K.R. Mangalam University

School of Engineering & Technology

DATA STRUCTURE

Lab File

Submitted by:

Kunal Chadhary

2401420049

Btech-CSE (Data Science)

Submitted to:

Dr. Swati Gupta

INDEX

LAB EXPERIMENT 01

Browser History Navigation System (Using Stack Concept)

1. Problem Statement

Build a Python program that simulates a web browser's navigation system. The user should be able to visit new pages, go back, move forward, view current history, view total history, and analyze visit statistics.

2. Objective

- To implement stack-based browser navigation using Python lists.
- To understand back-stack and forward-stack functionality.
- To track total browsing history independent of back/forward actions.
- To generate frequency statistics of visited pages.
- To build an interactive, menu-driven simulation.

3. Description

Modern browsers maintain two internal structures:

1. **Back History (Stack 1):**
   Stores pages visited before the current page.
   When you click "Back", the current page moves to the forward stack.
2. **Forward History (Stack 2):**
   Stores pages that can be visited again after going back.
   When a new page is visited, forward stack clears (just like real browsers).
3. **Total History:**
   A separate list used to track **every visit**, including revisits.

In this program:

- history works like the **back-stack**.

- forward_history works like the **forward-stack**.
- total_history maintains the **complete visitation record**.

This model helps understand how stack operations power browser navigation.

4. Algorithm

A. Visit Page (insert_page)

1. Add new page to history stack.
2. Clear forward history (new visit invalidates forward pages).
3. Add to total history.
4. Display the visited page.

B. Go Back (back_page)

1. Check if back history exists.
2. Pop last page and move it to forward history.
3. Current page becomes the previous one in history.
4. Add current page to total history.
5. Display transition.

C. Go Forward (forward_page)

1. Check if forward stack has a page.
2. Pop from forward history and push to back history.
3. Add to total history.
4. Display transition.

D. View History (view_history)

- Print all pages in the back-stack.

E. View Total History (view_total_history)

- Print every page ever visited in order.

F. History Statistics (history_stats)

1. Count total number of visits.
2. Calculate frequency of each page.
3. Display stats.

G. Main Menu Loop

1. Display options: visit, back, forward, view, stats, exit.
2. Execute user's selected operation.
3. Continue until exit.

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
```

```
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

The program successfully simulates browser-like navigation using stack operations.
It demonstrates how back-forward logic works internally and how total history is maintained separately.
The implementation also shows good use of lists, operations like push/pop, and simple frequency analysis.

LAB EXPERIMENT 02

Ticketing System Using Queue (Linear Queue Implementation)

1. Problem Statement

Create a Python program that simulates a simple ticketing system using the **queue data structure**.
The system should allow adding tickets, removing tickets, viewing the queue, checking size, and identifying overflow/underflow conditions.

2. Objective

- To understand queue operations (enqueue, dequeue) in a linear queue.
- To implement a queue using a **fixed-size array**.
- To detect and handle **overflow** and **underflow** conditions.
- To work with class-based implementation in Python.
- To display queue contents and size in real time.

3. Description

A **queue** is a linear data structure that follows the **FIFO (First In, First Out)** principle.
The first element inserted is the first one removed — similar to customers standing in line.

In this ticketing system:

- The queue is represented using a **Python list of fixed size**.
- front represents the index of the current first ticket.
- rear represents the index where the next ticket will be added.
- enqueue() inserts a new ticket at the rear.
- dequeue() removes a ticket from the front.

Special conditions:

- **Overflow:** Happens when rear == MAX − 1.
- **Underflow:** Happens when queue is empty (front > rear or front == -1).

This experiment models a real-world scenario such as ticket counters, customer service queues, etc.

4. Algorithm

A. Enqueue (Insert Ticket)

1. Check if queue is full.
2. If inserting first element, set front = 0.
3. Increase rear and insert ticket.
4. Display confirmation.

B. Dequeue (Remove Ticket)

1. Check if queue is empty.
2. Remove ticket at front.
3. Increment front to point to next element.
4. Display removed ticket.

C. isFull

- Return true if rear == MAX - 1.

D. isEmpty

- Return true if queue has no valid elements (front == -1 or front > rear).

E. size

- Return number of elements using:
  (rear - front + 1)

F. display

- If queue has elements, show from front up to rear.

