```
/*Author: Eric Gustin
  Assignment: CPSC223-01 HW09 hw9_tests.cpp
  Description: This program tests the biunary search tree collection.
  It tests every function in bst_collection.h
*/
#include <iostream>
#include <string>
#include <gtest/gtest.h>
#include "bst_collection.h"

using namespace std;

// Test 1
TEST(BasicCollectionTest, CorrectSize) {
 BSTCollection<string,double> c;
 ASSERT_EQ(0, c.size());
 c.insert("a", 10.0);
 ASSERT_EQ(1, c.size());
 c.insert("b", 20.0);
 ASSERT_EQ(2, c.size());
}
// Test 2
TEST(BasicCollectionTest, InsertAndFind) {
 BSTCollection<string,double> c;
 double v;
 ASSERT_EQ(false, c.find("a", v));
 c.insert("a", 10.0);
 ASSERT_EQ(true, c.find("a", v));
 ASSERT_EQ(v, 10.0);
 ASSERT_EQ(false, c.find("b", v));
 c.insert("b", 20.0);
 ASSERT_EQ(true, c.find("b", v));
 ASSERT_EQ(20.0, v);
}

// Test 3
TEST(BasicCollectionTest, RemoveElems) {
 BSTCollection<string,double> c;
 c.insert("d", 10.0);
 c.insert("g", 20.0);
 c.insert("l", 30.0);
 c.insert("j", 30.0);
 c.insert("i", 30.0);
 c.insert("k", 30.0);
  double v;
 vector<string> ks1;
 c.keys(ks1);
 for (int i = 0; i < ks1.size()-1; ++i)
  ASSERT_LE(ks1[i], ks1[i+1]);
 c.remove("l"); // remove a node with one child on the left
 ASSERT_EQ(false, c.find("l", v));
 ASSERT_EQ(true, c.find("i", v));
 ASSERT_EQ(true, c.find("k", v));
 ASSERT_EQ(5, c.size());

 BSTCollection<string,double> d;
 d.insert("f", 10.0);
 d.insert("v", 20.0);
 d.insert("z", 30.0);
 d.insert("j", 30.0);
 d.insert("g", 30.0);
 d.insert("k", 30.0);
 d.insert("n", 30.0);
 d.insert("l", 30.0);
 d.insert("o", 30.0);
 vector<string> ks2;
 d.remove("k"); // remove a node with one child on the right
 d.remove("y"); // try to remove a node that does not exist in the BST
 vector<string> ks3;
 d.keys(ks3);
 for (int i = 0; i < ks3.size()-1; ++i)
  ASSERT_LE(ks3[i], ks3[i+1]);
 ASSERT_EQ(false, d.find("k", v));
 ASSERT_EQ(true, d.find("l", v));
 ASSERT_EQ(true, d.find("o", v));
 ASSERT_EQ(8, d.size());


 BSTCollection<string,double> e;
 e.insert("f", 30.0);
 e.insert("i", 30.0);
 e.insert("c", 30.0);
```

```
 e.insert("b", 30.0);
 e.insert("d", 30.0);
 e.insert("cd", 30.0);
 e.insert("e", 30.0);
 e.remove("c"); // remove a node that has two children with the right child having two
 ASSERT_EQ(6, e.size());
 vector<string> ks4;
 e.keys(ks4);
 for (int i = 0; i < ks4.size()-1; ++i)
  ASSERT_LE(ks4[i], ks4[i+1]);
 ASSERT_EQ(false, e.find("c", v));
 ASSERT_EQ(true, e.find("e", v));
 ASSERT_EQ(true, e.find("cd", v));
 e.remove ("d");
 ASSERT_EQ(5, e.size());
 ASSERT_EQ(false, e.find("d", v));
 ASSERT_EQ(true, e.find("b", v));
 e.remove("f"); // remove a root node with two children
 ASSERT_EQ(4, e.size());
 ASSERT_EQ(false, e.find("f", v));
 ASSERT_EQ(true, e.find("i", v));
 e.remove("i"); // remove a root that has one child
 ASSERT_EQ(3, e.size());
 ASSERT_EQ(false, e.find("i", v));
 ASSERT_EQ(true, e.find("b", v));
 e.remove("b");
 e.remove("cd");
 ASSERT_EQ(1, e.size());
 ASSERT_EQ(true, e.find("e", v));
 e.remove("e");
 ASSERT_EQ(0, e.size());
 ASSERT_EQ(false, e.find("e", v));

 BSTCollection<string,double> f;
