Alternative Approach of Stack – Time Complexity

1. ***ISFULL function to check the stack is full or not as discussed before*:**

```
int isFull(Stack st)
{
    if (st.top == st.size - 1)
    {
        return 1;
    }
    return 0;
}
```

2. ***Double Stack function to double the stack if the stack is full* :**

```
void doubleStack(Stack *st)
{
    st->size = st->size * 2;
    st->s = (int *)realloc(st->s, st->size *
sizeof(int));
}
```

ReAlloc will reallocate the memory again with a new size of the array created.

3. *In Push , when the element is pushed in Stack then its checked that if its full then double the stack , else just increment the top and push the element.*

```
void push(Stack *st, int x)
{

    if (isFull(*st))
    {
        doubleStack(st);
        st->top++;
        st->s[st->top] = x;
        return;
    }

    else{

        st->top++;
        st->s[st->top] = x;
    }


}
```

Now, Coming back to execution:

At first we create object of Stack and declare a variable to decide what should be capacity of the stack in main function.

```cpp
int main()
{

    Stack st;
    int cap;
    cout << "Enter the capacity of the stack" << endl;

    cin >> cap;

}
```

4. We know during push it will take 2n elements as as the size gets doubled:

```
int main()
{

int elem;
  for (int i = 0; i < 2* cap; i++)
   {
   cout << "Enter the element to be pushed :" << endl;
   cin >> elem;
   push(&st, elem);
    }

}
```

5. $Now\ pop\ function\ must\ pop\ the\ first\ set\ of\ elements\ that\ are\ (inserted\ before\ the\ array\ got\ doubled\ ):$

```
int pop(Stack *st)
{

    if (isEmpty(*st))
    {
        cout << "Stack Underflow" << endl;
        return -1;

    }

    return st->s[st->top--];
}

int main()

{

if (!isEmpty(st))
    {
       for (int i = 0; i < cap; i++)
          {
   cout << "Popped element is: " << pop(&st) << endl;
           }


      }
       else
       {
           cout << "Stack Underflow" << endl;

        }
}
```

Else , every other functions are all same earlier discussed .

Now coming to Time Complexity

$\mathbf{1.}$ ***Time Complexity of Push Operation***:

Ans: ***As discussed earlier, Push Operation here takes Worst Case Complexity***: $\boldsymbol{T(n) = O(n)}$,

where as, amortized time complexity $\boldsymbol{= O(1)}$.

Q)Why we need amortized time complexity?

Ans)When an operation sometimes requires copying many elements, hence we need to resize then amortized analysis becomes necessary.

$2.$ ***Time Complexity of Pop Operation***:

Ans : ***As we are poping from each stack each at*** $O(1)$ ***time , now we are using for loop making***: $n \times O(1)$ ***time*** $= O(n)$ ***time at worst case***.

Q)Does here amortization time complexity require?

Ans: ***No , as no copying of elements and no resizing take place***.

And Other Time Complexity of all types of Operations remains same that is $O(1)$.

Now coming to Space Complexity

As the stack takes `n` times push takes n units of space in memory , hence space complexity $= T(n) = O(n)$.

Hence, we sum up:

Space complexity	$O(n)$
Creation of Stack	$O(1)$
Push	$O(1)$ *Average*
Pop()	*pop func*: $O(1)$ *, as for loop used to pop `n` times in main func making ,* $n \times O(1) = O(n)$.
Peek()	$O(1)$
IsEmpty()	$O(1)$
IsFull()	$O(1)$
DeleteStack()	$O(1)$

_ _ _ _ _ _ _ _ _ _ _ _ _ _