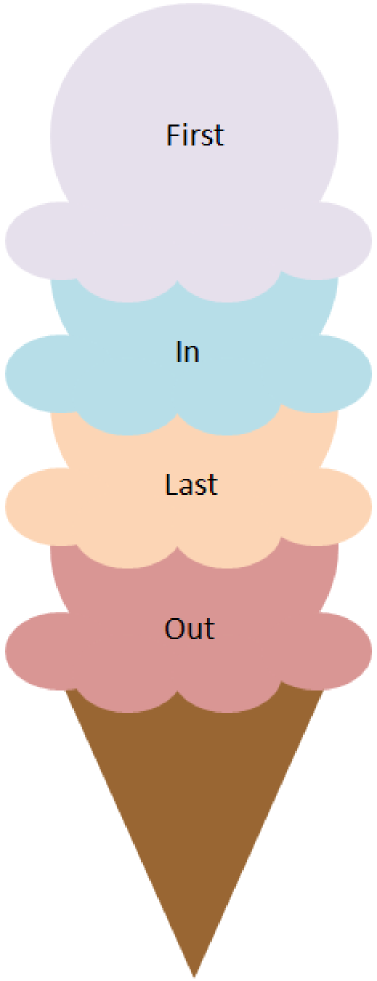
Stack



Stack ADT

- A **stack** is a data structure in which all access is restricted to the most recently inserted element.
  - Stack has only one end.
- Insertions and deletions follow last-in first-out (LIFO) scheme (principle).
  - It means the element added last will be removed first.

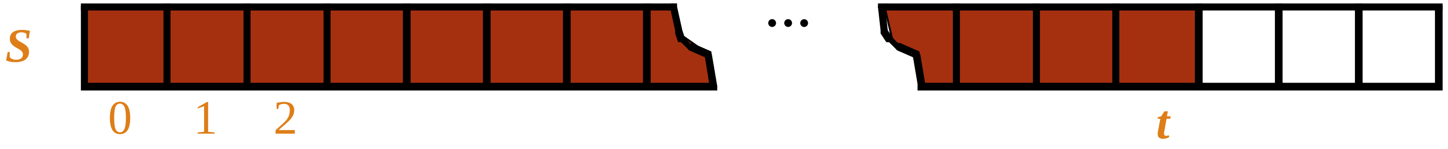
- **Main operations:**
  - **push(object):** insert element
  - **object pop():** remove and returns last element
- **Auxiliary operations:**
  - object top(): returns last element without removing it.
  - integer size(): returns number of elements stored.
  - boolean isEmpty(): returns whether no elements are stored.

Applications of Stacks

- Direct
  - Page visited history in a web browser.
  - Undo sequence in a text editors.
  - Chain of method calls in C++ runtime environment.
  - Stack is used to evaluate prefix, postfix and infix expressions.
  - An expression can be represented in prefix, postfix or infix notation. Stack can be used to convert one form of expression to another.
- Indirect
  - Auxiliary data structure for algorithms.
  - Component of other data structures.

Array-based Stack

- Add elements in an array ***S*** of capacity(size) ***N.***
- A variable *top* keeps track of the index of the top element.
- Size is *top*+1



Push and Pop Algorithms

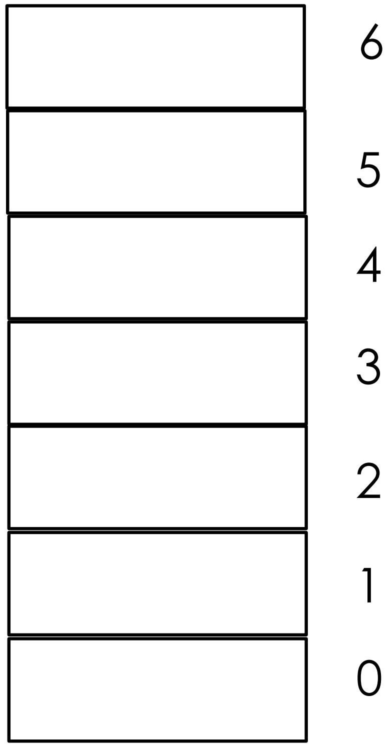
```
Algorithm push(Element):
    if top= N-1 then
        throw “Full Stack Exception”
    else
        top ← top + 1
        S[top] ← Element
```

