Stack :-

Page No.
Date

Stack :- // Questions.

Q Reverse a String using Stack.

Approach : Stack has a property of LIFO, means it reverse the item.

Code :

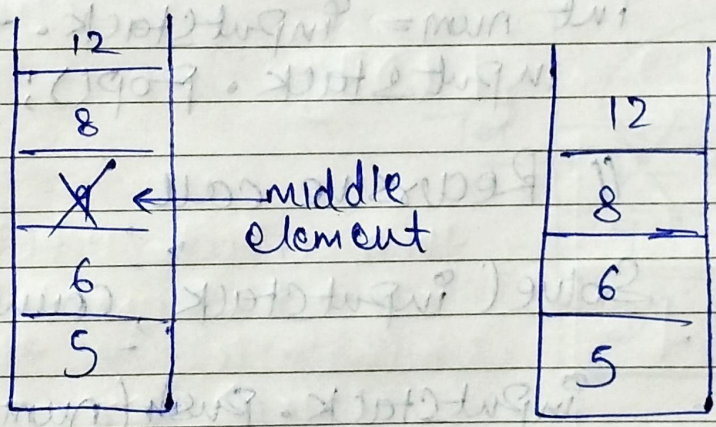
```
#include<iostream>
#include<stack>
using namespace std;
int main(){
    string str="TUSHAR";
    stack<char> s;
for(int i=0; i<str.length(); i++)
  {
    char ch=str[i];
      s.push(ch);
  }
string ans="";
while(!s.empty()){
    char ch=s.top();
    ans.push_back(ch);
```

```
    s.pop();
  }
  cout<<"Answer is : "<<ans<<endl;
  return 0;
}
```

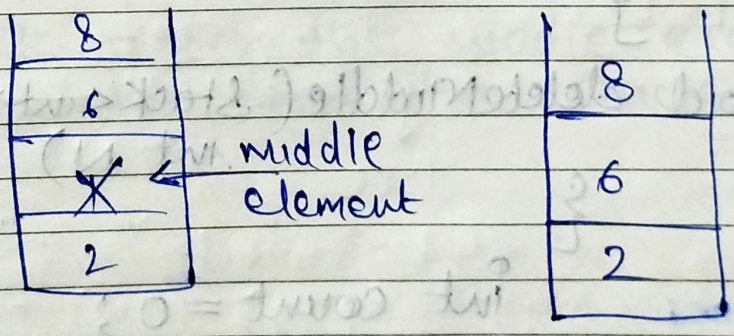
Time complexity = O(N)
Space complexity = O(N)

Q Delete the middle element in stack.

