--------: Data Structures and Program Design Assignment :--------

Name: Samadhnya S. Meshram

Roll Number: 239

Section: C

Date of Submission:15/10/25

Problem Set Assigned: 39

1.https://leetcode.com/problems/path-sum/

2.https://leetcode.com/problems/sort-list/

3.https://leetcode.com/problems/invert-binary-tree/

4.https://leetcode.com/problems/linked-list-cycle-ii/

5.https://leetcode.com/problems/binary-tree-cameras/

PROBLEM 1:

Problem link: https://leetcode.com/problems/path-sum/

Problem Title: Path Sum

A.Problem Description:

We are given a binary tree and a target sum a number.We need to find out if there is any path from the root node to a leaf node (a node with no children) such that the sum of all the node values on that path equals the target sum.

If such a path exists print True

if not print False

B.Algorithm /Approach:

1. Start from the root node.

2.If the tree is empty, return false because there's no path.

3. Subtract the current node's value from the target sum.

4.If the current node is a leaf node (no left and right child):

Check if the remaining sum equals 0 (means path sum found).

If yes-> return true.

5. Otherwise, go to both left and right child nodes and repeat the same process.

6. If any one path returns true then the finl answer is true.

7. If all paths are checked and no path matches return false.

C. Time and Space Complexity:

Time Complexity: O(N)

Space Complexity: O(h)- Where h is the height. In the worst case h = n; in the best case h = log(n).

D. dry run example:

```
 17
 / \
 20   3
        \
          5
```

Step 1

Start from root = 17

Remaining sum = 25- 17 = 8

Step 2 (Left Subtree):

Move to left child = 20

Remaining sum = 8- 20 =-12

Leaf node reached but sum not 0

No valid path here

Step 3 (Right Subtree):

Move to right child =

Remaining sum = 8 – 3

Step 4:

Node = 5

Remaining sum = 5- 5 = 0

Leaf node and remaining sum = 0

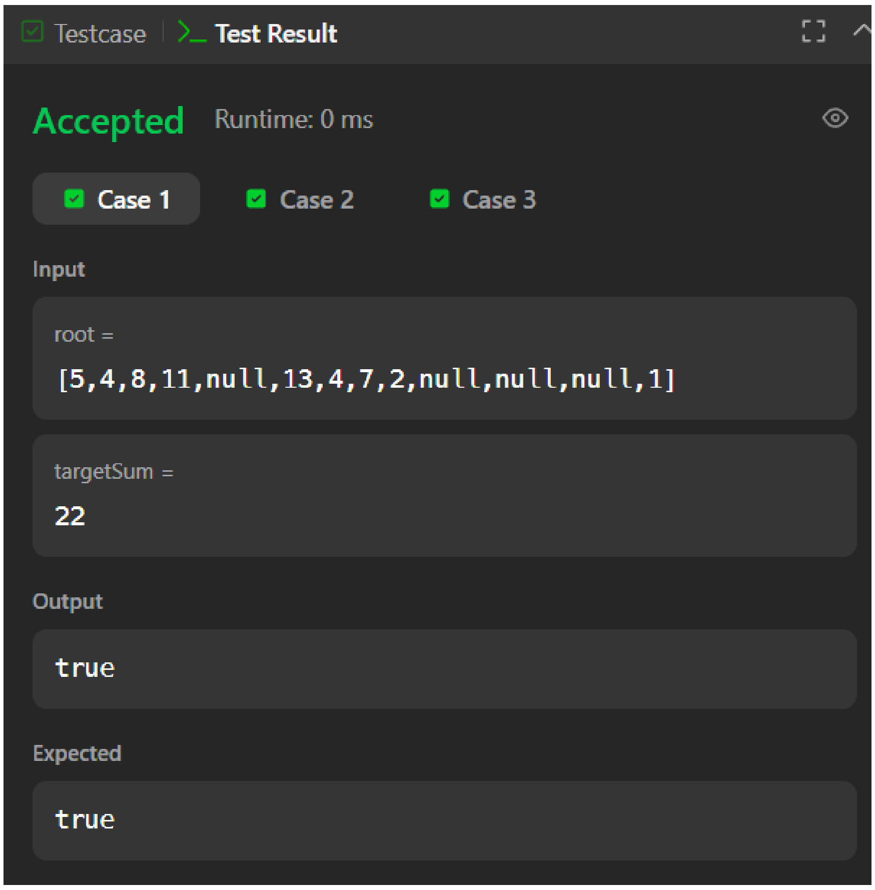
Path found-> (17-> 3-> 5)

O/P: Yes, there is path sum 25.

E.Code Implementation:

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */

// Recursive function to check for path sum
bool hasPathSum(struct TreeNode* root, int targetSum) {
    // If the current node is NULL, return false
    if (root == NULL) {
        return false;
    }

    // If the current node is a leaf node
    if (root->left == NULL && root->right == NULL) {
        // Check if the remaining targetSum equals the leaf's value
        return (targetSum == root->val);
    }

    // Subtract current node's value from targetSum
    int remainingSum = targetSum- root->val;

    // Recurse on left and right subtrees
    return hasPathSum(root->left, remainingSum) || hasPathSum(root-
>right, remainingSum);
}
```

F. O/P screenshot:



Accepted Runtime: 0 ms

Input

root =

