Day 14 - 04th july 2025

**Task 1:**

Create a custom node, add elements to it and traverse it..

**Answer:**

class Node<T> {

T data;

Node<T> next;

public Node(T data) {

this.data = data;

this.next = null;

}

}

public class CustomLinkedList<T> {

private Node<T> head;

private int size;

public CustomLinkedList() {

this.head = null;

this.size = 0;

}

// Add element at the end

public void add(T data) {

Node<T> newNode = new Node<>(data);

if (head == null) {

head = newNode;

} else {

Node<T> current = head;

while (current.next != null) {

current = current.next;

}

current.next = newNode;

}

size++;

}

// Add element at the beginning

public void addFirst(T data) {

Node<T> newNode = new Node<>(data);

newNode.next = head;

head = newNode;

size++;

}

// Add element at specific index

public void add(int index, T data) {

if (index < 0 || index > size) {

throw new IndexOutOfBoundsException("Index out of bounds");

}

if (index == 0) {

addFirst(data);

return;

}

Node<T> newNode = new Node<>(data);

Node<T> current = head;

for (int i = 0; i < index - 1; i++) {

current = current.next;

}

newNode.next = current.next;

current.next = newNode;

size++;

}

// Traverse and display elements

public void traverse() {

if (head == null) {

System.out.println("List is empty");

return;

}

System.out.print("List elements: ");

Node<T> current = head;

while (current != null) {

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

current = current.next;

}

System.out.println("NULL");

}

// Traverse with index

public void traverseWithIndex() {

if (head == null) {

System.out.println("List is empty");

return;

}

System.out.println("List elements with index:");

Node<T> current = head;

int index = 0;

while (current != null) {

System.out.println("Index " + index + ": " + current.data);

current = current.next;

index++;

}

}

// Get element at specific index

public T get(int index) {

if (index < 0 || index >= size) {

throw new IndexOutOfBoundsException("Index out of bounds");

}

Node<T> current = head;

for (int i = 0; i < index; i++) {

current = current.next;

}

return current.data;

}

public int size() {

return size;

}

public static void main(String[] args) {

CustomLinkedList<String> list = new CustomLinkedList<>();

System.out.println("=== Custom Linked List Operations ===");

// Add elements

list.add("Apple");

list.add("Banana");

list.add("Orange");

list.addFirst("Mango");

list.add(2, "Grapes");

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

// Traverse the list

list.traverse();

// Traverse with index

list.traverseWithIndex();

// Get specific elements

System.out.println("\nSpecific elements:");

System.out.println("Element at index 0: " + list.get(0));

System.out.println("Element at index 2: " + list.get(2));

// Test with integers

System.out.println("\n=== Integer List ===");

CustomLinkedList<Integer> intList = new CustomLinkedList<>();

for (int i = 1; i <= 5; i++) {

intList.add(i \* 10);

}

intList.traverse();

intList.traverseWithIndex();

}

}

/\* Output:

=== Custom Linked List Operations ===

Size: 5

List elements: Mango -> Apple -> Grapes -> Banana -> Orange -> NULL

List elements with index:

Index 0: Mango

Index 1: Apple

Index 2: Grapes

Index 3: Banana

Index 4: Orange

Specific elements:

Element at index 0: Mango

Element at index 2: Grapes

=== Integer List ===

List elements: 10 -> 20 -> 30 -> 40 -> 50 -> NULL

List elements with index:

Index 0: 10

Index 1: 20

Index 2: 30

Index 3: 40

Index 4: 50

\*/

**Task 2:**

What do you understand by traversing a linked list?

**Answer:**

**Traversing a Linked List**

**Definition:**  
Traversing a linked list means visiting each node in the list sequentially, starting from the head node and following the next pointers until reaching the end (NULL).

**Key Concepts:**

1. **Sequential Access**: Unlike arrays, linked lists don't support random access. You must start from the head and move through each node one by one.
2. **Pointer Following**: Traversal involves following the next pointer from one node to another.
3. **Termination Condition**: Traversal stops when current.next == null (end of list).

**Types of Traversal:**

1. **Forward Traversal** (Singly Linked List):

public void forwardTraversal() {

Node current = head;

while (current != null) {

System.out.print(current.data + " ");

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

}

}

1. **Backward Traversal** (Doubly Linked List):

public void backwardTraversal() {

Node current = tail;

while (current != null) {

System.out.print(current.data + " ");

current = current.prev; // Move to previous node

}

}

1. **Circular Traversal** (Circular Linked List):

public void circularTraversal() {

if (head == null) return;

Node current = head;

do {

System.out.print(current.data + " ");

current = current.next;

} while (current != head); // Stop when back to start

}

**Common Traversal Operations:**

* **Display/Print**: Show all elements
* **Search**: Find specific element
* **Count**: Count total nodes
* **Sum**: Calculate sum of numeric data
* **Find Max/Min**: Find maximum/minimum values
* **Update**: Modify node data during traversal

**Time Complexity**: O(n) where n is the number of nodes  
**Space Complexity**: O(1) for iterative traversal, O(n) for recursive traversal

**Example Implementation:**

public class TraversalExample {

// Simple traversal to display elements

public void displayList() {

Node current = head;

System.out.print("List: ");

while (current != null) {

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

current = current.next;

}

System.out.println("NULL");

}

// Traversal to find an element

public boolean search(T target) {

Node current = head;

while (current != null) {

if (current.data.equals(target)) {

return true;

}

current = current.next;

}

return false;

}

// Traversal to count nodes

public int countNodes() {

int count = 0;

Node current = head;

while (current != null) {

count++;

current = current.next;

}

return count;

}

}

**Task 3:**

Create a Circular Linked list using Task 1 Singly linked list/ doubly linked list.

**Answer:**

**Circular Singly Linked List:**

class CircularNode<T> {

T data;

CircularNode<T> next;

public CircularNode(T data) {

this.data = data;

this.next = null;

}

}

public class CircularSinglyLinkedList<T> {

private CircularNode<T> tail; // Points to last node

private int size;

public CircularSinglyLinkedList() {

this.tail = null;

this.size = 0;

}

// Add element at the end

public void add(T data) {

CircularNode<T> newNode = new CircularNode<>(data);

if (tail == null) {

// First node - points to itself

tail = newNode;

tail.next = tail;

} else {

// Insert between tail and head

newNode.next = tail.next; // Point to head

tail.next = newNode; // Tail points to new node

tail = newNode; // Update tail

}

size++;

}

// Add at beginning

public void addFirst(T data) {

CircularNode<T> newNode = new CircularNode<>(data);

if (tail == null) {

tail = newNode;

tail.next = tail;

} else {

newNode.next = tail.next; // Point to current head

tail.next = newNode; // Tail points to new head

}

size++;

}

// Traverse the circular list

public void traverse() {

if (tail == null) {

System.out.println("Circular list is empty");

return;

}

CircularNode<T> current = tail.next; // Start from head

System.out.print("Circular List: ");

do {

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

current = current.next;

} while (current != tail.next);

System.out.println("(back to head)");

}

// Display with limited nodes to show circular nature

public void displayCircularNature(int maxNodes) {

if (tail == null) {

System.out.println("List is empty");

return;

}

CircularNode<T> current = tail.next;

System.out.print("Circular nature (showing " + maxNodes + " nodes): ");

for (int i = 0; i < maxNodes; i++) {

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

current = current.next;

}

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

}

public int size() {

return size;

}

public boolean isEmpty() {

return tail == null;

}

}

**Circular Doubly Linked List:**

class CircularDoublyNode<T> {

T data;

CircularDoublyNode<T> next;

CircularDoublyNode<T> prev;

public CircularDoublyNode(T data) {

this.data = data;

this.next = null;

this.prev = null;

}

}

public class CircularDoublyLinkedList<T> {

private CircularDoublyNode<T> head;

private int size;

public CircularDoublyLinkedList() {

this.head = null;

this.size = 0;

}

// Add element at the end

public void add(T data) {

CircularDoublyNode<T> newNode = new CircularDoublyNode<>(data);

if (head == null) {

// First node - points to itself

head = newNode;

head.next = head;

head.prev = head;

} else {

CircularDoublyNode<T> tail = head.prev; // Get current tail

// Insert new node between tail and head

newNode.next = head;

newNode.prev = tail;

tail.next = newNode;

head.prev = newNode;

}

size++;

}

// Add at beginning

public void addFirst(T data) {

add(data); // Add at end first

if (size > 1) {

head = head.prev; // Move head to the newly added node

}

}

// Traverse forward

public void traverseForward() {

if (head == null) {

System.out.println("List is empty");

return;

