```
#lab assignment 1
#In an e-commerce system, customer account IDs are stored in a list, and you are tasked with writing a program that implements the following:
# • Linear Search: Check if a particular customer account ID exists in the list.
# • Binary Search: Implement Binary search to find if a customer account ID exists,
# improving the search efficiency over the basic linear.


#-----------------------------------------------------------------------------------------------



# Function for Linear Search
def linear_search(customer_ids, target_id):
    for i in range(len(customer_ids)):
        if customer_ids[i] == target_id:
            return i
    return -1



# Function for Binary Search
def binary_search(customer_ids, target_id):
    low = 0
    high = len(customer_ids) - 1

    while low <= high:
        mid = (low + high) // 2
        if customer_ids[mid] == target_id:
            return mid
        elif customer_ids[mid] < target_id:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

```
# Main Program
# Take list of customer IDs from user
n = int(input("Enter number of customer account IDs: "))
customer_ids = []

for i in range(n):
    cid = int(input(f"Enter Customer ID {i+1}: "))
    customer_ids.append(cid)

print("\nCustomer Account IDs entered:", customer_ids)

# Take target ID from user
target_id = int(input("\nEnter Customer ID to search: "))

# Perform Linear Search
result_linear = linear_search(customer_ids, target_id)
if result_linear != -1:
    print(f"[Linear Search] Customer ID {target_id} found at index
{result_linear}.")
else:
    print(f"[Linear Search] Customer ID {target_id} not found.")

# Perform Binary Search (requires sorted list)
customer_ids.sort()
result_binary = binary_search(customer_ids, target_id)
if result_binary != -1:
    print(f"[Binary Search] Customer ID {target_id} found at sorted index
{result_binary}.")
else:
    print(f"[Binary Search] Customer ID {target_id} not found.")
```

```
#lab assignment 2
# In a company, employee salaries are stored in a list as floating-point numbers. Write a
# Python program that sorts the employee salaries in ascending order using the following
# two algorithms:
# • Selection Sort: Sort the salaries using the selection sort algorithm.
# • Bubble Sort: Sort the salaries using the bubble sort algorithm.
# After sorting the salaries, the program should display top five highest salaries in the
# company.


#-----------------------------------------------------------------------------------------


# Function for Selection Sort
def selection_sort(salaries):
    n = len(salaries)
    for i in range(n):
        min_index = i
        for j in range(i + 1, n):
            if salaries[j] < salaries[min_index]:
                min_index = j
        salaries[i], salaries[min_index] = salaries[min_index], salaries[i]
    return salaries


# Function for Bubble Sort
def bubble_sort(salaries):
    n = len(salaries)
    for i in range(n):
        for j in range(0, n - i - 1):
            if salaries[j] > salaries[j + 1]:
                salaries[j], salaries[j + 1] = salaries[j + 1], salaries[j]
    return salaries
```

```
# Main Program
n = int(input("Enter number of employees: "))
salaries = []

for i in range(n):
    salary = float(input(f"Enter salary of employee {i + 1}: "))
    salaries.append(salary)

print("\nOriginal Salaries:", salaries)

# Selection Sort
sel_sorted = salaries.copy()
selection_sort(sel_sorted)
print("\nSalaries after Selection Sort (Ascending):", sel_sorted)

# Bubble Sort
bub_sorted = salaries.copy()
bubble_sort(bub_sorted)
print("Salaries after Bubble Sort (Ascending):", bub_sorted)

# Display Top 5 Highest Salaries
# We can take from the end of sorted list
top_five = bub_sorted[-5:] if len(bub_sorted) >= 5 else bub_sorted
top_five.reverse()  # Show in descending order
print("\nTop 5 Highest Salaries:", top_five)
```

```
#lab assignment 3
# Implementing a real-time undo/redo system for a text editing application using a Stack
# data structure. The system should support the following operations:
# • Make a Change: A new change to the document is made.
# • Undo Action: Revert the most recent change and store it for potential redo.
# • Redo Action: Reapply the most recently undone action.
# • Display Document State: Show the current state of the document after undoing or
# redoing an action.


