DSA Binary Search Trees

A **Binary Search Tree** is a Binary Tree where every node's left child has a lower value, and every node's right child has a higher value.

A clear advantage with Binary Search Trees is that operations like search, delete, and insert are fast and done without having to shift values in memory.

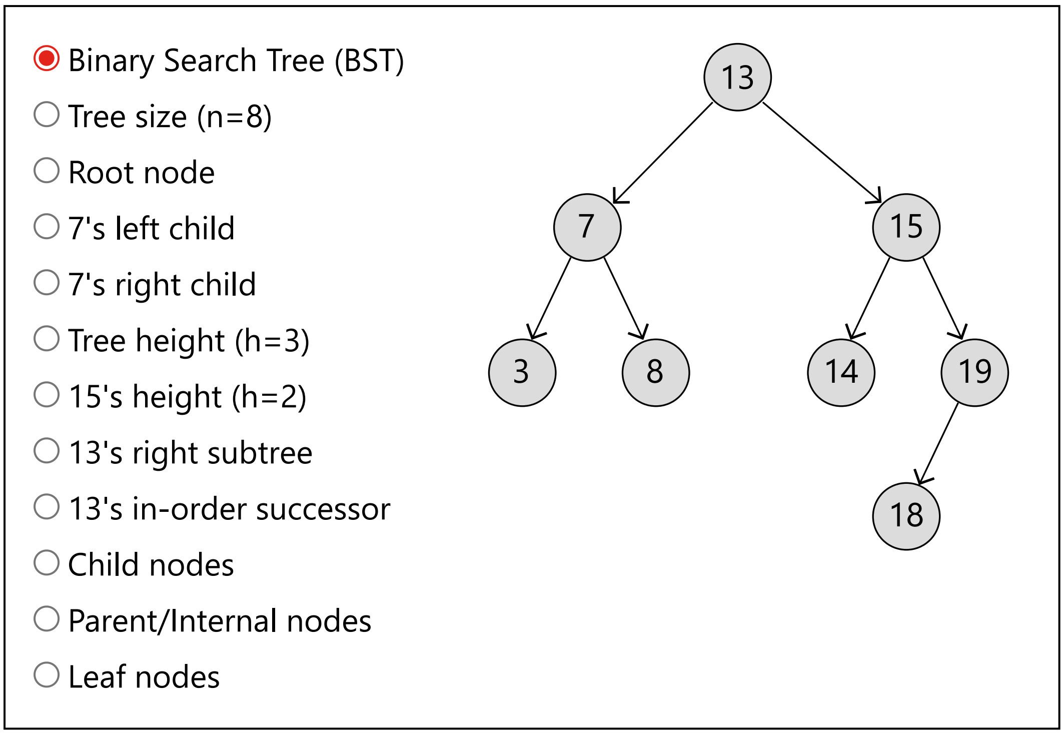
Binary Search Trees

A Binary Search Tree (BST) is a type of Binary Tree data structure, where the following properties must be true for any node "X" in the tree:

- The X node's left child and all of its descendants (children, children's children, and so on) have lower values than X's value.
- The right child, and all its descendants have higher values than X's value.
- Left and right subtrees must also be Binary Search Trees.

These properties makes it faster to search, add and delete values than a regular binary tree.

terminology.



The **size** of a tree is the number of nodes in it (n).

A **subtree** starts with one of the nodes in the tree as a local root, and consists of that node and all its descendants.

The **descendants** of a node are all the child nodes of that node, and all their child nodes, and so on. Just start with a node, and the descendants will be all nodes that are connected below that node.

The **node's height** is the maximum number of edges between that node and a leaf node.

A **node's in-order successor** is the node that comes after it if we were to do in-order traversal. In-order traversal of the BST above would result in node 13 coming before node 14, and so the successor of node 13 is node 14.

Traversal of a Binary Search Tree

Just to confirm that we actually have a Binary Search Tree data structure in front of us, we can check if the properties at the top of this page are true. So for every node in the figure above, check if all the values to the left of the node are lower, and that all values to the right are higher.

traversal.

