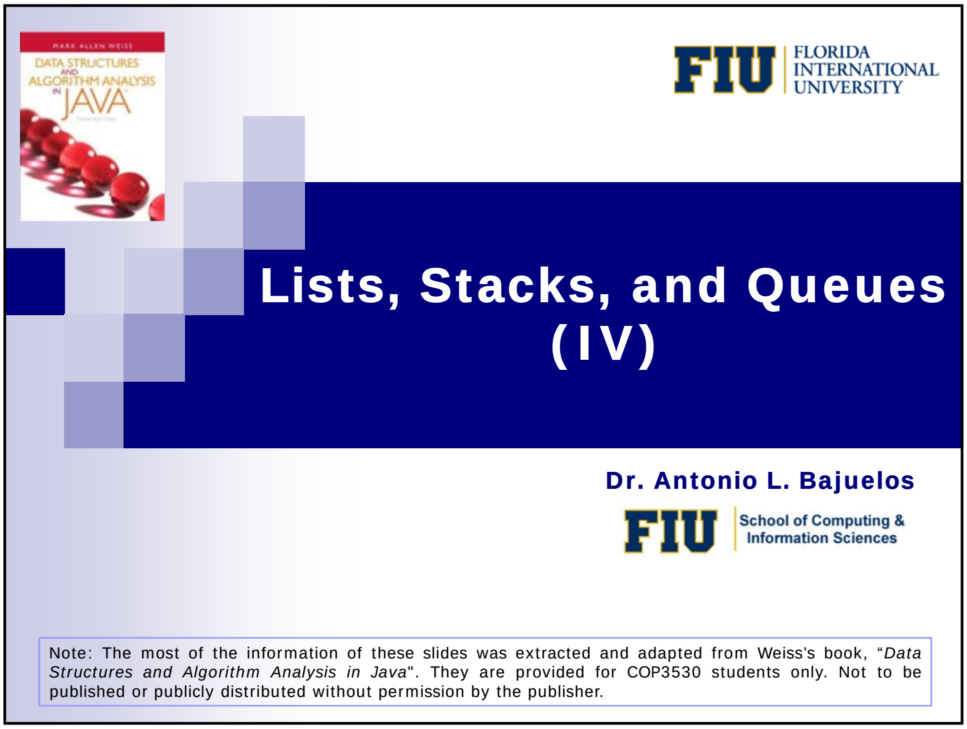


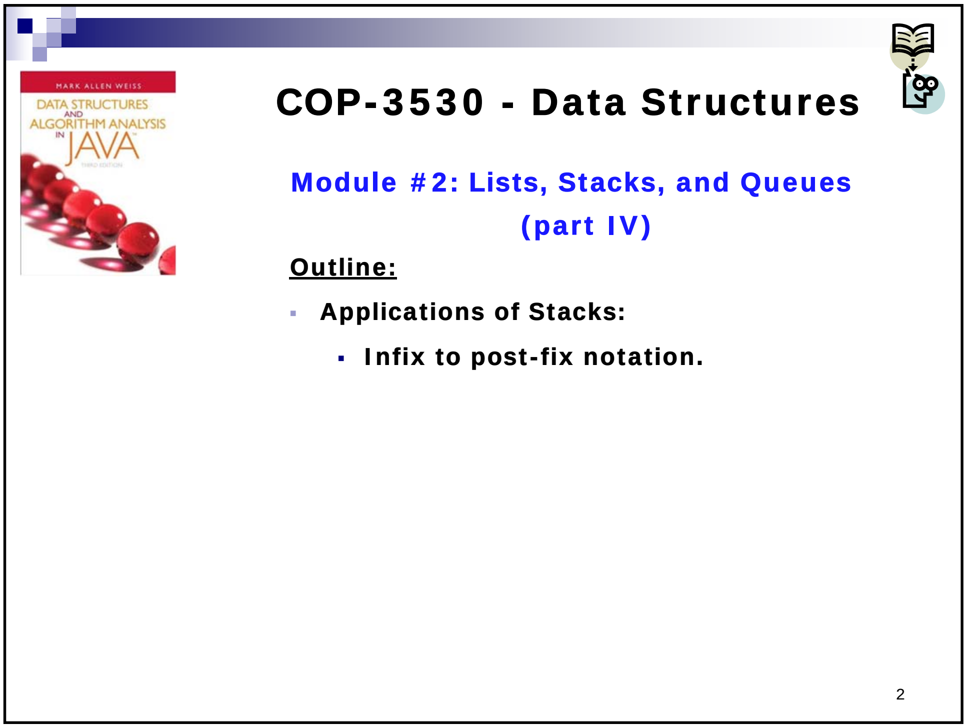
FIU | FLORIDA INTERNATIONAL UNIVERSITY

Lists, Stacks, and Queues (IV)

Dr. Antonio L. Bajuelos

FIU | School of Computing & Information Sciences

Note: The most of the information of these slides was extracted and adapted from Weiss's book, "*Data Structures and Algorithm Analysis in Java*". They are provided for COP3530 students only. Not to be published or publicly distributed without permission by the publisher.



COP-3530 - Data Structures

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

Outline:

- **Applications of Stacks:**
  - **Infix to post-fix notation.**

Applications of the Stacks

- ***Infix to Postfix Conversion***
  - □ Not only can a **stack** be used to evaluate a postfix expression!
  - □ The **stack** can also be used to convert an expression in standard form (**infix**) into **postfix**.
  - □ Suppose that we have only the **operators** **+**, *****, **(**, and **)**, and insisting on the usual precedence rules.
  - □ Suppose we want to convert the **infix expression**

    a + b * c + (d * e + f) * g

    into **postfix**.
  - □ A correct answer is

    a b c * + d e * f + g * +

Applications of the Stacks

