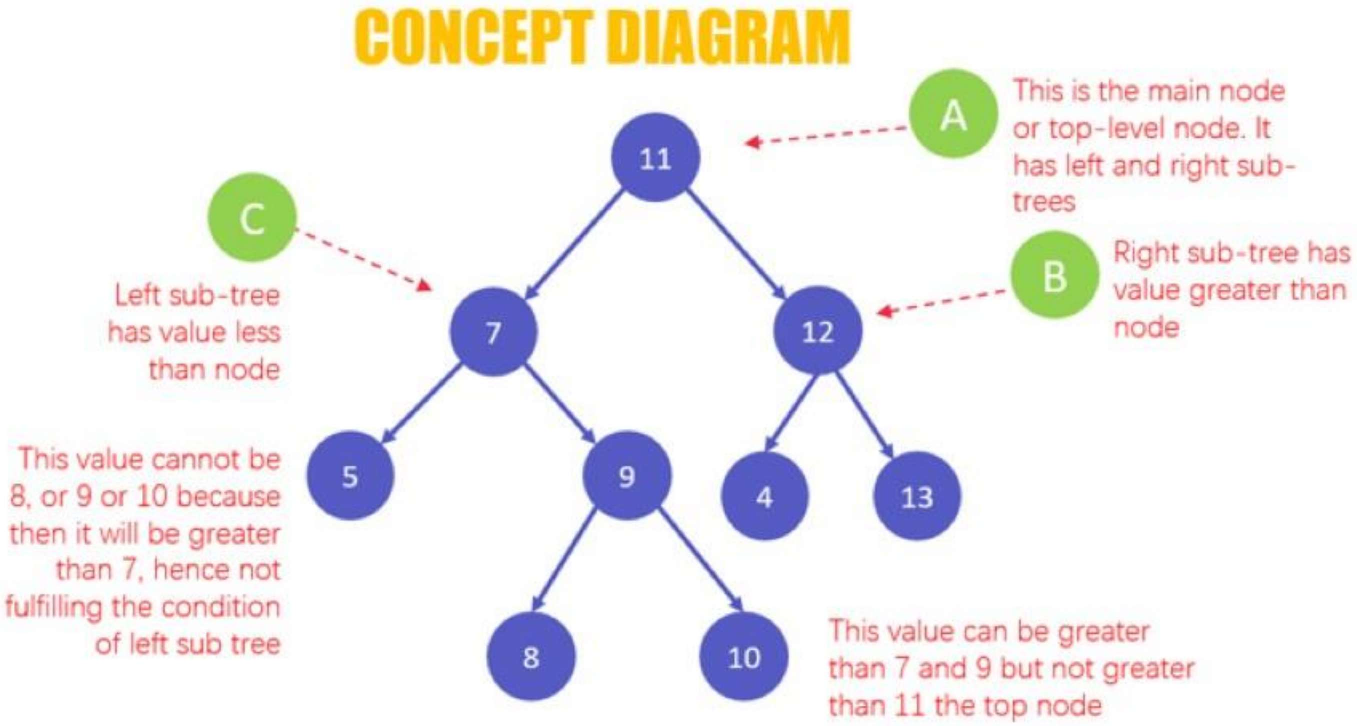
Binary Search Tree

What is a Binary Search Tree?

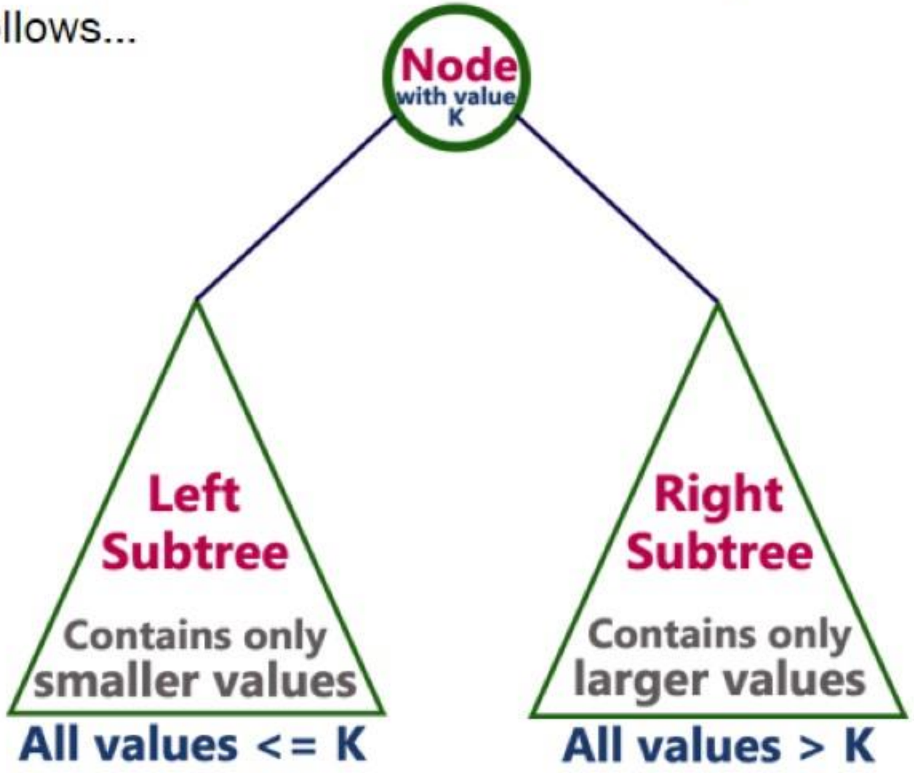
The binary search tree is an advanced algorithm used for analyzing the node, its left and right branches, which are modeled in a tree structure and returning the value. The BST is devised on the architecture of a basic binary search algorithm; hence it enables faster lookups, insertions, and removals of nodes. This makes the program really fast and accurate.


