1. ADT Stack

A Stack is a linear data structure that follows the Last-In, First-Out (LIFO) principle. Imagine a stack of plates; you can only add a new plate to the top, and you can only remove the top plate.

1.1. Stack Operations: Algorithms and Complexity Analysis

operation	description	algorithm	time complexity	space complexity
`push(item)`	Adds an element to the top of the stack.	1. Increment `top`. 2. Assign item to `stack[top]`. 3. If `top` exceeds `MAX_SIZE`, report “Stack Overflow”.	O(1)	O(1)
`pop()`	Removes and returns the top element from the stack.	1. If top is -1, report “Stack Underflow”. 2. Store stack[top] in a temporary variable. 3. Decrement top. 4. Return the stored element.	O(1)	O(1)
`peek()` / `top()`	Returns the top element without removing it.	1. If top is -1, report “Stack Empty”. 2. Return stack[top].	O(1)	O(1)
`isEmpty()`	Checks if the stack is empty.	1. Return true if top is -1, false otherwise.	O(1)	O(1)
`isFull()`	Checks if the stack is full (if implemented with a fixed-size array).	1. Return true if top is MAX_SIZE - 1, false otherwise.	O(1)	O(1)

2. Applications of Stacks

Stacks are widely used in various computing scenarios due to their LIFO nature.

2.1. Expression Conversion

Converting expressions between Infix, Prefix (Polish Notation), and Postfix (Reverse Polish Notation) forms is a classic application of stacks.

- Infix: Operators are between operands (e.g., `A + B`).
- Prefix: Operators precede operands (e.g., `+ A B`).
- Postfix: Operators follow operands (e.g., `A B +`).

2.1.1. Infix to Postfix Conversion

Algorithm:

- Initialize an empty stack and an empty result string/list.
- Scan the infix expression from left to right.
- If an operand is encountered, append it to the result.
- If an opening parenthesis `(` is encountered, push it onto the stack.
- If a closing parenthesis `)` is encountered, pop operators from the stack and append them to the result until an opening parenthesis is found. Pop and discard the opening parenthesis.
- If an operator is encountered:
  - While the stack is not empty and the top of the stack is an operator with higher or equal precedence than the current operator, pop the operator from the stack and append it to the result.
  - Push the current operator onto the stack.
- After scanning the entire expression, pop any remaining operators from the stack and append them to the result.

Complexity Analysis: The algorithm involves scanning the expression once and performing constant-time stack operations for each character. Therefore, the time complexity is O(N), where N is the length of the expression. Space complexity is O(N) in the worst case (for the stack).

2.1.2. Infix to Prefix Conversion

This is similar to Infix to Postfix, but involves scanning from right to left, and handling parentheses and precedence appropriately.

C Code for Infix to Postfix Conversion

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```c
#include <ctype.h> // For isalnum()

// Stack for operator storage
char operatorStack[MAX_SIZE];
int opTop = -1;

void opPush(char op) {
    operatorStack[++opTop] = op;
}

char opPop() {
    if (opTop == -1) return '\0'; // Error or empty
    return operatorStack[opTop--];
}

char opPeek() {
    if (opTop == -1) return '\0'; // Error or empty
    return operatorStack[opTop];
}

int opIsEmpty() {
    return (opTop == -1);
}

// Function to get precedence of an operator
int precedence(char op) {
    if (op == '+' || op == '-')
        return 1;
    else if (op == '*' || op == '/')
        return 2;
    else if (op == '^') // For exponentiation
        return 3;
    return 0; // For parentheses
}

// Function to convert infix expression to postfix
void infixToPostfix(char* infix) {
    char postfix[MAX_SIZE];
    int i, j;
    i = j = 0;

    while (infix[i] != '\0') {
        char symbol = infix[i];

        if (isalnum(symbol)) { // If operand, add to postfix
            postfix[j++] = symbol;
        } else if (symbol == '(') { // If opening parenthesis, push to stack
            opPush(symbol);
        } else if (symbol == ')') { // If closing parenthesis
```

```
            while (opTop != -1 && opPeek() != '(') {
                postfix[j++] = opPop();
            }
            if (opTop != -1 && opPeek() == '(') { // Pop '('
                opPop();
            }
        } else { // If operator
            while (opTop != -1 && precedence(opPeek()) >= precedence(symbol))
{
                postfix[j++] = opPop();
            }
            opPush(symbol);
        }
        i++;
    }

    while (opTop != -1) { // Pop remaining operators from stack
        postfix[j++] = opPop();
    }
    postfix[j] = '\0'; // Null-terminate the postfix string
    printf("Postfix expression: %s\n", postfix);
}


int main() {
    char infixExp[] = "a+b*(c^d-e)^(f+g*h)-i"; // Example
    infixToPostfix(infixExp);
    return 0;
}
```

2.2. Expression Evaluation

Stacks are also crucial for evaluating postfix and prefix expressions.

2.2.1. Postfix Expression Evaluation

Algorithm:

- Initialize an empty operand stack.
- Scan the postfix expression from left to right.
- If an operand is encountered, push it onto the operand stack.
- If an operator is encountered:
  - Pop the top two operands from the stack (operand2 then operand1).
  - Perform the operation (operand1 operator operand2).
  - Push the result back onto the operand stack.

- After scanning the entire expression, the final result will be the only element left on the operand stack.

Complexity Analysis: Similar to conversion, the algorithm scans the expression once and performs constant-time stack operations. Thus, the time complexity is O(N), and space complexity is O(N) in the worst case.

C Code for Postfix Expression Evaluation

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h> // For isdigit()

// Stack for operand storage (integers)
int operandStack[MAX_SIZE];
int operandTop = -1;

void operandPush(int num) {
    operandStack[++operandTop] = num;
}

int operandPop() {
    if (operandTop == -1) {
        printf("Error: Stack underflow during evaluation.\n");
        exit(EXIT_FAILURE);
    }
    return operandStack[operandTop--];
}

// Function to perform arithmetic operation
int operate(int op1, int op2, char operator) {
    switch (operator) {
        case '+': return op1 + op2;
        case '-': return op1 - op2;
        case '*': return op1 * op2;
        case '/': return op1 / op2;
        case '^': { // For power operation
            int res = 1;
            for (int i = 0; i < op2; i++) res *= op1;
            return res;
        }
    }
    return 0; // Should not reach here for valid operators
}

// Function to evaluate a postfix expression
int evaluatePostfix(char* postfix) {
    int i;
    for (i = 0; postfix[i] != '\0'; i++) {
        char symbol = postfix[i];

        if (isdigit(symbol)) { // If digit, convert to int and push
            operandPush(symbol - '0'); // Convert char digit to int
        } else { // If operator
            int op2 = operandPop(); // Pop second operand
            int op1 = operandPop(); // Pop first operand
            int result = operate(op1, op2, symbol);
            operandPush(result);
        }
    }
    return operandPop(); // The final result
}

int main() {
    char postfixExp[] = "231*+9-"; // Corresponds to (2 + 3 * 1) - 9 = 5 - 9
= -4
    printf("Result of postfix evaluation: %d\n",
evaluatePostfix(postfixExp));
    return 0;
}
```