Que.1) 20 friends put their wallets in a row. The first wallet contains 20 dollars, the second has 30 dollars, the third has 40 dollars, and so on, with each wallet having 10 dollars more than the previous one. Since the data is already sorted in ascending order, no sorting is required. But if you are given a chance to sort the wallets, which sorting technique would be best to apply? Write a C++ program to implement your chosen sorting approach.

Ans: Aim:

To implement **Insertion Sort** on a list of 20 wallets, where each wallet contains increasing amounts of money, and determine the most efficient sorting technique when the data is already sorted.

Objective:

To understand and implement the **Insertion Sort** algorithm.

To analyze the efficiency of Insertion Sort when applied to an already sorted array.

To simulate a real-life example where 20 friends keep increasing amounts of money in their wallets.

To verify that the sorting maintains the order of already sorted data.

Procedure:

Create an integer array to represent 20 wallets.

Initialize each wallet with increasing values starting from $20, increasing by $10 each time.

Implement the **Insertion Sort** algorithm to sort the array.

Display the wallet values **before and after** sorting.

Since the array is already sorted, Insertion Sort should run in **linear time O(n)**.

Q 2: In a park, 10 friends were discussing a game based on sorting. They placed their wallets in a row. The maximum money in any wallet is **$6**. Among them, **3 wallets contain exactly $2**, **2 wallets contain exactly $3**, **2 wallets are empty ($0)**, **1 wallet contains $1**, and **1 wallet contains $4**. Which sorting technique would you apply to sort the wallets on the basis of the money they contain? Write a program to implement your chosen sorting technique.

Aim

To sort the wallets of 10 friends based on the amount of money they contain (ranging from $0 to $6) using an efficient algorithm suited for small-range integer data.

Objective:

To analyze the problem where the data consists of small non-negative integers (wallet amounts from $0 to $6).

To choose and apply the most efficient sorting technique for such data.

To implement **Counting Sort**, which is ideal when:

1. The data size is small.
2. The range of values is known and limited.

To sort the wallet values in ascending order.

Procedure:

Analyze the given wallet values:

1.Total: 10 wallets.

2.Maximum value: $6.

3.Values: {0, 0, 1, 2, 2, 2, 3, 3, 4}

Since the values are small integers and repeat multiple times, **Counting Sort** is the most efficient choice.

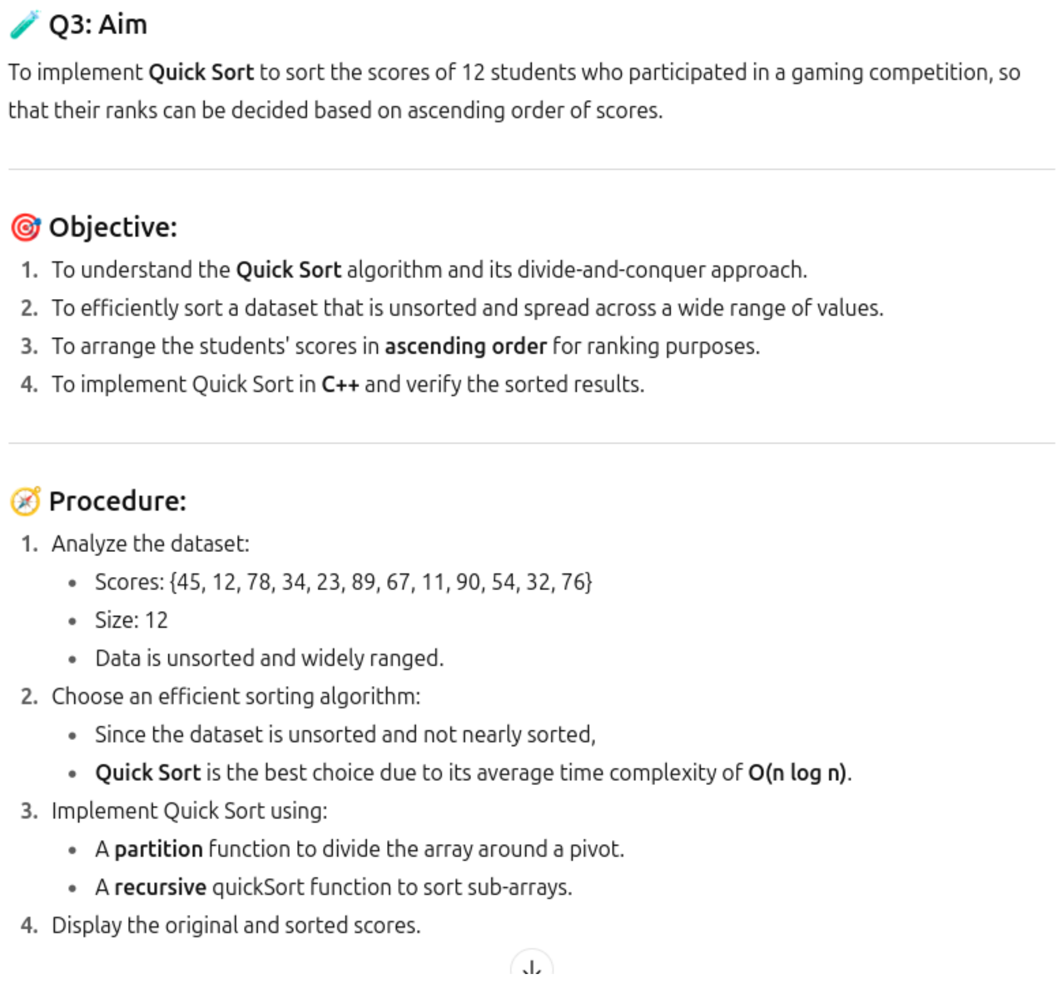
Initialize a **count array** of size 7 (0 to 6).

