Poll Everywhere

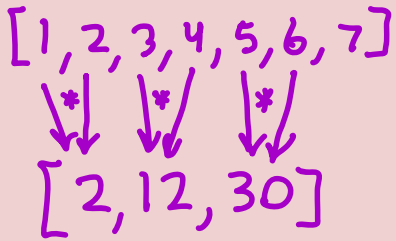
PollEv.com/2110fa25 text 2110fa25 to 22333

What is returned by `squish(list, (x, y) -> x * y)` if `list = [1,2,3,4,5,6,7]` ?

```
interface Combiner<T> { T combine(T x, T y); }

static <T> CS2110List<T> squish(CS2110List<T> list,  Combiner<T> f) {
  CS2110List<T> out = new SinglyLinkedList<>();
  Iterator<T> it = list.iterator();
  while (it.hasNext()) {
    T temp = it.next();
    if (it.hasNext()) { out.add(f.combine(temp, it.next())); }
  }
  return out;
}
```

1. grab next 2 elements from list
2. combine them
3. write to out



Lecture 15: Stacks and Queues

CS 2110

October 16, 2025

Today's Learning Outcomes

63. Identify use cases for the Stack and Queue data structures.

64. Describe composition relationships and scenarios where they may be preferable to inheritance.

57. Compare the performance of a List implemented with a dynamic array and a List implemented with a linked chain. Determine which is preferable for a given use case.

Processing Incoming Data

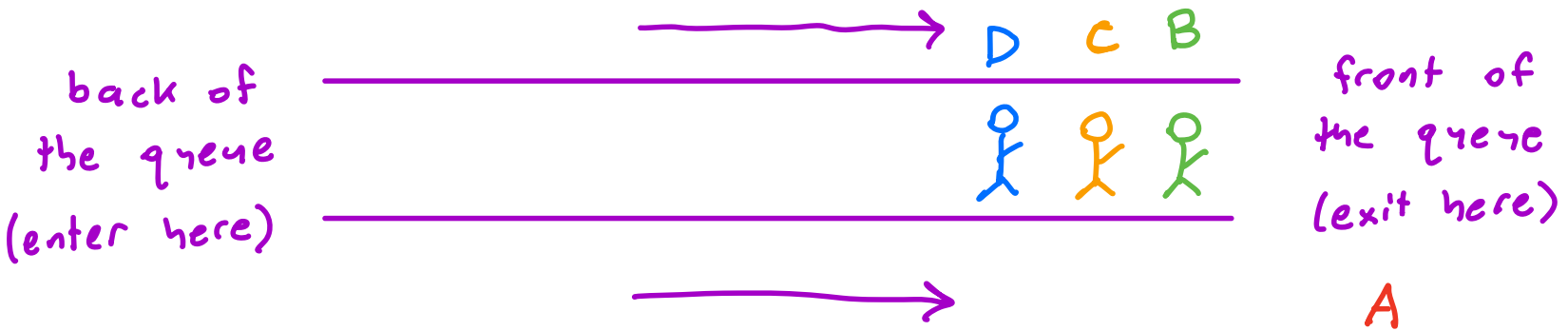
Many applications involve gathering up a lot of data, and then systematically processing its elements one at a time.

Examples:

- Processing sensor readings
- Managing customer transactions in an online store
- Correctly executing code instructions divided across multiple method calls
- Handling complicated game mechanics in deck-building games
- Iterating over nodes in more advanced linked structures (e.g. trees and graphs)
- Admitting incoming patients to a hospital (coming soon)

Queues and FIFO Ordering

people waiting in line



- anyone can join the queue at any time
- the only person who can exit the queue (be served next) is the person at the front, who has been there the longest

Queues enforce "First in, First out" (FIFO) order condition.

The Queue<T> ADT

Like all "Ordered collections", supports 4 operations:

1. Add another element

```
/** Adds to back of queue */
void enqueue(T elem);
```

2. Remove and return a specific element (ADT determines which)

```
/** Removes + returns front element */
T dequeue();
```

3. Access next element that will be removed (without removing)

```
/** Returns front element */
T peek();
```

4. Check if there are more elements to process

```
boolean isEmpty();
```

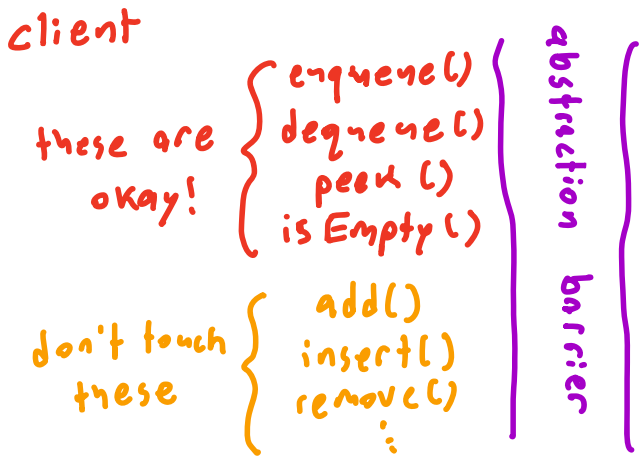
Coding Demo: Queues via Inheritance

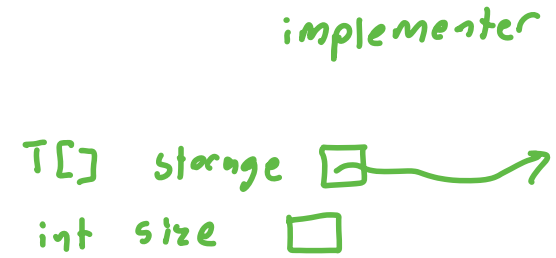
The Problem with Inheritance

When a class B extends class A, B's client interface automatically includes every method in A's client interface.

Sometimes, this exposes excess functionality to the client that lets them circumvent B's specs.

- add(), remove(), etc. break Queue's FIFO guarantee

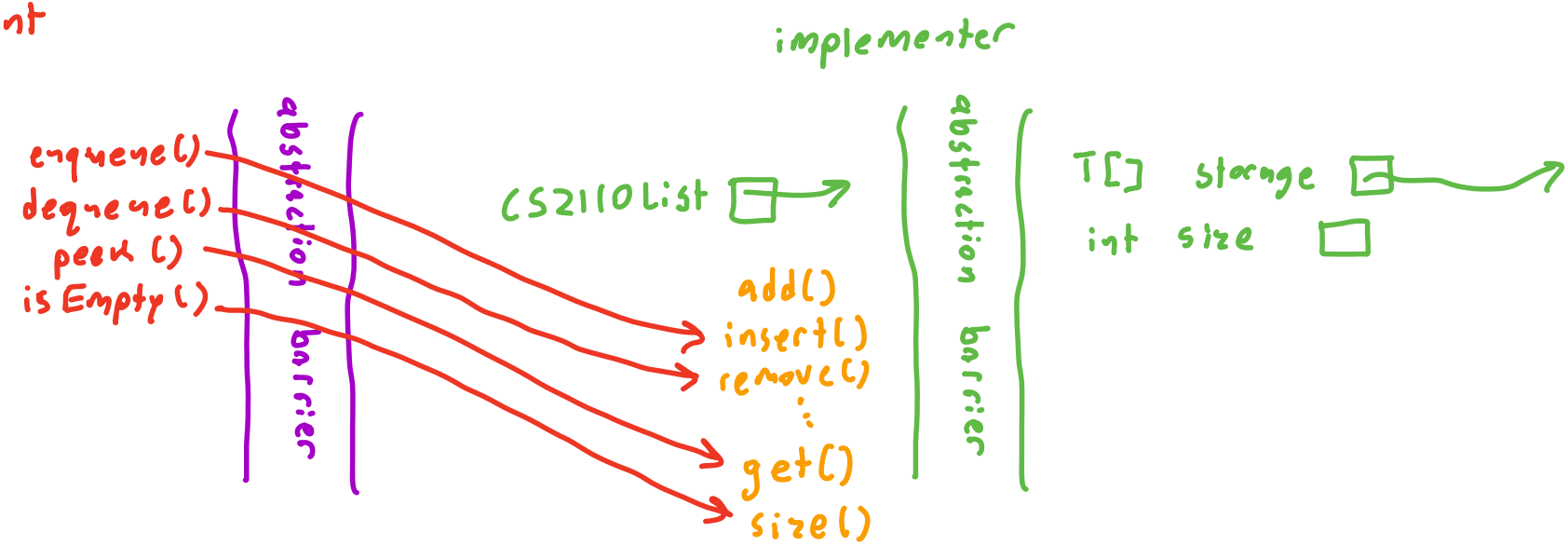




Composition Relationships

Idea: "Move" some methods behind the abstraction barrier, outside of the client's view.

