1 Reading

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

- Ch 4 and 5

2 Goals

- Practice using vectors in `C++`;
- Become familiar with the key-value pair collection ADT used in this course;
- Learn to use the `C++ std::pair` structure;
- Practice with (pure) abstract classes in `C++`;
- Practice with class templates (parametric types) in `C++`;
- Practice thinking about and developing new test cases; and
- Further setup your environment for the semester.

Note you are free to use whatever editor and environment you like for this class, but your programs must be able to compile and run on `ada` (which is running Ubuntu) using `g++`. Note that for this assignment you will need to use both CMake (`https://cmake.org/`) and Google Test (`https://github.com/google/googletest`). 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

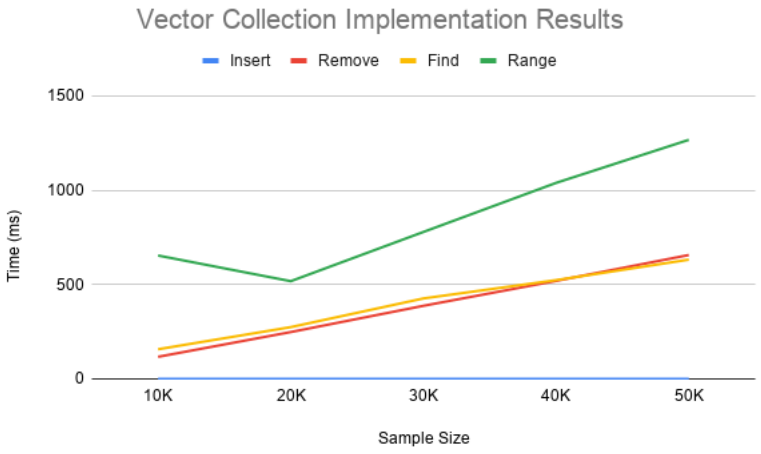
1. Your primary task is to finish implementing the `vector_collection` functions.
2. You must create and document additional test cases in the `hw3_tests.cpp` file. Your tests must be well thought out and cover cases for each `vector_collection` member function.
3. In addition, you must run your implementation through the performance test code and test files provided (via piazza). In particular, 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.)

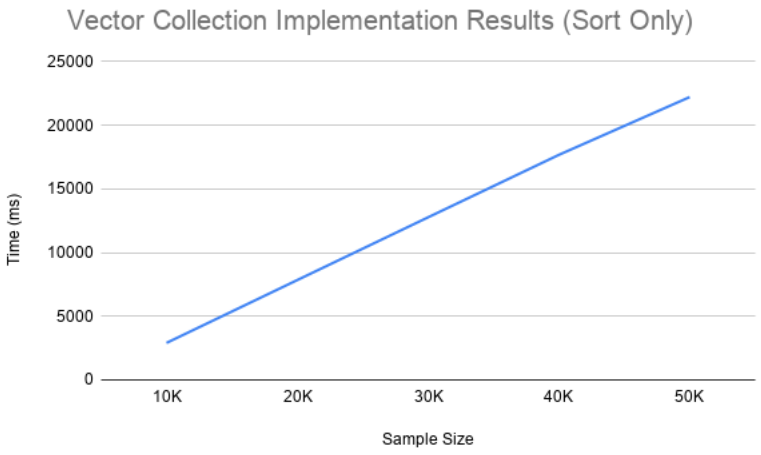
(b). Using the run results, create an overall average for each of the three runs, for each operation and test file. For example, you should end up with one average value for insert over the three 10K runs, one average value for insert over the three 20K runs, and so on.

(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. Create two graphs of the results using Excel, Google Sheets, or another similar tool. One graph should list all operations except sort, and another should contain only the sort results. Here is an example of what the graphs should generally look like (your results will likely be different).





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.

4 Additional Details

The key-value collection abstract data type (ADT). The `Collection` class defined in `collection.h` below is a purely abstract class for storing key-value pairs. Each collection consists of zero or more unique keys. Each key has an associated value, where a particular key-value combination is called a "key-value pair". All of the keys in a particular collection have to be of the same type, and all of the values in a collection have to be of the same type (which will typically be a different type than the key type). The class supports inserting a key-value pair into the collection, removing a key-value pair from the collection (via the key), finding the value associated with a key, finding the keys within a range of keys, sorting the keys in the collection, and returning the number of keys in the collection. It is assumed that a given key is only inserted once into a collection. The behavior of the collection is undefined (i.e., anything could happen) if the same key is inserted more than once (with either the same or a different value).