Count the frequency of each wallet value.

Reconstruct the sorted array using the count array.

Print the sorted wallet values.

Q 3: During a college fest, 12 students participated in a gaming competition. Each student's score was recorded as follows: 45, 12, 78, 34, 23, 89, 67, 11, 90, 54, 32, 76 The organizers want to arrange the scores in **ascending order** to decide the ranking of the players. Since the data set is unsorted and contains numbers spread across a wide range, the most efficient technique to apply here is **Quick Sort**. Write a C++ program to implement **Quick Sort** to arrange the scores in ascending order.



🧪 Q3: Aim

To implement **Quick Sort** to sort the scores of 12 students who participated in a gaming competition, so that their ranks can be decided based on ascending order of scores.

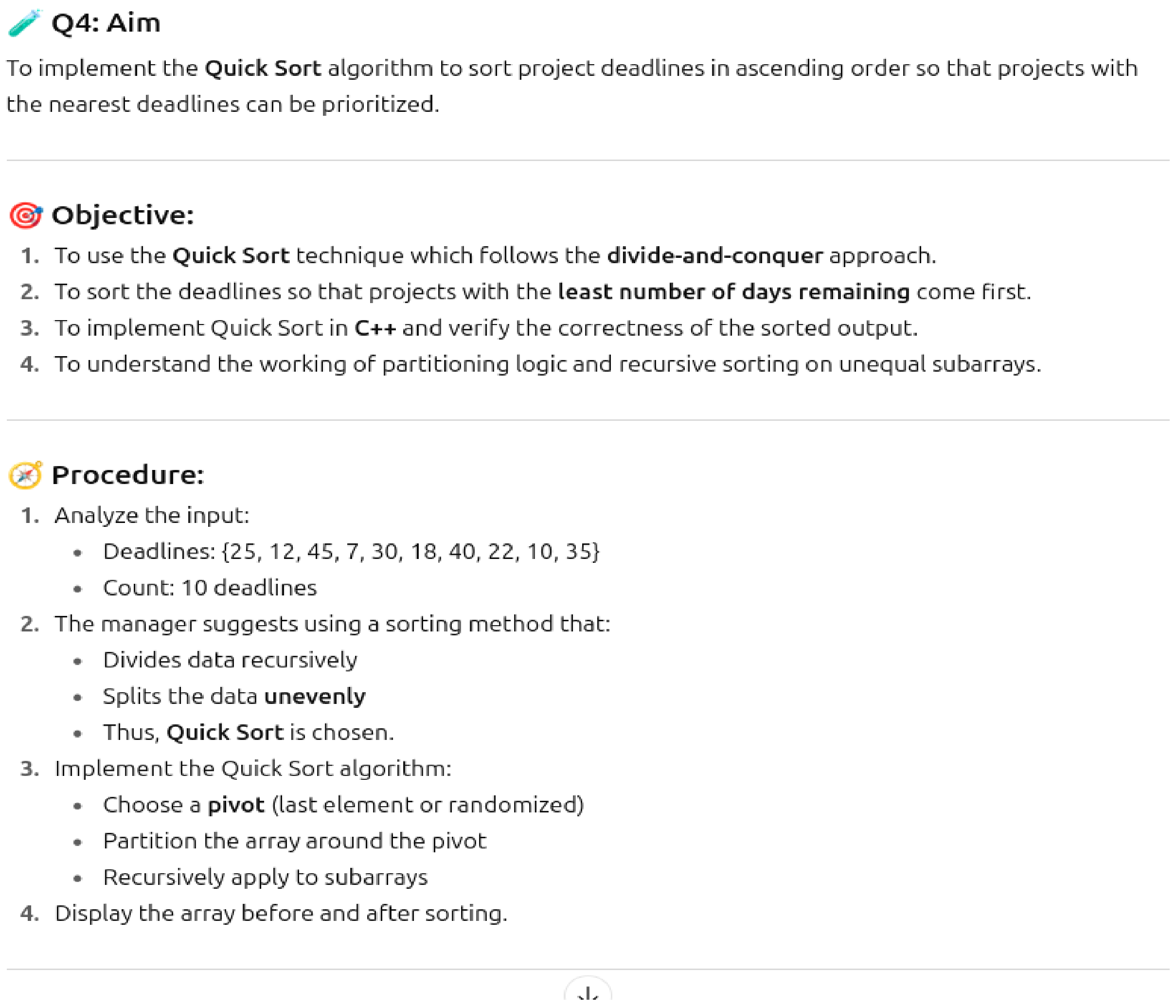
🎯 Objective:

