INDEX

S.No.	Assignment-3	Page No.	Signature
Q1.	Write a C++ program to implement the Merge Sort.		
Q2.	Write a C++ program to implement the Heap Sort.		
Q3.	Write a C++ program to perform the following operations. a) Insert an element into a binary search tree. b) Delete an element from a binary search tree. c) Search for a key element in a binary search tree.		
Q4.	Write C++ programs that use recursive functions to traverse the given binary tree in. a) Preorder b) Inorder c) Postorder		
Q5.	Write a C++ program to perform the following operations a) Insertion into a B-tree b) Deletion from a B-tree		
Q6.	Write a C++ program to perform the following operations a) Insertion into an AVL-tree b) Deletion from an AVL-tree		

Assignment-1

Q1.Write a C++ program to implement recursive and non-recursive

a) Linear search

-:Recurive Program:-

```
#include<iostream>
#include<conio.h>
using namespace std;
int Linear(int array[],int key,int size)
      { size=size-1;
      if(size <0) {
                  return -1;
      } else if(array[size]==key) {
                  cout<<"Key Found in Array at index:"<<size;
                  return 1;
      } else
      {           return Linear(array,key,size);

      }
}

int main() {
      cout<<"Enter The Size Of Array:  ";
      int size;
      cin>>size;
      int array[size], key,i;

      // Taking Input In Array

       for (int j=0;j<size;j++) {
                  cout<<"Enter "<<j<<" Element : ";
                  cin>>array[j];
      }
      //Your Entered Array Is
       for (int a=0;a<size;a++)
      {
                  cout<<"array[ "<<a<<" ] =  ";
                  cout<<array[a]<<endl;
      }
      cout<<"Enter Key To Search  in Array:  ";
       cin>>key;
      int result;

       result=Linear(array,key,size--);
```

```
    if(result!=1) {
            cout<<"Key NOT Found in Array  ";
    }
    getch();
    return 0;

}
```

Output:-

```
Enter The Size Of Array:  5
Enter 0 Element : 11
Enter 1 Element : 16
Enter 2 Element : 18
Enter 3 Element : 21
Enter 4 Element : 35
```

array[0]	=	11
array[1]	=	16
array[2]	=	18
array[3]	=	21
array[4]	=	35

```
Enter Key To Search  in Array:  21
Key Found in Array at index:3
```

-:Non-recurive Program:-

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


int Linear(int array[],int key,int size)
   { for(int i=0;i<size;i++)
   {
     if(array[i]==key)
     {
            cout<<"Key Found in Array at index:"<<i;
            return 1;
            break;
            }
     }
}
```

```
int main() {
	cout<<"Enter The Size Of Array:  ";
	 int size;

	cin>>size;
	int array[size], key,i;

	// Taking Input In Array

	 for (int j=0;j<size;j++) {
		cout<<"Enter "<<j<<" Element : ";
		cin>>array[j];
	}
	//Your Entered Array Is

	 for (int a=0;a<size;a++)
	{
		cout<<"array[ "<<a<<" ] =  ";
		cout<<array[a]<<endl;
	}
	cout<<"Enter Key To Search  in Array:  ";
	cin>>key;
	int result;
	result=Linear(array,key,size);
	if(result!=1) {
		cout<<"Key NOT Found in Array  ";
	}
	getch();
	return 0;

}
```

Output:-

```
Enter The Size Of Array:  5
Enter 0 Element : 21
Enter 1 Element : 54
Enter 2 Element : 65
Enter 3 Element : 78
Enter 4 Element : 91
```

array[0]	=	21
array[1]	=	54
array[2]	=	65
array[3]	=	78
array[4]	=	91

Enter Key To Search in Array: 65
Key Found in Array at index:2

b)Binary search.

-:Recurive Program:-

```
#include <iostream>
#include<conio.h>
using namespace std;
int binarySearch(int A[], int start, int end, int key)
{
   while(start <= end)
   {
      int mid = (start + end)/2;
      if(A[mid] == key)
      {
         return mid;
      }
      else if(A[mid] > key)
      {
         return binarySearch(A, start, mid - 1, key);
      }
      else
      {
       return binarySearch(A, mid + 1, end, key);
      }
   }
   return -1;
}
int main()
{
   int A[] = { 11, 22, 33, 44, 55, 66, 77};
   int n = sizeof(A) / sizeof(A[0]);  //n is the size of the array
   int key;
   cout << "Array Elements :--> ";
   for(int i=0; i<n; i++)
      cout << A[i] << " ";
   cout << "\nEnter Key Element to Search :--> ";
   cin >> key;
```

```
    int result = binarySearch(A, 0, n - 1, key);
    if(result == -1)
    {
     cout << "\nElement is not present in array";
    }
    else
    {
     cout << "\nElement is present at index " << result;
    }
    getch();
    return 0;
  }
```

Output:-

Array Elements :--> 11 22 33 44 55 66 77

Enter Key Element to Search :--> 44

Element is present at index 3

-:Non-recurive Program:-

```
#include<iostream>
#include<conio.h>
 using namespace std; int main()
{
   int i, arr[10], num, first, last, middle;
   cout<<"Enter 10 Elements : ";
   for(i=0; i<10; i++)
      cin>>arr[i];
   cout<<"\nEnter Element to be Search: ";
   cin>>num;
   first = 0;
   last = 9;
   middle = (first+last)/2;
   while(first <= last)
   {
      if(arr[middle]<num)
         first = middle+1;
      else if(arr[middle]==num)
      {
         cout<<"\nThe number, "<<num<<" found at Position "<<middle+1;
         break;
      }
      else
         last = middle-1;
      middle = (first+last)/2;
```

```
    }
    if(first>last)
       cout<<"\nThe number, "<<num<<" is not found in given Array";
    cout<<endl;
    getch();
    return 0;
}
```

Output:-

Enter 10 Elements : 11 21 31 41 51 61 71 81 91 101

Enter Element to be Search: 71

The number, 71 found at Position

7

Q2. Write a C++ program to implement Bubble sort.

