LAB # 10
LIST, TUPLE, DICTIONARY, CLASS OBJECT AND IMPLEMENT PRIORITY QUEUE

OBJECTIVE:
Familiarization with python language using list, tuple, dictionary, class and object and Manage a set of records with priority queue using queue and heapq module in python

Lab Tasks:

1. Store the names of a few of your friends in a list called names. Print each person's name by accessing each element in the list, one at a time.

- **CODE:**

```
names = ["Alice", "Bob", "Charlie", "David", "Eve"]
for name in names:
    print(name)
```

- **OUTPUT:**

```
Alice
Bob
Charlie
David
Eve
```

2. If you could invite anyone, living or deceased, to dinner, who would you invite? Make a list that includes at least three people you'd like to invite to dinner. Then use your list to print a message to each person, inviting them to dinner.

- **CODE:**

```
guests = ["Albert Einstein", "Marie Curie", "Leonardo da Vinci"]
for guest in guests:
    print(f"Dear {guest},\nI would be honored to invite you to dinner!\n")
```

- **OUTPUT:**

```
Dear Albert Einstein,
I would be honored to invite you to dinner!

Dear Marie Curie,
I would be honored to invite you to dinner!

Dear Leonardo da Vinci,
I would be honored to invite you to dinner!
```

3. **Changing Guest List:** You just heard that one of your guests can't make the dinner, so you need to send out a new set of invitations. You'll have to think of someone else to invite.

- Modify your list, replacing the name of the guest who can't make it with the name of the new person you are inviting.
- Print a second set of invitation messages, one for each person who is still in your list.

- **CODE:**

```
guests = ["Albert Einstein", "Marie Curie", "Leonardo da Vinci"]
for guest in guests:
    print(f"Dear {guest},\nI would be honored to invite you to dinner!\n")
```

- **OUTPUT:**

```
Dear Albert Einstein,
I would be honored to invite you to dinner!

Dear Nikola Tesla,
I would be honored to invite you to dinner!

Dear Leonardo da Vinci,
I would be honored to invite you to dinner!
```

4. Create a Class "Employee", it's a common base class for all the employee. Then initialize employee's parameter like empName and salary and create function like displayCount() contain total number of employee in your knowledge base and displayEmployee() contain empName and their salary.

- **CODE:**

```
class Employee:
    employee_count = 0
    def __init__(self, empName, salary):
        self.empName = empName
        self.salary = salary
        Employee.employee_count += 1  # Increment the employee count each time a new employee is created
    @classmethod
    def displayCount(cls):
        print(f"Total number of employees: {cls.employee_count}")

    # Function to display employee's name and salary
    def displayEmployee(self):
        print(f"Employee Name: {self.empName}, Salary: {self.salary}")


# Creating instances of Employee class
emp1 = Employee("John Doe", 50000)
emp2 = Employee("Jane Smith", 60000)
emp3 = Employee("Alice Johnson", 70000)

# Display the count of employees
Employee.displayCount()

# Display each employee's name and salary
emp1.displayEmployee()
emp2.displayEmployee()
emp3.displayEmployee()
```

- **OUTPUT:**

```
Total number of employees: 3
Employee Name: John Doe, Salary: 50000
Employee Name: Jane Smith, Salary: 60000
Employee Name: Alice Johnson, Salary: 70000
```

5. In contrast to the standard FIFO implementation of Queue, the LifoQueue uses last-in, first- out ordering (normally associated with a stack data structure). Implement LIFO queue using queue module.

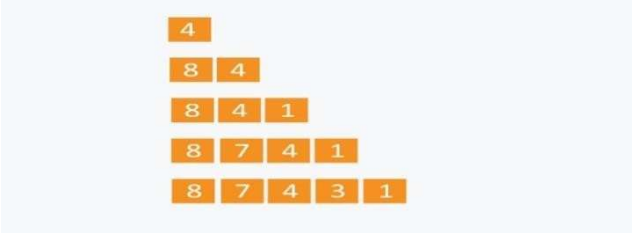
- **CODE:**

```
import queue

# Create a LIFO Queue
lifo_queue = queue.LifoQueue()

# Adding items to the queue (Push operation)
lifo_queue.put(10)
lifo_queue.put(20)
lifo_queue.put(30)

# Displaying the size of the queue
print(f"Queue size: {lifo_queue.qsize()}")

# Removing items from the queue (Pop operation)
print(f"Removed item: {lifo_queue.get()}")  # 30
print(f"Removed item: {lifo_queue.get()}")  # 20
print(f"Removed item: {lifo_queue.get()}")  # 10

# Queue is now empty
print(f"Queue size after removal: {lifo_queue.qsize()}")
```

- **OUTPUT:**

```
Queue size: 3
Removed item: 30
Removed item: 20
Removed item: 10
Queue size after removal: 0
```

