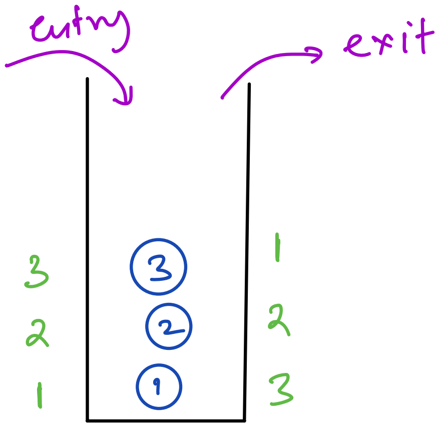
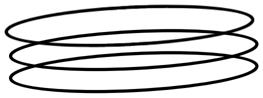
# Stacks 1 : Implementation & Basic Problems



→ linear Data Structure

→ Last In First Out (LIFO)

eg → 1. Pile of plates

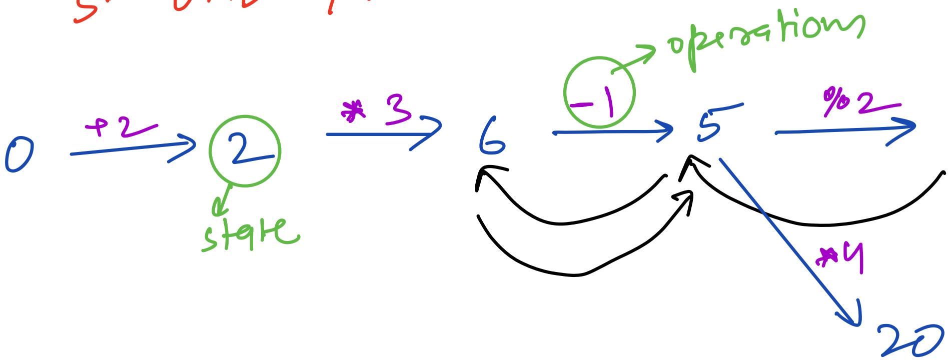


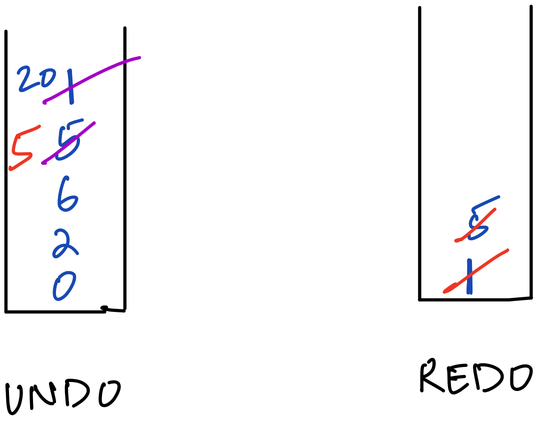
2. Stack of chairs

3. Stack of cards

4. Recursion

5. UNDO / REDO





Stack to store state

1. Operation → insert in UNDO stack + empty the redo stack

2. UNDO → move from undo to redo stack

3. REDO → move from redo to undo stack

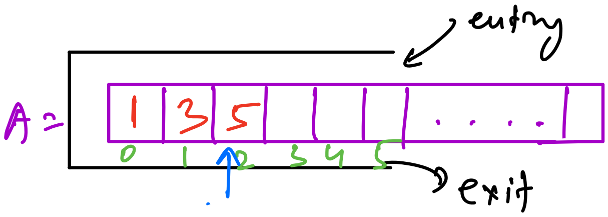
## Operations of Stack

1. push(x) → insert element x on top of stack
2. pop() → remove top element of stack
3. peek() / top() → get the top tee element of stack
4. isEmpty() → check if the stack is empty
5. clear() , size(), . . . .

TC = O(1)

Qun → Implement Stack using array.

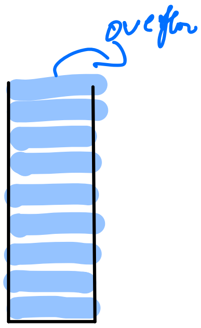
push(1)
push(3)
push(5)
isEmpty() → false
pop()
peek() → 5



top = ~~-1~~ ~~0~~ ~~1~~ 2

//stack → index '0' to 'top'

```
void push(x) {
    top++
    A[top] = x
}
```



```
int pop() {
    if(isEmpty()) //Underflow
        return -1 or INT_MIN
    top--;
    return A[top+1]
}
```

Overflow →

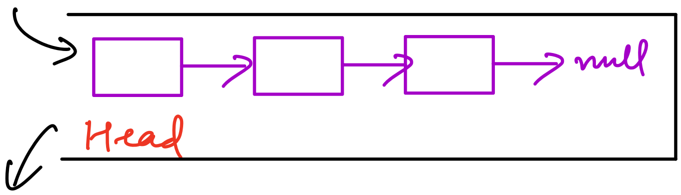
1. Restrict insertion over the defined size of array
2. Use dynamic array ✓

```
bool isEmpty() {
    return (top == -1);
}
```

```
int peek() {
  if(isEmpty()) return -1
  return A[top]
}
```

TC=O(1)

Ques → Implement stack using linked list.



push(x) → insert at Head

isEmpty(x) → (Head == null)

pop() → remove Head node

peek() → return Head.data

underflow case to handle

TC=O(1) ∀ operations

Ques → Check if the given sequence of parantheses containing only → { } ( ) [ ] is valid.

