CSCI 132:

Basic Data Structures and Algorithms

Stacks and Queues Conclusion, Priority Queue

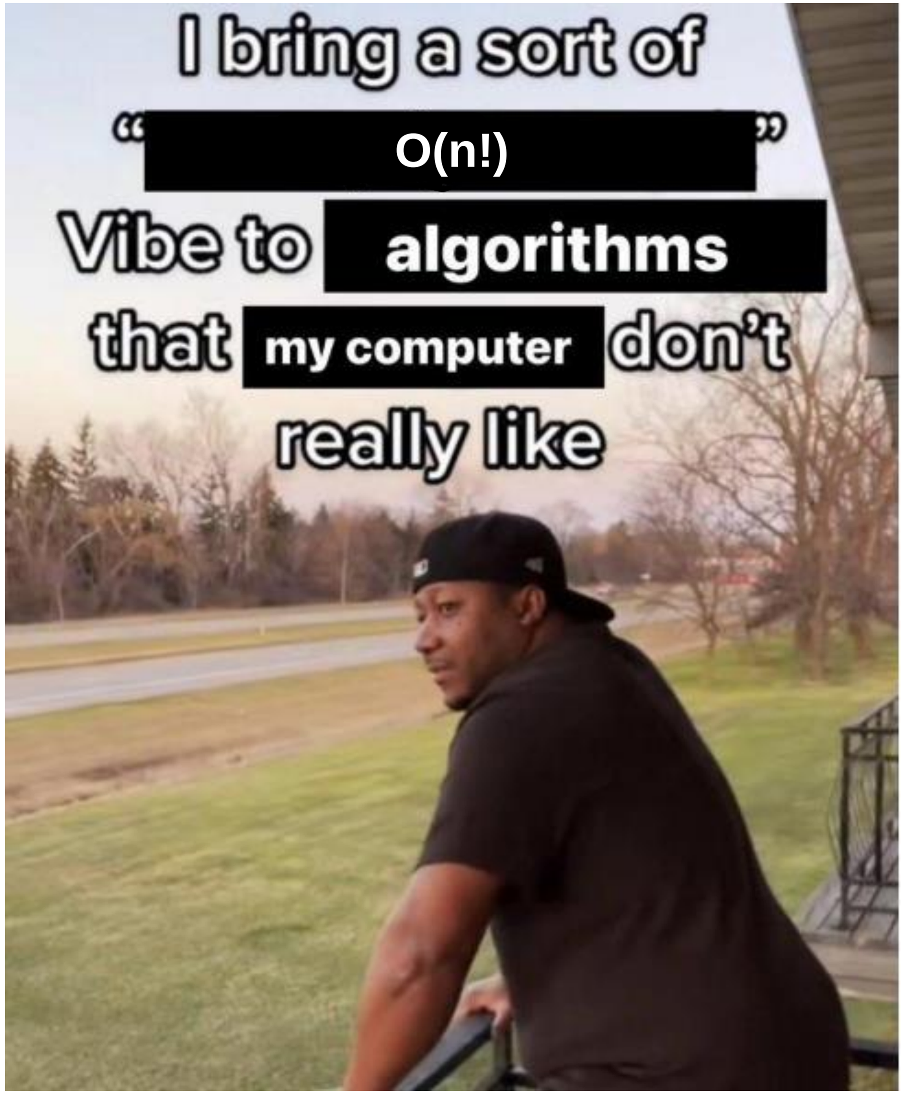
Reese Pearsall

Spring 2025

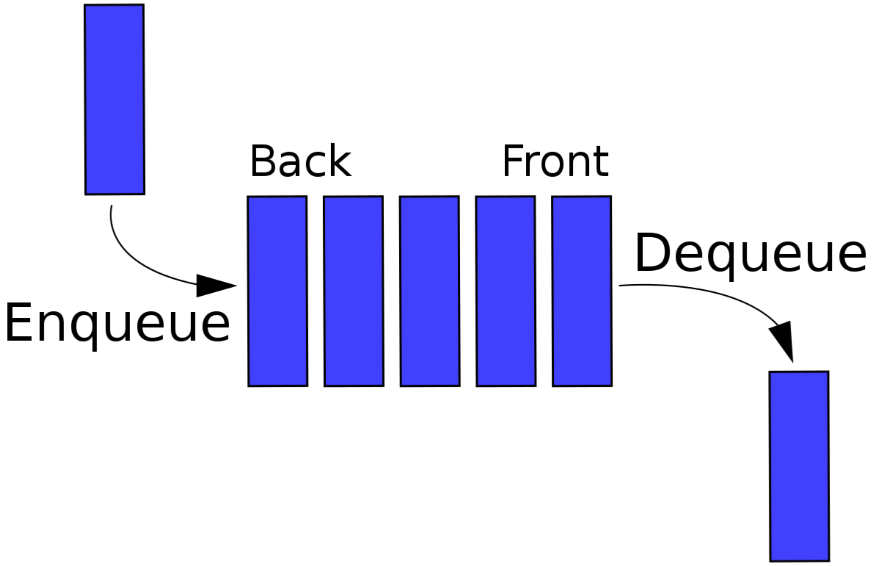
Announcements

Program 3 due **next Friday** at 11:59 PM

Next Friday will be an optional Program 3 help session (no lecture)

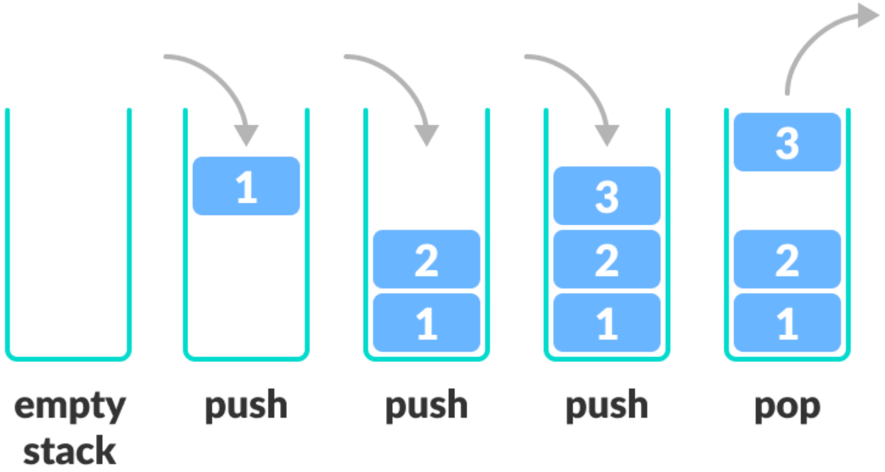


A **Queue** is a data structure that holds data, but operates in a First-in First-out (**FIFO**) fashion



`enqueue(), dequeue()`

A **stack** is a data structure that can hold data, and follows the **last in first out (LIFO)** principle



`push(), pop()`

We implemented both data structures using an Array or a Linked List

Takeaway: Adding and removing elements from a **queue** runs in constant time (O(1))

(FIFO)

Queue Runtime Analysis

	Linked List	**Array**
Creation	O(1)	O(n)
Enqueue	O(1)	O(1)
Dequeue	O(1)	O(1)
Peek	O(1)	O(1)
Print Queue	O(n)	O(n)

Takeaway: Adding and removing elements from a **stack** runs in constant time (O(1))

(LIFO)

Stack Runtime Analysis

	w/ Array	**w/ Linked List**
Creation	O(n)	O(1)
Push()	O(1)	O(1)
Pop()	O(1)	O(1)
peek()	O(1)	O(1)
Print()	O(n)	O(n)

Which data structure should *you* use?

it depends

Data structures always have tradeoffs.