Eg:-



Eg:-

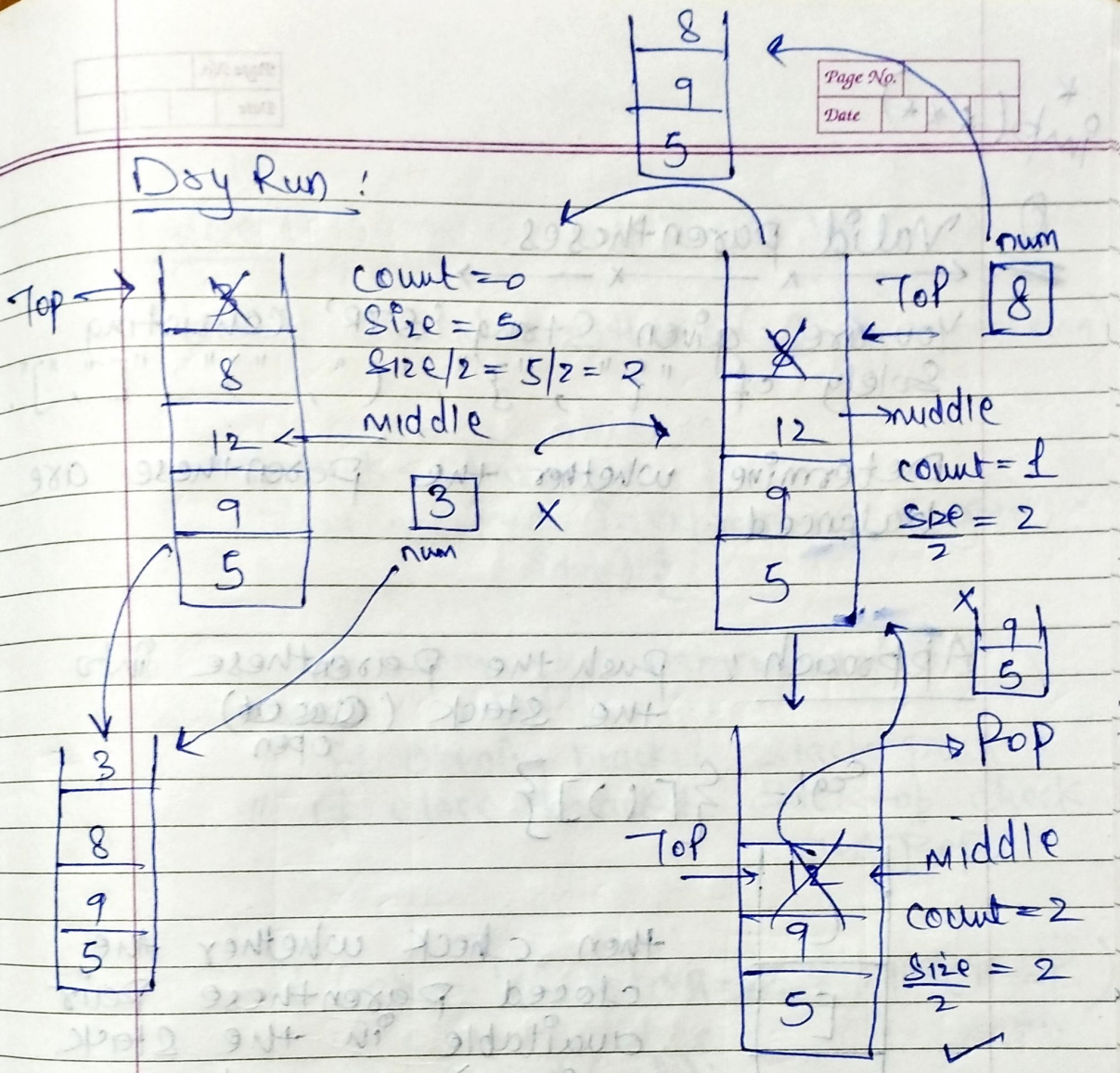


Approach : using a count variable to check to reach to the mid

Code :-

```
void Solve(stack<int> &inputstack,
           int count, int size)
{
    // base case
    if (count == size/2){
        inputstack.pop();
        return;
    }
    int num = inputstack.top();
    inputstack.pop();

    // Recursive call
    Solve(inputstack, count+1, size);

    inputstack.push(num);
}
void deleteMiddle(stack<int> inputstack,
                  int N)
{
    int count = 0;
    Solve(stackinput, count, N);
}
```

Dry Run :



The num element is push to the stack after the middle element is deleted from the stack.

int(***)

Page No.
Date

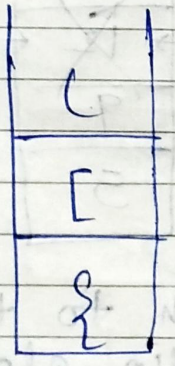
Q. Valid parentheses

You are given String 'STR' consisting solely of "{", "}", "(", ")", "[", "]".

Determine whether the paranthese are balanced.