Queue class manages list as a (private) field. Without inheritance, it can decide on its own client interface.



Coding Demo: Queues via Composition

Poll Everywhere

PollEv.com/2110fa25 text `2110fa25` to `22333`

If we implement a Queue via composition with a `SinglyLinkedList` which state representation is preferable?

deleting from end of SLL is O(N), so avoid it!

head = back of the queue	(A)
head = front of the queue	(B)
both options work equally well	(C)

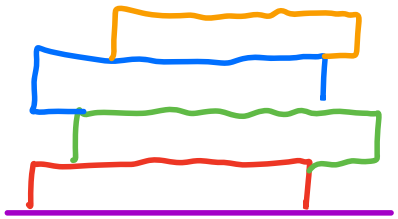
Queue Operation Complexity

	push()	pop()	peek()	isEmpty()
DAL	O(N)	O(1)	O(1)	O(1)
SLL	O(1)	O(1)	O(1)	O(1)

Can improve push() to amortized O(1) using a dynamic circular buffer (see notes)

Stacks and LIFO Ordering

When we stack up objects (e.g., books) we only ever have direct access to the one on top.



Top book = most recent one to be added to stack

Stacks are ordered collections that enforce a "Last in, First out" (LIFO) order condition

The runtime stack is a stack of call frames

currently executing method is at top of the stack and will be the first to be removed when it finishes executing

The Stack<T> ADT

Like all "Ordered collections", supports 4 operations:

1. Add another element

```
/** Adds to top of stack */
void push(T elem);
```

2. Remove and return a specific element (ADT determines which)

```
/** Removes + returns top element */
T pop();
```

3. Access next element that will be removed (without removing)

```
/** Returns top element */
T peek();
```

4. Check if there are more elements to process

```
boolean isEmpty();
```

Coding Demo: Implementing Stacks

Poll Everywhere

PollEv.com/2110fa25 text `2110fa25` to `22333`

If we implement a `Stack` via composition with a `SinglyLinkedList` which state representation is preferable?

deleting from end of SLL is O(N), so avoid it!

head = top of the stack	(A)
head = bottom of the stack	(B)
both options work equally well	(C)

Stack Operation Complexity

	`push()`	`pop()`	`peek()`	`isEmpty()`
DAL	Amortized O(1)	O(1)	O(1)	O(1)
SLL	O(1)	O(1)	O(1)	O(1)

Stack Application: Expression Parsing

Given a string representing a mathematical expression, can we determine its value (in O(N) time)?

"3 * (6+2)"

Idea: Read characters one at a time L → R and keep track of "state" of parsing as we go

Potential Issue: May need to keep track of arbitrarily large state

"2 * (3 + 4 * (5 + 6 * (7+8)))

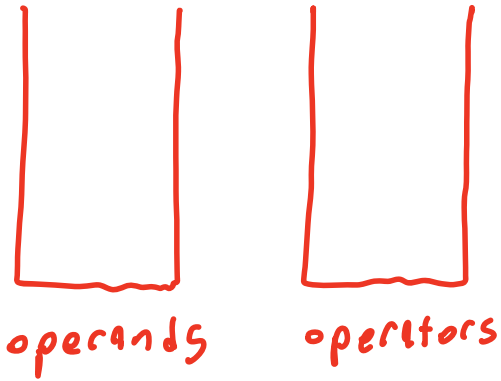
order of these operations: 6 5 4 3 2 1

simplification happens R → L, so we can use stacks to keep track of state

Operator and Operand Stacks

Idea: - Maintain 2 stacks, one for operands, one for operators

- Use order of operands to figure out when to simplify



3 * 4 + 6 * (2 + 7) * 5

* see lecture notes for an animated version of this example

\# → push onto operands

* → simplify any * on operators stack

\+ → simplify any +, * on operators stack

(→ push onto operators

) → simplify operators until we find corresponding (

Coding Demo: `ExpressionEvaluator`