ADVANCED DATA STRUCTURES AND ALGORITHMS

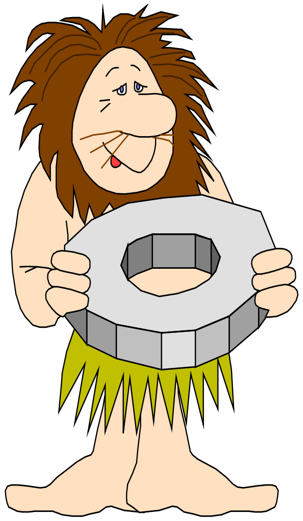
Associate Professor Dr. Raed Ibraheem Hamed

University of Human Development, College of Science and Technology
Computer Science Department

2015 – 2016

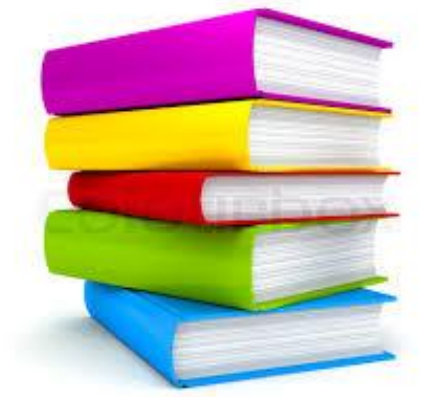
What this Lecture is about:

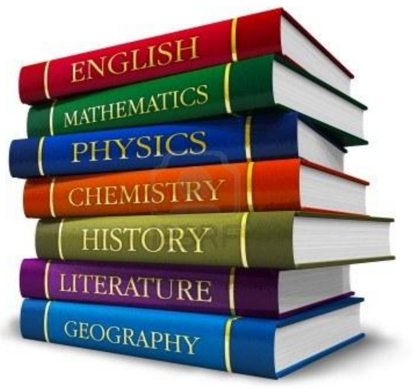
- Stack Structure
- Stack Definition
- Primary operations
- Stack Declaration
- push Algorithm
- pop Algorithm
- Implementation



Stack Structure

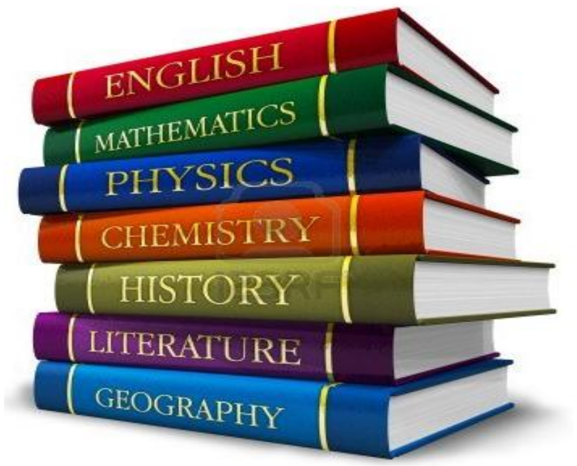
A *stack* is a collection that implements the last-in-first-out (**LIFO**) protocol. This means that the only accessible object in the collection is the last one that was inserted. A stack of books is a good analogy: You can't take a book from the stack without first removing the books that are stacked on top of it.





Stack Definition

A Data structure in which elements are added and removed from one end; “Last in, First out” LIFO Structure.



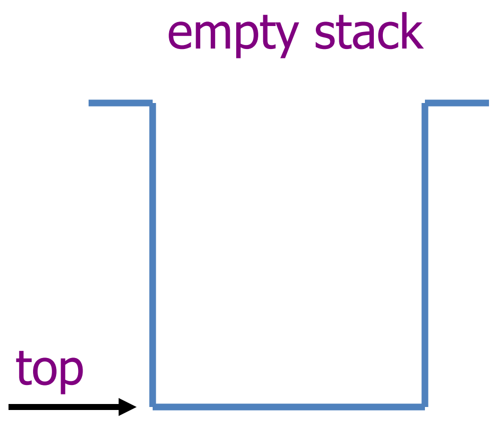
Stack Operations

The fundamental operations of a stack are:

1. **Create** : Create an empty stack.
2. **Push** : Add an element onto the top of the stack.
3. **Pop** : Remove the current element on the top of the stack.
4. **Retrieve or peak** : which reads data without removing it, on the top of the stack .

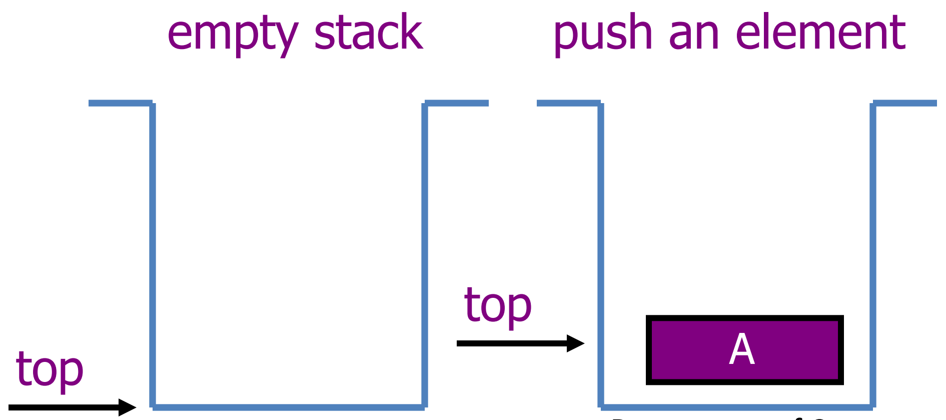
Primary operations

- Push
  - Add an element to the top of the stack
- Pop
  - Remove the element at the top of the stack



