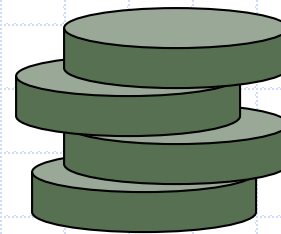
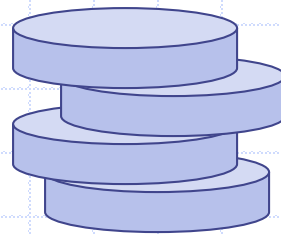
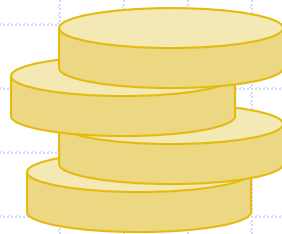


Stacks



Outline

- ◆ The Stack ADT
- ◆ Applications of Stacks
- ◆ Array-based implementation
- ◆ Growable array-based stack

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

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
- ◆ Main stack operations:
 - **push**(object o): inserts element o
 - **pop**(): removes and returns the last inserted element
- ◆ Auxiliary stack operations:
 - **top**(): returns a reference to the last inserted element without removing it
 - **size**(): returns the number of elements stored
 - **empty**(): returns a Boolean value indicating whether no elements are stored

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
- Saving local variables when one function calls another, and this one calls another, and so on.

◆ Indirect applications

- Auxiliary data structure for algorithms
- Component of other data structures



Let the Hunger Games begin!

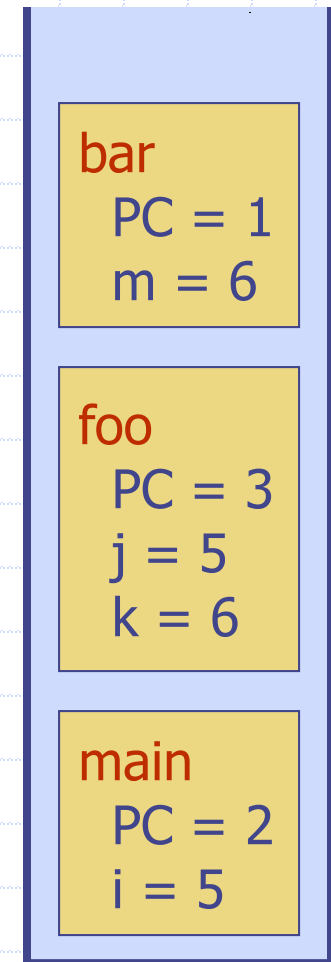
C++ Run-time Stack

- ◆ The C++ run-time system keeps track of the chain of active functions with a stack
- ◆ When a function is called, the run-time system pushes on the stack a frame containing
 - Local variables and return value
 - Program counter, keeping track of the statement being executed
- ◆ When a function returns, 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

