Stacks & Their Applications (Postfix/Infix)

Computer Science Department
University of Central Florida

COP 3502 – Computer Science I

Outline

- Stacks
  - What are they and how they work
- Stack Applications
  - Infix to Postfix conversion
  - Evaluation of Postfix expressions

Stacks – An Overview

- Abstract Data Type (ADT):
    - What is an Abstract Data Type?

    - To answer this, let us ask the negative of this:
        - What is NOT an Abstract Data Type (in C)?
            - int
                - int is a built-in type in the C language
            - double
                - double is a built-in type in the C language
            - There are certainly many others we can list
        - These data types are already built into the language.

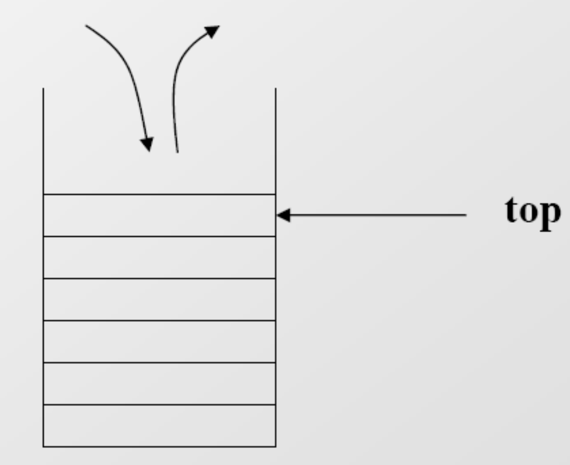
Stacks – An Overview

- Abstract Data Type (ADT):
  - So again, what is an Abstract Data Type?
    - It is a data type that is NOT built into the language
    - It is a data type that we will “build”
      - We will specify what it is, how it is used, etc.
    - It is often **defined in terms of its behavior** rather than its implemented representation
  - Nice definition from Wikipedia:
    - An abstract data type is defined indirectly, only by the operations that may be performed on it (i.e. behavior)
    - http://en.wikipedia.org/wiki/Abstract_data_type

Stacks – An Overview

- Stacks:
  - Stacks are an Abstract Data Type
    - They are NOT built into C
  - We must define them and their behaviors
  - So what is a stack?
    - A data structure that stores information in the form of a stack.
    - Consists of a variable number of homogeneous elements
      - i.e. elements of the same type

Stacks – An Overview



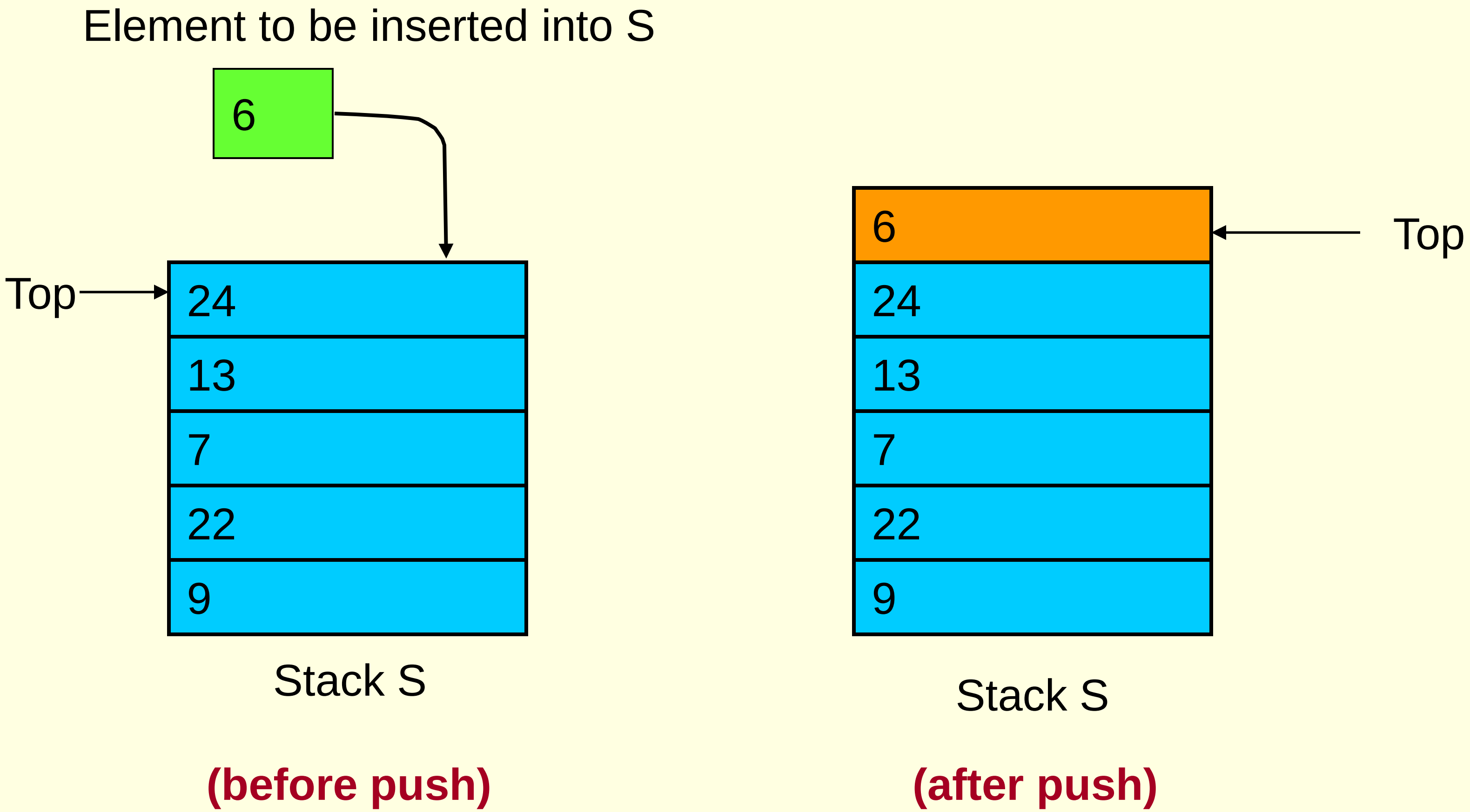
- Stacks:
  - Access Policy:
    - The access policy for a stack is simple: the first element to be removed from the stack is the last element that was placed onto the stack
      - The main idea is that the last item placed on to the stack is the first item removed from the stack
    - Known as the “Last in, First out” access policy
      - LIFO for short
    - The classical example of a stack is cafeteria trays.
      - New, clean trays are added to the top of the stack.
      - and trays are also taken from the top
      - So the last tray in is the first tray taken out

Stacks – An Overview

- Stacks:
  - Basic Operations:
    - PUSH:
      - This PUSHes an item on top of the stack
    - POP:
      - This POPs off the top item in the stack and returns it
  - Other important tidbit:
    - The end of the stack,
      - where PUSHes and POPs occur,
    - is usually referred to as the TOP of the stack

Stacks – An Overview

- PUSH Operation:



Element to be inserted into S

6

Top

24

13

7

22

9

Stack S

(before push)

