



K.R. Mangalam University
School of Engineering & Technology

DATA STRUCTURE

Lab File

Submitted by:

Manjeet Yadav

2401420045

Btech-CSE (Data Science)

Submitted to:

Dr. Swati Gupta

INDEX

S. No.	Experiment Title	Page No.
1	Browser History Navigation System (Using Stack Concept)	1 – 6
2	Ticketing System Using Queue (Linear Queue Implementation)	7 – 10
3	Singly Linked List Operations (Insert, Delete, Search, Display)	11 – 16
4	Circular Singly Linked List (Insert, Search, Delete, Display)	17 – 23
5	Reverse a String Using Stack	24 – 25
6	Check Balanced Parentheses Using Stack	26 – 28
7	<i>Lab Project: Inventory Stock Management System</i>	29 – 37

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

```

1  history=[]
2  forward_history=[]
3  total_history=[]
4
5  # Adds a new page to the browser history
6  def insert_page(page):
7      history.append(page)
8
9      forward_history.clear()
10     total_history.append(page)
11
12     print(f"visited page:{page}")
13
14    # Moves one step back in history (like a browser back button)
15  def back_page():
16      if not history:
17          print("No pages in history.")
18          return
19
20      last_page=history.pop()
21      forward_history.append(last_page)
22
23      current = history[-1]
24      total_history.append(current)
25
26      print(f"Going back from {last_page}")
27
28      if history:
29          print(f"Current page: {history[-1]}")
30
31      else:
32          print("No pages left in history.")
33
34    # Moves forward again if the user previously went back
35  def forward_page():
36      if not forward_history:
37          print("No forward pages available.")
38          return

```

```

39
40      next_page = forward_history.pop()
41      history.append(next_page)
42      total_history.append(next_page)
43      print(f"Going again to {next_page}")
44
45    # Shows only the live browsing history (current path)
46  def view_history():
47      if not history:
48          print("NO pages in history!")
49
50      else:
51          print("History:", "->".join(history))
52
53    # Shows the complete history including revisits
54  def view_total_history():
55      if not history:
56          print("NO pages in history!")
57
58      else:
59          print("History:", "->".join(total_history))
60
61    # Displays statistics like total visits and frequency of each page
62  def history_stats():
63      if not total_history:
64          print("No total history to analyze!")
65          return
66
67      print(f"\nTotal visits: {len(total_history)}")
68      freq = {}
69      for page in total_history:
70          if page in freq:
71              freq[page] += 1
72          else:
73              freq[page] = 1
74

```

```

75     for page, count in freq.items():
76         print(f'Page '{page}' visited {count} time(s)')
77
78
79     while True:
80
81         print("\n1) Visit page")
82         print("2) Go back")
83         print("3) Go Forward")
84         print("4) View History")
85         print("5) view Total History")
86         print("6) View history stats")
87         print("7) Exit")
88         choice =int(input("Enter your choice:"))
89
90         if choice == 1:
91             page=str(input("Insert page name:"))
92             insert_page(page)
93
94         elif choice == 2:
95             back_page()
96
97         elif choice ==3:
98             forward_page()
99
100        elif choice == 4:
101            view_history()
102
103        elif choice ==5:
104            view_total_history()
105
106        elif choice == 6:
107            history_stats()
108
109        elif choice == 7:
110            print("Exiting Browser....Goodbye!")
111            break
112
113     else:
114         print("Invalid Choice")
115

```

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

```

1  class TicketingSystem:
2      def __init__(self,MAX):
3          self.MAX = MAX
4          self.queue = [None] * MAX
5          self.front = -1
6          self.rear = -1
7
8          # To check Overflow
9      def isFull(self):
10         return self.rear == self.MAX - 1
11
12         # To check Underflow
13     def isEmpty(self):
14         return self.front == -1 or self.front > self.rear
15
16         # To Insert a ticket
17     def enqueue(self, ticket_id):
18         if (self.isFull()):
19             print("Sorry the queue is full, can't add ticket: ", ticket_id)
20             return
21
22         if (self.front == -1):
23             self.front = 0
24
25         self.rear += 1
26         self.queue[self.rear] = ticket_id
27         print("Ticket Added: ", ticket_id)
28
29         # To delete a ticket
30     def dequeue(self):
31         if self.isEmpty():
32             print("Sorry the Queue is empty, can't remove any ticket")
33             return
34
35         removed_ticket = self.queue[self.front]
36         print("Removed ticket: ", removed_ticket)
37         self.front += 1
38