```
#include <iostream>
#include<conio.h>
using namespace std;
void swap(int *var1, int *var2)
{
   int temp = *var1;
   *var1 = *var2;
   *var2 = temp;
}
//Here we will implement bubbleSort.
 void bubbleSort(int arr[], int n)
{
  int i, j;
  for (i = 0; i < n-1; i++)

     //Since, after each iteration rightmost i elements are sorted.
      for (j = 0; j < n-i-1; j++)
        if (arr[j] > arr[j+1])
           swap(&arr[j], &arr[j+1]);
}

// Function to print array.
void display(int arr[], int
size)
{
   int i;
```

```
   for (i=0; i < size; i++)
      cout << arr[i] << "\t";

   cout<<endl;
}

//Main function to run the program.
 int main()
{
   int array[] = {5, 3, 1, 9, 8, 2, 4,7};
   int size = sizeof(array)/sizeof(array[0]);
   cout<<"Before bubble sort: \n";
   display(array, size);//Calling display function to print unsorted array.
   bubbleSort(array, size);
   cout<<"After bubble sort: \n";
   display(array, size);//Calling display function to print sorted array.
   getch();
   return 0;
}
```

Output:-

Before bubble sort:

5 3 1 9 8 2 4 7

After bubble sort:

1 2 3 4 5 7 8 9

Q3. Write a C++ program to implement Selection sort.

```
#include<iostream>

#include<conio.h>

using namespace std;


//Display function to print values.

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

```
//The main function to drive other functions.
int main()
{
   int array[] = {5, 3, 1, 9, 8, 2, 4, 7};
   int size = sizeof(array)/sizeof(array[0]);
   cout << "Before sorting: \n";
   display(array, size);
   int i, j, min_idx,temp;

   //Loop to iterate elements of array.
   for (i = 0; i < size-1; i++)
   {
      //Here we try to find the min element in array.
      min_idx = i;
      for (j = i+1; j < size; j++){
         if (array[j] < array[min_idx])
            min_idx = j;
      }

      // Here we interchange the min element with first one.
      temp = array[min_idx];
      array[min_idx] = array[i];
      array[i] = temp;
   }
   cout << "After sorting: \n";
   display(array, size); //Using dispaly function to print sorted array.
   getch();
   return 0;
}
```

Output:-

```
Before sorting:
5     3     1     9     8     2     4     7
After sorting:
1     2     3     4     5     7     8     9
```

Q4. Write a C++ program to implement quick sort.

```
#include<iostream>
#include<conio.h>

using namespace std;
//Function to swap two elements.
Void swap(int* x, int* y)
{
```

```
   int temp = *x;
   *x = *y;
   *y = temp;
}

/* Partition function to do Partition
elements on the left side of pivot elements would be smaller than pivot
elements on the right side of pivot would be greater than the pivot
*/
int partition (int array[], int low, int high)
{
   //Pivot element selected as right most element in array each time.
   Int pivot = array[high];
   int swapIndex  = (low – 1);  //swapping index.
   For (int j = low; j <= high- 1; j++)
   {
      //Check if current element is smaller than pivot element.
      If (array[j] < pivot)
      {
         swapIndex ++;    //increment swapping index.
         Swap(&array[swapIndex], &array[j]);
      }
   }
   swap(&array[swapIndex + 1], &array[high]);
   return (swapIndex + 1);
}

//Recursive function to apply uicksort
void uicksort(int array[], int low, int high)
{
   if (low < high)
   {
     /* indexPI is partitioning index, partition() function will
      return index of partition */
      int indexPI  = partition(array, low, high);

      uicksort(array, low, indexPI  - 1);  //left partition
      uicksort(array, indexPI  + 1, high); //right partition
   }
}

//Function to display the array
void display(int array[], int size)
{
   int I;
   for (i=0; I < size; i++)
      cout<< array[i] <<” “;
```

```
}

//Main function to run the program
int main()
{
   int array[] = {7, 9, 1, 3, 5, 2, 6, 0, 4, 8};
   int size = sizeof(array)/sizeof(array[0]);
   cout<<”Before Sorting: \n”;
   display(array, size);
   uicksort(array, 0, size-1); cout<<”\
   nAfter Sorting: \n”; display(array, size);
   getch();
   return 0;
}
```

Output:-

```
Before Sorting:
7 9 1 3 5 2 6 0 4 8

After Sorting:
0 1 2 3 4 5 6 7 8 9
```

Q5. Write a C++ program to implement insertion sort

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

//Function to print
array.
void display(int arr[], int size)
{
   int i;
   for (i = 0; i < size; i++)
   cout << arr[i] << "\t";
   cout << "\n";
}

//Main function to run the program.
int main()
{
   int array[] = {5, 3, 1, 9, 8, 2, 4,7};
   int size = sizeof(array)/sizeof(array[0]);
   cout << "Before Insertion sort: \n";
   display(array, size);
   int i, key, j;
```

```
   for (i = 1; i < size; i++) { key = array[i]; j = i - 1;
   /* Here the elements in b/w arrary[i-1 & 0] which are greater than key are moved ahead by 1
position each*/
            while (j >= 0 && array[j] > key)
       {
          array[j + 1] = array[j];
          j = j - 1;  // moving backwards
       }

       // placing key item now
       array[j + 1] = key;
   }
   cout << "After Insertion sort: \n";
   display(array, size);
   getch();
   return 0;
}
```

Output:-

```
Before Insertion sort:
5      3      1      9      8      2      4      7
After Insertion sort:
1      2      3      4      5      7      8      9
```

Assignment-2

Q1. Write a C++ program to implement the Stack ADT using an array

```
#include<iostream>
#include<conio.h>
using namespace std;
class Stack
{
   public:
   int top;
   int maxSize;
   int* array;
   Stack(int max)
   {
      top=-1;
      maxSize=max;
      array=new int[max];
   }
   int isFull()
   {
      if(top==maxSize-1)
      cout<<"Will not be able to push maxSize reached"<<endl;
      return top==maxSize-1;
   }

   int isEmpty()
   {
      if(top==-1)
      cout<<"Will not be able to pop minSize reached"<<endl;
      return top==-1;
   }
   void push(int item)
   {
      if(isFull()) return;
      array[++top]=item;
      cout<<"We have pushed "<<item<<" to stack"<<endl;
   }
   int pop()
   {
      if(isEmpty()) return INT_MIN;
      return array[top--];
   }
   int peek()
   {
      if(isEmpty()) return INT_MIN;
      return array[top];
   }
```