Run time: O(1)

```
Algorithm pop():
    if isEmpty() then
        throw “Empty Stack Exception”
    else
        top ← top – 1
        return S[top + 1]
```

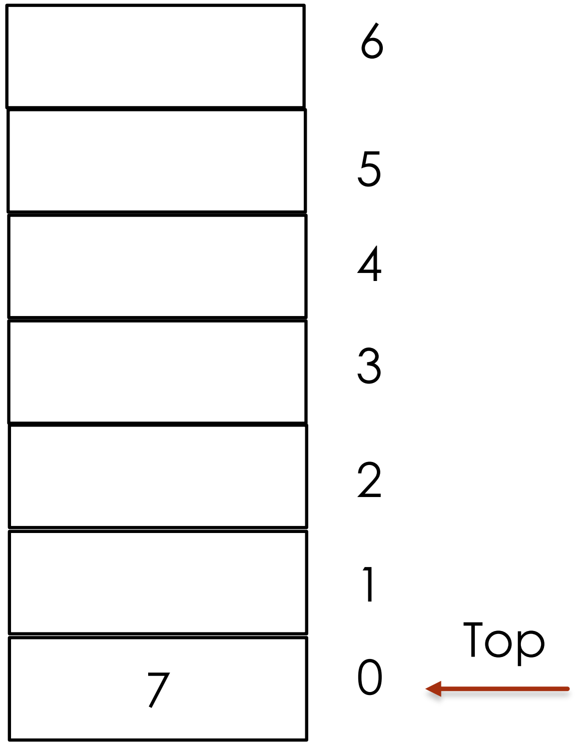
Run Time: O(1)

Stack Operations - Example



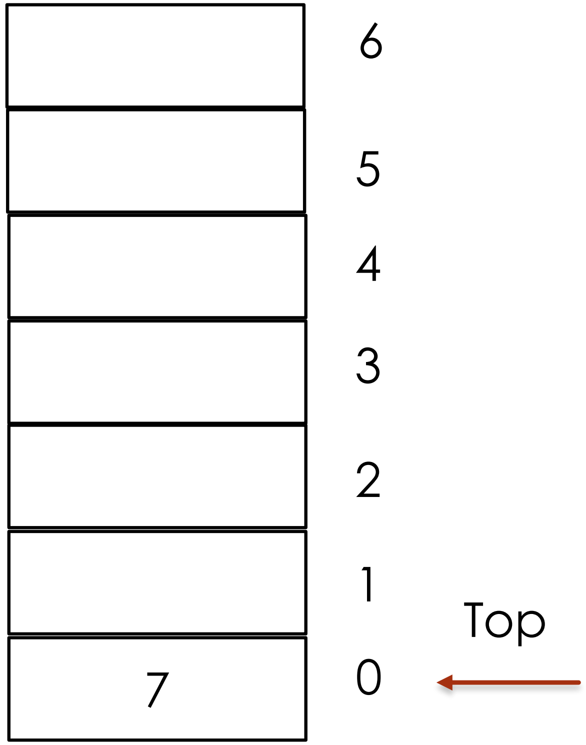
Stack S, N=7, Top=-1

```
Push (7)
{
      if top==N-1 Then
            "Overflow"
       else
            Top=Top+1
            S[top]=7
}
```



