PROBLEMS AND SOLUTIONS

Chapter 1: Introduction to Data Structure

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

This document contains frequently asked exam questions and analytical problems based on the topics of Data Representation, Abstract Data Types, Data Structures, Atomic and Structured Types, Linear and Non-Linear Data Types, Primitive and Non-Primitive Data Types, and Refinement Stages. These problem types are commonly asked in semester examinations and should be practiced thoroughly.

Section A: Classification Problems

Problem 1: Classify the following into Linear and Non-Linear data structures: Array, Tree, Stack, Graph, Queue, Linked List, Binary Search Tree, Matrix.

Solution:

A structure is linear if its elements form a sequence where each element (except the first and last) has exactly one predecessor and one successor. A structure is non-linear if elements can be connected to more than one other element, forming a hierarchy or network.

Linear Data Structures	Non-Linear Data Structures
Array, Stack, Queue, Linked List, Matrix	Tree, Graph, Binary Search Tree

---

Problem 2: Classify the following data types into Primitive and Non-Primitive: int, Array, char, Structure, float, Union, boolean, Linked List, Stack, double.

Solution:

Primitive data types are basic, built-in types directly supported by the language and hardware. Non-primitive data types are derived from primitive types and are used to represent collections or composite data.

Primitive Data Types	Non-Primitive Data Types
int, char, float, boolean, double	Array, Structure, Union, Linked List, Stack

Section B: Tracing ADT Operations

Problem 3: A Stack is empty. The following operations are performed in order: PUSH(10), PUSH(20), PUSH(30), POP(), PUSH(40), POP(), POP(). Show the contents of the stack after each operation and give the final state of the stack.

Solution:

A Stack follows the LIFO (Last In, First Out) principle, so PUSH adds an element to the top and POP removes the element currently at the top.

- PUSH(10) → Stack: [10] (10 is at top)
- PUSH(20) → Stack: [10, 20] (20 is at top)
- PUSH(30) → Stack: [10, 20, 30] (30 is at top)
- POP() → removes 30 → Stack: [10, 20]
- PUSH(40) → Stack: [10, 20, 40] (40 is at top)
- POP() → removes 40 → Stack: [10, 20]
- POP() → removes 20 → Stack: [10]

```
Final Stack Contents = [10]
```

Problem 4: A Queue is empty. The following operations are performed in order: ENQUEUE(A), ENQUEUE(B), ENQUEUE(C), DEQUEUE(), ENQUEUE(D), DEQUEUE(). Show the contents of the queue after each operation and give the final state of the queue.

Solution:

A Queue follows the FIFO (First In, First Out) principle, so ENQUEUE adds an element at the rear and DEQUEUE removes the element from the front.

- ENQUEUE(A) → Queue: [A] (front = A, rear = A)
- ENQUEUE(B) → Queue: [A, B] (front = A, rear = B)
- ENQUEUE(C) → Queue: [A, B, C] (front = A, rear = C)
- DEQUEUE() → removes A (front) → Queue: [B, C]
- ENQUEUE(D) → Queue: [B, C, D] (front = B, rear = D)
- DEQUEUE() → removes B (front) → Queue: [C, D]

```
Final Queue Contents = [C, D], with front = C and rear = D
```

Problem 5: Explain how a Queue ADT can be implemented using two Stacks. Trace the process for the operations ENQUEUE(1), ENQUEUE(2), DEQUEUE(), ENQUEUE(3), DEQUEUE().

Solution:

This construction uses two stacks, say Stack1 (for incoming elements) and Stack2 (for outgoing elements), to simulate FIFO behaviour using only LIFO stack operations.

Rule: For ENQUEUE, simply push the element onto Stack1. For DEQUEUE, if Stack2 is empty, pop all elements from Stack1 and push them onto Stack2 (this reverses their order), then pop and return the top of Stack2; if Stack2 is not empty, directly pop from Stack2.

