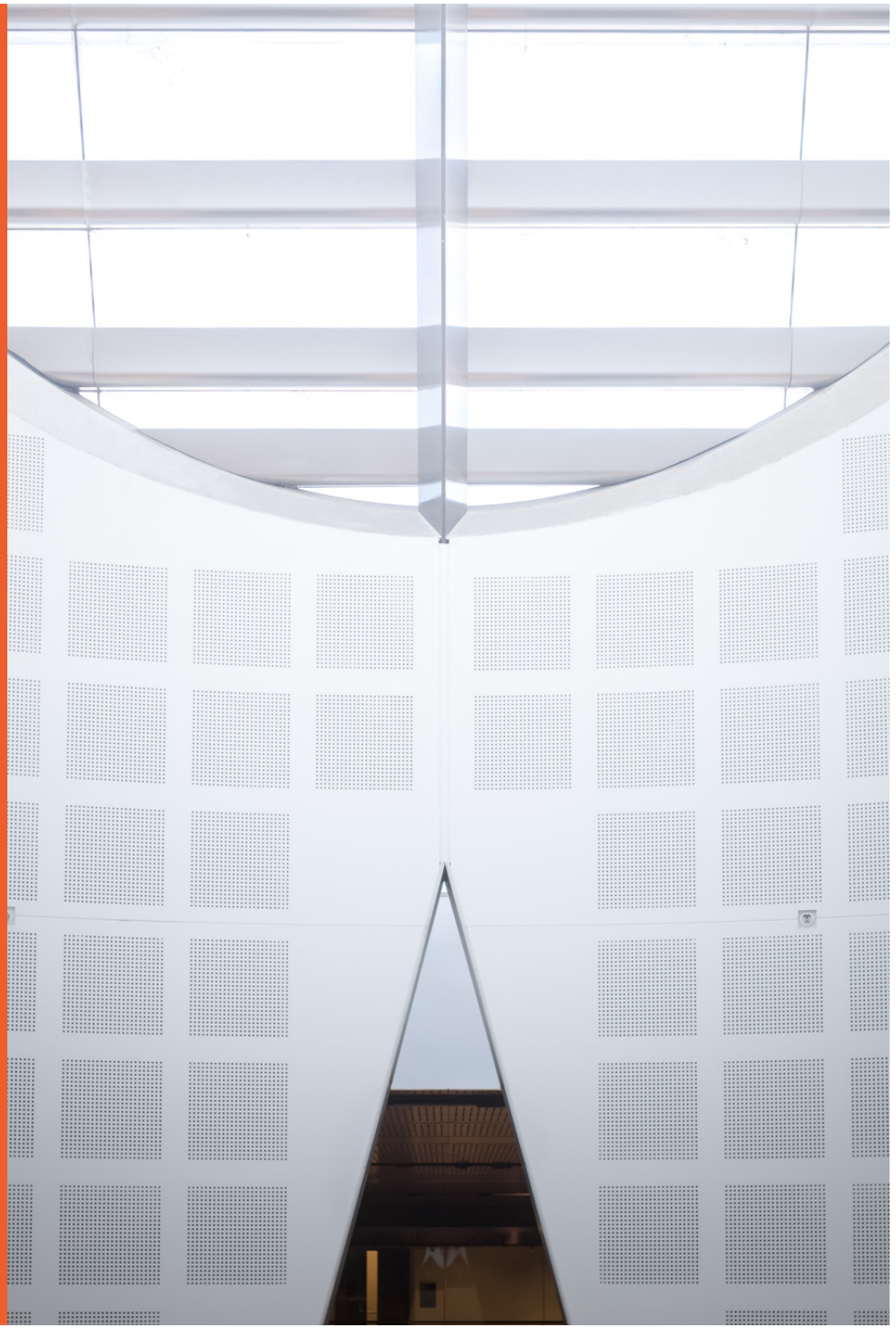
INFO1105/1905
Data Structures
Week 3a: Stacks, Queues, Deques

