 && (!visited[i])) {

                                // Push the adjacent node to the queue
                                q.push_back(i);
```

```
                                // Set
                                visited[i] = true;
                        }
                }
        }
}

// Driver code
int main()
{
        int v = 5, e = 4;

        // Create the graph
        Graph G(v, e);
        G.addEdge(0, 1);
        G.addEdge(0, 2);
        G.addEdge(1, 3);

        G.BFS(0);
}
```

Output:

0 1 2 3

Experiment No. 10

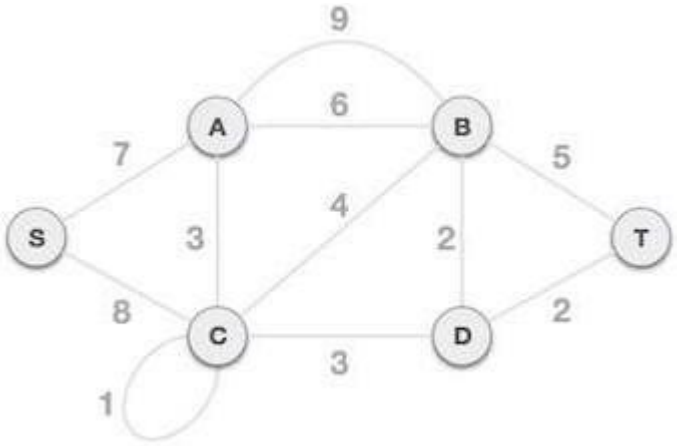
Aim: You have a business with several offices; you want to lease phone lines to connect them up with each other; and the phone company charges different amounts of money to connect different pairs of cities. You want a set of lines that connects all your offices with a minimum total cost. Solve the problem by suggesting appropriate data structures

Theory:

Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach. Prim's algorithm shares a similarity with the **shortest path first** algorithms.

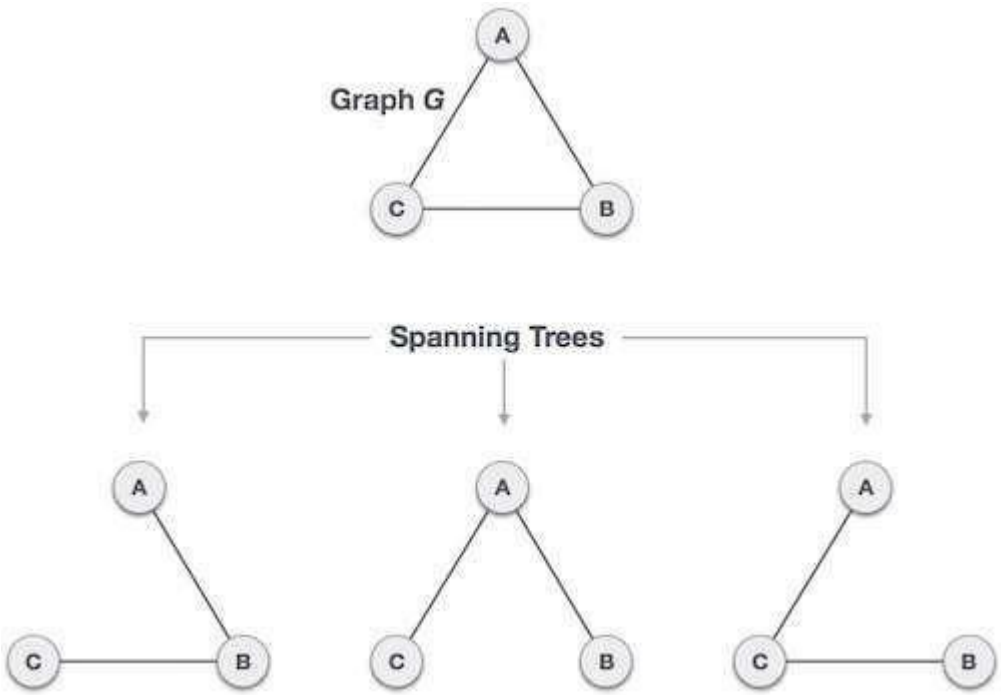
Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.

To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example −



A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected..

By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.



We found three spanning trees off one complete graph. A complete undirected graph can havemaximum $\mathbf{n^{n-2}}$ number of spanning trees, where **n** is the number of nodes. In the above addressed example, **n is 3,** hence $\mathbf{3^{3-2} = 3}$ spanning trees are possible.

General Properties of Spanning Tree

We now understand that one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph G −

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is **maximally acyclic**.

Mathematical Properties of Spanning Tree

- Spanning tree has **n-1** edges, where **n** is the number of nodes (vertices).
- From a complete graph, by removing maximum **e - n + 1** edges, we can construct a spanning tree.
