Class 26 Notes

35 456 121 + * 81 +

The multi-colored expression above a postfix expression.

EVERY computer science student should understand postfix . If you are at a Geek party and ignorant of postfix …well you would be very, very embarrassed and even politely asked to leave.

Infix and postfix expressions

How would you write a program that evaluates an arbitrary (string) expression? such as

((23 + 45)* (12-34+45) + (7 -4)) ?

Your program would have to take parentheses and operator precedence into account for **ANY** expression. This is not particularly easy or efficient.

There is a better, easier, and more efficient way to do this. And, that's what we will be looking at now.

Infix Notation

Here's simple numerical expression for adding 2 and 3: ***2+3***. Any first grader understands 3+2.

The numbers *2* and *3* are called ***operands*** and the symbol + is an ***operator.***

We say that the expression *2+3* is written in ***infix*** notation because the operator (+) comes **"in-between"** the two operands, 2 and 3.

Infix notation is the algebraic notation which we most commonly use. In fact, it is probably the only notation that you have ever seen.

Expressions like *2+3*4* and *3*4+2*5* are written in infix notation.

Look at the expression *2+3*4*. Although the plus sign is written first, *the* multiplication is performed first. That's because * has precedence over +. Multiplication has higher priority than addition.

When evaluating the expression *3*4+2*5,* the multiplication (3*4) is done first, then the second multiplication (2*5), and finally the two products are added giving a value of 22. Again, evaluation depends on ***operator precedence/priority***.

When using infix notation, the order of operations depends on the priority/precedence of the operators not the order in which the operators are written. In 3+4*5, the + is *written* first but the * is *performed* first

To complicate things even more, the order of operations can be changed by inserting parentheses.

While infix notation is common and natural for us, other algebraic notations exist where the **order of operations corresponds to the order in which the operators are written**. One such algebraic notation is called ***postfix* notation**.

Postfix Notation

Suppose we want to write an expression to add 3 and 2

- Using infix notation, we write 3 + 2
  the operators come **between** the operands
  + is the operator; 3 and 2 are operands

- Using postfix notation, we write 3 2 +
  operator + **follows** the operands 3 and 2
  That's why it is called **POST**fix

The infix expression expression (2+3) * (5-7) has an equivalent postfix representation: 2 3 **+** 5 7 **-** *****. Again, notice the operators are written in the order that they will be performed: **+** , **-** , and lastly, *****

Don't be concerned if you do not know how I got the postfix or how to evaluate postfix. That is coming. For now, just be aware that there is more than one way to write arithmetic expressions.

In a postfix expression,

- **an operator (+, - * /) is written after its operands.**
  2 3 + means add 2 and 3 → the numbers thatprecede the +

- **For postfix expressions, operations are performed in the order in which they are written (left to right).**

- **Postfix expressions do not require parentheses**. (That's great!)

Here are some examples of equivalent infix and postfix expressions. Again, at this point you probably do not see how they are equivalent.

- the infix expression *2+3*4; in* postfix it is 2 3 4*+
  Notice * is done first, then + → and that is their order in postfix

- the infix expression *3*4+2**5 is *3 4*2 5** + in postfix notation
  Here the order of operations is * then * then + → and that is the order in which they are written in postfix
  .
- the infix expression 3*(4+2)*5 is 3 4 2 + * 5 * in postfix notation
  Here + is first the *(green) then * (blue). And that's how they appear in the postfix version postfix +, then * and finally *

We are going to attack two problems

1. How to convert an infix expression to an equivalent postfix expression
   For example 3*(4+2)*5 in infix is equivalent to 3 4 2 + * 5 * in postfix

2. How to evaluate a postfix expression → 3 4 2 + * 5 * = 90

Problem 2 is the easier and very intuitive, so I will start with that.

Evaluation of postfix expressions.

For simplicity, I will assume:

- The only operators are +, -, * and /
- - mean subtraction, not negation
- Division is integer division
- The numbers and operators in a postfix expression are separated by spaces. For example, 23 30 + 10 * or 30 40 + 2 80 - *

Here is how to evaluate the postfix expression using a **stack of Integer**

- Scan the string left to right.

- When you read a number, push/save it on the stack; Its operator will follow

- when you read an operator (+ - * /), pop the top two numbers off the stack, perform the operation, and push the result back on the stack.