 f.insert("f", 30.0);
 f.insert("i", 30.0);
 f.insert("c", 30.0);
 f.insert("b", 30.0);
 f.insert("d", 30.0);
 f.insert("e", 30.0);
 f.remove("c"); // remove a node that has two children with the right child having a right child
 vector<string> ks5;
 f.keys(ks5);
 for (int i = 0; i < ks5.size()-1; ++i)
  ASSERT_LE(ks5[i], ks5[i+1]);
 ASSERT_EQ(false, f.find("c", v));
 ASSERT_EQ(true, f.find("e", v));
 ASSERT_EQ(true, f.find("d", v));
 ASSERT_EQ(true, f.find("b", v));
 ASSERT_EQ(5, f.size());
}

// Test 4
TEST(BasicCollectionTest, GetKeys) {
 BSTCollection<string,double> c;
 c.insert("a", 10.0);
 c.insert("b", 20.0);
 c.insert("c", 30.0);
 vector<string> ks;
 c.keys(ks);
 vector< string>::iterator iter;
 iter = find(ks.begin(), ks.end(), "a");
 ASSERT_NE(ks.end(), iter);
 iter = find(ks.begin(), ks.end(), "b");
 ASSERT_NE(ks.end(), iter);
 iter = find(ks.begin(), ks.end(), "c");
 ASSERT_NE(ks.end(), iter);
 iter = find(ks.begin(), ks.end(), "d");
 ASSERT_EQ(ks.end(), iter);
}
// Test 5
TEST(BasicCollectionTest, GetKeysAdvanced) {
 BSTCollection<string,double> c;
 c.insert("g", 10.0);
 c.insert("c", 20.0);
 c.insert("b", 30.0);
 c.insert("q", 10.0);
 c.insert("f", 20.0);
 c.insert("a", 30.0);
 c.insert("d", 10.0);
 c.insert("h", 20.0);
 c.insert("t", 30.0);
```

```
 c.insert("i", 10.0);
 c.insert("qq", 20.0);
 c.insert("qqq", 30.0);
 vector<string> ks;
 c.keys(ks);
 for (int i = 0; i < ks.size()-1; ++i)
  ASSERT_LE(ks[i], ks[i+1]);
}
// Test 6
TEST(BasicCollectionTest, GetKeyRange) {
 BSTCollection<string,double> c;
 c.insert("a", 10.0);
 c.insert("b", 20.0);
 c.insert("c", 30.0);
 c.insert("d", 40.0);
 c.insert("e", 50.0);
 vector<string> ks;
 c.find("b", "d", ks);
 vector<string>::iterator iter;
 iter = find(ks.begin(), ks.end(), "b");
 ASSERT_NE(ks.end(), iter);
 iter = find(ks.begin(), ks.end(), "c");
 ASSERT_NE(ks.end(), iter);
 iter = find(ks.begin(), ks.end(), "d");
 ASSERT_NE(ks.end(), iter);
 iter = find(ks.begin(), ks.end(), "a");
 ASSERT_EQ(ks.end(), iter);
 iter = find(ks.begin(), ks.end(), "e");
 ASSERT_EQ(ks.end(), iter);
}
// Test 7
TEST(BasicCollectionTest, GetKeyRangeAdvanced) {
 BSTCollection<string,double> c;
 c.insert("g", 10.0);
 c.insert("c", 20.0);
 c.insert("b", 30.0);
 c.insert("q", 10.0);
 c.insert("f", 20.0);
 c.insert("a", 30.0);
 c.insert("d", 10.0);
 c.insert("h", 20.0);
 c.insert("t", 30.0);
 c.insert("i", 10.0);
 c.insert("qq", 20.0);
 c.insert("qqq", 30.0);
 vector<string> ks;
 c.find("cd", "q", ks);
 ASSERT_EQ(ks.size(), 6);
}
// Test 8
TEST(BasicCollectionTest, KeySort) {
 BSTCollection<string,double > c;
 c.insert("a", 10.0);
 c.insert("e", 50.0);
 c.insert("c", 30.0);
 c.insert("b", 20.0);
 c.insert("d", 40.0);
 c.insert("aaa", 10.0);
 c.insert("dfe", 50.0);
 c.insert("chh", 30.0);
 c.insert("zsdb", 20.0);
 c.insert("zxsd", 40.0);
 vector<string> sorted_ks;
 c.sort(sorted_ks);
 ASSERT_EQ(c.size(), sorted_ks.size());
 for (int i = 0; i < int(sorted_ks.size())-1; ++i) {
  ASSERT_LE(sorted_ks[i], sorted_ks[i+1]);
 }
}
// Test 9
TEST(BasicCollectionTest, AssignOpTest) {
 BSTCollection<string,int> c1;
 c1.insert("c", 10);
