How to Traverse the Tree. DFS: Preorder, Inorder, Postorder; BFS.

There are two general strategies to traverse a tree:

- *Depth First Search* (`DFS`)

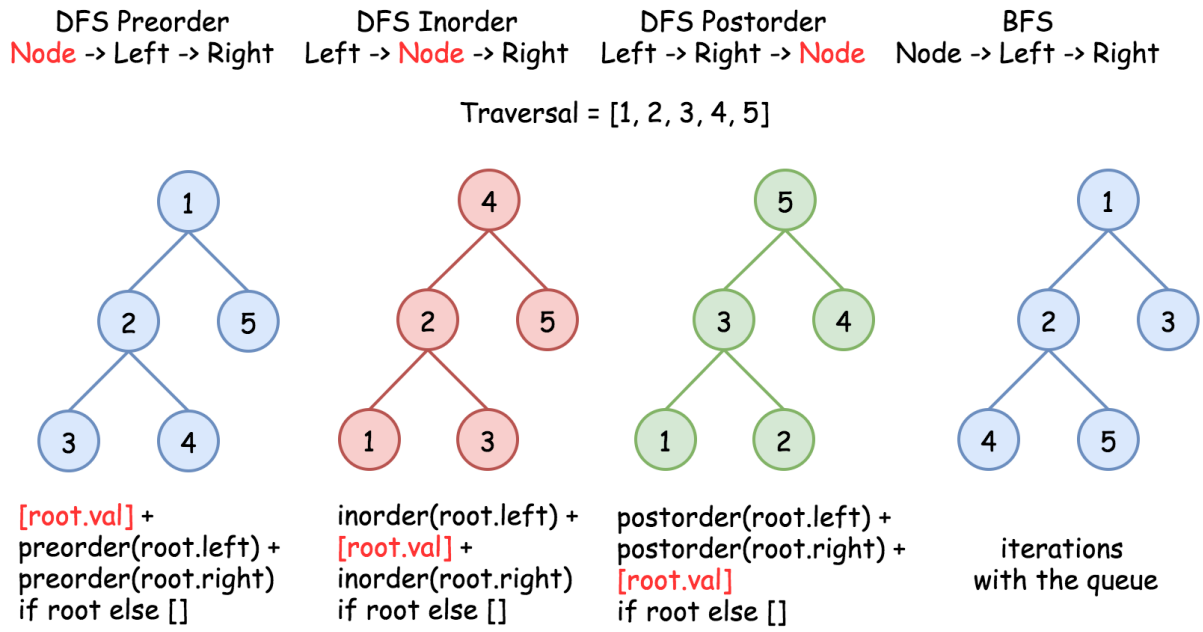
  In this strategy, we adopt the `depth` as the priority, so that one would start from a root and reach all the way down to certain leaf, and then back to root to reach another branch.

  The DFS strategy can further be distinguished as `preorder`, `inorder`, and `postorder` depending on the relative order among the root node, left node and right node.

- *Breadth First Search* (`BFS`)

  We scan through the tree level by level, following the order of height, from top to bottom. The nodes on higher level would be visited before the ones with lower levels.

On the following figure the nodes are enumerated in the order you visit them, please follow `1-2-3-4-5` to compare different strategies.



Construct BST from Inorder Traversal: Why the Solution is *Not* Unique

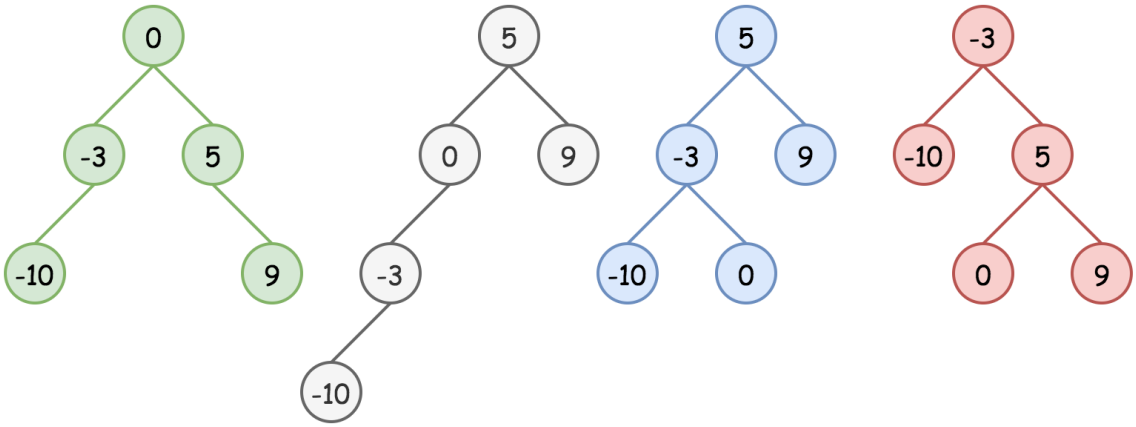
It's known that inorder traversal of BST is an array sorted in the ascending order.

Having sorted array as an input, we could rewrite the problem as *Construct Binary Search Tree from Inorder Traversal*.