CONCEPT DIAGRAM
A
This is the main node or top-level node. It has left and right sub-trees
B
Right sub-tree has value greater than node
C
Left sub-tree has value less than node
This value cannot be 8, or 9 or 10 because then it will be greater than 7, hence not fulfilling the condition of left sub tree
This value can be greater than 7 and 9 but not greater than 11 the top node
11
7
12
5
9
4
13
8
10

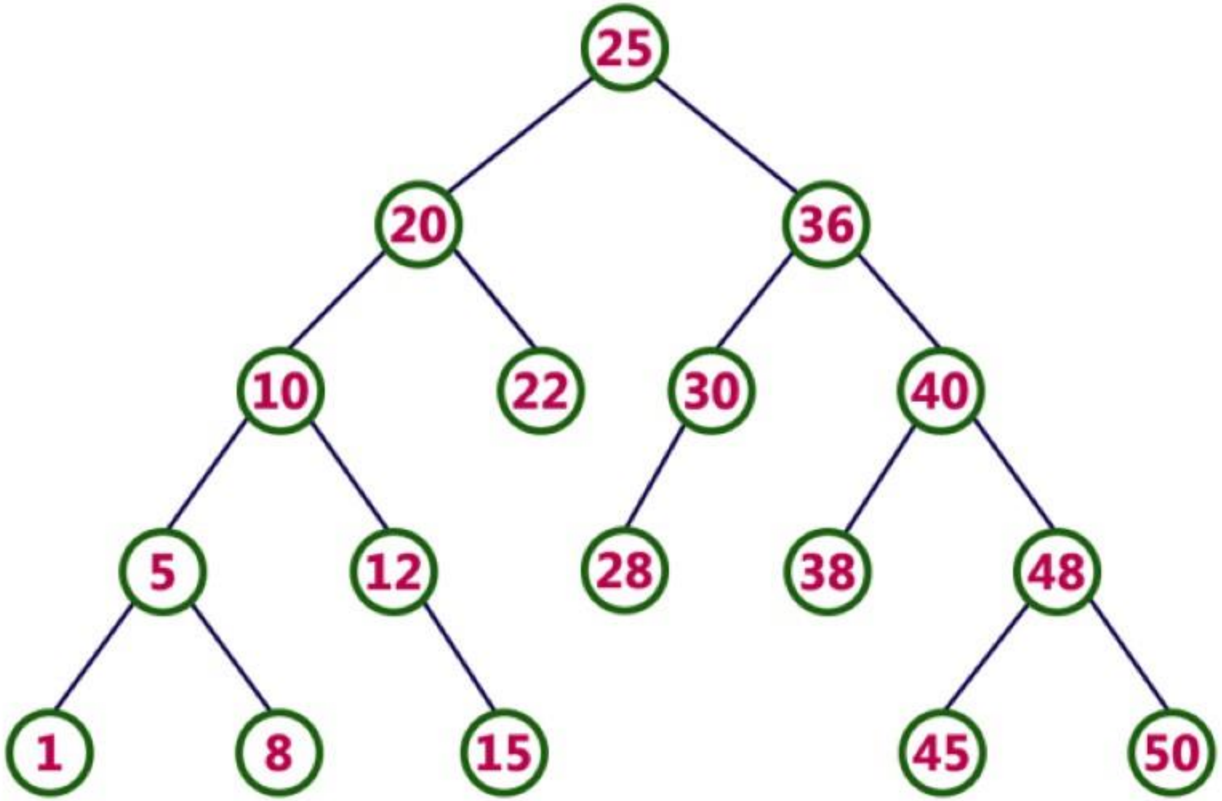
What is a Binary Search Tree?

The binary search tree is an advanced algorithm used for analyzing the node, its left and right branches, which are modeled in a tree structure and returning the value. The BST is devised on the architecture of a basic binary search algorithm; hence it enables faster lookups, insertions, and removals of nodes. This makes the program really fast and accurate.

To enhance the performance of binary tree, we use a special type of binary tree known as Binary Search Tree. Binary search tree mainly focuses on the search operation in a binary tree. Binary search tree can be defined as follows...



In a binary search tree, all the nodes in the left subtree of any node contains smaller values and all the nodes in the right subtree of any node contains larger values as shown in the following figure...



Operations on a Binary Search Tree

The following operations are performed on a binary search tree...

- Search
- Insertion
- Deletion

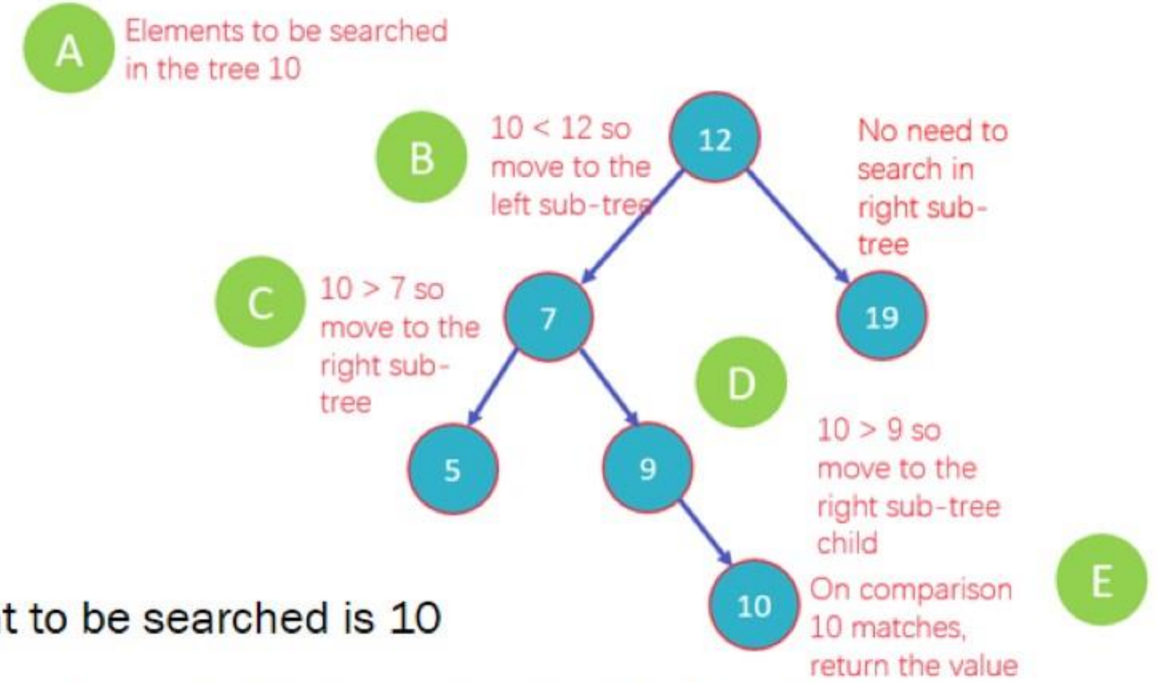
Example

The following tree is a Binary Search Tree. In this tree, left subtree of every node contains nodes with smaller values and right subtree of every node contains larger values.

