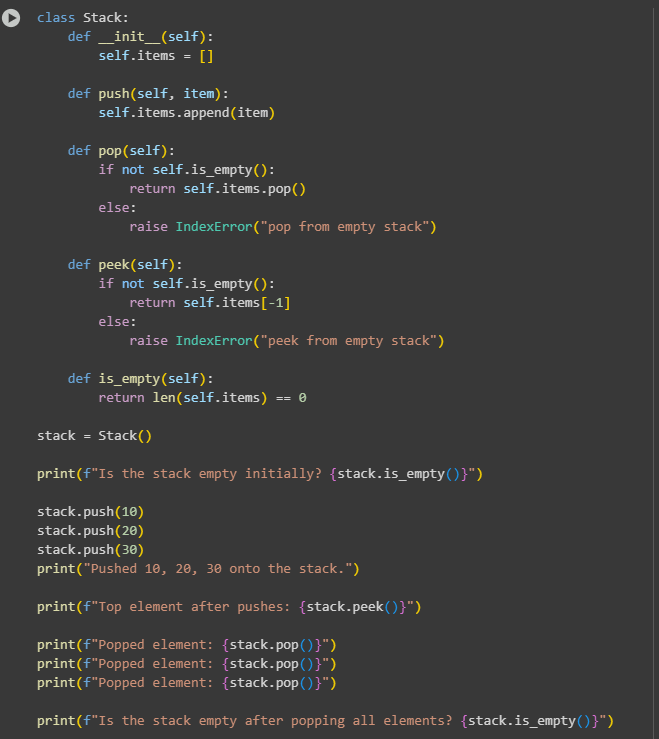
ASSISGNMENT-11

2403A52111

TASK-1:

PROMPT:Implement a Stack class in Python with push(), pop(), peek(), and is\_empty() methods. Create a working Stack class, test its operations, and discuss alternative implementations.

CODE:



A screen shot of a computer program

AI-generated content may be incorrect.

EXPLANATION:

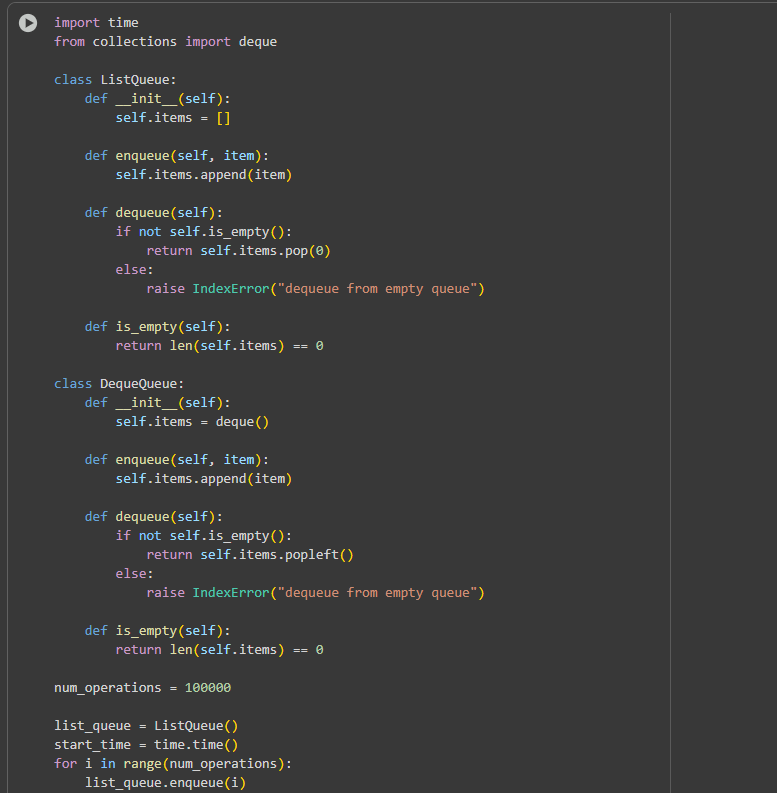
**Core Components**

* **\_\_init\_\_(self)**: This is the constructor method. It initializes the stack by creating an empty list, self.\_items, which will hold the elements of the stack. The leading underscore (\_) in \_items is a Python convention to suggest that this attribute should be treated as internal to the class.
* **is\_empty(self)**: This method checks if the stack is empty. It returns True if the \_items list is empty, and False otherwise. The expression not self.\_items is a concise way to do this, as an empty list evaluates to False in a boolean context.
* **push(self, item)**: This method adds an item to the top of the stack. It uses the list's built-in **append()** method, which adds the new element to the end of the list. This operation is highly efficient, with an amortized time complexity of O(1).
* **pop(self)**: This method removes and returns the item at the top of the stack. It first checks if the stack is empty to prevent an error. If the stack is not empty, it uses the list's **pop()** method, which removes and returns the last element. This operation is also very efficient, with a time complexity of O(1).
* **peek(self)**: This method allows you to look at the top item of the stack without removing it. It first checks if the stack is empty. If not, it returns the last element of the list using the index [-1], which is a common Python shortcut for accessing the final element.
* **size(self)**: This simple utility method returns the current number of items in the stack by using the len() function on the \_items list.

TASK-2:

PROMPT:Implement a Queue with enqueue(), dequeue(), and is\_empty() methods. Implement the queue first using a Python list and then with collections.deque. Compare their performance and explain the difference.

CODE:



A screenshot of a computer program

AI-generated content may be incorrect.

EXPLANATION OF THE CODE:

The QueueList class uses a Python list as its data container.

* **enqueue()**: This method adds an element to the end of the list using self.items.append(item). This is an efficient operation, with a time complexity of **O(1)** (constant time).
* **dequeue()**: This method removes the first element from the queue using self.items.pop(0). This operation is **inefficient** because it has a time complexity of **O(n)**, where *n* is the number of elements in the list. This is because when the first element is removed, all subsequent elements in the list must be shifted one position to the left to fill the gap.

**Queue Implementation with collections.deque**

The QueueDeque class uses the **collections.deque** data structure, which is specifically designed to handle queue-like operations efficiently.

* **enqueue()**: This method adds an element to the right side of the deque using self.items.append(item). This is an **O(1)** operation.
* **dequeue()**: This method removes an element from the left side of the deque using self.items.popleft(). This is also an **O(1)** operation. The deque is implemented as a doubly-linked list, allowing for fast additions and removals from both ends without having to shift other elements.

**Task**-3:

Prompt: Implement a Singly Linked List with insert\_at\_end(), delete\_value(), and traverse() methods.Create a functional linked list, explain key pointer updates with comments, and provide test cases to validate the code.

Code:

A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

EXPLANATION:

1. **Node Class:**
   * This is a simple class that represents a single element (node) in the linked list.
   * Each Node object has two attributes:
     + data: Stores the actual value or data for that node.
     + next: A pointer (reference) to the next node in the sequence. It's initialized to None for the last node in the list or when a node is first created.
2. **SinglyLinkedList Class:**
   * This class represents the entire linked list.
   * It has one main attribute:
     + head: A pointer to the first node in the list. If the list is empty, head is None.
   * It includes three key methods:
     + **insert\_at\_end(self, data):** This method adds a new node containing the given data to the very end of the list. It first checks if the list is empty; if so, the new node becomes the head. Otherwise, it traverses the list until it reaches the last node (the one whose next is None) and updates that node's next pointer to point to the new node.
     + **delete\_value(self, value):** This method removes the *first* occurrence of a node with the specified value. It handles two cases:
       - If the head node contains the value, the head pointer is simply moved to the next node.
       - If the value is in a node further down the list, it traverses the list with two pointers: current (pointing to the node being checked) and prev (pointing to the node before current). When the node with the matching value is found, the prev node's next pointer is updated to skip the current node, effectively removing it from the list.
     + **traverse(self):** This method iterates through the linked list starting from the head and prints the data of each node. It uses -> to visually represent the links between nodes and ends with -> None to indicate the end of the list. If the list is empty, it prints "List is empty."

