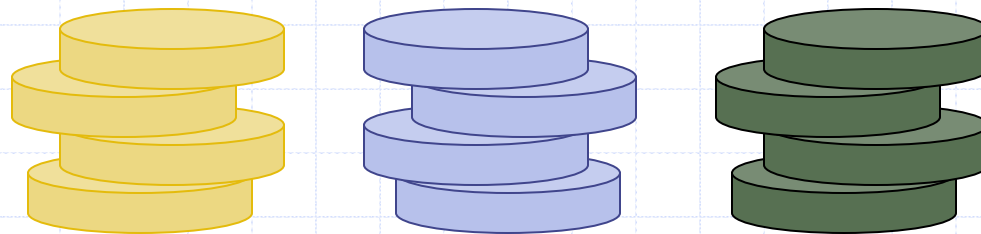


Stacks



Outline and Reading

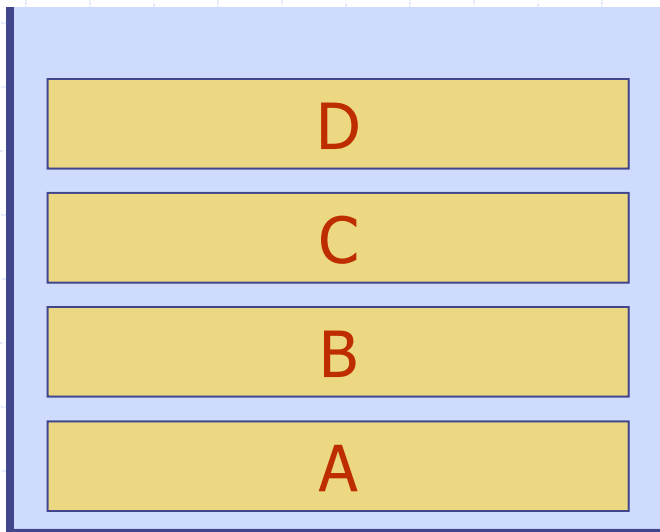
- The Stack ADT (§5.1.1)
- Examples of Algorithms using Stacks (§5.1.5)
- Array-based implementation (§5.1.2)
- Extendable Array (§6.1.5)

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 the operations
- Think of an ADT as an interface, and the operations are its methods
- Example: ADT modeling a simple stock trading system
 - Data stored: buy/sell orders
 - Operations supported:
 - order **buy**(stock, shares, price)
 - order **sell**(stock, shares, price)
 - boolean **isExecuted**(order)
 - boolean **cancel**(order)
 - Error conditions:
 - Buy/sell a nonexistent stock
 - Buy/sell a negative number of shares or with negative price
 - Cancel executed/nonexistent order

The Stack ADT

- The **stack** ADT stores arbitrary objects
- Insertions and deletions follow the last-in first-out scheme
- Think of a spring-loaded plate dispenser



- Stack operations:
 - **push(object)**: inserts an element
 - **object pop()**: removes and returns the last inserted element
 - **object top()**: returns the last inserted element without removing it
 - **integer size()**: returns the number of elements stored in the stack
 - **boolean isEmpty()**: indicates whether the stack has no elements

Errors and Exceptions

- Attempting the execution of an operation of ADT may sometimes cause an error condition
- Error conditions are called **exceptions** in Java (remember `NullPointerException`?)
- Exceptions are said to be “thrown” by a method where an error occurs
- An exception is an object that extends the built-in class `java.lang.RuntimeException`
- In the stack ADT, operations **pop** and **top** cause an error if the stack is empty
- We define class `EmptyStackException` to denote the exception thrown when attempting the execution of **pop** or **top** on an empty stack

Stack Interface in Java

- In Java, an ADT is expressed by means of an **interface**
- Java interface corresponding to our stack ADT
- Different from the built-in Java class `java.util.Stack`
- The stack supports generic elements thanks to the use of parameterized type `E` for the elements stored by the 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;  
}
```

Using a Stack in Java

- `ArrayStack<E>` is an implementation of the `Stack<E>` interface
- Note the substitution of the type `String` for the generic `E` element
- Three names are pushed onto the new stack in the method
- Note that the `top()` call will retrieve the string `"Yoonha"` without removing it
- `"Peter"` remains in the stack at the conclusion of the method

