

CSI2110

# Data Structures and Algorithms

Prof. WonSook Lee

[www.thewowimages.com](http://www.thewowimages.com)

# Overview

---

- Abstract Data Types
- Stack
- Queue
- Deque
  - double ended queue : pronounce "deck"

# Abstract Data Types (ADTs)

---

- An **Abstract Data Type** is an abstraction of a data structure.

The **ADT** specifies:

- what can be stored in the ADT
  - what operations can be done on/by the ADT
- 
- For example, if we are going to model a bag of marbles as an ADT, we could specify that:
    - this ADT stores marbles
    - this ADT supports putting in a marble and getting out a marble.

# Abstract Data Types (ADTs)

---

- Specify precisely the operations that can be performed
- The implementation is HIDDEN and can easily change

---

## EXAMPLES

---

Objects of type: Phone Book

Operations: find, add, remove

...

# Abstract Data Types (ADTs)

---

- There are lots of formalized and standard ADTs.
- In this course we are going to learn a lot of different standard ADTs. (**stacks**, queues, dictionary...)



# Stacks, Queues, and Deques

---

## ADT Stack

- Implementation with Arrays

- Implementation with Singly Linked List

## ADT Queue

- Implementation with Arrays

- Implementation with Singly Linked List

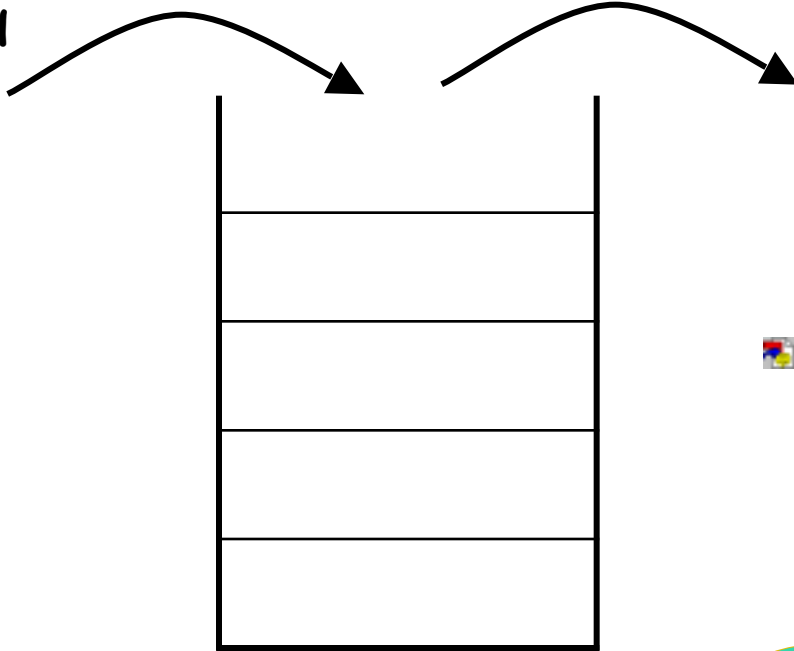
## ADT Double Ended Queues

- Implementation with doubly Linked List

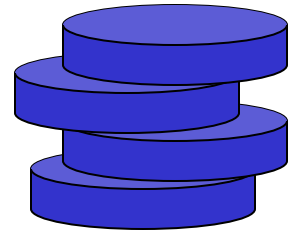
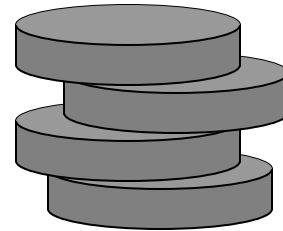
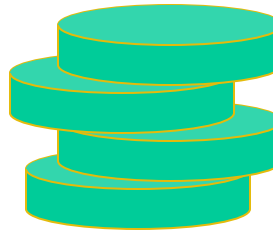
# Stacks

---

PUSH



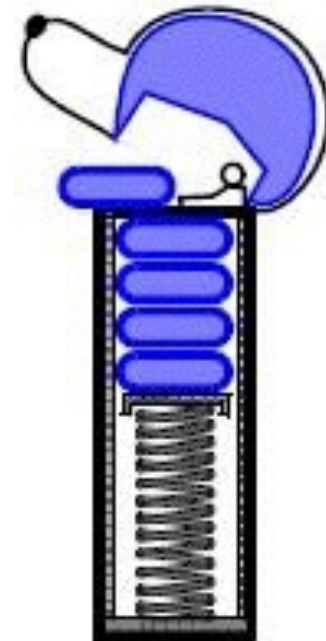
POP



# The Stacks

---

- A stack is a container for inserting and removing objects according to the principle, the last entered element, first go out. (last-in-first-out, or LIFO)
- The objects can be inserted all together, but only the last element (the most recently inserted) can be removed.
- Analogue: PEZ<sup>®</sup> dispenser





# The Stack Abstract Data Type

---

- Main methods:
  - `push(o)`: Inserts object `o` onto top of stack
  - `pop()`: Removes the top object of stack and returns it;  
if the stack is empty, an error occurs
- Support methods:
  - `size()`: Returns the number of objects in stack
  - `isEmpty()`: Return a boolean indicating if stack is empty.
  - `top()`: Return the top object of the stack, without removing it; if the stack is empty, an error occurs.

# Applications of Stacks

---

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

# Examples

---

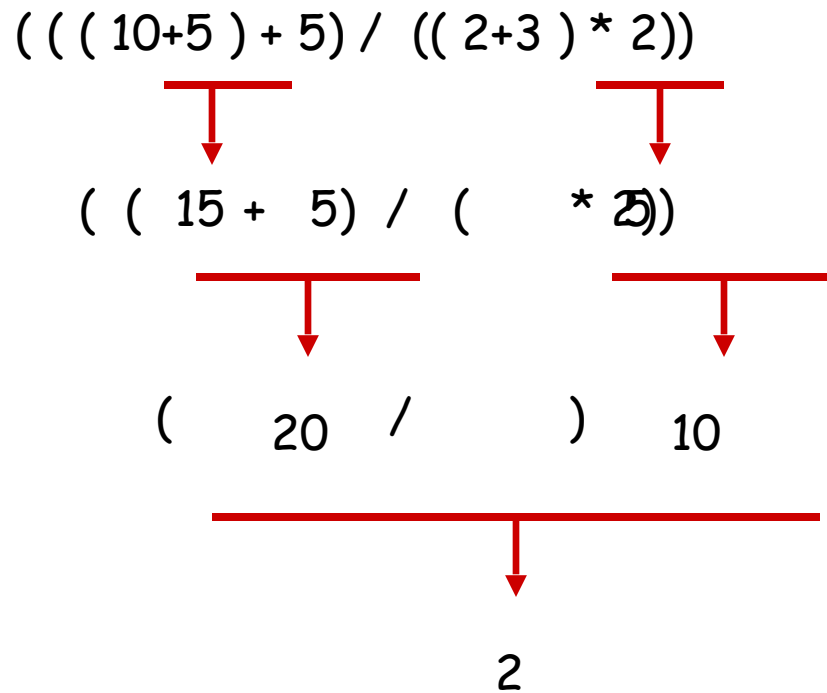
Evaluating an expression with two stacks

$$(( (10+5) + 5) / ( (2+3) * 2))$$

How do we solve it ?

# One possible sequence of operations

---



## Another one

---

$(( (10+5) + 5) / ((2+3) * 2))$



$(( 15 + 5) / ((2+3) * 2))$



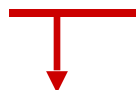
$( 20 / ((2+3) * 2))$



$( 20 / ( 5 * 2))$



$( 20 / ) 10$



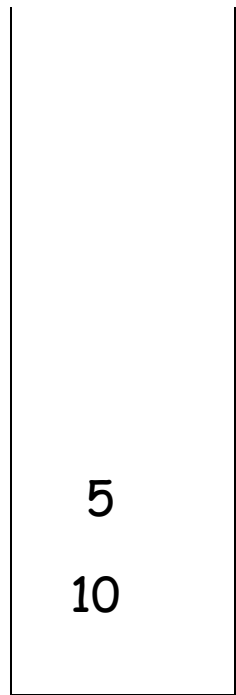
2

# With two Stacks

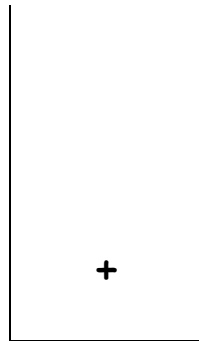
---

One for the operands

One for the operators



S1



S2

$((10+5) + 5) / ((2+3) * 2)$

# With two Stacks

---

One for the operands  
One for the operators

when find CLOSED parenthesis

$((15 + 5) / ((2 + 3) * 2))$

$(( (10 + 5) + 5) / ((2 + 3) * 2))$



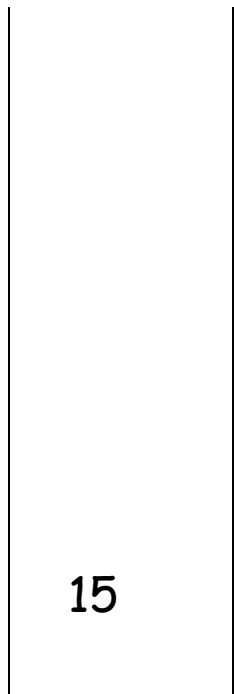
POP S1      5

POP S2      +

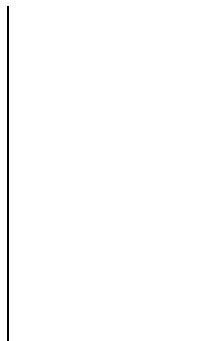
POP S1      10

Evaluate:  $10 + 5 = 15$

PUSH S1 the result



S1



S2

# With two Stacks

---

One for the operands  
One for the operators

CLOSED parenthesis

( 20 / (( 2+3) \* 2))

