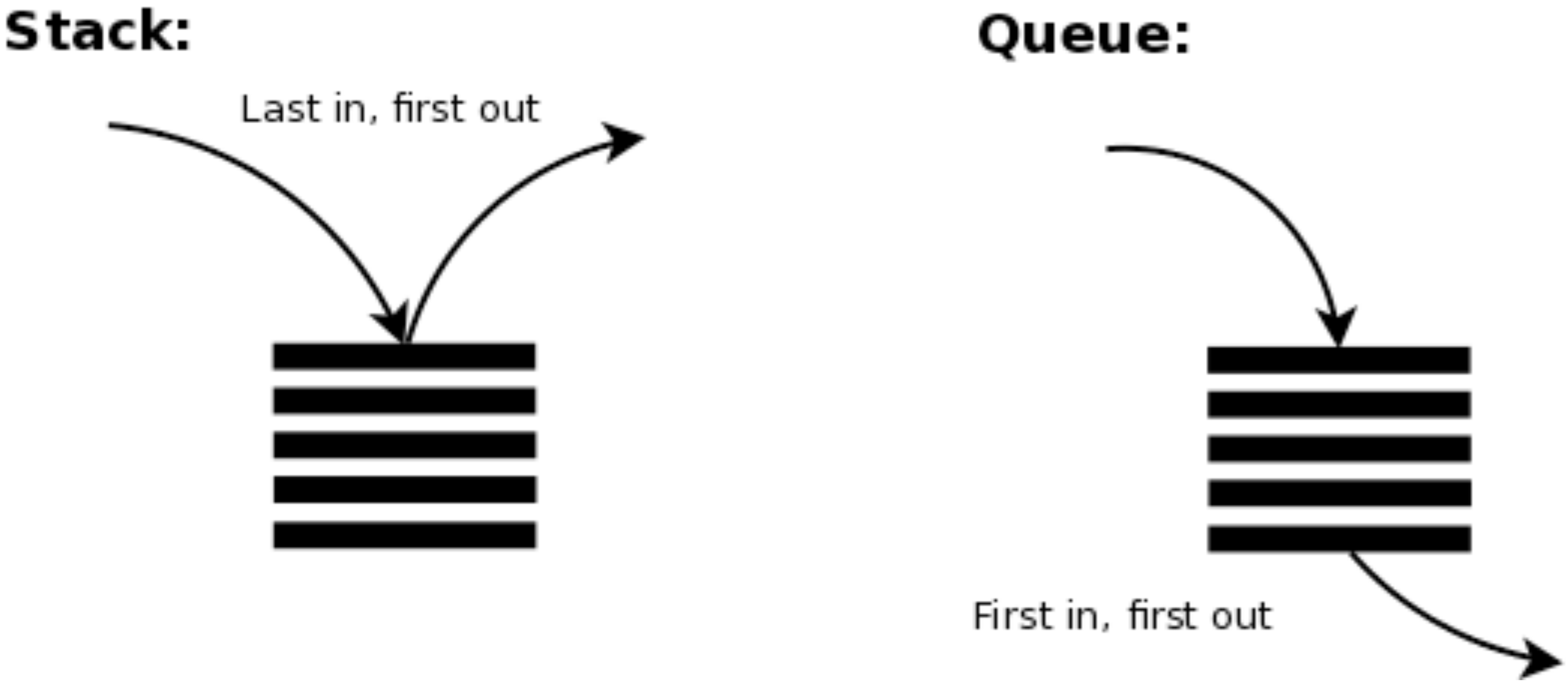
CS62 Class 8: Stacks & Queues

Basic Data Structures



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

Agenda

- Stacks: conceptual; implementation
- Queues: conceptual; implementation
- Algorithmic & affordance analysis
- Using queues in the Java Collections Framework

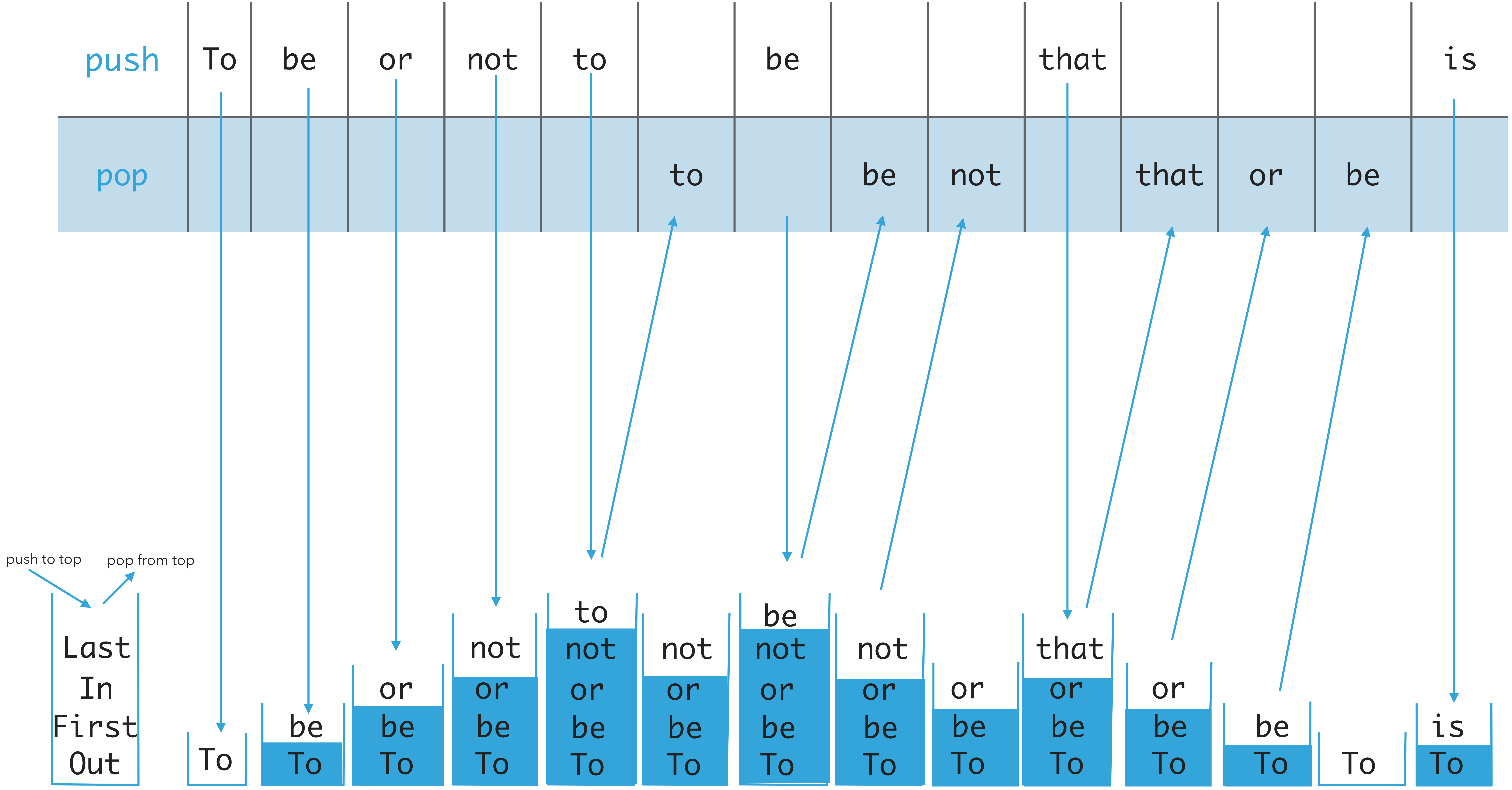
Stacks

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 a stack of plates at the dining hall.
  - 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, call ArrayList add(E element).
  - Adds at the end. Amortized $O^+(1)$.
- To pop an element, call ArrayList remove().
  - Removes and returns the element from the end. Amortized $O^+(1)$.
- To peek, call ArrayList 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.

Stack Operation & Run Time Summary

	ArrayLists: end	SLList: head	DLList: head or tail
push	add O+(1)	addFirst O(1)	addFirst/addLast O(1)
pop	remove O+(1)	removeFirst O(1)	removeFirst/ removeLast O(1)
peek	get(size-1) O(1)	get(0) O(1)	get(0)/get(size-1) O(1)

Due to ArrayLists not needing additional memory overhead of pointer maintenance, they are slightly preferred for stack implementation.

