Algorithms and Data Structures for Beginners

18 / 35

17 - Binary Search Tree

Mark Lesson Complete View Code Prev Next

Suggested Problems

Status	Star	Problem	Difficulty	Video Solution	Code
☐	☆	Search In A Binary Search Tree	Easy		

Binary Search Trees

Difference between Binary Trees and Binary Search Trees

Binary Search Trees (BST) are a variation of binary trees with a sorted property to them. The property tells us that every left child must be smaller than its parent and every right child must be greater than its parent. BSTs *do not* allow duplicates.

> *It should be noted that the BST property applies to subtrees as well. So while a node which has a value smaller than root will be on the left subtree, it is important to determine where exactly in the left subtree that value will be inserted.*

Motivation

The question here is why bother with BSTs if we have sorted arrays? With binary search, we can search values in $O(log\ n)$ time and if BST is offering the same functionality, can't we just use an array? All of this is correct. However, BST shines when we are trying to insert or delete a value. Both of these operations run in $O(log\ n)$ time for a BST, but $O(n)$ time with an array.

So, while BSTs don't offer benefit over sorted arrays for the search functionality, they are better for insertion and deletion. In this chapter, we will focus specfically on the search operation.

BST Search

Trees are best traversed using recursion. You could traverse iteratively, however, that requires maintaining a stack, which is a lot more complicated. For recursion, as discussed before, we need a base case and the function calling itself.

Let's take the tree `[2,1,3,null,null,null,4]` for example and search for `target = 3` .

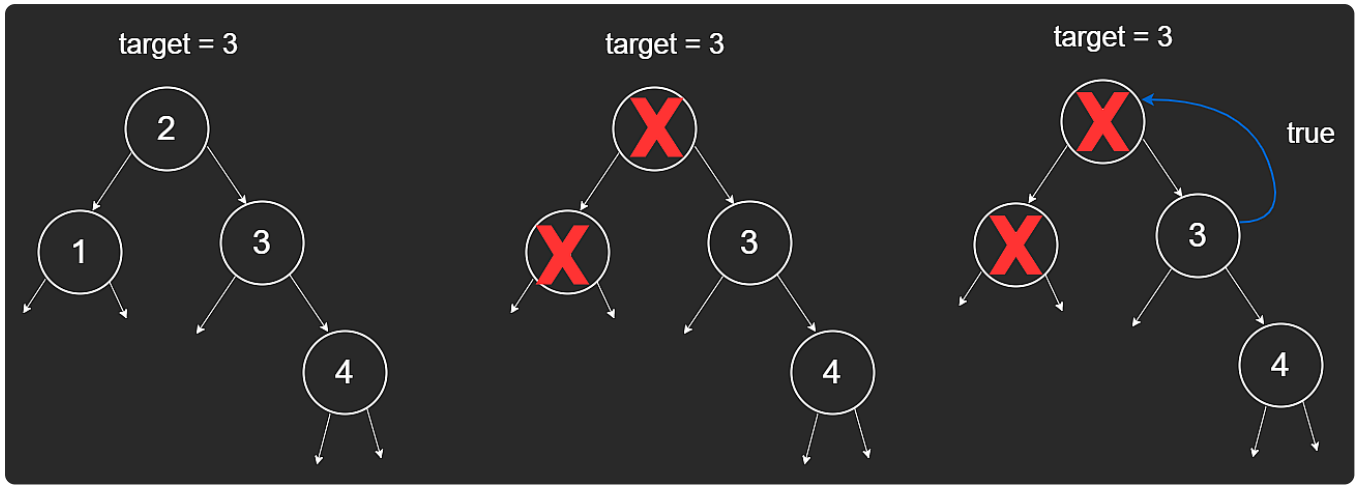
In binary search, if the current element was greater than the target, we went left and if the current element was smaller than the target, we went right. A similar approach can be taken here. We know all nodes to the left are smaller than our current node

and all nodes to the right are greater than our current node. Knowing this, we can go right if our current node is smaller than the target and go left if the current node is greater than the target.

If the target exists in the tree, we will return true. Otherwise, we return false.

In the case of the example, we first start by comparing the root value against the `target`. `2` is too small, so our target must be on the right, meaning we can eliminate the left-subtree. When we go right, the first node is `3`, which equals `target`, so we return true from the recursive call, meaning our target does exist in the tree.

This is demonstrated by the pseudocode and the visual below.




```
fn search(root, target):
    // Our base case
```

```
if root is null:
    return false
// We can ignore the left subtree because our target is larger thar
if target > root.val:
    return search(root.right, target)
// We can ignore the right subtree because our target is smaller th
else if target < root.val:
    return search(root.left, target)
else:
    return true
```

Time Complexity

If we have a balanced binary tree, our search algorithm will run in $O(log\ n)$ time. Balanced binary tree means that the height of the left-subtree is equal to the height of the right-subtree, or there is a difference of 1. In a balanced tree, we can eliminate half the nodes each time, which results in $O(log\ n)$, for reasons we discussed in the merge sort chapter.

If we had a skewed binary tree, this would result in a time complexity of $O(n)$. This is also the worst case scenario.

Closing Notes

The main benefit of binary search trees compared to sorted arrays is that we are able to insert, delete and search in $O(log\ n)$ time.

Github Privacy Terms