

CSC 172– Data Structures and Algorithms

Lecture #12

Spring 2018

Please put away all electronic devices



Student Feedback #1

- For Lab02, I received a grade of negative zero (-0.00000/100). Should I be concerned or is this just a Blackboard glitch/mistake?
- Yes. Please contact your Lab TAs asap.
- We will not consider any regrade request for any quiz (1-6), lab (1-5), project (1) after the midterm.

Student Feedback #2

- Website header still doesn't load on ipad or iphone
- I have added a quick fix! Thanks for letting me know.

Student Feedback #3

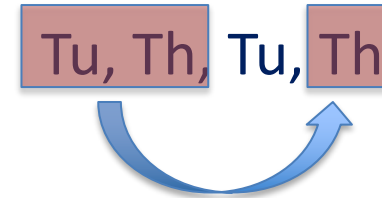
- The online lectures have a tendency to lose audio at different times of the whole lecture.
- Lecture video for 2/22/18 has no audio, sounds like maybe your microphone got unplugged or something
- Sorry about that.
- Lecture videos picks the audio only from the wireless microphone. If you ever find the mic is off, please let me know immediately. I (along with your friends who are not attending the lecture) will really appreciate it.

Student Feedback #4

- I use my computer to take notes and the policy of no electronics is hindering my ability to make the most out of the course. I would appreciate it some exceptions are made, for example, its okay to use electronics to take notes if sitting in the back rows.
- Please meet me in person. I allow exceptions. You can use laptops following these two steps:
 - 1. MUST: You must meet me in person before I grant you the privilege
 - 2. After step 1, You can use your laptop sitting in back rows. As long as no other students can be distracted by your laptop screen, you are good.
 - <https://www.nytimes.com/2017/11/22/business/laptops-not-during-lecture-or-meeting.html>

Student Feedback #5

- I am usually concerned and lost about the topic of the quiz for the week. It would be very helpful and appreciated if this information could be shared online and during the lecture
- I won't do that intentionally.
- The quiz on a particular week covers:
 - What we covered last week.
 - **Tu, Th, Tu, Th** (Nothing from the current week)
 - All quizzes may/may not have coding components. You should always be prepared to write code if required.
 - Usually, Labs and Workshops cover the same material.
 - As this information is common knowledge, I will NOT answer this question on Piazza.



Agenda

- Shunting Yard Algorithm
- How to Implement Stack and Queue
 - Cost of various operations

For all data structures, this is how we will proceed:

1. Use the data structures if readily available in Java
2. Will see how Java implements it
3. Implement / Figure out how to implement

Practice Problem (Infix to Postfix)

- $24 + 4 - 15 * 8 / (2 + 5 * 2 - 8) + 4$

24 4 + 15 8 * 2 5 2 * + 8 - / - 4 +

Input	List	Stack
24	24	
+		+
4	24 4	
-	24 4 +	-
15	24 4 + 15	
*		_*
8	24 4 + 15 8	
/	24 4 + 15 8 *	-/
(-/ (
2	24 4 + 15 8 * 2	
+		-/ (+
5	24 4 + 15 8 * 2 5	
*		-/ (+ *
2	24 4 + 15 8 * 2 5 2	
-	24 4 + 15 8 * 2 5 2 * +	-/ (-
8	24 4 + 15 8 * 2 5 2 * + 8	
)	24 4 + 15 8 * 2 5 2 * + 8 -	-/
+	24 4 + 15 8 * 2 5 2 * + 8 - / -	+
4	24 4 + 15 8 * 2 5 2 * + 8 - / - 4 +	Empty

STACK AND QUEUE IMPLEMENTATION

Java Stack

Modifier and Type	Method and Description
boolean	empty() Tests if this stack is empty.
E	peek() Looks at the object at the top of this stack without removing it from the stack.
E	pop() Removes the object at the top of this stack and returns that object as the value of this function.
E	push(E item) Pushes an item onto the top of this stack.
int	search(Object o) Returns the 1-based position where an object is on this stack.

