### 1:- Implement a Stack using an Array

**Problem: Implement a basic stack using an array with push, pop, and peek operations.**

**public** **class** StackDemo {

**static** **class** CustomStack {

**int**[] data;

**int** tos;

**public** CustomStack(**int** n) {

data = **new** **int**[n];

tos = -1;

}

**public** **int** size() {

**return** tos + 1;

}

**void** display() {

**if** (tos == -1) {

System.***out***.println("stack is empty");

**return**;

}

**for** (**int** i = tos; i >= 0; i--) {

System.***out***.print(data[i] + " ");

}

System.***out***.println();

}

**public** **void** push(**int** val) {

**if** (tos == data.length - 1) {

System.***out***.println("stack is overflow");

} **else** {

tos++;

data[tos] = val;

}

}

**public** **int** top() {

**if** (tos == -1) {

System.***out***.println("stack is underflow");

**return** -1;

} **else** {

**int** val = data[tos];

**return** val;

}

}

**public** **int** pop() {

**if** (tos == -1) {

System.***out***.println("stack is underflow");

**return** -1;

} **else** {

**int** val = data[tos];

tos--;

**return** val;

}

}

}

**public** **static** **void** main(String[] args) {

CustomStack st = **new** CustomStack(5);

st.push(1);

st.push(2);

st.push(3);

st.push(4);

st.display();

System.***out***.println("pop: top element will get remove");

st.pop();

st.display();

