# K.R. Mangalam University

## School of Engineering & Technology

### DATA STRUCTURE

**Lab File**

**Submitted by:** Anshika Trikha

**Roll No:** 2401420044

**Course:** B.Tech CSE (Data Science)

**Submitted to:** Dr. Swati Gupta

### INDEX

|  |  |
| --- | --- |
| **S.No.** | **Experiment Title** |
| 1 | Inventory Stock Management System |
| 2 | Ticketing System Using Queue (Linear Queue Implementation) |
| 3 | Browser History Navigation System (Using Stack Concept) |
| 4 | Circular Singly Linked List (Insert, Search, Delete, Display) |
| 5 | Reverse a String Using Stack |
| 6 | Check Balanced Parentheses Using Stack |

### LAB ASSIGNMENT – 1

**Inventory Stock Management System**

Problem Statement-

Design and implement an Inventory Stock Management System that stores product information such as SKU, product name, and quantity. The system should allow the user to insert new products, update existing quantities, search products, delete records, display all available stock, and perform basic stock analysis such as calculating the total stock, average stock, and identifying the item with maximum quantity.

Objective-

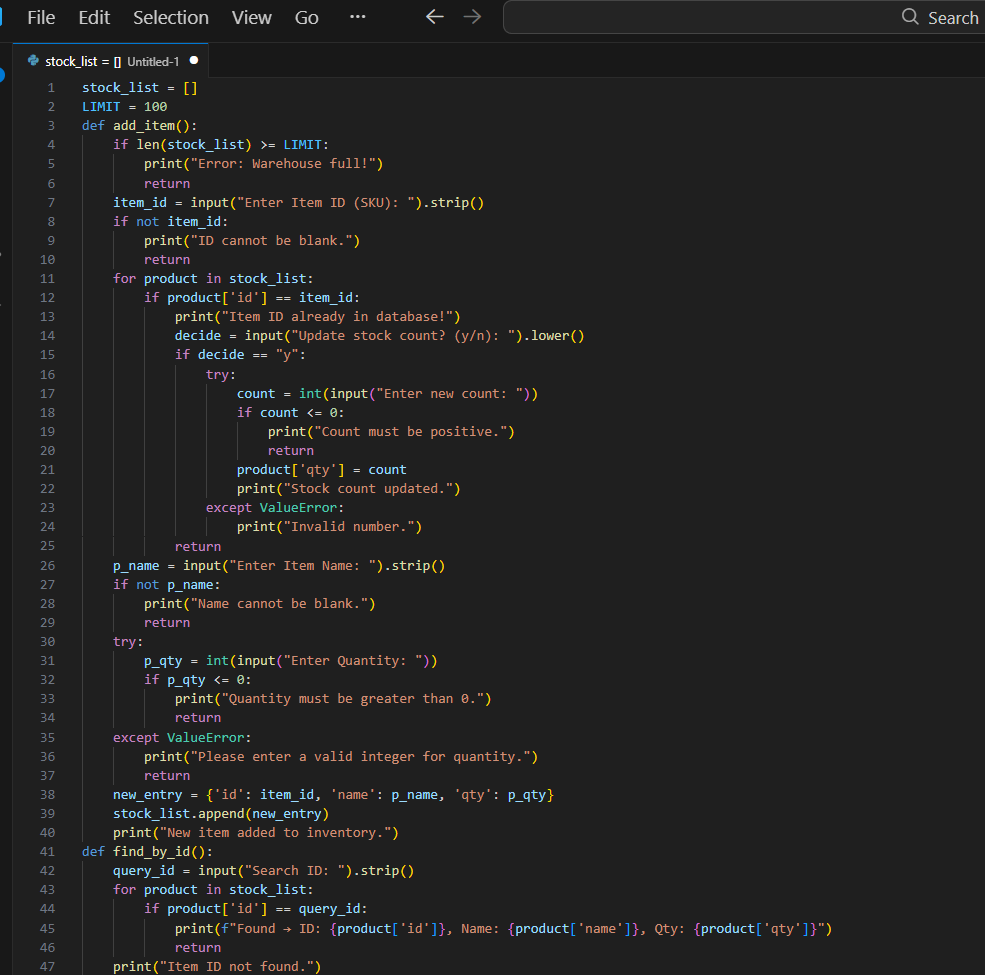
The objectives of this experiment are:

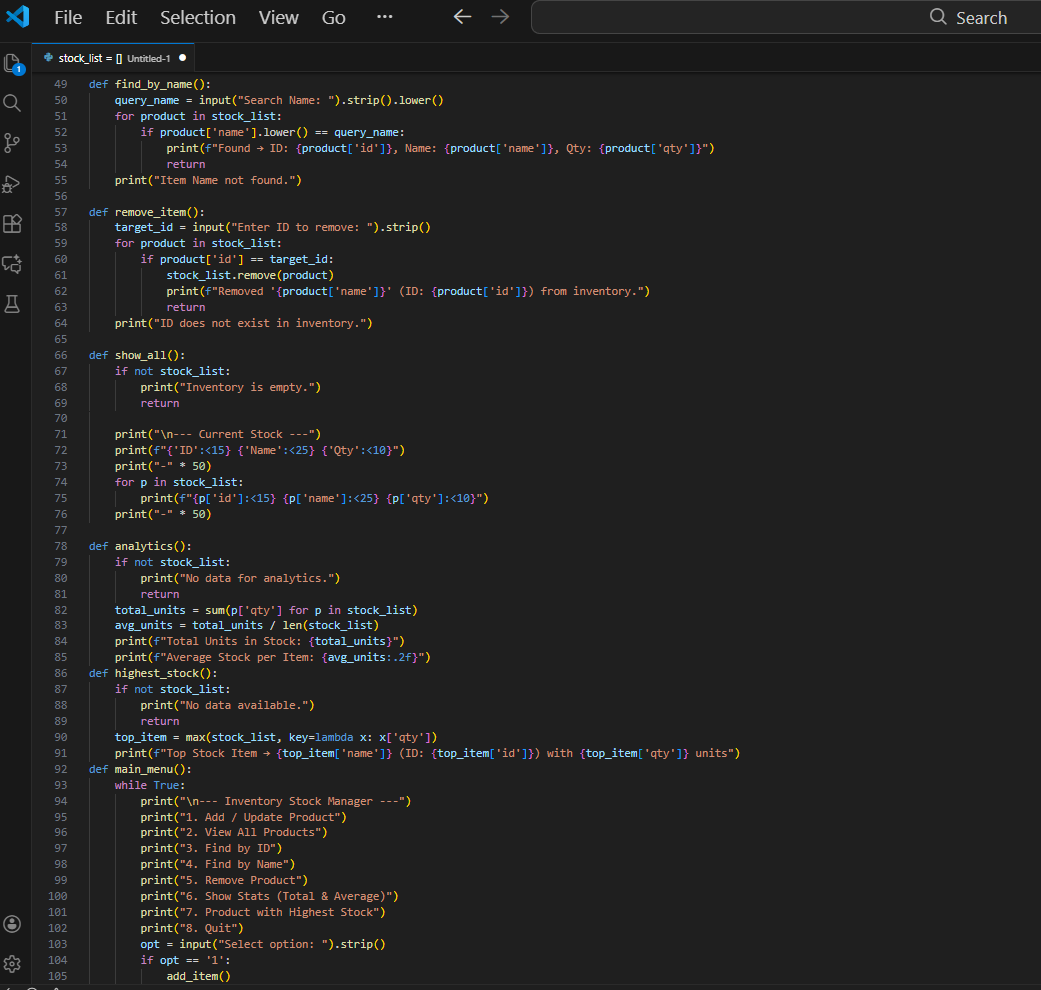
* To understand the use of lists, dictionaries, and functions in Python.
* To perform CRUD operations (Create, Read, Update, Delete) on inventory records.
* To implement searching techniques using SKU and Product Name.
* To apply data validation while inserting and updating stock information.
* To calculate total stock and average stock of all items.
* To identify the product that has the maximum stock in the inventory.
* To improve modular programming skills using user-defined functions.
* To develop a menu-driven Python program for real-world inventory management.

