DATA STRUCTURES AND ALGORITHM

(LAB MANUAL)

Prepared for

Tahreem Iqbal

COMSATS University Islamabad, Sahiwal Campus

Prepared By

Naeem ur Rahman Sajid

FA20-BCS-099

January 20, 2022

Table of Contents

1) Searching in Arrays

i) Linear Searching (sequential):

Search begins by comparing the first element of the list with the target element. If it matches, the search ends. Otherwise, move to the next element and compare. In this way, the target element is compared with all the elements until a match occurs. If the match does not occur and there are no more elements to be compared, conclude that the target element is absent in the list.

For example, consider the following list of elements.

5 9 7 8 11 2 6 4

To search for element 2 . First, compare the target element with the first element in the list is 5. Since both are not matching we move on to the next elements in the list and compare. Finally found the match after 6 comparisons.

Code: (Iterative Method)

```
#include <iostream>

using namespace std;

int main(){

        int s, flag=0 , n ;

        cout<< "Enter size of Array : ";cin>>s;

        int arr[s];

        for(int i=0 ; i<s ;i++ ) {          // loop for entering the array Values

        cout<< "Enter "<<i<<" index value : "; cin>>arr[i];

       }

        cout << "Enter a Value to Search : "; cin>>n ; //Taking value for the Search

        for(int i=0;i<s;i++) {   // Loop for the Liner Search

        if  ( n == arr[i] )        {  // Comparisons
```

```
                cout << "Value is at index : "<<i<<endl;
        flag++;                 break;      } // break loop when value found
        }
        If  (flag == 0) {  // Condition if value not found in array
        cout << "Value is not found !"<<endl; }
        return 0;
}
```

ii) Binary Searching:

Before searching, the list of items should be sorted in ascending order. First, compare the key value with the item in the mid position of the array. If there is a match, we can return immediately the position. if the value is less than the element in the middle location of the array, the required value lies in the lower half of the array. if the value is greater than the element in the middle location of the array, the required value lies in the upper half of the array. We repeat the above procedure on the lower half or upper half of the array.

Code: (Iterative Method)

```
#include <iostream>
using namespace std;
int i=0,j=0,t=0,n=0;
void input (int[],int&); // Function for Input values in Arrays
void sort  (int[],int&); // Function for sorting values in Array
void print (int[],int&); // Function for Display Array Values
int search (int[],int&); //Function for Binary Search
int main(){
        int s;
```

```
        cout << "Enter size of Array : ";cin>>s;
        int arr[s]; // Initializing array
        //Function Calls
         input (arr,s);
        print (arr,s);
        sort  (arr,s);
        print (arr,s);
        int num =  search (arr,s); // Biinary search function giving the index of search value
        if (num==-1){
                cout << "_____This Value is not Present in Array_____ "<<endl;
        } else{
                cout << "This "<<arr[num]<<" value is at index : "<<num<<endl; }
 return 0;}
void input (int arr[],int& s){
        cout << "Enter the Values in Array "<<endl;
        for (i=0;i<s;i++){
                cout << "Enter "<<i<<" index value : ";cin>>arr[i];
}}
void sort (int arr[],int& s){
        cout << "Sorting is Processing ! "<<endl;
        for(i=0;i<s-1;i++){
                for(j=0;j<s-1;j++){
                        if (arr[j]>arr[j+1]){
                           t = arr[j];
```

```
                    arr[j]=arr[j+1];
                    arr[j+1]=t;
        } } } }
void print (int arr[],int& s){
        cout << "Array is "<<endl;
        for(i=0;i<s;i++){
                cout << arr[i]<<"  ";  }
        cout <<endl;}
int search(int arr[],int& s){
        cout << "Enter a number to search : ";cin>>n;
        cout << "Search is in Processing ! "<<endl;
        int m=s/2; // Finding mid index of array
        if (n==arr[m]){  //Value at Mid Find
                return m;
          }    else if (n<arr[m]) { // Left Array search Condition
                for(i=m;i>=0;i--){
                        if(n==arr[i]){
                                return i; }
        }}   else if(n>arr[m]) {  // Right Array search Condition
                for(i=m+1;i<s;i++){
                        if (n==arr[i]){
                                return i; }
        }}
        return  -1 ;   } // if value is not found in the Array
```

2) Sorting in Arrays

i) Bubble Sort:

The bubble sort is an example of exchange sort. In this method, repetitive comparison is performed among elements and essential swapping of elements is done. Bubble sort is commonly used in sorting algorithms. It is easy to understand but time consuming i.e. takes more number of comparisons to sort a list. In this type, two successive elements are compared and swapping is done. Thus, step-by-step entire array elements are checked. It is different from the selection sort. Instead of searching the minimum element and then applying swapping, two records are swapped instantly upon noticing that they are not in order.

ALGORITHM:

Bubble_Sort (A [ ], N)

Step 1: Start

Step 2: Take an array of n elements

Step 3: for i = 0,…….n-2

Step 4: for j= i+1,…….n-1

Step 5: if arr [j] > arr [j+1] then Interchange arr[j] and arr [j+1] End of if

Step 6: Print the sorted array arr

Step 7: Stop

Code: (Iterative)

```
#include <iostream>
using namespace std;
void A_sort(int[],int&);
int main(){
        int  s =5, arr[5] ={5,2,7,9,3};
        display (arr,s);
```

```
        A_sort (arr,s);
        display (arr,s);
        return 0 ; }
void display(int arr[],int &s){
        for (int i=0;i<s;i++){
                cout << arr[i]<<" ";  }
        cout <<endl;  }
void A_sort(int arr[],int &s){
        int t ;
        for (int i=0;i<s-1  ;i++){  // First loop
        for (int j=0;j<s-i-1;j++){        // second loop
                if(arr[j]>arr[j+1]){ // swapping condition
                        t=arr[j];       // swapping values
                        arr[j]=arr[j+1];
                        arr[j+1]=t;  }
      }  }
 }
```

ii) Selection sort (Select the smallest and Ex-change):

The first item is compared with the remaining n-1 items, and whichever of all is lowest, is put in the first position. Then the second item from the list is taken and compared with the remaining (n-2) items, if an item with a value less than that of the second item is found on the (n-2) items, it is swapped (Interchanged) with the second item of the list and so on.

ALGORITHM:

Selection_Sort (A [ ], N)

Step 1: Start

Step 2: Repeat For K = 0 to N – 2 Begin

Step 3: Set POS = K

Step 4: Repeat for J = K + 1 to N – 1 Begin

If A [J] < A [POS]

Set POS = J

End For

Step 5: Swap A [K] with A [POS] End For

Step 6: Stop

Code: (Iterative)

```
#include <iostream>
using namespace std;
void print(int[],int&);  // Display Function
void selection_sort(int[],int&); // Selection sort function
int main(){
        int s = 7, arr[s]= {80,70,60,55,50,10,90};
        print(arr,s);
        selection_sort(arr,s);
```

```
        print(arr,s);
        return 0;
}
void print(int arr[],int&s){
        cout<<"Array is ! " <<endl;
        for(int i=0;i<s;i++){
                cout << arr[i]<<"  "; }
        cout << endl; }
void selection_sort(int arr[],int& s){
        int min=0 , temp;
        for(int j=0;j<s-1;j++){
                min = j;
        for( int i=0+j;i<s-1;i++){
                if(arr[min]>arr[i+1]) {
                        min=i+1;   }
      }
        temp =arr [min];
        arr[min]=arr[j];
        arr[j]=temp;
}   }
```

iii) Insertion sort:

It iterates, consuming one input element each repetition, and growing a sorted output list. Each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain

ALGORITHM:

Step 1: Start

Step 2: for i ← 1 to length (A)

Step 3: j ← i

Step 4: while j > 0 and A [j-1] > A[j]

Step 5: swap A [j] and A [j-1]

Step 6: j ← j - 1

Step 7: end while

Step 8: end for

Step 9: Stop

Code: (Iterative)

```
#include <iostream>
using namespace std;
void print(int[],int&); // Function for the Display values
void insertion_sort(int[],int&); //Function for Insertion
int main(){
        int s = 7, arr[s]= {80,70,60,55,50,10,90};
        print(arr,s);
        insertion_sort(arr,s);
        print(arr,s);
```

```
        return 0;  }
void print(int arr[],int&s){
        cout<<"Array is !" <<endl;
        for(int i=0;i<s;i++){
                cout << arr[i]<<"  "; }
        cout << endl; }
void insertion_sort(int arr[],int& s){
        int key , t;
        for(int i=1; i<s; i++){
                key = arr[i] ; // Sorted cells
                t=i-1;
                while(arr[t] > key && t >= 0 ){
                        arr[t+1]=arr[t];
                        t--;
                }
                arr[t+1]=key;
} }
```

iv) Quick Sort:

It is a divide and conquer algorithm. Quick sort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quick sort can then recursively sort the sub-arrays.

