```
#include <stack>
#include <queue>
#include <iostream>

template<typename T>
struct AVLNode {
        T Value;
        int Height = 1; //the height of the subtree
        AVLNode* Left;
        AVLNode* Right;
        AVLNode* Parent;

        AVLNode() = default;

        AVLNode(T v, AVLNode* l, AVLNode* r, AVLNode* p)
        :Value(v), Left(l), Right(r), Parent(p){}

        AVLNode & operator=(const AVLNode & N){
                Value = N.Value;
                Right = N.Right;
                Left = N.Left;
                Parent = N.Parent;
                Height = N.Height;
                return *this;
        }

};

template<typename T>
struct AVLTree {
        AVLNode<T>* Head;

        AVLTree()
        :Head(nullptr){}

        /*
        average: O(n)
        worst case: O(n)
        again, we have to traverse the whole set
        */
         AVLTree(const AVLTree<T> & t){
                if (t.Head == nullptr){
                        Head = nullptr;
                        return;
                }

                Head = new AVLNode<T>(*t.Head);
                AVLNode<T>* r = Head;
                AVLNode<T>* n;

                std::stack<AVLNode<T>*> Stack;
                std::stack<AVLNode<T>*> twoChild;

                 if (t.Head->Left){
                        Stack.push(t.Head->Left);
                 }
                 if (t.Head->Right){
                        Stack.push(t.Head->Right);
                 }

                twoChild.push(Head);
                while(!Stack.empty()){

                        n = Stack.top();
                        //do visit
                        AVLNode<T>* newAVLNode = new AVLNode<T>(*n);
                        if (r->Value > n->Value){
                                r->Left = newAVLNode;
                        }else{
```



```
                                r->Right = newAVLNode;
                        }
                        newAVLNode->Parent = r;
                        r = newAVLNode;

                        if (n->Left && n->Right){
                                twoChild.push(r);
                        }else if (!n->Left && !n->Right){
                                //this resets r to the last node with two childr
en so that our traversial of the new tree moves in step with our pre-order trave
rsial
                                if(!twoChild.empty()){
                                        r = twoChild.top();
                                        twoChild.pop();
                                }
                        }


                        Stack.pop();
                        if (n->Left){
                                Stack.push(n->Left);
                        }
                        if (n->Right){
                                Stack.push(n->Right);
                        }
                }
        }

        /*
        average: O(k+n)
        worst case: O(k+n)
        again, we have to traverse the whole set of both the old and the new tre
s
        */
        AVLTree & operator=(const AVLTree & t){
                AVLNode<T>* p = Head;
                AVLNode<T>* ThisParent;
                while (Head) {

                        if (!p->Left && !p->Right){
                                ThisParent = p->Parent;
                                erase(p);
                                p = ThisParent;
                        } else if (!p->Left) {
                                p = p->Right;
                        } else {
                                p = p ->Left;
                        }
                }
                AVLTree newAVLTree = AVLTree(t);
                Head = newAVLTree->Head;
                return *this;

        }

        /*
        average: O(log(n))
        worst case: O(log(n))
        since the avl tree is always balanced we reduce the search time for all
cases
        */
        AVLNode<T>* find(T val) {
                AVLNode<T>* p = Head;
                while(p) {
                        if (val > p->Value) {
                                p = p->Right;
                        }else if (val < p->Value) {
                                p = p->Left;
                        }else{
```

```
                                return p;
                        }
                }
                return p;
        }


        /*
        average case: O(2log(n))
        worst case: O(2.5log(n))
        we have the same comlexity for both since the tree is always balanced.
        on average we have to work our way down to find the insertion point and
then work our way back up and rebalance.
