Assignment: Data Structure

Submitted by:
Najim Khan
B.Tech CSE (DS)
Roll No: 2401420003

Q1: Define the Weather Record ADT with the specified attributes and methods

■ Summary
The Weather Record ADT stores temperature data for multiple years and cities.
It supports these attributes and methods:
Attributes: years, cities, data (2D array).
Methods:
- insert(year, city, temp)
- delete(year, city)
- retrieve(year, city)

```
class WeatherRecordADT:
    def __init__(self, years, cities):
        self.years = years
        self.cities = cities
        # 2D array with None as sentinel
        self.data = [[None for _ in range(cities)] for _ in range(years)]

    def insert(self, year, city, temp):
        self.data[year][city] = temp

    def delete(self, year, city):
        self.data[year][city] = None

    def retrieve(self, year, city):
        return self.data[year][city]
```

Q2: Implement a 2D array-based storage system for year-wise, city-wise temperature data

■ Summary
We store temperatures in a 2D array where rows = years and columns = cities.
Each cell data[year][city] holds the temperature.

```
wr = WeatherRecordADT(3, 4)

# Insert some records
wr.insert(0, 0, 25)
wr.insert(0, 1, 30)
wr.insert(1, 2, 28)
wr.insert(2, 3, 22)

# Display
print("2D Array Storage:")
```

```
for y in range(wr.years):
    print(f"Year {y}: {wr.data[y]}")
```

Output Example:

```
Year 0: [25, 30, None, None]
Year 1: [None, None, 28, None]
Year 2: [None, None, None, 22]
```

Q3: Develop row-major and column-major access methods and compare their efficiency

■ Summary
Row-major → traverse year by year.
Column-major → traverse city by city.
Both are O(Y × C) but row-major is faster in Python because lists are stored row-wise.

```
class WeatherRecordWithTraversal(WeatherRecordADT):
    def row_major(self):
        print("Row-major traversal:")
        for y in range(self.years):
            for c in range(self.cities):
                print(f"[{y},{c}] = {self.data[y][c]}", end="  ")
            print()

    def column_major(self):
        print("Column-major traversal:")
        for c in range(self.cities):
            for y in range(self.years):
                print(f"[{y},{c}] = {self.data[y][c]}", end="  ")
            print()
```

Q4: Implement a mechanism to handle sparse data

■ Summary
When many entries are missing, storing a full 2D array wastes memory.
We can:
- Use None as sentinel in the array.
- Use a dictionary ((year, city): temp) to store only existing records.

```
class WeatherRecordSparse(WeatherRecordWithTraversal):
    def __init__(self, years, cities):
        super().__init__(years, cities)
        self.sparse = {}  # dict for sparse storage

    def insert(self, year, city, temp):
        super().insert(year, city, temp)
        self.sparse[(year, city)] = temp

    def delete(self, year, city):
        super().delete(year, city)
        if (year, city) in self.sparse:
            del self.sparse[(year, city)]

    def retrieve_sparse(self, year, city):
        return self.sparse.get((year, city), None)
    def display_sparse(self):
```

```
        print("Sparse Storage:", self.sparse)
```

Q5: Analyze and document the time and space complexity

■ Summary
Operation | Array Storage | Sparse Dict
Insert | O(1) | O(1) avg
Delete | O(1) | O(1) avg
Retrieve | O(1) | O(1) avg
Space Usage | O(Y × C) | O(k), where k = number of non-empty records

- Array is good for dense datasets.
- Sparse Dict saves memory for sparse datasets.

```
# No code required for complexity analysis.
```