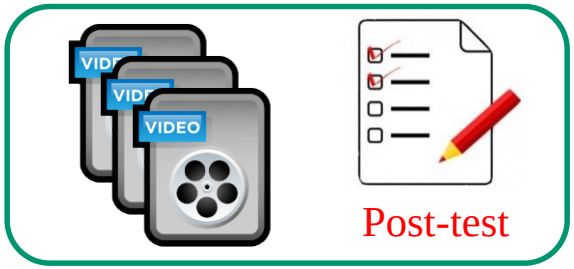
CMPUT 175
Introduction to Foundations of Computing

Stacks



You should view the vignettes:

Abstract data Types
Concept of Encapsulation
Encapsulation, the Gearbox example
Stacks

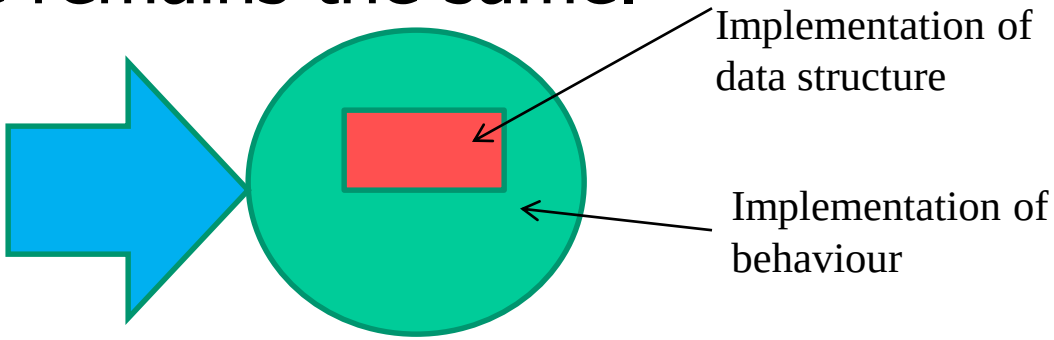
Objectives

- In this lecture we will learn about a group of data structures called linear structures.
- We will learn about Stacks, Queues and Deques and see how to implement them in python.
- We will discuss some example useful applications of these data structures.

From Formal ADT to Implementation

- An ADT is a formal description, not code; independent of any programming language
- We will see the description of these structures then their implementation
- There are many possible implementations
- The code using these implementations doesn't have to change if the implementation changes as long as the interface remains the same.

Access the data **via** interface (behaviour) and never directly to data structure implementation



Linear Structures

- A Linear Structure is a non-indexed container object that can grow and shrink one element at a time.
- However, it is not specified where the container grows or shrinks.
- Linear structures have two ends: Front and Rear (or Top and Bottom) (or Head and Tail)
- Linear structures are containers that differ on the location their items are added or removed

Examples of ADT

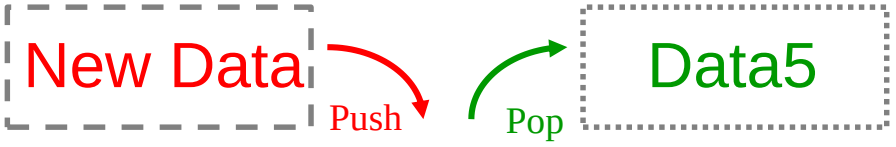
Linear Structures

- Lists
- Stacks
- Queues
- Deque
- Pile (sorted deque)
- Trees
- Heaps

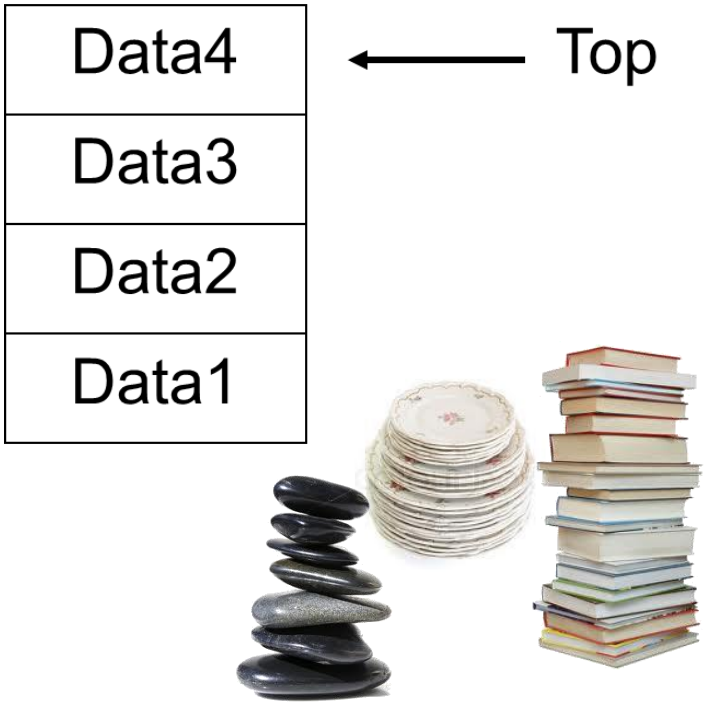
- Binary search trees
- B trees
- Maps
- Graphs
- …

