(/cs/) (https://www.baeldung.com/cs/)

How to Validate a Binary Search Tree?

Last modified: August 25, 2021

by baeldung (https://www.baeldung.com/cs/author/baeldung)

Data Structures (https://www.baeldung.com/cs/category/data-structures)

Trees (https://www.baeldung.com/cs/category/graph-theory/trees)

Binary Tree (https://www.baeldung.com/cs/tag/binary-tree)

If you have a few years of experience in Computer Science or research, and you're interested in sharing that experience with the community, have a look at our **Contribution Guidelines** (/cs/contribution-guidelines).

1. Introduction

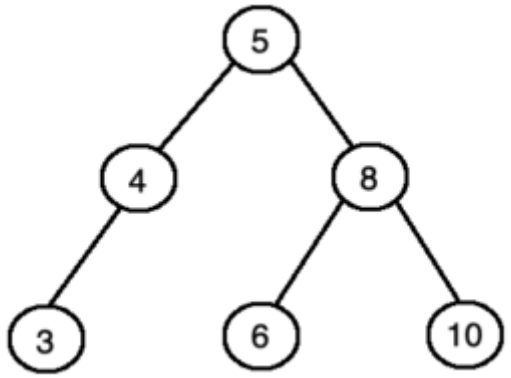
In this article, we'll discuss the problem of validating a binary search tree (/cs/binary-search-trees). After explaining what the problem is, we'll see a few algorithms for solving it. Then we'll see the pseudocode for these algorithms as well as a brief complexity analysis.

2. Problem Explanation

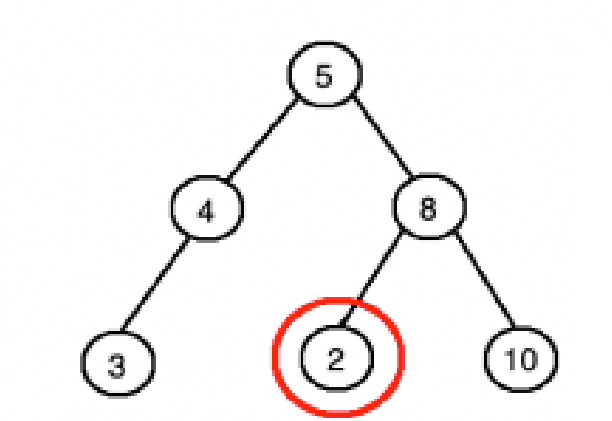
We're given as input a binary tree (/cs/binary-tree-intro) and would like to determine whether it's a valid binary search tree. In other words, we'll need to check four things:

- Is every node value in the root's left subtree less than the root's value?
- Is every node value in the root's right subtree greater than the root's value?
- Is the root's left subtree also a binary search tree?
- Is the root's right subtree also of a binary search tree?

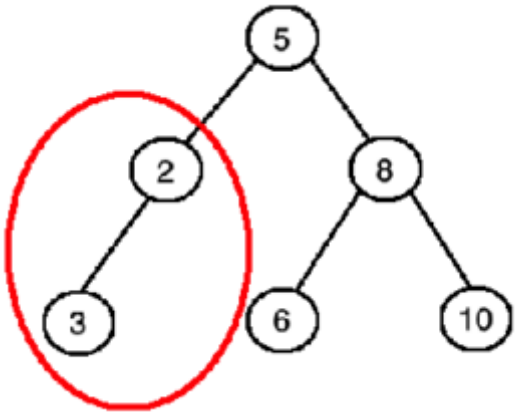
If these four conditions are met, then we can be sure that the binary tree is a binary search tree. For example, this tree is a binary search tree since the conditions are met:



Whereas this tree is not a binary search tree since we have a node value (circled in red) in the right subtree of the root which is less than the root's value:



Finally, this tree is not a binary search tree because the left subtree of the root (circled in red) is not a binary search tree:



3. Algorithms

3.1. Naive Approach

An extremely naive approach would be to traverse the tree and check at each node whether the left child's value is smaller and the right child's value is bigger. This approach is wrong because it'll only work on certain trees, not all trees.

A correct naive method (which is suggested by the conditions in the previous section) is this:

- If the tree is a single node, return *true*
- Traverse every node in the left subtree, checking whether each value is smaller than the root's value
- Traverse every node in the right subtree, checking whether each value is larger than the root's value
- If we found a node in the left subtree whose value is bigger than the root's or a node in the right subtree whose value is smaller than the root's, then return *false*
- Recursively check whether both the left and right subtrees of the root are also binary search trees and if yes then return *true*

3.2. Efficient Method Using Upper and Lower Limits

The previous algorithm is too slow and we can do better. **One possibility is to keep track of upper and lower limits for each node as we traverse the tree. At each node, we'll check whether that node's value falls within the current limits.**

Whenever we recurse on a left subtree we'll use the current node's value as the new upper limit and keep the original lower limit. If we recurse on a right subtree, we'll use the current node's value as the lower limit and keep the original upper limit. The inputs to the algorithm are $currentNode$ (we'll begin with the root), $lowerLimit$, and $upperLimit$.