Search Operation

Always initiate analyzing tree at the root node and then move further to either the right or left subtree of the root node depending upon the element to be located is either less or greater than the root

Search Operation



A
Elements to be searched in the tree 10
B
10 < 12 so move to the left sub-tree
12
No need to search in right sub-tree
C
10 > 7 so move to the right sub-tree
7
19
D
5
9
10 > 9 so move to the right sub-tree child
10
On comparison 10 matches, return the value
E

A. The element to be searched is 10

B. Compare the element with the root node 12, 10 < 12, hence you move to the left subtree. No need to analyze the right-subtree

C. Now compare 10 with node 7, 10 > 7, so move to the right-subtree

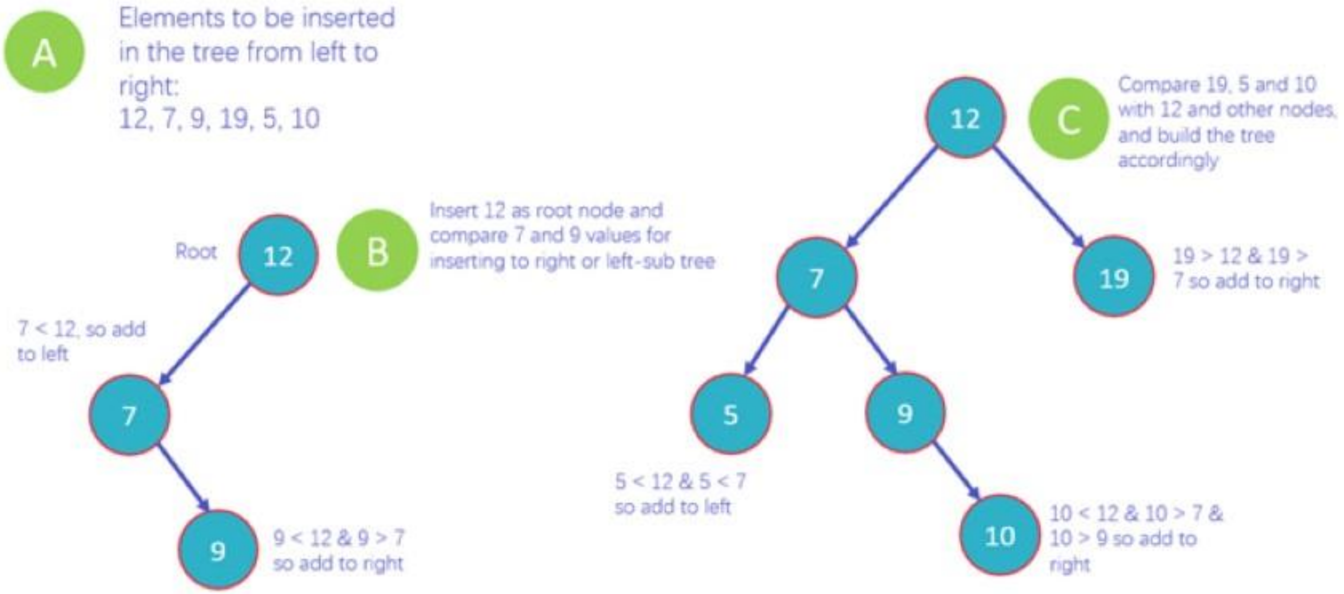
D. Then compare 10 with the next node, which is 9, 10 > 9, look in the right subtree child

E. 10 matches with the value in the node, 10 = 10, return the value to the user.

Insert Operation

This is a very straight forward operation. First, the root node is inserted, then the next value is compared with the root node. If the value is greater than root, it is added to the right subtree, and if it is lesser than the root, it is added to the left subtree.

Insert Operation


A
Elements to be inserted in the tree from left to right:
12, 7, 9, 19, 5, 10
Root
12
B
Insert 12 as root node and compare 7 and 9 values for inserting to right or left-sub tree
7 < 12, so add to left
7
9
9 < 12 & 9 > 7 so add to right
12
C
Compare 19, 5 and 10 with 12 and other nodes, and build the tree accordingly
7
19
19 > 12 & 19 > 7 so add to right
5
9
5 < 12 & 5 < 7 so add to left
10
10 < 12 & 10 > 7 & 10 > 9 so add to right

A. There is a list of 6 elements that need to be inserted in a BST in order from left to right

B. Insert 12 as the root node and compare next values 7 and 9 for inserting accordingly into the right and left subtree

C. Compare the remaining values 19, 5, and 10 with the root node 12 and place them accordingly. 19 > 12 place it as the right child of 12, 5 < 12 & 5 < 7, hence place it as left child of 7. Now compare 10, 10 is < 12 & 10 is > 7 & 10 is > 9, place 10 as right subtree of 9.

Delete Operations

For deleting a node from a BST, there are some cases, i.e. deleting a root or deleting a leaf node. Also, after deleting a root, we need to think about the root node.

Say we want to delete a leaf node, we can just delete it, but if we want to delete a root, we need to replace the root's value with another node. Let's take the following example:

Case 1- Node with zero children: this is the easiest situation, you just need to delete the node which has no further children on the right or left.

Case 2 – Node with one child: once you delete the node, simply connect its child node with the parent node of the deleted value..

