BUBBLE SORT

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

//Functions prototype
void bubbleSort(int [], int );
void printArray(int[],int );

//Function main() triggers program execution
void main() {
     int array[6] ={5,2,1,3,6,4};
     bubbleSort(array,6);
     printArray(array,6);
}

//Function or method that performs the bubble sort algorithm
void bubbleSort(int a[], int length)
{
     int i, j, temp;
     for(i=0; i<6; i++)
     {
          for(j=0; j<6-i-1; j++)
          {
               if( a[j] > a[j+1])
               {
                    temp = a[j];
                    a[j] = a[j+1];
                    a[j+1] = temp;
               }
          }
     }
}


//Function to print array
void printArray(int arr[],int size){
     for(int i=0;i<size;i++)
          cout<<arr[i]<<endl;
}
```

SELECTION SORT

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

//Functions prototype
void selectionSort(int[],int );
void printArray(int[],int );

//Function main() triggers program execution
void main() {
     int array[6] ={5,2,1,3,6,4};
     selectionSort(array,6);
     printArray(array,6);
}

//Function or method that performs the selection sort algorithm
//It compares the currently scanned element with the elements succeeding it
//(Forward Comparison)

void selectionSort(int a[], int length)
{
     int i,j,min,temp;
     for(i=0; i<length-1;i++){
          min= i; //set the first element of the array as the least element
          for(j=i+1;j<length;j++){
               if(a[j]<a[min])
                    min=j; //set index j  as the minimum (min)
          }
Succeeding
          //Swapping operation between the ith element and the current least
element(i.e min)
          temp=a[i];
          a[i]=a[min];
          a[min]=temp;
     }
}


//Function to print array
void printArray(int arr[],int size){
     for(int i=0;i<size;i++)
          cout<<arr[i]<<endl;
}
```

INSERTION SORT

```
#include <iostream>

using namespace std;
void insertionSort(int[],int );
void printArray(int[],int );

//Function main() triggers program execution
void main() {
     int array[6] ={5,2,1,2,1,6};
     insertionSort(array,6);
     printArray(array,6);
}


/*        A brief description on how this sorting works:
     * The scanning of elements start from the second element (i.e index 1) and
not the first element
     because the first element obviously is not preceded by any number.
     it compares the currently scanned number against the preceding element
(backward)
*/
void insertionSort(int arr[],int size)
{
     int i,j,temp;
     for(i=1;i<size;i++) {
          j= i;
          while(j>=0 && arr[j]< arr[j-1])
          {
               temp= arr[j];
               arr[j]= arr[j-1];
               arr[j-1]= temp;
               j--;
          }
     }
}

void printArray(int arr[],int size){
     for(int i=0;i<size;i++)
          cout<<arr[i]<<endl;
}
```

QUICK SORT

```
#include <iostream>
using namespace std;
/*  a[] is the array, p is starting index, that is 0,
and r is the last index of array (i.e array_length-1).  */

int partition(int a[],int startIndex, int lastIndex) {
     int i,j,temp,pivot;
     i= startIndex;
     j=lastIndex;

     //You can make any of the numbers in the list to be sorted, not necessarily
the 0th element
     pivot= a[startIndex];

     while(1)
     {
          while(a[i]<pivot && a[i]!= pivot)
               i++;
          while(a[j]>pivot && a[j]!= pivot)
               j--;
          if(i<j)
          {
               temp= a[i];
               a[i]=a[j];
               a[j]=temp;
          }
          else
               return j;
     }
}

void quickSort(int a[],int startIndex,int lastIndex)
{
     if(startIndex<lastIndex)
     {
          int pivot;
     //partition into two list (pivot not included in any) then returns the index
of ur pivot
     pivot= partition(a,startIndex,lastIndex);

     //Sort the first partitioned list (i.e elements less than the pivot)
     quickSort(a,startIndex,pivot-1);

     //Sort the second partitioned list (i.e elements greater than the pivot)
     quickSort(a,pivot+1,lastIndex);
     }
}
```

```
void printArray(int arr[],int size){
     for(int i=0;i<size;i++)
          cout<<arr[i]<<endl;
}

void main() {
     int array1[10] ={6,2,5,10,1,4,3,8,7,9};
     quickSort(array1,0,9);
     printArray(array1,10); }
```

