```
//  Yousef Zoumot
//  main.cpp
//  Coen70HW6.1 *Chapter 10 Problem #2
//
//  Created by Yousef Zoumot on 3/6/16.
//  Copyright (c) 2016 Yousef Zoumot. All rights reserved.
//

#include <iostream>
#include <math.h>
using namespace std;


template <class T>
struct Node {
    T value;
    Node *left;
    Node *right;

    Node(T val) {
        this->value = val;
    }

    Node(T val, Node<T> left, Node<T> right) {
        this->value = val;
        this->left = left;
        this->right = right;
    }
};
/////**********************************************************************
***/////
template <class T>
class BinaryTree {

private:
    Node<T> *root;
    void addRecursive(Node<T> *root, T val);
    void printRecursive(Node<T> *root);
    int nodesCountRecursive(Node<T> *root);
    int heightRecursive(Node<T> *root);
    bool deleteValueRecursive(Node<T>* parent, Node<T>* current,
T value);

public:
    void add(T val);
    void print();
    int nodesCount();
    int height();
    bool deleteValue(T value);
};
```

```
/////**********************************************************
*****/////
template <class T>
void BinaryTree<T>:: addRecursive(Node<T> *root, T val) {
    if (root->value > val) {
        if (!root->left) {
            root->left = new Node<T>(val);
        } else {
            addRecursive(root->left, val);
        }
    } else {
        if (!root->right) {
            root->right = new Node<T>(val);
        } else {
            addRecursive(root->right, val);
        }
    }
}
/////**********************************************************
*****/////
template <class T>
void BinaryTree<T>:: printRecursive(Node<T> *root) {
    if (!root) return;
    printRecursive(root->left);
    cout<<root->value<<' ';
    printRecursive(root->right);
}
/////**********************************************************
*****/////
template <class T>
int BinaryTree<T>:: nodesCountRecursive(Node<T> *root) {
    if (!root) return 0;
    else return 1 + nodesCountRecursive(root->left) +
nodesCountRecursive(root->right);
}
/////**********************************************************
*****/////
template <class T>
int BinaryTree<T>:: heightRecursive(Node<T> *root) {
    if (!root) return 0;
    else return 1 + max(heightRecursive(root->left),
heightRecursive(root->right));
}
/////**********************************************************
*****/////
template <class T>
bool BinaryTree<T>:: deleteValueRecursive(Node<T>* parent,
Node<T>* current, T value) {
    if (!current) return false;
```

```
    if (current->value == value) {
        if (current->left == NULL || current->right == NULL) {
            Node<T>* temp = current->left;
            if (current->right) temp = current->right;
            if (parent) {
                if (parent->left == current) {
                        parent->left = temp;
                } else {
                        parent->right = temp;
                }
            } else {
                    this->root = temp;
            }
        } else {
            Node<T>* substitute = current->right;
            while (substitute->left) {
                    substitute = substitute->left;
            }
            T temp = current->value;
            current->value = substitute->value;
            substitute->value = temp;
            return deleteValueRecursive(current, current->right,
temp);
        }
        delete current;
        return true;
    }
    return deleteValueRecursive(current, current->left, value) ||
    deleteValueRecursive(current, current->right, value);
}
/////**********************************************************
***/////
template <class T>
void BinaryTree<T>:: add(T val) {
    if (root) {
        this->addRecursive(root, val);
    } else {
        root = new Node<T>(val);
    }
}
/////**********************************************************
***/////
template <class T>
void BinaryTree<T>:: print() {
    printRecursive(this->root);
    cout<<"\n";
}
/////**********************************************************
***/////
template <class T>
```

```
int BinaryTree<T>:: nodesCount() {
    return nodesCountRecursive(root);
}
/////**********************************************************************
***/////
template <class T>
int BinaryTree<T>:: height() {
    return heightRecursive(this->root);
}
/////**********************************************************************
***/////
template <class T>
bool BinaryTree<T>:: deleteValue(T value) {
    return this->deleteValueRecursive(NULL, this->root, value);
}
/////**********************************************************************
***/////


int main(int argc, const char * argv[]) {
    // insert code here...
