Java Stack Implementation

Explain the concept of stack in simple but detail. Then implement them using java, give full code and implement all the methods. Then perform all the operation in main() method. Use primitive

Concept of Stack

A **stack** is a linear data structure that follows the **Last In, First Out (LIFO)** principle. This means that the last element added to the stack is the first one to be removed. Think of it like a stack of plates: you can only add or remove plates from the top of the stack.

Key Operations

1. **Push**: Add an element to the top of the stack.
2. **Pop**: Remove and return the top element from the stack.
3. **Peek**: Return the top element without removing it.
4. **isEmpty**: Check if the stack is empty.

Java Implementation

Here's a simple implementation of a stack using a primitive array:

java

```java
public class Stack {
    private int[] stack;
    private int top;
    private int capacity;

    // Constructor to initialize stack
    public Stack(int size) {
        capacity = size;
        stack = new int[capacity];
        top = -1; // Initial state of stack is empty
    }

    // Push an element onto the stack
    public void push(int value) {
        if (top == capacity - 1) {
            System.out.println("Stack is full");
            return;
        }
        stack[++top] = value;
    }

    // Pop an element from the stack
    public int pop() {
        if (isEmpty()) {
            System.out.println("Stack is empty");
            return -1; // Indicates stack is empty
        }
        return stack[top--];
    }
```

```java
    // Peek at the top element of the stack
    public int peek() {
        if (isEmpty()) {
            System.out.println("Stack is empty");
            return -1; // Indicates stack is empty
        }
        return stack[top];
    }

    // Check if the stack is empty
    public boolean isEmpty() {
        return top == -1;
    }

    // Display the elements in the stack
    public void display() {
        if (isEmpty()) {
            System.out.println("Stack is empty");
            return;
        }
        System.out.print("Stack elements: ");
        for (int i = 0; i <= top; i++) {
            System.out.print(stack[i] + " ");
        }
        System.out.println();
    }

    // Main method to demonstrate stack operations
    public static void main(String[] args) {
        Stack myStack = new Stack(5); // Create a stack of size 5

        myStack.push(10);
        myStack.push(20);
        myStack.push(30);
        myStack.push(40);
        myStack.push(50);

        myStack.display(); // Display stack

        System.out.println("Top element is: " + myStack.peek()); // Peek at top element

        System.out.println("Popped element is: " + myStack.pop()); // Pop element
        myStack.display(); // Display stack after pop

        myStack.push(60); // Push another element
        myStack.display(); // Display stack after push
    }
}
```

Explanation of the Code

1. **Constructor**: Initializes the stack with a given size and sets the top index to -1, indicating an empty stack.
2. **push()**: Adds an element to the stack, provided it's not full. It increments the top index and places the new element at this index.
3. **pop()**: Removes and returns the top element from the stack. It decrements the top index.
4. **peek()**: Returns the top element without modifying the stack.
5. **isEmpty()**: Checks if the stack is empty by verifying if the top index is -1.
6. **display()**: Prints all elements in the stack from bottom to top.
7. **main()**: Demonstrates the usage of the stack by performing various operations and displaying the stack's state.

This implementation uses a fixed-size array to store stack elements and performs basic stack operations.

Now give me 5 most asked coding question on stack and give code as well. First explain the problem statements using example then write code. Also explain about complexity if required. And give multiple solution if more ways can be important to know.

5 Most Asked Coding Questions on Stack

Here are five commonly asked coding questions related to stacks, including problem statements, explanations with examples, code implementations, and complexities.

---

1. Valid Parentheses

Problem Statement: Given a string containing just the characters `` `'('` ``, `` `')'` ``, `` `'{'` ``, `` `'}'` ``, `` `'['` ``, and `` `']'` ``, determine if the input string is valid. An input string is valid if the brackets are closed properly and nested correctly.
Example:

- Input: `` `"()[]{}"` ``
- Output: `` `true` ``
- Input: `` `"([)]"` ``
- Output: `` `false` ``

Explanation: The string `` `()[]{} ` ``is valid because all types of brackets are closed in the correct order. `` `([)]` `` is invalid because the brackets are not nested properly.
Java Code:

```java
import java.util.Stack;

public class ValidParentheses {
    public static boolean isValid(String s) {
        Stack<Character> stack = new Stack<>();
        for (char c : s.toCharArray()) {
            switch (c) {
                case '(': case '[': case '{':
                    stack.push(c);
                    break;
                case ')':
                    if (stack.isEmpty() || stack.pop() != '(') return false;
                    break;
                case ']':
                    if (stack.isEmpty() || stack.pop() != '[') return false;
                    break;
                case '}':
                    if (stack.isEmpty() || stack.pop() != '{') return false;
                    break;
            }
        }
        return stack.isEmpty();
```

```
    }

    public static void main(String[] args) {
        System.out.println(isValid("()[]{}")); // true
        System.out.println(isValid("([)]"));   // false
    }
}
```

Complexity:

- **Time Complexity**: O(n), where n is the length of the string.
- **Space Complexity**: O(n) due to the stack.

