LAB MANUAL

B. TECH CSE (Data Science)

SOET

(School of Engineering &Technology)

ENCS253-Data Structures Lab

Submitted by- Sakshi

Roll No - 2401420031

Submitted to- Swati Mam

Experiment Index

Experiment 1: Inventory Management System

Objective:

Design and implement an inventory tracking system using Python lists and dictionaries to manage product data, process sales transactions, and perform stock analysis operations.

Learning Outcomes:

✓ Master Python list and dictionary data structures for practical applications

✓ Implement CRUD (Create, Read, Update, Delete) operations effectively

✓ Apply data validation techniques and implement error handling mechanisms

✓ Analyze inventory data and generate meaningful business reports

✓ Understand product lifecycle management concepts

Theory:

An inventory management system is a critical component of retail and warehouse operations. It maintains comprehensive product records with attributes including SKU (Stock Keeping Unit), product name, and available quantity. The system supports multiple operations such as adding new products to stock, removing discontinued items, processing customer sales transactions, and analyzing stock levels. This enables businesses to track merchandise efficiently, prevent stockouts, minimize excess inventory, and make data-driven decisions about purchasing and distribution.

Key Operations:

▶ insert_product() - Add new product to inventory with validation

▶ display_inventory() - List all products in current stock

▶ process_sale() - Handle customer transactions and reduce stock

▶ identify_zero_stock() - Locate products that need reordering

▶ total_and_avg_stock() - Calculate stock statistics and trends

▶ max_stock_item() - Find product with highest quantity

Algorithm:

1. Initialize empty list to store products

2. Each product stored as dictionary with SKU, name, quantity

3. For insertion: check duplicate SKU, validate inputs, append if valid

4. For display: iterate through list and format output

5. For sales: find product by SKU, update quantity if sufficient stock

6. For analysis: iterate through all products to compute aggregates

Code:

```
# Inventory Management System

inventory = []

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

    # Prevent duplicate SKU
    for item in inventory:
        if item['sku'] == sku:
            print("Product with this SKU already exists!")
            return

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

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

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


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

```
    print("\n===== Current 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()


def process_sale():
    sku = input("Enter SKU for sale: ")
    try:
        qty_sold = int(input("Enter quantity sold: "))
    except ValueError:
        print("Invalid input. Quantity must be a number.")
        return

    for item in inventory:
        if item['sku'] == sku:
            if item['quantity'] >= qty_sold:
                item['quantity'] -= qty_sold
                print(f"Sale processed: {qty_sold} units of SKU {sku}.")
            else:
                print(f"Insufficient stock for SKU {sku}. Available: {item['quantity']}")
            return

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


def identify_zero_stock():
    zero_stock = [item['sku'] for item in inventory if item['quantity'] == 0]
    if zero_stock:
        print(f"Zero stock SKUs: {zero_stock}")
    else:
        print("No zero stock items found.")
```

```python
def total_and_avg_stock():
    if not inventory:
        print("Inventory is empty.")
        return

    total = sum(item['quantity'] for item in inventory)
    avg = total / len(inventory)
    print(f"Total Stock: {total}")
    print(f"Average Stock: {avg:.2f}")


def max_stock_item():
    if not inventory:
        print("Inventory is empty.")
        return

    max_item = max(inventory, key=lambda x: x['quantity'])
    print(f"Item with Maximum Stock: SKU {max_item['sku']}, "
          f"Name {max_item['name']}, Quantity {max_item['quantity']}")


def main():
    while True:
        print("\n===== Inventory Management Menu =====")
        print("1. Insert New Product")
        print("2. Display Inventory")
        print("3. Process Sale")
        print("4. Identify Zero Stock Items")
        print("5. Calculate Total & Average Stock")
        print("6. Find Maximum Stock Item")
        print("7. Exit")

        choice = input("Enter your choice (1-7): ")
```

```
        if choice == '1':
            insert_product()
        elif choice == '2':
            display_inventory()
        elif choice == '3':
            process_sale()
        elif choice == '4':
            identify_zero_stock()
        elif choice == '5':
            total_and_avg_stock()
        elif choice == '6':
            max_stock_item()
        elif choice == '7':
            print("Exiting Inventory Manager. Goodbye!")
            break
        else:
            print("Invalid choice. Please select from 1 to 7.")


if __name__ == "__main__":
    main()
```

Sample Output:

```
===== Inventory Management Menu =====
1. Insert New Product
2. Display Inventory
3. Process Sale
4. Identify Zero Stock Items
5. Calculate Total & Average Stock
6. Find Maximum Stock Item
7. Exit
Enter your choice (1-7):  1
Enter SKU:  201
Enter Product Name:  Tea
Enter Quantity:  45
Product inserted successfully.

