

# Stacks

*Dr. Aiman Hanna*

**Department of Computer Science & Software Engineering  
Concordia University, Montreal, Canada**

**These slides have been extracted, modified and updated from original slides of :**

**Data Structures and Algorithms in Java, 5th edition. John Wiley & Sons, 2010. ISBN 978-0-470-38326-1.**

**Data Structures and the Java Collections Framework by William J. Collins, 3rd edition, ISBN 978-0-470-48267-4.**

**Both books are published by Wiley.**

**Copyright © 2010-2011 Wiley**

**Copyright © 2010 Michael T. Goodrich, Roberto Tamassia**

**Copyright © 2011 William J. Collins**

**Copyright © 2011-2021 Aiman Hanna**

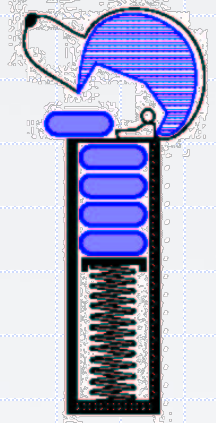
**All rights reserved**

# Abstract Data Types (ADTs)

- An abstract data type (ADT) is an abstraction/model of a data structure.
- An abstract data type is defined indirectly, only by the operations that may be performed on it. An ADT specifies:
  - ◆ Data stored
  - ◆ Operations on the data
  - ◆ Error conditions associated with operations

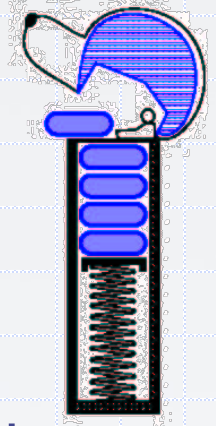
# Abstract Data Types (ADTs)

- 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



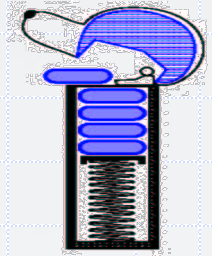
# The Stack ADT

- ❑ The **Stack** ADT stores arbitrary objects.
- ❑ Insertions and deletions follow the *last-in first-out (LIFO)* scheme. Think of a spring-loaded plate dispenser.
- ❑ Formally, a stack is an ADT that supports the following main operations:
  - **push**(object): inserts an element
  - object **pop**(): removes and returns the last inserted element
- ❑ Examples: operations of “Back” button on a browser or “undo” on text editors.



# The Stack ADT

- Secondary stack operations include:
  - 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



# The Stack ADT

- The following table shows a series of stack operations and their effects on an initially empty stack of integers:

Operation	Output	Stack Contents
push(5)	--	[5]
push(2)	--	[5, 2]
push(8)	--	[5, 2, 8]
pop()	8	[5, 2]
isEmpty()	false	[5, 2]
top()	2	[5, 2]
pop()	2	[5]
pop()	5	[]
pop()	<i>"error"</i>	[]

# The Java Built-in *Stack* Class

- Because of its importance, Java has a built-in class for the stack ([java.util.Stack](#)).
- The class has various methods, including:
  - `push()`, `pop()`, `peek()`, `empty()`, and `size()`.
- `pop()` and `peek()` throw [EmptyStackException](#) if operations are attempted on an empty stack. .
- While this class is convenient, it is very important to know how to design and implement a Stack class from scratch.

# 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

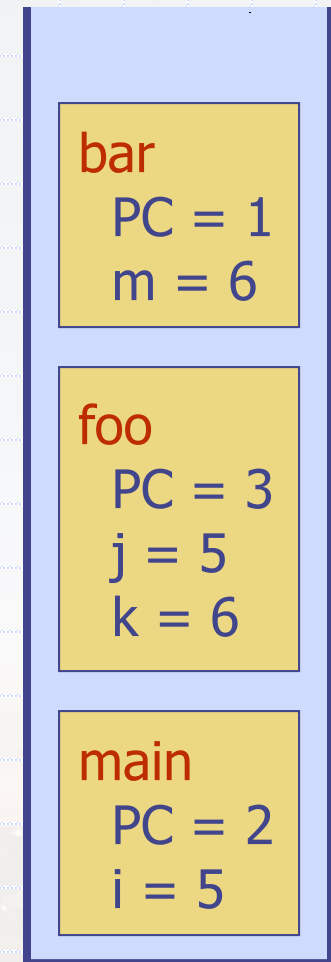
# Method Stack in the JVM

- ❑ 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) {  
    ...  
}
```



# 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.
  - Initialized to -1 when stack is created.

**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
  - We need to define this class; it is 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

# 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)

\* Notice that this is not a built-in Java implementation

# Example use in Java

