# Medical Tracker

CSCI 2270 Summer 2020: Final Project

Due: Friday, July 24, 11:59 PM MT

## Contents



## 1 Introduction

Medical Tracker Company is working on a contagion tracker for the latest contagion. The company needs your help to write software that will retrieve patient medical history. The application will use patient id's to retrieve the patient medical history. Thus, there is a need to store the id's in a data structure. Help the company organize these patient id's in a more optimal data structure.

## 2 The Objective

The data used for tracking patient data utilizes integer-based patient IDs. A local hospital has provided two sets of experimental data for us to test our algorithms. We need to perform analysis on both of these sets of data and find a data structure that strikes the best balance in run-time performance.

## 3 The Data Structures

In order to get a baseline for the experiment, you will first need to implement the Doubly Linked List. The candidate data structure to replace the Doubly Linked List is the Hash Table (multiple variants).

### 3.1 Doubly Linked List

Implement a class for doubly linked list. Your class should include at least insert, search, and display methods. The node definition should be defined as follows:

```
struct Node{
    int key;
    Node* next;
    Node* prev;
};
```

Your files should be named **exactly** as follows:

- dll.cpp
- dll.hpp
- dlldriver.cpp

### 3.2 Hash Table

Consult your lecture notes and Chapter 13 from the course textbook *Visualising Data Structure* (Hoenigman, 2015) for hash table reference. A sample .hpp file for hashing with chaining is provided to serve as a guide for your class implementations.

The hash function is to utilize the division method:

h(x) = x % m,

where x is the key value and m is the table size. Your table size should be set to 40009.

Next, implement three collision resolution mechanisms:

- Open addressing: linear probing
- Open addressing: quadratic probing
- Chaining with a linked list

For the open addressing variant, use a circular array mechanism for collision resolution, so as to avoid writing records outside of the array bounds. For example, consider that a key value hashes to index 40008, but that table location is already occupied by another record. In order to resolve the collision with linear probing, you increment the index value by 1. However, this results in index of 40009, which is outside of the array bounds. Instead, the next place in the table your algorithm checks should be at index 0. If index 0 is occupied, then check index 1, and so on.

Your files should be named **exactly** as follows:

- Hash Table with chaining
    - hashchain.cpp
    - hashchain.hpp
    - hashchaindriver.cpp
- Hash Table with open addressing, linear probing
    - hashlinear.cpp
    - hashlinear.hpp
    - hashlineardriver.cpp
- Hash Table with open addressing, quadratic probing
    - hashquad.cpp
    - hashquad.hpp
    - hashquaddriver.cpp

# 4 Experiment and Analysis

Two sets of data are provided in dataSetA.csv and dataSetB.csv. Perform the following experiment on each of the data structures (and all its variants), once for each data set.

## 4.1 Insert and search

Perform the following experiment for each data structure.

There are 10,000 elements provided in each data file, so declare an integer type array of length 10,000 (e.g. int testData[10000];). Also, declare two float type arrays of length 100 to record the time measurements (e.g. float insert[100]; and float search[100];)

Note: you are not allowed to shuffle your data when performing inserts.

1. **Set up test data.** Read in the entire test data into the array of integers.

2. **Insert.** Measure the total amount of time it takes to insert the first 100 records from dataSetA.csv. Divide the total time by 100 to get the average insert time. Record this value in insert[0].

3. **Search.** We want to run the search experiment such that every time we search for a value, it is guaranteed to already be present in our data structure. To this end, generate a set of 100 pseudo-random numbers in the interval of $[0, 99]$. Use these values as indices into your test data array. Search your data structure for each of the 100 elements, measuring the total time it takes to perform the 100 searches. Divide the total time by 100 to get the average search time. Record this value in search[0].

4. **Insert.** Measure the total amount of time it takes to insert the next 100 records from dataSetA.csv. Divide the total time by 100 to get the average insert time. Record this value in insert[1].

5. **Search.** Generate a set of 100 pseudo-random numbers in the interval of $[0, 199]$. Use these values as indices into your test data array. Search your data structure for each of the 100 elements, measuring the total time it takes to perform the 100 searches. Divide the total time by 100 to get the average search time. Record this value in search[1].

6. Continue to interweave the insert and search operation sets until you reach the end of the data file (there are 10,000 records in the file, so your number of deltas should be 100).

7. Record this data to an external data file so that it can be plotted later (e.g. insert_search_performance_doubly_ll_dataSetA.csv).

