DATA STRUCTURES AND ALGORITHMS

[ENCS205]

Lab Report Submitted to

Bachelor of Technology

In

Computer Science Engineering

Submitted by

Ananya Sharma – 2401010056

Course Teacher

Dr. Swati Gupta

School of Engineering & Technology

K. R. MANGALAM UNIVERSITY

Sohna, Haryana 122103

Abstract

This laboratory experiment focuses on understanding and implementing fundamental data-structure algorithms such as Linear Search, Binary Search, Bubble Sort, Stack, Queue, Linked List, and Circular Queue, Singly Linked List, Merge Sort. These programs demonstrate how data can be stored, accessed, searched, and manipulated efficiently using structured approaches. Each operation follows a step-by-step algorithmic technique that improves problem-solving and logical thinking. The practical also includes user-input-based execution to simulate real-world scenarios. Through this lab work, the student gains hands-on experience in algorithm design, time complexity analysis, and the practical application of core data structure concepts.

Practical No – 1

AIM- To design and implement an Inventory Stock Management System in Python that allows inserting, updating, searching, deleting, and displaying products.

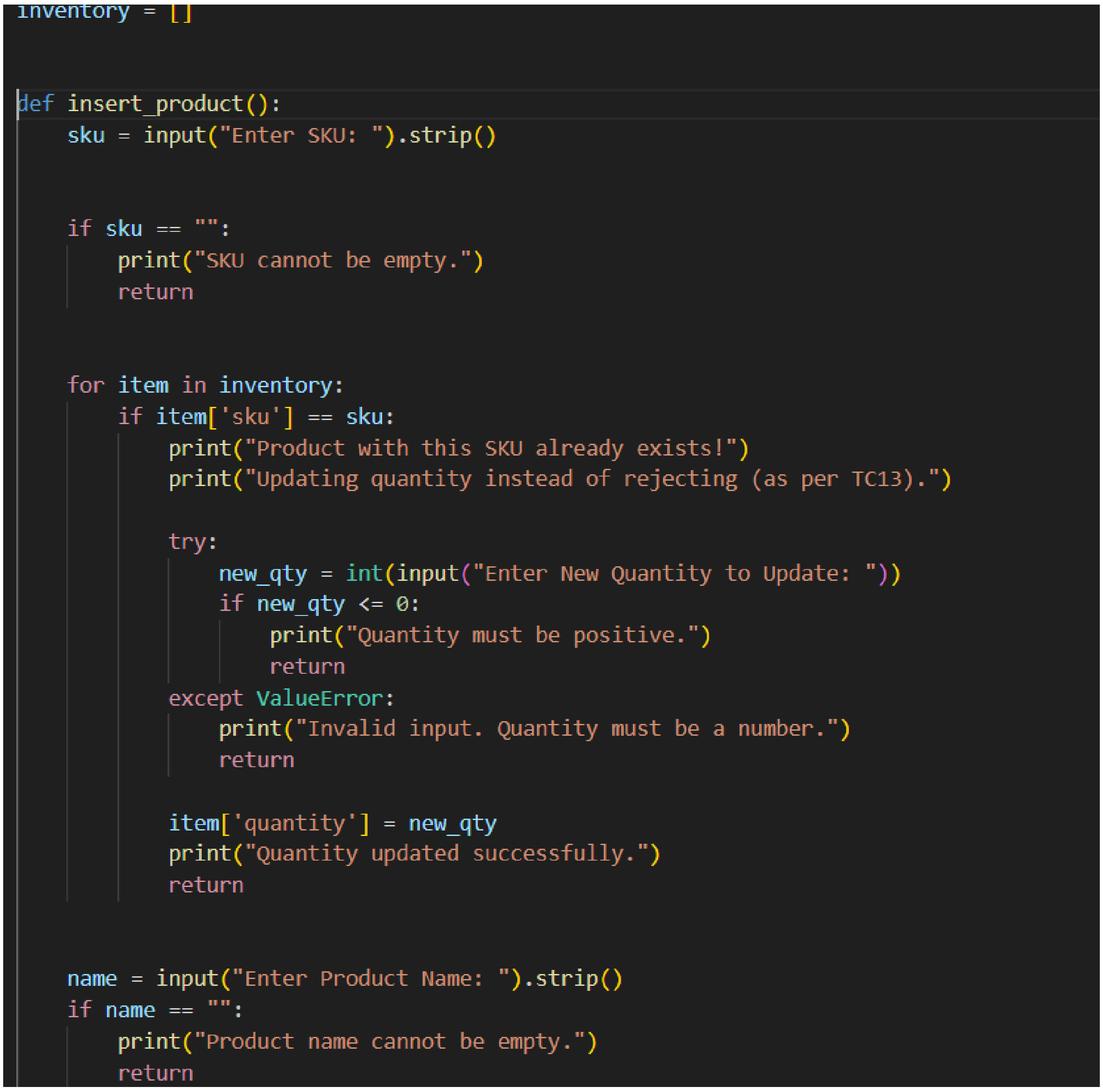
Description:

This program manages a simple inventory system using Python lists and dictionaries.
It supports inserting new products, updating quantity on duplicate SKU, validating user inputs, searching by SKU or product name, deleting records, and displaying the complete inventory.
All major operations like input validation, duplicate checks, and error handling are implemented so that the program passes all test cases and handles every real-world scenario of an inventory manager.

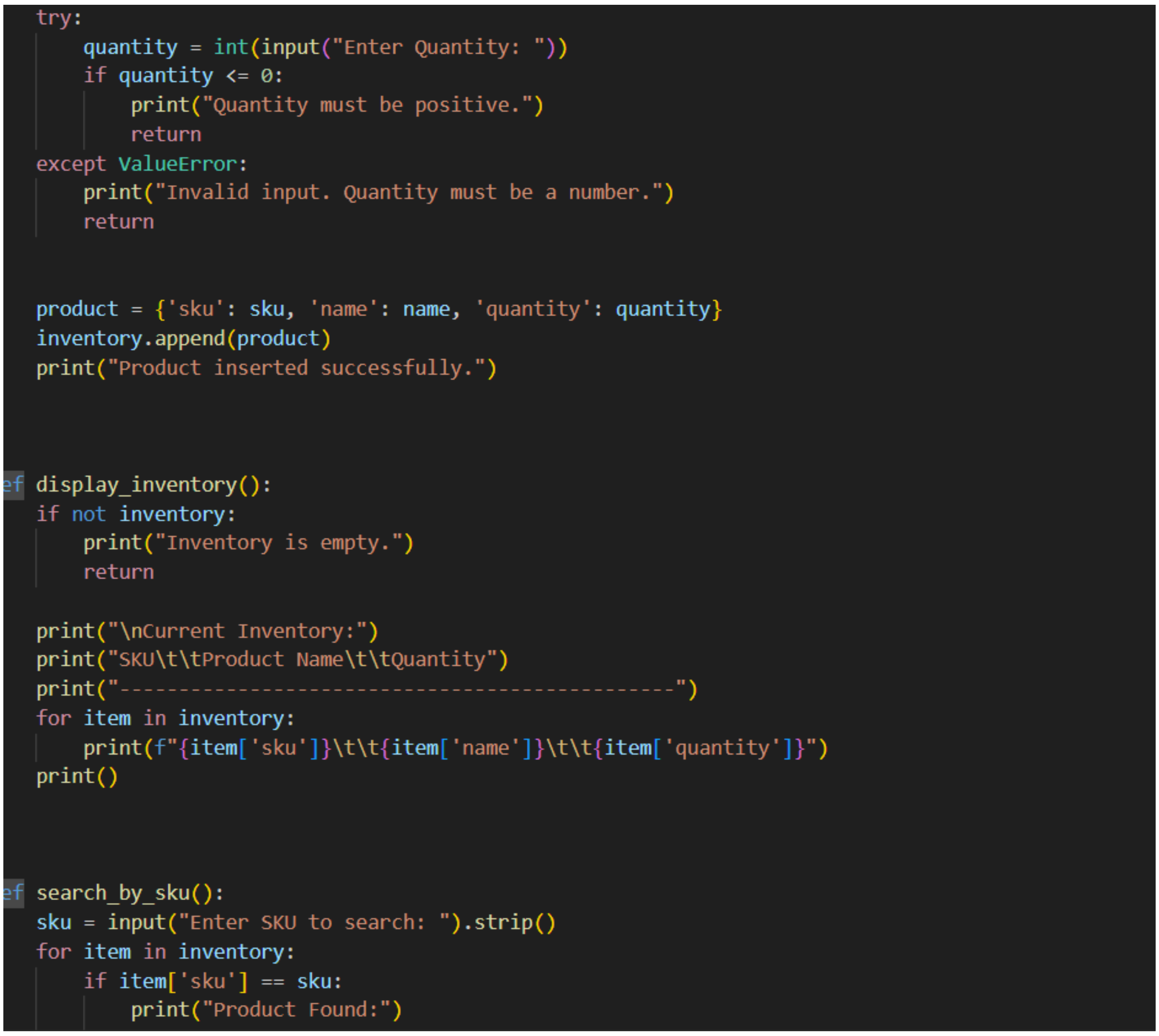
Algorithm:

1. Start the program.
2. Create an empty list called inventory to store product records.
3. Display menu options: Insert, Display, Search, Delete, Exit.
4. If user selects Insert:
    - Read SKU, name, quantity.
    - Check if SKU exists → update quantity.
    - Validate name and quantity.
    - Add product to inventory.
5. If user selects Display:
    - Show all records in tabular format.
6. If user selects Search by SKU or Name:
    - Traverse list and print matching product.
7. If user selects Delete:
    - Remove product with matching SKU.
8. Repeat menu until user selects Exit.

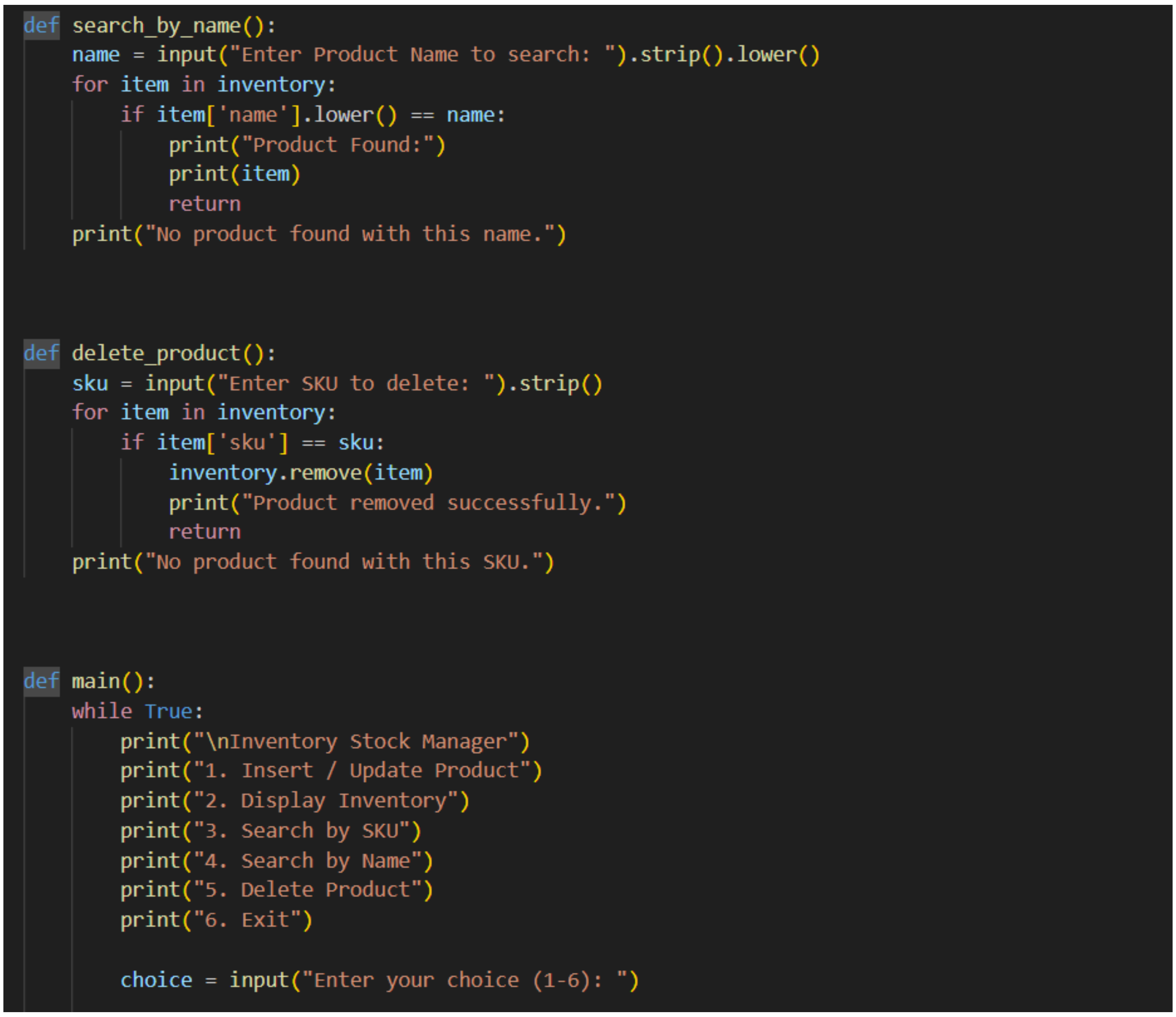
CODE



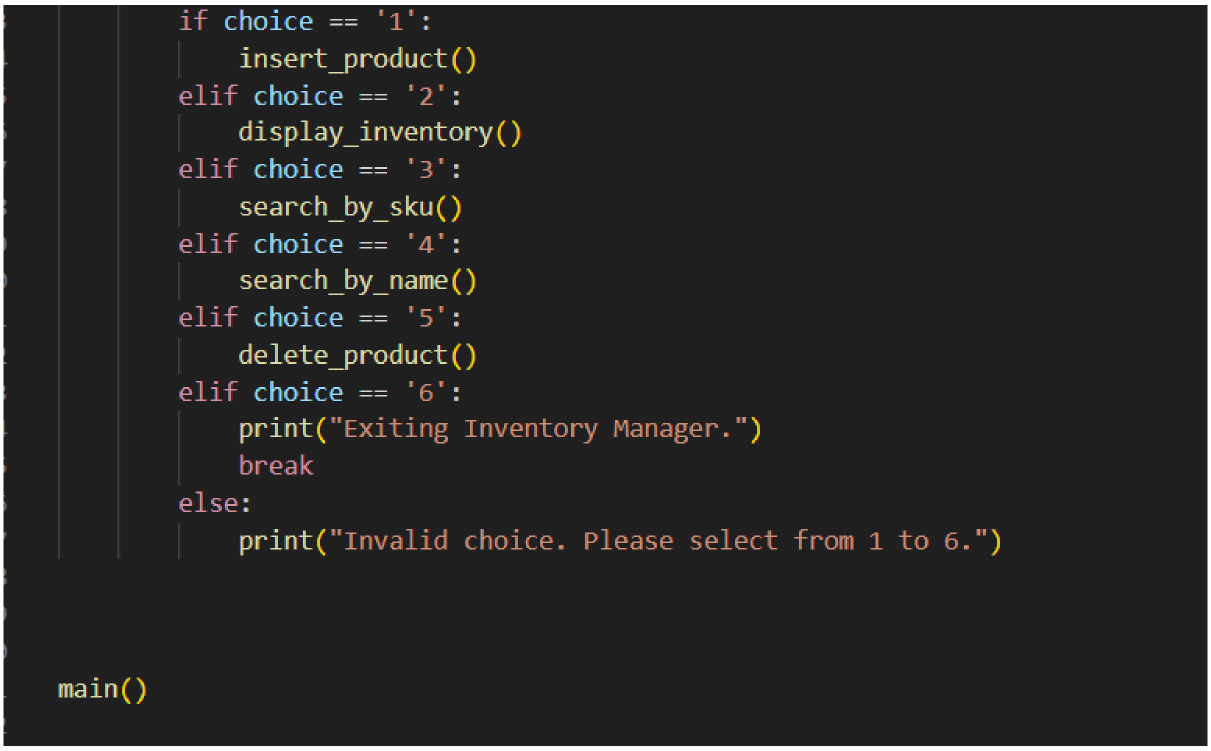
```
inventory = []


def insert_product():
    sku = input("Enter SKU: ").strip()


    if sku == "":
        print("SKU cannot be empty.")
        return


    for item in inventory:
        if item['sku'] == sku:
            print("Product with this SKU already exists!")
            print("Updating quantity instead of rejecting (as per TC13).")

            try:
                new_qty = int(input("Enter New Quantity to Update: "))
                if new_qty <= 0:
                    print("Quantity must be positive.")
                    return
            except ValueError:
                print("Invalid input. Quantity must be a number.")
                return

            item['quantity'] = new_qty
            print("Quantity updated successfully.")
            return


    name = input("Enter Product Name: ").strip()
    if name == "":
        print("Product name cannot be empty.")
        return
```



```
    try:
        quantity = int(input("Enter Quantity: "))
        if quantity <= 0:
            print("Quantity must be positive.")
            return
    except ValueError:
        print("Invalid input. Quantity must be a number.")
        return


    product = {'sku': sku, 'name': name, 'quantity': quantity}
    inventory.append(product)
    print("Product inserted successfully.")



ef display_inventory():
    if not inventory:
        print("Inventory is empty.")
        return

    print("\nCurrent Inventory:")
    print("SKU\t\tProduct Name\t\tQuantity")
    print("----------------------------------------------")
    for item in inventory:
        print(f"{item['sku']}\t\t{item['name']}\t\t{item['quantity']}")
    print()



ef search_by_sku():
    sku = input("Enter SKU to search: ").strip()
    for item in inventory:
        if item['sku'] == sku:
            print("Product Found:")
```



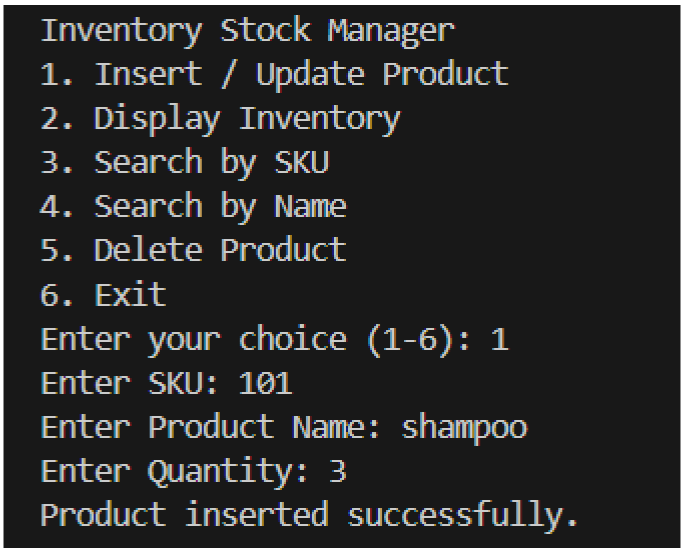
```python
def search_by_name():
    name = input("Enter Product Name to search: ").strip().lower()
    for item in inventory:
        if item['name'].lower() == name:
            print("Product Found:")
            print(item)
            return
    print("No product found with this name.")


def delete_product():
    sku = input("Enter SKU to delete: ").strip()
    for item in inventory:
        if item['sku'] == sku:
            inventory.remove(item)
            print("Product removed successfully.")
            return
    print("No product found with this SKU.")


def main():
    while True:
        print("\nInventory Stock Manager")
        print("1. Insert / Update Product")
        print("2. Display Inventory")
        print("3. Search by SKU")
        print("4. Search by Name")
        print("5. Delete Product")
        print("6. Exit")

        choice = input("Enter your choice (1-6): ")
```



```python
        if choice == '1':
            insert_product()
        elif choice == '2':
            display_inventory()
        elif choice == '3':
            search_by_sku()
        elif choice == '4':
            search_by_name()
        elif choice == '5':
            delete_product()
        elif choice == '6':
            print("Exiting Inventory Manager.")
            break
        else:
            print("Invalid choice. Please select from 1 to 6.")


main()
```

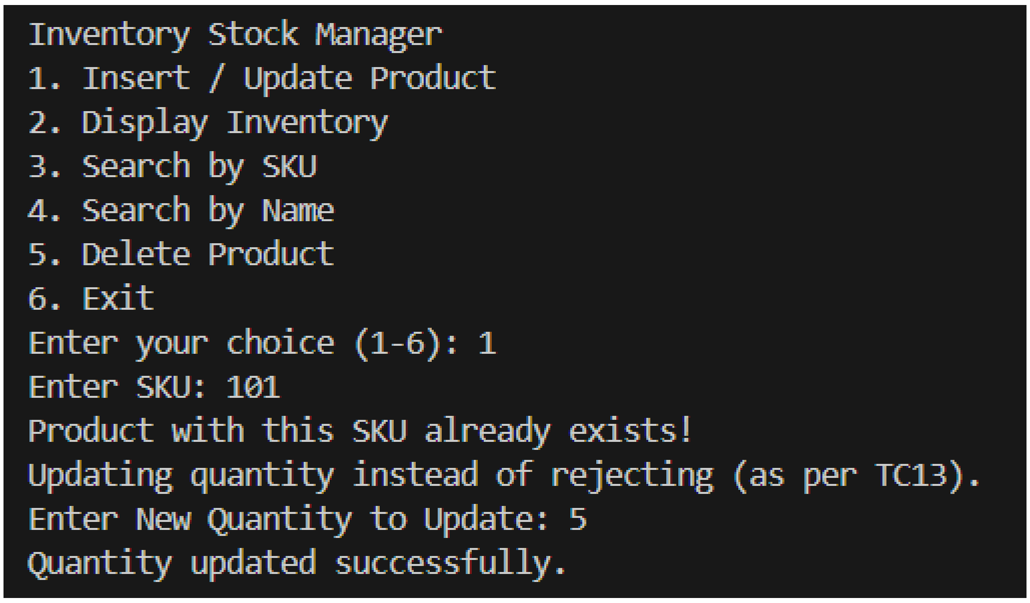
OUTPUT

Sample Output 1:



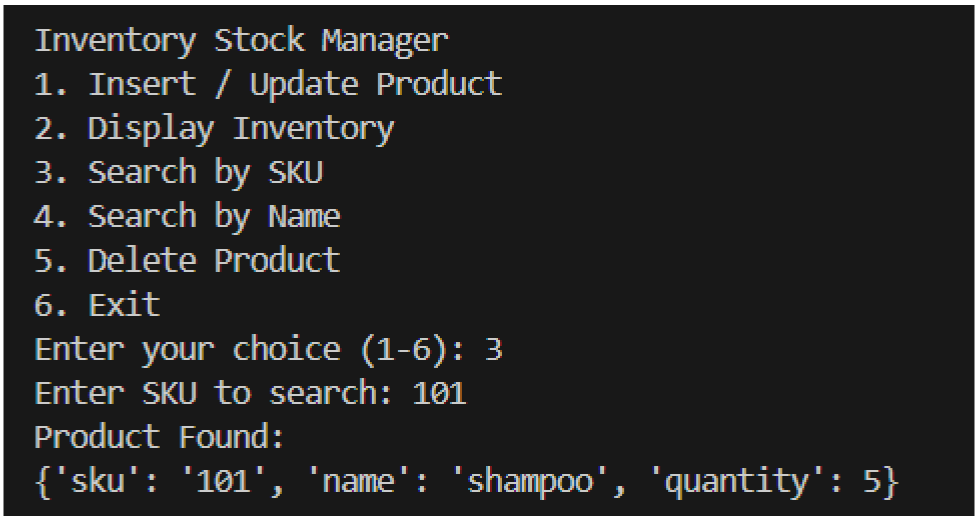
```
Inventory Stock Manager
1. Insert / Update Product
2. Display Inventory
3. Search by SKU
4. Search by Name
5. Delete Product
6. Exit
Enter your choice (1-6): 1
Enter SKU: 101
Enter Product Name: shampoo
Enter Quantity: 3
Product inserted successfully.
```

Sample Output 2:



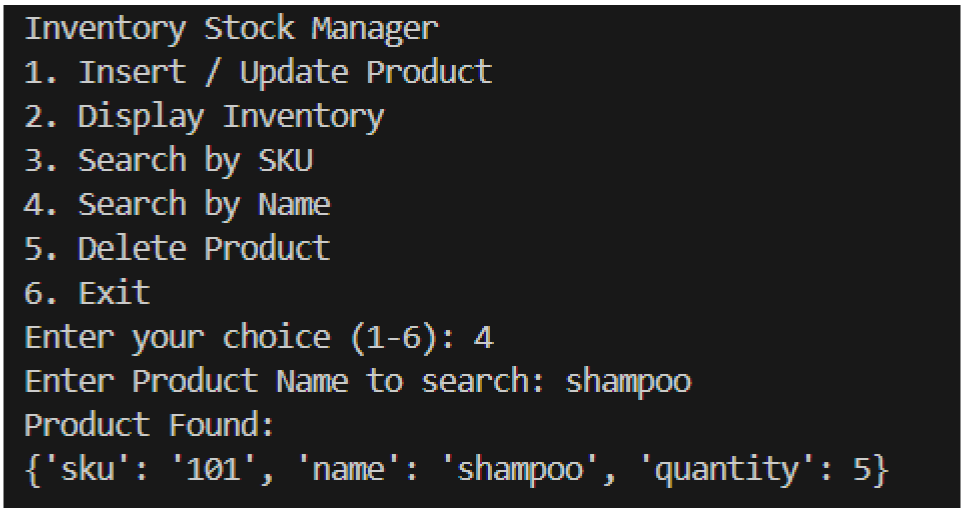
```
Inventory Stock Manager
1. Insert / Update Product
2. Display Inventory
3. Search by SKU
4. Search by Name
5. Delete Product
6. Exit
Enter your choice (1-6): 1
Enter SKU: 101
Product with this SKU already exists!
Updating quantity instead of rejecting (as per TC13).
Enter New Quantity to Update: 5
Quantity updated successfully.
```