1. To understand the **Quick Sort** algorithm and its divide-and-conquer approach.
2. To efficiently sort a dataset that is unsorted and spread across a wide range of values.
3. To arrange the students' scores in **ascending order** for ranking purposes.
4. To implement Quick Sort in **C++** and verify the sorted results.

🧭 Procedure:

1. Analyze the dataset:
    - Scores: {45, 12, 78, 34, 23, 89, 67, 11, 90, 54, 32, 76}
    - Size: 12
    - Data is unsorted and widely ranged.
2. Choose an efficient sorting algorithm:
    - Since the dataset is unsorted and not nearly sorted,
    - **Quick Sort** is the best choice due to its average time complexity of **O(n log n)**.
3. Implement Quick Sort using:
    - A **partition** function to divide the array around a pivot.
    - A **recursive** quickSort function to sort sub-arrays.
4. Display the original and sorted scores.

Q 4: A software company is tracking project deadlines (in days remaining to submit). The deadlines are: 25, 12, 45, 7, 30, 18, 40, 22, 10, 35. The manager wants to arrange the deadlines in **ascending order** to prioritize the projects with the least remaining time. For efficiency, the project manager hints to the team to apply a **divide-and-conquer technique that divides the array into unequal parts**. Write a C++ program to sort the project deadlines using the above sorting technique.



🧪 Q4: Aim

To implement the **Quick Sort** algorithm to sort project deadlines in ascending order so that projects with the nearest deadlines can be prioritized.

🎯 Objective:

1. To use the **Quick Sort** technique which follows the **divide-and-conquer** approach.
2. To sort the deadlines so that projects with the **least number of days remaining** come first.
3. To implement Quick Sort in **C++** and verify the correctness of the sorted output.
4. To understand the working of partitioning logic and recursive sorting on unequal subarrays.

🧭 Procedure:

1. Analyze the input:
   - Deadlines: {25, 12, 45, 7, 30, 18, 40, 22, 10, 35}
   - Count: 10 deadlines
2. The manager suggests using a sorting method that:
   - Divides data recursively
   - Splits the data **unevenly**
   - Thus, **Quick Sort** is chosen.
3. Implement the Quick Sort algorithm:
   - Choose a **pivot** (last element or randomized)
   - Partition the array around the pivot
   - Recursively apply to subarrays
4. Display the array before and after sorting.

Q 5: Suppose there is a square named **SQ-1**. By connecting the midpoints of SQ-1, we create another square named **SQ-2**. Repeating this process, we create a total of **50 squares** {SQ-1, SQ-2, …, SQ-50}. The areas of these squares are stored in an array. Your task is to **search whether a given area is present in the array or not**.What would be the best searching approach? Write a C++ program to implement this approach.