- ENQUEUE(1) → Stack1: [1], Stack2: []
- ENQUEUE(2) → Stack1: [1, 2], Stack2: []
- DEQUEUE() → Stack2 is empty, so move all of Stack1 to Stack2: Stack1: [], Stack2: [2, 1]. Pop from Stack2 → returns 1. Stack2: [2]
- ENQUEUE(3) → Stack1: [3], Stack2: [2]
- DEQUEUE() → Stack2 is not empty, so pop directly from Stack2 → returns 2. Stack2: []

This confirms correct FIFO order of output (1, then 2), even though the underlying implementation uses two LIFO stacks — a good example of how an ADT's behaviour (Queue) can be built from a different underlying data structure (Stack).

Section C: Differentiating Key Concepts

Problem 6: Differentiate between Abstract Data Type, Data Type, and Data Structure with one example each.

Solution:

Basis	Data Type	Abstract Data Type	Data Structure
Meaning	Classifies the kind of value stored	Logical model defined by operations, hiding implementation	Actual implementation used to organize data
Example	int, float	Stack (push, pop, peek)	Array-based stack or linked-list-based stack

Problem 7: Differentiate between Primitive and Non-Primitive Data Types with one example each.

Solution:

Basis	Primitive Data Type	Non-Primitive Data Type
Definition	Basic, built-in type, not derived from any other type	Derived from primitive types, used to group or collect values
Memory support	Directly supported by hardware	Constructed using language features (not direct hardware support)
Example	int, char, float, boolean	Array, Structure, Linked List, Tree

Problem 8: Differentiate between Linear and Non-Linear Data Structures with one example each.

Solution:

Basis	Linear Data Structure	Non-Linear Data Structure
Arrangement	Elements arranged sequentially	Elements arranged hierarchically or as a network
Traversal	Can be traversed in a single run	Requires multi-level or specialized traversal (e.g., BFS/DFS)
Example	Array, Stack, Queue, Linked List	Tree, Graph

Section D: Refinement Stages — Applied Problems

Problem 9: A software company wants to build a simple Phonebook application that stores contact names and phone numbers, and allows adding, searching, and deleting a contact. Walk through the stages of refinement to arrive at an implementation.

Solution:

1. Problem Definition: Store a set of (name, phone number) pairs and support add, search, and delete operations on this set.
2. ADT Specification: Define a Contact List ADT with operations insert(name, number), search(name) returning the number, and delete(name), without deciding yet how these will be implemented internally.

3. Data Structure Selection: Choose a suitable data structure — for example, a sorted array (for fast search using binary search) or a hash table (for very fast average-case search), depending on whether frequent insertions/deletions or frequent searches are more important.
4. Algorithm Design: Design the step-by-step logic for each operation — e.g., for a sorted array, the insert algorithm must find the correct sorted position and shift elements; the search algorithm uses binary search; the delete algorithm finds the entry and shifts remaining elements.
5. Implementation: Translate the chosen data structure and algorithms into actual code using a programming language such as C or Java.
6. Testing and Verification: Test the application with various cases — adding duplicate names, searching for a non-existent contact, deleting from an empty list — to confirm correctness.

Problem 10: A college wants to process student examination results: store each student's roll number and marks in five subjects, and generate a report showing each student's total marks sorted from highest to lowest. Walk through the stages of refinement.

Solution:

7. Problem Definition: Given roll numbers and marks in five subjects for n students, compute total marks per student and output the list sorted in descending order of total marks.
8. ADT Specification: Define a Student Record ADT holding roll number and an array/list of five marks, along with an operation to compute the total, and a List ADT to hold and sort all student records.
9. Data Structure Selection: Use a structure (record) type to group each student's roll number and marks together, and an array of such structures to hold all n student records.
10. Algorithm Design: Design an algorithm to compute the total marks for each structure (a simple loop summing five values), and a sorting algorithm (e.g., Merge Sort) to arrange the array of structures by total marks in descending order.
11. Implementation: Write the actual program, defining the structure, reading input, computing totals, applying the chosen sorting algorithm, and printing the sorted report.
12. Testing and Verification: Test with tied totals, a single student, and the maximum expected number of students, to confirm the report is generated correctly in all cases.