Case 3 Node with two children: this is the most difficult situation, and it works on the following two rules

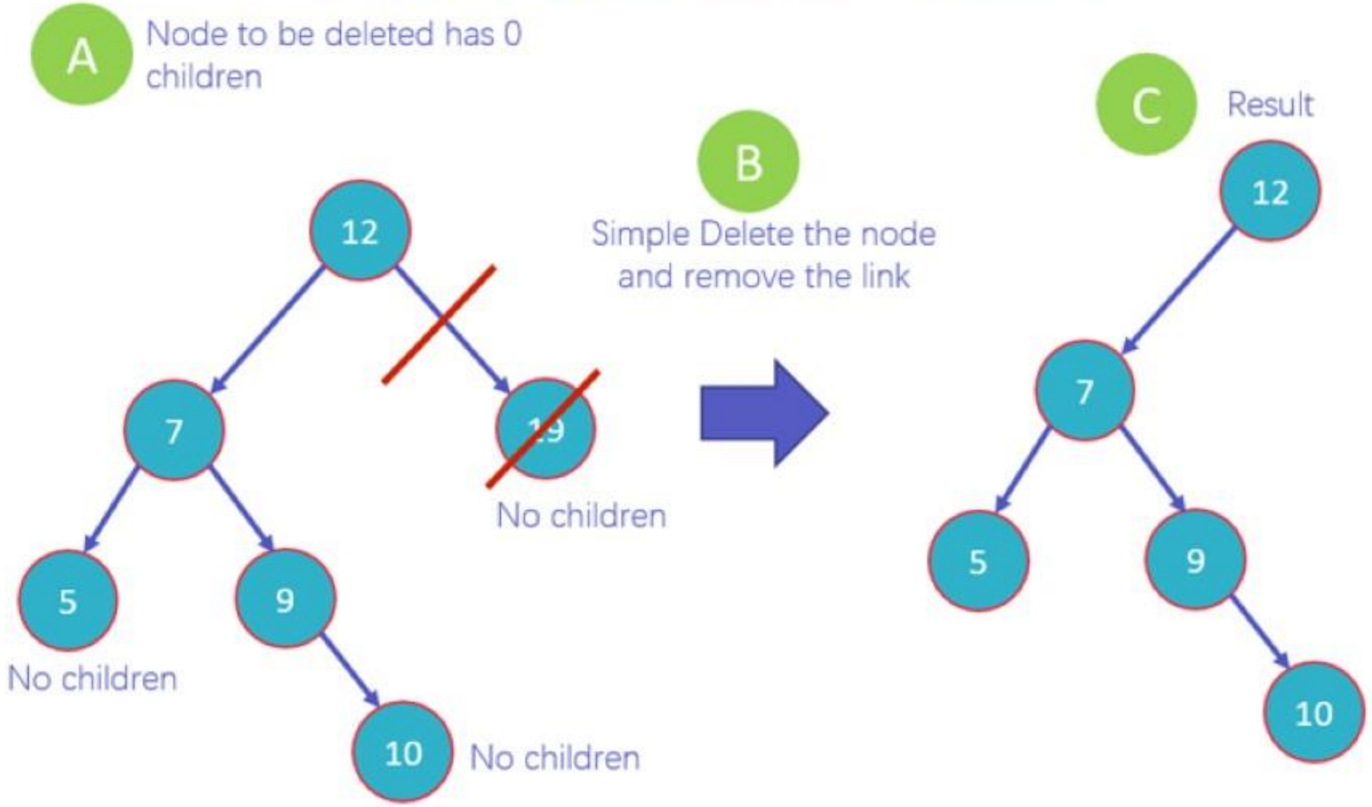
- 3a – In Order Predecessor: you need to delete the node with two children and replace it with the largest value on the left-subtree of the deleted node
- 3b – In Order Successor: you need to delete the node with two children and replace it with the largest value on the right-subtree of the deleted node

A. This is the first case of deletion in which you delete a node that has no children. As you can see in the diagram that 19, 10 and 5 have no children. But we will delete 19.

