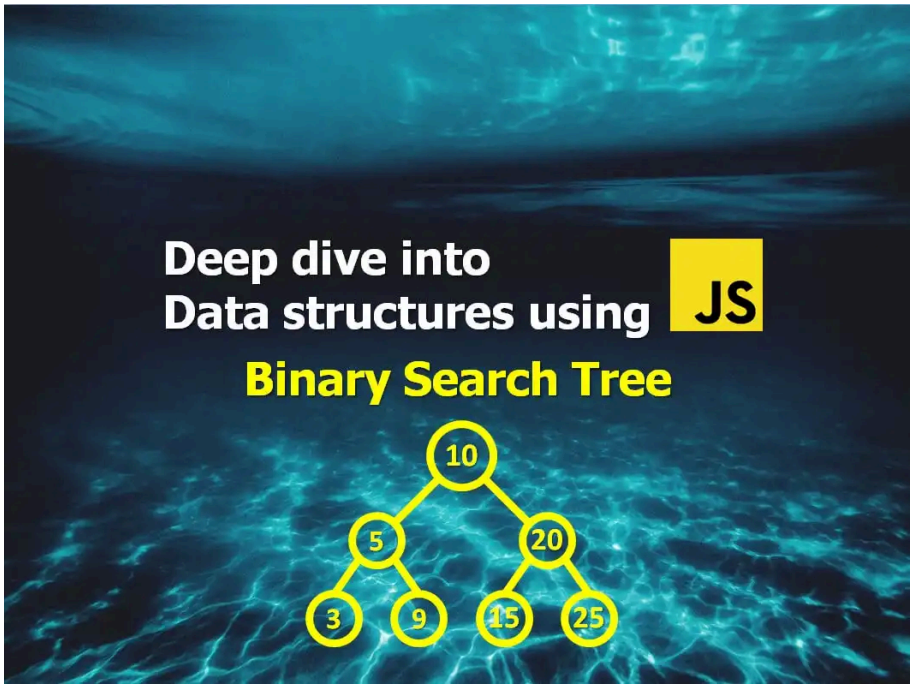
Deep Dive into Data structures using Javascript - Binary Search Tree



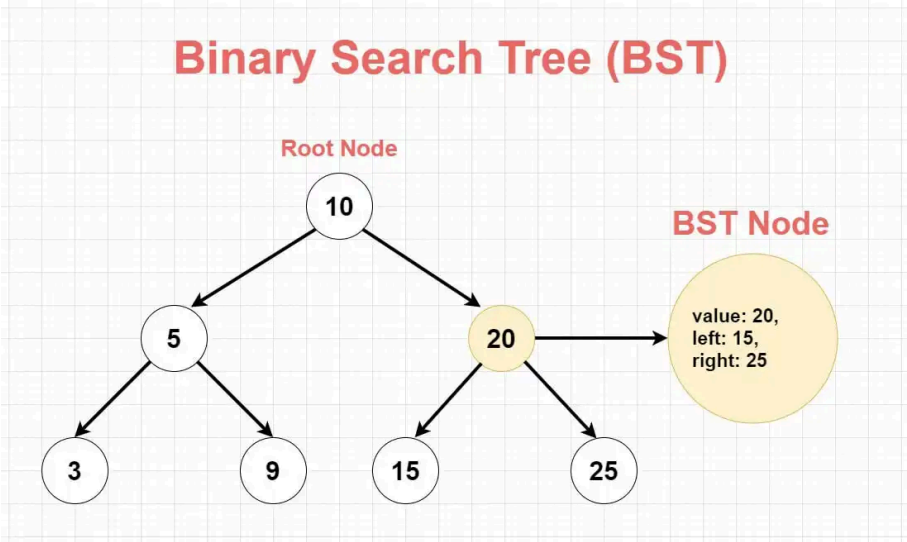
What is a Binary Search Tree?

Binary Search Tree (BST as short) is a variation of Binary tree data structure with specific restrictions that allows efficient insertion, deletion and lookup of items. Being a descendant of Binary Tree, it comes with each node having at most two children. Specific restriction is the left child node has a value less than its parent node, while the right child node has a value greater than its parent node.