```
/** A non-recursive 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();  
    }  
}
```

Time complexity is:  $O(n)$

Space complexity is:  $O(n)$

# 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) ...  
}
```



# Better Stack Implementation

- Use linked lists instead of arrays.
- No need to define a maximum size.
- When push(), add new element/node at the tail of the list.
- pop() removes the node at the tail of the list.

**What is the complexity?**

# Parentheses Matching

- Each "(", "{", or "[" must be paired with a matching ")", "}", or "]"

- correct: ( )(( )){([ ( ))}

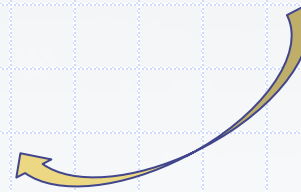
- correct: ((( ))(( )){([ ( ))}

- incorrect: )(( )){([ ( ))}

- incorrect: ({ [ ]})

- incorrect: (

Is it?



# 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**

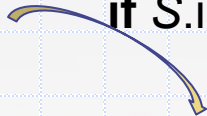
**return false** {wrong type}

**if**  $S.isEmpty()$  **then**

**return true** {every symbol matched}

**else return false** {some symbols were never matched}

Notice that item is  
removed here



# 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?

# Evaluating Arithmetic Expressions

Slide by Matt Stallmann  
included with permission.

$$14 - 3 * 2 + 7 = (14 - (3 * 2)) + 7$$

## Operator precedence

\* has precedence over +/−

Example:  $x + y * z$  is:

$x + (y * z)$  rather than  $(x + y) * z$

## Associativity

operators of the same precedence group  
evaluated from left to right

Example:  $x - y + z$  is:

$(x - y) + z$  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

Two stacks:

- opStk holds operators
- Use \$ to hold a 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**

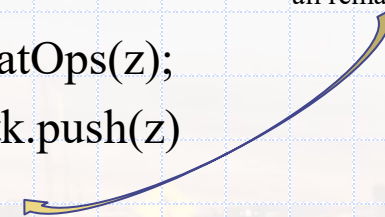
repeatOps(z);

opStk.push(z)

repeatOps(\$);

**return** valStk.top()

Force the execution of  
all remaining operators

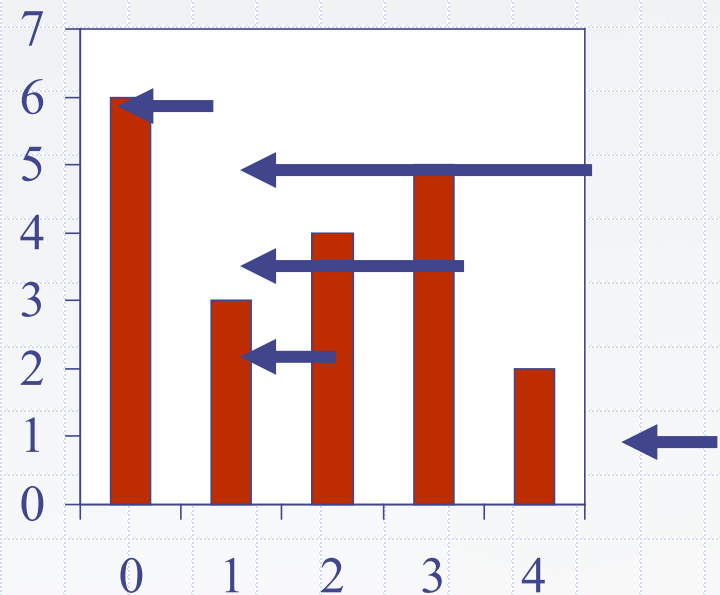


Slide by Matt Stallmann  
included with permission.

