K.R. Mangalam University

School of Engineering & Technology

DATA STRUCTURE

Lab File

Submitted by:

Hemant Saini

2401420051

Btech-CSE (Data Science)

Submitted to:

Dr. Swati Gupta

INDEX

LAB EXPERIMENT 01

Browser History Navigation System (Using Stack Concept)

1. Problem Statement

Create a Python-based application that imitates the navigation system of a web browser.
The system must allow the user to:

- Open new pages
- Move backward
- Move forward
- Display current navigation history
- Display complete visit log
- Generate visit frequency analysis

2. Objective

- To implement browser-style navigation using stack logic.
- To understand the working of backward and forward stacks.
- To maintain a complete log of all visited pages.
- To compute and present statistical information about visits.
- To build a menu-driven Python program for simulation.

3. Description

In real browsers, navigation is handled through two stacks:

Back Stack:

Stores all pages visited before the present page.
Whenever the user presses **Back**, the current page is pushed to the forward stack.

Forward Stack:

Holds pages that can be revisited when the user clicks **Forward**.
If a fresh page is opened, this forward stack becomes invalid and is cleared.

Complete Log (Total History):

A simple list where every visited page (including repeated visits) is recorded.

In this experiment:

- history works like the **backward movement stack**.
- forward_history represents the **forward movement stack**.
- total_history stores all page visits, even after going back or forward.

This approach clearly shows how browsers rely on stack operations to manage navigation flow.

4. Algorithm

A. Open a Page (visit_page)

1. Add the new page to the back stack
2. Clear the forward stack
3. Add the page to total history
4. Display confirmation

B. Move Backward (back_page)

1. Check if the back stack has more than one page
2. Pop the current page and push it to the forward stack
3. New current page becomes the previous entry
4. Record it in the total history

5. Display message

C. Move Forward (forward_page)

1. Check if any page exists in the forward stack
2. Pop it and push it to the back stack
3. Add it to total history
4. Show the current page

D. Display Back History

Print all pages stored in the back stack.

E. Display Total Visits

Show every single page ever visited.

F. Visit Frequency (history_stats)

1. Count number of total visits
2. Calculate how many times each page was visited
3. Print results

G. Menu Loop

Continuously display options until the user exits.

5. Program Code

```python
history=[]
forward_history=[]
total_history=[]

# Adds a new page to the browser history
def insert_page(page):
    history.append(page)

    forward_history.clear()
    total_history.append(page)

    print(f"visited page:{page}")

# Moves one step back in history (like a browser back button)
def back_page():
    if not history:
        print("No pages in history.")
        return

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

    current = history[-1]
    total_history.append(current)

    print(f"Going back from {last_page}")

    if history:
        print(f"Current page: {history[-1]}")

    else:
        print("No pages left in history.")

# Moves forward again if the user previously went back
def forward_page():
    if not forward_history:
        print("No forward pages available.")
        return
```

```python

    next_page = forward_history.pop()
    history.append(next_page)
    total_history.append(next_page)
    print(f"Going again to {next_page}")

# Shows only the live browsing history (current path)
def view_history():
    if not history:
        print("NO pages in history!")

    else:
        print("History:","->".join(history))

# Shows the complete history including revisits
def view_total_history():
    if not history:
        print("NO pages in history!")

    else:
        print("History:","->".join(total_history))

# Displays statistics like total visits and frequency of each page
def history_stats():
    if not total_history:
        print("No total history to analyze!")
        return

    print(f"\nTotal visits: {len(total_history)}")
    freq = {}
    for page in total_history:
        if page in freq:
            freq[page] += 1
        else:
            freq[page] = 1

```

```
    for page, count in freq.items():
        print(f"Page '{page}' visited {count} time(s)")


while True:

    print("\n1) Visit page")
    print("2) Go back")
    print("3) Go Forward")
    print("4) View History")
    print("5) view Total History")
    print("6) View history stats")
    print("7) Exit")
    choice =int(input("Enter your choice:"))

    if choice == 1:
        page=str(input("Insert page name:"))
        insert_page(page)

    elif choice == 2:
        back_page()

    elif choice ==3:
        forward_page()

    elif choice == 4:
        view_history()

    elif choice ==5:
        view_total_history()

    elif choice == 6:
        history_stats()

    elif choice == 7:
        print("Exiting Browser....Goodbye!")
        break

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

6. Sample Output

```
1) Visit page
2) Go back
3) Go Forward
4) View History
5) view Total History
6) View history stats
7) Exit
Enter your choice:
```

```
Enter your choice:1
Insert page name:google.com
visited page:google.com
```

```
Enter your choice:1
Insert page name:instagram.com
visited page:instagram.com
```

```
Enter your choice:2
Going back from instagram.com
Current page: google.com
```

```
Enter your choice:3
Going again to instagram.com
```

```
Enter your choice:4
History: google.com->instagram.com
```

```
Enter your choice:5
History: google.com->instagram.com->google.com->instagram.com
```