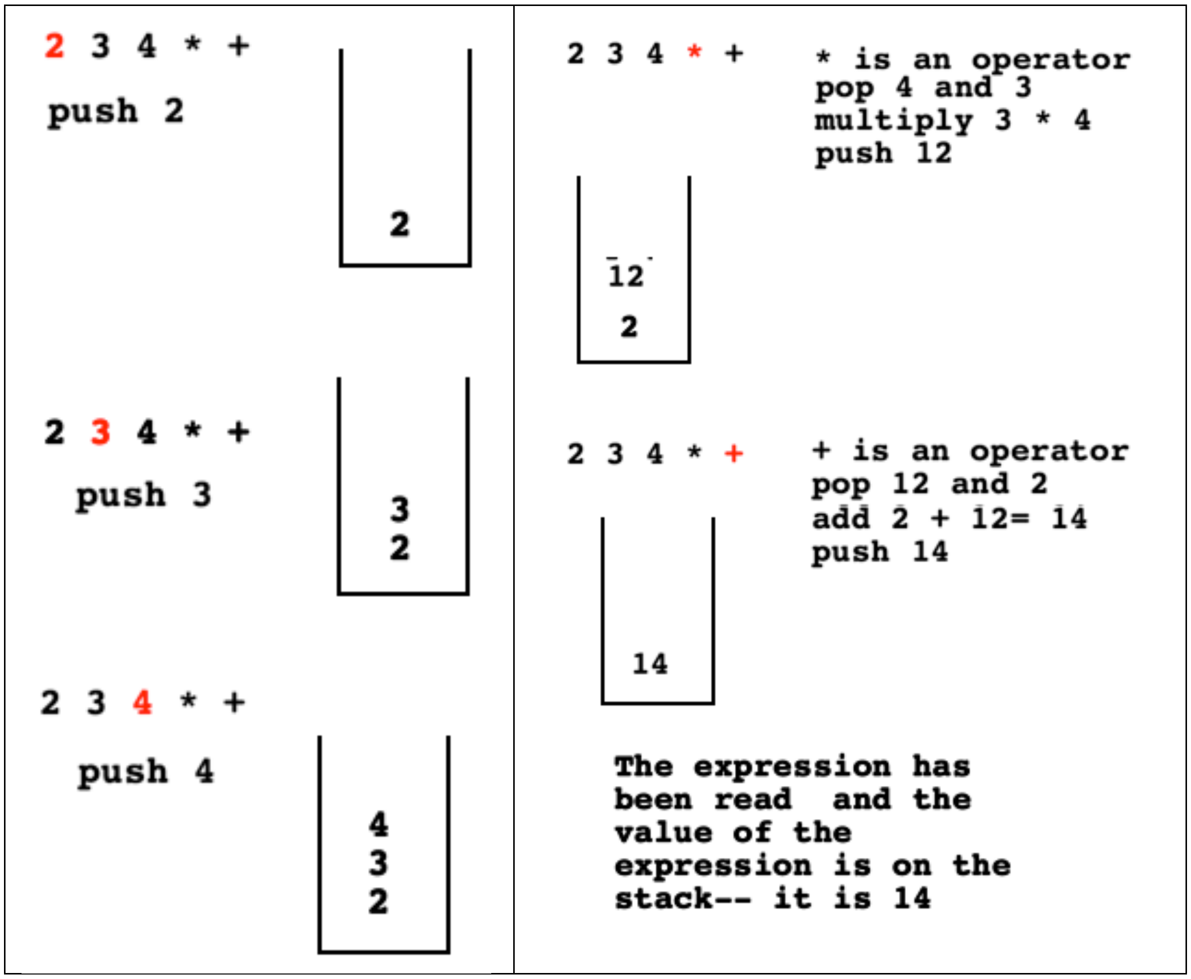
- When you are finished scanning the expression, the final value remains on the stack.

The next example illustrates the technique. The "current " number or operator is shown in red.

When reading the next example, read the first column then the second.

