Middlebury

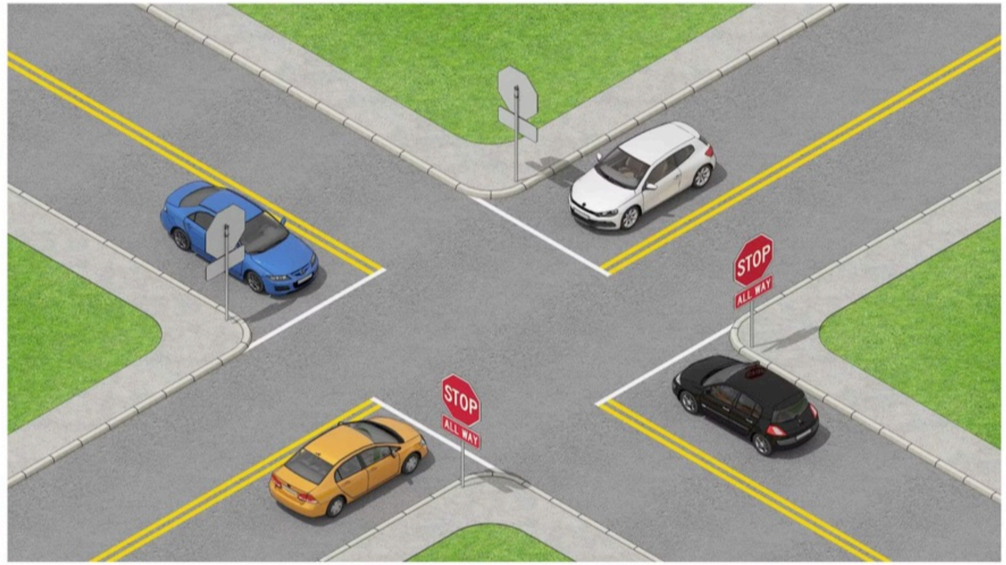
CSCI 201: Data Structures

Spring 2025

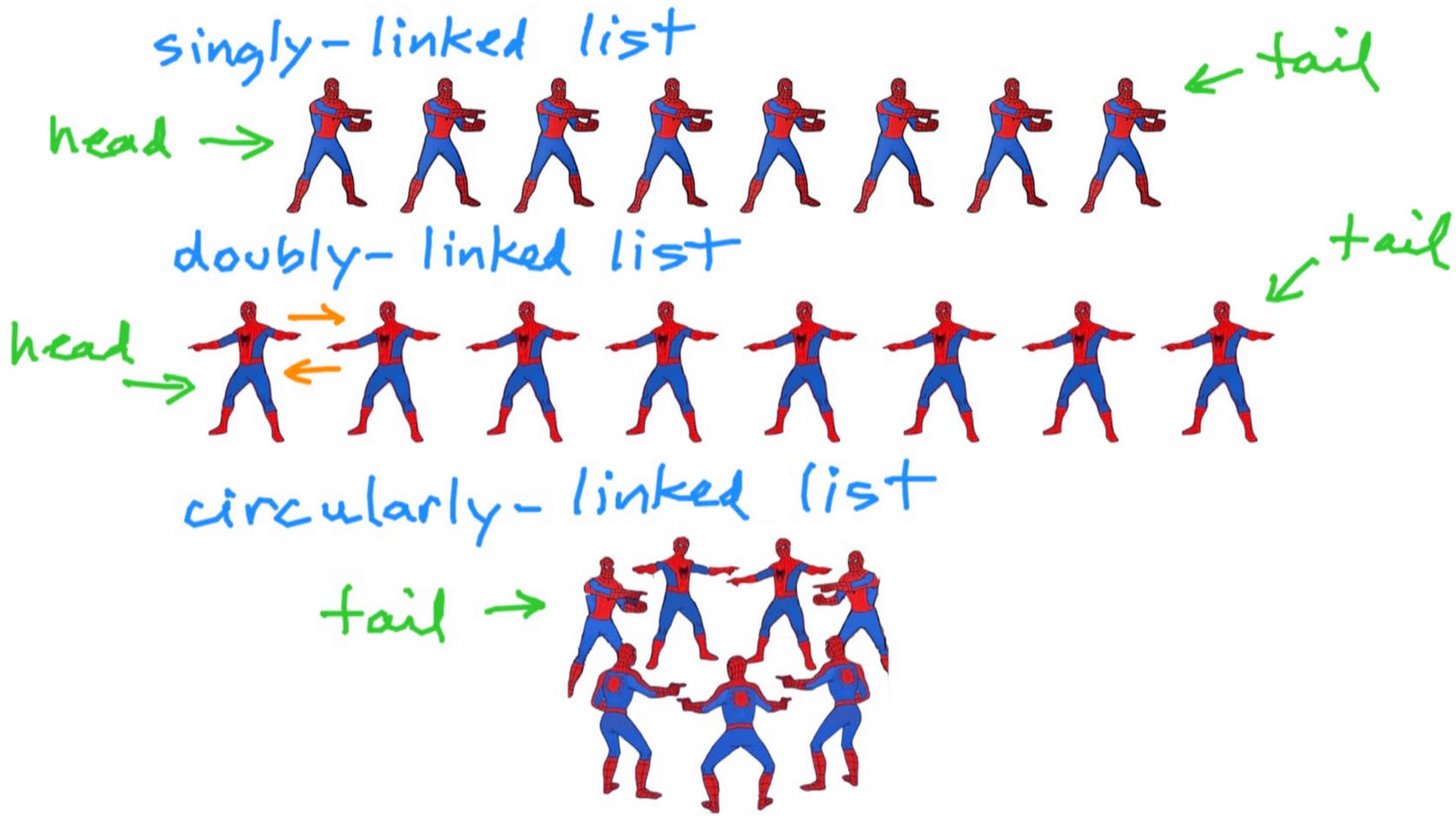
Lecture 7M: Stacks and Queues

