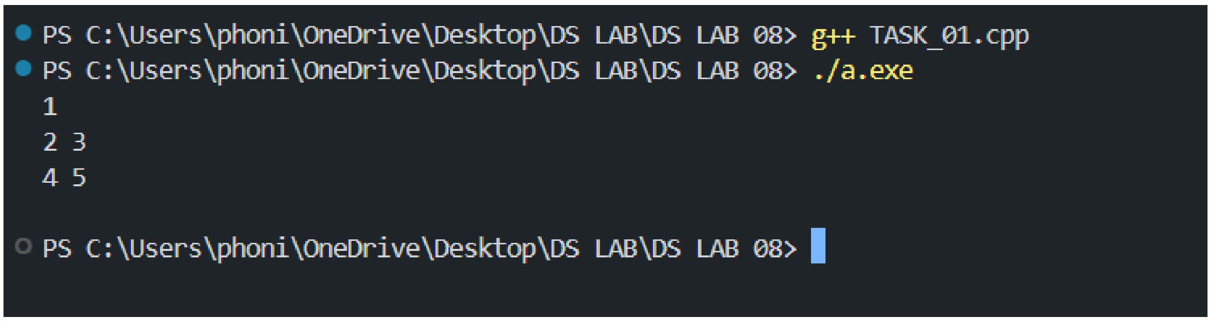
TASK 01

```
#include<iostream>

using namespace std;

template<typename T>
class Node{
    public:
        T key;
        Node<T>* left;
        Node<T>* right;
        Node(T val):key(val),left(nullptr),right(nullptr){};
};

template<typename T>
class Binary_tree{
    private:
        Node<T>* root;
    public:
        Binary_tree():root(nullptr){}
        /*
            There are multiple ways to input in Binary tree,
            like levelOrder Preorder, like in BST. Since it is
            not mention I will do levelOrder insertion;
        */
       void insert_levelOrder(T arr[],int size){
            root=insert_levelOrder(arr,size,0);
       }
        Node<T>* insert_levelOrder(T arr[],int size,int index){
            if(index>=size) return nullptr;

            Node<T>* newNode=new Node<T>(arr[index]);

            newNode->left=insert_levelOrder(arr,size,2*index+1);
            newNode->right=insert_levelOrder(arr,size,2*index+2);
            return newNode;
        }

        void preOrder_traversal(){
            preOrder_traversal(root);
        }

        void preOrder_traversal(Node<T>* root){
            if(!root) return;

            cout<<root->key<<" ";
            preOrder_traversal(root->left);
            preOrder_traversal(root->right);
        }

        void print_current_level(Node<T>* root,int level){
            if(!root) return;
            if(level==0) cout<<root->key<<" ";
            else if(level>0){
                print_current_level(root->left,level-1);
                print_current_level(root->right,level-1);
            }
        }
       int max(int i,int j,int k=0){
            int tmp=(j>k)? j:k;
            return (i>tmp)? i:tmp;
       }

        int height(Node<T>* root){
            if (!root) return -1;
                return max(height(root->left),height(root->right))+1;

        }

        void display_levelorder(){
            display_levelorder(root);
        }
        void display_levelorder(Node<T>* root){
            int h= height(root);
            for(int i=0 ; i<=h ; i++){
                print_current_level(root,i);
                cout<<endl;
            }
        }
};


int main(){

    Binary_tree<int> tree;
    int arr[] ={1, 2, 3, 4, 5};
    int size=sizeof(arr)/sizeof(arr[0]);

    tree.insert_levelOrder(arr,size);
    // tree.preOrder_traversal();
    tree.display_levelorder();
    return 0;
}
```



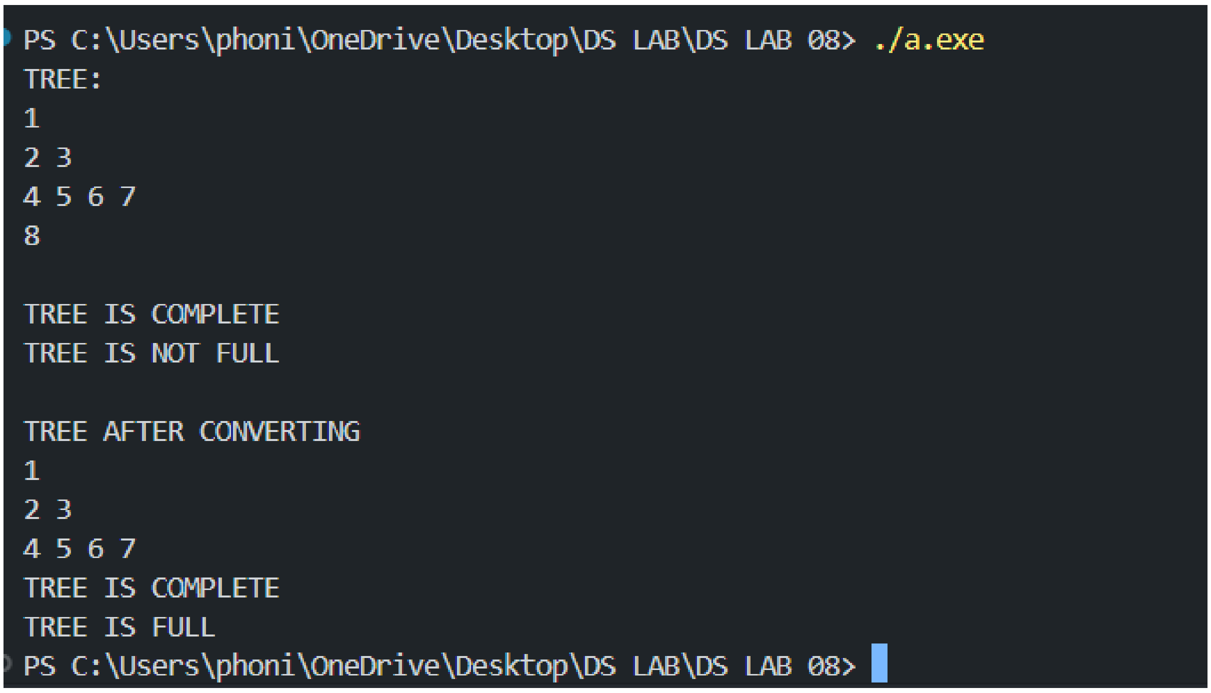
```
PS C:\Users\phoni\OneDrive\Desktop\DS LAB\DS LAB 08> g++ TASK_01.cpp
PS C:\Users\phoni\OneDrive\Desktop\DS LAB\DS LAB 08> ./a.exe
1
2 3
4 5

PS C:\Users\phoni\OneDrive\Desktop\DS LAB\DS LAB 08>
```