- ***Infix to Postfix Conversion***

a + b * c + (d * e + f) * g **into** a b c * + d e * f + g * +

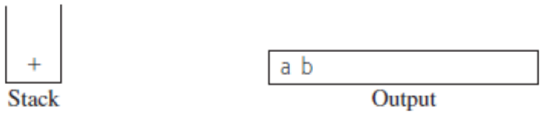
Algorithm:

- □ When an **operand** is read, it is immediately placed onto the output.
- □ **Operators** and (are not immediately output, so they must be saved onto the **stack**.
- □ If we see a), then we pop the stack, writing symbols until we encounter a (corresponding) (, which is popped but not output.
- □ If we see any other symbol +, *, (, then we pop entries from the stack until we find an entry of lower priority. One exception is that we never remove a (from the stack except when processing a). For the purposes of this operation, + has lowest priority and (highest.
- □ When the popping is done, we push the operator onto the stack.
- □ Finally, if we read the end of input, we pop the stack until it is empty, writing symbols onto the output.

Applications of the Stacks

- ***Infix to Postfix Conversion***
  - □ **Example:**

a + b * c + (d * e + f) * g **into** a b c * + d e * f + g * +



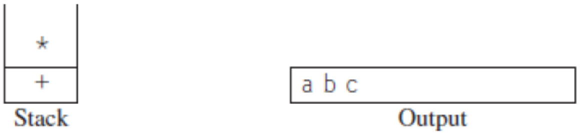
Next a * is read. The top entry on the operator stack has lower precedence than *, so nothing is output and * is put on the stack.

