Experiment 1: Library Borrowing Records Management

Program

```
from collections import Counter

borrow_records = {
    'Book1': 5,
    'Book2': 0,
    'Book3': 10,
    'Book4': 2,
    'Book5': 10,
    'Book6': 0
}

# 1. Average
avg_borrowed = sum(borrow_records.values()) / len(borrow_records)

# 2. Max and Min
max_borrow = max(borrow_records.values())
min_borrow = min(borrow_records.values())
max_borrow_books = [book for book, count in borrow_records.items() if count ==
max_borrow]
min_borrow_books = [book for book, count in borrow_records.items() if count ==
min_borrow]

# 3. Zero borrow count
zero_borrow_count = sum(1 for count in borrow_records.values() if count == 0)

# 4. Mode borrow count
mode_borrow = Counter(borrow_records.values()).most_common(1)[0][0]

print(f"Average books borrowed: {avg_borrowed}")
print(f"Highest borrowed books: {max_borrow_books} with {max_borrow} borrowings")
print(f"Lowest borrowed books: {min_borrow_books} with {min_borrow} borrowings")
print(f"Members with zero borrowings: {zero_borrow_count}")
print(f"Most frequently borrowed book count (mode): {mode_borrow}")
```

OUTPUT:

```
F:\Study\SE\DS\Manual>"C:/Program Files/Python312/python.exe" f:/Study/SE/DS/Manual/Lib.py
Average books borrowed: 4.5
Highest borrowed books: ['Book3', 'Book5'] with 10 borrowings
Lowest borrowed books: ['Book2', 'Book6'] with 0 borrowings
Members with zero borrowings: 2
Most frequently borrowed book count (mode): 0
```

Experiment 2: Customer Account ID Search

Program

```
def linear_search(arr, target):
    for elem in arr:
        if elem == target:
            return True
    return False

def binary_search(arr, target):
    arr.sort()
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return True
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return False

accounts = [101, 205, 302, 450, 101, 500]
target = 205

print("Linear Search:", linear_search(accounts, target))
print("Binary Search:", binary_search(accounts, target))
```

OUTPUT:

```
F:\Study\SE\DS\Manual>"C:/Program Files/Python312/python.exe" f:/Study/SE/DS/Manual/2.py
Linear Search: True
Binary Search: True
```

Experiment 3: Undo/Redo System using Stack

Program

```
class UndoRedoSystem:
    def __init__(self):
        self.undo_stack = []
        self.redo_stack = []
        self.document = ""

    def make_change(self, change):
        self.undo_stack.append(self.document)
        self.document = change
        self.redo_stack.clear()

    def undo(self):
        if self.undo_stack:
            self.redo_stack.append(self.document)
            self.document = self.undo_stack.pop()
        else:
            print("No actions to undo.")

    def redo(self):
        if self.redo_stack:
            self.undo_stack.append(self.document)
            self.document = self.redo_stack.pop()
        else:
            print("No actions to redo.")

    def display(self):
        print("Current Document State:", self.document)

# Usage example
editor = UndoRedoSystem()
editor.make_change("Hello")
editor.make_change("Hello, World")
editor.display()
editor.undo()
editor.display()
editor.redo()
editor.display()
```

OUTPUT:

```
F:\Study\SE\DS\Manual>"C:/Program Files/Python312/python.exe" f:/Study/SE/DS/Manual/3.py
Current Document State: Hello, World
Current Document State: Hello
Current Document State: Hello, World
```

Experiment 4: Event Processing System using Queue

Program

```
from collections import deque

class EventSystem:
    def __init__(self):
        self.queue = deque()

    def add_event(self, event):
        self.queue.append(event)

    def process_next(self):
        if self.queue:
            event = self.queue.popleft()
            print(f"Processing event: {event}")
        else:
            print("No events to process.")

    def display_pending(self):
        print("Pending Events:", list(self.queue))

    def cancel_event(self, event):
        try:
            self.queue.remove(event)
            print(f"Canceled event: {event}")
        except ValueError:
            print("Event not found or already processed.")

# Usage
es = EventSystem()
es.add_event("E1")
es.add_event("E2")
es.display_pending()
es.process_next()
es.cancel_event("E2")
es.display_pending()
```