===== Inventory Management Menu =====
1. Insert New Product
2. Display Inventory
3. Process Sale
4. Identify Zero Stock Items
5. Calculate Total & Average Stock
6. Find Maximum Stock Item
7. Exit
Enter your choice (1-7):  1
Enter SKU:  202
Enter Product Name:  Coffee
Enter Quantity:  56
Product inserted successfully.

===== Inventory Management Menu =====
1. Insert New Product
2. Display Inventory
3. Process Sale
4. Identify Zero Stock Items
5. Calculate Total & Average Stock
6. Find Maximum Stock Item
7. Exit
Enter your choice (1-7):  2
```

```
===== Current Inventory =====
SKU             Product Name            Quantity
-----------------------------------------
201             Tea             45
202             Coffee          56


===== Inventory Management Menu =====
1. Insert New Product
2. Display Inventory
3. Process Sale
4. Identify Zero Stock Items
5. Calculate Total & Average Stock
6. Find Maximum Stock Item
7. Exit
Enter your choice (1-7):  3
Enter SKU for sale:  201
Enter quantity sold:  4
Sale processed: 4 units of SKU 201.

===== Inventory Management Menu =====
1. Insert New Product
2. Display Inventory
3. Process Sale
4. Identify Zero Stock Items
5. Calculate Total & Average Stock
6. Find Maximum Stock Item
7. Exit
Enter your choice (1-7):  4
No zero stock items found.

===== Inventory Management Menu =====
1. Insert New Product
2. Display Inventory
3. Process Sale
4. Identify Zero Stock Items
5. Calculate Total & Average Stock
6. Find Maximum Stock Item
7. Exit
Enter your choice (1-7):  5
Total Stock: 97
Average Stock: 48.50
```

```
===== Inventory Management Menu =====
1. Insert New Product
2. Display Inventory
3. Process Sale
4. Identify Zero Stock Items
5. Calculate Total & Average Stock
6. Find Maximum Stock Item
7. Exit
Enter your choice (1-7):  6
Item with Maximum Stock: SKU 202, Name Coffee, Quantity 56

===== Inventory Management Menu =====
1. Insert New Product
2. Display Inventory
3. Process Sale
4. Identify Zero Stock Items
5. Calculate Total & Average Stock
6. Find Maximum Stock Item
7. Exit
Enter your choice (1-7):  7
Exiting Inventory Manager. Goodbye!
```

Time Complexity Analysis:

• Insert Product: O(n) - Must check all items for duplicate SKU

• Display Inventory: O(n) - Traverse all n products

• Process Sale: O(n) - Linear search to find product by SKU

• Identify Zero Stock: O(n) - Check all products for zero quantity

• Calculate Total/Average: O(n) - Sum all quantities

• Find Max Stock: O(n) - Compare all quantities

• Space Complexity: O(n) - Store n products with attributes

Experiment 2: Browser History Navigation System (Using Stack Concept)

Objective:

Implement a browser history navigation system using stack data structure to simulate back button functionality, enabling users to navigate through previously visited web pages.

Learning Outcomes:

✓ Understand stack LIFO principle and real-world applications

✓ Implement browser history using stack operations

✓ Handle navigation through history stack

✓ Develop interactive history management interface

✓ Apply stack to practical software features

Theory:

Web browsers use stack data structure to maintain browsing history. When a user visits a webpage, it is pushed onto the history stack. When the back button is clicked, the current page is popped from the stack, revealing the previously visited page. This LIFO (Last In First Out) principle provides an intuitive browsing experience. The forward history can be managed with another stack. This demonstrates how fundamental data structures power everyday software features.

Key Operations:

▶ visit_page(url) - Add new page to history

▶ go_back() - Return to previous page

▶ current_page() - Show currently displayed page

▶ history_list() - Display all visited pages

▶ is_empty() - Check if history is empty

Algorithm:

1. Create a stack to store browser history

2. When user visits a page: push page onto stack

3. When user clicks back: pop from stack to get previous page

4. Current page is always at stack top

5. Display current page or full history on request

Code:

```
# Browser Navigation using Stacks

history = []
forward_stack = []
visit_count = {}

def increase_visit(page):
    """Updates visit count of page"""
    visit_count[page] = visit_count.get(page, 0) + 1

def visit_page(page):
    history.append(page)                # push page to history
    forward_stack.clear()               # clear forward stack on new visit
    increase_visit(page)
    print(f"\n Visited: {page} | Total visits: {visit_count[page]}")

def go_back():
    if not history:
        print("\n No pages in history.")
        return

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

    if history:
        current_page = history[-1]
        increase_visit(current_page)
        print(f"\n← Going back from: {last_page}")
        print(f" Current page: {current_page} | Total visits: {visit_count[current_page]}")
    else:
        print(f"\n← Going back from: {last_page}")
        print("⚠ No pages left in history.")
```

```python
def go_forward():
    if not forward_stack:
        print("\n No forward pages available.")
        return

    next_page = forward_stack.pop()
    history.append(next_page)
    increase_visit(next_page)
    print(f"\n➡ Moved forward to: {next_page} | Total visits: {visit_count[next_page]}")

def show_history():
    print("\n----- History Info -----")
    print("History:" if history else "History is empty.")
    if history:
        print(" -> ".join(history))

    print("\nForward:" if forward_stack else "\nNo forward pages.")
    if forward_stack:
        print(" -> ".join(forward_stack))
    print("-----------------------")

def show_visit_frequency():
    print("\n Visit Frequency")
    if not visit_count:
        print("No pages visited yet.")
    else:
        for page, count in visit_count.items():
            print(f"{page} → {count} times")
```

```python
# ---------------- Menu ----------------
while True:
    print("\n===== Browser Menu =====")
    print("1. Visit Page")
    print("2. Back")
    print("3. Forward")
    print("4. Show History")
    print("5. Show Visit Frequency")
    print("6. Exit")

    choice = input("Enter choice: ")

    if choice == "1":
        page = input("Enter page name: ")
        visit_page(page)
    elif choice == "2":
        go_back()
    elif choice == "3":
        go_forward()
    elif choice == "4":
        show_history()
    elif choice == "5":
        show_visit_frequency()
    elif choice == "6":
        print("\n Exiting browser simulation. Goodbye!")
        break
    else:
        print(" Invalid choice. Try again.")
```

Sample Output:

```
===== Browser Menu =====
1. Visit Page
2. Back
3. Forward
4. Show History
5. Show Visit Frequency
6. Exit
Enter choice:  1
Enter page name:  Linkedin

 Visited: Linkedin | Total visits: 1

===== Browser Menu =====
1. Visit Page
2. Back
3. Forward
4. Show History
5. Show Visit Frequency
6. Exit
Enter choice:  1
Enter page name:  Github

 Visited: Github | Total visits: 1

===== Browser Menu =====
1. Visit Page
2. Back
3. Forward
4. Show History
5. Show Visit Frequency
6. Exit
Enter choice:  1
Enter page name:  Twitter

 Visited: Twitter | Total visits: 1
```

```
===== Browser Menu =====
1. Visit Page
2. Back
3. Forward
4. Show History
5. Show Visit Frequency
6. Exit
Enter choice:  2

← Going back from: Twitter
 Current page: Github | Total visits: 2

===== Browser Menu =====
1. Visit Page
2. Back
3. Forward
4. Show History
5. Show Visit Frequency
6. Exit
Enter choice:  3

→ Moved forward to: Twitter | Total visits: 2

===== Browser Menu =====
1. Visit Page
2. Back
3. Forward
4. Show History
5. Show Visit Frequency
6. Exit
Enter choice:  4

----- History Info -----
History:
Linkedin -> Github -> Twitter

No forward pages.
------------------------
```

```
===== Browser Menu =====
1. Visit Page
2. Back
3. Forward
4. Show History
5. Show Visit Frequency
6. Exit
Enter choice:  5

 Visit Frequency
Linkedin → 1 times
Github → 2 times
Twitter → 2 times

===== Browser Menu =====
1. Visit Page
2. Back
3. Forward
4. Show History
5. Show Visit Frequency
6. Exit
Enter choice:  6

