# Stack

Agenda:

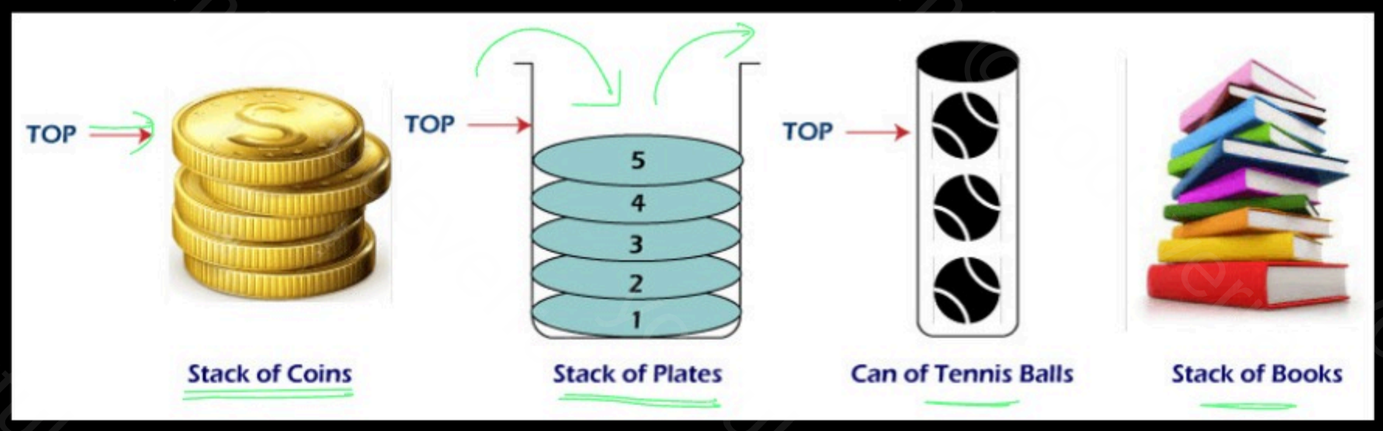
1. What is a stack?
2. Comparison with other data structures
3. Common Operations on stack
4. Applications of Stack
5. Python implementations using list, deque and linked list
6. Multiple Choice Questions (MCQs)
7. Monotonic Stack
8. Practice Problems and Leetcode problems

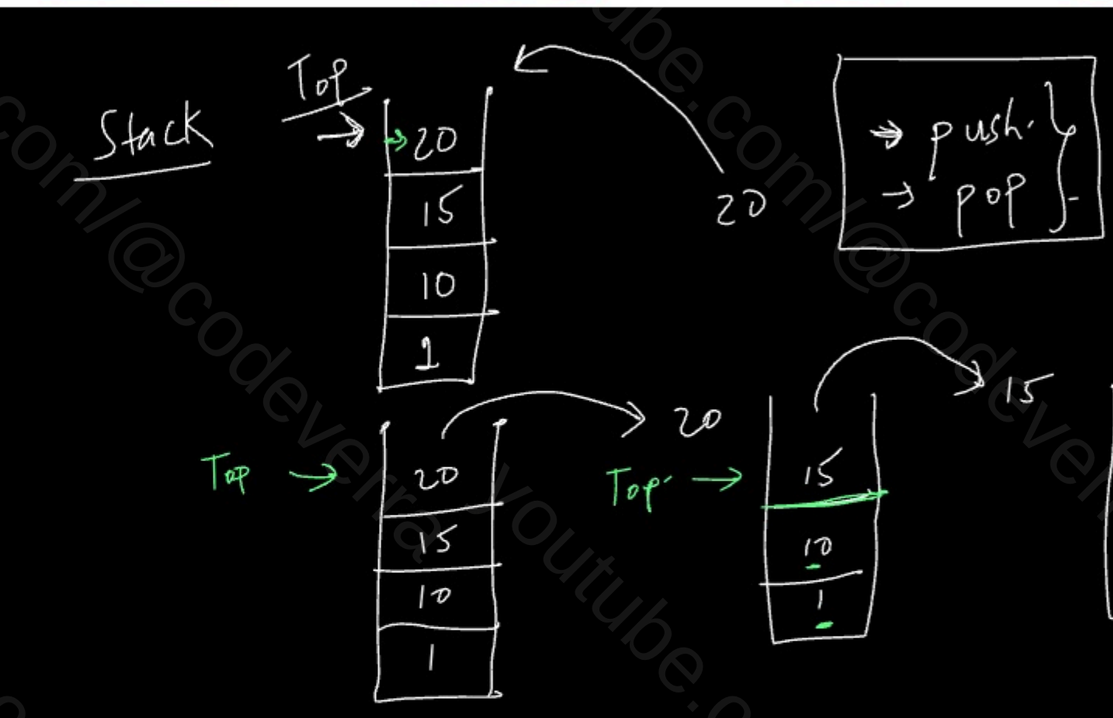
## Stack

- Stack is linear data structure that follows LIFO. (Last In First out).
- Stack is "Abstract" Data Type (ADT) that follows LIFO.

