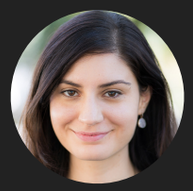
BASIC DATA STRUCTURES

CS062

DATA STRUCTURES AND ADVANCED PROGRAMMING

11: Stacks and Queues



Alexandra Papoutsaki
she/her/hers

Today, we will talk about stacks and queues, the last of the basic data structures.

Lecture 11: Stacks and Queues

- Stacks
- Queues
- Applications
- Java Collections

Some slides adopted from Algorithms 4th Edition and Oracle tutorials

Let's start with stacks. You already have some experience with them through the Calculator assignment.

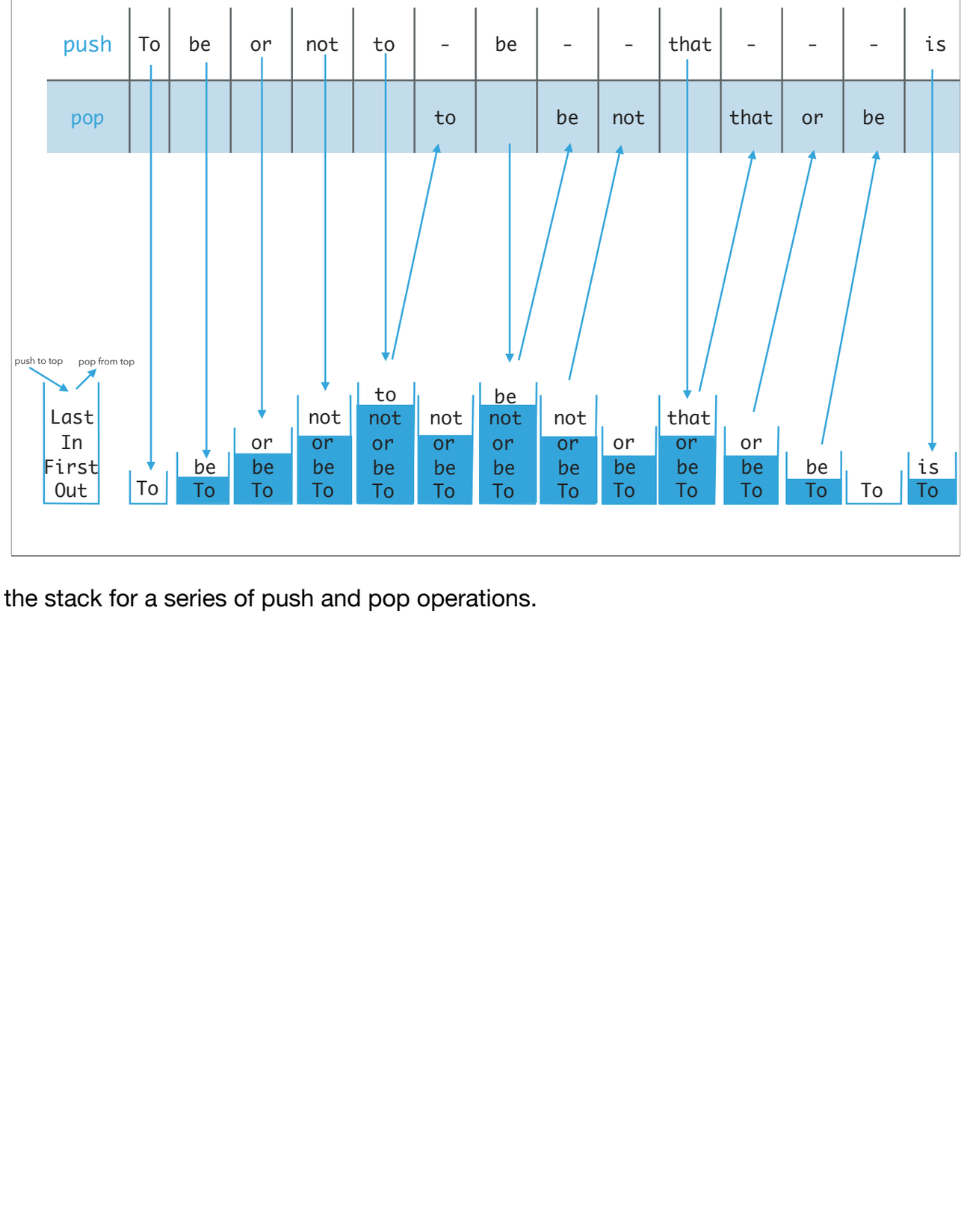
Stacks

- Dynamic linear data structures.
- Elements are inserted and removed following the LIFO paradigm.
- LIFO: Last In, First Out.
  - Remove the most recent element.
- Similar to lists, there is a sequential nature to the data.