🧪 Q5: Aim

To search for a specific square area among 50 squares generated by recursively connecting midpoints of a square using the **most efficient searching technique**.

🎯 Objective:

1. Understand how square areas decrease when new squares are formed by connecting midpoints.
2. Generate areas of 50 such squares and store them in an array.
3. Apply an efficient **searching algorithm** to check if a given area exists in the array.
4. Implement **Binary Search** as the best choice for sorted data.

🧭 Procedure:

1. Define the initial square area `A₁` (e.g., 1024).
2. Generate 50 areas where each area is half of the previous one.
3. Store these areas in a sorted array (descending order).
4. Accept a target area from the user.
5. Apply **Binary Search** on the array to check if the area exists.
6. Display appropriate message.

Q 6: Before a match, the chief guest wants to meet all the players. The head coach introduces the first player, then that player introduces the next player, and so on, until all players are introduced. The chief guest moves forward with each introduction, meeting the players one at a time. How would you implement the above activity using a Linked List? Write a C++ program to implement the logic.

🧪 Q6: Aim

To implement a **singly linked list** in C++ that simulates a chain of player introductions, where each player introduces the next one to the chief guest.

🎯 Objective:

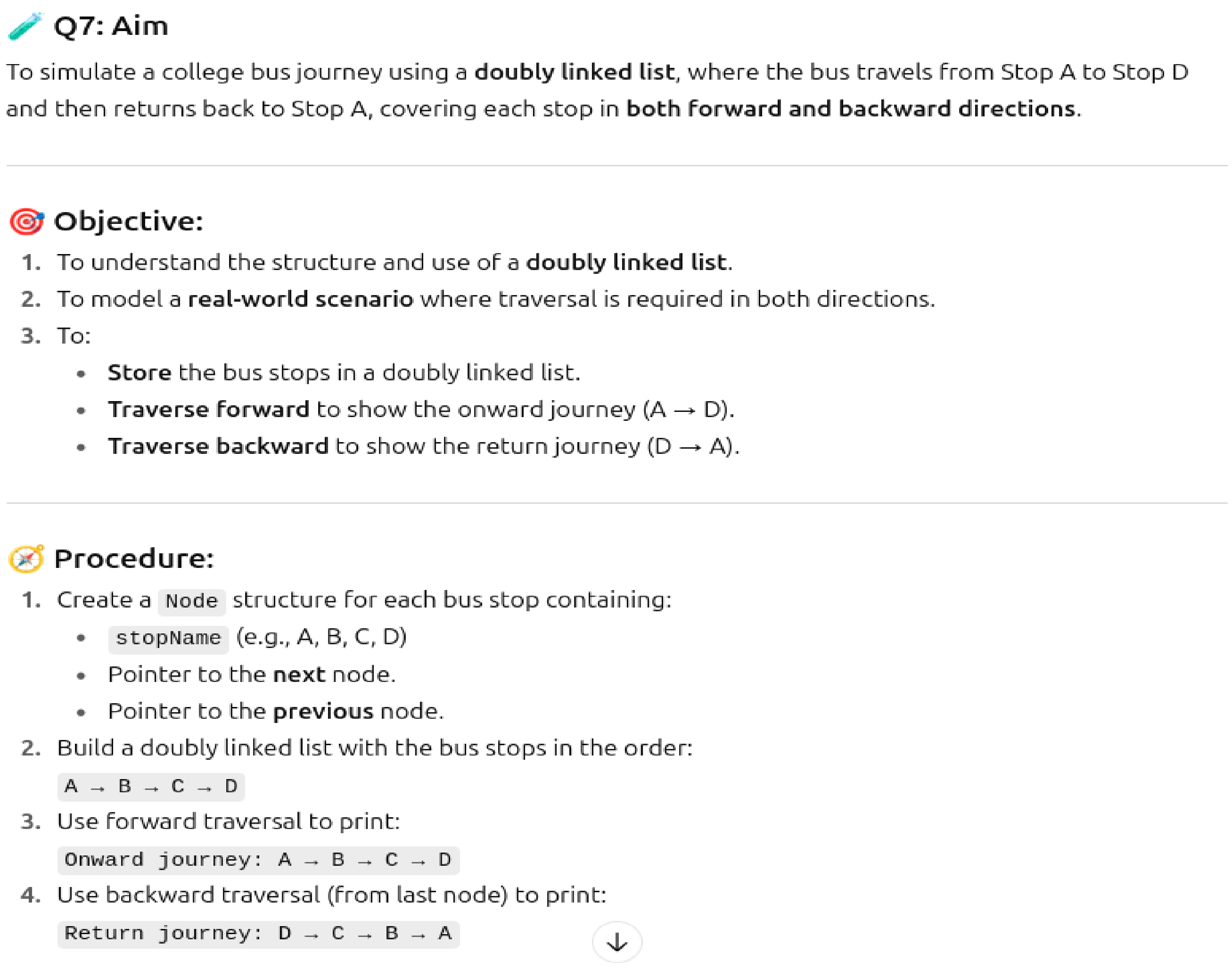
1. To understand and implement the **Linked List** data structure.
2. To simulate real-world behavior where:
   - The coach introduces the **first player**.
   - Each player **introduces the next one**.
   - The **chief guest** meets players in sequence.
3. To display the player introductions using **linked list traversal**.