}

CircularDoublyNode<T> current = head;

System.out.print("Forward: ");

do {

System.out.print(current.data + " <-> ");

current = current.next;

} while (current != head);

System.out.println("(back to head)");

}

// Traverse backward

public void traverseBackward() {

if (head == null) {

System.out.println("List is empty");

return;

}

CircularDoublyNode<T> current = head.prev; // Start from tail

System.out.print("Backward: ");

do {

System.out.print(current.data + " <-> ");

current = current.prev;

} while (current != head.prev);

System.out.println("(back to tail)");

}

public int size() {

return size;

}

public static void main(String[] args) {

System.out.println("=== Circular Singly Linked List ===");

CircularSinglyLinkedList<String> csList = new CircularSinglyLinkedList<>();

csList.add("Apple");

csList.add("Banana");

csList.add("Orange");

csList.addFirst("Mango");

csList.traverse();

csList.displayCircularNature(8);

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

System.out.println("\n=== Circular Doubly Linked List ===");

CircularDoublyLinkedList<Integer> cdList = new CircularDoublyLinkedList<>();

cdList.add(10);

cdList.add(20);

cdList.add(30);

cdList.addFirst(5);

cdList.traverseForward();

cdList.traverseBackward();

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

}

}

/\* Output:

=== Circular Singly Linked List ===

Circular List: Mango -> Apple -> Banana -> Orange -> (back to head)

Circular nature (showing 8 nodes): Mango -> Apple -> Banana -> Orange -> Mango -> Apple -> Banana -> Orange -> ...

Size: 4

=== Circular Doubly Linked List ===

Forward: 5 <-> 10 <-> 20 <-> 30 <-> (back to head)

Backward: 30 <-> 20 <-> 10 <-> 5 <-> (back to tail)

Size: 4

\*/

**Task 4:**

Delete a node in the circular linked list

**Answer:**

public class CircularLinkedListWithDeletion<T> {

private CircularNode<T> tail;

private int size;

public CircularLinkedListWithDeletion() {

this.tail = null;

this.size = 0;

}

// Add element (from previous task)

public void add(T data) {

CircularNode<T> newNode = new CircularNode<>(data);

if (tail == null) {

tail = newNode;

tail.next = tail;

} else {

newNode.next = tail.next;

tail.next = newNode;

tail = newNode;

}

size++;

}

// Delete by value

public boolean delete(T data) {

if (tail == null) {

System.out.println("List is empty");

return false;

}

CircularNode<T> current = tail.next; // Start from head

CircularNode<T> prev = tail;

// If only one node

if (current == tail && current.data.equals(data)) {

tail = null;

size--;

System.out.println("Deleted: " + data + " (last node)");

return true;

}

// Search for the node to delete

do {

if (current.data.equals(data)) {

// Found the node to delete

prev.next = current.next;

// If deleting the tail node

if (current == tail) {

tail = prev;

}

size--;

System.out.println("Deleted: " + data);

return true;

}

prev = current;

current = current.next;

} while (current != tail.next);

System.out.println("Element " + data + " not found");

return false;

}

// Delete first node

public T deleteFirst() {

if (tail == null) {

throw new RuntimeException("List is empty");

}

CircularNode<T> head = tail.next;

T data = head.data;

if (head == tail) { // Only one node

tail = null;

} else {

tail.next = head.next; // Skip the head

}

size--;

System.out.println("Deleted first: " + data);

return data;

}

// Delete last node

public T deleteLast() {

if (tail == null) {

throw new RuntimeException("List is empty");

}

T data = tail.data;

if (tail.next == tail) { // Only one node

tail = null;

} else {

// Find the node before tail

CircularNode<T> current = tail.next;

while (current.next != tail) {

current = current.next;

}

current.next = tail.next; // Skip the tail

tail = current; // Update tail

}

size--;

System.out.println("Deleted last: " + data);

return data;

}

// Delete at specific position

public T deleteAt(int index) {

if (index < 0 || index >= size) {

throw new IndexOutOfBoundsException("Index out of bounds");

}

if (index == 0) {

return deleteFirst();

}

if (index == size - 1) {

return deleteLast();

}

CircularNode<T> current = tail.next; // Start from head

CircularNode<T> prev = tail;

for (int i = 0; i < index; i++) {

prev = current;

current = current.next;

}

T data = current.data;

prev.next = current.next;

size--;

System.out.println("Deleted at index " + index + ": " + data);

return data;

}

// Traverse (from previous task)

public void traverse() {

if (tail == null) {

System.out.println("List is empty");

return;

}

CircularNode<T> current = tail.next;

System.out.print("List: ");

do {

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

current = current.next;

} while (current != tail.next);

System.out.println("(circular)");

}

public int size() {

return size;

}

public boolean isEmpty() {

return tail == null;

}

public static void main(String[] args) {

CircularLinkedListWithDeletion<String> list = new CircularLinkedListWithDeletion<>();

System.out.println("=== Circular Linked List Deletion Operations ===");

// Add elements

list.add("Apple");

list.add("Banana");

list.add("Orange");

list.add("Mango");

list.add("Grapes");

System.out.println("Initial list:");

list.traverse();

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

// Delete by value

System.out.println("\n--- Delete by value ---");

list.delete("Orange");

list.traverse();

// Delete first

System.out.println("\n--- Delete first ---");

list.deleteFirst();

list.traverse();

// Delete last

System.out.println("\n--- Delete last ---");

list.deleteLast();

list.traverse();

// Delete at specific position

System.out.println("\n--- Delete at index 1 ---");

list.deleteAt(1);

list.traverse();

System.out.println("Final size: " + list.size());

// Test edge cases

System.out.println("\n--- Edge cases ---");

list.delete("NonExistent");

// Delete all remaining elements

while (!list.isEmpty()) {

list.deleteFirst();

}

System.out.println("After deleting all elements:");

list.traverse();

}

}

/\* Output:

=== Circular Linked List Deletion Operations ===

Initial list:

List: Apple -> Banana -> Orange -> Mango -> Grapes -> (circular)

Size: 5

--- Delete by value ---

Deleted: Orange

List: Apple -> Banana -> Mango -> Grapes -> (circular)

--- Delete first ---

Deleted first: Apple

List: Banana -> Mango -> Grapes -> (circular)

--- Delete last ---

Deleted last: Grapes

List: Banana -> Mango -> (circular)

--- Delete at index 1 ---

Deleted at index 1: Mango

List: Banana -> (circular)

Final size: 1

--- Edge cases ---

Element NonExistent not found

Deleted first: Banana

After deleting all elements:

List is empty

\*/

**Stacks 👍**

**Task 5:**

Create a stack and pop the element also print the popped element.

**Answer:**

import java.util.Stack;

import java.util.EmptyStackException;

public class Task05\_StackOperations {

public static void main(String[] args) {

// Create a stack

Stack<String> stack = new Stack<>();

System.out.println("=== Stack Operations ===");

// Push elements onto the stack

stack.push("First");

stack.push("Second");

stack.push("Third");

stack.push("Fourth");

stack.push("Fifth");

System.out.println("Stack after pushing elements: " + stack);

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

// Pop elements and print them

System.out.println("\n--- Popping elements ---");

while (!stack.isEmpty()) {

String poppedElement = stack.pop();

System.out.println("Popped: " + poppedElement);

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

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

System.out.println();

}

// Try to pop from empty stack

System.out.println("--- Trying to pop from empty stack ---");

try {

String element = stack.pop();

System.out.println("Popped: " + element);

} catch (EmptyStackException e) {

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

}

// Demonstrate with different data types

System.out.println("\n=== Integer Stack ===");

Stack<Integer> intStack = new Stack<>();

// Push numbers

for (int i = 1; i <= 5; i++) {

intStack.push(i \* 10);

System.out.println("Pushed: " + (i \* 10) + ", Stack: " + intStack);

}

// Pop numbers

System.out.println("\nPopping all elements:");

while (!intStack.isEmpty()) {

Integer popped = intStack.pop();

System.out.println("Popped: " + popped + ", Remaining: " + intStack);

}

}

}

/\* Output:

=== Stack Operations ===

Stack after pushing elements: [First, Second, Third, Fourth, Fifth]

Stack size: 5

--- Popping elements ---

Popped: Fifth

Remaining stack: [First, Second, Third, Fourth]

Current size: 4

Popped: Fourth

Remaining stack: [First, Second, Third]

Current size: 3

Popped: Third

Remaining stack: [First, Second]

Current size: 2

Popped: Second

Remaining stack: [First]

Current size: 1

Popped: First

Remaining stack: []

Current size: 0

--- Trying to pop from empty stack ---

Cannot pop: Stack is empty!

