CSCI 2270 Spring 2024 Project

Due Sunday, April 28 @ 1159PM MDT

ANY FORMS OF PLAGIARISM OR USE OF CHATGPT WILL RESULT IN A 0 AND BE REPORTED TO THE UNIVERSITY. DURING INTERVIEW GRADING, YOU BE ABLE TO ANSWER ANY AND ALL QUESTIONS ASKED OF YOU. INABILITY TO ANSWER QUESTIONS MAY RESULT IN FURTHER STEPS TO VERIFY THAT YOU WROTE YOUR OWN CODE.

Learning Objectives:

- Hashing and Chaining
- Heaps and Priority Queues

Grading policy

- You are required to schedule a mandatory Interview Grading session with course staff. A sign-up sheet will be posted via a Canvas announcement by 4/22. Interview Grading will be conducted between 4/29 and 5/2. Inability to complete Interview Grading or sign up on time will result in a 0.
- Projects that do not compile will receive a max possible score of 50 and that assumes that the program features have been implemented.

Submission Guidelines

- Step 1 – Submit/push your final solution to your Github repo by the due date
- Step 2 – Zip everything in your GitHub repository solution (.zip extension) and upload it to the Canvas Final Project dropbox by the due date (located under Week 13 within Canvas)

Interview Grading

- Come prepared by having your solution open and read to compile so that you can run through the menu options one by one. Also, you should have your GitHub page open.
- Expect to answer a variety of questions about your code as well as conceptual questions pertaining to hashing, heaps, and priority queues.
- Your TA may show you a visual/diagram of a data structure and expect you to answer questions about it, such as where the items in the visual are in your source code/solution and how you implemented them. Therefore, you must be prepared to draw diagrams exhibiting that you understand your code.
- You will not be allowed to read off of prepared reviews.

Please read this document carefully before you begin working.

Introduction

In this project you will create and maintain a data structure for efficient storing, retrieving, and manipulating of restaurant reviews. Each restaurant review consists of four items:

1. **restaurantName:** name of the restaurant
2. **review:** review/review about the restaurant
3. **customer:** store the customer's name
4. **time:** stores information about when the review was made by the customer (time id in 24h format)

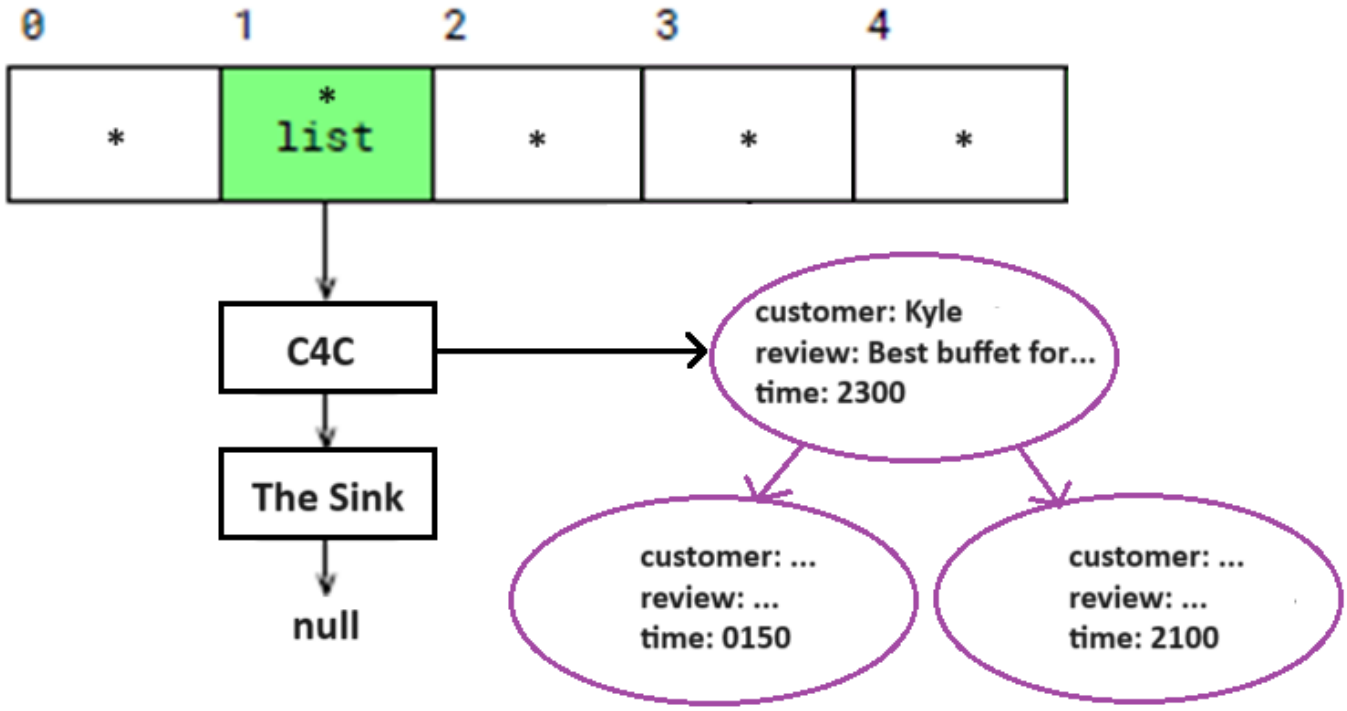
Here is one example of a restaurant review:

restaurantName: McDonald's review: I absolutely love McDonald's and would give it a 5-star rating! customer: Billy Bob time: 1603

You will build a hash table that uses chaining for collision resolution with a Linked List as the backing data structure. The Linked List will store the restaurants. You will also create a priority queue implemented using a heap to store the restaurant reviews.

Overview of the Data Structure

Store the restaurant reviews in a hash table that uses chaining. Each node in the chain will represent a distinct restaurant that has the same hash values. For example, ‘M’ and ‘The Sink’ will have the same hash value and they will reside in the same chain. With each restaurant node there will exist a Priority Queue. You need to organize the review information for a particular restaurant in the Priority Queue of the node based on the time. The most recent review info should be at the front of the queue.



During hashing, the hash function will take the restaurantName and will return the bucket index in the hash table. Based on the index, you must retrieve the head for the appropriate chain. Each restaurant will have a distinct node in the chain. Therefore, find the node with the current restaurantName and then depending on the purpose, manipulate the node. Note that, as mentioned above and depicted in the image, each node will have a Priority Queue associated with it. Node structure (**`struct node`**) is defined in hash.hpp as pictured above.

```
struct node {
   string restaurantName;
   PriorityQ pq;
   struct node* next;
};
```

Priority Queue

Each node within the hash table chain will store a priority queue containing information each of the reviews. In other words, each review is stored as an instance of the structure ReviewInfo. The queue will be organized on the basis of the data member *time* within ReviewInfo. Higher time values signify higher priority. Therefore, choose a proper heap implementation (i.e. max-heap vs min-heap). You will use an array-based implementation to build the heap. Build the array on the heap via the constructor. Note each element of the array is of type ReviewInfo.

- `PriorityQ(int capacity)`: constructor
- `~PriorityQ()`: destructor
- `int parent(int index):` returns parent index for an index
- `int leftChild(int index):` return the index of the left child for the given index
- `int rightChild(int index):` return the index of the right child for the given index
- `void peek():`
    - If queue is empty print `"no record found" << endl;`
    - Else print the information about the item (`rw`) at the top (item with highest priority)
        - `cout << "Restaurant: "<< rw.restaurantName << endl;`
        - `cout << "Customer:" << rw.customer << endl;`
        - `cout << "Review:" << rw.review << endl;`
        - `cout << "Time:" << rw.time << endl`
- `void heapify(int index):` maintain the heap as per the priority
- `void pop():` remove the top priority element from the priority queue. After popping you need to make sure that the heap property is maintained.
- `void insertElement(ReviewInfo value)`: If the currentSize reaches the capacity print the following: `"Maximum heap size reached. Cannot insert anymore."<<endl;` Else insert an element in the priority queue. After insertion you need to make sure that the heap property is maintained.
- `void print():` print the contents of the queue. Go over all the elements array and print the following:
    - `cout << "\t" << "Customer: " << heapArr[i].customer << endl;`

- `cout << "\t" << "Review: " << heapArr[i].review << endl;`
- `cout << "\t" << "Time: " << heapArr[i].time << endl;`
- `cout << "\t" << "=====" << endl;`

- You will use the following structure (defined in PriorityQueue.hpp):

```
struct ReviewInfo {
   string restaurantName;
   string review;
   string customer;
   int time;
};
```

Hashing

For this project you are required to implement hashing with chaining for collision resolution. Hash.cpp should implement all the functions required for hashing. Please refer to hash.hpp for the class definition and function declaration. Each node will contain a priority queue to store the information such as reviews, customers, and time. The *node* structure was discussed above and can be found in hash.hpp.

HashTable will have a dynamically allocated array pointer given as `node**table`. Note, the array will store the heads of the linked list chains. Therefore, each element of the array is of type `node*` and the other '*' represents the fact that an array is created with a pointer (recall how with a pointer we created dynamic memory allocation for arrays).