- ◆ 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()*

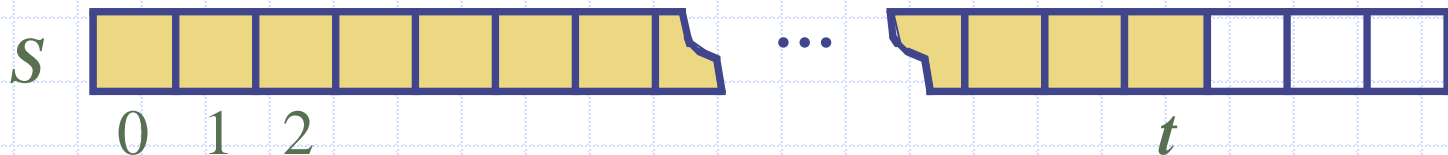
if *isEmpty()* **then**

throw *EmptyStackException*

else

$t \leftarrow t - 1$

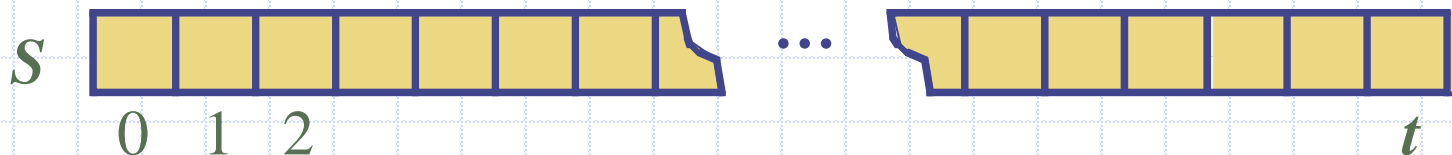
return $S[t + 1]$



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

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



Performance and Limitations

◆ Performance

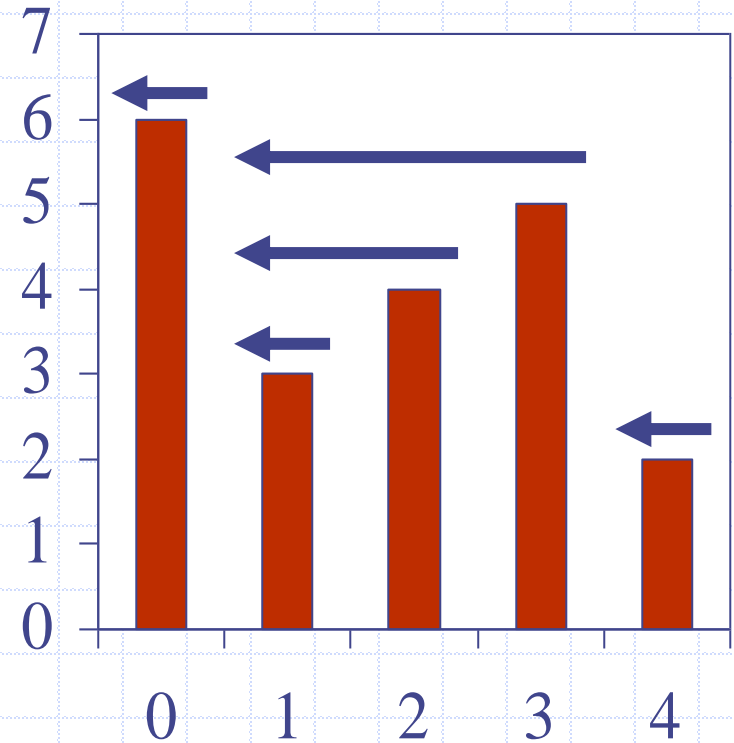
- Let n be the number of elements in the stack
- The space used is $O(n)$
- Each operation runs in time $O(1)$

◆ 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

Computing Spans

- ◆ We show how to use a stack as an auxiliary data structure in an algorithm
- ◆ Spans have applications to financial analysis
 - E.g., stock at 52-week high
- ◆ Given 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 $X[j] \leq X[i]$



