Stacks I (Chapter 5)

Eleni Mangina

Room B2.05
School of Computer Science and Informatics
University College Dublin, Ireland

Focus Shift…

- **Data Type:** a mechanism for classifying the type of data that a variable (or parameter) may hold.
  - E.g. int, double, boolean, …
- **Data Structure:** A composite data type that consists of fields and which represents data in a program (in OOP, this is part of a class).
  - E.g. Student, Module, Course, …
- **Abstract Data Type:** a generalized approach to representing and manipulating certain types of data.
  - ADT = Data Structure + Algorithms

Abstract Data Types

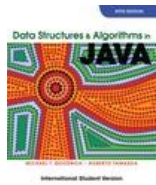
- ADTs are concepts not implementations.
  - Normally described as a functional specification.
  - Underpinned by multiple **implementation strategies**.

- They identify:
  - the **characteristics of the data** that may be stored within the data structure, and
  - the **associated operations** that may be applied to that data structure (e.g. addition / removal of items)

- Key Issues:
  - Understanding what potential implementation strategies exist.
  - Knowing which strategy to use for a given problem.

Abstract Data Types (ADTs)

- An abstract data type (ADT) is an abstraction of a data structure
- An ADT specifies:
  - Data stored
  - Operations on the data
  - Error conditions associated with operations
- Example: ADT modeling a simple stock trading system
  - The data stored are buy/sell orders
  - The operations supported are
    - order buy(stock, shares, price)
    - order sell(stock, shares, price)
    - void cancel(order)
  - Error conditions:
    - Buy/sell a nonexistent stock
    - Cancel a nonexistent order

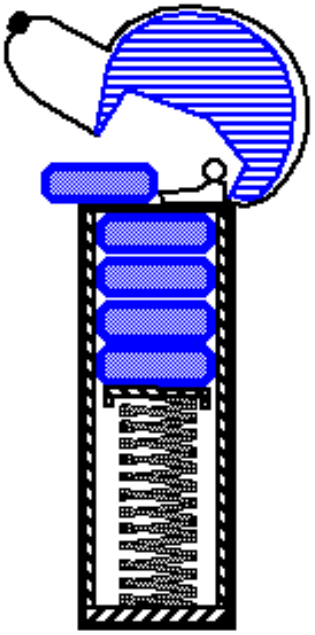


Approach (Version 1)

- Conceptual Overview
  - What the ADT is trying to achieve
- Functional Specification
  - What operations exist w.r.t. the ADT
- Implementation Strategies
  - Alternative approaches
  - Benefits / Limitations

Stacks: Concept

- A stack is a container of objects / values.
- Insertion and removal based on the last-in-first-out (LIFO) principle.
  - Objects can be inserted at any time, but only the last (the most-recently inserted) object can be removed.
- Terminology:
  - Items are “Pushed” onto the stack (insertion).
  - Items can be “Popped” off the stack (removal).
  - The “Top” of the stack is the last item pushed
- A PEZ® dispenser as an analogy…



Stacks: Functional Specification

- Stacks should work with objects.

- Core Operations:
  - push(o): Inserts object o onto top of stack
  - pop(): Removes the top object of stack and returns it

- Support Operations:
  - 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

Stack Interface in Java

- Java interface corresponding to our Stack ADT
- Requires the definition of class EmptyStackException
- Different from the built-in Java class java.util.Stack

```
public interface Stack<E> {

   public int size();

   public boolean isEmpty();

   public E top()
         throws EmptyStackException;

   public void push(E element);

   public E pop()
         throws EmptyStackException;
}
```

