upGrad

Data Structures & Algorithms

upGrad

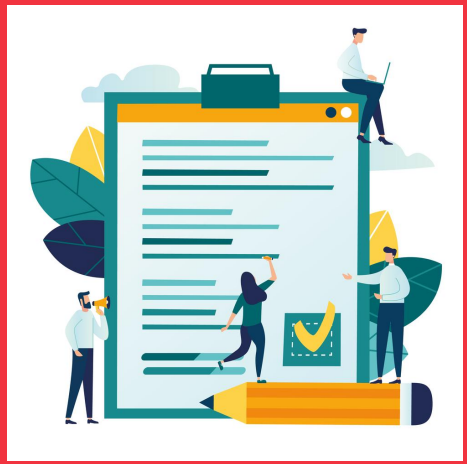
Course: Data Structures and Algorithm

Lecture On: BST

upGrad

In the previous class, we covered…

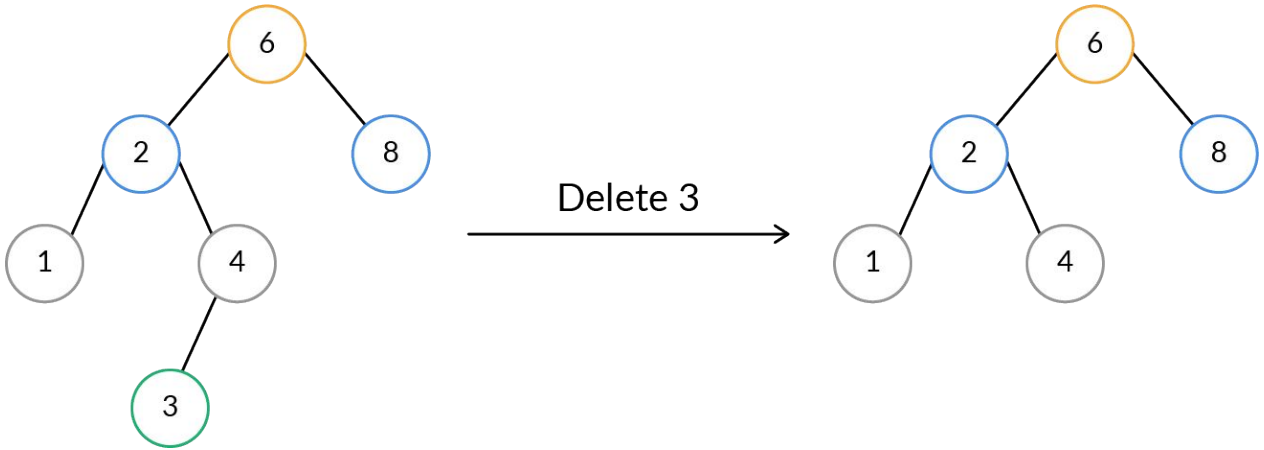
- Binary Search Tree



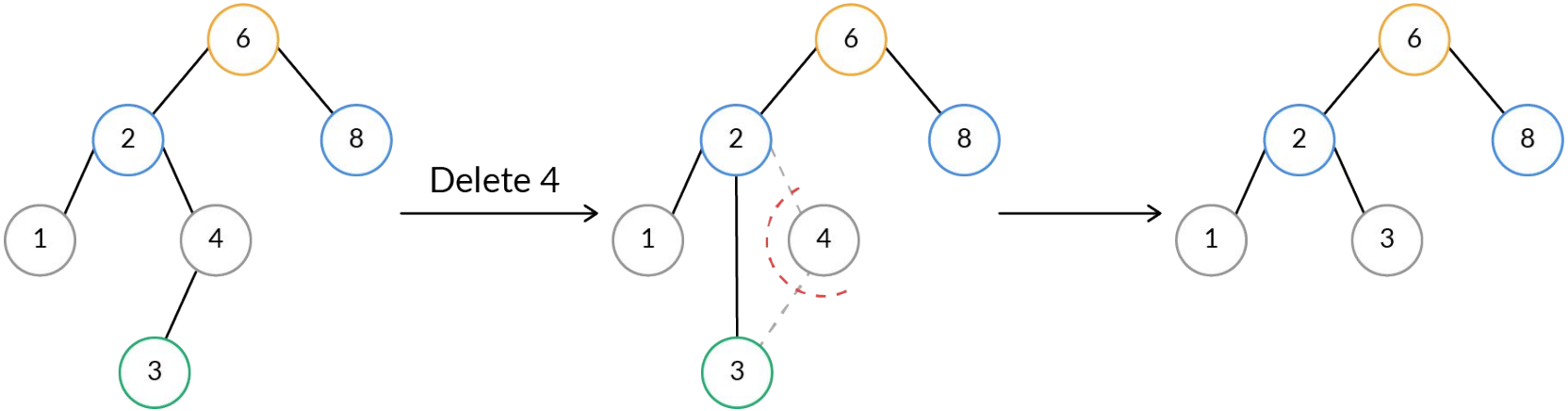
Today's Agenda

1 Binary Search Tree

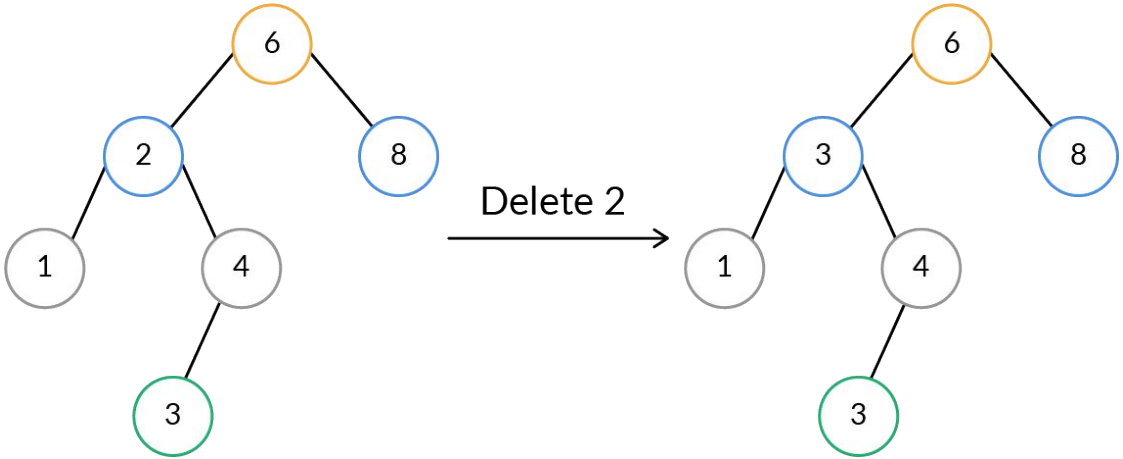
Let us now learn how we can delete an element from a Binary Search Tree. When we are to consider deleting a node, we must come up with a way to be able to delete the node irrespective of its type. The easiest situation for us would be if the node was a leaf!