Stacks

- Collection with access only to the last element inserted



- Last In - First Out (LIFO)
  - Add element: push
  - Remove element: pop
  - top
  - is empty
  - check size



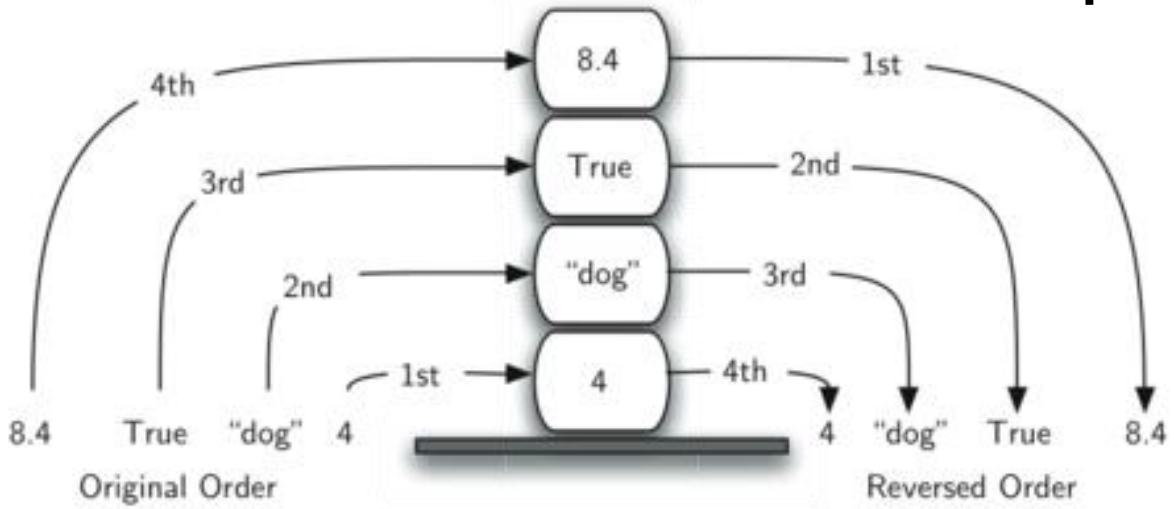
How useful are Stacks

We will see some examples:

- Reversing sequences
- Navigating the web
- Traversing a maze
- Parser matching parenthesis
- Infix, Prefix and postfix expressions

Reversing a sequence

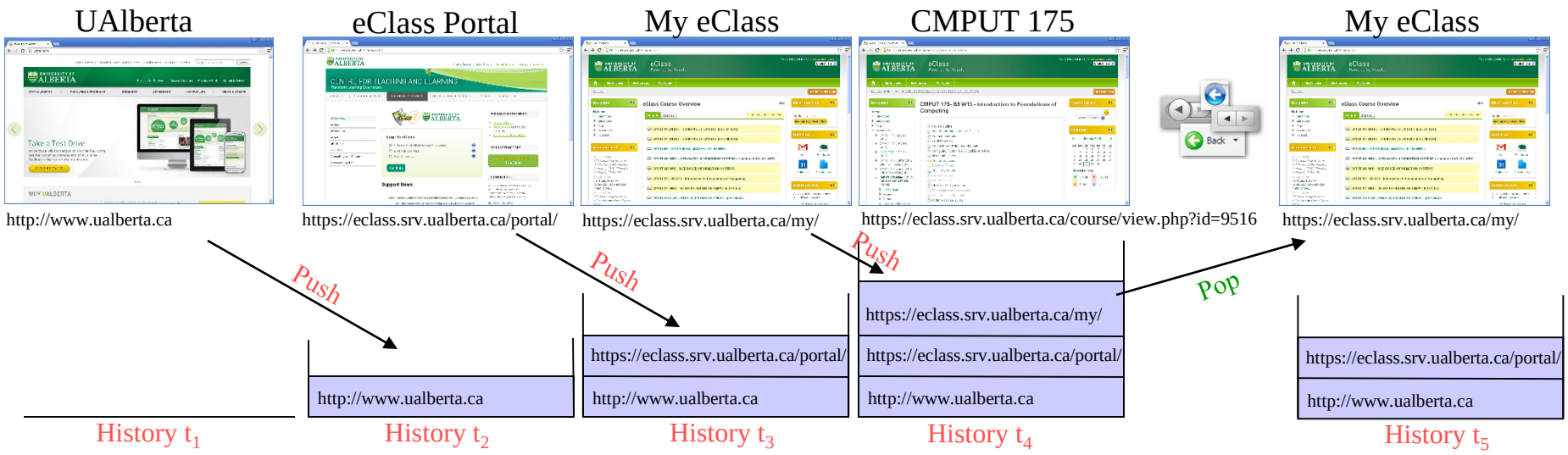
- A stack has a reversal property due to LIFO: Last element in is the first out
- By pushing elements of a sequence in a stack, popping them afterwards from the stack would reverse the sequence



From textbook page 84

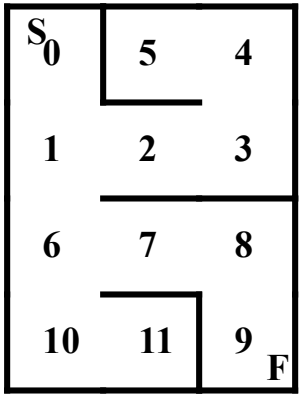
Navigating the web

- How does the browser remember the pages I visited when press the "back" button?
- Each time a pages is visited the browser "pushes" the URL into a history stack
- When the back button is pressed, the browser "pops" a URL from the history stack and displays the corresponding page.