A Stack Interface in Java

```
/** * Interface for a stack: a
   collection of objects that
*are inserted and removed according
   to the last-in first-
*out principle. This interface
   includes the main methods
*of java.util.Stack.
* @author Roberto Tamassia
@author Michael Goodrich
@see EmptyStackException */
public interface Stack<E> {
/** * Return the number of elements
   in the stack.
@return number of elements in the
   stack. */
   public int size();
   /** * Return whether the stack is
   empty.
   * @return true if the stack is
   empty, false otherwise. */
   public boolean isEmpty();
```

```
/** * Inspect the element at the top of the
stack.
          * @return top element in the stack.
          * @exception EmptyStackException if
the stack is empty. */
          public E top() throws
EmptyStackException;
          /** * Insert an element at the top
of the stack.
          * @param element to be inserted. */
          public void push (E element);
          /** * Remove the top element from
the stack.
          * @return element removed.
          * @exception EmptyStackException if
the stack is empty. */
          public E pop() throws
EmptyStackException ; }

/** * Runtime exception thrown when one tries
to perform operation top or
    * pop on an empty stack. */
public class EmptyStackException extends
    RuntimeException {
     public EmptyStackException(String err) {
    super(err); } }
```

Exceptions

- Attempting the execution of an operation of ADT may sometimes cause an error condition, called an exception
- Exceptions are said to be "thrown" by an operation that cannot be executed
- In the Stack ADT, operations pop and top cannot be performed if the stack is empty
- Attempting the execution of pop or top on an empty stack throws an EmptyStackException

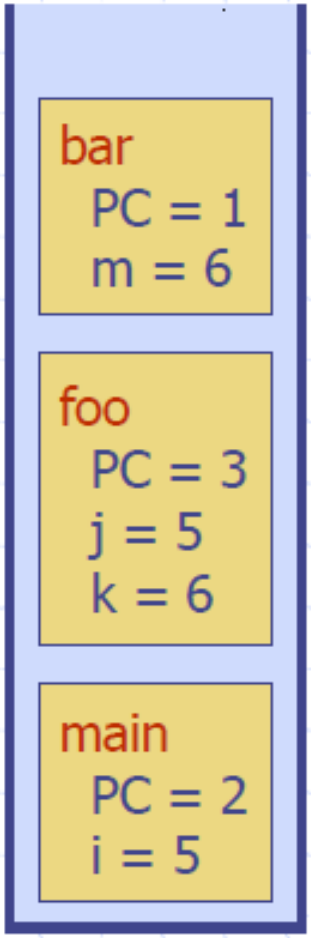
Applications of Stacks

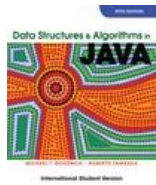
- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

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) {
  …
  }
```





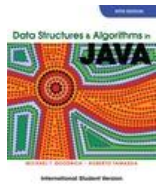
Stacks: Impl. Strategies

- Array-based Implementation:
  - Array holds the objects pushed onto the stack
    - Insertion begins at index 0.
    - Auxiliary value needed to keep track of the “top” of the stack.
  - Finite Capacity

- Link-based Implementation:
  - Objects stored in special “nodes”
  - Nodes maintain ordering information
    - Link to the next object in the stack.
  - Infinite Capacity

Array-Based Stack

- Create a stack using an array by specifying a maximum size N for our stack, e.g., N = 1000.
- The stack consists of:
  - an N-element array S and
  - an integer variable t, the index of the top element in array S.
- Illustration:



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
    throw EmptyStackException
  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

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

Array-Based Stack

Algorithm push(o):

Input: an object o

Output: none

t ← t + 1

S[t] ← o

Algorithm size():

Input: none

Output: count of objects on the stack

return t +1

Algorithm isEmpty():

Input: none

Output: true if the stack is empty, false otherwise

return t < 0

Algorithm pop():

Input: none

Output: the top object

e ← S[t]

S[t] ← null

t ← t-1

return e

Algorithm top():

Input: none

Output: the top object

return S[t]

Array-Based Stacks: Dry Runs

- View operations as atomic
  - Show the state of the array, S, and top element index, t, after each operation

- Example:

operation	t	0	1	2	3	4	5	6	7	8	9
Initial State	-1										
Push(H)	0	H									
Push(A)	1	H	A								
Push(A)	2	H	A	A							
Pop()	1	H	A								
Push(P)	2	H	A	P							
Push(P)	3	H	A	P	P						
Push(Y)	4	H	A	P	P	Y					

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

Array-Based Stacks: Impl.

- Class name: `ArrayStack`
  - Fields:
    - An array of objects, S
    - An integer, N (array size)
    - An integer, t (top index)
  - Constructors
    - Default Constructor (sets N to 1000)
    - Constructor with 1 integer parameter (used to set value of N)
  - Methods:
    - 1 per operation: methods names should match operation names – except for naming conventions (lower case first letter)
    - Implement methods based on pseudo code
- This is part of your next worksheet

Array-Based Stacks: Analysis

- Operation Running Times:

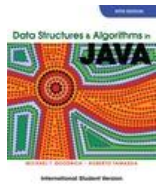
Operation	Running Time
Push(o)	O(1)
Pop()	O(1)
Top()	O(1)
IsEmpty()	O(1)
Size()	O(1)

- Issues:
  - What happens if we pop from an empty stack?
  - What happens if we push onto a full stack?

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()
      throws EmptyStackException {
    if isEmpty()
      throw new EmptyStackException
          ("Empty stack: cannot pop");
    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) …
   }
```

