CSCI 2270 Spring 2025 Project
Due Sunday, April 27 @ 1159PM MDT

ANY USE OF AI TOOLS SUCH AS CHATGPT WILL BE CONSTRUED AS PLAGIARISM AND RESULT IN A 0. YOU MUST COMPLETE THE PROJECT INDEPENDENTLY AND WRITE EVERY LINE OF CODE ON YOUR OWN.

Learning Objectives:

- Hashing with Chaining
- Priority Queues
- Graphs

Grading policy

- You are required to schedule a mandatory Interview Grading session with course staff. A sign-up sheet has been posted via a Canvas announcement. Interview Grading will be conducted between 4/28 and 5/1. Students who don't sign up for their interview on time or who skip their interview will receive 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. Submitting past 1159PM MDT on the due date will result in a 0 regardless of the reason. You are being given 2 weeks to complete this project. Start early.
- Step 2 – Zip everything in your GitHub repository solution (.zip extension) and upload it to the Canvas Course Project dropbox by the due date.

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 the respective data structures.
- 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 notes.
- Repeat interviews will not be given if you do not interview well. You will receive a 0 and it will be reported to the Honor Code Committee. In short, you must understand every line of code.

Please read this document carefully before you begin working. Please read this document carefully before you start working

Introduction

In this project you will create and maintain a data structure for efficient storing, retrieving and manipulating of character data and their inventories for a dungeon crawler. Each item consists of four data points given as:

1. **characterName:** Name of the enemy character whose inventory this item belongs to
2. **itemName:** Name of the item
3. **damage:** the strength of this item
4. **comment:** any descriptive comments about the item

Here is one example of an item:

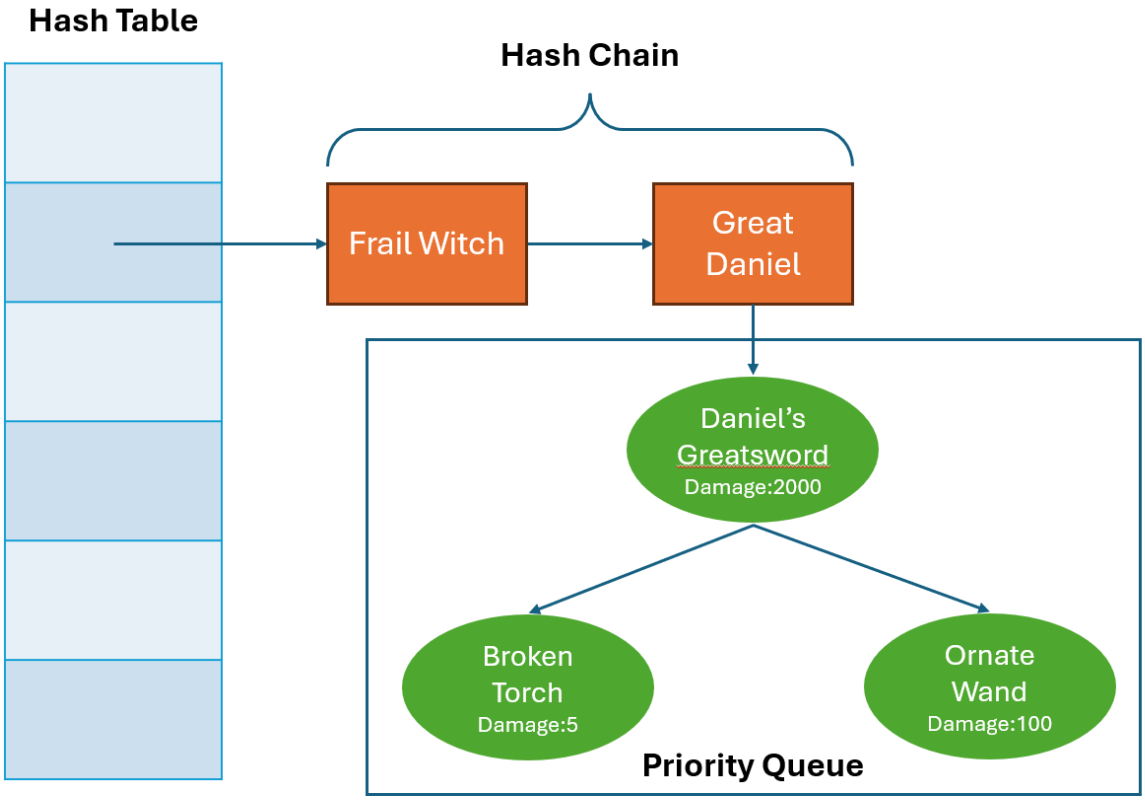
characterName: Great Daniel itemName: Daniel's Greatsword damage: 2000 comment: This is Great Daniel's great Greatsword

