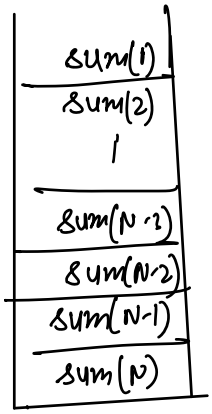
Stack.

(data structure) → (LIFO) → Last In First Out.

① Recursion & Call Stack.

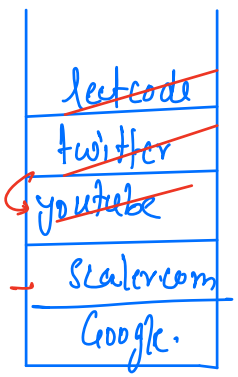


② Undo / Redo

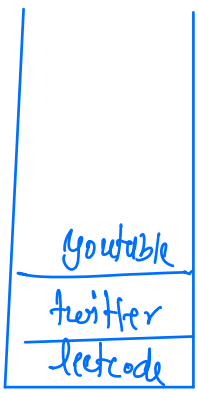
A → B → ~~C~~ → ~~D~~

③ Browser History. →

Backward & forward Tab



Backward.



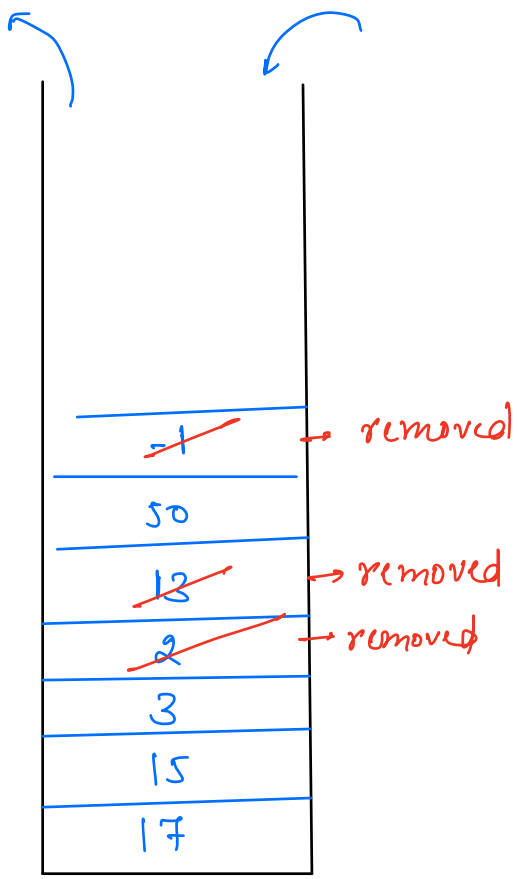
Forward.

Operations by stack.

O(1)

- → push(x) → insert x in stack
- → pop() → removes the topmost element
- → top() / peek() → access the topmost element
- → size(), isEmpty() → If stack is empty → true, otherwise → false.
  - size() ↓ return size of stack

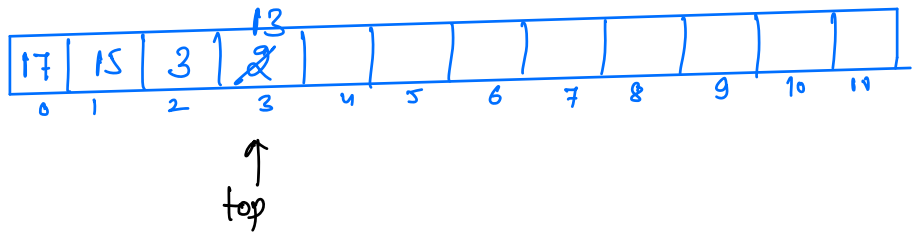
push (17)
push (15)
push (3)
push (2)
pop()
push (13)
pop()
push (50)
push (-1)
pop()
top() → 50



Implementation Of Stack.

→ Arrays

top → -1 → last position where top-most element exists.



push (17)
push (15)
push (3)
push (2)
pop()
push (13)
pop()
push (50)
push (-1)
pop()
top()

```
void push( int x) {
        if ( top == N-1) { //Stack overflow }
          top++ ;
          arr[top] = x;
}
```

```
void pop ( ) {
        if (top == -1) { //Stack is empty }
          top--
}
```

```
int top ( ) {
        if ( top == -1 ) { //Stack is empty }
        return arr[top];
}
```

```
int size ( ) {
        return top+1
}
```

max → 10^4 operations.

