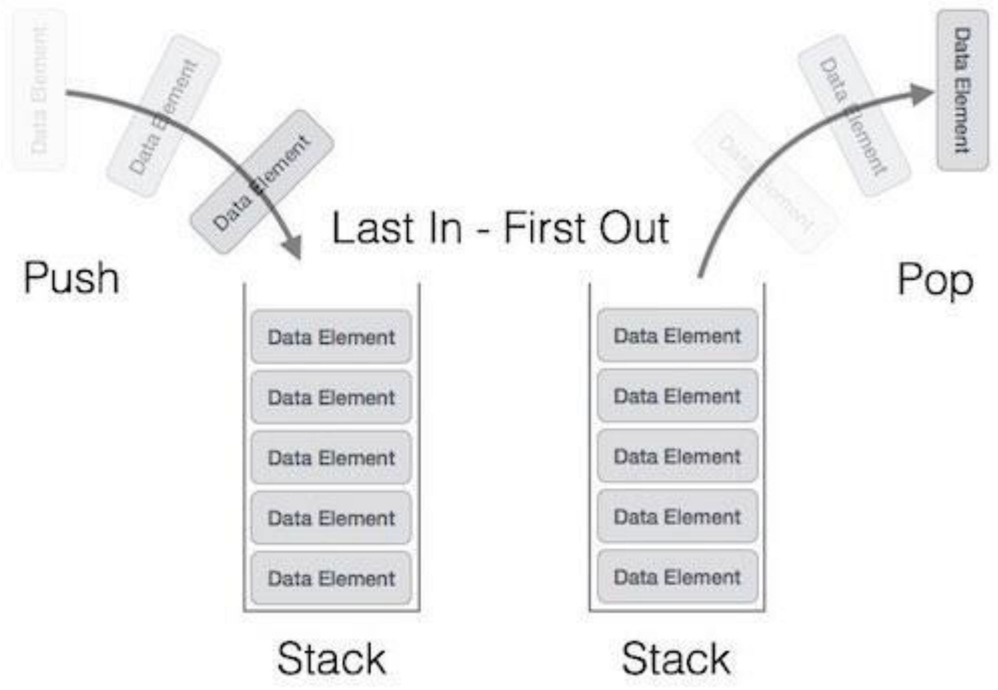
Stack :

- A stack is an Abstract Data Type (ADT), commonly used in most programming languages.
- It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

- A real-world stack allows operations at one end only.
    - For example, we can place or remove a card or plate from the top of the stack only.
- At any given time, we can only access the top element of a stack.
- This feature makes it LIFO data structure.(Last-in-first-out)
    - we can say **FILO**(First in Last out
    - Element placed (inserted or added) last, is accessed first.
- A stack can be implemented by means of Array, Structure, Pointer, and Linked List.
- Stack can either be a fixed size one or it may have a sense of dynamic resizing.
- Stack using arrays is a fixed size stack , i.e. stack is an **ordered list** of **similar data type**.
- In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.



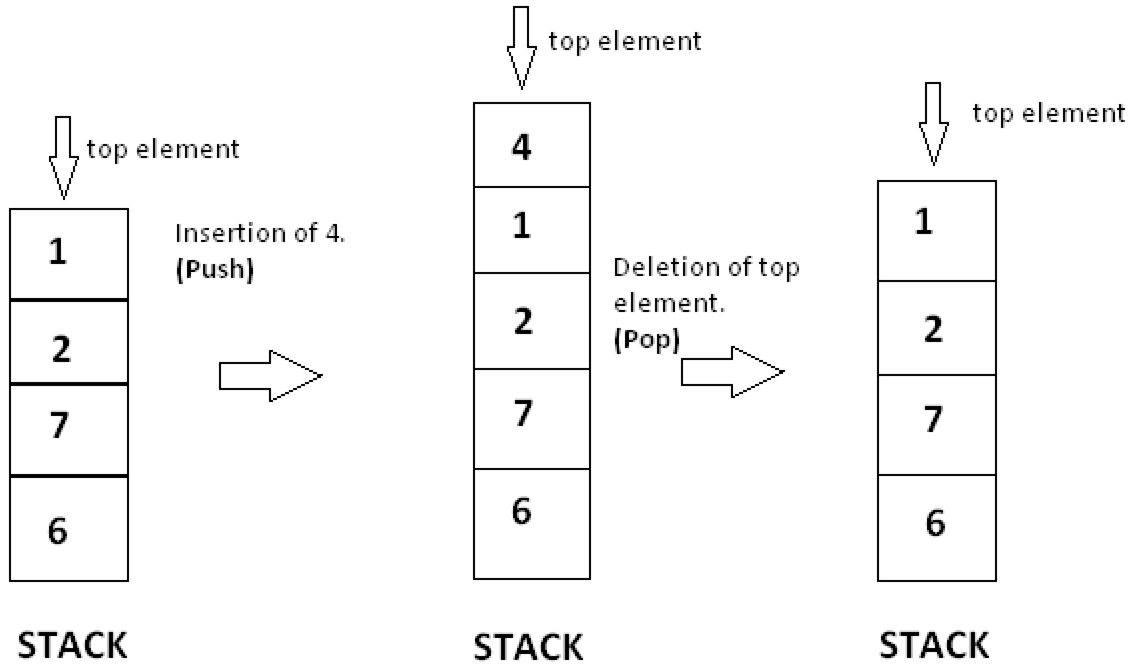
- Stack is said to be in **Overflow** state when it is completely full and is said to be in **Underflow** state if it is completely empty.

