**Binary Search Tree**

Table of Contents

[Binary Search Tree Operations 7](#_Toc66036495)

[Searching for a Node 7](#_Toc66036496)

[Finding the Minimum Value 8](#_Toc66036497)

[Successors, Predecessors, Ancestors and Descendants 9](#_Toc66036498)

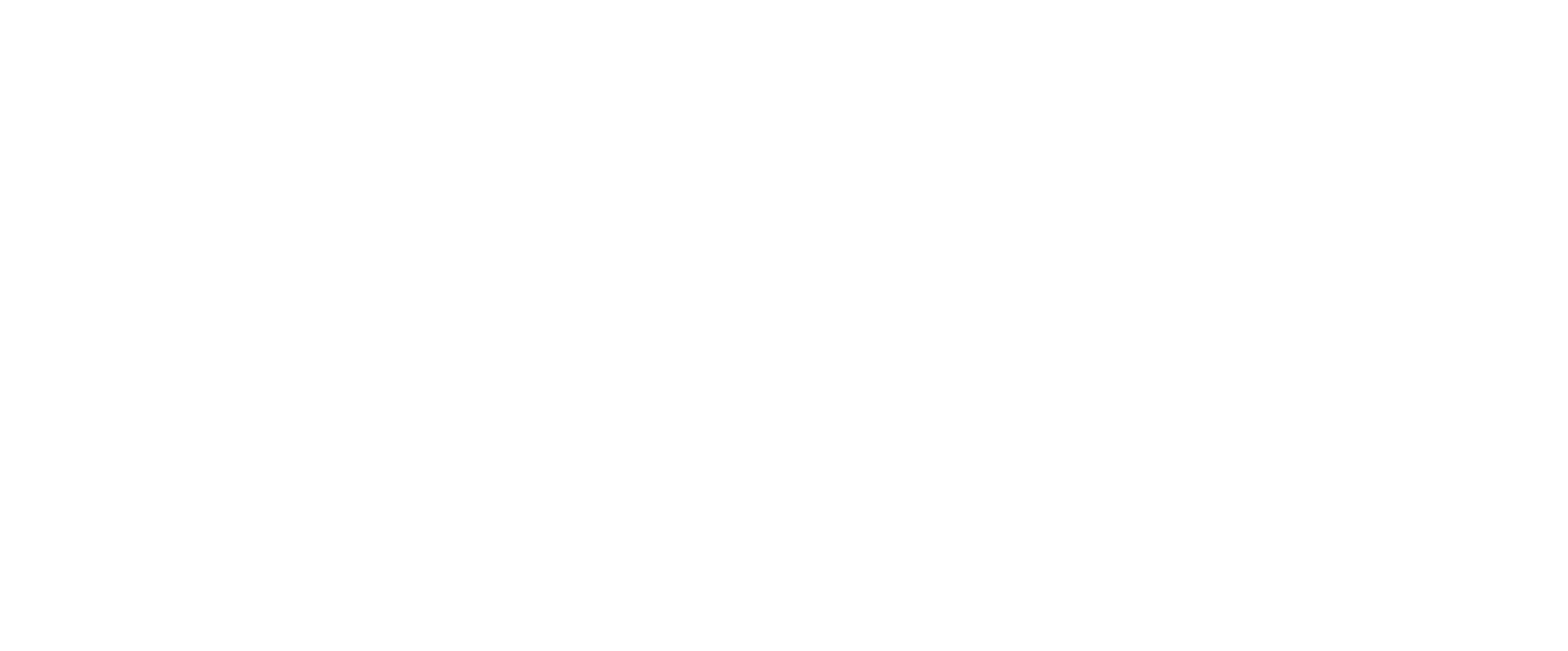
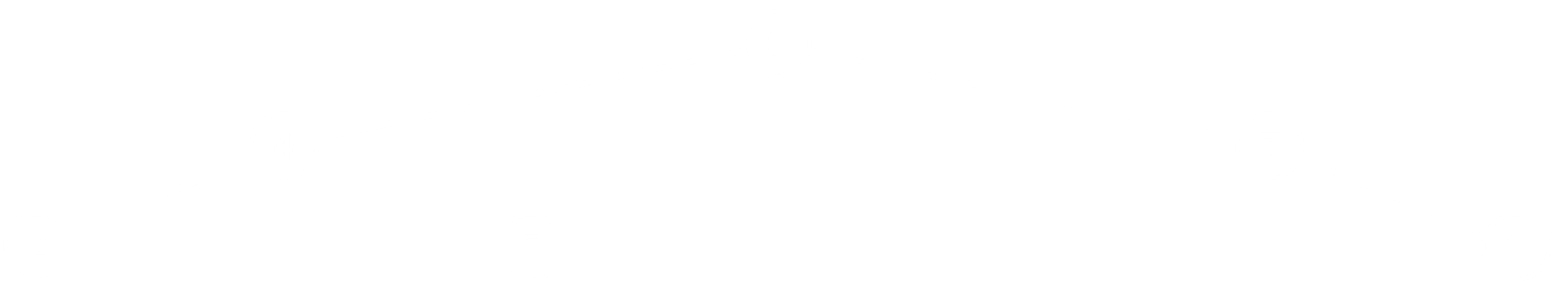
[Finding the Successor 9](#_Toc66036499)

[Inserting a Node 11](#_Toc66036500)

[Node Deletion 13](#_Toc66036501)

[Balanced Binary Search Trees (AVL Trees) 16](#_Toc66036502)

We previously looked at binary trees while studying heaps. There, the trees were nearly complete, and nicely arranged with each node going from left to right on the same level. This made it possible to store those trees inside arrays. However, binary trees can also look like these:



These are very difficult to store in arrays. They contain nodes in arbitrary positions, with some having both children, some having only the right child, etc. We can store such a tree like this:

struct node  
{  
 int value;  
 struct node\* leftChild;  
 struct node\* rightChild;  
 struct node\* parent;  
};

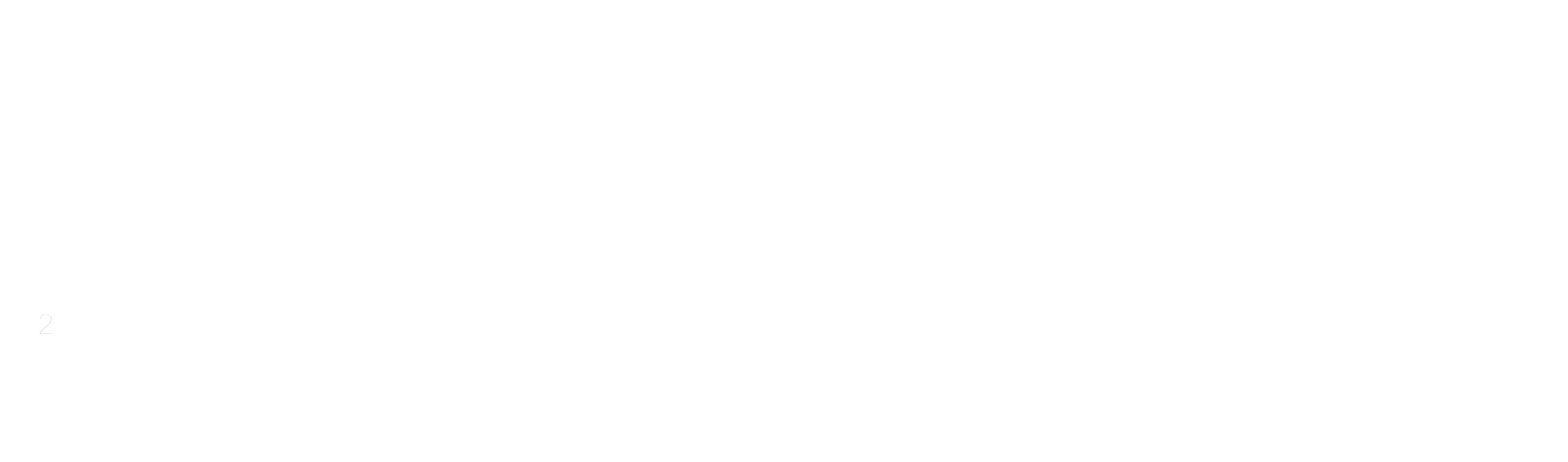
C++

Using this structure, we could practically store anything. At every node, we are storing pointers to the parent and left and right children. From the left tree in the image above, the node 6 points left to 4, right to 7 and parent to **NULL**. 4 on the other hand, points parent to 6, left to 2 and right to 5.

Sometimes however, we may have situations where each node has multiple children. This presents two problems. First, storing multiple children for each node means a lot of pointers, which becomes messy. Second, not every node may have the same number of children, but every node will have those pointers. For nodes with fewer children, the pointers are wasted memory. To solve these issues, we can do something differently. At each node, we could store the parent, and just the left-most child. Each node would also point to its immediate right sibling, so that a parent node must just go to the left-most node and can then continue from there going to each of the immediate right siblings in turn. This algorithm of course, assumes that the nodes will be arranged from left to right.

We will now go back to binary search trees, the ones with just two children. We will look into algorithms that will allow us to insert a node, remove a node, or search for a node, very fast, specifically in O(h).

In order to do this, we will arrange each node such that it has a left child that is an entire branch, and a right child that is another entire branch. Every node in the left branch will have a value that is lower than or equal to the parent’s value, and every node in the right branch will have a value that is higher than the parent’s value. It looks a little like this:



Consider 15. Every node to its left is smaller, and every node to its right is larger. Similarly, for 6, every value to its left is smaller, and every value to its right is larger. This type of data structure is non-linear, and this one in particular is called a binary search tree, due to its binary search capabilities. Note that if we wanted to insert, remove or look for any node, we would just have to start at the root and check if what we were looking for was equal, smaller or greater, and move accordingly. Since even in the worst case we would need to go from the root to one of the leaves, we would need to perform at most h searches, where h is the height of the tree. Thus, the algorithm has a time complexity of O(h).

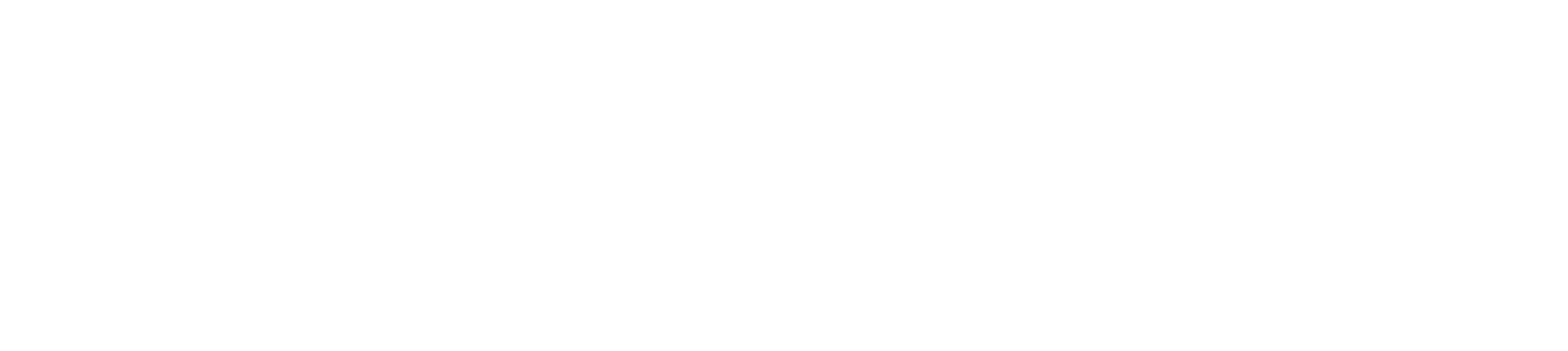
Notice that the order in which the tree given is set up depends purely on the order in which inputs were received. We got 15 first, then 6, which we put as its left child. If we had received 13 before 6 instead, the left child would be 13, which would change the entire tree.

We now need to consider how we can print the tree. We could do this in three ways, by using pre order traversal, in order traversal or post order traversal. In pre order traversal, for every node, we first print the parent, then go to the left branch and traverse that, then the right branch. For the tree above, we would print 15, go to 6, print 6, go to 3, print 3, go to 2, print 2, then go to 4, print 4, then go to 7 and so on. In in order traversal, we first traverse the left child, then print the node, then traverse the right child. In post order traversal, we first traverse the left child, then traverse the right child, and then print the node. We will only look into in order traversal in detail since we will be needing that. The code for in order traversal should look something like this:

void inorder (Node\* x)  
 if (x != NULL)  
 inorder(x->leftChild);  
 cout<<x->value<<" ";  
 inorder(x->rightChild);

PSEUDOCODE

We went to some node x, called the traversal algorithm on its left child, then printed the value of x and then called the traversal algorithm on the right child. When we try to call the algorithm on a child that has a **NULL** value, the if statement stops that recursion and goes back to the previous recursion. Notice that by changing the order of the statements we can create pre order and post order traversal algorithms.

