LEC 06

CSE 122

Stacks & Queues

Questions during Class?

Raise hand or send here

sli.do #cse122

BEFORE WE START

Talk to your neighbors:

Favorite summer activity?

Instructor: Ido Avnon

TAs: Abby Williams
Chloë Mi Cartier
Connor Sun
Cynthia Pan
Katharine Zhang
Marcus Sanches
Rohini Arangam

Lecture Outline

- Announcements
- Review: ADTs, Stacks & Queues
- Queue Manipulation
- Stack Manipulation
  - Problem Solving

Announcements

- Quizzes
  - Quiz 0 was yesterday
  - Feedback released sometime before Quiz 1 (July 25)
  - *Metacognition*: How did it go? Was your studying and preparation effective?
- Culminating Project Checkpoint 1 is due tomorrow by 11:59pm PT
- Programming Assignment 1 releasing on Friday
  - Focus on Stacks & Queues
  - Due Thursday, July 18th by 11:59pm PT

Lecture Outline

- Announcements
- **Review: Stacks & Queues**
- Queue Manipulation
- Stack Manipulation
  - Problem Solving

(PCM) Abstract Data Types

- **Abstract Data Type (ADT)**: A specification of a collection of data and the operations that can be performed on it.
  - Describes *what* a collection does, not *how* it does it (not implementation!)
  - Think of it as an *idea* of a data type

- We don't know exactly how a stack or queue is implemented, and we don't need to.
  - Only need to understand high-level idea of what a collection does

  - **Stack:** retrieves elements in reverse order as added.
  - **Queue:** retrieves elements in same order as added.

Wait, ADT? Interfaces?

- **Abstract Data Type (ADT)**: A *description of the idea* of a data structure including what operations are available on it and how those operations should behave. For example, the English explanation of what a list should be.

- **Interface**: Java construct that lets programmers *specify what methods a class should have*. For example the `List` interface in java.

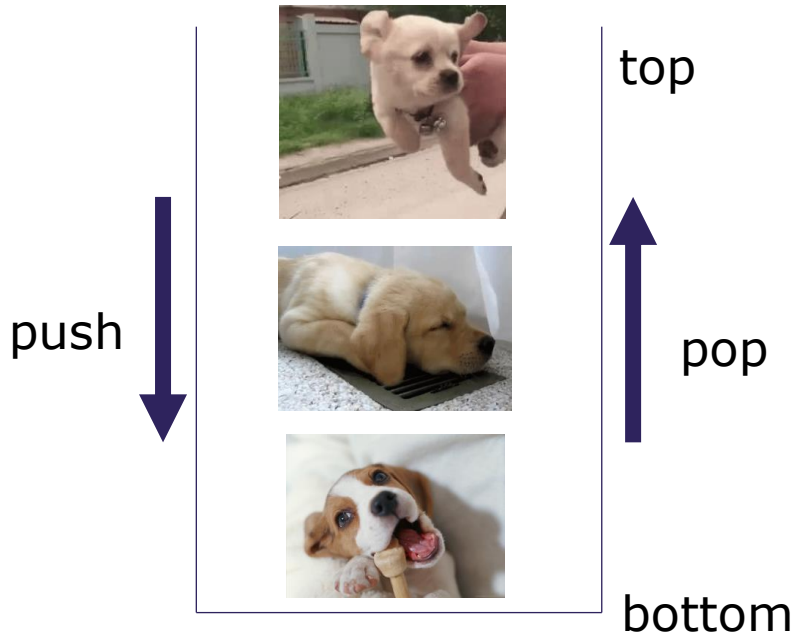
- **Implementation:** *Concrete code* that meets the specified interface. For example, the `ArrayList` and `LinkedList` classes that implement the `List` interface.