(( (10+5) + 5) / (( 2+3) \* 2))



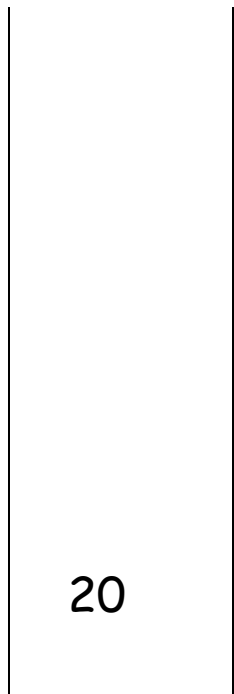
POP S1      5

POP S2      +

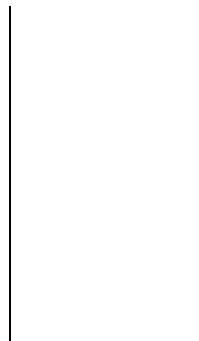
POP S1      15

Evaluate:  $15 + 5 = 20$

PUSH S1 the result



S1



S2

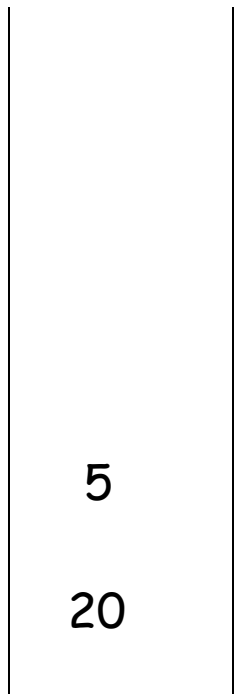


# With two Stacks

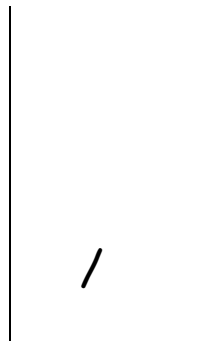
---

One for the operands

One for the operators



S1



S2

CLOSED parenthesis

( 20 / (( 2+3) \* 2))

(( (10+5) + 5) / (( 2+3) \* 2))



POP S1 3

POP S2 +

POP S1 2

Evaluate  $2 + 3 = 5$

PUSH S1 the result

# With two Stacks

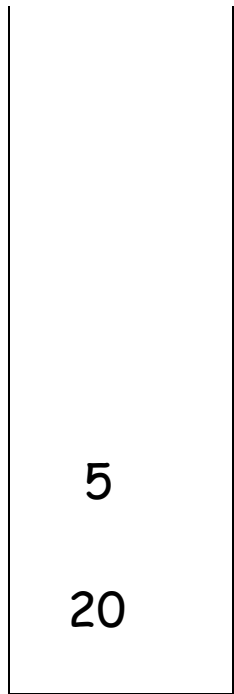
---

One for the operands

One for the operators

( 20 / ( 5 \* 2))

(( (10+5) + 5) / ((2+3) \* 2))



S1



S2

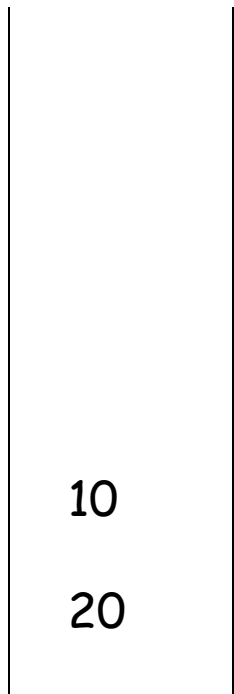
# With two Stacks

One for the operands  
One for the operators

CLOSED parenthesis

( 20 / ( 5 \* 2 ) )

(( (10+5) + 5) / ((2+3) \* 2))



S1



S2

POP S1      2

POP S2      \*

POP S1      5

Evaluate:  $5 * 2 = 10$

PUSH S1      result

# With two Stacks

---

One for the operands  
One for the operators

CLOSED parenthesis

( 20 / 10 )

(( (10+5) + 5) / ((2+3) \* 2))



10

/

20

S1

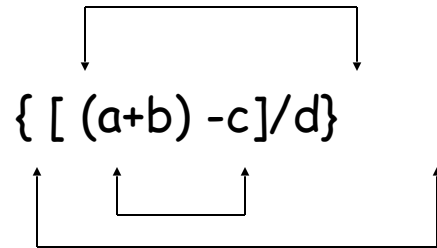
S2

20 / 10 = 2

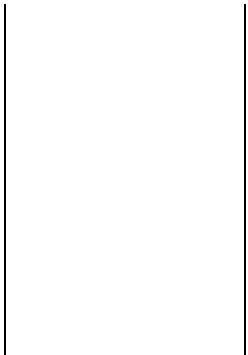
# Examples

---

Checking for balanced parenthesis



$\{ [ a+b) -c]/d \}$



# Example: Evaluation of arithmetic expressions (Postfix notation)

**Infix**

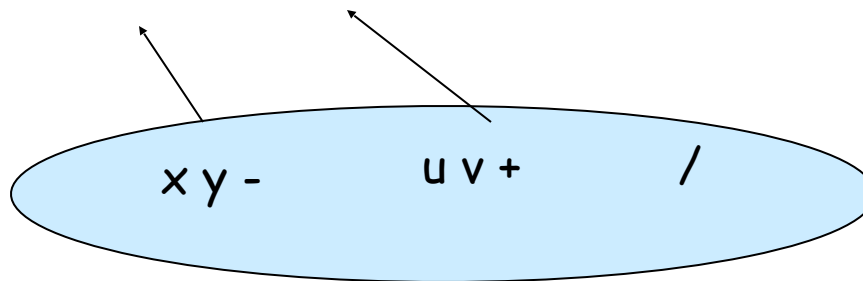
**A + B**

**Postfix**

**A B +**

$(x - y)/(u + v)$

$(x - y) (u + v) /$



INFIX

$a+b$

$(x - y + z)$

$(x - y - z)/(u + v)$

POSTFIX

$a b +$

$x y - z +$

$x y - z - u v + /$

In general:

A operator B

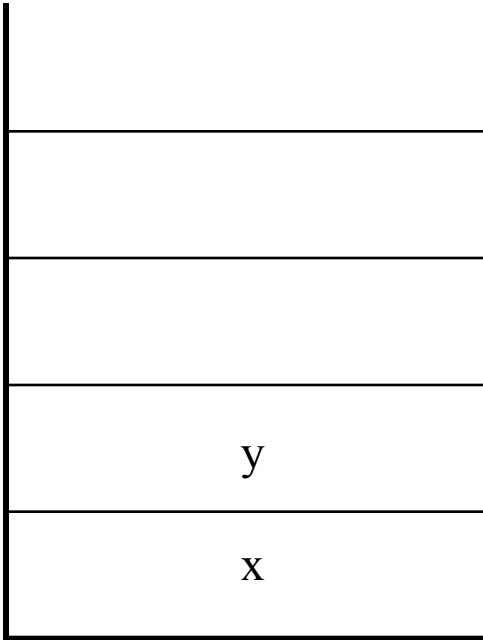
A B operator



$x \ y - z - u \ v + /$

**x:** push(x)

**y:** push(y)



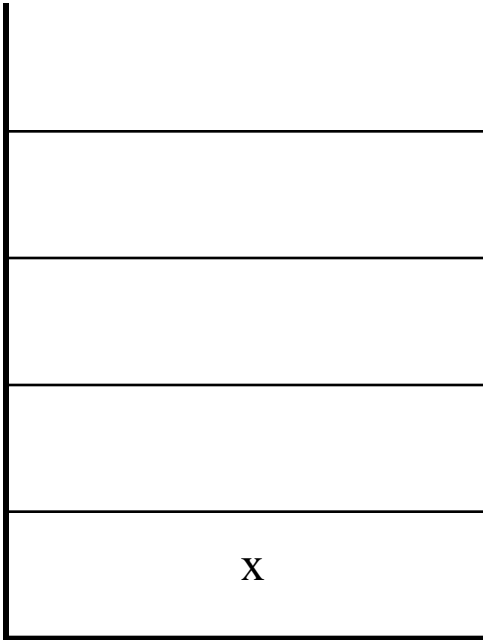


$x \ y - z - u \ v + /$

**x:** push(x)

**y:** push(y)

**-** : pop() (we get y)



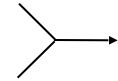
$x \ y - z - u \ v + /$

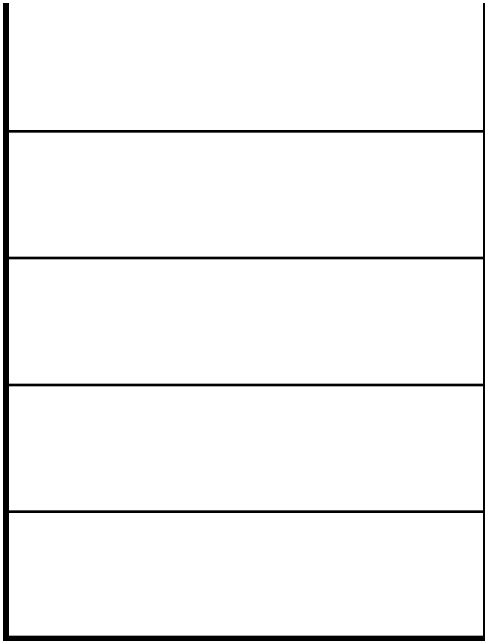
**x:** push(x)

