PROBLEMS AND SOLUTIONS — PART 4

Chapter 1: Introduction to Data Structure

More Important Exam-Oriented Questions and Analytical Problems with Step-by-Step Solutions

This is a fourth set of frequently asked exam questions and analytical problems on Data Representation, Abstract Data Types, Data Structures, and related concepts, covering problem types not included in Parts 1 to 3 — including circular linked lists, prefix evaluation, tree traversal, binary fractions, structure padding, and abstraction/encapsulation concepts. Practice all four parts together for complete exam coverage of this chapter.

Section A: Tracing More Data Structure Applications

Problem 1: Starting from an empty Circular Linked List, insert 10, 20, and 30 at the end, in that order. Show the structure of the list, and explain what makes it different from a normal Singly Linked List during traversal.

Solution:

In a Circular Linked List, the 'next' pointer of the LAST node does not point to NULL (as in a normal Singly Linked List); instead, it points back to the FIRST node, forming a closed loop.

- Insert 10 → List: 10 → (points back to 10)
- Insert 20 at end → List: 10 → 20 → (points back to 10)
- Insert 30 at end → List: 10 → 20 → 30 → (points back to 10)

```
Final structure: 10 → 20 → 30 → (back to 10) → 20 → 30 → ... (loops forever)
```

This means that, unlike a normal Singly Linked List (where traversal naturally stops at NULL), traversing a Circular Linked List requires an explicit stopping condition — such as stopping when the traversal pointer returns to the starting node again — otherwise the traversal would loop indefinitely.

Problem 2: Evaluate the prefix expression * + 2 3 4 using a Stack. Show the stack contents at each step and give the final result.

Solution:

Rule for prefix evaluation: scan the expression from RIGHT to LEFT. Push every operand onto the stack. When an operator is encountered, pop the top two operands (the first-popped is the left operand this time, since we are scanning right to left), apply the operator, and push the result back.

- Scanning right to left: Read '4' (operand) → push → Stack: [4]
- Read '3' (operand) → push → Stack: [4, 3]
- Read '2' (operand) → push → Stack: [4, 3, 2]
- Read '+' → pop 2 and 3 → compute 2 + 3 = 5 → push 5 → Stack: [4, 5]

- Read '*' → pop 5 and 4 → compute 4 * 5 = 20 → push 20 → Stack: [20]

```
Final Result = 20
```

Problem 3: Use a Stack to reverse the string "HELLO". Show the stack contents while pushing all characters, and while popping them to form the reversed string.

Solution:

Rule: Push every character of the string onto the stack in order. Then pop characters one by one — since a Stack is LIFO, the characters come out in exactly the reverse order they were pushed.

- Push 'H' → Stack: [H]
- Push 'E' → Stack: [H, E]
- Push 'L' → Stack: [H, E, L]
- Push 'L' → Stack: [H, E, L, L]
- Push 'O' → Stack: [H, E, L, L, O]
- Pop repeatedly: O, L, L, E, H → forms the string "OLLEH"

```
Reversed String = "OLLEH"
```

Problem 4: A Binary Tree has root node A. A's left child is B and right child is C. B's left child is D and right child is E (D and E have no children). Find the Preorder, Inorder, and Postorder traversal sequences of this tree.

Solution:

Preorder traversal visits nodes in the order: Root, Left subtree, Right subtree. Inorder traversal visits: Left subtree, Root, Right subtree. Postorder traversal visits: Left subtree, Right subtree, Root — and these rules are applied recursively at every node.

The tree structure is: A is root; A's left = B, A's right = C; B's left = D, B's right = E; C, D, and E are leaf nodes (no children).

- Preorder (Root, Left, Right): Visit A → go left to B's subtree: Visit B → go left to D (leaf): Visit D → go right to E (leaf): Visit E → back up, go right to C (leaf): Visit C. Sequence: A, B, D, E, C.
- Inorder (Left, Root, Right): D's subtree first: Visit D → Visit B → Visit E → back up to A: Visit A → Visit C. Sequence: D, B, E, A, C.
- Postorder (Left, Right, Root): D's subtree: Visit D → Visit E → Visit B → Visit C → Visit A (root last). Sequence: D, E, B, C, A.

```
Preorder = A B D E C;   Inorder = D B E A C;   Postorder = D E B C A
```

Section B: Differentiating Concepts

Problem 5: Differentiate between Singly Linked List and Circular Linked List.