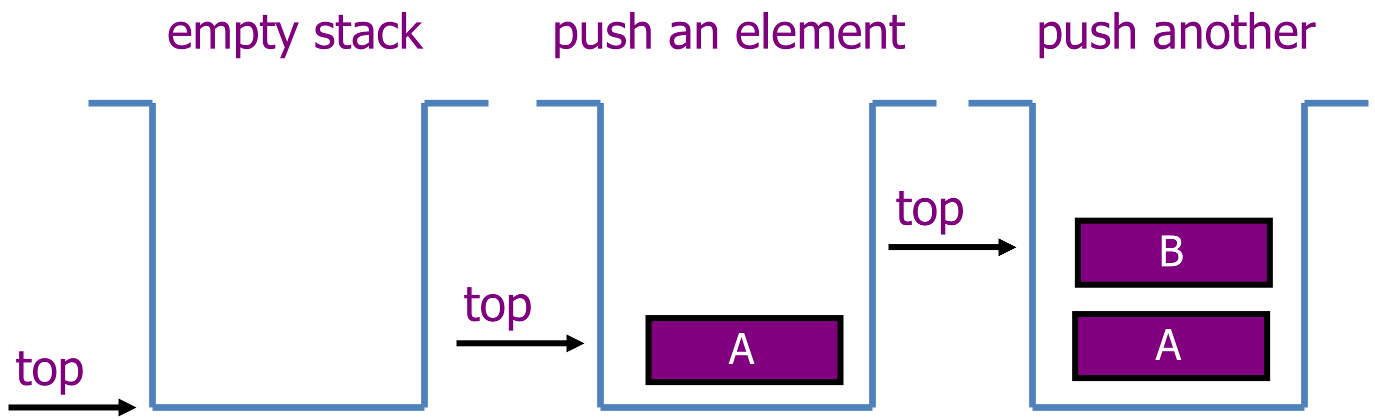
Primary operations

- Push
  - Add an element to the top of the stack
- Pop
  - Remove the element at the top of the stack



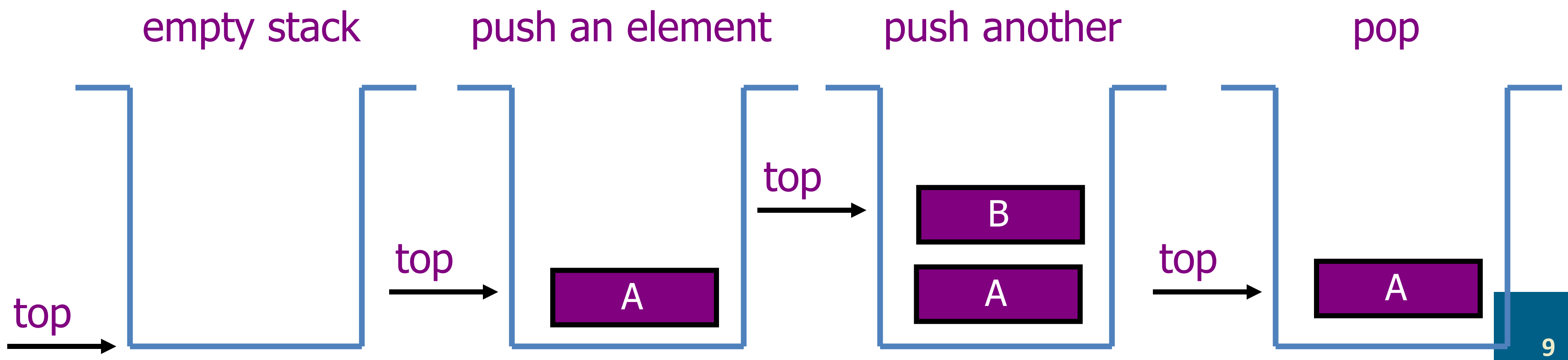
Primary operations

- Push
  - Add an element to the top of the stack
- Pop
  - Remove the element at the top of the stack

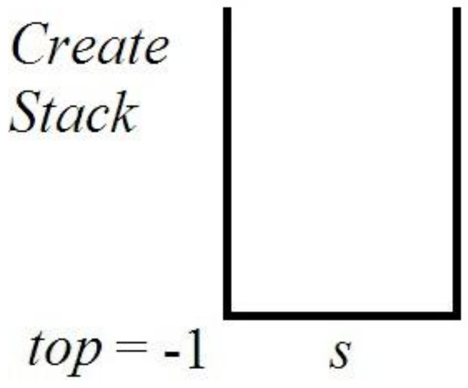


Primary operations

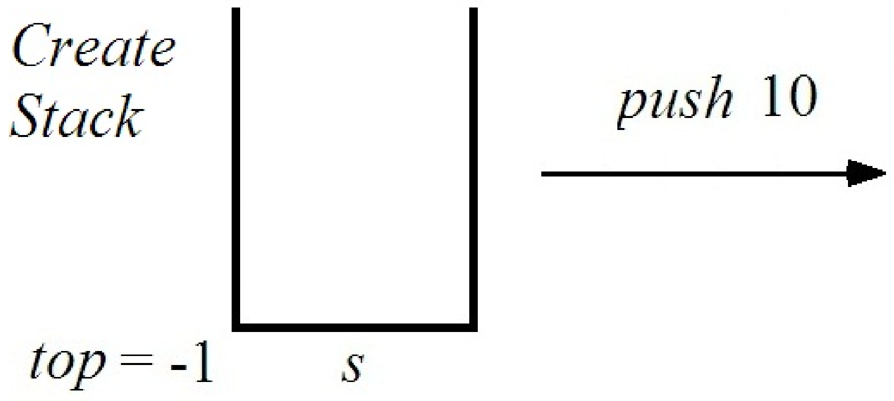
- Push
  - Add an element to the top of the stack
- Pop
  - Remove the element at the top of the stack



