Assignment-11.2

Name: Gampa Rithika
Roll No: 2303A51190
Batch:03

Task Description -1 – (Stack Using AI Guidance)

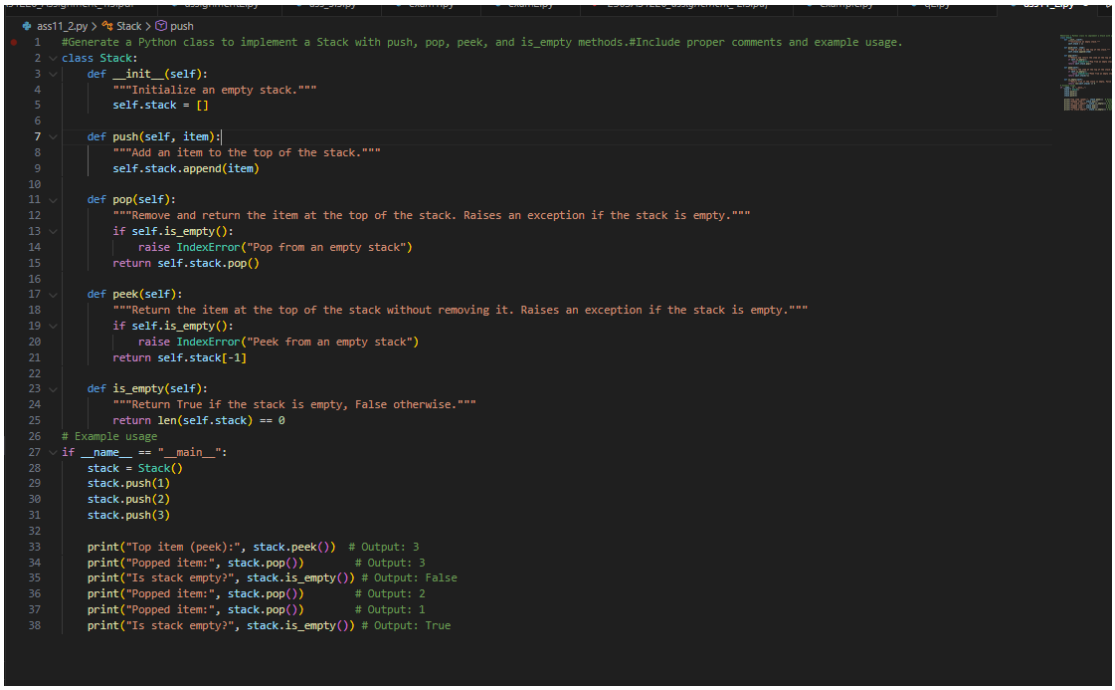
• Task: With the help of AI, design and implement a Stack data structure supporting basic stack operations.

Expected Output:

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

Prompt:

Generate a Python class to implement a Stack with push, pop, peek, and is_empty methods.Include proper comments and example usage.



```
ass11_2.py > Stack > push
#Generate a Python class to implement a Stack with push, pop, peek, and is_empty methods.#Include proper comments and example usage.
class Stack:
    def __init__(self):
        """Initialize an empty stack."""
        self.stack = []

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

    def pop(self):
        """Remove and return the item at the top of the stack. Raises an exception if the stack is empty."""
        if self.is_empty():
            raise IndexError("Pop from an empty stack")
        return self.stack.pop()

    def peek(self):
        """Return the item at the top of the stack without removing it. Raises an exception if the stack is empty."""
        if self.is_empty():
            raise IndexError("Peek from an empty stack")
        return self.stack[-1]

    def is_empty(self):
        """Return True if the stack is empty, False otherwise."""
        return len(self.stack) == 0
# Example usage
if __name__ == "__main__":
    stack = Stack()
    stack.push(1)
    stack.push(2)
    stack.push(3)

    print("Top item (peek):", stack.peek())  # Output: 3
    print("Popped item:", stack.pop())       # Output: 3
    print("Is stack empty?", stack.is_empty()) # Output: False
    print("Popped item:", stack.pop())       # Output: 2
    print("Popped item:", stack.pop())       # Output: 1
    print("Is stack empty?", stack.is_empty()) # Output: True
```