ALGORITHM:

Step 1: Pick an element, called a pivot, from the array.

Step 2: Partitioning: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.

Step 3: Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

Code: (Recursive)

```
#include<iostream>

using namespace std;

void quick_Sort(int); // Function for the quick Sort recursive method

void swap(int *a, int *b) { // Function for the swapping using pointers

  int t = *a;   *a = *b;   *b = t;

}

void printArray(int arr[], int size) { // Display Array values

    for (int i = 0; i < size; i++)  { cout << arr[i] << " "; }

    cout << endl; }

int partition(int array[], int low, int high) { // Partion the array and sorting According to it

  int pivot = array[high];

  int i = low ;
```

```
  for (int j = low; j < high; j++) {
    if (array[j] <= pivot) {       // Swapping conditions
     swap(&array[i], &array[j]);
      i++;  }
  }
 swap( &array [i], &array[high]);
  return i ; }  // returning the partitioning point
void quickSort(int array[], int low, int high) {
     if (low < high) {  // Base Condition
    int pi = partition(array, low, high);  // Call Partition method
    quickSort(array, low, pi - 1);  // recursive call on left side of array
    quickSort(array, pi + 1, high); // recursive call on right side of array
  }  }
int main() {
   int n=7;
  int data[n] = {8, 7, 6, 1, 0, 9, 2};
  cout << "Unsorted Array "<<endl;
  printArray(data, n);
  quickSort(data, 0, n - 1);
  cout << "Sorted array in ascending order "<<endl;
  printArray(data, n);
return 0; }
```

3) Implementation Using Arrays

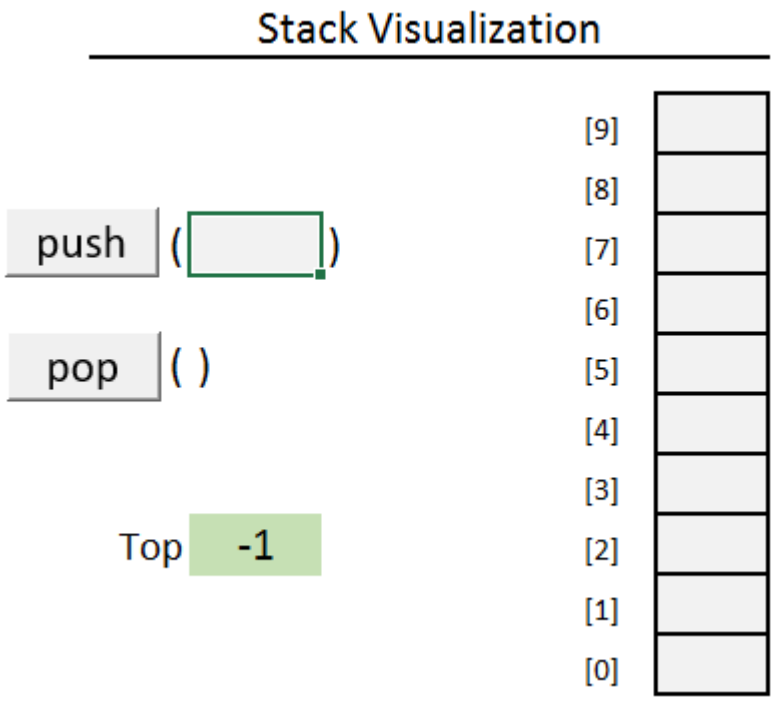
i) Stack ADT:

A stack is an abstract data structure that contains a collection of elements. Stack implements the LIFO mechanism i.e. the element that is pushed at the end is popped out first. Some of the principle operations in the stack are −

Push - This adds a data value to the top of the stack.

Pop - This removes the data value on top of the stack

Peek - This returns the top data value of the stack



ALGORITHM:

push()

Step 1: if top> =max-1 then

Step 2: Display the stack overflows Step

3: else then

Step 4: top ++

Step 5: assign stack [top]=x

Step 6: Display element is inserted

ALGORITHM:

pop()

Step 1: if top = =-1 then

Step 2: Display the stack is underflows

Step 3: else

Step 4: assign x=stack[top]

Step 5: top- -

Step 6: return x

Code: (STACK ADT C++ with basic Functions)

```
#include <iostream>
using namespace std;
int value(){  // Function for taking value from user
        int v;
        cout<< "Enter a Value : ";cin>>v;
        return v; }
class Stack_Array{
        int s;
        int *arr;
        int top;
```

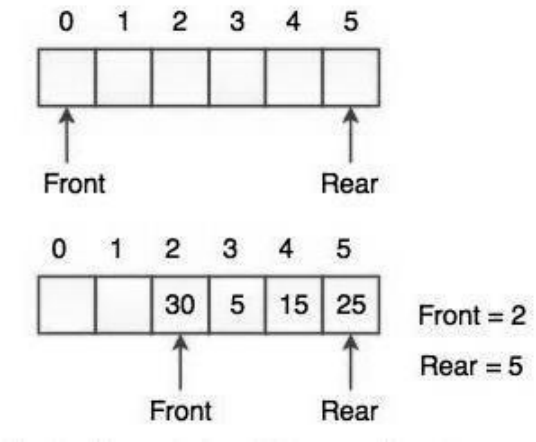
```
        public:
                Stack_Array(int size){        // constructor of Class
                s=size;
                arr = new int(s);              // dynamic allocation of array
                top = -1; }
                Stack_Array(){           // default constructor
                   s = 5;    arr = new int(s);
                   top = -1; }
                void push (int v);      // pushing value in stack
           int peek();                   // give the top value
           void display();              // Display all Stack
           int pop ();                  // delete value from top
};
int main(){
        Stack_Array s ;
        int ch=0;
        char c;
                cout<< " _____"<<endl;
                 cout<< "|     STACK ARRAY     |"<<endl;
                 cout<< "|_____|"<<endl;
                 cout<<endl<<endl;
        while(1){
        cout << "1.EXIT"<<endl;
        cout << "2.PUSH"<<endl;
```

```
cout << "3.POP"<<endl;
cout << "4.PEEK"<<endl;
cout << "5.DISPLAY"<<endl;
cout <<endl;
cout << "Enter the Choice : ";cin>>ch;
switch(ch){
case 1:{
   return 0;
        break; }
case 2:{
        s.push(value());
        break; }
case 3:{
        cout << "Pop Value is : "<<s.pop()<<endl;
        break; }
case 4:{
        cout<< "Peek Value is : "<<s.peek()<<endl;
        break; }
case 5:{
        s.display();
        break; }
default :
cout << "Wrong Input !"<<endl;
}  }
```

```
return 0; }
void Stack_Array::display(){
        if (top==-1){
                cout << "Empty Stack ! "<<endl;
        } else {
        cout << "Stack is "<<endl;
        for (int i = 0;i <= top; i++ ){
        cout << arr[i]<<"  "; }
        cout <<endl<<endl; }
}
void Stack_Array::push(int v) {
        if(top >= s-1){
                cout << "Overflow Stack ! "<<endl;
        } else {
        top++; arr[top] = v; }
}
int Stack_Array::pop() {
        if(top <= -1){
                cout << "Underflow Stack ! "<<endl;
        } else{
        int n = arr[top];   top--;
        return n; }   }
int Stack_Array::peek() {
        return arr[top]; }
```

ii) Queue ADT

Queue implemented using array can store only fixed number of data values. The implementation of queue data structure using array is very simple, just define a one dimensional array of specific size and insert or delete the values into that array by using FIFO (First In First Out) principle with the help of variables 'front' and 'rear'. Initially both 'front' and 'rear' are set to -1. Whenever, we want to insert a new value into the queue, increment 'rear' value by one and then insert at that position. Whenever we want to delete a value from the queue, then increment 'front' value by one and then display the value at 'front' position as deleted element.



- The Front and Rear of the queue point at the first index of the array. (Array index starts from0).
- While adding an element into the queue, the Rear keeps on moving ahead and always points to the position where the next element will be inserted. Front remains at the first index.

ALGORITHM:

Step 1: Start.

Step 2: Initialize front=0; rear=-1.

Step 3: Enqueue operation moves a rear by one position and inserts an element at the rear.

