Assignment: Data Structure

Submitted by:

## Manan Gola

B.Tech CSE (DS) Roll No: 2401420043

# 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 WeatherRecord:

def \_init\_(self, city, year, temperature):

self.city = city

self.year = year

self.temperature = temperature

def update\_temperature(self, new\_temp):

self.temperature = new\_temp

def display(self):

return f"City: {self.city}, Year: {self.year}, Temperature: {self.temperature}°C"

# Example usage

record = WeatherRecord("Delhi", 2024, 32.5)

print(record.display())

record.update\_temperature(33.8)

print(record.display())

# 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.

# Suppose: Rows = years, Columns = cities

years = [2022, 2023, 2024]

cities = ["Delhi", "Mumbai", "Chennai"]

# Initialize 2D array (rows = years, cols = cities)

temperature\_data = [[None for \_ in cities] for \_ in years]

# Insert sample values

temperature\_data[0][0] = 32.5 # Delhi, 2022

temperature\_data[1][1] = 30.0 # Mumbai, 2023

temperature\_data[2][2] = 35.2 # Chennai, 2024

# Display matrix

for i, year in enumerate(years):

print(year, ":", temperature\_data[i])

# 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.

# Using sentinel value (-1 means missing data)

rows, cols = 4, 4

sparse\_matrix = [[-1 for \_ in range(cols)] for \_ in range(rows)]

sparse\_matrix[0][1] = 25

sparse\_matrix[2][3] = 30

print("Sparse with Sentinel:")

for row in sparse\_matrix:

print(row)

# Using dictionary for sparse storage

sparse\_dict = {}

sparse\_dict[(0, 1)] = 25

sparse\_dict[(2, 3)] = 30

print("\nSparse with Dictionary:")

for key, val in sparse\_dict.items():

print(f"Position {key}: {val}")

# 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.