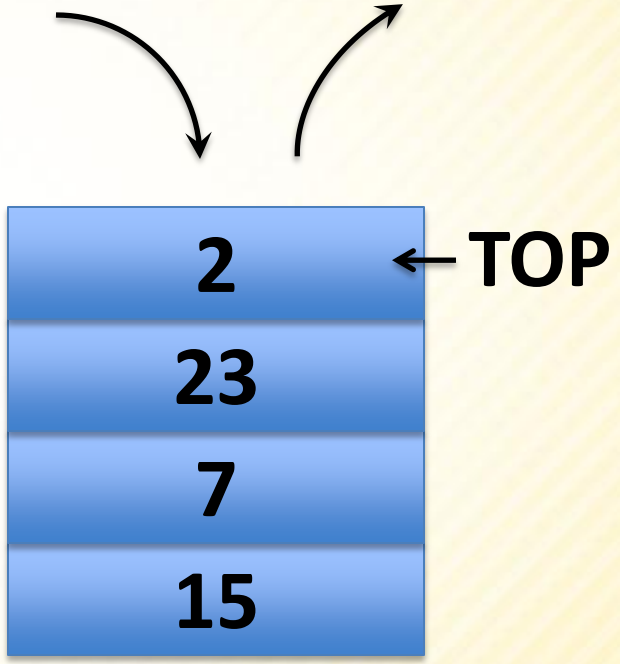
STACKS & THEIR APPLICATIONS

COP 3502

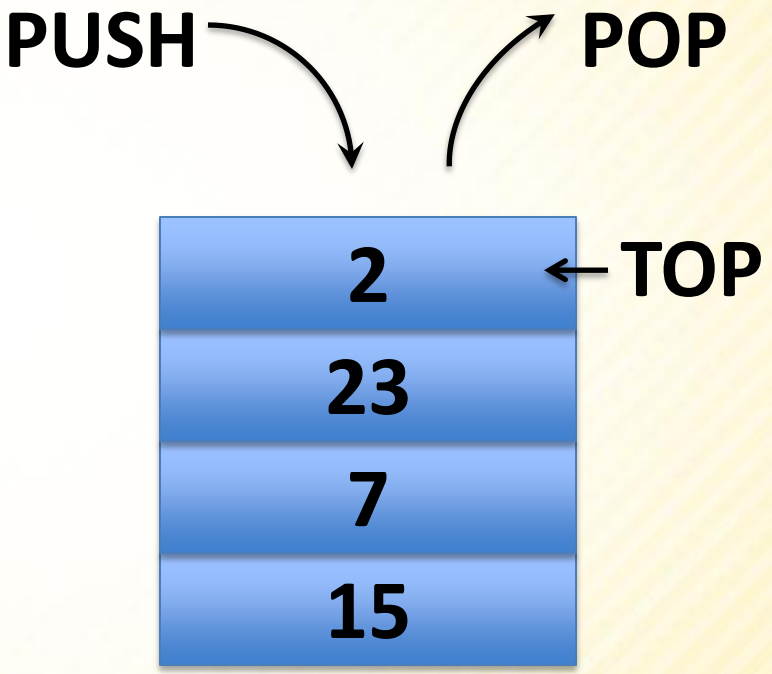
Stacks

- A stack is a data structure that stores information arranged like a stack.
  - We have seen stacks before when we used a stack to trace through recursive programs.
- The essential idea is that the last item placed into the stack is the first item removed from the stack
  - Or LIFO(Last In, First Out) for short.



Stacks

- There are two operations that modify the contents of a stack:
  - **Push** – inserts some data onto the stack.
  - **Pop** – that extracts the top-most element from the stack.



Abstract Data Type

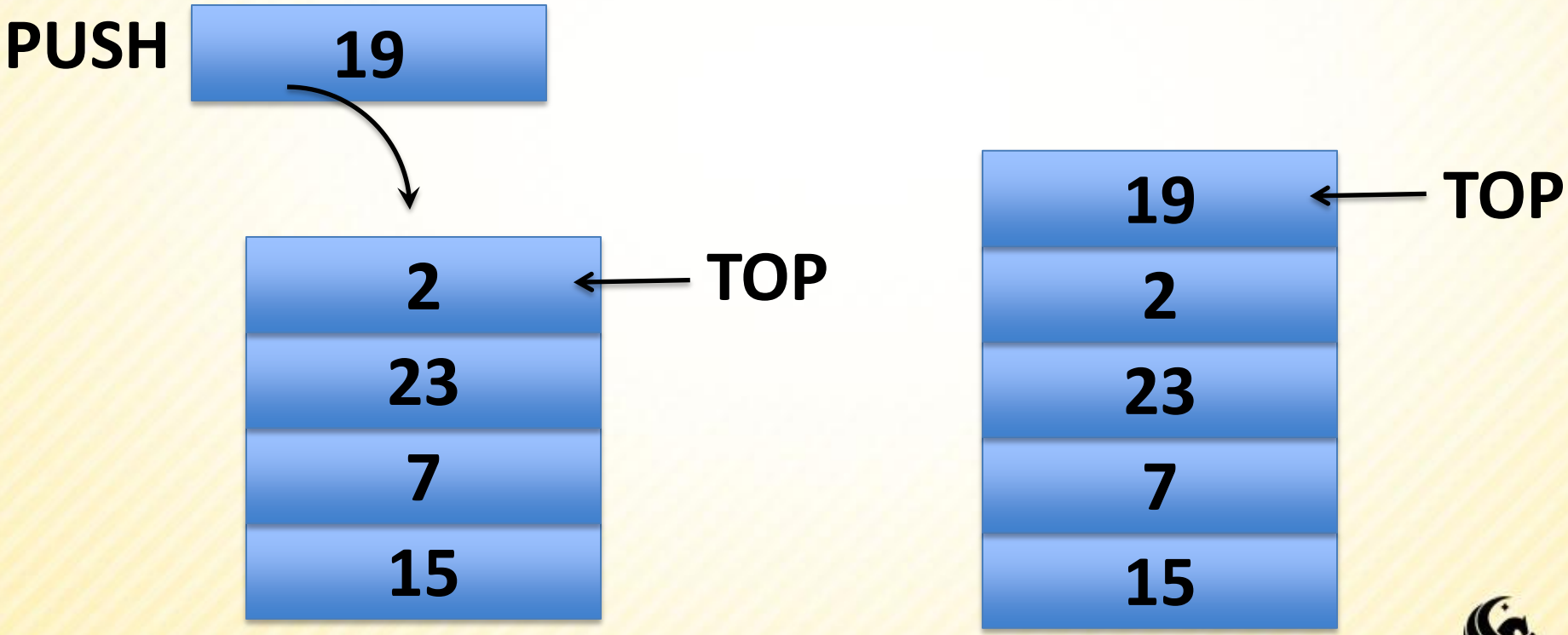
- Side Note: What is an Abstract Data Type (in C)?
  - An abstract data type is something that is not built into the language.
    - So int and double are built-in types in the C language.
  - An Abstract Data Type is something we "build"
    - and is often defined in terms of its behavior
  - Definition from Wikipedia:
    - An abstract data type is defined indirectly, only by the operations that may be performed on it (i.e. behavior).
  - So example of ADT's we've seen so far are:
    - Linked Lists, Doubly Linked Lists, Circularly Linked Lists
    - And now Stacks

