Discussion-4: Working with the Stack Abstract Data Type

Discussion Topic:

This week we will be introduced to working with the ADT stack in Java. What are some of the basic principles involved with utilizing the ADT stack? What are some applications that could effectively utilize a stack? Write a simple Stack program to demonstrate your understanding.
In your answer, specifically think of and give a real-life scenario where:

- Stack ADT can be used
- Stack ADT usage or application is mandatory

My Post:

Hello class,

In computer science, a Atack is a fundamental Abstract Data Type (ADT) that follows the Last-In-First-Out (LIFO) principle, which means that the last item added to the Stack is the first to be removed. The term stack comes from the analogy of a stack of plates in a spring-loaded dispenser, commonly found in cafeterias (Carrano & Henry, 2018). Where plates are pushed onto the top of the stack and popped off from the top, making the push and pop operations fundamental to the Stack ADT's functionality.

The operation usually implemented in a Stack ADT are:

- Push (e): Adds a new element to the top of the stack.
- Pop (e): Removes and returns the top element of the Stack.
- Peek (e): Returns the top element without removing it.
- IsEmpty (): Checks if the Stack is empty.
- Clear (): Removes all elements from the Stack

A real-life scenario where the Stack ADT is used is in web browser history navigation. For example, each time a user leaves a web page, the browser pushes the page URL onto the URL history Stack. When the browser's "Back" button is clicked, the previous page's URL is popped from the top of the history Stack, and the current page's URL is pushed onto another Stack, let's call it the URL forward Stack. Conversely, when the user clicks the "Forward" button, the current page's URL is pushed onto the URL history stack, and the URL at the top of the URL forward stack is popped and displayed.

A real-life scenario where Stack ADT is a mandatory choice is in algebraic expression evaluation, particularly when converting an infix algebraic expression to a postfix algebraic expression. In infix expressions, operators are placed between operands (e.g., A + B); this is the common way used by humans but requires parentheses and precedence rules. On the other hand, in postfix expression, operators come after operands (e.g., A B +); there is no need for parentheses, and is easier for computers to evaluate using a Stack, it is solely based on the order of the operands and operators. In this scenario, the Stack ADT is perfectly suited to manage the order of operations. A stack is used to temporarily hold operators until their precedence and associativity are determined ensuring that the algebraic expression is evaluated in the correct order. This makes the Stack an efficient and essential ADT in such algorithms.

Another mandatory application of a Stack ADT is in managing function calls in programming, particularly in environments that support recursion, such as the Java Virtual Machine (JVM) or C++ runtime environments. When a function is called, the function's local variables and return addresses are pushed onto the call Stack. This allows JVM, for example, to keep track of where to return in the program execution after the function execution is completed, particularly during recursive and nested function calls.

Below is a simple Java program that mimics the basic functionality of the Java Stack memory.

```java
import java.util.EmptyStackException;

// Custom Stack class
public class MyLinkedStack<T> {
	private Node<T> top;
	private int size;

	// Node class to represent each element in the stack
	private static class Node<T> {
		private T data;
		private Node<T> next;

		public Node(T data) {
			this.data = data;
		}
	}

	public MyStack() {
		this.top = null;
		this.size = 0;
	}

	public void push(T data) {
		Node<T> newNode = new Node<>(data);
		newNode.next = top; // set the next reference to the current top
		top = newNode; // make the new node the top of the stack
		size++;
	}

	public T pop() {
		if (isEmpty()) {
			throw new EmptyStackException();
		}
		T poppedData = top.data;
		top = top.next; // remove the top node
		size--;
		return poppedData;
	}

	public T peek() {
		if (isEmpty()) {
			throw new EmptyStackException();
		}
		return top.data;
	}

	public boolean isEmpty() {
		return top == null;
	}

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

```
// Class with main method to test the stack
class Main {
	public static void MimicJavaStackMemory(String[] args) {
		MyLinkedStack<Integer> stack = new MyStack<>();

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

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

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

		System.out.println("Top element after popping: " + stack.peek());
		System.out.println("Stack size after popping: " + stack.size());
	}
}
```

Outputs:

```
Top element: 30
Stack size: 3
Popped element: 30
Popped element: 20
Top element after popping: 10
Stack size after popping: 1
```

To summarize, a Stack is a fundamental Abstract Data Type (ADT) that follows the Last-In-First-Out (LIFO) principle, making it ideal for applications like browser history navigation and algebraic expression evaluation. It is also essential in managing function calls in programming, particularly in environments that support recursion.

-Alex

References:

Carrano, F. M., & Henry, T. M. (2018, January 31). 6.1 Stacks. *Data structures and abstractions with Java* (5th Edition). Pearson.