- A complete graph can have maximum $\mathbf{n^{n-2}}$ number of spanning trees.

Thus, we can conclude that spanning trees are a subset of connected Graph G and disconnected graphs do not have spanning tree.

Application of Spanning Tree

Spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common application of spanning trees are −

- **Civil Network Planning**
- **Computer Network Routing Protocol**
- **Cluster Analysis**

Let us understand this through a small example. Consider, city network as a huge graph and now plans to deploy telephone lines in such a way that in minimum lines we can connect to all city nodes. This is where the spanning tree comes into picture.

Minimum Spanning Tree (MST)

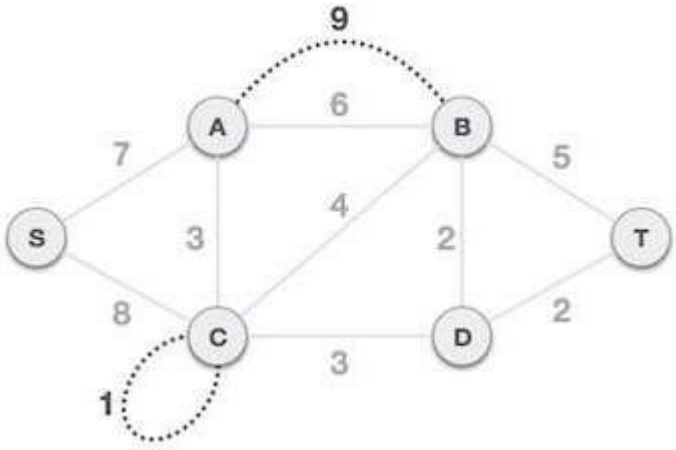
In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

Minimum Spanning-Tree Algorithm

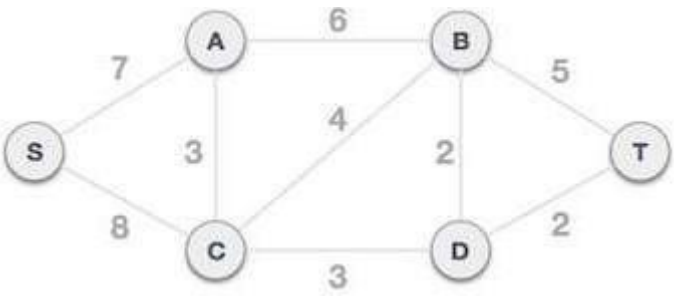
We shall learn about two most important spanning tree algorithms here −

- Kruskal's Algorithm
- Prim's Algorithm

Step 1 - Remove all loops and parallel edges



Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

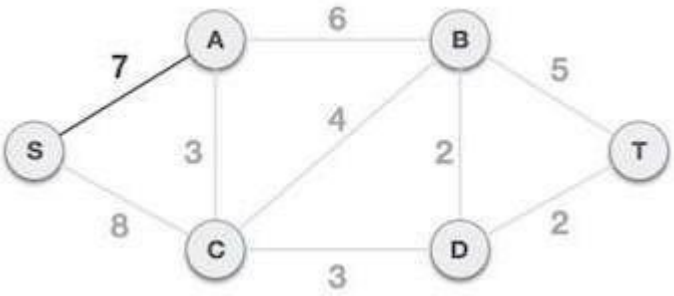


Step 2 - Choose any arbitrary node as root node

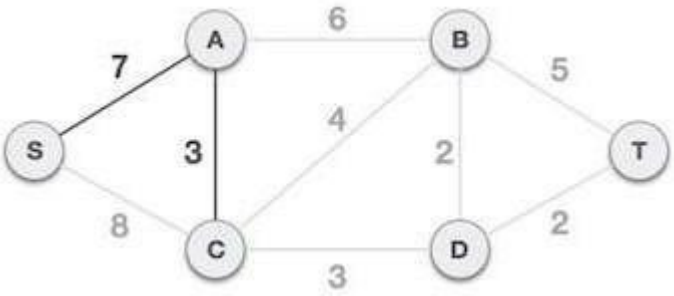
In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

Step 3 - Check outgoing edges and select the one with less cost

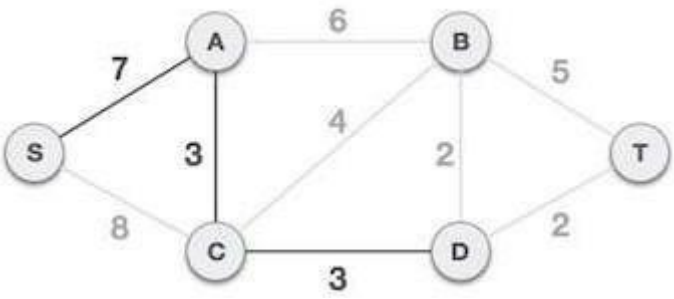
After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



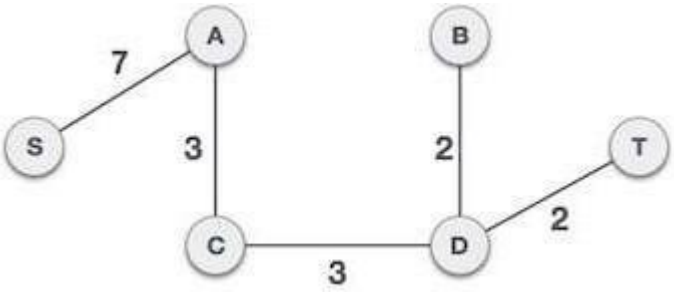
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



After adding node **D** to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.



We may find that the output spanning tree of the same graph using two different algorithms is same.

Program:

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


const int MAX=10;
class EdgeList; //forward declaration
class Edge //USED IN KRUSKAL
{
 int u,v,w;
public:
 Edge(){} //Empty Constructor
```

```
 Edge(int a,int b,int weight)
 {
  u=a;
  v=b;
  w=weight;
 }
 friend class EdgeList;
 friend class PhoneGraph;
};
//---- EdgeList Class ----------
class EdgeList