Therefore you will be using the following structure (can be found at PriorityQueue.hpp)-

```
struct ItemInfo{
    string characterName;
    string itemName;
    int damage;
    string comment;
};
```

Overview of the data structure

You are supposed to store the items in a hash table with chaining. Each node in the chain will represent the character that the item belongs to. In the example below: “Frail Witch” and “Great Daniel” will have the same hash value, and therefore they will reside in the same chain. Within each character node there will be a priority queue. You will need to organize the items in the priority queue by damage value. The Most powerful weapon will be at the front of the queue.



During hashing the hash function will take the characterName and will give you the bucket index in the hash table. Based on the index, retrieve the head appropriate chain. Each character will have a distinct node in the chain. Therefore, find the node responsible for the current characterName, and then depending on the purpose manipulate the node. Note, as mentioned above and depicted in the image each node will have a priority queue associated with it. Node structure is defined in the hash.hpp as follows-

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

Priority Queue

Each node of hash table chain will store a priority queue to store the information about an item. Note each item is stored as an instance of structure ItemInfo. The queue will be organized on the damage subfield of ItemInfo. The higher values signify higher priority, therefore choose proper heap implementation (think about the question - do you need a maxheap or minheap)? This is an array based implementation of heap. The array will be created dynamically from the constructor. Note each element of the array is of type ItemInfo

- `PriorityQ(int capacity)`:constructor
- `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
- `ItemInfo* peek()`: return a pointer to the top of the heap (or nullptr if empty)
- `void heapify(int index)`: maintain the heap as per the priority
- `void pop()`: remove the top priority element from the queue
- `void insertElement(ItemInfo value)`: Insert an element in the queue. After insertion you need to make sure that the heap property is maintained.
- `void print()`: print the contents of the queue. Go over the array and print
    - `cout<<"\t"<<"Item: "<<heapArr[i].itemName<<endl;`
    - `cout<<"\t"<<"Damage: "<<heapArr[i].damage <<endl;`
    - `cout<<"\t"<<"Comment: "<<heapArr[i].comment <<endl;`
    - `cout<<"\t"<<"================="<<endl;`
- `void deleteKey(string item)`: Delete an item from the queue specified by item name.

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 the hash.hpp for class definition and function declaration. Note each node will contain a priority queue to store the information such as itemName, damage, and comment. 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 only it is a collision. So, for example you entered a record for say "Frail Witch" and then you entered a second record for "Vile Beast", here there is no collision. Now you entered a record for "Great Daniel", and hash("Frail Witch") == hash("Great Daniel") then you have a collision. So the variable `numCollision` should be adjusted accordingly. Beyond this any other insertion of items under "Frail Witch" or "Great Daniel" 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.
- `node* createNode(string characterName, node* next)`: This function will create a linked list node for the chain with characterName. During creation of the node, create the priority queue of size 50. You will call it from the insertCharacter function.
- `unsigned int hashFunction(string characterName)`: 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* searchCharacter(string characterName)`: Search for a characterName in the hash table. If found returns the node, else returns null.
    a. Calculate the hash function for the argument characterName.
    b. Retrieve the chain head from the table
    c. Traverse the chain to find the node containing the characterName.
    d. On successful search return the node, else return null.