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

- Call stack when running your code.
- Back button in browser.
- Undo in word processor.
- Basic mechanisms in compilers, interpreters (pushdown automata, see CS101).
- Postfix expression evaluation (see HW4 - calculator).
  - If the next thing we read is a number, push it on the stack.
  - If the next thing we read is an operator, remove the top two things from the stack, apply the operator and push the answer back on the stack.
  - 3 + 5 * 7 is 3 5 7 * +

Queues

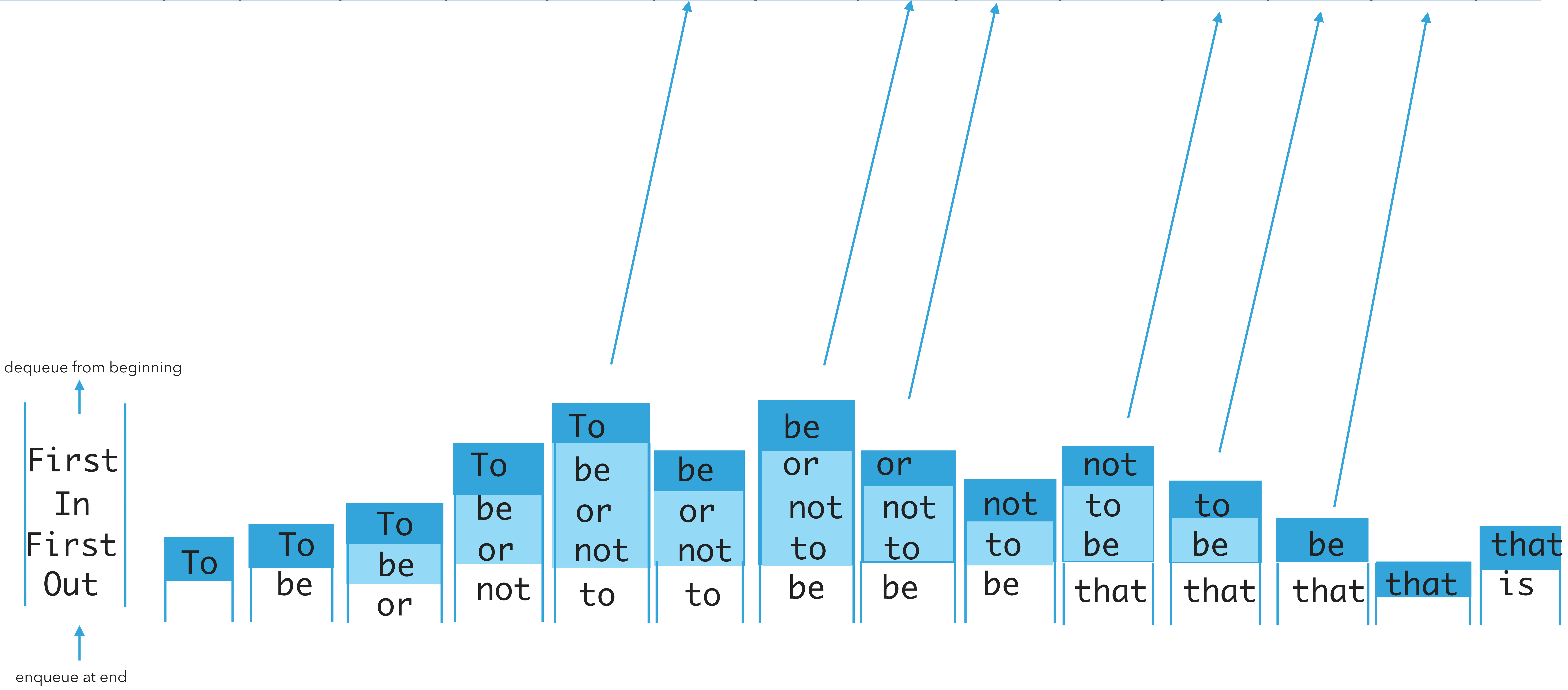
Queues

- 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

enqueue	To	be	or	not	to		be			that				is
dequeue						To		be	or		not	to	be	



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.

Why dequeue from head and not enqueue from head?
Because, as per last lecture, removeEnd() needs more than a tail pointer to set the second to last element's next = null

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 enqueue at the head and dequeue from tail?
    - Still $O(1)$.
  - However, DLLists have a lot of extra pointers. In practice, "jumping" around the memory can increase significantly the running time as well.