6. We have an array of 5 elements: [4, 8, 1, 7, 3] and we have to insert all the elements in the max-priority queue. First as the priority queue is empty, so 4 will be inserted initially. Now when 8 will be inserted it will move to front as 8 is greater than 4. While inserting 1, as it is the current minimum element in the priority queue, it will remain in the back of priority queue. Now 7 will be inserted between 8 and 4 as 7 is smaller than 8. Now 3 will be inserted before 1 as it is the 2nd minimum element in the priority queue. All the steps are represented in the diagram below:



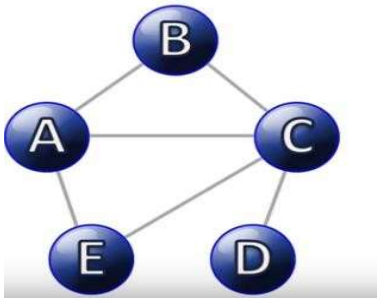
- **CODE:**

```
import heapq

# Initialize an empty list for the max-priority queue
max_queue = []

# Array of elements to insert into the queue
elements = [4, 8, 1, 7, 3]

# Insert elements one by one
print("Inserting elements into the priority queue:")
for num in elements:
    heapq.heappush(max_queue, -num)  # Push the negative value to simulate max-heap
    print("Inserted", num, "into the priority queue:", [-x for x in max_queue])

# Final state of the priority queue
print("\nFinal priority queue:", [-x for x in max_queue])

# Optional: Display the elements in order of their extraction (removal from the queue)
print("\nExtracting elements from the priority queue:")
while max_queue:
    print("Extracted:", -heapq.heappop(max_queue))
```

- **OUTPUT:**

```
Inserting elements into the priority queue:
Inserted 4 into the priority queue: [4]
Inserted 8 into the priority queue: [8, 4]
Inserted 1 into the priority queue: [8, 4, 1]
Inserted 7 into the priority queue: [8, 7, 1, 4]
Inserted 3 into the priority queue: [8, 7, 1, 4, 3]

Final priority queue: [8, 7, 1, 4, 3]

Extracting elements from the priority queue:
Extracted: 8
Extracted: 7
Extracted: 4
Extracted: 3
Extracted: 1
```

7. Implement graph using adjacency list using list or dictionary, make a class such as Vertex and Graph then make some function such as add_nodes, add_edges, add_neighbors, add_vertex, add_vertices and suppose whatever you want to need it.



- **CODE:**

```
class Vertex:
    def __init__(self, value):
        self.value = value  # Value of the vertex
        self.neighbors = []  # List to store neighbors (edges)

    def add_neighbor(self, vertex):
        """Add a neighbor to the vertex"""
        self.neighbors.append(vertex)

    def __str__(self):
        return f"Vertex({self.value}) with neighbors: {[neighbor.value for neighbor in self.neighbors]}"


class Graph:
    def __init__(self):
        self.vertices = {}  # Dictionary to store vertices by their value

    def add_vertex(self, value):
        """Add a vertex to the graph"""
        if value not in self.vertices:
            self.vertices[value] = Vertex(value)
        else:
            print(f"Vertex {value} already exists.")

    def add_vertices(self, values):
        """Add multiple vertices to the graph"""
        for value in values:
            self.add_vertex(value)

    def add_edge(self, from_value, to_value):
        """Add a directed edge from 'from_value' to 'to_value'"""
        if from_value in self.vertices and to_value in self.vertices:
            from_vertex = self.vertices[from_value]
            to_vertex = self.vertices[to_value]
```

```
            from_vertex.add_neighbor(to_vertex)
        else:
            print("One or both vertices not found in the graph.")

    def add_edges(self, edges):
        """Add multiple edges to the graph"""
        for edge in edges:
            self.add_edge(edge[0], edge[1])

    def get_neighbors(self, value):
        """Get all neighbors of a vertex"""
        if value in self.vertices:
            return [neighbor.value for neighbor in self.vertices[value].neighbors]
        else:
            print(f"Vertex {value} not found.")
            return []

    def __str__(self):
        return '\n'.join([str(vertex) for vertex in self.vertices.values()])


# Example usage:
graph = Graph()

# Add vertices
graph.add_vertices([1, 2, 3, 4, 5])

# Add edges
graph.add_edges([(1, 2), (1, 3), (2, 4), (3, 5)])

# Print the graph with adjacency lists
print("Graph:")
print(graph)

# Get neighbors of a vertex
print("\nNeighbors of vertex 1:")
print(graph.get_neighbors(1))
```

```
print("\nNeighbors of vertex 2:")
print(graph.get_neighbors(2))
```

- **OUTPUT:**

```
Graph:
Vertex(1) with neighbors: [2, 3]
Vertex(2) with neighbors: [4]
Vertex(3) with neighbors: [5]
Vertex(4) with neighbors: []
Vertex(5) with neighbors: []

Neighbors of vertex 1:
[2, 3]

Neighbors of vertex 2:
[4]
```

Files Upload On Github: