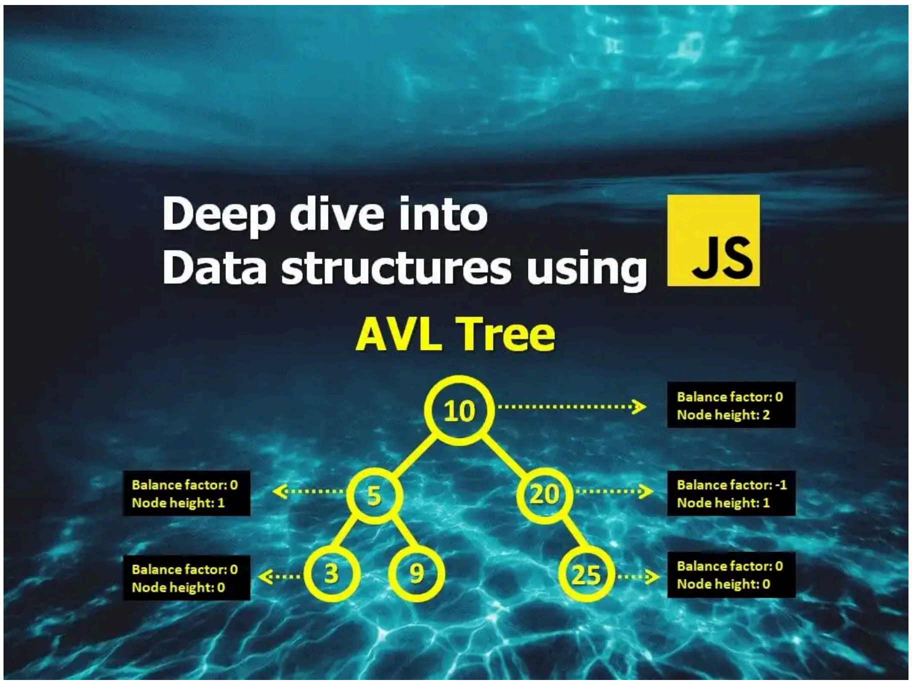
Deep Dive into Data structures using Javascript - AVL Tree



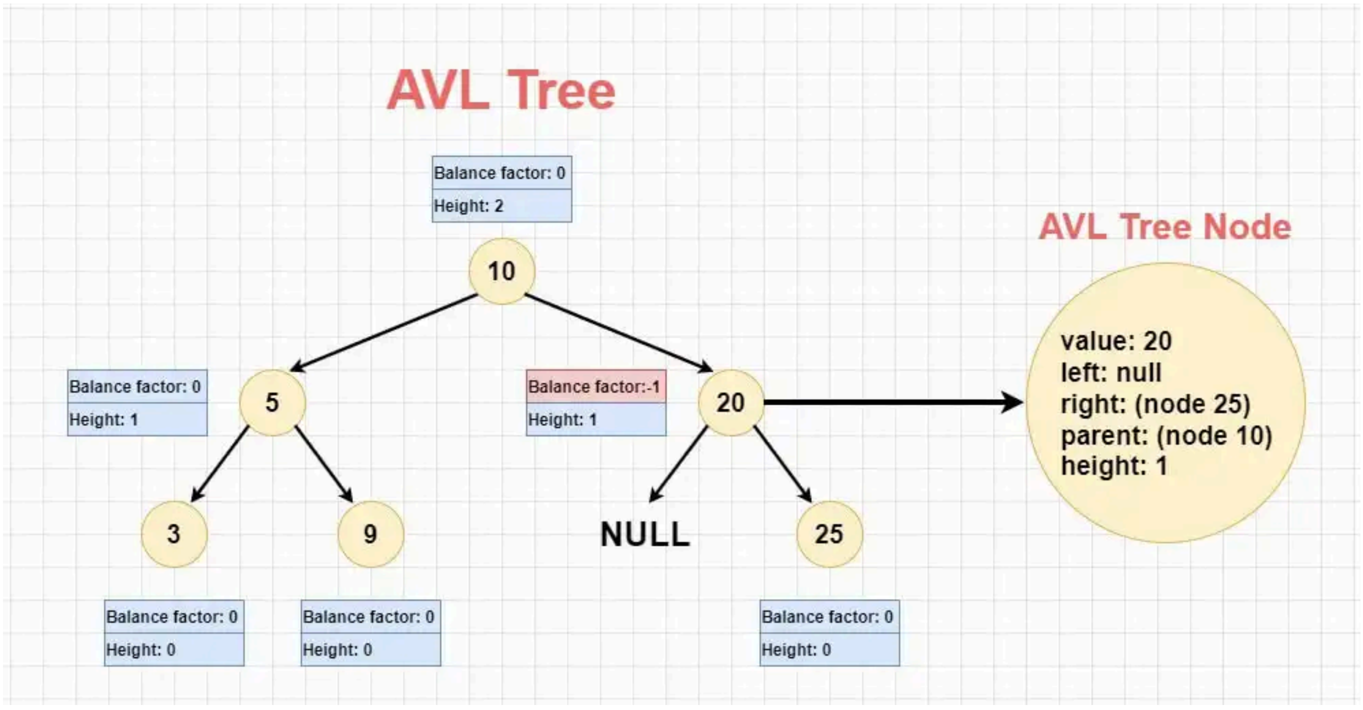
AVL Tree, also known as height-balanced Binary Search Tree (BST) is a genius invention by Adelson-Velsky and Landis, hence the acronym "AVL". Similar to Red-Black Trees, AVL Trees are designed to maintain balance, which guarantees consistent and reliable performance even with frequent modifications to the tree.