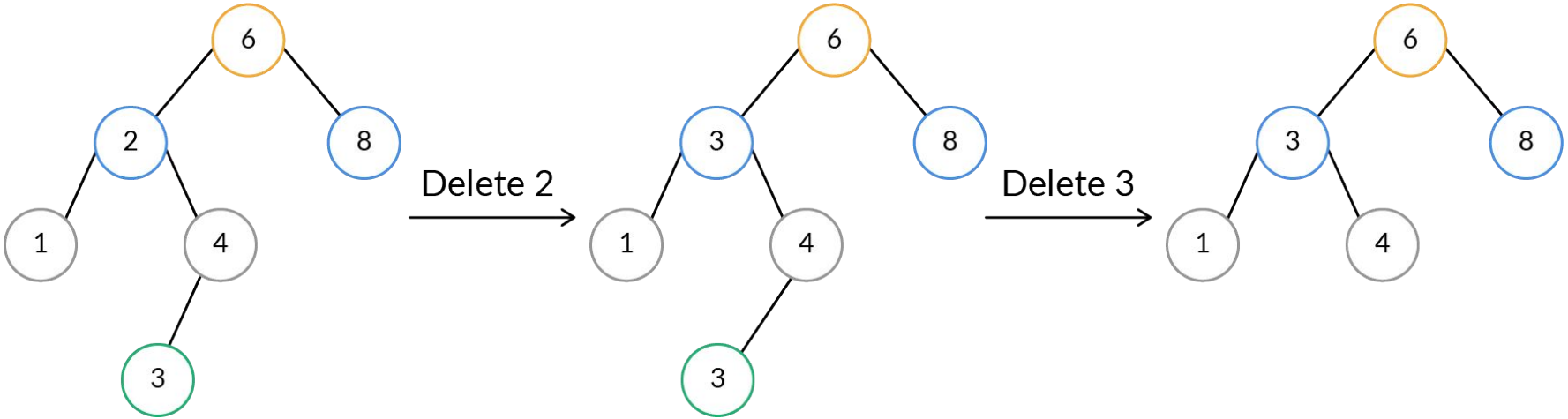
However, if the node has one or two children, then we would need to shuffle the remaining nodes a bit, preserve the properties of a Binary Search Tree, of course. If the node to be deleted has only one child, then it is still fairly easy. We can just attach the child node at the same place the deleted node was.



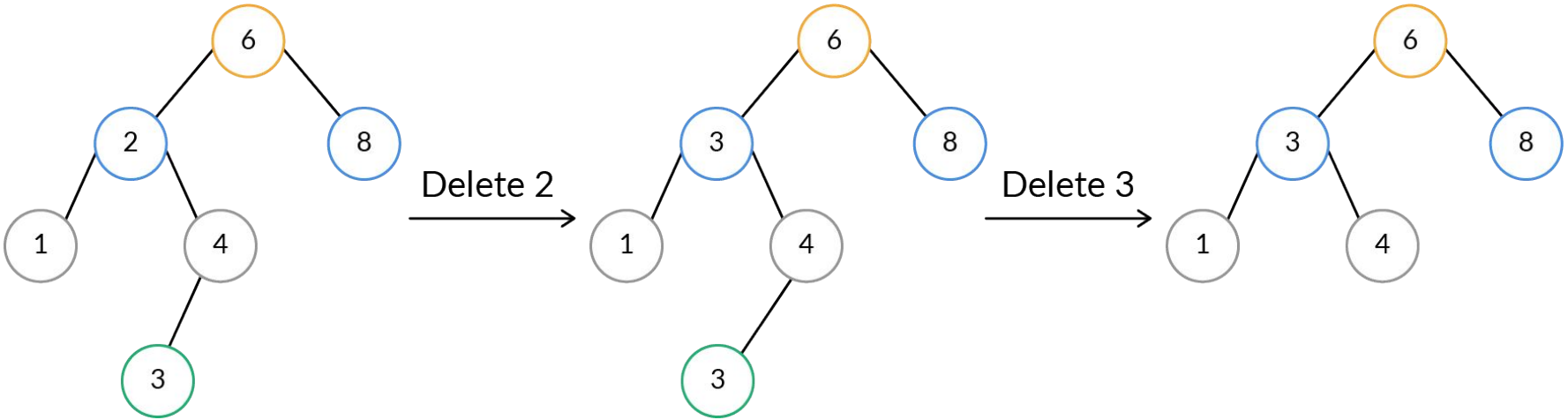
In case the node to be deleted has two children, we would need to do it in two steps. First, we need to search for the minimum element in the *right* subtree and replace the data of the node to be deleted with the data of this node. This essentially deletes the node that we want to delete.



If you look at the example, we now have two nodes with the data ‘3’. We will need to delete one of them. In order to delete this node, we can simply follow the steps that allow us to delete a leaf node, or a node that has only one child.



The minimum element in the right subtree of the node to be deleted can never have two children, which allows us to follow this particular algorithm. Can anyone figure out why this is so?



Binary Search Trees

upGrad

What is the time complexity of deleting a node in a Binary Search Tree? How does it change if the node to be deleted is a leaf, has one child or has two children?

Binary Search Trees

upGrad

So, now you know how helpful a Binary Search Tree is to us; it allows us to insert or delete an element in O(log n) time, while also supporting other operations such as finding the minimum elements, maximum elements and searching for any element in O(log n) time.

But all of these advantages rely heavily on one very important aspect. We had briefly touched upon it while discussing Binary Search Trees earlier. Can anyone recollect what that aspect is?

Binary Search Trees

upGrad

The best scenario for us would be a *complete* Binary Search Tree, which would allow us to exploit the O(log n) complexity to the hilt. The more degenerate a Binary Search Tree becomes, the more we regress on performance.

A completely degenerate Binary Search Tree is the worst possible scenario, giving us a very mediocre O(n) complexity in its operations.

However, we have seen that the structure of a Binary Search Tree relies on the order in which data is inserted into it, which, for all practical purposes, will be given to us by an external entity!

Does that mean we will be at the mercy of this external entity? Will it be the one dictating how performant our systems are?

Of course not!

