Previous Next

270. Closest BST Value

July 11, 2019 | 20.9K views

Average Rating: 4.83 (12 votes)

Given a non-empty binary search tree and a target value, find the value in the BST that is closest to the target.

Note:

- Given target value is a floating point.
- You are guaranteed to have only one unique value in the BST that is closest to the target.

Example:

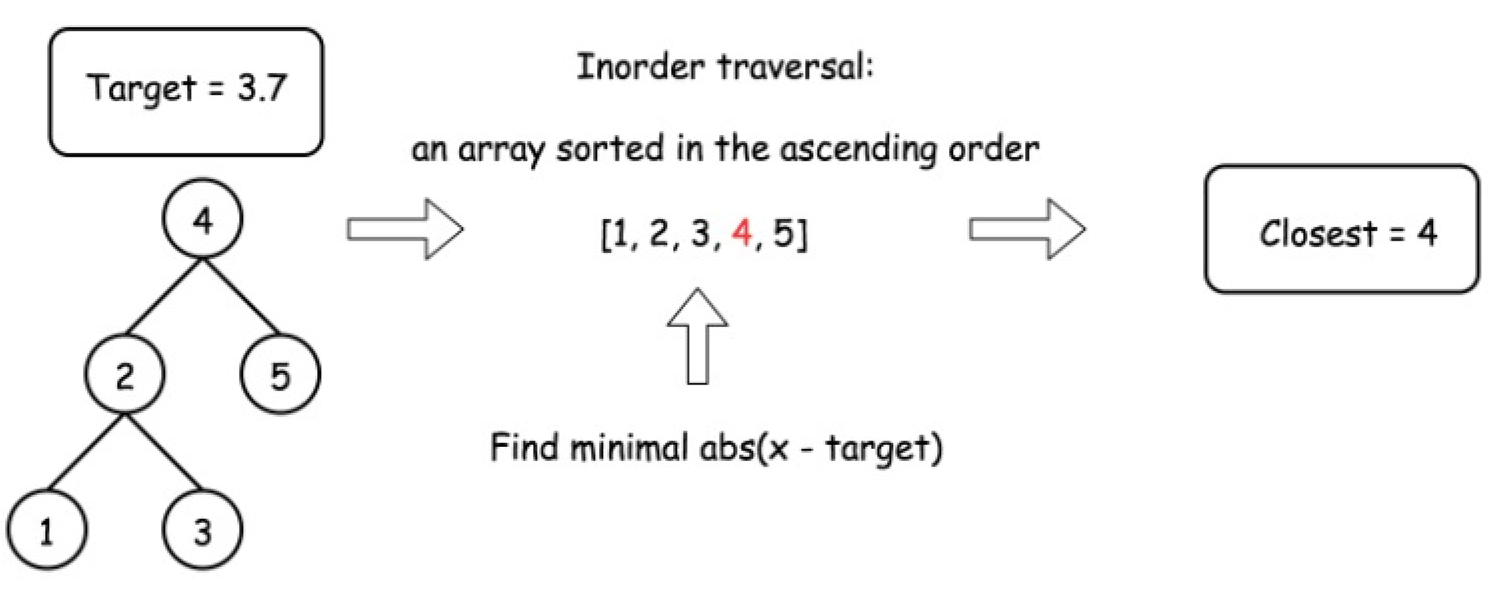
```
Input: root = [4,2,5,1,3], target = 3.714286

    4
   / \
  2   5
 / \
1   3

Output: 4
```

Solution

Approach 1: Recursive Inorder + Linear search, O(N) time

Intuition

The simplest approach (3 lines in Python) is to build inorder traversal and then find the closest element in a sorted array with built-in function `min`.



This approach is simple stupid, and serves to identify the subproblems.

Algorithm

- Build an inorder traversal array.
- Find the closest to target element in that array.

Implementation

Java Python Copy

```java
class Solution {
  public void inorder(TreeNode root, List<Integer> nums) {
    if (root == null) return;
    inorder(root.left, nums);
    nums.add(root.val);
    inorder(root.right, nums);
  }

  public int closestValue(TreeNode root, double target) {
    List<Integer> nums = new ArrayList();
    inorder(root, nums);
    return Collections.min(nums, new Comparator<Integer>() {
      @Override
      public int compare(Integer o1, Integer o2) {
        return Math.abs(o1 - target) < Math.abs(o2 - target) ? -1 : 1;
      }
    });
  }
}
```

Complexity Analysis

- Time complexity : $\mathcal{O}(N)$ because to build inorder traversal and then to perform linear search takes linear time.
- Space complexity : $\mathcal{O}(N)$ to keep inorder traversal.