TASK 02

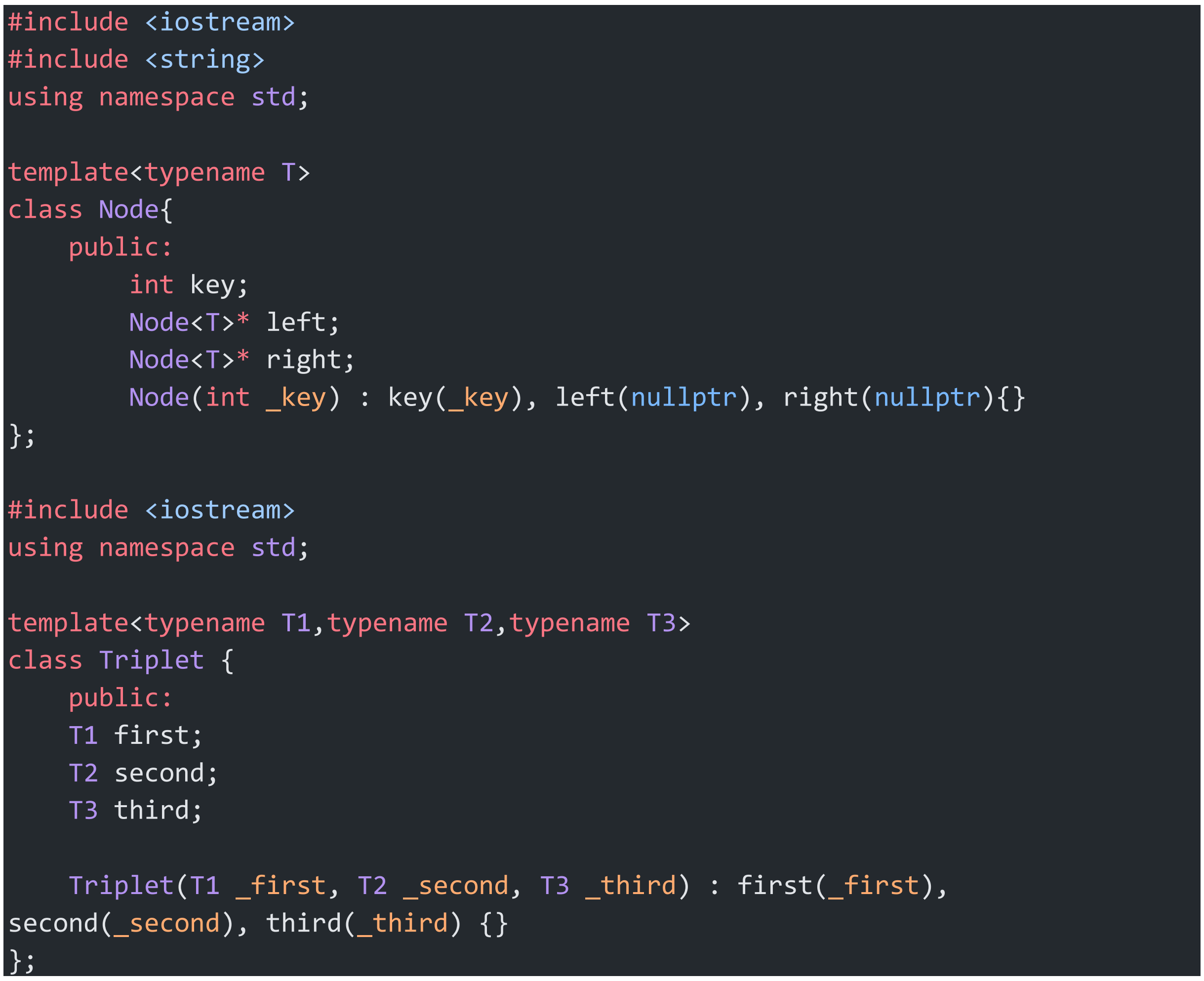
```cpp
#include<iostream>
#include "Queue.h"
using namespace std;

template<typename T>
class Node{
    public:
        T key;
        Node<T>* left;
        Node<T>* right;
        Node(T val):key(val),left(nullptr),right(nullptr){};
};

template<typename T>
class Binary_tree{
    private:
        Node<T>* root;
    public:
        Binary_tree():root(nullptr){}
        /*
            There are multiple ways to input in Binary tree,
            like levelOrder Preorder, like in BST. Since it is
            not mention I will do levelOrder insertion;
        */
       void insert_levelOrder(T arr[],int size){
            root=insert_levelOrder(arr,size,0);
       }
        Node<T>* insert_levelOrder(T arr[],int size,int index){
            if(index>=size) return nullptr;

            Node<T>* newNode=new Node<T>(arr[index]);

            newNode->left=insert_levelOrder(arr,size,2*index+1);
            newNode->right=insert_levelOrder(arr,size,2*index+2);
            return newNode;
        }
        void preOrder_traversal(){
            preOrder_traversal(root);
        }

        void preOrder_traversal(Node<T>* root){
            if(!root) return;

            cout<<root->key<<" ";
            preOrder_traversal(root->left);
            preOrder_traversal(root->right);
        }


        int height(Node<T>* root){
            if(!root) return 0;
            int left_h=height(root->left);
            int right_h=height(root->right);
            return (left_h>right_h)? left_h+1:right_h+1;
        }

        void print_current_level(Node<T>* root,int level){
            if(!root) return;
            if(level==0) cout<<root->key<<" ";
            if(level>0){
                print_current_level(root->left,level-1);
                print_current_level(root->right,level-1);
            }
        }

        void levelOrder_traversal(){
            levelOrder_traversal(root);
        }

        void levelOrder_traversal(Node<T>* root){
            int h=height(root);
            for(int i=0 ; i<h ; i++) {
                print_current_level(root,i);
                cout<<endl;
            }

        }

        bool is_full(){
            return is_full(root);
        }
        bool is_full(Node<T>* root){
            if(!root) return true;
```

```
            bool left=is_full(root->left);
            bool right=is_full(root->right);

            return ((!root->left && !root->right) || (root->left && root-
>right)) && left && right;
        }
        bool is_Complete() {
            return is_Complete(root);
        }
        bool is_Complete(Node<T>* root) {
            if (!root) return true;
            Queue<Node<T>*> q;
            q.enQueue(root);
            bool flag=false;

            while (!q.isEmpty()) {
                Node<T>* currentNode = q.deQueue();

                if(!currentNode){
                    flag=true;
                    continue;
                }
                if(flag) return false;
                if(currentNode->left) q.enQueue(currentNode->left);
                if(currentNode->right) q.enQueue(currentNode->right);
            }
            return true;
        }
        void convert(){
            convert(root);
        }

        void convert(Node<T>* root){
            if(!root) return;
            convert(root->left);
            convert(root->right);

            if(!(!root->left && !root->right || root->left && root->right)){
                if(root->left){
                    delete root->left;
                    root->left=nullptr;
                } else {
                    delete root->right;
                    root->right=nullptr;
                }
            }

        }
```

```cpp
};


int main(){

    Binary_tree<int> tree;
    //0 1 2 3 4 5 6 7
    //1 2 3 4 5 6 7 8
    int arr[] ={1, 2, 3, 4, 5, 6, 7, 8};
    int size=sizeof(arr)/sizeof(arr[0]);
    tree.insert_levelOrder(arr,size);

    cout<<"TREE:"<<endl;
    tree.levelOrder_traversal();
    cout<<endl;

    if(tree.is_Complete()) cout<<"TREE IS COMPLETE"<<endl;
    else cout<<"TREE IS NOT COMPLETE"<<endl;
    if(tree.is_full()) cout<<"TREE IS FULL"<<endl;
    else cout<<"TREE IS NOT FULL"<<endl;

    cout<<endl;
    tree.convert();
    cout<<"TREE AFTER CONVERTING"<<endl;
    tree.levelOrder_traversal();

    if(tree.is_Complete()) cout<<"TREE IS COMPLETE"<<endl;
    else cout<<"TREE IS NOT COMPLETE"<<endl;
    if(tree.is_full()) cout<<"TREE IS FULL"<<endl;
    else cout<<"TREE IS NOT FULL"<<endl;

    return 0;
}
```