(PCM) Stacks

- **Stack:** A collection based on the principle of adding elements and retrieving them in the **opposite** order.
  - Last-In, First-Out ("LIFO")
  - Elements are stored in order of insertion.
    - We do not think of them as having indexes.
  - Client can only add/remove/examine the last element added (the "top")

Basic **Stack** operations:

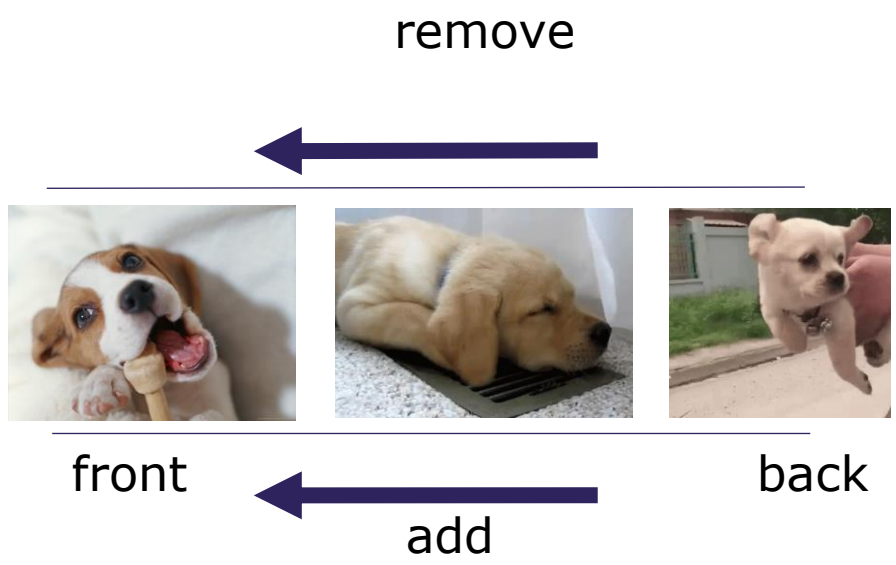
- **push**: Add an element to the top
- **pop**: Remove the top element
- **peek**: Examine the top element



(PCM) Queue

- **Queue**: Retrieves elements **in the order** they were added.
  - First-In, First-Out ("FIFO")
  - Elements are stored in order of insertion but don't have indexes.
  - Client can only add to the end of the queue, and can only examine/remove the front of the queue.

- Basic Queue operations:
  - **add** (enqueue): Add an element to the back.
  - **remove** (dequeue): Remove the front element.
  - **peek**: Examine the front element.



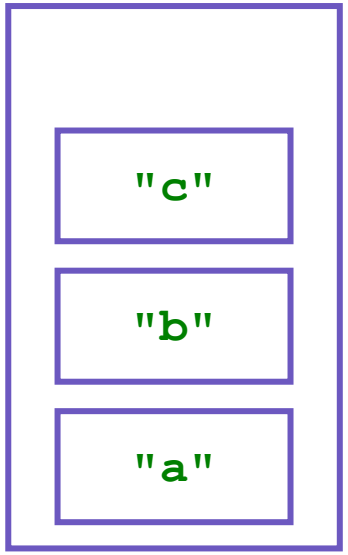
slido

Please download and install the Slido app on all computers you use

What are some real life stacks and queues?

ⓘ Start presenting to display the poll results on this slide.

(PCM) Programming with Stacks



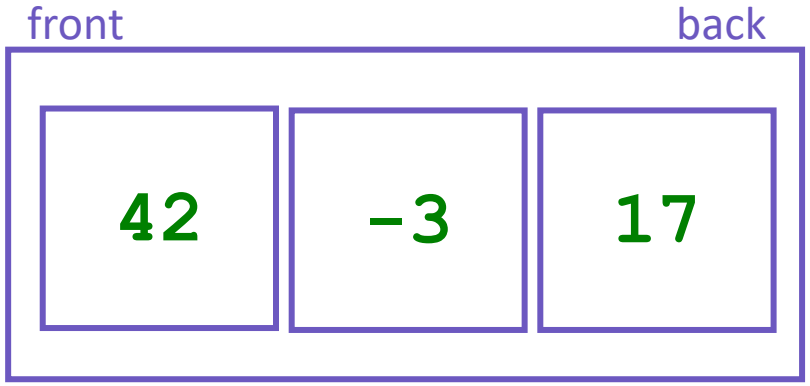
`Stack<E>()`	constructs a new stack with elements of type **E**
`push(value)`	places given value on top of stack
`pop()`	removes top value from stack and returns it; throws `EmptyStackException` if stack is empty
`peek()`	returns top value from stack without removing it; throws `EmptyStackException` if stack is empty
`size()`	returns number of elements in stack
`isEmpty()`	returns `true` if stack has no elements

```
Stack<String> s = new Stack<String>();
s.push("a");
s.push("b");
s.push("c");

System.out.println(s.pop());
```