The tone of this article is assuming you are at least familiar with the concept of Tree data structure and Binary Tree. If that's not the case or you need a quick refreshment, I'd suggest you start from the "Introduction to Trees" article by following the link below, then come back and continue here later:

Deep dive into data structures using Javascript - Introduction to Trees

Anatomy of a Binary Search Tree



A Binary Search Tree consists of connected nodes that resemble a hierarchy, where each node has at most two children. The tree starts with a root node, and every node in the tree descends from the root node. There can be only one root in the tree.

Each node in the BST consists of 3 properties:

value: Holds the value / data for the node.

left: Holds a reference (pointer) to the left child node.

right: Holds a reference (pointer) to the right child node.

As mentioned earlier, unique part of Binary Search Tree is the left child node is always having less and the right child node is always having greater value than their parent node. Which also means whenever you add or remove elements, this is something you have to take into account to keep the order of BST staying relevant overall the tree - because this rule applies to every single node inside the BST.

When to use Binary Search Tree

Binary search trees (BSTs) are useful in situations where fast search, insertion, and deletion operations are needed. They are particularly useful when the elements being stored are already sorted or when the elements are constantly changing.

Some real-world examples are:

- Storing and quickly searching through large amounts of data, such as a database index or financial data like stock prices.
- Dictionary or spell checker, where words are stored in a BST and can be quickly searched for and inserted.
- Autocomplete feature, where suggestions are stored in a BST and can be quickly searched for as the user types.
- Data compression, most frequently occurring elements in the data are represented by the higher levels of the tree and the less frequently occurring elements are represented by the lower levels. This allows for more efficient storage of the data, as the elements that appear more frequently in the data take up less space in the tree.

In general, Binary Search Trees are best suited when the data is static and we want to achieve a good balance between search and insert/delete operations.

What does make Binary Search Tree efficient? Let's see it by taking a look at the Big O of common operations in Binary Search Tree:

Big O Notation for Binary Search Trees

Balanced Binary Search Tree

Method	Worst case
insert()	O(log n)
delete()	O(log n)
search()	O(log n)
Breadth First / Depth First traversals	O(n)

Unbalanced Binary Search Tree

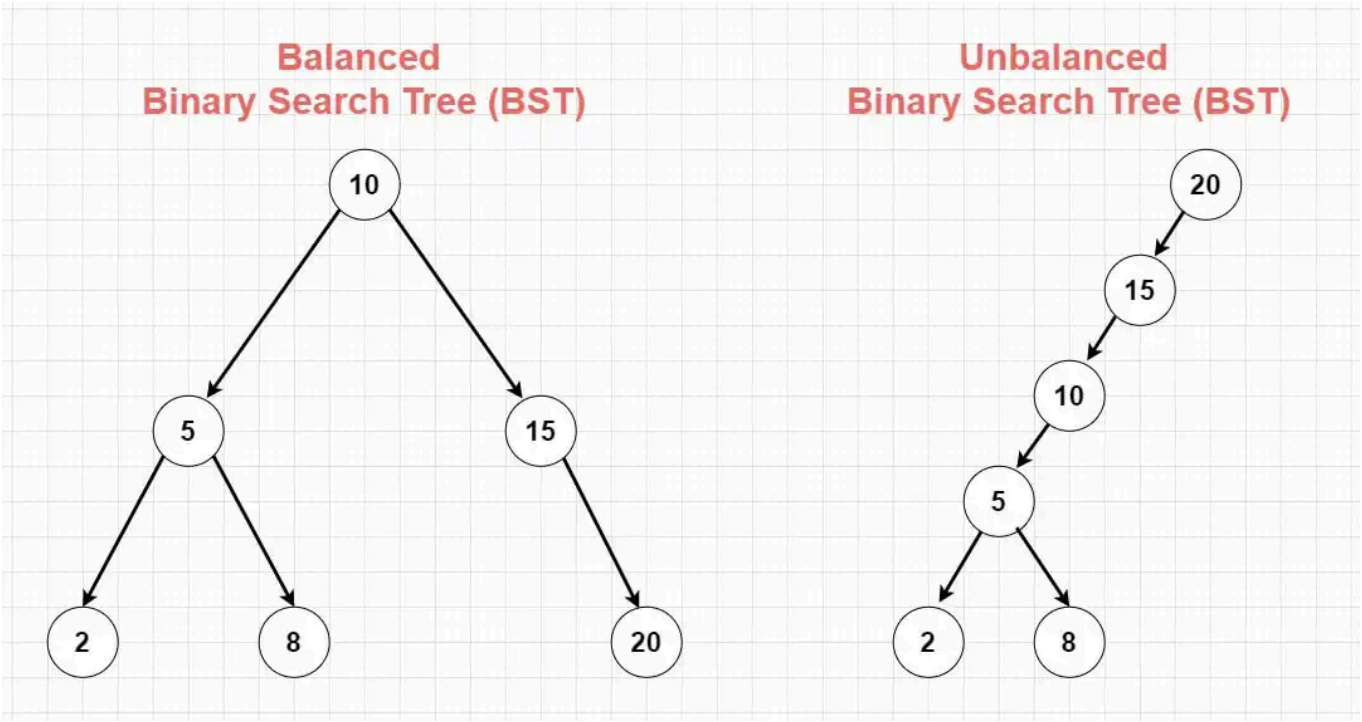
Method	Worst case
insert()	O(n)
delete()	O(n)
search()	O(n)
Breadth First / Depth First traversals	O(n)