Step 4: Dequeue operation deletes an element at the front of the list and moves the front by one Position.

Step 5: Display operation displays all the element in the list.

Step 6: Stop.

Code: : (QUEUE ADT C++ with basic Functions)

```
#include<iostream>
using namespace std;
class Queue {
        int front;
        int rare;
        int *arr;
        int size;
        public:
                Queue(){
                        size = 5;
                        arr = new int [size];
                        front =-1;
                        rare =-1; }
                Queue(int s){
                        size =s;
                        arr = new int [size];
                        front =-1;
                        rare =-1; }
                void display();              // Display the Queue elements
                void enqueue(int v);     // Put value into Queue
                void dequeue(); // Delete Value from the queue
```

```
                void peek();  // give the front value
};
int main() {
   Queue q;
   int c,a;
    while(1){
                cout << "1.EXIT"<<endl;
                cout << "2.INSERTION"<<endl;
                cout << "3.DELETION"<<endl;
                cout << "4.PEEK"<<endl;
                cout << "5.DISPLAY"<<endl;
      cout << "Enter your Choice : ";cin>>c;
      switch (c) {
                case 1:{
                        return 0;
                        break; }
                case 2:{
                        cout << "Enter Value : ";cin>>a;
                        q.enqueue(a);
                        break; }
                case 3:{
                    q.dequeue();
                        break; }
                case 4:{
```

```
                q.peek();
                    break; }
            case 5:{
                    q.display();
                    break; }
            default :
                    cout << "Wrong Input! "<<endl;
        } }
return 0;}
            void Queue::enqueue(int v){
              if ( ( front==0 && rare == size-1 ) ||  front == rare+1 ) {
                        cout<<"Over flow "<<endl; return ;
                    } else if (front== -1 && rare== -1){
                        front=0; rare =0; arr[rare]=v;
                   } else if (front!=0 && rare == size-1){
                        rare = 0;
                        arr[rare]=v;
                    }  else{
                       rare++; arr[rare]=v; }
            }
            void Queue::dequeue() {
                    if(front == -1 && rare == -1) {
                        cout<<"underflow "<<endl;
                    } else if (front == rare){
```

```
                front =-1; rare=-1;
            } else if( front == size-1 && rare == 0 ){
                front=rare;
            } else if (front == size-1 && rare != 0){
                front=0;
            } else {
                front++; }
}
    void Queue::peek(){
            if ( front == - 1 ) {
                cout<<"Empty Queue !"<<endl;
            } else {
                cout<<"Value at Peek is : " <<arr[front]<<endl;}
}
        void Queue::display(){
            if(front==-1 && rare==-1){
                cout << "Empty Queue"<<endl; return; }
            for(int i=0;i<size;i++){
                cout<< arr[i]<<"  "; }
            cout<<endl;
            cout << "Front is : "<<front+1<<endl;
            cout << "Rear  is : "<<rare+1<<endl;
}
```

4) Implementation of linked list ADT

i) Singly Linked List:

Linked list is a linear data structure. It is a collection of data elements, called nodes pointing to the next node by means of a pointer. In linked list, each node consists of its own data and the address of the next node and forms a chain.

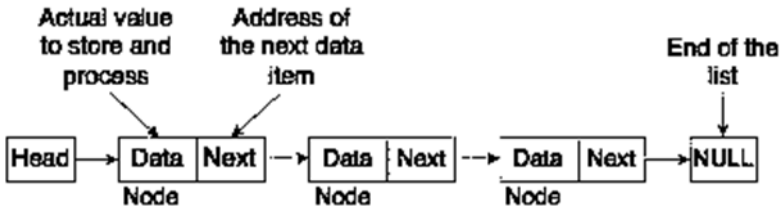
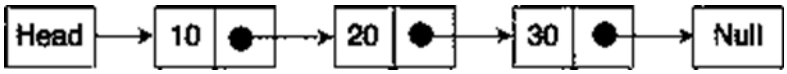


Fig. Linked List

Linked list contains a link element called first and each link carries a data item. Entry point into the linked list is called the **head of the list**. Link field is called next and each link is linked with its next link. Last link carries a link to null to mark the end of the list.

Linked list is a dynamic data structure. While accessing a particular item, start at the head and follow the references until you get that data item.



Linked list contains two fields - First field contains value and second field contains a link to the next node. The last node signifies the end of the list that means NULL. The real life example of Linked List is that of Railway Carriage. It starts from engine and then the coaches follow. Coaches can traverse from one coach to other, if they connected to each other.

ALGORITHM:

Step 1: Start

Step 2: Creation: Get the number of elements, and create the nodes having structures DATA, LINK and store the element in Data field, link them together to form a linked list.

Step 3: Insertion: Get the number to be inserted and create a new node store the value in DATA field. And insert the node in the required position.

Step 4: Deletion: Get the number to be deleted. Search the list from the beginning and locate the node then delete the node.

Step 5: Display: Display all the nodes in the list.

Step 6: Stop.

