Carter Mooring HW8

My testing strategy consisted of me starting by coding insert first and testing. The issue I ran into first was that to test search, you also have to use the find function to find it which is always hard for me because it is hard for me to continue coding when I am unsure if a large part of mine is working. I supposed my testing strategy could be to somehow test if insert works without needing find.

One of the main issues I was running into first was that I was getting segfaults and was unsure why. I addressed this by commenting a lot of my code to narrow down where it was happening which helped me out a lot. It turned out I was forgetting to set my nodes left and right to nullptr. Another thing that I was struggling with was I am usure why we use helper functions for functions where all we do is call a helper function?

```
---------------------------------------------bts_collection.h-------------------------------------------------------
#ifndef BST_COLLECTION_H
#define BST_COLLECTION_H
#include <vector>
#include "collection.h"

using namespace std;

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

  // create an empty BST
  BSTCollection();

  // copy a BST
  BSTCollection(const BSTCollection<K, V>&);

  // assigne a BST
  BSTCollection<K, V>& operator=(const BSTCollection<K,V>& rhs);

  // delete a BST
  ~BSTCollection();

  // insert a KV-pair from the Collection
  void insert(const K& key, const V& val);

  // remove a KV-pair from the Collection
  void remove(const K& key);
  // find the value associate with the key
  bool find(const K& key, V& val) const;

  // find the keys associated with the range
  void find(const K& k1, const K& k2, std::vector<K>& keys) const;

  // return all keys in the collection;
  void keys(std::vector<K>& keys) const;

  // return collection keys in sorted order
  void sort(std::vector<K>& keys) const;

  // return the number of keys in collection
  int size() const;

  // return the height of the tree
  int height() const;

private:

  // binary search tree node structures
  struct Node {
    K key;
    V value;
    Node* left;
    Node* right;
  };

  // root node of the search tree
  Node* root;

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

  // helper to recursively empty search tree
  void make_empty(Node* subtree_root);

  // helper to recursively build sorted list of keys
  void inorder(const Node* subtree, std::vector<K>& keys) const;

  // helper to recursively build sorted list of keys
  void preorder(const Node* subtree, std::vector<K>& keys) const;
  // helper to recursively find range of keys
  void range_search(const Node* subtree, const K& k1,
                    const K& k2, std::vector<K>& keys) const;

  // return the height of the tree rooted at subtree_root
  int height(const Node* subtree_root) const;
};

template<typename K, typename V>
BSTCollection<K,V>::BSTCollection() : collection_size(0), root(nullptr){}

template<typename K, typename V>
BSTCollection<K,V>::BSTCollection(const BSTCollection<K,V>& rhs)
  : collection_size(0), root(nullptr)
{
  *this = rhs;
}

template<typename K, typename V>
BSTCollection<K,V>& BSTCollection<K,V>::operator=(const BSTCollection<K,V>& rhs){
  if (this == &rhs) {
    return *this;
  }
  // delete current
  make_empty(root);
  // build tree
  std::vector<K> ks;
  preorder(rhs.root, ks);
  int val = 0;
  for(int i = 0; i < rhs.collection_size; i++){
    rhs.find(ks[i], val);
    insert(ks[i], val);
  }
  return *this;
}

template<typename K, typename V>
BSTCollection<K,V>::~BSTCollection()
{
  make_empty(root);
}

template<typename K, typename V>
```

```
void BSTCollection<K,V>::insert(const K& key, const V& val)
{
  Node* tree = new Node;
  tree->key = key;
  tree->value = val;
  tree->left = nullptr;
  tree->right = nullptr;

  if(root == nullptr){
    root = tree;
    tree->left = nullptr;
    tree->right = nullptr;
  }else{
    Node* cur = new Node;
    cur = root;
    while (cur != nullptr)
     if(tree->key < cur->key){
        if(cur->left == nullptr){
          cur->left = tree;
          cur = nullptr;
        }else{
          cur = cur->left;
        }
     }else{
        if(cur->right == nullptr){
          cur->right = tree;
          cur = nullptr;
        }else{
          cur = cur->right;
          tree->left = nullptr;
          tree->right = nullptr;
