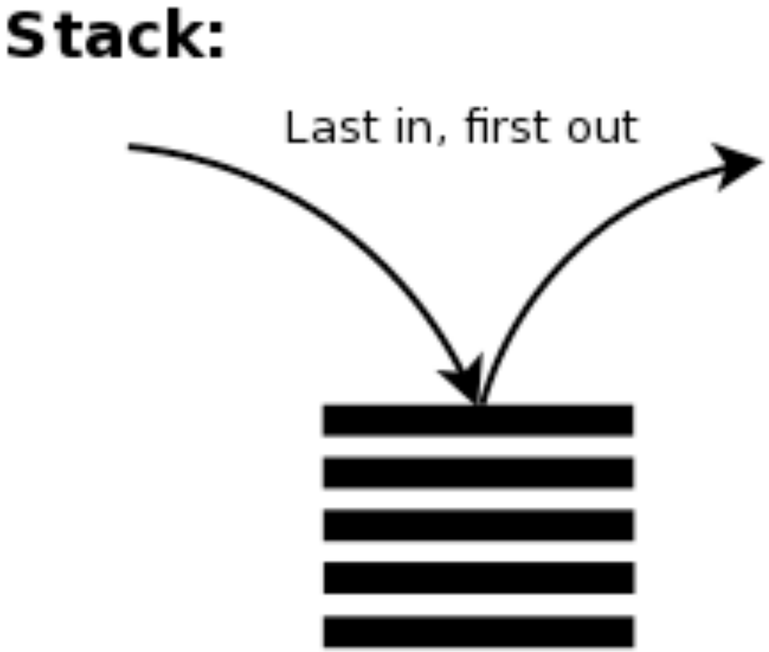
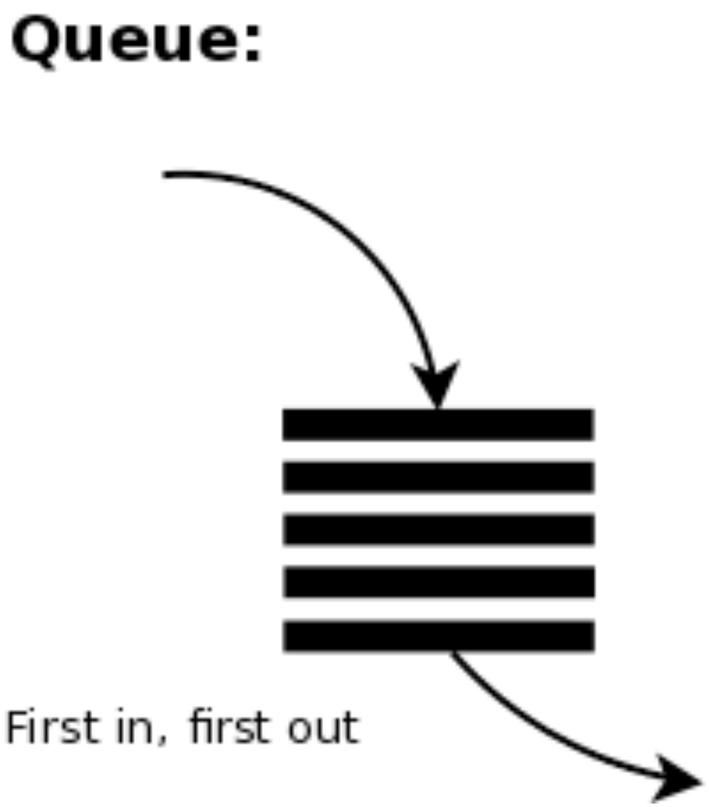
CS62 Class 10: Stacks & Queues

Basic Data Structures





https://gohighbrow.com/stacks-and-queues/

Agenda

- Darwin winners!
- Stacks: implementation
- Queues: implementation, history, affordance analysis
- Java Collections Framework (JCF)

Darwin winners!