MERGE SORT

```
#include <iostream>
#include <cmath>

using namespace std;

void merge(int a[], int start, int mid,int last) {
int b[10]; //same size with array to be sorted
int i,j,k;
i= start;
j=mid+1;
k=0;


while(i<=mid && j<= last)
{
     if(a[i]<a[j])
          b[k++]= a[i++];
     else
          b[k++]= a[j++];
}


while (i<= mid){
     b[k++]= a[i++];
}

while (j<= last){
     b[k++]= a[j++];
}
```

```
//Copy back the sorted list into a[]
for(j=last;j>=start;j--)
     a[j]=b[--k];

}

void mergeSort(int a[],int start, int last){
     int mid;
     if(start<last) {
          mid= floor((start+last)/2);
          mergeSort(a,start,mid);
          mergeSort(a,mid+1,last);
          merge(a,start,mid,last);
     }
}

void printArray(int arr[],int size){
     for(int i=0;i<size;i++)
          cout<<arr[i]<<endl;
}

void main() {
     int array[10] ={6,2,5,10,1,4,3,8,7,9};
     int array2[5]= {3,5,2,1,4};
     mergeSort(array,0,9);
     printArray(array,10);
     //return 0;
}
```

HEAP SORT

```
/* Below program is written in C++ language */
#include <iostream>
using namespace std;

void heapsort(int[], int);
void buildheap(int [], int);
void satisfyheap(int [], int, int);
void printArray(int [],int);
void main()
{
     int a[10], i, size;
     cout << "Enter size of list"; // less than 10, because max size of array is
10
     cin >> size;
     cout << "Enter" << size << "elements";
     for( i=0; i < size; i++)
     {
          cin >> a[i];
     }

     heapsort(a, size);
     printArray(a,size);
} //End Main

void heapsort(int a[], int length)
{
     buildheap(a, length);
     int heapsize, i, temp;
     heapsize = length - 1;
     for( i=heapsize; i >= 0; i--)
     {
          temp = a[0];
          a[0] = a[heapsize];
          a[heapsize] = temp;
          heapsize--;
          satisfyheap(a, 0, heapsize);
     } //End for

} //End heapsort
void buildheap(int a[], int length)
{
     int i, heapsize;
     heapsize = length - 1;
     for( i=(length/2); i >= 0; i--)
     {
          satisfyheap(a, i, heapsize);
     }
```

```
}

void satisfyheap(int a[], int i, int heapsize)
{
     int l, r, largest, temp;
     l = 2*i;
     r = 2*i + 1;

     if(l <= heapsize && a[l] > a[i])
     {
          largest = l;
     }

     else
     {
          largest = i;
     } //End else

     if( r <= heapsize && a[r] > a[largest])
     {
          largest = r;
     } //End if

     if(largest != i)
     {
          temp = a[i];
          a[i] = a[largest];
          a[largest] = temp;
          satisfyheap(a, largest, heapsize);
     } //End if
} //End satisfyheap

//Function to print array
void printArray(int arr[],int size)
{
     for(int i=0;i<size;i++)
          cout<<arr[i]<<endl;
}
```

SEARCHING ALGORITHMS

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

int linearSearch(int[],int,int);
int binarySearch(int[], int, int, int);

int main()
{
     int a[] ={8 , 2 , 6 , 3 , 5};
     int b[]= {2, 4, 7, 9, 13, 15};

     cout<<"Searching using Linear search\nElement found at index: "
<<linearSearch(a,3,5);
     cout<<endl<<endl;
     cout<<"Searching using Binary Search search\nElement found at index: "
<<binarySearch(b,13,0,5);

}


int linearSearch(int value [], int target, int arraySize)
{
     for(int i = 0; i < arraySize; ++i)
     {
          if (value[i] == target)
          {
               return i;
          }
     }
     return -1;
}

int binarySearch(int values[], int target,int  start, int end) {
     if (start > end) { return -1; } //does not exist
int middle = floor((start + end) / 2);
int value = values[middle];
if (value > target) { return binarySearch(values, target, start, middle-1); }
if (value < target) { return binarySearch(values, target, middle+1, end); }
return middle; //found!
}
```

STACK

```
/* Below program is written in C++ language */
#include <iostream>
using namespace std;

class Stack
{
     int top;

     public:
     int a[10]; //Maximum size of Stack

     Stack();
     void push(int);
     int pop();
     void isEmpty();
     void display();

};

