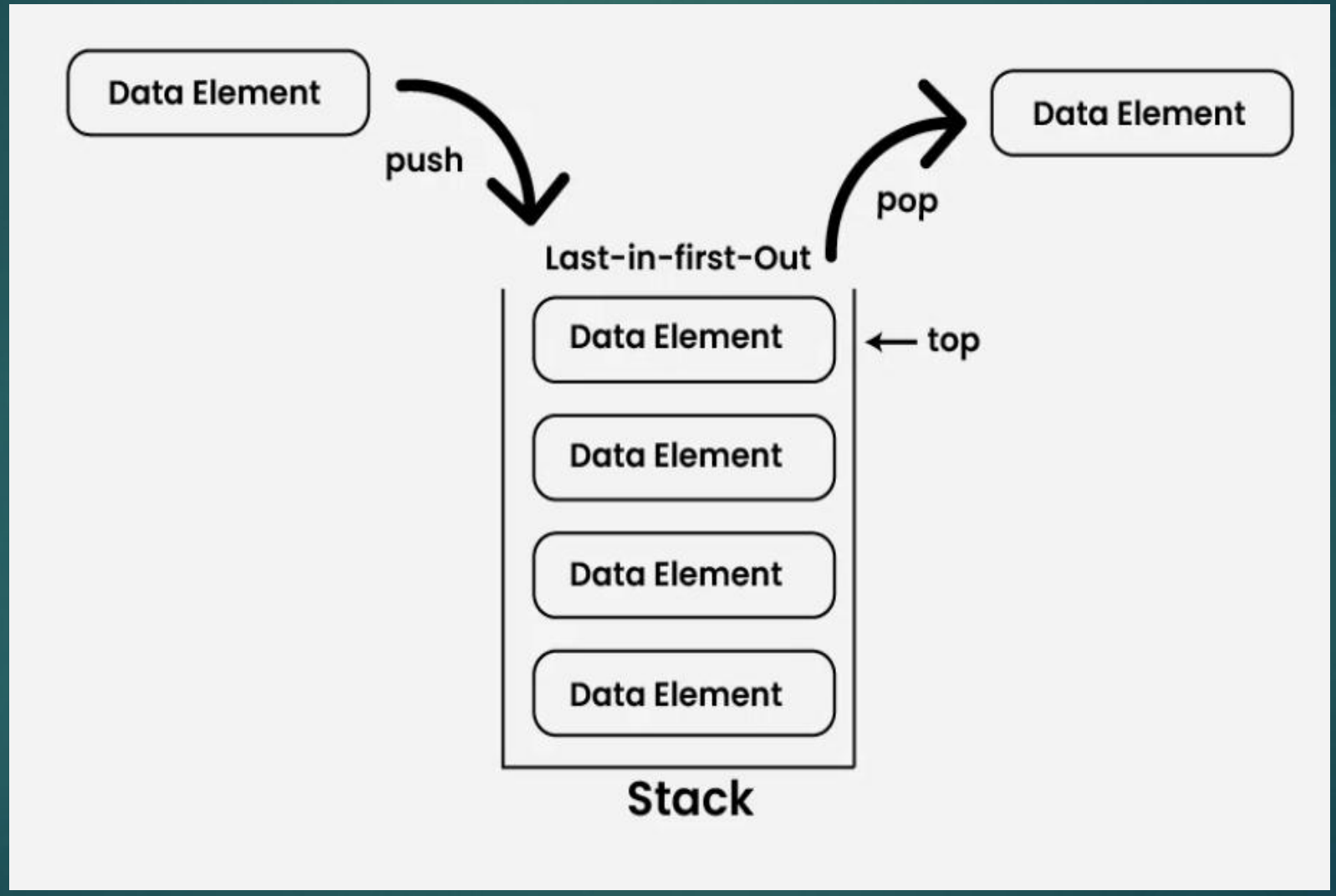
Stack

"A STACK IS LIKE A PILE OF PLATES IN THE KITCHEN — YOU ADD TO THE TOP, YOU TAKE FROM THE TOP, AND IF YOU MESS WITH THE MIDDLE, YOU'LL PROBABLY BREAK THE WHOLE THING."