Example

Python:

```
class TreeNode:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

def inOrderTraversal(node):
    if node is None:
        return
    inOrderTraversal(node.left)
    print(node.data, end=", ")
    inOrderTraversal(node.right)

root = TreeNode(13)
node7 = TreeNode(7)
node15 = TreeNode(15)
node3 = TreeNode(3)
node8 = TreeNode(8)
node14 = TreeNode(14)
node19 = TreeNode(19)
node18 = TreeNode(18)

root.left = node7
root.right = node15

node7.left = node3
node7.right = node8

node15.left = node14
node15.right = node19

node19.left = node18
```

Try it Yourself »

As we can see by running the code example above, the in-order traversal produces a list of numbers in an increasing (ascending) order, which means that this Binary Tree is a Binary Search Tree.

Search for a Value in a BST

Searching for a value in a BST is very similar to how we found a value using Binary Search on an array.

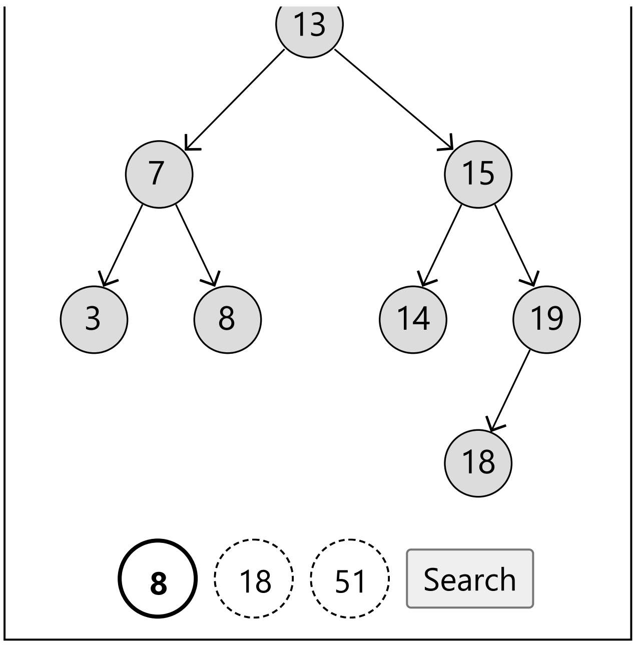
For Binary Search to work, the array must be sorted already, and searching for a value in an array can then be done really fast.

Similarly, searching for a value in a BST can also be done really fast because of how the nodes are placed.

How it works:

1. Start at the root node.
2. If this is the value we are looking for, return.
3. If the value we are looking for is higher, continue searching in the right subtree.
4. If the value we are looking for is lower, continue searching in the left subtree.
5. If the subtree we want to search does not exist, depending on the programming language, return `None`, or `NULL`, or something similar, to indicate that the value is not inside the BST.

Use the animation below to see how we search for a value in a Binary Search Tree.



The algorithm above can be implemented like this:

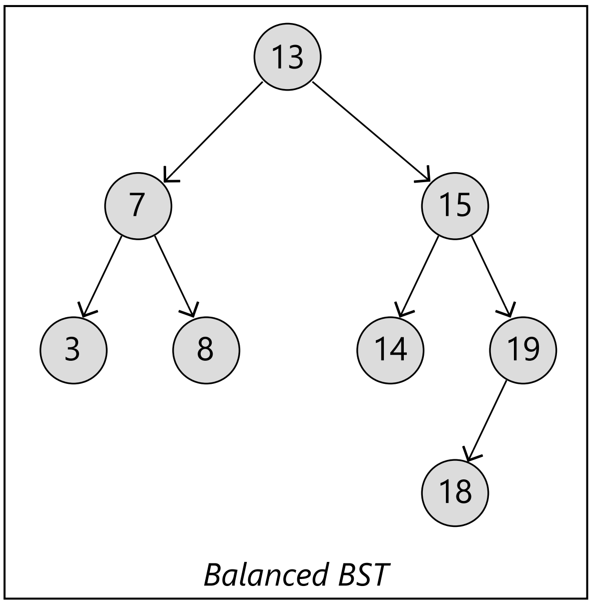
Example

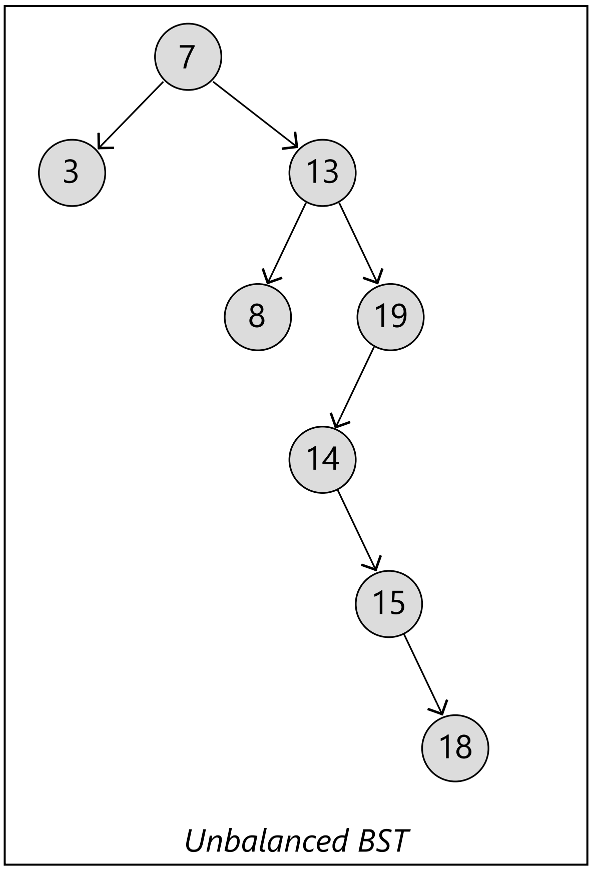
Python:

```
def search(node, target):
    if node is None:
        return None
    elif node.data == target:
        return node
    elif target < node.data:
        return search(node.left, target)
    else:
        return search(node.right, target)
```

Try it Yourself »

The time complexity for searching a BST for a value is $O(h)$, where h is the height of the tree.





Both Binary Search Trees above have the same nodes, and in-order traversal of both trees gives us the same result but the height is very different. It takes longer time to search the unbalanced tree above because it is higher.

We will use the next page to describe a type of Binary Tree called AVL Trees. AVL trees are self-balancing, which means that the height of the tree is kept to a minimum so that operations like search, insertion and deletion take less time.

Insert a Node in a BST

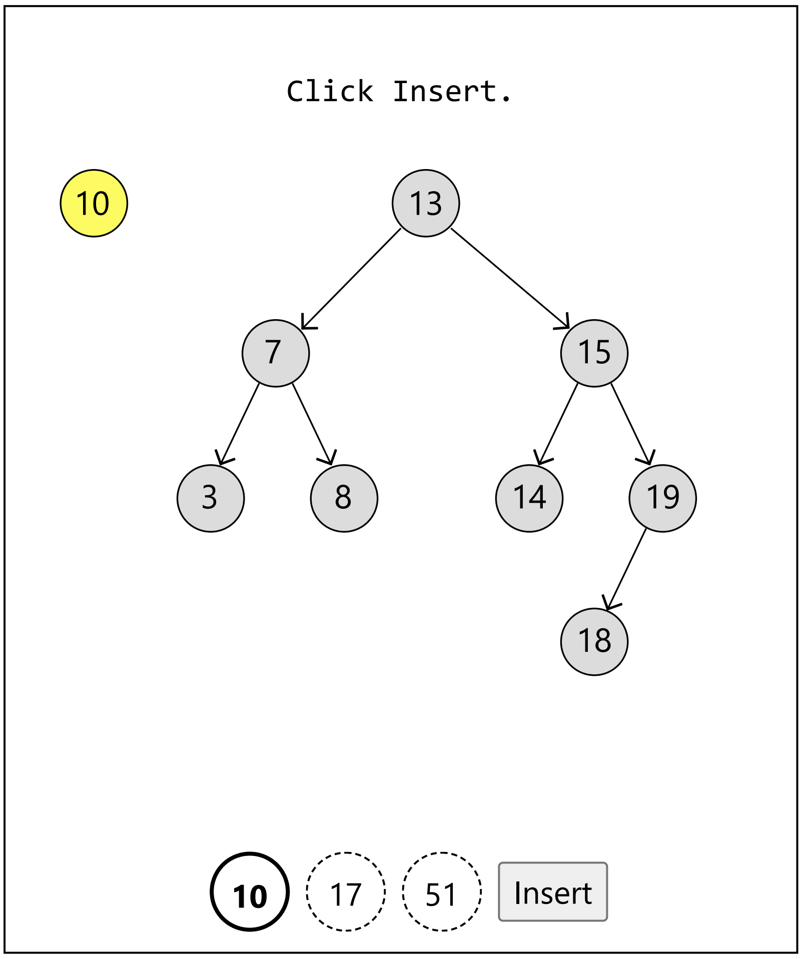
Inserting a node in a BST is similar to searching for a value.

