PROBLEMS AND SOLUTIONS — PART 3

Chapter 1: Introduction to Data Structure

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

This is a third 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 or 2 — including postfix evaluation, doubly linked lists, priority queues, deques, sparse matrices, and static vs dynamic data structures. Practice all three parts together for complete exam coverage of this chapter.

Section A: Tracing More Stack, Queue, and List Applications

Problem 1: Evaluate the postfix expression 5 3 4 * + 2 - using a Stack. Show the stack contents at each step and give the final result.

Solution:

Rule: Scan left to right. Push every operand onto the stack. When an operator is encountered, pop the top two operands (the second-popped is the left operand), apply the operator, and push the result back.

- Read '5' (operand) → push → Stack: [5]
- Read '3' (operand) → push → Stack: [5, 3]
- Read '4' (operand) → push → Stack: [5, 3, 4]
- Read '*' → pop 4 and 3 → compute 3 * 4 = 12 → push 12 → Stack: [5, 12]
- Read '+' → pop 12 and 5 → compute 5 + 12 = 17 → push 17 → Stack: [17]
- Read '2' (operand) → push → Stack: [17, 2]
- Read '-' → pop 2 and 17 → compute 17 - 2 = 15 → push 15 → Stack: [15]

```
Final Result = 15
```

Problem 2: Starting from an empty Doubly Linked List, perform: Insert 10 at the end; Insert 20 at the end; Insert 5 at the beginning; Delete the node containing 20. Show the list (with forward and backward links) after each step.

Solution:

In a Doubly Linked List, each node stores a pointer to both the next node AND the previous node, allowing traversal in both directions. Deleting a node requires updating the 'next' pointer of its previous node and the 'prev' pointer of its next node.

- Insert 10 at end → List: NULL ← 10 → NULL
- Insert 20 at end → List: NULL ← 10 ↔ 20 → NULL
- Insert 5 at beginning → List: NULL ← 5 ↔ 10 ↔ 20 → NULL

- Delete node containing 20 → node 10's 'next' pointer is updated to NULL → List: NULL ← 5 ↔ 10 → NULL

```
Final Doubly Linked List = 5 ↔ 10
```

Problem 3: A Priority Queue (higher number = higher priority) starts empty. Perform: Insert(TaskA, 5), Insert(TaskB, 1), Insert(TaskC, 3), ExtractMax(), Insert(TaskD, 4), ExtractMax(). Show which task is removed at each ExtractMax() call.

Solution:

Unlike a normal Queue (which removes strictly in insertion order), a Priority Queue always removes the element with the highest priority currently present, regardless of when it was inserted.

- Insert(TaskA, 5) → Queue holds: {A:5}
- Insert(TaskB, 1) → Queue holds: {A:5, B:1}
- Insert(TaskC, 3) → Queue holds: {A:5, B:1, C:3}
- ExtractMax() → highest priority is A (priority 5) → removes TaskA → Queue holds: {B:1, C:3}
- Insert(TaskD, 4) → Queue holds: {B:1, C:3, D:4}
- ExtractMax() → highest priority is D (priority 4) → removes TaskD → Queue holds: {B:1, C:3}

Note that TaskC was inserted before TaskD but is extracted after it, since Priority Queue ordering is based on priority value, not insertion order.

Problem 4: A Deque (Double-Ended Queue) is empty. Perform: InsertRear(1), InsertRear(2), InsertFront(3), DeleteRear(), InsertFront(4), DeleteFront(). Show the deque contents after each operation.

Solution:

A Deque allows insertion and deletion at BOTH the front and the rear ends, unlike a normal Queue which only allows insertion at the rear and deletion at the front.

- InsertRear(1) → Deque: [1]
- InsertRear(2) → Deque: [1, 2]
- InsertFront(3) → Deque: [3, 1, 2]
- DeleteRear() → removes 2 (rear) → Deque: [3, 1]
- InsertFront(4) → Deque: [4, 3, 1]
- DeleteFront() → removes 4 (front) → Deque: [3, 1]

```
Final Deque Contents = [3, 1]
```

Section B: Differentiating Concepts

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

Solution:

Basis	Singly Linked List	Doubly Linked List
Pointers per node	One (points to next node only)	Two (point to both next and previous node)
Traversal direction	Forward only	Both forward and backward

Memory usage per node	Lower (one pointer)	Higher (two pointers)
Deletion of a given node	Requires access to the previous node (must traverse from head)	Easier — previous node is directly accessible via the 'prev' pointer

---

Problem 6: Differentiate between a Linear Queue and a Circular Queue, and explain why a Circular Queue is generally preferred in an array-based implementation.