Does this problem have a unique solution, i.e. could inorder traversal be used as a unique identifier to encore/decode BST? The answer is *no*.
Here is the funny thing about BST. Inorder traversal is *not* a unique identifier of BST. At the same time both preorder and postorder traversals *are* unique identifiers of BST. From these traversals one could restore the inorder one: `inorder = sorted(postorder) = sorted(preorder)`, and inorder + postorder or inorder + preorder are both unique identifiers of whatever binary tree.

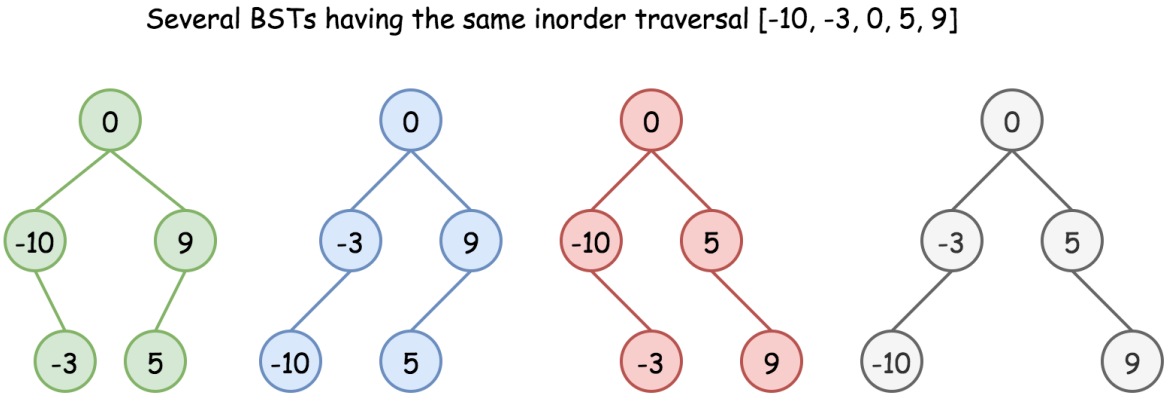
So, the problem "sorted array –> BST" has multiple solutions.

Several BSTs having the same inorder traversal [-10, -3, 0, 5, 9]

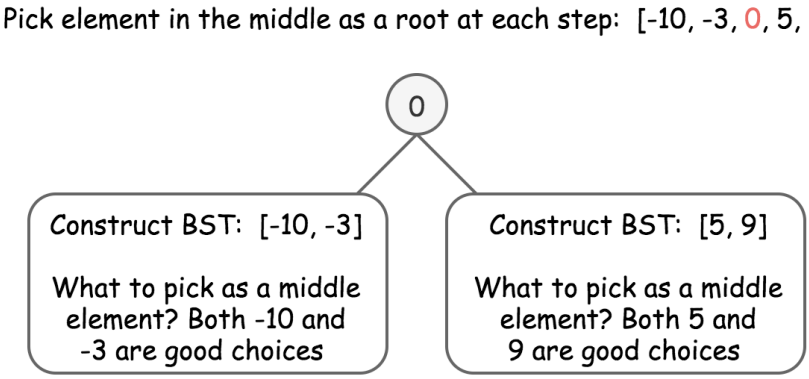


Here we have an additional condition: *the tree should be height-balanced*, i.e. the depths of the two subtrees of every node never differ by more than 1.

Does it make the solution to be unique? Still no.



Basically, the height–balanced restriction means that at each step one has to pick up the number in the middle as a root. That works fine with arrays containing odd number of elements but there is no predefined choice for arrays with even number of elements.

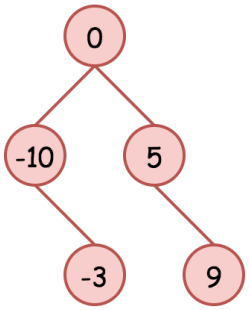


One could choose left middle element, or right middle one, and both choices will lead to *different* height–balanced BSTs. Let's see that in practice: in Approach 1 we will always pick up left middle element, in Approach 2 – right middle one. That will generate *different* BSTs but both solutions will be accepted.

Approach 1: Preorder Traversal: Always Choose Left Middle Node as a Root

Algorithm

Always choose left middle element as a root [-10, -3, 0, 5, 9]



- Implement helper function `helper(left, right)`, which constructs BST from nums elements between indexes `left` and `right`:
    - If left > right, then there is no elements available for that subtree. Return None.
    - Pick left middle element: `p = (left + right) // 2`.
    - Initiate the root: `root = TreeNode(nums[p])`.
    - Compute recursively left and right subtrees: `root.left = helper(left, p - 1)`, `root.right = helper(p + 1, right)`.
- Return `helper(0, len(nums) - 1)`.

