Theory: Binary search tree

20 minutes 10 / 10 problems solved

Start practicing

380 users solved this topic. Latest completion was **about 1 hour ago**.

A **binary search tree** is a **node-based** tree that has the following properties:

- for any node, values of all nodes within its left subtree are strictly **less** than the value of that node
- for any node, values of all nodes within its right subtree are **greater** than or equal to the value of that node
- any subtree of a binary search tree is also a binary search tree.

In other words, in a binary search tree nodes branch into **subtrees** ("children"), where the left nodes take a smaller value and the right nodes are greater or equal to the "parent". The uppermost node is called **root**, and the nodes with no subtrees are called **leaves**.

Current topic:

✓ Binary search tree

Topic depends on:

✓ Tree

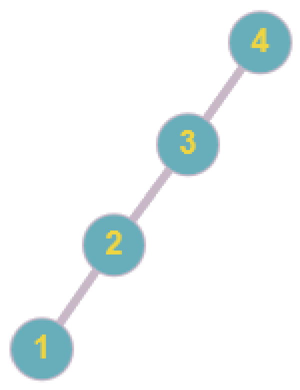
Topic is required for:

Binary search tree in Java

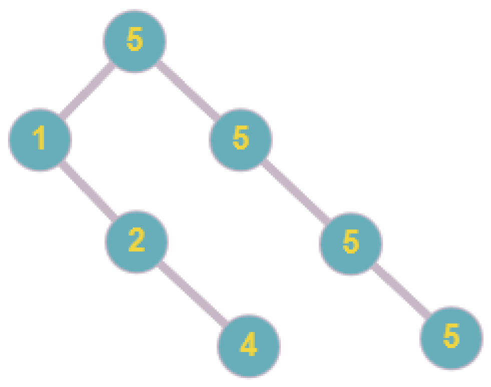
Table of contents:

§1. Examples

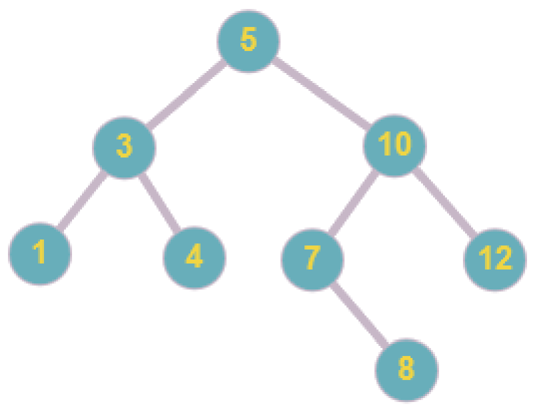
Take a look at some examples of binary search trees, note nodes' values and positions:



The root of the tree has the value of '4'. Since there is no right subtree, '4' is the highest value. The leftmost node has the smallest value of '1'.

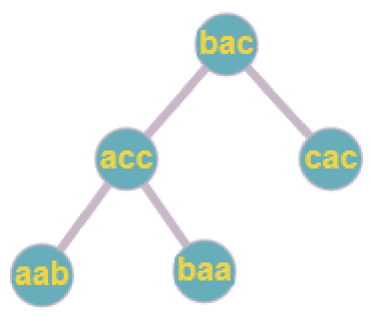


Nodes of the right subtree can take a value equal to their parent.



Each of subtrees is a binary search tree itself; they follow the same pattern: smaller values go to the left, and equal/greater values go to the right.

The **value** of a node can be of any type for which a comparison operator is defined: numbers, characters, strings, or objects (as long as they have a specific comparator). For example, here is a binary search tree with strings as its elements:



The value can also be a complex object as a user, a product, a search result, etc.