---

2. Daily Temperatures

Problem Statement: Given a list of daily temperatures, return a list where each element is the number of days you have to wait until a warmer temperature. If there is no future day for which this is possible, put 0 instead.
Example:

- Input: `[73, 74, 75, 71, 69, 72, 76, 73]`
- Output: `[1, 1, 4, 2, 1, 1, 0, 0]`

Explanation: For each day, find how many days until a warmer temperature.
Java Code:

```java
import java.util.Stack;

public class DailyTemperatures {
    public static int[] dailyTemperatures(int[] T) {
        int[] result = new int[T.length];
        Stack<Integer> stack = new Stack<>();
        for (int i = 0; i < T.length; i++) {
            while (!stack.isEmpty() && T[i] > T[stack.peek()]) {
                int idx = stack.pop();
                result[idx] = i - idx;
            }
            stack.push(i);
        }
        return result;
    }

    public static void main(String[] args) {
        int[] temperatures = {73, 74, 75, 71, 69, 72, 76, 73};
        int[] result = dailyTemperatures(temperatures);
        for (int r : result) {
            System.out.print(r + " ");
        }
    }
}
```

Complexity:

- **Time Complexity**: O(n), where n is the length of the temperature list.
- **Space Complexity**: O(n) due to the stack.

---

3. Min Stack

Problem Statement: Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.
Example:

- Operations: **`push(-2)`**, **`push(0)`**, **`push(-3)`**, **`getMin()`**, **`pop()`**, **`top()`**, **`getMin()`**
- Output: **`-3`**, **`-2`**

Explanation: The minimum element in the stack should be retrievable in constant time.
Java Code:

```java
import java.util.Stack;

public class MinStack {
    private Stack<Integer> stack = new Stack<>();
    private Stack<Integer> minStack = new Stack<>();

    public void push(int x) {
        stack.push(x);
        if (minStack.isEmpty() || x <= minStack.peek()) {
            minStack.push(x);
        }
    }

    public void pop() {
        int val = stack.pop();
        if (val == minStack.peek()) {
            minStack.pop();
        }
    }

    public int top() {
        return stack.peek();
    }

    public int getMin() {
        return minStack.peek();
    }

    public static void main(String[] args) {
        MinStack minStack = new MinStack();
        minStack.push(-2);
        minStack.push(0);
        minStack.push(-3);
        System.out.println(minStack.getMin()); // -3
        minStack.pop();
        System.out.println(minStack.top());    // 0
        System.out.println(minStack.getMin()); // -2
    }
}
```

Complexity:

- **Time Complexity**: O(1) for all operations.
- **Space Complexity**: O(n) for the stack and minStack.

---

4. Evaluate Reverse Polish Notation

Problem Statement: Evaluate the value of an arithmetic expression in Reverse Polish Notation (RPN). Valid operators are `+`, `-`, `*`, and `/`. Each operand may be an integer.

Example:

- Input: `["2", "1", "+", "3", "*"]`
- Output: `9`

Explanation: The expression is equivalent to `(2 + 1) * 3`, which equals `9`.

Java Code:

```java
import java.util.Stack;

public class EvaluateRPN {
    public static int evalRPN(String[] tokens) {
        Stack<Integer> stack = new Stack<>();
        for (String token : tokens) {
            if (token.equals("+") || token.equals("-") || token.equals("*") ||
token.equals("/")) {
                int b = stack.pop();
                int a = stack.pop();
                switch (token) {
                    case "+":
                        stack.push(a + b);
                        break;
                    case "-":
                        stack.push(a - b);
                        break;
                    case "*":
                        stack.push(a * b);
                        break;
                    case "/":
                        stack.push(a / b);
                        break;
                }
            } else {
                stack.push(Integer.parseInt(token));
            }
        }
        return stack.pop();
    }

    public static void main(String[] args) {
        String[] tokens = {"2", "1", "+", "3", "*"};
        System.out.println(evalRPN(tokens)); // 9
    }
}
```

Complexity:

- **Time Complexity**: O(n), where n is the number of tokens.
- **Space Complexity**: O(n) due to the stack.