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.



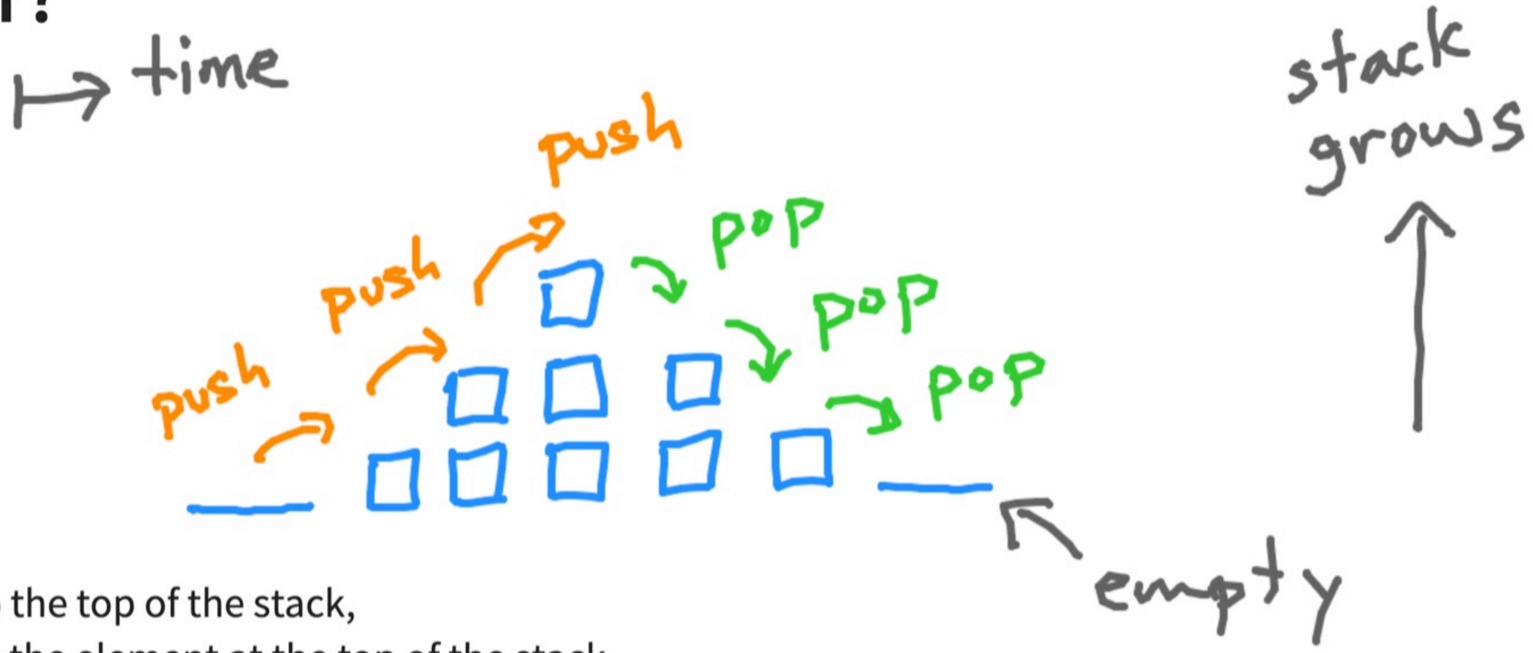
How do these `List`s perform for each method?