Maze Algorithm

- One common approach to solving search problems is to use a Stack to hold unsearched paths.

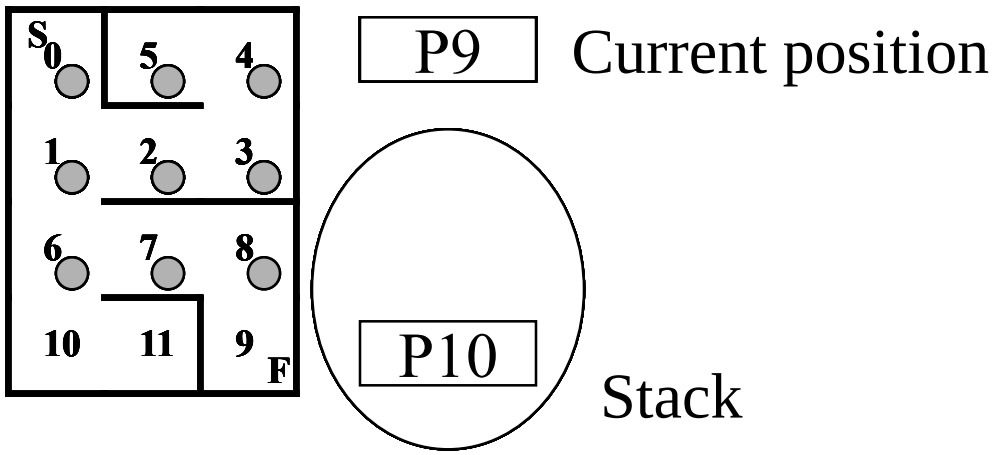


- Select the start square as the current square.
- Repeat as long as the current square is not null and is not the finish square:
  - "Visit" the square and mark it as visited.
  - Push one square on the stack for each unvisited legal move from the current square.
  - Pop the stack into the current square or bind the current square to null if the stack is empty.
- If the current square is the goal we are successful, otherwise there is no solution

Maze Trace

In each position
push all legal moves that were not visited
pop one position and move to it

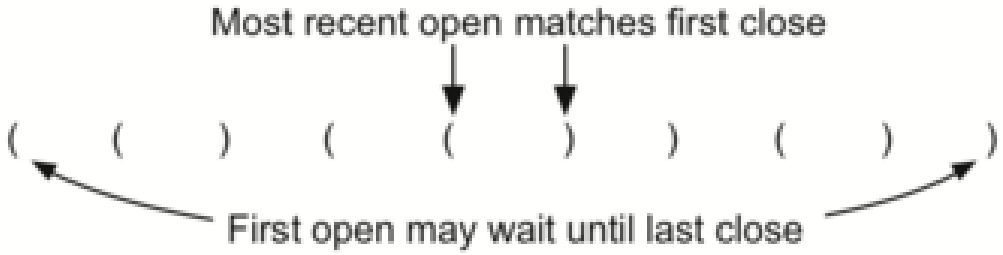
Fail if stack empty
Success if Final



Success!

Matching Parenthesis

- Push to stack opening parenthesis when encountering an opening parenthesis
- Pop from stack when encountering closing parenthesis and check if it corresponds
- Correct if all correspond and at the end the stack is empty

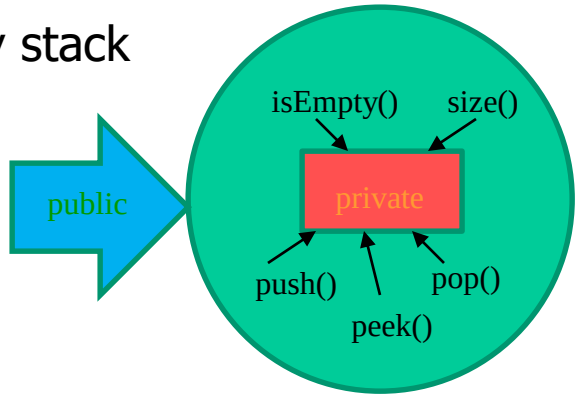


From textbook page 89

- ([{}]) ([){]} (({{{}})) [(())] (([]{}()))

Stack Abstract data Type

- **Stack()**
  - Create a new stack that is empty.
  - It needs no parameters and returns an empty stack
- **push(item)**
  - Adds a new item to the top of the stack.
  - It needs an item and returns nothing
- **pop()**
  - Remove the top item from the stack
  - It needs no parameters and returns the item.
  - The stack is modified.
- **peek()**
  - Returns the top item from the stack but does not remove it
  - It needs no parameters and the stack is not modified
- **isEmpty()**
  - Test to see whether the stack is empty
  - It needs no parameters and returns a Boolean value
- **size()**
  - Returns the number of items on the stack
  - It needs no parameters and returns an integer



