Lab 5

Problem 1

You are given an array of integers `arr[]`, where the elements first increase, reach a peak, and then decrease. The peak element is defined as an element that is greater than or equal to its neighbors. The array guarantees that there is exactly one peak element. Your goal is to determine the index of the peak element in the array.

You should solve this problem using an efficient approach with time complexity **O(log n)**. Justify your answer based on this input `[1, 2, 4, 5, 7, 8, 3]`

Answer

We will use a **binary search** approach, which allows us to find the peak element in **O(log n)** time complexity. Here's the reasoning behind the approach:

1. **Start by defining the middle element:** In a binary search, we begin by checking the middle element. If this element is greater than or equal to its neighbors, it is a peak.

2. **Check the neighbors:** If the middle element is smaller than its right neighbor, this indicates that the peak element lies in the right half of the array. Therefore, we move the search to the right half by updating the left boundary (`low = mid + 1`).

3. **Otherwise, check the left half:** If the middle element is smaller than its left neighbor, the peak element lies in the left half of the array. We then move the search to the left half by updating the right boundary (`high = mid - 1`).

4. **Repeat the process:** We keep halving the array until we find a peak element. Since we are reducing the search space by half with each step, the time complexity is **O(log n)**.

Solution Steps:

- **Step 1:** Initialize the search range with `low = 0` and `high = n - 1` (where `n` is the length of the array).

- **Step 2:** Calculate the middle index `mid = (low + high) / 2`.

- **Step 3:** Compare the middle element with its neighbors:

    - If `arr[mid] >= arr[mid - 1]` and `arr[mid] >= arr[mid + 1]`, we have found a peak element at index `mid`.
    - If `arr[mid] < arr[mid + 1]`, the peak must lie in the right half of the array, so update `low = mid + 1`.
    - If `arr[mid] < arr[mid - 1]`, the peak must lie in the left half of the array, so update `high = mid - 1`.
- **Step 4:** Continue adjusting the search range until a peak element is found.

Example Walkthrough (for input [1, 2, 4, 5, 7, 8, 3]):

- **Step 1**: Initial setup
  `low = 0`, `high = 6` (size of array - 1).
  Calculate `mid = (0 + 6) / 2 = 3`.
  The element at index `3` is `5`.
- **Step 2**: Compare `arr[3] = 5` with its neighbors:
    - `arr[3] = 5` is less than `arr[4] = 7` and greater than `arr[2] = 4`.
    - Since `arr[3] < arr[4]`, the peak is likely in the right half. Therefore, update `low = mid + 1 = 4`.
- **Step 3**: New range `low = 4`, `high = 6`
  Recalculate `mid = (4 + 6) / 2 = 5`.
  The element at index `5` is `8`.
- **Step 4**: Compare `arr[5] = 8` with its neighbors:
    - `arr[5] = 8` is greater than `arr[4] = 7` and `arr[6] = 3`.
    - Since `arr[5] >= arr[4]` and `arr[5] >= arr[6]`, `8` is a peak element, and its index is `5`.

Problem 2

You are given two sorted arrays `a[]` and `b[]`. Your task is to find and return the **median** of the combined array formed by merging `a[]` and `b[]`.

The median is the middle value in an odd-length array, or the average of the two middle values in an even-length array.

You must solve this problem using an efficient approach with a time complexity of **O(log(min(n, m)))**, where `n` and `m` are the lengths of the two arrays.

Justify your answer based on these inputs `a[] = [1, 3, 8]` and `b[] = [7, 9, 10, 11]`)

Answer

We aim to solve this problem using a binary search approach on the smaller array, which gives us an efficient time complexity of **O(log(min(n, m)))**.

Explanation of the Approach:

1. Ensure `a[]` is the Smaller Array:

- We apply binary search on the smaller array to minimize the search space, ensuring **O(log(min(n, m)))** complexity.
- If the length of `a[]` is greater than `b[]`, swap the arrays so that `a[]` is always the smaller one.

2. Partitioning the Arrays:

- We want to partition both arrays such that:
    - The total number of elements in the left partition is half the combined total (`(n + m + 1) // 2`).
    - The elements on the left side of the partition should be less than or equal to the elements on the right side of the partition.

3. Finding the Partition Points:

- Define `mid1` as the partition point in `a[]` and `mid2` as the partition point in `b[]`.
- To ensure the total number of elements on the left side is correct, we calculate `mid2` as: `mid2 = ((n + m + 1) // 2) - mid1`
- This formula for `mid2` ensures that the total number of elements in the left partition is exactly half of the combined length (rounded up in case of an odd total number of elements).