Stacks

- Stacks:
  - Stacks are an Abstract Data Type
    - They are NOT built into C
  - So we must define them and their behaviors
- Stack behavior:
  - A data structure that stores information in the form of a stack.
    - Contains any number of elements of the same type.
  - Access policy:
    - The last item placed on the stack is the first item removed from the stack.

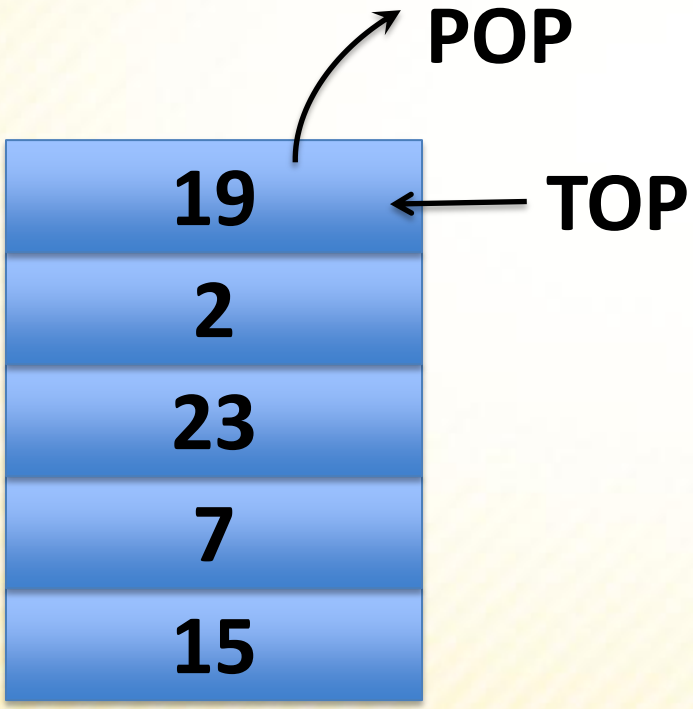
Stacks

- Stacks:
  - Basic Operations: PUSH and POP
    - PUSH – PUSH an item on the top of the stack.

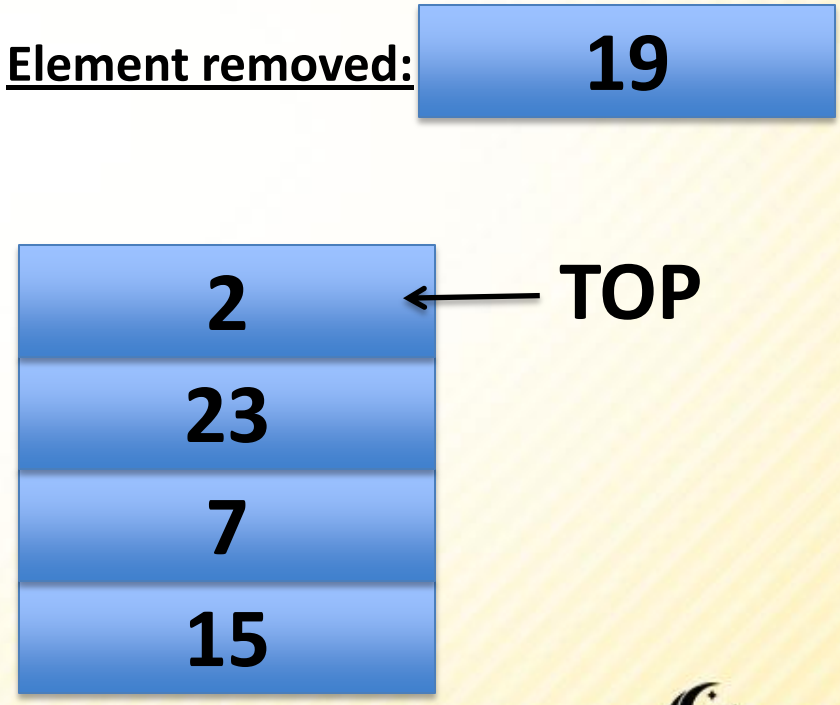


Stacks

- Stacks:
  - Basic Operations: PUSH and POP
    - POP – POP off the item in the stack and return it.