Implementing a vector-based key-value collection. The `VectorCollection` class implements the abstract `Collection` class using an underlying `vector` object (i.e., using a resizable array data structure). Your primary job is to provide this implementation. Because the `Collection` class uses templated types, each function implementation will need to be put into the `vector_collection.h` file (as opposed to having a separate cpp file).

Testing your vector-based implementation. You will do two types of tests of your vector-based collection implementation. The first will be to write a set of "unit" style tests, filling out the tests provided in the `hw3_test.cpp` file. To run the unit tests, you will execute the `hw3test` program (described below). The second set of tests will be performance related. You will use the `hw3_perf.cpp` file (which you won't need to modify) to run a "test harness" that will make a series of timed calls to your `VectorCollection` class and report the results. Running the `hw3perf` executable (built from `hw3_perf.cpp` and the `test_driver.h` files) over a test data file will generate the performance test results. Five different test data files will be provided (via piazza), where the first file contains 10,000 key-value pair insertions, the second file contains 20,000 key-value pair insertions, and so on, up to the last file of 50,000 key-value pair insertions. To run the performance tests for the 10,000 key-value insertions, you would execute the command:

```
$ ./hw3perf rand-10K.txt
```

Compiling and Running. For this assignment, we'll be using the CMake tool to help us compile all of our files at once. In our case, we'll be using CMake to generate a makefile, which we will

then use to build our code base. Listing 1 below gives the CMake file `CMakeLists.txt` that we'll be using. To run CMake, we'll simply run the following from the command line (once all of the source files are created):

```
$ cmake CMakeLists.txt
```

After this command successfully completes, you should see a new directory called `CMakeFiles` and two new files `CMakeCache.txt` and `cmake_install.cmake`. After you successfully run the `cmake` command above, you won't need to run it again. To compile and build the two executables `hw3test` and `hw3perf`, you will run the following command:

```
$ make
```

As you change your source code, you only need to rerun `make` to recompile your code.

5 Code Listings

Listing 1: `CMakeLists.txt`

```
cmake_minimum_required(VERSION 3.0)

set(CMAKE_CXX_STANDARD 11)

# locate gtest
find_package(GTest REQUIRED)
include_directories(${GTEST_INCLUDE_DIRS})

# create unit test executable
add_executable(hw3test hw3_tests.cpp)
target_link_libraries(hw3test ${GTEST_LIBRARIES} pthread)

# create performance executable
add_executable(hw3perf hw3_perf.cpp)
```

Listing 2: `collection.h`

```
#ifndef COLLECTION_H
#define COLLECTION_H

#include <vector>

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

  // insert a key-value pair into the collection
  virtual void insert(const K& key, const V& val) = 0;

  // remove a key-value pair from the collectiona
  virtual void remove(const K& key) = 0;

  // find and return the value associated with the key
  virtual bool find(const K& key, V& val) const = 0;

  // find and return the list of keys >= to k1 and <= to k2
  virtual void find(const K& k1, const K& k2, std::vector<K>& keys) const = 0;

  // return all of the keys in the collection
  virtual void keys(std::vector<K>& keys) const = 0;

  // return all of the keys in ascending (sorted) order
  virtual void sort(std::vector<K>& keys) const = 0;

  // return the number of keys in the collection
  virtual int size() const = 0;

};

#endif
```

Listing 3: vector_collection.h

```
#ifndef VECTOR_COLLECTION_H
#define VECTOR_COLLECTION_H

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

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

  // 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 and return the value associated with the key
  bool find(const K& key, V& val) const;

  // find and return the list of keys >= to k1 and <= to k2
  void find(const K& k1, const K& k2, std::vector<K>& keys) const;

  // return all of the keys in the collection
  void keys(std::vector<K>& keys) const;

  // return all of the keys in ascending (sorted) order
  void sort(std::vector<K>& keys) const;

  // return the number of keys in collection
  int size() const;

  private:
  std::vector<std::pair<K,V>> kv_list;

};


template<typename K, typename V>
void VectorCollection<K,V>::insert(const K& key, const V& val)
{
  std::pair<K,V> p(key, val);
  kv_list.push_back(p);
}


template<typename K, typename V>
void VectorCollection<K,V>::remove(const K& key)
{
  ... function body here ...
}
```

```
53
54    ... rest of function implementations here ...
55
56    #endif
```

