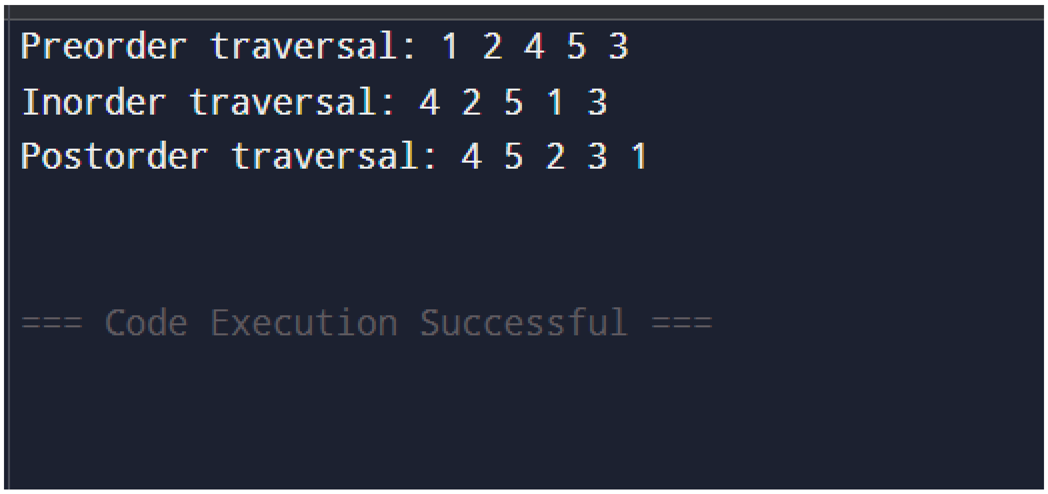
DSA ASSIGNMENT 8

QUESTION 1

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

struct Node {
    int data;
    Node* left;
    Node* right;
};

Node* createNode(int value) {
    Node* newNode=new Node();
    newNode->data=value;
    newNode->left=nullptr;
    newNode->right=nullptr;
    return newNode;
}

void preorder(Node* root) {
    if(root==nullptr)
        return;
    cout<<root->data<<" ";
    preorder(root->left);
    preorder(root->right);
}

void inorder(Node* root) {
    if(root==nullptr)
        return;
    inorder(root->left);
    cout<<root->data<<" ";
    inorder(root->right);
}

void postorder(Node* root) {
    if(root==nullptr)
        return;
    postorder(root->left);
    postorder(root->right);
    cout<<root->data<<" ";
}

int main(){
    Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);

    cout<<"Preorder traversal: ";
    preorder(root);
    cout<<endl;

    cout<<"Inorder traversal: ";
    inorder(root);
    cout<<endl;
    cout<<"Postorder traversal: ";
    postorder(root);
    cout<<endl;

    return 0;
}
```



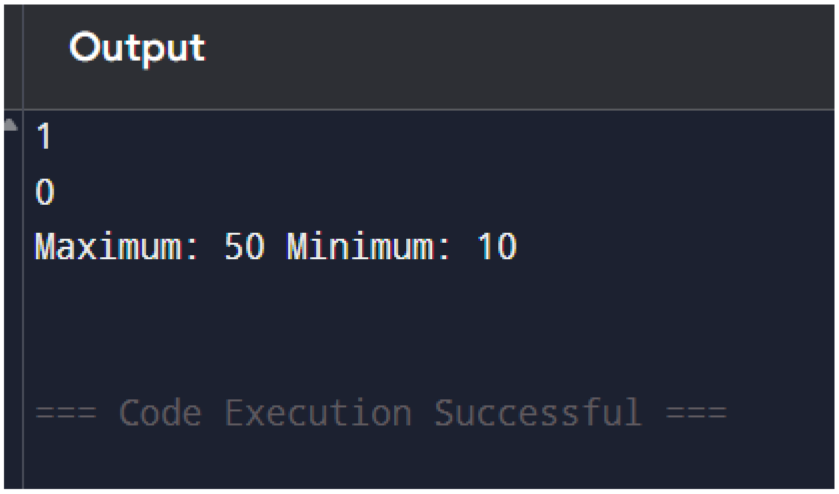
```
Preorder traversal: 1 2 4 5 3
Inorder traversal: 4 2 5 1 3
Postorder traversal: 4 5 2 3 1