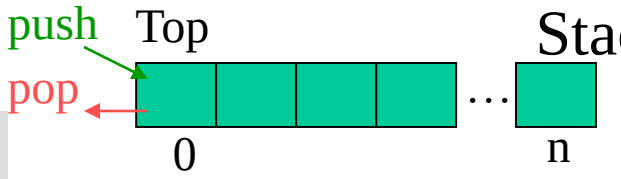
- **Reset()**
  - Empties the stack
  - It needs no parameters
  - The stack is modified

Stack Implementation in Python

- How to store the elements in the stack and allow the stack to grow and shrink one element at a time?
  - Using a python List
  - We chose the top and bottom of the stack to correspond to some fixed end of the list
- Implement each and every method as specified in the Stack ADT (Push, Pop, Peek, isEmpty, Size)
- Implement the class and instance creator

Implementation 1

- Assuming we chose to have the top of the stack correspond to the beginning of a list.




```
class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.insert(0,item)

    def pop(self):
        return self.items.pop(0)
```

```
def peek(self):
        return self.items[0]

def isEmpty(self):
        return self.items == []

def size(self):
        return len(self.items)
```

Not necessarily the best choice as we shall see

Printing the stack

- How to display the stack instance?
- The stack is implemented as a list and python knows how to display it.
- It is better to define a method to display the stack. Let's call it show()

```
def show(self):
        print (self.items)
```

```
def __str__(self):
        return str(self.items)
```

Converts the object into a string

Let's test it

```
s=Stack()
s.show()
print (s.isEmpty())
s.push("bob")
s.show()
print (s.isEmpty())
s.push("eva")
s.push("paul")
s.show()
print (s.size())
item=s.peek()
print (item, "is on top of",s)
item=s.pop()
s.show()
print (item,"was on the stack")
print (s.size())
```

```
[]
True

['bob']
False


['paul', 'eva', 'bob']
3

paul is on top of ['paul', 'eva', 'bob']

['eva', 'bob']
paul was on the stack
2
```

It seems to work as designed but these are very rudimentary tests. More stringent tests done in isolation are always required.

Matching parenthesis

- Push to stack opening parenthesis when encountering an opening parenthesis
- Pop from stack when encountering closing parenthesis and check if it corresponds
- Correct if all correspond and at the end the stack is empty

```
given parenthesis
create stack s
balanced ← True
index ← 0
while (balanced and index smaller than size of parenthesis) do
    if (parenthesis[index] in "([{")  push(parenthesis[index]) in s
    else
        if (stack s is empty) balanced ← False
        else
            top=pop() from s
            if (top and parenthesis[index] do not match) balanced←False
            endif
        endif
    endif
    index ← index +1
endwhile
if (balanced and stack s is empty) return True
else return False
endif
```

Implementing in Python

```
def parChecker(symbolString):
    s = Stack()
    balanced = True
    index = 0
    while index < len(symbolString) and balanced:
        symbol = symbolString[index]
        if symbol in "([{":
            s.push(symbol)
        else:
            if s.isEmpty():
                balanced = False
            else:
                top = s.pop()
                if not matches(top,symbol):
                    balanced = False
        index = index + 1

    if balanced and s.isEmpty():
        return True
    else:
        return False
```

```
def matches(open,close):
        opens = "([{"
        closers = ")]}"
        return opens.index(open) == closers.index(close)
```

```
parenthesis= "([](()){()})"
print(parChecker(parenthesis))
True

parenthesis= "([])[){[)})"
print(parChecker(parenthesis))
False
```

Let's time it

- We will test parChecker() using our Stack implementation calling parChecker() 10 times then 100 times in 3 repetitions using a long sequence of parenthesis.

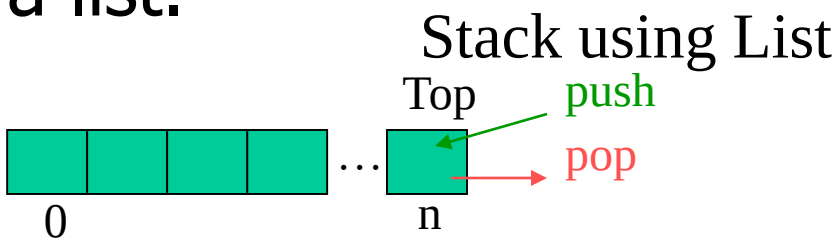
```
t=timeit.Timer("parChecker(p)",setup='from __main__ import parChecker;p="({[([{({[]})}])]})(){}[]{{{[[[(())]]]}}}("')
print ("10 times: %17.14f milliseconds"%(t.timeit(number=10)))
print(t.repeat(repeat=3,number=100),"milliseconds (3x100 times)")
```

```
10 times:  0.00111746616541 milliseconds
[0.010608966165413534, 0.010612721804511277, 0.010742304511278191] milliseconds (3x100 times)
```

Implementation 2

- Now we chose to have the top of the stack correspond to the end of a list.