4. Binary Search on the Smaller Array (`a[]`):

- We perform binary search on `a[]` by adjusting `mid1` and calculating `mid2` accordingly to ensure the left and right partitions are valid.
- For the partitions to be valid:
    - `a[mid1-1] <= b[mid2]` (ensures the left part of `a[]` is less than or equal to the right part of `b[]`).
    - `b[mid2-1] <= a[mid1]` (ensures the left part of `b[]` is less than or equal to the right part of `a[]`).

5. Calculating the Median:

- Once the correct partition is found, if the total number of elements is odd, the median is the maximum element in the left partition.
- If the total number of elements is even, the median is the average of the maximum element from the left partition and the minimum element from the right partition.

Example Walkthrough (for `a[] = [1, 3, 8]` and `b[] = [7, 9, 10, 11]`):

Step 1: Ensure `a[]` is the Smaller Array

- `a[]` has 3 elements, and `b[]` has 4 elements. So no need to swap them.

Step 2: Binary Search on `a[]`

- Set `lo = 0` and `hi = 3` (since `a[]` has 3 elements).
- Start the binary search.

Step 3: First Iteration of Binary Search

- Calculate `mid1 = (lo + hi) // 2 = 1`.
- Using the formula for `mid2`: `mid2 = ((n + m + 1) // 2) - mid1 = 3`
- Now, partition the arrays:
    - `a[mid1 - 1] = a[0] = 1`, `a[mid1] = a[1] = 3`.
    - `b[mid2 - 1] = b[2] = 10`, `b[mid2] = b[3] = 11`.

Step 4: Check Partition Validity

- To ensure the partition is valid:
    - We check that `a[mid1-1] <= b[mid2]` (i.e., `1 <= 11`), which is **true**.
    - We check that `b[mid2-1] <= a[mid1]` (i.e., `10 <= 3`), which is **false**.

Since the second condition is false, we need to adjust our partition by increasing `lo`.

Step 5: Second Iteration of Binary Search

- Update `lo = mid1 + 1 = 2`, and re-run the binary search.

Step 6: Second Iteration Calculation

- Now, `mid1 = (2 + 3) // 2 = 2`.
- Using the formula for `mid2`: `mid2 = 2`
- Now, partition the arrays:
  - `a[mid1 - 1] = a[1] = 3`, `a[mid1] = a[2] = 8`.
  - `b[mid2 - 1] = b[1] = 9`, `b[mid2] = b[2] = 10`.

Step 7: Check Partition Validity

- To ensure the partition is valid:
  - We check that `a[mid1-1] <= b[mid2]` (i.e., `3 <= 10`), which is **true**.
  - We check that `b[mid2-1] <= a[mid1]` (i.e., `9 <= 8`), which is **false**.

Again, we adjust `lo`.

Step 8: Final Iteration

- Update `lo = mid1 + 1 = 3`, and re-run the binary search.
- Now, `mid1 = (3 + 3) // 2 = 3`.
- Using the formula for `mid2`: `mid2 = 1`
- Now, partition the arrays:
  - `a[mid1 - 1] = a[2] = 8`, `a[mid1] = a[3] = +inf`.
  - `b[mid2 - 1] = b[0] = 7`, `b[mid2] = b[1] = 9`.
- `a[mid1] = a[3]`, but this would go out of bounds for `a[]` because `a[]` has only 3 elements. So, we treat `a[mid1]` as **positive infinity** (`+inf`), meaning it is larger than any other element.

Step 9: Check Partition Validity

- To ensure the partition is valid:
  - We check that `a[mid1-1] <= b[mid2]` (i.e., `8 <= 9`), which is **true**.
  - We check that `b[mid2-1] <= a[mid1]` (i.e., `7 <= +inf`), which is **true**.

Step 10: Final Calculation

- In this case, the correct partition results in:

- The total number of elements in the combined array is 7 (odd), so the median is the maximum element on the left side of the partition: **Median= max(7,8) = 8**

Problem 3

You are given a hash table with **5 buckets** (indexed from `0` to `4`).
Each bucket uses **separate chaining** to handle collisions (i.e., each bucket contains a linked list of records).

The **hash function** is defined as: h(key) = key % 5

Each record contains a **student ID** and **name**.

