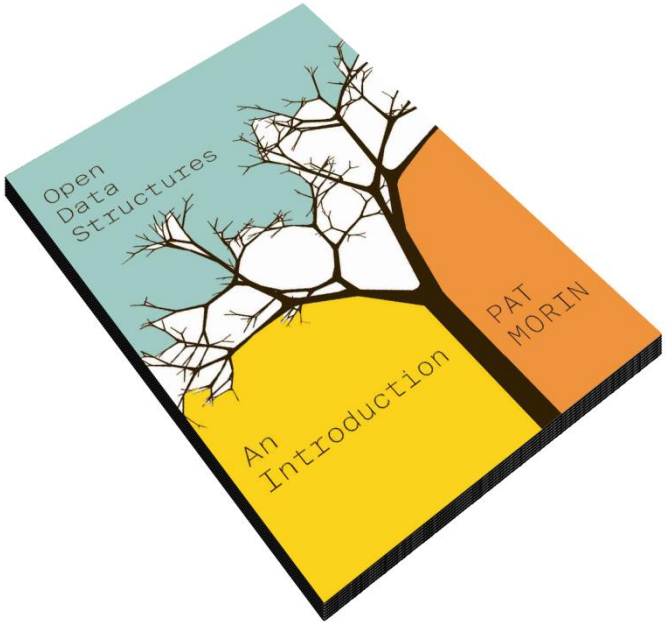
COMP2402
Abstract Data Types and Algorithms

Stack / List Operations Using an Array

Open Data Structures in Java

by Pat Morin

Chapter 2.1



Fundamental and Supporting Operations

the Abstract Data Type known as the "Stack"
Guarantees the following Fundamental Operations:

`push(o)`
Insert the Object o at the Top of the Stack

`pop()`
Remove and Return the Object from the Top of the Stack

Fundamental and Supporting Operations

the Abstract Data Type known as the "Stack"
Typically Has the following Supporting Operations:

`size()`
Return the Number of Elements in the Stack

`top()`
Return (Without Removing) the Object from the Top of the Stack

...anything else?

Backing (i.e., Underlying) Array

when an Array is used as the Underlying Data Structure for an Implementation of an Abstract Data Type, this Array is known as the Backing Array

an Advantage to using a Backing Array is the ability to perform Constant Time Access

a Disadvantage associated with a Backing Array is that the Underlying Structure Cannot Expand or Contract

the introduction of gaps is another issue associated with backing arrays, but this is of no concern for the implementation of the stack … why?

Time Complexity Analysis

the Time Complexity for Each Method is often Expressed with Respect to the Size of the Input and Communicated using Big-Oh Notation

generally speaking, it Represents how the method will Execute in the Worst-Case Scenario

What is the Worst-Case Time Complexity of `pop()`?

What is the Worst-Case Time Complexity of `push()`?

Time Complexity Analysis

What is the **Worst-Case Time Complexity** of `push()`?

What is the **Worst-Case Scenario**?

Backing Array is **Too Small** in **Relation** to the **Stack**
(i.e., the size of the stack, before push, is already at the length of the array)

in this scenario, the **Backing Array** must be **Resized**
(n.b., from the stack size to double the stack size)

Resizing the **Array** entails **Making** a **New Array** and
Copying From the **Old Array to** the **New Array**
this is a **Linear Time Operation** – **"O(n)"**
(n.b., where n is the size of the stack)

Time Complexity Analysis

What is the **Worst-Case Time Complexity** of `pop()`?

What is the **Worst-Case Scenario**?

Backing Array is **Too Large** in **Relation** to the **Stack**
(i.e., the length of the array is at least three times the size of the stack)

in this scenario, the **Backing Array** must be **Resized**
(n.b., from thrice the stack size to double the stack size)

Resizing the **Array** entails **Making** a **New Array** and
Copying From the **Old Array to** the **New Array**
this is a **Linear Time Operation** – **"O(n)"**
(n.b., where n is the size of the stack)

Fundamental and Supporting Operations

the Abstract Data Type known as the "List"
Guarantees the following Fundamental Operations:

`get(i)`
Return (Without Removing) the Object from the i^{th} List Position

`remove(i)`
Remove the Object from the i^{th} List Position

`set(i, o)`
Set the Element at the i^{th} Position of the List to Object o

`add(i, o)`
Insert the Object o into the List at the i^{th} Position

Time Complexity Analysis

What is the **Worst-Case Time Complexity** of:

`get(i)`?

`remove(i)`?

`set(i, o)`?

`add(i, o)`?

What are the **Worst-Case Scenarios**?