Listing 4: hw3_tests.cpp

```
#include <iostream>
#include <string>
#include <gtest/gtest.h>
#include "vector_collection.h"

using namespace std;

// Test 1
TEST(BasicCollectionTest, CorrectSize) {
  VectorCollection<string,double> c;
  ASSERT_EQ(c.size(), 0);
  c.insert("a", 10.0);
  ASSERT_EQ(c.size(), 1);
  c.insert("b", 20.0);
  ASSERT_EQ(c.size(), 2);
}

// Test 2
TEST(BasicCollectionTest, InsertAndFind) {
  VectorCollection<string,double> c;
  double v;
  ASSERT_EQ(c.find("a", v), false);
  c.insert("a", 10.0);
  ASSERT_EQ(c.find("a", v), true);
  ASSERT_EQ(v, 10.0);
  ASSERT_EQ(c.find("b", v), false);
  c.insert("b", 20.0);
  ASSERT_EQ(c.find("b", v), true);
  ASSERT_EQ(v, 20.0);
}

// Test 3
TEST(BasicCollectionTest, RemoveElems) {
  VectorCollection<string,double> c;
  c.insert("a", 10.0);
  c.insert("b", 20.0);
  c.insert("c", 30.0);
  double v;
  c.remove("a");
  ASSERT_EQ(c.find("a", v), false);
  c.remove("b");
  ASSERT_EQ(c.find("b", v), false);
  c.remove("c");
  ASSERT_EQ(c.find("c", v), false);
  ASSERT_EQ(c.size(), 0);
}

// Test 4
TEST(BasicCollectionTest, GetKeys) {
  VectorCollection<string,double> c;
  c.insert("a", 10.0);
  c.insert("b", 20.0);
```

```
53      c.insert("c", 30.0);
54      vector<string> ks;
55      c.keys(ks);
56      vector<string>::iterator iter;
57      iter = find(ks.begin(), ks.end(), "a");
58      ASSERT_NE(iter, ks.end());
59      iter = find(ks.begin(), ks.end(), "b");
60      ASSERT_NE(iter, ks.end());
61      iter = find(ks.begin(), ks.end(), "c");
62      ASSERT_NE(iter, ks.end());
63      iter = find(ks.begin(), ks.end(), "d");
64      ASSERT_EQ(iter, ks.end());
65    }
66
67    // Test 5
68    TEST(BasicCollectionTest, GetKeyRange) {
69      VectorCollection<string,double> c;
70      c.insert("a", 10.0);
71      c.insert("b", 20.0);
72      c.insert("c", 30.0);
73      c.insert("d", 40.0);
74      c.insert("e", 50.0);
75      vector<string> ks;
76      c.find("b", "d", ks);
77      vector<string>::iterator iter;
78      iter = find(ks.begin(), ks.end(), "b");
79      ASSERT_NE(iter, ks.end());
80      iter = find(ks.begin(), ks.end(), "c");
81      ASSERT_NE(iter, ks.end());
82      iter = find(ks.begin(), ks.end(), "d");
83      ASSERT_NE(iter, ks.end());
84      iter = find(ks.begin(), ks.end(), "a");
85      ASSERT_EQ(iter, ks.end());
86      iter = find(ks.begin(), ks.end(), "e");
87      ASSERT_EQ(iter, ks.end());
88    }
89
90    // Test 6
91    TEST(BasicCollectionTest, KeySort) {
92      VectorCollection<string,double> c;
93      c.insert("a", 10.0);
94      c.insert("e", 50.0);
95      c.insert("c", 30.0);
96      c.insert("b", 20.0);
97      c.insert("d", 40.0);
98      vector<string> sorted_ks;
99      c.sort(sorted_ks);
100     // check if sort order
101     for (int i = 0; i < int(sorted_ks.size()) - 1; ++i)
102     ASSERT_LE(sorted_ks[i], sorted_ks[i+1]);
103   }
104
105   int main(int argc, char** argv)
```

```
106   {
107     testing::InitGoogleTest(&argc, argv);
108     return RUN_ALL_TESTS();
109   }
```

Listing 5: hw3_perf.cpp

```cpp
#include <iostream>
#include "vector_collection.h"
#include "test_driver.h"

using namespace std;

int main(int argc, char** argv)
{
  // check command-line arguments
  if (argc != 2) {
    cout << "usage: " << argv[0] << " filename" << endl;
    return 1;
  }

  // run basic performance test
  VectorCollection<string,double> test_collection;
  TestDriver<string,double> driver(argv[1], &test_collection);
  driver.run_tests();
  driver.print_results();
}
```