

Today's Content

1. Stack Intro
2. Stack Implementation:
 - a. Using arrays
 - b. Using linked list
 - c. Inbuilt stack
3. Balanced parenthesis
4. Double char troubl

Stack:

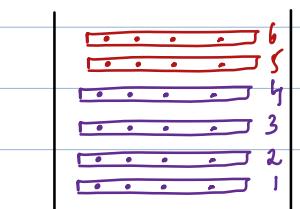
Insert

Top



Remove

Top



Stack: Inserting & deleting from top slot

Property: Last In First Out: LIFO

Use Case:

1. Recursion

2. Undo/Redo & ← →

3. Expression Evaluation.

Functions:

push(n): Insert n on top of stack

pop(): delete top most element

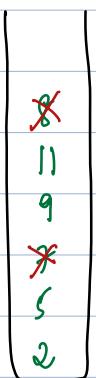
peek(): Return top element *# It will not be deleted*

size(): Return no: of ele in stack

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

#Dry Run:

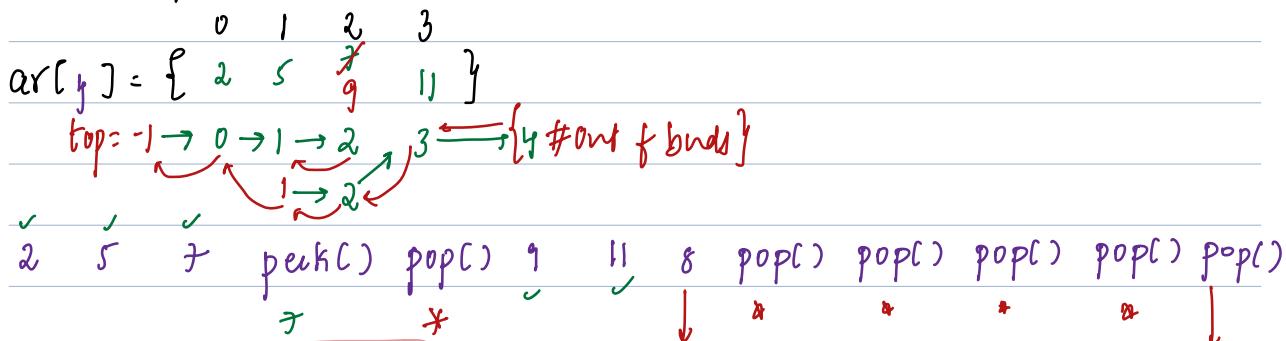
Ex: 2 5 ↑ ↑ pop() peek() 9 ↑ ↑ 8 pop() peek() peek()
 7* 5 ✓ ✓ 8* 11 11



#Obs

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

#Stack Implementation: Array:



Code

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

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

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

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

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

→ TODO: One check error name
It might be different.

#Stack Implementation: Linked List

#Version 1:

~~H = nullptr~~

H = ~~addr1~~

#addr1 #addr2 #addr3



→ Last node of linked list



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

→ #Iterate till tail & break link.

#Issue with above is single pop = $O(N)$

#Single Linked List:

insert_at_begin() : $O(1)$

delete_at_begin() : $O(1)$

✓

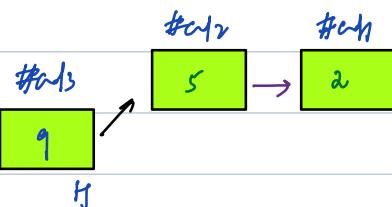
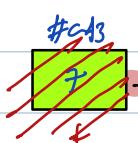
insert_at_end() : $O(1)$

delete_at_end() : $O(N)$

*

#Version 2:

~~H = nullptr~~



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

Node *t = H t->data;

Node *n = new Node(9);

H = H->next;

n->next = H

t->next = nullptr;

H = n;

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 error StackUnderflow" }

Node* t = h;

h = h->next;

t->next = NULL;

delete t;

c--;

```
int peek() {
```

if (h == NULL) { "throw error StackUnderflow" }

return h->data;

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

Inbuilt: C++

```
Stack<datatype> st;  
st.push(10);  
st.top(); // Returns top element of stack  
st.pop(); // Deletes top element of stack  
st.size();
```

Inbuilt: Java

```
Stack<datatype> st = new Stack<datatype>();  
st.push(e);  
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]; // Gets 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: C { [] }

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: { [()] }



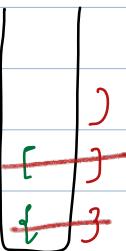
obs2: After Iterating on entire sequence.

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) # if all chars are open space can go to N.

Input string s # parenthesis sequence

stack init & t;

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

if (s[i] == '[' || s[i] == '{' || 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) {

3

Double Character Trouble:

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

Ex4: $ababab = ababab$

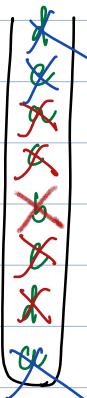
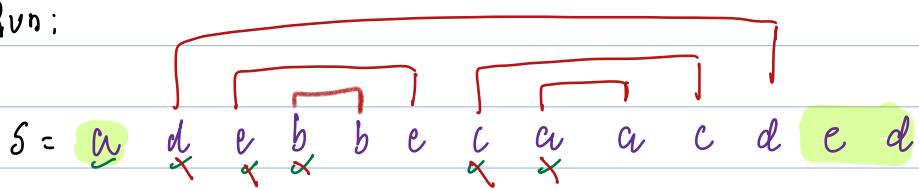
idea

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

ans = $a \ x \ d \ x \ e \ x \ b$

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

#Dry Run:



Given string s :

$ans = dea$

stacks chars st ; j

```
for (int i=0; i < s.length(); i++) {  
    if (st.size == 0 || s[i] != st.top()) {  
        st.push(s[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
2 ways to handle it

Way 1: Reverse final ans string

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