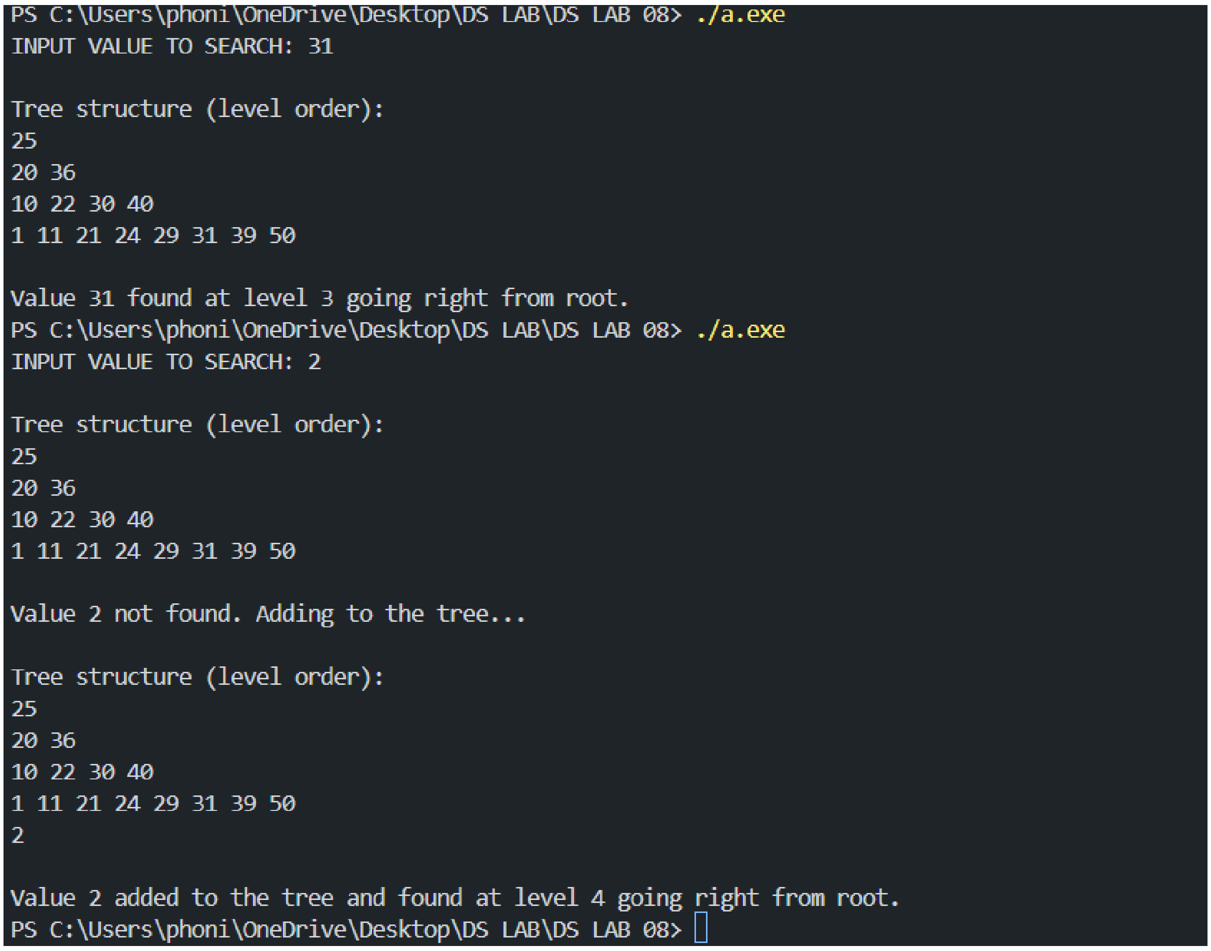

```
PS C:\Users\phoni\OneDrive\Desktop\DS LAB\DS LAB 08> ./a.exe
TREE:
1
2 3
4 5 6 7
8

TREE IS COMPLETE
TREE IS NOT FULL

TREE AFTER CONVERTING
1
2 3
4 5 6 7
TREE IS COMPLETE
TREE IS FULL
PS C:\Users\phoni\OneDrive\Desktop\DS LAB\DS LAB 08>
```

