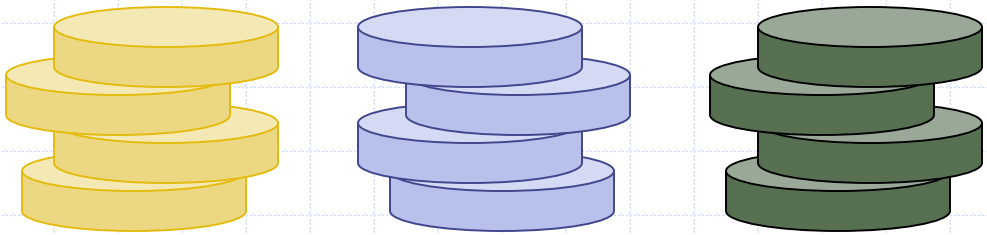
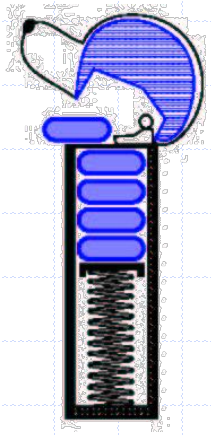
Modified version of the presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Stacks



The Stack ADT



- The Stack ADT stores arbitrary objects
- Main stack operations:
  - push(object): inserts an element
  - object pop(): removes and returns the last inserted element
- Insertions (push) and deletions (pop) follow the last-in first-out (LIFO) scheme

- 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

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();
}
```

Example

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

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)		
push(9)		
top( )		
push(4)		
size( )		
pop( )		
push(6)		
push(8)		
pop( )		

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)

Exceptions vs. Returning Null

- The presented Stack ADT, does not use exceptions
- Instead, it allows operations pop and top to be performed even if the stack is empty
- For an empty stack, pop and top simply return null
- In Java, most collections return null if "asked" for an object when they are empty
- The "right" way depends on a lot of things
- If you are not careful, you will need to handle a NullPointerException anyways…

Side Note: If the method is returning a collection, it is better to return an empty one instead of a null one

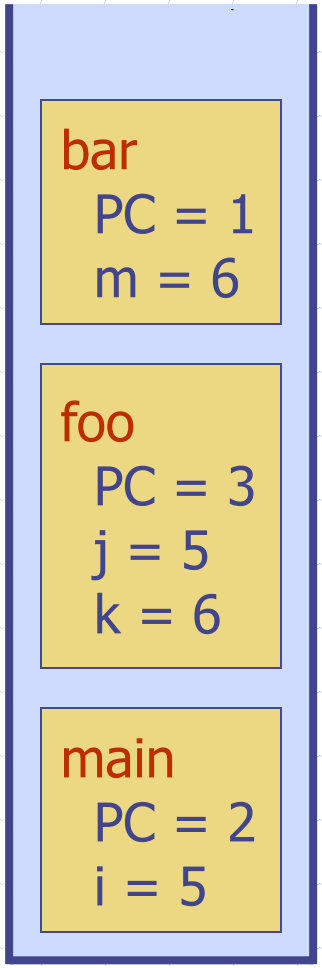
Applications of Stacks

- 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

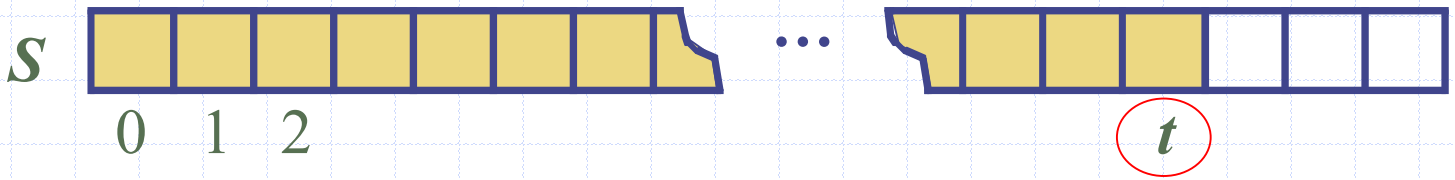
```
main() {
  int i = 5;
  foo(i);
  }

