1 Reading

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

- Ch 8 and 9 (Week # 6 and 7)

2 Goals

- More practice using vectors in `C++`;
- Practice using binary search in different situations.

Like with HW4, for this assignment you will need to use both CMake and Google Test. Both are installed on `ada` and the department's linux virtual machine. Note that `CMake` also requires the `make` command, which is also installed on `ada` that the virtual machines. You will need to install each of these on your own if you are using a different environment.

3 Instructions

1. Your primary task is to implement another vector based version of the `Collection` abstract class, but this time using binary search for insert, remove, and both find operations. (Note that you *must* use binary search for the operations as described in class and the lecture notes.) You will have almost identical files as for HW4, except instead of `linked_list_collection.h` you will have `binsearch_collection.h`, instead of `hw4_tests.cpp` you will have `hw5_tests.cpp`, and instead of `hw4_perf.cpp` you will have `hw5_perf.cpp`. You will also need to make small changes to your `CMakeLists.txt` file for HW5.

2. As with prior assignments, carefully consider the additional test cases you will need to write for `hw5_tests.cpp` (if any).

3. Like for HW4, you must run your implementation through the performance test code. Similar to HW4, 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					

4. Similar to HW4, create graphs showing the performance of your implementation compared with your HW3 vector-based implementation and your HW4 linked list implementation. Again, note that to make the comparison somewhat "fair" you will need to ensure you run the tests on the same machine as for HW3 and HW4, or else rerun the tests for HW3 and HW4 again as you do the tests for HW5.

5. 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.

6. 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 and Hints for HW5

- In the code listing below, we define the following helper function.

  ```
  bool binsearch(const K& key, int& index) const;
  ```

  You *must* implement and use this helper for performing binary search in your collection functions. The function returns true if the given `key` value is in the vector and false if it isn't. The function also returns the `index` of the location of the matching key-value pair if the key is found, and the index of where the key-value pair should be located if the key was not found.

- To insert a value into a vector, use the `insert` function. For instance, the following statement inserts the key-value pair `p` into `kv_list` before the `i`th index in the list (shifting all values from `i` to the right in the vector).

  ```
  kv_list.insert(kv_list.begin() + i, p);
  ```

- Again, your collection functions insert, remove, find by key, and find range of keys must all appropriately use the `binsearch` helper function to perform their tasks.

- Unlike in HW3 and HW4, you don't need to call the `sort` helper function provided by C++ since you are maintaining a sorted key-value vector in this as assignment. This means that your sort function should be blazingly fast!

4 Code Listings

Listing 1: `binsearch_collection.h`

```
1   #ifndef BINSEARCH_COLLECTION_H
2   #define BINSEARCH_COLLECTION_H
3
4   #include <vector>
5   #include "collection.h"
6
7   template<typename K, typename V>
8   class BinSearchCollection : public Collection<K,V>
9   {
10  public:
11
12    // insert a key-value pair into the collection
13    void insert(const K& key, const V& val);
14
15    // remove a key-value pair from the collection
16    void remove(const K& key);
17
18    // find and return the value associated with the key
19    bool find(const K& key, V& val) const;
20
21    // find and return the list of keys >= to k1 and <= to k2
```

```
22      void find(const K& k1, const K& k2, std::vector<K>& keys) const;
23
24      // return all of the keys in the collection
25      void keys(std::vector<K>& keys) const;
26
27      // return all of the keys in ascending (sorted) order
28      void sort(std::vector<K>& keys) const;
29
30      // return the number of keys in collection
31      int size() const;
32
33    private:
34
35      // helper function for binary search
36      bool binsearch(const K& key, int& index) const;
37
38      // vector storage
39      std::vector<std::pair<K,V>> kv_list;
40
41    };
42
43
44    // This function returns true and sets index if key is found in
45    // kv_list, and returns false and sets index to where key should go in
46    // kv_list otherwise. If list is empty, index is unchanged.
47    template<typename K, typename V>
48    bool BinSearchCollection<K,V>::binsearch(const K& key, int& index) const
49    {
50      ...
51    }
52
53    ... remaining function implementations here ...
54
55    #endif
```