```
public void makeTASStack() {  
    Stack<String> myStack = new  
        ArrayStack<String>();  
  
    myStack.push("Peter");  
    myStack.push("Tatyana");  
    myStack.push("Yoonha");  
  
    String topTA = myStack.top();  
    String ykim1 = myStack.pop();  
    String tdyshlov = myStack.pop();  
  
}
```

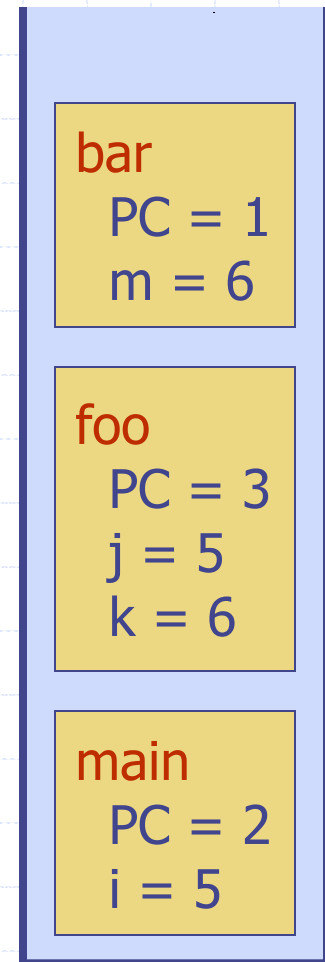
Applications of Stacks

- Direct applications
 - Page-visited history in a Web browser
 - Undo sequence in a text editor
 - Chain of method calls in the Java Virtual Machine (JVM)
- 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 using a stack
- When a method is called, the JVM pushes on the stack a **frame** containing
 - Local variables and return value
 - Program counter (PC), 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

```
main() {  
    int i = 5;  
    foo(i);  
}  
  
foo(int j) {  
    int k;  
    k = j+1;  
    bar(k);  
}  
  
bar(int m) {  
    ...  
}
```



Array-based Stack

- We can use an array to implement the stack ADT
- We add elements from left to right
- A variable keeps track of the index of the top element of the stack (the highest filled slot in the array)

Algorithm *size()*

return $t + 1$

Algorithm *pop()*

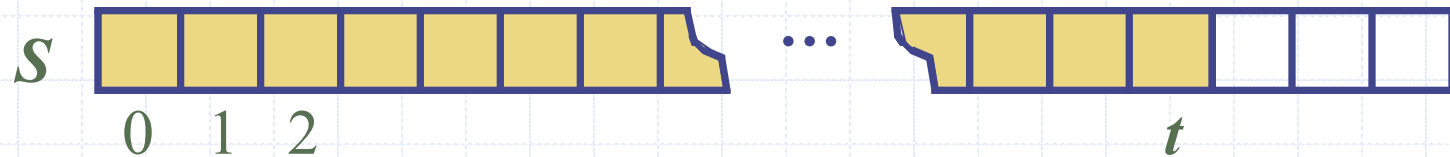
if *isEmpty()* **then**

throw *EmptyStackException*

else

$t \leftarrow t - 1$

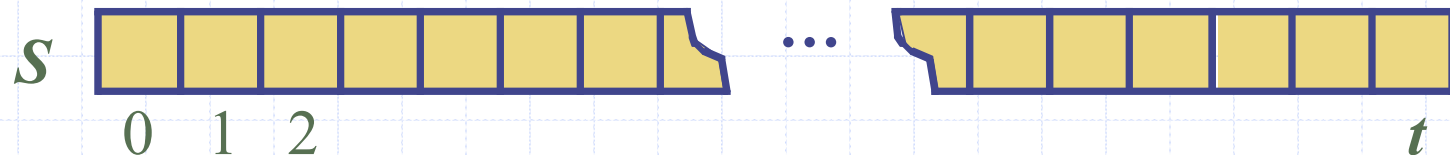
return $S[t + 1]$



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, must be added because we've chosen to use an array

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



Performance and Limitations

- Performance of the array-based implementation
 - Let N be the length of the array used by the data structure and n be the number of elements stored in the stack ($n \leq N$)
 - The space used is $O(N)$, irrespectively of n
 - Each operation of the stack ADT runs in $O(1)$ time
- Limitations
 - The maximum size N 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 Stack in Java

