# Ch 6 : Stacks, Queues, Deques

An abstract data type (ADT) is an abstraction of a data structure

# 6.1 Stacks

Stack: collection of objects that are inserted and removed according to the last-in, first-out (LIFO) principle.

*A user may insert objects into a stack at any time, but may only access or remove the most recently inserted object that remains*

Fundamental operations: “pushing” and “popping” of plates on the stack

When we need a new plate from the dispenser, we “pop” the top plate off the stack, and when we add a plate, we “push” it down on the stack to become the new top plate.

## 6.1.1 Stack Abstract Data Type

Stack ADT stores arbitrary objects

***2 update methods***:

* push(e): Adds element e to the top of the stack.
* pop( ): Removes and returns the top element from the stack (or null if the stack is empty).

***Accessor methods***:

* top( ): Returns the top element of the stack, without removing it (or null if the stack is empty).
* size( ): Returns the number of elements in the stack.
* isEmpty( ): Returns a boolean indicating whether the stack is empty.

***Stack in Java***: requires definition of class emptyStackException

In the Stack ADT, operations pop and top cannot be performed if the stack is empty , it will throw an ***EmptyStackException***.

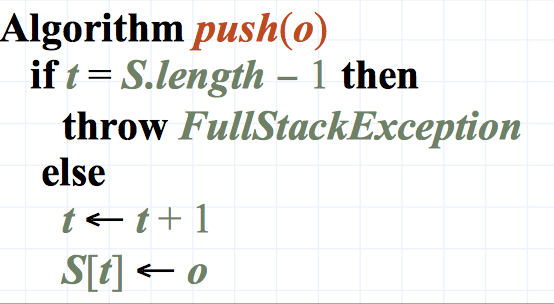
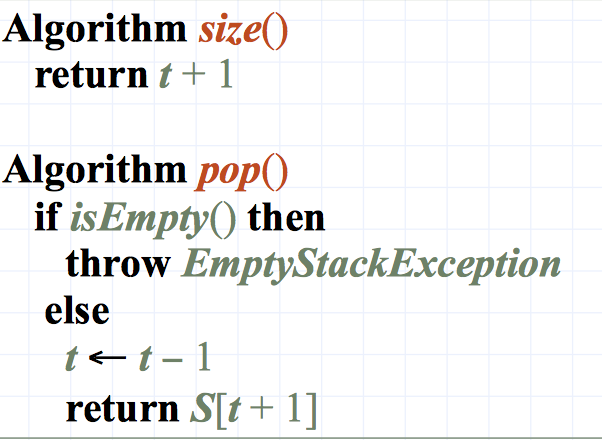
## 6.1.2 Simple Array-Based Stack

Simple way of implementing Stack ADT uses an array

Add element from left to right. A variable keeps track of index of top element.

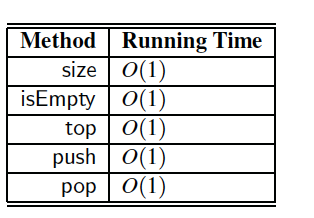


The array storing stack elements may become full

******= push operation will throw a ***FullStackException***

## 6.1.3 Performance and Limitations

Run time of each operation:



Space used depends of number of elements in the stack O(n)