So, let's outline this in detail:

- If $currentNode$ equals $null$, return $true$
- If $lowerLimit$ does not equal $null$ and the value at $currentNode$ is less than or equal to $lowerLimit$, return $false$
- If $upperLimit$ does not equal $null$ and the value at $currentNode$ is greater than or equal to $upperLimit$, return $false$
- Recursively check that the left subtree of $currentNode$ is a binary search tree, with the value of $currentNode$ as the upper limit and $lowerLimit$ as the lower limit. If the left subtree is not a binary search tree then return $false$.
- Recursively check that the right subtree of $currentNode$ is a binary search tree, with the value of $currentNode$ as the lower limit and $upperLimit$ as the upper limit. If the right subtree is not a binary search tree, then return $false$.
- Return $true$

The above algorithm is much faster than the previous one because we only visit each node exactly once.

3.3. Efficient Method Using Inorder Traversal

There is another algorithm we can use to solve this problem. **We can do an inorder traversal (https://en.wikipedia.org/wiki/Tree_traversal) of the given tree, storing the node values in a list, and then checking whether the list is sorted or not.**

If an inorder traversal produces a sorted order, then the tree must be a binary search tree. But why is this the case and how can we know for sure? We can answer these questions using mathematical induction (https://en.wikipedia.org/wiki/Mathematical_induction).

The base case is that if we have a single node then the statement holds. Inorder traversal of a single node will always return a sorted order, and a single node will always be a binary search tree.

Now we can inductively assume that the statement holds for any left subtree and any right subtree. Using this inductive hypothesis along with the base case, we can show that the original statement holds.

If the inorder traversal of our tree returns a sorted order, then the inorder traversal of the left and right subtrees must've been in sorted order as well. This is because the inorder traversal will always traverse the entire left subtree, then the root, and finally the entire right subtree.

This also means that every node value in the left subtree must be smaller than the root's value, and every node value in the right subtree must be larger than the root's value. Also, we know inductively that both the left and right subtrees must be binary search trees, so we're done with our proof.

This inorder traversal based method can be further improved by noticing that we do not have to store all the node values in a list. We can simply always check whether the previous element in the traversal is less than the current element.

4. Pseudocode

Here we'll show pseudocode for four different algorithms.

4.1. Naive Algorithm

The algorithm below is the naive approach where we first traverse the left and right subtrees and then recursively check whether the two subtrees are binary search trees:

Algorithm 1: Naive algorithm to validate binary search tree

```
1 function validateBST(root)
2     if root != null then
3         if valid(root.left, root.val, true) and valid(root.right, root.val,
            false) then
4             return validateBST(root.left) and
                validateBST(root.right);
```

```
5        else
6            return false;
7        end
8    end
9    return true;
10 end

11 function valid(root, nodeVal, lessThan)
12   if root != null then
13     if lessThan then
14       if root.val >= nodeVal then
15         return false;
16       end
17       return valid(root.left, nodeVal, lessThan) and
           valid(root.right, nodeVal, lessThan);
18     else
19       if root.val <= nodeVal then
20         return false;
21       end
22       return valid(root.left, nodeVal, lessThan) and
           valid(root.right, nodeVal, lessThan);
23     end
24   end
25   return true;
26 end
```

In the above code, we can see that there are two functions: $validateBST$ and $valid$. $validateBST$ takes as input the root node of a binary tree and returns $true$ if the given binary tree is a valid binary search tree. In line 3, we have a condition that checks whether all the node values in the left subtree are smaller than the root's value and all the node values in the right subtree are larger than the root's value.

If this condition is met then on line 4 we recursively check whether both the left and right subtrees are also binary search trees. Otherwise, we return $false$. Finally, we return $true$ if the root is equal to $null$.

The function $valid$ takes as input the root node of a subtree, the value that we want to compare all the subtree values to, and a boolean variable which tells us whether the values in this subtree should be less than or greater the given value. This function iterates through every node of the given subtree and checks whether all the nodes in the subtree are less than or greater than the given value (depending on whether $lessThan$ is $true$ or $false$).

4.2. Efficient Algorithm Using Upper and Lower Limits

Here, we're using the idea of lower and upper limits and traverse each node exactly once:

Algorithm 2: Algorithm to validate binary search tree using upper and lower limits

```
1  function isValid(node, low, high)
2      if node == null then
3          return true;
4      end
5      if low != null and node.val <= low then
6          return false;
7      end
8      if high != null and node.val >= high then
9          return false;
10     end
11     if isValid(node.right, node.value, high) == false then
12         return false;
13     end
14     if isValid(node.left, low, node.value) == false then
15         return false;
16     end
17     return true;
18 end
```

Algorithm 2 first checks whether the tree is empty. If it is we return $true$ since an empty tree is a valid binary search tree.