(Stack before Pop Operation)



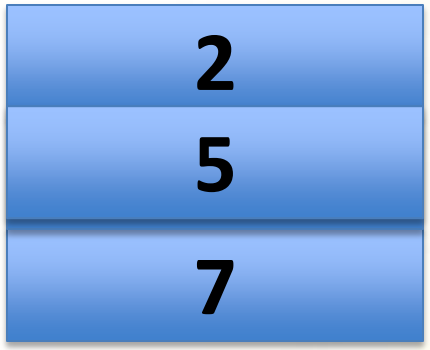
(Stack after Pop Operation)

Stacks

- Stacks:
  - Other useful operations:
    - **empty** – typically implemented as a boolean function that returns true if no items are in the stack.
    - **full** – returns true if no more items can be added to the stack.
      - In theory a stack should never be full, but any actual implementation has a limit on the # of elements it can store.
    - **top** – simply returns the value stored at the top of the stack without popping it off the stack.

Stacks

- Simple Example:
  - PUSH(7)
  - PUSH(3)
  - PUSH(2)
  - POP
  - POP
  - PUSH(5)
  - POP



Stacks

- Stack Applications:
  - Whenever we need a LIFO component to a system.
  - There are several examples outside the scope of this class you will see in CS2
    - (Depth First Search in a Graph)
  - For now, we'll go over 2 classical examples
    - Converting infix expressions to a postfix expression.
    - Evaluating a postfix expression.

Stacks

- An Infix Arithmetic Expression:
  - Consider the expression: 2 * 3 + 5 + 3 * 4
  - We know how to evaluate this expression because we usually see it in this form.
    - Multiply 2 * 3 = 6, store this in your head.
    - Add 5, now store 11 in your head.
    - Now multiply 3 * 4 = 12.
    - Retrieve 11 and add to 12.
  - This was easy because we know the rules of precedence for infix expressions.
  - But what if we didn't know these rules?

Stacks

- There are other ways of writing this expression:
  - **2 3 * 5 + 3 4 * +**
  - is the Postfix form of:
    - 2 * 3 + 5 + 3 * 4
  - And if you read from left to right, the operators are always in the correct evaluation order.

Stacks

- So there are 3 types of notations for expressions
  - Infix (A + B)
  - Postfix (A B +)
  - Prefix (+ A B)
- We're just going to worry about Infix and Postfix.
  - What does this have to do with Stacks??

Stacks

- We are going to use stacks to:
  1) Evaluate postfix expressions
  2) Convert infix expressions to a postfix expressions

Stacks

- Evaluating a Postfix Expression (A B +)
  - Consider the Postfix expression:
    - 2 3 * 5 + 3 4 * +
  - **The Rules are:**
    1) Each number gets pushed onto the stack.
    2) Whenever you get to an operator OP,
       - you **POP** off the last two values off the stack, **s1** and **s2** respectively.
       - Then you **PUSH** value **s2 OP s1** back onto the stack.
         - (If there were not two values to pop off, the expression is not in valid Postfix notation)
    3) When you are done, you should have a single value left on the stack that the expression evaluates to.

2 3 * 5 + 3 4 * +

The Rules are:

1) **Each number gets PUSHed onto the stack.**
2) Whenever you get to an operator **OP**,
   - you **POP** off the last two values off the stack, **s1** and **s2** respectively.
   - Then you **PUSH** value **s2 OP s1** back onto the stack.
     - (If there aren't 2 values to pop, the expression is not valid Postfix)
3) When you are done, you should have a single value left on the stack that the expression evaluates to.

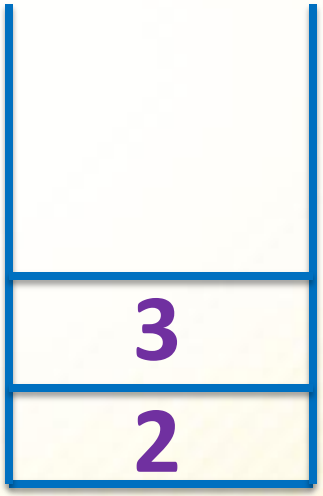
PUSH(2)

2 3 * 5 + 3 4 * +

The Rules are:

1) **Each number gets PUSHed onto the stack.**
2) Whenever you get to an operator OP,
   - you POP off the last two values off the stack, s1 and s2 respectively.
   - Then you PUSH value s2 OP s1 back onto the stack.
     - (If there aren't 2 values to pop, the expression is not valid Postfix)
3) When you are done, you should have a single value left on the stack that the expression evaluates to.

PUSH(3)

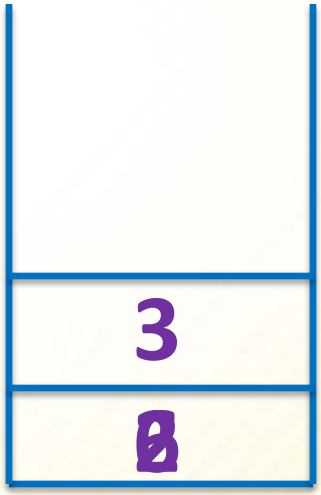


2 3 * 5 + 3 4 * +

The Rules are:

1) Each number gets pushed onto the stack.
2) **Whenever you get to an operator OP,**
   - **you POP off the last two values off the stack, s1 and s2 respectively.**
   - **Then you PUSH value s2 OP s1 back onto the stack.**
     - (If there aren't 2 values to pop, the expression is not valid Postfix)
3) When you are done, you should have a single value left on the stack that the expression evaluates to.

POP s1 = 3
POP s2 = 2
PUSH(2 * 3 = 6)

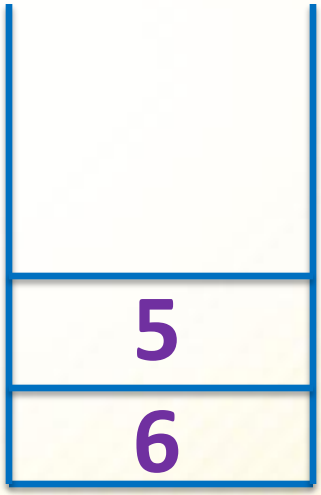


2 3 * 5 + 3 4 * +

The Rules are:

1) **Each number gets pushed onto the stack.**
2) Whenever you get to an operator OP,
   - you POP off the last two values off the stack, s1 and s2 respectively.
   - Then you PUSH value s2 OP s1 back onto the stack.
     - (If there aren't 2 values to pop, the expression is not valid Postfix)
3) When you are done, you should have a single value left on the stack that the expression evaluates to.