```
[5,4,8,11,null,13,4,7,2,null,null,null,1]
```

targetSum =

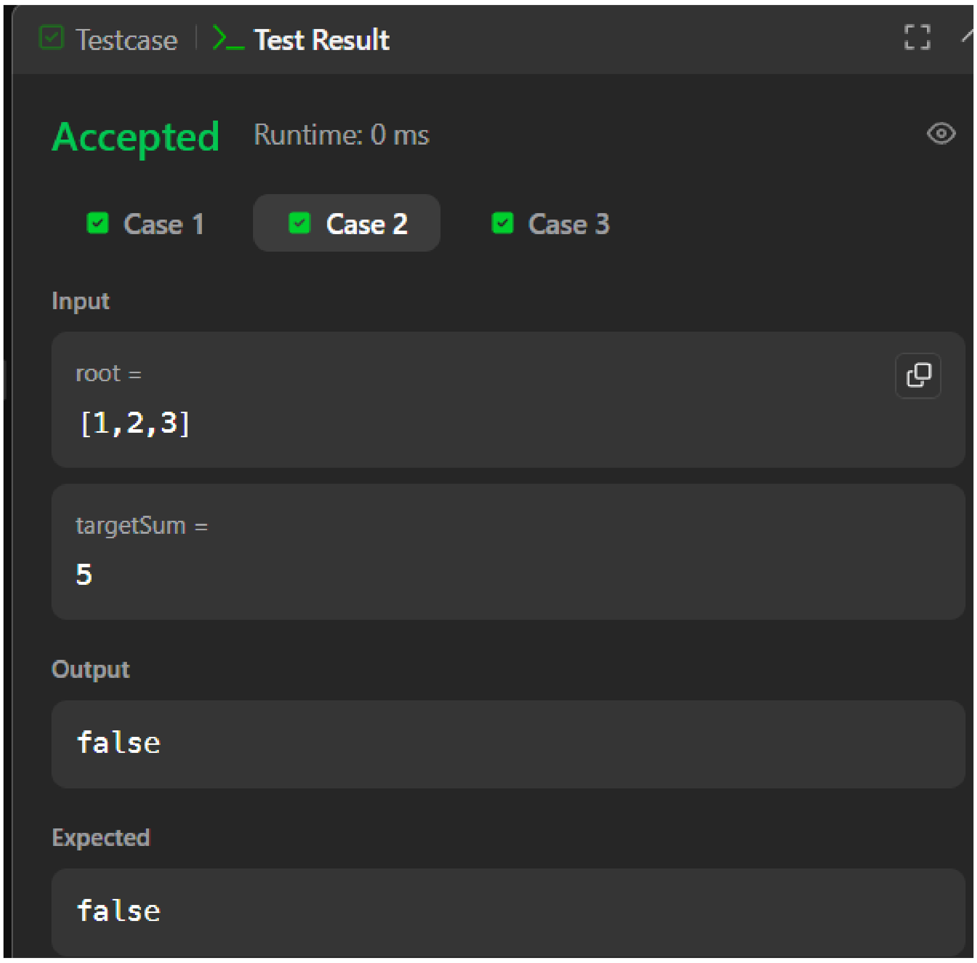
```
22
```

Output

```
true
```

Expected

```
true
```



Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

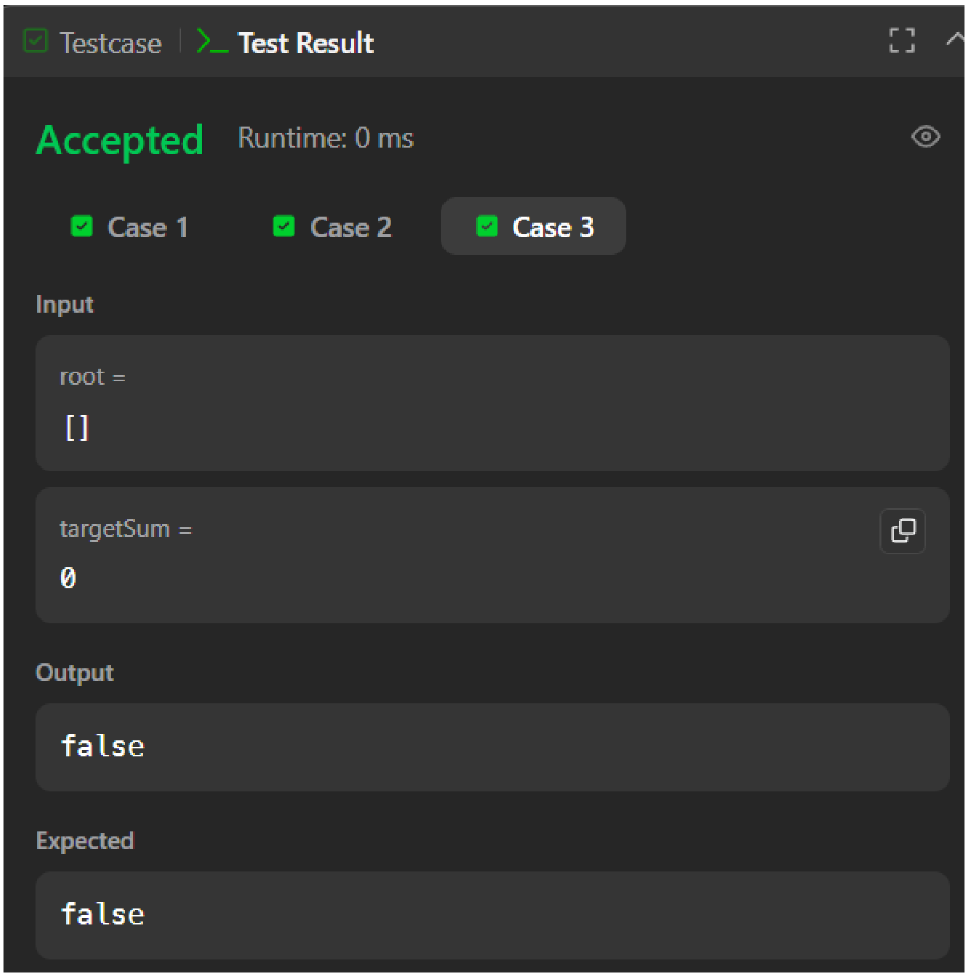
Input

root =
```
[1,2,3]
```

targetSum =
```
5
```

Output
```
false
```

Expected
```
false
```



Accepted Runtime: 0 ms

Input

root =

