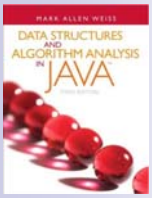
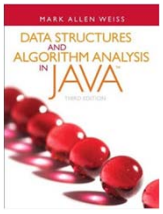


FIU FLORIDA INTERNATIONAL UNIVERSITY

Lists, Stacks, and Queues (III)

Dr. Antonio L. Bajuelos

FIU School of Computing & Information Sciences



COP-3530 - Data Structures

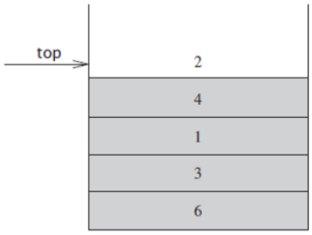
Module #2: Lists, Stacks, and Queues (part III)

Outline:

- The Stack model (LIFO list).
- Array implementation of a Stack.
- Linked-List implementation of a Stack.
- Complexity analysis:
  - Array vs Linked-List implementation.
- Applications of Stacks

Stack Model

- **Definition:** a **stack** is a **list** with the restriction that **insertions** and **deletions** can be performed in only one position, namely, the end of the list, called the **top**.
- **Stacks** are sometimes known as
  - **LIFO** - **(L)ast (I)n, (F)irst (O)ut** lists.
- **Operations:**
  - **push** – insert (on the top)
  - **pop** - delete the most recently inserted element



3

Implementation of Stacks

- **ArrayList** and **LinkedList** support stack operations.
- **Linked List** implementation of **Stacks**
  - Is used a **singly linked list**.
    - We perform a **push** by inserting at the front of the list.
    - We perform a **pop** by deleting the element at the front of the list.
    - A **top** operation merely examines the element at the front of the list, returning its value.

4

Arraylist Implementation of Stacks

- For the **Arraylist** implementation of the **stacks** we need:
  - **theArray**
  - **topOfStack**, which is −1 for an empty stack (this is how an empty stack is initialized).
- To **push** some element x onto the stack, we increment topOfStack and then set theArray[topOfStack] = x.
- To **pop**, we set the return value to theArray[topOfStack] and then decrement topOfStack.
- Notice that **pop** and **push** operations are performed in not only constant time, but very fast constant time.

5

Applications of the Stacks

- ***Balancing Symbols***
  - Compilers check programs for syntax errors, but frequently a lack of one symbol (such as a missing brace or comment starter) will cause the compiler to spill out a hundred lines of diagnostics without identifying the real error.
- A useful tool in this situation is a program that checks whether everything is balanced.
- **Example:** Every right brace, {, bracket, [, and parenthesis, (, must correspond to its left counterpart.
  - The sequence **[()]** is legal, but **[(])** is wrong.

6

Applications of the Stacks

- ***Balancing Symbols*** (cont…)
  - **Example:**
    - For simplicity, we will (only) just check for balancing of parentheses, (), brackets, [], and braces, {}, and ignore any other character that appears.
  - **Algorithm:**
    - Make an empty stack. Read characters until end of file.
    - If the character is an opening symbol: ({ [
      - push it onto the stack.
    - If it is a closing symbol:) }]
      - if the stack is empty report and error
      - Otherwise, pop the stack.
      - If the symbol popped is not the corresponding opening symbol, then report an error.
    - At end of file, if the stack is not empty report an error.

7

Applications of the Stacks

- ***Postfix Expressions***
  - Suppose we have a pocket calculator and would like to compute the cost of a shopping trip.
  - The list of items are 4.99, 5.99, and 6.99 and suppose that the first and the last items are taxable and the second is not.
  - The sequence 4.99 * 1.06 + 5.99 + 6.99 * 1.06 would give
    - the correct answer (18.69) on a **scientific calculator,**
    - the wrong answer (19.37) on a **simple calculator**.

8

Applications of the Stacks

- ***Postfix Expressions*** (cont…)

(((4.99 * 1.06) + 5.99) + (6.99 * 1.06)) = 19.37

  - A typical evaluation sequence for this example might be to
    - multiply 4.99 and 1.06 saving this answer as A1,
    - add 5.99 and A1, saving the result in A1,
    - multiply 6.99 and 1.06, saving the answer in A2,
    - add A1 and A2, leaving the final answer in A1.
  - We can write this sequence of operations as follows:

4.99 1.06 * 5.99 + 6.99 1.06 * +

  - This notation is known as **Postfix** or **Reverse Polish Notation (RPN)** and is evaluated exactly as we have described above.

9

Applications of the Stacks

- ***Postfix Expressions*** (cont…)

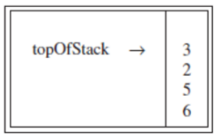
4.99 1.06 * 5.99 + 6.99 1.06 * +

  - How to **evaluate** this sequence?
    - The easiest way to do this is to use a **stack!!!**
    - **Algorithm:**
      - When a **number** is seen, it is pushed onto the stack;
      - when an **operator** is seen, the operator is applied to the two numbers (symbols) that are popped from the stack, and the result is pushed onto the stack.

10

Applications of the Stacks

- **Example: 6 5 2 3 + 8 * + 3 + ***
  - when a number is seen, it is pushed onto the stack;



  - when an operator is seen, the operator is applied to the two numbers (symbols) that are popped from the stack, and the result is pushed onto the stack;
    - Next a '+' is read, so 3 and 2 are popped from the stack and their sum, 5, is pushed.

11

Applications of the Stacks

- **Example: 6 5 2 3 + 8 * + 3 + ***

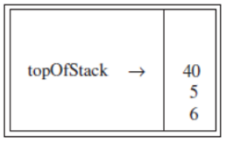
  - Next 8 is pushed

  - Now a '*' is seen, so 8 and 5 are popped and 5 * 8 = 40 is pushed.

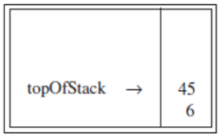
12

Applications of the Stacks

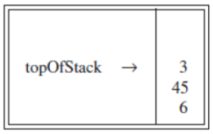
- **Example: 6 5 2 3 + 8 * + 3 + ***



- Next a '+' is seen, so 40 and 5 are popped and 5 + 40 = 45 is pushed.



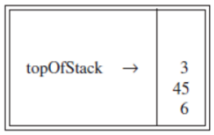
- Now, 3 is pushed.



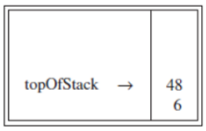
13

Applications of the Stacks

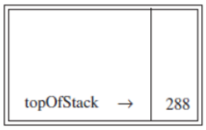
- **Example: 6 5 2 3 + 8 * + 3 + ***



- Next '+' pops 3 and 45 and pushes 45 + 3 = 48.



- Finally, a '*' is seen and 48 and 6 are popped; the result, 6 * 48 = 288, is pushed.



14

Applications of the Stacks

- ***Postfix Expressions*** (cont…)
  - **Important points:**
    - The time to evaluate a **postfix expression** is *O*(*N*) - processing each element in the input consists of stack operations and thus takes constant time.
    - If an expression is given in **postfix notation**, then no need to know any precedence rules; this is an obvious advantage.

15

16