=== Code Execution Successful ===
```

QUESTION 2

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

struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int val) {
        data = val;
        left = right = nullptr;
    }
};

Node* insertNode(Node* root, int value) {
    if (root == nullptr) {
        return new Node(value);
    }
    if (value < root->data) {
        root->left = insertNode(root->left, value);
    }
    else if (value > root->data) {
        root->right = insertNode(root->right, value);
    }
    return root;
}
```

```
bool searchRecursive(Node* root, int value) {
    if (root == nullptr) {
        return false;
    }
    if (root->data == value) {
        return true;
    }
    if (value < root->data) {
        return searchRecursive(root->left, value);
    } else {
        return searchRecursive(root->right, value);
    }

    return false; // FIX for warning
}

bool searchNonRecursive(Node* root, int value) {
    Node* curr = root;

    while (curr != nullptr) {
        if (value == curr->data) {
            return true;
        }
        else if (value < curr->data) {
            curr = curr->left;
        }
        else {
            curr = curr->right;
        }
    }
    return false;
}
```

```
int maximumElement(Node* root) {
    if (root == nullptr)
        return -1;

    Node* curr = root;
    while (curr->right != nullptr) {
        curr = curr->right;
    }
    return curr->data;
}

int minimumElement(Node* root) {
    if (root == nullptr)
        return -1;

    Node* curr = root;
    while (curr->left != nullptr) {
        curr = curr->left;
    }
    return curr->data;
}

Node* inorderSuccessor(Node* root, Node* target) {
    Node* succ = nullptr;

    while (root != nullptr) {
        if (target->data < root->data) {
            succ = root;
            root = root->left;
        }
        else if (target->data > root->data) {
            root = root->right;
        }
        else {
            if (root->right != nullptr) {
                Node* temp = root->right;
                while (temp->left != nullptr)
                    temp = temp->left;
                succ = temp;
            }
            break;
        }
    }
    return succ;
}

Node* inorderPredecessor(Node* root, Node* target) {
    Node* pred = nullptr;

    while (root != nullptr) {
        if (target->data > root->data) {
            pred = root;
            root = root->right;
        }
        else if (target->data < root->data) {
            root = root->left;
        }
        else {
            if (root->left != nullptr) {
                Node* temp = root->left;
                while (temp->right != nullptr)
                    temp = temp->right;
                pred = temp;
            }
            break;
        }
    }
    return pred;
}

int main() {
    Node* root = nullptr;
    root = insertNode(root, 10);
    root = insertNode(root, 40);
    root = insertNode(root, 30);
    root = insertNode(root, 50);
    root = insertNode(root, 20);

    bool found;
    found = searchNonRecursive(root, 20);
    cout << found << endl;

    found = searchRecursive(root, 60);
    cout << found << endl;

    int max, min;
    max = maximumElement(root);
    min = minimumElement(root);
    cout << "Maximum: " << max << " Minimum: " << min << endl;

    return 0;
}
```



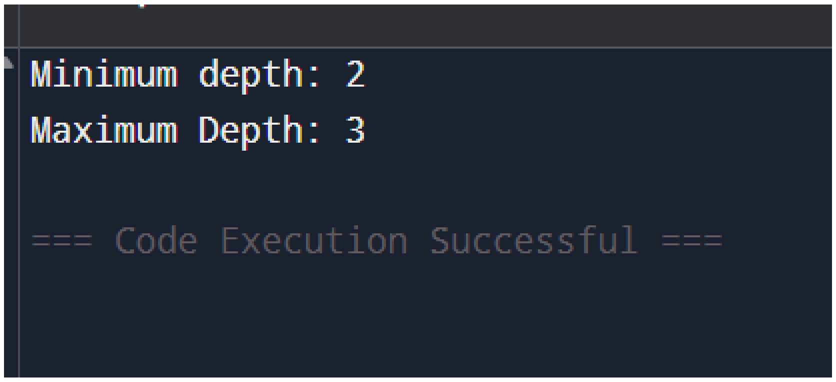
```
1
0
Maximum: 50 Minimum: 10


=== Code Execution Successful ===
```

QUESTION 3

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

struct Node{
  int data;
  Node*left;
  Node*right;

  Node(int val){
    data=val;
    left=right=nullptr;
  }
};

Node* insertElement(Node*root, int value){
  if (root==nullptr){
    return new Node(value);
  }
  if(value==root->data){
    return root;}
  if(value<root->data){
    root->left=insertElement(root->left, value);
  }
  else if(value>root->data){
    root->right=insertElement(root->right,value);
  }
  return root;
}
```

```
Node* findMin(Node* root) {
    while(root->left!=nullptr)
        root=root->left;
    return root;
}