B. Delete the value 19 and remove the link from the node.

C. View the new structure of the BST without 19

Delete Operation – Case 1



A Node to be deleted has 0 children

B Simple Delete the node and remove the link

C Result

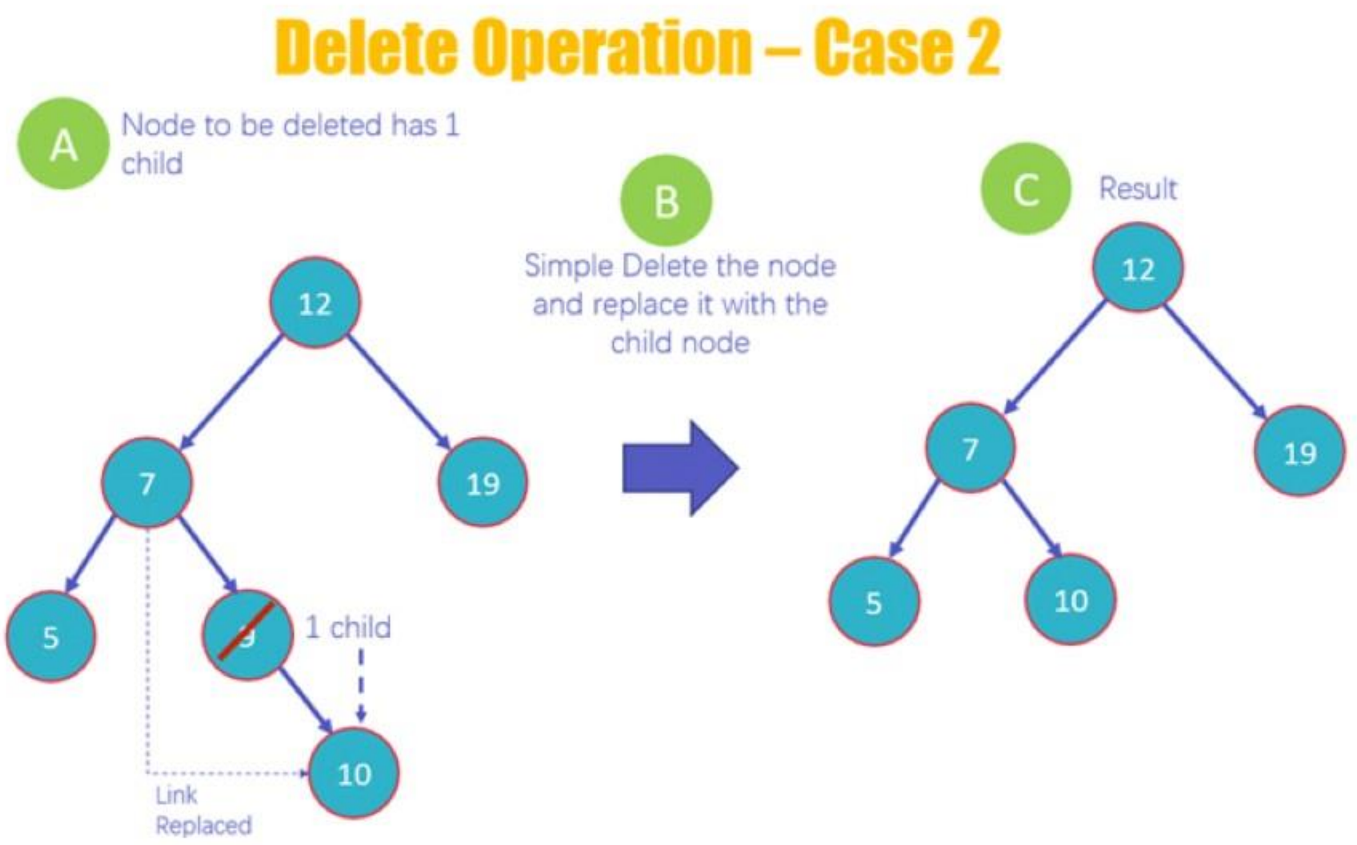
No children

No children

No children

Case 1

Node with zero children: this is the easiest situation, you just need to delete the node which has no further children on the right or left.



Delete Operation – Case 2

A Node to be deleted has 1 child

B Simple Delete the node and replace it with the child node

C Result

A. This is the second case of deletion in which you delete a node that has 1 child, as you can see in the diagram that 9 has one child.
B. Delete the node 9 and replace it with its child 10 and add a link from 7 to 10
C. View the new structure of the BST without 9

Case 2

Node with one child: once you delete the node, simply connect its child node with the parent node of the deleted value..

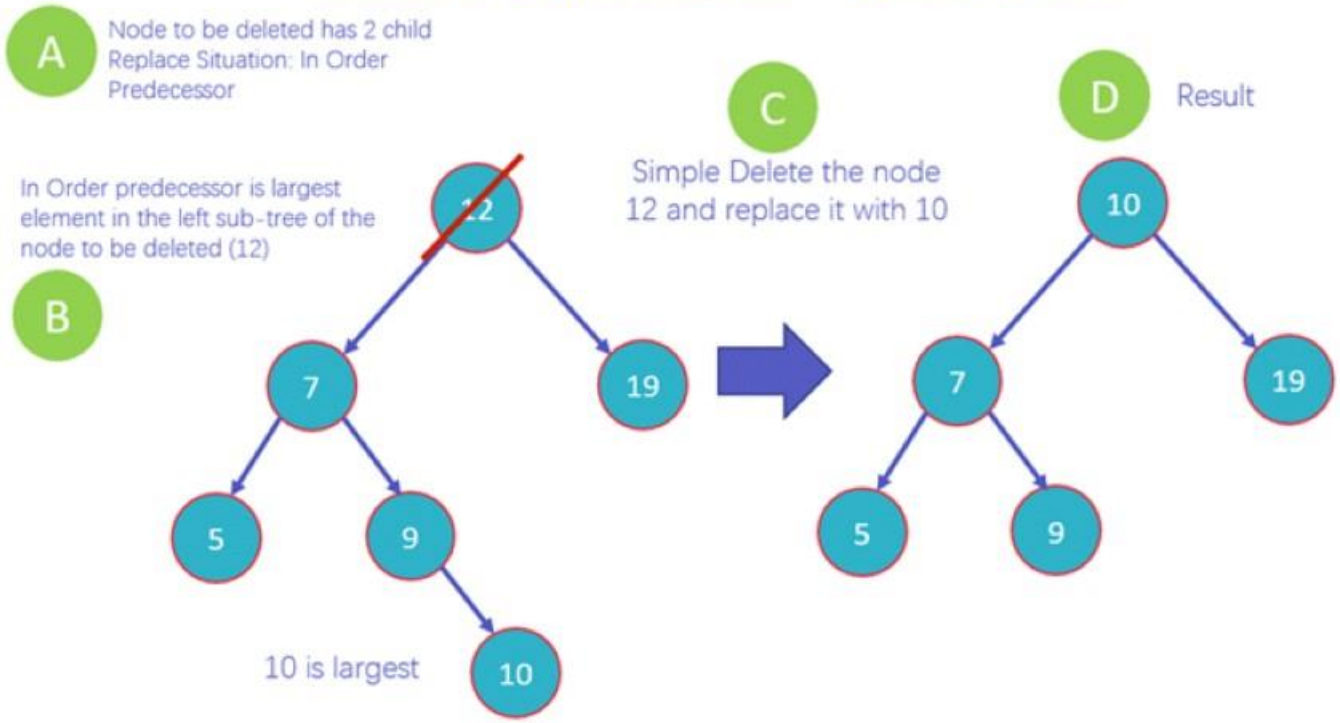
A. Here you will be deleting the node 12 that has two children

B. The deletion of the node will occur based upon the in order predecessor rule, which means that the largest element on the left subtree of 12 will replace it.

C. Delete the node 12 and replace it with 10 as it is the largest value on the left subtree

D. View the new structure of the BST after deleting 12

Delete Operation – Case 3 (a)



A Node to be deleted has 2 child Replace Situation: In Order Predecessor

In Order predecessor is largest element in the left sub-tree of the node to be deleted (12)

