Submitted To:

Engineer Sheharyar Khan

Name: Iqra Fatima

Reg. Number: 23-CP-62

Semester: 4th

Department: CPED

Data Structures and Algorithms

(DSA)  
Lab Report 7

# Examples

Example 1:

**Code:**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**Output:**

**A screenshot of a computer

AI-generated content may be incorrect.**

Example 2:

**Code:**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**A white background with black text

AI-generated content may be incorrect.**

**Output:**

**A close up of a number

AI-generated content may be incorrect.**

# Exercise

## Easy Problems

1. **Stack Push & Pop**: Implement a stack where users can push and pop elements interactively

**Code:**

**class Stack:**

**def \_\_init\_\_(self):**

**self.stack = []**

**def push(self, item):**

**self.stack.append(item)**

**print(f"Pushed: {item}")**

**def pop(self):**

**if self.is\_empty():**

**print("Stack is empty! Cannot pop.")**

**else:**

**print(f"Popped: {self.stack.pop()}")**

**def is\_empty(self):**

**return len(self.stack) == 0**

**def display(self):**

**print("Stack:", self.stack)**

**stack = Stack()**

**while True:**

**choice = input("Enter 'push', 'pop', or 'exit': ").strip().lower()**

**if choice == 'push':**

**item = input("Enter item to push: ")**

**stack.push(item)**

**elif choice == 'pop':**

**stack.pop()**

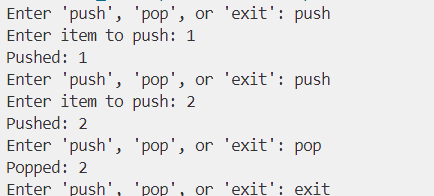
**elif choice == 'exit':**

**break**

**else:**

**print("Invalid choice!")**

**Output:**



1. **Check Stack is Empty**: Write a function to check if a stack is empty.

**Code:**

**def is\_empty(stack):**

**return len(stack) == 0**

**stack = []**

**print("Stack is empty:", is\_empty(stack))  # True**

**stack.append(5)**

**print("Stack is empty:", is\_empty(stack))  # False**

**Output:**

**A close up of a text

AI-generated content may be incorrect.**

1. **Peek Implementation**: Implement a peek operation to retrieve the topmost element.

**Code:**

**def peek(stack):**

**if stack:**

**return stack[-1]**

**return None  # If stack is empty**

**stack = [10, 20, 30]**

**print("Top element:", peek(stack))**

**Output:**

****

1. **Reverse a String using Stack**: Reverse a given string using stack operations.

**Code:**

**def reverse\_string(s):**

**stack = list(s)**

**reversed\_str = ""**

**while stack:**

**reversed\_str += stack.pop()**

**return reversed\_str**

**print(reverse\_string("hello"))  # Output: "olleh"**

**Output:**

****

1. **Check Balanced Parentheses**: Write a function to check if parentheses in an expression are balanced.

**Code:**

**def is\_balanced(expr):**

**stack = []**

**mapping = {')': '(', '}': '{', ']': '['}**

**for char in expr:**

**if char in "({[":**

**stack.append(char)**

**elif char in ")}]":**

**if not stack or stack.pop() != mapping[char]:**

**return False**

**return not stack**

**print(is\_balanced("({[]})"))  # Output: True**

**print(is\_balanced("({[})"))   # Output: False**

**Output:**



## Intermediate Problems

1. **Undo/Redo System**: Implement an undo/redo system using two stacks.

**Code:**

**class UndoRedo:**

**def \_\_init\_\_(self):**

**self.undo\_stack = []**

**self.redo\_stack = []**

**def do(self, action):**

**self.undo\_stack.append(action)**

**self.redo\_stack.clear()**

**print(f"Action performed: {action}")**

**def undo(self):**

**if self.undo\_stack:**

**action = self.undo\_stack.pop()**

**self.redo\_stack.append(action)**

**print(f"Undo: {action}")**

**else:**

**print("Nothing to undo!")**

**def redo(self):**

**if self.redo\_stack:**

**action = self.redo\_stack.pop()**

**self.undo\_stack.append(action)**

**print(f"Redo: {action}")**

**else:**

**print("Nothing to redo!")**

**editor = UndoRedo()**

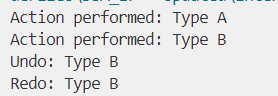
**editor.do("Type A")**

**editor.do("Type B")**

**editor.undo()**

**editor.redo()**

**Output:**

****

1. **Evaluate Postfix Expression**: Implement a function to evaluate a postfix expression.

**Code:**

**def evaluate\_postfix(expression):**

**stack = []**

**for token in expression.split():**

**if token.isdigit():**

**stack.append(int(token))**

**else:**

**b, a = stack.pop(), stack.pop()**

**if token == '+':**

**stack.append(a + b)**

**elif token == '-':**

**stack.append(a - b)**

**elif token == '\*':**

**stack.append(a \* b)**

**elif token == '/':**

**stack.append(a // b)**

**return stack.pop()**

**print(evaluate\_postfix("3 4 + 2 \* 7 /"))  # Output: 2**

**Output:**

****

1. **Browser Back & Forward Navigation**: Simulate browser history using stacks.