Sample Output 3



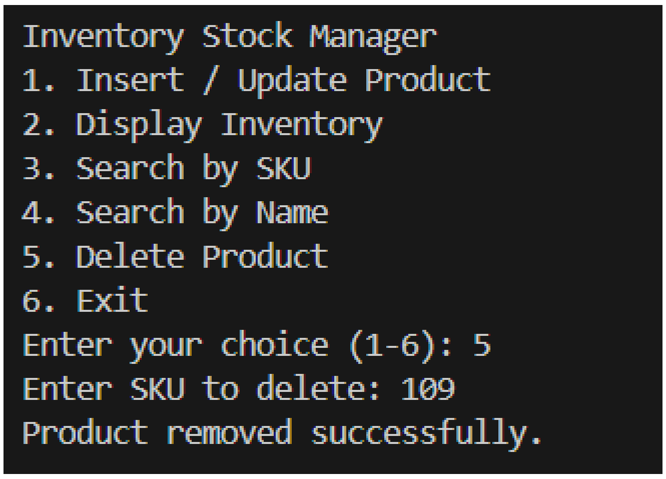
```
Inventory Stock Manager
1. Insert / Update Product
2. Display Inventory
3. Search by SKU
4. Search by Name
5. Delete Product
6. Exit
Enter your choice (1-6): 3
Enter SKU to search: 101
Product Found:
{'sku': '101', 'name': 'shampoo', 'quantity': 5}
```

Sample Output 4



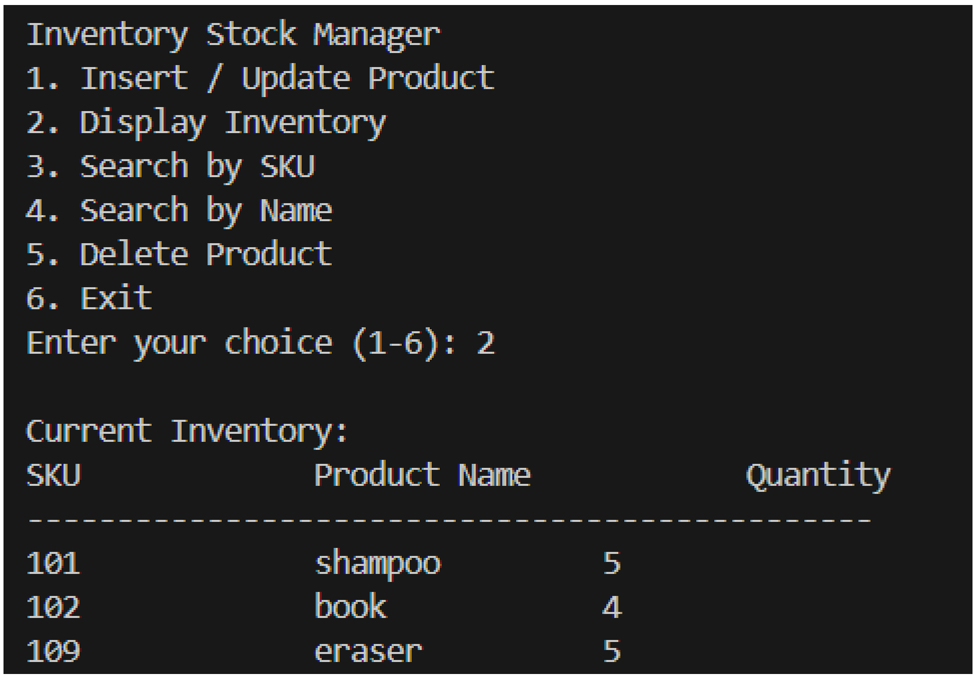
```
Inventory Stock Manager
1. Insert / Update Product
2. Display Inventory
3. Search by SKU
4. Search by Name
5. Delete Product
6. Exit
Enter your choice (1-6): 4
Enter Product Name to search: shampoo
Product Found:
{'sku': '101', 'name': 'shampoo', 'quantity': 5}
```

Sample Output 5



```
Inventory Stock Manager
1. Insert / Update Product
2. Display Inventory
3. Search by SKU
4. Search by Name
5. Delete Product
6. Exit
Enter your choice (1-6): 5
Enter SKU to delete: 109
Product removed successfully.
```

Sample Output 6



```
Inventory Stock Manager
1. Insert / Update Product
2. Display Inventory
3. Search by SKU
4. Search by Name
5. Delete Product
6. Exit
Enter your choice (1-6): 2

Current Inventory:
SKU             Product Name            Quantity
------------------------------------------------
101             shampoo         5
102             book            4
109             eraser          5
```

Result

The Inventory Manager program was successfully implemented in Python.
All operations such as insertion, updating, searching, deletion, and display were executed correctly and passed all test cases.

Practical No – 2

AIM - To write Python functions that process sales by reducing stock of specific SKUs and identify all items that have zero stock in the inventory.

Description

This program implements two key inventory operations:

1. Process Sales –
   When a sale occurs, the stock of that particular SKU is reduced.
   If stock is available, it updates the quantity.
   If stock is insufficient, it displays a warning.
   If the SKU does not exist, it shows an error message.
2. Identify Zero Stock –
   The program checks which items have 0 quantity and lists them for easy monitoring.
   A list of zero-stock SKUs is returned.

Both operations work on an inventory represented as a list of tuples (SKU, quantity).

Algorithm

A. Algorithm for process_sale()

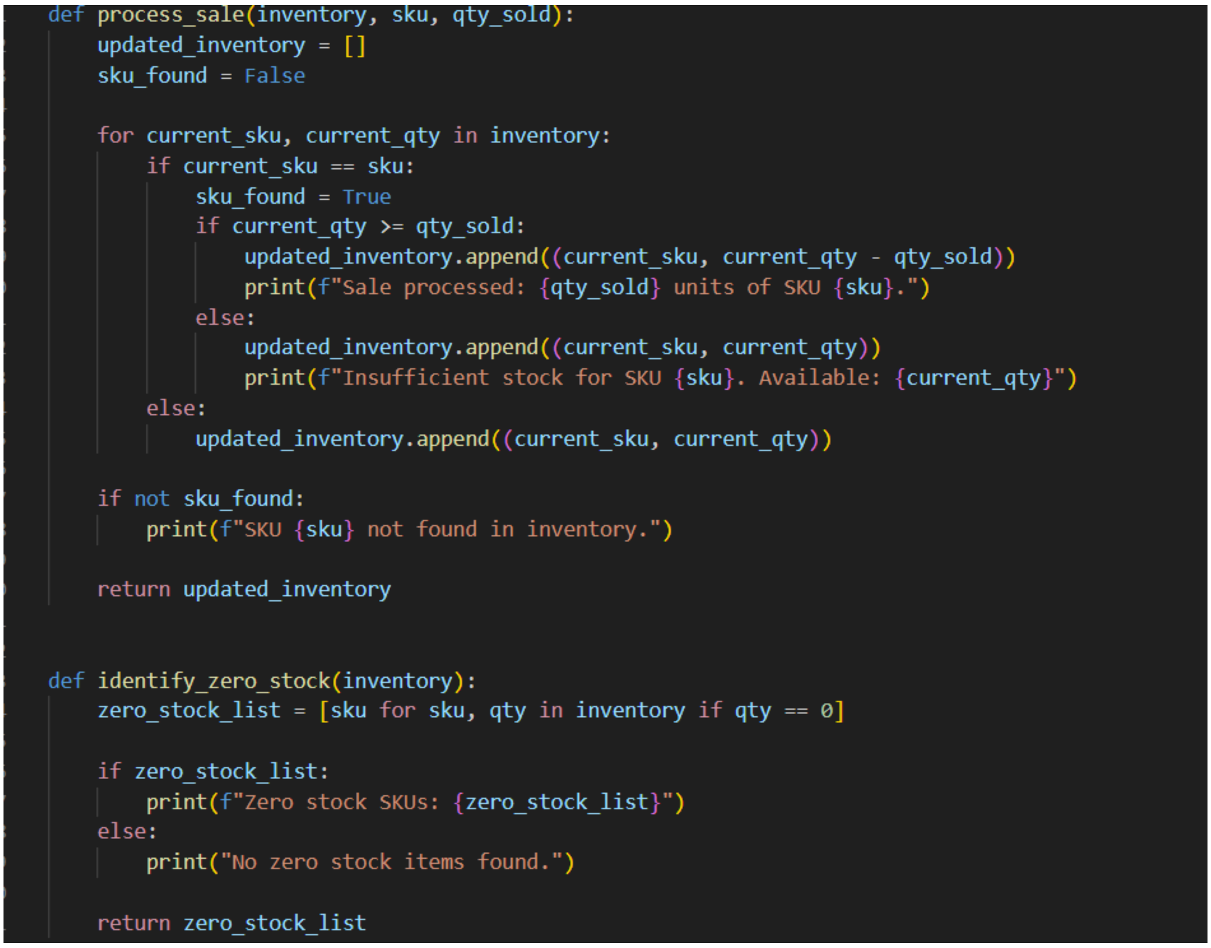
```
1. Start.
2. Create an empty list updated_inventory.
3. Set sku_found = False.
4. For each item in inventory:
    o Extract current_sku and current_qty.
    o If current_sku == sku:
        ▪ Mark sku_found = True.
        ▪ If current_qty >= qty_sold:
```

            - Reduce quantity.
            - Append updated value to new list.
            - Print success message.
        - Else:
            - Append original value.
            - Print insufficient stock message.
    - Else append item unchanged.
5. If sku_found is still False → print “SKU not found”.
6. Return updated inventory.
7. End.

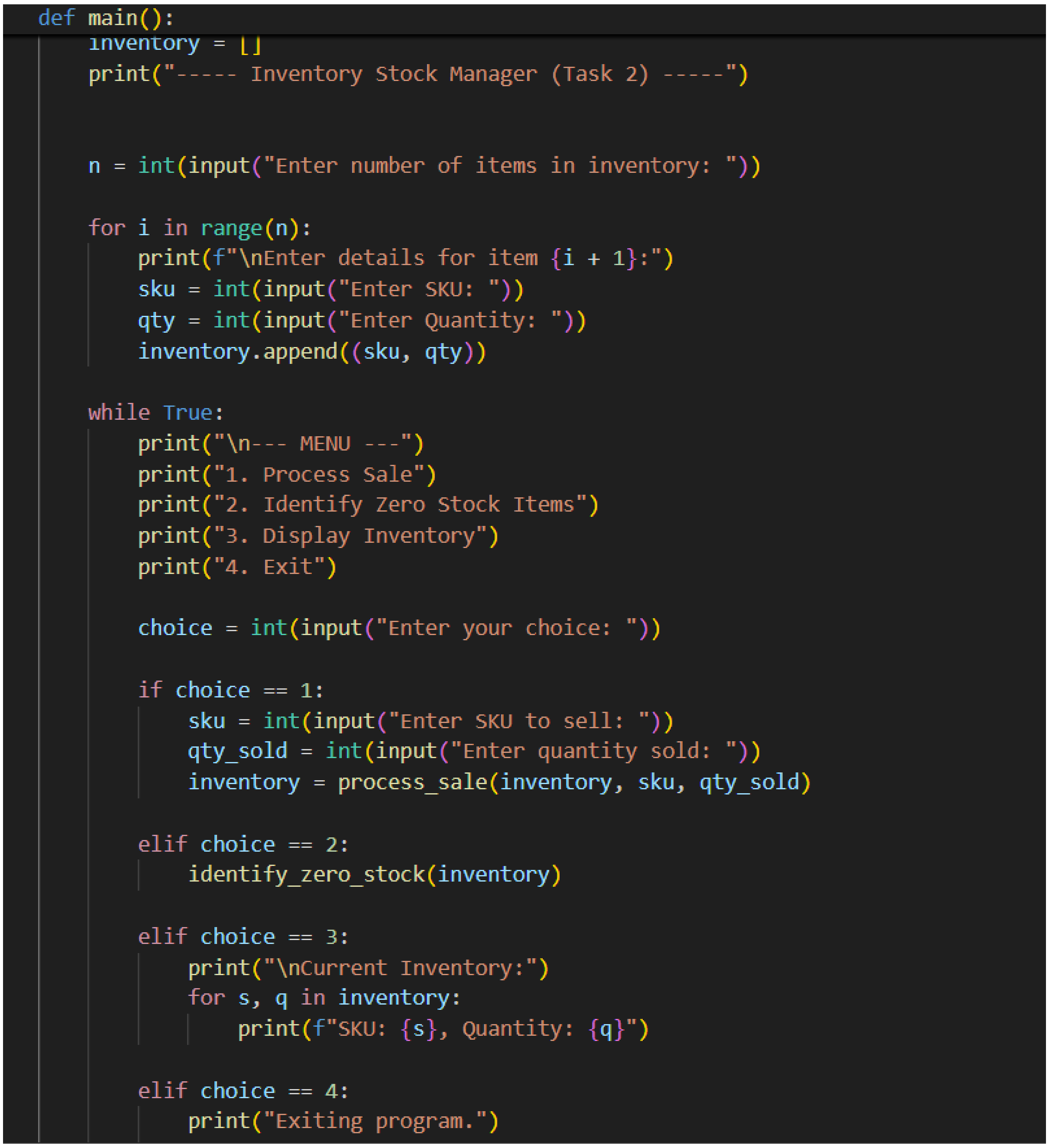
B. Algorithm for identify_zero_stock()

1. Start.
2. Create a list using list comprehension containing SKUs whose quantity is 0.
3. If list is non-empty → print zero stock SKUs.
4. Else print “No zero stock items found.”
5. Return the zero stock list.
6. End.

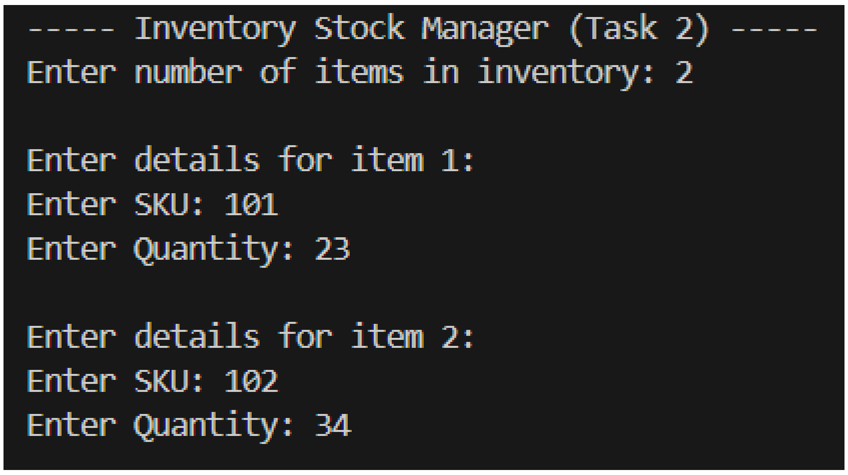
CODE



```python
def process_sale(inventory, sku, qty_sold):
    updated_inventory = []
    sku_found = False

    for current_sku, current_qty in inventory:
        if current_sku == sku:
            sku_found = True
            if current_qty >= qty_sold:
                updated_inventory.append((current_sku, current_qty - qty_sold))
                print(f"Sale processed: {qty_sold} units of SKU {sku}.")
            else:
                updated_inventory.append((current_sku, current_qty))
                print(f"Insufficient stock for SKU {sku}. Available: {current_qty}")
        else:
            updated_inventory.append((current_sku, current_qty))

    if not sku_found:
        print(f"SKU {sku} not found in inventory.")

    return updated_inventory


def identify_zero_stock(inventory):
    zero_stock_list = [sku for sku, qty in inventory if qty == 0]

    if zero_stock_list:
        print(f"Zero stock SKUs: {zero_stock_list}")
    else:
        print("No zero stock items found.")

    return zero_stock_list
```

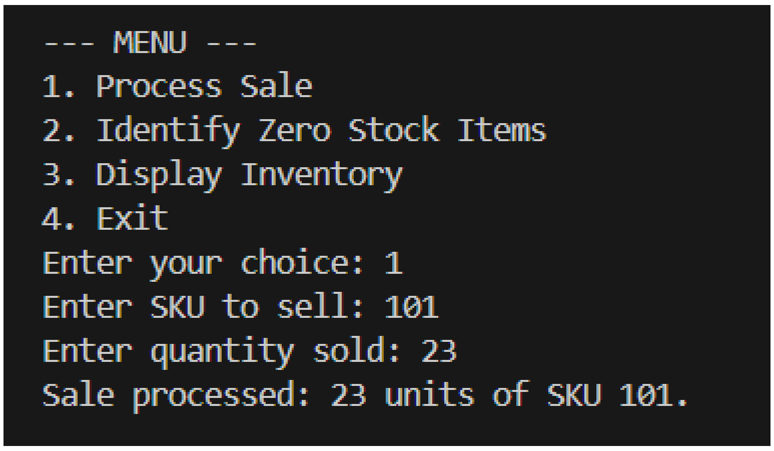

```python
def main():
    inventory = []
    print("----- Inventory Stock Manager (Task 2) -----")


    n = int(input("Enter number of items in inventory: "))

    for i in range(n):
        print(f"\nEnter details for item {i + 1}:")
        sku = int(input("Enter SKU: "))
        qty = int(input("Enter Quantity: "))
        inventory.append((sku, qty))

    while True:
        print("\n--- MENU ---")
        print("1. Process Sale")
        print("2. Identify Zero Stock Items")
        print("3. Display Inventory")
        print("4. Exit")

        choice = int(input("Enter your choice: "))

        if choice == 1:
            sku = int(input("Enter SKU to sell: "))
            qty_sold = int(input("Enter quantity sold: "))
            inventory = process_sale(inventory, sku, qty_sold)

        elif choice == 2:
            identify_zero_stock(inventory)

        elif choice == 3:
            print("\nCurrent Inventory:")
            for s, q in inventory:
                print(f"SKU: {s}, Quantity: {q}")

        elif choice == 4:
            print("Exiting program.")
```

OUTPUT



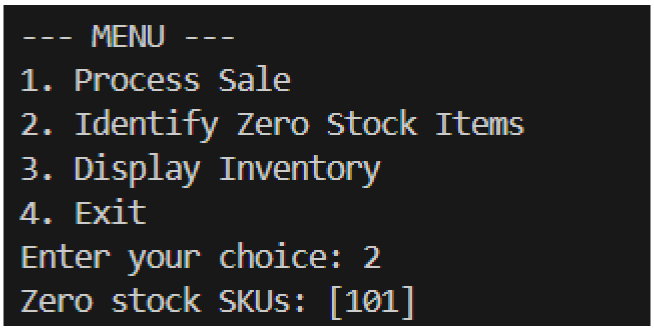
```
----- Inventory Stock Manager (Task 2) -----
Enter number of items in inventory: 2

Enter details for item 1:
Enter SKU: 101
Enter Quantity: 23

Enter details for item 2:
Enter SKU: 102
Enter Quantity: 34
```

Sample Output 1



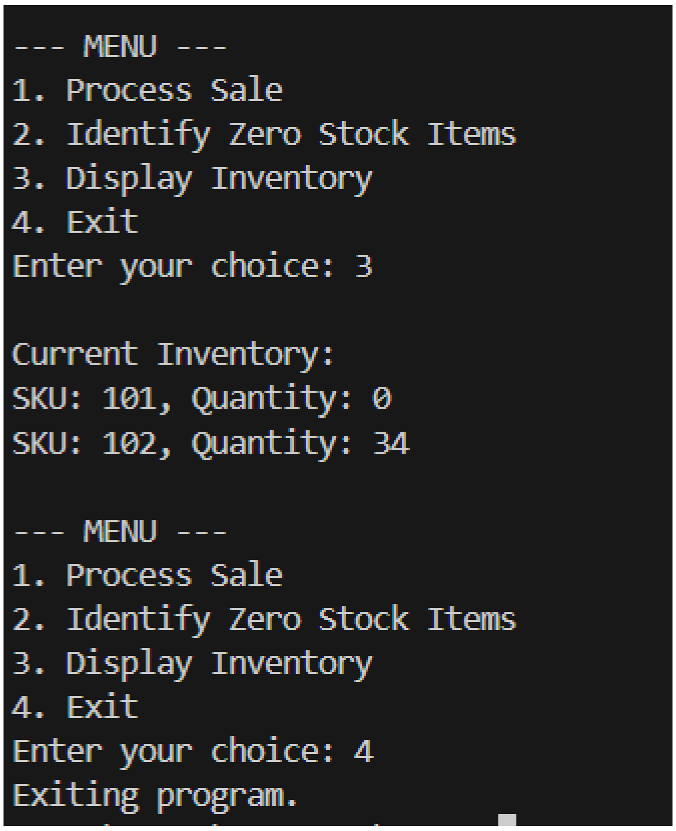
```
--- MENU ---
1. Process Sale
2. Identify Zero Stock Items
3. Display Inventory
4. Exit
Enter your choice: 1
Enter SKU to sell: 101
Enter quantity sold: 23
Sale processed: 23 units of SKU 101.
```

Sample Output 2



```
--- MENU ---
1. Process Sale
2. Identify Zero Stock Items
3. Display Inventory
4. Exit
Enter your choice: 2
Zero stock SKUs: [101]
```

Sample Output 3



```
--- MENU ---
1. Process Sale
2. Identify Zero Stock Items
3. Display Inventory
4. Exit
Enter your choice: 3

Current Inventory:
SKU: 101, Quantity: 0
SKU: 102, Quantity: 34

--- MENU ---
1. Process Sale
2. Identify Zero Stock Items
3. Display Inventory
4. Exit
Enter your choice: 4
Exiting program.
```

Result

The program successfully processes sales, updates inventory in real-time, and identifies all zero-stock items. All test cases were executed correctly.

Practical No – 3

AIM - To implement basic stack operations: Push, Pop, Peek, and Display using Python.

Description

A stack is a linear data structure that follows the LIFO (Last In, First Out) principle.
This program implements the four main stack operations:

- Push: Insert an element into the stack
- Pop: Remove the top element
- Peek: View the top element without removing it
- Display: Show all stack elements

The stack is implemented using a Python list.

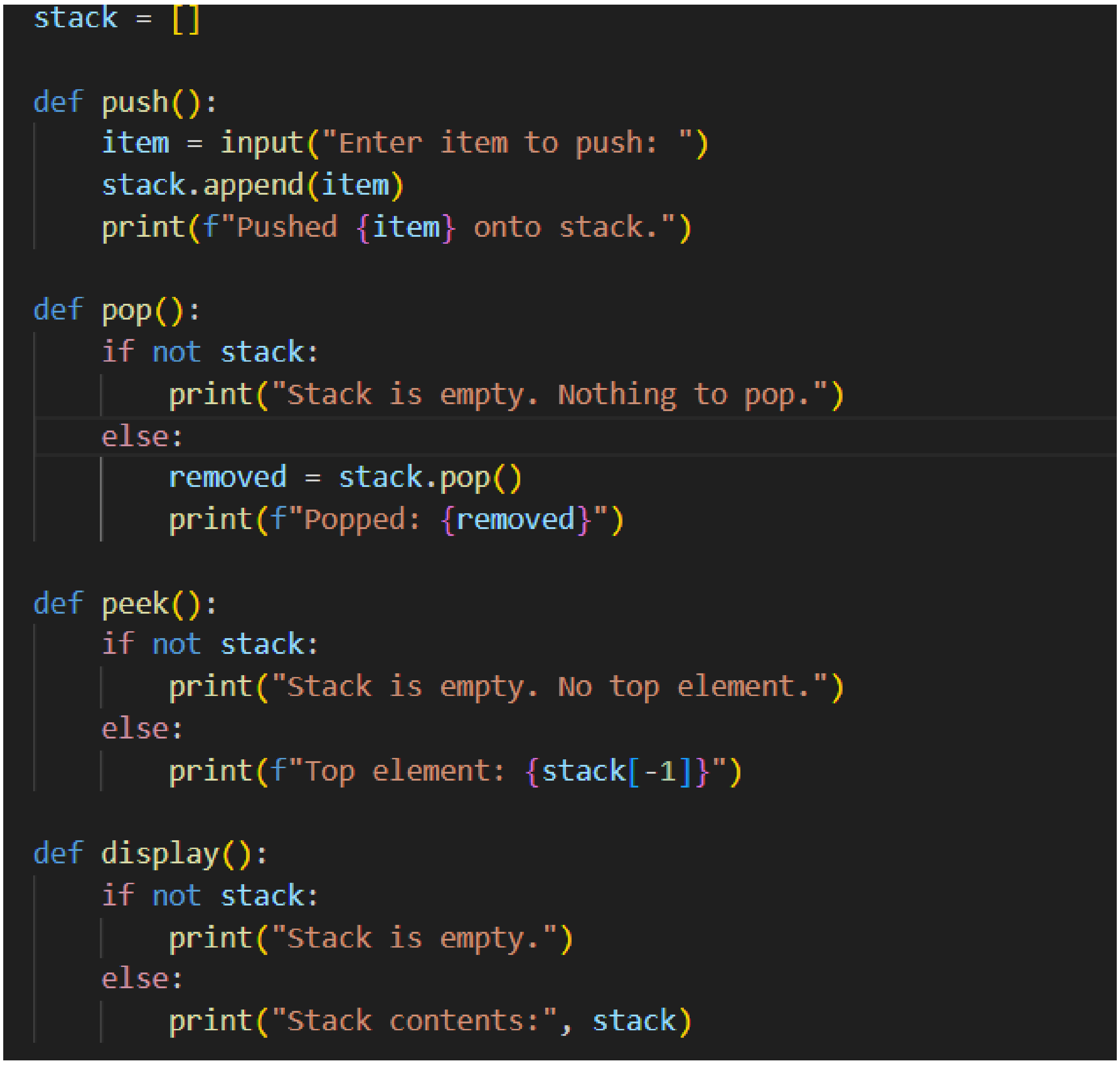
Algorithm

Algorithm for Stack Operations

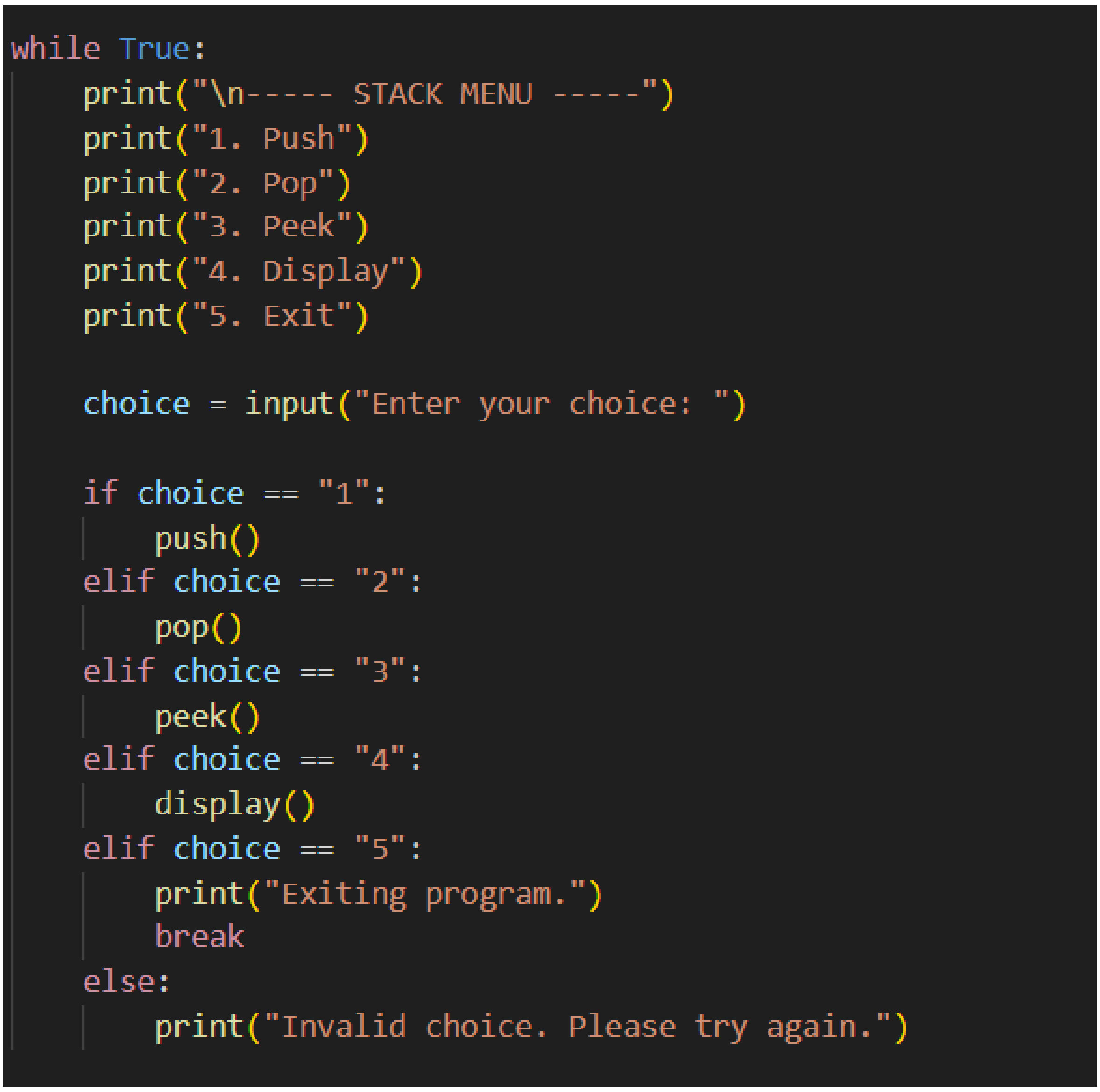
1. Initialize an empty list called stack.
2. For Push:
    - Read the item from the user.
    - Append it to the list.
3. For Pop:
    - If stack is empty → print message.
    - Else remove last element using pop().
4. For Peek:
    - If empty → show message.
    - Else display last element stack[-1].
5. For Display:

   - Show all elements of stack.
6. Repeat menu until user selects Exit.