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>
```

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.

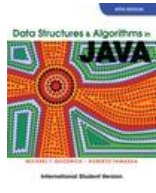
1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

Tag Matching Algorithm (in Java)

```
import java.io.*;
import java.util.Scanner;
import net.datastructures.*;
/** Simplified test of matching tags in an HTML document. */
public class HTML {
  /** Strip the first and last characters off a <tag> string. */
  public static String stripEnds(String t) {
    if (t.length() <= 2) return null;              // this is a degenerate tag
    return t.substring(1,t.length()-1);
  }
  /** Test if a stripped tag string is empty or a true opening tag. */
  public static boolean isOpeningTag(String tag) {
    return (tag.length() == 0) || (tag.charAt(0) != '/');
  }
```

Tag Matching Algorithm (cont.)

```
/** Test if stripped tag1 matches closing tag2 (first character is '/'). */
 public static boolean areMatchingTags(String tag1, String tag2) {
   return tag1.equals(tag2.substring(1)); // test against name after '/'
 }
 /** Test if every opening tag has a matching closing tag. */
 public static boolean isHTMLMatched(String[] tag) {
   Stack<String> S = new NodeStack<String>();     // Stack for matching tags
   for (int i = 0; (i < tag.length) && (tag[i] != null); i++) {
     if (isOpeningTag(tag[i]))
     S.push(tag[i]); // opening tag; push it on the stack
     else {
     if (S.isEmpty())
       return false;                    // nothing to match
     if (!areMatchingTags(S.pop(), tag[i]))
       return false;      // wrong match
     }
     }
   if (S.isEmpty()) return true; // we matched everything
   return false; // we have some tags that never were matched
 }
```



Tag Matching Algorithm (cont.)