Implementation

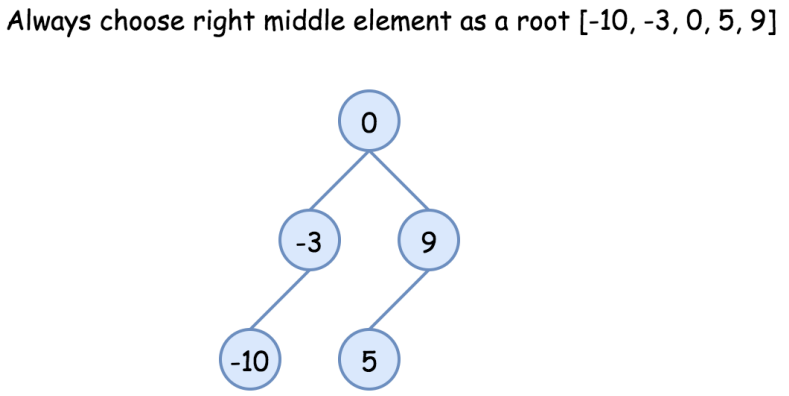
Complexity Analysis

- Time complexity: $\mathrm{O(N)}O(N)$ since we visit each node exactly once.
- Space complexity: $\mathrm{O(\log N)}O(\log N)$.

  The recursion stack requires $\mathrm{O(\log N)}O(\log N)$ space because the tree is height–balanced. Note that the $\mathrm{O(N)}O(N)$ space used to store the output does not count as auxiliary space, so it is not included in the space complexity.

Approach 2: Preorder Traversal: Always Choose Right Middle Node as a Root

Algorithm



- Implement helper function `helper(left, right)`, which constructs BST from nums elements between indexes `left` and `right`:
    - If left > right, then there is no elements available for that subtree. Return None.
    - Pick right middle element:
        - `p = (left + right) // 2`.
        - If `left + right` is odd, add 1 to p-index.
    - Initiate the root: `root = TreeNode(nums[p])`.
    - Compute recursively left and right subtrees: `root.left = helper(left, p - 1)`, `root.right = helper(p + 1, right)`.
- Return `helper(0, len(nums) - 1)`.

Implementation

Complexity Analysis

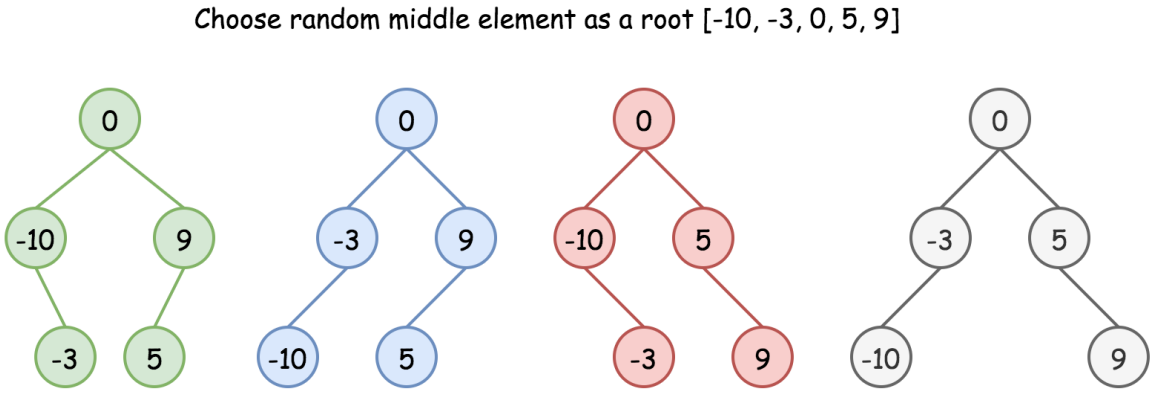
- Time complexity: $\mathrm{O(N)}O(N)$ since we visit each node exactly once.

- Space complexity: $O(\log N)$.

  The recursion stack requires $O(\log N)$ space because the tree is height-balanced. Note that the $O(N)$ space used to store the output does not count as auxiliary space, so it is not included in the space complexity.

Approach 3: Preorder Traversal: Choose Random Middle Node as a Root

This one is for fun. Instead of predefined choice we will pick randomly left or right middle node at each step. Each run will result in different solution and they all will be accepted.



Algorithm

- Implement helper function `helper(left, right)`, which constructs BST from nums elements between indexes `left` and `right`:
  - If left > right, then there is no elements available for that subtree. Return None.
  - Pick random middle element:
    - `p = (left + right) // 2`.
    - If `left + right` is odd, add randomly 0 or 1 to p-index.
  - Initiate the root: `root = TreeNode(nums[p])`.

    - Compute recursively left and right subtrees: `root.left = helper(left, p - 1)`, `root.right = helper(p + 1, right)`.

- Return `helper(0, len(nums) - 1)`.

Implementation

Complexity Analysis

- Time complexity: $\mathrm{O(N)}$ $O(N)$ since we visit each node exactly once.
- Space complexity: $\mathrm{O(\log N)}$ $O(\log N)$.

  The recursion stack requires $\mathrm{O(\log N)}$ $O(\log N)$ space because the tree is height–balanced. Note that the $\mathrm{O(N)}$ $O(N)$ space used to store the output does not count as auxiliary space, so it is not included in the space complexity.