{
 Edge data[MAX];
 int n;
public:
 friend class PhoneGraph;
 EdgeList()
 { n=0;}
 void sort();
 void print();

};
//----Bubble Sort for sorting edges in increasing weights' order --- //
void EdgeList::sort()
{
 Edge temp;
 for(int i=1;i<n;i++)
```

```
  for(int j=0;j<n-1;j++)
   if(data[j].w>data[j+1].w)
   {
    temp=data[j];
    data[j]=data[j+1];
    data[j+1]=temp;
   }
}
void EdgeList::print()
{
 int cost=0;
 for(int i=0;i<n;i++)
 {
  cout<<"\n"<<i+1<<" "<<data[i].u<<"--"<<data[i].v<<" = "<<data[i].w;
  cost=cost+data[i].w;
 }
 cout<<"\nMinimum cost of Telephone Graph = "<<cost;
}
//.....Phone Graph Class.....
class PhoneGraph
{
 int data[MAX][MAX];
 int n;

public:
 PhoneGraph(int num)
{
```

```
  n=num;
}
 void readgraph();
 void printGraph();
 int mincost(int cost[],bool visited[]);
 int prim();
 void kruskal(EdgeList &spanlist);
 int find(int belongs[], int vertexno);
 void unionComp(int belongs[], int c1,int c2);
};
void PhoneGraph::readgraph()
{
 cout<<"Enter Adjacency(Cost) Matrix: \n";
 for(int i=0;i<n;i++)
 {
  for(int j=0;j<n; j++)
   cin>>data[i][j];
 }
}
void PhoneGraph::printGraph()
{
 cout<<"\nAdjacency (COST) Matrix: \n";
 for(int i=0;i<n;i++)
 {
  for(int j=0;j<n;j++)
  {
   cout<<setw(3)<<data[i][j];
```

```
  }
  cout<<endl;
 }
}
int PhoneGraph::mincost(int cost[],bool visited[]) //finding vertex with minimum cost
{
 int min=9999,min_index; //initialize min to MAX value(ANY) as temporary
 for(int i=0;i<n;i++)
 {
  if(visited[i]==0 && cost[i]<min)
  {
   min=cost[i];
   min_index=i;
  }
 }
 return min_index; //return index of vertex which is not visited and having minimum cost

}
int PhoneGraph::prim()
{
 bool visited[MAX];
 int parents[MAX];
 int cost[MAX]; //saving minimum cost
 for(int i=0;i<n;i++)
 {
  cost[i]=9999; //set cost as infinity/MAX_VALUE
  visited[i]=0; //initialize visited array to false
```

```
}
cost[0]=0; //starting vertex cost
parents[0]=-1; //make first vertex as a root
for(int i=0;i<n-1;i++)
{
 int k=mincost(cost,visited);
 visited[k]=1;

 for(int j=0;j<n;j++)
 {
  if(data[k][j] && visited[j]==0 && data[k][j] < cost[j])
