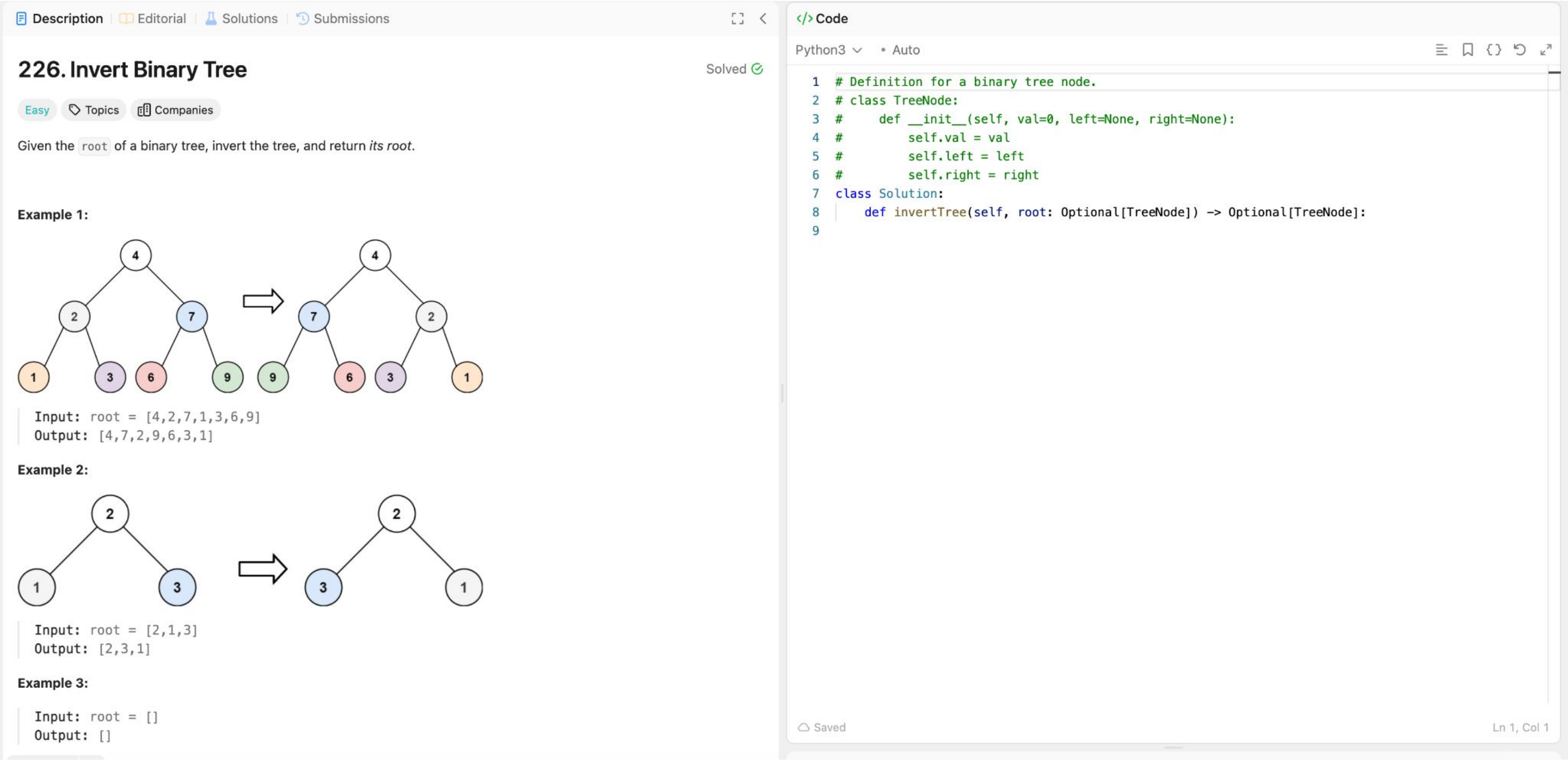


226. Invert Binary Tree

Given the `root` of a binary tree, invert the tree, and return *its root*.

Example 1:

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

Example 2:

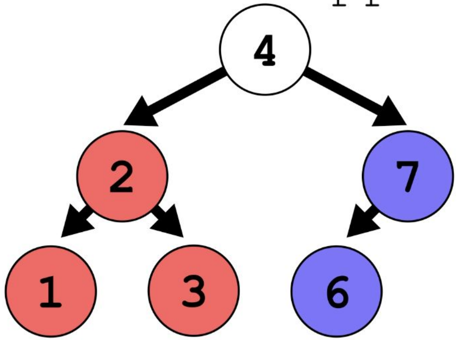
```
Input: root = [2,1,3]
Output: [2,3,1]
```

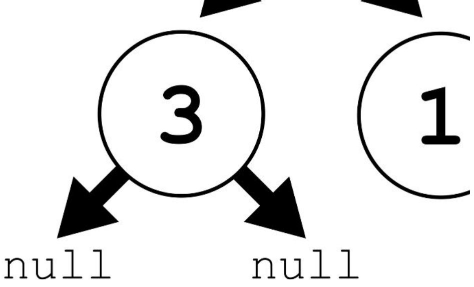
Example 3:

```
Input: root = []
Output: []
```

```python
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def invertTree(self, root: Optional[TreeNode]) -> Optional[TreeNode]:

```

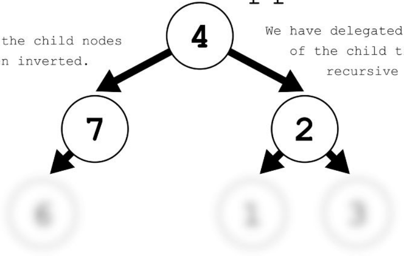
Recursive Approach





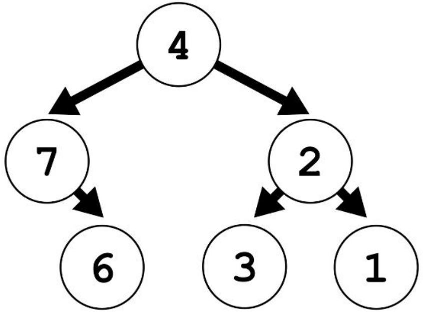
Play

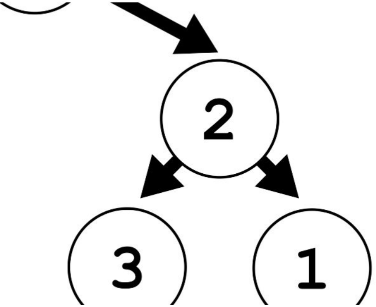
Recursive Approach

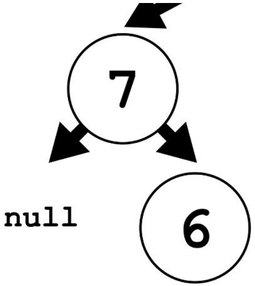


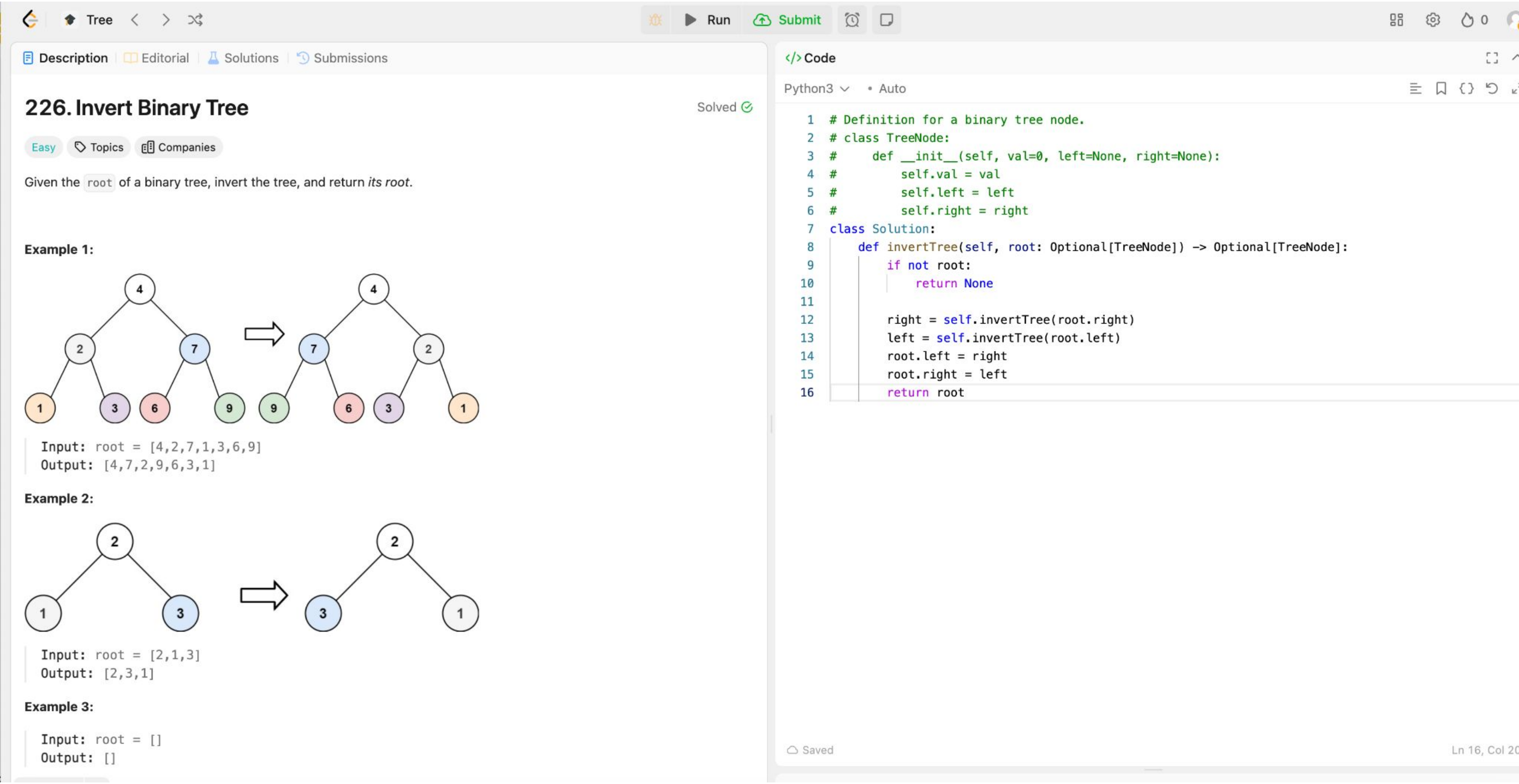
Assume that the child nodes have been inverted.

We have delegated the inversion of the child trees to our recursive calls.









226. Invert Binary Tree

Given the `root` of a binary tree, invert the tree, and return *its root*.

