Whatcom Community College

Computer Science 240 – Data Structure/Algorithm

Adib Thaqif, Don Strong, Andrew Jacobi, Micah Miller

**Final Check-in**

**Choices Game Project**

**Overview**

We have made several good progresses to the project. We manage to create a flexible event object that can be easily used in several data structures that we have implemented. We managed to create a working hash table and heap that works well with the event object. We managed to also create an aspect value object that will be used in the main. We have started doing the main file that would tie everything together (data structure, objects, text files).

For the event object, we managed to create an event object that would take in a string text, and two ints that would represent the reward and punishment values for the event. We successfully implemented an add, delete and isEmpty function that would be constantly used in the data structures associated with the event.

Next, we successfully implemented several text files (life, school, work) that would suit the way that we would present the events to the user. We also make it easier for us to work by deleting unnecessary formatting of the text files.

**To do lists**

Implementing a main(7th March)

Testing the main(8th March)

Trying to implement an additional feature (boost, add event) (9-10th March)

Testing the project (Final Week)

Minor tweaks(Final Week)

**Course materials implemented**

We also have two data structures that will be used for the project: An Event Hash Table and an Event Heap. Both of these data structures serve different purposes to the project. The Hash Table is the base data structure that would directly connect with the created text files, while the heap will be the communication between the hash table and the main file.

The heap is used to arrange the data so that it would prioritize the lowest impact value of the event (reward + abs(punishment)) near the top of the heap. This would create increase the difficulty of the game steadily for the player. We implemented a basic heap first, before specializing to accommodate for our needs, by eliminating unnecessary functions, and focus on making sure percolate up and down works based off the impact values of the events.

For our first data structure for this project, we elected to implement our own hash table. This proved to be a daunting task, thankfully with a lot of help from outside resources and our instructor, we managed to get a very simplistic – yet functional – version working. Many of the design decisions are on how we would design a good hash table that works for our project.

**Design Decisions**

For our first data structure for this project, we elected to implement our own hash table. This proved to be a daunting task, thankfully with a lot of help from outside resources and our instructor, we managed to get a very simplistic – yet functional – version working. Our implementation used a single dimensional array that takes in a size parameter in the constructor. For the use of our program, our “Manager” class will default to constructing it with a prime number to limit the number of collisions to deal with. The manager class read’s in each text file and creates the event object, then assigns it to a portion of the hash table that’s reserved for the specific aspect that the event object pertains to. We elected to go this route because it will give us a single storage option for all of our event objects that will be used throughout all four rounds (simulating years in college). The reason that we essentially subdivided our array is for simplicity when discerning which aspect pool to draw from.

1. The manager uses the aspect in the first line (which is one of three options: “JOB” “WORK” or “LIFE”; all of type String) to determine the subsection of the array it will be assigned to. It then takes the string length of the aspect (3, 4, or 5 respectively) mods it by the table size and assigns it to the array at that index. This practice welcomes collisions for which our linear probing technique assigns it to the next sequentially available index. Although it isn’t in good practice to use linear probing, this works for our style of game because we want them to be grouped together for easier retrieval later on.

2. Our get() method differs quite significantly from that of a typical hash table, however. Because each aspect is essentially stored under a single key, it makes for retrieving a specific event object a little tricky. Initially, this was a concern for us, although we managed to use this to our advantage. We wanted the game to avoid selecting repeating events whenever a new round and/or game was played, so we planned on using a random object to generate a random number which in turn would pull a random event object for us to present to the user. Knowing this eventual design concept, we decided that it would be best if our get method always returned a random event of a specific index. Although this seems to be the opposite of “good” practices for a hash table, we couldn’t find a reason why we would need to return a specific event object as opposed to always returning a random one. So our functionality for the get() method takes in a key of int type (either 3 for job, 4 for life, or 5 for class) and returns an event object at a random index within the range of each subarray. As stated earlier, this contrasts heavily with standard practices of a hash table, but it turns out that it works better for us in the long run.

**Testing the Project**

We have yet to test the project completely, but given rigorous experience from computer science classes before, we are able to test almost every single aspect of the project (text files, data structures, objects) before we move on to the next steps. Our goal to test the project would be next week of school, and from then on, we can take a look at how we would be able to expand the game creating a feature for the user to input their own event object. We would not be able to integrate the game to the GUI fully but having a text-based game would still be a successful project, nonetheless.