```
};

int main()
{
   Stack stack(10);
   stack.push(5);
   stack.push(10);
   stack.push(15);
   int flag=1;
   while(flag)
   {
      if(!stack.isEmpty())
         cout<<"We have popped "<< stack.pop()<<" from stack"<<endl;
      else
         cout<<"Can't Pop stack must be empty\n";

        flag=0;
   }
     getch();
     return 0;
}
```

Output:-

```
We have pushed 5 to stack
We have pushed 10 to stack
We have pushed 15 to stack
We have popped 15 from stack
```

Q2. Write a C++ program to implement the Queue ADT using an array.

```
#include <iostream>
using namespace std;
class Queue {
private:
   static const int MAX_SIZE = 100; // Maximum size of the queue
   int arr[MAX_SIZE];
   int front, rear;
public:
   Queue() : front(-1), rear(-1) {}
   bool isEmpty() { return front == -1 && rear == -1; }
   bool isFull() { return (rear + 1) % MAX_SIZE == front; }
   void enqueue(int value) {
      if (isFull()) {
         cout << "Queue is full. Cannot enqueue.\n";
         return;
      }
      if (isEmpty()) {front = rear = 0; }
     else {rear = (rear + 1) % MAX_SIZE;}
      arr[rear] = value;
      cout << value << " enqueued to the queue.\n";
   }
   void dequeue() {
      if (isEmpty()) {
         cout << "Queue is empty. Cannot dequeue.\n";
         return;
      }
      cout << arr[front] << " dequeued from the queue.\n";
      if (front == rear) { front = rear = -1; // Reset front and rear for an empty queue }
      else { front = (front + 1) % MAX_SIZE; }
   }
   int frontValue() {
      if (isEmpty()) {
         cout << "Queue is empty. No front value.\n";
         return -1; // Assuming -1 as an indicator of an empty queue
      }
      return arr[front];
   }
};
int main() {
   Queue myQueue;
   myQueue.enqueue(10);
   myQueue.enqueue(20);
   myQueue.enqueue(30);
   cout << "Front of the queue: " << myQueue.frontValue() << std::endl;
   myQueue.dequeue();
   myQueue.dequeue();
```

```
    cout << "Is the queue empty? " << (myQueue.isEmpty() ? "Yes" : "No") << std::endl;
    return 0;
}
```

Output:-

```
1) Insert element to queue
2) Delete element from queue
3) Display all the elements of queue
4) Exit
Enter your choice :
1
Insert the element in queue :
10
Enter your choice :
1
Insert the element in queue :
20
Enter your choice :
1
Insert the element in queue :
30
Enter your choice :
2
Element deleted from queue is : 10
Enter your choice :
3
Queue elements are : 20 30
Enter your choice :
4
Exit
```

Q3. Write a C++ program to implement list ADT to perform the following operations.

a) Insert an element into a list.
b) Delete an element from a list.
c) Search for a key element in a list
d) count number of nodes in a list.

```
#include <iostream>
Using namespace std;
class Node {
public:
   int data;
   Node* next;
   Node(int value) {
 This->data=value;
 This->next=NULL
}
};
class LinkedList {
private:
   Node* head;
public:
   LinkedList() {
 This->head=NULL
}
   // Function to insert an element into the list
   void insertElement(int value) {
      Node* newNode = new Node(value);
      newNode->next = head;
      head = newNode;
      cout << "Element " << value << " inserted into the list." << endl;
   }
   // Function to delete an element from the list
   void deleteElement(int value) {
      Node* current = head;
      Node* prev = NULL;
      while (current != NULL && current->data != value) {
         prev = current;
         current = current->next;
      }
      if (current == NULL) {
         cout << "Element " << value << " not found in the list." << endl;
         return;
      }
      if (prev == NULL) {
         head = current->next;
      } else {
```

```
            prev->next = current->next;
        }
        delete current;
        cout << "Element " << value << " deleted from the list." << endl;
    }
    // Function to search for a key element in the list
    bool searchElement(int value) {
        Node* current = head;
        while (current != NULL) {
            if (current->data == value) {
                cout << "Element " << value << " found in the list." << endl;
                return true;
            }
            current = current->next;
        }
        cout << "Element " << value << " not found in the list." << endl;
        return false;
    }
    // Function to count the number of nodes in the list
    int countNodes() {
        Node* current = head;
        int count = 0;
        while (current != NULL) {
            count++;
            current = current->next;
        }
        return count;
    }
};
int main() {
    LinkedList myList;
    myList.insertElement(10);
    myList.insertElement(20);
    myList.insertElement(30);
    myList.insertElement(40);
    cout << "Number of nodes in the list: " << myList.countNodes() <<endl;
    myList.searchElement(20);
    myList.searchElement(50);
    myList.deleteElement(30);
    cout << "Number of nodes in the list: " << myList.countNodes() << endl;
    return 0;
}
```

Output:-

```
Element 10 inserted into the list.
Element 20 inserted into the list.
Element 30 inserted into the list.
```

Element 40 inserted into the list.
Number of nodes in the list: 4
Element 20 found in the list.
Element 50 not found in the list.
Element 30 deleted from the list.
Number of nodes in the list: 3

Q4. Write C++ programs to implement the Stack ADT using a singly linked list.