As you see, we have 2 different time complexity tables above. Let's start with what do we mean by "Balanced" and "Unbalanced" before we move onto the use cases:

Balanced vs Unbalanced Binary Search Tree

In simple terms, Balanced Binary Search Tree is a tree where the height of left and right subtrees is almost same, and Unbalanced Binary Search Tree is a tree where the height of left and right subtrees is not same, which results in poor performance when searching, inserting, or deleting nodes.

If you take a look at the visual below, you'll see an Unbalanced Binary Search Tree starts to look very similar to a Linked List. And that's the reason why the time complexities on an Unbalanced version is O(n) while the Balanced version have the O(log n):



When does a Binary Search Tree becomes unbalanced? If the items are inserted in an order that creates this skewed / unbalanced order. For example, adding items in an ascending or descending order is a common reason. There are a couple ways to avoid this:

- Using self balanced BST variations like AVL or Red-Black Tree.
- Sort the elements in a balanced order before inserting into the BST.
- Rebalance the BST periodically using a method.

Although AVL or Red-Black Tree implementations of BST adds more complexity to the code, but they provide guaranteed logarithmic O(log n) time complexity for the operations. But for now, we will only focus on the foundational building blocks in this article.

Binary Search Tree implementation in Javascript

We will be using ES6 Classes to build our Binary Search Tree. In general, implementing non-linear data structures are quite complex - but no worries: we will first go through a couple of steps in detail until we understand the key concepts.

Build a class for the Tree Node

The main building block of a BST is the Node element. We can use a class for it, so we can reuse it whenever we need to create a new Node:

```
class Node {
  constructor(value) {
    this.value = value
    this.left = null
    this.right = null
  }
}
```

The Node class has a constructor that takes in a value as a parameter, and initializes the left and right properties to null. We can use these pointers later on to manage referencing other nodes or nullifying them whenever it is needed.

Build a class for the Binary Search Tree

Now that we have our Node class ready, we can build a class for our Binary Search Tree. We know there should be a “root” property in the class as a reference to the root node, which we can place in the constructor:

```
class BinarySearchTree {
  constructor() {
    this.root = null
  }
}
```

Understanding Iterative vs Recursive Approaches in Binary Search Tree methods

Before we start with method implementations, I want to point out the importance of understanding the difference between iterative and recursive approaches when working with Binary Search Trees. Both approaches have their own benefits and trade-offs, and it’s beneficial to understand which one to use in different scenarios.

Iterative Approach

An iterative approach uses a loop to traverse through the tree and perform operations. It is generally more efficient in terms of memory usage as it does not require the use of a call stack (as it avoids the overhead of the function call stack). However, it can be more difficult to read and understand for some people and can make the code less readable.

Recursive Approach

On the other hand, a recursive approach uses function calls to traverse through the tree. This approach is generally simpler to implement, as it follows the natural recursion of the tree structure (side note: Trees are sometimes also referred as recursive data structures). However, it

can lead to stack overflow errors when dealing with large trees, and can be less efficient in terms of memory usage.