Section E: Atomic Types, Structures, and Data Representation

Problem 11: Classify the following as Atomic Type or Structured Type: int, a Student record containing name and roll number, char, an array of 10 integers, float, a Book record containing title and price.

Solution:

An atomic type is indivisible and represents a single value, while a structured type is built by combining atomic (or other structured) types into a single composite unit.

Atomic Types	Structured Types
int, char, float	Student record (name, roll number), Array of 10 integers, Book record (title, price)

Problem 12: Convert the decimal number 25 into its binary representation, and explain how this value would be stored inside computer memory.

Solution:

To convert 25 to binary, repeatedly divide by 2 and record the remainders: 25 ÷ 2 = 12 remainder 1; 12 ÷ 2 = 6 remainder 0; 6 ÷ 2 = 3 remainder 0; 3 ÷ 2 = 1 remainder 1; 1 ÷ 2 = 0 remainder 1.

Reading the remainders from bottom to top gives the binary equivalent:

$(25)_{10} = (11001)_2$

Since a standard int typically occupies 4 bytes (32 bits) of memory, the value 25 would be stored as 00000000 00000000 00000000 00011001 in binary form, with leading zeros filling the remaining bits of the allocated memory.

Problem 13:

Given the following C structure definition, calculate the total memory required for one variable of this structure type (assume char = 1 byte, int = 4 bytes, float = 4 bytes, and no padding):

```
struct Student {
    char name[20];
    int roll;
    float marks;
};
```

Solution:

A structure's total size (ignoring padding/alignment) is the sum of the sizes of all its individual members.

- name[20] → 20 characters × 1 byte = 20 bytes
- roll → 1 int × 4 bytes = 4 bytes
- marks → 1 float × 4 bytes = 4 bytes

Total memory = 20 + 4 + 4 = 28 bytes.

```
sizeof(struct Student) = 28 bytes
```

(Note: In practice, compilers may add padding bytes for memory alignment, which can make the actual sizeof() result slightly larger than this theoretical value.)

Problem 14: State whether each of the following statements is True or False, with a brief justification: (a) Every non-primitive data type is also a structured type. (b) A Stack ADT can only be implemented using an array.

Solution:

(a) True. Non-primitive data types are, by definition, built by combining or deriving from primitive types — this is exactly what a structured (composite) type is. Arrays, structures, unions, and linked structures are all non-primitive and structured.

(b) False. A Stack ADT is defined purely by its behaviour (push, pop, peek following LIFO order), not by any specific implementation. It can be implemented using either an array (with a fixed or dynamically resized size) or a linked list (using pointers), among other options — the choice of underlying data structure is independent of the ADT's definition.

Problem 15: A List of 100 integers needs to support frequent insertions and deletions at arbitrary positions, but search operations are rare. Should this be implemented using an Array or a Linked List? Justify your answer using the concepts of data representation and data structure selection.

Solution:

An Array stores elements in contiguous memory locations. Inserting or deleting an element at an arbitrary position in an array requires shifting all subsequent elements, which takes O(n) time in the worst case for each such operation.

A Linked List stores elements as nodes connected via pointers, not necessarily in contiguous memory. Once the correct position (node) is located, insertion or deletion only requires updating a constant number of pointers, taking O(1) time for the actual insert/delete step (though locating the position may still take O(n) time).

Since the given requirement emphasizes frequent insertions and deletions (where a Linked List avoids the costly element-shifting that an Array requires) and treats search as rare (where the Array's advantage of fast indexed/binary search is not important here), a Linked List is the more suitable data structure for this scenario.