```
#include<iostream>
#include<conio.h>
using namespace std;
class node
{ public:
  int data;
  node* next;
};
struct node* head = NULL;
void push(int x)
{
  node* temp;
  temp = new node();
  temp->data = x;
  temp->next = head;
  head = temp;
}
bool isEmpty()
{
  if (head == NULL)
      return true;
  else
      return false;
}
int top_element()
{
  if (head == NULL)
  {
      cout << "stack is empty" << endl;
  }
  else
      return head->data;
}
void pop()
{
  node* temp;
  if (isEmpty())
  {
      cout << "stack is empty" << endl;
  }
  else
  {
      temp = head;
      head = head->next;
      delete(temp);
  }
```

```
}
void print_stack()
{
  struct node* curr;
  if (isEmpty())
  {
      cout << "stack is empty" << endl;
  }
  else
  {
      curr = head;
      cout << "Elements are: ";
      while (curr != NULL)
      {
        cout << curr->data << " ";
        curr = curr->next;
      }
      cout << endl;
  }
}
int main()
{
  push(5);
  push(3);
  push(6);
  print_stack();
  isEmpty();
  cout << "Top: "
       << top_element() << endl;
  pop();
  print_stack();
  cout << "Top: "
       << top_element() << endl;
   getch();
  return 0;
}
```

Output:-

```
Elements are: 6 3 5
Top: 6
Elements are: 3 5
Top: 3
```

Q5. Write C++ programs to implement the Queue ADT using a singly linked list.

```
#include<iostream>
#include<conio.h>
using namespace std;
class Node
{
public:
  int data;
  Node *next;
};
void enqueue (Node ** head, int data)
{
  Node *new_node = new Node ();
  // assign data value
  new_node->data = data;
  // change the next node of this new_node
  // to current head of Linked List
  new_node->next = *head;
  //changing the new head to this newly entered node
  *head = new_node;
}
void dequeue (Node ** head)
{
  Node *temp = *head;
  // if there are no nodes in Linked List can't delete
  if (*head == NULL)
    {
      cout << ("Linked List Empty, nothing to delete");
      return;
    }
  // move head to next node
  *head = (*head)->next;
  //cout<< ("Deleted: %d\n", temp->data);
  delete (temp);
}
void display (Node * node)
{
  //as linked list will end when Node is Null
  while (node != NULL)
    {
      cout << node->data << " ";
      node = node->next;
    }
  cout << endl;
}
int main ()
```

```
{
  Node *head = NULL;
  enqueue (&head, 10);
  enqueue (&head, 11);
  enqueue (&head, 12);
  enqueue (&head, 13);
  enqueue (&head, 14);
  enqueue (&head, 15);
  enqueue (&head, 16);
  enqueue (&head, 17);
  enqueue (&head, 18);
  cout << "Queue before deletion: ";
  display (head);
  dequeue (&head);
  cout << endl << "Queue after deletion: ";
  display (head);
  getch();
  return 0;
}
```

Output:-

Queue before deletion: 18 17 16 15 14 13 12 11 10

Queue after deletion: 17 16 15 14 13 12 11 10

Q6. Write C++ programs to implement the deque (double-ended queue) ADT using a doubly-linked list.

```
#include <iostream>
using namespace std;
struct Node {
	int data;
	Node *prev, *next;
	static Node* getnode(int data)
	{
		Node* newNode =new Node;
		newNode->data = data;
		newNode->prev = newNode->next = NULL;
		return newNode;
	}
};
// A structure to represent a deque
class Deque {
	Node* front;
	Node* rear;
	int Size;
public:
	Deque()
	{
		front = rear = NULL;
		Size = 0;
	}
	// Operations on Deque
	void insertFront(int data);
	void insertRear(int data);
	void deleteFront();
	void deleteRear();
	int getFront();
	int getRear();
	int size();
	bool isEmpty();
	void erase();
};
// Function to check whether deque
// is empty or not
bool Deque::isEmpty() { return (front == NULL); }
int Deque::size() { return Size; }
// Function to insert an element
// at the front end
void Deque::insertFront(int data)
{
	Node* newNode = Node::getnode(data);
	// If true then new element cannot be added
```

```
	// and it is an 'Overflow' condition
	if (newNode == NULL)
		cout << "OverFlow\n";
	else {
		// If deque is empty
		if (front == NULL)
			rear = front = newNode;

		// Inserts node at the front end
		else {
			newNode->next = front;
			front->prev = newNode;
			front = newNode;
		}
		// Increments count of elements by 1
		Size++;
	}
}
// Function to insert an element
// at the rear end
void Deque::insertRear(int data)
{
	Node* newNode = Node::getnode(data);
	// If true then new element cannot be added
	// and it is an 'Overflow' condition
	if (newNode == NULL)
		cout << "OverFlow\n";
	else {
		// If deque is empty
		if (rear == NULL)
			front = rear = newNode;
		// Inserts node at the rear end
		else {
			newNode->prev = rear;
			rear->next = newNode;
			rear = newNode;
		}

		Size++;
	}
}
// Function to delete the element
// from the front end
void Deque::deleteFront()
{
	// If deque is empty then
	// 'Underflow' condition
	if (isEmpty())
```

```
                cout << "UnderFlow\n";
        // Deletes the node from the front end and makes
        // the adjustment in the links
        else {
                Node* temp = front;
                front = front->next;
                // If only one element was present
                if (front == NULL)
                        rear = NULL;
                else
                        front->prev = NULL;
                delete(temp);
                // Decrements count of elements by 1
                Size--;
        }
}
// Function to delete the element
// from the rear end
void Deque::deleteRear()
{
        // If deque is empty then
        // 'Underflow' condition
        if (isEmpty())
                cout << "UnderFlow\n";
        // Deletes the node from the rear end and makes
        // the adjustment in the links
        else {
                Node* temp = rear;
                rear = rear->prev;
                if (rear == NULL)
                        front = NULL;
                else
                        rear->next = NULL;
                delete(temp);
                Size--;
        }
}
// Function to return the element
// at the front end
int Deque::getFront()
{
        // If deque is empty, then returns
        // garbage value
        if (isEmpty())
                return -1;
        return front->data;
}
int Deque::getRear()
```