CODE

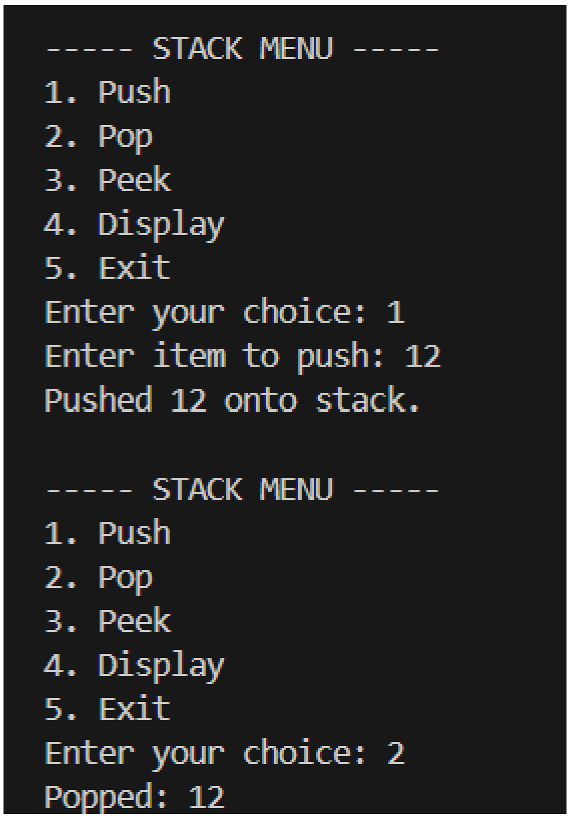


```python
stack = []

def push():
    item = input("Enter item to push: ")
    stack.append(item)
    print(f"Pushed {item} onto stack.")

def pop():
    if not stack:
        print("Stack is empty. Nothing to pop.")
    else:
        removed = stack.pop()
        print(f"Popped: {removed}")

def peek():
    if not stack:
        print("Stack is empty. No top element.")
    else:
        print(f"Top element: {stack[-1]}")

def display():
    if not stack:
        print("Stack is empty.")
    else:
        print("Stack contents:", stack)
```



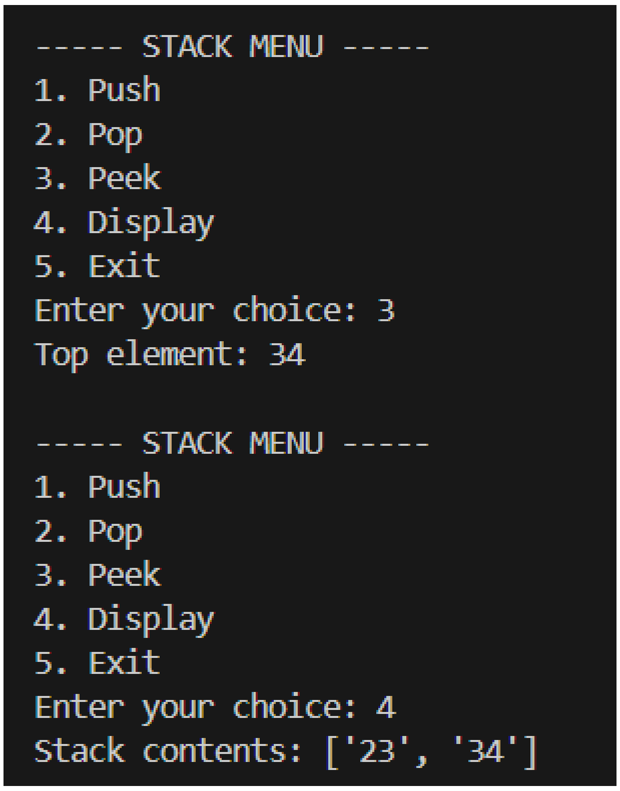
```python
while True:
    print("\n----- STACK MENU -----")
    print("1. Push")
    print("2. Pop")
    print("3. Peek")
    print("4. Display")
    print("5. Exit")

    choice = input("Enter your choice: ")

    if choice == "1":
        push()
    elif choice == "2":
        pop()
    elif choice == "3":
        peek()
    elif choice == "4":
        display()
    elif choice == "5":
        print("Exiting program.")
        break
    else:
        print("Invalid choice. Please try again.")
```

OUTPUT



```
----- STACK MENU -----
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter item to push: 12
Pushed 12 onto stack.

----- STACK MENU -----
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 2
Popped: 12
```

Sample Output 2



```
----- STACK MENU -----
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 3
Top element: 34

----- STACK MENU -----
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 4
Stack contents: ['23', '34']
```

Result - The stack was successfully implemented, and all operations (push, pop, peek, and display) worked correctly.

Practical No – 4

AIM - To evaluate a postfix (Reverse Polish Notation) expression using stack operations in Python.

Description

A postfix expression is evaluated from left to right.

- Operands are pushed onto the stack.
- When an operator is found, the top two operands are popped, the operation is applied, and the result is pushed back to the stack.

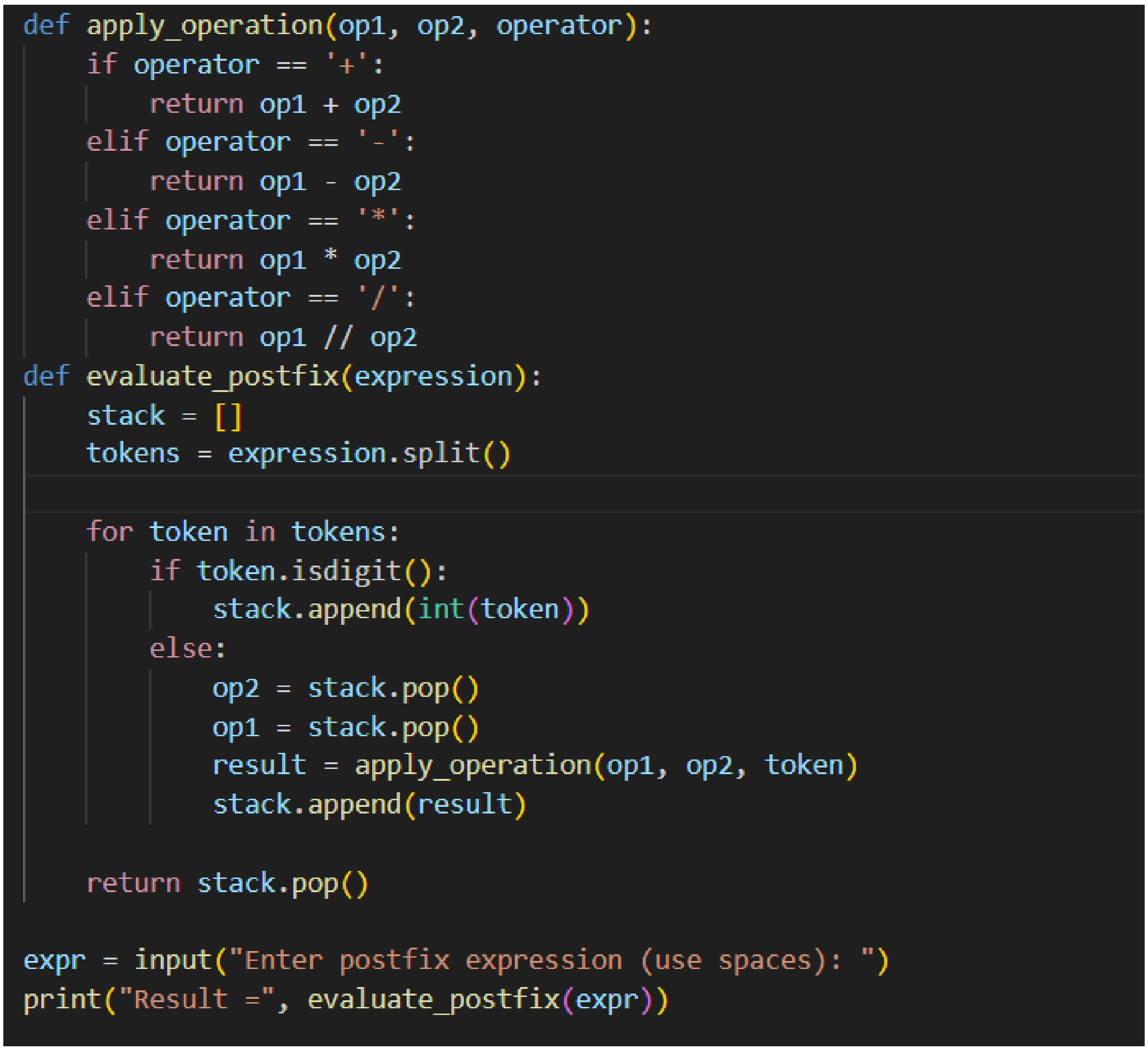
This program supports:
+ , - , * , / operations.

Algorithm

Algorithm to Evaluate Postfix Expression

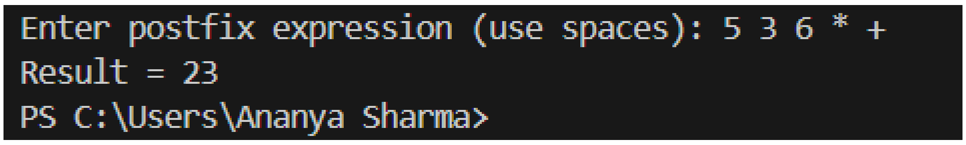
1. Create an empty stack.
2. Read the postfix expression and split it into tokens.
3. For each token:
    - If token is a number → push into stack.
    - If token is an operator:
        - Pop top 2 numbers.
        - Apply the operator.
        - Push the result back.
4. After processing all tokens, pop the final result.
5. Display the result.

CODE



```python
def apply_operation(op1, op2, operator):
    if operator == '+':
        return op1 + op2
    elif operator == '-':
        return op1 - op2
    elif operator == '*':
        return op1 * op2
    elif operator == '/':
        return op1 // op2
def evaluate_postfix(expression):
    stack = []
    tokens = expression.split()

    for token in tokens:
        if token.isdigit():
            stack.append(int(token))
        else:
            op2 = stack.pop()
            op1 = stack.pop()
            result = apply_operation(op1, op2, token)
            stack.append(result)

    return stack.pop()

expr = input("Enter postfix expression (use spaces): ")
print("Result =", evaluate_postfix(expr))
```

OUTPUT



```
Enter postfix expression (use spaces): 5 3 6 * +
Result = 23
PS C:\Users\Ananya Sharma>
```

Result

The postfix expression was successfully evaluated using a stack, and the program correctly computed the final result.

Practical no – 5

AIM- To simulate the Back and Forward buttons of a web browser using two stacks:
one for visited pages and another for forward navigation.

Description

A browser internally uses stacks to maintain navigation history.
When a user visits a new page, it gets pushed onto a stack called history.
When the user presses Back, the last visited page is popped from history and pushed into a second stack called forward_stack.
Pressing Forward pops a page from forward_stack back into history.

This program simulates:

- Visit Page
- Back Navigation
- Forward Navigation (Undo Back)
- Show History
- Show Forward List

Two stacks are used to replicate real browser navigation behavior.

Algorithm

1. Start

2. Initialize two stacks

- history to store visited pages
- forward_stack to store pages available to go forward

3.Display Menu

4. Visit Page

1. Input page name

2. Push page into history
3. Clear forward_stack because forward history becomes invalid
4. Display “Visited: page”

5. Back

1. If history has 1 or fewer pages → cannot go back
2. Pop top page from history
3. Push that page into forward_stack
4. Show new current page (history[-1])

6.Forward

1. If forward_stack is empty → cannot go forward
2. Pop top page from forward_stack
3. Push that page into history
4. Show page navigated forward to

7. **Show History**

- If empty → print message
- Else print pages as:
  Page1 -> Page2 -> Page3

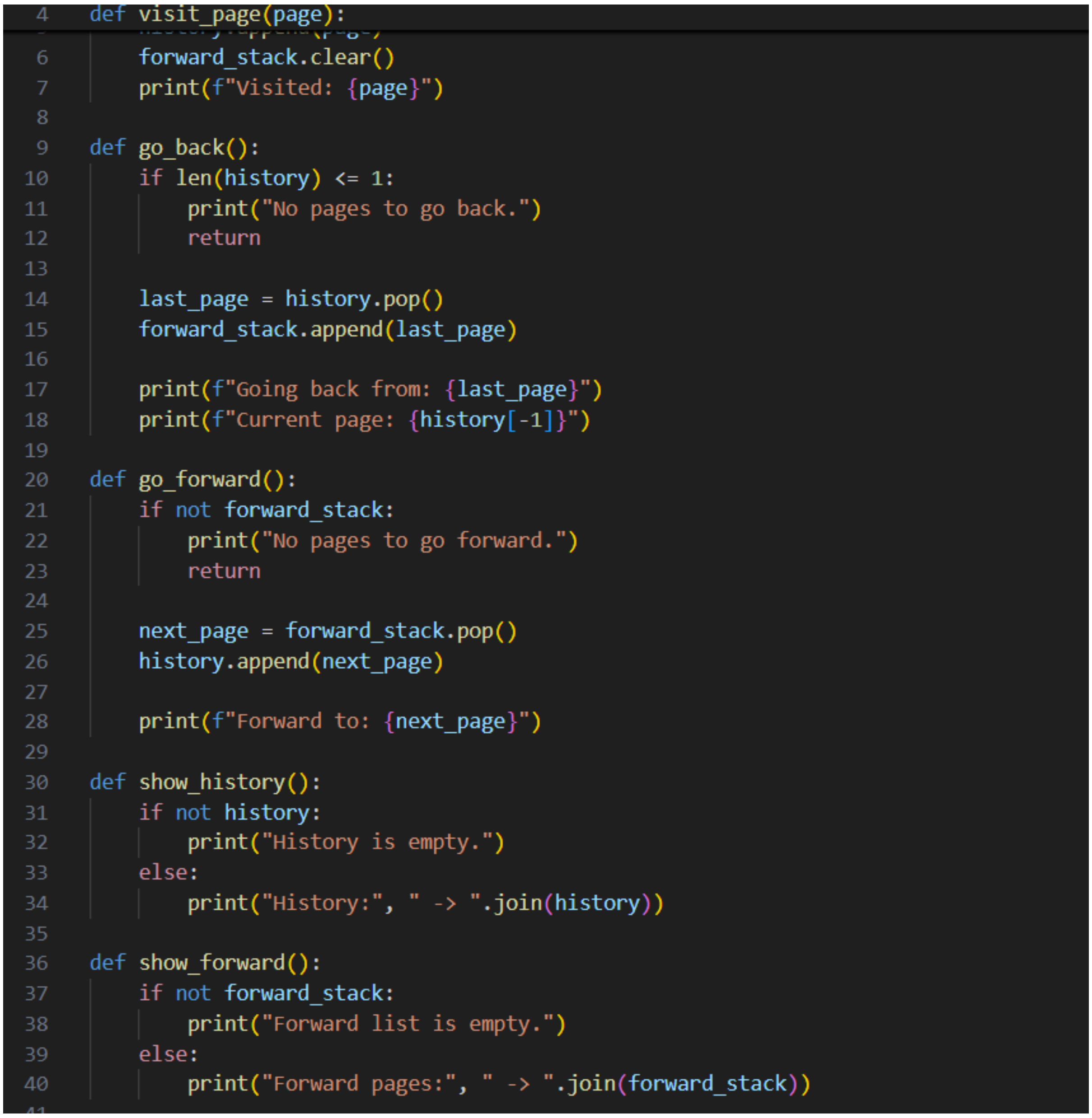
8. **Show Forward List**

- If empty → print message
- Else display forward stack

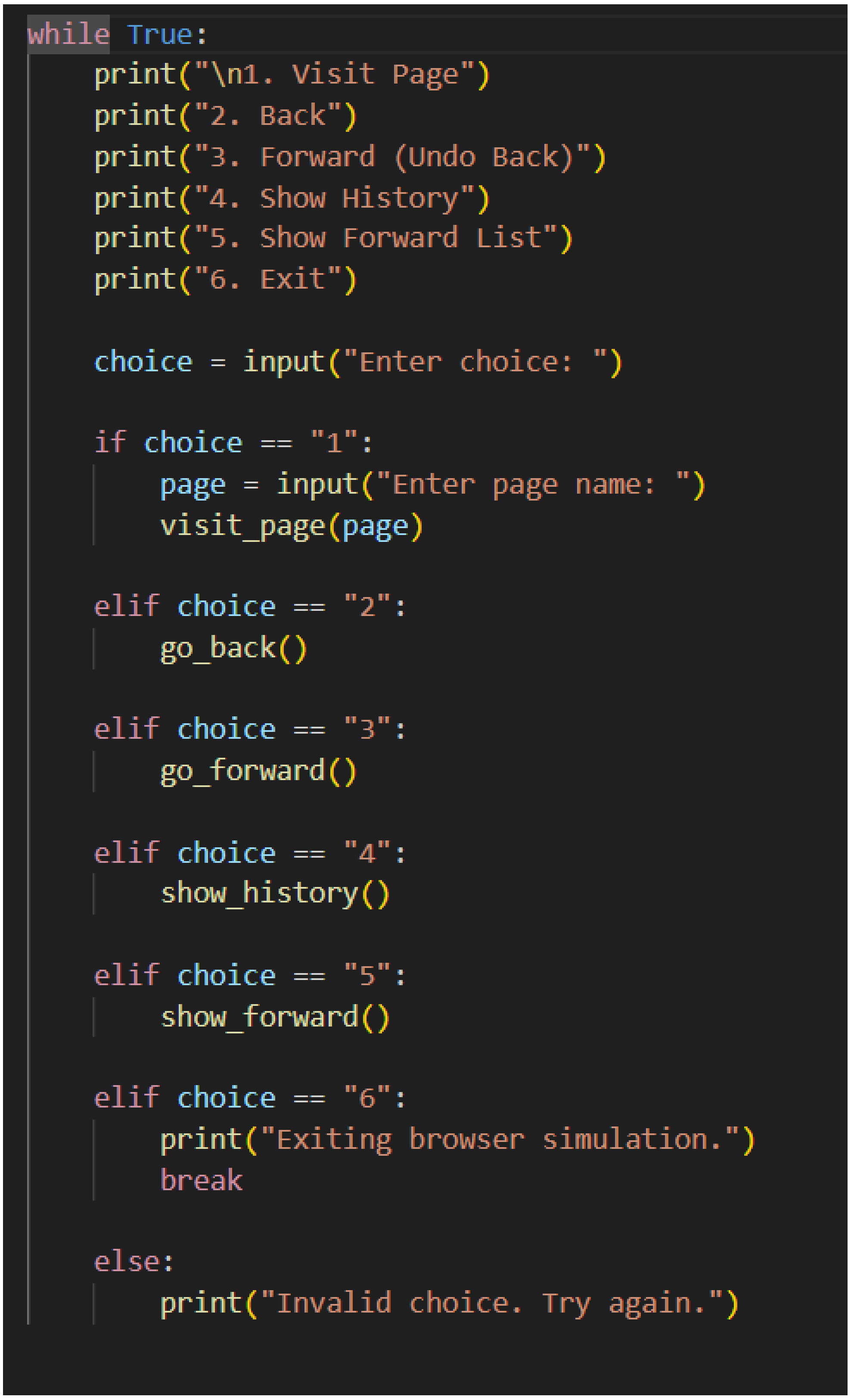
9. **Exit**

Stop the program

CODE

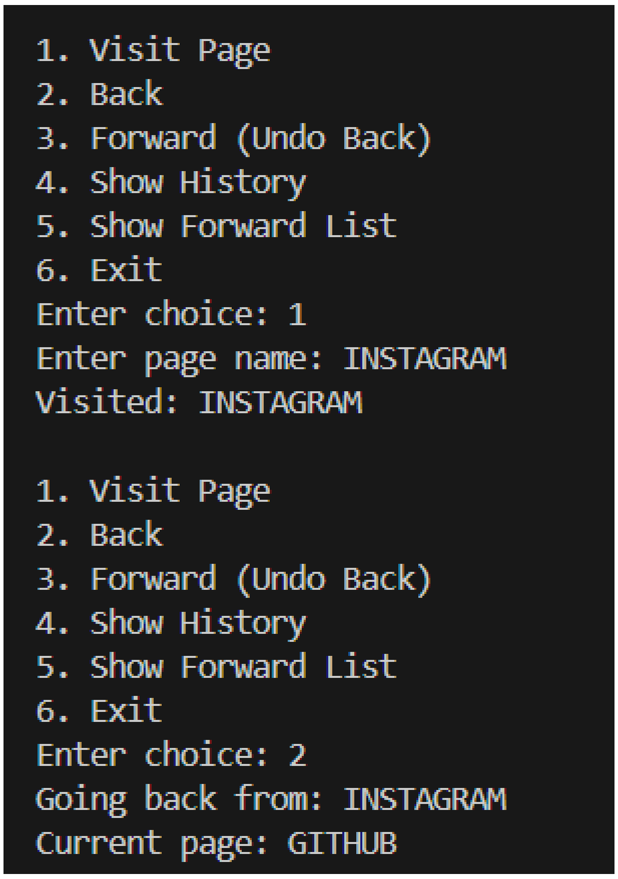


```python
def visit_page(page):
    history.append(page)
    forward_stack.clear()
    print(f"Visited: {page}")

def go_back():
    if len(history) <= 1:
        print("No pages to go back.")
        return

    last_page = history.pop()
    forward_stack.append(last_page)

    print(f"Going back from: {last_page}")
    print(f"Current page: {history[-1]}")

def go_forward():
    if not forward_stack:
        print("No pages to go forward.")
        return

    next_page = forward_stack.pop()
    history.append(next_page)

    print(f"Forward to: {next_page}")

def show_history():
    if not history:
        print("History is empty.")
    else:
        print("History:", " -> ".join(history))

def show_forward():
    if not forward_stack:
        print("Forward list is empty.")
    else:
        print("Forward pages:", " -> ".join(forward_stack))
```



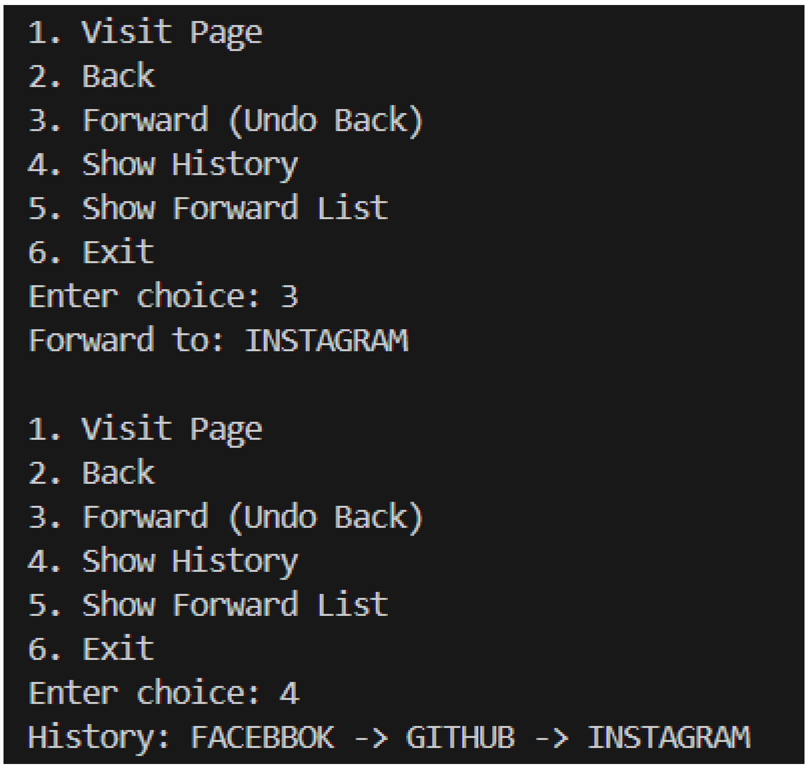
```python
while True:
    print("\n1. Visit Page")
    print("2. Back")
    print("3. Forward (Undo Back)")
    print("4. Show History")
    print("5. Show Forward List")
    print("6. Exit")

    choice = input("Enter choice: ")

    if choice == "1":
        page = input("Enter page name: ")
        visit_page(page)

    elif choice == "2":
        go_back()

    elif choice == "3":
        go_forward()

    elif choice == "4":
        show_history()

    elif choice == "5":
        show_forward()

    elif choice == "6":
        print("Exiting browser simulation.")
        break

    else:
        print("Invalid choice. Try again.")
```