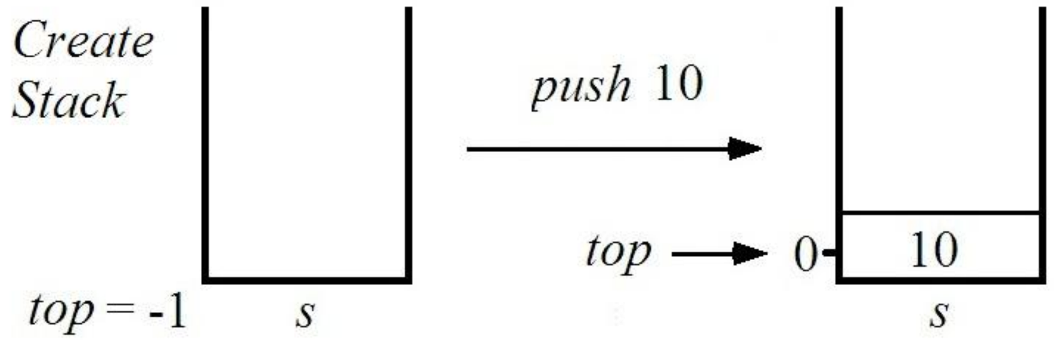
Stack Operations



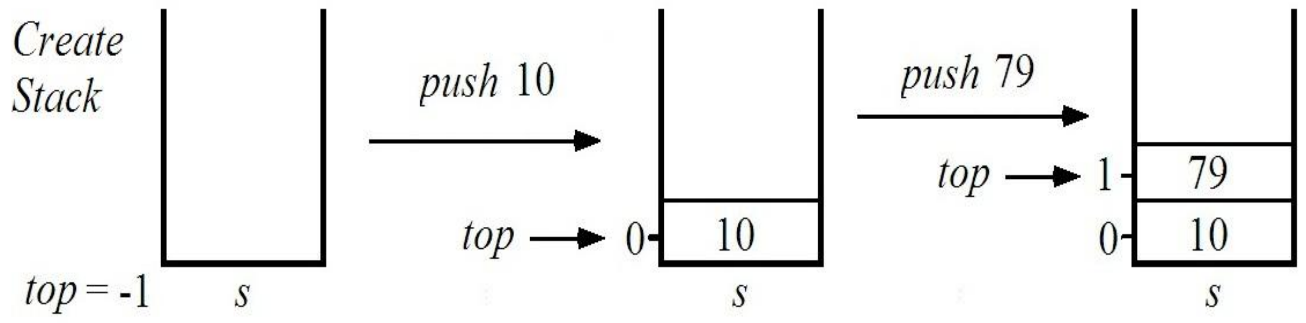
Stack Operations



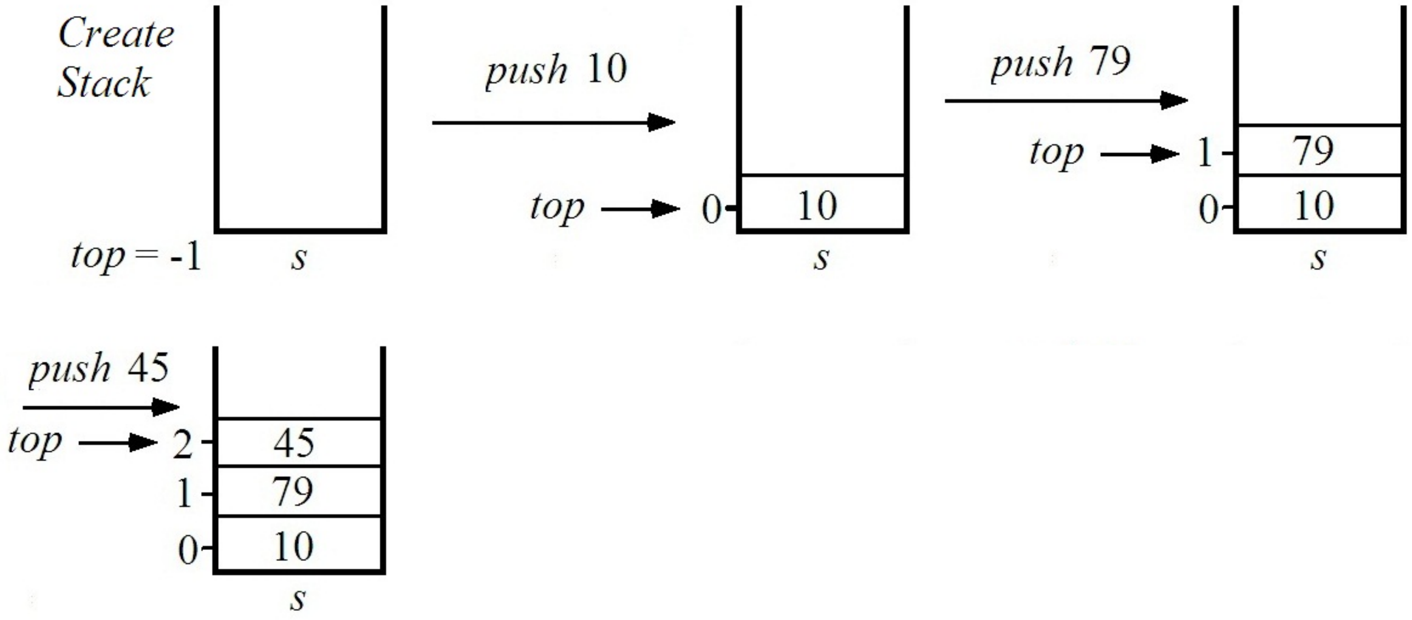
Stack Operations



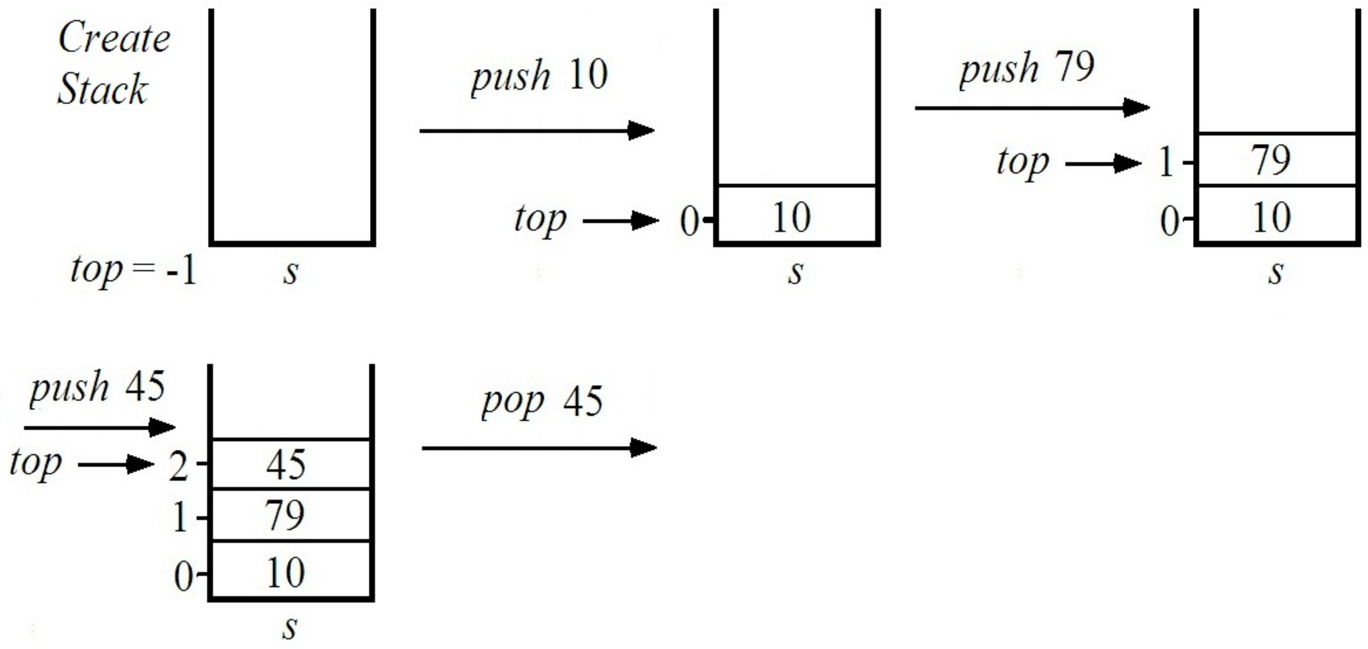
Stack Operations



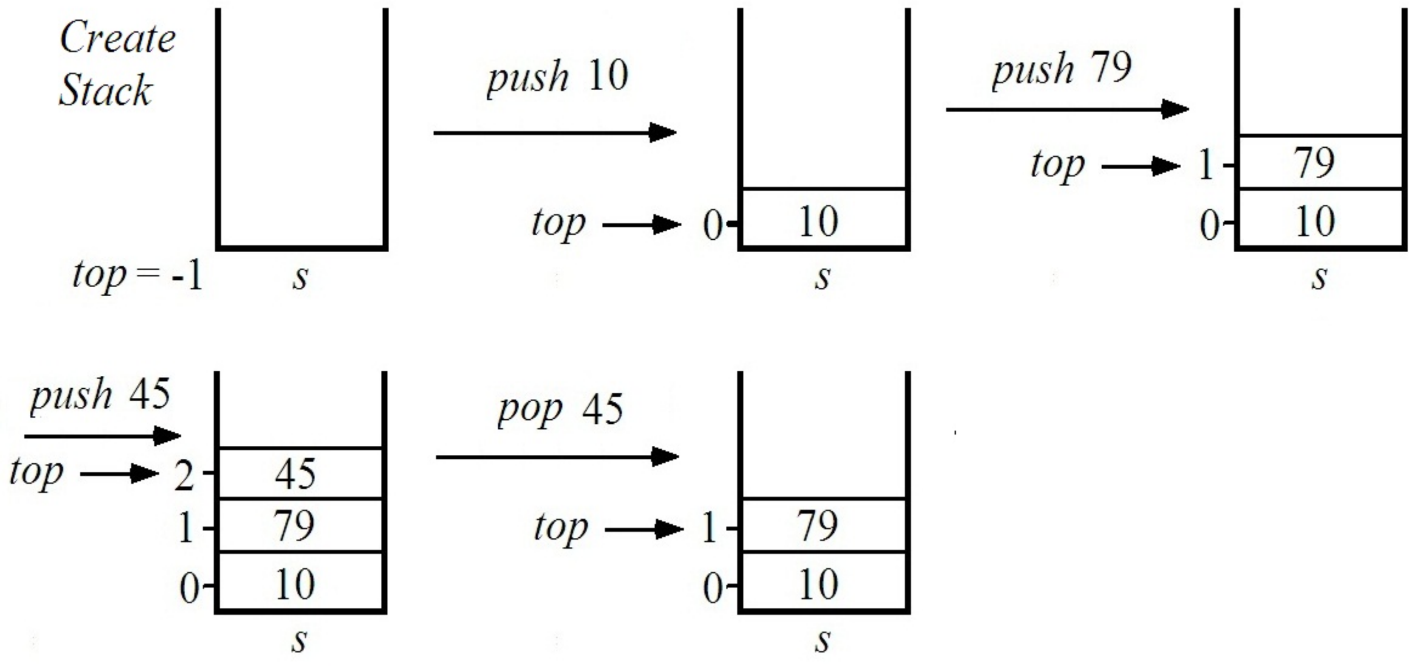
Stack Operations



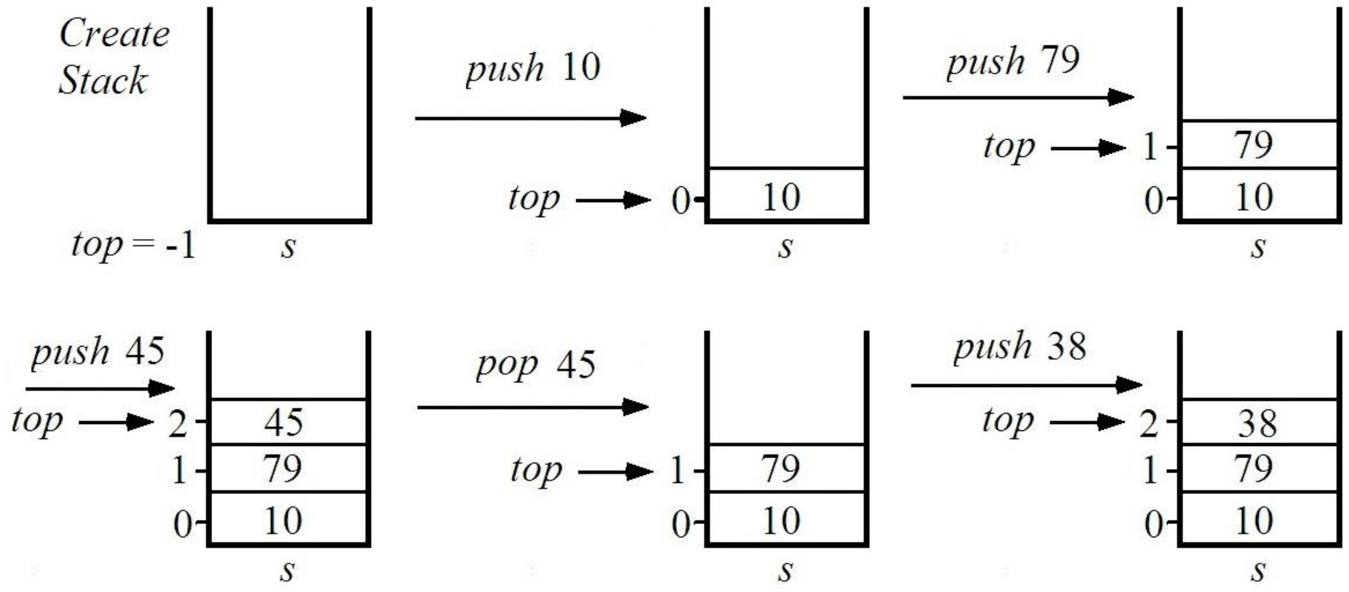
Stack Operations



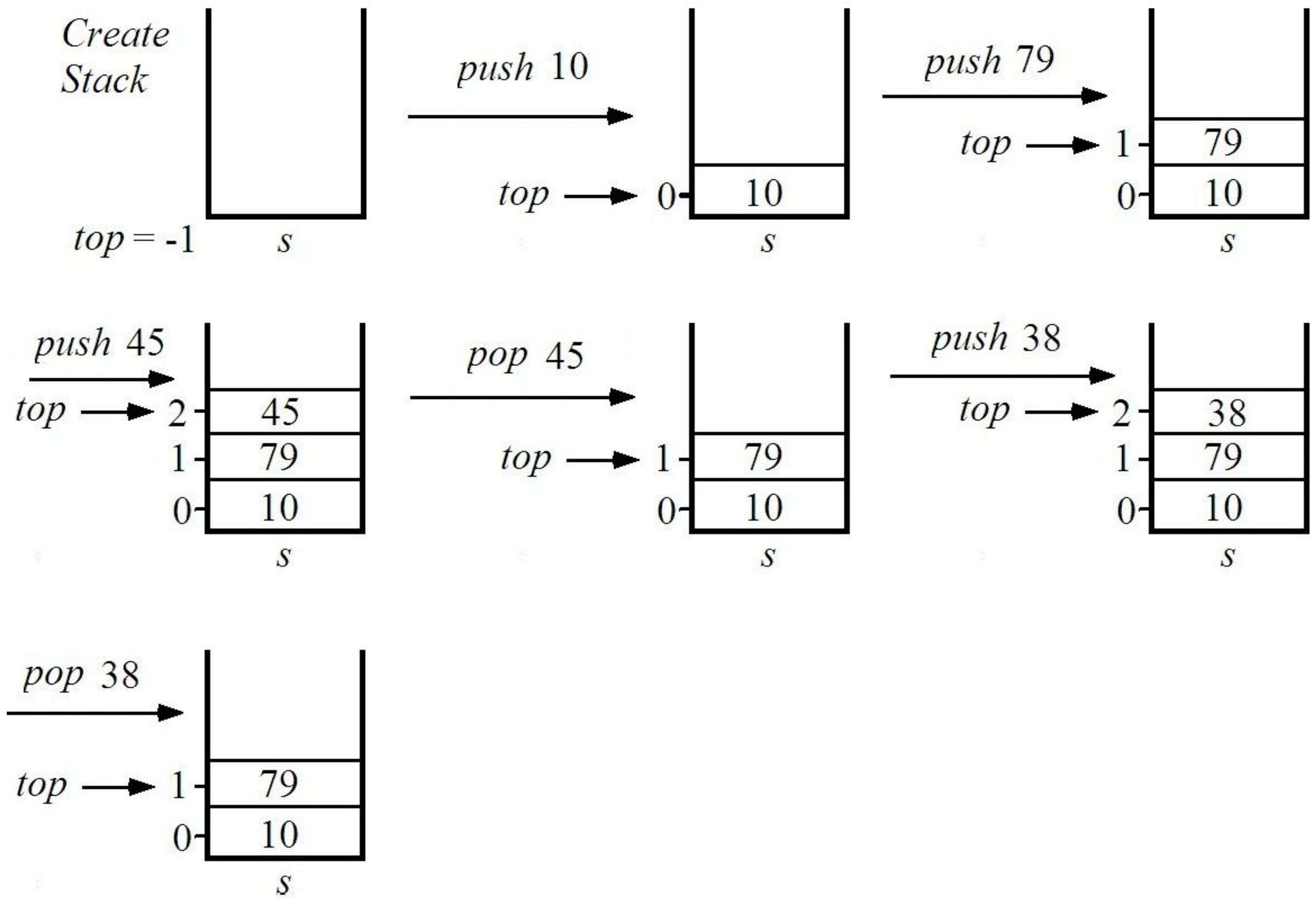
Stack Operations



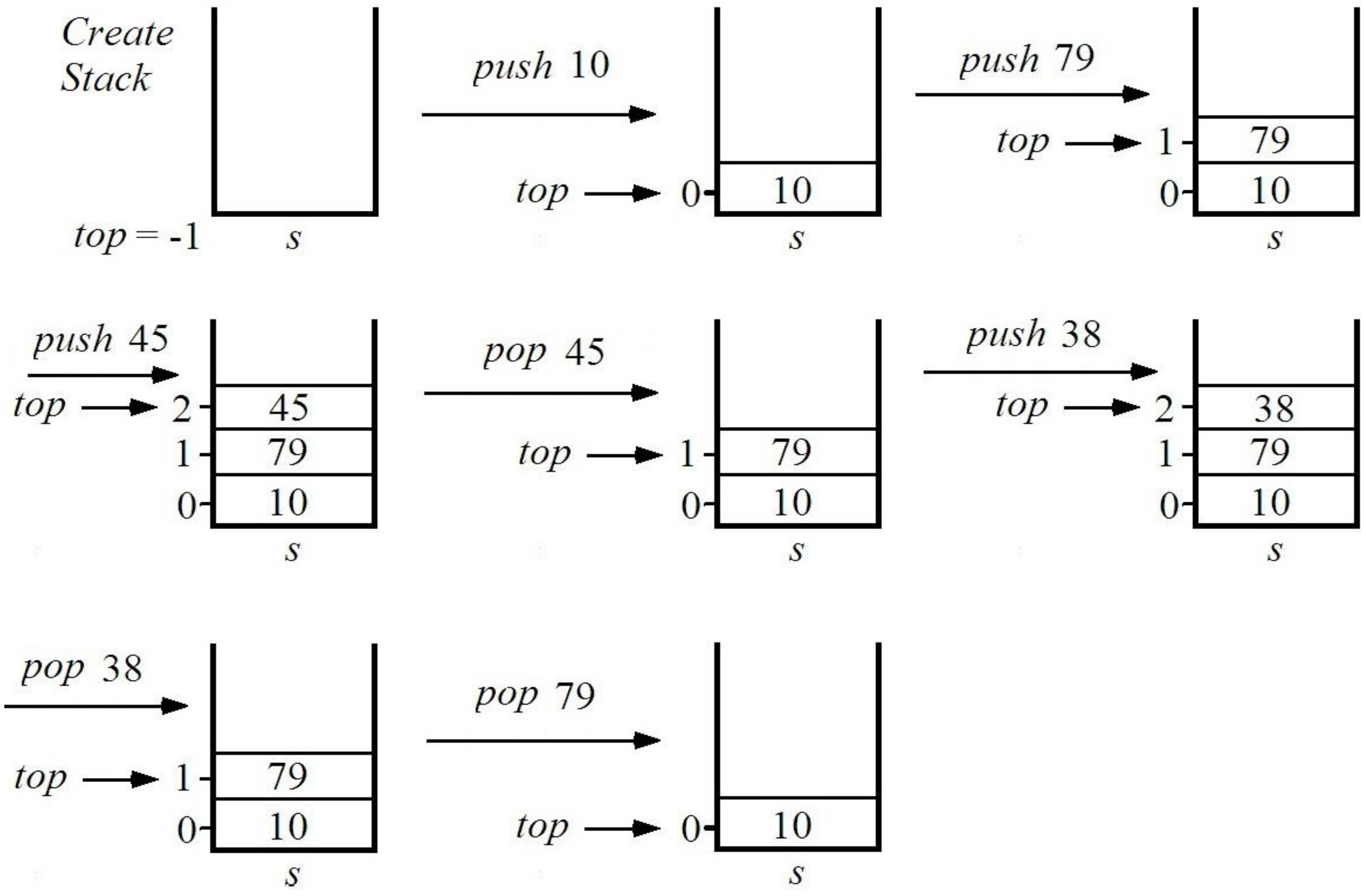
Stack Operations



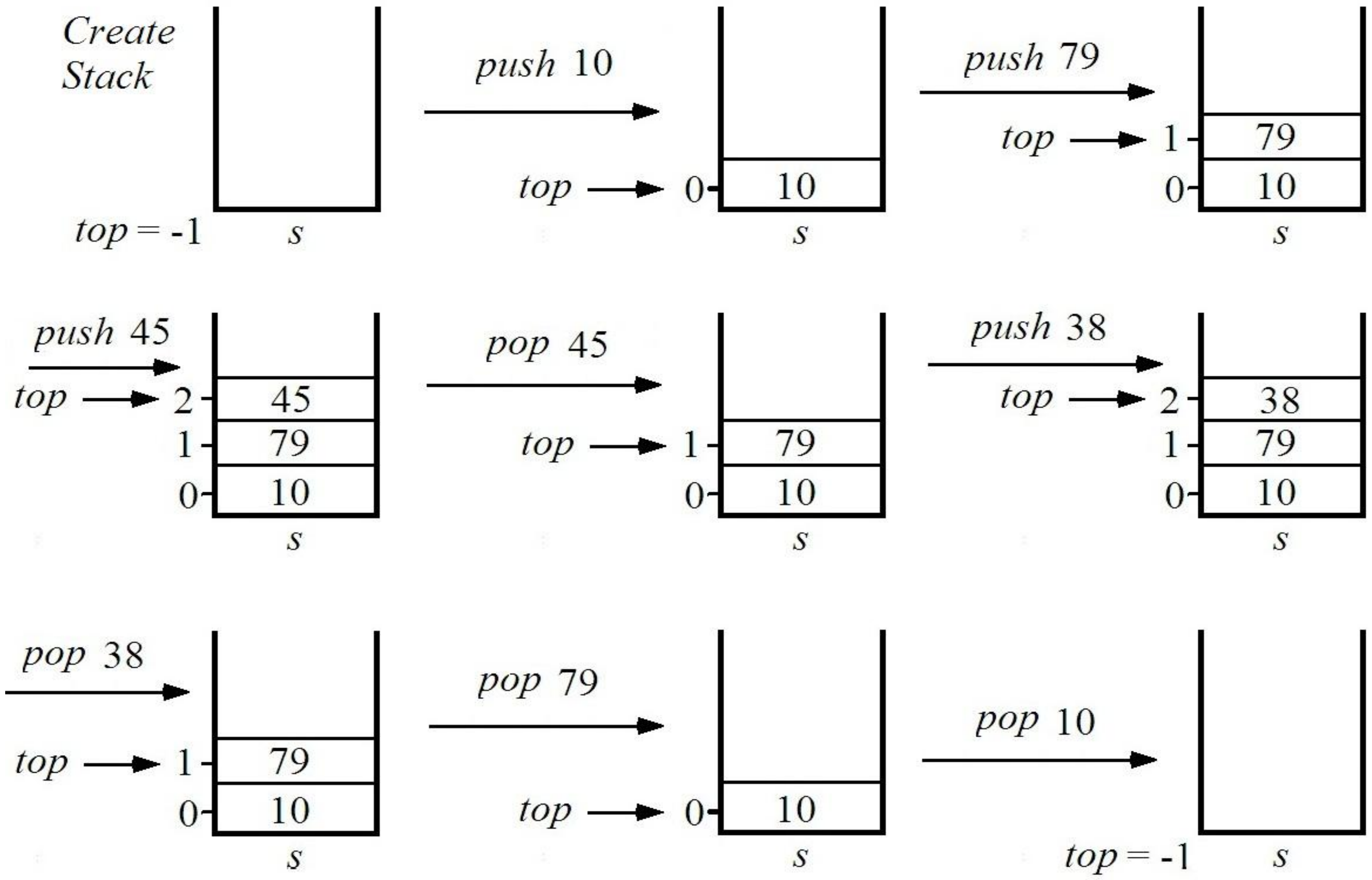
Stack Operations



Stack Operations



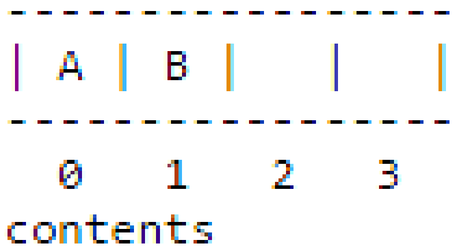
Stack Operations



