Tutorial 7: Stacks

Data Structures and Algorithms

4 March, 2025

1 Introduction to Stacks

A **stack** is a linear data structure that follows the **LIFO (Last In, First Out)** principle. This means that the last element inserted into the stack is the first one to be removed. It is often compared to a stack of plates, where adding a new plate places it on top, and removing a plate takes the top one first.

Stacks are widely used in computing due to their simplicity and efficiency. They allow only a limited set of operations, making them suitable for managing data in situations where order matters.

1.1 Basic Stack Operations

The primary operations on a stack are:

- **Push**: Adds an element to the top of the stack.
- **Pop**: Removes the top element from the stack.
- **Peek/Top**: Retrieves the top element without removing it.
- **isEmpty**: Checks whether the stack is empty.

Stacks can be implemented using **arrays** or **linked lists**. The choice of implementation depends on the specific use case, with arrays offering fixed-size storage and linked lists providing dynamic memory allocation.

1.2 Applications of Stacks

Stacks are commonly used in various applications, including:

- **Function call management**: The system call stack keeps track of function calls and their local variables.
- **Undo/Redo functionality**: Many applications use stacks to manage previous states.
- **Expression evaluation and conversion**: Converting between infix, postfix, and prefix expressions.

- **Backtracking algorithms**: Used in maze-solving and depth-first search algorithms.
- **Syntax validation**: Checking whether brackets and parentheses are balanced in programming languages.

1.3 Pseudocode for Stack Operations

The following pseudocode defines a simple stack and its core operations:

```
procedure PUSH(stack, element):
    if stack is full:
        print "Stack Overflow"
        return
    else:
        add element to top of stack
        return

procedure POP(stack):
    if stack is empty:
        print "Stack Underflow"
        return NULL
    else:
        remove top element from stack
        return top element

procedure PEEK(stack):
    if stack is empty:
        print "Stack is empty"
        return NULL
    else:
        return top element

procedure ISEMPTY(stack):
    return true if stack is empty, otherwise false
```

2 Parenthesis Matching Problem

A common application of stacks is checking whether an expression contains balanced parentheses. The problem involves ensuring that every opening bracket has a corresponding closing bracket in the correct order.

2.1 Problem Statement

Given a string containing different types of parentheses (‘()‘, ‘{}‘, ‘[]‘), determine if the parentheses are balanced. A balanced expression follows these rules:

- Each opening bracket has a corresponding closing bracket.
- Brackets are closed in the correct order.
- The stack should be empty at the end if the expression is balanced.

2.2 Pseudocode Solution

We can solve this problem using a stack:

```
procedure IS_BALANCED(expression):
    initialize an empty stack
    for each character in expression:
        if character is an opening bracket:
            push it onto the stack
        else if character is a closing bracket:
            if stack is empty:
                return false
            top_element = pop from stack
            if top_element does not match character:
                return false
    if stack is empty:
        return true
    else:
        return false
```

2.3 Example Solution

Students are expected to provide a step-by-step breakdown of how the stack changes during the process. Below is an example:

Input: ‘[( )] { } { [( ) ( )] ( ) }‘

- Read ‘[‘, push to stack - Stack: ‘[‘
- Read ‘(‘, push to stack - Stack: ‘[(‘
- Read ‘)‘, pop ‘(‘ - Stack: ‘[‘ (matched)
- Read ‘]‘, pop ‘[‘ - Stack: ‘ ‘ (matched)
- Read ‘{‘, push to stack - Stack: ‘{‘
- Read ‘}‘, pop ‘{‘ - Stack: ‘ ‘ (matched)
- Read ‘{‘, push to stack - Stack: ‘{‘
- Read ‘[‘, push to stack - Stack: ‘{ [‘
- Read ‘(‘, push to stack - Stack: ‘{ [(‘

- Read ‘)‘, pop ‘(‘ - Stack: ‘{ [‘ (matched)
- Read ‘(‘, push to stack - Stack: ‘{ [(‘
- Read ‘)‘, pop ‘(‘ - Stack: ‘{ [‘ (matched)
- Read ‘]‘, pop ‘[‘ - Stack: ‘{‘ (matched)
- Read ‘(‘, push to stack - Stack: ‘{ (‘
- Read ‘)‘, pop ‘(‘ - Stack: ‘{‘ (matched)
- Read ‘}‘, pop ‘{‘ - Stack: ‘ ‘ (matched)

Since the stack is empty at the end, the expression is balanced.