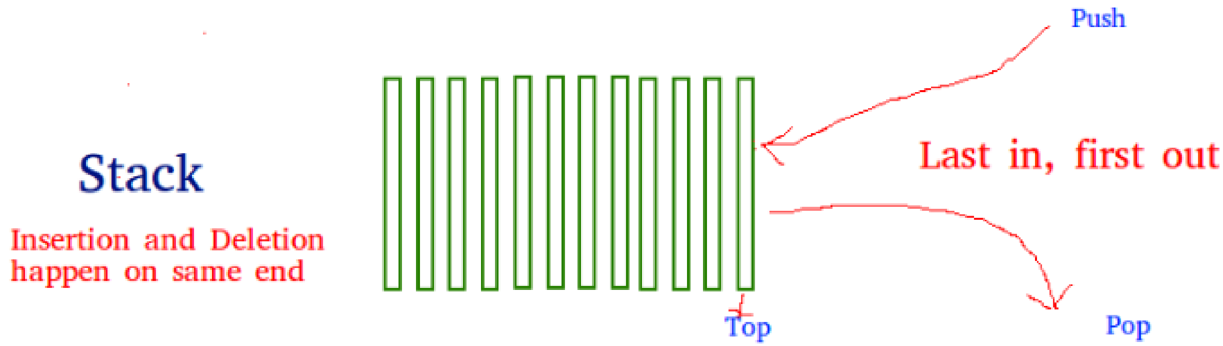
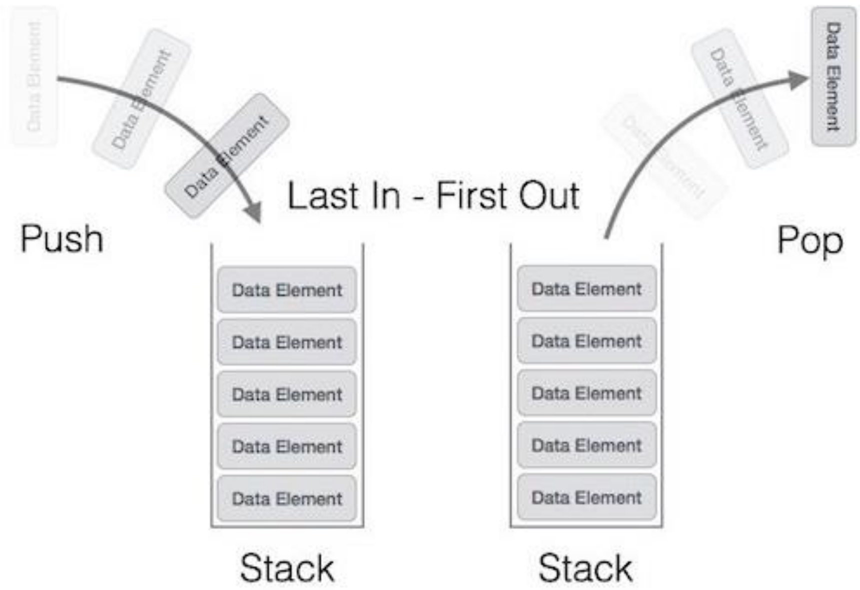
# Stack:

Stacks are a type of container adaptors with LIFO(Last In First Out) type of working, where a new element is added at one end (top) and an element is removed from that end only.  Stack uses an encapsulated object of either vector or deque (by default) or list (sequential container class) as its underlying container, providing a specific set of member functions to access its elements.




A stack is an Abstract Data Type (ADT), Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.




## Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

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

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks –

- **isFull()** − check if stack is full.
- **isEmpty()** − check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

First we should learn about procedures to support stack functions −

## isfull()

Algorithm of isfull() function −

```
begin procedure isfull

   if top equals to MAXSIZE
      return true
   else
      return false
   endif

end procedure
```

## isempty()

Algorithm of isempty() function −

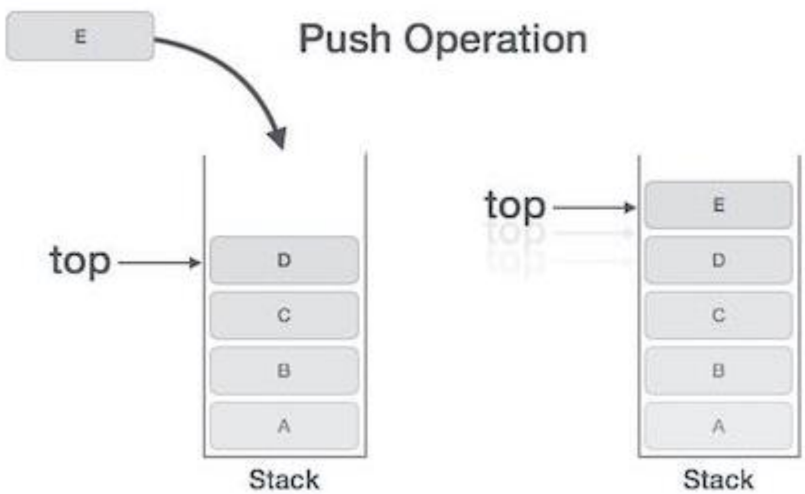
```
begin procedure isempty

   if top =-1
      return true
   else
      return false
   endif

end procedure
```

# 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.

If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.



