CSCI 2270: Data Structures – Project Specifications
University Database

Due Sunday, December 4 @ 11:59 PM MT

1 Introduction

The objective of this project is to write an application that reads a list of University courses from a data file and then inserts them into two separate hash tables. The first hash table will use the Open Addressing collision resolution scheme and the second hash table will use the Chaining collision resolution scheme. Users should be able to search the hash tables for a specific course number or display all of the courses. It's your objective to compare and contrast the hashing collision resolution methods. Additionally, the application should maintain a list of courses taught by each professor using a binary search tree data structure. Users should be able to search for a professor and view all of the courses taught by the professor.

2 Core Features

2.1 Overview

You have been provided with starter code. The contents of the starter code are as follows:

- main.cpp
  - Driver program
- util.h
  - Used for utility methods and structures (contains the *Course* structure that's used by both hashing collision schemes)
- HashOpenAddressing.cpp
  - Used to maintain course records using Open Addressing with Quadratic Probing
- HashOpenAddressing.h
- HashChaining.cpp
  - Used to maintain course records using Chaining with a Doubly Linked List
- HashChaining.h
- ProfBST.cpp
  - Used to maintain a Binary Search Tree of courses taught by every professor
- ProfBST.h

You are required to use these files and not add any additional ones but are more than welcome to add helper functions as needed.

2.2 Implementation Requirements

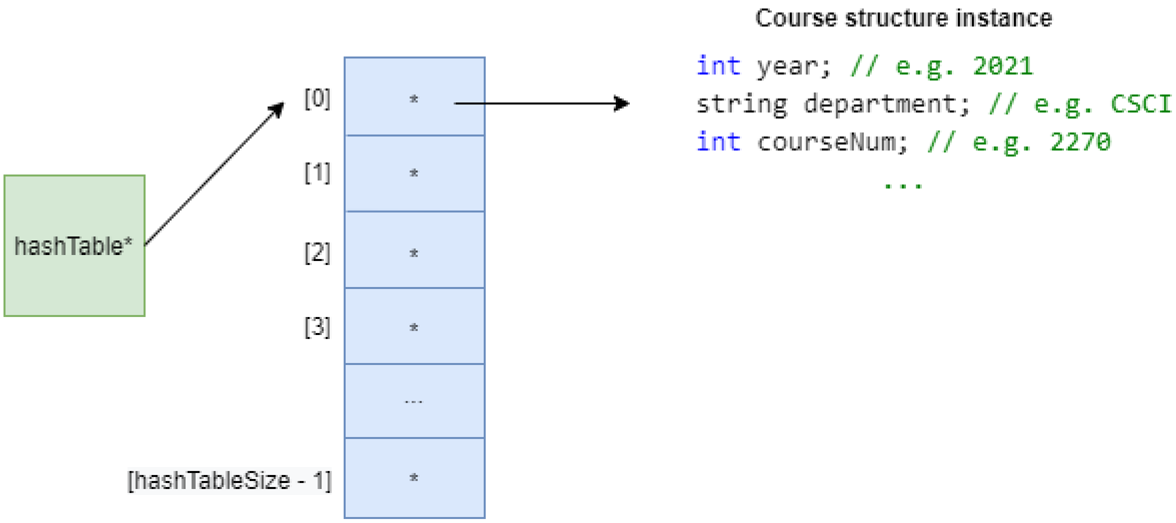
2.2.1 Preliminary Steps

Your *main()* function should accept three command-line arguments: the program name, csv input file, and the hash table size. If the user doesn't execute the program using the correct format, prompt them to run the program again and display the usage. For example:

```
Invalid number of arguments.
Usage: ./<program name> <csv file> <hashTable size>
```

Note: you will be provided with 2 csv files to test against: *cscicoursesA.csv* and *cscicoursesB.csv*.

