Advanced Trees

C Manasa

Asst. professor

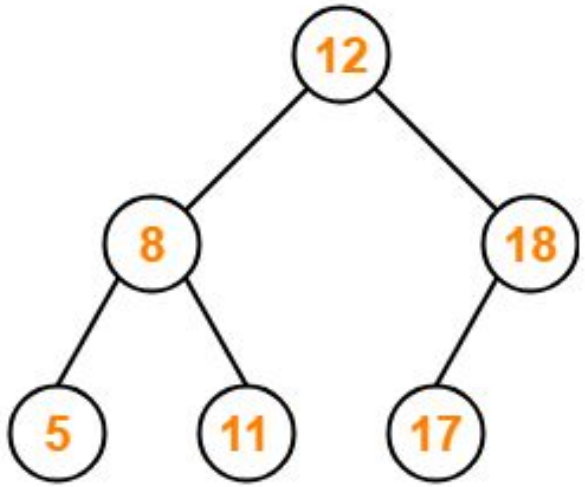
Dept. of ISE

Contents

- AVL Trees
- B+ Trees

AVL Trees

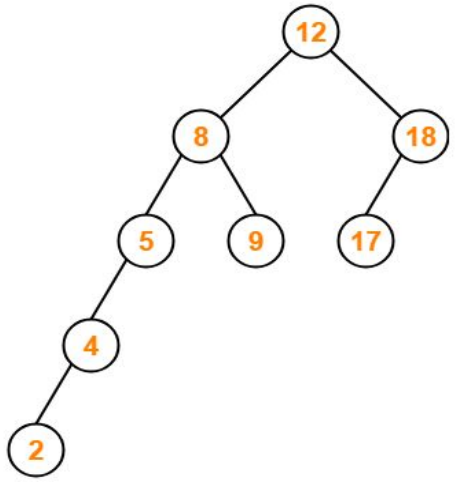
- AVL trees are special kind of binary search trees.
- In AVL trees, height of left subtree and right subtree of every node differs by at most one.
- AVL trees are also called as **self-balancing binary search trees**.



AVL Tree Example

AVL Trees

Following tree is not an example of AVL Tree-



Not an AVL Tree

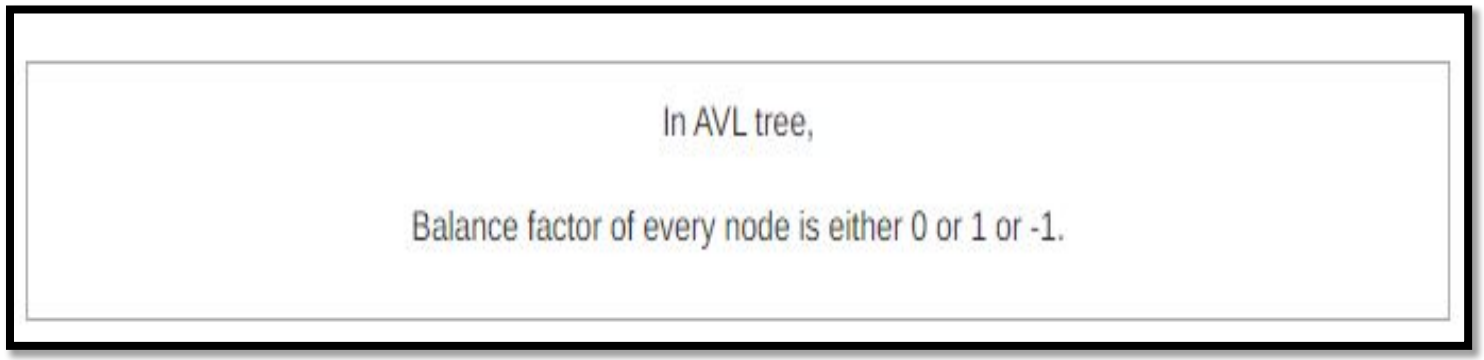
This tree is not an AVL tree because-

- The difference between height of left subtree and right subtree of root node = 4 – 2 = 2.
- This difference is greater than one.

Balance Factor

In AVL tree,

- Balance factor is defined for every node.
- Balance factor of a node = Height of its left subtree – Height of its right subtree



AVL Tree Operations

Like BST Operations, commonly performed operations on AVL tree are-

- Search Operation
- Insertion Operation
- Deletion Operation

After performing any operation on AVL tree, the balance factor of each node is checked.

There are following two cases possible-

Case-01:

After the operation, the balance factor of each node is either 0 or 1 or -1. In this case, the AVL tree is considered to be balanced. The operation is concluded.

Case-02:

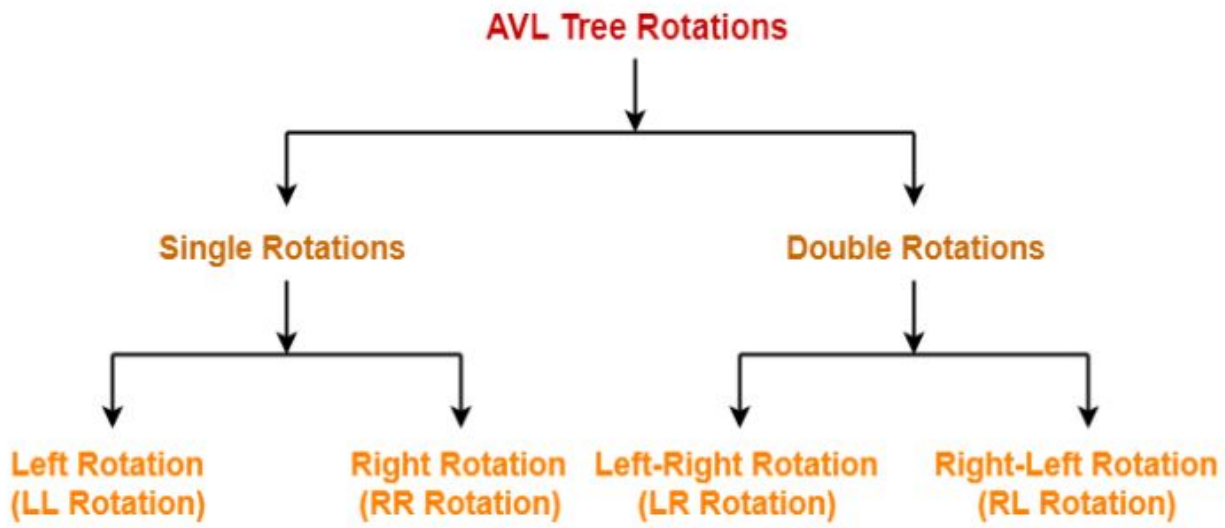
After the operation, the balance factor of at least one node is not 0 or 1 or -1. In this case, the AVL tree is considered to be imbalanced. Rotations are then performed to balance the tree.

AVL Tree Rotations

Rotation is the process of moving the nodes to make tree balanced.

Kinds of Rotations:

There are 4 kinds of rotations possible in AVL Trees-

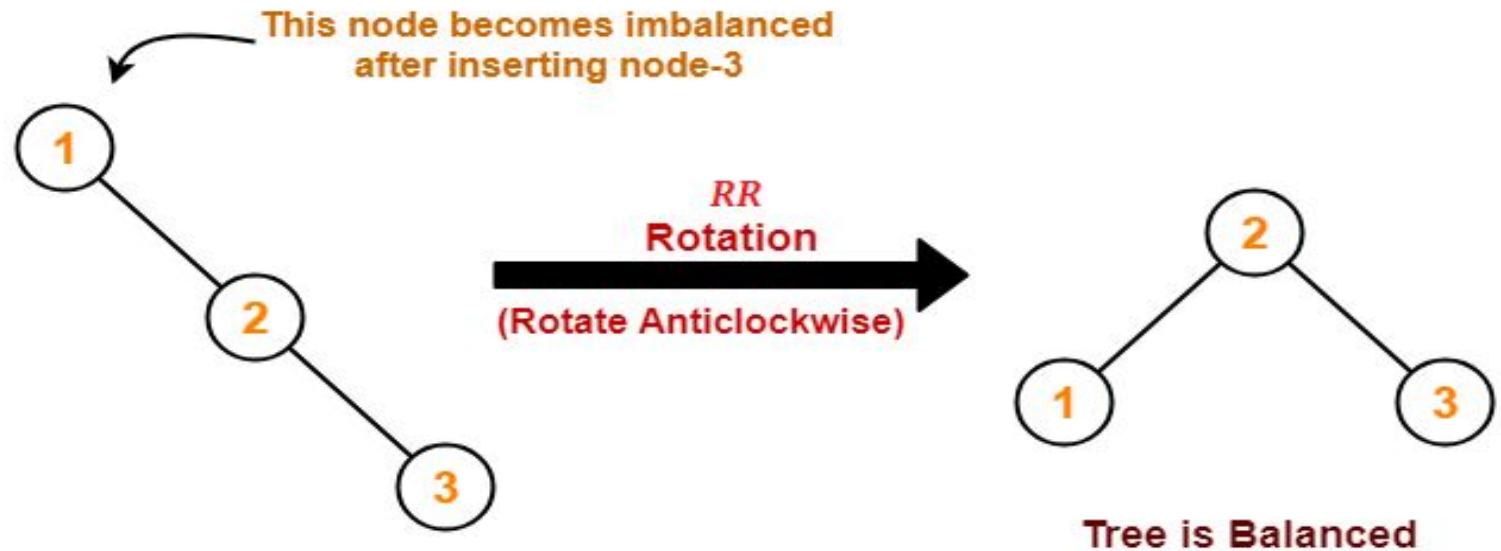


1. Left Rotation (LL Rotation)
2. Right Rotation (RR Rotation)
3. Left-Right Rotation (LR Rotation)
4. Right-Left Rotation (RL Rotation)

AVL Tree Rotations

Cases Of Imbalance And Their Balancing Using Rotation Operations-

Case-01:



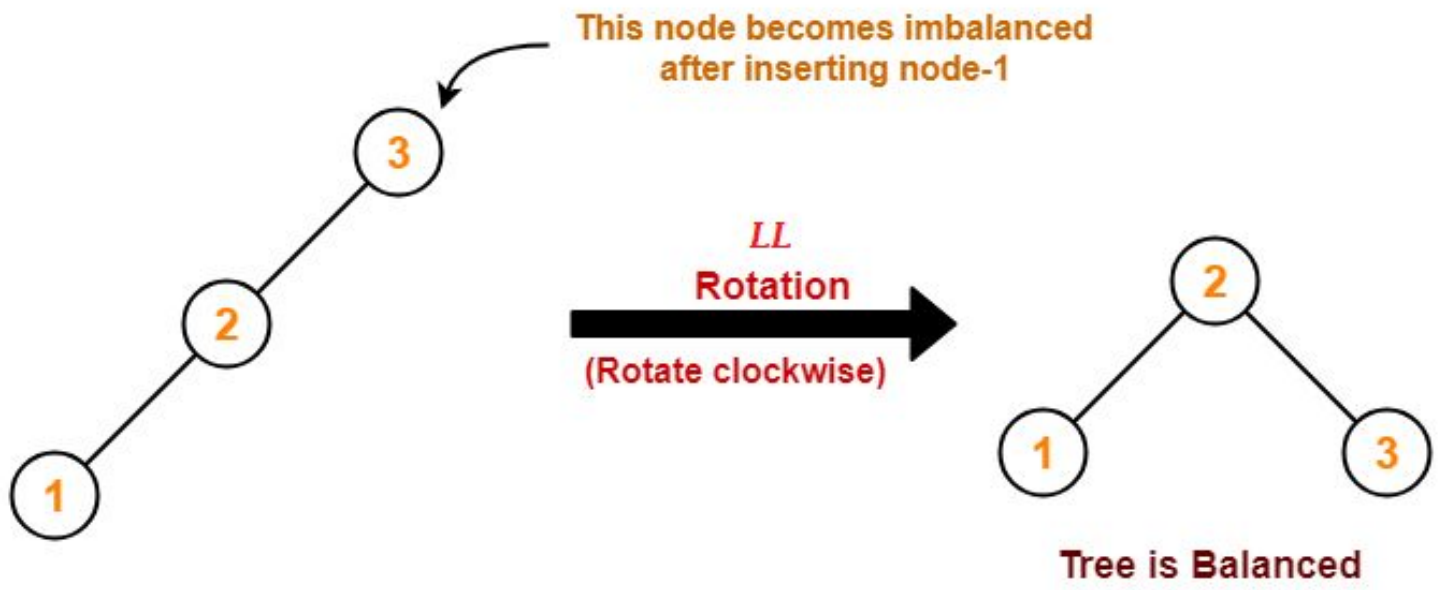
Tree is Balanced

Insertion Order : 1 , 2 , 3

Tree is Imbalanced

AVL Tree Rotations

Case-02:

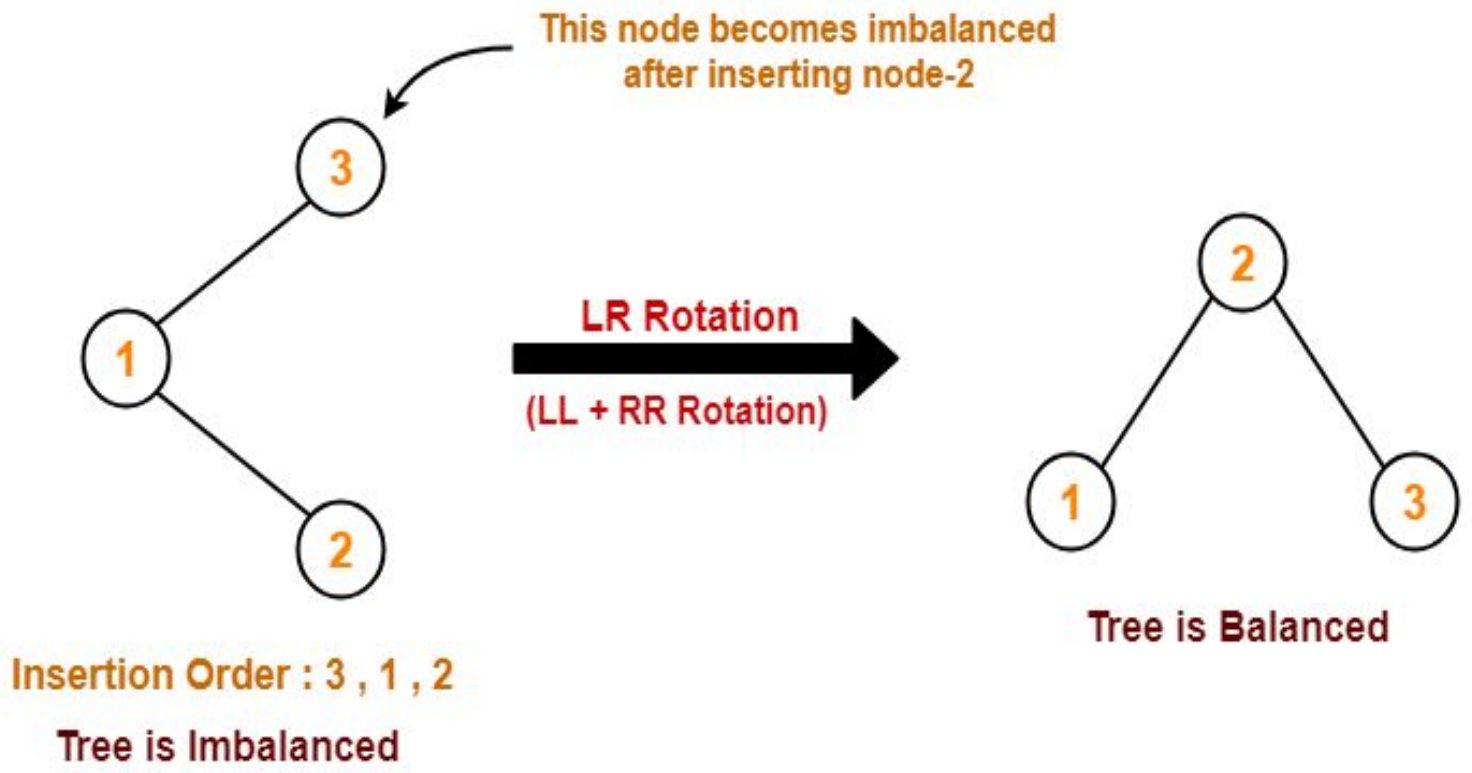


Insertion Order : 3 , 2 , 1

Tree is Imbalanced

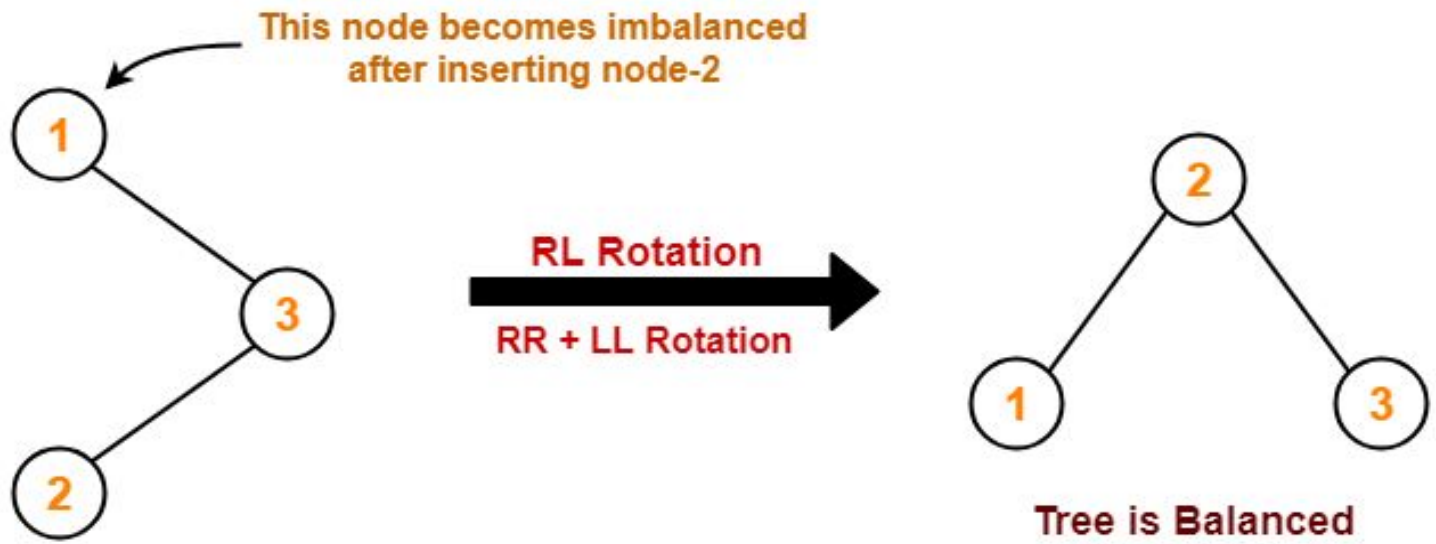
AVL Tree Rotations

Case-03:



AVL Tree Rotations

Case-04:



Insertion Order : 1 , 3 , 2

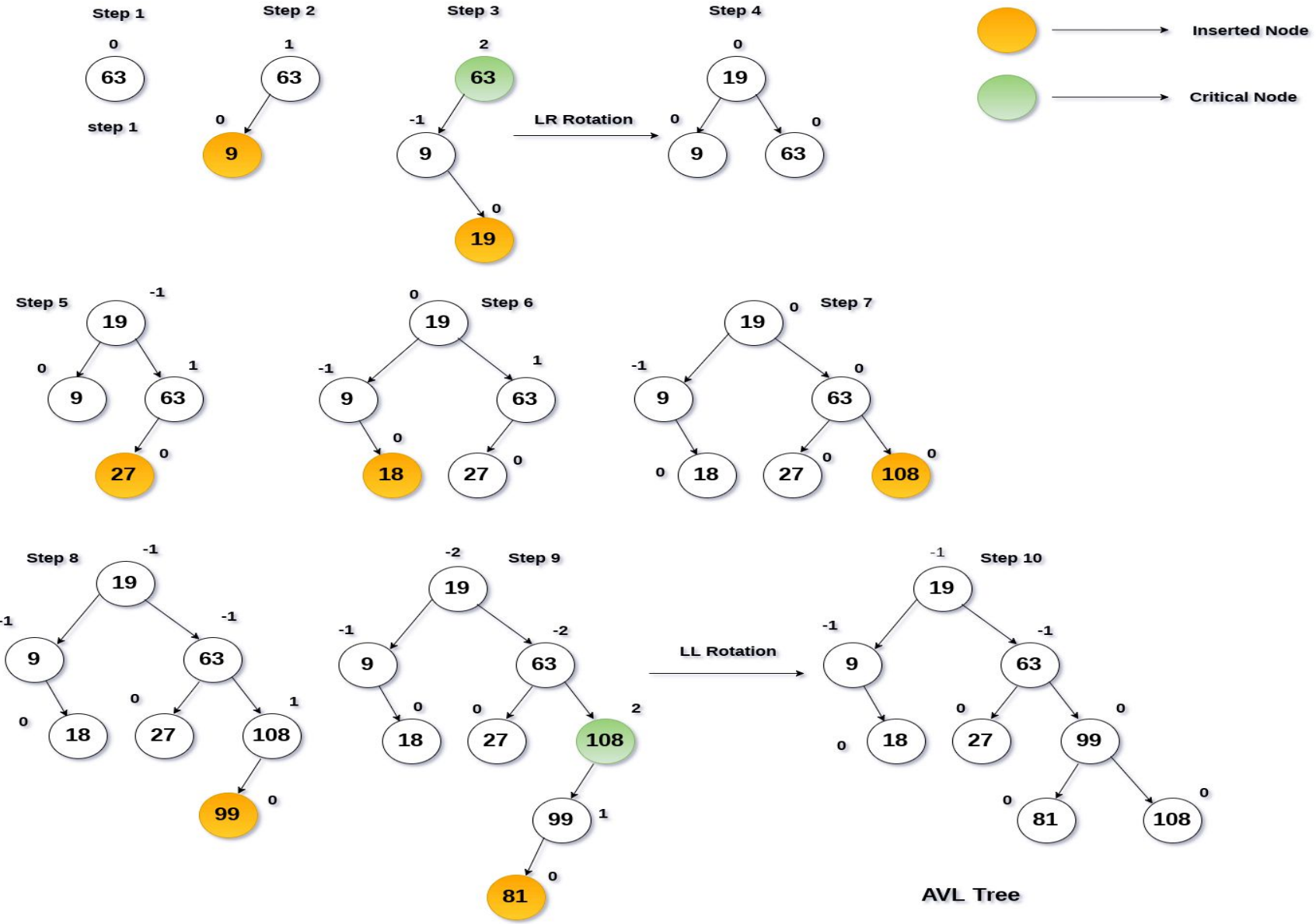
Tree is Imbalanced

Example

Construct an AVL tree by inserting the following elements in the given order.

