

CS 310: Stack and Queue

Connor Baker

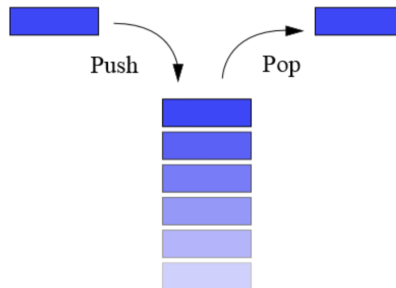
February 12, 2019

Review

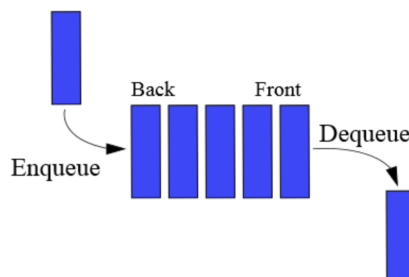
- **Iterators**
 - Motivation: why do we need iterators?
 - Implementation: how do we support efficient iterations?
 - Nested class / inner (anonymous) class
- **Take-home**
 - When you use a data structure, use an **Iterator** to improve efficiency and uniformity
 - When you design or implement a data structure, consider providing an **Iterator** for the above reason

New Topic

- **Stack**
 - A data structure that works like a stack (what a twist!)

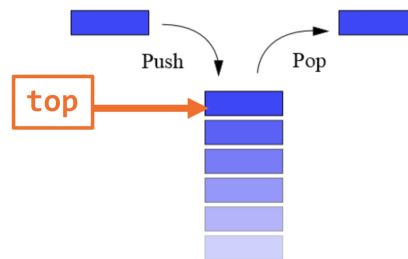


- **Queue**
 - A data structure that works like people waiting in a line (or queue if you're British)



Stack

- Features
 - LIFO
 - Always operates at the top of the stack
- Basic operations
 - `push(x)`: add `x` to the top of the stack (grows the stack)
 - `pop()`: remove the top of the stack (shrinks the stack)
 - `top()`: return the top of the stack (size is not changed)
 - `isEmpty()`: true when nothing is in it, false otherwise
- Implementation
 - Based on array / linked list



Stack Example

- You need to be able to draw the stack contents

```
s = new Stack();
s.push(4);
s.push(10);
s.push(5);
s.pop();
s.push(11);
```

Stacks based on Arrays

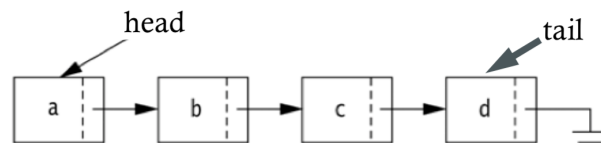
```
class AStack<T>{
    private ArrayList<T> stuff;
    public AStack(); // Constructor
    public void push(T x); // like add(x) or append(x)
    public void pop(); // like remove(size()-1)
    public T top(); // like get(size()-1)
    public boolean isEmpty(); // like size()==0
}
```

- Use an `ArrayList` as the underlying storage
- The top of the stack is the end of the array
 - Operations are performed only at the end which makes it faster with an array based implementation
- What's the Big-O?
 - $O(1)$

Stack Based on Linked List

```
Class LStack<T>{  
    private LinkedList<T> stuff;  
    public LStack(); // Assume head as stack top  
    public void push(T x); // like insert(0,x)  
    public void pop(); // like remove(0)  
    public T top(); // like get(0)  
    public boolean isEmpty(); // like size()==0  
}
```

- Use a Linked List as the underlying storage
 - Operate only at one end
- Big-O?
 - $O(1)$



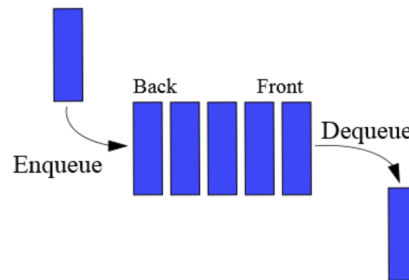
Stack Applications

- Check the symbolic balancing of an equation
 - $\{(<> [{<>}])\}\}$ vs. $\{(< [{<>>}])\}\}$
- Postfix calculation
 - $6\ 5\ 2\ 3\ +\ 8\ \times\ +\ 3\ +\ \times\ =$
- Infix to Postfix conversion
 - $a + b \times c + (d \times e + f) \times g \rightarrow abc \times + de \times f + g \times +$
- Call stack
 - `fib(4)`
- Tree traversal – preorder traversal
- Graph search – depth first search
- And a bunch of over applications

Queue

- Features
 - FIFO
 - Only remove from front
 - Only add to back
- Basic operations
 - `enqueue(x)` or `add(x)`: x enters at the back

- `dequeue()` or `poll()`: front leaves
- `getFront()` or `peek()`: returns the item at the front
- `isEmpty()`: true when nothing is in it, false otherwise

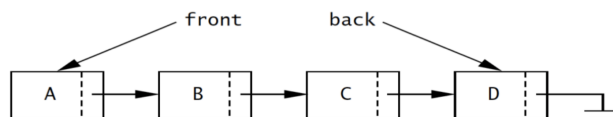


Queue Example

- You need to be able to draw the queue contents
- What is the value of `v`?

```
q = new Queue();
q.enqueue(4);
q.enqueue(10);
q.enqueue(5);
q.dequeue();
v = getFront();
q.dequeue();
q.enqueue(11);
q.enqueue(25);
```

