

AVL Deletion

This lesson will cover the deletion operation in AVL trees, discussing all the four deletion cases.

We'll cover the following



- Deletion in AVL Trees
- Algorithm for Deletion
- *
 - 1. Delete the given node:
 - 2. Traverse Upwards:
 - 3. Rebalance the Tree
 - Case 1: Left-Left
 - Case 2: Left-Right
 - Case 3: Right-Right
 - Case 4: Right-Left

Deletion in AVL Trees

Deletion is almost similar to the insertion operation in AVLs with just one exception. The deletion operation adds an extra step after rotation and balancing the first unbalanced node in the insertion method. After fixing the first unbalanced node through rotations, start moving up and fix the next unbalanced node. Keep fixing the unbalanced nodes until you reach the root.

Algorithm for Deletion

Here is a high-level description of the algorithm for deletion.

1. Delete the given node:

Delete the given node in the same way as in BST deletion. At this point, the tree will become unbalanced and, to rebalance the tree, we would need to perform some kind of rotation (left or right). At first, we need to define the structure of AVL Tree and some nodes relative to the *currentNode* which is inserted using step one.

2. Traverse Upwards:

Start traversing from the given node upwards till you find the first unbalanced node. Let's look at some of the terms which we will be using while re-balancing the tree.

- *Node U* — an unbalanced node
- *Node C* — child node of node U
- *Node G* — grandchild node of node U

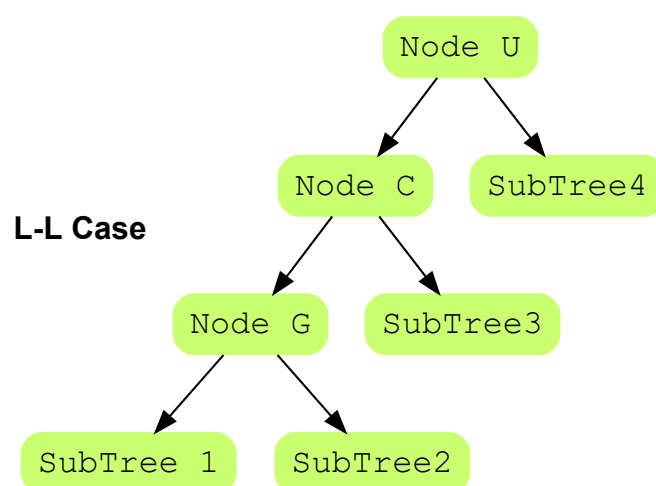
3. Rebalance the Tree

In order to rebalance the tree, we will perform rotations on the subtree where U is the root node. There are two types of rotations (left, right). We came across four different scenarios based on the arrangements of Nodes U, C and, G.

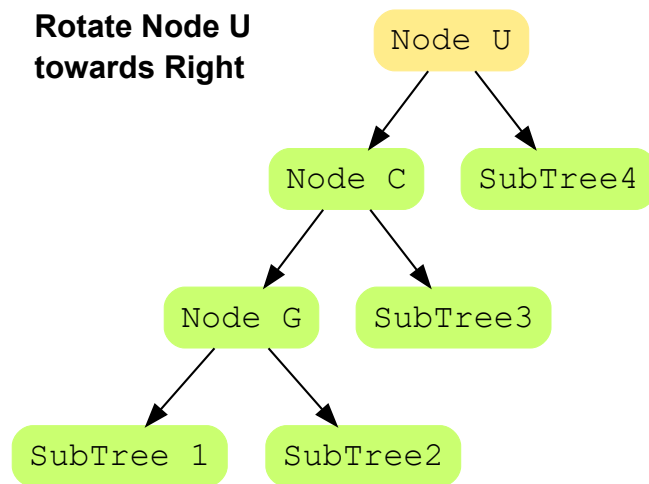
- *Left-Left*: Node C is the left-child of Node U, and Node G is left-child of Node C
- *Left-Right*: Node C is the left-child of Node U, and Node G is right-child of Node C
- *Right-Right*: Node C is the right-child of Node U, and Node G is right-child of Node C
- *Right-Left*: Node C is right-child of Node U, and Node G is left-child of Node C

After performing successful rotation for the first unbalanced Node U, traverse up and find the next un-balanced Node and perform the same series of operations to balance. Keep on balancing the unbalanced nodes from first Node U to ancestors of Node U until we reach the root. After that point, we will have a fully balanced AVL Tree that follows its property.