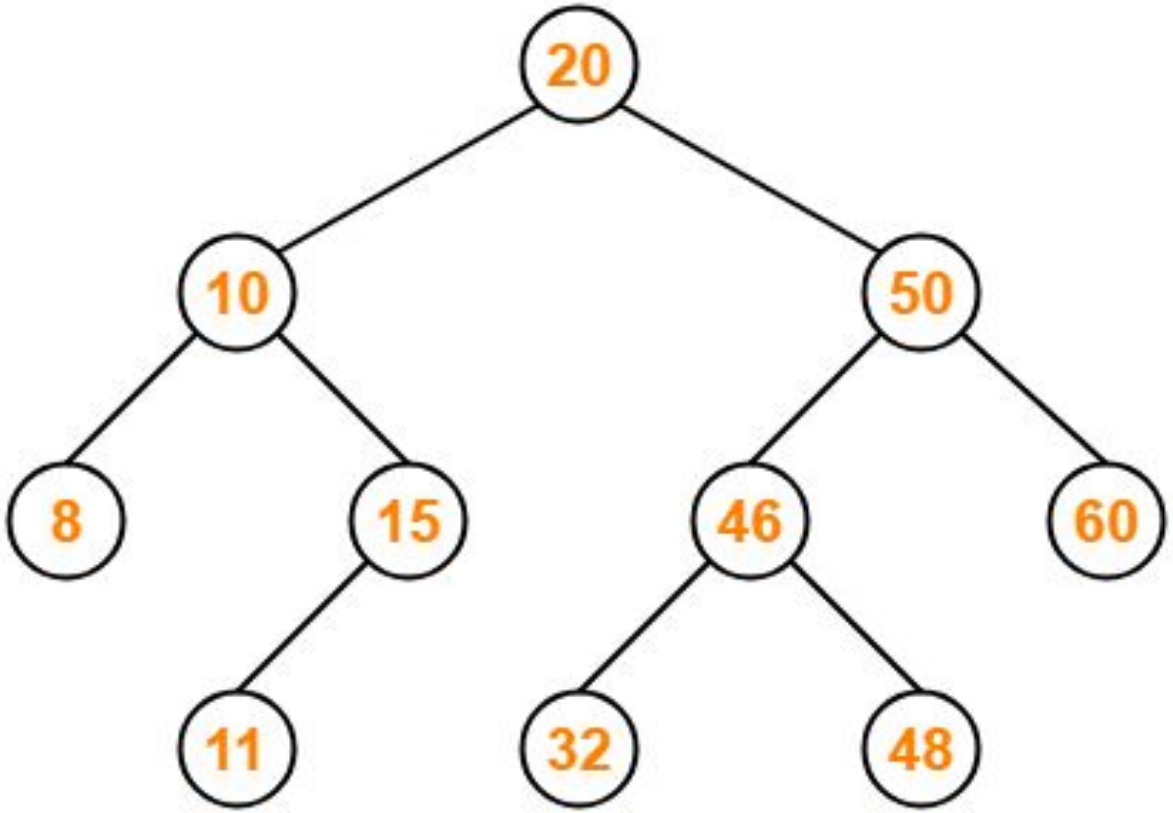
63, 9, 19, 27, 18, 108, 99, 81

Example


Step 1
Step 2
Step 3
Step 4
Inserted Node
Critical Node
LR Rotation
step 1
Step 5
Step 6
Step 7
Step 8
Step 9
Step 10
LL Rotation
AVL Tree

Try

- Construct an AVL tree having the following elements 50 , 20 , 60 , 10 , 8 , 15 , 32 , 46 , 11 , 48



Deletion in AVL Tree

Deletion in an AVL tree is similar to that in a BST. Deletion of a node tends to disturb the balance factor. Thus to balance the tree, we again use the Rotation mechanism.

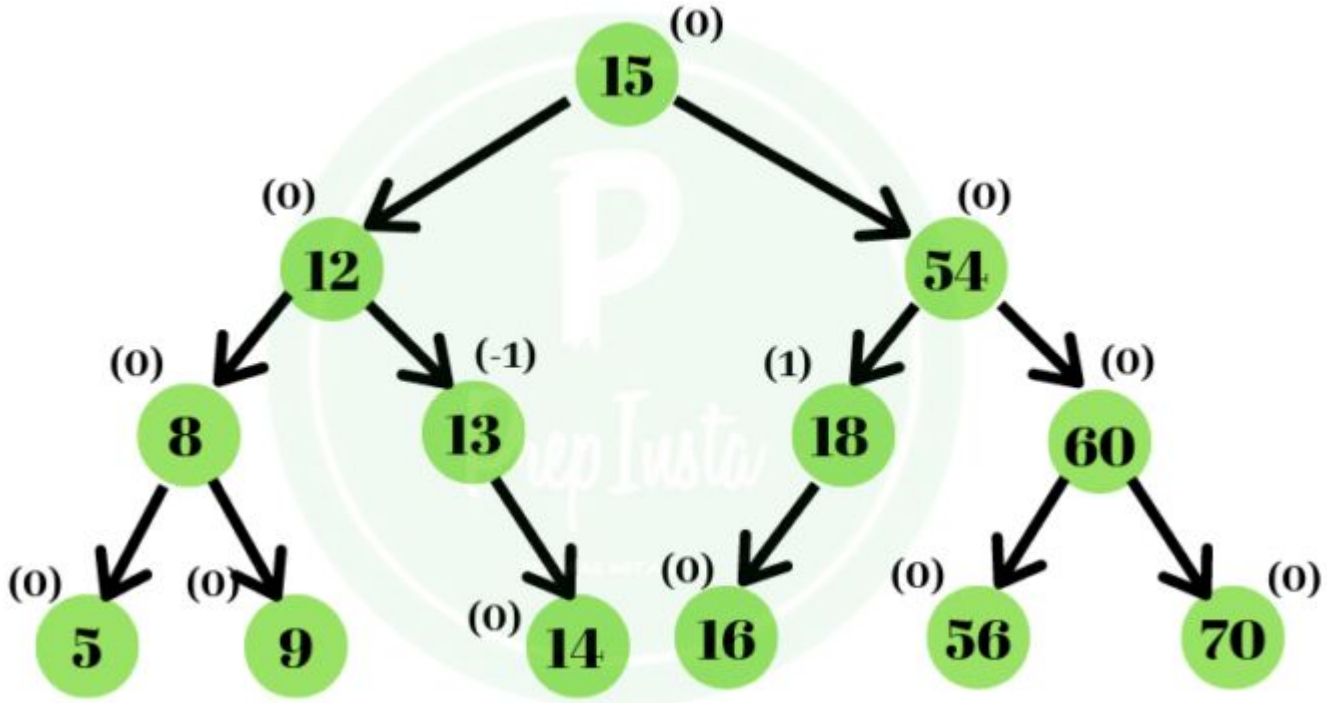
Deletion in AVL tree consists of two steps:

- Removal of the node
- Re-balancing of the tree

Process to delete node :

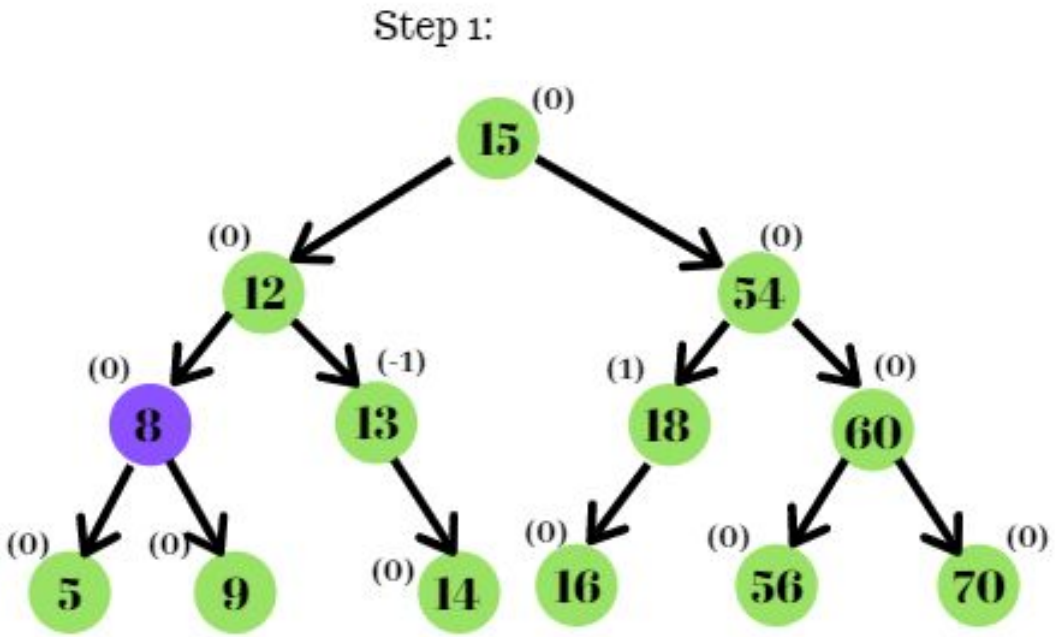
- If the node to be deleted is a leaf node, it is simply removed from the tree.
- If the node to be deleted has one child node, the child node is replaced with the node to be deleted simply.
- If the node to be deleted has two child nodes then, Either replace the node with it’s inorder predecessor , i.e, the largest element of the left sub tree or replace the node with it’s inorder successor , i.e, the smallest element of the right sub tree.

