Name:B.Arjun

Enrollno:2403a510a9

Assignment:11.1

Task1:

Prompt:

Generate a Stack class with push, pop, peek, and is empty methods.

Code:

```python
class Stack:
    def __init__(self):
        self.items = []  # initialize an empty list for the stack

    def push(self, item):
        """Add an item to the top of the stack"""
        self.items.append(item)

    def pop(self):
        """Remove and return the top item of the stack"""
        if not self.is_empty():
            return self.items.pop()
        return "Stack is empty"

    def peek(self):
        """Return the top item without removing it"""
        if not self.is_empty():
            return self.items[-1]
        return "Stack is empty"

    def is_empty(self):
        """Check if the stack is empty"""
        return len(self.items) == 0


# Example usage:
s = Stack()
s.push(10)
s.push(20)
print(s.peek())     # 20
print(s.pop())      # 20
print(s.is_empty())  # False
print(s.pop())      # 10
print(s.is_empty())  # True
```

Output:

```
PS C:\Users\ASHMITHA\Desktop\ai asss> & C:/Users/ASHMITHA/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/ASHMITHA/Desktop/ai asss/task1.py"
20
20
False
10
True
```

Observation:A stack object is created using stack = Stack().

At this point, the stack is empty → stack.is_empty() returns True.

Push Operation

Elements (5, 10, 15) are pushed one by one.

Internally, they are stored in a Python list → [5, 10, 15].

Each push() call adds the element to the top of the stack (end of the list).

Peek Operation

stack.peek() checks the last element without removing it.

Returns 15 (the top element) while keeping the stack as [5, 10, 15].

Pop Operation

stack.pop() removes and returns the top element.

Returns 15 and updates the stack to [5, 10].

Peek after Pop

stack.peek() now returns 10 (new top of the stack).

Final Check

stack.is_empty() returns False since stack still has [5, 10].

Task2:

Prompt:

Implement a Queue using Python lists.

Code:

task2.py > q

```python
class Queue:
    def __init__(self):
        self.items = []

    def enqueue(self, item):
        """Add an item to the end of the queue"""
        self.items.append(item)

    def dequeue(self):
        """Remove and return the item from the front of the queue"""
        if not self.is_empty():
            return self.items.pop(0)
        return "Queue is empty"

    def peek(self):
        """Return the front item without removing it"""
        if not self.is_empty():
            return self.items[0]
        return "Queue is empty"

    def is_empty(self):
        """Check if the queue is empty"""
        return len(self.items) == 0
    def size(self):
        """Return the number of items in the queue"""
        return len(self.items)
q = Queue()
q.enqueue(10)
q.enqueue(20)
q.enqueue(30)

print(q.peek())     # 10 (front of queue)
print(q.dequeue())  # 10 (removed from front)
print(q.size())     # 2
print(q.is_empty()) # False
```

Output:

```
PS C:\Users\ASHMITHA\Desktop\ai asss> & C:/Users/ASHMITHA/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/ASHMITHA/Desktop/ai asss/task2.py"
10
10
2
False
```

Observation:

The queue works on the **FIFO (First-In, First-Out)** principle. When elements 10, 20, and 30 are enqueued, they are stored in the order [10, 20, 30]. The peek() method shows 10 as the front element without removing it. The dequeue() operation removes the first inserted element (10), leaving [20, 30] in the queue. The size() method then shows 2, and the is_empty() method correctly returns False since the queue still contains elements.

Task3:

Prompt:

generate a Singly Linked List with insert and display methods.