Queue Based on Linked Lists

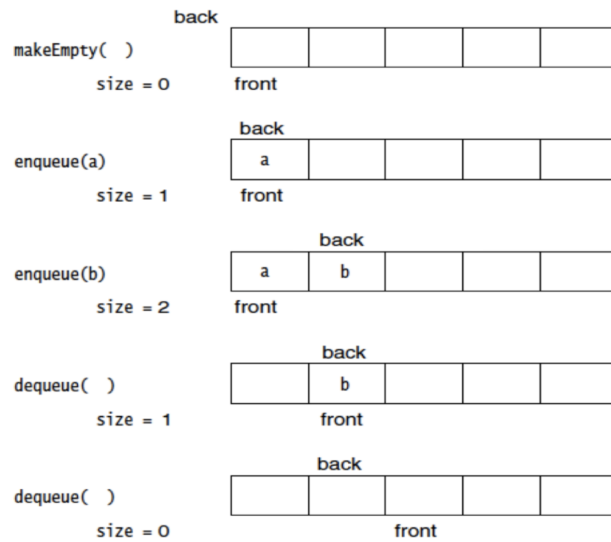


- Append to one end, and remove from the other end
 - For example, `head` → `front`, `tail` → `back`
 - `enqueue(x)`: insert at the tail
 - `dequeue()`: remove from head
 - `getFront()`: return head contents
 - `isEmpty()`: `size() == 0`

Queue Based on Arrays

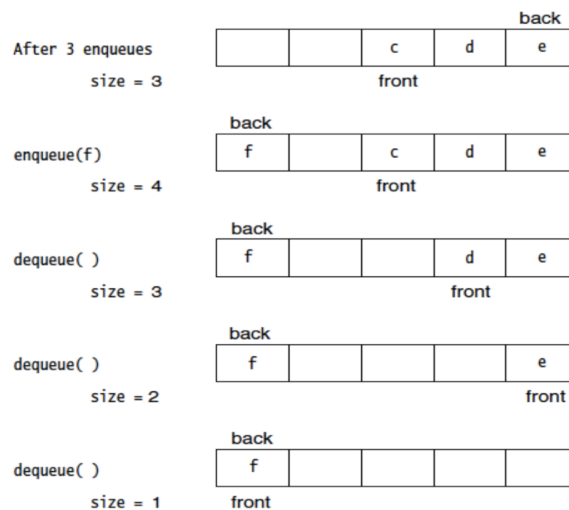
- Naive implementation:
 - `enqueue(x)`: insert at the end
 - `dequeue()`: remove from start and shifting internally
 - * In fact, a *lot* of shifting! Shifting is done for every single `dequeue()`!
 - Alternatively, we could just mark the front and the back in the array and update them with `enqueue` and `dequeue`

Queue Based on Arrays



- Between the front and the back, we have a valid queue
 - There's no shifting: $O(1)$ for `dequeue()`!
 - But it does use a sizeable amount of space

Queue: Array with Wraparound



- Exercise: what needs to be changed to implement the wraparound functionality?

Big-O Comparison

- Stack

Implementation	push()	pop()	top()	isEmpty()	size
Array	1*	1	1	1	1
Linked List	1	1	1	1	1

*Amortized analysis

Implementation	<code>enqueue()</code>	<code>dequeue()</code>	<code>getFront()</code>	<code>isEmpty()</code>	<code>size</code>
Array	1*	1	1	1	1
Linked List	1	1	1	1	1

*Amortized analysis

Why use a Stack or Queue

- Restricted operations give us good worst cases
 - $O(1)$ for all supported operations
 - $O(n)$ for space
- Simple data structures
 - Focus on limited operations
 - Can be made out of primitive data structures (arrays and linked lists)
- Good for representing time-related data
 - Call stack
 - Packet queues

Review: Queues

- FIFO
- Supported operations:
 - `enqueue(x)`: insert at the tail
 - `dequeue()`: remove from head
 - `getFront()`: return head contents
 - `size()`: returns the size of the queue
 - `isEmpty()`
- Applications:
 - Simulate a process with a FIFO order
 - Scheduling queue of a CPU or disk or printer
 - Serve as a buffer for file I/O, network communications, etc.

Priority Queues

- Much of the time tasks that we use a queue for have different priorities
 - It is convention that the lower the priority, the better
 - Symmetric code if higher is better
 - Dequeue the ones with the “best” priority first
- Common priority queue operations
 - `void insert(T x, int p)`: insert `x` with priority `p`
 - `T findMin()`: return the object with the “best” priority
 - `deleteMin()`: remove the object with the “best” priority

Priority Queue Design

Data Structure	<code>insert()</code>	<code>findMin()</code>	<code>deleteMin()</code>	Notes()
Sorted Array	$O(n)$	$O(1)$	$O(1)$	min at high index
Sorted Linked List	$O(n)$	$O(1)$	$O(1)$	min at head or tail

- Other data structures exist as good candidates of priority queues
 - Binary search trees
 - Heaps
 - We'll cover these later

Summary

- Stacks and queues
 - Try implementing them
 - Project 2
- Next lecture: Trees, recursion
 - Reading: Chapter 18, Chapter 7