```
Enter your choice:6

Total visits: 4
Page 'google.com' visited 2 time(s)
Page 'instagram.com' visited 2 time(s)
```

```
Enter your choice:7
Exiting Browser....Goodbye!
```

7.Time Complexities

Operation	Complexity
Visit Page	O(1)
Back	O(1)
Forward	O(1)
View History	O(n)
History Stats	O(n)

8. Conclusion

This experiment successfully demonstrates how a browser manages navigation using stack structures.
The program replicates real browser behavior, where back and forward actions rely on two stacks, while a separate list maintains the complete visit record.
The simulation highlights efficient use of list operations, stack behavior, and simple statistical analysis.

LAB EXPERIMENT 02

Ticketing System Using Queue (Linear Queue Implementation)

- **1. Problem Statement**
- Develop a Python program that models a basic ticketing system using the concept of a linear queue.
  The application should support:
- Adding new tickets
- Removing tickets
- Showing current queue elements
- Checking queue size
- Detecting overflow and underflow situations

- **2. Objective**
- To practice queue operations such as enqueue and dequeue.
- To implement a fixed-size linear queue using a Python list.
- To understand and handle boundary conditions like queue full/empty.
- To use a class to organize queue functionality.
- To provide real-time display of queue content and size.

- **3. Description**
- A queue is an ordered data structure that works on the **FIFO (First-In-First-Out)** rule.
  The item that enters first is also the first to be removed, making it similar to people waiting in line.

- In this experiment:
- The queue is implemented using a list with a fixed capacity.
- front denotes the position of the current first ticket.
- rear denotes where the next ticket will be added.
- enqueue() adds tickets at the rear end.
- dequeue() removes tickets from the front end.
- **Special Conditions:**
- **Overflow:** Triggered when rear == MAX - 1 (queue is completely filled).
- **Underflow:** Occurs when the queue has no valid data (front == -1 or front > rear).
- This system represents real-life queues such as ticket counters, bank queues, or helpdesk lines.

- **4. Algorithm**
- **A. Enqueue (Add Ticket)**
- Verify whether the queue is full.
- If this is the first entry, set front = 0.
- Increment rear and store the ticket value.
- Show a message that the ticket is added.
- **B. Dequeue (Remove Ticket)**
- Check whether the queue is empty.
- Read and remove the item at front.
- Move front one step forward.
- Print the removed ticket.
- **C. isFull**

- Returns True when:
  rear == MAX - 1
- **D. isEmpty**
- Returns True when:
  front == -1 **or** front > rear
- **E. size**
- Number of active elements is:
  rear - front + 1
- **F. display**
- Show all queue elements from front to rear.

5. Program Code

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

    # To check Overflow
    def isFull(self):
        return self.rear == self.MAX - 1

    # To check Underflow
    def isEmpty(self):
        return self.front == -1 or self.front > self.rear

    # To Insert a ticket
    def enqueue(self, ticket_id):
        if (self.isFull()):
            print("Sorry the queue is full, can't add ticket: ", ticket_id)
            return

        if (self.front== -1):
            self.front = 0

        self.rear += 1
        self.queue[self.rear] = ticket_id
        print("Ticket Added: ", ticket_id)

    # TO delete a ticket
    def dequeue(self):
        if self.isEmpty():
            print("Sorry the Queue is empty, can't remove any ticket")
            return

        removed_ticket = self.queue[self.front]
        print("Removed ticket: ", removed_ticket)
        self.front += 1

```

```
    # To check size of queue
    def size(self):
        if self.isEmpty():
            return 0

        return (self.rear - self.front + 1)

    # To display the whole Queue
    def display(self):
        if self.isEmpty():
            print("Sorry, The Queue is Empty!")
        else:
            print("Current Queue:", self.queue[self.front:self.rear+1])


# Example to run
queue = TicketingSystem(5)

queue.enqueue("T-101")
queue.enqueue("T-102")
queue.enqueue("T-103")

queue.display()

queue.dequeue()

queue.display()

print("Size of queue:", queue.size())
print("Is Queue Full?", queue.isFull())
print("Is Queue Empty?", queue.isEmpty())
```

6. Sample Output

```
Ticket Added:  T-101
Ticket Added:  T-102
Ticket Added:  T-103
Current Queue: ['T-101', 'T-102', 'T-103']
Removed ticket:  T-101
Current Queue: ['T-102', 'T-103']
Size of queue: 2
Is Queue Full? False
Is Queue Empty? False
```

7. Time Complexities

- Enqueue → O(1)
- Dequeue → O(1)
- Size → O(1)
- Display → O(n)

Real-World Applications

- Token systems in banks

- Ticket counters
- Customer support queues
- Print job management
- Call-center request processing

8. Observations / Conclusion

This experiment clearly shows how a linear queue operates using a fixed-size array.
All major queue operations—enqueue, dequeue, viewing the queue, and checking size—are successfully implemented.
The program also properly identifies overflow and underflow conditions, strengthening the understanding of queue-based systems.
Such queue models are widely used in real-world service systems involving ordered processing.