```
{
	// If deque is empty, then returns
	// garbage value
	if (isEmpty())
		return -1;
	return rear->data;
}
void Deque::erase()
{
	rear = NULL;
	while (front != NULL) {
		Node* temp = front;
		front = front->next;
		delete(temp);
	}
	Size = 0;
}
int main()
{
	Deque dq;
	cout << "Insert element '5' at rear end\n";
	dq.insertRear(5);
	cout << "Insert element '10' at rear end\n";
	dq.insertRear(10);
	cout << "Rear end element: " << dq.getRear() << endl;
	dq.deleteRear();
	cout << "After deleting rear element new rear"
		<< " is: " << dq.getRear() << endl;
	cout << "Inserting element '15' at front end \n";
	dq.insertFront(15);
	cout << "Front end element: " << dq.getFront() << endl;
	cout << "Number of elements in Deque: " << dq.size()
		<< endl;
	dq.deleteFront();
	cout << "After deleting front element new "
		<< "front is: " << dq.getFront() << endl;
	return 0;
}
```

Output:-

Insert element '5' at rear end
Insert element '10' at rear end
Rear end element: 10
After deleting rear element new rear is: 5
Inserting element '15' at front end
Front end element: 15
Number of elements in Deque: 2

After deleting front element new front is: 5

Assignment-3

Q1.) Write a C++ program to implement the Merge Sort.

```
#include <iostream>
using namespace std;
void merge(int arr[], int p, int q, int r) {
 int n1 = q - p + 1;
 int n2 = r - q;
 int L[n1], M[n2];
 for (int i = 0; i < n1; i++)
 L[i] = arr[p + i];
 for (int j = 0; j < n2; j++)
 M[j] = arr[q + 1 + j];
 int i, j, k;
 i = 0;
 j = 0;
 k = p;
 while (i < n1 && j < n2) {
 if (L[i] <= M[j]) {
 arr[k] = L[i];
 i++;}
else {
 arr[k] = M[j];
 j++;
 }
 k++;
 }
 while (i < n1) {
 arr[k] = L[i];
 i++;
 k++;
 }
 while (j < n2) {
 arr[k] = M[j];
 j++;
 k++;
 }
}
void mergeSort(int arr[], int l, int r) {
 if (l < r) {
 int m = l + (r - l) / 2;
 mergeSort(arr, l, m);
 mergeSort(arr, m + 1, r);
 merge(arr, l, m, r);
 }}
void printArray(int arr[], int size) {
 for (int i = 0; i < size; i++)
```

```
 cout << arr[i] << " ";
 cout << endl;
}
int main() {
 int arr[] = {6, 5, 12, 10, 9, 1};
 int size = sizeof(arr) / sizeof(arr[0]);
 mergeSort(arr, 0, size - 1);
 cout << "Sorted array: \n";
 printArray(arr, size);
 return 0;}
```

Output:-

Sorted array:
1 5 6 9 10 12

Q2. Write a C++ program to implement the Heap Sort.

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

 void heapify(int arr[], int n, int i) {
 int largest = i;
 int left = 2 * i + 1;
 int right = 2 * i + 2;

 if (left < n && arr[left] > arr[largest])
 largest = left;
 if (right < n && arr[right] > arr[largest])
 largest = right;
 if (largest != i) {
 swap(arr[i], arr[largest]);
 heapify(arr, n, largest);
 }
 }

 void heapSort(int arr[], int n) {
 for (int i = n / 2 - 1; i >= 0; i--)
 heapify(arr, n, i);
 for (int i = n - 1; i >= 0; i--) {
 swap(arr[0], arr[i]);


 heapify(arr, i, 0);
 }
 }
 void printArray(int arr[], int n) {
 for (int i = 0; i < n; ++i)
 cout << arr[i] << " ";
 cout << "\n";
 }

 int main() {
 int arr[] = {1, 12, 9, 5, 6, 10};
 int n = sizeof(arr) / sizeof(arr[0]);
 heapSort(arr, n);
 cout << "Sorted array is \n";
 printArray(arr, n);
 return 0;
}
```

Output:-

Sorted array is
1 5 6 9 10 12

Q3Write a C++ program to perform the following operations.

a) Insert an element into a binary search tree.

b) Delete an element from a binary search tree.

c) Search for a key element in a binary search tree.

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

class BST {
	int data;
	BST *left, *right;

public:
	BST();
	BST(int);
	BST* Insert(BST*, int);
	void Inorder(BST*);
	bool search(BST*,int);
	void searchData(BST*);
	bool deleteD(BST* root, int val);
};
BST::BST(){
	 data=0;
	 left=NULL;
	 right=NULL;
}
BST::BST(int value)
{
	data = value;
	left = right = NULL;
}
BST* BST::Insert(BST* root, int value)
{
	if (!root) {
		return new BST(value);
	}
	if (value > root->data) {
		root->right = Insert(root->right, value);
	}
	else if (value < root->data) {
		root->left = Insert(root->left, value);
	}
	return root;
}
// Inorder traversal function.
// This gives data in sorted order.
void BST::Inorder(BST* root)
{
```

```
        if (!root) {
                return;
        }
        Inorder(root->left);
        cout << root->data << " ";
        Inorder(root->right);
}
bool BST::search(BST* root, int data)
{

    // Base Cases: root is null or key is present at root
    if (root == NULL || root->data == data)
        return root;

    // Key is greater than root's key
    if (root->data < data)
        return search(root->right, data);

    // Key is smaller than root's key
    return search(root->left, data);
}
 void BST::searchData(BST* root){
        int val;
        cout<<endl<<"Enter the value for search"<<endl;
        cin>>val;
        if (search(root, val) == NULL)
      cout <<endl<< val<< " not found" << endl;
    else
        cout <<endl<< val << " found" << endl;
 }
bool BST::deleteD(BST* root, int val)
{
    // Base case
    if (root == NULL)
        return root;

    // Recursive calls for ancestors of
    // node to be deleted
    if (root->data > val) {
        root->left = (BST*)deleteD(root->left, val);
        return root;
    }
    else if (root->data < val) {
        root->right = (BST*)deleteD(root->right, val);
        return root;
    }

    // We reach here when root is the node
```

```
    // to be deleted.