PUSH(5)

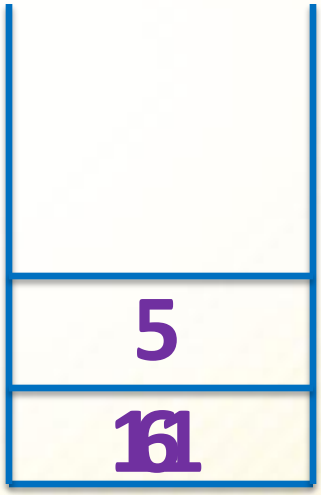


2 3 * 5 + 3 4 * +

The Rules are:

1) Each number gets pushed onto the stack.
2) **Whenever you get to an operator OP,**
   - **you POP off the last two values off the stack, s1 and s2 respectively.**
   - **Then you PUSH value s2 OP s1 back onto the stack.**
     - **(If there aren't 2 values to pop, the expression is not valid Postfix)**
3) When you are done, you should have a single value left on the stack that the expression evaluates to.

POP s1 = 5
POP s2 = 6
PUSH(6 + 5 = 11)

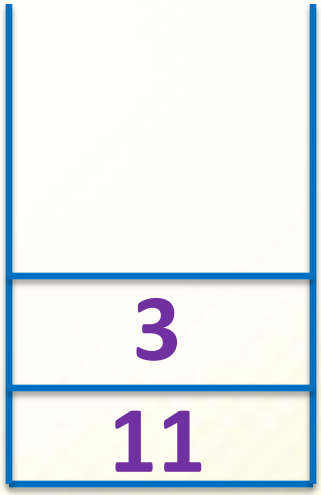


2 3 * 5 + 3 4 * +

The Rules are:

1) **Each number gets pushed onto the stack.**
2) Whenever you get to an operator OP,
   - you POP off the last two values off the stack, s1 and s2 respectively.
   - Then you PUSH value s2 OP s1 back onto the stack.
     - (If there aren't 2 values to pop, the expression is not valid Postfix)
3) When you are done, you should have a single value left on the stack that the expression evaluates to.

PUSH(3)

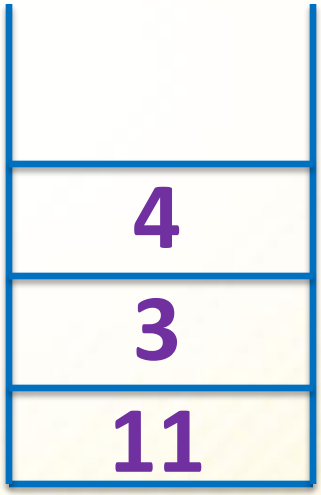


2 3 * 5 + 3 4 * +

The Rules are:

1) **Each number gets pushed onto the stack.**
2) Whenever you get to an operator OP,
   - you POP off the last two values off the stack, s1 and s2 respectively.
   - Then you PUSH value s2 OP s1 back onto the stack.
     - (If there aren't 2 values to pop, the expression is not valid Postfix)
3) When you are done, you should have a single value left on the stack that the expression evaluates to.

PUSH(4)

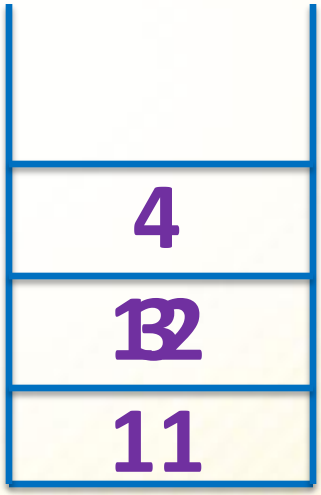


2 3 * 5 + 3 4 * +

The Rules are:

1) Each number gets pushed onto the stack.
2) **Whenever you get to an operator OP,**
   - **you POP off the last two values off the stack, s1 and s2 respectively.**
   - **Then you PUSH value s2 OP s1 back onto the stack.**
     - **(If there aren't 2 values to pop, the expression is not valid Postfix)**
3) When you are done, you should have a single value left on the stack that the expression evaluates to.

POP s1 = 4
POP s2 = 3
PUSH(3 * 4 = 12)

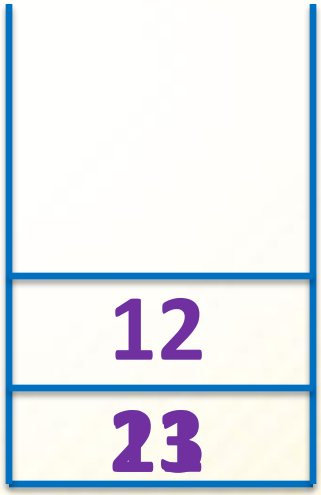


2 3 * 5 + 3 4 * +

The Rules are:

1) Each number gets pushed onto the stack.
2) Whenever you get to an operator OP,
   - you POP off the last two values off the stack, s1 and s2 respectively.
   - Then you PUSH value s2 OP s1 back onto the stack.
     - (If there aren't 2 values to pop, the expression is not valid Postfix)
3) When you are done, you should have a single value left on the stack that the expression evaluates to.

