AI ASSISTED CODING

LAB ASSIGNMENT – 11

**NAME :** YUVARAJ REGULA

**HALL.NO :** 2403A52386

**BATCH.NO :** AIML-14

**Prompt 01**:

Write a simple Python Stack class with push, pop, peek, and is\_empty methods. Add short comments and docstrings.

**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.***

**Code explanation:**

* class Stack: This line defines a new class named Stack.
* "A simple Stack class implementation."""This is a docstring that explains the purpose of the class.
* def \_\_init\_\_(self):: This is the constructor method. It's called when you create a new Stack object.
  + self.\_items = []: Inside the constructor, an empty list named \_items is initialized. This list will be used internally to store the elements of the stack. The underscore prefix (\_) is a convention to indicate that this is an internal detail of the class.
* def push(self, item): This method adds an item to the top of the stack.
  + self.\_items.append(item): The append() method of the list adds the item to the end of the \_items list, which is considered the "top" of the stack in this implementation.
* def pop(self):: This method removes and returns the item from the top of the stack.
  + if not self.is\_empty():: It first checks if the stack is not empty using the is\_empty() method.
  + return self.\_items.pop(): If the stack is not empty, the pop() method of the list is used to remove and return the last element (the top of the stack).
  + else: raise IndexError("pop from empty stack"): If the stack is empty, it raises an IndexError with a descriptive message.
* def peek(self):: This method returns the item at the top of the stack without removing it.
  + if not self.is\_empty():: It also checks if the stack is not empty.
  + return self.\_items[-1]: If not empty, it returns the last element of the list (\_items[-1]) using negative indexing, which accesses the last element.
  + else: raise IndexError("peek from empty stack"): If empty, it raises an IndexError.
* def is\_empty(self): This method checks if the stack is empty.
  + return len(self.\_items) == 0: It returns True if the length of the \_items list is 0, and False otherwise.
* def size(self):: This method returns the number of items in the stack.
  + return len(self.\_items): It simply returns the length of the \_items list.

**Performing with example :A screenshot of a computer program

AI-generated content may be incorrect.A screen shot of a computer program

AI-generated content may be incorrect.**

**Prompt 02:**

Create a Python Queue class using lists with enqueue, dequeue, and is\_empty methods. Then suggest a faster version using collections. Deque.

**Code:**

A screenshot of a computer program

AI-generated content may be incorrect.A computer screen shot of text

AI-generated content may be incorrect.

**Code explanation:  
class Queue:**: This line defines a new class named Queue.

* **"""A simple Queue class implementation using a list."""**: This is a docstring explaining the class's purpose and implementation detail (using a list).
* **def \_\_init\_\_(self):**: This is the constructor.
  + **self.\_items = []**: An empty list \_items is initialized to hold the queue elements.
* **def enqueue(self, item):**: This method adds an item to the back (rear) of the queue.
  + **self.\_items.append(item)**: The append() method is used to add the item to the end of the list. This is an efficient operation (O(1) on average).
* **def dequeue(self):**: This method removes and returns the item from the front of the queue.
  + **if not self.is\_empty():**: It checks if the queue is not empty.
  + **return self.\_items.pop(0)**: If not empty, pop(0) is used to remove and return the element at index 0 (the front of the list). **However, this operation can be inefficient (O(n))** because all subsequent elements need to be shifted to the left to fill the gap.
  + **else: raise IndexError("dequeue from empty queue")**: If the queue is empty, it raises an IndexError.
* **def is\_empty(self):**: This method checks if the queue is empty by checking if the length of the \_items list is 0.

**Performing with example :A screenshot of a computer program

AI-generated content may be incorrect.A screenshot of a computer program

AI-generated content may be incorrect.**

**Prompt 03:**

Make a Python singly linked list with insert\_at\_end, delete\_value, and traverse methods. Add comments to explain pointer change*s.*

**Code:A screen shot of a computer program

AI-generated content may be incorrect.A screenshot of a computer code

AI-generated content may be incorrect.**

**Code explanation:**

The Node Class:

* class Node:: This defines the building block of the linked list. Each Node object will hold a piece of data and a reference to the next node in the sequence.
* def \_\_init\_\_(self, data):: This is the constructor for the Node.
  + self.data = data: This line stores the actual data that the node will hold.
  + self.next = None: This is the crucial part for linking. self.next is a pointer (or reference) to the next node in the list. When a node is initially created, it doesn't point to anything yet, so it's set to None.