Stack::Stack()
{
     top = -1;
}

void Stack::push(int x)
{
     if( top >= 10)
     {
          cout << "Stack Overflow";
     }
     else
     {
          a[++top] = x;
          cout << "Element Inserted";
     }
}

int Stack::pop()
{
     if(top < 0)
     {
     cout << "Stack Underflow";
     return 0;
     }

     else
     {
          int d = a[top--];
```

```
            return d;
      }
}

void Stack::isEmpty()
{
      if(top < 0)
      {
            cout << "Stack is empty";
      }

      else
      {
            cout << "Stack is not empty";
      }

}

void Stack::display()
      {
      for(int i=top; i>=0;i--)
            cout<<a[i]<<endl;

      }

int main()
{
      Stack s1;
      s1.push(1);
      s1.push(2);
      s1.push(3);
      s1.push(4);
      s1.push(5);
      s1.pop();
      s1.pop();

      //Display stack
      s1.display();
      return 0;
}
```

QUEUE

```
/* Below program is wtitten in C++ language */
#include <iostream>
#define SIZE 100

using namespace std;
class Queue
{
int a[100];
int rear; //same as tail
int front; //same as head
public:
Queue()
{
     rear = front = -1;
}

void enqueue(int x); //declaring enqueue, dequeue and display functions
int dequeue();
void display();
};

void Queue :: enqueue(int x)
{
     if( rear == SIZE-1)
     {
          cout << "Queue is full";
     }

     else
     {
          if (front == -1) front =0; //Set the first element as front
          a[++rear] = x;
     }
}

int Queue :: dequeue()
{
     return a[++front]; //following approach [B], explained above
}

void Queue :: display()
{
     int i;
     for( i = front; i <= rear; i++)
     {
          cout << a[i] <<endl;
     }
}
```

```
int main()
{
     Queue q1;
     q1.enqueue(1);
     q1.enqueue(2);
     q1.enqueue(3);
     q1.enqueue(4);
     q1.enqueue(5);
     q1.dequeue();

     //Display stack
     q1.display();
     return 0;
}
```

QUEUE USING STACK

```
//Stack.h
#include <iostream>
using namespace std;

class Stack
{
     int top;

     public:
     int a[10]; //Maximum size of Stack

     Stack();
     void push(int);
     int pop();
     bool isEmpty();
     void display();
     int Top();

};

Stack::Stack()
{
     top = -1;
}

void Stack::push(int x)
{
     if( top >= 10)
     {
          cout << "Stack Overflow" <<endl;
     }
     else
     {
          a[++top] = x;
          cout << "Element Inserted"<<endl;
     }
}

int Stack::Top() {
     return top;
}

int Stack::pop()
{
     if(top < 0)
     {
     cout << "Stack Underflow"<<endl;
```

```
	return 0;
	}

	else
	{
		int d = a[top--];
		return d;
	}
}

bool Stack::isEmpty()
{
	if(top < 0)
	{

		return true;
	}

	else
	{

		return false;
	}

}

void Stack::display()
	{
	for(int i=top; i>=0;i--)
		cout<<a[i]<<endl;

	}


#include <iostream>
#include "Stack.h"

class QueueUsingStack {
	Stack S1, S2;

public:
//defining methods
void enqueue(int x);
int dequeue();
void display();
};

void QueueUsingStack::enqueue(int x) {
	S1.push(x);
```

```
}

int QueueUsingStack::dequeue(){
     int x;
     //Transfer S1 into S2 so as to remove element in FIFO order and not LIFO of
a stack
     while(!S1.isEmpty()) {
           x= S1.pop();
          S2.push(x);
     }
     //Remove the element
     S2.pop();
     while(!S2.isEmpty()){
          x= S2.pop();
          S1.push(x);
     }
     return x;
}

void QueueUsingStack::display(){
     S1.display();
}

int main()
{
     QueueUsingStack q1;
     q1.enqueue(1);
     q1.enqueue(2);
     q1.enqueue(3);
     q1.enqueue(4);
     q1.enqueue(5);
     q1.dequeue();

     //Display stack
     q1.display();
     return 0;
}
```

LINKED LIST

```
//Node.h, header file to be included for LinkedList and CircularLinkedList
class Node {
public:
int data;
//pointer to the next node
Node* next;

Node() {
     data = 0;
     next = NULL;
}

Node(int x) {
     data = x;
     next = NULL;
}


int getData() {
return data;
}

void setData(int x) {
     data = x;
}

Node* getNext() {
     return next;
}

void setNext(Node *n) {
     next = n;
}

};