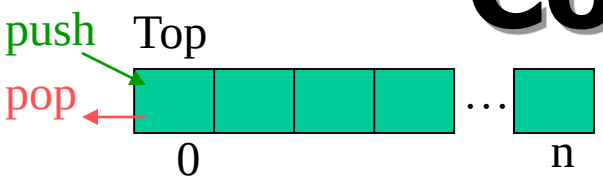
Avoids shifting elements when a new one is added or removed

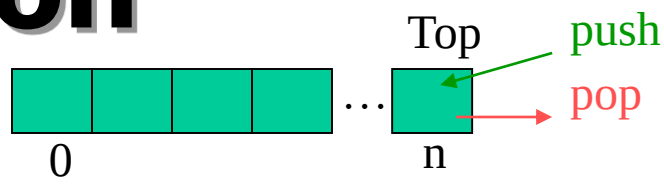



```
class Stack:
    def __init__(self):
        self.items = []

        def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()
```

```
def peek(self):
        return self.items[len(self.items)-1]

def isEmpty(self):
        return self.items == []

def size(self):
        return len(self.items)
```

Comparison





Implementation 1

```
def __init__(self):
        self.items = []

def push(self, item):
        self.items.insert(0,item)

def pop(self):
        return self.items.pop(0)

def peek(self):
        return self.items[0]

def isEmpty(self):
        return self.items == []

def size(self):
        return len(self.items)
```

Implementation 2

```
def __init__(self):
        self.items = []

def push(self, item):
        self.items.append(item)

def pop(self):
        return self.items.pop()

def peek(self):
        return self.items[len(self.items)-1]

def isEmpty(self):
        return self.items == []

def size(self):
        return len(self.items)
```

Let's test implementation 2

```
s=Stack()
s.show()
print (s.isEmpty())
s.push("bob")
s.show()
print (s.isEmpty())
s.push("eva")
s.push("paul")
s.show()
print (s.size())
item=s.peek()
print (item, "is on top of",s)
item=s.pop()
s.show()
print (item,"was on the stack")
print (s.size())
```

```
[]
True

['bob']
False


['bob', 'eva', 'paul']
3

paul is on top of ['bob', 'eva', 'paul']

['bob', 'eva']
paul was on the stack
2
```

It seems to work like implementation 1.

Time implementation2

- We replace the 1st Stack class implementation with the second one and rerun parCheck using our new Stack implementation like we did before

```
t=timeit.Timer("parChecker(p)",setup='from __main__ import parChecker;p="({[([{({[]})}])]})(){}[]{{{[[[(())]]]}}}("')
print ("10 times: %17.14f milliseconds"%(t.timeit(number=10)))
print(t.repeat(repeat=3,number=100),"milliseconds (3x100 times)")
```

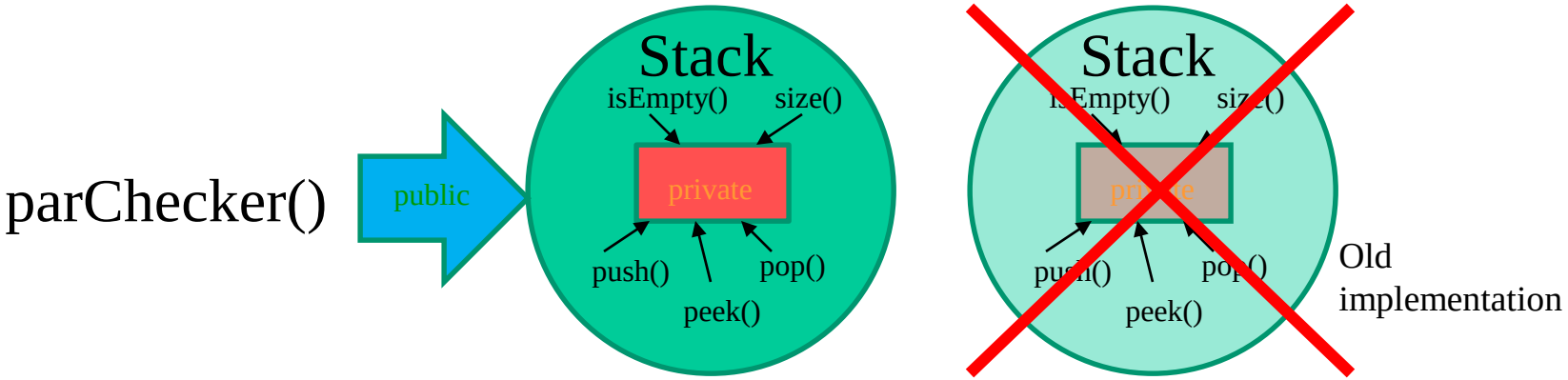
```
10 times:  0.00090155639098 milliseconds
[0.009034458646616 54, 0.009115150375939851, 0.008975492481203 01] milliseconds (3x100 times)
```

Remember, with previous implementation

```
10 times:  0.00111746616541 milliseconds
[0.010608966165413534, 0.010612721804511277, 0.010742304511278191] milliseconds (3x100 times)
```

Lesson to learn

- Thanks to the encapsulation of the Stack implementation and the clear interface we were able to swap the implementation of Stak with another one without having to change or update the program [parChecker()] that uses it.