Code: (SINGLY LINKED LIST ADT C++)

```
#include <iostream>
using namespace std;
struct node{    // Linked List Node Structure
        int data;
        node* next;
};
class LinkedList{     // Singly Linked List Class
        node *start;
        public:
                LinkedList(){  // Constructor of class
                        start=NULL; }
                node* create_node(int);         // create the node dynamically and return address
                void insert_start();              // Insert the node at start of list
                void display();                   // Display all data of list
                void insert_end();               // Insert at last of list
                void linear_search();           // Search in list
                int  count_node();              // count no of nodes in list
                void sort_bubble();           // Sort the list data
                void insert_after_given_node_num();  // insert the node after the given position
                void insert_sorted_list();  // insertion in the sorted list
```

```
        void delete_pos();          // delete at specific position
        void reverse_list();   // Reverse the list data
};
int main(){
    int ch=0;
    char c;
    LinkedList l;
        cout<< " _____"<<endl;
        cout<< "|  SINGLY LINKED LIST   |"<<endl;
        cout<< "|_____|"<<endl;
       cout<<endl<<endl;
    while(1){
    cout << "1.Display"<<endl;
    cout << "2.Insert at Start"<<endl;
    cout << "3.Insert at End"<<endl;
    cout << "4.Sortion Assending"<<endl;
    cout << "5.Linear Search"<<endl;
    cout << "6.Count Nodes"<<endl;
    cout << "7.Insert after given no of nodes"<<endl;
    cout << "8.Insert In Sorted List"<<endl;
    cout << "9.Delete Node at Position"<<endl;
    cout << "10.Revesr Linked List"<<endl;
    cout << "11.EXIT PROGRAM"<<endl;
    cout <<endl;
```

```
cout << "Enter the Choice : ";cin>>ch;
switch(ch){
        case 1: {
        l.display(); break; }
case 2:{
        l.insert_start(); break; }
case 3:{
        l.insert_end(); break; }
case 4:{
        l.sort_bubble(); break; }
case 5:{
        l.linear_search(); break; }
case 6:{
        cout << "Number of Nodes are : "<<l.count_node()<<endl; break; }
case 7:{
        l.insert_after_given_node_num(); break; }
case 8:{
        l.insert_sorted_list(); break; }
case 9:{
        l.delete_pos(); break; }
case 10:{
        l.reverse_list(); break; }
case 11:{
        return 0; break; }
```

```
        default :
                cout << "Wrong Input !"<<endl;
        }
} return 0 ;}


node* LinkedList::create_node(int v){
        struct node *temp,*s;
        temp = new (struct node);
        if (temp == NULL){
        cout << "Memory is not Allocated !"<<endl;
        return 0; }
        else{
        temp->data=v;
        temp->next=NULL;
        return temp; }
}
void LinkedList::insert_start(){
        int n;
        cout << "Enter the value to insert at Start : ";cin>>n;
        struct node *temp,*s;
        temp = create_node(n);
        if (start==NULL){
                start= temp; }
        else{
```

```
                s = start;
                start=temp;
            start->next = s ; }
}
void LinkedList::display(){
        struct node *temp;
        temp = start;
        cout << "Linked List is !"<<endl;
        if (start==NULL){
                cout << "There is no Linked List in Data"<<endl;
        } else{
         while(temp!=NULL){
                cout << temp->data<<"  ";
                 temp = temp->next; } }
        cout <<endl;
}
void LinkedList::insert_end(){
        int v;
        cout << "Enter the value to Insert at End : ";cin>>v;
        struct node *temp,*s;
        temp = create_node(v);
        s=start;
        if (start==NULL){
                start=temp; }
```

```
        else{
        while (s->next!=NULL){
                s=s->next; }
        s->next=temp; }
}
void LinkedList::linear_search(){
        int n,c=0,ps=0;
        struct node *temp;
        cout << "Enter Value to Search : ";cin>>n;
        if(start==NULL){
                cout << "LinkedList is Empty !"<<endl;
        } else {
                temp=start;
                ps=0;c=0;
                while (temp!=NULL){
                        ps++;
                if(temp->data == n){
                cout << temp->data << " : is in the Linked List at Postion : "<<ps<<endl;
                c=1;break;
                } else{
                        temp=temp->next; }
                }
                if (c==0){
                        cout << "Number is Not found "<<endl; }
```

```
        } }
int LinkedList::count_node(){
        if (start==NULL){
                return 0; }
        else {
        node *temp=start;
        int n=0;
        while(temp!=NULL){
                n++; temp=temp->next;  }
        return n; }
}
void LinkedList::sort_bubble(){
        int s = count_node();
        if(s==0){
                cout << "List is Empty"<<endl; }
        else{
                node *loc, *locn ;
                int d;
        for(int i=0;i<s-1;i++){
                loc = start;
                locn = start->next;
                for(int j=1;j < s-i;j++){
                        if(loc->data > locn->data){
                                d=loc->data;
```

```
                                loc->data=locn->data;
                                locn->data=d; }
                        loc = loc->next;
                        locn = locn->next;
                }  }
        cout << "Sorted ";
        display();
}       }
void LinkedList::insert_after_given_node_num(){
        int n,s,v;
        node *New;
        cout << "Enter Insertion position of node : ";cin>>n;
        cout<< "Enter the Value to Input : ";cin>>v;
        s=count_node();
        if(n>s+1){
                cout << "Error to Insert at that point"<<endl;
        } else if (start==NULL){
                New = create_node(v);
                start=New;
        } else if (start!=NULL && n==1){
                New = create_node(v);
                New->next=start;
                start=New;
        } else if (n>1){
```

```
                node *temp,*New,*locp;
                temp=start;locp=NULL;
                for(int a=1;a<n;a++){
                        locp=temp;
                        temp=temp->next; }
                New= create_node(v);
                locp->next=New;
                New->next=temp;
}}
void LinkedList::insert_sorted_list(){
        int v;
        cout << "Enter the value : ";cin>>v;
        if(start==NULL){
                start=create_node(v);
        } else {
                node *temp,*locp,*New;
                New = create_node(v);
                temp=start;
                locp=NULL;
                while(temp!=NULL){
                        if(start->data>=v){
                                New->next=start;
                                start=New;
```

```
                temp=NULL;
            } else if (temp->next==NULL){
                temp->next=New;
                New->next=NULL;
                temp=NULL;
            } else if(temp->data >= v ){
                New->next=temp;
                locp->next=New;
                temp=NULL;
            } else {
            locp=temp;
            temp=temp->next; }
} } }
void LinkedList::delete_pos(){
    if(start==NULL){
        cout << "Linked List Is Empty !"<<endl;
    } else {
        int p, s = count_node() ;
        cout << "Enter Position to Delete : ";cin>>p;
        if(p>s){
            cout << "Error to Delete ! "<<endl;
        } else if (p==1){
            start=start->next;
        } else {
```

```
                node *temp,*locp=NULL;
                temp=start;
                        for (int i=1;i<p;i++){
                                locp=temp;
                                temp=temp->next; }
                        locp->next=temp->next;
} } }
void LinkedList::reverse_list(){
        int s=count_node();
        if(s==0){
                cout << "There is no Linked list in Data !"<<endl;
        } else if (s==1) {
                return ;
        } else {
                int arr[s],a=0; node *temp; temp=start;
                while (temp!=NULL){
                        arr[a]=temp->data;
                        temp=temp->next;
                        a++; }
                temp=start;
                for(int i=s-1;i>=0;i--){
                        temp->data=arr[i];
                        temp=temp->next;  }}}
```

ii) Doubly Linked List:

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List. Following are the important terms to understand the concept of doubly linked list.

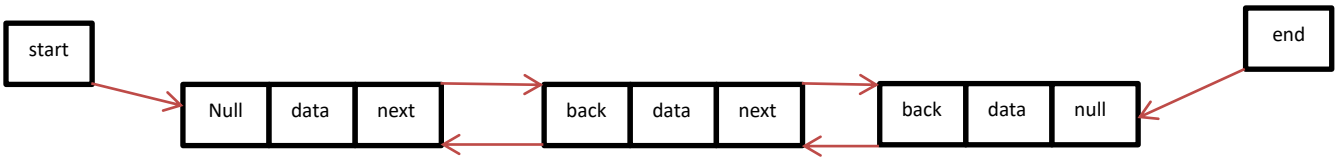
Data − Each data of a linked list can store a data called an element.

Next − Each link of a linked list contains a link to the next link called Next.

Back − Each link of a linked list contains a link to the previous link called back.

Linked List − A Linked List contains the connection link to the first link called First and to the last link called Last.

Doubly Linked List Representation



- Doubly Linked List contains a link element called start and end.
- Each link carries a data field(s) and two link fields called next and back.
- Each link is linked with its next link using its next link.
- Each link is linked with its previous link using its back link.
- The last link carries a link as null to mark the end of the list.

Basic Operations:

Insertion − Adds an element at the beginning of the list.

Deletion − Deletes an element at the beginning of the list.

Insert Last − Adds an element at the end of the list.

Delete Last − Deletes an element from the end of the list.

Insert After − Adds an element after an item of the list.

Delete − Deletes an element from the list using the key.

Display forward − Displays the complete list in a forward manner.

Display backward − Displays the complete list in a backward manner.