- Metaphor of cafeteria plate dispenser.
  - Want a plate? Pop the top plate.
  - Add a plate? Push it to make it the new top.
  - Want to see the top plate? Peek.
  - We want to make push and pop as time efficient as possible.

Stacks are dynamic linear data structures that follow the LIFO paradigm: The element that was last in is the first one to go out. Similar to lists, there is a sequential nature to the data. You can think of stacks as a metaphor for the cafeteria plate dispenser. If you want a plate you pop (that is remove) the top plate. If you want to add a plate, you push it to the top and it becomes the new top of the stack. If you just want to see the top plate, you peek. It makes sense that we want to make the push and pop operations as efficient as possible.

Example of stack operations



Here is an example of how we visualize the stack for a series of push and pop operations.

Implementing stacks with ArrayLists

- Where should the top go to make push and pop as efficient as possible?
- The end/rear represents the top of the stack.
- To push an element `add(E element)`.
  - Adds at the end. Amortized $O^+(1)$.
- To pop an element `remove()`.
  - Removes and returns the element from the end. Amortized $O^+(1)$.
- To peek `get(size()-1)`.
  - Retrieves the last element. $O(1)$.
- If the front/beginning were to represent the top of the stack, then:
  - Push, pop would be $O(n)$ and peek $O(1)$.

What are our choices when it comes to implementing stacks? One option is that we could use arrayLists. If our goal is to make push and pop as efficient as possible, where should the top be? We will use the end/read to represent the stack. That means that pushing an element will be done by calling add and that will have amortized O+(1) running time. To pop an element, we will use remove which will remove and return the element from the end which is again amortized O+(1). To peek, we will use get at the last index which will be done in constant time. If we were to use the front as the top of the stack, then both push and pop would be O(n) and peek O(1). Bad idea.

Implementing stacks with singly linked lists

- Where should the top go to make push and pop as efficient as possible?
- The *head* represents the top of the stack.
- To push an element `add(E element)`.
  - Adds at the head. $O(1)$.
- To pop an element `remove()`.
  - Removes and retrieves from the head. $O(1)$.
- To peek `get(0)`.
  - Retrieves the head. $O(1)$.
- If the last node were to represent the top of the stack, then:
  - Push, pop, peek would all be $O(n)$.

What if we want to implement stacks using singly linked lists? The top of the stack will be the head node. To push, we will call the add method which adds at the head and takes constant time. To pop an element we will all remove which will remove the head again in constant time. Get is also constant. If we were to choose the last node as the top of the stack, then push, pop, and peek would all be linear.

Implementing stacks with doubly linked lists

- Where should the top go to make push and pop as efficient as possible?
- The head represents the top of the stack.
- To push an element `addFirst(E element)`.
  - Adds at the head. $O(1)$.
- To pop an element `removeFirst()`.
  - Removes and retrieves from the head. $O(1)$.
- To peek `get(0)`.
  - Retrieves the head's element. $O(1)$.
- If the *tail* were to represent the top of the stack, we'd need to use `addLast(E element)`, `removeLast()`, and `get(size()-1)` to have $O(1)$ complexity.
- Guaranteed constant performance but memory overhead with pointers.

How about doubly linked lists? We could choose the head to represent the top of the stack. Then pushing would call addFirst in constant time. popping would remove the head by calling removeFirst in O(1). To peek we would get the 0-th index in O(1). We could also do the same with using the tail as the top of the stack again in constant time. How do linked lists compare to array lists? Guaranteed constant performance for push/pop/peek if implemented as we have discussed, but unnecessary memory overhead with extra pointers.