Code:

task3.py > ...

```python
class Node:
    def __init__(self, data):
        self.data = data      # store data
        self.next = None      # pointer to the next node


class SinglyLinkedList:
    def __init__(self):
        self.head = None      # start with an empty list

    def insert(self, data):
        """Insert a new node at the end of the linked list"""
        new_node = Node(data)
        if self.head is None:     # if list is empty
            self.head = new_node
        else:
            temp = self.head
            while temp.next:      # traverse until last node
                temp = temp.next
            temp.next = new_node  # link new node at the end

    def display(self):
        """Display all nodes in the linked list"""
        if self.head is None:
            print("List is empty")
        else:
            temp = self.head
            while temp:
                print(temp.data, end=" -> ")
                temp = temp.next
            print("None")   # indicate end of list
ll = SinglyLinkedList()
ll.insert(10)
ll.insert(20)
ll.insert(30)

ll.display()   # Output: 10 -> 20 -> 30 -> None
```

Output:

```
PS C:\Users\ASHMITHA\Desktop\ai asss> & C:/Users/ASHMITHA/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/ASH
MITHA/Desktop/ai asss/task3.py"
10 -> 20 -> 30 -> None
PS C:\Users\ASHMITHA\Desktop\ai asss>
```

Observation:

The program creates a singly linked list where each node contains some data and a pointer to the next node. Using the insert() method, new nodes (10, 20, and 30) are successfully added to the end of the list. The display() method then traverses the list starting from the head and prints the nodes in sequence as 10 -> 20 -> 30 -> None, which shows the correct linkage between nodes and confirms that insertion and traversal are working properly.

Task4:

Prompt:

Create a BST with insert and in-order traversal methods.

Code:

task4.py > ...

```python
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
class BST:
    def __init__(self):
        self.root = None
    def insert(self, data):
        """Insert a new node into the BST"""
        if self.root is None:
            self.root = Node(data)
        else:
            self._insert(self.root, data)

    def _insert(self, current, data):
        if data < current.data:  # go left
            if current.left is None:
                current.left = Node(data)
            else:
                self._insert(current.left, data)
        elif data > current.data:  # go right
            if current.right is None:
                current.right = Node(data)
            else:
                self._insert(current.right, data)
        # if data == current.data, do nothing (no duplicates)

    def inorder(self):
        """Perform in-order traversal of the BST"""
        return self._inorder(self.root)

    def _inorder(self, node):
        result = []
        if node:
            result += self._inorder(node.left)   # left
            result.append(node.data)             # root
```

```python
    def _inorder(self, node):
        result = []
        if node:
            result += self._inorder(node.left)   # left
            result.append(node.data)             # root
            result += self._inorder(node.right)  # right
        return result
bst = BST()
bst.insert(50)
bst.insert(30)
bst.insert(70)
bst.insert(20)
bst.insert(40)
bst.insert(60)
bst.insert(80)
print("In-order Traversal:", bst.inorder())
```

Output:

```
> & C:/Users/ASHMITHA/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/ASH
MITHA/Desktop/ai asss/task4.py"
In-order Traversal: [20, 30, 40, 50, 60, 70, 80]
PS C:\Users\ASHMITHA\Desktop\ai asss>
```

Observation:

The program constructs a Binary Search Tree (BST) by inserting nodes such that smaller values are placed in the left subtree and larger values in the right subtree. After inserting the nodes 50, 30, 70, 20, 40, 60, 80, the BST structure is correctly formed. The inorder() traversal visits nodes in the order **Left → Root → Right**, which produces the output [20, 30, 40, 50, 60, 70, 80]. This result is sorted, confirming that the BST insertion and in-order traversal methods work as expected.

Task5:

Prompt:

Implement a hash table with basic insert, search, and delete methods.