**y:** push(y)

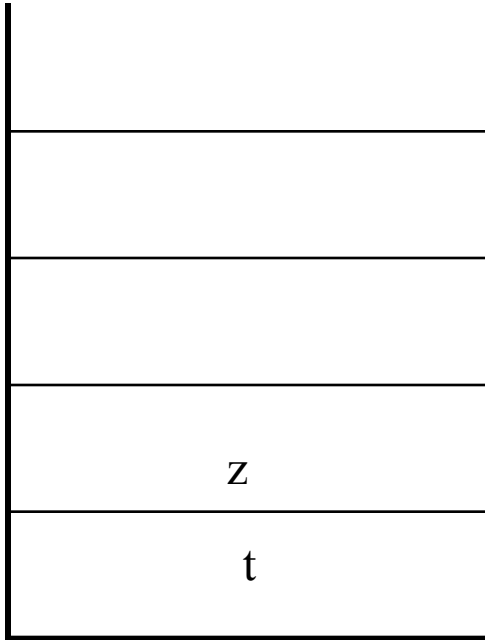
**-** : pop() (we get y)

pop() (we get x)

  $x - y$  = t



$x \ y - z - u \ v + /$



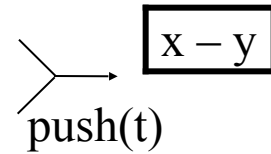
**x:** push(x)

**y:** push(y)

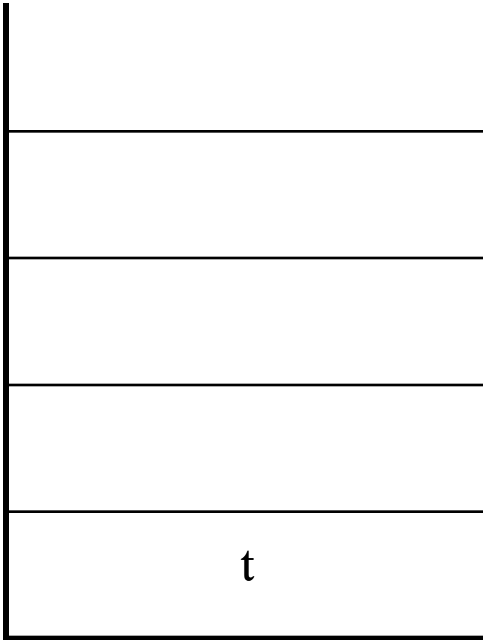
**-** : pop() (we get y)

pop() (we get x)

**z:** push(z)

  $x - y = t$

$x \ y - z - u \ v + /$



**x:** push(x)

**y:** push(y)

**-** : pop() (we get y)

pop() (we get x)

**z:** push(z)

**-** : pop() (we get z)

push(t)  $x - y$  = t

$x \ y - z - u \ v + /$

**x:** push(x)

**y:** push(y)

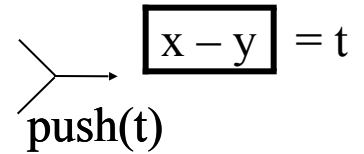
**-** : pop() (we get y)

pop() (we get x)

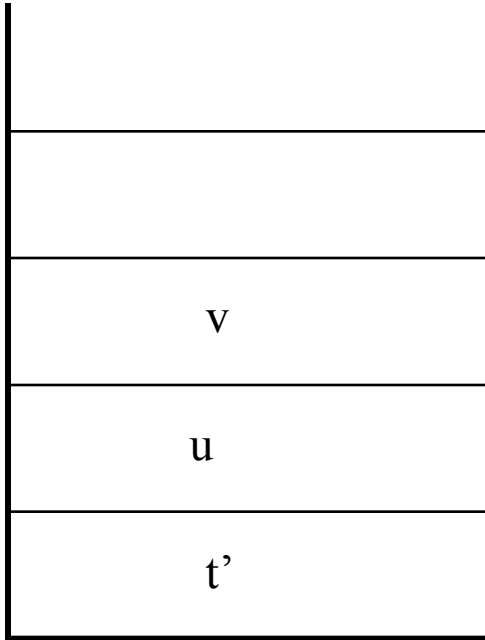
**z:** push(z)

**-** : pop() (we get z)

pop() (we get t)

  $x - y = t$   
push(t)

$x \ y - z - u \ v + /$



**x:** push(x)

**y:** push(y)

**-** : pop() (we get y)

pop() (we get x)

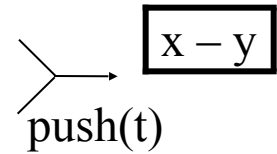
**z:** push(z)

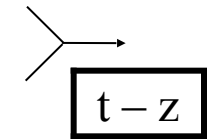
**-** : pop() (we get z)

pop() (we get t)

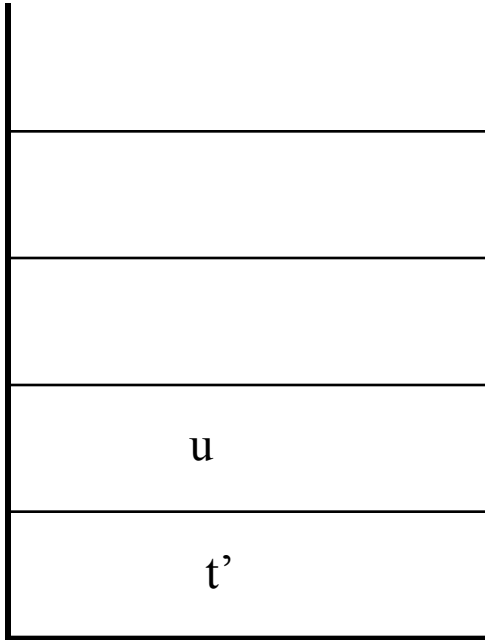
**u:** push(u)

**v:** push(v)

  $x - y = t$   
push(t)

  $t - z = t'$   
push( $t'$ )

$x \ y - z - u \ v + /$



**x:** push(x)

**y:** push(y)

**-** : pop() (we get y)

pop() (we get x)

**z:** push(z)

**-** : pop() (we get z)

pop() (we get t)

**u:** push(u)

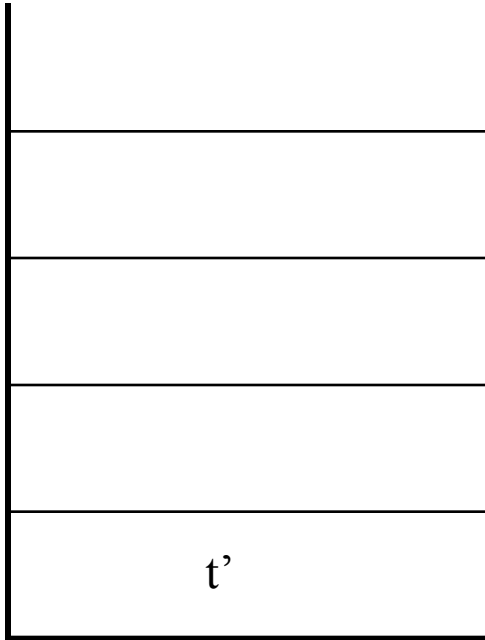
**v:** push(v)

**+** : pop() (we get v)

$\boxed{x - y} = t$   
push(t)

$\boxed{t - z} = t'$   
push( $t'$ )

$x \ y - z - u \ v + /$



**x:** push(x)

**y:** push(y)

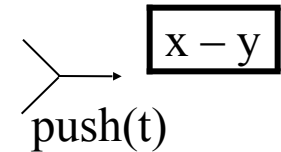
**- :** pop() (we get y)

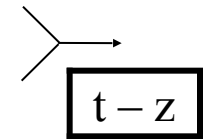
pop() (we get x)

**z:** push(z)

**- :** pop() (we get z)

pop() (we get t)

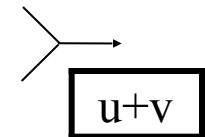
  $x - y = t$   
push(t)

  $t - z = t'$   
push( $t'$ )

