```
/*Author: Eric Gustin
  Assignment: CPSC223-01 HW05 hw5_tests.cpp
  Description: This program tests the binsearch collection. It tests
  every function in binsearch_collection, and accounts for many edge cases.
*/

#include <iostream>
#include <string>
#include <gtest/gtest.h>
#include "binsearch_collection.h"

using namespace std;

// Test 1
TEST(BasicsListTest, CorrectSize) {
 BinSearchCollection<string,double> a;

 ASSERT_EQ(a.size(), 0);
 a.insert("AAPL", 220.29);
 ASSERT_EQ(a.size(), 1);
 a.insert("AMZN", 1739.81);
 ASSERT_EQ(a.size(), 2);
 a.insert("FB", 179.50);
 ASSERT_EQ(a.size(), 3);
 a.insert("GOOGL", 1240.54);
 ASSERT_EQ(a.size(), 4);
 a.insert("NFLX", 263.86);
 ASSERT_EQ(a.size(), 5);

 // make sure doesn't drop to -1
 BinSearchCollection<string,int> zero_list;
 zero_list.remove("nothing");
 ASSERT_EQ(zero_list.size(), 0);
}

// Test 2
TEST(BasicListTest, InsertAndFind) {
 BinSearchCollection<string,double> b;
 b.insert("e", 50.0);
 b.insert("f", 60.0);
 b.insert("g", 70.0);
 b.insert("a", 10.0);
 b.insert("b", 20.0);
 b.insert("i", 90.0);
 b.insert("j", 100.0);
 b.insert("k", 110.0);
 b.insert("l", 120.0);
 b.insert("c", 30.0);
 b.insert("d", 40.0);
 b.insert("h", 80.0);
 b.insert("m", 130.0);
 b.insert("o", 140.0);
 double my_key1;
 ASSERT_EQ(b.find("a", my_key1), true);
 ASSERT_EQ(10.0, my_key1);
 double my_key2;
 ASSERT_EQ(b.find("o", my_key2), true);
 ASSERT_EQ(140.0, my_key2);
 double my_key3;
 ASSERT_EQ(b.find("f", my_key3), true);
 ASSERT_EQ(60.0, my_key3);
 double my_key4 = 0.0;
 ASSERT_EQ(b.find("z", my_key4), false);
 ASSERT_EQ(0.0, my_key4);
}

 // Test 3
TEST(BasicListTest, RemoveElems) {
 BinSearchCollection<string,double>* c = new BinSearchCollection<string,double>;
 // attempt to remove from an empty object
 c->remove("");
 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);
 c->insert("f", 60.0);

 ASSERT_EQ(c->size(), 6);
 c->remove("q"); // try to remove an element that is not in c. should do nothing
 ASSERT_EQ(c->size(), 6);
 c->remove("a"); // remove first element
 ASSERT_EQ(c->size(), 5);

 double v;
 ASSERT_EQ(c->find("a", v), false);
 ASSERT_EQ(c->find("b", v), true);
 ASSERT_EQ(v, 20.0);
 ASSERT_EQ(c->find("c", v), true);
 ASSERT_EQ(v, 30.0);
 ASSERT_EQ(c->find("d", v), true);
 ASSERT_EQ(v, 40.0);
 ASSERT_EQ(c->find("e", v), true);
 ASSERT_EQ(v, 50.0);
 ASSERT_EQ(c->find("f", v), true);
 ASSERT_EQ(v, 60.0);
 c->remove("f"); // remove last element
 ASSERT_EQ(c->size(), 4);
 vector<string> mine;
 c->keys(mine);
 ASSERT_EQ(c->find("f", v), false);
 c->remove("d"); // remove non edge element
 ASSERT_EQ(c->size(), 3);
 ASSERT_EQ(c->find("d", v), false);
 c->remove("b");
 c->remove("c");
 c->remove("e"); // remove the only element in the list
 ASSERT_EQ(c->size(), 0);
 delete c;
}

 // Test 4
TEST(BasicListTest, GetKeys) {
 BinSearchCollection<string,double>* d = new BinSearchCollection<string,double>;
 d->insert("a", 10.0);
 d->insert("b", 20.0);
 d->insert("c", 30.0);
 vector<string> ks;
 d->keys(ks);
 vector<string>::iterator iter;
 iter = find(ks.begin(), ks.end(), "a");
 ASSERT_NE(iter, ks.end());
 iter = find(ks.begin(), ks.end(), "b");
 ASSERT_NE(iter, ks.end());
