



Middlebury

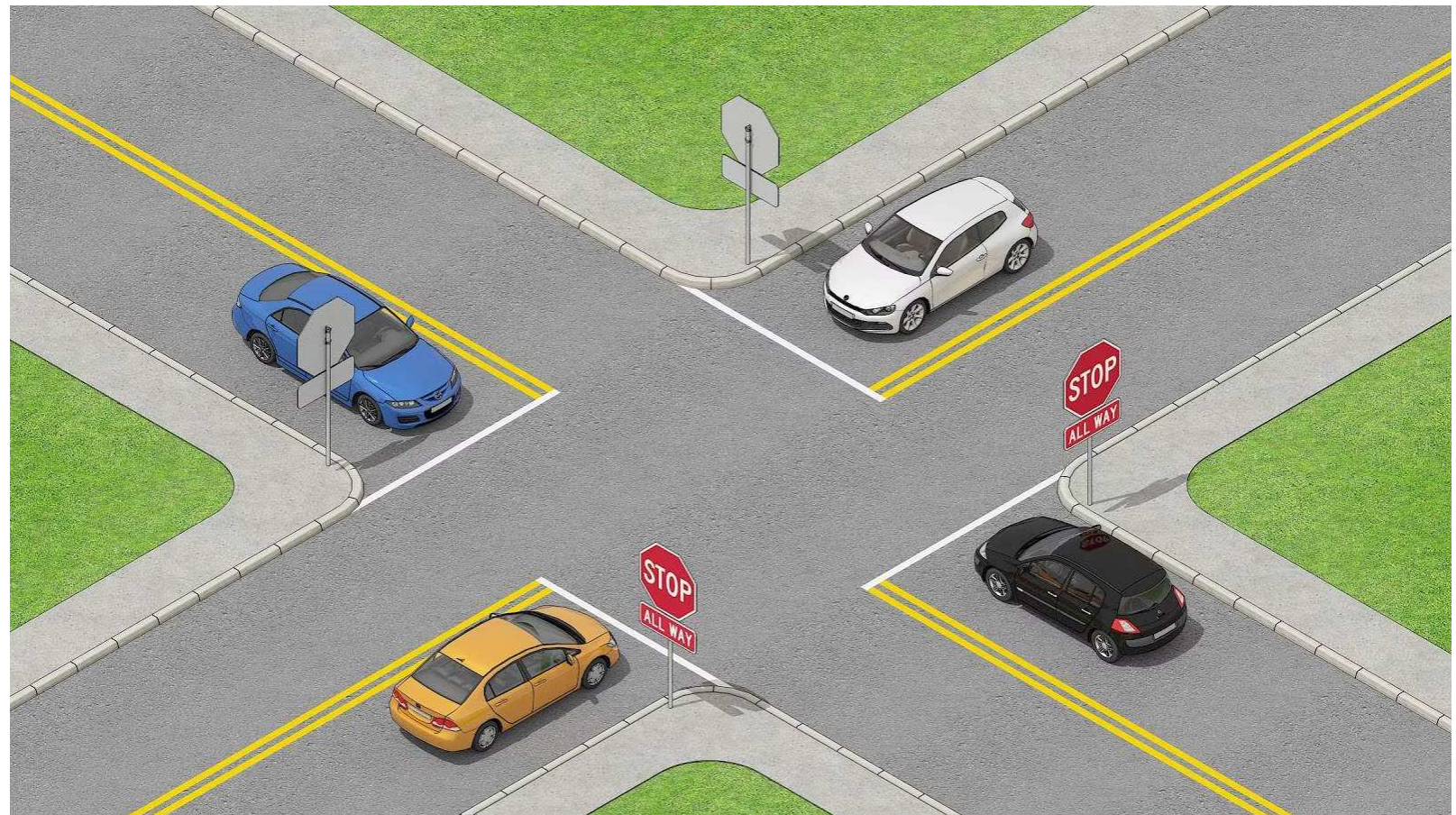
CSCI 201: Data Structures

Fall 2024

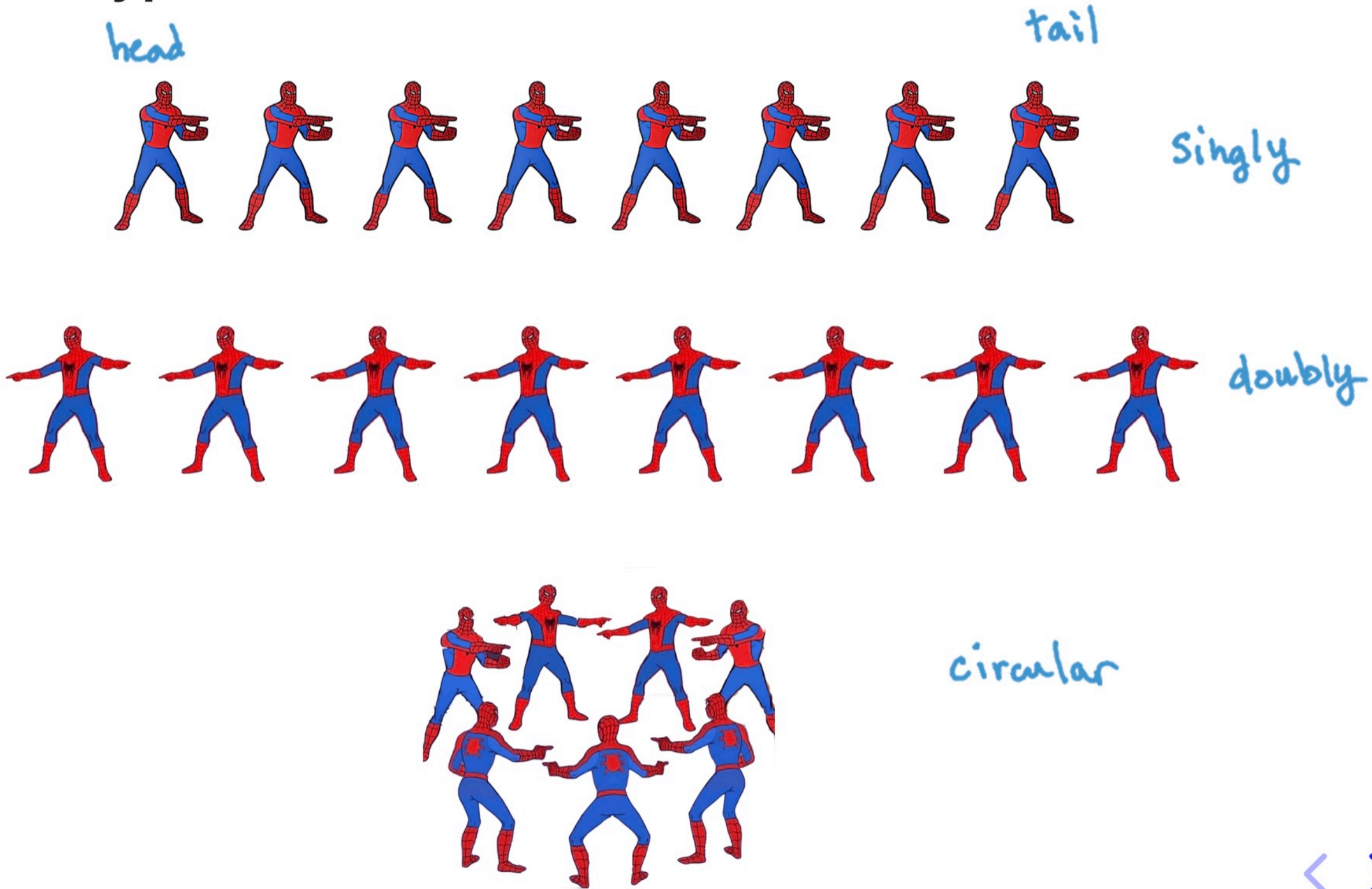
Lecture 7T: Stacks and Queues

Goals for today:

- Analyze the performance of various list operations.
- Implement a **Stack** for LIFO behavior, using either an **ArrayList** or a **LinkedList**.
- Implement a **Queue** for FIFO behavior, using either an **ArrayList** or a **LinkedList**.
- List the advantages/disadvantages of using either an **ArrayList** or **LinkedList** for a **Stack** or **Queue**.



Recall types of linked lists.



How do these **Lists** perform for each method?