 Exiting browser simulation. Goodbye!
```

Time Complexity Analysis:

• Visit Page: O(1) - Push operation is constant time

• Go Back: O(1) - Pop operation is constant time

• Current Page: O(1) - Access top element

• Display History: O(n) - Show all n pages

• Space Complexity: O(n) - Store up to n pages

Real-World Applications:

• Web browser back/forward navigation

• Undo functionality in text editors

• File system navigation in operating systems

• Function call stack in programming

Experiment 3: Stack Operations

Objective:

Demonstrate the Last-In-First-Out (LIFO) behavior of stacks through a comprehensive menu-driven program that implements all fundamental stack operations.

Learning Outcomes:

• Understand fundamental stack Abstract Data Type (ADT)

• Implement push, pop, peek, and display operations

• Build interactive programs with user input validation

• Learn practical applications of LIFO principle

• Understand stack usage in real-world problem solving

Theory:

A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle. Elements can only be inserted (pushed) or removed (popped) from the top of the stack. Peek operation allows viewing the top element without removal, and display shows all elements. Stacks are fundamental in computer science with applications in function call management, expression evaluation, syntax parsing, undo/redo functionality, and depth-first search algorithms.

Key Operations:

▶ push() - Insert element at the top of stack

▶ pop() - Remove and return element from top

▶ peek() - View top element without removing

▶ display() - Show all stack elements

▶ is_empty() - Check if stack is empty

Algorithm:

Push Operation:

1. Accept input element from user
2. Add element to top of stack
3. Display confirmation message

Pop Operation:

1. Check if stack is empty

2. If empty, display error message
3. If not empty, remove top element and display it

Peek Operation:

1. Check if stack is empty
2. If empty, display error message
3. If not empty, display top element without removing

Display Operation:

1. Check if stack is empty
2. If empty, display empty message
3. If not empty, show all elements in order

Code:

```
stack = []

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

def pop():
    if not stack:
        print("Stack is empty. Cannot pop.")
    else:
        popped = stack.pop()
        print(f"Popped {popped}")

def peek():
    if not stack:
        print("Stack is empty.")
    else:
        print(f"Top item is: {stack[-1]}")

def display():
    if not stack:
        print("Stack is empty.")
    else:
        print("Stack items (Top → Bottom):", stack[::-1])

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

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

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

Sample Output:

```
===== STACK MENU =====
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice:  1
Enter item to push:  21
Pushed 21

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

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

===== STACK MENU =====
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice:  1
Enter item to push:  34
Pushed 34
```

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

===== STACK MENU =====
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice:  4
Stack items (Top → Bottom): ['34', '21']

===== STACK MENU =====
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice:  5
Exiting program. Goodbye!
```

Time Complexity Analysis:

• Push Operation: O(1) - Constant time to add at top

• Pop Operation: O(1) - Constant time to remove from top

• Peek Operation: O(1) - Constant time to view top

• Display Operation: O(n) - Linear time to show n elements

• Space Complexity: O(n) - For storing n elements in stack

Real-World Applications:

• Function call stack in programming languages

• Expression evaluation (infix to postfix conversion)

• Undo/Redo functionality in applications

• Backtracking in maze solving and game algorithms

- Browser back/forward navigation
- Syntax parsing in compilers

Experiment 4 Ticketing System Using Queue

Objective:

Design and implement a queue-based customer service ticketing system that manages customer requests following the First-In-First-Out (FIFO) principle for fair and efficient service.

Learning Outcomes:

✓ Understand and apply the FIFO (First-In-First-Out) principle

✓ Implement enqueue and dequeue operations on a linear queue

✓ Handle boundary conditions and edge cases

✓ Apply queue data structure to real-world service scenarios

✓ Implement queue status checking and display functions

Theory:

A queue is a linear data structure that implements the FIFO principle, where elements enter at the rear (enqueue) and exit from the front (dequeue), maintaining insertion order. Queues are fundamental in computer systems and real-world applications including customer service lines, print job queues, message processing systems, and CPU scheduling. This ticketing system demonstrates how queues manage customer requests fairly by processing them in the order they arrive, ensuring no customer is skipped.

Key Operations:

▶ enqueue(request_id) - Add new ticket request to queue rear

▶ dequeue() - Process and remove the next ticket from queue front

▶ is_empty() - Check if queue contains any pending tickets

▶ is_full() - Check if queue has reached maximum capacity

▶ size() - Return current number of tickets in queue

▶ display() - Show all waiting tickets in order

Algorithm:

1. Create fixed-size array and pointers (front, rear)

2. Initialize front = -1, rear = -1 (empty state)

3. Enqueue: Increment rear, set front=0 if first element, add ticket

4. Dequeue: Get ticket at front, increment front pointer

5. Reset if all tickets processed (front > rear)

Code:

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

    def enqueue(self, passenger_name):
        if self.is_full():
            print("Queue is full! Cannot add new passenger.")
            return
        self.rear += 1
        if self.front == -1:
            self.front = 0
        self.queue[self.rear] = passenger_name
        print(f"Passenger '{passenger_name}' added to the queue.")

    def dequeue(self):
        if self.is_empty():
            print("Queue is empty! No passenger to serve.")
            return
        name = self.queue[self.front]
        print(f"Serving and removing passenger '{name}'...")
        self.front += 1

        if self.front > self.rear:
            self.front = self.rear = -1

    def is_empty(self):
        return self.front == -1

    def is_full(self):
        return self.rear == self.max_size - 1

    def size(self):
        if self.is_empty():
            return 0
        return self.rear - self.front + 1
```

```
    def display(self):
        if self.is_empty():
            print("Queue is empty.")
        else:
            print("Current Passengers in Queue:")
            for i in range(self.front, self.rear + 1):
                print(self.queue[i], end=" ")
            print()


if __name__ == "__main__":
    q = TicketQueue(5)

    q.enqueue("Riya")
    q.enqueue("Arjun")
    q.enqueue("Sneha")
    q.display()

    q.dequeue()
    q.display()

    q.enqueue("Karan")
    q.enqueue("Amit")
    q.enqueue("Diya")
    q.display()

    print("Current Queue Size:", q.size())
    print("Is Queue Full?", q.is_full())
```