list of size n

assume with tail

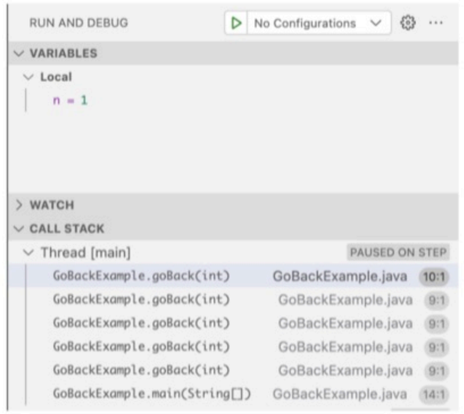
	Array List	Singly Linked List (no tail)	Singly Linked List (with tail)	Doubly Linked List
`add` to end	O(1) amortized	O(n)	O(1)	O(1)
`add` to front	O(n)	O(1)	O(1)	O(1)
`contains`	O(n)	O(n)	O(n)	O(n)
`insert` at index	O(n)	O(n)	O(n)	O(n)
`get` at index	O(1)	O(n)	O(n)	O(n)
`set` at index	O(1)	O(n)	O(n)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

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

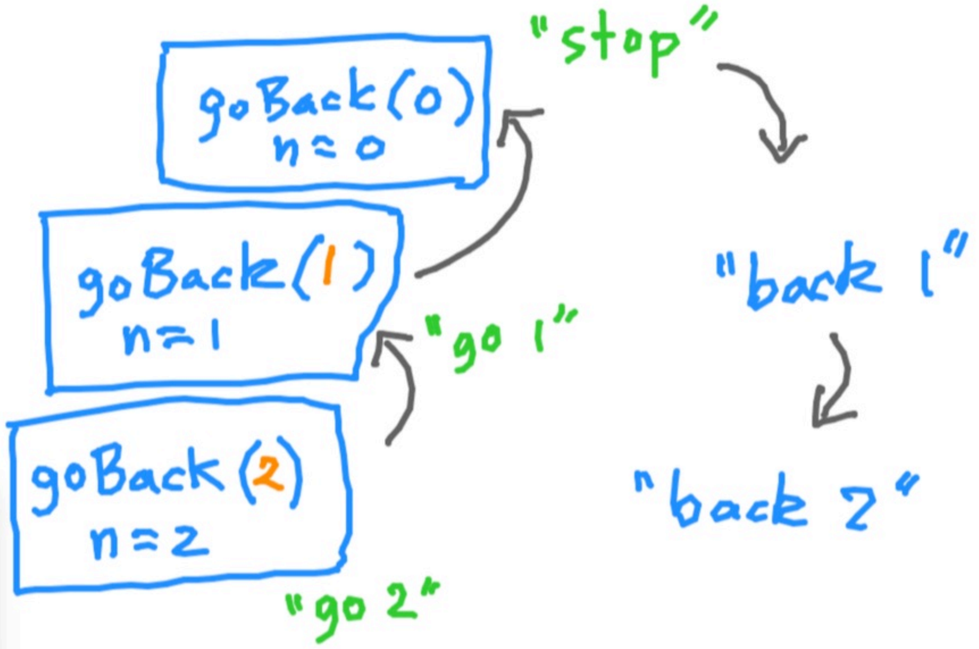


- `push`: adds an element to the top of the stack,
- `pop`: removes and returns the element at the top of the stack,
- `empty`: determines if there are any elements left in the stack,
- `peek`: returns the element at the top of the stack (without removing).

