LAB ASSIGNMENT 11.3

< AI ASSISTED CODING >

Program : B. Tech (CSE)

Specialization : AIML

Course Title : AI Assisted coding

Semester : III

Academic Session : 2025-2026

Name of Student : Vangapandla Koushik

Enrollment No : 2403A52004

Batch No : 01

Date : 08-10-2025

* ACTIVITY

Task Description #1 – Stack class implementation

Task: Ask AI to implement a stack class with push(), pop(), peek() and

is\_empty() methods.

Prompt: Create python class implementation for a stack data structure, including methods for adding elements (push), removing elements (pop), viewing the top element without removing it (peek), and checking if the stack is empty (is\_empty)?"

Code :

A computer screen shot of a program code

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

A computer screen shot of a program

AI-generated content may be incorrect.

Observation :

* Implementation: The stack is implemented using a Python list (self.items). Lists are a suitable choice for this as they provide dynamic resizing and efficient append/pop operations from the end.
* LIFO Principle: The push and pop methods correctly follow the Last-In, First-Out (LIFO) principle that defines a stack. push adds to the end of the list, and pop removes from the end.
* Error Handling: The pop and peek methods include basic error handling to check if the stack is empty before attempting an operation that would fail. This prevents potential IndexError exceptions.
* is\_empty Method: The is\_empty method provides a clear and simple way to check the state of the stack by checking the length of the underlying list.
* User Interaction: The example usage includes a while loop and input() calls to allow the user to interact with the stack by choosing actions and providing input. This makes the code easy to test and understand.
* Clear Output: The methods include print statements to show the action being performed (e.g., "Pushed: item", "Popped: item"), which is helpful for demonstrating the stack's behavior.

Task Description #2 – Queue Implementation

Task: Use AI to generate a Queue class with enqueue(), dequeue(), and

is\_empty().

Prompt : create a Python code for a Queue data structure, including functions to add elements to the rear (enqueue), remove elements from the front (dequeue), and determine if the queue is empty (is\_empty)?"

Code :

A computer code with many different colored text

AI-generated content may be incorrect.

A computer code with red text

AI-generated content may be incorrect.

Output :

A white background with black text

AI-generated content may be incorrect.

Observation :

* Implementation: Like the Stack, the Queue is implemented using a Python list (self.items).
* FIFO Principle: The enqueue and dequeue methods correctly follow the First-In, First-Out (FIFO) principle that defines a queue. enqueue adds to the end of the list, while dequeue removes from the beginning of the list using pop(0).
* pop(0) Efficiency: While using pop(0) on a Python list works for a queue, it's important to note that removing from the beginning of a list can be less efficient for very large lists compared to adding and removing from the end (as in the Stack). For performance-critical applications with large queues, using Python's collections.deque might be a better choice as it's optimized for adding and removing from both ends.
* Error Handling: Similar to the Stack, the dequeue method includes error handling to check if the queue is empty before attempting to remove an item.
* is\_empty Method: The is\_empty method is a simple and effective way to check if the queue contains any elements.
* User Interaction: The example usage provides a clear way for the user to interact with the queue and test its methods through input prompts.
* Clear Output: The print statements in the methods help visualize the process of enqueuing and dequeuing items.

Task Description #3 – Linked List Implementation

Task: Ask AI to create a singly linked list with insert\_at\_end(),

insert\_at\_beginning(), and display().

Prompt : create Python code for a singly linked list data structure that includes functions to add nodes at the end (insert\_at\_end), add nodes at the beginning (insert\_at\_beginning), and print the elements of the list (display)?".

Code :

A screenshot of a computer code

AI-generated content may be incorrect.

A computer code with text

AI-generated content may be incorrect.

A computer screen shot of a program

AI-generated content may be incorrect.

Output :

A white text with black text

AI-generated content may be incorrect.