Next, create 2 hash tables and set their sizes to the number of items in their respective csv files, which is the hash table size argument that was passed in as an argument. One of the hash tables will represent a hash table that uses Open Addressing collision resolution with Quadratic Probing whereas the other hash table will use Chaining collision resolution with a doubly linked list. Name the objects accordingly so that it is easy to distinguish between them. The constructor will need to use a double pointer, so it will be arranged as a pointer (*hashTable*) to a hash table of size *hashTableSize* that consists of pointers to courses, which are instances of the struct *Course*. Figure 1 represents how the layout for *Course **hashTable* should be implemented.



```
int year; // e.g. 2021
string department; // e.g. CSCI
int courseNum; // e.g. 2270
...
```

Figure 1: hashTable Double Pointer Layout

2.2.2 User Interface

Once the program is executed, display a menu that looks as follows:

```
=======Main Menu=======
1. Populate hash tables
```

2. Search for a course
3. Search for a professor
4. Display all courses
5. Exit

This menu should be displayed to the user after every input so that the user can continuously run menu options. Each menu item is detailed below.

Each of the options within this menu will call functions for the respective hashing collision programs. For example, option 1 (Populate hash tables) will call *HashOpenAddressing::bulkInsert()* and *HashChaining::bulkInsert()*.

2.2.3 Populate Hash Tables

This option must be selected first in order for any of the other menu options to function. **It is assumed that, each time you run your program, option 1 will be executed once.**

You will be provided with 2 csv files to test against: cscicoursesA.csv and cscicoursesB.csv. Parse the file that was passed into the program, create a Course instance for every course, and then insert each course one by one into the respective hash table.

The first hash table will use the Open Addressing collision resolution whereas the second hash table will use the Chaining collision resolution. The Open Addressing scheme must use Quadratic Probing.

newIndex = (index + i * i) % hashTableSize

The Quadratic Probing hash function should utilize the division method where *x* is the key (e.g. 2700 for csci2270) and *hashTableSize* is the number of course records in the csv file:

h(x) = x % hashTableSize

For the Open Addressing variant, use a circular array mechanism for collision resolution, so as to avoid writing records outside of the array bounds.

The chaining scheme should use a Doubly Linked List, so you will need *previous* and *next* pointers. A depiction of how the Chaining hash table should look is in Figure 2.

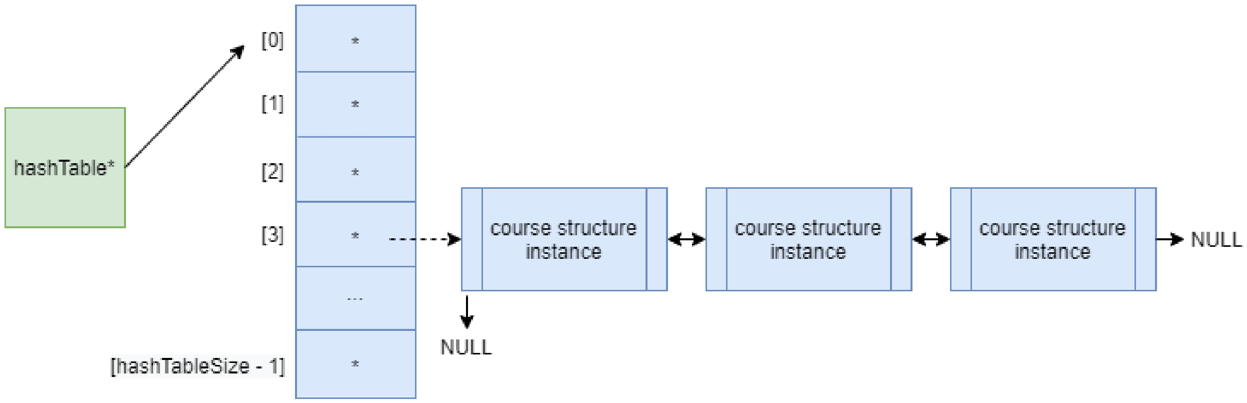

hashTable*
[0]
[1]
[2]
[3]
[hashTableSize - 1]
*
*
*
*
...
*
course structure instance
course structure instance
course structure instance
NULL
NULL