Recall the Call Stack!

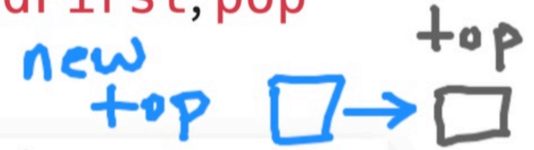



```java
public static void goBack(int n) {
  if (n == 0) {
    System.out.println("Stop");
  } else {
    System.out.println("Go " + n);
    goBack(n - 1);
    System.out.println("Back " + n);
  }
}
```



Designing a Stack with a LinkedList.

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




```java
class ListNode<E> {
  private E data;
  public ListNode<E> next;

  public ListNode(E data) {
    this.data = data;
  }

  public E get() {
    return data;
  }
}

public class Stack<E> {
  private ListNode<E> top;

  public Stack() {
    top = null;
  }
}
```

```java
public class Stack<E> {
  public E push(E data) {
    ListNode<E> node = new ListNode<E>
    node.next = top;
    top = node;
    return top.get();
  }

  public E peek() {
    // assuming top is not null
    return top.get();
  }

  public E pop() {
    // assuming top is not null
    ListNode<E> node = top;
    top = top.next;
    return node.get();
  }

  public boolean empty() {
    return top == null;
  }
}
```

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 }
```

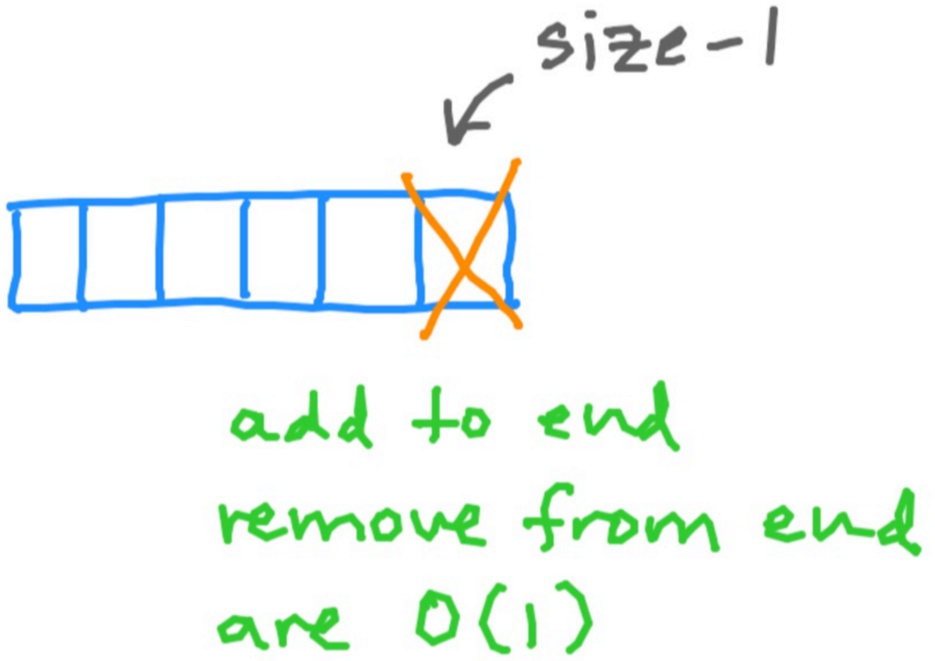
- We only ever need the *last* item (top of the stack).
- We have to store the `data`:
  - `int` uses 32 bits (4 bytes), `float` uses 4 bytes, `double` uses 8 bytes, etc.
- We also need to store a reference to the `next` `ListNode<E>` object.
- References are like addresses of where an object lives: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

wrapper classes Integer, Float, Double, etc.

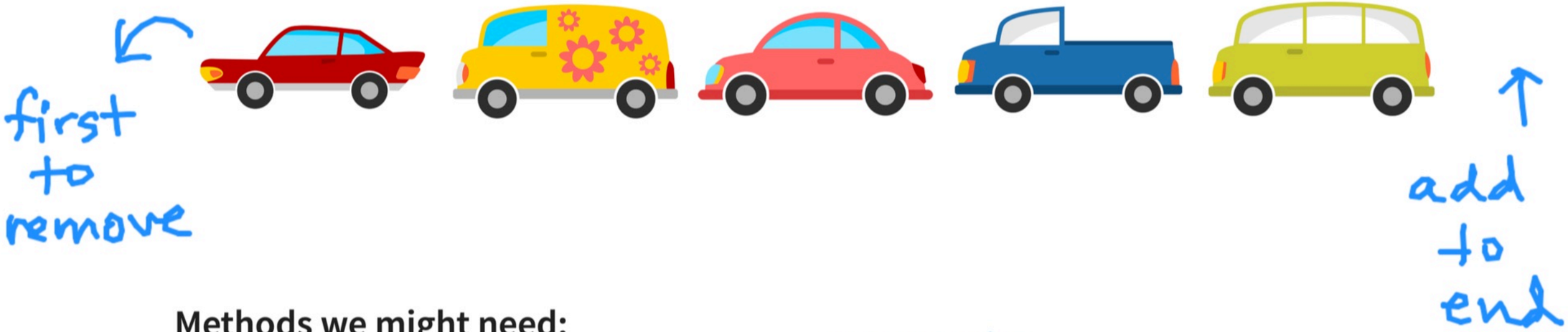
Designing a Stack with an ArrayList.

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

