# Lecture 5-The Stack

- **The Stack:** is non-primitive linear data structure in which addition of new data items and deletion of already existing data items is done from only one end known as the top of the stack (ToS).
- As all the deletion and insertion in a stack is done from top of the stack, the last added element will be the first to be removed from the stack.
- This is the reason why the stack is also called Last In – First Out (LIFO) type of list.
- It is interesting to notice that the most frequently accessible element in the stack are the most top elements, while the least accessible elements located at the bottom of the stack.
- Figure 1 represents a stack with the size of five locations and includes three elements.

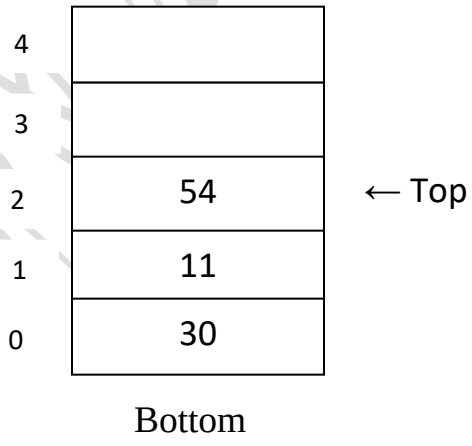



**Figure 1. An example of stack data structure**

## Real Life Stack Applications

- As human beings, we deal with the stack almost all the time.
- Some of real life applications are stack of coins, books, plates…etc, as shown in figure 2.

**Figure 2. Real life stack applications**

## Operations Performed on Stack

- There are two basic operations that can be performed on the stack, these are:

1. **PUSH:** is the process of adding a new element to the top of the stack. As a new element pushed to the stack, top will be incremented by one and denote to the added element. Adding a new element when the stack is full is called stack overflow.
2. **POP:** is the process of deleting an element from the top of the stack. After every pop operation, the top is decremented by one. If there are no elements in the stack and pop operation is performed, the result is called stack underflow.

➢ Figure 3 shows an example of push and pop operations.

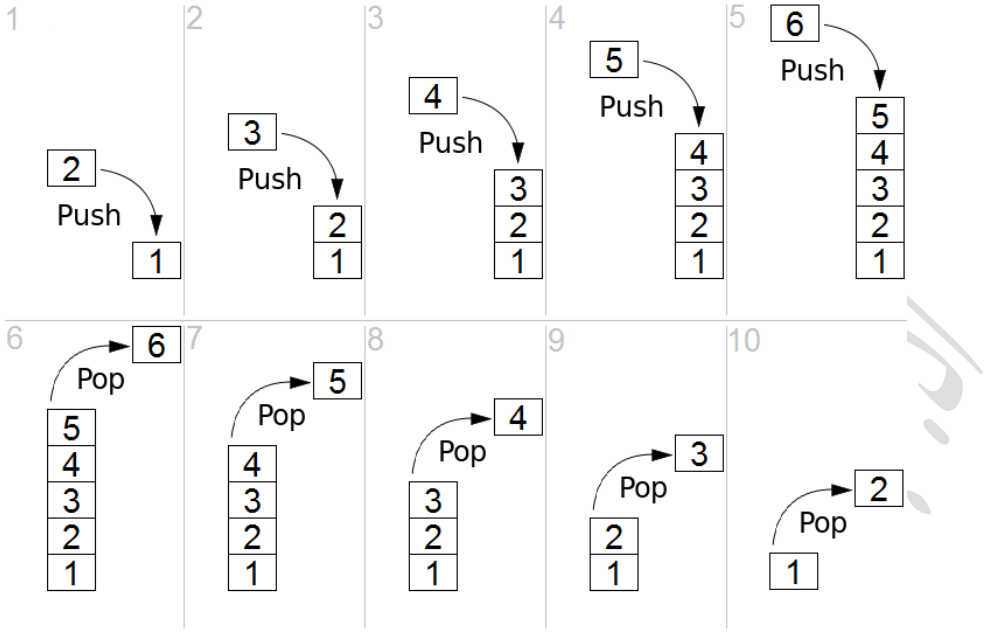



**Figure 3. PUSH and POP operations**

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:

- peek() − get the top data element of the stack, without removing it.
- 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 −

1- peek()

```
begin procedure peek
   return stack[top]
end procedure
```

Implementation of peek() function in C# programming language −

```
int peek() {
   return stack[top];

}
```

2-isfull()

```
begin procedure isfull
   if top equals to MAXSIZE
      return true
   else
      return false
   endif
end procedure
```

Implementation of isfull() function in C# programming language −

```
bool isfull() {
   if(top == MAXSIZE)
      return true;
   else
      return false;
}
```

3-isempty()

Algorithm of isempty() function −

```
begin procedure isempty
   if top less than 1
      return true
   else
      return false
   endif
end procedure
```

Implementation of isempty() function in C# programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty. Here's the code −

```
bool isempty() {
   if(top == -1)
      return true;
   else
      return false; }
```

## 4-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 –**

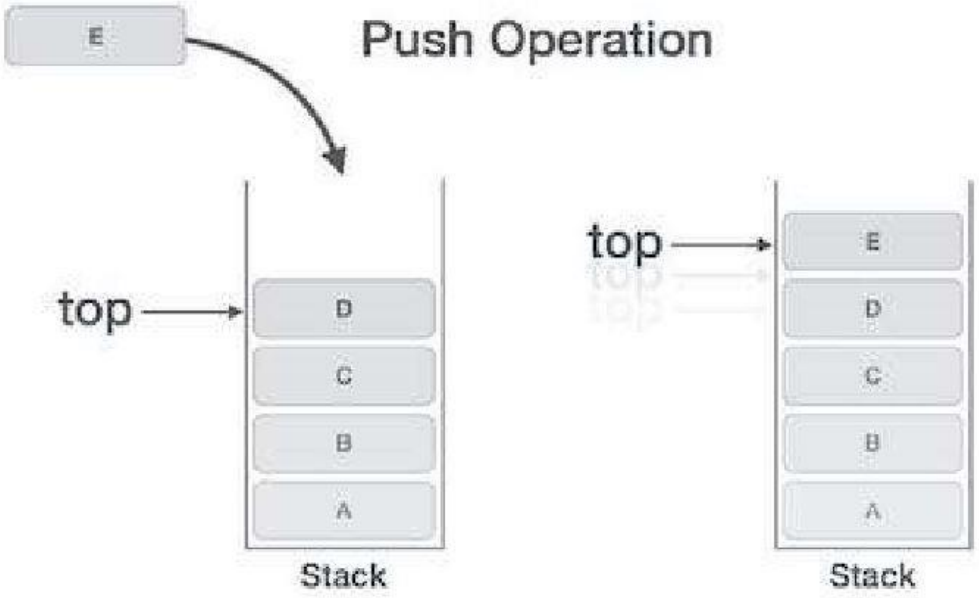
**Step 1 − Checks if the stack is full.**

**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.**



**Algorithm for PUSH Operation**

**begin procedure push: stack, data**

**if stack is full**

**return null**

**endif**

**top ← top + 1**

**stack[top] ← data**

**end procedure**

**Implementation of this algorithm in C# is very easy. See the following code −**

```
void push(int data) {
  if(!isFull()) {
    top = top + 1;
    stack[top] = data;
  }
else {
    printf("Could not insert data, Stack is full.\n");
  } }
```

