NAME : DARIPELLI SRIYA

ENROLLMENT NO : 2403A51350

BATCH NO: 01

SUBJECT: AI ASSISTANT CODING

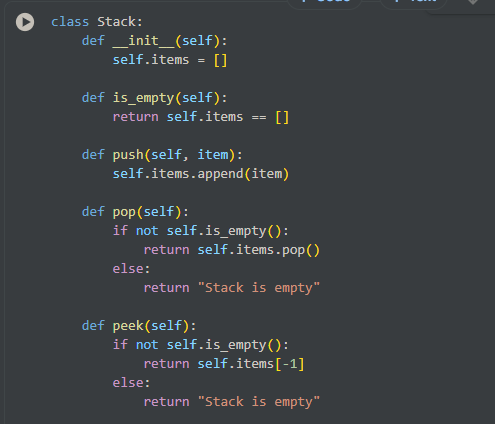
ASSIGNMENT: 11.3

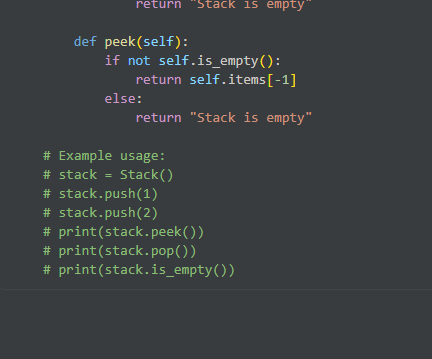
Task Description #1 – Stack class implementation  
Task: Ask AI to implement a stack class with push(), pop(), peek() and  
is\_empty() methods

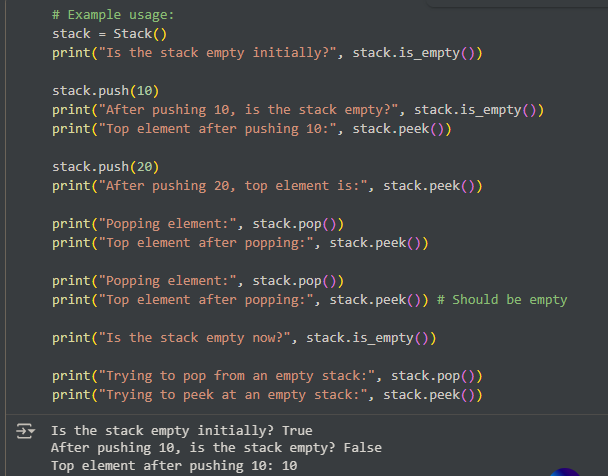
Prompt:

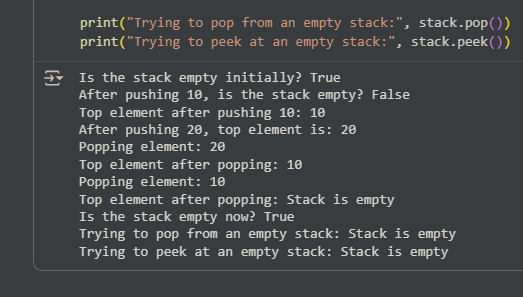
Implement a stack class with push(), pop(), peek() and  
is\_empty() methods

SCREENSHOT:









EXPLAINATION:

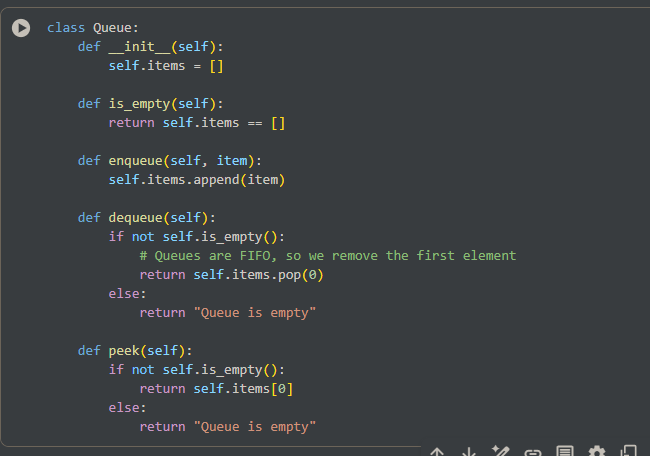
* \_\_init\_\_(self): This is the constructor of the class. It initializes an empty list called self.items to store the elements of the stack.
* is\_empty(self): This method checks if the stack is empty by comparing self.items to an empty list. It returns True if the stack is empty, and False otherwise.
* push(self, item): This method adds an item to the top of the stack by appending it to the self.items list.
* pop(self): This method removes and returns the item from the top of the stack. It first checks if the stack is empty using is\_empty(). If not empty, it uses the pop() method of the list to remove and return the last element (which is the top of the stack). If the stack is empty, it returns the string "Stack is empty".
* peek(self): This method returns the item at the top of the stack without removing it. It also checks if the stack is empty. If not empty, it returns the last element of the self.items list using indexing [-1]. If the stack is empty, it returns "Stack is empty".

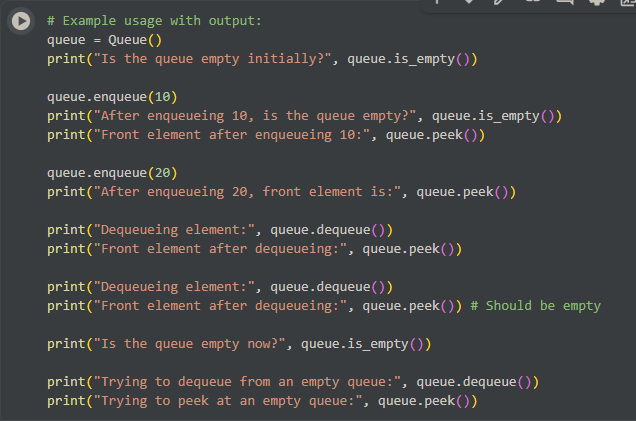
Task Description #2 – Queue Implementation  
Task: Use AI to generate a Queue class with enqueue(), dequeue(), and  
is\_empty()

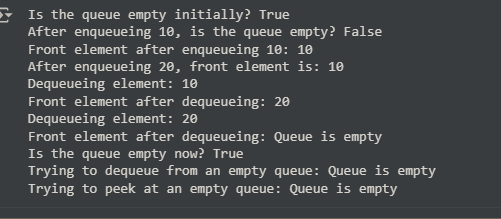
PROMPT:

generate a Queue class with enqueue(), dequeue(), and  
is\_empty() with output

SCREENSHOTS:







EXPLAINATION:

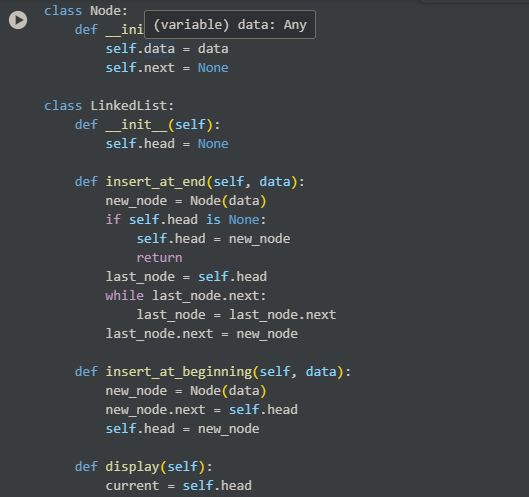
* \_\_init\_\_(self): This is the constructor of the class. It initializes an empty list called self.items to store the elements of the queue.
* is\_empty(self): This method checks if the queue is empty by comparing self.items to an empty list. It returns True if the queue is empty, and False otherwise.
* enqueue(self, item): This method adds an item to the end of the queue by appending it to the self.items list.
* dequeue(self): This method removes and returns the item from the front of the queue. It first checks if the queue is empty using is\_empty(). If not empty, it uses the pop(0) method of the list to remove and return the first element (which is the front of the queue). If the queue is empty, it returns the string "Queue is empty".
* peek(self): This method returns the item at the front of the queue without removing it. It also checks if the queue is empty. If not empty, it returns the first element of the self.items list using indexing [0]. If the queue is empty, it returns "Queue is empty".