        the actual rotation is a constant time operation, but in the worst case
we would have to rotate every other node which would be O(1/2log(n))
        */
        void insert(T val) {
                if (Head == nullptr){
                        AVLNode<T>* temp = new AVLNode<T>(val, nullptr, nullptr,
 nullptr);
                        Head = temp;
                        return;
                }
                AVLNode<T>* p = Head;
                AVLNode<T>* ThisParent;
                bool DidGoLeft;
                while(p) {
                        ThisParent = p;
                        if (val >=p->Value) {
                                p = p->Right;
                                DidGoLeft = false; //I'm not entirely happy with
 this
                        }else {
                                p = p->Left;
                                DidGoLeft = true;
                        }
                }
                AVLNode<T>* newAVLNode =  new AVLNode<T>(val, nullptr, nullptr,
ThisParent);
                if (DidGoLeft){
                        ThisParent->Left = newAVLNode;
                }else{
                        ThisParent->Right = newAVLNode;
                }
                if(!newAVLNode){
                        //breakpoint
                }
                balance(newAVLNode);
        }



         /*
        average case: O(log)n))
        worst case: O(2log(n))
        the average case arises when we are deleting a node with zero or one chi
ldren
        the worst case is if we have a large set and we are deleting the head an
d have to traverse all the way to the bottom to get the successor,
        and then all the way back up to rebalance
        */
        void erase(AVLNode<T>* k){
                AVLNode<T>* p = k->Parent;
                AVLNode<T>* parent = k->Parent;
                if (k->Left && k->Right){
                        AVLNode<T>* r = successor(k);
                        k->Value = r->Value;
                        parent = r;
                        erase(r); //r could have a right child
```



```
                }else if(k->Left){
                        if (p->Left == k){
                                p->Left = k->Left;
                        }else{
                                p->Right = k->Left;
                        }
                        k->Left->Parent = p;
                        delete k;
                }else if(k->Right){
                        if (p->Left == k){
                                p->Left = k->Right;
                        }else{
                                p->Right = k->Right;
                        }
                        k->Right->Parent = p;
                        delete k;
                }else{
                        if (p->Left == k){
                                p->Left = nullptr;
                        }else{
                                p->Right = nullptr;
                       }
                        delete k;
                }
                balance(parent);
        }

        AVLNode<T>* successor(AVLNode<T>* k){
                AVLNode<T>* p = k->Right;
                AVLNode<T>* output;
                while (p){
                        output = p;
                        p = p->Left;
                }
                return output;
        }

        void print(){
                std::cout << "digraph G {" << std::endl;
                AVLNode<T>* n;
                std::stack<AVLNode<T>*> Stack;
                int counter = 0;
                int otherCounter = 0;

                Stack.push(Head);
                while(!Stack.empty()){
                        n = Stack.top();
                        //do visit
                        if (!n->Parent){
                                std::cout << "HEAD" << "->" << n->Value << ";" <
< std::endl;
                        }else{
                                std::cout << n->Parent->Value <<  "->" << n->Val
ue  << ";" << std::endl;
                        }
                        if (!n->Left) {
                                std::cout << n->Value << "->" << "Leftnull" << cou
nter << ";" << std::endl;

                        }
                        if (!n->Right) {
                                std::cout << n->Value <<"->" << "Righttnull" << cou
nter << ";" << std::endl;

                        }
                        counter++;

                        Stack.pop();
                        if (n->Right){
```

```
                                Stack.push(n->Right);
                        }
                        if (n->Left){
                                Stack.push(n->Left);
                        }
                }
                std::cout << "}";
        }

        //a convenience function because i'm that lazy
        void stitch(AVLNode<T>* parent, AVLNode<T>* child, bool IsLeft){
                if (IsLeft){
                        parent->Left = child;
                }else{
                        parent->Right = child;
                }
                if(child){
                        child->Parent = parent;
                }
        }

        //assumes: https://upload.wikimedia.org/wikipedia/commons/2/23/Tree_rota
tion.png
        void rotateRight(AVLNode<T>* q, AVLNode<T>* p){
                if (Head == q){
                        Head = p;
                }
                if (q->Parent){
                        AVLNode<T>* superRoot = q->Parent;
                        if(superRoot->Left == q){
                                stitch(superRoot, p, true);
                        }else{
                                stitch(superRoot, p, false);
                        }
                }else{
                        p->Parent = nullptr;
                }
                AVLNode<T>* a = p->Left;
                AVLNode<T>* b = p->Right;
                AVLNode<T>* c = q->Right;

                stitch(p, a, true);
                stitch(p, q, false);
                stitch(q, b, true);
                stitch(q, c, false);
        }

        void rotateLeft(AVLNode<T>* p, AVLNode<T>* q){
                if (Head == p){
                        Head = q;
                }

