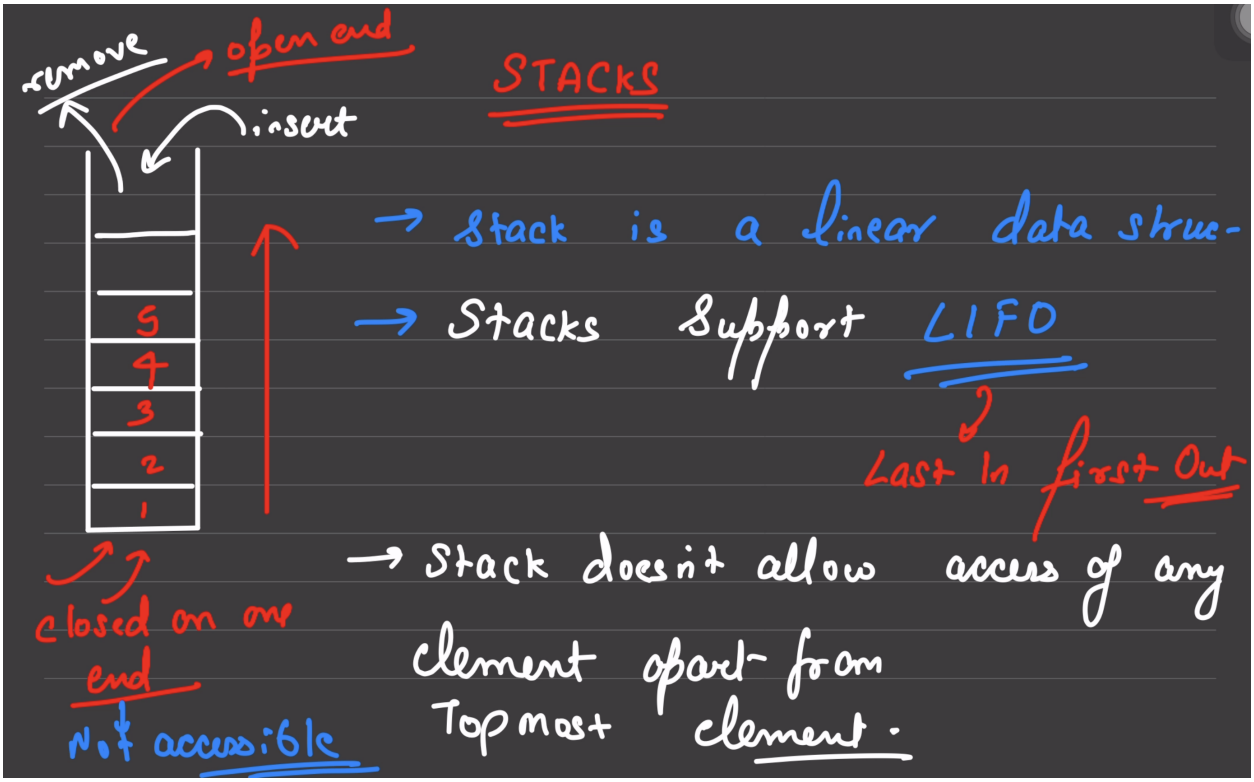
Stacks

Created	@August 12, 2022 8:38 PM
Class	
Type	
Materials	
Reviewed	☐



- Stacks are linear data structure.
- Stack as a data structure supports `LIFO` i.e. `Last In First Out`
- Stack allows access from one side of the structure only as the other side is closed that means we cannot even insert or remove an element from the close side.
- Stack doesn't allow random access of all the elements like arrays, we can at any point of time only access the top most element or we can say the element added at

the last only. So if we even want to remove an element we can only remove the top most element.

- In terms of stacks the insert operation is called as `PUSH` and removal operation is called as `POP`

Advantages and Use cases of Stacks:

- Stacks are present in the memory of a process for implementation of Call Stacks (this is the same call stack we use for recursion and function calls)
- Apart from call stack we have Kernel Stack also for the process (this is mainly for context switching)
- Stacks are used for any use case where we have to remember how an entity originated or we can say for any child entity we want to remember the parent.
- Undo / Redo operations in many softwares stacks are useful.
- Mobile devices having IOS or Android, then inside any app whenever we move to a new screen and come back to the original one, it’s due to stack based implementation.

How can we implement Stacks ?

