ASSIGNMENT – 11.2

B .Sri Laxmi

Gayathri

2303A52033

Batch- 38

Task Description -1 – (Stack Using AI Guidance)

• Task: With the help of AI, design and implement a Stack data structure supporting basic stack operations.

Expected Output:

• A Python Stack class supporting push, pop, peek, and empty- check operations with proper documentation.

PROMPT:

Design and implement a Stack class in Python. The stack should support push, pop, peek, and is_empty operations. Use a list to store elements internally. Include proper documentation and example usage.

CODE:

```
class Stack:
    def __init__(self):
        self._items = []
    def push(self, item):
        self._items.append(item)
    def pop(self):
        if self.is_empty():
            raise IndexError("pop from empty stack")
        return self._items.pop()
    def peek(self):
        if self.is_empty():
            raise IndexError("peek from empty stack")
        return self._items[-1]
    def is_empty(self):
        return len(self._items) == 0
    def size(self):
        return len(self._items)
    def __str__(self):
        return f"Stack: {self._items}"
    def __repr__(self):
        return f"Stack({self._items})"
```

```python
# Example Usage:
my_stack = Stack()
print(f"Is stack empty? {my_stack.is_empty()}")  # Expected: True
my_stack.push(10)
my_stack.push(20)
my_stack.push(30)
print(f"Stack after pushes: {my_stack}") # Expected: Stack: [10, 20, 30]
print(f"Size of stack: {my_stack.size()}") # Expected: 3
print(f"Is stack empty? {my_stack.is_empty()}")  # Expected: False
print(f"Peek: {my_stack.peek()}") # Expected: 30
print(f"Stack after peek: {my_stack}") # Expected: Stack: [10, 20, 30]
popped_item = my_stack.pop()
print(f"Popped item: {popped_item}") # Expected: 30
print(f"Stack after pop: {my_stack}") # Expected: Stack: [10, 20]
print(f"Peek after pop: {my_stack.peek()}") # Expected: 20
my_stack.pop()
my_stack.pop()
print(f"Stack after all pops: {my_stack}") # Expected: Stack: []
print(f"Is stack empty? {my_stack.is_empty()}")  # Expected: True
try:
    my_stack.pop()
except IndexError as e:
    print(f"Error when popping from empty stack: {e}")
try:
    my_stack.peek()
except IndexError as e:
    print(f"Error when peeking from empty stack: {e}")
```

OUTPUT:

```
Is stack empty? True
Stack after pushes: Stack: [10, 20, 30]
Size of stack: 3
Is stack empty? False
Peek: 30
Stack after peek: Stack: [10, 20, 30]
Popped item: 30
Stack after pop: Stack: [10, 20]
Peek after pop: 20
Stack after all pops: Stack: []
Is stack empty? True
Error when popping from empty stack: pop from empty stack
Error when peeking from empty stack: peek from empty stack
```

EXPLANATION:

This code creates a simple Stack using a Python list. A stack follows the Last In, First Out (LIFO) rule, which means the last item added is removed first. The push() method adds an item to the stack, and the pop() method removes the top item. The peek() method shows the top item without removing it. The is_empty() method checks if the stack has no elements, and size() tells how many items are in the stack. The example part shows how items are added, removed, checked, and how errors are handled when trying to remove or view items from an empty stack.

Task Description -2 – (Queue Design)

• Task:Use AI assistance to create a Queue data structure

following FIFO principles

Expected Output:

• A complete Queue implementation including enqueue, dequeue,

front element access, and size calculation

PROMPT:

Design and implement a Queue class in Python. The queue should follow FIFO (First In, First Out) principle. Include methods: enqueue, dequeue, front, and size. Use a list to store elements and show example usage**.**