    BinaryTree<int> *bst1=new BinaryTree<int>();
    bst1->add(5);
    bst1->add(4);
    bst1->add(7);
    bst1->add(2);
    bst1->add(9);
    bst1->add(8);
    bst1->print();
    bst1->deleteValue(5);
    bst1->print();
    return 0;
}
```

```
2 4 5 7 8 9
2 4 7 8 9
Program ended with exit code: 0
```

```
//  Yousef Zoumot
//  main.cpp
//  Coen70HW6.2  *Chapter 10 Problem #3
//
//  Created by Yousef Zoumot on 3/6/16.
//  Copyright (c) 2016 Yousef Zoumot. All rights reserved.
//


#include<iostream>
#include<cstdio>
#include<sstream>
#include<algorithm>
#define pow2(n) (1 << (n))
using namespace std;

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

class avlTree
{
public:
    avlTree()
    {
        root = NULL;
    }
    int height(){return heightRecursive(root);};
    void insert(int value){root=insertRecursive(root , value);};
    void remove(int value){root=removeRecursive(root, value);};
    void display();
    void inOrder();
    void preOrder();
    void postOrder();

private:
    avl_Node *root;
    int heightRecursive(avl_Node *);
    int heightDifferenceRecursive(avl_Node *);
    avl_Node *rightright_rotationRecursive(avl_Node *);
    avl_Node *leftleft_rotationRecursive(avl_Node *);
    avl_Node *leftright_rotationRecursive(avl_Node *);
    avl_Node *rightleft_rotationRecursive(avl_Node *);
    avl_Node* balanceRecursive(avl_Node *);
    avl_Node* insertRecursive(avl_Node *, int );
    avl_Node* removeRecursive(avl_Node *, int );
    void displayRecursive(avl_Node *, int);
```

```
    void inOrderRecursive(avl_Node *);
    void preOrderRecursive(avl_Node *);
    void postOrderRecursive(avl_Node *);
    avl_Node* minValueNode(avl_Node* node);
};

void avlTree:: display(){
    if(root ==NULL)
        cout<<"This AVL Tree is empty"<< "\n";
    else{
        displayRecursive(root, 1);
    }
    cout<<"\n";
    cout<<"\n";
    cout<<"\n";
    cout<<"\n";
    cout<<"\n";
    cout<<"\n";

}

void avlTree:: preOrder(){
    preOrderRecursive(root);
}

void avlTree:: inOrder(){
    inOrderRecursive(root);
}

void avlTree:: postOrder(){
    postOrderRecursive(root);
}

avl_Node* avlTree::  minValueNode(avl_Node* node)
{
    avl_Node* current = node;

    /* loop down to find the leftmost leaf */
    while (current->left != NULL)
        current = current->left;

    return current;
}

//* Height of AVL Tree

int avlTree::heightRecursive(avl_Node *temp)
{
    int h = 0;
    if (temp != NULL)
```

```
    {
        int l_height = heightRecursive (temp->left);
        int r_height = heightRecursive (temp->right);
        int max_height = max (l_height, r_height);
        h = max_height + 1;
    }
    return h;
}


// * Height Difference

int avlTree::heightDifferenceRecursive(avl_Node *temp)
{
    int l_height = heightRecursive (temp->left);
    int r_height = heightRecursive (temp->right);
    int b_factor= l_height - r_height;
    return b_factor;
}


//  Right- Right rotationRecursive

avl_Node *avlTree::rightright_rotationRecursive(avl_Node *parent)
{
    avl_Node *temp;
    temp = parent->right;
    parent->right = temp->left;
    temp->left = parent;
    return temp;
}

// Left- Left rotationRecursive

avl_Node *avlTree::leftleft_rotationRecursive(avl_Node *parent)
{
    avl_Node *temp;
    temp = parent->left;
    parent->left = temp->right;
    temp->right = parent;
    return temp;
}


//  Left - Right rotationRecursive

avl_Node *avlTree::leftright_rotationRecursive(avl_Node *parent)
{
    avl_Node *temp;
    temp = parent->left;
```

```
    parent->left = rightright_rotationRecursive (temp);
    return leftleft_rotationRecursive (parent);
}


//  Right- Left rotationRecursive

avl_Node *avlTree::rightleft_rotationRecursive(avl_Node *parent)
{
    avl_Node *temp;
    temp = parent->right;
    parent->right = leftleft_rotationRecursive (temp);
    return rightright_rotationRecursive (parent);
}


//  Balancing AVL Tree

