1 Reading

Read and complete all in-text exercises for the following chapters:

- Ch 11 (Week #9 and #10)

2 Goals

- Practice implementing binary search trees in `C++`;
- Practice navigating / traversing binary trees;
- More recursion and pointer practice.

This assignment, like HW7, requires using CMake and make with Google Test.

3 Instructions

1. Your primary task is to implement a new BSTCollection implementation of the abstract Collection class. For this assignment, you will **not** implement the `remove` function. But, you will implement all other Collection member functions.
2. You will have almost identical files as for HW7, except instead of `hash_table_collection.h` you will have `bs_collection.h`, and instead of `hw7_tests.cpp` you will have `hw8_tests.cpp`. You will not do any performance testing (since we won't implement all Collection functions).
3. As with prior assignments, carefully consider the additional test cases you will need to write for `hw8_tests.cpp`.
4. Hand in a hard-copy printout of your source code, with a cover sheet. Be sure to *carefully* read over and follow all guidelines outlined in the cover sheet. Your hard-copy should be stapled and turned in during class on the due date. Include the table and graphs as part of the hard-copy.
5. Submit your source code using the `dropoff` command on `ada`. Your source code must be submitted by class on the due date. You only need to submit the code needed to build, compile, and run your programs.

Additional Information for HW8

- You must strictly adhere to the class specification below. In particular, you cannot implement any additional helper functions.
- Your `insert` and single-key `find` functions must be implemented using looping (i.e., without recursion). However, your range `find`, `keys`, and `sort` functions must be implemented using recursion (using the corresponding helper functions). In addition, the `make_empty` function, the `height` helper function, and your assignment operator must also be recursive (again, using the prescribed helpers below).
- You must expand on the test cases provided below. Note that these are just a start and will not guarantee that your implementation is correct!

4 Code Listings

Listing 1: `bst_collection.h`

```
1  #ifndef BST_COLLECTION_H
2  #define BST_COLLECTION_H
3
4  #include <vector>
5  #include "collection.h"
6
7
8  template<typename K, typename V>
9  class BSTCollection : public Collection<K,V>
10 {
11 public:
12
13   // create an empty linked list
14   BSTCollection();
15
16   // copy a linked list
17   BSTCollection(const BSTCollection<K,V>& rhs);
18
19   // assign a linked list
20   BSTCollection<K,V>& operator=(const BSTCollection<K,V>& rhs);
21
22   // delete a linked list
23   ~BSTCollection();
24
25   // insert a key-value pair into the collection
26   void insert(const K& key, const V& val);
27
28   // remove a key-value pair from the collection
29   void remove(const K& key);
30
31   // find the value associated with the key
32   bool find(const K& key, V& val) const;
33
```

```
34    // find the keys associated with the range
35    void find(const K& k1, const K& k2, std::vector<K>& keys) const;
36
37    // return all keys in the collection
38    void keys(std::vector<K>& keys) const;
39
40    // return collection keys in sorted order
41    void sort(std::vector<K>& keys) const;
42
43    // return the number of keys in collection
44    int size() const;
45
46    // return the height of the tree
47    int height() const;
48
49  private:
50
51    // binary search tree node structure
52    struct Node {
53      K key;
54      V value;
55      Node* left;
56      Node* right;
57    };
58
59    // root node of the search tree
60    Node* root;
61
62    // number of k-v pairs in the collection
63    int collection_size;
64
65    // helper to recursively empty search tree
66    void make_empty(Node* subtree_root);
67
68    // helper to recursively build sorted list of keys
69    void inorder(const Node* subtree, std::vector<K>& keys) const;
70
71    // helper to recursively build sorted list of keys
72    void preorder(const Node* subtree, std::vector<K>& keys) const;
73
74    // helper to recursively find range of keys
75    void range_search(const Node* subtree, const K& k1, const K& k2,
76                      std::vector<K>& keys) const;
77
78    // return the height of the tree rooted at subtree_root
79    int height(const Node* subtree_root) const;
80
81  };
82
83
84  template<typename K, typename V>
85  BSTCollection<K,V>::BSTCollection() :
86    collection_size(0), root(nullptr)
```

```
87  {
88  }
89
90
91  template<typename K, typename V>
92  void BSTCollection<K,V>::make_empty(Node* subtree_root)
93  {
94    // ... TODO ...
95    // this is a recursive helper function
96  }
97
98
99  template<typename K, typename V>
100 BSTCollection<K,V>::~BSTCollection()
101 {
102   make_empty(root);
103 }
104
105
106 template<typename K, typename V>
107 BSTCollection<K,V>::BSTCollection(const BSTCollection<K,V>& rhs)
108   : collection_size(0), root(nullptr)
109 {
110   *this = rhs;
111 }
112
113
114 template<typename K, typename V>
115 BSTCollection<K,V>& BSTCollection<K,V>::operator=(const BSTCollection<K,V>& rhs)
116 {
117   if (this == &rhs)
118     return *this;
119   // delete current
120   make_empty(root);
121   // build tree
122   std::vector<K> ks;
123   preorder(rhs.root, ks);
124   // ... TODO ...
125   return *this;
126 }
127
128
129 template<typename K, typename V>
130 void BSTCollection<K,V>::insert(const K& key, const V& val)
131 {
132   // ... TODO ...
133   // NOTE: You cannot use recursion or any helpers for insert
134 }
135
136
137 template<typename K, typename V>
138 void BSTCollection<K,V>::remove(const K& key)
139 {
```

```
140   // ... Leave empty for now ...
141   // ... SAVE FOR HW 9 ...
142 }
143
144
145 template<typename K, typename V>
146 bool BSTCollection<K,V>::find(const K& key, V& val) const
147 {
148   // ... TODO ...
149   // NOTE: You cannot use recursion or any helpers for find
150 }
151
152
153 template<typename K, typename V> void
154 BSTCollection<K,V>::
155 range_search(const Node* subtree, const K& k1, const K& k2, std::vector<K>& ks) const
156 {
157   // ... TODO ...
158   // this is a recursive helper function
159 }
160
161
162 template<typename K, typename V> void
163 BSTCollection<K,V>::find(const K& k1, const K& k2, std::vector<K>& ks) const
164 {
165   // defer to the range search (recursive) helper function
166   range_search(root, k1, k2, ks);
167 }
168
169
170 template<typename K, typename V>
171 void BSTCollection<K,V>::inorder(const Node* subtree, std::vector<K>& ks) const
172 {
173   // ... TODO ...
174   // this is a recursive helper function
175 }
176
177
178 template<typename K, typename V>
179 void BSTCollection<K,V>::preorder(const Node* subtree, std::vector<K>& ks) const
180 {
181   // ... TODO ...
182   // this is a recursive helper function
183 }
184
185
186 template<typename K, typename V>
187 void BSTCollection<K,V>::keys(std::vector<K>& ks) const
188 {
189   // defer to the inorder (recursive) helper function
190   inorder(root, ks);
191 }
192
```

```
193
194 template<typename K, typename V>
195 void BSTCollection<K,V>::sort(std::vector<K>& ks) const
196 {
197   // defer to the inorder (recursive) helper function
198   inorder(root, ks);
199 }
200
201
202 template<typename K, typename V>
203 int BSTCollection<K,V>::size() const
204 {
205   return collection_size;
206 }
207
208
209 template<typename K, typename V>
210 int BSTCollection<K,V>::height(const Node* subtree_root) const
211 {
212   // ... TODO ...
213   // this is a recursive helper function
214 }
215
216
217 template<typename K, typename V>
218 int BSTCollection<K,V>::height() const
219 {
220   // defer to the height (recursive) helper function
221   return height(root);
222 }
223
224 #endif
```

