Data Structures and Algorithms

(DSA)

Lab Report 7

Name: Iqra Fatima

Reg. Number: 23-CP-62

Semester: 4th

Department: CPED

Submitted To:

Engineer Sheharyar Khan

Examples

Example 1:

Code:

```python
#Task 1: Implementing a Stack using Arrays
class StackArray:
    def __init__(self, size):
        self.stack = []
        self.size = size

    def push(self, item):
        if len(self.stack) < self.size:
            self.stack.append(item)
        else:
            print("Stack Overflow: Cannot add more elements!")

    def pop(self):
        if self.stack:
            return self.stack.pop()
        else:
            print("Stack Underflow: No elements to pop!")
            return None

    def peek(self):
        return self.stack[-1] if self.stack else None

    def isEmpty(self):
        return len(self.stack) == 0

    def display(self):
        print("Stack:", self.stack)

# Example Usage
stack = StackArray(5)
stack.push(10)
stack.push(20)
stack.push(30)
stack.display()
stack.pop()
stack.display()
```

Output:

```
Stack: [10, 20, 30]
Stack: [10, 20]
```

Example 2:

Code:

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

class StackLinkedList:
    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:
            print("Stack Underflow: No elements to pop!")
            return None
        popped_data = self.top.data
        self.top = self.top.next
        return popped_data

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

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

    def display(self):
        current = self.top
        print("Stack:", end=" ")
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")
```

```
# Example Usage
stack = StackLinkedList()
stack.push(10)
stack.push(20)
stack.push(30)
stack.display()
stack.pop()
stack.display()
```

Output:

```
Stack: 30 -> 20 -> 10 -> None
Stack: 20 -> 10 -> None
PS D:\4thSemester\DSA(Python)\DSA_Lab\DSA_
```

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:

```
Enter 'push', 'pop', or 'exit': push
Enter item to push: 1
Pushed: 1
Enter 'push', 'pop', or 'exit': push
Enter item to push: 2
Pushed: 2
Enter 'push', 'pop', or 'exit': pop
Popped: 2
Enter 'push', 'pop', or 'exit': exit
```

2. **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:

```
Stack is empty: True
Stack is empty: False
```

3. **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:

```
Top element: 30
PS D:\4thSemester\DS
```

4. **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:

```
olleh
```

5. **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:

```
True
False
```

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:

```
Action performed: Type A
Action performed: Type B
Undo: Type B
Redo: Type B
```

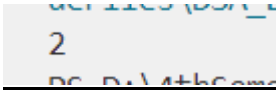
2. **Evaluate Postfix Expression**: Implement a function to evaluate a postfix expression.

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:



3. **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:

```
Visiting: google.com
Visiting: youtube.com
Back to: google.com
Forward to: youtube.com
```

4. **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, 2, 3, 4]
```

5. **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:

```
5 4 3 2 1
```

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:

```
Returning from function(1)
Returning from function(2)
Returning from function(3)
Returning from function(4)
Returning from function(5)
```

2. **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:

```
13
PS D:\4thS
```

3. **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:

```
Move disk 1 from A to C
Move disk 1 from A to B
Move disk 1 from C to B
Move disk 1 from A to C
Move disk 1 from B to A
Move disk 1 from B to C
Move disk 1 from A to C
```