# Implement_Stack_from_Scratch

July 19, 2024

## 1 Implement Stack From Scratch

A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle. It is an abstract data type with two main operations: push (inserting an element onto the stack) and pop (removing the top element from the stack).

Advantages of using a stack compared to other data structures include:

1. Simplicity: Stacks have a simple and intuitive interface, making them easy to understand and use.
2. Efficiency: The push and pop operations in a stack have a time complexity of O(1), making them very efficient.
3. Memory management: Stacks use a fixed amount of memory, making them suitable for situations where memory usage needs to be controlled.
4. Function call management: Stacks are commonly used to manage function calls in programming languages, allowing for efficient recursion and backtracking.
5. Undo/Redo functionality: Stacks are often used to implement undo and redo functionality in applications, allowing users to revert or repeat actions.

Overall, stacks are a versatile data structure that can be used in various applications, providing efficient and convenient operations for managing data.

## 2 What are the Way to implements Stack?

There are several ways to implement a stack in programming:

1. **Array-Based Implementation**: This is a simple and direct method where an array is used to store the stack elements. The index of the array can act as the stack pointer. However, the size of the stack is fixed in this case and needs to be defined at the start.
2. **Linked List Implementation**: In this method, a linked list is used to implement a stack. The head of the list can act as the top of the stack. This method allows for a dynamic size of the stack.
3. **Standard Library**: Most programming languages have built-in data types or classes for stack implementation. For example, in Java, you can use the Stack class; in Python, you can use a list as a stack with append() for push operation and pop() for pop operation.

4. **Using Queues**: A stack can also be implemented using two queues. The push operation is done by enqueue operation and the pop operation is done by making the dequeue operation costly.

5. **Wrapper Class**: You can also create a wrapper class around any of the above implementations, providing methods like push(), pop(), peek(), isEmpty(), and isFull().

Remember, the choice of implementation depends on the specific requirements and constraints of your program.

### 2.1 Stack Implementation Using Array

Sure, here's a simple implementation of a stack using an array in Java:

[ ]:
```java
public class Stack {
    private int maxSize;
    private int top;
    private int[] stackArray;

    public Stack(int size) {
        maxSize = size;
        stackArray = new int[maxSize];
        top = -1;
    }

    public void push(int value) {
        if (!isFull()) {
            stackArray[++top] = value;
        } else {
            System.out.println("The stack is full, can't insert value " +␣
↪value);
        }
    }

    public void pop() {
        if (!isEmpty()) {
            top--;
        } else {
            System.out.println("The stack is empty, can't pop a value");
        }
    }

    public int peek() {
        if (!isEmpty()) {
            return stackArray[top];
        } else {
            System.out.println("The stack is empty, can't peek");
            return -1;
        }
```

```
    }

    public boolean isEmpty() {
        return (top == -1);
    }

    public boolean isFull() {
        return (top == maxSize - 1);
    }
}
```

In this code:

- The `push` method inserts a value at the top of the stack.
- The `pop` method removes the value at the top of the stack.
- The `peek` method returns the value at the top of the stack without removing it.
- The `isEmpty` method checks if the stack is empty.
- The `isFull` method checks if the stack is full.

### 2.2 Stack Implementation Using ArrayList

Here's an implementation of a stack using an ArrayList in Java:

[ ]:
```
import java.util.ArrayList;

public class Stack {
    private ArrayList<Integer> stackList;

    public Stack() {
        stackList = new ArrayList<Integer>();
    }

    public void push(int value) {
        stackList.add(value);
    }

    public int pop() {
        if (!isEmpty()) {
            return stackList.remove(stackList.size() - 1);
        } else {
            System.out.println("The stack is empty");
            return -1;
        }
    }

    public boolean isEmpty() {
        return stackList.isEmpty();
    }
```

```
    public int size() {
        return stackList.size();
    }

    public int peek() {
        if (!isEmpty()) {
            return stackList.get(stackList.size() - 1);
        } else {
            System.out.println("The stack is empty");
            return -1;
        }
    }
}

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

        stack.push(10);
        stack.push(20);
        stack.push(30);

        System.out.println("Size of stack: " + stack.size());

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

        System.out.println("Pop element: " + stack.pop());

        System.out.println("Size of stack after pop: " + stack.size());
    }
}
```

This code will output:

[ ]:
```
Size of stack: 3
Top element: 30
Pop element: 30
Size of stack after pop: 2
```

In this code:

- The `push` method adds a value to the end of the ArrayList.
- The `pop` method removes the last value from the ArrayList.
- The `peek` method returns the last value in the ArrayList without removing it.
- The `isEmpty` method checks if the ArrayList is empty.
- The `size` method returns the number of elements in the ArrayList.