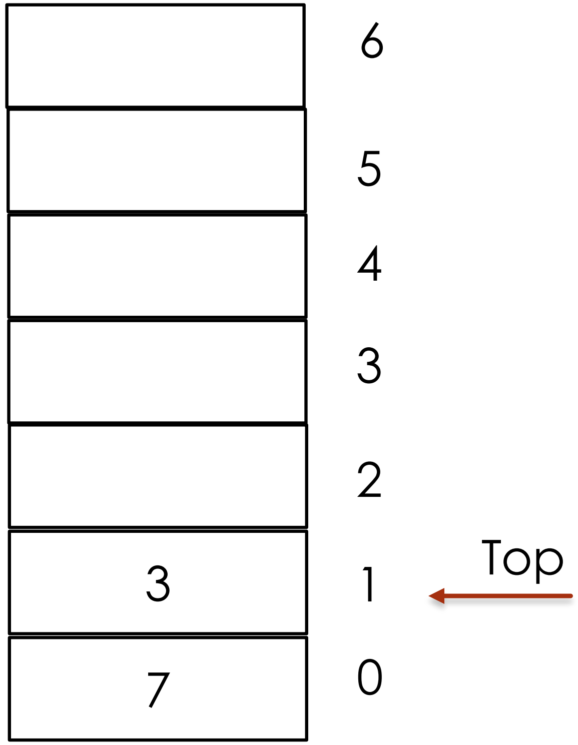
Stack S, N=7

Stack Operations - Example



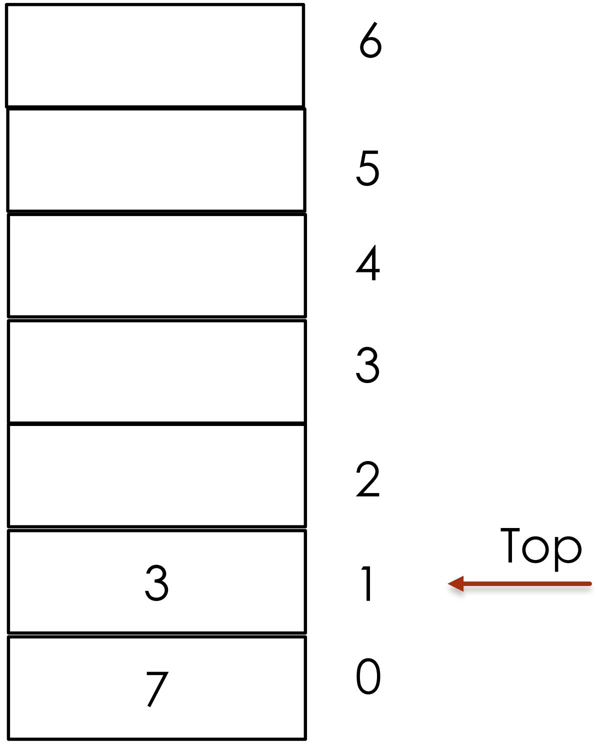
Stack S, N=7

```
Push (3)
{
      if top==N-1 Then
            "Overflow"
       else
            Top=Top+1
            S[top]=3
}
```



