Advanced Programming

Stacks

Sources: Data Structures and Abstractions with Java, by Carrano
chapter 5, 3rd edition or
chapter 20, 2nd edition
Chapter Contents

• Specifications of the ADT Stack
• Using a Stack to Process Algebraic Expressions
  ▪ Checking for Balanced Parentheses, Brackets, and Braces in an Infix Algebraic Expression
  ▪ Transforming an Infix Expression to a Postfix Expression
  ▪ Evaluating Postfix Expressions
  ▪ Evaluating Infix Expressions
• The Program Stack
  ▪ Recursive Methods
• Using a Stack Instead of Recursion
  ▪ An Iterative Binary Search
• Java Class Library: The Class Stack
Specifications of the ADT Stack

• Organizes entries according to order in which added
• Additions are made to one end, the top
• The item most recently added is always on the top

Fig. 20-1 Some familiar stacks.
Specifications of the ADT Stack

- Specification of a stack of objects

```java
public interface StackInterface {
    /** Task: Adds a new entry to the top of the stack.
     * @param newEntry an object to be added to the stack */
    public void push(Object newEntry);

    /** Task: Removes and returns the top of the stack.
     * @return either the object at the top of the stack or null if the stack was empty */
    public Object pop();

    /** Task: Retrieves the top of the stack.
     * @return either the object at the top of the stack or null if the stack is empty */
    public Object peek();

    /** Task: Determines whether the stack is empty.
     * @return true if the stack is empty */
    public boolean isEmpty();

    /** Task: Removes all entries from the stack */
    public void clear();
}
```
Specifications of the ADT Stack

Fig. 20-1 A stack of strings after (a) push adds Jim; (b) push adds Jess; (c) push adds Jill; (d) push adds Jane; (e) push adds Joe; (f) pop retrieves and removes Joe; (g) pop retrieves and removes Jane
Using a Stack to Process Algebraic Expressions

- **Infix expressions**
  - Binary operators appear *between* operands
  - $a + b$

- **Prefix expressions**
  - Binary operators appear *before* operands
  - $+ a b$

- **Postfix expressions**
  - Binary operators appear *after* operands
  - $a b +$
  - Easier to process – no need for parentheses nor precedence
Checking for Balanced \((\), \[\], \{\})

Fig. 20-3 The contents of a stack during the scan of an expression that contains the balanced delimiters \{ [ () ] \}
Checking for Balanced \( (), [], {} \)

Fig. 20-4 The contents of a stack during the scan of an expression that contains the unbalanced delimiters \{ [ ( ] ) \}
Checking for Balanced ( ), [ ], { }

Fig. 20-5 The contents of a stack during the scan of an expression that contains the unbalanced delimiters \([ () ]\)
Checking for Balanced \((\), \([\)], \{\})

Fig. 20-6 The contents of a stack during the scan of an expression that contains the unbalanced delimiters \(\{ [ ( ) ] \)}
Checking for Balanced () , [] , {} 

Algorithm checkBalance(expression)
// Returns true if the parentheses, brackets, and braces in an expression are paired correctly.

isBalanced = true

while ( (isBalanced == true) and not at end of expression) 
{ nextCharacter = next character in expression 
  switch (nextCharacter) 
  { case '(': case '[': case '{': 
      Push nextCharacter onto stack 
    break 
    case '(': case ']': case '}': 
      if (stack is empty) isBalanced = false 
    else 
    { openDelimiter = top of stack 
      Pop stack 
      isBalanced = true or false according to whether openDelimiter and nextCharacter are a pair of delimiters 
    } 
    break 
  } 
} 

if (stack is not empty) isBalanced = false

return isBalanced
Transforming Infix to Postfix

Fig. 20-7 Converting the infix expression $a + b * c$ to postfix form
Transforming Infix to Postfix

Fig. 20-8(a) Converting infix expression to postfix form: \( a - b + c \)
Transforming Infix to Postfix

<table>
<thead>
<tr>
<th>Next Character</th>
<th>Postfix</th>
<th>Operator Stack (bottom to top)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a ^</td>
<td>a</td>
<td>^</td>
</tr>
<tr>
<td>b ^</td>
<td>a b</td>
<td>^</td>
</tr>
<tr>
<td>c</td>
<td>a b c ^</td>
<td>^ ^</td>
</tr>
<tr>
<td></td>
<td>a b c ^ ^</td>
<td>^ ^</td>
</tr>
</tbody>
</table>

Fig. 20-8(b) Converting infix expression to postfix form: \( a ^ b ^ c \)
## Infix-to-Postfix Algorithm

<table>
<thead>
<tr>
<th>Symbol in Infix</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operand</td>
<td>Append to end of output expression</td>
</tr>
<tr>
<td>Operator ^</td>
<td>Push ^ onto stack</td>
</tr>
<tr>
<td>Operator +, -, *, or /</td>
<td>Pop operators from stack, append to output expression until stack empty or top has lower precedence than new operator. Then push new operator onto stack</td>
</tr>
<tr>
<td>Open parenthesis</td>
<td>Push ( onto stack</td>
</tr>
<tr>
<td>Close parenthesis</td>
<td>Pop operators from stack, append to output expression until we pop an open parenthesis. Discard both parentheses.</td>
</tr>
</tbody>
</table>
Transforming Infix to Postfix

Fig. 20-9 Steps to convert the infix expression $\frac{a}{b} * (c + (d - e))$ to postfix form.
Evaluating Postfix Expression

Fig. 20-10 The stack during the evaluation of the postfix expression $a \ b \ /$ when $a$ is 2 and $b$ is 4
Transforming Infix to Postfix

Fig. 20-11 The stack during the evaluation of the postfix expression $a \ b + c /$ when $a$ is 2, $b$ is 4 and $c$ is 3