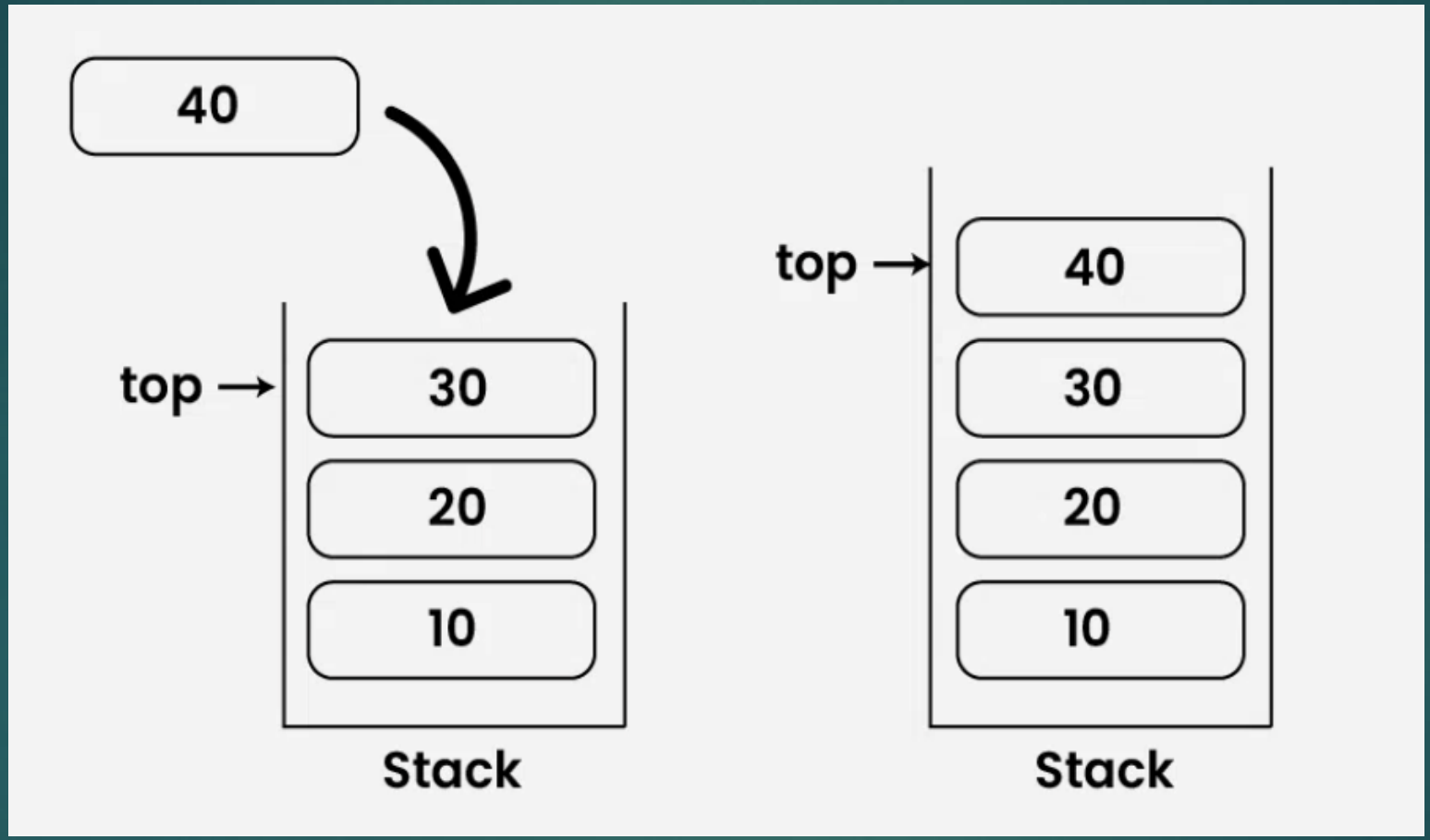
- **Stack** is a linear data structure that follows **LIFO (Last In First Out)** principle, the last element inserted is the first to be popped out.
- Consider a stack of cloths/books/plates. When we add a plate, we add at the top. When we remove, we remove from the top.
- Types of Stack: Fixed size and dynamic size
- Basic Stack operations:
  - **push()** to insert an element into the stack
  - **pop()** to remove an element from the stack
  - **top()** Returns the top element of the stack.
  - **isEmpty()** returns true if stack is empty else false.
  - **isFull()** returns true if the stack is full else false.


