

CS121 Data Structures

Stacks

Monika Stepanyan
mstepanyan@aua.am



Spring 2024

Important Data and Statistics



- ▶ 22 classes remaining
- ▶ 25 days till the Midterm exam I
- ▶ 17 days till the Spring Break
- ▶ HW1 due Sunday, February 25, 23:59
- ▶ “Assignment Requirements” file has been updated. Make sure to read it.

Stacks

A **stack** is a collection of objects that are inserted and removed according to the **last-in, first-out (LIFO)** principle

A user can only access or remove the most recently inserted object, at the 'top' of the stack

Stacks are one of the simplest and fundamental data structures.

The Stack Abstract Data Type

Formally, a stack is an ADT that supports the following methods:

`push(e)`: Adds element `e` to the top of the stack

`pop()`: Removes and returns the top element from the stack
(or `null` if the stack is empty)

`top()`: Returns the top element of the stack, without
removing it (or `null` if the stack is empty)

`size()`: Returns the number of elements in the stack

`isEmpty()`: Returns a boolean indicating whether the stack is
empty

By convention, elements added to the stack can have arbitrary
type and a newly created stack is empty

Example

Method	Return Value	Stack Contents
push(5)	—	(5)
push(3)	—	(5, 3)
size()	2	(5, 3)
pop()	3	(5)
isEmpty()	false	(5)
pop()	5	()
isEmpty()	true	()
pop()	null	()
push(7)	—	(7)
push(9)	—	(7, 9)
top()	9	(7, 9)
push(4)	—	(7, 9, 4)
size()	3	(7, 9, 4)
pop()	4	(7, 9)
push(6)	—	(7, 9, 6)
push(8)	—	(7, 9, 6, 8)
pop()	8	(7, 9, 6)

The Stack Application Programming Interface (API)

In Java, we define an **interface** corresponding to our Stack ADT

```
1 public interface Stack<E> {  
2     int size( );  
3     boolean isEmpty( );  
4     void push(E e);  
5     E top( );  
6     E pop( );  
7 }
```

Note that this is different from `java.util.Stack`

Our Stack ADT	Class <code>java.util.Stack</code>
<code>size()</code>	<code>size()</code>
<code>isEmpty()</code>	<code>empty()</code>
<code>push(e)</code>	<code>push(e)</code>
<code>pop()</code>	<code>pop()</code>
<code>top()</code>	<code>peek()</code>

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

The array storing the stack elements may become full

A push operation will then throw a `FullStackException`, which is:

- ▶ Limitation of the array-based implementation
- ▶ Not intrinsic to the Stack ADT



Array-Based Stack Implementation

```
1 public class ArrayStack<E> implements Stack<E> {
2     public static final int CAPACITY=1000; // default array capacity
3     private E[] data; // generic array used for storage
4     private int t = -1; // index of the top element in stack
5     public ArrayStack() { this(CAPACITY); } // constructs stack with default capacity
6     public ArrayStack(int capacity) { // constructs stack with given capacity
7         data = (E[]) new Object[capacity]; // safe cast; compiler may give warning
8     }
9     public int size() { return (t + 1); }
10    public boolean isEmpty() { return (t == -1); }
11    public void push(E e) throws IllegalStateException {
12        if (size() == data.length) throw new IllegalStateException("Stack is full");
13        data[++t] = e; // increment t before storing new item
14    }
15    public E top() {
16        if (isEmpty()) return null;
17        return data[t];
18    }
19    public E pop() {
20        if (isEmpty()) return null;
21        E answer = data[t];
22        data[t] = null; // dereference to help garbage collection
23        t--;
24        return answer;
25    }
26 }
```


Array-Based Stack: Analysis

Drawback: fixed-capacity array, limiting the ultimate stack size

If the application needs much less space than the reserved capacity, memory is wasted

If we try to push an element into a full stack, the implementation throws an exception and refuses to store the new element

Analysis: each method executes a constant number of statements involving arithmetic operations, comparisons, and assignments, or calls to `size` and `isEmpty`, which both run in constant time

Method	Running Time
<code>size</code>	$O(1)$
<code>isEmpty</code>	$O(1)$
<code>top</code>	$O(1)$
<code>push</code>	$O(1)$
<code>pop</code>	$O(1)$

Space usage: $O(N)$, where N is the size of the array, independent from the number $n \leq N$ of elements in the stack

Linked-List-Based Stack

A second simple way of implementing the Stack ADT uses a singly linked list

We add elements (to the top of the stack) at the front of the list

Thus all methods execute in constant time

The linked-list approach has memory usage proportional to the number of actual elements currently in the stack

No arbitrary capacity limits

Using the **adapter** design pattern, we use an existing class and its methods to define a related, but different, class or interface

E.g. we can adapt our `SinglyLinkedList` class to define a new `LinkedStack` class

Linked-List-Based Stack Implementation

```
1 public class LinkedStack<E> implements Stack<E> {  
2     private SinglyLinkedList<E> list = new SinglyLinkedList<>(); // an empty list  
3     public LinkedStack() { } // new stack relies on the initially empty list  
4     public int size() { return list.size(); }  
5     public boolean isEmpty() { return list.isEmpty(); }  
6     public void push(E element) { list.addFirst(element); }  
7     public E top() { return list.first(); }  
8     public E pop() { return list.removeFirst(); }  
9 }
```

Reversing an Array

How would you reverse an array using a stack?

