Data Structure Assignment- 1
Course Code: ENCS205

TITLE:

Weather Data Storage using Arrays (Python Implementation)

Name:
PARTH GOYAL
Roll No:
2401420028
Course:
B.Tech-CSE-DS

INTRODUCTION

This project presents a simple, easy-to-understand weather data storage tool written in Python. The system stores temperature observations organized by year and city using a 2-dimensional array representation. Because many combinations of year and city may be empty, the design includes a method to mark missing entries and support sparse datasets. The program demonstrates basic operations such as adding new records, removing entries, fetching data for a city in a particular year, and iterating through stored values in both row-major and column-major order.

OBJECTIVES

- **Build a compact Weather Record ADT (date, city, temperature).**
- **Store temperature data in a 2D array with rows representing years and columns representing cities.**
- **Implement insertion, deletion, and retrieval operations for weather records.**
- **Demonstrate and compare row-major and column-major traversal.**
- **Provide a way to manage sparse data (sentinel or coordinate list).**
- **Analyse time and space complexity of the main operations.**

SYSTEM DESIGN

Data Model (Weather Record ADT):

Each weather measurement is represented by a small record containing:

- **date — string in DD/MM/YYYY format**
- **city — city name string**
- **temperature — floating point value (°C)**

In the code this ADT is implemented as a simple dictionary per record, created using a helper function.

Storage Layout:

- **YEARS: a list maintaining distinct years (each list index = a matrix row).**
- **CITIES: a list maintaining distinct cities (each list index = a matrix column).**
- **MATRIX: a 2D Python list (MATRIX[row][col]) where each cell either holds:**
  - **a list of records for that (year, city) pair, or**
  - **a sentinel (None) when there is no data (to mark sparsity).**
- **RECORDS: a flat list of all inserted records (for simpler display/CRUD operations).**

This hybrid layout lets the program both (a) maintain a simple 2D representation for row/column traversal and (b) allow flat CRUD operations via the records list.

Key Operations (overview):

- **INSERT() — add a single record (updates YEARS, CITIES, MATRIX, and RECORDS).**
- **CREATE_RECORDS() — create several records in a short batch (interactive).**
- **DELETE() — remove records by date and city (cleans matrix cell and RECORDS).**
- **RETRIEVE() — list all records for a specified city and year.**
- **POPULATE() — fill the structure with a small demo dataset (for testing).**
- **POPULATE_ARRAY() — let the user enter years and cities and fill the matrix interactively.**
- **ROW_MAJOR() / COLUMN_MAJOR() — iterate through MATRIX in row-major or column-major order.**
- **SPARSE() — produce a coordinate-style list of only the non-empty entries.**
- **PRINT_TABLE() — display a compact table summary showing counts per cell.**
- **COMPLEXITY() — print time & space analysis for main operations.**

