1 Reading

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

- Ch 12 (Week #12 and #13)

2 Goals

- Finishing implementation of basic binary search trees in `C++`;
- Practice navigating / traversing binary trees;
- More recursion and pointer practice.

This assignment, like HW8, requires using CMake and make with Google Test.

3 Instructions

1. Your primary task is to implement the `remove` operation for your BSTCollection implementation from HW 8.
2. HW9 will have identical files as for HW8.
3. As with prior assignments, carefully consider the additional test cases you will need to write for `hw9_tests.cpp`. This means adding specific tests for your remove implementation. Be sure to consider a range of cases related to removing nodes.
4. Like for the other collection implementations, you must run your implementation through the performance test code. Similar to HW7, 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. Similar to HW7, create graphs showing the performance of your implementation compared to your previous Collection implementations (vector, linked list, binary search tree, and hash table 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 HW9 (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 HW9

- 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* subtree_root)
```

4 Code Listings

Listing 1: bst_collection.h

```
#ifndef BST_COLLECTION_H
#define BST_COLLECTION_H

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


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

  // create an empty linked list
  BSTCollection();

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

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

  // delete a linked list
  ~BSTCollection();

  // 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;
  };

  // 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;

  // helper to recursively remove key node from subtree
  Node* remove(const K& key, Node* subtree_root);

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

};

...

template<typename K, typename V>
typename BSTCollection<K,V>::Node*
BSTCollection<K,V>::remove(const K& key, Node* subtree_root)
{
  // TODO
  // Note: must use recursion to find node to remove
  //   and must use iteration to remove node (for case
  //   where the subtree root has two child nodes)
}

template<typename K, typename V>
void BSTCollection<K,V>::remove(const K& key)
{
  root = remove(key, root);
}

...

#endif
```