```java
public class Stack<E> {
  private ArrayList<E> items;

  public E push(E data) {
    return items.add(data);
  }

  public E peek() {
    // assuming there is at least one item
    return items.get(items.size() - 1);
  }

  public E pop() {
    // assuming there is at least one item
    return items.remove(items.size() - 1);
  }

  public boolean empty() {
    return items.isEmpty();
  }
}
```



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).

enqueue

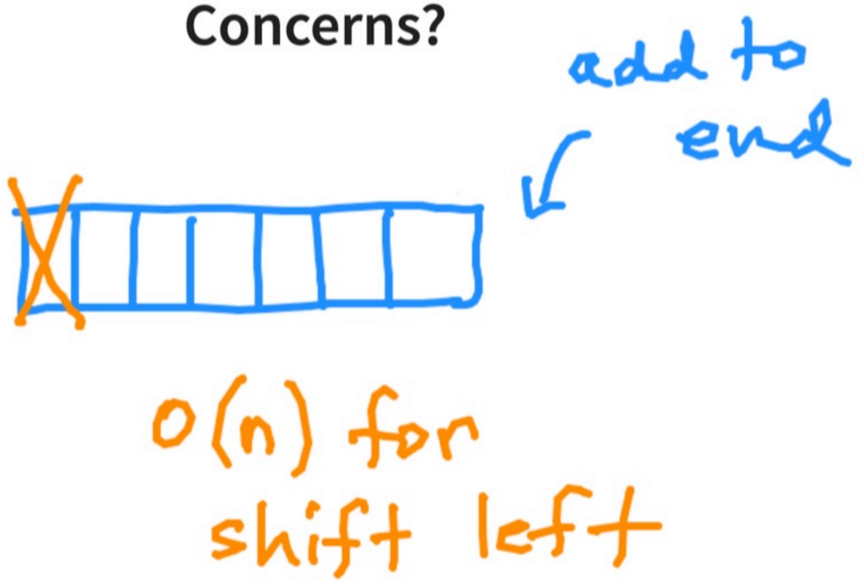
dequeue

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

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

