A blue and white sign with black text

Description automatically generated

**School of Engineering & Technology**

**Report**

**Data Structure Lab**

**ENCS253**

**Assignment 1**

**Submitted By**  **Submitted To**

Subhasmita Mishra Dr. Vandana Batra

2401360011

B.Tech CSE(UX/UI)

Semester 3

**Index**

* Description of the Inventory Item ADT.
* Strategy for implementing arrays, sparse representations, and operations in C++ or Python.
* Approach to complexity analysis for each function.
* Analysis of the system’s efficiency and functionality.

**Description of the Inventory Item ADT**

The Inventory Item Abstract Data Type (ADT) is designed to encapsulate the properties and behaviour of an item in an inventory system.

* Attributes:
  1. itemID (int) – Unique identifier for each item.
  2. itemName (string) – Name of the item.
  3. quantity (int) – Number of units available.
  4. price (float) – Cost per unit of the item.
* Operations / Methods:
  1. Constructor – Initializes an item with given itemID, itemName, quantity, and price.
  2. display() – Prints the item’s details in a human-readable format.

This ADT provides a structured way to manage inventory items while keeping data encapsulated in objects.

**Strategy for Implementing Arrays, Sparse Representations, and Operations**

**1. Arrays**

* **Purpose**: Store all inventory items and their attributes in a structured format.
* **Implementation**:
  + InventoryItem items[100] – Array of objects to hold inventory items.
  + float priceQuantityTable[100][2] – 2D array storing price and quantity for each item for row- and column-wise operations.
* **Operations**:
  + Adding, deleting, or searching items involves traversing the array.
  + Row-major order iterates item-wise; column-major order iterates attribute-wise.

**2. Sparse Representation**

* **Purpose**: Optimize storage by storing only items with **non-zero quantities**.
* **Implementation**:
  + int sparseMatrix[100][2] – Stores only itemID and quantity of non-zero items.
* **Operations**:
  + During addition, only non-zero quantity items are stored in sparseMatrix.
  + During display, the sparse matrix is iterated to show relevant items.

**3. Operations Implemented**

| **Operation** | **Description** |
| --- | --- |
| addItemRecord() | Adds a new item to the inventory, updates both arrays and sparse representation. |
| removeItemRecord() | Deletes an item by ID; shifts remaining items to maintain array integrity. |
| searchByItem() | Searches for an item using ID or name via linear search. |
| managePriceQuantity() | Displays price and quantity using row-major and column-major formats. |
| optimizeSparseStorage() | Displays items with non-zero quantities using sparse matrix. |

**Approach to Complexity Analysis**

1. addItemRecord()

* Time Complexity: O(1) – Appends at the end of arrays; no traversal needed.
* Space Complexity: O(1) per item added.

2. removeItemRecord()

* Time Complexity: O(n) – In worst-case, all elements are shifted.
* Space Complexity: O(1) – In-place shifting; no extra memory needed.

3. searchByItem()

* Time Complexity: O(n) – Linear search through all items.
* Space Complexity: O(1) – Only temporary variables used.

4. managePriceQuantity()

* Time Complexity: O(n) – Iterates through all items for row- and column-wise display.
* Space Complexity: O(1) – Uses existing arrays; no additional storage.

5. optimizeSparseStorage()

* Time Complexity: O(n) – Each item in sparseMatrix checked once.
* Space Complexity: O(k) – k is number of non-zero quantity items.

**Analysis of the System’s Efficiency and Functionality**

**Efficiency**

* **Insertion** is highly efficient for up to 100 items since it is O(1).
* **Deletion and search** are linear; performance may degrade for larger datasets, but acceptable for small inventory sizes.
* **Sparse representation** reduces unnecessary storage and improves performance for reporting non-zero quantity items.

**Functionality**

* The system supports:
  1. Adding and deleting items dynamically.
  2. Searching by ID or name.
  3. Displaying price and quantity in multiple formats.
  4. Optimized display using sparse storage.
  5. Menu-driven interface for ease of use.

**Limitations**

* Maximum of 100 items (fixed-size arrays).
* Linear search may be slow for large inventories.
* Sparse matrix is simple; advanced data structures (hash maps) could improve search efficiency.

**Conclusion**

The Inventory Management System efficiently demonstrates the use of ADTs, arrays, and sparse representations in C++. It balances simplicity with functionality and provides a clear framework for inventory operations with measurable time and space complexity.