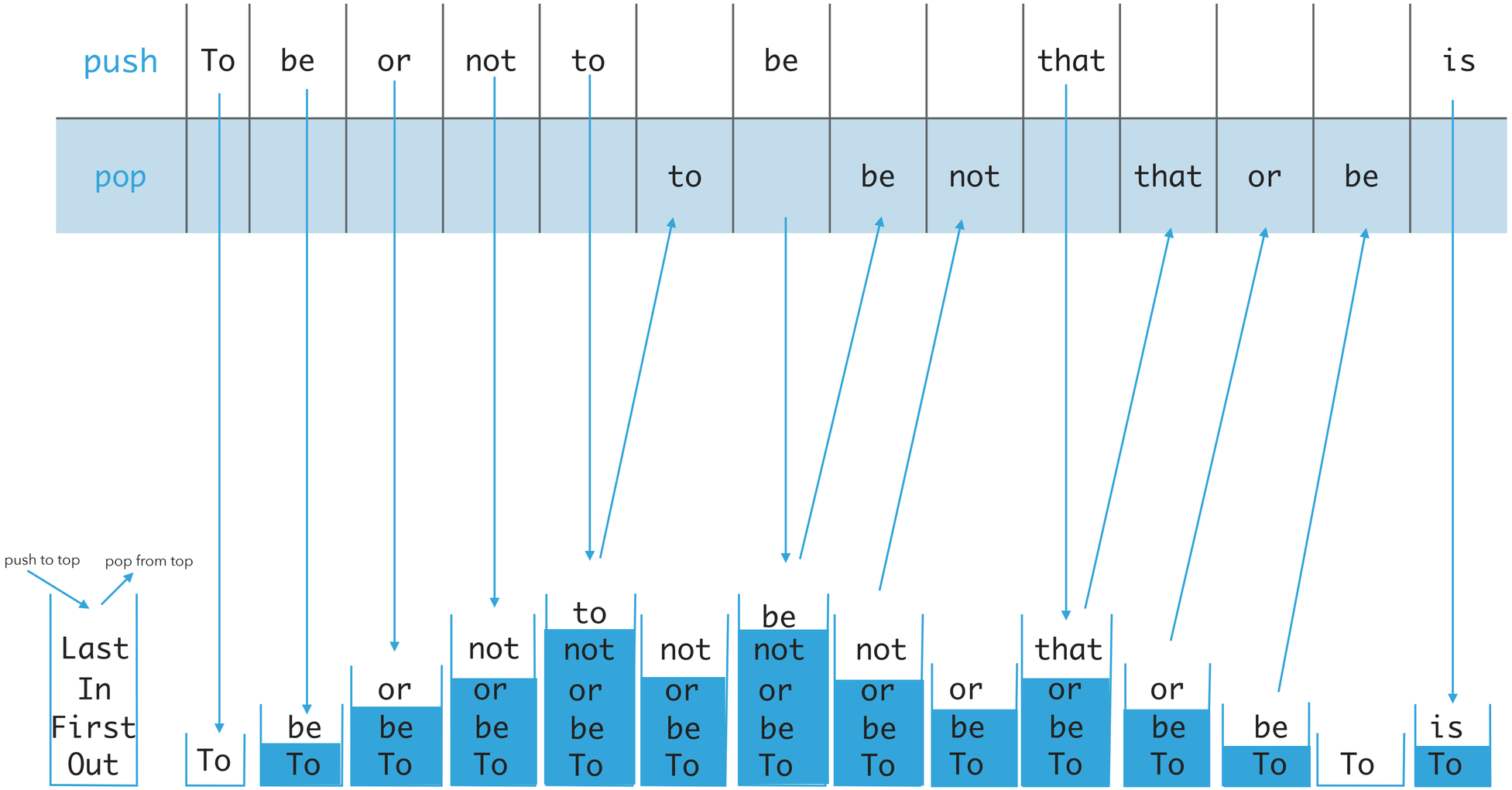
Stacks

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.

Example of stack 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. $O(1)$.
- *Q: What if the front/beginning were to represent the top of the stack? What are the run times?*
  - Push, pop would be $O(n)$ and peek $O(1)$.
    - This is because adding/removing to the front requires shifting all the elements to the right which takes O(n) time. But indexing into an Array is always O(1).

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. $O(1)$.
- *Q: What if the last node was to represent the top of the stack? What are the run times?*
  - Push, pop, peek would all be $O(n)$.
    - This is because we need to iterate through all the SLL's .next pointers.

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.

Implementation of stacks (see linked code)

- `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.

Worksheet time!

1. Suppose you use a stack to perform an intermixed sequence of push and pop operations. The push operations put the integers 0 through 9 in order onto the stack. You can pop the top of the stack at any time. Which of the following sequence(s) of pops are valid?