- Collection of items
- Linear
- LIFO.
- push and pop

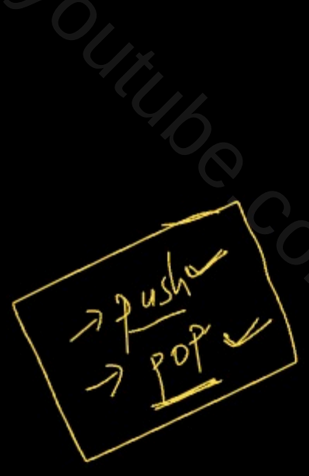
- ADT → Contract / Behaviour / Interface.
  - LIFO.
  - push, POP.
- Data Structure → Implementation

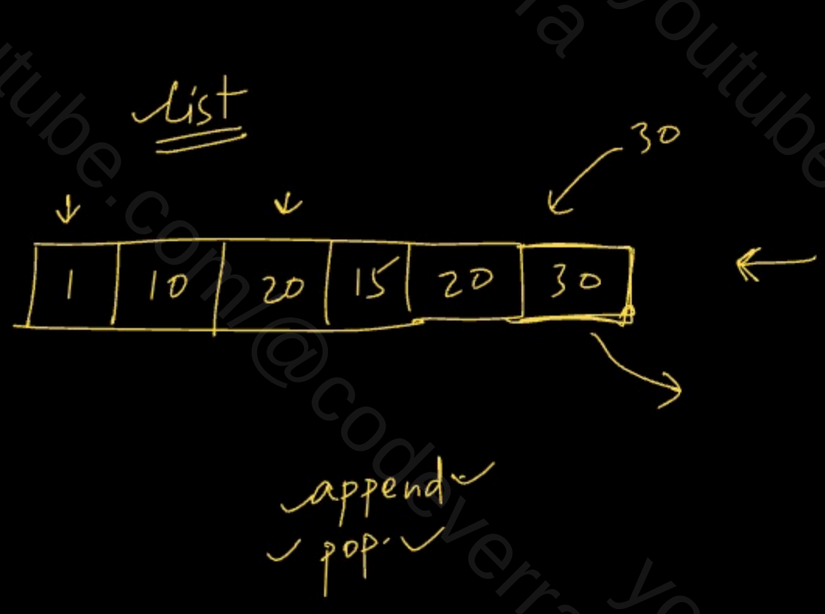



✓ Last in First out

↳ Last element added → first to be removed.

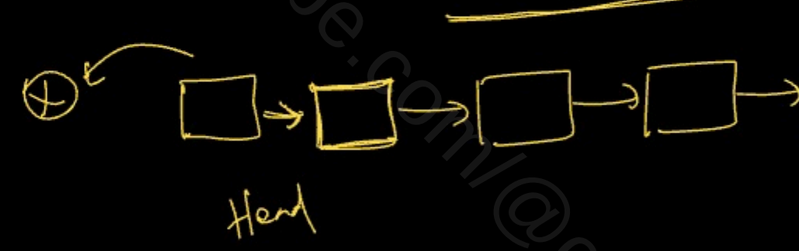



Time Complexity

push → O(1)

pop → O(1)




Implementation of Stack →

1. List
2. Linked List
3. Deque (Double Ended Queue)
   ↳ (Doubly Linked List)

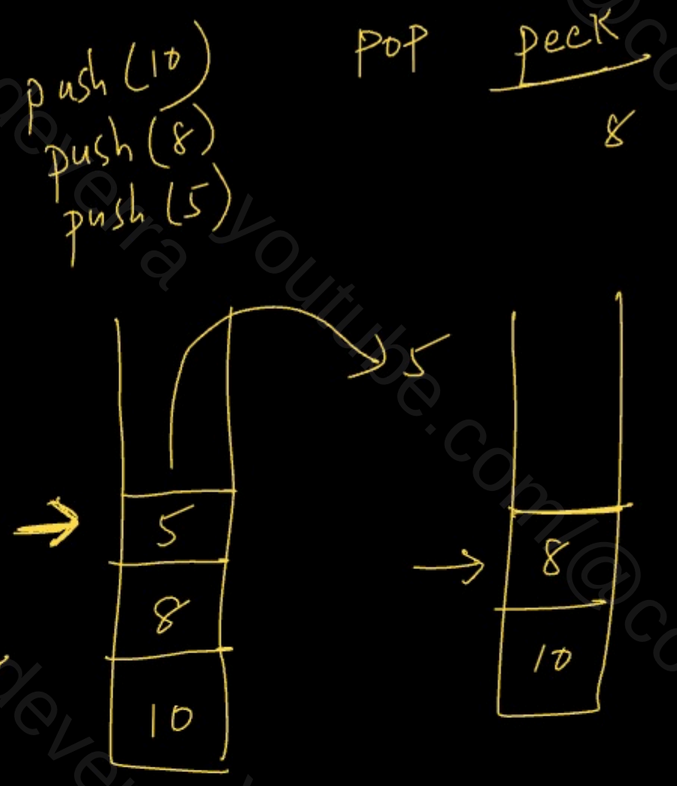
ADT

LIFO

{ push
pop
peek }

## Common operations on Stack

1. push(x) → Insert element at the top.
2. pop() → Remove and return top element.
3. peek() / top() → Read top without removing.
4. is_empty() → Check if stack has no elements
5. size() → Number of elements in stack.




## Python Implementation

```
class Stack:
    def push(self):
        ...
    def pop(self):
        ...
```

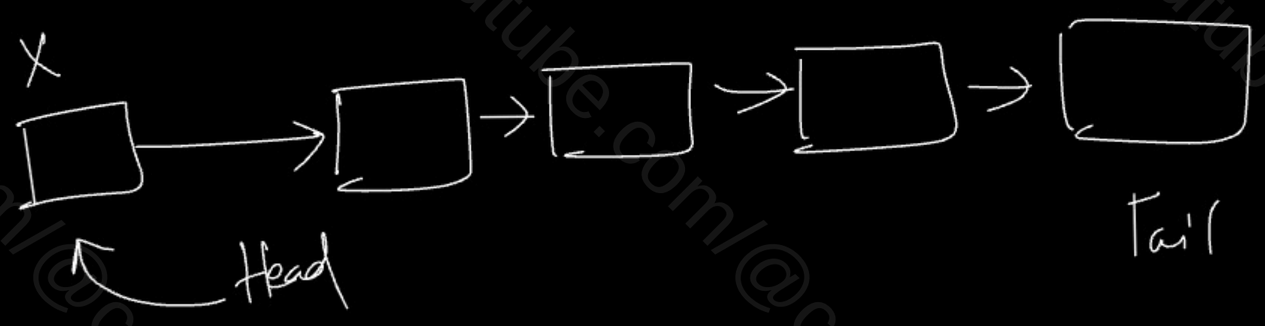
- push
- pop
- peek
- size
- empty

→ List
→ Linked List
→ deque

```
x = [10, 8, 5]
x.pop()
[10, 8]
x[-1]
```

## Stack using Linked List

act as top of stack




### push

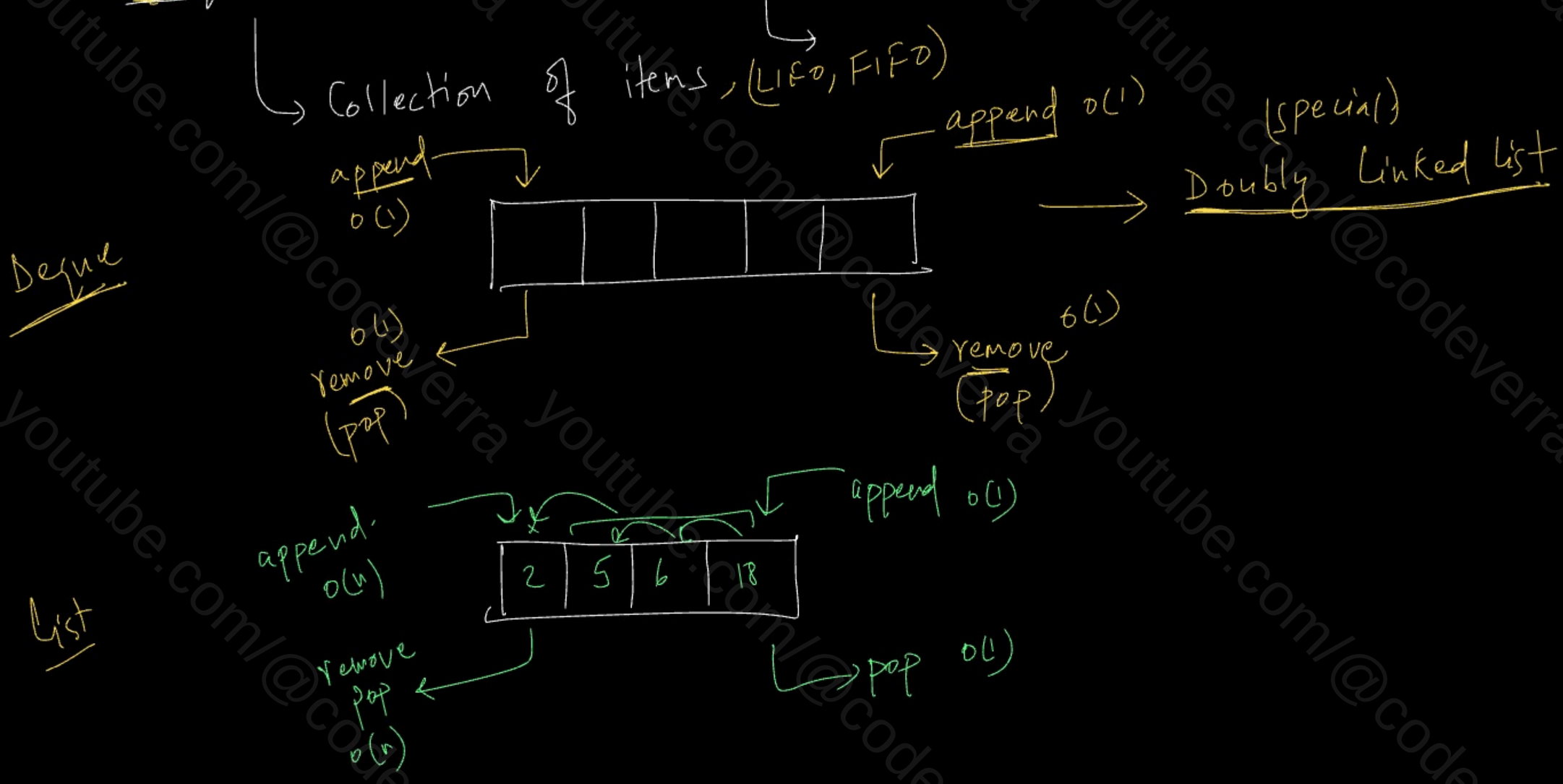
### pop

```
val = head.data
head = head.next
```