#-----------------------------------------------------------------------------------------


undo_stack = []   # To store performed changes
redo_stack = []   # To store undone changes

document = ""     # Initial empty document


# Function to make a change
def make_change(change):
    global document
    undo_stack.append(document)  # Save current state before making change
    document += change           # Apply new change
    redo_stack.clear()           # Clear redo history (since new change is made)
    print(f"Change applied: '{change}'")


# Function to undo last change
def undo():
    global document
```

```
    if not undo_stack:
        print("Nothing to undo.")
    else:
        redo_stack.append(document)    # Save current state for redo
        document = undo_stack.pop()    # Revert to previous state
        print("Undo performed.")


# Function to redo last undone change
def redo():
    global document
    if not redo_stack:
        print("Nothing to redo.")
    else:
        undo_stack.append(document)    # Save current state for undo
        document = redo_stack.pop()    # Reapply last undone state
        print("Redo performed.")


# Function to display current document
def display_document():
    print(f"\nCurrent Document State: '{document}'\n")


# Main Menu
while True:
    print("\n--- Text Editor Menu ---")
    print("1. Make a Change")
    print("2. Undo Last Action")
    print("3. Redo Last Action")
    print("4. Display Document")
    print("5. Exit")

    choice = input("Enter your choice: ")

    if choice == '1':
```

```
    change = input("Enter text to add: ")
    make_change(change)
    display_document()

elif choice == '2':
    undo()
    display_document()

elif choice == '3':
    redo()
    display_document()

elif choice == '4':
    display_document()

elif choice == '5':
    print("Exiting... Goodbye!")
    break

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

```
#lab assignment 4
# Implement a real-time event processing system using a Queue data structure. The
# system should support the following features:
# • Add an Event: When a new event occurs, it should be added to the event queue.
# • Process the Next Event: The system should process and remove the event that has
# been in the queue the longest.
# • Display Pending Events: Show all the events currently waiting to be processed.
# • Cancel an Event: An event can be canceled if it has not been processed.


#-----------------------------------------------------------------------------------------



event_queue = []


# Function to add a new event
def add_event(event_name):
    event_queue.append(event_name)  # Enqueue operation
    print(f"Event '{event_name}' added to the queue.")


# Function to process (remove) the next event
def process_event():
    if len(event_queue) == 0:
        print("No events to process.")
    else:
        event = event_queue.pop(0)  # Dequeue operation (FIFO)
        print(f"Processing event: '{event}'")
```

```
# Function to display pending events
def display_events():
    if len(event_queue) == 0:
        print("No pending events.")
    else:
        print("\nPending Events in Queue:")
        for i in range(len(event_queue)):
            print(f"{i + 1}. {event_queue[i]}")
        print()


# Function to cancel an event (if not processed)
def cancel_event(event_name):
    if event_name in event_queue:
        event_queue.remove(event_name)
        print(f"Event '{event_name}' has been canceled.")
    else:
        print(f"No such pending event: '{event_name}'")


# Main Menu Loop
while True:
    print("\n--- Real-Time Event Processing System ---")
    print("1. Add an Event")
    print("2. Process Next Event")
    print("3. Display Pending Events")
    print("4. Cancel an Event")
    print("5. Exit")

    choice = input("Enter your choice: ")

    if choice == '1':
        event_name = input("Enter event name: ")
        add_event(event_name)
```

```
elif choice == '2':
    process_event()

elif choice == '3':
    display_events()

elif choice == '4':
    event_name = input("Enter event name to cancel: ")
    cancel_event(event_name)

elif choice == '5':
    print("Exiting the system... Goodbye!")
    break

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

```
#lab assignment 5
# Create a Student Record Management System using linked list
# • Use a singly/doubly linked list to store student data (Roll No, Name, Marks).
# • Perform operations: Add, Delete, Update, Search, and Sort.
# • Display records in ascending/descending order based on marks or roll number.