**After running the experiment on each of the data structures with data from dataSetA.csv, repeat for every data structure on dataSetB.csv.**

## 4.2 Visualising the Results

First of all, you should try to figure out whether the two provided experimental data sets have certain significant qualities that would affect which data structures would perform better or worse. Plot the two data sets to get a visual understanding. You can use a program of your choice for generating the plots (Excel, MATLAB, Python, etc.).

Next, we need to compare the performance of all the different methods. We can plot the insert data and search data for each data set in a single figure.

For the hash tables, additionally include a second vertical scale to show the number of collisions per 100 operations. Include plots for both insert-collisions and search-collisions (your hash table figures should have 4 plots each).

The list of figures to generate for this report is:

- Doubly Linked List: one figure for dataSetA, one figure for dataSetB
- Hash Table with chaining: one figure for dataSetA, one figure for dataSetB
- Hash Table with open addressing, linear probing: one figure for dataSetA, one figure for dataSetB
- Hash Table with open addressing, quadratic probing: one figure for dataSetA, one figure for dataSetB
- A summary figure, for the inserts. Here you should pick the best of the hash table results. Plot the inserts for the chosen hash table and the doubly linked list.
- A summary figure, for the searches. Again, pick the best of the hash table results. Plot the inserts for the chosen hash table and the doubly linked list.

As an example, your Doubly Linked List plots should look similar to figure 1. For your final report, you should have a single plot showing the given data, then each of the figures described above, resulting in a total number of 10 figures.

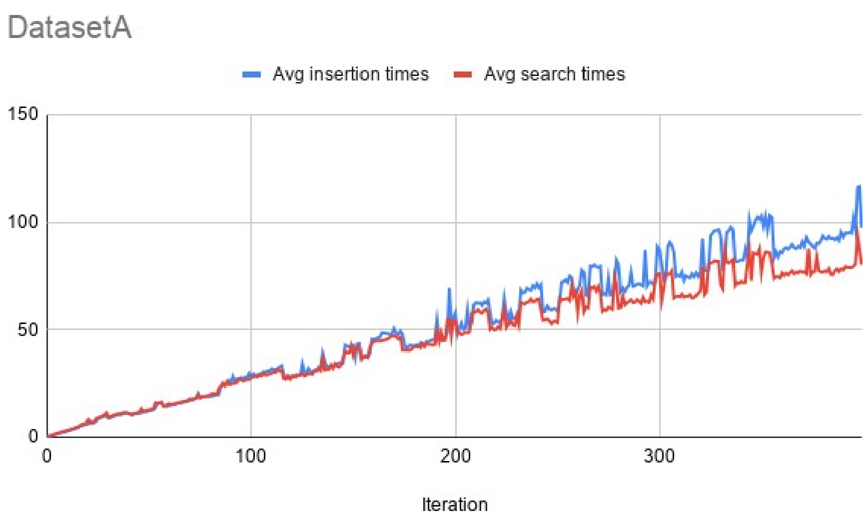


Figure 1: Average insertion and search times for a Doubly Linked List with Data Set A

## 4.3 The Report, Submission, and Interview Grading

Provide a concise summary of your findings in a report along with all of your figures. Describe which data structure you find to be the best for the Medical Tracking application. Refer to the different data types and provide some hypothesis as to why the different data structures perform better or worse. Your report can take up multiple pages due to the figures, but try to limit your write-up to 250 words.

Submit all of your neatly commented code for each data structure implementation, along with a driver file for each experiment. You will not use Coderunner for this project. Instead, we will manually compile and run your programs. Your programs must be named **exactly** as follows. We will run scripts to grade your submissions so incorrect file naming will result in your programs not being graded.

- Doubly Linked List
    - dll.cpp
    - dll.hpp
    - dlldriver.cpp
- Hash Table with chaining
    - hashchain.cpp
    - hashchain.hpp
    - hashchaindriver.cpp
- Hash Table with open addressing, linear probing
    - hashlinear.cpp
    - hashlinear.hpp
    - hashlineardriver.cpp
- Hash Table with open addressing, quadratic probing
    - hashquad.cpp
    - hashquad.hpp
    - hashquaddriver.cpp

Finally, you must work on this project individually. **There will be mandatory interview grading.** If a student does not complete the interview grading, the student's project will result in a score of 0. So, it is the students' responsibility to schedule an interview. Scheduling registration links will be provided via Moodle.