Approach: Push the paranthese into the stack (open)

Eg:- {[()]}



then check whether the closed paranthese pair available in the stack (Top) or not

if

True
// balanced
Eg:- {[()]}
→ {[()]}

false
// false
Eg:- {[)}
X

Code :-

```
bool isValidparenthesis ( string expression)
{
    Stack<char> S;
    for( int i=0; i<expression.length();
                i++) {

        char ch= expression[i];

// if opening bracket, stack push
// if close bracket, stacktop check
                        and Pop

    if (ch=='(' || ch=='{' || ch=='[')
      {  S.push( ch);
      }
    else { // closing bracket
        if( !S.empty()) {
            char top = S.top();
        if( (ch==')' && top=='(') ||
            (ch=='}' && top=='{') ||
            (ch==']' && top=='['))
```

```
        { S.Pop();
        }
          else {
                  return false;
                }
          }
        else {
                return false;
              }
        }
    }
      if(s.empty())
            return true;
       else
            return false;
    }
```

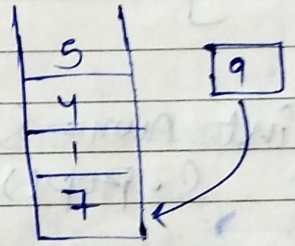
Time complexity = O(N)
Space complexity = O(N)

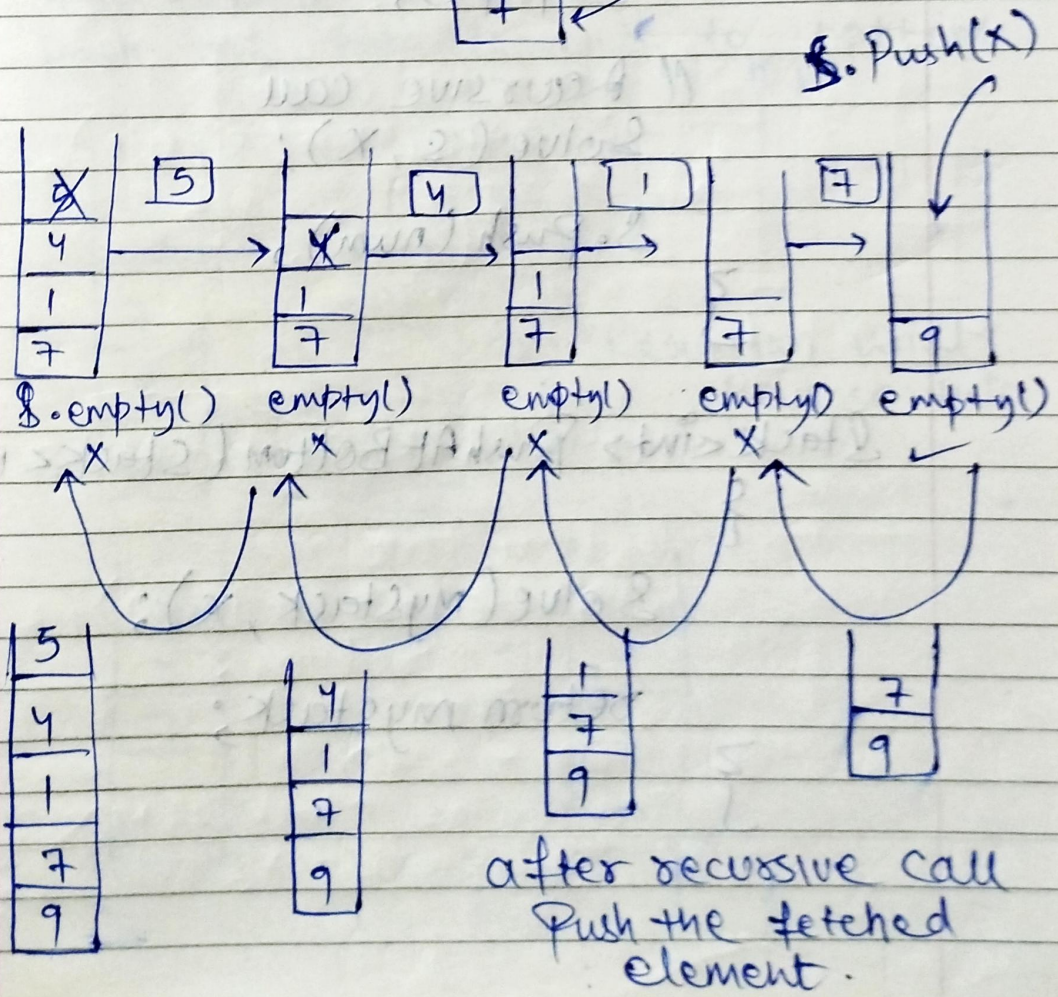
Page No.
Date

Q Insert an element at its bottom in a Given Stack.