Importance of this reliable performance cannot be overstated. Working with large datasets or in situations that demand quick operations, we can't afford the O(n) time complexity that standard BSTs can slip into during their worst-case scenarios. AVL Trees take charge here, offering a balanced tree structure that helps to maintain speed and steady performance for all crucial tasks.

The tone of this article is assuming you are already familiar with Binary Search Trees. If not, I'd recommend you to start from the "Binary Search Tree" article by following the link below, then come back and continue here later:

Deep Dive into Data structures using Javascript - Binary Search Tree

Anatomy of an AVL Tree



AVL Tree is a variant of Binary Search Tree (BST), but with additional properties to maintain balance. Like BSTs, AVL Trees have nodes with a maximum of two children, where the left child has a smaller value and the right child has a greater value than the parent node.

The key feature of AVL Tree is the ability to remain balanced after every operation, using the height property as a balance factor. This balance is crucial as it guarantees efficient operations even with large data sets or time-sensitive applications.

Each node in an AVL Tree has five properties:

- **value:** The data or value stored in the node.
- **left:** A reference to the left child node.
- **right:** A reference to the right child node.
- **parent:** A reference to the parent node.
- **height:** The height of the node within the tree structure.

Let’s take a closer look at the height property and it’s role:

Height

The height of a node in an AVL Tree represents the longest path from that node to a leaf. Leaf nodes have a height of 0. The height property is essential for maintaining balance in the tree and calculating the balance factor.

AVL Tree rules

Maintaining balance in an AVL Tree requires following a set of rules known as AVL Tree properties:

- **Equal Height Paths:** Maintaining equal height paths is crucial in an AVL Tree. This property ensures that all paths from a node to its descendant leaf nodes have a similar height. Performing rotations when it is needed helps maintaining this balance.
- **Tree Rotations:** To correct any imbalances in the AVL Tree, rotations are performed. The rotations address nodes with balance factors outside of -1, 0, or 1. There are four types of rotations: left, right, left-right, and right-left, used based on the specific imbalance detected.

- **Balance Factor:** Each node in an AVL Tree has a balance factor calculated by finding the difference in height between its left and right children. A balanced AVL Tree ensures that every node's balance factor is -1, 0, or 1. If a node's balance factor goes beyond this range, the tree becomes unbalanced and needs adjustments through rotations.
- **Leaf Nodes (Null Nodes):** Leaf nodes, or null nodes, are considered to have a height of -1. This convention aids in balance factor calculation, helping to maintain the tree's balance during operations.

