2 Week 2: February 5

2.1 Variants of "ended" lists

We distinguish between five types of lists that allow insertions and deletions only at the ends:

- A **stack** supports insertions and deletions only at the same end; this is consistent with our standard "list" data type. The active end of the stack is usually called the *top*; the insertion operation is called *push*, and the deletion operation is called *pop*. The correct physical metaphor is a stack of plates.
- A **queue** supports insertions only at one end and deletions only from the other end. The insertion end is often called the "back" and the deletion end is often called the "front", to be consistent with the metaphor of a queue of people waiting to get into a Taylor Swift concert. The inertion and deletion operations are often called *enqueue* and *dequeue*, but I prefer the more mnemonic names *push* and *pull*.
- A **double-ended queue** or **deque** (pronounced "deck") supports both insertions and deletions at both ends. The insertion and deletion operators are often called *insert-front*, *insert-back*, *delete-front*, and *delete-back*, or something similar. I prefer the mnemonic names *push*, *shove*, *pop*, and *pull*.[1]
- An **output-restricted deque** or **stack-ended queue** or **steque** (pronounced "steck") supports insertions at both ends, but deletions only at one end. In other words, a steque is a deque that does not support *pull*.[2]
- Am **input-restricted deque** or **queue-ended stack** or **quack** supports deletions at both ends, but insertions only at one end. In other words, a steque is a deque that does not support *shove*.[3]

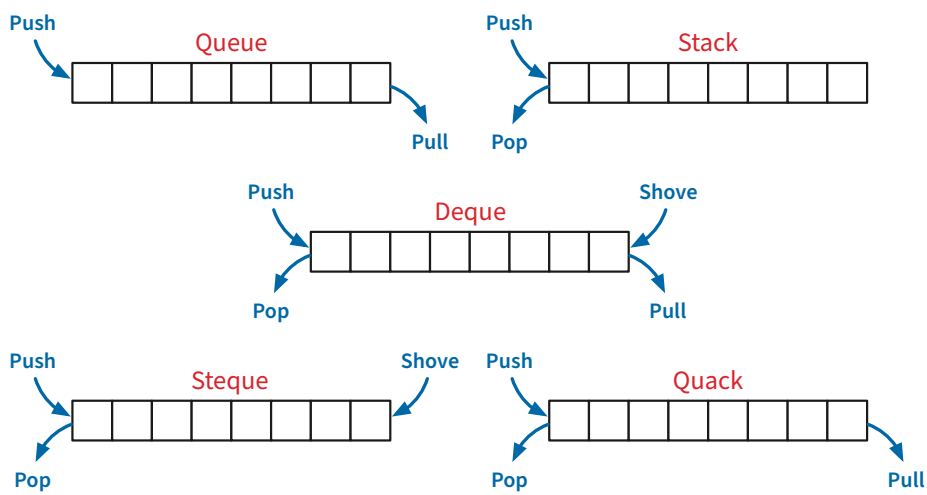

Figure 1: Five types of "ended" lists and their insertion/deletion operations

[1] It's unfortunate that "shove" doesn't start with the letter P, but is there a better alternative? Poke? Prod? Plunge?

[2] We could have forbidden *pop* instead of *pull*, but preserving the standard stack operations is more consistent with the literature.

[3] Again, we could have forbidden *push* instead of *shove*, but ducks really like peas.

All of these data structures can be implemented as dynamically sized array-lists, with $O(1)$-time random access and $O(1)$-amortized-time pushes, pops, pulls, and/or shoves.

For example, we could maintain a data array along with three integers head, tail, and *cap*. As usual, cap stores the size of the data array. The integers head and tail respectively point to the indices in the data array where the next item would be respectively *shoved* or *pushed*. The number of items in the sequence is exactly $(\mathsf{tail} - \mathsf{head} + 1) \bmod \mathsf{cap}$; in particular, the array-list is either completely full or completely empty when either $\mathsf{tail} = \mathsf{head} - 1$ or ($\mathsf{head} = 0$ and $\mathsf{tail} = \mathsf{cap} - 1$). If we double the data array when it becomes full and halve the data array when it becomes sufficiently empty, the amortized analysis is *exactly* the same as Homework 1.

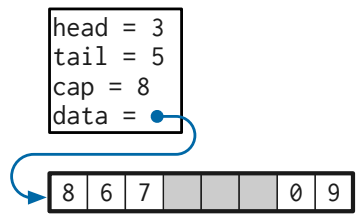

Figure 2: A deque implemented as an array-list

2.2 Simulating a queue with two stacks

See the Homework 2 handout.