6

Top

24

13

7

22

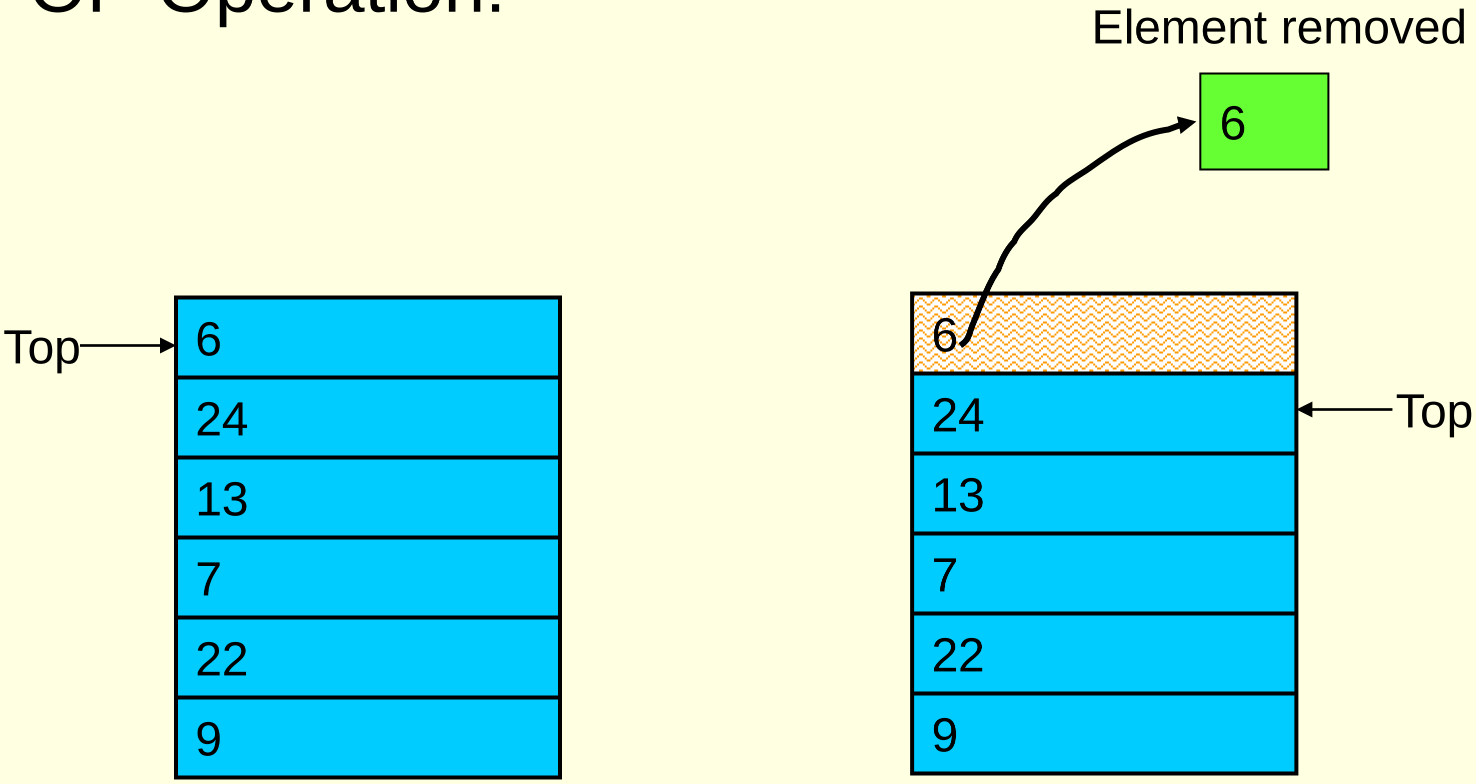
9

Stack S

(after push)

Stacks – An Overview

- POP Operation:



(stack before pop)

(stack after pop)

Stacks – An Overview

- Stacks:
  - Other useful operations:
    - empty:
      - Typically implemented as a boolean function
      - Returns TRUE if no items are in the stacck
    - full:
      - Returns TRUE if no more items can be added to the stack
      - In theory, a stack should NEVER become full
      - Actual implementations do have limits on the number of elements a stack can store
    - top:
      - Simply returns the value at the top of the stack without actually popping the stack.

Stacks – An Overview

- Stacks:
  - Other useful operations:
    - Note:
      - Each of those operations ACCESS the stack
        - But they do NOT modify the stack
      - A PUSH can only be done if the stack isn't full
      - A POP can only be done on a non-empty stack
  - Implementation of a stack:
    - Can be done using both static and dynamic memory
      - Array or a linked list
      - Implemented as an array, the stack could possibly become full
      - As a linked list, this is MUCH LESS LIKELY to occur
    - We will cover detailed implementations NEXT TIME.

Using Stacks

- So when is stack useful?
  - When data needs to be stored and then retrieved in reverse order
  - There are several examples/applications outside the scope of this class
    - Be patient and they will come up
  - For now, we go over two classical examples…
    - This examples help facilitate learning about stacks and their operations.

Stack Application(s)

- Evaluating Arithmetic Expressions
  - Consider the expression 5 * 3 + 2 + 6 * 4
  - How do we evaluate this?
    - DUH
    - No, but seriously, think about what happens during the evaluation
    - We multiply 5 and 3 to get 15, and then we add 2 to that result to get 17.
    - Then we store that off in our head somewhere.
    - We then multiply 6 and 4 to get 24
    - Lastly, we then retrieve the stored value (17) and add it to 24 to get the result…41.

Stack Application(s)

- Evaluating Arithmetic Expressions
  - Consider the expression 5 * 3 + 2 + 6 * 4
  - That was only easy cuz we knowza some math and rules of precedence, etc.
    - What if you didn’t know those rules?
    - Well, there’s an easy way of writing out this sequence of events.
  - It does seem weird at first glance…but it works!
  - 5 3 * 2 + 6 4 * +
    - you read this left to right
    - the operators are ALWAYS in the correct evaluation order
  - This notation is called **postfix** notation.

Stack Application(s)

- Basically, there are 3 types of notations for expressions
  - **Infix**: operator is between operands
    - A + B
  - **Postfix**: operator follows operands
    - A B +
  - **Prefix**: operator comes before operands
    - + A B

  - Again, in a postfix expression, operators are ALWAYS in correct evaluation order.

Stack Application(s)

- Evaluation of infix expressions has 2 basic steps:
  - Convert infix expression to a postfix expression.
  - Evaluate the newly converted postfix expression.

- And guess what…
  - Stacks are useful in both of these steps
- Let's start with seeing how to actually evaluate that crazy looking expression

Stack Application(s)

- Evaluating a Postfix Expression (A B +)
  - Of course, we use a stack
    - Each operator in a postfix expression refers to the previous two operands
    - When you read an operand
      - PUSH it onto the stack
    - When you read an operator, it's associated operands are POPed off the stack
      - The indicated operation (based on the operand) is performed on the two operators
      - Result is PUSHed back onto the stack so it can be available for use as an operand for the next operator.
      - Process stops when there are no more operators in expression
      - Final result is obtained by popping off remaining value in stack.

Stack Application(s)

- Evaluating a Postfix Expression
  - Consider the simple expression: 5 * 3 + 2 + 6 * 4
  - As mentioned, this converts to the following postfix expression: 5 3 * 2 + 6 4 * +
  - So follow the rules and evaluate this!