avl_Node *avlTree::balanceRecursive(avl_Node *temp)
{
    int bal_factor = heightDifferenceRecursive (temp);
    if (bal_factor > 1)
    {
        if (heightDifferenceRecursive (temp->left) > 0)
            temp = leftleft_rotationRecursive (temp);
        else
            temp = leftright_rotationRecursive (temp);
    }
    else if (bal_factor < -1)
    {
        if (heightDifferenceRecursive (temp->right) > 0)
            temp = rightleft_rotationRecursive (temp);
        else
            temp = rightright_rotationRecursive (temp);
    }
    return temp;
}


//insertRecursive Element into the tree

avl_Node *avlTree::insertRecursive(avl_Node *root, int value)
{
    if (root == NULL)
    {
        root = new avl_Node;
        root->data = value;
        root->left = NULL;
        root->right = NULL;
        return root;
```

```
    }
    else if (value < root->data)
    {
        root->left = insertRecursive(root->left, value);
        root = balanceRecursive (root);
    }
    else if (value >= root->data)
    {
        root->right = insertRecursive(root->right, value);
        root = balanceRecursive (root);
    }
    return root;
}

//removes element from tree

avl_Node *avlTree:: removeRecursive(avl_Node *root, int key)
{


        // PERFORM STANDARD BST DELETE

        if (root == NULL)
            return root;

        // If the key to be deleted is smaller than the root's
key,
        // then it lies in left subtree
        if ( key < root->data )
            root->left = removeRecursive(root->left, key);

        // If the key to be deleted is greater than the root's
key,
        // then it lies in right subtree
        else if( key > root->data )
            root->right = removeRecursive(root->right, key);

        // if key is same as root's key, then This is the node
        // to be deleted
        else
        {
            // node with only one child or no child
            if( (root->left == NULL) || (root->right == NULL) )
            {
                avl_Node* temp = root->left ? root->left : root-
>right;

                // No child case
                if(temp == NULL)
                {
```

```
                    temp = root;
                    root = NULL;
                }
                else // One child case
                    *root = *temp; // Copy the contents of the
non-empty child

                delete temp;
            }
            else
            {
                // node with two children: Get the inorder
successor (smallest
                // in the right subtree)
                avl_Node* temp = minValueNode(root->right);

                // Copy the inorder successor's data to this node
                root->data = temp->data;

                // Delete the inorder successor
                root->right = removeRecursive(root->right, temp-
>data);
            }
        }

        // If the tree had only one node
        if (root == NULL)
            return root;

        // GET THE BALANCE FACTOR OF THIS NODE (to check whether
        //  this node became unbalanced)
        int balance = heightDifferenceRecursive(root);

        // If unbalanced, there are 4 cases

        // Left Left Case
        if (balance > 1 && heightDifferenceRecursive(root->left)
>= 0)
            return leftleft_rotationRecursive(root);

        // Left Right Case
        if (balance > 1 && heightDifferenceRecursive(root->left)
< 0)
        {
            return leftright_rotationRecursive(root);
        }

        // Right Right Case
        if (balance < -1 && heightDifferenceRecursive(root-
>right) <= 0)
```

```
            return rightright_rotationRecursive(root);

        // Right Left Case
        if (balance < -1 && heightDifferenceRecursive(root-
>right) > 0)
        {
            return rightleft_rotationRecursive(root);
        }

        return root;

}

//displayRecursive AVL Tree