Queue Operation & Run Time Summary

	ArrayLists	SLList: head	SLList with tail pointer	DLL
enqueue	add O+(1)	addFirst O(1)	addLast O(1)	addLast O(1)
dequeue	remove(0) O(n)	remove(size-1) O(n)	removeFirst O(1)	removeFirst O(1)

A SLList with a tail pointer is the most efficient implementation of a queue, since you don’t need to maintain unused prev pointers like a DLL.

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

Worksheet answers

- Match the description to the Java code snippet.

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

History of queues

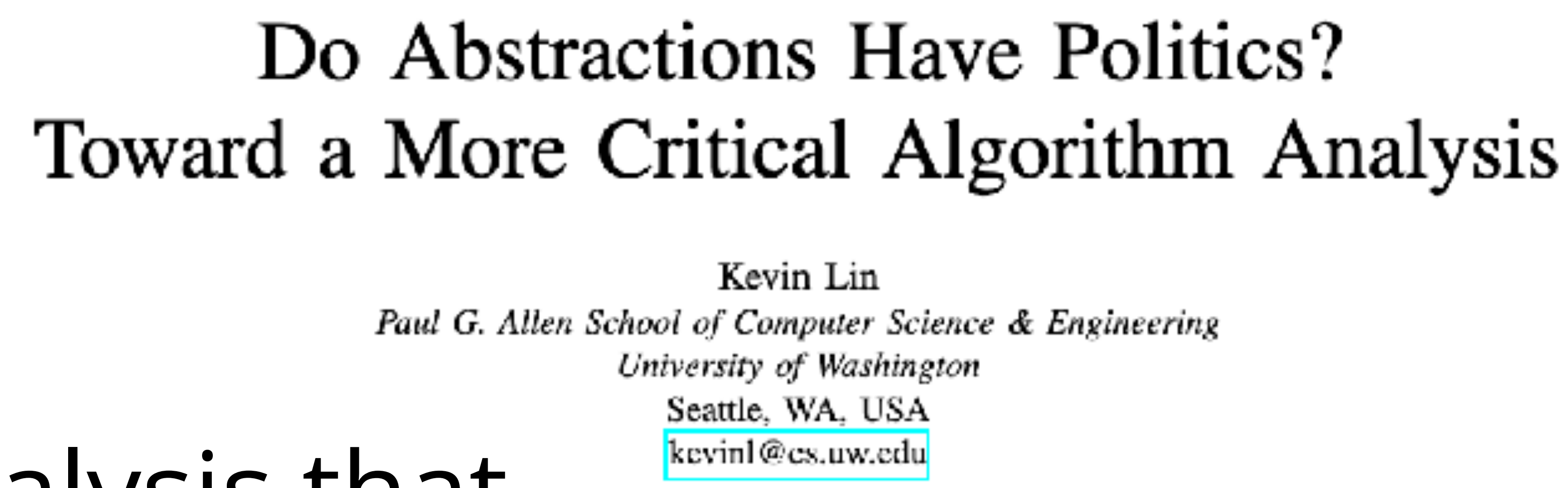
- Queues have existed in math for a long time (queueing theory, 1909 by Danish mathematician Agner Krarup Erlang - studying how to improve telephone call centers, proved that you could apply a Poisson distribution to model the problem)
- World War II made research into queueing theory very popular to 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