TASK 03


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

template<typename T>
class Node{
    public:
        int key;
        Node<T>* left;
        Node<T>* right;
        Node(int _key) : key(_key), left(nullptr), right(nullptr){}
};

#include <iostream>
using namespace std;

template<typename T1,typename T2,typename T3>
class Triplet {
    public:
    T1 first;
    T2 second;
    T3 third;

    Triplet(T1 _first, T2 _second, T3 _third) : first(_first),
second(_second), third(_third) {}
};
```

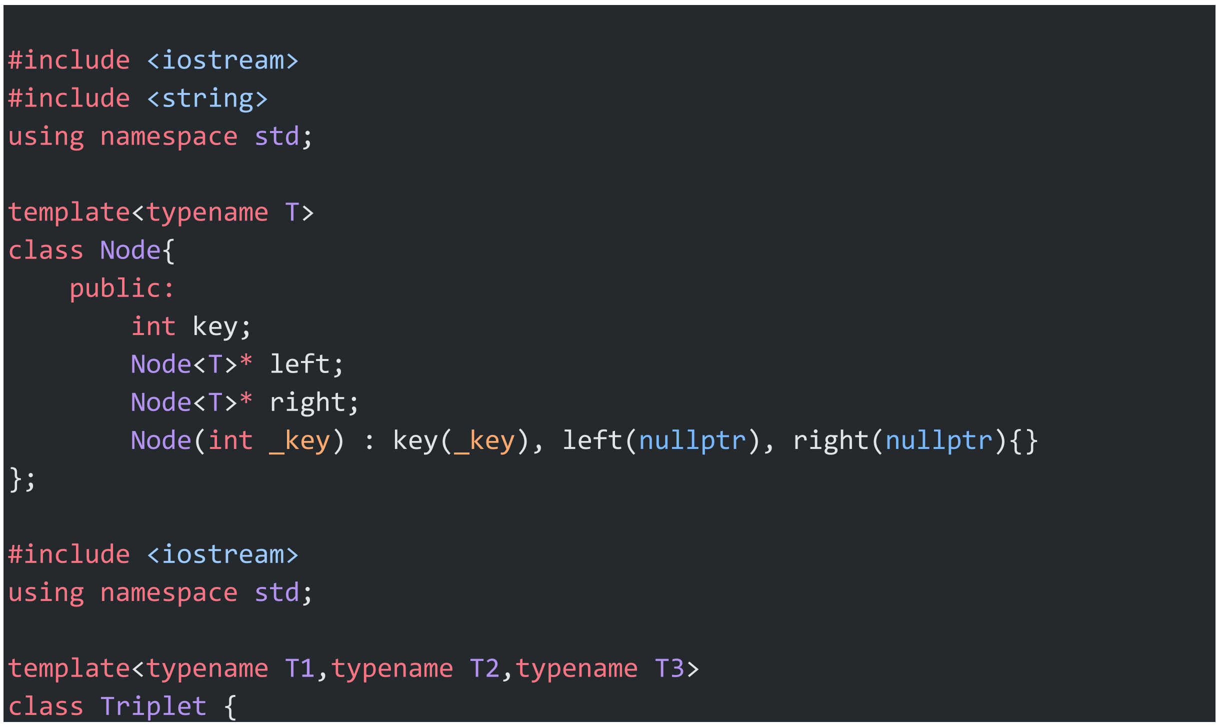
```
template<typename T>
class BinarySearchTree{
    private:
        Node<T>* root;
    public:
        BinarySearchTree() : root(nullptr){};

        Node<T>* getRoot(){
            return root;
        }

        void addNode(T key) {
            addNode(new Node<T>(key), root);
        }

        void addNode(Node<T>* newNode,Node<T>*& root) {
            if (!root){
                root=newNode;
                return;
            }
            if (newNode->key==root->key) return;

            if (newNode->key<root->key) addNode(newNode, root->left);
            else addNode(newNode, root->right);

            return;
        }

        void print_current_level(Node<T>* root,int level){
            if(!root) return;
            if(level==0) cout<<root->key<<" ";
            else if(level>0){
                print_current_level(root->left,level-1);
                print_current_level(root->right,level-1);
            }
        }
       int max(int i,int j,int k=0){
            int tmp=(j>k)? j:k;
            return (i>tmp)? i:tmp;
       }

        int height(Node<T>* root){
            if (!root) return -1;
                return max(height(root->left),height(root->right))+1;