void avlTree::displayRecursive(avl_Node *ptr, int level)
{
    int i;
    if (ptr!=NULL)
    {
        displayRecursive(ptr->right, level + 1);
        printf("\n");
        if (ptr == root)
            cout<<"Root -> ";
        for (i = 0; i < level && ptr != root; i++)
            cout<<"        ";
        cout<<ptr->data;
        displayRecursive(ptr->left, level + 1);
    }
}


//inOrderRecursive Traversal of AVL Tree

void avlTree::inOrderRecursive(avl_Node *tree)
{
    if (tree == NULL)
        return;
    inOrderRecursive (tree->left);
    cout<<tree->data<<"  ";
    inOrderRecursive (tree->right);
}

//  preOrderRecursive Traversal of AVL Tree

void avlTree::preOrderRecursive(avl_Node *tree)
{
    if (tree == NULL)
        return;
    cout<<tree->data<<"  ";
```

```
    preOrderRecursive (tree->left);
    preOrderRecursive (tree->right);

}


// postOrderRecursive Traversal of AVL Tree

void avlTree::postOrderRecursive(avl_Node *tree)
{
    if (tree == NULL)
        return;
    postOrderRecursive ( tree ->left );
    postOrderRecursive ( tree ->right );
    cout<<tree->data<<"  ";
}

int main(int argc, const char * argv[]) {
    avlTree avlt1 = avlTree();
    avlt1.insert(5);
    avlt1.insert(4);
    avlt1.insert(3);
    avlt1.insert(2);
    avlt1.insert(1);
    avlt1.display();
    avlt1.remove(4);
    avlt1.display();
}
```

```
                    5
Root -> 4
                        3
                    2
                        1




                    5
                        3
Root -> 2
                    1



Program ended with exit code: 0
```

```
//  Yousef Zoumot
//  main.cpp
//  Coen70HW6.3 * Chapter 10 Problem 4
//
//  Created by Yousef Zoumot on 3/6/16.
//  Copyright (c) 2016 Yousef Zoumot. All rights reserved.
//

#include <iostream>
#include <math.h>
#include <vector>
#include <utility>

#define p(x) (((x)-1)/2)  //returns parent location
#define l(x) ((x)*2+1)  //returns left child location
#define r(x) ((x)*2+2)  //returns right child location


using namespace std;

template <class T>
struct Node {
    T value;
    Node *left;
    Node *right;

    Node(T val) {
        this->value = val;
    }

    Node(T val, Node<T> left, Node<T> right) {
        this->value = val;
        this->left = left;
        this->right = right;
    }
};

/////**********************************************************************
***/////
template <class T>
class BinaryTree {

private:
    Node<T> *root;
    int count;
    void addRecursive(Node<T> *root, T val);
    void printRecursive(Node<T> *root);
    int nodesCountRecursive(Node<T> *root);
    int heightRecursive(Node<T> *root);
    bool deleteValueRecursive(Node<T>* parent, Node<T>* current,
```

```
T value);

public:
    vector<bool > is_present;
    void updatePresent1();
    void updatePresent2(Node<T> *tree, int);
    void is_present_user(int i);
    int size(){return count;};
    void add(T val);
    void print();
    int nodesCount();
    int height();
    bool deleteValue(T value);
};

template<class T>
void BinaryTree<T>:: updatePresent1(){
    is_present.resize(pow(2, height())-1);
    updatePresent2(root, 0);
}

template<class T>
void BinaryTree<T>:: updatePresent2(Node<T> *tree, int index){
    if (tree == NULL){
        is_present[index]=false;
            return;
    }
    else{
        is_present[index]=true;
    }

        index=l(index);
        updatePresent2(tree->left, index);
        index=r(index);
        updatePresent2(tree->right, index);
}

template <class T>
void BinaryTree<T>:: is_present_user(int i){
    updatePresent1();
    if(is_present[i]){
        cout<<"True";
    }
    else{
        cout<<"False";
    }
    cout<<"\n";
}

/////*****************************************************************
```

```
***/////
template <class T>
void BinaryTree<T>:: addRecursive(Node<T> *root, T val) {
    if (root->value > val) {
        if (!root->left) {