2. The SinglyLinkedList Class:

* class SinglyLinkedList:: This class represents the entire linked list. It manages the sequence of nodes.
* def \_\_init\_\_(self):: This is the constructor for the SinglyLinkedList.
  + self.head = None: This initializes the head of the list to None. The head is a pointer to the very first node in the list. An empty list has no head.
* def insert\_at\_end(self, data):: This method adds a new node with the given data to the end of the list.
  + new\_node = Node(data): A new Node object is created with the provided data.
  + if not self.head:: This checks if the list is currently empty.
    - self.head = new\_node: If the list is empty, the new node becomes the head.
    - return: The method finishes here if the list was empty.
  + last\_node = self.head: If the list is not empty, a temporary pointer last\_node is set to the current head.
  + while last\_node.next:: This loop traverses the list until last\_node points to the last node (the one whose next pointer is None).
    - last\_node = last\_node.next: The last\_node pointer moves to the next node in each iteration.
  + last\_node.next = new\_node: Once the loop finishes, last\_node is at the end. Its next pointer is updated to point to the new\_node, effectively adding the new node to the end of the list.
* def delete\_value(self, value):: This method deletes the first node that contains the given value.
  + if not self.head:: Checks if the list is empty. If so, there's nothing to delete.
  + if self.head.data == value:: Checks if the head node contains the value to be deleted.
    - self.head = self.head.next: If the head contains the value, the head pointer is updated to point to the next node, effectively removing the original head.
    - return: The method finishes after deleting the head.
  + current\_node = self.head: If the value is not in the head, a current\_node pointer is initialized to the head.
  + while current\_node.next and current\_node.next.data != value:: This loop traverses the list until either the next node exists and its data matches the value, or the end of the list is reached.
    - current\_node = current\_node.next: The current\_node pointer moves to the next node.
  + if current\_node.next:: After the loop, this checks if a node with the value was found (i.e., current\_node.next is not None).
    - current\_node.next = current\_node.next.next: If the node was found, the next pointer of the node before the one to be deleted (current\_node) is updated to point to the node after the one to be deleted (current\_node.next.next). This bypasses the node to be deleted, removing it from the list's sequence.
* def traverse(self):: This method goes through the list from beginning to end and prints the data of each node.
  + current\_node = self.head: A current\_node pointer starts at the head.
  + while current\_node:: This loop continues as long as current\_node is not None (meaning there are still nodes to visit).
    - print(current\_node.data, end=" -> "): The data of the current node is printed, followed by " -> ". end=" -> " prevents a newline after each print, keeping the output on one line.
    - current\_node = current\_node.next: The current\_node pointer moves to the next node.
  + print("None"): After the loop finishes (when current\_node becomes None), "None" is printed to indicate the end of the list.

**Performing with example:**

**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.**

**Prompt 04:**

Write a Python BST with insert, search, and inorder\_traversal methods. Add comments for recursion and traversal.

**Code&Output: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 code

AI-generated content may be incorrect.**

**Code explanation:**

1. **Node Class:** This class represents a single node within the BST. Each node has:
   * key: The value stored in the node.
   * left: A reference to the left child node.
   * right: A reference to the right child node.
2. **BST Class:** This class represents the entire Binary Search Tree.
   * \_\_init\_\_(self): The constructor initializes an empty BST by setting the root to None.
   * insert(self, key): This method inserts a new node with the given key into the BST. It handles the case of an empty tree and then calls a recursive helper method \_insert\_recursive.
   * \_insert\_recursive(self, current\_node, key): This is a recursive helper function for insertion. It compares the key to the current\_node's key and moves down the left or right subtree until it finds an appropriate place to insert the new node.
   * search(self, key): This method searches for a node with the given key in the BST. It calls a recursive helper method \_search\_recursive.
   * \_search\_recursive(self, current\_node, key): This is a recursive helper function for searching. It checks if the current\_node is the target node or if the target is in the left or right subtree and recursively searches accordingly.
   * inorder\_traversal(self): This method performs an in-order traversal of the BST, which visits nodes in ascending order of their keys. It calls a recursive helper method \_inorder\_recursive.
   * \_inorder\_recursive(self, current\_node, result): This is a recursive helper function for in-order traversal. It visits the left subtree, then the current node, and then the right subtree, appending the node's key to the result list.
3. **Example Usage:** The code then demonstrates how to use the BST class by:
   * Creating a BST instance.
   * Inserting several values into the tree.
   * Performing an in-order traversal and printing the result.
   * Searching for existing and non-existing keys and printing the results.

**Prompt 05:**

Implement a Python graph using an adjacency list with BFS and DFS methods. Add short comments to explain how each traversal works.

**Code& Output:A screenshot of a computer program

AI-generated content may be incorrect.**

**A screen shot of a computer program

AI-generated content may be incorrect.**

**Code explanation:**

1. **Graph Class:**
   * \_\_init\_\_(self): The constructor initializes an empty graph using defaultdict(list). This creates a dictionary where keys are nodes and values are lists of their neighbors. If a node is accessed that isn't already in the dictionary, a new entry is created with an empty list as its value.
   * add\_edge(self, u, v): This method adds a directed edge from node u to node v. It appends v to the list of neighbors for u. The commented-out line self.graph[v].append(u) would be included for an undirected graph to add the edge in both directions.
2. **bfs(self, start\_node):**
   * This method performs a Breadth-First Search starting from the start\_node.
   * It uses a deque (double-ended queue) to manage the nodes to visit. BFS explores the graph level by level.
   * visited is a set to keep track of nodes that have already been visited to avoid infinite loops.
   * The while queue: loop continues as long as there are nodes to visit.
   * queue.popleft() removes the node at the beginning of the queue (FIFO - First-In, First-Out).
   * It then iterates through the neighbors of the current node and adds unvisited neighbors to the visited set and the queue.
   * The result list stores the nodes in the order they are visited during the traversal.
3. **dfs(self, start\_node):**
   * This method performs a Depth-First Search starting from the start\_node.
   * It uses recursion (implicitly through function calls) to explore as deep as possible along each branch before backtracking.
   * visited is a set to keep track of visited nodes.
   * The inner function \_dfs\_recursive(node) is the recursive helper.
   * It marks the current node as visited and adds it to the result list.
   * Then, it iterates through the neighbors of the current node. If a neighbor hasn't been visited, it recursively calls \_dfs\_recursive on that neighbor.
4. **Example Usage:** The code demonstrates how to create a Graph instance, add edges to define the graph structure, and then perform BFS and DFS starting from a specific node (node 2 in this case), printing the order of visited nodes for each traversal.