Hint: draw out the stack!

a. 4 3 2 1 0 9 8 7 6 5

b. 4 6 8 7 5 3 2 9 0 1

c. 2 5 6 7 4 8 9 3 1 0

d. 0 4 6 5 3 8 1 7 2 9

Worksheet answers

1. Suppose you use a stack to perform an intermixed sequence of push and pop operations. The push operations put the integers 0 through 9 in order onto the stack. You can pop the top of the stack at any time. Which of the following sequence(s) of pops are valid?

a. 4 3 2 1 0 9 8 7 6 5 — Yes

b. 4 6 8 7 5 3 2 9 0 1 — No because of the 0

c. 2 5 6 7 4 8 9 3 1 0 — Yes

d. 0 4 6 5 3 8 1 7 2 9 — No because of the 1

Stack applications

- Java Virtual Machine.
- Basic mechanisms in compilers, interpreters (see CS101).
- Back button in browser.
- Undo in word processor.
- Postfix expression evaluation (see HW4).

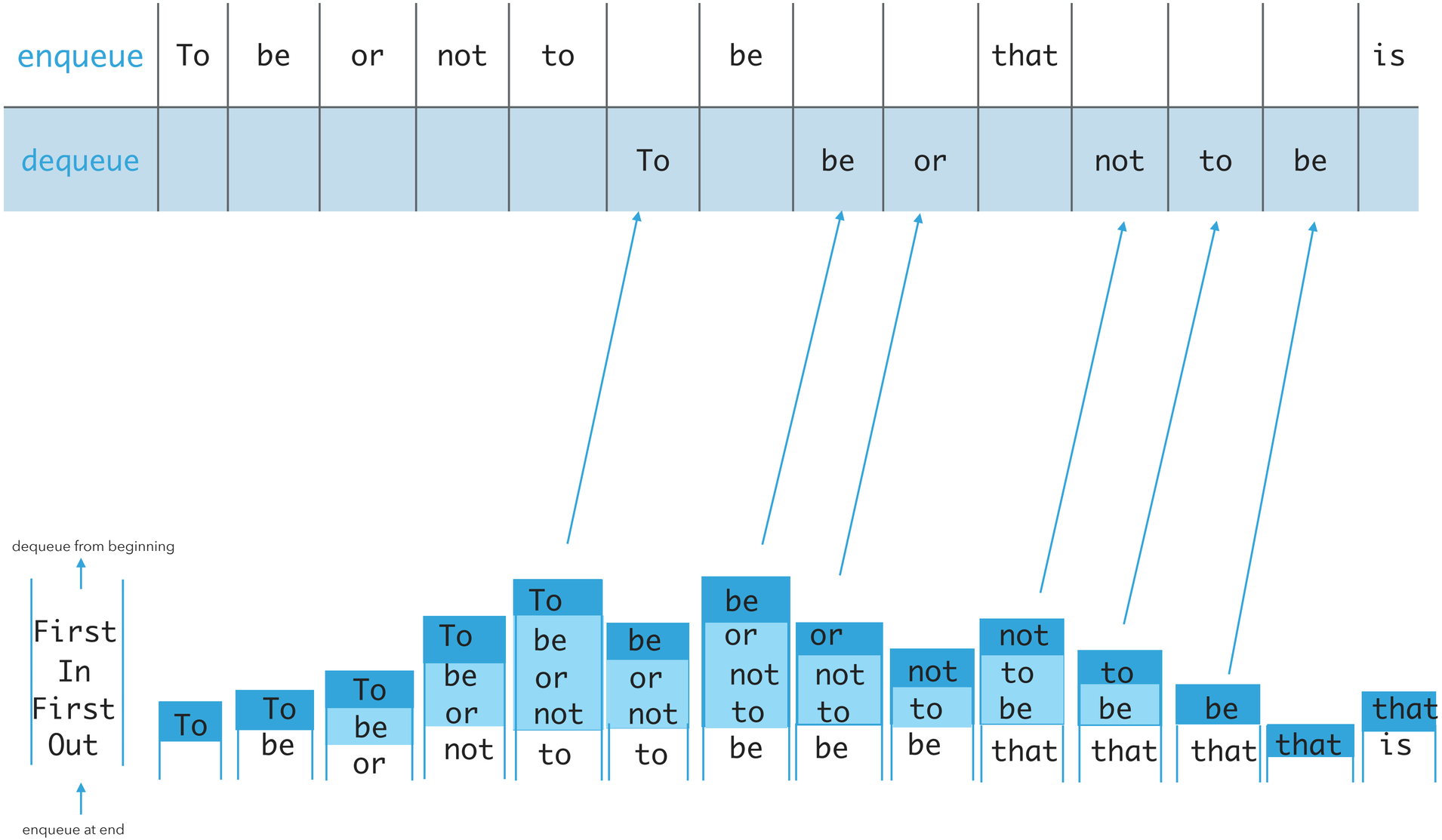
Queues

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.