IMPLEMENTATION

C: > Users > goyal > Desktop > DS > DSAssignment_2401420028.py > ...

```python
# WEATHER DATA STORAGE SYSTEM

YEARS = []      # list of years (rows)
CITIES = []     # list of cities (columns)
MATRIX = []     # 2D array: MATRIX[row][col] stores list of record dicts or None
RECORDS = []    # flat list of all records (date, city, temperature)
SENTINEL = None # sentinel for sparse cells


# Helpers
def YEAR_INDEX(y):
    try:
        return YEARS.index(y)
    except ValueError:
        return -1

def CITY_INDEX(c):
    try:
        return CITIES.index(c)
    except ValueError:
        return -1

def MAKE_RECORD(date, city, temperature):
    return {"date": date, "city": city, "temperature": float(temperature)}


# 1. INSERT
def INSERT():
    date = input("Enter Date (DD/MM/YYYY): ").strip()
```

```
        city = input("Enter City: ").strip()
        temp_s = input("Enter Temperature: ").strip()
        try:
            temp = float(temp_s)
        except ValueError:
            print("Invalid temperature!")
            return
        parts = date.split("/")
        if len(parts) != 3:
            print("Invalid Date Format! Use DD/MM/YYYY")
            return
        year = int(parts[2])

        # Update YEARS and CITIES
        if year not in YEARS:
            YEARS.append(year)
            YEARS.sort()
        if city not in CITIES:
            CITIES.append(city)

        # Matrix size
        rows = len(YEARS)
        cols = len(CITIES)
        while len(MATRIX) < rows:
            MATRIX.append([SENTINEL for _ in range(cols)])
        for r in range(len(MATRIX)):
            while len(MATRIX[r]) < cols:
                MATRIX[r].append(SENTINEL)

    r = YEAR_INDEX(year)
    c = CITY_INDEX(city)
    if MATRIX[r][c] == SENTINEL:
        MATRIX[r][c] = []
    rec = MAKE_RECORD(date, city, temp)
    MATRIX[r][c].append(rec)
    RECORDS.append(rec)
    print("Record Inserted Successfully!")


# 2. CREATE BATCH
def CREATE_RECORDS():
    try:
        n = int(input("Enter number of weather records to create: ").strip())
    except ValueError:
        print("Invalid number.")
        return
    for i in range(n):
        print("\nEnter details for weather record " + str(i+1) + ":")
        date = input("Enter date (DD/MM/YYYY): ").strip()
        city = input("Enter city name: ").strip()
        temp_s = input("Enter temperature (in Celsius): ").strip()
        try:
            temp = float(temp_s)
        except ValueError:
            print("Invalid temperature - skipping this record.")
            continue
```

```
86
87          parts = date.split("/")
88          if len(parts) != 3:
89              print("Invalid date format - skipping this record.")
90              continue
91          year = int(parts[2])
92          if year not in YEARS:
93              YEARS.append(year)
94              YEARS.sort()
95          if city not in CITIES:
96              CITIES.append(city)
97          rows = len(YEARS)
98          cols = len(CITIES)
99          while len(MATRIX) < rows:
100             MATRIX.append([SENTINEL for _ in range(cols)])
101         for r in range(len(MATRIX)):
102             while len(MATRIX[r]) < cols:
103                 MATRIX[r].append(SENTINEL)
104         r = YEAR_INDEX(year)
105         c = CITY_INDEX(city)
106         if MATRIX[r][c] == SENTINEL:
107             MATRIX[r][c] = []
108         rec = MAKE_RECORD(date, city, temp)
109         MATRIX[r][c].append(rec)
110         RECORDS.append(rec)
111     print("Batch creation complete.")
112
113
114 # 3. DELETE (by date and city)
115 def DELETE():
116     date = input("Enter Date (DD/MM/YYYY) to delete: ").strip()
117     city = input("Enter City: ").strip()
118     parts = date.split("/")
119     if len(parts) != 3:
120         print("Invalid Date Format!")
121         return
122     year = int(parts[2])
123     if year not in YEARS or city not in CITIES:
124         print("Record Not Found!")
125         return
126     r = YEAR_INDEX(year)