Implementation of stacks

- `Stack.java`: simple interface with `push`, `pop`, `peek`, `isEmpty`, and `size` methods.
- `ArrayListStack.java`: for implementation of stacks with ArrayLists. Must implement methods of `Stack` interface.
- `LinkedStack.java`: for implementation of stacks with singly linked lists. Must implement methods of `Stack` interface.

I highly encourage you to look into the following three files linked at the end of the presentation. Stack is an interface with five abstract methods, push, pop, peek, isEmpty, and size. ArrayListStack and LinkedStack both implement this interface and provide alternative implementations of stacks using array lists or linked lists respectively.

Lecture 11: Stacks and Queues

- Stacks
- Queues
- Applications
- Java Collections

Whenever you hear about stacks, chances are you will also hear about queues, a mirrored concept.

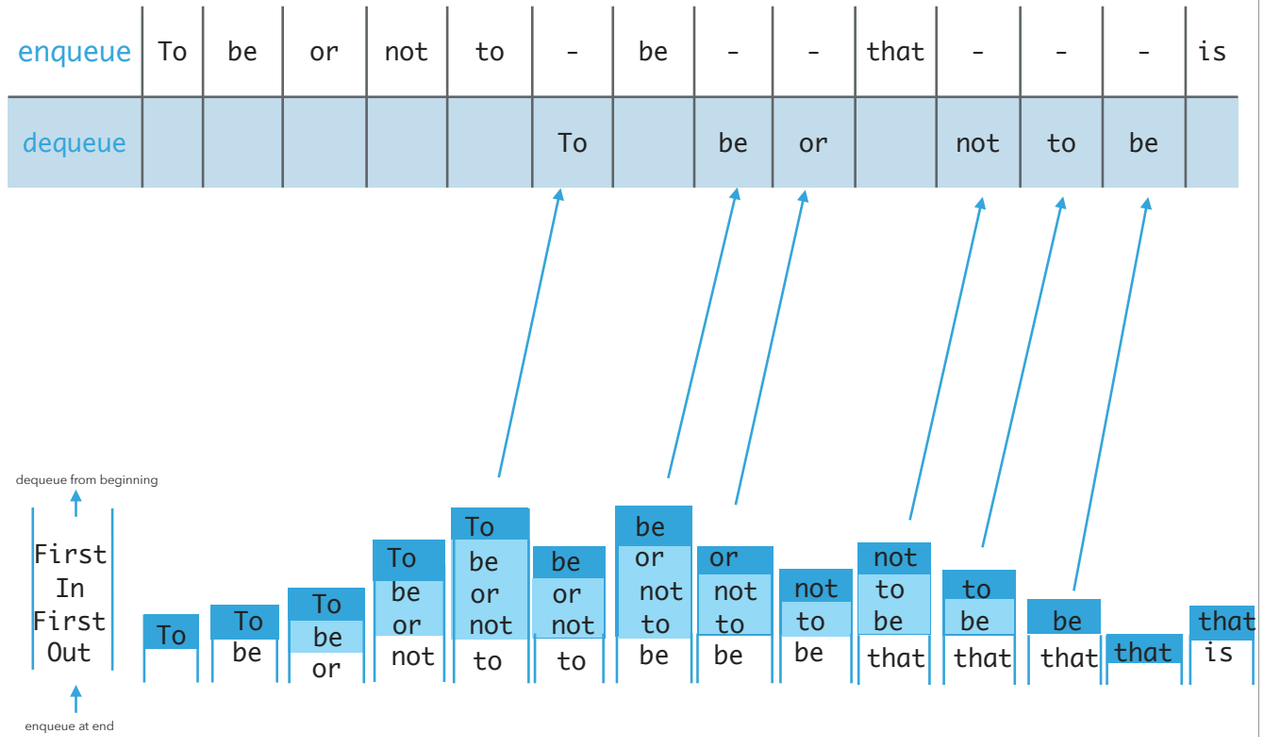
Queues

- Dynamic linear data structures.
- Elements are inserted and removed following the FIFO paradigm.
- FIFO: First In, First Out.
  - Remove the *least* recent element.
- Similar to lists, there is a sequential nature to the data.