**v:** push(v)

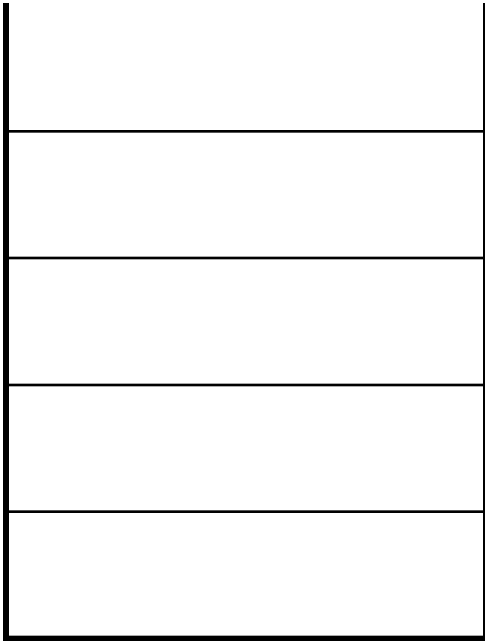
**+** : pop() (we get v)

pop() (we get u)

  $u + v = t''$



$x \ y - z - u \ v + /$



**x:** push(x)

**y:** push(y)

**-** : pop() (we get y)

pop() (we get x)

**z:** push(z)

**-** : pop() (we get z)

pop() (we get t)

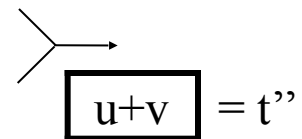
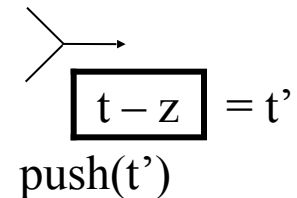
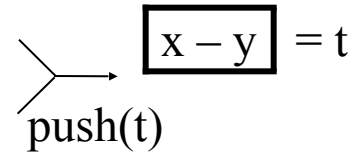
**v:** push(v)

**+** : pop() (we get v)

pop() (we get u)

**/** : pop() (we get t')

Result = t'/t''



# Implementing a Stack with an Array

---

The stack consists of an  $N$ -element array  $S$  and an integer variable  $t$ , the index of the top element in array  $S$ .



**Algorithm** size():

return  $t + 1$

**Algorithm** isEmpty():

return  $(t < 0)$

**Algorithm** top():

if isEmpty() then

**ERROR**

return  $S[t]$

---

```
Algorithm push(obj):  
    if size() = N then  
        ERROR  
     $t \leftarrow t + 1$   
     $S[t] \leftarrow \text{obj}$ 
```

```
Algorithm pop():  
    if isEmpty() then  
        ERROR  
     $e \leftarrow S[t]$   
     $S[t] \leftarrow \text{null}$   
     $t \leftarrow t - 1$   
    return  $e$ 
```



# Performance and Limitations

---

- Performance

Time	
size()	$O(1)$
isempty()	$O(1)$
top()	$O(1)$
push(obj)	$O(1)$
pop()	$O(1)$

Space:  $O(n)$

$n$  = size of  
the Array

- Limitations

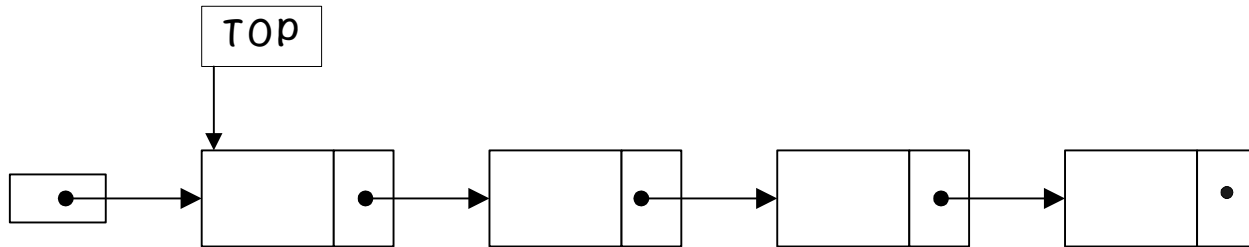
**STATIC STRUCTURE**

What does it mean  
 $O(1)$  ?

we will see the  
formal definition  
next lecture

# Implementing a Stack with a Singly Linked List

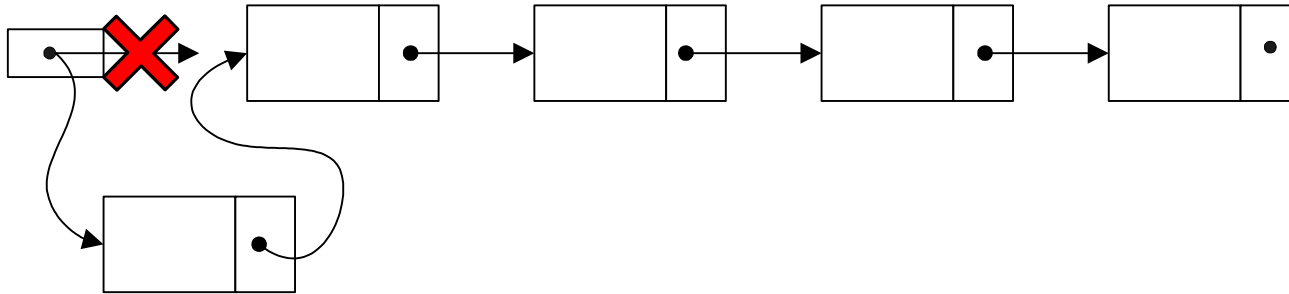
---



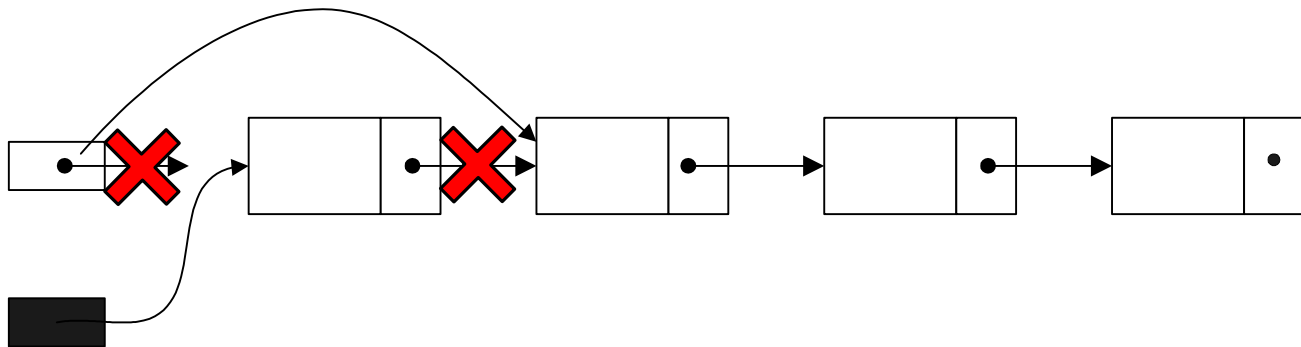
-Singly linked list plus a variable containing the current size of the list

**DYNAMIC STRUCTURE**

**PUSH:** Add at the front

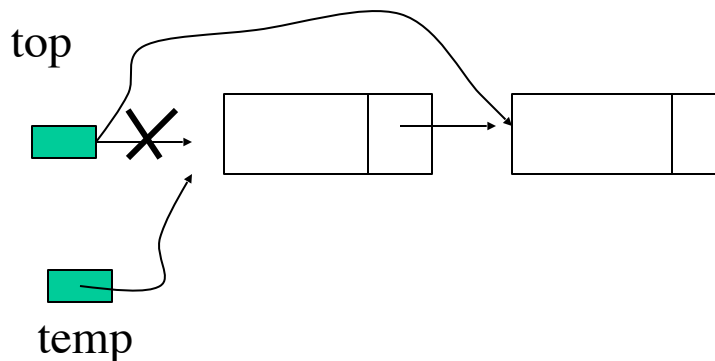
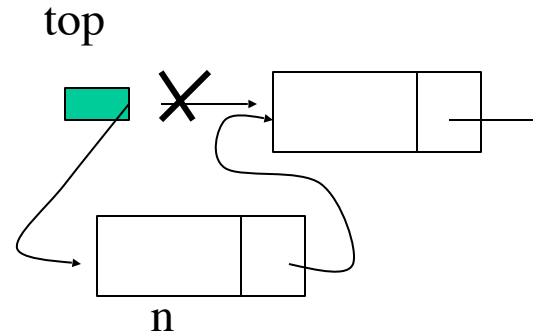


**POP:** Take the first



node:  
node.item  
node.next

```
Algorithm push(obj):  
  n ← new Node  
  n.item ← obj  
  n.next ← top  
  top ← n  
  size++
```



```
Algorithm pop():  
  if isEmpty() then  
    ERROR  
  temp ← top.item  
  top ← top.next  
  size--  
  return temp
```

# Performance

---

Time:

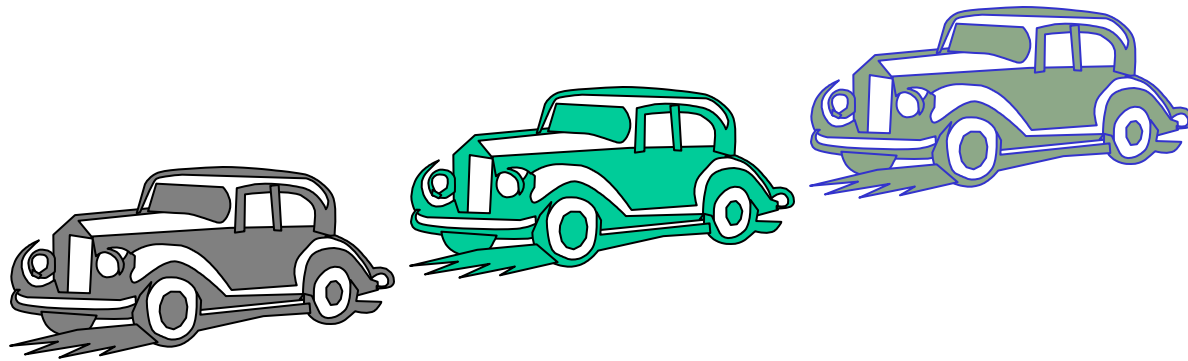
size()	$O(1)$
isEmpty()	$O(1)$
top()	$O(1)$
push(obj)	$O(1)$
pop()	$O(1)$

Space: Variable



# The Queue

---



# The Queue

---



first-in-first-out (FIFO)



- A queue is different from a stack in the way how the insertion and removal work.
- The elements can be inserted all together, but only the element which stayed longest time in the queue can be out.
- Elements are inserted at the **rear** (**enqueued**) and removed from the **front** (**dequeued**)

# Applications of Queues

---

- Direct applications
  - Waiting lists, bureaucracy
  - Access to shared resources (e.g., printer)
  - Multiprogramming
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

# Fun Example: Palindromes

---

"eye"

"madam"

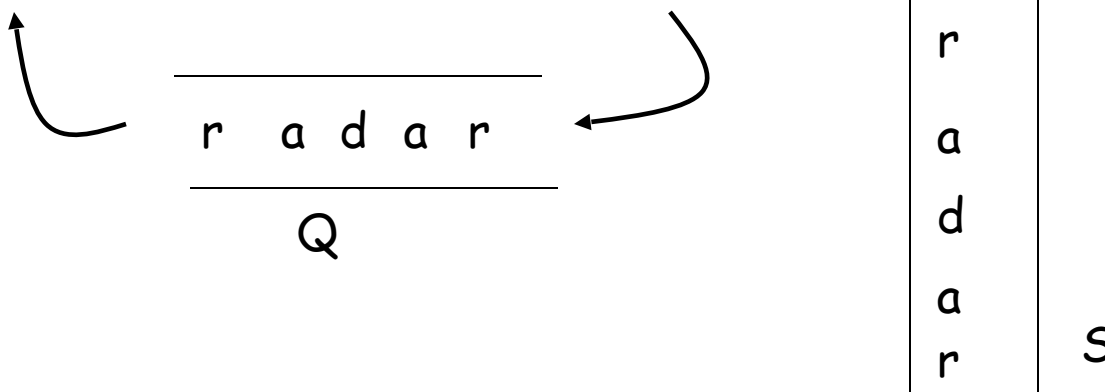
"radar"

"madam i'm adam"

"Able was I ere I saw Elba"

Read the line into a stack and into a queue

Compare the outputs of the queue and the stack



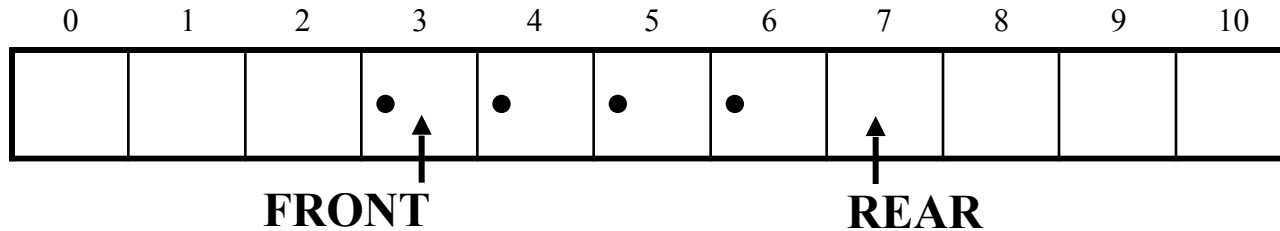
# The Queue Abstract Data Type

---

- Fundamental methods:
  - enqueue(o):** Insert object o at the rear of the queue
  - dequeue():** Remove the object from the front of the queue and return it; **an error occurs if the queue is empty**
- Support methods:
  - size():** Return the number of objects in the queue
  - isEmpty():** Return a boolean value that indicates whether the queue is empty
  - front():** Return, but do not remove, the front object in the queue; **an error occurs if the queue is empty**

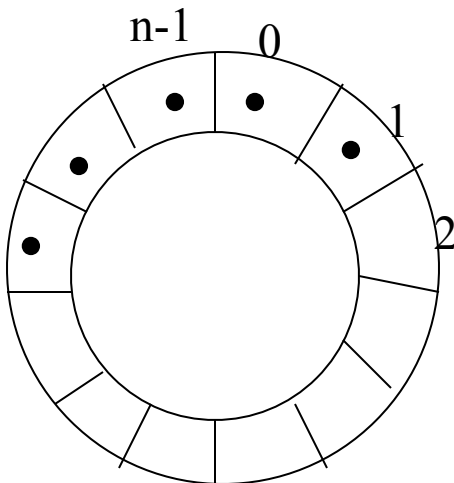
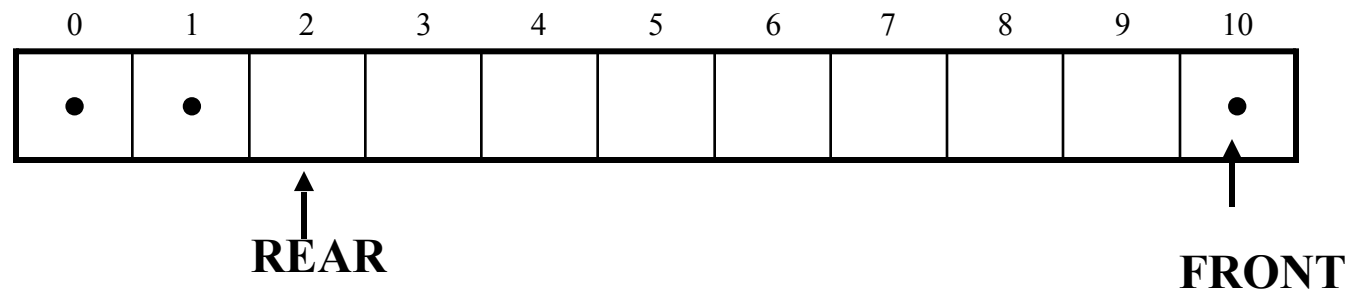
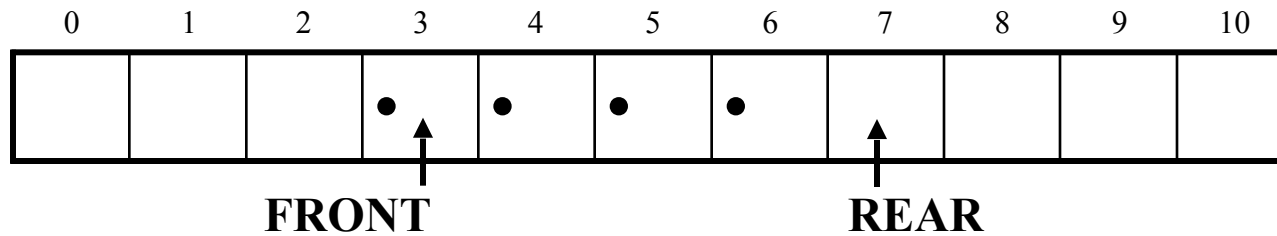
# Implementing a Queue with an Array

