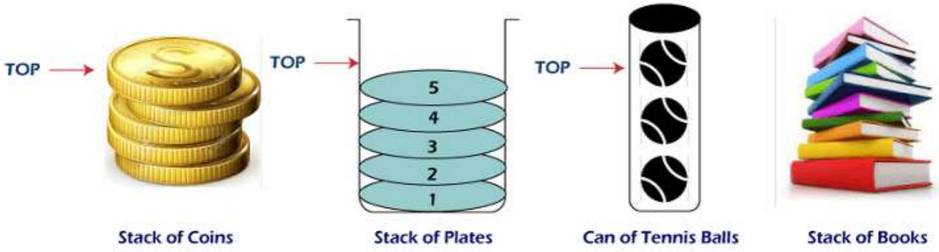
8/11/2023

STACK CLASS - 1

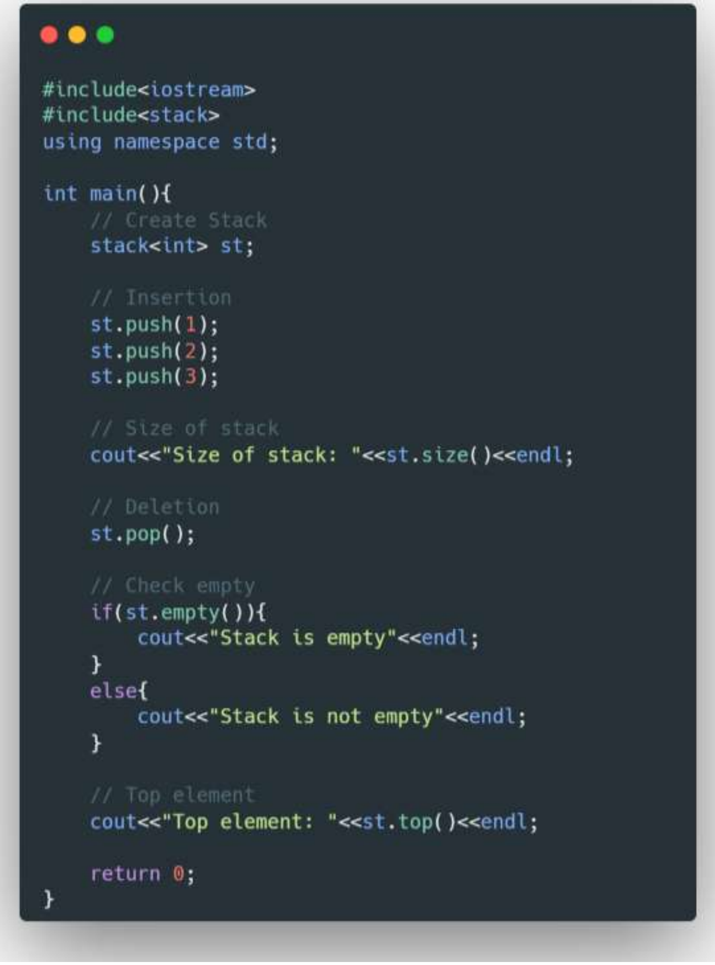
📂 What is stack?

1. *Stack is a linear data structure where insertion or deletion of elements is done from only one side/end which is called top of the stack.*

2. ***LIFO:*** *Last In First Out*

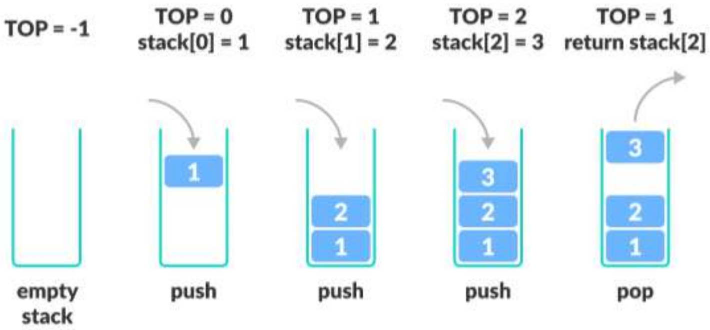


Important Operation of a Stack



```
#include<iostream>
#include<stack>
using namespace std;

int main(){
    // Create Stack
    stack<int> st;

    // Insertion
    st.push(1);
    st.push(2);
    st.push(3);

    // Size of stack
    cout<<"Size of stack: "<<st.size()<<endl;

    // Deletion
    st.pop();

    // Check empty
    if(st.empty()){
        cout<<"Stack is empty"<<endl;
    }
    else{
        cout<<"Stack is not empty"<<endl;
    }

    // Top element
    cout<<"Top element: "<<st.top()<<endl;

    return 0;
}
```

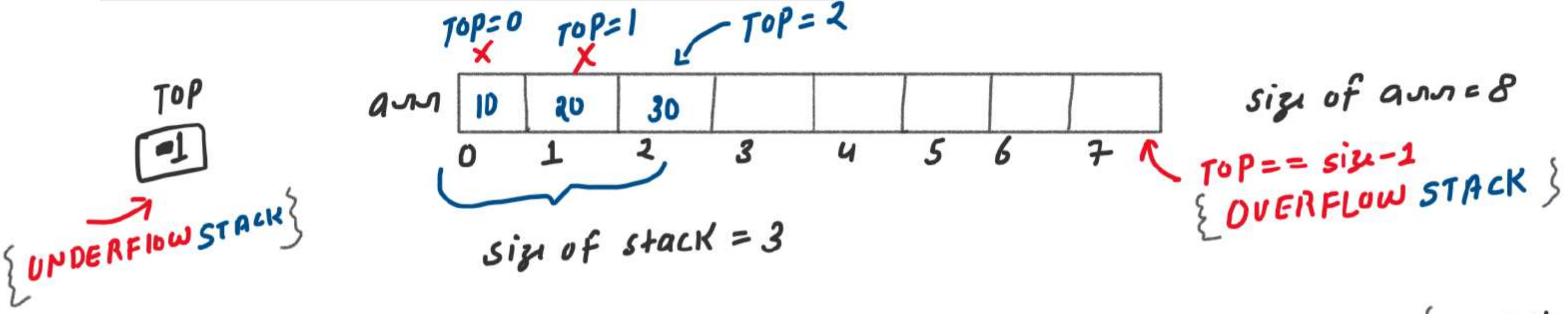
push() method: *insert new element to top*
pop() method: *delete element from top*
empty() method: *return true or false value*
top() method: *return top element*
size() method: *return size of stack*



OUTPUT:

```
Size of stack: 3
Stack is not empty
Top element: 2
```

2. Implementation of a stack using dynamic array



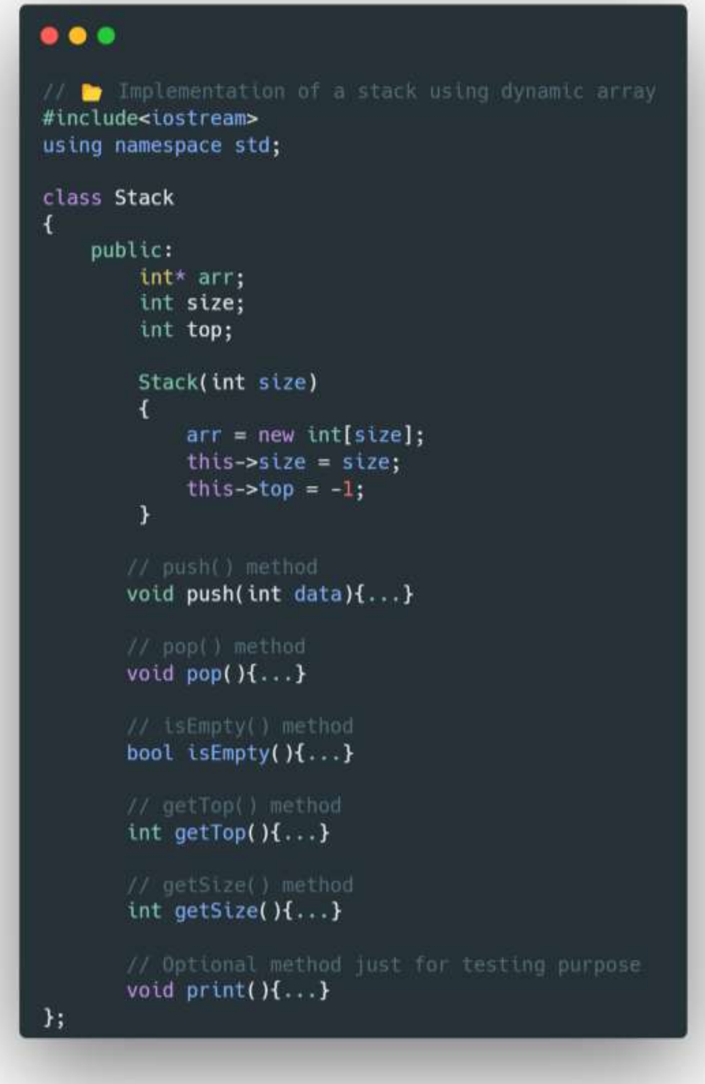
```
push() {
    top++;
    arr[TOP]=data;
}
```

```
pop() {
    TOP--;
}
```