```
public final static int CAPACITY = 1000; // Tag array size
/* Parse an HTML document into an array of html tags */
public static String[] parseHTML(Scanner s) {
  String[] tag = new String[CAPACITY]; // our tag array (initially all null)
  int count = 0;                       // tag counter
  String token;                        // token returned by the scanner s
  while (s.hasNextLine()) {
    while ((token = s.findInLine("<[^>]*>")) != null)  // find the next tag
      tag[count++] = stripEnds(token); // strip the ends off this tag
    s.nextLine(); // go to the next line
  }
  return tag; // our array of (stripped) tags
}
public static void main(String[] args) throws IOException { // tester
  if (isHTMLMatched(parseHTML(new Scanner(System.in))))
    System.out.println("The input file is a matched HTML document.");
  else
    System.out.println("The input file is not a matched HTML document.");
}
}
```

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)

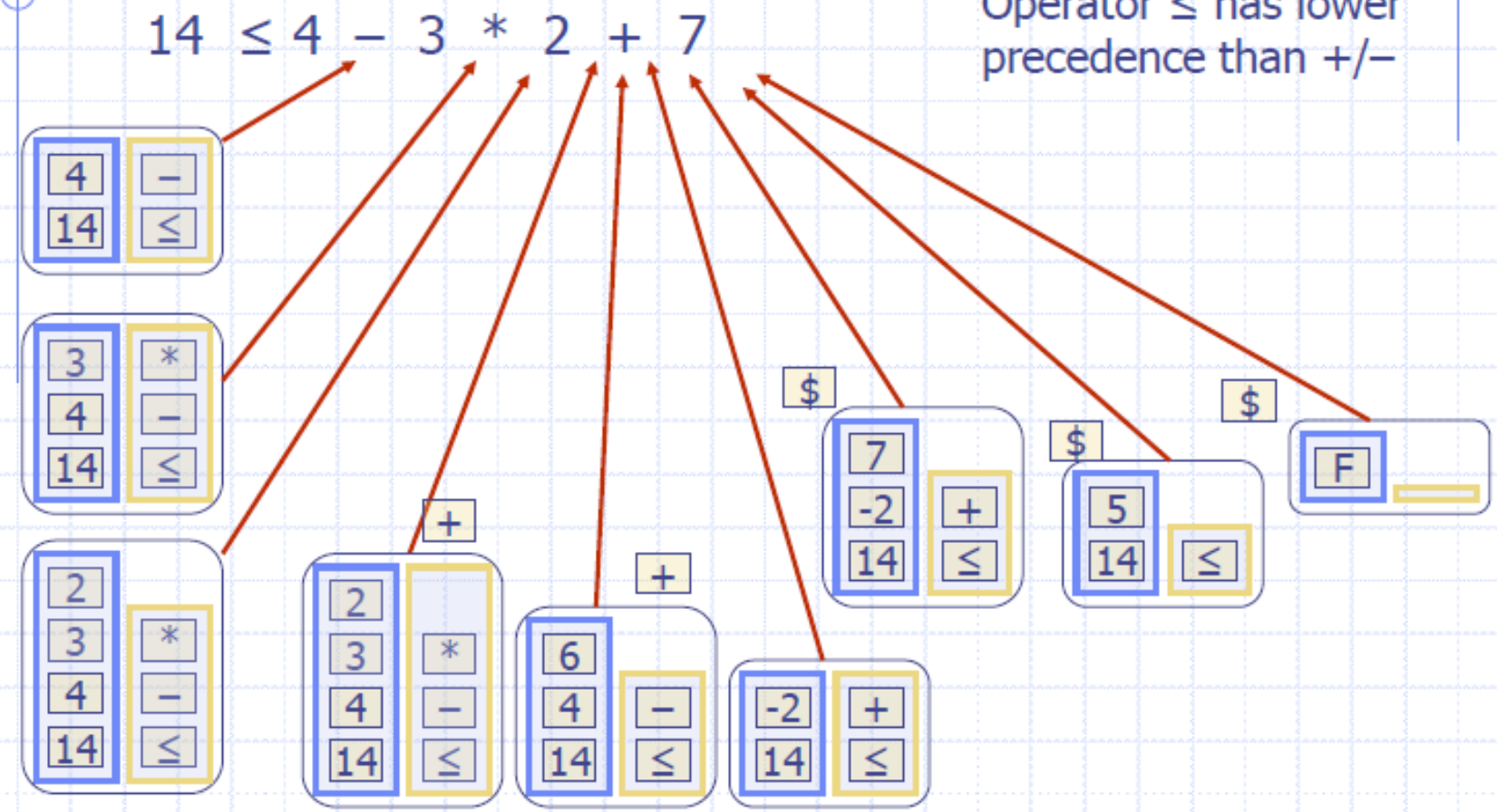
Output: the value of the expression