    // If one of the children is empty
    if (root->left == NULL) {
        BST* temp = root->right;
        delete root;
        return temp;
    }
    else if (root->right == NULL) {
        BST* temp = root->left;
        delete root;
        return temp;
    }

    // If both children exist
    else {

        BST* succParent = root;

        // Find successor
        BST* succ = root->right;
        while (succ->left != NULL) {
            succParent = succ;
            succ = succ->left;
        }

        // Delete successor.  Since successor
        // is always left child of its parent
        // we can safely make successor's right
        // right child as left of its parent.
        // If there is no succ, then assign
        // succ->right to succParent->right
        if (succParent != root)
            succParent->left = succ->right;
        else
            succParent->right = succ->right;
        // Copy Successor Data to root
        root->data = succ->data;
        // Delete Successor and return root
        delete succ;
        return root;
    }
}
int main()
{
            BST b, *root = NULL;
            root = b.Insert(root, 50);
            b.Insert(root, 30);
```

```
        b.Insert(root, 20);
        b.Insert(root, 40);
        b.Insert(root, 70);
        b.Insert(root, 60);
        b.Insert(root, 80);

        b.Inorder(root);
        b.searchData(root);
        int val=30;
        cout<<endl<<"Delete a Leaf Node :30"<<endl;
        b.deleteD(root,val);
        return 0;
}
```

Output:-

```
20 30 40 50 60 70 80
Enter the value for search
20

20 found

Delete a Leaf Node :30
```

Q4. Write C++ programs that use recursive functions to traverse the given binary tree in.

a) Preorder

```
#include<iostream>
#include <stdlib.h>
using namespace std;
struct node {
 int data;
 struct node *left;
 struct node *right;
};
struct node *createNode(int val) {
 struct node *temp = (struct node *)malloc(sizeof(struct node));
 temp->data = val;
 temp->left = temp->right = NULL;
 return temp;
}
void preorder(struct node *root) {
 if (root != NULL) {
 cout<<root->data<<" ";
 preorder(root->left);
 preorder(root->right);
 }
}
struct node* insertNode(struct node* node, int val) {
 if (node == NULL) return createNode(val);
 if (val < node->data)
 node->left = insertNode(node->left, val);
 else if (val > node->data)
 node->right = insertNode(node->right, val);
 return node;
}
 int main() {
 struct node *root = NULL;
 root = insertNode(root, 4);
 insertNode(root, 5);
 insertNode(root, 2);
 insertNode(root, 9);
 insertNode(root, 1);
 insertNode(root, 3);
 cout<<"Pre-Order traversal of the Binary Search Tree is: ";
 preorder(root);
 return 0;
}
```

Output:-

```
Pre-Order traversal of the Binary Search Tree is: 4 2 1 3 5 9
```

b) Inorder

```
#include<iostream>
#include <stdlib.h>
using namespace std;
struct node {
 int data;
 struct node *left;
 struct node *right;
};
struct node *createNode(int val) {
 struct node *temp = (struct node *)malloc(sizeof(struct node));
 temp->data = val;
 temp->left = temp->right = NULL;
 return temp;
}
void inorder(struct node *root) {
 if (root != NULL) {
 inorder(root->left);
 cout<<root->data<<" ";
 inorder(root->right);
 }
}
struct node* insertNode(struct node* node, int val) {
 if (node == NULL) return createNode(val);
 if (val < node->data)
 node->left = insertNode(node->left, val);
 else if (val > node->data)
 node->right = insertNode(node->right, val);
 return node;
}
int main() {
 struct node *root = NULL;
 root = insertNode(root, 4);
 insertNode(root, 5);
 insertNode(root, 2);
 insertNode(root, 9);
 insertNode(root, 1);
 insertNode(root, 3);
 cout<<"In-Order traversal of the Binary Search Tree is: ";
 inorder(root);
 return 0;
}
```

Output:-

In-Order traversal of the Binary Search Tree is: 1 2 3 4 5 9

c) Postorder

```
#include<iostream>
#include <stdlib.h>
using namespace std;
struct node {
 int data;
 struct node *left;
 struct node *right;
};
struct node *createNode(int val) {
 struct node *temp = (struct node *)malloc(sizeof(struct node));
 temp->data = val;
 temp->left = temp->right = NULL;
 return temp;
}
void postorder(struct node *root) {
 if (root != NULL) {
 postorder(root->left);
 postorder(root->right);
 cout<<root->data<<" ";
 }
}
struct node* insertNode(struct node* node, int val) {
 if (node == NULL) return createNode(val);
 if (val < node->data)
 node->left = insertNode(node->left, val);
 else if (val > node->data)
 node->right = insertNode(node->right, val);
 return node;
}
int main() {
 struct node *root = NULL;
 root = insertNode(root, 4);
 insertNode(root, 5);
 insertNode(root, 2);
 insertNode(root, 9);
 insertNode(root, 1);
 insertNode(root, 3);
 cout<<"Post-Order traversal of the Binary Search Tree is: ";
 postorder(root);
 return 0;
}
```

Output:-

Post-Order traversal of the Binary Search Tree is: 1 3 2 9 5 4

Q5. Write a C++ program to perform the following operations
a) Insertion into a B-tree
b) Deletion from a B-tree

```
#include <iostream>
using namespace std;
class BTreeNode {
  int *keys;
  int t;
  BTreeNode **C;
  int n;
  bool leaf;
   public:
  BTreeNode(int _t, bool _leaf);
  void traverse();
  int findKey(int k);
  void insertNonFull(int k);
  void splitChild(int i, BTreeNode *y);
  void deletion(int k);
  void removeFromLeaf(int idx);
  void removeFromNonLeaf(int idx);
  int getPredecessor(int idx);
  int getSuccessor(int idx);
  void fill(int idx);
  void borrowFromPrev(int idx);
  void borrowFromNext(int idx);
  void merge(int idx);
  friend class BTree;
};
class BTree {
  BTreeNode *root;
  int t;
   public:
  BTree(int _t) {
    root = NULL;
    t = _t;
  }
```