Arithmetic Expressions

- Arithmetic expression we have series of operators and operands
- Operators are +, -, / * and ^
- Each Operators, except ^, has two operands
- In an expression we can have many operators
- Operators have different priorities (precedence)
- ^ has highest precedence
- * and / have higher precedence than + and -

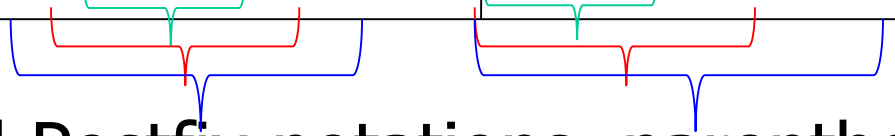
Infix, Prefix and Postfix Expressions

- Arithmetic expression: X * Y + Z
- Infix: Operator appears between variables – eg. X * Y
- Because of operator precedence rules
- X * Y + Z is equivalent to (X * Y) + Z
- Parenthesis remove ambiguity
- X * Y + Z is different from X * (Y + Z)
- How do we remover ambiguity to the computer? Should we have fully parenthesized expressions?

Infix, Prefix and Postfix

- Infix: X * Y → operator between operands
- Prefix: * X Y → operators before operands
- Postfix: X Y * → operators after operands

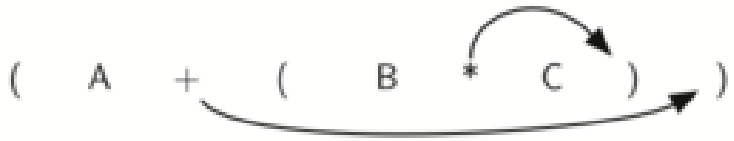
Infix Expression	Prefix Expression	Postfix Expression
X + Y * Z	+ X * Y Z	X Y Z * +
(X + Y) * Z	*+X Y Z	X Y+ Z *
(X + Y) * Z + V	+*+ X Y Z V	X Y + Z * V +



- With Prefix and Postfix notations, parenthesis are not required to disambiguate expressions.

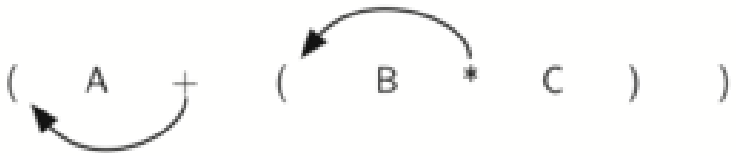
Conversion from Infix to Pre and Postfix expressions

- Transform to fully parenthesized then convert



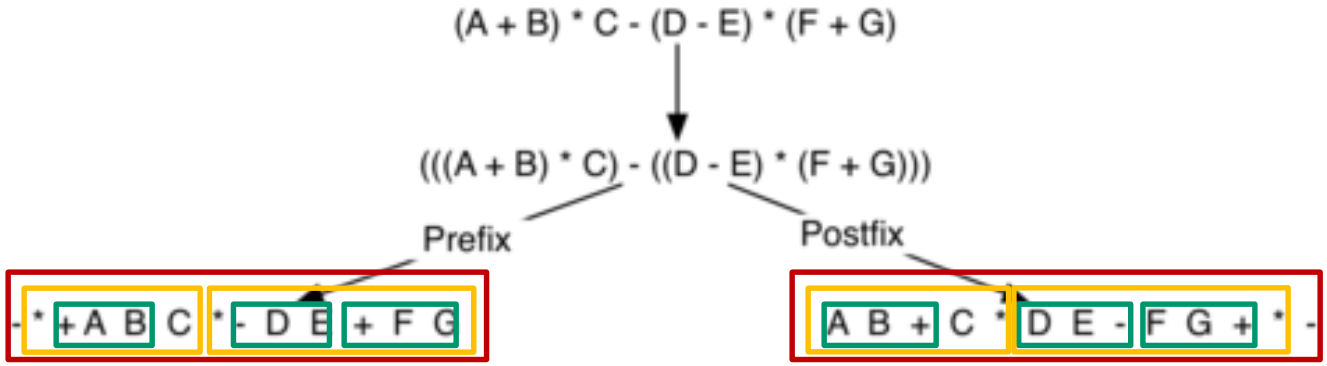
Moving Operators to the Right for Postfix Notation

A B C * +



Moving Operators to the Left for Prefix Notation

+ A * B C



From textbook pages 98-99

Converting a Complex Expression to Prefix and Postfix Notations

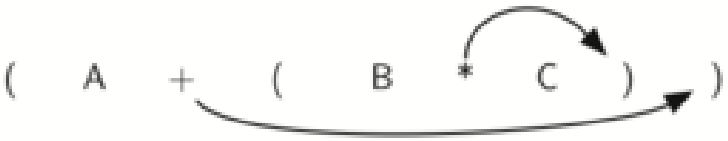
- Operands don't move. Only operators change position

Advantage of Postfix notation

- Also called Reverse Polish notation
- Invented in 1920 by Polish logician Jan Łukasiewicz
- It is unambiguous and does not require parenthesis or brackets
- Can be used for arithmetic, logic and algebra
- We will see how to convert an infix expression to a postfix expression using a Stack. Then we will evaluate a postfix expression using a stack to find out the value of the expression.