LAB EXPERIMENT 03

Singly Linked List Operations (Insertion, Deletion, Search, Display)

1. Problem Statement

Create a Python program that implements a singly linked list and supports essential operations such as:

- Inserting nodes at the beginning, end, and at any specific position
- Deleting nodes from the beginning and end
- Searching for a value
- Displaying all nodes in the list

2. Objective

- To understand how linked lists work internally using node-to-node connections.
- To perform insertion at different positions in the list.
- To carry out deletion operations efficiently.
- To apply sequential search on linked nodes.
- To observe how dynamic memory linking is handled using references.
- To use Python classes to design and implement the linked list structure.

3. Description

A singly linked list is a dynamic structure composed of **nodes**, where each node contains:

- **Data** (value stored in the node)
- **Next pointer**, referencing the following node

The first node is known as the **head** or **start** of the list.

Unlike arrays, linked lists do not occupy continuous memory locations. Every node is connected using pointers, making insertions and deletions easier since elements do not need to be shifted.

In this program we implement:

- Node representation

| data | next → |

- Operations supported:
    - Add node at the beginning
    - Add node at the end
    - Insert at a particular position
    - Search a value
    - Delete from beginning
    - Delete from end
    - Display the entire list

This experiment helps understand how self-referencing objects form a chain-like structure similar to pointers.

4. Algorithm

A. Insert at Beginning

1. Create a new node.
2. Set the new node's next pointer to the current head.
3. Update the head to this new node.

B. Insert at End

1. Generate a new node.
2. If the list is empty → make it the head.
3. Otherwise traverse to the last node and link it to the new node.

C. Insert at a Specific Position

1. If the position is 1 → perform beginning insertion.
2. Move through the list until reaching position − 1.
3. Link the new node between previous and next nodes.
4. If position exceeds the list length → add at the end.

D. Search for an Element

1. Start from head.
2. Compare each node's data with the desired value.
3. Indicate whether the element was found.

E. Delete from Beginning

1. If list is empty → display an error.
2. Set head to the next node.

F. Delete from End

1. If list is empty → error message.
2. If only one node exists → set head = None.
3. Otherwise locate the second-last node and detach the last node.

G. Display the Linked List

1. Start from head.
2. Print each node until you reach a None reference.

5. Program Code

```python
class Node:
    def __init__(self, Element):

        self.Element = Element
        self.Link = None

class Linked_List:
    def __init__(self):

        self.Start = None



    def insert_at_beginning(self, Element):

        new_node = Node(Element)

        new_node.Link = self.Start
        self.Start = new_node

```

```python

    def insert_at_end(self, Element):

        new_node = Node(Element)

        if self.Start is None:

            self.Start = new_node
            return

        temp = self.Start

        while temp.Link:
            temp = temp.Link

        temp.Link = new_node

    def insert_at_middle(self, Element, position):
        new_node = Node(Element)

        if position ==1:
            new_node.Link = self.Start
            self.Start = new_node
            return

        temp = self.Start
        current_pos = 1

        while temp is not None and current_pos < position - 1:
            temp = temp.Link
            current_pos += 1

        if temp is None:
            print("Position outside list range — inserting at end.")
            self.insert_at_end(Element)
            return

```

```
        new_node.Link = temp.Link
        temp.Link = new_node


    def search(self, key):
        temp = self.Start

        while temp.Link:
            if temp.Element == key:
                print("Found")
                return True

            temp = temp.Link
        print("Not Found")
        return False



    def delete_from_beggining(self):
        if self.Start is None:
            print("List is Empty!")
            return

        self.Start = self.Start.Link



    def delete_from_end(self):
        if self.Start is None:
            print("List is Empty!")
            return

        if self.Start.Link is None:
            self.Start = None
            return

```

```
        temp = self.Start
        while(temp.Link.Link):
            temp = temp.Link

        temp.Link = None



    def display(self):
        temp = self.Start

        while temp:
            print(temp.Element, end=" -> ")
            temp = temp.Link

        print("None")


LL = Linked_List()

LL.insert_at_beginning(10)
LL.insert_at_beginning(20)
LL.insert_at_end(30)
LL.insert_at_end(40)
LL.display()

LL.insert_at_middle(50,2)
LL.display()

LL.search(20)
LL.search(2)

LL.delete_from_beggining()
LL.display()
LL.delete_from_end()
LL.display()
```

6. Sample Output

7. Time Complexities

Operation	Complexity
Insert (begin/end)	O(1) / O(n)
Insert at position	O(n)
Delete (begin/end)	O(1) / O(n)
Search	O(n)
Display	O(n)

8. Conclusion

The singly linked list program efficiently performs all required operations such as inserting nodes in multiple positions, deleting nodes, searching, and traversing through the list.
This task highlights how dynamic node linking works and how pointer manipulation shapes the entire structure.
It also shows how linked lists differ from arrays by offering flexible insertions and deletions without shifting elements.