Example: (Read the first column, then second column)

The red number is the current number being read
Evaluate the postfix expression *2 3 4* * +
Start with an empty stack



```
2 3 4 * +
push 2
```

```
2 3 4 * +
push 3
```

```
2 3 4 * +
push 4
```

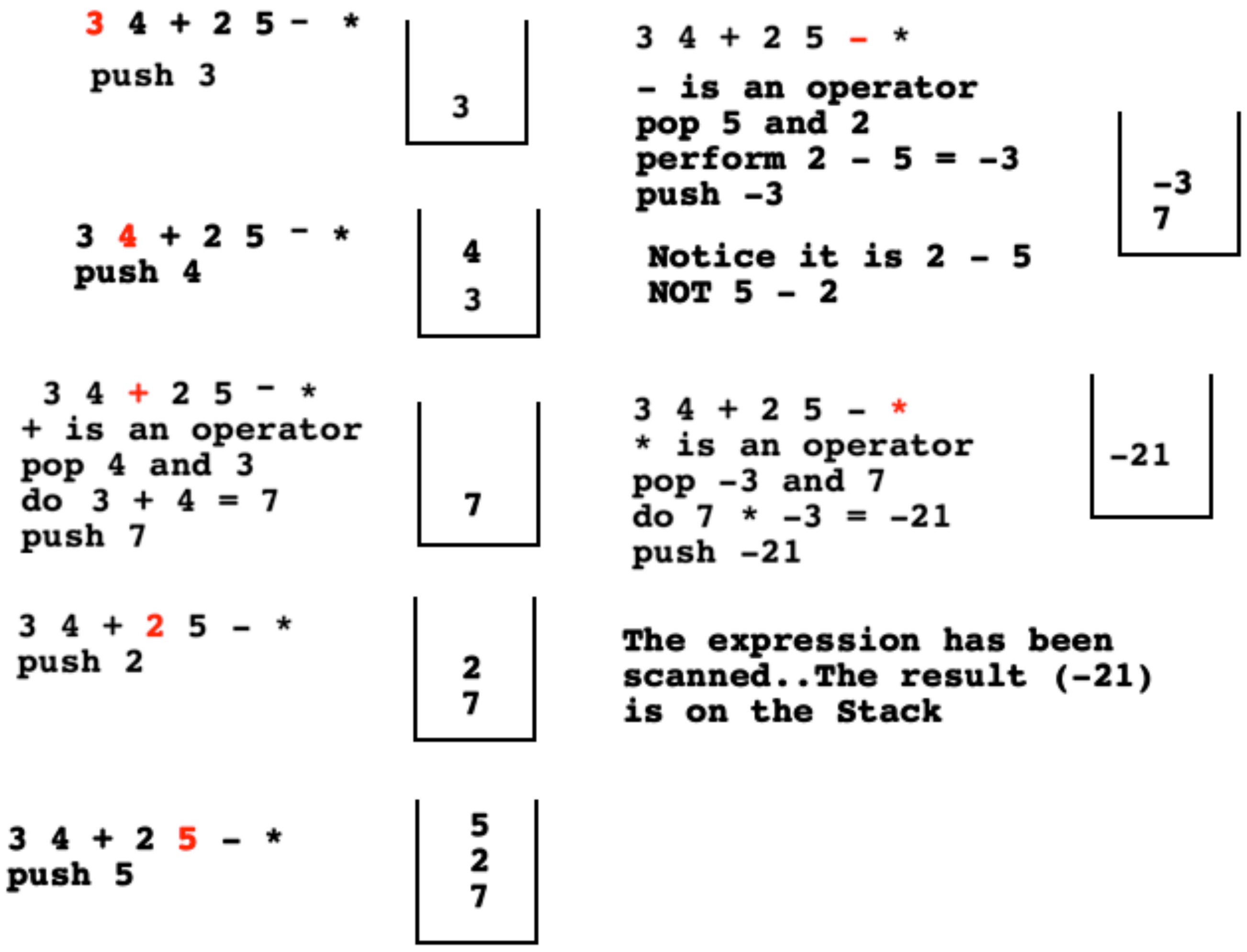
```
2 3 4 * +
* is an operator
pop 4 and 3
multiply 3 * 4
push 12
```

```
2 3 4 * +
+ is an operator
pop 12 and 2
add 2 + 12= 14
push 14
```

```
The expression has
been read  and the
value of the
expression is on the
stack-- it is 14
```