The class of HashTable will also store a variable for `numCollision` to keep track of the number of collisions. Remember, if two keys x and y are such that x!=y but hash(x)==hash(y), then it is a collision. For example, assume you add a review for '`McDonald's`' and then you add a second review for '`McDonald's`'. In this case, there is no collision as adding multiple reviews into the priority queue for the same restaurant node doesn't count as a collision. Next, assume you enter in a record for '`Chiptole`'. Since hash(`McDonald's`) == hash(`Chipotle`), you now have a collision. So, the variable `numCollision` should be incremented accordingly. Beyond this, any other insertion of '`McDonlad's`' or '`Chipotle`' will not change `numCollision.`

Member functions for HashTable are listed as follows-

- `HashTable(int bsize)`: This is the constructor. Create a HashTable of size bsize.
- `~HashTable()`: This is the destructor. Free any memory that was allocated on the heap.
- `node* createNode(string restaurantName, node* next)`: This function will create a linked list node for the chain with restaurantName. During creation of the node, create the priority queue of size 50. You will call it from the insertItem function.
- `unsigned int hashFunction(string restaurantName)`: This function calculates the hash value for a given string. To calculate the hash value of a given string, sum up the ascii values of all the characters from the string. Then take the% operator with respect to tableSize.

$$hashFunction(S) = (\sum_{ch \in S} ch)\%tableSize$$

- `node* searchItem(string restaurantName)`: Search for a restaurantName in the hash table. If found return the node, else return null.
    a. Calculate the hash function for the argument restaurantName.
    b. Retrieve the chain head from the table
    c. Traverse the chain to find the node containing the restaurantName.
    d. On successful search return the node else return null.
- `void insertItem(ReviewInfo restaurant)`: Insert a restaurant intto the hash table.
    a. First, search for the node with restaurant.restaurantName
    b. If a node exists in the hash table for that restaurant, insert the ReviewInfo restaurant into the priorityQueue of that node.
    c. If the search method returns a null then
        - Create a new node for that restaurant.restaurantName.
        - Add the ReviewInfo restaurant in the priorityQueue of that node. Use `insertElement` of the priority queue.
        - If the corresponding chain head in the hashTable bucket is null then this is the first node of that chain. So, update the chain head in the table accordingly.
        - If the corresponding chain head in the hashTable bucket is not null, then this is a collision. Update the numCollision. And add the node to the start of the Linked List and update the update the chain head in the table.
- `void setup(string fname)`:This function will populate the hash table from a file. Path of the file will be passed from driver code. The file path will be passed as fname. The file is ; separated file and the format of a line is-

    restaurantName;review;customer;time

    For example:

McDonald's;I absolutely love McDonald's and would give it a 5-star rating!;Billy Bob;1603

Inside the setup function you will read each line from the file. For each line of the file, create a ReviewInfo structure instance out of the line you just read from the file. Call the insertItem function with the instance of ReviewInfo.

- `int getNumCollision()`:Returns the numCollision.

- **`void displayTable()`**: It will print the chains of the tables along with bucket indices. It will not print any priority queue information
    a. Basically iterate over the table
    b. For each entry in the table, traverse the linked list and print the restaurantName of the node.
    c. An example is given here with a table of size 5 and the provided test.txt. For more detail example please refer to the run example at the end of the document.

```
------------------
Enter your choice >>7
0|Lazy Dog-->Five Guys-->In-N-Out Burger-->NULL
1|C4C-->The Sink-->NULL
2|Panda Express-->NULL
3|Wendy's-->NULL
4|McDonald's-->Chipotle-->NULL
------------------
```

Driver Code

You should pass two command line arguments to the *main* function:

1. Initial file from which hash table will be populated. The file uses a semicolon as a delimiter. The format of the file is:

   restaurantName;review;customer;time

   For example:

McDonald's;I absolutely love McDonald's and would give it a 5-star rating!;Billy Bob;1603

2. Size of the hash table

The driver code should be a menu-driven function and display the following menu options to the user. Note that the menu should consistently display until the user chooses to exit.

```
------------------
1: Build the data structure (execute this option one time)
2: Add a review
3: Retrieve most recent review for a restaurant
4: Pop most recent review for a restaurant
5: Print reviews for a restaurant
6: Display number of collisions
7: Display table results
8: Exit
------------------
```

The displayMenu() function is given as part of the starter code. See the following for more details on the menu options:

1: Build the data structure (execute this option one time): Build the hash table from the filename passed as command line arguments. Call the **`setup`** of hash table. Note that during a single execution of your program, this option should only be chosen once. Therefore, any subsequent attempt of choosing option 1 should not call **`setup`** but rather display a message to the customer noting that the data structure has already been built.

```
------------------
1: Build the data structure (execute this option one time)
2: Add a review
3: Retrieve most recent review for a restaurant
4: Pop most recent review for a restaurant
5: Print reviews for a restaurant
6: Display number of collisions
7: Display table results
8: Exit
------------------
Enter your choice >>1
```

```
------------------
1: Build the data structure (execute this option one time)
2: Add a review
3: Retrieve most recent review for a restaurant
4: Pop most recent review for a restaurant
5: Print reviews for a restaurant
6: Display number of collisions
7: Display table results
8: Exit
------------------
Enter your choice >>1
The data structure has already been built.
```