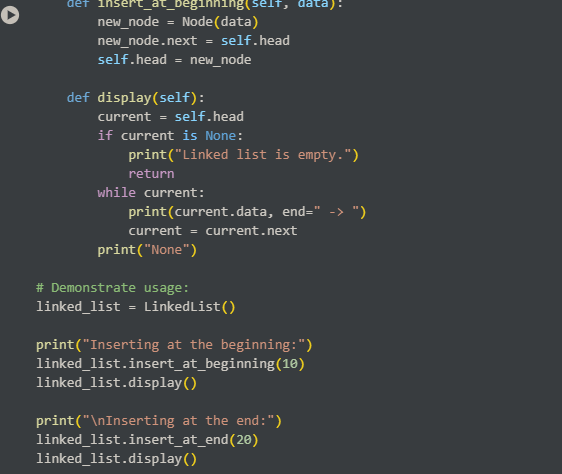
Task Description #3 – Linked List Implementation  
Task: Ask AI to create a singly linked list with insert\_at\_end(),  
insert\_at\_beginning(), and display()

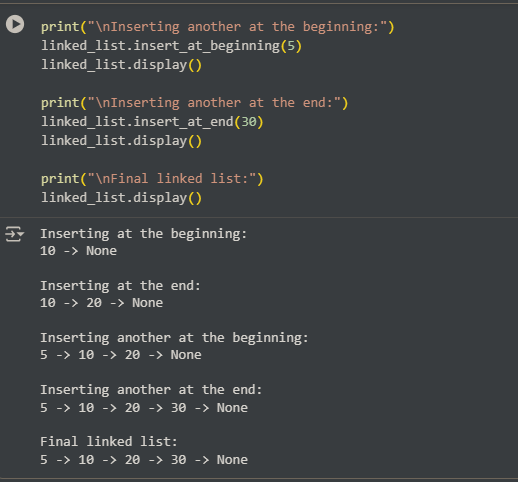
PROMPT:

create a singly linked list with insert\_at\_end(),  
insert\_at\_beginning(), and display() with output

SCREENSHOTS:







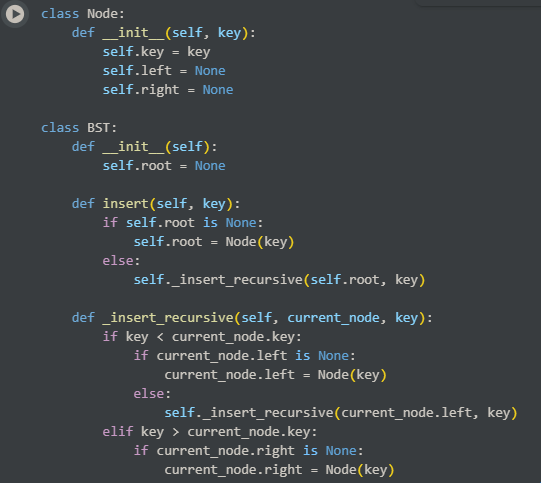
EXPLAINATION:

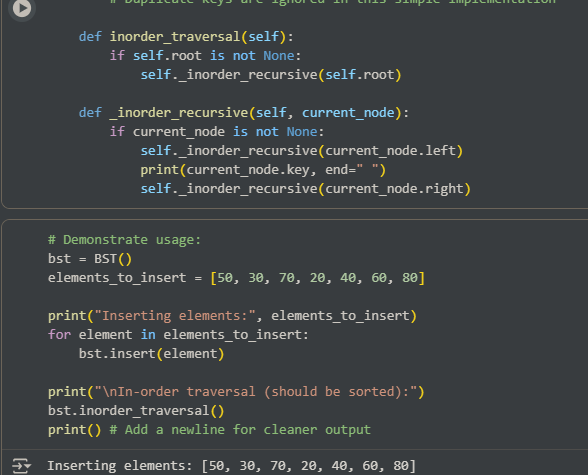
* **Node Class:**
  + This class represents an individual element (a node) within the linked list.
  + \_\_init\_\_(self, data): The constructor for a Node. It takes data as an argument and stores it in self.data. It also initializes self.next to None, indicating that this node doesn't point to another node yet.
* **LinkedList Class:**
  + This class represents the entire linked list.
  + \_\_init\_\_(self): The constructor for the LinkedList. It initializes self.head to None. The head is a reference to the first node in the list. An empty list has no head, hence None.
  + insert\_at\_end(self, data): This method adds a new node with the given data to the end of the linked list.
    - It creates a new\_node instance.
    - If the list is empty (self.head is None), the new\_node becomes the head.
    - If the list is not empty, it traverses the list starting from the head until it finds the last node (the one whose next is None).
    - It then sets the next of the last node to the new\_node.
  + insert\_at\_beginning(self, data): This method adds a new node with the given data to the beginning of the linked list.
    - It creates a new\_node instance.
    - It sets the next of the new\_node to the current self.head.
    - It then updates self.head to point to the new\_node, making it the new first node.
  + display(self): This method prints the data in each node of the linked list from the head to the end.
    - It starts from the self.head.
    - If the list is empty, it prints "Linked list is empty."
    - If not empty, it iterates through the nodes, printing the data of each node followed by " -> ".
    - Finally, it prints "None" to indicate the end of the list.

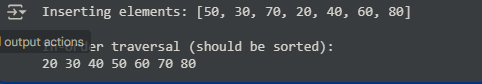
Task Description #4 – Binary Search Tree (BST)  
Task: Ask AI to generate a simple BST with insert() and  
inorder\_traversal()

PROMPT:

generate a simple BST with insert() and  
inorder\_traversal() with output







Explaination:

* **Node Class:**
  + \_\_init\_\_(self, key): Represents a single node in the BST. It stores a key (the value of the node) and has pointers left and right initialized to None for its child nodes.
* **BST Class:**
  + \_\_init\_\_(self): The constructor for the BST. It initializes the root of the tree to None, indicating an empty tree.
  + insert(self, key): This is the public method to insert a new key into the BST.
    - If the tree is empty (self.root is None), the new key becomes the root.
    - Otherwise, it calls the private helper method \_insert\_recursive to find the correct position for the new key.
  + \_insert\_recursive(self, current\_node, key): This is a recursive helper method for inserting a key.
    - It compares the key to the current\_node.key.
    - If the key is less than the current\_node.key, it goes to the left child. If the left child is None, it creates a new Node there. Otherwise, it recursively calls \_insert\_recursive on the left child.
    - If the key is greater than the current\_node.key, it goes to the right child following similar logic.
    - Duplicate keys are ignored in this implementation.
  + inorder\_traversal(self): This is the public method to perform an in-order traversal of the BST.
    - If the tree is not empty, it calls the private helper method \_inorder\_recursive.
  + \_inorder\_recursive(self, current\_node): This is a recursive helper method for the in-order traversal.
    - It follows the pattern: visit left subtree, visit the current node, visit right subtree.
    - It recursively calls itself on the left child, then prints the current\_node.key, and finally recursively calls itself on the right child. This traversal method visits nodes in ascending order of their keys in a BST.
* It creates a BST object.
* It defines a list of elements\_to\_insert.
* It then iterates through this list and calls the insert() method for each element to build the BST.
* Finally, it calls the inorder\_traversal() method, which prints the elements of the tree in sorted order, demonstrating that the insert() method maintained the BST property.