Node* deleteElement(Node*root, int value){
    if(root==nullptr){
        return root;
    }
    if (value<root->data){
        root->left=deleteElement(root->left, value);
    }
    else if(value>root->data){
        root->right=deleteElement(root->right,value);
    }
    else{
        if(root->left==nullptr && root->right==nullptr){
            delete root;
            root=nullptr;
        }
        else if(root->left==nullptr){
            Node*temp=root;
            root=root->right;
            delete temp;
        }
        else if(root->right==nullptr){
            Node*temp=root;
            root=root->left;
            delete temp;
        }
        else{
            Node*temp=findMin(root->right);
            root->data=temp->data;
            root->right=deleteElement(root->right,temp->data);
        }

    }
    return root;
}

int maxDepth(Node*root){
    if (root==nullptr){
        return 0;
    }
    int leftDepth=maxDepth(root->left);
    int rightDepth=maxDepth(root->right);

    return 1+max(leftDepth,rightDepth);
}

int minDepth(Node*root){
    if (root==nullptr){
        return 0;
    }
    if (root->left==nullptr){
        return 1+minDepth(root->left);
    }
    if(root->right==nullptr){
        return 1+minDepth(root->right);
    }

    return 1+min(minDepth(root->left), minDepth(root->right));
```

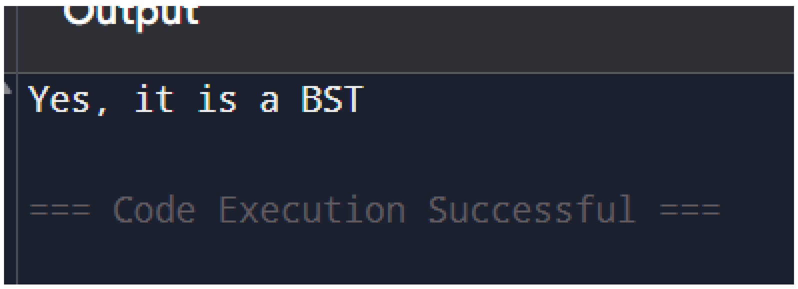
```
}

int main(){
    Node*root=nullptr;
    root=insertElement(root,10);
    root=insertElement(root,40);
    root=insertElement(root,30);
    root=insertElement(root,50);
    root=insertElement(root,20);

    root=deleteElement(root,10);
    int maxmDepth=maxDepth(root);
    int minmDepth=minDepth(root);

    cout<<"Minimum depth: "<<minmDepth<<"\nMaximum Depth: "<<maxmDepth;

    return 0;
}
```



```
Minimum depth: 2
Maximum Depth: 3

=== Code Execution Successful ===
```

QUESTION 4

```
#include <iostream>

#include <limits.h>

using namespace std;


struct Node{

    int data;

    Node* left;

    Node* right;


    Node(int value){

        data=value;

        left=right=nullptr;

    }

};


bool isBSTUtil(Node* root, int minVal, int maxVal){

    if (root == nullptr){

        return true;

    }


    if (root->data <= minVal || root->data >= maxVal){

        return false;

    }


    return isBSTUtil(root->left, minVal, root->data) && isBSTUtil(root->right, root->data, maxVal);

}


bool isBST(Node* root){

    return isBSTUtil(root, INT_MIN, INT_MAX);
}

int main(){
    Node* root=new Node(10);
    root->left=new Node(5);
    root->right=new Node(20);
    root->left->left=new Node(3);
    root->left->right=new Node(7);

    if(isBST(root))
        cout << "Yes, it is a BST";
    else
        cout << "No, it is NOT a BST";

    return 0;
}
```

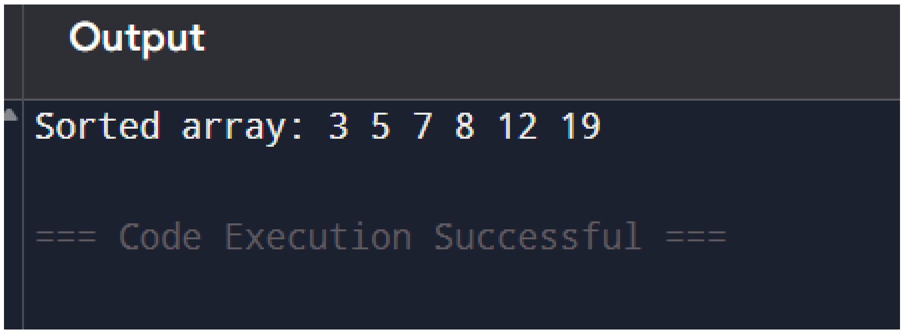


```
Yes, it is a BST

=== Code Execution Successful ===
```

QUESTION 5

```
#include <iostream>

using namespace std;


void heapifyMax(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]);

    heapifyMax(arr, n, largest);