      }
    }
 }
 collection_size++;
}

template<typename K, typename V>
void BSTCollection<K,V>::remove(const K& key)
{
  // leave empty for HW9
}
```

```
template<typename K, typename V>
bool BSTCollection<K,V>::find(const K& key, V& val) const{
  Node* tree = new Node;
  Node* cur = root;
  tree->key = key;
  tree->value = val;
  tree->left = nullptr;
  tree->right = nullptr;
  bool found = false;
  int size = collection_size;

  while(!found && size !=0){
    if(tree->key == cur->key){
      val = cur->value;
      return true;
    }else if(tree->key > cur->key){
      cur = cur->right;
    }else if(tree->key < cur->key){
      cur = cur->left;
    }
    size--;
  }
  return false;
}

template<typename K, typename V>
void BSTCollection<K,V>::find(const K& k1, const K& k2,std::vector<K>& ks) const{
  // defer to the range search (recursive) helper function
  range_search(root, k1, k2, ks);
}

template<typename K, typename V>
void BSTCollection<K,V>::range_search(const Node* subtree, const K& k1, const K& k2,
std::vector<K>& ks)
const {
  // use as recursive helper function
  if(subtree == nullptr){
    return;
  }
  range_search(subtree->left, k1, k2, ks);

  if(k1 <= subtree->key && k2 >= subtree->key){
    ks.push_back(subtree->key);
  }
  range_search(subtree->right, k1, k2, ks);
}

template<typename K, typename V>
void BSTCollection<K,V>::keys(std::vector<K>& ks) const
{
  // defer to the inorder (recursive) helper function
  inorder(root, ks);
}

template<typename K, typename V>
void BSTCollection<K,V>::sort(std::vector<K>& ks) const
{
  // defer to the inorder (recursive) helper function
  inorder(root, ks);
}

template<typename K, typename V>
int BSTCollection<K,V>::size() const
{
  return collection_size;
}

template<typename K, typename V>
int BSTCollection<K,V>::height(const Node* subtree_root) const{
  // recursive helper
  if(subtree_root == nullptr){
    return 0;
  }

  int left = height(subtree_root->left);
  int right = height(subtree_root->right);

  if(left > right){
    return(left+1);
  }else{
    return(right+1);
  }
}

template<typename K, typename V>
void BSTCollection<K,V>::make_empty(Node* subtree_root) {
  if(subtree_root == nullptr){
    return;
  }

  make_empty(subtree_root->left);
  make_empty(subtree_root->right);
  subtree_root == nullptr;

}

template<typename K, typename V>
void BSTCollection<K,V>::inorder(const Node* subtree, std::vector<K>& ks) const
{
  // recursive helper function
  if(subtree == nullptr){
    return;
  }
  inorder(subtree->left, ks);
  ks.push_back(subtree->key);
  inorder(subtree->right, ks);
}

template<typename K, typename V>
void BSTCollection<K,V>::preorder(const Node* subtree, std::vector<K>& ks) const
{
  // recursive helper function
  if(subtree == nullptr){
    return;
  }
  ks.push_back(subtree->key);
  inorder(subtree->left, ks);
  inorder(subtree->right, ks);
}

template<typename K, typename V>
int BSTCollection<K,V>::height() const
{
  // defer to the height (recursive) helper function
  return height(root);
}

#endif
```

```
-----------------------------------------------hw8_tests.cpp--------------------------------------------
#include <iostream>
#include <string>
#include <algorithm>
#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 -- should fail for hw8
TEST(BasicCollectionTest, RemoveElems) {
  BSTCollection<string,double> c;
  c.insert("a", 10.0);
  c.insert("b", 20.0);
  c.insert("c", 30.0);
  double v;
  //c.remove("a");
  //ASSERT_EQ(false, c.find("a", v));
  ASSERT_EQ(true, c.find("b", v));
  ASSERT_EQ(true, c.find("c", v));
  //c.remove("b");
  //ASSERT_EQ(false, c.find("b", v));
  //ASSERT_EQ(true, c.find("c", v));
  //c.remove("c");
  //ASSERT_EQ(false, c.find("c", v));
  //ASSERT_EQ(0, c.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, 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 6
TEST(BasicCollectionTest, KeySort) {
  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> 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 7
TEST(BasicCollectionTest, AssignOpTest) {
  BSTCollection<string,int> c1;
  c1.insert("c", 10);
  c1.insert("b", 15);
  c1.insert("d", 20);
  c1.insert("a", 20);
  BSTCollection<string, int> c2;
  c2 = c1;
  ASSERT_EQ(c1.size(), c2.size());
  ASSERT_EQ(c1.height(), c2.height());
}

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