Code: (DOUBLY LINKED LIST ADT C++)

```
#include <iostream>
using namespace std;
struct node {
        int data;
        node* next;
        node* back;
};
class DobbleyLinkedList{
        node * start,* end;
        public:
                DobbleyLinkedList(){
                        start =NULL;
                        end=NULL; }
                int value();
                node* create_node(int n);       // Create a doubly liked list node and return address
                void insert_at_start();         //Insert node at start
                void insert_at_end();           // insert node at last
                void display();                 // Display the values in both forward and backward
```

```
        int  count_node();              // count nodes in a list
        void delete_at_start();         // delete a node from start
        void delete_at_end();           // delete a node from end
        void sort_bubble();             // sort the data in list
        void inser_multiple();          // multiple nodes inserted in a list
        void insert_sorted();           // insert in a list in sorted manner
        void insert_pos();              // Insert a node after at given position
};
int main(){
    DobbleyLinkedList l;
    int c;
        cout<< " _____"<<endl;
        cout<< "|   Doubley LinkedList  |"<<endl;
        cout<< "|_____|"<<endl;
        cout<<endl<<endl;
    while(1){
    cout << "01.EXIT PROGRAM"<<endl;
    cout << "02.DISPLAY"<<endl;
    cout << "03.INSERT AT FIRST"<<endl;
    cout << "04.INSERT AT END"<<endl;
    cout << "05.INSERT SORTED LIST"<<endl;
    cout << "06.DELETE AT START"<<endl;
    cout << "07.DELETE AT END"<<endl;
    cout << "08.SORTING LINKED LIST"<<endl;
```

```
cout << "09.INSERT MULTIPLE NODES"<<endl;
cout << "10.COUNT NODES IN LINKED LIST"<<endl;
cout << "11.INSERT AT POSITION"<<endl;
cout <<endl;
cout << "Enter the Choice : ";cin>>c;
switch(c){
case 1:{
        return 0; }
case 2: {
        l.display(); break; }
case 3: {
        l.insert_at_start(); break; }
case 4: {
        l.insert_at_end(); break; }
case 5:{
        l.insert_sorted(); break; }
case 6:{
        l.delete_at_start(); break; }
case 7:{
        l.delete_at_end(); break; }
case 8:{
        l.sort_bubble(); break; }
case 9:{
        l.inser_multiple(); break; }
```

```
        case 10:{
                cout << "Number of Nodes are : "<<l.count_node()<<endl; break; }
        case 11:{
                l.insert_pos(); break; }
        default :
                cout << "Wrong Input !"<<endl;
} }
return 0; }
int DobbleyLinkedList::value(){
        int n; cout << "Enter a Value : ";cin>>n;
   return n;
}
node* DobbleyLinkedList::create_node(int n){
        node* t =NULL;
        t= new node;
        if (t==NULL){
                cout << "Node is Not Created !"<<endl;
                return NULL;
        } else {
         t->data=n;
        t->back=NULL;  t->next=NULL;
        return t;
}
}
```

```
void  DobbleyLinkedList::display(){
        node *t;
        if(start==NULL){
                cout << "Linked List is Empty ! "<<endl;
        } else {
        cout << "Doubly Linked List Data "<<endl;
        for (t=start ; t!=NULL ; t=t->next ){
                cout << t->data<<"  ";
        } cout << endl;
        cout << "Doubly Linked List Data reverse "<<endl;
        for (t=end ; t!=NULL ; t=t->back ){
                cout << t->data<<"  ";
        } cout <<endl;}
}
void DobbleyLinkedList::insert_at_start(){
        if(start==NULL && end==NULL){
                start = create_node(value());
                end=start;
        } else {
                node *t=create_node(value());
                t->next=start;
                start->back=t;
                start = t; }
}
```

```
void DobbleyLinkedList::insert_at_end(){
        if(start==NULL && end==NULL){
                start = create_node(value());
                end=start;
        } else {
                node *t=create_node(value());
                end->next=t;
                t->back=end;
                end=t;
                end->next=NULL;       }
}
int DobbleyLinkedList::count_node(){
        int n=0;
        node *t;
        for (t=start ; t!=NULL ; t=t->next ){              n++;       }
        return n;
}
void DobbleyLinkedList::delete_at_start(){
        if (start==NULL && end==NULL){
                cout << "Linked List UnderFlow ! "<<endl;
        } else if (start==end){
                start=NULL;
                end=NULL;
        } else {
```

```
                start=start->next;
                start->back=NULL;     }
}
void DobbleyLinkedList::delete_at_end(){
                if (start==NULL && end==NULL){
                cout << "Linked List UnderFlow ! "<<endl;
        } else if (start==end){
                start=NULL;
                end=NULL;
        } else {
                end=end->back;
                end->next=NULL;       }
}
void DobbleyLinkedList::sort_bubble(){
        if(start==NULL){
                cout << "Empty Linked List "<<endl;
        } else{
                node   *t, *s;
                int temp , check;
                s = start;
                while( s != NULL){
                        t = s->next;
                        check = 0;
                       while( t != NULL){
```

```
                        if ( s->data > t->data ) {
                                temp = s->data;
                                s->data = t->data;
                                t->data = temp;
                                check++;  }
                        t = t->next;
                        }
                        if(check == 0){
                                return ; }
                        s = s->next;
                }
        }
}
void DobbleyLinkedList::insert_sorted(){
        if(start==NULL){
                start = create_node(value());
                end = start;
        } else {
                int v = value();
                node *New =  create_node(v);
                node *t=start;
                  if( t->data > v){
                    New->next=start;
                        New->back=start->back;
```

```
                start->back = New;
                start = New;
            } else {
            while(t!=NULL){
                if( t->data > v){
                    break;
            }
         t=t->next;
        }
        if(t==NULL){
            t=end;
            New->back=t;
            New->next=t->next;
                t->next=New;
                end = New;
            } else{
                t=t->back;
                New->back=t;
            New->next=t->next;
                t->next=New;
                New->next->back=New;}
        }
    }
}
```

```
void DobbleyLinkedList::inser_multiple(){
   int n;
        cout << "Enter Number of Nodes to create : ";cin>>n;
        if(n >= 0){
        while( n!=0){
                insert_at_end();
                n--; }
        } else{
                cout << "Invalid Input "<<endl;}
}
void DobbleyLinkedList::insert_pos(){
        int p=0,s;
        cout << "Enter the position : ";cin>>p;
        s=count_node();
        if ( s+1 >= p && p>0) {
                node *t = start;
                node *temp = create_node(value());
                int pcheck = 1;
                if(start==NULL){
                        start = temp;
                        end = start;
                } else if (p==1){
                        temp->next=start;
                        start->back=temp;
```

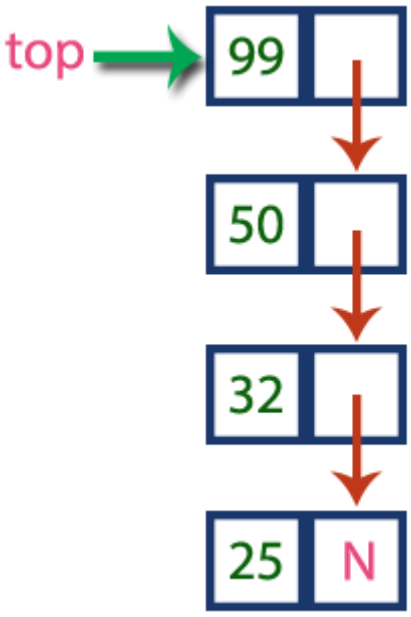
```
                start=temp;
            } else if (p == s+1) {
                temp->back=end;
                end->next=temp;
                end=temp;
            } else {
            while (t != NULL && pcheck<p-1){
                t=t->next;
                pcheck++; }
            temp->back=t;
            temp->next=t->next;
            t->next->back=temp;
            t->next=temp; }
        } else{
            cout<<"Position is Invalid !"<<endl;
    }
}
```

5) Implementation of linked list

i) STACK ADT:

The major problem with the stack implemented using an array is, it works only for a fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself. Stack implemented using an array is not suitable, when we don't know the size of data which we are going to use. A stack data structure can be implemented by using a linked list data structure. The stack implemented using linked list can work for an unlimited number of values. That means, stack implemented using linked list works for the variable size of data. So, there is no need to fix the size at the beginning of the implementation. The Stack implemented using linked list can organize as many data values as we want.

In linked list implementation of a stack, every new element is inserted as '**top**' element. That means every newly inserted element is pointed by '**top**'. Whenever we want to remove an element from the stack, simply remove the node which is pointed by 'top' by moving 'top' to its previous node in the list. The next field of the first element must be always **NULL**.



Code (STACK ADT Linked list based in C++)

```
#include <iostream>
using namespace std;
struct node{
        int data;
        node *next;
};
class stackLinkedList{
       node   *top;
        public:
                stackLinkedList(){
                        top=NULL;
                }
                int value();                    // Take value from user and return it
                node* create_node(int);     // Create a list node and return address
                void Display();                 // Display stack data
                void push();                   // Push a value into stack
                void pop();                     // Delete a value from stack top
};
int main(){
                stackLinkedList s; int c;
                cout<< " _____"<<endl;
                cout<< "|    Stack LinkedList   |"          <<endl;
                cout<< "|_____|"<<endl;
```

```
        cout<<endl<<endl;
    while(1){
    cout << "1.EXIT PROGRAM"<<endl;
    cout << "2.DISPLAY"<<endl;
    cout << "3.PUSH"<<endl;
    cout << "4.POP"<<endl;
    cout <<endl;
    cout << "Enter the Choice : ";cin>>c;
    switch(c){
    case 1: {
            return 0; }
    case 2:{
            s.Display(); break; }
    case 3: {
            s.push(); break; }
    case 4: {
            s.pop(); break; }
    default :
            cout << "Wrong Input !"<<endl;
    }
}
return 0; }
int stackLinkedList::value(){
    int n;
```

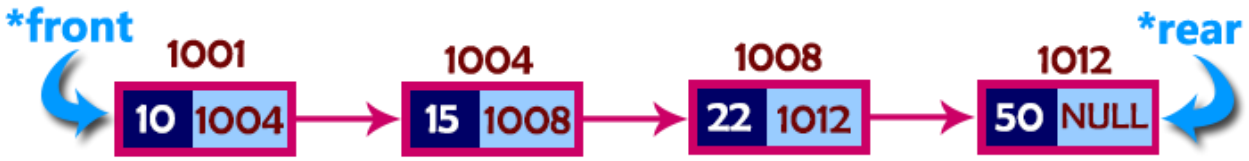
```
        cout << "Enter a Value : ";cin>>n;
    return n;
}
node* stackLinkedList::create_node(int n){
        node* t =NULL;
        t= new node;
        if (t==NULL){
                cout << "Node is Not Created !"<<endl;
                return NULL;
        } else {
        t->data=n;
        t->next=NULL;
        return t;}
}
void stackLinkedList::push(){
                if(top==NULL){
                top = create_node(value());
        } else {
                node *t=create_node(value());
                t->next=top;
                top = t; }
}
void stackLinkedList::pop(){
        if (top==NULL ){
```

```
                cout << "Stack LinkedList UnderFlow ! "<<endl;
        } else {
                top=top->next;          }
}
void stackLinkedList::Display(){
        node *t;
        if(top==NULL){
                cout << "Stack is Empty ! "<<endl;
        } else {
        cout << "Stack Likedlist Data "<<endl;
        for (t=top ; t!=NULL ; t=t->next ){
                cout << t->data<<"  ";
        }
        cout << endl;
}
}
```

ii) QUEUE ADT:

The major problem with the queue implemented using an array is, It will work for an only fixed number of data values. That means, the amount of data must be specified at the beginning itself. Queue using an array is not suitable when we don't know the size of data which we are going to use. A queue data structure can be implemented using a linked list data structure. The queue which is implemented using a linked list can work for an unlimited number of values. That means, queue using linked list can work for the variable size of data (No need to fix the size at the beginning of the implementation). The Queue implemented using linked list can organize as many data values as we want.