🧭 Procedure:

1. Define a `Node` structure representing each player.
2. Each node contains:
   - The **player's name**.
   - A pointer to the **next player**.
3. Insert players into the linked list in the given order.
4. Traverse the linked list from the **head (first player)** and display introductions.
5. Simulate the chief guest meeting each player one by one.

Q 7: A college bus travels from stop A → stop B → stop C → stop D and then returns in reverse order D → C → B → A. Model this journey using a **doubly linked list**. Write a program to:

- Store bus stops in a doubly linked list.
- Traverse forward to show the onward journey.
- Traverse backward to show the return journey.



🧪 Q7: Aim

To simulate a college bus journey using a **doubly linked list**, where the bus travels from Stop A to Stop D and then returns back to Stop A, covering each stop in **both forward and backward directions**.

🎯 Objective:

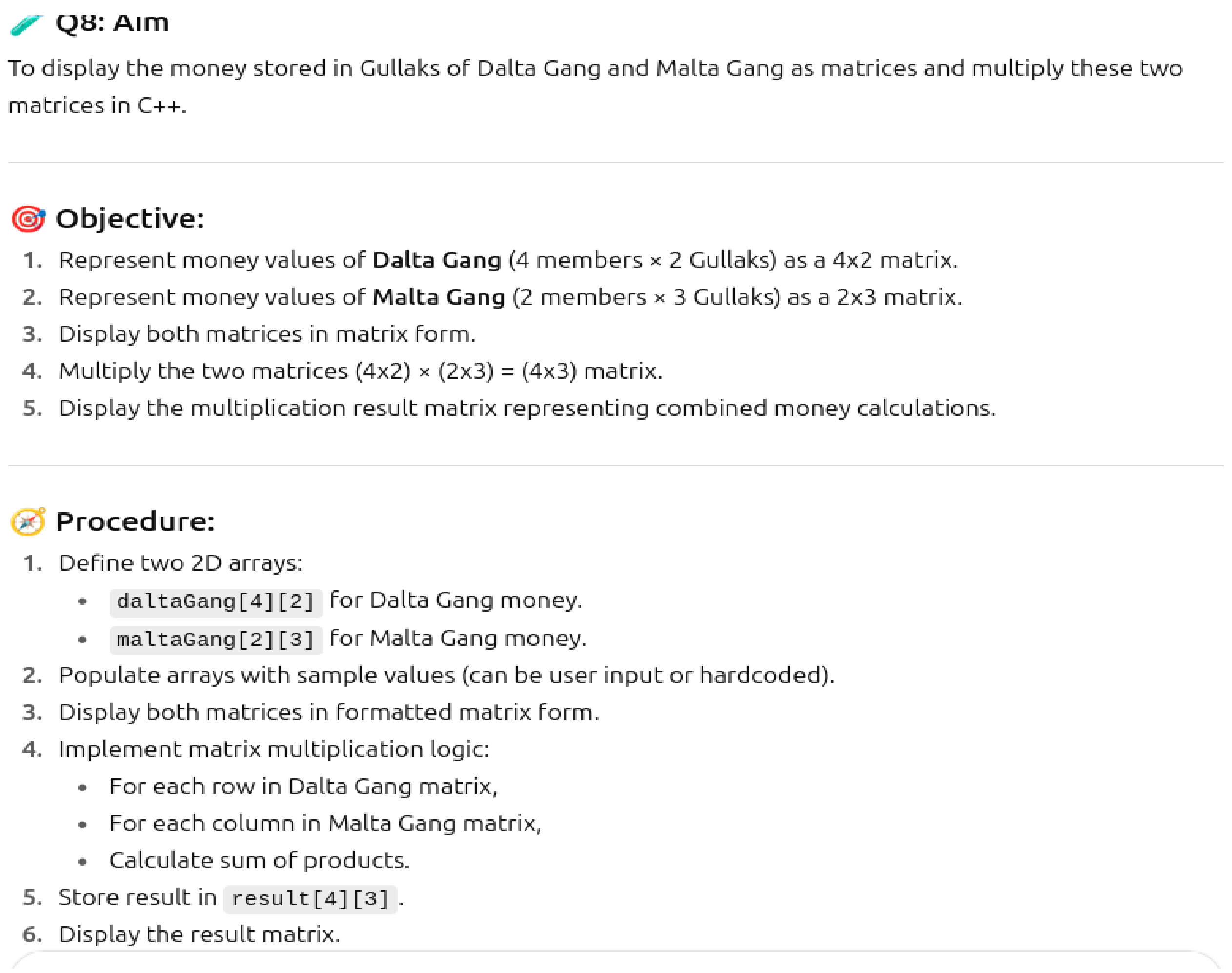
1. To understand the structure and use of a **doubly linked list**.
2. To model a **real-world scenario** where traversal is required in both directions.
3. To:
   - **Store** the bus stops in a doubly linked list.
   - **Traverse forward** to show the onward journey (A → D).
   - **Traverse backward** to show the return journey (D → A).