```
  void traverse() {
    if (root != NULL)
      root->traverse();
  }
  void insertion(int k);
  void deletion(int k);
};
// B tree node
BTreeNode::BTreeNode(int t1, bool leaf1) {
  t = t1;
  leaf = leaf1;
  keys = new int[2 * t - 1];
  C = new BTreeNode *[2 * t];
  n = 0;
}
// Find the key
int BTreeNode::findKey(int k) {
  int idx = 0;
  while (idx < n && keys[idx] < k)
    ++idx;
  return idx;
}
// Deletion operation
void BTreeNode::deletion(int k) {
  int idx = findKey(k);
  if (idx < n && keys[idx] == k) {
    if (leaf)
      removeFromLeaf(idx);
    else
      removeFromNonLeaf(idx);
  } else {
    if (leaf) {
      cout << "The key " << k << " is does not exist in the tree\n";
      return;
    }
    bool flag = ((idx == n) ? true : false);
    if (C[idx]->n < t)
```

```
      fill(idx);
    if (flag && idx > n)
      C[idx - 1]->deletion(k);
    else
      C[idx]->deletion(k);
  }
  return;
}
// Remove from the leaf
void BTreeNode::removeFromLeaf(int idx) {
  for (int i = idx + 1; i < n; ++i)
    keys[i - 1] = keys[i];
  n--;
  return;
}
// Delete from non leaf node
void BTreeNode::removeFromNonLeaf(int idx) {
  int k = keys[idx];
  if (C[idx]->n >= t) {
    int pred = getPredecessor(idx);
    keys[idx] = pred;
    C[idx]->deletion(pred);
  }
  else if (C[idx + 1]->n >= t) {
    int succ = getSuccessor(idx);
    keys[idx] = succ;
    C[idx + 1]->deletion(succ);
  }
  else {
    merge(idx);
    C[idx]->deletion(k);
  }
  return;
}
int BTreeNode::getPredecessor(int idx) {
  BTreeNode *cur = C[idx];
  while (!cur->leaf)
```

```
    cur = cur->C[cur->n];
  return cur->keys[cur->n - 1];
}
int BTreeNode::getSuccessor(int idx) {
  BTreeNode *cur = C[idx + 1];
  while (!cur->leaf)
    cur = cur->C[0];
  return cur->keys[0];
}
void BTreeNode::fill(int idx) {
  if (idx != 0 && C[idx - 1]->n >= t)
    borrowFromPrev(idx);
  else if (idx != n && C[idx + 1]->n >= t)
    borrowFromNext(idx);
  else {
    if (idx != n)
      merge(idx);
    else
      merge(idx - 1);
  }
  return;
}
// Borrow from previous
void BTreeNode::borrowFromPrev(int idx) {
  BTreeNode *child = C[idx];
  BTreeNode *sibling = C[idx - 1];
  for (int i = child->n - 1; i >= 0; --i)
    child->keys[i + 1] = child->keys[i];
  if (!child->leaf) {
    for (int i = child->n; i >= 0; --i)
      child->C[i + 1] = child->C[i];
  }
  child->keys[0] = keys[idx - 1];
  if (!child->leaf)
    child->C[0] = sibling->C[sibling->n];
  keys[idx - 1] = sibling->keys[sibling->n - 1];
  child->n += 1;
```

```
  sibling->n -= 1;
  return;
}
// Borrow from the next
void BTreeNode::borrowFromNext(int idx) {
  BTreeNode *child = C[idx];
  BTreeNode *sibling = C[idx + 1];
  child->keys[(child->n)] = keys[idx];
  if (!(child->leaf))
    child->C[(child->n) + 1] = sibling->C[0];
  keys[idx] = sibling->keys[0];
  for (int i = 1; i < sibling->n; ++i)
    sibling->keys[i - 1] = sibling->keys[i];
  if (!sibling->leaf) {
    for (int i = 1; i <= sibling->n; ++i)
      sibling->C[i - 1] = sibling->C[i];
  }
  child->n += 1;
  sibling->n -= 1;
  return;
}
// Merge
void BTreeNode::merge(int idx) {
  BTreeNode *child = C[idx];
  BTreeNode *sibling = C[idx + 1];
  child->keys[t - 1] = keys[idx];
  for (int i = 0; i < sibling->n; ++i)
    child->keys[i + t] = sibling->keys[i];
  if (!child->leaf) {
    for (int i = 0; i <= sibling->n; ++i)
      child->C[i + t] = sibling->C[i];
  }
  for (int i = idx + 1; i < n; ++i)
    keys[i - 1] = keys[i];
  for (int i = idx + 2; i <= n; ++i)
    C[i - 1] = C[i];
  child->n += sibling->n + 1;
```

```
  n--;
  delete (sibling);
  return;
}
// Insertion operation
void BTree::insertion(int k) {
  if (root == NULL) {
    root = new BTreeNode(t, true);
    root->keys[0] = k;
    root->n = 1;
  } else {
    if (root->n == 2 * t - 1) {
      BTreeNode *s = new BTreeNode(t, false);
      s->C[0] = root;
      s->splitChild(0, root);
      int i = 0;
      if (s->keys[0] < k)
        i++;
      s->C[i]->insertNonFull(k);
      root = s;
    } else
      root->insertNonFull(k);
  }
}
// Insertion non full
void BTreeNode::insertNonFull(int k) {
  int i = n - 1;
  if (leaf == true) {
    while (i >= 0 && keys[i] > k) {
      keys[i + 1] = keys[i];
      i--;
    }
    keys[i + 1] = k;
    n = n + 1;
  } else {
    while (i >= 0 && keys[i] > k)
      i--;
```