When to use Iterative or Recursive Approach

When working with Binary Search Trees, it's important to consider the size of the tree and the complexity of the operations you'll be performing. For small trees or simple operations, the recursive approach may be more appropriate. However, for large trees or complex operations, the iterative approach may be more efficient and less prone to errors. It's worth noting that JavaScript has a call stack size limit depending on the environment, which means that a recursive function can only be called until it hits the limit before a stack overflow error occurs. Therefore, it is important to keep in mind the size of the tree and the complexity of the operation when deciding which approach to use.

In this article, I will be providing implementation examples for both iterative and recursive approaches for the most common Binary Search Tree operations such as insert, search and delete. This will help you to understand the pros and cons of each approach, and help you make informed decisions when working with BSTs in your own projects.

Binary Search Tree methods

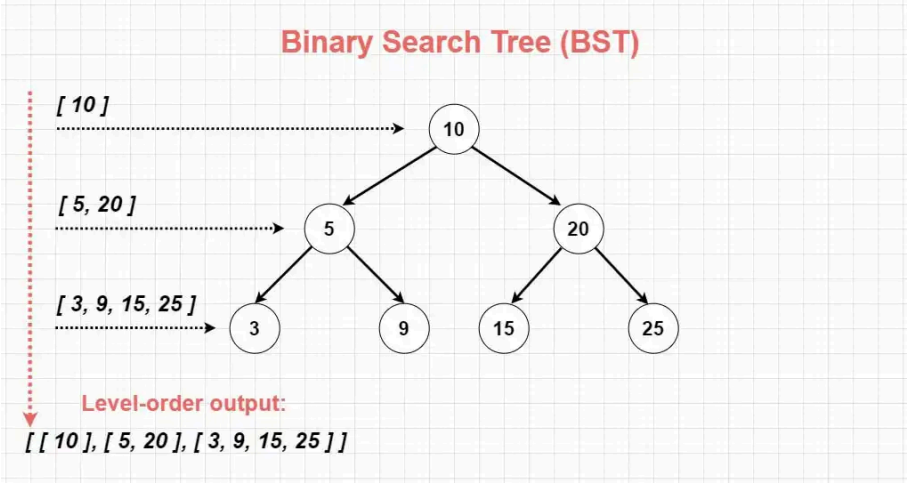
Now that we have the base building blocks and also understanding of the differences between Iterative and Recursive approaches, we can extend the BinarySearchTree class by introducing methods. Here is the list of methods we are going to implement:

- `insert(value, isRecursive(optional))` - *insert a new value into the BST. "isRecursive" parameter is optional, and if set to true, the method will use the "insertRecursive" helper method to insert the value into the tree. If not provided or set to false, the "insertIterative" helper method will be used.*
- `_insertIterative(value)` - *helper method that is used to insert a new value into the BST iteratively.*
- `_insertRecursive(value)` - *helper method that is used to insert a new value into the BST recursively.*

- `find(value, isRecursive(optional))` - *find an element that matches the given value in the BST. "isRecursive" parameter is optional, and if set to true, the method will use the "findRecursive" helper method to find the value. If not provided or set to false, the "findIterative" helper method will be used.*
- `_findIterative(value)` - *helper method that is used to find an element that matches the given value in the BST iteratively.*
- `_findRecursive(value)` - *helper method that is used to find an element that matches the given value in the BST recursively.*
- `delete(value, isRecursive(optional))` - *remove a node that matches the given value from the BST. "isRecursive" parameter is optional, and if set to true, the method will use the "deleteRecursive" helper method to remove the node. If not provided or set to false, the "deleteIterative" helper method will be used.*
- `_deleteIterative(value)` - *helper method that is used to remove a node that matches the given value from the BST iteratively.*
- `_deleteRecursive(value)` - *helper method that is used to remove a node that matches the given value from the BST recursively.*
- `showAsLevelOrdered` - *display the BST as a level-order multi-dimensional array, uses "level order traversal" under the hood. The final output array will represent the tree in level-order, with each sub-array representing a level of the tree.*

Now you might be asking, what does "Level order traversal" means that I mentioned in `showAsLevelOrdered` description? It is a subject related to Tree Traversals which needs it's own article. To keep things simple I won't be diving deep into Tree traversal details here - but rather give a quick overview about it.