- Metaphor of a line of people waiting to buy tickets.
- Just arrived? Enqueue person to the end of line.
- First to arrive? Dequeue person at the top of line.
- We want to make enqueue and dequeue as time efficient as possible.

A queue is a dynamic linear data structure that follows the FIFO paradigm. The first element to be inserted will be the first one to be removed. It's like a queue/line of people waiting to buy a ticket. When we add an element to the queue, we say we enqueue it. When we remove it, we dequeue it. As always, we want to make these basic operations as time efficient as possible.

Example of queue operations



This is an example of how enqueuing and dequeuing works. Note that the visualization of queues here is vertical but they are often presented horizontally.

Implementing queue with ArrayLists

- Where should we enqueue and dequeue elements?
- To enqueue an element `add()` at the end of arrayList. Amortized $O^+(1)$.
- To dequeue an element `remove(0)`. $O(n)$.
- What if we add at the beginning and remove from end?
  - Now dequeue is cheap ($O^+(1)$) but enqueue becomes expensive ($O(n)$).

Let's think how we could implement the concept of a queue with an array list. We can enqueue an element by using add which would append it to the end of the array list. This will result in amortized O+(1). Then we would have to dequeue the first element which would be O(n) (since we need to shift all elements to the left). We could dequeue by removing from the end (amortized O+(1) but now enqueueing would be expensive O(n).

Implementing queue with singly linked list

- Where should we enqueue and dequeue elements?
  - To enqueue an element `add()` at the *head* of SLL ($O(1)$).
  - To dequeue an element `remove(size()-1)` ($O(n)$).
- What if we add at the end and remove from beginning?
  - Now dequeue is cheap ($O(1)$) but enqueue becomes expensive ($O(n)$).
- $O(1)$ for both if we have a tail pointer.
  - enqueue at the tail, dequeue from the head.
  - Simple modification in code, big gains!
  - Version that recommended textbook follows.

What about using singly linked lists? We can enqueue by adding the element to the head which can be done in constant time. To dequeue, we would have to resize the last element which is done in linear time. What if we flip them by enqueuing at the end and dequeuing from the beginning? Now dequeue is cheap (constant) but enqueue becomes expensive (linear). Here is a neat idea. If we add a tail pointer, we can enqueue at the tail and dequeue from the head (Why not the other way around?). This is a tiny modification in our code but it has huge gains!

Implementing queue with doubly linked list

- Where should we enqueue and dequeue elements?
  - To enqueue an element `addLast()` at the tail of DLL ($O(1)$).
  - To dequeue an element `removeFirst()` ($O(1)$).
  - What if we add at the head and remove from tail?
    - Both are $O(1)$!
  - A lot of extra pointers! Also, in practice, "jumping" around the memory can increase significantly the running time.

Let's now think about doubly linked lists. We could enqueue at the tail and dequeue at the head in constant time. Or flip them and again we would do it in constant time. Why might we not prefer this option? Firstly, dlls require a lot of extra pointers! Also in practice "jumping" around the memory can increase significantly the running time.

Implementation of queues

- `Queue.java`: simple interface with enqueue, dequeue, peek, `isEmpty`, and `size` methods.
- `ArrayListQueue.java`: for implementation of queues with ArrayLists. Must implement methods of `Queue` interface.
- `LinkedQueue.java`: for implementation of queues with doubly linked lists. Must implement methods of `Queue` interface.

Please make sure you review the following three files linked at the end of the presentation. Queue is an interface with five abstract methods, enqueue, dequeue, peek, isEmpty, and size. ArrayListQueue and LinkedQueue both implement this interface and provide alternative implementations of queues using array lists or linked lists respectively.

Lecture 11: Stacks and Queues

- Stacks
- Queues
- Applications
- Java Collections

Let's talk a bit about the applications of stacks and queues.

