

AI ASSISTED CODING

M.Rohith Reddy

2303A51111

BATCH – 03

24– 02 – 2026

ASSIGNMENT – 11.2

Lab – 11 : Data Structures with AI : Implementing Fundamental Structures.

Task – 01 : Stack Using AI Guidance.

Prompt : Generate a Python class implementation of a Stack data structure with push, pop, peek, and is_empty methods. Add proper docstrings, comments, and a small example demonstrating usage.

Code :

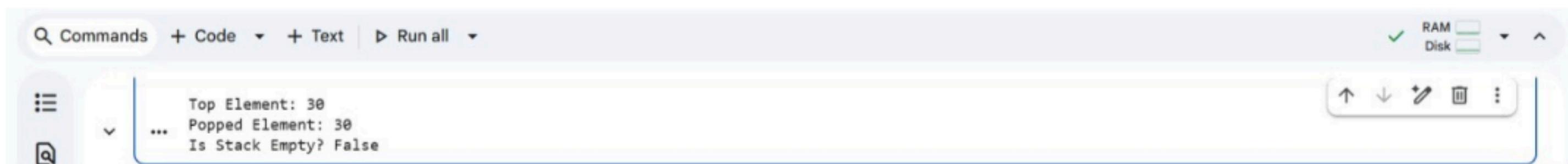


The screenshot shows a Python code editor interface. The code in the editor is:

```
class Stack:
    def __init__(self):
        self.items = []
    def push(self, item):
        self.items.append(item)
    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        return "Stack is Empty"
    def peek(self):
        if not self.is_empty():
            return self.items[-1]
        return "Stack is Empty"
    def is_empty(self):
        return len(self.items) == 0
s = Stack()
s.push(10)
s.push(20)
s.push(30)
print("Top Element:", s.peek())
print("Popped Element:", s.pop())
print("Is Stack Empty?", s.is_empty())
```

The status bar at the bottom right indicates the time is 1:20 PM and the Python version is Python 3.

Output:



The screenshot shows the terminal window of the code editor displaying the execution output:

```
Top Element: 30
Popped Element: 30
Is Stack Empty? False
```

Explanation :

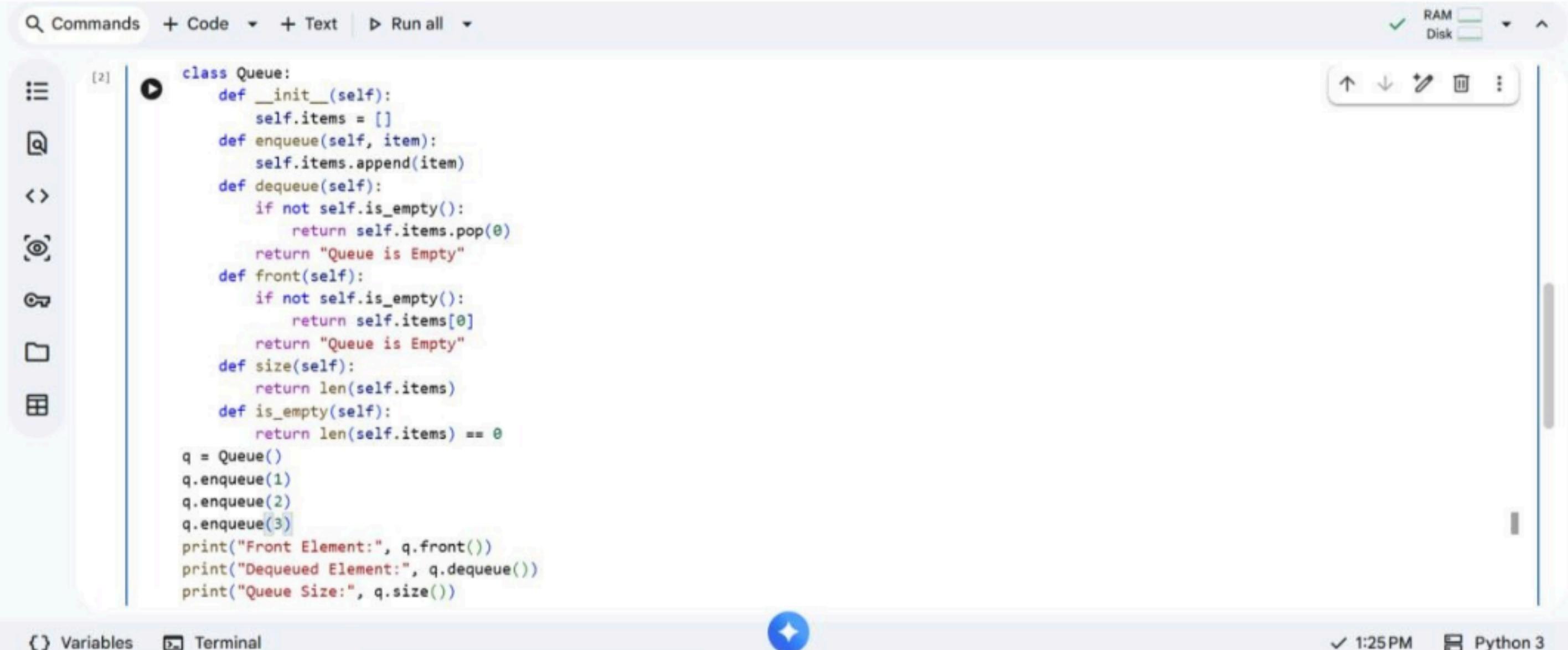
The Stack follows the LIFO (Last In First Out) principle. Elements are added using push() and removed using pop(). The peek() method returns the top