Traversals are a way to traverse through all the nodes in a tree data structure, visiting each node in a specific order. "Level order traversal" is one of them, which is also known as "Breadth-first traversal". With level order, nodes are visited level by level, starting from the root. This type of traversal is useful for displaying the tree in a readable format, as it shows the tree's structure clearly. Here is a rough sketch of how the output looks like:



One thing I'd like to mention is, it's completely normal to not fully understand Binary Search Trees at first glance. It's quite common whenever you are dealing with any non-linear data structure. The rule of thumb is, if you give enough time it will all make sense!

I hope this article helped you to understand what Binary Search Tree is and how it works! I'd like to encourage you to experiment with the implementation below in your favorite code editor. When you are experimenting with Binary Search Tree, I highly recommend using the Visualgo on the side. It has a great visualition that can show you how everything works visually.

Here is the link to Binary Search Tree visualization on Visualgo: https://visualgo.net/en/bst

Thanks for reading!

Implementation of Binary Search Tree (BST) in Javascript

```
class Node {
  constructor(value) {
    this.value = value
    this.left = null
    this.right = null
  }
}


class BinarySearchTree {
  constructor() {
    // initialize the root
    this.root = null
  }

  // handles "insertion" with both variations: iterative and recursive
  // "isRecursive" is an optional parameter, default handler is iterative
  insert(value, isRecursive) {
    if (!isRecursive) return this._insertIterative(value)
    return this._insertRecursive(value)
  }

  // handles "find" with both variations: iterative and recursive
  // "isRecursive" is an optional parameter, default handler is iterative
  find(value, isRecursive) {
    if (!isRecursive) return this._findIterative(value)
    return this._findRecursive(value)
  }
```

```
// handles "delete" with both variations: iterative and recursive
// "isRecursive" is an optional parameter, default handler is iterative
delete(value, isRecursive) {
  if (!isRecursive) return this._deleteIterative(value)
  return this._deleteRecursive(value)
}

// Iterative insertion
_insertIterative(value) {
  // Create a new node with the value to be inserted
  const newNode = new Node(value)

  // If the tree is empty, make the new node as the root and early exit
  if (this.root === null) {
    return this.root = newNode
  }

  // Iterative insert method to be called if the root is not empty:
  const iterativeInsert = (currentNode, value) => {
    while (true) {
      // If the value already exists in the tree,
      // throw an error or silently fail:
      if (value === currentNode.value) {
        // OPTIONAL: replace the below line with return
        // if you want it to throw an error on duplicates
        // throw new Error("Duplicate value not allowed")
        return
      }




      // Check if the input value is less than the current node's value
      if (value < currentNode.value) {
        // If the left child is null, insert the new node as the left child
        if (currentNode.left === null) {
          return currentNode.left = newNode
        }
        // If the left child is not null, move the current node to the left child
        currentNode = currentNode.left
      } else {
          // Check if the input value is greater than
          // or equal to the current node's value
          if (currentNode.right === null) {
            // If the right child is null, insert the new node as the right child
            return currentNode.right = newNode
          } else {
            // If the right child is not null, move the current node to the right child
            currentNode = currentNode.right
          }
      }
    }
  }

  // Call the inner method with root node
  // as the starting point, and value as target:
  iterativeInsert(this.root, value)
}

// Recursive insertion
_insertRecursive(value) {
```

```
  // Create a new node with the value to be inserted
  const newNode = new Node(value)

  // if the tree is empty, insert the new node as the root and early exit
  if (this.root === null) {
    return this.root = newNode
  }

  // Recursive insert method to be called if the root is not empty:
  const recursiveInsert = (currentNode, value) => {
      // If the value already exists in the tree, throw an error or silently fail:
      if (value === currentNode.value) {
        // OPTIONAL: replace the below line with return
        // if you want it to throw an error on duplicates
        // throw new Error("Duplicate value not allowed")
        return
      }

      // if the value is greater than the current node's value, go right
      if (value > currentNode.value) {
          // if there is no right child, insert the new node here
          if (currentNode.right === null) {
            currentNode.right = newNode
            return
          } else {
              // otherwise, recursively call insert on the right child