```
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

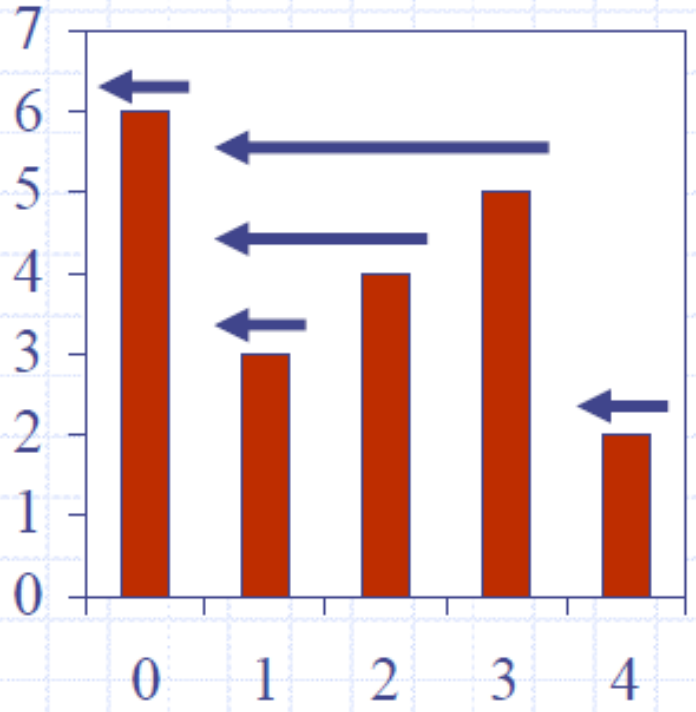
Slide by Matt Stallmann included with permission.

Operator ≤ has lower precedence than +/–

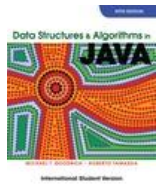


Computing Spans (not in book)

- 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



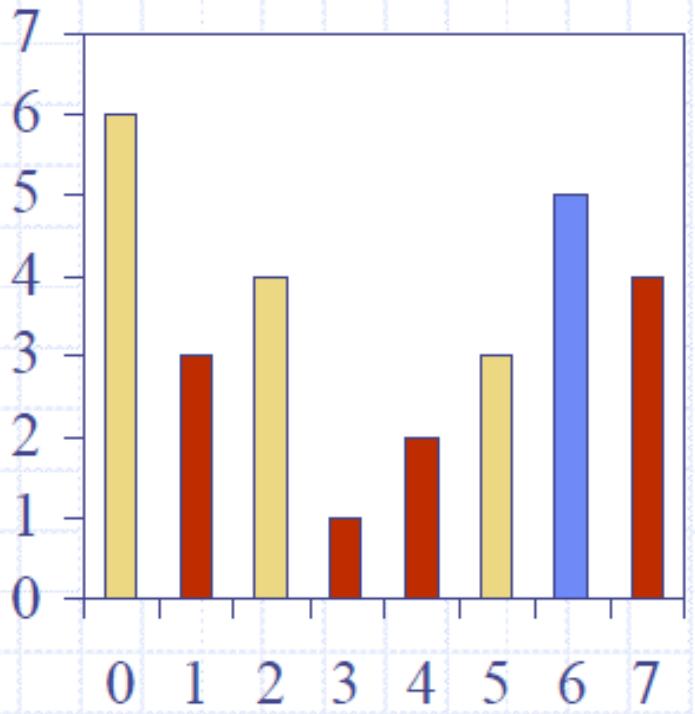
Quadratic Algorithm

```
Algorithm spans1(X, n)                          #
    Input array X of n integers
    Output array S of spans of X                #
    S ← new array of n integers                 n
    for i ← 0 to n − 1 do                       n
        s ← 1                                   n
        while s ≤ i ∧ X[i − s] ≤ X[i]           1 + 2 + …+ (n − 1)
            s ← s + 1                           1 + 2 + …+ (n − 1)
        S[i] ← s                                n
    return S                                    1
```

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

Computing Spans with a Stack

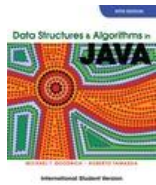
- We keep in a stack the indices of the elements visible when "looking back"
- We scan the array from left to right
  - Let i be the current index
  - We pop indices from the stack until we find index j such that $X[i] < X[j]$
  - We set $S[i] \leftarrow i - j$
  - We push x onto the stack