How it works:

1. Start at the root node.
2. Compare each node:
   - Is the value lower? Go left.

Inserting nodes as described above means that an inserted node will always become a new leaf node.

Use the simulation below to see how new nodes are inserted.



All nodes in the BST are unique, so in case we find the same value as the one we want to insert, we do nothing.

This is how node insertion in BST can be implemented:

Example

Python:

```
def insert(node, data):
    if node is None:
        elif data > node.data:
            node.right = insert(node.right, data)
    return node
```

Try it Yourself »

Find The Lowest Value in a BST Subtree

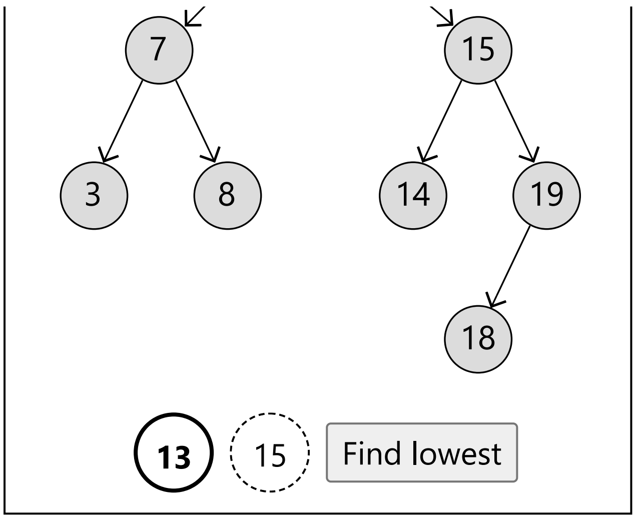
The next section will explain how we can delete a node in a BST, but to do that we need a function that finds the lowest value in a node's subtree.

How it works:

1. Start at the root node of the subtree.
2. Go left as far as possible.
3. The node you end up in is the node with the lowest value in that BST subtree.

In the figure below, if we start at node 13 and keep going left, we end up in node 3, which is the lowest value, right?

And if we start at node 15 and keep going left, we end up in node 14, which is the lowest value in node 15's subtree.



This is how the function for finding the lowest value in the subtree of a BST node looks like:

Example

Python:

```python
def minValueNode(node):
    current = node
    while current.left is not None:
        current = current.left
    return current
```

Try it Yourself »

We will use this `minValueNode()` function in the section below, to find a node's in-order successor, and use that to delete a node.

Delete a Node in a BST

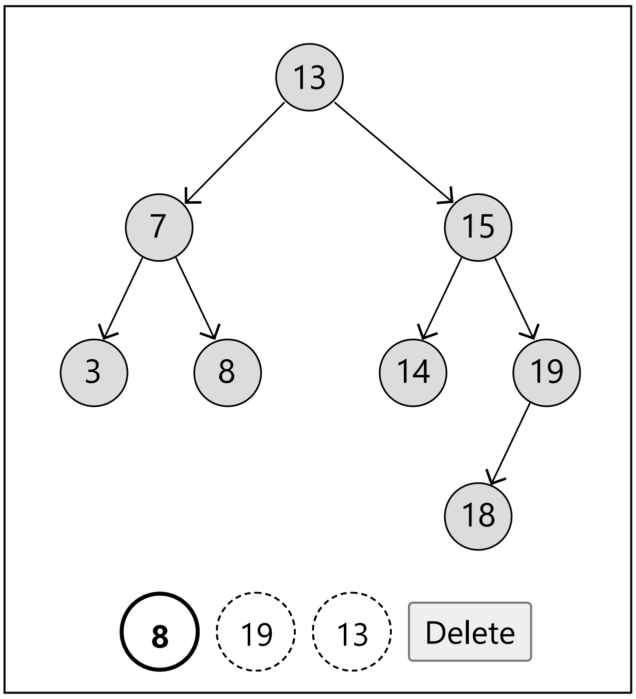
To delete a node, our function must first search the BST to find it.