#-----------------------------------------------------------------------------------------------



class Node:
    def __init__(self, roll, name, marks):
        self.roll = roll
        self.name = name
        self.marks = marks
        self.next = None


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

    # Add a new student at the end
    def add_student(self, roll, name, marks):
        new_node = Node(roll, name, marks)
        if self.head is None:
            self.head = new_node
        else:
            temp = self.head
            while temp.next:
                temp = temp.next
            temp.next = new_node
        print("Student record added successfully.")
```

```
# Delete a student by roll number
def delete_student(self, roll):
    temp = self.head

    if temp is None:
        print("No records found.")
        return

    if temp.roll == roll:
        self.head = temp.next
        print(f"Student with Roll No {roll} deleted.")
        return

    prev = None
    while temp and temp.roll != roll:
        prev = temp
        temp = temp.next

    if temp is None:
        print("Record not found.")
        return

    prev.next = temp.next
    print(f"Student with Roll No {roll} deleted.")

# Update student record
def update_student(self, roll, new_name, new_marks):
    temp = self.head
    while temp:
        if temp.roll == roll:
            temp.name = new_name
            temp.marks = new_marks
            print("Record updated successfully.")
            return
        temp = temp.next
```

```
        print("Record not found.")

    # Search student by roll number
    def search_student(self, roll):
        temp = self.head
        while temp:
            if temp.roll == roll:
                print(f"\nRecord Found:")
                print(f"Roll No: {temp.roll}, Name: {temp.name}, Marks:
{temp.marks}")
                return
            temp = temp.next
        print("Record not found.")

    # Display all student records
    def display_students(self):
        if self.head is None:
            print("No student records to display.")
            return
        temp = self.head
        print("\nStudent Records:")
        print("-" * 35)
        print("Roll No\tName\t\tMarks")
        print("-" * 35)
        while temp:
            print(f"{temp.roll}\t{temp.name}\t\t{temp.marks}")
            temp = temp.next
        print("-" * 35)

    # Sort records (by marks or roll number)
    def sort_records(self, key="roll", order="asc"):
        if self.head is None or self.head.next is None:
            print("Not enough records to sort.")
            return

        swapped = True
```

```
        while swapped:
            swapped = False
            temp = self.head
            while temp.next:
                if key == "roll":
                    condition = temp.roll > temp.next.roll
                elif key == "marks":
                    condition = temp.marks > temp.next.marks
                else:
                    print("Invalid sort key.")
                    return

                # Adjust for descending order
                if order == "desc":
                    condition = not condition

                if condition:
                    temp.roll, temp.next.roll = temp.next.roll, temp.roll
                    temp.name, temp.next.name = temp.next.name, temp.name
                    temp.marks, temp.next.marks = temp.next.marks, temp.marks
                    swapped = True
                temp = temp.next
        print(f"Records sorted by {key} in {order} order successfully.")


student_list = StudentList()

while True:
    print("\n--- Student Record Management System ---")
    print("1. Add Student")
    print("2. Delete Student")
    print("3. Update Student")
    print("4. Search Student")
    print("5. Display All Students")
    print("6. Sort Records")
    print("7. Exit")
```

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

if choice == '1':
    roll = int(input("Enter Roll No: "))
    name = input("Enter Name: ")
    marks = float(input("Enter Marks: "))
    student_list.add_student(roll, name, marks)

elif choice == '2':
    roll = int(input("Enter Roll No to delete: "))
    student_list.delete_student(roll)

elif choice == '3':
    roll = int(input("Enter Roll No to update: "))
    new_name = input("Enter New Name: ")
    new_marks = float(input("Enter New Marks: "))
    student_list.update_student(roll, new_name, new_marks)

elif choice == '4':
    roll = int(input("Enter Roll No to search: "))
    student_list.search_student(roll)

elif choice == '5':
    student_list.display_students()

elif choice == '6':
    key = input("Sort by (roll/marks): ").lower()
    order = input("Order (asc/desc): ").lower()
    student_list.sort_records(key, order)

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

else:
```