Example 1:

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

Example 2:

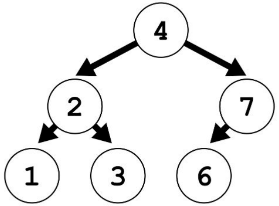
```
Input: root = [2,1,3]
Output: [2,3,1]
```

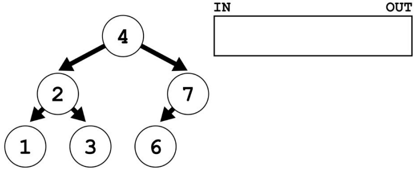
Example 3:

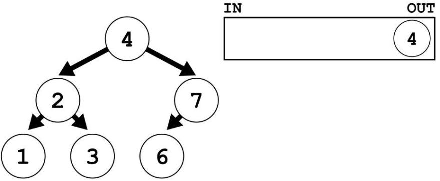
```
Input: root = []
Output: []
```

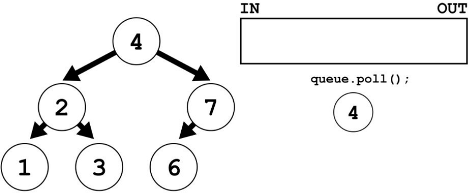
```python
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def invertTree(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
        if not root:
            return None

        right = self.invertTree(root.right)
        left = self.invertTree(root.left)
        root.left = right
        root.right = left
        return root
```

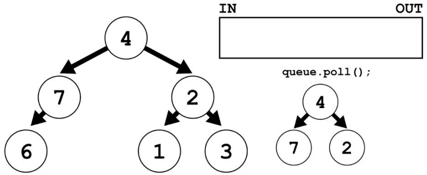
Iterative Approach

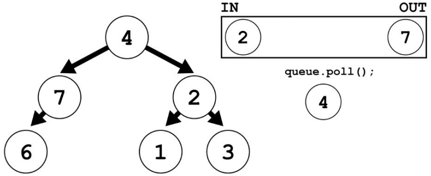


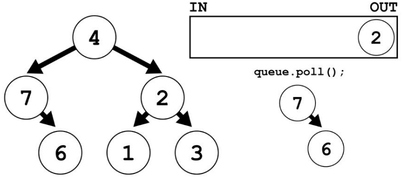


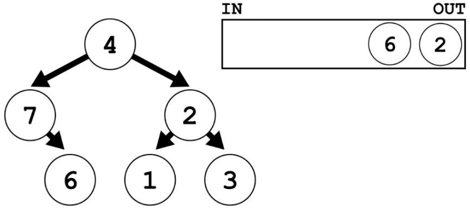


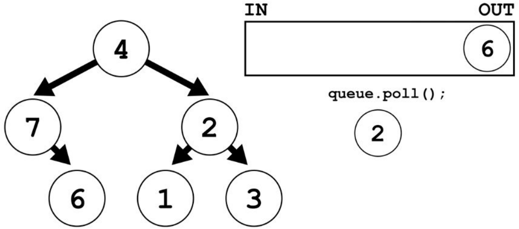


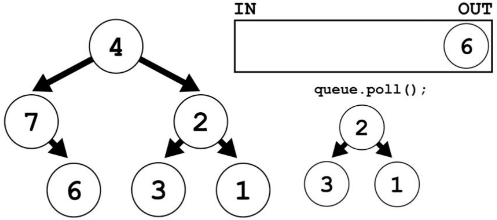


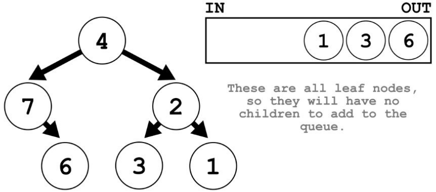




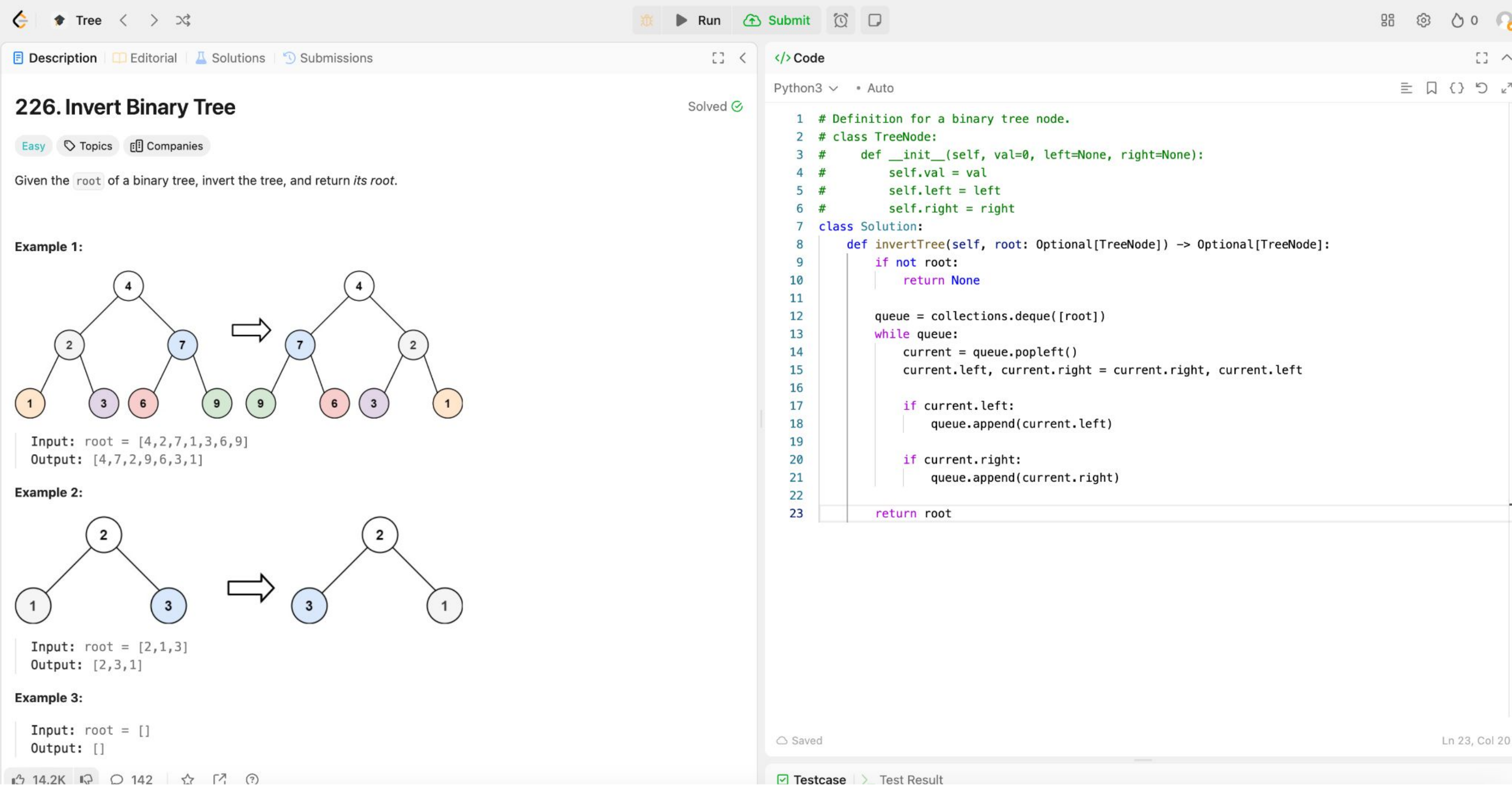








These are all leaf nodes, so they will have no children to add to the queue.



226. Invert Binary Tree

Given the `root` of a binary tree, invert the tree, and return *its root*.

Example 1:

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

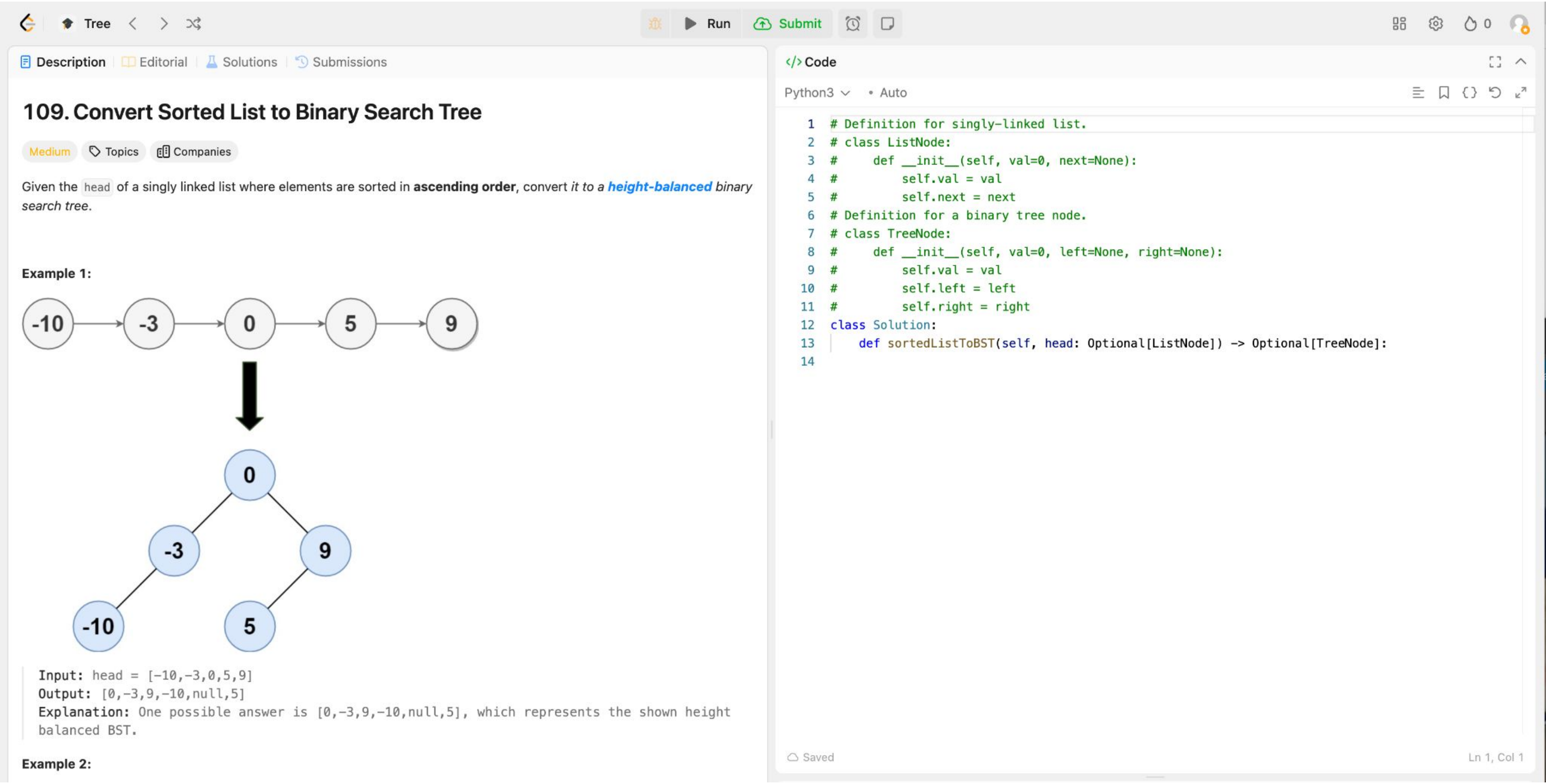
Example 2:

```
Input: root = [2,1,3]
Output: [2,3,1]
```