General Infix to Postfix Conversion using Stack

- When we converted A+B*C we obtained ABC*+; the sequence of operands ABC didn't change but the sequence of operators got reversed from + * to * +



Moving Operators to the Right for Postfix Notation

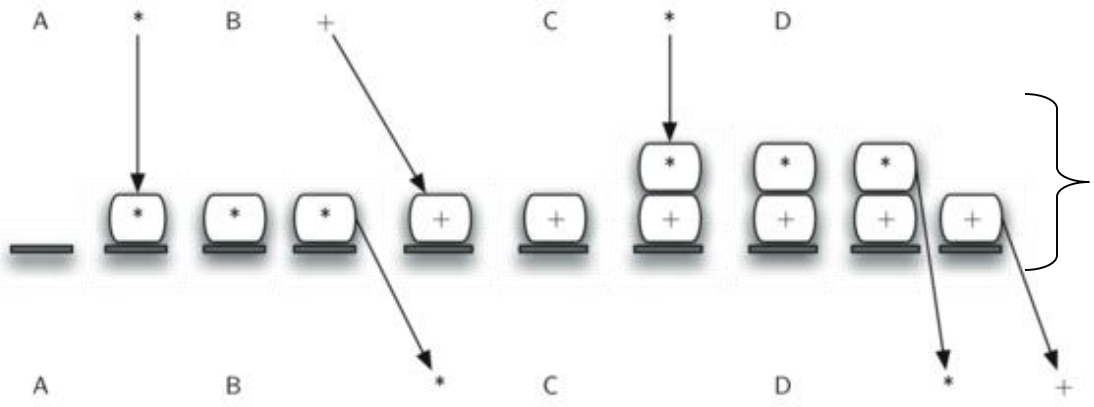
A B C * +

- This observation hints on the use of a Stack to do this conversion algorithmically.
- Assuming the infix expression to convert is a string in which each character being either an operator, an operand or parenthesis all separated by a space, we can split these into a list and analyze and convert the expression from the list going left to right one single element (called token) at a time.

General Infix to Postfix Conversion using Stack

- If token is an operand, append it to the end of an output list
- If the token is an opening parenthesis, push it to the stack
- If the token is a closing parenthesis, pop elements from the stack until the corresponding opening parenthesis and append each operator to the end of the output list.
- If the token is an operator, check the top of the stack, as long as there are operators with higher precedence than the token, pop them out and append them to the output list, then push the token to the stack.
- At the end empty stack into the output

Input string
"A * B + C * D"
Input list
[A,*,B,+,C,*,D]



Output list
[A,B,*,C,D,*,+]

Infix to Postfix in Python

```
def infixToPostfix(infixexpr):
    alphaOperand="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    digitOperand="0123456789"
    prec = {}
    prec["*"] = 3
    prec["/"] = 3
    prec["+"] = 2
    prec["-"] = 2
    prec["("] = 1
    opStack = Stack()
    postfixList = []
    tokenList = infixexpr.split()
```

```
    for token in tokenList:
        if token in alphaOperand or token in digitOperand:
            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)
```

Traverse input and for each token

- if operand, append to output
- if “(“ push to stack
- if “)”, pop and output until “(“
- If operator pop all operators with more precedence and append to output, then push operator

At the end empty stack into output

More complicated example

Token	Action	Output	Operator Stack	Notes
3	Add token to output	3		
+	Push token to stack	3	+	
4	Add token to output	3 4	+	
*	Push token to stack	3 4	* +	* has higher precedence than +
2	Add token to output	3 4 2	* +	
/	Pop stack to output	3 4 2 *	+	/ and * have same precedence
	Push token to stack	3 4 2 *	/ +	/ has higher precedence than +
(	Push token to stack	3 4 2 *	(/ +	
5	Add token to output	3 4 2 * 5	(/ +	
−	Push token to stack	3 4 2 * 5	− (/ +	
1	Add token to output	3 4 2 * 5 1	− (/ +	
)	Pop stack to output	3 4 2 * 5 1 −	(/ +	Repeated until "(" found
	Pop stack	3 4 2 * 5 1 −	/ +	Discard matching parenthesis
+	Pop / from stack to output	3 4 2 * 5 1 − /	+	+ has lower precedence than /
	Push token to stack	3 4 2 * 5 1 − / +	+	and equal precedence to +
(	Push token to stack	3 4 2 * 5 1 − / +	(+	
2	Add token to output	3 4 2 * 5 1 − / + 2	(+	
+	Push token to stack	3 4 2 * 5 1 − / + 2	+ (+	
3	Add token to output	3 4 2 * 5 1 − / + 2 3	+ (+	
)	Pop stack to output	3 4 2 * 5 1 − / + 2 3 +	(+	
		3 4 2 * 5 1 − / + 2 3 +	+	Repeated until "(" found
end	Pop entire stack to output	**3 4 2 * 5 1 − / + 2 3 + +**		Stack is now empty

