Kevin Lunden

4-16-20

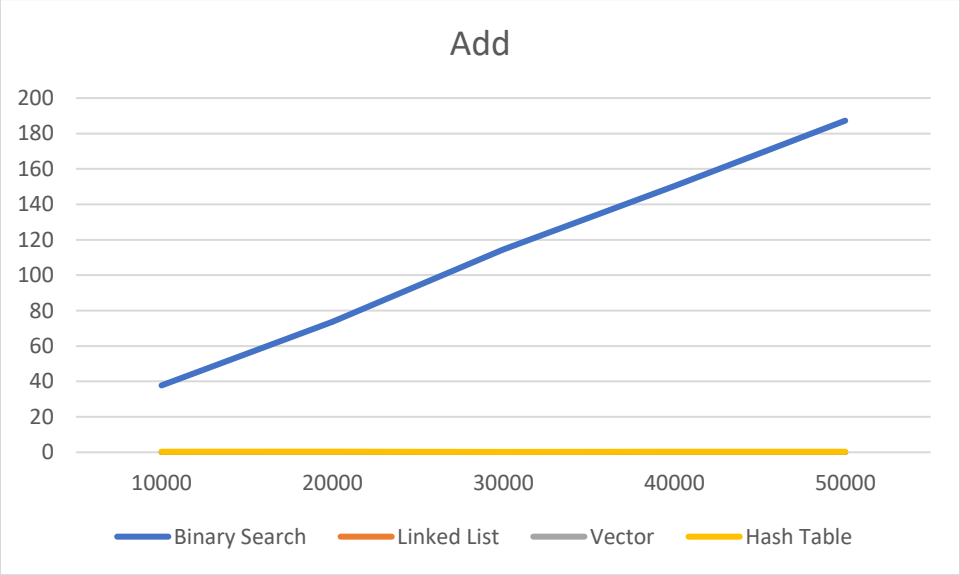
CPSC223

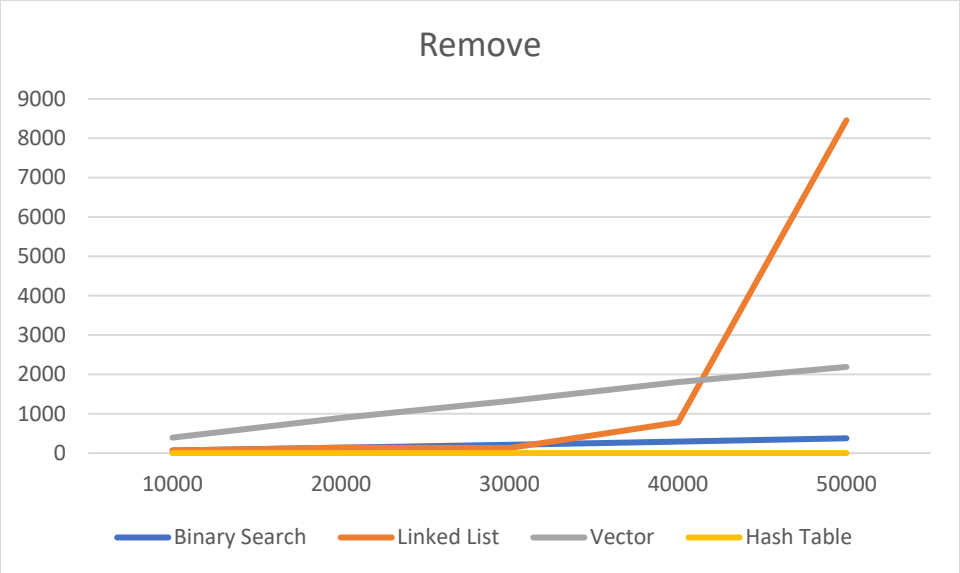
HW8

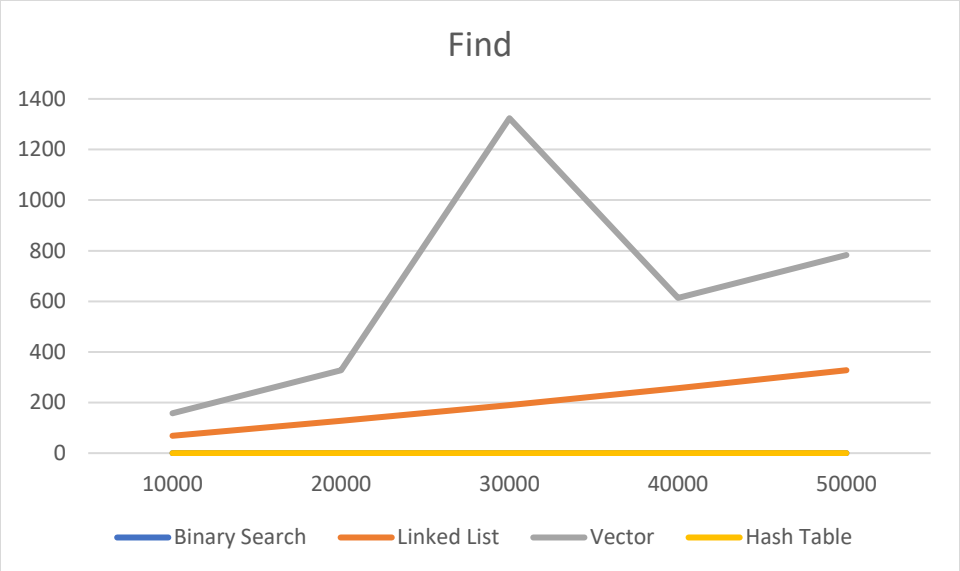
My testing strategy involved me testing things on paper, implementing them and then writing the tests that would cause my code to replicate what my work on paper supposedly figured out.