- `void insertItem(ItemInfo item):` Insert a new item to the hash table.
    a. First search for the node with item.characterName
    b. If a node exists in the hash table for that character, insert the ItemInfo item into the priorityQueue of that node.
    c. If the search method returns a null then
        - Create a new node for that characterName.
        - Add the ItemInfo item into 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 the linkedlist accordingly.
- `void buildBulk(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
  characterName;itemName;damage;comment
  For example

Great Daniel;Daniel's Greatsword;2000;Great Daniel's great greatsword

  Inside the buildBulk function you will read each line from the file. For each line of the file, create an itemInfo structure instance out of the line you just read from the file. Call the insertItem function with the instance of itemInfo.
- `int getNumCollision():` Returns the numCollision.
- `void printTable():` 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 characterName of the node.
    c. An example output can be seen at the end of the document.

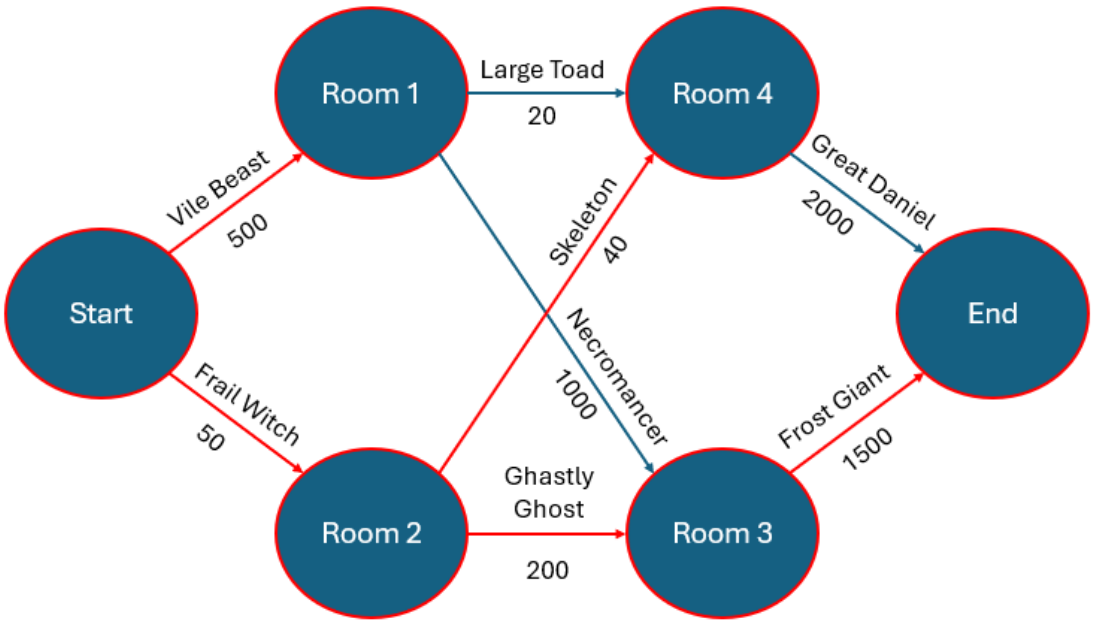
- `void deleteEntry(string characterName, string itemName)`: Delete an item record with characterName and user as specified in the argument.

a. First search in hashTable for the characterName.
b. If the search returns null, cout `"no record found".`
c. If search returns a valid node, delete the record with characterName and itemName from the priority queue associated with the concerned node. Use the `deleteKey` function of the priority queue.
d. If this deletion makes the priority queue of the concerned node empty delete the node. Handle the boundary cases like deletion of head accordingly.

Graph Search

The final component of the project is to apply your work on the item storage and retrieval to implement a Dijkstra's shortest path search through a dungeon. All graph code other than your search implementation is provided. In your search, edges will be labeled with a characterName rather than with a weight directly. To determine the edge weight, you must search your hash table and subsequent priority queue for the item with maximum damage. The damage value return from your search will act as edge weight. Using these values, your shortest path should result in the easiest route through the dungeon.

Example dungeon below, all solved nodes are highlighted with the shortest path Dijkstra's found:



Driver code

You should pass two command line arguments-

1. Initial file from which hashtable will be populated. The file is ; separated file and the format of a line is characterName;itemName;damage;comment

   For example

Great Daniel;Daniel's Greatsword;2000;Great Daniel's great greatsword

2. Size of the hash table

Driver code should be a menu driven function. It should present the following menu options to the user and it must not stop until the user chooses to exit.