see textbook chapter 6

Professor Alan Fekete
Dr John Stavrakakis
School of Information Technologies

using material from the textbook and A/Prof Kalina Yacef,



THE UNIVERSITY OF SYDNEY

Copyright warning

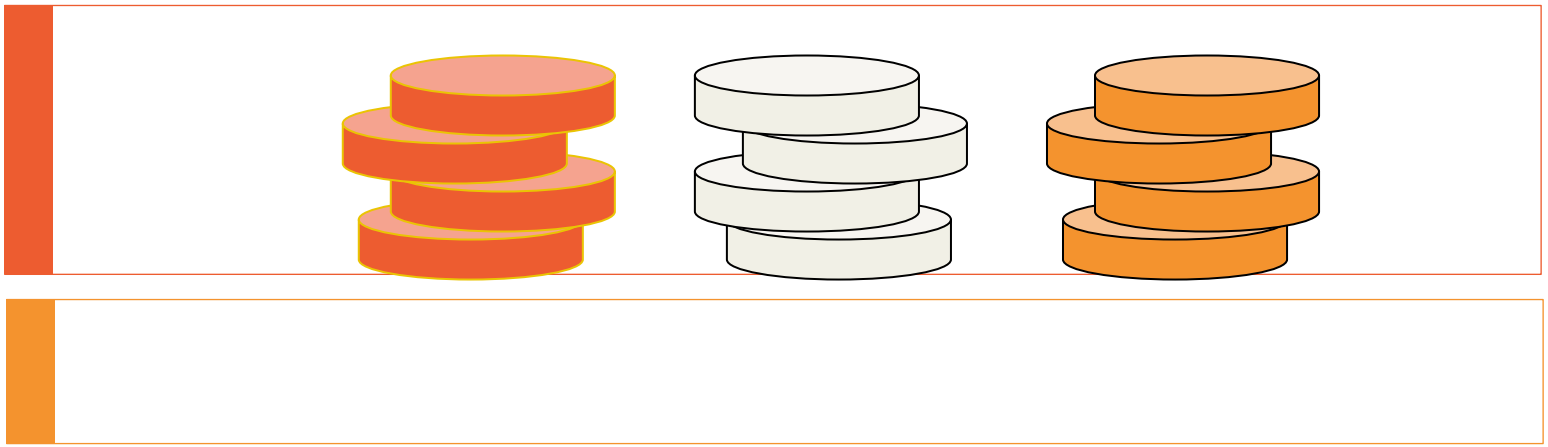
Overview

- We introduce three more ADTs (Stack, Queue, Deque)
- Each is a restricted form of List, where insertion and removal happen only in particular places (not in arbitrary places of the collection)
  - operations have special names, that are part of the culture on computing
  - several interesting applications can use these ADTs effetcively
  - there are ways to implement these ADTs that are simpler than (and/or more efficient than) implementations of general Lists
- Instead of using these ADTs, one could use a general-purpose List, and a library implementation of that List

Reminder! Quiz 1

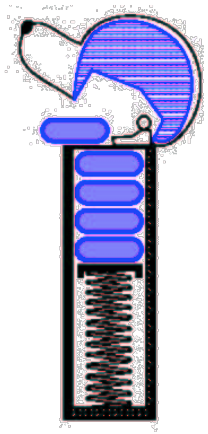
- Quiz 1 will take place during lab in week 4
- Done online, over a 20 minutes duration,
  - during the last 30 minutes of the lab period, or as indicated by your tutor
- A few multiple choice questions,
  - covering material from lectures of weeks 1 and 2 (labs 2 and 3), and also a bit of material from today's lecture
    - recursion
    - linked list re-arrangements
    - costs of operations on lists (array or linked)
    - tracing operations on stacks, queues, deques
    - key ideas with big-Oh notation

Stacks



The Stack ADT

- The Stack ADT stores arbitrary objects
- Insertions and deletions follow the last-in first-out scheme (LIFO)
- Think of a spring-loaded plate dispenser
- Main stack operations:
  - push(object): inserts an element
  - object pop(): removes and returns the last inserted element
- Auxiliary stack operations:
  - object top(): returns the last inserted element without removing it
  - integer size(): returns the number of elements stored
  - boolean isEmpty(): indicates whether no elements are stored



Example

Method	Return Value	Stack Contents
push(5)	–	(5)
push(3)	–	(5, 3)
size( )	2	(5, 3)
pop( )	3	(5)
isEmpty( )	false	(5)
pop( )	5	( )
isEmpty( )	true	( )
pop( )	null	( )
push(7)	–	(7)
push(9)	–	(7, 9)
top( )	9	(7, 9)
push(4)	–	(7, 9, 4)
size( )	3	(7, 9, 4)
pop( )	4	(7, 9)
push(6)	–	(7, 9, 6)
push(8)	–	(7, 9, 6, 8)
pop( )	8	(7, 9, 6)

Stack Interface in Java

- Java interface corresponding to our Stack ADT
- Assumes null is returned from top() and pop() when stack is empty
- Different from the built-in Java class java.util.Stack

```
public interface Stack<E> {
  int size();
  boolean isEmpty();
  E top();
  void push(E element);
  E pop();
}
```

Exceptions vs. Returning Null

- Attempting the execution of an operation of an ADT may sometimes cause an error condition
- Java supports a general abstraction for errors, called exception
- An exception is said to be "thrown" by an operation that cannot be properly executed
- In our Stack ADT, we do not use exceptions
- Instead, we allow operations pop and top to be performed even if the stack is empty
- For an empty stack, pop and top simply return null

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

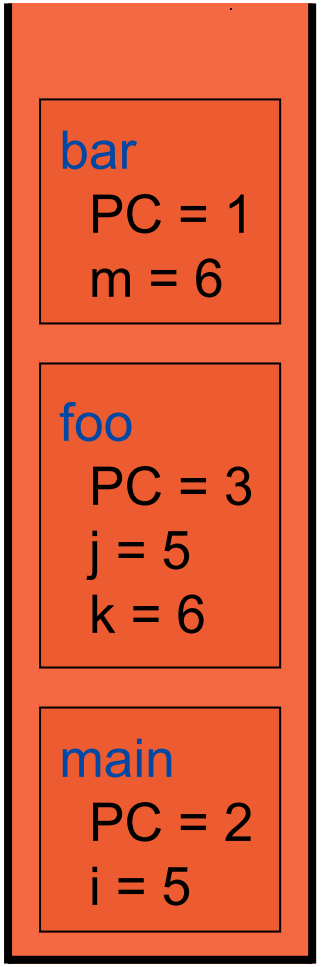
Method Stack in the JVM

- The Java Virtual Machine (JVM) keeps track of the chain of active methods with a stack
- When a method is called, the JVM pushes on the stack a frame containing
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When a method ends, its frame is popped from the stack and control is passed to the method on top of the stack
- Allows for recursion

```
main() {
  int i = 5;
  foo(i);
  }

foo(int j) {
  int k;
  k = j+1;
  bar(k);
  }

bar(int m) {
  …
  }
```



Array-based Stack

- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable keeps track of the index of the top element

```
Algorithm size()
  return t + 1

Algorithm pop()
  if isEmpty() then
    return null
  else
    ans ← S[t]
    t ← t - 1
    return ans
```

Array-based Stack (cont.)

