## Overview

- Abstract Data Types
- Stacks
- Queues
- Deques

## Abstract Data Types (ADTs)

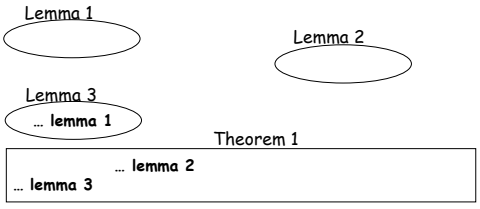
- An Abstract Data Type is an abstraction of a data structure.

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.

## Modularity - "Information Hiding"

Example of Modularity:




I could change the "implementation" (proof) of lemma 2 without changing the theorem.

## 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

…

- 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…)

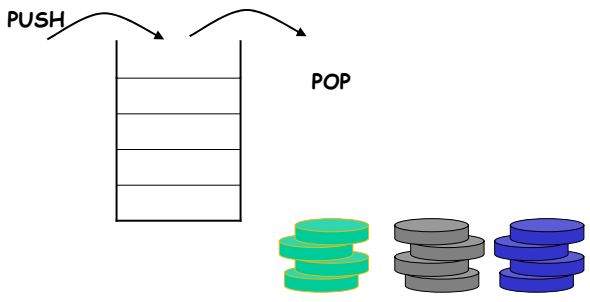
## Stacks, Queues, and Deques

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




## 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

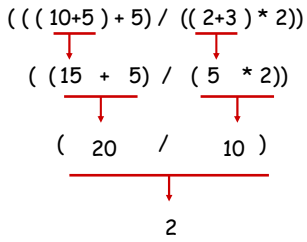
Evaluating an expression with two stacks

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

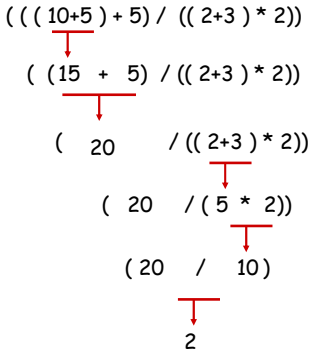
Hypothesis:
parenthesis are correct
only positive numbers

How do we solve it ?

## One possible sequence of operations



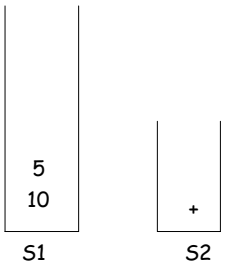

## Another one



## With two Stacks

One for the operands
One for the operators

( ( (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))

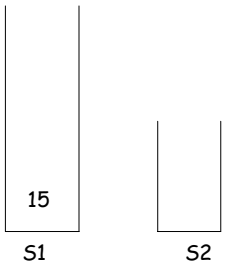
POP S1 5

POP S2 +

POP S1 10

Evaluate: 10 + 5 = 15

PUSH S1 the result




## With two Stacks

One for the operands
One for the operators

CLOSED parenthesis

( 20 / (( 2+3) * 2))
( ( (10+5) + 5) / ( (2+3) * 2))

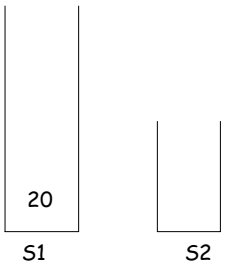
POP S1 5

POP S2 +

POP S1 15

Evaluate: 15 + 5 = 20

PUSH S1 the result




## With two Stacks

One for the operands
One for the operators

CLOSED parenthesis

( 20 / (( 2+3) * 2))
( ( (10+5) + 5) / ( (2+3) * 2))

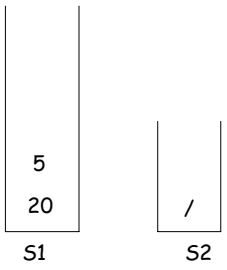
POP S1 3

POP S2 +

POP S1 2

Evaluate 2+ 3 = 5

PUSH S1 the result

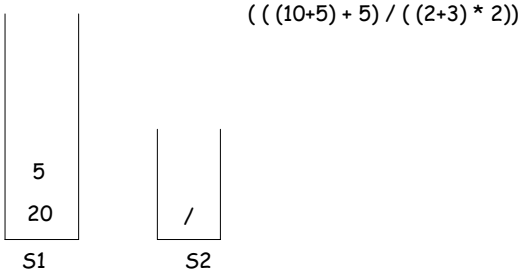


## With two Stacks

One for the operands
One for the operators

( 20 / ( 5 * 2))