There are multiple ways to implement stacks:

- We can implement Stacks using Linked List
- We can implement Stacks using Arrays
- We can implement Stacks using Queues

Stack using LL

```
class Node {
    constructor(d) {
        this.data = d;
        this.next = null;
```

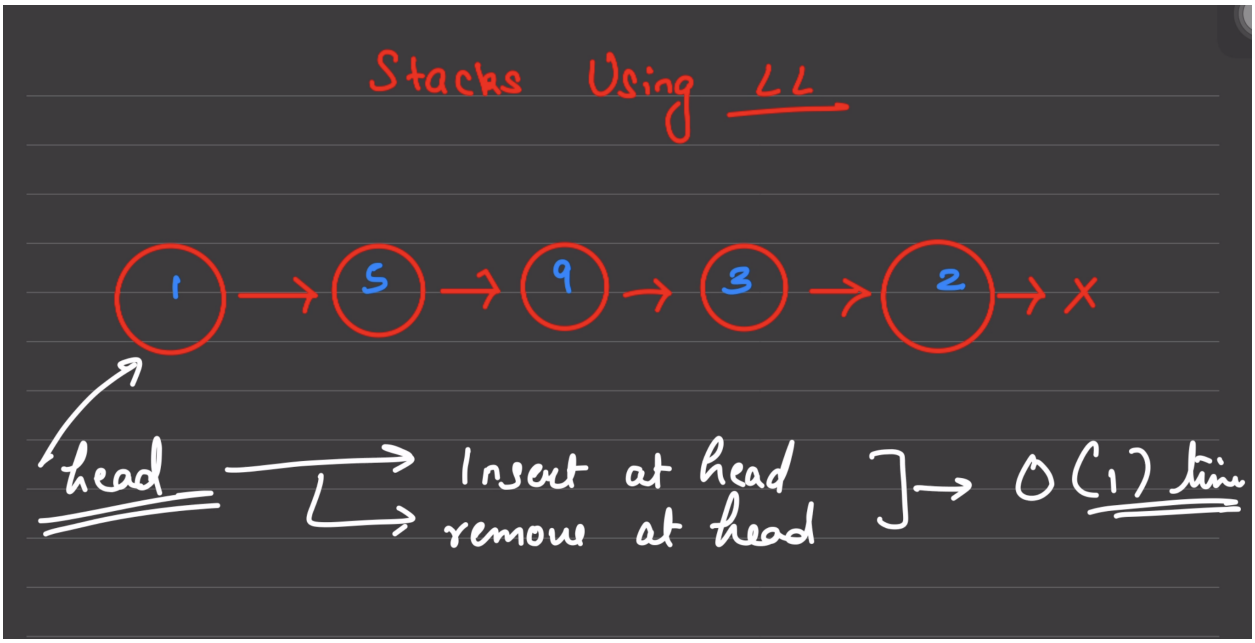
```
    }
}

class LinkedList {
  // singly
  constructor() {
    // when we initialise a new linked list head will be empty
    this.head = null;
    this.start = null; // temporary variable for problem solving
    this.flag = null; // temporary variable for problem solving
  }

  addAtHead(data) {
        /**
         * Time: O(1)
         * Space: O(1)
         */
    let newNode = new Node(data); // created a new node
    newNode.next = this.head; // set the next of new node to head
    this.head = newNode; // update the head to the new node
  }

  removeAtHead() {
        /**
         * Time: O(1)
         * Space: O(1)
         */
    if(this.head == null) return;
    let temp = this.head.next; // stored access to new head
    this.head.next = null; // de linked the old head
    this.head = temp; // updated the head
  }

    getHead() {
        /**
         * Time: O(1)
         */
        if(this.head != null) {
            // LL is not empty
            return this.head.data;
        }
        return undefined; // in case of empty LL
    }
}


class Stack {
    constructor() {
        this.ll = new LinkedList();
    }

    push(x) {
        // insert an element x in the stack
        /**
```