OUTPUT



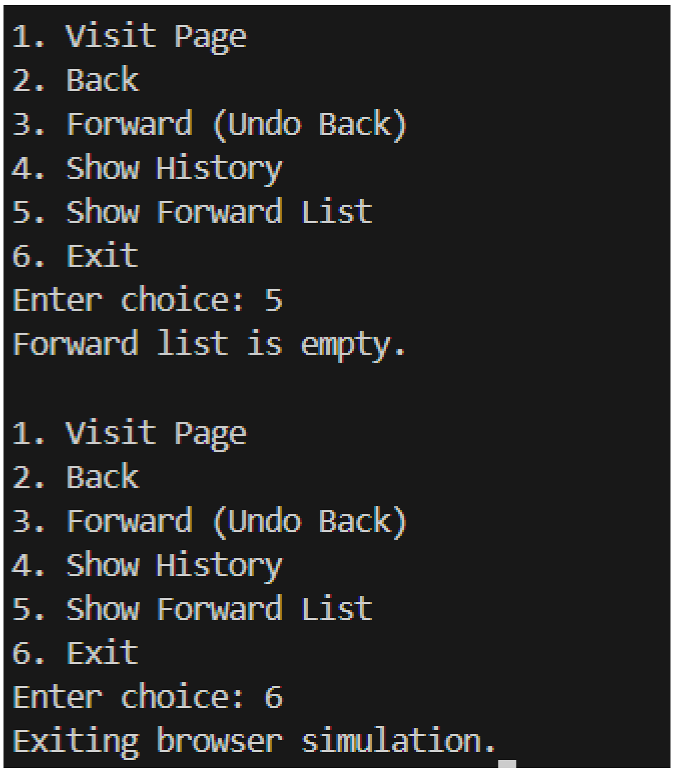
```
1. Visit Page
2. Back
3. Forward (Undo Back)
4. Show History
5. Show Forward List
6. Exit
Enter choice: 1
Enter page name: INSTAGRAM
Visited: INSTAGRAM

1. Visit Page
2. Back
3. Forward (Undo Back)
4. Show History
5. Show Forward List
6. Exit
Enter choice: 2
Going back from: INSTAGRAM
Current page: GITHUB
```

Sample Output 2



```
1. Visit Page
2. Back
3. Forward (Undo Back)
4. Show History
5. Show Forward List
6. Exit
Enter choice: 3
Forward to: INSTAGRAM

1. Visit Page
2. Back
3. Forward (Undo Back)
4. Show History
5. Show Forward List
6. Exit
Enter choice: 4
History: FACEBBOK -> GITHUB -> INSTAGRAM
```

Sample Output 3



```
1. Visit Page
2. Back
3. Forward (Undo Back)
4. Show History
5. Show Forward List
6. Exit
Enter choice: 5
Forward list is empty.

1. Visit Page
2. Back
3. Forward (Undo Back)
4. Show History
5. Show Forward List
6. Exit
Enter choice: 6
Exiting browser simulation.
```

Result

The browser back and forward simulation was successfully implemented using two stacks.
The program correctly handled visiting a page, going back, going forward, and displaying browsing history.

Practical No – 6

AIM- To implement three different applications of stack—reversing a string, checking balanced parentheses, and performing undo operation—using a single menu-driven program in Python.

Description

A stack is a linear data structure that follows the LIFO (Last-In, First-Out) principle.
This practical demonstrates three real-life applications of stacks:

1. Reverse a String

Characters of the string are pushed into a stack and then popped out to reverse the order.

2. Check Balanced Parentheses

Opening brackets are pushed on the stack.
Closing brackets are matched with the top of the stack.
If all brackets match correctly and stack becomes empty, the expression is balanced.

3. Undo Operation (Text Editor)

Each typed character is pushed into the stack.
Undo operation removes the last character by popping the stack.

The complete program is menu-driven, allowing the user to choose any operation.

Algorithm

Main Menu Algorithm

1. Start
2. Display menu with options:
    - Reverse String
    - Check Balanced Parentheses

    - Undo Operation
    - Exit
3. Take user's choice
4. Call the corresponding function
5. Repeat until Exit option is selected

Algorithm for Reverse String

1. Input a string
2. Create an empty stack
3. For each character in the string → push character onto stack
4. While stack not empty → pop each character and add to result
5. Display reversed string

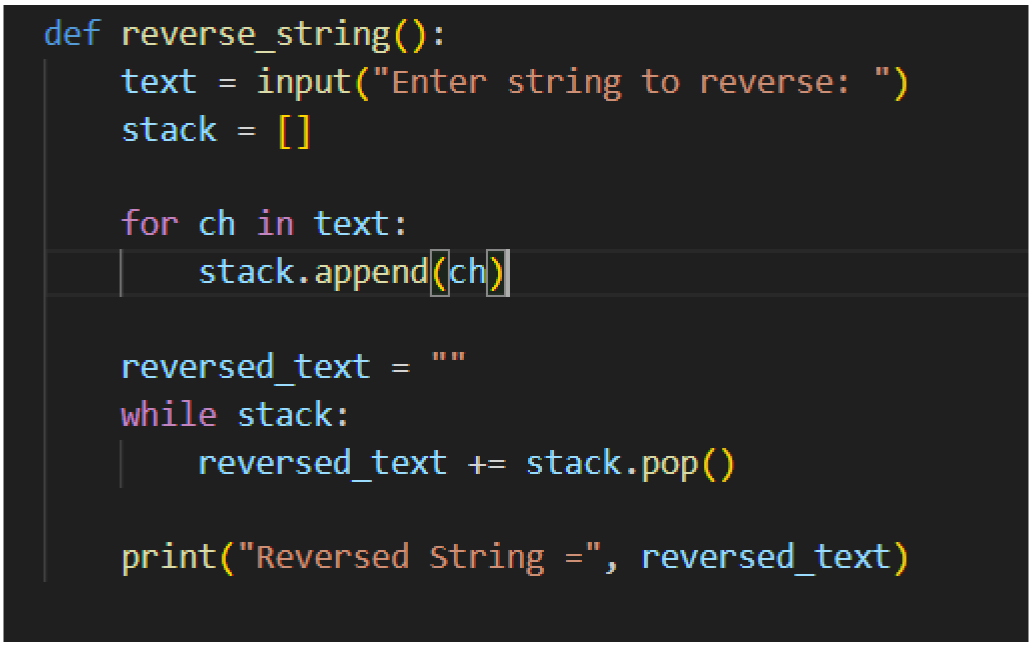
Algorithm for Balanced Parentheses

1. Input an expression
2. Create an empty stack
3. For each character:
    - If opening bracket → push
    - If closing bracket →
        - If stack empty → Not Balanced
        - Else pop and check if types match
4. After traversal:
    - If stack empty → Balanced
    - Else → Not Balanced

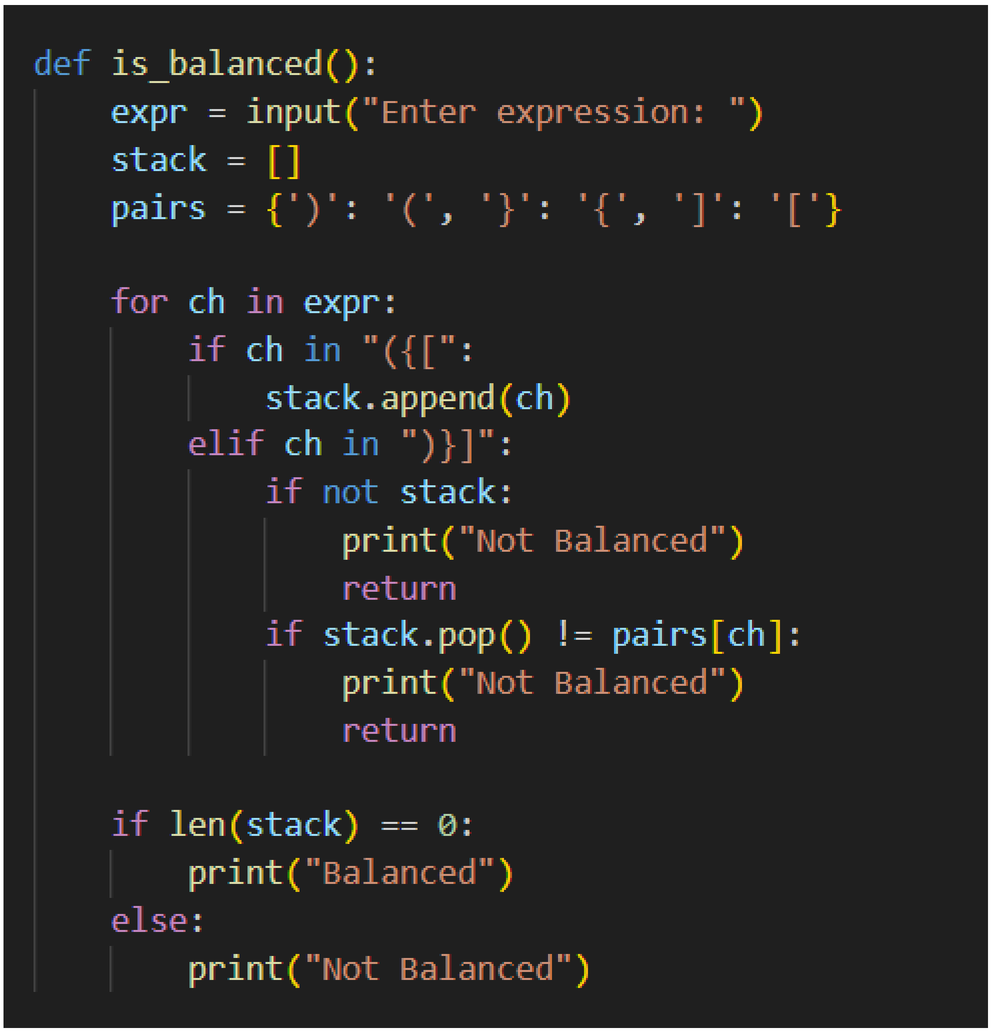
Algorithm for Undo Operation

1. Initialize empty stack
2. Display submenu:
    - Type character → push
    - Undo → pop
    - Show text → print stack contents
    - Exit module
3. Perform chosen action
4. Repeat until Exit

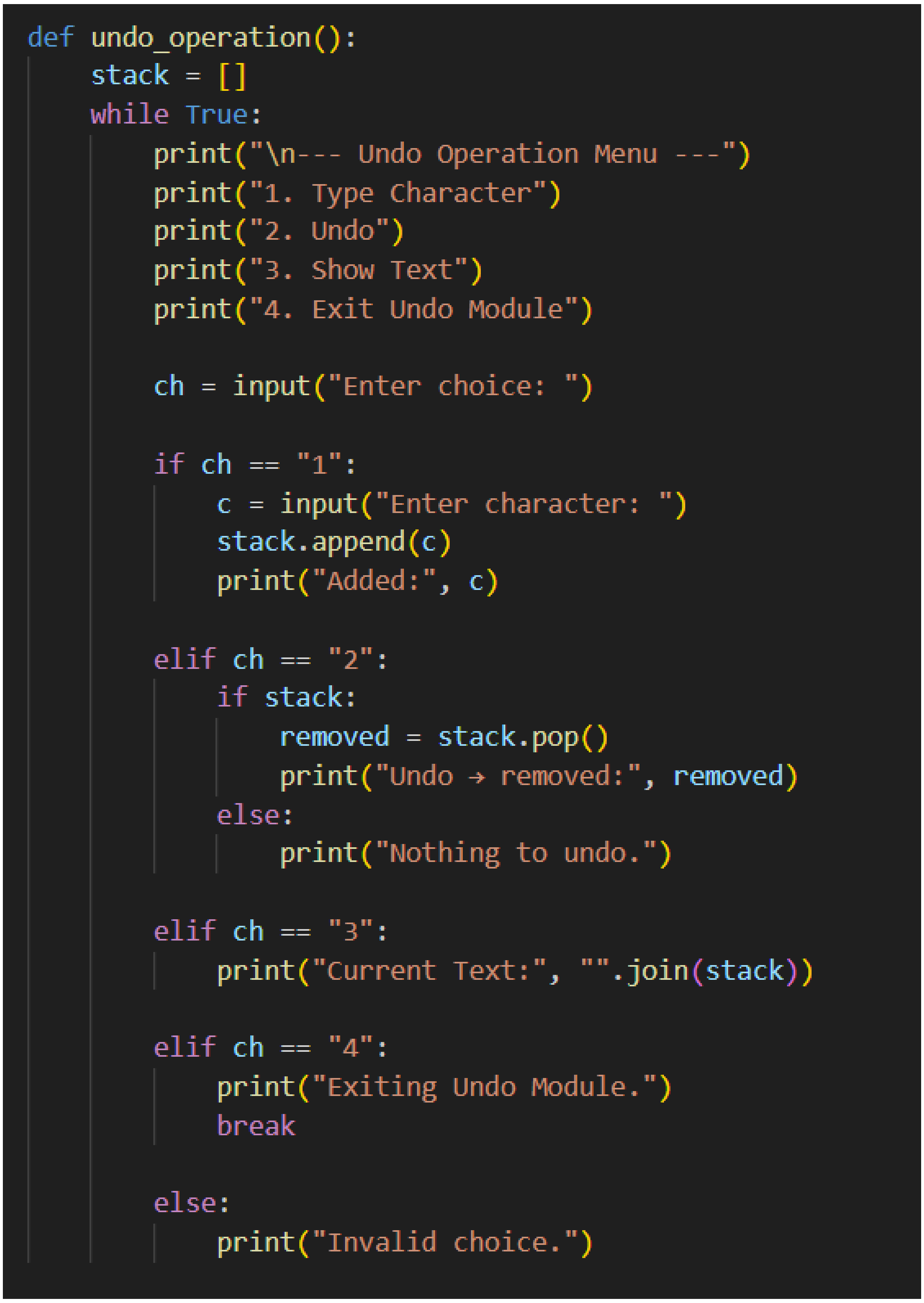
CODE

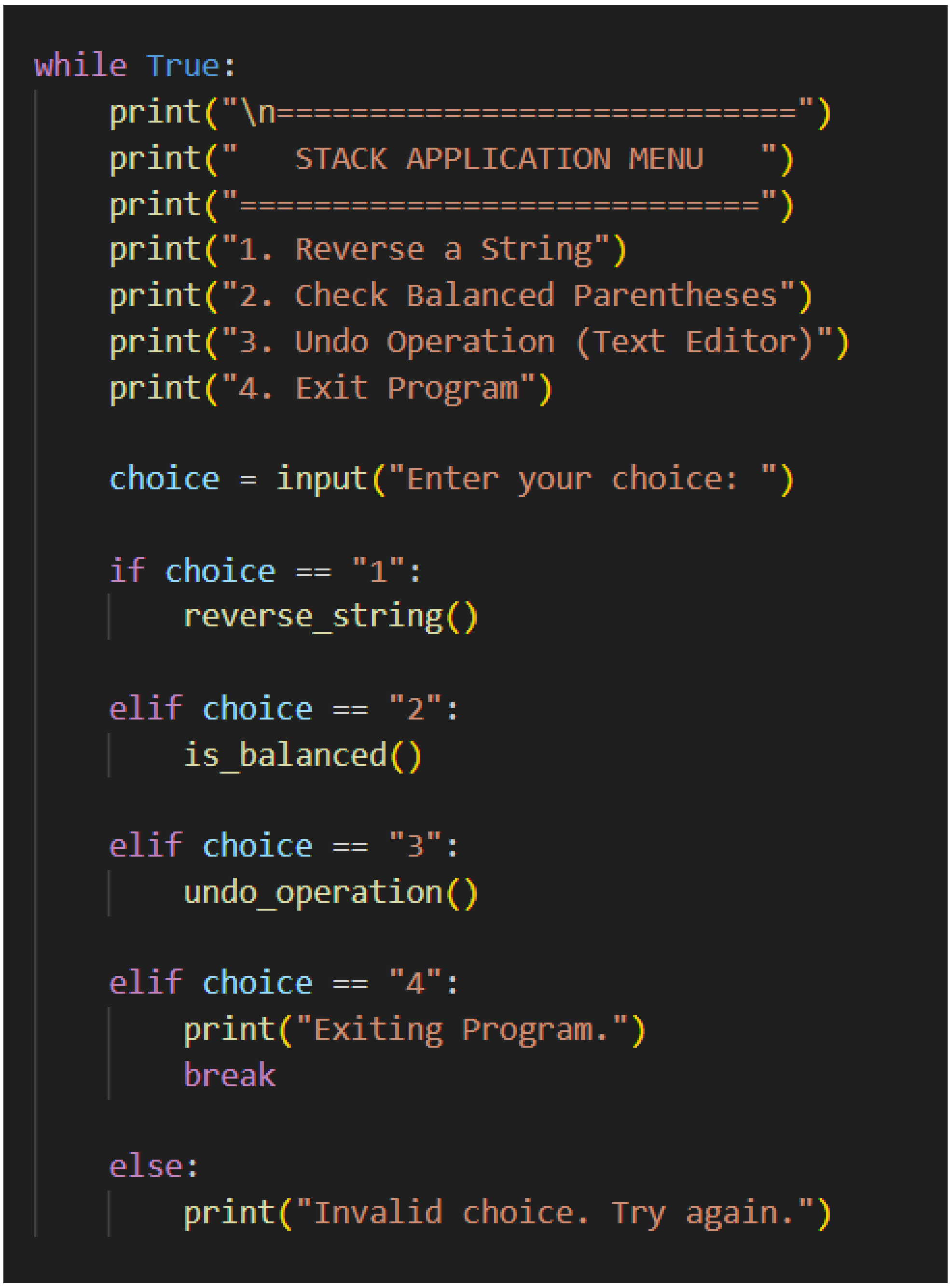


```python
def reverse_string():
    text = input("Enter string to reverse: ")
    stack = []

    for ch in text:
        stack.append(ch)

    reversed_text = ""
    while stack:
        reversed_text += stack.pop()

    print("Reversed String =", reversed_text)
```



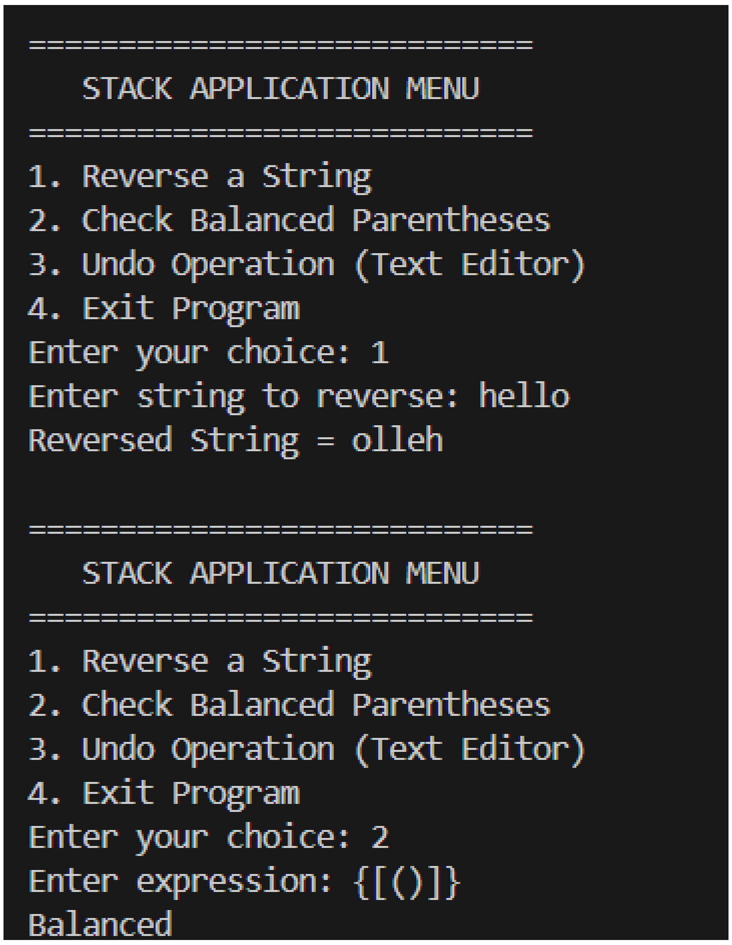
```python
def is_balanced():
    expr = input("Enter expression: ")
    stack = []
    pairs = {')': '(', '}': '{', ']': '['}

    for ch in expr:
        if ch in "({[":
            stack.append(ch)
        elif ch in ")}]":
            if not stack:
                print("Not Balanced")
                return
            if stack.pop() != pairs[ch]:
                print("Not Balanced")
                return

    if len(stack) == 0:
        print("Balanced")
    else:
        print("Not Balanced")
```


```python
def undo_operation():
    stack = []
    while True:
        print("\n--- Undo Operation Menu ---")
        print("1. Type Character")
        print("2. Undo")
        print("3. Show Text")
        print("4. Exit Undo Module")

        ch = input("Enter choice: ")

        if ch == "1":
            c = input("Enter character: ")
            stack.append(c)
            print("Added:", c)

        elif ch == "2":
            if stack:
                removed = stack.pop()
                print("Undo → removed:", removed)
            else:
                print("Nothing to undo.")

        elif ch == "3":
            print("Current Text:", "".join(stack))

        elif ch == "4":
            print("Exiting Undo Module.")
            break

        else:
            print("Invalid choice.")
```



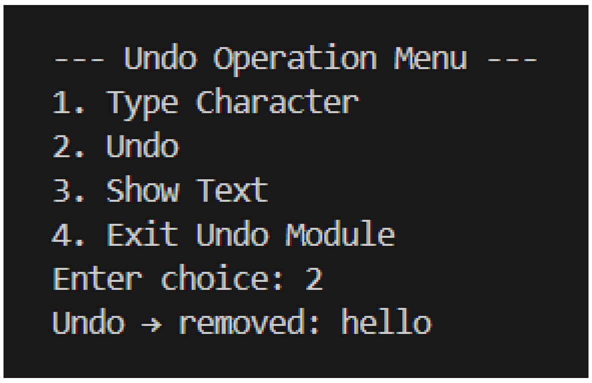
```python
while True:
    print("\n============================")
    print("   STACK APPLICATION MENU   ")
    print("============================")
    print("1. Reverse a String")
    print("2. Check Balanced Parentheses")
    print("3. Undo Operation (Text Editor)")
    print("4. Exit Program")

    choice = input("Enter your choice: ")

    if choice == "1":
        reverse_string()

    elif choice == "2":
        is_balanced()

    elif choice == "3":
        undo_operation()

    elif choice == "4":
        print("Exiting Program.")
        break

    else:
        print("Invalid choice. Try again.")
```

OUTPUT



```
=============================
   STACK APPLICATION MENU
=============================
1. Reverse a String
2. Check Balanced Parentheses
3. Undo Operation (Text Editor)
4. Exit Program
Enter your choice: 1
Enter string to reverse: hello
Reversed String = olleh

=============================
   STACK APPLICATION MENU
=============================
1. Reverse a String
2. Check Balanced Parentheses
3. Undo Operation (Text Editor)
4. Exit Program
Enter your choice: 2
Enter expression: {[()]}
Balanced
```

Sample Output 2



```
 --- Undo Operation Menu ---
 1. Type Character
 2. Undo
 3. Show Text
 4. Exit Undo Module
 Enter choice: 2
 Undo → removed: hello
```

Result-

The combined stack application program was successfully implemented.

Practical No - 7

AIM- To implement a singly linked list and perform insertion, deletion, searching, and display operations.

Description

A singly linked list stores elements in nodes where each node contains:

- data
- pointer to next node

This practical performs the following operations:

- Insert node at beginning
- Insert node at end
- Insert node at a given position
- Delete node from beginning
- Delete node from end
- Delete node from a given position
- Search a value in the linked list
- Display linked list

The program is menu-driven.