# Stack using Deque

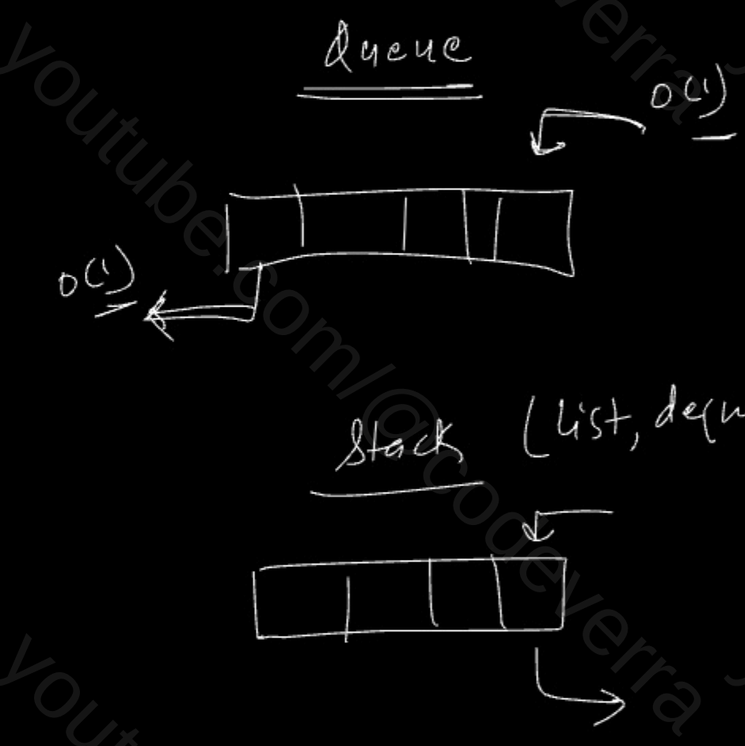
Deque → Double Ended Queue

→ Collection of items, (LIFO, FIFO)




Deque → append, pop, appendleft, popleft → O(1)

access → O(n)

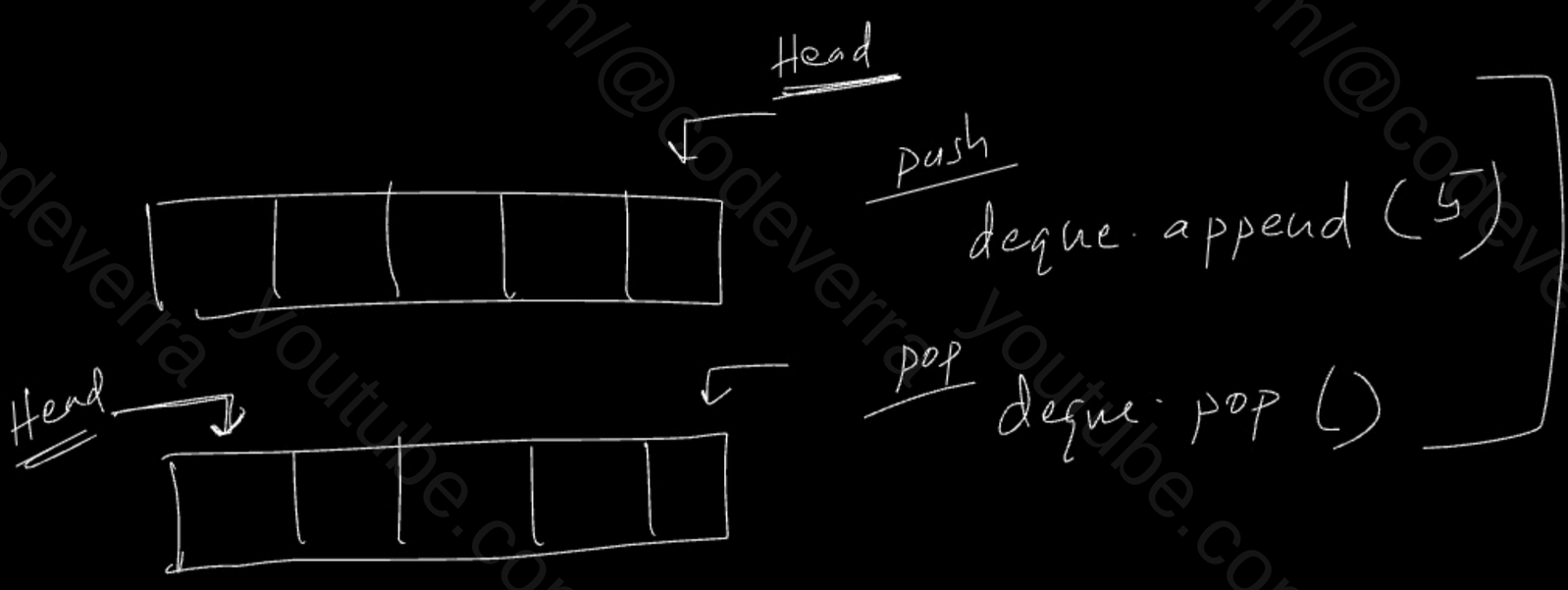



List → accessing → O(1)

append, pop → O(1)

Collections module → Deque

Deque → Stack, Queue



Time complexity of common operations on Stack:

- Push → O(1)
- Pop → O(1)
- Peek → O(1)
- Search (not typical) → O(n)
- Space → O(n)

From Wikipedia:

In computer science, a **stack** is an abstract data type that serves as a collection of elements with two main operations:

- **Push**, which adds an element to the collection, and
- **Pop**, which removes the most recently added element.

Additionally, a peek operation can, without modifying the stack, return the value of the last element added (the item at the top of the stack). The name stack is an analogy to a set of physical items stacked one atop another, such as a stack of plates.
The order in which an element added to or removed from a stack is described as **last in, first out**, referred to by the acronym **LIFO**.[nb 1] As with a stack of physical objects, this structure makes it easy to take an item off the top of the stack, but accessing a datum deeper in the stack may require removing multiple other items first.[1]
Considered a sequential collection, a stack has one end which is the only position at which the push and pop operations may occur, the top of the stack, and is fixed at the other end, the bottom. A stack may be implemented as, for example, a singly linked list with a pointer to the top element.
A stack may be implemented to have a bounded capacity. If the stack is full and does not contain enough space to accept another element, the stack is in a state of stack overflow.