- The array storing the stack elements may become full
- A push operation will then throw a IllegalStackException
  - Limitation of the array-based implementation
  - Not intrinsic to the Stack ADT
  - but included in Java interface

Algorithm *push*(*o*)
  if $t = S.length - 1$ **then**
    throw *IllegalStackException*
  else
    $t \leftarrow t + 1$
    $S[t] \leftarrow o$

Performance and Limitations

- Performance
  - Let n be the number of elements in the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$
- 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

textbook's Array-based Stack in Java

```
public class ArrayStack<E>
      implements Stack<E> {

   // holds the stack elements
   private E data[ ];

   // index to top element
   private int t = -1;

   // constructor
   public ArrayStack(int capacity) {
       data = (E[ ]) new Object[capacity];
    }
```

```
   public E pop() {
       if isEmpty()
          return null;
       E answer= data[t];
       // facilitate garbage collection:
       data[t] = null;
       t--;
       return answer;
    }

…  (other methods of Stack interface)
```

Example Use in Java

```
public class Tester {
   // … other methods
   public intReverse(Integer a[]) {
       Stack<Integer> s;
                          s = new
ArrayStack<Integer>();
       … (code to reverse array a) …
    }
```

```
   public floatReverse(Float f[]) {
       Stack<Float> s;
       s = new ArrayStack<Float>();
       … (code to reverse array f) …
    }
```

Example of stack usage: Reversing an array non-recursively

```
/** A nonrecursive generic method for reversing an array */
public static <E> void reverse(E[] a) {
  Stack<E> S = new ArrayStack<E>(a.length);
  for (int i=0; i < a.length; i++)
    S.push(a[i]);
  for (int i=0; i < a.length; i++)
    a[i] = S.pop();
}
```

```
/** Tester routine for reversing arrays */
public static void main(String args[]) {
  Integer[] a = {4, 8, 15, 16, 23, 42}; // autoboxing allows this
  String[] s = {"Jack", "Kate", "Hurley", "Jin", "Boone"};
  System.out.println("a = " + Arrays.toString(a));
  System.out.println("s = " + Arrays.toString(s));
  System.out.println("Reversing...");
  reverse(a);
  reverse(s);
  System.out.println("a = " + Arrays.toString(a));
  System.out.println("s = " + Arrays.toString(s));
}
```

```
a = [4, 8, 15, 16, 23, 42]
s = [Jack, Kate, Hurley, Jin, Michael]
Reversing...
a = [42, 23, 16, 15, 8, 4]
s = [Michael, Jin, Hurley, Kate, Jack]
```

Performance and Limitations of Array-based Stacks

- Performance
  - Let n be the number of elements in the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$
- 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
- Another implementation: Linked List-based

Linked List based Stacks

- No arbitrary capacity limit
- Top of stack is front of list: insertion and deletion in constant time

Stack method	**Singly Linked list method**
size() isEmpty() push(e) pop() top()	list.size() list.isEmpty() list.addFirst(e) list.removeFirst() list.first ()

Application of Stacks in programming

- Parenthesis matching
- HTML tag matching
- Function/method calls
  - Including recursive calls
- Evaluating Arithmetic Expressions

Parentheses Matching

- Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “[”
  - correct: ( )(( )){([( )])}
  - correct: ((( )(( )){([( )])}
  - incorrect:)(( )){([( )])}
  - incorrect: ({[ ])}
  - incorrect: (

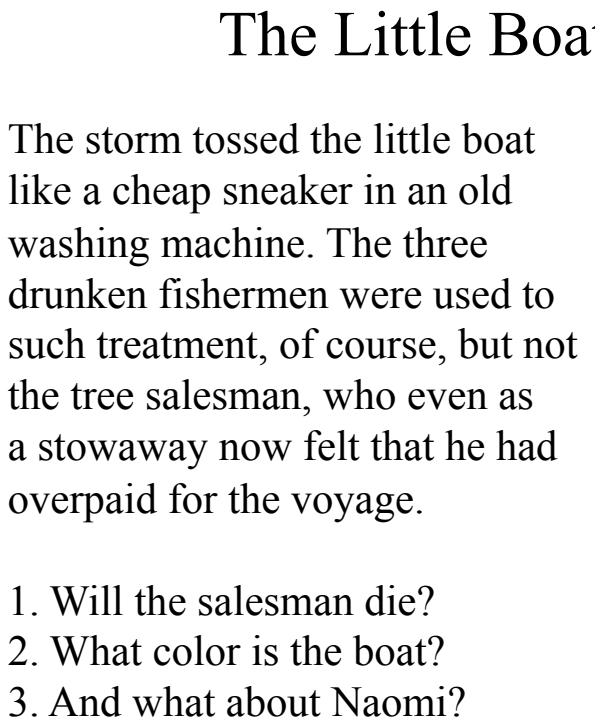
Parentheses Matching Algorithm

Algorithm ParenMatch(*X,n*):
Input: An array *X* of *n* tokens, each of which is either a grouping symbol, a variable, an arithmetic operator, or a number
Output: **true** if and only if all the grouping symbols in *X* match
Let *S* be an empty stack
for *i*=0 to *n*-1 **do**
    if *X*[*i*] is an opening grouping symbol **then**
      S.push(*X*[*i*])
    else if *X*[*i*] is a closing grouping symbol **then**
      if *S*.isEmpty() **then**
          return false {nothing to match with}
      if *S*.pop() does not match the type of *X*[*i*] **then**
          return false {wrong type}
if *S*.isEmpty() **then**
    return true {every symbol matched}
else return false {some symbols were never matched}