Data Element
push
Data Element
pop
Last-in-first-Out
Data Element
top
Data Element
Data Element
Data Element
Stack

- **Push Operation on Stack**

- Add an item to the stack. If the stack is full, then it gives an overflow condition
- Algorithm for Push Operation:

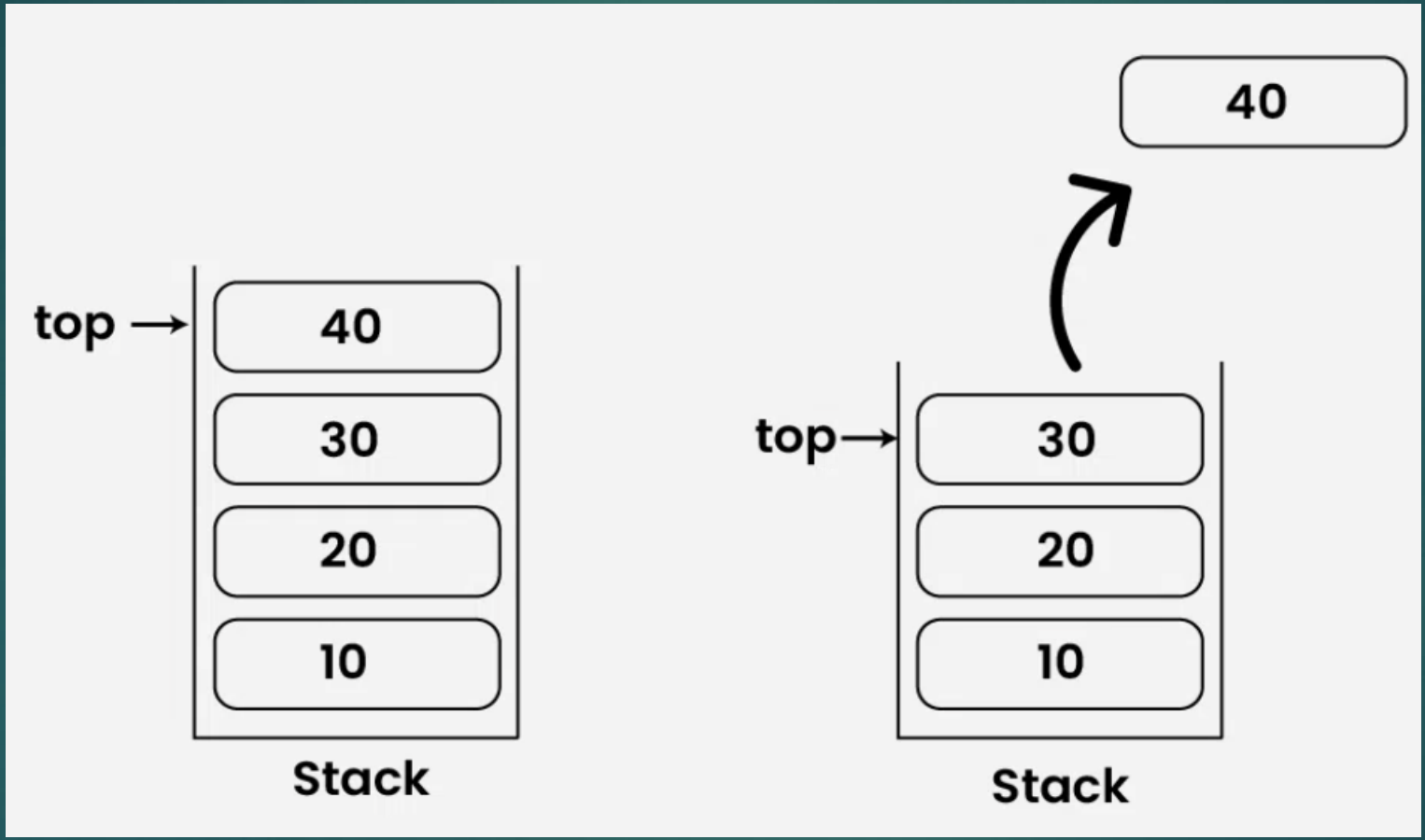
1. Before pushing the element to the stack, we check if the stack is full.
2. If the stack is full **(top == capacity-1)** , then **Stack Overflows** and we cannot insert the element to the stack.
3. Otherwise, we increment the value of top by 1 **(top = top + 1)** and the new value is inserted at top position.
4. The elements can be pushed into the stack till we reach the capacity of the stack.


