1 Reading

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

- Ch 12 (Week #12 and #13)

2 Goals

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

This assignment requires using CMake and make with Google Test.

3 Instructions

1. Your primary task is to finish the red-black tree (RBTCollection) implementation of the abstract Collection class. For this assignment, you will only need to implement the `remove` function (which is very tricky).
2. You will have almost identical files as for HW10, except instead of instead of `hw10_tests.cpp` you will have `hw11_tests.cpp`.
3. As with prior assignments, carefully consider the additional test cases you will need to write for `hw11_tests.cpp`. You should have multiple tests to make sure that your tree rotations are working properly (and consider the various cases). You can use the `height` function to help in your tests. I also found it useful to write a pre-order `print` function to see each node's value and color.
4. Like for the other collection implementations, you must run your implementation through the performance test code. As in prior assignments, you must:
   (a). Run your program at least three times for each of the five test files and record the results. (Note that you must run each of the test files the same number of times.)
   (b). Using the run results, create an overall average for each of the three runs, for each operation and test file.
   (b). Create a table of the results. Your table should be formatted similarly to the following (yet to be filled in) table.

	`rand-10k`	`rand-20k`	`rand-30k`	`rand-40k`	`rand-50k`
Insert Average					
Remove Average					
Find Average					
Range Average					
Sort Average					

5. Create graphs showing the performance of your implementation compared to your previous Collection implementations (vector, linked list, vector-based binary search, hash table, and binary search tree based). Again, note that to make the comparison "fair" you will need to ensure you run the tests on the same machine as the previous results, or better, rerun the tests for for these again as you do the tests for HW11 (especially if your prior results have been marked as being unusual, unexpected, or off).

6. 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.

7. 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 HW11

- You must strictly adhere to the class specification below. In particular, you cannot implement any additional helper functions (except for `print` as stated above, or other functions that will help you test your code).
- You must implement your remove function according to the approached described in class. This means implementing the recursive `remove` helper function:

  ```
  Node* remove(const K& key, Node* parent, Node* subtree_root, bool& found)
  ```

  See below for additional details of the function.
- You will also implement a remove helper function specifically for doing the rebalancing step on backtracking. The function has the signature:

  ```
  Node* remove_color_adjust(Node* parent)
  ```

  Again, see below for additional details.
- The double-black coloring of nodes will be stored as Boolean flags `is_dbl_black_left` and `is_dbl_black_right` in the `Node` structure.
- Most of your functions from HW10 can remain the same for HW11. Note, however, that because of the two additional attributes of the `Node` struct, you will need to set these to false wherever you initialize the `Node` objects. For example, within the assignment operator, the insert helper, and possibly within your copy operation (if you do a recursive copy of the tree).