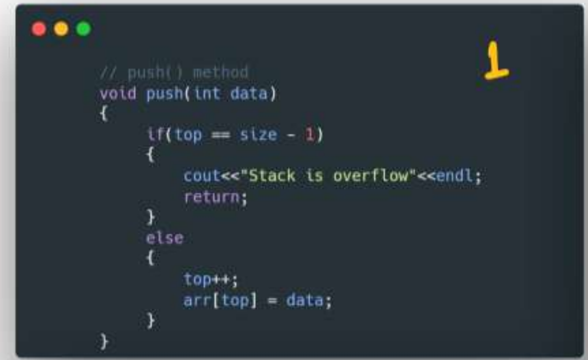
```
isEmpty() {
    if(top == -1)
        ↳ Empty
}
```

```
getTop() {
    return arr[TOP];
}
```

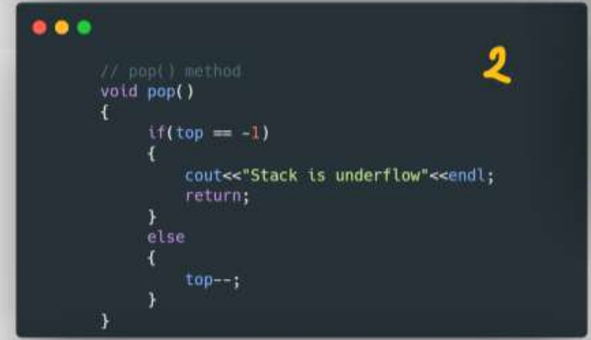
```
getSize() {
    return
        TOP+1;
}
```



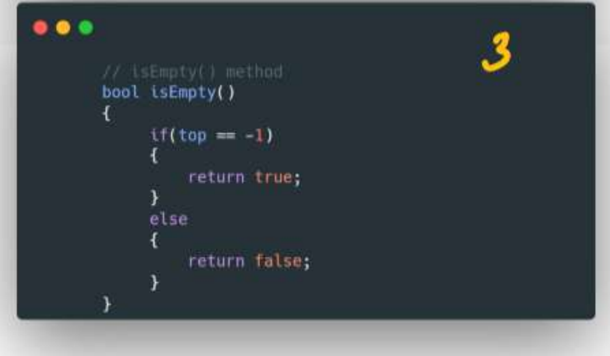
```cpp
// 📁 Implementation of a stack using dynamic array
#include<iostream>
using namespace std;

class Stack
{
    public:
        int* arr;
        int size;
        int top;

        Stack(int size)
        {
            arr = new int[size];
            this->size = size;
            this->top = -1;
        }

       // push() method
       void push(int data){...}

       // pop() method
       void pop(){...}

       // isEmpty() method
       bool isEmpty(){...}

       // getTop() method
       int getTop(){...}

       // getSize() method
       int getSize(){...}

       // Optional method just for testing purpose
       void print(){...}
};
```



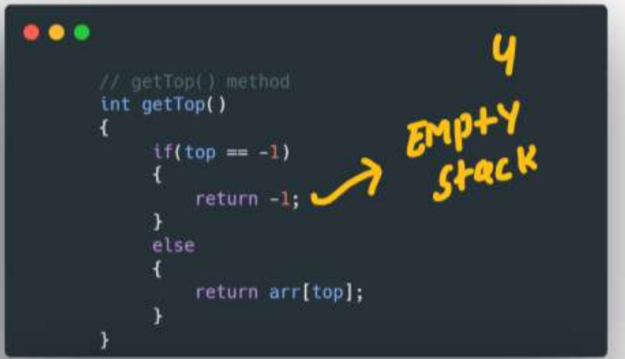
```cpp
// push() method
void push(int data)
{
    if(top == size - 1)
    {
        cout<<"Stack is overflow"<<endl;
        return;
    }
    else
    {
        top++;
        arr[top] = data;
    }
}
```



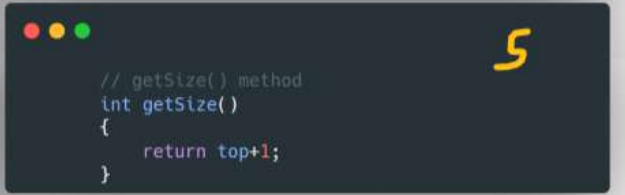
```cpp
// pop() method
void pop()
{
    if(top == -1)
    {
        cout<<"Stack is underflow"<<endl;
        return;
    }
    else
    {
        top--;
    }
}
```



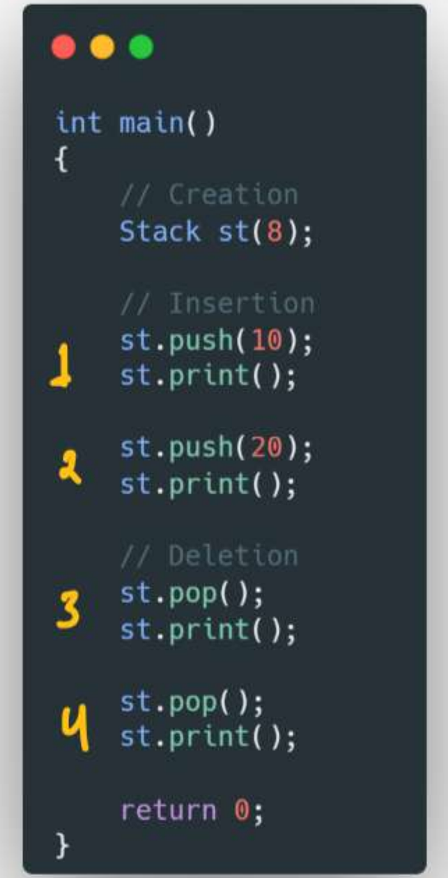
```cpp
// isEmpty() method
bool isEmpty()
{
    if(top == -1)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```



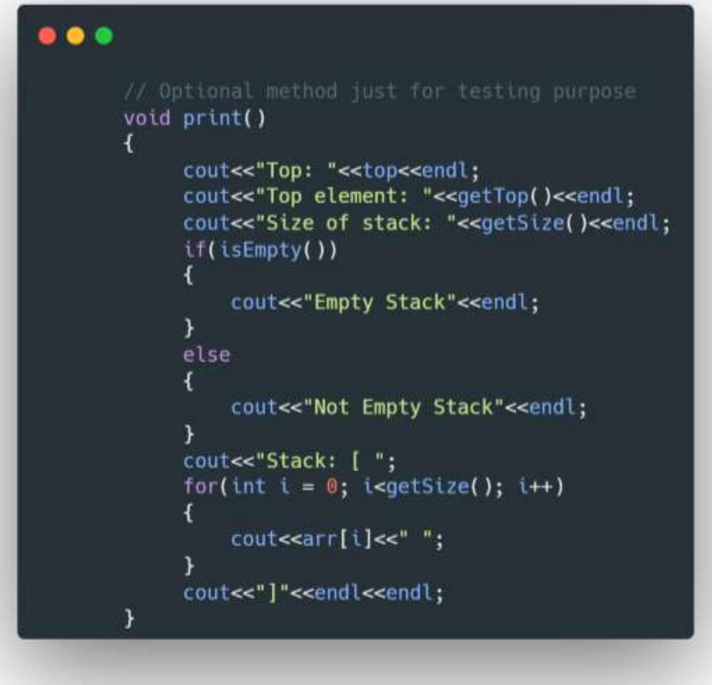
```cpp
// getTop() method
int getTop()
{
    if(top == -1)
    {
        return -1;
    }
    else
    {
        return arr[top];
    }
}
```



```cpp
// getSize() method
int getSize()
{
    return top+1;
}
```



```
int main()
{
    // Creation
    Stack st(8);

    // Insertion
1   st.push(10);
    st.print();

2   st.push(20);
    st.print();

    // Deletion
3   st.pop();
    st.print();

4   st.pop();
    st.print();

    return 0;
}
```



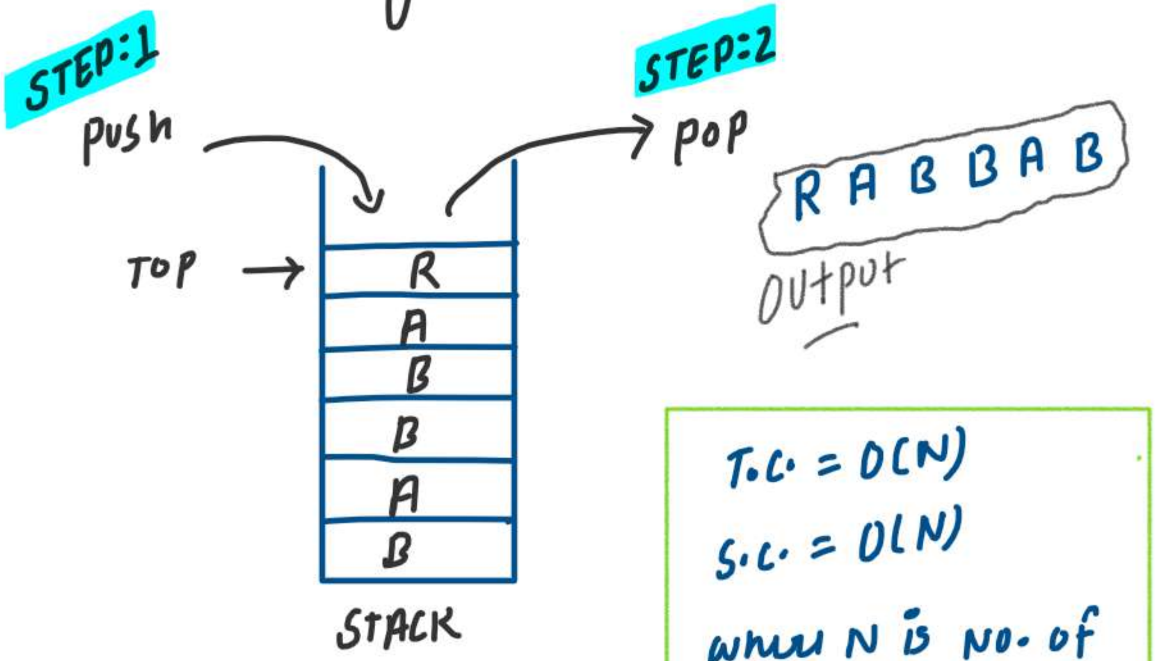
```
// Optional method just for testing purpose
void print()
{
    cout<<"Top: "<<top<<endl;
    cout<<"Top element: "<<getTop()<<endl;
    cout<<"Size of stack: "<<getSize()<<endl;
    if(isEmpty())
    {
        cout<<"Empty Stack"<<endl;
    }
    else
    {
        cout<<"Not Empty Stack"<<endl;
    }
    cout<<"Stack: [ ";
    for(int i = 0; i<getSize(); i++)
    {
        cout<<arr[i]<<" ";
    }
    cout<<"]"<<endl<<endl;
}
```

OUTPUT:

```
  Top: 0
  Top element: 10
1 Size of stack: 1
  Not Empty Stack
  Stack: [ 10 ]

  Top: 1
  Top element: 20
2 Size of stack: 2
  Not Empty Stack
  Stack: [ 10 20 ]

  Top: 0
  Top element: 10
3 Size of stack: 1
  Not Empty Stack
  Stack: [ 10 ]

  Top: -1
  Top element: -1
4 Size of stack: 0
  Empty Stack
  Stack: [ ]
```

📂 Program 1: Reverse string using stack

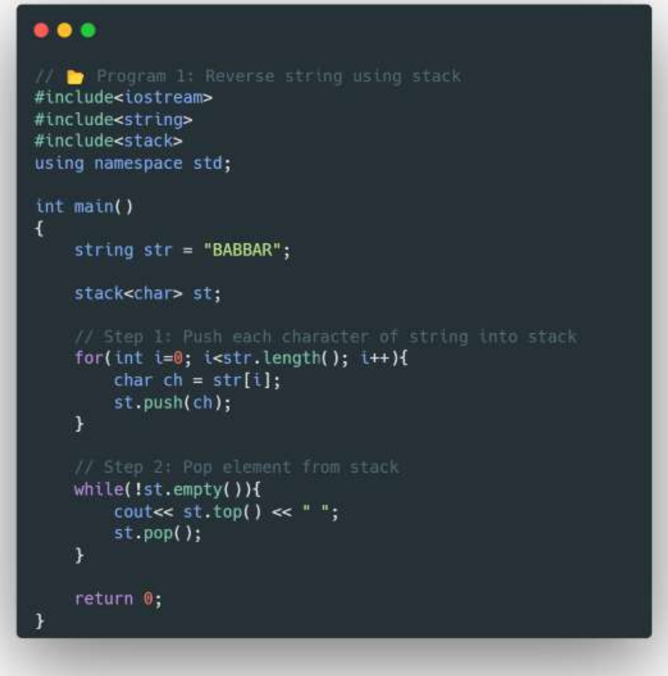
String = " BABBAR "



T.C. = O(N)

S.C. = O(N)

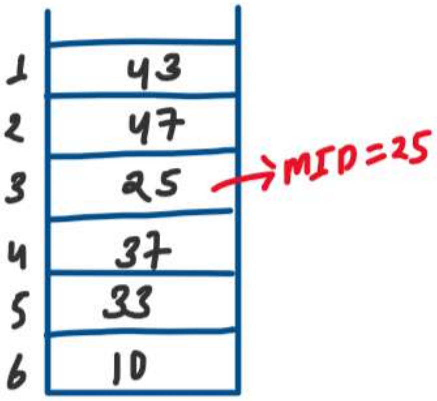
where N is No. of character.



```cpp
// 📂 Program 1: Reverse string using stack
#include<iostream>
#include<string>
#include<stack>
using namespace std;

int main()
{
    string str = "BABBAR";

    stack<char> st;

    // Step 1: Push each character of string into stack
    for(int i=0; i<str.length(); i++){
        char ch = str[i];
        st.push(ch);
    }

    // Step 2: Pop element from stack
    while(!st.empty()){
        cout<< st.top() << " ";
        st.pop();
    }

    return 0;
}
```

📂 Program 2: Middle element of a stack

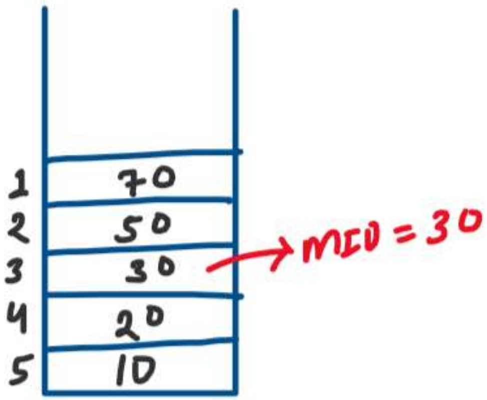
EVEN



Size of stack = 6

$$\text{Mid} = \frac{6}{2} = 3$$

ODD



Size of stack = 5

$$\text{Mid} = \left\lfloor \frac{5}{2} \right\rfloor + 1 = 3$$

Approach 1: Recursion and backtracking

T.C. = O(N)

S.C. = O(N)

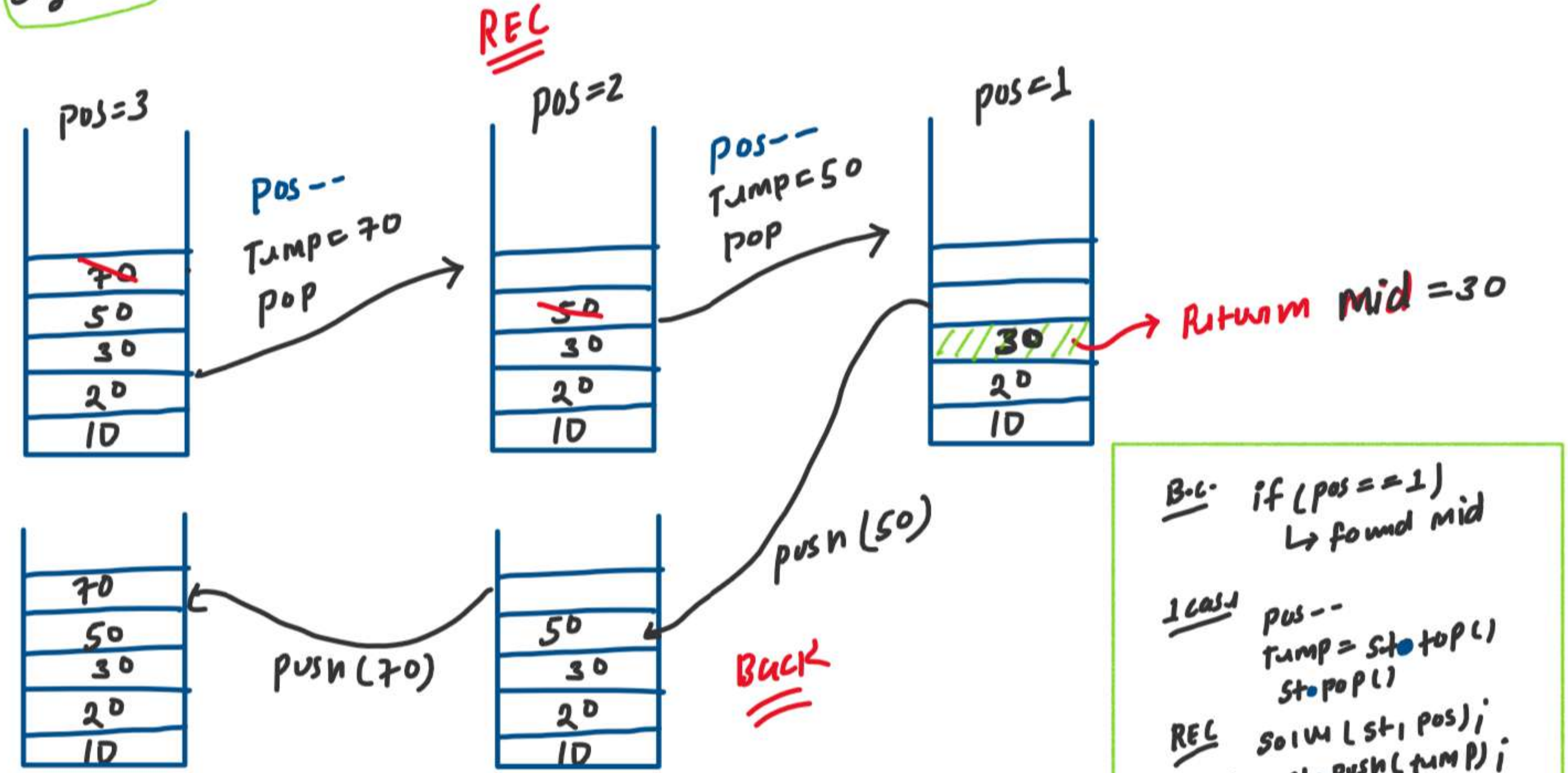
where N is no. of Elements of stack

STACK ODD

Size = 5

$pos = \frac{5}{2} + 1$

$= 3$

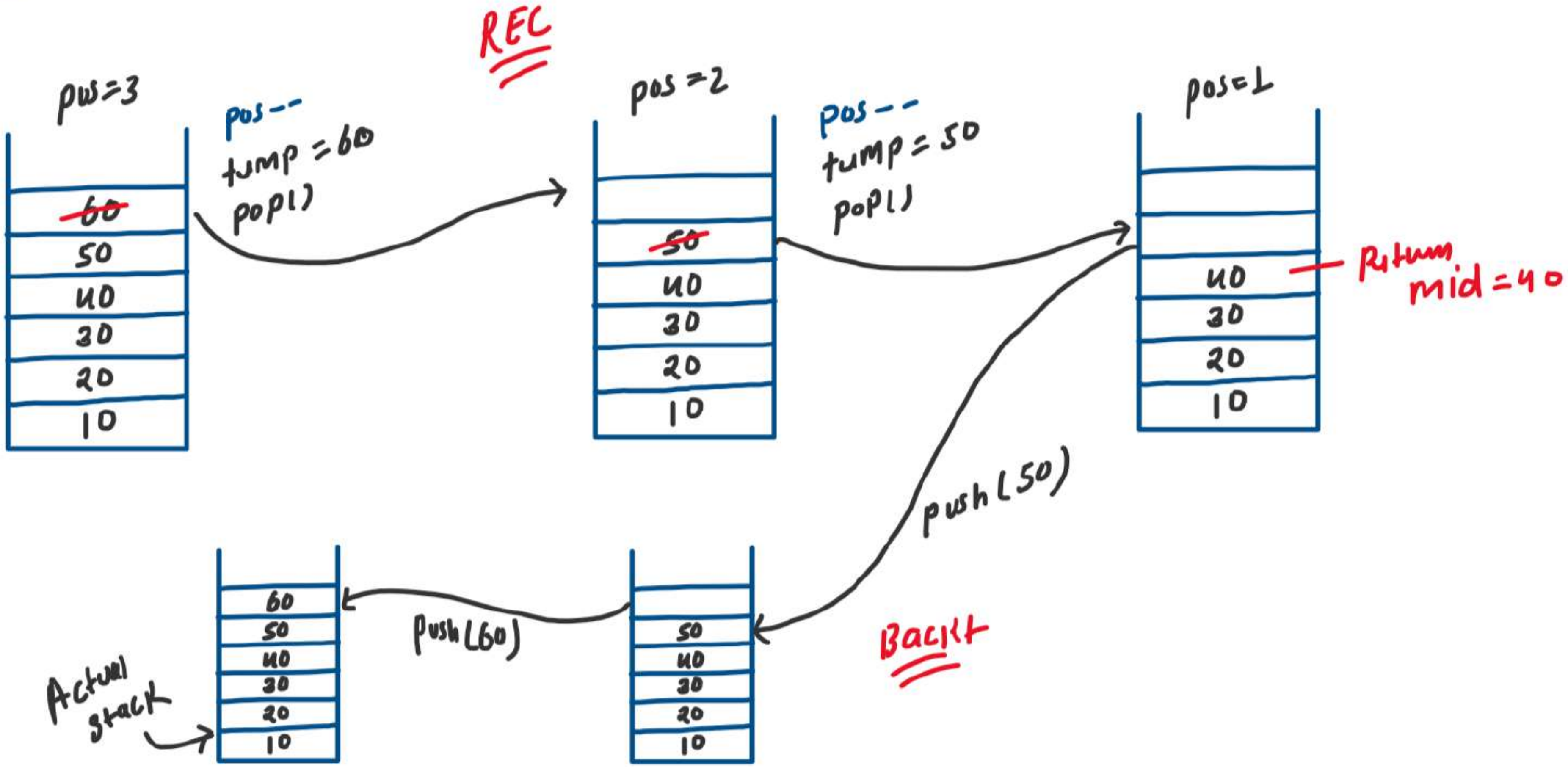


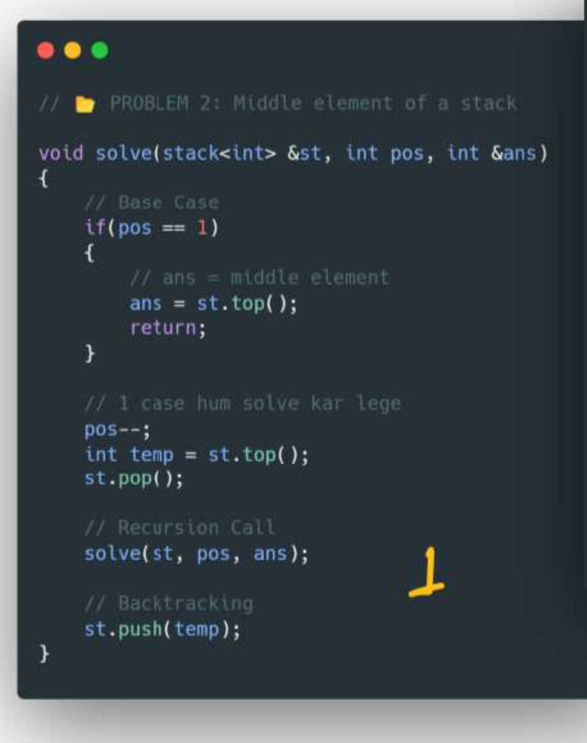
```
B.C.     if (pos == 1)
           ↳ found mid