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




## With two Stacks

One for the operands
One for the operators

CLOSED parenthesis

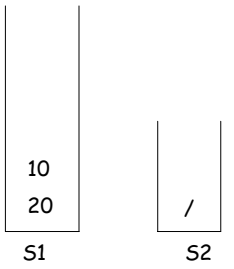
( 20 / ( 5 * 2))

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

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

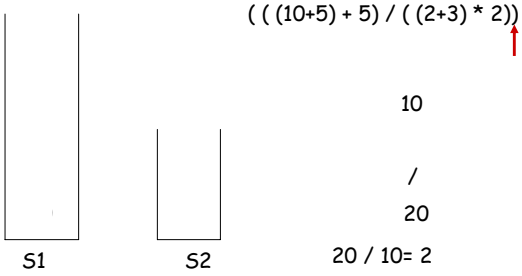
CLOSED parenthesis

( 20 / 10 )

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

10
/
20

20 / 10= 2



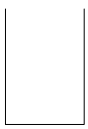

## Examples

Checking for balanced parenthesis

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

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

You will see this in the lab



# 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 ← t + 1
    S[t] ← obj

Algorithm pop():
    if isEmpty() then
        ERROR
    e ← S[t]
    S[t] ← null
    t ← 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

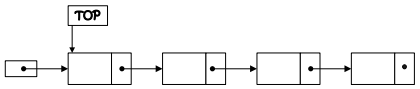
- Limitations

STATIC STRUCTURE

What does O(1) mean ?

we will see the formal definition next week

# Implementing a Stack with a Singly Linked List

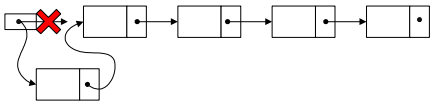



size = 4

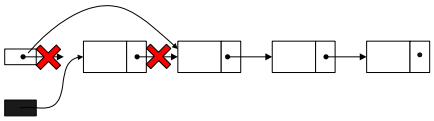
-Singly linked list plus a variable containing the current size of the list

DYNAMIC STRUCTURE

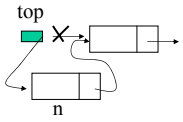
**PUSH**: Add at the front



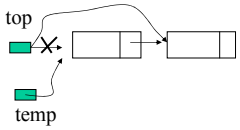
**POP**: Take the first



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:

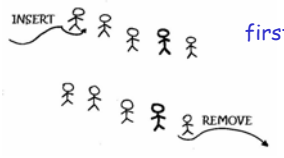
| | |
|---|---|
| size() | O(1) |
| isempty() | O(1) |
| top() | O(1) |
| push(obj) | O(1) |
| pop() | O(1) |

Space: Variable

Limitations: ?

## The Queue

## The Queue




first-in-first-out (FIFO)

Elements are inserted at the **rear** (enqueued) and removed from the **front** (dequeued)

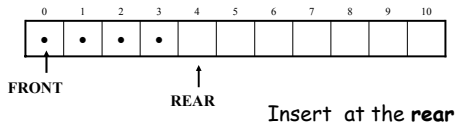
## Applications of Queues

- 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

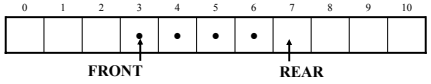
## The Queue Abstract Data Type

- 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

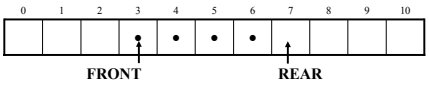
## Implementing a Queue with an Array

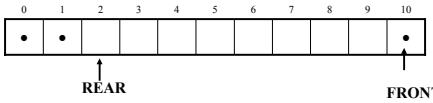


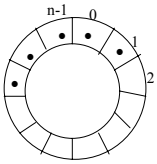

Insert at the **rear** and remove from **front**



# Implementing a Queue with an Array



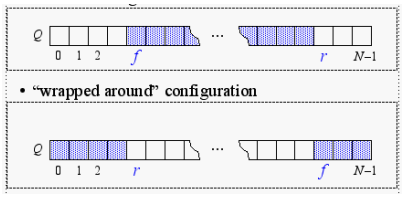



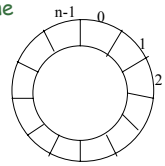

Remove: Front = (Front+1) **mod n**

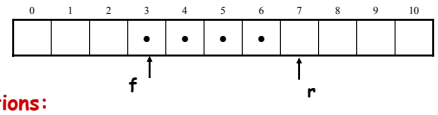
Insert: Rear = (Rear+1) **mod n**

- Array in a circular fashion
- 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