Solution:

Basis	Singly Linked List	Circular Linked List
Last node's pointer	Points to NULL	Points back to the first (head) node
Traversal end condition	Stops automatically when NULL is reached	Must explicitly check for return to the starting node to avoid an infinite loop

Suitable applications	General-purpose sequential storage	Round-robin scheduling, repeating playlists, and other cyclic processes

---

Problem 6: Differentiate between a Tree and a Graph, even though both are Non-Linear Data Structures.

Solution:

Basis	**Tree**	**Graph**
Structure	Hierarchical, with exactly one root node and no cycles allowed	A general network of nodes (vertices) and connections (edges); cycles are allowed
Path between nodes	Exactly one unique path exists between any two nodes	Multiple paths may exist between two nodes, or none at all
Edges	Exactly (n − 1) edges for n nodes	Can have any number of edges, from 0 up to n(n−1)/2 (for an undirected graph)

In fact, every Tree is technically a special, restricted case of a Graph (one that is connected and has no cycles), but not every Graph qualifies as a Tree.

Problem 7: Differentiate between Homogeneous and Heterogeneous Data Structures, using Array and Structure as examples.

Solution:

Basis	**Homogeneous Data Structure**	**Heterogeneous Data Structure**
Element types	All elements must be of the SAME data type	Elements (members) can be of DIFFERENT data types
Example	Array (e.g., an array of integers)	Structure (e.g., a Student record with a name string, an integer roll number, and a float marks value)

Section C: Data Representation and Memory Calculation

Problem 8: A 3D array A[5][6][7] (dimensions: 5 × 6 × 7) is stored in Row-Major order, starting at base address 5000, with each element occupying 4 bytes. Find the memory address of the element A[2][3][4].

Solution:

For a 3D array with dimensions [D1][D2][D3], the Row-Major address formula extends the 2D formula: Address(A[i][j][k]) = Base + (i × D2 × D3 + j × D3 + k) × size_of_element.

Here, Base = 5000, D2 = 6, D3 = 7, i = 2, j = 3, k = 4, size_of_element = 4 bytes.

Address = 5000 + (2 × 6 × 7 + 3 × 7 + 4) × 4 = 5000 + (84 + 21 + 4) × 4 = 5000 + 109 × 4 = 5000 + 436 = 5436.

```
Address of A[2][3][4] = 5436
```

Problem 9: Convert the decimal number 10.75 into its binary equivalent.

Solution:

Convert the integer part and the fractional part separately, then combine them.

Integer part (10): 10 ÷ 2 = 5 remainder 0; 5 ÷ 2 = 2 remainder 1; 2 ÷ 2 = 1 remainder 0; 1 ÷ 2 = 0 remainder 1. Reading remainders bottom to top: 1010.

Fractional part (0.75): multiply by 2 repeatedly and record the integer part obtained each time. 0.75 × 2 = 1.50 → record 1, carry forward 0.50; 0.50 × 2 = 1.00 → record 1, carry forward 0.00 (stop, since the fractional part is now 0).

Reading the recorded bits top to bottom for the fractional part gives: .11

$(10.75)_{10} = (1010.11)_2$

Problem 10:

Given the following structure, calculate its actual size in memory, taking into account memory alignment (padding), on a system where an int must be aligned to a 4-byte boundary (char = 1 byte, int = 4 bytes):

```
struct Example {
    char c;
    int i;
};
```

Solution:

A naive sum of member sizes would give 1 (char) + 4 (int) = 5 bytes. However, real compilers add padding bytes so that each member is stored at a memory address that is a multiple of its own size (its 'alignment requirement'), which improves memory access speed on most hardware.

Here, c occupies byte 0. But i (an int) requires 4-byte alignment, meaning it must start at an address that is a multiple of 4. Since byte 1 is not a multiple of 4, the compiler inserts 3 padding bytes (bytes 1, 2, 3) before placing i starting at byte 4.

So the layout becomes: c (byte 0), padding (bytes 1–3), i (bytes 4–7). Total size = 1 + 3 (padding) + 4 = 8 bytes.

```
sizeof(struct Example) = 8 bytes (not 5 bytes, due to padding/alignment)
```

This demonstrates why the actual sizeof() result for a structure can be larger than the simple sum of its member sizes, and why the ORDER in which members are declared can also affect the total size of a structure.

Section D: ADT Design and Conceptual Understanding

Problem 11: Write only the Abstract Data Type (ADT) specification (the set of operations, without implementation details) for a simple Bank Account system that supports depositing money, withdrawing money, checking the current balance, and transferring money to another account.

Solution:

- deposit(amount) — increases the account balance by the given amount.
- withdraw(amount) — decreases the account balance by the given amount, only if sufficient balance is available.
- checkBalance() — returns the current balance of the account.
- transfer(amount, targetAccount) — withdraws the given amount from this account and deposits it into the target account, only if sufficient balance is available.