1 case   pos--
         Temp = St.top()
         St.pop()

REC      Solve(st, pos);
Back     St.push(temp);
```

STACK Even size = 6

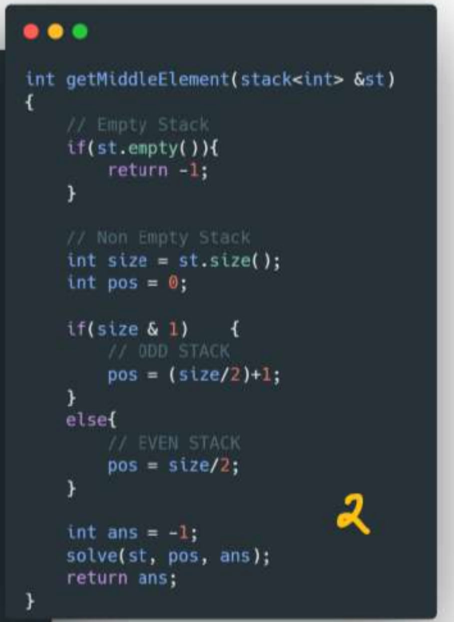
pos = $\frac{6}{2}$

= 3





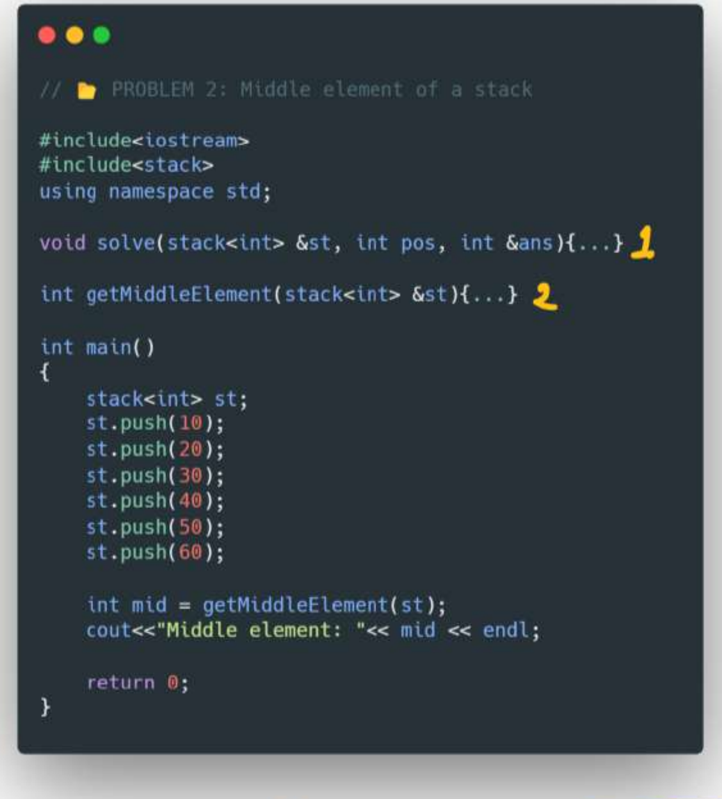
```
// 📁 PROBLEM 2: Middle element of a stack

void solve(stack<int> &st, int pos, int &ans)
{
    // Base Case
    if(pos == 1)
    {
        // ans = middle element
        ans = st.top();
        return;
    }

    // 1 case hum solve kar lege
    pos--;
    int temp = st.top();
    st.pop();

    // Recursion Call
    solve(st, pos, ans);

    // Backtracking
    st.push(temp);
}
```



```
int getMiddleElement(stack<int> &st)
{
    // Empty Stack
    if(st.empty()){
        return -1;
    }