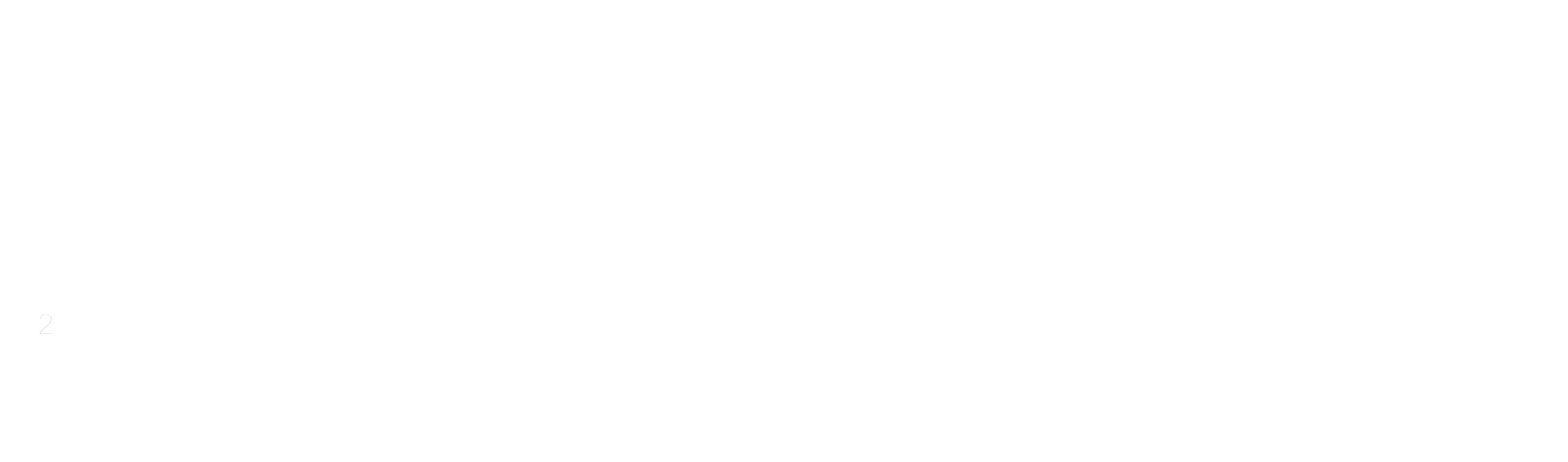


The following table shows the series of events that would take place if we used this algorithm on the root of the tree we gave above.

|  |  |  |
| --- | --- | --- |
| Step | Event | Current Stack |
| inorder(6) | Go to node 6 | |  |  |  |  | | --- | --- | --- | --- | | IO(6) |  |  |  | |
| inorder(4) | Go to left child of node 6 (node 4) | |  |  |  |  | | --- | --- | --- | --- | | IO(6) | IO(4) |  |  | |
| inorder(2) | Go to left child of node 4 (node 2) | |  |  |  |  | | --- | --- | --- | --- | | IO(6) | IO(4) | IO(2) |  | |
| inorder(**NULL**) | Go to left child of node 2 (**NULL**) | |  |  |  |  | | --- | --- | --- | --- | | IO(6) | IO(4) | IO(2) | IO(**NULL**) | |
| print(2) | Left child of node 2 traversal complete | |  |  |  |  | | --- | --- | --- | --- | | IO(6) | IO(4) | IO(2) |  | |
| inorder(**NULL**) | Go to right child of node 2 (**NULL**) | |  |  |  |  | | --- | --- | --- | --- | | IO(6) | IO(4) | IO(2) | IO(**NULL**) | |
| print(4) | Node 2 traversal complete | |  |  |  |  | | --- | --- | --- | --- | | IO(6) | IO(4) |  |  | |
| inorder(5) | Go to right child of node 4 (node 5) | |  |  |  |  | | --- | --- | --- | --- | | IO(6) | IO(4) | IO(5) |  | |
| inorder(**NULL**) | Go to left child of node 5 | |  |  |  |  | | --- | --- | --- | --- | | IO(6) | IO(4) | IO(5) | IO(**NULL**) | |
| print(5) | Left child of node 5 traversal complete | |  |  |  |  | | --- | --- | --- | --- | | IO(6) | IO(4) | IO(5) |  | |
| inorder(**NULL**) | Go to right child of node 5 | |  |  |  |  | | --- | --- | --- | --- | | IO(6) | IO(4) | IO(5) | IO(**NULL**) | |
| print(6) | Node 5, 4 traversals complete | |  |  |  |  | | --- | --- | --- | --- | | IO(6) |  |  |  | |
| inorder(7) | Go to right child of node 6 (node 7) | |  |  |  |  | | --- | --- | --- | --- | | IO(6) | IO(7) |  |  | |
| inorder(**NULL**) | Go to left child of node 7 (**NULL**) | |  |  |  |  | | --- | --- | --- | --- | | IO(6) | IO(7) | IO(**NULL**) |  | |
| print(7) | Left child of node 7 traversal complete | |  |  |  |  | | --- | --- | --- | --- | | IO(6) | IO(7) |  |  | |
| inorder(8) | Go to right child of node 7 (node 8) | |  |  |  |  | | --- | --- | --- | --- | | IO(6) | IO(7) | IO(8) |  | |
| inorder(**NULL**) | Go to left child of node 8 (**NULL**) | |  |  |  |  | | --- | --- | --- | --- | | IO(6) | IO(7) | IO(8) | IO(**NULL**) | |
| print(8) | Left child of node 8 traversal complete | |  |  |  |  | | --- | --- | --- | --- | | IO(6) | IO(7) | IO(8) |  | |
| inorder(**NULL**) | Go to right child of node 8 (**NULL**) | |  |  |  |  | | --- | --- | --- | --- | | IO(6) | IO(7) | IO(8) | IO(**NULL**) | |
|  | Node 8, 7, 6 traversals complete | |  |  |  |  | | --- | --- | --- | --- | |  |  |  |  | |

Notice the order in which we ended up printing the nodes, 2, 4, 5, 6, 7, 8. Using an in-order traversal algorithm, we can create a sorted list, given that we already have a created binary search tree.

## Binary Search Tree Operations



### Searching for a Node

We start our search at the root. We compare the value of the key we are looking for against the value of the root. We know that at every node, the branch from the left child contains only those values that are smaller and the branch from the right child contains only those values that are larger. Thus, if we find that our key is smaller than the value of the current node, we go to the left child of that node, and if we find that our key is larger than the value of the current node, we go to the right child of that node. We repeatedly keep doing this until we find the node we are looking for, or if we reach a **NULL** value, which indicates that we reached the end of the tree without finding our key, meaning it is not present in the tree.

Node\* searchTree (Node\* x, int key)  
{  
 if (x == **NULL** || key == x->value) return x;  
 else if (key < x->value) return searchTree (x->leftChild, key);  
 else if (key > x->value) return searchTree (x->rightChild, key);  
}

C++

Notice that, in the worst case, with a tree of height h, we will have to call the function h times, when the node we want is at the lowest level of the tree. Thus, the worst-case time complexity of this function is O(h).

Instead of using the function above in a recursive manner, we could also have done it in an iterative manner like this:

Node\* searchTree (Node\* x, int key)  
{  
 while (x->value != key && x != **NULL**)  
 {  
 if (key > x->value) x = x->rightChild;  
 else if (key < x->value) x = x->leftChild;  
 }  
 return x;  
}

C++

### Finding the Minimum Value

We know that for every node, the nodes with a smaller value are stored in the left branch. Thus, if we just keep following the left branch until we find a left branch that has a NULL value, we will find the minimum value present in the tree.

Node\* findMinimum (Node\* x)  
{  
 while (x->leftChild != **NULL**) x = x->leftChild;  
 return x;  
}

C++

We can create a function to find the maximum value in a similar manner. We began this search at the root and had to go to the lowest level, so the time complexity is again O(h).

### Successors, Predecessors, Ancestors and Descendants

The successor of a particular node is the node that comes after it in a particular traversal algorithm. For example, if we traversed the tree given above using in order traversal, the successor of 13 is 15, since we know that in order traversal gives us values in ascending order. The predecessor of a particular node is the node that comes before it in a particular traversal algorithm. Again, for the tree given above being traversed using in order traversal, the predecessor of 13 is 9.

Ancestors are all nodes that are ‘above’ that particular node. For 13, ancestors are 7, 6 and 15. Descendants are all nodes that are ‘below’ that particular node. For 13, the descendant is just 9.

### Finding the Successor

To find the successor of a particular node, we first need to find the node itself. This is because we do not directly have access to any node other than the root. We can find the node using the searchTree function we defined above, which will return the node to us so that we can perform further operations on it. This will also allow us to ensure that the node for which we are trying to find the successor actually exists in the tree.

Notice that the successor for any node is going to be the minimum value that is greater than that node. Well, since all values greater than the node are in its right branch, we can just get the minimum value from that branch using the findMinimum function. However, we face a problem there. What if the node has no right branch at all, like 13 in the tree above? In that scenario, we will find that the right branch is **NULL**. If it had any right branch, that would have to be the next largest value. We will start looking upwards towards the ancestors of the node now, since it is possible that the node we are looking at is part of the left branch of some ancestor node, meaning that ancestor node has the next largest value. We know that that node is the next largest for certain, since any nodes that are larger than that one will be part of the right branch of that node, not the left branch where we are. We keeping going up, and if we fail to find any nodes that are larger, we will reach the root, at which point we will find that the next parent is **NULL**. This is where we conclude that there is no larger node.

Node\* findSuccessor (Node\* root, int key)  
{  
 Node\* x = searchTree(root, key); *//getting the node we want* if (x->rightChild != **NULL**) return findMinimum(x->rightChild);  
 else  
 {  
 Node\* parent = x->parent;  
 while (parent != **NULL** && x->value > parent->value)  
 parent = parent->parent;  
 return parent;  
 }  
}

C++

Again, in the worst case, we had to travel all the way back to the root of the tree from the lowest level, which means the worst-case time complexity is O(h).

### Inserting a Node

Every function we have looked at till now has been for a tree that has already been created. We will now look at how to actually create the tree by inserting nodes. The first step here is to make a function that creates nodes, like we did for linked lists. If our tree is a binary search tree, it could look something like this:

Node\* createNode(int value)  
{  
 Node\* newNode = new Node();  
 newNode->value = value;  
 newNode->parent = **NULL**;  
 newNode->leftChild = **NULL**;  
 newNode->rightChild = **NULL**;  
}

C++

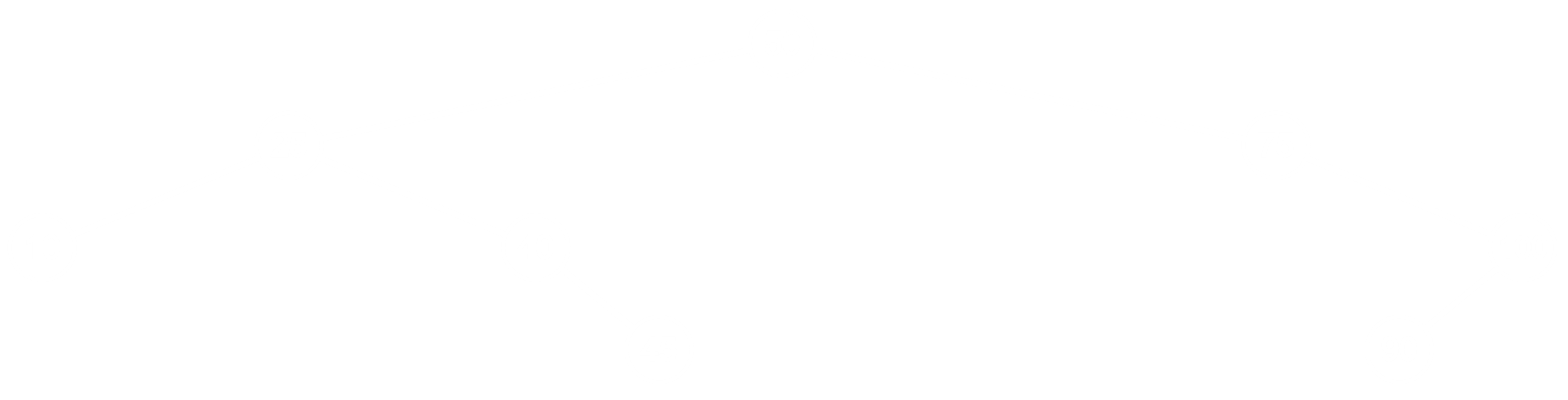
To insert a node, we start from the root and keep checking if the node we are inserting is smaller or larger. If it is smaller or equal, we go left, and if it is larger, we go right. Once we find a **NULL** value, we have reached a point where we can insert the node. However, there we have to keep track of the parent of the node we are on right now as well, since we will be unable to reach the parent again to link the new node once we have reached a **NULL** value. We also need to check the condition where we have no root at all yet, and return the new node to the main function if this occurs, so that we can use the root for further operations.