Figure 2: Chaining Method Layout

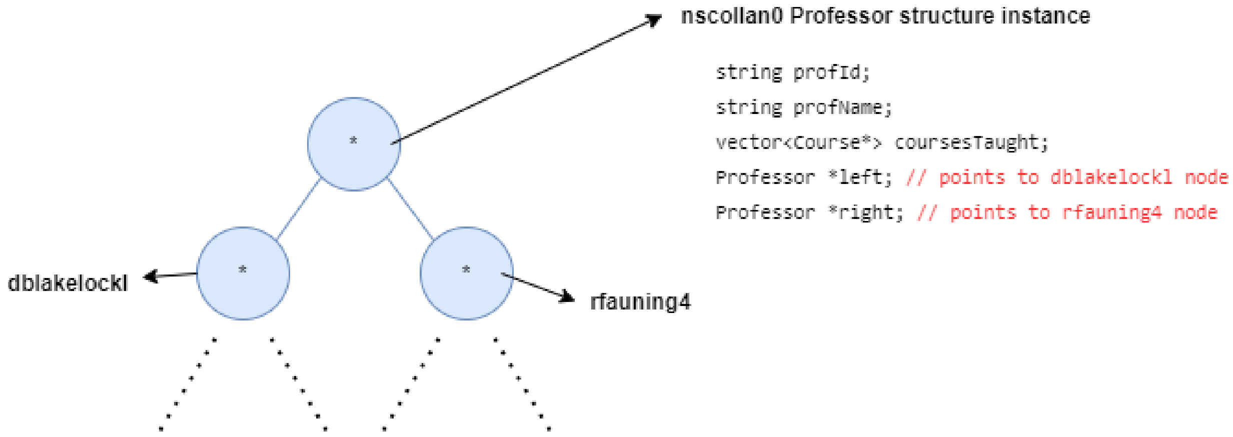
Every time there is a collision for either hashing collision resolution method, increment the respective method's collision counter. Additionally, every time you search for an index to insert a value, increment the number of searches. For example, if there's a collision and it takes 5 (quadratic) probes to find an empty bucket, then the number of searches will amount to 5. Display both the **total** number of collisions and the **total** number of searches for each hashing method, so you will need to keep running total of each one until you're finished reading all records from the csv file.

```
Enter an option: 1

[OPEN ADDRESSING] Hash table populated
------------------------------------------------------
Collisions using open addressing: 397
Search operations using open addressing: 7135

[CHAINING] Hash table populated
------------------------------------------
Collisions using chaining: 386
Search operations using chaining: 2614
```

In addition to populating the hash tables with courses, you will need to perform the following while reading each course record from the csv file. Both hashing methods will need to populate their own Binary Search trees with all of the professors and the courses they have taught. To accomplish this, both hash header files (classes) contain a *profDb* object that you will use to access methods within *ProfBST*. From *bulkInsert()*, call the *ProfBST.addProfessor()* method to create an instance of the *Professor* struct (see *ProfBST.h*) and insert a pointer to the *Professor* struct instance into the BST. In other words, each BST node will be a pointer to a Professor object instance (see Figure 3). Use the *root*, *left*, and *right* pointers (specified within *ProfBST.h*) to build your BST. You will need helper methods within *ProfBST*.

Do not insert a professor into the BST twice. To prevent this, before adding a professor to the BST, check if he/she exists by searching for the professor in the BST via their id (e.g. nscollan0). Whether or the professor already exists in the BST or needs to be added to the BST, the course (*Course* instance) needs to be added (pushed) to *vector<Course*> coursesTaught*. The same vector is used for each respective professor. For example, if nscollan0 appears in the csv file 14 times, then her *coursesTaught* vector will contain 14 pointers to instances of *Course*.