element without removing it, and `is_empty()` checks whether the stack contains elements.

Task - 02 : Queue Design.

Prompt : Create a Python Queue class implementing FIFO behaviour with enqueue, dequeue, front, and size methods. Include comments and sample usage.

Code :



The screenshot shows a Python code editor interface. The code defines a `Queue` class with methods for enqueueing, dequeuing, getting the front element, and checking the size. It also includes a small test block at the bottom. The status bar indicates the time as 1:25PM and the Python version as 3.

```
class Queue:
    def __init__(self):
        self.items = []
    def enqueue(self, item):
        self.items.append(item)
    def dequeue(self):
        if not self.is_empty():
            return self.items.pop(0)
        return "Queue is Empty"
    def front(self):
        if not self.is_empty():
            return self.items[0]
        return "Queue is Empty"
    def size(self):
        return len(self.items)
    def is_empty(self):
        return len(self.items) == 0
q = Queue()
q.enqueue(1)
q.enqueue(2)
q.enqueue(3)
print("Front Element:", q.front())
print("Dequeued Element:", q.dequeue())
print("Queue Size:", q.size())
```

Output :



The screenshot shows the output window of the code editor. It displays the results of running the queue code, which include the front element (1), the dequeued element (1), and the current queue size (2).

```
Front Element: 1
Dequeued Element: 1
Queue Size: 2
```

Explanation :

The Queue follows the FIFO (First In First Out) principle. Elements are inserted using `enqueue()` and removed using `dequeue()`. The `front()` method returns the first element, and `size()` gives the total number of elements.

Task - 03 : Singly Linked List Construction.

Prompt: Design a Singly Linked List in Python with a Node class, insertion at the end, and traversal/display functionality. Add comments explaining each part.

Code & Output :

Commands + Code + Text ▶ Run all ✓ RAM Disk

TASK 03

[3]

class Node:

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

class SinglyLinkedList:
    def __init__(self):
        self.head = None

    def insert(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            return
        temp = self.head
        while temp.next:
            temp = temp.next
        temp.next = new_node

    def display(self):
```

↑ ↓ ⚡ 🗑️ ⏮

Variables Terminal Python 3 ✓ 1:29 PM

The screenshot shows a Python code editor interface. On the left, there's a sidebar with various icons: a search bar, 'Commands' (dropdown), '+ Code' (dropdown), '+ Text' (dropdown), 'Run all' (dropdown), a RAM/Disk status indicator (green checkmark, RAM: 100%, Disk: 100%), and a toolbar with up/down arrows, a pencil, a refresh, and a three-dot menu. The main area has a code editor with a blue border. The code is as follows:

```
[3] def display(self):
    temp = self.head
    while temp:
        print(temp.data, end=" -> ")
        temp = temp.next
    print("None")

ll = SinglyLinkedList()
ll.insert(10)
ll.insert(20)
ll.insert(30)
print("Linked List:")
ll.display()
```

Below the code, the output is displayed in a light gray box:

```
*** Linked List:
10 -> 20 -> 30 -> None
```

Explanation :

A Singly Linked List consists of nodes where each node stores data and a reference to the next node. Insertion adds a new node at the end of the list. Traversal iterates through nodes sequentially to display all elements.

Task – 04 : Binary Search Tree Operations.

Prompt: Implement a Binary Search Tree in Python with insertion and in-order traversal methods. Include comments explaining how BST property is maintained.

Code & Output :