        }
        void display_levelorder(){
            display_levelorder(root);
        }
        void display_levelorder(Node<T>* root){
            int h= height(root);
            for(int i=0 ; i<=h ; i++){
                print_current_level(root,i);
                cout<<endl;
            }
        }

        Triplet<bool,int,string> search(T val){
            return search(root,val,0,"");
        }

        Triplet<bool,int,string> search(Node<T>* root,T val,int 
rootlevel,string child=""){
            if(!root) return Triplet<bool,int,string>(false,-1,"");

            Triplet<bool,int,string> left=search(root-
>left,val,rootlevel+1,"left");
            Triplet<bool,int,string> right=search(root-
>right,val,rootlevel+1,"right");

            bool check=(root->key==val) || left.first || right.first;
            int level=(root->key==val)? rootlevel:(left.first)? 
left.second:(right.first)? right.second:-1;
            string currentChild=(root->key==val)? child:(left.first)? 
left.third:(right.first)? right.third:"";

            return Triplet<bool,int,string>(check,level,currentChild);

        }
};




int main() {
    BinarySearchTree<int> tree;
    tree.addNode(25);
    tree.addNode(20);
    tree.addNode(36);
    tree.addNode(10);
    tree.addNode(22);
    tree.addNode(30);
    tree.addNode(40);
    tree.addNode(1);
```

```cpp
    tree.addNode(11);
    tree.addNode(21);
    tree.addNode(24);
    tree.addNode(50);
    tree.addNode(31);
    tree.addNode(29);
    tree.addNode(39);

    int searchValue;
    cout<<"INPUT VALUE TO SEARCH: ";
    cin>>searchValue;

    Triplet<bool, int, string> result = tree.search(searchValue);

    cout<<endl<<"Tree structure (level order):"<<endl;
    tree.display_levelorder();

    if (result.first) {
        cout<<"Value "<<searchValue<<" found at level "<<result.second<<"
going "<<result.third<<" from root."<<endl;
    }else{
        cout<<"Value "<<searchValue<<" not found. Adding to the
tree..."<<endl;
        tree.addNode(searchValue);
        cout<<endl<<"Tree structure (level order):"<<endl;
        tree.display_levelorder();

        result=tree.search(searchValue);
        cout<<"Value "<<searchValue<<" added to the tree and found at level
"<<result.second<<" going "<<result.third<<" from root."<<endl;
    }


    return 0;
}
```



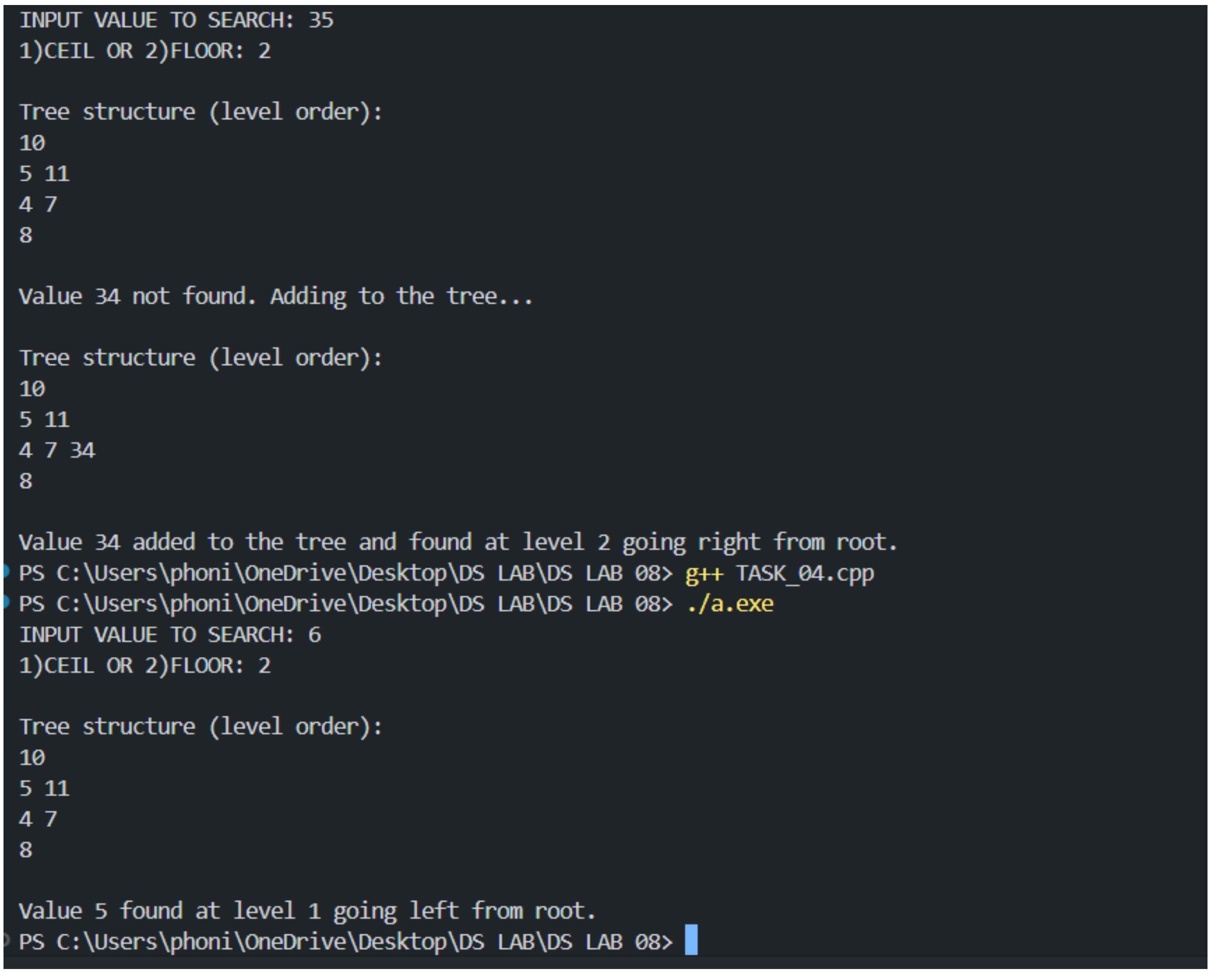
```
PS C:\Users\phoni\OneDrive\Desktop\DS LAB\DS LAB 08> ./a.exe
INPUT VALUE TO SEARCH: 31

Tree structure (level order):
25
20 36
10 22 30 40
1 11 21 24 29 31 39 50

Value 31 found at level 3 going right from root.
PS C:\Users\phoni\OneDrive\Desktop\DS LAB\DS LAB 08> ./a.exe
INPUT VALUE TO SEARCH: 2

Tree structure (level order):
25
20 36
10 22 30 40
1 11 21 24 29 31 39 50

Value 2 not found. Adding to the tree...