```java
public class Queue<E> {
  private ArrayList<E> items;

  public E add(E data) {
    return items.add(data);
  }

  public E peek() {
    // assuming there is at least one item
    return items.get(0);
  }

  public E remove() {
    // assuming there is at least one item
    return items.remove(0);
  }

  public int size() {
    return items.size();
  }
}
```

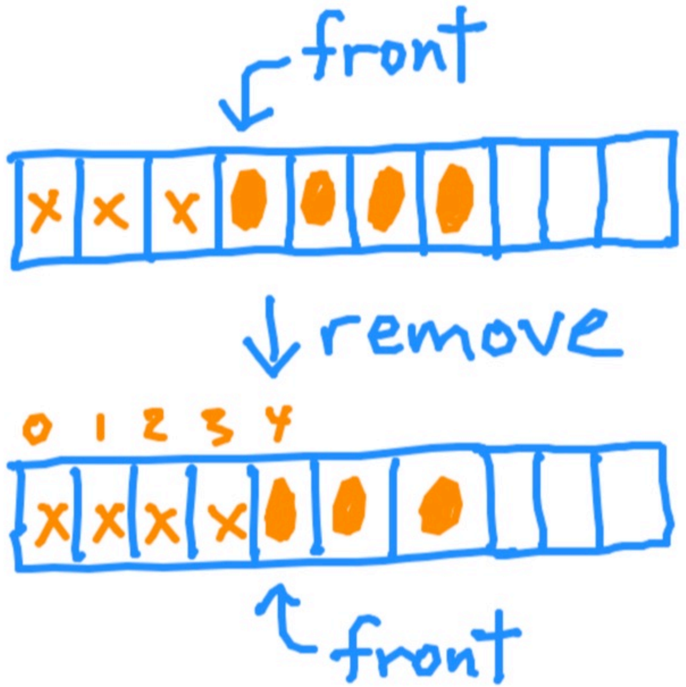
Concerns?



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 `remove`d next. See slido # 1507784.

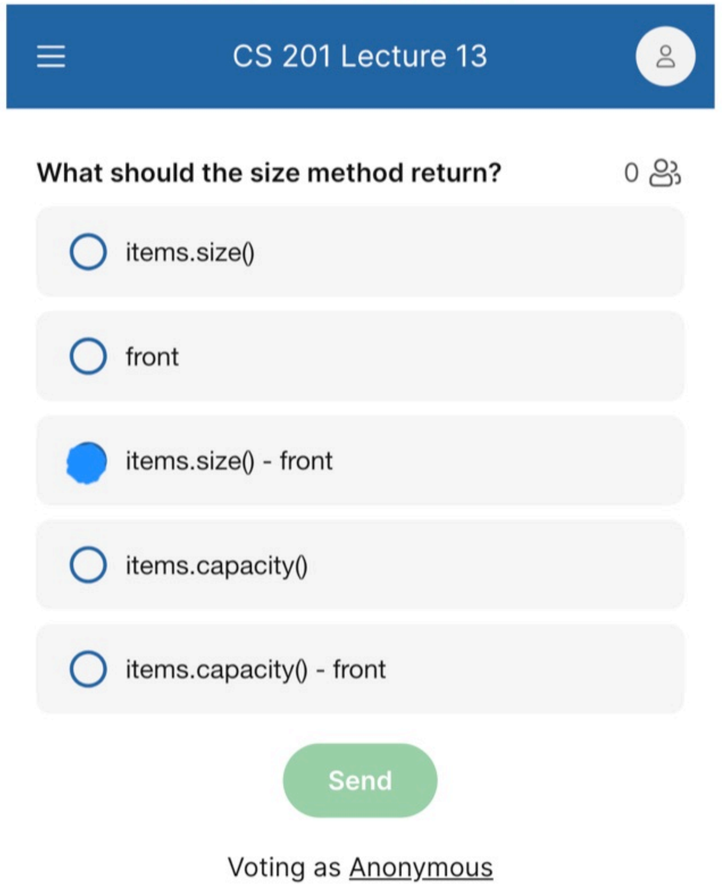
```
1  public class Queue<E> {
2    private ArrayList<E> items;
3    int front; // initialize to zero
4
5    public E add(E data) {
6      return items.add(data);
7    }
8
9    public E peek() {
10     // assuming there is at least one item
11     return items.get(front);
12   }
13
14   public E remove() {
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 `remove`d next. See slido # 1507784.

```java
public class Queue<E> {
  private ArrayList<E> items;
  int front; // initialize to zero

  public E add(E data) {
    return items.add(data);
  }

  public E peek() {
    // assuming there is at least one item
    return items.get(front);
  }

  public E remove() {
    // assuming there is at least one item
    return items.get(front++);
  }

  public int size() {
    // TODO
  }
}
```