5. Program Code

```python
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

```

```python
    # 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

7. Time Complexities

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

Real-Life Uses

- Customer service counters
- Bank token system
- Ticket booths
- Printers' job scheduling

8. Observations / Conclusion

The ticketing system clearly demonstrates queue operations and constraint handling.
The program successfully performs enqueue, dequeue, and queue inspection while preventing overflow/underflow errors.
This reinforces understanding of linear queue implementation using arrays.

LAB EXPERIMENT 03

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

1. Problem Statement

Develop a Python program to implement a **singly linked list** supporting basic operations such as insertion (beginning, end, middle), deletion (beginning, end), searching an element, and displaying the list.

2. Objective

- To understand the internal working of linked lists using nodes.
- To perform insertion at multiple positions.
- To implement deletion operations.
- To search elements sequentially.
- To visualize dynamic memory linking through pointers.
- To practice class-based implementation of a linked list.

3. Description

A **linked list** is a dynamic linear data structure where each element (node) contains:

- **Element / Data**
- **Link / Pointer** to the next node

The **first node** is called the **Start (Head)**.
Unlike arrays, linked lists do **not store elements in contiguous memory**, making insertions and deletions more efficient (no shifting required).

This program implements:

Node structure **11**

| Element | Link → next node |

Supported operations:

- Insert at beginning
- Insert at end
- Insert at a given position
- Search for an element
- Delete from beginning
- Delete from end
- Display list

This experiment helps understand self-referential structures and pointer-like behavior in Python.

4. Algorithm

A. Insert at Beginning

1. Create a new node.
2. Point its Link to current Start.
3. Update Start to new node.

B. Insert at End

1. Create a new node.
2. If list empty → new node becomes Start.
3. Otherwise traverse to last node and link it to new node.

C. Insert at Middle (Position-based)

1. If position = 1 → insert at beginning.
2. Traverse until position − 1.
3. Insert new node in between nodes.

4. If position exceeds length → append at end.

D. Search

1. Traverse from Start.
2. Compare each node's element with key.
3. Return result.

E. Delete from Beginning

1. If list empty → print error.
2. Move Start to next node.

F. Delete from End

1. If list empty → error.
2. If only one node → set Start to None.
3. Otherwise find second-last node and set its link to None.

G. Display

1. Traverse from Start.
2. Print each node until None.

5. Program Code

```
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


```

```
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_beggining(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
```