LAB EXPERIMENT 04

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

1. Problem Statement

Develop a Python program that implements a Circular Singly Linked List (CSLL). The program should support the following functions:

- Insert a node at the beginning
- Insert a node at the end
- Insert a node at a specific index
- Search for a given element
- Delete a node from the beginning
- Delete a node from the end
- Display all nodes in circular order

2. Objective

The aim of this experiment is to:

- Learn how circular linked lists maintain continuous linking.
- Practice managing node references when the last node connects back to the first.
- Perform all major list operations: insertion, deletion, search, and traversal.
- Understand how circular traversal differs from traditional linked lists.
- Strengthen dynamic data structure concepts in Python using classes.

3. Theory / Description

A **Circular Singly Linked List** is an extension of a normal singly linked list.
The key difference is:

➡️ **The last node does not point to None; instead, it points back to the first node**, forming a continuous ring.

Node Format

Each node contains:

Data → Reference (next node)

Important Characteristics

- No terminal node contains None as its next link.
- Traversal continues until the pointer cycles back to the first node.
- Very useful in:
    - Round-robin scheduling
    - Games with cyclic turns
    - Repetitive looping tasks
    - Buffers and real-time systems

Operations Implemented

- Insert at beginning
- Insert at end
- Insert at a given position
- Search for an element
- Delete from beginning
- Delete from end
- Display complete circular list with proper looping

This experiment covers the essential working of circular structures and pointer movement.

4. Algorithm

A. Insert at Beginning

1. Create the new node.
2. If the list is empty:
    - Point the node to itself
    - Set it as Start.
3. Otherwise:
    - Traverse to the last node.
    - Link the new node to the old Start.
    - Link last node to new node.
    - Update Start.

B. Insert at End

1. Create new node.
2. If empty: link node to itself and mark as Start.
3. Else traverse to last node.
4. Link last node → new node → Start.

C. Insert at Any Position

1. If list is empty → first node becomes Start.
2. If position = 1 → insertion at beginning.
3. Traverse until the node before the target position.
4. If end reached before position → insert at last.
5. Otherwise insert between two existing nodes.

D. Search for an Element

1. Begin at Start.
2. Move through each node until Start is reached again.
3. Check each node's data with the target value.
4. Report whether element is found.

E. Delete from Beginning

1. If empty → deletion not possible.
2. If only one node → remove it, set Start to None.
3. Traverse to last node.
4. Redirect last node to second node.
5. Update Start.

F. Delete from End

1. If empty → no node to delete.
2. If only one node → clear list.
3. Traverse to the node just before the last.
4. Link it directly back to Start.

G. Display List

1. Start from the first node.
2. Keep printing nodes until Start is encountered again.
3. Indicate circular return (e.g., "→ back to Start").

5. Program Code

```
class Node:
    def __init__(self, Element):

        self.Element = Element
        self.Link = None

class Circular_Linked_List:
    def __init__(self):

        self.Start = None


    def insert_at_beginning(self, Element):

        new_node = Node(Element)

        if self.Start is None:

            self.Start = new_node
            new_node.Link = self.Start

            return

        temp = self.Start

        while temp.Link != self.Start:
            temp = temp.Link

        new_node.Link = self.Start
        temp.Link = new_node
        self.Start = new_node


    def insert_at_end(self, Element):

        new_node = Node(Element)

        if self.Start is None:
            self.Start= new_node
            new_node.Link = self.Start

            return

        temp = self.Start

        while temp.Link != self.Start:
            temp = temp.Link

        temp.Link = new_node
        new_node.Link = self.Start
```

```
52
53
54      def insert_at_middle(self, Element, position):
55          new_node = Node(Element)
56
57          if self.Start is None:
58
59              print("List is empty, it will be inserted as first node! ")
60
61              self.Start = new_node
62              new_node.Link = self.Start
63
64              return
65
66          if position == 1:
67              self.insert_at_beginning(Element)
68
69              return
70
71          temp = self.Start
72          current_pos = 1
73
74          while current_pos < position - 1 and temp.Link != self.Start:
75
76              temp = temp.Link
77              current_pos += 1
78
79          if temp.Link == self.Start and current_pos < position - 1:
80
81              print("Position outside range — inserting at end.")
82              self.insert_at_end(Element)
83
84              return
85
86          new_node.Link = temp.Link
87          temp.Link = new_node
88
89
```

```
90      def search(self, key):
91          temp = self.Start
92
93          while temp.Link:
94              if temp.Element == key:
95                  print("Found")
96                  return True
97
98              temp = temp.Link
99          print("Not Found")
100         return False
101
102
103
104     def delete_from_beggining(self):
105         if self.Start is None:
106             print("List is empty, nothing to delete.")
107             return
108
109         temp = self.Start
110
111         # If there is only one node
112         if self.Start.next == self.Start:
113             self.Start = None
114             print("Deleted the only node in the list.")
115             return
116
117         # Move to the last node
118         last = self.Start
119         while last.next != self.Start:
120             last = last.next
121
122         # Point last node to next of Start
123         last.next = self.Start.next
124         self.Start = self.Start.next
125         print("Node deleted from beginning.")
126
```

```python

    def delete_from_end(self):
        if self.Start is None:
            print("List is Empty!")
            return

        if self.Start.Link is None:
            self.Start = None
            return

        temp = self.Start
        while(temp.Link.Link):
            temp = temp.Link

        temp.Link = None



    def display(self):

        if self.Start is None:

            print("List is empty")
            return

        temp = self.Start

        while True:

            print(temp.Element, end=" -> ")
            temp = temp.Link

            if temp == self.Start:
                break

        print("(back to Start)")