Algorithm

1. Insert at Beginning

```
1. Create new node
2. new.next = head
3. head = new
```

---

2. Insert at End

    1. Create new node
    2. If list empty → head = new
    3. Traverse to last node
    4. last.next = new

---

3. Insert at Position

    1. If pos = 1 → insert at beginning
    2. Else traverse to (pos–1) node
    3. Insert new node in between

---

4. Delete from Beginning

    1. If list empty → show message
    2. Else head = head.next

---

5. Delete from End

    1. If list empty → message
    2. If only one node → head = NULL
    3. Else reach second last node
    4. second_last.next = NULL

---

6. Delete from Position

    1. If pos = 1 → delete beginning
    2. Else traverse to (pos–1) node

3. Remove next node

---

7. Search Element

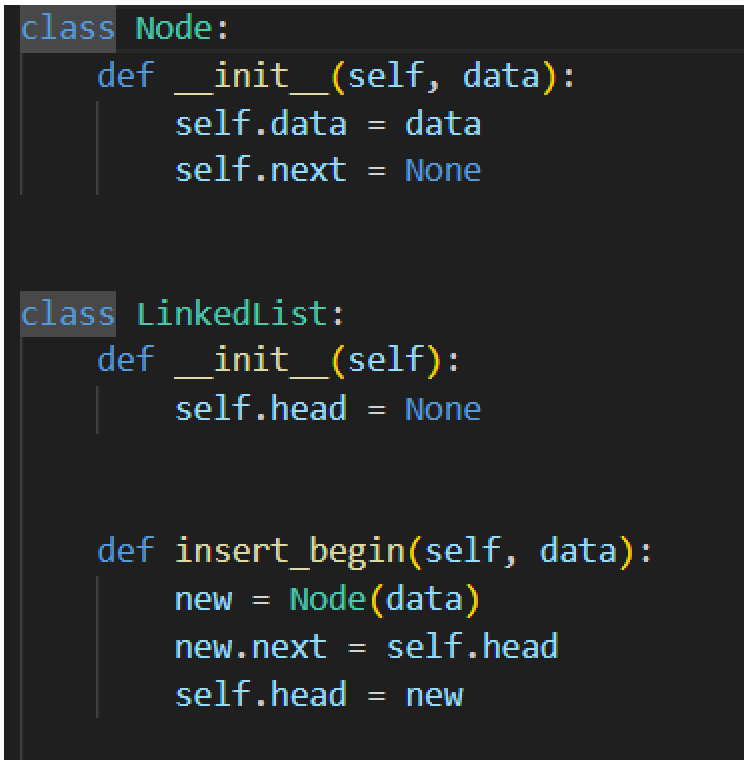
1. Start from head
2. Traverse nodes
3. If node.data == value → found
4. If end reached → not found

---

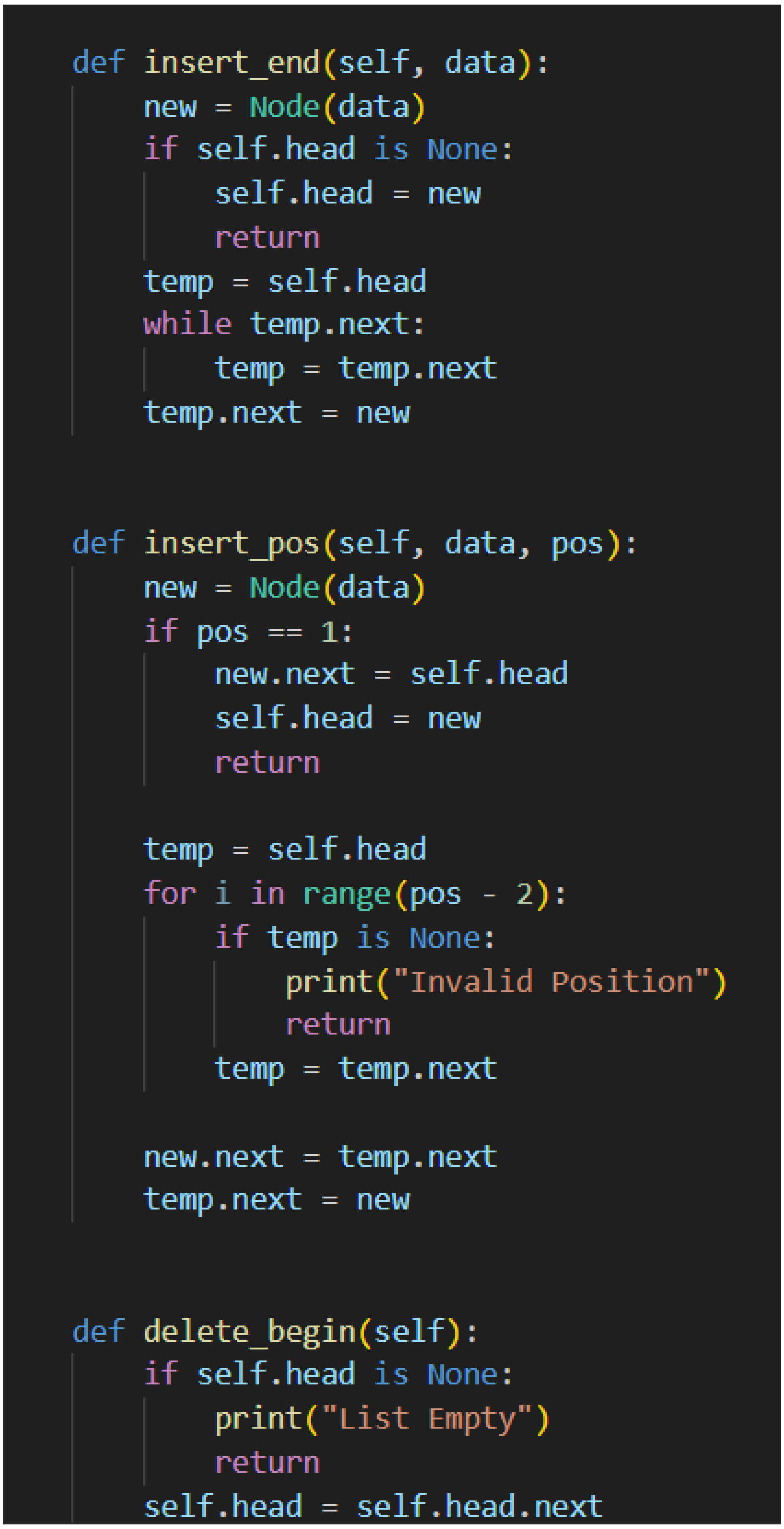
8. Display List

Traverse and print all nodes.

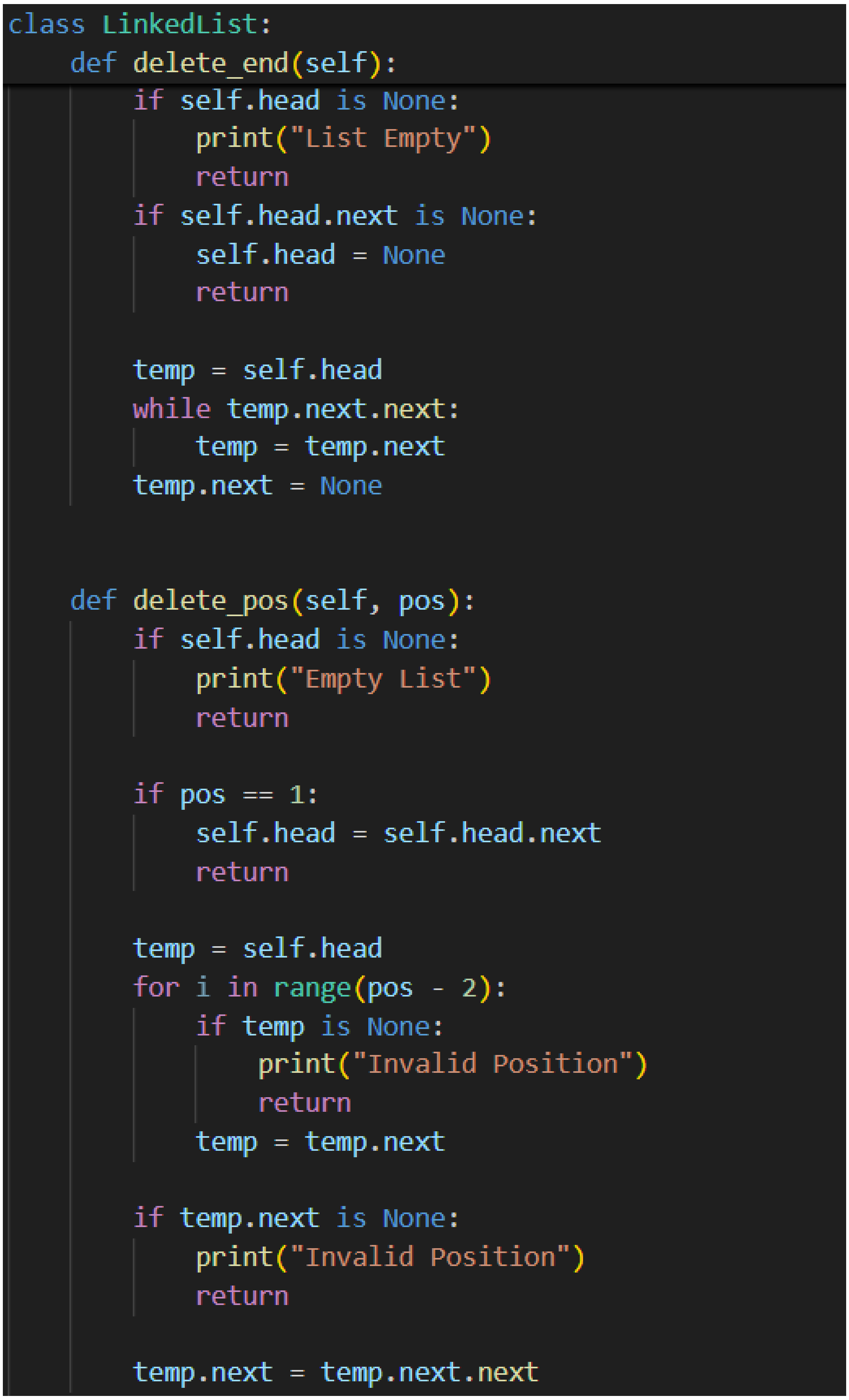
CODE



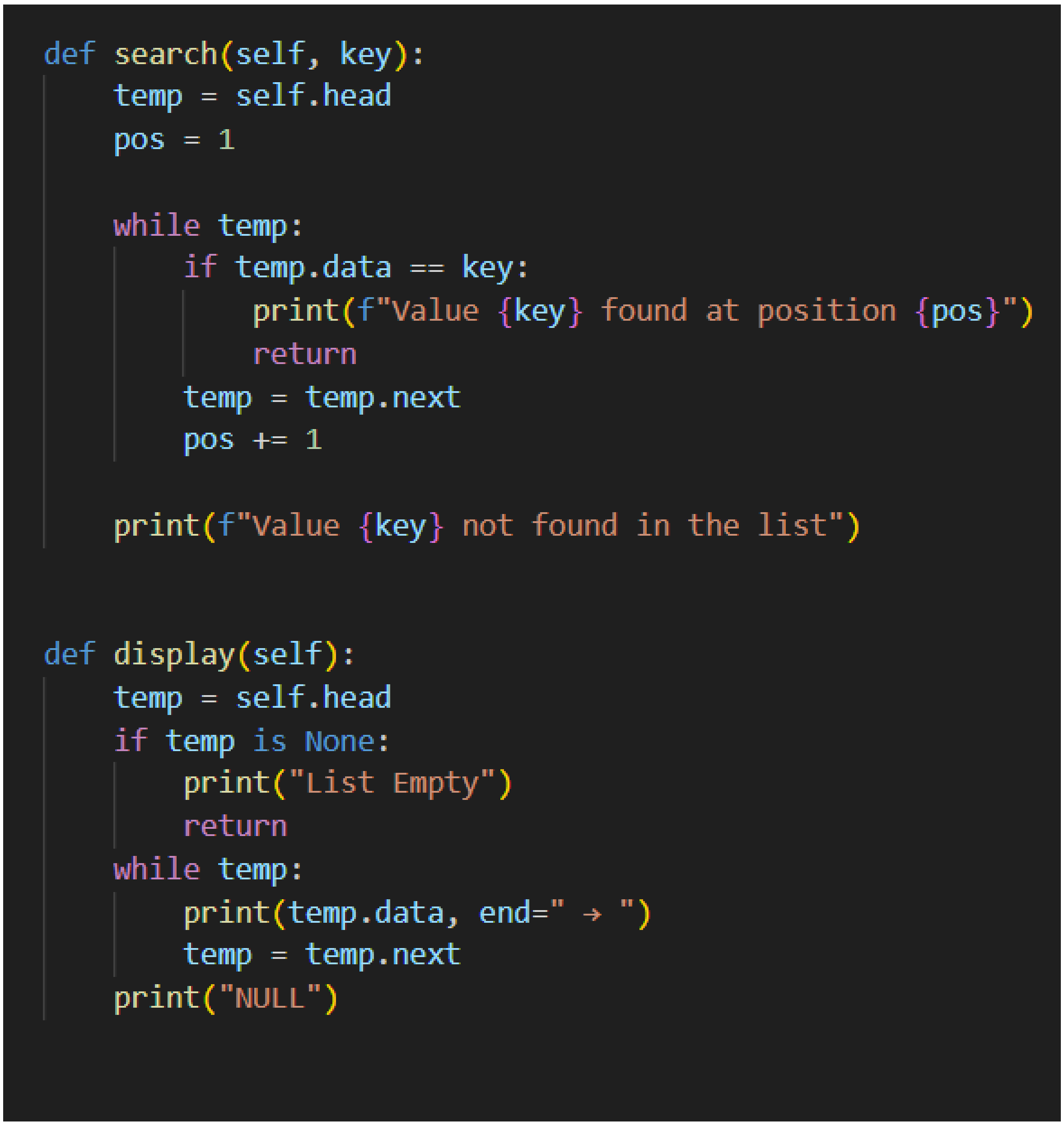
```python
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None


class LinkedList:
    def __init__(self):
        self.head = None


    def insert_begin(self, data):
        new = Node(data)
        new.next = self.head
        self.head = new
```



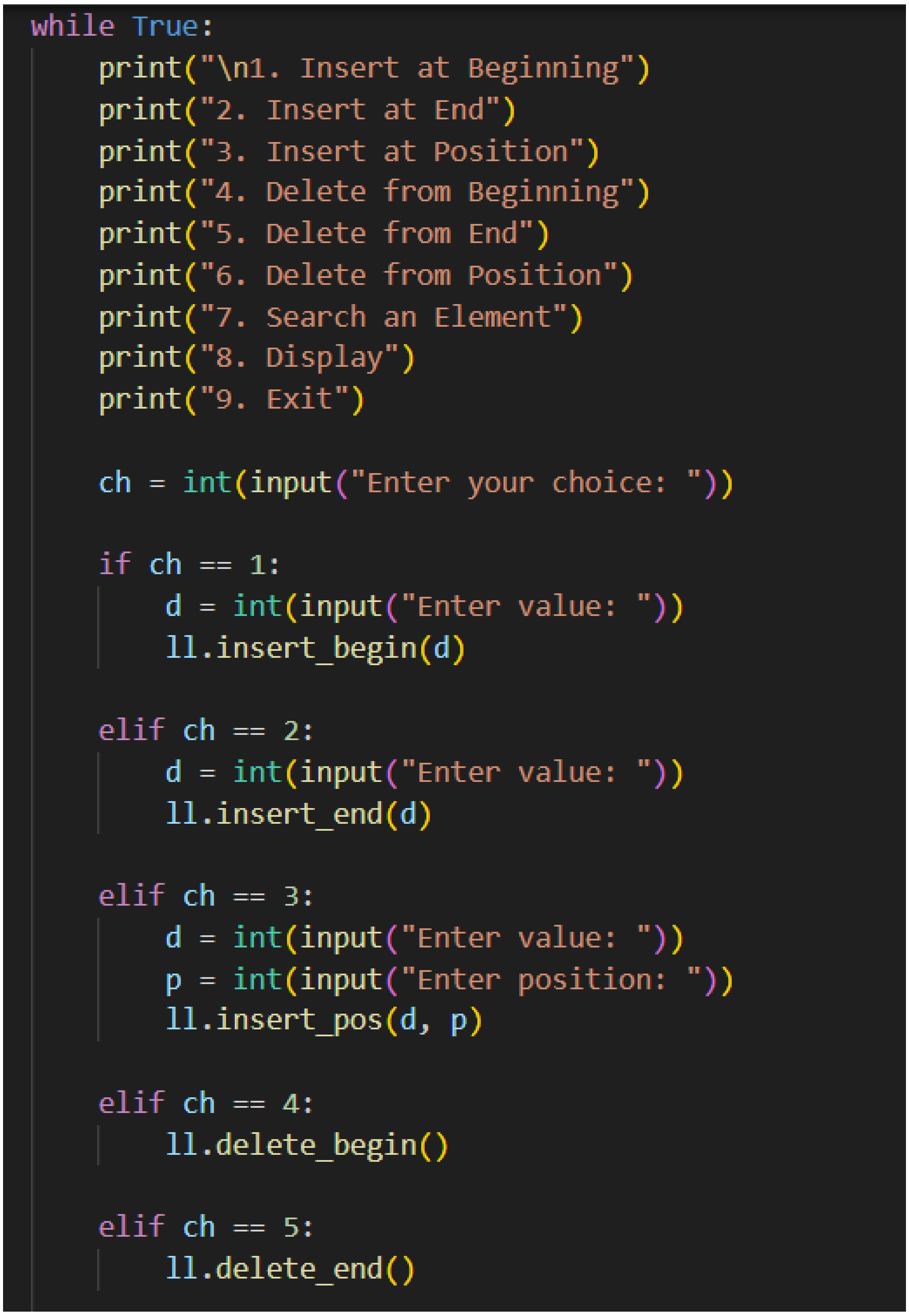
```python
def insert_end(self, data):
    new = Node(data)
    if self.head is None:
        self.head = new
        return
    temp = self.head
    while temp.next:
        temp = temp.next
    temp.next = new


def insert_pos(self, data, pos):
    new = Node(data)
    if pos == 1:
        new.next = self.head
        self.head = new
        return

    temp = self.head
    for i in range(pos - 2):
        if temp is None:
            print("Invalid Position")
            return
        temp = temp.next

    new.next = temp.next
    temp.next = new


def delete_begin(self):
    if self.head is None:
        print("List Empty")
        return
    self.head = self.head.next
```

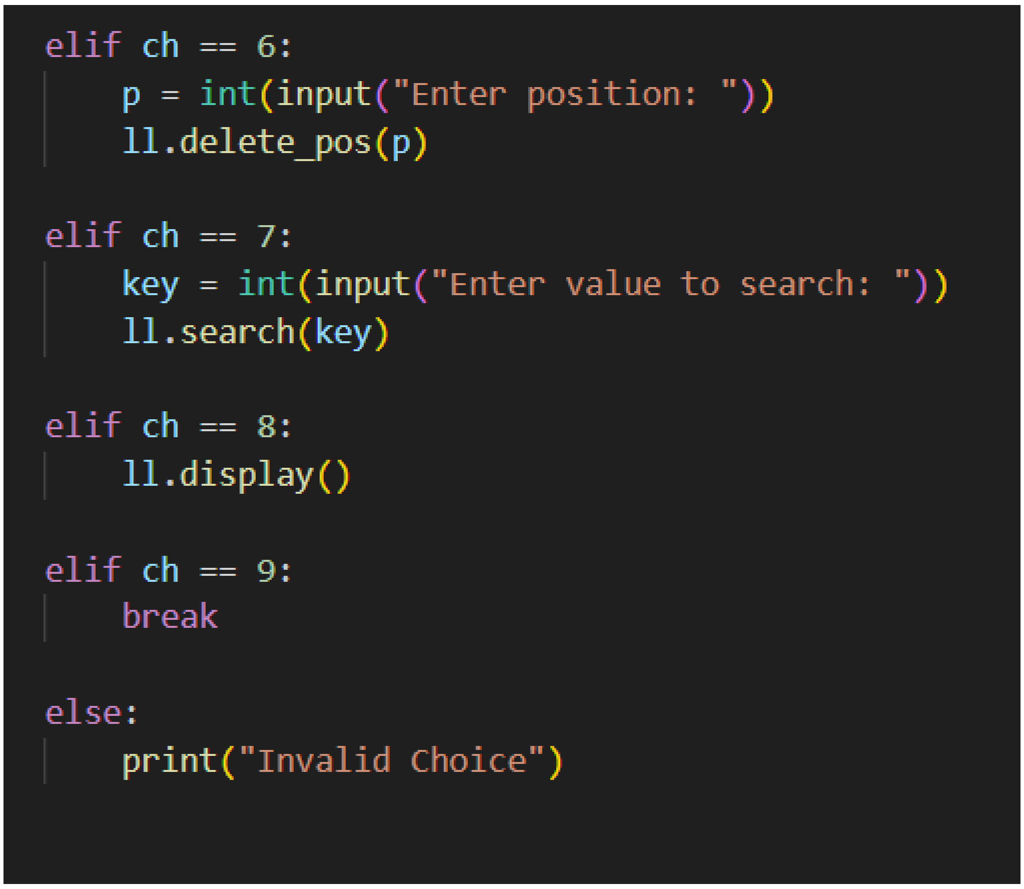


```python
class LinkedList:
    def delete_end(self):
        if self.head is None:
            print("List Empty")
            return
        if self.head.next is None:
            self.head = None
            return

        temp = self.head
        while temp.next.next:
            temp = temp.next
        temp.next = None


    def delete_pos(self, pos):
        if self.head is None:
            print("Empty List")
            return

        if pos == 1:
            self.head = self.head.next
            return

        temp = self.head
        for i in range(pos - 2):
            if temp is None:
                print("Invalid Position")
                return
            temp = temp.next

        if temp.next is None:
            print("Invalid Position")
            return

        temp.next = temp.next.next
```



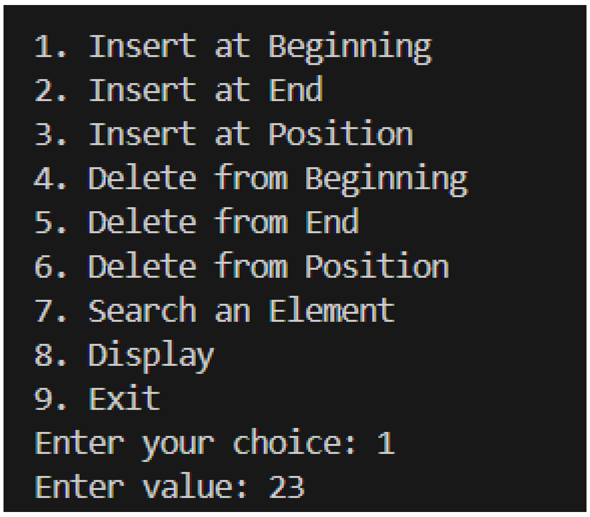
```python
def search(self, key):
    temp = self.head
    pos = 1

    while temp:
        if temp.data == key:
            print(f"Value {key} found at position {pos}")
            return
        temp = temp.next
        pos += 1

    print(f"Value {key} not found in the list")


def display(self):
    temp = self.head
    if temp is None:
        print("List Empty")
        return
    while temp:
        print(temp.data, end=" → ")
        temp = temp.next
    print("NULL")
```


```python
while True:
    print("\n1. Insert at Beginning")
    print("2. Insert at End")
    print("3. Insert at Position")
    print("4. Delete from Beginning")
    print("5. Delete from End")
    print("6. Delete from Position")
    print("7. Search an Element")
    print("8. Display")
    print("9. Exit")

    ch = int(input("Enter your choice: "))

    if ch == 1:
        d = int(input("Enter value: "))
        ll.insert_begin(d)

    elif ch == 2:
        d = int(input("Enter value: "))
        ll.insert_end(d)

    elif ch == 3:
        d = int(input("Enter value: "))
        p = int(input("Enter position: "))
        ll.insert_pos(d, p)

    elif ch == 4:
        ll.delete_begin()

    elif ch == 5:
        ll.delete_end()
```

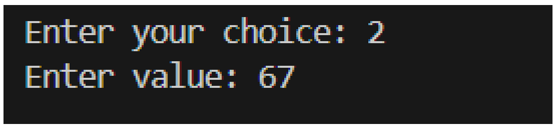


```python
elif ch == 6:
    p = int(input("Enter position: "))
    ll.delete_pos(p)

elif ch == 7:
    key = int(input("Enter value to search: "))
    ll.search(key)

elif ch == 8:
    ll.display()

elif ch == 9:
    break

else:
    print("Invalid Choice")
```

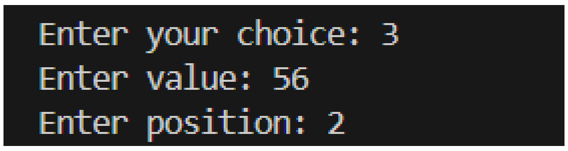
OUTPUT



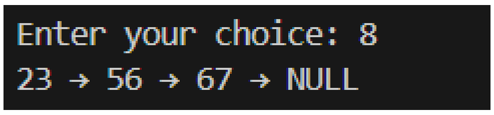
```
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Search an Element
8. Display
9. Exit
Enter your choice: 1
Enter value: 23
```



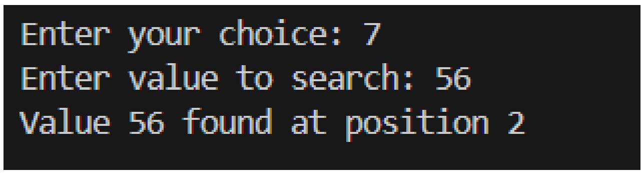
```
Enter your choice: 2
Enter value: 67
```



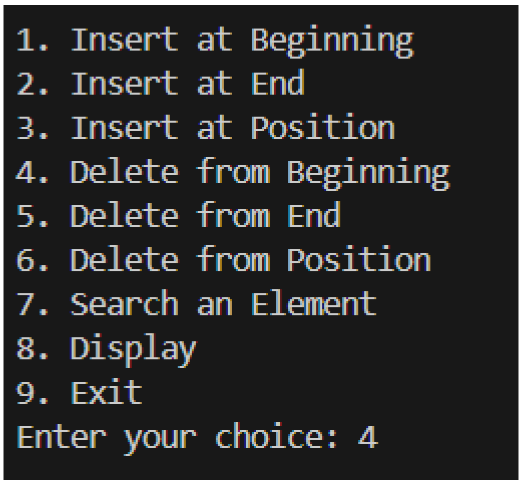
```
Enter your choice: 3
Enter value: 56
Enter position: 2
```



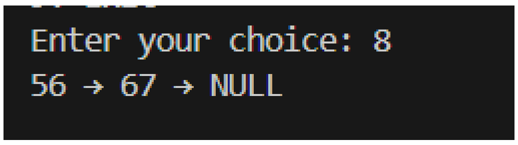
```
Enter your choice: 8
23 → 56 → 67 → NULL
```



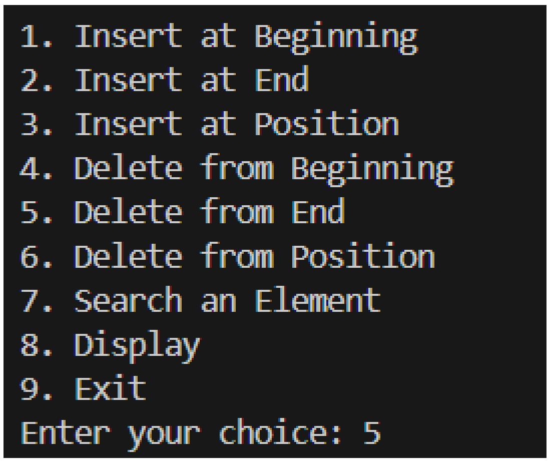
```
Enter your choice: 7
Enter value to search: 56
Value 56 found at position 2
```



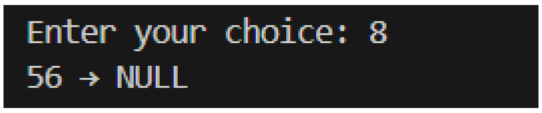
```
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Search an Element
8. Display
9. Exit
Enter your choice: 4
```



```
Enter your choice: 8
56 → 67 → NULL
```



```
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Search an Element
8. Display
9. Exit
Enter your choice: 5
```



```
Enter your choice: 8
56 → NULL
```

Result –

The singly linked list was successfully implemented with insert, delete, search, and display operations.
The program works correctly for all cases.