```
[]
```

targetSum =

```
0
```

Output

```
false
```

Expected

```
false
```

PROBLEM 2:

Problem link: https://leetcode.com/problems/sort-list/

Problem Title: Sort list

A.Problem Description:

We are asked to create a linked list by taking numbers (nodes) as input.
Then, we have to sort the linked list in **ascending order** and print it.
Example: If we enter values: 5 2 8 1, the sorted list should be: 1-> 2->5-> 8.
We use a **linked list** where each node stores a value and a pointer to the next node.

B.Algorithm /Approach:

1. Create a **node structure** with data and next pointer.

2.Take input values and create the linked list using insert() function.

3.To sort, we will use **bubble sort** by swapping values inside nodes.

4.After sorting, we print the list.

C. Time and Space Explaination:

Time Complexity: $O(n^2)$

Space Complexity: O(n)

D. dry run example:

Insert: 5,3,1,4

1. 5 and 3-> swap-> 3-> 5-> 1-> 4

2. 5 and 1-> swap-> 3-> 1-> 5-> 4

3. 5 and 4-> swap-> 3-> 1-> 4-> 5

4. 3 and 1-> swap-> 1-> 3-> 4-> 5

5.3 and 4-> yes

6. 4 and 5-> yes

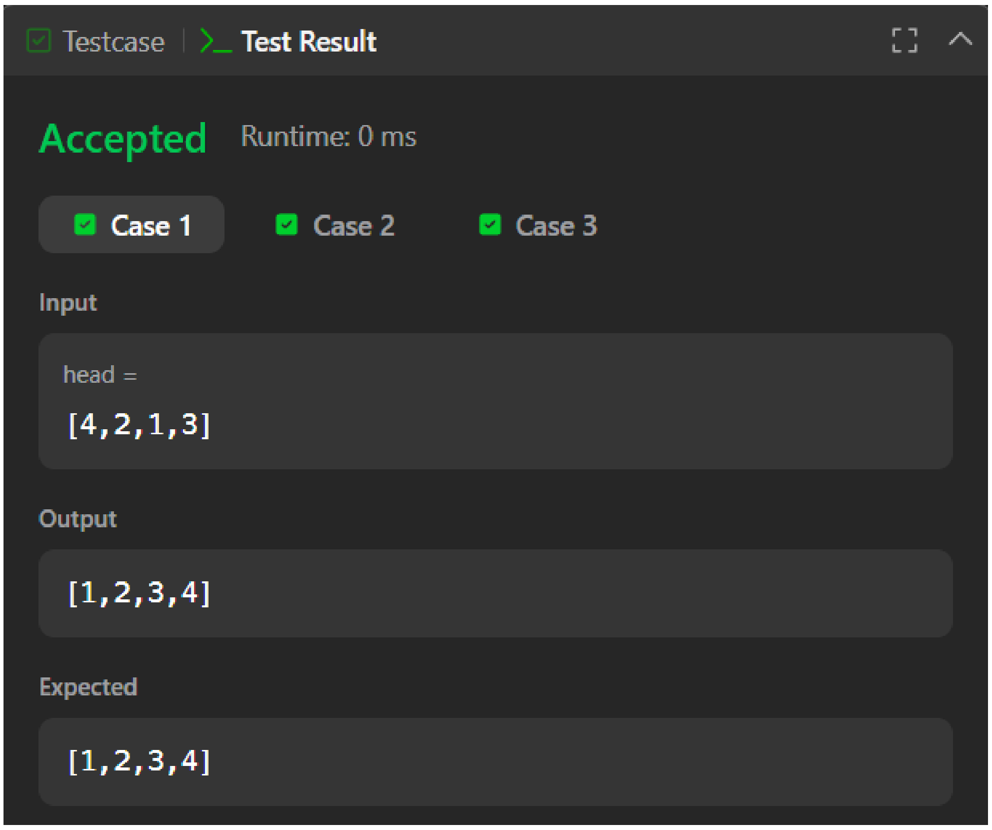
O/P: 1-> 3->4-> 5

E. Code Implementation:

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */

// Helper function to merge two sorted linked lists
struct ListNode* merge(struct ListNode* l1, struct ListNode* l2) {
    // Dummy node to form the new sorted list
    struct ListNode dummy;
    struct ListNode* tail = &dummy;
    dummy.next = NULL;
```

```c
    // Merge nodes one by one in sorted order
    while (l1 && l2) {
        if (l1->val < l2->val) {
            tail->next = l1;
            l1 = l1->next;
        } else {
            tail->next = l2;
            l2 = l2->next;
        }
        tail = tail->next;
    }

    // Attach any remaining nodes
    if (l1) tail->next = l1;
    else    tail->next = l2;

    return dummy.next;
}

// Helper function to find the middle node and split the list
struct ListNode* getMid(struct ListNode* head) {
    struct ListNode* slow = head;
    struct ListNode* fast = head;
    struct ListNode* prev = NULL;

    while (fast && fast->next) {
        prev = slow;
        slow = slow->next;
        fast = fast->next->next;
    }

    // Cut the list into two halves
    if (prev) prev->next = NULL;

    return slow; // this is the middle node
}

// Main sorting function using merge sort
struct ListNode* sortList(struct ListNode* head) {
    // Base case: empty or single node
    if (!head || !head->next) return head;
// Step 1: Split the list into two halves
    struct ListNode* mid = getMid(head);
  // Step 2: Recursively sort both halves
    struct ListNode* left = sortList(head);
    struct ListNode* right = sortList(mid);

    // Step 3: Merge both sorted halves
    return merge(left, right);
}
```

