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`

```
#ifndef RBT_COLLECTION_H
#define RBT_COLLECTION_H

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

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

  // create an empty linked list
  RBTCollection();

  // copy a linked list
  RBTCollection(const RBTCollection<K,V>& rhs);

  // assign a linked list
  RBTCollection<K,V>& operator=(const RBTCollection<K,V>& rhs);
```

```

  // delete a linked list
  ~RBTCollection();

  // insert a key-value pair into the collection
  void insert(const K& key, const V& val);

  // remove a key-value pair from the collection
  void remove(const K& key);

  // find the value associated 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;

  // optional print function (for testing)
  void print() const;

private:

  // binary search tree node structure
  struct Node {
    K key;
    V value;
    Node* left;
    Node* right;
    bool is_black;            // true if black, false if red
    bool is_dbl_black_left;   // for remove rotations
    bool is_dbl_black_right;  // for remove rotations
  };

  // 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);

```

```
  // recursive helper to remove node with given key
  Node* remove(const K& key, Node* parent, Node* subtree_root, bool& found);

  // helper to perform a single rebalance step on a red-black tree on remove
  Node* remove_color_adjust(Node* parent);

  // recursive helper to do red-black insert key-val pair (backtracking)
  Node* insert(const K& key, const V& val, Node* subtree_root);

  // helper functions to perform a single right rotation
  Node* rotate_right(Node* k2);

  // helper functions to perform a single left rotation
  Node* rotate_left(Node* k2);

  // 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_root, 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>
void RBTCollection<K,V>::remove(const K& key)
{
  // check if anything to remove
  if (root == nullptr)
    return;
  // create a "fake" root to pass in as parent of root
  Node* root_parent = new Node;
  root_parent->key = root->key;
  root_parent->left = nullptr;
  root_parent->right = root;
  root_parent->is_black = true;
  root_parent->is_dbl_black_left = false;
  root_parent->is_dbl_black_right = false;
  // call remove
  bool found = false;
  root_parent = remove(key, root_parent, root, found);
  // update results
  if (found) {
    collection_size--;
    root = root_parent->right;
```

```
    if (root) {
      root->is_black = true;
      root->is_dbl_black_right = false;
      root->is_dbl_black_left = false;
    }
  }
  delete root_parent;
}


template<typename K, typename V>
typename RBTCollection<K,V>::Node*
RBTCollection<K,V>::remove(const K& key, Node* parent, Node* subtree_root,
                           bool& found)
{
  if (subtree_root && key < subtree_root->key)
    subtree_root = remove(key, subtree_root, subtree_root->left, found);
  else if (subtree_root && key > subtree_root->key)
    subtree_root = remove(key, subtree_root, subtree_root->right, found);
  else if (subtree_root && key == subtree_root->key) {
    found = true;
    // leaf node
    if (!subtree_root->left && !subtree_root->right) {
      // if node is black then set double-black, adjust parent,
      // and delete subtree root ...
    }
    // left non-empty but right empty
    else if (subtree_root->left && !subtree_root->right) {
      // similar to above
    }
    // left empty but right non-empty
    else if (!subtree_root->left && subtree_root->right) {
      // similar to avove
    }
    // left and right non empty
    else {
      // find inorder successor (right, then iterate left)
      // then call remove again on inorder successor key and subtree root's
      // right child once the key and value copy is complete
    }
  }

  if (!found)
    return parent;

  // backtracking, adjust color at parent
  return remove_color_adjust(parent);
}


template<typename K, typename V>
typename RBTCollection<K,V>::Node*
RBTCollection<K,V>::remove_color_adjust(Node* subtree_root)
```

```
{
  // subtree root is "grandparent" g, with left child gl and right child gr
  Node* g = subtree_root;
  Node* gl = g->left;
  Node* gr = g->right;
  // parent p is either gl or gr
  Node* p = nullptr;
  bool left_parent = false;
  if (gl && (gl->is_dbl_black_left || gl->is_dbl_black_right)) {
    p = gl;
    left_parent = true;
  }
  else if (gr && (gr->is_dbl_black_left || gr->is_dbl_black_right))
    p = gr;
  else
    return subtree_root;

  // parent's left child is a double black node
  if (p->is_dbl_black_left) {
    // do the following cases
    // case 1: red sibling
    // case 2: black sibling with red child (outside)
    // case 2: black sibling with red child (inside)
    // case 3: black sibling with black children, red parent
    // case 3: black sibling with black children, black parent
  }

  // parent's right child is a double black node
  if (p->is_dbl_black_right) {
    // do the following cases
    // case 1: red sibling
    // case 2: black sibling with red child (outside)
    // case 2: black sibling with red child (inside)
    // case 3: black sibling with black children, red parent
    // case 3: black sibling with black children, black parent
  }

  // connect up the subtree_root to the parent
  if (left_parent)
    subtree_root->left = p;
  else
    subtree_root->right = p;

  return subtree_root;
}

...

#endif
```