CSCI 132:
Basic Data Structures and Algorithms

Queues (Array Implementation)

Reese Pearsall
Spring 2023

Announcements

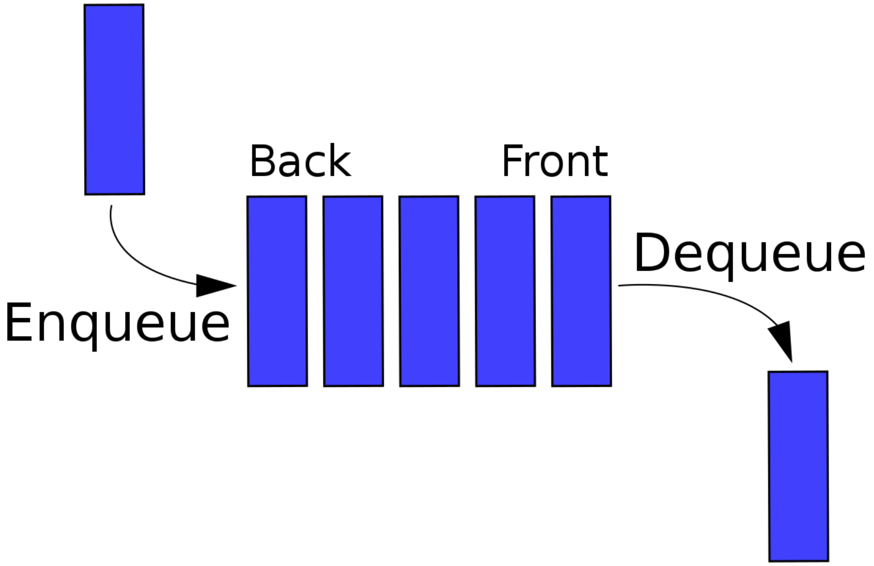
No class on Friday

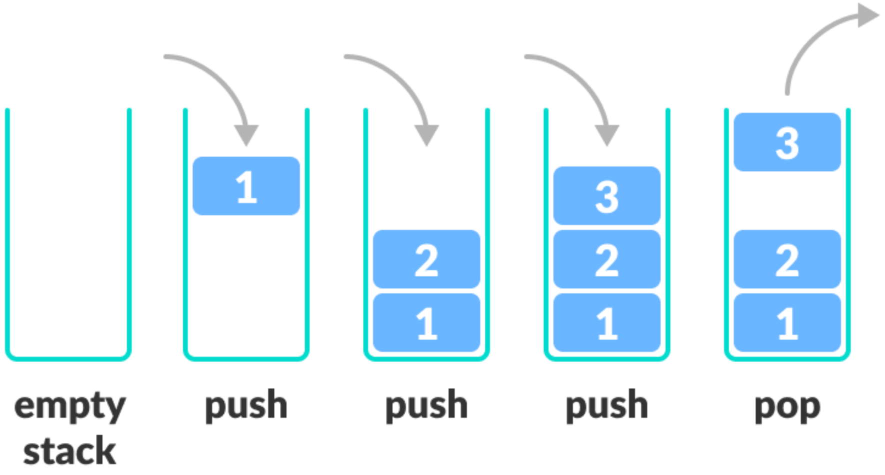
Program 4 due Wednesday 4/19*

Monday’s lecture (4/10) will be asynchronous (don’t come to class)

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

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





`enqueue(), dequeue()`

`push(), pop()`

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

Queue Runtime Analysis

```
public QueueLinkedList() {
        this.orders = new LinkedList<Order>();
        this.size = 0;
}
```

```
public QueueArray2() {
        this.orders = new Order[6];
        this.size = 0;
        this.front = 0;
        this.capacity = this.orders.length; //6
}
```

	Linked List	Array
Creation		
Enqueue		
Dequeue		
Peek		
Print Queue		

Queue Runtime Analysis

```
public QueueLinkedList() {
        this.orders = new LinkedList<Order>();
        this.size = 0;
}
```

O(1)

```
public QueueArray2() {
        this.orders = new Order[6];
        this.size = 0;
        this.front = 0;
        this.capacity = this.orders.length; //6
}
```