40
top
30
20
10
Stack
top
40
30
20
10
Stack

- **Pop Operation on Stack**

- Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition**.**
- Algorithm for Pop Operation:

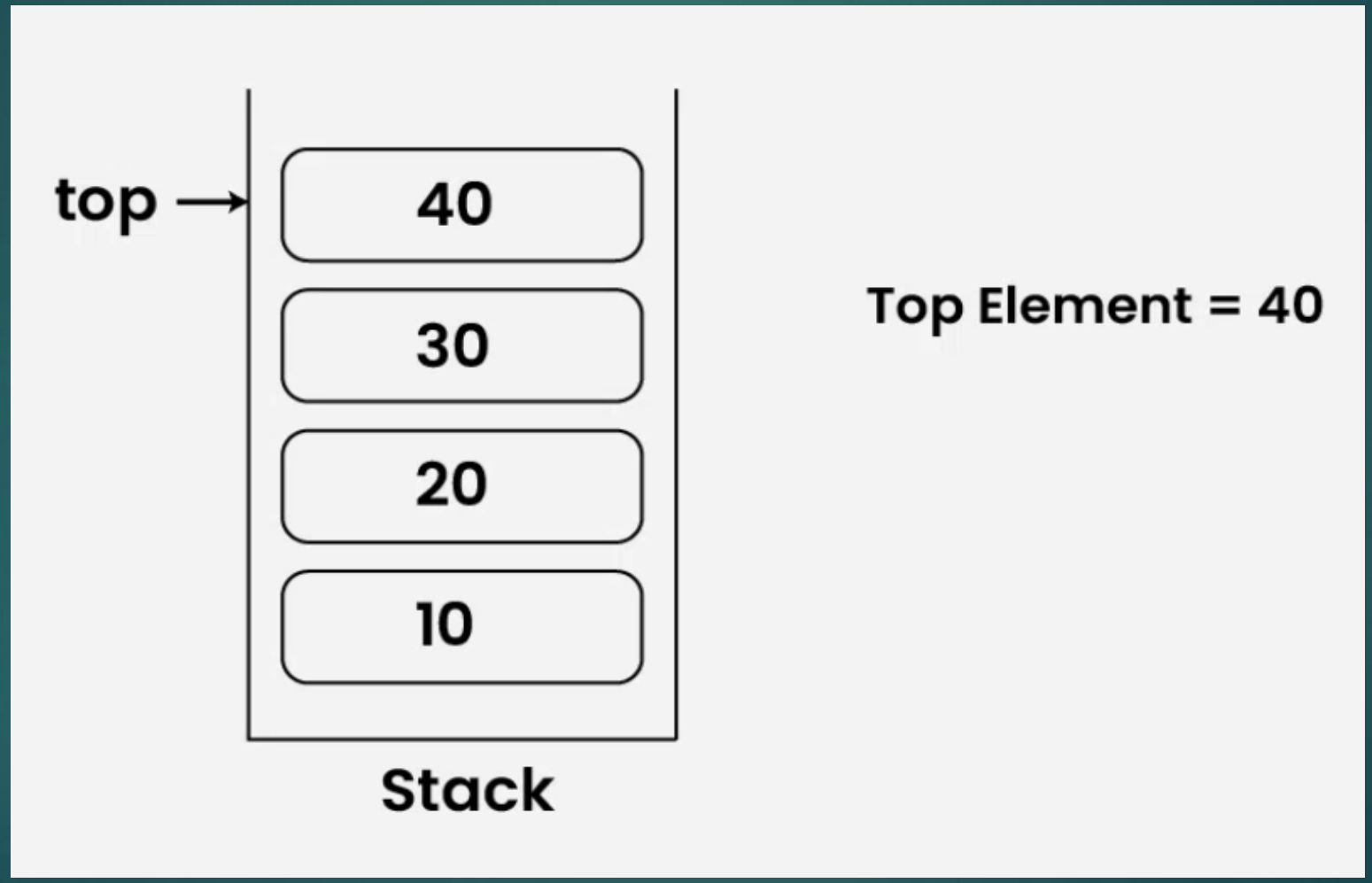
1. Before popping the element from the stack, we check if the stack is empty.
2. If the stack is empty (top == -1), then **Stack Underflows** and we cannot remove any element from the stack.
3. Otherwise, we store the value at top, decrement the value of top by 1 **(top = top - 1)** and return the stored top value.