B

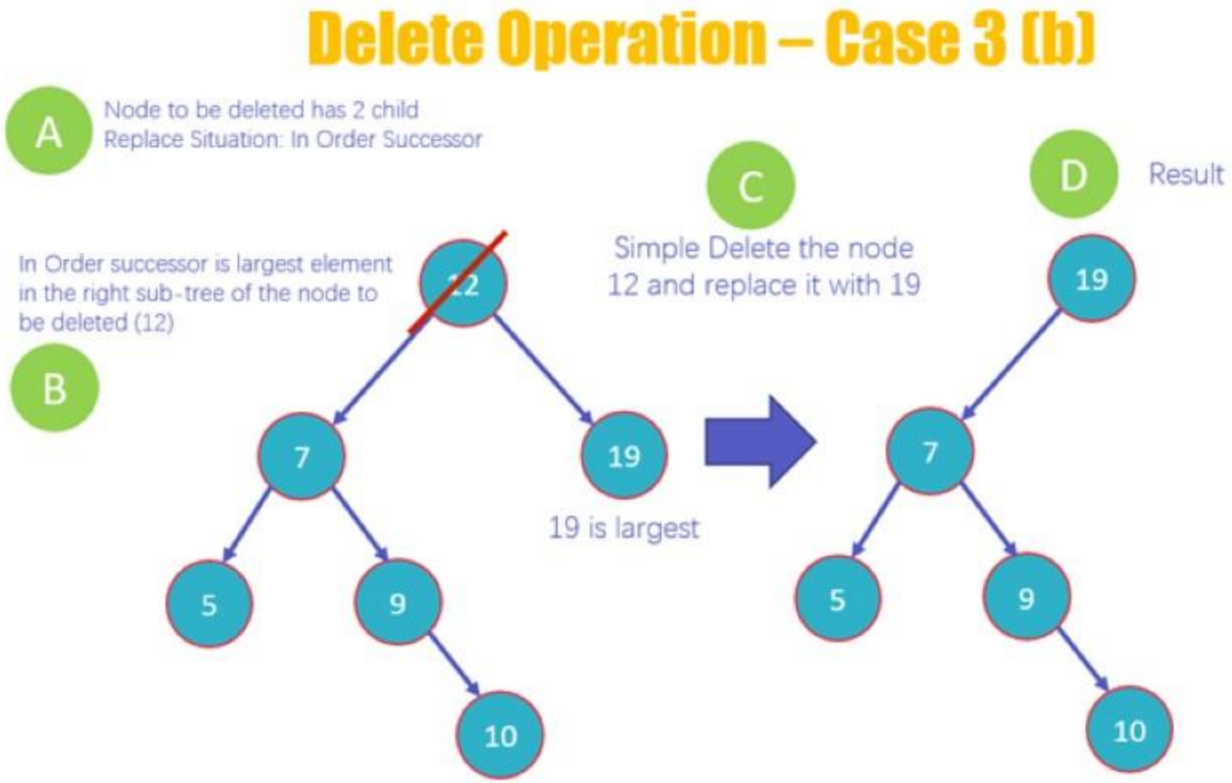
10 is largest

C Simple Delete the node 12 and replace it with 10

D Result

Case 3A

3a – In Order Predecessor: you need to delete the node with two children and replace it with the largest value on the left-subtree of the deleted node



Delete Operation – Case 3 (b)

A Node to be deleted has 2 child
Replace Situation: In Order Successor

B In Order successor is largest element in the right sub-tree of the node to be deleted (12)

C Simple Delete the node 12 and replace it with 19

19 is largest

D Result

A. Delete a node 12 that has two children

B. The deletion of the node will occur based upon the In Order Successor rule, which means that the largest element on the right subtree of 12 will replace it

C. Delete the node 12 and replace it with 19 as it is the largest value on the right subtree

D. View the new structure of the BST after deleting 12

Case 3B

3b – In Order Successor: you need to delete the node with two children and replace it with the largest value on the right-subtree of the deleted node

AVL Trees

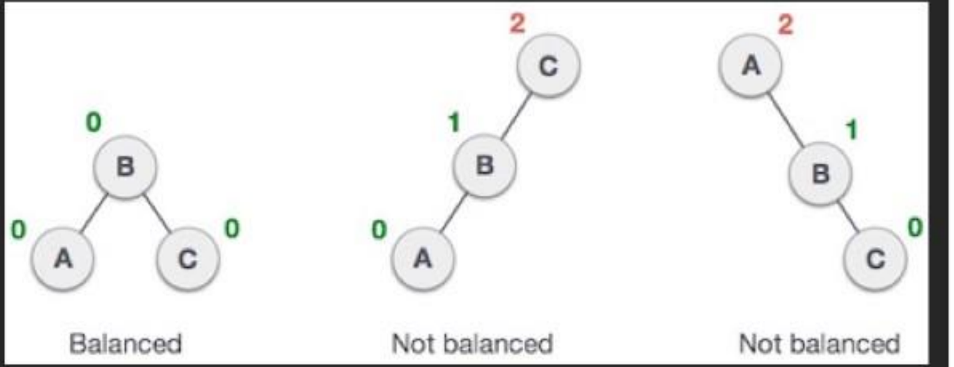
What are AVL Trees?

AVL trees are binary search trees in which the difference between the height of the left and right subtree is either -1, 0, or +1.

AVL trees are also called a self-balancing binary search tree. These trees help to maintain the logarithmic search time. It is named after its inventors (AVL) Adelson, Velsky, and Landi.

Balance Factor

Here we see that the first tree is balanced and the next two trees are not balanced –



In the second tree, the left subtree of C has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of A has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

AVL Rotations

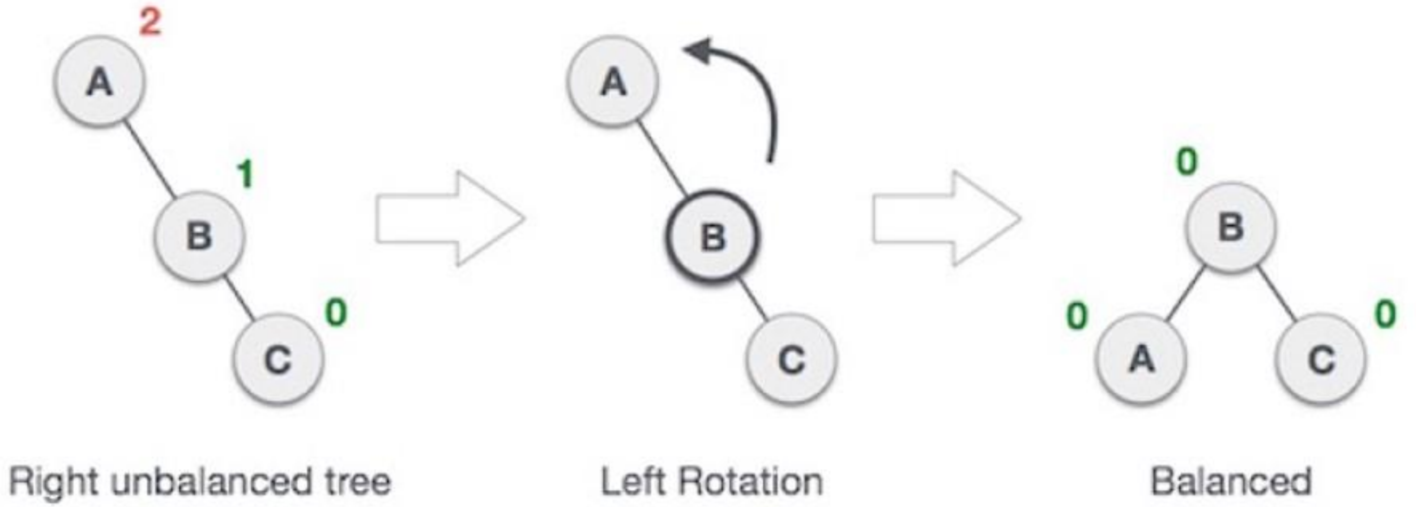
To balance itself, an AVL tree may perform the following four kinds of rotations –

- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation

The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

Left Rotation

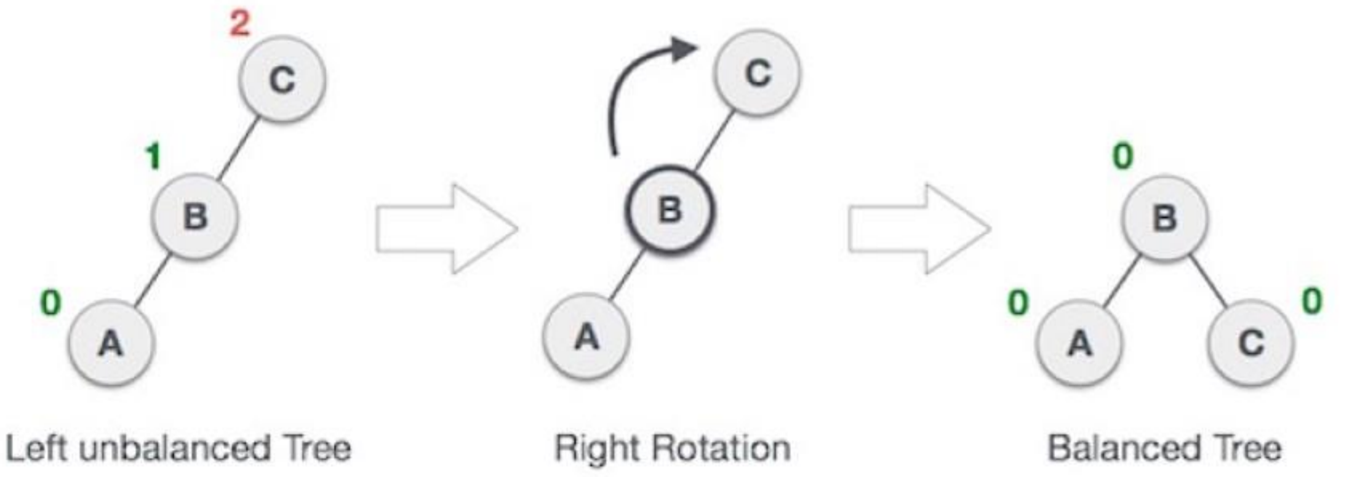
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation –



In our example, node A has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making A the left-subtree of B.

Right Rotation

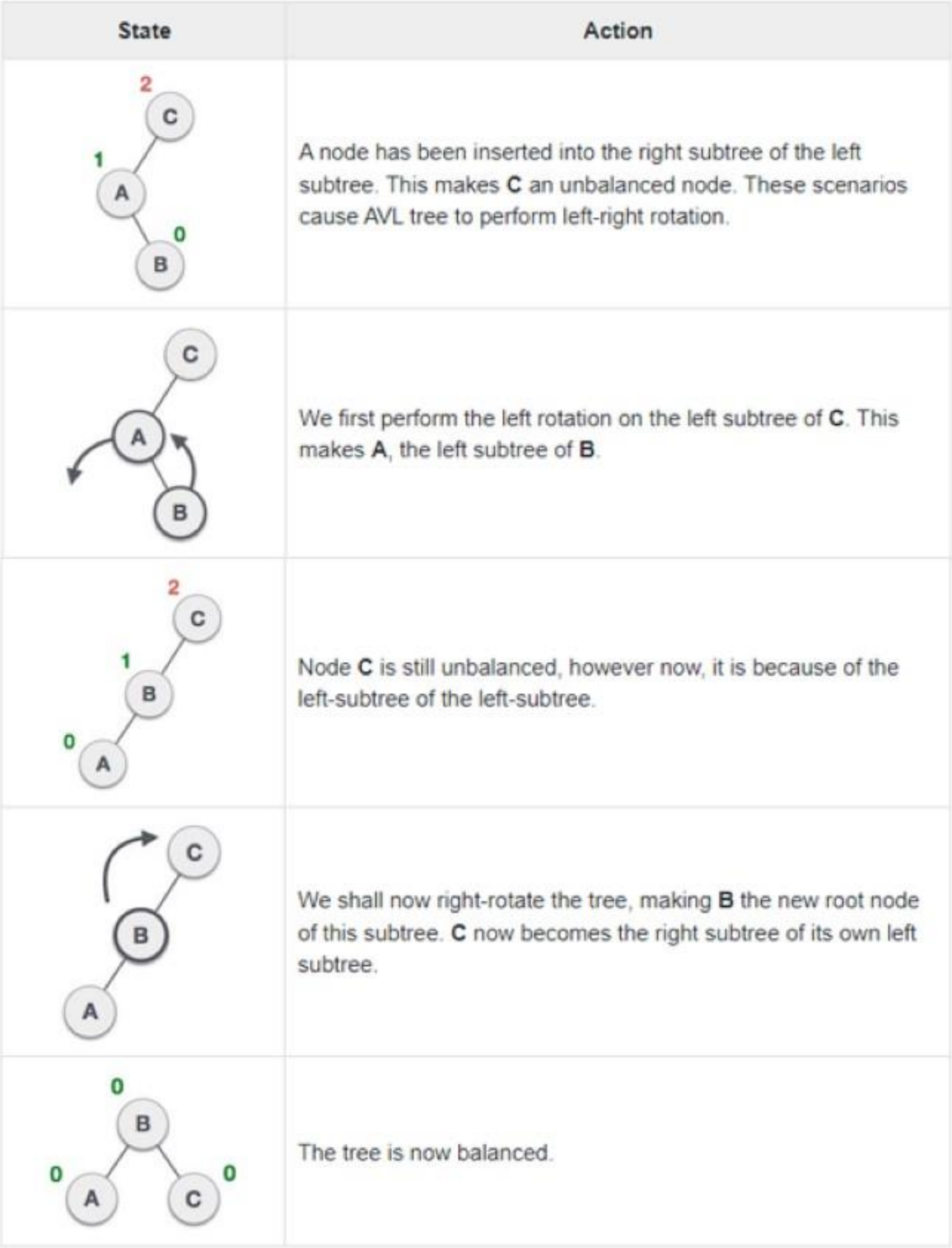
AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.