Tree structure (level order):
25
20 36
10 22 30 40
1 11 21 24 29 31 39 50
2

Value 2 added to the tree and found at level 4 going right from root.
PS C:\Users\phoni\OneDrive\Desktop\DS LAB\DS LAB 08>
```

TASK 04



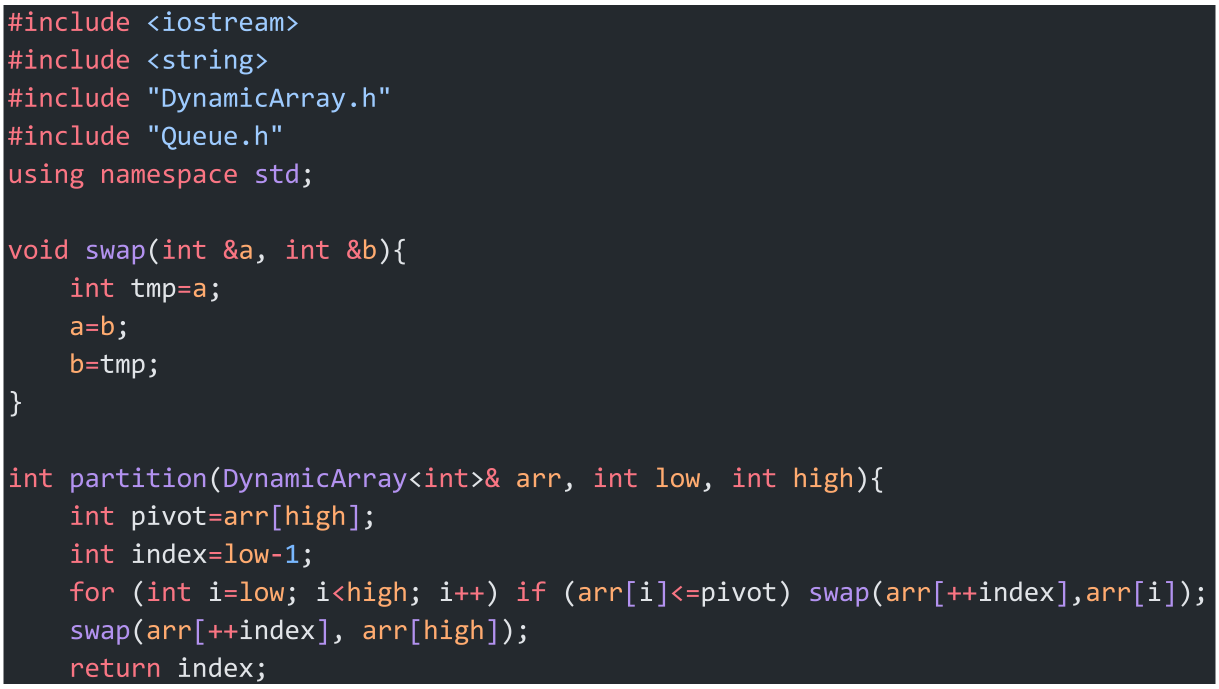
```cpp
#include <iostream>
#include <string>
using namespace std;

template<typename T>
class Node{
    public:
        int key;
        Node<T>* left;
        Node<T>* right;
        Node(int _key) : key(_key), left(nullptr), right(nullptr){}
};

#include <iostream>
using namespace std;

template<typename T1,typename T2,typename T3>
class Triplet {
    public:
    T1 first;
    T2 second;
    T3 third;

    Triplet(T1 _first, T2 _second, T3 _third) : first(_first),
second(_second), third(_third) {}
};


template<typename T>
class BinarySearchTree{
    private:
        Node<T>* root;
    public:
        BinarySearchTree() : root(nullptr){};

        Node<T>* getRoot(){
            return root;
        }

        void addNode(T key) {
            addNode(new Node<T>(key), root);
        }

        void addNode(Node<T>* newNode,Node<T>*& root) {
            if (!root){
                root=newNode;
                return;
            }
            if (newNode->key==root->key) return;

            if (newNode->key<root->key) addNode(newNode, root->left);
            else addNode(newNode, root->right);

            return;
        }

        void print_current_level(Node<T>* root,int level){
            if(!root) return;
            if(level==0) cout<<root->key<<" ";
            else if(level>0){
                print_current_level(root->left,level-1);
                print_current_level(root->right,level-1);
            }
        }
       int max(int i,int j,int k=0){
            int tmp=(j>k)? j:k;
            return (i>tmp)? i:tmp;
       }

        int height(Node<T>* root){
            if (!root) return -1;
                return max(height(root->left),height(root->right))+1;

        }

        void display_levelorder(){
            display_levelorder(root);
        }
        void display_levelorder(Node<T>* root){
            int h= height(root);
            for(int i=0 ; i<=h ; i++){
                print_current_level(root,i);
                cout<<endl;
            }
        }

        Triplet<bool,int,string> search(T val){
            return search(root,val,0,"");
        }

        Triplet<bool,int,string> search(Node<T>* root,T val,int
rootlevel,string child=""){
            if(!root) return Triplet<bool,int,string>(false,-1,"");