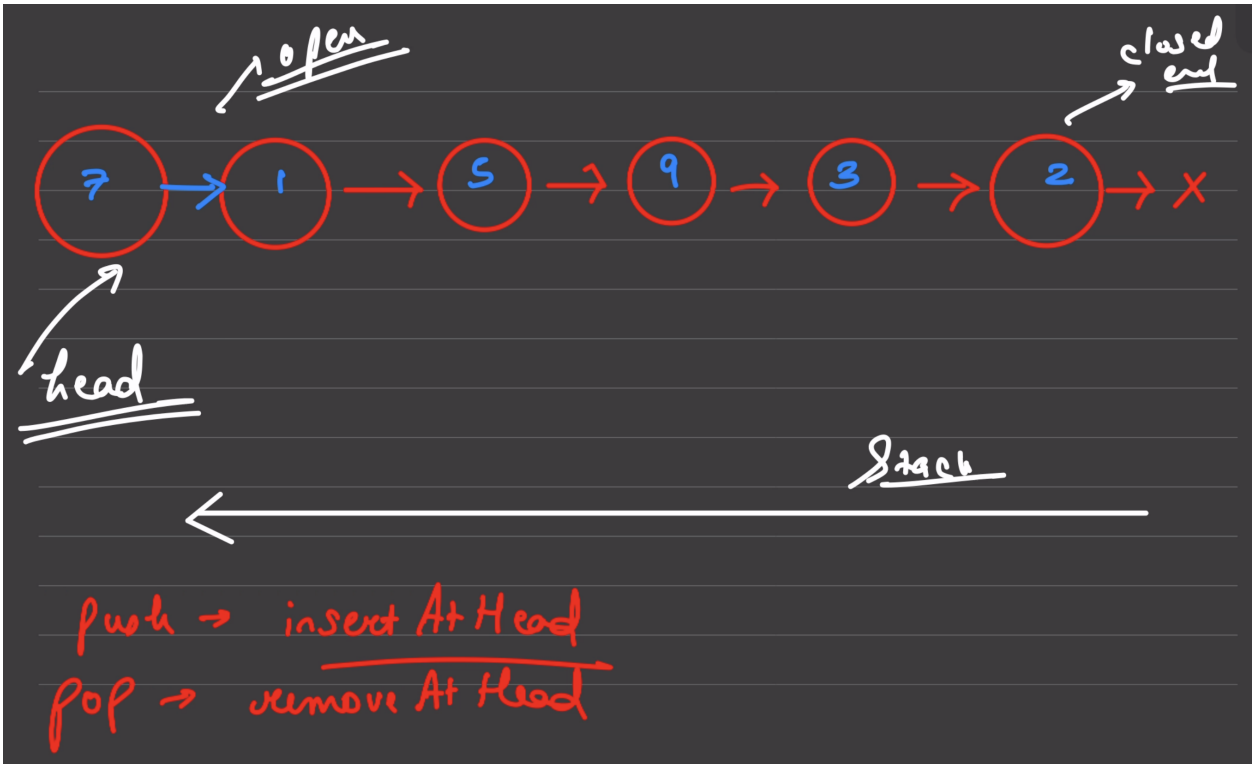
```
         * Time: O(1)
         */
        this.ll.addAtHead(x);
    }

    pop() {
        // remove the last added element from the stack
        /**
         * Time: O(1)
         */
        this.ll.removeAtHead();
    }

    top() {
        // this function will help us to get the topmost element of the stack
        /**
         * Time: O(1)
         */
        return this.ll.getHead();
    }
}

let st = new Stack();
st.push(1);
st.push(2);
st.push(3);
st.push(4);
console.log(st.top());
st.pop();
console.log(st.top());
```

For a Linked List, to be used to implement Stack like behaviour, we can use `insertAtHead` and `removeAtHead` functions in the LL class, as both of these functions work on one end of the LL i.e. head, and the other end which is the tail is not accessible. Also if in the Stack class we just expose the push and pop operation, then there is no way to access the elements of the stack which are apart from the topmost element.

Using this idea, push and pop and top, all three operations work in O(1) time which is highly efficient.

LL are one of the most efficient ways to implement stacks because, the other implementation using array can have the same time complexity, but LL are more efficient in terms of space than arrays. And the other implementation using queues, is not as optimised as LL or arrays in terms of time complexity.