O(n), n = | array |

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

Queue Runtime Analysis

```
public void enqueue(Order newOrder) {

        this.orders.addLast(newOrder);
        this.size++;

}
```

```
public void enqueue(Order newOrder) {

        if(this.size == this.capacity) {
                System.out.println("Error... queue is full");
                return;
        }

        int insert_spot = (front + size) % capacity;
        this.orders[insert_spot] = newOrder;

        this.size++;
        System.out.println("Added " + newOrder.getName() + " at index #" + insert_spot);
}
```

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

Queue Runtime Analysis

```java
public void enqueue(Order newOrder) {

        this.orders.addLast(newOrder); O(1)
        this.size++; O(1)

}
```

O(1)

```java
public void enqueue(Order newOrder) {

        if(this.size == this.capacity) {
                System.out.println("Error... queue is full");
                return;
        }
        int insert_spot = (front + size) % capacity;
        this.orders[insert_spot] = newOrder;

        this.size++;
        System.out.println("Added " + newOrder.getName() + " at index #" + insert_spot);
}
```

O(1) O(1) O(1) O(1) O(1)

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

Queue Runtime Analysis

```java
public Order dequeue() {
        if(this.size != 0) {
                Order removed = this.orders.removeFirst();
                System.out.println(removed.getName() + "'s order
                size--;
                return removed;
        }
        else {
                return null;
        }
}
```

```java
public void dequeue() {

        if(this.size == 0) {
                System.out.println("Error... queue is empty");
                return;
        }
        else {
                Order o = this.orders[front];
                this.orders[front] = null;
                front = (front + 1) % capacity;
                this.size--;
                System.out.println(o.getName() + "'s order was removed");

        }

}
```

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

Queue Runtime Analysis

```
public Order dequeue() {
        if(this.size != 0) {
                Order removed = this.orders.removeFirst();
                System.out.println(removed.getName() + "'s order
                size--;
                return removed;
        }
        else {
                return null;
        }
}
```

O(1)

O(1)

```
public void dequeue() {

        if(this.size == 0) {
                System.out.println("Error... queue is empty");
                return;
        }
        else {
                Order o = this.orders[front];
                this.orders[front] = null;
                front = (front + 1) % capacity;
                this.size--;
                System.out.println(o.getName() + "'s order was removed");

        }

}
```

O(1)

O(1)

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

Queue Runtime Analysis

```
return this.orders.getFirst()
```

```
return this.orders[front]
```

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

Queue Runtime Analysis

`return this.orders.getFirst()` O(1)

`return this.orders[front]` O(1)

	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		

Queue Runtime Analysis

```
public void printQueue() {
        int counter = 1;
        for(Order each_order: this.orders) {
                each_order.printOrder(counter);
                counter++;
        }
}
```

```
public void printQueue() {

        int start = front;
        int counter = 1;
        int n = 0;
        while(n != this.size) {
                System.out.println(counter + ". " + this.orders[start].getName());
                start = (start + 1) % capacity;
                counter++;
                n++;
        }


}
```

	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		

Queue Runtime Analysis

```
public void printQueue() {
        int counter = 1; O(1)
        for(Order each_order: this.orders) {O(n)
            O(1)each_order.printOrder(counter);
            O(1)counter++;
        }
}
```

n = # of elements in queue

```
public void printQueue() {

        int start = front; O(1)
        int counter = 1; O(1)
        int n = 0;O(1)
        while(n != this.size) { O(n)
                System.out.println(counter + ". " + this.orders[start].getName());
        O(1)    start = (start + 1) % capacity;
                counter++;
                n++;
        }

}
```

n = # of elements in queue

	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 **queue** runs in constant time (`O(1)`)

(FIFO)

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

(LIFO)

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)

Which data structure should *you* use?

it depends

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)

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)

Queue Runtime Analysis

Applications of Queue Data Structures

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

	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)