Valid → 1. count of { = count of }

count of ( = count of )

count of [ = count of ]

2. Travelling from left to right #open >= #close

3. ∀ opening of one type there closing of the sam

type in order w/o overlapping with another type

{ [ ) } ( ] → not valid

( ( ) { } ) → valid

{ ( [ ) ] } → not valid

( { } [ ] ( ) [ ] ( ) ) → valid

✓ ✓ ✓ ✓ ✓ ✓

{ [ [ ( ) ] ( ) ] }

check if latest unpaired bracket is '('

LIFO => stack

```
stack st.
for ( i=0 to n-1) {
    if(s[i] == '(' || s[i] == '{' || s[i] == '[') {
        st.push(s[i])
    }
    else {
        if( st.isEmpty()) return false → invalid
        ch= st.pop()
        if(s[i] == ')' && ch != '(') return false
        if(s[i] == '}' && ch != '{') return false
        if(s[i] == ']' && ch != '[') return false
    }
}
return ~~true~~ st.isEmpty()
```

{ ~~[ ]~~ ~~{ }~~ → ~~true~~ false

TC = O(N)    SC = O(N)

Ques → Given a string, remove equal pair of consecutive elements multiple times till it is possible & return the final string as answer.

eg

a b ~~cc~~ d    abd

a ~~bb~~ ~~cc~~ d    ad

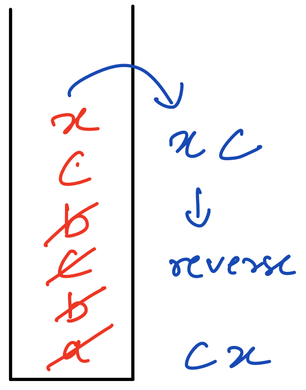
a b ~~cc~~ b    a ~~bb~~    a

a ~~bb~~ c b c ~~dd~~ c    a c b ~~cc~~    a c b

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

a ~~c c~~ a c x    ~~a a~~ c x    c x

✓ ✓ ✓ ✓ ✓ ✓
a ~~b~~ ~~b~~ c ~~b~~ ~~b~~ c a c x



store from left to right
remove right to left } LIFO → stack

```
for ( i = 0 to n-1) {
    if( ! st.isEmpty() && st.peek() == s[i]) {
            st.pop()
    }
    else {
            st.push(s[i])
    }
}
```

TC = O(N)
SC = O(N)

// create string using stack
// return the reversed string

| Infix | Postfix |
|---|---|
| 2+5 | 2 5 + |
| operand1 operator operand2 | operand1 operand2 operator |

$((6 - 5) * 2)$

$$\underbrace{\underbrace{6\ 5\ -}_{6-5=1}\ 2\ *}_{1*2=2}$$

$(6 - (5*2))$

$$\underbrace{6\ \underbrace{5\ 2\ *}_{5*2=10}\ -}_{6-10=-4}$$

$$\underbrace{\overset{y}{3}\ \overset{x}{5}\ +}\ 2\ -\ \underbrace{2\ 5\ *}\ -$$

$$\underbrace{8\ 2\ -}\ 10\ -$$

$$\underbrace{6\ 10\ -}$$

$$-4$$

```
for (i = 0 to n-1) {
    if (A[i] is operand / integer)
        st.push(A[i])
    else {
        operator = get operator (A[i])
        x = st.pop()
        y = st.pop()

        z = y operator x
        st.push(z)
```

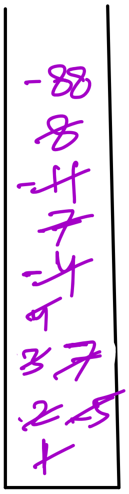
3

3

return st.peek()

1 2 $\overset{y}{3}$ $\overset{x}{4}$ + - + 7 - 8 *

| x | y | z |
|---|---|---|
| 4 | 3 | 3+4=7 |
| 7 | 2 | 2-7=-5 |
| -5 | 1 | 1+(-5)=-4 |
| 7 | -4 | -4-(7)=-11 |
| 8 | -11 | -11×8 = -88 |


-88
~~8~~
~~-11~~
~~7~~
~~-4~~
~~4~~
~~3~~ ~~7~~
~~2~~ ~~-5~~
~~1~~