            Triplet<bool,int,string> left=search(root-
>left,val,rootlevel+1,"left");
            Triplet<bool,int,string> right=search(root-
>right,val,rootlevel+1,"right");

            bool check=(root->key==val) || left.first || right.first;
            int level=(root->key==val)? rootlevel:(left.first)?
left.second:(right.first)? right.second:-1;
            string currentChild=(root->key==val)? child:(left.first)?
left.third:(right.first)? right.third:"";

            return Triplet<bool,int,string>(check,level,currentChild);

        }
};


int ceil(int num){
    return num+1;
}
int floor(int num){
    return num-1;
```

```cpp
}


int main() {
    BinarySearchTree<int> tree;
    tree.addNode(10);
    tree.addNode(5);
    tree.addNode(11);
    tree.addNode(4);
    tree.addNode(7);
    tree.addNode(8);



    int searchValue;
    int choice;
    cout<<"INPUT VALUE TO SEARCH: ";
    cin>>searchValue;
    cout<<"1)CEIL OR 2)FLOOR: ";
    cin>>choice;
    if(choice==1){
        searchValue=ceil(searchValue);
    } else {
        searchValue=floor(searchValue);
    }
    Triplet<bool, int, string> result = tree.search(searchValue);

    cout<<endl<<"Tree structure (level order):"<<endl;
    tree.display_levelorder();

    if (result.first) {
        cout<<"Value "<<searchValue<<" found at level "<<result.second<<"
going "<<result.third<<" from root."<<endl;
    }else{
        cout<<"Value "<<searchValue<<" not found. Adding to the
tree..."<<endl;
        tree.addNode(searchValue);
        cout<<endl<<"Tree structure (level order):"<<endl;
        tree.display_levelorder();

        result=tree.search(searchValue);
        cout<<"Value "<<searchValue<<" added to the tree and found at level
"<<result.second<<" going "<<result.third<<" from root."<<endl;
    }


    return 0;
}
```



```
INPUT VALUE TO SEARCH: 35
1)CEIL OR 2)FLOOR: 2

Tree structure (level order):
10
5 11
4 7
8

Value 34 not found. Adding to the tree...

Tree structure (level order):
10
5 11
4 7 34
8

Value 34 added to the tree and found at level 2 going right from root.
PS C:\Users\phoni\OneDrive\Desktop\DS LAB\DS LAB 08> g++ TASK_04.cpp
PS C:\Users\phoni\OneDrive\Desktop\DS LAB\DS LAB 08> ./a.exe
INPUT VALUE TO SEARCH: 6
1)CEIL OR 2)FLOOR: 2

Tree structure (level order):
10
5 11
4 7
8

Value 5 found at level 1 going left from root.
PS C:\Users\phoni\OneDrive\Desktop\DS LAB\DS LAB 08>
```

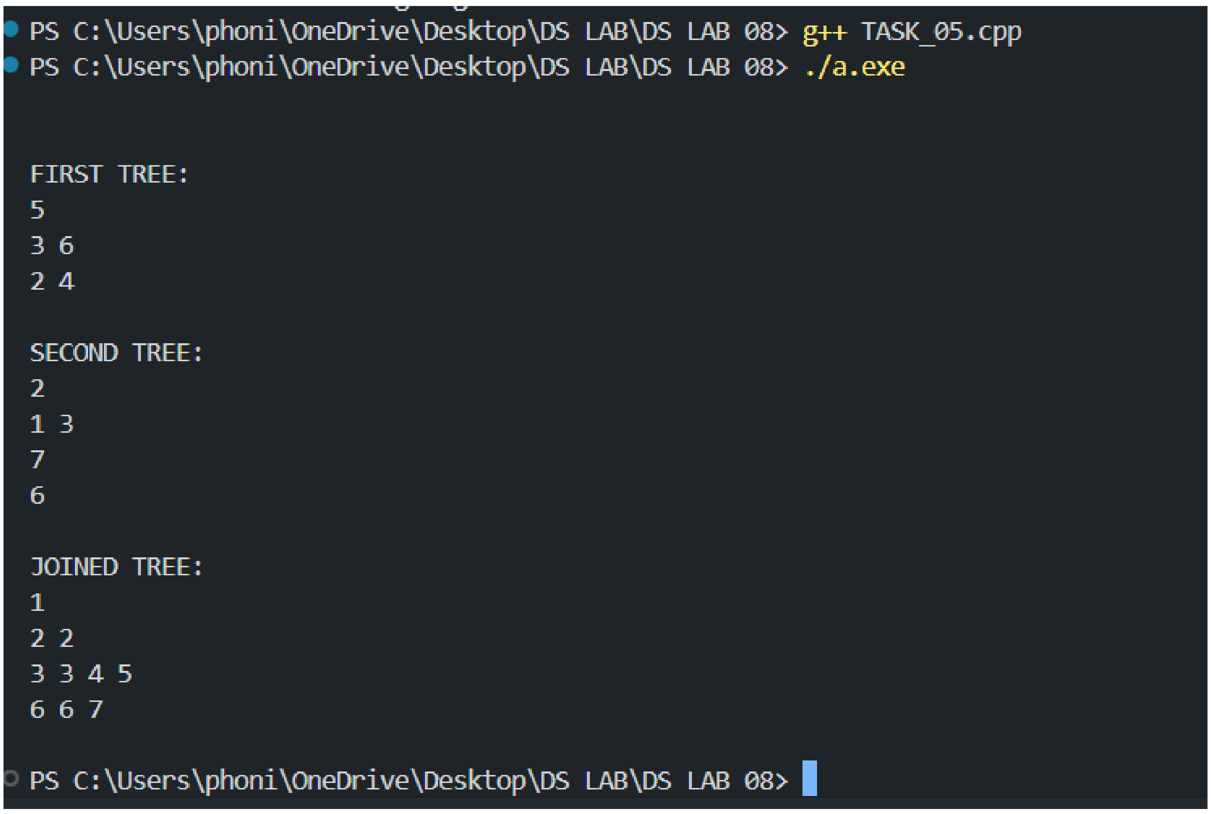
TASK 05



