1.What is a Stack?

- Definition: A stack is a linear data structure that follows the LIFO (Last In, First Out) principle, meaning the last element added to the stack is the first one to be removed. It operates similarly to a stack of plates where you can only add or remove the top plate.

- Example: Consider the operations:

1. Push(1)

2. Push(2)

3. Push(3)

The stack after these operations is: [1, 2, 3].

- Pop operation will remove 3, leaving `[1, 2]`.

- The next Pop operation removes 2, leaving `[1]`, and finally, the Pop operation removes 1, leaving an empty stack.

2. Stack Operations

- Push(x): Adds element `x` to the top of the stack.

- Pop(): Removes the top element from the stack.

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

- isEmpty(): Checks if the stack is empty.

- Size(): Returns the number of elements in the stack.

- Time Complexities:

- Push: O(1)

- Pop: O(1)

- Peek: O(1)

- isEmpty: O(1)

- Size: O(1)

These operations are constant time because the stack only needs to access the top element for all these operations.

3. Applications of Stacks

- Expression Evaluation: Stacks are used in evaluating arithmetic expressions, especially in \*\*postfix notation\*\* (Reverse Polish Notation). For example, `3 4 + 2 \*` evaluates to `(3 + 4) \* 2`.

- Undo Mechanism: Stacks can track previous states, as seen in undo functionality in text editors. Every time an action is performed, the state is pushed onto the stack, and when "undo" is pressed, the top state is popped.

- Function Call Management: The call stack in programming keeps track of function calls, local variables, and return addresses during program execution.

4. Stack Overflow and Underflow

- Stack Overflow: Occurs when an attempt is made to push an element onto a full stack (e.g., the stack has reached its capacity). In recursive functions, this may happen if the recursion depth exceeds the system's stack limit.

- Prevention: Increase the stack size or use dynamic resizing in the implementation. For recursion, consider using iterative solutions or tail recursion optimizations.

- Stack Underflow: Occurs when an attempt is made to pop an element from an empty stack.

- Prevention: Always check if the stack is empty (using `isEmpty()` function) before performing a pop operation.

5. Converting Infix to Postfix

- Algorithm: The goal is to reorder the expression such that operators come after operands in a way that eliminates the need for parentheses.

Steps:

1. Initialize an empty stack for operators and an empty list for the output.

2. Iterate over each character of the infix expression:

- If it is an operand, add it to the output list.

- If it is an operator:

- Pop operators from the stack and add them to the output list until an operator with lower precedence or a left parenthesis is at the top of the stack.

- Push the current operator onto the stack.

- If it is a left parenthesis, push it onto the stack.

- If it is a right parenthesis, pop operators from the stack to the output until a left parenthesis is encountered.

3. After the entire expression is processed, pop any remaining operators from the stack to the output list.

- \*\*Example\*\*: Convert `A + B \* C` to postfix:

- Operand `A` → output: `A`

- Operator `+` → push onto stack.

- Operand `B` → output: `A B`

- Operator `\*` → push onto stack because `\*` has higher precedence than `+`.

- Operand `C` → output: `A B C`

- Pop `\*` from stack → output: `A B C \*`

- Pop `+` from stack → output: `A B C \* +`

Resulting postfix expression: `A B C \* +`.

6. Balanced Parentheses Problem

- \*\*Balanced Parentheses\*\*: An expression is balanced if every opening parenthesis has a corresponding closing parenthesis and they are correctly nested.

- Algorithm (Using Stack):

1. Initialize an empty stack.

2. Traverse the string character by character:

- If the character is an opening parenthesis `(`, push it onto the stack.

- If the character is a closing parenthesis `)`, pop the top of the stack. If the stack is empty or the popped element is not an opening parenthesis, the expression is unbalanced.

3. After processing all characters, if the stack is not empty, the expression is unbalanced (extra opening parentheses).

- Example: For the expression `(A + B) \* (C + D)`, the parentheses are balanced because each opening parenthesis has a matching closing parenthesis.

7.Evaluating Postfix Expressions

- Algorithm:

1. Initialize an empty stack.

2. Traverse the postfix expression from left to right:

- If the element is an operand, push it onto the stack.

- If the element is an operator, pop the two topmost operands from the stack, apply the operator, and push the result back onto the stack.

3. After processing the entire expression, the result will be the only element left in the stack.

- Example: Evaluate `3 4 + 2 \* 7 /`:

- Push 3, then 4 → stack: `[3, 4]`

- Operator `+`: Pop 4 and 3, compute `3 + 4 = 7`, push 7 → stack: `[7]`

- Push 2 → stack: `[7, 2]`

- Operator `\*`: Pop 2 and 7, compute `7 \* 2 = 14`, push 14 → stack: `[14]`

- Push 7 → stack: `[14, 7]`

- Operator `/`: Pop 7 and 14, compute `14 / 7 = 2`, push 2 → stack: `[2]`

- Result: 2

### 8. \*\*Stack as a Memory Model\*\*

- The call stack in programming keeps track of function calls. When a function is called, its local variables and the return address are pushed onto the stack. After the function finishes executing, the stack is popped to return control to the previous function.

- In the case of recursion, each recursive call pushes a new frame onto the stack, and when recursion unwinds, each frame is popped off.

9. Reverse a String Using Stack

- Algorithm:

1. Initialize an empty stack.

2. Push each character of the string onto the stack.

3. Pop each character from the stack and append it to the result string.

- Why Stack?: The stack's LIFO nature makes it ideal for reversing the order of characters.

- Example: For the string "Hello":

1. Push `H`, `e`, `l`, `l`, `o` onto the stack.

2. Pop characters in reverse order to get "olleH".

10. Stack in Depth-First Search (DFS)

- \*\*DFS with Stack\*\*: DFS explores a graph by pushing nodes onto a stack. It starts from a source node, visits its neighbors, and pushes them onto the stack. The next node to be visited is the one on top of the stack.

- Comparison:

- Stack-based DFS: Uses an explicit stack to store the path, which can be manually managed in iterative implementations.

- Recursion-based DFS: The system uses the call stack to manage function calls, with each recursive call pushing the current state of the search onto the stack.

- Memory Usage:

- Recursion-based DFS may face stack overflow issues for deep recursion, while stack-based DFS has more control over the depth and can be optimized for memory.