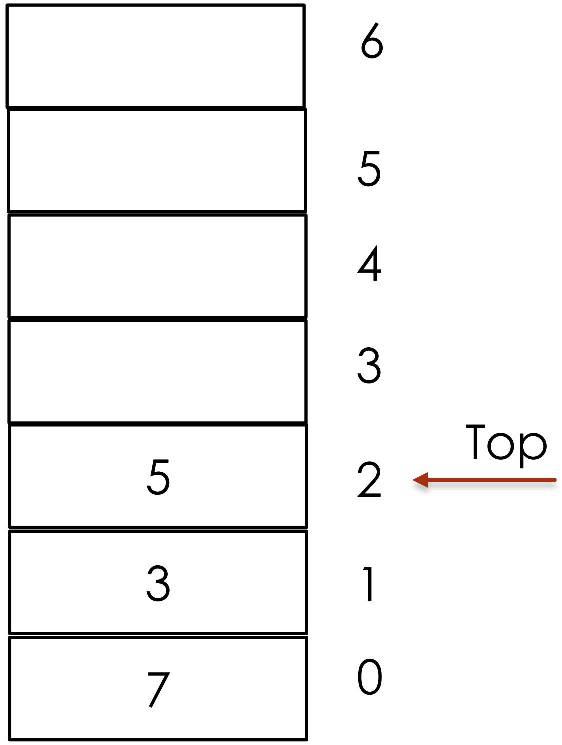
Stack S, N=7

Stack Operations - Example



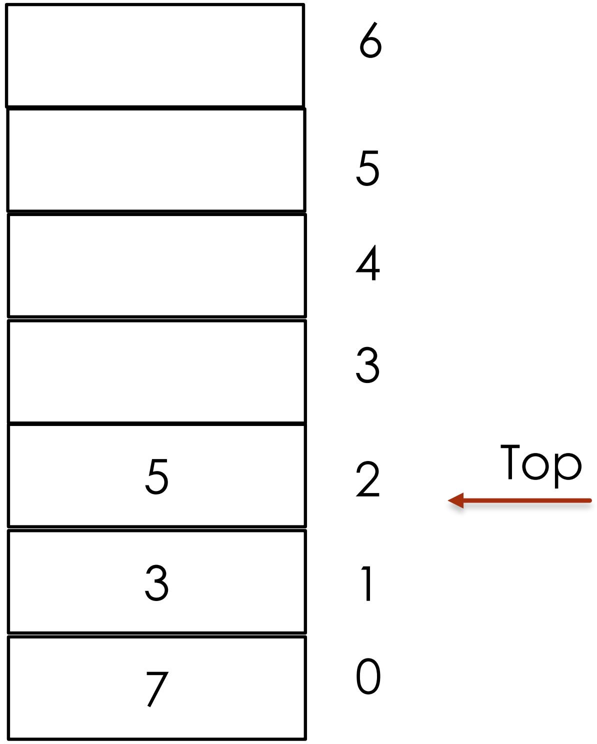
Stack S, N=7

```
Push (5)
{
    if top==N-1 Then
        "Overflow"
    else
        Top=Top+1
        S[top]=5
}
```



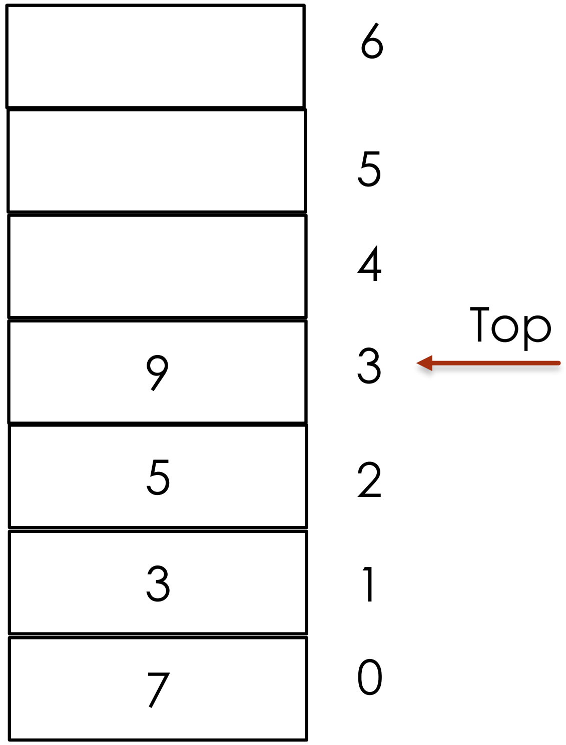
Stack S, N=7

Stack Operations - Example



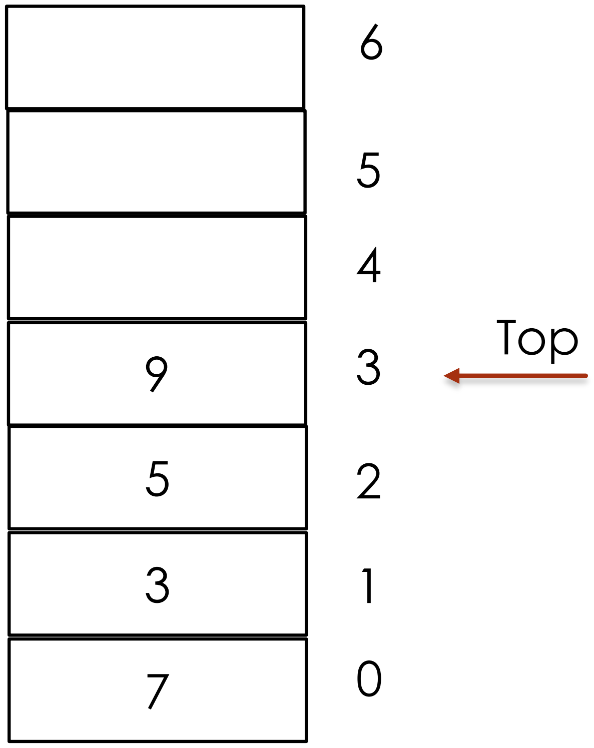
Stack S, N=7

```
Push (9)
{
     if top==N-1 Then
          "Overflow"
      else
          Top=Top+1
          S[top]=9
}
```