Initial Insertions:

The following student records are inserted **in order**:

Student ID	Name
7	Alice
12	Bob
17	Carol
3	David
8	Eva

Tasks:

1) **Build the Hash Table**
   a) Use the given hash function to place each student into the correct bucket.
   b) For each insertion, **show the bucket index** and whether a **collision occurs**.
   c) Draw the **final state** of the hash table, showing linked lists where collisions happen.

2) **Perform Operations** Using the final hash table:
   a) **Search for student ID 12**:
      - Which bucket is accessed?
      - How is the student found?

   b) **Insert a new student**: (ID = 22, Name = Frank)
      - Which bucket does this go to?
      - Does a collision occur?
      - Update the hash table accordingly.

   c) **Delete student ID 17**:
      - Which bucket is accessed?
      - Explain how the record is removed from the linked list.
      - Show the updated state of the bucket.

<u>Answer</u>

1. Build the Hash Table

Insertion 1: ID = 7, Name = Alice

```
h(7) = 7 % 5 = 2
```

- Bucket 2 is empty → Insert **Alice** at index 2.

Insertion 2: ID = 12, Name = Bob

```
h(12) = 12 % 5 = 2
```

- Bucket 2 already contains Alice → **Collision!**
- Insert **Bob** into the linked list at index 2 (e.g., append after Alice).

Insertion 3: ID = 17, Name = Carol

```
h(17) = 17 % 5 = 2
```

- Bucket 2 already has Alice → Bob → **Collision again!**
- Insert **Carol** at the end of the list in bucket 2.

Insertion 4: ID = 3, Name = David

`h(3) = 3 % 5 = 3`

- Bucket 3 is empty → Insert **David** at index 3.

Insertion 5: ID = 8, Name = Eva

`h(8) = 8 % 5 = 3`

- Bucket 3 has David → **Collision!**
- Insert **Eva** after David in the linked list at index 3.

Final Hash Table Structure:

Index	Linked List (Chained Records)
0	(Empty)
1	(Empty)
2	Alice → Bob → Carol
3	David → Eva
4	(Empty)

2. Perform Operations

a) Search for Student ID = 12

- Compute hash: `h(12) = 12 % 5 = 2`
- Go to bucket 2: Alice → Bob → Carol
- Traverse the linked list:
    - Check Alice (ID 7): Not a match

    - Check Bob (ID 12): Match found
- **Result**: Student found → Name: **Bob**

b) Insert Student ID = 22, Name = Frank

- Compute hash: `h(22) = 22 % 5 = 2`
- Bucket 2 already has: Alice → Bob → Carol → **Collision**
- Insert **Frank** at the end of the list in bucket 2

Updated Bucket 2: Alice → Bob → Carol → Frank

c) Delete Student ID = 17

- Compute hash: `h(17) = 17 % 5 = 2`
- Traverse linked list in bucket 2:
  Alice → Bob → Carol → Frank
- Locate Carol (ID 17)
- Remove Carol from the list by adjusting Bob's `next` pointer to skip Carol

Updated Bucket 2: Alice → Bob → Frank

Updated Hash Table Structure After All Operations:

Index	**Linked List (Chained Records)**
0	(Empty)
1	(Empty)
2	Alice → Bob → Frank
3	David → Eva
4	(Empty)

Problem 4

You are tasked with selecting a **hash table collision resolution technique** for an application that stores **user IDs** as keys and **user information** as values. The two techniques you are considering are:

1. **Separate Chaining (Open Hashing)**: Uses linked lists for handling collisions.
2. **Open Addressing (Closed Hashing)**: Resolves collisions by searching for the next available slot.

Task:

1. **Efficiency Analysis**: Which technique is more efficient when the hash table is **sparsely populated** (low load factor) versus when it is **nearly full** (high load factor)? Discuss their performance in these two scenarios.
2. How does each technique perform in scenarios with **frequent deletions**? Consider how the deletion operation might affect **open addressing** (e.g., clustering issues).
3. If the application involves **frequent insertions**, which collision resolution technique would you recommend? Why?

Answer

Efficiency Analysis:

- **Sparsely Populated (Low Load Factor)**: When the hash table is sparsely populated, **open addressing** tends to be more efficient because there are fewer collisions, and most slots are empty. This results in fast insertions and lookups, as the probe sequence for finding an available slot is usually short. **Separate chaining**, on the other hand, requires more memory because each bucket stores a linked list, which could lead to increased memory overhead.

- **Nearly Full (High Load Factor)**: When the table is near full, **separate chaining** performs better. Open addressing suffers from increased clustering (especially with linear probing), which significantly degrades performance as the load factor grows. Separate chaining handles high load factors more gracefully since linked lists can grow dynamically within each bucket without the need for resizing or probing. This results in more stable performance, even as the table becomes nearly full.