F. O/P Screenshot:



Testcase | Test Result

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

head =

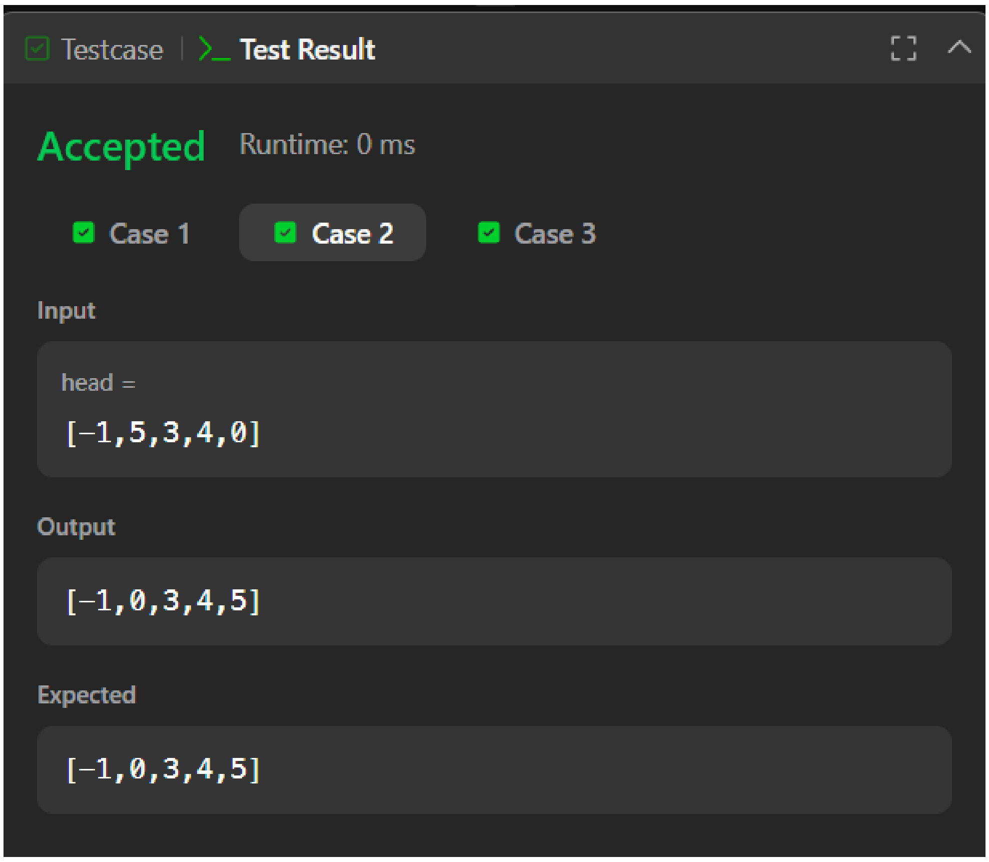
```
[4,2,1,3]
```

Output

```
[1,2,3,4]
```

Expected

```
[1,2,3,4]
```



Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

head =

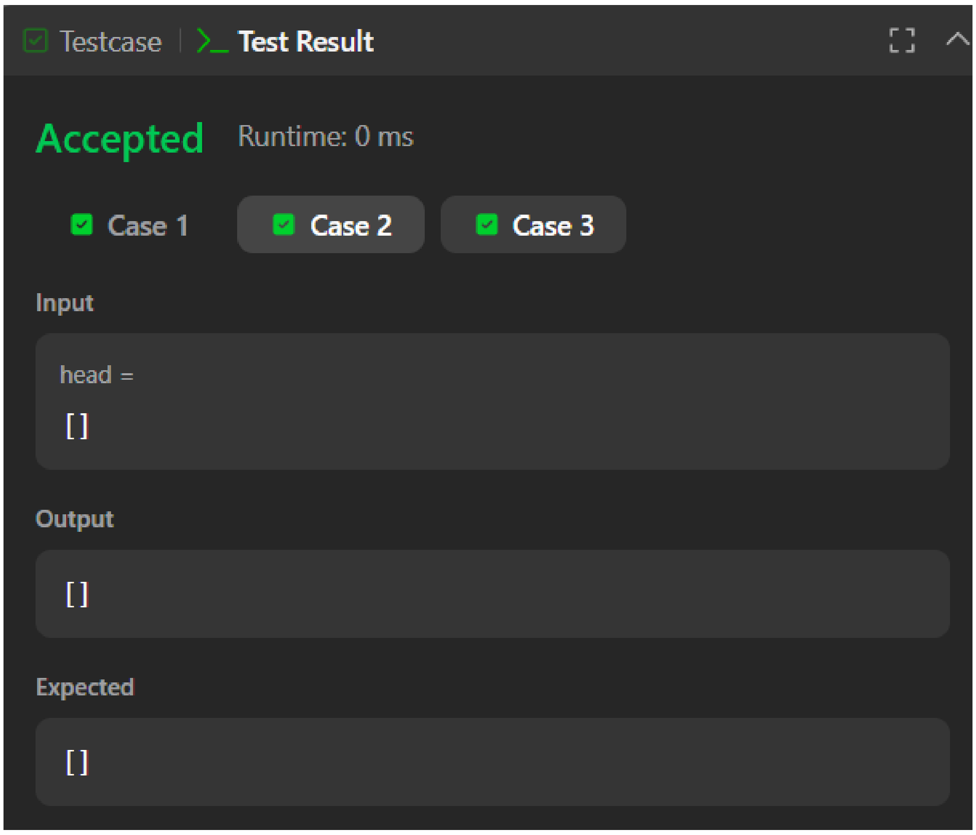
```
[-1,5,3,4,0]
```

Output

```
[-1,0,3,4,5]
```

Expected

```
[-1,0,3,4,5]
```



Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

head =

```
[]
```

Output

```
[]
```

Expected

```
[]
```

PROBLEM 3:

Problem link: https://leetcode.com/problems/invert-binary-tree/

Problem Title: Invert Binary Tree

A Problem Description:

The task is to invert a binary tree, which means converting the tree into its mirror image. This involves swapping the left and right children of every node in the binary tree. You are given the root node of the tree and must return the root of the inverted tree. If the tree is empty (i.e., the root is NULL), you simply return NULL.

