CPTS 223 Advanced Data Structure C/C++ Fall 2024

MA3: C++ STL Map

1 Learning Objectives

At the conclusion of this programming assignment, participants should be able to:

- Apply the C++ STL std::map associative container.

2 Prerequisite

Before starting this programming assignment, participants should be able to:

- Analyze a basic set of requirements and apply top-down design principles for a problem
- Design, implement, and test medium programs in an object-oriented language
- Build programs through CMake and maintain repository via Git
- Finish reading §4.8 Sets and Maps in the Standard Library of Textbook and slides on Canvas. After reading, you will understand:
  - How to insert an element into a map
  - How to access the key and value of an element in the map (e.g., ".first" and ".second")
  - How to find an element by key
  - How to use iterator when searching/inserting/deleting.

3 Overview and Requirements

3.1 Overview

We understand that a map is an abstract data type. It is also sometimes known as an associative container or generalized dictionary. A map is composed of a collection of key-value pairs that are ordered by *unique keys*. In C++, we can use the STL std::map. The C++ std::map is implemented using a *top-down red-black tree* (see §12.2 of Textbook).

To get a feel for the official C++ devdocs on maps, it is highly recommended to checkout the header for STL map at `https://devdocs.io/cpp/header/map`. Moreover, the following site does a nice job of summarizing the operations that you will need to use for the assignment: `https://en.cppreference.com/w/cpp/container/map`.

In this MA, you will start with the starter code found on Canvas (see Figure 1). Feel free to add more files as you see fit. You will need to use a std::map to store and manage the Twitter user information provided in the existing file. It is worth noting that the amount of data is purposely very small, so that we can focus more on the map operations instead of handling large-scale data. In the code "generateUsers.hpp", you are provided with some constraints on how the data members of the struct "User" are used and the implementation of generating random data or reading existing data. You may not change "generateUsers.hpp". The members of "User" is listed as follows.

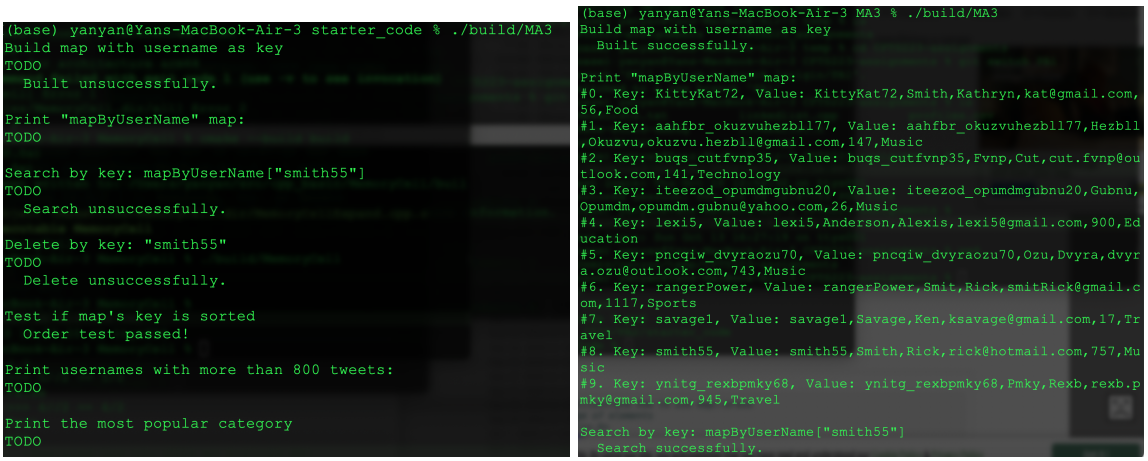

Figure 1: Left: example running result of the starter code. Right: example running result of the final code.

```
struct User {
    string userName; // unique (can be key)
    string lastName;
    string firstName;
    string email; // unique (can be key)
    int numPosts; // # of posts
    string mostViewedCategory; // the most viewed category by this user
};
```

We assume that the "userName" is unique, i.e., a "userName" can only be used by an account. We further suppose that the "emails' is also unique, i.e., an email can only be associated with an account. In this way, we can use both the "userName" or the "email" as the key of the map, which will be both implemented in this MA (see two map scenarios in Section 3.2 later).