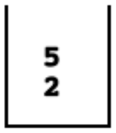
The infix expression (3 + 4) * (2 - 5) = -21
The postfix equivalent **3 4 + 2 5 - *.**

Here is how we evaluate the postfix with a stack:



```
3 4 + 2 5 - *
push 3

3 4 + 2 5 - *
push 4

3 4 + 2 5 - *
+ is an operator
pop 4 and 3
do 3 + 4 = 7
push 7

3 4 + 2 5 - *
push 2

3 4 + 2 5 - *
push 5

3 4 + 2 5 - *
- is an operator
pop 5 and 2
perform 2 - 5 = -3
push -3

Notice it is 2 - 5
NOT 5 - 2

3 4 + 2 5 - *
* is an operator
pop -3 and 7
do 7 * -3 = -21
push -21

The expression has been
scanned..The result (-21)
is on the Stack
```

Be careful with – and / operators
Suppose you are evaluating 2 5 – (which is 2-5 in infix notation)
You would push 2 and then 5 the stack



When you pop the numbers 5 and 2 make sure that you do the operation in the correct order 2 – 5 (= -3) not 5 – 2 (= 3). Do second pop() – first pop(). Same for /. (Because a*b= b*a and a+b = b+ a it is not so crucial for * and +)

Algorithm to evaluate postfix expressions.

Assume

- the only operators allowed are the +,-,*, and /, where / signifies **integer** division and - signifies subtraction (not negation)
- all input postfix expressions are correct.
- There is a space separating numbers and operators

Thus, a typical postfix string is 23 12 * 7 3 / +, which in infix notation is 23*12 + 7/3 (value is 8).

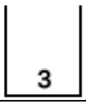
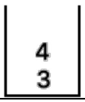
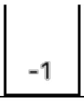
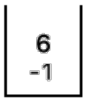
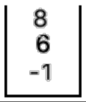
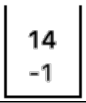
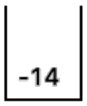
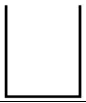
Making these assumptions, the algorithm for postfix evaluation is

```
Initialize a stack of Integer
while numbers or operators remain in the postfix string
{
        read the next number or operator, s (String)
        if s is an operator     //    + - *  or /
          {
            pop the stack twice obtaining two integers, x and y
            perform the operation ( + - *  or  /)  on x and y
            push the result
          }
          else push the number
}

Pop the final value from the stack.
```

Notice that you are reading the numbers as Strings. To do the arithmetic you have to convert to int. Integer.parseInt(s) does this

Example : The postfix equivalent of (3-4)*(6+8) is 3 4 - 6 8 + *

Here is how the algorithm works

3 4 - 6 8 + *	Push 3 
3 4 - 6 8 + *	Push 4 
3 4 - 6 8 + *	Pop 4 Pop 3 Perform the subtraction 3 – 4 = -1 Push -1 
3 4 - 6 8 + *	Push 6 
3 4 - 6 8 + *	Push 8 
3 4 - 6 8 + *	Pop 8 Pop 6 Perform the addition 6+8 = 14 Push 14 
3 4 - 6 8 + *	Pop 14 Pop -1 Perform the mult: -1 * 14 = -14 Push -14 
Input read	Pop -14 → that is the result Stack is empty 

How to convert Infix to postfix.

Here are some infix expressions with equivalent postfix. Again. notice that in the postfix expression the operators are written in the order that they are performed and without parentheses.

In the following examples, a red operator is first performed, then green, then blue

infix: (2+3)*(4+5)	postfix: 23+45+*
infix: 2+3*4+5	postfix: 234*+5+
infix: 2+3+4+5	postfix: 23+4+5+

I will use the general term token to denote a number, an operator (+ - * /) or parentheses (or).

So, the seven tokens of the expression (23 + 10) * 15 are:

(
23
+
10
)
*
15

The nine tokens of the expression 35+18+23/45+2 are

35, +, 18, +, 23, /, 45, + and 2

One other note: A token is a String not a number or a char: "23" , "+" and ")" are tokens.