This ADT specification hides how the balance is actually stored or updated internally (e.g., in a simple variable, a database record, or a more complex ledger structure) — only the available operations and their effects are defined.

Problem 12: The List ADT operation 'insert at the beginning' is supported by both an Array-based implementation and a Linked-List-based implementation. Compare the time complexity of this single operation across the two implementations, and explain the reason for the difference.

Solution:

Array-based implementation: Inserting a new element at the beginning requires shifting EVERY existing element one position to the right, to make room at index 0. For n existing elements, this requires n shift operations, giving a time complexity of O(n).

Linked-List-based implementation: Inserting a new element at the beginning only requires creating a new node and updating a constant number of pointers — the new node's 'next' pointer is set to the current head, and the head pointer is updated to the new node. No existing nodes need to move. This takes O(1) time, regardless of how many elements are already in the list.

This difference arises because an Array stores elements in contiguous memory (so maintaining order after insertion requires physically moving data), while a Linked List stores elements as independently allocated nodes connected by pointers (so reordering only requires updating references, not moving data).

This is a clear example of how the SAME ADT operation (insert at beginning of a List) can have very different efficiency depending on the underlying Data Structure chosen to implement it.

Problem 13: Explain the concepts of Abstraction and Encapsulation as they apply to the Bank Account ADT described in Problem 11.

Solution:

Abstraction means presenting only the essential features (WHAT the Bank Account can do) to the user, while hiding the unnecessary internal complexity (HOW it is done). A user of the Bank Account ADT only needs to know that deposit(), withdraw(), checkBalance(), and transfer() exist and what they do — not how the balance is stored, validated, or updated internally.

Encapsulation means bundling the data (the account balance, account number, etc.) together with the operations that act on that data (deposit, withdraw, etc.) into a single unit, while restricting direct outside access to the internal data. For example, no outside code should be able to directly set the balance variable to an arbitrary value — it can only be changed through the defined operations (deposit and withdraw), which can enforce rules such as 'balance cannot go negative'.

Together, Abstraction and Encapsulation are what make the ADT concept powerful: they allow the internal implementation of the Bank Account to be changed later (e.g., switching from a simple variable to a more complex transaction-logging system) without affecting any code that uses the ADT's defined operations.

Section E: Applied Refinement and Conceptual Review

Problem 14: A developer wants to build a Simple Calculator application that accepts two numbers and an operator (+, −, ×, or ÷) from the user and displays the result. Walk through the stages of refinement to arrive at an implementation.

Solution:

1. Problem Definition: Given two numbers and an operator symbol, compute and display the result of applying that operator to the two numbers.

2. ADT Specification: Define a Calculator ADT with a single operation, compute(num1, operator, num2), which returns the result, without deciding yet how each operator will be internally handled.
3. Data Structure Selection: Since only two numbers and one operator are involved at a time, no complex data structure is needed — simple atomic-type variables (two floats/doubles for the numbers, one character for the operator) are sufficient.
4. Algorithm Design: Design the logic — typically a conditional (if-else or switch) structure that checks the operator symbol and performs the corresponding arithmetic operation, including a check for division by zero when the operator is ÷.
5. Implementation: Write the actual program, reading the two numbers and the operator from the user, applying the designed logic, and displaying the computed result.
6. Testing and Verification: Test with an invalid operator symbol, a division by zero case, and negative numbers, to confirm the calculator handles all cases correctly.

Problem 15: A social networking application needs to model 'friend connections' between users, where any user can be friends with any number of other users, and friendships can form cycles (e.g., A is friends with B, B is friends with C, and C is friends with A). Should this be modeled using a Linear or Non-Linear data structure, and which specific data structure is most suitable? Justify your answer.

Solution:

This should be modeled using a Non-Linear data structure, specifically a Graph, where each user is represented as a vertex (node), and each friendship is represented as an edge connecting two vertices.

A Linear data structure (like an Array or Linked List) is unsuitable here because it can only represent a simple sequential relationship between elements (each element having at most one predecessor and one successor), but friend connections do not follow any such sequence — a single user can be connected to many other users simultaneously, with no inherent order.

A Tree is also unsuitable, since a Tree does not allow cycles (as established in Problem 6), but the scenario explicitly describes a cycle (A–B–C–A) as a valid situation, which only a general Graph structure can represent.

Therefore, a Graph is the correct choice, since it naturally supports any number of connections per node and allows cycles, exactly matching the real-world structure of a social network's friendships.