

Today's Contest

1. Stack Intro

2. Stack Implementation:

a. Using arrays

b. Using linked list

c. Inbuilt stack

3. Balanced parenthesis

4. Double char trouble

Stack:

Insert

Top



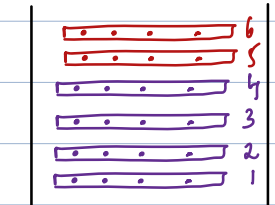
Remove

Top



Stack: Insertion & deletion from top side

Property: Last In First Out: **LIFO**



Use Case:

1. Recursion
2. Undo/Redo & \leftrightarrow
3. Expression Evaluation.

Functions:

push(): Insert x on top of stack

pop(): delete top most element

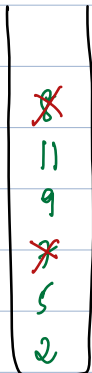
peek(): Return top element ~~# It will not be deleted~~

size(): Return no. of elem in stack

#Note: When ever we use stack, we can only use above 4 functions.

#DryRun:

En: 2 5 7 pop() peek() 9 11 8 pop() peek() peek()
7* 5 ✓ ✓ ✓ 8* 11 11



#obs

1. Only element we can ans is peek
2. Push & pop happen from same side.
3. All stack operations take $O(1)$ time.

#Stack Implementation: Array:

ar[4] = { 2, 5, ~~7~~, 11 }

top = -1 → 0 → 1 → 2 → 3 → 4 {out of bounds}

2 5 7 peek() pop() 9 11 8 pop() pop() pop() pop() pop()

Code

Memory limit. overflow underflow
int ar[N], top = -1; #Index of peak element.

```
void push(int n) {
    if (top == N-1) { "throw err Stack Overflow" }
    ar[++top] = n;
}
```

```
void pop() {
    if (top == -1) { "throw err Stack Underflow" }
    top--;
}
```

```
int peek() {
    if (top == -1) { "throw err Stack Underflow" }
    return ar[top];
}
```

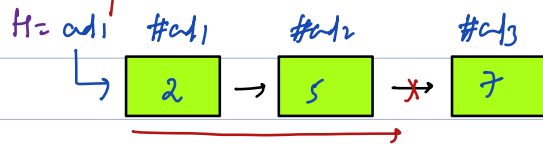
TODO: One check error name
It might be different.

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

#Stack Implementation: Linked list

#Version 1:

$H = \text{nullptr}$



2 5 7 pop() peek() 9 11 8 pop() pop() pop() pop()

Iterate till tail & break link.

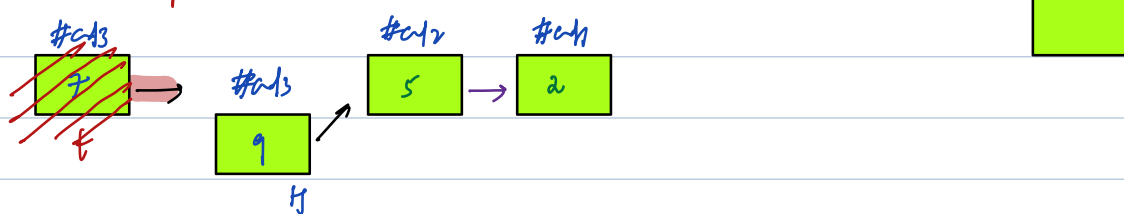
Since with above is single pop = $O(N)$

#Single Linked list:

Insert_at begin() : $O(1)$	delete_at begin() : $O(1)$	✓
Insert_at end() : $O(1)$	delete_at end() : $O(N)$	✗

#Version 2:

$H = \text{nullptr}$



2 5 7 pop() peek() 9 11 8 pop() pop() pop() pop()

Node $t = H$ $H \rightarrow \text{delete}$ Node $nn = \text{new Node}(9)$
 $H = H \rightarrow \text{next}$ $nn \rightarrow \text{next} = H$
 $t \rightarrow \text{next} = \text{nullptr}$ $H = nn$
 delete t

#Note: For size keep a count variable.