//Linked list Class
#include <iostream>
#include "Node.h"
using namespace std;

class LinkedList {
public:
Node *head;
//declaring the functions
```

```
//function to add Node at front
int addAtFront(Node *n);

//function to check whether Linked list is empty
bool isEmpty();

//function to add Node at the End of list
int addAtEnd(Node *n);

//function to search a value
Node* search(int k);

//function to delete any Node
Node* deleteNode(int x);

//function to get last Node
Node* getLastNode();

//function to print the LinkedList elements
void display();

//Constructor
LinkedList();
};

LinkedList::LinkedList()
{
     head = NULL;
}

int LinkedList :: addAtFront(Node *n) {
     int i = 0;
     //making the next of the new Node point to Head
     n->next = head;
     //making the new Node as Head
     head = n;
     i++;
     //returning the position where Node is added
     return i;
}

int LinkedList :: addAtEnd(Node *n) {
     int i=0;

     //If list is empty
     if(head == NULL)
     {
          //making the new Node as Head
          head = n;
```

```
            i++;
            //making the next pointe of the new Node as Null
            n->next = NULL;
            return i;
      }

      else
      {
            //getting the last node
            Node *n2 = getLastNode();
            n2->next = n; //Making the last Node point to the new Node
            n->next=NULL; //Making the new Node the last Node
      }

}

Node* LinkedList :: getLastNode() {
      //creating a pointer pointing to Head
      Node* ptr = head;
      //Iterating over the list till the node whose Next pointer points to null
      //Return that node, because that will be the last node.
      while(ptr->next!= NULL)
      {
            //if Next is not Null, take the pointer one step forward
            ptr = ptr->next;
      }
      return ptr;
}

Node* LinkedList :: search(int x)
{
            for(Node *ptr = head; ptr != NULL; ptr= ptr->next)
                  {
                        //if x is found
                        if(ptr->data == x)
                        {
                              return ptr;
                              break;  //go out of the for loop when x has been found
                        }
                  }

      return NULL;      //When x is not found
}

Node* LinkedList :: deleteNode(int x)
{
      //searching the Node with data x
      Node *n = search(x);
      Node *ptr = head;
```

```
	//if search(x) returned NULL i.e x was not found
	if (n == NULL)
	{
		cout <<endl <<x <<" not found on the list, hence no deletion" <<endl;
		return NULL;
	}

	//if x is found at the head Node
	if(ptr == n)
	{
		head=head->next;
		return n;
	}

	else
	{

		while(ptr->next != n)
		{
			ptr = ptr->next;
		}

		//Let ptr points to the next of the deleted node,n
		ptr->next = n->next;
		return n;
	}
}

bool LinkedList :: isEmpty()
{
	if(head == NULL)
	{
	return true;
	}

	else { return false; }

}

void LinkedList::display() {

	//If list is empty just return (no element to display)
	if(isEmpty()) return;

	for (Node* ptr= head;ptr!=NULL;ptr=ptr->next)
	{
		cout<<(ptr->data)<<endl;
	}
```

```
}

int main()
{
     LinkedList L;
     //Adding the node to the list
     L.addAtEnd(new Node(1));
     L.addAtEnd(new Node(2));
     L.addAtEnd(new Node(3));
     L.addAtEnd(new Node(4));
     L.addAtEnd(new Node(5));
     L.addAtEnd(new Node(6));
     L.addAtEnd(new Node(7));
     L.addAtEnd(new Node(8));
     L.addAtEnd(new Node(9));
     L.addAtEnd(new Node(10));

     //Display elements of the linked list
     L.display();

     //Perform some deletions
     L.deleteNode(1);
     L.deleteNode(2);
     L.deleteNode(3);
     L.deleteNode(4);
     L.deleteNode(5);


     //Display the list after deletion with a caption
     cout<<endl<<"After deletion 1,2,3,4 and 5"<<endl;
     L.display();
}
```

CIRCULAR LINKED LIST

```
#include <iostream>
#include "Node.h" //include the header file containing Node class definition
using namespace std;

class CircularLinkedList {

     Node *head, *tail;

public:

     //declaring the functions

     //function to add Node at front
     int addAtFront(Node *n);

     //function to check whether Linked list is empty
     bool isEmpty();

     //function to add Node at the End of list
     int addAtEnd(Node *n);

     //function to search a value
     Node* search(int k);

     //function to delete any Node
     Node* deleteNode(int x);

     //Function to display the list
     void display();

     //CircularLinkedList constructor
     CircularLinkedList()
     {
          head = NULL; // Empty list
     }
};

