Eric Blasko

CSE 330 Data Structurs

Winder 2018

Lab 7 – Binary Search Tree

* **Status**

100% complete

* **Time Complexity**

The following member functions are O(1) as they are constant in time. No need to traverse through tree

* BinarySearchTree()
* BinarySearchTree(BinarySearchTree && rhs);
* BinarySearchTree & operator=(const BinarySearchTree & rhs);
* BinarySearchTree & operator=(BinarySearchTree && rhs);
* bool isEmpty() const {return root == nullptr;}

The following member functions are O(n) where n is the size of the tree. Each requires traversal through entire tree. While helper functions only make a single call to another function, there time is based on the function being called

* BinarySearchTree(const BinarySearchTree & rhs) {root = clone(rhs.root);}
* BinaryNode \* clone(BinaryNode \* t) const;
* ~BinarySearchTree() {makeEmpty(root);}
* void makeEmpty(BinaryNode \* & t);
* void makeEmpty() {makeEmpty(root);}
* void printTree(ostream & out = cout) const;
* void printTree(ostream & out, BinaryNode \* t, string indent, const string & tag) const;
* void inorder(BinaryNode \* t) const;
* void postorder(BinaryNode \* t) const;
* void preorder(BinaryNode \* t) const;
* void inorder() const {inorder(root);}
* void postorder() const {postorder(root);}
* void preorder() const {preorder(root);}

The following member functions are O(logn) where n is the size of the tree. Only one node is traversed at each depth of tree. While helper functions only make a single call to another function, there time is based on the function being called

* const Comparable & findMin() const;
* const Comparable & findMax() const;
* BinaryNode \* findMin(BinaryNode \* t) const;
* BinaryNode \* findMax(BinaryNode \* t) const;
* void insert(const Comparable & x, BinaryNode \* & t);
* void insert(Comparable && x, BinaryNode \* & t);
* void insert(const Comparable & x) { insert(x,root);}
* void insert(Comparable && x) {insert(move(x),root);}
* bool contains(const Comparable & x) const;
* bool contains(const Comparable & x, BinaryNode \* t) const;
* void remove(const Comparable & x, BinaryNode \* & t);
* void remove(const Comparable & x) {remove(x,root);}
* **Source Code**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Eric Blasko

\* BinarySearchTree.h

\* 02/28/2018

\* This class is a replica of the Binary Search tree in the standard library. Each node of the tree

\* has at most two children, with the left being of lesser value and the right being of higher value.

\* Most functions consist of a helper function to tell the main function where to start, which is always

\* at the root

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#ifdef BINARY\_SEARCH\_TREE\_H

#define BINARY\_SEARCH\_TREE\_H

#include <ostream>

#include <cassert>

using namespace std;

template <typename Comparable>

class BinarySearchTree

{

public:

BinarySearchTree(): root{nullptr} {}

BinarySearchTree(const BinarySearchTree & rhs) {root = clone(rhs.root);}

BinarySearchTree(BinarySearchTree && rhs);

~BinarySearchTree() {makeEmpty(root);}

BinarySearchTree & operator=(const BinarySearchTree & rhs);

BinarySearchTree & operator=(BinarySearchTree && rhs);

const Comparable & findMin() const;

const Comparable & findMax() const;

bool contains(const Comparable & x) const;

bool isEmpty() const {return root == nullptr;}

void printTree(ostream & out = cout) const;

void makeEmpty() {makeEmpty(root);}

void insert(const Comparable & x) { insert(x,root);}

void insert(Comparable && x) {insert(move(x),root);}

void remove(const Comparable & x) {remove(x,root);}

void inorder() const {inorder(root);}

void postorder() const {postorder(root);}

void preorder() const {preorder(root);}

private:

struct BinaryNode

{

Comparable element;

BinaryNode \* left;

BinaryNode \* right;

BinaryNode(const Comparable & theElement, BinaryNode \* lt, BinaryNode \* rt):

element{theElement}, left{lt}, right{rt} {}

BinaryNode(Comparable && theElement, BinaryNode \* lt, BinaryNode \* rt):

element{move(theElement)}, left{lt}, right{rt} {}

};