Code:

```python
class HashTable:
    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(size)]  # list of buckets

    def _hash(self, key):
        """Simple hash function"""
        return hash(key) % self.size

    def put(self, key, value):
        """Insert or update a key-value pair"""
        index = self._hash(key)
        bucket = self.table[index]

        # check if key already exists -> update
        for i, (k, v) in enumerate(bucket):
            if k == key:
                bucket[i] = (key, value)
                return
        # otherwise insert new pair
        bucket.append((key, value))

    def get(self, key):
        """Search for a key and return its value"""
        index = self._hash(key)
        bucket = self.table[index]

        for k, v in bucket:
            if k == key:
                return v
        return None  # key not found

    def remove(self, key):
        """Delete a key-value pair"""
        index = self._hash(key)
        bucket = self.table[index]

```

```python
class HashTable:

    def remove(self, key):
        """Delete a key-value pair"""
        index = self._hash(key)
        bucket = self.table[index]

        for i, (k, v) in enumerate(bucket):
            if k == key:
                del bucket[i]
                return True
        return False  # key not found

    def display(self):
        """Show the hash table contents"""
        for i, bucket in enumerate(self.table):
            print(f"Bucket {i}: {bucket}")


# Example usage
ht = HashTable()

# Insert
ht.put("apple", 10)
ht.put("banana", 20)
ht.put("grape", 30)

# Display
ht.display()

# Search
print("Search 'banana':", ht.get("banana"))

# Delete
ht.remove("apple")
ht.display()
```

Output:

```
PS C:\Users\ASHMITHA\Desktop\ai asss> & C:/Users/ASHMITHA/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/ASH
MITHA/Desktop/ai asss/task5.py"
Bucket 0: []
Bucket 1: []
Bucket 2: []
Bucket 3: []
Bucket 4: [('banana', 20)]
Bucket 5: []
Bucket 6: []
Bucket 7: [('apple', 10)]
Bucket 8: [('grape', 30)]
Bucket 9: []
Search 'banana': 20
Bucket 0: []
Bucket 1: []
Bucket 2: []
Bucket 3: []
Bucket 4: [('banana', 20)]
Bucket 5: []
Bucket 6: []
Bucket 7: []
Bucket 8: [('grape', 30)]
Bucket 9: []
PS C:\Users\ASHMITHA\Desktop\ai asss>
```

Observation:

The program successfully implements a hash table using lists with chaining to handle collisions. Keys are mapped to specific buckets using a hash function (hash(key) % size). When inserting, key–value pairs such as ("apple", 10), ("banana", 20), and ("grape", 30) are placed in their respective buckets. The get() method retrieves values correctly (e.g., searching for "banana" returns 20). The remove() method deletes a key–value pair (e.g., "apple" is removed, leaving only "banana" and "grape"). The display() method confirms the distribution of elements across buckets. Thus, insertion, search, and deletion operations work as expected in the hash table.

Task6:

Prompt:

Implement a graph using an adjacency list.

Code:

```
task6.py > ...
class Graph:
    def __init__(self):
        self.adj_list = {}   # dictionary to hold adjacency list

    def add_vertex(self, vertex):
        """Add a new vertex to the graph"""
        if vertex not in self.adj_list:
            self.adj_list[vertex] = []

    def add_edge(self, v1, v2):
        """Add an undirected edge between v1 and v2"""
        if v1 in self.adj_list and v2 in self.adj_list:
            self.adj_list[v1].append(v2)
            self.adj_list[v2].append(v1)  # remove this line for directed graph

    def display(self):
        """Display the adjacency list of the graph"""
        for vertex in self.adj_list:
            print(vertex, "->", self.adj_list[vertex])
g = Graph()
# Add vertices
g.add_vertex("A")
g.add_vertex("B")
g.add_vertex("C")
g.add_vertex("D")
# Add edges
g.add_edge("A", "B")
g.add_edge("A", "C")
g.add_edge("B", "D")
g.add_edge("C", "D")
# Display graph
g.display()
```

Output:

```
PS C:\Users\ASHMITHA\Desktop\ai asss> & C:/Users/ASHMITHA/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/ASHMITHA/Desktop/ai asss/task6.py"
A -> ['B', 'C']
B -> ['A', 'D']
C -> ['A', 'D']
D -> ['B', 'C']
PS C:\Users\ASHMITHA\Desktop\ai asss>
```