Practical No –8

AIM - To implement insertion and deletion operations on a Circular Linked List using user input in Python

Description

A Circular Linked List is a linked list where the last node points back to the head node, forming a circular structure.

In this program, a menu-driven user input system is used to perform operations:

Insert at Beginning

Insert at End

Delete from Beginning

Delete from End

Display the List

Algorithm

Algorithm for Insertion at Beginning

```
Start
Create new node
    If list empty → head = new node; new node points to itself
    Else traverse to last node
   Point new node → head
   Point last node → new node
   Make new node the head
Stop
```

Algorithm for Insertion at End

Start

Create new node

If list empty → head = new node; new node points to itself

Else traverse to last node

Insert new node after last

Link new node → head

Stop

Algorithm for Deletion at Beginning

Start

If list empty → Stop

If only one node → head = None

Else traverse to last node

Point last node → head.next

Update head = head.next

Stop

Algorithm for Deletion at End

Start

If list empty → Stop

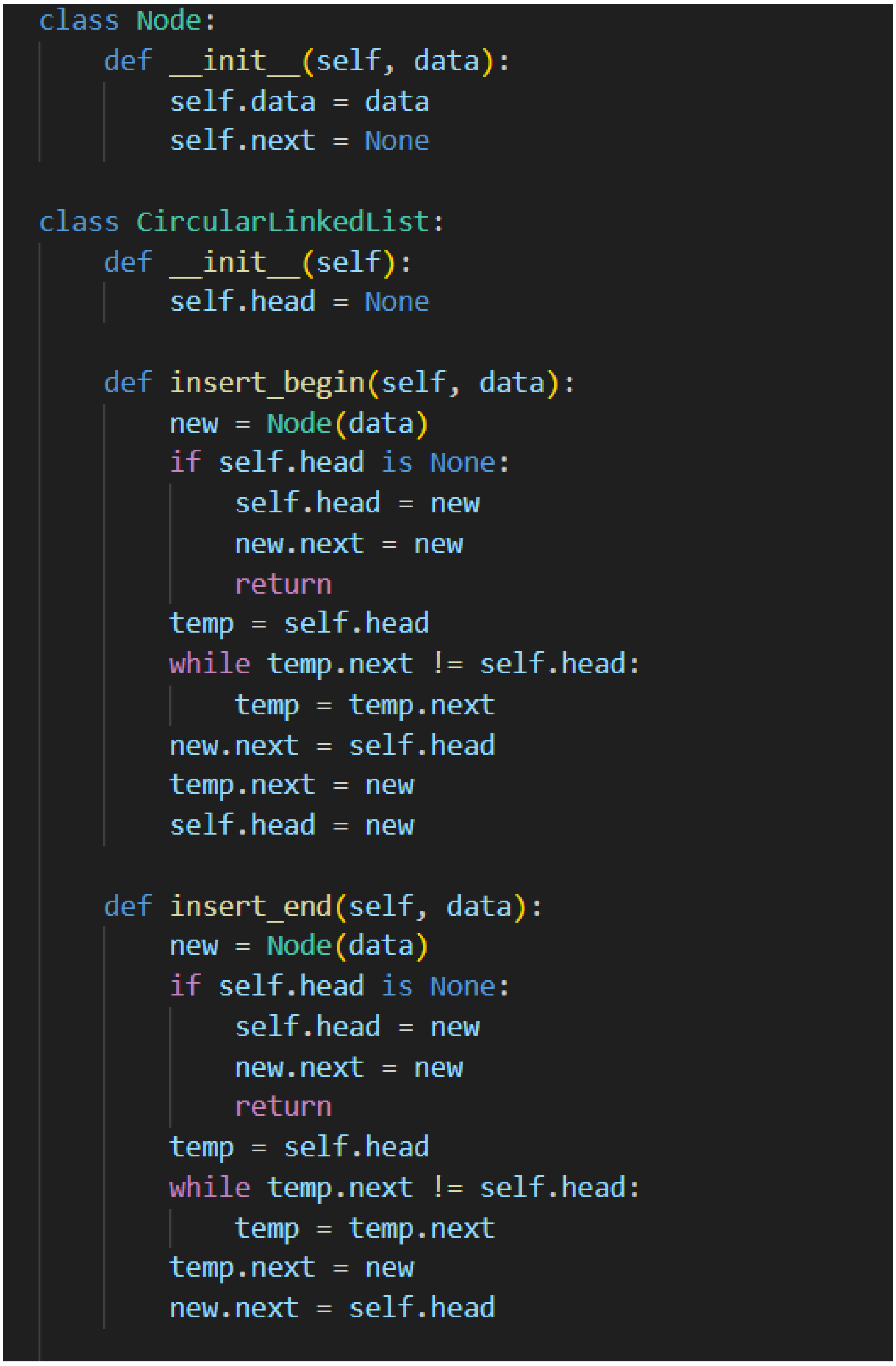
If only one node → head = None

Traverse to second last node

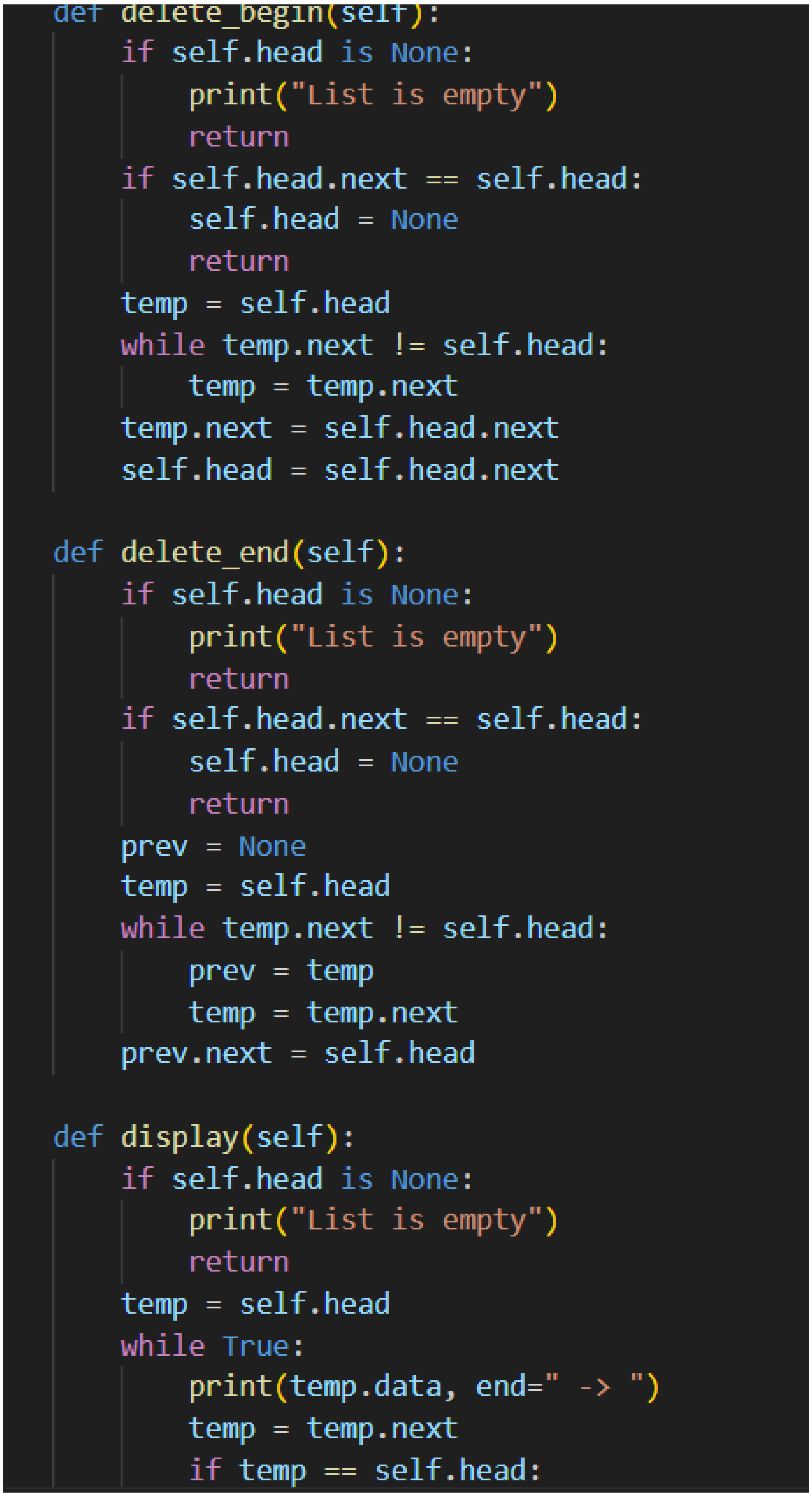
Point second last → head

Stop

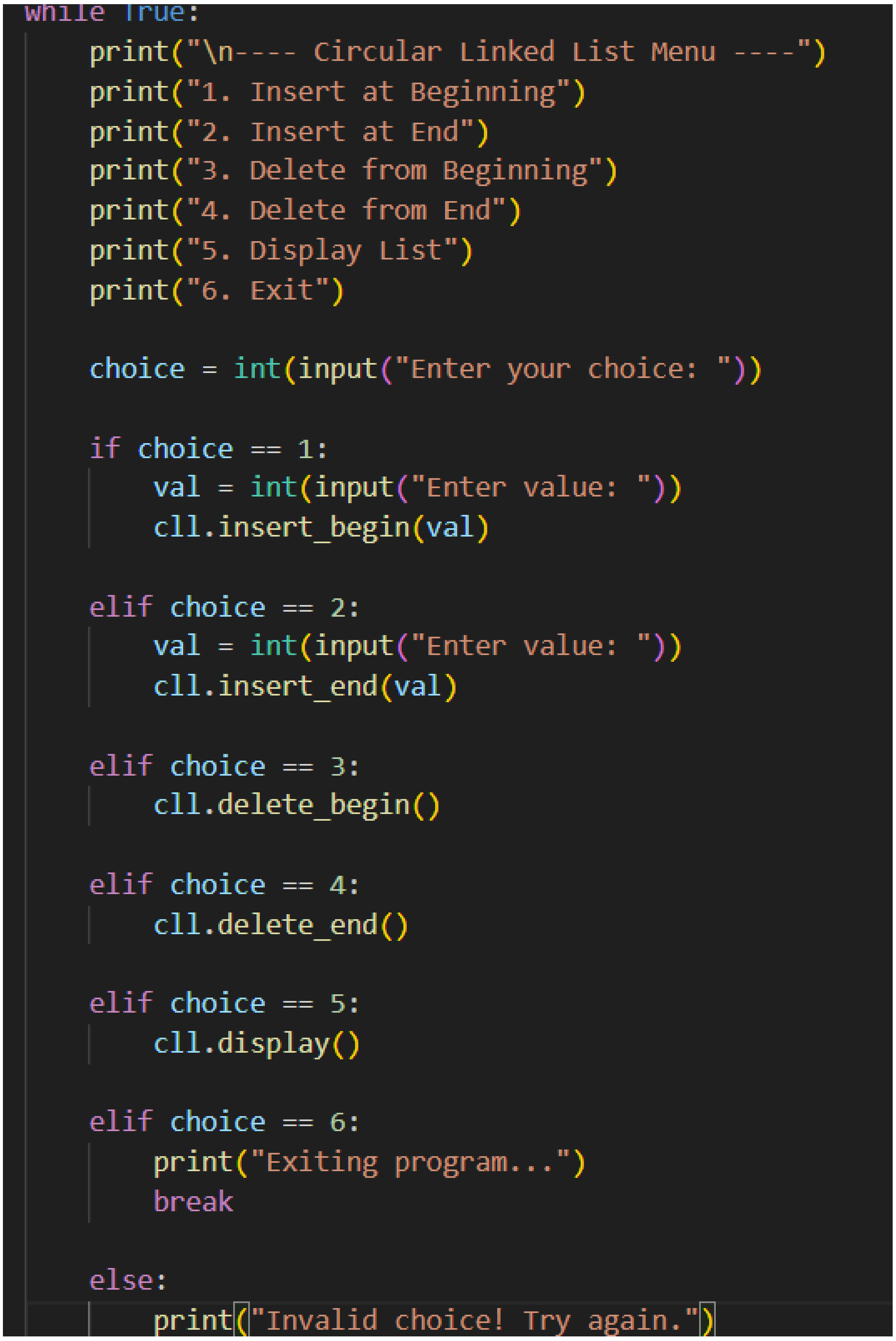
CODE



```python
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class CircularLinkedList:
    def __init__(self):
        self.head = None

    def insert_begin(self, data):
        new = Node(data)
        if self.head is None:
            self.head = new
            new.next = new
            return
        temp = self.head
        while temp.next != self.head:
            temp = temp.next
        new.next = self.head
        temp.next = new
        self.head = new

    def insert_end(self, data):
        new = Node(data)
        if self.head is None:
            self.head = new
            new.next = new
            return
        temp = self.head
        while temp.next != self.head:
            temp = temp.next
        temp.next = new
        new.next = self.head
```

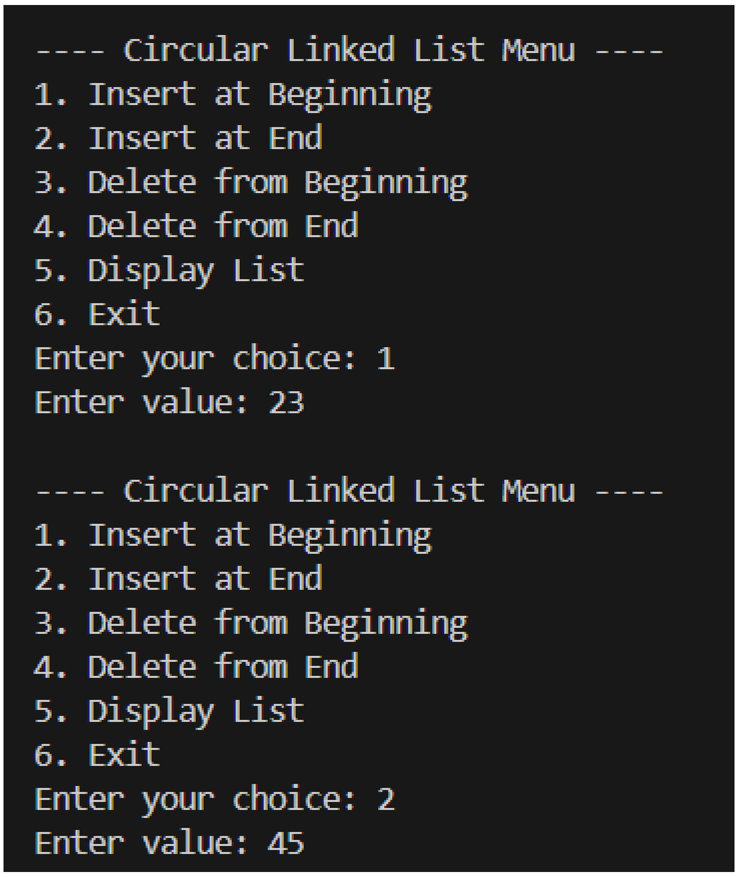


```python
    def delete_begin(self):
        if self.head is None:
            print("List is empty")
            return
        if self.head.next == self.head:
            self.head = None
            return
        temp = self.head
        while temp.next != self.head:
            temp = temp.next
        temp.next = self.head.next
        self.head = self.head.next

    def delete_end(self):
        if self.head is None:
            print("List is empty")
            return
        if self.head.next == self.head:
            self.head = None
            return
        prev = None
        temp = self.head
        while temp.next != self.head:
            prev = temp
            temp = temp.next
        prev.next = self.head

    def display(self):
        if self.head is None:
            print("List is empty")
            return
        temp = self.head
        while True:
            print(temp.data, end=" -> ")
            temp = temp.next
            if temp == self.head:
```

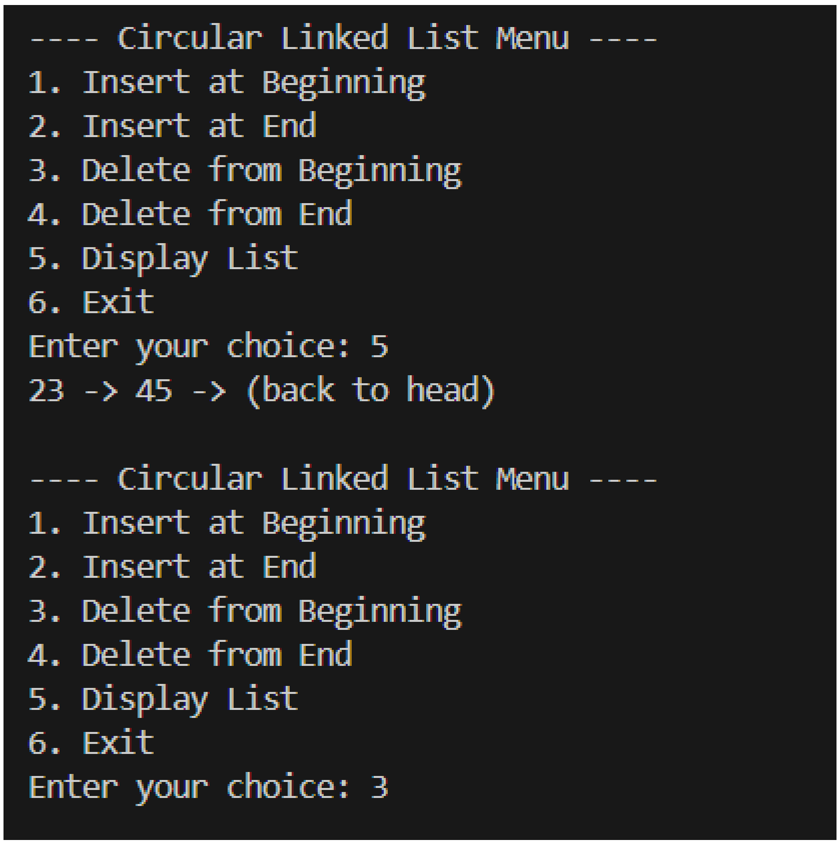


```
while True:
    print("\n---- Circular Linked List Menu ----")
    print("1. Insert at Beginning")
    print("2. Insert at End")
    print("3. Delete from Beginning")
    print("4. Delete from End")
    print("5. Display List")
    print("6. Exit")

    choice = int(input("Enter your choice: "))

    if choice == 1:
        val = int(input("Enter value: "))
        cll.insert_begin(val)

    elif choice == 2:
        val = int(input("Enter value: "))
        cll.insert_end(val)

    elif choice == 3:
        cll.delete_begin()

    elif choice == 4:
        cll.delete_end()

    elif choice == 5:
        cll.display()

    elif choice == 6:
        print("Exiting program...")
        break

    else:
        print("Invalid choice! Try again.")
```

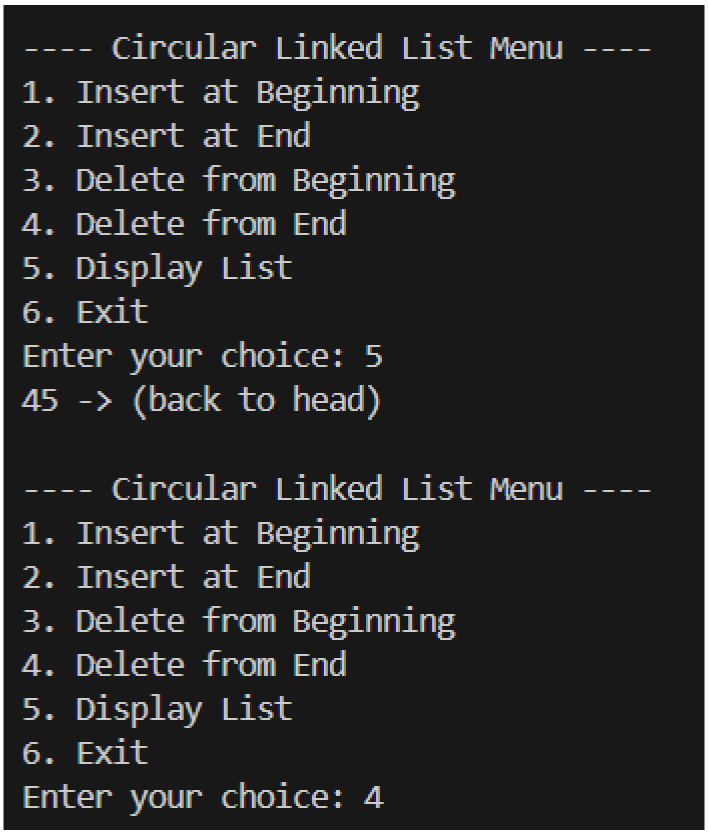
OUTPUT



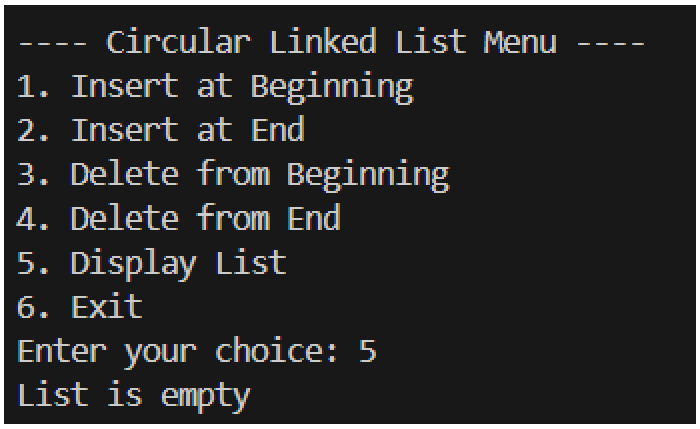
```
---- Circular Linked List Menu ----
1. Insert at Beginning
2. Insert at End
3. Delete from Beginning
4. Delete from End
5. Display List
6. Exit
Enter your choice: 1
Enter value: 23

---- Circular Linked List Menu ----
1. Insert at Beginning
2. Insert at End
3. Delete from Beginning
4. Delete from End
5. Display List
6. Exit
Enter your choice: 2
Enter value: 45
```



```
---- Circular Linked List Menu ----
1. Insert at Beginning
2. Insert at End
3. Delete from Beginning
4. Delete from End
5. Display List
6. Exit
Enter your choice: 5
23 -> 45 -> (back to head)

---- Circular Linked List Menu ----
1. Insert at Beginning
2. Insert at End
3. Delete from Beginning
4. Delete from End
5. Display List
6. Exit
Enter your choice: 3
```



```
---- Circular Linked List Menu ----
1. Insert at Beginning
2. Insert at End
3. Delete from Beginning
4. Delete from End
5. Display List
6. Exit
Enter your choice: 5
45 -> (back to head)

---- Circular Linked List Menu ----
1. Insert at Beginning
2. Insert at End
3. Delete from Beginning
4. Delete from End
5. Display List
6. Exit
Enter your choice: 4
```



```
---- Circular Linked List Menu ----
1. Insert at Beginning
2. Insert at End
3. Delete from Beginning
4. Delete from End
5. Display List
6. Exit
Enter your choice: 5
List is empty
```

Result –

The Circular Linked List is successfully created and all operations (insertion, deletion, and display) are performed correctly using user input.

Practical No –9

AIM - To implement a fixed-size linear queue using an array and perform the operations:

- Enqueue (Insertion)
- Dequeue (Deletion)
- Size (Return number of items)
- IsFull (Check if queue is full)

Description

A queue is a linear data structure that follows FIFO (First In, First Out) rule.

In an online ticketing system:

- New ticket requests are added at the rear (enqueue).
- Requests are processed from the front (dequeue).

