1 Reading

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

- Ch 10 (Week # 8)
- Note that we won't start the topic until late in the week.

2 Goals

- More practice using pointers in `C++`;
- Practice writing insertion sort, merge sort, and quick sort

In this assignment, you can use either CMake or `g++`. I won't be providing you with specific unit tests. However, you will want to do some smaller tests to help you develop correct sorting algorithms before jumping into performance testing.

3 Instructions

1. Your primary task is to implement insertion sort, merge sort, and quick sort over your linked list implementation from HW 4. Your implementations of these three sort algorithms, however, cannot move data between any nodes in the linked list. Instead, to perform each search, nodes must be moved within the list by adjusting next pointers (i.e., by "splicing" out and "re-connecting" nodes within the linked list). All of your work will go into the `linked_list_collection.h` file from HW 4.
2. You must run your sort implementations through the performance test code provided in `hw6_sort.cpp`. You will do two rounds of performance testing, one round on the random insert data sets (`rand-10k.txt`, etc.), and another round on the descending insert data sets that will be provided (`desc-10k.txt`, `desc-20k.txt`, and so on). Similar to before, you will need to:
   (a). Run your program at least three times for each of the ten test files and sort algorithms, recording the results. (Note that you must run each of the tests the same number of times.)
   (b). Using the run results, create an overall average for each of each each sort operation and test file.
   (b). Create two tables of results. Your tables should be formatted similarly to the following (yet to be filled in) tables.

	`rand-10k`	`rand-20k`	`rand-30k`	`rand-40k`	`rand-50k`
Insertion Sort Average					
Quick Sort Average					
Merge Sort Average					

	`desc-10k`	`desc-20k`	`desc-30k`	`desc-40k`	`desc-50k`
Insertion Sort Average					
Quick Sort Average					
Merge Sort Average					

3. Similar to previous homeworks, create two graphs showing the performance of your sort implementations. One graph should be for the first table above, and the other graph should be for the second table above.

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 and Hints for HW6

- Your main job is to implement the `insertion_sort()`, `merge_sort()`, and `quick_sort()` public member functions in `LinkedListCollection` below.
- The `insertion_sort()` function does not require any helper functions.
- The `merge_sort()` function uses a helper function (as described in class). Note that the signature requires additional syntax to work with `C++` templates (see below). The helper function does the majority of the work—halving the list, recursively calling the helper, and merging the resulting sorted lists.
- The `quick_sort()` function also uses a helper function (as described in class). Note that the signature also requires additional syntax. The helper function for quick sort also does the majority of the work, like with merge sort.
- You **may not use any other helper functions** for implementing the three sorting functions.
- Your sorting functions **may not perform any data moves between nodes**. Instead, all modification of list order must be by rearranging nodes in the linked list.
- To run the performance test code (`hw6_sort.cpp`) for your sorting algorithm implementations you provide the sorting algorithm (either `i`, `q`, or `m` for insertion, quick, and merge sort, respectively) followed by the test file. Assuming the executable is named `hw6sort`

  ```
  hw6sort i desc-10k.txt
  ```

  runs your insertion sort algorithm over the smallest descending insert order data set (where keys are inserted from largest to smallest). Alternatively,

  ```
  hw6sort q desc-10k.txt
  ```

  runs your quick sort algorithm implementation, and

  ```
  hw6sort m desc-10k.txt
  ```

  runs your merge sort algorithm implementation.

4 Code Listings

Listing 1: `linked_list_collection.h`

```
#ifndef LINKED_LIST_COLLECTION_H
#define LINKED_LIST_COLLECTION_H

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


```

```cpp
template<typename K, typename V>
class LinkedListCollection : public Collection<K,V>
{
public:

  // create an empty linked list
  LinkedListCollection();

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

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

  // delete a linked list
  ~LinkedListCollection();

  // 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;

  // in place sorting
  void insertion_sort();
  void merge_sort();
  void quick_sort();

private:

  // linked list node structure
  struct Node {
    K key;
    V value;
    Node* next;
  };
  Node* head;
  Node* tail;
  int length;

  // helper to delete linked list
  void make_empty();

  // merge sort helper (see course notes for details)
  Node* merge_sort(Node* left, int len);

  // quick sort helper (see course notes for details)
  Node* quick_sort(Node* start, int len);

};

...

template<typename K, typename V>
void LinkedListCollection<K,V>::insertion_sort()
{
  ... insertion sort implementation goes here ...
}


template<typename K,typename V>
typename LinkedListCollection<K,V>::Node*
LinkedListCollection<K,V>::merge_sort(Node* left, int len)
{
  ... merge sort helper implementation goes here ...
}


template<typename K, typename V>
void LinkedListCollection<K,V>::merge_sort()
{
  ... merge sort implementation goes here ...
}


template<typename K, typename V>
typename LinkedListCollection<K,V>::Node*
LinkedListCollection<K,V>::quick_sort(Node* start, int len)
{
  ... quick sort helper implementation goes here ...
}


template<typename K, typename V>
void LinkedListCollection<K,V>::quick_sort()
{
  ... quick sort implementation goes here ...
}


#endif
```