---

5. Largest Rectangle in Histogram

Problem Statement: Given an array of integers representing the heights of bars in a histogram, find the area of the largest rectangle that can be formed in the histogram.

Example:

- Input: `[2, 1, 5, 6, 2, 3]`

- Output: `10`

Explanation: The largest rectangle has a height of 2 and a width of 5 (spanning indices 2 to 6).
Java Code:

```java
import java.util.Stack;

public class LargestRectangle {
    public static int largestRectangleArea(int[] heights) {
        Stack<Integer> stack = new Stack<>();
        int maxArea = 0, index = 0;

        while (index < heights.length) {
            if (stack.isEmpty() || heights[stack.peek()] <= heights[index]) {
                stack.push(index++);
            } else {
                int topOfStack = stack.pop();
                int area = heights[topOfStack] * (stack.isEmpty() ? index : index -
stack.peek() - 1);
                maxArea = Math.max(maxArea, area);
            }
        }

        while (!stack.isEmpty()) {
            int topOfStack = stack.pop();
            int area = heights[topOfStack] * (stack.isEmpty() ? index : index -
stack.peek() - 1);
            maxArea = Math.max(maxArea, area);
        }

        return maxArea;
    }

    public static void main(String[] args) {
        int[] heights = {2, 1, 5, 6, 2, 3};
        System.out.println(largestRectangleArea(heights)); // 10
    }
}
```

Complexity:

- **Time Complexity**: O(n), where n is the number of bars in the histogram.
- **Space Complexity**: O(n) due to the stack.

These questions and their solutions cover a variety of important concepts related to stacks and demonstrate how to handle different scenarios using this data structure.

Give 2 more

Here are two additional commonly asked coding questions related to stacks:

---

6. Next Greater Element

Problem Statement: Given an array of integers, find the next greater element for each element in the array. The next greater element for an element `x` is the first greater element that appears to the right of `x` in the array. If no such element exists, output `-1` for that element.
Example:

- Input: `[4, 5, 2, 10]`
- Output: `[5, 10, 10, -1]`

Explanation:

- For `4`, the next greater element is `5`.
- For `5`, the next greater element is `10`.
- For `2`, the next greater element is `10`.
- For `10`, there is no greater element, so it is `-1`.

Java Code:

```java
import java.util.Stack;

public class NextGreaterElement {
    public static int[] nextGreaterElements(int[] nums) {
        int n = nums.length;
        int[] result = new int[n];
        Stack<Integer> stack = new Stack<>();

        for (int i = 2 * n - 1; i >= 0; i--) {
            while (!stack.isEmpty() && stack.peek() <= nums[i % n]) {
                stack.pop();
            }
            result[i % n] = stack.isEmpty() ? -1 : stack.peek();
            stack.push(nums[i % n]);
        }

        return result;
    }

    public static void main(String[] args) {
        int[] nums = {4, 5, 2, 10};
        int[] result = nextGreaterElements(nums);
        for (int r : result) {
            System.out.print(r + " ");
        }
    }
}
```

Complexity:

- **Time Complexity**: O(n), where n is the number of elements in the array. Each element is pushed and popped from the stack at most once.
- **Space Complexity**: O(n) due to the stack and result array.

---

7. Sort a Stack

Problem Statement: Given a stack, sort it such that the smallest elements are on the top and the largest elements are at the bottom. You can use an additional stack to help with sorting, but you cannot use any other data structures.

Example:

- Input: `[3, 1, 4, 2]`
- Output: `[1, 2, 3, 4]`

Explanation: The stack should be sorted in ascending order.
Java Code:

```java
import java.util.Stack;

public class SortStack {
    public static Stack<Integer> sort(Stack<Integer> stack) {
        Stack<Integer> tempStack = new Stack<>();
        while (!stack.isEmpty()) {
            int temp = stack.pop();
            while (!tempStack.isEmpty() && tempStack.peek() > temp) {
                stack.push(tempStack.pop());
            }
            tempStack.push(temp);
        }
        return tempStack;
    }

    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<>();
        stack.push(3);
        stack.push(1);
        stack.push(4);
        stack.push(2);

        Stack<Integer> sortedStack = sort(stack);
        while (!sortedStack.isEmpty()) {
            System.out.print(sortedStack.pop() + " ");
        }
    }
}
```

Complexity:

- **Time Complexity**: O(n^2), where n is the number of elements in the stack. Each element may be pushed and popped multiple times.
- **Space Complexity**: O(n) due to the use of an additional stack for sorting.