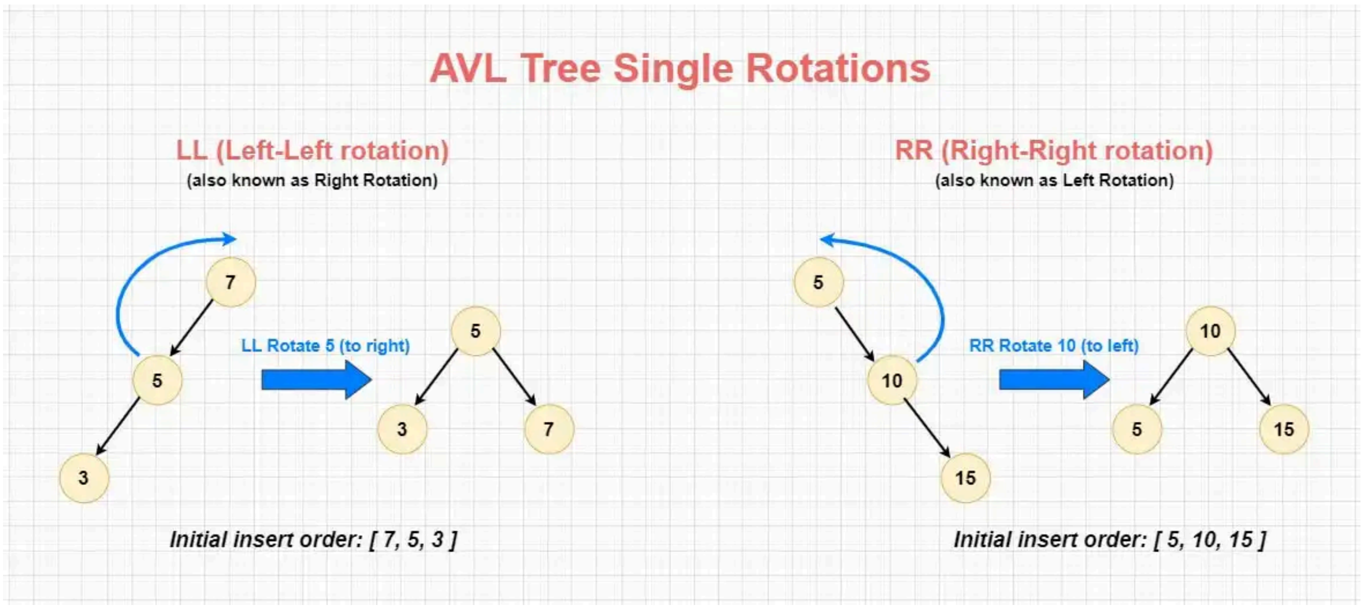
All these rules might sound confusing at the moment, let's take a look at some basic examples of how all these rotations to get some sense of it.

AVL Tree rotations in detail

AVL tree rotations are fundamental operations used to maintain the AVL tree's balance after insertions or deletions. The balance of an AVL tree is defined by the difference in height between the left and right subtrees of a node. If this difference is more than 1 or less than -1, the tree is considered unbalanced, and rotations are used to restore the balance.

There are 4 types of rotations in total - which we can categorize under Single and Double:

Single Rotations


AVL Tree Single Rotations
LL (Left-Left rotation)
(also known as Right Rotation)
RR (Right-Right rotation)
(also known as Left Rotation)
7
5
3
LL Rotate 5 (to right)
5
3
7
5
10
15
RR Rotate 10 (to left)
10
5
15
Initial insert order: [7, 5, 3]
Initial insert order: [5, 10, 15]

Left-Left (LL) Rotation (also known as Right Rotation):

The Left-Left rotation is used when a node is unbalanced to the left side and the left child of the unbalanced node is itself balanced to the left side or is perfectly balanced. The unbalanced node is rotated to the right to restore balance.