Observation:

The program successfully implements a hash table using lists with chaining to handle collisions. Keys are mapped to specific buckets using a hash function (hash(key) % size). When inserting, key–value pairs such as ("apple", 10), ("banana", 20), and ("grape", 30) are placed in their respective buckets. The get() method retrieves values correctly (e.g., searching for "banana" returns 20). The remove() method deletes a key–value pair (e.g., "apple" is removed, leaving only "banana" and "grape"). The display() method confirms the distribution of elements across buckets. Thus, insertion, search, and deletion operations work as expected in the hash table.

Task7:

Prompt:

Implement a priority queue using Python’s heapq module.

Code:

```
task7.py > ...
class PriorityQueue:

    def push(self, priority, item):
        """Insert an item with a given priority"""
        heapq.heappush(self.heap, (priority, item))

    def pop(self):
        """Remove and return the item with the smallest priority"""
        if self.is_empty():
            return "Priority Queue is empty"
        return heapq.heappop(self.heap)[1]  # return only the item

    def peek(self):
        """Return the item with the smallest priority without removing it"""
        if self.is_empty():
            return "Priority Queue is empty"
        return self.heap[0][1]

    def is_empty(self):
        """Check if the priority queue is empty"""
        return len(self.heap) == 0

    def display(self):
        """Display the internal heap (priority, item) pairs"""
        print(self.heap)
# Example usage
pq = PriorityQueue()
pq.push(2, "Task B")
pq.push(1, "Task A")
pq.push(3, "Task C")
pq.display()          # [(1, 'Task A'), (2, 'Task B'), (3, 'Task C')]
print(pq.peek())      # Task A (highest priority = lowest number)
print(pq.pop())       # Task A
print(pq.pop())       # Task B
print(pq.is_empty())  # False
print(pq.pop())       # Task C
print(pq.is_empty())  # True
```

```python
# Example usage
pq = PriorityQueue()
pq.push(2, "Task B")
pq.push(1, "Task A")
pq.push(3, "Task C")
pq.display()          # [(1, 'Task A'), (2, 'Task B'), (3, 'Task C')]
print(pq.peek())      # Task A (highest priority = lowest number)
print(pq.pop())       # Task A
print(pq.pop())       # Task B
print(pq.is_empty())  # False
print(pq.pop())       # Task C
print(pq.is_empty())  # True
```

Output:

```
PS C:\Users\ASHMITHA\Desktop\ai asss> & C:/Users/ASHMITHA/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/ASHMITHA/Desktop/ai asss/task7.py"
[(1, 'Task A'), (2, 'Task B'), (3, 'Task C')]
Task A
Task A
Task B
False
Task C
True
```

Observation:

The program implements a priority queue using Python's heapq module, which maintains elements in a min-heap structure. Each item is stored as a (priority, value) pair, where the element with the **lowest priority number** is always served first. When inserting tasks with priorities 2 (Task B), 1 (Task A), and 3 (Task C), the heap is internally arranged as [(1, 'Task A'), (2, 'Task B'), (3, 'Task C')]. The peek() method shows "Task A" as the highest-priority element without removing it. Successive pop() operations return "Task A", "Task B", and "Task C" in the correct priority order. The is_empty() method accurately detects when the queue becomes empty. This confirms that insertion, retrieval, and deletion all work as expected in the priority queue.

Task8:

Prompt:

Implement a double-ended queue using collections.deque.