### 2.3 Using LinkedList

[ ]:
```java
// Java Code for Linked List Implementation

public class StackAsLinkedList {

	StackNode root;

	static class StackNode {
		int data;
		StackNode next;

		StackNode(int data) { this.data = data; }
	}

	public boolean isEmpty()
	{
		if (root == null) {
			return true;
		}
		else
			return false;
	}

	public void push(int data)
	{
		StackNode newNode = new StackNode(data);

		if (root == null) {
			root = newNode;
		}
		else {
			StackNode temp = root;
			root = newNode;
			newNode.next = temp;
		}
		System.out.println(data + " pushed to stack");
	}

	public int pop()
	{
		int popped = Integer.MIN_VALUE;
		if (root == null) {
			System.out.println("Stack is Empty");
		}
		else {
			popped = root.data;
```

```
                root = root.next;
            }
            return popped;
        }

        public int peek()
        {
            if (root == null) {
                System.out.println("Stack is empty");
                return Integer.MIN_VALUE;
            }
            else {
                return root.data;
            }
        }

        // Driver code
        public static void main(String[] args)
        {

            StackAsLinkedList sll = new StackAsLinkedList();

            sll.push(10);
            sll.push(20);
            sll.push(30);

            System.out.println(sll.pop()
                                        + " popped from stack");

            System.out.println("Top element is " + sll.peek());
        }
}
```

[ ]:
```
public class Stack {
    private int maxSize;
    private int top;
    private int[] stackArray;

    public Stack(int size) {
        maxSize = size;
        stackArray = new int[maxSize];
        top = -1;
    }

    public void push(int value) {
        if (isFull()) {
            System.out.println("Stack is full. Cannot push element.");
```

```
            return;
        }
        stackArray[++top] = value;
    }

    public int pop() {
        if (isEmpty()) {
            System.out.println("Stack is empty. Cannot pop element.");
            return -1;
        }
        return stackArray[top--];
    }

    public int peek() {
        if (isEmpty()) {
            System.out.println("Stack is empty. Cannot peek element.");
            return -1;
        }
        return stackArray[top];
    }

    public boolean isEmpty() {
        return top == -1;
    }

    public boolean isFull() {
        return top == maxSize - 1;
    }
}
```

[ ]:
```
Stack stack = new Stack(10);
stack.push(10);
stack.push(20);
stack.push(30);
stack.push(40);
stack.push(50);
stack.push(60);
```

[ ]:
```
System.out.println("Peek: " + stack.peek());
System.out.println("Pop: " + stack.pop());

System.out.println("Is stack empty? " + stack.isEmpty());
System.out.println("Is stack full? " + stack.isFull());

stack.push(70);
stack.push(80);
stack.push(90);
```

```
System.out.println("Peek: " + stack.peek());
System.out.println("Pop: " + stack.pop());

System.out.println("Is stack empty? " + stack.isEmpty());
System.out.println("Is stack full? " + stack.isFull());
```

```
Peek: 50
Pop: 50

Is stack empty? false
Is stack full? false
Peek: 90
Pop: 90
Is stack empty? false
Is stack full? false
```

here's an example of a simple stack implementation in Java, including the main operations: push, pop, peek, isEmpty, and isFull.

[ ]:
```
class Stack {
    private int maxSize;
    private int top;
    private int[] stackArray;

    public Stack(int size) {
        maxSize = size;
        stackArray = new int[maxSize];
        top = -1;
    }

    public void push(int value) {
        if (top < maxSize - 1) {
            stackArray[++top] = value;
            System.out.println("Pushed " + value + " to the stack");
        } else {
            System.out.println("Stack is full. Can't push " + value);
        }
    }

    public int pop() {
        if (top >= 0) {
            System.out.println("Popped " + stackArray[top] + " from the stack");
            return stackArray[top--];
        } else {
            System.out.println("Stack is empty. Can't pop");
            return -1;
        }
```

```
    }

    public int peek() {
        if (top >= 0) {
            System.out.println("Peek: " + stackArray[top]);
            return stackArray[top];
        } else {
            System.out.println("Stack is empty. Can't peek");
            return -1;
        }
    }

    public boolean isEmpty() {
        return (top == -1);
    }

    public boolean isFull() {
        return (top == maxSize - 1);
    }
}

public class Main {
    public static void main(String[] args) {
        Stack stack = new Stack(3);

        stack.push(10);
        stack.push(20);
        stack.push(30);

        stack.peek();

        stack.pop();

        System.out.println("Is stack empty? " + stack.isEmpty());
        System.out.println("Is stack full? " + stack.isFull());
    }
}
```

