CS293 Lab 2
Rail Planner

Introduction

Welcome to CS293: Data Structures and Algorithms Lab 2. The goal of this lab, and of the course, in general, is to make clever data structures to store and utilise data effectively. Remember, beyond some initial specifications, the problem statement, the approach and the expected result is very open ended. Feel free to unleash your creativity, within reason, and craft together something brilliant. Please read all instructions carefully. Now, let us come to the Problem Statement.

Problem Statement

Starting off with some context, train travel in many places around the world is very convenient. It is the brilliant and clever utilisation, storage and accessibility of data that makes it so. In this lab, Let us put our CS213 knowledge to use and strive to develop something along similar lines. The goal is to make a rail planner program which will be able to store data about train schedules and routes and be able to answer queries which users can make to plan their rail travel across complex rail systems.

Woah! Looks like we have a lot to do! So let us get right into it. We will split the problem into 2 major parts.

Part 1 - Dictionary

In the first part, you will implement the dictionary type data structure as you have studied in CS213.

- Implement a dictionary of strings (consider each string is null terminated, of max length 32 including the null). Use the dictionary to map a string to an integer, for now. This will change depending on your design for the 2nd part.
- The hash code map should use a polynomial accumulation scheme (be careful of overflow)
- The compression map should use the Fibonacci hashing scheme as mentioned in the lecture video
- Use a fixed length array for the dictionary. say 64 should be adequate.
- To resolve hash collisions, use the simple linear probing scheme, with appropriate NULL and TOMBSTONE markers (as described in the lecture slides)

Please write all code for this code in a *dictionary.cpp* file using the header file *dictionary.h*. Do NOT change the header file, beyond what's necessary. You can use *dictionaryTest.cpp* to test your dictionary.

Running Instructions

Follow the following instructions for testing your dictionary manually.

- Compile and run dictionaryTest.cpp in manual mode The Interactive part of the Testing can take these instructions
  - `QUIT`: end the interactive mode and exit the program
  - `INS <key> <value>`: Insert the key value pair into the Dictionary
  - `DEL <key>`: Delete the key from the dictionary if found
  - `FIND <key>`: Find and returns the key value pair

Follow the following instructions for testing your dictionary automatically. (These are optional to run)

- Run python sample.py to generate sample strings for testing.
- Compile and run dictionaryTest.cpp and redirect output to results.txt by using the command

```
./name-of-exec > results.txt
```

- Run python plot.py to generate the plot.

Part 2 - Handling Queries

With the appropriate data structures at our disposal, we have the necessary arsenal to jump into a model of the core algorithms behind rail planner systems. The key idea is to use the dictionary you implemented in the first part cleverly and possibly the queue you implemented in the first lab. Our goal is to write a planner class which can handle the following type of queries

- `ADD <station_one> <start_time> <station_two> <finish_time>`

  where you can treat the input to be a (string("ADD"), string, float, string, float) tuple (these strings have no whitespaces). This adds a train route from station one to station two starting at start time and ending at finish time. Note that train journeys from A to B to C will be stored as 2 different two different journeys, so do not worry about intermediate stations (for now).

- `QUERY_STATION <station_one> <start_time>`

where you can treat the input to be a (string("QUERY_STATION"), string, float) tuple (these strings have no whitespaces). This should list all trains (as a float, string tuple as start time and destination) starting from station one including and after start time. Display any suitable error message if the station does not exist.

- `QUERY_JOURNEY <station_one> <start_time> <station_two>`

  where you can treat the input to be a (string("QUERY_JOURNEY"), string, float, string) tuple (these strings have no whitespaces). This should list the soonest (least possible start time) possible journey (as the start time for a direct journey or a float, string, float tuple as start time, intermediate destination, leave time for a journey with one intermediate stop as the case may be) after start time from station one to station two with at most one stop in between (For instance, to go from A to C, A to B and B to C is acceptable but not A to B to D to C). Display any suitable error message if no such path exists. (Due to the restriction of at most one stop in between, no graph theory is required for this question)

- EXIT - Used to end the query loop

For Part 2, the code structure is completely up to you. You can change the header file as you like as long as you match the input-output specification.

Running Instructions

One sample test-case has been provided to you as sample-planner.txt and its corresponding output as sample-planner-out.txt. Simply run your code files taking in input from the sample-planner.txt file and compare the output with the output file.

Submission Instructions

Please follow the submission instructions clearly

- Submit all files in a single tar file. You can use the command

  ```
  tar -czvf <roll-no>-L2.tar.gz <name-of-folder>.
  ```

  Make a single submission ONLY.

- The first part should be completed in the lab timings while the second part can be completed until the deadline next week. Submit once both parts are completed