a + b * c + (d * e + f) * g

Applications of the Stacks

- ***Infix to Postfix Conversion***
  - □ **Example:**

a + b * c + (d * e + f) * g

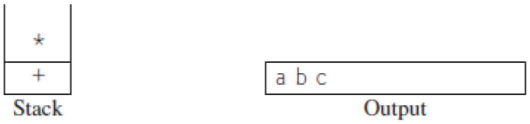


a + b * c + (d * e + f) * g

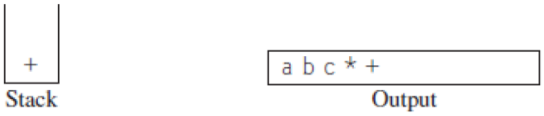
Applications of the Stacks

- ***Infix to Postfix Conversion***
  - □ **Example:**

a + b * c + (d * e + f) * g



The next symbol is a +. Checking the stack, we find that we will pop a * and place it on the output; pop the other +, which is not of *lower* but equal priority, on the stack; and then push the +.

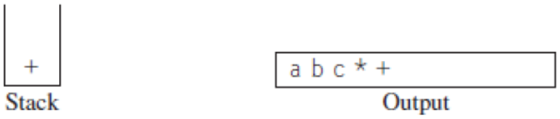


a + b * c + (d * e + f) * g

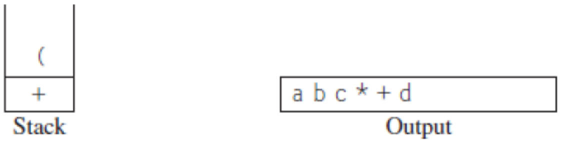
Applications of the Stacks

- ***Infix to Postfix Conversion***

a + b * c + (d * e + f) * g

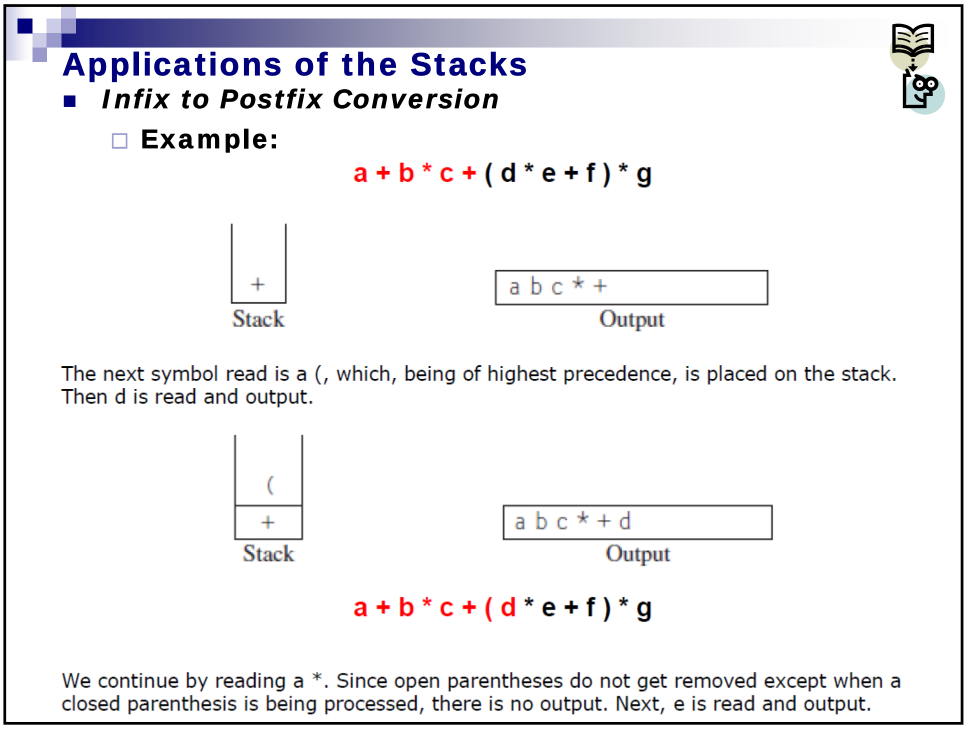


The next symbol read is a (, which, being of highest precedence, is placed on the stack. Then d is read and output.



a + b * c + (d * e + f) * g

We continue by reading a *. Since open parentheses do not get removed except when a closed parenthesis is being processed, there is no output. Next, e is read and output.