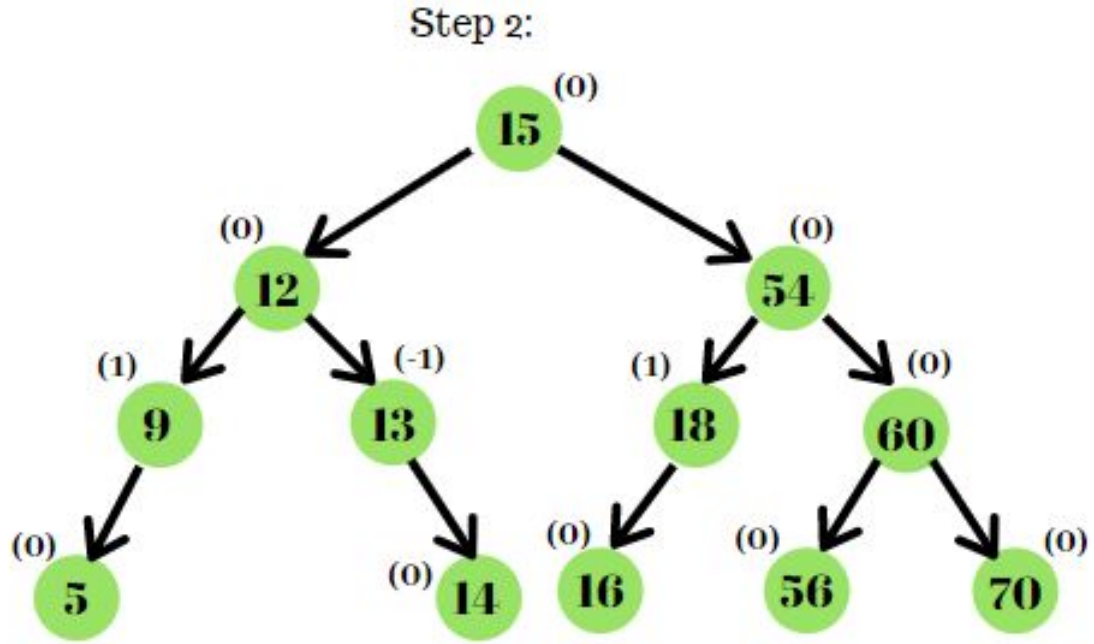
Deletion in AVL Tree



Deletion in AVL Tree

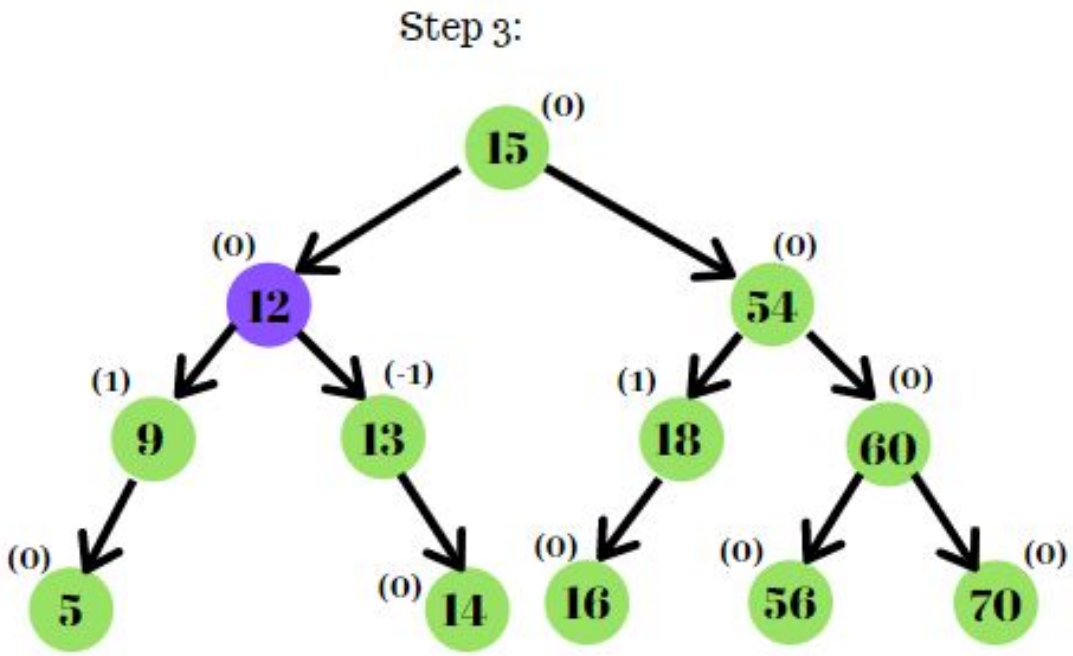
The node to be deleted from the tree is 8.

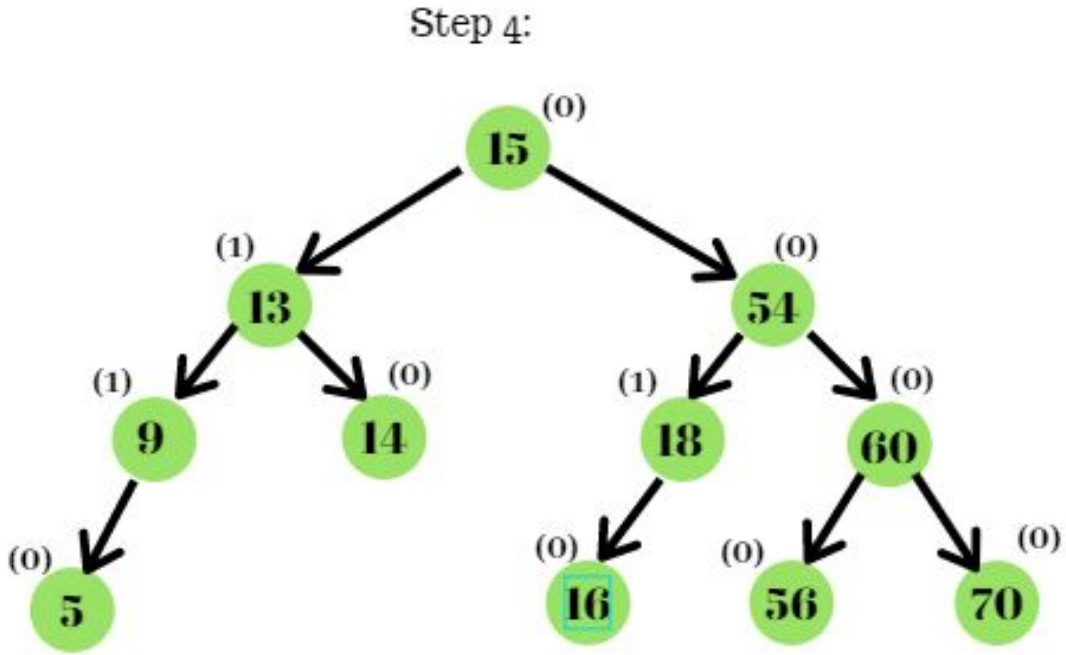




Deletion in AVL Tree

The next element to be deleted is 12.

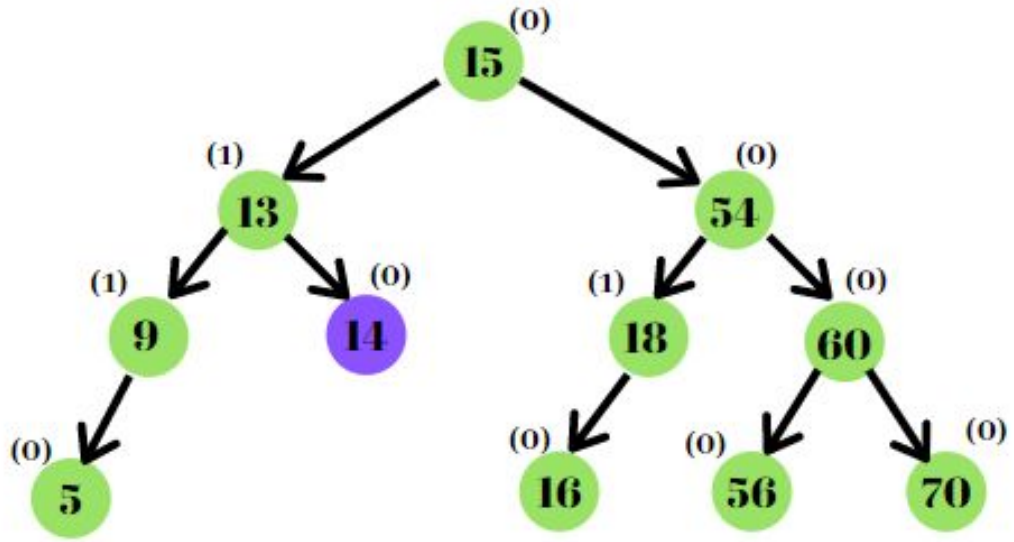




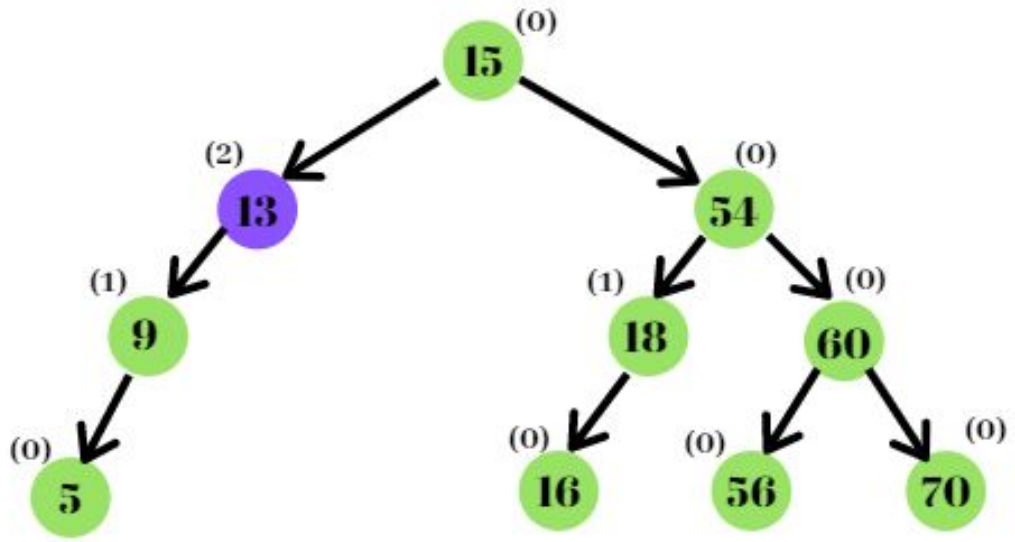
Deletion in AVL Tree

The next node to be eliminated is 14.

Step 5:



Step 6:

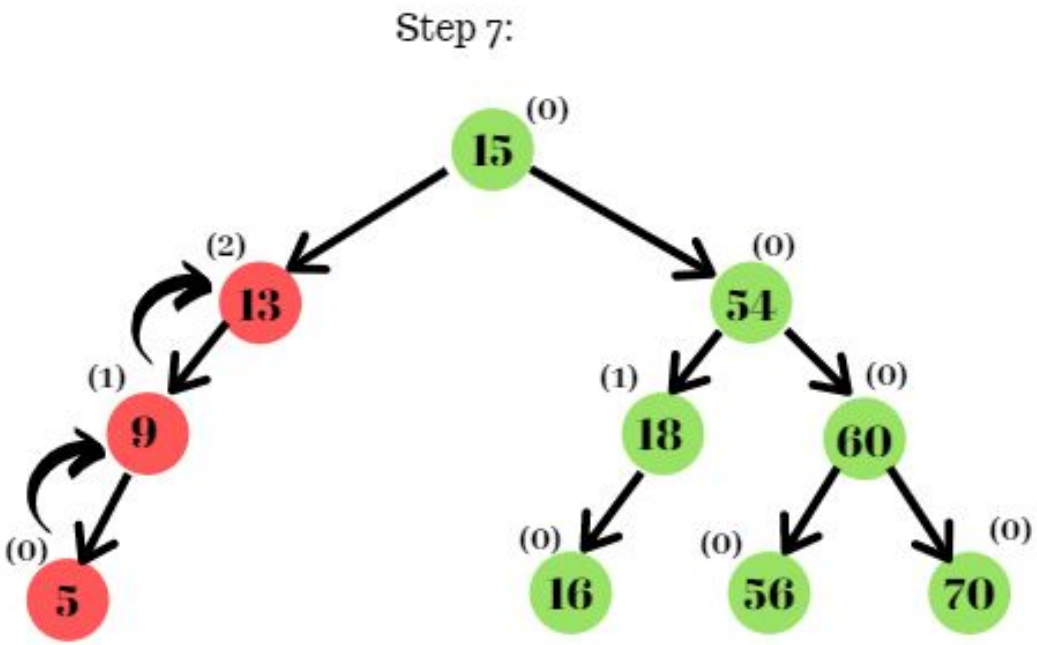


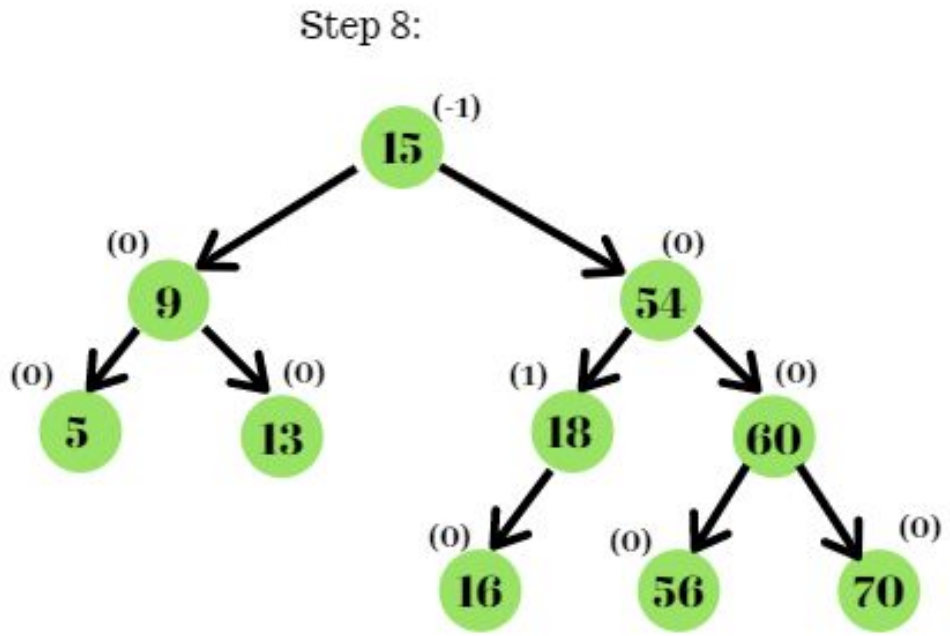
Step 7:

Step 8:

Deletion in AVL Tree

The next node to be eliminated is 14.





B Trees

B-Tree is a self-balancing search tree. In most of the other self-balancing search trees (like AVL and Red-Black Trees). To understand the use of B-Trees, we must think of the huge amount of data that cannot fit in main memory. When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to the main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, max, min, ..etc) require O(h) disk accesses where h is the height of the tree.

The height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. Generally, the B-Tree node size is kept equal to the disk block size. Since the height of the B-tree is low so total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red-Black Tree, ..etc.

B Trees

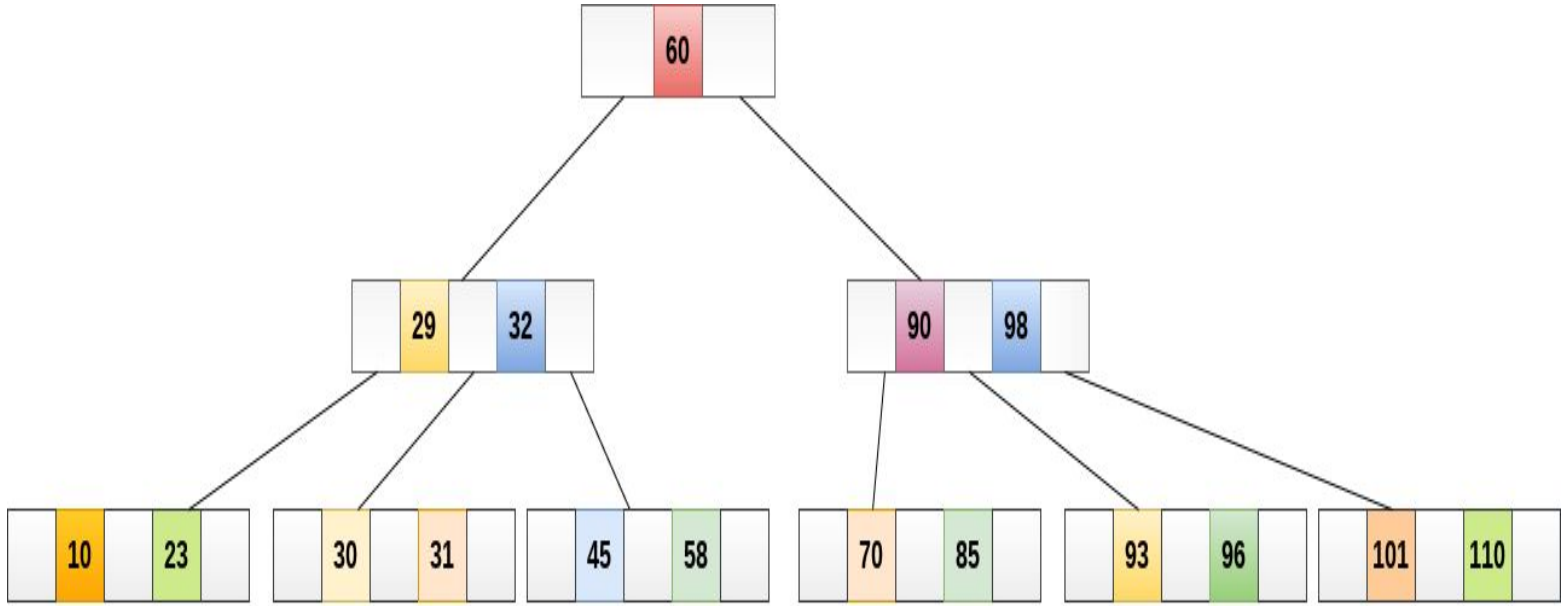
A B tree of order m contains the following properties.

1. Every node in a B-Tree contains at most m children.
2. Every node in a B-Tree except the root node and the leaf node contain at least m/2 children.
3. The root nodes must have at least 2 nodes.
4. All leaf nodes must be at the same level.

It is not necessary that, all the nodes contain the same number of children but, each node must have m/2 number of nodes.

B Trees

A B tree of order 4 is shown in the following image.



While performing some operations on B Tree, any property of B Tree may violate such as number of minimum children a node can have. To maintain the properties of B Tree, the tree may split or join.

Operations

Searching :

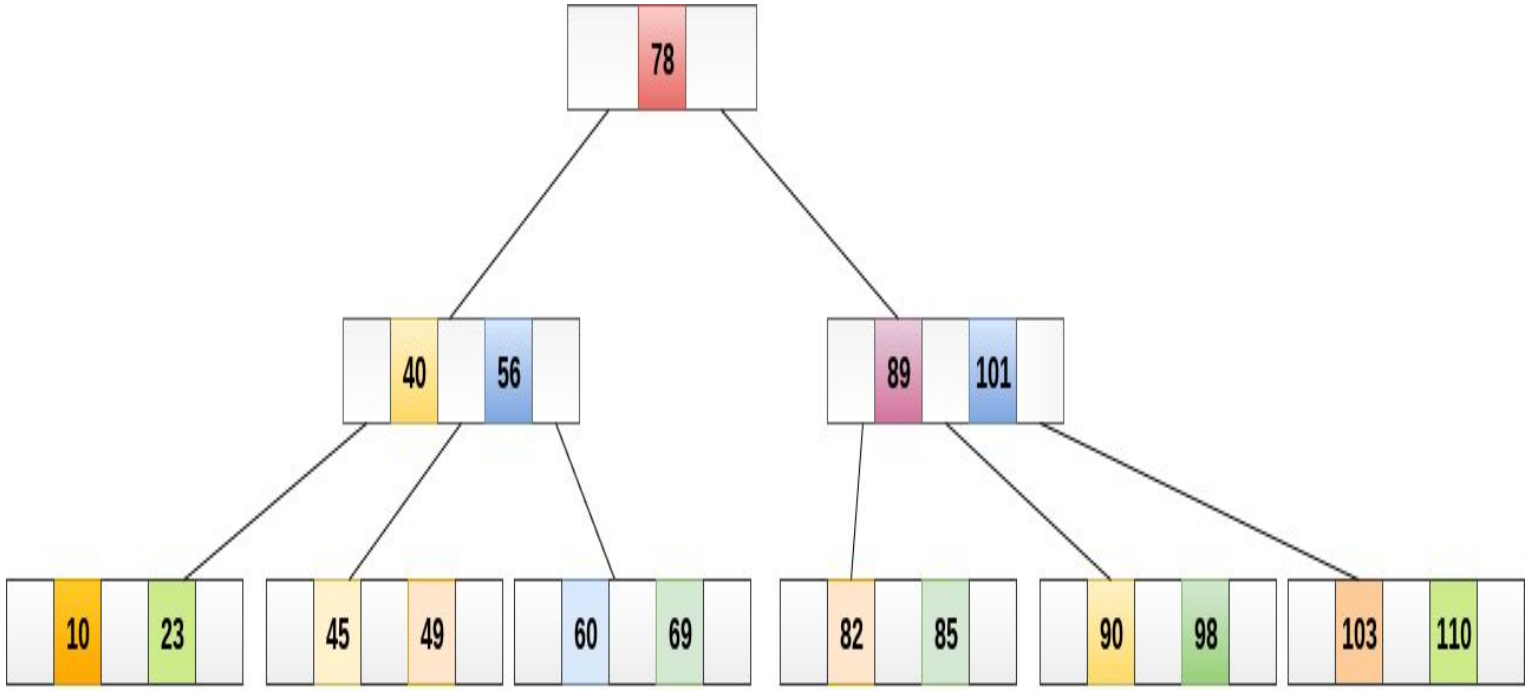
Searching in B Trees is similar to that in Binary search tree. For example, if we search for an item 49 in the following B Tree. The process will something like following :

1. Compare item 49 with root node 78. since 49 < 78 hence, move to its left sub-tree.
2. Since, 40<49<56, traverse right sub-tree of 40.
3. 49>45, move to right. Compare 49.
4. match found, return.

Searching in a B tree depends upon the height of the tree. The search algorithm takes O(log n) time to search any element in a B tree.