    // Non Empty Stack
    int size = st.size();
    int pos = 0;

    if(size & 1)    {
        // ODD STACK
        pos = (size/2)+1;
    }
    else{
        // EVEN STACK
        pos = size/2;
    }

    int ans = -1;
    solve(st, pos, ans);
    return ans;
}
```

T.C. = O(N/2) ⇒ O(N)
S.C. = O(N)

T.C. = O(1)
S.C. = O(1)

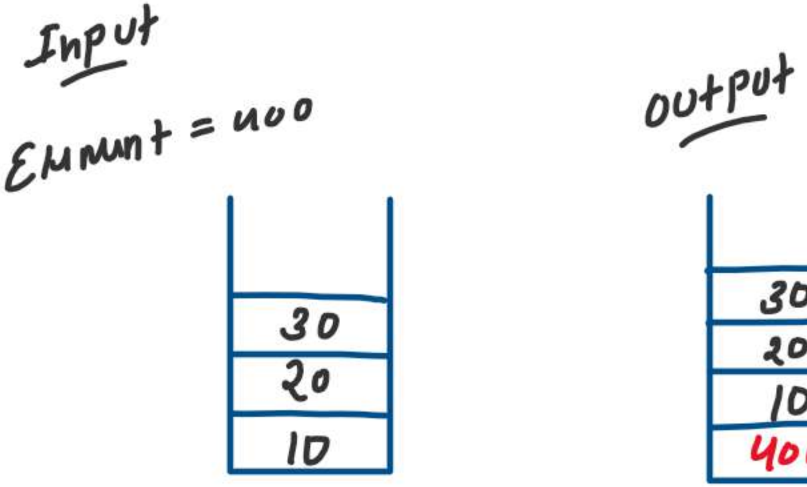


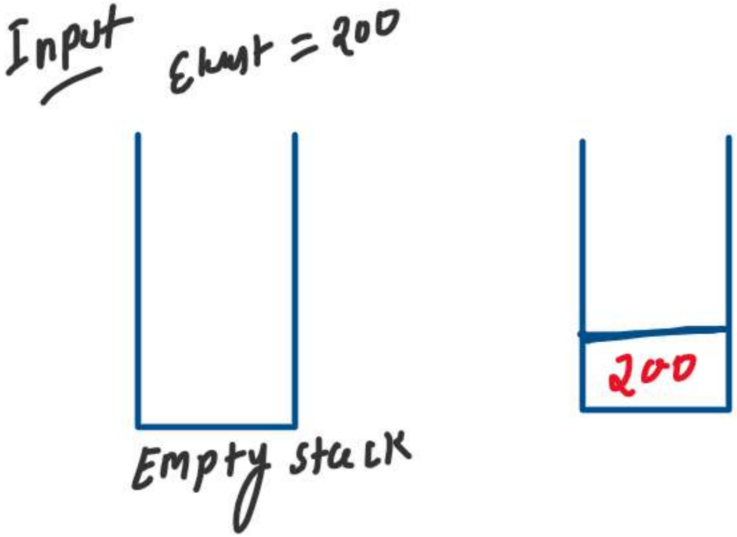
```
// 📁 PROBLEM 2: Middle element of a stack

#include<iostream>
#include<stack>
using namespace std;

void solve(stack<int> &st, int pos, int &ans){...}

int getMiddleElement(stack<int> &st){...}

int main()
{
    stack<int> st;
    st.push(10);
    st.push(20);
    st.push(30);
    st.push(40);
    st.push(50);
    st.push(60);

    int mid = getMiddleElement(st);
    cout<<"Middle element: "<< mid << endl;

    return 0;
}
```

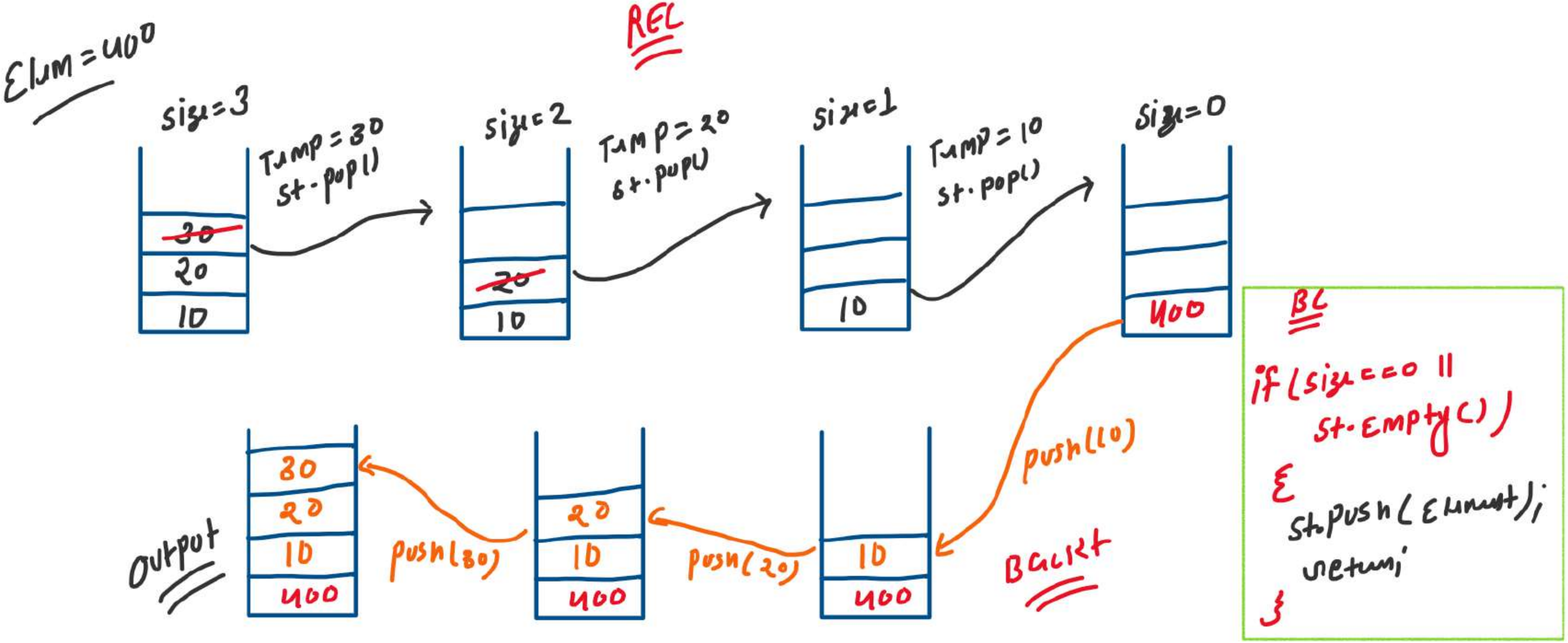
Total T.C = O(N)
S.C. = O(N)

📂 Program 3: Insert at bottom of a stack

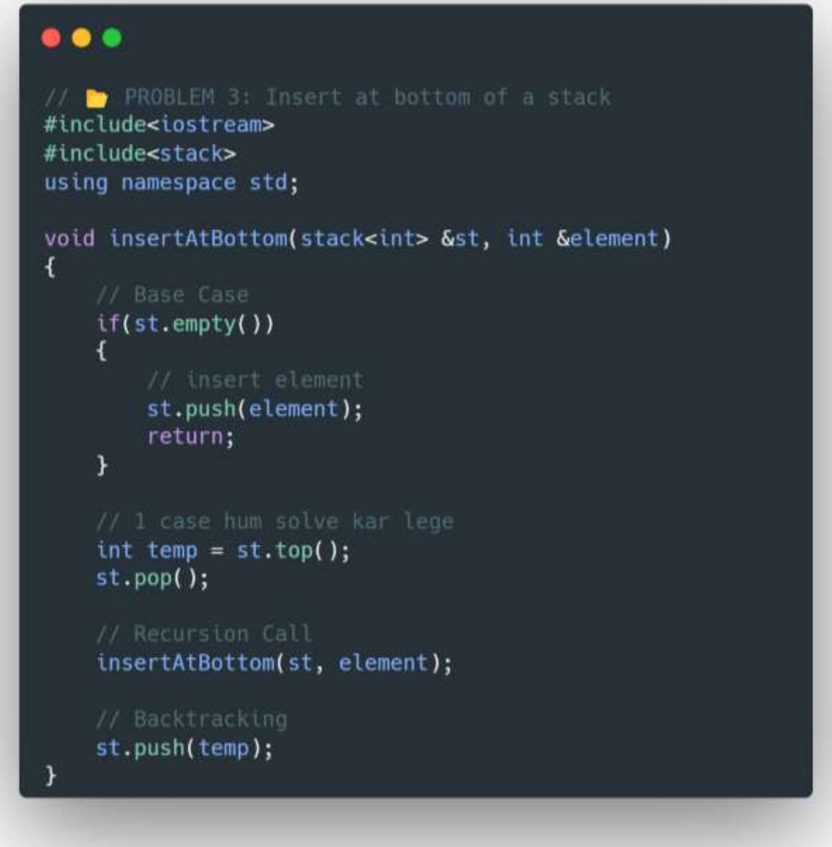
Approach 1: Recursion and backtracking







```
if (size==0 ||
      st.Empty())
{
  st.push(Element);
  return;
}
```



```
// 📂 PROBLEM 3: Insert at bottom of a stack
#include<iostream>
#include<stack>
using namespace std;

void insertAtBottom(stack<int> &st, int &element)
{
    // Base Case
    if(st.empty())
    {
        // insert element
        st.push(element);
        return;
    }

    // 1 case hum solve kar lege
    int temp = st.top();
    st.pop();

    // Recursion Call
    insertAtBottom(st, element);