top
40
30
20
10
Stack
40
top
30
20
10
Stack

- **Top or Peek Operation on Stack**

- Returns the top element of the stack.
- Algorithm for Top Operation:

1. Before returning the top element from the stack, we check if the stack is empty.
2. If the stack is empty (top == -1), we simply print "Stack is empty".
3. Otherwise, we return the element stored at **index = top** .


top
40
30
20
10
Stack
Top Element = 40

- **isEmpty Operation on Stack**

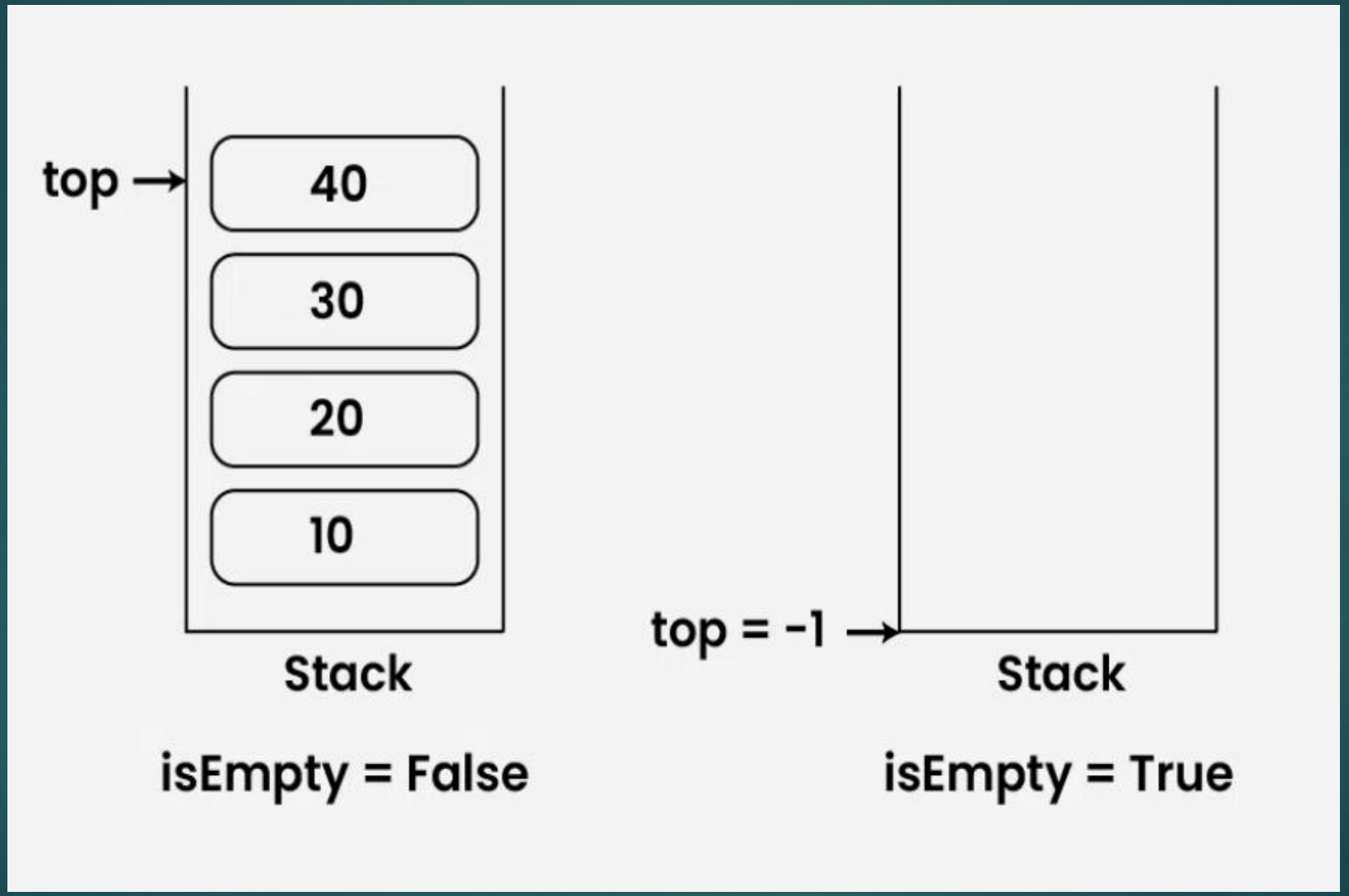
- Returns true if the stack is empty, else false.
- Algorithm for isEmpty Operation:

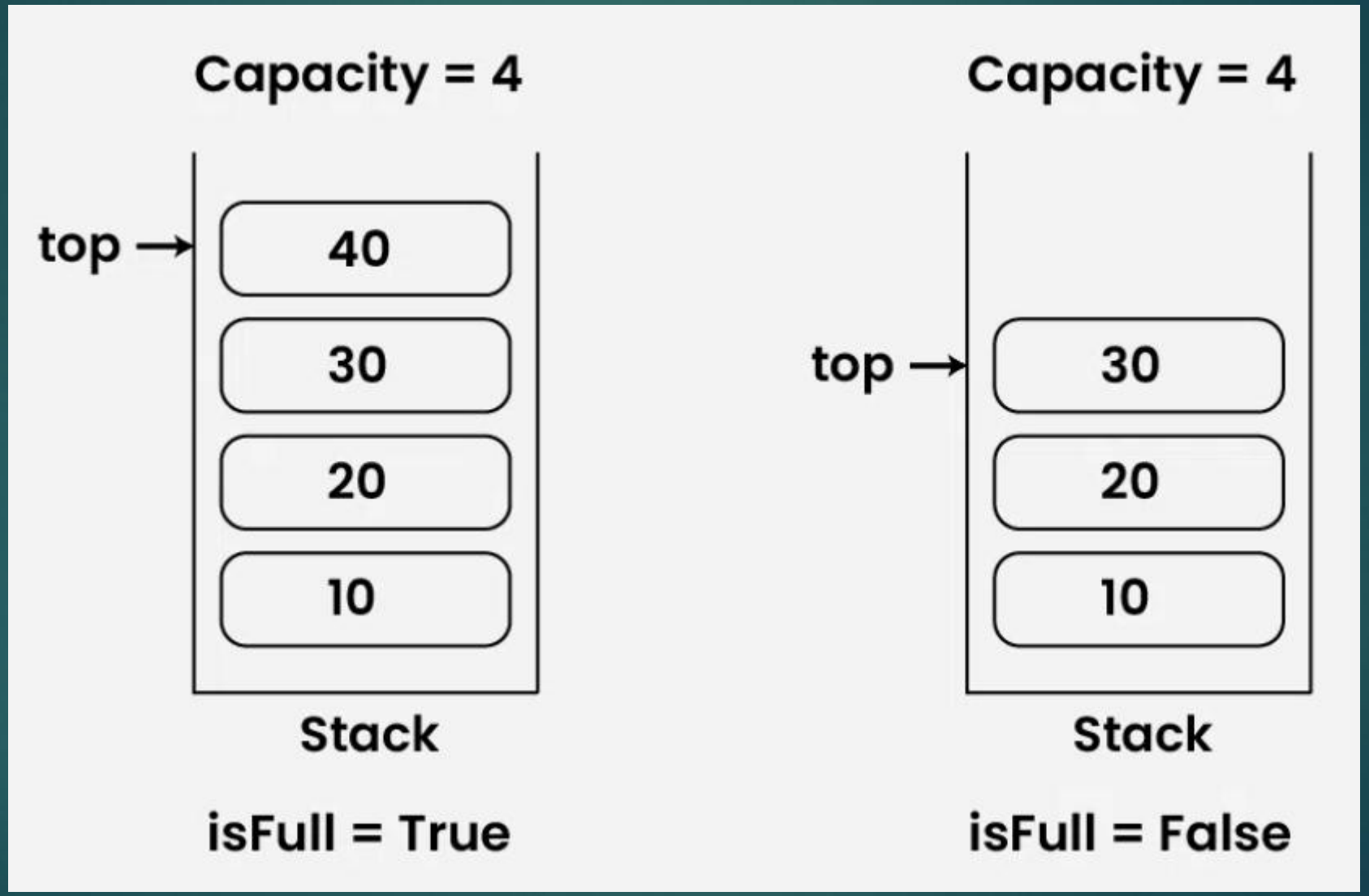
1. Check for the value of top in stack.
2. If **(top == -1)**, then the stack is empty so return **true** .
3. Otherwise, the stack is not empty so return **false** .

