Stacks and Queues

Instructors: Sam McCauley and Dan Barowy

April 4, 2022

Admin

- Welcome back!
- Lots handed back this week (midterm pretty much graded; probably back on Friday)
- Think about applying to be a TA!
- Any questions?

Second Half of the Course

- First half was foundations
  - Some basic data structures
  - How to use Java
  - Analyzing performance, proving correctness (big-*O* notation and induction)
- Second half: focus more on new data structures
  - Go beyond list-like data structures
  - This week: improve simplicity
  - Later: (drastically) improve performance

Main Ideas For Today

Two Simple Data Structures

- Can only perform a *subset* of the operations that arrays or vectors or linked lists can
- Goal: simple interface, flexible, good performance
  - Data structures that do less can be easier to work with
- These two data structures are ubiquitous in computer science

Two Simple Data Structures

Stack

Queue

(NB: no cutting allowed in our queues!)

Stacks

Stack

- Can only add or remove to the *top* of the stack
- (No adding to the middle or removing from the middle.)
- The item we remove is the *most recently added* item

Stack Operations

Stacks have their own vocabulary.

- `push()`: Add a new item to the top of the stack
  - Think `addLast()`
- `pop()`: Remove (and return) the top item on the stack
  - Think `removeLast()`
- `peek()`: Return the top item on the stack without removing
  - Think something like `get(size() - 1)`

Stack Operations

There are a few ways to implement a stack: it could be that we're adding and removing the first element!

- `push()`: Add a new item to the top of the stack
  - *Alternative:* `addFirst()`
- `pop()`: Remove (and return) the top item on the stack
  - *Alternative:* `removeFirst()`
- `peek()`: Return the top item on the stack without removing
  - *Alternative:* `get(0)`

Stack

- Only three operations to worry about!
- How can we implement a stack using data structures we already have?
- What are the tradeoffs between some of the options we have?

Implementing a Stack with an Array

- Can be found in the `structure5` `StackArray` class. Let's take a look.
- Downside: need to declare array size up front; stack cannot grow beyond this size.
- How can we keep track of what to add/remove?
  - Keep an `int top` holding the location of the top element in the stack
- Running time for operations?
  - `push()`: $O(1)$
  - `pop()`: $O(1)$
  - `peek()`: $O(1)$

Implementing a Stack with a Vector

- Can be found in the `structure5` `StackVector` class. Let's take a look.
- Don't need `int top` anymore
- Running time for operations?
  - `push()`: $O(n)$ in the worst case. $O(1)$ "on average"!
  - `pop()`: $O(1)$
  - `peek()`: $O(1)$
- Downside? $O(n)$ extra space

Implementing a Stack with a Linked List

- Is a `SinglyLinkedList` a good idea? (Or do we need a `DoublyLinkedList` for efficiency?)
- Singly linked works fine if we have the top element as the *head* of the list. Let's take a look at `StackList`
- Running time for operations?
  - `push()`: $O(1)$
  - `pop()`: $O(1)$
  - `peek()`: $O(1)$
- Downside? $O(n)$ extra space

When to Use Stacks?

- Classic example: JVM call stack!
  - Keeps track of what methods we have called
  - Each time a new method is called, we push it on the top of the stack
  - When the method returns, pop it off the top of the stack
- Useful in implementing backtracking search
- Or any last-in-first-out usage

Fitting into `structure5`

- Probably want a `Stack` interface, with methods like `pop()`, `push()`, `peek()`
- Recall: Why are interfaces useful?
  - Can declare a `Stack` object and access stack methods like `pop()` or `peek()` on it.
  - Can change the underlying `Stack` class it's instantiated with, without changing how it's used!