    // Backtracking
    st.push(temp);
}
```

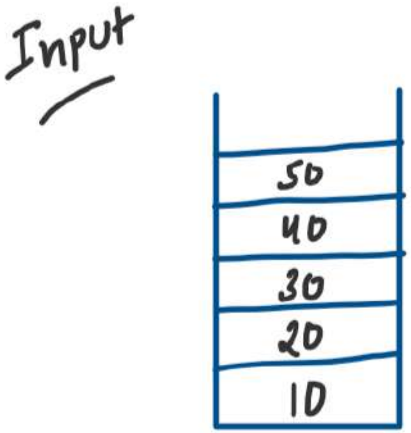
T.C. = O(N)

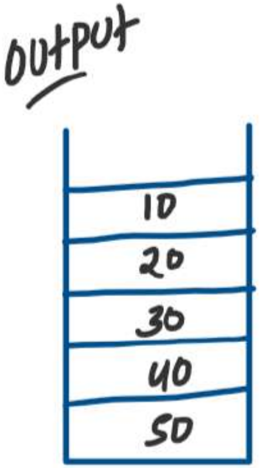
S.C. = O(N)

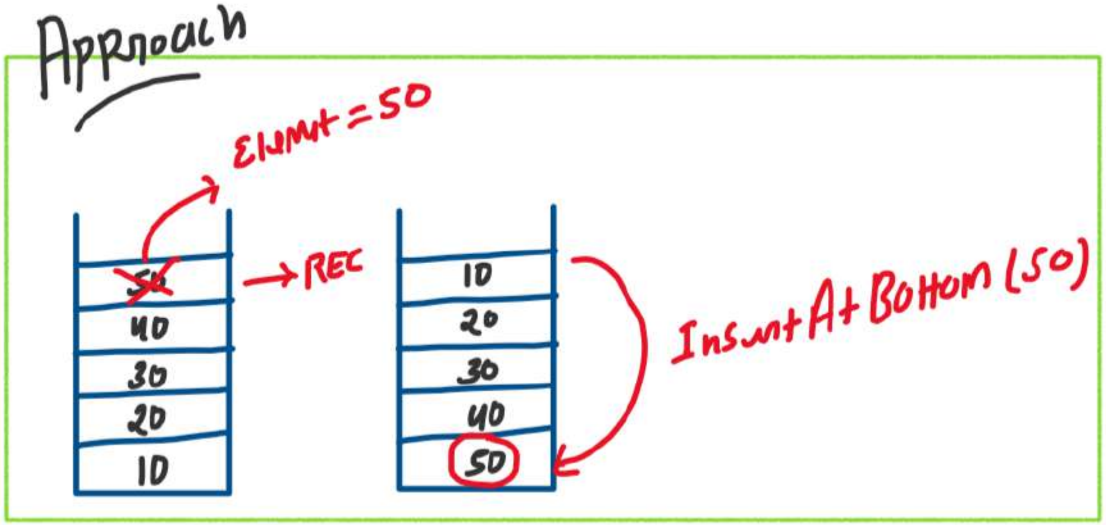
where N is no. of Elements of stack

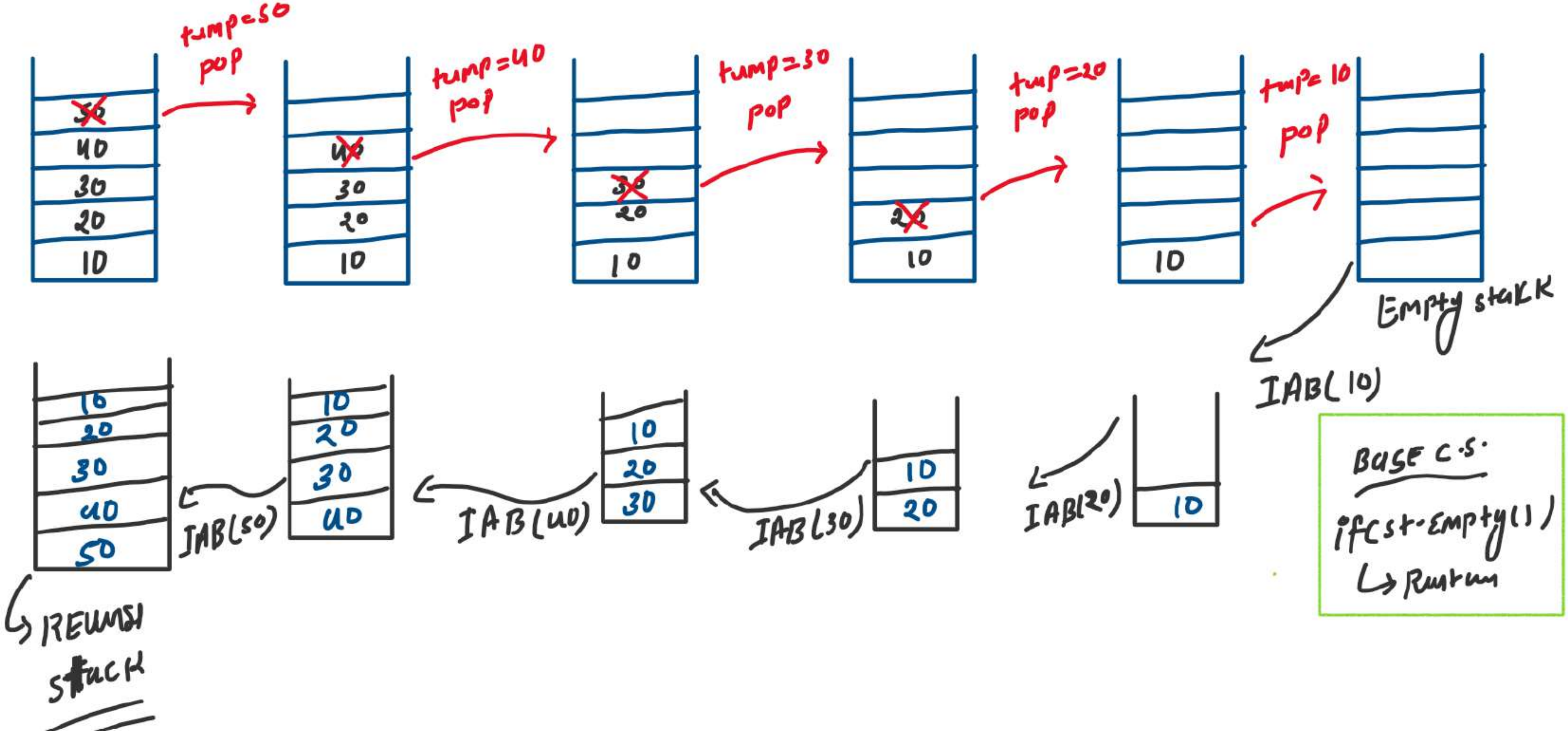
Program 4: Reverse a stack

Approach 1: Recursion and backtracking

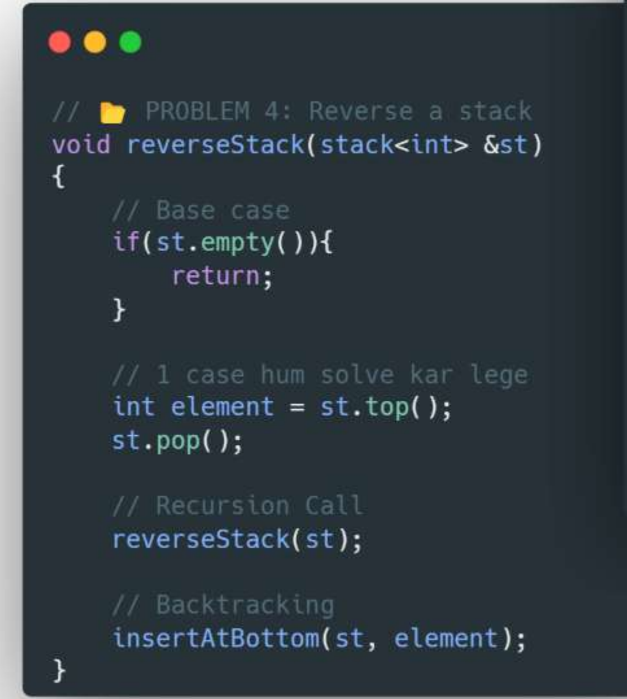








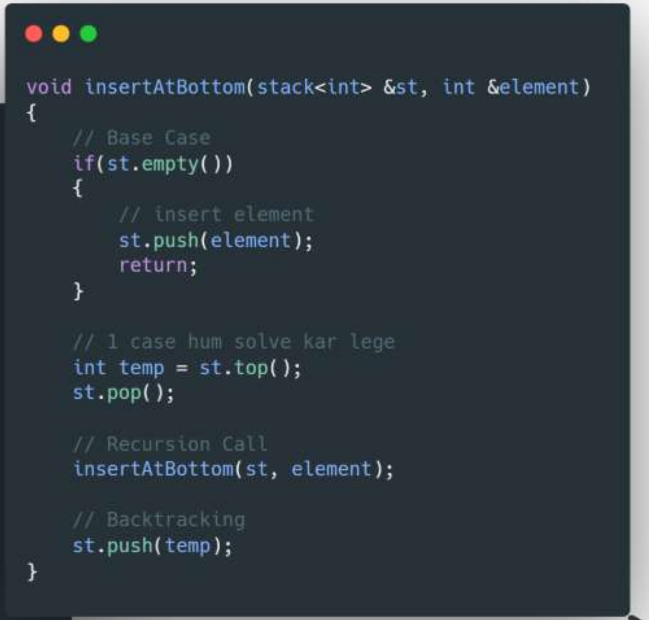
temp=50
pop
~~50~~
40
30
20
10
temp=40
pop
~~40~~
30
20
10
temp=30
pop
~~30~~
20
10
temp=20
pop
~~20~~
10
temp=10
pop
10
Empty stack
IAB(10)
Base C.S.
if(st.Empty())
↳ Return
10
IAB(20)
10
20
IAB(30)
10
20
30
IAB(40)
10
20
30
40
IAB(50)
10
20
30
40
50
↳ REVERSE stack