Input:
3 + 4 * 2 / (5 - 1) + (2 + 3)

operator	precedence
*	3
/	3
+	2
-	2
(	1

output:
3 4 2 * 5 1 - / + 2 3 + +

Evaluation

3 + 4 * 2 / (5 - 1) + (2 + 3)
3 + 4 * 2 / 4 + 5
3 + 8 / 4 + 5
3 + 2 + 5 = 10

Traverse input and for each token
- if operand, append to output
- if "(" push to stack
- if ")", pop and output until "("
- if operator pop all operators with more precedence and append to output, then push operator

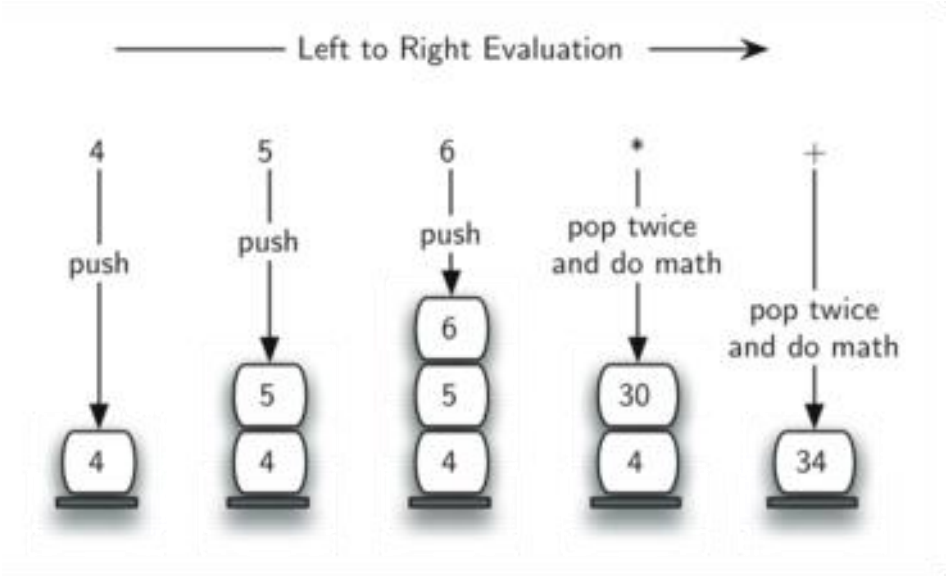
At the end empty stack into output

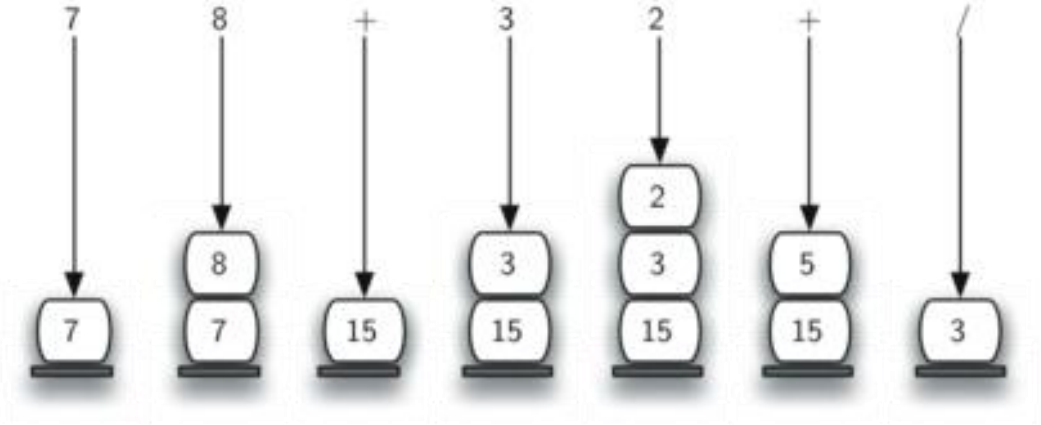
3 4 2 * 5 1 - / + 2 3 + +
3 8 5 1 - / + 2 3 + +
3 8 4 / + 2 3 + +
3 2 + 2 3 + +
5 2 3 + +
5 5 +
10

Postfix Evaluation using Stack

- In postfix notation, each operator applies to the previous two operands
- We should store the operands in a stack
- Traverse the expression from left to right and push operands on the stack and each time we encounter and operator, we pop two operands, apply the operator on them and push the result on the stack. The final result is on top of the stack.

Examples of Evaluation





Where is the result at the end?

On top of the Stack

Python Implementation

```
from pythonds.basic.stack import Stack

def postfixEval(postfixExpr):
    operandStack = Stack()
    tokenList = postfixExpr.split()

    for token in tokenList:
        if token in "0123456789":
            operandStack.push(int(token))
        else:
            operand2 = operandStack.pop()
            operand1 = operandStack.pop()
            result = doMath(token,operand1,operand2)
            operandStack.push(result)
    return operandStack.pop()
```

```
def doMath(op, op1, op2):
    if op == "*":
        return op1 * op2
    elif op == "/":
        return op1 / op2
    elif op == "+":
        return op1 + op2
    else:
        return op1 - op2
```

3 + 4 * 2 / (5 - 1) + (2 + 3)

```
print(postfixEval("3 4 2 * 5 1 - / + 2 3 + +"))
10
```