Example 3:

```
Input: root = []
Output: []
```

```python
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def invertTree(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
        if not root:
            return None

        queue = collections.deque([root])
        while queue:
            current = queue.popleft()
            current.left, current.right = current.right, current.left

            if current.left:
                queue.append(current.left)

            if current.right:
                queue.append(current.right)

        return root
```



109. Convert Sorted List to Binary Search Tree

Medium · Topics · Companies

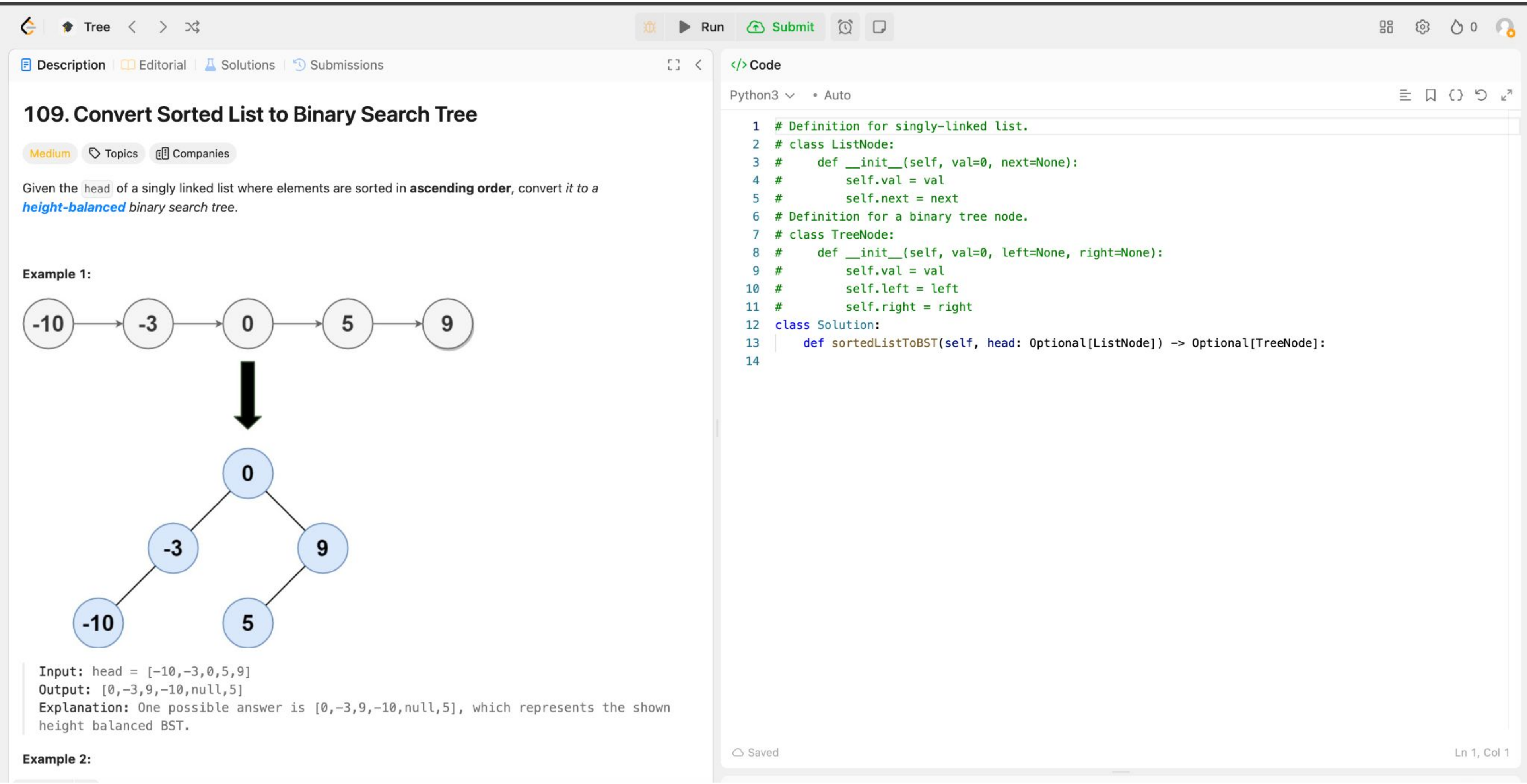
Given the `head` of a singly linked list where elements are sorted in **ascending order**, convert *it to a* ***height-balanced*** *binary search tree*.

Example 1:

```
Input: head = [-10,-3,0,5,9]
Output: [0,-3,9,-10,null,5]
Explanation: One possible answer is [0,-3,9,-10,null,5], which represents the shown height
balanced BST.
```

Example 2:

```python
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def sortedListToBST(self, head: Optional[ListNode]) -> Optional[TreeNode]:

```



109. Convert Sorted List to Binary Search Tree

Medium Topics Companies

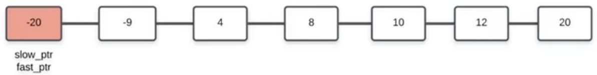
Given the `head` of a singly linked list where elements are sorted in **ascending order**, convert *it to a* ***height-balanced*** *binary search tree*.

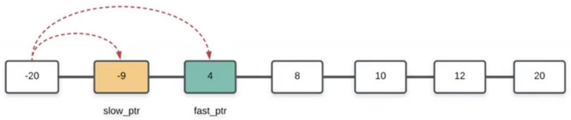
Example 1:

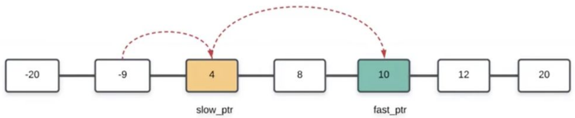
```
Input: head = [-10,-3,0,5,9]
Output: [0,-3,9,-10,null,5]
Explanation: One possible answer is [0,-3,9,-10,null,5], which represents the shown
height balanced BST.
```

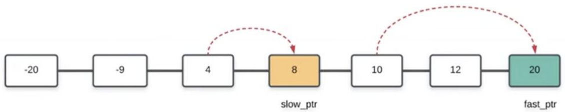
Example 2:

```python
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def sortedListToBST(self, head: Optional[ListNode]) -> Optional[TreeNode]:

```

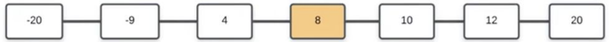








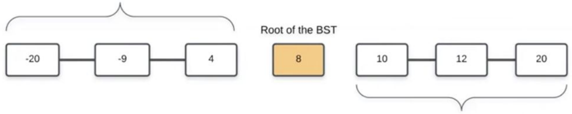
Root of the BST

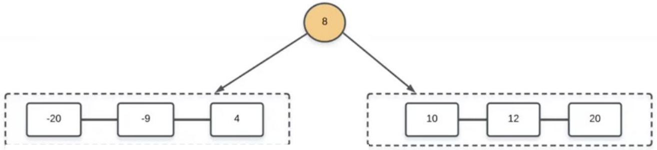


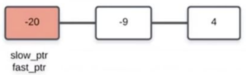
Left Half

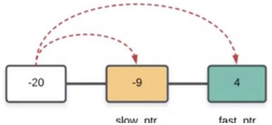
Root of the BST

Right Half

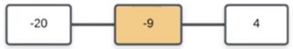


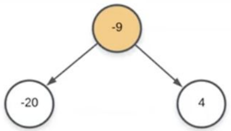


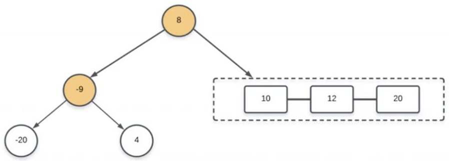


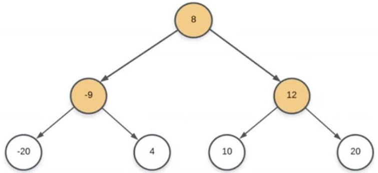


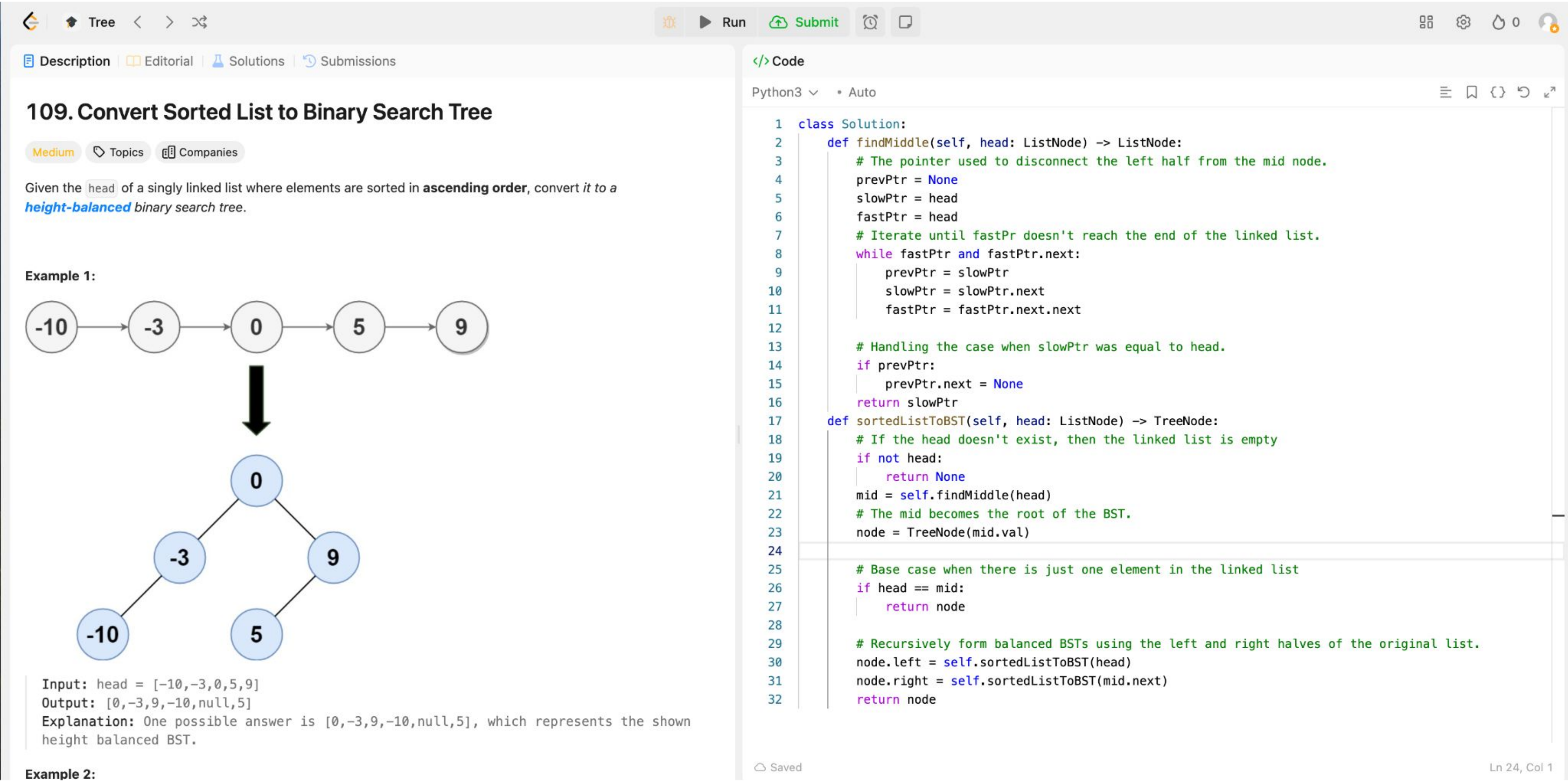
Middle Element











109. Convert Sorted List to Binary Search Tree

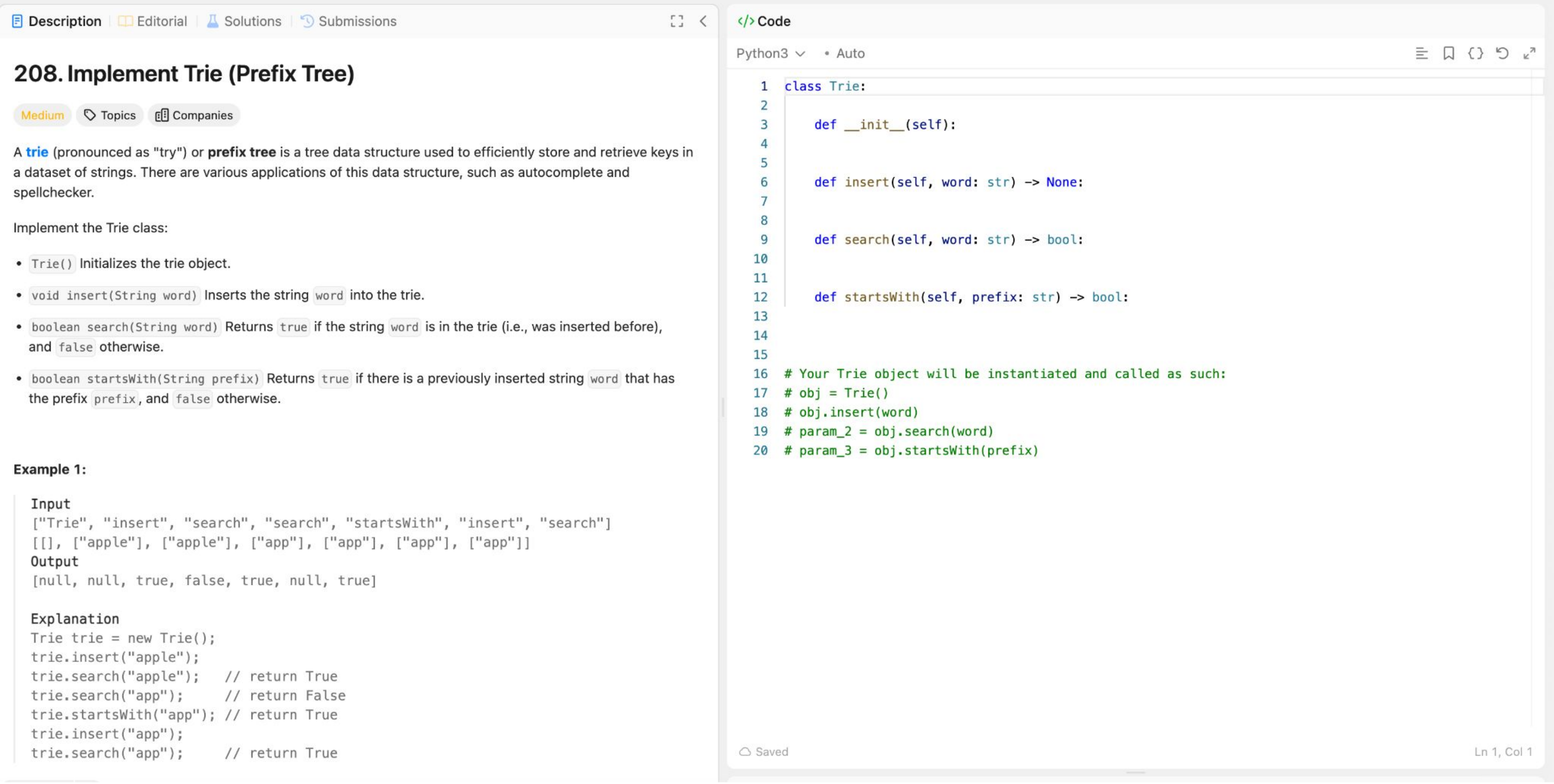
Given the `head` of a singly linked list where elements are sorted in **ascending order**, convert *it to a* ***height-balanced*** *binary search tree*.

Example 1:

```
Input: head = [-10,-3,0,5,9]
Output: [0,-3,9,-10,null,5]
Explanation: One possible answer is [0,-3,9,-10,null,5], which represents the shown
height balanced BST.
```

Example 2:

```python
class Solution:
    def findMiddle(self, head: ListNode) -> ListNode:
        # The pointer used to disconnect the left half from the mid node.
        prevPtr = None
        slowPtr = head
        fastPtr = head
        # Iterate until fastPr doesn't reach the end of the linked list.
        while fastPtr and fastPtr.next:
            prevPtr = slowPtr
            slowPtr = slowPtr.next
            fastPtr = fastPtr.next.next

        # Handling the case when slowPtr was equal to head.
        if prevPtr:
            prevPtr.next = None
        return slowPtr
    def sortedListToBST(self, head: ListNode) -> TreeNode:
        # If the head doesn't exist, then the linked list is empty
        if not head:
            return None
        mid = self.findMiddle(head)
        # The mid becomes the root of the BST.
        node = TreeNode(mid.val)

        # Base case when there is just one element in the linked list
        if head == mid:
            return node

        # Recursively form balanced BSTs using the left and right halves of the original list.
        node.left = self.sortedListToBST(head)
        node.right = self.sortedListToBST(mid.next)
        return node
```



208. Implement Trie (Prefix Tree)

A **trie** (pronounced as "try") or **prefix tree** is a tree data structure used to efficiently store and retrieve keys in a dataset of strings. There are various applications of this data structure, such as autocomplete and spellchecker.

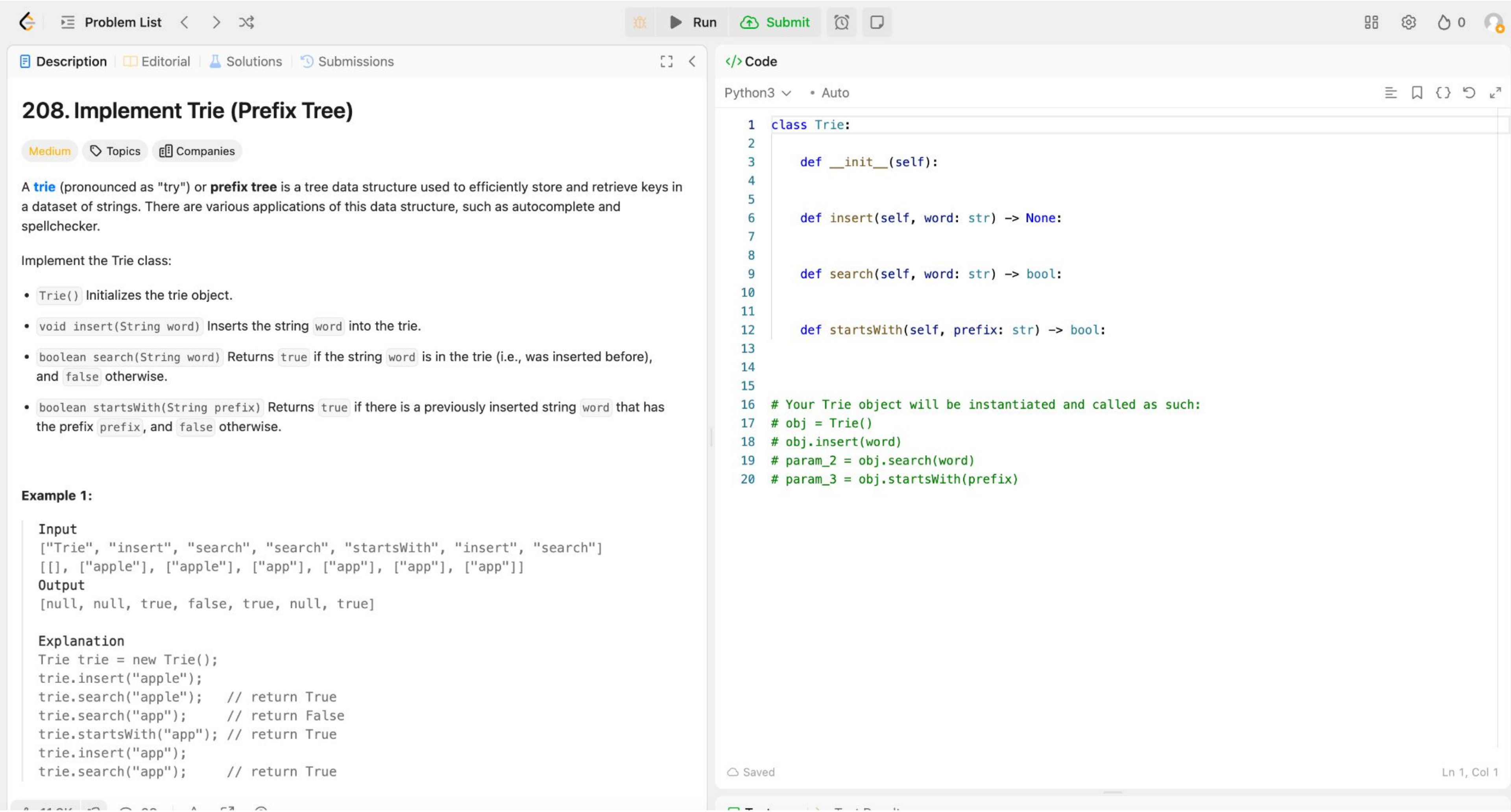
Implement the Trie class:

- `Trie()` Initializes the trie object.
- `void insert(String word)` Inserts the string `word` into the trie.
- `boolean search(String word)` Returns `true` if the string `word` is in the trie (i.e., was inserted before), and `false` otherwise.
- `boolean startsWith(String prefix)` Returns `true` if there is a previously inserted string `word` that has the prefix `prefix`, and `false` otherwise.