HTML Tag Matching

For fully-correct HTML, each <name> should pair with a matching </name>

```
<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little
boat like a cheap sneaker in an
old washing machine. The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even as
a stowaway now felt that he
had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>
```



Tag Matching Algorithm (in Java)

```
public static boolean isHTMLMatched(String html) {
  Stack<String> buffer =  new LinkedStack<>( );
  int j = html.indexOf('<'); // find first '<' character (if any)
  while (j != −1) {
    int k = html.indexOf('>', j+1); // find next '>' character
    if (k == −1)
      return false;  // invalid tag
    String tag = html.substring(j+1, k);  // strip away < >
    if (!tag.startsWith("/")) // this is an opening tag
      buffer.push(tag);
    else { // this is a closing tag
      if (buffer.isEmpty( ))
        return false;  // no tag to match
      if (!tag.substring(1).equals(buffer.pop( )))
        return false;  // mismatched tag
    }
    j = html.indexOf('<', k+1);  // find next '<' character (if any)
  }
  return buffer.isEmpty( );  // were all opening tags matched?
}
```

Evaluating Arithmetic Expressions

Slide by Matt Stallmann included with permission.

14 – 3 * 2 + 7 = (14 – (3 * 2)) + 7

Operator precedence

* has precedence over +/–

Associativity

operators of the same precedence group evaluated from left to right

Example: (x – y) + z rather than x – (y + z)

Idea: push each operator on the stack, but first pop and perform higher and *equal* precedence operations.

Algorithm for Evaluating Expressions

Slide by Matt Stallmann included with permission.

Two stacks:

- opStk holds operators
- valStk holds values
- Use $ as special "end of input" token with lowest precedence

Algorithm doOp()

```
x ← valStk.pop();
y ← valStk.pop();
op ← opStk.pop();
valStk.push( y op x )
```

Algorithm repeatOps(refOp):

```
while ( valStk.size() > 1 ∧
    prec(refOp) ≤ prec(opStk.top()))
  doOp()
```

Algorithm EvalExp()

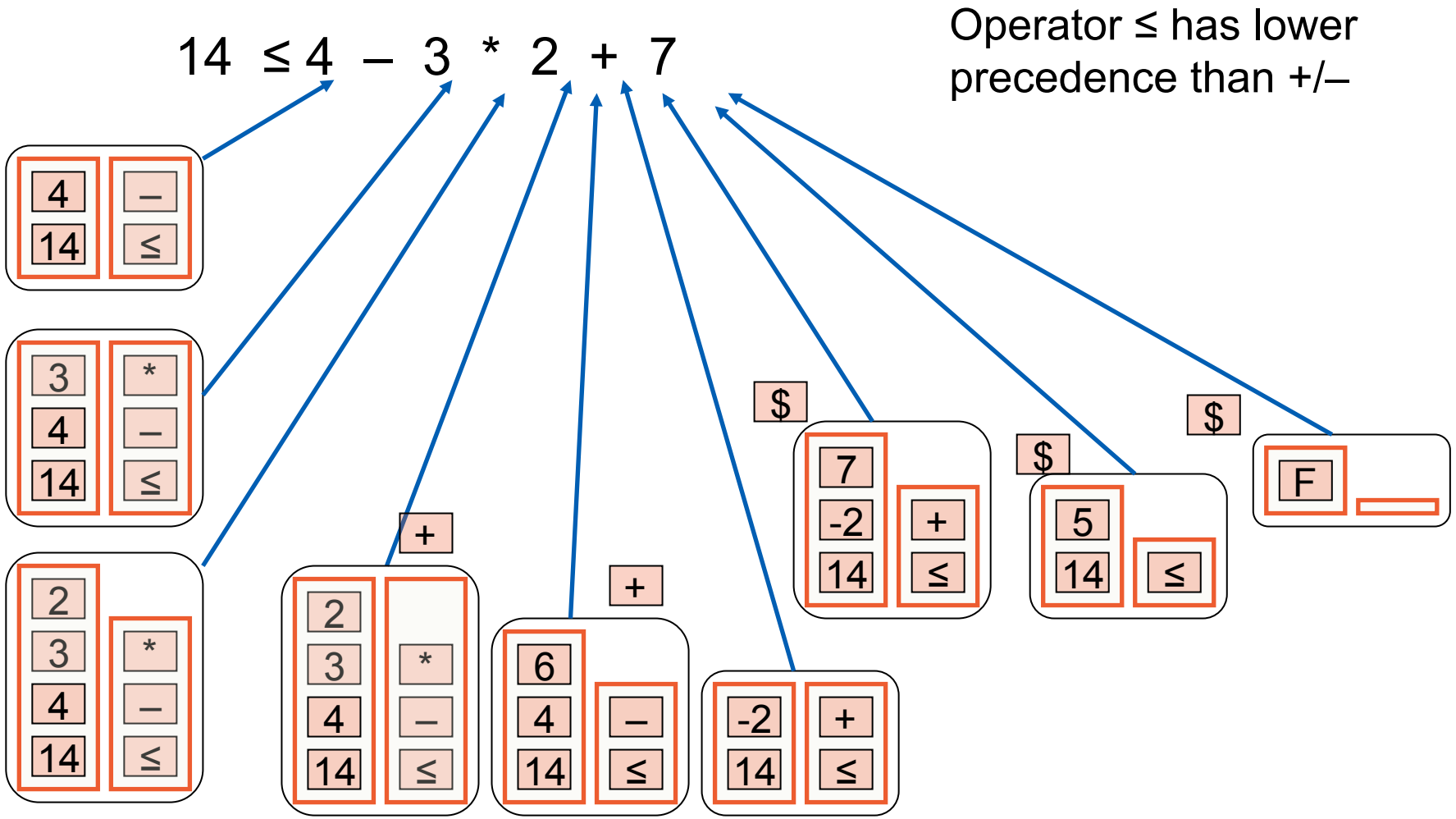
Input: a stream of tokens representing an arithmetic expression (with numbers)