**Code:**

**class BrowserHistory:**

**def \_\_init\_\_(self):**

**self.back\_stack = []**

**self.forward\_stack = []**

**def visit(self, site):**

**print(f"Visiting: {site}")**

**self.back\_stack.append(site)**

**self.forward\_stack.clear()**

**def back(self):**

**if len(self.back\_stack) > 1:**

**self.forward\_stack.append(self.back\_stack.pop())**

**print(f"Back to: {self.back\_stack[-1]}")**

**else:**

**print("No more history to go back!")**

**def forward(self):**

**if self.forward\_stack:**

**self.back\_stack.append(self.forward\_stack.pop())**

**print(f"Forward to: {self.back\_stack[-1]}")**

**else:**

**print("No forward history!")**

**browser = BrowserHistory()**

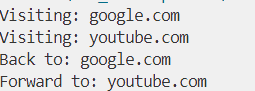
**browser.visit("google.com")**

**browser.visit("youtube.com")**

**browser.back()**

**browser.forward()**

**Output:**

****

1. **Sort a Stack**: Implement a function to sort a stack using recursion.

**Code:**

**def sorted\_insert(stack, element):**

**if not stack or element > stack[-1]:**

**stack.append(element)**

**else:**

**temp = stack.pop()**

**sorted\_insert(stack, element)**

**stack.append(temp)**

**def sort\_stack(stack):**

**if stack:**

**temp = stack.pop()**

**sort\_stack(stack)**

**sorted\_insert(stack, temp)**

**stack = [3, 1, 4, 2]**

**sort\_stack(stack)**

**print(stack)**

**Output:**

****

1. **Recursive Stack Traversal**: Implement stack traversal using recursion instead of loops.

**Code:**

**def traverse(stack):**

**if stack:**

**print(stack.pop(), end=" ")**

**traverse(stack)**

**stack = [1, 2, 3, 4, 5]**

**traverse(stack)**

**Output:**

****

## Advanced Problems

1. **Call Stack Simulation**: Simulate recursive function calls using a stack.

**Code:**

**def call\_stack\_simulation(n):**

**stack = []**

**while n > 0:**

**stack.append(n)**

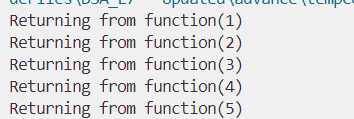
**n -= 1**

**while stack:**

**print(f"Returning from function({stack.pop()})")**

**call\_stack\_simulation(5)**

**Output:**



1. **Stack-Based Expression Evaluator**: Implement an advanced calculator supporting parentheses and operator precedence.

**Code:**

**import operator**

**def evaluate(expression):**

**precedence = {'+': 1, '-': 1, '\*': 2, '/': 2}**

**ops = {'+': operator.add, '-': operator.sub, '\*': operator.mul, '/': operator.floordiv}**

**stack, postfix = [], []**

**for token in expression.split():**

**if token.isdigit():**

**postfix.append(int(token))**

**else:**

**while stack and precedence.get(stack[-1], 0) >= precedence[token]:**

**postfix.append(stack.pop())**

**stack.append(token)**

**while stack:**

**postfix.append(stack.pop())**

**eval\_stack = []**

**for token in postfix:**

**if isinstance(token, int):**

**eval\_stack.append(token)**

**else:**

**b, a = eval\_stack.pop(), eval\_stack.pop()**

**eval\_stack.append(ops[token](a, b))**

**return eval\_stack.pop()**

**print(evaluate("3 + 5 \* 2"))**

**Output:**



1. **Tower of Hanoi Problem**: Solve the Tower of Hanoi using a stack-based approach.

**Code:**

**class Move:**

**def \_\_init\_\_(self, n, source, auxiliary, destination):**

**self.n = n**

**self.source = source**

**self.auxiliary = auxiliary**

**self.destination = destination**

**def tower\_of\_hanoi\_stack(n):**

**stack = []**

**moves = []**

**# Push initial move onto the stack**

**stack.append(Move(n, 'A', 'B', 'C'))**

**while stack:**

**move = stack.pop()**

**if move.n == 1:**

**# Base case: Move a single disk**

**moves.append(f"Move disk 1 from {move.source} to {move.destination}")**

**else:**

**# Push moves in reverse order to simulate recursion using stack**

**stack.append(Move(move.n - 1, move.auxiliary, move.source, move.destination))**

**stack.append(Move(1, move.source, move.auxiliary, move.destination)) # Move largest disk**

**stack.append(Move(move.n - 1, move.source, move.destination, move.auxiliary))**

**return moves**

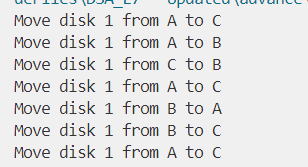
**# Example: Solve Tower of Hanoi for 3 disks**

**moves = tower\_of\_hanoi\_stack(3)**

**for step in moves:**

**print(step)**

**Output:**

****