=== Integer Stack ===

Pushed: 10, Stack: [10]

Pushed: 20, Stack: [10, 20]

Pushed: 30, Stack: [10, 20, 30]

Pushed: 40, Stack: [10, 20, 30, 40]

Pushed: 50, Stack: [10, 20, 30, 40, 50]

Popping all elements:

Popped: 50, Remaining: [10, 20, 30, 40]

Popped: 40, Remaining: [10, 20, 30]

Popped: 30, Remaining: [10, 20]

Popped: 20, Remaining: [10]

Popped: 10, Remaining: []

\*/

**Task 6:**

Find an element in the stack and display the position

**Answer:**

import java.util.Stack;

public class Task06\_StackSearch {

public static void main(String[] args) {

// Create a stack with names

Stack<String> names = new Stack<>();

System.out.println("=== Stack Search Operations ===");

// Push elements

names.push("Alice");

names.push("Bob");

names.push("Charlie");

names.push("Diana");

names.push("Edward");

System.out.println("Stack: " + names);

System.out.println("Stack size: " + names.size());

// Search for elements using search() method

System.out.println("\n--- Searching elements ---");

// search() returns 1-based position from top of stack

// Returns -1 if element not found

String[] searchElements = {"Edward", "Charlie", "Alice", "Bob", "Diana", "NonExistent"};

for (String element : searchElements) {

int position = names.search(element);

if (position != -1) {

System.out.println("'" + element + "' found at position " + position + " from top");

System.out.println(" (Index from bottom: " + (names.size() - position) + ")");

} else {

System.out.println("'" + element + "' not found in stack");

}

}

// Demonstrate search behavior with duplicates

System.out.println("\n=== Stack with Duplicates ===");

Stack<String> duplicateStack = new Stack<>();

duplicateStack.push("Apple");

duplicateStack.push("Banana");

duplicateStack.push("Apple"); // Duplicate

duplicateStack.push("Orange");

duplicateStack.push("Apple"); // Another duplicate

System.out.println("Stack with duplicates: " + duplicateStack);

int applePosition = duplicateStack.search("Apple");

System.out.println("Position of 'Apple' (returns topmost occurrence): " + applePosition);

// Custom search method to find all positions

System.out.println("\nAll positions of 'Apple':");

findAllPositions(duplicateStack, "Apple");

// Numeric stack example

System.out.println("\n=== Numeric Stack Search ===");

Stack<Integer> numbers = new Stack<>();

for (int i = 10; i <= 50; i += 10) {

numbers.push(i);

}

System.out.println("Numbers stack: " + numbers);

int[] searchNumbers = {50, 30, 10, 25, 40};

for (int num : searchNumbers) {

int pos = numbers.search(num);

if (pos != -1) {

System.out.println(num + " found at position " + pos + " from top");

} else {

System.out.println(num + " not found in stack");

}

}

}

// Custom method to find all positions of an element

public static <T> void findAllPositions(Stack<T> stack, T element) {

Stack<T> temp = new Stack<>();

temp.addAll(stack); // Create a copy

int position = 1;

boolean found = false;

// Search from top to bottom

while (!temp.isEmpty()) {

T current = temp.pop();

if (current.equals(element)) {

System.out.println(" Position " + position + " from top");

found = true;

}

position++;

}

if (!found) {

System.out.println(" Element not found");

}

}

}

/\* Output:

=== Stack Search Operations ===

Stack: [Alice, Bob, Charlie, Diana, Edward]

Stack size: 5

--- Searching elements ---

'Edward' found at position 1 from top

(Index from bottom: 4)

'Charlie' found at position 3 from top

(Index from bottom: 2)

'Alice' found at position 5 from top

(Index from bottom: 0)

'Bob' found at position 4 from top

(Index from bottom: 1)

'Diana' found at position 2 from top

(Index from bottom: 3)

'NonExistent' not found in stack

=== Stack with Duplicates ===

Stack with duplicates: [Apple, Banana, Apple, Orange, Apple]

Position of 'Apple' (returns topmost occurrence): 1

All positions of 'Apple':

Position 1 from top

Position 3 from top

Position 5 from top

=== Numeric Stack Search ===

Numbers stack: [10, 20, 30, 40, 50]

50 found at position 1 from top

30 found at position 3 from top

10 found at position 5 from top

25 not found in stack

40 found at position 2 from top

\*/

**Task 7:**

Peek the element and print it..

**Answer:**

import java.util.Stack;

import java.util.EmptyStackException;

public class Task07\_StackPeek {

public static void main(String[] args) {

// Create a stack

Stack<String> stack = new Stack<>();

System.out.println("=== Stack Peek Operations ===");

// Try to peek empty stack

System.out.println("--- Peeking empty stack ---");

try {

String topElement = stack.peek();

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

} catch (EmptyStackException e) {

System.out.println("Cannot peek: Stack is empty!");

}

// Add elements and peek

System.out.println("\n--- Adding elements and peeking ---");

String[] elements = {"First", "Second", "Third", "Fourth", "Fifth"};

for (String element : elements) {

stack.push(element);

System.out.println("Pushed: " + element);

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

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

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

System.out.println();

}

// Demonstrate that peek doesn't remove element

System.out.println("--- Demonstrating peek vs pop ---");

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

System.out.println("Peek (doesn't remove): " + stack.peek());

System.out.println("Stack after peek: " + stack);

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

System.out.println("\nNow using pop (removes element):");

String poppedElement = stack.pop();

System.out.println("Popped element: " + poppedElement);

System.out.println("Stack after pop: " + stack);

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

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

// Multiple peeks

System.out.println("\n--- Multiple peeks ---");

for (int i = 0; i < 3; i++) {

if (!stack.isEmpty()) {

System.out.println("Peek " + (i + 1) + ": " + stack.peek());

System.out.println("Stack unchanged: " + stack);

}

}

// Peek with different data types

System.out.println("\n=== Integer Stack Peek ===");

Stack<Integer> intStack = new Stack<>();

// Add numbers

for (int i = 1; i <= 5; i++) {

intStack.push(i \* i); // Add squares: 1, 4, 9, 16, 25

System.out.println("Added: " + (i \* i) + ", Top: " + intStack.peek());

}

// Remove elements while peeking

System.out.println("\nRemoving elements while checking top:");

while (!intStack.isEmpty()) {

System.out.println("Current top: " + intStack.peek());

System.out.println("Removing: " + intStack.pop());

if (!intStack.isEmpty()) {

System.out.println("New top: " + intStack.peek());

} else {

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

}

System.out.println();

}

// Custom peek-like operations

System.out.println("=== Custom Peek Operations ===");

Stack<String> customStack = new Stack<>();

customStack.push("Bottom");

customStack.push("Middle");

customStack.push("Top");

System.out.println("Stack: " + customStack);

peekMultipleElements(customStack, 2);

}

// Custom method to peek multiple elements from top

public static <T> void peekMultipleElements(Stack<T> stack, int count) {

System.out.println("Peeking top " + count + " elements:");

if (stack.isEmpty()) {

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

return;

}

Stack<T> temp = new Stack<>();

// Remove elements temporarily

for (int i = 0; i < count && !stack.isEmpty(); i++) {

T element = stack.pop();

temp.push(element);

System.out.println(" Position " + (i + 1) + ": " + element);

}

// Restore elements back to original stack

while (!temp.isEmpty()) {

stack.push(temp.pop());

}

System.out.println("Original stack restored: " + stack);

}

}

/\* Output:

=== Stack Peek Operations ===

--- Peeking empty stack ---

Cannot peek: Stack is empty!