OUTPUT:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS C:\Users\shyam_3jvctb4\OneDrive\Documents\AI Assistant Coding> & C:/Users/shyam_3jvctb4/AppData/Local/Microsoft/WindowsA
Documents/AI Assistant Coding/ass11_2.py"
Top item (peek): 3
Popped item: 3
Is stack empty? False
Popped item: 2
Popped item: 1
Is stack empty? True
PS C:\Users\shyam_3jvctb4\OneDrive\Documents\AI Assistant Coding>
```

Code Explanation:

This program implements a Stack using a Python list. The push() method adds elements to the top, pop() removes the top element, peek() shows the top element without removing it, and is_empty() checks if the stack is empty. The example demonstrates that the stack follows the LIFO (Last In First Out) principle.

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:

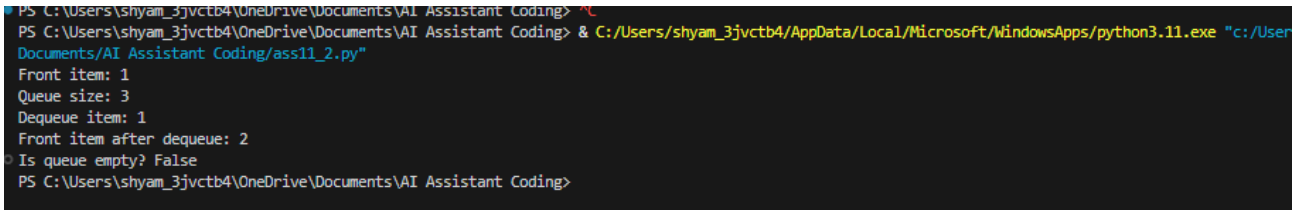
Create a Python Queue class following FIFO with enqueue, dequeue, front, size, and is_empty methods including comments, example usage, and time complexity.

```python
#Create a Python Queue class following FIFO with enqueue, dequeue, front, size, and is_empty methods including comments, example usage, and time complexity.
class Queue:
    def __init__(self):
        """Initialize an empty queue."""
        self.items = []
    def enqueue(self, item):
        """Add an item to the end of the queue.
        Time Complexity: O(1)
        """
        self.items.append(item)
    def dequeue(self):
        """Remove and return the item at the front of the queue.
        Time Complexity: O(n) due to list shifting after removal.
        """
        if not self.is_empty():
            return self.items.pop(0)
        else:
            raise IndexError("Dequeue from an empty queue")
    def front(self):
        """Return the item at the front of the queue without removing it.
        Time Complexity: O(1)
        """
        if not self.is_empty():
            return self.items[0]
        else:
            raise IndexError("Front from an empty queue")
    def size(self):
        """Return the number of items in the queue.
        Time Complexity: O(1)
        """
        return len(self.items)
    def is_empty(self):
        """Check if the queue is empty.
        Time Complexity: O(1)
        """
        return len(self.items) == 0
# Example usage
if __name__ == "__main__":
    q = Queue()
    q.enqueue(1)
    q.enqueue(2)
    q.enqueue(3)

    print("Front item:", q.front())  # Output: Front item: 1
    print("Queue size:", q.size())    # Output: Queue size: 3

    print("Dequeue item:", q.dequeue())  # Output: Dequeue item: 1
    print("Front item after dequeue:", q.front())  # Output: Front item after dequeue: 2
    print("Is queue empty?", q.is_empty())  # Output: Is queue empty? False
```

OUTPUT:



Code Explanation:

This program implements a Queue using a Python list following the FIFO (First In First Out) principle. The enqueue() method adds elements to the end of the queue, while dequeue() removes elements from the front. The front() method returns the first element without removing it, size() returns the total number of elements, and is_empty() checks whether the queue is empty. The example shows that the first inserted element is removed first, confirming FIFO behavior.

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:

Implement a Singly Linked List in Python with Node class, insert_at_end, and display methods including comments and example usage.

```python
#Implement a Singly Linked List in Python with Node class, insert_at_end, and display methods including comments and example
class Node:
    """A class representing a node in a singly linked list."""
    def __init__(self, data):
        self.data = data  # The data stored in the node
        self.next = None  # Pointer to the next node in the list
class SinglyLinkedList:
    """A class representing a singly linked list."""
    def __init__(self):
        self.head = None  # Initialize the head of the list to None

    def insert_at_end(self, data):
        """Inserts a new node with the given data at the end of the list."""
        new_node = Node(data)  # Create a new node with the provided data
        if self.head is None:
            self.head = new_node  # If the list is empty, set head to the new node
            return
        last_node = self.head
        while last_node.next:  # Traverse to the end of the list
            last_node = last_node.next
        last_node.next = new_node  # Link the last node to the new node

    def display(self):
        """Displays the contents of the list."""
        current_node = self.head
        while current_node:
            print(current_node.data, end=' -> ')  # Print the data of each node
            current_node = current_node.next
        print("None")  # Indicate the end of the list
# Example usage
if __name__ == "__main__":
    linked_list = SinglyLinkedList()  # Create a new singly linked list
    linked_list.insert_at_end(10)  # Insert nodes at the end of the list
    linked_list.insert_at_end(20)
    linked_list.insert_at_end(30)

    print("Contents of the linked list:")
    linked_list.display()  # Display the contents of the list
    # Output should be: 10 -> 20 -> 30 -> None
```

OUTPUT:

```
PS C:\Users\shyam_3jvctb4\OneDrive\Documents\AI Assistant Coding> & C:/Use
Documents/AI Assistant Coding/ass11_2.py"
Contents of the linked list:
10 -> 20 -> 30 -> None
PS C:\Users\shyam_3jvctb4\OneDrive\Documents\AI Assistant Coding>
```

Code Explanation:

This program implements a Singly Linked List using a Node class and a SinglyLinkedList class. The Node class stores the data and a reference (next) to the next node. The insert_at_end() method adds a new node at the end by traversing the list until the last node and linking the new node. The display() method prints each node's data followed by None, showing the end of the list. The example demonstrates inserting elements (10, 20, 30) and displaying them in order.

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

Prompt:

Develop a Binary Search Tree in Python with insert and inorder traversal methods including comments, example usage, and time complexity.

