Discussion-5: Advanced Concepts Related to Recursion

Discussion Topic:

Write a simple recursive program to demonstrate your understanding of the concept. What are the biggest benefits to creating a program or method that utilizes recursion in Java? In what scenario would it be appropriate to utilize a stack over a recursive implementation? Please provide an example to illustrate your points.

In your answer, specifically think of and give a real-life scenario where:

Recursion is used
Recursion is preferable over iteration

My Post:

In computer science, recursion is an algorithmic technique in which a function calls itself to solve smaller instances of the same problem. In programming, recursion is an amazing technique with the help of which we can reduce the length of our code and make it easier to read and write" (GeeksforGeeks, 2024, p1). Recursion is used in programming to solve complex problems involving repetition and hierarchical structures such as tree traversals, graph algorithms, and divide-and-conquer problems like sorting and searching.

The basic components found in recursive functions are base cases and recursive cases.
A base case is a condition that, when met, ends the recursion process (Ricciardi, 2024). A recursive case is a set of code lines that are executed until a base condition is met.
A classic example where recursion is well suited is in computing the factorial of a number. A factorial can be defined as a non-negative integer n, denoted $n!$, the product of all positive integers less than or equal to n:

$$n! = n \times (n-1)!$$

In Java:

```
public class Factorial {
    public static int factorial(int n) {
    // --- Base case: factorial of 0 is 1 ----
    if (n == 0) {
            return 1;
    // --- Recursive case ---
    } else {
        return n * factorial(n - 1);
    }
}
```
Note that the factorial() method calls itself until it reaches the base case where n = 0

There are various benefits to using recursion. One of the biggest benefits of using recursion is that it allows programmers to easily break down complex problems into simpler, more manageable subproblems. This approach is often referred to as the divide-and-conquer approach, it is implemented in algorithms like mergesort, where recursion divides a complex sort problem into smaller problems leading to a more efficient sort solution than the linear sort iterating solution.
Additionally, recursion helps with code readability by simplifying and shortening code lines. When using recursion, programmers can write problems involving repetition or hierarchical structures (trees) without the need to implement complex loops.
Recursion also simplifies, and it is efficient at handling dynamic and random data structures such as linked lists and tree structures. For instance, when traversing a binary tree, using recursion simplifies the implementation of the process without the need to implement a stack.

Although recursion has various advantages, in some scenarios using a stack is preferable over recursion. For example, recursion can generate a stack overflow error, 'StackOverflowError ', if the recursive depth (number of recursion calls) becomes too large. This can happen in cases like deep tree traversals or depth-first search algorithms, where the number of recursion calls may exceed the system's call stack capacity. In Java, the limit of the call stack varies depending on the platform used and the Java Virtual Machine implemented. Java stack size can be configured using the JVM argument '-Xss', for example 'java -Xss1M MyProgram' where 1M is the size of the call back for MyProgram (Goodrich, Tamassia, & Goldwasser, 2023). It is best practice to use a stack or tail recursion, if possible, in this scenario. "A recursion is a tail recursion if any recursive call that is made from one context is the very last operation in that context, with the return value of the recursive call (if any) immediately returned by the enclosing recursion" (Goodrich, Tamassia, & Goldwasser, 2023, 5.5 Eliminating tail recursion). Note that while some programming languages optimize tail-recursive functions, Java does not. Thus, in Java, an optimized tail-recursive function needs to be implemented implicitly.

Below are examples of implementing a depth-first search (DFS) traversal of a tree, using recursion with a possibility of 'StackOverflowError 'and a stack (Dequee) eliminating the possibility of a 'StackOverflowError ':

Using recursion possibility of 'StackOverflowError '

```java
public class DFS {
	// Node class
	static class Node {
		int value;
		Node left, right;

		// Constructor
		Node(int value) {
			this.value = value;
			left = right = null; // Left and right children are null initially
		}
	}

	// Recursive Depth-First Search (DFS) method
	public static void depthFirstSearchRecursive(Node node) {
		// --- Base case ---
		if (node == null) {
			return;
```

```
                }

                // Process the current node (visit)
                System.out.println(node.value);

                // Recursively traverse the left subtree
                depthFirstSearchRecursive(node.left);

                //--- Recursive case ---
                depthFirstSearchRecursive(node.right);

                /*
                 * Potential stack overflow issue: Each recursive call adds a new frame to the
                 * call stack. If the tree is too deep (e.g., with many levels), the recursion
                 * depth can exceed the system's maximum stack size, causing a
                 * StackOverflowError.
                 */
        }

        public static void main(String[] args) {
                // Create a binary tree
                Node root = new Node(1);
                root.left = new Node(2);
                root.right = new Node(3);
                root.left.left = new Node(4);
                root.left.right = new Node(5);

                System.out.println("DFS Traversal using Recursion:");
                depthFirstSearchRecursive(root);
        }
}
```

Using the stack approach eliminating the possibility of a ‘StackOverflowError’

```
import java.util.ArrayDeque;
import java.util.Deque;

public class DFS {
        // Single node in the binary tree
        static class Node {
                int value;
                Node left, right;

                // Node Constructor
                Node(int value) {
                        this.value = value;
                        left = right = null; // Left and right children are null initially
                }
        }

        // Depth-First Search (DFS) traversal method
        public static void depthFirstSearch(Node root) {
                Deque<Node> stack = new ArrayDeque<>();

                stack.push(root);

                // traverse the stack until is empty
                while (!stack.isEmpty()) {
                        // Pop the top node from the stack
                        Node current = stack.pop();
                System.out.println(current.value);


                if (current.right != null) {
                    stack.push(current.right); // Add right child to stack
                }

                if (current.left != null) {
                    stack.push(current.left); // Add left child to stack
                }
            }
        }

        public static void main(String[] args) {
            // Create a binary tree
            Node root = new Node(1);
            root.left = new Node(2);
            root.right = new Node(3);
            root.left.left = new Node(4);
            root.left.right = new Node(5);

            System.out.println("DFS Traversal using Deque:");
            depthFirstSearch(root);
        }
}
```

```
Output:
DFS Traversal using Deque:
1
2
4
5
3
```

To summarize, recursion is a technique in which a function calls itself to solve smaller instances of the same problem, it is often used in problems like tree traversal, graph algorithms, and divide-and-conquer strategies. While recursion simplifies complex problems and code readability, excessive recursive calls can lead to stack overflow errors, particularly in deeply nested structures such as trees, making iterative approaches using explicit stacks preferable in certain cases.

References:

Arslan, Ş. (2023, February 25). A Comprehensive tree traversal guide in Javascript - General and binary tree traversals. Shinar Arslan Blog. https://www.sahinarslan.tech/posts/a-comprehensive-tree-traversal-guide-in-javascript-general-and-binary-tree-traversals

GeeksforGeeks (2024, August 18). *Introduction to recursion*. GeeksforGeeks. https://www.geeksforgeeks.org/introduction-to-recursion-2/

Goodrich T, M., Tamassia, R., & Goldwasser H. M. (2023, June). Chapter 5: Algorithms recursion. *Data structures and algorithms*. zyBook ISBN: 979-8-203-40813-6.

Ricciardi, A. (2024, July 8). *Module 5: Discussion Forum* [Post]. Computer Science: CSC320 – Programming 2, Colorado State University Global.