Code:

task8.py > ...

```python
from collections import deque

class DequeExample:
    def __init__(self):
        self.deque = deque()

    def add_front(self, item):
        """Insert an item at the front of the deque"""
        self.deque.appendleft(item)

    def add_rear(self, item):
        """Insert an item at the rear of the deque"""
        self.deque.append(item)

    def remove_front(self):
        """Remove and return the front item"""
        if self.is_empty():
            return "Deque is empty"
        return self.deque.popleft()

    def remove_rear(self):
        """Remove and return the rear item"""
        if self.is_empty():
            return "Deque is empty"
        return self.deque.pop()

    def peek_front(self):
        """View the front item without removing it"""
        if self.is_empty():
            return "Deque is empty"
        return self.deque[0]

    def peek_rear(self):
        """View the rear item without removing it"""
        if self.is_empty():
            return "Deque is empty"
        return self.deque[-1]
```

```python

    def display(self):
        """Display the current deque"""
        print(list(self.deque))


# Example usage
dq = DequeExample()

dq.add_rear(10)
dq.add_rear(20)
dq.add_front(5)
dq.display()           # [5, 10, 20]

print(dq.remove_front())  # 5
print(dq.remove_rear())   # 20
dq.display()              # [10]

print(dq.peek_front())    # 10
print(dq.peek_rear())     # 10
print(dq.is_empty())      # False

```

Output:

```
PS C:\Users\ASHMITHA\Desktop\ai asss> & C:/Users/ASHMITHA/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/ASHMITHA/Desktop/ai asss/task8.py"
[5, 10, 20]
5
20
[10]
10
10
False
```

Observation:

The program uses Python's collections.deque to implement a double-ended queue that allows insertion and deletion of elements from both the front and the rear in constant time. When elements 10 and 20 are added at the rear and 5 at the front, the deque becomes [5, 10, 20]. The remove_front() operation deletes 5, and the remove_rear() operation deletes 20, leaving [10] in the deque. Both peek_front() and peek_rear() correctly show 10 as the only remaining element. The is_empty() method returns False, confirming that the deque still contains elements. This demonstrates that the deque supports efficient double-ended operations as expected.

Task9:

Prompt:

generate a comparison table of different data structures (stack, queue, linked list, etc.) including time complexities.

data structure comparison table:

Data Structure	**Insertion**	**Deletion**	**Access (by index)**	**Search**	**Remarks**
Stack (LIFO)	O(1) (push)	O(1) (pop)	O(n)	O(n)	Access is only from top; used for undo/recursion
Queue (FIFO)	O(1) (enqueue)	O(1) (dequeue)	O(n)	O(n)	Access only front/rear; used in scheduling
Deque (Double-ended queue)	O(1) (front/rear)	O(1) (front/rear)	O(n)	O(n)	Supports both stack & queue operations
Singly Linked List	O(1) (front), O(n) (end)	O(1) (front), O(n) (end)	O(n)	O(n)	Dynamic size, sequential access
Doubly Linked List	O(1) (front/end)	O(1) (front/end)	O(n)	O(n)	Can traverse in both directions
Binary Search Tree (BST)	O(log n) (avg), O(n) (worst)	O(log n) (avg), O(n) (worst)	O(log n) (avg), O(n) (worst)	O(log n) (avg), O(n) (worst)	Efficient if balanced, but can degrade
Balanced BST (AVL, Red-Black)	O(log n)	O(log n)	O(log n)	O(log n)	Always balanced for efficiency

Data Structure	Insertion	Deletion	Access (by index)	Search	Remarks
Hash Table	O(1) (avg), O(n) (worst with collisions)	O(1) (avg)	N/A	O(1) (avg)	Fast lookups, may require rehashing
Array	O(1) (end), O(n) (middle)	O(1) (end), O(n) (middle)	O(1)	O(n) (linear), O(log n) if sorted + binary search	Fixed size unless dynamic
Graph (Adjacency List)	O(1) (edge insert)	O(1) (edge remove, avg)	O(V)	O(V+E) (traversal)	Efficient for sparse graphs

Observation:

The table provides a clear side-by-side comparison of common data structures, showing their **time complexities for insertion, deletion, access, and search**, along with key remarks.