Stack

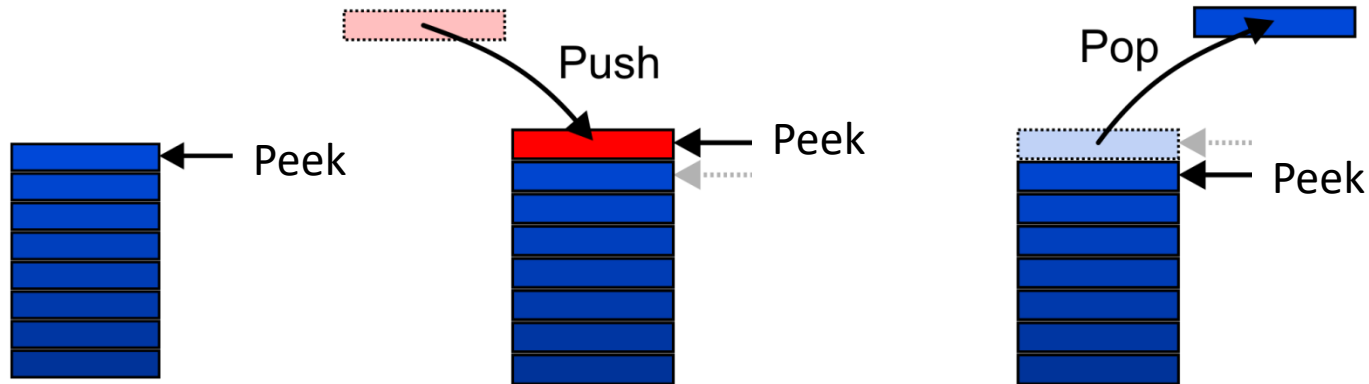
A **Stack** is a data type which emphasizes specific operations:

- Uses an explicit **linear ordering**
- Inserted objects are *pushed onto* the stack
- The *top* of the stack is the most recently object pushed onto the stack
- When an object is *popped* from the stack, the current *top* is erased

Stack

Also called a *last-in–first-out (LIFO)* behavior

- Graphically, we may view these operations as follows:



Applications

Numerous applications:

- Parsing code:
 - Matching parenthesis
 - HTML
- Tracking function calls
- Dealing with undo/redo operations
- Reverse-Polish calculators

The stack is a very simple data structure

- Given any problem, if it is possible to use a stack, this significantly simplifies the solution

Implementations

We will look at **three** implementations of stacks:

The optimal asymptotic run time of any algorithm is $\Theta(1)$

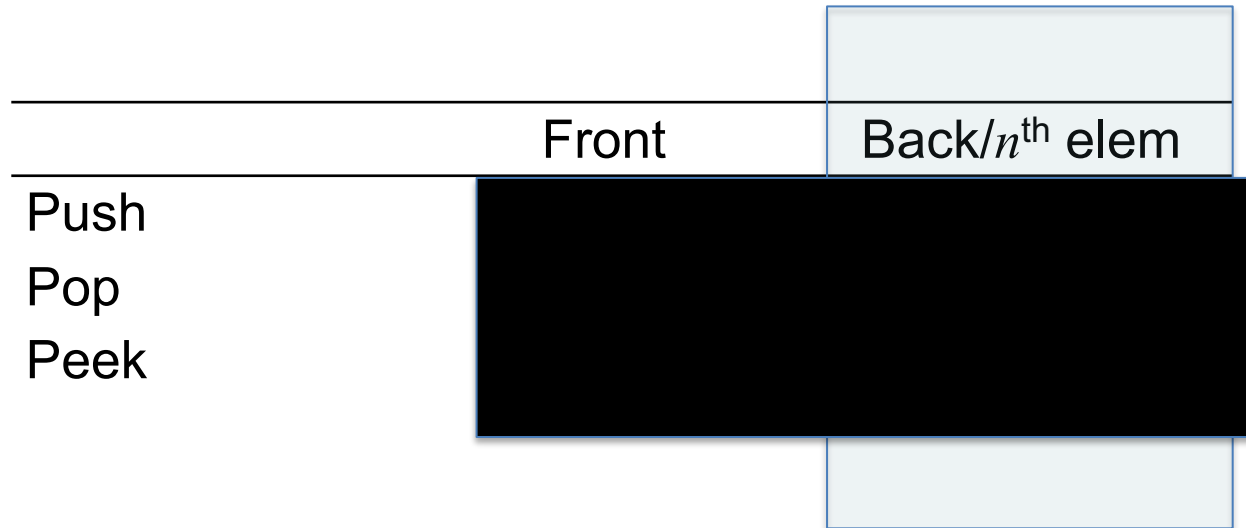
We will look at

- Singly Linked Lists
- Doubly Linked Lists
- Arrays (or ArrayList or Vectors)

Note: Java implements Stack by extending Vector<E> class which is very similar to the ArrayList<E> (with some changes).

The core container for both, ArrayList and Vector is an Array.

Arrays




The desired behavior of a Stack may be reproduced by performing all operations at the back of an array.

Arrays

	Front	Back/ n^{th} elem
Push	$O(n)$	$\Theta(1)$
Pop	$O(n)$	$\Theta(1)$
Peek	$\Theta(1)$	$\Theta(1)$

The desired behavior of a Stack may be reproduced by performing all operations at the back of an array.

Doubly Linked List (or LinkedList)

	Front	Back/ n^{th} elem
Push		
Pop		
Peek		

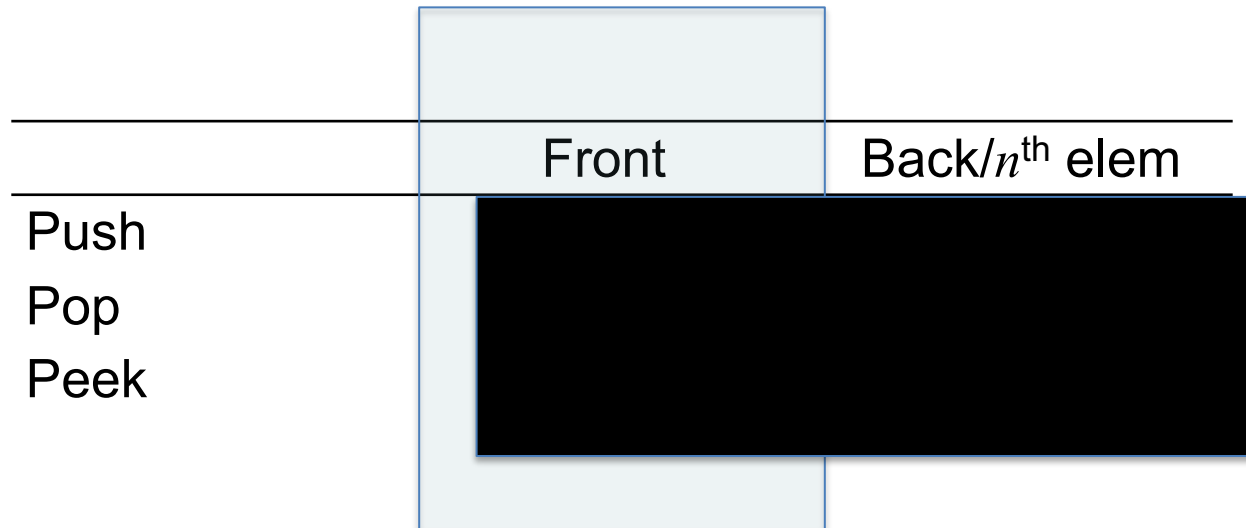
The desired behavior of a Stack may be reproduced by performing all operations at either side. But you can't avoid overhead of a doubly-linked list anyway.

Doubly Linked List (or LinkedList)

	Front	Back/ n^{th} elem
Push	$\Theta(1)$	$\Theta(1)$
Pop	$\Theta(1)$	$\Theta(1)$
Peek	$\Theta(1)$	$\Theta(1)$

The desired behavior of a Stack may be reproduced by performing all operations at either side. But you can't avoid overhead of a doubly-linked list anyway.