Example of queue operations



Worksheet time!

2. Suppose you use a queue to perform an intermixed sequence of enqueue and dequeue operations. The enqueue operations put the integers 0 through 9 in order in the queue. You can dequeue at any time. Which of the following sequence(s) of dequeues are valid?

   a. 4 3 2 1 0 9 8 7 6 5

   b. 0 1 2 3 4 5 6 7 8 9

   c. 0 4 6 5 3 8 1 7 2 9

   d. 0 1 2 3 5 6 7 9 8 4

Worksheet answers

2. Suppose you use a queue to perform an intermixed sequence of enqueue and dequeue operations. The enqueue operations put the integers 0 through 9 in order in the queue. You can dequeue at any time. Which of the following sequence(s) of dequeues are valid?

a. 4 3 2 1 0 9 8 7 6 5 No

b. 0 1 2 3 4 5 6 7 8 9 Yes

c. 0 4 6 5 3 8 1 7 2 9 No

d. 0 1 2 3 5 6 7 9 8 4 No

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 ($O(n)$).

This isn't great - we always will have a constant time operation. Can linked lists do better?

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.

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.

So, in reality, a modified singly linked list with a tail pointer is the “best” implementation of a queue.

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.

Worksheet time!

- Think of a common real life application for a stack. How would it change if we used a queue?
- Think of a common real life application for a queue. How would it change if we used a stack?

Worksheet time!

- Match the description to the Java code snippet.

a. To-do list b. Inserts a task into a to-do list c. Retrieves a task from a to-do list d. Can be used to reverse characters in a word e. A list of to-do lists f. Inserts a to-do list into a list	1. `q2.enqueue(q1);` 2. `Queue<Queue<String>> q2 = new Queue<Queue<String>>();` 3. `Queue<String> q1 = new Queue<String>();` 4. `q1.enqueue("Pay bills.");` 5. `String s = q1.dequeue();` 6. `Stack<Character> s1 = new Stack<Character>();`

Worksheet answers

- Match the description to the Java code snippet.

a. To-do list 3 b. Inserts a task into a to-do list 4 c. Retrieves a task from a to-do list 5 d. Can be used to reverse characters in a word 6 e. A list of to-do lists 2 f. Inserts a to-do list into a list 1	1. `q2.enqueue(q1);` 2. `Queue<Queue<String>> q2 = new Queue<Queue<String>>();` 3. `Queue<String> q1 = new Queue<String>();` 4. `q1.enqueue("Pay bills.");` 5. `String s = q1.dequeue();` 6. `Stack<Character> s1 = new Stack<Character>();`

History of queues

- Queues have existed in math for a long time (queueing theory, 1909 by Danish mathematician Agner Krarup Erlang - studying how to improve telephone call centers, proved that you could apply a Poisson distribution to model the problem)
- World War II made research into queueing theory very popular to support wartime efforts
  - During the Cold War, British mathematician David Kendall used queueing theory to manage supplies in the Berlin Airlift