  {
   parents[j]=k;
   cost[j]=data[k][j];
  }
 }
}
cout<<"Minimum Cost Telephone Map:\n";
for(int i=1;i<n;i++)
{
 cout<<i<<" -- "<<parents[i]<<" = "<<cost[i]<<endl;
}
int mincost=0;
for (int i = 1; i < n; i++)
mincost+=cost[i];    //data[i][parents[i]];
return mincost;
}
```

```
// -------Kruskal's Algorithm
void PhoneGraph::kruskal(EdgeList &spanlist)
{
 int belongs[MAX]; //Separate Components at start (No Edges, Only vertices)
 int cno1,cno2; //Component 1 & 2
 EdgeList elist;
 for(int i=1;i<n;i++)
 for(int j=0;j<i;j++)
  {
   if(data[i][j]!=0)
   {
    elist.data[elist.n]=Edge(i,j,data[i][j]); //constructor for initializing edge
    elist.n++;
   }
  }
 elist.sort(); //sorting in increasing weight order
 for(int i=0;i<n;i++)
  belongs[i]=i;

 for(int i=0;i<elist.n;i++)
 {
  cno1=find(belongs,elist.data[i].u); //find set of u
  cno2=find(belongs,elist.data[i].v); ////find set of v
  if(cno1!=cno2) //if u & v belongs to different sets
  {
   spanlist.data[spanlist.n]=elist.data[i]; //ADD Edge to spanlist
   spanlist.n=spanlist.n+1;
```

```
   unionComp(belongs,cno1,cno2); //ADD both components to same set
  }
 }
}
void PhoneGraph::unionComp(int belongs[],int c1,int c2)
{
 for(int i=0;i<n;i++)
 {
  if(belongs[i]==c2)
   belongs[i]=c1;
 }
}
int PhoneGraph::find(int belongs[],int vertexno)
{
 return belongs[vertexno];
}

//----------MAIN PROGRAM----------
int main() {
 int vertices,choice;
 EdgeList spantree;
 cout<<"Enter Number of cities: ";
 cin>>vertices;
 PhoneGraph p1(vertices);
 p1.readgraph();
 do
 {
```

```
cout<<"\n1.Find Minimum Total Cost(By Prim's Algorithm)"
  <<"\n2.Find Minimum Total Cost(by Kruskal's Algorithms)"
  <<"\n3.Re-Read Graph(INPUT)"
  <<"\n4.Print Graph"
  <<"\n0. Exit"
  <<"\nEnter your choice: ";
cin>>choice;
switch(choice)
{
case 1:
 cout<<" Minimum cost of Phone Line to cities is: "<<p1.prim();
 break;
case 2:
 p1.kruskal(spantree);
 spantree.print();
 break;
case 3:
 p1.readgraph();
 break;
case 4:
 p1.printGraph();
 break;
default:
 cout<<"\nWrong Choice!!!";
 }
}while(choice!=0);
```

```
 return 0;
}
```