127     c = CITY_INDEX(city)
128     if MATRIX[r][c] == SENTINEL:
129         print("No Records Found!")
130         return
131     before = len(MATRIX[r][c])
132     # Remove matching date entries in that cell
133     MATRIX[r][c] = [rec for rec in MATRIX[r][c] if rec["date"] != date]
134     after = len(MATRIX[r][c])
135     # Update RECORDS list to remove matching entries
136     RECORDS[:] = [rec for rec in RECORDS if not (rec["date"] == date and rec["city"] == city)]
137     if after == 0:
138         MATRIX[r][c] = SENTINEL
139     if before == after:
140         print("No Record Found for Given Date.")
141     else:
142         print("Record Deleted Successfully!")
```

```
143
144
145 # 4. RETRIEVE (by city and year)
146 def RETRIEVE():
147     city = input("Enter City: ").strip()
148     year_s = input("Enter Year (YYYY): ").strip()
149     try:
150         year = int(year_s)
151     except ValueError:
152         print("Invalid Year!")
153         return
154     if year not in YEARS or city not in CITIES:
155         print("No Records Found!")
156         return
157     r = YEAR_INDEX(year)
158     c = CITY_INDEX(city)
159     if MATRIX[r][c] == SENTINEL:
160         print("No Data Available!")
161         return
162     print(f"Records for {city} in {year}:")
163     for rec in MATRIX[r][c]:
164         print(f"Date: {rec['date']}  Temp: {rec['temperature']}°C")
165
166
167 # 5. POPULATE DEMO DATA
168 def POPULATE():
169     demo = [
170         ("01/01/2024", "Delhi", 15.5),
171         ("02/01/2024", "Delhi", 16.0),
172         ("01/01/2024", "Mumbai", 24.0),
173         ("15/02/2023", "Delhi", 20.2),
174         ("10/03/2025", "Chennai", 29.0)
175     ]
176     for d, c, t in demo:
177         if c not in CITIES:
178             CITIES.append(c)
179         if int(d.split("/")[2]) not in YEARS:
180             YEARS.append(int(d.split("/")[2]))
181     YEARS.sort()
182     rows, cols = len(YEARS), len(CITIES)
183     while len(MATRIX) < rows:
184         MATRIX.append([SENTINEL for _ in range(cols)])
185     for r in range(rows):
186         while len(MATRIX[r]) < cols:
187             MATRIX[r].append(SENTINEL)
188     for d, c, t in demo:
189         y = int(d.split("/")[2])
190         r = YEAR_INDEX(y)
191         col = CITY_INDEX(c)
192         if MATRIX[r][col] == SENTINEL:
193             MATRIX[r][col] = []
194         rec = MAKE_RECORD(d, c, t)
195         MATRIX[r][col].append(rec)
196         RECORDS.append(rec)
197     print("Demo Data Populated!")
198
199
```

```python
# 6. ROW-MAJOR ACCESS
def ROW_MAJOR():
    print("ROW-MAJOR ACCESS:")
    for r in range(len(YEARS)):
        for c in range(len(CITIES)):
            if MATRIX[r][c] == SENTINEL:
                print(f"Year {YEARS[r]} City {CITIES[c]}: No Data")
            else:
                print(f"Year {YEARS[r]} City {CITIES[c]}: {MATRIX[r][c]}")


# 7. COLUMN-MAJOR ACCESS
def COLUMN_MAJOR():
    print("COLUMN-MAJOR ACCESS:")
    for c in range(len(CITIES)):
        for r in range(len(YEARS)):
            if MATRIX[r][c] == SENTINEL:
                print(f"City {CITIES[c]} Year {YEARS[r]}: No Data")
            else:
                print(f"City {CITIES[c]} Year {YEARS[r]}: {MATRIX[r][c]}")


# 8. SPARSE REPRESENTATION
def SPARSE():
    print("SPARSE REPRESENTATION:")
    sparse_list = []
    for r in range(len(YEARS)):
        for c in range(len(CITIES)):
            if MATRIX[r][c] != SENTINEL:
                for rec in MATRIX[r][c]:
                    sparse_list.append((YEARS[r], CITIES[c], rec["date"], rec["temperature"]))
    if not sparse_list:
        print("No Data Available!")
    else:
        for item in sparse_list:
            print(item)


# 9. DISPLAY RECORDS (flat list)
def DISPLAY_RECORDS():
    if not RECORDS:
        print("No weather records found.")
        return
    print("\nWeather Record Details:")
    for rec in RECORDS:
        print("Date: " + rec['date'] + ", Temperature: " + str(rec['temperature']) + " °C, City: " + rec['city'])