Sample Output:

```
Passenger 'Riya' added to the queue.
Passenger 'Arjun' added to the queue.
Passenger 'Sneha' added to the queue.
Current Passengers in Queue:
Riya Arjun Sneha
Serving and removing passenger 'Riya'...
Current Passengers in Queue:
Arjun Sneha
Passenger 'Karan' added to the queue.
Passenger 'Amit' added to the queue.
Queue is full! Cannot add new passenger.
Current Passengers in Queue:
Arjun Sneha Karan Amit
Current Queue Size: 4
Is Queue Full? True
```

Time Complexity Analysis:

• Enqueue: O(1) - Constant time to add at rear

• Dequeue: O(1) - Constant time to remove from front

• Display: O(n) - Linear time to show all n tickets

• Space Complexity: O(m) - For maximum queue size m

Experiment 5:Singly Linked List Operations

Objective:

Implement a complete singly linked list with all essential operations including insertion at multiple positions, deletion from various locations, searching, and traversal.

Learning Outcomes:

✓ Understand node-based dynamic data structures and pointer concepts

✓ Perform insertion operations at beginning, end, and specific positions

✓ Remove nodes safely handling all edge cases

✓ Traverse and search through linked list structures

✓ Contrast linked lists with array-based storage

Theory:

A singly linked list is a dynamic data structure composed of nodes, where each node contains data and a reference (pointer) to the next node. Unlike arrays, linked lists do not require contiguous memory allocation and support flexible dynamic sizing. Each node maintains a reference to its successor, creating a chain structure. The first node is called the head, and the last node points to None. This structure is fundamental for applications requiring dynamic memory management and frequent insertions/deletions.

Key Operations:

▶ insert_at_beginning(data) - Add new node at the start

▶ insert_at_end(data) - Add new node at the end

▶ delete_at_beginning() - Remove first node

▶ delete_at_end() - Remove last node

▶ delete_by_value(key) - Remove specific node by value

▶ search(key) - Find and return position of element

▶ display() - Print all nodes in order

Code:

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


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

    def insert_at_beginning(self, data):
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node
        print(f"Inserted {data} at beginning")

    def insert_at_end(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            print(f"Inserted {data} at end (first node)")
            return
        temp = self.head
        while temp.next:
            temp = temp.next
        temp.next = new_node
        print(f"Inserted {data} at end")

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

```
def search(self, key):
    temp = self.head
    position = 1
    while temp:
        if temp.data == key:
            print(f"{key} found at position {position}")
            return True
        temp = temp.next
        position += 1
    print(f"{key} not found in list")
    return False

def delete_at_beginning(self):
    if self.head is None:
        print("List is empty. Cannot delete.")
        return
    removed = self.head.data
    self.head = self.head.next
    print(f"Deleted {removed} from beginning")

def delete_at_end(self):
    if self.head is None:
        print("List is empty. Cannot delete.")
        return
    if self.head.next is None:
        removed = self.head.data
        self.head = None
        print(f"Deleted {removed} from end (list is now empty)")
        return
    temp = self.head
    while temp.next.next:
        temp = temp.next
    removed = temp.next.data
    temp.next = None
    print(f"Deleted {removed} from end")
```

```
    def delete_by_value(self, key):
        if self.head is None:
            print("List is empty")
            return

        if self.head.data == key:
            removed = self.head.data
            self.head = self.head.next
            print(f"Deleted {removed} from list (was at head)")
            return

        prev = None
        temp = self.head
        while temp and temp.data != key:
            prev = temp
            temp = temp.next

        if temp is None:
            print(f"{key} not found, cannot delete")
        else:
            prev.next = temp.next
            print(f"Deleted {key} from list")


# Main Program (Example Changed)
if __name__ == "__main__":
    ll = LinkedList()
    ll.insert_at_beginning(50)
    ll.insert_at_beginning(60)
    ll.insert_at_end(70)
    ll.insert_at_end(80)

    print("\nInitial list:")
    ll.display()

    print("\nSearch tests:")
    ll.search(70)
    ll.search(200)
```

```
print("\nDelete operations:")
ll.delete_at_beginning()
ll.display()

ll.delete_at_end()
ll.display()

ll.delete_by_value(70)
ll.display()

ll.delete_by_value(999)

ll.delete_at_end()
ll.display()
```