```

```

39     # To check size of queue
40     def size(self):
41         if self.isEmpty():
42             return 0
43
44         return (self.rear - self.front + 1)
45
46     # To display the whole Queue
47     def display(self):
48         if self.isEmpty():
49             print("Sorry, The Queue is Empty!")
50         else:
51             print("Current Queue:", self.queue[self.front:self.rear+1])
52
53
54     # Example to run
55     queue = TicketingSystem(5)
56
57     queue.enqueue("T-101")
58     queue.enqueue("T-102")
59     queue.enqueue("T-103")
60
61     queue.display()
62
63     queue.dequeue()
64
65     queue.display()
66
67     print("Size of queue:", queue.size())
68     print("Is Queue Full?", queue.isFull())
69     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

```

1  class Node:
2      def __init__(self, Element):
3
4          self.Element = Element
5          self.Link = None
6
7  class Linked_List:
8      def __init__(self):
9
10         self.Start = None
11
12
13
14     def insert_at_beginning(self, Element):
15
16         new_node = Node(Element)
17
18         new_node.Link = self.Start
19         self.Start = new_node
20
21

```

```

22
23     def insert_at_end(self, Element):
24
25         new_node = Node(Element)
26
27         if self.Start is None:
28
29             self.Start = new_node
30             return
31
32         temp = self.Start
33
34         while temp.Link:
35             temp = temp.Link
36
37         temp.Link = new_node
38
39     def insert_at_middle(self, Element, position):
40         new_node = Node(Element)
41
42         if position ==1:
43             new_node.Link = self.Start
44             self.Start = new_node
45             return
46
47         temp = self.Start
48         current_pos = 1
49
50         while temp is not None and current_pos < position - 1:
51             temp = temp.Link
52             current_pos += 1
53
54         if temp is None:
55             print("Position outside list range - inserting at end.")
56             self.insert_at_end(Element)
57             return
58

```

```

59     new_node.Link = temp.Link
60     temp.Link = new_node
61
62
63     def search(self, key):
64         temp = self.Start
65
66         while temp.Link:
67             if temp.Element == key:
68                 print("Found")
69                 return True
70
71             temp = temp.Link
72         print("Not Found")
73         return False
74
75
76
77     def delete_from_beginning(self):
78         if self.Start is None:
79             print("List is Empty!")
80             return
81
82         self.Start = self.Start.Link
83
84
85
86     def delete_from_end(self):
87         if self.Start is None:
88             print("List is Empty!")
89             return
90
91         if self.Start.Link is None:
92             self.Start = None
93             return
94

```

```

95     temp = self.Start
96     while(temp.Link.Link):
97         temp = temp.Link
98
99     temp.Link = None
100
101
102
103     def display(self):
104         temp = self.Start
105
106         while temp:
107             print(temp.Element, end=" -> ")
108             temp = temp.Link
109
110         print("None")
111
112
113
114     LL = Linked_List()
115
116     LL.insert_at_beginning(10)
117     LL.insert_at_beginning(20)
118     LL.insert_at_end(30)
119     LL.insert_at_end(40)
120     LL.display()
121
122     LL.insert_at_middle(50,2)
123     LL.display()
124
125     LL.search(20)
126     LL.search(2)
127
128     LL.delete_from_beginning()
129     LL.display()
130     LL.delete_from_end()
131     LL.display()

```

6. Sample Output