↑

initial size of stack.

push(10)

pop()

push(20)

pop()

push(30)

pop().

push(40)

pop()

⋮

100 operations.

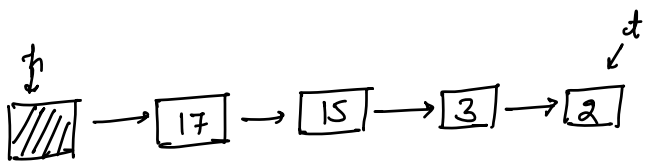
→ huge memory wastage.

* **Linked-list.** → insertion / deletion

head / tail.

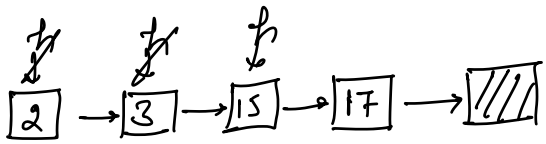
push (17)
push (15)
push (3)
push (2)
pop()
push (13)
pop()
push (50)
push (-1)
pop()
top()

① insertion → At tail → O(1)
removal → At tail → O(N) →



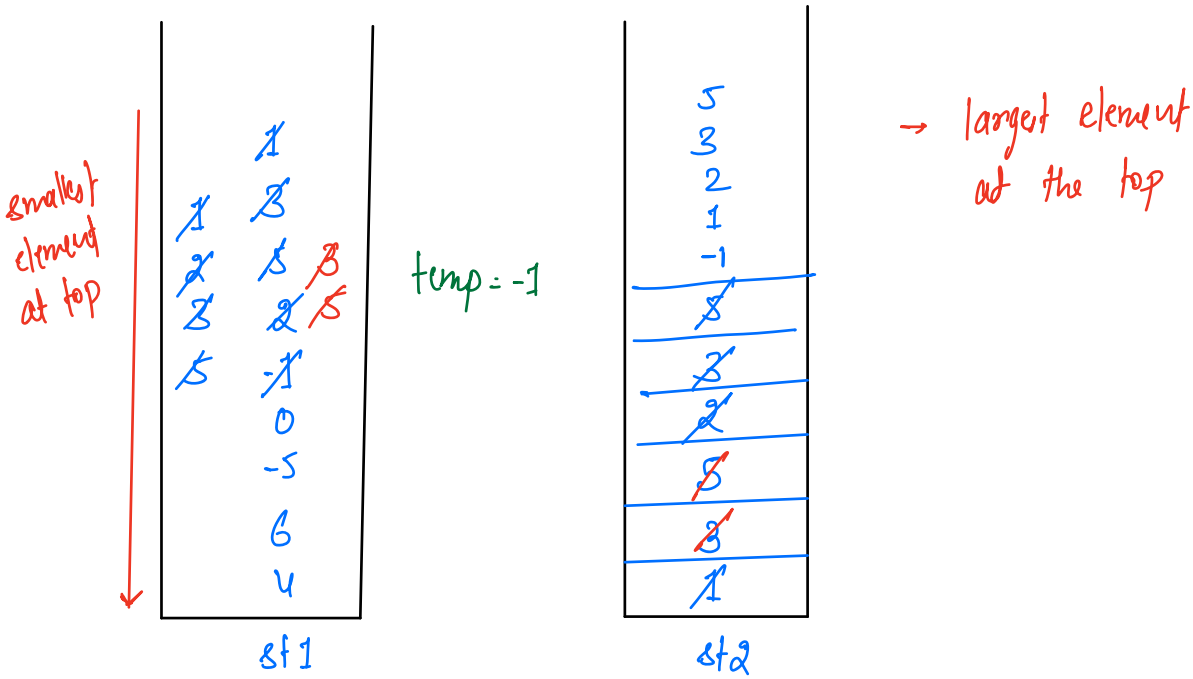
Soution :
DLL can be used.
⇓
(but more space will be used)

② insertion → At head → O(1)
removal → At head → O(1)



Q1) Sort a given stack in ascending order. [Smallest element should be present at top]

→ [Use another 1 stack.]



⇒ for one element, 2N operations will be performed in worst case.

