1 Reading

This week either continue or finish Week # 8 readings. No additional readings are assigned.

2 Goals

- Practice implementing hash tables in `C++`;
- Practice writing resizable array in `C++`;
- More linked list and pointer practice.

This assignment, like HW5, requires using CMake and make with Google Test.

3 Instructions

1. Your primary task is to implement a new HashTableCollection implementation of the abstract Collection class. Your implementation will be similar in spirit to Java's HashMap class such that it uses resizing and rehashing (i.e. a resizable table array) with a load factor of 75%, an initial table capacity of 16, and where resizing doubles the size of the table array. All of your work will go into the `hash_table_collection.h` file (outlined below).

2. You will have almost identical files as for HW5, except instead of `binsearch_collection.h` you will have `hash_table_collection.h`, instead of `hw5_tests.cpp` you will have `hw7_tests.cpp`, and instead of `hw5_perf.cpp` you will have `hw7_perf.cpp`. You will also need to make small changes to your `CMakeLists.txt` file for HW7.

3. As with prior assignments, carefully consider the additional test cases you will need to write for `hw7_tests.cpp`.

4. Like for HW5, you must run your implementation through the performance test code. Similar to HW5, 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 HW5, create graphs showing the performance of your implementation *compared with your HW3, HW4, and HW5 Collection implementations*. Again, note that to make the comparison "fair" you will need to ensure you run the tests on the same machine as for HW3, HW4, and HW5, or better, rerun the tests for for these again as you do the tests for HW7 (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 and Hints for HW6

- As shown below, the hash table array is represented by the private member variable:

  ```
  Node** table_array;
  ```

  This definition states that `table_array` is a pointer to a pointer to a `Node` object (note that "pointer to a pointer to" is not a typo!). This notation is often used to define a dynamically sized array, where in our case we have an array of `Node` pointers (i.e., an array of `Node*`). In particular, an array name in C/C++ is really just a pointer to the first element of the array (plus some extra `[]` syntax). For arrays, the notation `table_array[i]` is just "syntactic sugar" and is equivalent to the dereference operation

  ```
  *(table_array + i)
  ```

  where `table_array + i` uses "pointer arithmetic" to move `i` locations past the location `table_array` points to. To initially create the table array, we use the `new` command:

  ```
  table_array = new Node*[table_capacity];
  ```

  Since `new` returns a pointer to the right-hand-side type, which in this case is `Node*`, the `table_array` variable must therefore be of type `Node**` for the above statement to be well typed. Note that we use `Node*` as the right-hand-side type because we are creating an array of linked-list "head" pointers to use for separate chaining.

- The `collection_size` member variable stores the current number of elements being stored in the collection. This variable has the same purpose as the `length` variable in HW4.

- The `table_capacity` member variable stores the current length of the hash table array. The value of this variable changes after a resize and rehash call. Note that a resize and rehash call is only made when an insert takes the load factor of the hash table over the 75% threshold. The `table_capacity` should double from its previous value as part of each resize and rehash.

- The `load_factor_threshold` member variable stores the load factor threshold that the table's load factor is compared against. When the table's load factor goes above the load factor threshold value, the table is resized and rehashed prior to inserting a new key-value pair. For this assignment, the `load_factor_threshold` is set at 75% and doesn't change throughout the lifetime of the object (signified by the `const` qualifier in the variable declaration).

- The linked lists (chains) associated with each hash table array index are inserted into from the front (i.e., at the "head" position). No tail pointer is needed for managing these linked lists. Instead of using a head pointer directly, each table array element represents the head pointer. Thus, the first linked list node in the first array element is `table_array[0]` with key value `table_array[0]->key`. The second node is `table_array[0]->next`, and so on.

- To implement the `keys` member function as well as the range-version of `find`, you must iterate through all nodes in all chains of the hash table array. As a hint, this can be done with a for loop (iterating through the array indices), and then within the for loop using a while loop to navigate through the corresponding linked list chain.

- Unlike previous assignments, the code listing below provides a number of hints to help with the implementation of the hash table. Specifically, some code is provided below to help get you started, and comments are used to layout the general steps of some member functions. A comment of the form:

  ```
  // ...
  ```

  signifies that you need to add code in this spot according to the comment on the previous line. If any of the "hints" are unclear or confusing, please feel free to ask for clarification either directly or (preferrably) via Piazza.

4 Code Listings

Listing 1: hash_table_collection.h

```
#ifndef HASH_TABLE_COLLECTION_H
#define HASH_TABLE_COLLECTION_H

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

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

  // create an empty linked list
  HashTableCollection();

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

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

  // delete a linked list
  ~HashTableCollection();

  // 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;
```

```cpp

  // 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:

  // helper to empty entire hash table
  void make_empty();

  // resize and rehash the hash table
  void resize_and_rehash();

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

  // number of k-v pairs in the collection
  int collection_size;

  // number of hash table buckets (default is 16)
  int table_capacity;

  // hash table array load factor (set at 75% for resizing)
  const double load_factor_threshold;

  // hash table array
  Node** hash_table;
};


template<typename K, typename V>
HashTableCollection<K,V>::HashTableCollection() :
  collection_size(0), table_capacity(16), load_factor_threshold(0.75)
{
  // dynamically allocate the hash table array
  hash_table = new Node*[table_capacity];
  // initialize the hash table chains
  for (int i = 0; i < table_capacity; ++i)
    hash_table[i] = nullptr;
}

template<typename K, typename V>
void HashTableCollection<K,V>::make_empty()
{
```

```
  // make sure hash table exists
  // ...
  // remove each key
  // ...
  // remove the hash table
  delete hash_table;
}

template<typename K, typename V>
HashTableCollection<K,V>::~HashTableCollection()
{
  make_empty();
}


template<typename K, typename V>
HashTableCollection<K,V>::HashTableCollection(const HashTableCollection<K,V>& rhs)
  : hash_table(nullptr)
{
  *this = rhs;
}

template<typename K, typename V>
HashTableCollection<K,V>&
HashTableCollection<K,V>::operator=(const HashTableCollection<K,V>& rhs)
{
  // check if rhs is current object and return current object
  if (this == &rhs)
    return *this;
  // delete current object
  make_empty();
  // initialize current object
  // ...
  // create the hash table
  // ...
  // do the copy
  // ...
  return *this;
}

template<typename K, typename V>
void HashTableCollection<K,V>::resize_and_rehash()
{
  // setup new table
  int new_capacity = table_capacity * 2;
  // ... similarly with collection size ...
  // dynamically allocate the new table
  Node** new_table = new Node*[new_capacity];
  // initialize new table
  // ...
  // insert key values
  std::vector<K> ks;
  keys(ks);
```

```
  for (K key : ks) {
    // hash the key
    // ...
    // create a new node in new table
    // ...
  }
  // clear the current data
  make_empty();
  // update to the new settings
  hash_table = new_table;
  // ... update remaining vars ...
}

template<typename K, typename V>
void HashTableCollection<K,V>::insert(const K& key, const V& val)
{
  // check current load factor versus load factor threshold,
  // and resize and copy if necessary by calling resize_and_rehash()
  // ...
  // hash the key
  // ...
  // create the new node
  // ...
  // update the size
  // ...
}

...

template<typename K, typename V>
void HashTableCollection<K,V>::sort(std::vector<K>& ks) const
{
  keys(ks);
  std::sort(ks.begin(), ks.end());
}

template<typename K, typename V>
int HashTableCollection<K,V>::size() const
{
  return collection_size;
}

#endif
```