Output:

Sample INPUT: vertices =7

```
{{0, 28, 0, 0, 0,10,0},
 {28,0,16,0,0,0,14},
 {0,16,0,12,0,0,0},
 {0,0,12,0,22,0,18},
 {0,0,0,22,0,25,24},
 {10,0,0,0,25,0,0},
 {0,14,0,18,24,0,0},
};
```

Minimum Cost: 99

Experiment No 11

Aim- A classic problem that can be solved by backtracking is called the Eight Queens problem, which comes from the game of chess. The chess board consists of 64 square arranged in an 8 by 8 grid. The board normally alternates between black and white square, but this is not relevant for the present problem.The queen can move as far as she wants in any direction, as long as she follows a straight line, Vertically, horizontally, or diagonally. Write C++ program for generating all possible configurations for 4-queen's problem.

Theory:

This problem is to find an arrangement of N queens on a chess board, such that no queen can attack any other queens on the board. The chess queens can attack in any direction as horizontal, vertical, horizontal and diagonal way. A binary matrix is used to display the positions of N Queens, where no queens can attack other queens.

Algorithm

isValid(board, row, col)

Input: The chess board, row and the column of the board.

Output – True when placing a queen in row and place position is a valid or not.

```
Begin
  if there is a queen at the left of current col, then
    return false
  if there is a queen at the left upper diagonal, then
    return false
  if there is a queen at the left lower diagonal, then
    return false;
  return true //otherwise it is valid place
End
```

solveNQueen(board, col)

Input − The chess board, the col where the queen is trying to be placed.

Output − The position matrix where queens are placed.

```
Begin
  if all columns are filled, then
    return true
  for each row of the board, do
    if isValid(board, i, col), then
      set queen at place (i, col) in the board
      if solveNQueen(board, col+1) = true, then
```

```
        return true
      otherwise remove queen from place (i, col) from board.
  done
  return false
End
```

Programme code:

```
#include<iostream>

using namespace std;

#define N 8


void printBoard(int board[N][N]) {

  for (int i = 0; i < N; i++) {

    for (int j = 0; j < N; j++)

      cout << board[i][j] << " ";

    cout << endl;

  }

}


bool isValid(int board[N][N], int row, int col) {

  for (int i = 0; i < col; i++)    //check whether there is queen in the left or not

    if (board[row][i])

      return false;

  for (int i=row, j=col; i>=0 && j>=0; i--, j--)

    if (board[i][j])       //check whether there is queen in the left upper diagonal or not

      return false;
```

```
   for (int i=row, j=col; j>=0 && i<N; i++, j--)
      if (board[i][j])      //check whether there is queen in the left lower diagonal or not
         return false;
   return true;
}

bool solveNQueen(int board[N][N], int col) {
   if (col >= N)          //when N queens are placed successfully
      return true;
   for (int i = 0; i < N; i++) {      //for each row, check placing of queen is possible or not
      if (isValid(board, i, col) ) {
         board[i][col] = 1;      //if validate, place the queen at place (i, col)
         if ( solveNQueen(board, col + 1))    //Go for the other columns recursively
            return true;

         board[i][col] = 0;       //When no place is vacant remove that queen
      }
   }
   return false;       //when no possible order is found
}

bool checkSolution() {
   int board[N][N];
   for(int i = 0; i<N; i++)
      for(int j = 0; j<N; j++)
         board[i][j] = 0;      //set all elements to 0
```

```
  if ( solveNQueen(board, 0) == false ) {      //starting from 0th column
     cout << "Solution does not exist";
     return false;
  }
  printBoard(board);
  return true;
}

int main() {
  checkSolution();
}
```

Input and Output

```
Input:
The size of a chess board. Generally, it is 8. as (8 x 8 is the size of a
normal chess board.)
Output:
The matrix that represents in which row and column the N Queens can be
placed.
If the solution does not exist, it will return false.

1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0

In this output, the value 1 indicates the correct place for the queens.
The 0 denotes the blank spaces on the chess board.
```

Experiment No.12

Aim: A Dictionary stores keywords & its meanings. Provide facility for adding new keywords, deleting keywords, updating values of any entry. Provide facility to display whole data sorted in ascending/

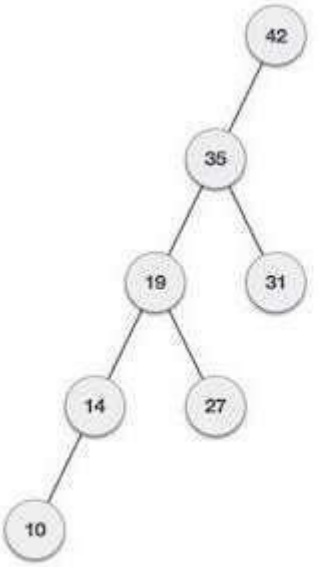
Descending order. Also find how many maximum comparisons may require for finding any keyword. Use height balance tree and find the complexity for finding a keyword.