To convert infix to postfix you should

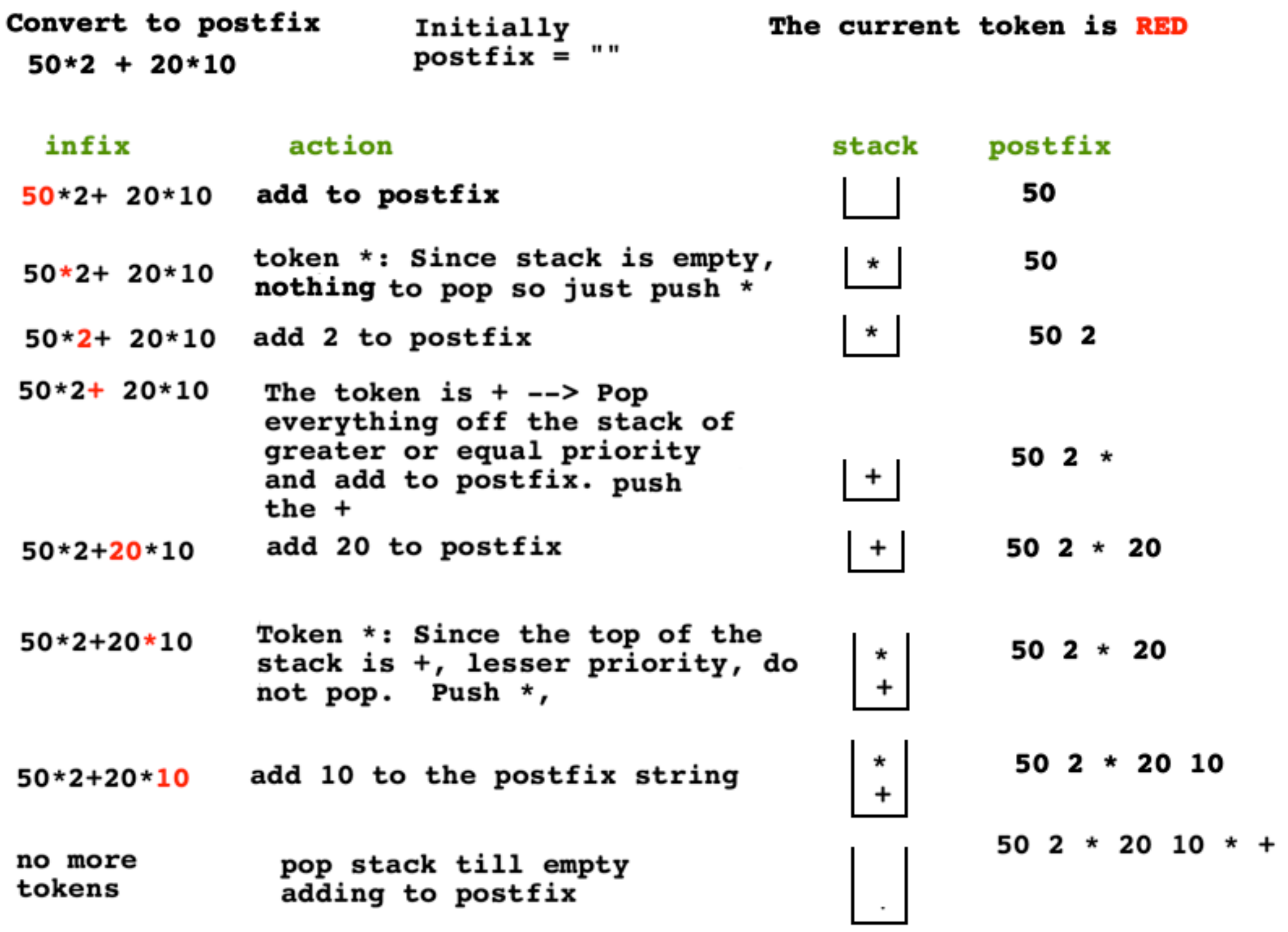
- Create a **stack of String**
- Read the infix expression left to right**, token by token**

  You will see how this is done in the homework
- Place operators ("+" "-" "*" "/") on the stack—not the numbers
- The operators * and / have greater priority than + and –
- + and – have equal priority
- * and / have equal priority
- (has lowest priority→ everything is greater than (