Applications of the Stacks

- ***Infix to Postfix Conversion***
  - □ **Example:**

a + b * c + (d * e + f) * g

+

Stack

a b c * +

Output

The next symbol read is a (, which, being of highest precedence, is placed on the stack. Then d is read and output.

(

+

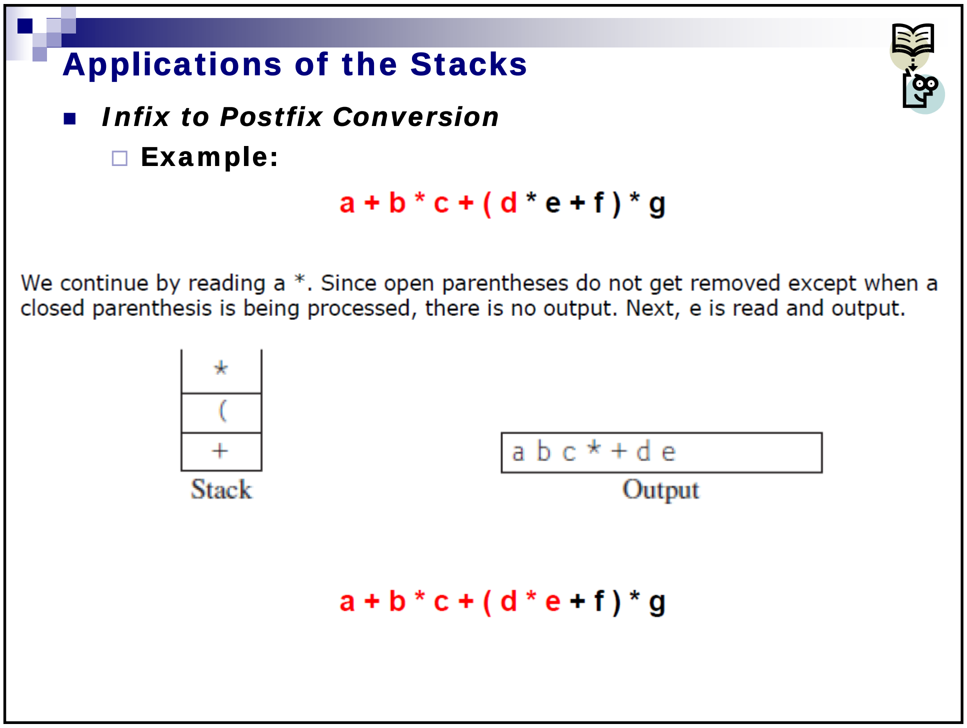
Stack

a b c * + d

Output

a + b * c + (d * e + f) * g

We continue by reading a *. Since open parentheses do not get removed except when a closed parenthesis is being processed, there is no output. Next, e is read and output.



Applications of the Stacks

- ***Infix to Postfix Conversion***
  - □ **Example:**

a + b * c + (d * e + f) * g

We continue by reading a *. Since open parentheses do not get removed except when a closed parenthesis is being processed, there is no output. Next, e is read and output.