$$14 \leq 4 - 3 * 2 + 7$$


# Computing Spans (not in book)

- Using 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  $X[j]$  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

**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**  $S$

#

$n$

$n$

$n$

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

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

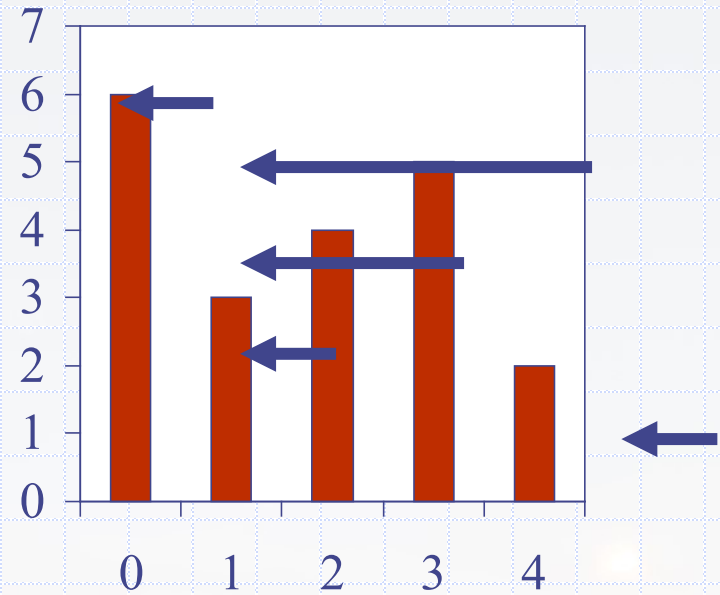
$n$

1

◆ 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 push the index as long as the one prior to it has a smaller value
  - We pop all elements otherwise
- ◆ Each index of the array
  - Is pushed into the stack exactly once
  - Is popped from the stack at most once
- Stack height for the pushed index represents the needed value