B. Algorithm / Approach:

1 Start at the root node.

2 For each node, first recursively invert its left and right subtrees.

3 After the recursive calls, swap the left and right child pointers.

4 Return the current node (which now has its children inverted).

C. Time and Space Complexity:

Time Complexity: O(n)

Space Complexity: O(h)
h is the height of tree. worst case h = n. best case h = log n.

D.dry run example:

```
      4
    / \
  2     7
/ \    / \
1  3    6  9
```

Steps:

Start at root (4).

Recursively invert left subtree (2-> 1,3 becomes 2-> 3,1).

Recursively invert right subtree (7-> 6,9 becomes 7-> 9,6).

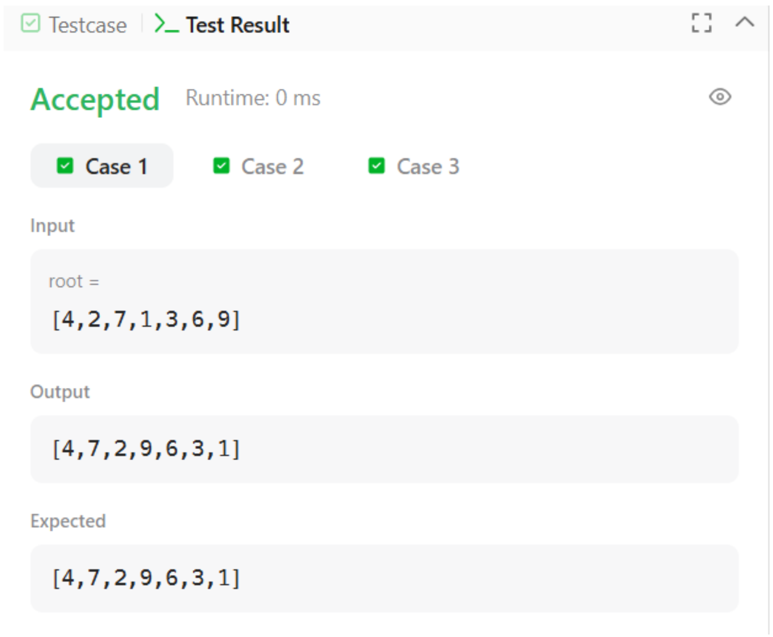
Swap left and right of node 4-> (now left is 7, right is 2).

```
O/P:     4
        / \
       7    2
     / \    / \
   9  6      3  1
```

E. Code Implementation:

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */

// Function to invert a binary tree
struct TreeNode* invertTree(struct TreeNode* root) {
    // Base case: if the current node is NULL, return NULL
    if (root == NULL) {
        return NULL;
    }
  // Recursively invert the left and right subtrees
    struct TreeNode* leftInverted = invertTree(root->left);
    struct TreeNode* rightInverted = invertTree(root->right);
  // Swap the left and right children
    root->left = rightInverted;
    root->right = leftInverted;

    // Return the current node (with its children inverted)
    return root;
}
```

F. O/P screenshot:



Input

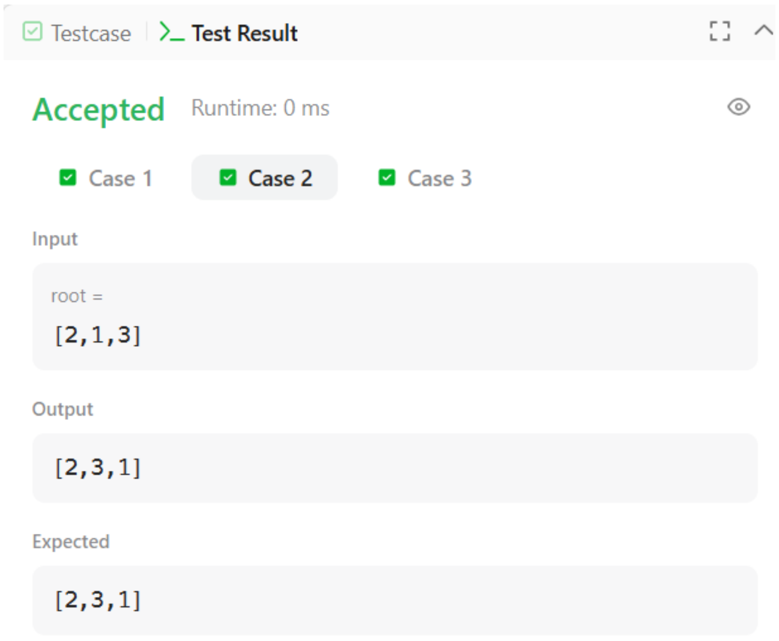
```
root =
[4,2,7,1,3,6,9]
```

Output

```
[4,7,2,9,6,3,1]
```

Expected

```
[4,7,2,9,6,3,1]
```



Input

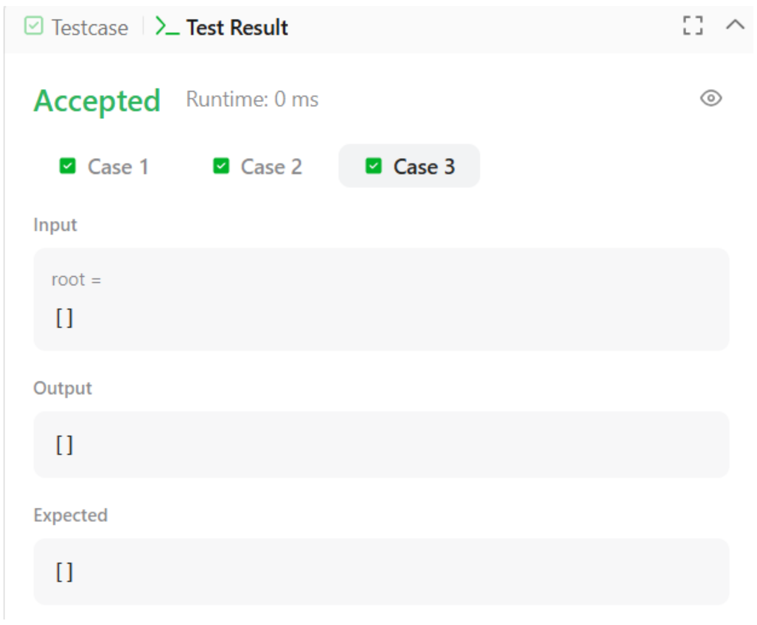
```
root =
[2,1,3]
```

Output