With stacks and queues, the important thing to consider is **the order** of how you want your data to be read

Stacks → LIFO
Queues → FIFO*

Queue Runtime Analysis

	Linked List	Array
Creation	O(1)	O(n)
Enqueue	O(1)	O(1)
Dequeue	O(1)	O(1)
Peek	O(1)	O(1)
Print Queue	O(n)	O(n)

Stack Runtime Analysis

	w/ Array	w/ Linked List
Creation	O(n)	O(1)
Push()	O(1)	O(1)
Pop()	O(1)	O(1)
peek()	O(1)	O(1)
Print()	O(n)	O(n)

Applications of Queue Data Structures

- Online waiting rooms
- Operating System task scheduling
- Web Server Request Handlers
- Network Communication
- CSCI 232 Algorithms

Queue Runtime Analysis

	Linked List	**Array**
Creation	O(1)	O(n)
Enqueue	O(1)	O(1)
Dequeue	O(1)	O(1)
Peek	O(1)	O(1)
Print Queue	O(n)	O(n)

Applications of Stack Data Structures

- Tracking function calls in programming
- Web browser history
- Undo/Redo buttons
- Recursion/Backtracking
- CSCI 232 Algorithms

Stack Runtime Analysis

	w/ Array	**w/ Linked List**
Creation	O(n)	O(1)
Push()	O(1)	O(1)
Pop()	O(1)	O(1)
peek()	O(1)	O(1)
Print()	O(n)	O(n)

In the real world, when you want to use a Queue, Stack, Deque, or a Priority Queue, you will likely import this data structure

```
import.java.util.Stack
```

