A Practical Introduction to **Data Structures** and **Algorithms** in JavaScript

Algorithms

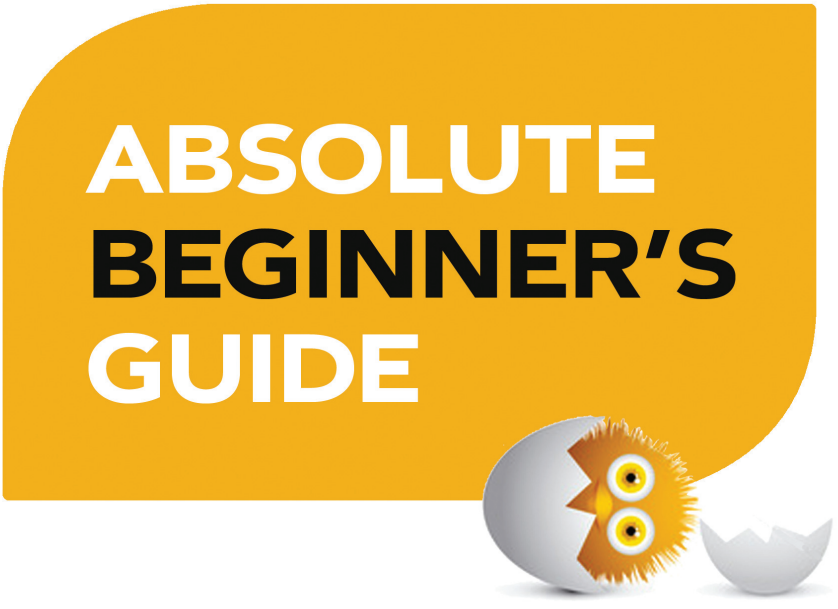
ABSOLUTE BEGINNER'S GUIDE

No experience necessary!

Kirupa Chinnathambi

Absolute Beginner's Guide to Algorithms

A Practical Introduction to Data Structures and Algorithms in JavaScript



Kirupa Chinnathambi

Pearson

```
        } else {
          // Case 3: Node has two children
          // Find the inorder successor of the node to remove
          let successor = currentNode.right;
          let successorParent = currentNode;

          while (successor.left !== null) {
            successorParent = successor;
            successor = successor.left;
          }

          // Replace the node to remove with the inorder successor
          if (successorParent.left === successor) {
            successorParent.left = successor.right;
          } else {
            successorParent.right = successor.right;
          }

          currentNode.data = successor.data;
          return true;
        }
      } else if (value < currentNode.data) {
        // If the value we're looking for is less than
        // the current node's value, go left
        parentNode = currentNode;
        currentNode = currentNode.left;
      } else {
        // If the value we're looking for is greater than
        // the current node's value, go right
        parentNode = currentNode;
        currentNode = currentNode.right;
      }
    }

    // If we reach this point, the value was not found in the tree
    return false;
  }
}
```

Take a brief glance through the preceding lines of code. The comments call out important landmarks, especially as they relate to the binary search tree behavior we have been looking at. To see this code in action, here is an example:

```
let myBST = new BinarySearchTree();

myBST.insert(10);
myBST.insert(5);
myBST.insert(15);
myBST.insert(3);
myBST.insert(7);
myBST.insert(13);
myBST.insert(18);
myBST.insert(20);
myBST.insert(12);
myBST.insert(14);
myBST.insert(19);
myBST.insert(30);
```

We are creating a new binary search tree and adding some nodes to it. This tree will look like Figure 9-21.

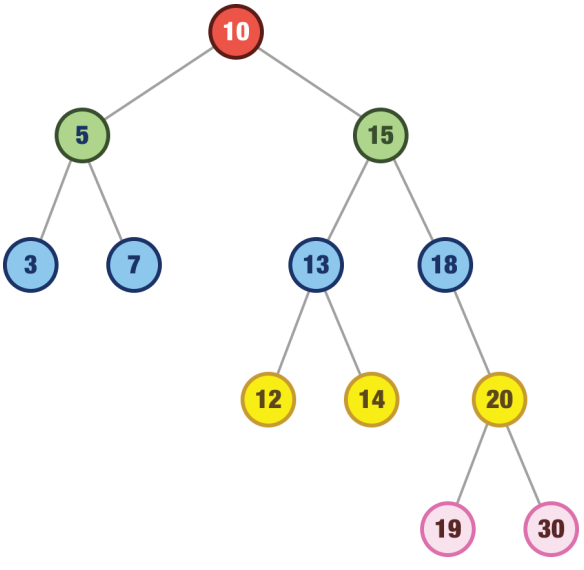


FIGURE 9-21

Our new binary tree

Let's say that we want to remove the 15 node:

```
myBST.remove(15);
```

Our tree will rearrange itself to look like Figure 9-22.

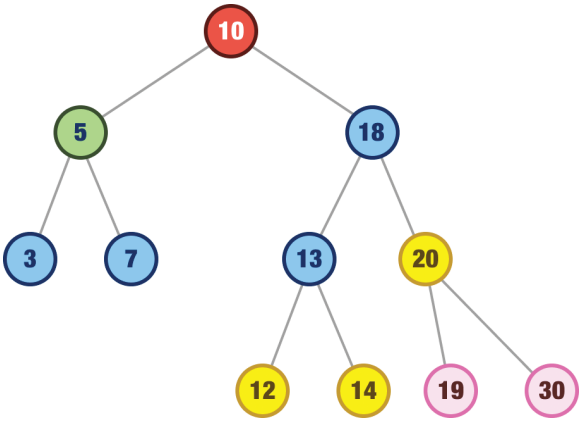


FIGURE 9-22

Our binary tree after removing the 15 node

The 15 node is gone, but the 18 node takes its place as the rightful inorder successor. Feel free to play with more node additions and removals to see how things will look. To easily see how all of the nodes are related to each other, the easiest way is to inspect your binary search tree in the Console and expand each left and right node until you have a good idea of how things shape up (Figure 9-23).