## Algorithm for PUSH Operation

A simple algorithm for Push operation can be derived as follows −

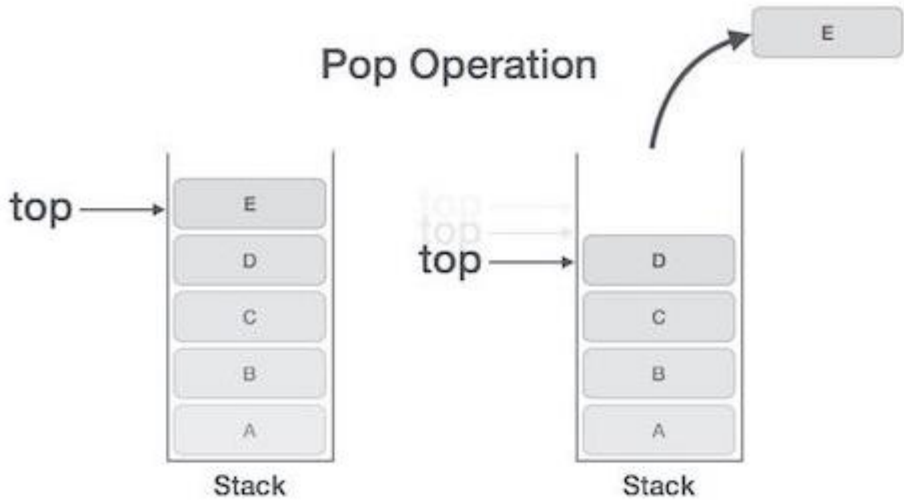
```
begin procedure push: stack, data

   if stack is full
      return null
   endif

   top ← top + 1
   stack[top] ← data

end procedure
```

## 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. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

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

A simple algorithm for Pop operation can be derived as follows −

```
begin procedure pop: stack

   if stack is empty
      return null
   endif

   data ← stack[top]
   top ← top - 1
   return data

end procedure
```

```
#include <iostream>
#include<conio.h>
using namespace std;
const int SIZE = 10;

template <class T>
class stack
{
private:

int tos;
```

```
T array[SIZE];

public:
stack()
{
tos = -1;
}

bool isEmpty()
{
if(tos==-1)
return true;
else
return false;
}

bool isFull()
{
if(tos==(SIZE-1))
return true;
else
return false;
}

void push(T);
T pop();
void displayItems();
};

template <class T>
void stack <T> ::push(T element)
{
if(isFull())
{
cout << "Stack is full.\n";
}
else
{
tos++;
```

```
array[tos] = element;
}
}

template <class T>
T stack<T>::pop()
{ T x;
if(isEmpty()) {
cout << "Stack is empty.\n";
return 0; // return null on empty stack
}
else
{
      x= array[tos];
      tos--;

   return  x;     // or we can write →  return array[tos--];
}}
template <class T>
void stack<T>::displayItems(){
    int i; //for loop
    cout<<"STACK is: ";
    for(i=(tos); i>=0; i--)
        cout<<array[i]<<" ";
    cout<<endl;
}



void main()
{
stack <char> s1,s2;
s1.push('a');
s1.push('b');
s1.push('c');
s2.push('x');
s2.push('y');
s2.push('z');
```

```
cout<<"content of the s1 after pushing"<<endl;
s1.displayItems();
cout<<"content of the s2 after pushing"<<endl;
s2.displayItems();


cout<<s1.pop()<<endl;
cout<<s1.pop()<<endl;
cout<<s2.pop()<<endl;
cout<<s2.pop()<<endl;


cout<<"content of the s1 after pushing and poping is:"<<endl;
s1.displayItems();

cout<<"\ncontent of the s1 after pushing and poping is:"<<endl;
s2.displayItems();

stack<double> ds1, ds2;
ds1.push(1.1);
ds1.push(3.3);
ds1.push(5.5);
ds2.push(2.2);
ds2.push(4.4);
ds2.push(6.6);
cout<<endl;

for(int i=0; i<3; i++)
cout << "Pop ds1: " << ds1.pop() << "\n";
for(int i=0; i<3; i++)
cout << "Pop ds2: " << ds2.pop() << "\n";

getch();
}
```

```
content of the s1 after pushing
STACK is: c b a
content of the s2 after pushing
STACK is: z y x
c
b
z
y
content of the s1 after pushing and poping is:
STACK is: a

content of the s1 after pushing and poping is:
STACK is: x

Pop ds1: 5.5
Pop ds1: 3.3
Pop ds1: 1.1
Pop ds2: 6.6
Pop ds2: 4.4
Pop ds2: 2.2
```