---



Insert at the **rear** and remove from **front**

# Implementing a Queue with an Array



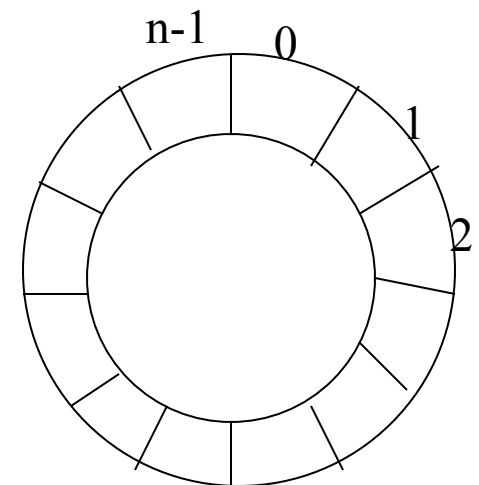
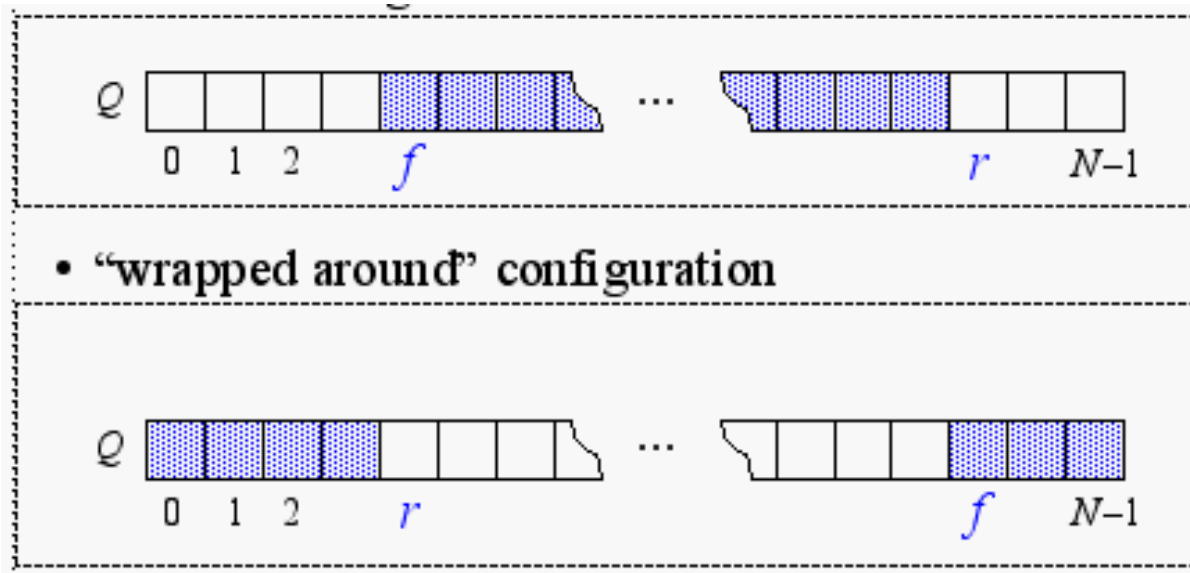
Remove:  $\text{Front} = (\text{Front} + 1)$

**mod n**

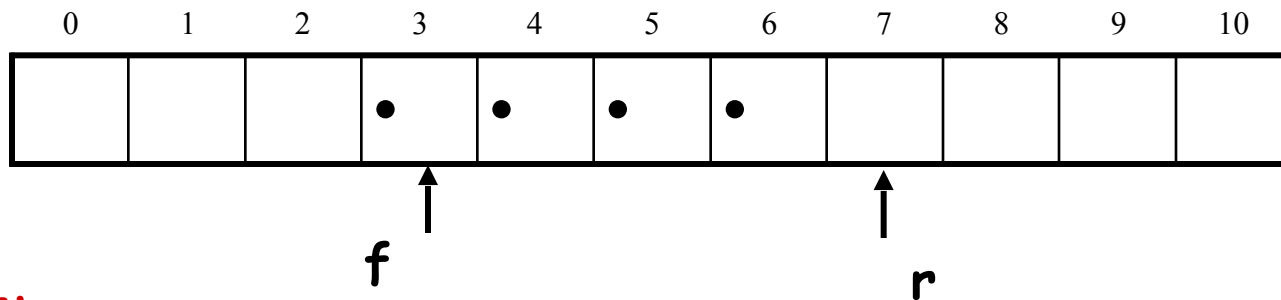
Insert:  $\text{Rear} = (\text{Rear} + 1)$

**mod n**

- Array in a circular fashion ( "wrapped around" )
- Size fixed at the beginning
- The queue consists of an N-element array Q and two integer variables:
  - f, index of the front element
  - r, index of the element after the rear one





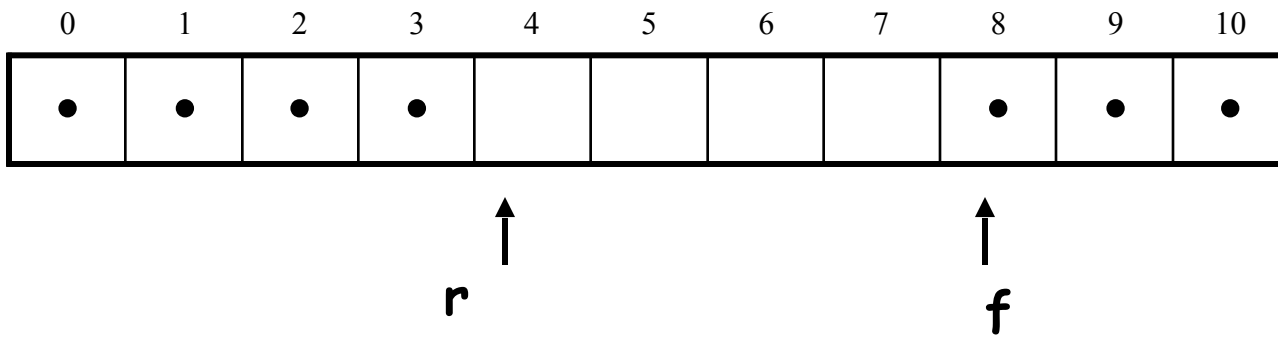


Questions:

What does  $f = r$  mean?

The queue is empty

How do we compute the number of elements in the queue from  $f$  and  $r$ ?



$$(N - f + r) \bmod N$$

In the example:

$$(11 - 8 + 4) \bmod 11 = 7$$

Algorithm **size()**:  
    return  $(N - f + r) \bmod N$

Algorithm **isEmpty()**:  
    return  $(f = r)$

Algorithm **front()**:  
    if isEmpty() then  
        ERROR  
    return  $Q[f]$

Algorithm **dequeue()**:  
    if isEmpty() then  
        ERROR  
    temp  $\leftarrow Q[f]$   
     $Q[f] \leftarrow \text{null}$   
     $f \leftarrow (f + 1) \bmod N$   
    return temp

Algorithm **enqueue(o)**:  
    if size =  $N - 1$  then  
        ERROR  
     $Q[r] \leftarrow o$

# Performance

---

Time:

size()	$O(1)$
isEmpty()	$O(1)$
front()	$O(1)$
enqueue(o)	$O(1)$
dequeue()	$O(1)$

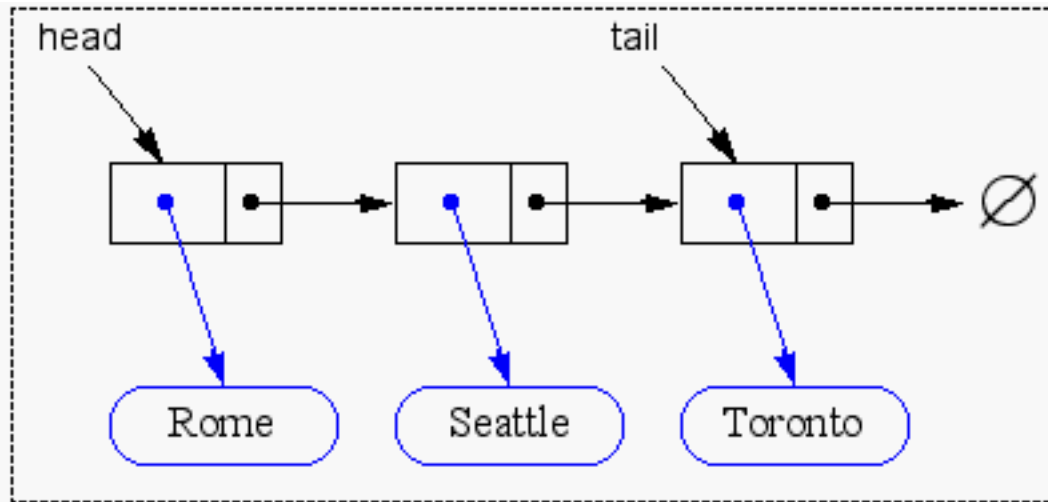
Space:  $O(N)$

# Implementing a Queue with a Singly Linked List

---

Nodes connected in singly linked list

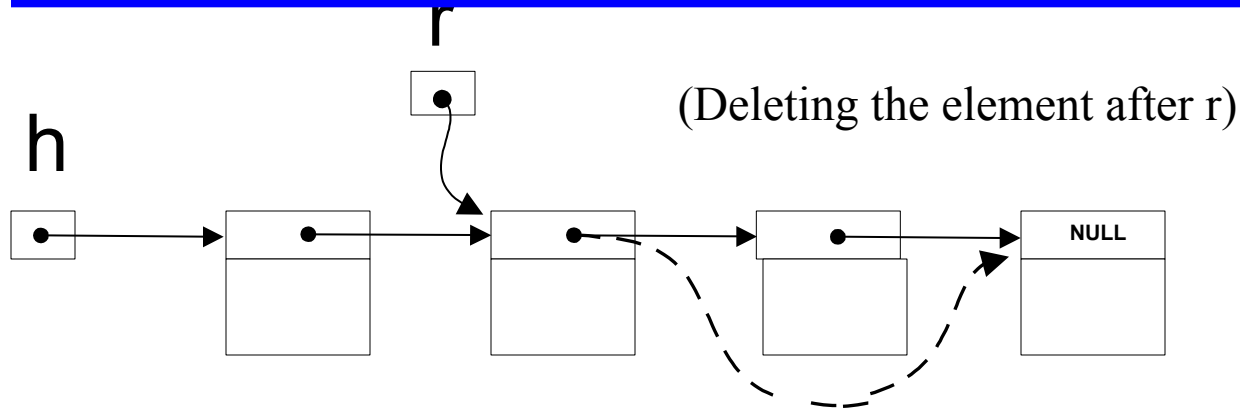
We keep a pointer to the head and one to the tail



The head of the list is the front of the queue, the tail of the list is the rear of the queue.