§2. Basic operations

There are three basic operations on a binary search tree:

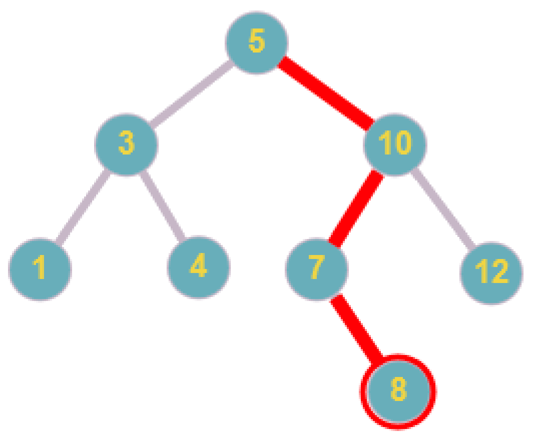
- **find** (key): check whether the tree contains the specified key
- **insert** (key): add a node with the specified key to the tree.
- **remove** (key): remove a node with the specified key from the tree, if such node exists

In all three algorithms of basic operations we start from the root and, depending on the value of the current node, we move down the tree either to the left or the right subtree. Let's take a closer look at the basic operations.

§3. Find operation

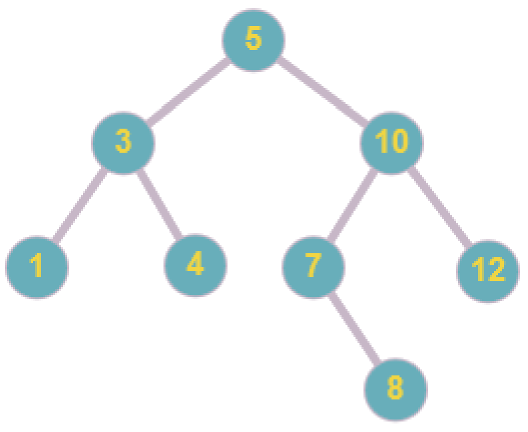
If the value of a node coincides with the key, we get the message that the item is found. Otherwise, depending on whether the value of a node is less or greater than the key, we move to the left or right subtree respectively and repeat the action.

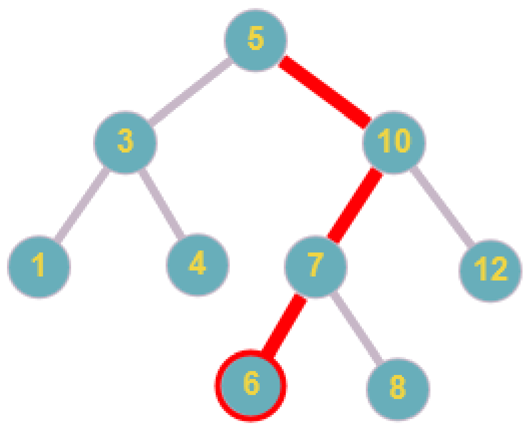
The algorithm ends when we find the element containing the key or get to tree leaves and find no node containing the key. The image below shows a search for an item with the key "8".



§4. Insert operation

If the key is less than the value of the current node, go down to its left subtree, otherwise move to the right. We repeat this action until we cannot go down any further. If there is no subtree to which we can descend, then we insert here a node with the key. The picture below shows the insertion of the element "6".