BinaryNode \* root;

void insert(const Comparable & x, BinaryNode \* & t);

void insert(Comparable && x, BinaryNode \* & t);

void remove(const Comparable & x, BinaryNode \* & t);

BinaryNode \* findMin(BinaryNode \* t) const;

BinaryNode \* findMax(BinaryNode \* t) const;

bool contains(const Comparable & x, BinaryNode \* t) const;

void makeEmpty(BinaryNode \* & t);

void printTree(ostream & out, BinaryNode \* t, string indent, const string & tag) const;

BinaryNode \* clone(BinaryNode \* t) const;

void inorder(BinaryNode \* t) const;

void postorder(BinaryNode \* t) const;

void preorder(BinaryNode \* t) const;

};

//Copies entire contents of a binary search tree to create another identical instance

template <typename Comparable>

typename BinarySearchTree<Comparable>::BinaryNode \* BinarySearchTree<Comparable>::clone(BinaryNode \* t) const

{

if(t == nullptr)

return nullptr;

else

{

BinaryNode \* newNode = new BinaryNode(t->element, clone(t->left), clone(t->right));

assert(newNode);

return newNode;

}

}

//Move constructor

template <typename Comparable>

BinarySearchTree<Comparable>::BinarySearchTree(BinarySearchTree && rhs)

{

root = rhs.root;

rhs.root = nullptr;

}

//deletes every element in the BST starting with leaves, and working there way back to root. root

//is left null

template <typename Comparable>

void BinarySearchTree<Comparable>::makeEmpty(BinaryNode \* & t)

{

if(t != nullptr)

{

makeEmpty(t->left);

makeEmpty(t->right);

delete t;

}

t = nullptr;

}

//Copies contents of source BST to current BST

template <typename Comparable>

BinarySearchTree<Comparable> & BinarySearchTree<Comparable>::operator=(const BinarySearchTree & rhs)

{

BinarySearchTree copy = rhs;

swap(\*this, copy);

return \*this;

}

//Moves contents of source BST to current BST

template <typename Comparable>

BinarySearchTree<Comparable> & BinarySearchTree<Comparable>::operator=(BinarySearchTree && rhs)

{

root = rhs.root;

rhs.root = nullptr;

return \*this;

}

//helper function which tells findMin to start at root

template <typename Comparable>

const Comparable & BinarySearchTree<Comparable>::findMin() const

{

BinaryNode \* t = findMin(root);

if(t == nullptr)

{

BinaryNode \* temp = new BinaryNode(Comparable(),nullptr,nullptr);

return temp->element;

}

return t->element;

}

//helper function which tells findMax to start at root

template <typename Comparable>

const Comparable & BinarySearchTree<Comparable>::findMax() const

{

BinaryNode \* t = findMax(root);

if(t == nullptr)

{

BinaryNode \* temp = new BinaryNode(Comparable(),nullptr,nullptr);

return temp->element;

}

return t->element;

}

//Recursivly calls until reaches bottom left leaf.

template <typename Comparable>

typename BinarySearchTree<Comparable>::BinaryNode \* BinarySearchTree<Comparable>::findMin(BinaryNode \* t) const

{

if (t == nullptr)

return nullptr;

if (t->left == nullptr)

return t;

return findMin(t->left);

}

//Recursivly calls until reaches bottom right leaf.

template <typename Comparable>

typename BinarySearchTree<Comparable>::BinaryNode \* BinarySearchTree<Comparable>::findMax(BinaryNode \* t) const

{

if (t == nullptr)

return nullptr;

if (t->right == nullptr)

return t;

return findMax(t->right);

}

//helper functions for printing contents of tree. Tells to start at root

template <typename Comparable>

void BinarySearchTree<Comparable>::printTree(ostream & out) const

{

cout << "Print Tree\n";

printTree(out,root, "", "");

}

//Recursivly goes through elements of tree to print to console. Each deeper call increases indent

template <typename Comparable>

void BinarySearchTree<Comparable>::printTree(ostream & out, BinaryNode \* t, string indent, const string & tag) const