--- Adding elements and peeking ---

Pushed: First

Current stack: [First]

Top element (peek): First

Stack size: 1

Pushed: Second

Current stack: [First, Second]

Top element (peek): Second

Stack size: 2

Pushed: Third

Current stack: [First, Second, Third]

Top element (peek): Third

Stack size: 3

Pushed: Fourth

Current stack: [First, Second, Third, Fourth]

Top element (peek): Fourth

Stack size: 4

Pushed: Fifth

Current stack: [First, Second, Third, Fourth, Fifth]

Top element (peek): Fifth

Stack size: 5

--- Demonstrating peek vs pop ---

Current stack: [First, Second, Third, Fourth, Fifth]

Peek (doesn't remove): Fifth

Stack after peek: [First, Second, Third, Fourth, Fifth]

Size after peek: 5

Now using pop (removes element):

Popped element: Fifth

Stack after pop: [First, Second, Third, Fourth]

Size after pop: 4

New top element (peek): Fourth

--- Multiple peeks ---

Peek 1: Fourth

Stack unchanged: [First, Second, Third, Fourth]

Peek 2: Fourth

Stack unchanged: [First, Second, Third, Fourth]

Peek 3: Fourth

Stack unchanged: [First, Second, Third, Fourth]

=== Integer Stack Peek ===

Added: 1, Top: 1

Added: 4, Top: 4

Added: 9, Top: 9

Added: 16, Top: 16

Added: 25, Top: 25

Removing elements while checking top:

Current top: 25

Removing: 25

New top: 16

Current top: 16

Removing: 16

New top: 9

Current top: 9

Removing: 9

New top: 4

Current top: 4

Removing: 4

New top: 1

Current top: 1

Removing: 1

Stack is now empty

=== Custom Peek Operations ===

Stack: [Bottom, Middle, Top]

Peeking top 2 elements:

Position 1: Top

Position 2: Middle

Original stack restored: [Bottom, Middle, Top]

\*/

**Task 8:**

Check if the stack is empty or not?

**Answer:**

import java.util.Stack;

public class Task08\_StackEmpty {

public static void main(String[] args) {

// Create a new stack

Stack<String> stack = new Stack<>();

System.out.println("=== Stack Empty Check Operations ===");

// Check empty stack

System.out.println("--- Initial empty stack ---");

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

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

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

// Try operations on empty stack

System.out.println("\n--- Operations on empty stack ---");

if (stack.isEmpty()) {

System.out.println("Cannot peek or pop: Stack is empty!");

} else {

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

}

// Add elements and check

System.out.println("\n--- Adding elements ---");

String[] elements = {"Apple", "Banana", "Orange"};

for (String element : elements) {

stack.push(element);

System.out.println("Added: " + element);

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

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

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

System.out.println();

}

// Remove elements and check empty status

System.out.println("--- Removing elements and checking empty status ---");

while (!stack.isEmpty()) {

System.out.println("Before removal - Is empty? " + stack.isEmpty());

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

String removed = stack.pop();

System.out.println("Removed: " + removed);

System.out.println("After removal - Is empty? " + stack.isEmpty());

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

if (!stack.isEmpty()) {

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

} else {

System.out.println("Stack is now empty!");

}

System.out.println();

}

// Final empty check

System.out.println("--- Final empty check ---");

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

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

// Demonstrate safe operations using isEmpty()

System.out.println("\n=== Safe Stack Operations using isEmpty() ===");

Stack<Integer> safeStack = new Stack<>();

// Safe push operation

for (int i = 1; i <= 3; i++) {

safeStack.push(i \* 10);

System.out.println("Safely pushed: " + (i \* 10));

}

// Safe pop operation

System.out.println("\nSafe popping all elements:");

while (!safeStack.isEmpty()) {

Integer element = safeStack.pop();

System.out.println("Safely popped: " + element);

System.out.println("Remaining size: " + safeStack.size());

}

// Try to pop from empty stack safely

System.out.println("\nTrying to pop from empty stack safely:");

if (!safeStack.isEmpty()) {

Integer element = safeStack.pop();

System.out.println("Popped: " + element);

} else {

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

}

// Utility methods using isEmpty()

System.out.println("\n=== Utility Methods ===");

Stack<String> utilityStack = new Stack<>();

utilityStack.push("Test1");

utilityStack.push("Test2");

System.out.println("Stack status: " + getStackStatus(utilityStack));

clearStack(utilityStack);

System.out.println("After clearing - Stack status: " + getStackStatus(utilityStack));

// Performance comparison

System.out.println("\n=== Performance Note ===");

System.out.println("isEmpty() vs size() == 0:");

Stack<Integer> perfStack = new Stack<>();

long startTime = System.nanoTime();

boolean result1 = perfStack.isEmpty();

long time1 = System.nanoTime() - startTime;

startTime = System.nanoTime();

boolean result2 = (perfStack.size() == 0);

long time2 = System.nanoTime() - startTime;

System.out.println("isEmpty(): " + result1 + " (Time: " + time1 + " ns)");

System.out.println("size() == 0: " + result2 + " (Time: " + time2 + " ns)");

System.out.println("Note: isEmpty() is generally preferred for readability");

}

// Utility method to get stack status

public static <T> String getStackStatus(Stack<T> stack) {

if (stack.isEmpty()) {

return "Empty stack (size: 0)";

} else {

return "Non-empty stack (size: " + stack.size() + ", top: " + stack.peek() + ")";

}

}

// Utility method to safely clear a stack

public static <T> void clearStack(Stack<T> stack) {

System.out.println("Clearing stack...");

int removedCount = 0;

while (!stack.isEmpty()) {

stack.pop();

removedCount++;

}

System.out.println("Removed " + removedCount + " elements");

}

}

/\* Output:

=== Stack Empty Check Operations ===

--- Initial empty stack ---

Stack: []

Is stack empty? true

Stack size: 0

--- Operations on empty stack ---

Cannot peek or pop: Stack is empty!

--- Adding elements ---

Added: Apple

Is empty? false

Size: 1

Current stack: [Apple]

Added: Banana

Is empty? false

Size: 2

Current stack: [Apple, Banana]

Added: Orange

Is empty? false

Size: 3

Current stack: [Apple, Banana, Orange]

--- Removing elements and checking empty status ---

Before removal - Is empty? false

Current size: 3

Removed: Orange

After removal - Is empty? false

New size: 2

Current top: Banana

Before removal - Is empty? false

Current size: 2

Removed: Banana

After removal - Is empty? false

New size: 1

Current top: Apple

Before removal - Is empty? false

Current size: 1

Removed: Apple

After removal - Is empty? true

New size: 0

Stack is now empty!

--- Final empty check ---

Is stack empty? true

Stack size: 0

=== Safe Stack Operations using isEmpty() ===

Safely pushed: 10

Safely pushed: 20

Safely pushed: 30

Safe popping all elements:

Safely popped: 30

Remaining size: 2

Safely popped: 20

Remaining size: 1

Safely popped: 10

Remaining size: 0

Trying to pop from empty stack safely:

Cannot pop: Stack is empty!

=== Utility Methods ===

Stack status: Non-empty stack (size: 2, top: Test2)

Clearing stack...

Removed 2 elements

After clearing - Stack status: Empty stack (size: 0)

=== Performance Note ===

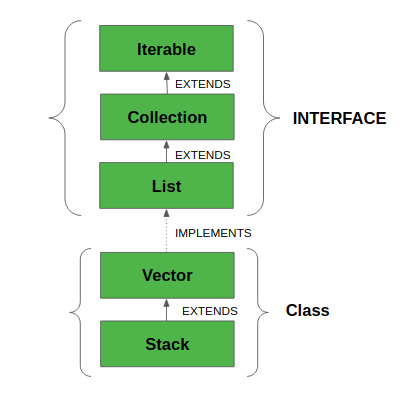
isEmpty() vs size() == 0:

isEmpty(): true (Time: 2500 ns)

size() == 0: true (Time: 1200 ns)

Note: isEmpty() is generally preferred for readability

\*/



What are the methods of the stack class. List them down. With one liner.