🧭 Procedure:

1. Create a `Node` structure for each bus stop containing:
   - `stopName` (e.g., A, B, C, D)
   - Pointer to the **next** node.
   - Pointer to the **previous** node.
2. Build a doubly linked list with the bus stops in the order:
   `A → B → C → D`
3. Use forward traversal to print:
   `Onward journey: A → B → C → D`
4. Use backward traversal (from last node) to print:
   `Return journey: D → C → B → A`

Q 8: There are two teams named Dalta Gang and Malta Gang. Dalta Gang has 4 members, and each member has 2 Gullaks (piggy banks) with some money stored in them. Malta Gang has 2 members, and each member has 3 Gullaks. Both gangs store their Gullak money values in a 2D array. Write a C++ program to:

- Display the stored data in matrix form.
- To multiply Dalta Gang matrix with Malta Gang Matrix



Q8: Aim

To display the money stored in Gullaks of Dalta Gang and Malta Gang as matrices and multiply these two matrices in C++.

Objective:

1. Represent money values of **Dalta Gang** (4 members × 2 Gullaks) as a 4x2 matrix.
2. Represent money values of **Malta Gang** (2 members × 3 Gullaks) as a 2x3 matrix.
3. Display both matrices in matrix form.
4. Multiply the two matrices (4x2) × (2x3) = (4x3) matrix.
5. Display the multiplication result matrix representing combined money calculations.

Procedure:

1. Define two 2D arrays:
   - `daltaGang[4][2]` for Dalta Gang money.
   - `maltaGang[2][3]` for Malta Gang money.
2. Populate arrays with sample values (can be user input or hardcoded).
3. Display both matrices in formatted matrix form.
4. Implement matrix multiplication logic:
   - For each row in Dalta Gang matrix,
   - For each column in Malta Gang matrix,
   - Calculate sum of products.
5. Store result in `result[4][3]`.
6. Display the result matrix.

SECTION -B

Q 1: To store the names of family members, an expert suggests organizing the data in a way that allows efficient searching, traversal, and insertion of new members. For this purpose, use a **Binary Search Tree (BST)** to store the names of family members, starting with the letters:

<Q, S, R, T, M, A, B, P, N>

Write a C++ program to Create a **Binary Search Tree (BST)** using the given names and find and display the **successor** of the family member whose name starts with **M**.

🧪 Q1: Aim

To implement a **Binary Search Tree (BST)** that stores names of family members and efficiently find the **successor** of a given member (here, the member whose name starts with 'M').

