Data Structures: Stacks Week Plan

Authors: Refat Othman and Diaeddin Rimawi.

Lecture 1: Introduction to Stacks (2 hours)

1. What is a Stack?

- Abstract data type representing a collection of elements.
- Follows the **LIFO** principle (Last In, First Out).
- Operations:
  - **push**: Add an element to the top
  - **pop**: Remove the top element
  - **peek**: View the top element without removing it
- Real-life analogy: A stack of plates — only the top plate can be added or removed

2. Core Stack Operations (with visual examples)

- **Push Example**:
  - Stack (top at left): `Top -> 2 -> 1` → push(3) → `Top -> 3 -> 2 -> 1`
- **Pop Example**:
  - Stack: `Top -> 3 -> 2 -> 1` → pop() → `Top -> 2 -> 1`
- **Peek Example**:
  - Stack: `Top -> 2 -> 1` → peek() → `2`

3. Stack Implementations

- **Using a Linked List** (Top at Head - Efficient Access O(1)):

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedStack:
    def __init__(self):
        self.top = None

    def push(self, data):
        new_node = Node(data)
        new_node.next = self.top
        self.top = new_node
```

```
    def pop(self):
        if self.top is None:
            return None
        to_return = self.top.data
        self.top = self.top.next
        return to_return

    def peek(self):
        return None if self.top is None else self.top.data

    def is_empty(self):
        return self.top is None

    def clear(self):
        self.top = None
```

- **Using an Array** (Top at End):

```
class ArrayStack:
    def __init__(self, capacity):
        self.stack = [None] * capacity
        self.capacity = capacity
        self.n = -1

    def is_empty(self):
        return self.n == -1

    def is_full(self):
        return self.n == self.capacity - 1

    def push(self, data):
        if self.is_full():
            print("Stack is full. Cannot push.")
            return
        self.n += 1
        self.stack[self.n] = data

    def pop(self):
        if self.is_empty():
            return None
        value = self.stack[self.n]
        self.n -= 1
        return value

    def peek(self):
        if self.is_empty():
            return None
        return self.stack[self.n]
```

- **Resizing Strategy (Homework Idea):**

- Double the array size when full
- Shrink to half when the stack is one-quarter full
- Compare: Linked List uses more space (due to links), Arrays offer better cache performance and amortized O(1) operations

4. Applications of Stack

4.1 Undo/Redo in Editors / Web Browser Backtracking

Two stacks can be used: one for undo, another for redo.

```
undo_stack = []
redo_stack = []

def perform_action(action):
    undo_stack.append(action)
    redo_stack.clear()

def undo():
    if undo_stack:
        last_action = undo_stack.pop()
        redo_stack.append(last_action)
        print(f"Undo: {last_action}")

def redo():
    if redo_stack:
        last_redo = redo_stack.pop()
        undo_stack.append(last_redo)
        print(f"Redo: {last_redo}")

# Example:
perform_action("Type A")
perform_action("Type B")
undo()
redo()
```

4.2 JVM Call Stack The Java Virtual Machine (JVM) is the engine that runs Java applications. When a Java program is executed, the JVM creates a call stack to keep track of method calls. Each time a method is invoked, a stack frame is pushed onto the stack to store information such as local variables, the return address, and operand data. When the method finishes execution, its stack frame is popped from the stack. This mechanism ensures proper execution flow, especially in nested or recursive method calls.

```
call_stack = []

def call_function(func_name):
    call_stack.append(func_name)
    print(f"Calling: {func_name}")

def return_from_function():
```

```
    if call_stack:
        finished = call_stack.pop()
        print(f"Returning from: {finished}")

# Example usage:
call_function("main")
call_function("foo")
return_from_function()
return_from_function()
```

4.3 Recursive Function Calls (Runtime Stack)

Recursive functions naturally use a stack to track calls.

```
def factorial(n):
    call_stack = []
    result = 1
    while n > 1:
        call_stack.append(n)
        n -= 1
    while call_stack:
        result *= call_stack.pop()
    return result

# Example usage:
print(factorial(5))  # Output: 120
```

Lecture 2: Exercises on Stacks (2 hours)

1. Exercise 1: Valid Parentheses

- Input: "{[()]}"
- Output: `true` if balanced; `false` otherwise
- Real-world: Compilers, syntax checkers

2. Exercise 2: Evaluate Reverse Polish Notation (Postfix)

- Input: ["2", "1", "+", "3", "*"] → Output: `9`
- Use a stack to process operands and operators
- Application: Calculator design

3. Exercise 3: Palindrome String Check

- Problem: Given a string, determine if it is a palindrome. A palindrome reads the same forward and backward (e.g., "radar", "level").
- Input: "madam"

- Output: `True` because "madam" is the same backward.

Assignment for Next Week Task: Convert an infix expression to postfix (Reverse Polish Notation)

- Example: "(1 + 2) * 3" → "1 2 + 3 *"
- Use a stack to store operators based on precedence