4 Code Listings

Listing 1: `rbt_collection.h`

```
1  #ifndef RBT_COLLECTION_H
2  #define RBT_COLLECTION_H
3
4  #include <vector>
5  #include "collection.h"
6
7  template<typename K, typename V>
8  class RBTCollection : public Collection<K,V>
9  {
10 public:
11
12   // create an empty linked list
13   RBTCollection();
14
15   // copy a linked list
16   RBTCollection(const RBTCollection<K,V>& rhs);
17
18   // assign a linked list
19   RBTCollection<K,V>& operator=(const RBTCollection<K,V>& rhs);
```

```
20
21    // delete a linked list
22    ~RBTCollection();
23
24    // insert a key-value pair into the collection
25    void insert(const K& key, const V& val);
26
27    // remove a key-value pair from the collection
28    void remove(const K& key);
29
30    // find the value associated with the key
31    bool find(const K& key, V& val) const;
32
33    // find the keys associated with the range
34    void find(const K& k1, const K& k2, std::vector<K>& keys) const;
35
36    // return all keys in the collection
37    void keys(std::vector<K>& keys) const;
38
39    // return collection keys in sorted order
40    void sort(std::vector<K>& keys) const;
41
42    // return the number of keys in collection
43    int size() const;
44
45    // return the height of the tree
46    int height() const;
47
48    // optional print function (for testing)
49    void print() const;
50
51  private:
52
53    // binary search tree node structure
54    struct Node {
55      K key;
56      V value;
57      Node* left;
58      Node* right;
59      bool is_black;              // true if black, false if red
60      bool is_dbl_black_left;     // for remove rotations
61      bool is_dbl_black_right;    // for remove rotations
62    };
63
64    // root node of the search tree
65    Node* root;
66
67    // number of k-v pairs in the collection
68    int collection_size;
69
70    // helper to recursively empty search tree
71    void make_empty(Node* subtree_root);
72
```

```
73    // recursive helper to remove node with given key
74    Node* remove(const K& key, Node* parent, Node* subtree_root, bool& found);
75
76    // helper to perform a single rebalance step on a red-black tree on remove
77    Node* remove_color_adjust(Node* parent);
78
79    // recursive helper to do red-black insert key-val pair (backtracking)
80    Node* insert(const K& key, const V& val, Node* subtree_root);
81
82    // helper functions to perform a single right rotation
83    Node* rotate_right(Node* k2);
84
85    // helper functions to perform a single left rotation
86    Node* rotate_left(Node* k2);
87
88    // helper to recursively build sorted list of keys
89    void inorder(const Node* subtree, std::vector<K>& keys) const;
90
91    // helper to recursively build sorted list of keys
92    void preorder(const Node* subtree, std::vector<K>& keys) const;
93
94    // helper to recursively find range of keys
95    void range_search(const Node* subtree_root, const K& k1, const K& k2,
96                      std::vector<K>& keys) const;
97
98    // return the height of the tree rooted at subtree_root
99    int height(const Node* subtree_root) const;
100
101 };
102
103 ...
104
105 template<typename K, typename V>
106 void RBTCollection<K,V>::remove(const K& key)
107 {
108   // check if anything to remove
109   if (root == nullptr)
110     return;
111   // create a ``fake'' root to pass in as parent of root
112   Node* root_parent = new Node;
113   root_parent->key = root->key;
114   root_parent->left = nullptr;
115   root_parent->right = root;
116   root_parent->is_black = true;
117   root_parent->is_dbl_black_left = false;
118   root_parent->is_dbl_black_right = false;
119   // call remove
120   bool found = false;
121   root_parent = remove(key, root_parent, root, found);
122   // update results
123   if (found) {
124     collection_size--;
125     root = root_parent->right;
```

```
126       if (root) {
127         root->is_black = true;
128         root->is_dbl_black_right = false;
129         root->is_dbl_black_left = false;
130       }
131     }
132     delete root_parent;
133   }
134
135
136   template<typename K, typename V>
137   typename RBTCollection<K,V>::Node*
138   RBTCollection<K,V>::remove(const K& key, Node* parent, Node* subtree_root,
139                              bool& found)
140   {
141     if (subtree_root && key < subtree_root->key)
142       subtree_root = remove(key, subtree_root, subtree_root->left, found);
143     else if (subtree_root && key > subtree_root->key)
144       subtree_root = remove(key, subtree_root, subtree_root->right, found);
145     else if (subtree_root && key == subtree_root->key) {
146       found = true;
147       // leaf node
148       if (!subtree_root->left && !subtree_root->right) {
149         // if node is black then set double-black, adjust parent,
150         // and delete subtree root ...
151       }
152       // left non-empty but right empty
153       else if (subtree_root->left && !subtree_root->right) {
154         // similar to above
155       }
156       // left empty but right non-empty
157       else if (!subtree_root->left && subtree_root->right) {
158         // similar to avove
159       }
160       // left and right non empty
161       else {
162         // find inorder successor (right, then iterate left)
163         // then call remove again on inorder successor key and subtree root's
164         // right child once the key and value copy is complete
165       }
166     }
167
168     if (!found)
169       return parent;
170
171     // backtracking, adjust color at parent
172     return remove_color_adjust(parent);
173   }
174
175
176   template<typename K, typename V>
177   typename RBTCollection<K,V>::Node*
178   RBTCollection<K,V>::remove_color_adjust(Node* subtree_root)
```

```
179 {
180   // subtree root is ``grandparent'' g, with left child gl and right child gr
181   Node* g = subtree_root;
182   Node* gl = g->left;
183   Node* gr = g->right;
184   // parent p is either gl or gr
185   Node* p = nullptr;
186   bool left_parent = false;
187   if (gl && (gl->is_dbl_black_left || gl->is_dbl_black_right)) {
188     p = gl;
189     left_parent = true;
190   }
191   else if (gr && (gr->is_dbl_black_left || gr->is_dbl_black_right))
192     p = gr;
193   else
194     return subtree_root;
195
196   // parent's left child is a double black node
197   if (p->is_dbl_black_left) {
198     // do the following cases
199     // case 1: red sibling
200     // case 2: black sibling with red child (outside)
201     // case 2: black sibling with red child (outside)
202     // case 3: black sibling with black children, red parent
203     // case 3: black sibling with black children, red parent
204   }
205
206   // parent's right child is a double black node
207   if (p->is_dbl_black_right) {
208     // do the following cases
209     // case 1: red sibling
210     // case 2: black sibling with red child (outside)
211     // case 2: black sibling with red child (inside)
212     // case 3: black sibling with black children, red parent
213     // case 3: black sibling with black children, red parent
214   }
215
216   // connect up the subtree_root to the parent
217   if (left_parent)
218     subtree_root->left = p;
219   else
220     subtree_root->right = p;
221
222   return subtree_root;
223 }
224
225 ...
226
227 #endif
```