Example 1:

```
Input
["Trie", "insert", "search", "search", "startsWith", "insert", "search"]
[[], ["apple"], ["apple"], ["app"], ["app"], ["app"], ["app"]]
Output
[null, null, true, false, true, null, true]

Explanation
Trie trie = new Trie();
trie.insert("apple");
trie.search("apple");   // return True
trie.search("app");     // return False
trie.startsWith("app"); // return True
trie.insert("app");
trie.search("app");     // return True
```

```python
class Trie:

    def __init__(self):


    def insert(self, word: str) -> None:


    def search(self, word: str) -> bool:


    def startsWith(self, prefix: str) -> bool:



# Your Trie object will be instantiated and called as such:
# obj = Trie()
# obj.insert(word)
# param_2 = obj.search(word)
# param_3 = obj.startsWith(prefix)
```



208. Implement Trie (Prefix Tree)

A **trie** (pronounced as "try") or **prefix tree** is a tree data structure used to efficiently store and retrieve keys in a dataset of strings. There are various applications of this data structure, such as autocomplete and spellchecker.

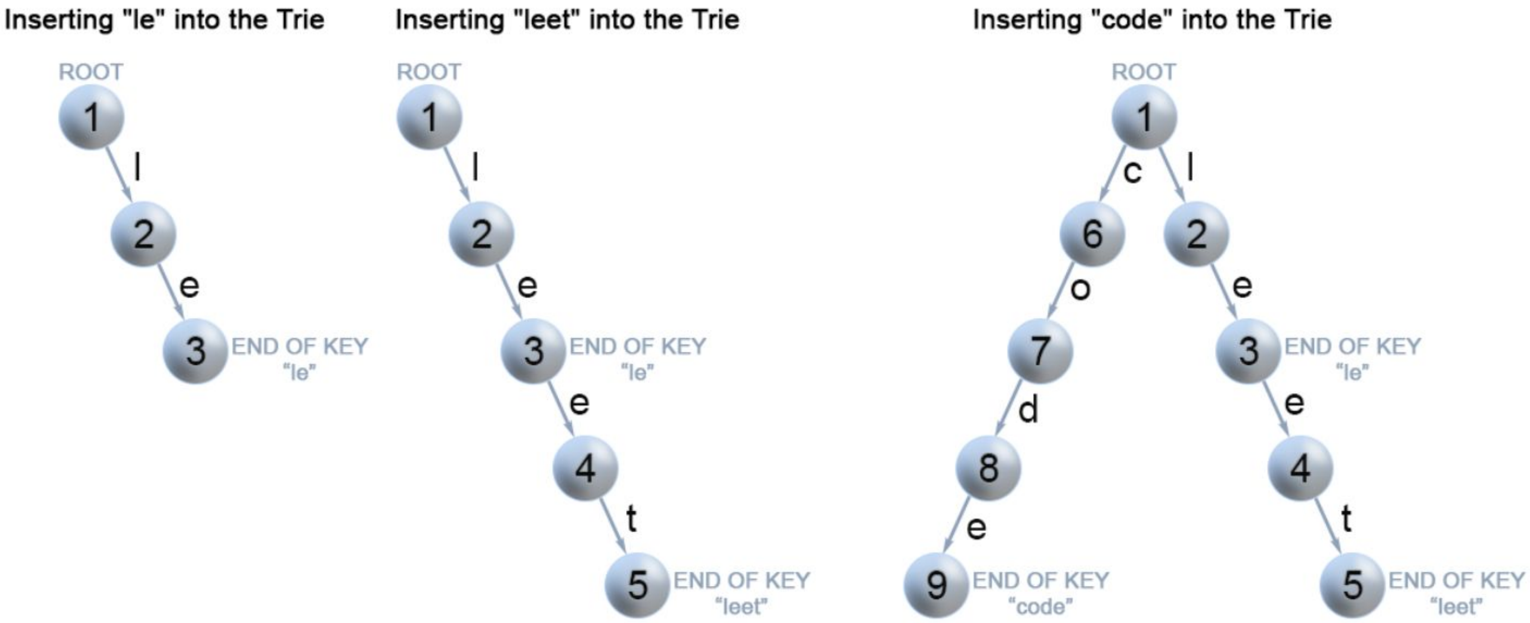
Implement the Trie class:

- `Trie()` Initializes the trie object.
- `void insert(String word)` Inserts the string `word` into the trie.
- `boolean search(String word)` Returns `true` if the string `word` is in the trie (i.e., was inserted before), and `false` otherwise.
- `boolean startsWith(String prefix)` Returns `true` if there is a previously inserted string `word` that has the prefix `prefix`, and `false` otherwise.

Example 1:

```
Input
["Trie", "insert", "search", "search", "startsWith", "insert", "search"]
[[], ["apple"], ["apple"], ["app"], ["app"], ["app"], ["app"]]
Output
[null, null, true, false, true, null, true]

Explanation
Trie trie = new Trie();
trie.insert("apple");
trie.search("apple");   // return True
trie.search("app");     // return False
trie.startsWith("app"); // return True
trie.insert("app");
trie.search("app");     // return True
```

```python
class Trie:

    def __init__(self):


    def insert(self, word: str) -> None:


    def search(self, word: str) -> bool:


    def startsWith(self, prefix: str) -> bool:



# Your Trie object will be instantiated and called as such:
# obj = Trie()
# obj.insert(word)
# param_2 = obj.search(word)
# param_3 = obj.startsWith(prefix)
```



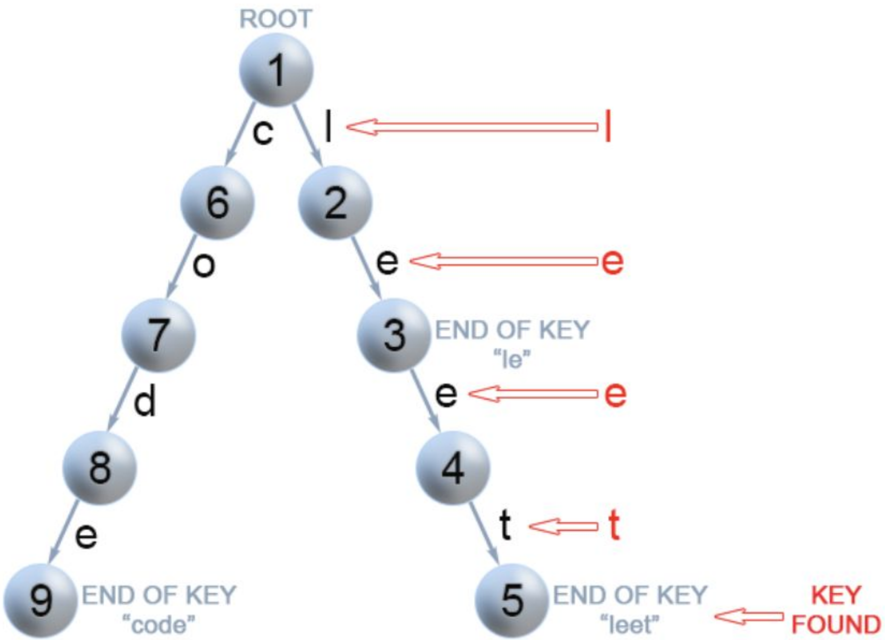
Inserting "le" into the Trie

Inserting "leet" into the Trie

Inserting "code" into the Trie

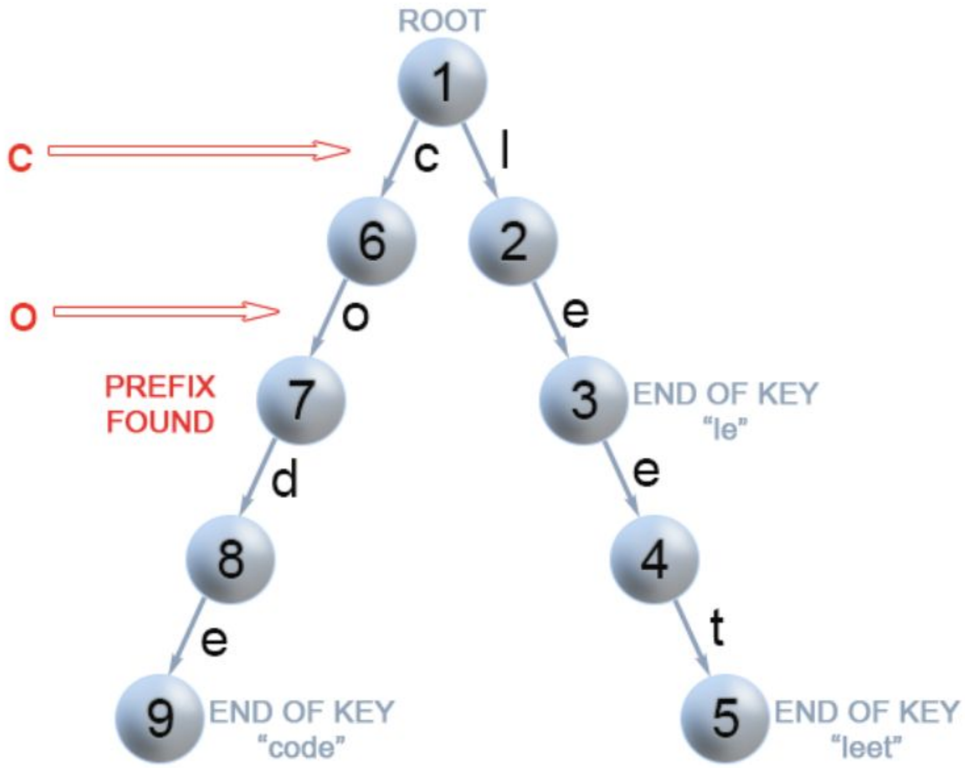
Building a Trie from dataset {le, leet, code}