Interface example

```
Stack<Integer> s = new StackArray<Integer>(10); //max size 10
//can swap with the next line to remove max size:
//Stack<Integer> s = new StackList<Integer>();

for(int i = 0; i < 10; i++) {
   s.push(i);
}
for(int i = 0; i < 10; i++) {
   System.out.println(s.pop());
}
```

Queues

Queues

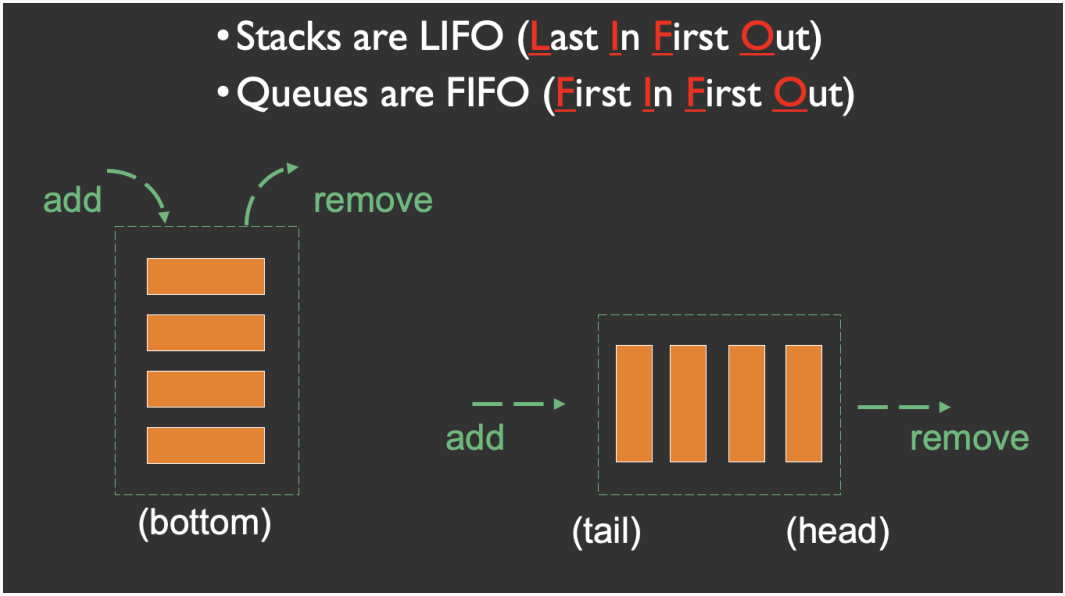
- Same idea as stacks: can only access one element
- Stacks are FILO (**F**irst **I**n **L**ast **O**ut)
- Queues are FIFO (**F**irst **I**n **F**irst **O**ut)

Queues

- Think of a queue as waiting in line
- The first to join the queue is the first to leave

Stacks vs Queues

- Stacks are LIFO (Last In First Out)
- Queues are FIFO (First In First Out)



Queue Operations

- `enqueue()`: insert a value at the back of the queue
  - Think `addLast()`
- `dequeue()`: remove and return the value from the front of the queue
  - Think `removeFirst()`
- `peek()`: access the first value of the queue without removing it

Queue Interface Main Idea

```
public interface Queue<E> {
   public void enqueue(E item);
   public E dequeue();
   public E peek();
   public int size();
}
```

How to Implement a Queue?

- What data structures can we use?
  - Array: leads to the `QueueArray` class
    - What do we need to store?
    - Need both the `head`, and the `count` of items stored in the queue
  - Vector: leads to the `QueueVector` class
    - For a Vector we can just call `addLast` and `removeFirst`; don't need to change anything
  - Linked List: leads to the `QueueList` class
    - We'll discuss in a second. For now: can we use a Singly Linked List? Doubly Linked List? What are the tradeoffs?

QueueArray

- Like `StackArray` has a max number of elements it can store
- Keeps two `int`s in addition to the array: `head` and `count`
- Key idea: we *wrap around* the array as new items are enqueued and old items are dequeued
- Let's look at the code
- Cost for `enqueue()`?
  - $O(1)$
- Cost for `dequeue()`?
  - $O(1)$