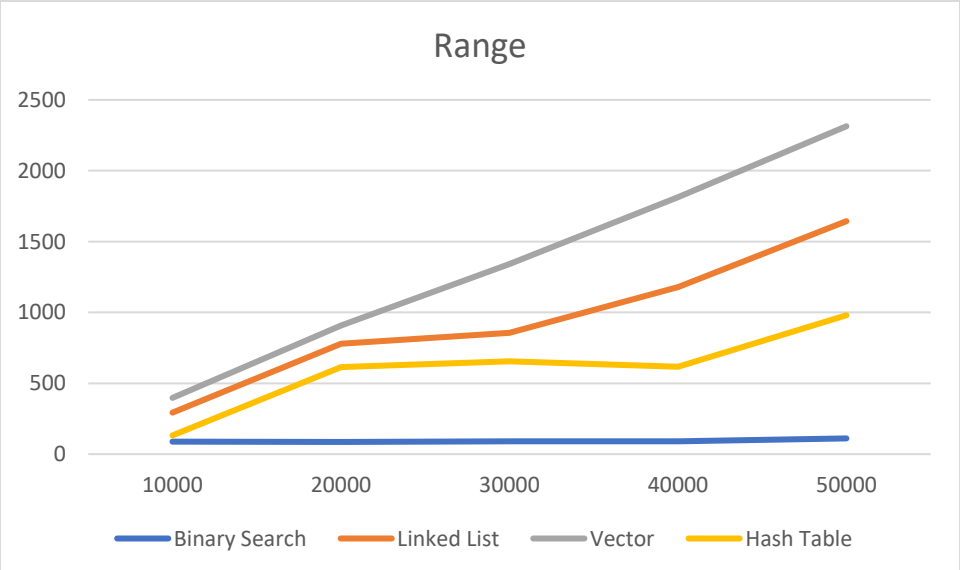
The hash table was difficult to implement. It took a long time to get through the many functions that were required. I think I should try getting individual functions working on their own one at a time but it's very tempting to keep coding multiple functions while I'm in that mindset that I know what I'm doing.