You are given a stack/deque of integer 'My-Stack' and an integer 'x'. your task is to insert 'x' to the bottom of 'My-Stack' and return updated Stack/deque.

Approach: Eg:-





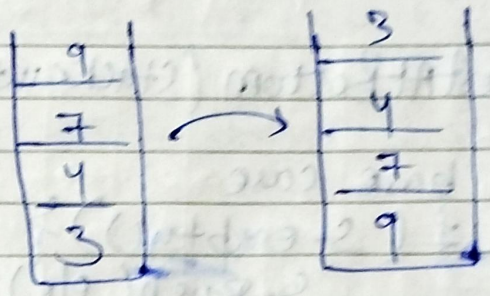
Code :-

```
void Solve (stack<int>&s , int x){
    // base case
    if(s.empty()){
        s.push(x);
        return;
    }

    int num=s.top();
    s.pop();

    // Recursive call
    Solve(s,x);

    s.push(num);
}

stack<int> PushAtBottom(stack<int>mystack,intx
{
    Solve(mystack,x);

    return mystack;
}
```

Page No.
Date

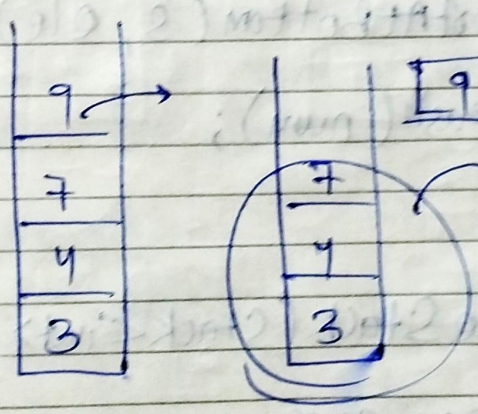
Q Reverse a Stack.

Eg:-

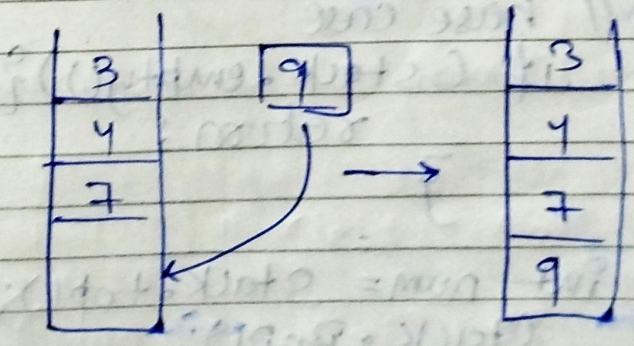


Approach: Solve only one case, and recursion will handle other

num = s.top(), Store top element and then Pop. Then recursion will reverse the remaining Stack, and then push the top element to bottom.



ck, int x)



Code:

```
void InsertAtBottom(stack<int>& s, int ele)
{
    // base case
    if(s.empty()){
        s.push(ele);
        return;
    }

    int num = s.top();
    s.pop();

    // Recursive call
    InsertAtBottom(s, ele);

    s.push(num);
}

void reverseStack(stack<int> & stack)
{
    // base case
    if(stack.empty()){
        return;
    }

    int num = stack.top();
    stack.pop();
```

// Recursive call
reverseStack(stack);

InsertAtBottom(stack, num);

}

(*) Time Complexity = $O(N^2)$
Space Complexity = $O(N)$

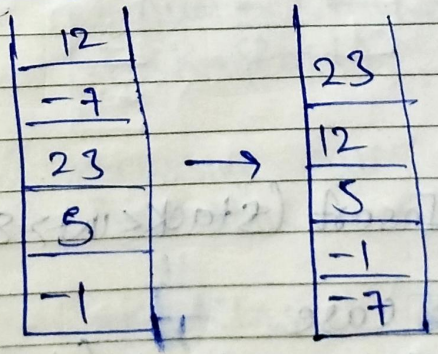
imp (***)

Q Sort a Stack:

We are given a stack and have to sort it but we can't use any loop like while, for ... etc.