Output: the value of the expression

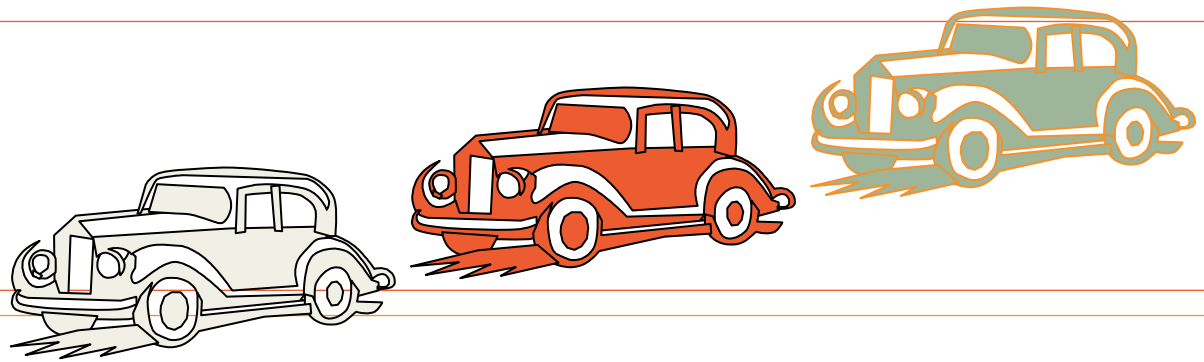
```
while there's another token z
 if isNumber(z) then
    valStk.push(z)
    else
    repeatOps(z);
    opStk.push(z)
repeatOps($);
return valStk.top()
```

Algorithm on an Example Expression

Slide by Matt Stallmann included with permission.



Queues



The Queue ADT

- The Queue ADT stores arbitrary objects
- Insertions and deletions follow the first-in first-out scheme (FIFO)
- Insertions are at the rear of the queue and removals are at the front of the queue
- Main queue operations:
  - enqueue(object): inserts an element at the end of the queue
  - object dequeue(): removes and returns the element at the front of the queue
- Auxiliary queue operations:
  - object front(): returns the element at the front without removing it
  - integer size(): returns the number of elements stored
  - boolean isEmpty(): indicates whether no elements are stored
- Boundary cases:
  - Attempting the execution of dequeue or front on an empty queue returns null

Example

Operation	*Output*	*Q*
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
dequeue()	*5*	(3)
enqueue(7)	–	(3, 7)
dequeue()	*3*	(7)
front()	*7*	(7)
dequeue()	*7*	()
dequeue()	*null*	()
isEmpty()	*true*	()
enqueue(9)	–	(9)
enqueue(7)	–	(9, 7)
size()	*2*	(9, 7)
enqueue(3)	–	(9, 7, 3)
enqueue(5)	–	(9, 7, 3, 5)
dequeue()	*9*	(7, 3, 5)