- **isFull Operation on Stack**

- Returns true if the stack is full, else false.
- Algorithm for isEmpty Operation:

1. Check for the value of top in stack.
2. If **(top == capacity-1)**, then the stack is full so return **true** .
3. Otherwise, the stack is not full so return **false** .


top
40
30
20
10
Stack
isEmpty = False
top = -1
Stack
isEmpty = True


Capacity = 4
top
40
30
20
10
Stack
isFull = True
Capacity = 4
top
30
20
10
Stack
isFull = False

Operations	Time Complexity	Space Complexity
push()	O(1)	O(1)
pop()	O(1)	O(1)
top() or peek()	O(1)	O(1)
isEmpty()	O(1)	O(1)
isFull()	O(1)	O(1)

```
public class Main {
    public static void main(String[] args) {
        Stack s = new Stack(5);
        s.push(10);
        s.push(20);
        s.push(30);
        System.out.println(s.pop() + " popped from stack");

        System.out.println("Top element is: " + s.peek());

        System.out.print("Elements present in stack: ");
        while (!s.isEmpty()) {
            System.out.print(s.peek() + " ");
            s.pop();
        }
    }
}
```

```java
class Stack {
    int top, cap;
    int[] a;

    public Stack(int cap) {
        this.cap = cap;
        top = -1;
        a = new int[cap];
    }

    public boolean push(int x) {
        if (top >= cap - 1) {
            System.out.println("Stack Overflow");
            return false;
        }
        a[++top] = x;
        return true;
    }

    public int pop() {
        if (top < 0) {
            System.out.println("Stack Underflow");
            return 0;
        }
        return a[top--];
    }
```

```
    public int peek() {
        if (top < 0) {
            System.out.println("Stack is Empty");
            return 0;
        }
        return a[top];
    }

    public boolean isEmpty() {
        return top < 0;
    }
}
```

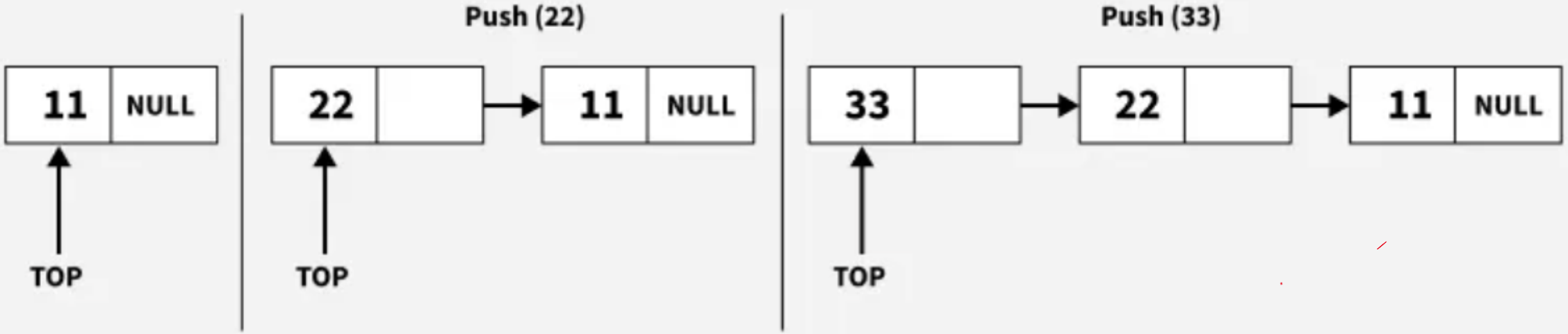
```java
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<Integer> s = new ArrayList<>();

        // Push elements
        s.add(10);
        s.add(20);
        s.add(30);

        // Pop and print the top element
        System.out.println(s.get(s.size() - 1) + " popped from stack");
        s.remove(s.size() - 1);

        // Peek at the top element
        System.out.println("Top element is: " + s.get(s.size() - 1));

        // Print all elements in the stack
        System.out.print("Elements present in stack: ");
        while (!s.isEmpty()) {
            System.out.print(s.get(s.size() - 1) + " ");
            s.remove(s.size() - 1);
        }
    }
}
```