Our goal is to successfully perform several operations for the map. Specifically, we will implement (i) build the map by reading the existing csv file, (ii) print the map with an iterator, (iii) search the element of the map by (unique) key, (iv) delete the element of the map by key, (v) test whether the keys are ordered, (vi) print keys that satisfy some certain conditions in their corresponding values, (vii) analyze statistics of some values. Figure 1 also shows an example result for these tasks.

3.2 Requirements

The task of this MA is to complete 7 functions in "main.cpp" for both two map scenarios.

- (70 pts) Map Scenario 1: Search based on userName (as the key of the map).

    1. (10 pts) `buildMapByUserName`: creates and returns a new map from an input vector of "Users". The code that generates the vector of "Users" from the existing csv file is provided in the starter code. The key for this scenario is "User.userName" (string type) and value of this map is the entire "User", so we declare "`map<string, User> mapByUserName`" for this scenario in "main.cpp".
    2. (10 pts) `printMap`: prints the entire map from the smallest key to the largest key (recall that STL map is implemented by red-black tree, so the keys are stored according to their orders). This function should print the elements alphabetically based on their keys, e.g., "abc" will be smaller than "xyz".
    3. (10 pts) `testSearchByKey`: returns `true` if the query key is present in the map, while returns `false` if the query key is not present in the map. We will search if a particular key "smith55" is already present in the map.

4. (10 pts) `testDeleteByKey`: returns `true` if it finds the query key and erases the corresponding target element from the map. We will delete the element with the key "smith55" from the map.
5. (10 pts) `isMapSorted`: returns `true` if the elements in the map, when accessed sequentially via an iterator, are sorted in ascending order by key. It returns `false` if the keys are not fully sorted.
6. (10 pts) `printActiveUsers`: prints "userNames" who have more than 800 tweets.
7. (10 pts) `printMostPopularCategory` prints the category that is labeled as mostViewedCategory by the largest number of users. For example, if 5 out of 10 users have Sports as their most viewed category, the function will print Sports.

- (30 pts) Map Scenario 2: Search based on EmailAddress

1. (10 pts) `buildMapByEmail`: creates/returns a new map that uses "emails" as the key from the same vector of "Users" as in Scenario 1. The value of this map is also the entire "User".
2. (10 pts) `testSearchByKey`: returns `true` if the query key is present in the map (the same with the `testSearchByKey` in Map Scenario 1). We will search if the key "kat@gmail.com" is already present.
3. (10 pts) `testDeleteByKey`: eturns true if it finds the query key and erases the corresponding target element from the map. We will delete the element with the key "kat@gmail.com".

4 How to Submit: Github (and Share with TA)

1. In your Git repository for this class's coding assignments, **create a new branch called "MA3"** (Refer to this YouTube video about how to create a branch from Github web interface or the terminal). In the current working directory, also create a new directory called "MA3" and place all MA3 files in the "MA3" directory. All files for MA3 should be added, committed and pushed to the remote origin which is your private GitHub repository created when during PA1 (NO NEED TO CREATE A NEW REPO).
2. You should submit at least the following files:
   - the header file ("generateUsers.hpp", provided by the starter code, where all users' information is generated).
   - the main C++ source file ("main.cpp"), where you implement the required functions and their calls.
   - the data file "existingData.csv", which is provided by the starter code.
   - a "CMakeLists.txt" file containing your CMake building commands on Linux/WSL/MacOS which can compile your code.
3. You can refer the example GitHub template project for how to use CMake at `https://github.com/DataOceanLab/CPTS-223-Examples`.
4. Please invite the GitHub accounts of TAs (see Syllabus page and check TA's names as well as their Github usernames before submitting) as the collaborators of your repository. You should submit a URL link to the branch of your private GitHub repository on Canvas. Otherwise, we will not be able to see your repository and grade your submission.
5. Please push all commits of this branch before the due date for submitting your Github link to Canvas portal. Otherwise it might be considered as late submission.

Here is a checklist for submitting your code:

- Invite all TAs and instructor as collaborators of your created repository "CPTS223_assignments". Their github username can be found in Syllabus on Canvas.
- Commit and push your local code to your repository.
- Make sure that your repository reflects and contains all your latest files.
- Copy the link to your repository in the Canvas submission portal.

5 Grading Guidelines

This assignment is worth 100 points. We will grade according to the following criteria: See Section 3.2 for individual points.