*

(

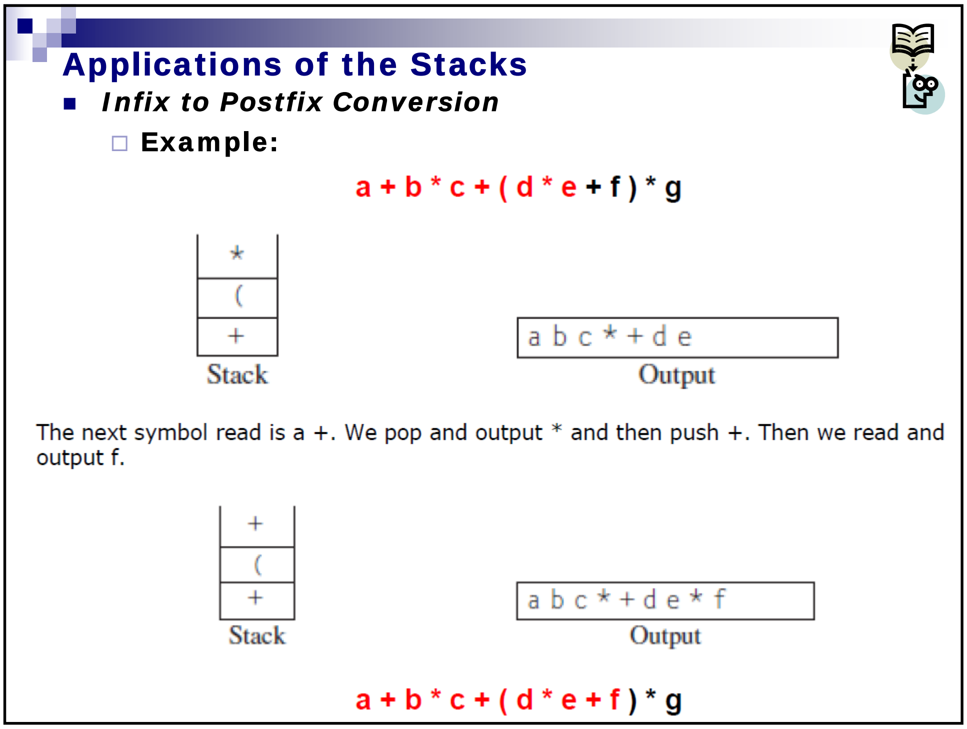
+

Stack

a b c * + d e

Output

a + b * c + (d * e + f) * g



Applications of the Stacks

- ***Infix to Postfix Conversion***
  - □ **Example:**

a + b * c + (d * e + f) * g

Stack
*
(
+

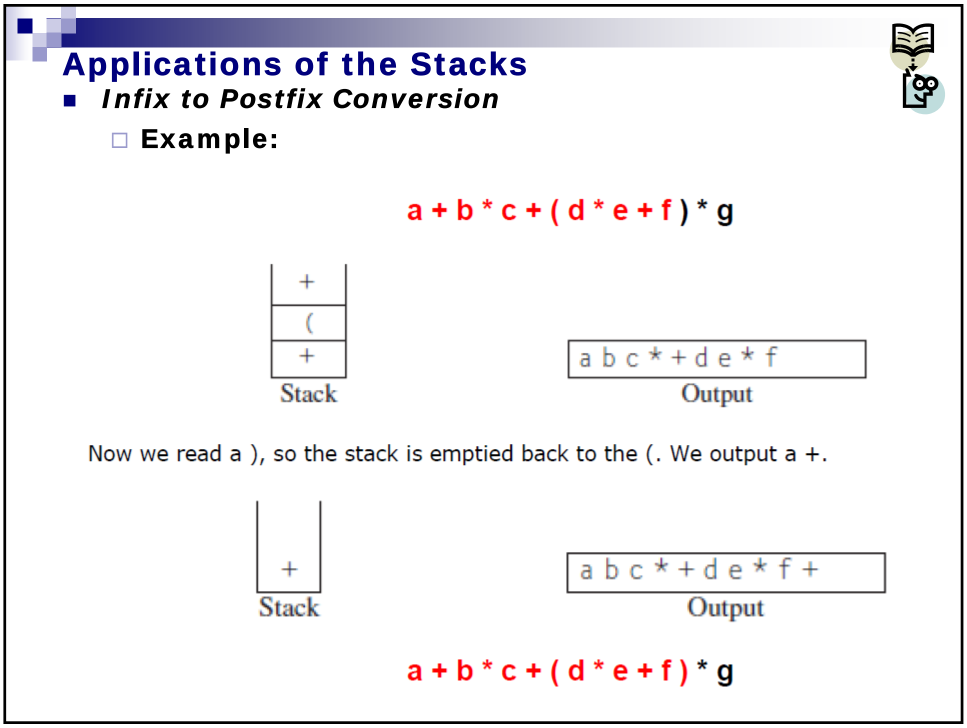
Output: a b c * + d e

The next symbol read is a +. We pop and output * and then push +. Then we read and output f.