**TASK-**4:

PROMPT: Implement a Binary Search Tree with insert(), search(), and inorder\_traversal() methods.Provide a complete BST class, fill in missing methods, and test the functionality with sample data.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

EXPLANATION:

1. **Node Class**:
   * Represents an individual element within the linked list.
   * \_\_init\_\_(self, data=None): The constructor initializes a node with optional data and sets the next pointer to None by default. The next pointer will point to the next node in the sequence.
2. **SinglyLinkedList Class**:
   * Represents the entire linked list.
   * \_\_init\_\_(self): The constructor initializes an empty linked list by setting the head attribute to None. The head attribute always points to the first node in the list.
   * **insert\_at\_end(self, data)**: This method adds a new node containing the provided data to the end of the linked list.
     + It creates a new\_node.
     + If the list is empty (self.head is None), the new\_node becomes the head.
     + Otherwise, it traverses the list starting from the head until it reaches the last node (the node whose next is None).
     + It then updates the next pointer of the last node to point to the new\_node.
   * **delete\_value(self, value)**: This method removes the *first* node it finds that contains the specified value.
     + It first checks if the head node contains the value. If it does, it updates the head pointer to the next node, effectively removing the original head.
     + If the value is not in the head, it traverses the list using two pointers: current (which starts at the head and moves through the list) and prev (which trails one node behind current).
     + When the current node's data matches the value, the prev node's next pointer is updated to point to current.next, bypassing and removing the current node from the list.
     + If the loop finishes without finding the value, the method simply returns.
   * **traverse(self)**: This method iterates through the linked list from the head to the end and prints the data of each node.
     + If the list is empty, it prints "List is empty."
     + Otherwise, it starts from the head and follows the next pointers, printing each node's data and an arrow -> until it reaches the end (None). It finishes by printing None to indicate the end of the list.

These methods provide the fundamental operations for managing a singly linked list: adding elements, removing elements, and viewing the elements in order.

TASK-5:

PROMPT: Implement a graph using an adjacency list and create BFS() and DFS() traversal methods.Provide a working graph class, explain the logic of both traversal algorithms, and compare the recursive vs. iterative approaches for DFS.

CODE: A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

EXPLANATION:

* **raph class:** This is the blueprint for creating graph objects.
* **\_\_init\_\_(self):** This is the constructor. It initializes an empty dictionary called adj\_list which will store the adjacency list. The keys of the dictionary are the nodes, and the values are lists of their neighbors.
* **add\_edge(self, u, v):** This method adds an edge between two nodes u and v. Since this is an undirected graph, it adds v to u's list of neighbors and u to v's list of neighbors.
* **bfs(self, start\_node):** This method performs a Breadth-First Search starting from start\_node. It uses a queue to explore the graph level by level, ensuring that all nodes at a certain distance from the start node are visited before moving to nodes at the next distance. It also uses a visited set to keep track of nodes that have already been visited to avoid infinite loops.
* **dfs\_recursive(self, start\_node, visited=None):** This method performs a Depth-First Search recursively starting from start\_node. It explores as far as possible along each branch before backtracking. It uses a visited set to keep track of visited nodes.
* **dfs\_iterative(self, start\_node):** This method performs a Depth-First Search iteratively starting from start\_node using a stack. It achieves the same result as the recursive version but uses a stack to manage the nodes to visit.
* **Sample Graph and Testing:** The code then creates an instance of the Graph class, adds some edges to create a sample graph, and finally calls the bfs, dfs\_recursive, and dfs\_iterative methods to demonstrate the traversal algorithms.