 c1.insert("sdb", 15);
 c1.insert("d", 20);
 c1.insert("fga", 20);
 c1.insert("kjhc", 10);
 c1.insert("ddb", 15);
 c1.insert("vcd", 20);
 c1.insert("asa", 20);
 BSTCollection<string,int> c2;
 c2.insert("naan", 20);
```

```
 c2.insert("none", 40);
 c2 = c1;
 vector<string> ks2;
 c2.keys(ks2);
 vector<string> ks1;
 c1.keys(ks1);
 for (int j = 0; j < ks2.size(); ++j)
  ASSERT_EQ(ks1[j], ks2[j]);
 ASSERT_EQ(c1.size(), c2.size());
 ASSERT_EQ(c1.height(), c2.height());
}
// Test 10
TEST(BasicCollectionTest, Height) {
 BSTCollection<string,int> d;
 d.insert("c", 10);
 d.insert("b", 15);
 d.insert("d", 20);
 d.insert("a", 20);
 d.insert("z", 10);
 d.insert("f", 15);
 d.insert("e", 20);
 d.insert("g", 20);
 ASSERT_EQ(d.height(), 5);
}
// Test 11
TEST(BasicCollectionTest, CopyList) {
 BSTCollection<string,int> w;
 w.insert("hi", 11);
 w.insert("computer", 12);
 w.insert("forever", 122);
 w.insert("never", 2110);
 w.insert("bye", 11);
 w.insert("laptop", 12);
 w.insert("infinite", 122);
 w.insert("none", 2110);
 BSTCollection<string,int> w_copy = w;
 ASSERT_EQ(w_copy.size(), w.size());

 vector<string> w_keys;
 vector<string> w_copy_keys;
 w.sort(w_keys);
 w_copy.sort(w_copy_keys);
 for (int i = 0; i <= 3; ++i)
 {
  ASSERT_EQ(w_keys[i], w_copy_keys[i]);
 }
 // copy constructor with empty hash table
 BSTCollection<string,int> empty;
 BSTCollection<string,int> empty_copy(empty);
 ASSERT_EQ(empty.size(), 0);
 ASSERT_EQ(empty_copy.size(), 0);
}
// Test 12
TEST(BasicCollectionTest, Assign) {
 BSTCollection<string,int> y;
 y.insert("gofish", 10);
 y.insert("safety", 20);
 y.insert("xyz", 30);
 y.insert("ergo", 40);
 y.insert("aapl", 50);
 y.insert("good", 60);

 BSTCollection<string,int> z;
 z.insert("n", 10);
 z.insert("a", 100);
 z.insert("s", 1000);
 z = y;

 ASSERT_EQ(z.size(), y.size());

 vector<string> z_keys;
 vector<string> y_keys;
 y.keys(y_keys);
 z.keys(z_keys);
 ASSERT_EQ(z.size(), 6);
 for (int i = 0; i <= 5; ++i)
 {
  ASSERT_EQ(y_keys[i], z_keys[i]);
 }
 // assignment operator will not do copying if trying to copy itself.
 z = z;
 ASSERT_EQ(z.size(), 6);
```

```
 ASSERT_EQ(z.height(), y.height());
}
// Test 13
TEST(BasicCollectionTest, Negatives) {
 BSTCollection<double,string>* l = new BSTCollection<double,string>;
 l->insert(999.0, "DigitalLogic");
 l->insert(400.4, "AlgsAndDataStruct");
 l->insert(-33.2, "discreteMath");
 l->insert(-0.1, "Globals");
 l->insert(0.0, "Human Nature");

 vector<double> sorted_ints;
 l->sort(sorted_ints);
 for (int i = 0; i < int(sorted_ints.size()-1); ++i)
 {
  ASSERT_LE(sorted_ints[i], sorted_ints[i+1]);
 }
 delete l;
}
// Test 14
TEST(BasicCollectionTest, SizeZero) {
 BSTCollection<int,int>* m = new BSTCollection<int,int>;
 ASSERT_EQ(m->size(), 0);
 m->remove(2); // remove from BST that is empty
 ASSERT_EQ(m->size(), 0);
 int my_val;
 ASSERT_EQ(m->find(0, my_val), false);
 vector<int> keys_ints;
 m->keys(keys_ints);
 ASSERT_EQ(keys_ints.size(), 0);
 m->sort(keys_ints);
 ASSERT_EQ(keys_ints.size(), 0);
 delete m;
}

int main(int argc, char** argv)
{
 testing::InitGoogleTest(&argc, argv);
 return RUN_ALL_TESTS();
}
```