Listing 2: hw8_tests.cpp

```
#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 -- THIS TEST 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
```

```
53  TEST(BasicCollectionTest, GetKeys) {
54    BSTCollection<string,double> c;
55    c.insert("a", 10.0);
56    c.insert("b", 20.0);
57    c.insert("c", 30.0);
58    vector<string> ks;
59    c.keys(ks);
60    vector<string>::iterator iter;
61    iter = find(ks.begin(), ks.end(), "a");
62    ASSERT_NE(ks.end(), iter);
63    iter = find(ks.begin(), ks.end(), "b");
64    ASSERT_NE(ks.end(), iter);
65    iter = find(ks.begin(), ks.end(), "c");
66    ASSERT_NE(ks.end(), iter);
67    iter = find(ks.begin(), ks.end(), "d");
68    ASSERT_EQ(ks.end(), iter);
69  }
70
71  // Test 5
72  TEST(BasicCollectionTest, GetKeyRange) {
73    BSTCollection<string,double> c;
74    c.insert("a", 10.0);
75    c.insert("b", 20.0);
76    c.insert("c", 30.0);
77    c.insert("d", 40.0);
78    c.insert("e", 50.0);
79    vector<string> ks;
80    c.find("b", "d", ks);
81    vector<string>::iterator iter;
82    iter = find(ks.begin(), ks.end(), "b");
83    ASSERT_NE(ks.end(), iter);
84    iter = find(ks.begin(), ks.end(), "c");
85    ASSERT_NE(ks.end(), iter);
86    iter = find(ks.begin(), ks.end(), "d");
87    ASSERT_NE(ks.end(), iter);
88    iter = find(ks.begin(), ks.end(), "a");
89    ASSERT_EQ(ks.end(), iter);
90    iter = find(ks.begin(), ks.end(), "e");
91    ASSERT_EQ(ks.end(), iter);
92  }
93
94  // Test 6
95  TEST(BasicCollectionTest, KeySort) {
96    BSTCollection<string,double> c;
97    c.insert("a", 10.0);
98    c.insert("e", 50.0);
99    c.insert("c", 30.0);
100   c.insert("b", 20.0);
101   c.insert("d", 40.0);
102   vector<string> sorted_ks;
103   c.sort(sorted_ks);
104   ASSERT_EQ(c.size(), sorted_ks.size());
105   for (int i = 0; i < int(sorted_ks.size()) - 1; ++i) {
```

```
106       ASSERT_LE(sorted_ks[i], sorted_ks[i+1]);
107     }
108   }
109
110   // Test 7
111   TEST(TreeCollectionTest, AssignOpTest) {
112     BSTCollection<string,int> c1;
113     c1.insert("c", 10);
114     c1.insert("b", 15);
115     c1.insert("d", 20);
116     c1.insert("a", 20);
117     BSTCollection<string,int> c2;
118     c2 = c1;
119     ASSERT_EQ(c1.size(), c2.size());
120     ASSERT_EQ(c1.height(), c2.height());
121   }
122
123
124   int main(int argc, char** argv)
125   {
126     testing::InitGoogleTest(&argc, argv);
127     return RUN_ALL_TESTS();
128   }
```