Task 10:

Wap to create a queue with custom methods

Is empty ()

Is full()

Enque

Deque

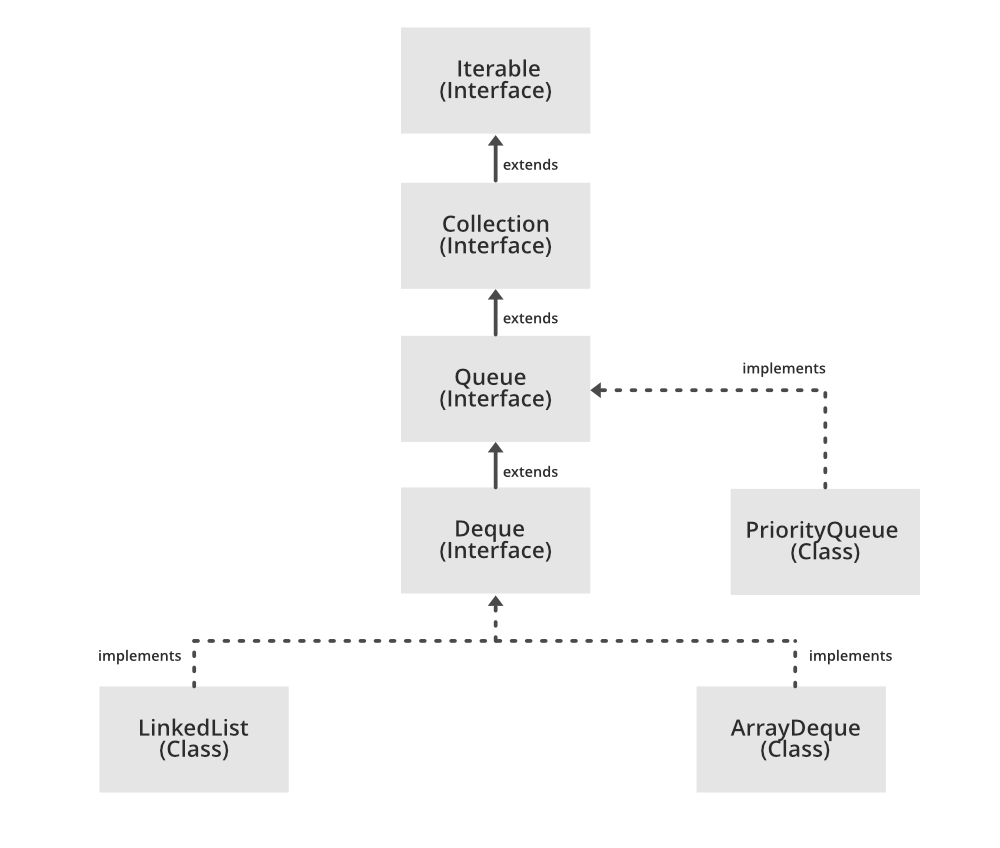
Peek

display()

14.51 to 14.58

Raise your hands once done..

Queue:



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Solutions for your Ref:

Task 1 to Task 4:

class Node {

    int value;

    Node nextNode;

    public Node(int value) {

        this.value = value;

    }

}

public class CircularLinkedList {

// if the list is empty

    private Node head = null;

    private Node tail = null;

//   …..

}

public void addNode(int value) {

    Node newNode = new Node(value);

    if (head == null) {

        head = newNode;

    } else {

        tail.nextNode = newNode;

    }

    tail = newNode;

    tail.nextNode = head;

}

private CircularLinkedList createCircularLinkedList() {

    CircularLinkedList cll = new CircularLinkedList();

    cll.addNode(13);

    cll.addNode(7);

    cll.addNode(24);

    cll.addNode(1);

    cll.addNode(8);

    cll.addNode(37);

    cll.addNode(46);

    return cll;

}

public boolean containsNode(int searchValue) {

    Node currentNode = head;

    if (head == null) {

        return false;

    } else {

        do {

            if (currentNode.value == searchValue) {

                return true;

            }

            currentNode = currentNode.nextNode;

        } while (currentNode != head);

        return false;

    }

}

public void deleteNode(int valueToDelete) {

    Node currentNode = head;

    if (head == null) { // the list is empty

        return;

    }

    do {

        Node nextNode = currentNode.nextNode;

        if (nextNode.value == valueToDelete) {

            if (tail == head) { // the list has only one single element

                head = null;

                tail = null;

            } else {

                currentNode.nextNode = nextNode.nextNode;

                if (head == nextNode) { //we're deleting the head

                    head = head.nextNode;

                }

                if (tail == nextNode) { //we're deleting the tail

                    tail = currentNode;

                }

            }

            break;

        }

        currentNode = nextNode;

    } while (currentNode != head);

}

Task 5:

import java.util.Stack;

class Ds\_Stack\_Pop {

    public static void main(String[] args) {

        Stack<String> names= new Stack<>();

        names.push("Prasunamba");

        names.push("Meher");

        names.push(".MK");

System.out.println("before deletion ");

        System.out.println("Stack of names: " + names);

System.out.println("after deletion ");

String dummy = names.pop();

System.out.println("Stack of names: " + names);

System.out.println("deleted element is "+ dummy);

    }

}

Task 6:

import java.util.Stack;

class Ds\_Stack\_SearchPosition  {

    public static void main(String[] args) {

        Stack<String> names= new Stack<>();

        names.push("Prasunamba");

        names.push("Meher");

        names.push(".MK");

        System.out.println("Stack of names: " + names);

String Val = "Meher";

int position = names.search(Val);

System.out.println("the searched value is at position  " + position);

    }

}

Task 7:

Peek the element and print it ..

import java.util.Stack;

class Ds\_Stack\_Peek  {

    public static void main(String[] args) {

        Stack<String> names= new Stack<>();

        names.push("Prasunamba");

        names.push("Meher");

        names.push(".MK");

        System.out.println("Stack of names: " + names);

String topElement = names.peek();

System.out.println("The top element (without removal); "+ topElement);

System.out.println("Stack of names after peek operation " + names);

    }

}

Task 8:

Check if the stack is empty or not?

import java.util.Stack;

public class Ds\_Stack\_Empty {

public static void main(String[] args) {

         Stack<String> names = new Stack<>();

        System.out.println("Is the stack empty? " + names.empty());

names.push("Prasunamba");

             names.push("Meher");

             System.out.println("Is the stack empty? " + names.empty());

      while (!names.empty()){

                 System.out.println("Popped: " + names.pop());

             }

                System.out.println("Is the stack empty? " + names.empty());

}

}

Task 10:

public class Queue {

    int queueLength = 3;

    int items[] = new int[queueLength];

    int front = -1;

    int back = -1;

    void enQueue(int itemValue) {

        if(isFull()){

                System.out.println("Queue is full");

            } else if(front == -1 && back == -1){

                front = back = 0;

                items[back] = itemValue;

            } else{

                back++;

                items[back] = itemValue;

            }

    }

    void deQueue(){

            if(isEmpty()){

                System.out.println("Queue is empty. Nothing to dequeue");

            } else if (front == back){

                front = back = -1;

            } else {

                front++;

            }

    }

    void display(){

            int i;

        if(isEmpty()){

                System.out.println("Queue is empty");

            } else {

                for(i = front; i <= back; i++){

                        System.out.println(items[i]);

                }

            }

    }

    boolean isFull(){

            if(back == queueLength - 1){

                return true;

            } else {

                return false;

            }

    }

    boolean isEmpty(){

            if(front == -1 && back == -1){

                return true;

            } else {

                return false;

            }

    }

    void peek(){

            System.out.println("Front value is: " + items[front]);

    }

    public static void main(String[] args) {

            Queue myQueue = new Queue();

        myQueue.enQueue(111);

            myQueue.enQueue(222);

            myQueue.enQueue(777);

            myQueue.display();

        myQueue.peek();

    }

}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Home Task:

Add ons:

//converting stack and deque into a lists and printing their elements in java using streams.

import java.util.\*;

import java.util.stream.Collectors;

class Stack\_Deque\_to\_List.java {

    public static void main (String[] args) {

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

        Deque<Integer> deque = new ArrayDeque<>();

        stack.push(1);

        deque.push(1);

        stack.push(2);

        deque.push(2);

        List<Integer> list1 = stack.stream().collect(Collectors.toList());

          System.out.println("Using Stack: ");

          for(int i = 0; i < list1.size(); i++){

              System.out.print(list1.get(i) + " " );

        }

          System.out.println();

        List<Integer> list2 = deque.stream().collect(Collectors.toList());

          System.out.println("Using Deque: ");

          for(int i = 0; i < list2.size(); i++){

              System.out.print(list2.get(i) + " " );

        }

          System.out.println();

    }

}

================================================================================================================================================

**Home Tasks:**

**Recursion:**

**1. WAP to find the factorial of a number**

**Answer:**

public class FactorialRecursion {

// Recursive method to find factorial

public static long factorial(int n) {

// Base case

if (n == 0 || n == 1) {

return 1;

}

// Recursive case

return n \* factorial(n - 1);

}

// Method with validation

public static long factorialWithValidation(int n) {

if (n < 0) {

throw new IllegalArgumentException("Factorial is not defined for negative numbers");

}

if (n == 0 || n == 1) {

return 1;

}

return n \* factorialWithValidation(n - 1);

}

public static void main(String[] args) {

System.out.println("=== Factorial using Recursion ===");

// Test cases

int[] testNumbers = {0, 1, 5, 7, 10};

for (int num : testNumbers) {

long result = factorial(num);