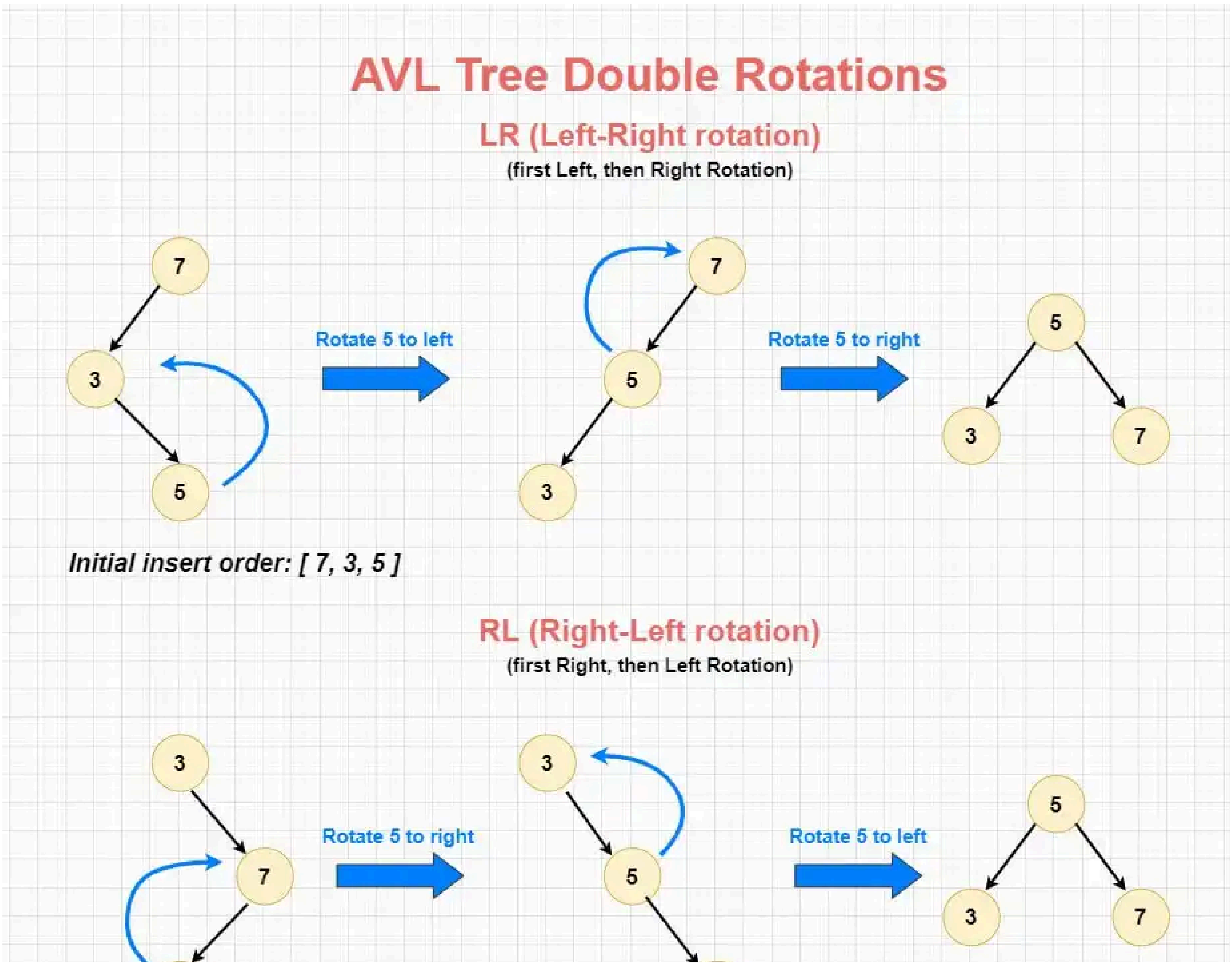
In the Left-Left rotation, the left child of the unbalanced node becomes the new root of the subtree and the original root becomes the right child of the new root. If the new root had a right child before the rotation, it becomes the left child of the original root after the rotation.

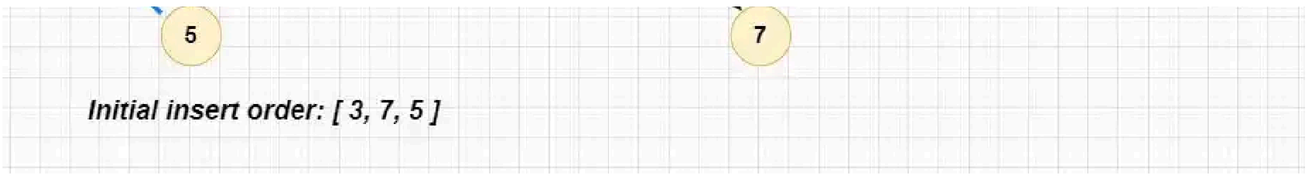
Right-Right (RR) Rotation (also known as Left Rotation):

The Right-Right rotation is the mirror image of the LL rotation. It is used when a node is unbalanced to the right side and the right child of the unbalanced node is itself balanced to the right side or is perfectly balanced.

In the Right-Right rotation, the right child of the unbalanced node becomes the new root of the subtree and the original root becomes the left child of the new root. If the new root had a left child before the rotation, it becomes the right child of the original root after the rotation.

Double Rotations





Left-Right (LR) Rotation (first Left, then Right):

The Left-Right rotation is used when a node is unbalanced to the left side, but the left child of the unbalanced node is balanced to the right side.

The Left-Right rotation is performed in two steps: a RR rotation on the left child of the unbalanced node followed by an LL rotation on the original unbalanced node. This effectively balances the tree by moving the right child of the left child up to the place of the unbalanced node and rearranging the other nodes accordingly.

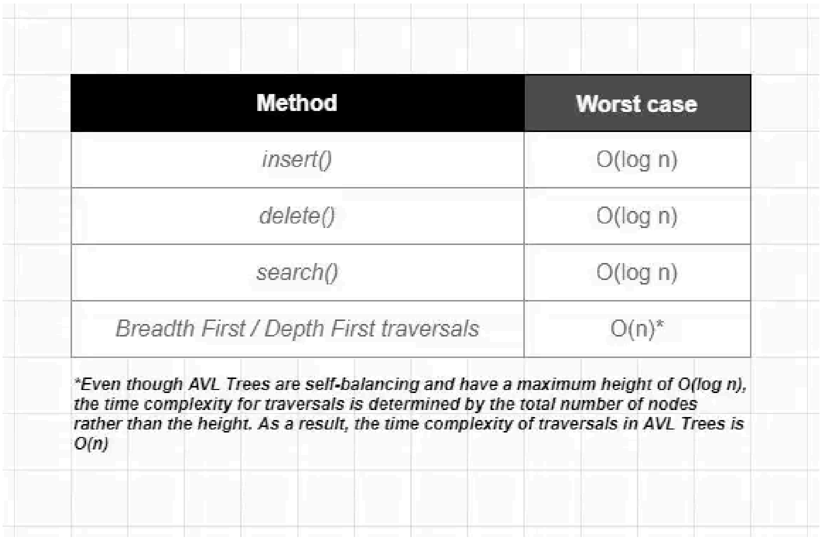
Right-Left (RL) Rotation (first Right, then Left):

The Right-Left rotation is the mirror image of the Left-Right rotation. It is used when a node is unbalanced to the right side and the right child of the unbalanced node is balanced to the left side.

The Right-Left rotation is performed in two steps: an LL rotation on the right child of the unbalanced node followed by an RR rotation on the original unbalanced node. This effectively balances the tree by moving the left child of the right child up to the place of the unbalanced node and rearranging the other nodes accordingly.

When to use AVL Tree

Let's start with taking a look at the Big O of common operations in AVL Tree:



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

**Even though AVL Trees are self-balancing and have a maximum height of O(log n), the time complexity for traversals is determined by the total number of nodes rather than the height. As a result, the time complexity of traversals in AVL Trees is O(n)*

If you find yourself in a situation where you might want to use an AVL Tree, most likely you'll be looking for a self-balancing Binary Search Tree variation. There are various self-balancing trees, but AVL Tree and Red-Black Tree are the most common options.

Even though both of them are self-balancing, one is preferred over the other in certain cases. For more details on the comparison, please take a look at the section **"Red-Black Tree vs AVL Tree"** in the Red-Black Tree article:

Deep Dive into Data structures using Javascript - Red-Black Tree

The basic thumb or rule is:

- Use AVL Tree if you have a lookup intensive scenario.
- Use Red-Black Tree if you have an insertion / deletion intensive scenario.

The reason AVL trees are more efficient for lookup-intensive applications is because they are more strictly balanced (balance logic follows the height instead of colors) and therefore have smaller heights, leading to faster search times.

Some specific scenarios where you might want to use an AVL tree:

Lookup intensive Databases and File Systems: For certain types of databases, particularly those with a high volume of read operations and fewer write operations, AVL trees may be preferable over Red-Black Trees. For example, in a system where you are constantly retrieving user profile information (read operation), but user profile information is updated less frequently (write operation), an AVL tree's superior lookup performance would be beneficial.

Dictionary and Library Management Systems: AVL Trees are ideal for scenarios like a dictionary where the primary operation is searching for a word and its meaning. Similarly, in library management, searching for books based on different parameters happens more often than inserting or deleting books in/from the system.

Small to Medium-Sized Data Sets: AVL trees can be a good choice for scenarios where the data set is relatively small to medium-sized. The self-balancing property of AVL trees ensures efficient search operations, making them suitable when the data set can fit comfortably in memory without causing excessive memory usage.

Note that when dealing with larger data sets or when storage constraints are a concern, other indexing structures like B-trees, which are designed for disk-based or large data sets, may be more appropriate. B-trees optimize disk access patterns and perform well with larger data sizes, making them a preferred choice for scenarios where the data cannot be entirely stored in memory.

AVL Tree implementation in Javascript

We will be using ES6 Classes to build our AVL Tree. Here is the list of methods we are going to implement:

- `insert(value)` - This method is used to insert a new value into the AVL Tree. The method starts by creating a new node and placing it in the correct position, following the standard Binary Search Tree rules. After the node is inserted, the `updateHeight` and `balance` methods are called to restore the AVL Tree properties.
- `delete(value)` - This method removes a node with a given value from the tree. It first finds the node using the search method, then removes it while maintaining the AVL Tree properties. After the node is deleted, the `updateHeight` and `balance` methods are called to restore the AVL Tree properties.
- `search(value)` - This method finds a node in the tree that matches a given value.
- `_balance(node)` - This is a helper method used to balance the tree. Depending on the balance factor of the given node, this method will decide if a rotation is necessary and which type of rotation should be applied.
- `_updateHeight(node)` - This is a helper method used to update the height of a node after insertions or deletions.
- `_getNodeHeight(node) and getBalanceFactor(node)` - These helper methods are used to calculate the height of a node and the balance factor respectively. They're used during insertion, deletion and rotation operations to maintain AVL Tree properties.
- `_leftRotation(node) and _rightRotation(node)`- These helper methods are used to perform left-right and right-left rotations. These rotations are used when the tree is in a state where a simple left or right rotation would not suffice to restore the AVL Tree properties.
- `_leftRightRotation(node) and _rightLeftRotation(node)` - These helper methods are used to perform left-right and right-left rotations. These rotations are used when the tree is in a state where a simple left or right rotation would not suffice to restore the AVL Tree properties.
- `_findMin(node)` - This method finds the node with the smallest value in the tree. It's used during the delete operation to find the successor of a node that's being deleted.

- `levelOrderTraversal()` - Display the AVL Tree as a level-order multi-dimensional array, using the "level order traversal". The final output array will represent the tree with each sub-array representing a level of the tree.

I'd like to mention that understanding AVL Trees for the first time will be a tough challange. They reside at the advanced and complex end of the Data structures spectrum. Fully grasping all the concepts will require investing time into experimentation and exploration. Out of the box self balancing is a powerful magic - but it comes with a cost of implementation complexity.

When studying the insertion and deletion methods, you will come across various conditions and steps combined with AVL Tree-specific operations like rotations. Understanding these operations is crucial as they form the core mechanics of how an AVL Tree preserves its balance during modifications.

Rotations, specifically left and right rotations, play a vital role in rebalancing the tree when its properties are violated due to an insertion or deletion. They rearrange the nodes to restore the AVL Tree's properties. Developing a solid understanding of these rotations will help you better grasp complex insertion and deletion methods.

To enhance your understanding and visualize the mechanics of AVL Trees, I highly recommend playing around with this amazing AVL Tree Visualizer at the link down below:

AVL Tree Visualizer

Additionally, I've also included line-by-line explanations for each method in the implementation for you to follow up what is happening in the code. Hands-on practice along with a visualizer is a good starting point to deepen your understanding.

I hope this article helped you to give a good foundation on what AVL Trees are and how they work! I'd like to encourage you to experiment with the implementation below in your favorite code editor. Thanks for reading!

Implementation of AVL Tree in Javascript

```
class AVLTreeNode {
  constructor(value) {
    this.value = value;
    this.left = null;
    this.right = null;
    this.parent = null;
    this.height = 0;
  }
}

class AVLTree {
  constructor() {
    this.root = null;
  }

  insert(value) {
    // Define an inner helper function for the recursive insertion
    const insertHelper = (node, parent = null) => {
      // If the current node is null, this means we've found the place to insert the new value
      if (!node) {
        // Create a new node with the value
        const newNode = new AVLTreeNode(value);
        // Set the parent of the new node to be the passed in parent node
        newNode.parent = parent;
        // Return the newly created node
        return newNode;
      }
      // If the value to insert is less than the value of the current node
      if (value < node.value) {
        // Recursively call 'insertHelper' on the left child, passing in the current node as the parent
        node.left = insertHelper(node.left, node);
      } else if (value > node.value) {
        // If the value to insert is greater than the value of the current node
        // Recursively call 'insertHelper' on the right child, passing in the current node as the parent
        node.right = insertHelper(node.right, node);
      }
      // After the recursive calls, update the height of the current node
      // This is done after because the node's height may change due to the insert
      this._updateHeight(node);
      // Balance the node and return it. This is to ensure that AVL tree properties are maintained
      return this._balance(node);
    };
    // Call the 'insertHelper' function on the root node
    this.root = insertHelper(this.root);
    // Ensure that the root node's parent is null after insertions
    if (this.root.parent) {
      this.root.parent = null;
    }
  }

  delete(value) {
    // Define an inner helper function for the recursive deletion
    const deleteNode = (value, node) => {
      // If the current node is null, this means the value is not found in the tree
      if (!node) {
        return null;




      }
      // If the value to delete is less than the value of the current node
      if (value < node.value) {
        // Recursively call 'deleteNode' on the left child
        node.left = deleteNode(value, node.left);
      } else if (value > node.value) {
        // If the value to delete is greater than the value of the current node
        // Recursively call 'deleteNode' on the right child
        node.right = deleteNode(value, node.right);
      } else {
        // If the value is found
        // Case 1: Node is a leaf node
        if (!node.left && !node.right) {
          return null;
        } else if (!node.left) {
          // Case 2: Node has only right child
          // Assign the parent of the node to be deleted to its right child
          node.right.parent = node.parent;
          // Return right child
          return node.right;
        } else if (!node.right) {
          // Case 3: Node has only left child
          // Assign the parent of the node to be deleted to its left child
          node.left.parent = node.parent;
          // Return left child
          return node.left;
        } else {
          // Case 4: Node has two children
          // Find the in-order successor (minimum value in right subtree)
          const successor = this._findMin(node.right);
          // Replace the value of the node to be deleted with the value of the successor
          node.value = successor.value;
          // Delete the successor
          node.right = deleteNode(successor.value, node.right);
        }
      }
      // After the recursive calls, update the height of the current node
      // This is done after because the node's height may change due to the delete
      this._updateHeight(node);
      // Balance the node and return it. This is to ensure that AVL tree properties are maintained
      return this._balance(node);
    };
    // Call the 'deleteNode' function on the root node
    this.root = deleteNode(value, this.root);
    // Ensure that the root node's parent is null after deletions
    // because root node parent should always be null.
    if (this.root && this.root.parent) {
      this.root.parent = null;
    }
  }

  search(value, node = this.root) {
    // Check if the current node is null (reached a leaf node or an empty tree)
    if (!node) {
      // Value not found, return false
      return false;
    }
    // Check if the current node's value matches the search value




    if (value === node.value) {
      // Value found, return the node
      return node;
    }
    // If the search value is less than the current node's value, go to the left subtree
    if (value < node.value) {
      // Recursively call the search function on the left child node
      return this.search(value, node.left);
    }
    // If the search value is greater than the current node's value, go to the right subtree
    // This assumes the tree follows the convention of left child nodes having lesser values and right ch
    return this.search(value, node.right);
  }

  _balance(node) {
    // The balance function is used to restore the balance of an AVL tree
    // after an insertion or deletion operation that might have caused imbalance.

    // If the node is null, simply return the null node
    if (!node) {
      return node;
    }

    // Compute the balance factor of the node
    // The balance factor of a node is the difference in height between its left and right subtrees
    const nodeBF = this._getBalanceFactor(node);

    // Check if the left subtree is heavy (balance factor > 1)
    if (nodeBF > 1) {
      // Check if the left subtree is right heavy
      // This is the case of Left-Right imbalance
      // which requires a Left-Right rotation to fix
      if (this._getBalanceFactor(node.left) < 0) {
        return this._leftRightRotation(node);
      }
      // If the left subtree is left heavy, a simple Right rotation is enough to restore balance
      return this._rightRotation(node);
    }
    // Check if the right subtree is heavy (balance factor < -1)
    else if (nodeBF < -1) {
      // Check if the right subtree is left heavy
      // This is the case of Right-Left imbalance
      // which requires a Right-Left rotation to fix
      if (this._getBalanceFactor(node.right) > 0) {
        return this._rightLeftRotation(node);
      }
      // If the right subtree is right heavy, a simple Left rotation is enough to restore balance
      return this._leftRotation(node);
    }

    // If the node is already balanced, no rotation is required, simply return the node
    return node;
  }

  _updateHeight(node) {
    // To update the height of a node in an AVL tree, we first calculate the height of its left and right
    const leftSubtreeHeight = this._getNodeHeight(node.left);
    const rightSubtreeHeight = this._getNodeHeight(node.right);




    // The height of a node is defined as 1 plus the maximum height of its left or right subtree
    // This is based on the definition of the height of a node in a binary tree (the number of edges on t
    // We use the built-in 'Math.max' function to find the maximum between the heights of the left and ri
    // Then, we add 1 to this maximum height to calculate the new height of the node
    node.height = Math.max(leftSubtreeHeight, rightSubtreeHeight) + 1;
  }

  _getNodeHeight(node) {
    // Check if the node is null
    // This is important because we may call this method on a leaf node's child, which is null
    if (!node) {
      // If the node is null, return -1, as by convention, the height of a null node is considered -1
      return -1;
    }
    // If the node is not null, return its height property
    // The height of a node is stored in its 'height' property in our AVL Tree implementation
    return node.height;
  }

  _getBalanceFactor(node) {
    // Get the height of the left subtree of the given node
    // We use the '_getNodeHeight' method, which takes a node as an argument and returns its height
    // If the left child of the node is null, '_getNodeHeight' returns -1
    const leftSubtreeHeight = this._getNodeHeight(node.left);

    // Similarly, get the height of the right subtree of the given node
    const rightSubtreeHeight = this._getNodeHeight(node.right);
    // The balance factor of a node in an AVL tree is the difference between the height of its left subtr
    // So, we calculate the balance factor by subtracting the height of the right subtree from the height
    // This can result in a negative number, zero, or a positive number
    // According to the AVL tree property, this value should always be -1, 0, or 1 for all nodes in a bal
    return leftSubtreeHeight - rightSubtreeHeight;
  }

  _leftRotation(node) {
    // A left rotation is performed when the right child of a node has a greater height than its left chi
    // We start the rotation by creating a temporary reference to the node's right child because this noc
    const temp = node.right;

    // The right child of the current node is then replaced by the left child of the 'temp' node (i.e., t
    node.right = temp.left;

    // If the 'temp' node had a left child, we update its parent to be the 'node'
    if (temp.left) {
      temp.left.parent = node;
    }

    // The 'node' becomes the left child of the 'temp' node
    temp.left = node;

    // The parent of the 'temp' node becomes the parent of the 'node', which maintains the binary search
    temp.parent = node.parent;

    // The parent of the 'node' is updated to be the 'temp' node because 'temp' is the new parent after r
    node.parent = temp;




    // If the 'temp' node has a parent (i.e., it's not the root of the tree), we update the parent's chil
    if (temp.parent) {
      if (temp.parent.left === node) {
        temp.parent.left = temp;
      } else if (temp.parent.right === node) {
        temp.parent.right = temp;
      }
    }

    // After the rotation, we update the height of the 'node' and 'temp' nodes because their subtrees mig
    this._updateHeight(node);
    this._updateHeight(temp);

    // The 'temp' node is returned because it's the new parent node after the left rotation
    return temp;
  }

  _rightRotation(node) {
    // A right rotation is performed when the left child of a node has a greater height than its right ch
    // We start the rotation by creating a temporary reference to the node's left child because this node
    const temp = node.left;

    // The left child of the current node is then replaced by the right child of the 'temp' node (i.e., t
    node.left = temp.right;

    // If the 'temp' node had a right child, we update its parent to be the 'node'
    if (temp.right) {
      temp.right.parent = node;
    }
    // The 'node' becomes the right child of the 'temp' node
    temp.right = node;

    // The parent of the 'temp' node becomes the parent of the 'node', which maintains the binary search
    temp.parent = node.parent;

    // The parent of the 'node' is updated to be the 'temp' node because 'temp' is the new parent after r
    node.parent = temp;

    // If the 'temp' node has a parent (i.e., it's not the root of the tree), we update the parent's chil
    if (temp.parent) {
      if (temp.parent.left === node) {
        temp.parent.left = temp;
      } else if (temp.parent.right === node) {
        temp.parent.right = temp;
      }
    }

    // After the rotation, we update the height of the 'node' and 'temp' nodes because their subtrees mig
    this._updateHeight(node);
    this._updateHeight(temp);

    // The 'temp' node is returned because it's the new parent node after the right rotation
    return temp;
  }

  _leftRightRotation(node) {
    // A left-right rotation is performed when the right child of the left child of a node has a greater




    // This situation creates an imbalance that cannot be fixed by a simple rotation

    // First, a left rotation is performed on the left child of the node
    // The purpose of this step is to convert the imbalance into a form that can be fixed by a simple rot
    node.left = this._leftRotation(node.left);

    // After converting the imbalance, we perform a right rotation on the original node
    // This step fixes the imbalance and restores the AVL tree property
    return this._rightRotation(node);
  }

  _rightLeftRotation(node) {
    // A right-left rotation is performed when the left child of the right child of a node has a greater
    // This situation creates an imbalance that cannot be fixed by a simple rotation

    // First, a right rotation is performed on the right child of the node
    // The purpose of this step is to convert the imbalance into a form that can be fixed by a simple rot
    node.right = this._rightRotation(node.right);

    // After converting the imbalance, we perform a left rotation on the original node
    // This step fixes the imbalance and restores the AVL tree property
    return this._leftRotation(node);
  }

  _findMin(node = this.root) {
    // The _findMin method is used to find the node with the smallest value in a binary search tree.
    // By the properties of a binary search tree, the smallest node is the leftmost node.

    // We start from the given node, defaulting to the root of the tree if no node is provided.
    let currentNode = node;

    // While there is a node to inspect and that node has a left child
    while (currentNode && currentNode.left) {
      // Move to the left child of the current node
      // This is because in a binary search tree, all values in the left subtree are less than the value
      currentNode = currentNode.left;
    }

    // When there are no more left children to move to, we've found the smallest value node
    // Return this node
    return currentNode;
  }

  // Displays an array that will represent the tree
  // in level-order, with each sub-array representing a level of the tree.
  levelOrderTraversal() {
    // 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;
  }
}
```

```
export default AVLTree;
```