Theory:

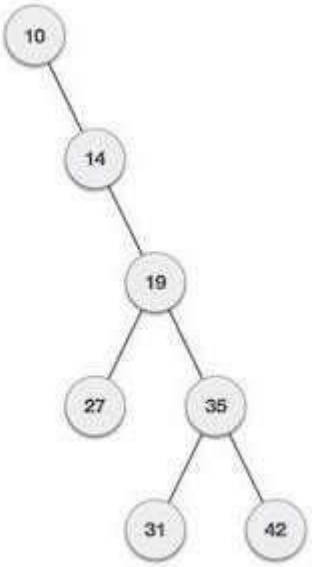
An empty tree is height balanced tree if T is a nonempty binary tree with TL and TR as its left and right sub trees. The T is height balance if and only if Its balance factor is 0, 1, -1.

AVL (Adelson- Velskii and Landis) Tree: A balance binary search tree. The best search time, that is O (log N) search times. An AVL tree is defined to be a well-balanced binary search tree in which each of its nodes has the AVL property. The AVL property is that the heights of the left and right sub-trees of a node are either equal or if they differ only by 1.

What if the input to binary search tree comes in a sorted (ascending or descending) manner? It will then look like this −



If input 'appears' non-increasing manner

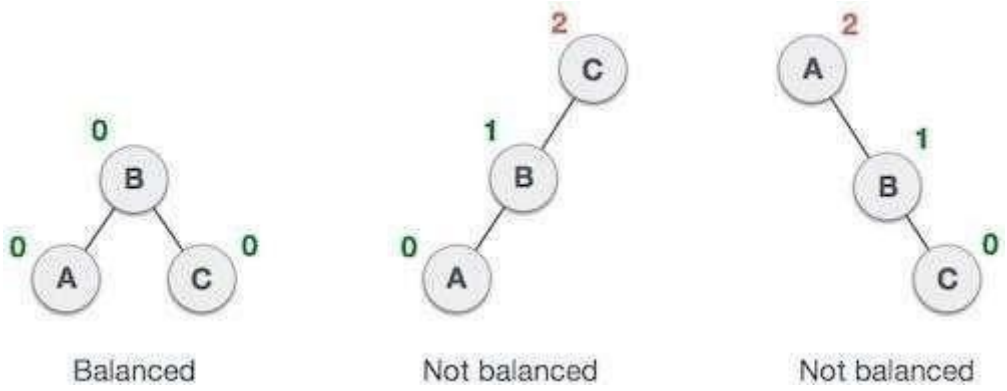


If input 'appears' in non-decreasing manner

It is observed that BST's worst-case performance is closest to linear search algorithms, that is O(n). In real-time data, we cannot predict data pattern and their frequencies. So, a need arises to balance out the existing BST.

Named after their inventor **Adelson**, **Velski** & **Landis**, **AVL trees** are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the **Balance Factor**.

Here we see that the first tree is balanced and the next two trees are not balanced −



In the second tree, the left subtree of **C** has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of **A** has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

$$\textbf{\textit{BalanceFactor}} = \text{height(left-sutree)} - \text{height(right-sutree)}$$

If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

AVL Rotations

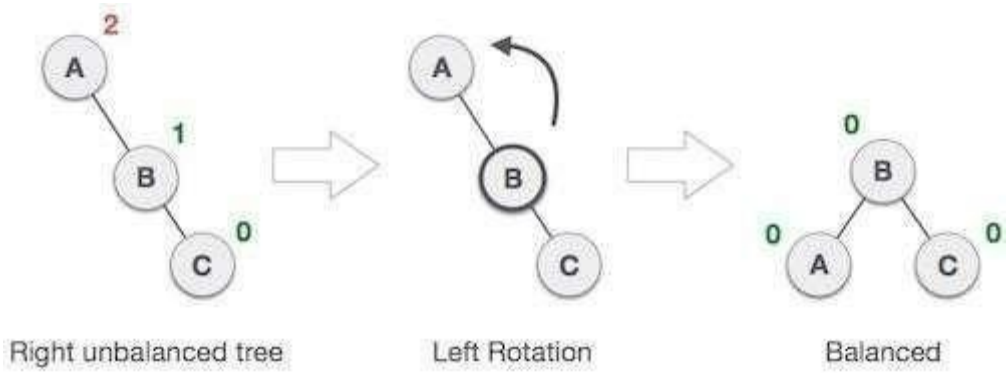
To balance itself, an AVL tree may perform the following four kinds of rotations −

- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation

The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

Left Rotation

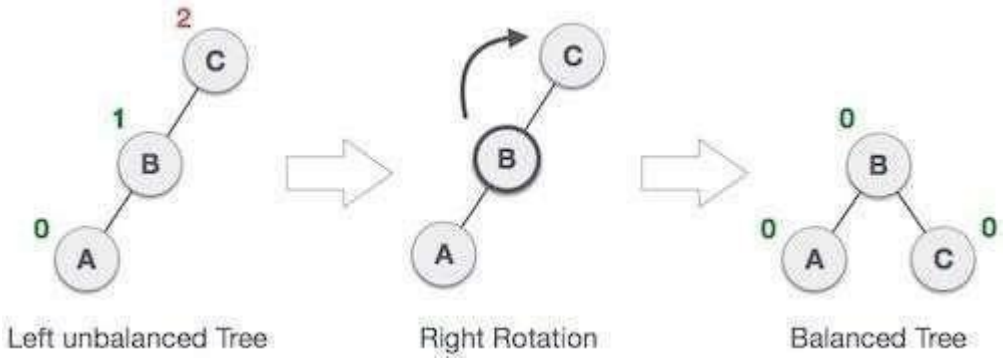
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation −



In our example, node **A** has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making **A** the left-subtree of B.

Right Rotation

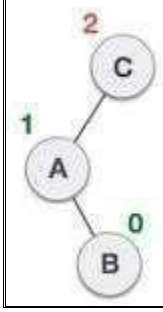
AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.

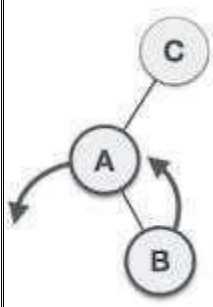
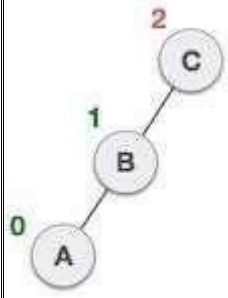
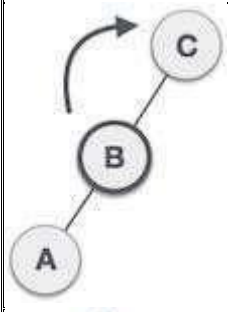
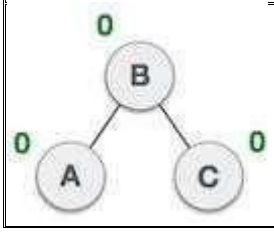


As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

Left-Right Rotation

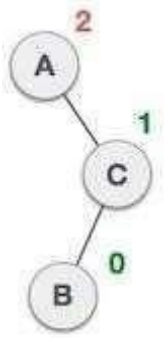
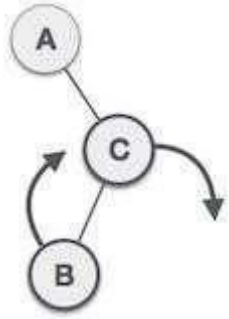
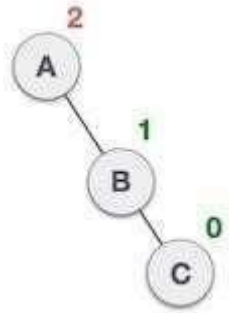
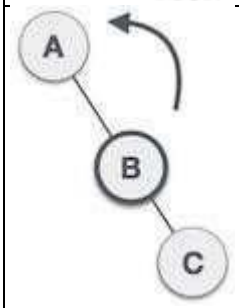
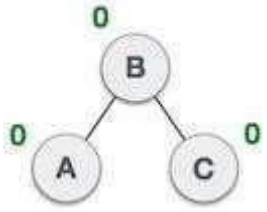
Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.

State	Action
	A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.