Node\* insertNode (Node\* root, int value)  
{  
 Node\* cur = root;  
 Node\* newNode = createNode(value);  
 if (cur == **NULL**) return newNode;  
 else  
 {  
 Node\* parent;  
 while (cur != **NULL**)  
 {  
 parent = cur;  
 if (value > cur->value) cur = cur->rightChild;  
 else if (value <= cur->value) cur = cur->leftChild;  
 }  
  
 newNode->parent = parent;  
 if (value > parent->value) parent->rightChild = newNode;  
 else if (value <= parent->value) parent->leftChild = newNode;  
 return root;  
 }  
}

C++

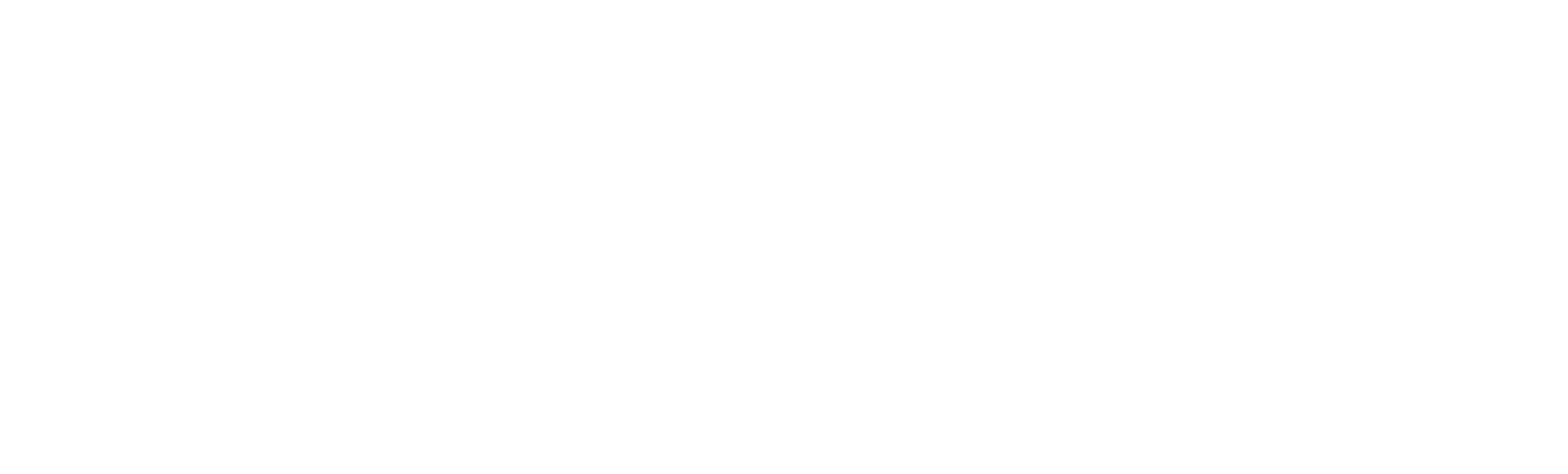
### Node Deletion

Say we have the following binary search tree:

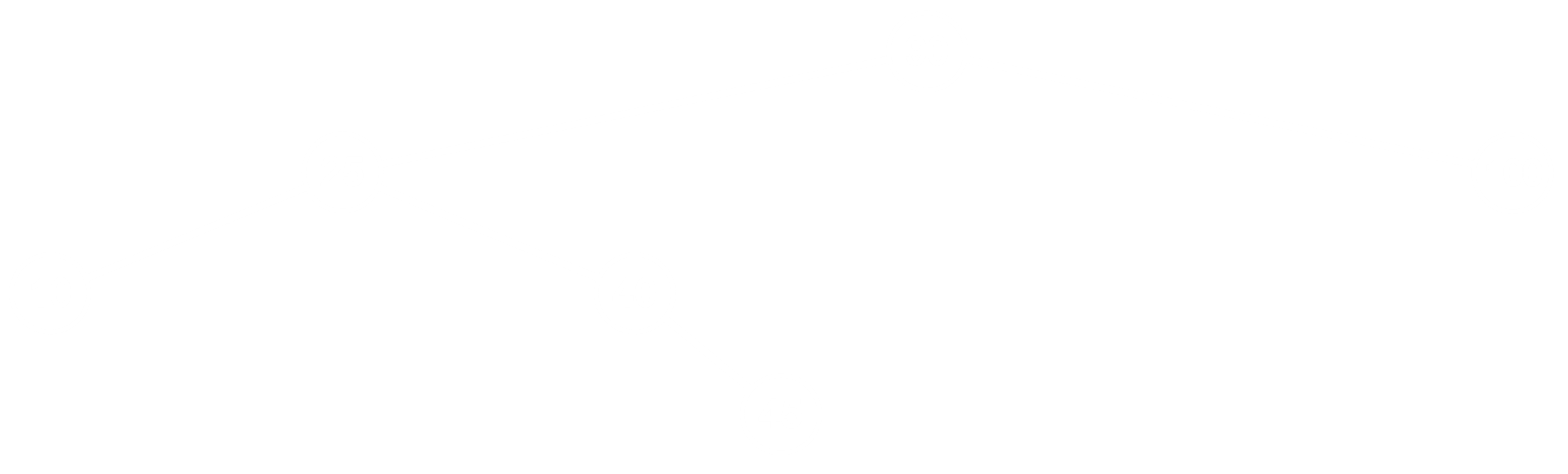


If we wanted to delete 90, we would first search the tree to make sure that node exists, then we would go to its parent, check if 90 is the left child or the right child, and set that child to **NULL** accordingly. Since 90 is a leaf node, the process for deletion is very simple.

Say we wanted to delete 75 instead. This is still fairly simple since 75 only has one child subtree, the one starting with 100. Thus, we can just set the respective child of the parent of 75 to 100 instead. The properties of a binary search tree are still maintained, and we get a tree that looks like this:



Things get a little more complicated if we want to delete a node that has two children though. Say we want to delete 50. There are two ways we can do this without violating the conditions of a binary tree. First, we could get the successor of 50, i.e. the smallest value from its right subtree (which is 90). Doing so would result in a node that still has all values larger than 90 on the right, since 90 was the smallest value from the right, and all values smaller than 90 on the left, since 90 was larger than 50 which means it is also larger than all values smaller than 50 (that are on the left). Alternatively, we could get the predecessor of 50, which is the largest value from the left subtree. This approach follows a similar thinking. Using the first approach, we get a tree like this:



The pseudocode for deleting a node is:

deleteNode(key)  
 temp = searchTree(key)  
 if (temp == **NULL**) *//error message* else if (both children of temp == **NULL**)  
 set corresponding child of parent to **NULL**  
 else if (one child of temp != **NULL**)  
 set corresponding child of parent to that child  
 else  
 replacement = findMinimum(temp->rightChild)  
 replace temp with replacement

PSEUDOCODE

In the actual code, we also need to check for the condition where the root is being deleted and change multiple pointers during replacement. All of this has been included in this code:

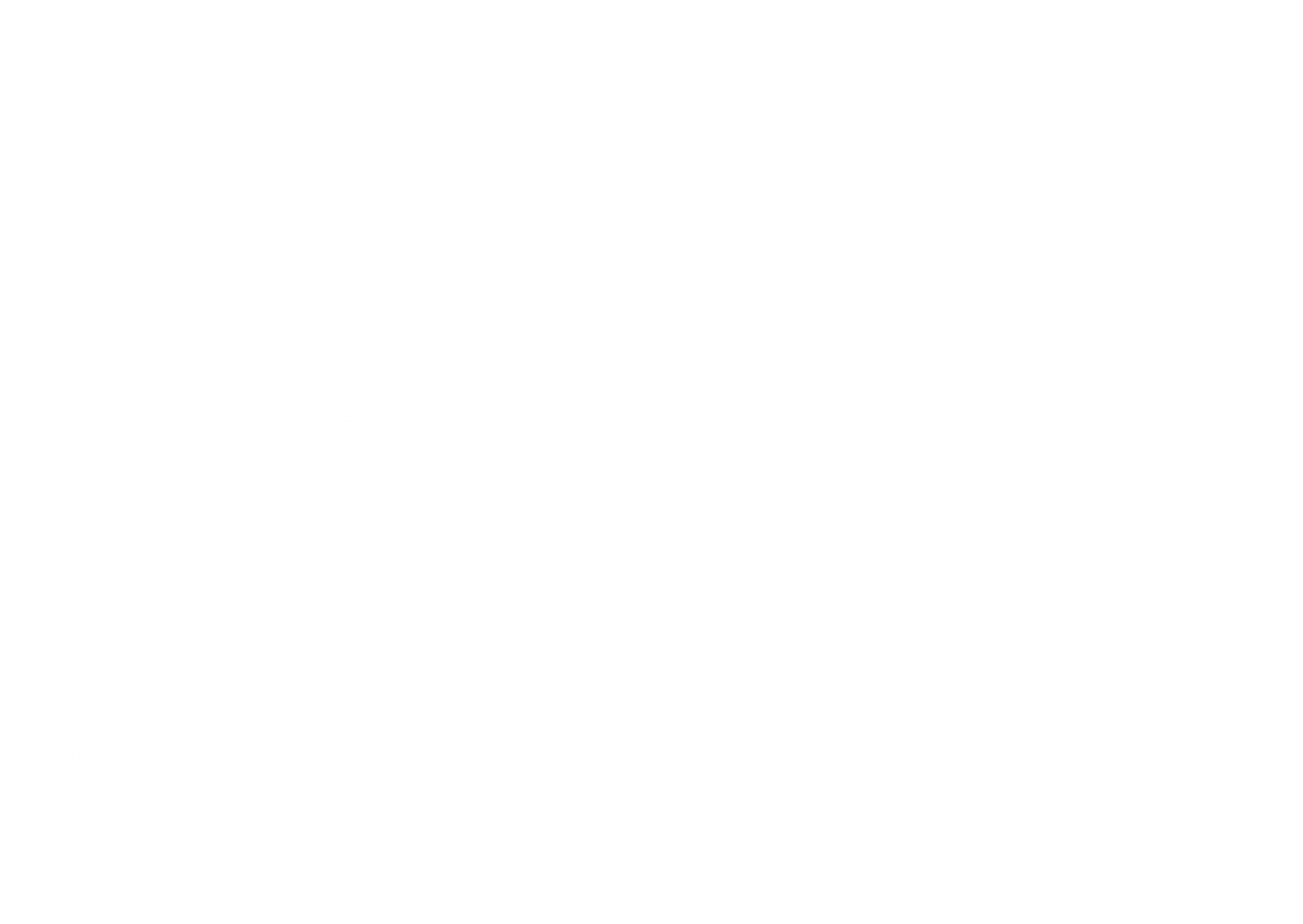
Node\* deleteNode(Node\* root, int key)  
{  
 Node\* temp = searchTree(root, key); *//finding node* if (temp == **NULL**) return root; *//if node not found, no change* if (temp->leftChild != **NULL** && temp->rightChild != **NULL**)  
 *//node has two children* {  
 Node\* replacement = findMinimum(temp->rightChild);  
 *//replacement is minimum value of right subtree* if (replacement->parent == temp) temp->rightChild = **NULL**;  
 *//if replacement parent is temp, right child of temp is NULL* else replacement->parent->leftChild = **NULL**;  
 *//else left child of parent is NULL* replacement->parent = temp->parent; *//changing replacement's pointers*  
 replacement->leftChild = temp->leftChild;  
 replacement->rightChild = temp->rightChild;  
 if (temp->parent == **NULL**) return replacement; *//if temp was root* else if (temp->parent->leftChild == temp) *//else change parent's child*  
 temp->parent->leftChild = replacement;  
 else temp->parent->rightChild = replacement;  
 }  
 else if (temp->leftChild != **NULL**)  
 *//if node does not have two children, but has left child* {  
 temp->leftChild->parent = temp->parent; *//changing left child's parent* if (temp->parent == **NULL**) return temp->leftChild;  
 *//if node was root, return left child  
 //else change parent's child* if (temp->parent->leftChild == temp)  
 temp->parent->leftChild = temp->leftChild;  
 else temp->parent->rightChild = temp->leftChild;  
 }  
 else *//if node has right child or no children* {  
 if (temp->rightChild != **NULL**) temp->rightChild->parent = temp->parent;  
 *//if right child exists, change parent* if (temp->parent == **NULL**) return temp->rightChild;  
 *//if node is root, return right child. This could be NULL.*

else if(temp->parent->leftChild == temp) *//else change parent's child*  
 temp->parent->leftChild = temp->rightChild;  
 else temp->parent->rightChild = temp->rightChild;  
 }  
 return root; *//root returned if root was untouched*}

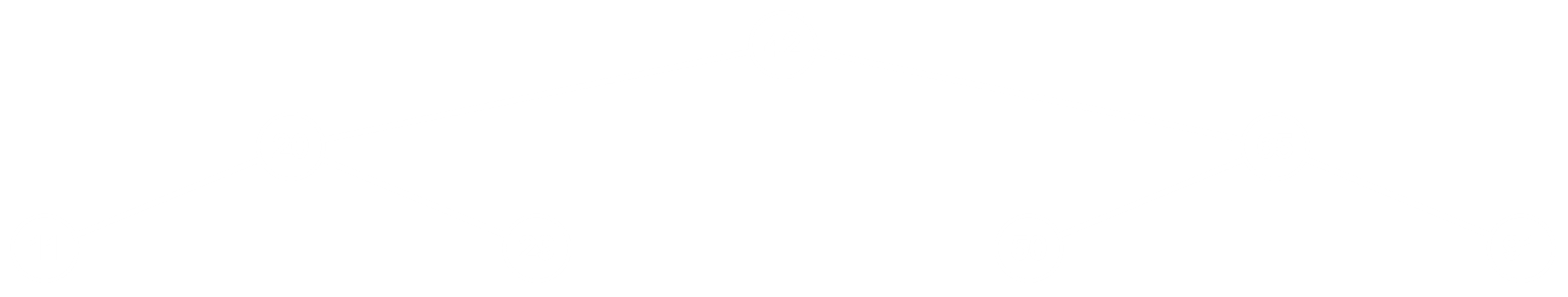
C++

## Balanced Binary Search Trees (AVL Trees)

For all the operations we have seen so far regarding binary search trees, the time complexity has been O(height). This is due to the fact that we do not know the order in which nodes are added to the tree beforehand. Thus, we may get trees that look like this:



This is still a binary search tree, but since nodes have been added in descending order, we get this structure. Cases like this prevent us from making full use of the functions we have created. The above tree could perform the functions with a time complexity of O(6), i.e. O(height). This tree is said to be unbalanced and skewed to the left. If we had a tree that was balanced, like the one below, we could perform our functions with a time complexity of O(log n), which is the best we can hope for.