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

```
#lab assignment 6
# Implement a hash table of size 10 and use the division method as a
hash function. In
# case of a collision, use chaining. Implement the following operations:
# • Insert(key): Insert key-value pairs into the hash table.
# • Search(key): Search for the value associated with a given key.
# • Delete(key): Delete a key-value pair from the hash table



#-----------------------------------------------------------------------------------------

class HashTable:
    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(self.size)]  # Each bucket is a list (for
chaining)

    def _hash_function(self, key):
        return key % self.size

    def insert(self, key, value):
        index = self._hash_function(key)
        # Check if key exists, update value
        for pair in self.table[index]:
            if pair[0] == key:
                pair[1] = value
                print(f"Updated key {key} with value '{value}' at index {index}")
                return
        # Insert new key-value pair
        self.table[index].append([key, value])
        print(f"Inserted key {key} with value '{value}' at index {index}")

    def search(self, key):
        index = self._hash_function(key)
        for pair in self.table[index]:
            if pair[0] == key:
```

```
                print(f"Found key {key} at index {index} with value '{pair[1]}'")
                return pair[1]
        print(f"Key {key} not found")
        return None

    def delete(self, key):
        index = self._hash_function(key)
        for i, pair in enumerate(self.table[index]):
            if pair[0] == key:
                self.table[index].pop(i)
                print(f"Deleted key {key} from index {index}")
                return
        print(f"Key {key} not found for deletion")

    def display(self):
        print("\nHash Table:")
        for i, bucket in enumerate(self.table):
            print(f"Index {i}: {bucket}")


# Example usage
if __name__ == "__main__":
    ht = HashTable()

    # Insert keys
    ht.insert(10, "Apple")
    ht.insert(20, "Banana")
    ht.insert(30, "Mango")
    ht.insert(25, "Peach")   # Will cause a collision (25 % 10 = 5)
    ht.insert(5, "Orange")   # Same index as 25

    ht.display()

    # Search keys
    ht.search(20)
    ht.search(100)
```

```
# Delete keys
ht.delete(30)
ht.delete(100)

ht.display()
```

```
# #lab assignment 7
# Design and implement a hash table of fixed size. Use the division
method for the hash
# function and resolve collisions using linear probing. Allow the user to
perform the
# following operations: • Insert a key
# • Search for a key
# • Delete a key
# • Display the table.


#-----------------------------------------------------------------------------------------


class HashTable:
    def __init__(self, size):
        self.size = size
        self.table = [None] * size  # Initialize the table with None values

    def hash_function(self, key):
        return key % self.size  # Division method

    def insert(self, key):
        index = self.hash_function(key)
        # Linear probing to resolve collisions
        while self.table[index] is not None:
            if self.table[index] == key:  # Avoid duplicate insertions
                print(f"{key} already exists in the hash table.")
                return
            index = (index + 1) % self.size  # Move to next slot

        self.table[index] = key
        print(f"Inserted {key} at index {index}.")

    def search(self, key):
        index = self.hash_function(key)
```

```
        start_index = index

        while self.table[index] is not None:
            if self.table[index] == key:
                print(f"Found {key} at index {index}.")
                return index
            index = (index + 1) % self.size
            if index == start_index:  # Full cycle completed
                break

        print(f"{key} not found in the hash table.")
        return -1

    def delete(self, key):
        index = self.search(key)
        if index != -1:
            self.table[index] = None
            print(f"Deleted {key} from index {index}.")
        else:
            print(f"{key} does not exist in the hash table.")

    def display(self):
        print("\nHash Table:")
        for i, value in enumerate(self.table):
            print(f"Index {i}: {value}")


# Example usage
hash_table = HashTable(10)  # Create hash table of size 10
hash_table.insert(15)
hash_table.insert(25)
hash_table.insert(35)
hash_table.display()
hash_table.search(25)
hash_table.delete(25)
hash_table.display()
```

```
# lab assignment 8
# Consider a particular area in your city. Note the popular locations A, B, C . . . in that
# area. Assume these locations represent nodes of a graph. If there is a route between two
# locations, it is represented as connections between nodes.
# Find out the sequence in which you will visit these locations, starting from (say A)
# using (i) BFS and (ii) DFS.
# Represent a given graph using an adjacency matrix to perform DFS and an adjacency
# list to perform BFS.


#-------------------------------------------------------------------------------------------


from collections import deque

# Mapping location names to indices for matrix representation
locations = ['A', 'B', 'C', 'D']
location_index = {name: idx for idx, name in enumerate(locations)}

# DFS using Adjacency Matrix
# Graph represented as adjacency matrix
adj_matrix = [
    # A  B  C  D
    [0, 1, 1, 0],  # A
    [1, 0, 0, 1],  # B
    [1, 0, 0, 1],  # C
    [0, 1, 1, 0]   # D
]

def dfs_matrix(start):
    visited = [False] * len(locations)
    result = []
```

```
    def dfs(vertex):
        visited[vertex] = True
        result.append(locations[vertex])
        for neighbor in range(len(adj_matrix)):
            if adj_matrix[vertex][neighbor] == 1 and not visited[neighbor]:
                dfs(neighbor)

    dfs(location_index[start])
    return result



# BFS using Adjacency List
# Graph represented as adjacency list
adj_list = {
    'A': ['B', 'C'],
    'B': ['A', 'D'],
    'C': ['A', 'D'],
    'D': ['B', 'C']
}

def bfs_list(start):
    queue = deque()
    visited = set()
    result = []

    queue.append(start)
    visited.add(start)

    while queue:
        vertex = queue.popleft()
        result.append(vertex)
        for neighbor in adj_list[vertex]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
```

```
    return result


# Run the algorithms
while True:
    print("\n===== Menu =====")
    print("1. DFS")
    print("2. BFS")
    print("3. Exit")
    choice = input("Enter your choice: ")

    if choice == '1':
        start_location = 'A'
        dfs_result = dfs_matrix(start_location)
        print("DFS Traversal (using adjacency matrix):", dfs_result)

    elif choice == '2':
        start_location = 'A'
        bfs_result = bfs_list(start_location)
        print("BFS Traversal (using adjacency list):", bfs_result)

    elif choice == '3':
        break

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