Stack using arrays

```
class Stack {
    #arr;
```

```
    constructor() {
        this.#arr = [];
    }

    push(x) {
        /**
         * Time: O(1)
         */
        this.#arr.push(x); // add an element to the last
    }

    pop() {
        /**
         * Time: O(1)
         */
        this.#arr.pop(); // remove element from last index
    }

    top() {
        /**
         * Time: O(1)
         */
        return this.#arr[this.#arr.length - 1]; // return the element at the last index
    }
}

let st = new Stack();

st.push(10);
st.push(33);
st.push(1);
console.log(st.top());
st.pop();
console.log(st.top());
```

In order to implement stack using arrays, we can use the property in an array that the element added and removed from the last are efficiently added / removed in all of the languages generally. So we can say that the end of the array can act as the open end of stack and the start of the array can act as the closed end. We can add an element to the last of the array to mimic push pf stack and then remove the last element to mimic pop and access the last element to mimic top.

The only problem here is that if we will make our array public then in that case any end user can access the array outside the class and hence can access indexes randomly. And we don’t want that. So it’s better to make array as private member.

Problem 1:

Given a string of different type of opening and closing brackets.

- () → Parenthesis
- {} → Curly braces
- [] → Square brackets

We will be given a string that will be having all the types of brackets listed above.

Ex: `{(([]))}{}{{}}`

We have to check whether the given string of brackets is balanced or not ? What do we mean by saying balanced ? We mean that for every type of open bracket there should an equivalent closing bracket. There should not be any closing bracket without an opening bracket on the left of it and no opening bracket without a corresponding closing bracket on the right.

Examples:

- {}{} → Balanced
- {(())} → Balanced
-){}{(())} → Not balanced
-)(→ not balanced
- (())()() → Balanced
-)((())) → Not balanced
-)(())(→ not Balanced

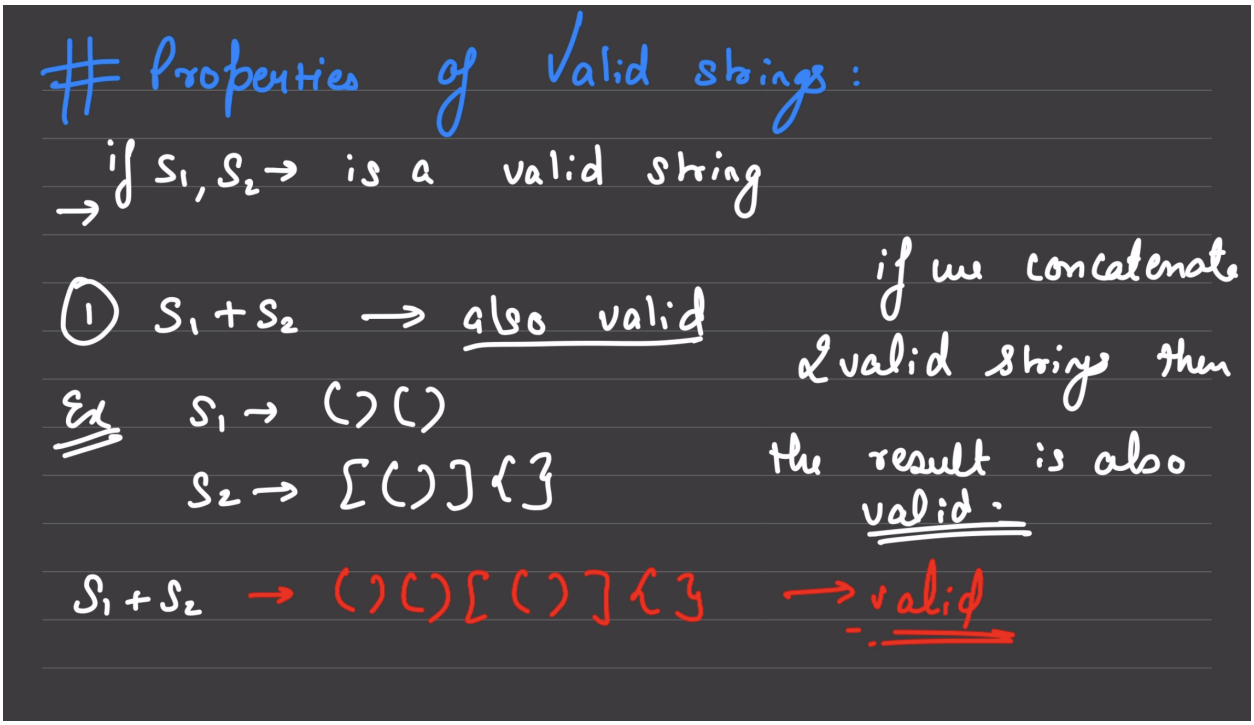
The length of the string will be n and `1 <= n <=` 10^7.