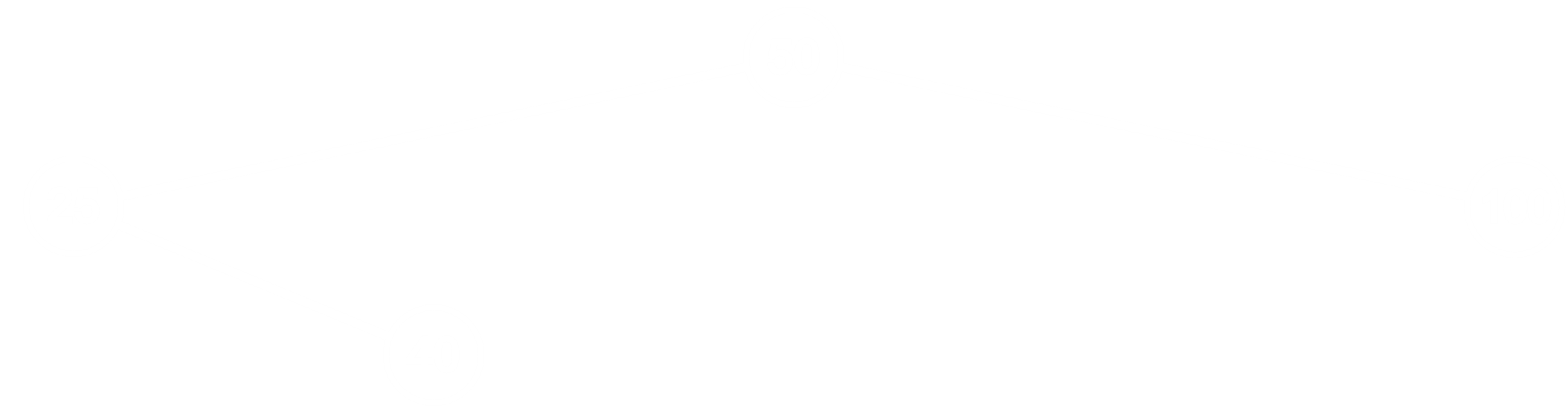
The difference in heights of the left and right children for any particular node is known as that nodes balance factor. For example, for the node 42, and , thus, the balance factor is . In general, balance factors of , or will be considered as balanced, while anything else will be considered imbalanced. Null values are given a height of , while leaf nodes have a height of .

We can find the height of any node using this function:

int height (Node \* x)  
{  
 if (x == **NULL**) return -1;  
 else if (height(x->leftChild) > height(x->rightChild))  
 return height(x->leftChild) + 1;  
 else return height(x->rightChild) + 1;  
}

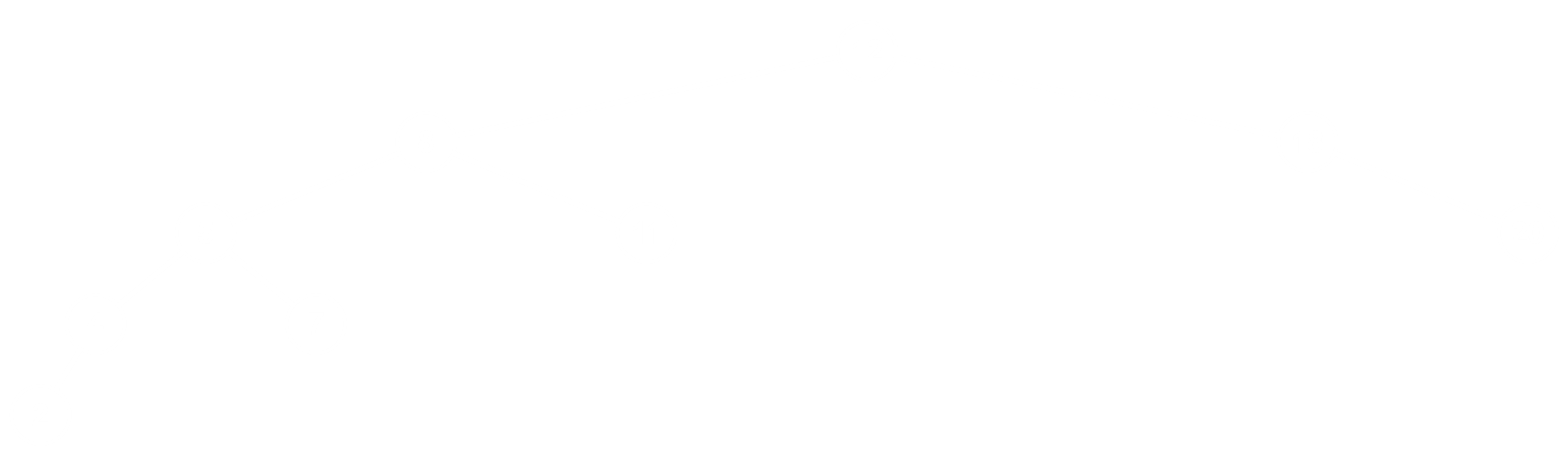
C++

As we add nodes, the height of its parents increases.



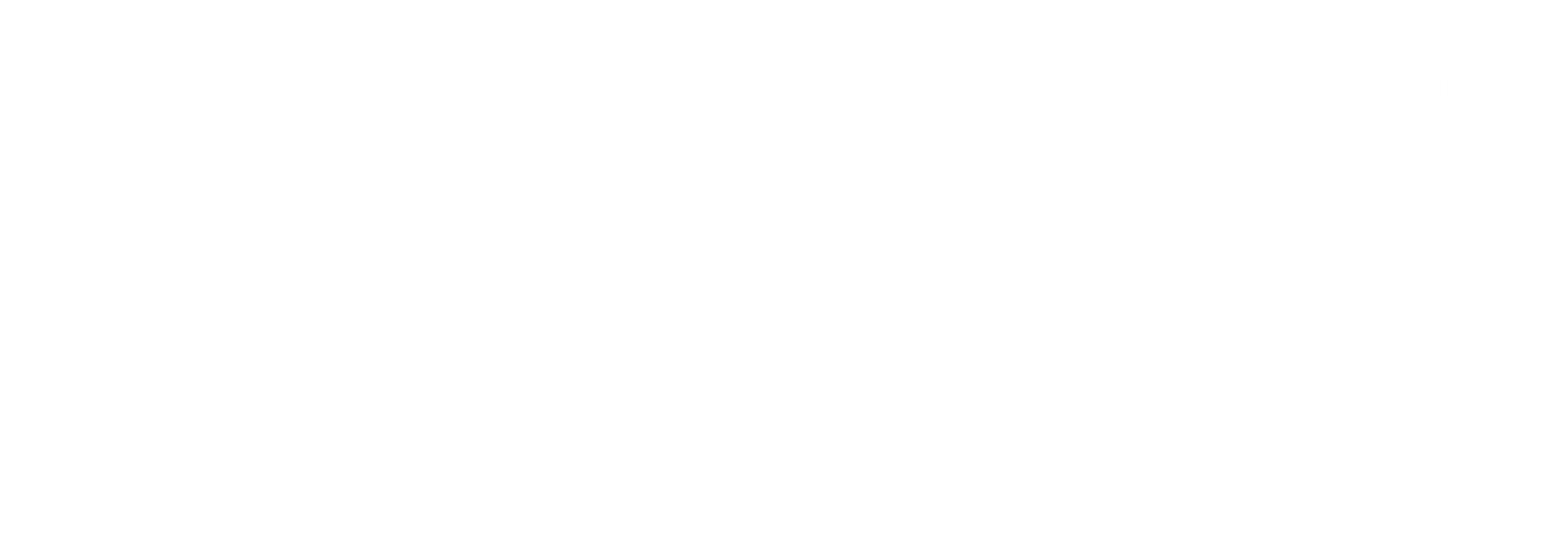
Here, 40 and 100 have heights of 0, 25 has a height of 1 and 50 has a height of 2. The balance factors of 40 and 100 are 0, the balance factor of 25 is -1 since and the balance factor of 50 is 1 since .

Now consider this tree:

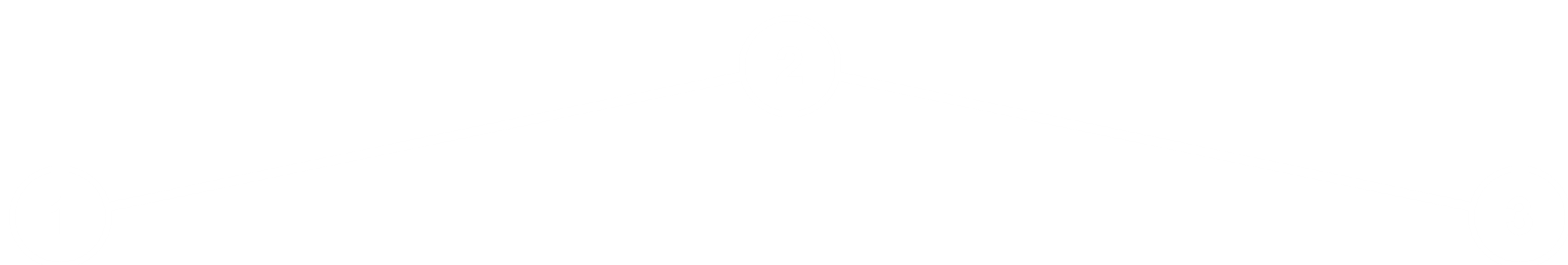


Following the left side of the tree, 2 has a balance factor of 0, 4 has a balance factor of 1, 7 has a balance factor of 0, 5 has a balance factor of 1 and 11 has a balance factor of 0. All of these are acceptable. However, 8 has a balance factor of 2, which is a problem. We have found the first point of imbalance, and we need to fix it. We will do this through something called rotation.

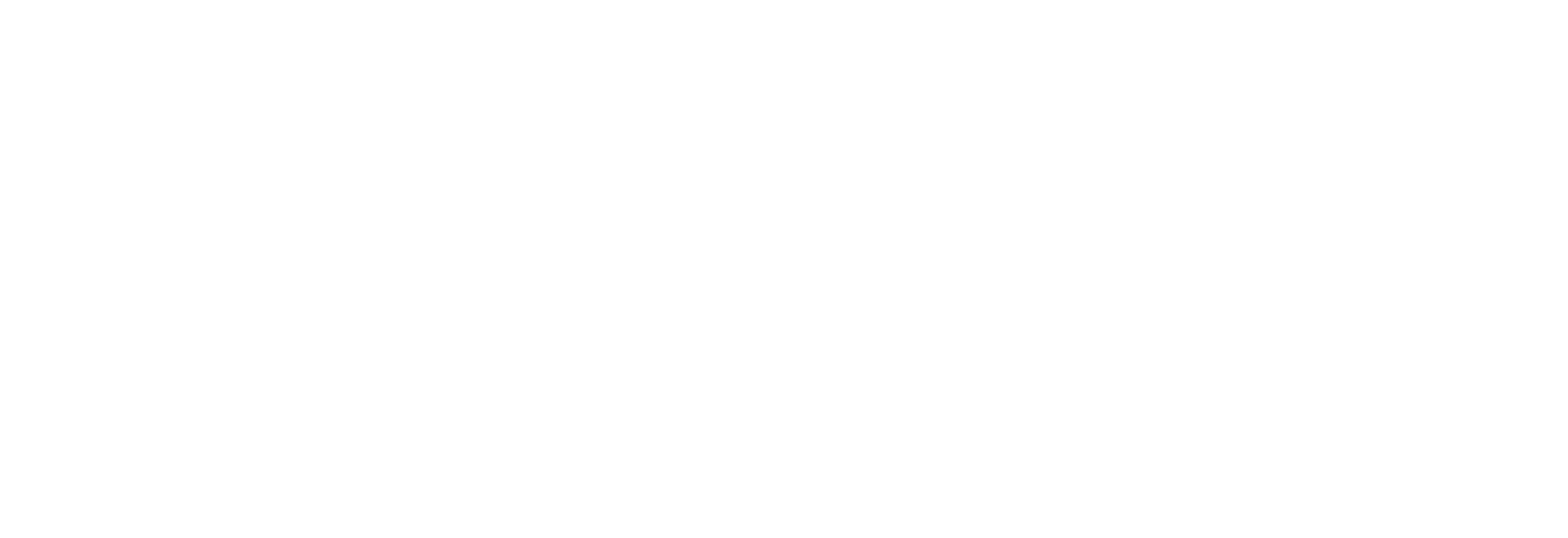
Consider this scenario:



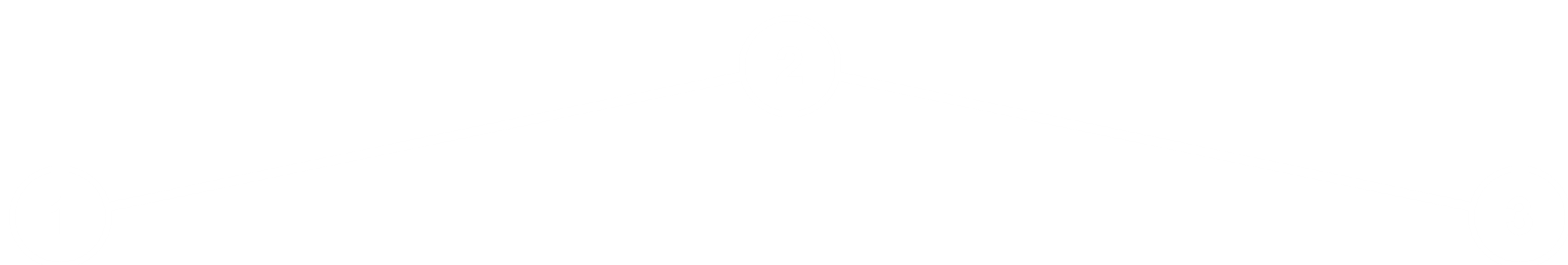
The first node, 3 is left heavy. Its heavy side child, 2 is also left heavy. This scenario is thus called LL. To fix this, we perform right rotation. Thus, the tree becomes like this:



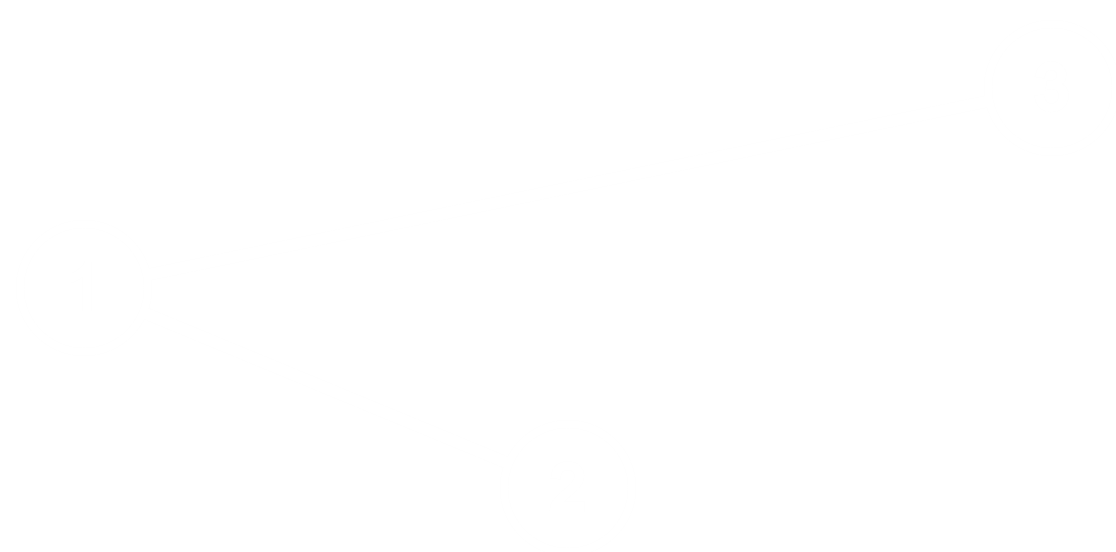
If the tree were like this:



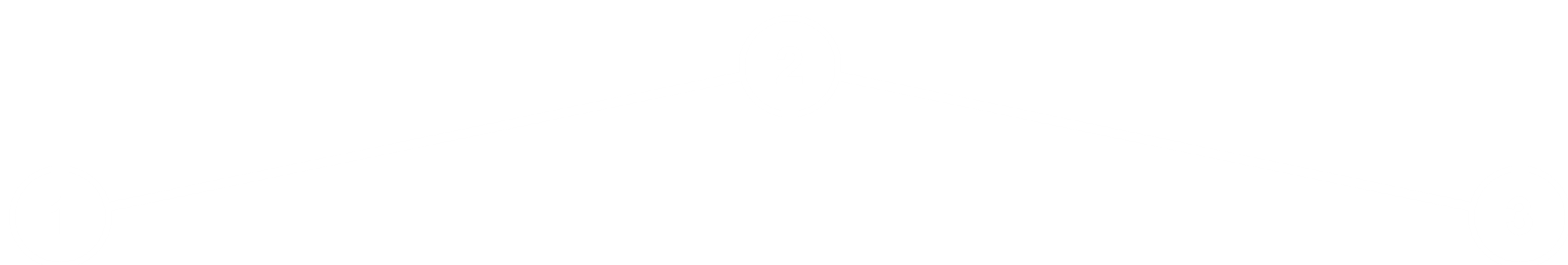
Then the first node is right heavy, and its heavy side child, 2, is also right heavy. This is thus a RR situation, and we will fix it by performing left rotation, giving us a tree that looks like this:



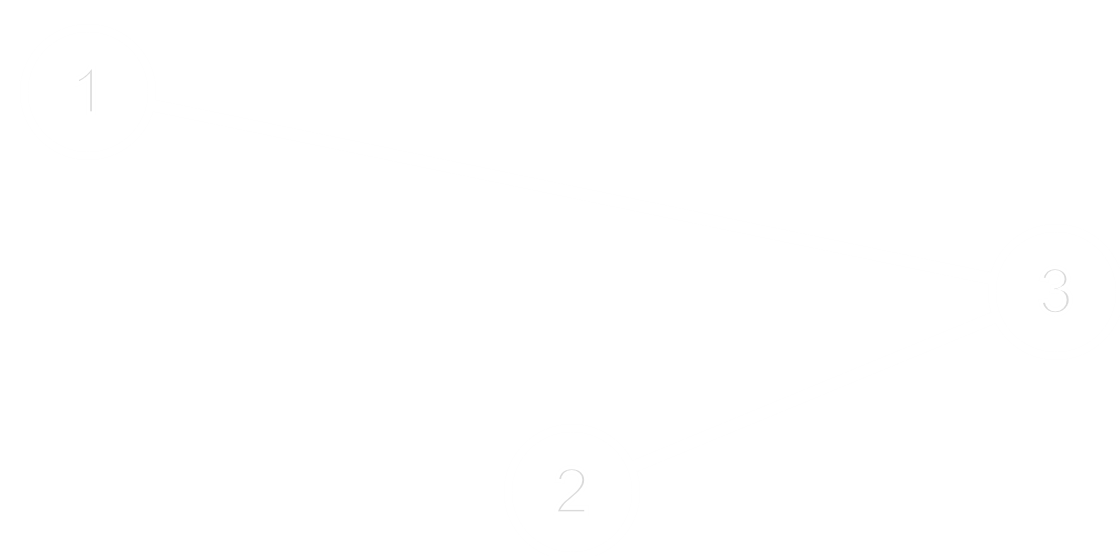
The next two scenarios are a little more complicated. Consider this tree:



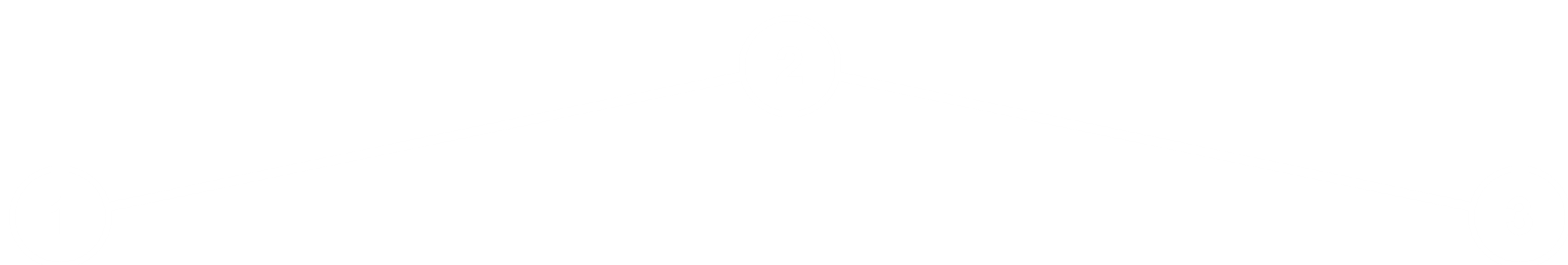
The original node is left heavy, but its heavy side child is right heavy. This is called LR. Here, if we perform simple right rotation, we will get 1 at the top with 2 on the left and 3 on the right. This breaks the properties of a BST since all values larger than 1 must be on its right. Thus, to fix this, we must first perform left rotation on the child node, making it LL, and then right rotation on the original node. We will get this tree at the end:



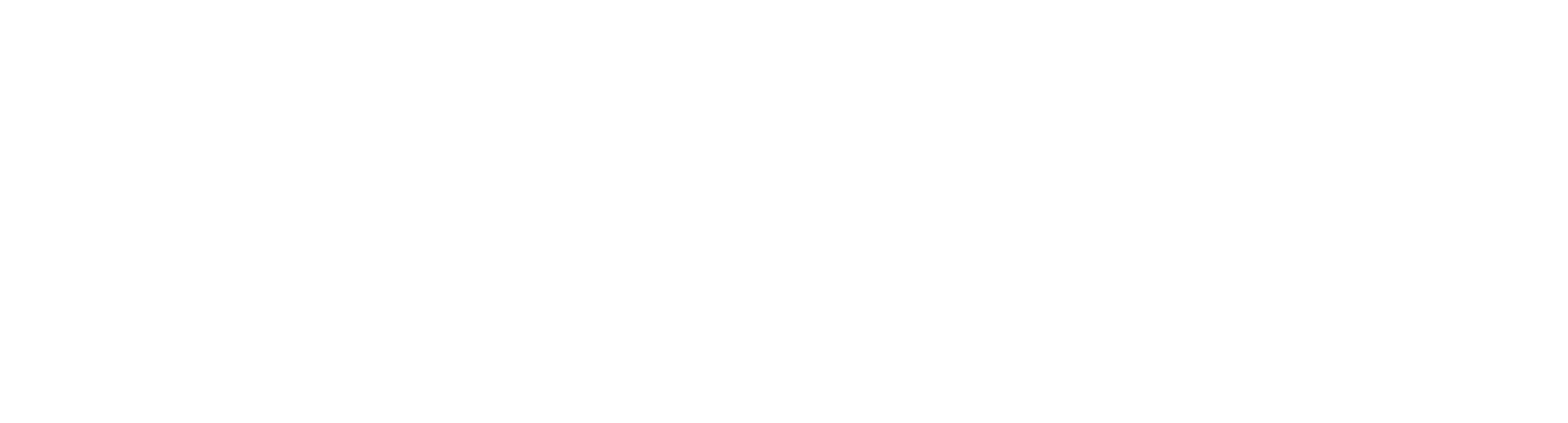
The final scenario is this one:



The original node is right heavy, while its heavy side child is left heavy, thus RL. Here we must first perform right rotation on the child, and then left rotation on the original node. Again, the resulting tree is:



To make the inner workings of these rotation clear, we will look at examples that are a little more complicated now.



The balance factor of 21 is 0. The balance factor of 35 is 1. 35 is left heavy. The balance factor of 45 is 2. 45 is also left heavy, and we have found an imbalance. This is an LL situation.