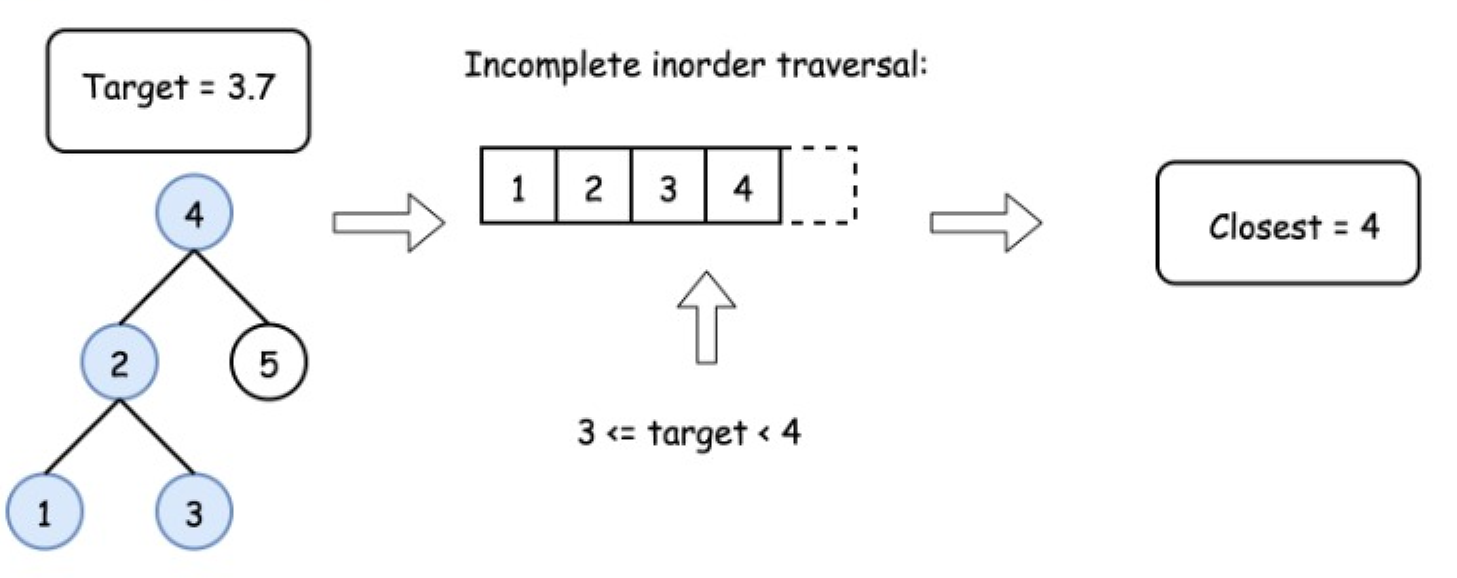
Approach 2: Iterative Inorder, O(k) time

Intuition

Let's optimise Approach 1 in the case when index k of the closest element is much smaller than the tree heigh H.

First, one could merge both steps by traversing the tree and searching the closest value at the same time.

Second, one could stop just after identifying the closest value, there is no need to traverse the whole tree. The closest value is found if the target value is in-between of two inorder array elements `nums[i] <= target < nums[i + 1]`. Then the closest value is one of these elements.



Algorithm

- Initiate stack as an empty array and predecessor value as a very small number.
- While root is not null:
  - To build an inorder traversal iteratively, go left as far as you can and add all nodes on the way into stack.
  - Pop the last element from stack `root = stack.pop()`.
  - If target is in-between of `pred` and `root.val`, return the closest between these two elements.
  - Set predecessor value to be equal to `root.val` and go one step right: `root = root.right`.
- We're here because during the loop one couldn't identify the closest value. That means that the closest value is the last value in the inorder traversal, i.e. current predecessor value. Return it.