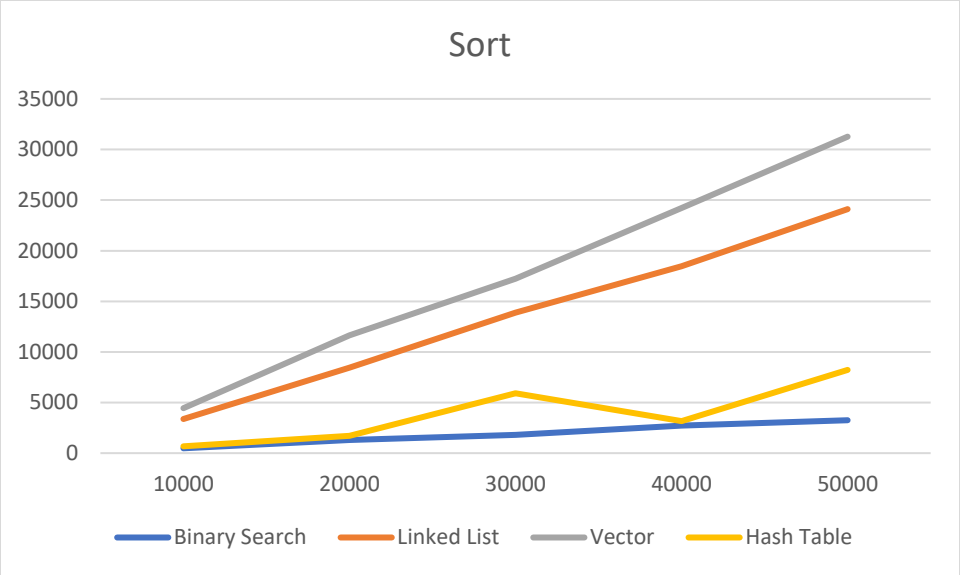
| | Rand-10k | Rand-20k | Rand-30k | Rand-40k | Rand-50k |
|---|---|---|---|---|---|
| Add | 0.3366 ms | 0.3633ms | 0.211ms | 0.31 ms | 0.251 ms |
| Remove | 0.012 ms | 0.016 ms | 0.019 ms | 0.019 ms | 0.018 ms |
| Find | 0.004 ms | 0.009 ms | 0.014 ms | 0.0065 ms | 0.0115 ms |
| Range | 132 ms | 614 ms | 654 ms | 617 ms | 980 ms |
| Sort | 673 ms | 1715 ms | 5924 ms | 3151 ms | 8222 ms |




Remove
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
10000
20000
30000
40000
50000
Binary Search
Linked List
Vector
Hash Table


Find
1400
1200
1000
800
600
400
200
0
10000
20000
30000
40000
50000
Binary Search
Linked List
Vector
Hash Table