{

if (t == nullptr)

return;

out << indent << tag << t->element << endl;

indent += " ";

printTree(out, t->left, indent, "L ");

printTree(out, t->right, indent, "R ");

}

//recursivily compares values to find correct place to insert in tree

template <typename Comparable>

void BinarySearchTree<Comparable>::insert(const Comparable & x, BinaryNode \* & t)

{

if(t == nullptr)

t = new BinaryNode(x, nullptr, nullptr);

else if (x < t->element)

insert(x, t->left);

else if (x > t->element)

insert(x, t->right);

else

;

}

//recursivily compares value to find correct place to insert in tree

template <typename Comparable>

void BinarySearchTree<Comparable>::insert(Comparable && x, BinaryNode \* & t)

{

if(t == nullptr)

t = new BinaryNode(move(x),nullptr,nullptr);

else if(x < t->element)

insert(move(x), t->left);

else if(x > t->element)

insert(move(x), t->right);

else

;

}

//prints elements in tree in order (left,node,right)

template <typename Comparable>

void BinarySearchTree<Comparable>::inorder(BinaryNode \* t) const

{

if (t == nullptr)

return;

inorder(t->left);

cout << t->element << " ";

inorder(t->right);

}

//prints elements in tree post order (left,right,node)

template <typename Comparable>

void BinarySearchTree<Comparable>::postorder(BinaryNode \* t) const

{

if(t == nullptr)

return;

postorder(t->left);

postorder(t->right);

cout << t->element << " ";

}

//prints elements in tree pre order (node, left, right)

template <typename Comparable>

void BinarySearchTree<Comparable>::preorder(BinaryNode \* t) const

{

if(t == nullptr)

return;

cout << t->element << " ";

preorder(t->left);

preorder(t->right);

}

//helper function which tells contain to start at root

template <typename Comparable>

bool BinarySearchTree<Comparable>::contains(const Comparable & x) const

{

return contains(x,root);

}

//recursively calls to one side of tree till found or at nullptr

template <typename Comparable>

bool BinarySearchTree<Comparable>::contains(const Comparable & x, BinaryNode \* t) const

{

if(t == nullptr)

return false;

else if(x < t->element)

return contains(x,t->left);

else if(x > t->element)

return contains(x,t->right);

else

return true;

}

//first part searches tree bases on size of element. If element has two children, find

//the min value to the right of that node and assing to that position. then find that

//node and remove. If one child move child to spot being deleted. If no children

//delete node.

template <typename Comparable>

void BinarySearchTree<Comparable>::remove(const Comparable & x, BinaryNode \* & t)

{

if(t == nullptr)

return;

if(x < t->element)

remove(x,t->left);

else if(x > t->element)

remove(x,t->right);

else if(t->left != nullptr && t->right != nullptr)

{

t->element = findMin(t->right)->element;

remove(t->element,t->right);

}

else

{

BinaryNode \*oldNode = t;

t = (t->left != nullptr) ? t->left : t->right;

delete oldNode;

}

}

#endif

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Eric Blasko

\* BST\_test1.cpp

\* 02/28/2018

\* This program test the methods of the class BinarySearcHTree.h. The program will demonstraight the

\* insert, findMin, findMax, remove, and print methods. PrintTree will print elements according to the

\* depth that they are in the tree

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <iostream>

#include "BinarySearchTree.h"

using namespace std;

//Main function to test methods of BinarySearchTree.h

int main()

{

BinarySearchTree<int> tree;

tree.insert(6);

tree.insert(8);

tree.insert(2);

tree.insert(1);

tree.insert(5);

tree.insert(3);

tree.insert(4);

tree.insert(1);

tree.printTree();

cout << "Min = " << tree.findMin() << endl;

cout << "Max = " << tree.findMax() << endl;

cout << "Remove 2\n";

tree.remove(2);

tree.printTree();

cout << "Contains 2? " << tree.contains(2) << endl;

cout << "Contains 4? " << tree.contains(4) << endl;

cout << "Copy Constructor\n";