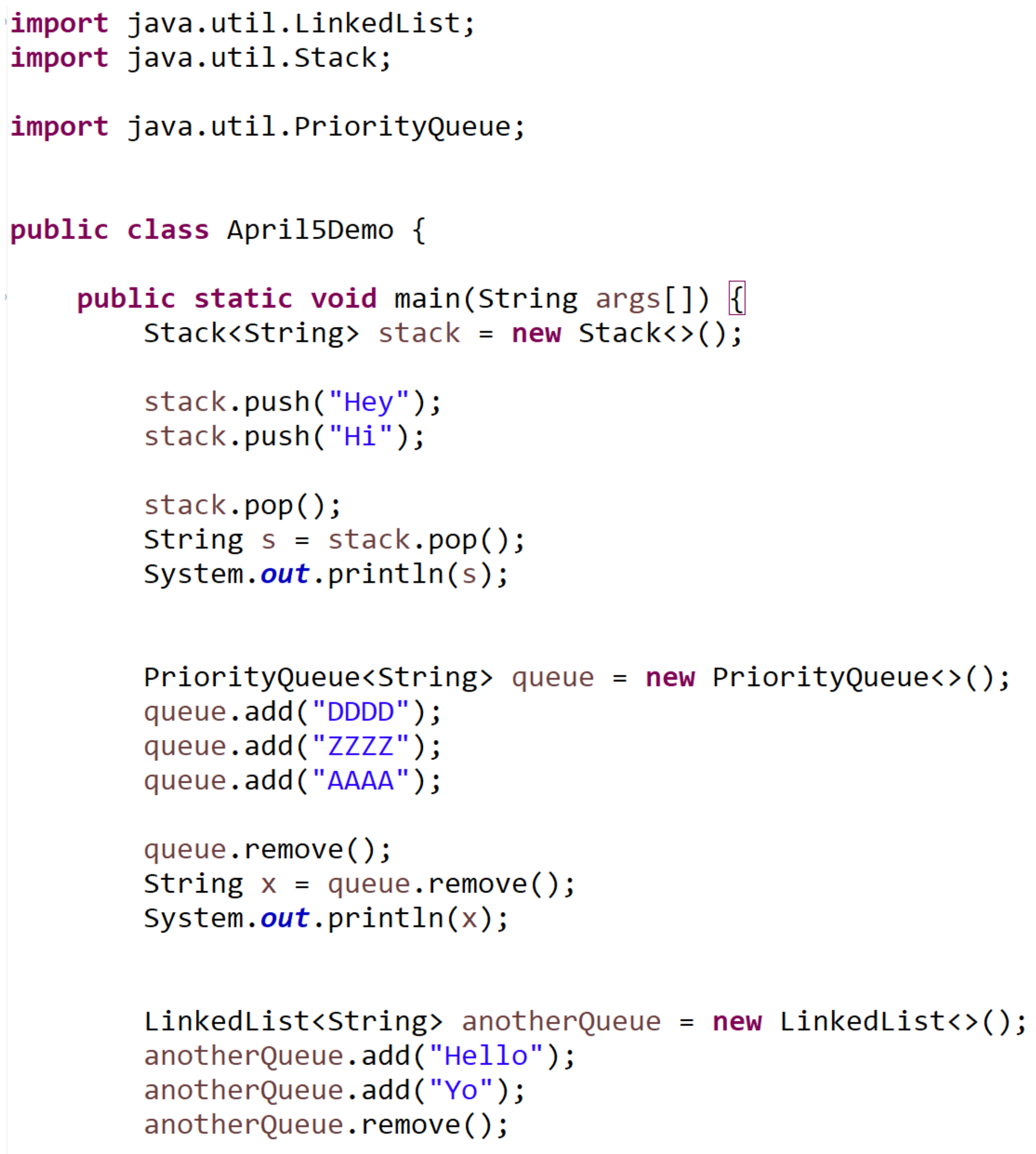
```
import.java.util.Queue
```

`java.util.Queue` is an interface. We cannot create a Queue object.
Instead, we create an instance of an object *that implements* this interface

Some of the Classes that implement the Queue interface:

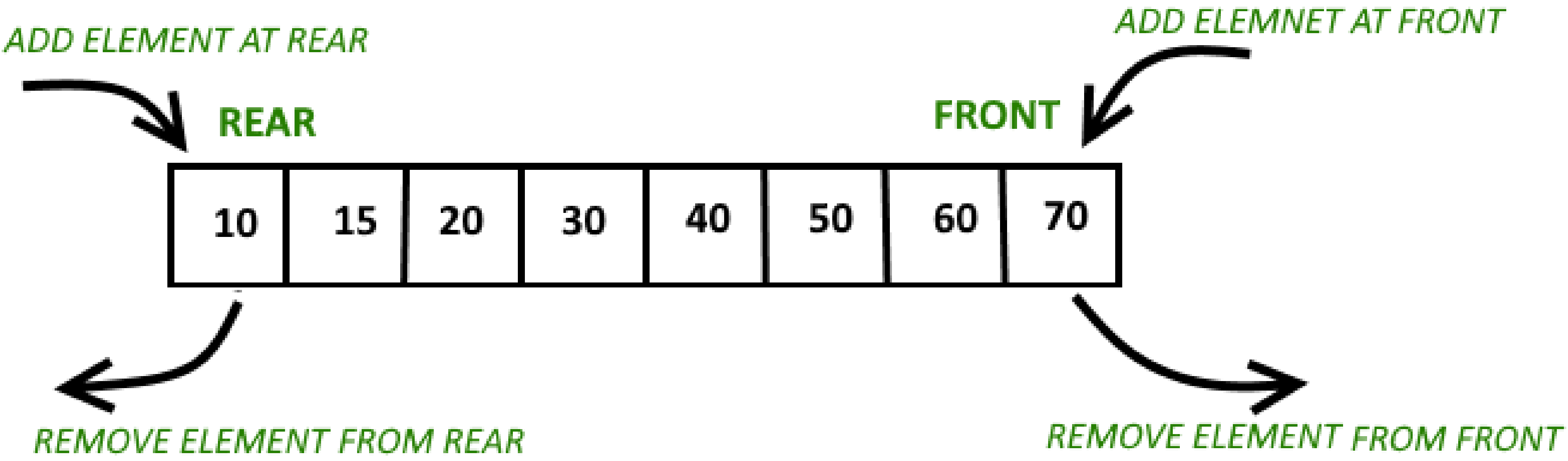
1. PriorityQueue (`java.util.PriorityQueue`)
2. Linked List (`java.util.LinkedList`)