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

bar(int m) {
  …
  }
```



Array-based Stack

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

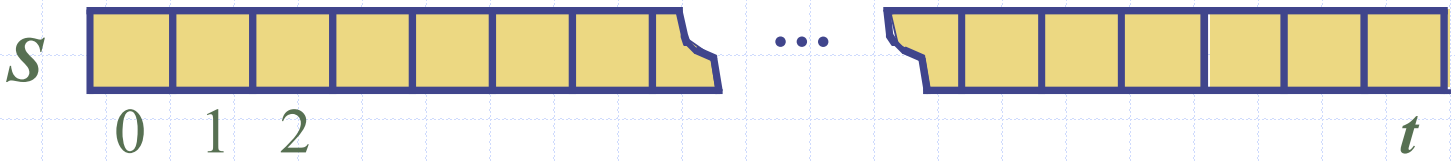
```
Algorithm size()
  return t + 1

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



Array-based Stack (cont.)

- The array storing the stack elements may become full
- A push operation will then throw a FullStackException
  - Limitation of the array-based implementation
  - Not intrinsic to the Stack ADT
  - Could also dynamically grow the array

Algorithm *push*(o)
  if $t = S.length - 1$ **then**
    throw *IllegalStateException*
  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 unless the array is dynamically grown
  - Trying to push a new element into a full stack causes an implementation-specific exception

Array-based Stack in Java

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

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

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

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

```
   public E pop() {
       if isEmpty()
            return null;
       E temp = S[top];
       // facilitate garbage collection:
       S[top] = null;
       top = top – 1;
       return temp;
    }

… (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) …
    }
```

Throwback: How did we reverse an array?

Parentheses Matching

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

Parentheses Matching (Java)

```
public static boolean isMatched(String expression) {
  final String opening = "({["; // opening delimiters
  final String closing = ")}]"; // respective closing delimiters
  Stack<Character> buffer = new LinkedStack<>( );




}
```

Parentheses Matching (Java)

```
public static boolean isMatched(String expression) {
  final String opening = "({["; // opening delimiters
  final String closing = ")}]"; // respective closing delimiters
  Stack<Character> buffer = new LinkedStack<>( );
  for (char c : expression.toCharArray( )) {
    if (opening.indexOf(c) != −1) // this is a left delimiter
      buffer.push(c);
```

Gives the index of object in the stack or returns -1 if non-existent

```
}
```

Parentheses Matching (Java)

```
public static boolean isMatched(String expression) {
  final String opening = "({["; // opening delimiters
  final String closing = ")}]"; // respective closing delimiters
  Stack<Character> buffer = new LinkedStack<>( );
  for (char c : expression.toCharArray( )) {
    if (opening.indexOf(c) != −1) // this is a left delimiter
      buffer.push(c);
   else if (closing.indexOf(c) != −1) { // this is a right delimiter
     if (buffer.isEmpty( )) // nothing to match with
       return false;
     if (closing.indexOf(c) != opening.indexOf(buffer.pop( )))
       return false; // mismatched delimiter
     }


}
```

Parentheses Matching (Java)

```
public static boolean isMatched(String expression) {
  final String opening = "({["; // opening delimiters
  final String closing = ")}]"; // respective closing delimiters
  Stack<Character> buffer = new LinkedStack<>( );
  for (char c : expression.toCharArray( )) {
    if (opening.indexOf(c) != −1) // this is a left delimiter
      buffer.push(c);
   else if (closing.indexOf(c) != −1) { // this is a right delimiter
     if (buffer.isEmpty( )) // nothing to match with
       return false;
     if (closing.indexOf(c) != opening.indexOf(buffer.pop( )))
       return false; // mismatched delimiter
     }
   }
  return buffer.isEmpty( ); // were all opening delimiters 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>
```

The Little Boat

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.