Applications of Queues

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

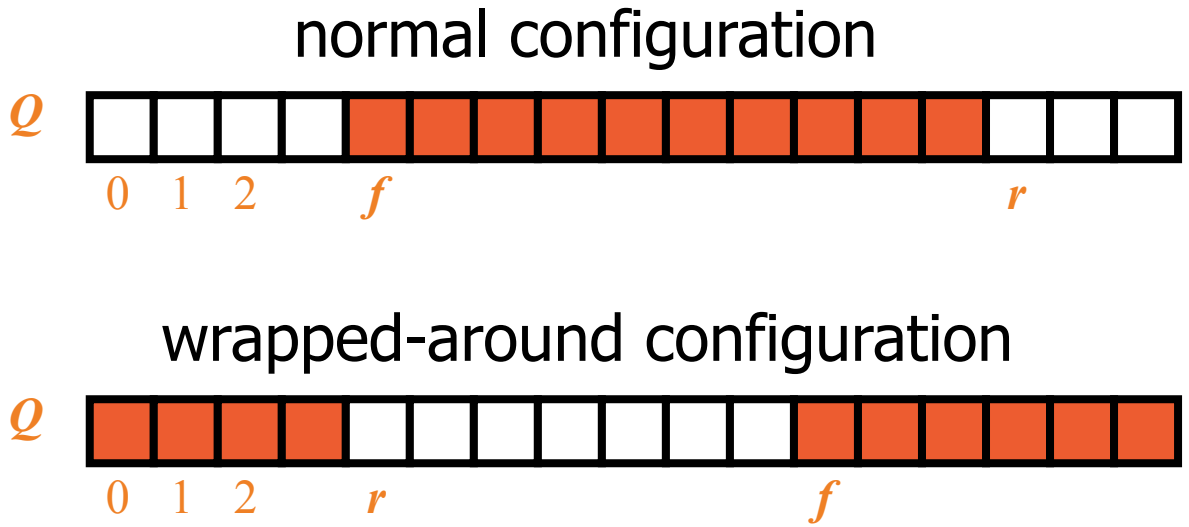
textbook's Queue Interface in Java

- Java interface corresponding to our Queue ADT
- Assumes that first() and dequeue() return null if queue is empty

```
public interface Queue<E> {
   int size();
   boolean isEmpty();
   E first();
   void enqueue(E element);
   E dequeue();
}
```

Array-based Queue

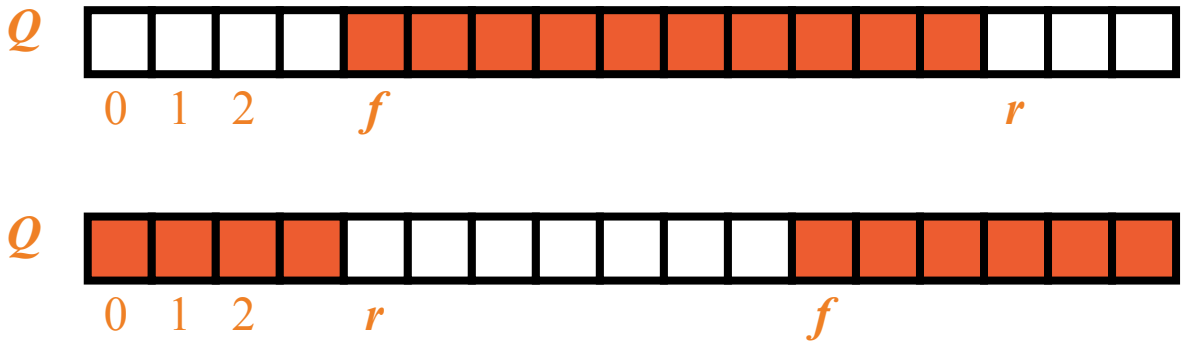
- Use an array of size N in a circular fashion
- Two variables keep track of the front and size
  - f index of the front element
  - sz number of stored elements
- When the queue has fewer than **N** elements, array location **r = (f + sz) *mod* N** is the first empty slot past the rear of the queue



Queue Operations

Algorithm ***size***()
  return sz

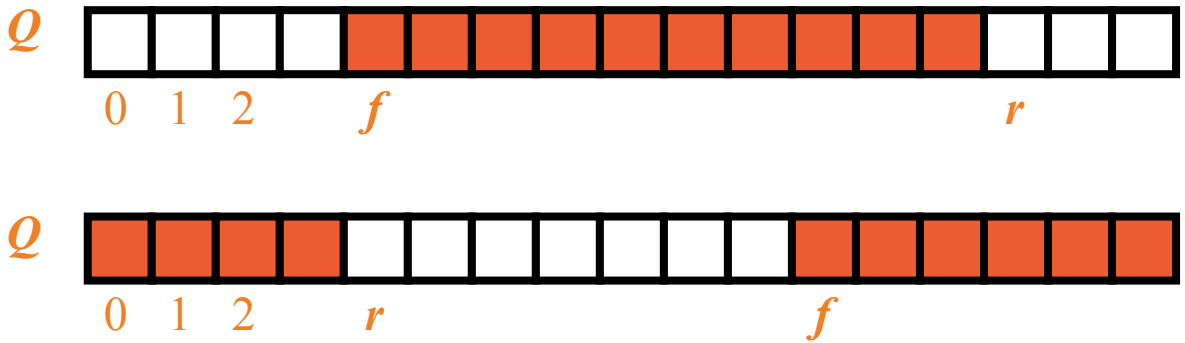
Algorithm ***isEmpty***()
  return $(sz = 0)$



Queue Operations (cont.)

- Operation enqueue throws an exception if the array is full
- This exception is implementation-dependent

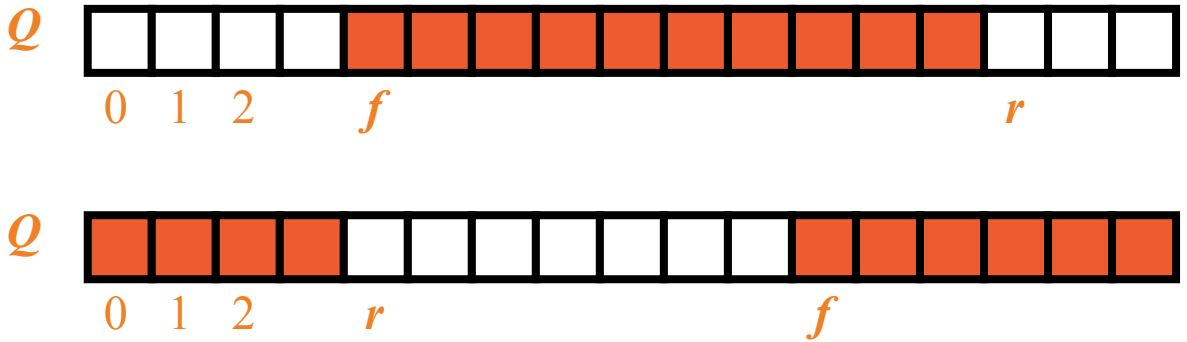
Algorithm *enqueue*(*o*)
  if *size*() = *N* **then**
    throw *IllegalStateException*
  else
    Q[*r*] ← *o*
    sz ← (*sz* + 1)



Queue Operations (cont.)