In linked list implementation of a queue, the last inserted node is always pointed by '**rear**' and the first node is always pointed by '**front**'.



Code: (QUEUE ADT Linked list based in C++)

```
#include <iostream>

using namespace std;

struct Node{

        int data;

        Node *next;

        Node(int d){

                data = d;

                next =NULL; }

};

class Queue{
```

```
Node *rear;
Node *front;
int value(){            // takes value from the user
        int v;
        cout<< "Enter the Value : ";cin>>v;
        return v; }
public :
        Queue(){            // Constructor for giving null to the rear and front pointers
                front = NULL;
                rear = NULL;
        }
        void enqueue(){    // Function for insertion in queue
                if(front==NULL){
                        front = new Node(value());
                        rear =  front;
                } else {
                        Node *temp = new Node(value());
                        rear->next = temp;
                        rear = temp;  }
        }
        void dequeue(){   // Function for deleting the value form queue
                if(front==NULL){
                        cout<< "Under Flow Queue"<<endl;
                } else{
```

```
                cout << "Dequeue Value is : "<<front->data<<endl;
                front = front->next;
                if(front == NULL){
                    rear = NULL; }
            }
        }
        void peek(){            // Function for giving value at front
            if (front==NULL){
                cout << "Empty Queue "<<endl;
            } else{
                cout<< "Peek Value is : "<<front->data<<endl;  }
        }
        void display(){         // Function for display all values in queue
            if(front==NULL){
                cout <<"Empty Queue"<<endl;
                return ; }
            cout << "Queue Linked List"<<endl;
            Node *temp = front;
            while(temp!=NULL){
                cout << temp->data<<"   ";
                temp = temp->next; }
            cout<<endl;   }
};
int main(){
```

```
        Queue q;   int c,a;
while(1){
                cout << "1.EXIT"<<endl;
                cout << "2.ENQUEUE"<<endl;
                cout << "3.DEQUEUE"<<endl;
                cout << "4.PEEK"<<endl;
                cout << "5.Display"<<endl;
               cout << "Enter your Choice : ";cin>>c;
switch (c) {
                case 1:{
                        return 0; }
                case 2:{
                        q.enqueue(); break; }
                case 3:{
                        q.dequeue(); break; }
                case 4:{
                   q.peek(); break; }
                case 5:{
                        q.display(); break; }
                default :{
                        cout << "Wrong Input! "<<endl; break; }
} }
return 0; }
```

iii) PIRORITY QUEUE:

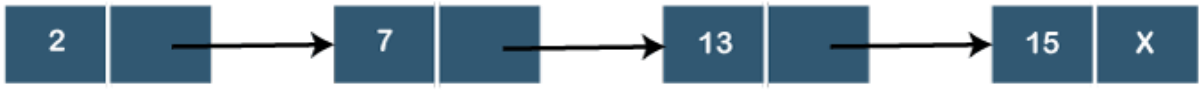
A priority queue is a type of queue in which each element in a queue is associated with some priority, and they are served based on their priorities. If the elements have the same priority, they are served based on their order in a queue.

Mainly, the value of the element can be considered for assigning the priority. For example, the highest value element can be used as the highest priority element. We can also assume the lowest value element to be the highest priority element. In other cases, we can also set the priority based on our needs.

The following are the functions used to implement priority queue using linked list:

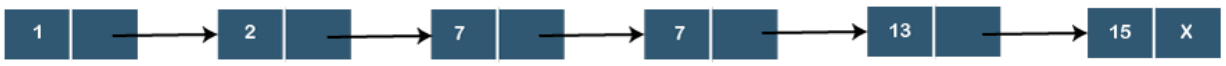
- **push( )** : It is used to insert a new element into the Queue.
- **pop( )** : It removes the highest priority element from the Queue.
- **peek( )** : This function is used to retrieve the highest priority element from the queue without removing it from the queue.

Consider the below-linked list that consists of elements 2, 7, 13, 15.



Suppose we want to add the node that contains the value 1. Since the value 1 has more priority than the other nodes so we will insert the node at the beginning of the list shown as below:

Now we have to add 7 element to the linked list. We will traverse the list to insert element 7. First, we will compare element 7 with 1; since 7 has lower priority than 1, so it will not be inserted before 7. Element 7 will be compared with the next node, i.e., 2; since element 7 has a lower priority than 2, it will not be inserted before 2.. Now, the element 7 is compared with a next element, i.e., since both the elements have the same priority so they will be served based on the first come first serve. The new element 7 will be added after the element 7 shown as below:



Code (PRIORITY QUEUE ADT Linked list based in C++)

```
#include <iostream>
using namespace std;
struct Node{
	char data;
	int priority;
	Node *next;
	Node(int d,int p){
		priority = p;
		data = d;
		next =NULL; }
};
class PriorityQueue{
	Node *rear;
	Node *front;
	char value(){   // Taking data value from the user
		char v;
		cout<< "Enter the Value : ";cin>>v;
		return v;
	}
	int priorityValue(){   //  Taking priority value from the user
		int p;
		cout <<"Enter the Priority : ";cin>>p;
		return p;
```

```
}
public :
    PriorityQueue(){  // Constructor
        front = NULL;
        rear = NULL;
    }
    void enqueue(){  // Inserting value in queue according to given priority
        if(rear==NULL){
            rear = new Node(value(),priorityValue());
            front = rear;
            return;
        }
        Node *ptr = front;
        Node *New =new Node(value(),priorityValue());
        Node *save;
        if(New->priority<front->priority){
            New->next = front;
            front = New;
            return ;
        }
        while (ptr!=NULL){
         if(ptr->priority<=New->priority){
            save = ptr;
            ptr = ptr->next;
```

```
                }else if(ptr->priority>New->priority){
                        New->next = ptr;
                        save->next = New;
                        return;
                }
        }
    save->next = New;
        }
        void dequeue(){  // Deleting value from the queue
                if(front==NULL){
                        cout<< "Under Flow Queue"<<endl;
                } else{
                        cout << "Dequeue Value is : "<<front->data<<endl;
                        front = front->next;
                        if(front == NULL){
                                rear = NULL;  }
                }
    }
        void peek(){  // Value with highest priority
                if (front==NULL){
                        cout << "Empty Queue "<<endl;
                } else{
                        cout<< "Peek Value is : "<<front->data<<endl; }
        }
```

```
        void display(){        // display all values in queue with its priority
                if(front==NULL){
                        cout <<"Empty Queue"<<endl;
                        return ;
                }
                cout << "Queue Linked List"<<endl;
                Node *temp = front;
                while(temp!=NULL){
                        cout <<"[ "<< temp->data<<" , "<<temp->priority<<" ]  ";
                        temp = temp->next;
                }
                cout<<endl;
        }
};
int main(){
        PriorityQueue q ;
        int c,a;
while(1){
                cout << "1.EXIT"<<endl;
                cout << "2.ENQUEUE"<<endl;
                cout << "3.DEQUEUE"<<endl;
                cout << "4.PEEK"<<endl;
                cout << "5.Display"<<endl;
      cout << "Enter your Choice : ";cin>>c;
```

```
switch (c) {
                case 1:{
                        return 0; }
                case 2:{
                        q.enqueue(); break; }
                case 3:{
                        q.dequeue(); break; }
                case 4:{
                   q.peek(); break; }
                case 5:{
                        q.display(); break; }
                default :{
                        cout << "Wrong Input! "<<endl;
                } }
 }
        return 0; }
```

6) Binary Tree (ADT) Implementation:

In the binary tree, each node can have at most two children. Each node can have zero, one or two children. Each node in the binary tree contains the following information:

Data that represents value stored in the node.

Left that represents the pointer to the left child.

Right that represents the pointer to the right child.

