STACK

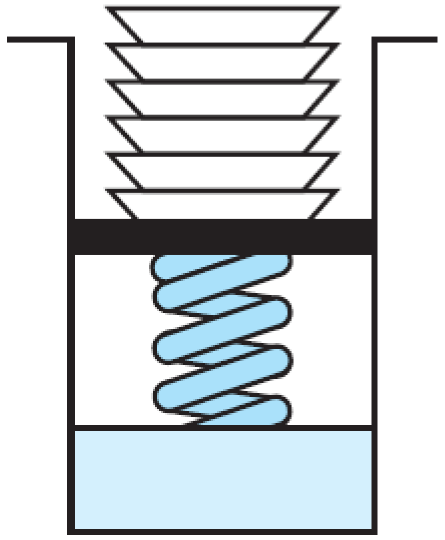
XUEFAN LI

08/26/2018

CONTENT

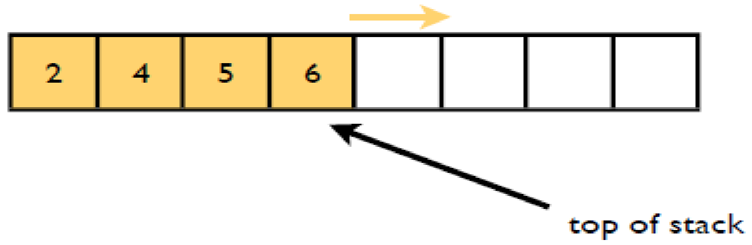
1. 1 CONCEPT

- a stack is an abstract data type with two principal operations: push, which adds an element to the collection, and pop, which removes the most recently added element that was not yet removed.
- LIFO
- isEmpty()
- push(newEntry)
- pop()
- top()

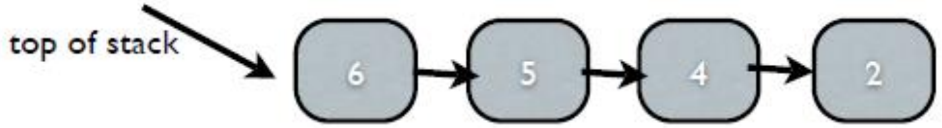


1.2 IMPLEMENTATION

- Array based VS linked list based
- Array implementation of stack



- Linked list implementation of stack



```
Linked list:
def push(self, elem):
    self._top=Lnode(elem,self._top)

def pop(self):
    if self._top is None:
        raise StackUnderflow()
    p=self._top
    self._top=p.next
    return p.elem
```

