CS121 Data Structures A, C
Stacks

Varduhi Yeghiazaryan
vyeghiazaryan@aua.am

Fall 2021

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 a simplest and fundamental data structure. They are among the most important, as they are used in many applications

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

```java
public interface Stack<E> {
  int size( );
  boolean isEmpty( );
  void push(E e);
  E top( );
  E pop( );
}
```

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

Our Stack ADT	**Class** `java.util.Stack`
size()	size()
isEmpty()	empty()
push(e)	push(e)
pop()	pop()
top()	peek()

In C++, the Standard Template Library (STL) provides an implementation of a stack

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

```java
public class ArrayStack<E> implements Stack<E> {
  public static final int CAPACITY=1000;  // default array capacity
  private E[ ] data;                      // generic array used for storage
  private int t = -1;                     // index of the top element in stack
  public ArrayStack( ) { this(CAPACITY); } // constructs stack with default capacity
  public ArrayStack(int capacity) {       // constructs stack with given capacity
    data = (E[ ]) new Object[capacity];   // safe cast; compiler may give warning
  }
  public int size( ) { return (t + 1); }
  public boolean isEmpty( ) { return (t == -1); }
  public void push(E e) throws IllegalStateException {
    if (size( ) == data.length) throw new IllegalStateException("Stack is full");
    data[++t] = e;                        // increment t before storing new item
  }
  public E top( ) {
    if (isEmpty( )) return null;
    return data[t];
  }
  public E pop( ) {
    if (isEmpty( )) return null;
    E answer = data[t];
    data[t] = null;                       // dereference to help garbage collection
    t--;
    return answer;
  }
}
```

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**
size	$O(1)$
isEmpty	$O(1)$
top	$O(1)$
push	$O(1)$
pop	$O(1)$

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 modify an existing class so that its methods match those of 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

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

Reversing an Array

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

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 ← 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.empty() then
            return false                    ▷nothing to match with
        if S.top() does not match the type of X[i] then
            return false                    ▷wrong type
        S.pop()
if S.empty() then
    return true                             ▷every symbol matched
else
    return false                            ▷some symbols were never matched
```

Matching Parentheses: Implementation

```java
/** Tests if delimiters in the given expression are properly matched. */
public static boolean isMatched(String expression) {
  final String opening = "({[";          // opening delimiters
  final String closing = ")}]";          // respective closing delimiters
  Stack<Character> buffer = new LinkedStack<>();
  for (char c : expression.toCharArray()) {
    if (opening.indexOf(c) != −1)        // this is a left delimiter
      buffer.push(c);
    else if (closing.indexOf(c) != −1) { // this is a right delimiter
      if (buffer.isEmpty())              // nothing to match with
        return false;
      if (closing.indexOf(c) != opening.indexOf(buffer.pop()))
        return false;                    // mismatched delimiter
    }
  }
  return buffer.isEmpty();               // were all opening delimiters matched?
}
```

Matching Tags in a Markup Language

Application of matching delimiters: validation of markup languages such as HTML or XML

```
<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>
```

(a)

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?

(b)

HTML Document

In an HTML document, portions of text are delimited by **HTML tags**

Opening HTML tag: `<name>`

Closing HTML tag: `</name>`

Commonly used tags:

1. `<body>` document body
2. `<h1>` section header
3. `<center>` center justify
4. `<p>` paragraph
5. `<ol>` numbered (ordered) list
6. `<li>` list item

How can we check if an HTML document has matching tags?

Summary

Reading

Section 6.1 Stacks

Questions?