```cpp
#include <iostream>
#include <string>
#include "DynamicArray.h"
#include "Queue.h"
using namespace std;

void swap(int &a, int &b){
    int tmp=a;
    a=b;
    b=tmp;
}

int partition(DynamicArray<int>& arr, int low, int high){
    int pivot=arr[high];
    int index=low-1;
    for (int i=low; i<high; i++) if (arr[i]<=pivot) swap(arr[++index],arr[i]);
    swap(arr[++index], arr[high]);
    return index;
```

```
}

void quickSort(DynamicArray<int>& arr, int low, int high){
    if(low>=high) return;
    int pi=partition(arr, low, high);
    quickSort(arr, low, pi-1);
    quickSort(arr, pi+1, high);
}
template<typename T>
class Node {
public:
    int key;
    Node* left;
    Node* right;

    Node(int _key) : key(_key), left(nullptr), right(nullptr) {}
};

template<typename T>
class BinarySearchTree {
private:
    Node<T>* root;

    void deleteTree(Node<T>* node) {
        if (node) {
            deleteTree(node->left);
            deleteTree(node->right);
            delete node;
        }
    }

public:
    BinarySearchTree() : root(nullptr) {}

    ~BinarySearchTree() {
        deleteTree(root);
    }

    Node<T>* getRoot() {
        return root;
    }

    void insert_levelOrder(DynamicArray<T> arr,int size){
        root=insert_levelOrder(arr,size,0);
    }
    Node<T>* insert_levelOrder(DynamicArray<T> arr,int size,int index){
        if(index>=size) return nullptr;
```

```cpp
        Node<T>* newNode=new Node<T>(arr[index]);

        newNode->left=insert_levelOrder(arr,size,2*index+1);
        newNode->right=insert_levelOrder(arr,size,2*index+2);
        return newNode;
    }


    void addNode(T key) {
        addNode(new Node<T>(key), root);
    }

    void addNode(Node<T>* newNode, Node<T>*& root) {
        if (!root) {
            root = newNode;
            return;
        }
        if (newNode->key == root->key) return;

        if (newNode->key < root->key) addNode(newNode, root->left);
        else addNode(newNode, root->right);
    }

    void print_current_level(Node<T>* root, int level) {
        if (!root) return;
        if (level == 0) cout << root->key << " ";
        else if (level > 0) {
            print_current_level(root->left, level - 1);
            print_current_level(root->right, level - 1);
        }
    }

    void display_levelorder() {
        display_levelorder(root);
    }

    void display_levelorder(Node<T>* root) {
        int h = height(root);
        for (int i = 0; i <= h; i++) {
            print_current_level(root, i);
            cout << endl;
        }
    }

    int max(int i, int j, int k = 0) {
        int tmp = (j > k) ? j : k;
        return (i > tmp) ? i : tmp;
    }
    int height(Node<T>* root) {
        if (!root) return -1;
        return max(height(root->left), height(root->right)) + 1;
    }

    DynamicArray<int> read_levelOrder() {
        return read_levelOrder(root);
    }

    DynamicArray<int> read_levelOrder(Node<int>* root) {
        if (!root) return DynamicArray<int>();

        Queue<Node<int>*> q;
        q.enQueue(root);

        DynamicArray<int> nodes;
        while (!q.isEmpty()) {
            Node<int>* tmp = q.deQueue();
            if(tmp) nodes.push_back(tmp->key);
            if (tmp->left) q.enQueue(tmp->left);
            if (tmp->right) q.enQueue(tmp->right);
        }
    cout<<endl;
        return nodes;
    }

};

    DynamicArray<int> joinTrees(DynamicArray<int>& arr1,DynamicArray<int>&
arr2) {
        DynamicArray<int> joinedNodes;
        for (int i = 0; i < arr1.size(); ++i) {
            joinedNodes.push_back(arr1[i]);
        }
        for (int i = 0; i < arr2.size(); ++i) {
            joinedNodes.push_back(arr2[i]);
        }
        quickSort(joinedNodes,0,joinedNodes.size()-1);
        return joinedNodes;
    }

int main() {
    BinarySearchTree<int> BST01;
    BST01.addNode(5);
    BST01.addNode(3);
    BST01.addNode(6);
    BST01.addNode(2);
    BST01.addNode(4);
    BinarySearchTree<int> BST02;
    BST02.addNode(2);
    BST02.addNode(1);
    BST02.addNode(3);
    BST02.addNode(7);
    BST02.addNode(6);

    DynamicArray<int> firstTreeNodes=BST01.read_levelOrder();
    DynamicArray<int> secondTreeNodes=BST02.read_levelOrder();
    DynamicArray<int> joinedTreeNodes = joinTrees(firstTreeNodes,
secondTreeNodes);

    cout<<"FIRST TREE: "<<endl;
    BST01.display_levelorder();
    cout<<"SECOND TREE: "<<endl;
    BST02.display_levelorder();

    BinarySearchTree<int> joinedBST;
    joinedBST.insert_levelOrder(joinedTreeNodes,joinedTreeNodes.size());

    cout<<"JOINED TREE: "<<endl;
    joinedBST.display_levelorder();
    return 0;
}
```



```
PS C:\Users\phoni\OneDrive\Desktop\DS LAB\DS LAB 08> g++ TASK_05.cpp
PS C:\Users\phoni\OneDrive\Desktop\DS LAB\DS LAB 08> ./a.exe


FIRST TREE:
5
3 6
2 4

SECOND TREE:
2
1 3
7
6

JOINED TREE:
1
2 2
3 3 4 5
6 6 7

PS C:\Users\phoni\OneDrive\Desktop\DS LAB\DS LAB 08>
```