(If you need a FIFO queue, Linked List is the way to go…)



```
import java.util.LinkedList;
import java.util.Stack;

import java.util.PriorityQueue;


public class April5Demo {

    public static void main(String args[]) {
        Stack<String> stack = new Stack<>();

        stack.push("Hey");
        stack.push("Hi");

        stack.pop();
        String s = stack.pop();
        System.out.println(s);


        PriorityQueue<String> queue = new PriorityQueue<>();
        queue.add("DDDD");
        queue.add("ZZZZ");
        queue.add("AAAA");

        queue.remove();
        String x = queue.remove();
        System.out.println(x);


        LinkedList<String> anotherQueue = new LinkedList<>();
        anotherQueue.add("Hello");
        anotherQueue.add("Yo");
        anotherQueue.remove();
```

A double-ended queue, or a **deque** (deck) is a type of queue in which insertion and removal of elements can either be performed from the front or the rear



ADD ELEMENT AT REAR

ADD ELEMNET AT FRONT

REAR

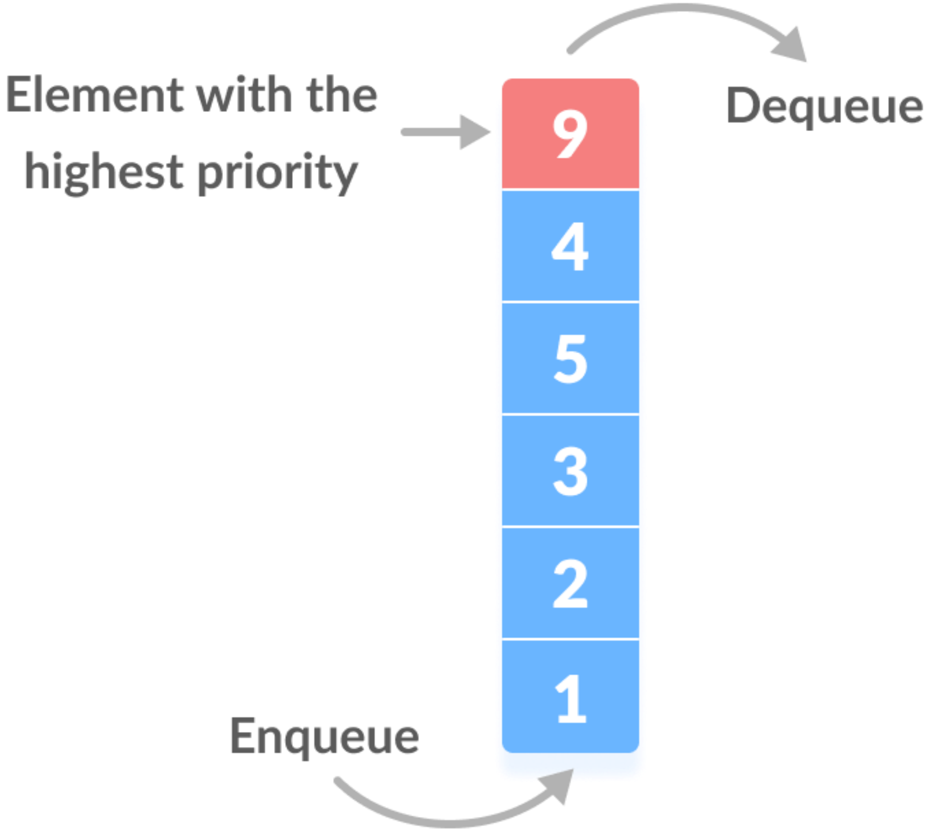
FRONT

10	15	20	30	40	50	60	70

REMOVE ELEMENT FROM REAR

REMOVE ELEMENT FROM FRONT

Most of the time, queues will operate in a FIFO fashion, however there may be times we want to dequeue the item with the **highest priority**



Priority queue in a data structure is an extension of a linear queue that possesses the following properties: Every element has a certain priority assigned to it

When we enqueue something, we might need to "shuffle" that item into the correct spot of the priority queue