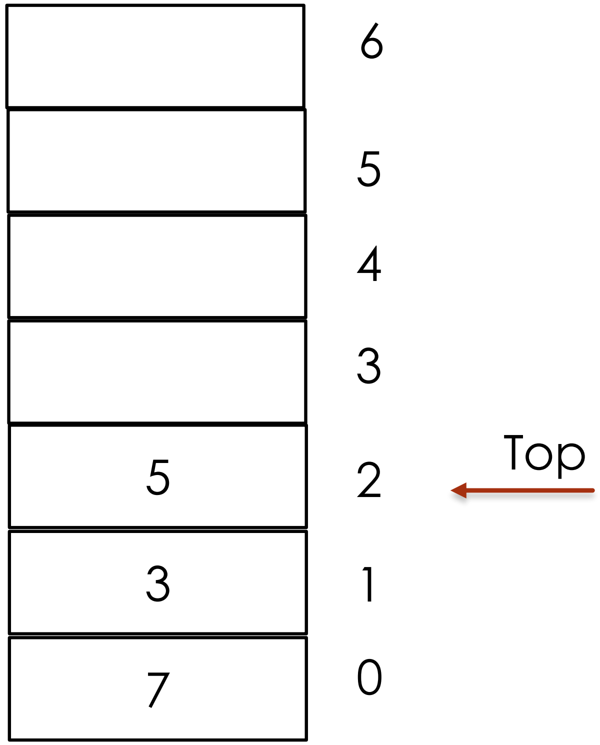
Stack S, N=7

Stack Operations - Example



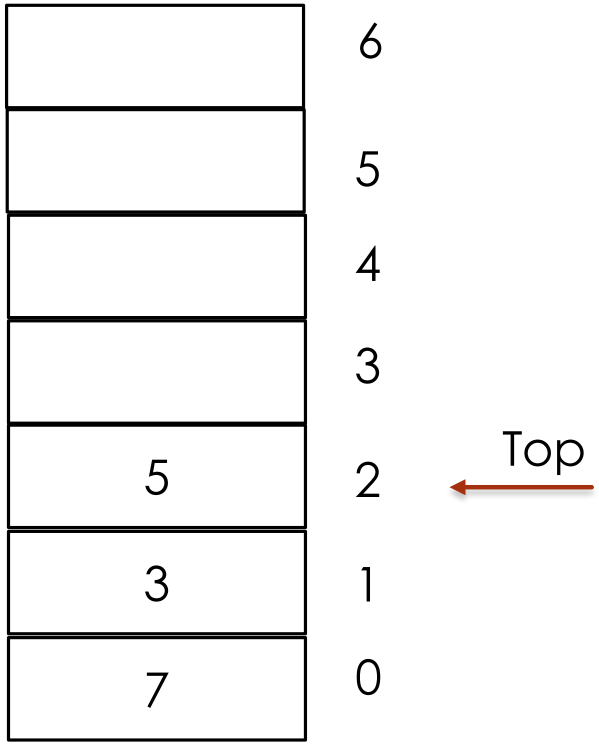
Stack S, N=7

```
Pop ()
{
    if isEmpty() then
        "Underflow"
    else
        Top=Top-1
        return S[top+1]
}
```



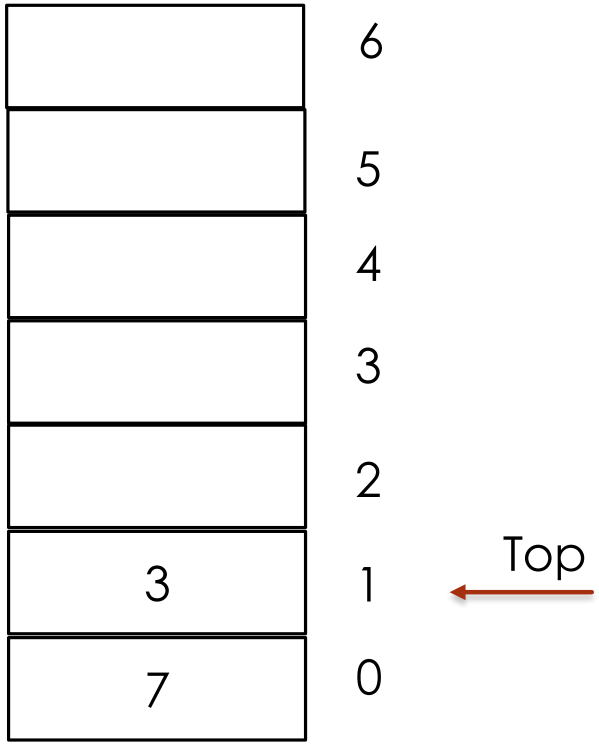
Stack S, N=7

Stack Operations - Example



Stack S, N=6

```
Pop ()
{
    if isEmpty() then
        "Underflow"
    else
        Top--
        return S[top+1]
}
```