	Array List	Singly Linked List (no tail)	Singly Linked List (with tail)	Doubly Linked List
add to end	$O(1)$ amortized	$O(n)$	$O(1)$	$O(1)$
add to front	$O(n)$	$O(1)$	$O(1)$	$O(1)$
contains	$O(n)$	$O(n)$	$O(n)$	$O(n)$
insert at index	$O(n)$	$O(n)$	$O(n)$	$O(n)$
get at index	$O(1)$	$O(n)$	$O(n)$	$O(n)$
set at index	$O(1)$	$O(n)$	$O(n)$	$O(n)$
remove at index	$O(n)$	$O(n)$	$O(n)$	$O(n)$
remove from front	$O(n)$	$O(1)$	$O(1)$	$O(1)$
remove from end	$O(1)$	$O(n)$	$O(n)$	$O(1)$
size	$O(1)$	$O(1)$	$O(1)$	$O(1)$

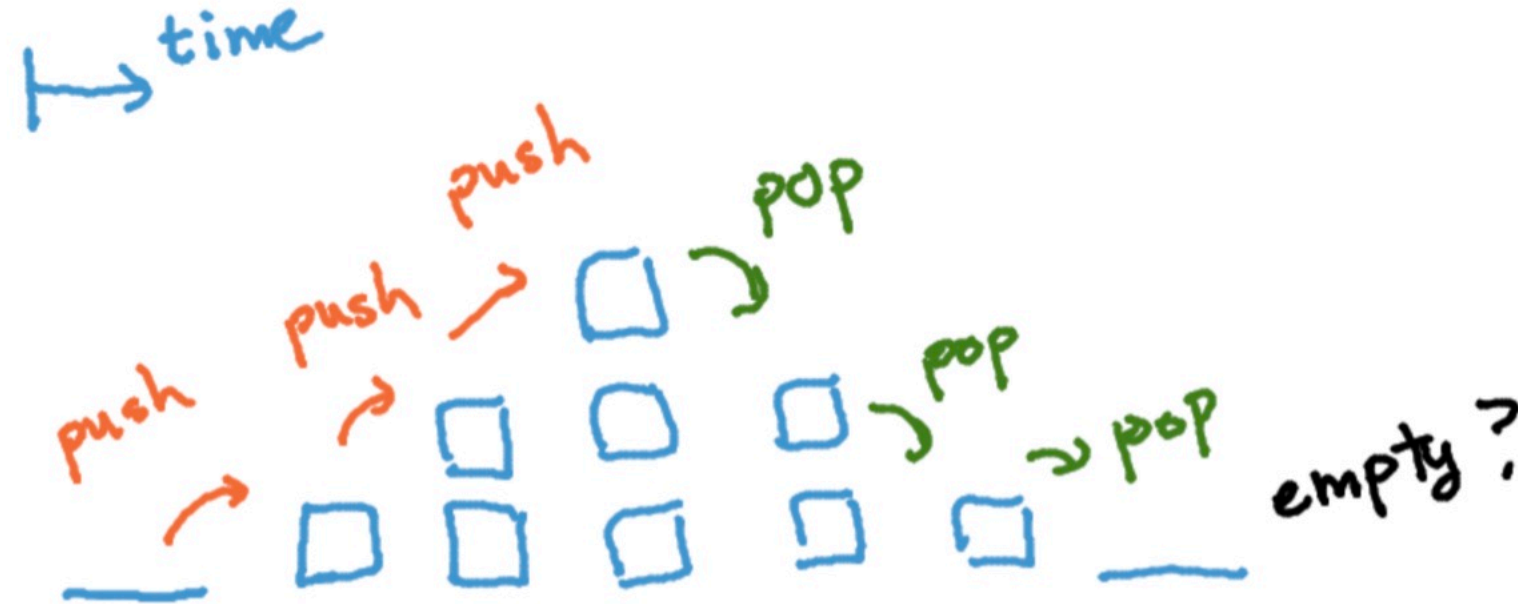
↗ assume with tail

$a = \{ 7, 3, 2, 6, 5 \}$

$a[3] =$

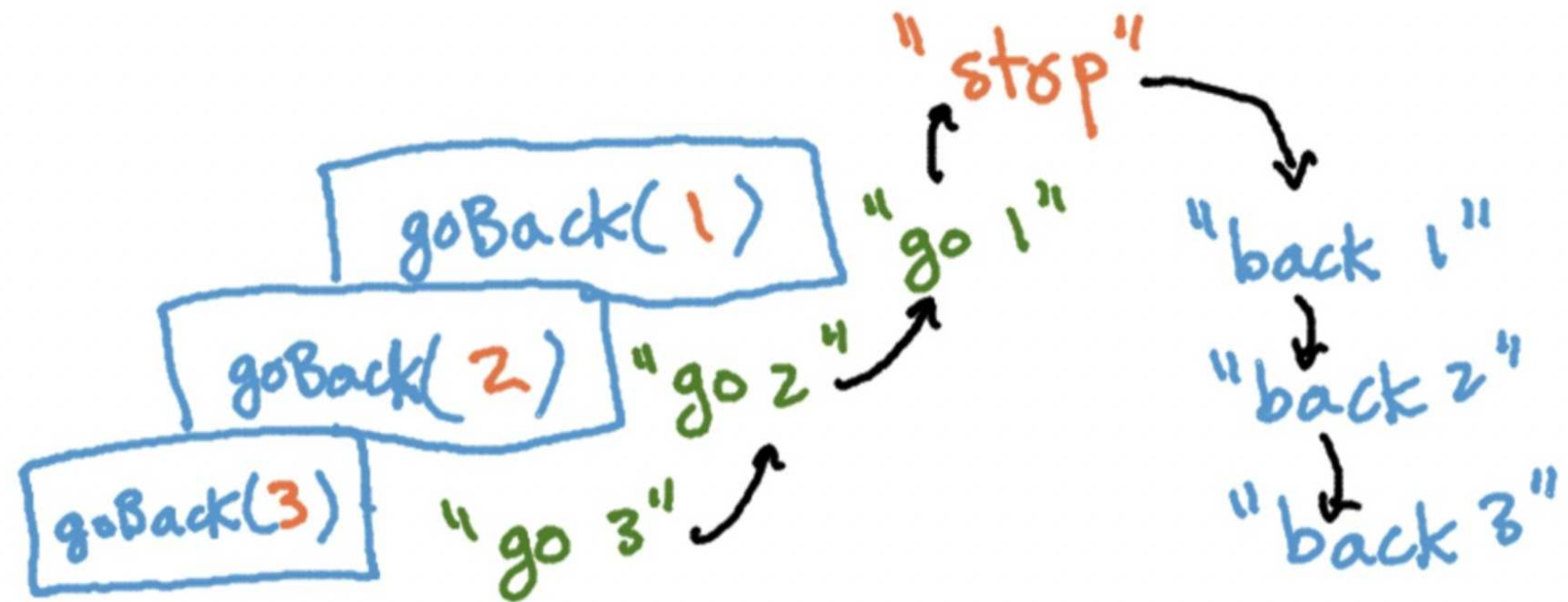


What methods do we need to achieve last-in-first-out (LIFO) behavior?



- **push**: adds an element to the top of the stack,
- **pop**: removes and returns the element at the top of the stack,
- **empty**: determines if there are any elements left in the stack,
- **peek**: returns the element at the top of the stack (without removing).

Recall the **Call Stack**!



```
1 public static void goBack(int n) {  
2     if (n == 0) {  
3         System.out.println("Stop");  
4     } else {  
5         System.out.println("Go " + n);  
6         goBack(n - 1);  
7         System.out.println("Back " + n);  
8     }  
9 }
```

Designing a **Stack** with a **LinkedList**.

Main idea: keep track of **top** (**head**) node, **push** similar to **addFirst**, **pop** similar to **removeFirst**.



```
1 class ListNode<E> {
2     private E data;
3     public ListNode<E> next;
4
5     public ListNode(E data) {