Case 1: Left-Left

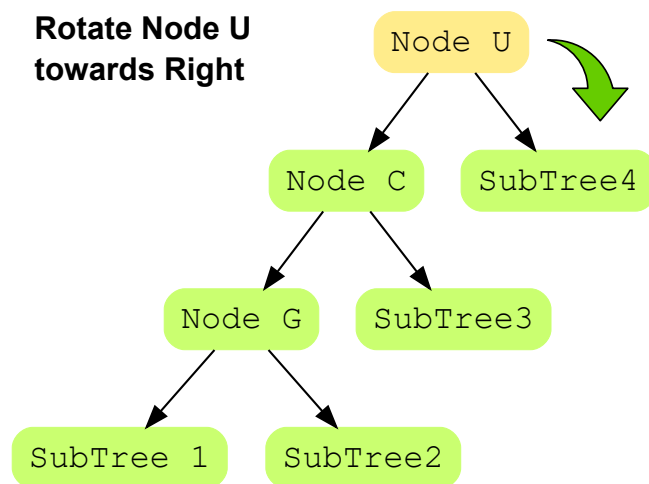


Rotate Node U
towards Right



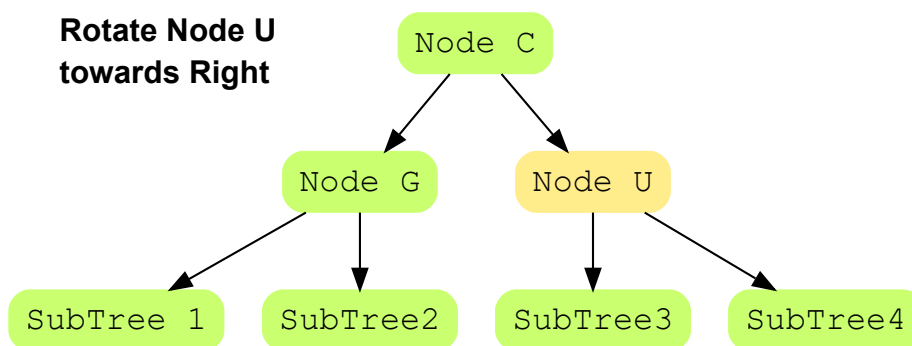
2 of 4

Rotate Node U
towards Right



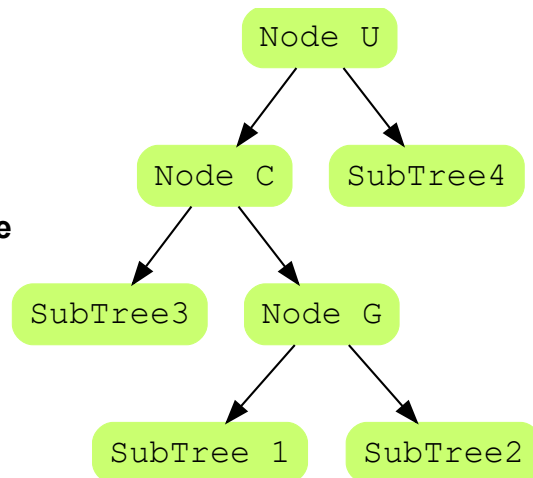
3 of 4

Rotate Node U
towards Right