Code

```
class Node {  
    public:  
        int data;  
        Node* next;  
        Node(int n) {  
            data = n, next = NULL;  
        }  
}
```

```
Node* H = NULL;  
int c = 0;
```

```
void push(int n) {  
    Node* nn = new Node(n);  
    nn->next = H;  
    H = nn;  
    c++;  
}
```

```
void pop() {  
    if (H == NULL) { "throw err Stack Underflow"  
    Node* t = H;  
    H = H->next;  
    t->next = NULL;  
    delete t;  
    c--;  
}
```

```
int peek() {  
    if (H == NULL) { "throw err Stack Underflow"  
    return H->data;  
}
```

```
int size() { return c; }
```

Inbuilt: C++

```
stack<datatype> st;
```

```
st.push(10);
```

```
st.top(); # Returns top element of stack
```

```
st.pop(); # Delete's top element of stack
```

```
st.size();
```

Inbuilt: Java

```
Stack<datatype> st = new Stack<datatype>();
```

```
st.push(10);
```

```
st.peek(); // Returns top element of stack
```

```
st.pop(); // Removes & returns the top element of stack
```

```
st.size();
```

11:00

Inbuilt: python

```
from collections import deque
```

```
st = [];
```

```
st.append(10); // Add element at back
```

```
st.pop(); // remove & returns last element
```

```
st[-1]; // Access last element
```

```
len(st); // Get size of list
```

Check if a given sequence of parenthesis is balanced or not?

Type of brackets { } [] ()

Balanced parenthesis:

A sequence is balanced, if for every opening, there is a closing bracket & they have to be correctly matched.

Ex,

1. (({ })) True

2. { { }) False

3. { (}) False

Hint: ({ [] })

Bracket we open last, we need to close it first: LIFO

Ex: { [[] { }] }



obs: if a current closing parenthesis, is not matching last open parenthesis, it's not a balanced sequence.

Ex2: { [() }

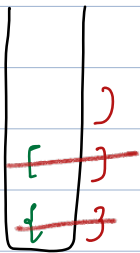


obs2: After iterating on entire sequence.

if stack size > 0:

Few open parenthesis are yet to be closed.
It's not a balanced sequence.

Ex3: { [] }))



obs3: If a current closing parenthesis

if stack size == 0:

No opening for closing parenthesis.
It's not a balanced sequence.

Steps: TC: $O(N)$ SC: $O(N)$ # of all chars are open space can go to N.

Input string s # parenthesis sequence

stack & int st;

for (int i = 0; i < s.length(); i++) {

if (s[i] == '[' or s[i] == '{' or s[i] == '(') {
st.push(s[i]);

else { # s[i] is closing bracket

if (st.size() == 0) { return false; }

if (st.top() is not matching s[i]) { return false; }

st.pop(); # open parenthesis balanced with closing parenthesis

}

return st.size() == 0; # if size == 0 it's a balanced parenthesis.

boolean isbalanced (String s){

}

Double Character Problem:

Given a string S , Remove equal pair of adjacent characters

Return the string without adjacent duplicates

Ex1: $a \cancel{b} \cancel{b} d = ad$

Ex2: $a \cancel{b} \cancel{c} \cancel{c} \cancel{b} d e = ade$

Ex3: $a \cancel{b} \cancel{c} \cancel{b} \cancel{c} \cancel{b} \cancel{c} a c y = ay$

Ex4: $ababab = ababab$

#Idea

$S = \vec{a} \vec{d} \vec{e} \vec{b} \vec{b} e c a a c d e d$

ans = $a d e \cancel{b}$

obs: In ans we add and delete char from same side, hence can use stack idea.

Dry Run:

S = a d e b b e c a a c d e d

~~a~~
~~d~~
~~e~~
~~b~~
~~b~~
~~e~~
~~c~~
~~a~~
~~a~~
~~c~~
~~d~~
~~e~~
~~d~~

Ans = dea

Given string S;

stack < char > st;

for (int i = 0; i < S.length(); i++) {

if (st.size() == 0 || st[i] != st.top()) {

st.push(st[i]);

else {

st.pop();

}

string ans = "";

while (st.size() > 0) {

char ch = st.top();

st.pop();

ans += ch;

}

Issue: final ans we are getting in reverse order
& Ways to handle it

Way 1: Reverse final ans string

Way 2: Iterate on S from last to 0th index &
directly return final S.