T.C → N·2N → $\underline{O(N^2)}$, S.C → O(N)

pseudo-code.

st1 , st2. {aux. stack}

```
while ( !st.isEmpty() ) {
        int temp = st1.top()
            st1.pop()
        while (!st2.isEmpty && st2.top() > temp){
            x = st2.top()
            st2.pop()
            st1.push(x)
        }
        s2.push(temp);
}
```

→ Shift back all the elements from st2 to st1

```
while ( !st2.isEmpty() ){
    x = st2.top()
    st2.pop();
    st1.push(x)
}
```

Q: Check if the given sequence of parenthesis is valid or not.

( ), { }, [ ]

( ) [ ] → Yes.

( ) [{ }] → Yes.

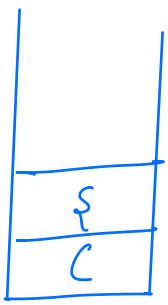
({) } → No

([}]) → No

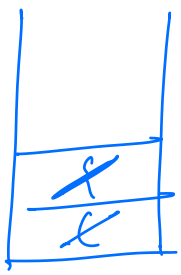
({ [{ } ( )] } ( )) ↓

({ [{ ((((((

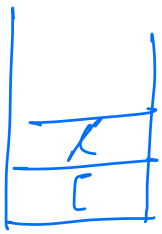
({) } → false.



({ })] → false.



[( ) → false.



pseudo-code.

```
Stack<character> st;
for( i= 0;  i < N;  i++){
        ch =  str[i];
    if (ch == '('  || ch == '{' || ch == '[')
        {
            st.push(ch);
        }
    else{
        if (st.size() == 0) {  // more closing brackets
            {
                return false
            }
        else if ( st.top() is not the counterpart
                of current closing parethesis
            {
                return false   // mis-match
            }
        else {
            {
                st.pop()
            }
        }
}
if (st.isEmpty()) { return true }
else { return false }
```

$\begin{bmatrix} T.C \rightarrow O(N) \\ S.C \rightarrow O(N) \end{bmatrix}$

[Stack<Character> st = new Stack<>();]

*→ **Double Character Trouble**

Given a string s. Remove equal pair of adjacent characters. Return the string without adjacent duplicates.

① a ~~b~~ ~~b~~ d → ad

② a b ~~c~~ ~~c~~ b de → a ~~b~~ ~~b~~ de → ade.

③ a ~~b~~ ~~b~~ b d → abd

④ ~~a~~ ~~b~~ ~~b~~ ~~c~~ ~~b~~ ~~b~~ ~~c~~ ~~a~~ c x → cx

⑤ a b c d c c d c a a b x x d a d

d
a
d
~~x~~
~~a~~
~~c~~
~~d~~
~~c~~
~~b~~
a

→ dada

⇓ reverse

adad → ans.

traverse the string.

$\downarrow$

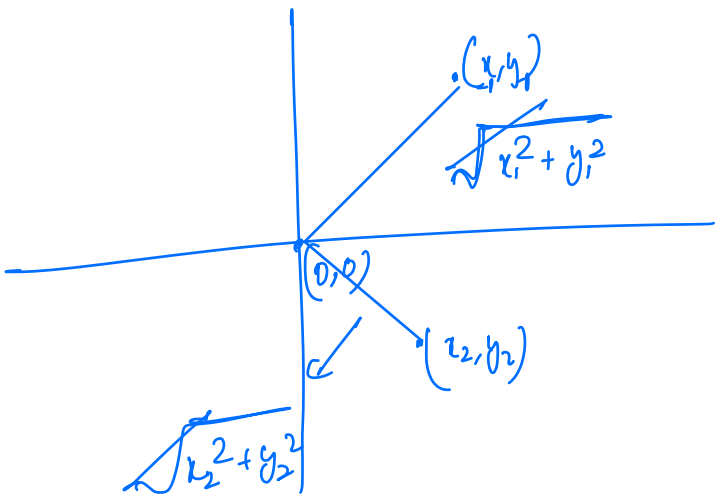
curr ch == st.top()

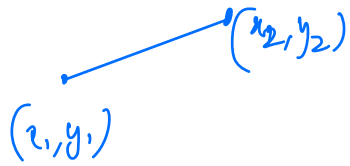
Yes $\swarrow$ pop()

No $\searrow$ push curr ch

$\boxed{\begin{array}{l} T.C \rightarrow O(N) \\ S.C \rightarrow O(N) \end{array}}$

* Reverse the string to get the final ans.





$$\sqrt{(y_2 - y_1)^2 + (x_2 - x_1)^2}$$

	P_0	P_1	P_2	P_3
a →	1	2	4	3
b →	2	3	1	3
	0	1	2	3

Arrays.sort(arr , comparator)