Linear 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
```



Insert: Working with Objects

- Note: Stacks works with Objects, and not primitive data types…
  - This means that we cannot use int, long, char, double, …
- Java has object equivalents for all primitive data types:
  - Integer == int, Double == double, Long == long, Character == char
- Conversing to objects:
  - E.g. `Integer year = new Integer(2008);`
- Extracting values from objects:
  - E.g. `int intYear = year.intValue();`

Insert: Working with Objects

- When you declare a parameter of type object, this means that you can pass **any object** as an argument.
  - For example, if we had a variable `stack` that holds a reference to an `ArrayStack` object, then we can do the following:
    - `stack.push("Hello World");`
    - `stack.push(new Integer(55));`
    - `stack.push(new URL("http://www.google.com"));`

- Finally, when you declare a return value of type object, this means that the method can return any type of object.
  - So, when we get the return value, we must convert it to be a reference to the appropriate kind of object (this is known as **casting**):
    - `URL url = (URL) stack.pop();`
    - `Integer integer = stack.pop ();`
    - `String string = stack.pop();`

Array based Implementation

```
/** * Implementation of the stack ADT using a fixed-length array. An
*exception is thrown if a push operation is attempted when the size
*of the stack is equal to the length of the array. This class
*includes the main methods of the built-in class java.util.Stack. */
public class ArrayStack<E> implements Stack<E> {
        protected int capacity; // The actual capacity of the stack array
        public static final int CAPACITY = 1000; // default array capacity
        protected E S[]; // Generic array used to implement the stack
        protected int top = -1; // index for the top of the stack
        public ArrayStack() { this(CAPACITY); // default capacity }
        public ArrayStack(int cap) {
                capacity = cap; S = (E[]) new Object[capacity];
                // compiler may give warning, but this is ok }
        public int size() { return (top + 1); }
        public boolean isEmpty() { return (top < 0); }
        public void push(E element) throws FullStackException {
                if (size() == capacity) throw new
                 FullStackException("Stack is full."); S[++top] = element; }
        public E top() throws EmptyStackException {
                if (isEmpty()) throw new EmptyStackException
                ("Stack is empty."); return S[top]; }
        public E pop() throws EmptyStackException {
                E element; if (isEmpty()) throw new
                EmptyStackException("Stack is empty.");
                element = S[top]; S[top--] = null;
                 // dereference S[top] for garbage collection. return element;
        }
```

```
public String toString() {
                String s; s = "["; if (size() > 0) s+= S[0];
                if (size() > 1) for (int i = 1; i <= size()-1; i++) { s += ", " + S[i]; }
                 return s + "]"; }
// Prints status information about a recent operation and the stack.
 public void status(String op, Object element) {
                System.out.print("------> " + op); // print this operation
                System.out.println(", returns " + element);
                // what was returned
                System.out.print("result: size = " + size() + ",
                                        isEmpty = " + isEmpty());
                System.out.println(", stack: " + this);
                 // contents of the stack }
 /** * Test our program by performing a series of operations on stacks,
  * printing the operations performed, the returned elements and the
* contents of the stack involved, after each operation. */
public static void main(String[] args) {
Object o; ArrayStack<Integer> A = new ArrayStack<Integer>();
A.status("new ArrayStack<Integer> A", null);
A.push(7); A.status("A.push(7)", null); o = A.pop();
A.status("A.pop()", o);  A.push(9);
A.status("A.push(9)", null); o = A.pop(); A.status("A.pop()", o);
ArrayStack<String> B = new ArrayStack<String>();
B.status("new ArrayStack<String> B", null); B.push("Bob");
B.status("B.push(\"Bob\")", null); B.push("Alice");
B.status("B.push(\"Alice\")", null); o = B.pop();
B.status("B.pop()", o); B.push("Eve");
B.status("B.push(\"Eve\")", null); } }
```

Example output? Run the code