Operations

Searching :



Operations

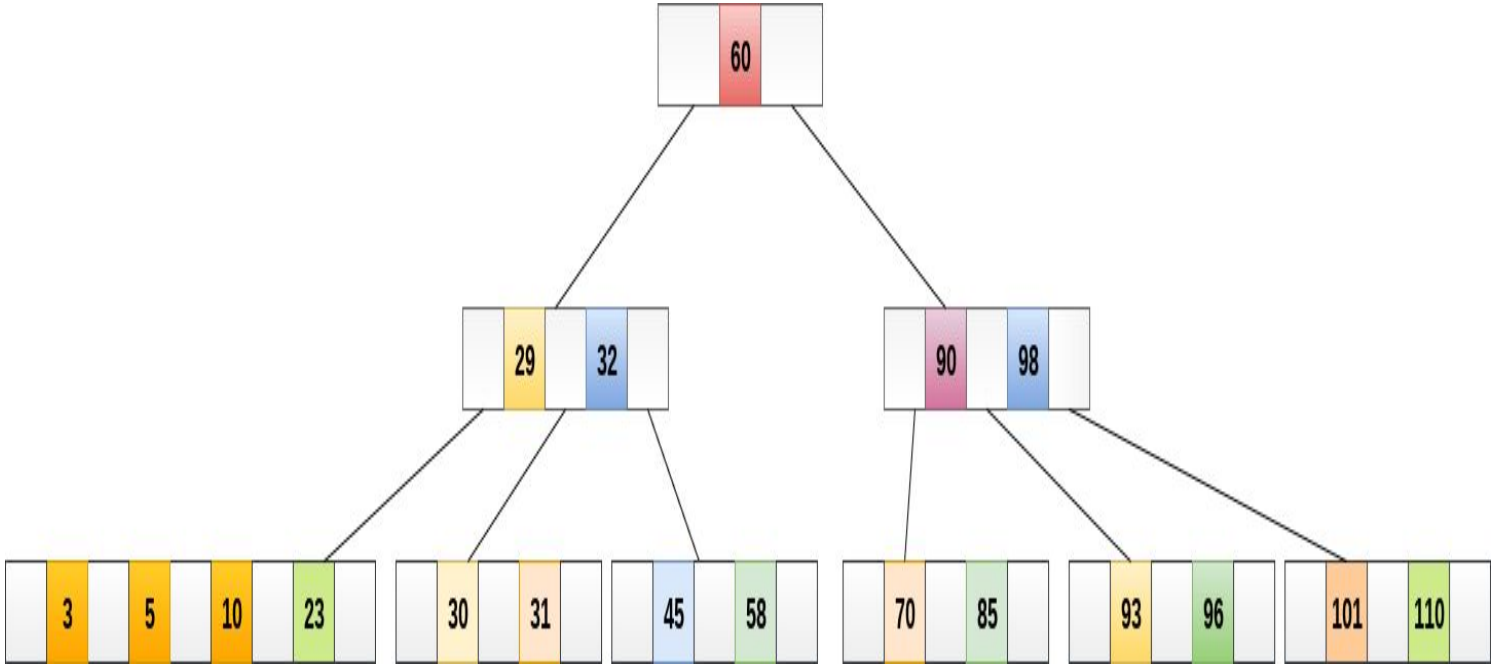
Inserting

1. Insertions are done at the leaf node level. The following algorithm needs to be followed in order to insert an item into B Tree.
2. Traverse the B Tree in order to find the appropriate leaf node at which the node can be inserted.
3. If the leaf node contain less than m-1 keys then insert the element in the increasing order.
4. Else, if the leaf node contains m-1 keys, then follow the following steps.
    - Insert the new element in the increasing order of elements.
    - Split the node into the two nodes at the median.
    - Push the median element upto its parent node.
    - If the parent node also contain m-1 number of keys, then split it too by following the same steps.

Operations

Example:

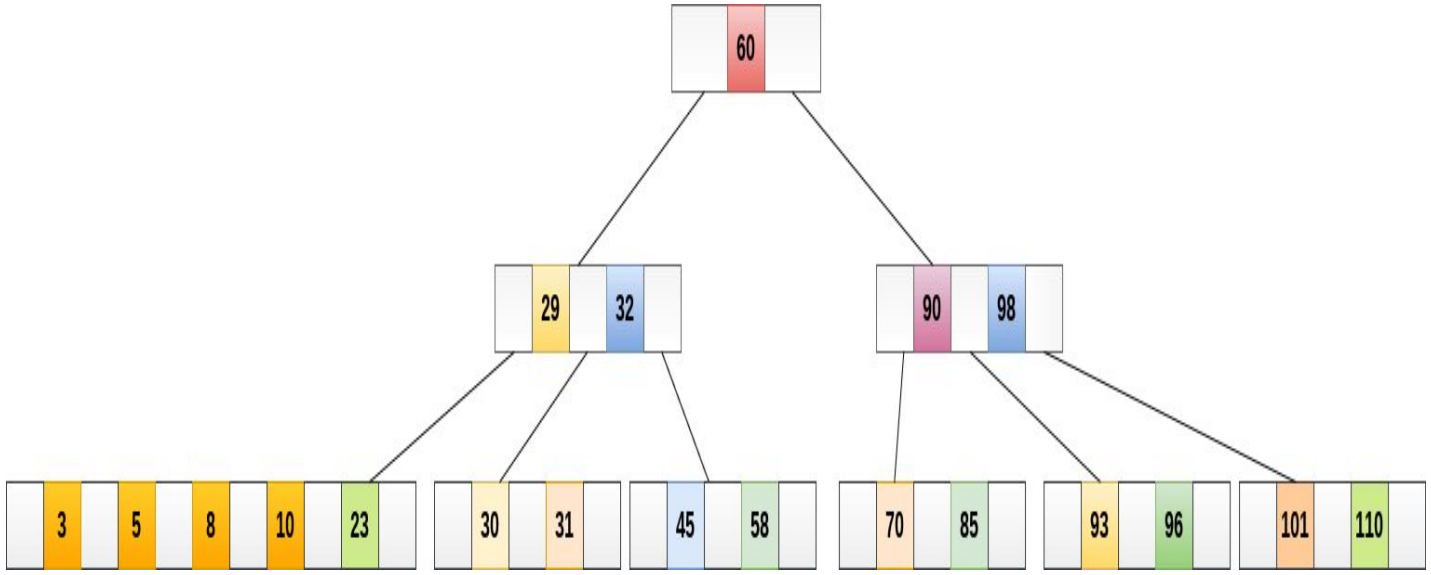
Insert the node 8 into the B Tree of order 5 shown in the following image.



Operations

Example:

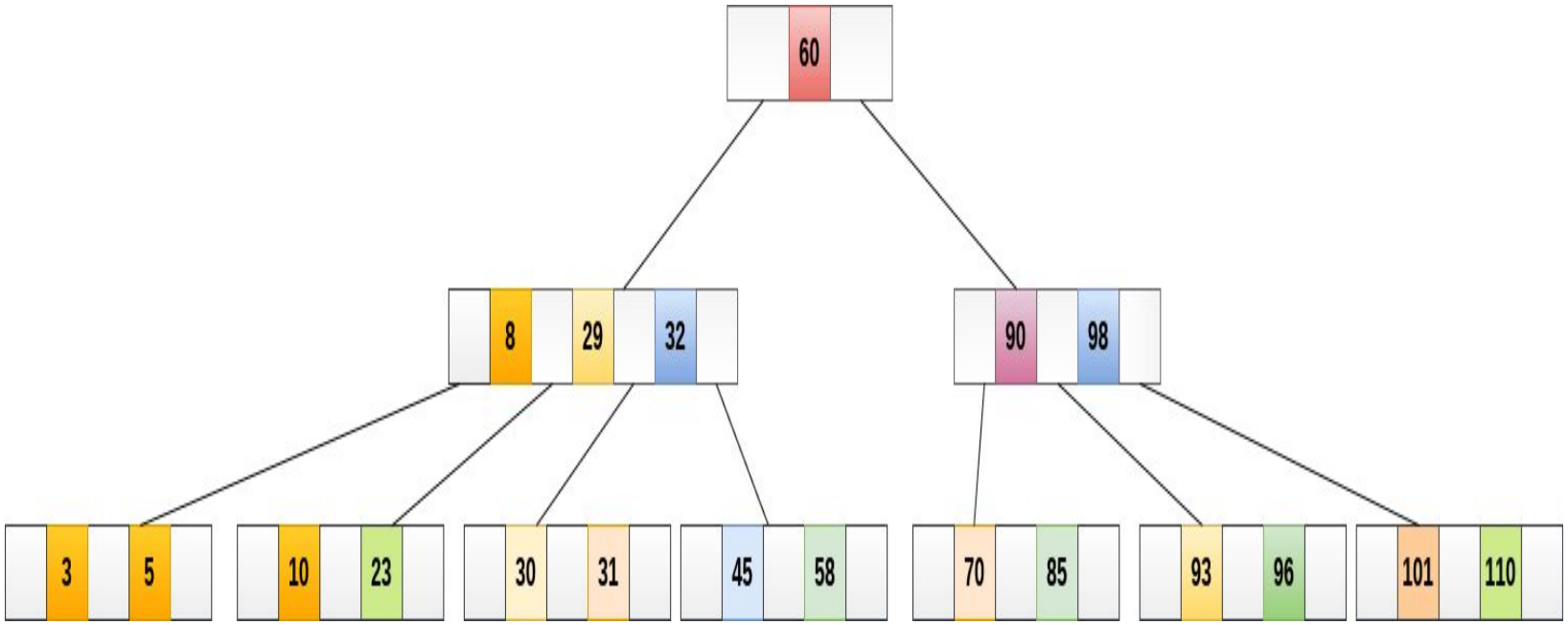
8 will be inserted to the right of 5, therefore insert 8.



Operations

Example:

The node, now contain 5 keys which is greater than (5 -1 = 4) keys. Therefore split the node from the median i.e. 8 and push it up to its parent node shown as follows.



B+ Tree

B+ Tree is an extension of B Tree which allows efficient insertion, deletion and search operations.

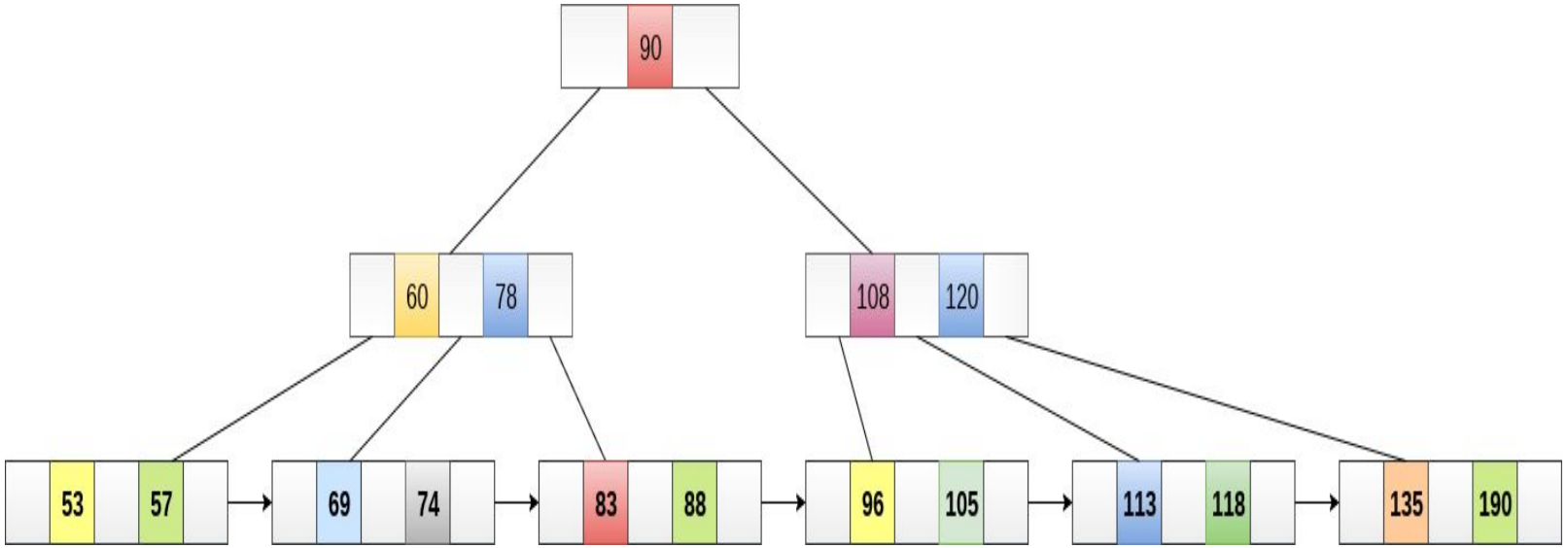
In B Tree, Keys and records both can be stored in the internal as well as leaf nodes. Whereas, in B+ tree, records (data) can only be stored on the leaf nodes while internal nodes can only store the key values.