As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

Left-Right Rotation

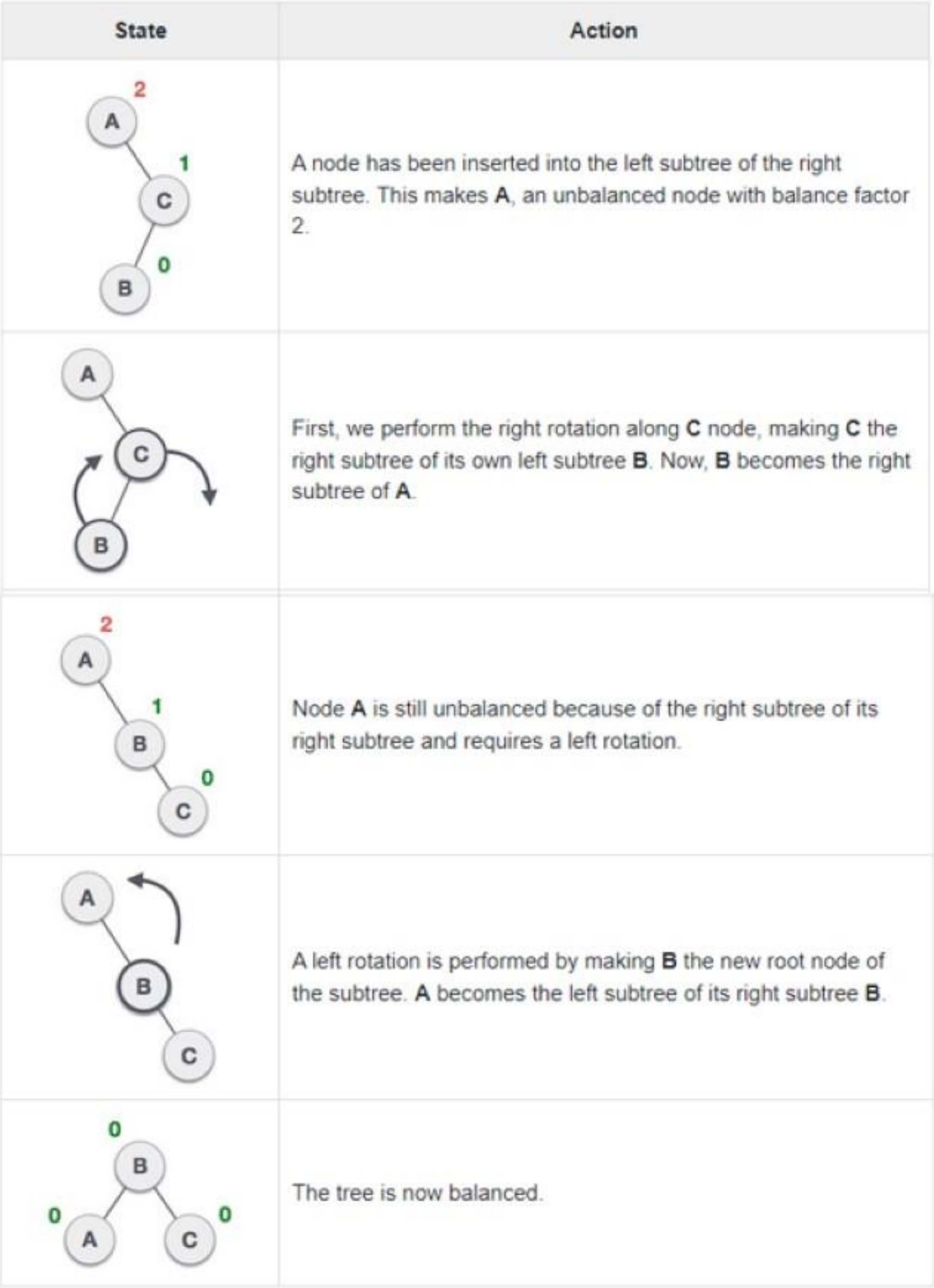
Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.



State	Action
2 C 1 A 0 B	A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.
C A B	We first perform the left rotation on the left subtree of **C**. This makes **A**, the left subtree of **B**.
2 C 1 B 0 A	Node **C** is still unbalanced, however now, it is because of the left-subtree of the left-subtree.
C B A	We shall now right-rotate the tree, making **B** the new root node of this subtree. **C** now becomes the right subtree of its own left subtree.
0 B 0 A C 0	The tree is now balanced.

Right-Left Rotation

The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.


2
A
1
C
0
B
A
C
B
2
A
1
B
0
C
A
B
C
0
B
0
A
C
0

State	Action
	A node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2.
	First, we perform the right rotation along **C** node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes the right subtree of **A**.
	Node **A** is still unbalanced because of the right subtree of its right subtree and requires a left rotation.
	A left rotation is performed by making **B** the new root node of the subtree. **A** becomes the left subtree of its right subtree **B**.
	The tree is now balanced.

Youtube Tutodials:

- [mycodeschool]. (2014, January 25). Data structures: Binary Search Tree. YouTube. Retrieved September 22, 2022, from https://www.youtube.com/watch?v=pYT9F8_LFTM
- [Back To Back SWE]. (2020, April 28). AVL Trees & Rotations (Self-Balancing Binary Search Trees). YouTube. Retrieved September 22, 2022, from https://www.youtube.com/watch?v=vRwi_UcZGjU

Source:

- Walker, A. (2022, September 3). Binary Search Tree (BST) with Example. Guru99. Retrieved September 22, 2022, from https://www.guru99.com/binary-search-tree-data-structure.html
- Data Structure and Algorithms - AVL Trees. (n.d.). Retrieved September 22, 2022, from https://www.tutorialspoint.com/data_structures_algorithms/avl_tree_algorithm.htm