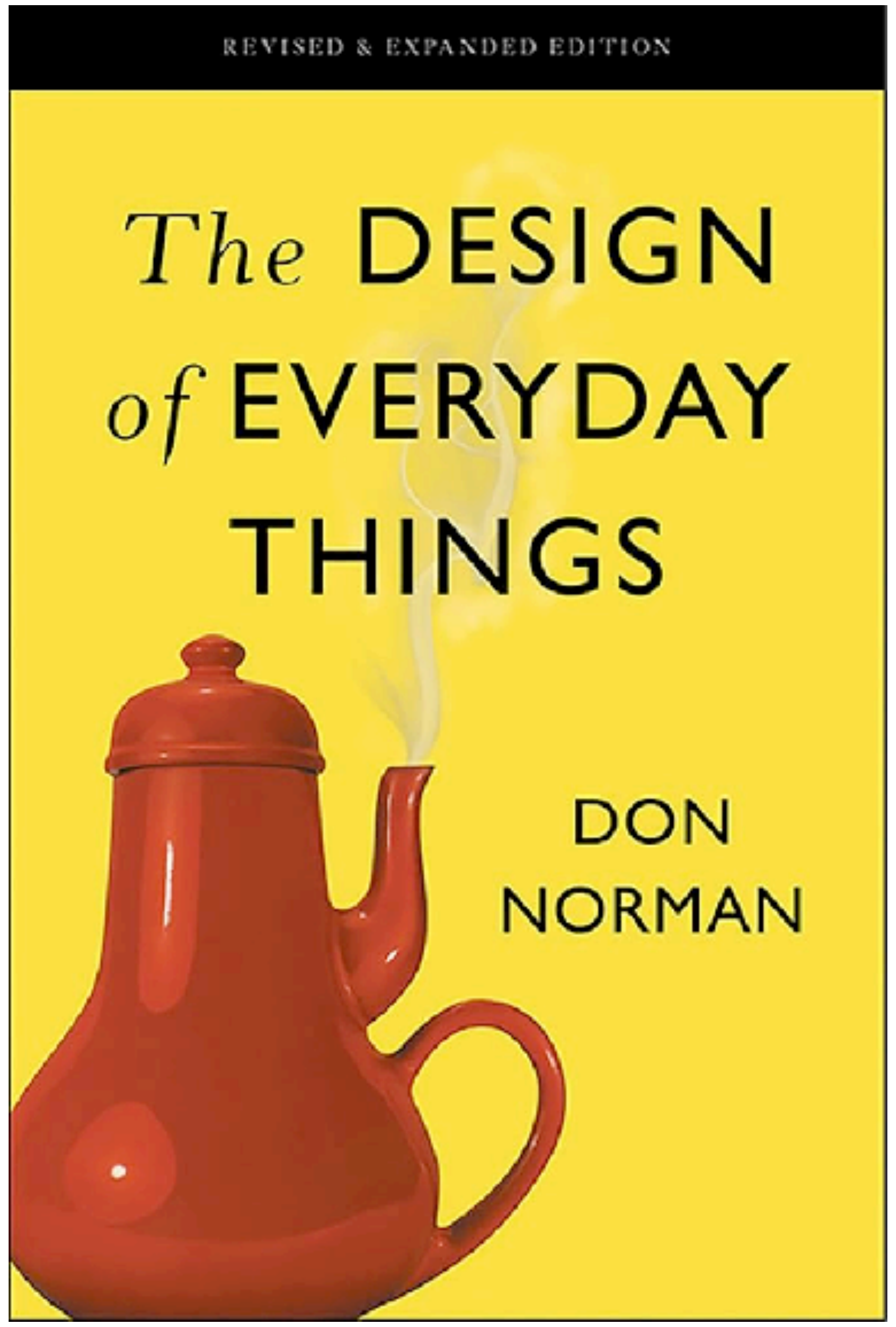
- Music (e.g. Spotify) queue.
- Streaming data buffers (Netflix, Hulu, etc.).
- Asynchronous data transfer (file I/O, sockets).
- Requests in shared resources (printers).
- Traffic analysis.
- Waiting times at calling center.

Affordance analysis


Do Abstractions Have Politics?
Toward a More Critical Algorithm Analysis

Kevin Lin
Paul G. Allen School of Computer Science & Engineering
University of Washington
Seattle, WA, USA
kevinl@cs.uw.edu

- "Affordance analysis is an alternative algorithm analysis that draws on science and technology studies, philosophy of technology, and human-computer interaction to examine how computational abstractions such as data structures and algorithms embody affordances."
- An affordance is a property of an object that make specific outcomes more likely, such as signifying its use; e.x., a doorknob affords turning, a teapot handle affords holding so you don't burn yourself while pouring liquid
- Affordances are human-designed and imbue values
- Who is left behind, and who is prioritized in the design of a data structure/algorithm?