The leaf nodes of a B+ tree are linked together in the form of a singly linked lists to make the search queries more efficient.

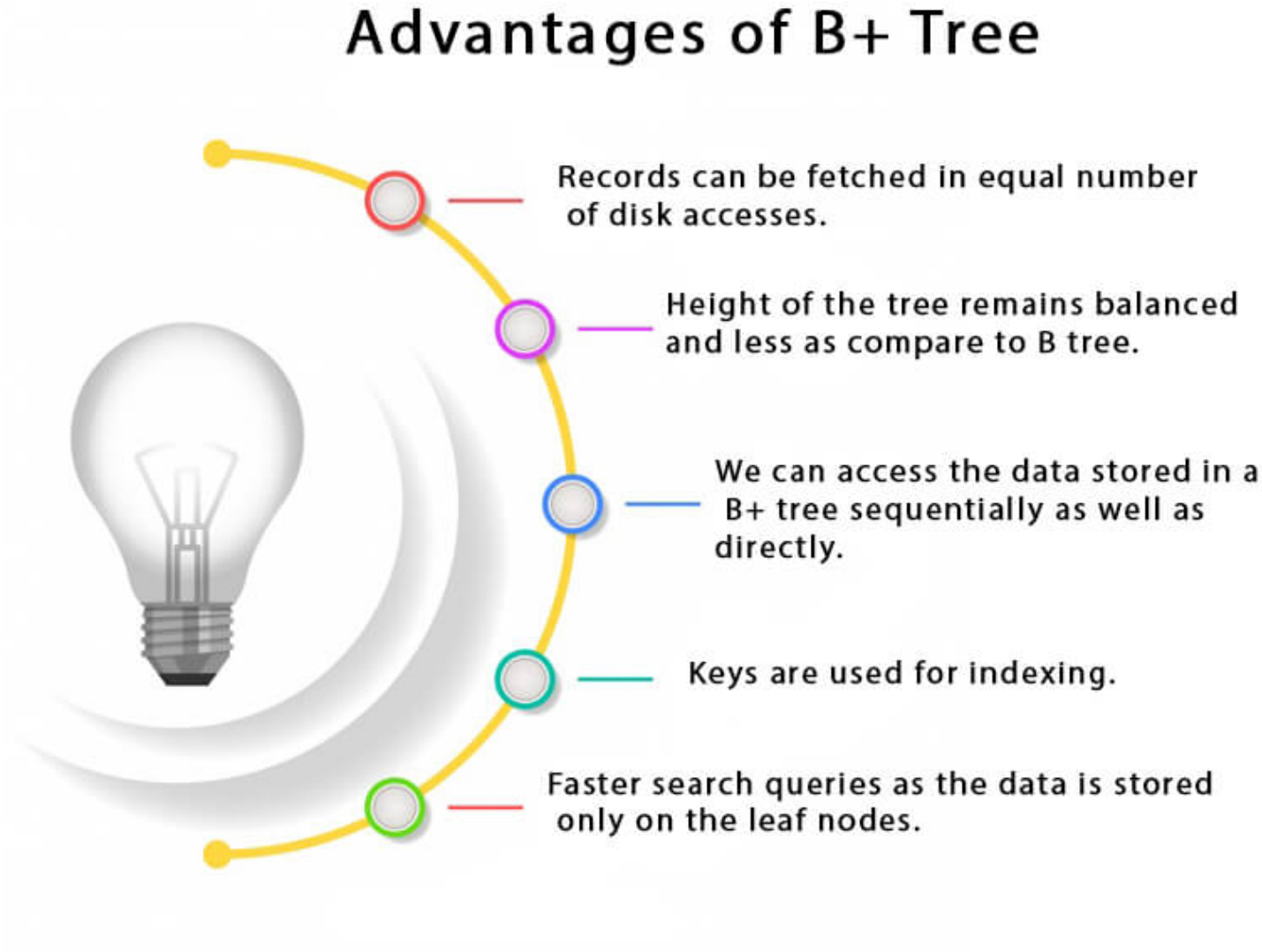
B+ Tree are used to store the large amount of data which can not be stored in the main memory. Due to the fact that, size of main memory is always limited, the internal nodes (keys to access records) of the B+ tree are stored in the main memory whereas, leaf nodes are stored in the secondary memory.

B+ Tree

The internal nodes of B+ tree are often called index nodes. A B+ tree of order 3 is shown in the following figure.



B+ Tree


Advantages of B+ Tree
Records can be fetched in equal number of disk accesses.
Height of the tree remains balanced and less as compare to B tree.
We can access the data stored in a B+ tree sequentially as well as directly.
Keys are used for indexing.
Faster search queries as the data is stored only on the leaf nodes.

B Tree VS B+ Tree

SN	B Tree	B+ Tree
1	Search keys cannot be repeatedly stored.	Redundant search keys can be present.
2	Data can be stored in leaf nodes as well as internal nodes	Data can only be stored on the leaf nodes.
3	Searching for some data is a slower process since data can be found on internal nodes as well as on the leaf nodes.	Searching is comparatively faster as data can only be found on the leaf nodes.
4	Deletion of internal nodes is so complicated and time consuming.	Deletion will never be a complex process since element will always be deleted from the leaf nodes.
5	Leaf nodes cannot be linked together.	Leaf nodes are linked together to make the search operations more efficient.

Insertion in B+ Tree

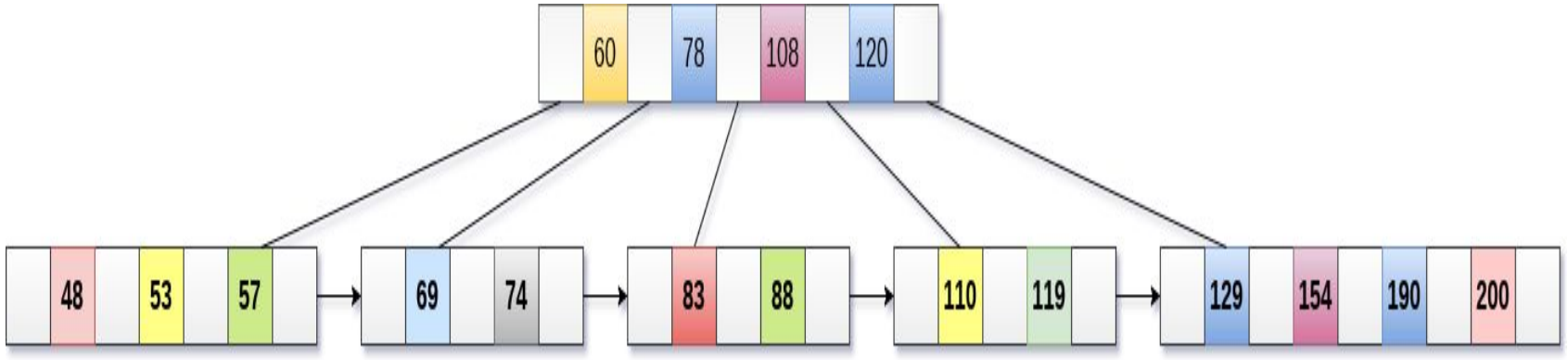
Step 1: Insert the new node as a leaf node

Step 2: If the leaf doesn't have required space, split the node and copy the middle node to the next index node.

Step 3: If the index node doesn't have required space, split the node and copy the middle element to the next index page.

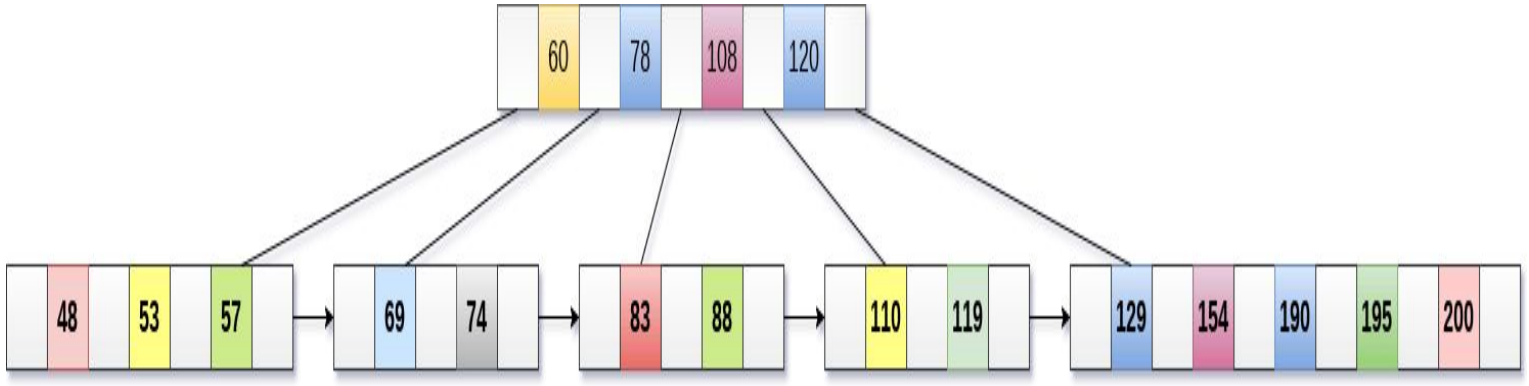
Example :

- Insert the value 195 into the B+ tree of order 5 shown in the following figure.

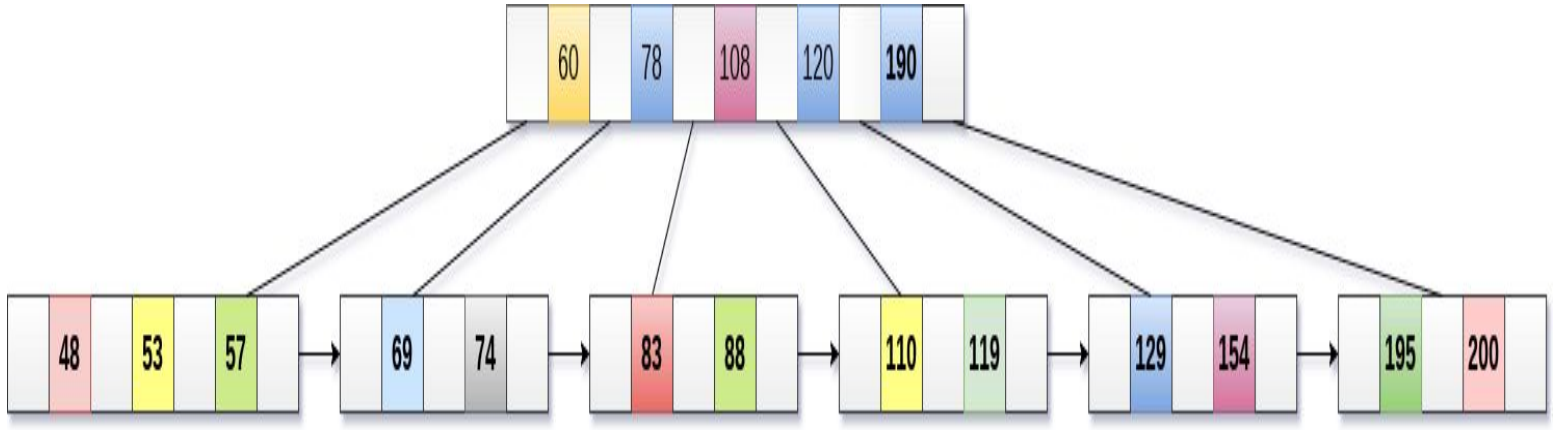


Insertion in B+ Tree

195 will be inserted in the right sub-tree of 120 after 190. Insert it at the desired position.

