PROBLEMS AND SOLUTIONS — PART 2

Chapter 1: Introduction to Data Structure

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

This is a second set of frequently asked exam questions and analytical problems on Data Representation, Abstract Data Types, Data Structures, Atomic and Structured Types, Linear and Non-Linear Data Types, and related concepts — covering problem types not included in Part 1, such as expression conversion, circular queues, linked list tracing, unions, and memory address calculation. Practice both parts together for complete exam coverage of this chapter.

Section A: Tracing Stack, Queue, and List Applications

Problem 1: Using a Stack, check whether the expression { [( ) ( )] } has balanced parentheses/brackets. Show the stack contents as each symbol is processed.

Solution:

Rule: For every opening symbol ({, [, (), push it onto the stack. For every closing symbol (},],)), pop the stack and check that the popped symbol is the matching opening symbol; if the stack is empty when a closing symbol appears, or the popped symbol does not match, the expression is unbalanced.

- Read '{' → push → Stack: [{]
- Read '[' → push → Stack: [{, [ ]
- Read '(' → push → Stack: [{, [, ( ]
- Read ')' → pop '(' → matches → Stack: [{, [ ]
- Read '(' → push → Stack: [{, [, ( ]
- Read ')' → pop '(' → matches → Stack: [{, [ ]
- Read ']' → pop '[' → matches → Stack: [{]
- Read '}' → pop '{' → matches → Stack: [ ] (empty)

Since the stack is empty at the end and every closing symbol matched correctly, the expression is balanced.

Problem 2: Convert the infix expression A + B * C - D to postfix (Reverse Polish) form using the Stack-based operator precedence algorithm. Show the stack and output at each step.

Solution:

Rule: Operands go directly to output. An operator is pushed onto the stack after popping (to output) any operators on top of the stack that have equal or higher precedence than it. At the end, pop all remaining operators to output.

- Read 'A' (operand) → Output: A

- Read '+' → stack empty, push → Stack: [+], Output: A
- Read 'B' (operand) → Output: AB
- Read '*' → higher precedence than '+' on stack, push → Stack: [+, *], Output: AB
- Read 'C' (operand) → Output: ABC
- Read '-' → pop '*' (≥ precedence) → Output: ABC*; pop '+' (≥ precedence) → Output: ABC*+; stack now empty, push '-' → Stack: [-]
- Read 'D' (operand) → Output: ABC*+D
- End of expression → pop remaining '-' → Output: ABC*+D-

```
Postfix Expression = A B C * + D -
```

Problem 3: Starting from an empty Singly Linked List, perform the following operations in order: Insert 10 at the beginning; Insert 20 at the end; Insert 15 after 10; Delete node containing 20. Show the linked list after each step.

Solution:

Insert at beginning creates a new node and makes it the new head. Insert at end traverses to the last node and links a new node after it. Insert after a given node links a new node between it and its current next node. Delete removes a node and re-links its previous node directly to its next node.

- Insert 10 at beginning → List: 10 → NULL
- Insert 20 at end → List: 10 → 20 → NULL
- Insert 15 after 10 → List: 10 → 15 → 20 → NULL
- Delete node containing 20 → List: 10 → 15 → NULL

```
Final Linked List = 10 → 15 → NULL
```

Problem 4: A Circular Queue is implemented using an array of capacity 5 (index 0 to 4), with front = 0, rear = -1, and count = 0 initially. Perform: Enqueue(10), Enqueue(20), Enqueue(30), Dequeue(), Enqueue(40), Enqueue(50), Enqueue(60). Show the array indices used and the final queue contents, noting where wrap-around occurs.

Solution:

Rule: For Enqueue, rear = (rear + 1) % capacity, then place the element at index rear, and increase count. For Dequeue, remove the element at index front, then front = (front + 1) % capacity, and decrease count.

- Enqueue(10) → rear = (-1+1)%5 = 0, arr[0]=10, count=1
- Enqueue(20) → rear = 1, arr[1]=20, count=2
- Enqueue(30) → rear = 2, arr[2]=30, count=3
- Dequeue() → removes arr[front]=arr[0]=10; front = (0+1)%5 = 1; count=2
- Enqueue(40) → rear = 3, arr[3]=40, count=3
- Enqueue(50) → rear = 4, arr[4]=50, count=4
- Enqueue(60) → rear = (4+1)%5 = 0 (wrap-around to index 0), arr[0]=60 (old value 10 already removed, so this is safe), count=5 (queue now full)

Final state: front = 1, rear = 0, and reading the queue in FIFO order from front to rear (with wrap-around) gives:

```
Queue Contents (front to rear) = 20, 30, 40, 50, 60
```

This demonstrates the key advantage of a Circular Queue over a simple linear array queue: index 0 (freed by the earlier Dequeue) is reused instead of being permanently wasted.