Approach:

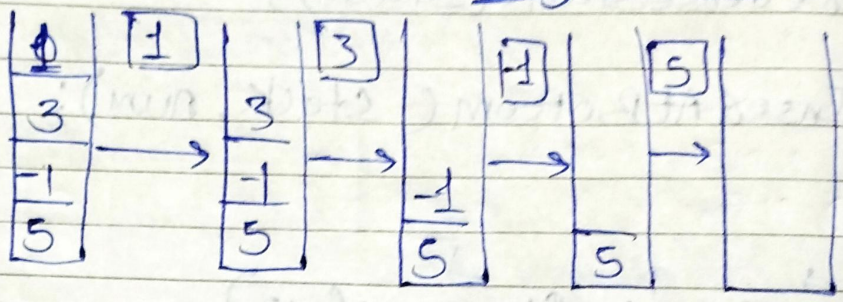
Eg:-



First empty the stack one by one just like in InsertAtBottom and then push the element in sorted

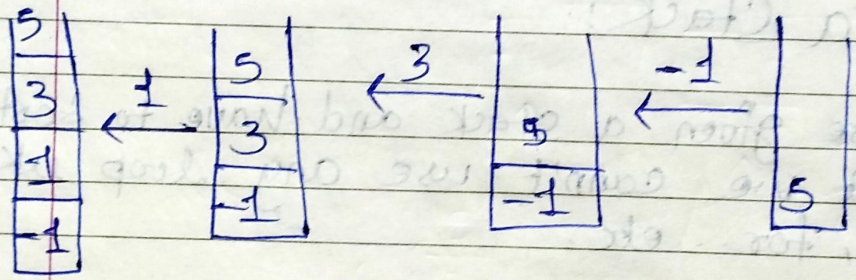
manner.

Dry Run :-



empty() ✓

Now, start pull element in sorted manner.



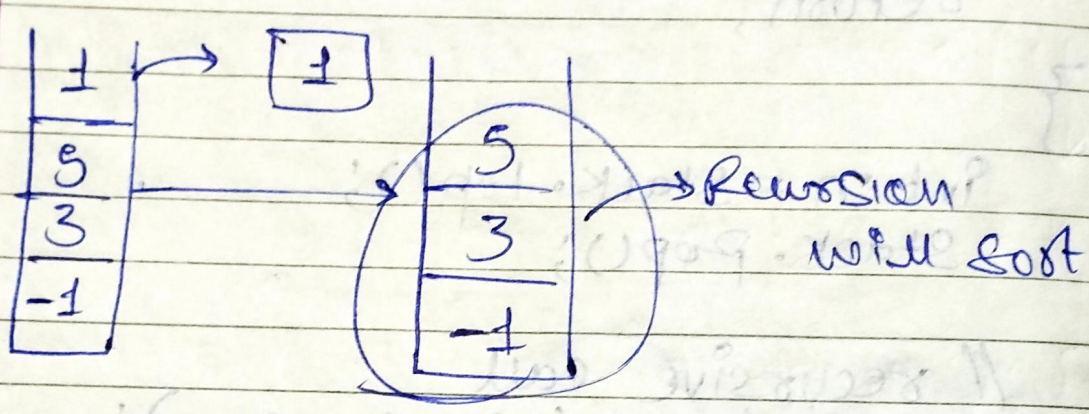
✓ Sorted.