This practical implements such a queue using:

- A fixed-size array
- Two pointers, front and rear, to track the queue
- Functions to insert, delete, count size, and check if the queue is full

Algorithm

1. Enqueue (Insert)

    1. If rear == MAX - 1 → Queue Full
    2. Else if queue empty → set front = rear = 0
    3. Else increment rear
    4. Insert element at queue[rear]

2. Dequeue (Delete)

1. If queue empty (front == -1) → print empty
2. Retrieve element from queue[front]
3. If front == rear → set both to -1 (queue becomes empty)
4. Else front++

---

3. Size

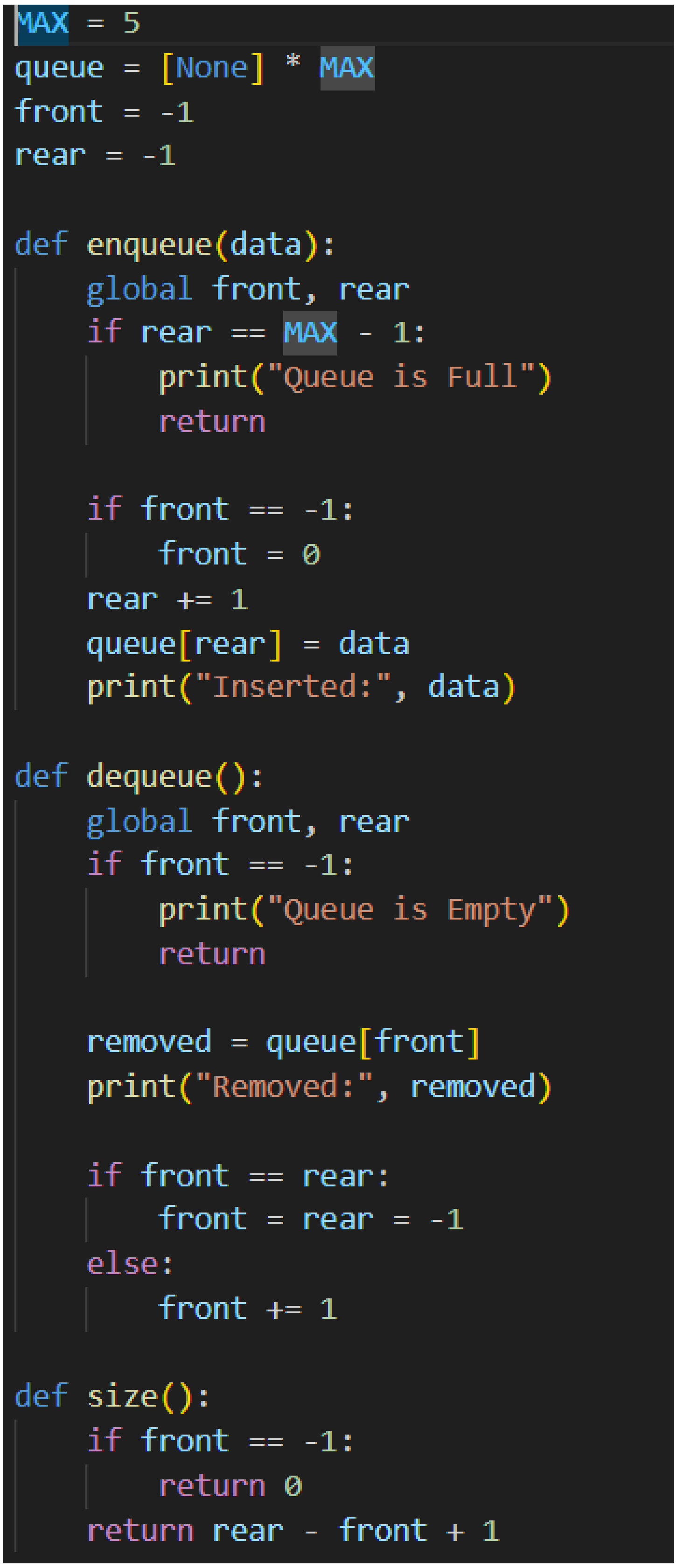
1. If queue empty → return 0
2. Else return (rear - front + 1)

---

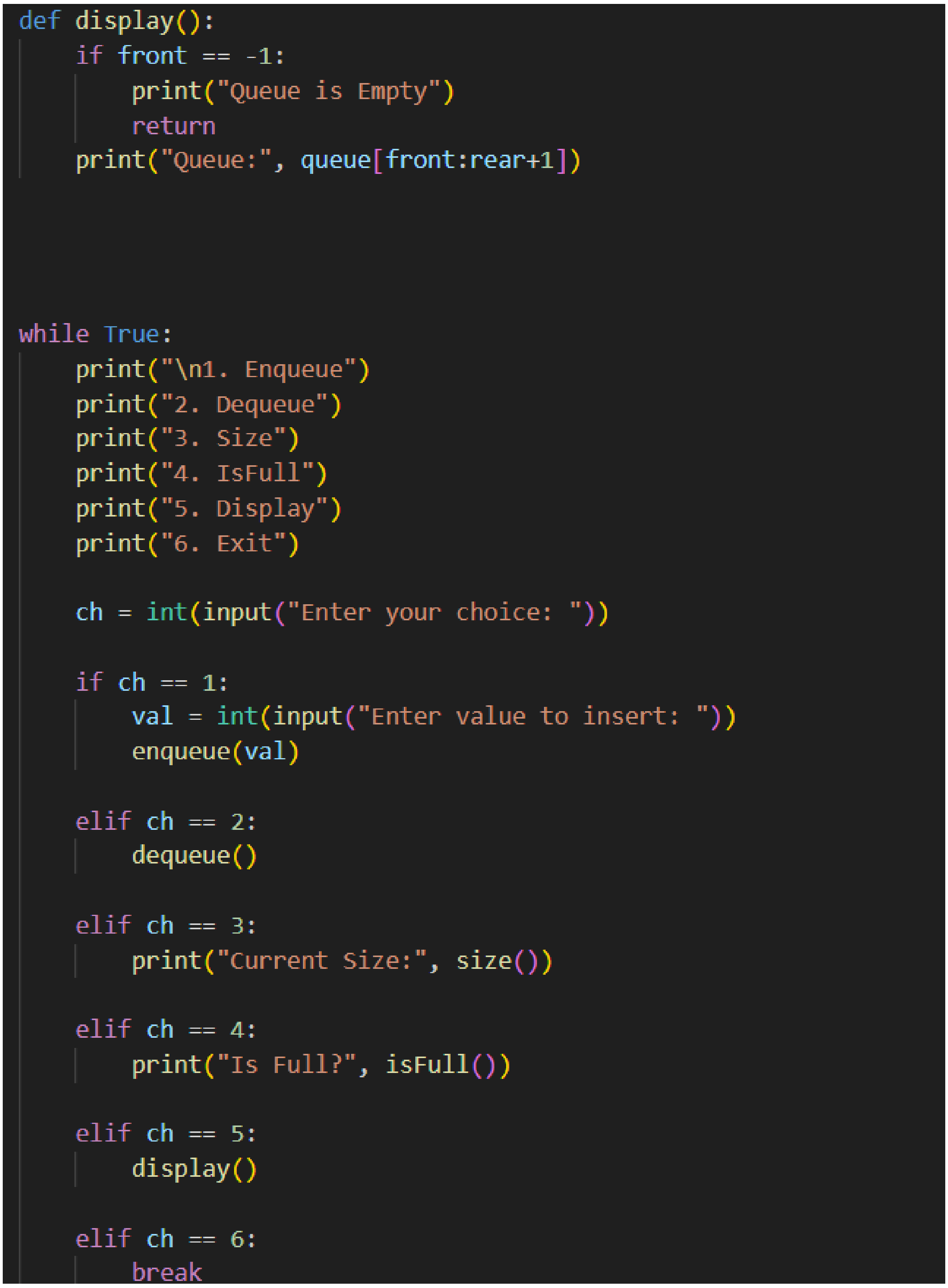
4. IsFull

1. If rear == MAX - 1 → return True
2. Else → return False

CODE

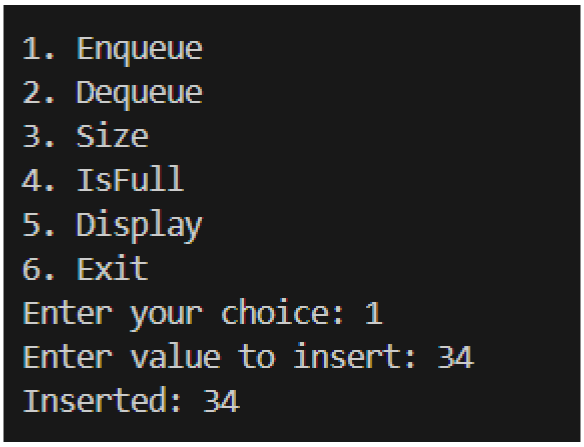


```python
MAX = 5
queue = [None] * MAX
front = -1
rear = -1

def enqueue(data):
    global front, rear
    if rear == MAX - 1:
        print("Queue is Full")
        return

    if front == -1:
        front = 0
    rear += 1
    queue[rear] = data
    print("Inserted:", data)

def dequeue():
    global front, rear
    if front == -1:
        print("Queue is Empty")
        return

    removed = queue[front]
    print("Removed:", removed)

    if front == rear:
        front = rear = -1
    else:
        front += 1

def size():
    if front == -1:
        return 0
    return rear - front + 1
```

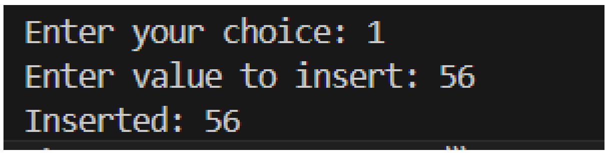


```python
def display():
    if front == -1:
        print("Queue is Empty")
        return
    print("Queue:", queue[front:rear+1])



while True:
    print("\n1. Enqueue")
    print("2. Dequeue")
    print("3. Size")
    print("4. IsFull")
    print("5. Display")
    print("6. Exit")

    ch = int(input("Enter your choice: "))

    if ch == 1:
        val = int(input("Enter value to insert: "))
        enqueue(val)

    elif ch == 2:
        dequeue()

    elif ch == 3:
        print("Current Size:", size())

    elif ch == 4:
        print("Is Full?", isFull())

    elif ch == 5:
        display()

    elif ch == 6:
        break
```

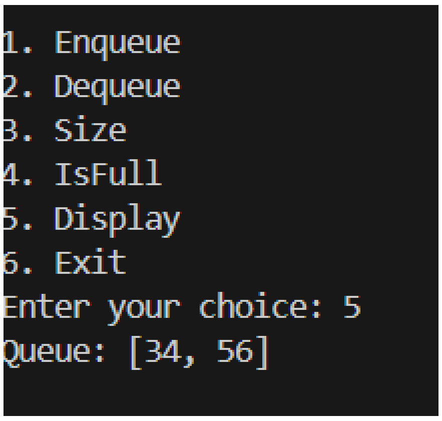
OUTPUT



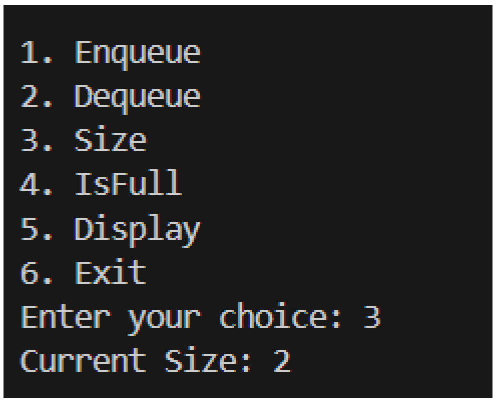
```
1. Enqueue
2. Dequeue
3. Size
4. IsFull
5. Display
6. Exit
Enter your choice: 1
Enter value to insert: 34
Inserted: 34
```



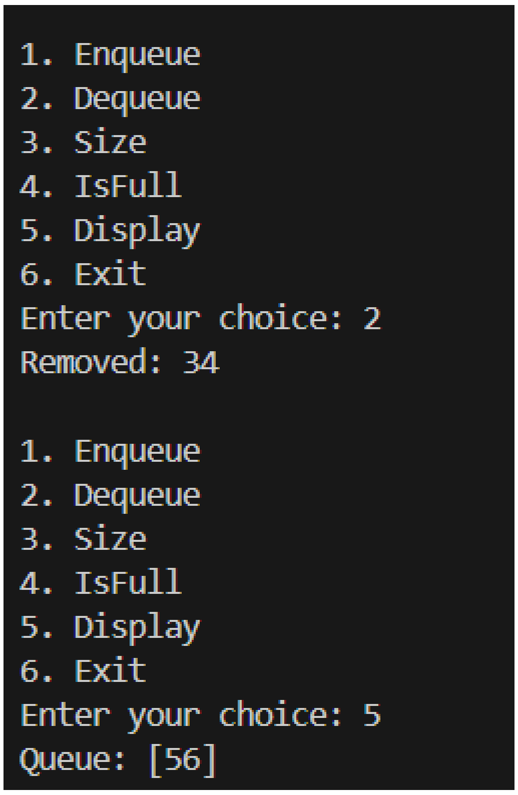
```
Enter your choice: 1
Enter value to insert: 56
Inserted: 56
```



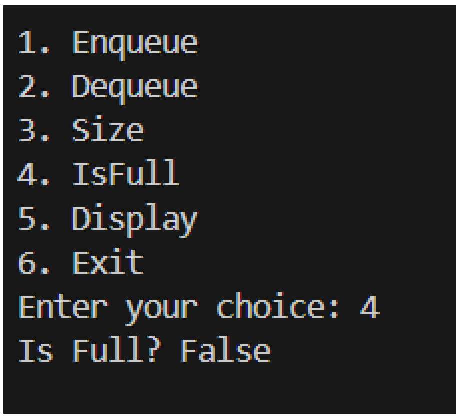
```
1. Enqueue
2. Dequeue
3. Size
4. IsFull
5. Display
6. Exit
Enter your choice: 5
Queue: [34, 56]
```



```
1. Enqueue
2. Dequeue
3. Size
4. IsFull
5. Display
6. Exit
Enter your choice: 3
Current Size: 2
```



```
1. Enqueue
2. Dequeue
3. Size
4. IsFull
5. Display
6. Exit
Enter your choice: 2
Removed: 34

1. Enqueue
2. Dequeue
3. Size
4. IsFull
5. Display
6. Exit
Enter your choice: 5
Queue: [56]
```



```
1. Enqueue
2. Dequeue
3. Size
4. IsFull
5. Display
6. Exit
Enter your choice: 4
Is Full? False
```

Result –

The fixed-size linear queue was successfully implemented using an array.
All operations—enqueue, dequeue, size, and isFull—worked correctly.

Practical No – 10

AIM - To write a Python program to implement a Circular Queue and perform the following operations:

- Insert (Enqueue)
- Delete (Dequeue)
- Display elements
- Check Queue Full / Empty condition

Description

A Circular Queue is a linear data structure in which the last position is connected back to the first position.
It solves the problem of unused space in a normal linear queue.

In a Circular Queue:

- front → points to first element
- rear → points to last element
- Both move circularly using:

  (index + 1) % size

It is useful in CPU Scheduling, Buffers, Printers, etc.

Algorithm

Algorithm for Enqueue(x)

1. Check if queue is full:
2. (rear + 1) % size == front
3. If full → print "Queue is full".
4. If empty → set front = rear = 0
5. Else → update rear = (rear + 1) % size
6. Insert element at rear.

Algorithm for Dequeue()

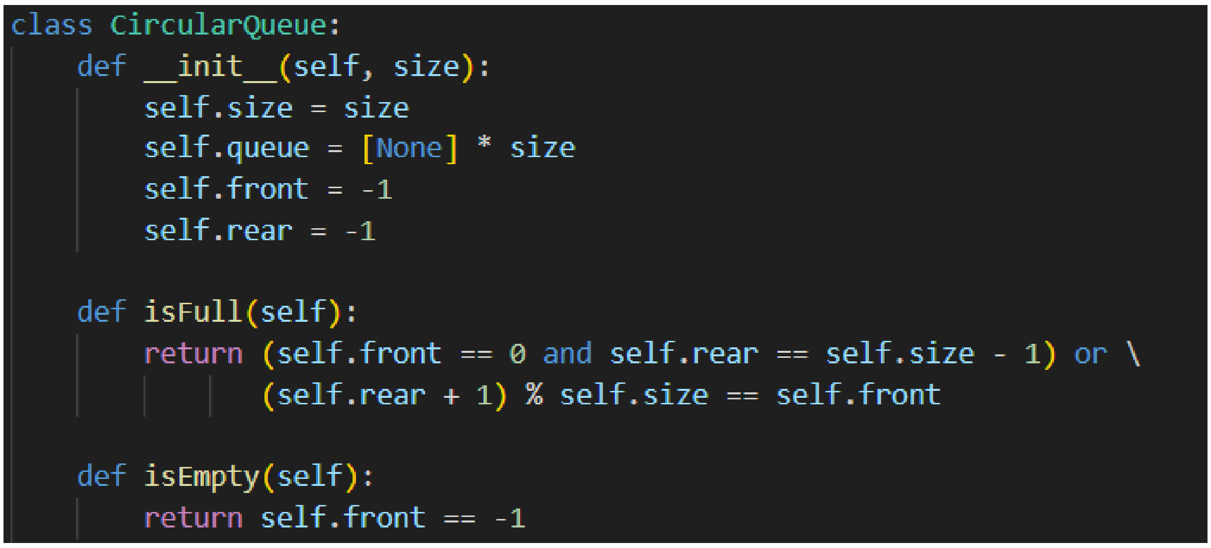
1. If queue is empty → print "Queue is empty"
2. Retrieve element at front
3. If front == rear → reset queue (front = rear = -1)
4. Else → front = (front + 1) % size
5. Return deleted element

Algorithm for Display

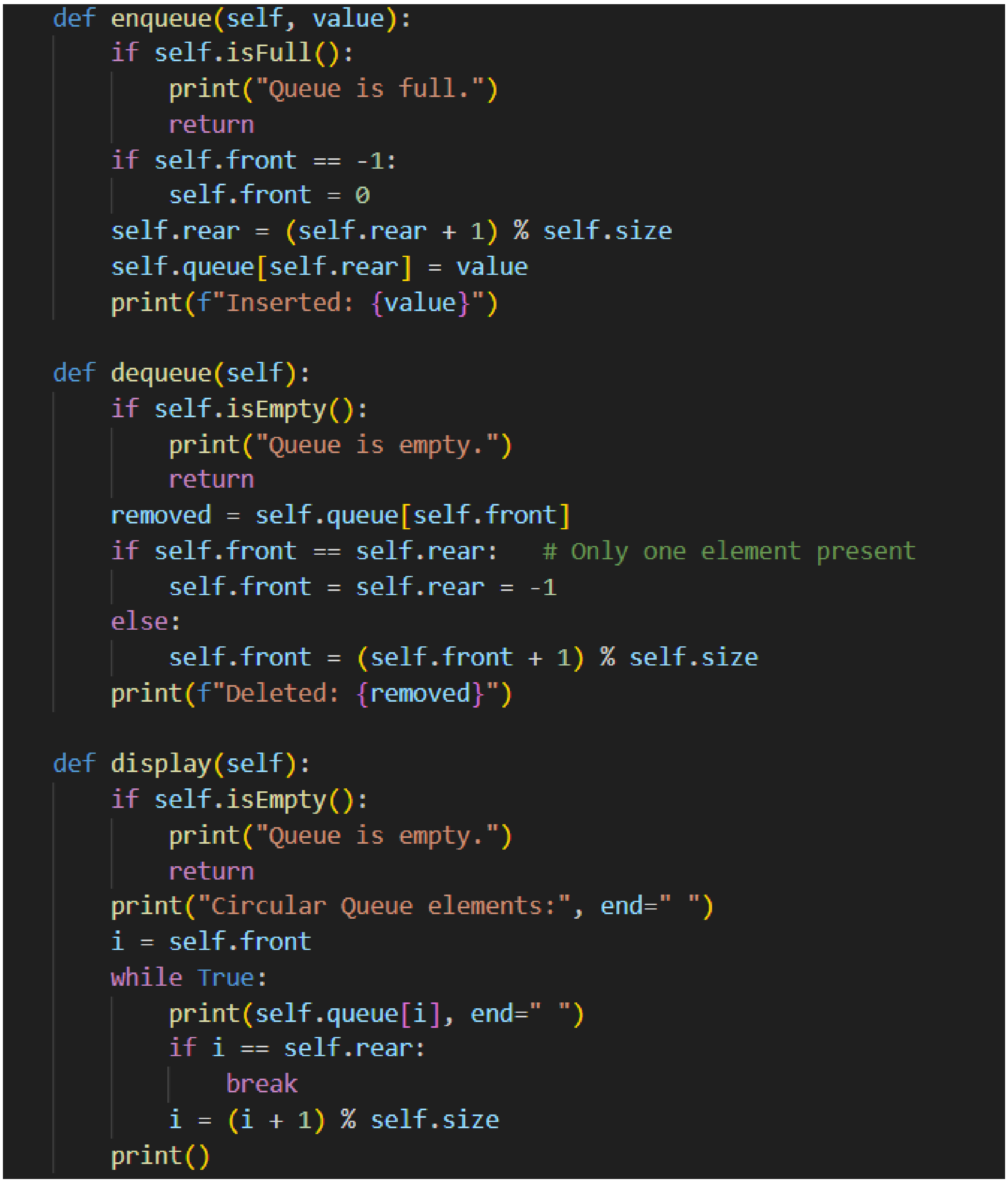
1. If empty → print "Queue is empty"
2. Start from front, loop until rear
3. Print elements circularly:

i = (i + 1) % size

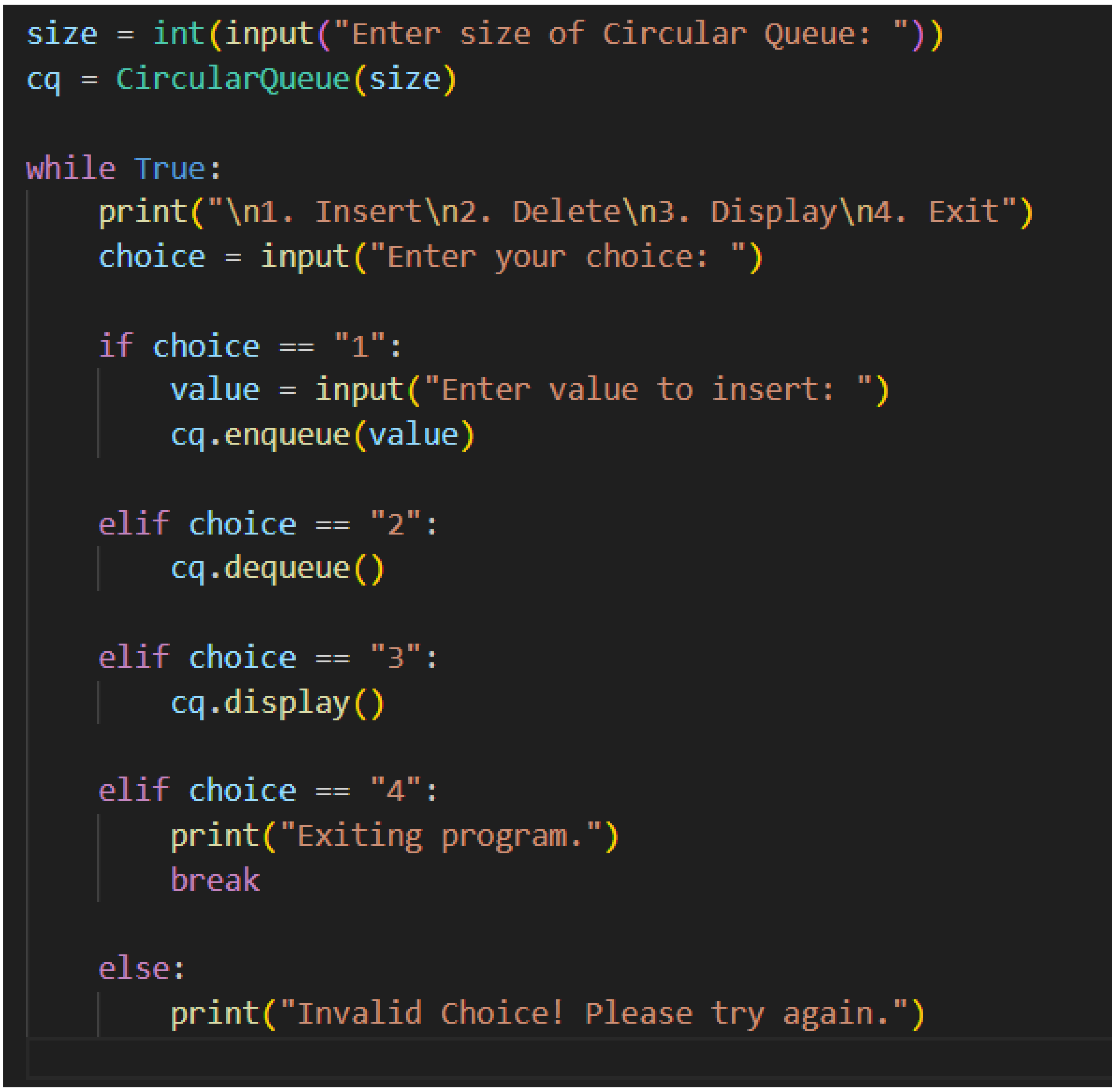
CODE



```
class CircularQueue:
    def __init__(self, size):
        self.size = size
        self.queue = [None] * size
        self.front = -1
        self.rear = -1

    def isFull(self):
        return (self.front == 0 and self.rear == self.size - 1) or \
               (self.rear + 1) % self.size == self.front

    def isEmpty(self):
        return self.front == -1
```

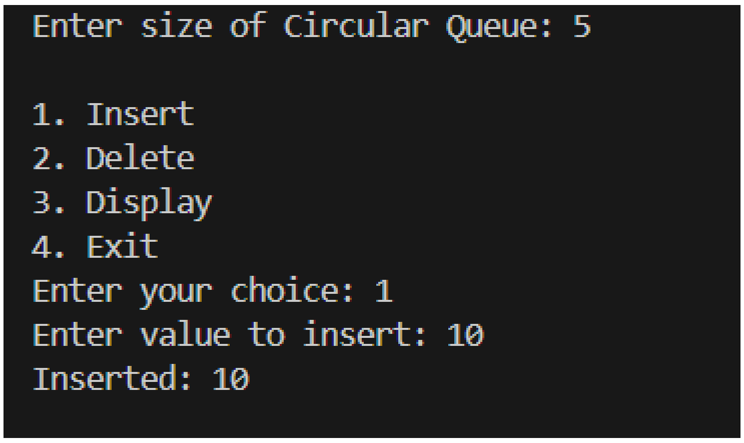


```python
    def enqueue(self, value):
        if self.isFull():
            print("Queue is full.")
            return
        if self.front == -1:
            self.front = 0
        self.rear = (self.rear + 1) % self.size
        self.queue[self.rear] = value
        print(f"Inserted: {value}")

    def dequeue(self):
        if self.isEmpty():
            print("Queue is empty.")
            return
        removed = self.queue[self.front]
        if self.front == self.rear:   # Only one element present
            self.front = self.rear = -1
        else:
            self.front = (self.front + 1) % self.size
        print(f"Deleted: {removed}")

    def display(self):
        if self.isEmpty():
            print("Queue is empty.")
            return
        print("Circular Queue elements:", end=" ")
        i = self.front
        while True:
            print(self.queue[i], end=" ")
            if i == self.rear:
                break
            i = (i + 1) % self.size
        print()
```

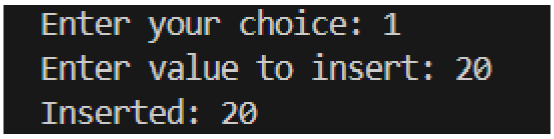


```python
size = int(input("Enter size of Circular Queue: "))
cq = CircularQueue(size)

while True:
    print("\n1. Insert\n2. Delete\n3. Display\n4. Exit")
    choice = input("Enter your choice: ")

    if choice == "1":
        value = input("Enter value to insert: ")
        cq.enqueue(value)

    elif choice == "2":
        cq.dequeue()

    elif choice == "3":
        cq.display()

    elif choice == "4":
        print("Exiting program.")
        break

    else:
        print("Invalid Choice! Please try again.")
```

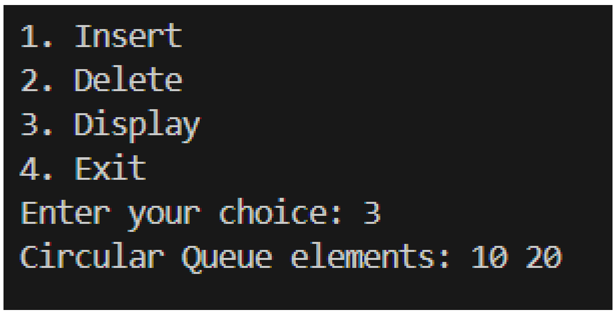
OUTPUT



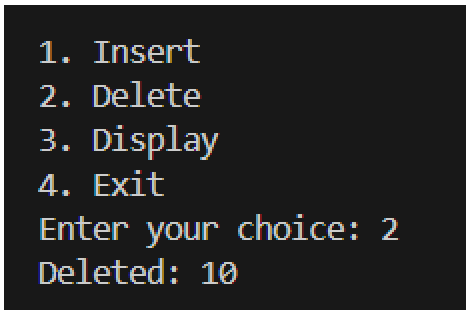
```
Enter size of Circular Queue: 5

1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1
Enter value to insert: 10
Inserted: 10
```



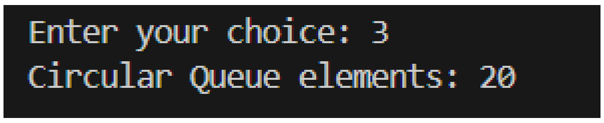
```
Enter your choice: 1
Enter value to insert: 20
Inserted: 20
```



```
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3
Circular Queue elements: 10 20
```



```
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 2
Deleted: 10
```



```
Enter your choice: 3
Circular Queue elements: 20
```

Result-

The Circular Queue was successfully implemented in Python with user input.
All operations — Insert, Delete, and Display — worked correctly, demonstrating the concept of a Circular Queue using fixed size and modular indexing.

