PROBLEMS AND SOLUTIONS — PART 5

Chapter 1: Introduction to Data Structure

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

This is a fifth 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 4 — including Binary Search Tree construction, palindrome checking, Column-Major addressing, hexadecimal conversion, and the time-space trade-off concept. Practice all five parts together for complete exam coverage of this chapter.

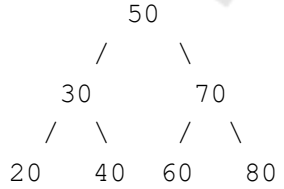
Section A: Tracing More Data Structure Applications

Problem 1: Construct a Binary Search Tree (BST) by inserting the following values in order: 50, 30, 70, 20, 40, 60, 80. Show the resulting tree structure.

Solution:

Rule: In a BST, for every node, all values in its left subtree are smaller than the node, and all values in its right subtree are greater. To insert a value, compare it with the current node starting from the root, moving left if smaller or right if larger, until an empty spot is found.

- Insert 50 → becomes the root.
- Insert 30 → 30 < 50 → goes to the left of 50.
- Insert 70 → 70 > 50 → goes to the right of 50.
- Insert 20 → 20 < 50 → left; 20 < 30 → goes to the left of 30.
- Insert 40 → 40 < 50 → left; 40 > 30 → goes to the right of 30.
- Insert 60 → 60 > 50 → right; 60 < 70 → goes to the left of 70.
- Insert 80 → 80 > 50 → right; 80 > 70 → goes to the right of 70.



Problem 2: Using a Stack, check whether the string "MADAM" is a palindrome. Show the stack contents and the comparison steps.

Solution:

Rule: Push the characters of the first half of the string onto the stack (skip the middle character if the length is odd). Then, for each remaining character in the second half, pop the stack and compare — if all comparisons match, the string is a palindrome.

"MADAM" has 5 characters (indices 0–4): M(0) A(1) D(2) A(3) M(4). The middle character (index 2, 'D') is skipped.

- Push M (index 0) → Stack: [M]
- Push A (index 1) → Stack: [M, A]
- Skip D (index 2, the middle character)
- Compare index 3 ('A') with pop() → pops 'A' → match → Stack: [M]
- Compare index 4 ('M') with pop() → pops 'M' → match → Stack: [] (empty)

Since every comparison matched and the stack is now empty, "MADAM" IS a palindrome.

Problem 3: A Stack is empty. Perform: PUSH(5), PUSH(10), PEEK(), PUSH(15), POP(), PEEK(). State the value returned by each PEEK() and POP() call, and show the stack contents after each operation.

Solution:

PEEK() (also called TOP()) returns the value of the topmost element WITHOUT removing it from the stack, unlike POP() which removes it.

- PUSH(5) → Stack: [5]
- PUSH(10) → Stack: [5, 10]
- PEEK() → returns 10 (top element); Stack remains: [5, 10] (unchanged)
- PUSH(15) → Stack: [5, 10, 15]
- POP() → removes and returns 15; Stack: [5, 10]
- PEEK() → returns 10 (top element); Stack remains: [5, 10] (unchanged)

This demonstrates the key difference between PEEK (read-only, does not modify the stack) and POP (modifies the stack by removing the top element).

Problem 4: A Stack is implemented using an array of capacity 5 (indices 0 to 4), with a variable 'top' initialized to -1 (indicating an empty stack). Perform: PUSH(5), PUSH(10), PUSH(15), POP(), PUSH(20). Show the array contents and the value of 'top' after each operation.

Solution:

Rule: For PUSH, increment 'top' first, then place the new value at index 'top'. For POP, simply decrement 'top' (the old value at that index is considered removed/logically deleted, even if it still physically remains in the array).

- PUSH(5) → top = 0, arr[0] = 5 → Array: [5, _, _, _, _], top = 0
- PUSH(10) → top = 1, arr[1] = 10 → Array: [5, 10, _, _, _], top = 1
- PUSH(15) → top = 2, arr[2] = 15 → Array: [5, 10, 15, _, _], top = 2
- POP() → top = 1 (value 15 is now logically removed) → Array: [5, 10, 15, _, _], top = 1
- PUSH(20) → top = 2, arr[2] = 20 (overwrites the old 15) → Array: [5, 10, 20, _, _], top = 2

This shows how the abstract Stack ADT (push/pop/peek) is concretely realized using an array and a single integer index variable ('top') as the underlying data structure.

Section B: Differentiating Concepts

Problem 5: Differentiate between a Full Binary Tree and a Complete Binary Tree.