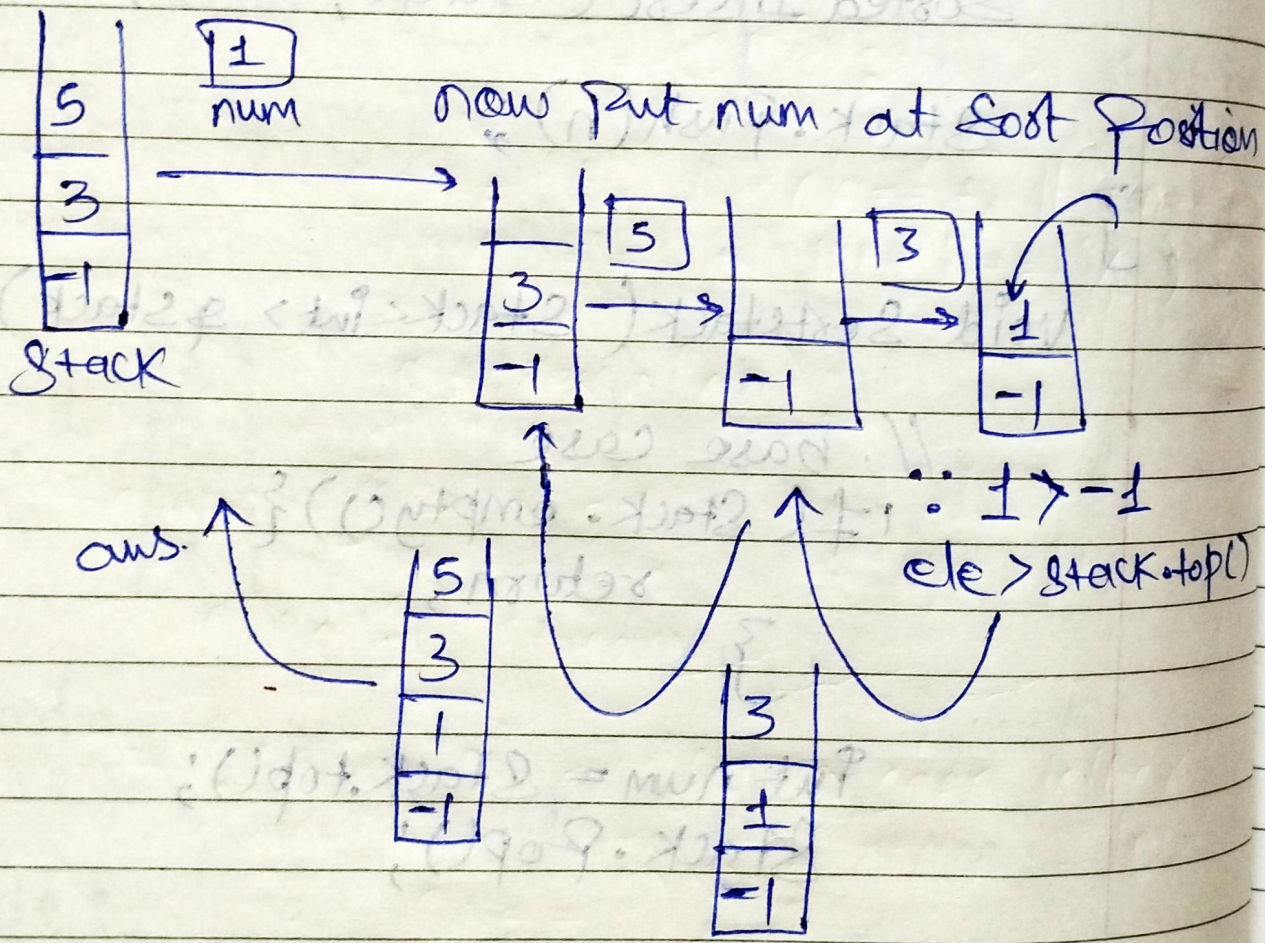
Code :-

```
void SortedInsert (stack<int> &stack, int num
{
    // base case
    if( stack.empty() || (!stack.empty()
        && stack.top<num)) {
```

```
    stack.push(num);
    return;
}
    int n = stack.top();
    stack.pop();

    // recursive call
    SortedInsert(stack, num);

    stack.push(n);
}
void SortStack(stack<int> &stack){

    // base case
    if(stack.empty()){
        return;
    }

    int num = stack.top();
    stack.pop();

    // recursive call
    SortStack(stack);

    SortedInsert(stack, num);

}
```

Dry Run:


1
5
3
-1
1
5
3
-1
Recursion will sort


5
3
-1
stack
1
num
now put num at sort position
5
3
-1
3
1
-1
∵ 1 > -1
ele > stack.top()
ans.
5
3
1
-1
3
1
-1