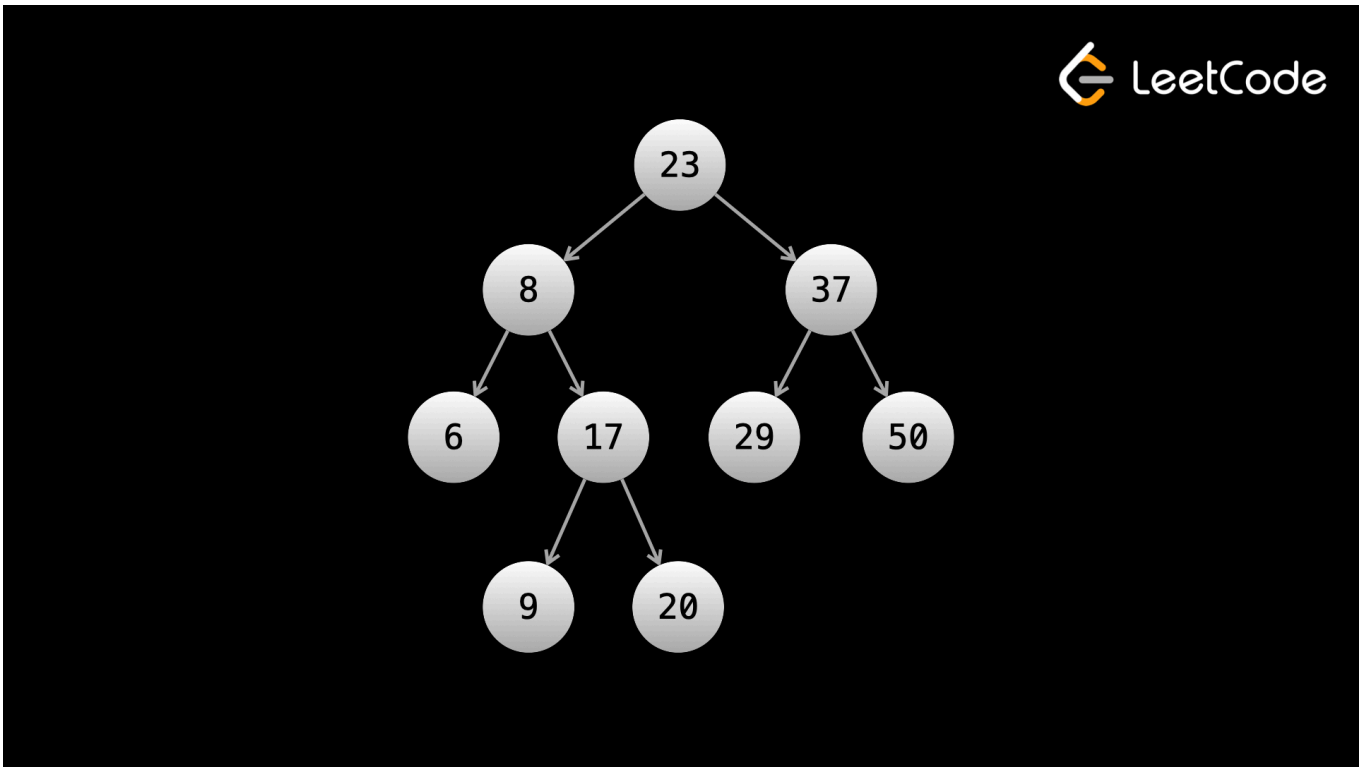
Binary search trees

A BST has the following property:

For each node, all values in its left subtree are less than the value in the node, and all values in its right subtree are greater than the value in the node.

This property implies that values in a BST must be unique.

An example of a BST:



With a binary search tree, operations like searching, adding, and removing can be done in $O(logn)$ time on average, where n is the number of nodes in the tree, using something called **binary search**, which is the focus of an upcoming chapter.

Searching for an element has an average time complexity of $O(logn)$. In the worst case scenario, let's say with a tree that had no `right` children and was just a straight line (basically a linked list), the time complexity would be $O(n)$.

Trivia to know: an inorder DFS traversal prioritizing left before right on a BST will handle the nodes in sorted order.

Example 1: 938 - Range Sum of BST

Given the `root` node of a **binary search tree** and two integers `low` and `high`, return the sum of values of all nodes with a value in the inclusive range [low, high].

The trivial approach would be to do a normal BFS or DFS, visit every node, and only add nodes whose values are between `low` and `high` to sum. However, we can make use of the BST property to develop a more efficient algorithm. In a BST, every node has a value greater than all nodes in the left subtree and a value less than all nodes in the right subtree. Therefore, if the current node's value is less than `low`, we know it is pointless to check the left subtree because all nodes in the left subtree will be out of range. Similarly, if the current node's value is greater than `high`, it is pointless to check the right subtree. This optimization can save a potentially huge amount of computation!

```
class Solution:
    def rangeSumBST(self, root: Optional[TreeNode], low: int, high: int) ->
int:
        if not root:
            return 0

        ans = 0
        if low <= root.val <= high:
            ans += root.val
        if low < root.val:
            ans += self.rangeSumBST(root.left, low, high)
        if root.val < high:
            ans += self.rangeSumBST(root.right, low, high)

        return ans
```