                AVLNode<T> debug = *p;
                if (p->Parent){
                        AVLNode<T>* superRoot = p->Parent;
                        if(superRoot->Left == p){
                                stitch(superRoot, q, true);
                        }else{
                                stitch(superRoot, q, false);
                        }
                }else{
                        q->Parent = nullptr;
                }

                AVLNode<T>* a = p->Left;
                AVLNode<T>* b = q->Left;
                AVLNode<T>* c = q->Right;

                stitch(q, p, true);
```



```
                stitch(q, c, false);
                stitch(p, a, true);
                stitch(p, b, false);
        }

        int getHeight(AVLNode<T>* p){//yet another lazy function
                if(p){
                        return p->Height;
                }
                return 0;
        }

        void balance(AVLNode<T>* p){//where p is a newly inserted node

                if(!p){
                        //breakpoint
                }
                if (p->Parent == nullptr){
                        //std::cout <<"root";
                        return;
                }
                //std::cout<<"balancing";

                int balance;
                AVLNode<T>* q = p->Parent;
                while(q){
                        if(!p){
                                break;
                        }
                        //std::cout << " " << p->Value;

                        //AVLNode<T> debug = *q;
                        AVLNode<T> debug2 = *p;

                        p->Height = std::max(getHeight(p->Left), getHeight(p->Ri
ght)) + 1;
                        //std::cout << p->Height;
                        balance = getHeight(p->Right) - getHeight(p->Left);

                        if (balance <= -2){
                        //      std::cout<< "rotate right";
                                rotateRight(p, p->Left);
                                //balance = 0;
                                //AVLNode<T>* temp = q;
                                q = p->Parent;
                                p = p->Right;
                        }
                        if (balance >= 2){
                                //std::cout<< "rotate left";
                                rotateLeft(p, p->Right);
                               // balance = 0;
                                //AVLNode<T>* temp = q;
                                q = p->Parent;
                                p = p->Left;
                        }
                        if(p){
                                p = q->Parent;
                        }
                        if(q){
                                q = q->Parent;
                        }
                }
                //std::cout << std::endl;
        }


        ~AVLTree(){
                AVLNode<T>* p = Head;
                AVLNode<T>* ThisParent;
```

```
                while (p) {
                        if (!p->Left && !p->Right){
                                ThisParent = p->Parent;
                                if(ThisParent->Left == p){
                                        ThisParent->Left = nullptr;
                                }else{
                                        ThisParent->Right = nullptr;
                                }
                                delete p;
                                p = ThisParent;
                        } else if (!p->Left) {
                                p = p->Right;
                        } else {
                                p = p ->Left;
                        }
                }
        }

};
```



```
#include "AVLtree.hpp"
#include<iostream>
#include <vector>
#include <string>

AVLTree<int>* TestInsert(){
        AVLTree<int>* t = new AVLTree<int>();
        std::vector<int> values = {50,49,48,47,46,45,44,43,42,41};
        for (int i = 0; i < values.size(); i++){
                t->insert(values[i]);
                //t->print();
        }
        //std::cout<<t->Head->Right->Left->Right->Right->Value;
//      t->print();

        /*
        if (t->Head->Value != 50){
                std::cout << "insert failed";
        }

        if (t->Head->Left->Value != 25){
                std::cout << "insert failed on 25";
        }

        if (t->Head->Left->Left->Value != 10){
                std::cout << "insert failed on 10";
        }*/
        //std::cout<< "test";
        return t;
}

AVLTree<int>* TestLeftRotate(){
        AVLTree<int>* t = new AVLTree<int>();
        /*std::vector<int> values = {50, 51, 52,9,8,7,6,5,4,3};
        for (int i = 0; i < values.size(); i++){
                t->insert(values[i]);
        }*/

        t->insert(50);
        t->insert(51);
        t->insert(48);
        t->insert(47);

        //t->print();

        t->rotateLeft(t->Head, t->Head->Right);


        return t;

}


AVLTree<int>* TestRightRotate(){
        AVLTree<int>* t = new AVLTree<int>();
        /*std::vector<int> values = {50, 51, 52,9,8,7,6,5,4,3};
        for (int i = 0; i < values.size(); i++){
                t->insert(values[i]);
        }*/

        t->insert(50);
        t->insert(51);
        t->insert(48);
        t->insert(47);

        //t->print();

        t->rotateRight(t->Head, t->Head->Left);
```

```
        return t;

}


AVLTree<int>* TestDelete() {
        AVLTree<int>* t = new AVLTree<int>();
        std::vector<int> values = {50,49,48,47,46,45,44,43,42,41};
        for (int i = 0; i < values.size(); i++){
                t->insert(values[i]);
                //t->print();
        }


        AVLNode<int>* q = t->find(44);
        t->print();
        t->erase(q);
        //t->print();
        return t;
}

AVLTree<int>* TestFind(){
        AVLTree<int>* t = new AVLTree<int>();
        std::vector<int> values = {50, 25, 100, 10, 75, 76, 74};
        for (int i = 0; i < values.size(); i++){
                t->insert(values[i]);
        }
        AVLNode<int>* p = t->Head->Left->Left;
        AVLNode<int>* output = t->find(10);
        if (p != output){
                return t;
        }
        AVLTree<int>* q = new AVLTree<int>(); //it worked!