1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

HTML Tag Matching (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 < >
```

Starts looking from this index

```
}
```

HTML Tag Matching (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
    }
}
```

HTML Tag Matching (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()
```

Note that this ignores parentheses!

Algorithm EvalExp()

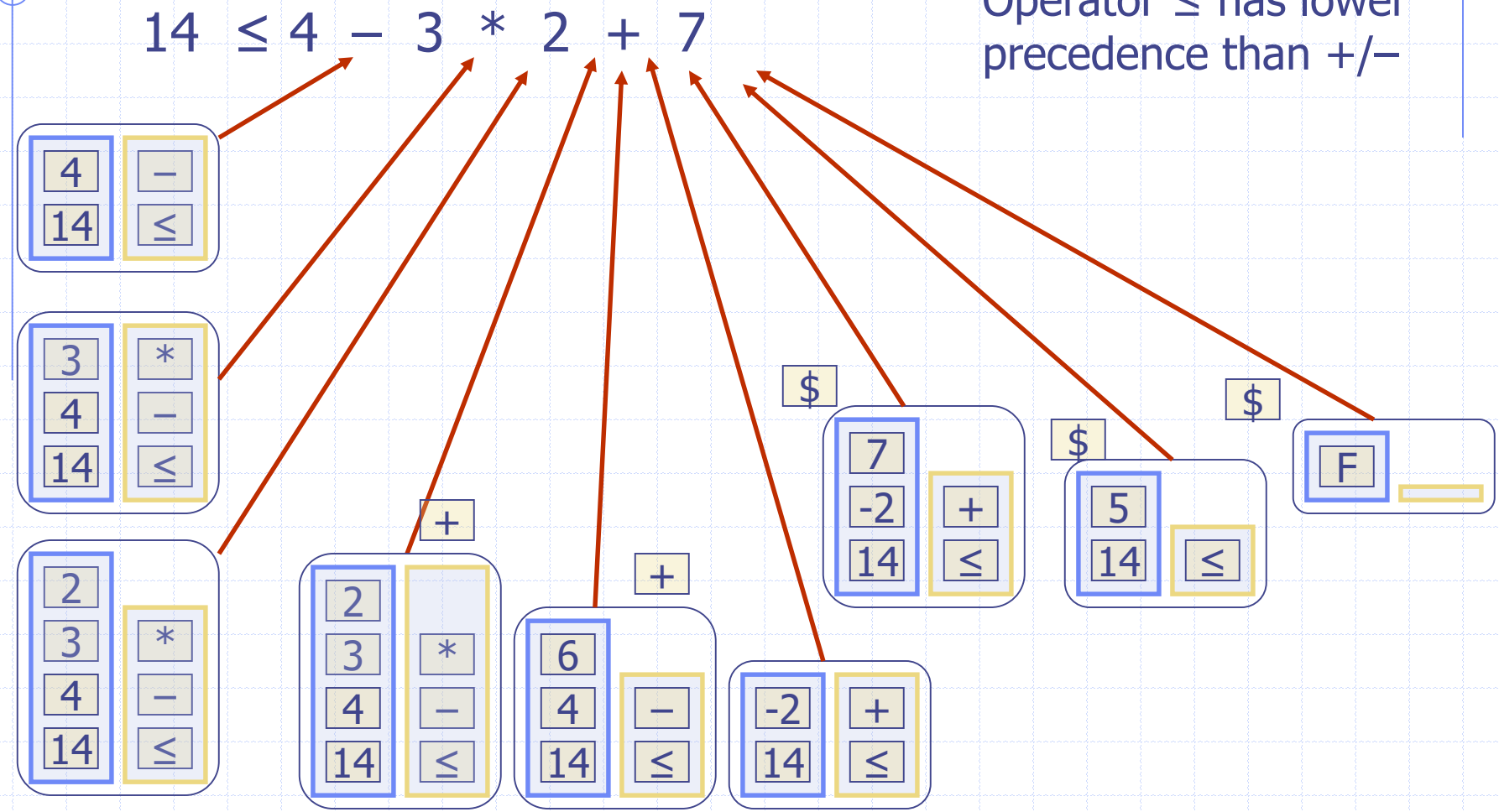
Input: a stream of tokens representing an arithmetic expression (with numbers)
Output: the value of the expression

