Kevin Lunden

3-3-20

CPSC223

HW5

My testing strategy involved me testing things on paper, implementing them and then writing the tests that would cause my code to replicate what my work on paper supposedly figured out.

I didn’t face many issues with this assignment. I faced a few bugs; the binary search function took time and my find range function had a simple bug that I only found after adding custom tests. But overall, it went smoothly, and I was able to finish in timely manner.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Rand-10k | Rand-20k | Rand-30k | Rand-40k | Rand-50k |
| Add | 37.7515 ms | 73.7592 ms | 114.593 ms | 150.267 ms | 187.249 ms |
| Remove | 67.585 ms | 141.149 ms | 215.737 ms | 298.438 ms | 377.387 ms |
| Find | .0105 ms | .0195 ms | .0215 ms | .046 ms | .063 ms |
| Range | 88 ms | 86 ms | 90 ms | 90 ms | 111 ms |
| Sort | 478 ms | 1305 ms | 1794 ms | 2728 ms | 3250 ms |

//----------------------------------------------------------------------

// Author: Kevin Lunden

// Course: CPSC 223, Spring 2020

// Assign: 5

// File: binsearch\_collection.h

//

// TODO: More practice with vectors. Practice using binary search.

//----------------------------------------------------------------------

#ifndef BINSEARCH\_COLLECTION\_H

#define BINSEARCH\_COLLECTION\_H

#include <vector>

#include <algorithm>

#include "collection.h"

using namespace std;

template<typename K, typename V>

class BinSearchCollection : public Collection<K,V>

{

public:

// add a new key-value pair into the collection

void add(const K& a\_key, const V& a\_val);

// remove a key-value pair from the collectiona

void remove(const K& a\_key);

// find and return the value associated with the key

bool find(const K& search\_key, V& the\_val) const;

// find and return the values with keys >= to k1 and <= to k2

void find(const K& k1, const K& k2, std::vector<V>& vals) const;

// return all of the keys in the collection

void keys(std::vector<K>& all\_keys) const;

// return all of the keys in ascending (sorted) order

void sort(std::vector<K>& all\_keys\_sorted) const;

// return the number of key-value pairs in the collection

int size() const;

private:

// helper function for binary search

bool binsearch(const K& key, int& index) const;

// vector storage

std::vector<std::pair<K,V>> kv\_list;

};

template <typename K, typename V>

bool BinSearchCollection<K,V>::binsearch(const K& key, int& index) const

{

pair<K,V> p;

int bottom = 0;

int top = kv\_list.size()-1;

int middle;

while (bottom <= top)

{

middle = (top+bottom)/2;

p = kv\_list.at(middle);

if (key == p.first)

{

index = middle;

return true;

}

else if (key > p.first)

bottom = middle+1;

else if (key < p.first)

top = middle-1;

}

if (kv\_list.size() != 0)

index = bottom;

else

index = 0;

return false;

}

// add a new key-value pair into the collection

template <typename K, typename V>

void BinSearchCollection <K,V>::add(const K& a\_key, const V& a\_val)

{

int index;

pair<K,V> tmp;

tmp.first = a\_key;

tmp.second = a\_val;

if (binsearch(a\_key, index) == false)

kv\_list.insert(kv\_list.begin()+index, tmp);

}

// remove a key-value pair from the collectiona

template <typename K, typename V>

void BinSearchCollection <K,V>::remove(const K& a\_key)

{

int index;

if (binsearch(a\_key, index) == true)

kv\_list.erase(kv\_list.begin()+index);

}

// find and return the value associated with the key

template <typename K, typename V>

bool BinSearchCollection <K,V>::find(const K& search\_key, V& the\_val) const

{

int index;

pair<K,V> tmp;

if (binsearch(search\_key, index) == true)

{

tmp = kv\_list.at(index);

the\_val = tmp.second;

return true;

}

return false;

}

// find and return the values with keys >= to k1 and <= to k2

template <typename K, typename V>

void BinSearchCollection <K,V>::find(const K& k1, const K& k2, std::vector<V>& vals) const

{

int index;

binsearch(k1, index);

pair<K,V> tmp = kv\_list[index];

while (tmp.first <= k2 && index < kv\_list.size())