BinarySearchTree<int> copy(tree);

cout << "Remove 6\n";

copy.remove(6);

copy.printTree();

cout << "Inorder Traversal\n";

copy.inorder();

cout << endl;

cout << "Clear tree\n";

copy.makeEmpty();

copy.printTree();

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Eric Blasko

\* BST\_test2.cpp

\* 02/28/2018

\* This program test the different traversal options available int the BinarhSearchTree.h

\* class. The following traversals follow the pattersn: inorder(left,root,right),

\* postorder(left,right,root), and preorder(root,left,right)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <iostream>

#include "BinarySearchTree.h"

using namespace std;

//Main function that test methods of BinarySearchTree.h

int main()

{

BinarySearchTree<int> complete\_tree;

complete\_tree.insert(20);

complete\_tree.insert(10);

complete\_tree.insert(30);

complete\_tree.insert(35);

complete\_tree.insert(25);

complete\_tree.insert(5);

complete\_tree.insert(15);

complete\_tree.printTree();

cout << "Inorder Traversal\n";

complete\_tree.inorder();

cout << endl;

cout << "Postorder Traversal\n";

complete\_tree.postorder();

cout << endl;

cout << "Preorder Traversal\n";

complete\_tree.preorder();

cout << endl;

}

* **Sample Runs**

**Test 1**

Script started on 2018-02-26 20:19:30-0800

]0;005670557@csusb.edu@jb358-0:~/cse330/lab7[005670557@csusb.edu@jb358-0 lab7]$ g++ -c BST\_test1.cpp

]0;005670557@csusb.edu@jb358-0:~/cse330/lab7[005670557@csusb.edu@jb358-0 lab7]$ g++ BST\_test1.o

]0;005670557@csusb.edu@jb358-0:~/cse330/lab7[005670557@csusb.edu@jb358-0 lab7]$ ./a.out

Print Tree

6

L 2

L 1

R 5

L 3

R 4

R 8

Min = 1

Max = 8

Remove 2

Print Tree

6

L 3

L 1

R 5

L 4

R 8

Contains 2? 0

Contains 4? 1

Copy Constructor

Remove 6

Print Tree

8

L 3

L 1

R 5

L 4

Inorder Traversal

1 3 4 5 8

Clear tree

Print Tree

]0;005670557@csusb.edu@jb358-0:~/cse330/lab7[005670557@csusb.edu@jb358-0 lab7]$ exit

Script done on 2018-02-26 20:19:56-0800

**Test 2**

Script started on 2018-02-26 20:20:05-0800

]0;005670557@csusb.edu@jb358-0:~/cse330/lab7[005670557@csusb.edu@jb358-0 lab7]$ g++ -c BST\_test2.cpp

]0;005670557@csusb.edu@jb358-0:~/cse330/lab7[005670557@csusb.edu@jb358-0 lab7]$ g++ -[KBST\_test2.o

]0;005670557@csusb.edu@jb358-0:~/cse330/lab7[005670557@csusb.edu@jb358-0 lab7]$ ./a.out

Print Tree

20

L 10

L 5

R 15

R 30

L 25

R 35

Inorder Traversal

5 10 15 20 25 30 35

Postorder Traversal

5 15 10 25 35 30 20

Preorder Traversal

20 10 5 15 30 25 35

]0;005670557@csusb.edu@jb358-0:~/cse330/lab7[005670557@csusb.edu@jb358-0 lab7]$ exit

Script done on 2018-02-26 20:20:32-0800

* **Discussion**

In the helper member functions for findMIn() and findMax(), both had a line of code that returned Comparable(). This resulted in a warning from the compiler about returning a temporary value. This is because the member function is supposed to return a constant Comparable value which is unchanged, but the compiler recognizes that the value will go out of scope after it returns. To get around this warning, we could simply create a temporary node which has null pointer and a default value of Comparable(). This allows us to return a default value that the compiler doesn’t recognize as a temporary value. After the function returns, the temporary node will go out of scope and the destructor will be called.