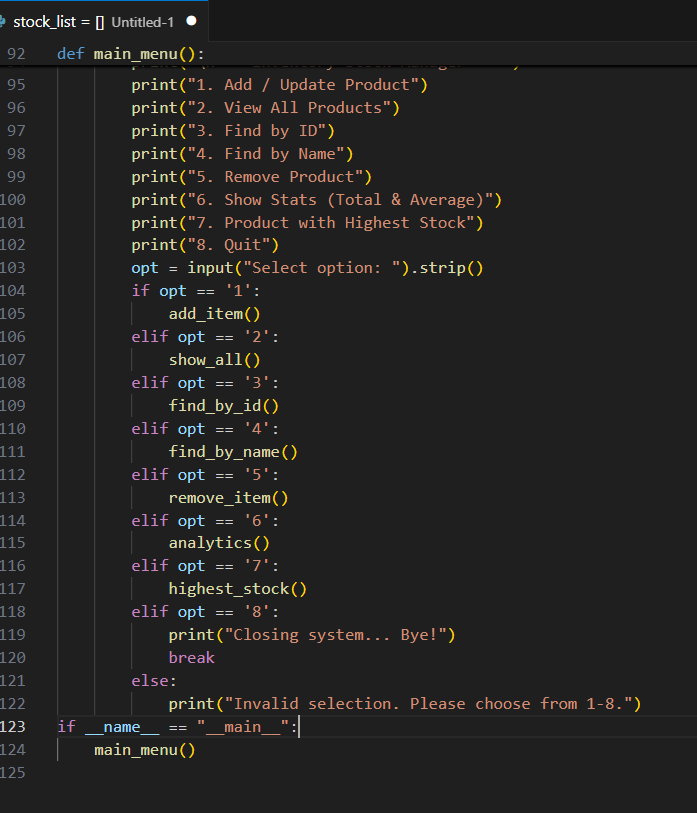
Description-

The Inventory Stock Management System is a menu-driven Python program designed to manage product data such as SKU, product name, and quantity. The program stores all product records in a list of dictionaries, allowing efficient access and modification. It provides multiple operations including inserting new products, updating stock when duplicate SKUs are found, searching for items using SKU or product name, deleting product entries, and displaying the complete inventory in tabular format. The system also includes additional analytical features such as calculating the total quantity of all products, computing the average stock per item, and identifying the product with the highest stock value. Proper data validation, such as checking for empty names, invalid quantities, and exceeding maximum capacity, ensures accurate and reliable inventory management. Overall, the program demonstrates the use of data structures, modular programming, and real-world problem-solving techniques in Python.

**INPUT CODE-**

****

****

****

**OUTPUT-**

****

Conclusion-

In this experiment, a complete Inventory Stock Management System was successfully developed using Python. The program allows users to efficiently manage product records through various operations including insertion, searching, deletion, and display. The additional analytical functions for total stock, average stock, and maximum-stock item enhance the practical utility of the system. This assignment reinforces the understanding of lists, dictionaries, user-defined functions, and menu-driven programming. Overall, the experiment provides hands-on experience in implementing real-world data handling and improves the ability to design structured, modular, and interactive Python applications.

### LAB ASSIGNMENT – 2

**Ticketing System Using Queue (Linear Queue Implementation)**

Problem Statement-

Design and implement a Ticketing System using the concept of a Linear Queue in Python. The system should allow users to generate tickets, serve customers in a first-come-first-served manner, display the current queue, and indicate whether the queue is empty or full. The program must use basic queue operations such as enqueue, dequeue, and display.