Range
2500
2000
1500
1000
500
0
10000
20000
30000
40000
50000
Binary Search
Linked List
Vector
Hash Table




```
//---------------------------------------------------------------------
// Author: Kevin Lunden
// Course: CPSC 223, Spring 2020
// Assign: 8
// File:   hash_table_collection.h
//---------------------------------------------------------------------


#ifndef HASH_TABLE_COLLECTION_H
#define HASH_TABLE_COLLECTION_H

#include <vector>
#include <algorithm>
#include <functional>
#include "collection.h"


template<typename K, typename V>
class HashTableCollection : public Collection<K,V>
{
public:

  // create an empty hash table with default number of buckets
  HashTableCollection();

  // hash table copy constructor
  HashTableCollection (const HashTableCollection <K,V>& rhs);

  // hash table assignment operator
  HashTableCollection <K,V>& operator=(const HashTableCollection <K ,V >& rhs);
```

```
  // delete a linked list
  ~HashTableCollection();

  // 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 collection
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 to empty entire hash table
  void make_empty();

  // helper to resize and rehash the hash table
  void resize_and_rehash();

  // linked list node structure
  struct Node {
    K key;
    V value;
    Node* next;
  };

  // number of k-v pairs in the collection
  int collection_size;

  // number of hash table buckets (default is 16)
  int table_capacity;

  // hash table array load factor (set at 75% for resizing)
double load_factor_threshold;
```

```
  // hash table array
  Node** hash_table;

};

// TODO: implement the above functions here ...
// create an empty linked
template <typename K, typename V>
HashTableCollection <K,V>::HashTableCollection() : collection_size(0), table_capacity(16),
load_factor_threshold(0.75)
{
 // dynamically allocate the hash table array
 hash_table = new Node*[table_capacity];

 for (int i = 0; i < table_capacity; ++i)
  hash_table[i] = nullptr;
}

// copy a linked list
template <typename K, typename V>
HashTableCollection<K,V>::HashTableCollection(const HashTableCollection<K,V>& rhs) :
hash_table(nullptr)
{
 *this = rhs;
}

// assign a linked list
template <typename K, typename V>
HashTableCollection <K,V>& HashTableCollection <K,V>::operator=(const HashTableCollection<K,V>&
rhs)
{
 // check if rhs is current object and return current object
 if (this == &rhs)
  return *this;

 make_empty ();

// initialize current object
 collection_size = 0;
 load_factor_threshold = 0.75;

 hash_table = new Node*[rhs.table_capacity];

 //copy
 Node* ptr;
 for(int i = 0; i < rhs.table_capacity; i++)
 {
  ptr = rhs.hash_table[i];
```

```
  while(ptr != nullptr)
  {
   insert(ptr->key, ptr->value);
   ptr = ptr->next;
  }
 }
 return *this;
}

// delete a linked list
template <typename K, typename V>
HashTableCollection <K,V>::~HashTableCollection()
{
  make_empty();
}

// insert a key-value pair into the collection
template <typename K, typename V>
void HashTableCollection<K,V>::add(const K& key, const V& val)
{
 // check current load factor
 std::hash<K> hash_fun;
 size_t value = hash_fun(key);
 size_t index = value % table_capacity;

 Node* ptr = new Node;
 ptr->key = key;
 ptr->value = val;

//resize if necessary
 if(double(collection_size / table_capacity) > load_factor_threshold)
 {
  resize_and_rehash();
 }

 if(hash_table[index] == nullptr)
 {
  ptr->next = nullptr;

  //set the bucket pointer to the head of new node
  hash_table[index] = ptr;
 }
 else if (hash_table[index] != nullptr)
 {
  ptr->next = hash_table[index];

  //bucket now points to head of new node
  hash_table[index] = ptr;
```

```
 }

 collection_size++;
}

// remove a key-value pair from the collection
template <typename K, typename V>
void HashTableCollection<K,V>::remove(const K& key)
{
 std::hash<K> hash_fun;
 size_t value = hash_fun(key);
 size_t index = value % table_capacity;
 Node* ptr;
 Node* temp;

 //make sure hash table isn't empty
 if(hash_table != nullptr)
 {
  ptr = hash_table[index];
  if(hash_table[index] != nullptr)
{
   //if the first node has the key
   if(ptr->key == key)
   {
    //if there is only one node
    if(ptr->next == nullptr)
    {
     delete ptr;
     hash_table[index] = nullptr;
     collection_size--;
    }
    else
    {
     hash_table[index] = ptr->next;
     ptr->next = nullptr;
     delete ptr;
     collection_size--;
    }
   }
   else
   {
    temp = ptr;
    if(ptr != nullptr)
    {
     ptr = ptr -> next;

     //while not at the last node of the bucket
     while(ptr != nullptr)
```

```
    {
     if(ptr->key == key)
     {
      if(ptr->next = nullptr)
      {
       temp->next = nullptr;
       delete ptr;
       collection_size--;
      }
      else
      {
       temp->next = ptr->next;
       ptr->next = nullptr;
       delete ptr;
       collection_size--;
      }
     }
     temp = ptr;
     ptr = ptr->next;
    }
   }
  }
 }
}

// find the value associated with the key
template <typename K, typename V>
bool HashTableCollection<K,V>::find(const K& key, V& val) const
{
 Node* temp;
 std::hash<K> hash_fun;
 size_t value = hash_fun(key);
