CSE250 Week 6: Linear Data Structures (Stack, Queue, Deque, Buffer, Linked List, chs 6,7,12)

First, some notes about the text and flow of concepts.

- The text introduces **amortized complexity** early in chapter 7. This aspect of the "doubling" implementation of arrays was enshrined in the C++ `vector` class in the years before the 1998 ANSI standard and adopted by many other languages, hence its early prominence. But I prefer to cover it after the basics of linked lists.
- The text downplays the role of the **principal constant**. Never mind that Lipton and I are credited for the "Galactic Algorithms" critique of ignoring the constant. Awareness of it is good motivation for the hit you take by the "doubling" implementation, why linked lists may be preferred to arrays in certain situations, why sentinels are used in chapter 12, and more.
- The text seems to lack a simple statement of the main downside of linked lists: that they often require random access in ways that become stream-unfriendly on large scale. You can make that case based on the skipped threading and networking chapters 8--11. Streamability is actually a main motivation for the native `List` data type in Scala.
- The `mutable.DoubleLinkedList` type has become deprecated, evidently for reasons that reflect the last two points.
- A **buffer** (ch. 6) is best understood (IMHO) as an ADT that allows "splicing" of sequence data. This will be motivated after we understand the efficiency hits this causes, compared to the simpler linear data structures.
- Your earlier classes have introduced stacks and queues (not to mention linked lists and even trees for many of you). Hence I will take a comparative ("dialectic") approach on a bigger-picture level that connects chapters 7 and 12.

We will frame this around the question:

Can Stack and Queue share the same implementation? How about Double-Ended Queue?

Well, we need to specify what ingredients define Stack and Queue as ADTs. Here `A` refers to the client type, as in `Stack[A]` and `Queue[A]` in Scala.

Operations:

1. Construct an empty Stack or Queue. Target: $O(1)$ time.
2. Constructing a Stack or Queue from n data elements will begin with an empty one and add the data points one at a time. Target: $O(n)$ time.
3. Test `isEmpty`. Target: $O(1)$ time.
4. For a nonempty Stack or Queue, one particular element is at the *front*. It can be returned without removing it. This is called `peek` in the text, `top` for `Stack` in Scala, and `front` for `Queue` in Scala. I will prefer to say `peekFront` to apply to all the data structures, including Deque for double-ended queues (which Scala currently skips). Target: $O(1)$ time.

5. For a nonempty Stack the front element can be deleted by `pop()`. The parentheses conventionally signify that this mutates the data structure. It is usual to return the popped element, even though you could have gotten it first via `peek`. Target: $O(1)$ time.
6. For a nonempty Queue, the front element can be removed by what Scala calls `dequeue()` but most people say `pop()` here too. Both are `popFront()` in general terms. Target: $O(1)$ time.
7. For any Stack, a new element can be pushed in front via `push(item: A)`. This is the same as prepending. Target: $O(1)$ time.
8. For any Queue, a new element can be added (only) at the rear, i.e., appended. This is `enqueue(item: A)` in Scala, but I will prefer to say `pushRear` (analogous to C++ `push_rear`) for queues---and sometimes clarify `push` as `pushFront` for stacks. Target: $O(1)$ time.
9. Getting the current `size` of the stack or queue. Target: $O(1)$ time.

Notice that the $O(1)$ time targets for push and pushRear automatically imply meeting the $O(n)$ time target for constructing an n-element container. All of these time targets arew asymptotically the best possible.

Here are some logical rules that the **contract** of the ADTs must meet:

- For empty containers, Stack().size = 0 and Queue.size = 0.
- For any stack object `s` and item `a`, (`val m = s.size; s.push(a); val n = s.size`) $\Longrightarrow$ $n = m + 1$. That is to say, a Stack always accepts a new item---quite unlike a Set which can silently reject a new item if it is a duplicate. Similar with a Queue.
- For a Stack object `s` and item `a`, (`s.push(a); val b = s.peekFront`) $\Longrightarrow$ $b = a$. Here we might insist on the object being the same by-reference, rather than just getting b as a possible copy satisfying `b.equals(a)`.
- For a Queue, this doesn't happen---or rather, it happens if and only if the queue was initially empty and now has just the one element `a`.
- And so on---one can express the full First-in, First-Out (FIFO) property of a queue, or rather Last-In, Last-out, as well as Last-In, First-Out (LIFO) for a stack, in logical terms.

Putting the two together, the main data-handling operations we want to support are:

a) `peekFront`
b) `pushFront()`
c) `pushRear()`
d) `popFront()`

How about also

- `peekRear` and
- `popRear()` ?

Adding these last two (with $O(1)$ time targets) defines a **double-ended queue**, called a **Deque** ("deck" or "deke", but don't confuse with `dequeue`). Here is one of the basic dfata-structure facts of life:

> A **singly** linked list can support all of these operations in $O(1)$ time except `popRear()`.

To do all, one needs double links. But this takes a hit on the principal constant, because the basic operations need to update a second link. This is so even for a circularly linked list.

Geeks for Geeks linked list comparisons.

An Array implementation can do all, but this involves the issue of **overflow**. The solution to overflow is "amortized doubling", but that also takes a hit on the principal constant.

Friday: details of Scala implementations, including the "ClassTags" technicality (connected with the need for "null values" of the client type A and the possible use of **sentinels**).