}

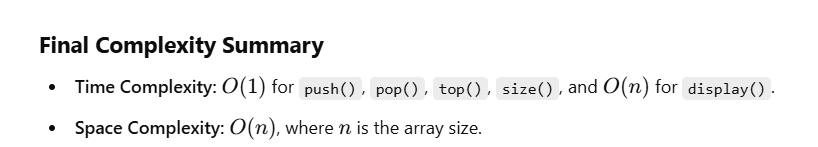
}

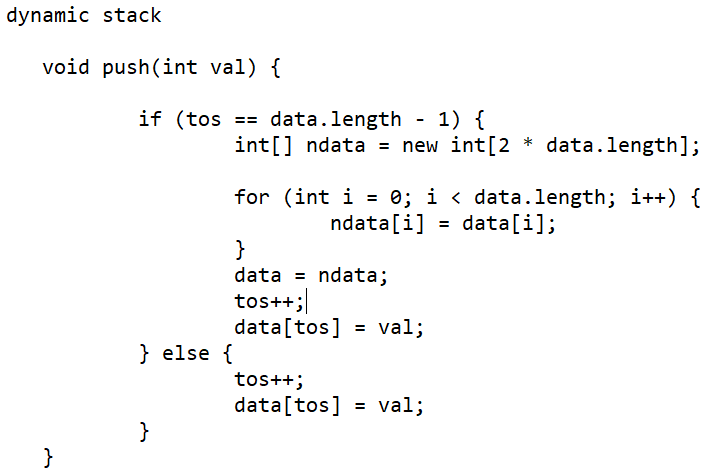
**//output**

4 3 2 1

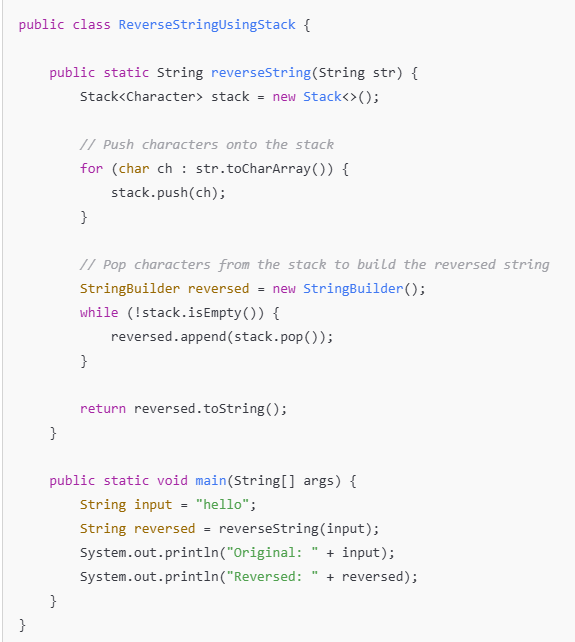
pop: top element will get remove

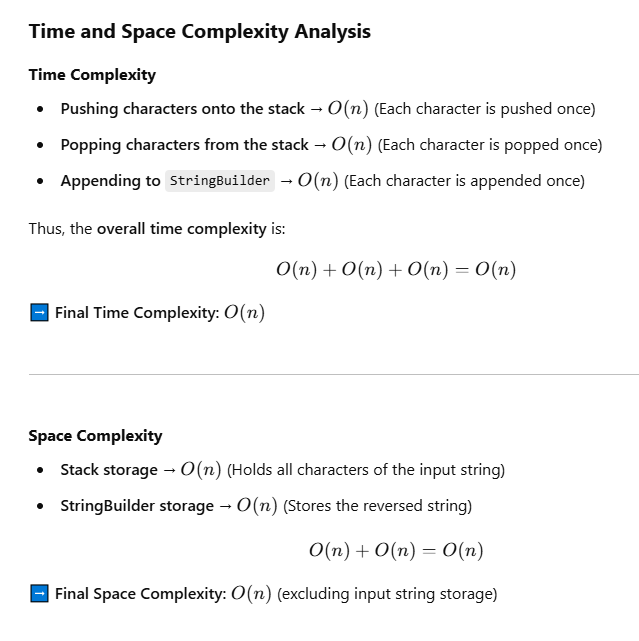
3 2 1

****



2:- Reverse string using stack





3:- stack balanced expression :-

**package** com.example.demo;

**import** java.util.Stack;

**public** **class** StackBalancedExpression {

**private** **static** **boolean** checkBalancedExpression(String expression1) {

Stack<Character> st = **new** Stack();

**for** (**char** ch : expression1.toCharArray()) {

**if** (ch == '(' || ch == '{' || ch == '[') {

st.push(ch);

} **else** **if** (ch == ')' || ch == '}' || ch == ']') {

**if** (st.isEmpty()) {

**return** **false**;

}

**char** top = st.pop();

**if** ((ch == ')' && top != '(') || (ch == '}' && top != '{') || (ch == ']' && top != '[')) {

**return** **false**;

}

}

}

**return** st.isEmpty();

}

**public** **static** **void** main(String[] args) {

String expression1 = "{[()]}"; // balanced

**boolean** balancedExpression = *checkBalancedExpression*(expression1);

**if** (balancedExpression) {

System.***out***.println("expression is balanced");

} **else** {

System.***out***.println("expression is not balanced");

}

}

}

**Complexity Analysis:**

* **Time Complexity**: O(N)
* **Space Complexity**: O(N)

4:- next greater element

**package** com.example.demo;

**import** java.util.Stack;

**public** **class** NextGreaterElement {

**public** **static** **void** main(String[] args) {

**int**[] arr = { 2, 5, 9, 3, 1, 12, 6, 8, 7 };

**int**[] nextGE = *nextGE*(arr);

**for** (**int** val : nextGE) {

System.***out***.print(val + " ");

}

}

**private** **static** **int**[] nextGE(**int**[] arr) {

**int**[] nge = **new** **int**[arr.length];

Stack<Integer> st = **new** Stack<>();

st.push(arr[arr.length - 1]);

nge[arr.length - 1] = -1;

**for** (**int** i = arr.length - 2; i >= 0; i--) {

**while** (st.size() > 0 && arr[i] > st.peek()) {

st.pop();

}

**if** (st.size() == 0) {

nge[i] = -1;

} **else** {

nge[i] = st.peek();

}

st.push(arr[i]);

}

**return** nge;

}

}

// output 5 9 12 12 12 -1 8 -1 -1

**Complexity Analysis:**

* **Time Complexity**: O(N)
* **Space Complexity**: O(N)

### 5:--- Find Minimum Element in Stack in O(1)

**Problem: Implement a stack that supports getMin() in O(1) time.**

**package** com.example.demo;

**import** java.util.Stack;

**public** **class** MinElementStack {

**static** **class** MinStack {

Stack<Integer> stack = **new** Stack();

Stack<Integer> minStack = **new** Stack();

**public** **void** push(**int** item) {

stack.push(item);

**if** (minStack.isEmpty() || item < minStack.peek()) {

minStack.push(item);

}

}

**public** **int** pop() {

**if** (stack.isEmpty()) {

**throw** **new** IllegalStateException("stack is underflow");

}

**int** item = stack.pop();

**if** (item == minStack.peek()) {

minStack.pop();

}

**return** item;

}

**public** **int** getMin() {

**if** (stack.isEmpty()) {

**throw** **new** IllegalStateException("stack is underflow");

}

**return** minStack.peek();

}

}

**public** **static** **void** main(String[] args) {

MinStack ms = **new** MinStack();

ms.push(5);

ms.push(3);

ms.push(2);

ms.push(8);

System.***out***.println(ms.getMin());

}

}

**// output is 2**

**Complexity Analysis:**

* **Time Complexity**: O(1)
* **Space Complexity**: O(N)

6:- stock span problem :

Note :- push index into stack

Find maximum towards left :----

if found then max index- today price index

if not found then index +1

**package** com.example.demo;

**import** java.util.Scanner;

**import** java.util.Stack;

/\*

\*

\* 1. You are given a number n, representing the size of array a.