Singly Linked List



The desired behavior of a Stack may be reproduced by performing all operations at the front of a singly linked list.

Singly Linked List

	Front	Back/ n^{th} elem
Push	$\Theta(1)$	$\Theta(1)$
Pop	$\Theta(1)$	$O(n)$
Peek	$\Theta(1)$	$\Theta(1)$

The desired behavior of a Stack may be reproduced by performing all operations at the front of a singly linked list.

QUEUE

Queue

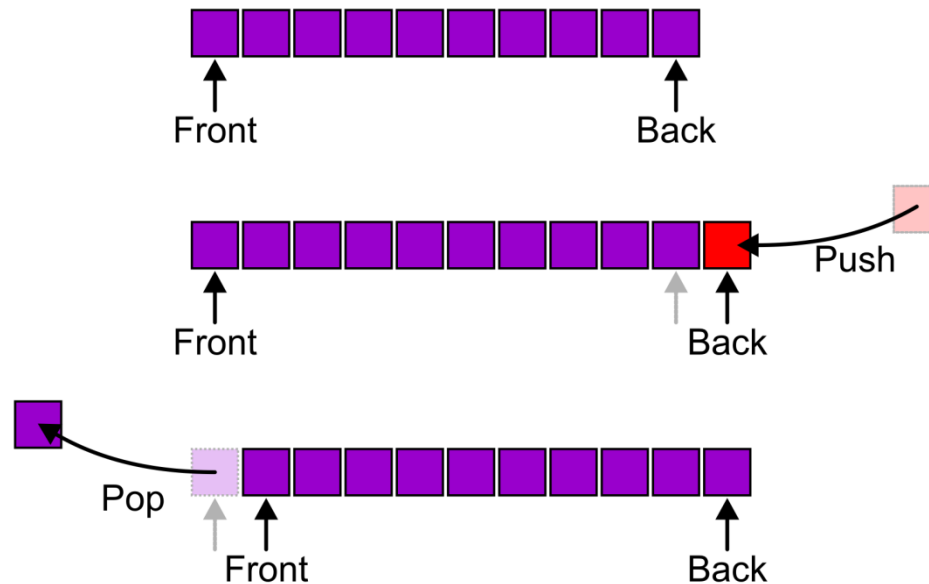
A **Queue** (Queue ADT) is an abstract data type that emphasizes specific operations:

- Uses an explicit **linear ordering**
- Insertions and removals are performed individually
- The object designated as the **front** of the queue is the object which was in the queue the longest
- The remove operation removes the current **front** of the queue

Queue

Also called a *first-in–first-out* (FIFO) data structure

- Graphically, we may view these operations as follows:



Queue

Queue Interface Structure

Type of Operation	Throws exception	Returns special value
Insert	<code>add(e)</code>	<code>offer(e)</code>
Remove	<code>remove()</code>	<code>poll()</code>
Examine	<code>element()</code>	<code>peek()</code>

add / offer

- The **add** method, which Queue inherits from Collection, inserts an element unless it would violate the queue's capacity restrictions, in which case it throws `IllegalStateException`.
- The **offer** method, which is intended solely for use on bounded queues, differs from `add` only in that it indicates failure to insert an element by returning `false`.

remove / poll

The **remove** and **poll** methods both remove and return the head of the queue.

The remove and poll methods differ in their behavior only when the queue is empty. Under these circumstances, remove throws NoSuchElementException, while poll returns null.

element / peek

- The element and peek methods return, but do not remove, the head of the queue.
- They differ from one another in precisely the same fashion as remove and poll: If the queue is empty, element throws NoSuchElementException, while peek returns null.