https://cs.pomona.edu/classes/cs62/history/stack&queue/

Modern queue applications

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

Affordance analysis of queues

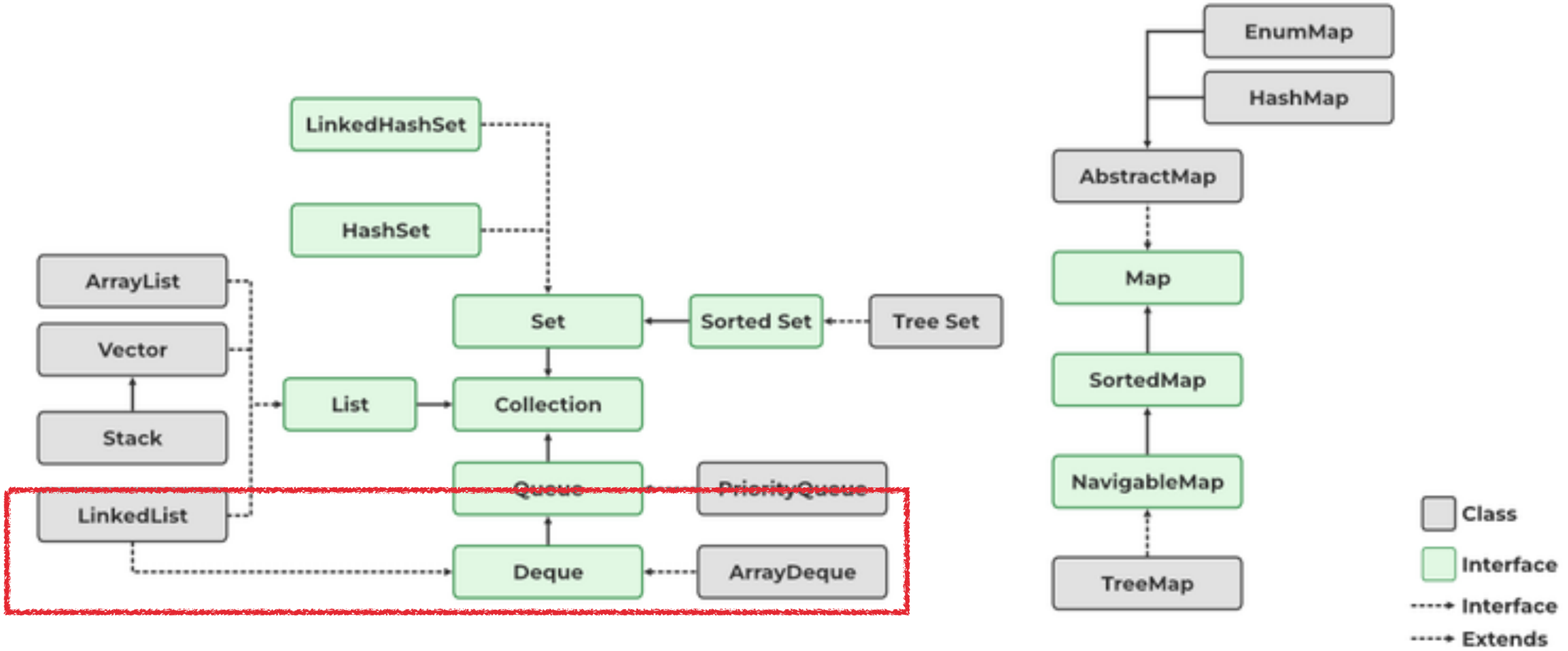
- Who is left behind, and who is prioritized in the design of a data structure/ algorithm? What is the human cost of efficiency?
- Discussion Q: Consider some modern applications of queues: hospital emergency rooms, airport security lines, or customer support centers. Should these systems always operate on a FIFO queue, or should some people be given priority? How do we balance fairness with efficiency in such systems? How would you change any modern queue application to make it, in your opinion, more “fair”? Who benefits, and who is left out?

https://cs.pomona.edu/classes/cs62/history/stack&queue/

https://arxiv.org/pdf/2101.00786

JCF

“Deque” interface & ArrayDeque (or LinkedList)



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).

```
import 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

Lecture 10 wrap-up

- Exit ticket: https://forms.gle/JPZZ89BRCjpN4t5G9
- HW4 due Tues 11:59pm
- Lecture 10 is the last lecture that will be on Checkpoint 1! Checkpoint 1 is next next Monday March 4. Lab next week will be a review session.
  - If you have SDRC accommodations, let them know ASAP (schedule a proctoring time)
- If you're an underrepresented minority in CS and want to go to the ACM Tapia conference in Dallas next fall (good for networking, bad for finding jobs), let the department know ASAP (and fill out a conference scholarship application by 3/4): https://tinyurl.com/tapia2025

Resources

- Stacks & queues from the textbook: https://algs4.cs.princeton.edu/13stacks/
- Oracle Queue: https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html
- See slides following this for 2 more practice problems

Bonus practice problem

- Write a method `isBalanced` that given a String of parentheses and curly brackets, it determines whether they are properly balanced. For example, it should return true if given "[()]{}{[()()]()}" and false for "[(])".

Bonus answer

```
private boolean isBalanced(String input) {
       Stack<Character> stack = new Stack<>();
        for (char parenthesis : input.toCharArray()) {
            if (parenthesis == '('  || parenthesis == '[' || parenthesis ==
'{') {
                stack.push(parenthesis);
            } else {
                if (stack.isEmpty()) {
                    return false;
                }
                char top = stack.pop();

                if (parenthesis == ')' && top != '(' || parenthesis == ']'
&& top != '['  || parenthesis == '}' && top != '{') {
                    return false;
                }
            }
        }
        return stack.isEmpty();
    }
```

Bonus practice problem 2

- What does the following code fragment do to the queue q?

```
Stack<String> stack = new Stack<>();
while (!q.isEmpty()) {
    stack.push(q.dequeue());
}
while (!stack.isEmpty()) {
    q.enqueue(stack.pop());
}
```

Bonus answer 2

It inverts the queue, i.e., flips the order of the queue (the first element in the queue is now the last).