Create Balanced Binary Search Tree From Sorted List

Last modified: August 25, 2021

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

Algorithms (https://www.baeldung.com/cs/category/algorithms)

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. Overview

In this tutorial, we'll discuss creating a balanced binary search tree (BST (/cs/binary-search-trees)) from a sorted list. Firstly, we'll explain the meaning of balanced binary search trees.

Then, we'll discuss the top-down and bottom-up approaches and compare them.

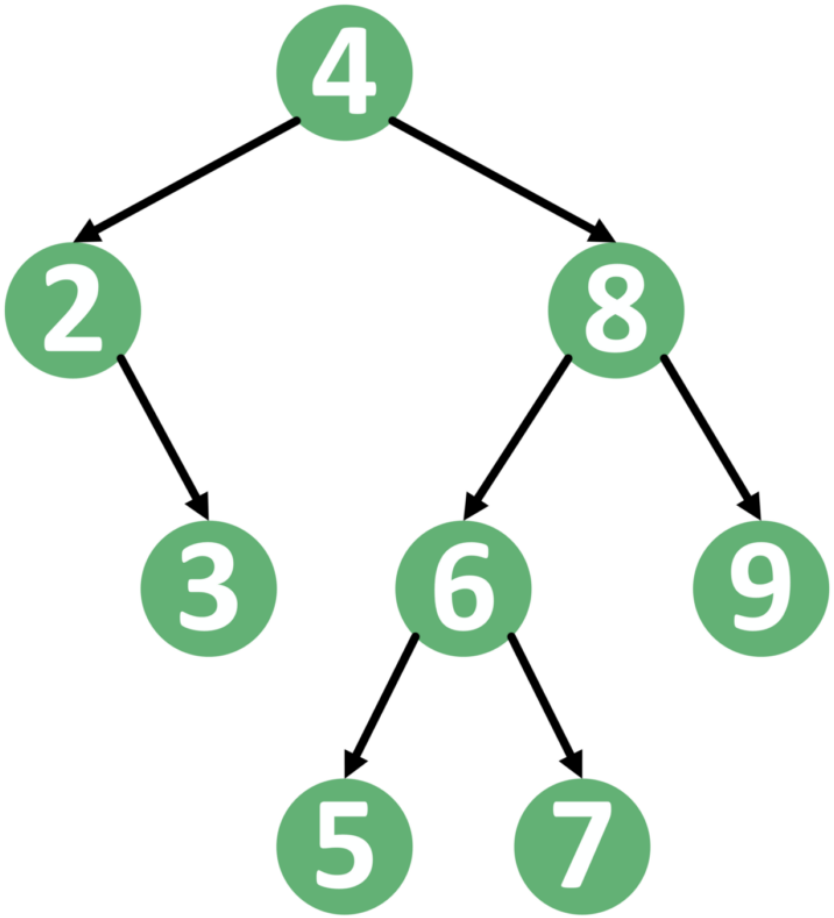
2. Balanced Binary Seach Tree

In the beginning, let's define the meaning of balanced binary search trees. **A balanced (/java-balanced-binary-tree) binary tree is a tree whose height is** $O(log(n))$, where n is the number of nodes inside the tree.

For each node inside the balanced tree, the height of the left subtree mustn't differ by more than one from the height of the right subtree.

On the other hand, **a binary search tree is a binary tree, where the left subtree of each node contains values smaller than the root of the subtree. Similarly, the right subtree contains values larger than the root of the subtree**.

Let's take an example of a balanced binary search tree:



As we can see, the left subtree of the node 4 contains nodes 2 and 3, which are smaller than 4. Similarly, the right subtree contains nodes $5, 6, 7, 8$, and 9, which are larger than 4.

We can note that this condition holds for all nodes inside the tree. Therefore, this is a binary search tree.

On the other hand, we can note that the left subtree of the node 4 has a height equal to 2. Similarly, the height of the right subtree of the node 4 is 3. Hence, the difference is 1.

We can also note that this condition holds for all nodes inside the tree. Hence, this is a balanced binary search tree.

3. Creating a Balanced BST

When creating a balanced BST we need to keep the height condition in mind. First of all, let's think about the best node to put as the root.

Since we need the tree to be balanced, **we must put the middle value as the root**. After that, **we can add the values before the middle to the left of the tree.** Therefore, all smaller values will be added to the left subtree.

Similarly, **we'll add the values after the middle to the right of the tree**, which will cause all larger values to be added to the right subtree.