Solution:

Basis	Full Binary Tree	Complete Binary Tree
Definition	Every node has either 0 or exactly 2 children (no node has only 1 child)	All levels are completely filled except possibly the last, and the last level is filled from left to right
Focus	Concerned with the number of children per node	Concerned with how levels are filled, left to right

These two properties are independent — a tree can be Full but not Complete, Complete but not Full, both, or neither, depending on its exact shape.

Problem 6:

Differentiate between Compile-Time Memory Allocation and Run-Time Memory Allocation, using the following two code snippets as reference:

```
int arr1[10];              // Snippet 1
int *arr2 = malloc(n * sizeof(int));   // Snippet 2 (n decided at run time)
```

Solution:

Basis	Compile-Time Allocation (Snippet 1)	Run-Time Allocation (Snippet 2)
When memory is reserved	During compilation, before the program actually runs	During program execution, only when the allocation statement runs
Size requirement	Must be a fixed constant known at compile time	Can depend on a value (like n) computed while the program is running
Typical mechanism	Simple variable/array declaration	Explicit function calls such as malloc() or new

---

Problem 7: Explain the difference between an ADT's Interface and its Implementation, using the Stack ADT as an example.

Solution:

The Interface of an ADT is the set of operation names, along with what inputs they take and what they return (their 'signature') — it is everything a user of the ADT can see and directly interact with. For a Stack, the interface consists of operations like push(x), pop(), and peek(), along with a description of what each does.

The Implementation is the actual internal code and data organization that makes each interface operation work correctly behind the scenes — for example, whether push() is implemented by writing to an array index and incrementing a 'top' variable, or by creating a new linked list node and updating a head pointer.

A user of the Stack ADT calls push() and pop() through the interface without needing to know (or care) which implementation is being used underneath — this separation is precisely what allows the implementation to be changed later (e.g., from array-based to linked-list-based) without requiring any change to the code that uses the Stack.

Section C: Data Representation and Memory Calculation

Problem 8: A 2D array A[4][3] (4 rows, 3 columns) is stored in Column-Major order, starting at base address 1000, with each element occupying 4 bytes. Find the memory address of the element A[2][1].

Solution:

In Column-Major order (unlike Row-Major), all elements of a given COLUMN are stored together in memory before moving to the next column. The address formula becomes: Address(A[i][j]) = Base + (j × number_of_rows + i) × size_of_element.

Here, Base = 1000, number_of_rows = 4, i = 2, j = 1, size_of_element = 4 bytes.

Address = 1000 + (1 × 4 + 2) × 4 = 1000 + (4 + 2) × 4 = 1000 + 6 × 4 = 1000 + 24 = 1024.

```
Address of A[2][1] (Column-Major) = 1024
```

(Note: This is different from the Row-Major formula used in earlier problems — the choice of storage order changes which formula applies, even for the same array indices.)

Problem 9: Convert the decimal number 202 into its hexadecimal equivalent, and explain why hexadecimal notation is commonly used to represent memory addresses.

Solution:

To convert 202 to hexadecimal, repeatedly divide by 16 and record the remainders (using A, B, C, D, E, F for remainders 10 through 15): 202 ÷ 16 = 12 remainder 10 (which is 'A'); 12 ÷ 16 = 0 remainder 12 (which is 'C').

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

$(202)_{10} = (CA)_{16}$

Hexadecimal notation is commonly used for memory addresses because each single hexadecimal digit represents exactly 4 binary bits (since $16 = 2^4$). This means a long, hard-to-read binary address can be written far more compactly and readably in hexadecimal — for example, a 32-bit binary address needs only 8 hexadecimal digits to represent it exactly, with no loss of information.

Problem 10:

Given the following nested structure definitions (assume int = 4 bytes, no padding), calculate the total size of one variable of type struct Polygon:

```
struct Point {
    int x;
    int y;
};
struct Polygon {
    struct Point vertices[4];
    int numSides;
};
```

Solution:

First, find the size of struct Point: x (4 bytes) + y (4 bytes) = 8 bytes.

struct Polygon contains an ARRAY of 4 such Point structures, plus one additional int member.

- vertices[4] → 4 × sizeof(struct Point) = 4 × 8 = 32 bytes
- numSides → 1 int × 4 bytes = 4 bytes

Total size = 32 + 4 = 36 bytes.

```
sizeof(struct Polygon) = 36 bytes
```

This demonstrates that structures can be nested — a structure can contain another structure (or an array of structures) as one of its members, and its total size is built up from the sizes of all its nested components.

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 Music Playlist system that supports adding a song, removing a song, playing the next song, and shuffling the playlist.