```
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

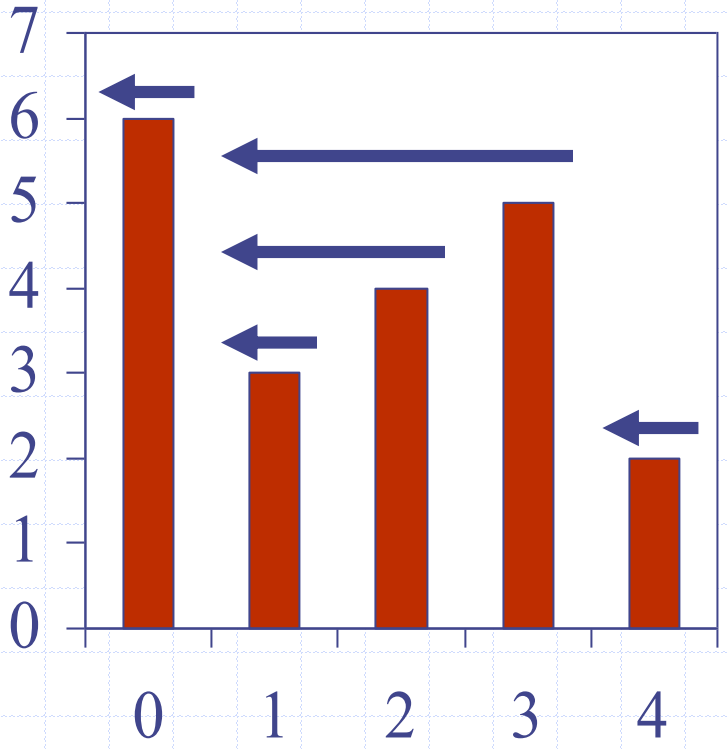
Slide by Matt Stallmann included with permission.

Operator ≤ has lower precedence than +/–



Computing Spans (not in book)

- Using a stack as an auxiliary data structure in an algorithm
- Given an an array X, the span $S[i]$ of $X[i]$ is the maximum number of consecutive elements $X[j]$ immediately preceding $X[i]$ and such that $X[j] \leq X[i]$
- Spans have applications to financial analysis
  - E.g., stock at 52-week high



X	6	3	4	5	2
S	1	1	2	3	1

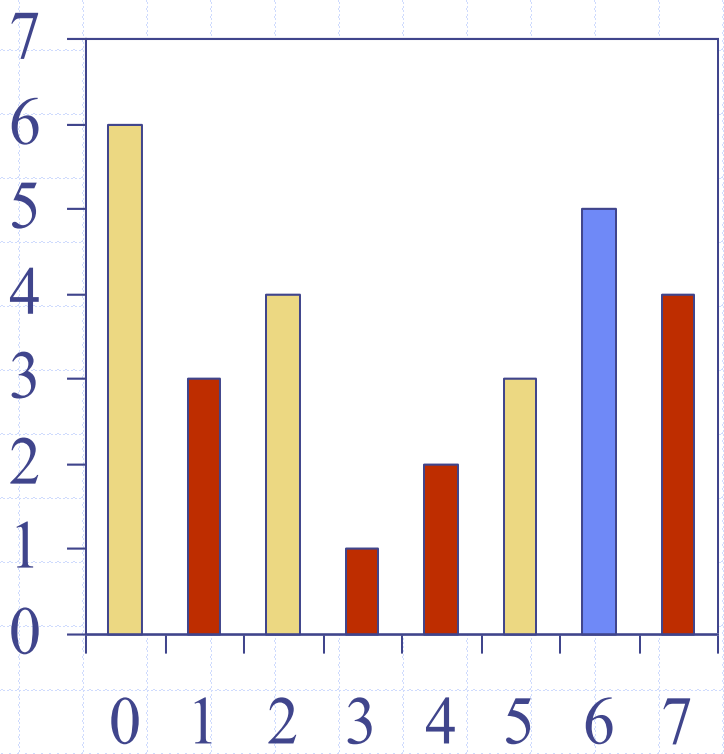
Quadratic Algorithm

Algorithm *spans1*(X, n)	
Input array X of n integers	
Output array S of spans of X	#
$S \leftarrow$ new array of n integers	n
for $i \leftarrow 0$ **to** $n - 1$ **do**	n
$s \leftarrow 1$	n
while $s \leq i \wedge X[i - s] \leq X[i]$	$1 + 2 + \ldots + (n - 1)$
$s \leftarrow s + 1$	$1 + 2 + \ldots + (n - 1)$
$S[i] \leftarrow s$	n
return S	1

- Algorithm *spans1* runs in $O(n^2)$ time

Computing Spans with a Stack

- We keep in a stack the indices of the elements visible when "looking back"
- We scan the array from left to right
  - Let i be the current index
  - We pop indices from the stack until we find index j such that $X[i] < X[j]$
  - We set $S[i] \leftarrow i - j$
  - We push x onto the stack



Linear Time Algorithm

- Each index of the array:
  - Is pushed into the stack exactly one
  - Is popped from the stack at most once
- The statements in the while-loop are executed at most n times
- Algorithm *spans2* runs in $O(n)$ time

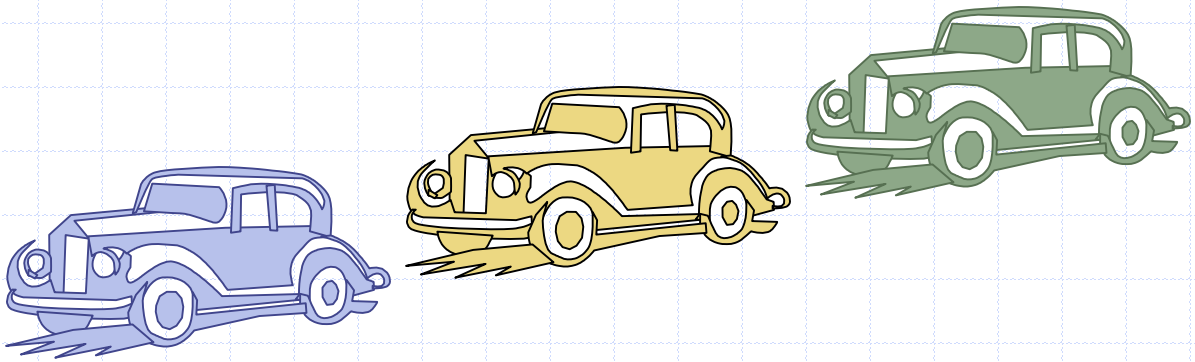
```
Algorithm spans2(X, n)                                  #
    S ← new array of n integers                         n
    A ← new empty stack                                 1
    for i ← 0 to n − 1 do                               n
        while (¬A.isEmpty() ∧
                X[A.top()] ≤ X[i] ) do                  n
            A.pop()                                     n
        if A.isEmpty() then                             n
            S[i] ← i + 1                                n
        else
            S[i] ← i − A.top()                          n
        A.push(i)                                       n
    return S                                            1
```

Linked-List Based Stack Implementation

- A Singly Linked List with only a head pointer
- Use add to head and remove from head for push and pop respectively

Modified version of the presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Queues



The Queue ADT

- The Queue ADT stores arbitrary objects
- 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
- Insertions and deletions follow the first-in first-out (FIFO) scheme
- Insertions are at the rear of the queue and removals are at the front of the queue
- Auxiliary queue operations:
  - object first(): 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 first on an empty queue returns null

Example

Operation		*Output*	*Q*
enqueue(5)	–	(5)	
enqueue(3)	–	(5, 3)	
dequeue()			
enqueue(7)			
dequeue()			
first()			
dequeue()			
dequeue()			
isEmpty()			
enqueue(9)			
enqueue(7)			
size()			
enqueue(3)			
enqueue(5)			
dequeue()			