To perform right rotation, we need to do a few things. 35 will become the root in this situation. Thus, the parent of 45, which is **NULL** but could have been something else as well, needs to be set as the parent of 35, while 35 is set as the parent of 45. The left child of 35 stays as it is, but its right child, which was 38, must be 45 now. So where does 38 go? 38 becomes the left child of 45. If the right child of 35 had been **NULL**, we still could not skip this step since not changing the left child of 45 would mean it still pointed to 35, which would cause problems. The right child of 45 stays as it is. There are no other changes. Thus, to perform this single rotation, we are having to change a total of 6 pointers. We also need to change the pointers in the right order to avoid messing things up. For the node x at which we found the LL imbalance:

x->leftChild->parent = x->parent  
x->parent->child = x->leftChild  
x->parent = x->leftChild

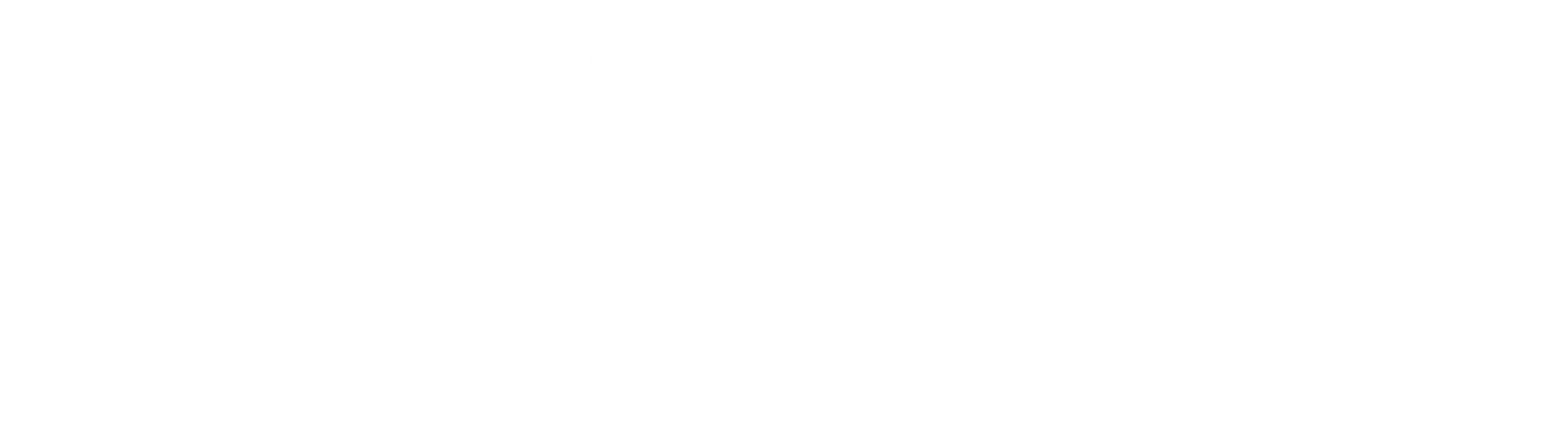
x->leftChild = x->parent->rightChild  
x->leftChild->parent = x  
x->parent->rightChild = x

PSEUDOCODE

The resulting tree looks like this:



Similarly, if we had a tree with any node that is imbalanced and right heavy and has a child that is also right heavy, we would perform left rotation. For this tree:



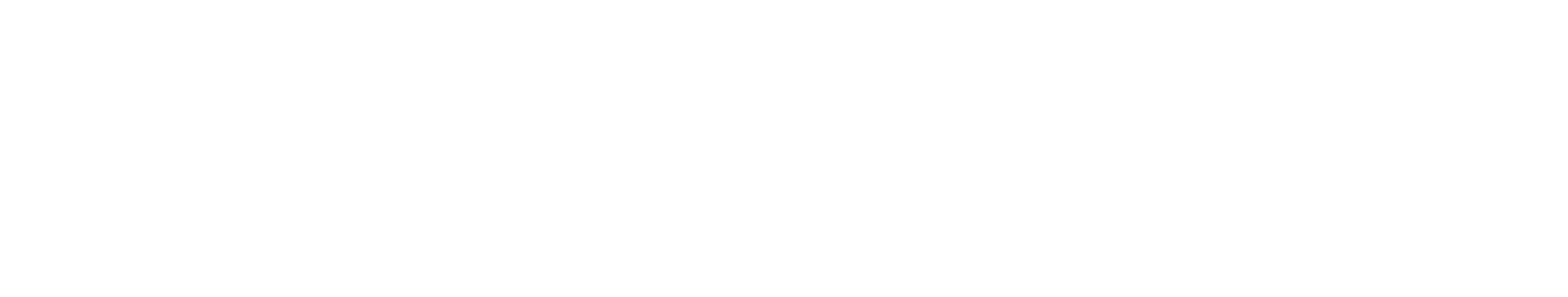
The balance factor of 23, 28, 45 and 35 is 0. The balance factor of 25 is -1, which is still acceptable, but the balance factor of 21 is -2, which is not acceptable. 21 is right heavy, and its heavy side child, 25 is also right heavy. Thus, we perform left rotation.

The parent of 25 is switched to the parent of 21, while 21 becomes the left child of 25. The existing left child of 25, 23, is switched to the right child of 21. The right child of 25 and the left child of 21 stay as they are. Again, this takes 6 separate pointer changes. If we consider the point of imbalance, 21, to be x, for the RR situation,

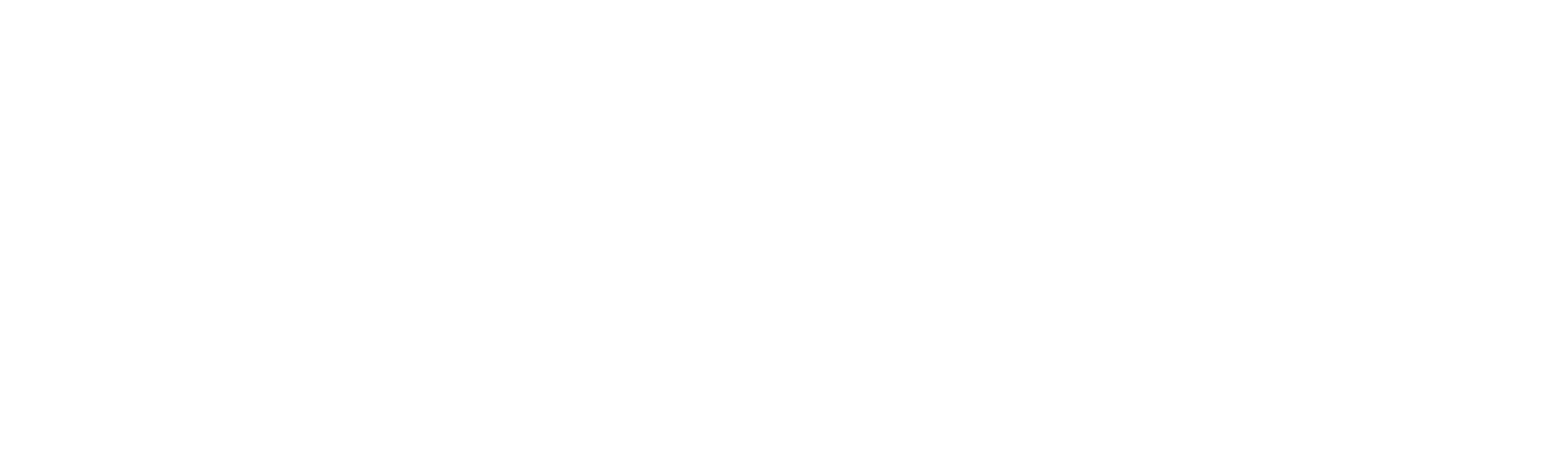
x->rightChild->parent = x->parent  
x->parent->child = x->rightChild  
x->parent = x->rightChild  
x->rightChild = x->parent->leftChild  
x->rightChild->parent = x  
x->parent->leftChild = x

C++

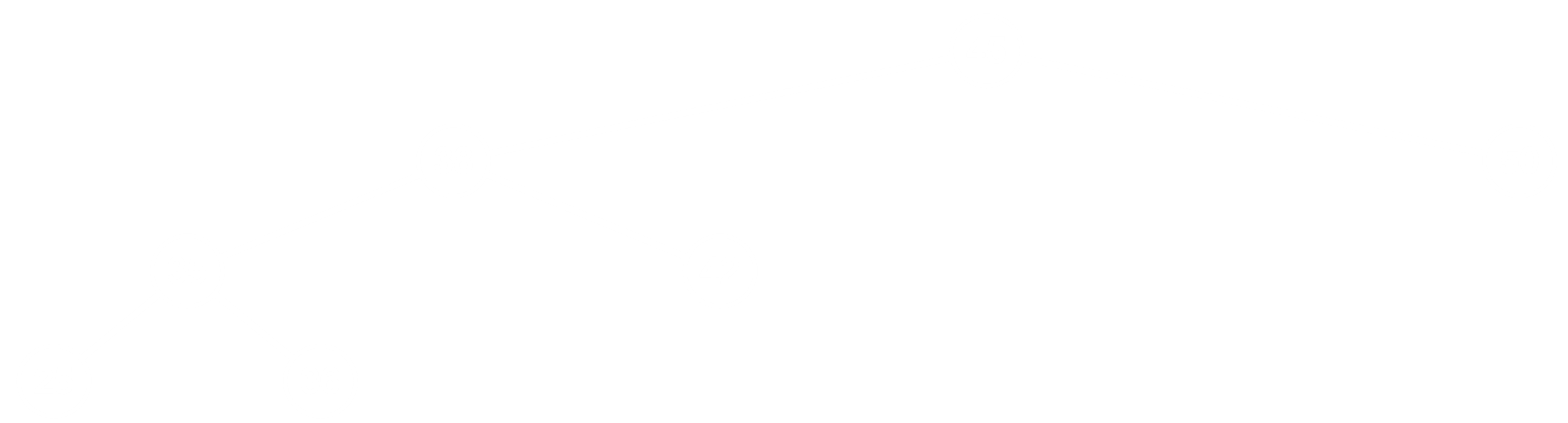
The resulting tree looks like this:



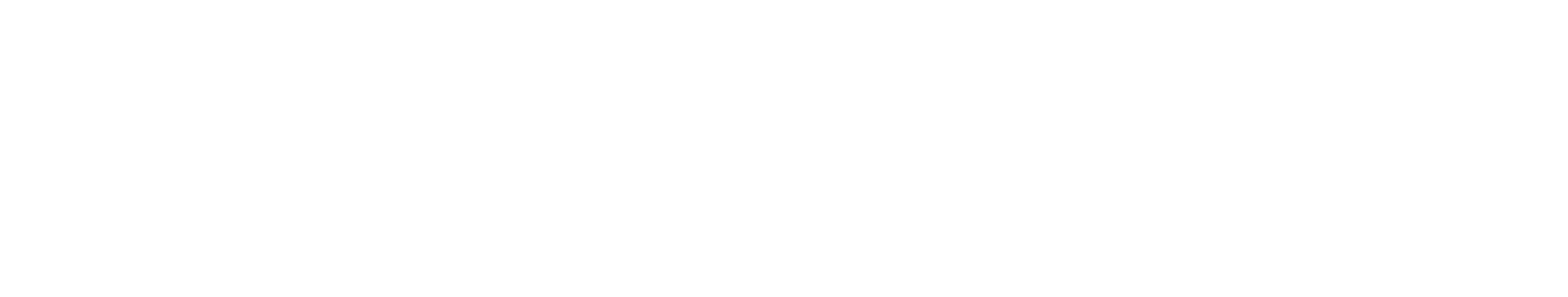
Now we go to the more complicated situations. First, look at the following tree:



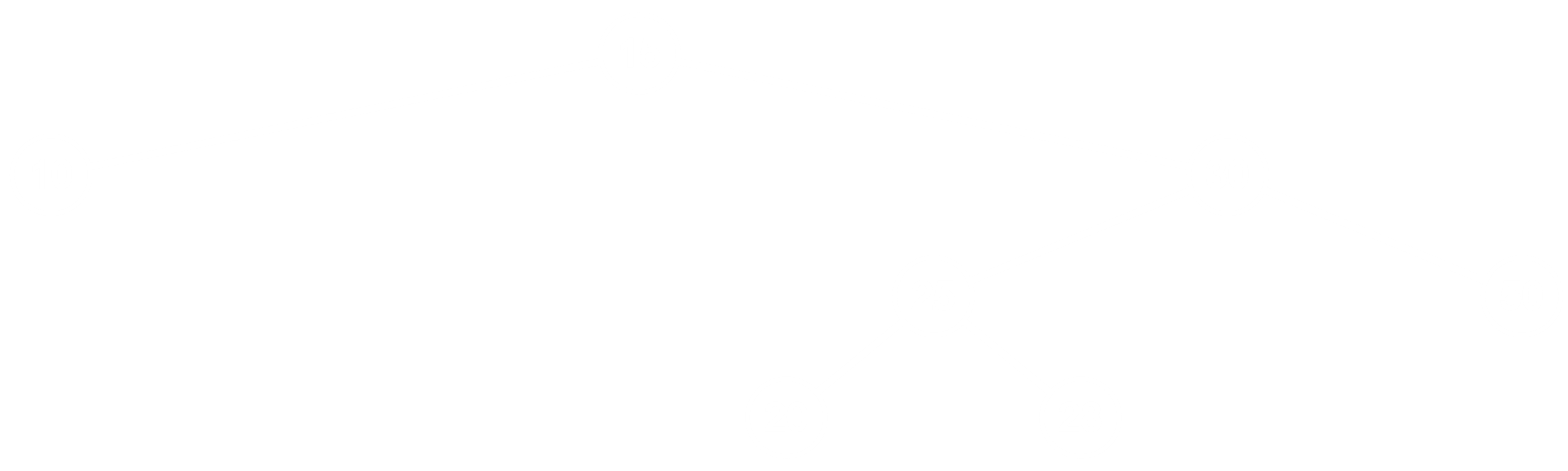
The balance factor of 36, 38, 42, 25 and 50 is 0. The balance factor of 35 is -1. The balance factor of 45 is 2 so we have a problem. 45 is left heavy and the heavy side child of 45 is 35, which is right heavy. We have an LR situation. To fix this, we need to perform left rotation on the child, 35, and right rotation on the parent 45. We simply use the two methods described earlier one after another to do this. After the left rotation on the child we have this:



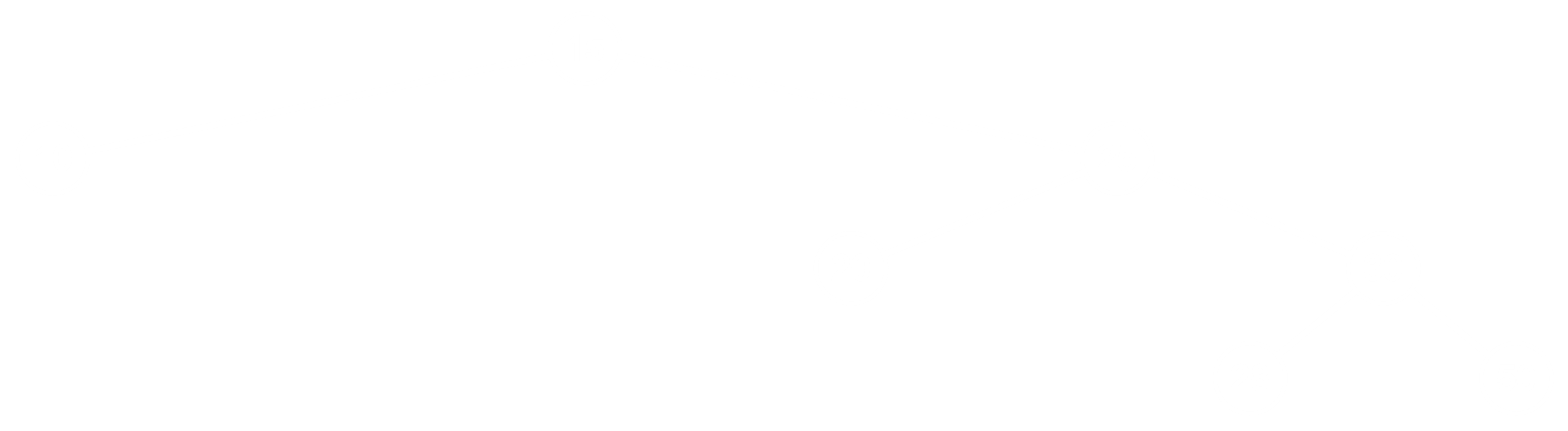
After right rotation on the parent we have this:



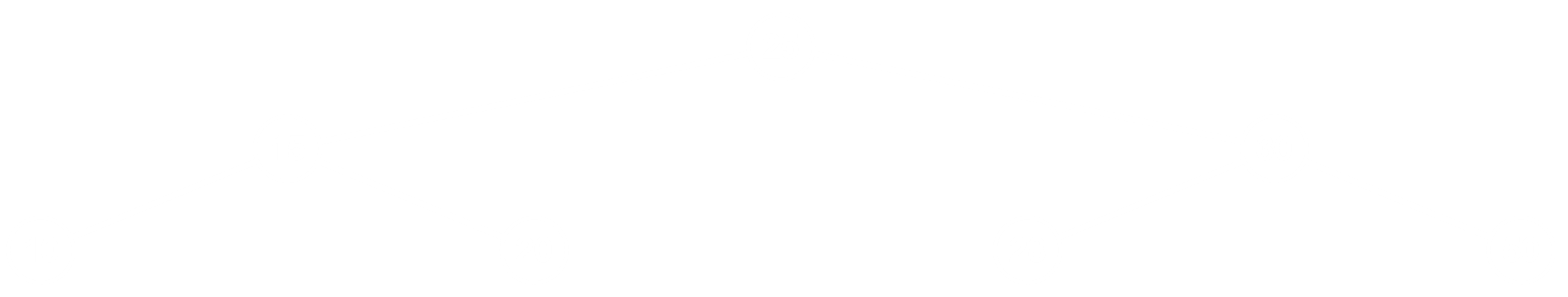
Finally, we look at the last form, RL. Look at this tree:



The balance factors of 28, 20, 25, 10, and 50 are 0. The balance factor for 30 is 1 and the balance factor for 15 is -2. The final bit, is the problem. We have a right heavy imbalanced node, 25, with a left heavy child, 30. Thus, we must first perform right rotation on the child, and then left rotation on the parent. Again, we do this using the same functions we described earlier on. After the right rotation on the child, we get this:

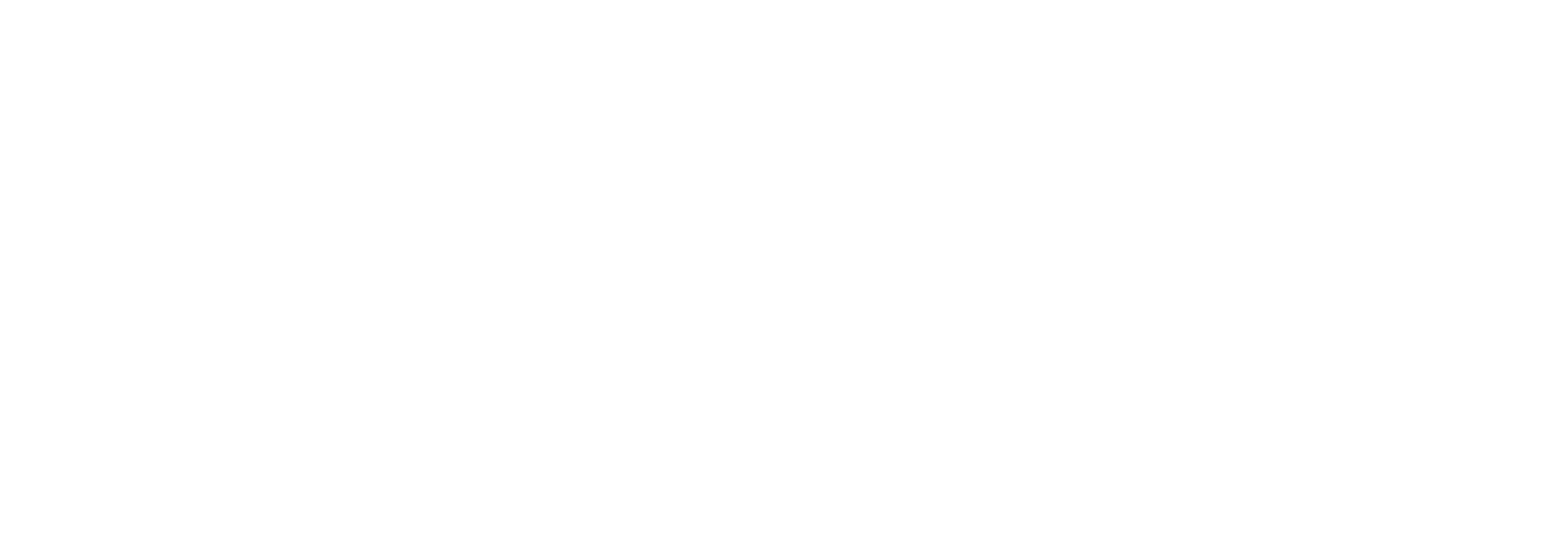


And after the left rotation on the parent we get this:

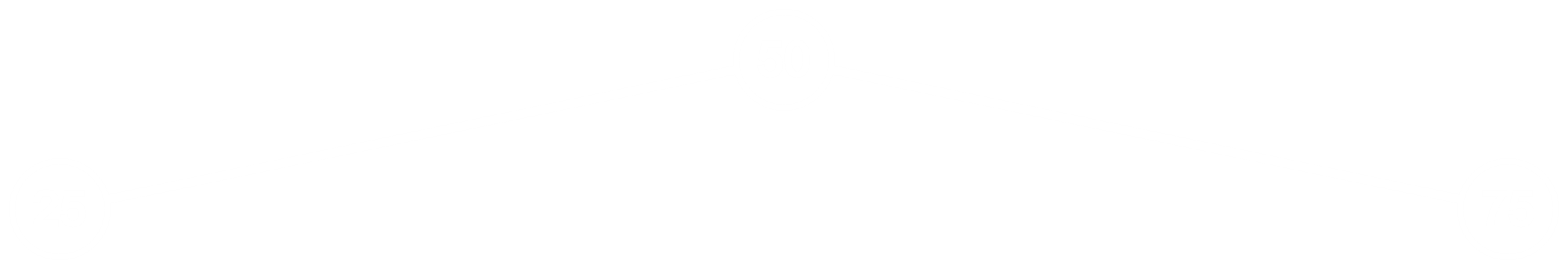


None of the rotations are ever actually done on a created BST. Instead, we check for imbalances as we add nodes and correct them immediately. If we do not do this, then we might have errors higher up in the tree which we miss. Since all the examples we saw thus far had no ancestors attached to them, we did not have to face this problem. To get an idea of how a BST is balanced during insertion, look at the following example.

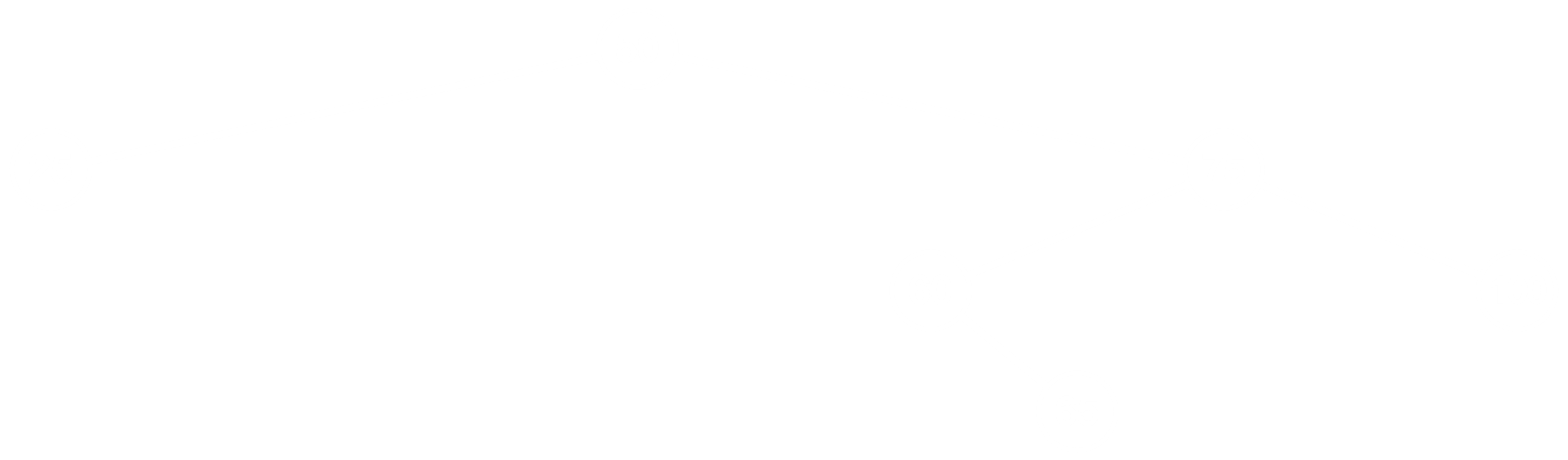
Say we add three nodes one after another, and get a tree that looks like this:



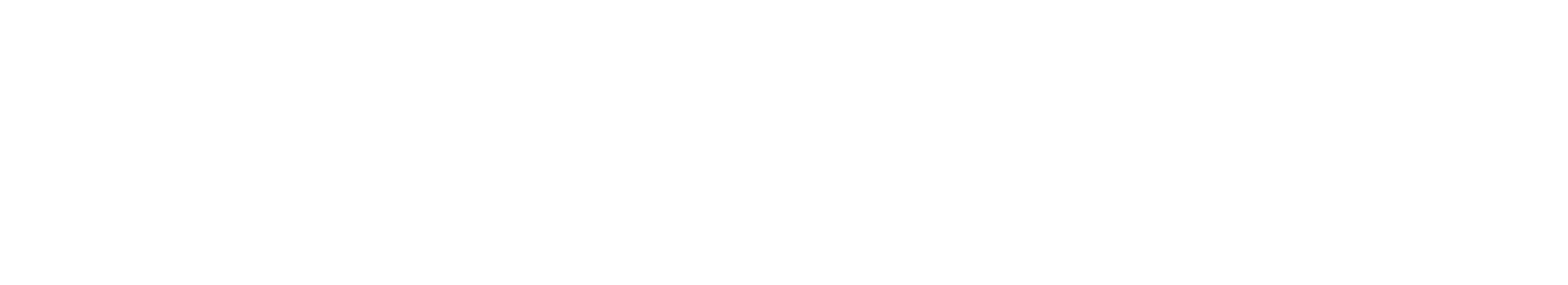
As soon as the third node, 75, is added, we get an imbalance. The node 25, has a balance factor of -2. We have to correct this before we move on, since if we do not then this imbalance will not be found later on. We use right rotation since this is an RR case. The resulting tree is:



Now we add a few more nodes:

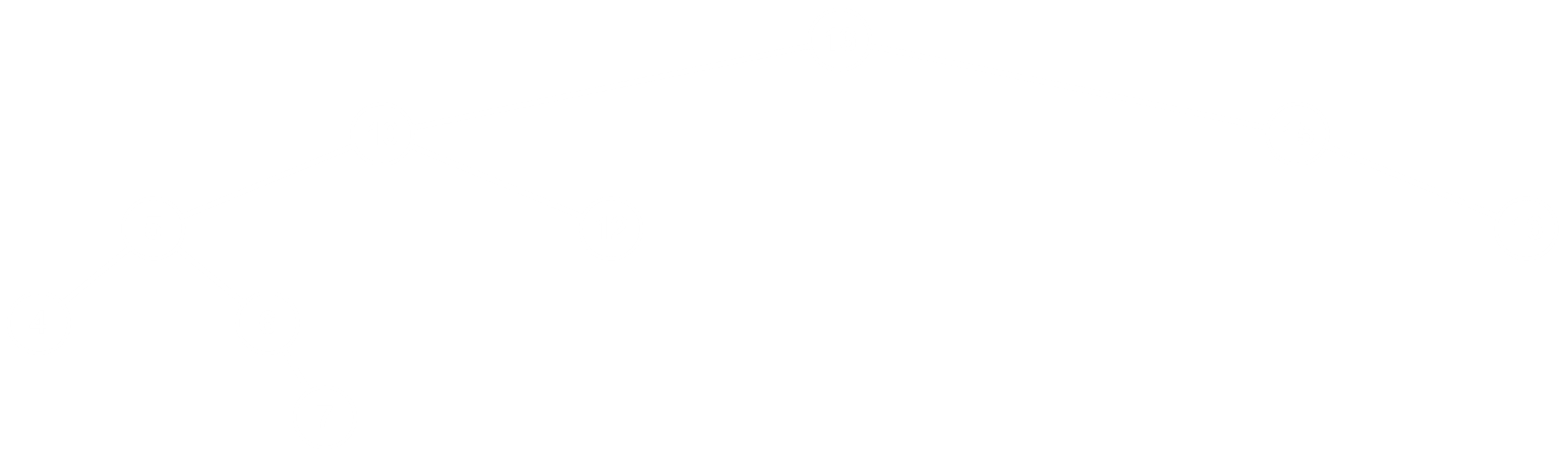


Now we get another error. The balance factor of 50 has become -2. Notice that had we not corrected the previous imbalance immediately, we would now have had an imbalance above this one as well, which we would be unable to correct due to the way we set up our algorithm. Since here, the imbalanced node has a right heavy child and that child is left heavy, the situation is RL. We first perform right rotation on the child, and then left rotation on the parent. The result, is this:

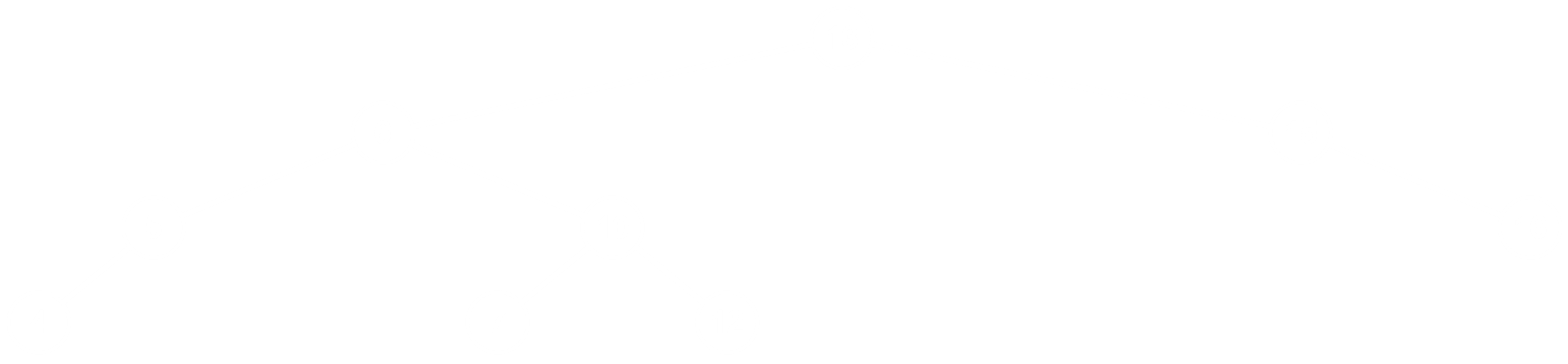


Notice that unlike every previous example, this is not a perfectly balanced tree. However, it is not imbalanced either, so it is acceptable.

For further proof of the fact that we do not need to correct anything other than just the first imbalance we find, look at this example:

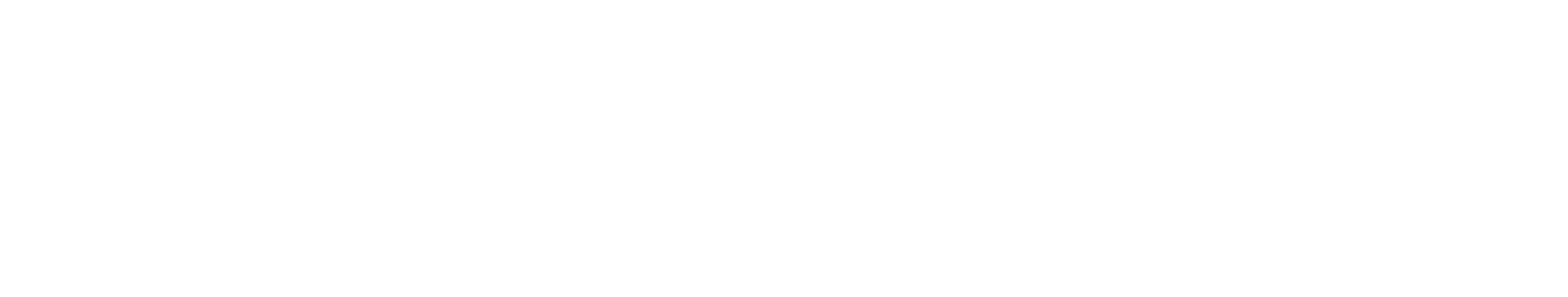


Going up, we find the first imbalance at 10, where the balance factor is 2. Of course, this also means that the ancestors of 10 are also imbalanced, but we do not need to worry about that. Correcting the imbalance at 10 will give us this tree:

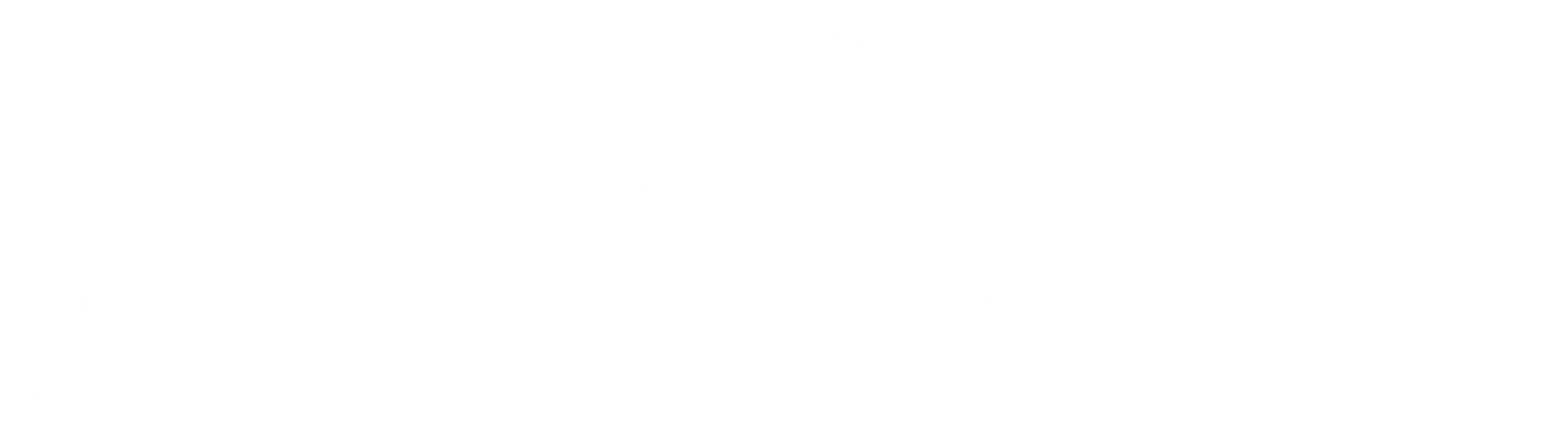


If we did not have 16 in the original tree, then 13 would still have been imbalanced. However, since we are performing these operations while inserting nodes, we would have caught the problem at 13 in that case before the problem at 10 had even arisen. Thus, fixing just one imbalance as soon as it occurs, while inserting nodes, fixes any imbalances higher up as well.

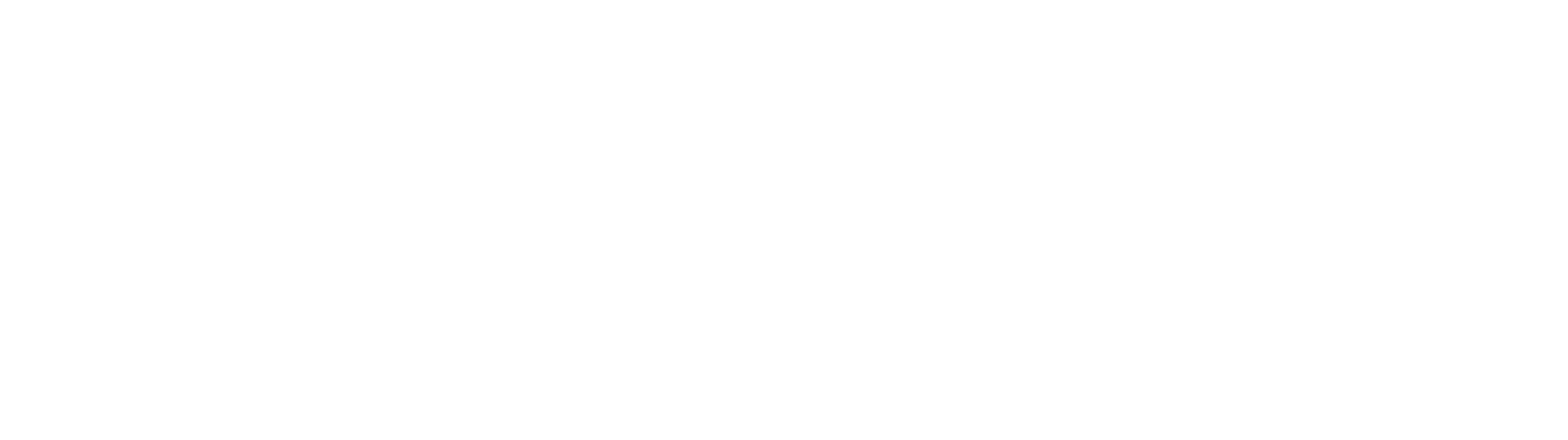
So far, we have only discussed inserting nodes. Deleting nodes in an AVL trees is done using the normal BST deletion function, but it makes life a little more difficult. In that scenario, we might have imbalances that are above the node we just deleted. This will be easier to understand with a few examples:



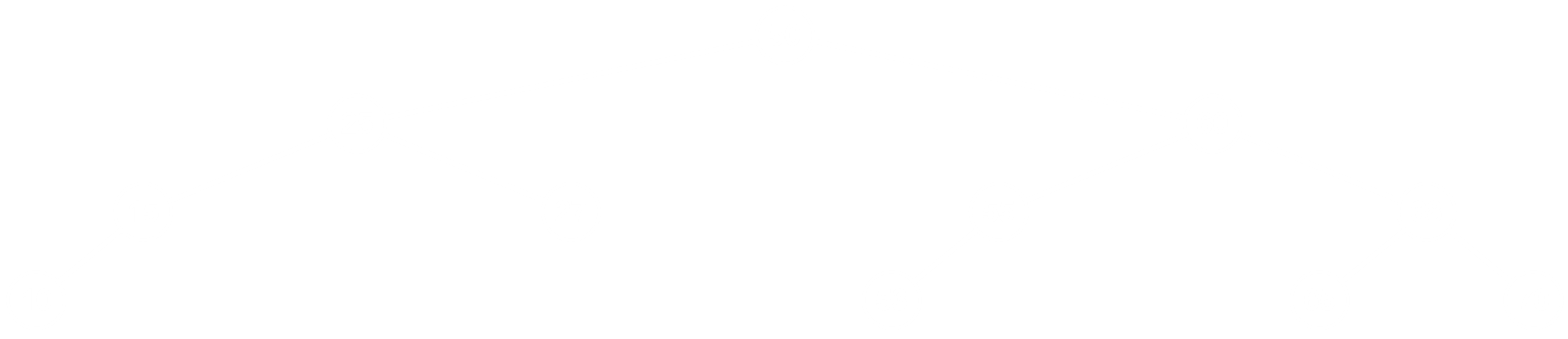
In this tree, if we delete 8, then the tree is still balanced, albeit barely. If we delete 5 as well, then we have an imbalanced tree. However, we can just perform right rotation on 4 to fix the situation. This is easy enough. Now take a look at this tree:



If we delete 80, then we get an imbalance at 70, since the balance factor there becomes 2. We fix this with right rotation, since we have an LL situation. The tree becomes like this:



However, we are still left with an imbalance. Looking at 60, we see that the balance factor is 2. Thus, we have to fix that as well now. Perform right rotation, we get this:



This tree is balanced. Thus, we can see that if we delete a node, we need to check for imbalances at each of its parents and fix them accordingly.