              recursiveInsert(currentNode.right, value)
          }
      } else {




          // if the value is less than the current node's value, go left
          if (currentNode.left === null) {
              currentNode.left = newNode
              return
          } else {
              // otherwise, recursively call insert on the left child
              recursiveInsert(currentNode.left, value)
          }
      }
  }

  // Call the inner method with root node
  // as the starting point, and value as target:
  recursiveInsert(this.root, value)
}

// Iterative search
_findIterative(value, currentNode = this.root) {
  // Start at the root of the tree (or a passed in node)
  while (currentNode) {
    // Check if the current node's value is equal to the search value
    if (currentNode.value === value) {
      return currentNode.value; // If found, return the value
    }
    // If the search value is less than the current node's value,
    // move to the left child
    if (value < currentNode.value) {
      currentNode = currentNode.left;
    } else {
```

```
      // If the search value is greater than the current node's value,
      // move to the right child
      currentNode = currentNode.right;
    }
  }
  // If the value is not found, return null
  return console.log('Given value not found in the tree:', value)
}

// Recursive search
_findRecursive(value, currentNode = this.root) {
  // base case: if current node is null, value is not in the tree
  if (currentNode === null) {
    return console.log('Given value not found in the tree:', value)
  }

  // if current node value is equal to the given value, return value
  if (currentNode.value === value) {
    return currentNode.value
  }

  // if given value is less than current node value, search left subtree
  if (value < currentNode.value) {
    return this._findRecursive(value, currentNode.left)
  }

  // if given value is greater than current node value, search right subtree
  if (value > currentNode.value) {
    return this._findRecursive(value, currentNode.right)




  }
}

// Iterative delete
_deleteIterative(value) {
  // Start at the root
  let currentNode = this.root
  // At this point parentNode is null, since root has no parent
  let parentNode = null

  // STEP 1 - Find the targetNode and it's parentNode to be deleted:
  while (currentNode !== null && currentNode.value !== value) {
    /*
    Here, the currentNode is set as the parentNode
    before moving to the left or right child node
    depending on the value being searched for.
    This is so that the parentNode variable always
    holds the parent of the currentNode.

    When the loop exits, the currentNode variable holds the target node
    that needs to be deleted and "parentNode" variable holds its parent.
    */
    parentNode = currentNode

    // if the value is less than the current node's value, move to the left child
    if (value < currentNode.value) {
      currentNode = currentNode.left
    // if the value is greater than the current node's value, move to the right child
    } else {
```

```
    currentNode = currentNode.right
  }
}

// STEP 2 - If the currentNode is null, the value was not found in the tree
if (currentNode === null) {
  console.log('Given value not found in the tree:', value)
  return
}

// STEP 3 - Handle the 3 deletion cases:
// leaf node (node with 0 children),
// node with 1 children, node with 2 children

// CASE 1: LEAF NODE
// If the current node has no children, it is a leaf node.
if (currentNode.left === null && currentNode.right === null) {
  /*
  In this case, if the parent node is null
  (which means the current node is the root),
  the root is set to null.
  */
  if (parentNode === null) {
    this.root = null
  } else {
  /*
  If the parent node is not null, the left or right reference
  of the parent node is set to null, depending on
  if the current node is the left or right child of the parent node.




  */
    if (parentNode.left === currentNode) {
      parentNode.left = null
    } else {
      parentNode.right = null
    }
  }
}

// CASE 2: NODE WITH SINGLE CHILD
// If the current node has 1 children at the left or right side

// ONLY HAS CHILDREN AT RIGHT SIDE, LEFT IS NULL
else if (currentNode.left === null) {
  /*
  In this case, if the parent node is null
  (which means the current node is the root),
  the root is set to the current node's right child.
  */
  if (parentNode === null) {
    this.root = currentNode.right
  } else {
    /*
    If the parent node is not null, the left or right
    reference of the parent node is set to
    the current node's right child, depending on
    if the current node is the left or right child of the parent node.
    */
    if (parentNode.left === currentNode) {
```

```
      parentNode.left = currentNode.right
    } else {
      parentNode.right = currentNode.right
    }
  }
// ONLY HAS CHILDREN AT LEFT SIDE, RIGHT IS NULL
  /*
  Same as the above block of code, but the children at the opposite side
  */
} else if (currentNode.right === null) {
  if (parentNode === null) {
    this.root = currentNode.left
  } else {