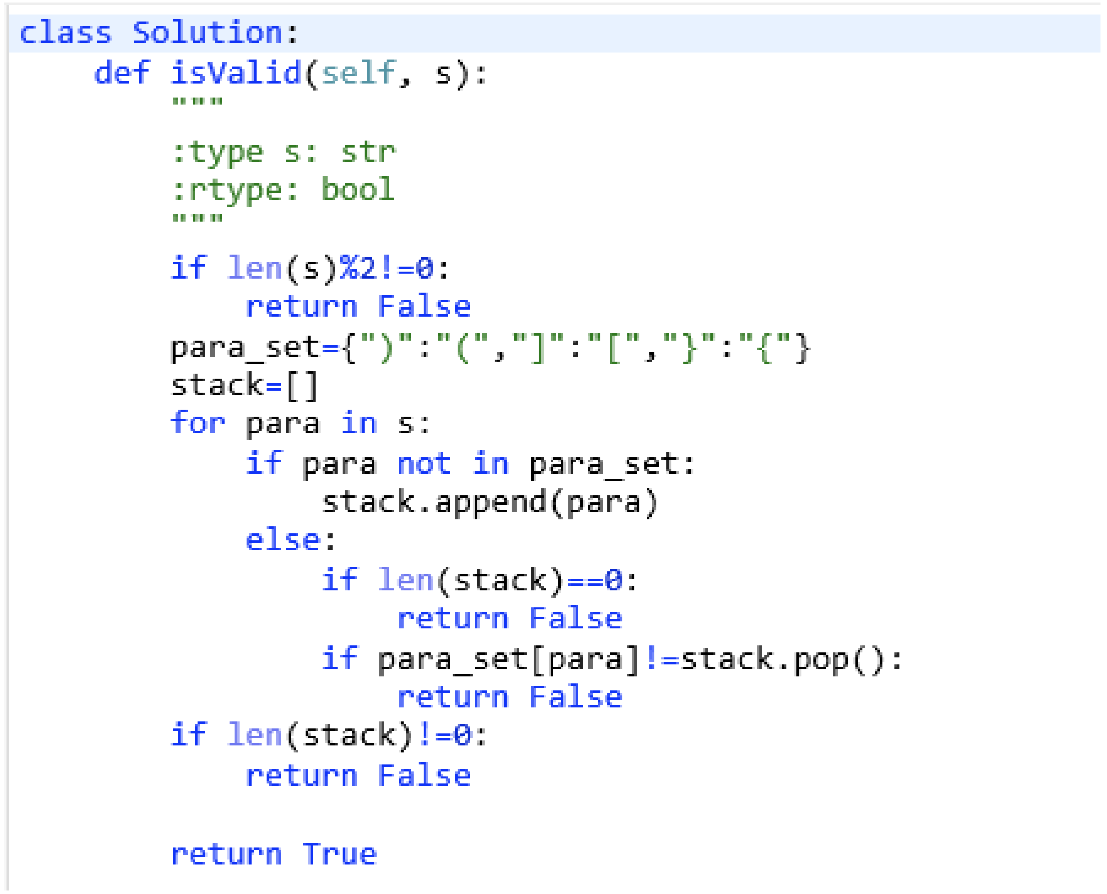
2. APPLICATION

- **Example 1: valid parenthesis**
- Given a string containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

An input string is valid if:

1.Open brackets must be closed by the same type of brackets.

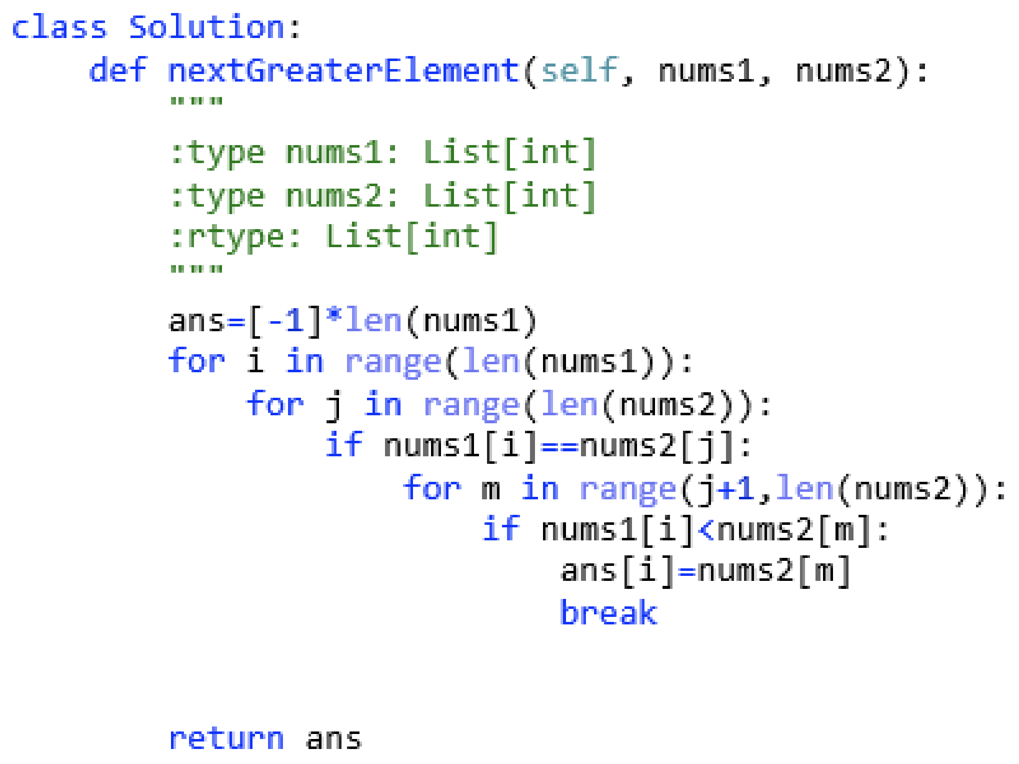
2.Open brackets must be closed in the correct order.



```python
class Solution:
    def isValid(self, s):
        """
        :type s: str
        :rtype: bool
        """
        if len(s)%2!=0:
            return False
        para_set={")":"(","]":"[","}":"{"}
        stack=[]
        for para in s:
            if para not in para_set:
                stack.append(para)
            else:
                if len(stack)==0:
                    return False
                if para_set[para]!=stack.pop():
                    return False
        if len(stack)!=0:
            return False

        return True
```