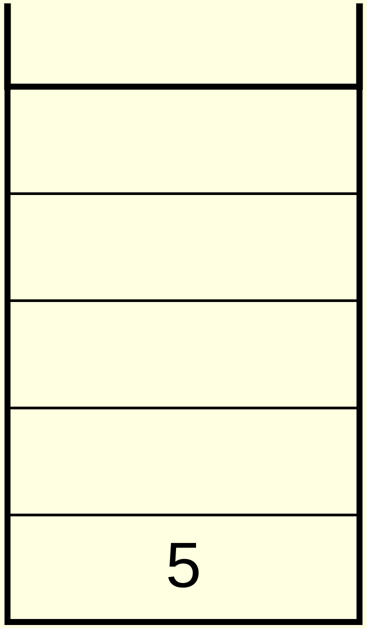
5 3 * 2 + 6 4 * +

- Step 1: We have an operand, 5. What do we do?

The rule stated:

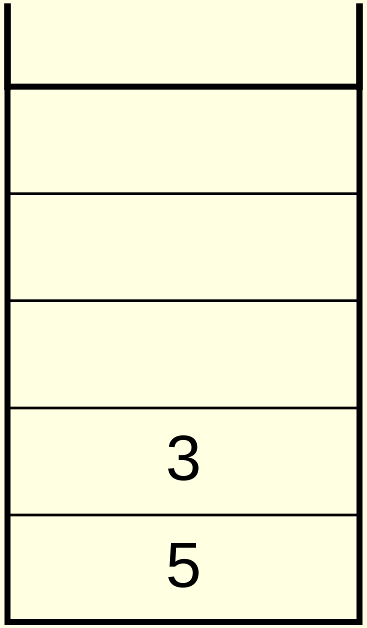
When you encounter an operand, PUSH it onto the stack.

So we PUSH 5 onto the stack.



5 **3** * 2 + 6 4 * +

- Step 2: PUSH 3 on the stack



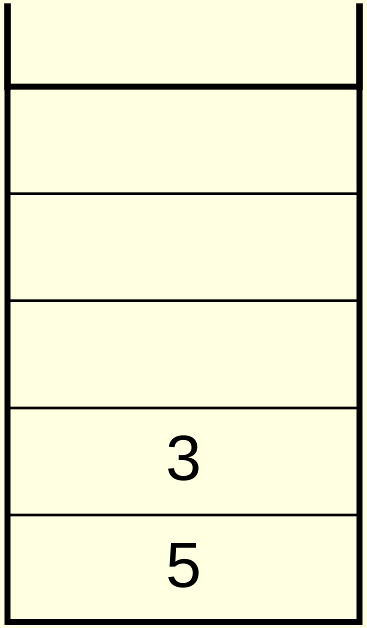
3

5

5 3 * 2 + 6 4 * +

- Step 3: We have an operator! What do we do?

1. POP the top two operands off the stack.
2. Perform the indicated operation.
3. PUSH the result back onto the stack.

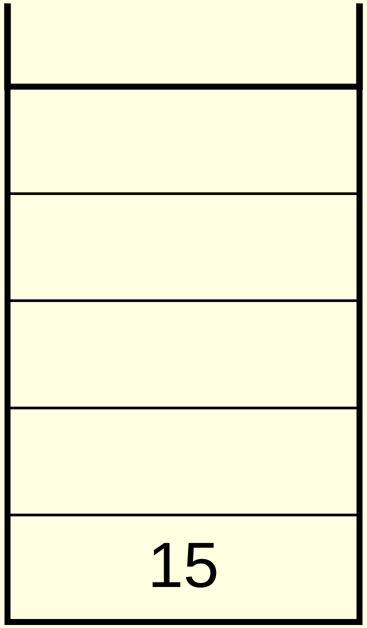


1. So POP 3 and 5.
2. 5*3 = 15
3. PUSH 15 back on the stack

5 3 * 2 + 6 4 * +

- Step 3: We have an operator! What do we do?

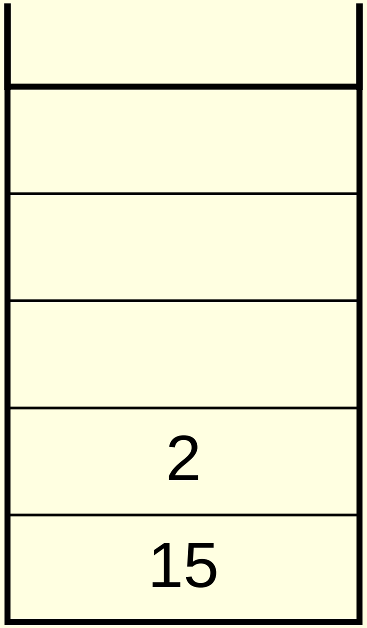
1. POP the top two operands off the stack.
2. Perform the indicated operation.
3. PUSH the result back onto the stack.



1. So POP 3 and 5.
2. 5*3 = 15
3. PUSH 15 back on the stack

5 3 * **2** + 6 4 * +

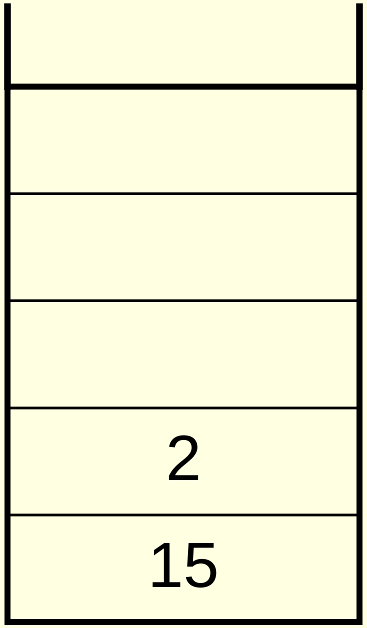
- Step 4: PUSH 2 on the stack



5 3 * 2 + 6 4 * +

- Step 5: We have an operator! What do we do?

1. POP the top two operands off the stack.
2. Perform the indicated operation.
3. PUSH the result back onto the stack.

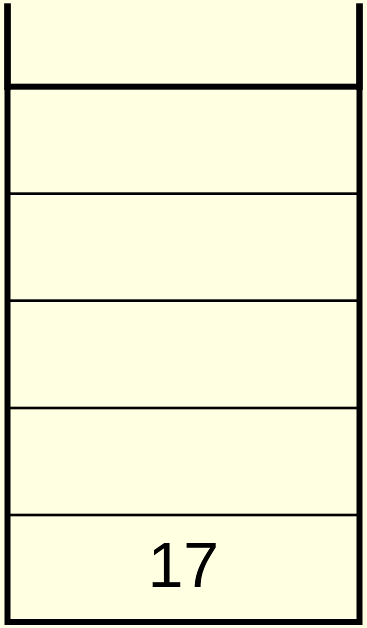


1. So POP 2 and 15.
2. 15 + 2 = 17
3. PUSH 17 back on the stack

5 3 * 2 $\underline{+}$ 6 4 * +

- Step 5: We have an operator! What do we do?