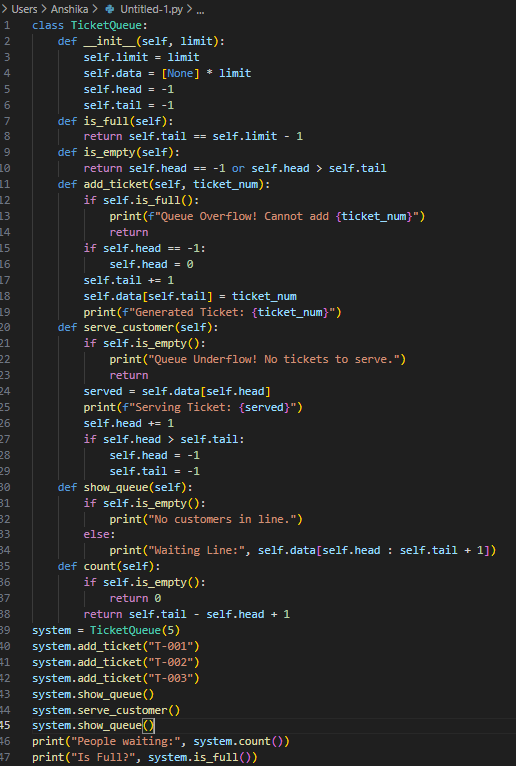
**Objective-**

* To understand the concept of the Queue data structure.
* To apply the FIFO (First-In First-Out) principle in programming.
* To implement queue operations such as enqueue, dequeue, and display.
* To identify overflow and underflow conditions in linear queues.
* To practice modular and object-oriented programming using classes.

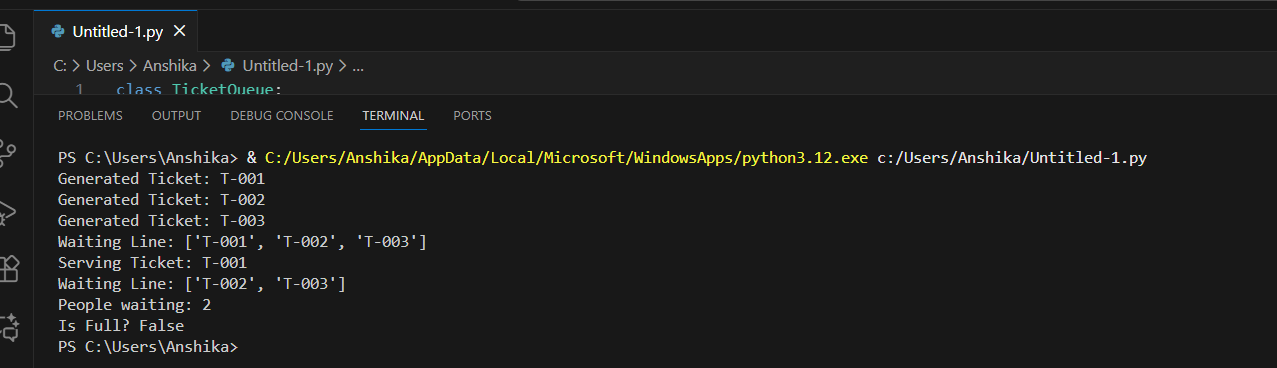
Description-

A queue is a linear data structure that follows the FIFO (First-In First-Out) principle. This assignment implements a simple Ticketing System where customers wait in line to be served. The system uses a Python list to store ticket numbers, and two pointers (front and rear) are maintained to manage queue operations. The user can generate a new ticket (enqueue), serve the next customer (dequeue), and display all customers currently waiting in queue. The program also checks and handles situations where the queue becomes full (overflow) or empty (underflow). This assignment demonstrates practical implementation of queue operations in a real-life scenario.

**INPUT CODE-**

****

OUTPUT –



Conclusion-

The assignment demonstrates how queue operations work and how FIFO behavior applies to real-world ticketing processes. The exercise strengthens understanding of queue operations such as insertion, deletion, overflow, underflow, and queue traversal. Overall, the program enhances object-oriented coding skills and provides practical experience in implementing fundamental data structures.

### LAB ASSIGNMENT – 3

**Browser History Navigation System (Using Stack Concept)**

Problem Statement-

Design and implement a Browser History Navigation System using the Stack data structure in Python. The system should allow users to visit pages, go back to previously visited pages, move forward, and display the browsing history. The implementation must demonstrate stack operations such as push and pop for backward and forward navigation.

**Objective-**

* To understand the concept of Stack and LIFO (Last-In First-Out) principle.
* To implement stack operations such as push and pop in Python.
* To simulate real-world browser navigation using Back and Forward functionality.
* To maintain two stacks to handle backward and forward navigation.
* To design a menu-driven program demonstrating stack-based history handling.
* To improve ability to solve real-world navigation problems using data structures.