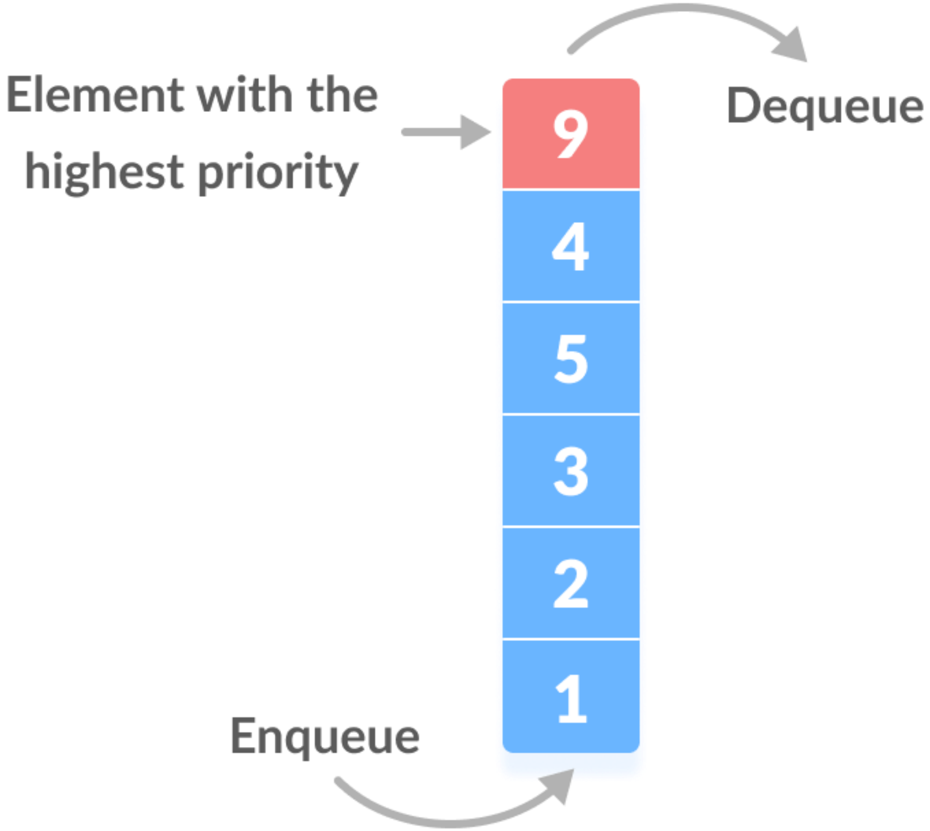
Stack Runtime Analysis

Applications of Stack Data Structures

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

	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)

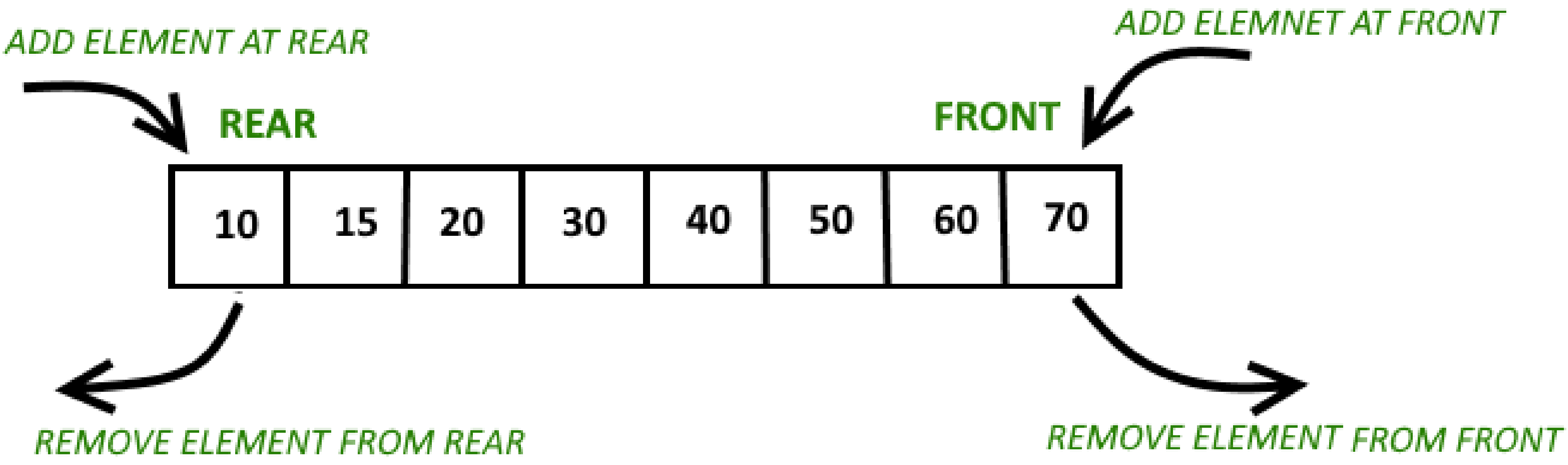
Most of the time, queues will operate in a FIFO fashion, however there may be times we want to dequeue the item with the **highest priority**



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

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



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…)

```java
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();
```