2. APPLICATION

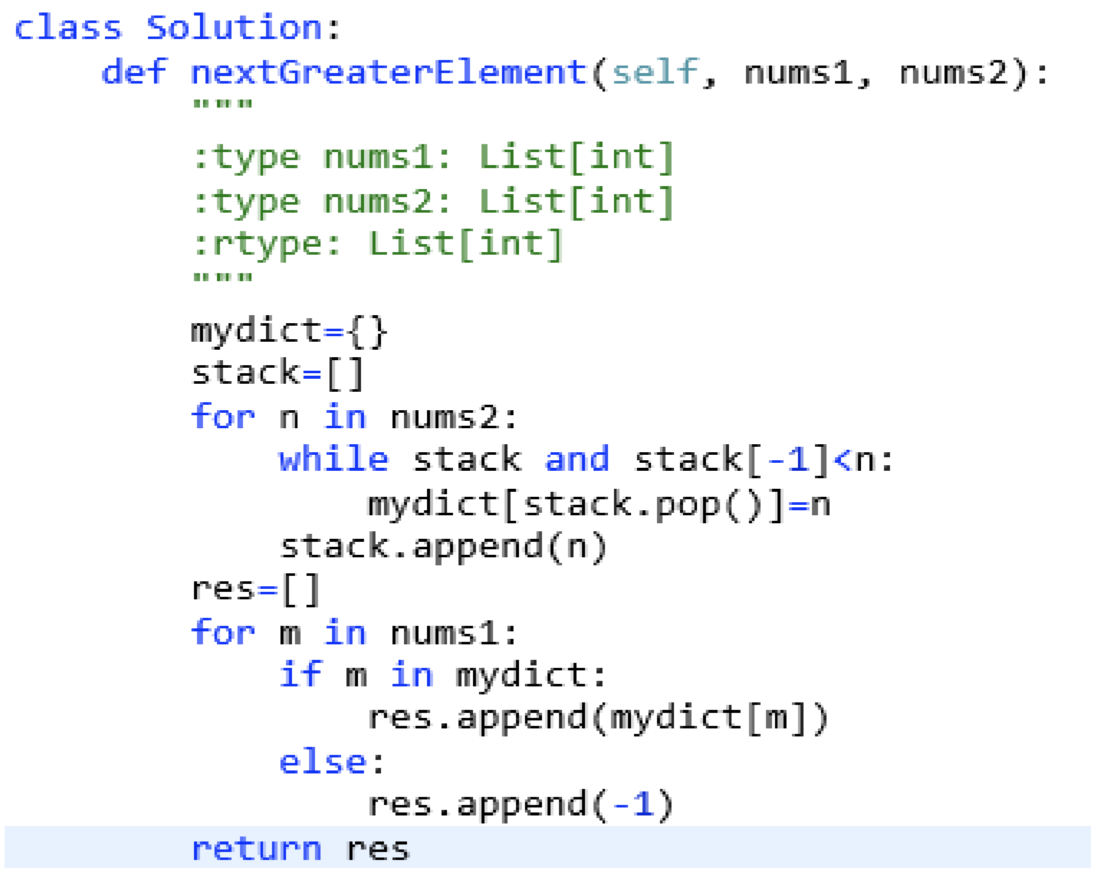
- **Example 2 Next Greater Element I**
- You are given two arrays (without duplicates) nums1 and nums2 where nums1's elements are subset of nums2. Find all the next greater numbers for nums1's elements in the corresponding places of nums2.
- The Next Greater Number of a number x in nums1 is the first greater number to its right in nums2. If it does not exist, output -1 for this number.
- E.g. Nums1=[4,1,2] nums2=[1,3,4,2] Output=[-1,3,-1]
- Time complexity: O(n^3) ---No!



```python
class Solution:
    def nextGreaterElement(self, nums1, nums2):
        """
        :type nums1: List[int]
        :type nums2: List[int]
        :rtype: List[int]
        """
        ans=[-1]*len(nums1)
        for i in range(len(nums1)):
            for j in range(len(nums2)):
                if nums1[i]==nums2[j]:
                    for m in range(j+1,len(nums2)):
                        if nums1[i]<nums2[m]:
                            ans[i]=nums2[m]
                            break


        return ans
```