Implementing a stack with an array:

First, if we want to store letters, we can use type char. **Next**, since a stack usually holds a bunch of items with the same type (**e.g., char**), we can use an array to hold the contents of the stack.
Now, consider how we'll use this array of characters, call it **contents**, to hold the contents of the stack.

Let's choose the array to be of size 4 for now. So, an array getting **A**, then **B**, will look like:



```
-----------------
| A | B |   |   |
-----------------
  0   1   2   3
contents
```

Implementing a stack with an array:

What happens if we apply the following set of operations?

1. Push(stack, 'D')
2. Push(stack, 'E')
3. Push(stack, 'F')
4. Push(stack, 'G')

giving:

```
stack (made up of 'contents' and 'top')
-----------------    -----
| D | E | F | G |    | 3 |
-----------------    -----
  0   1   2   3       top
contents
```

Stack Declaration

When implementing a data structure, the first issue to be addressed is which foundational data structure to use. Often, the choice is between an array-based implementation and a linked-list implementation. The next sections present an **array-based implementation of stacks**.

```
public static class StackX
{
private static int Size;        // size of stack array
private static int[ ] item;
private static int top;         // top of stack
.....
}
```

Stack Functions

For implementing the stack operation we must construct a Java function for each original operations and the other operations:

Create Stack Function

```
public StackX(int s)            // constructor
{
   Size = s;                     // set array size
   item = new int[Size];         // create array
   top = -1;                     // no items yet
}
```

Push Algorithm

```
Input:  x as input New element.
            Stack before pushing x
Output: Stack after pushing x.
1-  [overflow]
      if      top>=N
      Then      Over flow
2-  [increment pointer]
         top ← top+1
3-  [insert element]
          Stack[top] ← New element x
```

Pop Algorithm

Input: stack before popping element

Output: Stack after popping element.

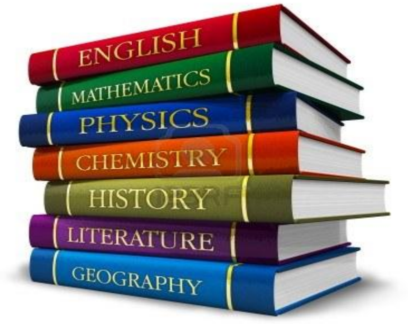
```
1-[underflow]
    if    top<=-1   Then      under flow
2-[unstack element]
    element ← Stack[top]
3-[decrement pointer]
        top ← top-1
```

Implementation

```
public static boolean is Full()
{
 if(top>=size-1)
    return true;
 else
    return false;
}
public static  boolean is Empty()
{
 if(top==-1)
    return true;
 else
    return false;   }
```

