Data Structures and Memory Stack in Function Calls

Presented by HieuTM-BH01236

Overview

- Introduction to essential data structures
- Focus on the memory stack
- How the memory stack supports function calls

Abstract Data Types (ADTs)

- Definition: ADTs are data models that define data and the operations that can be performed on them, abstracting away the implementation details.
- Purpose: Improve software design, development, and testing by providing clear specifications.

Benefits of ADTs

- Encapsulation: Hides the internal representation of data.
- Modularity: Breaks down complex systems into manageable components.
- Reusability: Promotes code reuse across different projects.
- Maintainability: Simplifies updates and modifications.

Stack ADT

A collection of elements with two principal operations: push (add an element) and pop (remove the most recently added element).

Operations Stack ADT

01

Push(x): Adds element x to the top.

02

Pop(): Removes and returns the top element.

03

Peek(): Returns the top element without removing it.

04

IsEmpty(): Checks if the stack is empty.

Node Class

- Defines a generic node class to represent each element in the stack.
- Contains data and a reference to the next node.

```
private static class Node<T> {
    private T data;
    private Node<T> next;

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

StackADT Class

- push(T item): Adds an element to the top of the stack.
- pop(): Removes and returns the top element of the stack. Throws EmptyStackException if the stack is empty.
- peek(): Returns the top element without removing it. Throws EmptyStackException if the stack is empty.
- isEmpty(): Checks if the stack is empty.

```
private Node<T> top;  // Top of the stack

// Push operation: Add an element to the top of the
public void push(T item) {
    Node<T> t = new Node<>(data: item);
    t.next = top;
    top = t;
}

// Pop operation: Remove and return the top element
public T pop() {
    if (top == null) {
        throw new EmptyStackException();
    }
    T item = top.data;
    top = top.next;
    return item;
}

// Peek operation: Return the top element of the sta
public T peek() {
    if (top == null) {
        throw new EmptyStackException();
    }
    return top.data;
}

// isEmpty operation: Check if the stack is empty
public boolean isEmpty() {
    return top == null;
}
```

Main Method

Demonstrates the use of the stack operations, including pushing elements onto the stack, peeking at the top element, popping elements from the stack, and checking if the stack is empty.

```
public static void main(String[] args) {
        StackADT<Integer> stack = new StackADT<>();

    // Test push operation
    stack.push(item: 10);
    stack.push(item: 20);
    stack.push(item: 30);

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

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

    // Test isEmpty operation
    System.out.println("Is stack empty? " + stack.isEmpty());  // Output: false

    // Pop remaining elements
    System.out.println("Popped element is: " + stack.pop());  // Output: 20
    System.out.println("Popped element is: " + stack.pop());  // Output: 10

    // Test isEmpty operation again
    System.out.println("Is stack empty? " + stack.isEmpty());  // Output: true
}
```

```
run:
Top element is: 30
Popped element is: 30
Is stack empty? false
Popped element is: 20
Popped element is: 10
Is stack empty? true
BUILD SUCCESSFUL (total time: 0 seconds)
```

FIFO Queue: Concrete Data Structure

Definition: A collection of elements with two principal operations: enqueue (add an element to the end) and dequeue (remove the front element).

Operations:

- Enqueue(x): Adds element x to the end.
- Dequeue(): Removes and returns the front element.
- Front(): Returns the front element without removing it.
- IsEmpty(): Checks if the queue is empty.

Example Queue

- A queue is initialized using LinkedList.
- Elements are added to the queue using the add method.
- The remove method removes the head of the queue and returns it
- The peek method returns the head of the queue without removing it.
- The size method returns the number of elements in the queue.
- The isEmpty method checks whether the queue is empty.
- The poll method retrieves and removes the head of the queue, or returns null if the queue is empty.

```
public static void main(String[] args) {
    // TODO code application logic here
     Queue<Integer> queue = new LinkedList<>();
    // Add elements to the queue
    queue.add(e: 10);
    queue.add(e: 20);
    queue.add(e: 30);
    queue.add(e: 40);
    queue.add(e: 50);
    // Display the elements of the queue
    System.out.println("Queue: " + queue);
    // Remove elements from the queue
    int removedElement = queue.remove();
    System.out.println("Removed element: " + removedElement);
    // Display the queue after removal
    System.out.println("Queue after removal: " + queue);
    // Peek at the front element of the queue
    int frontElement = queue.peek();
    System.out.println("Front element: " + frontElement);
    // Check the size of the queue
    int size = queue.size();
    System.out.println("Size of the queue: " + size);
    // Check if the queue is empty
    boolean isEmpty = queue.isEmpty();
    System.out.println("Is the queue empty? " + isEmpty);
    // Poll an element from the queue
    int polledElement = queue.poll();
    System.out.println("Polled element: " + polledElement);
    // Display the queue after polling
    System.out.println("Queue after polling: " + queue);
}
```

Memory Stack

A memory stack is a fundamental data structure in computer science and computing, operating on the Last In, First Out (LIFO) principle. This means that the most recently added item is the first to be removed. The memory stack is crucial for managing the execution of function calls in programming.

Memory Stack in Implementing Function Calls

The memory stack is essential in managing function calls, especially in recursive functions and nested function calls. The stack is used to store:

- Function Parameters: The arguments passed to the function.
- Return Address: The address in the code to return to after the function execution is complete.
- Local Variables: The variables that are declared within the function.
- Saved Registers: The state of CPU registers before the function call.

Example Memory Stack