- Operation dequeue returns null if the queue is empty

```
Algorithm dequeue()
  if isEmpty() then
    return null
  else
    o ← Q[f]
    f ← (f + 1) mod N
    sz ← (sz - 1)
    return o
```



textbook's Array-based Implementation

```
1  /** Implementation of the queue ADT using a fixed-length array. */
2  public class ArrayQueue<E> implements Queue<E> {
3    // instance variables
4    private E[ ] data;                          // generic array used for storage
5    private int f = 0;                          // index of the front element
6    private int sz = 0;                         // current number of elements
7
8    // constructors
9    public ArrayQueue( ) {this(CAPACITY);}  // constructs queue with default capacity
10   public ArrayQueue(int capacity) {           // constructs queue with given capacity
11     data = (E[ ]) new Object[capacity];       // safe cast; compiler may give warning
12   }
13
14   // methods
15   /** Returns the number of elements in the queue. */
16   public int size( ) { return sz; }
17
18   /** Tests whether the queue is empty. */
19   public boolean isEmpty( ) { return (sz == 0); }
20
```

textbook's Array-based Implementation (2)

```
21    /** Inserts an element at the rear of the queue. */
22    public void enqueue(E e) throws IllegalStateException {
23      if (sz == data.length) throw new IllegalStateException("Queue is full");
24      int avail = (f + sz) % data.length;       // use modular arithmetic
25      data[avail] = e;
26      sz++;
27    }
28
29    /** Returns, but does not remove, the first element of the queue (null if empty). */
30    public E first( ) {
31      if (isEmpty( )) return null;
32      return data[f];
33    }
34
35    /** Removes and returns the first element of the queue (null if empty). */
36    public E dequeue( ) {
37      if (isEmpty( )) return null;
38      E answer = data[f];
39      data[f] = null;                           // dereference to help garbage collection
40      f = (f + 1) % data.length;
41      sz--;
42      return answer;
43    }
```

Comparison to java.util.Queue

- Our Queue methods and corresponding methods of java.util.Queue:

Our Queue ADT	Interface java.util.Queue	
	throws exceptions	returns special value
enqueue(e)	add(e)	offer(e)
dequeue( )	remove( )	poll( )
first( )	element( )	peek( )
size( )	size( )	
isEmpty( )	isEmpty( )	

Linked list Queue implementation

- Linked-list implementation of the ADT Queue
  - Overcomes size limitations
  - But
    - size per element slightly bigger
    - more complicated, extra care needed for cases where queue is empty before enqueue and after dequeue
    - need to keep pointers to both the first and the last nodes in the list if not then additions will not be O(1)
  - First = head of list

Queue with linked lists - enqueue

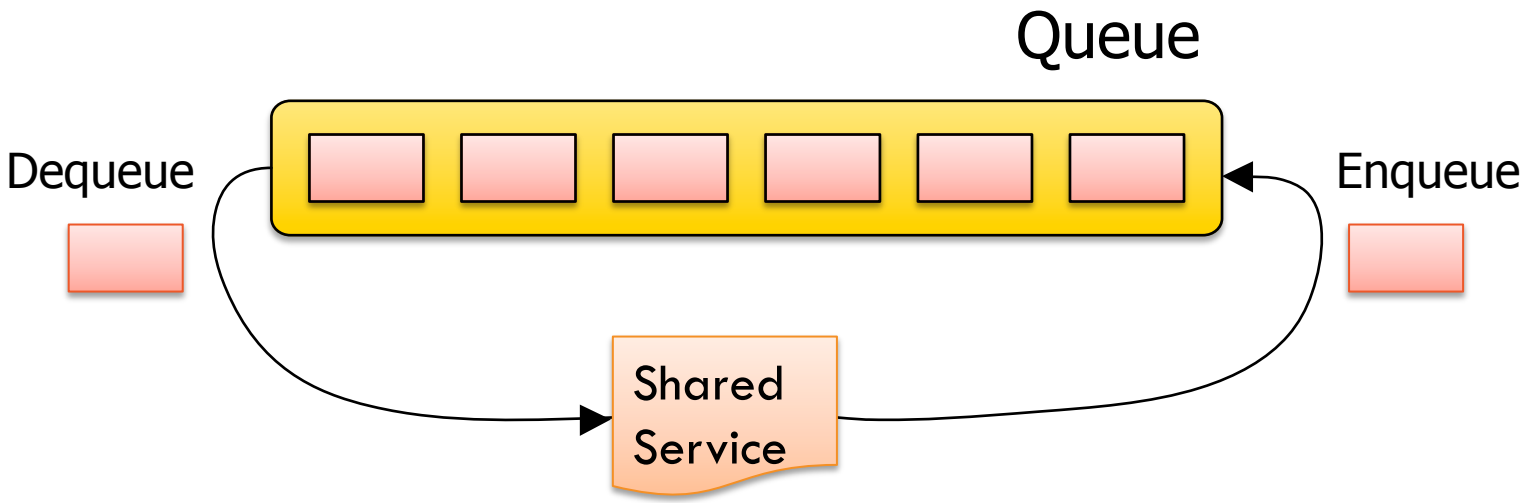
```
public void enqueue(E elem) {
  Node<E> node = new Node<E>();
  node.setElement(elem);
  node.setNext(null); // node will be new tail node
  if (size == 0)
    head = node; // special case of a previously empty queue
  else
    tail.setNext(node); // add node at the tail of the list
  tail = node; // update the reference to the tail node
  size++;
}
```