Algorithm

1. Define Node class which has three attributes namely: data left and right. Here, left represents the left child of the node and right represents the right child of the node.
2. When a node is created, data will pass to data attribute of the node and both left and right will be set to null.
3. Define another class which has an attribute root.
    a. Root represents the root node of the tree and initialize it to null.
4. insert() will add a new node to the tree:
    a. It checks whether the root is null, which means the tree is empty. It will add the new node as root.
    b. Else, it will add root to the queue.
    c. The variable node represents the current node.
    d. First, it checks whether a node has a left and right child. If yes, it will add both nodes to queue.
    e. If the left child is not present, it will add the new node as the left child.
    f. If the left is present, then it will add the new node as the right child.
5. traversal( ) will display nodes of the tree in following fashion.
    a. preorder traverses the entire tree then prints out root followed by left child then followed by the right child.
    b. inorder traverses the entire tree then prints out left child followed by root then followed by the right child.
    c. postorder traverses the entire tree then prints out left child followed by right child then followed by the root child.

Code:

```
#include <iostream>
#include <queue>
using namespace std;
struct Node {
	int data;
	Node *left;
	Node *right;
	Node(int d){
		data = d;
		left=NULL;
		right=NULL; }
};
class BinaryTree{
	Node* root;
	int value(){
		int n;cout<<"Enter the data : ";cin>>n;
		return n;
	}
	public:
		BinaryTree(){  // Constructor
			root = NULL;
		}
	void preOrder (Node * n);  // preorder function
```

```
        void inOrder  (Node * n); //inorder function
        void postOrder(Node * n);  // post order function
        void traversal();    //function for the traversals choice
        void insert();   // function for insertion in Binary tree (left then right)
};
int main(){

        int choice;
        BinaryTree b;
                cout<< " _____"<<endl;
                 cout<< "|     Binary Tree       |"<<endl;
                 cout<< "|_____|"<<endl;
                 cout<<endl<<endl;
        while(1){
        cout << "1.EXIT"<<endl;
        cout << "2.INSERT"<<endl;
        cout << "3.TRAVERSALS"<<endl;
        cout <<endl;
        cout << "Enter the Choice : ";cin>>choice;
        switch(choice){
                 case 1: {
                 return 0;
        }
        case 2:{
```

```
                b.insert(); break; }
        case 3:{
                b.traversal(); break; }
        default :
        cout << "Wrong Input !"<<endl;
} }
return 0;}
        void BinaryTree::traversal(){
                if(root==NULL){
                        cout << "No tree in Memory !"<<endl;
                        return;
                }
                int ch;
                cout << "1.PreOrder"<<endl;
                cout << "2.InOrder"<<endl;
                cout << "3.PostOrder"<<endl;
                cout << "Enter the choice : ";cin>>ch;
                switch (ch){
                        case 1:{
                                cout << "Pre Order traversal ! "<<endl;
                                preOrder(root);
                                cout<<endl; break; }
                        case 2:{
                                cout << "In Order traversal ! "<<endl;
```

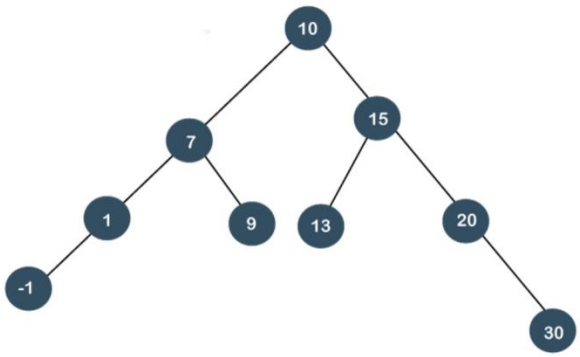
```
                inOrder(root);
                cout <<endl; break; }
        case 3:{
                cout << "Post Order traversal ! "<<endl;
                postOrder(root);
                cout<<endl; break; }
        default:{
                cout<< "Wrong choice ! "<<endl; }
}
}
void BinaryTree::preOrder(Node *n){
        if(n==NULL){
                return; }
        cout << n->data <<"  ";
        preOrder(n->left);
        preOrder(n->right);
}
void BinaryTree::inOrder(Node *n){
        if(n==NULL){
                return; }
        inOrder(n->left);
        cout<< n->data <<"  ";
        inOrder(n->right);
}
```

```
void BinaryTree::postOrder(Node *n){
        if(n==NULL){
                return; }
        postOrder(n->left);
        postOrder(n->right);
        cout<< n->data <<"  ";
}
void BinaryTree::insert(){
        if(root==NULL){
                root = new Node(value());
                return ; }
        queue<Node*> q;
        Node *n=root;
        while(n!=NULL){
          q.push(n->left);  q.push(n->right);
          if(n->left==NULL){
          n->left=new Node(value());  return ;
          }else if(n->right==NULL){
          n->right=new Node(value());  return ;
          }else{
                n = q.front();
                q.pop();
    } } }
```

7) Binary Search Tree (ADT) Implementation:

Binary tree is a special case of trees where each node can have at most 2 children. Also, these children are named: left child or right child. A very useful specialization of binary trees is binary search tree (BST) where nodes are conventionally ordered in a certain manner.

- The left subtree has less values than current node
- The right subtree has greater values than current node



Advantages of Binary search tree

- Searching an element in the Binary search tree is easy as we always have a hint that which subtree has the desired element.
- As compared to array and linked lists, insertion and deletion operations are faster in BST.