6         this.data = data;
7     }
8
9     public E get() {
10        return data;
11    }
12 }
13
14 public class Stack<E> {
15     private ListNode<E> top;
16
17     public Stack() {
18         top = null;
19     }
20 }
```

```
1 public class Stack<E> {
2     public E push(E data) {
3         ListNode<E> node = new ListNode<E>
4         node.next = top;
5         top = node;
6         return top.get();
7     }
8
9     public E peek() {
10        // assuming top is not null
11        return top.get();
12    }
13
14     public E pop() {
15        // assuming top is not null
16        ListNode<E> node = top;
17        top = top.next;
18        return node.get();
19    }
20
21     public E empty() {
22        return top == null;
23    }
24 }
```

Linked lists are fun, but let's consider the impact of this design choice.

```
1 class ListNode<E> {
2     private E data;
3     public ListNode<E> next;
4
5     public ListNode(E data) {
6         this.data = data;
7     }
8
9     public E get() {
10        return data;
11    }
12 }
13
14 public class Stack<E> {
15     private ListNode<E> top;
16
17     public Stack() {
18         top = null;
19     }
20 }
```

- We only ever need the *last* item (top of the stack).
- We have to store the **data**:
 - **int** uses 32 bits (4 bytes), **float** uses 4 bytes, **double** uses 8 bytes, etc.
- We also need to store a reference to the **next** **ListNode<E>** object.
- References are like addresses of where an object lives: on a 64-bit system, addresses use 64 bits.