- **Advantages of array based stack implementation**

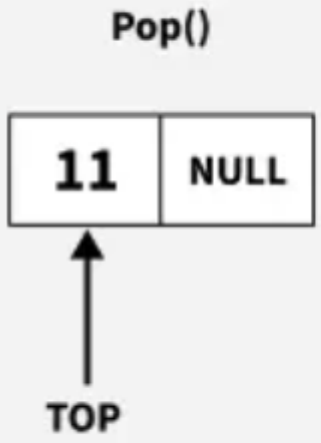
- **Fast Access** – Direct indexing makes push and pop operations run in O(1) time (when size is within capacity).
- **Simplicity** –Easy to implement using a fixed-size array with a single top (integer index).
- **Memory Locality** – Elements are stored in contiguous memory locations, improving cache performance.
- **No Pointer Overhead** – Unlike linked lists, arrays don't require extra memory for storing node pointers.

- **Disadvantages of array based stack implementation**

- **Fixed Size** – The maximum size must be defined in advance; resizing requires creating a new array and copying elements.
- **Wasted Memory** – If the stack is under-utilized, unused slots still occupy memory.
- **Overflow Risk** – Pushing beyond capacity causes **stack overflow** unless dynamically resized.
- **Inefficient Resizing** – Increasing size dynamically (if allowed) involves O(n) copying, affecting performance.



To push a new element onto the stack, create a temporary node temp. Assign the data value and link the temp node to the current top by setting temp->link = top. Finally, update the top pointer to point to the newly created node by setting top = temp.




```
temp = top;
top = top → link;
temp → link = NULL;
free (temp);
```

```
public class Main {
    public static void main(String[] args) {
        Stack st = new Stack();

        st.push(11);
        st.push(22);
        st.push(33);
        st.push(44);

        System.out.println(st.peek());

        st.pop();
        st.pop();

        System.out.println(st.peek());
    }
}
```

```
class Node {
    int data;
    Node next;

    Node(int new_data) {
        this.data = new_data;
        this.next = null;
    }
}
```

```java
// Stack using linked list
class Stack {
    Node head;

    Stack() {
        this.head = null;
    }

    // Check if stack is empty
    boolean isEmpty() {
        return head == null;
    }

    // Push an element onto stack
    void push(int new_data) {
        Node new_node = new Node(new_data);
        new_node.next = head;
        head = new_node;
    }

    // Pop the top element
    void pop() {
        if (isEmpty()) return;
        head = head.next;
    }

    // Return the top element
    int peek() {
        if (!isEmpty()) return head.data;
        return Integer.MIN_VALUE;
    }
}
```

- **Benefits of implementing a stack using a singly linked list**

- **Dynamic memory allocation**: The size of the stack can be increased or decreased dynamically by adding or removing nodes from the linked list, without the need to allocate a fixed amount of memory for the stack upfront.

- **Efficient memory usage:** Since nodes in a singly linked list only have a next pointer and not a prev pointer, they use less memory than nodes in a doubly linked list.

- **Easy implementation**: Implementing a stack using a singly linked list is straightforward and can be done using just a few lines of code.