However, getting the middle element depends on the type of list we have. If the list supports random access, like an array, we can simply get the middle element, put it as the root, then recursively add the left and right subtrees. We call this the top-down approach.

However, if we only have a pointer to the first element in the list, getting the middle element is a little trickier. Therefore, we'll use the bottom-up approach in this case.

First, we'll discuss the top-down approach. Then, we'll move to the bottom-up approach.

4. Top-Down Approach

The top-down approach uses a sorted array to create a balanced BST. Therefore, we can access any index inside the array in constant time.

Let's have a look at the top-down approach to creating a balanced BST:

Algorithm 1: Top-Down Approach to Create a Balanced BST

```
Data: A: The sorted array
      L: The left side of the current range
      R: The right side of the current range
Result: The root of the balanced BST
Function build(A, L, R):
    if L > R then
        return null;
    end
    mid ← (L + R) / 2;
    root.value ← A[mid];
    root.left ← build(A, L, mid − 1);
    root.right ← build(A, mid + 1, R);
    return root;
end
```

The initial call for this function passes the array A, the value of L is zero and R is $n-1$, where n is the size of the array.

In the beginning, we check to see if we reached an empty range. In this case, we simply return $null$, indicating an empty object.

Next, we get the middle index and initiate the root node with $A[mid]$. After that, we make two recursive calls. **The first call is for the range** $[L, mid-1]$, which represents the elements before the index mid. On the other hand, **the second call is for the range** $[mid+1, R]$, which corresponds to the elements after the index mid.

The first call returns the root of the left subtree. Therefore, we assign its value to the left pointer of the root node. Similarly, the second call returns the root of the right subtree. Hence, we assign its value to the right pointer of the root node.

The complexity of the top-down approach is $O(n)$, where n is the number of elements inside the array.

5. Bottom-Up Approach

The bottom-up approach uses a linked list to build a balanced BST. As a result, we'll have a pointer to the head of the list and we can only move this pointer forward in constant time.

Take a look at the bottom-up approach to creating a balanced BST:

Algorithm 2: Bottom-Up Approach to Create a Balanced BST

```
Data: head: Pointer to the first element in the linked list
      L: The left side of the current range
      R: The right side of the current range
Result: The root of the balanced BST
Function build(head, L, R):
    if L > R then
        return null;
    end
    mid ← (L + R) / 2;
    root.left ← build(head, L, mid − 1);
    root.value ← head.data;
    head ← head.next;
    root.right ← build(head, mid + 1, R);
    return root;
end
```

The initial call for this function is similar to the top-down approach. We'll pass a pointer the beginning of the list $head$, the value of L is zero and R is $n - 1$, where n is the number of elements.

First, we check whether we have reached an empty range. If so, we return $null$ indicating an empty tree.

Then, We build the left subtree recursively. Next, since the left subtree is completely built, it means the $head$ pointer is now pointing to the element with index mid. Therefore, we store its value inside the root.

After that, we move the $head$ pointer one step forward because when the recursive call for the left subtree finished, we assumed that the pointer was on the mid element. Hence, we need to move the pointer forward so that the next recursive call can use it from there.

Finally, we perform a recursive call to build the right subtree and then return the root node.

The reason for calling this approach bottom-up is that we keep going into recursive calls until we reach the leftmost node. We create this node and then move to other recursive calls to build their nodes as well.

The complexity of the top-down approach is $O(n)$, where n is the number of elements inside the linked list.

6. Comparison

	Top-Down	Bottom-Up
Main Idea	Create the middle node. Then, recursively create the left and right subtrees	Keep going recursively to the leftmost node. Create it, and move to the next node
Data Structure	Sorted array	Sorted linked list
Complexity	$O(n)$	$O(n)$

Usually, the top-down approach is considered easier to understand and implement, because it's straight forward. On the other hand, the bottom-up approach needs a little more understanding of the recursive calls, and how exactly is the tree built from the leftmost to the rightmost node.

Both approaches have the same time complexity. However, each approach is based on a different data structure.

If the data is inside a sorted array, then using the top-down approach is usually easier than the bottom-up approach. On the other hand, if the data is inside a sorted linked list, then we need to use the bottom-up approach.

7. Conclusion

In this tutorial, we presented two approaches to building a balanced binary search tree from a sorted list.

Firstly, we explained the general concept of balanced binary search trees. Secondly, we presented the top-down approach and the bottom-up approach.

In the end, we compared both approaches and showed when to use each one.

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).

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)