        AVLNode<int>* goodHead = new AVLNode<int>(666, nullptr, nullptr, nullptr
);
        q->Head = goodHead;
        return q;
}


AVLTree<int>* TestCopConstruct(){
        AVLTree<int>* t = new AVLTree<int>();
        std::vector<int> values = {50, 25, 100, 10, 75, 76, 74};
        for (int i = 0; i < values.size(); i++){
                t->insert(values[i]);
        }

        AVLTree<int>* alsot = new AVLTree<int>(*t);
        //t->print();
        //alsot->print();

        AVLTree<int>* q = new AVLTree<int>(); //it worked!
        AVLNode<int>* goodHead = new AVLNode<int>(666, nullptr, nullptr, nullptr
);
        q->Head = goodHead;
        return alsot;
}

AVLTree<int>* TestCopAssign(){
        AVLTree<int>* t = new AVLTree<int>();
        std::vector<int> values = {50, 25, 100, 10, 75, 76, 74};
        for (int i = 0; i < values.size(); i++){
                t->insert(values[i]);
        }

        AVLTree<int>* alsot = new AVLTree<int>();
        std::vector<int> values2 = {1,2,3,4,5,6,99,11,525,1245};
        for (int i = 0; i < values2.size(); i++){
```



```
                alsot->insert(values2[i]);
        }


        alsot = new AVLTree<int>(*t);
        //t->print();
        //alsot->print();

        AVLTree<int>* q = new AVLTree<int>(); //it worked!
        AVLNode<int>* goodHead = new AVLNode<int>(666, nullptr, nullptr, nullptr
);
        q->Head = goodHead;
        return alsot;
}

/*
AVLTree<int>* TestHeight(){
        AVLTree<int>* t = new AVLTree<int>();
        std::vector<int> values = {50, 25, 100, 10, 75, 76, 74};
        for (int i = 0; i < values.size(); i++){
                t->insert(values[i]);
        }
        int output = t->Head->SubTreeHeight();
        if (output == 4){
                AVLTree<int>* q = new AVLTree<int>(); //it worked!
                AVLNode<int>* goodHead = new AVLNode<int>(666, nullptr, nullptr,
 nullptr);
                q->Head = goodHead;
                return q;
        }
        return t;
}
*/

int main(){
        AVLTree<int>* t;
        t = TestInsert();
        t = TestLeftRotate();
        t = TestRightRotate();
        t = TestDelete();

        //t = TestFind();
        /*t = TestCopConstruct();
        t = TestCopAssign();
        //t = TestHeight();*/
        t->print();
        return 0;
}
```

```
#include <stack>
#include <iostream>

template<typename T>
struct Node {
        T Value;
        Node* Left;
        Node* Right;
        Node* Parent;

        //borrowed from my linked list project last semester
        Node() = default;

        Node(T v, Node* l, Node* r, Node* p)
        :Value(v), Left(l), Right(r), Parent(p){}

        Node & operator=(const Node & N){
                Value = N.Value;
                Right = N.Right;
                Left = N.Left;
                Parent = N.Parent;
                return *this;
        }
};

template<typename T>
struct Tree { //the non-balanced tree
        Node<T>* Head;

        Tree()
        :Head(nullptr){}


        /*
        average case: O(n)
        worst case: O(n)
        they are the same because no matter what we have to traverse the entire
tree
        all of the operations with the main traversial stack and the 'twoChild'
stack are constant time because they are pointer operations
        */
         Tree(const Tree<T> & t){
                if (t.Head == nullptr){
                        Head = nullptr;
                        return;
                }

                Head = new Node<T>(*t.Head);
                Node<T>* r = Head;
                Node<T>* n;
                Node<T>* temp;
                std::stack<Node<T>*> Stack;
                std::stack<Node<T>*> twoChild;

                 if (t.Head->Left){
                        Stack.push(t.Head->Left);
                 }
                 if (t.Head->Right){
                        Stack.push(t.Head->Right);
                 }

                twoChild.push(Head);
                while(!Stack.empty()){

                        n = Stack.top();
                        //do visit
                        Node<T>* newNode = new Node<T>(*n);