After the node is found there are three different cases where deleting a node must be done differently.

1. If the node is a leaf node, remove it by removing the link to it.
2. If the node only has one child node, connect the parent node of the node you want to remove to that child node.
3. If the node has both right and left child nodes: Find the node's in-order successor, change values with that node, then delete it.

In step 3 above, the successor we find will always be a leaf node, and because it is the node that comes right after the node we want to delete, we can swap values with it and delete it.

Use the animation below to see how different nodes are deleted.



Node 8 is a leaf node (case 1), so after we find it, we can just delete it.

Node 19 has only one child node (case 2). To delete node 19, the parent node 15 is connected directly to node 18, and then node 19 can be removed.

Node 13 has two child nodes (case 3). We find the successor, the node that comes right after during in-order traversal, by finding the lowest node in node 13's right subtree, which is node 14. Value 14 is put into node 13, and then we can delete node 14.

This is how a BST can be implemented with functionality for deleting a node:

```
 1  def delete(node, data):
 2      if not node:
 3          return None
 4  
 5      if data < node.data:
 6          node.left = delete(node.left, data)
 7      elif data > node.data:
 8          node.right = delete(node.right, data)
 9      else:
10          # Node with only one child or no child
11          if not node.left:
12              temp = node.right
13              node = None
14              return temp
15          elif not node.right:
16              temp = node.left
17              node = None
18              return temp
19  
20          # Node with two children, get the in-order successor
21          node.data = minValueNode(node.right).data
22          node.right = delete(node.right, node.data)
23  
24      return node
```

Try it Yourself »

Line 1: The `node` argument here makes it possible for the function to call itself recursively on smaller and smaller subtrees in the search for the node with the `data` we want to delete.

Line 2-8: This is searching for the node with correct `data` that we want to delete.

Line 9-22: The node we want to delete has been found. There are three such cases:

> **Case 1**: Node with no child nodes (leaf node). `None` is returned, and that becomes the parent node's new left or right value by recursion (line 6 or 8).

value of the node we want to delete, and then we can delete the successor node.

Line 24: node is returned to maintain the recursive functionality.

BST Compared to Other Data Structures

Binary Search Trees take the best from two other data structures: Arrays and Linked Lists.

Data Structure	**Searching for a value**	**Delete / Insert leads to shifting in memory**
Sorted Array	$\boldsymbol{O(\log n)}$	Yes
Linked List	$O(n)$	**No**
Binary Search Tree	$\boldsymbol{O(\log n)}$	**No**

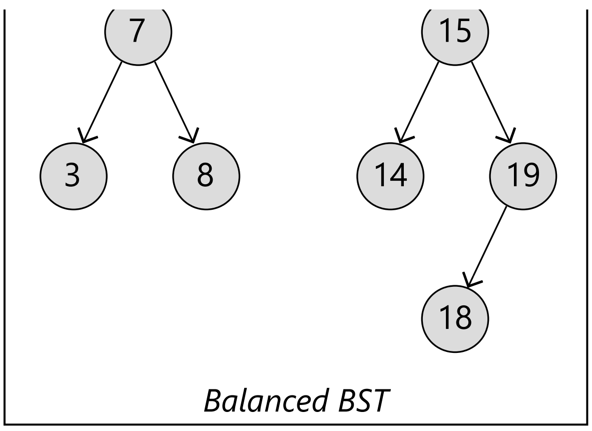
Searching a BST is just as fast as Binary Search on an array, with the same time complexity $O(\log n)$.

And deleting and inserting new values can be done without shifting elements in memory, just like with Linked Lists.

BST Balance and Time Complexity

On a Binary Search Tree, operations like inserting a new node, deleting a node, or searching for a node are actually $O(h)$. That means that the higher the tree is (h), the longer the operation will take.

The reason why we wrote that searching for a value is $O(\log n)$ in the table above is because that is true if the tree is "balanced", like in the image below.