Then in lines 5 through 10 we have two if statements that check whether the current node's value is within the lower and upper bounds.

Lines 11 through 16 contain two if statements which recursively check whether the left and right subtrees also obey the lower and upper bounds. **Note that for the left subtree the new upper bound is the current node's value, and for the right subtree the new lower bound is the current node's value.**

Finally, we return $true$ if we get past all the if statements since this means the tree must be a binary search tree.

4.3. Efficient Algorithm Using Inorder Traversal

This algorithm uses an inorder traversal and stores all the node values in a list. Then it checks whether the list is sorted:

Algorithm 3: Algorithm to validate binary search tree using in-order traversal

```
1  function isValid(root)
2      list = [];
3      inorder(root, list);
4      return isSorted(list);
5  end
6  function inorder(root, list)
7      if root != null then
8          inorder(root.left, list);
9          list.add(root.value);
10         inorder(root.right, list);
11     end
12 end
13 function isSorted(list)
14     for i = 1 to list.size() - 1 do
15         if list[i] <= list[i − 1] then
16             return false;
17         end
18     end
19     return true;
20 end
```

4.4. Space-Optimized Inorder Traversal Based Algorithm

The algorithm below is an optimized version of the above algorithm in the sense that it does not use an extra list to store the values:

Algorithm 4: Optimized algorithm to validate binary search tree using inorder traversal

```
1  function isValid(root)
2      isValid = [true];
3      prev = [null];
4      inorder(root, prev, isValid);
5      return isValid[0];
6  end
7  function inorder(root, prev, isValid)
8      if root != null then
9          inorder(root.left, prev, isValid);
10         if prev[0]! = null and prev[0].value >= root.value then
11             isValid[0] = false;
12             return;
13         end
14         prev[0] = root;
15         inorder(root.right, prev, isValid);
16     end
17 end
```

In this last algorithm, the function $inorder$ takes as input the root node, an array $prev$ which stores the previous node in the traversal as its only element, and an array $isValid$ of one boolean which indicates whether the tree is valid or not.

Initially, the boolean variable inside $isValid$ is set to true. If at any point in the traversal the previous node's value is greater than or equal to the current value, then the boolean in $isValid$ is set to false and we quit the $inorder$ function by returning.

5. Complexity

The naive algorithm is the slowest of all four algorithms. Its time complexity is equal to $O(n^2)$ where n is the number of nodes in the tree.

We'll show that the naive algorithm has a worst-case time complexity of $O(n^2)$ by showing an example where it is $O(n^2)$ and then showing that it cannot be worse than $O(n^2)$.

What is a case when the running time is $O(n^2)$? One possibility is if the tree happens to be a path of length $n-1$. Since we're iterating through all the descendants of each node, the total number of times we visit some node in such a path is equal to $n-1+n-2+n-3+...+1=O(n^2)$. This is because in a path of length $n-1$ the root node has $n-1$ descendants, the second node has $n-2$ descendants, and so on.

How do we know that the worst-case complexity of the naive algorithm cannot be worse than $O(n^2)$?

The maximum number of descendants possible for any node is $n-1$ since there are n nodes in the tree. So even if every node had $n-1$ descendants, we would still only have $n(n-1)$ iterations in the algorithm. This is still $O(n^2)$.

The algorithm which uses upper and lower limits has a time complexity of $O(n)$ since we visit each node exactly once.

In the inorder traversal based algorithm, we perform an inorder traversal which takes $O(n)$ time. Then we check whether a list is sorted which also takes $O(n)$ time. This gives us an overall time complexity of $O(n)$.

Finally, the space-optimized inorder traversal based algorithm has a time complexity of $O(n)$ since we're simply doing an inorder traversal with constant-time operations.

6. Conclusion

In this article, we have seen four different algorithms for validating a binary search tree. We have also seen pseudocode for these algorithms as well as a time complexity analysis. **Our optimal solution for this problem runs in $O(n)$ time.**

Comments are closed on this article!

CATEGORIES

SERIES

DRAWING CHARTS IN LATEX (/CS/CATEGORY/SERIES)

ABOUT

ABOUT BAELDUNG (HTTPS://WWW.BAELDUNG.COM/ABOUT)
THE FULL ARCHIVE (/CS/FULL_ARCHIVE)
WRITE FOR BAELDUNG (/CS/CONTRIBUTION-GUIDELINES)
EDITORS (HTTPS://WWW.BAELDUNG.COM/EDITORS)

TERMS OF SERVICE (HTTPS://WWW.BAELDUNG.COM/TERMS-OF-SERVICE)
PRIVACY POLICY (HTTPS://WWW.BAELDUNG.COM/PRIVACY-POLICY)
COMPANY INFO (HTTPS://WWW.BAELDUNG.COM/BAELDUNG-COMPANY-INFO)
CONTACT (/CONTACT)