2. APPLICATION

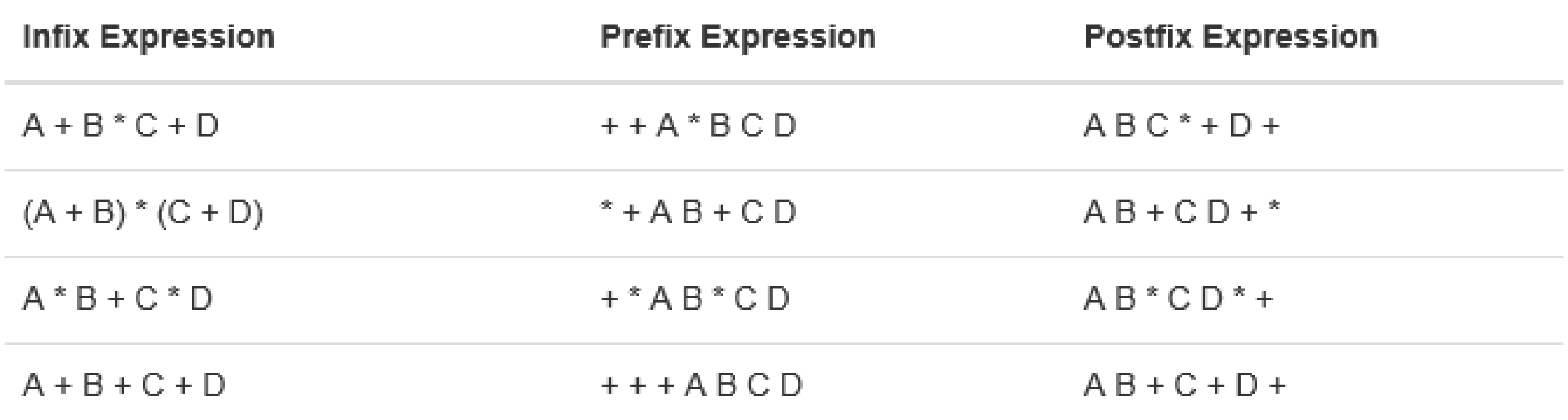
- **Example 2 Next Greater Element I**
- You are given two arrays (without duplicates) nums1 and nums2 where nums1’s elements are subset of nums2. Find all the next greater numbers for nums1's elements in the corresponding places of nums2.
- The Next Greater Number of a number x in nums1 is the first greater number to its right in nums2. If it does not exist, output -1 for this number.
- Time complexity: O(n)



```
class Solution:
    def nextGreaterElement(self, nums1, nums2):
        """
        :type nums1: List[int]
        :type nums2: List[int]
        :rtype: List[int]
        """
        mydict={}
        stack=[]
        for n in nums2:
            while stack and stack[-1]<n:
                mydict[stack.pop()]=n
            stack.append(n)
        res=[]
        for m in nums1:
            if m in mydict:
                res.append(mydict[m])
            else:
                res.append(-1)
        return res
```