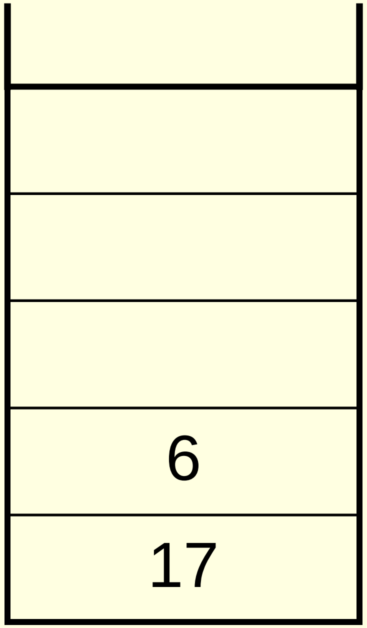
1. POP the top two operands off the stack.
2. Perform the indicated operation.
3. PUSH the result back onto the stack.



1. So POP 2 and 15.
2. 15 + 2 = 17
3. PUSH 17 back on the stack

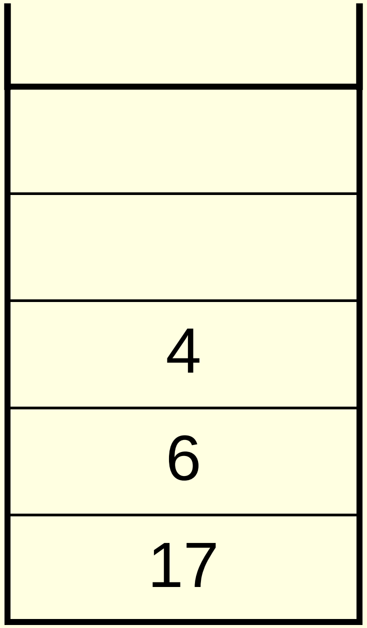
5 3 * 2 + **<u>6</u>** 4 * +

- Step 6: PUSH 6 on the stack



5 3 * 2 + 6 **4** * +

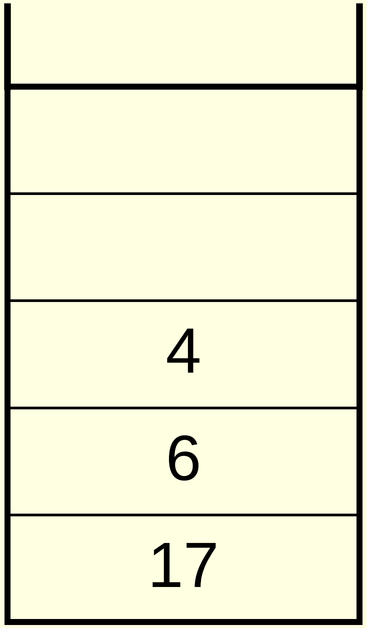
- Step 7: PUSH 4 on the stack



5 3 * 2 + 6 4 * +

- Step 8: We have an operator! What do we do?

1. POP the top two operands off the stack.
2. Perform the indicated operation.
3. PUSH the result back onto the stack.

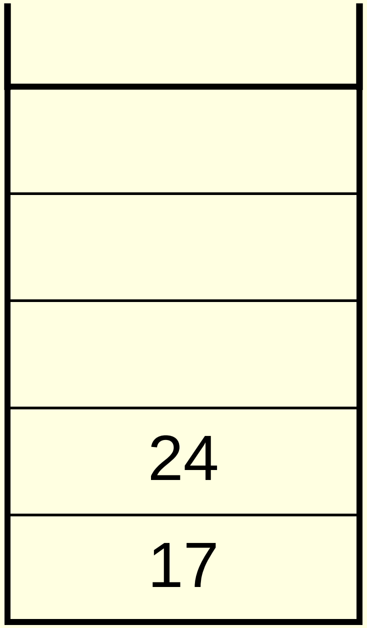


1. So POP 4 and 6.
2. 6 * 4 = 24
3. PUSH 24 back on the stack

5 3 * 2 + 6 4 * +

- Step 8: We have an operator! What do we do?

1. POP the top two operands off the stack.
2. Perform the indicated operation.
3. PUSH the result back onto the stack.



24

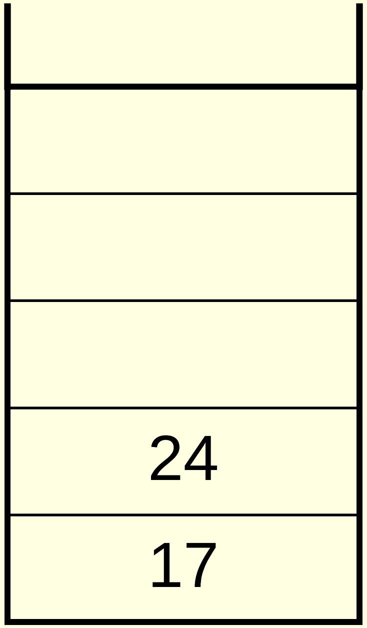
17

1. So POP 4 and 6.
2. 6 * 4 = 24
3. PUSH 24 back on the stack

5 3 * 2 + 6 4 * +

- Step 9: We have an operator! What do we do?

1. POP the top two operands off the stack.
2. Perform the indicated operation.
3. PUSH the result back onto the stack.

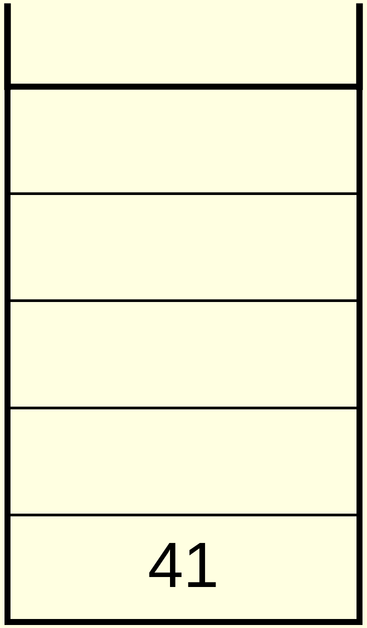


1. So POP 24 and 17.
2. 17 + 24 = 41
3. PUSH 41 back on the stack

5 3 * 2 + 6 4 * +

- Step 9: We have an operator! What do we do?

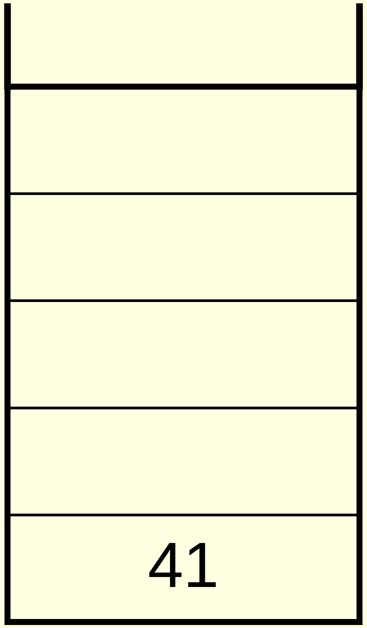
1. POP the top two operands off the stack.
2. Perform the indicated operation.
3. PUSH the result back onto the stack.



1. So POP 24 and 17.
2. 17 + 24 = 41
3. PUSH 41 back on the stack

5 3 * 2 + 6 4 * +

- Step 10: There are no more operators. So pop the final value off the stack.



Result is 41

We're Done!

