Implementer’s Design Decision Hierarchy

Abstract Data Type
Which ADT is the best fit?

Data Structure
Which data structure offers the best performance for our input/workload?

Implementation Details
How do we maintain invariants?

List

Resizable Array
Linked Nodes

data is an array of items, never null. The i-th item in the list is always stored in data[i].

?: How do we determine whether one data structure is faster than another? Does it depend on the implementation details?

?: How do invariants relate to data structures?
Big-O Runtime Analysis

What does it mean for a data structure to be slow or fast?

**Big-O runtime analysis**: count how many steps a program takes to execute an input of size N.

Suppose our list has N items.

A method that takes a constant number of steps (e.g., 23) is in $O(1)$.

A method that takes a linear number of steps (e.g., $4N + 3$) is in $O(N)$.

- **Q**: How does constant or linear relate to analyzing runtime "with respect to big inputs"?

- **Q**: What are the big-O runtimes for ArrayList and LinkedList removeFront?

- **Q**: Can we say that an ADT is slower or faster than another ADT?
Recall that the Stack ADT specifies two important methods:

- `push(Item item)`: Puts the item on the top of the stack.
- `Item pop()`: Removes and returns the top item of the stack.

Assume for the resizable array that we use the `addLast` and `removeLast` methods from `ArrayList`. Assume for linked nodes that we use the `addFirst` and `removeFirst` methods from `LinkedList`, and we have a reference to the front of the `LinkedList`.

Which Stack implementation is faster overall?

- Resizable array
- Linked nodes
- Both are about the same
- Not sure

?: How do the Stack ADT methods compare to List ADT methods?

?: How do the implementations for `ArrayList` methods differ from `ArrayStack` methods?
LinkedStack

State

Node top

int size

Behavior

push – create a new node linked to top; update top to new node; increment size
pop – return top item; update top; decrement size

Runtime

push – O(1) always
pop – O(1)

?: If the push and pop operations of LinkedStack is always at least as good or better than ArrayStack, would we ever want to use ArrayStack?

Hiding Program Complexity

Contract: Assuming they agree to the ADT's possible values and operations, the client and the implementer can improve their programs at the same time.

Invariants: A checklist of assumptions the implementer needs to maintain every time they add a behavior to a data structure.

If the List ADT does everything the Stack and Queue ADTs can do, why use Stack or Queue instead of List?

?: How do invariants affect the implementation of ArrayList and ArrayStack?

Q1: If the List ADT does everything the Stack and Queue ADTs can do, why use Stack or Queue instead of List?
ArrayQueue: Design 1

Same design as ArrayStack: borrow ArrayList’s addLast and removeFront.
It’s basically just an ArrayList.

---

Reconsidering Data Structure Invariants

ArrayQueue (Design 1) is basically just an ArrayList.
Recall the representation invariant for the underlying data array in an ArrayList.
- data is an array of items, never null.
- The i-th item in the list is always stored in data[i].

1. How does maintaining this invariant affect the runtimes for add and remove?

2. Propose an invariant that could result in faster runtimes for add and remove.

Q1: How does this invariant relate to the runtimes for add and remove?

Q2: Propose an invariant that could result in faster runtimes for add and remove.
The i-th item does not need to be data[i] so the front of the queue does not need to be the front of the array!

Q1: Give an invariant that describes this behavior in your own words.

front represents the index of the front of the queue (except when the queue is empty) while back represents the index for the next item.

front represents the index of the front of the queue (except when the queue is empty) while back represents the index for the next item.

?: What’s the runtime for ArrayQueue (Design 2) add and remove?

?: Is it necessary to maintain an integer index for remembering the back of the array?

?: We found a faster way to implement ArrayQueue. Is it possible to take these invariants and use them to implement a faster ArrayList?
Give an invariant that describes ArrayQueue (Design 2) in your own words.

**Q**

**Q1:** Which method has a worse runtime: add or remove?

**Q2:** How would you improve the runtime?

?: How does this change your visualization of the data structure?
LinkedQueue: Design 2

Add a front pointer.

?: What are other possible designs for LinkedQueue? What set of invariants can result in a slower LinkedQueue implementation?

Implementer’s Design Decision Hierarchy

Abstract Data Type
Which ADT is the best fit?

Data Structure
Which data structure offers the best performance for our input/workload?

Implementation Details
How do we maintain invariants?

Today, we studied the ADT implementer’s view of the Design Decision Hierarchy. A recurring theme in computer science is that problem representations (implementation details) reflect problem solutions (data structures).

One neat observation: by simplifying the ADT interface, we gave the implementer more control over how they implemented their data structures. The more complex the ADT, the more restrictive the invariants, which means the implementer might not be able to make as many runtime optimizations.

?: We’ll later look at the ADT client’s perspective. How does the client determine which ADT is the best fit? To what extent does the client need to worry about ADT and data structure complexity?