2. APPLICATION

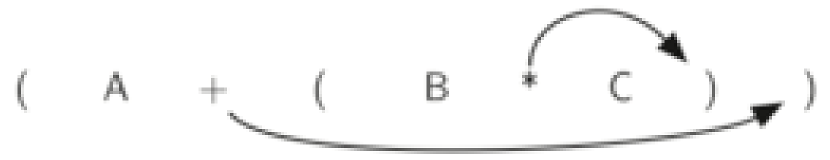
- **Infix, prefix and postfix**



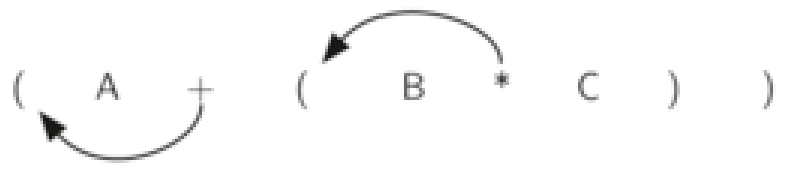
Infix Expression	Prefix Expression	Postfix Expression
A + B * C + D	+ + A * B C D	A B C * + D +
(A + B) * (C + D)	* + A B + C D	A B + C D + *
A * B + C * D	+ * A B * C D	A B * C D * +
A + B + C + D	+ + + A B C D	A B + C + D +

- http://interactivepython.org/runestone/static/pythonds/BasicDS/InfixPrefixandPostfixExpressions.html