Searching for key “leet” in the Trie

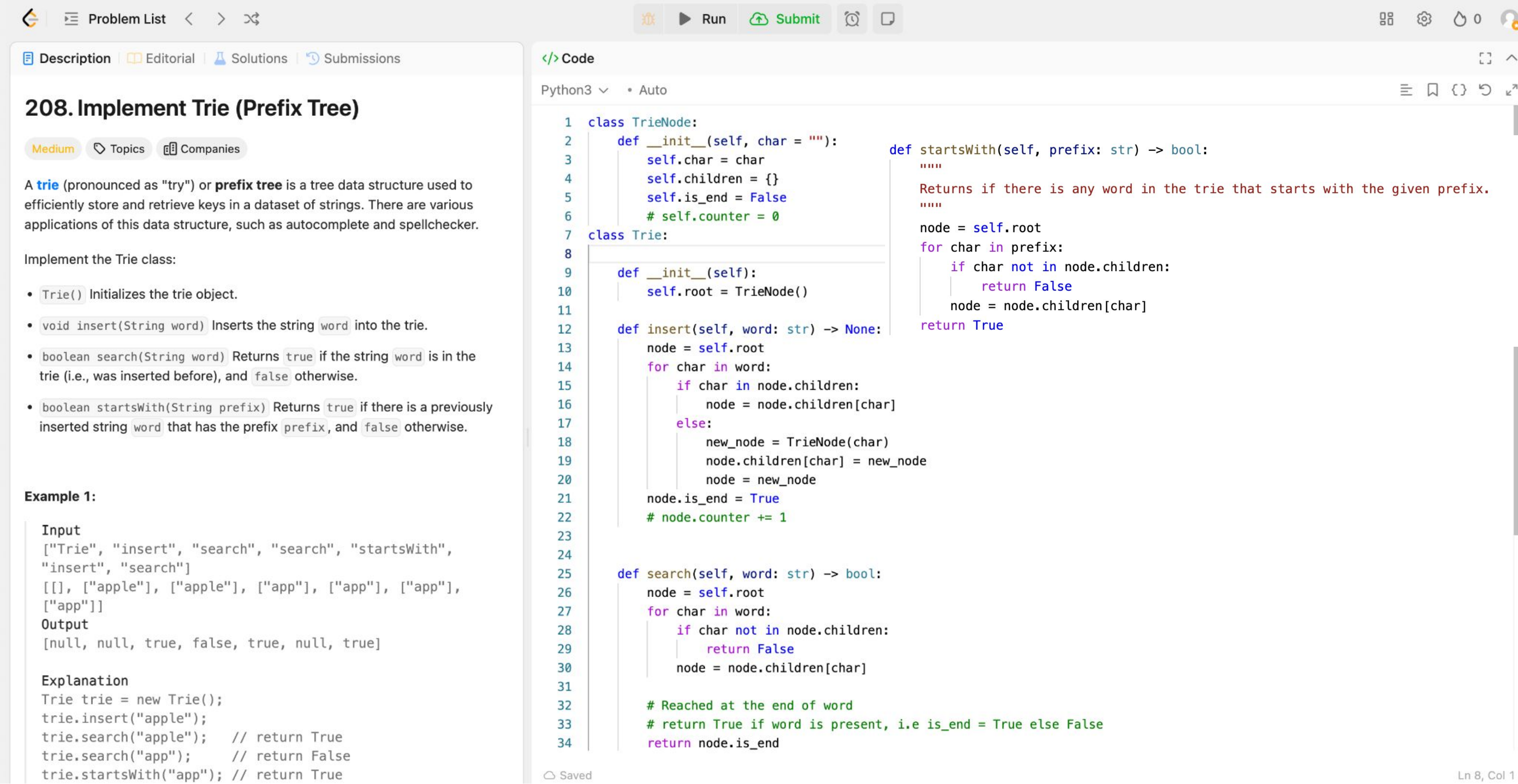


Searching for a key in a Trie from dataset {le, leet, code}

Searching for ‘’co” in the Trie



Searching for a prefix in a Trie from dataset {le, leet, code}



208. Implement Trie (Prefix Tree)

A **trie** (pronounced as "try") or **prefix tree** is a tree data structure used to efficiently store and retrieve keys in a dataset of strings. There are various applications of this data structure, such as autocomplete and spellchecker.

Implement the Trie class:

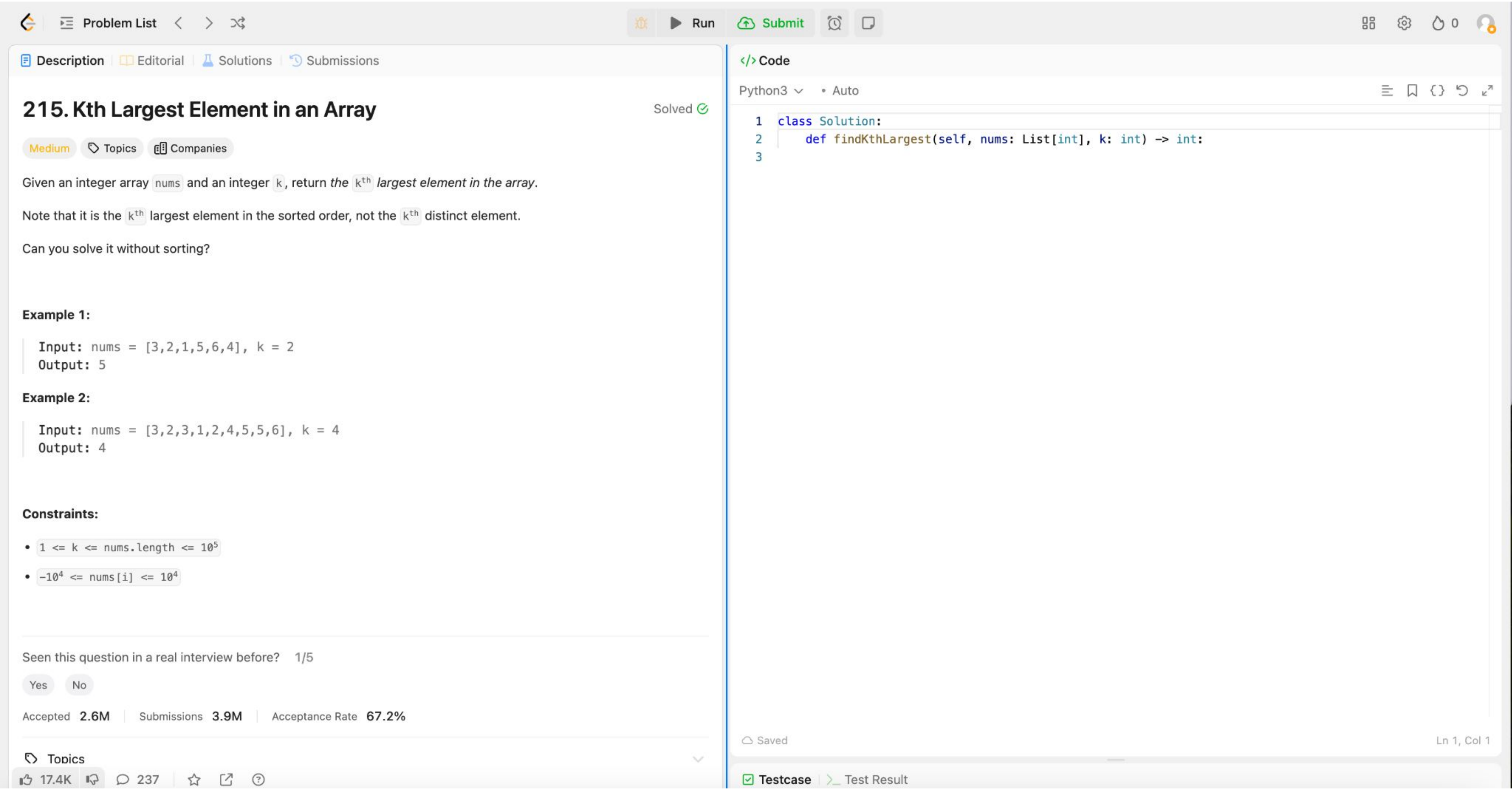
- `Trie()` Initializes the trie object.
- `void insert(String word)` Inserts the string `word` into the trie.
- `boolean search(String word)` Returns `true` if the string `word` is in the trie (i.e., was inserted before), and `false` otherwise.
- `boolean startsWith(String prefix)` Returns `true` if there is a previously inserted string `word` that has the prefix `prefix`, and `false` otherwise.

Example 1:

```
Input
["Trie", "insert", "search", "search", "startsWith",
"insert", "search"]
[[], ["apple"], ["apple"], ["app"], ["app"], ["app"],
["app"]]
Output
[null, null, true, false, true, null, true]

Explanation
Trie trie = new Trie();
trie.insert("apple");
trie.search("apple");   // return True
trie.search("app");     // return False
trie.startsWith("app"); // return True
```

```python
class TrieNode:
    def __init__(self, char = ""):
        self.char = char
        self.children = {}
        self.is_end = False
        # self.counter = 0
class Trie:

    def __init__(self):
        self.root = TrieNode()

    def insert(self, word: str) -> None:
        node = self.root
        for char in word:
            if char in node.children:
                node = node.children[char]
            else:
                new_node = TrieNode(char)
                node.children[char] = new_node
                node = new_node
        node.is_end = True
        # node.counter += 1


    def search(self, word: str) -> bool:
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]

        # Reached at the end of word
        # return True if word is present, i.e is_end = True else False
        return node.is_end
```

```python
def startsWith(self, prefix: str) -> bool:
    """
    Returns if there is any word in the trie that starts with the given prefix.
    """
    node = self.root
    for char in prefix:
        if char not in node.children:
            return False
        node = node.children[char]
    return True
```



215. Kth Largest Element in an Array

Given an integer array `nums` and an integer `k`, return *the* `kth` *largest element in the array*.

Note that it is the `kth` largest element in the sorted order, not the `kth` distinct element.

Can you solve it without sorting?