Here is the algorithm that converts a valid infix expression to a postfix expression
Here the operators are placed on the stack, not the numbers

```
String postfix = “”          // we will gradually build the postfix expression

While there are more tokens     // read left to right
 {
     1. get the next token in the infix string

     2.  if the token is number, append  the token (with a space at the end)
         to the postfix string  → postfix = postfix + token+ " ";

     3.  if the token is an operator  (i.e. +, -, *, or /)
          {
             while an operator of greater or equal priority is on the stack
              {
                    pop  the stack and
                    append the popped operator (with a space at the end)
                    to the postfix String

              }
              push the current token onto the stack
         }

     4.  if the token is a left parenthesis  (
             push the parenthesis onto the stack

     5. if the token is a right parenthesis )
         {
             while the top of the stack is not a matching left parenthesis (
             {
                      pop the stack
                      add the popped operator to the postfix expression
                      with a space at the end
             }
           pop the (off the stack and discard it.  Do not add it to postfix
        }

  } // end while
  Pop any remaining items on the stack and add them to postfix
  (with  a space after each).
```

Examples.
In the following examples, the "current token" is red.



Convert to postfix
50*2 + 20*10

Initially
postfix = ""

The current token is RED

infix	action	stack	postfix
50*2+ 20*10	add to postfix		50
50*2+ 20*10	token *: Since stack is empty, nothing to pop so just push *	*	50
50*2+ 20*10	add 2 to postfix	*	50 2
50*2+ 20*10	The token is + --> Pop everything off the stack of greater or equal priority and add to postfix. push the +	+	50 2 *
50*2+20*10	add 20 to postfix	+	50 2 * 20
50*2+20*10	Token *: Since the top of the stack is +, lesser priority, do not pop. Push *,	* +	50 2 * 20
50*2+20*10	add 10 to the postfix string	* +	50 2 * 20 10
no more tokens	pop stack till empty adding to postfix		50 2 * 20 10 * +

Convert to postfix
(2-3+4)* 5+6

Initially
postfix = ""

The current token is RED

Note: a left paren "(" has lowest priority

infix	action	stack	postfix
(2-3+4)* 5+6	push (	(	""
(2-3+4)*5+6	add 2 to postfix	(	2
(2-3+4)* 5+6	token -: the top of stack is not of higher priority, So no pops. push -	- (	2
(2-3+4)* 5+6	token 3:add to postfix	- (	2 3
(2-3+4)* 5+6	token +: Pop everthing from stack of greater or equal priority (-) and add to postfix. Then push +	+ (	2 3 -
(2-3+4)* 5+6	Token is 4, add to postfix	+ (	2 3 - 4 2 3 - 4
(2-3+4)* 5+6	token): pop everything and add to postfix until). pop the) but do not add to postfix		2 3 - 4 +
(2-3+4)* 5+6	token *:stack empty so nothing to pop , push *	*	2 3 - 4 +
(2-3+4)* 5+6	add 5 to postfix	*	2 3 - 4 + 5
(2-3+4)*5+6	Token:+; Pop evertyhin of greater or equal priority (just one *) and add to postfix. Then push +	+	2 3 - 4 + 5 *
(2-3+4)*5+6	add 6 to postfix	+	2 3 - 4 + 5 * 6
no more tokens	pop everything left on stack and add to postfix		2 3 - 4 + 5 * 6 +

Here is another example:

Convert to postfix
2+3-4+5-6

Initially
postfix = ""

The current token is RED

infix	action	stack	postfix
2+3-4+5-6	add "2" to posfix		2
2+3-4+5-6	token:+ Stack empty, nothibg to pop, so push +	+	2
2+3-4+5-6	add 3 to postfix	+	2 3
2+3-4+5-6	token is -: pop everything of greater or equal priority and add to postfix. So pop the + and then push the minus	-	2 3 +
2+3-4+5-6	4 is the current token Add 4 to postfix	-	2 3 + 4
2+3-4+5-6	token is +: pop everything of greater or equal priority and add to postfix. So pop the minus(-) then push the plus (+)	+	2 3 + 4 -
2+3-4+5-6	token is 5, add to postfix	+	2 3 + 4 - 5
2+3-4+5-6	token is -: pop everything of greater or equal priority and add to postfix. So pop the + and then push the minus	-	2 3 + 4 - 5 +
2+3-4+5-6	token is 6. Add to postfix	-	2 3 + 4 - 5 + 6
no more tokens	pop everythin left on the stack and add to postfix.		2 3 + 4 - 5 + 6 -