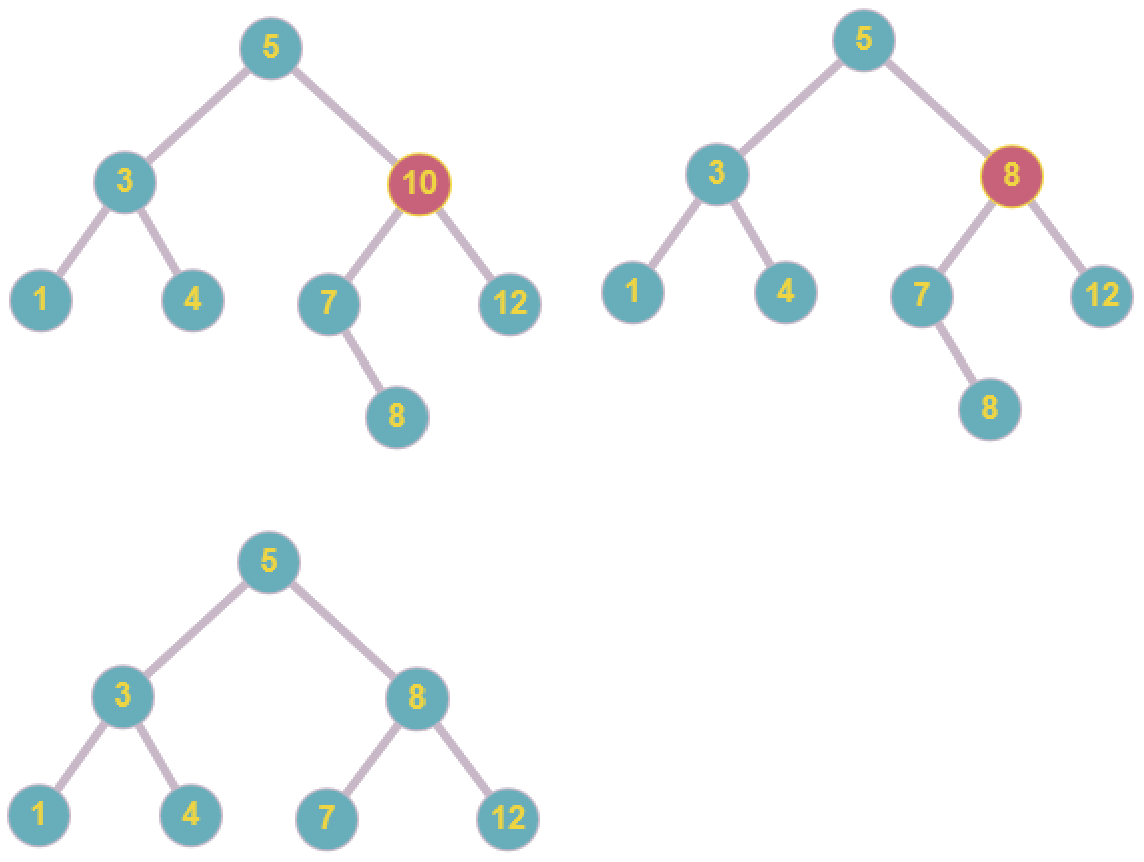


§5. Remove operation

This operation is a bit more complicated since here we have to consider several possible options. First, using the search operation, we find the position of the element which is to be deleted. If it does not exist, there is nothing left to do. If it does, there are three possible options:

1. if the node has **no children**, we simply delete it;
2. if the node has only **one child** (either right or left), we delete the node and put its child in its place;
3. if the node has **both left and right children**, keep calm and follow the steps: first, identify the node that you want to delete, then find the rightmost child in its left subtree. Put its value into the initial node and remove the duplicate in the bottom by using the algorithm for either the first or the second case. Note that you may use the leftmost child from the right subtree instead; whichever option you choose, moving those two nodes will not break tree structure.

Take a look at how the node with both children is removed (this corresponds to the third option in the algorithm). Suppose we want to remove node 10: we find the rightmost node in its left subtree (which in our case is 8), put it in place of the deleted node and finally remove the duplicate:



The variety of possible options can make the task look difficult from the first glance, but everything will become clear to you as you practice.

There are more complex operations on a binary search tree such as split, merge, etc., but their implementation is based on the use of three basic operations: find, insert, and remove.

§6. Conclusion

We have covered the properties of binary search trees, some terms, and three basic operations. All these operations have an **O(log n) complexity** on average and **O(n)** in the worst case. The latter can be achieved by inserting a sequence of non-decreasing values in the tree. In this case, the tree will become a linear list, and all operations on this tree will work for **O(n)**. To achieve **O(log n)** complexity in the worst cases, you must maintain tree balance. You will learn how to do this in future lessons.

Report a typo

44 users liked this theory. **0** didn't like it. **What about you?**

Start practicing

Comments (2) Hints (0) Useful links (1) Show discussion