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

Naïve Compute Spans 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**

$s \leftarrow 1$

while $s \leq i \wedge X[i - s] \leq X[i]$

$s \leftarrow s + 1$

$S[i] \leftarrow s$

return S

#

n

n

n

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

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

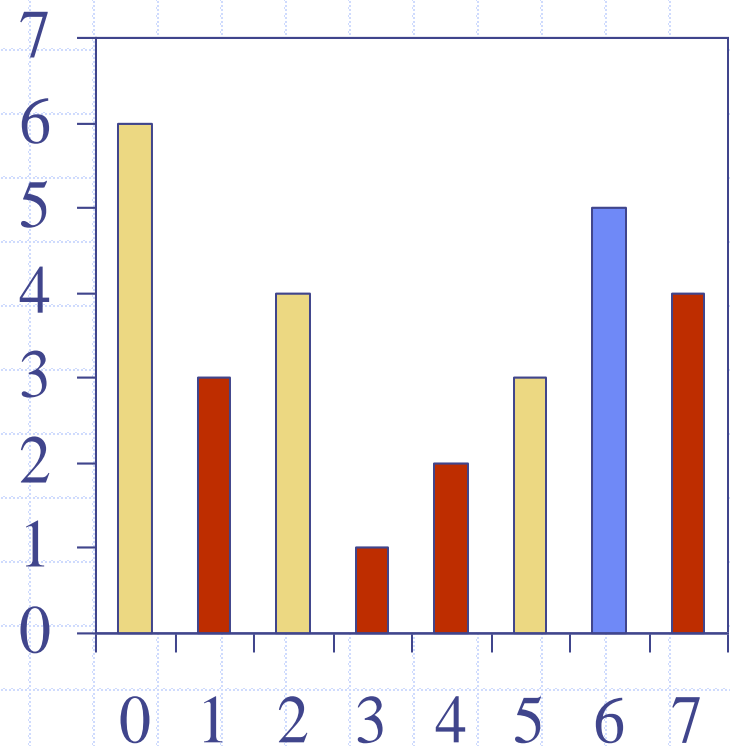
n

1

- ◆ [TEAMS] What is the big-Oh?
- ◆ 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 i 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 <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	
$j \leftarrow top()$	n
$S[i] \leftarrow i - j$	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 by a constant c
 - doubling strategy: double the size

```
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

- ◆ We replace the array $k = n/c$ times
- ◆ The total time $T(n)$ of a series of n push operations is proportional to

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

- ◆ Since c is a constant, $T(n)$ is $O(n + k^2)$, i.e., $O(n^2)$
- ◆ The amortized time of a push operation is
 - $O(n)$

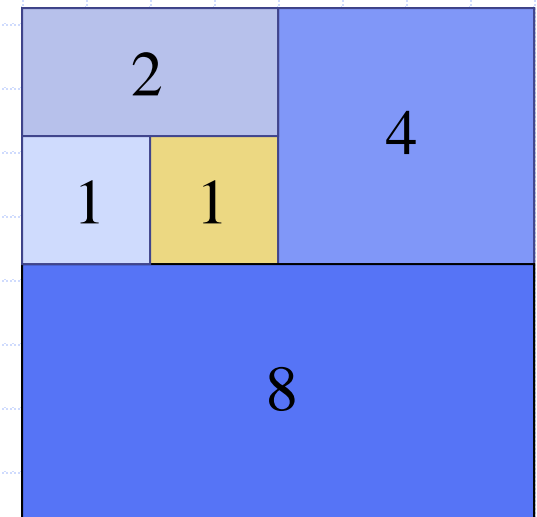
Doubling Strategy Analysis

- ◆ We replace the array $k = \log_2 n$ times
- ◆ The total time $T(n)$ of a series of n push operations is proportional to

$$\begin{aligned} n + 1 + 2 + 4 + 8 + \dots + 2^k \\ = n + 2^{k+1} - 1 \\ = 2n - 1 \end{aligned}$$

- ◆ $T(n)$ is $O(n)$
- ◆ The amortized time of a push operation is
 - $O(1)$

geometric series



Stack Interface in C++

- ◆ Interface corresponding to our Stack ADT
- ◆ Requires the definition of class `EmptyStackException`
- ◆ Similar to STL `vector`
- ◆ STL construct is `stack`

```
template <typename Object>
class Stack {
public:
    int size();
    bool empty();
    Object& top()
        throw(EmptyStackException);
    void push(Object o);
    Object pop()
        throw(EmptyStackException);
};
```

Array-based Stack in C++

```
template <typename Object>
class ArrayStack {
private:
    int capacity;      // stack capacity
    Object *S;         // stack array
    int top;           // top of stack
public:
    ArrayStack(int c) {
        capacity = c;
        S = new Object[capacity];
        t = -1;
    }
```

```
    bool empty()
    { return (t < 0); }

    Object pop()
    {
        throw(EmptyStackException) {
            if(isEmpty())
                throw EmptyStackException
                    ("Access to empty stack");
            return S[t--];
        }
    }
    // ... (other functions omitted)
```