Example

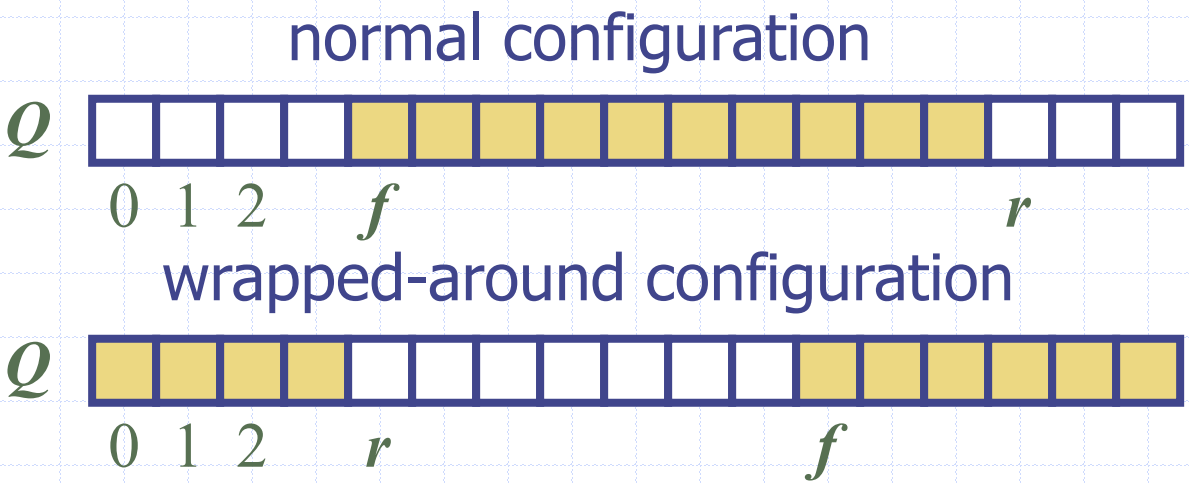
Operation		*Output*	*Q*
enqueue(5)	–	(5)	
enqueue(3)	–	(5, 3)	
dequeue()	*5*	(3)	
enqueue(7)	–	(3, 7)	
dequeue()	*3*	(7)	
first()	*7*	(7)	
dequeue()	*7*	()	
dequeue()	*null*	()	
isEmpty()	*true*	()	
enqueue(9)	–	(9)	
enqueue(7)	–	(9, 7)	
size()	*2*	(9, 7)	
enqueue(3)	–	(9, 7, 3)	
enqueue(5)	–	(9, 7,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

Applications of Queues

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

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 $r = (f + sz) \bmod N$ is the first empty slot past the rear of the queue