Brief Interlude: Human Stupidity

Stack Application(s)

- Again, there are 2 steps to evaluate infix expressions.
  1. Evaluate the postfix expression (once converted)
     - Been there, done that
       - (for those that were sleeping, that was the long previous example)
  2. But before we can evaluate, we must first convert the infix exp. to a postfix exp.
     - Infix: 5 * 3 + 2 + 6 * 4
     - Postfix: 5 3 * 2 + 6 4 * +
     - How do we do this…

Stack Application(s)

- Converting Infix Exp. to Postfix Exp.
  - Again, we use a stack
    - But now, this is strictly an “operator only” stack
      - Only the operators are stored on the stack
      - Upon reading an operand, the operand is immediately placed into output list (printed out straight away).
    - There are several rules on how this stack should be used
      - But with an example, it should be easy to understand

Stack Application(s)

- Converting Infix Exp. to Postfix Exp.
  - Rules

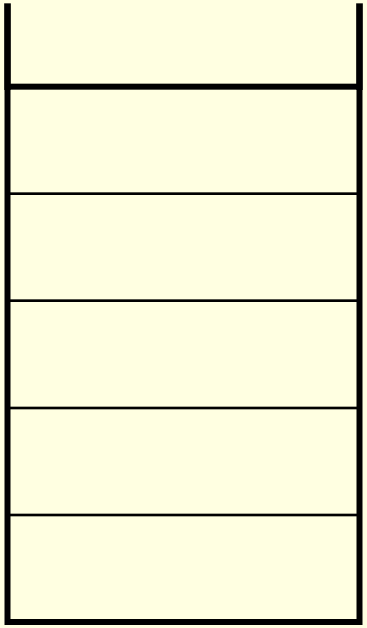
1. Assume the operation is a legal one (meaning, it is possible to evaluate it).
2. Upon reading an operand, it is immediately placed into the output list (printed straight away).
3. Only the operators are placed in the stack.
4. To start, the stack is empty. The infix expression is read left to right.
5. The first operator read is pushed directly onto the stack. For all subsequent operators, the priority of the “incoming-operator” (the one being read) will be compared with the operator on the top of the stack.
6. If the priority of the incoming-operator is higher than the priority of the operator on the top of the stack, then the incoming-operator will be simply PUSHed on the stack.
7. If the priority of the incoming-operator is same or lower than the priority of the operator at the top of the stack, then the operator at top of the stack will be POPed and printed on the output expression.
8. The process is repeated if the priority of the incoming-operator is still same or lower than the next operator-in-the stack.
9. When a left parenthesis is encountered in the expression it is immediately pushed on the stack, as it has the highest priority. However, once it is inside the stack, all other operators are pushed on top of it, as its inside-stack priority is lowest.
10. When a right parenthesis is encountered, all operators up to the left parenthesis are popped from the stack and printed out. The left and right parentheses will be discarded. When all characters from the input infix expression have been read, the operators remaining inside the stack, are printed out in the order in which they are popped.

Stack Application(s)

- Converting Infix Exp. to Postfix Exp.
  - Rules
    - One more thing, before we begin, we must know the order of precedence for the operators
    - The priority is as follows, with the first being the top priority (highest precedence)
      1. (Left parenthesis inside the expression
      2. * /
      3. + -
      4. (Left parenthesis inside the stack
    - The left parenthesis has the highest priority when it is read from the expression, but once it is on the stack, it assumes the lowest priority.

5 * 3 + 2 + 6 * 4

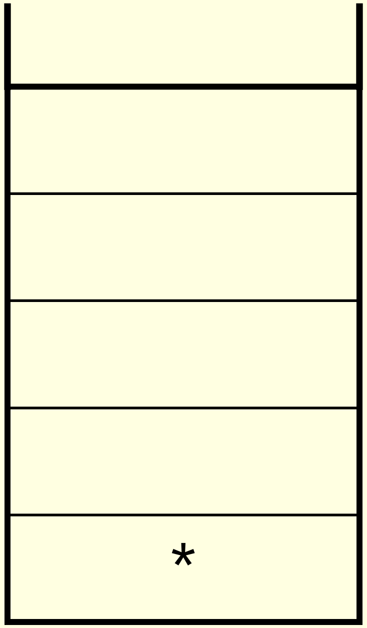
- Step 1: 5 is an operand. It is placed directly onto output list.



Resulting Postfix Expression: 5

5 * 3 + 2 + 6 * 4

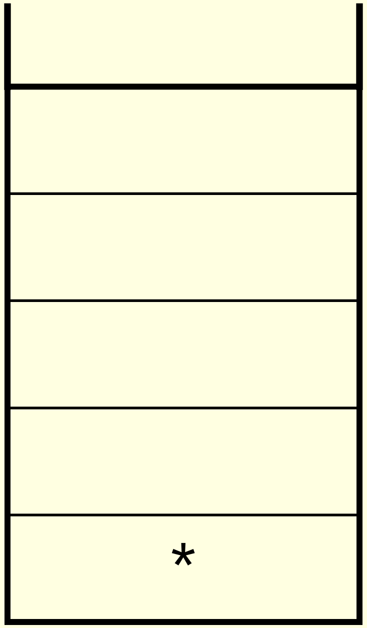
- Step 2: * is an operator. The stack is empty; so PUSH * into the stack.



Resulting Postfix Expression: 5

5 * **3** + 2 + 6 * 4

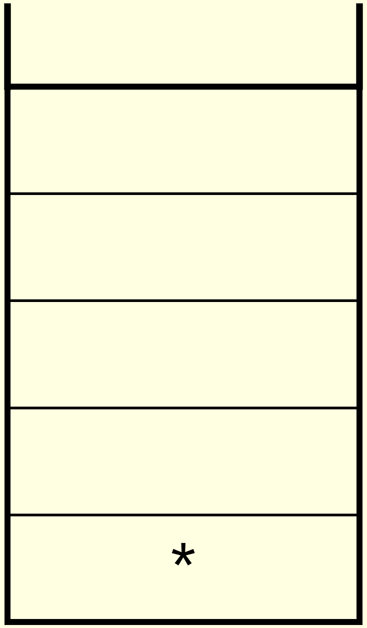
- Step 3: 3 is an operand. It is placed directly onto output list.



Resulting Postfix Expression: 5 3

5 * 3 **+** 2 + 6 * 4

- Step 4: + is an operator. The stack is not empty; compare precedence of + to *.



Resulting Postfix Expression: 5 3

Stack Application(s)

- Converting Infix Exp. to Postfix Exp.
  - Rules
    - One more thing, before we begin, we must know the order of precedence for the operators
    - **The priority is as follows, with the first being the top priority (highest precedence)**
      1. **(Left parenthesis inside the expression**
      2. *** /**
      3. **+ -**
      4. **(Left parenthesis inside the stack**
    - The left parenthesis has the highest priority when it is read from the expression, but once it is on the stack, it assumes the lowest priority.