                        if (r->Value > n->Value){
                                r->Left = newNode;
```



```
                        }else{
                                r->Right = newNode;
                        }
                        newNode->Parent = r;
                        r = newNode;

                        if (n->Left && n->Right){
                                twoChild.push(r);
                        }else if (!n->Left && !n->Right){
                                //this resets r to the last node with two childr
en so that our traversial of the new tree moves in step with our pre-order trave
rsial
                                if(!twoChild.empty()){
                                        r = twoChild.top();
                                        twoChild.pop();
                                }
                        }


                        Stack.pop();
                        if (n->Left){
                                Stack.push(n->Left);
                        }
                        if (n->Right){
                                Stack.push(n->Right);
                        }
                }
        }
        /*
        average case: O(k+n)
        worst case: O(k+n)
        where k is the size of the old tree and n is the size of the new tree
        again we have to traverse the entire tree in order to actuially delete i
t and copy
        */
        Tree & operator=(const Tree & t){
                Node<T>* p = Head;
                Node<T>* ThisParent;
                while (Head) {

                        if (!p->Left && !p->Right){
                                ThisParent = p->Parent;
                                erase(p);
                                p = ThisParent;
                        } else if (!p->Left) {
                                p = p->Right;
                        } else {
                                p = p ->Left;
                        }
                }
                Tree newTree = Tree(t);
                Head = newTree->Head;
                return *this;

        }

        /*
        average case: O(log(n))
        worst case: O(n)
        the average case arises when we have a nicely destributed set of values
        the worst case is if we have a sorted list inputed
        */
        Node<T>* find(T val) {
                Node<T>* p = Head;
                while(p) {
                        if (val > p->Value) {
                                p = p->Right;
                        }else if (val < p->Value) {
                                p = p->Left;
```

```
                        }else{
                                return p;
                        }
                }
                return p;
        }

        /*
        average case: O(log(n))
        worst case: O(n)
        the average case arises when we have a nicely destributed set of value.

        the worst case is if we have a sorted list inputed and we are inserting
at the bottom
        */
        void insert(T val) {
                if (Head == nullptr){
                        Node<T>* temp = new Node<T>(val, nullptr, nullptr, nullp
tr);
                        Head = temp;
                        return;
                }
                Node<T>* p = Head;
                Node<T>* ThisParent;
                bool DidGoLeft;
                while(p) {
                        ThisParent = p;
                        if (val >=p->Value) {
                                p = p->Right;
                                DidGoLeft = false; //I'm not entirely happy with
 this
                        }else {
                                p = p->Left;
                                DidGoLeft = true;
                        }
                }
                Node<T>* newNode =  new Node<T>(val, nullptr, nullptr, ThisParen
t);
                if (DidGoLeft){
                        ThisParent->Left = newNode;
                }else{
                        ThisParent->Right = newNode;
                }
                if (!newNode){
                        //breakpoint here!