```
        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

```
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)

Operation	Complexity
Display	O(n)

8. Conclusion

The linked list program successfully performs insertion at various positions, deletion, search, and traversal.
This demonstrates how dynamic memory links nodes and how pointer manipulation affects the structure.
It also highlights how linked lists differ from arrays in flexibility and memory usage.

LAB EXPERIMENT 04

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

1. Problem Statement

Create a Python program to implement a **Circular Singly Linked List** that supports the following operations:

- Insertion at the beginning
- Insertion at the end
- Insertion at a specific position
- Searching for an element
- Deleting from beginning
- Deleting from end
- Displaying the list

2. Objective

- To understand circular linked list structure and pointer management.
- To implement node-based dynamic memory linking.
- To perform insert, delete, and search operations.
- To visualize how the last node loops back to the first node.
- To practice traversal in circular structures.

3. Theory / Description

A Circular Linked List is a variation of a singly linked list in which the last node points back to the first node, forming a loop.

Node Structure

| Element | Link → Next Node |

Key Features of CLL

- No node has None as the next link.
- Traversal continues until the pointer returns to the Start node.
- Useful in round-robin scheduling, cyclic data access, multiplayer turn systems, etc.

Operations Implemented:

- Insert at beginning
- Insert at end
- Insert at middle (given position)
- Search element
- Delete from beginning
- Delete from end
- Display list with circular indication

This program covers almost all fundamental operations of a CLL.

4. Algorithm

A. Insert at Beginning

1. Create new node.
2. If list empty:
    - Point node to itself and set Start.
3. Else:
    - Traverse to last node.
    - Point new node → Start

- Last node → new node
- Update Start to new node.

B. Insert at End

1. Create new node.
2. If list empty: create self-link and set Start.
3. Else traverse to last node and attach new node.
4. Point new node back to Start.

C. Insert at Middle (Position-Based)

1. If list empty → insert as first node.
2. If position = 1 → call beginning insertion.
3. Traverse until position − 1 or reaching Start again.
4. If position out of range → insert at end.
5. Insert node between nodes.

D. Search

1. Start at Start node.
2. Traverse until loop returns to Start.
3. Compare each element with key.
4. Print “Found” or “Not Found”.

E. Delete from Beginning

1. If list empty → cannot delete.
2. If only one node → remove and set Start to None.
3. Traverse to last node.
4. Point last node → second node.
5. Update Start to second node.

F. Delete from End

1. If list empty → no deletion.
2. If only one node → empty the list.
3. Traverse to second-last node.
4. Point its link to Start.

G. Display

1. Start at Start node.
2. Print each element until pointer returns to Start.
3. End with (back to Start) for clarity.

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 linked list correctly maintains the looped structure during all operations.
Insertion, deletion, and search work at all positions.
Traversal stops when coming back to Start, showing proper circular linking.
This experiment strengthens understanding of dynamic node-based structures beyond linear linked lists.

LAB EXPERIMENT 05

Reverse a String Using Stack

1. Problem Statement

Write a Python program to reverse a string using the concept of a **stack**.

2. Objective

- To understand stack operations (push and pop).
- To use a list as a stack for character-level processing.
- To reverse a string using Last-In-First-Out (LIFO) behavior.
- To practice basic string and list manipulation in Python.

3. Description

A **stack** is a linear data structure based on the **LIFO (Last In, First Out)** principle. When reversing a string:

1. Each character is **pushed** onto the stack.
2. Characters are then **popped** one by one.
3. Since popping returns elements in reverse order, the result becomes the reversed string.

This demonstrates how stack logic can be used for string manipulation.

4. Algorithm

Reverse String Using Stack

1. Create an empty stack.
2. Traverse each character in the string and push it onto the stack.
3. Initialize an empty string reversed_string.

4. While stack is not empty:
    - Pop characters and append them to reversed_string.
5. Return the 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

Write a Python program to check whether a given expression has **balanced parentheses**, including:

- ()
- []
- {}

The program should use the **stack** data structure to validate the expression.

2. Objective

- To understand stack operations and LIFO behavior.
- To validate expressions containing multiple types of brackets.
- To learn how stacks are used in expression parsing and compiler design.

3. Description

Balanced parentheses mean:

- Every opening bracket must have a matching closing bracket.
- Brackets must close **in the correct order** (nested structure).

Example:

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

The stack helps track opening brackets:

1. Push every opening bracket onto the stack.

2. When a closing bracket appears:
    - The stack must not be empty.
    - The top element must be the correct matching opening bracket.
    - Pop if it matches.
3. At end:
    - If the stack is empty → Balanced
    - Else → Not Balanced

This is a classic application of stacks in compilers and syntax validation.

4. Algorithm

Check Balanced Parentheses

1. Initialize empty stack.
2. For each character in expression:
    - If it's an opening bracket → push to stack.
    - If it's a closing bracket:
        - If stack empty → not balanced.
        - Check top of stack for matching bracket.
        - If matches → pop, else → not balanced.
3. After traversal:
    - If stack empty → Balanced
    - Else → 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

The program accurately checks whether parentheses are balanced using stack logic.
It handles all three bracket types and properly detects mismatches or incomplete expressions.
This experiment demonstrates real-world uses of stacks in expression evaluation and syntax checking.

LAB PROJECT

Inventory Stock Management System

1. Problem Statement

Create a Python-based inventory management system that supports adding products, searching items, deleting records, checking stock levels, tracking zero quantity items, performing sales, and generating stock analytics such as total, average, and maximum quantity.

2. Objective

- To implement an extended menu-driven inventory program using Python.
- To practice list and dictionary usage for structured data handling.
- To implement CRUD operations (Create, Read, Update, Delete).
- To perform stock analysis (total stock, average stock, max stock).
- To handle exceptions and validate user inputs.
- To simulate real-world inventory behaviour like sales and zero-stock checking.

3. Description

Inventory management systems maintain product information such as **SKU**, **name**, and **quantity**.
In this program:

- **List** stores all products (Inventory = []).
- **Each product** is represented as a **dictionary**:
  {"sku": ..., "name": ..., "quantity": ...}
- Functions provide modular operations like inserting, searching, deleting, and checking stock.

- **Menu-driven approach** allows the user to repeatedly choose operations.
- **Validation & exception handling** ensure the program doesn't crash due to invalid input.
- The program also provides **analytical features** such as:
    - Checking zero-stock items
    - Calculating total & average stock
    - Finding the product(s) with maximum stock
    - Simulating product sales with different outcomes (normal sale, insufficient stock, zero stock)

This makes the system closer to a real small-scale inventory management application.

4. Algorithm

A. Insert Product

1. Ask number of products to be added.
2. For each product:
    - Input SKU and check for duplicates.
    - Input valid product name.
    - Input quantity and ensure it's a positive number.
    - Store product in inventory as a dictionary.

B. Search Product

- **By SKU / By Name**

1. Input key (SKU or name).
2. Traverse inventory.
3. If matched, display product details.

C. Delete Product

- Supports both **delete by SKU** and **delete by Name**.

1. Search inventory for match.
2. Remove product if found.

D. Selling a Product

1. Input SKU and quantity to sell.
2. Check three conditions:
    - Quantity ≥ requested → normal sale
    - Quantity > 0 but less → insufficient stock
    - Quantity = 0 → zero stock
3. Update inventory accordingly.

E. Check Zero Stock

1. Loop through inventory.
2. Display all items with quantity 0.

F. Total & Average Stock

1. Total = sum of all quantities.
2. Average = total / number of products.

G. Maximum Stock Item

1. Find the highest quantity.
2. Display all items that match that quantity.

H. Display Inventory

1. Print all product details in tabular form.

I. Main Menu Loop

- Repeatedly show options 1–9
- Perform selected operation
- Exit when user chooses option 9

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
```

```python

            # Creating product dictionary for adding it to inventory
            product = {'sku':sku,'name':name,'quantity':quantity}
            Inventory.append(product)

            print("Product inserted successfully!")

# Function to search product by SKU
def search_by_sku():
    if not Inventory:
        print("Inventory is empty! Nothing to search.")
        return

    sku = input("Enter SKU to search: ")
    found = False
    for item in Inventory:
        if item['sku'] == sku:
            print("\nProduct Found:")
            print(f"SKU: {item['sku']}, Name: {item['name']}, Quantity: {item['quantity']}")
            found = True
            break
    if not found:
        print("No product found with that SKU.")

# Function to search product by Name
def search_by_name():
    if not Inventory:
        print("Inventory is empty! Nothing to search.")
        return

    name = input("Enter Product Name to search: ").strip().lower()
    found = False
    for item in Inventory:
        if item['name'].lower() == name:   # case-insensitive search
            print("\nProduct Found:")
            print(f"SKU: {item['sku']}, Name: {item['name']}, Quantity: {item['quantity']}")
            found = True
            break
    if not found:
        print("No product found with that Name.")