> Valid Parentheses - LeetCode
>
> Given a string s containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid. An input string is valid if: Open brackets must be closed by the same type of brackets. Open brackets must be closed in the correct order.
>
> https://leetcode.com/problems/valid-parentheses/

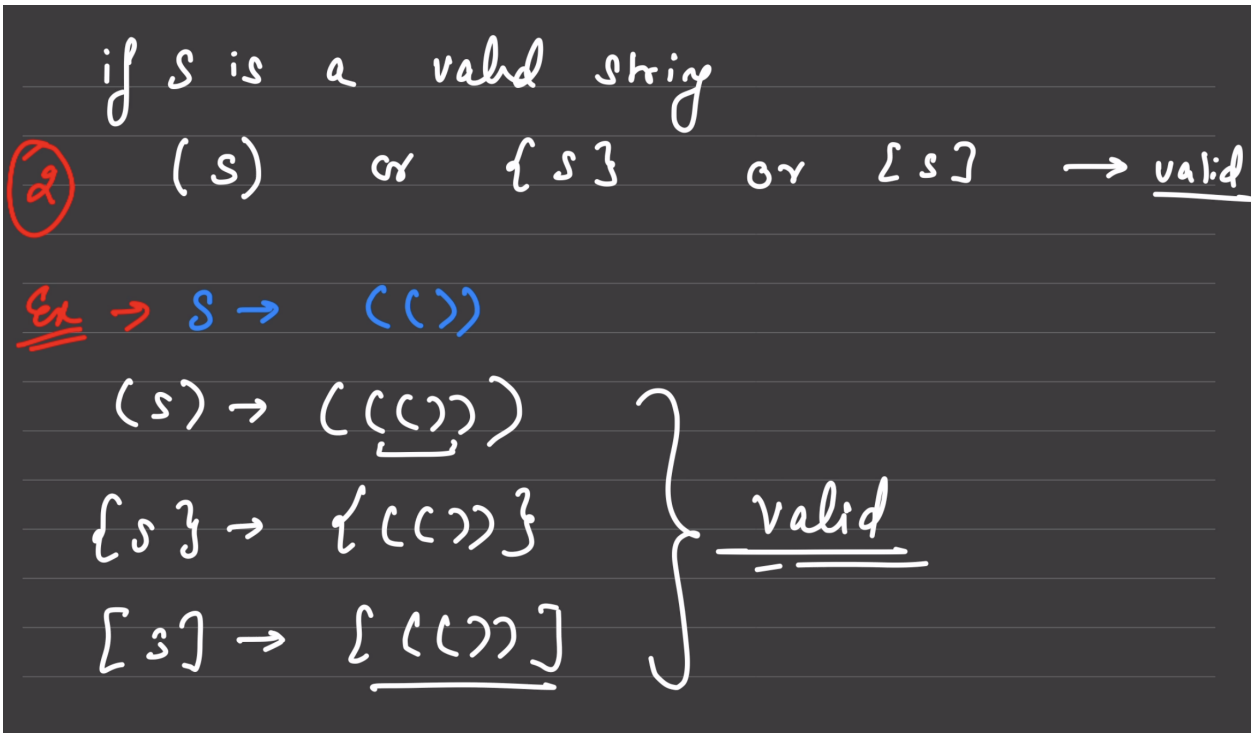