POP s1 = 12
POP s2 = 11
PUSH(11 + 12 = 23)



Stacks

- So now we know how to evaluate a Postfix expression using a stack.
- NOW, we're going to convert an Infix expression to Postfix using a stack.
  - Before we get started, we need to know operator precedence:

1) (In the expression, left paren has highest priority
2) * / Going from left to right whichever comes 1st
3) + - Going from left to right whichever comes 1st
4) (In the stack, the left parentheses has lowest priority

Stacks

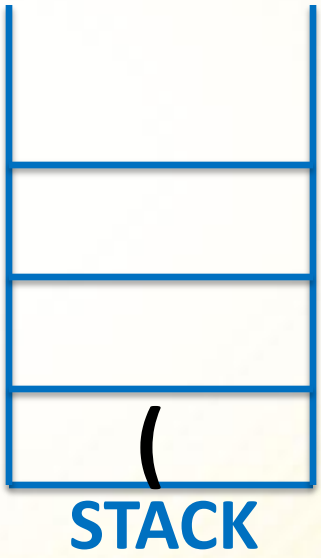
- Converting an Infix expression to a Postfix expression.
- Rules:
  1) For all operands, automatically place them in the output expression.
  2) For an operator (+, -, *, /, or a parenthesis)
     - **IF the operator is an open parenthesis**, push it onto the stack.
     - **ELSE IF the operator is an arithmetic one**, then do this:
       - Continue popping off items off the stack and placing them in the output expression until you hit an operator with lower precedence than the current operator or until you hit an open parenthesis.
       - At this point, push the current operator onto the stack.
     - **ELSE** Pop off all operators off the stack one by one, placing them in the output expression until you hit thefirst(matching) open parenthesis. When this occurs, pop off the open parenthesis and discard both ()s.

(7 * (6 + 3) + (2 – 3) + 1)

- **Rules:**

1) For all operands, automatically put in output expression.
2) **For an operator +, -, *, /, or (,)**
   - **IF the operator is an open paren, PUSH it.**
   - **ELSE IF the operator is an arithmetic one**, then do this:
     - Continue **POP**ing items and placing them in the output until you hit an **OP** with **<** precedence than the curr **OP** or until you hit an open paren.
     - At this point, **PUSH** the curr **OP**.
   - **ELSE POP** off all **OPs** off the stack 1 by 1, placing them in the output expression until you hit the 1st) open paren. When this occurs, **POP** off the open paren and discard both ()s.

PUSH (

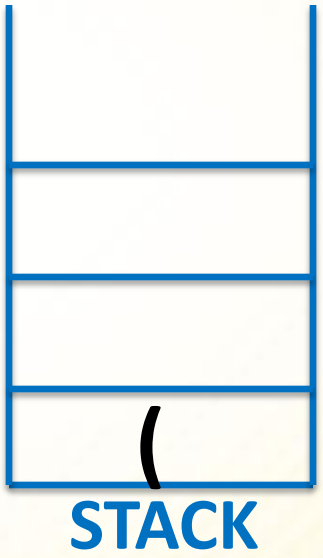


Output:

(7 * (6 + 3) + (2 – 3) + 1)

- **Rules:**

1) **For all operands, automatically put in output expression.**
2) For an operator +, -, *, /, or (,)
   - **IF the operator is an open paren**, PUSH it.
   - **ELSE IF the operator is an arithmetic one**, then do this:
     - Continue **POP**ing items and placing them in the output until you hit an **OP** with **<** precedence than the curr **OP** or until you hit an open paren.
     - At this point, **PUSH** the curr **OP**.
   - **ELSE POP** off all **OPs** off the stack 1 by 1, placing them in the output expression until you hit the 1st) open paren. When this occurs, **POP** off the open paren and discard both ()s.



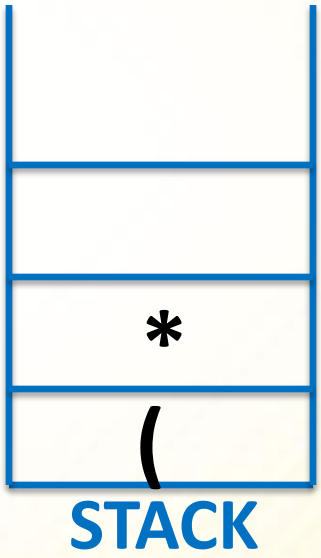
Output: 7

(7 * (6 + 3) + (2 – 3) + 1)

- **Rules:**

1) For all operands, automatically put in output expression.
2) **For an operator +, -, *, /, or (,)**
   - **IF the operator is an open paren**, PUSH it.
   - **ELSE IF the operator is an arithmetic one**, then do this:
     - Continue **POP**ing items and placing them in the output until you hit an **OP** with **<** precedence than the curr **OP** or until you hit an open paren.
     - At this point, **PUSH** the curr **OP**.
   - **ELSE POP** off all **OPs** off the stack 1 by 1, placing them in the output expression until you hit the 1st) open paren. When this occurs, **POP** off the open paren and discard both ()s.

PUSH *



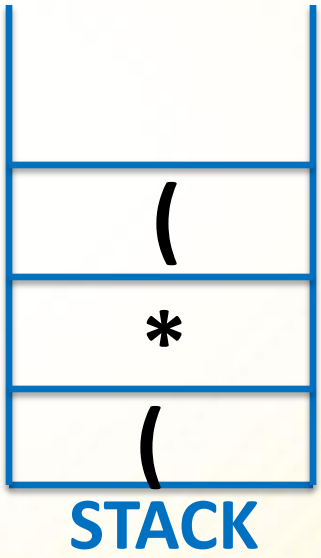
Output: 7

(7 * (6 + 3) + (2 – 3) + 1)

- **Rules:**

1) For all operands, automatically put in output expression.
2) **For an operator +, -, *, /, or (,)**
   - **IF the operator is an open paren, PUSH it.**
   - **ELSE IF the operator is an arithmetic one**, then do this:
     - Continue **POP**ing items and placing them in the output until you hit an **OP** with **<** precedence than the curr **OP** or until you hit an open paren.
     - At this point, **PUSH** the curr **OP**.
   - **ELSE POP** off all **OPs** off the stack 1 by 1, placing them in the output expression until you hit the 1st) open paren. When this occurs, **POP** off the open paren and discard both ()s.