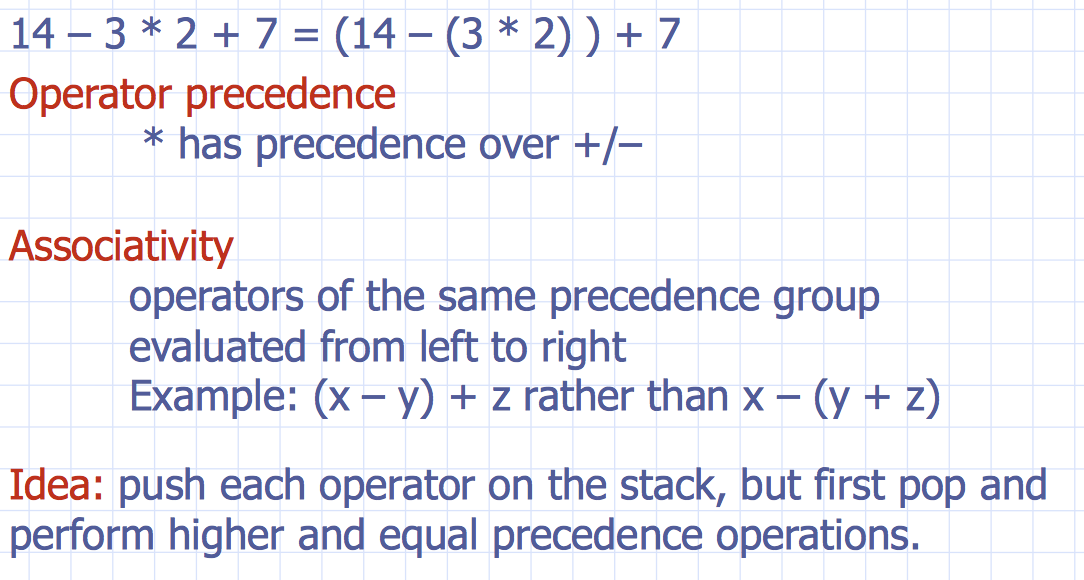
Limitations

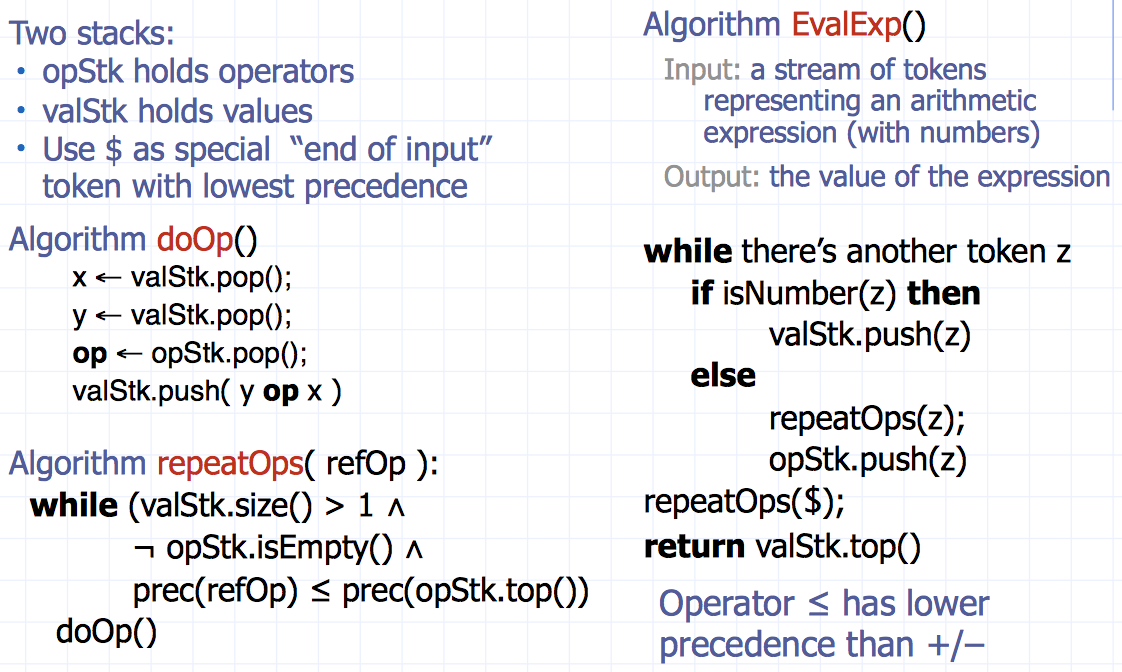
• The maximum size of the stack must be defined a priori, and cannot be changed

• Trying to push a new element into a full stack causes an implementation-specific exception