Description-

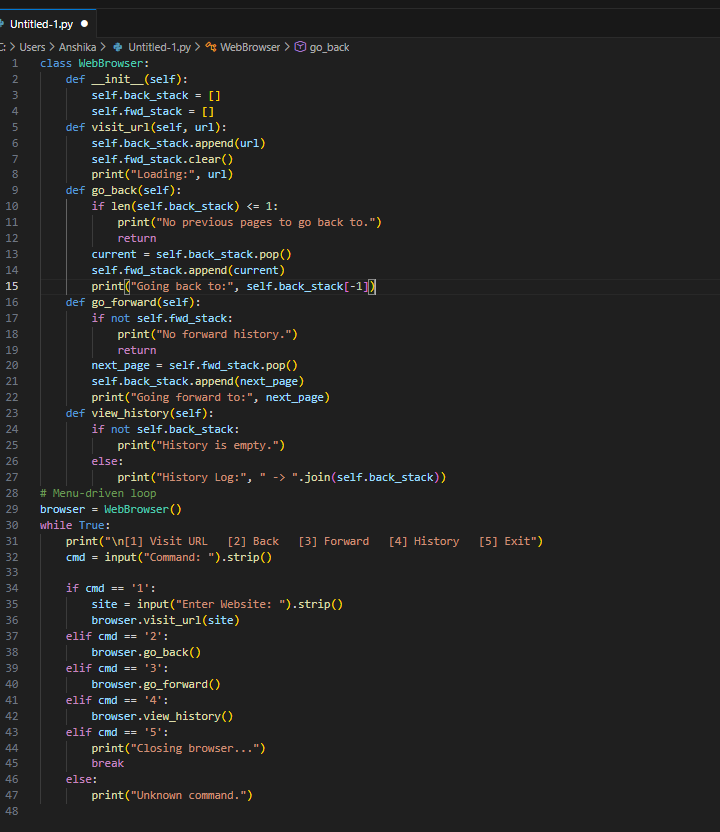
A browser uses two stacks to manage navigation:

A history stack that stores all visited pages (Back Stack).

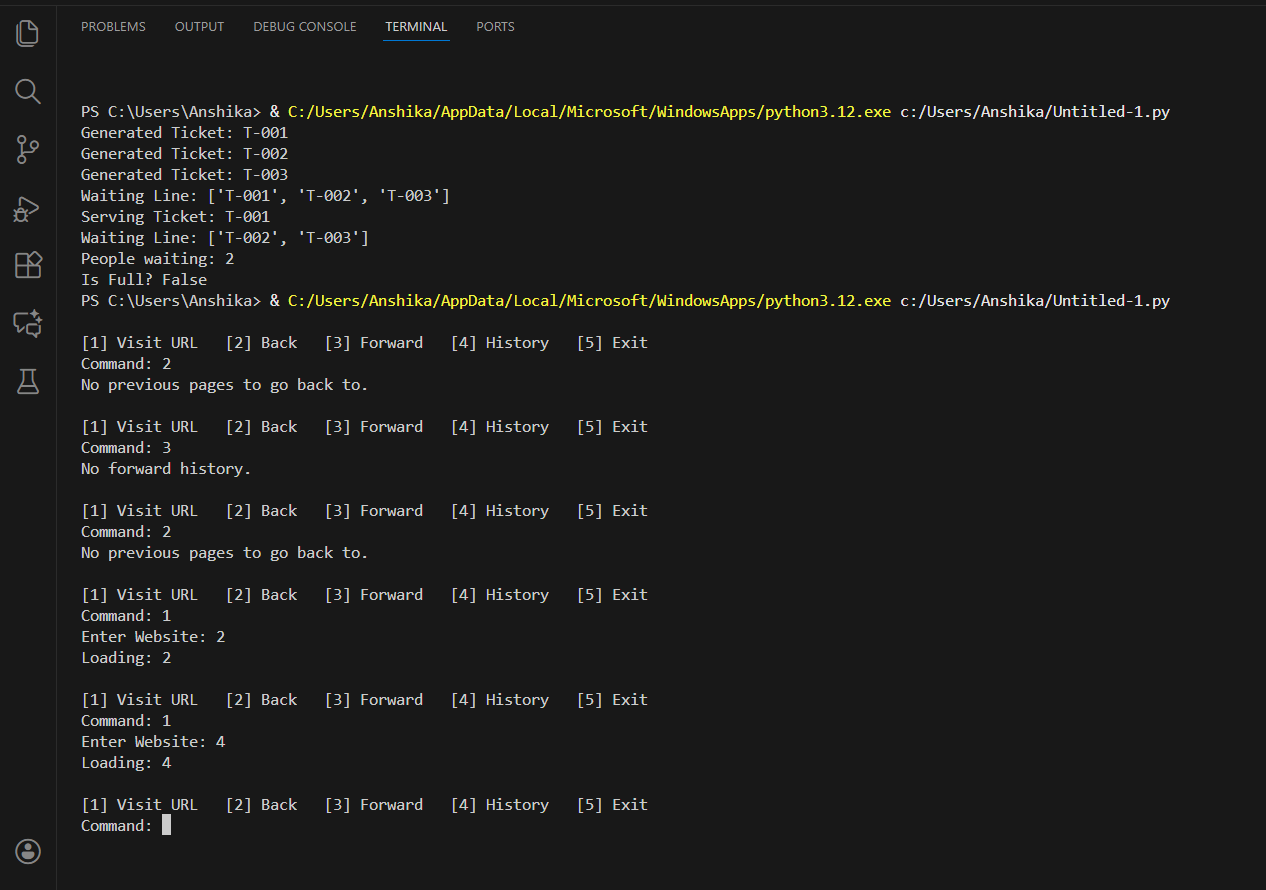
A forward stack that stores pages when the user clicks "Back".