2: Add a review. This will prompt for information about the restaurant. The user will input restaurantName, customer, review, and time accordingly. Then, create an instance of `ReviewInfo` and insert it into the hash table. Refer to the example at the end of this document.

```
------------------
1: Build the data structure (execute this option one time)
2: Add a review
3: Retrieve most recent review for a restaurant
4: Pop most recent review for a restaurant
5: Print reviews for a restaurant
6: Display number of collisions
7: Display table results
8: Exit
------------------
Enter your choice >>2
Restaurant Name: Raising Cane's
Customer: Larry
Review: I love chicken
Time: 1100
```

3. Retrieve the most recent review for a restaurant: First, ask the user to provide the name of the restaurant. Then, search for that restaurant in the hash table. If found, print the most recent review information from the priority queue. Use the search function of the hash table and peek function of the priority queue. If no such restaurant exists in the hash table print `"no record found".`

```
------------------
1: Build the data structure (execute this option one time)
2: Add a review
3: Retrieve most recent review for a restaurant
4: Pop most recent review for a restaurant
5: Print reviews for a restaurant
6: Display number of collisions
7: Display table results
8: Exit
```

```
------------------
Enter your choice >>3
Restaurant name:Raising Cane's
retrieved result
Restaurant Name: Raising Cane's
Customer:Larry
Review:I love chicken
Time:1100
```

4. Pop most recent review for a restaurant: Prompt for the restaurant name. Search for the restaurant in the hash table using the `searchItem` function. If found, pop the top priority review from the priority queue using the function **`pop()`.** Remember to update the currentSize.

```
------------------
1: Build the data structure (execute this option one time)
2: Add a review
3: Retrieve most recent review for a restaurant
4: Pop most recent review for a restaurant
5: Print reviews for a restaurant
6: Display number of collisions
7: Display table results
8: Exit
------------------
Enter your choice >>4
Restaurant name:Raising Cane's
------------------
1: Build the data structure (execute this option one time)
2: Add a review
3: Retrieve most recent review for a restaurant
4: Pop most recent review for a restaurant
5: Print reviews for a restaurant
6: Display number of collisions
7: Display table results
8: Exit
------------------
Enter your choice >>3
Restaurant name:Raising Cane's
retrieved result
no record found
```

5. Print reviews for a restaurant: Ask for the restaurantName and read the restaurantName from the console. Then, search for the restaurant in the hash table. If found, print the review information from the associated priority queue.

```
------------------
1: Build the data structure (execute this option one time)
2: Add a review
3: Retrieve most recent review for a restaurant
4: Pop most recent review for a restaurant
```

```
5: Print reviews for a restaurant
6: Display number of collisions
7: Display table results
8: Exit
------------------
Enter your choice >>5
Restaurant name:Five Guys
Restaurant: Five Guys
        Customer: Casper
        Review: Mmmm five guys.
        Time: 2230
        =====
        Customer: Kavin
        Review: I've eaten here several times and the food has always
been good.
        Time: 1830
        =====
        Customer: Jamie Anderson
        Review: This place is awesome. Great food and great service.
My kids love this place too. Glad to find a quality burger joint close
to our house.
        Time: 845
        =====
        Customer: Joe Adams
        Review: Five Guys, oh man, they know how to serve up some
delicious burgers and fries!
        Time: 1721
        =====
```

If the restaurant is not found in the hash table, then print `"no record found"`.

6. Display number of collisions: Display the number of collisions

```
-----------------
1: Build the data structure (execute this option one time)
2: Add a review
3: Retrieve most recent review for a restaurant
4: Pop most recent review for a restaurant
5: Print reviews for a restaurant
6: Display number of collisions
7: Display table results
8: Exit
------------------
Enter your choice >>6
Number of collisions:5
```

7. Display table results: Calls `displayTable` to display the contents of the hash table.

```
------------------
1: Build the data structure (execute this option one time)
2: Add a review
3: Retrieve most recent review for a restaurant
```

```
4: Pop most recent review for a restaurant
5: Print reviews for a restaurant
6: Display number of collisions
7: Display table results
8: Exit
------------------
Enter your choice >>7
0|Lazy Dog-->Five Guys-->In-N-Out Burger-->NULL
1|C4C-->The Sink-->NULL
2|Panda Express-->NULL
3|Raising Cane's-->Wendy's-->NULL
4|McDonald's-->Chipotle-->NULL
------------------
```

8. Exit: Exit the program and free any memory that was allocated on the heap for both the Hash table and Priority Queue. You must use the respective destructors to accomplish this. Use Valgrind to verify that you freed the memory correctly. You can find the GDB and Valgrind debugger guide on Canvas under Modules.