Section B: Differentiating Concepts

Problem 5: Differentiate between Array and Linked List on the basis of memory allocation, access time, and insertion/deletion cost.

Solution:

Basis	Array	Linked List
Memory allocation	Contiguous memory, usually fixed size	Non-contiguous memory (nodes), allocated dynamically as needed
Access time	O(1) direct access using an index	O(n) — must traverse nodes sequentially from the head
Insertion/Deletion (at arbitrary position)	O(n) — requires shifting elements	O(1) once the position is located, since only pointers are updated

Problem 6: Differentiate between Stack and Queue as Abstract Data Types, giving one real-life example of each.

Solution:

Basis	Stack	Queue
Principle	LIFO (Last In, First Out)	FIFO (First In, First Out)
Access points	Insertion and removal both happen at the same end (top)	Insertion happens at the rear; removal happens at the front
Real-life example	A pile of plates — the last plate placed is the first one removed	A queue/line of people at a ticket counter — the first person to join is the first served

Problem 7:

Differentiate between Structure and Union, using the following equivalent definitions (assume char = 1 byte, int = 4 bytes, float = 4 bytes):

```
struct S { char name[20]; int roll; float marks; };
union  U { char name[20]; int roll; float marks; };
```

Solution:

A structure allocates separate memory for every member, so all members can hold values simultaneously. A union allocates a single shared memory block, large enough for its largest member only, so all members overlap in memory and only one member can meaningfully hold a value at a time.

For struct S: total size = 20 + 4 + 4 = 28 bytes (sum of all members).

For union U: total size = max(20, 4, 4) = 20 bytes (size of the largest member only).

```
sizeof(struct S) = 28 bytes;   sizeof(union U) = 20 bytes
```

This shows that a union is used when memory efficiency is important and only one of several possible fields needs to be stored at any given time, while a structure is used when all fields must be stored and accessed together.

Section C: Data Representation and Memory Address Calculation

Problem 8: Find the ASCII value of the character 'A' and represent it in 8-bit binary form.

Solution:

The character 'A' has a standard ASCII (decimal) value of 65.

Converting 65 to binary: 65 ÷ 2 = 32 remainder 1; 32 ÷ 2 = 16 remainder 0; 16 ÷ 2 = 8 remainder 0; 8 ÷ 2 = 4 remainder 0; 4 ÷ 2 = 2 remainder 0; 2 ÷ 2 = 1 remainder 0; 1 ÷ 2 = 0 remainder 1. Reading remainders bottom to top: 1000001.

Padding this 7-bit result to a standard 8-bit representation by adding a leading zero:

```
'A' = 65 (decimal) = 01000001 (8-bit binary)
```

Problem 9: A 2D integer array A[10][20] is stored in Row-Major order in memory, starting at base address 1000, with each integer occupying 4 bytes. Find the memory address of the element A[3][5].

Solution:

In Row-Major order, all elements of a row are stored together before moving to the next row. The address formula is: Address(A[i][j]) = Base Address + (i × number_of_columns + j) × size_of_element.

Here, Base Address = 1000, number_of_columns = 20, i = 3, j = 5, size_of_element = 4 bytes.

Address = 1000 + (3 × 20 + 5) × 4 = 1000 + (60 + 5) × 4 = 1000 + 65 × 4 = 1000 + 260 = 1260.

```
Address of A[3][5] = 1260
```

Section D: ADT Design and Conceptual Understanding

Problem 10: Write only the Abstract Data Type (ADT) specification (the set of operations, without implementation details) for a Library Book Management system that needs to add a new book, search for a book by title, issue a book to a member, and return a book.

Solution:

An ADT specification lists only WHAT operations are available and what they do, without deciding whether the underlying implementation will use an array, a linked list, or any other data structure.

- addBook(title, author, ISBN) — adds a new book record to the library collection.
- searchBook(title) — returns the book record matching the given title, if it exists.
- issueBook(ISBN, memberID) — marks the book with the given ISBN as issued to the specified member, if it is currently available.
- returnBook(ISBN) — marks the book with the given ISBN as available again in the collection.

Note that this specification says nothing about how books are actually stored (array, linked list, hash table, etc.) — that decision belongs to the Data Structure Selection stage, which comes after the ADT is defined.

Problem 11: A Stack of maximum capacity 3 currently contains elements [5, 10] (10 at top). The following operations are performed in order: PUSH(15), PUSH(20), POP(), POP(), POP(), POP(). Identify at which step, if any, a Stack Overflow or Stack Underflow occurs, and explain why.