Stack Application(s)

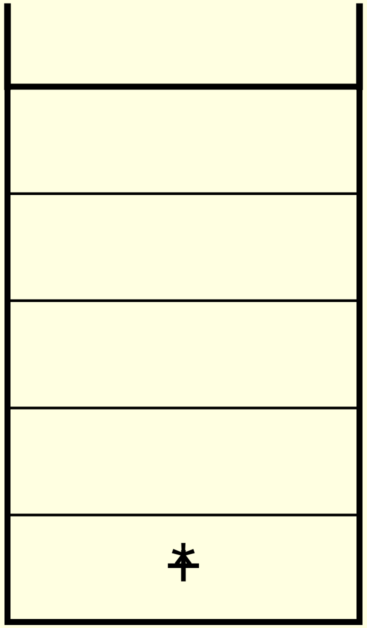
- Converting Infix Exp. to Postfix Exp.
  - Rules
    1. Assume the operation is a legal one (meaning, it is possible to evaluate it).
    2. Upon reading an operand, it is immediately placed into the output list (printed straight away).
    3. Only the operators are placed in the stack.
    4. To start, the stack is empty. The infix expression is read left to right.
    5. The first operator read is pushed directly onto the stack. For all subsequent operators, the priority of the "incoming-operator" (the one being read) will be compared with the operator on the top of the stack.
    6. If the priority of the incoming-operator is higher than the priority of the operator on the top of the stack, then the incoming-operator will be simply PUSHed on the stack.
    7. **If the priority of the incoming-operator is same or lower than the priority of the operator at the top of the stack, then the operator at top of the stack will be POPed and printed on the output expression.**
    8. The process is repeated if the priority of the incoming-operator is still same or lower than the next operator-in-the stack.
    9. When a left parenthesis is encountered in the expression it is immediately pushed on the stack, as it has the highest priority. However, once it is inside the stack, all other operators are pushed on top of it, as its inside-stack priority is lowest.
    10. When a right parenthesis is encountered, all operators up to the left parenthesis are popped from the stack and printed out. The left and right parentheses will be discarded. When all characters from the input infix expression have been read, the operators remaining inside the stack, are printed out in the order in which they are popped.

5 * 3 <u>+</u> 2 + 6 * 4

- Step 4: + is an operator. The stack is not empty; compare precedence of + to *.

+ is lower priority than *.

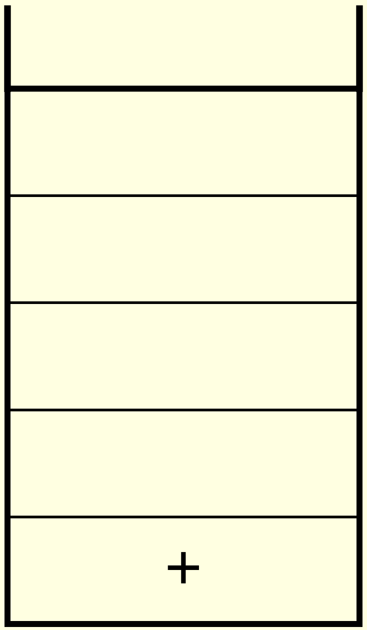
So we POP * and PUSH + onto the stack.



Resulting Postfix Expression: 5 3 *

5 * 3 + **2** + 6 * 4

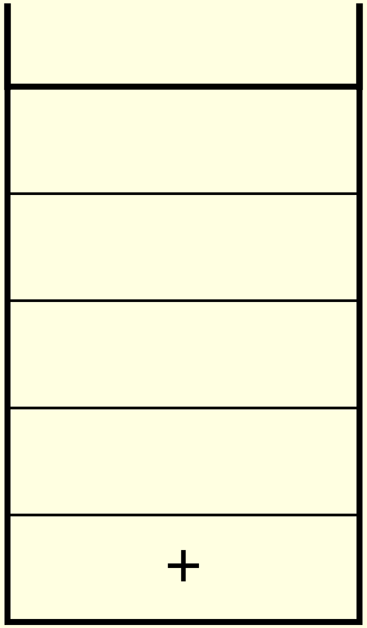
- Step 5: 2 is an operand. It is placed directly onto output list.



Resulting Postfix Expression: 5 3 * 2

5 * 3 + 2 $\underline{+}$ 6 * 4

- Step 6: + is an operator. The stack is not empty; compare precedence of + to +.



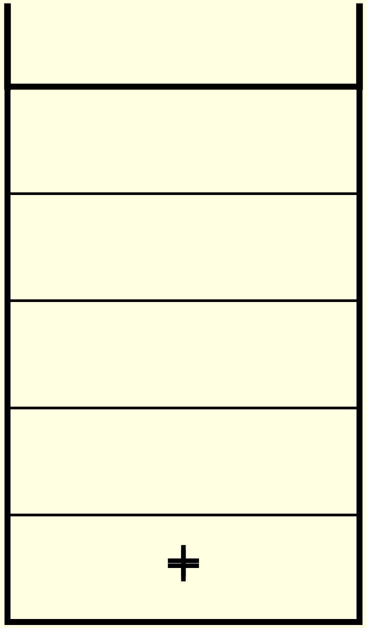
Resulting Postfix Expression: 5 3 * 2

5 * 3 + 2 + 6 * 4

- Step 6: + is an operator. The stack is not empty; compare precedence of + to +.

+ is same priority as +.

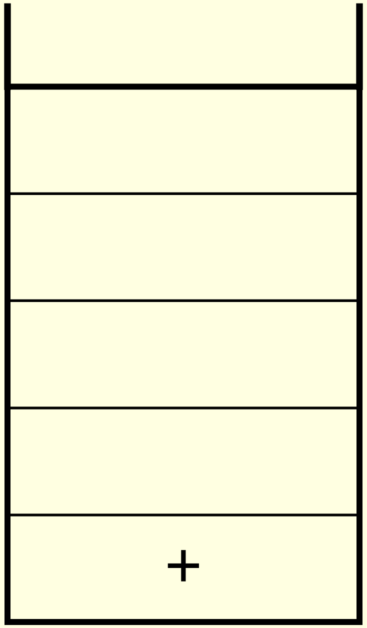
So we POP + and PUSH + onto the stack.



Resulting Postfix Expression: 5 3 * 2 +

5 * 3 + 2 + **6** * 4

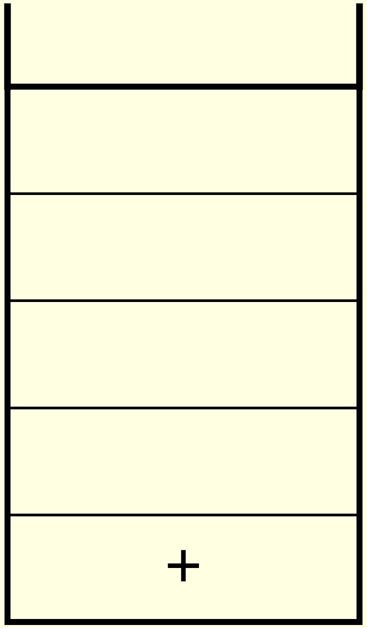
- Step 7: 6 is an operand. It is placed directly onto output list.



Resulting Postfix Expression: 5 3 * 2 + 6

5 * 3 + 2 + 6 * 4

- Step 8: * is an operator. The stack is not empty; compare precedence of * to +.



Resulting Postfix Expression: 5 3 * 2 + 6

Stack Application(s)

- Converting Infix Exp. to Postfix Exp.
  - Rules
    - One more thing, before we begin, we must know the order of precedence for the operators
    - **The priority is as follows, with the first being the top priority (highest precedence)**
      1. **(Left parenthesis inside the expression**
      2. ***   /**
      3. **+   -**
      4. **(Left parenthesis inside the stack**
    - The left parenthesis has the highest priority when it is read from the expression, but once it is on the stack, it assumes the lowest priority.

