1 Reading

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

- 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 implement a new red-black tree (RBTCollection) implementation of the abstract Collection class. For this assignment, you will **not** implement the `remove` function. The primary work will be in implementing a new insert function for your BSTCollection class plus extending the Node structure to hold a "color."

2. You will have almost identical files as for HW9, except instead of `bst_collection.h` you will have `rbt_collection.h`, and instead of `hw9_tests.cpp` you will have `hw10_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 `hw10_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.

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 HW10

- You must strictly adhere to the class specification below. In particular, you cannot implement any additional helper functions.
- We will use a recursive version of insert that performs rebalancing during backtracking. Note that the extra credit assignment provides an opportunity to rebalancing "down" the tree (as opposed to "up" the tree).
- The color of each node (either black or red) will be stored as a single Boolean flag `is_black` in the `Node` structure. Thus, given a `Node` pointer `subtree_root`, if `subtree_root->is_black` then the node is black and if `!subtree_root->is_black` then the node is red.
- All of your functions from HW9 can remain the same except for your insert function.
- The above includes also using your remove function implemented from HW9. Note that you won't be doing performance tests as part of the assignment (we'll do performance tests as part of HW11). However, it can be very useful to run the performance tests to ensure you are on the right track. Note that your inserts should take roughly similar time as the previous assignment (i.e., big deviations in time suggest you might not be implementing your red-black tree insert properly).

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;

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
  };

  // 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* subtree_root);

  // 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;

  void preorder_copy(Node* lhs_subtree, const Node* rhs_subtree);

  // 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>
typename RBTCollection<K,V>::Node* RBTCollection<K,V>::rotate_right(Node* k2)
{
  // TODO
  // ...
}

template<typename K, typename V>
typename RBTCollection<K,V>::Node* RBTCollection<K,V>::rotate_left(Node* k2)
{
  // TODO
  // ...
}

template<typename K, typename V>
typename RBTCollection<K,V>::Node*
RBTCollection<K,V>::insert(const K& key, const V& val, Node* subtree_root)
{
  // TODO
  // ...
  // Note that this function must call your
  // rotate_right and rotate_left helpers
  // ...

  return subtree_root;
}


template<typename K, typename V>
void RBTCollection<K,V>::insert(const K& key, const V& val)
{
  root = insert(key, val, root);
  root->is_black = true;
  collection_size++;
}

...

#endif
```