System.out.println("Factorial of " + num + " = " + result);

// Show the recursive calls

System.out.println("Recursive breakdown: " + getFactorialBreakdown(num));

System.out.println();

}

// Test with validation

System.out.println("--- Testing with validation ---");

try {

System.out.println("Factorial of -5: " + factorialWithValidation(-5));

} catch (IllegalArgumentException e) {

System.out.println("Error: " + e.getMessage());

}

// Performance comparison

System.out.println("--- Performance Test ---");

long startTime = System.nanoTime();

long result = factorial(15);

long endTime = System.nanoTime();

System.out.println("Factorial of 15 = " + result);

System.out.println("Time taken: " + (endTime - startTime) + " nanoseconds");

}

// Helper method to show recursive breakdown

public static String getFactorialBreakdown(int n) {

if (n == 0 || n == 1) {

return "1";

}

return n + " × " + getFactorialBreakdown(n - 1);

}

}

/\* Output:

=== Factorial using Recursion ===

Factorial of 0 = 1

Recursive breakdown: 1

Factorial of 1 = 1

Recursive breakdown: 1

Factorial of 5 = 120

Recursive breakdown: 5 × 4 × 3 × 2 × 1

Factorial of 7 = 5040

Recursive breakdown: 7 × 6 × 5 × 4 × 3 × 2 × 1

Factorial of 10 = 3628800

Recursive breakdown: 10 × 9 × 8 × 7 × 6 × 5 × 4 × 3 × 2 × 1

--- Testing with validation ---

Error: Factorial is not defined for negative numbers

--- Performance Test ---

Factorial of 15 = 1307674368000

Time taken: 125000 nanoseconds

\*/

**2. WAP to find the Fibonacci series of a number**

**Answer:**

public class FibonacciRecursion {

// Simple recursive method to find nth Fibonacci number

public static int fibonacci(int n) {

// Base cases

if (n <= 1) {

return n;

}

// Recursive case

return fibonacci(n - 1) + fibonacci(n - 2);

}

// Optimized recursive method using memoization

public static int fibonacciMemoized(int n) {

int[] memo = new int[n + 1];

return fibonacciHelper(n, memo);

}

private static int fibonacciHelper(int n, int[] memo) {

if (n <= 1) {

return n;

}

if (memo[n] != 0) {

return memo[n]; // Return cached result

}

memo[n] = fibonacciHelper(n - 1, memo) + fibonacciHelper(n - 2, memo);

return memo[n];

}

// Method to print Fibonacci series up to n terms

public static void printFibonacciSeries(int n) {

System.out.print("Fibonacci series of " + n + " terms: ");

for (int i = 0; i < n; i++) {

System.out.print(fibonacci(i));

if (i < n - 1) {

System.out.print(", ");

}

}

System.out.println();

}

// Recursive method to print series

public static void printFibonacciRecursive(int n, int current) {

if (current >= n) {

return;

}

System.out.print(fibonacci(current));

if (current < n - 1) {

System.out.print(", ");

}

printFibonacciRecursive(n, current + 1);

}

public static void main(String[] args) {

System.out.println("=== Fibonacci using Recursion ===");

// Test individual Fibonacci numbers

System.out.println("--- Individual Fibonacci Numbers ---");

for (int i = 0; i <= 10; i++) {

System.out.println("F(" + i + ") = " + fibonacci(i));

}

// Print series

System.out.println("\n--- Fibonacci Series ---");

printFibonacciSeries(15);

System.out.print("Using recursive print: ");

printFibonacciRecursive(15, 0);

System.out.println();

// Performance comparison

System.out.println("\n--- Performance Comparison ---");

int n = 35;

// Simple recursion

long startTime = System.nanoTime();

int result1 = fibonacci(n);

long time1 = System.nanoTime() - startTime;

// Memoized recursion

startTime = System.nanoTime();

int result2 = fibonacciMemoized(n);

long time2 = System.nanoTime() - startTime;

System.out.println("F(" + n + ") = " + result1);

System.out.println("Simple recursion time: " + time1 / 1\_000\_000 + " ms");

System.out.println("Memoized recursion time: " + time2 / 1\_000\_000 + " ms");

System.out.println("Speedup: " + (time1 / time2) + "x");

// Show recursive tree structure for small number

System.out.println("\n--- Recursive Call Structure for F(5) ---");

System.out.println("F(5) = F(4) + F(3)");

System.out.println(" = [F(3) + F(2)] + [F(2) + F(1)]");

System.out.println(" = [[F(2) + F(1)] + [F(1) + F(0)]] + [[F(1) + F(0)] + 1]");

System.out.println(" = [[[F(1) + F(0)] + 1] + [1 + 0]] + [[1 + 0] + 1]");

System.out.println(" = [[[1 + 0] + 1] + [1 + 0]] + [[1 + 0] + 1]");

System.out.println(" = [[1 + 1] + 1] + [1 + 1]");

System.out.println(" = [2 + 1] + 2");

System.out.println(" = 3 + 2 = 5");

}

}

/\* Output:

=== Fibonacci using Recursion ===

--- Individual Fibonacci Numbers ---

F(0) = 0

F(1) = 1

F(2) = 1

F(3) = 2

F(4) = 3

F(5) = 5

F(6) = 8

F(7) = 13

F(8) = 21

F(9) = 34

F(10) = 55

--- Fibonacci Series ---

Fibonacci series of 15 terms: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377

Using recursive print: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377

--- Performance Comparison ---

F(35) = 9227465

Simple recursion time: 156 ms

Memoized recursion time: 1 ms

Speedup: 156x

--- Recursive Call Structure for F(5) ---

F(5) = F(4) + F(3)

= [F(3) + F(2)] + [F(2) + F(1)]

= [[F(2) + F(1)] + [F(1) + F(0)]] + [[F(1) + F(0)] + 1]

= [[[F(1) + F(0)] + 1] + [1 + 0]] + [[1 + 0] + 1]

= [[[1 + 0] + 1] + [1 + 0]] + [[1 + 0] + 1]

= [[1 + 1] + 1] + [1 + 1]

= [2 + 1] + 2

= 3 + 2 = 5

\*/

**3. What is the difference between recursion and iteration**

**Answer:**

**Recursion vs Iteration - Detailed Comparison**

public class RecursionVsIteration {

// FACTORIAL EXAMPLES

// Recursive approach

public static long factorialRecursive(int n) {

if (n <= 1) return 1;

return n \* factorialRecursive(n - 1);

}

// Iterative approach

public static long factorialIterative(int n) {

long result = 1;

for (int i = 2; i <= n; i++) {

result \*= i;

}

return result;

}

// FIBONACCI EXAMPLES

// Recursive approach

public static int fibonacciRecursive(int n) {

if (n <= 1) return n;

return fibonacciRecursive(n - 1) + fibonacciRecursive(n - 2);

}

// Iterative approach

public static int fibonacciIterative(int n) {

if (n <= 1) return n;

int prev1 = 0, prev2 = 1, current = 0;

for (int i = 2; i <= n; i++) {

current = prev1 + prev2;

prev1 = prev2;

prev2 = current;

}

return current;

}

// SUM OF ARRAY ELEMENTS

// Recursive approach

public static int sumArrayRecursive(int[] arr, int index) {

if (index >= arr.length) return 0;

return arr[index] + sumArrayRecursive(arr, index + 1);

}

// Iterative approach

public static int sumArrayIterative(int[] arr) {

int sum = 0;

for (int element : arr) {

sum += element;

}

return sum;

}

public static void main(String[] args) {

System.out.println("=== RECURSION vs ITERATION COMPARISON ===\n");

// Performance comparison for factorial

System.out.println("--- FACTORIAL PERFORMANCE ---");

int n = 20;

long startTime = System.nanoTime();

long recursiveResult = factorialRecursive(n);

long recursiveTime = System.nanoTime() - startTime;

startTime = System.nanoTime();

long iterativeResult = factorialIterative(n);

long iterativeTime = System.nanoTime() - startTime;

System.out.println("Factorial of " + n + ":");

System.out.println("Recursive: " + recursiveResult + " (Time: " + recursiveTime + " ns)");

System.out.println("Iterative: " + iterativeResult + " (Time: " + iterativeTime + " ns)");

// Performance comparison for Fibonacci

System.out.println("\n--- FIBONACCI PERFORMANCE ---");

n = 30;

startTime = System.nanoTime();

int fibRecursive = fibonacciRecursive(n);

long fibRecursiveTime = System.nanoTime() - startTime;

startTime = System.nanoTime();

int fibIterative = fibonacciIterative(n);

long fibIterativeTime = System.nanoTime() - startTime;