            root->left = new Node<T>(val);
        } else {
            addRecursive(root->left, val);
        }
    } else {
        if (!root->right) {
            root->right = new Node<T>(val);
        } else {
            addRecursive(root->right, val);
        }
    }
}
/////*****************************************************************
***/////
template <class T>
void BinaryTree<T>:: printRecursive(Node<T> *root) {
    if (!root) return;
    printRecursive(root->left);
    cout<<root->value<<' ';
    printRecursive(root->right);
}
/////*****************************************************************
***/////
template <class T>
int BinaryTree<T>:: nodesCountRecursive(Node<T> *root) {
    if (!root) return 0;
    else return 1 + nodesCountRecursive(root->left) +
nodesCountRecursive(root->right);
}
/////*****************************************************************
***/////
template <class T>
int BinaryTree<T>:: heightRecursive(Node<T> *root) {
    if (!root) return 0;
    else return 1 + max(heightRecursive(root->left),
heightRecursive(root->right));
}
/////*****************************************************************
***/////
template <class T>
bool BinaryTree<T>:: deleteValueRecursive(Node<T>* parent,
Node<T>* current, T value) {
    if (!current) return false;
    if (current->value == value) {
        if (current->left == NULL || current->right == NULL) {
```

```
            Node<T>* temp = current->left;
            if (current->right) temp = current->right;
            if (parent) {
                if (parent->left == current) {
                    parent->left = temp;
                } else {
                    parent->right = temp;
                }
            } else {
                this->root = temp;
            }
        } else {
            Node<T>* substitute = current->right;
            while (substitute->left) {
                substitute = substitute->left;
            }
            T temp = current->value;
            current->value = substitute->value;
            substitute->value = temp;
            return deleteValueRecursive(current, current->right,
temp);
        }
        delete current;
        return true;
    }
    return deleteValueRecursive(current, current->left, value) ||
    deleteValueRecursive(current, current->right, value);
}
/////**********************************************************************
***//////
template <class T>
void BinaryTree<T>:: add(T val) {
    if (root) {
        this->addRecursive(root, val);
    } else {
        root = new Node<T>(val);
    }
}
/////**********************************************************************
***//////
template <class T>
void BinaryTree<T>:: print() {
    printRecursive(this->root);
    cout<<"\n";
}
/////**********************************************************************
***//////
template <class T>
int BinaryTree<T>:: nodesCount() {
    return nodesCountRecursive(root);
```

```
}
/////**********************************************************************
**//////
template <class T>
int BinaryTree<T>:: height() {
    return heightRecursive(this->root);
}
/////**********************************************************************
**//////
template <class T>
bool BinaryTree<T>:: deleteValue(T value) {
    return this->deleteValueRecursive(NULL, this->root, value);
}
/////**********************************************************************
**//////


int main(int argc, const char * argv[]) {
    // insert code here...
    BinaryTree<int> *bst1=new BinaryTree<int>();
    bst1->add(5);
    bst1->add(4);
    bst1->add(7);
    bst1->add(2);
    bst1->add(9);
    bst1->add(8);
    bst1->print();
    bst1->deleteValue(5);
    bst1->print();
    bst1->is_present_user(0);
    bst1->is_present_user(1);
    bst1->is_present_user(7);
    return 0;
}
```

```
2 4 5 7 8 9
2 4 7 8 9
True
True
False
Program ended with exit code: 0
```