Stack S, N=6

Arithmetic Expression

Arithmetic Expressions

- An **arithmetic expression** is an expression that results in a numeric value.
- It is a correct combination of numbers, operators, parenthesis, and variables.
- Expressions are usually represented in what is known as **Infix notation**, in which each operator is written between two operands
- **Example: A + B**
  - **A** and **B** are called ***Operands***
  - **+** is called the **operator**

Arithmetic Expressions

- Infix form
  - **Need precedence rules**
  - **May use parentheses.**
- Example: 2+4*3 What is the result?
  - Apply precedence rules (* has higher precedence than +)
  - We may **use parentheses** rules (2+4)*3 or 2+(4*3)

Rules of Precedence for Arithmetic Operators

Operator	Rule of Precedence
^	Exponentiation (^) is performed first
* /	Multiplication (*) and division (/) are performed following exponentiation.
+ -	Addition (+) and subtraction (-) are performed last.

- Use parentheses to override precedence rules

There are two more forms for representing an arithmetic expressions in which they do not need precedence rules or parentheses:

- postfix
- prefix

Arithmetic Expressions

- Postfix form: Refers to the notation in which the **operator** symbol is placed **after** its two operands
  - Operator appears **after** the operands
    - Infix: (4+3)*5 → Postfix: 4 3 + 5 *
    - Infix: 4+(3*5) → Postfix: 4 3 5 * +
  - **No precedence rules or parentheses!**
- **Prefix Form**: Refers to the notation in which the **operator** symbol is placed **before** its two operands.
  - Operator appears **before** the operands
    - Infix: (4+3)*5 → Prefix: *+4 3 5
    - Infix: 4+(3*5) → Prefix: +4 *3 5
  - **No precedence rules or parentheses!**
- **Two Questions:**
  - **How to convert an infix form to postfix and prefix forms.**
  - **How to evaluate an expression given in postfix and prefix forms.**

Stack Applications

Arithmetic Expression

- Conversions
- Evaluations

Example: Infix to Postfix

- Example1 :

  A + B * C + D

  ⇒ A+BC*+D

  ⇒ ABC*++D

  ⇒ ABC*+D+

- Example2:

  A * B + C * D

  ⇒ AB*+C*D

  ⇒ AB*+CD*

  ⇒ AB*CD*+

- Example3:

  A +B*C-D/E*F

  ⇒ A +BC*-D/E*F

  ⇒ A +BC*-DE/*F

  ⇒ A +BC*-DE/F*

  ⇒ A BC*+-DE/F*

  ⇒ A BC*+DE/F*-

- Example4:

  (A+B)*(C+D)

  ⇒ (A B+)*(C+D)

  ⇒ (A B+)*(CD+)

  ⇒ (AB)+(CD+)*

  ⇒ AB+CD+*

Example: Infix to Prefix

- Example1 :

  A + B * C + D

  ⇒ A+*BC+D

  ⇒ A++*BCD

  ⇒ +A+*BCD

- Example2:

  A * B + C * D

  ⇒ A*B+*CD

  ⇒ *AB+*CD

  ⇒ +*AB*CD

- Example3:

  A +B*C-D/E*F

  ⇒ A +B*C-D/*EF

  ⇒ A +B*C-/D*EF

  ⇒ A +*BC-/D*EF

  ⇒ A +-*BC/D*EF

  ⇒ +A -*BC/D*EF

- Example4:

  (A +B)*(C+D)

  ⇒ (A+ B)*(+CD)

  ⇒ (+A B)*(+CD)

  ⇒ *(+A B)(+CD)

