Stacks

- Stack Applications
- Evaluating Postfix Expressions
- Introduction to Project 2
- Reading: L&C Section 3.2, 3.4-3.8
A Conceptual View of a Stack

Adding an Element

Top of Stack

Removing an Element
Applications for a Stack

• A stack can be used as an underlying mechanism for many common applications
  – Evaluate postfix and prefix expressions
  – Reverse the order of a list of elements
  – Support an “undo” operation in an application
  – Backtrack in solving a maze
Evaluating Infix Expressions

• Traditional arithmetic expressions are written in *infix* notation (aka algebraic notation)
  (operand) (operator) (operand) (operator) (operand)
  \[ 4 + 5 \times 2 \]

• When evaluating an infix expression, we need to use the precedence of operators
  – The above expression evaluates to \( 4 + (5 \times 2) = 14 \)
  – NOT in left to right order as written \( (4 + 5) \times 2 = 18 \)

• We use parentheses to override precedence
Evaluating Postfix Expressions

• *Postfix* notation is an alternative method to represent the same expression
  (operand) (operand) (operand) (operator) (operator)
  4 5 2 * +

• When evaluating a postfix expression, we do not need to know the precedence of operators

• Note: We do need to know the precedence of operators to convert an infix expression to its corresponding postfix expression
Evaluating Postfix Expressions

• We can process from left to right as long as we use the proper evaluation algorithm
• Postfix evaluation algorithm calls for us to:
  – Push each operand onto the stack
  – Execute each operator on the top element(s) of the stack (An operator may be unary or binary and execution may pop one or two values off the stack)
  – Push result of each operation onto the stack
Evaluating Postfix Expressions

- Expression = 7 4 -3 * 1 5 + / *
Evaluating Postfix Expressions

• Core of evaluation algorithm using a stack
  
  ```java
  while (tokenizer.hasMoreTokens()) {
      token = tokenizer.nextToken(); // returns String
      if (isOperator(token)) {
          int op2 = (stack.pop()).intValue(); // Integer
          int op1 = (stack.pop()).intValue(); // to int
          int res = evalSingleOp(token.charAt(0), op1, op2);
          stack.push(new Integer(res));
      }
      else { // String to int to Integer conversion here
          stack.push (new Integer(Integer.parseInt(token)));
      }
  } // Note: Textbook’s code does not take advantage of
  // Java 5.0 auto-boxing and auto-unboxing
  ```
Evaluating Postfix Expressions

• Instead of this:
  
  ```java
  int op2 = (stack.pop()).intValue(); // Integer to int
  int op1 = (stack.pop()).intValue(); // Integer to int
  int res = evalSingleOp(token.charAt(0), op1, op2);
  ```

• Why not this:
  
  ```java
  int res = evalSingleOp(token.charAt(0),
                         (stack.pop()).intValue(),
                         (stack.pop()).intValue());
  ```

• In which order are the parameters evaluated?

• Affects order of the operands to evaluation
Evaluating Postfix Expressions

• The parameters to the `evalSingleOp` method are evaluated in left to right order.

• The pops of the operands from the stack occur in the opposite order from the order assumed in the interface to the method.

• Results:

<table>
<thead>
<tr>
<th>Original</th>
<th>Alternative</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 3 / = 2</td>
<td>6 3 / = 0</td>
</tr>
<tr>
<td>3 6 / = 0</td>
<td>3 6 / = 2</td>
</tr>
</tbody>
</table>
Evaluating Postfix Expressions

• Our consideration of the alternative code above demonstrates a very good point
• Be sure that your code keeps track of the state of the data stored on the stack
• Your code must be written consistent with the order data will be retrieved from the stack to use the retrieved data correctly
The java.util.Stack Class

• The java.util.Stack class is part of the Java collections API.
• It is a subclass of the java.util.Vector class which is a subclass of java.util.AbstractList.
• java.util.Stack inherits some methods from its superclasses that are inappropriate for a stack collection – a bad design decision!
• A good programmer will avoid using those methods on a Stack object.
The java.util.Stack Class

• java.util.Stack has a search method that returns the distance from the top of the stack to the first occurrence of that object on the stack, e.g. search returns 1 for the top element of the stack
• If the object is not found, search returns -1
• The search method is O(n)
• Does this method violate the principles of a stack as a data structure? (Discussion)
Introduction to Project 2

• Mazes are a traditional form of puzzle
• The maze in this project is a bit unique

Lost in Migration

The maze is one of the most ancient and primordial of all puzzles. From the ancient Greek labyrinth in the Odyssey to the title lettering of this year's movie *Inception*, getting lost in a maze symbolizes the harrowing complexity of life.

Modern maze makers like Robert Abbott and Adrian Fisher have invented new ways to get us pleasurably lost. Here is a maze in which you move by jumping.

Your challenge is to get from the green square to the red square by jumping from square to square. Every square has a bird. You may jump any distance in the direction that the bird is pointing. For instance, from the green Start square you may jump one or two squares diagonally up and to the right. There is more than one solution path.

This puzzle was inspired by Lost in Migration, a brain game on lumosity.com. —Scott Kim
Introduction to Project 2

• You are provided partial code for the four classes shown in the UML class diagram.

• The code you need to write:
  – Add code to Maze constructor to build a queue of Bird objects and save it as the bird queue for each Bird in the maze.
  – Complete the Bird class getNextBird method.
  – Complete Solve1 main method - iterative.
  – Complete Solve2 main method - recursive.