Binary Search Trees

upGrad

We must come up with a way to achieve a time complexity as close to O(log n) as possible irrespective of the order in which we are given data. In order to do this, we must be able to create a Binary Search Tree that is as **‘balanced’** as possible.

Let us see how we can do that.

AVL Trees

Adelson-Velskii and Landis came up with an ingenious method by which we can create a ***self-balancing*** Binary Search Tree. Before we proceed, we must define what a 'balanced' tree actually is and what amount of 'balancing' is considered acceptable.

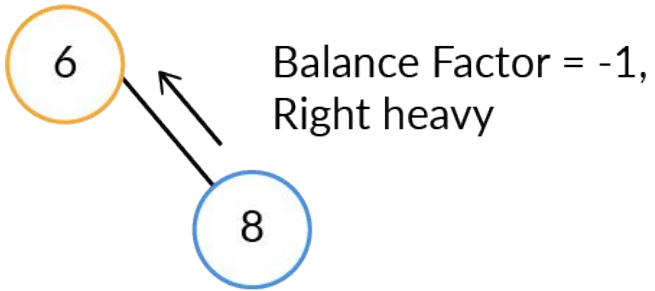
If we have a Binary Search Tree 'T', it is said to be height-balanced if:

- The left and right subtrees of 'T' are height-balanced themselves.
- The absolute difference in the heights of the left and right subtrees is either 0 or 1.

AVL Trees

To design an algorithm around this, we define something known as the *balance factor* of a node. This balance factor is the difference between the heights of the left and right subtrees of the node. In a balanced Binary Search Tree, the acceptable values of the balance factor of each node are as follows:

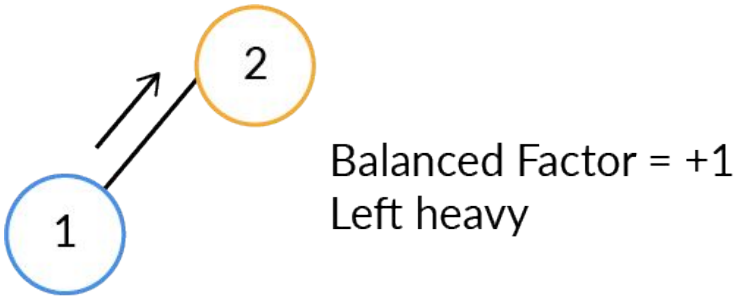
- - 1; these nodes are considered to be ‘right-heavy’



AVL Trees

To design an algorithm around this, we define something known as the *balance factor* of a node. This balance factor of a node is the difference between the height of the left subtree and the right subtree of the node. In a balanced BST, the only acceptable values of the balance factor of each node are as follows:

- + 1; these nodes are considered to be ‘left-heavy’



AVL Trees

To design an algorithm around this, we define something known as the *balance factor* of a node. This balance factor of a node is the difference between the height of the left subtree and the right subtree of the node. In a balanced BST, we only acceptable values of the balance factor of each node are –

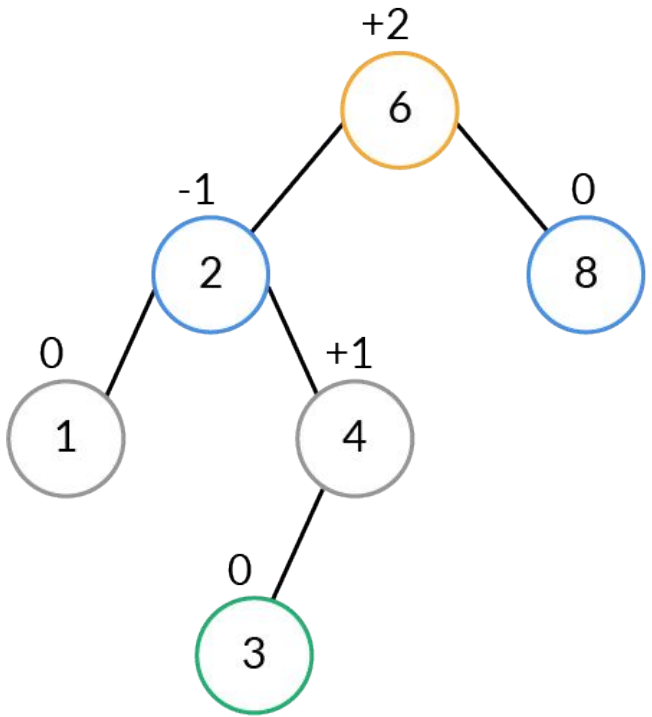
- 0; these nodes are perfectly balanced!

upGrad

AVL Trees

Now, let us figure how we can automatically balance a Binary Search Tree. First and foremost, we must keep in mind that the properties of a Binary Search Tree cannot be violated. This would also mean that the inorder traversal of the tree must be kept the same.

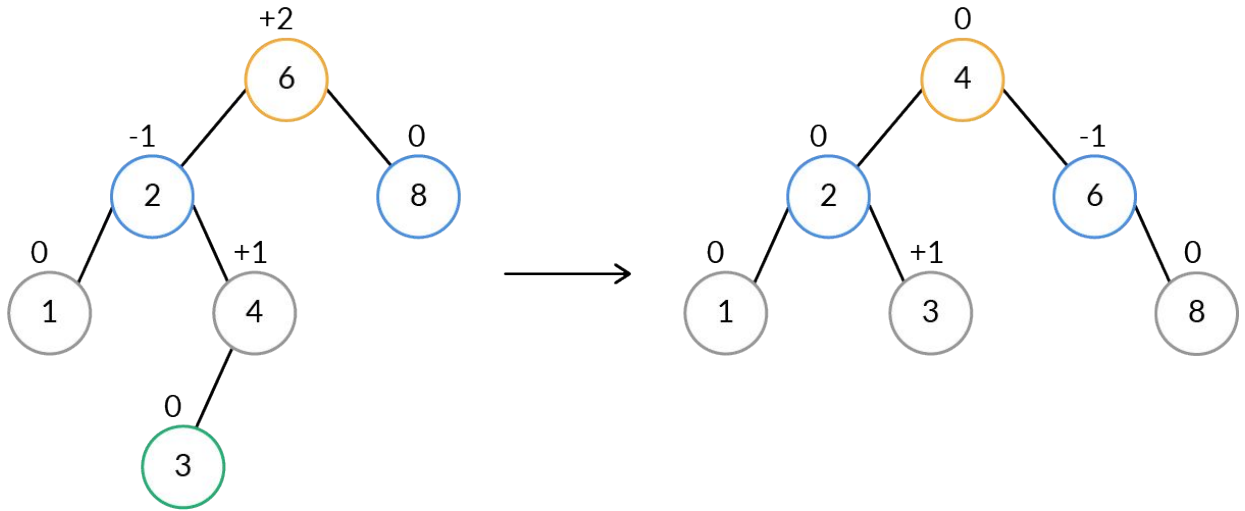
To check whether a Binary Search Tree needs balancing, we first calculate the balance factor for each node in the Tree.