Reversing an Array

How would you reverse an array using a stack?

```
1  /** A generic method for reversing an array. */
2  public static <E> void reverse(E[ ] a) {
3      Stack<E> buffer = new ArrayStack<>(a.length);
4      for (int i=0; i < a.length; i++)
5          buffer.push(a[i]);
6      for (int i=0; i < a.length; i++)
7          a[i] = buffer.pop();
8  }
```

Matching Parentheses

In arithmetic expressions that may contain various pairs of grouping symbols, like

- ▶ Parentheses: '(' and ')'
- ▶ Braces: '{' and '}'
- ▶ Brackets: '[' and ']'

each opening symbol must match its corresponding closing symbol.

Example: $[(5 + x) - (y + z)]$

Problem: check if the grouping symbols of an arithmetic expression match up correctly

- ▶ $() (()) \{ ([()]) \}$

Matching Parentheses

In arithmetic expressions that may contain various pairs of grouping symbols, like

- ▶ Parentheses: '(' and ')'
- ▶ Braces: '{' and '}'
- ▶ Brackets: '[' and ']'

each opening symbol must match its corresponding closing symbol.

Example: $[(5 + x) - (y + z)]$

Problem: check if the grouping symbols of an arithmetic expression match up correctly

- ▶ $()(())\{([()])\}$ Correct
- ▶ $((()(())\{([()])\}))$

Matching Parentheses

In arithmetic expressions that may contain various pairs of grouping symbols, like

- ▶ Parentheses: '(' and ')'
- ▶ Braces: '{' and '}'
- ▶ Brackets: '[' and ']'

each opening symbol must match its corresponding closing symbol.

Example: $[(5 + x) - (y + z)]$

Problem: check if the grouping symbols of an arithmetic expression match up correctly

- ▶ $()(())\{([()])\}$ Correct
- ▶ $((()(())\{([()])\})\})$ Correct
- ▶ $)(())\{([()])\}$

Matching Parentheses

In arithmetic expressions that may contain various pairs of grouping symbols, like

- ▶ Parentheses: '(' and ')'
- ▶ Braces: '{' and '}'
- ▶ Brackets: '[' and ']'

each opening symbol must match its corresponding closing symbol.

Example: $[(5 + x) - (y + z)]$

Problem: check if the grouping symbols of an arithmetic expression match up correctly

- ▶ $()(())\{([())\}$ Correct
- ▶ $((())(())\{([())\})$ Correct
- ▶ $)(())\{([())\}$ Incorrect
- ▶ $(\{ [] \})$

Matching Parentheses

In arithmetic expressions that may contain various pairs of grouping symbols, like

- ▶ Parentheses: '(' and ')'
- ▶ Braces: '{' and '}'
- ▶ Brackets: '[' and ']'

each opening symbol must match its corresponding closing symbol.

Example: $[(5 + x) - (y + z)]$

Problem: check if the grouping symbols of an arithmetic expression match up correctly

- ▶ $()(())\{([())\}$ Correct
- ▶ $((())(())\{([())\})$ Correct
- ▶ $)(())\{([())\}$ Incorrect
- ▶ $(\{ [] \})$ Incorrect
- ▶ (

Matching Parentheses

In arithmetic expressions that may contain various pairs of grouping symbols, like

- ▶ Parentheses: '(' and ')'
- ▶ Braces: '{' and '}'
- ▶ Brackets: '[' and ']'

each opening symbol must match its corresponding closing symbol.

Example: $[(5 + x) - (y + z)]$

Problem: check if the grouping symbols of an arithmetic expression match up correctly

- ▶ $()(())\{([())\}$ Correct
- ▶ $((()(())\{([())\})\})$ Correct
- ▶ $)(())\{([())\}$ Incorrect
- ▶ $(\{[]\})$ Incorrect
- ▶ $($ Incorrect

Matching Parentheses (cont'd)

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

if $X[i]$ is an opening grouping symbol **then**

$S.\text{push}(X[i])$

else if $X[i]$ is a closing grouping symbol **then**

if $S.\text{empty}()$ **then**

return false

▷nothing to match with

if $S.\text{top}()$ does not match the type of $X[i]$ **then**

return false

▷wrong type

$S.\text{pop}()$

if $S.\text{empty}()$ **then**

return true

▷every symbol matched

else

return false

▷some symbols were never matched

Matching Parentheses: Implementation

```
1  /** Tests if delimiters in the given expression are properly matched. */
2  public static boolean isMatched(String expression) {
3      final String opening  = "{[(";           // opening delimiters
4      final String closing  = "}]";           // respective closing delimiters
5      Stack<Character> buffer = new LinkedStack<>();
6      for (char c : expression.toCharArray()) {
7          if (opening.indexOf(c) != -1)        // this is a left delimiter
8              buffer.push(c);
9          else if (closing.indexOf(c) != -1) {  // this is a right delimiter
10             if (buffer.isEmpty())             // nothing to match with
11                 return false;
12             if (closing.indexOf(c) != opening.indexOf(buffer.pop()))
13                 return false;                // mismatched delimiter
14         }
15     }
16     return buffer.isEmpty();                 // were all opening delimiters matched?
17 }
```

Summary

Reading

Section 6.1 Stacks

Questions?