Medical Tracker

CSCI 2270 Fall 2020: Final Project
Due: Sunday, December 6, 11:59 PM MDT

Contents

Part A

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 a data set of experimental data for us to test our algorithms. We need to perform analysis on this set 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 structures to replace the Doubly Linked List and the Hash Table.

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:

```
1  struct Node{
2        int key ;
3       Node * next;
4        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 is provided to serve as a guide for your class implementation.

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 the following collision resolution mechanism:

- Open addressing: quadratic probing

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. Also, assume we are using linear probing. In order to resolve the collision, we would increment the index value by 1. However, this results in an 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. You will need to apply the equivalent for quadratic probing.

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

- Hash Table with open addressing, quadratic probing
    - hashquad.cpp
    - hashquad.hpp
    - hashquaddriver.cpp

4 Experiment and Analysis

A set of data is provided in dataSetA.csv. Perform the following experiment on each of the data structures for the data set.

4.1 Insert and Search

Perform the following experiment for each data structure.

There are 10,000 elements provided in the 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).

4.2 Visualizing the Results

Plot the data set to get a visual understanding of how the data structures perform. You can use a program of your choice for generating the plots (Excel, MATLAB, Python, etc.). We can plot the insert data and search data for each data set in a single figure.

For the hash table, 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 figure should have 4 plots).

The list of figures to generate for this report is:

- Doubly Linked List: one figure for dataSetA
- Hash Table with open addressing, quadratic probing: one figure for dataSetA
- A summary figure, for the inserts. Plot the inserts for the hash table and the doubly linked list.
- A summary figure, for the searches. Plot the inserts for the 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 4 figures.

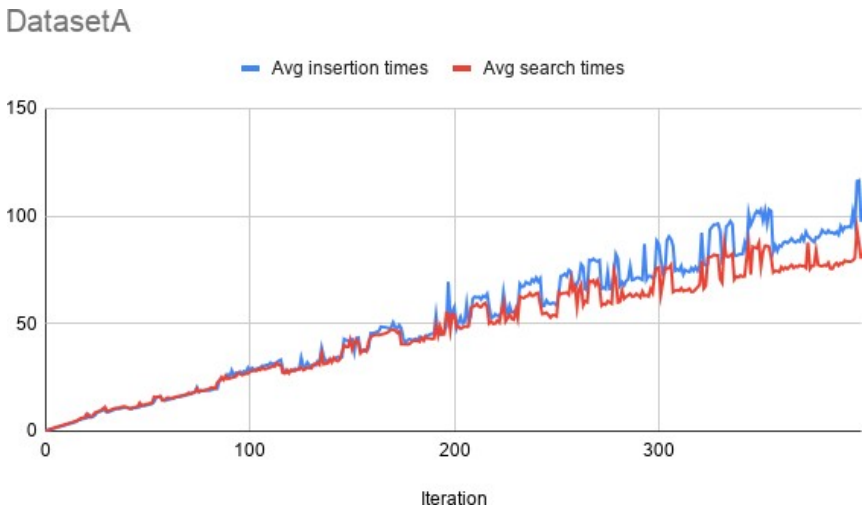

Figure 1: Average insertion and search times for a Doubly Linked List with Data Set A

4.3 The Report, Submission, and Interview Grading

Provide a concise summary of your findings in a report along with your figures. Describe which data structure you find to be the best. Refer to the different data types and provide some hypothesis as to why the different data structures perform better or worse. Explain your findings in no less than 150 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 open addressing, quadratic probing
  - hashquad.cpp
  - hashquad.hpp
  - hashquaddriver.cpp

Finally, you must work on this project individually and follow the coding policies laid out in the Syllabus. All code must be your own. **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 by your TA.

Part B

1 The Objective

Medical Tracker Company would like you to improve the software application by sorting the patient id's. They would like you to compare a couple of different algorithms to accomplish this. Help the company sort the patient id's in a more optimal data structure.

2 The Data Structures

In order to get a baseline for the experiment, you will first need to implement Bubble Sort. The candidate data structure to replace the Bubble Sort is the Heap Data Structure.

2.1 Bubble Sort

Implement a class for bubble sort to organize the patient id's. Your class should include at least insert, search, display, and sort methods.

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

- bubblesort.cpp
- bubblesort.hpp
- bubblesortdriver.cpp

2.2 Heap

Implement a heap data structure and use heap sort to organize the patient id's. You can either use min-heap or max-heap. Your class should include at least insert, search, display, and heapify methods. Do not use the STL library.

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

- heapsort.cpp
- heapsort.hpp
- heapsortdriver.cpp

3 Experiment and Analysis

Use the test data (dataSetA.csv) from Part A. Perform the following experiment on each of the data structures.

3.1 Insert and Sort

Perform the following experiment for each data structure.

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

sort[100]).

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

2. **Insert.** Insert/Copy the first 100 elements of the testData array to a temporary array of size 100 (e.g. tmpArr[100]).

3. **Sort.** Sort the 100 elements from the previous step (in the temporary array). Measure the total amount of time it takes to sort the elements. Record this value as sort[0]. Note: do not time the copy operations from testData[] to the temporary array. Only measure the sort time.

4. **Insert.** Insert/Copy the first 200 elements of the testData array to a temporary array of size 200 (e.g. tmpArr[200]).

5. **Sort.** Sort the 200 elements from the previous step (in the temporary array). Measure the total amount of time it takes to sort the elements. Record this value as sort[1]. Note: do not time the copy operations from testData[] to the temporary array. Only measure the sort time.

6. Continue to interweave the insert/copy and sort 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). That means n = 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, … , 10,000.

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

3.2 Visualizing the Results

We need to compare the performance of the different sorting methods.

The list of figures to generate for this report is:

- Bubble Sort: one figure for dataSetA displaying the times to sort the 100 plot points (i.e. the sort array).
- Heapsort: one figure for dataSetA displaying the times to sort the 100 plot points (i.e. the sort array).

3.3 The Report, Submission, and Interview Grading

Provide a concise summary of your findings in a report along with your figures. Describe which data structure you find to be the best. Refer to the different data types and provide some hypothesis as to why the different data structures perform better or worse. Explain your findings in no less than 150 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.

- Bubble Sort
    - bubblesort.cpp
    - bubblesort.hpp
    - bubblesortdriver.cpp
- Heapsort
    - heapsort.cpp
    - heapsort.hpp
    - heapsortdriver.cpp

Finally, you must work on this project individually and follow the coding policies laid out in the Syllabus. All code must be your own. **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 by your TA.