CODE:

```
class Queue:
    def __init__(self):
        self._items = []
    def enqueue(self, item):
        self._items.append(item)
    def dequeue(self):
        if self.is_empty():
            raise IndexError("dequeue from empty queue")
        return self._items.pop(0) # Removing from the front
    def front(self):
        if self.is_empty():
            raise IndexError("front from empty queue")
        return self._items[0]
    def is_empty(self):
        return len(self._items) == 0
    def size(self):
        return len(self._items)
    def __str__(self):
        return f"Queue: {self._items}"
    def __repr__(self):
        return f"Queue({self._items})"
```

```
my_queue = Queue()
print(f"Is queue empty? {my_queue.is_empty()}")  # Expected: True
print(f"Queue size: {my_queue.size()}")  # Expected: 0
my_queue.enqueue("Alice")
my_queue.enqueue("Bob")
my_queue.enqueue("Charlie")
print(f"Queue after enqueues: {my_queue}") # Expected: Queue: ['Alice', 'Bob', 'Charlie']
print(f"Queue size: {my_queue.size()}") # Expected: 3
print(f"Is queue empty? {my_queue.is_empty()}")  # Expected: False
print(f"Front of queue: {my_queue.front()}") # Expected: Alice
print(f"Queue after front: {my_queue}") # Expected: Queue: ['Alice', 'Bob', 'Charlie']
dequeued_item = my_queue.dequeue()
print(f"Dequeued item: {dequeued_item}") # Expected: Alice
print(f"Queue after dequeue: {my_queue}") # Expected: Queue: ['Bob', 'Charlie']
print(f"Front of queue after dequeue: {my_queue.front()}") # Expected: Bob
my_queue.dequeue()
my_queue.dequeue()
print(f"Queue after all dequeues: {my_queue}") # Expected: Queue: []
print(f"Is queue empty? {my_queue.is_empty()}")  # Expected: True
# Attempting to dequeue from an empty queue (will raise an IndexError)
try:
    my_queue.dequeue()
except IndexError as e:
    print(f"Error when dequeuing from empty queue: {e}")
# Attempting to get front from an empty queue (will raise an IndexError)
try:
    my_queue.front()
except IndexError as e:
    print(f"Error when getting front of empty queue: {e}")
```

OUTPUT:

```
Is queue empty? True
Queue size: 0
Queue after enqueues: Queue: ['Alice', 'Bob', 'Charlie']
Queue size: 3
Is queue empty? False
Front of queue: Alice
Queue after front: Queue: ['Alice', 'Bob', 'Charlie']
Dequeued item: Alice
Queue after dequeue: Queue: ['Bob', 'Charlie']
Front of queue after dequeue: Bob
Queue after all dequeues: Queue: []
Is queue empty? True
Error when dequeuing from empty queue: dequeue from empty queue
Error when getting front of empty queue: front from empty queue
```

EXPLANATION:

This code creates a simple Queue using a Python list. A queue works on the First In, First Out (FIFO) rule, which means the first item added will be removed first. The enqueue() method adds an item to the end of the queue. The dequeue() method removes the first item from the queue. The front() method shows the first item without removing it. The is_empty() method checks if the queue is empty, and size() tells how many items are in the queue. The example shows how items are added, removed inorder, and how errors are handled when the queue is empty.

Task Description -3 –(Singly Linked List Construction)

• Task: Utilize AI to build a singly linked list supporting insertion

and traversal.

Expected Output:

• Correctly functioning linked list with node creation, insertion

logic, and display functionality.

PROMPT:

Design and implement a Stack class in Python. The stack should support push, pop, peek, and is_empty operations. Use a list to store elements internally. Include proper documentation and example usage

CODE:

```python
class Node:
    def __init__(self, data):
        self.data = data
        self.next_node = None
class SinglyLinkedList:
    def __init__(self):
        self.head = None
    def insert_at_end(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            return
        last_node = self.head
        while last_node.next_node:
            last_node = last_node.next_node
        last_node.next_node = new_node
    def display(self):
        current = self.head
        if current is None:
            print("Linked List is empty.")
            return
        elements = []
        while current:
            elements.append(str(current.data))
            current = current.next_node
        print(" -> ".join(elements))
    def __str__(self):
        elements = []
        current = self.head
        while current:
            elements.append(str(current.data))
            current = current.next_node
        return " -> ".join(elements) if elements else "Empty List"
```

```python
# Example Usage:
my_list = SinglyLinkedList()
print("Initial list:")
my_list.display() # Expected: Linked List is empty.
my_list.insert_at_end(10)
my_list.insert_at_end(20)
my_list.insert_at_end(30)
print("List after insertions:")
my_list.display() # Expected: 10 -> 20 -> 30
my_list.insert_at_end(40)
print("List after one more insertion:")
my_list.display() # Expected: 10 -> 20 -> 30 -> 40
# Test with an empty list display again
empty_list = SinglyLinkedList()
print("\nEmpty list display:")
empty_list.display() # Expected: Linked List is empty.
```

OUTPUT:

```
Initial list:
Linked List is empty.
List after insertions:
10 -> 20 -> 30
List after one more insertion:
10 -> 20 -> 30 -> 40

Empty list display:
Linked List is empty.
```

EXPLANATION:

This code creates a simple Singly Linked List. The Node class stores the data and a link to the next node. The SinglyLinkedList class keeps track of the first node using head. The insert_at_end() method adds new elements to the end of the list. The display() method prints all elements in order. The example shows inserting numbers and displaying them, and it also shows what happens when the list is empty.