Queue with linked lists - dequeue

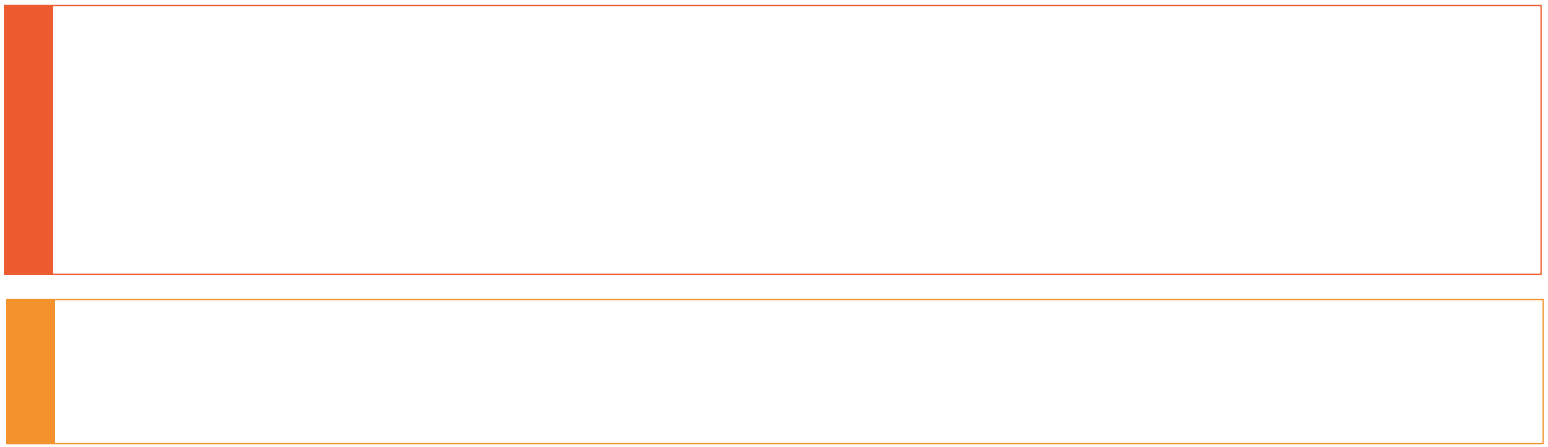
```
public E dequeue() throws EmptyQueueException {
  if (size == 0)
    throw new EmptyQueueException("Queue is empty.");
  E tmp = head.getElement();
  head = head.getNext();
  size--;
  if (size == 0)
    tail = null; // the queue is now empty
  return tmp;
}
```

Application: Round Robin Schedulers

- We can implement a round robin scheduler using a queue Q by repeatedly performing the following steps:
  1. e = Q.dequeue()
  2. Service element e
  3. Q.enqueue(e)



Deques



Double-ended queues: Deques

- A linear structure that allows insertions and deletions at both ends

Method	Time
size, isEmpty	$O(1)$
getFirst, getLast	$O(1)$
addFirst, addLast	$O(1)$
removeFirst, removeLast	$O(1)$

Table 5.4: Performance of a deque realized by a doubly linked list.

Double-ended queue operations

The deque abstract data type is richer than both the stack and the queue ADTs. The fundamental methods of the deque ADT are as follows:

addFirst(e): Insert a new element e at the head of the deque.

addLast(e): Insert a new element e at the tail of the deque.

removeFirst(): Remove and return the first element of the deque; an error occurs if the deque is empty.

removeLast(): Remove and return the last element of the deque; an error occurs if the deque is empty.

Additionally, the deque ADT may also include the following support methods:

getFirst(): Return the first element of the deque; an error occurs if the deque is empty.

getLast(): Return the last element of the deque; an error occurs if the deque is empty.

size(): Return the number of elements of the deque.

isEmpty(): Determine if the deque is empty.

java.util.Deque interface

```
public interface Deque<E>  {
    int size();
    boolean isEmpty();
    E getFirst();
    E getLast();
    void addFirst (E element);
    void addLast (E element);
    E removeFirst();
    E removeLast();
    // other methods see next slide
```

java.util.Deque: extra methods

offerFirst(e):	Inserts e at the head of the deque unless the deque is full, in which case this method returns **false**; if the method succeeds, it returns **true**.
offerLast(e):	Inserts e at the tail of the deque unless the deque is full, in which case this method returns **false**; if the method succeeds, it returns **true**.
pollFirst():	Removes and returns the first element of the deque; if the deque is empty, this method returns **null**.
pollLast():	Removes and returns the last element of the deque; if the deque is empty, this method returns **null**.
peekFirst():	Returns, but does not remove, the first element of the deque; if the deque is empty, this method returns **null**.
peekLast():	Returns, but does not remove, the last element of the deque; if the deque is empty, this method returns **null**.