2. You are given n numbers, representing the prices of a share on n days.

3. You are required to find the stock span for n days.

4. Stock span is defined as the number of days passed between the current day and the first day before today when price was higher than today.

e.g.

for the array [2 5 9 3 1 12 6 8 7]

span for 2 is 1

span for 5 is 2

span for 9 is 3

span for 3 is 1

span for 1 is 1

span for 12 is 6

span for 6 is 1

span for 8 is 2

span for 7 is 1

\*input 5

output

2

5

9

3

1

\*/

**public** **class** StockSpan {

**private** **static** **int**[] stockSpan(**int**[] arr) {

**int**[] span = **new** **int**[arr.length];

Stack<Integer> st = **new** Stack<>();

st.push(0);

span[0] = 1;

**for** (**int** i = 1; i < arr.length; i++) {

**while** (st.size() > 0 && arr[i] > arr[st.peek()]) {

st.pop();

}

**if** (st.size() == 0) {

span[i] = i + 1;

} **else** {

span[i] = i - st.peek();

}

st.push(i);

}

**return** span;

}

**public** **static** **void** print(**int**[] a) {

**for** (**int** val : a) {

System.***out***.println(val);

}

}

**public** **static** **void** main(String[] args) {

**int**[] arr = { 2, 5, 9, 3, 1, 12, 6, 8, 7 };

**int**[] stockSpan = *stockSpan*(arr);

**for** (**int** val : stockSpan) {

System.***out***.print(val + " ");

}

}

}

7: Implement stack using linked list

**Implementing Stack Using Linked List in Java (Simple Way)**

A stack is a **Last In, First Out (LIFO)** data structure. We can implement it using a **singly linked list**, where:

* The **top** of the stack corresponds to the **head** of the linked list.
* **Push (Insert)** operation adds a node at the beginning.
* **Pop (Remove)** operation removes a node from the beginning.

**package** com.example.demo;

//Node class to represent each element in the stack

**class** Node {

**int** data;

Node next;

// Constructor to initialize a new node

Node(**int** data) {

**this**.data = data;

**this**.next = **null**;

}

}

**class** StackLinkedList {

**private** Node head;

// Push operation (Insert at the beginning)

**public** **void** push(**int** data) {

Node newNode = **new** Node(data);

newNode.next = head;

head = newNode;

System.***out***.println(data + " pushed to stack");

}

// Pop operation (Remove from the beginning)

**public** **int** pop() {

**if** (head == **null**) {

System.***out***.println("Stack is empty. Cannot pop.");

**return** -1;

}

**int** poppedValue = head.data; // Get the top (head) data

head = head.next; // Move head to next node

**return** poppedValue;

}

// Peek operation (View top element)

**public** **int** peek() {

**if** (head == **null**) {

System.***out***.println("Stack is empty. No top element.");

**return** -1;

}

**return** head.data;

}

// Check if stack is empty

**public** **boolean** isEmpty() {

**return** head == **null**;

}

// Display stack elements

**public** **void** display() {

**if** (head == **null**) {

System.***out***.println("Stack is empty.");

**return**;

}

Node temp = head;

System.***out***.print("Stack: ");

**while** (temp != **null**) {

System.***out***.print(temp.data + " -> ");

temp = temp.next;

}

System.***out***.println("NULL");

}

//Main class to test stack

**public** **static** **void** main(String[] args) {

StackLinkedList stack = **new** StackLinkedList();

stack.push(10);

stack.push(20);

stack.push(30);

