CS2040S: Data Structures and Algorithms

Recitation 1

Problem 1. (Do You Even Lift?)

Image credit: https://imgflip.com/

Goal: Think about what goes into specifying an algorithm precisely, and how to come up with appropriate metrics. In real-world problems, the correctness of your proposed algorithm depends a lot on how you formulate the problem and how your solution is justified in solving it. Feel free to be creative!

To help you develop some intuitions for this problem, you might be interested in playing with this lift simulation game: https://play.elevatorsaga.com/. The only caveat here is that it is designed for Javascript programming, which is not Java — the language of instruction for this module.

Before we begin, let's first introduce 2 important concepts. *Abstract Data Type* (ADT) is a data type specification which details the permitted values, supported operations and behaviors of the supported operations. In other words, it captures the set of behaviours from the perspective of you the programmer interacting with the data type. Sounds familiar? The Java Interface is one way to specify ADTs!

ADTs are technically different from *Data Structures* (DS), which are concrete representations of data. Typically DS details how an ADT is realized in implementation. In other words, DS is a specification from the point of view of an implementer, not a user. As an example, the stack ADT can be implemented using Array or Linked List DSes. Think of a DS as the actual Java class which implements some ADT interface(s).

One of the ADTs relevant to this problem is the *queue* which imposes a First-in-First-Out (FIFO)

policy on its data membership (just like a queue in real life). The queue ADT is specified by the following semantics:

Operation	Behaviour
`enqueue(v)`	Insert an element `v` at the rear of queue
`dequeue()`	Remove and return the frontmost item in queue. If queue is empty, return `NULL`
`peek()`	Return the frontmost item in queue without removing it. If queue is empty, return `NULL`

Note that for queues, `enqueue` is sometimes called `push` or `offer`, `dequeue` is sometimes called `pop` or `poll`.

Another similar ADT is the *deque* (often pronounced "deck" so as not to be confused with `dequeue`) which is a double-ended queue. The deque ADT is specified by the following semantics:

Operation	Behaviour
`push_front(v)`	Insert an element `v` at front of deque
`push_back(v)`	Insert an element `v` at the rear of deque
`pop_front()`	Remove and return the frontmost item in deque. If deque is empty, return `NULL`
`pop_back()`	Remove and return rearmost item in deque. If deque is empty, return `NULL`
`peek_front()`	Return the frontmost item of deque without removing it. If deque is empty, return `NULL`
`peek_back()`	Return the rearmost item of deque without removing it. If deque is empty, return `NULL`

Note that for deques, `push_front` is sometimes called `push`, `pop_front` is sometimes called `pop`, `push_back` is sometimes called `inject` and `pop_back` is sometimes called `eject`.

Problem 1.a. Consider how lifts operate. What is the algorithm deciding which floor a lift should go next? Now imagine you are writing the (embedded) controller for the lift and have to implement such an algorithm. Write the pseudocode for the algorithm, being sufficiently precise in specifying exactly what the lift should do at every instance.

ProTip: Don't try to solve every aspect of the problem from the get-go! Start by solving the most important aspect of the problem in the simplest scenario and come up with the most naive working solution you can think of. Thereafter, gradually build upon that basic working solution to incorporate more complexities. Be patient! Good solutions take many iterations to get right and you may even have to redefine the problem and start over when you discover deeper problem insights along the way!

Guiding questions:

- What is the problem you are trying to solve?
- What are the inputs (knowns) and outputs (unknowns)?
- What are some problem scenarios?

- What internal state does the lift need to maintain?
- Which DS/ADTs we have learnt so far (i.e. arrays, queues, stacks) should we use to store and manage the state?
- What are the invariants: the relationships between variables?
- What are the corner cases for your proposed algorithm and how should you handle them?
- What are the strengths and weakness of your proposed algorithm? Which are the ideal scenarios where it performs optimally and which are the adversarial scenarios where it will perform the worst?
- What is the worst time complexity of your algorithm?

Did you know? In magnetic hard drives, the elevator algorithm (a.k.a. SCAN) is actually used to determine which disk sector to move the read/write arm to. Read more at https://en.wikipedia.org/wiki/Elevator_algorithm.

Problem 1.b. If we want to evaluate how good our elevator algorithm is, what metrics should we use? Realize that when evaluating an algorithm, it isn't always obvious what metrics we care about, and you have to think hard to make sure you are optimizing the right thing!

ProTip: The choice of metrics should provide an good measure of how well the chosen objective is met. Metrics therefore drives the solution. You may be looking at multiple metrics at once, which is in the case where you are optimizing for multiple objectives. Often times, a single total metric can be obtained by linearly combining multiple metrics via weighing the relative importance of the objectives they correspond to.

Problem 1.c. How could you design elevators to work better? What would a better algorithm look like? What additional data or inputs would you need? How would you test your solution? If you were to simulate it, how would you program the simulator? The latter is more an exercise in OOP, so not as critical in the context of this module.

Problem 1.d. In your groups, discuss how you might implement a queue ADT in Java using an **array**. For now, assume that the queue will have at most M items. Is there anything else you want as part of your queue interface? What are the corner cases and how exactly do you want it to behave in those corner cases? What about implementing a deque with an array?