K.R. MANGALAM UNIVERSITY

THE COMPLETE WORLD OF EDUCATION

Assignment 1

Weather Data Storage System

Department: CSE

Session: 2025-26

Programme: BTech

Semester: 3rd

Course Code: ENCS205

Course: Data Structures

Assignment No: 01

Submitted by: Hemant Saini

Roll no. 2401420051

Submitted to: Dr Swati

1. Introduction

This assignment focuses on the implementation of a Weather Data Storage System using Abstract Data Types (ADTs) and 2D arrays in Python. The system is designed to store, retrieve, and manage weather-related data such as temperature for multiple cities over multiple years. It also demonstrates handling of sparse datasets, and comparison of row-major and column-major access.

2. Problem Statement

The task is to develop a software solution in Python that organizes and retrieves temperature data for different cities and years using structured data formats like 2D arrays and ADTs.

3. Weather Record ADT Design

Attributes:

- a. Date: String (day/month/year)
- b. City: String (name of the city)
- c. Temperature: double (temperature value)

Methods:

- • insert(data): Insert a new weather record
- • delete(data): Remove a record by city and date
- • retrieve(city, year): Retrieve temperature data for a specific city and year
- • rowMajorAccess(): Print all records row by row (year-wise)
- • columnMajorAccess(): Print all records column by column (city-wise)

4. Implementation (Python Code)

```
class WeatherRecord:
  def __init__(self, date, city, temperature):
    self.date = date
    self.city = city
    self.temperature = temperature

class WeatherStorage:
  def __init__(self, years, cities):
```

```
        self.years = years
        self.cities = cities
        self.city_index = {city: idx for idx, city in enumerate(cities)}
        self.data = [[None for _ in cities] for _ in years]

    def insert(self, record):
        year = int(record.date.split("/")[-1])
        if year in self.years and record.city in self.city_index:
            row = self.years.index(year)
            col = self.city_index[record.city]
            self.data[row][col] = record.temperature

    def delete(self, date, city):
        year = int(date.split("/")[-1])
        if year in self.years and city in self.city_index:
            row = self.years.index(year)
            col = self.city_index[city]
            self.data[row][col] = None

    def retrieve(self, city, year):
        if year in self.years and city in self.city_index:
            row = self.years.index(year)
            col = self.city_index[city]
            return self.data[row][col]
        return None

    def row_major_access(self):
        for row in range(len(self.years)):
            for col in range(len(self.cities)):
                print(self.years[row], self.cities[col], self.data[row][col])

    def column_major_access(self):
        for col in range(len(self.cities)):
            for row in range(len(self.years)):
                print(self.years[row], self.cities[col], self.data[row][col])
```

5. Row-Major vs Column-Major Access

Row-Major Access: Data is accessed row by row (efficient in Python when stored as nested lists).
Column-Major Access: Data is accessed column by column (useful for city-based analysis).

6. Sparse Data Handling

Sparse datasets are handled using sentinel values (None). Additionally, a sparse matrix representation was implemented, storing only non-empty records as tuples: (year, city, temperature). This reduces space usage when many entries are missing.

7. Complexity Analysis

Insert/Delete/Retrieve: O(1)

Row/Column Access: O(n × m), where n = years and m = cities

Space Complexity: O(n × m) for dense storage, O(k) for sparse (where k is non-empty records)

8. Conclusion

This assignment demonstrates the design and implementation of a Weather Data Storage System using ADTs and 2D arrays in Python. It highlights efficient data retrieval, comparison of memory access patterns, and sparse data handling techniques, fulfilling the objectives of the assignment.