```

```python
# Function to delete product by Name
def delete_by_name():
    if not Inventory:
        print("Inventory is empty! Nothing to delete.")
        return

    name = input("Enter Product Name of the product to delete: ").strip().lower()
    found = False
    for item in Inventory:
        if item['name'].lower() == name:
            Inventory.remove(item)   # delete the product
            print(f"Product with Name '{item['name']}' deleted successfully.")
            found = True
            break
    if not found:
        print("No product found with that Name.")

# Function to delete product by SKU
def delete_by_sku():
    if not Inventory:
        print("Inventory is empty! Nothing to delete.")
        return

    sku = input("Enter SKU of the product to delete: ")
    found = False
    for item in Inventory:
        if item['sku'] == sku:
            Inventory.remove(item)   # delete the product
            print(f"Product with SKU {sku} deleted successfully.")
            found = True
            break
    if not found:
        print("No product found with that SKU.")

#normal sale,insufficient stock and zero stock
def normal_sale(sku, qty):
    for item in Inventory:
        if item['sku']==sku:
            if item['quantity'] >= qty:
                item['quantity'] -= qty
                print(f"Sale successful. Remaining stock of {item['name']}: {item['quantity']}")
            elif 0 < item['quantity'] < qty:
                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)
Delete	O(n)
Sale	O(n)
Display	O(n)
Max Stock	O(n)

Advantages

- Simple, interactive, and menu-driven
- Good for beginners learning data structures
- Supports stock analytics
- Handles errors gracefully

Limitations

- No database support (data lost on exit)
- Search and delete are linear (can be optimized using dict or map)

Future Enhancements

- Add file handling to save inventory permanently
- Add update product details feature
- Add sorting options
- Add barcode/SKU auto-generator

7. Conclusion

This program successfully manages inventory in a structured way.
It covers all essential operations of a real inventory system — insertion, deletion, search, sale, and stock analysis.
The use of lists, dictionaries, functions, and exception handling makes the solution modular and robust.