1: Build the datastructure (call it only once)

2: Add a new item

3: Peek most powerful item from character

4: Pop most powerful item from character

5: Print all items for character

6: Get number of collisions

7: Print the table

8: Find easiest route through dungeon

9: Exit

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

1: Build the datastructure (call it only once): Build the hash table from the filename passed as command line arguments. Call the `buildBulk` of hash table. Note during the one execution this option can be chosen once only. Therefore any subsequent attempt of choosing option 1 should not be allowed.

2: Add an item. This will prompt for information about the item. User will input characterName, itemName, damage, and comment accordingly. Then you will create an instance of ItemInfo and will insert it to the hash table. Refer to the example at the end of this document.

3. **Retrieve most powerful item for a character:** First ask the user to provide the name of the character. Then look for that character in the hash table. If found, print the most powerful item information from the priority queue. Use the search function of the hash table and peek function of the priority queue. If no such character exists in the hash table print `"no record found".` Refer to the example at the end of this document.

4. **Pop most powerful item for a character:** Prompt for the character name and the item name. Search for the character in the hash table. If found, delete the itemInfo with the item name from the priority queue of the hash table node. Use the function `deleteEntry` of hash table.

5. **Print all items for a character:** Prompt for the character name, then search for the character in the hash table. If found, print the item information from the associated priority queue. If the character is not found in the hash table, then print `"no record found".` Refer to the example at the end of this document.

6. **Get number of collision:** Print the number of collisions. Refer to the example at the end of this document.

7. **Print the table:** Call the `printTable` of the hash table. Refer to the example at the end of this document.

8. **Find easiest route through dungeon:** Runs a Dijkstra's graph search on dungeon specified by file name. There are dungeon text files (e.g. dungeon1.txt, dungeon2.txt, etc) provided. Before attempting this you must ensure that your other functions are all working successfully. Implement Dijkstra's algorithm in the provided function EnemyGraph::findEasiestPath. The AdjecentVertex edges contain an enemy_id that you must search for in your hash table, use the max damage item as the edge weight for your shortest path. Return the shortest path and print it. Refer to the example at the end of this document.

9. **Exit:** Exit the execution and free all memory for all of the data structures. In short, your program should not have any memory leaks. Use Valgrind to detect memory leaks, which you will find instructions for by using the document titeld "GDB and Valgrind Guide for C++", which was posted to Canvas at the beginning of the semester (see the Resources section under Modules). This will account for 5% of your project grade. Focus on eliminating the "definitely lost" leaks.