These problems provide further practice with stack operations and demonstrate different ways to handle and manipulate stacks in practical scenarios.

Now give list of most commonly asked question on stack in interviews. These have to be non coding questions.

Here's a list of commonly asked non-coding interview questions about stacks:

1. What is a Stack?

Question: Explain the stack data structure and its key operations.
Explanation: A stack is a linear data structure that follows the Last In, First Out (LIFO) principle. The primary operations are:

- **Push**: Add an element to the top of the stack.
- **Pop**: Remove and return the top element of the stack.
- **Peek**: Return the top element without removing it.
- **isEmpty**: Check if the stack is empty.

2. What are the Applications of Stacks?

Question: Describe some practical applications of stacks.
Explanation:

- **Function Call Management**: Stacks are used in programming languages to manage function calls and local variables.
- **Undo Mechanisms**: Used in applications like text editors to implement undo functionality.
- **Expression Evaluation**: Used for evaluating expressions in postfix notation (Reverse Polish Notation).
- **Syntax Parsing**: Used in compilers for syntax parsing and to check for balanced parentheses.

3. What is the Difference Between a Stack and a Queue?

Question: Compare stacks and queues.
Explanation:

- **Stack**: Follows Last In, First Out (LIFO) principle. Only the top element is accessible.
- **Queue**: Follows First In, First Out (FIFO) principle. Elements are added at the rear and removed from the front.

4. Explain How to Implement a Stack Using Queues.

Question: Describe how you would implement a stack using two queues.
Explanation:

- **Approach 1**:
  - **Push Operation**: Always enqueue the new element into an empty queue, then dequeue all elements from the other queue and enqueue them to the new queue.
  - **Pop Operation**: Dequeue the front element from the non-empty queue.
- **Approach 2**:
  - **Push Operation**: Enqueue the new element into one queue.
  - **Pop Operation**: Move all elements except the last one to another queue, then dequeue the last element from the original queue and swap the two queues.

5. What is the Time Complexity of Stack Operations?

Question: Discuss the time complexity for common stack operations.
Explanation:

- **Push**: O(1) - Adding an element to the stack takes constant time.
- **Pop**: O(1) - Removing the top element from the stack takes constant time.
- **Peek**: O(1) - Accessing the top element without removing it takes constant time.

- **isEmpty**: O(1) - Checking if the stack is empty takes constant time.

6. What Are the Limitations of Using a Stack?

Question: Discuss some limitations or drawbacks of using stacks.
Explanation:

- **Limited Access**: You can only access the top element directly. To access other elements, you need to pop elements off the stack.
- **Fixed Size**: In some implementations, stacks have a fixed size, which can lead to overflow if not managed properly.
- **Memory Consumption**: For large data sets, stack operations may lead to significant memory usage.

7. How Can a Stack Be Used to Reverse a String?

Question: Explain how a stack can be used to reverse a string.
Explanation:

- Push all characters of the string onto the stack.
- Pop characters from the stack one by one and append them to a new string. This reversed string is the result.

8. Describe the Stack Overflow and Underflow Conditions.

Question: What are stack overflow and stack underflow, and how do they occur?
Explanation:

- **Stack Overflow**: Occurs when trying to push an element onto a stack that is already full (in a fixed-size stack).
- **Stack Underflow**: Occurs when trying to pop an element from an empty stack.

9. Explain the Use of a Stack in Expression Parsing.

Question: How is a stack used in parsing expressions, such as converting infix expressions to postfix?
Explanation: Stacks are used in expression parsing to handle operators and operands efficiently:

- **Infix to Postfix Conversion**: Use a stack to keep track of operators and ensure that they are placed in the correct order in the postfix expression.
- **Evaluation**: Use a stack to evaluate expressions in postfix notation by pushing operands and performing operations when operators are encountered.

10. How Do You Implement a Stack in Recursion?

Question: Explain how recursion can be used to simulate a stack.
Explanation: Recursion inherently uses a call stack to manage function calls. Each recursive call is pushed onto the call stack and popped off when the function completes, simulating stack behavior.
These questions cover theoretical aspects and practical uses of stacks, helping interviewers assess a candidate's understanding of stack operations, applications, and related concepts.