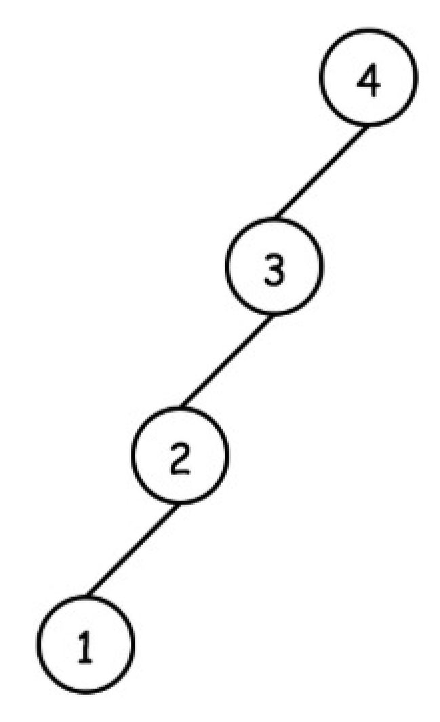
Implementation

Java Python Copy

```java
class Solution {
  public int closestValue(TreeNode root, double target) {
    LinkedList<TreeNode> stack = new LinkedList();
    long pred = Long.MIN_VALUE;

    while (!stack.isEmpty() || root != null) {
      while (root != null) {
        stack.add(root);
        root = root.left;
      }
      root = stack.removeLast();

      if (pred <= target && target < root.val)
        return Math.abs(pred - target) < Math.abs(root.val - target) ? (int)pred : root.val;

      pred = root.val;
      root = root.right;
    }
    return (int)pred;
  }
}
```

Complexity Analysis

- Time complexity : $\mathcal{O}(k)$ in the average case and $\mathcal{O}(H + k)$ in the worst case, where k is an index of closest element. It's known that average case is a balanced tree, in that case stack always contains a few elements, and hence one does $2k$ operations to go to kth element in inorder traversal (k times to push into stack and then k times to pop out of stack). That results in $\mathcal{O}(k)$ time complexity. The worst case is a completely unbalanced tree, then you first push H elements into stack and then pop out k elements, that results in $\mathcal{O}(H + k)$ time complexity.



- Space complexity : up to $\mathcal{O}(H)$ to keep the stack in the case of non-balanced tree.

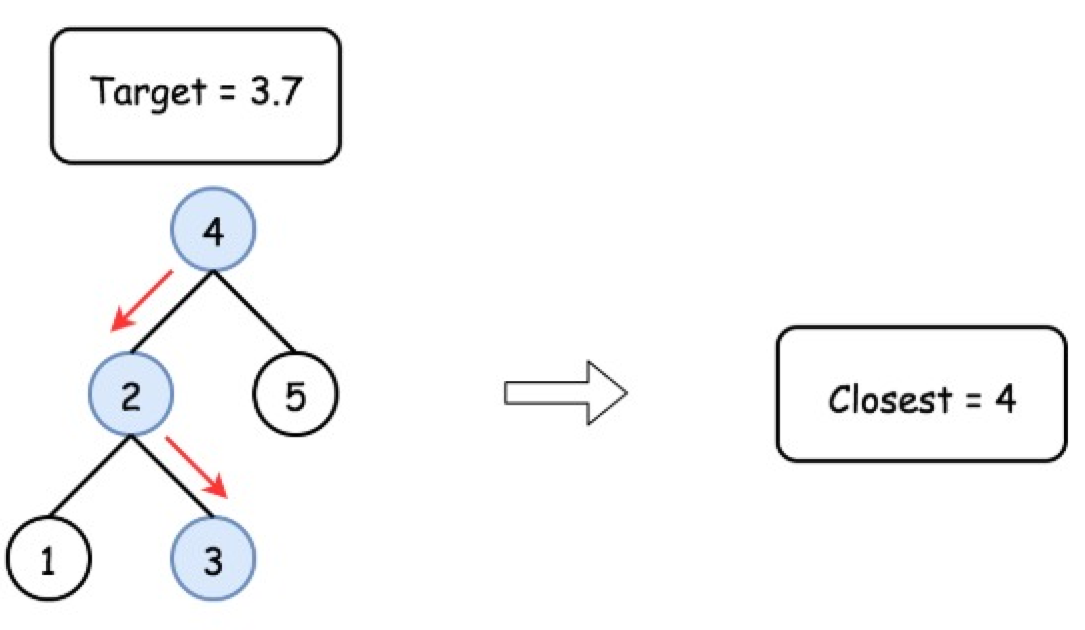
Approach 3: Binary Search, O(H) time

Intuition

Approach 2 works fine when index k of closest element is much smaller than the tree height H.

Let's now consider another limit and optimise Approach 1 in the case of relatively large k, comparable with N.

Then it makes sense to use a binary search: go left if target is smaller than current root value, and go right otherwise. Choose the closest to target value at each step.



Kudos for this solution go to @stefanpochmann.

Implementation

Java Python Copy

```java
class Solution {
  public int closestValue(TreeNode root, double target) {
    int val, closest = root.val;
    while (root != null) {
      val = root.val;
      closest = Math.abs(val - target) < Math.abs(closest - target) ? val : closest;
      root =  target < root.val ? root.left : root.right;
    }
    return closest;
  }
}
```

Complexity Analysis

- Time complexity : $\mathcal{O}(H)$ since here one goes from root down to a leaf.
- Space complexity : $\mathcal{O}(1)$.

Analysis written by @liaison and @andvary

Rate this article:

Previous Next