Frequent Deletions:

- **Open Addressing**: Deletion in open addressing is more complex because after a deletion, there might be gaps in the probe sequence. This can disrupt the search for future elements, especially in **linear probing**, leading to **clustering**. While this can be mitigated by using **double hashing** or **quadratic probing**, the performance still degrades as deletions can cause additional probing and rehashing.

- **Separate Chaining**: Deletion is straightforward in separate chaining because elements are stored in linked lists, and a deletion simply removes an element from the list at the corresponding bucket. However, if many deletions occur, the linked lists could become sparse, and managing the memory for empty slots may require additional cleanup operations. Overall, separate chaining tends to handle deletions more efficiently than open addressing.

Recommendation for Frequent Insertions:

- For **frequent insertions**, **separate chaining** is recommended. This is because, with open addressing, frequent insertions can lead to a **high load factor**, causing clustering issues that degrade performance, especially as the table fills up. Separate chaining allows for more flexibility in handling insertions because the linked lists can grow dynamically, and each bucket can hold multiple values without affecting other elements. Additionally, separate chaining does not suffer from clustering, which can be a significant issue with open addressing when insertion rates are high.

Problem 5

Imagine you're part of a company's performance evaluation team. The organization is structured hierarchically like a **binary tree**, where:

- Each **node** represents an **employee**, including team leads and individual contributors.
- Each **employee node** contains a **performance score** for a specific quarter.
- A **leaf node** is an individual contributor with no subordinates.
- **Internal nodes** (team leads or managers) manage up to two direct reports (left and right child in the tree).

Now, to **validate fair performance reporting**, the company defines a rule:

> "Each team lead's performance score must equal the **total performance score** of their **entire team** (i.e., the sum of their direct subordinates' performance scores and all levels below)."

Such a tree is called a **"Sum Tree"**.

Task:

Your job is to **verify** whether the company's hierarchy satisfies the **Sum Tree property**, i.e., whether the performance score of every team lead equals the **sum of the scores of all their subordinates**.

You must describe:

- The **algorithm runs in O(n) time complexity** you'd use to validate the Sum Tree property.
- A **step-by-step walkthrough** using the provided example tree.

Note:

- A leaf employee (individual contributor) is considered valid by default.
- An empty team (no employees) is considered to contribute 0 to the total.

Example Hierarchy (Binary Tree):

```
            Alice(190)

          /            \

    Bob(50)          Charlie(70)

                   /            \

              Dana(30)       Eva(40)
```

Expected Output: Yes, this is a valid Sum Tree.

<u>Answer</u>

1. Algorithm Explanation:

We use a **bottom-up (post-order) recursive approach** to check whether the binary tree is a Sum Tree **in O(n)** time:

Key Ideas:

1. **Leaf Nodes** (individual contributors) are **always valid** Sum Trees.

2. **Empty Nodes** (no employee/subtree) contribute `0` to the total sum.

3. For any **non-leaf node** (manager):
    - First, **recursively check** if the left and right subtrees are Sum Trees.
    - If either is not, the whole tree isn't.
4. If both subtrees are valid:
    - We can **calculate the total sum in O(1)** using the following:
        - If a child is a leaf → use its `value`
        - If a child is an internal node and a Sum Tree → its subtree sum is **2 × node value** (because child's value = sum of its subordinates)
5. Check if `node.value == left_sum + right_sum`. If yes, this node is valid.

Why it's O(n):

- Every node is visited **exactly once**, and at each step, we do constant-time checks and calculations.
- No repeated subtree sum computations.

2. Step-by-Step Process:

Step 1: Start with Leaf Nodes

- **Dana(30)** and **Eva(40)** are leaves → **Valid Sum Trees**
    - Subtree sum of Dana = 30
    - Subtree sum of Eva = 40

Step 2: Charlie(70)

- Charlie is not a leaf.
- Left = Dana (leaf) → contributes 30
- Right = Eva (leaf) → contributes 40
- Expected: `Charlie.value = 30 + 40 = 70` → **Valid**

- Since both children are leaves, their values are used directly.
- **Subtree sum under Charlie = 70 + 30 + 40 = 140**, but for the algorithm we return only **70** to its parent, because Charlie is treated as one unit now (since it passed the check).