QueueVector

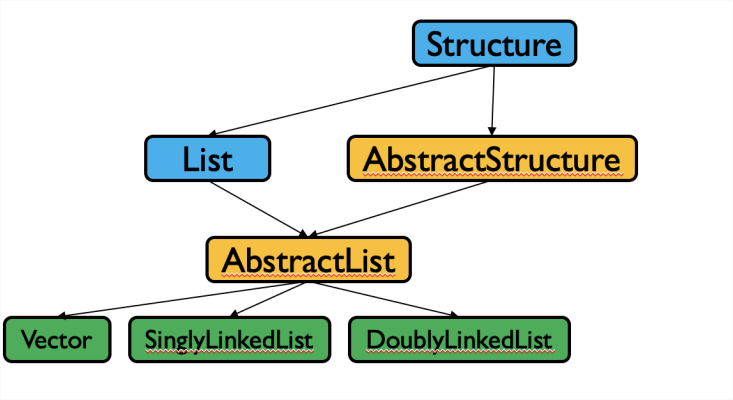
- Just call `addLast` and `removeFirst`
- Time for `enqueue()`?
  - $O(1)$
- Time for `dequeue()`?
  - $O(n)$ (this is terrible! Never use a `QueueVector`.)

QueueList

- We want efficient `addLast` and `removeFirst`
- Singly linked lists have inefficient `addLast`
  - Side note: it's easy to modify so that we get $O(1)$ for both using singly linked nodes, i.e. by adding a `tail` pointer
  - Can also use `CircularList`; this is what the code does (see textbook)
- Let's consider a doubly linked list for the sake of discussion (only downside: slightly wasteful for space)
- Time for `enqueue()`?
  - $O(1)$
- Time for `dequeue()`?
  - $O(1)$

Fitting Into Structure5

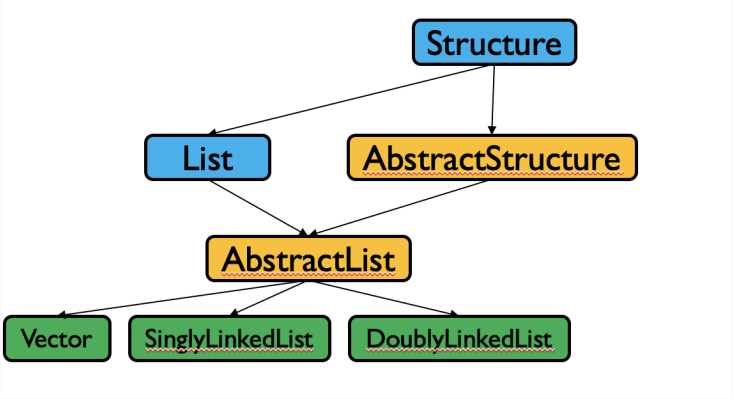
Putting the Classes Together



Blue: Interface
Yellow: Abstract Class
Green: Class

Remember that we can create simpler, more flexible code using interfaces and abstract classes. How can stacks and queues fit into `structure5`?

Putting the Classes Together



Blue: Interface
Yellow: Abstract Class
Green: Class

Where do stacks and queues go here? Are they a `List`? Are they a `Structure`? Let's look at both interfaces.

Stacks and Queues

- They are not a `List`: don't have methods like `get(int i)` or `indexOf()`
- They probably could be a `Structure`: methods like `size()` and `clear()` make sense, as do `add()` and `remove()`
  - This is a judgement call to some extent!
  - In `structure5`, stacks and queues do implement `Structure`

Filling out `structure5`

- First: a `Linear` interface common to both stacks and queues, and an `AbstractLinear` abstract class
    - These don't do too much; feel free to look at them
- Then, the `Stack` and `Queue` interface extend the `Linear` interface
- Have an `AbstractStack` and `AbstractQueue` abstract class
- Finally, each stack class `implements Stack` and `extends AbstractStack` (likewise for queues)

AbstractStack

- What methods are common to all stacks?
- Hint: abstract classes are very good for implementing methods that just call other methods
- Hint 2: the `Structure` interface promised some methods that don't quite line up with the stack terminology...
- Idea: we can implement `push()` by calling `add()` and `pop()` by calling `remove()`, and so on
- Same for `AbstractQueue`!
- Let's take a look at them

Current Structure5 Universe