State	Action
	We first perform the left rotation on the left subtree of **C**. This makes **A**, the left subtree of **B**.
	Node **C** is still unbalanced, however now, it is because of the left-subtree of the left-subtree.
	We shall now right-rotate the tree, making **B** the new root node of this subtree. **C** now becomes the right subtree of its own left subtree.
	The tree is now balanced.

Right-Left Rotation

The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

	A node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2.
	First, we perform the right rotation along **C** node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes the right subtree of **A**.
	Node **A** is still unbalanced because of the right subtree of its right subtree and requires a left rotation.
	A left rotation is performed by making **B** the new root node of the subtree. **A** becomes the left subtree of its right subtree **B**.
	The tree is now balanced.

Algorithm AVL TREE:

Insert:-

1. If P is NULL, then
    I. P = new node
    II. P ->element = x
    III. P ->left = NULL
    IV. P ->right = NULL
    V. P ->height = 0
2. else if x>1 => x<P ->element
    a.) insert(x, P ->left)
    b.) if height of P->left -height of P ->right =2
        1. insert(x, P ->left)
        2. if height(P ->left) -height(P ->right) =2
            if x<P ->left ->element
                P =singlerotateleft(P)
            else
                P =doublerotateleft(P)
3. else
    if x<P ->element

        a.) insert(x, P -> right)
        b.) if height (P -> right) -height (P ->left) =2
            if(x<P ->right) ->element
                P =singlerotateright(P)
            else
                P =doublerotateright(P)
4. else

Print already exits

5. int m, n, d.

6. m = AVL height (P->left)
7. n = AVL height (P->right)
8. d = max(m, n)
9. P->height = d+1
10. Stop

RotateWithLeftChild(AvlNode k2)

- AvlNode k1 = k2.left;
- k2.left = k1.right;
- k1.right = k2;
- k2.height = max(height(k2.left), height(k2.right)) + 1;
- k1.height = max(height(k1.left), k2.height) + 1;
- return k1;

RotateWithRightChild(AvlNode k1)

- AvlNode k2 = k1.right;
- k1.right = k2.left;
- k2.left = k1;
- k1.height = max(height(k1.left), height(k1.right)) + 1;
- k2.height = max(height(k2.right), k1.height) + 1;
- return k2;

doubleWithLeftChild(AvlNode k3)

- k3.left = rotateWithRightChild(k3.left);
- return rotateWithLeftChild(k3);

doubleWithRightChild(AvlNode k1)

- k1.right = rotateWithLeftChild(k1.right);
- return rotateWithRightChild(k1);

Conclusion: This program gives us the knowledge height balanced binary tree.

Program:

Function to get max element:

```
int getmax(int h1,int h2)
{
if(h1>h2)
   return h1;
return h2;
```

```
}
   avlnode * dict::rrotate(avlnode *y)
   {
   cout<<endl<<"rotating right - "<< y->key;
   avlnode *x=y->left;
   avlnode *t=x->right;
   x->right= y;
   y->left=t;
   y->ht=getmax(getht(y->left),getht(y->right))+1;
   x->ht=getmax(getht(x->left),getht(x->right))+1;
   return x;
   }
   avlnode * dict::lrotate(avlnode *x)
   {
   cout<<endl<<"rotating left - "<< x->key;

   avlnode *y=x->right;
   avlnode *t=y->left;
   x->right= t;
   y->left=x;
   y->ht=getmax(getht(y->left),getht(y->right))+1;
   x->ht=getmax(getht(x->left),getht(x->right))+1;
   return y;
   }
```

OUTPUT:

```
[@localhost ~]$ ./a.out
Menu
1. Insert node
2.Inorder Display tree
Enter Choice1

enter key and meaning(single char)a
a

enter key and meaning(single char)b
b

height - 2
node bal - -1 current key is b
```

```
enter key and meaning(single char)c
c

height - 2
node bal - -1 current key is c
height - 3
node bal - -2 current key is c
rotating left - a
enter key and meaning(single char)z
z

height - 2
node bal - -1 current key is z
height - 3
node bal - -1 current key is z
do u want to continue?(1 for continue)1

Menu
1.Insert node
2. Inorder Display tree
Enter Choice2
 a a b b c c z z
do u want to continue?(1 for continue)0
```