If you want to go above and beyond, you can create a method that will print an ASCII-art representation of a tree in our console, so do let me know if you have already done something like that.

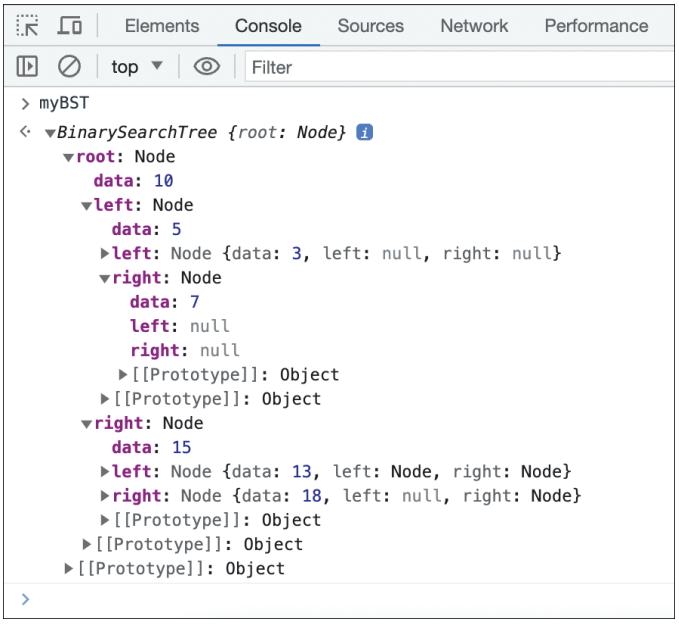


FIGURE 9-23

Output of running our code

Performance and Memory Characteristics

The performance of our binary search tree is related to how balanced or unbalanced the tree is. In a perfectly balanced tree, the common operations like searching, inserting, and deleting nodes will take **O(log *n*)** time (Figure 9-24).

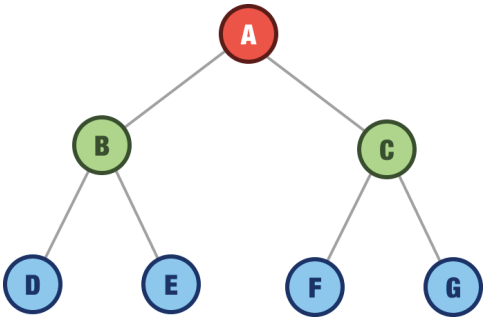


FIGURE 9-24

A perfectly balanced tree

This is because we can avoid taking very uniquely long paths to find any single node. The worst-case scenario is when our tree is heavily unbalanced (Figure 9-25).

In this tree, if our node happens to be deep in the right subtree, we'll be exploring a lot of nodes relative to the total number of nodes in a tree. This gets us closer to a running time of O(*n*), which is the worst-case scenario.

As for the amount of memory a binary search tree takes up, that doesn't depend on how balanced or unbalanced our tree is. It is always O(*n*) where each node takes up a fixed amount of memory.

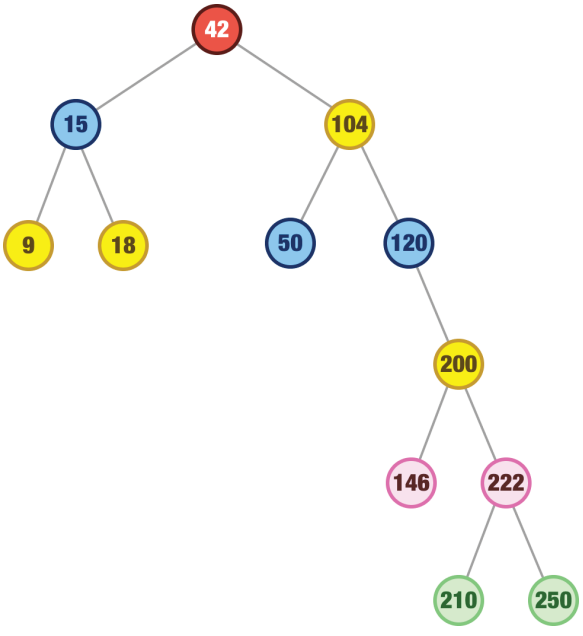


FIGURE 9-25

Our final binary search tree

Conclusion

Binary search trees are pretty sweet. They are a type of binary tree with some added constraints to make them more suited for heavy-duty data wrangling. The constraints are to ensure the left child is always smaller than the parent and the right child is always greater. There are a few more rules around how nodes should arrange and rearrange themselves when they get added or removed.

This type of structure allows us to efficiently perform search, insert, and delete operations in O(log *n*) time complexity, making binary search trees a popular data structure. However, as we saw a few moments ago, the performance of a binary search tree can be impacted by its balancedness. For heavily unbalanced trees, this can lead to worst-case scenarios with the time complexity of O(*n*).

SOME ADDITIONAL RESOURCES

- ? Ask a question: **https://forum.kirupa.com**
- Errors/Known issues: **https://bit.ly/algorithms_errata**
- Source repository: **https://bit.ly/algorithms_source**