# Problems on Stack

## ① Reverse a String

String = "codeverra" ←

Output = "arreveDoc"

✔ Constraints
0

slicing method → ✔ string[::-1]

" " →

stack method →

String = "codeverra" → O(n)

"arreveDoc"

Stack =


✔ ['a', 'r', ....., 'c'] O(1)

''.join([ ])

✔ Time Complexity: O(n)

✔ Space Complexity O(n)

① a empty stack
② iterate on string & add to stack
③ pop from stack & append to result
④ result list → string
⑤ return string

input → string

TC : O(n)
SC : O(n)

✔ slicing approach
stack
string append

→ TC O(n) SC O(n)

input → list

✔ Two pointer approach

→ string = ['c', 'o', 'd', 'e']

['e', 'd', 'o', 'c']

pseudo code

```
L, R = 0, len(string)-1
while L < R:
    string[L], string[R] = string[R], string[L]
    L = L+1
    R = R-1
return string
```

TC → O(n)
SC → O(1)
↳ inplace

string = 'apple' → inefficient

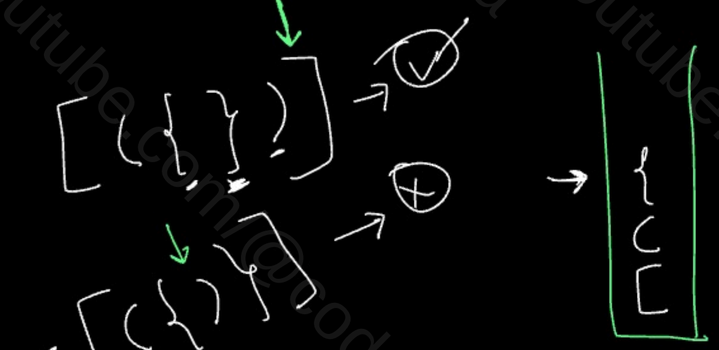
```
result = ""
for char in string:
    result = char + result
```

['a' + 'p' + 'p' + 'l' + 'e'] → O(n²)

→ in place operation
SC → O(1)

# ② Valid Paranthesis

→ open count = closed count
→ open should be closed
↳ in order




open = "[{("
close = "]})"

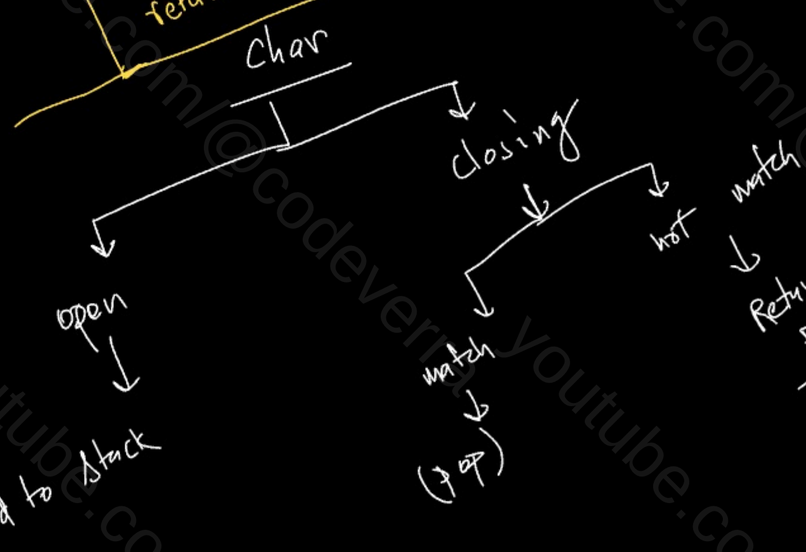
map = { "]" : "[",
"}" : "{",
")" : "(" }

## Pseudo code

```
stack = [ ]
for char in s:
    if char in map: # closing.
        if not stack or stack[-1] ≠ map[char]:
            return False
        else: stack.pop()
    else: stack.append(char)
return len(stack) == 0
```

Valid paranthesis

[()][ → ✗



# ③ Min Stack

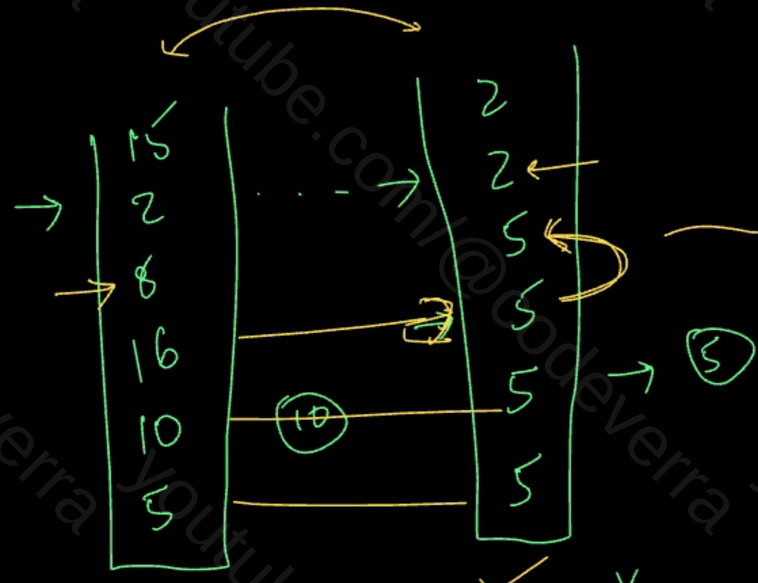
- Values = 5, 10, 16, 8, 2, 15

push → O(1)
pop → O(1)
seek / top → O(1)

O(n)

O(1)