Stack
+
(
+

Output: a b c * + d e * f

a + b * c + (d * e + f) * g



Applications of the Stacks

- ***Infix to Postfix Conversion***
  - □ **Example:**

a + b * c + (d * e + f) * g

Stack
+
(
+

Output: a b c * + d e * f

Now we read a), so the stack is emptied back to the (. We output a +.

Stack
+

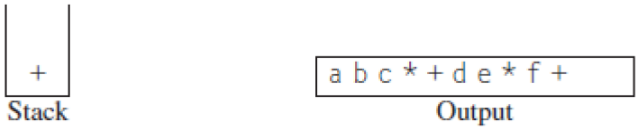
Output: a b c * + d e * f +

a + b * c + (d * e + f) * g

Applications of the Stacks

- *Infix to Postfix Conversion*
  - □ **Example:**

a + b * c + (d * e + f) * g



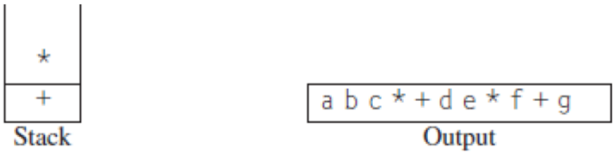
+

Stack

a b c * + d e * f +

Output

We read a * next; it is pushed onto the stack. Then g is read and output.



*

+

Stack

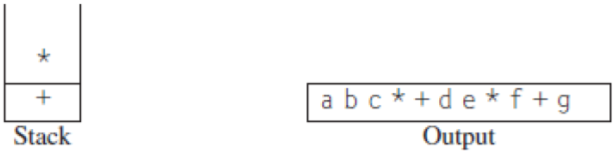
a b c * + d e * f + g

Output

a + b * c + (d * e + f) * g

Applications of the Stacks

- *Infix to Postfix Conversion*
  - □ **Example:**



*

+

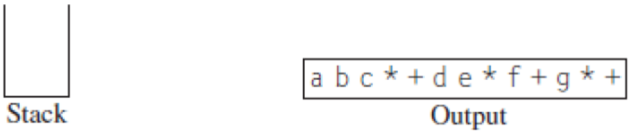
Stack

a b c * + d e * f + g

Output

a + b * c + (d * e + f) * g

The input is now empty, so we pop and output symbols from the stack until it is empty



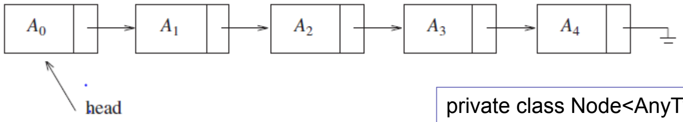
Stack

a b c * + d e * f + g * +

Output

Exercise

- Efficiently implement a **stack** class using a **singly linked list**, with no header or tail nodes.



```
public class SingleStack<AnyType>
{
  SingleStack()  {
    head = null;
  }
  void push(AnyType x)  {
    Node<AnyType> p = new Node<AnyType>(x,head);
    head = p;
  }
  AnyType top()   {
    return head.data;
  }

  void pop()  {
    head = head.next;
  }
```

```
private class Node<AnyType>
  {
    Node()
    { this(null, null); }
    Node(AnyType x)
    { this(x, null); }
    Node(AnyType x, Node p)
    {
```

a + b * c + (d * e + f) * g

Next a * is read. The top entry on the operator stack has lower precedence than *, nothing is output and * is put on the stack.

a + b * c + (d * e + f) * g