- `Stack` has other methods that we will ask you not to use 😬

(PCM) Programming with Queues



`add(`**value**`)`	places given value at back of queue
`remove()`	removes value from front of queue and returns it; throws a `NoSuchElementException` if queue is empty
`peek()`	returns front value from queue without removing it; returns `null` if queue is empty
`size()`	returns number of elements in queue
`isEmpty()`	returns `true` if queue has no elements

```
Queue<Integer> q = new LinkedList<Integer>();
q.add(42);
q.add(-3);
q.add(17);

System.out.println(q.remove());
```

IMPORTANT: When constructing a queue you must use a new `LinkedList` object instead of a new `Queue` object. (More on that with Interfaces.)

Stacks in Computer Science

- Programming languages and compilers:
  - method calls are placed onto a stack (*call*↔*push, return* ↔ *pop*)
  - compilers use stacks to evaluate expressions
- Matching up related pairs of things:
  - find out whether a string is a palindrome
  - examine a file to see if its braces { } match
  - convert "infix" expressions to pre/postfix
- Sophisticated algorithms:
  - searching through a maze with "backtracking”
  - many programs use an "undo stack" of previous operations

Queues in Computer Science

- Operating systems:
  - queue of print jobs to send to the printer
  - queue of programs / processes to be run
  - queue of network data packets to send
- Programming:
  - modeling a line of customers or clients
  - storing a queue of computations to be performed in order
- Real world examples:
  - people on an escalator or waiting in a line
  - cars at a gas station (or on an assembly line)

Lecture Outline

- Announcements
- Review: Stacks & Queues
- **Queue Manipulation**
- Stack Manipulation
  - Problem Solving

Lecture Outline

- Announcements
- Review: Stacks & Queues
- Queue Manipulation
- **Stack Manipulation**
  - Problem Solving

Practice : Think

sli.do #cse122

What does this method return?

```
// numbers: bottom [1, 2, 3, 4, 5] top
public static int sum(Stack<Integer> numbers) {
    int total = 0;
    for (int i = 0; i < numbers.size(); i++) {
        int number = numbers.pop();
        total += number;
        numbers.push(number);
    }

    return total;
}
```

A) 0
B) 5
C) 12
D) 15
E) 25
F) Error / Exception

Practice : Pair

sli.do #cse122

What does this method return?

```
// numbers: bottom [1, 2, 3, 4, 5] top
public static int sum(Stack<Integer> numbers) {
    int total = 0;
    for (int i = 0; i < numbers.size(); i++) {
        int number = numbers.pop();
        total += number;
        numbers.push(number);
    }

    return total;
}
```

A) 0
B) 5
C) 12
D) 15
E) 25
F) Error / Exception

Practice : Think

sli.do #cse122

What does this method return?

```
// numbers: bottom [1, 2, 3, 4, 5] top
public static int sum(Stack<Integer> numbers) {
    Queue<Integer> q = new LinkedList<>();

    int total = 0;
    for (int i = 0; i < numbers.size(); i++) {
        int number = numbers.pop();
        total += number;
        q.add(number);
    }

    return total;
}
```

A) 0
B) 5
C) 12
D) 15
E) 25
F) Error / Exception

Practice : Pair

sli.do #cse122

What does this method return?

```
// numbers: bottom [1, 2, 3, 4, 5] top
public static int sum(Stack<Integer> numbers) {
    Queue<Integer> q = new LinkedList<>();

    int total = 0;
    for (int i = 0; i < numbers.size(); i++) {
        int number = numbers.pop();
        total += number;
        q.add(number);
    }

    return total;
}
```

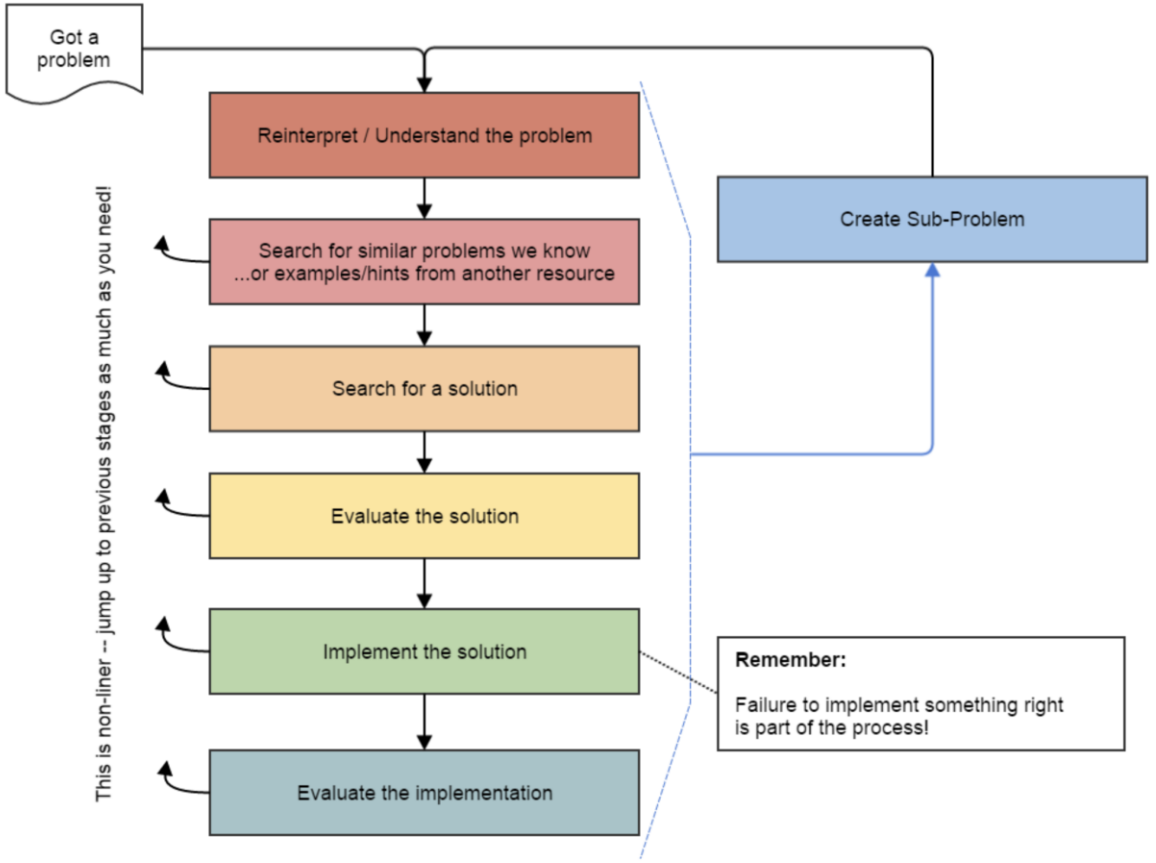
A) 0
B) 5
C) 12
D) 15
E) 25
F) Error / Exception

Lecture Outline

- Announcements
- Review: Stacks & Queues
- Queue Manipulation
- Stack Manipulation
  - **Problem Solving**

Problem Solving

- On their own, Stacks & Queues are quite simple with practice (few methods, simple model)
- Some of the problems we ask are complex *because* the tools you have to solve them are restrictive
  - sum(Stack) is hard with a Queue as the auxiliary structure
- We challenge you on purpose here to practice **problem solving**



Source: Oleson, Ko (2016) - Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance

Common Problem-Solving Strategies

- **Analogy** – Is this similar to a problem you've seen?
  - sum(Stack) is probably a lot like sum(Queue), start there!
- **Brainstorming** – Consider steps to solve problem before writing code
  - Try to do an example "by hand" → outline steps
- **Solve Sub-Problems** – Is there a smaller part of the problem to solve?
  - Move to queue first
- **Debugging** – Does your solution behave correctly on the example input.
  - Test on input from specification
  - Test edge cases ("What if the Stack is empty?")
- **Iterative Development** – Can we start by solving a different problem that is easier?
  - Just looping over a queue and printing elements

Common Stack & Queue Patterns

- Stack → Queue and Queue → Stack
  - We give you helper methods for this on problems
- Reverse a Stack with a S→Q + Q→S
- "Cycling" a queue: Inspect each element by repeatedly removing and adding to back `size` times
  - Careful: Watch your loop bounds when queue's size changes
- A "splitting" loop that moves some values to the Stack and others to the Queue