                        Node<T> debug = *newNode;
                }
        }

        /*
        average case: O(1)
        worst case: O(log(n))
        the average case arises when we are deleting a node with zero or one chi
ldren
        the worst case is if we have a large set and we are deleting the head an
d have to traverse all the way to the bottom to get the successor
        */
        void erase(Node<T>* k){
                Node<T>* p = k->Parent;
                if (k->Left && k->Right){
                        Node<T>* r = successor(k);
                        k->Value = r->Value;
                        erase(r); //r could have a right child
                }else if(k->Left){
                        if (p->Left == k){
                                p->Left = k->Left;
                        }else{
                                p->Right = k->Left;
                        }
```



```
                        k->Left->Parent = p;
                        delete k;
                }else if(k->Right){
                        if (p->Left == k){
                                p->Left = k->Right;
                        }else{
                                p->Right = k->Right;
                        }
                        k->Right->Parent = p;
                        delete k;
                }else{
                        if (p->Left == k){
                                p->Left = nullptr;
                        }else{
                                p->Right = nullptr;
                       }
                        delete k;
                }
        }
        //at worst O(log(n)), see above
        Node<T>* successor(Node<T>* k){
                Node<T>* p = k->Right;
                Node<T>* output;
                while (p){
                        output = p;
                        p = p->Left;
                }
                return output;
        }

        //O(n)
        void print(){
                std::cout << "digraph G {" << std::endl;
                Node<T>* n;
                std::stack<Node<T>*> Stack;
                int counter = 0;
                int otherCounter = 0;

                Stack.push(Head);
                while(!Stack.empty()){
                        n = Stack.top();
                        //do visit
                        if (!n->Parent){
                                std::cout << "HEAD" << "->" << n->Value << ";" <
< std::endl;
                        }else{
                                std::cout << n->Parent->Value <<  "->" << n->Val
ue  << ";" << std::endl;
                        }
                        if (!n->Left) {
                                std::cout << n->Value << "->" << "Leftnull" << cou
nter << ";" << std::endl;

                        }
                        if (!n->Right) {
                                std::cout << n->Value <<"->" << "Righttnull" << cou
nter << ";" << std::endl;

                        }
                        counter++;

                        Stack.pop();
                        if (n->Right){
                                Stack.push(n->Right);
                        }
                        if (n->Left){
                                Stack.push(n->Left);
                        }
                }
```

```
                std::cout << "}";
        }

        /*
        average: O(n)
        worst case: O(n)
        again, we have to traverse the whole set
        */
        ~Tree(){
                Node<T>* p = Head;
                Node<T>* ThisParent;
                while (p) {
                        if (!p->Left && !p->Right){
                                ThisParent = p->Parent;
                                if(ThisParent->Left == p){
                                        ThisParent->Left = nullptr;
                                }else{
                                        ThisParent->Right = nullptr;
                                }
                                delete p;
                                p = ThisParent;
                        } else if (!p->Left) {
                                p = p->Right;
                        } else {
                                p = p ->Left;
                        }
                }
        }

};
```



```
#include "tree.hpp"
//#include "benchmark.hpp"
#include<iostream>
#include <vector>
#include <string>

Tree<int>* TestInsert(){
        Tree<int>* t = new Tree<int>();
        t->insert(50);
        t->insert(51);
        t->insert(52);
        t->insert(53);

//      t->print();

        /*
        if (t->Head->Value != 50){
                std::cout << "insert failed";
        }

        if (t->Head->Left->Value != 25){
                std::cout << "insert failed on 25";
        }

        if (t->Head->Left->Left->Value != 10){
                std::cout << "insert failed on 10";
        }*/
        return t;
}

Tree<int>* TestDelete() {
        Tree<int>* t = new Tree<int>();
        t->insert(50);
        t->insert(25);
        t->insert(100);
        t->insert(10);
        t->insert(75);
        t->insert(76);
        t->insert(74);
        //t->print();

        t->erase(t->Head->Right->Left->Right);
        return t;
}

Tree<int>* TestFind(){
        Tree<int>* t = new Tree<int>();
        std::vector<int> values = {50, 25, 100, 10, 75, 76, 74};
        for (int i = 0; i < values.size(); i++){
                t->insert(values[i]);
        }
        Node<int>* p = t->Head->Left->Left;
        Node<int>* output = t->find(10);
        if (p != output){
                return t;
        }
        Tree<int>* q = new Tree<int>(); //it worked!
        Node<int>* goodHead = new Node<int>(666, nullptr, nullptr, nullptr);
        q->Head = goodHead;
        return t;
}


Tree<int>* TestCopConstruct(){
        Tree<int>* t = new Tree<int>();
        std::vector<int> values = {50, 25, 100, 10, 75, 76, 74};
        for (int i = 0; i < values.size(); i++){
                t->insert(values[i]);
        }
```

```
        Tree<int>* alsot = new Tree<int>(*t);
        //t->print();
        //alsot->print();

        Tree<int>* q = new Tree<int>(); //it worked!
        Node<int>* goodHead = new Node<int>(666, nullptr, nullptr, nullptr);
        q->Head = goodHead;
        return alsot;
}

Tree<int>* TestCopAssign(){
        Tree<int>* t = new Tree<int>();
        std::vector<int> values = {50, 25, 100, 10, 75, 76, 74};
        for (int i = 0; i < values.size(); i++){
                t->insert(values[i]);
        }

        Tree<int>* alsot = new Tree<int>();
        std::vector<int> values2 = {1,2,3,4,5,6,99,11,525,1245};
        for (int i = 0; i < values2.size(); i++){
                alsot->insert(values2[i]);
        }


        alsot = new Tree<int>(*t);
        //t->print();
        //alsot->print();

        Tree<int>* q = new Tree<int>(); //it worked!