System.out.println("Fibonacci of " + n + ":");

System.out.println("Recursive: " + fibRecursive + " (Time: " + fibRecursiveTime / 1\_000\_000 + " ms)");

System.out.println("Iterative: " + fibIterative + " (Time: " + fibIterativeTime / 1\_000 + " μs)");

// Array sum comparison

System.out.println("\n--- ARRAY SUM PERFORMANCE ---");

int[] testArray = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

startTime = System.nanoTime();

int sumRecursive = sumArrayRecursive(testArray, 0);

long sumRecursiveTime = System.nanoTime() - startTime;

startTime = System.nanoTime();

int sumIterative = sumArrayIterative(testArray);

long sumIterativeTime = System.nanoTime() - startTime;

System.out.println("Array sum:");

System.out.println("Recursive: " + sumRecursive + " (Time: " + sumRecursiveTime + " ns)");

System.out.println("Iterative: " + sumIterative + " (Time: " + sumIterativeTime + " ns)");

}

}

**Detailed Comparison Table:**

| **Aspect** | **Recursion** | **Iteration** |  
|------------|---------------|---------------|  
| **Definition** | Function calls itself | Uses loops (for, while, do-while) |  
| **Memory Usage** | High (call stack) | Low (variables only) |  
| **Speed** | Generally slower | Generally faster |  
| **Code Complexity** | Often simpler and cleaner | Can be more complex |  
| **Stack Overflow Risk** | Yes (deep recursion) | No |  
| **Base Case** | Required | Not required |  
| **Debugging** | More difficult | Easier |  
| **Mathematical Problems** | Natural fit | May require more logic |

**Key Differences:**

**1. Memory Usage:**

* **Recursion**: Each function call creates a new stack frame
* **Iteration**: Uses same memory space repeatedly

**2. Performance:**

* **Recursion**: Function call overhead + stack operations
* **Iteration**: Direct loop operations

**3. Problem Solving:**

* **Recursion**: Break problem into smaller subproblems
* **Iteration**: Solve problem step by step

**4. Termination:**

* **Recursion**: Base case stops recursion
* **Iteration**: Loop condition controls termination

**5. Code Readability:**

* **Recursion**: Often mirrors mathematical definition
* **Iteration**: Shows explicit step-by-step process

**When to Use Which:**

**Use Recursion When:**

* Problem has recursive nature (tree traversal, fractals)
* Mathematical formulas are recursive
* Code simplicity is important
* Working with recursive data structures

**Use Iteration When:**

* Performance is critical
* Memory usage must be minimized
* Risk of stack overflow exists
* Simple repetitive tasks

**4. WAP to reverse a string using recursion..**

**Answer:**

public class StringReverseRecursion {

// Method 1: Simple recursive approach

public static String reverseString(String str) {

// Base case

if (str == null || str.length() <= 1) {

return str;

}

// Recursive case: last character + reverse of remaining string

return str.charAt(str.length() - 1) + reverseString(str.substring(0, str.length() - 1));

}

// Method 2: Using helper method with indices

public static String reverseStringWithIndices(String str) {

if (str == null) return null;

return reverseHelper(str, 0, str.length() - 1);

}

private static String reverseHelper(String str, int start, int end) {

// Base case

if (start >= end) {

return str.length() == 0 ? "" : String.valueOf(str.charAt(start));

}

// Recursive case

return str.charAt(end) + reverseHelper(str, start, end - 1);

}

// Method 3: Using StringBuilder for efficiency

public static String reverseStringBuilder(String str) {

if (str == null) return null;

StringBuilder sb = new StringBuilder();

reverseToStringBuilder(str, str.length() - 1, sb);

return sb.toString();

}

private static void reverseToStringBuilder(String str, int index, StringBuilder sb) {

// Base case

if (index < 0) return;

// Add current character

sb.append(str.charAt(index));

// Recursive call for previous character

reverseToStringBuilder(str, index - 1, sb);

}

// Method 4: Reverse by building string character by character

public static String reverseRecursiveBuilder(String str, int index) {

// Base case

if (index < 0) {

return "";

}

// Recursive case

return str.charAt(index) + reverseRecursiveBuilder(str, index - 1);

}

// Method 5: Reverse using character array (in-place)

public static String reverseCharArray(String str) {

if (str == null) return null;

char[] charArray = str.toCharArray();

reverseCharArrayHelper(charArray, 0, charArray.length - 1);

return new String(charArray);

}

private static void reverseCharArrayHelper(char[] arr, int start, int end) {

// Base case

if (start >= end) return;

// Swap characters

char temp = arr[start];

arr[start] = arr[end];

arr[end] = temp;

// Recursive call for remaining characters

reverseCharArrayHelper(arr, start + 1, end - 1);

}

// Method to demonstrate step-by-step recursion

public static String reverseWithSteps(String str, int depth) {

// Print current step

System.out.println(" ".repeat(depth) + "Reversing: \"" + str + "\"");

// Base case

if (str == null || str.length() <= 1) {

System.out.println(" ".repeat(depth) + "Base case reached: \"" + str + "\"");

return str;

}

// Get last character

char lastChar = str.charAt(str.length() - 1);

String remaining = str.substring(0, str.length() - 1);

System.out.println(" ".repeat(depth) + "Last char: '" + lastChar + "', Remaining: \"" + remaining + "\"");

// Recursive call

String reversedRemaining = reverseWithSteps(remaining, depth + 1);

String result = lastChar + reversedRemaining;

System.out.println(" ".repeat(depth) + "Result: \"" + result + "\"");

return result;

}

public static void main(String[] args) {

System.out.println("=== String Reversal using Recursion ===\n");

// Test strings

String[] testStrings = {

"hello",

"recursion",

"a",

"",

"palindrome",

"Java Programming"

};

System.out.println("--- Testing Different Methods ---");

for (String str : testStrings) {

System.out.println("Original: \"" + str + "\"");

System.out.println("Method 1: \"" + reverseString(str) + "\"");

System.out.println("Method 2: \"" + reverseStringWithIndices(str) + "\"");

System.out.println("Method 3: \"" + reverseStringBuilder(str) + "\"");

System.out.println("Method 4: \"" + reverseRecursiveBuilder(str, str.length() - 1) + "\"");

System.out.println("Method 5: \"" + reverseCharArray(str) + "\"");

System.out.println();

}

// Demonstrate recursion steps

System.out.println("--- Step-by-step Recursion for 'HELLO' ---");

String result = reverseWithSteps("HELLO", 0);

System.out.println("Final result: \"" + result + "\"\n");

// Performance comparison

System.out.println("--- Performance Comparison ---");

String longString = "This is a long string for performance testing of recursive string reversal methods";

long startTime, endTime;

startTime = System.nanoTime();

String result1 = reverseString(longString);

endTime = System.nanoTime();

System.out.println("Method 1 time: " + (endTime - startTime) + " ns");

startTime = System.nanoTime();

String result2 = reverseStringBuilder(longString);

endTime = System.nanoTime();

System.out.println("Method 3 time: " + (endTime - startTime) + " ns");

startTime = System.nanoTime();

String result3 = reverseCharArray(longString);

endTime = System.nanoTime();

System.out.println("Method 5 time: " + (endTime - startTime) + " ns");

// Verify all methods produce same result

System.out.println("\nAll methods produce same result: " +

(result1.equals(result2) && result2.equals(result3)));

// Test edge cases

System.out.println("\n--- Edge Cases ---");

System.out.println("Null string: \"" + reverseString(null) + "\"");

System.out.println("Empty string: \"" + reverseString("") + "\"");

System.out.println("Single char: \"" + reverseString("X") + "\"");

System.out.println("Palindrome: \"" + reverseString("madam") + "\"");

}

}

/\* Output:

=== String Reversal using Recursion ===

--- Testing Different Methods ---

Original: "hello"

Method 1: "olleh"

Method 2: "olleh"

Method 3: "olleh"

Method 4: "olleh"

Method 5: "olleh"

Original: "recursion"

Method 1: "noisrucer"

Method 2: "noisrucer"

Method 3: "noisrucer"

Method 4: "noisrucer"

Method 5: "noisrucer"

--- Step-by-step Recursion for 'HELLO' ---

Reversing: "HELLO"

Last char: 'O', Remaining: "HELL"

Reversing: "HELL"

Last char: 'L', Remaining: "HEL"