AVL Trees

We can see that the root of the Binary Search Tree has an unacceptable balance factor, and must be balanced.

What if we were to balance the Binary Search Tree in the way we have shown below? Are the properties of the Binary Search Tree preserved? Is the in-order traversal still the same?



AVL Trees

But how do we do this?

Balancing performs a few set of operations, which we call ‘rotations’. We need to look for a few patterns in the Binary Search Tree and then perform these ‘rotations’ to balance the Tree.

To make an AVL Tree, we have four types of rotations:

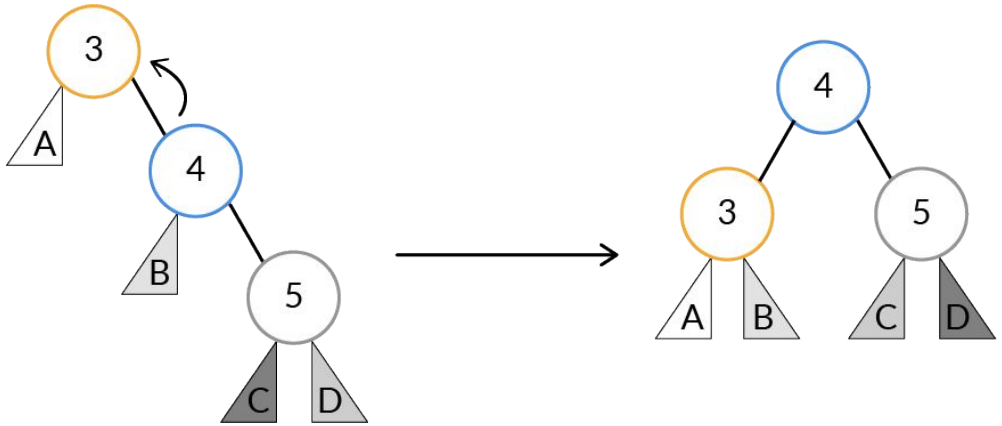
- The left-left (LL) rotation
- The right-right (RR) rotation
- The left-right (LR) rotation
- The right-left (RL) rotation

AVL Trees

Left-Left (LL) Rotation

Apply this rotation if you see that a node such as node '3' in the picture is not balanced, and is *right-heavy*.

Note the position of the subtree rooted at 'B' and how it gets adjusted after the rotation.

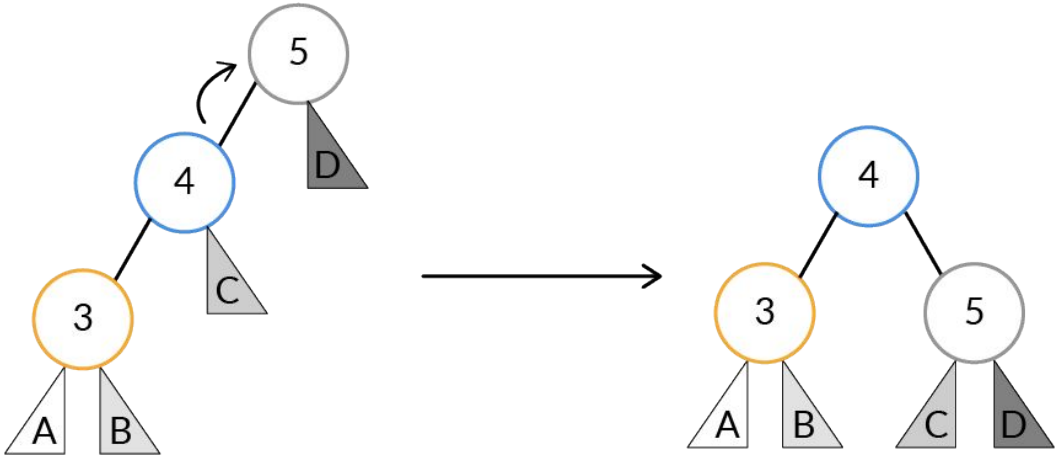


AVL Trees

Right-Right (RR) Rotation

Apply this rotation if you see that a node such as node '5' in the picture is not balanced, and is *left-heavy*.

Note the position of the subtree rooted at 'C' and how it gets adjusted after the rotation.