```

```python

# Example usage
cll = Circular_Linked_List()

cll.insert_at_end(10)
cll.insert_at_end(20)
cll.insert_at_end(30)

print("After inserting 10, 20, 30 at end:")
cll.display()

cll.insert_at_beginning(5)
print("\nAfter inserting 5 at beginning:")
cll.display()

cll.insert_at_end(40)
print("\nAfter inserting 40 at end:")
cll.display()

cll.insert_at_middle(15, 4)
print("\nAfter inserting 15 at 4th position:")
cll.display()

print("\nSearching 20:")
cll.search(20)

print("\nSearching 99:")
cll.search(99)

print("\nDeleting from beginning:")
cll.delete_from_beggining()
cll.display()

print("\nDeleting from end:")
cll.delete_from_end()
cll.display()
```

6. Sample Output

```
After inserting 5 at beginning:
5 -> 10 -> 20 -> 30 -> (back to Start)

After inserting 40 at end:
5 -> 10 -> 20 -> 30 -> 40 -> (back to Start)

After inserting 15 at 4th position:
5 -> 10 -> 20 -> 15 -> 30 -> 40 -> (back to Start)

Searching 20:
Found

Searching 99:
Not Found

Deleting from beginning:
Node deleted from beginning.
10 -> 20 -> 15 -> 30 -> 40 -> (back to Start)

Deleting from end:
Node deleted from end.
10 -> 20 -> 15 -> 30 -> (back to Start)
```

7. Time Complexities

Operation	Complexity
Insert (begin/end)	O(n)
Insert at position	O(n)
Delete (begin/end)	O(n)
Search	O(n)
Display	O(n)

8. Conclusion

The Circular Singly Linked List implementation correctly preserves the loop structure during all operations. Insertions, deletions, and searches work smoothly at any position. Traversal stops only when the pointer returns to the Start node, confirming proper circular linking.

This experiment improves understanding of advanced linked list forms and demonstrates how circular structures differ from simple linear ones.

LAB EXPERIMENT 05

Reverse a String Using Stack

1. Problem Statement

Develop a Python program that reverses a given string by applying stack operations.
The program should use the basic LIFO mechanism to invert the order of characters.

2. Objective

This experiment aims to:

- **Understand how push and pop operations work in a stack.**
- **Use Python lists to simulate stack behavior.**
- **Reverse a string using the Last-In-First-Out principle.**
- **Strengthen concepts related to sequential data manipulation.**

3. Description

A stack is a restricted linear structure that allows insertion and removal from only one end.
It follows the LIFO rule, meaning the last added element is removed first.

To reverse a string using a stack:

1. **Each character of the string is pushed onto the stack.**
2. **Characters are then popped out one by one.**
3. **Since popping retrieves items in the opposite order, the final output becomes the reversed string.**

This experiment demonstrates a simple but effective use of stack logic in string manipulation.

4. Algorithm

Reverse a String Using Stack

1. **Start with an empty stack.**
2. **Traverse the input string and push each character onto the stack.**
3. **Create an empty string reversed_string.**
4. **While the stack contains elements:**
    - **Pop the top character.**
    - **Append it to reversed_string.**
5. **Return the final reversed string.**

5. Program Code

```python
def reverse_string(string):

    stack=[]

    for i in string:
        stack.append(i)

    reversed_string=""

    while stack:
        reversed_string += stack.pop()

    return reversed_string

Name="Kunal"
print("Original String:", Name)
print("Reversed String:", reverse_string(Name))
```

6. Sample Output

```
Original String: Kunal
Reversed String: lanuK
```

7. Conclusion

The program successfully reverses the input string using a stack.
It shows how stack operations naturally invert the sequence of elements, making them ideal for reversal tasks.
This reinforces the concept of LIFO and its use in real applications.

LAB EXPERIMENT 06

Check Balanced Parentheses Using Stack

1. Problem Statement

Design a Python program that determines whether an expression contains properly balanced and correctly nested parentheses.
The expression may include any combination of the following bracket types:

- ()
- []
- {}

The verification must be performed using the stack data structure.

2. Objective

The goals of this experiment are:

- To apply stack principles and understand LIFO operation.
- To validate expressions that contain different forms of brackets.
- To explore how stacks assist in parsing expressions used in programming languages and compilers.

3. Description

An expression is said to have **balanced parentheses** when:

- Every opening bracket has a corresponding closing bracket.
- Brackets close in the proper nested order.