Step 3: Bob(50)

- Bob is a leaf (no children) → **Valid Sum Tree**
- Subtree sum under Bob = 50

Step 4: Alice(120)

- Left = Bob (leaf) → contributes 50
- Right = Charlie (a non-leaf but confirmed Sum Tree) → use `2 × Charlie.value = 2 × 70 = 140`
- Check: `Alice.value = 50 + 140 = 190` → **Valid**

Final Output: Yes, this is a valid Sum Tree.

Problem 6

We have a network of devices represented by a binary tree, where each device is connected to others, with parent-child relationships indicating direct connections. When **a failure** (such as a network outage, malfunction, or device failure) **occurs at a particular device** (the target node), **the network can no longer reach the failed device or its directly connected devices.** Our goal is to determine the order in which devices fail as the failure progresses throughout the network. By determining this order, we can understand how the failure will affect connectivity, identify critical devices whose failure could cause widespread disruption, and prioritize actions to prevent further failures or mitigate their impact on the overall network.

1. **Describe the Algorithm** you would use to simulate the failure spreading through the binary tree.
2. **Justify your approach** by showing the steps with the following example.

Example Tree:

```
          12
        /    \
     13       10
             /    \
          14       15
         /  \     /  \
       21   24  22   23
```

Target Node = 14

Expected Output:

```
14

21, 24, 10

15, 12

22, 23, 13
```

Answer

Algorithm Explanation:

To simulate the failure spreading in the binary tree, follow these steps:

1. **Find the Target Node**:
    - First, we need to locate the target node in the tree. We can use **Depth-First Search (DFS)** or **Breadth-First Search (BFS)** to traverse the tree and find the node where the failure will start.
    - We will use **DFS** for this explanation.
2. **Failure Spreads to Connected Nodes**:
    - Once the target node is found, print the target node (this is the first failed device).
    - The failure spreads to all directly connected nodes. These are the **parent** (if the target node is not the root) and **children** of the current node.

3. **Use a Queue for Level-by-Level Spread**:
    - We will use a **queue** to manage the spread of failure. After the initial failure starts at the target node, we add all its neighbors (children and parent) to the queue.
    - For each subsequent level, we process the queue, print the devices that fail simultaneously, and add their children (if any) to the queue for the next step.
4. **Repeat the Process**:
    - This process continues until all devices in the network have failed, i.e., until the queue is empty.

Step-by-Step Process with Example:

Step 1: Find the Target Node (14)

DFS Traversal:

- We perform a **DFS** traversal to find the target node. Start at the root (12), traverse down to the left child (13), and then move to the right child (10).
- From node **10**, we move down to node **14**, which is the target.
- Once we find node **14**, we print it as it fails first.

Queue state after the failure starts at 14:

- We add **14's** neighbors (parent and children) to the queue.
    - **Left Child (21)**
    - **Right Child (24)**
    - **Parent (10)**

Queue: `[21, 24, 10]`
Output so far: `14`

Step 2: Spread Failure to Neighbors (Level 1)

Failure Spreads to Neighbors:

- The failure spreads to **21, 24, and 10** in the first step.
- These nodes fail simultaneously, so we print them all.
- After spreading, we add their connected neighbors to the queue (if they exist).

Processing the queue:

- **Node 21** has no children, so no new nodes are added to the queue.
- **Node 24** has no children, so no new nodes are added to the queue.
- **Node 10** has **children 15** and **parent 12**.
- We add **15** and **12** to the queue.

Queue state after Level 1:

- **Queue**: `[15, 12]`

Output so far: `14`, `21,24,10`

Step 3: Spread Failure to Neighbors (Level 2)

Failure Spreads to Neighbors of 15 and 12:

- The failure spreads to **15** and **12** in the next level.
- We print them simultaneously.
- After spreading, we add their connected neighbors to the queue (if they exist).

Processing the queue:

- **Node 15** has **children 22, 23**.
    - Add **22** and **23** to the queue.
- **Node 12** has no children but has **parent 13**.
    - Add **13** to the queue.

Queue state after Level 2:

- **Queue**: `[22, 23, 13]`

Output so far: `14`, `21,24,10`, `15,12`

Step 4: Spread Failure to Neighbors (Level 3)

Failure Spreads to Neighbors of 22, 23, and 13:

- The failure spreads to **22**, **23**, and **13**.
- We print these nodes simultaneously.
- After spreading, we add any connected neighbors to the queue.

Processing the queue:

- **Node 22** has no children.
- **Node 23** has no children.
- **Node 13** has no children, as it is a leaf node.

Queue state after Level 3:

- The queue is empty.

Output so far: `14`, `21,24,10`, `15,12`, `22,23,13`