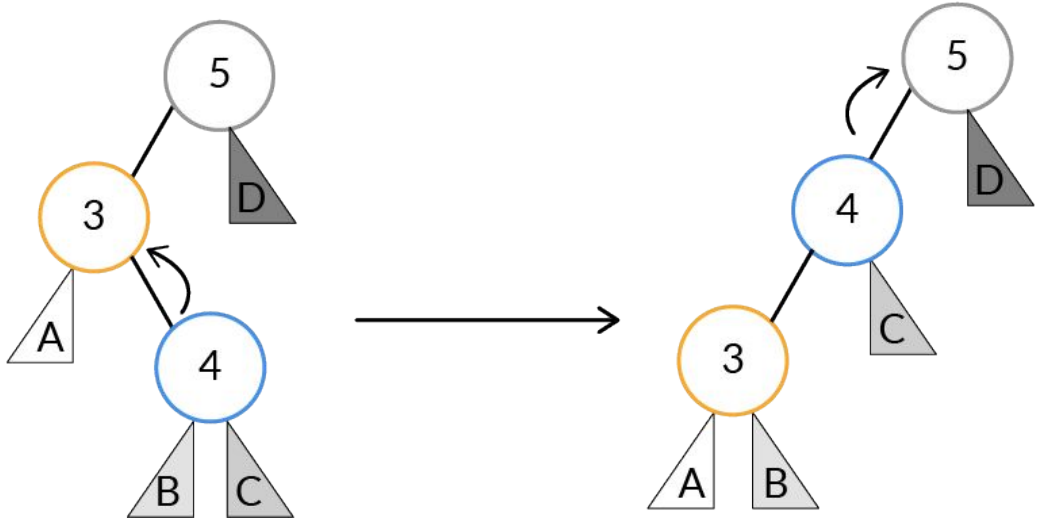


AVL Trees

Left-Right (LR) Rotation

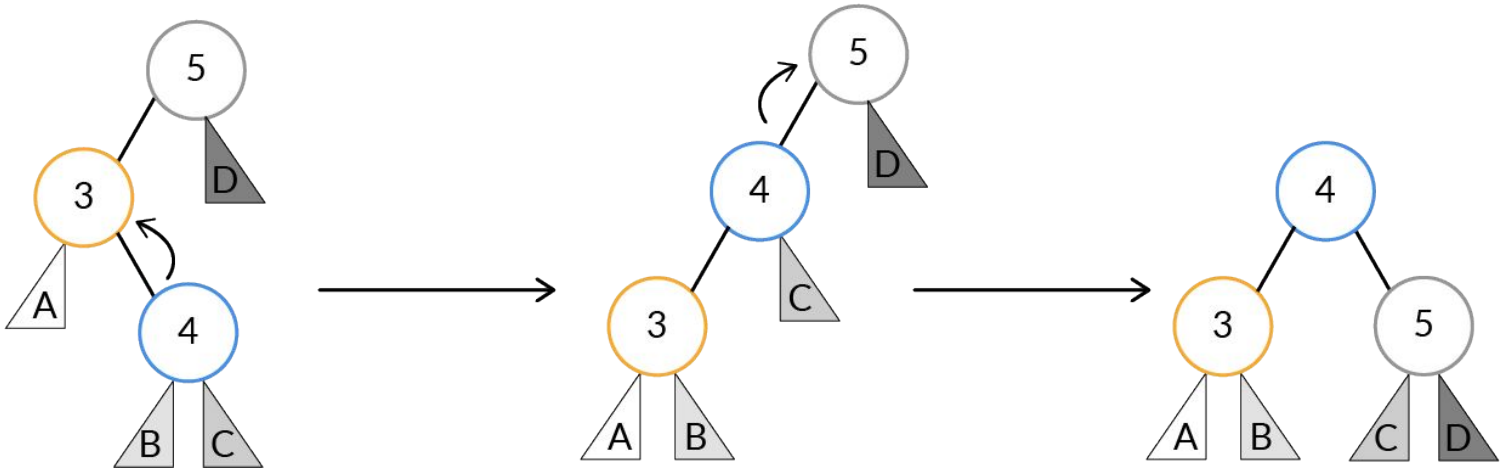
This and the next rotations are a bit more tricky. Apply this rotation if you see that a node such as node ‘5’ in the picture is not balanced, and is *left-heavy*.

As a first step, ‘rotate’ nodes ‘3’ and ‘4’. Observe how the subtree rooted at ‘B’ gets adjusted. Once we do this, we can see that we can do a simple RR rotation!



AVL Trees

Left-Right (LR) Rotation

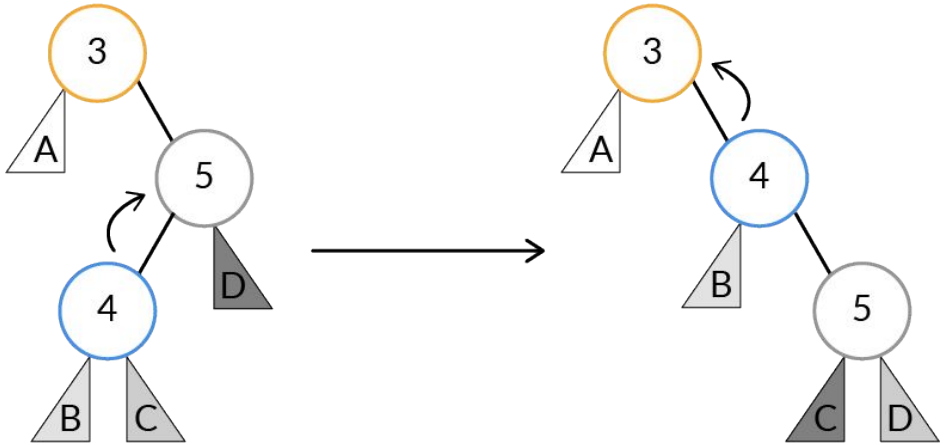


AVL Trees

Right-Left (RL) Rotation

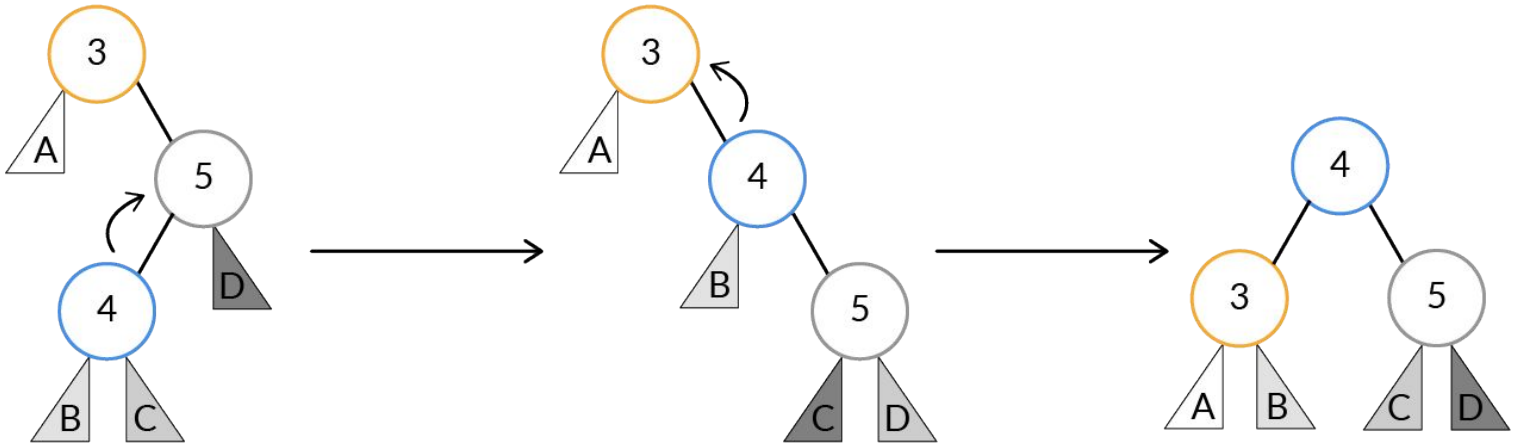
Apply this rotation if you see that a node such as node ‘3’ in the picture is not balanced, and is *right-heavy*.

As a first step, ‘rotate’ nodes ‘4’ and ‘5’. Observe how the subtree rooted at ‘C’ gets adjusted. Once we do this, we can see that we can do a simple LL rotation!