- How to convert?
- Infix->Postfix



- Infix->Prefix



CONVERSION FROM INFIX TO POSTFIX

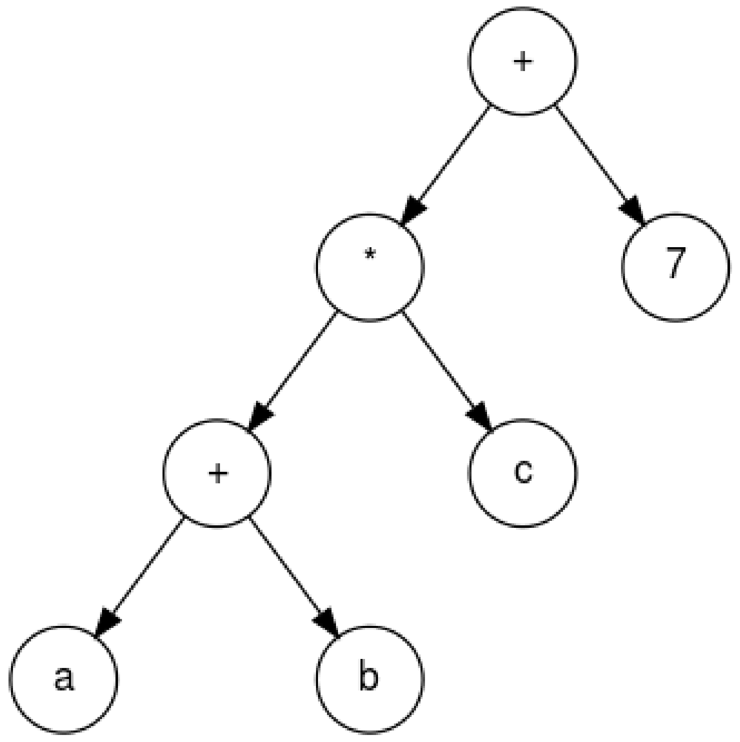
- 1.Create an empty stack called opstack for keeping operators. Create an empty list for output.
- 2. Scan:
- If operand, append it to the end of the output list.
- If left parenthesis, push it on the opstack.
- If right parenthesis, pop the opstack until the corresponding left parenthesis is removed.
- If operator, *, /, +, or -, push it on the opstack. However, first remove any operators already on the opstack that have higher or equal precedence and append them to the output list.
- 3. Pop and append anything left to the output list

```
def infixToPostfix(infixexpr):
    prec = {}
    prec["*"] = 3
    prec["/"] = 3
    prec["+"] = 2
    prec["-"] = 2
    prec["("] = 1
    opStack = Stack()
    postfixList = []
    tokenList = infixexpr.split()

    for token in tokenList:
        if token in "ABCDEFGHIJKLMNOPQRSTUVWXYZ" or token in "0123456789":
            postfixList.append(token)
        elif token == '(':
            opStack.push(token)
        elif token == ')':
            topToken = opStack.pop()
            while topToken != '(':
                postfixList.append(topToken)
                topToken = opStack.pop()
        else:
            while (not opStack.isEmpty()) and \
               (prec[opStack.peek()] >= prec[token]):
                  postfixList.append(opStack.pop())
            opStack.push(token)

    while not opStack.isEmpty():
        postfixList.append(opStack.pop())
    return " ".join(postfixList)
```

EXPRESSION TREE

In-order traversal-> infix

```
Algorithm infix (tree)
/*Print the infix expression for an expression tree.
 Pre : tree is a pointer to an expression tree
 Post: the infix expression has been printed*/
 if (tree not empty)
    if (tree token is operator)
       print (open parenthesis)
    end if
    infix (tree left subtree)
    print (tree token)
    infix (tree right subtree)
    if (tree token is operator)
       print (close parenthesis)
    end if
 end if
end infix
```

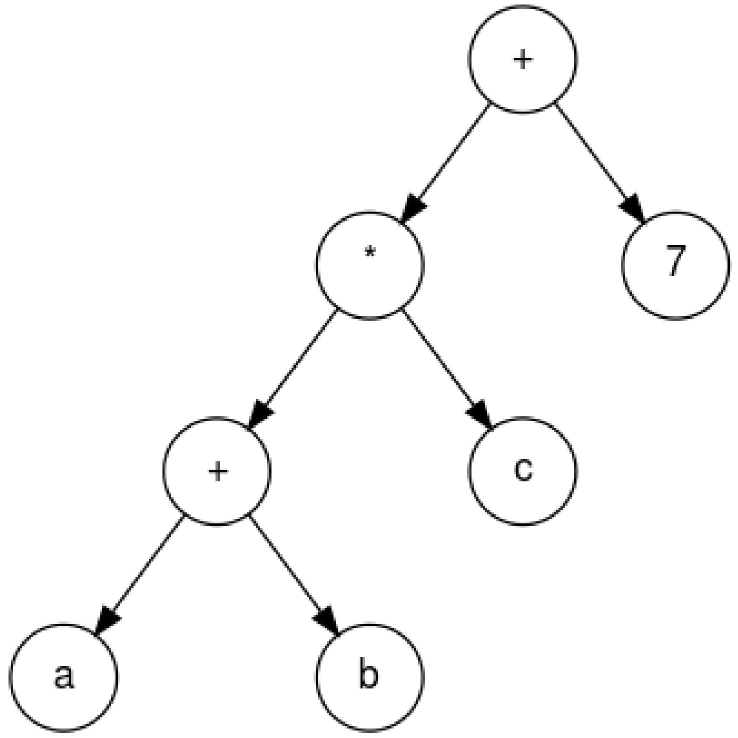


(a+b)*c+7

EXPRESSION TREE

Post-order traversal-> postfix

```
Algorithm postfix (tree)
/*Print the postfix expression for an expression tree.
 Pre : tree is a pointer to an expression tree
 Post: the postfix expression has been printed*/
 if (tree not empty)
    postfix (tree left subtree)
    postfix (tree right subtree)
    print (tree token)
 end if
end postfix
```