PUSH (

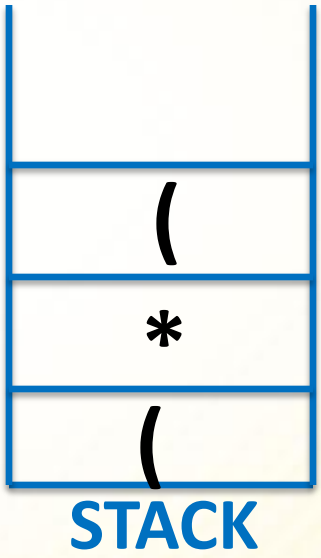


Output: 7

(7 * (6 + 3) + (2 – 3) + 1)

- **Rules:**

1) **For all operands, automatically put in output expression.**
2) For an operator +, -, *, /, or (,)
   - **IF the operator is an open paren**, PUSH it.
   - **ELSE IF the operator is an arithmetic one**, then do this:
     - Continue **POP**ing items and placing them in the output until you hit an **OP** with **<** precedence than the curr **OP** or until you hit an open paren.
     - At this point, **PUSH** the curr **OP**.
   - **ELSE POP** off all **OPs** off the stack 1 by 1, placing them in the output expression until you hit the 1st) open paren. When this occurs, **POP** off the open paren and discard both ()s.



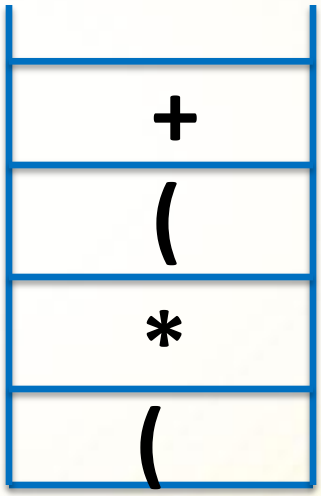
Output: 7 6

(7 * (6 + 3) + (2 – 3) + 1)

- **Rules:**

1) For all operands, automatically put in output expression.
2) **For an operator +, -, *, /, or (,)**
   - **IF the operator is an open paren**, PUSH it.
   - **ELSE IF the operator is an arithmetic one**, then do this:
     - Continue **POP**ing items and placing them in the output until you hit an **OP** with **<** precedence than the curr **OP** or until you hit an open paren.
     - At this point, **PUSH** the curr **OP**.
   - **ELSE POP** off all **OPs** off the stack 1 by 1, placing them in the output expression until you hit the 1st) open paren. When this occurs, **POP** off the open paren and discard both ()s.

PUSH +

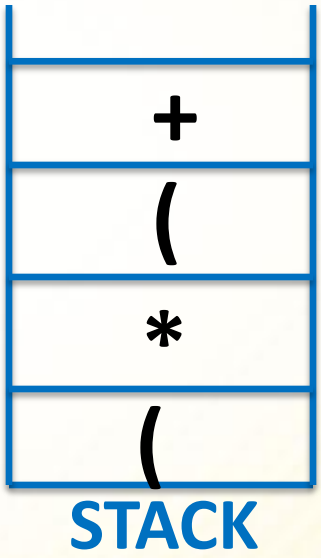


Output: 7 6

(7 * (6 + 3) + (2 – 3) + 1)

- **Rules:**

1) **For all operands, automatically put in output expression.**
2) For an operator +, -, *, /, or (,)
   - **IF the operator is an open paren**, PUSH it.
   - **ELSE IF the operator is an arithmetic one**, then do this:
     - Continue **POP**ing items and placing them in the output until you hit an **OP** with **<** precedence than the curr **OP** or until you hit an open paren.
     - At this point, **PUSH** the curr **OP**.
   - **ELSE POP** off all **OPs** off the stack 1 by 1, placing them in the output expression until you hit the 1st) open paren. When this occurs, **POP** off the open paren and discard both ()s.



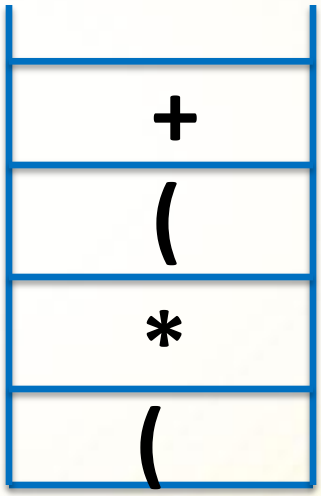
Output: 7 6 3

(7 * (6 + 3) + (2 – 3) + 1)

- **Rules:**

1) For all operands, automatically put in output expression.
2) **For an operator +, -, *, /, or (,)**
   - **IF the operator is an open paren**, PUSH it.
   - **ELSE IF the operator is an arithmetic one**, then do this:
     - Continue **POP**ing items and placing them in the output until you hit an **OP** with **<** precedence than the curr **OP** or until you hit an open paren.
     - At this point, **PUSH** the curr **OP**.
   - **ELSE POP** off all **OPs** off the stack 1 by 1, placing them in the output expression until you hit the 1st) open paren. When this occurs, **POP** off the open paren and discard both ()s.

POP +
POP (



STACK

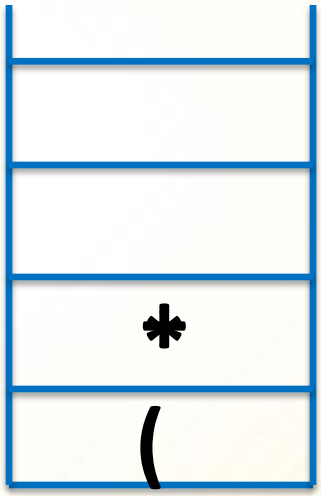
Output: 7 6 3 +

(7 * (6 + 3) + (2 – 3) + 1)

- **Rules:**

1) For all operands, automatically put in output expression.
2) **For an operator +, -, *, /, or (,)**
   - **IF the operator is an open paren**, PUSH it.
   - **ELSE IF the operator is an arithmetic one**, then do this:
     - Continue **POP**ing items and placing them in the output until you hit an **OP** with **<** precedence than the curr **OP** or until you hit an open paren.
     - At this point, **PUSH** the curr **OP**.
   - **ELSE POP** off all **OPs** off the stack 1 by 1, placing them in the output expression until you hit the 1st) open paren. When this occurs, **POP** off the open paren and discard both ()s.