        Node<int>* goodHead = new Node<int>(666, nullptr, nullptr, nullptr);
        q->Head = goodHead;
        return alsot;
}



int main(){
        Tree<int>* t;
        t = TestInsert();
        t = TestDelete();
        t = TestFind();
        t = TestCopConstruct();
        t = TestCopAssign();
        //t->print();
        return 0;
}
```



```
#include <chrono>
#include <iostream>
#include <random>
#include <vector>
#include "tree.hpp"


void TestInsert(){
 std::mt19937 prbg;

  for (int n = 1000; n <= 500000; n += 10000) {

    // Get the starting time point. The type is deduced because it's hard
    // to spell (it is std::chrono::system_clock::time_point).
    auto start = std::chrono::system_clock::now();

    // The actual test.
    Tree<int>* tree = new Tree<int>();
    //std::vector<int> seq;
    for (int i = 0; i < n; ++i) {
      std::uniform_int_distribution<int> rand(0, i);

      //int num = rand(prbg);                // Generate a random number
      tree->insert(i);
      //auto iter = linear_search(seq, num); // Find the insertion point
      //seq.insert(iter, num);
    }

    // Get the current system time in nanoseconds.
    auto stop = std::chrono::system_clock::now();

    // Print the number of nanoseconds each test takes.
    std::cout << n << "," << (stop - start).count() << std::endl;
  }

}

void TestFind(){
 std::mt19937 prbg;

  for (int n = 1000; n <= 50000; n += 1000) {

    Tree<int>* tree = new Tree<int>();
    std::vector<int> seq;
    for (int i = 0; i < n; ++i) {
      std::uniform_int_distribution<int> rand(0, i);

      //int num = rand(prbg);                // Generate a random number
      tree->insert(i);
      //auto iter = linear_search(seq, num); // Find the insertion point
      seq.push_back(i);
    }
    auto start = std::chrono::system_clock::now();
    for (int f = 0; f < seq.size(); f++){
        Node<int>* test = tree->find(f);
    }
    // Get the current system time in nanoseconds.
    auto stop = std::chrono::system_clock::now();

    // Print the number of nanoseconds each test takes.
    std::cout << n << "," << (stop - start).count() << std::endl;
  }

}


int main()
{
 // TestInsert();
```

```
  TestFind();
}
```



```
#include <chrono>
#include <iostream>
#include <random>
#include <vector>
#include "AVLtree.hpp"


void TestInsert(){
 std::mt19937 prbg;

  for (int n = 1000; n <= 50000; n += 1000) {

    // Get the starting time point. The type is deduced because it's hard
    // to spell (it is std::chrono::system_clock::time_point).
    auto start = std::chrono::system_clock::now();

    // The actual test.
    AVLTree<int>* tree = new AVLTree<int>();
    //std::vector<int> seq;
    for (int i = 0; i < n; ++i) {
      std::uniform_int_distribution<int> rand(0, i);

      int num = rand(prbg);                // Generate a random number
      tree->insert(num);
      //auto iter = linear_search(seq, num); // Find the insertion point
      //seq.insert(iter, num);
    }

    // Get the current system time in nanoseconds.
    auto stop = std::chrono::system_clock::now();

    // Print the number of nanoseconds each test takes.
    std::cout << n << "," << (stop - start).count() << std::endl;
  }

}

void TestFind(){
 std::mt19937 prbg;

  for (int n = 1000; n <= 50000; n += 1000) {

    AVLTree<int>* tree = new AVLTree<int>();
    std::vector<int> seq;
    for (int i = 0; i < n; ++i) {
      std::uniform_int_distribution<int> rand(0, i);

      int num = rand(prbg);                // Generate a random number
      tree->insert(n);
      //auto iter = linear_search(seq, num); // Find the insertion point
      seq.push_back(n);
    }
    auto start = std::chrono::system_clock::now();
    for (int f = 0; f < seq.size(); f++){
        AVLNode<int>* test = tree->find(f);
    }
    // Get the current system time in nanoseconds.
    auto stop = std::chrono::system_clock::now();

    // Print the number of nanoseconds each test takes.
    std::cout << n << "," << (stop - start).count() << std::endl;
  }

}

int main()
{
  //TestInsert();
  TestFind();
```

```
}
```