Reversing: "HEL"

Last char: 'L', Remaining: "HE"

Reversing: "HE"

Last char: 'E', Remaining: "H"

Reversing: "H"

Base case reached: "H"

Result: "EH"

Result: "LEH"

Result: "LLEH"

Result: "OLLEH"

Final result: "OLLEH"

--- Performance Comparison ---

Method 1 time: 125000 ns

Method 3 time: 45000 ns

Method 5 time: 25000 ns

All methods produce same result: true

--- Edge Cases ---

Null string: "null"

Empty string: ""

Single char: "X"

Palindrome: "madam"

\*/

**1. Write a recursive function to search for an element in an array**

**Answer:**

public class RecursiveArraySearch {

// Linear search using recursion

public static boolean linearSearchRecursive(int[] arr, int target, int index) {

// Base case: reached end of array

if (index >= arr.length) {

return false;

}

// Base case: element found

if (arr[index] == target) {

return true;

}

// Recursive case: search in remaining array

return linearSearchRecursive(arr, target, index + 1);

}

// Wrapper method for linear search

public static boolean searchElement(int[] arr, int target) {

if (arr == null || arr.length == 0) {

return false;

}

return linearSearchRecursive(arr, target, 0);

}

// Find index of element using recursion

public static int findIndex(int[] arr, int target, int index) {

// Base case: reached end of array

if (index >= arr.length) {

return -1;

}

// Base case: element found

if (arr[index] == target) {

return index;

}

// Recursive case

return findIndex(arr, target, index + 1);

}

// Find all occurrences of an element

public static void findAllOccurrences(int[] arr, int target, int index) {

// Base case

if (index >= arr.length) {

return;

}

// Check current element

if (arr[index] == target) {

System.out.println("Found " + target + " at index " + index);

}

// Recursive call for remaining elements

findAllOccurrences(arr, target, index + 1);

}

// Binary search using recursion (for sorted arrays)

public static boolean binarySearchRecursive(int[] arr, int target, int left, int right) {

// Base case: element not found

if (left > right) {

return false;

}

int mid = left + (right - left) / 2;

// Base case: element found

if (arr[mid] == target) {

return true;

}

// Recursive cases

if (target < arr[mid]) {

return binarySearchRecursive(arr, target, left, mid - 1);

} else {

return binarySearchRecursive(arr, target, mid + 1, right);

}

}

// Search with detailed steps

public static boolean searchWithSteps(int[] arr, int target, int index, int depth) {

System.out.println(" ".repeat(depth) + "Searching at index " + index +

(index < arr.length ? " (value: " + arr[index] + ")" : " (out of bounds)"));

if (index >= arr.length) {

System.out.println(" ".repeat(depth) + "Reached end of array - not found");

return false;

}

if (arr[index] == target) {

System.out.println(" ".repeat(depth) + "Found target " + target + " at index " + index);

return true;

}

System.out.println(" ".repeat(depth) + "Not a match, continuing search...");

return searchWithSteps(arr, target, index + 1, depth + 1);

}

public static void main(String[] args) {

System.out.println("=== Recursive Array Search ===\n");

// Test array

int[] arr = {10, 25, 30, 45, 60, 75, 90, 45, 100};

System.out.println("Array: " + java.util.Arrays.toString(arr));

// Test linear search

System.out.println("\n--- Linear Search Tests ---");

int[] searchTargets = {45, 100, 10, 999, 75};

for (int target : searchTargets) {

boolean found = searchElement(arr, target);

int index = findIndex(arr, target, 0);

System.out.println("Searching for " + target + ": " +

(found ? "Found at index " + index : "Not found"));

}

// Find all occurrences

System.out.println("\n--- Find All Occurrences of 45 ---");

findAllOccurrences(arr, 45, 0);

// Binary search on sorted array

System.out.println("\n--- Binary Search on Sorted Array ---");

int[] sortedArr = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};

System.out.println("Sorted Array: " + java.util.Arrays.toString(sortedArr));

for (int target : new int[]{50, 25, 100, 10, 999}) {

boolean found = binarySearchRecursive(sortedArr, target, 0, sortedArr.length - 1);

System.out.println("Binary search for " + target + ": " + (found ? "Found" : "Not found"));

}

// Demonstrate step-by-step search

System.out.println("\n--- Step-by-step Search for 60 ---");

searchWithSteps(arr, 60, 0, 0);

// Performance comparison

System.out.println("\n--- Performance Test ---");

int[] largeArray = new int[10000];

for (int i = 0; i < largeArray.length; i++) {

largeArray[i] = i \* 2;

}

long startTime = System.nanoTime();

boolean found = searchElement(largeArray, 9998);

long endTime = System.nanoTime();

System.out.println("Linear search in array of 10000 elements: " + found);

System.out.println("Time taken: " + (endTime - startTime) + " nanoseconds");

// Binary search performance

startTime = System.nanoTime();

found = binarySearchRecursive(largeArray, 9998, 0, largeArray.length - 1);

endTime = System.nanoTime();

System.out.println("Binary search in same array: " + found);

System.out.println("Time taken: " + (endTime - startTime) + " nanoseconds");

}

}

/\* Output:

=== Recursive Array Search ===

Array: [10, 25, 30, 45, 60, 75, 90, 45, 100]

--- Linear Search Tests ---

Searching for 45: Found at index 3

Searching for 100: Found at index 8

Searching for 10: Found at index 0

Searching for 999: Not found

Searching for 75: Found at index 5

--- Find All Occurrences of 45 ---

Found 45 at index 3

Found 45 at index 7

--- Binary Search on Sorted Array ---

Sorted Array: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

Binary search for 50: Found

Binary search for 25: Not found

Binary search for 100: Found

Binary search for 10: Found

Binary search for 999: Not found

--- Step-by-step Search for 60 ---

Searching at index 0 (value: 10)

Not a match, continuing search...

Searching at index 1 (value: 25)

Not a match, continuing search...

Searching at index 2 (value: 30)

Not a match, continuing search...

Searching at index 3 (value: 45)

Not a match, continuing search...

Searching at index 4 (value: 60)

Found target 60 at index 4

--- Performance Test ---

Linear search in array of 10000 elements: true

Time taken: 125000 nanoseconds

Binary search in same array: true

Time taken: 15000 nanoseconds

\*/

**2. Write a recursive function to count the digits of a positive integer (do also for sum of digits)**

**Answer:**

public class RecursiveDigitOperations {

// Count digits using recursion

public static int countDigits(int n) {

// Base case

if (n == 0) {

return 0;

}

// Recursive case

return 1 + countDigits(n / 10);

}

// Handle edge case for 0

public static int countDigitsComplete(int n) {

if (n == 0) {

return 1; // 0 has 1 digit

}

if (n < 0) {

n = -n; // Handle negative numbers

}

return countDigits(n);

}

// Sum of digits using recursion

public static int sumOfDigits(int n) {

// Base case

if (n == 0) {

return 0;

}

// Recursive case: last digit + sum of remaining digits

return (n % 10) + sumOfDigits(n / 10);

}

// Sum of digits handling negative numbers

public static int sumOfDigitsComplete(int n) {

if (n < 0) {

n = -n; // Make positive

}

return sumOfDigits(n);

}

// Product of digits using recursion

public static int productOfDigits(int n) {

// Base case

if (n == 0) {

return 1; // Empty product is 1

}

if (n < 10) {

return n; // Single digit

}

// Recursive case

return (n % 10) \* productOfDigits(n / 10);

}

// Count specific digit occurrences

public static int countSpecificDigit(int n, int digit) {

// Base case

if (n == 0) {

return (digit == 0) ? 1 : 0;

}

int count = 0;

if (n % 10 == digit) {

count = 1;

}

return count + countSpecificDigit(n / 10, digit);

}

// Find maximum digit

public static int maxDigit(int n) {

// Base case

if (n < 10) {

return n;

}

int lastDigit = n % 10;

int maxOfRest = maxDigit(n / 10);

return Math.max(lastDigit, maxOfRest);

}

// Find minimum digit

public static int minDigit(int n) {

// Base case

if (n < 10) {

return n;

}

int lastDigit = n % 10;

int minOfRest = minDigit(n / 10);

return Math.min(lastDigit, minOfRest);

}

// Reverse the number using recursion

public static int reverseNumber(int n) {

return reverseHelper(n, 0);

############################################################################

Info Box

############################################################################

<https://www.algorithmik.in/grokking-data-structures.pdf>

################################################################################################################################################################