Solution

Every valid string will follow the following properties

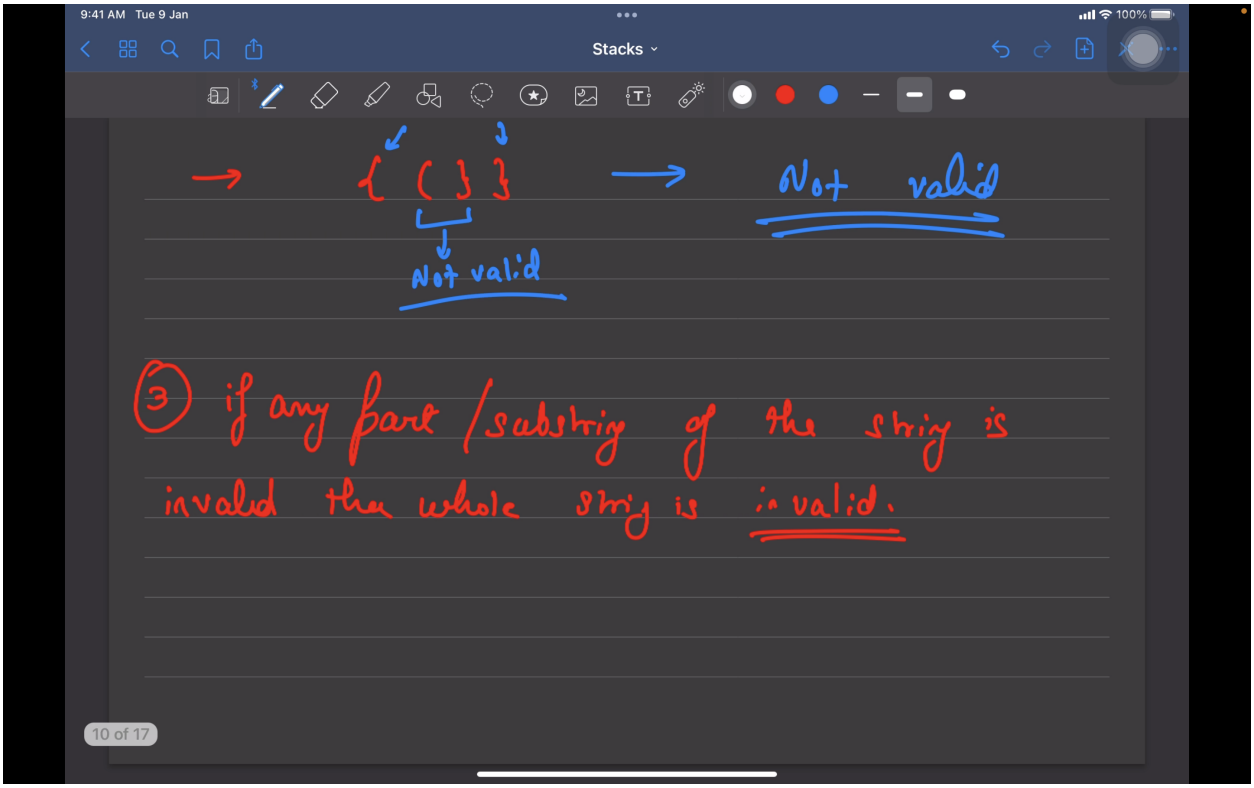
- If s1 and s2 are valid strings then s1 + s2 is also valid



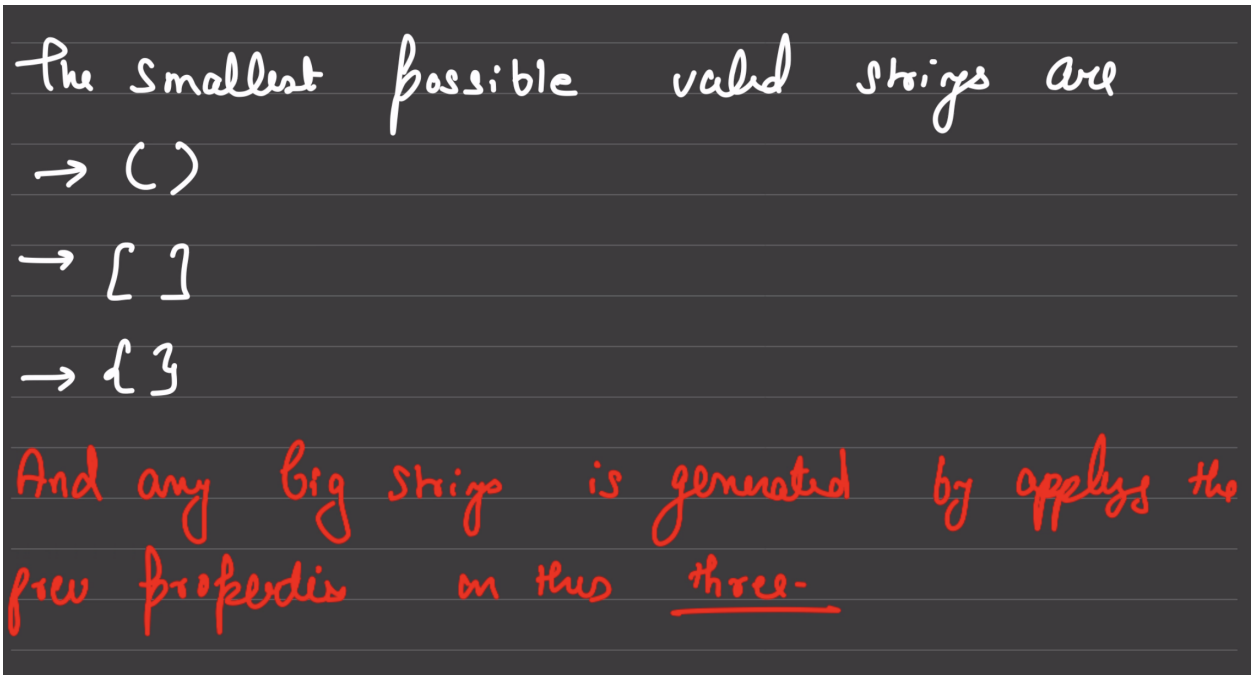
- If s is a valid string then if we enclose s with () or {} or [] then the resultant string will also be valid