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`

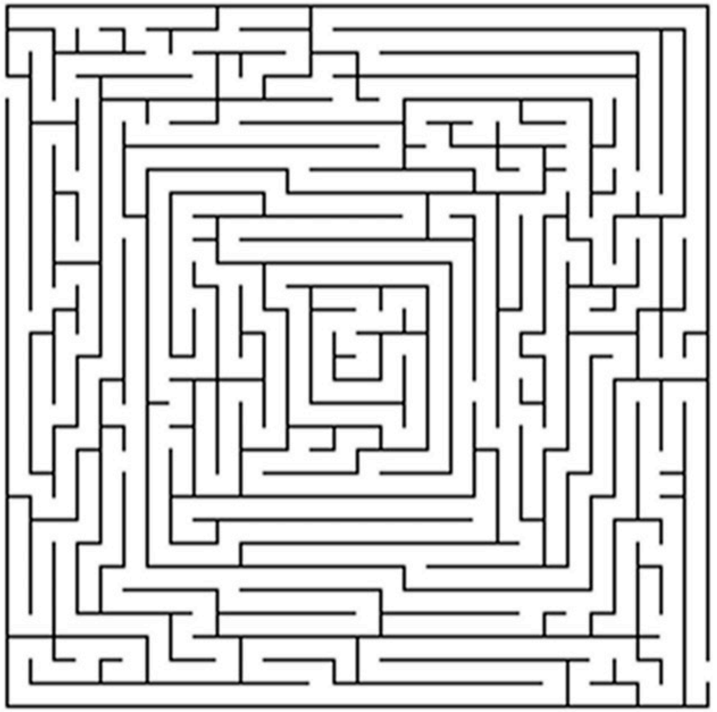
Possible solutions using `ArrayList`, `ArrayDeque`.

```java
public boolean isValid(String s) {
  ArrayList<Character> stack = new ArrayList<>();
  for (int i = 0; i < s.length(); i++) {
      char c = s.charAt(i);
      if (c == '(' || c == '[' || c == '{') {
          stack.add(c);
      } else {
          if (stack.isEmpty())
              return false;
          char p = stack.remove(stack.size() - 1);
          if (p == '(' && c != ')')
              return false;
          if (p == '[' && c != ']')
              return false;
          if (p == '{' && c != '}')
              return false;
      }
  }
  return stack.isEmpty();
}
```

```java
public boolean isValid(String s) {
  ArrayDeque<Character> stack = new ArrayDeque<>();
  for (int i = 0; i < s.length(); i++) {
    char c = s.charAt(i);
    if (c == '(' || c == '[' || c == '{') {
      stack.push(c);
    } else {
      if (stack.isEmpty())
          return false;
      char p = stack.pop();
      if (p == '(' && c != ')')
          return false;
      if (p == '[' && c != ']')
          return false;
      if (p == '{' && c != '}')
          return false;
    }
  }
  return stack.isEmpty();
}
```

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!

This week

- **Lab 5** due tonight (implement the `MiddDocs` text editor).
- **In-class Midterm** on Wednesday. Please bring a laptop.
- See **Midterm Study Guide** on Canvas.
- Programming portion posted today at 2:00 pm, due Thursday 4/3 by 11:59 pm.
- **Reminder that generative AI is not allowed in any way on the midterm.**
- No help sessions after tonight (help sessions cancelled 4/1 through 4/7).