Also, you are **not** required to balance the BST or perform rotations. Simply insert the pointer to the respective professor struct instance into the BST based on the professor's id. For example, in Figure 3, *nscollan0* is a parent to *rfauning4* since *n* comes before *r*.



```
string profId;
string profName;
vector<Course*> coursesTaught;
Professor *left; // points to dblakelockl node
Professor *right; // points to rfauning4 node
```

Figure 3: BST Layout

2.2.4 Search For A Course

Prompt the user to enter a course year (e.g. 2021), course number (e.g. 2270), and a professor's ID (e.g. nscollan0). Search for the course in both hash tables by calling the two search functions *HashOpenAddressing.search()* and *HashChaining.search()*. For open addressing, you will use Quadratic Probing to search for the course, which should be able to probe all of the buckets within the hash table. For chaining, you will search through the doubly linked list. In order to find a course, you will need to check that the course year, course number, and professor's ID all match.

If the course is found, display the number of searches it took to find the course. Whether the course was found or not, *search()* should call *displayCourseInfo()*. In *displayCourseInfo()*, display a message to the user if the course wasn't found. If the course was found, display the course year, course number, course name, and professor's full name.

=======Main Menu=======
1. Populate hash tables
2. Search for a course
3. Search for a professor

```
4. Display all courses
5. Exit

Enter an option: 2

Enter the course year (e.g. 2021):
2021
Enter a course number (e.g. 2270):
1300
Enter a Professor's ID (e.g. llytellf):
rfauning4

[OPEN ADDRESSING] Search for a course
------------------------------------
Search operations using open addressing: 2
2021 COMP SCI I: PROGRAMMING 1300 Rene Fauning

[CHAINING] Search for a course
------------------------------------
Search operations using chaining: 1
2021 COMP SCI I: PROGRAMMING 1300 Rene Fauning
```

2.2.5 Search For A Professor

This menu option will display all of the courses taught by a professor by searching the BST and then displaying the professor's *coursesTaught* vector. First, prompt the user to enter a professor id and then call *ProfBST::publicSearchProfessor()*. Next, call helper functions to loop through your BST and locate the respective professor (his or her BST node that contains a pointer to the professor's Professor struct instance). If the professor isn't found, display a message to the screen (e.g. "Professor not found.") else use the Professor pointer to display the contents of the *coursesTaught* vector. Sample output is below.

```
=======Main Menu=======
1. Populate hash tables
2. Search for a course
3. Search for a professor
4. Display all courses
5. Exit

Enter an option: 3
Enter a Professor's ID (e.g. nscollan0):
nscollan0
```

```
[OPEN ADDRESSING] Search for a professor
---------------------------------------
Name: Nicky Scollan
- 1300: COMP SCI I: PROGRAMMING, 2021
- 3287: DATABASES INFO SYSTEMS, 2021
- 5454: DESIGN ANLYS OF ALGOS, 2020
- 7000: TP-INFORMATION RETRIEVAL, 2019
- 5832: NATURAL LANG PROCESSING, 2018
- 5622: MACHINE LEARNING, 2017
- 5454: DESIGN ANLYS OF ALGOS, 2016
- 5817: DATABASE SYSTEMS, 2014
- 6000: Introduction to the Computer Science PhD Program, 2013
- 4113: Unix System Administration, 2012
- 7000: Current Topics in Computer Science, 2012
- 1000: Computer Science as a Field of Work and Study, 2011
- 4830: Special Topics in Computer Science, 2010
- 3753: Operating Systems, 2006


[CHAINING] Search for a professor
---------------------------------------
Name: Nicky Scollan
- 1300: COMP SCI I: PROGRAMMING, 2021
- 3287: DATABASES INFO SYSTEMS, 2021
- 5454: DESIGN ANLYS OF ALGOS, 2020
- 7000: TP-INFORMATION RETRIEVAL, 2019
- 5832: NATURAL LANG PROCESSING, 2018
- 5622: MACHINE LEARNING, 2017
- 5454: DESIGN ANLYS OF ALGOS, 2016
- 5817: DATABASE SYSTEMS, 2014
- 6000: Introduction to the Computer Science PhD Program, 2013
- 4113: Unix System Administration, 2012
- 7000: Current Topics in Computer Science, 2012
- 1000: Computer Science as a Field of Work and Study, 2011
- 4830: Special Topics in Computer Science, 2010
- 3753: Operating Systems, 2006
```