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

https://arxiv.org/pdf/2101.00786

Affordance analysis of queues

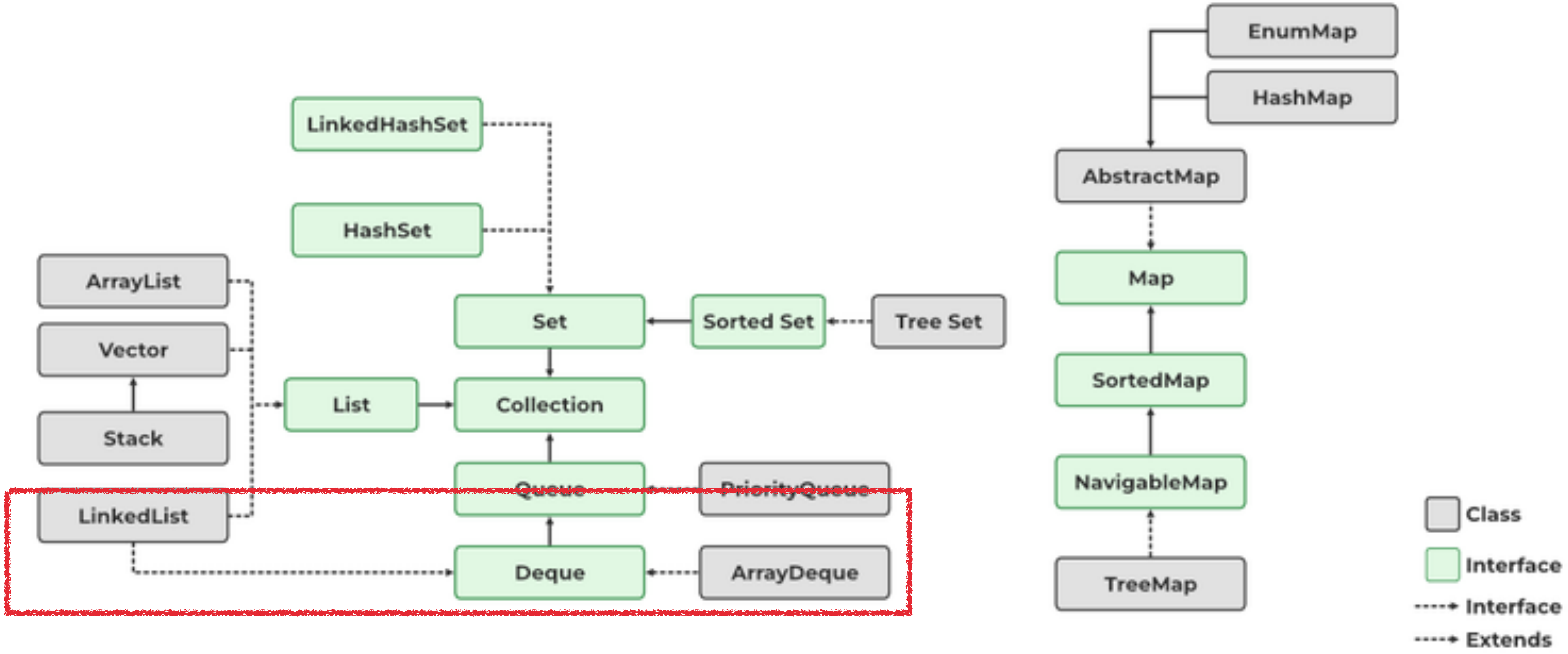
- Discussion Q: Consider some modern applications of queues: hospital emergency rooms, airport security lines, or customer support centers. Should these systems always operate on a FIFO queue, or should some people be given priority? How do we balance fairness with efficiency in such systems? How would 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

Using stacks/queues in JCF

“Deque” interface & ArrayDeque (or LinkedList)



Deque **in Java Collections**

- Do not use `Stack`. Deprecated class. (If you need a stack, it's easy to build your own stack, or just use the code provided with this lecture.)
- The `Queue` class is an interface, not a class you can import.
- It's recommended to use the `Deque` interface over the `Queue` interface.
  - 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 queue = new ArrayDeque(); //preferable`
  - `Deque queue = new LinkedList(); //also works`

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

Lecture 8 wrap-up

- Darwin Part II due Tues 11:59pm
  - Since I'm gone next week, winner of the competition will be announced in class 10/6
- HW4: Calculator released (build a calculator using stacks; individual assignment, no starter code)
- **This is the last lecture that will be on Checkpoint 1!** Checkpoint 1 is next Monday. Lab this week is a review session. Review materials (5 practice problems) are also already up online (Lab 5 slides)
- If you have SDRC accommodations, let them know ASAP (schedule a proctoring time)

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

Worksheet partially lifted from https://www.cs.princeton.edu/courses/archive/fall18/cos126/precepts/p15-stacks-queues/Stacks-and-Queues-Worksheet.pdf

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).