Solution:

Basis	**Linear Queue**	**Circular Queue**
Structure	Front and rear move only in one direction through the array	Front and rear wrap around back to index 0 after reaching the end
Space utilization	Freed space at the front (after dequeues) cannot be reused, leading to wasted space	Freed space is reused via wrap-around (modulo arithmetic), so space is never wasted

A Circular Queue is generally preferred because, in a Linear Queue, once elements are dequeued from the front, those array positions become permanently unusable even though they are technically empty, potentially causing the queue to report 'full' while much of the array is actually free. The Circular Queue solves this by treating the array as circular, so removed positions can be reused immediately by future insertions.

Problem 7: Differentiate between Static Data Structures and Dynamic Data Structures, with one example of each.

Solution:

Basis	**Static Data Structure**	**Dynamic Data Structure**
Memory allocation	Fixed amount of memory allocated at compile time	Memory allocated and deallocated at run time, as needed
Size flexibility	Size cannot be changed once declared	Size can grow or shrink during program execution
Example	Array	Linked List

Section C: Data Representation and Memory Calculation

Problem 8:

Represent the following 4×4 sparse matrix using the Triplet (row, column, value) representation, and calculate the memory saved compared to storing the full matrix (assume each value, including row and column indices, takes 4 bytes).

```
| 0  0  5  0 |
| 0  8  0  0 |
| 0  0  0  0 |
| 3  0  0  0 |
```

Solution:

A sparse matrix has most of its elements equal to zero. Instead of storing all 16 elements, the Triplet representation stores only the non-zero elements, each as a (row, column, value) triple, along with a header giving the matrix dimensions and the count of non-zero elements.

Non-zero elements here: (0, 2, 5), (1, 1, 8), (3, 0, 3) — 3 non-zero elements out of 16 total.

Full matrix storage: 16 elements × 4 bytes = 64 bytes.

Triplet representation storage: (1 header triple for rows, columns, non-zero count) + 3 data triples = 4 triples × 3 fields × 4 bytes = 48 bytes.

```
Memory saved = 64 - 48 = 16 bytes
```

(Note: The savings become far more significant for larger, sparser matrices — e.g., a 100×100 matrix with only 5 non-zero elements would save a very large amount of memory using this representation.)

Problem 9:

Calculate the total memory required for one variable of the following structure on a 64-bit system (assume int = 4 bytes, and a pointer = 8 bytes on a 64-bit system, with no padding):

```
struct Node {
    int data;
    struct Node *next;
};
```

Solution:

This structure represents a typical Linked List node, containing one data field and one pointer field (used to link to the next node).

- data → 1 int × 4 bytes = 4 bytes
- next → 1 pointer × 8 bytes (on a 64-bit system) = 8 bytes

Total memory = 4 + 8 = 12 bytes.

```
sizeof(struct Node) = 12 bytes (on a 64-bit system, ignoring padding)
```

(Note: On a 32-bit system, a pointer typically occupies only 4 bytes, which would make sizeof(struct Node) = 8 bytes instead — this shows that structure size can depend on the underlying system architecture.)

Problem 10: Convert the binary number $(110101)_2$ into its decimal equivalent.

Solution:

To convert binary to decimal, multiply each bit by 2 raised to the power of its position (counting from 0 on the right) and sum the results.

$(110101)_2 = 1\times2^5 + 1\times2^4 + 0\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0$

$= 1\times32 + 1\times16 + 0\times8 + 1\times4 + 0\times2 + 1\times1 = 32 + 16 + 0 + 4 + 0 + 1 = 53.$

$$(110101)_2 = (53)_{10}$$

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 Parking Lot Management system that needs to park a vehicle, remove a vehicle, check if the lot is full, and report the number of available slots.

Solution:

- parkVehicle(vehicleNumber) — assigns an available slot to the vehicle and marks it occupied, if a slot is free.
- removeVehicle(vehicleNumber) — frees the slot occupied by the given vehicle, making it available again.