```
public class ArrayStack<E>
    implements Stack<E> {
    // holds the stack elements
    private E[] Stack;

    // index to top elem, -1 means empty
    private int top = -1;

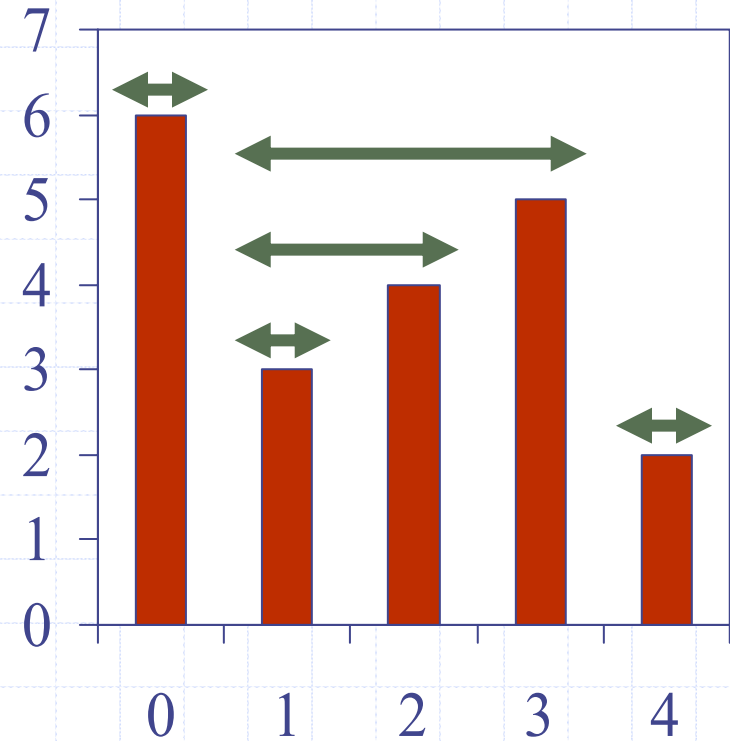
    // constructor
    public ArrayStack(int capacity) {
        Stack = (E[]) new Object[capacity];
    }
}
```

```
public E pop()
    throws EmptyStackException {
    if isEmpty()
        throw new
            EmptyStackException
            ("Empty stack: cannot pop");
    E topE = S[top];
    // facilitates garbage collection
    S[top] = null;
    top = top - 1;
    return topE;
}
```

Note: throwing an exception will cause a method to end, like a return

Computing Spans

- We show how to use a stack as an auxiliary data structure in an algorithm
- Given an array X , the span $S[i]$ of $X[i]$ is the maximum number of consecutive elements $X[j]$ such that
 - $0 \leq j \leq i$
 - $X[j] \leq X[i]$
- Spans have applications to financial analysis
 - E.g., stock at 16-month high



$S[i]$	1	1	2	3	1
$X[i]$	6	3	4	5	2
i	0	1	2	3	4

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

for $i \leftarrow 0$ **to** $n - 1$ **do**

$j \leftarrow i$

while $(j \geq 0) \wedge (X[j] \leq X[i])$

$j \leftarrow j - 1$

$S[i] \leftarrow i - j + 1$

return S

#

n

n

n

$1 + 2 + \dots + (n + 1)$

$1 + 2 + \dots + n$

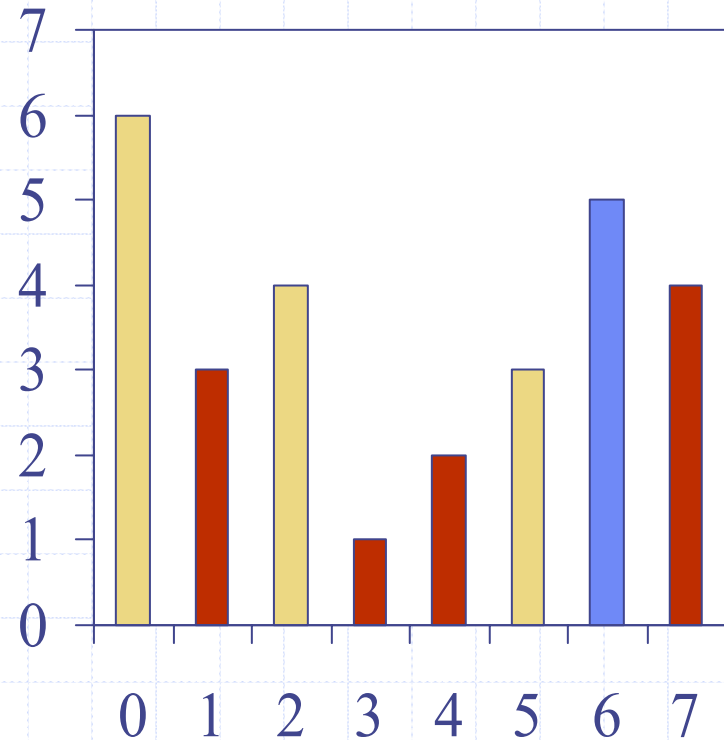
n

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 $X[j] > X[i]$
 - We set $S[i] \leftarrow i - j$
 - We push i onto the stack
 - Special case when the stack becomes empty



Linear Algorithm

- Each index of the array
 - Is pushed into the stack exactly once
 - Is popped from the stack at most once
- Thus, each statement in the while-loop is executed at most n times
- Algorithm *spans2* runs in $O(n)$ time

Algorithm <i>spans2</i> (X, n)	#
$S \leftarrow$ new array of n integers	n
$A \leftarrow$ new empty stack	1
for $i \leftarrow 0$ to $n - 1$ do	n
while $\neg A.isEmpty() \wedge$ $(X[top()] \leq X[i])$ do	n
$j \leftarrow A.pop()$	n
if $A.isEmpty()$ then	n
$S[i] \leftarrow i + 1$	n
else	
$S[i] \leftarrow i - j$	n
$A.push(i)$	n
return S	1

Growable Array-based Stack

- In a push operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one
- How large should the new array be?
 - incremental strategy: increase the size of the array by a constant c
 - doubling strategy: double the size of the array

```
Algorithm push(o)  
if  $t = S.length - 1$  then  
     $A \leftarrow$  new array of  
        size ...  
    for  $i \leftarrow 0$  to  $t$  do  
         $A[i] \leftarrow S[i]$   
         $S \leftarrow A$   
 $t \leftarrow t + 1$   
 $S[t] \leftarrow o$ 
```

Comparison of the Strategies

- We compare the incremental strategy and the doubling strategy by analyzing the total time $T(n)$ needed to perform a series of n push operations
- We assume that we start with an empty stack represented by an array of size 1
- We call **amortized time** of a push operation the average time taken by a push over the series of operations, i.e., $T(n)/n$

Incremental Strategy Analysis

push #	2	$c + 2$	$2c + 2$	$3c + 2$...
replacement time	$c + 1$	$2c + 1$	$3c + 1$	$4c + 1$...

- The number k of array replacements is about n/c
- The total time $T(n)$ of a series of n push operations is proportional to

$$\begin{aligned}n + k + c + 2c + 3c + 4c + \dots + kc &= \\n + k + c(1 + 2 + 3 + \dots + k) &= \\n + k + ck(k + 1)/2 &= \end{aligned}$$

- Since c is a constant and k is about n/c , we have that $T(n)$ is $O(n^2)$
- The **amortized time** of a push operation is $O(n)$

Doubling Strategy Analysis

- The number of times k we replace the array about $\log_2 n$
- Thus, the total time $T(n)$ of a series of n push operations is proportional to

$$n + (1 + 2 + 4 + 8 + \dots + 2^k) =$$

$$n + (2^{k+1} - 1) = n + (2n - 1) = 3n - 1$$

- $T(n)$ is linear, that is, $T(n)$ is $O(n)$
- The **amortized time** of a push operation is constant, i.e., $O(1)$

geometric series