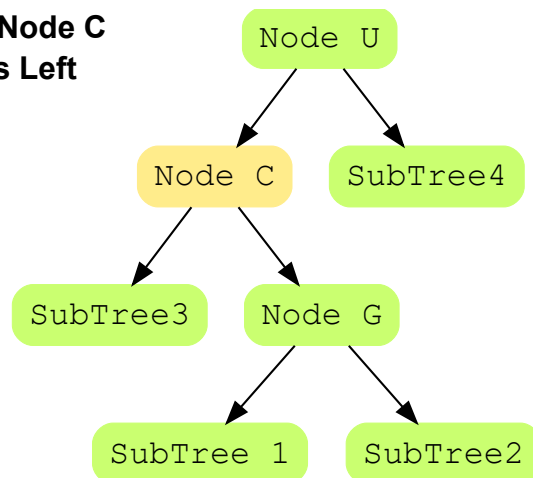
4 of 4

L-R Case



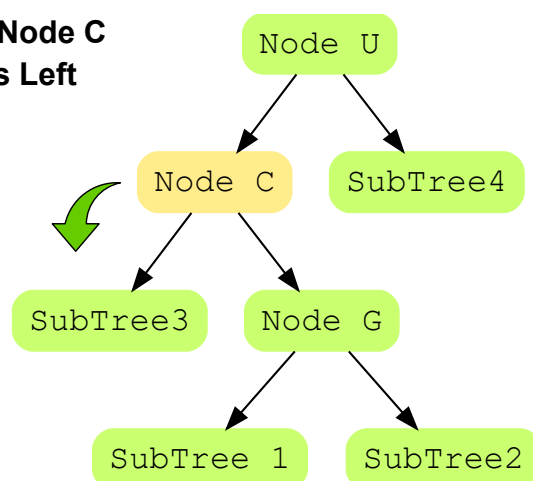
1 of 7

**Rotate Node C
towards Left**



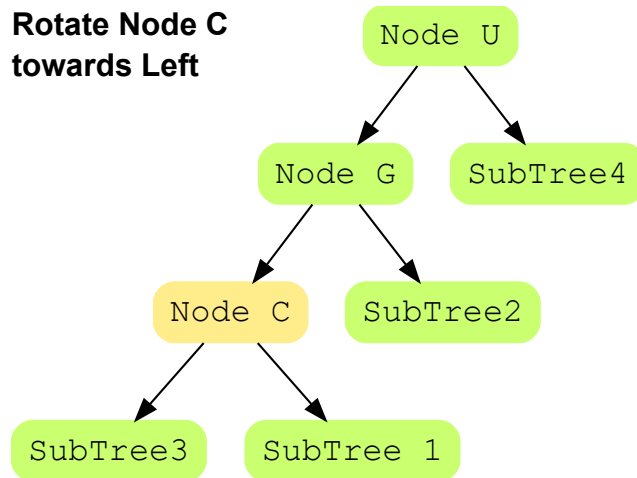
2 of 7

**Rotate Node C
towards Left**



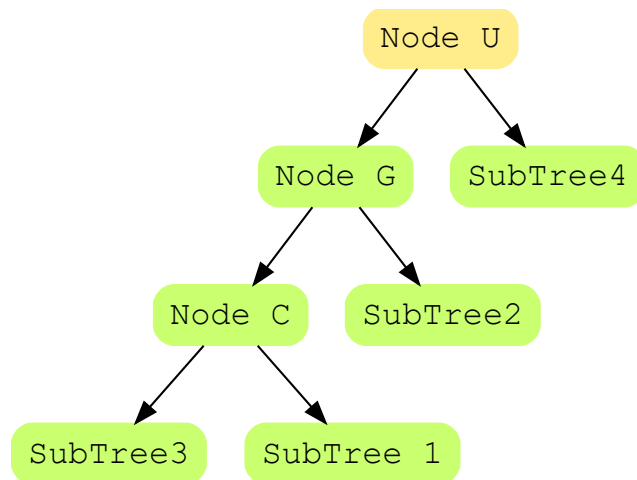
3 of 7

