## Stacks, Queues, and Priority Queues

## This unit studies three related data structures: stacks, queues, and priority queues.

## Stack

|  |
| --- |
| D |
| C |
| B |
| A |

A common algorithmic task is to process some data set in reverse order. Typically you put some data in temporary storage, then take it out, always in a Last-In, First-Out (LIFO) order. A *stack* is the data structure that was invented to help manage this process. A real-world example of a stack is the dispenser of the trays in the lunch room. You always take the top tray from the top, never from the middle or the bottom. Similarly, the lunch workers always put trays on the top, never at the middle or bottom. The cheat sheet lists the methods in the stack class. Instantiate a stack of strings here:

Every Java Collection, including a stack, has the ability to print itself. You should know that stack.toString() will return a string with **bottom** element in the stack on the left and the *top* element on the right, formatted like [**A**, B, C, *D*].

Efficiency: All java.util.Stack<E> methods execute in O(1) time amortized over several stack operations. This is because the underlying storage is java.util.ArrayList<E> which supports constant time access for add-to-end and remove-from-end. Of course, you do not need to know anything about the underlying ArrayList in order to use the Stack methods.

## Queue

A common algorithmic task is to process some data set in the same order that it was received, which is First-In-First-Out (FIFO) order. An example of a **queue** is a lunch line. People always enter the line at the rear and leave the line at the front. Cuts are not allowed. Similarly, playlists on at least one non-premium service uses a queue to hold a user’s designated songs. The front song in the queue gets played first.

On the APCS Quick Reference Guide, notice that Queue is an interface that is implemented by the concrete class, java.util.LinkedList<E>. Write the code to instantiate a queue of Integers called queue:

What methods are allowed by the Queue interface?

**In our labs, if you instantiate a queue, you are limited to using the four Queue methods.**

Sometimes adding to the rear is called “enqueue,” and removing from the front is called “dequeue.”

Every Java Collection, including a queue, has the ability to print itself. You should know that queue.toString() will return a string with the **front** element in the queue on the left and the *rear* element on the right, formatted like [**A**, B, C, *D*].

Efficiency: All java.util.Queue<E> methods execute in O(1)time amortized over several queue operations. This is because the underlying storage is java.util.LinkedList<E> which is a doubly-linked list that supports constant time access for add-to-end and remove-from-front. Of course, you do not need to know anything about the underlying LinkedList to use a queue.

## Priority Queue

In a **priority queue** the elements enter in a random order but are removed according to their priority. For example:

* letters (of the alphabet) enter the queue randomly, but come out in alphabetical order.
* in the lunch line, all seniors get served first, juniors next, sophomores next, and freshman last.
* in a hospital, patients who are more likely to die, but are not dead yet, get the organ transplant first.
* in a print queue, shorter print runs get processed before the longer print runs.

Instantiate a priority queue of Customers here:

Look at your cheat sheet. What can PriorityQueues do?

Efficiency: How would you implement a priority queue to get the desired behavior? What would be the Big-O for your data structure?

There is a way to implement a priority queue to get O(log n) behavior, but we have to discuss trees and heaps first. Of course, you do not need to know anything about the underlying heap to use a priority queue.