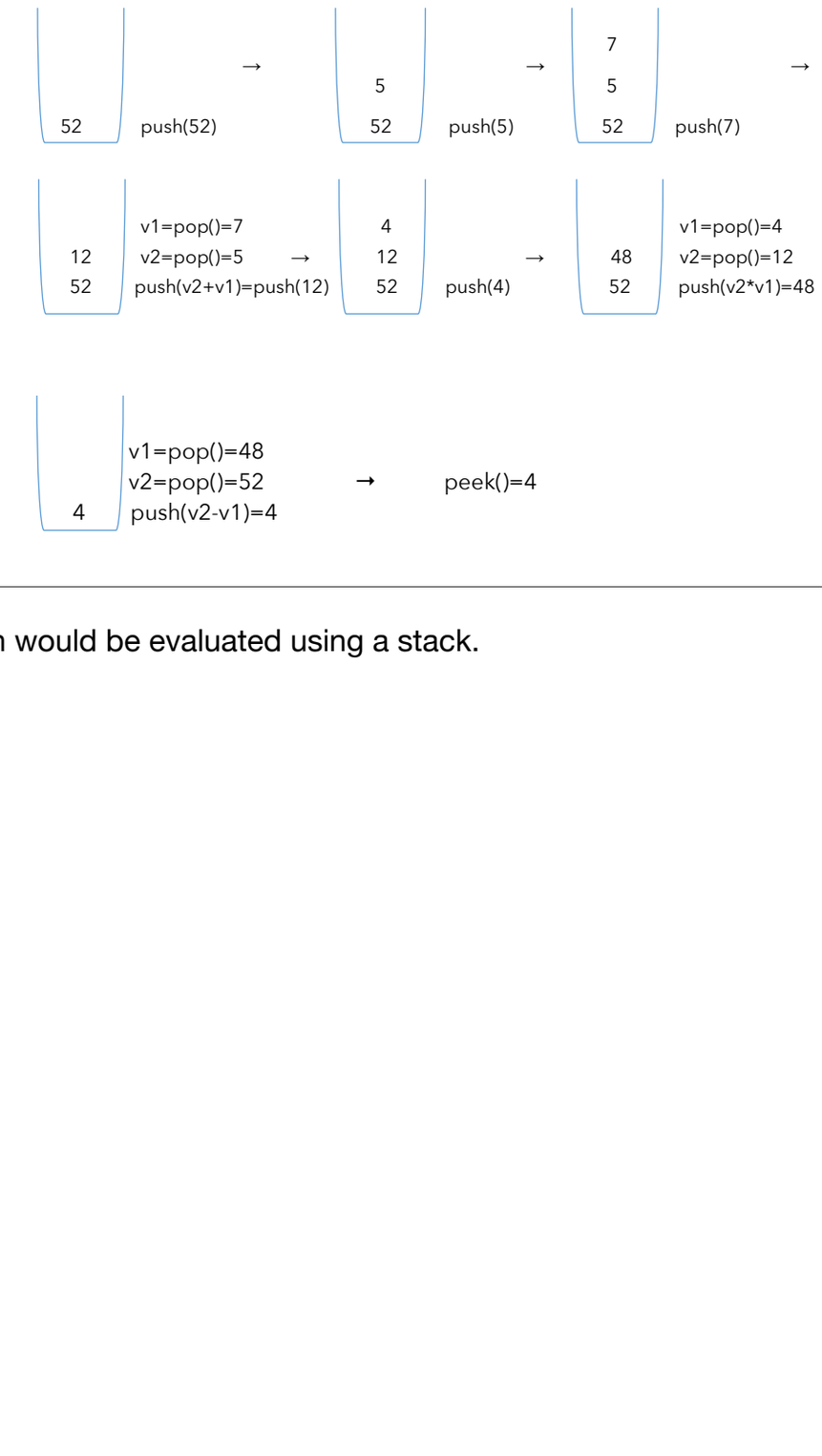
Stack applications

- Java Virtual Machine.
- Basic mechanisms in compilers, interpreters (see CS101).
- Back button in browser.
- Undo in word processor.
- Postfix expression evaluation.

Stacks appear a lot in programming languages and compilers. Think for example the call stack of methods. The back button in your favorite browser is essentially a stack that allows you to pop the latest website you have visited. Similarly, the undo button in Google Docs and Microsoft Word is a stack. And of course, as you know now by the Calculator assignment, you can use a stack to build a postfix expression evaluator.

