Brian Loughran
Elevator Stack Simulation Analysis Document
Johns Hopkins Unviersity
Data Structures

General Design

The major data structure used in this program is a stack which is used to model people loading into a narrow elevator. This is an appropriate model since in stacks only the top element can leave the stack, and in this narrow elevator only the person closest to the door can leave the elevator. Appropriate methods like pop(), push() and isEmpty() are set up to do basic stack operations. I also included some helper methods, like isFull(), which checks if the elevator is full, getNames(), which prints out the name of everybody on the elevator, and getNextFloor(), which returns the next floor that the passengers would like to leave at.

Another important class in the program is the Person() class. Person stores data about each person that rides the elevator, including important information such as their name, the floor that they got on the elevator, the floor that they departed, and the number of times they had to exit temporarily due to the narrow nature of the elevator. Some error checking is done in the Person() class, to account for the fact that the input file may not always be properly formatted. There are also methods for writing the person's information to a string.

Information is read from the input file in the main() method. This input is read in line-by-line, and any input that is improperly formatted is ignored by the program. The output of the program is written to an output.txt file. The first thing that is written to the output file is any input that is not properly formatted, to inform the user that this information will not be included in the simulation. Naturally, the next thing that is printed to the output file is the information that was actually recognize and will be included in the simulation. Once all the information is read into the program, the simulation is run, printing important information like what floor we are on, if the elevator is full or empty, among other things. One important enhancement is that the output also includes printing the names of the people on the elevator when arriving and departing. This makes the output more transparent, and you can more easily track what is going on in the program, allowing you to track each rider of the elevator as they move from floor to floor.

Alternative Approaches

I chose an array implementation, rather than a linked list implementation. I did this mainly because of the limited size of the elevator. Since the elevator only has a capacity of 5 people, it is impossible to load more people. I thought it would be easier to watch for this case using an array implementation than a linked list implementation. Using the linked list implementation you would have to keep track of the number of people currently on the elevator, and cap at 5.

While this is very possible to do, that data comes free with the array implementation, so it seemed to make sense to use the array implementation.

Learning and Looking Back

From this project I learned a lot about the stack data structure and setting up the details of its implementation. This is an interesting stack problem because it flexes all of the operations that stack can reasonably accomplish. People are constantly being push()ed and pop()ped from the stack as the elevator moves from floor to floor. At times the stack will be empty, which means you cannot do a pop() operation. At times the elevator will be full, so you cannot do a push() operation (and the unfortunate person has to take the stairs). Some optional methods were included in the stack like isEmpty() and isFull(). This was a good, real-world case where a stack is a good data structure to use.

One thing I may do differently if I were to restart this project would be to change how I handled people going into the hallway. As of now I just create a 5 person empty list and assign people to the hallway if someone wants to get off and increment through the list to push() the people back onto the elevator. This works, but this could also be implemented as its own stack. Another class could have been created to handle people being in the hallway, with its own push() and pop() methods. This class could be simpler than the elevatorStack class, since it does not need to have so many helper methods. This would also give more practice implementing the stack data structure.

Summary of Findings

At ABC corporation, there are a number of issues with the current elevator system. The most glaring is that there are a number of employees that have to take the stairs. Not only is this a hassle, but can also decrease accessibility to the building for handicapped people. Another issue is born due to the narrow nature of the elevator, causing people to have to temporarily unload into the hallway, which greatly reduces the efficiency of the elevator. The time that people are loading, unloading, and otherwise shuffling about could be used to make more trips, and to carry more people. My recommendation is to figure out how to fix the second elevator. This will likely reduce, and potentially eliminate the people who have to take the stairs, and will reduce the load on the first elevator. While it would be ideal to have more room in the narrow elevator, it would likely be very costly to widen the elevator shaft. This may be a consideration at ABC if budget is no option, however that is rarely the case, and it is likely more efficient to fix the second elevator.