LT 10b – Stack ADT

Python Lists : Operations (Recap)

```
a = list(range(5))          → [0,1,2,3,4]
a[4]                        → 4
b = a[2:]                   → [2,3,4]
c = list(range(5, 10))      → [5,6,7,8,9]
d = a + c                   → [0,1,2,3,4,5,6,7,8,9]
```

Python Lists : Operations (Recap)

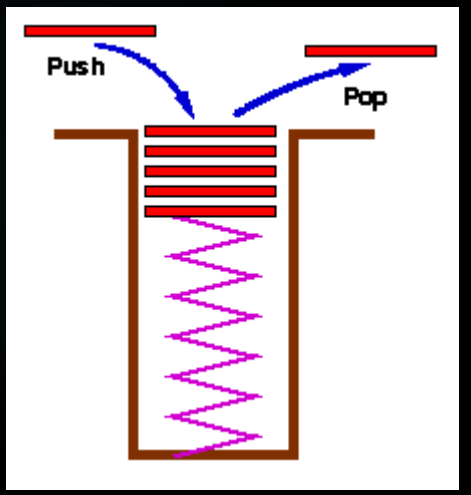
```
lst = [1, 2, 3,4]


lst.append(5)
lst                  → [1,2,3,4,5]
lst.pop()            → 5
lst                  → [1,2,3,4]
len(lst)             → 4
lst[len(lst)-1]      → 4
lst.clear()          → []
```

What are Linear Structures?

- Linear structures can be thought of as having two ends.
- Sometimes these ends are referred to as the “left” and the “right” or in some cases the “front” and the “rear.”
- You could also call them the “top” and the “bottom.”

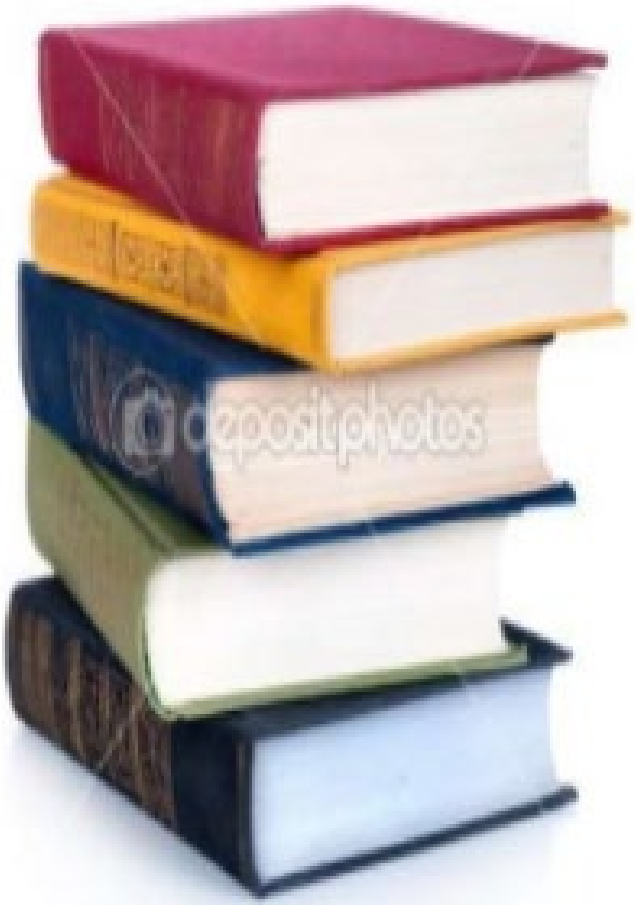
What is a Stack ?

- A stack is a linear data structure with the last-in-first-out (LIFO) property.



- This means that the last thing we added (**pushed**) is the first thing that will get **popped** out.

Examples of Stack :



Stack ADT : Constructor, Setters (Modifiers), Getter (Accessor) (LT10b Question 1-4)

`make_stack(seq)` : create a stack with the sequence

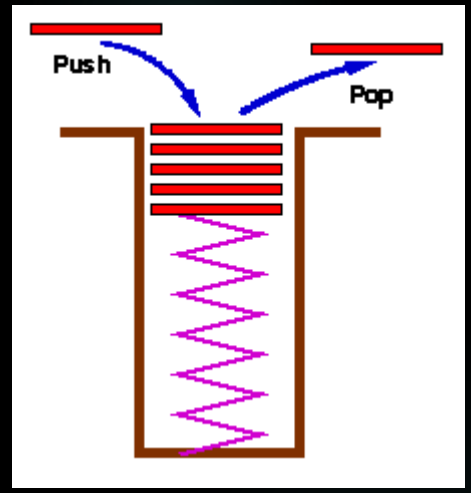
`make_empty_stack()` : returns a new, empty stack

`push(stack,item)` : adds item to stack

`pop(stack)` : removes the most recently added item from stack and returns it

`peek(stack)` : returns the top object of the stack but does not remove it from the stack

Stack operations




```
s = make_stack()

pop(s)          → None  : empty stack, nothing to pop
push(s, 5)
push(s, 3)
pop(s)          → 3
pop(s)          → 5
is_empty(s)     → True
```

LT 10b Stack ADT – Applications

Applications of Stack (T10b Question 1-3)

- Reversing a sequence
- Checking for balanced brackets or parentheses
- Postfix Notation for Arithmetic Calculation

Reversing a Sequence (Tutorial Question 1)

Checking for Balanced Braces

(Tutorial Question 2)

([ ] ({ ( ) })) is balanced and (({ }[ )]) is not.

Simple counting is not enough to check for balance.

You can do it with a stack. Going from left to right …

1. If you see a '(', '[', or '{', push it onto the stack;
2. If you see a ')', ']', or '}', pop the stack and check whether you got the corresponding '(', '[', or '{'.

When you reach the end, check that the stack is empty.

Arithmetic Notation for Calculation

(Tutorial Question 3)

- Infix notation : the operator is written in between the operands.

  Eg : A + B

- Prefix notation : the operator is written before the operands.

  Eg : + A B

- Postfix notation : the operator is written after the operands.

  Eg : A B +

Examples of Postfix Notations :

Infix Notation	Postfix Notation
3 * 4 + 5	3 4 * 5 +
3 * (4 + 5) / 2	3 4 5 + * 2 /
(3 + 4) / (5 – 2)	3 4 + 5 2 – /
7 – (2 * 3 + 5) * (8 – 4 / 2)	7 2 3 * 5 + 8 4 2 / – * –
3 – 2 + 1	3 2 – 1 +

Computing with Postfix Notation

(Tutorial Question 3)

- 3 * 4 + 5 = **3 4 * 5 +**

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

The End

Convert Infix to Postfix Notation

(Side Quest - Question 1)

1. Write a code to convert a simple infix notation involving only + and -.

For example : convert_infix_to_postfix('3+4') -> '34+'

convert_infix_to_postfix('3+4-2') -> '342-+'

2. Write a code to convert an infix notation involving all the +, -, * and /. Note that we would perform * and / first before + and -.

For example : convert_infix_to_postfix('3*4-2') -> '34*2-'

convert_infix_to_postfix('1+2*3') -> '123*+'

3. Write a code to convert an infix notation involving brackets and operators, +, -, * and /. Note that we would perform all the operations within the brackets first.

For example : convert_infix_to_postfix('(1+2)*3') -> '12+3*'

convert_infix_to_postfix('(1+2)*(3+4)') -> '12+34+*'

Note : You may assume that the operands are 1-digit integers and only binary operators + , - , * , / are used in the string.