**Rotate Node C
towards Left**



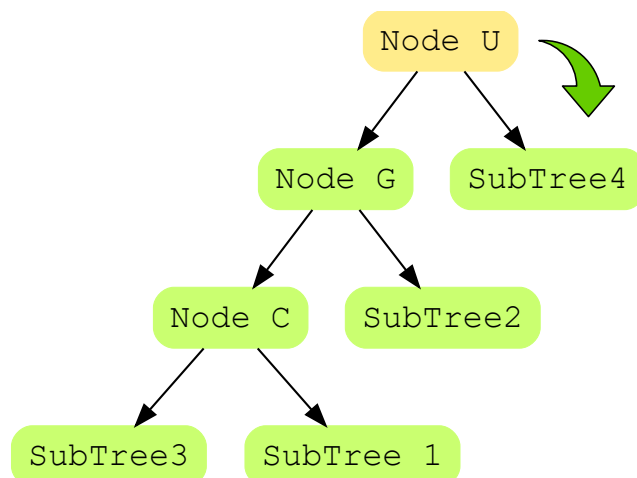
4 of 7

**Rotate Node U
towards Right**

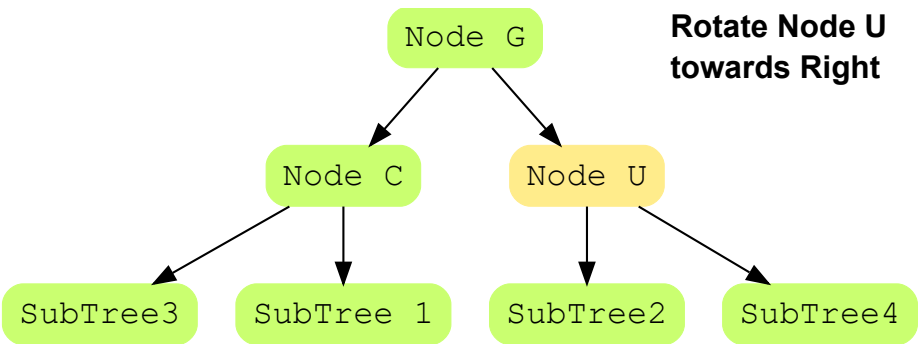


5 of 7

**Rotate Node U
towards Right**



6 of 7

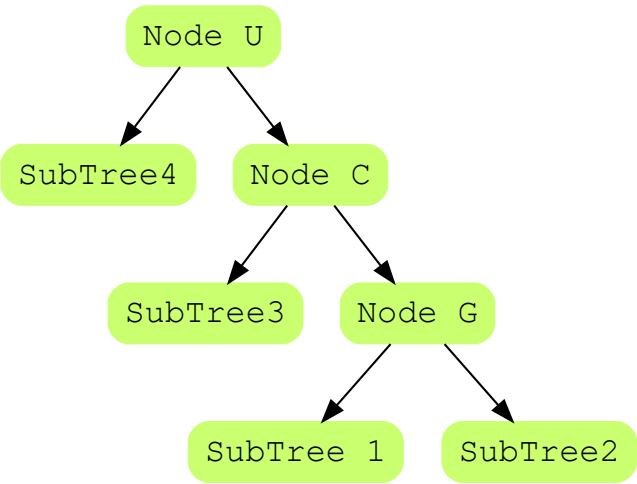


7 of 7

- []