Stacks are used in a variety of applications in programming, including:

1. **Function Call Stack**: Stacks are used to manage function calls and returns. When a function is called, its details (like return address, local variables) are pushed onto a stack. When the function returns, these details are popped from the stack.
2. **Expression Evaluation and Syntax Parsing**: Stacks are used in compilers for syntax checking of expressions and their evaluation. They help in checking the proper opening and closing of brackets, parsing postfix expressions, etc.
3. **Backtracking**: Stacks are used to solve problems where you need to perform actions and

then undo them, such as in depth-first search, finding paths in a maze, the "undo" feature in text editors, and backtracking in algorithms.

4. **Memory Management**: Stacks are used in managing memory in modern operating systems. Each thread has a stack associated with it.
5. **Browser History**: Web browsers use stacks to manage the history of visited web pages. The "back" button pops the top page from the stack.

These are just a few examples. Stacks are a fundamental data structure and are used in many more scenarios.

This code creates a stack of size 3, pushes three integers onto it, peeks at the top element, pops an element, and checks if the stack is empty or full.

Here's a brief explanation of the stack operations:

1. `push`: This operation adds an element to the top of the stack. In the provided code, `stack.push(70);` is pushing the integer 70 onto the top of the stack.
2. `pop`: This operation removes the top element from the stack and returns it. In the code, `stack.pop();` is removing the top element from the stack.
3. `peek` or `top`: This operation returns the top element of the stack without removing it. In the code, `stack.peek();` is returning the top element of the stack.
4. `isEmpty`: This operation checks if the stack is empty. If the stack is empty, it returns true; otherwise, it returns false. In the code, `stack.isEmpty();` is checking if the stack is empty.
5. `isFull`: This operation checks if the stack is full. If the stack is full, it returns true; otherwise, it returns false. In the code, `stack.isFull();` is checking if the stack is full.

These are the basic operations that can be performed on a stack. They allow us to add, remove, and inspect elements in a Last-In-First-Out (LIFO) manner.

## 3 Stack Using Java Collection FrameWork

In Java, the `java.util.Stack` class is part of the Java Collection Framework and extends the `Vector` class. It provides the basic push, pop, and peek operations, as well as several methods for searching for elements in the stack.

Here's an example of how to use it:

[ ]:
```java
import java.util.*;

public class Stack {
    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<>();

        stack.push(10);
        stack.push(20);
        stack.push(30);
```

```
        System.out.println("Current stack: " + stack);

        int poppedElement = stack.pop();
        System.out.println("Popped element: " + poppedElement);

        int topElement = stack.peek();
        System.out.println("Top element: " + topElement);

        System.out.println("Final stack: " + stack);
    }
}
```

```
|           Stack<Integer> stack = new Stack<>();
type Stack does not take parameters

|           Stack<Integer> stack = new Stack<>();
cannot infer type arguments for Stack
  reason: cannot use '<>' with non-generic class Stack
```

It seems like there's a naming conflict in your code. You have a class named `Stack` and you're also trying to use `java.util.Stack`.

The `Stack` in `Stack<Integer> stack = new Stack<>();` is referring to your own class `Stack`, not `java.util.Stack`.

To fix this, you can either rename your class to something else, or you can use the fully qualified class name for `java.util.Stack` when declaring your stack. Here's how you can do it:

[ ]:
```
import java.util.Stack;

public class MyStack {
    public static void main(String[] args) {
        java.util.Stack<Integer> stack = new java.util.Stack<>();

        stack.push(10);
        stack.push(20);
        stack.push(30);

        System.out.println("Current stack: " + stack);

        int poppedElement = stack.pop();
        System.out.println("Popped element: " + poppedElement);

        int topElement = stack.peek();
        System.out.println("Top element: " + topElement);
```

```
        System.out.println("Final stack: " + stack);
    }
}

MyStack stack = new MyStack();
```

In this code, I've renamed your class to `MyStack` to avoid the naming conflict.

This code will output:

[ ]:
```
Current stack: [10, 20, 30]
Popped element: 30
Top element: 20
Final stack: [10, 20]
```

In this code:

- The `push` method adds an element to the top of the stack.
- The `pop` method removes the top element from the stack and returns it.
- The `peek` method returns the top element from the stack without removing it.

## 4 Thank You!