Basic Operations

Primary operations −

- **push()** − Pushing (storing) an element on the stack.
- **pop()** − Removing (accessing) an element from the stack.

PUSH and POP functions of Stack

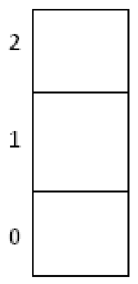


To use a stack efficiently, we need to check the status of stack as well. Additional Functions:

- **peek()** − get the top data element of the stack, without removing it.
- **isFull()** − check if stack is full.
- **isEmpty()** − check if stack is empty.

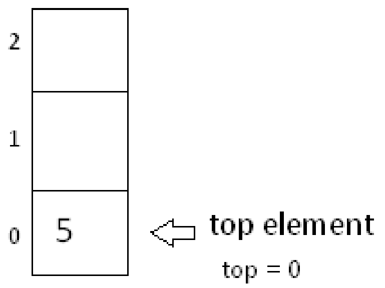
Push {st, value, size)

```
# include "mystack.h"
Void Push {st, value, size)
Int pop (st,size)
V1 = pop(st,size)
5,10,24             24,10,5
```

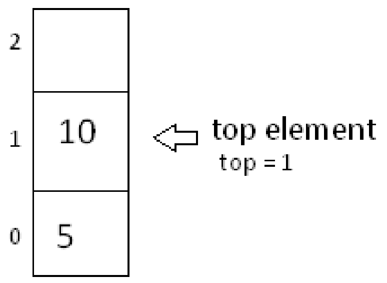
1) 

Initially **stack** is empty.
top = -1.

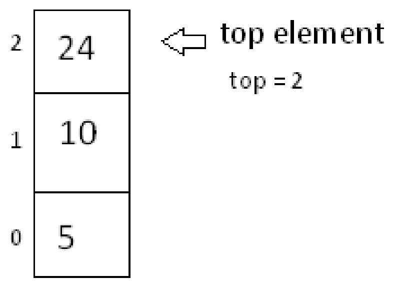
2) **push(stack, 5, 3)**



3) **push(stack, 10, 3)**



4) **push(stack, 24, 3)**

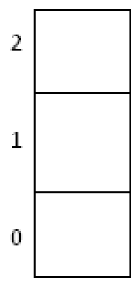


5) As **top = 2,** current size of stack is top+1 , i.e 3 . Now stack is full,as 3 is maximum size of stack

6) **push(stack, 12, 3)**

As ,stack is full ,it will show **OVERFLOW CONDTION!**

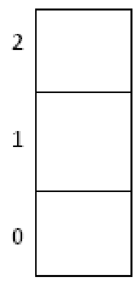
7) Deleting all elements from stack. ⇨ **pop(stack, 3)**
pop(stack, 3)
pop(stack, 3)



EMPTY STACK !!

top = -1

8) **pop(stack, 3)**



As **stack** is empty, further deleting will cause

UNDERFLOW CONDITION!

Stack operations may involve initializing the stack, using it and then de-initializing it.

Stack DS uses

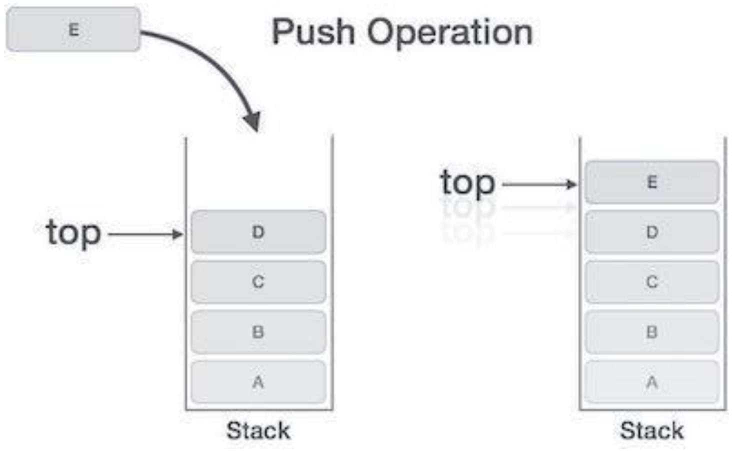
An array of size N

One pointer (TOP) : to point the last PUSHed data on the stack.

Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps −

- **Step 1** − Checks if the stack is full.
- **Step 2** − If the stack is full, produces an error and exit.
- **Step 3** − If the stack is not full, increments **top** to point next empty space.
- **Step 4** − Adds data element to the stack location, where top is pointing.
- **Step 5** − Returns success.



Struct stack{

Int s[10];

Int Top;

}

Algorithm for PUSH Operation

Top=-1

```
Push(Stack, data, MAX)
//Max = Size of array
//MAX =3

Begin
1.  If  (Stack.Top  >= MAX -1)     (top> Max-1)
            a. Print Overflow
            b. return null
      Endif
2.   Set stack.Top ← stack.Top + 1  .......stack.top=stack.top+1
3.   Stack.S[stack.Top] ← data
End
```

MAX	MAX-1	TOP	S	
3	**2**	**-1**	**{}**	
		0	**{10}**	**Push(stack,10,3)**
		1	**{10,5}**	**Push(stack,5,3)**
		2	**{10,5,9}**	**Push(stack, 9,3)**
			Overflow	**Push(stack,6,3)**

```
Push(Stack, data, MAX)
//Max = Size of array
//MAX =3

Begin
4.  If  (Stack.Top  > MAX -1)     (top> Max-1)
            a. Print Overflow
            b. return null
      Endif
5.   Stack.S[stack.Top] ← data
6.   Set stack.Top ← stack.Top + 1

End
```

MAX	MAX-1	TOP	S	
3	**2**	**0**	**{}**	
		1	**{10}**	**Push(stack,10,3)**
		2	**{10,5}**	**Push(stack,5,3)**
		3	**{10,5,9}**	**Push(stack, 9,3)**
			Overflow	**Push(stack,6,3)**

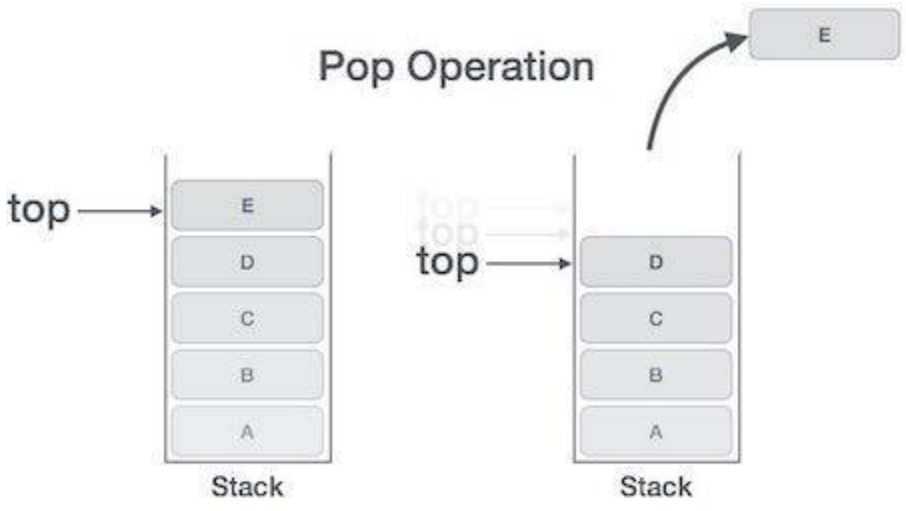
Change the algorithm Discussed with Initial TOP= 0

Pop Operation

- Accessing the content while removing it from the stack, is known as a Pop Operation.
- In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value.

A Pop operation may involve the following steps −

- **Step 1** − Checks if the stack is empty.
- **Step 2** − If the stack is empty, produces an error and exit.
- **Step 3** − If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** − Decreases the value of top by 1.
- **Step 5** − Returns success.