 size_t index = value % table_capacity;

 if(hash_table != nullptr)
 {
  if(hash_table[index] == nullptr)
  {
   return false;
  }
  else
  {
   temp = hash_table[index];

   while(temp != nullptr)
   {
    if(temp->key == key)
```

```
   {
    val = temp -> value;
    return true;
   }
   else
   {
    temp = temp->next;
   }
  }
 }
}
 return false;
}

// find and return the values with keys >= to k1 and <= to k2
template <typename K, typename V>
void HashTableCollection<K,V>::find(const K& k1, const K& k2, std::vector<V>& vals) const
{
 Node* ptr;

 for(int i = 0; i < table_capacity; i++)
 {
  ptr = hash_table[i];

  while(ptr != nullptr)
  {
   if(ptr->key >= k1 && ptr->key <= k2)
   {
    vals.push_back(ptr->value);
   }

   ptr = ptr->next;
  }
 }
}

// return all keys in the collection
template <typename K, typename V>
void HashTableCollection<K,V>::keys(std::vector <K>& keys) const
{
 Node* ptr;

for(int i = 0; i < table_capacity; i++)
 {
  ptr = hash_table[i];

  while(ptr != nullptr)
  {
```

```
   keys.push_back(ptr->key);
   ptr = ptr->next;
  }
 }
}

// return collection keys in sorted order
template<typename K,typename V>
void HashTableCollection<K,V>::sort(std::vector<K>& ks) const
{
 keys(ks);
 std::sort(ks.begin(), ks.end());
}

// return the number of keys in collection
template <typename K, typename V>
int HashTableCollection <K,V>::size() const
{
 return collection_size;
}

// helper to empty entire hash table
template <typename K, typename V>
void HashTableCollection <K,V>::make_empty()
{
 Node* ptr;
 Node* next;

 if(hash_table != nullptr)
 {
  for(int i = 0; i < collection_size; i++)
  {
   ptr = hash_table[i];

  //iterate through entire linked list in spot of hash table_capacity
   while(hash_table[i] != nullptr)
   {
    next = ptr->next;
    delete ptr;
    ptr = next;
    hash_table[i] = ptr;
   }
  }
 }

 delete hash_table;
}
```

```
template <typename K, typename V>
void HashTableCollection<K,V>::resize_and_rehash()
{
 int new_capacity = table_capacity * 2;

 Node** new_table = new Node*[new_capacity];

 // initialize new table
 for(int i = 0; i < new_capacity; i++)
  new_table[i] = nullptr;

 // insert key values
 std::vector <K> ks;
 keys(ks);
 size_t index;
 V inVal;

 for(K key:ks)
 {
  std::hash<K> hash_fun;
  size_t value = hash_fun(key);
  size_t index = value % table_capacity;

  Node* newNode = new Node;
  bool temp = find(key, inVal);

newNode->key = key;
  newNode->value = inVal;

  if(new_table[index] == nullptr)
  {
   hash_table[index] = newNode;
   newNode->next = nullptr;
  }
  else
  {
   newNode->next = new_table[index];
   new_table[index] = newNode;
  }
 }

 make_empty();
 hash_table = new_table;
 table_capacity = new_capacity;
}

#endif
```

```
//---------------------------------------------------------------------
// Author: Kevin Lunden
// Course: CPSC 223, Spring 2020
// Assign: 8
// File:   hw8_test.cpp
//
// TODO: Tests functions of hash table collection.
//---------------------------------------------------------------------


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

using namespace std;

// Test 1
TEST(BasicListTest, CorrectSize) {
  HashTableCollection<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
// Tests multiple adds with negative and large numbers
// Also tests adding same keys
TEST(BasicListTest, DiffCorrectSize) {
  HashTableCollection<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);
  c.add("d", 1000.0);
  c.add("d", -10.0);
  ASSERT_EQ(5, c.size());
}


// Test 3
TEST(BasicListTest, SimpleFind) {
```

```
  HashTableCollection<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 4
TEST(BasicListTest, SimpleRemoveElems) {
  HashTableCollection<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("c", v));
}

// Test 5
TEST(BasicListTest, SimpleRange) {
  HashTableCollection<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 6
// Tests if entire range works
TEST(BasicListTest, DiffRange) {
  HashTableCollection<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(10, 60, vs);
  ASSERT_EQ(6, vs.size());
 // note that the following "find" is a C++ built-in function
  ASSERT_NE(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_NE(vs.end(), find(vs.begin(), vs.end(), "e"));
  ASSERT_NE(vs.end(), find(vs.begin(), vs.end(), "f"));
}

// Test 7
TEST(BasicListTest, SimpleSort) {
  HashTableCollection<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 8
// Test with negative and large numbers
```

```
TEST(BasicListTest, DiffSort) {
  HashTableCollection<string,int> c;
  c.add("a", -10);
  c.add("e", 5000);
  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 9
// Test with multi-letter strings
TEST(BasicCollectionTest, SimpleEqual){
  HashTableCollection <string, double> c;
  c.add("de", 40.0); //make sure remove works on 2 char strings
  c.add("t", 67.0);
  double v;
  c.remove("de");
  ASSERT_EQ(c.find("de", v), false);
  c.remove("i"); //make sure removing a string that doesnt exist works
  ASSERT_EQ(c.find("t", v), true);
  c.remove("t");
  c.remove("f");
  ASSERT_EQ(c.size(), 0); //make sure remove works when empty
}

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