Sample Output:

```
Inserted 50 at beginning
Inserted 60 at beginning
Inserted 70 at end
Inserted 80 at end

Initial list:
Linked List: 60 -> 50 -> 70 -> 80 -> None

Search tests:
70 found at position 3
200 not found in list

Delete operations:
Deleted 60 from beginning
Linked List: 50 -> 70 -> 80 -> None
Deleted 80 from end
Linked List: 50 -> 70 -> None
Deleted 70 from list
Linked List: 50 -> None
999 not found, cannot delete
Deleted 50 from end (list is now empty)
List is empty
```

Time Complexity Analysis:

• Insert at beginning: O(1) - Direct head update

• Insert at end: O(n) - Must traverse to last node

• Delete at beginning: O(1) - Update head pointer

• Delete at end: O(n) - Must find second-last node

• Search: O(n) - Linear search through nodes

• Space Complexity: O(n) - Store n nodes

Experiment 6: Circular Singly Linked List Operations

Objective:

Implement a circular singly linked list where the last node points back to the first node, supporting all standard operations with circular traversal logic.

Learning Outcomes:

✓ Distinguish circular structure from standard linked lists

✓ Implement insertion and deletion maintaining circular property

✓ Handle circular traversal and boundary conditions

✓ Apply circular lists to round-robin and continuous processing scenarios

Theory:

A circular singly linked list maintains a circular structure where the last node contains a reference to the first node instead of NULL. This creates a closed loop enabling continuous iteration without detecting an end. Circular lists are particularly useful for round-robin scheduling, carousel-like data structures, and applications requiring repeated cycling through elements.

Key Operations:

▶ insert_at_beginning(data) - Add at start, maintain circle

▶ insert_at_end(data) - Add at end, maintain circle

▶ delete_at_beginning() - Remove from start

▶ delete_at_end() - Remove from end

▶ delete_by_value(key) - Delete specific node

▶ search(key) - Find element in circular list

▶ display_circular() - Print all nodes showing circle

Code:

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


# Circular Singly Linked List class
class CircularLinkedList:
    def __init__(self):
        self.head = None

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

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

```
def search(self, key):
    if self.head is None:
        print("List is empty")
        return False
    temp = self.head
    position = 1
    while True:
        if temp.data == key:
            print(f"Found {key} at position {position}")
            return True
        temp = temp.next
        position += 1
        if temp == self.head:
            break
    print(f"{key} Not Found")
    return False

def delete_at_beginning(self):
    if self.head is None:
        print("List is empty. Underflow.")
        return

    temp = self.head

    if temp.next == self.head:
        print(f"Deleted {temp.data} (only node present)")
        self.head = None
        return

    while temp.next != self.head:
        temp = temp.next
    removed = self.head.data
    temp.next = self.head.next
    self.head = self.head.next
    print(f"Deleted {removed} from beginning")
```

```
def delete_at_end(self):
    if self.head is None:
        print("List is empty. Underflow.")
        return

    temp = self.head
    if temp.next == self.head:
        print(f"Deleted {temp.data} (only node present)")
        self.head = None
        return

    prev = None
    while temp.next != self.head:
        prev = temp
        temp = temp.next
    removed = temp.data
    prev.next = self.head
    print(f"Deleted {removed} from end")

def delete_by_value(self, key):
    if self.head is None:
        print("List is empty. Underflow.")
        return

    temp = self.head
    if temp.data == key:
        self.delete_at_beginning()
        return

    prev = None
    while temp.next != self.head and temp.data != key:
        prev = temp
        temp = temp.next
```

```
        if temp.data == key:
            prev.next = temp.next
            print(f"Deleted {key} from list")
        else:
            print(f"Node with value {key} not found")

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


# Main Program (Values Changed)
if __name__ == "__main__":
    cll = CircularLinkedList()

    cll.insert_at_beginning(50)
    cll.insert_at_beginning(60)
    cll.insert_at_end(70)
    cll.insert_at_end(80)

    print("Initial Circular Linked List:")
    cll.display()

    print("\nSearch Tests:")
    cll.search(70)
    cll.search(200)
```

```python
print("\nDelete at Beginning:")
cll.delete_at_beginning()
cll.display()

print("\nDelete at End:")
cll.delete_at_end()
cll.display()

print("\nDelete by Value (70):")
cll.delete_by_value(70)
cll.display()

print("\nDelete by Value (999 - Not Found):")
cll.delete_by_value(999)
cll.display()
```