2 stack




minstack.append(minstack[-1])

{ actual stack
  helper stack → track min

to track min value

5, 10, 16

5, 5, 5

min_value → O(1)

{ Space complexity ↑ extra stacks
  Time complexity ↓

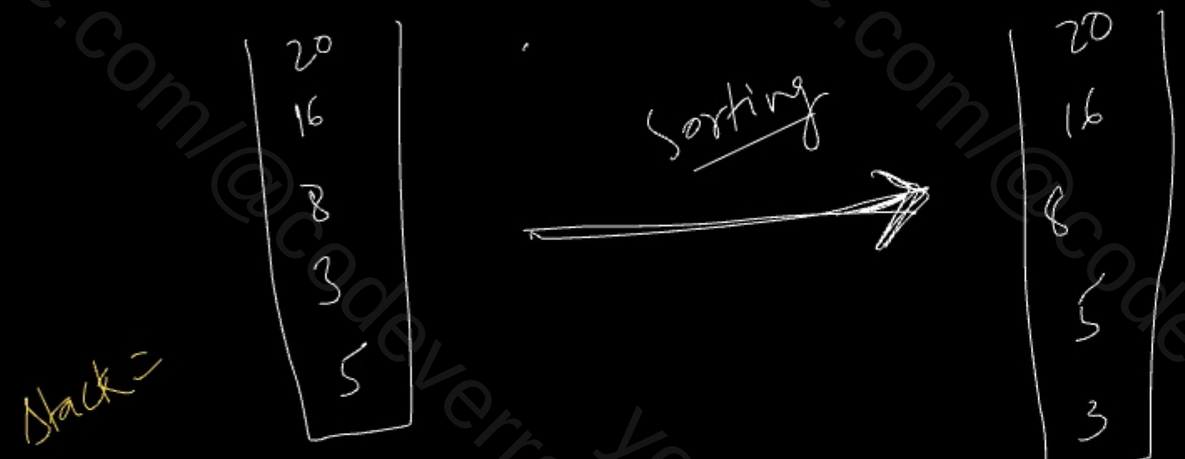
push(val)

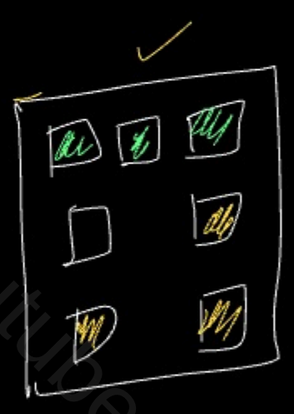
main stack → push
min stack → if not stack or val ≤ minstack[-1]
  ✓ minstack.append(val)
  ✓ else minstack.append(minstack[-1])

# ④ Sorting a Stack

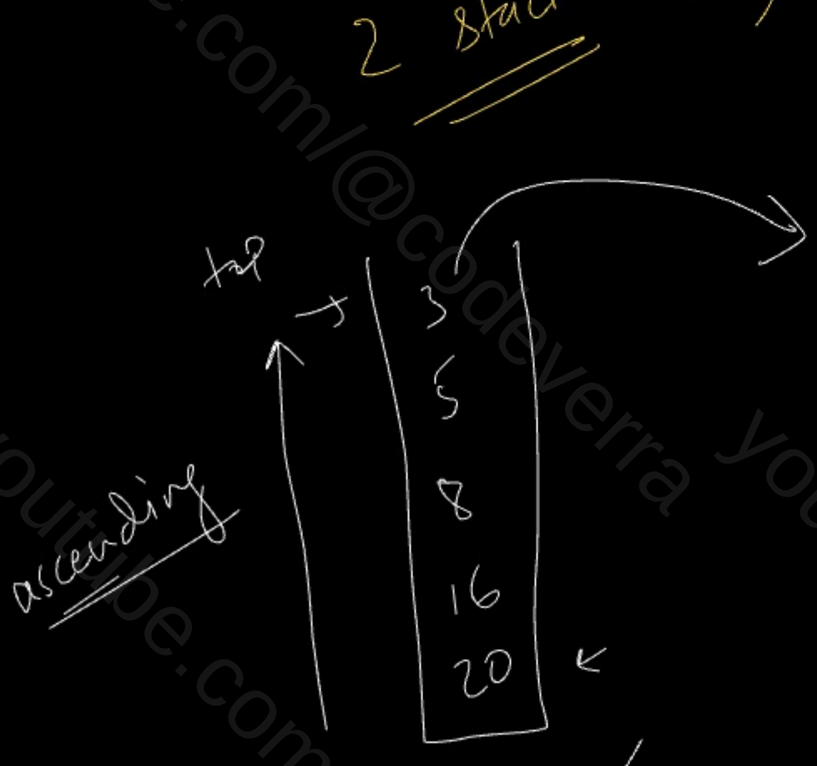
↳ stack is preferred for sorting (inefficient) → $O(n^2)$