```
// 📂 PROBLEM 4: Reverse a stack
void reverseStack(stack<int> &st)
{
    // Base case
    if(st.empty()){
        return;
    }

    // 1 case hum solve kar lege
    int element = st.top();
    st.pop();

    // Recursion Call
    reverseStack(st);

    // Backtracking
    insertAtBottom(st, element);
}
```

T.C. = O(N)

S.C. = O(N)



```
void insertAtBottom(stack<int> &st, int &element)
{
    // Base Case
    if(st.empty())
    {
        // insert element
        st.push(element);
        return;
    }

    // 1 case hum solve kar lege
    int temp = st.top();
    st.pop();

    // Recursion Call
    insertAtBottom(st, element);

    // Backtracking
    st.push(temp);
}
```

T.C. = O(N)

S.C. = O(N)

Total

T.C. = O(N) * O(N)

$= O(N^2)$

S.C. = O(N)

when N is No. of Elements.

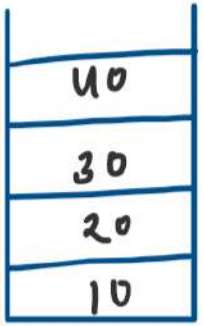
📂 Program 5: Insert element in a sorted stack

Approach 1: Recursion and backtracking

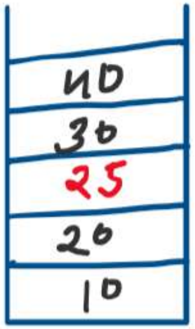
EX1

Input

Element = 25



output



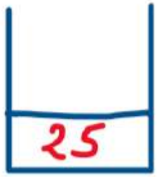
Ex2

Important Test CASE

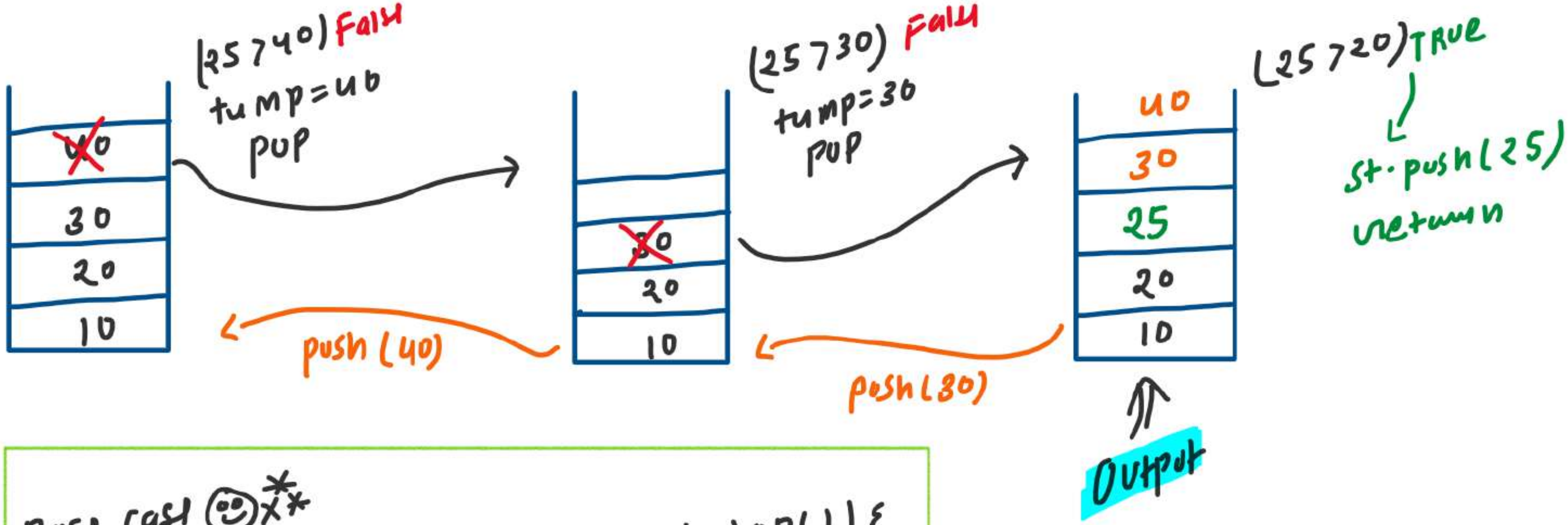
Input Elemt = 5

Empty stack

output

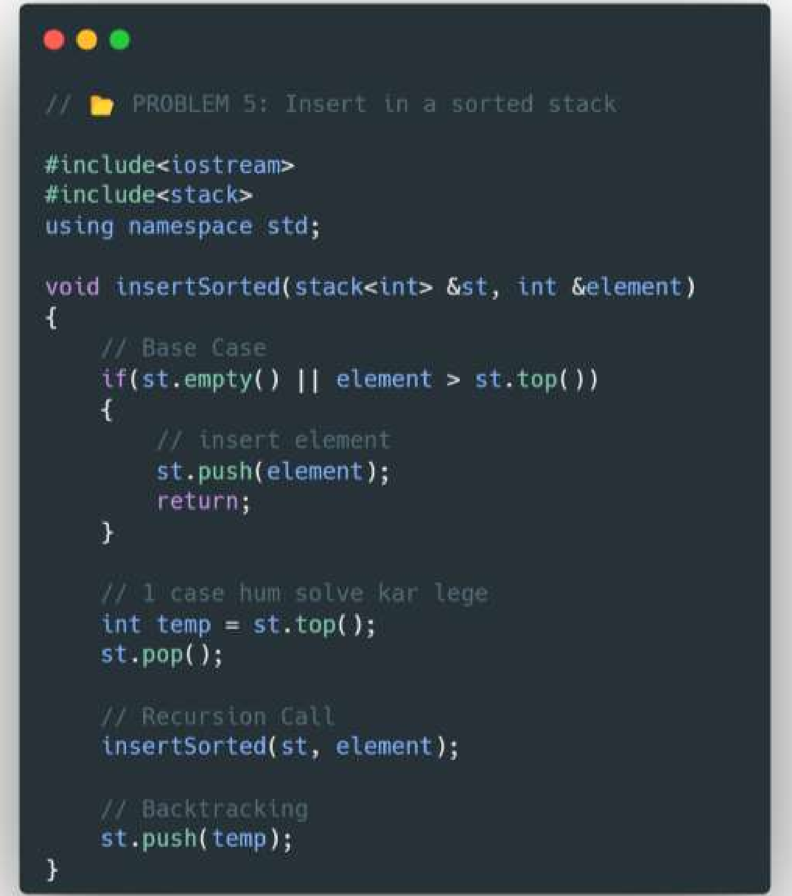


Elem=25



Base case ☺ ***

```
if ( st.empty() || Element > st.top()){
    └──> st.push(Element) ;
           return ;
     }
```



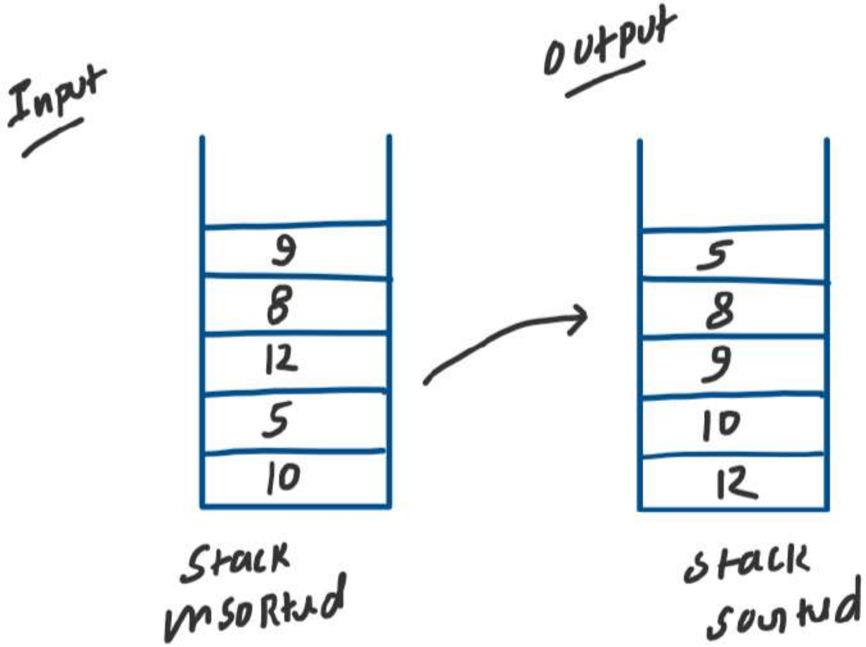
```cpp
// 📂 PROBLEM 5: Insert in a sorted stack

#include<iostream>
#include<stack>
using namespace std;

void insertSorted(stack<int> &st, int &element)
{
    // Base Case
    if(st.empty() || element > st.top())
    {
        // insert element
        st.push(element);
        return;
    }

    // 1 case hum solve kar lege
    int temp = st.top();
    st.pop();

    // Recursion Call
    insertSorted(st, element);