Algorithm for Pop Operation

```
Pop( Stack)

Begin
    1. Top= top-1
    2. If Top = -1
        a. print “Under flow”
        b. return null
       Endif
    3. Data ← Stack[Top]
    4. Set Top = Top – 1
```

```
          5.      Return Data

 Return(stack[top])
 End
```

```
Peek(Stack, pos)

Begin

        Return Stack.s[s.top])

End
```

```
isfull(Stack)

Begin
1.  If Stack.Top = MAXSIZE
            Return "True"
    Else
            Return "False"
  Endif

End
```

```
isempty()

Begin

 1.  If Stack.Top < 1
            Return "True"
  Else
            Return "False"
  Endif
End
```

Below mentioned are the time complexities for various operations that can be performed on the Stack data structure.

- **Push Operation** : O(1)
- **Pop Operation** : O(1)
- **Top Operation** : O(1)
- **Search Operation** : O(n)

Applications

Stack data structures are useful when the order of actions is important. They ensure that a system does not move onto a new action before completing those before. Here are some common examples where a stack is used:

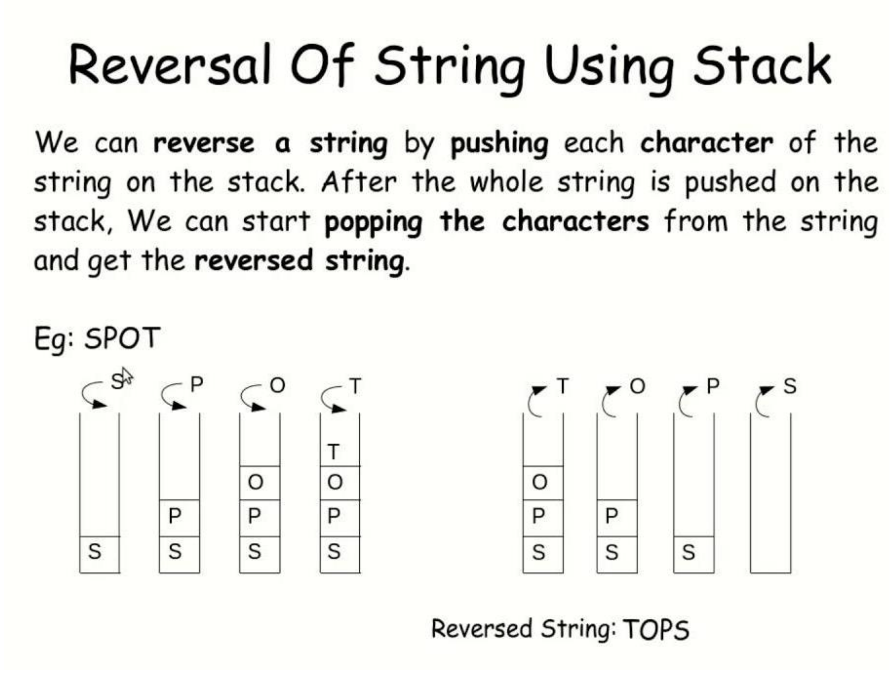
- **Reversing —** By default a data stack will reverse whatever is input. For example, if we wanted to reverse the string "Hello World". We would push() each character onto the stack, and then pop() each character off.

- **Undo/redo —** This approach can be used in editors to implement the undo and redo functionality. The state of the program can be pushed to a stack each time a change is made. In order to undo, use pop() to remove the last change.

- **Backtracking —** This can be used when writing an algorithm to solve a problem involving choosing paths, for example a maze. A path is chosen and if it results in a dead end this latest branch in the path must be removed (pop()) and another route chosen. Each time a path is chosen it is pushed to the stack (E.g. Recursion)

- **Call stack —** Programming languages use a data stack to execute code. When a function is called it is added to the call-stack and removed once completed.

1. Reversing the String



Reversal Of String Using Stack

We can **reverse a string** by **pushing** each **character** of the string on the stack. After the whole string is pushed on the stack, We can start **popping the characters** from the string and get the **reversed string**.

Eg: SPOT

Following is simple algorithm to reverse a string using stack.

```
1) Create an empty stack.
2) One by one push all characters of string to stack.
3) One by one pop all characters from stack and put
   them back to string.
```

Reverse (st1,st2)

```
1. stack1.top =-1, i=1
2. Repeat step 3 to 4 while st1[i] <> Null
3. push(st1[i], stack1)
4. I = i+1

5. I =1
6. repeat step 7 to 8 while not (isempty(stack1))
7 st2[i] = pop(stack1)
8 I = i+1
```

Main ()

1. Input s1
2. Input s2
3. Reverse (s1,rs1)

4. Reverse(s2,rs2)
5. Strcat(rs1,rs2,s3)

6. Output s3

Len(s1) will find the length of the string

Palindrome(s1)

1. hl = len(s1)/2
2.

aabaa

typedef Struct stack{ char str[10]; int top;}