When a page is visited, it is pushed onto the history stack. When the user goes back, the current page is popped from the history stack and pushed onto the forward stack. When the user goes forward, the top page from the forward stack is popped and returned to the history stack. This assignment demonstrates how stacks can be used to model browser navigation and provides hands-on experience in implementing LIFO-based functionalities.

**CODE-**

****

**OUTPUT –**

****

Conclusion-

The system demonstrates how backward and forward navigation in real browsers works using two stacks. This assignment strengthens the understanding of the LIFO principle and illustrates how stacks can solve real-world navigation problems efficiently. Overall, the experiment provides practical experience in using data structures to model real applications.

### LAB ASSIGNMENT – 4

**Circular Singly Linked List (Insert, Search, Delete, Display)**

Problem Statement-

Design and implement a Circular Singly Linked List in Python that supports insertion of nodes at the beginning, in the middle (after a given node), and at the end of the list. The program should also perform deletion of nodes from the beginning and end, along with search and display operations, while maintaining the circular nature of the linked list.

**Objective-**

* To understand the structure and working of a Circular Singly Linked List.
* To implement node-based operations using Python classes and pointers (links).
* To perform insertion at the beginning, in the middle (after a given value), and at the end of the circular linked list.
* To delete nodes from the beginning and end of the circular linked list.
* To search for a specific value in the circular list.
* To display all elements of the circular linked list in circular order.

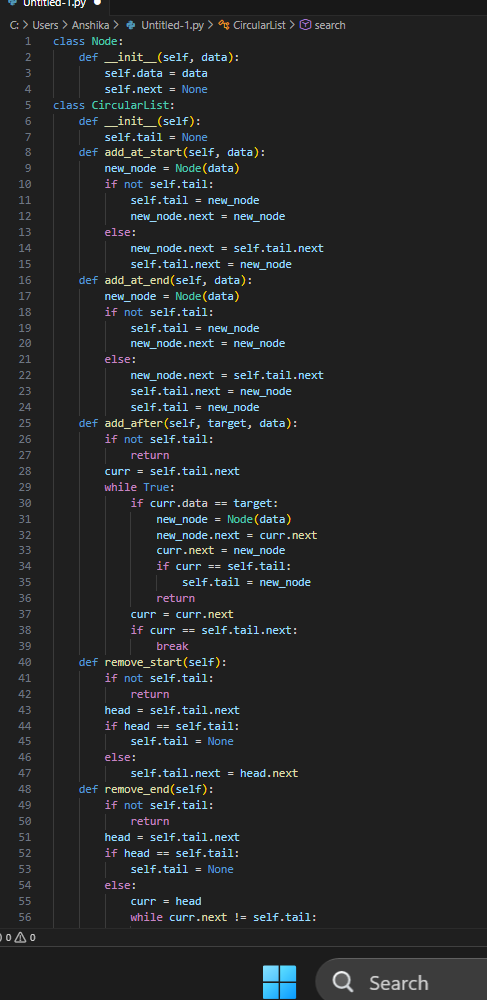
Description-

A Circular Singly Linked List is a variation of a singly linked list in which the last node does not point to NULL, but instead points back to the first node, forming a circle. In this assignment, a Circular Singly Linked List is implemented using a Node class and a CircularSinglyLinkedList class. The following operations are performed:

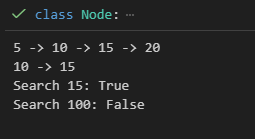
* **Insertion at Beginning:** A new node is added before the first node. The last node’s link is adjusted so that it still points to the new first node.
* **Insertion at End:** A new node is added after the last node and becomes the new last node. Its next pointer points back to the first node.
* **Insertion in Middle (After a Given Value):** A new node is inserted after a node that contains a specific value. Links are updated so that the circular structure is preserved.
* **Deletion from Beginning:** The first node is removed. The last node’s link is updated to point to the new first node.
* **Deletion from End:** The last node is removed. The second last node becomes the new last node and its next pointer points back to the first node.
* **Search:** The list is traversed in circular fashion to check if a given value exists.
* **Display:** All nodes are printed starting from the first node until the list comes back to the starting node, indicating circular traversal.

This assignment demonstrates how to carefully manage node links in a circular structure during insertions and deletions to maintain the integrity of the list.

**CODE-**



**OUTPUT-**

****

Conclusion-

The experiment provides a clear understanding of how circular linked lists operate and how to correctly adjust links while preserving the circular structure. Overall, this assignment strengthens knowledge of dynamic data structures, pointer manipulation, and traversal logic in circular linked lists.

### LAB ASSIGNMENT – 5

**Reverse a String Using Stack**

Problem Statement-

Design and implement a Python program to reverse a given string using the stack data structure. The program should push each character of the string onto a stack and then pop characters to obtain the reversed string, demonstrating the Last-In First-Out (LIFO) behavior of stacks.

**Objective-**

* To understand the concept of stack and its LIFO (Last-In First-Out) property.
* To apply stack operations such as push and pop in Python.
* To reverse a string using stack-based logic instead of built-in functions.
* To demonstrate practical use of stacks in solving real-world problems.
* To improve problem-solving and data-structure implementation skills.

Description-

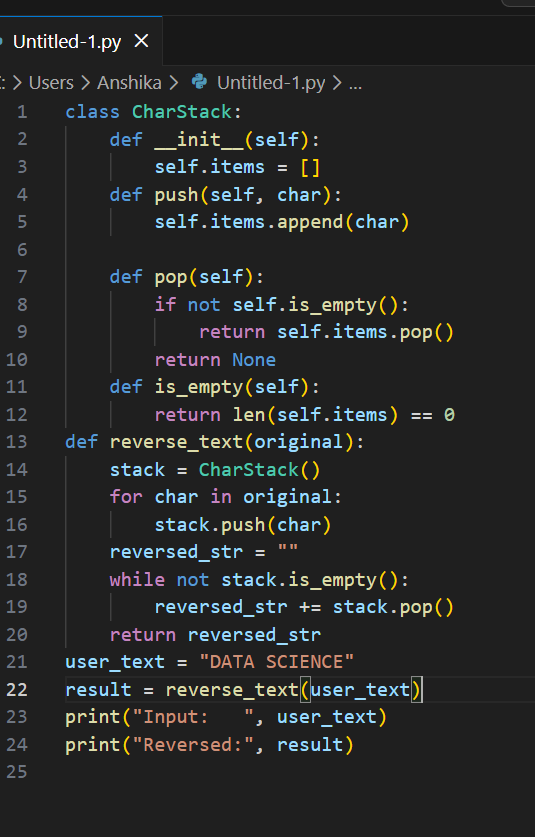
A stack is a linear data structure that follows the LIFO principle, meaning the last element inserted is the first one removed. This assignment uses a stack to reverse a string:

Each character of the input string is pushed onto a stack.

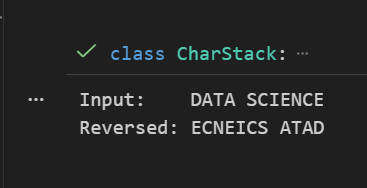
Characters are then popped from the stack, resulting in the string being reversed.