```
● 20 -> 10 -> 30 -> 40 -> None
20 -> 50 -> 10 -> 30 -> 40 -> None
Found
Not Found
50 -> 10 -> 30 -> 40 -> None
50 -> 10 -> 30 -> None
```

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:
 - o Point the node to itself
 - o Set it as Start.
3. Otherwise:
 - o Traverse to the last node.
 - o Link the new node to the old Start.
 - o Link last node to new node.
 - o 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

```
1  class Node:
2      def __init__(self, Element):
3          self.Element = Element
4          self.Link = None
5
6  class Circular_Linked_List:
7      def __init__(self):
8          self.Start = None
9
10
11
12
13     def insert_at_beginning(self, Element):
14
15         new_node = Node(Element)
16
17         if self.Start is None:
18
19             self.Start = new_node
20             new_node.Link = self.Start
21
22             return
23
24         temp = self.Start
25
26         while temp.Link != self.Start:
27             temp = temp.Link
28
29         new_node.Link = self.Start
30         temp.Link = new_node
31         self.Start = new_node
32
33
34     def insert_at_end(self, Element):
35
36         new_node = Node(Element)
37
38         if self.Start is None:
39
40             self.Start= new_node
41             new_node.Link = self.Start
42
43             return
44
45         temp = self.Start
46
47         while temp.Link != self.Start:
48             temp = temp.Link
49
50         temp.Link = new_node
51         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

```

```

127
128
129     def delete_from_end(self):
130         if self.Start is None:
131             print("List is Empty!")
132             return
133
134         if self.Start.Link is None:
135             self.Start = None
136             return
137
138         temp = self.Start
139         while(temp.Link.Link):
140             temp = temp.Link
141
142         temp.Link = None
143
144
145
146     def display(self):
147
148         if self.Start is None:
149
150             print("List is empty")
151             return
152
153         temp = self.Start
154
155         while True:
156
157             print(temp.Element, end=" -> ")
158             temp = temp.Link
159
160             if temp == self.Start:
161                 break
162
163         print("(back to Start)")
164
165
166     # Example usage
167     cll = Circular_Linked_List()
168
169     cll.insert_at_end(10)
170     cll.insert_at_end(20)
171     cll.insert_at_end(30)
172
173     print("After inserting 10, 20, 30 at end:")
174     cll.display()
175
176     cll.insert_at_beginning(5)
177     print("\nAfter inserting 5 at beginning:")
178     cll.display()
179
180     cll.insert_at_end(40)
181     print("\nAfter inserting 40 at end:")
182     cll.display()
183
184     cll.insert_at_middle(15, 4)
185     print("\nAfter inserting 15 at 4th position:")
186     cll.display()
187
188     print("\nSearching 20:")
189     cll.search(20)
190
191     print("\nSearching 99:")
192     cll.search(99)
193
194     print("\nDeleting from beginning:")
195     cll.delete_from_beginning()
196     cll.display()
197
198     print("\nDeleting from end:")
199     cll.delete_from_end()
200     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:
 - o Pop the top character.
 - o Append it to `reversed_string`.
5. Return the final reversed string.

5. Program Code

```

1  def reverse_string(string):
2
3      stack=[]
4
5      for i in string:
6          stack.append(i)
7
8      reversed_string=""
9
10     while stack:
11         reversed_string += stack.pop()
12
13     return reversed_string
14
15 Name="Kunal"
16 print("Original String:", Name)
17 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 $\rightarrow \{[()]\}, ([])$
- Not Balanced $\rightarrow ([]), \{[], ()$

The checking process uses a stack:

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

5. Program Code