Task Description -4 – (Binary Search Tree Operations)

• Task: Implement a Binary Search Tree with AI support focusing

on insertion and traversal.

Expected Output:

• BST program with correct node insertion and in-order traversal

output.

PROMPT:

Design and implement a Binary Search Tree (BST) in Python. Include node creation, insertion method, and in-order traversal function. The tree should follow BST rules for inserting elements. Provide example usage with traversal output**.**

CODE:

```python
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
class BinarySearchTree:
    def __init__(self):
        self.root = None
    def insert(self, data):
        new_node = Node(data)
        if self.root is None:
            self.root = new_node
        else:
            self._insert_recursive(self.root, new_node)
    def _insert_recursive(self, current_node, new_node):
        if new_node.data < current_node.data:
            if current_node.left is None:
                current_node.left = new_node
            else:
                self._insert_recursive(current_node.left, new_node)
        else: # new_node.data >= current_node.data (handling duplicates by placing on right
            if current_node.right is None:
                current_node.right = new_node
            else:
                self._insert_recursive(current_node.right, new_node)
    def in_order_traversal(self):
        if self.root is None:
            print("BST is empty.")
            return
        elements = []
        self._in_order_recursive(self.root, elements)
        print(" -> ".join(map(str, elements)))
    def _in_order_recursive(self, current_node, elements):
```

```python
            return
        elements = []
        self._in_order_recursive(self.root, elements)
        print(" -> ".join(map(str, elements)))
    def _in_order_recursive(self, current_node, elements):
        if current_node.left:
            self._in_order_recursive(current_node.left, elements)
        elements.append(current_node.data)
        if current_node.right:
            self._in_order_recursive(current_node.right, elements)
    def __str__(self):
        if self.root is None:
            return "Empty BST"
        elements = []
        self._in_order_recursive(self.root, elements)
        return " -> ".join(map(str, elements))
    def __repr__(self):
        return f"BinarySearchTree(root={self.root.data if self.root else None})"
```

```python
# Example Usage:
bst = BinarySearchTree()
print("Initial BST (in-order traversal):")
bst.in_order_traversal() # Expected: BST is empty
# Insert elements
bst.insert(50)
bst.insert(30)
bst.insert(70)
bst.insert(20)
```

```
bst.insert(35) # Inserting between 30 and 40
bst.insert(75) # Inserting between 70 and 80
bst.insert(50) # Inserting a duplicate
print("\nBST after insertions (in-order traversal - should be sorted):")
bst.in_order_traversal() # Expected: 20 -> 30 -> 35 -> 40 -> 50 -> 50 -> 60 -> 70 -> 75 -> 80
print("\nString representation of BST:")
print(bst) # Expected: 20 -> 30 -> 35 -> 40 -> 50 -> 50 -> 60 -> 70 -> 75 -> 80
# Test with another empty BST
empty_bst = BinarySearchTree()
print("\nEmpty BST traversal:")
empty_bst.in_order_traversal() # Expected: BST is empty.
```

```
Initial BST (in-order traversal):
BST is empty.

BST after insertions (in-order traversal - should be sorted):
20 -> 30 -> 35 -> 40 -> 50 -> 50 -> 60 -> 70 -> 75 -> 80

String representation of BST:
20 -> 30 -> 35 -> 40 -> 50 -> 50 -> 60 -> 70 -> 75 -> 80

Empty BST traversal:
BST is empty.
```

EXPLANATION:

This code creates a simple **Binary Search Tree (BST)**. The Node class stores a value and has two links: left and right. The BinarySearchTree class keeps track of the first node called the root. The insert() method adds numbers to the tree following the rule: smaller numbers go to the left, and bigger numbers go to the right. The tree uses recursion to find the correct place for each new value. The in_order_traversal() method prints the elements in sorted order. If the tree is empty, it shows a message saying the BST is empty.