```java
package stack;

import java.util.Stack;

public class MemoryStackExample {

    public static void main(String[] args) {
        // Example string to be reversed
        String input = "Hello, World!";

        // Creating a stack
        Stack<Character> stack = new Stack<>();

        // Pushing each character of the string into the stack
        for (char ch : input.toCharArray()) {
            stack.push(item: ch);
        }

        // Popping characters from the stack and forming the reversed string
        StringBuilder reversed = new StringBuilder();
        while (!stack.isEmpty()) {
            reversed.append(obj:stack.pop());
        }

        // Output the reversed string
        System.out.println("Original string: " + input);
        System.out.println("Reversed string: " + reversed.toString());
    }
}
```

Package and Imports:

- The code is part of the stack package and imports the Stack class from java.util.

Class and Main Method:

- MemoryStackExample class contains the main method, the entry point of the application.

Input String:

- Defines the string input as "Hello, World!".

Create Stack:

- Creates a Stack<Character> to store characters of the input string.

Push Characters:

- Loops through each character of input and pushes it onto the stack.

Reverse String:

- Pops each character from the stack and appends it to a StringBuilder, reversing the string in the process.

```
run:
Original string: Hello, World!
Reversed string: !dlroW ,olleH
BUILD SUCCESSFUL (total time: 0 seconds)
```

Output Results: Prints the original and reversed strings.

```java
public class MemoryStack {
    private int maxSize;
    private int[] stackArray;
    private int top;

    public MemoryStack(int size) {
        maxSize = size;
        stackArray = new int[maxSize];
        top = -1; // stack is initially empty
    }


    public void push(int value) {
        if (top == maxSize - 1) {
            System.out.println("Stack is full. Cannot push " + value);
        } else {
            stackArray[++top] = value;
        }
    }


    public int pop() {
        if (isEmpty()) {
            System.out.println(x: "Stack is empty. Cannot pop.");
            return -1;
        } else {
            return stackArray[top--];
        }
    }
```

- Class Definition: MemoryStack class manages a stack using an array.
- Instance Variables:
- maxSize: Maximum size of the stack.
- stackArray: Array to store stack elements.
- top: Index of the top element in the stack.
- Constructor: Initializes the stack with a specified size and sets top to -1 (indicating an empty stack).
- Push Method: Adds an element to the top of the stack if it's not full.
- Pop Method: Removes and returns the top element of the stack if it's not empty.

```java
public int peek() {
    if (isEmpty()) {
        System.out.println(x: "Stack is empty. Cannot peek.");
        return -1;
    } else {
        return stackArray[top];
    }
}


public boolean isEmpty() {
    return (top == -1);
}


public boolean isFull() {
    return (top == maxSize - 1);
}

public void display() {
    if (isEmpty()) {
        System.out.println(x: "Stack is empty.");
    } else {
        System.out.print(s: "Stack (bottom to top): ");
        for (int i = 0; i <= top; i++) {
            System.out.print(stackArray[i] + " ");
        }
        System.out.println();
    }
}
```

- Peek Method: Returns the top element without removing it if the stack is not empty.
- isEmpty Method: Checks if the stack is empty.
- isFull Method: Checks if the stack is full.
- Display Method: Prints all elements in the stack from bottom to top.
- Main Method: Demonstrates the stack operations: pushing, popping, peeking, checking if the stack is empty or full, and displaying the stack.

Output - Stack (run)

```
run:
Stack (bottom to top): 10 20 30 40 50
Top element: 50
Popped element: 50
Stack (bottom to top): 10 20 30 40
Popped element: 40
Stack (bottom to top): 10 20 30
Is stack empty? false
Stack (bottom to top): 10 20 30 60
Is stack full? false
BUILD SUCCESSFUL (total time: 0 seconds)
```

This output demonstrates stack operations including pushing, popping, peeking, and checking if the stack is empty or full.

Application Design Requirements

Features:

- Input student information: ID, Name, Marks
- Manage number of students
- Output detailed student information and ranking
- CRUD operations: Add, Edit, Delete
- Sort and Search functionalities

Student Ranking Table:

- Marks ranges and corresponding ranks

Proposed Alternative Algorithm

Current Algorithm: Simple Bubble Sort
Alternative Algorithm: QuickSort

- Advantages: Faster average performance, especially with large datasets
- Evaluation: Compare time complexity and performance with real data

Conculsion

Summary: ADTs provide a structured approach to data management, enhancing software design and development.

- Importance of ADTs,
- Example implementations
- Benefits of using well-defined ADTs in software development

Thank you very much!