- **Stack** and **Queue** offer constant-time insertion and deletion but require linear time for accessing elements since only the top or front/rear is directly accessible.
- **Deque** combines the properties of stack and queue, supporting fast operations at both ends.
- **Linked Lists** (singly and doubly) allow dynamic sizing, with fast insertion/deletion at the front (and rear for doubly linked), but access by index requires traversal (O(n)).
- **BSTs** provide logarithmic-time operations on average, but can degrade to linear time in the worst case if unbalanced; balanced BSTs (AVL/Red-Black) maintain O(log n) efficiency.
- **Hash Tables** allow near-constant-time insertion, deletion, and search on average but may slow down due to collisions or rehashing.
- **Arrays** provide constant-time access by index but have slower insertions/deletions in the middle (O(n)), although appending at the end is O(1).

- **Graphs (Adjacency List)** efficiently store sparse graphs, allowing constant-time edge insertions and traversal in O(V+E).

Task10:

Prompt:

I want to create a Python program to simulate a cafeteria order system where students are served in the order they arrive. Use a Queue data structure to maintain the orders. The program should have methods to:

1. Add a student order.
2. Serve the next student in line.
3. Peek at the next student without removing them.
4. Check if the queue is empty.
5. Display all current orders.

Include docstrings for each method and the class. Also, provide at least 3 assert test cases to verify that the queue operations (enqueue, dequeue, peek) work correctly.

Code:

task10.py > ...

```python
class CafeteriaQueue:
    """
    A class to simulate a cafeteria order queue using FIFO principle.
    """

    def __init__(self):
        self.queue = []

    def add_order(self, student_name):
        """
        Add a student's order to the end of the queue.
        :param student_name: str
        """
        self.queue.append(student_name)

    def serve_order(self):
        """
        Serve the next order in the queue (FIFO).
        :return: str
        """
        if self.is_empty():
            return "No orders to serve."
        return self.queue.pop(0)

    def peek_next(self):
        """
        Peek at the next order without removing it.
        :return: str
        """
        if self.is_empty():
            return "No orders in the queue."
        return self.queue[0]
```

task10.py > ...

```python
class CafeteriaQueue:

    def is_empty(self):
        """
        Check if the queue is empty.
        :return: bool
        """
        return len(self.queue) == 0

    def display_orders(self):
        """
        Display all current orders in the queue.
        """
        return list(self.queue)


# Example Usage
if __name__ == "__main__":
    cafe_queue = CafeteriaQueue()

    # Adding orders
    cafe_queue.add_order("Alice")
    cafe_queue.add_order("Bob")
    cafe_queue.add_order("Charlie")

    print("Current Orders:", cafe_queue.display_orders())
    print("Next to Serve:", cafe_queue.peek_next())
    print("Serving Order:", cafe_queue.serve_order())
    print("Orders After Serving:", cafe_queue.display_orders())

    # Assert Test Cases
    assert cafe_queue.peek_next() == "Bob"
    assert cafe_queue.serve_order() == "Bob"
    assert cafe_queue.serve_order() == "Charlie"
    assert cafe_queue.is_empty() == True
```

Output:

```
PS C:\Users\ASHMITHA\Desktop\ai asss> & C:/Users/ASHMITHA/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/ASHMITHA/Desktop/ai asss/task10.py"
Current Orders: ['Alice', 'Bob', 'Charlie']
Next to Serve: Alice
Serving Order: Alice
Orders After Serving: ['Bob', 'Charlie']
PS C:\Users\ASHMITHA\Desktop\ai asss>
```

Observation:

The program implements a **FIFO queue** to manage student orders in a cafeteria. Orders are added using add_order(), ensuring that the first student to place an order is the first to be served. The peek_next() method correctly shows the next student without removing them from the queue, while serve_order() removes and returns the next student in line. After serving all students, the is_empty() method confirms that the queue is empty. The assert test cases validate that enqueue, dequeue, and peek operations function as expected, demonstrating that the queue reliably maintains the correct order of service.