This process demonstrates how stacks can manipulate data order and support operations like undo, backtracking, and parsing. The program uses a simple list to represent the stack and implements stack operations manually to show internal working clearly.

**CODE-**

****

**OUTPUT CODE –**

****

Conclusion-

This experiment illustrates how the LIFO behavior of stacks can be applied to reverse data sequences. The assignment strengthens understanding of stack operations and highlights the usefulness of stacks in text processing, compilers, and algorithmic problem-solving.

### LAB ASSIGNMENT – 6

**Check Balanced Parentheses Using Stack**

Problem Statement-

Design and implement a Python program to check whether an expression containing parentheses, brackets, and braces is balanced or not using the stack data structure. The program must verify that every opening bracket has a corresponding closing bracket in the correct order.

**Objective-**

* To understand the use of stacks in syntax checking and expression validation.
* To apply push and pop operations to match pairs of parentheses.
* To check balance for multiple types of brackets: (), {}, and [].
* To demonstrate how stacks are used in compilers and parsers.
* To gain practical experience in using stacks to solve real-world problems such as expression checking.

Description-

Balanced parentheses are an important concept in expression validation. In this assignment, a stack is used to keep track of opening brackets:

For every opening bracket (, {, or [, it is pushed onto the stack.

For every closing bracket ), }, or ]:

The program checks whether the stack is not empty.

It then checks if the top of the stack matches the correct type of opening bracket. If not, the expression is unbalanced.

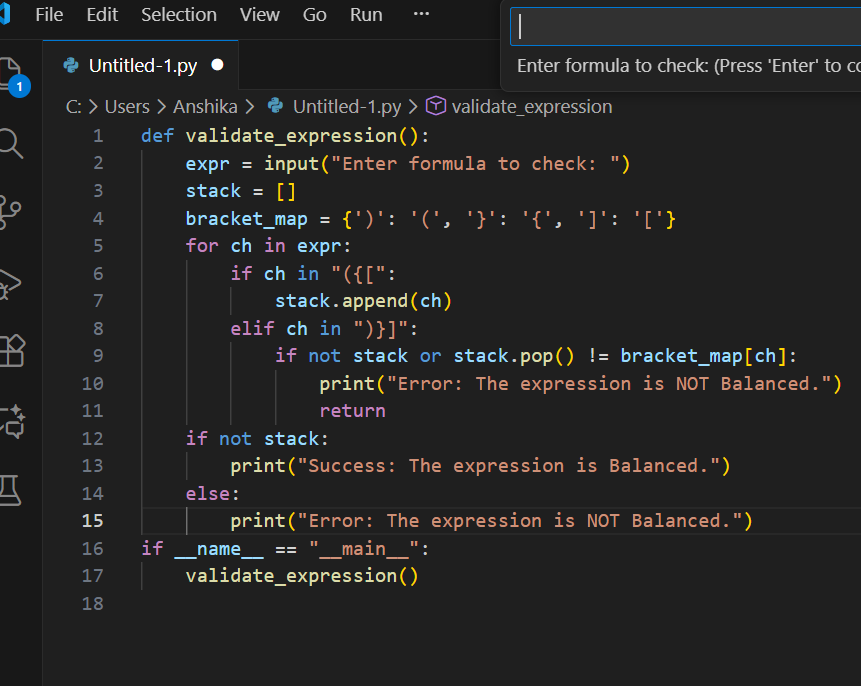
After scanning the entire expression:

If the stack is empty → the expression is balanced.

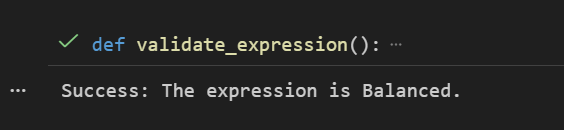
If the stack contains leftover opening brackets → the expression is not balanced.

This algorithm is widely used in compilers to validate code structure, making the stack essential for parsing tasks.

**CODE-**

****

**OUTPUT CODE –**

****

**CONCLUSION - The implemented program correctly checks whether an expression containing parentheses and brackets — (), {}, [] — is balanced or not. It uses a stack data structure to maintain the most recent unmatched opening bracket, and on encountering a closing bracket it verifies if the last opening bracket matches. This LIFO behaviour of stack ensures that nested and sequential bracket matching is handled properly.**