Sample Output:

```
Initial Circular Linked List:
60 -> 50 -> 70 -> 80 -> (back to head)

Search Tests:
Found 70 at position 3
200 Not Found

Delete at Beginning:
Deleted 60 from beginning
50 -> 70 -> 80 -> (back to head)

Delete at End:
Deleted 80 from end
50 -> 70 -> (back to head)

Delete by Value (70):
Deleted 70 from list
50 -> (back to head)

Delete by Value (999 - Not Found):
Node with value 999 not found
50 -> (back to head)
```

Time Complexity Analysis:

• Insert at beginning: O(n) - Must find last node to maintain circle

• Insert at end: O(n) - Traverse to last node

• Delete at beginning: O(n) - Find last node first

• Delete at end: O(n) - Must traverse to second-last

• Search: O(n) - Check each node in circle

• Space Complexity: O(n) - For n nodes

Experiment 7: Reverse String Using Stack

Objective:

Utilize stack data structure to reverse a given string by leveraging the LIFO (Last In First Out) principle for elegant and efficient reversal.

Learning Outcomes:

✓ Effectively utilize stack LIFO behavior for string manipulation

✓ Understand practical applications of stack in string processing

✓ Analyze algorithm efficiency and compare with alternative approaches

Theory:

By pushing each character of a string onto a stack and then popping all characters, the order automatically reverses due to the LIFO behavior. This is an elegant method for string reversal compared to array indexing or string slicing. The stack-based approach demonstrates the practical utility of data structures in solving common programming problems.

Algorithm:

1. Create empty stack

2. For each character in string, push onto stack

3. While stack not empty, pop character and append to result

4. Return reversed string

Code:

```
def reverse_string(s):
    stack = []

    # Push characters to stack
    for char in s:
        stack.append(char)

    # Pop characters to form reversed string
    reversed_str = ""
    while stack:
        reversed_str += stack.pop()

    return reversed_str


result = reverse_string("Structure")
print("Reversed string:", result)
```

Sample Output:

```
Reversed string: erutcurtS
```

Time & Space Complexity:

• Time Complexity: O(n) - Process each character exactly once

• Space Complexity: O(n) - Store all n characters in stack

Experiment 8: Balanced Parentheses Using Stack

Objective:

Validate whether parentheses and brackets in an expression are properly balanced using a stack data structure.

Learning Outcomes:

✓ Apply stack data structure to expression validation problems

✓ Handle multiple bracket types and nesting levels

✓ Implement character-level parsing and matching logic

Theory:

An expression has balanced parentheses if each opening bracket has a corresponding closing bracket, brackets are properly nested, and no closing bracket appears before its matching opening bracket. A stack elegantly solves this by pushing opening brackets and popping them when matching closing brackets are found. This algorithm is used in compilers, interpreters, and code editors to validate syntax.

Algorithm:

1. Initialize empty stack and bracket mapping

2. For each character in expression:

   a. If opening bracket, push to stack

   b. If closing bracket, check stack top matches

   c. If mismatch or stack empty, return false

3. At end, return true if stack is empty

Code:

```python
def is_balanced(expression):
    stack = []
    opening = "({["
    closing = ")}]"
    matching = {')': '(', '}': '{', ']': '['}

    for char in expression:
        if char in opening:
            stack.append(char)
        elif char in closing:
            if not stack or stack[-1] != matching[char]:
                return False
            stack.pop()

    return len(stack) == 0


# Example test cases
print(is_balanced("{[()]}"))
print(is_balanced("{[(])}"))
print(is_balanced("(((()))"))
print(is_balanced("((()"))
```

Sample Output:

```
True
False
True
False
```

Time & Space Complexity:

• Time Complexity: O(n) - Scan expression once

• Space Complexity: O(n) - Worst case all opening brackets in stack

Experiment 9: Linear Search

Objective:

Implement a straightforward searching algorithm that sequentially checks each element in an array until the target value is located or the entire array has been examined. Linear search is particularly useful for small datasets or when the array is not sorted.