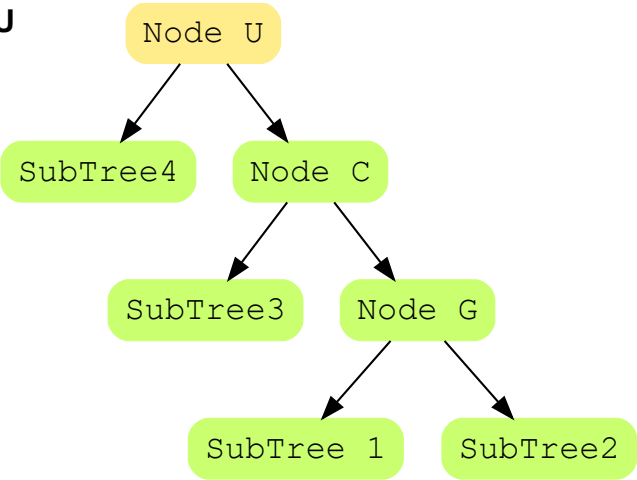
Case 3: Right-Right #

R-R Case



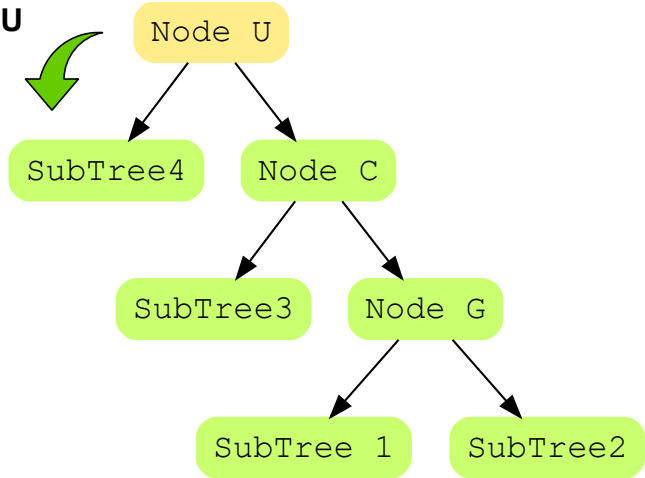
1 of 4

Rotate Node U
towards Left



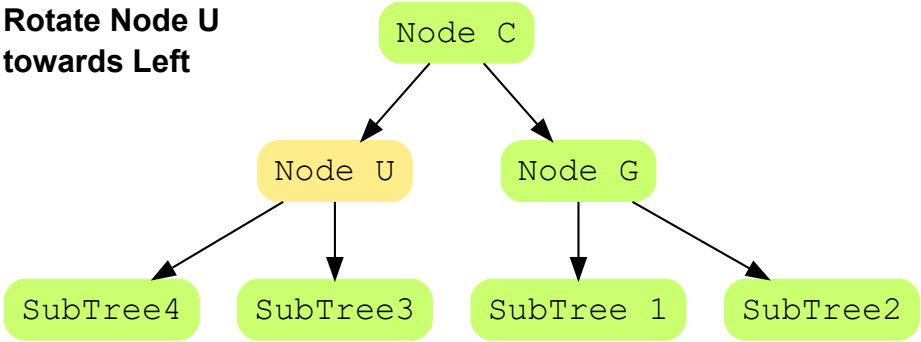
2 of 4

Rotate Node U
towards Left



3 of 4

Rotate Node U
towards Left

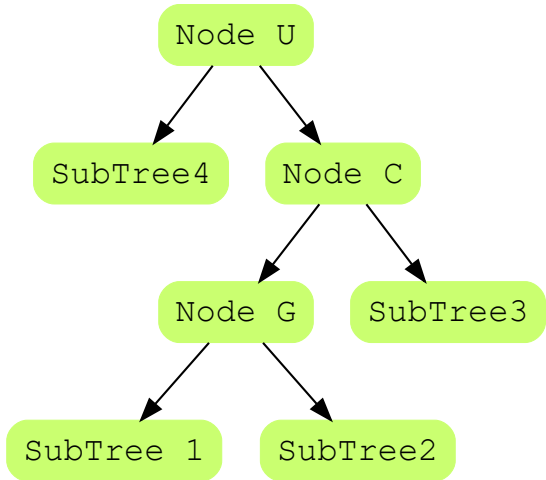


4 of 4

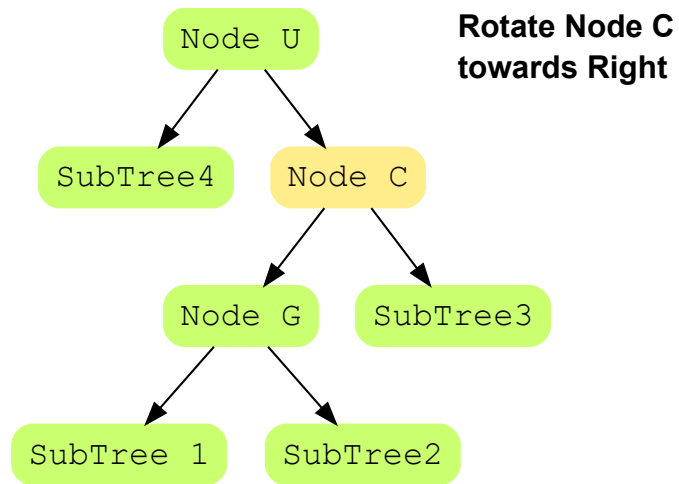
– []