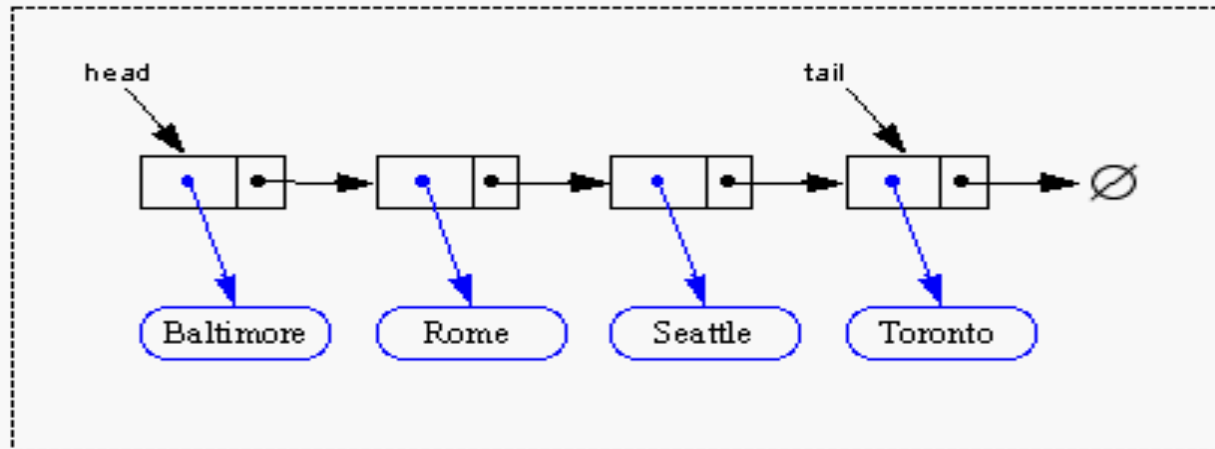
**Why not the opposite?**

# From the last lesson : Deletion

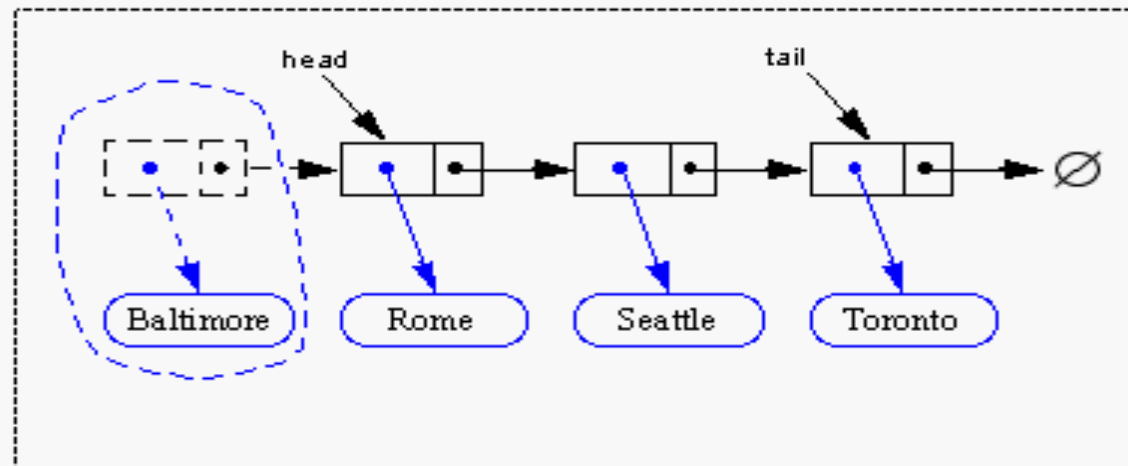


First element (easy)	$h \leftarrow h.Next$
Element after r (easy)	$r.Next \leftarrow r.Next.Next$
Element at r (difficult)	<ul style="list-style-type: none"><li>• Use a pointer to the preceding element, or</li><li>• Exchange the contents of the element at r with the contents of the element following r, and delete The element after r. **Very difficult if r points to the last element!</li></ul>

# Removing at the Head



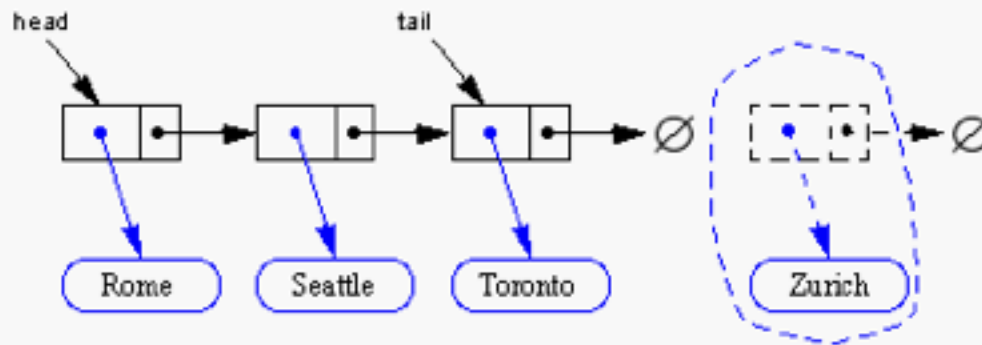
- advance head reference



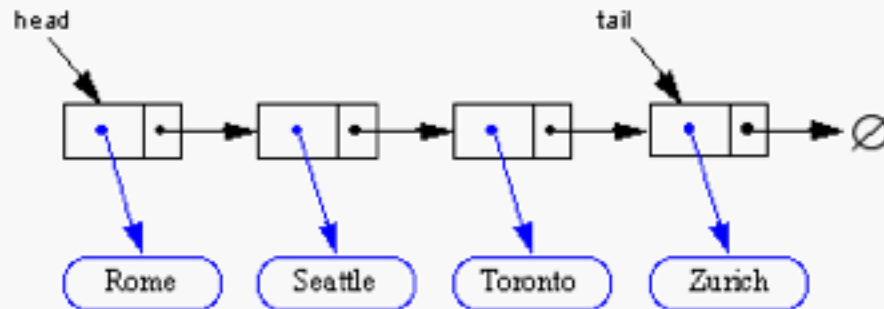
- inserting at the head is just as easy

# Inserting at the Tail

- create a new node



- chain it and move the tail reference



- how about removing at the tail?



# Performance

---

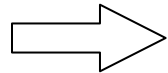
Time:

size()	$O(1)$
isEmpty()	$O(1)$
front()	$O(1)$
enqueue(o)	$O(1)$
dequeue()	$O(1)$

Space: Variable

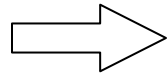
---

If we know in advance a reasonable upper bound for the number of elements in the queue, then



ARRAYS

Otherwise



LISTS

# A more general ADT: Double-Ended Queues (Deque)

---

A **double-ended** queue, or **deque**, supports insertion and deletion from the front and back.

