*ASSUMPTIONS PAGE*

**ELEVATOR LOGIC**

For the purposes of this simulation, a basic structure of how the elevator works is as follows:

Users will appear on every floor.

One user will press the elevator call button before all the others. This will cause the elevator to come down to that person.

\*Note that when there are multiple elevators called, the second person to press the call button will have the elevator head to them.

Elevator will head to the floor of the first person who pressed the call button

Other users will be lined up in order on the elevator’s system. All of the users that are getting off on the same floor as the initial caller will be given priority. All other users are considered next destination

System will check other floors along the way to the second users destination since they will also have lines formed waiting for the elevator to pick them up

The elevator will keep functioning like this until the program terminates

**PROGRAM LOGIC**

Many businesses make use of elevators; however, their usages vary on their times. Many businesses have what are essentially ‘peak hours’ depending on the venue. Schools may have their peak hours right when the day begins while malls may have them in the afternoons or evening. For this project, we are assuming that we are in a decently sized aquarium. The aquarium has been open for a few hours so there are already customers wandering the many floors. The aquarium is in charge of managing maximum building capacity issues, so this program will not take those values into account. Due to the nature of elevator usage, this program will display about a 1 minute, real-time window for three separate times: high volume ( or the peak business hours), medium volume (or an average setting), and low volume (or close to the end/beginning of the day). To create a program that will display how an elevator functions, we had to make a few assumptions on the nature of the average user, the flow volume, and how the elevator will determine where to go next.

**USER ASSUMPTIONS**

The program makes a few assumptions about the average user. We assume that the typical user will simply enter the elevator with no hiccups. This would mean this program will not consider the user needing to press the emergency stop or elevator breakage. Due to the relaxed environment, we also assume that the average user is in no rush to get to their destinations. We will assume that these users will have no need to push the door close button and will simply opt to let the elevator doors close/open in their own times. People can, however, decide that the elevator is simply not worth waiting for and will take the stairs or the more scenic route of the central ramp leading to the largest tank. Prospective users will decide to do so when the line to the elevator hits 10 people, after which they will simply take the stairs. To keep the program run time short while still displaying the general idea, it has been decided that the program will run through a few iterations before stopping person generation and sending off the last few people who were in line

Because the aquarium has been open for a few hours, we already know that there is at least one user on each available floor. We will use a randomizer to determine which floor they decide to get on and just assume that users from each floor will decide to use the elevator regardless. A randomizer will be used to determine which floor has called the elevator first. It will then run as usual.

**ELEVATOR ASSUMPTIONS**

We have also made a few assumptions to the function of the elevators to simplify the process and to make the program easier to follow. For this program’s purposes, we have decided to assume that in single/multi-elevator situations, the elevator will begin on the first floor. The time that it takes for the elevator doors to open/close is negligible, so that will not be calculated in to wait times. We also assume that it will only take a few seconds for an elevator to travel between floors before opening its doors once more. We are also assuming that these elevators can hold a maximum of 15 carriers. If that amount is hit, people will not enter the elevator until it has been cleared out below that threshold.

**FLOW OF TIME**

Just to shorten the program, time is simulated. In reality, user times are more variable and may show different volumes and elevator capacities/weights, however it isn’t practical to run a program in that format. As such we will assume:

* + The elevator floors will stay open for a constant amount of time.
  + To show the ‘waiting time’ of the users, we will use the clock function. Because the elevators in a multi-system may not run concurrently, the times will not be extremely accurate. They will however, still be able to indicate how long they have been on the elevator
  + This program will use the an arrival/departure variable to keep track of time when the person is ‘created’ and when they have left

*ALGORITHM EFFICIENCY*

Algorithms can run at various efficiencies, though the practicality of using them can be a slight barrier. Below are the major functions and their Big-O notations:

Floor Class

* Generate ID 🡪 O(1) complexity
* Queue\_Switcher 🡪 O(1) complexity
* Add From Queue and Add New Person 🡪 O(n) complexity
  + This can be simplified by using a doubly lined list insertion instead. They have O(1) complexity for insertion, though accessing it has O(n) complexity.
* Exit Elevator 🡪 O(n) complexity
  + This can be simplified by using a doubly linked list for deletion. They have O(1) complexity for deletion
* Enter Elevator🡪 O(n) complexity
  + This program is O(n) complexity due to the other functions I have to use in conjunction with it.
  + This could possibly be simplified using a doubly linked list for insertion. They have O(1) complexity
* Queue Generator🡪 O(1) complexity
* Floor Start Randomizer🡪 O(1) complexity
* Multi Elevator system🡪 O(1) complexity

Elevator Class

* Update Elevator Location 🡪 O(n) complexity

Person Class

* Destination Generator 🡪 O(n) complexity
  + This could be fixed to O(1) creating a static table of numbers, making an index (by using sizeof operator and table/ sizeof operator and table pointer). The index of the table becomes the new number.
* Depart Display 🡪 O(1) complexity

MAIN

* Single Elevator Sys🡪 O(n^2) complexity
  + Because this function calls on all the other major functions, the complexity increases a lot. This can be improved by lowering the complexity of the other functions using some ideas that I’ve listed
* Multi Elevator Sys 🡪 O(n^2) complexity
  + This has a large complexity for the same reasons as the previous ones.

It is most certainly possible to increase the efficiency of the entire program. One such way would be to use doubly-linked list instead of the combination of vectors, queues, and deques that I have used in this project. Using the randomizer and the queues that were already formed in this program, we could sort the people by using add/remove functions for the doubly linked list. When the people have left, we could delete the nodes. Insertion and deletion of linked list would only have a O(1) complexity

**REFERENCES**

A computer science portal for geeks. (n.d.). Retrieved July, 2020, from <https://www.geeksforgeeks.org/>

Know thy complexities!(n.d.) Retrieved July, 2020 from

<https://www.bigocheatsheet.com/>

Stack Exchange(n.d) Retrieved July, 2020 from

<https://stackexchange.com/>

UML Class Diagrams Explained with Examples, June 13, 2019. Retrieved July, 2020 from

<https://creately.com/blog/diagrams/class-diagram-relationships/>

**GITHUB URL**

<https://github.com/n-edmond/Elevator_Queue>