//Definition of functions outside the class
int CircularLinkedList :: addAtFront(Node *n) {
     int i=0;
     //If list is empty
     if(head == NULL)
     {
          //making the new Node as Head and tail (i.e List have just one Node
after adding n)
          head = tail = n;
          //making the  tail to point back to head
```

```
            tail->next= head; // or tail->next = n; since n is the head
            i++;
            return i;
      }

      else
      {
            n->next = head; //Making the current Head as the next element to the
new Node
            head= n; //Making the new Node, n as the  new head
            i++;
            return i;
      }
}

bool CircularLinkedList::isEmpty()
{
      return head== NULL;
}

int CircularLinkedList :: addAtEnd(Node *n)
{
      int i=0;

      //If list is empty
      if(head == NULL)
      {
            //making the new Node as Head and tail (i.e List have just one Node
after adding n)
            head = tail = n;
            //making the  tail to point back to head
            tail->next= head; // or tail->next = n; since n is the head
            i++;
            return i;
      }

      else
      {
            //getting the tail point to the new Node, n
            tail->next = n;
            tail= n; //n is now the new tail
            n->next = head; //Making n point to the head (Circular link)
            i++;
            return i;
      }
}


Node* CircularLinkedList :: search(int x)
```

```
{
	Node *ptr = head;

	//if List is empty dont even bother searching
	if(isEmpty()) return NULL;

	//if x is found at the head Node
	if(ptr != NULL && ptr->data == x) return ptr;

	//Loop from the Node next to the head Node until the last Node (i.e Node
that point to head)
	for (ptr= head->next; ptr!= head; ptr = ptr->next)
	{
		//if x is found
		if(ptr->data == x)
		{
			return ptr;
			break;  //go out of the for loop when x has been found
		}

	}


	return NULL;	//When x is not found
}

Node* CircularLinkedList :: deleteNode(int x)
{
	//searching the Node with data x (Node to delete)
	Node *n = search(x);
	Node *ptr = head;

	//if search(x) returned NULL i.e x was not found
	if (n == NULL)
	{
		cout <<endl <<x <<" not found on the list, hence no deletion" <<endl;
		return NULL;
	}

	//If list is empty and you tried to delete
	if(ptr == NULL)
	{
		return NULL;
	}

	//else if x was found at head
	else if(ptr->data == n->data)
	{
		//If the head is only the Node on the list
```

```
            if(head->next == head) { head=tail= NULL; return n; }

            head= ptr->next;
            tail->next = head;
            return n;
      }

      //list was not empty and x was not found at the head Node
      else
      {
            //Loop and stop at a node whose next has x
            while(ptr->next->data != n->data)
            {
                  ptr = ptr->next;
            }

            /*Recall Node n is the Node to be deleted
            Let the Node preceeding the Node to be deleted point to the Node next
to n
            */
            ptr->next = n->next;
            return n;  //returns Node n (Node to be deleted)
      }
}

void CircularLinkedList::display()
{

      //If list is empty just return (no element to display)
      if(head==NULL) return;

      Node* ptr= head; //pointer that points to the head

      cout<<(ptr->data)<<endl; //display the data stored at the head Node
seperately

      //Loop from the Node next to the head Node until the last Node (i.e Node
that point to head)
      //then display the data stored there
      for (ptr=head->next;ptr!=head;ptr=ptr->next)
      {
            cout<<(ptr->data)<<endl;
      }
}
int main()
{
      CircularLinkedList L;
      //Adding the node to the list
      L.addAtFront(new Node(1));
```

```
    L.addAtEnd(new Node(2));
    L.addAtEnd(new Node(3));
    L.addAtEnd(new Node(4));
    L.addAtEnd(new Node(5));
    L.addAtEnd(new Node(6));
    L.addAtEnd(new Node(7));
    L.addAtEnd(new Node(8));
    L.addAtEnd(new Node(9));
    L.addAtEnd(new Node(10));

    //Display elements of the linked list
    L.display();

    //Perform some deletions
    L.deleteNode(1);
    L.deleteNode(2);
    L.deleteNode(3);
    L.deleteNode(4);
    L.deleteNode(5);
    L.deleteNode(5); //Just to show that they wont be deletion as there was only
one 5 on the list which as been deleted previously

    //Display the list after deletion with a caption
    cout<<endl<<"After deletion 1,2,3,4 and 5"<<endl;
    L.display();
    cout<<L.isEmpty();
}
```