## review Parentheses Matching Algorithm, HTML Tag Matching,

## 6.1.4 Evaluating arithmetic expressions

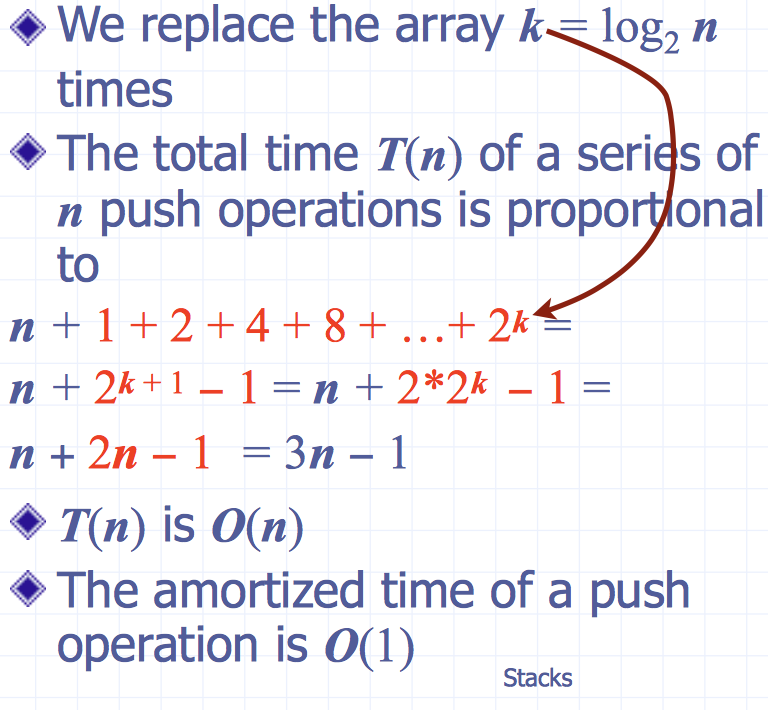




## Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-10-19 at 7.34.34 PM.pngreview First and Second Algorithm in slides

Growable Array-based stack: we can replace the array with larger one. Incremental strategy: increase the size by constant c or double size.

Compare both strategies with their total time T(n) by series of n operations**.**

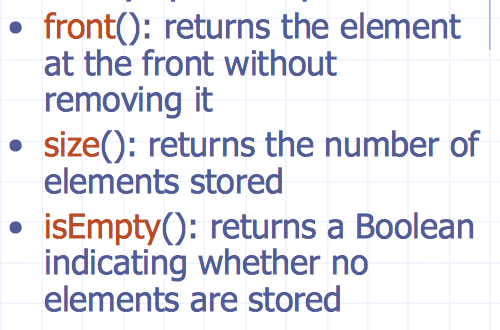
**Amortized time** of a push operation = average time taken by a push over the series of operations, T (n)/n