- isFull() — returns True if no slots are currently available, False otherwise.
- availableSlots() — returns the count of currently unoccupied slots.

As with any ADT specification, this defines only WHAT the system can do, without committing to HOW slots are tracked internally (e.g., using an array of booleans, a linked list of free slots, or some other structure) — that decision belongs to a later Data Structure Selection stage.

Problem 12: A text editor needs to implement an 'Undo' feature, where the most recently performed action should be the first one undone when the user requests an undo. Which Abstract Data Type is best suited for this, and why?

Solution:

A Stack is the most suitable ADT for implementing an Undo feature, because it follows the LIFO (Last In, First Out) principle — exactly matching the requirement that the most recently performed action must be undone first.

In practice, every action the user performs (typing, deleting, formatting, etc.) is pushed onto an 'undo stack'. When the user requests Undo, the top action is popped from the stack and reversed. This is a classic real-world application of the Stack ADT in software design.

Problem 13: An integer array A[5] is stored starting at base address 2000, with each integer occupying 4 bytes. Calculate the memory address of each element A[0] through A[4], and explain why array indexing conventionally starts at 0 rather than 1.

Solution:

The address of any element A[i] is calculated as: Address(A[i]) = Base Address + i × size_of_element.

- A[0] → 2000 + 0×4 = 2000
- A[1] → 2000 + 1×4 = 2004
- A[2] → 2000 + 2×4 = 2008
- A[3] → 2000 + 3×4 = 2012
- A[4] → 2000 + 4×4 = 2016

Array indexing starts at 0 because the index essentially represents the OFFSET (in units of element size) from the base address, not a position count. The first element sits exactly at the base address itself, meaning it requires an offset of 0 elements from the start — which is why it is naturally labelled index 0, not index 1.

Section E: Applied Refinement and Conceptual Review

Problem 14: A cinema wants to build a Movie Ticket Booking System that allows a customer to view available seats for a show, book a specific seat, and cancel a booking. Walk through the stages of refinement to arrive at an implementation.

Solution:

1. Problem Definition: For a given show with a fixed number of seats, allow viewing available seats, booking a specific available seat, and cancelling an existing booking.
2. ADT Specification: Define a Seating ADT with operations viewAvailableSeats(), bookSeat(seatNumber, customerName), and cancelBooking(seatNumber), without deciding the internal representation yet.

3. Data Structure Selection: Use an array (or a similar structured collection) of seat records, where each record holds a seat number, its booking status (booked/available), and the customer name if booked — since the number of seats in a show is fixed and known in advance.
4. Algorithm Design: Design algorithms for each operation — e.g., viewAvailableSeats() scans the array and lists seats marked available; bookSeat() checks if the given seat is available before marking it booked; cancelBooking() marks a booked seat as available again.
5. Implementation: Write the actual program, defining the seat record structure, the array of seats, and the logic for each operation.
6. Testing and Verification: Test booking an already-booked seat (should fail), cancelling a seat that was never booked (should be handled gracefully), and booking the very last available seat.

Problem 15: State whether each of the following statements is True or False, with a brief justification: (a) A Circular Queue can be implemented using an array. (b) A Stack allows insertion and deletion from both ends. (c) In Row-Major storage, all elements of a given row are stored in contiguous memory locations. (d) A Doubly Linked List always uses more memory per node than a Singly Linked List.

Solution:

(a) True. A Circular Queue is commonly implemented using a fixed-size array, along with front and rear indices that wrap around using the modulo operator, allowing efficient reuse of freed space.

(b) False. A Stack allows insertion (push) and deletion (pop) only from ONE end, called the top. Allowing operations at both ends is instead the defining feature of a Deque, not a Stack.

(c) True. This is exactly the definition of Row-Major order — all elements belonging to the same row are placed together in memory before moving on to the elements of the next row, which is what makes the Row-Major address formula (Base + (i × columns + j) × size) work correctly.

(d) True. Each node in a Doubly Linked List stores an additional pointer (to the previous node) compared to a Singly Linked List node, which only stores a pointer to the next node. This extra pointer means every Doubly Linked List node requires strictly more memory than the equivalent Singly Linked List node.