AVL Trees

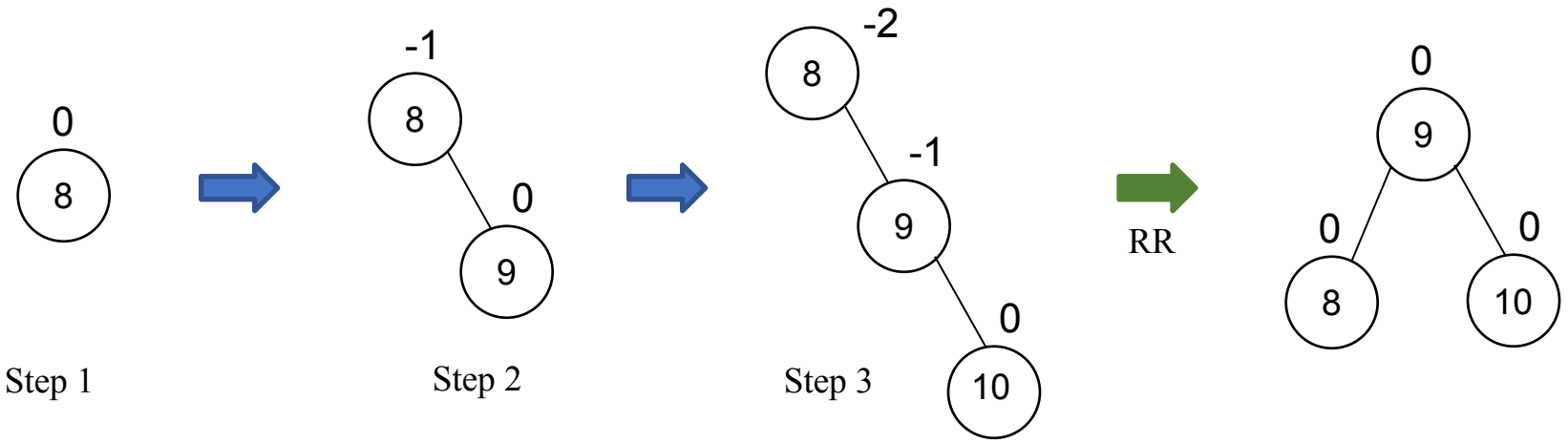
Right-Left (RL) Rotation



upGrad

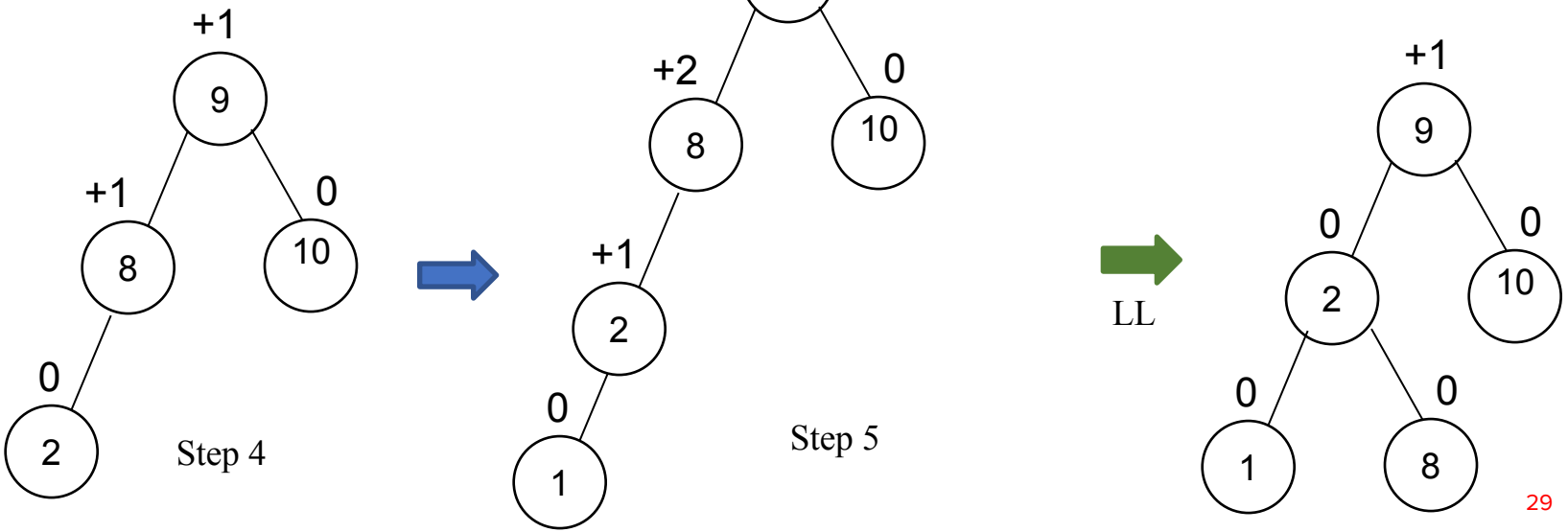
AVL Trees

Let us try and build a self-balancing Binary Search Tree from the following elements – 8, 9, 10, 2, 1, 5, 3, 6, 4, 7, 11, 12.