Example 1:

```
Input: nums = [3,2,1,5,6,4], k = 2
Output: 5
```

Example 2:

```
Input: nums = [3,2,3,1,2,4,5,5,6], k = 4
Output: 4
```

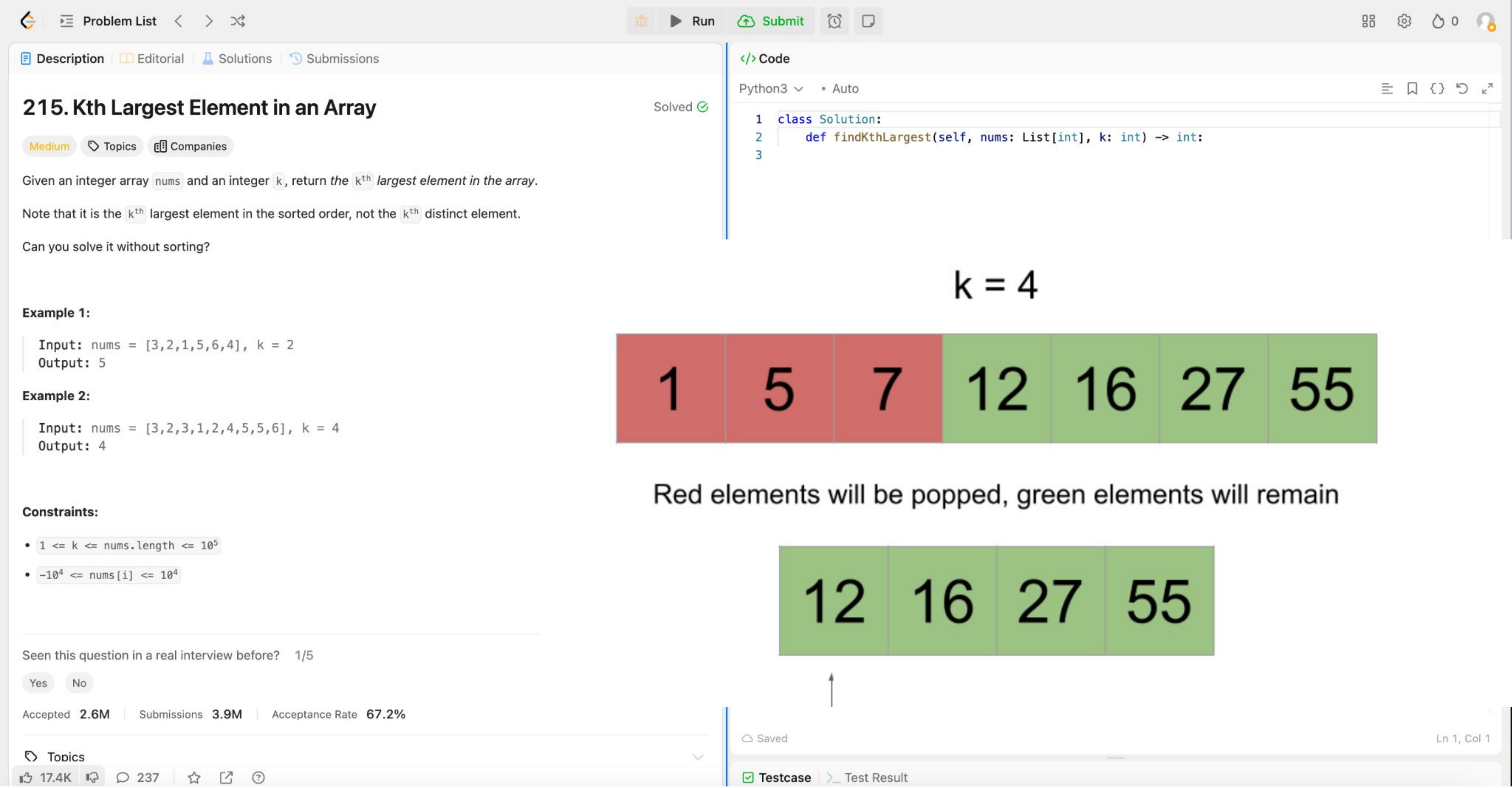
Constraints:

- $1 <= k <= nums.length <= 10^5$
- $-10^4 <= nums[i] <= 10^4$

Seen this question in a real interview before? 1/5

Accepted 2.6M | Submissions 3.9M | Acceptance Rate 67.2%

```python
class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:

```



215. Kth Largest Element in an Array

Solved ✓

Medium Topics Companies

Given an integer array `nums` and an integer `k`, return *the* k^{th} *largest element in the array*.

Note that it is the k^{th} largest element in the sorted order, not the k^{th} distinct element.

Can you solve it without sorting?

Example 1:

```
Input: nums = [3,2,1,5,6,4], k = 2
Output: 5
```

Example 2:

```
Input: nums = [3,2,3,1,2,4,5,5,6], k = 4
Output: 4
```

Constraints:

- `1 <= k <= nums.length <= 10`^{5}
- `-10`^{4} `<= nums[i] <= 10`^{4}

Seen this question in a real interview before? 1/5

Yes No

Accepted 2.6M | Submissions 3.9M | Acceptance Rate 67.2%

Topics

17.4K 237

```python
class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
```

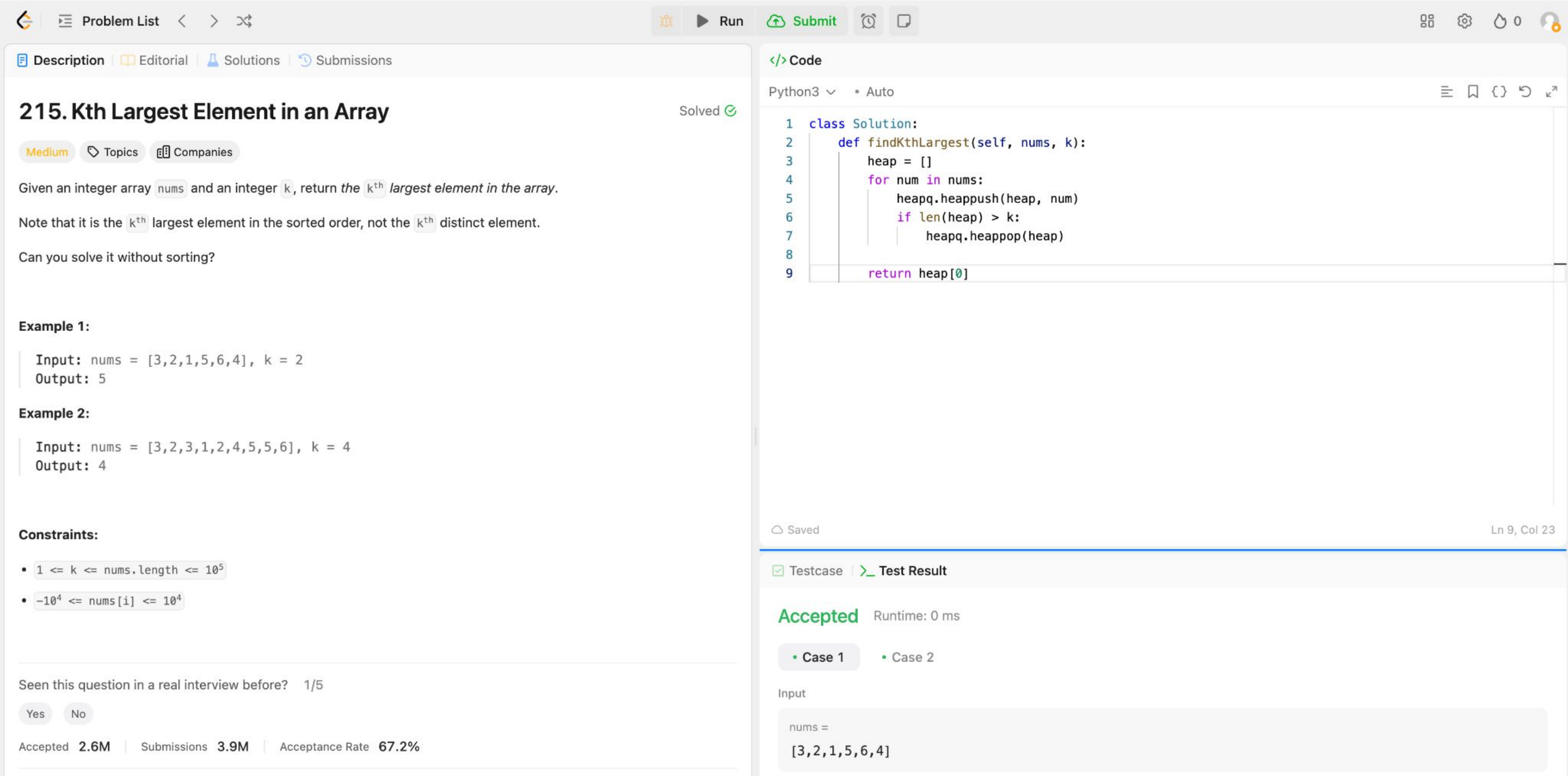
k = 4

🟥 1	🟥 5	🟥 7	🟩 12	🟩 16	🟩 27	🟩 55

Red elements will be popped, green elements will remain

🟩 12	🟩 16	🟩 27	🟩 55

↑



215. Kth Largest Element in an Array

Given an integer array `nums` and an integer `k`, return *the* k^{th} *largest element in the array*.

Note that it is the k^{th} largest element in the sorted order, not the k^{th} distinct element.

Can you solve it without sorting?

Example 1:

```
Input: nums = [3,2,1,5,6,4], k = 2
Output: 5
```

Example 2:

```
Input: nums = [3,2,3,1,2,4,5,5,6], k = 4
Output: 4
```

Constraints:

- $1 <= k <= nums.length <= 10^5$
- $-10^4 <= nums[i] <= 10^4$

Seen this question in a real interview before? 1/5

Yes No

Accepted 2.6M | Submissions 3.9M | Acceptance Rate 67.2%

```python
class Solution:
    def findKthLargest(self, nums, k):
        heap = []
        for num in nums:
            heapq.heappush(heap, num)
            if len(heap) > k:
                heapq.heappop(heap)

        return heap[0]
```

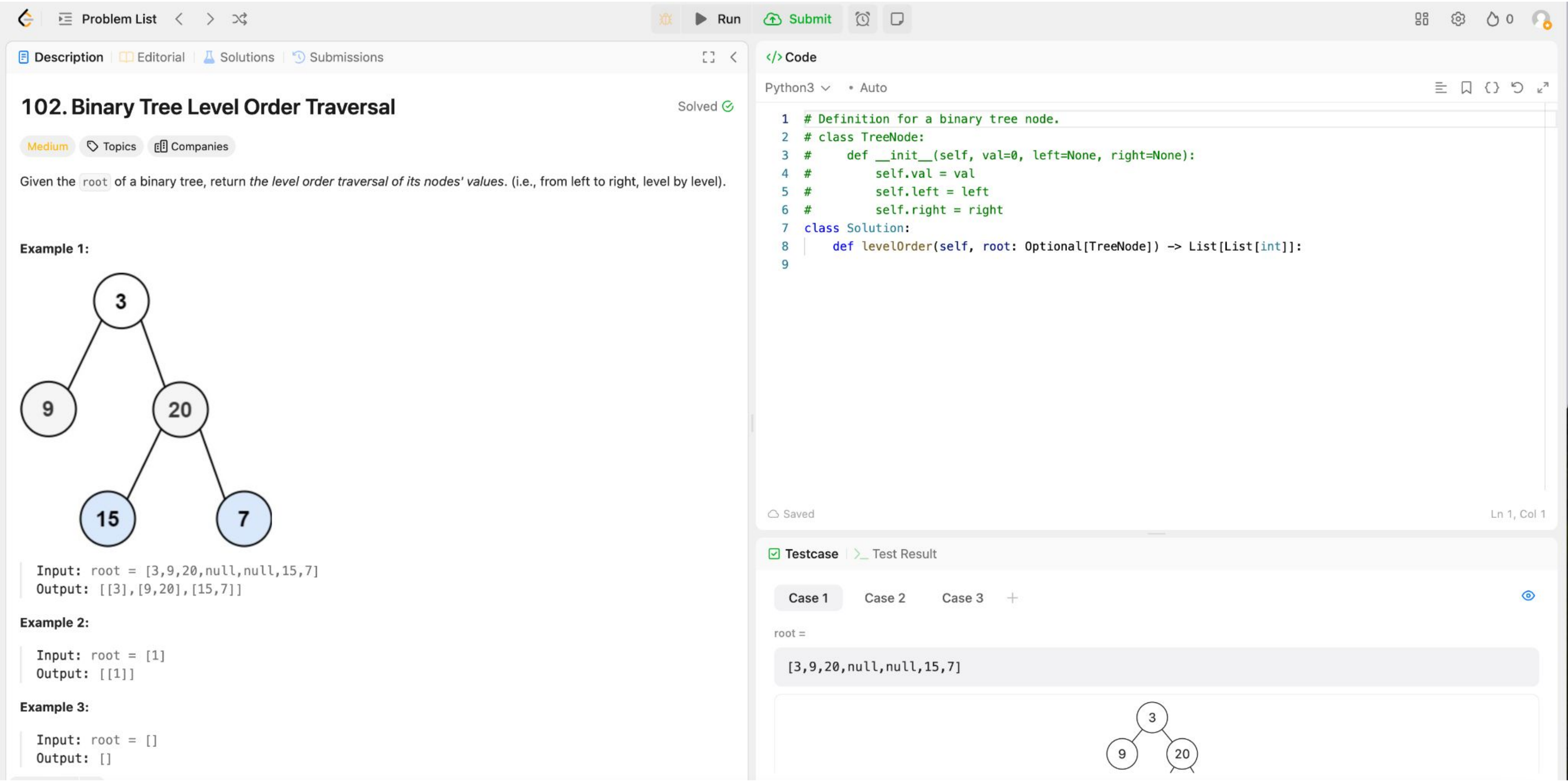
Accepted Runtime: 0 ms

• Case 1 • Case 2

Input

nums =

[3,2,1,5,6,4]