Sample test case execution:

```
$ ./a.out test.txt 10
------------------
1: Build the datastructure (call it only once)
2: Add a new item
3: Peek most powerful item from character
4: Pop most powerful item from character
5: Print all items for character
6: Get number of collisions
7: Print the table
8: Find easiest route through dungeon
9: Exit
------------------
Give your choice >> 1
------------------
1: Build the datastructure (call it only once)
2: Add a new item
3: Peek most powerful item from character
4: Pop most powerful item from character
5: Print all items for character
6: Get number of collisions
7: Print the table
8: Find easiest route through dungeon
9: Exit
------------------
Give your choice >> 7
table[0]: Stoic Elf, Large Toad, Great Daniel, Frank the Frost Giant
table[1]: Gelatinous Blob, Ghastly Ghost
table[2]: Frost Giant, Skeleton, Vile Beast, Frail Witch, Maria the Marauder
table[3]: Necromancer, Scottish Ogre
table[4]: Wiggly Wizard

------------------
```

```
1: Build the datastructure (call it only once)
2: Add a new item
3: Peek most powerful item from character
4: Pop most powerful item from character
5: Print all items for character
6: Get number of collisions
7: Print the table
8: Find easiest route through dungeon
9: Exit
------------------
Give your choice >> 6
Number of collision: 9
------------------
1: Build the datastructure (call it only once)
2: Add a new item
3: Peek most powerful item from character
4: Pop most powerful item from character
5: Print all items for character
6: Get number of collisions
7: Print the table
8: Find easiest route through dungeon
9: Exit
------------------
Give your choice >> 8
Input dungeon(file name):
dungeon2.txt
1750
------------------
1: Build the datastructure (call it only once)
2: Add a new item
3: Peek most powerful item from character
4: Pop most powerful item from character
5: Print all items for character
```

```
6: Get number of collisions
7: Print the table
8: Find easiest route through dungeon
9: Exit
------------------
Give your choice >> 3
Character name: Wiggly Wizard
retrieved result
Character: Wiggly Wizard
Item: Staff of Hair Loss
Damage: 200
Comment: The horror!
------------------
1: Build the datastructure (call it only once)
2: Add a new item
3: Peek most powerful item from character
4: Pop most powerful item from character
5: Print all items for character
6: Get number of collisions
7: Print the table
8: Find easiest route through dungeon
9: Exit
------------------
Give your choice >> 2
CharacterName: Wiggly Wizard
ItemName: Wnad of Big Boom
Damage: 250
Comment: Makes boom go big
------------------
1: Build the datastructure (call it only once)
2: Add a new item
3: Peek most powerful item from character
4: Pop most powerful item from character
```

```
5: Print all items for character
6: Get number of collisions
7: Print the table
8: Find easiest route through dungeon
9: Exit
------------------
Give your choice >> 3
Character name: Wiggly Wizard
retrieved result
Character: Wiggly Wizard
Item: Wnad of Big Boom
Damage: 250
Comment: Makes boom go big
------------------
1: Build the datastructure (call it only once)
2: Add a new item
3: Peek most powerful item from character
4: Pop most powerful item from character
5: Print all items for character
6: Get number of collisions
7: Print the table
8: Find easiest route through dungeon
9: Exit
------------------
Give your choice >> 5
Character name:Wiggly Wizard
Character: Wiggly Wizard
        Item: Wnad of Big Boom
        Damage: 250
        Comment: Makes boom go big
        ================================
        Item: Staff of Hair Loss
        Damage: 200
```

```
        Comment: The horror!
        ================================
        Item: Staff of Vague Illness
        Damage: 100
        Comment: Good enough to get out of class
        ================================
        Item: Staff of Immobility
        Damage: 40
        Comment: Everybody freeze!
        ================================
------------------
1: Build the datastructure (call it only once)
2: Add a new item
3: Peek most powerful item from character
4: Pop most powerful item from character
5: Print all items for character
6: Get number of collisions
7: Print the table
8: Find easiest route through dungeon
9: Exit
------------------
Give your choice >> 4
Character name:Wiggly Wizard
------------------
1: Build the datastructure (call it only once)
2: Add a new item
3: Peek most powerful item from character
4: Pop most powerful item from character
5: Print all items for character
6: Get number of collisions
7: Print the table
8: Find easiest route through dungeon
9: Exit
```

```
------------------
Give your choice >> 4
Character name:Wiggly Wizard
------------------
1: Build the datastructure (call it only once)
2: Add a new item
3: Peek most powerful item from character
4: Pop most powerful item from character
5: Print all items for character
6: Get number of collisions
7: Print the table
8: Find easiest route through dungeon
9: Exit
------------------
Give your choice >> 4
Character name:Wiggly Wizard
------------------
1: Build the datastructure (call it only once)
2: Add a new item
3: Peek most powerful item from character
4: Pop most powerful item from character
5: Print all items for character
6: Get number of collisions
7: Print the table
8: Find easiest route through dungeon
9: Exit
------------------
Give your choice >> 4
Character name:Wiggly Wizard
PQ emptied...
------------------
1: Build the datastructure (call it only once)
2: Add a new item
```

```
3: Peek most powerful item from character
4: Pop most powerful item from character
5: Print all items for character
6: Get number of collisions
7: Print the table
8: Find easiest route through dungeon
9: Exit
------------------
Give your choice >> 7
table[0]: Stoic Elf, Large Toad, Great Daniel, Frank the Frost Giant
table[1]: Gelatinous Blob, Ghastly Ghost
table[2]: Frost Giant, Skeleton, Vile Beast, Frail Witch, Maria the Marauder
table[3]: Necromancer, Scottish Ogre
table[4]:
------------------
1: Build the datastructure (call it only once)
2: Add a new item
3: Peek most powerful item from character
4: Pop most powerful item from character
5: Print all items for character
6: Get number of collisions
7: Print the table
8: Find easiest route through dungeon
9: Exit
------------------
Give your choice >> 9
Goodbye...
```