Solution:

Stack Overflow occurs when PUSH is attempted on a stack that is already at maximum capacity. Stack Underflow occurs when POP is attempted on a stack that is already empty.

- PUSH(15) → Stack: [5, 10, 15] (now at capacity 3 — full, but this push itself was valid)

- PUSH(20) → Stack is already full (capacity 3) → STACK OVERFLOW — this push is rejected, Stack remains [5, 10, 15]
- POP() → removes 15 → Stack: [5, 10]
- POP() → removes 10 → Stack: [5]
- POP() → removes 5 → Stack: [] (empty)
- POP() → Stack is already empty → STACK UNDERFLOW — this pop is rejected

This illustrates why real implementations of the Stack ADT must always check for fullness before PUSH and emptiness before POP, to avoid these error conditions.

Problem 12: State whether each of the following statements is True or False, with a brief justification: (a) Every Queue can be considered a special type of List. (b) An Array can be considered a special, restricted case of a Structure.

Solution:

(a) True. A List ADT generally allows insertion and deletion at any position, while a Queue restricts these operations to only two specific ends (insertion at the rear, deletion at the front). Since a Queue's behaviour is a constrained subset of what a general List allows, a Queue can be viewed as a specialized/restricted form of a List.

(b) False. This statement has the relationship backwards. An Array is a structured type where all elements must be of the SAME data type, whereas a Structure is a structured type where members can be of DIFFERENT data types. They are both structured types derived from atomic types, but neither is a restricted case of the other — they serve different purposes (homogeneous collection vs heterogeneous grouping).

Section E: Applied Refinement and Structure Selection

Problem 13: A company wants to build an Employee Payroll System that stores each employee's ID, name, basic salary, and allowances, and calculates the net salary (basic + allowances) for all employees, printing a payslip for each. Walk through the stages of refinement to arrive at an implementation.

Solution:

1. Problem Definition: Given ID, name, basic salary, and allowances for n employees, compute the net salary for each and generate a payslip.
2. ADT Specification: Define an Employee Record ADT holding ID, name, basic salary, and allowances, with an operation to compute net salary, and a Payroll List ADT to hold and process all employee records.
3. Data Structure Selection: Use a structure (record) type to group each employee's ID, name, basic salary, and allowances together, and an array (or linked list, if the number of employees changes frequently) of such structures to hold all employee records.
4. Algorithm Design: Design a simple loop-based algorithm that, for each structure in the collection, computes net salary = basic salary + allowances, and formats this into a payslip.
5. Implementation: Write the program, defining the structure, reading employee data, computing net salaries in the loop, and printing formatted payslips.
6. Testing and Verification: Test with an employee having zero allowances, the maximum expected number of employees, and check that all payslip values are calculated and formatted correctly.

Problem 14: A printer needs to process print jobs sent by multiple users in the exact order they were submitted, since it is unfair for a job submitted later to be printed before one submitted earlier. Which Abstract Data Type is most suitable for managing these print jobs, and which underlying data structure would you choose to implement it? Justify your choice.

Solution:

The most suitable ADT here is a Queue, since it enforces the FIFO (First In, First Out) principle — exactly the fairness requirement described, where jobs must be printed in the same order they were submitted.

For the underlying data structure, a Linked List implementation of the Queue is generally preferred over an Array implementation, because the number of print jobs waiting can vary unpredictably over time (jobs are added and removed continuously), and a Linked List can grow or shrink dynamically without needing a pre-defined maximum capacity, unlike a plain array.

This is a practical, real-world example of how the Queue ADT (a logical concept) is actually implemented in operating systems as a 'print spooler' using a linked-list-based data structure.

Problem 15: Explain, with an example, why choosing an unsuitable data structure for a given problem can significantly affect the performance of a program — compare using an Unsorted Array versus a Binary Search Tree (BST) for an application that performs frequent search operations on a large, changing dataset.

Solution:

Searching an Unsorted Array requires checking elements one by one from the beginning until the target is found or the array ends, giving a worst-case and average-case time complexity of O(n) per search.

A Binary Search Tree (BST), when reasonably balanced, allows searching by repeatedly comparing the target with the current node and moving left or right, discarding half of the remaining elements at each step, giving an average time complexity of O(log n) per search.

For a large dataset (say, n = 1,000,000) with frequent searches, an Unsorted Array would require up to 1,000,000 comparisons per search in the worst case, whereas a balanced BST would require only about $\log_2(1{,}000{,}000) \approx 20$ comparisons per search.

This massive difference demonstrates that selecting the right data structure for a problem's actual access pattern (frequent searching, in this case) is critical — using the wrong data structure does not change what the program computes, but can make it thousands of times slower for large inputs.