Examples:

- Balanced → {[()]}, ([])
- Not Balanced → ([)], {], ((

The checking process uses a stack:

1. Each time an opening bracket appears, it is **pushed** onto the stack.
2. For a closing bracket:
    - The stack must contain at least one element.
    - The top element on the stack must be the matching opening bracket.
    - If matched, **pop** the stack; if not, the expression is invalid.
3. After scanning the entire expression:
    - If the stack is empty → the expression is balanced.
    - If elements remain → unbalanced.

This approach is widely used in expression validation, syntax checking, and interpreter design.

4. Algorithm

Balanced Parentheses Verification

1. Create an empty stack.
2. Read each character of the expression:
    - If it is an opening bracket → push it onto the stack.
    - If it is a closing bracket:
        - If the stack is empty → expression is unbalanced.
        - Compare the closing bracket with the top of stack:
            - If they match → pop the top element.
            - If they do not match → unbalanced.
3. After scanning the string:
    - If the stack is empty → the expression is balanced.
    - Otherwise → not balanced.

5. Program Code

```python
def check_paranthesis(input):
    stack=[]

    for i in input:
        if i in "([{":
            stack.append(i)

        elif i == ')':
            if not stack:
                return "Not Balanced"

            if stack[-1] != '(':
                return "Not Balanced"
            stack.pop()

        elif i == ']':
            if not stack:
                return "Not Balanced"

            if stack[-1] != '[':
                return "Not Balanced"
            stack.pop()

        elif i == '}':
            if not stack:
                return "Not Balanced"

            if stack[-1] != '{':
                return "Not Balanced"
            stack.pop()

    if not stack:
        return "Balanced"
    else:
        return "Not Balanced"

print(check_paranthesis("({[])"))
print(check_paranthesis("[]"))
print(check_paranthesis("["))
print(check_paranthesis("[]}"))
print(check_paranthesis("[()]"))
print(check_paranthesis("[({})]"))
```

6. Sample Output

```
Not Balanced
Balanced
Not Balanced
Not Balanced
Balanced
Balanced
```

7. Conclusion

This program effectively determines whether an expression has properly balanced parentheses by relying on stack operations.
It handles all three bracket types and correctly identifies mismatched or incomplete expressions.
Overall, the experiment highlights the importance of stacks in parsing, expression evaluation, and various compiler-related tasks.

LAB PROJECT

Inventory Stock Management System

1. Problem Statement

Develop a Python program that functions as a basic inventory manager. The system should allow users to add new products, locate items, delete existing records, verify stock levels, identify out-of-stock products, process sales, and generate stock-related insights such as total, average, and highest quantity.

2. Objective

The experiment is designed to:

- Implement a comprehensive menu-driven inventory application using Python.
- Utilize lists and dictionaries for storing structured product data.
- Practice CRUD-based operations: Create, Read, Update, and Delete.
- Perform analytical tasks including total stock, average availability, and maximum stock detection.
- Handle invalid inputs using exception management.
- Simulate real-world inventory behaviour such as stock depletion and zero-quantity alerts.

3. Description

An inventory management system maintains essential details for each product, such as SKU (unique identifier), product name, and available quantity.

In this program:

- The **inventory** is stored as a list.

- Each **product** is a dictionary structured like:
- {"sku": ..., "name": ..., "quantity": ...}
- Individual functions handle tasks such as inserting items, searching, deleting, stock checking, and sales handling.
- A user-friendly menu lets the user choose operations repeatedly.
- Input validation ensures smooth execution and prevents runtime errors.
- Analytical features include:
    - Listing all zero-quantity items
    - Computing total and average stock
    - Identifying items with the highest stock
    - Performing sales with proper stock verification

These operations make the program similar to a small-scale inventory system used in shops or warehouses.

4. Algorithm

A. Adding Products

1. Ask the user how many products they want to insert.
2. For each product:
    - Enter a SKU and ensure it’s not already in use.
    - Enter a valid product name.
    - Enter quantity and make sure it’s non-negative.
    - Store the product in the inventory list.

B. Searching Items

Search can be performed either using SKU or product name.

1. Request the search key.

2. Go through the inventory list.
3. If a match is found, show the full product details.

C. Deleting Items

Supports removing items by SKU or by name.

1. Look for the matching product.
2. Delete it from the inventory if found.

D. Selling Products

1. Ask for the SKU and number of units to sell.
2. Handle three cases:
    - Stock is sufficient → complete sale and reduce quantity.
    - Stock is available but less than requested → notify insufficient stock.
    - Stock is zero → display zero-stock warning.
3. Update inventory accordingly.

E. Zero Stock Check

1. Traverse the inventory.
2. List all products whose quantity is zero.

F. Stock Calculations

1. **Total stock** = Sum of all quantities.
2. **Average stock** = Total ÷ Number of items.

G. Highest Stock Item

1. Determine the largest quantity value.
2. Display all products whose quantity equals that maximum.

H. Display Inventory

Show all products stored in the inventory in a neat tabular format.

I. Main Menu Loop

- Display menu options 1–9.
- Execute the corresponding function based on user choice.
- Program ends when the user selects the exit option.