POP *
PUSH +



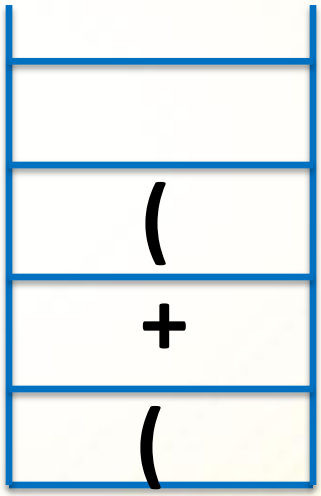
Output: 7 6 3 + *

(7 * (6 + 3) + (2 – 3) + 1)

- **Rules:**

1) For all operands, automatically put in output expression.
2) **For an operator +, -, *, /, or (,)**
   - **IF the operator is an open paren, PUSH it.**
   - **ELSE IF the operator is an arithmetic one**, then do this:
     - Continue **POP**ing items and placing them in the output until you hit an **OP** with **<** precedence than the curr **OP** or until you hit an open paren.
     - At this point, **PUSH** the curr **OP**.
   - **ELSE POP** off all **OPs** off the stack 1 by 1, placing them in the output expression until you hit the 1st) open paren. When this occurs, **POP** off the open paren and discard both ()s.

PUSH (



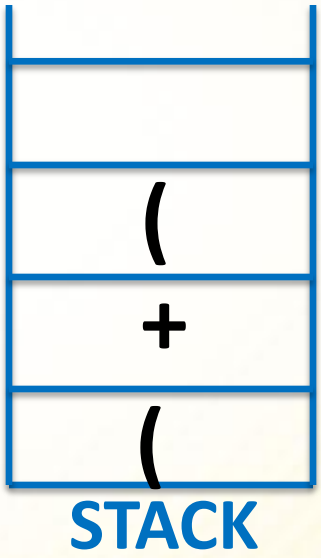
STACK

Output: 7 6 3 + *

(7 * (6 + 3) + (2 – 3) + 1)

- **Rules:**

1) **For all operands, automatically put in output expression.**
2) For an operator +, -, *, /, or (,)
   - **IF the operator is an open paren**, PUSH it.
   - **ELSE IF the operator is an arithmetic one**, then do this:
     - Continue **POP**ing items and placing them in the output until you hit an **OP** with **<** precedence than the curr **OP** or until you hit an open paren.
     - At this point, **PUSH** the curr **OP**.
   - **ELSE POP** off all **OPs** off the stack 1 by 1, placing them in the output expression until you hit the 1st) open paren. When this occurs, **POP** off the open paren and discard both ()s.



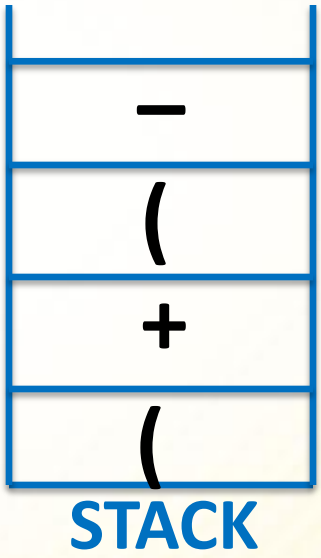
Output: 7 6 3 + * 2

(7 * (6 + 3) + (2 – 3) + 1)

- **Rules:**

1) For all operands, automatically put in output expression.
2) For an operator +, -, *, /, or (,)
   - **IF the operator is an open paren**, PUSH it.
   - **ELSE IF the operator is an arithmetic one**, then do this:
     - Continue **POP**ing items and placing them in the output until you hit an **OP** with **<** precedence than the curr **OP** or until you hit an open paren.
     - At this point, **PUSH** the curr **OP**.
   - **ELSE POP** off all **OPs** off the stack 1 by 1, placing them in the output expression until you hit the 1st) open paren. When this occurs, **POP** off the open paren and discard both ()s.

PUSH –

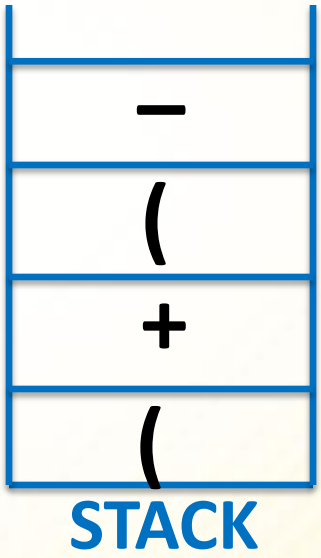


Output: 7 6 3 + * 2

(7 * (6 + 3) + (2 – 3) + 1)

- **Rules:**

1) For all operands, automatically put in output expression.
2) For an operator +, -, *, /, or (,)
   - **IF the operator is an open paren**, PUSH it.
   - **ELSE IF the operator is an arithmetic one**, then do this:
     - Continue **POP**ing items and placing them in the output until you hit an **OP** with **<** precedence than the curr **OP** or until you hit an open paren.
     - At this point, **PUSH** the curr **OP**.
   - **ELSE POP** off all **OPs** off the stack 1 by 1, placing them in the output expression until you hit the 1st) open paren. When this occurs, **POP** off the open paren and discard both ()s.