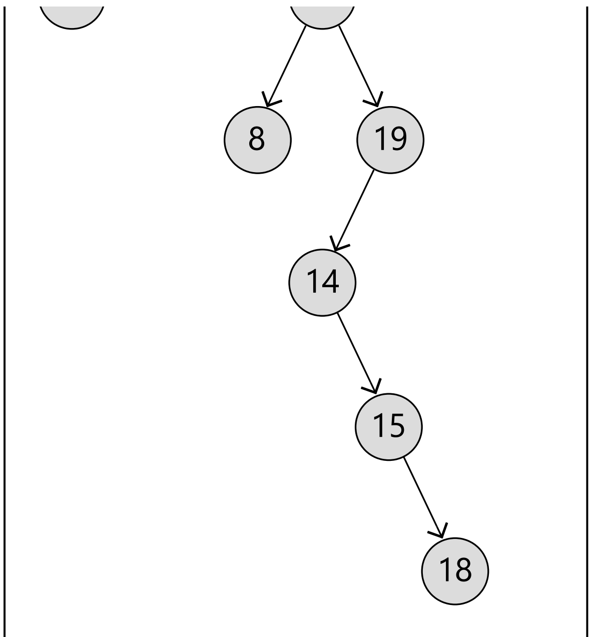


We call this tree balanced because there are approximately the same number of nodes on the left and right side of the tree.

The exact way to tell that a Binary Tree is balanced is that the height of the left and right subtrees of any node only differs by one. In the image above, the left subtree of the root node has height $h = 2$, and the right subtree has height $h = 3$.

For a balanced BST, with a large number of nodes (big n), we get height $h \approx \log_2 n$, and therefore the time complexity for searching, deleting, or inserting a node can be written as $O(h) = O(\log n)$.

But, in case the BST is completely unbalanced, like in the image below, the height of the tree is approximately the same as the number of nodes, $h \approx n$, and we get time complexity $O(h) = O(n)$ for searching, deleting, or inserting a node.



Unbalanced BST

So, to optimize operations on a BST, the height must be minimized, and to do that the tree must be balanced.

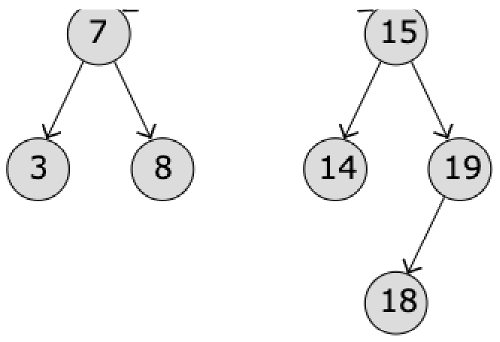
And keeping a Binary Search Tree balanced is exactly what AVL Trees do, which is the data structure explained on the next page.

DSA Exercises

Test Yourself With Exercises

Exercise:

Inserting a node with value 6 in this Binary Search Tree:



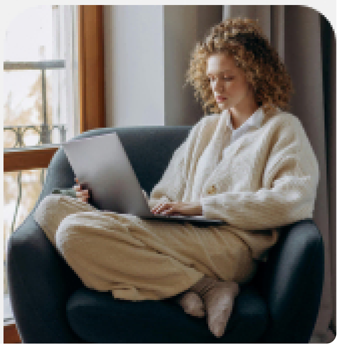
Where is the new node inserted?

```
The node with value 6
becomes the right child node
of the node with value     .
```

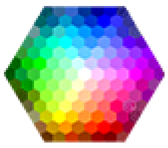
Submit Answer »

Start the Exercise

❮ Previous Sign in to track progress Next ❯



COLOR PICKER



PHP Tutorial
Java Tutorial
C++ Tutorial
jQuery Tutorial

Top References

HTML Reference
CSS Reference
JavaScript Reference
SQL Reference
Python Reference
W3.CSS Reference
Bootstrap Reference
PHP Reference
HTML Colors
Java Reference
AngularJS Reference
jQuery Reference

Top Examples

HTML Examples
CSS Examples
JavaScript Examples
How To Examples
SQL Examples
Python Examples
W3.CSS Examples
Bootstrap Examples
PHP Examples
Java Examples
XML Examples
jQuery Examples

Get Certified

HTML Certificate
CSS Certificate
JavaScript Certificate
Front End Certificate
SQL Certificate
Python Certificate
PHP Certificate
jQuery Certificate
Java Certificate
C++ Certificate
C# Certificate
XML Certificate

FORUM ABOUT ACADEMY