AVL Tree

Today, we're going to talk about something called an AVL tree. Now, AVL tree might sound like a strange name, but it's actually named after the people who invented it, Georgy Adelson-Velsky and Landis.

Now, what's special about an AVL tree is that it's a type of binary search tree, just like the regular binary search trees we've learned about. But, it's a bit special because it's a self-balancing tree.

In an AVL tree, each node carries some extra information called a balance factor. This balance factor can have three values: -1, 0, or +1.

This balance factor is what makes an AVL tree unique. It helps the tree automatically balance itself as you insert or delete nodes. So, if the tree starts leaning too much to one side, it automatically adjusts itself to stay balanced.

So, that's the AVL tree, a self-balancing binary search tree named after its inventors, Adelson-Velsky and Landis. It keeps itself balanced using these special balance factors.

Balance Factor

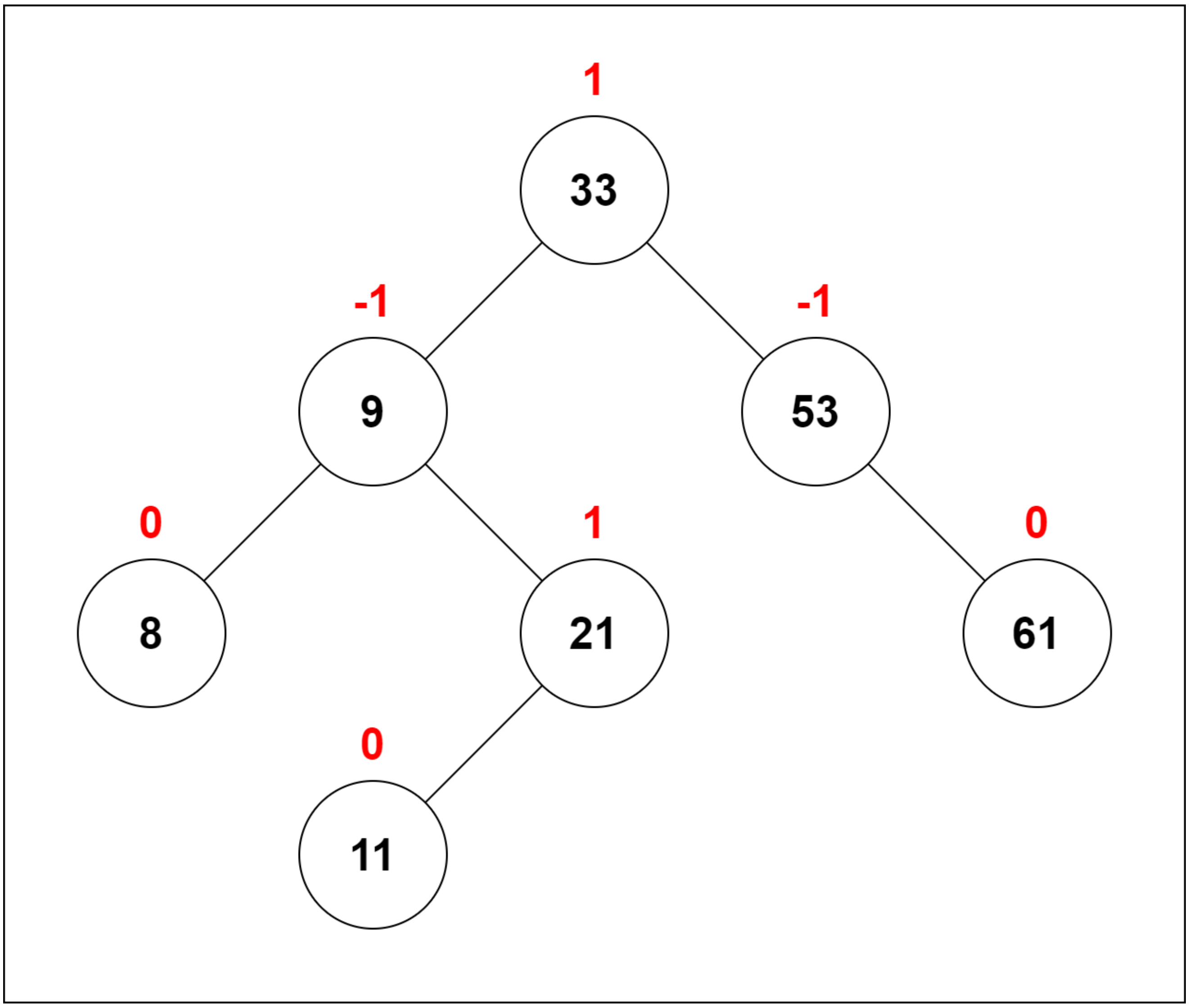
This balance factor is pretty important because it's what helps the AVL tree stay balanced, just like its name suggests.

So, what exactly is the balance factor? Well, it's the difference between the height of the left subtree and the height of the right subtree of a particular node in the AVL tree. In simpler terms, it's like checking if one side of the tree is much taller than the other.

Mathematically, you can calculate it like this: Balance Factor = (Height of Left Subtree - Height of Right Subtree) or (Height of Right Subtree - Height of Left Subtree).

Now, here's the crucial part: the value of the balance factor should always be one of these three numbers: -1, 0, or +1. If it's anything else, it means the tree is getting out of balance, and that's when the AVL tree works its magic to bring it back into balance.

To help you visualize, let's take a look at an example of a balanced AVL tree. This tree is a great example of how the balance factor keeps things in check.



Operations on an AVL tree

Alright, let's delve into some of the operations we can perform on an AVL tree. These operations are what keep the tree balanced and working smoothly.

1. Rotating Subtrees: One of the key operations in AVL trees is rotating subtrees. Think of this like rearranging branches in the tree to maintain balance. There are different types of rotations - like right rotations and left rotations - and they are used depending on the situation to keep the tree in check.

2. Insert Nodes: This operation is all about adding new nodes to the AVL tree while maintaining its balance. When you insert a new node, the tree might become unbalanced, so rotations are often needed here too to ensure it stays balanced.

3. Delete Nodes: Deleting nodes is the flip side of inserting. When you remove a node from the tree, it can also disrupt the balance, so AVL trees have mechanisms to re-balance themselves after deletions.

These operations might sound a bit technical, but they're essentially the tools AVL trees use to make sure they stay nice and balanced, which in turn keeps their search and other operations speedy.

Rotating the subtrees in an AVL Tree

Now, let's talk about rotating subtrees in an AVL tree. Think of this as a way to rearrange parts of the tree to keep it balanced. There are two main types of rotations we use in AVL trees:

1. Left Rotate: This is like shifting a branch to the left. Imagine you have a tree, and you want to balance it better. You can perform a left rotation to move nodes around and achieve that balance.

2. Right Rotate: Conversely, a right rotation is like moving a branch to the right. It's another tool to help with balance. When a tree becomes unbalanced, you can use a right rotation to rearrange nodes and bring it back into balance.

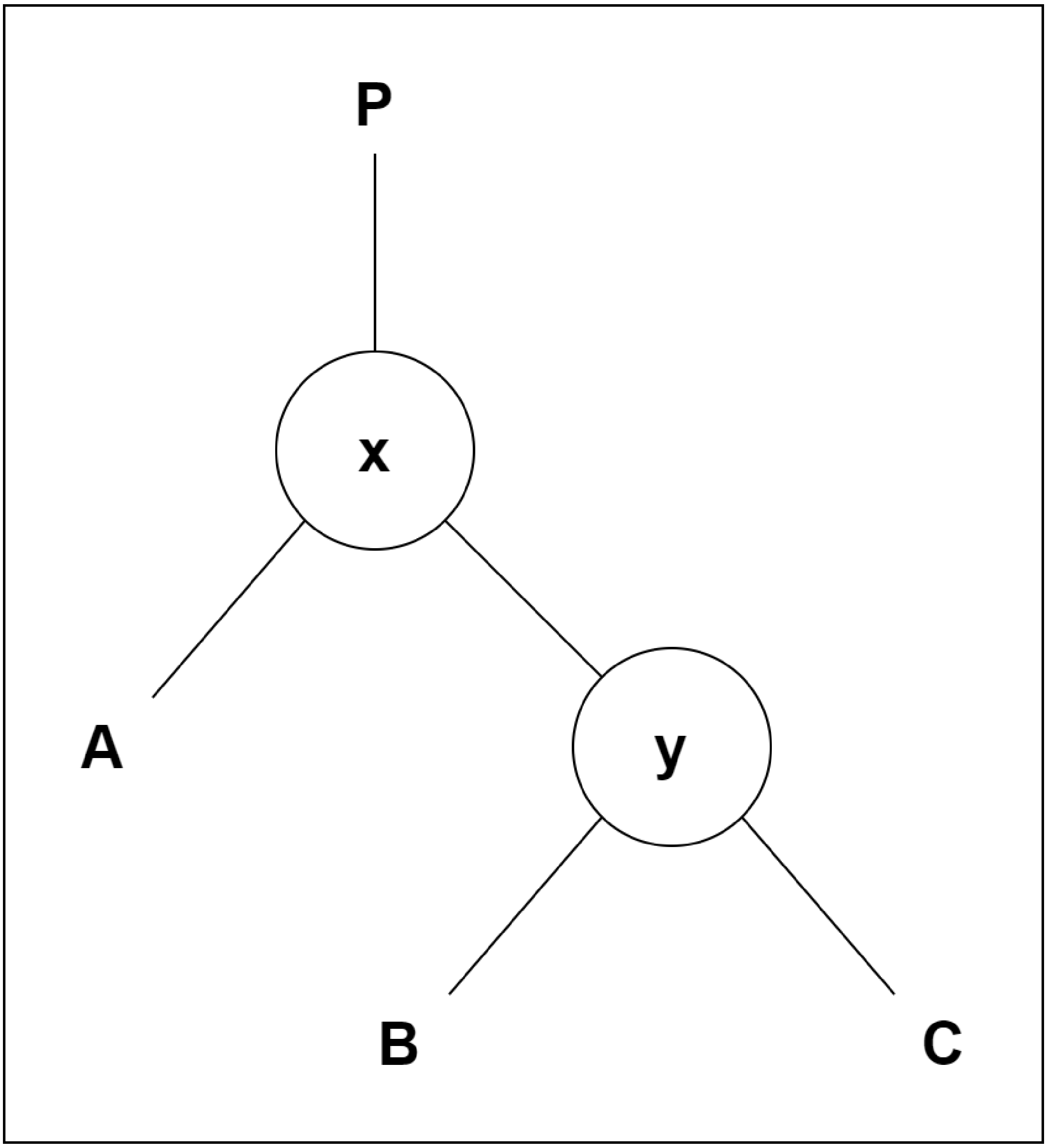
These rotations are essential in maintaining the AVL tree's self-balancing property, making sure it stays efficient for operations like searching and inserting nodes.

Left Rotate

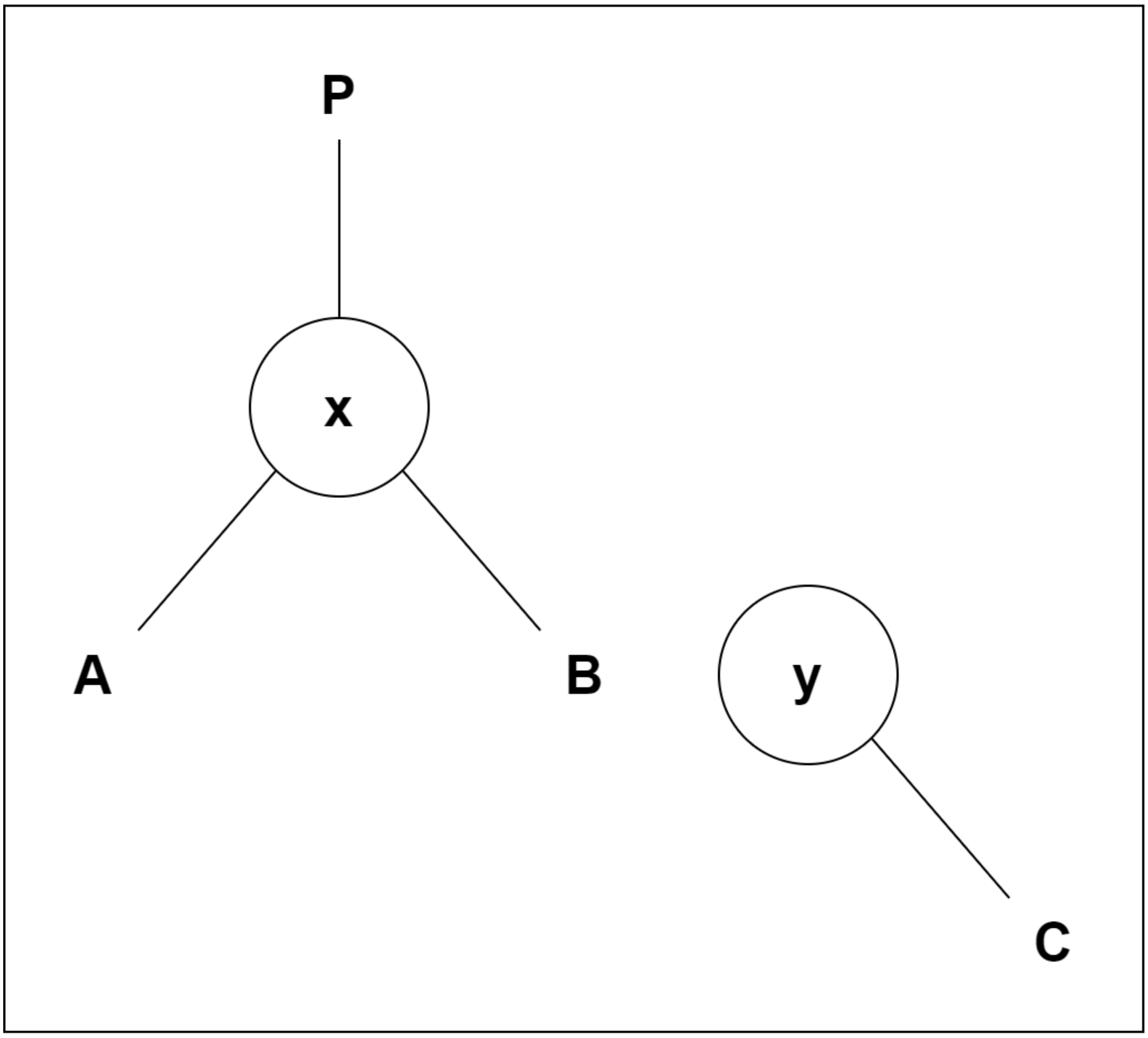
Now, let's dive into the details of left rotation in an AVL tree. This operation helps in rebalancing the tree.

Here's how it works:

1. First, we have our tree, which might have become unbalanced due to an insertion or some other operation.

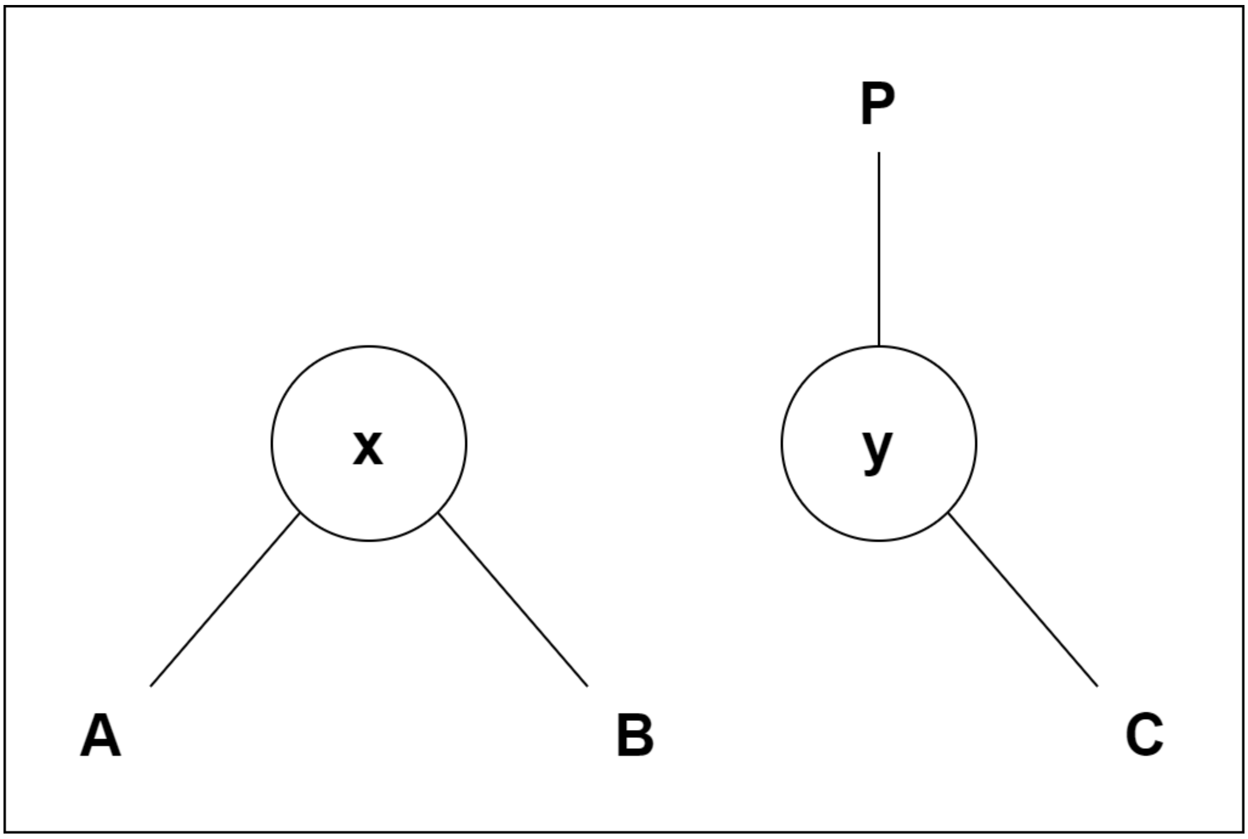


2. Now, we identify a specific node in the tree, let's call it 'y', which needs to be rotated to the left.

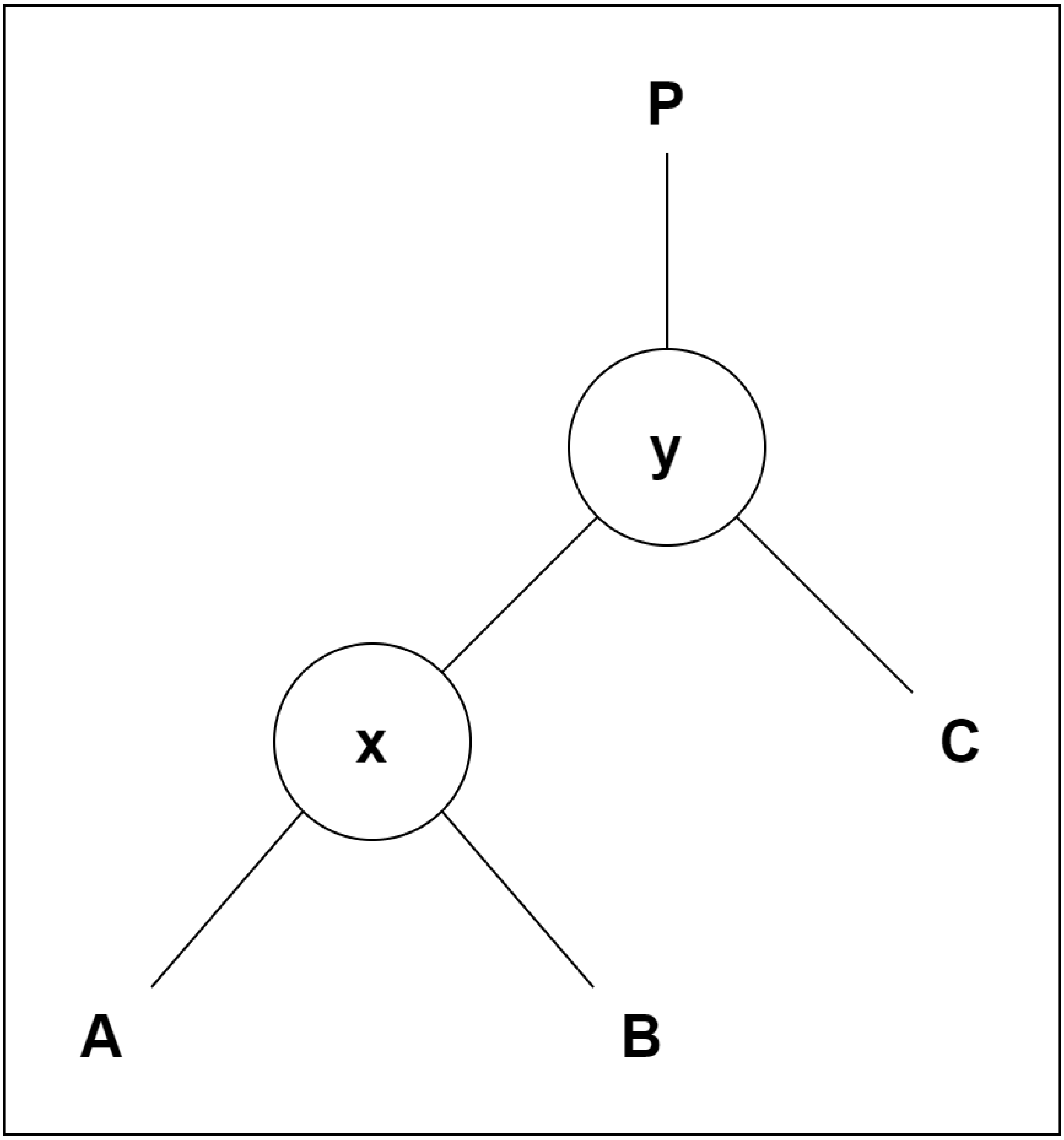


3. If 'y' has a left subtree, we make 'x' (the parent of 'y') the new parent of 'y's left subtree.
4. If 'x' was the root of the whole tree (meaning it has no parent), then 'y' becomes the new root of the tree.
5. If 'x' has a parent ('p'), we adjust the relationship between 'p' and 'y'. If 'x' was the left child of 'p', 'y' becomes the new left child of 'p'. If 'x' was the right child of 'p', 'y'

becomes the new right child of 'p'.



6. Finally, we update the parent of 'x', making 'y' the new parent of 'x'.



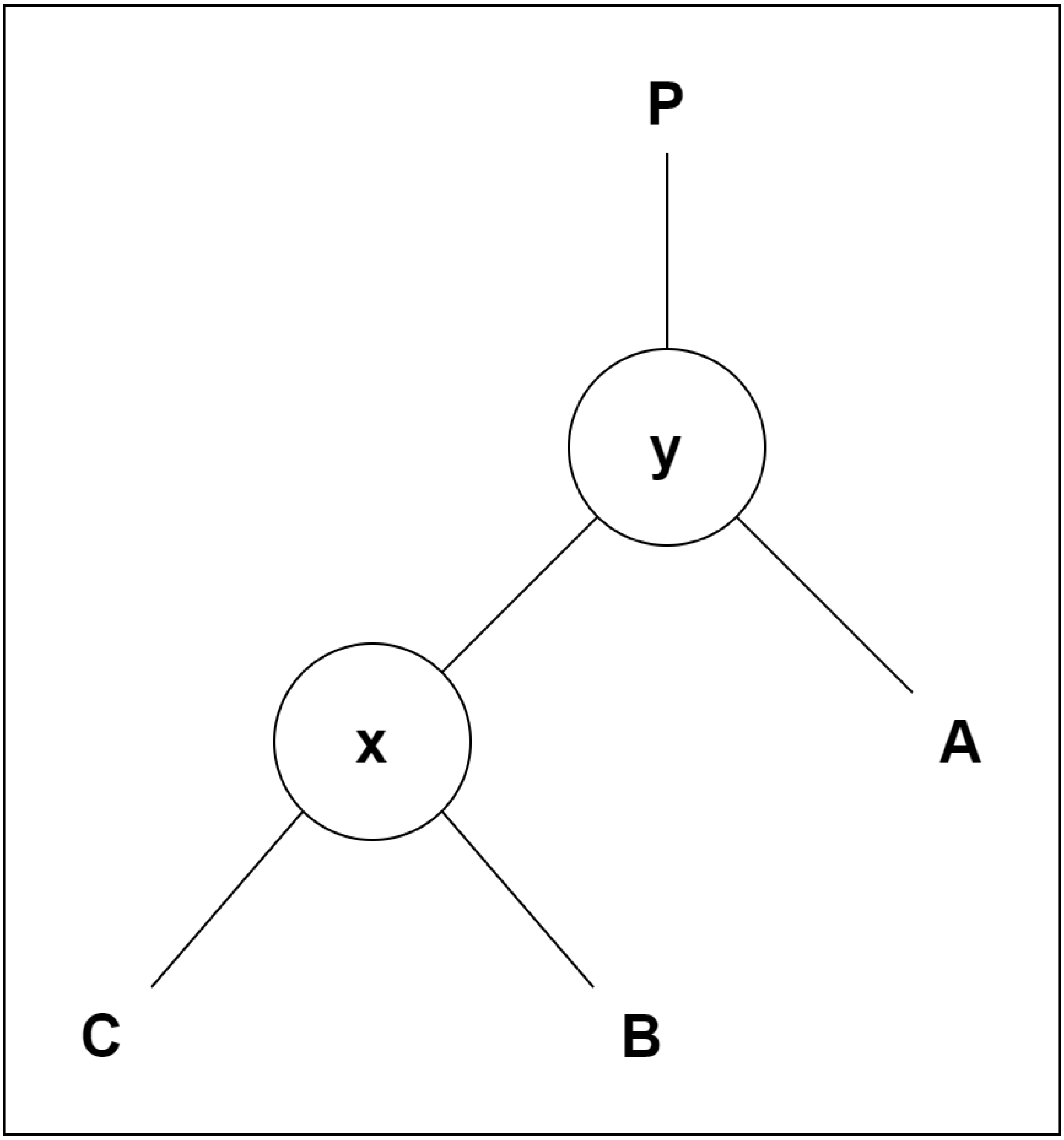
This left rotation helps maintain the balance of the AVL tree, ensuring that it remains efficient for various operations.

Right Rotate

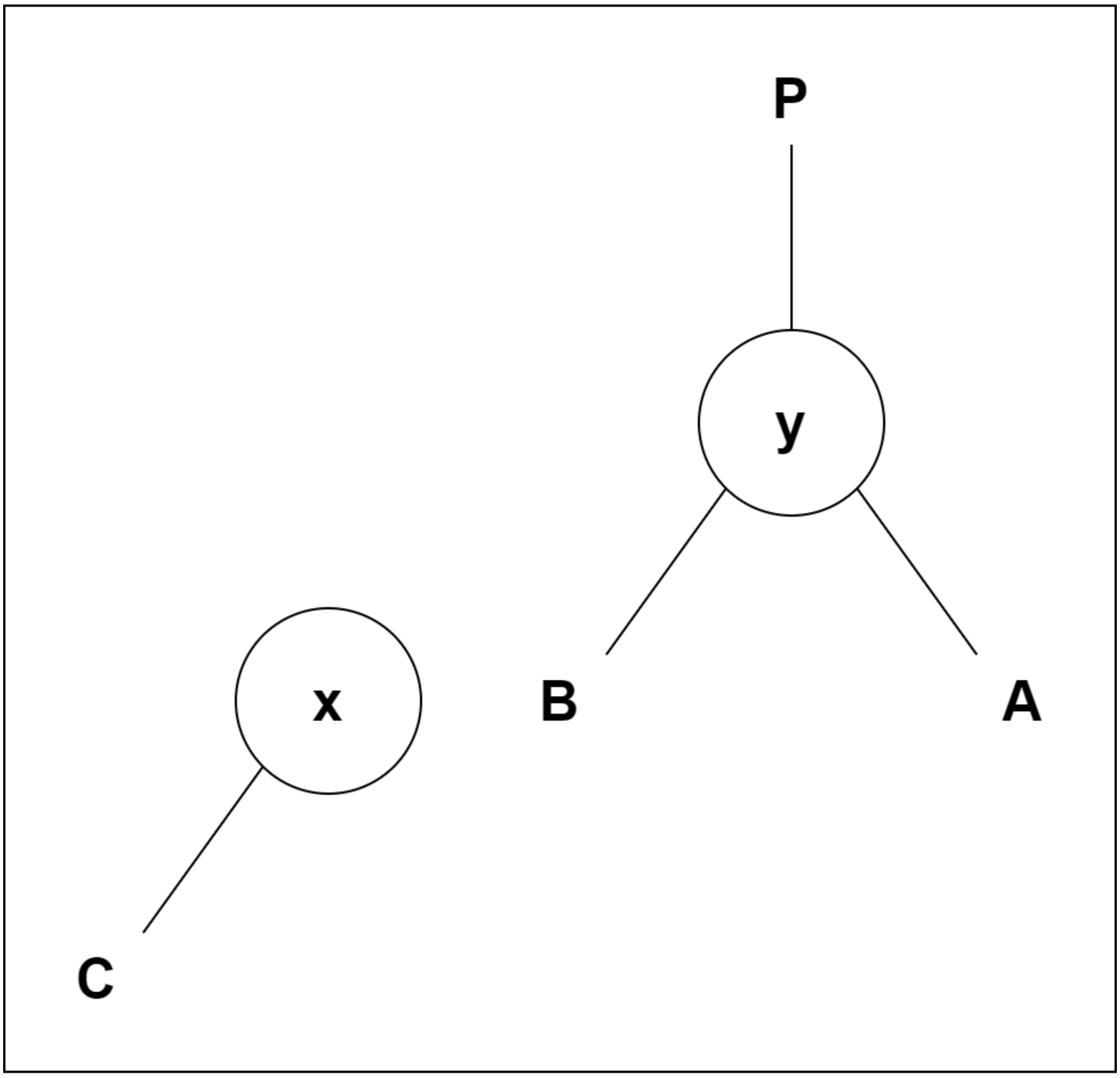
Now, let's explore right rotation in an AVL tree. This operation helps us maintain the tree's balance.

Here's how it works:

1. We start with our AVL tree, which might have become unbalanced due to an insertion or some other operation.

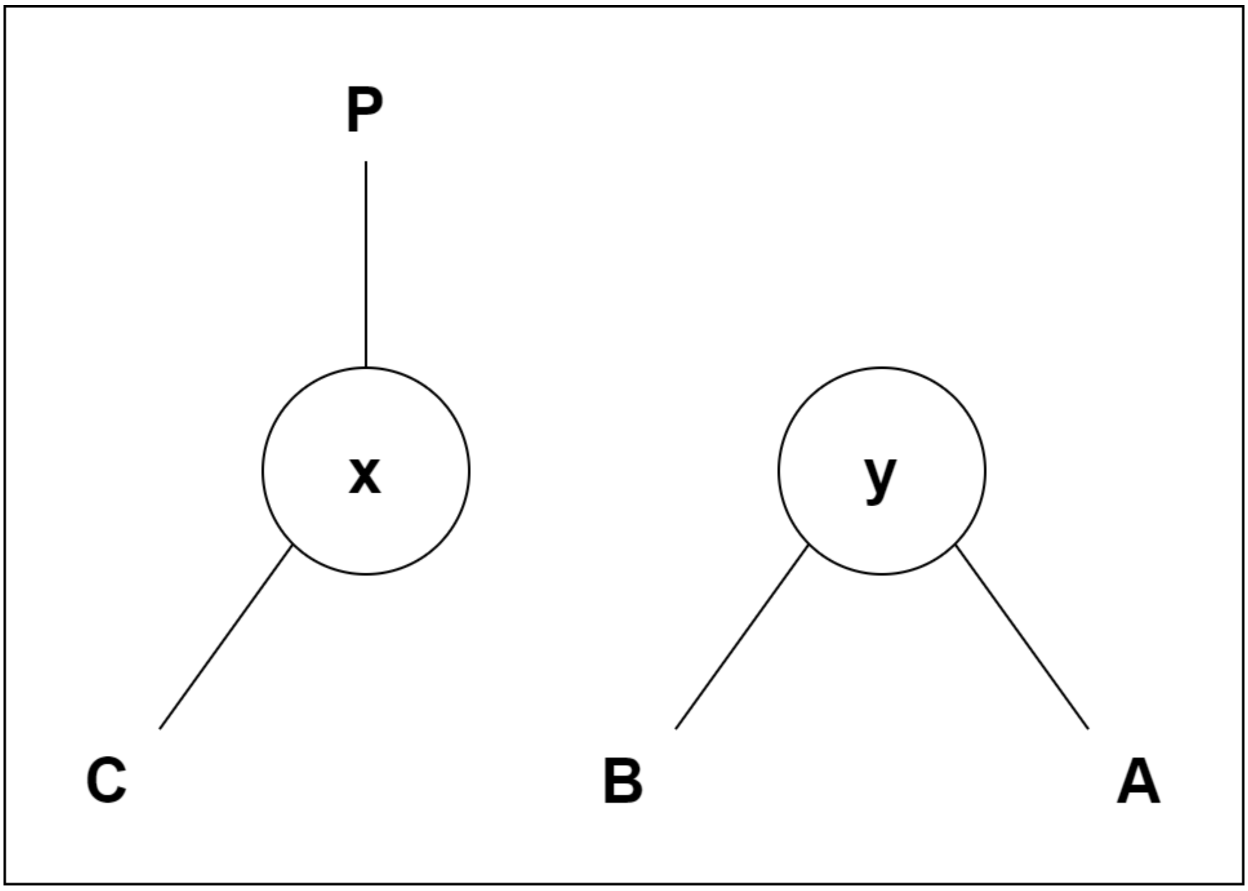


2. We identify a specific node in the tree, let's call it 'x', that needs to be rotated to the right.

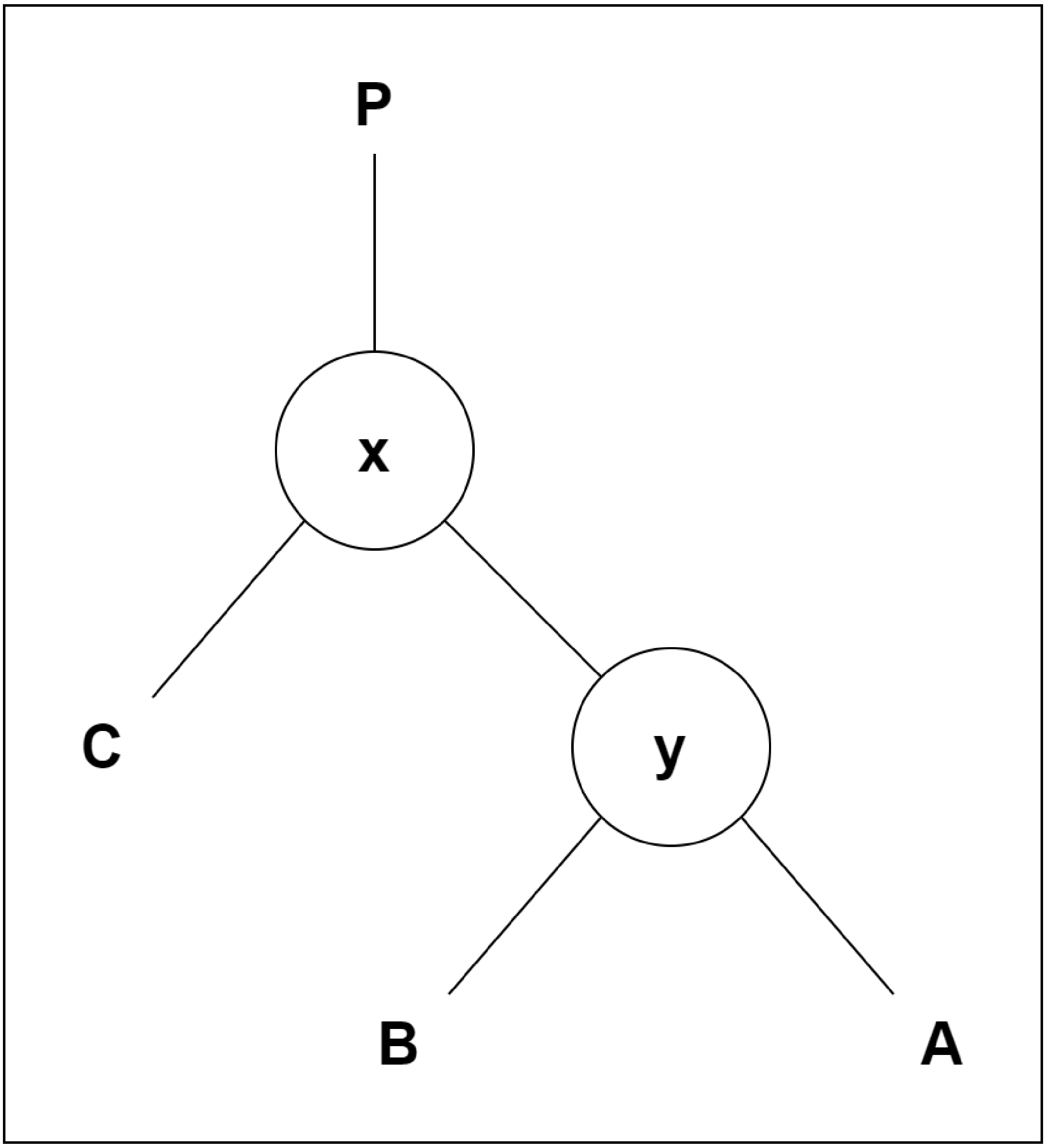


3. If 'x' has a right subtree, we make 'y' (the parent of 'x') the new parent of 'x's right subtree.
4. If 'y' was the root of the entire tree (meaning it has no parent), then 'x' becomes the new root of the tree.
5. If 'y' has a parent ('p'), we adjust the relationship between 'p' and 'x'. If 'y' was the right child of 'p', 'x' becomes the new right child of 'p'. If 'y' was the left child of 'p', 'x'

becomes the new left child of 'p'.



6. Finally, we update the parent of 'y', making 'x' the new parent of 'y'.



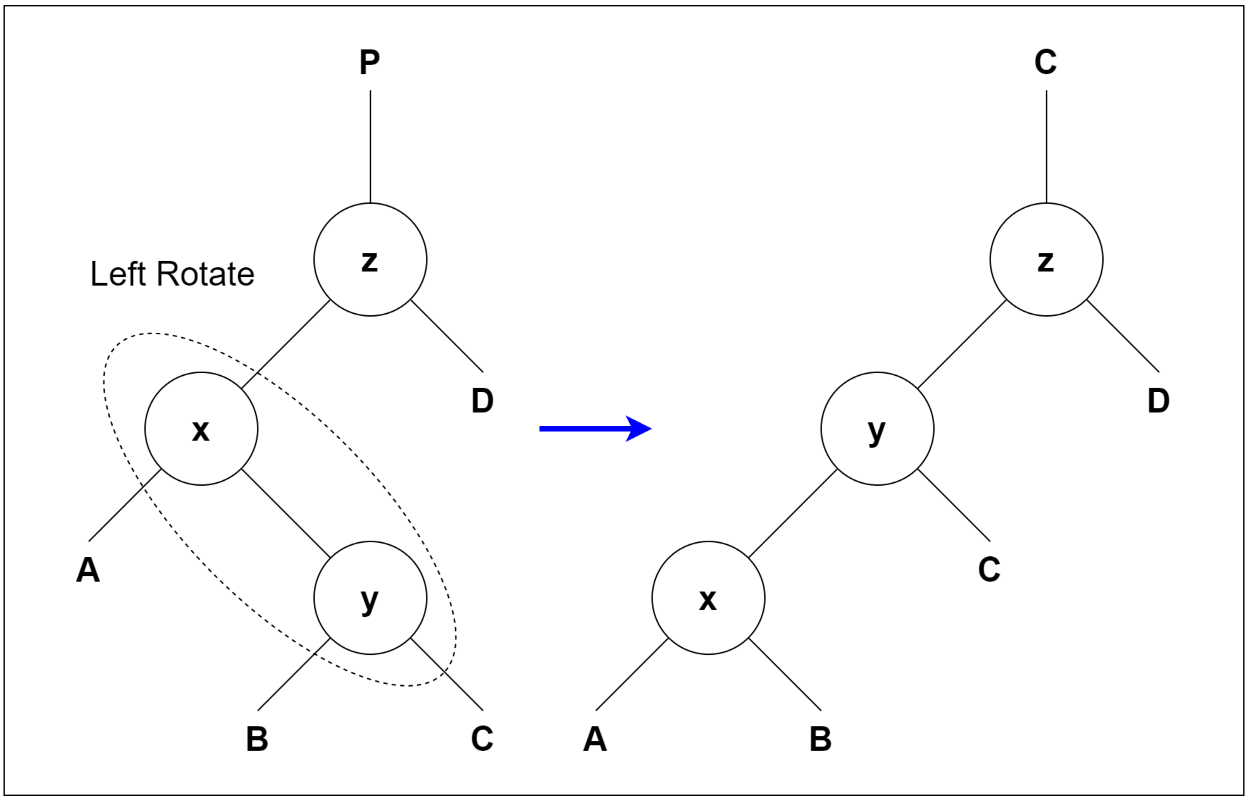
This right rotation operation helps maintain the balance of the AVL tree, ensuring that it remains efficient for various operations.

Left-Right and Right-Left Rotate

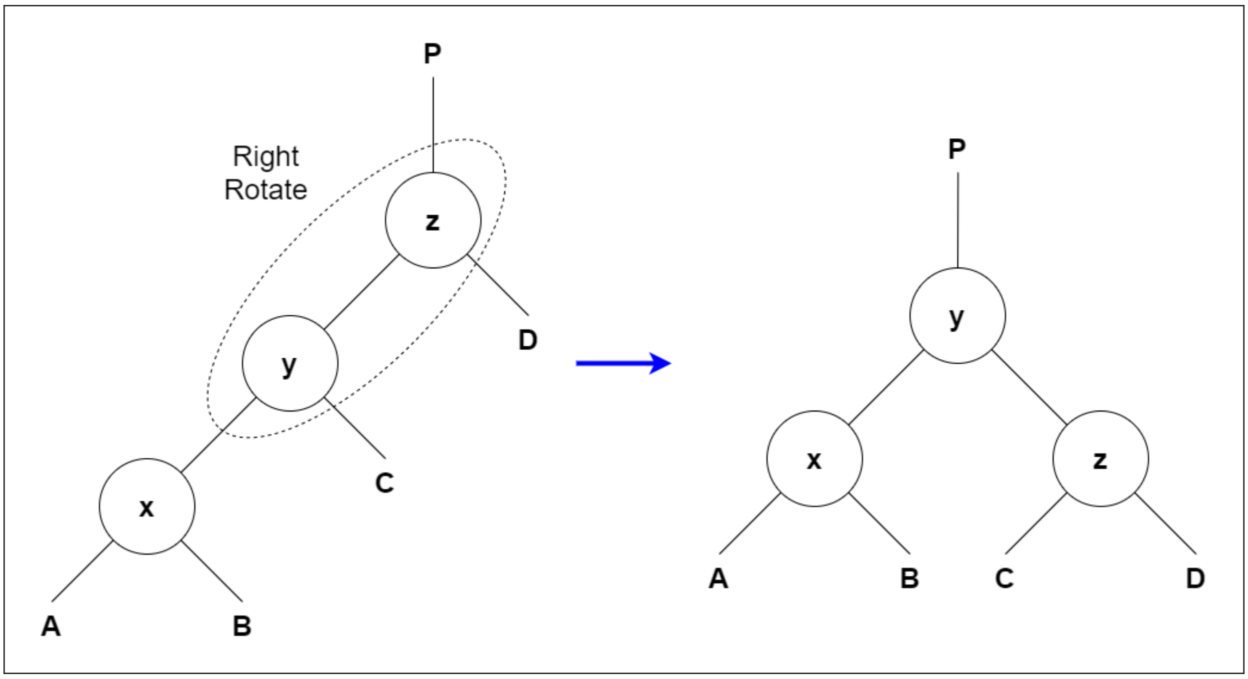
Now, let's discuss two more rotation operations in AVL trees: left-right rotation and right-left rotation. These rotations help us balance the tree when it becomes unbalanced.

Left-Right Rotation:

1. First, we perform a left rotation on nodes 'x' and 'y'.

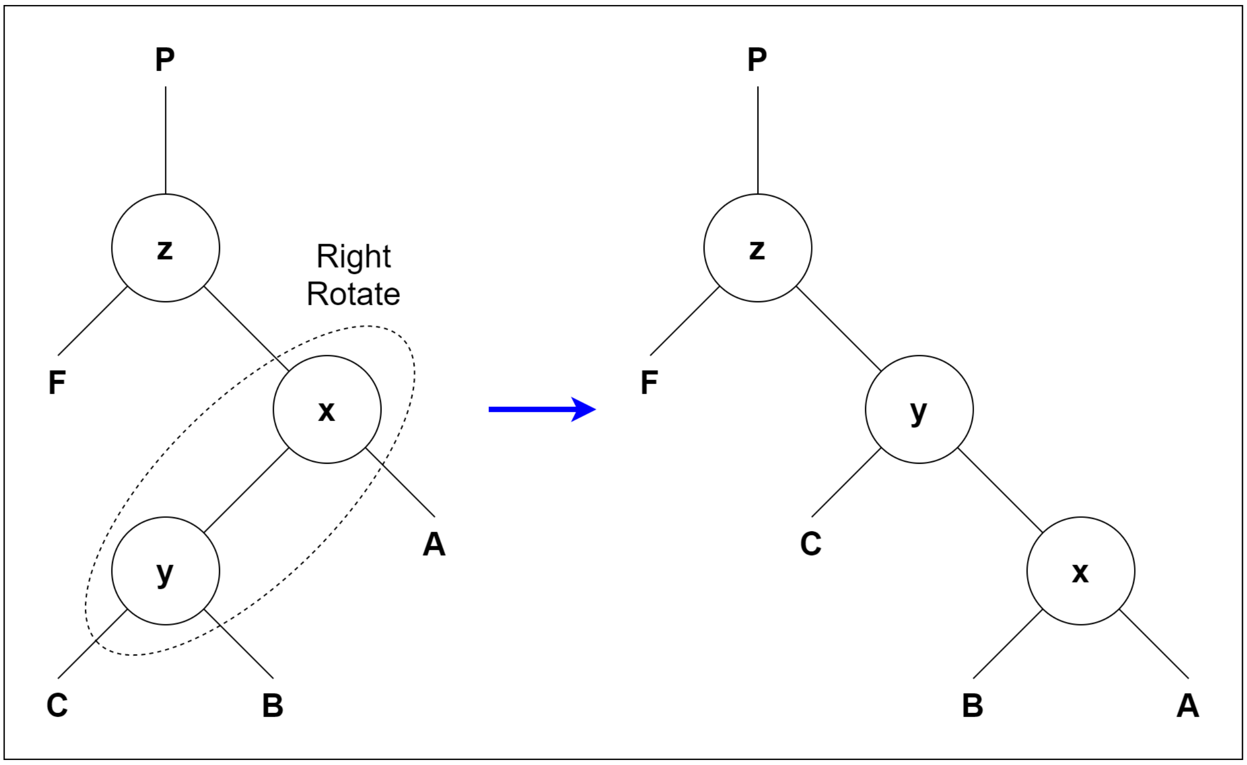


2. Then, we follow it with a right rotation on nodes 'y' and 'z'. This rearranges the nodes and rebalances the tree.

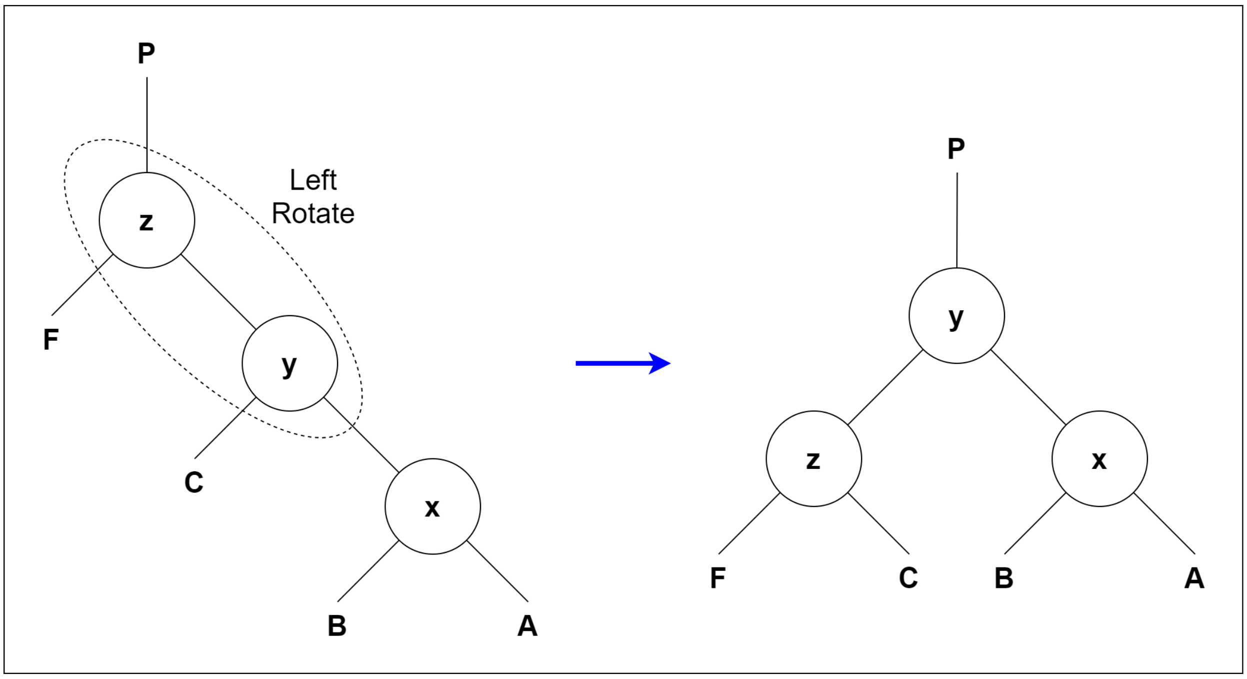


Right-Left Rotation:

1. Conversely, in right-left rotation, we start with a right rotation on nodes 'x' and 'y'.



2. After that, we do a left rotation on nodes 'z' and 'y'.

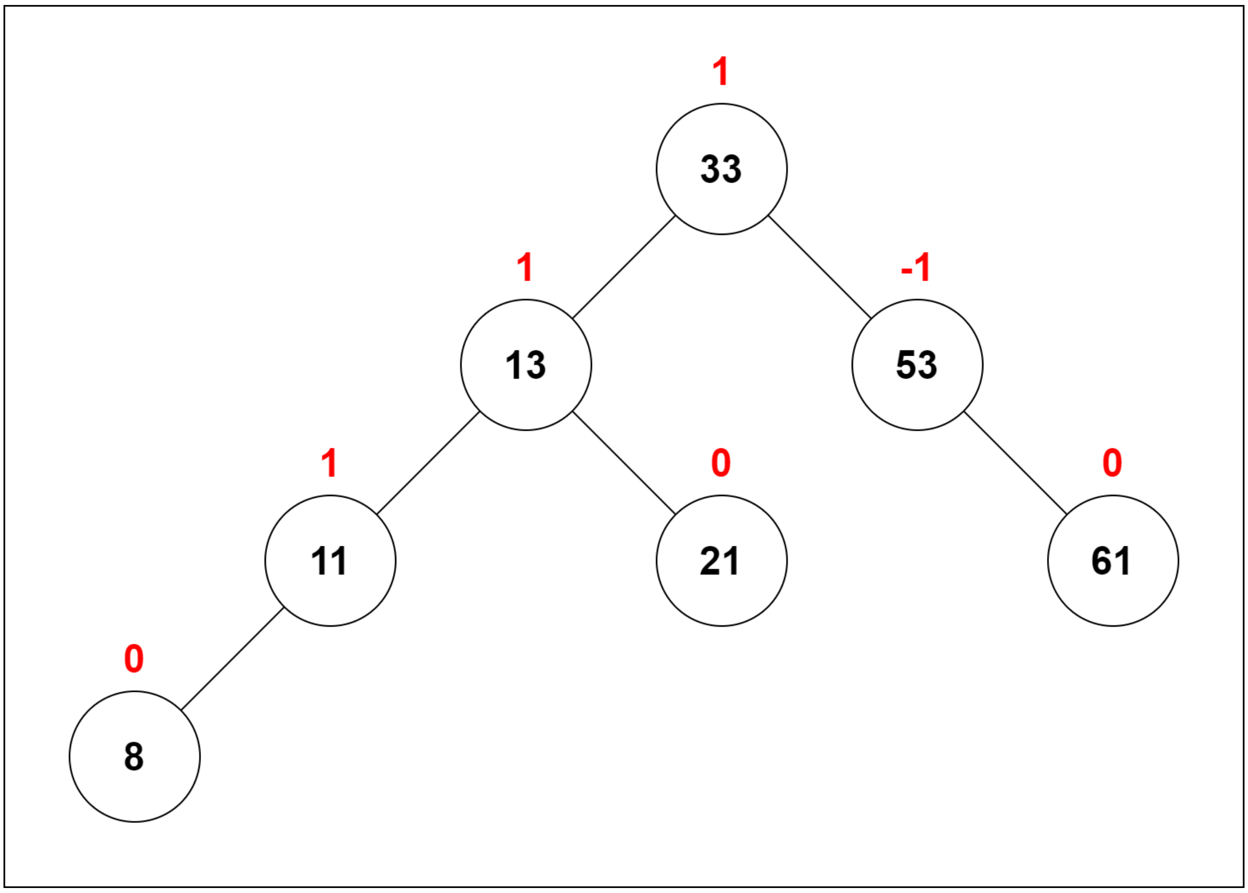


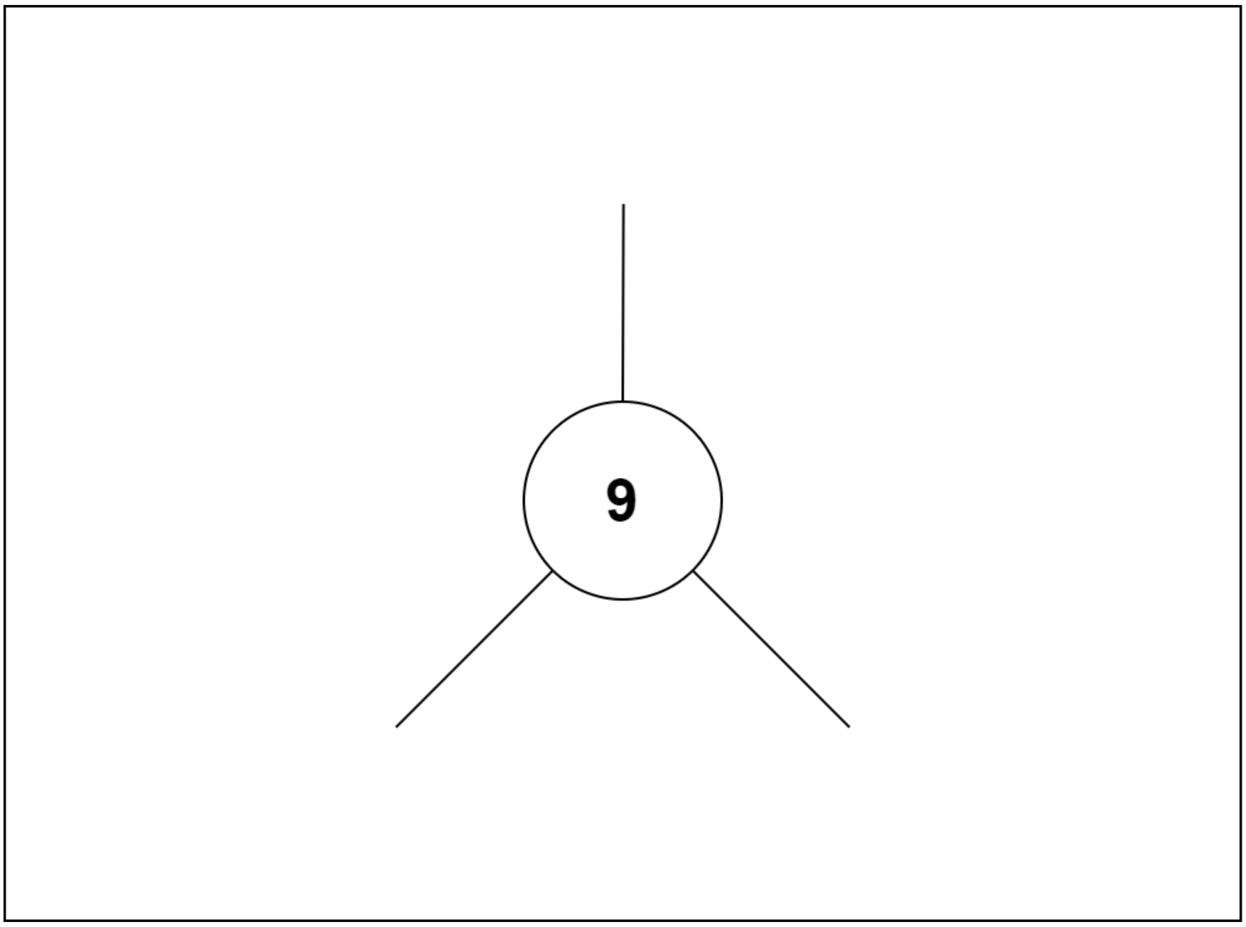
These rotation operations are essential for maintaining the balance of the AVL tree, ensuring it remains efficient for various operations. Balancing is crucial because it keeps the tree's height in check and guarantees that search, insertion, and deletion operations are efficient with a time complexity of O(log n).

Algorithm to insert a newNode

Alright, let's break down the algorithm for inserting a new node into an AVL tree step by step:

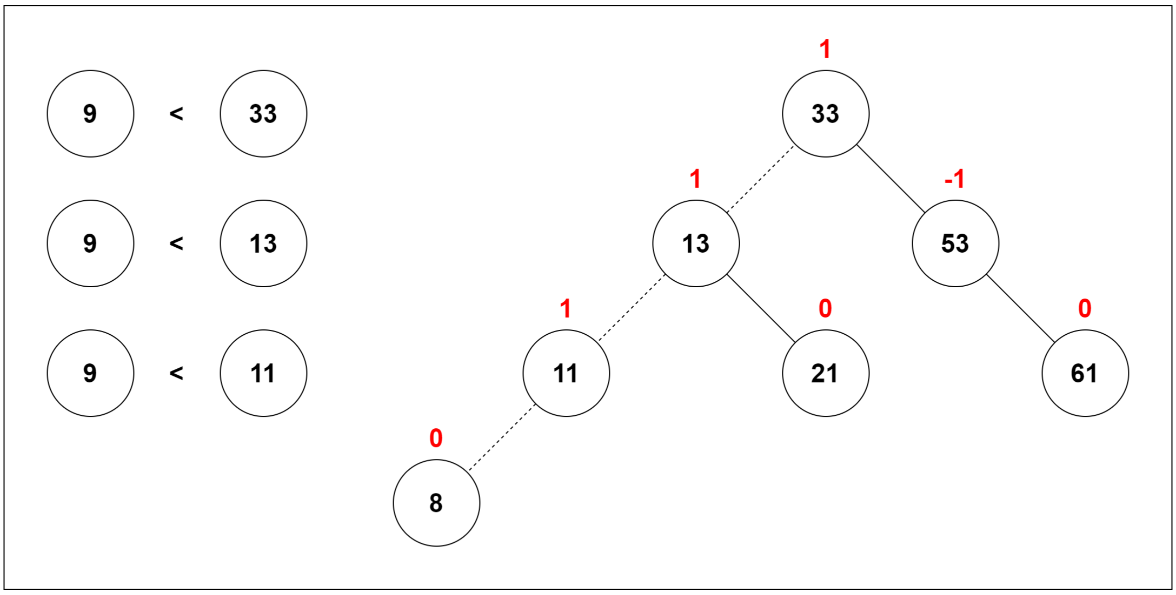
1. Initially, you have the AVL tree, and you want to insert a new node into it.



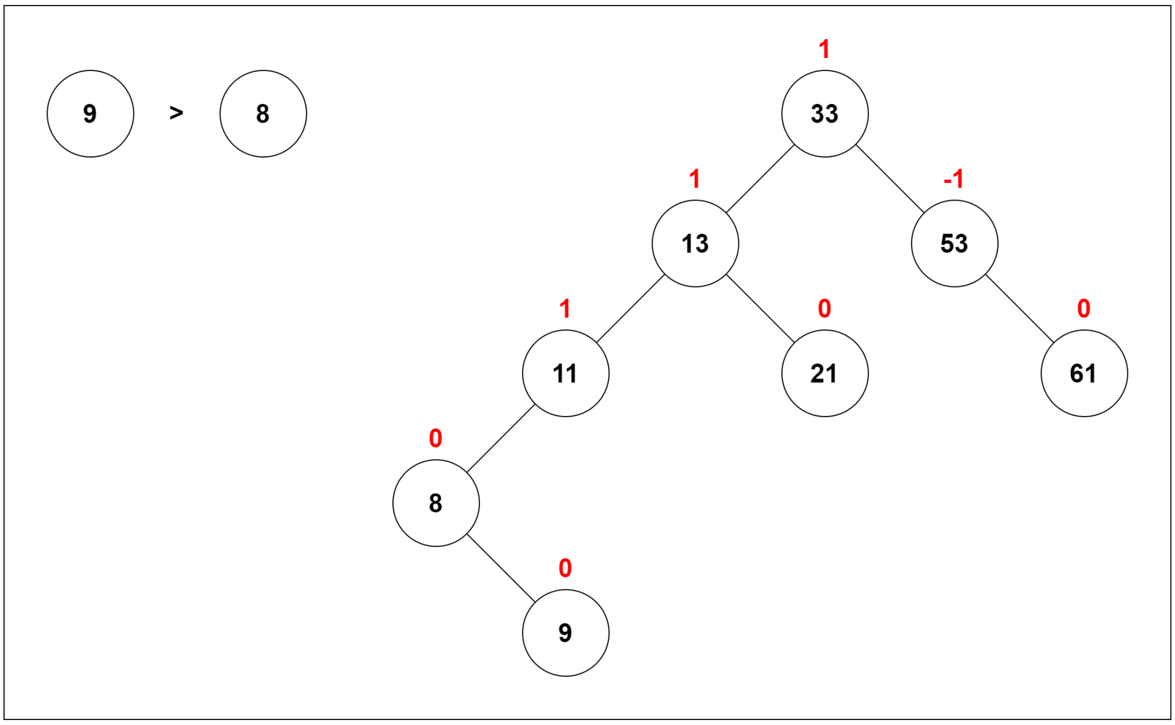


2. You start at the root of the tree.
3. You compare the key of the new node (let's call it 'newKey') with the key of the current node (let's call it 'rootKey').

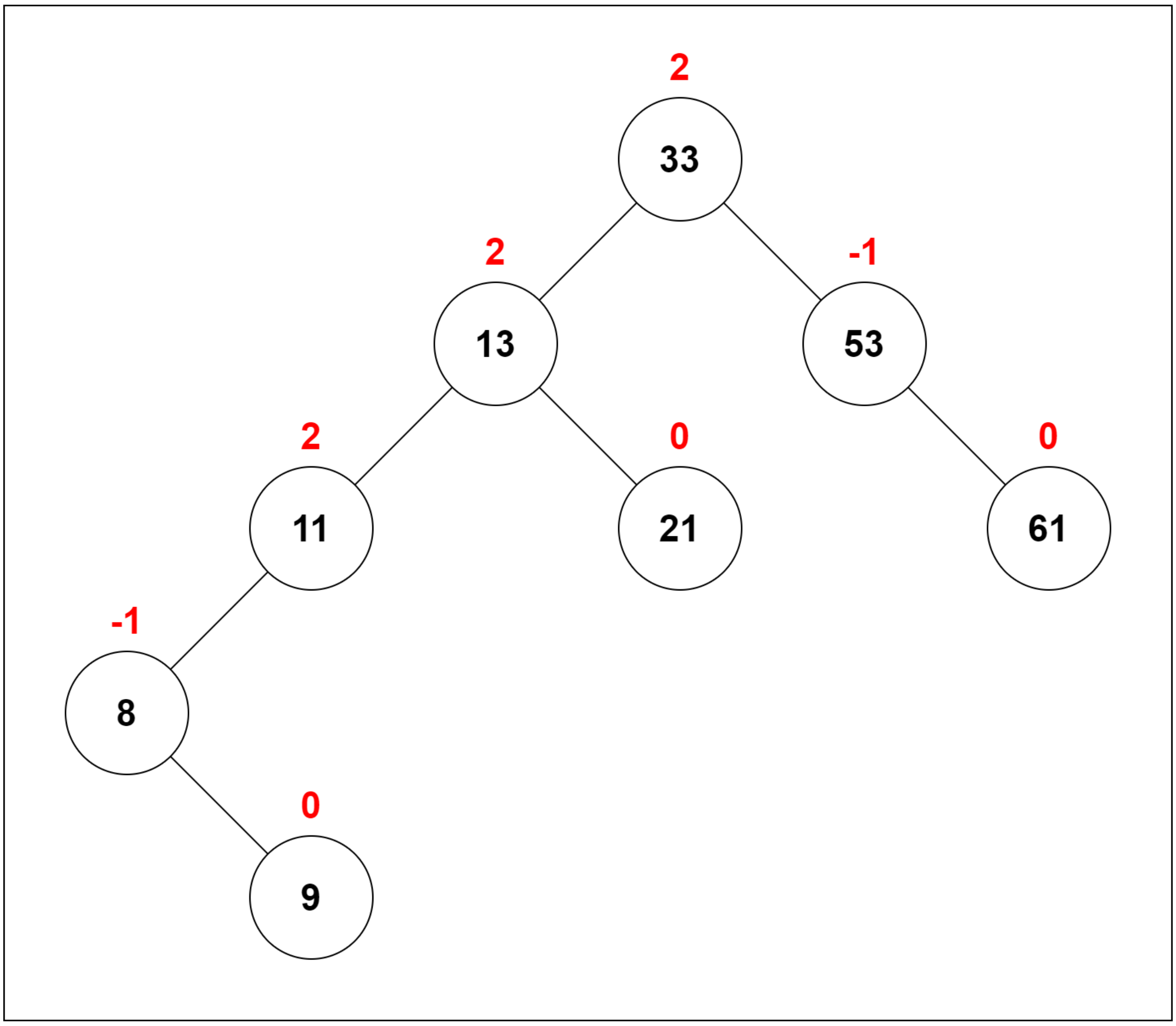
- If 'newKey' is less than 'rootKey', you move to the left subtree and repeat this process.
- If 'newKey' is greater than 'rootKey', you move to the right subtree and repeat this process.
- If 'newKey' is equal to 'rootKey', you've found the spot where the new node should go, so you insert the new node as a leaf node here and return.



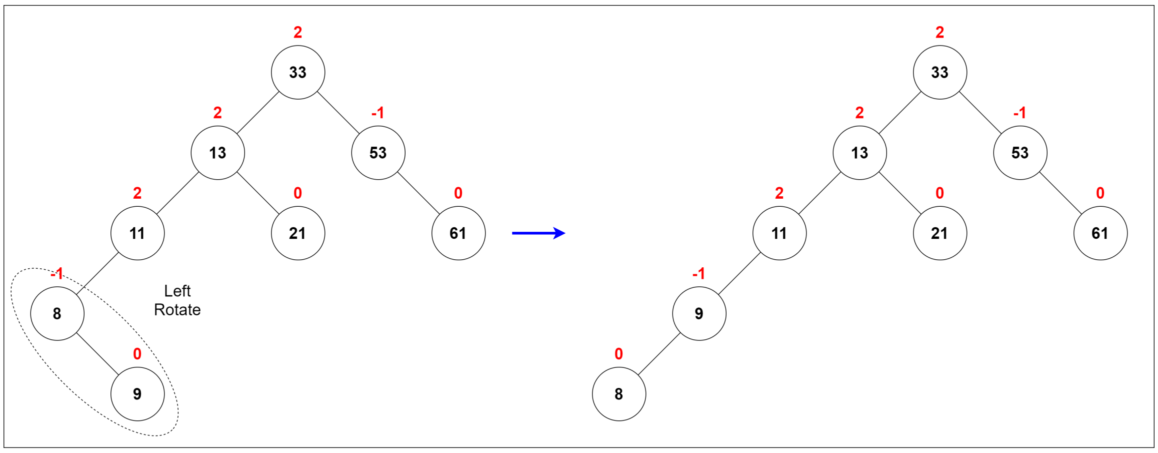
4. Now, you compare the key of the new node ('newKey') with the key of the leaf node you reached (let's call it 'leafKey').
   - If 'newKey' is less than 'leafKey', you make the new node the left child of the leaf node.
   - If 'newKey' is greater than 'leafKey', you make the new node the right child of the leaf node.

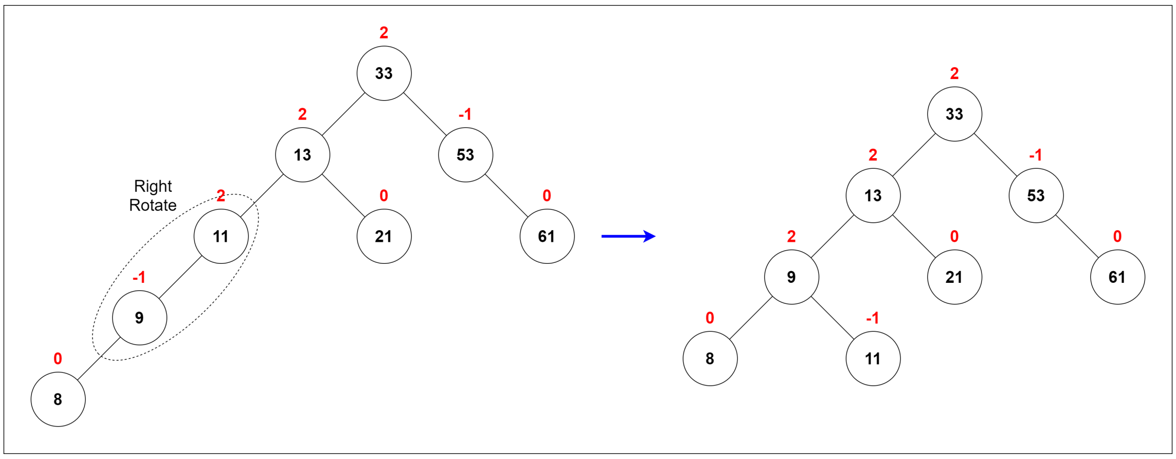


5. After inserting the new node, you need to update the balance factor of the nodes along the path you took from the root to the leaf node.

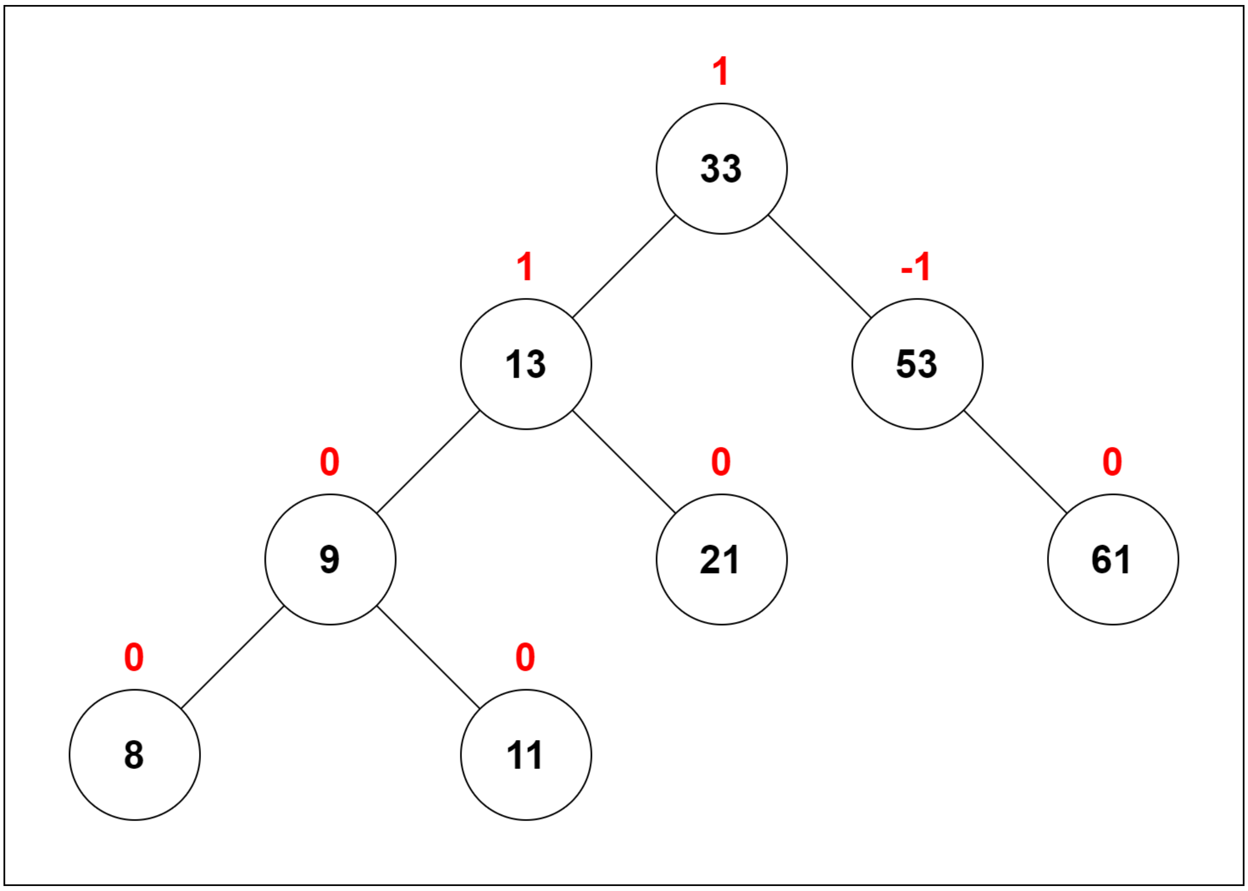


6. If, after insertion and updating the balance factors, you find that the tree is unbalanced (the balance factor is greater than 1 or less than -1), you need to rebalance it.
    - If the balance factor is greater than 1, it means the left subtree is taller. Depending on the relationship between 'newKey' and the key of the left child ('leftChildKey'), you perform either a right rotation or a left-right rotation.
        - If 'newKey' is less than 'leftChildKey', you do a right rotation.
        - Otherwise, you do a left-right rotation.
    - If the balance factor is less than -1, it means the right subtree is taller. Again, depending on the relationship between 'newKey' and the key of the right child ('rightChildKey'), you perform either a left rotation or a right-left rotation.
      - If 'newKey' is greater than 'rightChildKey', you do a left rotation.
      - Otherwise, you do a right-left rotation.





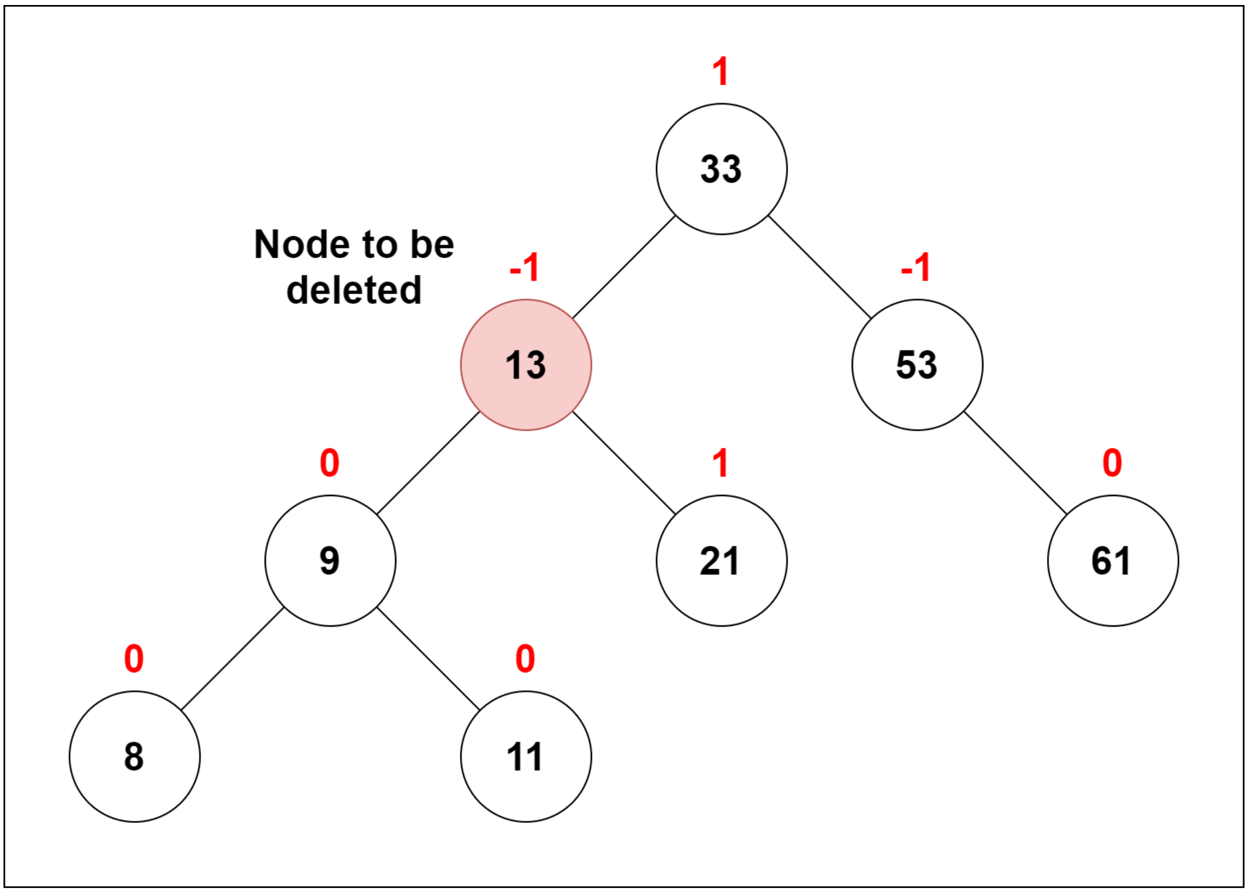
7. After performing the necessary rotations, your tree is now balanced with the new node inserted at the correct position.



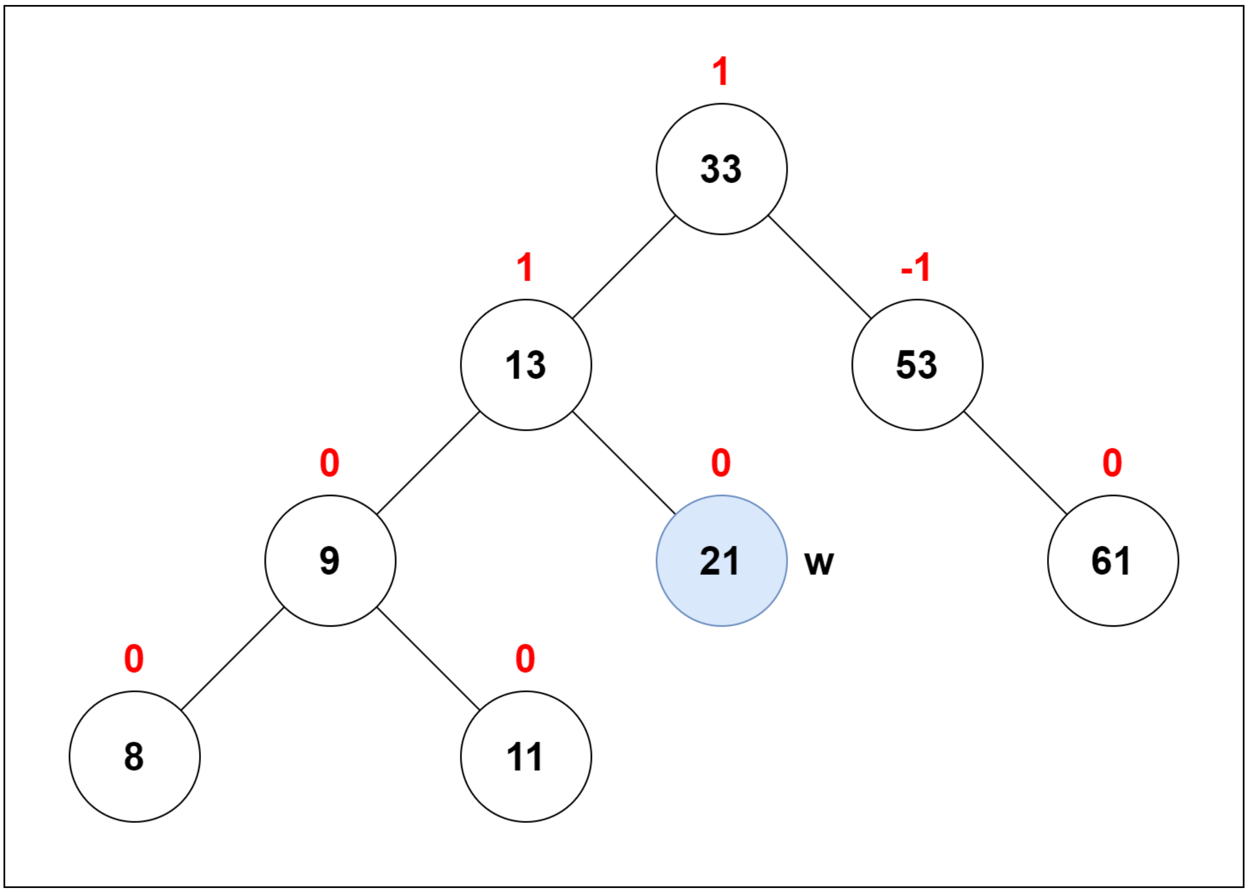
This algorithm ensures that the AVL tree remains balanced, which means that the height difference between the left and right subtrees of any node is at most 1. Balancing the tree is crucial for maintaining efficient search, insertion, and deletion operations with a time complexity of O(log n).

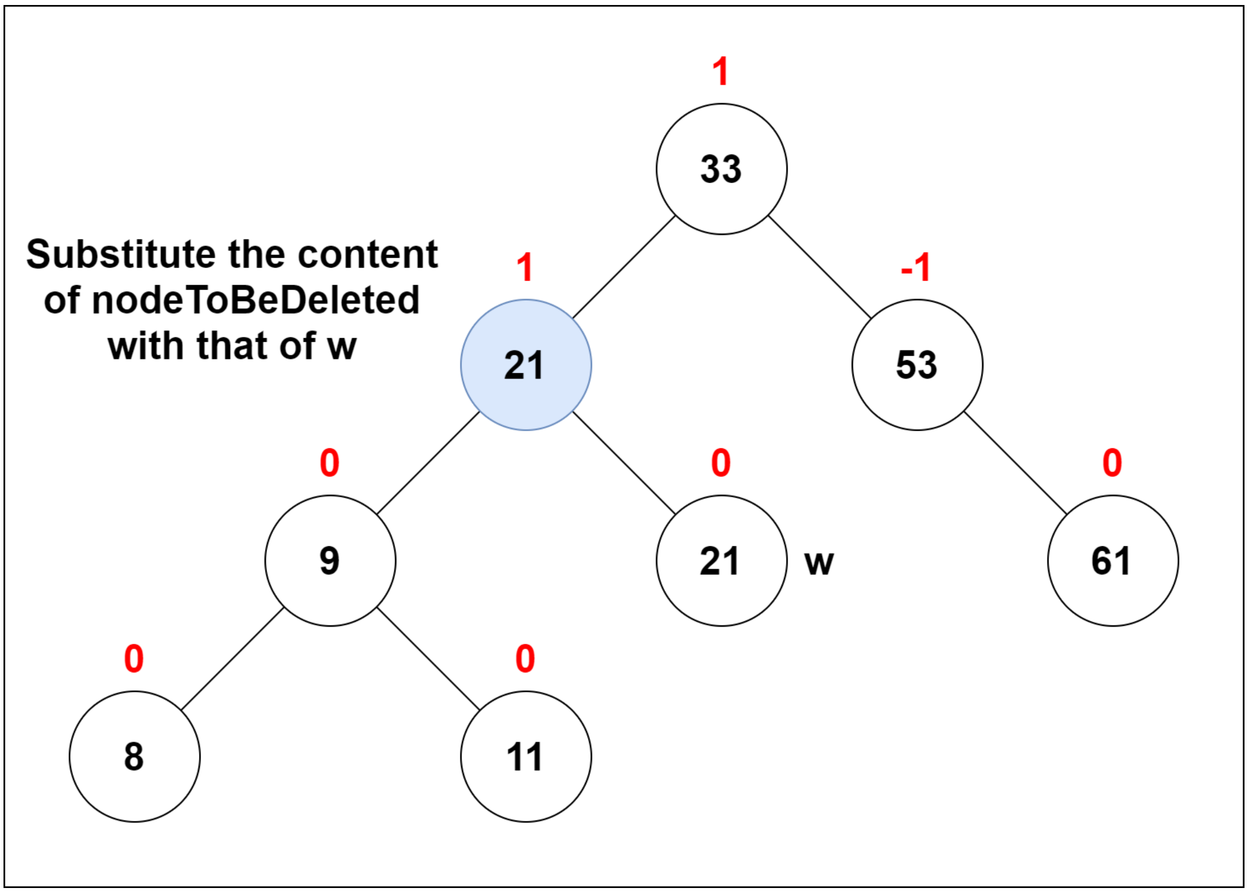
Algorithm to Delete a node

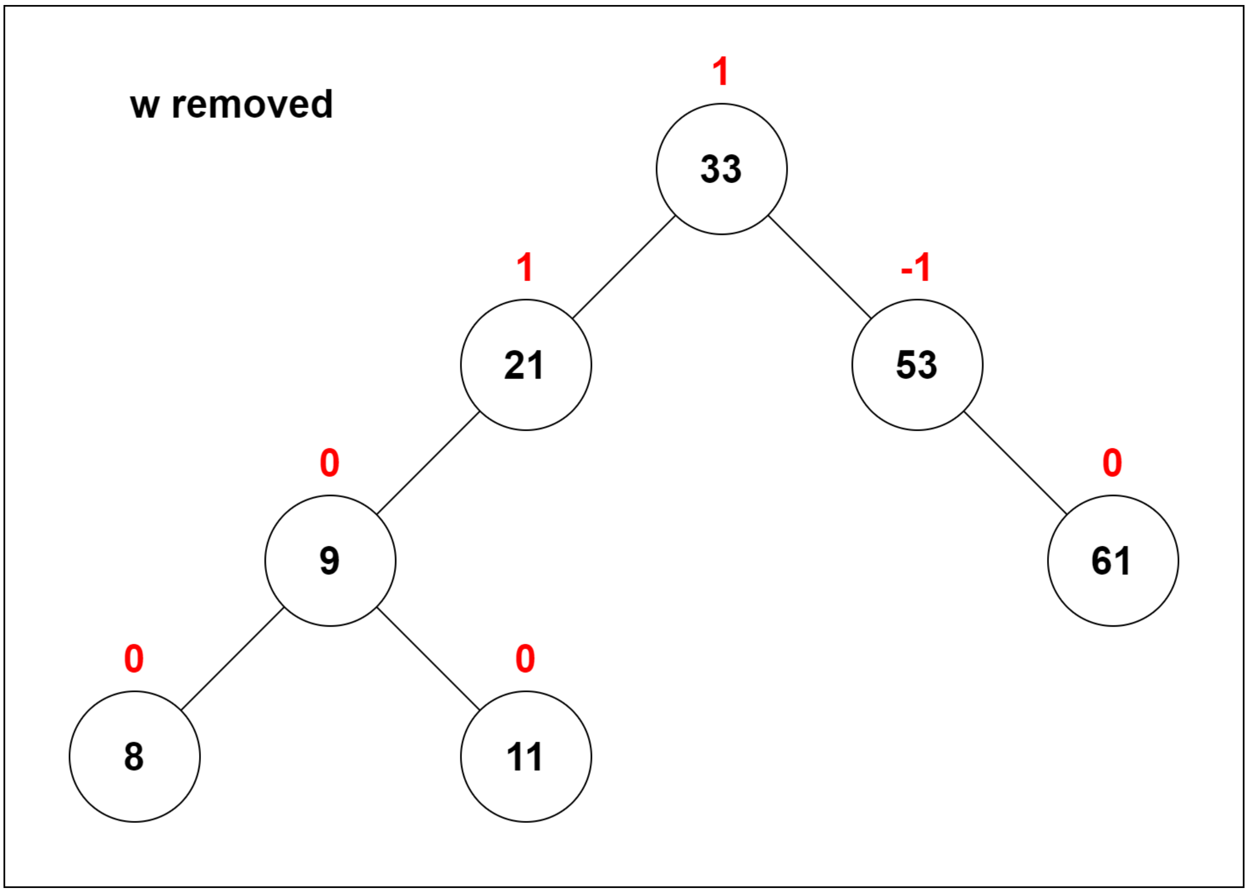
let's explain the algorithm for deleting a node in an AVL tree step by step:

1. You start by locating the node you want to delete, which we'll call 'nodeToBeDeleted'. This is done recursively in the code.


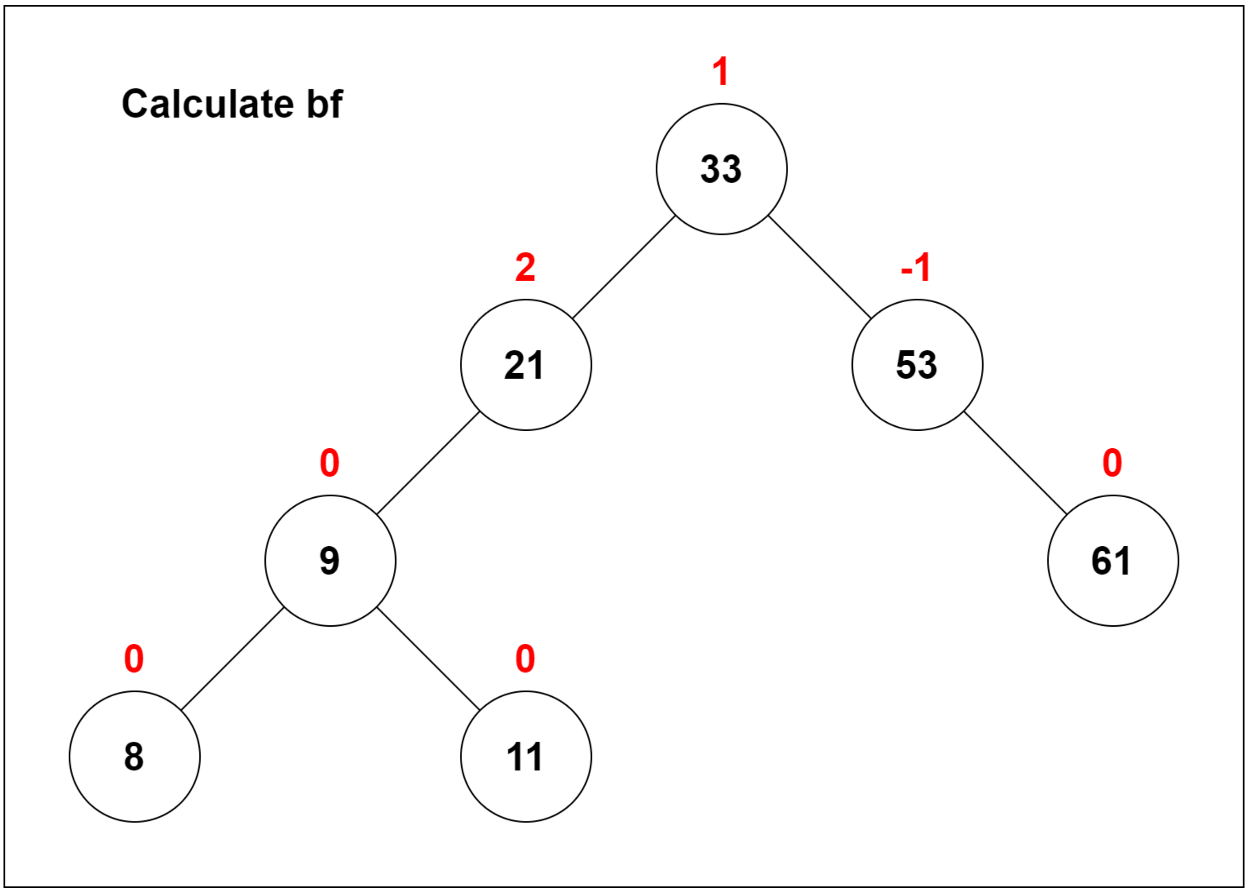
2. There are three cases for deleting a node:
- If 'nodeToBeDeleted' is a leaf node (meaning it has no children), you can simply remove it from the tree.
- If 'nodeToBeDeleted' has one child, you replace 'nodeToBeDeleted' with its child, and then remove the child.
- If 'nodeToBeDeleted' has two children, you need to find the inorder successor of 'nodeToBeDeleted'. The inorder successor is the node with the minimum key value in the right subtree of 'nodeToBeDeleted'. This is done to maintain the ordering of the tree.


1
33
1
13
-1
53
0
9
0
21
w
0
61
0
8
0
11


Substitute the content of nodeToBeDeleted with that of w
1
33
1
21
-1
53
0
9
0
21
w
0
61
0
8
0
11

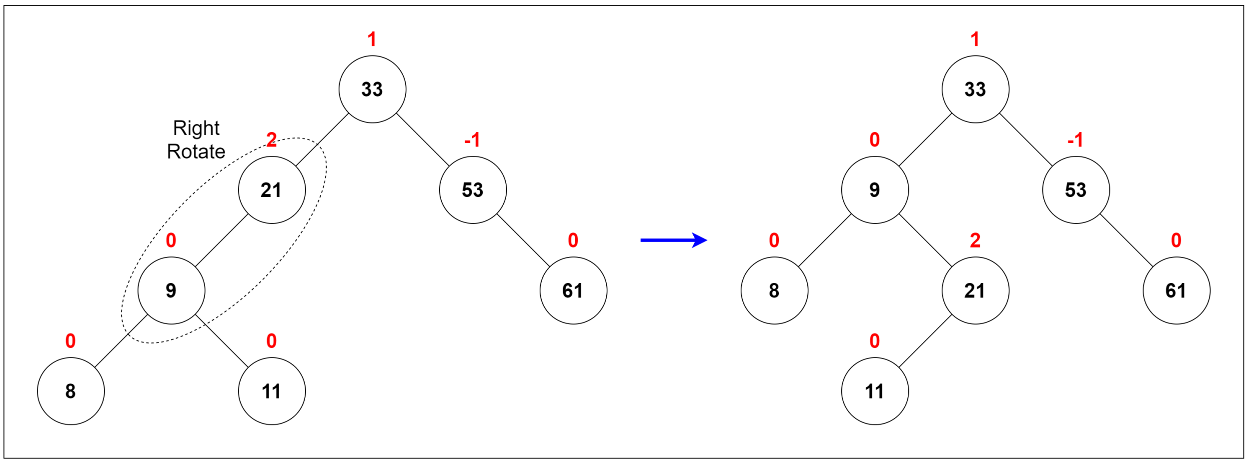


3. After deleting the node, you need to update the balance factor of the affected nodes.

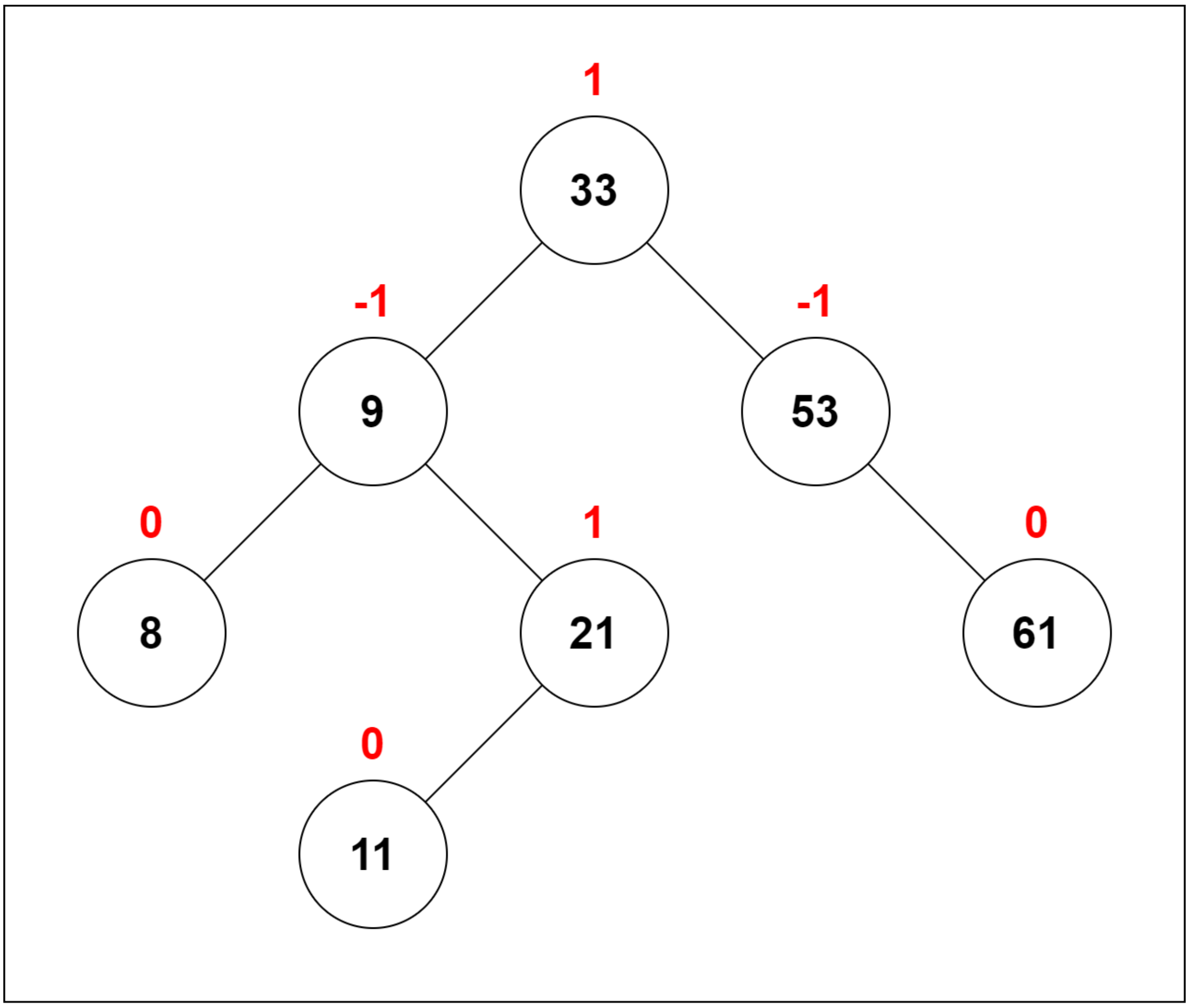


4. If the balance factor of any node is not -1, 0, or 1 (indicating an imbalance), you need to rebalance the tree.
   - If the balance factor of the current node is greater than 1, it means the left subtree is taller.
   - If the balance factor of the left child is greater than or equal to 0, you perform a right

rotation.
- Otherwise, you perform a left-right rotation.
- If the balance factor of the current node is less than -1, it means the right subtree is taller.
- If the balance factor of the right child is less than or equal to 0, you perform a left rotation.
- Otherwise, you perform a right-left rotation.



5. After performing the necessary rotations, your AVL tree is rebalanced, and the deleted node is successfully removed.



This algorithm ensures that the AVL tree remains balanced after a node deletion, which is crucial for maintaining the efficient search, insertion, and deletion operations of the tree.

C++ Example

```cpp
// AVL tree implementation in C++

#include <iostream>
using namespace std;

class Node {
   public:
  int key;
  Node *left;
  Node *right;
  int height;
};

int max(int a, int b);

// Calculate height
int height(Node *N) {
  if (N == NULL)
    return 0;
  return N->height;
}

int max(int a, int b) {
  return (a > b) ? a : b;
}

// New node creation
Node *newNode(int key) {
  Node *node = new Node();
  node->key = key;
  node->left = NULL;
  node->right = NULL;
  node->height = 1;
  return (node);
}

// Rotate right
Node *rightRotate(Node *y) {
  Node *x = y->left;
  Node *T2 = x->right;
  x->right = y;
  y->left = T2;
  y->height = max(height(y->left),
          height(y->right)) +
        1;
  x->height = max(height(x->left),
          height(x->right)) +
        1;
  return x;
}

// Rotate left
Node *leftRotate(Node *x) {
  Node *y = x->right;
  Node *T2 = y->left;
  y->left = x;
  x->right = T2;
  x->height = max(height(x->left),
          height(x->right)) +
        1;
  y->height = max(height(y->left),
          height(y->right)) +
        1;
  return y;
}

// Get the balance factor of each node
int getBalanceFactor(Node *N) {
  if (N == NULL)
    return 0;
  return height(N->left) -
       height(N->right);
}

// Insert a node
Node *insertNode(Node *node, int key) {
  // Find the correct postion and insert the node
  if (node == NULL)
    return (newNode(key));
  if (key < node->key)
    node->left = insertNode(node->left, key);
  else if (key > node->key)
    node->right = insertNode(node->right, key);
  else
    return node;

  // Update the balance factor of each node and
  // balance the tree
  node->height = 1 + max(height(node->left),
             height(node->right));
  int balanceFactor = getBalanceFactor(node);
  if (balanceFactor > 1) {
    if (key < node->left->key) {
      return rightRotate(node);
    } else if (key > node->left->key) {
      node->left = leftRotate(node->left);
      return rightRotate(node);
    }
  }
  if (balanceFactor < -1) {
    if (key > node->right->key) {
      return leftRotate(node);
    } else if (key < node->right->key) {
      node->right = rightRotate(node->right);
      return leftRotate(node);
    }
  }
  return node;
}

// Node with minimum value
Node *nodeWithMimumValue(Node *node) {
  Node *current = node;
  while (current->left != NULL)
    current = current->left;
  return current;
}

// Delete a node
Node *deleteNode(Node *root, int key) {
  // Find the node and delete it
  if (root == NULL)
    return root;
  if (key < root->key)
    root->left = deleteNode(root->left, key);
  else if (key > root->key)
    root->right = deleteNode(root->right, key);
  else {
    if ((root->left == NULL) ||
      (root->right == NULL)) {
      Node *temp = root->left ? root->left : root->right;
      if (temp == NULL) {
        temp = root;
        root = NULL;
      } else
        *root = *temp;
      free(temp);
    } else {
      Node *temp = nodeWithMimumValue(root->right);
      root->key = temp->key;
      root->right = deleteNode(root->right,
                   temp->key);
    }
```

```cpp
    }

    if (root == NULL)
      return root;

    // Update the balance factor of each node and
    // balance the tree
    root->height = 1 + max(height(root->left),
                 height(root->right));
    int balanceFactor = getBalanceFactor(root);
    if (balanceFactor > 1) {
      if (getBalanceFactor(root->left) >= 0) {
        return rightRotate(root);
      } else {
        root->left = leftRotate(root->left);
        return rightRotate(root);
      }
    }
    if (balanceFactor < -1) {
      if (getBalanceFactor(root->right) <= 0) {
        return leftRotate(root);
      } else {
        root->right = rightRotate(root->right);
        return leftRotate(root);
      }
    }
    return root;
  }

  // Print the tree
  void printTree(Node *root, string indent, bool last) {
    if (root != nullptr) {
      cout << indent;
      if (last) {
        cout << "R----";
        indent += "   ";
      } else {
        cout << "L----";
        indent += "|  ";
      }
      cout << root->key << endl;
      printTree(root->left, indent, false);
      printTree(root->right, indent, true);
    }
  }

  int main() {
    Node *root = NULL;
    root = insertNode(root, 33);
    root = insertNode(root, 13);
    root = insertNode(root, 53);
    root = insertNode(root, 9);
    root = insertNode(root, 21);
    root = insertNode(root, 61);
    root = insertNode(root, 8);
    root = insertNode(root, 11);
    printTree(root, "", true);
    root = deleteNode(root, 13);
    cout << "After deleting " << endl;
    printTree(root, "", true);
  }
```

Complexities of Different Operations on an AVL Tree

Let's talk about the complexities of different operations on an AVL tree:

1. **Insertion:** When you insert a new node into an AVL tree, it takes O(log n) time. This means that the time it takes to insert a new node grows logarithmically with the number of nodes in the tree. This is because AVL trees are self-balancing, and the tree structure is adjusted to maintain balance during insertion.
2. **Deletion:** Deleting a node from an AVL tree also takes O(log n) time. Just like with insertion, the time it takes to delete a node grows logarithmically with the number of nodes in the tree. This is because, during deletion, the tree structure is adjusted to ensure it remains balanced.
3. **Search:** When you search for a node in an AVL tree, it also takes O(log n) time. The logarithmic time complexity ensures that searching for a specific node is quite efficient, even as the tree grows in size.

Insertion	Deletion	Search
O(log n)	O(log n)	O(log n)

So, whether you're inserting, deleting, or searching for a node in an AVL tree, you can expect these operations to be efficient with a time complexity of O(log n), where 'n' is the number of nodes in the tree. This is one of the key advantages of using AVL trees for tasks that involve these operations.

AVL Tree Applications

Let's explore some applications of AVL trees:

1. **Indexing Large Records in Databases:** AVL trees are incredibly useful in databases, especially when you have a large amount of data to organize and search through. They help in creating efficient indexes, making it much quicker to locate specific records within the database. This is crucial for tasks like data retrieval and database management.
2. **Searching in Large Databases:** When you need to search for specific data in a large database, AVL trees can significantly speed up the process. Their self-balancing property ensures that search operations, even in massive datasets, have a predictable and efficient time complexity. This makes AVL trees a go-to choice for implementing search algorithms in databases.

In essence, AVL trees play a vital role in optimizing data storage and retrieval in various database applications, where speed and efficiency are of utmost importance.