## 5- 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 −**

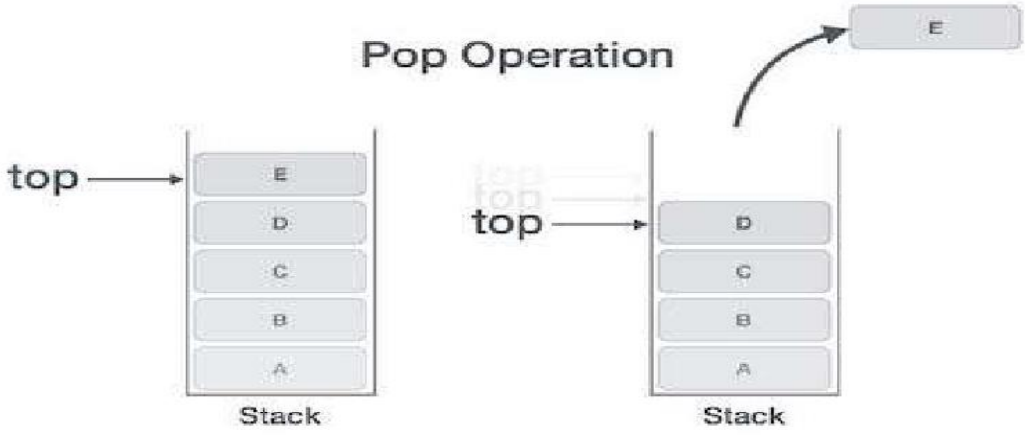
**Step 1 − Checks if the stack is empty.**

**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
```

i+--mplementation of this algorithm in C, is as follows −

```
int pop(int data) {
   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   }else {
      printf("Could not retrieve data, Stack is empty.\n");
   }
}
```

**Stack Terminology**

1. **Size:** this term refers to the maximum size of the stack or the number of possibly added elements.
2. **TOP:** this term refers to the top of the stack. It is a stack pointer used to check overflow and under flow conditions. The initial value of TOP is -1 when the stack is empty.
3. **Stack underflow:** is the situation when the stack contains no elements. At this point the top of the stack points to the stack bottom.

4. **Stack overflow:** is the situation when the stack is full and no more elements can be added. At this point the top of the stack points to the highest location in the stack.

**Example:**

1-What is the obtained string after performing the following sequence of push and pop:

PUSH(A), PUSH(B), POP, PUSH(C), POP, PUSH(D), PUSH(E), POP, POP, POP

**Solution:**

| step | Operation | Stack | Output |
|---|---|---|---|
| 1 | PUSH(A) | A | |
| 2 | PUSH(B) | AB | |
| 3 | POP | A | B |
| 4 | PUSH(C) | AC | B |
| 5 | POP | A | BC |
| 6 | PUSH(D) | AD | BC |
| 7 | PUSH(E) | ADE | BC |
| 8 | POP | AD | BCE |
| 9 | POP | A | BCED |
| 10 | POP | | BCEDA |

So, the obtained string is (BCEDA)

2-What is the obtained number after performing the following sequence of push and pop:

PUSH(1), POP, PUSH(2), PUSH(3), POP, PUSH(4), POP, PUSH(5), POP, POP

**Solution:**

| step | Operation | Stack | Output |
|---|---|---|---|
| 1 | PUSH(1) | 1 | |
| 2 | POP | | 1 |
| 3 | PUSH(2) | 2 | 1 |
| 4 | PUSH(3) | 23 | 1 |
| 5 | POP | 2 | 13 |
| 6 | PUSH(4) | 24 | 13 |
| 7 | POP | 2 | 134 |
| 8 | PUSH(5) | 25 | 134 |
| 9 | POP | 2 | 1345 |
| 10 | POP | | 13452 |

So the obtained number is (13452).

3-What is the required sequence of push and pop to obtain the string (CBDAE) from the initial input (ABCDE).

**Solution:**

| step | Operation | Stack | Output |
|---|---|---|---|
| 1 | PUSH(A) | A | |
| 2 | PUSH(B) | AB | |
| 3 | PUSH(C) | ABC | |
| 4 | POP | AB | C |
| 5 | POP | A | CB |
| 6 | PUSH(D) | AD | CB |
| 7 | POP | A | CBD |
| 8 | POP | | CBDA |
| 9 | PUSH(E) | E | CBDA |
| 10 | POP | | CBDAE |