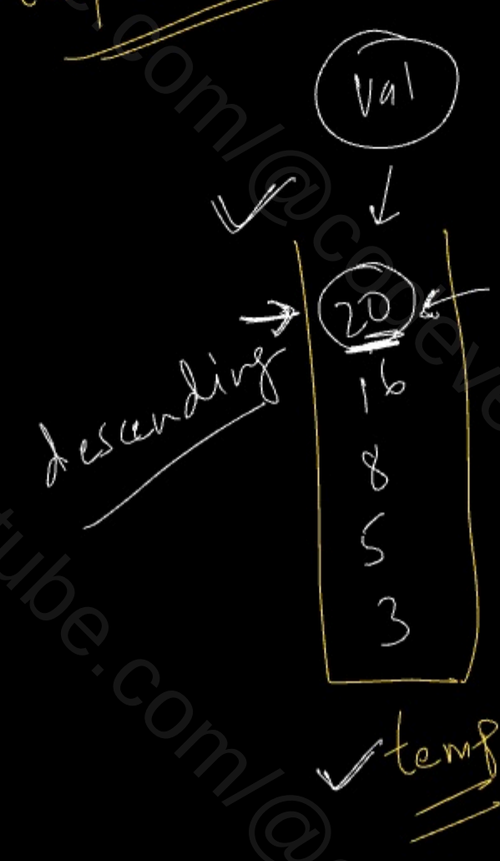
- $O(n)$
- $O(1)$
- $O(n^2)$ ←



2 stack → temp stack



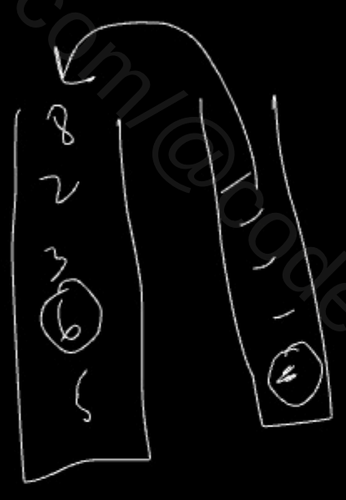




peek
push
pop

Space Complexity → $O(n)$

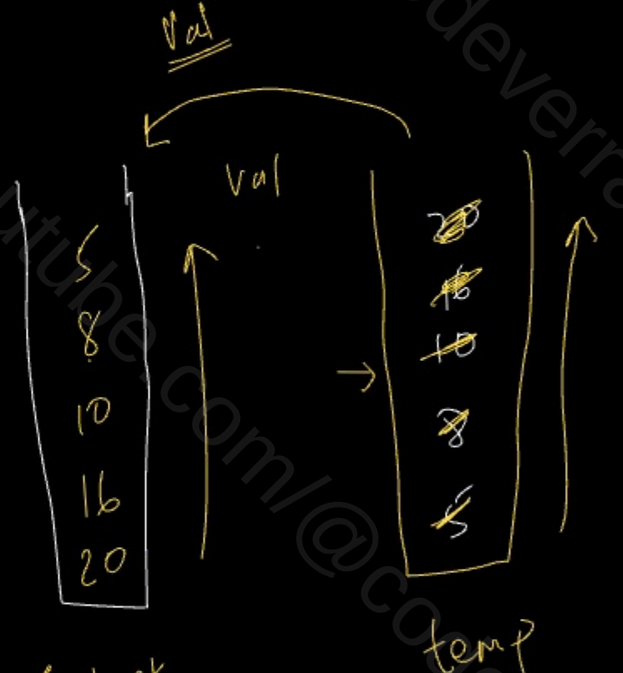
Time Complexity → $O(n^2)$ ←

1 element → n operations
n element → $n^2$ operations
→ inefficient

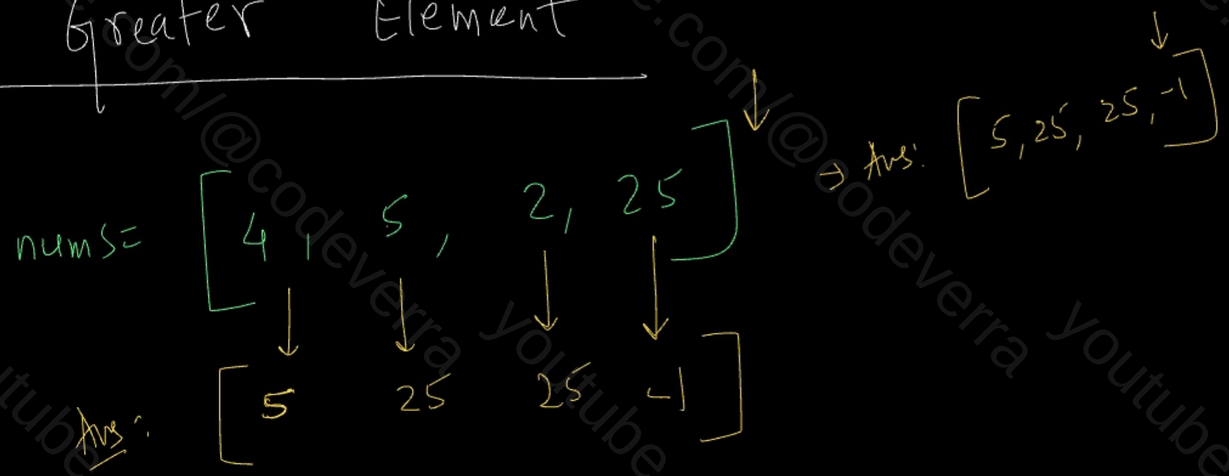


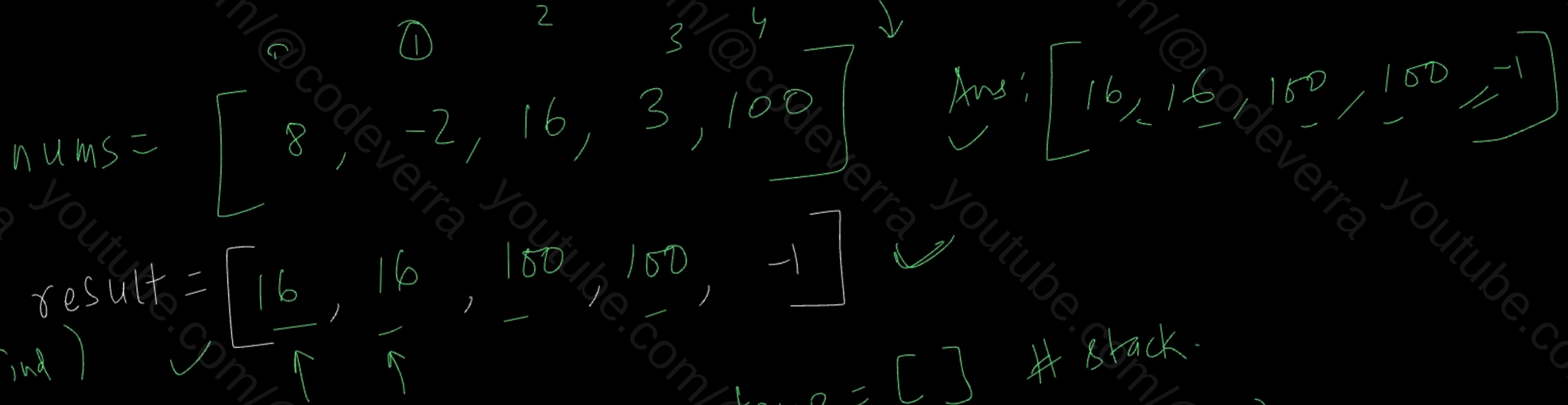

## Pseudo Code

```
while stack:
    val = stack.pop()
    while temp & val < temp[-1]:
        stack.append(temp.pop())
    temp.append(val)
while temp:
    stack.append(temp.pop)
```



# (5) Next Greater Element





Time Complexity
↳ O(n)

Next Greater Element

```
temp = [ ]   # stack.
result = [-1] * lens(nums)
for (ind, val) in enumerate (nums):