```
#lab assignment 9
# Implement various operations on a Binary Search Tree, such as
insertion, deletion,
# display, and search.


#-----------------------------------------------------------------------------------------



class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None


class BST:
    def __init__(self):
        self.root = None

    def insert(self, key):
        newnode = Node(key)
        if self.root is None:
            self.root = newnode
        else:
            self._insert(self.root, key)

    def _insert(self, node, key):
        newnode = Node(key)
        if key < node.key:
            if node.left is None:
                node.left = newnode
            else:
                self._insert(node.left, key)
        else:
```

```
        if node.right is None:
            node.right = newnode
        else:
            self._insert(node.right, key)

    def search(self, key):
        return self._search(self.root, key)

    def _search(self, node, key):
        if node is None:
            return False
        if key == node.key:
            return True
        if key < node.key:
            return self._search(node.left, key)
        return self._search(node.right, key)

    def display(self):
        print("In-order display:")
        self._inorder(self.root)
        print()

    def _inorder(self, root):
        if root is not None:
            self._inorder(root.left)
            print(root.key, end=" ")
            self._inorder(root.right)

    def delete(self, key):
        self.root = self._delete(self.root, key)

    def _delete(self, node, key):
        if node is None:
            return None

        if key < node.key:
```

```
            node.left = self._delete(node.left, key)
        elif key > node.key:
            node.right = self._delete(node.right, key)
        else:
            # Case 1: No child
            if node.left is None and node.right is None:
                return None
            # Case 2: One child
            if node.left is None:
                return node.right
            if node.right is None:
                return node.left
            # Case 3: Two children
            min_larger_node = self._min_value_node(node.right)
            node.key = min_larger_node.key
            node.right = self._delete(node.right, min_larger_node.key)
        return node

    def _min_value_node(self, node):
        """Find the node with the minimum key."""
        current = node
        while current.left is not None:
            current = current.left
        return current


# Example Usage
bst = BST()
print(bst.search(40))
bst.insert(50)
bst.insert(30)
bst.insert(70)
bst.insert(20)
bst.insert(40)
bst.insert(60)
bst.insert(80)
```

```
bst.display()  # Display BST
print("Searching for 40:", bst.search(40))   # Search operation
print("Searching for 100:", bst.search(100)) # Search operation
bst.delete(70)  # Delete operation
bst.display()   # Display after deletion
```