```
    if (C[i + 1]->n == 2 * t - 1) {
      splitChild(i + 1, C[i + 1]);
      if (keys[i + 1] < k)
        i++;
    }
    C[i + 1]->insertNonFull(k);
  }
}
// Split child
void BTreeNode::splitChild(int i, BTreeNode *y) {
  BTreeNode *z = new BTreeNode(y->t, y->leaf);
  z->n = t - 1;
  for (int j = 0; j < t - 1; j++)
    z->keys[j] = y->keys[j + t];
  if (y->leaf == false) {
    for (int j = 0; j < t; j++)
      z->C[j] = y->C[j + t];
  }
  y->n = t - 1;
  for (int j = n; j >= i + 1; j--)
    C[j + 1] = C[j];
  C[i + 1] = z;
  for (int j = n - 1; j >= i; j--)
    keys[j + 1] = keys[j];
  keys[i] = y->keys[t - 1];
  n = n + 1;
}
// Traverse
void BTreeNode::traverse() {
  int i;
  for (i = 0; i < n; i++) {
    if (leaf == false)
      C[i]->traverse();
    cout << " " << keys[i];
  }
  if (leaf == false)
    C[i]->traverse();
```

```
}
// Delete Operation
void BTree::deletion(int k) {
  if (!root) {
    cout << "The tree is empty\n";
    return;
  }
  root->deletion(k);
  if (root->n == 0) {
    BTreeNode *tmp = root;
    if (root->leaf)
      root = NULL;
    else
      root = root->C[0];
    delete tmp;
  }
  return;
}
int main() {
  BTree t(3);
  t.insertion(8);
  t.insertion(9);
  t.insertion(10);
  t.insertion(11);
  t.insertion(15);
  t.insertion(16);
  t.insertion(17);
  t.insertion(18);
  t.insertion(20);
  t.insertion(23);
  cout << "The B-tree is: ";
  t.traverse();
  t.deletion(20);
  cout << "\nThe B-tree is: ";
  t.traverse();
}
```

Output:-

The B-tree is: 8 9 10 11 15 16 17 18 20 23

The B-tree is: 8 9 10 11 15 16 17 18 23

Q6. Write a C++ program to perform the following operations

a) Insertion into an AVL-tree

b) Deletion from an AVL-tree

```
#include<iostream>
using namespace std;
class Node
{
	public:
	int key;
	Node *left;
	Node *right;
	int height;
};
int max(int a, int b);
int height(Node *N)
{
	if (N == NULL)
		return 0;
	return N->height;
}
int max(int a, int b)
{
	return (a > b)? a : b;
}
Node* newNode(int key)
{
	Node* node = new Node();
	node->key = key;
	node->left = NULL;
	node->right = NULL;
	node->height = 1;
	return(node);
}
Node *rightRotate(Node *y)
{
	Node *x = y->left;
```

```
        Node *T2 = x->right;
        x->right = y;
        y->left = T2;
        y->height = max(height(y->left),height(y->right)) + 1;
        x->height = max(height(x->left),height(x->right)) + 1;
        return x;
}
Node *leftRotate(Node *x)
{
        Node *y = x->right;
        Node *T2 = y->left;
        y->left = x;
        x->right = T2;
        x->height = max(height(x->left),
                                height(x->right)) + 1;
        y->height = max(height(y->left),
                                height(y->right)) + 1;
        return y;
}
int getBalance(Node *N)
{
        if (N == NULL)
                return 0;
        return height(N->left) -
                height(N->right);
}
Node* insert(Node* node, int key)
{
        if (node == NULL)
                return(newNode(key));
        if (key < node->key)
                node->left = insert(node->left, key);
        else if (key > node->key)
                node->right = insert(node->right, key);
        else
                return node;
        node->height = 1 + max(height(node->left),
```

```
                                        height(node->right));
        int balance = getBalance(node);
        if (balance > 1 && key < node->left->key)
                return rightRotate(node);
        if (balance < -1 && key > node->right->key)
                return leftRotate(node);
        if (balance > 1 && key > node->left->key)
        {
                node->left = leftRotate(node->left);
                return rightRotate(node);
        }
        if (balance < -1 && key < node->right->key)
        {
                node->right = rightRotate(node->right);
                return leftRotate(node);
        }
        return node;
}
Node * minValueNode(Node* node)
{
        Node* current = node;
        while (current->left != NULL)
                current = current->left;
        return current;
}
Node* deleteNode(Node* root, int key)
{
        if (root == NULL)
                return root;
        if ( key < root->key )
                root->left = deleteNode(root->left, key);
        else if( key > root->key )
                root->right = deleteNode(root->right, key);
        else
        {
                if( (root->left == NULL) ||
                        (root->right == NULL) )
```

```
        {
                Node *temp = root->left ?
                                        root->left :
                                        root->right;
                if (temp == NULL)
                {
                        temp = root;
                        root = NULL;
                }
                else
                *root = *temp;
                delete(temp);
        }
        else
        {
                Node* temp = minValueNode(root->right);
                root->key = temp->key;
                root->right = deleteNode(root->right,
                                                        temp->key);
        }
}
if (root == NULL)
return root;
root->height = 1 + max(height(root->left),
                                        height(root->right));
int balance = getBalance(root);
if (balance > 1 &&
        getBalance(root->left) >= 0)
        return rightRotate(root);
if (balance > 1 &&
        getBalance(root->left) < 0)
{
        root->left = leftRotate(root->left);
        return rightRotate(root);
}
if (balance < -1 && getBalance(root->right) <= 0)
        return leftRotate(root);
```

```
        if (balance < -1 && getBalance(root->right) > 0)
        {
                root->right = rightRotate(root->right);
                return leftRotate(root);
        }
        return root;
}
void preOrder(Node *root)
{
        if(root != NULL)
        {
                cout << root->key << " ";
                preOrder(root->left);
                preOrder(root->right);
        }
}
int main()
{
Node *root = NULL;
        root = insert(root, 9);
        root = insert(root, 5);
        root = insert(root, 10);
        root = insert(root, 0);
        root = insert(root, 6);
        root = insert(root, 11);
        root = insert(root, -1);
        root = insert(root, 1);
        root = insert(root, 2);
        cout << "Preorder traversal of the "
                        "constructed AVL tree is \n";
        preOrder(root);
        root = deleteNode(root, 10);
        cout << "\nPreorder traversal after"
                << " deletion of 10 \n";
        preOrder(root);
        return 0;
}
```

Output:-
Preorder traversal of the constructed AVL tree is
9 1 0 -1 5 2 6 10 11
Preorder traversal after deletion of 10
1 0 -1 9 5 2 6 11