```python
#Implement a Binary Search Tree in Python with insert and in-order traversal methods.Provide comments and example execution.
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key
class BinarySearchTree:
    def __init__(self):
        self.root = None

    def insert(self, key):
        if self.root is None:
            self.root = Node(key)
        else:
            self._insert_recursively(self.root, key)
    def _insert_recursively(self, node, key):
        if key < node.val:
            if node.left is None:
                node.left = Node(key)
            else:
                self._insert_recursively(node.left, key)
        else:
            if node.right is None:
                node.right = Node(key)
            else:
                self._insert_recursively(node.right, key)
    def in_order_traversal(self):
        return self._in_order_recursively(self.root)
    def _in_order_recursively(self, node):
        res = []
        if node:
            res = self._in_order_recursively(node.left)
            res.append(node.val)
            res = res + self._in_order_recursively(node.right)
        return res
# Example execution
if __name__ == "__main__":
    bst = BinarySearchTree()
    bst.insert(50)
    bst.insert(30)
    bst.insert(20)
    bst.insert(40)
    bst.insert(70)
    bst.insert(60)
    bst.insert(80)

    print("In-order traversal of the binary search tree:")
    print(bst.in_order_traversal())
    # Output should be: [20, 30, 40, 50, 60, 70, 80]
```

OUTPUT:

```
PS C:\Users\shyam_3jvctb4\OneDrive\Documents\AI Assistant Coding> & C:/Users/shyam_3jvctb4/AppData/Local
Documents/AI Assistant Coding/ass11_2.py"
In-order traversal of the binary search tree:
[20, 30, 40, 50, 60, 70, 80]
PS C:\Users\shyam_3jvctb4\OneDrive\Documents\AI Assistant Coding>
```

Code Explanation:

This program implements a Binary Search Tree (BST) using a Node class and a BinarySearchTree class. The insert() method adds elements according to the BST property (left child < parent < right child) using recursion. The in_order_traversal() method visits nodes in left-root-right order, which returns the elements in sorted order. The example inserts multiple values and prints them in ascending order, confirming correct BST behavior.

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:

Create a Hash Table in Python using chaining for collision handling with insert, search, and delete methods including comments and example usage.

```python
#Create a Hash Table in Python using chaining for collision handling with insert, search, and delete methods including comments and example usage.
class HashTable:
    def __init__(self, size=10):
        """Initialize the hash table with a specified size."""
        self.size = size
        self.table = [[] for _ in range(size)]  # Create a list of empty lists for chaining

    def _hash(self, key):
        """Generate a hash for the given key."""
        return hash(key) % self.size

    def insert(self, key, value):
        """Insert a key-value pair into the hash table."""
        index = self._hash(key)
        # Check if the key already exists and update it
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                self.table[index][i] = (key, value)  # Update existing key
                return
        # If key does not exist, add new key-value pair
        self.table[index].append((key, value))

    def search(self, key):
        """Search for a value by its key in the hash table."""
        index = self._hash(key)
        for k, v in self.table[index]:
            if k == key:
                return v  # Return the value if found
        return None  # Return None if key is not found

    def delete(self, key):
        """Delete a key-value pair from the hash table."""
        index = self._hash(key)
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                del self.table[index][i]  # Remove the key-value pair
                return True  # Return True if deletion was successful
        return False  # Return False if key was not found
# Example usage
if __name__ == "__main__":
    hash_table = HashTable()

    # Insert key-value pairs
    hash_table.insert("name", "Alice")
    hash_table.insert("age", 30)
    hash_table.insert("city", "New York")

    # Search for values
    print(hash_table.search("name"))  # Output: Alice
    print(hash_table.search("age"))   # Output: 30
    print(hash_table.search("country"))  # Output: None
```

OUTPUT:

```
PS C:\Users\shyam_3jvctb4\OneDrive\Documents\AI Assistant Coding> & C:/Users/shyam_3jvctb4/
Documents/AI Assistant Coding/ass11_2.py"
Alice
30
None
True
None
False
PS C:\Users\shyam_3jvctb4\OneDrive\Documents\AI Assistant Coding>
```

Code Explanation:

This program implements a Hash Table using chaining for collision handling. The _hash() method generates an index using Python's built-in hash() function. The insert() method adds key-value pairs and updates the value if the key already exists. The search() method looks for a key in the corresponding bucket and returns its value, while the delete() method removes the key-value pair if found. Chaining is implemented using a list of lists, allowing multiple elements to be stored at the same index in case of collisions.