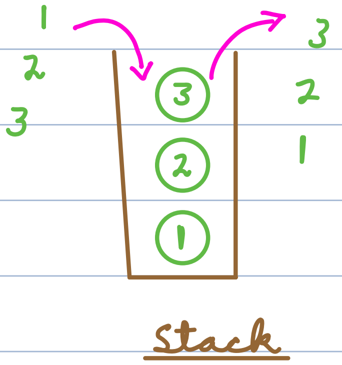


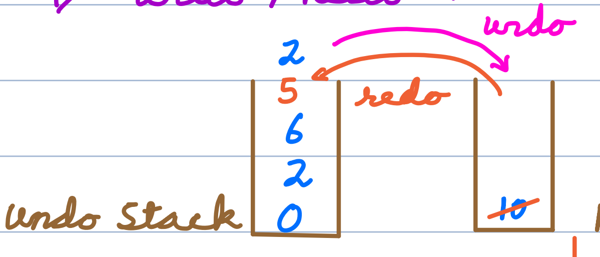
LIFO → Last In First Out

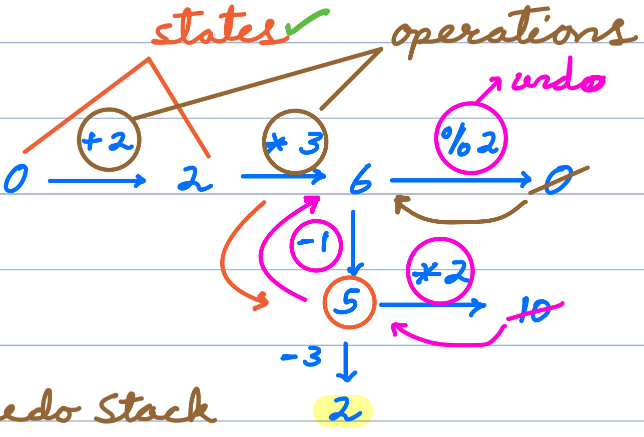
1. **Pile of Plates**:
   Imagine a scenario where you have a pile of plates, you can only put a plate on the pile from top also only pick a plate from top. You can't really see the plates underneath the top one without first removing the top plate, which means only the first plate is accessible to you.

2. **Stack of Chairs**:
   We usually place identical chairs on the top of on another, which makes them look like a stack. Similar to the previous example you can only position or choose a chair from top, and you won't be able to take or see the chair in middle without picking out all chairs on top of that one.

3) Recursion ✓

4) Undo / Redo →





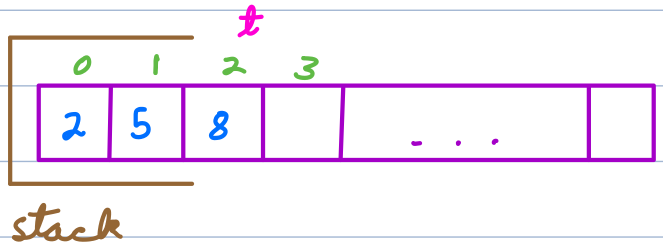
↓
empty redo stack if
any new operation is performed.

Operations

1) push (x) → insert x at top of the stack
2) pop () → remore top data from stack
3) peek () / top () → get the top data from stack
4) isEmpty () → to check if stack is empty

TC = O(1) ✓

Q → Implement stack using arrays.



0 —— t

initial value of t = -1

push (2)
push (5)
push (10)
peek () → 10
pop ()
push (8)

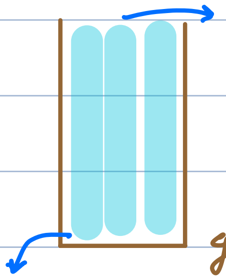
```
boolean isEmpty () {
    return (t == -1)
}
```

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

```
int peek () {
    if (isEmpty ())
        return -1
```

```
int pop () {
    if (isEmpty ()) //underflow
        return -1
    return A[t]
}
```

```
    x = A[t]
    t--
    return x
}
```



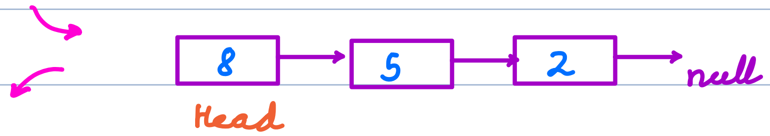
overflow
1) Not to allow insertion if array is full.
2) Use dynamic array ✓

empty ⇒ underflow

Q → Implement stack using Linked List.

delete tail node → TC = O(N)

push (2)
push (5)
push (10)
peek () → 10
pop ()
push (8)



initially → Head = null

```
boolean isEmpty () {
    return Head == null
}
```

push (x) → insert x at Head
pop () → delete Head
peek () → return Head data

Q → Check if the given sequence of parenthesis is valid.

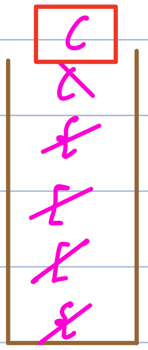
( ) { } [ ]

(([ ])) ✓ ({ [ ]) ✗ ([ )] ✗

↳ ✗

✓ ✓ ✓ ↓ ✓ ↓ ↓ ↓ ✓ ↓ ✓

{ [[ ] { }] } ( ) (



[ ] ✓ ( ) ✓

{ } ✓

[ ] ✓

{ } ✓

store in particular order

maintaining latest data handy → use Stack

```
for i → 0 to (N-1) {
    ch = s[i]
    if (ch == '(' || ch == '{' || ch == '[')
        st.push(ch)
    else if (ch == ')') {
        top = st.pop()
        if (top != '(')   return false
    }
    else if (ch == '}') {
        top = st.pop()
        if (top != '{')   return false
    }
    else if (ch == ']') {
        top = st.pop()
        if (top != '[')   return false
    }
}  return   st.isEmpty()
```

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

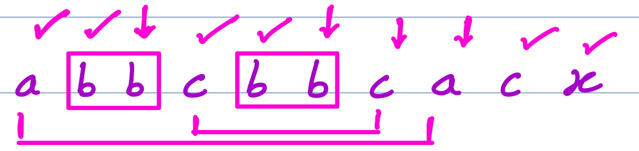
Q → Given a string, remove equal pair of consecutive elements till possible.

eg → a c ~~d d~~ b → acb

a b ~~d d~~ b → a ~~b b~~ → a

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

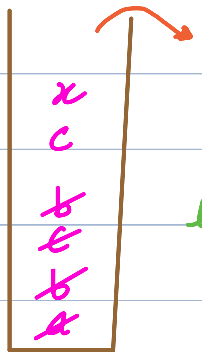
~~a a~~ c x → cx



store + maintain order

+ keep track of latest

⇒ use stack



c x

append char on left side to maintain order.

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

ans = ""
while (!st.isEmpty()) {
    ans = st.pop() + ans   // append
}
```

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

Q → Evaluate postfix expression.

Infix	Postfix
operand1 operator operand2	operand1 operand2 operation
a + b	a b +

Eg → 4 3 3 * + 2 -

9

4 + 9 = 13   13 - 2 = 11

5 2 * 3 - → 5*2 = 10

10 3 - → 10 - 3 = 7

10

3 5 + 2 - 2 5 * - } 6 - 10 = -4

8   8-2=6

```
for i → 0 to (N-1) {
    ch = s[i]
    if( isDigit(ch))
        st.push
    else {        // ch → operator '+'
        y = st.pop()
        x = st.pop()             x + y
        st.push( x ch y) // operation
    }
} return st.pop()
```

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