```

1  def check_paranthesis(input):
2      stack=[]
3
4      for i in input:
5          if i in "([{":
6              stack.append(i)
7
8          elif i == ')':
9              if not stack:
10                  return "Not Balanced"
11
12              if stack[-1] != '(':
13                  return "Not Balanced"
14              stack.pop()
15
16          elif i == ']':
17              if not stack:
18                  return "Not Balanced"
19
20              if stack[-1] != '[':
21                  return "Not Balanced"
22              stack.pop()
23
24          elif i == '}':
25              if not stack:
26                  return "Not Balanced"
27
28              if stack[-1] != '{':
29                  return "Not Balanced"
30              stack.pop()
31
32      if not stack:
33          return "Balanced"
34      else:
35          return "Not Balanced"
36
37 print(check_paranthesis("{{}}"))
38 print(check_paranthesis("[]"))
39 print(check_paranthesis("["))
40 print(check_paranthesis("[]]"))
41 print(check_paranthesis("[()]]"))
42 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:
 - o Stock is sufficient → complete sale and reduce quantity.
 - o Stock is available but less than requested → notify insufficient stock.
 - o 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 \div 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

```

1  # Inventory list that will store all data
2  Inventory = []
3
4  # Inserting a new product
5  def insert_product():
6
7      try:
8          count = int(input("Enter how many products do you want to add:"))
9      except ValueError:
10         print("Invalid input. Please enter a number.")
11
12    for i in range(count):
13        print(f"\n--- Product {i+1} ---")
14        sku = input("Enter SKU: ")
15
16        # Check for duplicate SKU
17        duplicate = False
18        for item in Inventory:
19            if item['sku'] == sku:
20                print("Product with this SKU already exists! Skipping...")
21                duplicate=True
22                break
23        if duplicate:
24            continue
25
26
27        # Taking input for product name
28        name = input("Enter Product Name: ").strip()
29        while not name: # keep asking until valid
30            name = input("Please enter a valid Product Name: ").strip()
31
32
33        # Taking input for quantity also doing exception handling
34        try:
35            quantity=int(input("Enter Quantity: "))
36            if quantity<=0:
37                print("Quantity must be greater than 0. Skipping this product.")
38                continue
39
40        except ValueError:
41            print("Invalid input! Please enter quantity in numeric.")
42            continue