OUTPUT:

```
F:\Study\SE\DS\Manual>"C:/Program Files/Python312/python.exe" f:/Study/SE/DS/Manual/4.py
Pending Events: ['E1', 'E2']
Processing event: E1
Canceled event: E2
Pending Events: []
```

Experiment 5: Hashtable with Chaining

Program

```
class HashTableChaining:
    def __init__(self):
        self.size = 10
        self.table = [[] for _ in range(self.size)]

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

    def insert(self, key, value):
        idx = self.hash_function(key)
        for i, (k, v) in enumerate(self.table[idx]):
            if k == key:
                self.table[idx][i] = (key, value)
                return
        self.table[idx].append((key, value))

    def search(self, key):
        idx = self.hash_function(key)
        for k, v in self.table[idx]:
            if k == key:
                return v
        return None

    def delete(self, key):
        idx = self.hash_function(key)
        for i, (k, v) in enumerate(self.table[idx]):
            if k == key:
                del self.table[idx][i]
                return True
        return False
ht = HashTableChaining()
# Insert some key-value pairs
ht.insert(5, "Apple")
ht.insert(15, "Banana")
ht.insert(25, "Orange")
ht.insert(35, "Grape")
# Search for values
print("Value for key 15:", ht.search(15))
print("Value for key 25:", ht.search(25))
print("Value for key 10:", ht.search(10))
# Delete some entries
print("Deleting key 15:", ht.delete(15))
print("Deleting key 10:", ht.delete(10))
# Verify deletion
print("Value for key 15 after deletion:",
ht.search(15))
```

OUTPUT:

```
F:\Study\SE\DS\Manual>"C:/Program
Files/Python312/python.exe"
f:/Study/SE/DS/Manual/5.py

Value for key 15: Banana
Value for key 25: Orange
Value for key 10: None
Deleting key 15: True
Deleting key 10: False
Value for key 15 after deletion: None
```

Experiment 6: Hashtable with Linear Probing

Program

```
class HashTableLinearProbing:
    def __init__(self):
        self.size = 10
        self.table = [None] * self.size
        self.deleted = object()

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

    def insert(self, key):
        idx = self.hash_function(key)
        for i in range(self.size):
            probe = (idx + i) % self.size
            if self.table[probe] is None or self.table[probe] ==
self.deleted:
                self.table[probe] = key
                return
            elif self.table[probe] == key:
                # Key already present
                return

    def search(self, key):
        idx = self.hash_function(key)
        for i in range(self.size):
            probe = (idx + i) % self.size
            if self.table[probe] is None:
                return False
            if self.table[probe] == key:
                return True
        return False

    def delete(self, key):
        idx = self.hash_function(key)
        for i in range(self.size):
            probe = (idx + i) % self.size
            if self.table[probe] is None:
                return False
            if self.table[probe] == key:
                self.table[probe] = self.deleted
                return True
        return False

    def display(self):
        print(self.table)

if __name__ == "__main__":
    ht = HashTableLinearProbing()
    # Insert a set of keys
    for key in [5, 15, 25, 35, 3, 13]:
        ht.insert(key)
```

```
    print("After inserts:")
    ht.display()
    # Search for existing and non-existing keys
    for key in [15, 99, 3]:
        print(f"Search {key}: {ht.search(key)}")
    # Delete a key and attempt to delete a non-existing key
    print("Delete 15:", ht.delete(15))
    print("Delete 99:", ht.delete(99))
    print("After deletions:")
    ht.display()
    # Insert into slot freed by deletion
    ht.insert(99)
    print("After inserting 99:")
    ht.display()
```

OUTPUT:

```
F:\Study\SE\DS\Manual>"C:/Program Files/Python312/python.exe" f:/Study/SE/DS/Manual/6.py
After inserts:
[None, None, None, 3, 13, 5, 15, 25, 35, None]
Search 15: True
Search 99: False
Search 3: True
Delete 15: True
Delete 99: False
After deletions:
[None, None, None, 3, 13, 5, <object object at 0x0000019B91E38240>, 25, 35, None]
After inserting 99:
[None, None, None, 3, 13, 5, <object object at 0x0000019B91E38240>, 25, 35, 99]
```