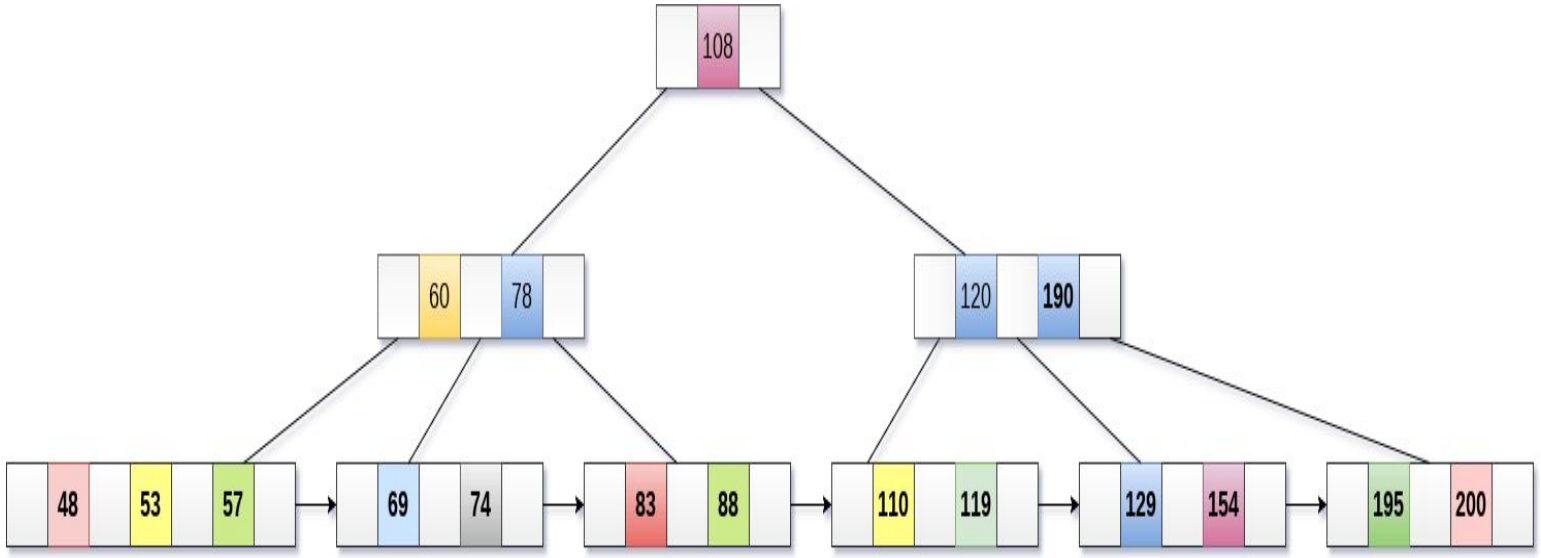


The node contains greater than the maximum number of elements i.e. 4, therefore split it and place the median node up to the parent.



Insertion in B+ Tree

Now, the index node contains 6 children and 5 keys which violates the B+ tree properties, therefore we need to split it, shown as follows.



Deletion in B+ Tree

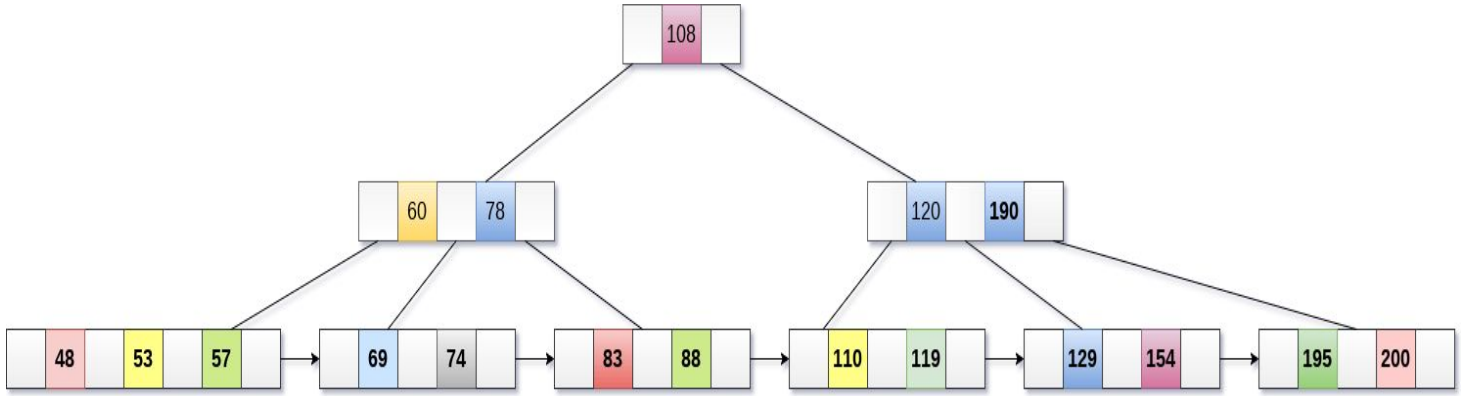
Step 1: Delete the key and data from the leaves.

Step 2: if the leaf node contains less than minimum number of elements, merge down the node with its sibling and delete the key in between them.

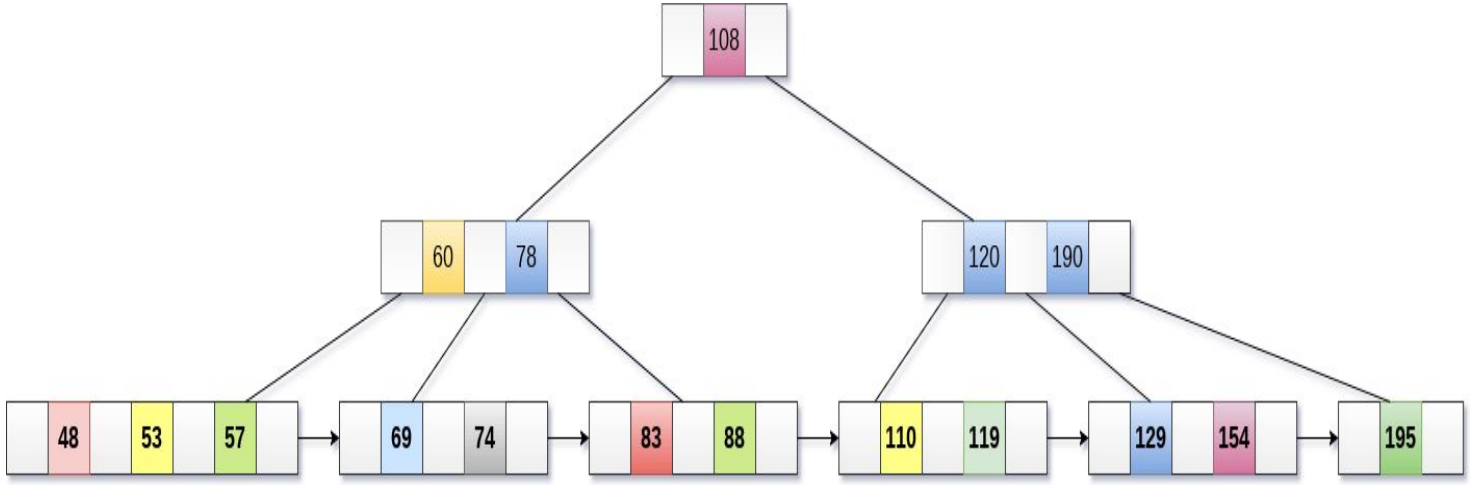
Step 3: if the index node contains less than minimum number of elements, merge the node with the sibling and move down the key in between them.

Deletion in B+ Tree

Example: Delete the key 200 from the B+ Tree shown in the following figure.

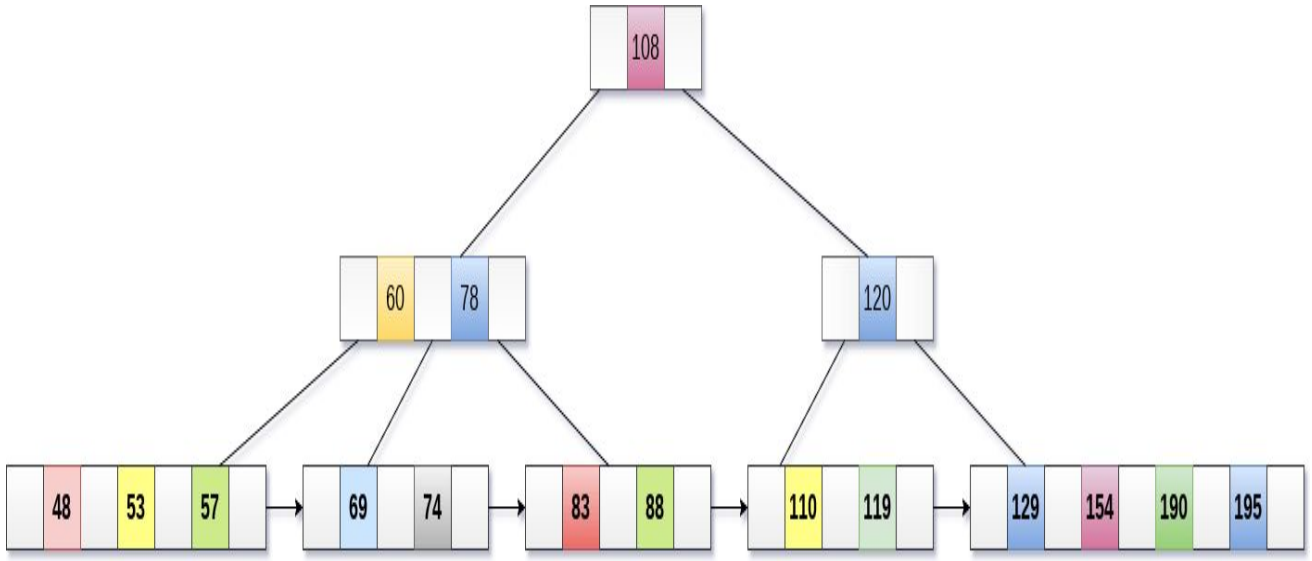


200 is present in the right sub-tree of 190, after 195. delete it.



Deletion in B+ Tree

Merge the two nodes by using 195, 190, 154 and 129.



Now, element 120 is the single element present in the node which is violating the B+ Tree properties. Therefore, we need to merge it by using 60, 78, 108 and 120.

Now, the height of B+ tree will be decreased by 1.

Deletion in B+ Tree