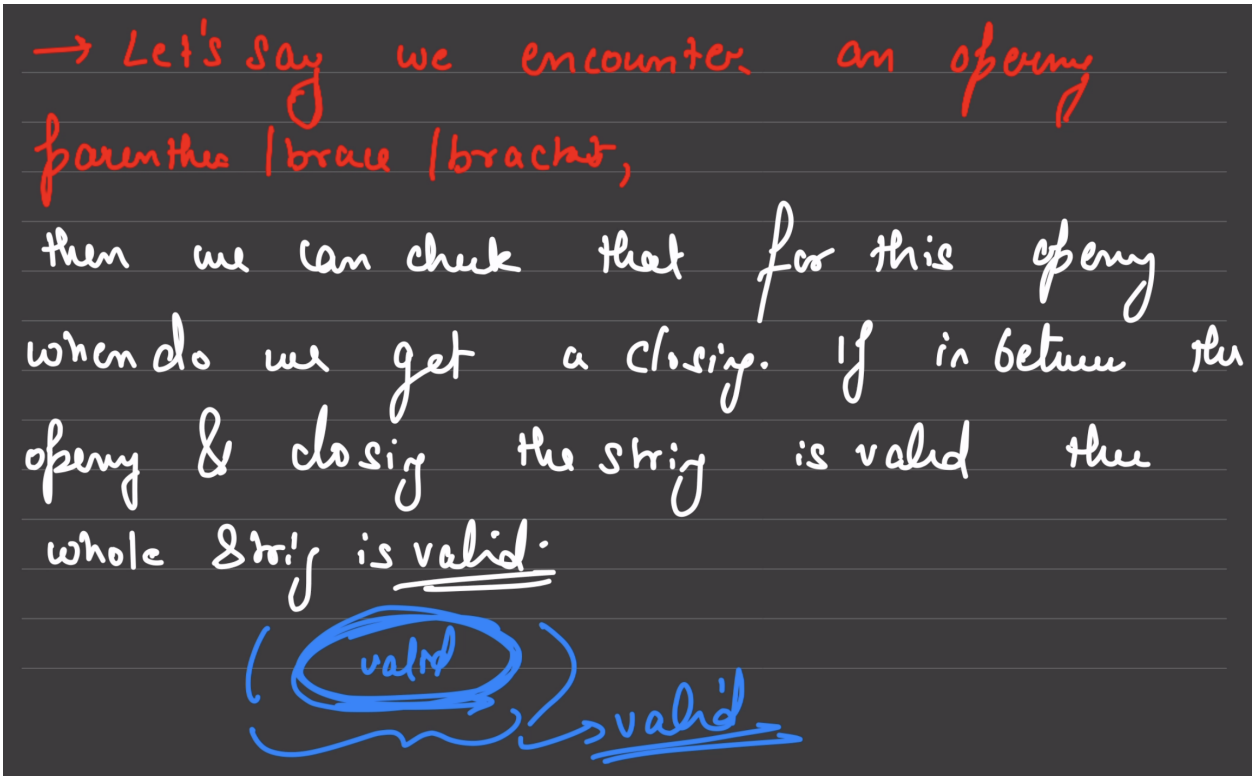
- If we find a closing parenthesis without an opening one then it makes the whole string valid