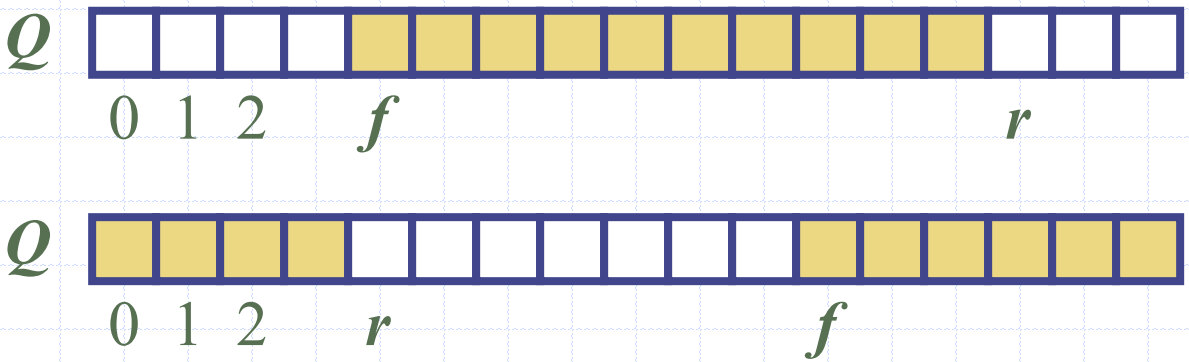


Queue Operations

- ***mod***: the modulo operation, returns the remainder of division

Algorithm *size*()
  return *sz*

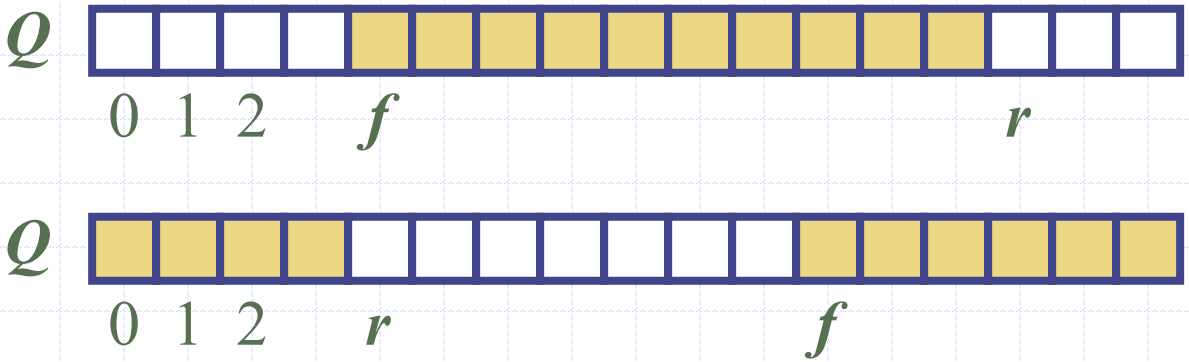
Algorithm *isEmpty*()
  return (*sz* == 0)



Queue Operations (cont.)

- Enqueue throws an exception if the array is full unless the array is dynamically grown
- This exception is implementation-dependent

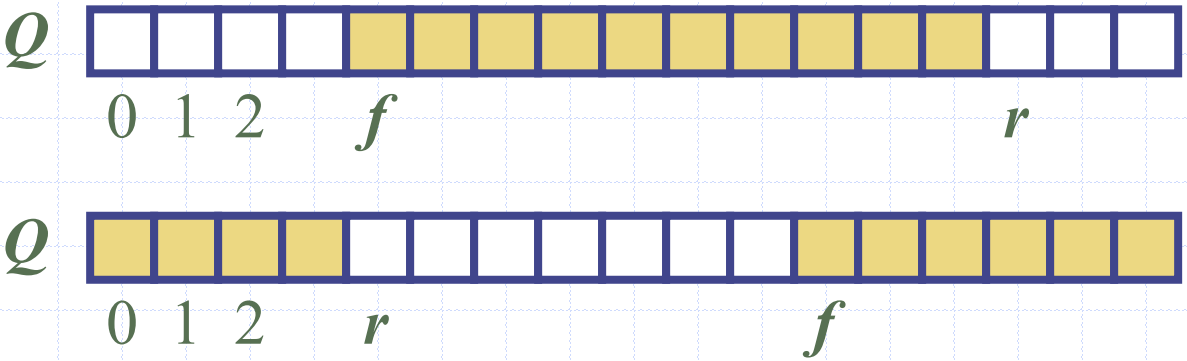
Algorithm *enqueue*(*o*)
  if *size*() = *N* – 1 **then**
    throw *IllegalStateException*
  else
    $r \leftarrow (f + sz) \bmod N$
    $Q[r] \leftarrow o$
    $sz \leftarrow (sz + 1)$



Queue Operations (cont.)

- Dequeue returns null if the queue is empty

Algorithm ***dequeue***()
  if *isEmpty*() **then**
    return *null*
  else
    $o \leftarrow Q[f]$
    $f \leftarrow (f + 1) \bmod N$
    $sz \leftarrow (sz - 1)$
    return o



http://www.cs.usfca.edu/~galles/visualization/QueueArray.html

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 e);
   E dequeue();
}
```

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
```