    // Backtracking
    st.push(temp);
}
```

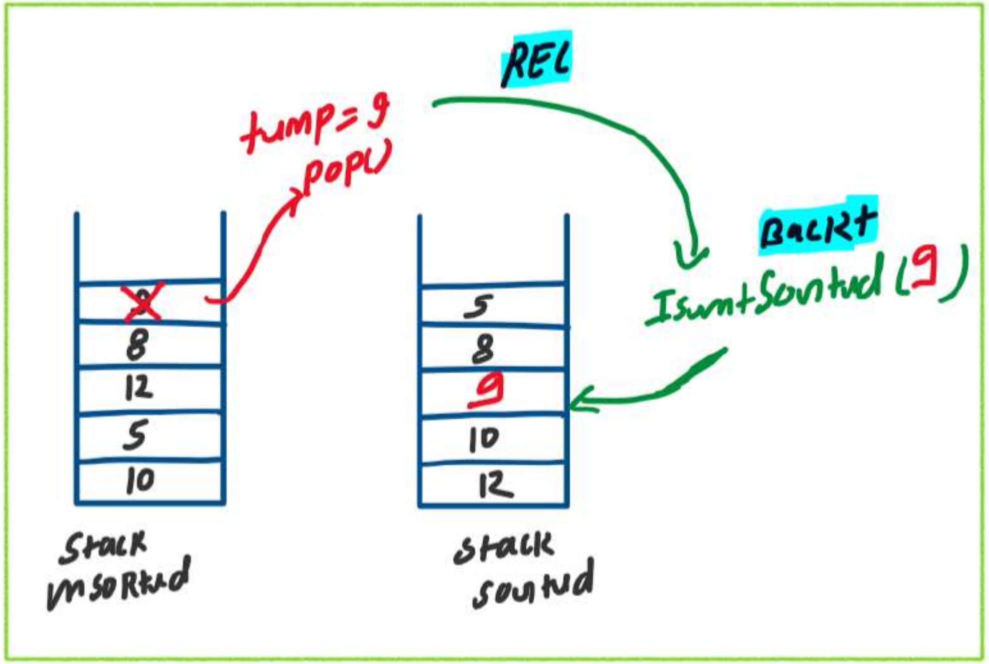
T.C. = O(N)

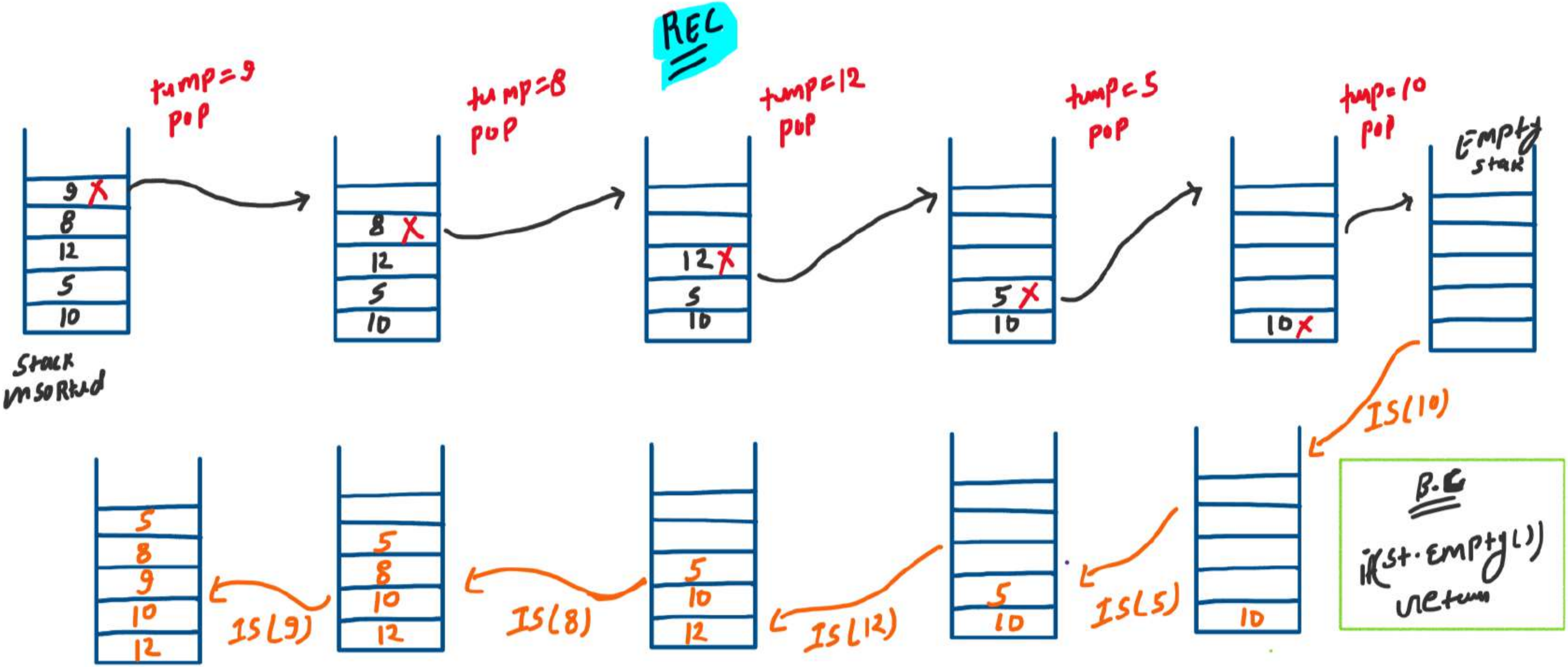
S.C. = O(N)

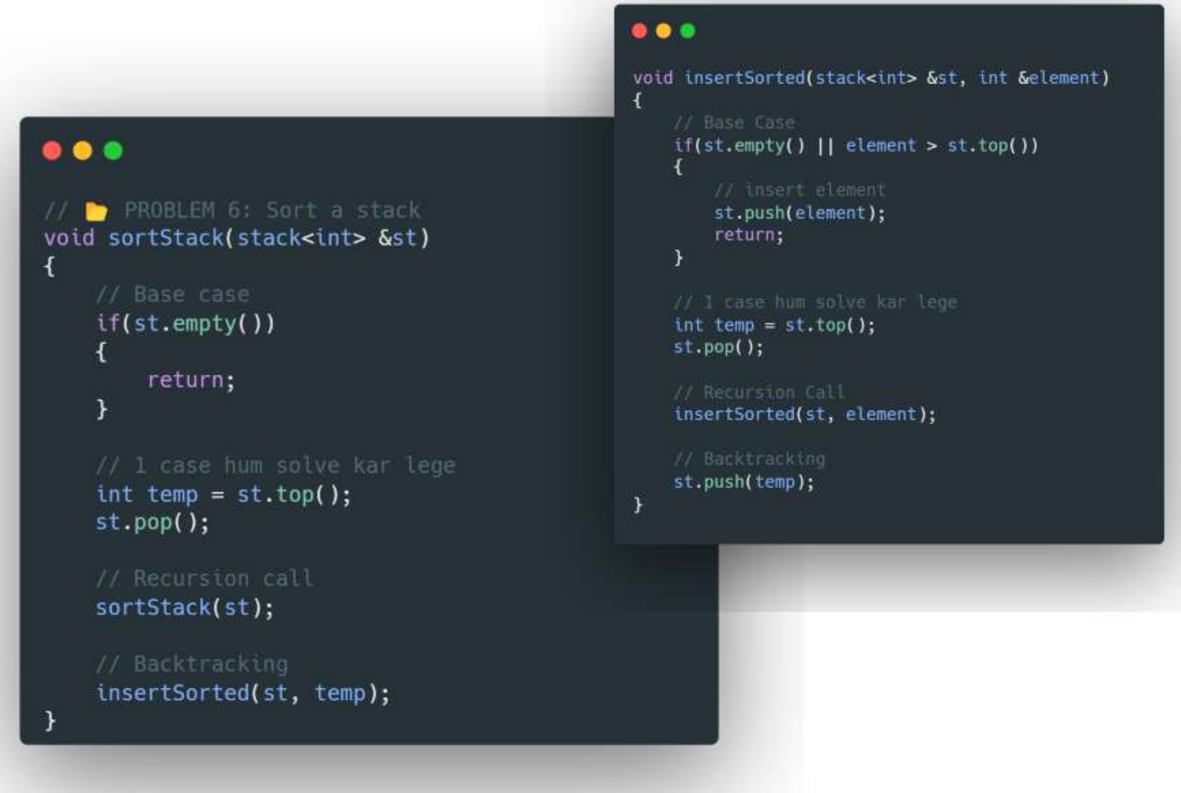
where N is No. of Elements of stack

📂 Program 6: Sort a stack






REC
temp=9
pop
temp=8
pop
temp=12
pop
temp=5
pop
temp=10
pop
Empty stack
9
8
12
5
10
Stack unsorted
8
12
5
10
12
5
10
5
10
10
IS(10)
10
IS(5)
5
10
IS(12)
5
10
12
IS(8)
5
8
10
12
IS(9)
5
8
9
10
12
B.C
if(st.empty())
return



```
// 📁 PROBLEM 6: Sort a stack
void sortStack(stack<int> &st)
{
    // Base case
    if(st.empty())
    {
        return;
    }

    // 1 case hum solve kar lege
    int temp = st.top();
    st.pop();

    // Recursion call
    sortStack(st);

    // Backtracking
    insertSorted(st, temp);
}
```

```
void insertSorted(stack<int> &st, int &element)
{
    // Base Case
    if(st.empty() || element > st.top())
    {
        // insert element
        st.push(element);
        return;
    }

    // 1 case hum solve kar lege
    int temp = st.top();
    st.pop();

    // Recursion Call
    insertSorted(st, element);

    // Backtracking
    st.push(temp);
}
```

T.C = $O(N^2)$

Because of Nested Recursion call

S.C = $O(N)$

Where N is No. of Elements of stack

READ MORE

https://cplusplus.com/reference/stack/stack/

https://en.cppreference.com/w/cpp/container/stack