{

vals.push\_back(tmp.second);

index++;

if (index < kv\_list.size())

tmp = kv\_list[index];

}

}

// return all of the keys in the collection

template <typename K, typename V>

void BinSearchCollection <K,V>::keys(std::vector<K>& all\_keys) const

{

if (kv\_list.size() > 0)

{

for (pair<K,V> tmp : kv\_list)

all\_keys.push\_back(tmp.first);

}

}

// return all of the keys in ascending (sorted) order

template <typename K, typename V>

void BinSearchCollection <K,V>::sort(std::vector<K>& all\_keys\_sorted) const

{

this->keys(all\_keys\_sorted);

}

// return the number of key-value pairs in the collection

template <typename K, typename V>

int BinSearchCollection <K,V>::size() const

{

return kv\_list.size();

}

#endif

//----------------------------------------------------------------------

// Author: Kevin Lunden

// Course: CPSC 223, Spring 2020

// Assign: 5

// File: hw5\_test.cpp

//

// TODO: Tests BinSearchCollection.h

//----------------------------------------------------------------------

#include <iostream>

#include <string>

#include <gtest/gtest.h>

#include "binsearch\_collection.h"

using namespace std;

// Test 1

TEST(BasicListTest, CorrectSize) {

BinSearchCollection<string,double> c;

ASSERT\_EQ(0, c.size());

c.add("b", 10.0);

ASSERT\_EQ(1, c.size());

c.add("a", 20.0);

ASSERT\_EQ(2, c.size());

c.add("c", 20.0);

ASSERT\_EQ(3, c.size());

}

// Test 2

TEST(BasicListTest, SimpleFind) {

BinSearchCollection<string,double> c;

double v;

ASSERT\_EQ(false, c.find("b", v));

c.add("b", 10.0);

ASSERT\_EQ(true, c.find("b", v));

ASSERT\_EQ(10.0, v);

ASSERT\_EQ(false, c.find("a", v));

c.add("a", 20.0);

ASSERT\_EQ(true, c.find("a", v));

ASSERT\_EQ(20.0, v);

}

// Test 3

TEST(BasicListTest, SimpleRemoveElems) {

BinSearchCollection<string,int> c;

c.add("b", 10);

c.add("a", 20);

c.add("d", 30);

c.add("c", 30);

ASSERT\_EQ(4, c.size());

int v;

c.remove("a");

ASSERT\_EQ(3, c.size());

ASSERT\_EQ(false, c.find("a", v));

c.remove("b");

ASSERT\_EQ(2, c.size());

ASSERT\_EQ(false, c.find("b", v));

c.remove("c");

ASSERT\_EQ(1, c.size());

ASSERT\_EQ(false, c.find("c", v));

c.remove("d");

ASSERT\_EQ(0, c.size());

ASSERT\_EQ(false, c.find("d", v));

}

// Test 4

TEST(BasicListTest, SimpleRange) {

BinSearchCollection<int,string> c;

c.add(50, "e");

c.add(10, "a");

c.add(30, "c");

c.add(40, "d");

c.add(60, "f");

c.add(20, "b");

vector<string> vs;

c.find(20, 40, vs);

ASSERT\_EQ(3, vs.size());

// note that the following "find" is a C++ built-in function

ASSERT\_EQ(vs.end(), find(vs.begin(), vs.end(), "a"));

ASSERT\_NE(vs.end(), find(vs.begin(), vs.end(), "b"));

ASSERT\_NE(vs.end(), find(vs.begin(), vs.end(), "c"));

ASSERT\_NE(vs.end(), find(vs.begin(), vs.end(), "d"));

ASSERT\_EQ(vs.end(), find(vs.begin(), vs.end(), "e"));

ASSERT\_EQ(vs.end(), find(vs.begin(), vs.end(), "f"));

}

// Test 5

TEST(BasicListTest, SimpleSort) {

BinSearchCollection<string,int> c;

c.add("a", 10);

c.add("e", 50);

c.add("c", 30);

c.add("b", 20);

c.add("d", 40);

vector<string> sorted\_ks;

c.sort(sorted\_ks);

ASSERT\_EQ(5, sorted\_ks.size());

// check if in sorted order

for (int i = 0; i < int(sorted\_ks.size()) -1; ++i)