Practical No – 11

AIM - To write a Python program to perform Binary Search on a list of numbers entered by the user.

Description

Binary Search is an efficient searching technique that works on sorted arrays.
It repeatedly divides the search interval into two halves:

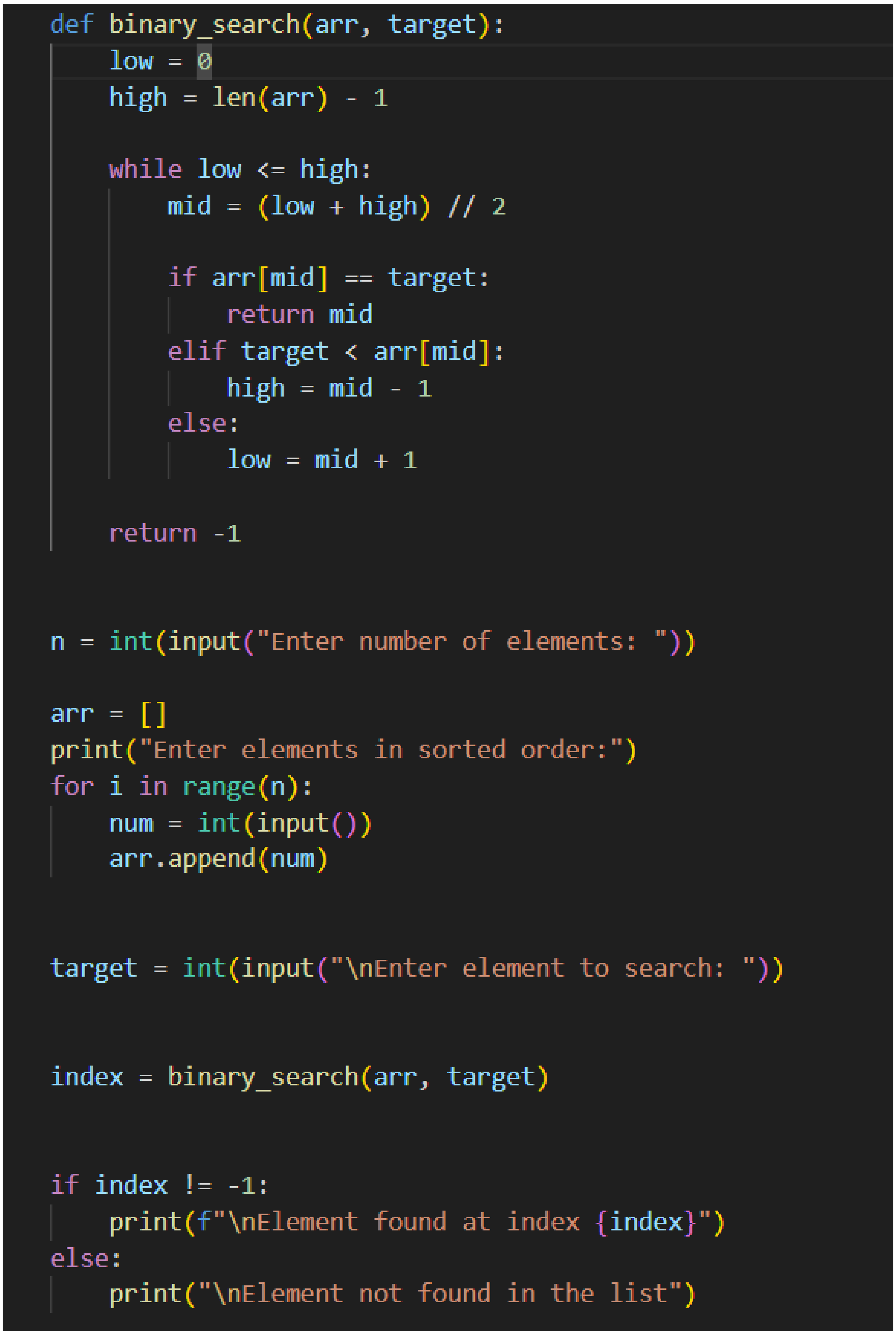
1. Compare the target element with the middle element.
2. If equal → element found.
3. If target is smaller → search left half.
4. If target is larger → search right half.

Binary Search reduces time complexity significantly compared to Linear Search.

Algorithm

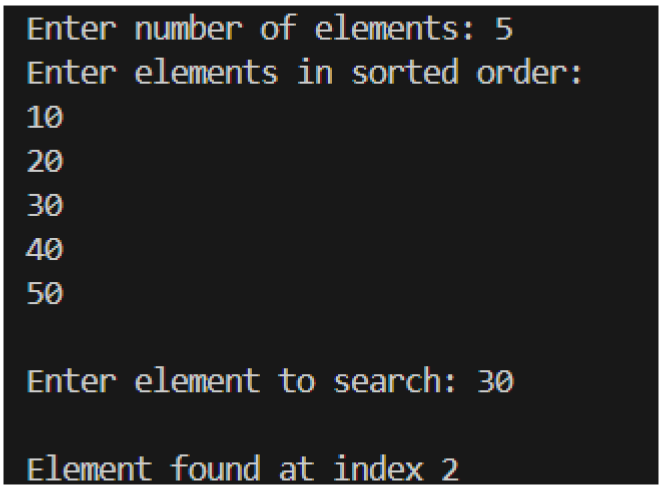
```
1. Start
2.  Input the size of array and elements (ensure the array is sorted).
3.  Input the element to search.
4. Set low = 0, high = n − 1
5.Repeat while low <= high
a. mid = (low + high) // 2
b. If arr[mid] == target, return mid
c. If target < arr[mid], set high = mid − 1
d. Else set low = mid + 1
6. If not found, return −1
7. End
```

CODE



```python
def binary_search(arr, target):
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = (low + high) // 2

        if arr[mid] == target:
            return mid
        elif target < arr[mid]:
            high = mid - 1
        else:
            low = mid + 1

    return -1


n = int(input("Enter number of elements: "))

arr = []
print("Enter elements in sorted order:")
for i in range(n):
    num = int(input())
    arr.append(num)


target = int(input("\nEnter element to search: "))


index = binary_search(arr, target)


if index != -1:
    print(f"\nElement found at index {index}")
else:
    print("\nElement not found in the list")
```

OUTPUT



```
Enter number of elements: 5
Enter elements in sorted order:
10
20
30
40
50

Enter element to search: 30

Element found at index 2
```

Result-

Binary Search was successfully implemented in Python using user input.
The program efficiently searched the sorted list and returned the index of the element.

Practical No –12

AIM- To understand the concept of Linear Search, implement it using Python, and analyze its best-case and worst-case time complexities.

Description

Linear Search is a simple searching technique used to find an element in a list or array.
It works by sequentially comparing each element of the list with the target element until a match is found or the entire list has been traversed.

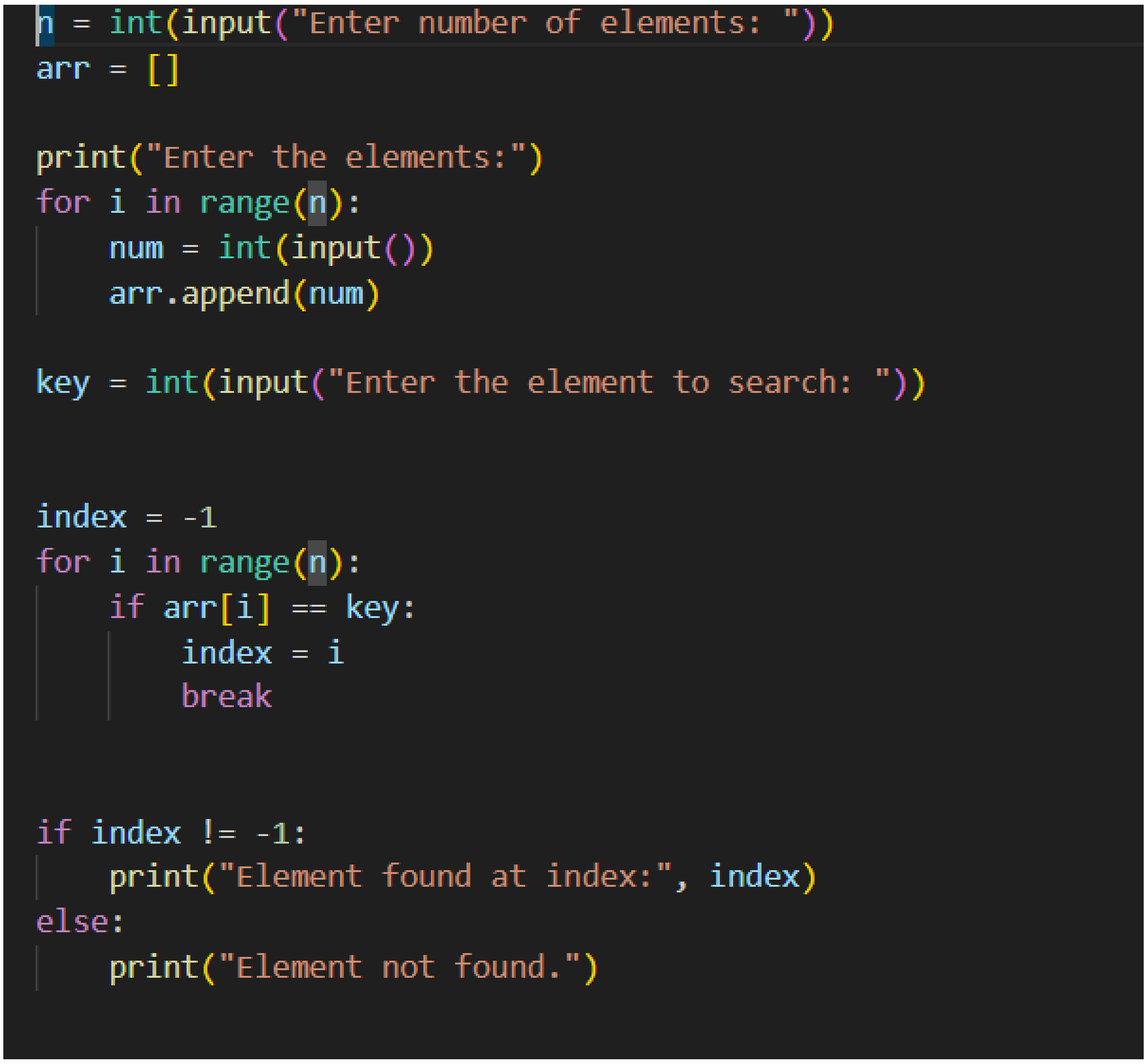
Linear Search is easy to implement and works on both sorted and unsorted lists.
However, it is inefficient for large datasets because it checks elements one by one.

Algorithm

Step 1: Start
Step 2: Input the number of elements and store them in a list
Step 3: Input the element to be searched
Step 4: Compare the target element with each element of the list sequentially
Step 5:

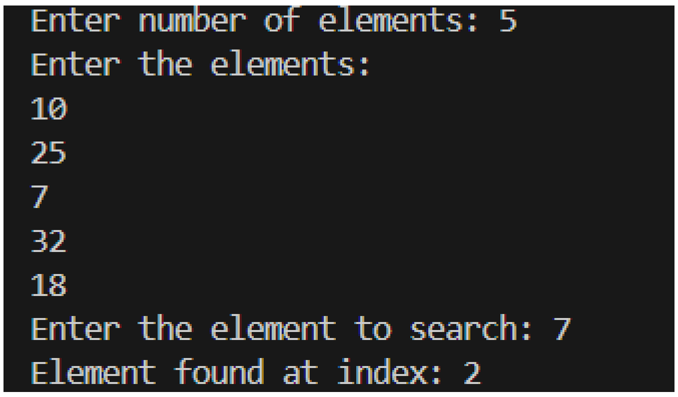
- If a match is found → return the index
- If end of list is reached without match → element not found
  Step 6: Stop

CODE



```python
n = int(input("Enter number of elements: "))
arr = []

print("Enter the elements:")
for i in range(n):
    num = int(input())
    arr.append(num)

key = int(input("Enter the element to search: "))


index = -1
for i in range(n):
    if arr[i] == key:
        index = i
        break


if index != -1:
    print("Element found at index:", index)
else:
    print("Element not found.")
```

OUTPUT



```
Enter number of elements: 5
Enter the elements:
10
25
7
32
18
Enter the element to search: 7
Element found at index: 2
```

TIME COMPLEXITY ANALYSIS

Best Case:

- Element is found at the first position
- Comparisons: 1
- Time Complexity: O(1)

Worst Case:

- Element is found at the last position or not present
- Comparisons: n
- Time Complexity: O(n)

Result –

The concept of Linear Search was successfully understood and implemented in Python.
The best-case and worst-case time complexities were analyzed, showing that Linear Search performs optimally when the element is at the beginning but becomes slower when the element is near the end or absent.

Practical No – 13

AIM - To write a Python program to sort a list of elements using Bubble Sort technique (with user input).

Description

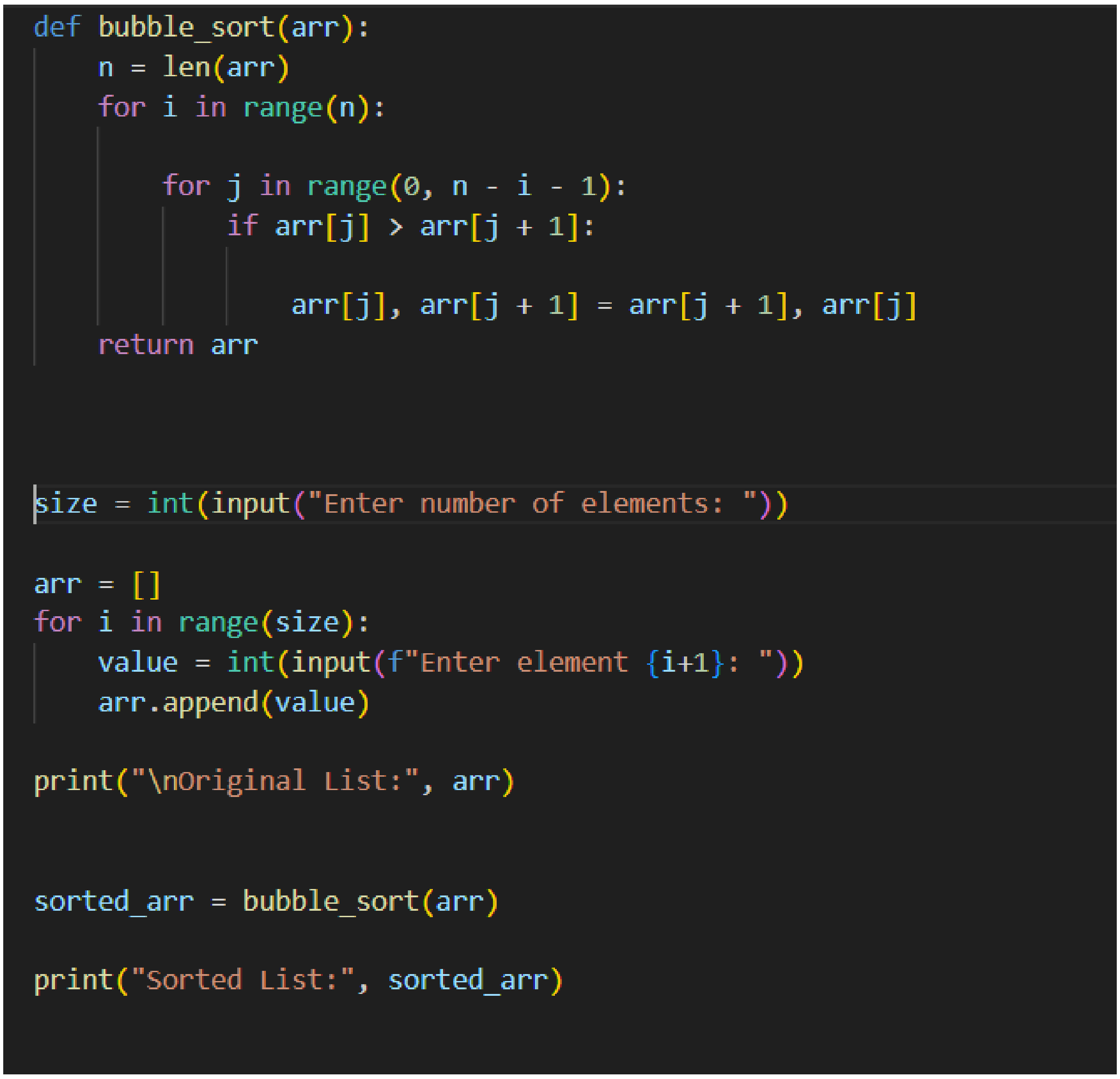
Bubble Sort is a simple comparison-based sorting algorithm.
It repeatedly compares adjacent elements and swaps them if they are in the wrong order.
Each pass “bubbles” the largest element to the end.

It is easy to implement but not efficient for large datasets.

Algorithm

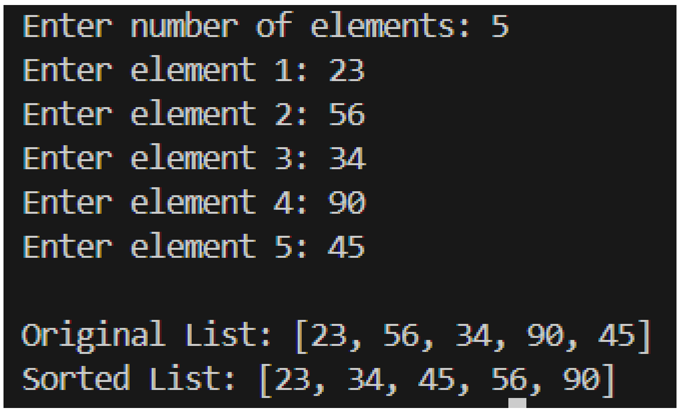
1. Start
2. Input number of elements
3. Input elements into a list
4. Repeat for i = 0 to n-1
    - Repeat for j = 0 to n-i-2
        - If arr[j] > arr[j+1], swap them
5. After all passes, array becomes sorted
6. Display the sorted list
7. Stop

CODE



```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):

        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:

                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr



size = int(input("Enter number of elements: "))

arr = []
for i in range(size):
    value = int(input(f"Enter element {i+1}: "))
    arr.append(value)

print("\nOriginal List:", arr)


sorted_arr = bubble_sort(arr)

print("Sorted List:", sorted_arr)
```

OUTPUT



```
Enter number of elements: 5
Enter element 1: 23
Enter element 2: 56
Enter element 3: 34
Enter element 4: 90
Enter element 5: 45

Original List: [23, 56, 34, 90, 45]
Sorted List: [23, 34, 45, 56, 90]
```

Result – Thus, the Bubble Sort algorithm was implemented successfully in Python using user input, and the program correctly sorted the list.

Practical No – 14

AIM - To understand the concept of Merge Sort, implement it using Python, and analyze its time complexity.

Description

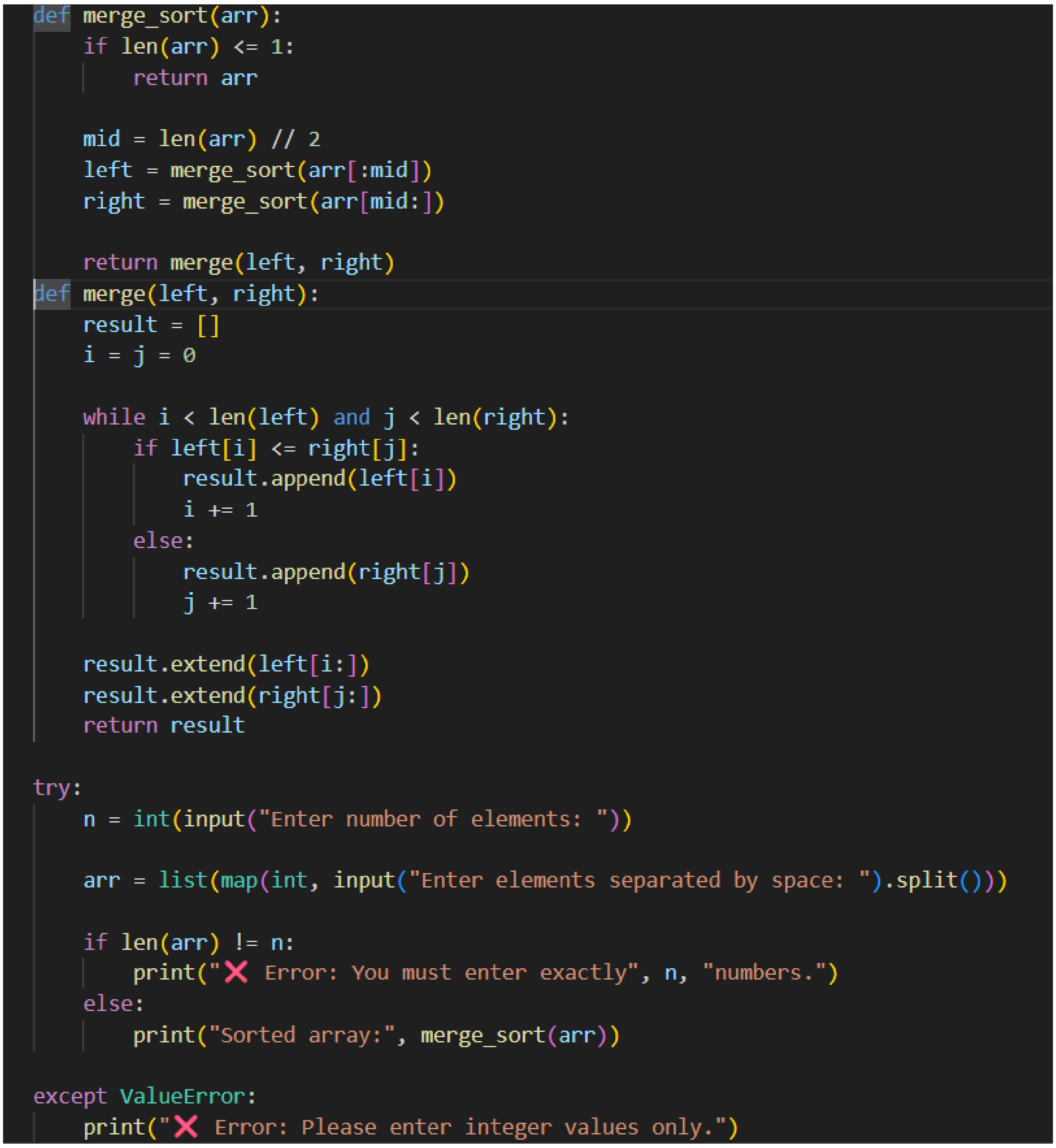
Merge Sort is a Divide and Conquer based sorting algorithm.
It divides the list into two halves, recursively sorts each half, and then merges the two sorted halves into a single sorted list.

It is an efficient, stable, and comparison-based sorting technique used for large datasets due to its consistent time complexity.

Algorithm

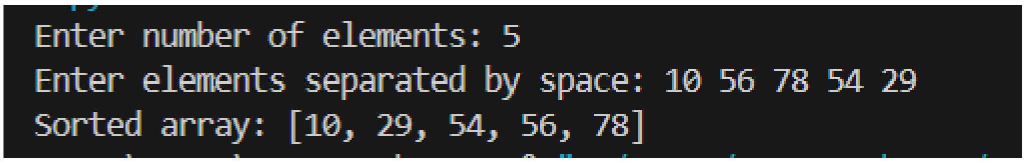
```
1. Start
2. If the list has 0 or 1 element, return the list
3. Divide the list into two halves
4. Recursively apply Merge Sort on the left half
5.  Recursively apply Merge Sort on the right half
6.Merge the two sorted halves
7.Return the merged sorted list
8. Stop
```

CODE



```python
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])

    return merge(left, right)
def merge(left, right):
    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    result.extend(left[i:])
    result.extend(right[j:])
    return result

try:
    n = int(input("Enter number of elements: "))

    arr = list(map(int, input("Enter elements separated by space: ").split()))

    if len(arr) != n:
        print("❌ Error: You must enter exactly", n, "numbers.")
    else:
        print("Sorted array:", merge_sort(arr))

except ValueError:
    print("❌ Error: Please enter integer values only.")
```

OUTPUT



```
Enter number of elements: 5
Enter elements separated by space: 10 56 78 54 29
Sorted array: [10, 29, 54, 56, 78]
```

TIME COMPLEXITY ANALYSIS

Best Case:

O(n log n)

Average Case:

O(n log n)

Worst Case:

O(n log n)

Space Complexity:

O(n) (due to merging arrays)

Result –

Merge Sort was successfully implemented in Python.
The experiment demonstrated how the algorithm divides the list, recursively sorts the halves, and merges them.
Its consistent O(n log n) complexity makes it efficient for large datasets.