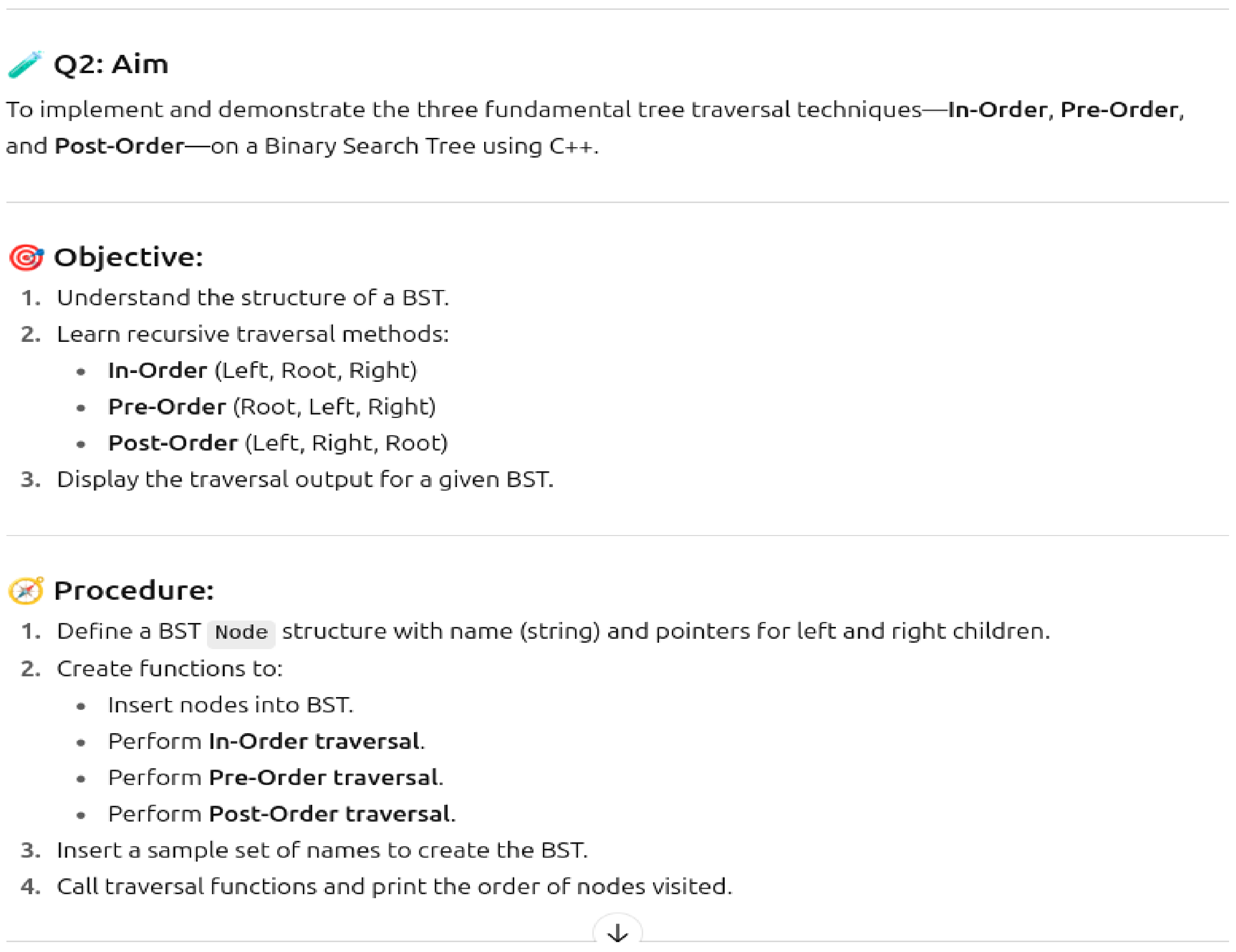
🎯 Objective:

1. Insert given names (`Q, S, R, T, M, A, B, P, N`) into a BST.
2. Traverse the BST to display names (optional for verification).
3. Find the **in-order successor** of the node with name starting with `'M'`.
4. Display the successor's name.