Postfix expression evaluation example

Example: (52 - ((5 + 7) * 4) ⇒ 52 5 7 + 4 * -



This is an example of how a postfix expression would be evaluated using a stack.

Queue applications

- Spotify playlist.
- Data buffers (netflix, Hulu, etc.).
- Asynchronous data transfer (file I/O, sockets).
- Requests in shared resources (printers).
- Traffic analysis.
- Waiting times at calling center.

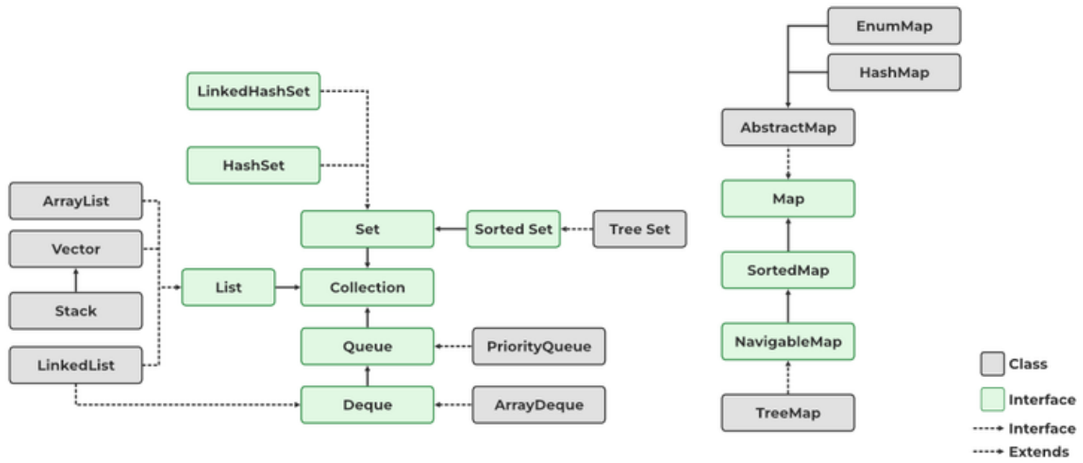
Queues are very common in applications that have to do with streaming, e.g., Spotify, netflix, hulu etc. Printers also are based off queues. Any traffic analysis, airport control, waiting times at a calling center, are all off based queues.

Lecture 11: Stacks and Queues

- Stacks
- Queues
- Applications
- Java Collections

Let's now see what built-in classes Java Collections offer for stacks and queues.

The Java Collections Framework



As a reminder, here is an overview of the Java collections framework.

Deque in Java Collections

- Do not use `Stack`. Deprecated class.
- `Queue` is an interface…
- It's recommended to use the `Deque` interface instead.
  - Double-ended queue (can add and remove from either end).

```
java.util.Deque;

public interface Deque<E> extends Queue<E>
```

- You can choose between `LinkedList` and `ArrayDeque` implementations.

```
Deque deque = new ArrayDeque(); //preferable
```

https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html

Things are a bit muddy. It used to be the case we would use the class Stack but it is no longer recommended. Queue on the other hand is an interface. Instead, we are recommended to use the Deque (deck) interface, which represents a double-ended queue and can be used to either make a stack or a queue. You can choose between LinkedList and ArrayDeque implementations. Experiments have shown that ArrayDeque is actually faster in practice despite the theoretical arguments about linked lists' superiority.

Lecture 11: Stacks and Queues

- Stacks
- Queues
- Applications
- Java Collections

And that's all for today.

Readings:

- Oracle's guides:
  - Collections: https://docs.oracle.com/javase/tutorial/collections/intro/index.html
  - Deque: https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html
  - ArrayList: https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html
- Recommended Textbook:
  - Chapter 1.3 (Page 126–157)
- Recommended Textbook Website:
  - Stacks and Queues: https://algs4.cs.princeton.edu/13stacks/

Code

- Lecture 11 code

Practice Problems:

- 1.3.2–1.3.8, 1.3.32–1.3.33

Make sure you review the code!