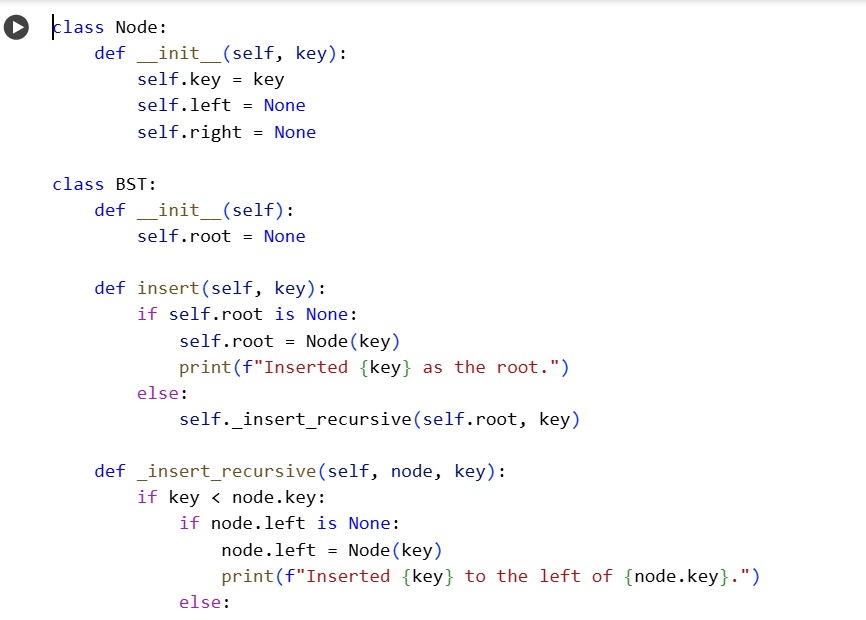
Observation :

* Node Class: The code correctly defines a Node class as a building block for the linked list. Each node stores data and a reference to the next node in the sequence.
* Singly Linked: The implementation is for a singly linked list because each node only has a reference to the next node, not the previous one.
* Head Pointer: The SinglyLinkedList class maintains a head pointer, which references the first node in the list. This is the entry point for traversing or modifying the list.
* insert\_at\_beginning: This method is efficient. It creates a new node, sets its next pointer to the current head, and then updates the list's head to point to the new node. This takes constant time (O(1)).
* insert\_at\_end: This method requires traversing the list from the head to the last node to append the new node. In the worst case (a long list), this takes time proportional to the number of nodes (O(n)). An edge case for an empty list is handled correctly.
* display: This method also requires traversing the list from the head to collect and print the data of each node. This operation takes time proportional to the number of nodes (O(n)).
* User Interaction: The example usage with the while loop and input() allows for easy testing and demonstration of the linked list's functionality.
* Clear Output: The print statements in the methods provide feedback on the operations being performed.

Task Description #4 – Binary Search Tree (BST)  
Task: Ask AI to generate a simple BST with insert() and  
inorder\_traversal().

Prompt : create Python code for a basic Binary Search Tree (BST) data structure, including a method to insert new nodes (insert) and a method to perform an in-order traversal of the tree (inorder\_traversal)?"

Code :



A computer screen shot of a code

AI-generated content may be incorrect.

A computer screen shot of a program

AI-generated content may be incorrect.

Output :

A white background with black text

AI-generated content may be incorrect.

Observation :

* Node Class: The code defines a Node class to represent the nodes in the tree. Each node stores a key (the value) and references to its left and right children.
* Root Node: The BST class maintains a root pointer, which is the starting point of the tree.
* insert Method: The insert method handles the logic for adding new nodes while maintaining the BST property (left child key < parent key < right child key). It uses a recursive helper function \_insert\_recursive to traverse the tree and find the correct position for the new key.
* Recursive Insertion: The \_insert\_recursive function demonstrates the recursive nature of inserting into a BST. It compares the new key with the current node's key to decide whether to go left or right until an empty spot is found.
* Handling Duplicates: The code includes a check to see if the key already exists in the tree and prints a message if it does, preventing duplicate keys.
* inorder\_traversal Method: The inorder\_traversal method performs an in-order traversal of the tree, which visits the left subtree, then the current node, and then the right subtree. This traversal method is commonly used for BSTs because it visits the nodes in ascending order of their keys
* Recursive Traversal: The \_inorder\_recursive function is a recursive helper that implements the in-order traversal logic.