Learning Outcomes:

• Understand sequential search mechanism and implementation

• Analyze best, average, and worst-case performance scenarios

• Learn to apply linear search in practical applications

• Develop debugging and testing skills through step-by-step search

• Compare linear search with other search algorithms

Theory:

Linear search is the simplest search algorithm that checks every element in a list sequentially from start to end until the target element is found or the list ends. It requires no preprocessing or sorting of data, making it ideal for unsorted or small datasets. Although it has O(n) time complexity, its simplicity and lack of sorting requirement make it practical for many real-world applications.

Key Operations:

▶ linear_search(arr, target): Searches array for target element

▶ Returns index if found, -1 if not found

▶ Suitable for small datasets

▶ Works on both sorted and unsorted data

Algorithm:

1. Start from the first element (index 0) of the array

2. Compare current element with the target value

3. If they match, return the current index immediately

4. If no match, move to the next element

5. Repeat steps 2-4 for all elements

6. If the entire array is checked without finding target, return -1

Code:

```python
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i   # return index
    return -1  # not found


# New Example
arr = [5, 12, 7, 25, 18, 30, 42]
target = 18

result = linear_search(arr, target)
print("Element found at index:", result)
```

Sample Output:

```
Element found at index: 4
```

Time Complexity Analysis:

• Best Case: O(1) - Target element is at the first position

• Average Case: O(n) - Target element is somewhere in the middle

• Worst Case: O(n) - Target element is at the end or not present

• Space Complexity: O(1) - No extra memory space required

Practical Applications:

• Searching unsorted lists in small datasets

• Finding elements in linked lists

• Simple database searches

• Educational purposes to understand search concepts

Experiment 10: Binary Search

Objective:

Develop an efficient searching algorithm that dramatically reduces search time by repeatedly dividing the search interval in half. Binary search is essential for working with large sorted datasets where linear search becomes impractical.

Learning Outcomes:

• Master the divide-and-conquer search strategy

• Understand significant performance advantages over linear search

• Analyze logarithmic time complexity implications

• Learn importance of sorted data preprocessing

• Apply binary search to real-world problems

Theory:

Binary search is a highly efficient algorithm that works on sorted arrays. It compares the target value to the middle element of the array. If they match, the search is complete. If the target is larger, the algorithm searches only the upper half; if smaller, it searches the lower half. This halving process continues until the element is found or the search space is exhausted. With each comparison, the search space is reduced by half, resulting in O(log n) complexity.

Key Operations:

▶ binary_search(arr, target): Searches sorted array for target

▶ Uses divide-and-conquer approach

▶ Returns index if found, -1 if not found

▶ Requires array to be sorted beforehand

▶ Much faster than linear search for large datasets

Algorithm:

1. Initialize low pointer to array start (index 0)

2. Initialize high pointer to array end (index n-1)

3. While low <= high:

   a. Calculate mid = (low + high) // 2

   b. Compare array[mid] with target

c. If equal, return mid

d. If array[mid] < target, set low = mid + 1 (search upper half)

e. If array[mid] > target, set high = mid - 1 (search lower half)

4. If loop exits, target not found, return -1

Code:

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

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

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

    return -1  # element not found


# New Example
arr = [5, 12, 19, 27, 33, 45, 52, 60]
target = 33

result = binary_search(arr, target)
print("Element found at index:", result)
```

Sample Output:

```
Element found at index: 4
```

Time Complexity Analysis:

• Best Case: O(1) - Target element is at the middle on first comparison

• Average Case: O(log n) - Multiple halvings needed to locate element

• Worst Case: O(log n) - Element at end or not present

• Space Complexity: O(1) - Iterative implementation uses constant space

Binary vs Linear Search Comparison:

Parameter	Linear Search	Binary Search
Time Complexity	O(n)	O(log n)
Sorted Array Required	No	Yes
Best for	Small/Unsorted Data	Large Sorted Data
Implementation	Simple	More Complex

Practical Applications:

• Searching in large sorted databases

• Finding elements in sorted arrays

• Dictionary lookups in sorted word lists

• Range queries in data structures