ASSERT\_LE(sorted\_ks[i], sorted\_ks[i+1]);

}

// Test 6

// Tests more additions

TEST(BasicListTest, CorrectSizeAdvanced) {

BinSearchCollection<string,double> c;

ASSERT\_EQ(0, c.size());

c.add("b", 10.0);

ASSERT\_EQ(1, c.size());

c.add("a", 20.0);

ASSERT\_EQ(2, c.size());

c.add("c", 20.0);

ASSERT\_EQ(3, c.size());

c.add("d", 30.0);

c.add("e", 40.0);

ASSERT\_EQ(5, c.size());

}

// Test 7

// Tests multiple additions with same values

TEST(BasicListTest,AdvancedFind) {

BinSearchCollection<string,double> c;

double v;

ASSERT\_EQ(false, c.find("b", v));

c.add("b", 10.0);

ASSERT\_EQ(true, c.find("b", v));

ASSERT\_EQ(10.0, v);

ASSERT\_EQ(false, c.find("a", v));

c.add("a", 20.0);

ASSERT\_EQ(true, c.find("a", v));

ASSERT\_EQ(20.0, v);

c.add("c", 20.0);

ASSERT\_EQ(true, c.find("c", v));

ASSERT\_EQ(20.0, v);

}

// Test 8

// Tests to see if correct values have been removed

TEST(BasicListTest, AdvancedRemoveElems) {

BinSearchCollection<string,int> c;

c.add("b", 10);

c.add("a", 20);

c.add("d", 30);

c.add("c", 30);

ASSERT\_EQ(4, c.size());

int v;

c.remove("a");

ASSERT\_EQ(3, c.size());

ASSERT\_EQ(true, c.find("b", v));

ASSERT\_EQ(true, c.find("c", v));

ASSERT\_EQ(true, c.find("d", v));

ASSERT\_EQ(false, c.find("a", v));

c.remove("b");

ASSERT\_EQ(2, c.size());

ASSERT\_EQ(false, c.find("b", v));

c.remove("c");

ASSERT\_EQ(1, c.size());

ASSERT\_EQ(false, c.find("c", v));

c.remove("d");

ASSERT\_EQ(0, c.size());

ASSERT\_EQ(false, c.find("d", v));

}

// Test 9

// Tests to make sure range can include entire list

// and include larger numbers

TEST(BasicListTest, AdvancedRange) {

BinSearchCollection<int,string> c;

c.add(50, "e");

c.add(10, "a");

c.add(30, "c");

c.add(40, "d");

c.add(100, "g");

c.add(2000, "h");

c.add(60, "f");

c.add(20, "b");

vector<string> vs;

c.find(20, 40, vs);

ASSERT\_EQ(3, vs.size());

vector<string> vs2;

c.find(10, 2000, vs2);

ASSERT\_EQ(8, vs2.size());

// note that the following "find" is a C++ built-in function

ASSERT\_EQ(vs.end(), find(vs.begin(), vs.end(), "a"));

ASSERT\_NE(vs.end(), find(vs.begin(), vs.end(), "b"));

ASSERT\_NE(vs.end(), find(vs.begin(), vs.end(), "c"));

ASSERT\_NE(vs.end(), find(vs.begin(), vs.end(), "d"));

ASSERT\_EQ(vs.end(), find(vs.begin(), vs.end(), "e"));

ASSERT\_EQ(vs.end(), find(vs.begin(), vs.end(), "f"));

}

// Test 10

// Tests sorting list with larger and negative numbers

TEST(BasicListTest, AdvancedSort) {

BinSearchCollection<string,int> c;

c.add("g", 100);

c.add("b", 10);

c.add("f", 50);

c.add("d", 30);

c.add("a", -100);

c.add("h", 1000);

c.add("c", 20);

c.add("e", 40);

c.add("i", 10000);

vector<string> sorted\_ks;

c.sort(sorted\_ks);

ASSERT\_EQ(9, sorted\_ks.size());

// check if in sorted order

for (int i = 0; i < int(sorted\_ks.size()) -1; ++i)

ASSERT\_LE(sorted\_ks[i], sorted\_ks[i+1]);

}

int main(int argc, char\*\* argv)

{

testing::InitGoogleTest(&argc, argv);

return RUN\_ALL\_TESTS();

}