🧭 Procedure:

1. Define a `Node` structure for BST nodes containing:
   - Name (string)
   - Left child pointer
   - Right child pointer
2. Insert the given names into the BST following BST rules:
   - Left subtree nodes have names lexicographically smaller.
   - Right subtree nodes have names lexicographically greater.
3. Search for the node whose name starts with `'M'`.
4. Find the **successor**:
   - If node has a right subtree, successor is the minimum value node in the right subtree.
   - Otherwise, successor is the lowest ancestor whose left child is also ancestor of the node.
5. Print the successor's name.

Q 2: Implement the In-Order, Pre- Order and Post-Order traversal of Binary search tree with help of C++ Program.



🧪 Q2: Aim

To implement and demonstrate the three fundamental tree traversal techniques—**In-Order**, **Pre-Order**, and **Post-Order**—on a Binary Search Tree using C++.

🎯 Objective:

1. Understand the structure of a BST.
2. Learn recursive traversal methods:
   - **In-Order** (Left, Root, Right)
   - **Pre-Order** (Root, Left, Right)
   - **Post-Order** (Left, Right, Root)
3. Display the traversal output for a given BST.

🧭 Procedure:

1. Define a BST `Node` structure with name (string) and pointers for left and right children.
2. Create functions to:
   - Insert nodes into BST.
   - Perform **In-Order traversal**.
   - Perform **Pre-Order traversal**.
   - Perform **Post-Order traversal**.
3. Insert a sample set of names to create the BST.
4. Call traversal functions and print the order of nodes visited.

Q 3: Write a C++ program to search an element in a given binary search Tree.

🧪 Q3: Aim

To write a C++ program that efficiently searches for a given element (string or integer) in a Binary Search Tree.

🎯 Objective:

1. Understand the BST property for searching.
2. Implement a recursive function to search an element in BST.
3. Indicate whether the element is found or not.

🧭 Procedure:

1. Define a `Node` structure representing a BST node.
2. Insert a few elements into the BST.
3. Implement a `searchBST` function:
   - If root is `nullptr`, element not found.
   - If root's value matches the key, return true.
   - If key is less than root's value, search in left subtree.
   - Else, search in right subtree.
4. Call the search function with the element to find.
5. Print the result.

Q 4: In a university, the roll numbers of newly admitted students are: 45, 12, 78, 34, 23, 89, 67, 11, 90, 54

The administration wants to store these roll numbers in a way that allows **fast searching, insertion, and retrieval in ascending order**. For efficiency, they decide to apply a **Binary Search Tree (BST)**.

Write a C++ program to construct a **Binary Search Tree** using the above roll numbers and perform an **in-order traversal** to display them in ascending order.

🧪 Q4: Aim

To create a Binary Search Tree (BST) from the given student roll numbers and perform an in-order traversal to display the roll numbers in ascending order.

🎯 Objective:

1. Insert roll numbers into BST following BST properties.
2. Use in-order traversal to display the stored roll numbers in ascending order.
3. Demonstrate efficient search, insertion, and retrieval capabilities of BST.

🧭 Procedure:

1. Define a `Node` structure containing the roll number and left/right child pointers.
2. Insert each roll number from the list into the BST.
3. Implement an **in-order traversal** to visit nodes in ascending order.
4. Print the visited roll numbers during traversal.

Q 5: In a university database, student roll numbers are stored using a **Binary Search Tree (BST)** to allow efficient searching, insertion, and deletion. The roll numbers are: 50, 30, 70, 20, 40, 60, 80. The administrator now wants to **delete a student record** from the BST. Write a C++ program to delete a node (student roll number) entered by the user.

🧪 Q5: Aim

To write a C++ program that deletes a specified student roll number node from a Binary Search Tree (BST) and maintains the BST properties after deletion.

🎯 Objective:

1. Construct BST from given roll numbers.
2. Implement deletion of a node in BST considering all cases:
    - Node with no children (leaf node).
    - Node with one child.
    - Node with two children.
3. Verify the deletion by performing an in-order traversal after deletion.

🧭 Procedure:

1. Define a `Node` structure with roll number and pointers to left and right child.
2. Insert the given roll numbers into the BST.
3. Implement a function `deleteNode` that:
    - Finds the node to delete.
    - Handles three deletion cases:
        - Leaf node: delete directly.
        - Node with one child: replace node with its child.
        - Node with two children: replace node value with inorder successor (minimum value in right subtree) and delete successor node.
4. Use in-order traversal to display BST after deletion.
5. Take input for the roll number to delete from the user.