5. Program Code

```python
# Inventory list that will store all data
Inventory = []

# Inserting a new product
def insert_product():

        try:
            count = int(input("Enter how many products do you want to add:"))
        except ValueError:
            print("Invalid input. Please enter a number.")

        for i in range(count):
            print(f"\n--- Product {i+1} ---")
            sku = input("Enter SKU: ")

            # Check for duplicate SKU
            duplicate = False
            for item in Inventory:
                if item['sku'] == sku:
                    print("Product with this SKU already exists! Skipping...")
                    duplicate=True
                    break
            if duplicate:
                continue


            # Taking input for product name
            name = input("Enter Product Name: ").strip()
            while not name:  # keep asking until valid
                name = input("Please enter a valid Product Name: ").strip()


            # Taking input for quanity also doing exception handling
            try:
                quantity=int(input("Enter Quantity: "))
                if quantity<=0:
                    print("Quantity must be greater than 0. Skipping this product.")
                    continue

            except ValueError:
                print("Invalid input! Please enter quantity in numeric.")
                continue
```

```
43
44              # Creating product dictionary for adding it to inventory
45              product = {'sku':sku,'name':name,'quantity':quantity}
46              Inventory.append(product)
47
48              print("Product inserted successfully!")
49
50  # Function to search product by SKU
51  def search_by_sku():
52      if not Inventory:
53          print("Inventory is empty! Nothing to search.")
54          return
55
56      sku = input("Enter SKU to search: ")
57      found = False
58      for item in Inventory:
59          if item['sku'] == sku:
60              print("\nProduct Found:")
61              print(f"SKU: {item['sku']}, Name: {item['name']}, Quantity: {item['quantity']}")
62              found = True
63              break
64      if not found:
65          print("No product found with that SKU.")
66
67  # Function to search product by Name
68  def search_by_name():
69      if not Inventory:
70          print("Inventory is empty! Nothing to search.")
71          return
72
73      name = input("Enter Product Name to search: ").strip().lower()
74      found = False
75      for item in Inventory:
76          if item['name'].lower() == name:   # case-insensitive search
77              print("\nProduct Found:")
78              print(f"SKU: {item['sku']}, Name: {item['name']}, Quantity: {item['quantity']}")
79              found = True
80              break
81      if not found:
82          print("No product found with that Name.")
83
```

```
84  # Function to delete product by Name
85  def delete_by_name():
86      if not Inventory:
87          print("Inventory is empty! Nothing to delete.")
88          return
89
90      name = input("Enter Product Name of the product to delete: ").strip().lower()
91      found = False
92      for item in Inventory:
93          if item['name'].lower() == name:
94              Inventory.remove(item)   # delete the product
95              print(f"Product with Name '{item['name']}' deleted successfully.")
96              found = True
97              break
98      if not found:
99          print("No product found with that Name.")
100
101 # Function to delete product by SKU
102 def delete_by_sku():
103     if not Inventory:
104         print("Inventory is empty! Nothing to delete.")
105         return
106
107     sku = input("Enter SKU of the product to delete: ")
108     found = False
109     for item in Inventory:
110         if item['sku'] == sku:
111             Inventory.remove(item)   # delete the product
112             print(f"Product with SKU {sku} deleted successfully.")
113             found = True
114             break
115     if not found:
116         print("No product found with that SKU.")
117
118 #normal sale,insufficient stock and zero stock
119 def normal_sale(sku, qty):
120     for item in Inventory:
121         if item['sku']==sku:
122             if item['quantity'] >= qty:
123                 item['quantity'] -= qty
124                 print(f"Sale successful. Remaining stock of {item['name']}: {item['quantity']}")
125             elif 0 < item['quantity'] < qty:
126                 print(f"Insufficient stock. Available quantity of {item['name']}: {item['quantity']}")
```

```
            else:
                print(f"Stock of {item['name']} is zero.")
            return
        else:
            print("Product not found.")

#check zero stock
def check_zero_stock():
    zero_stock_items = [item for item in Inventory if item['quantity'] == 0]
    if zero_stock_items:
        print("Products with zero stock:")
        for item in zero_stock_items:
            print(f"SKU: {item['sku']}, Name: {item['name']}")
    else:
        print("No products with zero stock.")

# Function to calculate total and average stock
def total_and_avg(inventory):
    if not inventory:
        print("Inventory is empty.")
        return 0, 0
    total = sum(item['quantity'] for item in inventory)
    avg = total / len(inventory)
    print(f"Total stock: {total}, Average stock per product: {avg:.2f}")
    return total, avg

# Function to find the item with maximum stock
def show_max_quantity_products():
    if not Inventory:
        print("Inventory is empty.")
        return
    max_qty = max(item["quantity"] for item in Inventory)
    max_products = [item for item in Inventory if item["quantity"] == max_qty]
    print(f"\nProducts with maximum quantity ({max_qty}):")
    for p in max_products:
        print(f"SKU: {p['sku']}, Name: {p['name']}, Quantity: {p['quantity']}")

# Function to display Inventory
def display_inventory():
    if not Inventory:
        print("Inventory is empty!")