  ⇒ *+A B+CD

Infix to Postfix Algorithm

```
While (we have not reached the end of infix expression) // Read from left to right.
    If (an operand is found) then
        Add it to Postfix
    If (a left parenthesis '(' is found) then
        Push it onto the stack
    If (a right parenthesis ')' is found) then
        While (the stack is not empty AND the top item is not a left parenthesis)
            Pop the stack and add the popped value to Postfix
        End-While
        Pop the left parenthesis from the stack and discard it
    If (an operator is found) then
        If (the stack is empty or if the top element is a left parenthesis) then
            Push the operator onto the stack
```

Infix to Postfix Algorithm

```
Else
            While (the stack is not empty AND the top of the stack  is not a left parenthesis
                  AND precedence of the operator <= precedence of the top of the stack)
              Pop the stack and add the top value to Postfix
            End-While
            Push the latest operator onto the stack
End-While
While (the stack is not empty)
   Pop the stack and add the popped value to Postfix
End-While
```

Infix to Postfix Algorithm

```
While (we have not reached the end of infix expression)
    If (an operand is found) then
      Add it to Postfix
    If (a left parenthesis '(' is found) then
      Push it onto the stack
    If (a right parenthesis ')' is found) then
      While (the stack is not empty AND the top item is not a left parenthesis)
        Pop the stack and add the popped value to Postfix
      End-While
      Pop the left parenthesis from the stack and discard it
    If (an operator is found) then
      If (the stack is empty or if the top element is a left parenthesis) then
        Push the operator onto the stack
      Else
        While (the stack is not empty AND the top of the stack  is not a left parenthesis  AND precedence of
              the operator <= precedence of the top of the stack)
          Pop the stack and add the top value to Postfix
        End-While
        Push the latest operator onto the stack
End-While
While (the stack is not empty)
   Pop the stack and add the popped value to Postfix
End-While
```

Infix to Postfix Algorithm Example

- Infix Form: (A+B*C-D)/(E*F)

Token	Stack	Postfix
(	(	
A	(	A
+	(+	
B	(+	AB
*	(+*	
C	(+*	ABC
-	(-	ABC*+
D	(-	ABC*+D
)		ABC*+D-
/	/	
(	/(	
E	/(	ABC*+D-E
*	/(*	
F	/(*	ABC*+D-EF
)	/	ABC*+D-EF*
		ABC*+D-EF*/

- How to convert infix to prefix?
- What is the algorithm of converting an infix to prefix?

Hint: update the little things from the algorithm of converting infix to postfix.

Evaluating a postfix expression Algorithm

```
While (we have not reached the end of expression) // Read from left to right.
        If an operand is found then
            push it onto the stack
        If an operator is found then
            // Pop Twice
            A=Pop()
            B=Pop()
            Evaluate B operator A using the operator just found.
            Push the resulting value onto the stack.
End-While
    Pop the stack (this is the final value)
```

Evaluating a postfix expression - Example

- Postfix: 244*+6-23*/
  - Infix: (2+4*4-6)/(2*3)=2

Token	Stack
2	2
4	2 4
4	2 4 4
*	2 16
+	18
6	18 6
-	12
2	12 2
3	12 2 3
*	12 6
/	2

How to evaluate a prefix expression?

Hint: update the little things from the algorithm of converting infix to postfix.

Stack Implementation

- **Array**: We will use this first.
- **Linked Lists**: Later to be implemented with list.

Lab Assignment

- Implement the Stack in C++ using OOP.

Exercises

Exercises

- A linear list of elements in which deletion and insertion can be done from one side is known as a?
  a) Queue.
  b) Stack.
  c) Tree.
  d) Linked list.
- A Stack follows
  a) FIFO (First In First Out) principle.
  b) LIFO (Last In First Out) principle.
  c) Ordered array.
  d) Linear tree.

Exercises

- Convert the following infix expression to postfix expressions **using Stack** data structure.
  - (5 * (((9 + 8) * (4 * 6)) + 7))
  - 6 * (5 + (2 + 3) * 8 + 3)
- Convert the following infix expression to prefix expressions **using Stack** data structure.
  - a + b * c + (d * e + f) * g
- For each of the of the following postfix expressions, find the infix.
  - 6 5 2 3 + 8 * + 3 + *
  - a b c * + d e * f + g * +
- Evaluate the following postfix expression
  - 6 2 5 3 + 4 * + 3 + *