    if (parentNode.left === currentNode) {
      parentNode.left = currentNode.left
    } else {
      parentNode.right = currentNode.left
    }
  }
}

// CASE 3: NODE WITH TWO CHILDREN
// If the current node has 2 children (left & right)
else {
  /*
  If the current node has two children,
  we need to find the in-order successor of the current node.

  You can go 2 ways to find the inorder successor: either smallest value




  in right subtree, or largest element in left subtree. In this method
  we will use the first: finding the smallest value in right subtree.

  The in-order successor is the node with the smallest value
  that is greater than the current node's value.

  We start by initializing the inorderSuccessor variable to the
  current node's right child and inorderSuccessorParent to the current node.

  Then, we iterate through the right child's left subtree
  until we find the node with the smallest value.
  */
  let inorderSuccessor = currentNode.right
  let inorderSuccessorParent = currentNode

  while (inorderSuccessor.left !== null) {
    inorderSuccessorParent = inorderSuccessor
    inorderSuccessor = inorderSuccessor.left
  }

  currentNode.value = inorderSuccessor.value
  /*
  Once we find the in-order successor,
  we replace the current node's value with the in-order successor's value.

  Then, we remove the in-order successor from the tree
  by setting its parent's left or right pointer to its right child.
  (since the in-order successor is guaranteed to have
  no left children, we can simply remove it by linking
```

```
    its parent to its right child).

    Finally, the method returns and the tree is updated with
    the desired value removed and its in-order successor taking its place.
    */
    if (inorderSuccessorParent === currentNode) {
      inorderSuccessorParent.right = inorderSuccessor.right
    } else {
      inorderSuccessorParent.left = inorderSuccessor.right
    }
  }
}

// Recursive delete
_deleteRecursive(value) {
  // Recursive delete method to be called
  const recursiveDelete = (currentNode, value) => {
    // Base case: if the node is null, the value is not in the tree
    if (currentNode === null) {
      console.log('Given value not found in the tree:', value)
      return
    }
    // if the value is less than the node's value,
    // then it lies in left subtree
    if (value < currentNode.value) {
      currentNode.left = recursiveDelete(currentNode.left, value)
    }
    // if the value is more than the node's value,
    // then it lies in right subtree




    else if (value > currentNode.value) {
      currentNode.right = recursiveDelete(currentNode.right, value)
    }
    // if value is same as node's value, then this is the node
    // to be deleted
    else {
      // node with only one child or no child
      if (currentNode.left === null) {
        return currentNode.right
      } else if (currentNode.right === null) {
        return currentNode.left
      }

      // node with two children
      // get the inorder successor (smallest in the right subtree)
      let inorderSuccessor = currentNode.right
      while (inorderSuccessor.left !== null) {
        inorderSuccessor = inorderSuccessor.left
      }
      // copy the inorder successor's content to this node
      currentNode.value = inorderSuccessor.value

      // delete the inorder successor
      currentNode.right = recursiveDelete(currentNode.right, inorderSuccessor.value)
    }
    return currentNode
  }

  /*
```

```
    And as usual, we always start the traversal from
    the top of tree "this.root". recursiveDelete helper method
    will return the complete updated tree after removal,
    so we simply point the root to the updated tree:
    */
    this.root = recursiveDelete(this.root, value)
  }

  // Displays an array that will represent the tree
  // in level-order, with each sub-array representing a level of the tree.
  showAsLevelOrdered() {
    // Create an empty array to store the traversed nodes
    const temp = []
    // Create an array to keep track of the current level of nodes
    const queue = []

    // If the tree has a root, add it to the queue
    if (this.root) {
      queue.push(this.root)
    }

    // Keep traversing the tree while there are nodes in the queue
    while (queue.length) {
      // Create an array to store the nodes of the current level
      const subTemp = []
      // Store the number of nodes in the current level
      const len = queue.length

      // Iterate through the current level of nodes




      for (let i = 0; i < len; i += 1) {
        // Dequeue the first node in the queue
        const node = queue.shift()
        // Push the node's value to the subTemp array
        subTemp.push(node.value)
        // If the node has a left child, add it to the queue
        if (node.left) {
          queue.push(node.left)
        }
        // If the node has a right child, add it to the queue
        if (node.right) {
          queue.push(node.right)
        }
      }

      // Push the subTemp array to the temp array
      temp.push(subTemp)
    }
    // Return the final temp array
    return temp
  }
}
```