Example

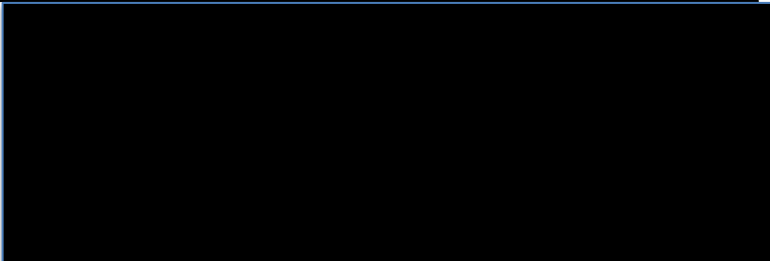
```
public static void main (String[] args)
{
    Queue<Integer> queue = new LinkedList<>();

    for(int i = 0; i <= 10; i=i+2) {
        queue.add(i);
    }

    System.out.println("Front of queue = "+ queue.element());
    System.out.println("Removing element = "+ queue.remove());
    System.out.println("Front of queue = "+ queue.element());
    System.out.println("Removing element = "+ queue.remove());
    System.out.println("Removing element = "+ queue.remove());
    System.out.println("Front of queue = "+ queue.element());
    queue.add(12);
    System.out.println("Front of queue = "+ queue.element());
}
```

```
Front of queue = 0
Removing element = 0
Front of queue = 2
Removing element = 2
Removing element = 4
Front of queue = 6
Front of queue = 6
```

Arrays

	Front	Back/ n^{th} elem
add/offer		
remove/poll		
element/peek		


The desired behavior of a Queue can't be achieved using an array. Unless....

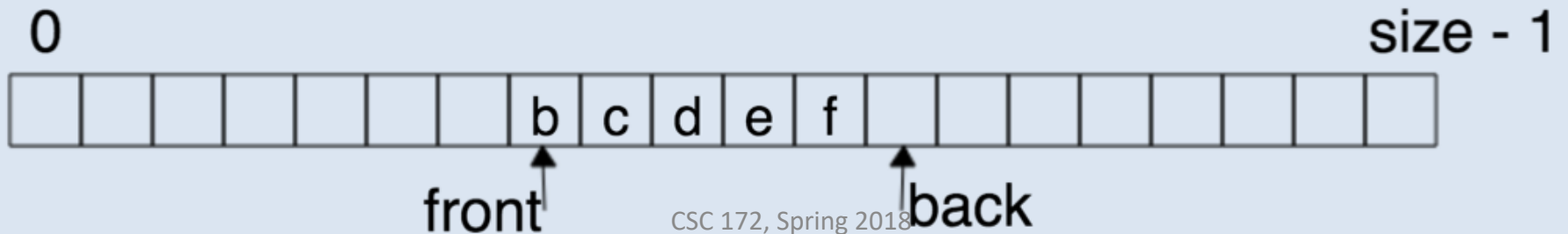
Arrays

	Front	Back/ n^{th} elem
add/offer	$O(n)$	$\Theta(1)$
remove/poll element/peek	$O(n)$	$\Theta(1)$
	$\Theta(1)$	$\Theta(1)$

The desired behavior of a Queue can't be achieved using an array. Unless....

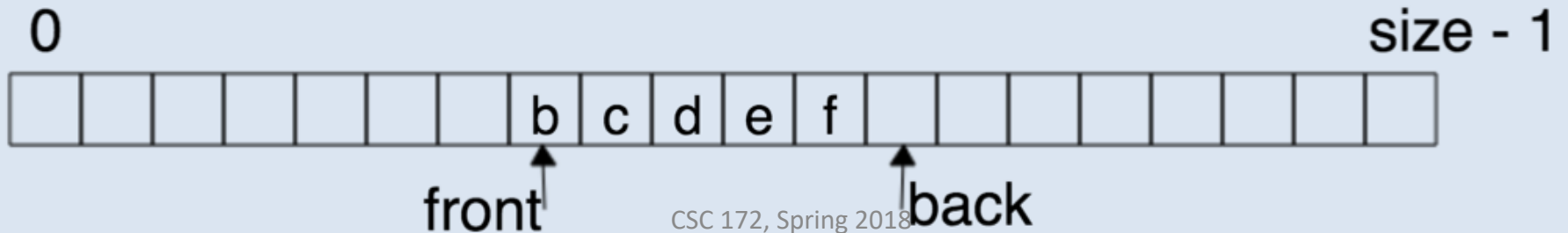
Circular Array Queue

	Front	Back/ n^{th} elem
add/offer		
remove/poll		
element/peek		



Circular Array Queue

	Front	Back/ n^{th} elem
add/offer	$\Theta(1)$	$\Theta(1)$
remove/poll	$\Theta(1)$	$\Theta(1)$
element/peek	$\Theta(1)$	$\Theta(1)$



Singly Linked List

	Front	Back/ n^{th} elem
add/offer remove/poll element/peek		

Queue is a FIFO data structure

The desired behavior of a Queue may be achieved by adding elements at the back and removing elements from the front.