# 10. POPULATE ARRAY FROM USER
def POPULATE_ARRAY():
    years_input = input("Enter years separated by commas (e.g. 2023,2024): ").strip()
    cities_input = input("Enter cities separated by commas (e.g. Delhi,Mumbai): ").strip()
    years = [int(y.strip()) for y in years_input.split(",") if y.strip() != ""]
    cities = [c.strip() for c in cities_input.split(",") if c.strip() != ""]
    # Set globals
    global YEARS, CITIES, MATRIX, RECORDS
    YEARS = years
```

```
    CITIES = cities
    MATRIX = []
    RECORDS = []
    rows = len(YEARS)
    cols = len(CITIES)
    for i in range(rows):
        row = []
        for j in range(cols):
            val = input(f"Enter temperature for {CITIES[j]} in {YEARS[i]} (or press Enter to skip): ").strip()
            if val == "":
                row.append(SENTINEL)
            else:
                try:
                    t = float(val)
                except ValueError:
                    print("Invalid numeric input, storing as SENTINEL.")
                    row.append(SENTINEL)
                    continue
                rec_date = "01/01/" + str(YEARS[i])  # approximate date when only year given
                rec = MAKE_RECORD(rec_date, CITIES[j], t)
                row.append([rec])  # store as list of records
                RECORDS.append(rec)
        MATRIX.append(row)
    print("User-populated matrix created.")


# 11. COMPLEXITY ANALYSIS
def COMPLEXITY():
    print("\nTIME COMPLEXITY:")
    print("Insert: O(R + C) worst case (search + resize), typically O(1) for appending to cell list")
    print("Delete: O(R + C + M) where M = records in cell")
    print("Retrieve: O(R + C + M)")
    print("Row/Col Major Access: O(R * C)")
    print("\nSPACE COMPLEXITY:")
    print("Matrix: O(R * C) + Records: O(K)")


# 12. PRINT TABLE
def PRINT_TABLE():
    if not YEARS or not CITIES:
        print("No table to print.")
        return
    header = "Year/City".ljust(12)
    for city in CITIES:
        header += city.ljust(12)
    print(header)
    for i in range(len(YEARS)):
        row_str = str(YEARS[i]).ljust(12)
        for j in range(len(CITIES)):
            val = MATRIX[i][j]
            if val is None or val == SENTINEL:
                row_str += "None".ljust(12)
            else:
                # Show number of records in cell
                row_str += (str(len(val)) + " rec").ljust(12)
        print(row_str)
```

```python


# MAIN MENU
def MAIN():
    while True:
        print("\n--- WEATHER DATA STORAGE SYSTEM ---")
        print("1. Insert Record")
        print("2. Create Multiple Records")
        print("3. Delete Record (by date & city)")
        print("4. Retrieve Records (by city & year)")
        print("5. Populate Demo Data")
        print("6. Populate Matrix (user input years & cities)")
        print("7. Row-Major Access")
        print("8. Column-Major Access")
        print("9. Sparse Representation")
        print("10. Display Flat Records")
        print("11. Print Table Summary")
        print("12. Complexity Analysis")
        print("13. Exit")

        choice = input("Enter Choice (1-13): ").strip()

        if choice == '1':
            INSERT()
        elif choice == '2':
            CREATE_RECORDS()
        elif choice == '3':
            DELETE()
        elif choice == '4':
            RETRIEVE()
        elif choice == '5':
            POPULATE()
        elif choice == '6':
            POPULATE_ARRAY()
        elif choice == '7':
            ROW_MAJOR()
        elif choice == '8':
            COLUMN_MAJOR()
        elif choice == '9':
            SPARSE()
        elif choice == '10':
            DISPLAY_RECORDS()
        elif choice == '11':
            PRINT_TABLE()
        elif choice == '12':
            COMPLEXITY()
        elif choice == '13':
            print("Exiting Weather System... Good luck!")
            break
        else:
            print("Invalid Choice! Please enter 1-13.")


if __name__ == "__main__":
    MAIN()
```