  }

}


void heapSortIncreasing(int arr[], int n){

  for (int i=n/2-1;i>=0;i--)

    heapifyMax(arr, n, i);


  for (int i=n-1;i>0;i--) {

    swap(arr[0], arr[i]);

    heapifyMax(arr, i, 0);

  }

}
```

```
void heapifyMin(int arr[], int n, int i) {
    int smallest=i;
    int left=2*i+1;
    int right=2*i+2;
    if (left<n && arr[left]<arr[smallest])
        smallest = left;

    if (right< n && arr[right]<arr[smallest])
        smallest=right;

    if (smallest!=i){
        swap(arr[i], arr[smallest]);
        heapifyMin(arr, n, smallest);
    }
}

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

    for (int i=n-1;i>0;i--){
        swap(arr[0], arr[i]);
        heapifyMin(arr, i, 0);
    }
}
int main(){
    int arr[]={12, 3, 19, 8, 5, 7};
    int n = sizeof(arr)/sizeof(arr[0]);

    heapSortIncreasing(arr, n);

    cout << "Sorted array: ";
    for (int x : arr) cout << x << " ";
}
```

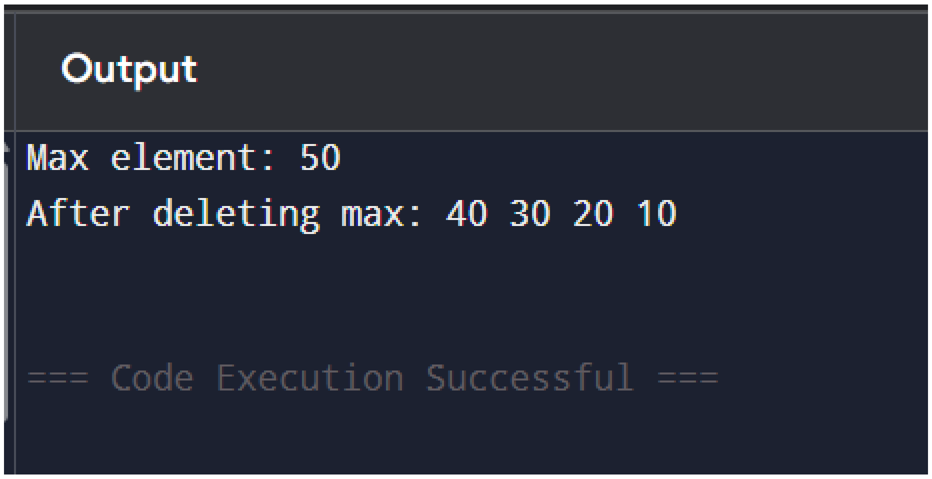

Output

```
Sorted array: 3 5 7 8 12 19


=== Code Execution Successful ===
```

QUESTION 6

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

#define MAX 100

int heapArr[MAX];
int heapSize = 0;

void heapifyUp(int index){
    while (index > 0){
        int parent = (index - 1) / 2;
        if (heapArr[parent] < heapArr[index]) {
            swap(heapArr[parent], heapArr[index]);
            index = parent;
        } else break;
    }
}
void heapifyDown(int index){
    while (true){
        int left = 2 * index + 1;
        int right = 2 * index + 2;
        int largest = index;

        if (left < heapSize && heapArr[left] > heapArr[largest])
            largest = left;

        if (right < heapSize && heapArr[right] > heapArr[largest])
            largest = right;

        if (largest != index) {
            swap(heapArr[index], heapArr[largest]);
            index = largest;
        } else break;
    }
}

void insertElement(int value){
    if (heapSize >= MAX) {
        cout << "Heap overflow!\n";
        return;
    }
    heapArr[heapSize] = value;
    heapifyUp(heapSize);
    heapSize++;
}
int getMax(){
    if (heapSize == 0){
        cout << "Heap empty!\n";
        return -1;
    }
    return heapArr[0];
}

void deleteMax(){
    if (heapSize == 0){
        cout << "Heap empty!\n";
        return;
    }

    heapArr[0] = heapArr[heapSize - 1];
    heapSize--;

    heapifyDown(0);
```

```
}

void printHeap(){
    for (int i = 0; i < heapSize; i++)
        cout << heapArr[i]<<" ";
    cout << endl;
}
int main(){
    insertElement(40);
    insertElement(10);
    insertElement(50);
    insertElement(30);
    insertElement(20);

    cout<<"Max element: "<<getMax()<<endl;

    deleteMax();
    cout << "After deleting max: ";
    printHeap();

    return 0;
}
```



```
Max element: 50
After deleting max: 40 30 20 10


=== Code Execution Successful ===
```