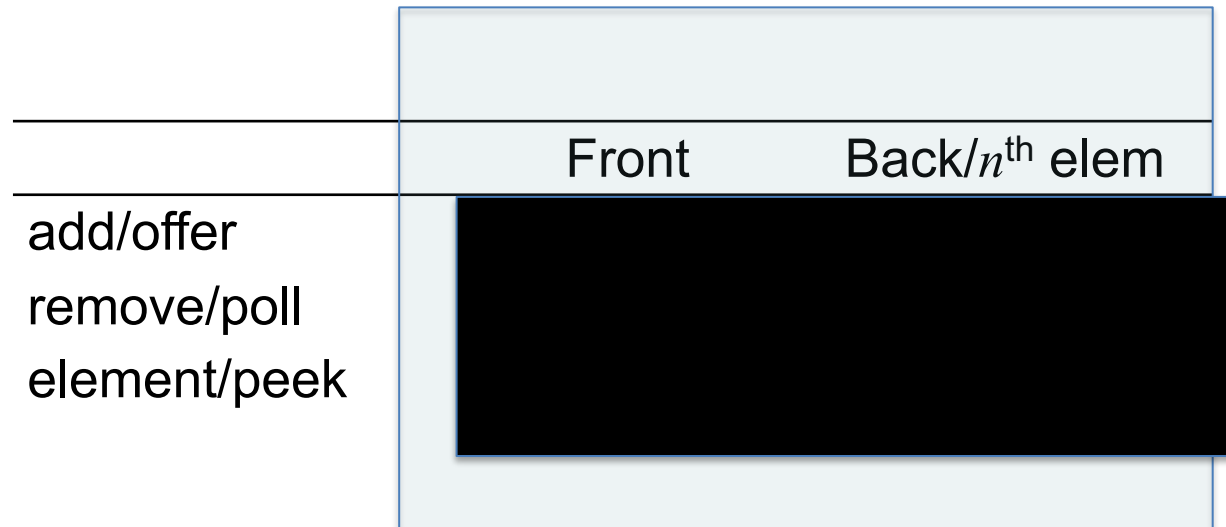
Singly Linked List

	Front	Back/ n^{th} elem
add/offer	$\Theta(1)$	$\Theta(1)$
remove/poll element/peek	$\Theta(1)$	$O(n)$

Queue is a FIFO data structure

The desired behavior of a Queue may be achieved by adding elements at the back and removing elements from the front.

Doubly Linked List



The desired behavior of a Queue may be reproduced by a doubly linked list perfectly.

Doubly Linked List

	Front	Back/ n^{th} elem
add/offer	$\Theta(1)$	$\Theta(1)$
remove/poll	$\Theta(1)$	$\Theta(1)$
element/peek	$\Theta(1)$	$\Theta(1)$

The desired behavior of a Queue may be reproduced by a doubly linked list perfectly.

Summary

- **Stack:**
 - Last-in-first-out (LIFO)
 - Usually uses an array to store the information
 - Optimal runtime $\theta(1)$ for all operations
- **Queue:**
 - First-in-first-out (FIFO)
 - Usually uses a linked list to store the information
 - Optimal runtime $\theta(1)$ for all operations

Acknowledgement

- Douglas Wilhelm Harder.
 - Thanks for making an excellent set of slides for ECE 250 *Algorithms and Data Structures* course
- Prof. Hung Q. Ngo:
 - Thanks for those beautiful slides created for CSC 250 (Data Structures) course at UB.
- Many of these slides are taken from these two sources.