```
[2,3,1]
```

Expected

```
[2,3,1]
```



Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

```
root =
[]
```

Output

```
[]
```

Expected

```
[]
```

Problem 4:

Problem link: https://leetcode.com/problems/linked-list-cycle-ii/

Problem Title: Linked List Cycle II

A Problem Description:

You are given the head of a singly linked list. The list may contain a **cycle**—which means that a node's next pointer may point to a previous node, forming a loop. Your task is to **detect whether a cycle exists**, and if it does, **return the node** where the cycle begins. If there is no cycle, return NULL.

B Algorithm / Approach:

1 Use two pointers: slow (moves 1 step) and fast (moves 2 steps).

2 If there is a cycle, these two pointers will eventually meet inside the loop.

3 If fast or fast->next becomes NULL, the list has no cycle

4 Find the Start of the Cycle

5 Reset one pointer to the head of the list.

6 Move both slow and head one step at a time.

7 The point where they meet is the **start of the cycle.**

C Time and Space Complexity

Time Complexity: O(n)

Space Complexity: O(1)

D. dry run example:

```
List: 3-> 2->  0-> -4
                      |      |
                      <-<-<-
```

Cycle at node with value: 2

Cycle at node with value: 2

1.slow and fast start at head (3).

2.Move slow by 1 and fast by 2:

slow = 2, fast = 0

slow = 0, fast = 2

slow =-4, fast =-4-> cycle detected

Reset one pointer to head (3), keep other at meeting point (-4)

Move both one step at a time:

head = 3, slow =-4

head = 2, slow = 2-> match!

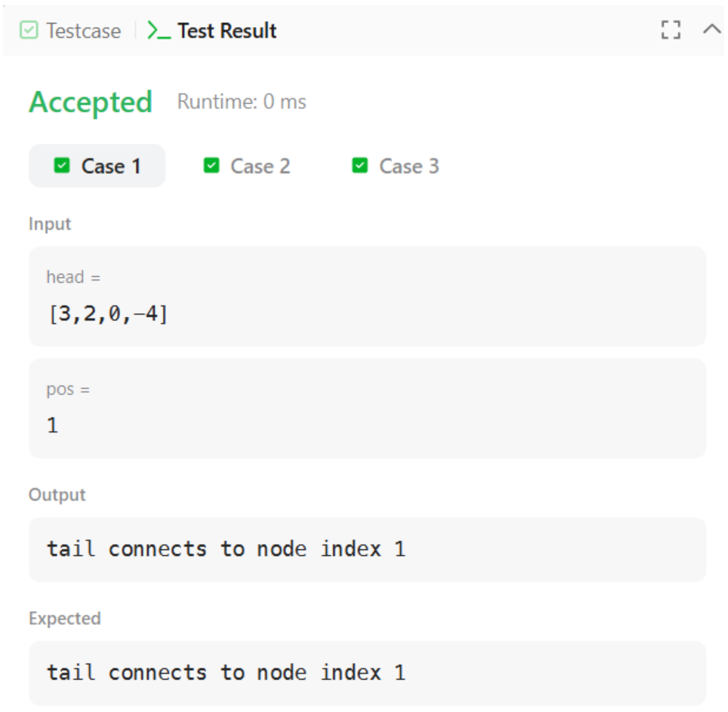
O/p: value 2

E Code Implementation:

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */

struct ListNode *detectCycle(struct ListNode *head) {
   //  if the list is empty or has only one node
   if (head == NULL || head->next == NULL) {
      return NULL;
   }
   struct ListNode* slow = head;
   struct ListNode* fast = head;

   //  Detect if a cycle exists
   while (fast != NULL && fast->next != NULL) {
      slow = slow->next;
      fast = fast->next->next;

      if (slow == fast) {
         // Cycle detected
         // Find the start of the cycle
         struct ListNode* entry = head;
```

```
            while (entry != slow) {
                entry = entry->next;
                slow = slow->next;
            }

            return entry;  // start of the cycle
        }
    }
// If we reach here, no cycle was found
    return NULL;
}
```

F O/P:



Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

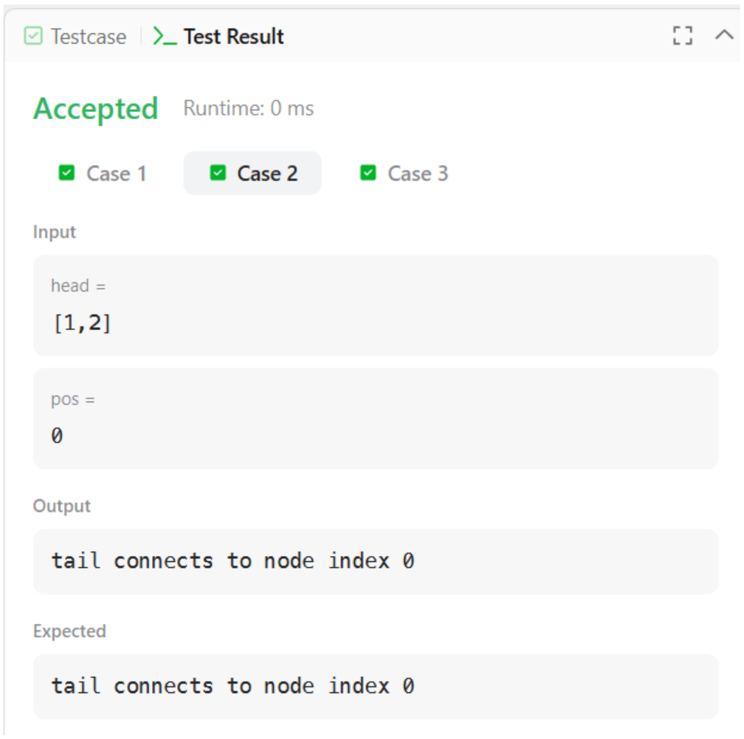
Input

head =
```
[3,2,0,-4]
```

pos =
```
1
```

Output
```
tail connects to node index 1
```

Expected
```
tail connects to node index 1
```



Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

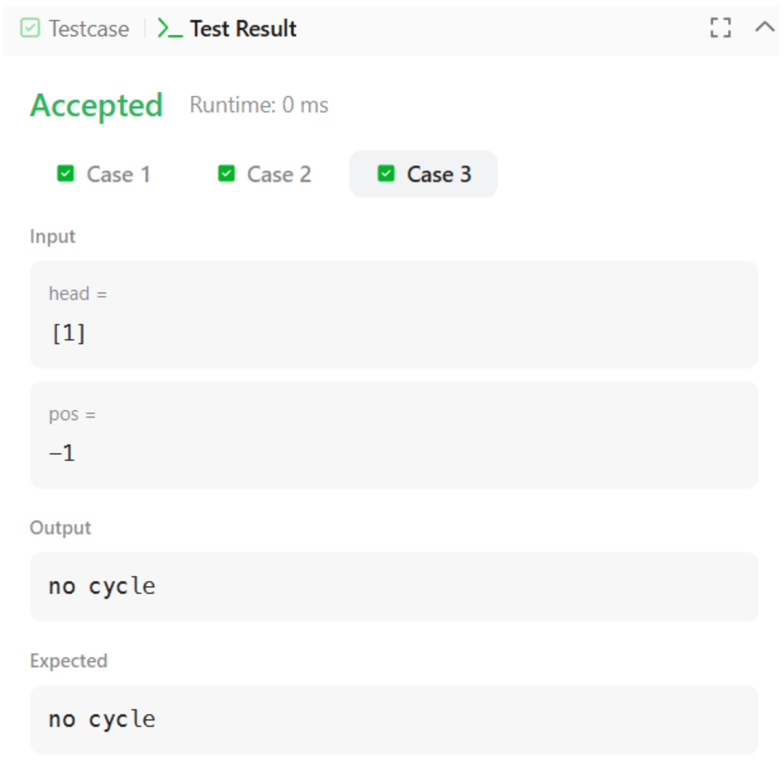
Input

head =
```
[1,2]
```

pos =
```
0
```

Output
```
tail connects to node index 0
```

Expected
```
tail connects to node index 0
```


Testcase | Test Result

Accepted Runtime: 0 ms

✓ Case 1 ✓ Case 2 ✓ Case 3

Input

head =

```
[1]
```

pos =

```
-1
```

Output

```
no cycle
```

Expected

```
no cycle
```

Problem 5: Binary tree cameras:

Problem link: https://leetcode.com/problems/binary-tree-cameras/

Problem Title: Linked List Cycle II

A Problem Description:

You are given the root of a binary tree. Each camera placed on a node can monitor:

The node itself,

Its **immediate left and right children**,

And its **parent**.

The goal is to **place as few cameras as possible** such that **every node in the tree is monitored**.

If no cameras are placed, and a node is left unmonitored, that's invalid. You must return the **minimum number of cameras required** to cover the entire tree.

B Algorithm / Approach:

Main Idea:

Every node can be in **one of three states**:

1. **State 0 (NOT COVERED)** – This node is not monitored and doesn't have a camera.
2. **State 1 (HAS CAMERA)** – This node has a camera installed.
3. **State 2 (COVERED)** – This node is monitored by its child or parent, but does not have a camera itself.

Traversal Logic:

- Traverse the left and right children first (post-order).
- If **either child is not covered (state 0)**-> install a camera at the current node (return state 1).
- If **either child has a camera (state 1)**-> this node is already covered (return state 2).
- If **both children are covered (state 2)** but do **not** have cameras-> this node is **not covered** (return state 0).

Final Step:

- After the entire DFS traversal, if the **root is not covered**, place a final camera at the root.

C. Time and Space Complexity:

- **Time Complexity**: O(n)
- **Space Complexity**: O(h)
  h is the height of the tree. worst case, h = n; best case, h = log n

D. Dry Run Example:

```
I/p:      0

        /   \

      0       0

                \

                  0
```

Steps:1Start at the bottom-most node (leftmost leaf of left subtree)-> returns **NOT COVERED (0)**.

2 Parent of that leaf sees child is not covered-> **places a camera**-> returns **HAS CAMERA (1)**.

3 Move to right subtree's child (0)-> no children-> **NOT COVERED (0)**.

4 Parent (0) sees child not covered-> **places another camera**-> returns **HAS CAMERA (1)**.

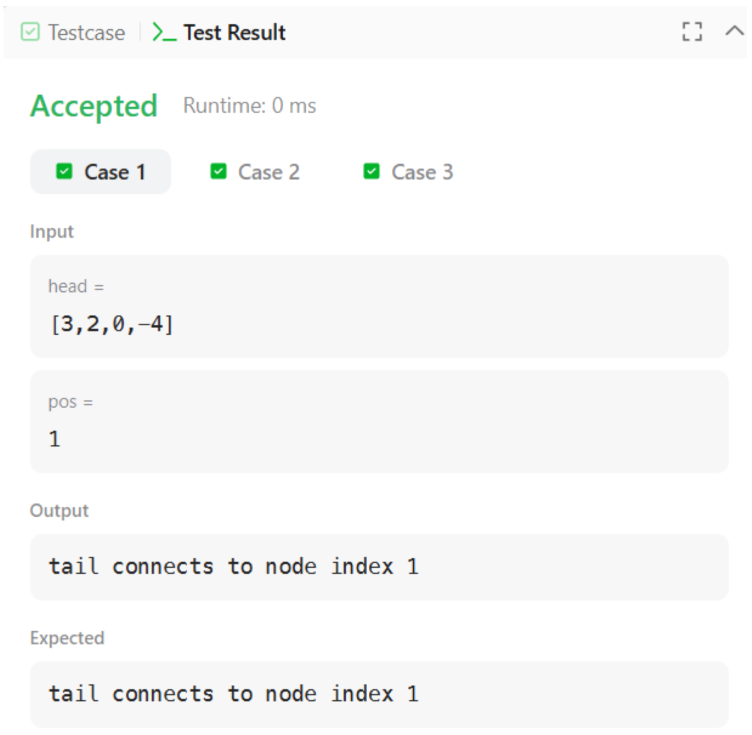
5 Root has both children **covered**, so it is already **covered**.