102. Binary Tree Level Order Traversal

Solved ✓

Medium Topics Companies

Given the `root` of a binary tree, return *the level order traversal of its nodes' values*. (i.e., from left to right, level by level).

Example 1:

```
Input: root = [3,9,20,null,null,15,7]
Output: [[3],[9,20],[15,7]]
```

Example 2:

```
Input: root = [1]
Output: [[1]]
```

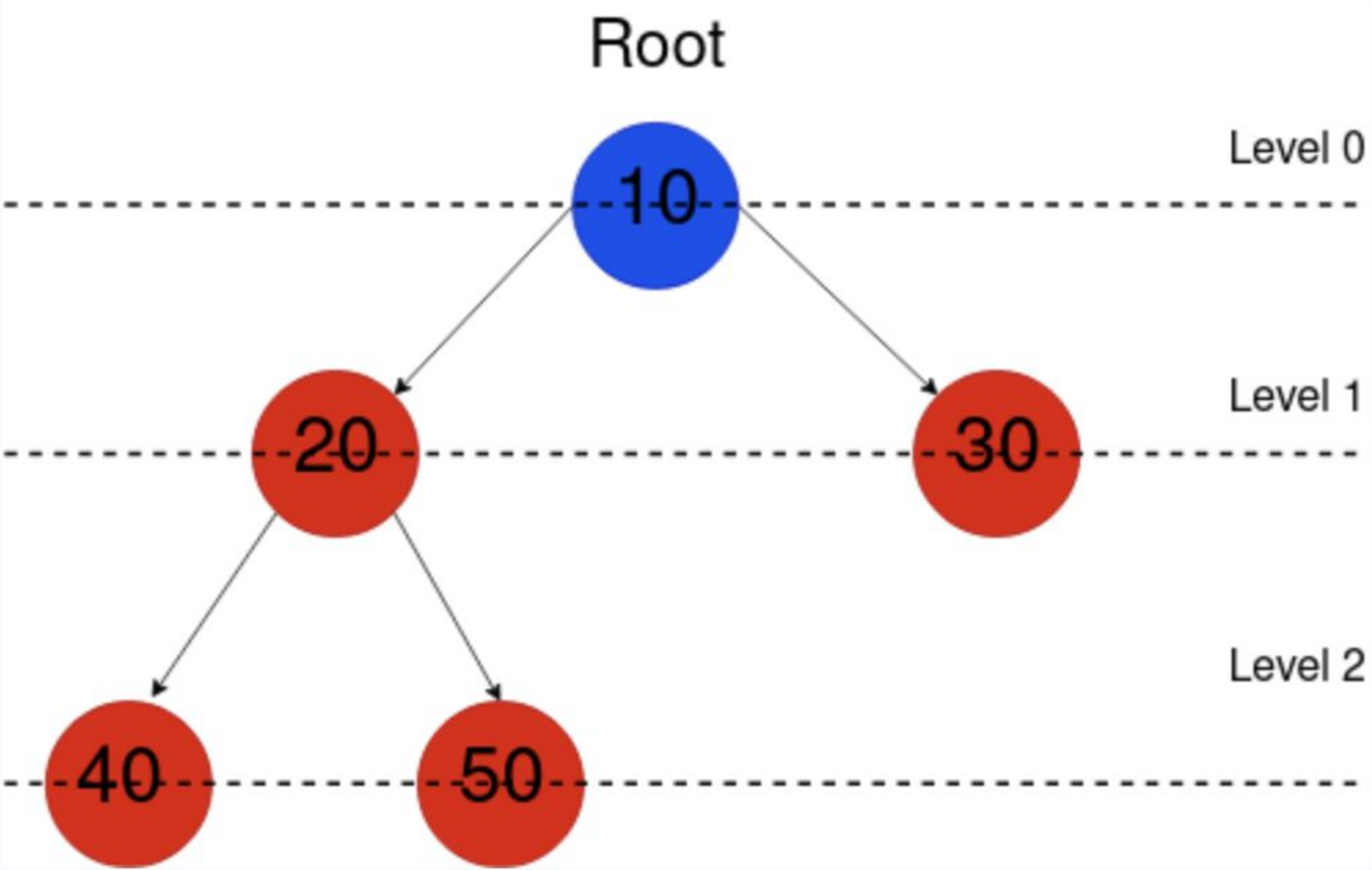
Example 3:

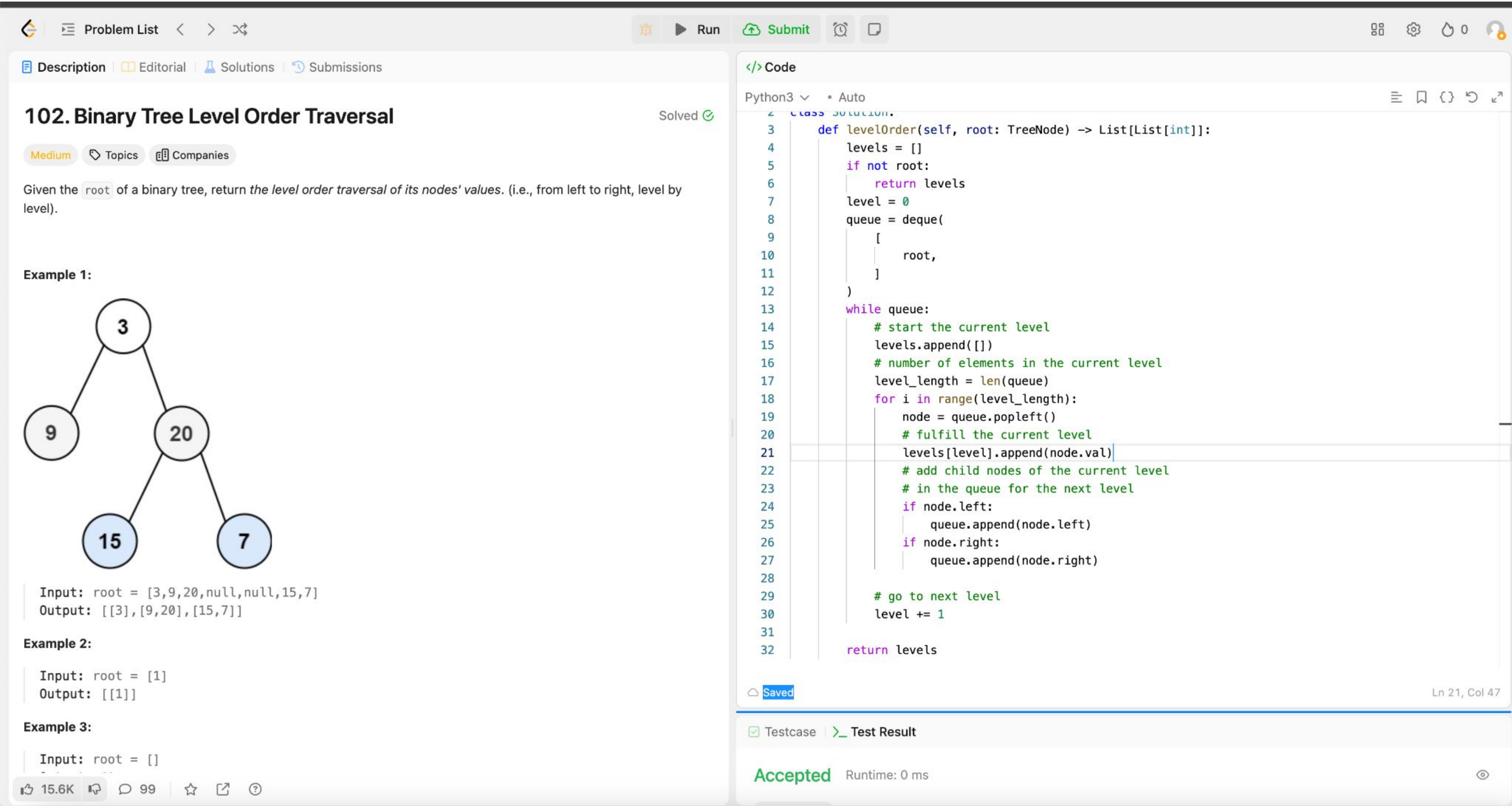
```
Input: root = []
Output: []
```

```python
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def levelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:

```

root =

```
[3,9,20,null,null,15,7]
```


Root
Level 0
10
Level 1
20
30
Level 2
40
50



102. Binary Tree Level Order Traversal

Given the `root` of a binary tree, return *the level order traversal of its nodes' values*. (i.e., from left to right, level by level).

Example 1:

```
Input: root = [3,9,20,null,null,15,7]
Output: [[3],[9,20],[15,7]]
```

Example 2:

```
Input: root = [1]
Output: [[1]]
```

Example 3:

```
Input: root = []
```

```python
class Solution:
    def levelOrder(self, root: TreeNode) -> List[List[int]]:
        levels = []
        if not root:
            return levels
        level = 0
        queue = deque(
            [
                root,
            ]
        )
        while queue:
            # start the current level
            levels.append([])
            # number of elements in the current level
            level_length = len(queue)
            for i in range(level_length):
                node = queue.popleft()
                # fulfill the current level
                levels[level].append(node.val)
                # add child nodes of the current level
                # in the queue for the next level
                if node.left:
                    queue.append(node.left)
                if node.right:
                    queue.append(node.right)

            # go to next level
            level += 1

        return levels
```

Accepted Runtime: 0 ms