stack.display(); // Output: 30 -> 20 -> 10 -> NULL

System.***out***.println("Top element: " + stack.peek()); // Output: 30

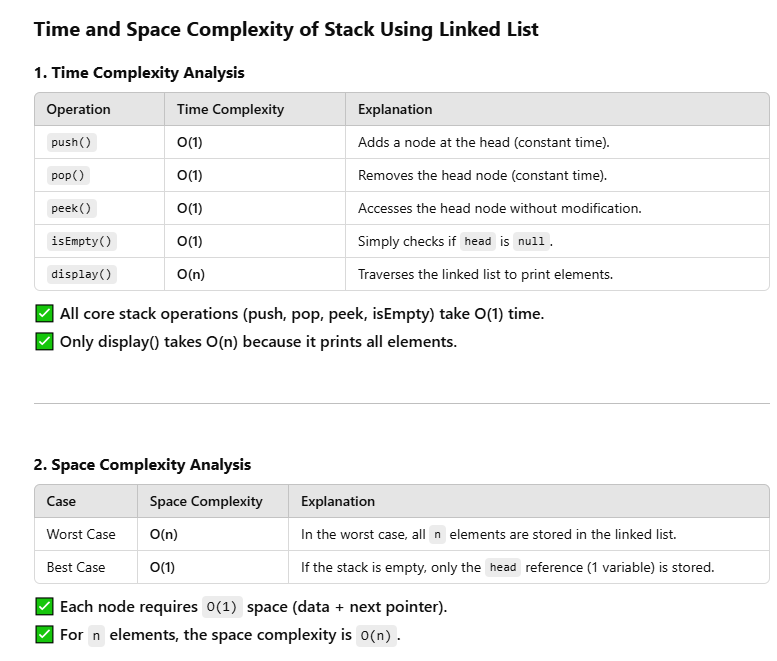
System.***out***.println("Popped element: " + stack.pop()); // Output: 30

stack.display(); // Output: 20 -> 10 -> NULL

System.***out***.println("Top element after pop: " + stack.peek()); // Output: 20

}

}



8: stack using queue

Here’s how you can implement the **first approach** where the **push operation is costly**, i.e., we ensure that the most recent element is always at the front of the primary queue (q1):

**Steps**

1. **Push(x)**: Add the new element to q2, then move all elements from q1 to q2 so that the most recent element will be at the front of q1.
2. **Pop()**: Simply dequeue from q1 (since the most recent element is at the front).
3. **Top()**: Peek the front of q1.
4. **isEmpty()**: Check if q1 is empty.

**package** com.example.demo;

**import** java.util.LinkedList;

**import** java.util.Queue;

**public** **class** StackUsingQueue {

**private** Queue<Integer> q1 = **new** LinkedList<>();

**private** Queue<Integer> q2 = **new** LinkedList<>();

// Push operation (costly)

**public** **void** push(**int** x) {

// Step 1: Add the new element to q2

q2.add(x);

// Step 2: Move all elements from q1 to q2

**while** (!q1.isEmpty()) {

q2.add(q1.poll());

}

// Step 3: Swap q1 and q2

Queue<Integer> temp = q1;

q1 = q2;

q2 = temp;

}

// Pop operation (simple)

**public** **int** pop() {

**if** (q1.isEmpty()) {

**throw** **new** RuntimeException("Stack is empty");

}

**return** q1.poll();

}

// Top operation (peek the front of q1)

**public** **int** top() {

**if** (q1.isEmpty()) {

**throw** **new** RuntimeException("Stack is empty");

}

**return** q1.peek();

}

// Check if the stack is empty

**public** **boolean** isEmpty() {

**return** q1.isEmpty();

}

**public** **static** **void** main(String[] args) {

StackUsingQueue stack = **new** StackUsingQueue();

stack.push(10);

stack.push(20);

stack.push(30);

System.***out***.println("Top element: " + stack.top()); // 30

System.***out***.println("Popped element: " + stack.pop()); // 30

System.***out***.println("Popped element: " + stack.pop()); // 20

System.***out***.println("Is stack empty? " + stack.isEmpty()); // false

System.***out***.println("Popped element: " + stack.pop()); // 10

System.***out***.println("Is stack empty? " + stack.isEmpty()); // true

}

}

=================imp walkthrough

**Walkthrough Example**

Let’s walk through an example with a better understanding of how the queues behave.

**Step 1: Push(10)**

* **Before the push**:  
  q1: [], q2: []
* **After adding 10 to q2**:  
  q1: [], q2: [10]
* **Swap q1 and q2**:  
  q1: [10], q2: []

**Step 2: Push(20)**

* **Before the push**:  
  q1: [10], q2: []
* **After adding 20 to q2**:  
  q1: [10], q2: [20]
* **Move elements from q1 to q2**:  
  q1: [], q2: [20, 10]
* **Swap q1 and q2**:  
  q1: [20, 10], q2: []

**Step 3: Push(30)**

* **Before the push**:  
  q1: [20, 10], q2: []
* **After adding 30 to q2**:  
  q1: [20, 10], q2: [30]
* **Move elements from q1 to q2**:  
  q1: [], q2: [30, 20, 10]
* **Swap q1 and q2**:  
  q1: [30, 20, 10], q2: []

Now, q1 contains the stack in the correct order, with 30 at the front.