[2024 Fall] UNIST CSE221 Data Structures
Programming Assignment #4:
Student Score Management System

Prof: Taesik Gong (taesik.gong@unist.ac.kr) TA: Yeongjun Kwak (kyj05137@unist.ac.kr)

Introduction

In this programming assignment, you will implement a Student Score Management System to handle student data efficiently. This assignment focuses on using different types of data structures to store, retrieve, and sort information, specifically using Hash Tables and Skip Lists. You will implement three classes, each responsible for a specific part of the system. By integrating these data structures, you'll gain a deeper understanding of ordered and unordered data management, as well as practical application of hashing and skip lists.

Assumptions:

- Each student ID is unique and follows an 8-digit format (e.g., 20241234).
- The score is an integer between 0 and 100, inclusive (i.e., [0, 100]).
- Multiple students can have the same score.

Part 1: StudentMap (HashTable)

Implement a hash table to store student scores (value) by student ID (key). This unordered map allows quick retrieval and update operations by student ID.

- Data Structure: Hash Table
- Size of Hash Table: 1000
- Collision Handling Method: **Separate Chaining**
- You should properly implement an appropriate **hash function**.
- Class: StudentMap
- Required Methods:
    1. **Constructor** & **Destructor**
    2. **add_student**(int student_id, int score): Adds a student to the hash table.

3. **update_score**(int student_id, int new_score): Updates the score of an existing student.
4. **get_score**(int student_id): Retrieves the score of a specific student. If there are multiple matches, returns the first student's ID.
5. **remove_student**(int student_id): Removes a student by ID from the hash table.

- Exception Handling: Throw a runtime_error if update_score, get_score, or remove_student is called on a non-existent student.

Part 2: StudentOrderedMap (SkipList)

Implement a skip list to maintain an ordered map of students based on their scores (keys), where the student ID is the value. This data structure allows efficient insertion and traversal, keeping students sorted by score. Here, scores (keys) can be duplicated.

- Data Structure: Skip List
  - Hint: use std::rand() in <cstdlib> for the implementation of flipping coins.
- Class: StudentOrderedMap
- You should properly handle duplicate keys.
- Required Methods:
  1. **Constructor** & **Destructor**
  2. **add_student**(int student_id, int score): Adds a student in sorted order by score.
  3. **update_score**(int student_id, int new_score): Updates the score and maintains the sorted order.
  4. **get_student**(int score): Retrieves the student ID associated with the given score. If there are multiple matches, it returns the lowest student ID.
  5. **remove_student**(int student_id): Removes a student and score by student ID.
- Exception Handling: Throw a runtime_error if update_score, get_score, or remove_student is called on a non-existent student.

Part 3: StudentDatabase

Utilize StudentMap and StudentOrderedMap classes to make a class StudentDatabase, which is a system, ensuring synchronized score updates across both structures with additional functionalities.

- Class: StudentDatabase
- Required Methods:
    1. **add_student**(int student_id, int score): Adds a student to both StudentMap and StudentOrderedMap.
    2. **update_score**(int student_id, int new_score): Updates a student's score in both data structures.
    3. **get_score**(int student_id): Retrieves a student's score from StudentMap.
    4. **get_student**(int score): Retrieves the student ID associated with the given score from StudentOrderedMap.
    5. **remove_student**(int student_id): Removes a student from both StudentMap and StudentOrderedMap.
    6. **get_top_k_students**(int k): Return the top k students by score in descending order. Implement this method by traversing StudentOrderedMap. If there are multiple students with the same score, lower student IDs get priority. Fill the remaining with {-1, -1}, if there are fewer than k students.
    7. **get_rank**(int score): Return the rank of a specific score, where rank 1 is the highest score. Implement this method by traversing StudentOrderedMap. If there are multiple students with the same score, apply this rule:
        a. All identical scores receive the same rank, and the next rank number continues sequentially. For example, if the scores are [100, 90, 90, 80], their ranks would be [1, 2, 2, 3].
- Exception Handling:
    - Ensure all exceptions raised by StudentMap and StudentOrderedMap are properly handled in StudentDatabase.
    - Throw a runtime_error if there's no score matched in get_rank.

Skeleton Code

We provide the following files for you to start with:

- main.cpp
- student_map.h
- student_map.cpp
- student_ordered_map.h
- student_ordered_map.cpp
- student_database.h

- student_database.cpp
- Makefile

Your job is to complete the implementation of all parts. **You <u>can change the skeleton code except for specified areas</u>** (e.g., some areas *.h files). We provide an example **Makefile** for compilation. You can add your own files if necessary, but note that you should explain them in ReadMe.txt and address them in your Makefile. You can test your code with main.cpp we provide, but the code written in main.cpp does not cover all the test cases in grading.

Submission

- **Submission Format**: Compress all the relevant files (*.cpp, *h, Makefile, ReadMe.txt, etc.) into a single **zip** file named **STUDENT_ID.zip** (e.g., 20242222.zip) and submit it.
    a. all necessary source files (*.cpp and *h)
    b. **Makefile**: Create your own Makefile to compile your project. You can use the Makefile we provided. This will be part of your grade, so make sure it compiles your project correctly when typing make in the terminal. See the "compilation" section in the grading criteria below for details.
    c. **ReadMe.txt**: Include a Readme file explaining any specific considerations for grading your project or any deviations from the instructions.
- **Deadline: 11th October 11:59PM**
    a. **Late Submission**: The late submission follows the policy that was explained in the course overview lecture (see lecture notes in Blackboard).

Restrictions

- **Allowed Libraries**: You are permitted to use **only** the following C++ standard libraries: **<iostream>**, **<string>**, **<exception>, <cstdlib>**.
- **C++ Version**: It is recommended that you use a recent version of C++ for this assignment. **Minimum Version: C++11**. **Recommended Version: C++17 or higher**.

Example Output

Below is the console output of running main.cpp with correct implementation.

```
Starting tests...
Testing StudentMap:
Score of 20241234: 95
Updated Score of 20241234: 98
Expected exception: Student ID not found

Testing StudentOrderedMap:
Student with score 88: ID 20244321
Updated ID 20241234 to score 99
Student with score 99: ID 20241234
Student with score 88 after removal: ID 20246666

Testing StudentDatabase:

Score of 20241234: 95
Updated Score of 20241234: 99
Top 5 students by score:
Student ID: 20241234, Score: 99
Student ID: 20245678, Score: 97
Student ID: 20241111, Score: 90
Student ID: 20242222, Score: 90
Student ID: 20249999, Score: 90

Rank of score 90: 3
Rank of score 88: 4
Expected exception: Score not found

After removing student ID 20245678:
Top 5 students by score:
Student ID: 20241234, Score: 99
Student ID: 20241111, Score: 90
Student ID: 20242222, Score: 90
Student ID: 20249999, Score: 90
Student ID: 20244321, Score: 88
Expected exception: Score not found
Rank of score 99: 1
Tests completed.
```

Grading Criteria

- **Submission Format**
  - It is mandatory to ensure the integrity of the provided header files. Your submission should follow the instructions in the “submission” section.
- **Compilation**
  - The “**make**” command MUST work to create the “**main**” program (see the example Makefile provided).
  - While you only need to submit part of the source files, assume that the provided *.h files and main.cpp will be included during compilation, so you don’t need to worry about compilation errors due to missing those files.
  - If compilation fails, the score will be zero.
  - If a runtime exception halts the program (e.g., a crash due to a segmentation fault), the score will be based on progress up to the point of termination.

- **Main Parts**
    - Various test cases will be used to assess the functionality of each requirement described in this document. Scores will only be given to correctly passed test cases.