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    }
```

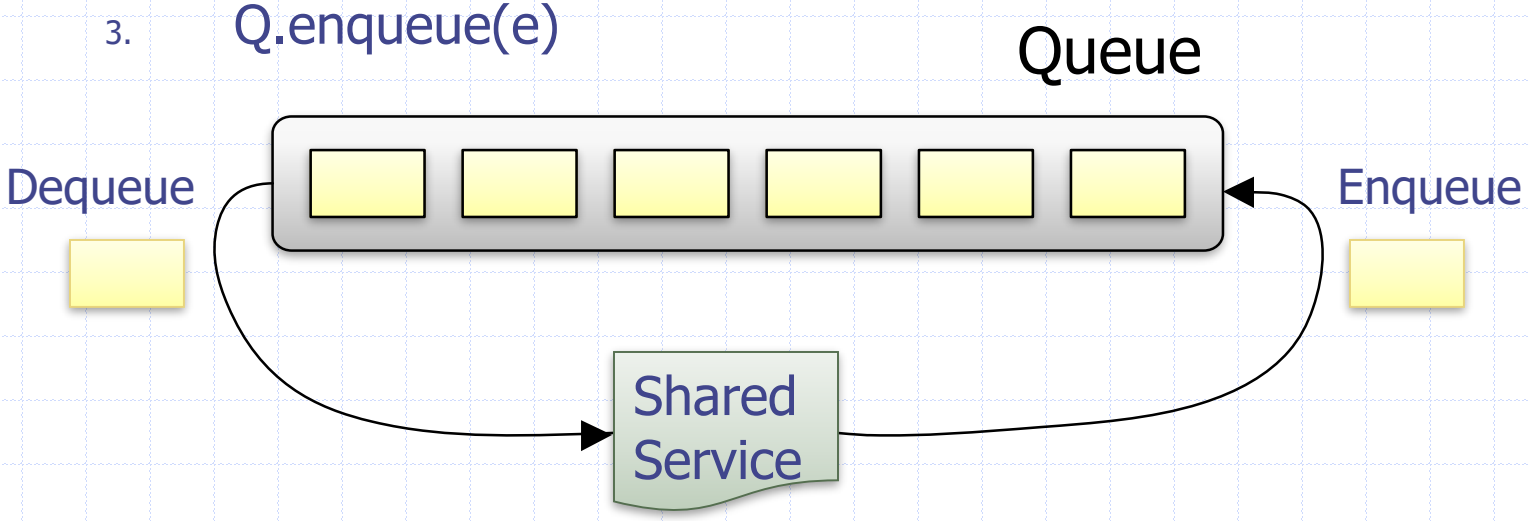
Comparison to java.util.Queue

- 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( )	

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)



Linked-List Based Queue Implementation

- A Singly Linked List with both a head and a tail pointer
- Enqueue: Add to tail (rear)
- Dequeue: Remove from head (front)

Dynamic Arrays vs Linked-Lists: V2

Linked-List:

- Fast add/remove from tail/head and in between if the node is known
- Constant time adding/removing in the above cases
- Extra storage for pointers
- No random access

Dynamic Array:

- Fast random access
  - Plus Cache Locality helps!
- No extra storage
- Amortized constant time adding/removing
  - Issues with worst case upper bound of a single operation
  - Depends on OS memory management against fragmentation

Fun Fact: Linux kernel is full of linked lists!

Example

- Online Insertion Sorting
- Let there be n elements in the storage, sorted in ascending order

- A new element comes, what is the complexity of inserting this element:
  - If storage is an array?
    - O(n) (or O(log(n)) if you do binary search) steps to find the place, O(n) steps to insert: O(n)
  - If storage is a list?
    - O(n) steps to find the place, O(1) steps to insert: O(n)
  - In practice, the array implementation is slower

Double Ended Queue

- Generalization of queue, elements can be added to or removed from either the front (head) or back (tail)

- Often abbreviated as ***deque*** (not to be confused with the dequeue operation!), pronounced as *deck*

The Deque ADT

- The Deque ADT stores arbitrary objects
- Main operations:
  - addFront(object): inserts an element at the front of the deque
  - addRear(object): inserts an element at the back of the deque
  - object removeFront(): removes and returns the element at the front of the deque
  - object removeRear(): removes and returns the element at the back of the deque
- Auxiliary deque operations:
  - object first(): returns the element at the front without removing it
  - object last(): 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

Implementations

- As a doubly-linked list
  - E.g. Java *LinkedList* Class
- As a dynamic array
  - E.g. Java *ArrayDeque* Class

- More information in the problem session!