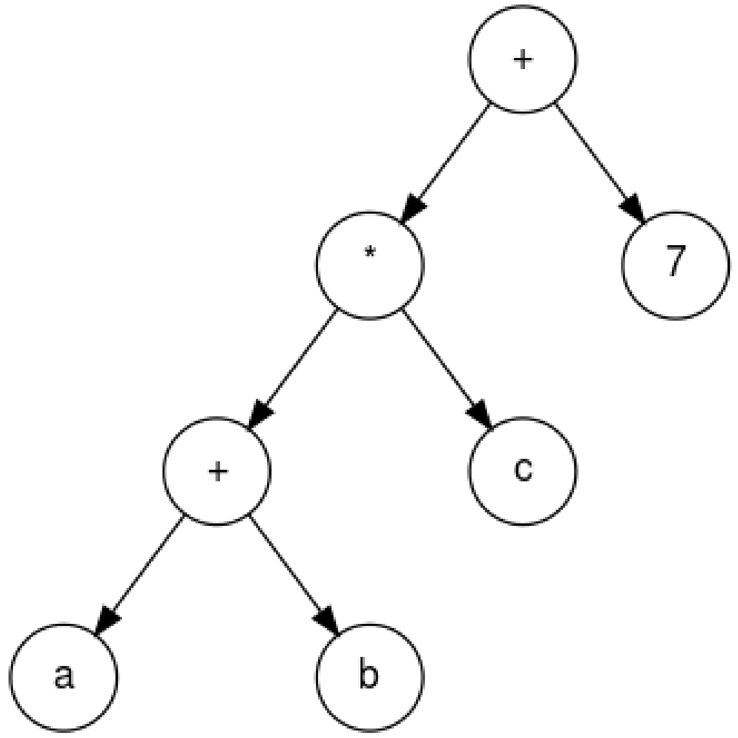


ab+c*7+

EXPRESSION TREE

Pre-order traversal-> prefix

```
Algorithm prefix (tree)
/*Print the prefix expression for an expression tree.
 Pre : tree is a pointer to an expression tree
 Post: the prefix expression has been printed*/
 if (tree not empty)
    print (tree token)
    prefix (tree left subtree)
    prefix (tree right subtree)
 end if
end prefix
```



+*+abc7

EXAMPLE 3

GIVEN A BINARY TREE, RETURN THE INORDER TRAVERSAL OF ITS NODES' VALUES.

- **Recursively is easy:**

```
class Solution:
    def inorderTraversal(self, root):
        """
        :type root: TreeNode
        :rtype: List[int]
        """
        return  (self.inorderTraversal(root.left) + [root.val] + self.inorderTraversal(root.right)) if root else  []
```

Iteratively:

```
current = root
s = [] # initialze stack
done = 0
res=[]
while(not done):
    if current is not None:
        s.append(current)
        current = current.left
    else:
        if(len(s) >0):
            current = s.pop()
            res.append(current.val)
            current = current.right
        else:
            done = 1
return res
```

HOMEWORK

- **Example 1 extension:**
- 1. Generate parenthesis
- 2. Largest valid parenthesis
- **Example 2 extension:**
- 3. Next greater element II
- 4. Next greater element III
- **Example 3 extension:**
- 5. Binary tree post order traversal
- 6. Binary tree pre order traversal
- 7. Binary Tree Zigzag Level Order Traversal

- 8. Largest Rectangle in Histogram

Thanks!