    while temp & val > temp[-1][1]:
        i, v = temp.pop()
        result[i] = val
    temp.append ((val, ind))
return result
```

# Monotonic Stack (Decreasing)

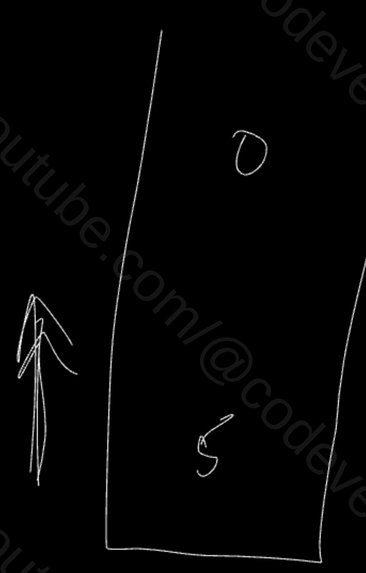
$O(n)$

nums = [4, 2, 1, 2, 5]

[5, 5, 2, 5, -1]

(5)

Monotonic decreasing stack




→ Next greater element

Next smaller element

↳ Monotonic Increasing stack

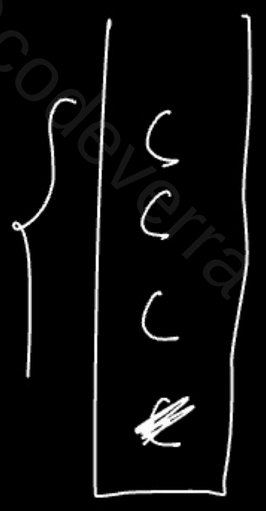
# ⑥ Minimum add for Valid paranthesis

s = "(()"

s = "())(((" → ③

Ans: 1.

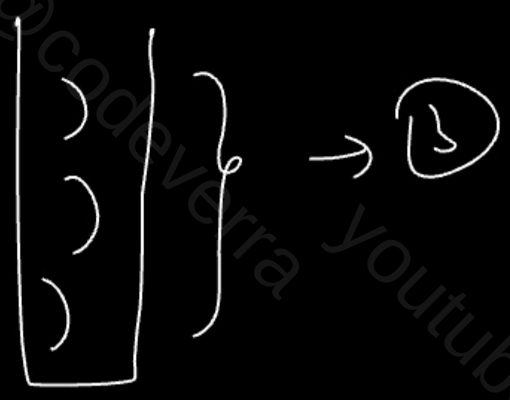
Stack



→ Ans = 3

s = "))))"

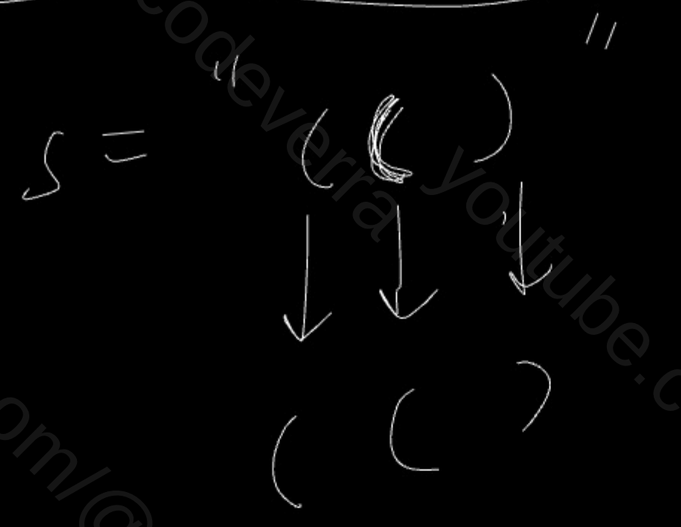
((()))



→ ①

optimization from SC O(n) → O(1)

s = "(())"



open

close

s = "(())(())"



```
{ open = ++
  Close = ++
  return open + close
```

→ ②