```

```

43     # Creating product dictionary for adding it to inventory
44     product = {'sku':sku,'name':name,'quantity':quantity}
45     Inventory.append(product)
46
47     print("Product inserted successfully!")
48
49 # Function to search product by SKU
50 def search_by_sku():
51     if not Inventory:
52         print("Inventory is empty! Nothing to search.")
53         return
54
55     sku = input("Enter SKU to search: ")
56     found = False
57     for item in Inventory:
58         if item['sku'] == sku:
59             print("\nProduct Found:")
60             print(f"SKU: {item['sku']}, Name: {item['name']}, Quantity: {item['quantity']}")
61             found = True
62             break
63     if not found:
64         print("No product found with that SKU.")
65
66 # Function to search product by Name
67 def search_by_name():
68     if not Inventory:
69         print("Inventory is empty! Nothing to search.")
70         return
71
72     name = input("Enter Product Name to search: ").strip().lower()
73     found = False
74     for item in Inventory:
75         if item['name'].lower() == name:    # case-insensitive search
76             print("\nProduct Found:")
77             print(f"SKU: {item['sku']}, Name: {item['name']}, Quantity: {item['quantity']}")
78             found = True
79             break
80     if not found:
81         print("No product found with that Name.")
82
83 # Function to delete product by Name
84 def delete_by_name():
85     if not Inventory:
86         print("Inventory is empty! Nothing to delete.")
87         return
88
89     name = input("Enter Product Name of the product to delete: ").strip().lower()
90     found = False
91     for item in Inventory:
92         if item['name'].lower() == name:
93             Inventory.remove(item)    # delete the product
94             print(f"Product with Name '{item['name']}' deleted successfully.")
95             found = True
96             break
97     if not found:
98         print("No product found with that Name.")
99
100 # Function to delete product by SKU
101 def delete_by_sku():
102     if not Inventory:
103         print("Inventory is empty! Nothing to delete.")
104         return
105
106     sku = input("Enter SKU of the product to delete: ")
107     found = False
108     for item in Inventory:
109         if item['sku'] == sku:
110             Inventory.remove(item)    # delete the product
111             print(f"Product with SKU {sku} deleted successfully.")
112             found = True
113             break
114     if not found:
115         print("No product found with that SKU.")
116
117 #normal sale,insufficient stock and zero stock
118 def normal_sale(sku, qty):
119     for item in Inventory:
120         if item['sku']==sku:
121             if item['quantity'] >= qty:
122                 item['quantity'] -= qty
123                 print(f"Sale successful. Remaining stock of {item['name']}: {item['quantity']}")
124             elif 0 < item['quantity'] < qty:
125                 print(f"Insufficient stock. Available quantity of {item['name']}: {item['quantity']}")
```

```

127     |     else:
128     |         print(f"Stock of {item['name']} is zero.")
129     |         return
130     else:
131         print("Product not found.")
132
133 #check zero stock
134 def check_zero_stock():
135     zero_stock_items = [item for item in Inventory if item['quantity'] == 0]
136     if zero_stock_items:
137         print("Products with zero stock:")
138         for item in zero_stock_items:
139             print(f"SKU: {item['sku']}, Name: {item['name']}")
140     else:
141         print("No products with zero stock.")
142
143 # Function to calculate total and average stock
144 def total_and_avg(inventory):
145     if not inventory:
146         print("Inventory is empty.")
147         return 0, 0
148     total = sum(item['quantity'] for item in inventory)
149     avg = total / len(inventory)
150     print(f"Total stock: {total}, Average stock per product: {avg:.2f}")
151     return total, avg
152
153 # Function to find the item with maximum stock
154 def show_max_quantity_products():
155     if not Inventory:
156         print("Inventory is empty.")
157         return
158     max_qty = max(item["quantity"] for item in Inventory)
159     max_products = [item for item in Inventory if item["quantity"] == max_qty]
160     print(f"\nProducts with maximum quantity ({max_qty}):")
161     for p in max_products:
162         print(f"SKU: {p['sku']}, Name: {p['name']}, Quantity: {p['quantity']}")
163
164 # Function to display Inventory
165 def display_inventory():
166     if not Inventory:
167         print("Inventory is empty!")
168
169     print("\nCurrent Inventory:")
170     print("SKU\t\tProduct Name\t\tQuantity")
171     print("-----")
172
173     for item in Inventory:
174         print(f"{item['sku']}\t{item['name']}\t{item['quantity']}")
175     print()
176
177 # Main Program loop
178 def main():
179     while True:
180         print("\nInventory Stock Manager")
181         print("1. Insert New Product")
182         print("2. Display Inventory")
183         print("3. Search Product")
184         print("4. Delete product")
185         print("5. Sell a product")
186         print("6. Check Zero quantity")
187         print("7. Calculate Total and Average Stock")
188         print("8. Find Maximum Stock Item")
189         print("9. Exit")
190
191         choice = input("Enter your choice (1-6): ")
192
193         if choice == '1':
194             insert_product()
195         elif choice == '2':
196             display_inventory()
197         elif choice == '3':
198             print("1. Search by SKU")
199             print("2. Search by Name")
200             option=int(input("Select the searching option:"))
201             if option==1:
202                 search_by_sku()
203             elif option==2:
204                 search_by_name()
205             else:
206                 option=int(input("Invalid option! Please Enter again:"))

```

```

208     elif choice == '4':
209         print("1. Delete by SKU")
210         print("2. Delete by Name")
211         option=int(input("Select the Deleting option:"))
212         if option==1:
213             delete_by_sku()
214         elif option==2:
215             delete_by_name()
216
217     elif choice == '5':
218         sku = input("Enter SKU for sale: ")
219         try:
220             qty = int(input("Enter quantity to sell: "))
221             normal_sale(sku, qty)
222         except ValueError:
223             print("Invalid input. Quantity must be a number.")
224
225     elif choice == '6':
226         check_zero_stock()
227
228     elif choice == '7':
229         total_and_avg(Inventory)
230
231     elif choice == '8':
232         show_max_quantity_products(Inventory)
233
234     elif choice == '9':
235         print("Exiting Inventory Manager.goodbye")
236         break
237     else:
238         print("Invalid choice. Please select from 1 to 9.")
239
240 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.