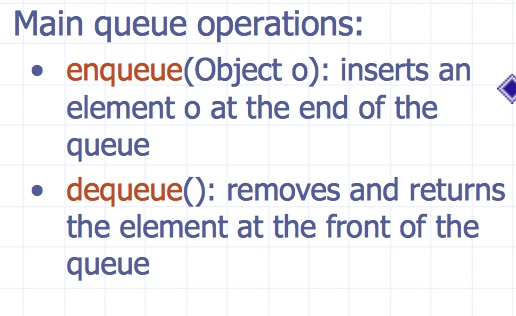
# 6.2 Queues

stores objects

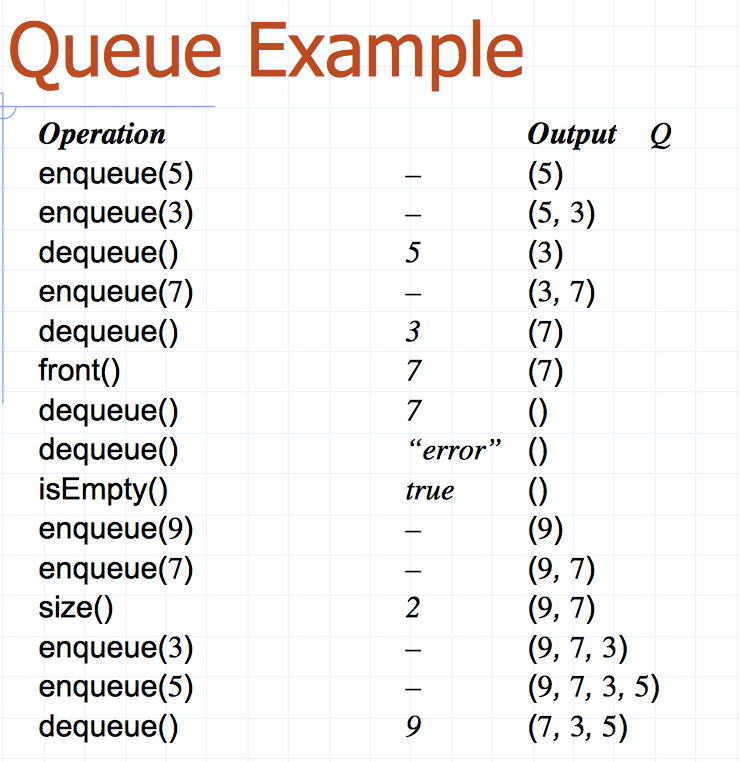
***first-in first-ou scheme***

***insertions***: at the end

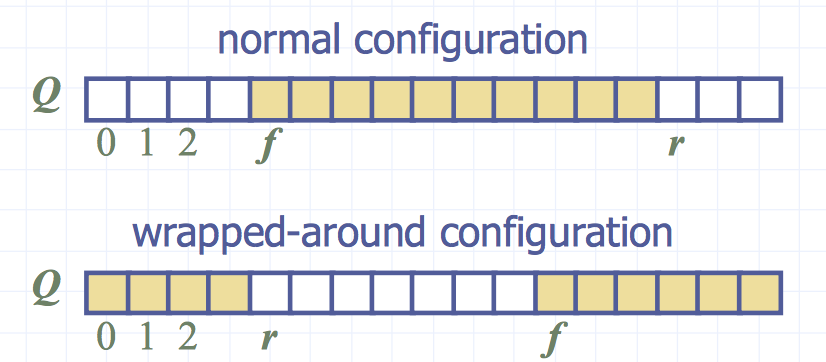
***removals***: front of queue



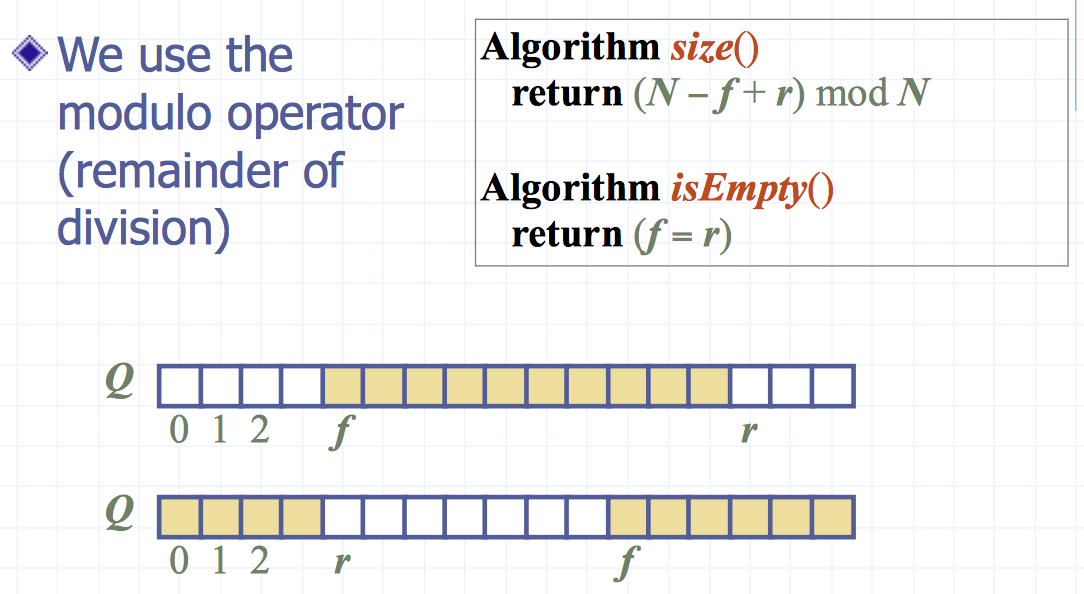
**Exceptions**: try execution of dequeue or front on an empty queue throws an ***EmptyQueueException***



## 6.2.1 Array-based queue

* Use an array of size N in a circular fashion
* Two variables keep track of the front and rear f index of the front element
* r index immediately past the rear element
* Array location r is kept empty

When you get at the end of the array but it is empty, what to do? => circular fashion, to get back at the beginning when done



The modulo operator is ideal for treating an array circularly. When we dequeue an element and want to “advance” the front index, we use the arithmetic f = ( f +1) % N.

## Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-10-20 at 6.56.35 PM.pngMacintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-10-20 at 6.56.30 PM.png6.2.2 Errors thrown

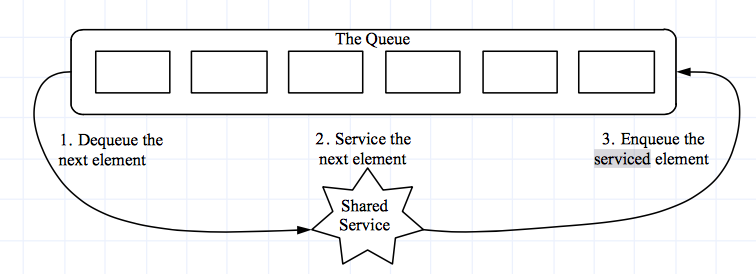
Queue interface in Java, need to define class EmptyQueueException

## 6.2.3 Round Robin Schedulers

can implement it using a queue, by by repeatedly performing the following steps:

1. e = Q.dequeue()

2. Service element e

3. Q.enqueue(e)

## 6.2.4 Growable array-based queue

In enqueue operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one

*Similar to what we did for an array-based stack*

The enqueue operation has amortized running time

• O(n) with the incremental strategy

• O(1) with the doubling strategy