 iter = find(ks.begin(), ks.end(), "c");
 ASSERT_NE(iter, ks.end());
 iter = find(ks.begin(), ks.end(), "d");
 ASSERT_EQ(iter, ks.end());
 delete d;
}

 // Test 5
TEST(BasicListTest, GetKeyRange) {
 BinSearchCollection<string,double>* e = new BinSearchCollection<string,double>;
 e->insert("a", 10.0);
 e->insert("b", 20.0);
 e->insert("c", 30.0);
 e->insert("d", 40.0);
 e->insert("e", 50.0);
 vector<string> ks;

 e->find("b", "d", ks);
 vector<string>::iterator iter;
 iter = find(ks.begin(), ks.end(), "b");
 ASSERT_NE(iter, ks.end());
 iter = find(ks.begin(), ks.end(), "c");
```

```
 ASSERT_NE(iter, ks.end());
 iter = find(ks.begin(), ks.end(), "d");
 ASSERT_NE(iter, ks.end());
 iter = find(ks.begin(), ks.end(), "a");
 ASSERT_EQ(iter, ks.end());
 iter = find(ks.begin(), ks.end(), "e");
 ASSERT_EQ(iter, ks.end());
 delete e;
 // test if find range works for strings
 // longer than one character
 BinSearchCollection<string,double>* f = new BinSearchCollection<string,double>;
 f->insert("apples", 10);
 f->insert("golf", 20);
 f->insert("hydro", 30);
 f->insert("zebras", 40);

 vector<string> ks2;
 f->find("bees", "yoyo", ks2);
 vector<string>::iterator iter2;
 iter = find(ks2.begin(), ks2.end(), "golf");
 ASSERT_NE(iter, ks2.end());
 iter = find(ks2.begin(), ks2.end(), "hydro");
 ASSERT_NE(iter, ks2.end());
 iter = find(ks2.begin(), ks2.end(), "apples");
 ASSERT_EQ(iter, ks2.end());
 iter = find(ks2.begin(), ks2.end(), "zebras");
 ASSERT_EQ(iter, ks2.end());
 ASSERT_EQ(ks2.size(), 2);

 // find only 1 key
 vector<string> ks3;
 f->find("zebras", "zebras", ks3);
 ASSERT_EQ(ks3.size(), 1);
 ASSERT_EQ(ks3[0], "zebras");
 // find no keys
 vector<string> ks4;
 f->find("bye", "apple", ks4);
 ASSERT_EQ(ks4.size(), 0);
 delete f;
}

 // Test 6
TEST(BasicListTest, KeySort) {
 BinSearchCollection<string,double>* g = new BinSearchCollection<string,double>;
 g->insert("e", 50.0);
 g->insert("a", 10.0);
 g->insert("d", 40.0);
 g->insert("b", 20.0);
 g->insert("c", 30.0);

 // check to see if it is already sorted (it should be)
 vector<string> pre_sorted_ks;
 for(int i = 0; i < int(pre_sorted_ks.size()) - 1; ++i)
  ASSERT_LE(pre_sorted_ks[i], pre_sorted_ks[i+1]);

 vector<string> sorted_ks;
 g->sort(sorted_ks);
 // check if sort order
 for(int i = 0; i < int(sorted_ks.size()) - 1; ++i)
  ASSERT_LE(sorted_ks[i], sorted_ks[i+1]);
 delete g;

 BinSearchCollection<string,double>* h = new BinSearchCollection<string,double>;
 h->insert("anagrams", 23);
 h->insert("hemmingson", 1);
 h->insert("regis", 99);
 h->insert("salem oregon", 98);
 h->insert("string", 44);
 h->insert("turing", 23);
 vector<string> sorted_stringInt;
 h->sort(sorted_stringInt);
 //check if sort order
 for (int i = 0; i < int(sorted_stringInt.size())-1; ++i)
```

```
  ASSERT_LE(sorted_stringInt[i], sorted_stringInt[i+1]);
 delete h;
}

TEST(BasicListTest, Negatives) {
 BinSearchCollection<double,string>* l = new BinSearchCollection<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(BasicListTest, SizeZero) {
 BinSearchCollection<int,int>* m = new BinSearchCollection<int,int>;
 ASSERT_EQ(m->size(), 0);
 m->remove(2);
 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();
}
```