Solution:

- addSong(songTitle) — adds the given song to the end of the playlist.
- removeSong(songTitle) — removes the given song from the playlist, if it exists.
- playNext() — plays the song immediately following the currently playing song, and updates the current position.
- shuffle() — rearranges the order of songs in the playlist into a random order.

As always, this ADT specification defines only WHAT the playlist system can do — it does not commit to whether the playlist is stored as an array, a linked list, or any other structure internally.

Problem 12: Explain the concept of a 'Time-Space Trade-off' in the context of choosing between a Sorted Array and a Hash Table to store and search a large collection of records.

Solution:

A Sorted Array allows searching using Binary Search, giving a search time complexity of O(log n), and requires no memory beyond storing the n records themselves — it is relatively space-efficient.

A Hash Table can typically achieve an average search time complexity of O(1) (constant time), which is faster than the Sorted Array's O(log n). However, to keep collisions low and maintain this fast average performance, a Hash Table generally needs to allocate a table (array of buckets) that is somewhat larger than the actual number of records stored, using extra memory beyond just the n records.

This illustrates a Time-Space Trade-off: the Hash Table trades additional memory (space) for faster average search time, while the Sorted Array uses less memory but accepts a somewhat slower (though still efficient) search time. Neither choice is universally 'better' — the right choice depends on whether the application's constraints favour saving time or saving memory.

Problem 13: Explain why the Stack and Queue ADTs are often described as 'restricted' versions of the more general List ADT.

Solution:

A general List ADT typically allows insertion and deletion at ANY position within the sequence — the beginning, the end, or anywhere in the middle.

A Stack restricts these operations so that insertion (push) and deletion (pop) are allowed ONLY at one end (the top) — no other position may be directly accessed for insertion or deletion.

A Queue restricts these operations differently: insertion (enqueue) is allowed only at the rear, and deletion (dequeue) is allowed only at the front — two fixed, specific positions, rather than any arbitrary position.

Since both Stack and Queue only permit a smaller, fixed subset of the operations a general List would allow (at fixed positions, rather than arbitrary ones), they are considered specialized, 'restricted' forms of the List ADT — this restriction is precisely what gives them their well-defined, predictable LIFO and FIFO behaviour.

Section E: Applied Refinement and Conceptual Review

Problem 14: A small shop wants to build a basic Inventory Management System that stores each item's name, quantity, and price, and allows adding a new item, updating an item's quantity, removing an item, and generating a report of items whose quantity has fallen below a given threshold. Walk through the stages of refinement to arrive at an implementation.

Solution:

1. Problem Definition: Store name, quantity, and price for each item, and support adding items, updating quantities, removing items, and generating a low-stock report given a threshold value.
2. ADT Specification: Define an Inventory ADT with operations addItem(name, quantity, price), updateQuantity(name, newQuantity), removeItem(name), and lowStockReport(threshold), without deciding the internal storage yet.
3. Data Structure Selection: Use a structure (record) type to group each item's name, quantity, and price together, and an array (or a hash table keyed by item name, for faster lookup in a large inventory) of such structures to hold all items.
4. Algorithm Design: Design the logic for each operation — e.g., lowStockReport(threshold) scans through all item structures and lists every item whose quantity field is less than the given threshold.
5. Implementation: Write the actual program, defining the item structure, the chosen collection of items, and the code for each of the four operations.
6. Testing and Verification: Test with an empty inventory, an item quantity that is exactly equal to the threshold (boundary case), and attempting to remove an item that does not exist.

Problem 15: State whether each of the following statements is True or False, with a brief justification: (a) A Full Binary Tree must always also be a Complete Binary Tree. (b) Compile-time memory allocation determines the required memory size before the program actually runs. (c) A Hash Table always uses less memory than a Sorted Array for storing the same n records. (d) The interface of an ADT reveals the internal code used to implement each operation.

Solution:

(a) False. A tree can satisfy the Full Binary Tree property (every node has 0 or 2 children) without satisfying the Complete Binary Tree property (levels filled left to right with no gaps) — for example, a root with two children, where only the right child further has two children of its own, is Full but not Complete, since the last level is not filled from left to right.

(b) True. Compile-time allocation, by definition, reserves a fixed amount of memory (based on a constant size known in advance) during the compilation phase, before the program is actually executed.

(c) False. As explained in Problem 12, a Hash Table generally uses MORE memory than a Sorted Array for the same n records, since it needs extra table space (beyond the n records) to keep collisions low and maintain fast average-case performance — this is the essence of the time-space trade-off.

(d) False. This describes the IMPLEMENTATION of an ADT, not its interface. The interface only reveals the operation names and their signatures (inputs and outputs) — it deliberately hides the internal code and logic used to actually carry out each operation.