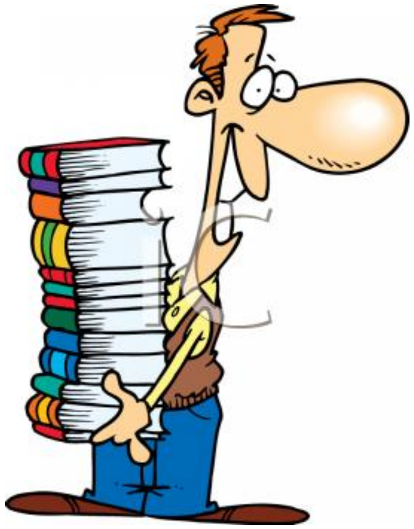
Implementation

```
public static void push(int item)
{
 if(isFull())
        System.out.println("error...the stack is full“);
else
  {
   top=top+1;
   stack[top]=item;
  }
  }
```



Implementation

```
public static int pop()
{
 if(isEmpty())
      System.out.println("error...the stack is empty“);
else
 {
  item=stack[top];
  top=top-1;
  return item;
 }
return 0;
}
```



Implementation

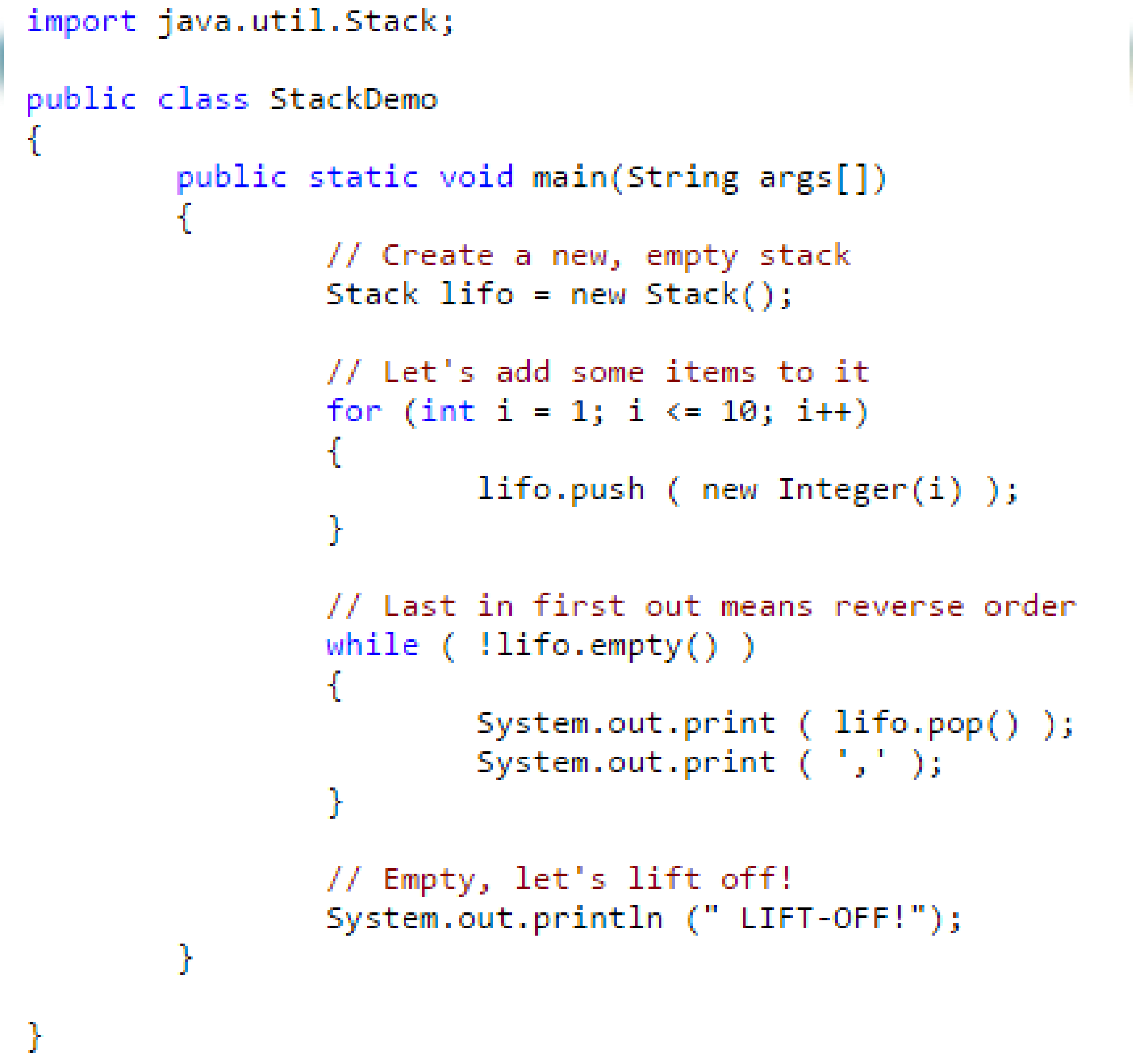
- Retrieve or Peak element from Stack Function

```
public int retrieve()
// take item from top of Stack
{
    return stackArray[top];
    // access item,
}
```

Applications of STACKS

- Stacks can be used **to reverse a sequence**. For example, if a string ”**Computers**” is entered by the user the stack can be used to create and display the reverse string “**sretupmoC**” as follows.
- The program simply *pushes* all of the characters of the string into the stack. Then it *pops* and *display* until the stack is empty.

A Sample Application That Uses A Stack To Reverse A List Of Numbers.



```java
import java.util.Stack;

public class StackDemo
{
        public static void main(String args[])
        {
                // Create a new, empty stack
                Stack lifo = new Stack();

                // Let's add some items to it
                for (int i = 1; i <= 10; i++)
                {
                        lifo.push ( new Integer(i) );
                }

                // Last in first out means reverse order
                while ( !lifo.empty() )
                {
                        System.out.print ( lifo.pop() );
                        System.out.print ( ',' );
                }

                // Empty, let's lift off!
                System.out.println (" LIFT-OFF!");
        }

}
```

Thank you

???