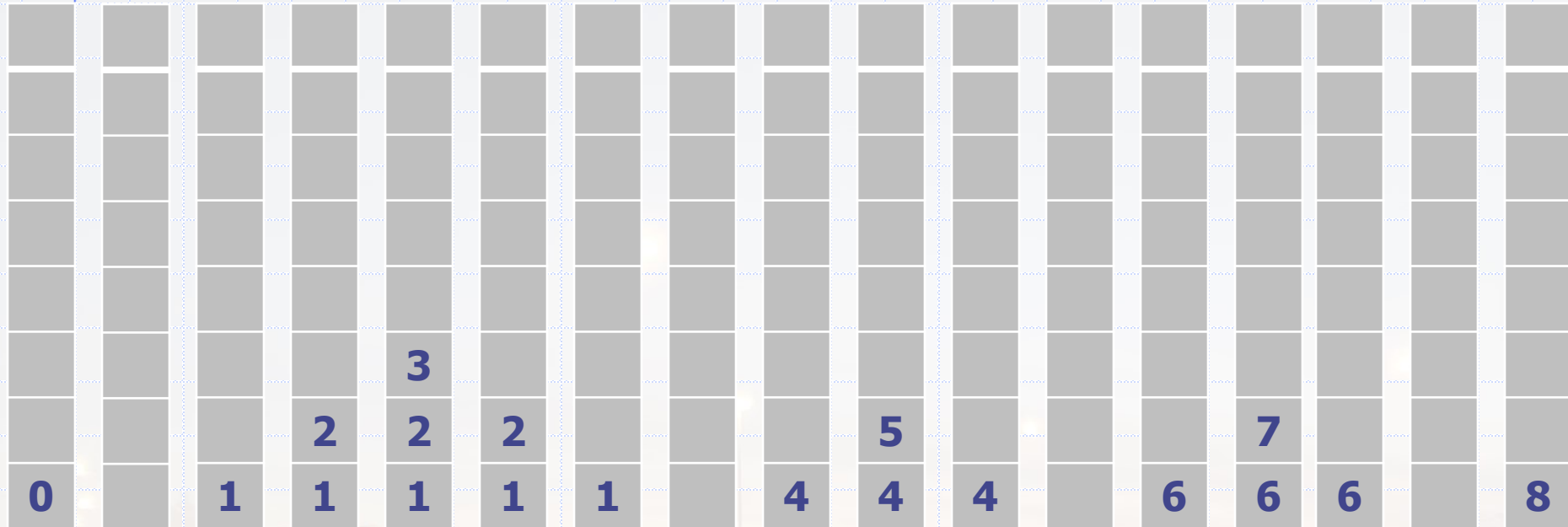


# Computing Spans with a Stack

□ Example

index  $X$   
Value

0	1	2	3	4	5	6	7	8
60	30	40	50	20	70	30	80	20



$S$

Stacks  
1 1 2 3 1 2 1 2 1

Stacks

# Linear Algorithm

- ◆ We iterate on the array  $n$  times
- ◆ Each index is then pushed and popped at most once, which totals to  $n + n$
- ◆ We record the value at each index of the resulted Span array, which totals to  $n$  times
- ◆ Consequently algorithm *spans2* has a complexity of  $O(n)$

# Growable Array-based Stack

- ❑ In a **push()** operation, if the stack is full (no more empty locations in the array), we can throw an exception and abort/reject the operation.
- ❑ Alternatively, we can extend the array; which is actually replacing it 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 add(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 refer to the average time taken by a push() operations over the series of operations, i.e.,  $T(n)/n$ , the ***amortized time*** of a push() operation.

# Incremental Strategy Analysis

- We need to find the amortized time to perform one `push()` operation.
  - That is the total time to perform  $n$  `push()` operations /  $n$ .
- In general, we need to replace the array  $k = n/c$  times for all  $n$  `push()` to take place.
  - For instance if  $n = 100$ , and  $c = 4$ , we need to go through 25 ( $100/4$ ) replacements for all `push()` operations to take place.
  - Notice also that each replacement is larger than the previous one by  $c$ .

# Incremental Strategy Analysis

- Consequently, the total time  $T(n)$  of a series of  $n$  push() operations is proportional to:

$$n + c + 2c + 3c + 4c + \dots + kc =$$

$$n + c(1 + 2 + 3 + \dots + k) =$$

$$n + ck(k + 1)/2$$

- Since  $c$  is a constant,  $T(n)$  is  $O(n + k^2)$ , i.e.,  $O(n^2)$ . That is
- $T(n)$  is the complexity to perform  $n$  push() operations. Hence, the amortized time of one single push() operation is  $O(n)$ .



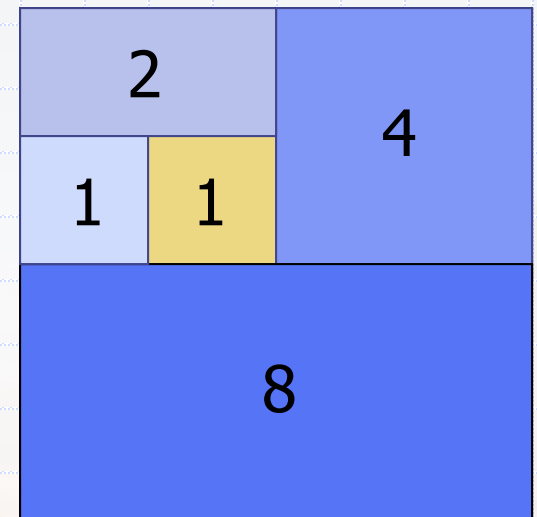
# Doubling Strategy Analysis

- We replace the array  $k = \log_2 n$  times.
  - For instance, to perform 1000 push() operations, we need to expand the array 10 times (1 -> 2 -> 4 -> 8 -> 16 -> 32 -> 64 -> 128 -> 256 -> 512 -> 1024).

- The total time  $T(n)$  of a series of  $n$  add operations is proportional to

$$\begin{aligned} n + 1 + 2 + 4 + 8 + \dots + 2^k &= \\ n + 2^{k+1} - 1 &= n + 2n - 1 = \\ 3n - 1. \end{aligned}$$

geometric series



# Doubling Strategy Analysis

- Consequently,  $T(n)$  (which is needed to perform  $n$  `push()` operations) is  $O(n)$ .
- Hence, the amortized time of a single `push()` operation is  $O(1)$ .

geometric series