Basic Operations in BST

1. Insertions
2. Deletions
3. Searching
4. Traversals

Code:

```
#include <iostream>
using namespace std;
struct Node{
	int data;
	Node *left;
	Node *right;
	Node(int d){
		data  = d;
		left  = NULL;
		right = NULL; }
};
class BST{
	Node *root;
	Node *loc;
	Node *locp;
	int value(){
		int v; cout << "Enter the Value : ";cin>>v;
		return v; }
	void find(int d){      // Function to find a specific value in BST
		loc = NULL;
		locp = NULL;
		if(root==NULL){       // loc=null,lop=null
			cout<<"Empty tree"<<endl;
```

```
                return; }
        if (root->data == d){  // loc=root,locp=null
        loc = root;
        cout<<"Value found"<<endl;
        return; }
                Node *temp = root , *save;
                while(temp!=NULL){  // loc !=null,locp!=null
                        if(temp->data==d){
                                locp = save;
                                loc = temp;
                                cout<<"Value found"<<endl;
                                return; }
                        if(d<temp->data){
                                save  = temp;
                                temp = temp->left;
                        } else {
                                save  = temp;
                                temp = temp->right; }
}
                cout<<"Value Not found"<<endl;
                locp = save;  loc = NULL; }
        void postorder(Node *n){  // recursive postorder function
                if(n==NULL){
                        return; }
```

```
        postorder(n->left);
        postorder(n->right);
        cout << n->data <<"  "; }
void preorder(Node *n){ // recursive preorder function
        if(n==NULL){
                return; }
        cout<< n->data <<"  ";
        preorder(n->left);
        preorder(n->right); }
void inorder(Node *n){ // recursive inorder function
        if(n==NULL){
                return; }
        inorder(n->left);
        cout<< n->data <<"  ";
        inorder(n->right); }
Node *insert(Node *n,int d){  // recursive BST insertion function
        if(n==NULL){
                return new Node(d); }
        else {
                if(d < n->data ){
                        n->left  = insert(n->left,d);
                } else{
                        n->right  = insert(n->right,d); } }
        return n;  }
```

```
        Node *search(int d){  // Iterative BST search function
                if(root==NULL){
                        cout<< "No Tree in Memory"<<endl;
                        return 0; }
                Node *temp = root;
                while( temp!=NULL  ){
                        if(temp->data == d){
                                return temp; }
                        if(d<temp->data){
                                temp = temp->left; }
                        else{
                                temp = temp->right; }
        }
                return NULL; }
        Node *getMinimumKey(Node* n){   // Get inOrder predecessor of  node
                while (n->left!=NULL){
                n = n->left;  }
  return n; }
        void deleteNode(Node* root, int key) {  // Delete node using find function
  // search key in the BST and set its parent pointer
  find(key);
  // return if the key is not found in the tree
  if (loc == NULL) {
     return; }
```

```
// Case 1: node to be deleted has no children, i.e., it is a leaf node
if (loc->left == NULL && loc->right == NULL) {
    // if the node to be deleted is a root node, then set it to null
    if (loc == root) {
        root=NULL;
    } else {
        if (locp->left == loc) {
            locp->left = NULL;
        } else {
            locp->right = NULL;  }
  }  }  else if (loc->left && loc->right) {   // Case 2: node to be deleted has two children
    // find its inorder successor node
    Node* successor = getMinimumKey(loc->right);
    Node* curr = loc;
    int val = successor->data; // store successor value
    // recursively delete the successor. Note that the successor
    // will have at most one child (right child)
    deleteNode(root, successor->data);
                curr->data = val;   // copy value of the successor to the current node
} else { // Case 3: node to be deleted has only one child
    // choose a child node
    Node* child ;
                if(loc->left){
        child = loc->left;
```

```
                }else{
                        child  =loc->right; }
        // if the node to be deleted is not a root node, set its parent
        // to its child
                if (loc != root){
            if (loc == locp->left) {
                locp->left = child;
            } else {
                locp->right = child;   }
        } else {
            root = child;}
    }
}
            public:
            BST(){  //constructor
                root = NULL;
                loc = NULL;
                locp = NULL;
            }
            void insertTree( ){ // Insertion using a find function
                int d = value();
                find(d);
                if(loc==NULL && locp==NULL){
```

```
                root = new Node(d);
            } else if(locp != NULL && loc == NULL){
                if(d<locp->data){
                    locp->left = new Node (d);
                } else{
                    locp->right = new Node (d); }
            } else{
                cout << "Value Already in Tree"<<endl; }
}
    void insertRecurtion(){     // recursion insert node in BST
    root =  insert(root,value());
    }
    bool searchTree( ){  // Display search value
        Node *n  = search(value());
        if (n){
            cout<< "Value found which is : "<<n->data<<endl;
            return true;
        }
        return false;}
    void deleteTree(){
        deleteNode(root,value());
    }
    void traversal(){  // Traversal choice function
        if(root==NULL){
```

```
                cout << "No tree in Memory !"<<endl;
                return; }
        int ch;
        cout << "1.PreOrder"<<endl;
        cout << "2.InOrder"<<endl;
        cout << "3.PostOrder"<<endl;
        cout << "Enter the choice : ";cin>>ch;
        switch (ch){
                case 1:{
                        cout << "Pre Order traversal ! "<<endl;
                        preorder(root); cout<<endl; break; }
                case 2:{
                        cout << "In Order traversal ! "<<endl;
                        inorder(root);  cout <<endl;   break; }
                case 3:{
                        cout << "Post Order traversal ! "<<endl;
                        postorder(root); cout<<endl;  break;  }
                default:
                        cout<< "Wrong choice ! "<<endl;
        }
} };
int main(){
        int choice;
        BST b;
```

```
        cout<< " _____"<<endl;
        cout<< "|  Binary Serach Tree   |"<<endl;
        cout<< "|_____|"<<endl;
        cout<<endl<<endl;
    while(1){
    cout << "1.EXIT"<<endl;
    cout << "2.INSERT (FIND)"<<endl;
    cout << "3.DELETION"<<endl;
    cout << "4.TRAVERSALS"<<endl;
    cout << "5.INSERT (RECURSTION)"<<endl;
    cout << "6.SEARCH (RECURSTION)"<<endl;
    cout <<endl;
    cout << "Enter the Choice : ";cin>>choice;
    switch(choice){
        case 1:{
        return 0; }
    case 2:{
        b.insertTree(); break; }
    case 3:{
        b.deleteTree(); break;}
    case 4:{
        b.traversal(); break; }
    case 5:{
        b.insertRecurtion(); break; }
```

```
        case 6:{
                if(!b.searchTree()){
                        cout<< "Value not Found"<<endl; } break; }
        default :
        cout << "Wrong Input !"<<endl;
        }
}
return 0; }
```

8) Graph Implementation Using Adjacency List

Basic Operations for Graphs

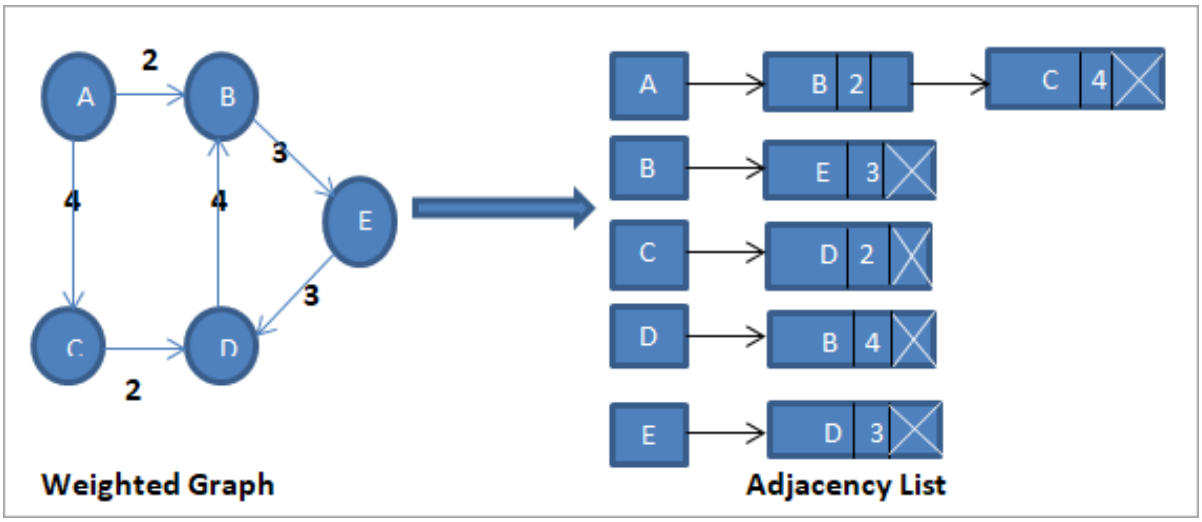
Following are the basic operations that we can perform on the graph data structure:

Add a vertex: Add vertex to the graph.

Add an edge: Adds an edge between the two vertices of a graph.

Display the graph vertices: Display the vertices of a graph.

Here we are going to display the adjacency list for a weighted directed graph. We have used two structures to hold the adjacency list and edges of the graph. The adjacency list is displayed as **(start_vertex, end_vertex, weight).**



Code:

```
#include <iostream>

using namespace std;

// stores adjacency list items
```

```
struct adjNode {
    int val, cost;
    adjNode* next;
};
// structure to store edges
struct graphEdge {
    int start_ver, end_ver, weight;
};
class DiaGraph{
    // insert new nodes into adjacency list from given graph
    adjNode* getAdjListNode(int value, int weight, adjNode* head)   {
        adjNode* newNode = new adjNode;
        newNode->val = value;
        newNode->cost = weight;

        newNode->next = head;   // point new node to current head
        return newNode;
    }
    int N;  // number of nodes in the graph
public:
    adjNode **head;              //adjacency list as array of pointers
    // Constructor
    DiaGraph(graphEdge edges[], int n, int N)  {
        // allocate new node
```

```
        head = new adjNode*[N]();
        this->N = N;
        // initialize head pointer for all vertices
        for (int i = 0; i < N; ++i)
            head[i] = NULL;
        // construct directed graph by adding edges to it
        for (unsigned i = 0; i < n; i++)  {
            int start_ver = edges[i].start_ver;
            int end_ver = edges[i].end_ver;
            int weight = edges[i].weight;
            // insert in the beginning
            adjNode* newNode = getAdjListNode(end_ver, weight, head[start_ver]);


                    // point head pointer to new node
            head[start_ver] = newNode;
             }
    }
     // Destructor
    ~DiaGraph() {
    for (int i = 0; i < N; i++)
        delete[] head[i];
        delete[] head;
     }
};
```

```
// print all adjacent vertices of given vertex
void display_AdjList(adjNode* ptr, int i)
{
    while (ptr != NULL) {
        cout << "(" << i << ", " << ptr->val
            << ", " << ptr->cost << ") ";
        ptr = ptr->next;
    }
    cout << endl;
}
// graph implementation
int main()
{
    // graph edges array.
    graphEdge edges[] = {
        // (x, y, w) -> edge from x to y with weight w
        {0,1,2},{0,2,4},{1,4,3},{2,3,2},{3,1,4},{4,3,3}
    };
    int N = 6;      // Number of vertices in the graph
    // calculate number of edges
    int n = sizeof(edges)/sizeof(edges[0]);
    // construct graph
    DiaGraph diagraph(edges, n, N);
    // print adjacency list representation of graph
```

```
    cout<<"Graph adjacency list "<<endl<<"(start_vertex, end_vertex, weight):"<<endl;
    for (int i = 0; i < N; i++)
    {
        // display adjacent vertices of vertex i
        display_AdjList(diagraph.head[i], i);
    }
    return 0;
}
```