```

```
    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()

# Main Program loop
def main():
    while True:
        print("\nInventory Stock Manager")
        print("1. Insert New Product")
        print("2. Display Inventory")
        print("3. Search Product")
        print("4. Delete product")
        print("5. Sell a product")
        print("6. Check Zero quantity")
        print("7. Calculate Total and Average Stock")
        print("8. Find Maximum Stock Item")
        print("9. Exit")

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

        if choice == '1':
            insert_product()
        elif choice == '2':
            display_inventory()
        elif choice == '3':
            print("1. Search by SKU")
            print("2. Search by Name")
            option=int(input("Select the searching option:"))
            if option==1:
                search_by_sku()
            elif option==2:
                search_by_name()
            else:
                option=int(input("Invalid option! Please Enter again:"))

```

```
        elif choice == '4':
            print("1. Delete by SKU")
            print("2. Delete by Name")
            option=int(input("Select the Deleting option:"))
            if option==1:
                delete_by_sku()
            elif option==2:
                delete_by_name()

        elif choice == '5':
            sku = input("Enter SKU for sale: ")
            try:
                qty = int(input("Enter quantity to sell: "))
                normal_sale(sku, qty)
            except ValueError:
                print("Invalid input. Quantity must be a number.")

        elif choice == '6':
            check_zero_stock()

        elif choice =='7':
            total_and_avg(Inventory)

        elif choice == '8':
            show_max_quantity_products(Inventory)

        elif choice == '9':
            print("Exiting Inventory Manager.goodbye")
            break
        else:
            print("Invalid choice. Please select from 1 to 9.")

main()
```

6. Sample Output

```
Inventory Stock Manager
1. Insert New Product
2. Display Inventory
3. Search Product
4. Delete product
5. Sell a product
6. Check Zero quantity
7. Calculate Total and Average Stock
8. Find Maximum Stock Item
9. Exit
Enter your choice (1-6):
```

```
Enter your choice (1-6): 1
Enter how many products do you want to add:2

--- Product 1 ---
Enter SKU: 101
Enter Product Name: Pen
Enter Quantity: 5
Product inserted successfully!

--- Product 2 ---
Enter SKU: 102
Enter Product Name: pencil
Enter Quantity: 10
Product inserted successfully!
```

```
Enter your choice (1-6): 2

Current Inventory:
SKU             Product Name            Quantity
-----------------------------------------------
101             Pen             5
102             pencil          10
```

```
Enter your choice (1-6): 3
1. Search by SKU
2. Search by Name
Select the searching option:1
Enter SKU to search: 101

Product Found:
SKU: 101, Name: Pen, Quantity: 5
```

```
Enter your choice (1-6): 3
1. Search by SKU
2. Search by Name
Select the searching option:2
Enter Product Name to search: Pencil

Product Found:
SKU: 102, Name: pencil, Quantity: 10
```

```
Enter your choice (1-6): 5
Enter SKU for sale: 101
Enter quantity to sell: 2
Sale successful. Remaining stock of Pen: 3
```

```
Enter your choice (1-6): 6
No products with zero stock.
```

```
Enter your choice (1-6): 7
Total stock: 13, Average stock per product: 6.50
```

```
Enter your choice (1-6): 4
1. Delete by SKU
2. Delete by Name
Select the Deleting option:1
Enter SKU of the product to delete: 101
Product with SKU 101 deleted successfully.
```

```
Enter your choice (1-6): 2

Current Inventory:
SKU             Product Name            Quantity
-----------------------------------------------
102             Pencil          10
```

```
Enter your choice (1-6): 5
Enter SKU for sale: 102
Enter quantity to sell: 10
Sale successful. Remaining stock of Pencil: 0
```

```
Enter your choice (1-6): 6
Products with zero stock:
SKU: 102, Name: Pencil
```

```
Enter your choice (1-6): 9
Exiting Inventory Manager.goodbye
```

7. Time Complexity Summary

Operation	Complexity
Insert	O(n)
Search (SKU/Name)	O(n)

Operation	Complexity
Delete	O(n)
Sale	O(n)
Display	O(n)
Max Stock	O(n)

Advantages

- Easy-to-use, menu-driven structure.
- Suitable for beginners learning Python lists and dictionaries.
- Provides useful stock analytics.
- Includes input validation and exception handling.

Limitations

- No permanent data storage (data resets after program ends).
- Linear search and deletion operations (not optimized).

Future Improvements

- Add file handling to save inventory data permanently.
- Provide product update/editing options.
- Add sorting features (by name, quantity, SKU).
- Implement automatic SKU generation or scanning support.

Conclusion

The inventory management program performs all core functions effectively—adding, searching, deleting, selling, and analyzing products.
By using lists, dictionaries, modular functions, and proper input handling, the system becomes structured and user-friendly.
This experiment provides practical exposure to real-world inventory processing using Python.