Stack Application(s)

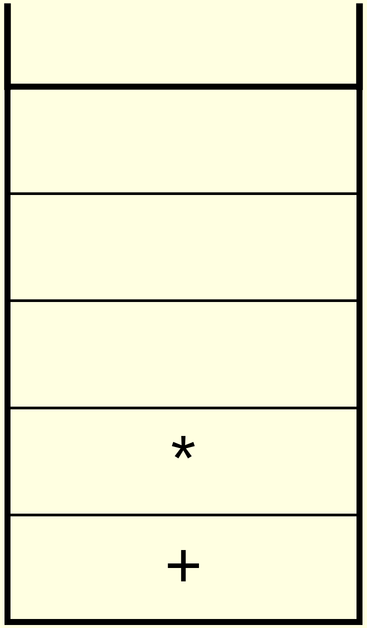
- Converting Infix Exp. to Postfix Exp.
  - Rules
    1. Assume the operation is a legal one (meaning, it is possible to evaluate it).
    2. Upon reading an operand, it is immediately placed into the output list (printed straight away).
    3. Only the operators are placed in the stack.
    4. To start, the stack is empty. The infix expression is read left to right.
    5. The first operator read is pushed directly onto the stack. For all subsequent operators, the priority of the "incoming-operator" (the one being read) will be compared with the operator on the top of the stack.
    6. **If the priority of the incoming-operator is higher than the priority of the operator on the top of the stack, then the incoming-operator will be simply PUSHed on the stack.**
    7. If the priority of the incoming-operator is same or lower than the priority of the operator at the top of the stack, then the operator at top of the stack will be POPed and printed on the output expression.
    8. The process is repeated if the priority of the incoming-operator is still same or lower than the next operator-in-the stack.
    9. When a left parenthesis is encountered in the expression it is immediately pushed on the stack, as it has the highest priority. However, once it is inside the stack, all other operators are pushed on top of it, as its inside-stack priority is lowest.
    10. When a right parenthesis is encountered, all operators up to the left parenthesis are popped from the stack and printed out. The left and right parentheses will be discarded. When all characters from the input infix expression have been read, the operators remaining inside the stack, are printed out in the order in which they are popped.

5 * 3 + 2 + 6 * 4

- Step 8: * is an operator. The stack is not empty; compare precedence of * to +.

* is a higher priority than +.

So we simply PUSH * onto the stack.



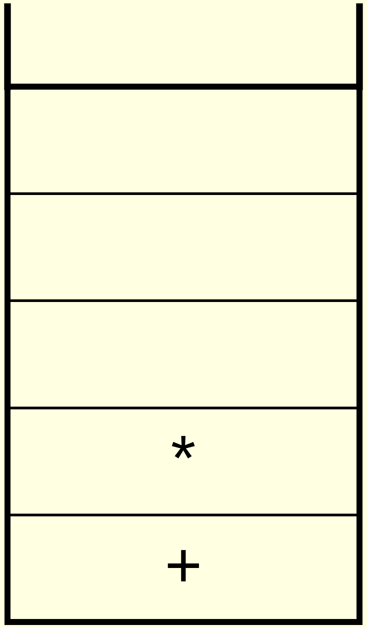
*

+

Resulting Postfix Expression: 5 3 * 2 + 6

5 * 3 + 2 + 6 * **4**

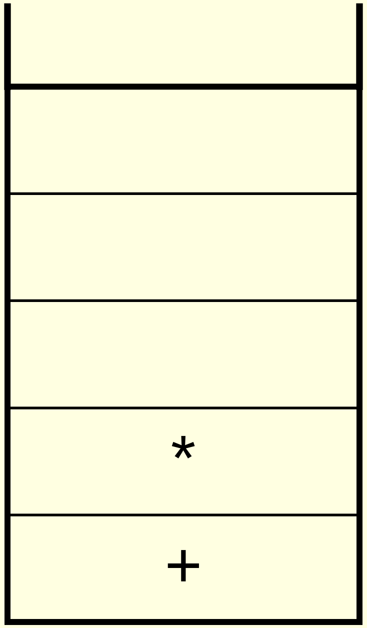
- Step 9: 4 is an operand. It is placed directly onto output list.



Resulting Postfix Expression: 5 3 * 2 + 6 4

5 * 3 + 2 + 6 * 4

- Step 10: Infix exp. has been completely read. POP remaining operators that are in the stack.



*

+

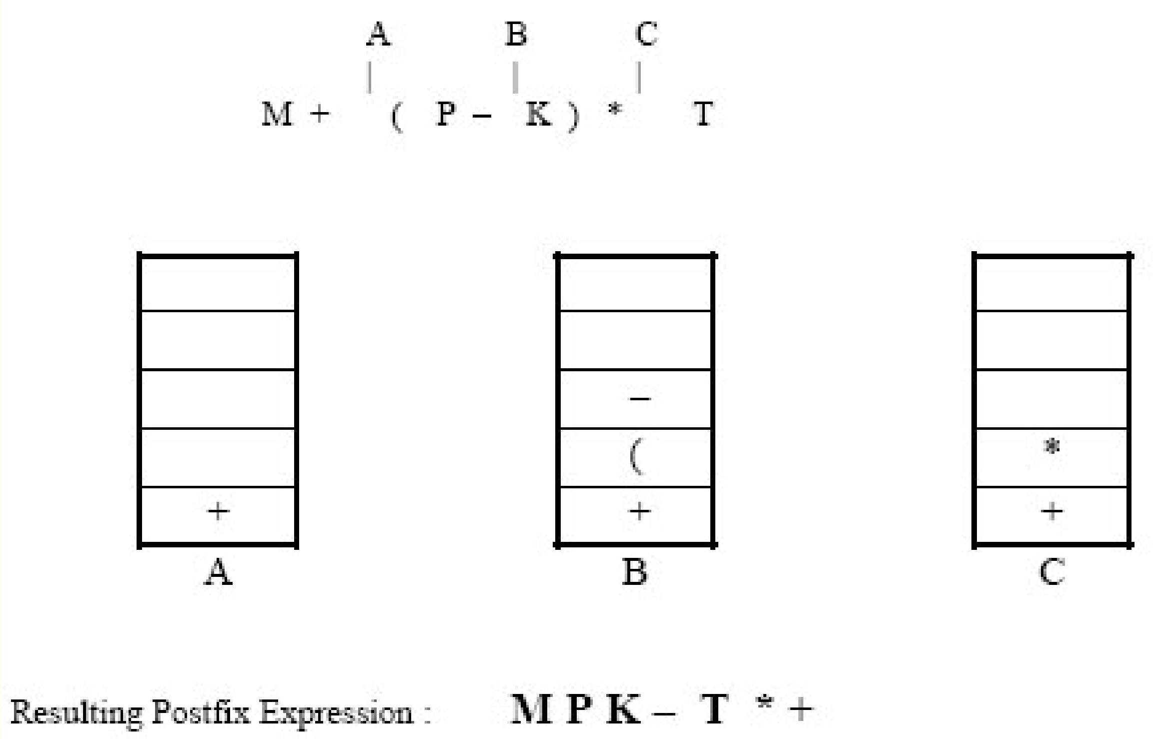
And now we’re done.
We have an equivalent
postfix expression.