o/p: Output: **2 cameras needed.**

E. Code Implementation:

```
// Main function to return minimum number of cameras needed
int minCameraCover(struct TreeNode* root) {
    cameraCount = 0;
    // If root is not covered after DFS, place a camera at root
    if (dfs(root) == 0) {
        cameraCount++;
    }   return cameraCount;
}
```

F o/p:

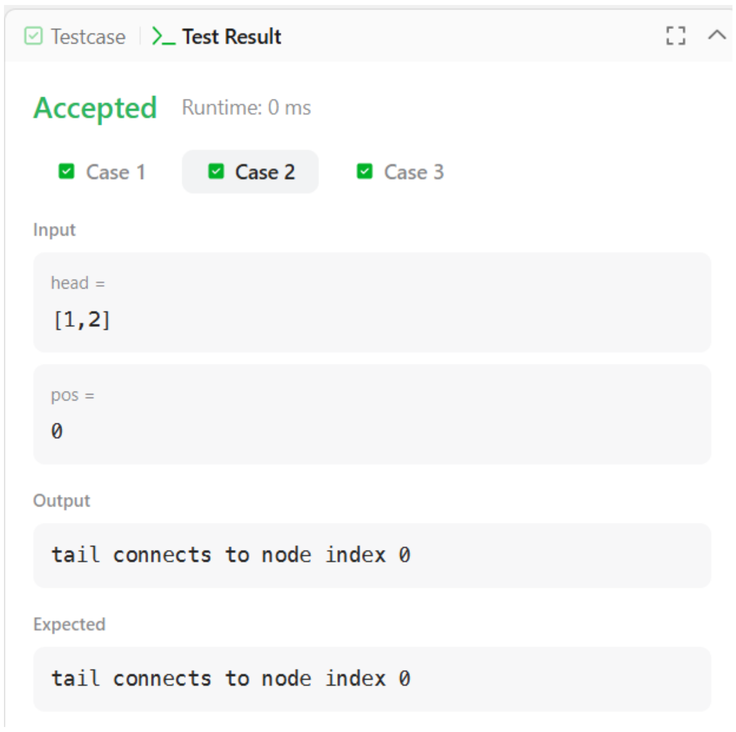


Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

head =
```
[3,2,0,-4]
```
pos =
```
1
```
Output
```
tail connects to node index 1
```
Expected
```
tail connects to node index 1
```

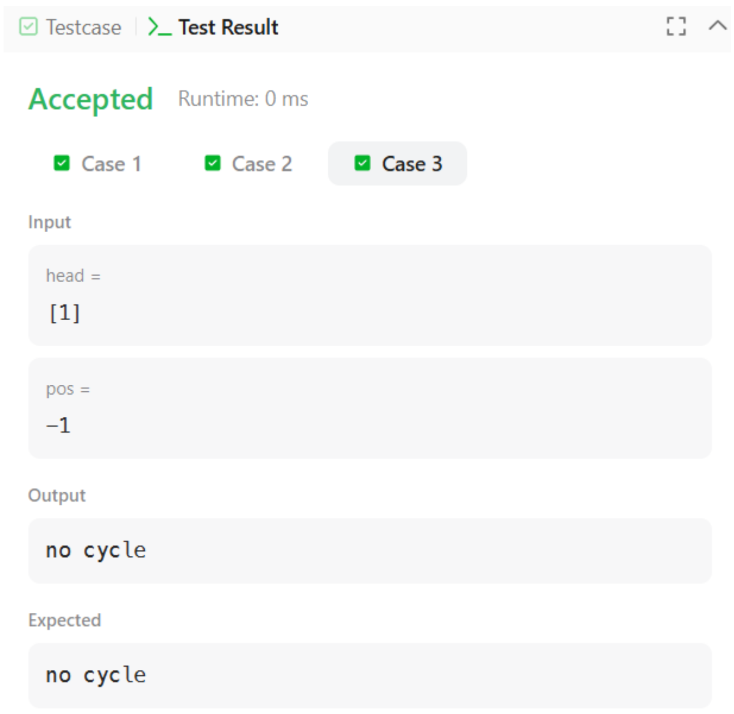


Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

head =
```
[1,2]
```
pos =
```
0
```
Output
```
tail connects to node index 0
```
Expected
```
tail connects to node index 0
```



Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

head =
```
[1]
```

pos =
```
-1
```

Output
```
no cycle
```

Expected
```
no cycle
```

Summary of Learning :

I learned key concepts in data structures and algorithms through problems like Path Sum, where I used DFS to find a target sum in a binary tree, and Sort List, where I applied merge sort on a linked list. I also inverted a binary tree using recursion, and solved the Binary Tree Cameras problem using a greedy DFS approach to place the minimum number of cameras efficiently.

Self-Attestation Declaration

I hereby declare that this code and report are written and understood by me.

I have not copied from others, online sources, or used AI tools such as ChatGPT, GitHub Copilot, or similar systems to generate my solutions.

I am ready to explain any part of my code during viva or evaluation.

Signature: *Samadhnya* **Date:** 15/10/25