Wrappers
Integer
Float,
Double,
etc. ...

Designing a **Stack** with an **ArrayList**.

Main idea: use an **ArrayList** to hold items, **push** to end, and **pop** last item.

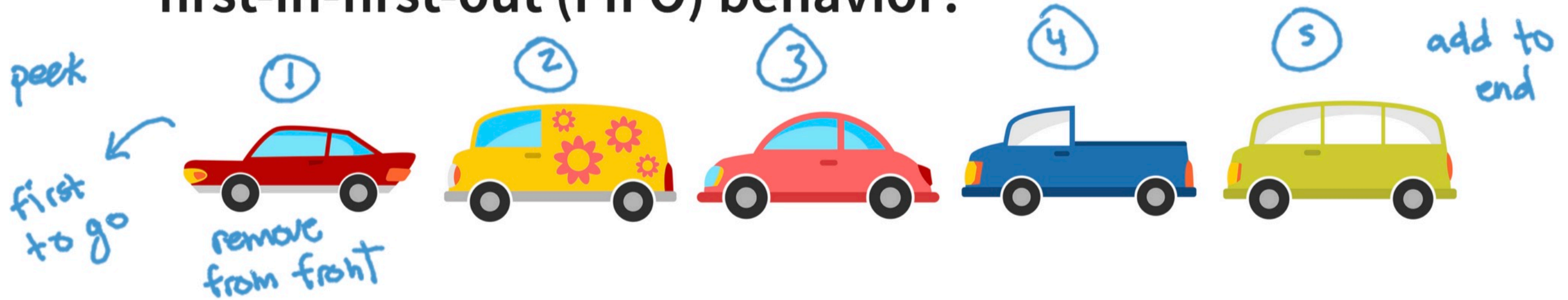
```
1 public class Stack<E> {  
2     private ArrayList<E> items;  
3  
4     public E push(E data) {  
5         return items.add(data);  
6     }  
7  
8     public E peek() {  
9         // assuming there is at least one  
10        return items.get(items.size() - 1);  
11    }  
12  
13    public E pop() {  
14        // assuming there is at least one item  
15        return items.remove(items.size() - 1);  
16    }  
17  
18    public E boolean empty() {  
19        return items.isEmpty();  
20    }  
21 }
```

concerns?



remove
from end
is $O(1)$ 😊

How about designing a data structure to achieve first-in-first-out (FIFO) behavior?



Methods we might need:

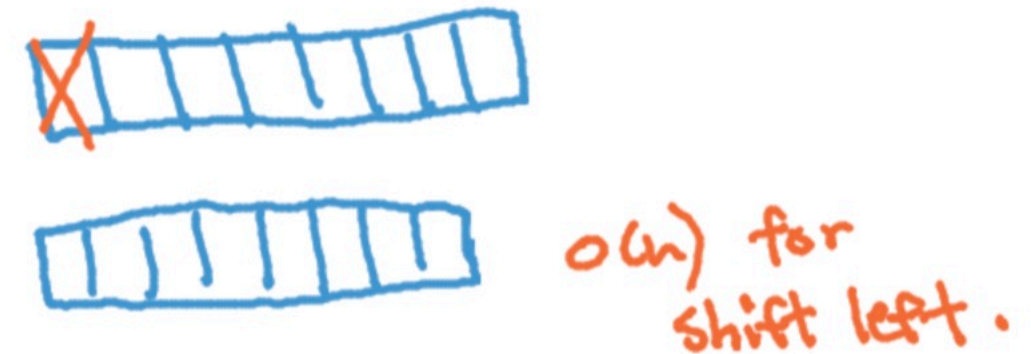
- **add**: adds an element to the back of the queue,
- **remove**: removes and returns the first element in the queue,
- **size**: returns number of elements in the queue,
- **peek**: returns the first element in the queue (without removing).

Designing a **Queue** with an **ArrayList**: Attempt #1.

Main idea: use an **ArrayList** to hold items, **add** to the end, **remove** first item.

```
1 public class Queue<E> {  
2     private ArrayList<E> items;  
3  
4     public E add(E data) {  
5         return items.add(data);  
6     }  
7  
8     public E peek() {  
9         // assuming there is at least one item  
10        return items.get(0);  
11    }  
12  
13    public E remove() {  
14        // assuming there is at least one item  
15        return items.remove(0);  
16    }  
17  
18    public int size() {  
19        return items.size();  
20    }  
21 }
```

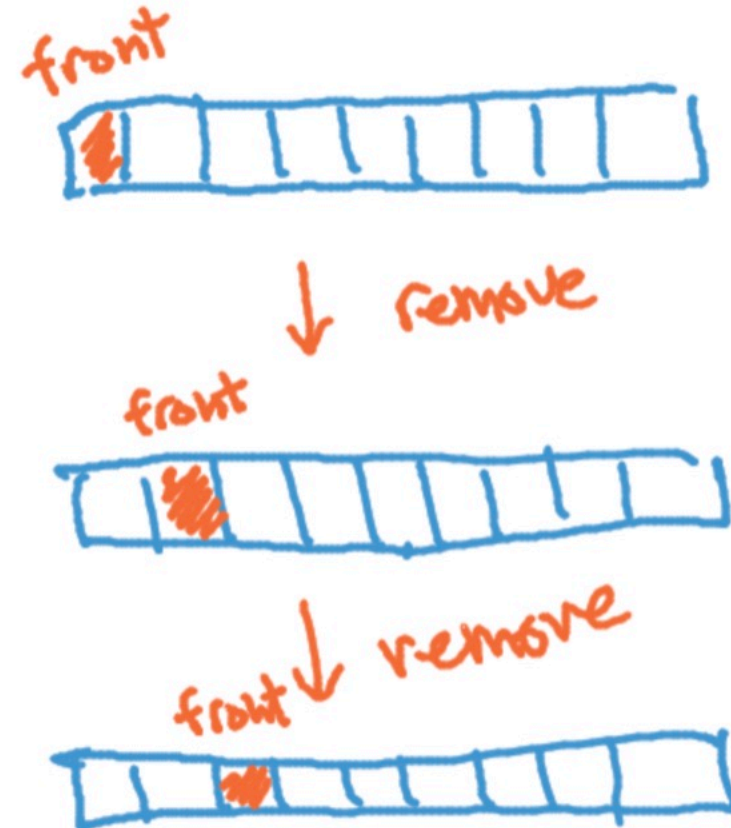
Concerns?



Designing a **Queue** with an **ArrayList**: Attempt #2.

Main idea: use an **ArrayList** to hold items, **add** to the end, use **front** index to remember which item should be **removed** next.

```
1 public class Queue<E> {  
2     private ArrayList<E> items;  
3     int front; // initialize to zero  
4  
5     public E add(E data) {  
6         return items.add(data);  
7     }  
8  
9     public E peek() {  
10        // assuming there is at least one item  
11        return items.get(front);  
12    }  
13  
14    public E remove() {  
15        // assuming there is at least one item  
16        return items.get(front++);  
17    }  
18  
19    public int size() {  
20        // TODO  
21    }  
22 }
```



Designing a **Queue** with an **ArrayList**: Attempt #2.

Main idea: use an **ArrayList** to hold items, **add** to the end, use **front** index to remember which item should be **removed** next.

```
1 public class Queue<E> {
2     private ArrayList<E> items;
3     int front; // initialize to zero
4
5     public E add(E data) {
6         return items.add(data);
7     }
8
9     public E peek() {
10        // assuming there is at least one item
11        return items.get(front);
12    }
13
14    public E remove() {
15        // assuming there is at least one item
16        return items.get(front++);
17    }
18
19    public int size() {
20        // TODO
21    }
22 }
```



cs201-lecture07T



What should the size method return?
(slido.com #4307757)

15

items.size() - front

80%

items.size()

13%

front

7%

items.capacity()

0%

items.capacity() - front

0%



Designing a Queue with a LinkedList.

Main idea: keep track of **head** to remember which element should be **removed** first and the **tail** node to **add** to the end.

```
1 class ListNode<E> {
2     private E data;
3     public ListNode<E> next;
4
5     public ListNode(E data) {
6         this.data = data;
7     }
8
9     public E get() {
10        return data;
11    }
12 }
13
14 public class Queue<E> {
15     private ListNode<E> head;
16     private ListNode<E> tail;
17     private int size;
18
19     public Queue() {
20         head = null;
21         tail = null;
22         size = 0;
23     }
24 }
```

```
1 public class Queue<E> {
2     public E add(E data) {
3         ListNode<E> node = new ListNode<E>
4         if (size == 0) {
5             head = node;
6         } else {
7             tail.next = node;
8         }
9         size++;
10        tail = node;
11        return node.get();
12    }
13
14    public E peek() {
15        // assuming head is not null
16        return head.get();
17    }
18
19    public E remove() {
20        // assuming head is not null
21        E data = head.get();
22        head = head.next;
23        size--;
24        return data;
25    }
26
27    public int size() {
28        return size;
29    }
30 }
```

Utilities built into **Java** (**import.java.util.***).

- **Stack**: array-based implementation of a stack.
- **Queue**: interface for queue operations, implemented by
 - **LinkedList**: doubly-linked list,
 - **ArrayDeque**: array-based Double-Ended-Queue.

*Stack<Integer> stack
= new Stack<>();*

Class ArrayDeque<E>

java.lang.Object
java.util.AbstractCollection<E>
java.util.ArrayDeque<E>

Type Parameters:

E - the type of elements held in this collection

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, Deque<E>, Queue<E>

```
public class ArrayDeque<E>  
extends AbstractCollection<E>  
implements Deque<E>, Cloneable, Serializable
```

Resizable-array implementation of the `Deque` interface. Array deques have no capacity restrictions; they grow as necessary to support usage. They are not thread-safe; in the absence of external synchronization, they do not support concurrent access by multiple threads. Null elements are prohibited. This class is likely to be faster than `Stack` when used as a stack, and faster than `LinkedList` when used as a queue.

Most `ArrayDeque` operations run in amortized constant time. Exceptions include `remove`, `removeFirstOccurrence`, `removeLastOccurrence`, `contains`, `iterator.remove()`, and the bulk operations, all of which run in linear time.

Exercise: LeetCode #20

20. Valid Parentheses

Easy

Topics

Companies

Hint

Given a string `s` containing just the characters `'('`, `')'`, `'{'`, `'}'`, `'['` and `']'`, determine if the input string is valid.

An input string is valid if:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.
3. Every close bracket has a corresponding open bracket of the same type.

Example 1:

Input: `s = "()"`

Output: `true`

Example 2:

Input: `s = "() [] {}"`

Output: `true`

Example 3:

Input: `s = "()["`

Output: `false`

Example 4:

Input: `s = "( [])"`

Output: `true`

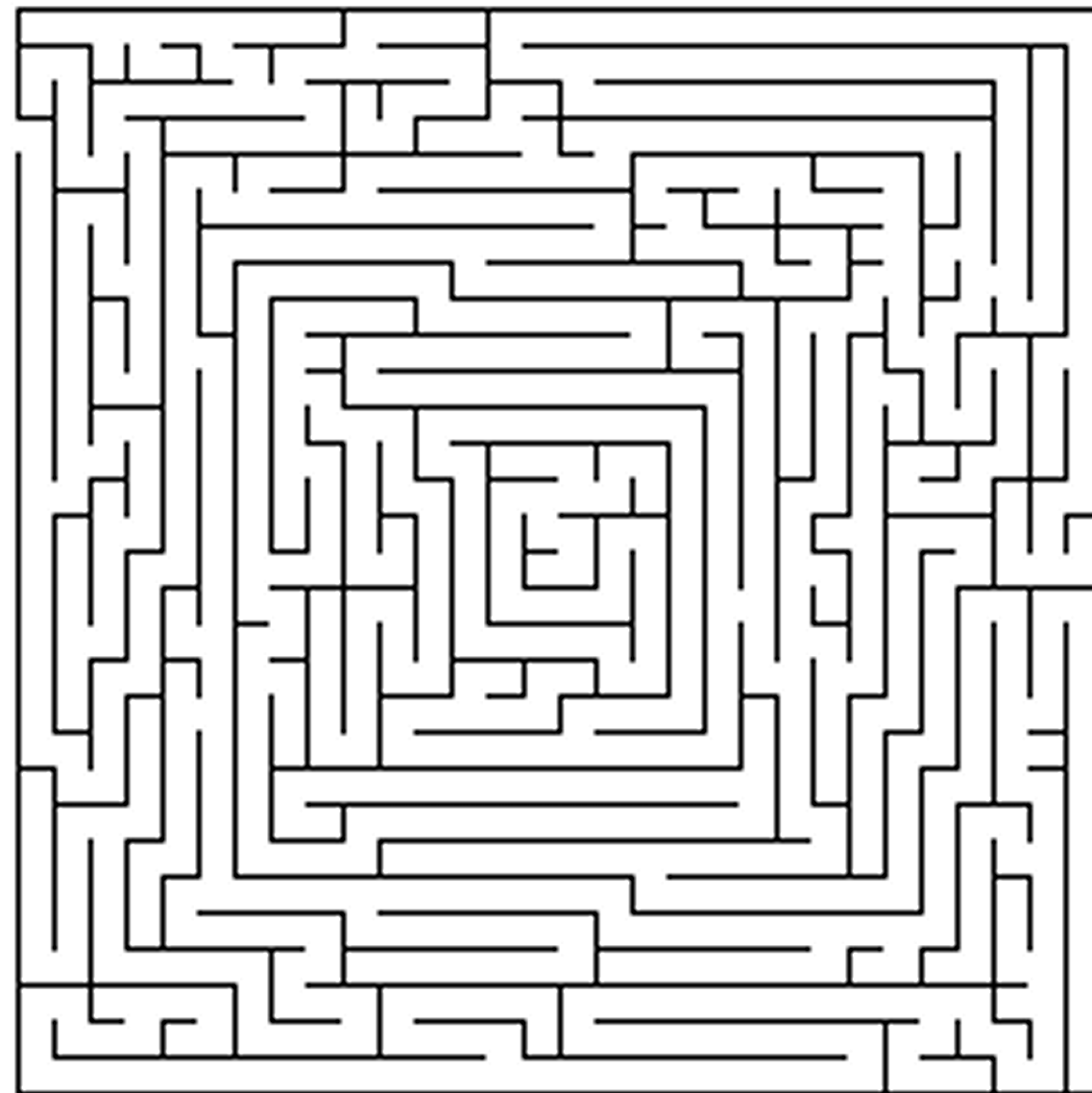
Possible solutions using **ArrayList**, **ArrayDeque**.

```
1 public boolean isValid(String s) {
2     ArrayList<Character> stack = new ArrayList<>();
3     for (int i = 0; i < s.length(); i++) {
4         char c = s.charAt(i);
5         if (c == '(' || c == '[' || c == '{') {
6             stack.add(c);
7         } else {
8             if (stack.isEmpty())
9                 return false;
10            char p = stack.remove(stack.size() - 1);
11            if (p == '(' && c != ')')
12                return false;
13            if (p == '[' && c != ']')
14                return false;
15            if (p == '{' && c != '}')
16                return false;
17        }
18    }
19    return stack.isEmpty();
20 }
```

```
1 public boolean isValid(String s) {
2     ArrayDeque<Character> stack = new ArrayDeque<>();
3     for (int i = 0; i < s.length(); i++) {
4         char c = s.charAt(i);
5         if (c == '(' || c == '[' || c == '{') {
6             stack.push(c);
7         } else {
8             if (stack.isEmpty())
9                 return false;
10            char p = stack.pop();
11            if (p == '(' && c != ')')
12                return false;
13            if (p == '[' && c != ']')
14                return false;
15            if (p == '{' && c != '}')
16                return false;
17        }
18    }
19    return stack.isEmpty();
20 }
```

Here's another problem to think about!

Which data structures would you use to keep track of your path through a maze?
If you hit a wall, what should you do?



More when we talk about **graphs** in a few weeks!

Good luck on the Midterm!

- No class on Thursday 10/24 or lab on Friday 10/25:
 - I will be in my office during class/lab time in case you have questions about the midterm.
 - I can only answer *clarification* questions (no debugging help).
- Lab 5 due tonight (implement the **MiddDocs** text editor).
- See Midterm Study Guide on Canvas.
- Midterm has 2 parts, both are due at 11:59pm on Friday 10/25 (no time limit on either part).
- **Reminder that generative AI is not allowed in any way on the midterm.**
- No exit ticket today.