Main methods:

<code>insertFirst(e):</code>	Insert e at the beginning of deque.
<code>insertLast(e):</code>	Insert e at end of deque
<code>removeFirst():</code>	Removes and returns first element
<code>removeLast():</code>	Removes and returns last element

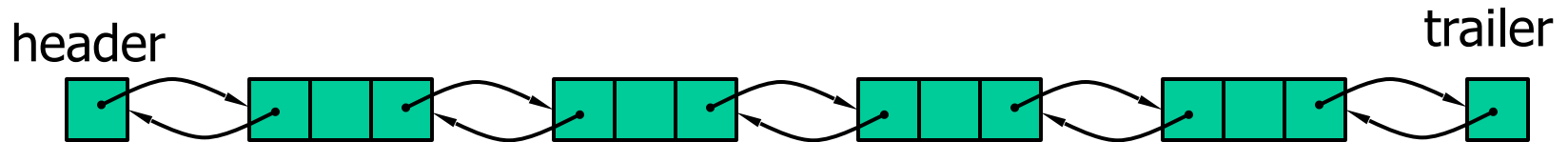
Support methods:

- `first()`
- `last()`
- `size()`
- `isEmpty()`

# Implementing Deques with Doubly Linked Lists

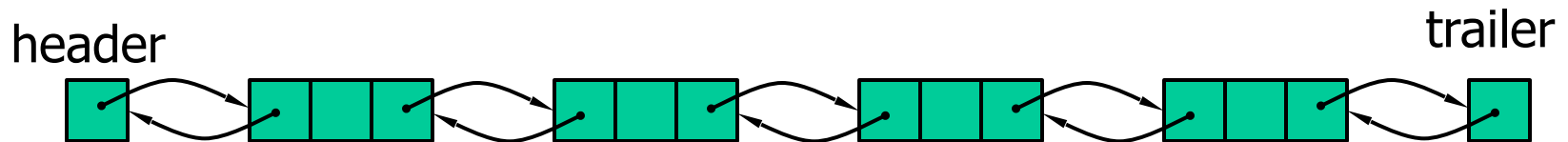
---

Deletions at the tail of a singly linked list cannot be done efficiently



To implement a deque, we use a **doubly linked** list with special header and trailer nodes

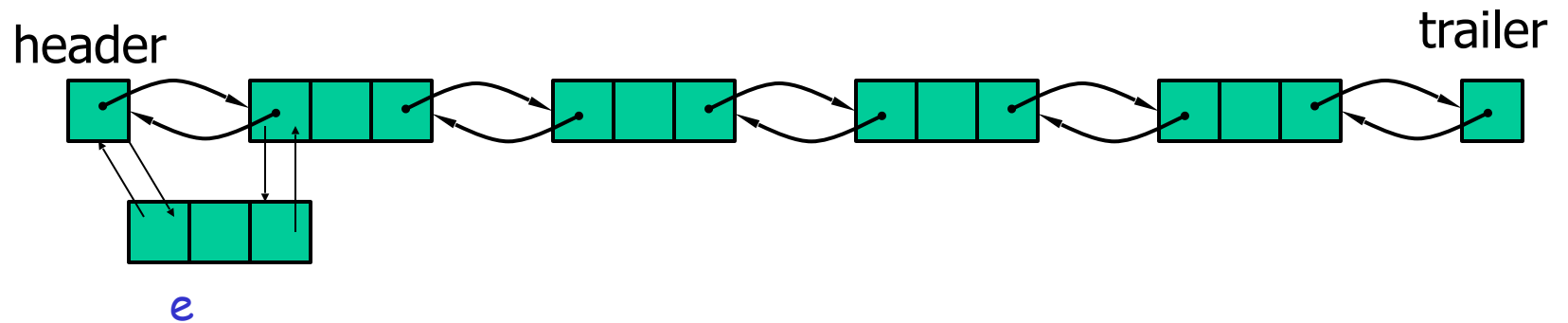
- 
- The **header** node goes before the first list element. It has a valid next link but a null prev link.
  - The **trailer** node goes after the last element. It has a valid prev reference but a null next reference.



NOTE: the header and trailer nodes are sentinel or "dummy" nodes because they do not store elements.

insertFirst(e):

---



$e.next \leftarrow header.next$

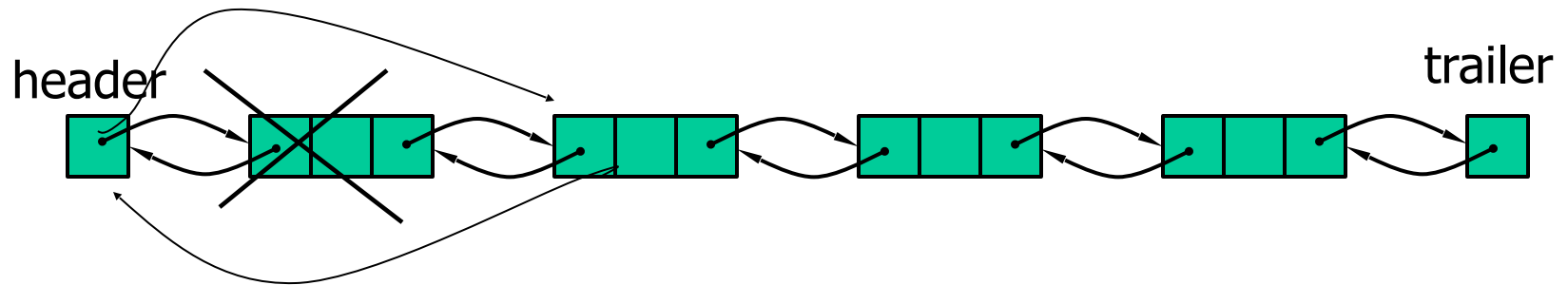
$header.next \leftarrow e$

$e.prev \leftarrow header$

$e.next.prev \leftarrow e$

removeFirst():

---



`header.next.next.prev ← header`

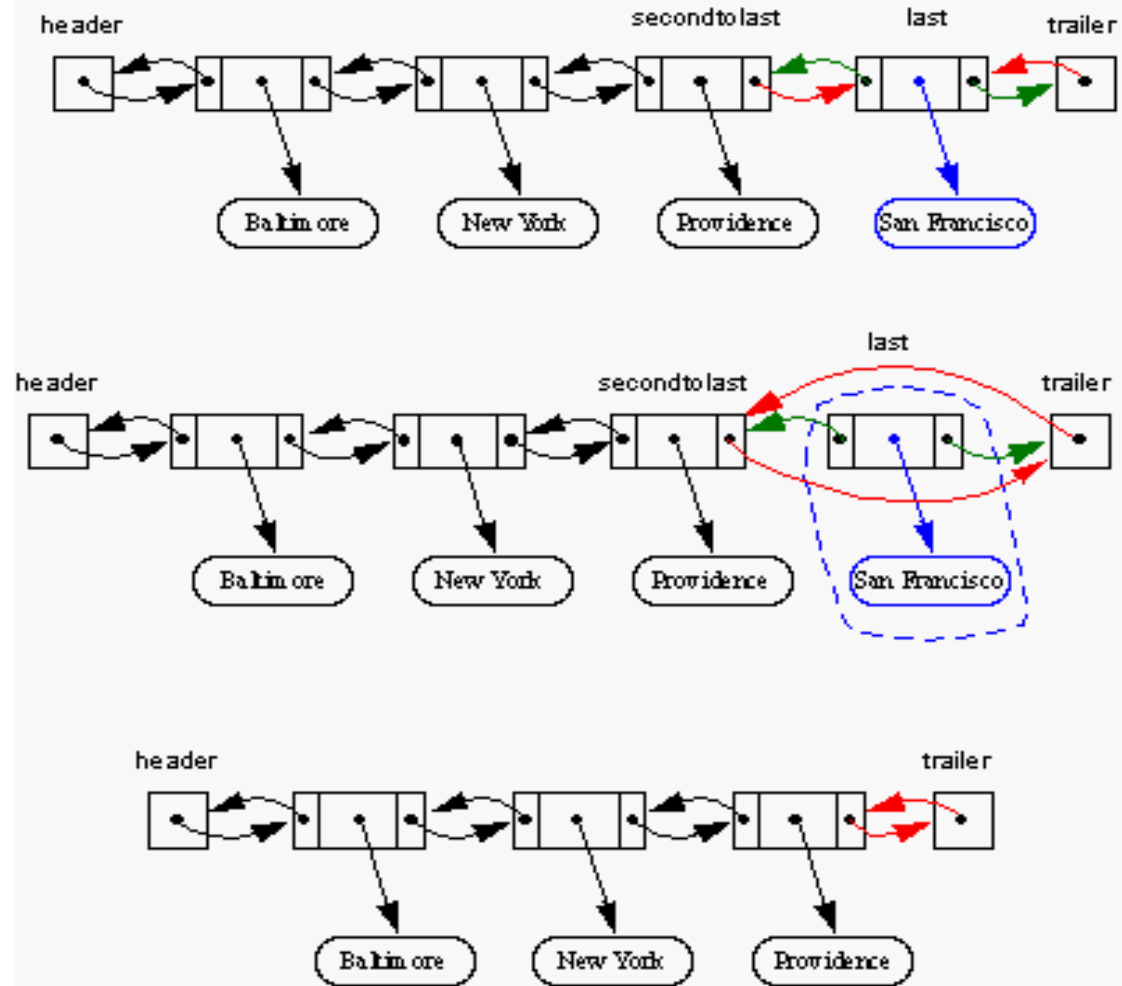
`header.next ← header.next.next`

OR:

`header.next ← header.next.next`

`header.next.prev ← header`

Here's a visualization of the code for `removeLast()`.



With this implementation, all methods have constant execution time ( complexity  $O(1)$ )



# Implementing Stacks and Queues with Deques

---

## Stacks with Deques:

Stack Method	Deque Implementation
size()	size()
isEmpty()	isEmpty()
top()	last()
push(e)	insertLast(e)
pop()	removeLast()

## Queues with Deques:

Queue Method	Deque Implementation
size()	size()
isEmpty()	isEmpty()
front()	first()
enqueue()	insertLast(e)
dequeue()	removeFirst()

## Java implementation of Doubly Linked List

---

A node of a doubly linked list has a next and a prev link.

The doubly linked list supports the following methods:

- setElement(Object e)
- setNext(Object newNext)
- setPrev(Object newPrev)
- getElement()
- getNext()
- getPrev()