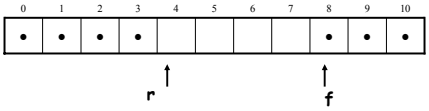




**Questions:**

**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) mod N

In the example:
(11 - 8 + 4) mod 11 = 7

```
Algorithm size():
        return (N - f + r) mod N

Algorithm isEmpty():
        return (f = r)

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

```
Algorithm dequeue():
        if isEmpty() then
                ERROR
        temp ← Q[f]
        Q[f] ← null
        f ← (f + 1) mod N
        return temp

Algorithm enqueue(o):
        if size = N - 1 then
                ERROR
        Q[r] ← o
```

## Performance

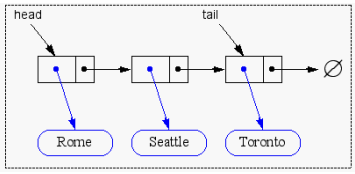
Time:

| | |
|---|---|
| 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

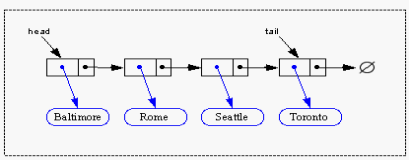
Nodes connected in singly linked list
We keep a pointer to the head and one to the tail



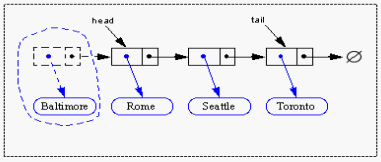

The head of the list is the front of the queue, the tail of the list is the rear of the queue.

***Why not the opposite?***

## Removing at the Head

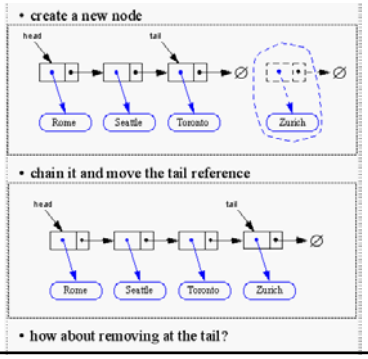



- **advance head reference**




- **inserting at the head is just as easy**

## Inserting 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:

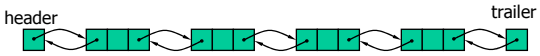
- insertFirst(e): Insert e at the beginning of deque.
- insertLast(e): Insert e at end of deque
- removeFirst(): Removes and returns first element
- removeLast(): Removes and returns last element

Support methods:

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

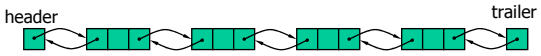
# Implementing Deques with Doubly Linked Lists

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



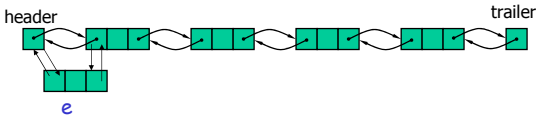

To implement a deque, we use a doubly linked list with special header and trailer nodes

- 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):



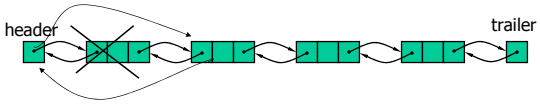

e.next ← header.next

header.next ← e

e.prev ← header

e.next.prev ← e

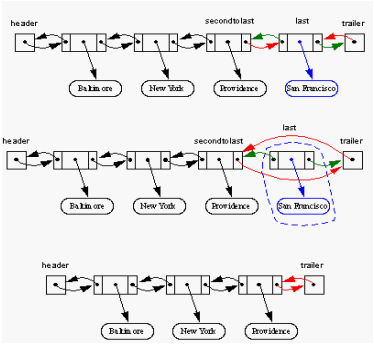
## removeFirst():




header.next ← header.next.next

header.next.prev ← header

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




With this implementation, all methods have 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() |