1 Reading

This week there is no reading assigned (in part due to Exam 1).

2 Goals

- Practice (refresh) using pointers and dynamic memory in `C++`;
- Become more familiar with the key-value pair collection ADT; and
- Practice thinking about and developing new test cases.

Like with HW3, 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 a linked list version of the `Collection` abstract class. *Your linked list must maintain both a head and tail pointer such that insertions are made at the end of the linked list.* You will have almost identical files as for HW3, except instead of `vector_collection.h` you will have `linked_list_collection.h`, instead of `hw3_tests.cpp` you will have `hw4_tests.cpp`, and instead of `hw3_perf.cpp` you will have `hw4_perf.cpp`. You will also need to make small changes to your `CMakeLists.txt` file for HW4. Finally, note that for this assignment, you will again use the built-in `sort` function (see below).

2. You should consider additional test cases for your `hw4_tests.cpp` file specific for linked lists. In general, your tests should be identical to `hw3_tests.cpp`, but with additional tests cases for your linked list implementation.

3. Like for HW3, you must run your implementation through the performance test code. Similar to HW3, you must:

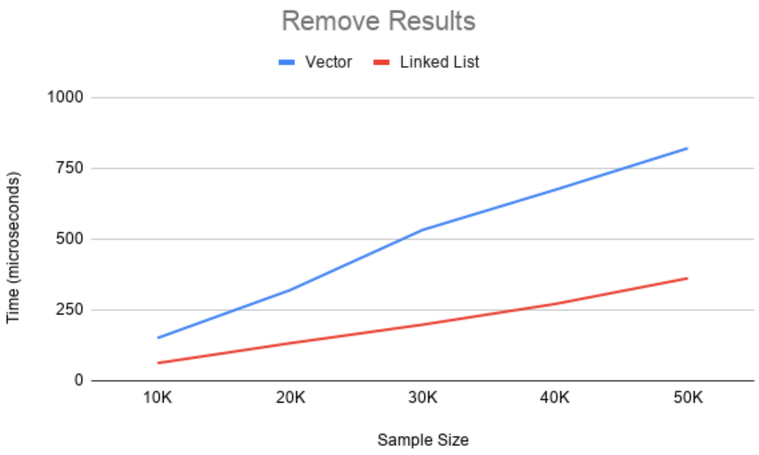
   (a). Run your program at least three times for each of the five test files and record the results. (Note that you must run each of the test files the same number of times.)

   (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	rannd-30k	rand-40k	rand-50k
Insert Average					
Remove Average					
Find Average					
Range Average					
Sort Average					

4. Similar to HW3, you will create graphs showing the performance of your implementation. Unlike HW3, you will have one graph for each operation (insert, remove, find, range, and sort). Each graph will compare the performance of your linked list implementation to the vector-based implementation from HW3. (Note that to make the comparison somewhat "fair" you will need to ensure you run the tests on the same machine as for HW3, or else rerun the tests for HW3 as you do the tests for HW4.) Here is an example graph for the remove operation (your results will likely vary).



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.

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"


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;

private:
  // linked list node structure
  struct Node {
    K key;
    V value;
    Node* next;
  };
  Node* head;                     // pointer to first list node
  Node* tail;                     // pointer to last list node
  int length;                     // number of linked list nodes in list
};

...

template<typename K, typename V>
void LinkedListCollection<K,V>::sort(std::vector<K>& keys) const
{
  Node* ptr = head;
  while (ptr != nullptr) {
    keys.push_back(ptr->key);
    ptr = ptr->next;
  }
  std::sort(keys.begin(), keys.end());
}

..

#endif
```