Case 4: Right-Left #

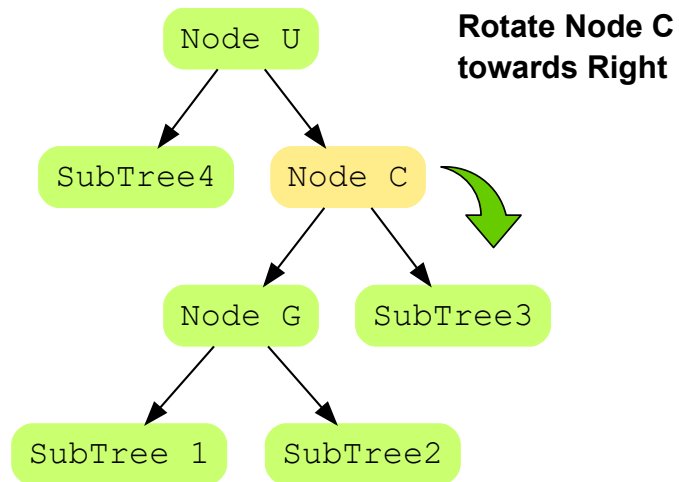
R-L Case



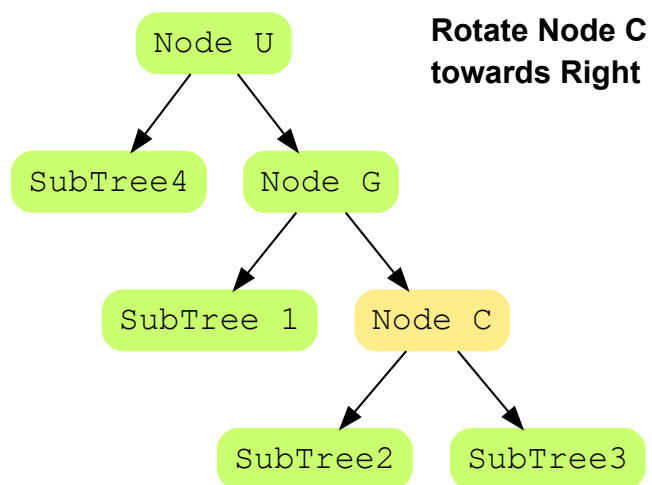
1 of 8



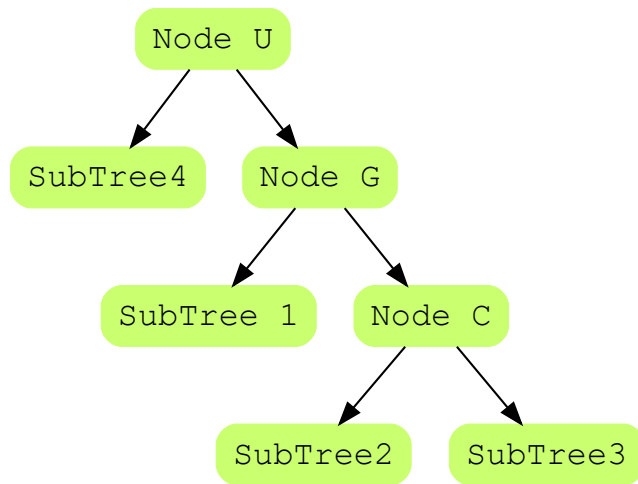
2 of 8



3 of 8

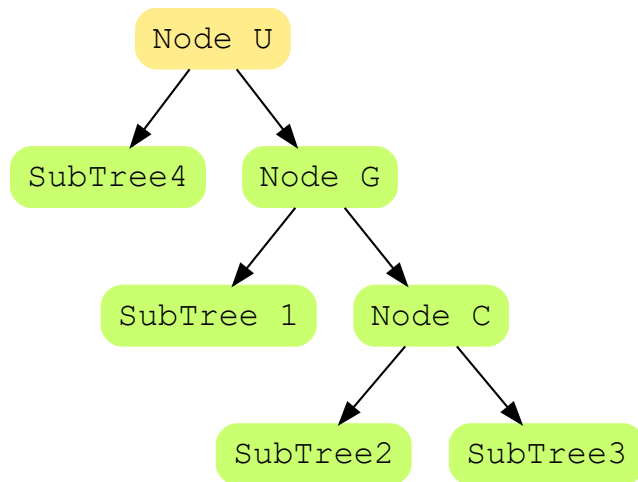


4 of 8



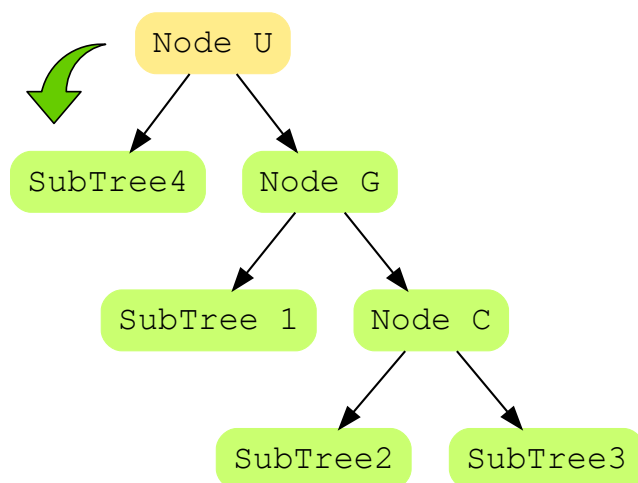
5 of 8

**Rotate Node U
towards Left**

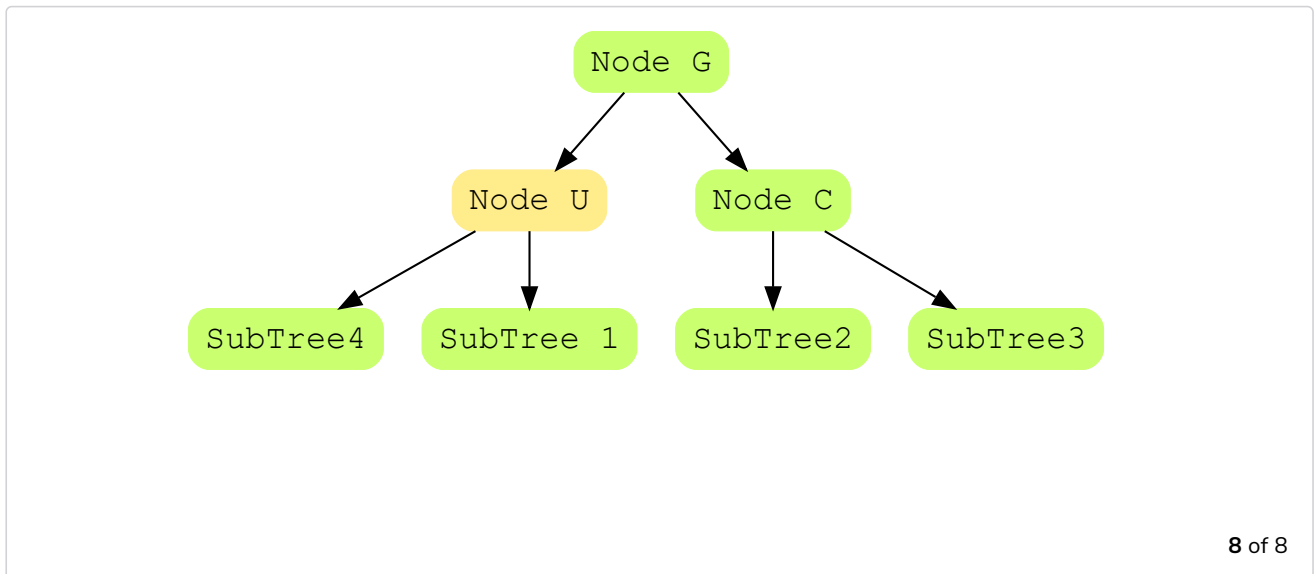


6 of 8

**Rotate Node U
towards Left**



7 of 8



— []

In the next lesson, we will study another variation of Binary Trees known as Red-Black Trees and we will go through how insertion and deletion work in a Red-Black Tree.


← Back

AVL Insertion

Next →

What is a Red-Black Tree?

☒ Mark as Completed

 Report an Issue

 Ask a Question

(https://discuss.educative.io/tag/avl-deletion__introduction-to-trees__data-structures-for-coding-interviews-in-python)