Experiment 7: Graph Traversal of City Locations using BFS and DFS

Program	OUTPUT:

`# Graph nodes: A, B, C, D, E (index 0 to 4)`
`# Adjacency matrix representation for DFS`
`graph_matrix = [`
`  [0, 1, 1, 0, 0],  # A connected to B and C`
`  [1, 0, 0, 1, 0],  # B connected to A and D`
`  [1, 0, 0, 1, 1],  # C connected to A, D, E`
`  [0, 1, 1, 0, 1],  # D connected to B, C, E`
`  [0, 0, 1, 1, 0]   # E connected to C, D`
`]`
`def dfs(node, visited):`
`  visited[node] = True`
`  print(chr(node + 65), end=' ')  # Convert 0->A, 1->B, etc.`
`  for neighbor in range(len(graph_matrix)):`
`  if graph_matrix[node][neighbor] == 1 and not visited[neighbor]:`
`  dfs(neighbor, visited)`
`print("DFS traversal starting from A:")`
`visited = [False] * 5`
`dfs(0, visited)  # Start at node A (0)`
`# Adjacency list representation for BFS`
`graph_list = {`
`  0: [1, 2],  # A connected to B, C`
`  1: [0, 3],  # B connected to A, D`
`  2: [0, 3, 4],# C connected to A, D, E`
`  3: [1, 2, 4],# D connected to B, C, E`
`  4: [2, 3]  # E connected to C, D`
`}`
`from collections import deque`
`def bfs(start):`
`  visited = [False] * 5`
`  queue = deque([start])`
`  visited[start] = True`
`  while queue:`
`  node = queue.popleft()`
`  print(chr(node + 65), end=' ')`
`  for neighbor in graph_list[node]:`
`  if not visited[neighbor]:`
`  visited[neighbor] = True`
`  queue.append(neighbor)`
`print("\nBFS traversal starting from A:")`
`bfs(0)`
	F:\Study\SE\DS\Manual>"C:/Program Files/Python312/python.exe" f:/Study/SE/DS/Manual/7.py\
	
	DFS traversal starting from A:
	A B D C E
	
	BFS traversal starting from A:
	A B C D E

Experiment 8: Pizza Delivery Minimum Time Problem using Graphs

Program

```
import heapq
def dijkstra(graph, start):
    n = len(graph)
    distances = [float('inf')] * n
    distances[start] = 0
    pq = [(0, start)]  # (distance, node)

    while pq:
        dist, node = heapq.heappop(pq)
        if dist > distances[node]:
            continue
        for neighbor, weight in graph[node]:
            new_dist = dist + weight
            if new_dist < distances[neighbor]:
                distances[neighbor] = new_dist
                heapq.heappush(pq, (new_dist, neighbor))
    return distances
# Graph represented as adjacency list with weights
# Node 0 = Pizza shop, nodes 1-4 = customer locations
graph = {
    0: [(1, 10), (2, 15)],
    1: [(0, 10), (3, 12), (4, 15)],
    2: [(0, 15), (4, 10)],
    3: [(1, 12), (4, 2)],
    4: [(1, 15), (2, 10), (3, 2)]
}
distances = dijkstra(graph, 0)
print("Minimum time from pizza shop to each location:")
for node in range(len(distances)):
    print(f"Location {node}: {distances[node]} minutes")
# Sum or route planning to minimize total delivery time can be complex,
# this example shows shortest paths from shop to each location.
```

OUTPUT:

```
F:\Study\SE\DS\Manual>"C:/Program Files/Python312/python.exe" f:/Study/SE/DS/Manual/8.py
Minimum time from pizza shop to each location:
Location 0: 0 minutes
Location 1: 10 minutes
Location 2: 15 minutes
Location 3: 22 minutes
Location 4: 24 minutes
```