```
class BSTNode:
    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):
        if self.root is None:
            self.root = BSTNode(data)
        else:
            self._insert_recursive(self.root, data)
    def _insert_recursive(self, node, data):
        if data < node.data:
            if node.left is None:
                node.left = BSTNode(data)
            else:
                self._insert_recursive(node.left, data)
        else:
            if node.right is None:
```

```
def inorder(self):
    self._inorder_recursive(self.root)
def _inorder_recursive(self, node):
    if node:
        self._inorder_recursive(node.left)
        print(node.data, end=" ")
        self._inorder_recursive(node.right)
bst = BinarySearchTree()
bst.insert(50)
bst.insert(30)
bst.insert(70)
bst.insert(20)
bst.insert(40)
print("In-order Traversal:")
bst.inorder()
```

... In-order Traversal:
20 30 40 50 70

Explanation :

A Binary Search Tree maintains the property: Left child < Root < Right child. Insertion places elements according to this rule, and in-order traversal prints elements in sorted order.

Task - 05 : Hash Table Implementation.

Prompt : Create a Hash Table in Python using chaining for collision handling. Implement insert, search, and delete operations with comments and example usage.

Code & Output :

```

class HashTable:
    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(size)]

    def hash_function(self, key):
        return key % self.size

    def insert(self, key, value):
        index = self.hash_function(key)
        for pair in self.table[index]:
            if pair[0] == key:
                pair[1] = value
                return
        self.table[index].append([key, value])

    def search(self, key):
        index = self.hash_function(key)
        for pair in self.table[index]:
            if pair[0] == key:
                return pair[1]
        return "Key Not Found"

    def delete(self, key):
        index = self.hash_function(key)
        for i, pair in enumerate(self.table[index]):
            if pair[0] == key:
                self.table[index].pop(i)
                return "Deleted Successfully"
        return "Key Not Found"

ht = HashTable()
ht.insert(10, "Apple")
ht.insert(20, "Banana")
ht.insert(30, "Cherry")

print("Search 20:", ht.search(20))
print(ht.delete(20))
print("Search 20 after deletion:", ht.search(20))

```

Explanation :

A Hash Table stores data using a hash function to compute an index. Collisions are handled using chaining (linked lists at each index). It supports fast insertion, searching, and deletion operations.

Task : Over Flow and Under Flow.

Prompt : Generate a Python program to implement a fixed-size Stack with push, pop, peek, is_empty, and is_full methods. The program should display “Stack Overflow” when full and “Stack Underflow” when empty, with proper comments and example usage.

Code:

```
UNDER FLOW & OVER FLOW
```

```
[6] class Stack:
    def __init__(self, size):
        self.size = size
        self.stack = []
        self.top = -1
    def is_empty(self):
        return self.top == -1
    def is_full(self):
        return self.top == self.size - 1
    def push(self, value):
        if self.is_full():
            print("Stack Overflow! Cannot push", value)
        else:
            self.stack.append(value)
            self.top += 1
            print(value, "pushed into stack")
    def pop(self):
        if self.is_empty():
            print("Stack Underflow! Stack is empty")
        else:
            popped_value = self.stack.pop()
```

```
[6]     popped_value = self.stack.pop()
        self.top -= 1
        print(popped_value, "popped from stack")
    def peek(self):
        if self.is_empty():
            print("Stack is empty")
        else:
            print("Top element is:", self.stack[self.top])
    def display(self):
        if self.is_empty():
            print("Stack is empty")
        else:
            print("Stack elements:", self.stack)
s = Stack(3)
s.push(10)
s.push(20)
s.push(30)
s.push(40)
s.display()
s.pop()
s.pop()
s.pop()
s.pop()
```

Output :

```
10 pushed into stack
20 pushed into stack
30 pushed into stack
Stack Overflow! Cannot push 40
Stack elements: [10, 20, 30]
30 popped from stack
20 popped from stack
10 popped from stack
Stack Underflow! Stack is empty
```

Explanation : The program implements a fixed-size Stack following the LIFO

(Last In First

Out) principle using a list and a top pointer. The push() method checks if the stack is full and displays “Stack Overflow”, while pop() checks if it is empty and displays “Stack Underflow”. Helper methods like is_empty() and is_full() ensure proper boundary checking and safe stack operations.