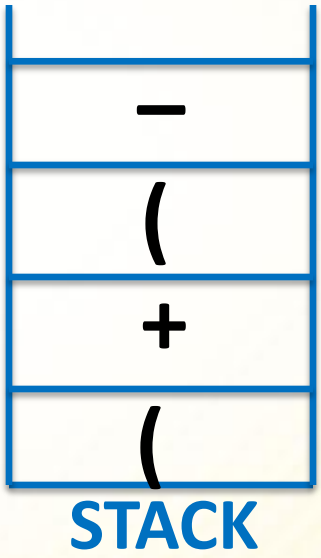
Output: 7 6 3 + * 2 3

(7 * (6 + 3) + (2 – 3) + 1)

- **Rules:**

1) For all operands, automatically put in output expression.
2) For an operator +, -, *, /, or (,)
   - **IF the operator is an open paren**, PUSH it.
   - **ELSE IF the operator is an arithmetic one**, then do this:
     - Continue **POP**ing items and placing them in the output until you hit an **OP** with **<** precedence than the curr **OP** or until you hit an open paren.
     - At this point, **PUSH** the curr **OP**.
   - **ELSE POP** off all **OPs** off the stack 1 by 1, placing them in the output expression until you hit the 1st) open paren. When this occurs, **POP** off the open paren and discard both ()s.

POP –
POP (



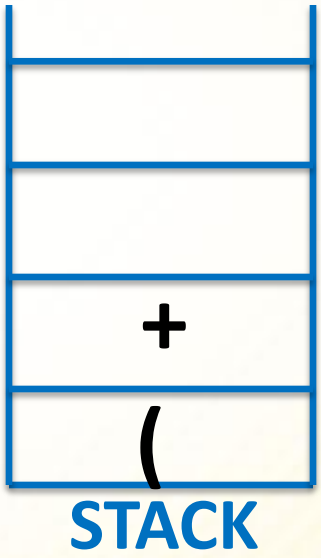
Output: 7 6 3 + * 2 3 –

(7 * (6 + 3) + (2 – 3) + 1)

- **Rules:**

1) For all operands, automatically put in output expression.
2) For an operator +, -, *, /, or (,)
   - **IF the operator is an open paren**, PUSH it.
   - **ELSE IF the operator is an arithmetic one**, then do this:
     - Continue **POP**ing items and placing them in the output until you hit an **OP** with **<** precedence than the curr **OP** or until you hit an open paren.
     - At this point, **PUSH** the curr **OP**.
   - **ELSE POP** off all **OPs** off the stack 1 by 1, placing them in the output expression until you hit the 1st) open paren. When this occurs, **POP** off the open paren and discard both ()s.

POP +
PUSH +

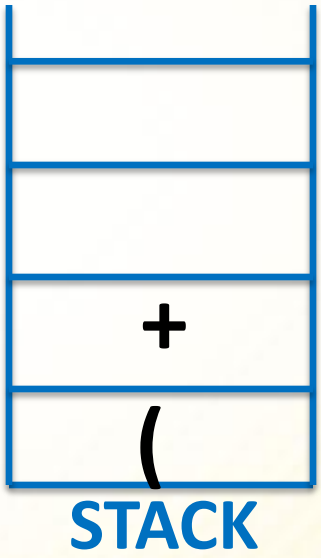


Output: 7 6 3 + * 2 3 – +

(7 * (6 + 3) + (2 – 3) + 1)

- **Rules:**

1) For all operands, automatically put in output expression.
2) For an operator +, -, *, /, or (,)
   - **IF the operator is an open paren**, PUSH it.
   - **ELSE IF the operator is an arithmetic one**, then do this:
     - Continue **POP**ing items and placing them in the output until you hit an **OP** with **<** precedence than the curr **OP** or until you hit an open paren.
     - At this point, **PUSH** the curr **OP**.
   - **ELSE POP** off all **OPs** off the stack 1 by 1, placing them in the output expression until you hit the 1st) open paren. When this occurs, **POP** off the open paren and discard both ()s.



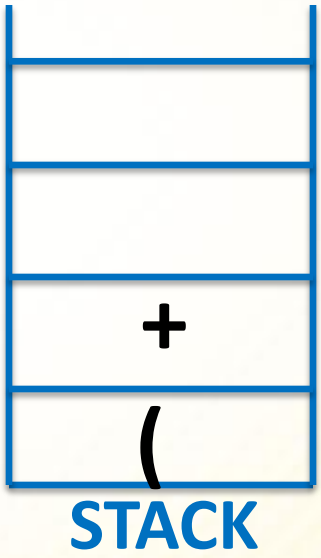
Output: 7 6 3 + * 2 3 – + 1

(7 * (6 + 3) + (2 – 3) + 1)

- **Rules:**

1) For all operands, automatically put in output expression.
2) For an operator +, -, *, /, or (,)
   - **IF the operator is an open paren**, PUSH it.
   - **ELSE IF the operator is an arithmetic one**, then do this:
     - Continue **POP**ing items and placing them in the output until you hit an **OP** with **<** precedence than the curr **OP** or until you hit an open paren.
     - At this point, **PUSH** the curr **OP**.
   - **ELSE POP** off all **OPs** off the stack 1 by 1, placing them in the output expression until you hit the 1st) open paren. When this occurs, **POP** off the open paren and discard both ()s.

POP +
POP (



Output: 7 6 3 + * 2 3 – + 1 +

Converting Infix to Postfix

- Given the Infix expression:
  - **(7 * (6 + 3) + (2 – 3) + 1)**
- Our final Postfix expression was this:
  - **7 6 3 + * 2 3 – + 1 +**

- We can check if this is correct by evaluating the Postfix expression like we did before.
  - Let's do that on the board…
  - We should get **(7 * (6 + 3) + (2 – 3) + 1) =**
    - **62**

Infix to Postfix

- **Rules:**

1) For all operands, automatically put in output expression.
2) For an operator +, -, *, /, or (,)
   - **IF the operator is an open paren**, PUSH it.
   - **ELSE IF the operator is an arithmetic one**, then do this:
     - Continue **POP**ing items and placing them in the output until you hit an **OP** with **<** precedence than the curr **OP** or until you hit an open paren.
     - At this point, **PUSH** the curr **OP**.
   - **ELSE POP** off all **OPs** off the stack 1 by 1, placing them in the output expression until you hit the 1st) open paren. When this occurs, **POP** off the open paren and discard both ()s.

- **Try this example:**
  - **5 * 3 + 2 + 6 * 4**
  - **Don't forget to check your answer!**