Resulting Postfix Expression: 5 3 * 2 + 6 4 * +

Stack Application(s)

- Two more examples:

The contents of the operator stack at the indicated points in the infix expressions (points A, B and C) are shown below for each case

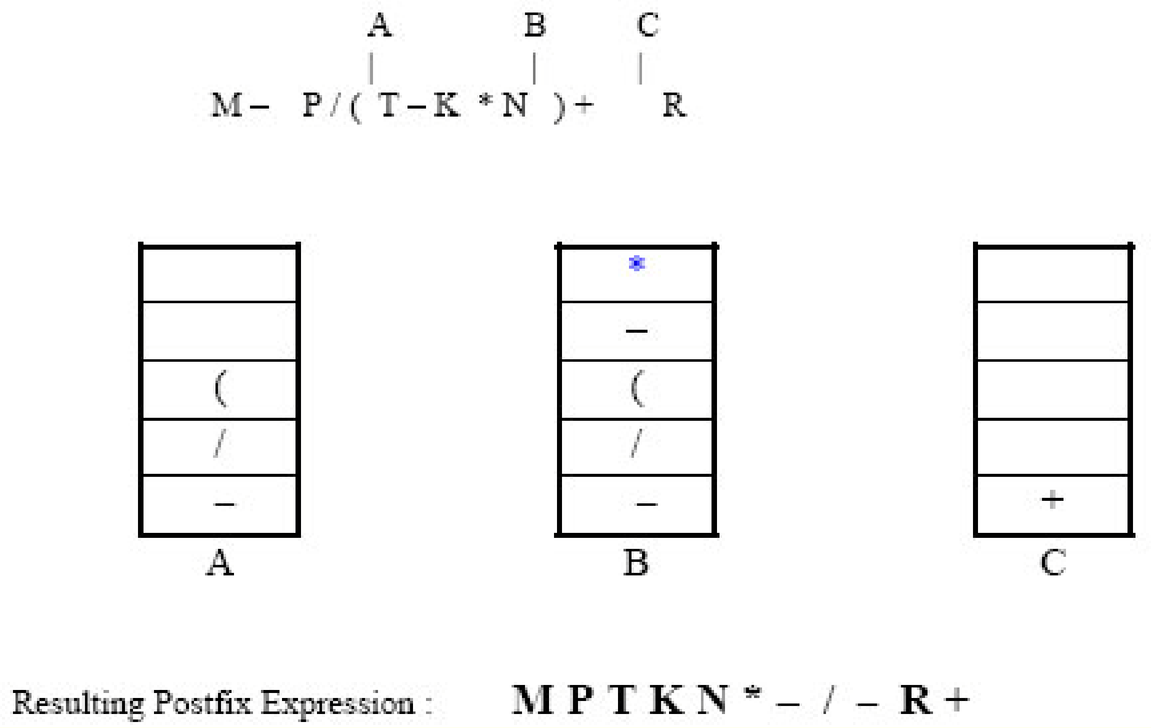


Resulting Postfix Expression : M P K – T * +

Stack Application(s)

- ## Last example:

The contents of the operator stack at the indicated points in the infix expressions (points A, B and C) are shown below for each case



Resulting Postfix Expression : M P T K N * – / – R +

Stack Application(s)

- You now know how to:
  1. Convert an Infix expression to a Postfix expression
  2. And then evaluate that resulting expression

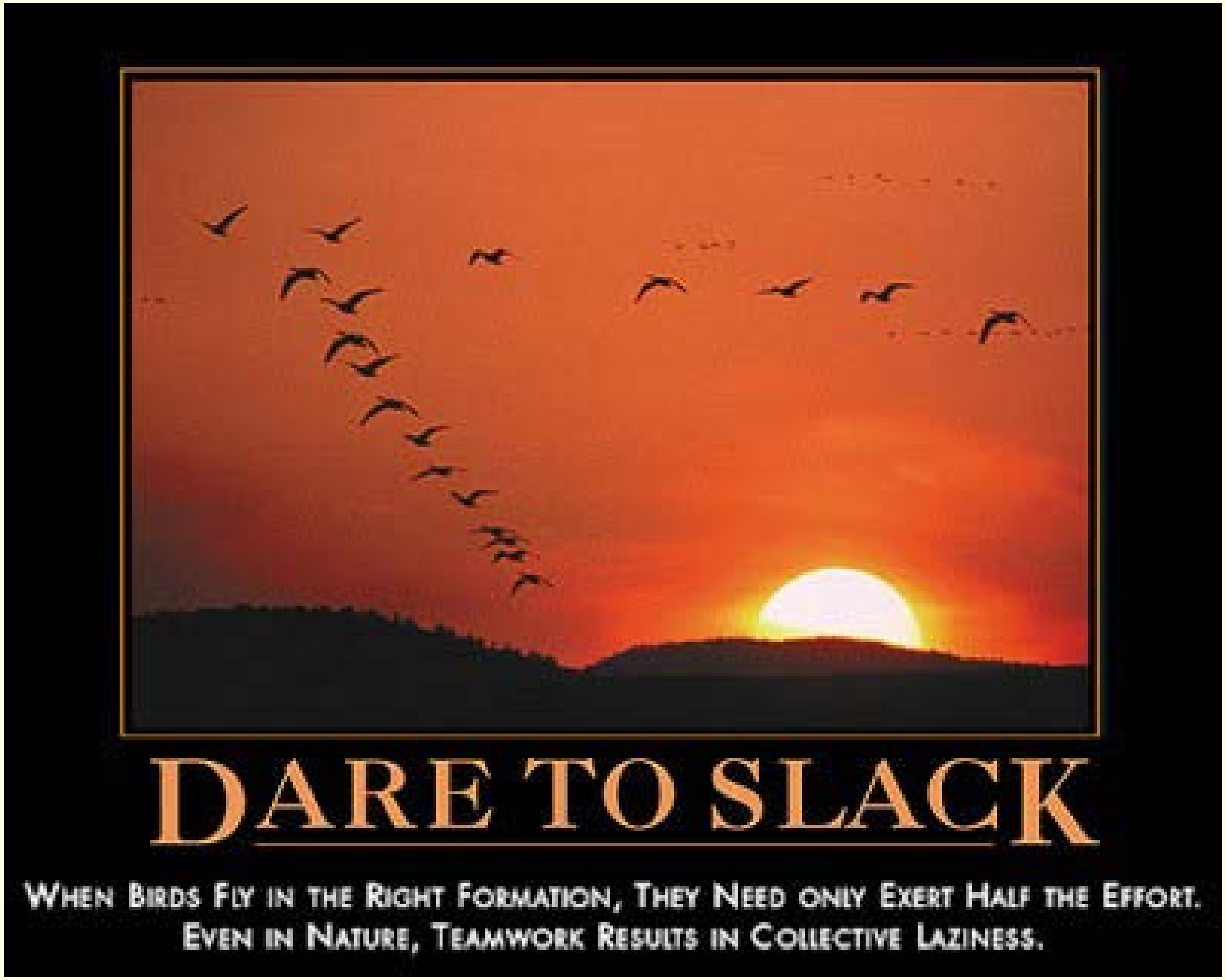
Stack Application(s)

WASN’T

THAT

DANDY!

Daily Demotivator


DARE TO SLACK
When Birds Fly in the Right Formation, They Need only Exert Half the Effort.
Even in Nature, Teamwork Results in Collective Laziness.

Stacks & Their Applications (Postfix/Infix)

Computer Science Department
University of Central Florida

COP 3502 – Computer Science I