Task Description -5 – (Hash Table Implementation)

• Task: Create a hash table using AI with collision handling

Expected Output:

• Hash table supporting insert, search, and delete using chaining or open

PROMPT:

```python
class Node:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.next = None
class HashTable:
    def __init__(self, capacity=10):
        self.capacity = capacity
        self.buckets = [None] * self.capacity # Each bucket will be the head of a linked list
    def _hash_function(self, key):
        return hash(key) % self.capacity
    def insert(self, key, value):
        index = self._hash_function(key)
        # If the bucket is empty, create a new Node as the head
        if self.buckets[index] is None:
            self.buckets[index] = Node(key, value)
            return
        current = self.buckets[index]
        prev = None
        while current:
            if current.key == key:
                current.value = value # Key already exists, update value
                return
            prev = current
            current = current.next
        # Key does not exist in the linked list, add new node to the end
        prev.next = Node(key, value)
    def search(self, key):
        index = self. hash function(key)
```

```python
        while current:
            if current.key == key:
                return current.value
            current = current.next
        return None # Key not found
    def delete(self, key):
        index = self._hash_function(key)
        current = self.buckets[index]
        prev = None

        while current:
            if current.key == key:
                if prev:
                    prev.next = current.next
                else:
                    self.buckets[index] = current.next # Node to delete is the head
                return True # Key found and deleted
            prev = current
            current = current.next
        return False # Key not found
    def display(self):
        print("--- Hash Table Contents ---")
        for i in range(self.capacity):
            print(f"Bucket {i}:")
            current = self.buckets[i]
            if current is None:
                print("  (empty)")
            else:
                elements = []
                while current:
                    elements.append(f"({current.key}: {current.value})")
                    current = current.next
                print("  " + " -> ".join(elements))
        print("----------------------------")
```

```python
# Example Usage:
my_hash_table = HashTable(capacity=5)
print("Initial Hash Table:")
my_hash_table.display()
# Insert operations
my_hash_table.insert("apple", 10)
my_hash_table.insert("banana", 20)
my_hash_table.insert("cherry", 30)
my_hash_table.insert("date", 40)
my_hash_table.insert("elderberry", 50)
my_hash_table.insert("fig", 60) # This will likely cause a collision
print("\nHash Table after insertions:")
my_hash_table.display()
# Search operations
print("\n--- Search Results ---")
print(f"Search 'apple': {my_hash_table.search('apple')}") # Expected: 10
print(f"Search 'cherry': {my_hash_table.search('cherry')}") # Expected: 30
print(f"Search 'grape': {my_hash_table.search('grape')}") # Expected: None
print(f"Search 'fig': {my_hash_table.search('fig')}") # Expected: 60
# Update an existing key
my_hash_table.insert("apple", 100)
print(f"Updated 'apple': {my_hash_table.search('apple')}") # Expected: 100
print("\nHash Table after updating 'apple':")
my_hash_table.display()
# Delete operations
print("\n--- Delete Operations ---")
print(f"Delete 'banana': {my_hash_table.delete('banana')}") # Expected: True
print(f"Search 'banana' after deletion: {my_hash_table.search('banana')}") # Expected: None
print(f"Delete 'grape' (not found): {my_hash_table.delete('grape')}") # Expected: False
print("\nHash Table after deleting 'banana':")
my_hash_table.display()
# Delete a key that is part of a chain
print(f"Delete 'fig': {my_hash_table.delete('fig')}") # Expected: True
print("\nHash Table after deleting 'fig':")
my_hash_table.display()
# Test edge case: delete from an empty bucket or non-existent key
empty_ht = HashTable(2)
empty_ht.display()
print(f"Delete 'nonexistent': {empty_ht.delete('nonexistent')}") # Expected: False
```

OUTPUT:

```
Hash Table after insertions:
--- Hash Table Contents ---
Bucket 0:
  (date: 40)
Bucket 1:
  (empty)
Bucket 2:
  (apple: 10) -> (cherry: 30)
Bucket 3:
  (banana: 20) -> (elderberry: 50) -> (fig: 60)
Bucket 4:
  (empty)
----------------------------

--- Search Results ---
Search 'apple': 10
Search 'cherry': 30
Search 'grape': None
Search 'fig': 60
Updated 'apple': 100

Hash Table after updating 'apple':
--- Hash Table Contents ---
Bucket 0:
  (date: 40)
Bucket 1:
  (empty)
Bucket 2:
  (apple: 100) -> (cherry: 30)
Bucket 3:
  (banana: 20) -> (elderberry: 50) -> (fig: 60)
Bucket 4:
  (empty)
```

EXPLANATION:

This code creates a simple **Hash Table** to store key–value pairs. It uses a hash function to decide which bucket (index) each key should go into. If two keys go to the same bucket, it handles the collision using a linked list (chaining). The insert() method adds a key and value to the table, and if the key already exists, it updates the value. The search() method finds and returns the value for a given key. The delete() method removes a key from the table. The display() method shows all the buckets and their stored data. The example shows inserting, searching, updating, deleting, and handling cases where a key is not found.