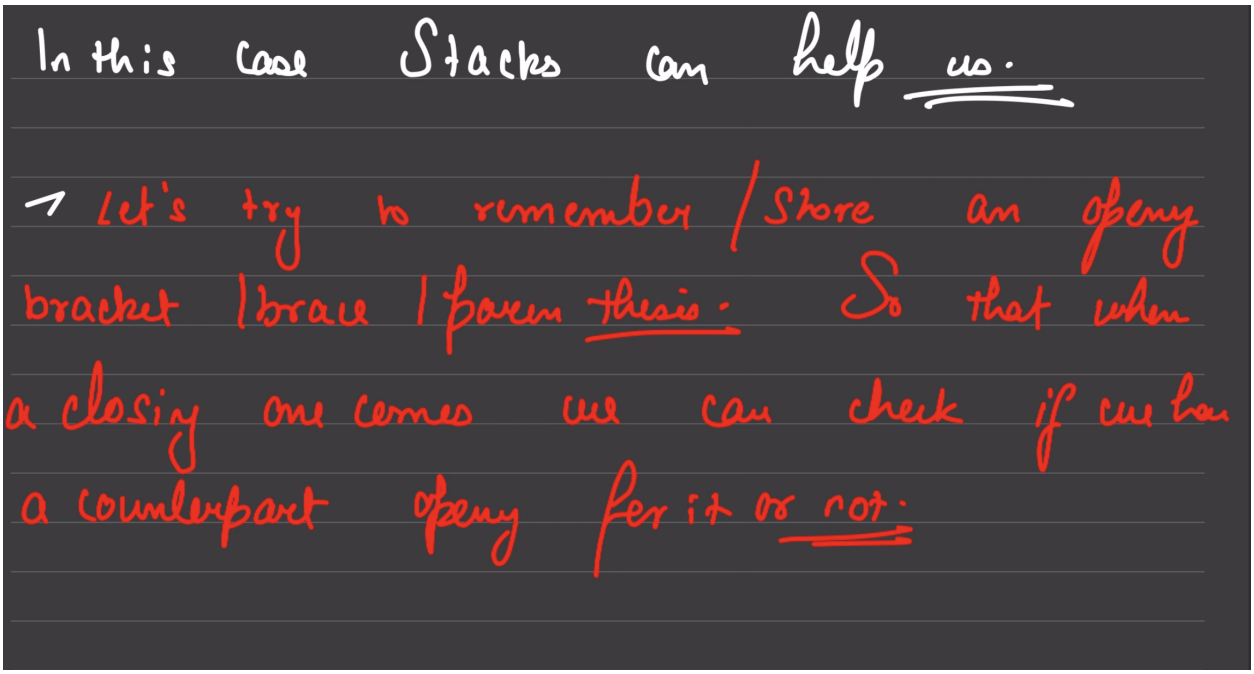


One important fact:



In order to solve the problem we can use stacks,






```
/**
 * @param {string} s
 * @return {boolean}
 */
class Stack {
    #arr;
    constructor() {
        this.#arr = [];
    }

    push(x) {
        /**
         * Time: O(1)
         */
        this.#arr.push(x); // add an element to the last
    }

    pop() {
        /**
         * Time: O(1)
         */
        this.#arr.pop(); // remove element from last index
    }

    isEmpty() {
        return this.#arr.length == 0;
    }
```

```
    top() {
        /**
         * Time: O(1)
         */
        return this.#arr[this.#arr.length - 1]; // return the element at the last index
    }
}

var isValid = function(s) {
    /**
     * Time: O(n)
     * Space: O(n) -> [[[[
     */
    let st = new Stack();
    for(let i = 0; i < s.length; i++) {
        if(s[i] == '(' || s[i] == '{' || s[i] == '[') st.push(s[i]);
        else {
            if(st.isEmpty()) {
                if(s[i] == ')' || s[i] == ']' || s[i] == '}') return false;
            }
            if(s[i] == ')' && !st.isEmpty()) {
                if(st.top() == '(') st.pop();
                else return false;
            }

            if(s[i] == ']' && !st.isEmpty()) {
                if(st.top() == '[') st.pop();
                else return false;
            }
            if(s[i] == '}' && !st.isEmpty()) {
                if(st.top() == '{') st.pop();
                else return false;
            }

        }
    }
    return st.isEmpty();
};
```