2.2.6 Display All Courses

Prompt the user to enter which hashing scheme they want to display the records for. Call *displayAllCourses()* and display the course year, course name, course number, and professor for each course on one line.

```
=======Main Menu=======
1. Populate hash tables
2. Search for a course
3. Search for a professor
4. Display all courses
5. Exit

Enter an option: 4
Which hash table would you like to display the courses for (O=Open Addressing, C=Chaining)?
O

[OPEN ADDRESSING] displayAllCourses()
--------------------------------
2006 Computer Science 1: Programming 1300 Liz Naris

2012 Computer Science 1: Programming 1300 Ashlen Tromans

2007 Computer Science as a Field of Work and Study 1000 Fidela Hamor

2019 FUND CONCEPTS/PROG LANG 5535 Zorah Anscott

2014 FUND CONCEPTS/PROG LANG 5535 Zorah Anscott

2013 Introduction to Artificial Intelligence 3202 Rene Fauning

2006 Operating Systems 3753 Nicky Scollan

2017 NEW PHD 6000 Imogen Fredi

2018 COMP SCI I: PROGRAMMING 1300 Alanah Dummigan

2012 Current Topics in Computer Science 7000 Nicky Scollan

2015 COMP SCI II: DATA STRUCT 2270 Daisey Blakelock
```

…

(all course records will display)

2.2.7 Exit

When the user decides to exit, your program will call the destructor. Your destructor should free all memory from the hash tables, the BST, and finally, exit the program. Your program should not have any memory leaks. We will use Valgrind to detect memory leaks. You can use the Debugging Guide provided earlier in the semester for instructions on how to do this.

2.2.8 Miscellaneous

You must utilize, at a minimum, **everything** that exists in the starter code. For example, your solution must use a double pointer and you must use all of the provided functions. The parameter types should not be modified either. Your task is to populate the functions within the starter code but you are welcome to add helper functions as needed.

Secondly, your code should be well-documented. Every file should have a commenting section at the top and every function should have a comment section above it stating the name of the function, its purpose, the input parameters, and the output parameters. Consider using the Google C++ Style Guide. A common style used for functions is as follows:

```
/**
 * hash - calculates the hash of a key
 *
 * @param courseNumber
 * @return int
 */
int HashChaining::hash(int courseNumber) {
   int hash = courseNumber % hashTableSize;
   return hash;
}
```

3 Project Submission and Grading

3.1 Deliverables

Push your solution in the following files to the **master branch of your GitHub repo by the due date**.

- main.cpp
- util.h
- HashOpenAddressing.cpp
- HashOpenAddressing.h
- HashChaining.cpp
- HashChaining.h
- ProfBST.cpp
- ProfBST.h
- Edit the README.md file within your GitHub repo and explain the following:
    - Does one hashing collision resolution work better than the other? Consider the datasets we used and a much larger dataset with 1 million records. Explain your answer.
    - We used the same BST for both hashing mechanisms. What alternative data structure would you advise? Why?
    - Explain a few ways you could improve the application.

3.2 Interview Grading

There will be mandatory interview grading for this project. It is the students' responsibility to schedule an interview with **their** TA (scheduling links will be provided to you at a later date). We expect that you will be able to successfully answer all questions asked of you. If a student does not complete the interview grading, they will receive a 0 as their project score.

3.3 Coding Conduct

You are required to work individually on this project. All code must be written on your own. You are not allowed to use any code from the Internet. See the Collaboration Policies section within the Syllabus for more information.