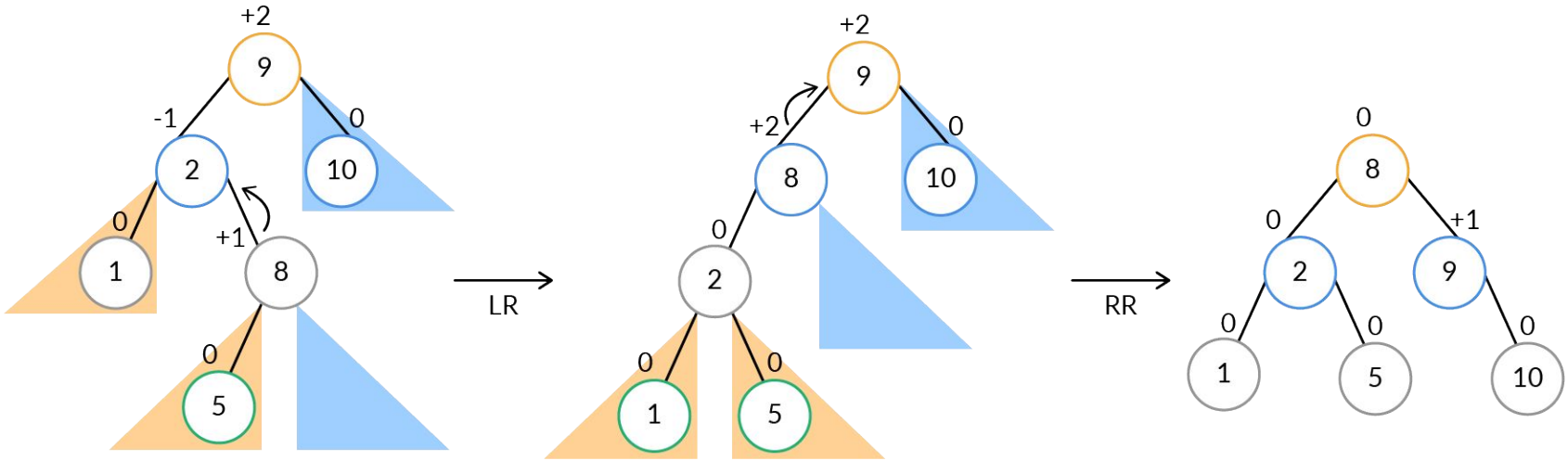
AVL Trees

Let us try and build a self-balancing Binary Search Tree from this example – 8, 9, 10, 2, 1, 5, 3, 6, 4, 7, 11, 12.



AVL Trees

Let us try and build a self-balancing BST from this example – 8, 9, 10, 2, 1, 5, 3, 6, 4, 7, 11, 12.



AVL Trees

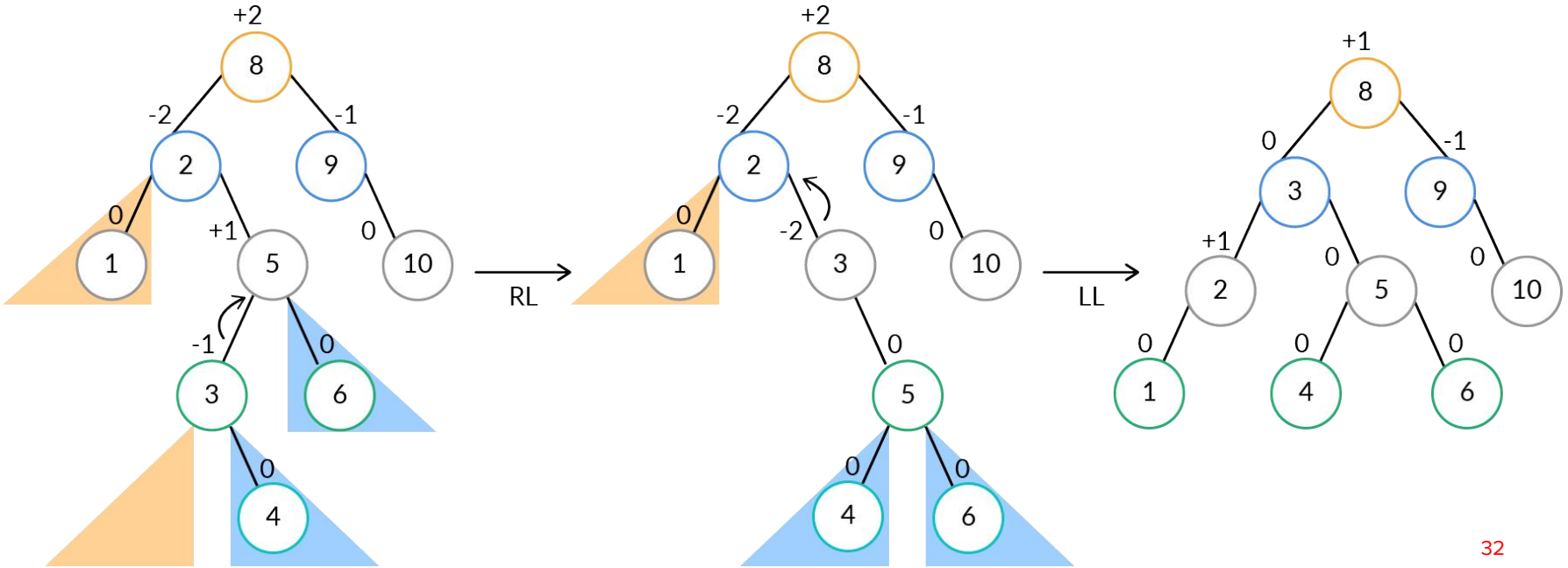
Let us try and build a self-balancing BST from this example – 8, 9, 10, 2, 1, 5, 3, 6, 4, 7, 11, 12.

Putting in 3 and 6 does not make the tree imbalanced...

Binary Search Trees

upGrad

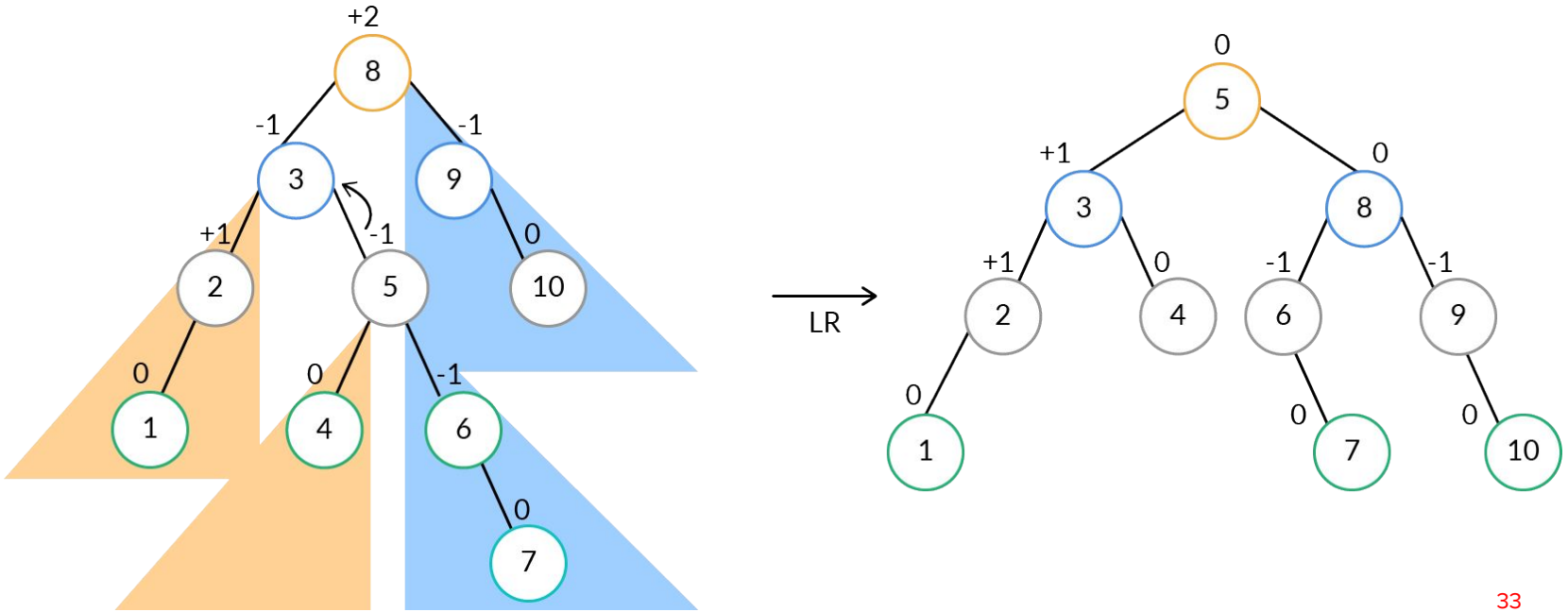
Let us try and build a self-balancing BST from this example – 8, 9, 10, 2, 1, 5, 3, 6, 4, 7, 11, 12.