An iterative version:

```
class Solution:
    def rangeSumBST(self, root: Optional[TreeNode], low: int, high: int) ->
int:
        stack = [root]
        ans = 0
        while stack:
            node = stack.pop()
            if low <= node.val <= high:
                ans += node.val
```

```
            if node.left and low < node.val:
                stack.append(node.left)
            if node.right and node.val < high:
                stack.append(node.right)

        return ans
```

Although the time complexity is still $O(n)$ for the case when all nodes in the tree are between `low` and `high`, on average this algorithm will perform better than simply searching all nodes. For example, if you had a full tree with a million nodes, and the root's value was greater than `high`, then you can immediately save `500,000` visits based on the logic that all nodes in the right subtree are greater than the root's value which is already outside the range.

The space complexity is $O(n)$ for the stack/recursion call stack.

Example 2: 530 - Minimum Absolute Difference in BST

Given the `root` of a BST, return the minimum absolute difference between the values of any two different nodes in the tree.

One approach would be to go through the tree and put all the values in an array, then loop over all pairs of the array to find the minimum difference. This would be $O(n^2)$. A better approach would be to sort the array, and iterate over the adjacent elements. The answer must be between adjacent elements in the sorted array, so this improves the time complexity to $O(nlogn)$ due to the sort. Can we do better?

If you perform an inorder traversal on a BST, you will visit the nodes in sorted order. Therefore, if we do an inorder DFS, we can get the nodes in sorted order without the $O(nlogn)$ sort, resulting in an overall time complexity of $O(n)$.

We will pass an array `values` in our dfs function. To perform the inorder traversal, we first call on the left child, then add the current value to `values`, then call on the right child. This will add the values in sorted order.

```
class Solution:
    def getMinimumDifference(self, root: Optional[TreeNode]) -> int:
        def dfs(node):
            if not node:
                return
```

```
            left = dfs(node.left)
            values.append(node.val)
            right = dfs(node.right)

        values = []
        dfs(root)
        ans = float("inf")
        for i in range(1, len(values)):
            ans = min(ans, values[i] - values[i - 1])

        return ans
```

We mentioned earlier that for iterative DFS, preorder is easy but the other two are more difficult. Here's the iterative solution which involves the inorder traversal:

```
class Solution:
    def getMinimumDifference(self, root: Optional[TreeNode]) -> int:
        def iterative_inorder(root):
            stack = []
            values = []
            curr = root

            while stack or curr:
                if curr:
                    stack.append(curr)
                    curr = curr.left
                else:
                    curr = stack.pop()
                    values.append(curr.val)
                    curr = curr.right

            return values

        values = iterative_inorder(root)
        ans = float("inf")
        for i in range(1, len(values)):
            ans = min(ans, values[i] - values[i - 1])

        return ans
```

It is a lot more complicated to implement iteratively than recursively. So if ever given the option, you should stick with recursion. You *could* be asked to do both approaches in an interview, so it might be worth remembering the process for iterative inorder, but it is uncommon.

The time and space complexity to this approach is $O(n)$. We are able to get the values in sorted order in linear time by taking advantage of the BST property.

Example 3: 98 - Validate Binary Search Tree

Given the `root` of a binary tree, determine if it is a valid BST.

Using recursion, we can construct a function `dfs` that takes a `node` and returns true if the tree rooted at `node` is a BST. First, what arguments do we need to pass (other than the node)? In a BST, the root can be any value because it is not the child of any node, but every node in the left subtree should be less than it, and every node in the right subtree should be greater than it. To enforce this, we can use two integer arguments `small` and `large`, and make sure `small < node.val < large` holds.

If we are defining `(small, large)` as the interval for allowed values, how do we update them to maintain the BST property? At each `node`, the left subtree nodes should be less than `node.val`, so we can update `large = node.val`. The right subtree nodes should be greater than `node.val`, so we can update `small = node.val`. For the root node, we can initialize `small = -infinity` and `large = infinity` - the root can be any value since it has no parent.

What is the base case? An empty tree (no nodes) is technically a BST. Therefore, we can return true when the current node is `null`.

```
class Solution:
    def isValidBST(self, root: Optional[TreeNode]) -> bool:
        def dfs(node, small, large):
            if not node:
                return True

            if not (small < node.val < large):
                return False

            left = dfs(node.left, small, node.val)
            right = dfs(node.right, node.val, large)
```

```
            # tree is a BST if left and right subtrees are also BSTs
            return left and right

        return dfs(root, float("-inf"), float("inf"))
```

The return condition in our function being `AND` enforces all subtrees needing to also be BSTs.

Iterative version:

```
class Solution:
    def isValidBST(self, root: Optional[TreeNode]) -> bool:
        stack = [(root, float("-inf"), float("inf"))]
        while stack:
            node, small, large = stack.pop()
            if not (small < node.val < large):
                return False

            if node.left:
                stack.append((node.left, small, node.val))
            if node.right:
                stack.append((node.right, node.val, large))

        return True
```

The time and space complexity are both $O(n)$ for the same reasons.

SEE PROBLEMS 701, 270