OUTPUT

```
--- WEATHER DATA STORAGE SYSTEM ---
1. Insert Record
2. Create Multiple Records
3. Delete Record (by date & city)
4. Retrieve Records (by city & year)
5. Populate Demo Data
6. Populate Matrix (user input years & cities)
7. Row-Major Access
8. Column-Major Access
9. Sparse Representation
10. Display Flat Records
11. Print Table Summary
12. Complexity Analysis
13. Exit
Enter Choice (1-13):
```

```
ROW-MAJOR ACCESS:
Year 2023 City Delhi: [{'date': '15/02/2023', 'city': 'Delhi', 'temperature': 20.2}]
Year 2023 City Mumbai: No Data
Year 2023 City Chennai: No Data
Year 2024 City Delhi: [{'date': '01/01/2024', 'city': 'Delhi', 'temperature': 15.5}, {'date': '02/01/2024', 'city': 'Delhi', 'temperature': 16.0}]
Year 2024 City Mumbai: [{'date': '01/01/2024', 'city': 'Mumbai', 'temperature': 24.0}]
Year 2024 City Chennai: No Data
Year 2025 City Delhi: No Data
Year 2025 City Mumbai: No Data
Year 2025 City Chennai: [{'date': '10/03/2025', 'city': 'Chennai', 'temperature': 29.0}]
```

```
COLUMN-MAJOR ACCESS:
City Delhi Year 2023: [{'date': '15/02/2023', 'city': 'Delhi', 'temperature': 20.2}]
City Delhi Year 2024: [{'date': '01/01/2024', 'city': 'Delhi', 'temperature': 15.5}, {'date': '02/01/2024', 'city': 'Delhi', 'temperature': 16.0}]
City Delhi Year 2025: No Data
City Mumbai Year 2023: No Data
City Mumbai Year 2024: [{'date': '01/01/2024', 'city': 'Mumbai', 'temperature': 24.0}]
City Mumbai Year 2025: No Data
City Chennai Year 2023: No Data
City Chennai Year 2024: No Data
City Chennai Year 2025: [{'date': '10/03/2025', 'city': 'Chennai', 'temperature': 29.0}]
```

```
SPARSE REPRESENTATION:
(2023, 'Delhi', '15/02/2023', 20.2)
(2024, 'Delhi', '01/01/2024', 15.5)
(2024, 'Delhi', '02/01/2024', 16.0)
(2024, 'Mumbai', '01/01/2024', 24.0)
(2025, 'Chennai', '10/03/2025', 29.0)
```

```
TIME COMPLEXITY:
Insert: O(R + C) worst case (search + resize), typically O(1) for appending to cell list
Delete: O(R + C + M) where M = records in cell
Retrieve: O(R + C + M)
Row/Col Major Access: O(R * C)

SPACE COMPLEXITY:
Matrix: O(R * C) + Records: O(K)
```

COMPLEXITY ANALYSIS

Let R = number of distinct years, C = number of distinct cities, K = total number of stored records, and M = number of records in a particular matrix cell.

Insert-

- **Worst case: O(R + C) if a new year/city forces index searches and matrix resizing. Typical insertion into an already-existing cell is O(1) (append to list).**

Delete (by date & city)-

- **O(R + C + M) — locating the cell plus scanning/removing items from that cell; also we scan RECORDS list to remove matching entries.**

Retrieve (city & year)-

- **O(R + C + M) — index lookup plus reading items in the cell.**

Row-major / Column-major traversal-

- **O(R * C) — visits every cell once (plus any cost to print records inside cells).**

Space-

- **O(R * C) for the matrix references/slots, plus O(K) storage for all record objects.**

This analysis is aligned with the implemented approach and is suitable for the assignment report.

CONCLUSION

This project demonstrates a practical application of 2D arrays and simple ADTs to model weather data. The implementation highlights key data-structure concepts:

- **representing tabular data with arrays,**
- **how traversal order (row-major vs column-major) affects iteration,**
- **methods to manage sparse datasets,**
- **and how to reason about time and space complexity for basic operations.**

Overall, the system is intentionally simple, easy to explain, and structured so that each required assignment component is independently demonstrable.