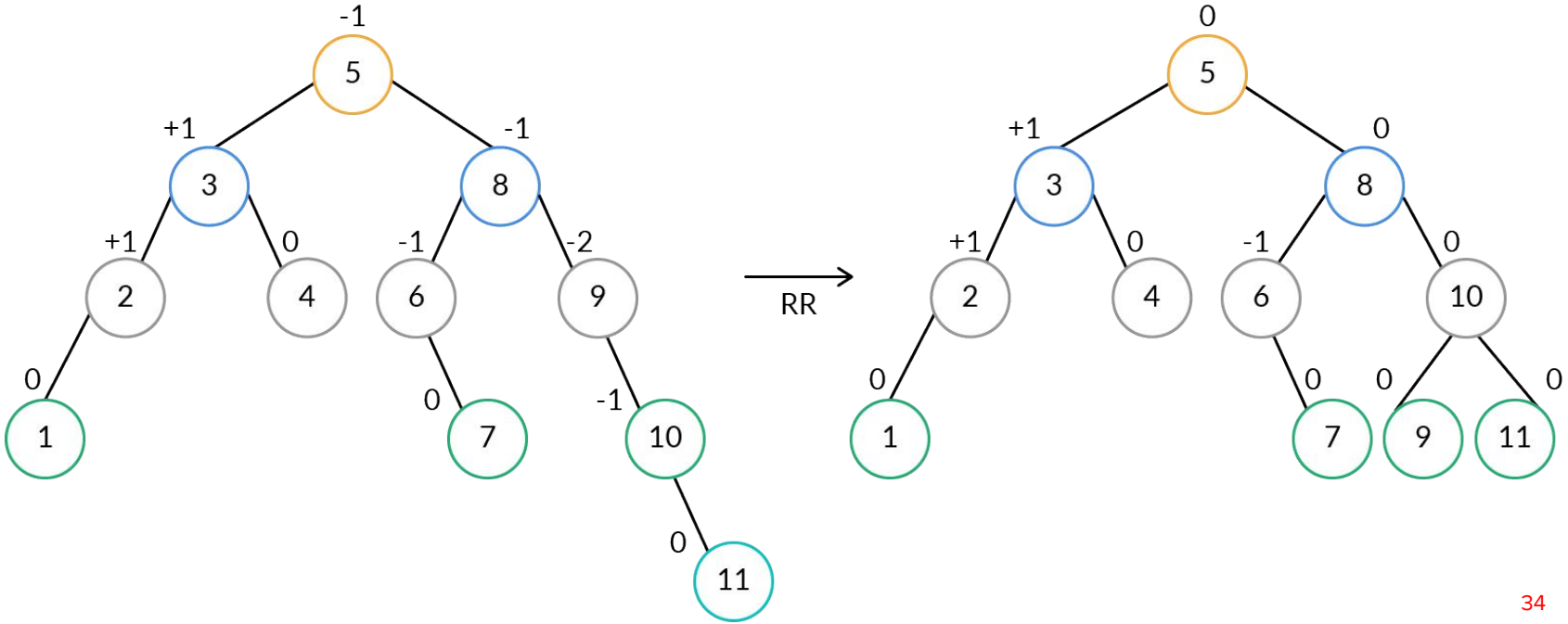
Binary Search Trees

upGrad

Let us try and build a self-balancing BST from this example – 8, 9, 10, 2, 1, 5, 3, 6, 4, 7, 11, 12.



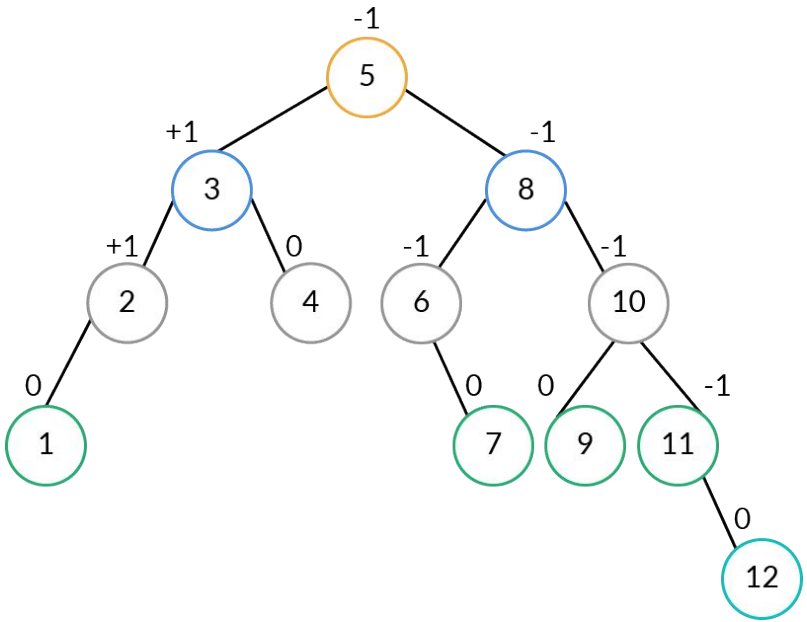
Let us try and build a self-balancing BST from this example – 8, 9, 10, 2, 1, 5, 3, 6, 4, 7, 11, 12.



Binary Search Trees

upGrad

8, 9, 10, 2, 1, 5, 3, 6, 4, 7, 11, 12.



Tasks to complete after the session

Homework
MCQs
Coding Questions

upGrad

#RahoAmbitious

Thank You!