NANYANG TECHNOLOGICAL UNIVERSITY

Data Structures & Algorithms in Python: AVL Trees

Dr. Owen Noel Newton Fernando

College of Computing and Data Science

Terminology

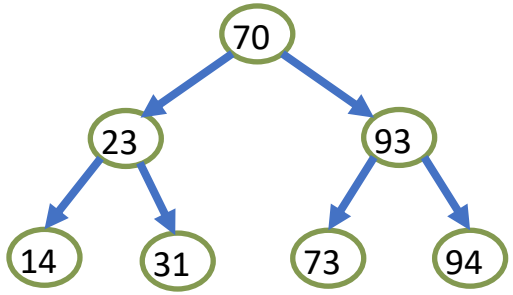
- **Height of a tree**: The number of **edges** on the longest path from the root to a leaf
- **Depth/Level of a node**: The number of **edges** from the node to the root of its tree.

Terminology

- **Height of a tree**: The number of **edges** on the longest path from the root to a leaf
- **Depth/Level of a node**: The number of **edges** from the node to the root of its tree.
- **Empty Binary Tree**: A binary tree with no nodes. It is still considered as a tree.

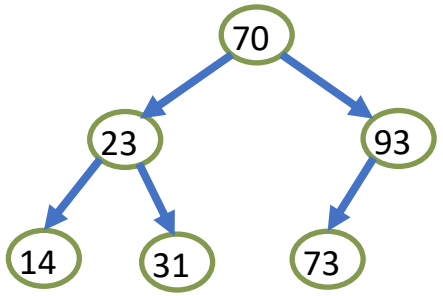
Terminology

- **Height of a tree**: The number of **edges** on the longest path from the root to a leaf
- **Depth/Level of a node**: The number of edges from the node to the root of its tree.
- **Empty Binary Tree**: A binary tree with no nodes. It is still considered as a tree.
- **Perfect Binary Tree**: A binary tree of height ***H*** with no missing nodes. All leaves are at level *H* and all other nodes each have two children



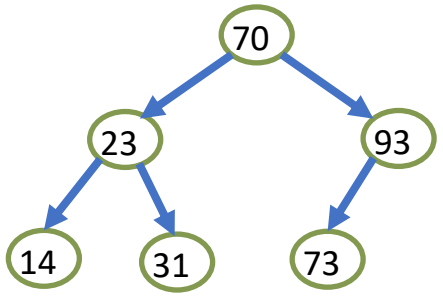
Terminology

- **Height of a tree**: The number of **edges** on the longest path from the root to a leaf
- **Depth/Level of a node**: The number of edges from the node to the root of its tree.
- **Empty Binary Tree**: A binary tree with no nodes. It is still considered as a tree.
- **Perfect Binary Tree**: A binary tree of height *H* with no missing nodes. All leaves are at level *H* and all other nodes each have two children
- **Complete Binary Tree**: A binary tree of height H that is full to level H-1 and has level H filled in from left to right



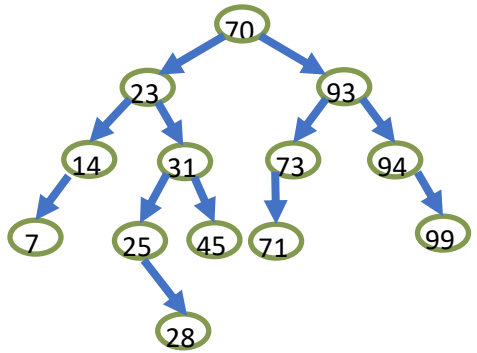
Terminalogy

- **Height of a tree**: The number of **edges** on the longest path from the root to a leaf
- **Depth/Level of a node**: The number of edges from the node to the root of its tree.
- **Empty Binary Tree**: A binary tree with no nodes. It is still considered as a tree.
- **Perfect Binary Tree**: A binary tree of height *H* with no missing nodes. All leaves are at level *H* and all other nodes each have two children
- **Complete Binary Tree**: A binary tree of height H that is full to level H-1 and has level H filled in from left to right



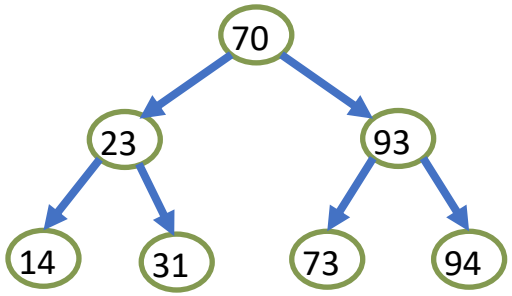
Terminology

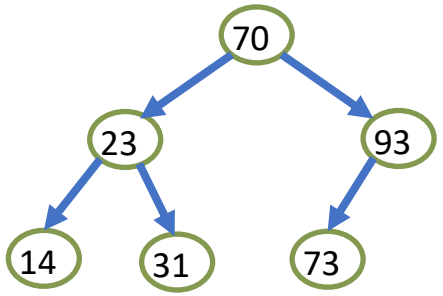
- **Height of a tree**: The number of **edges** on the longest path from the root to a leaf
- **Depth/Level of a node**: The number of edges from the node to the root of its tree.
- **Empty Binary Tree**: A binary tree with no nodes. It is still considered as a tree.
- **Perfect Binary Tree**: A binary tree of height *H* with no missing nodes. All leaves are at level *H* and all other nodes each have two children
- **Complete Binary Tree**: A binary tree of height H that is full to level H-1 and has level H filled in from left to right
- **Balanced Binary Tree**: A binary tree in which the **left and right subtrees of any node have heights that differ by at most 1**

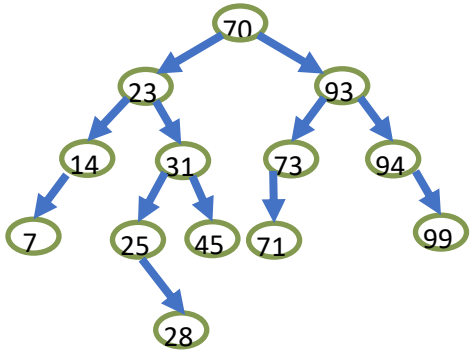


Terminology

- **Height of a tree**: The number of **edges** on the longest path from the root to a leaf
- **Depth/Level of a node**: The number of edges from the node to the root of its tree.
- **Empty Binary Tree**: A binary tree with no nodes. It is still considered as a tree.
- **Perfect Binary Tree**: A binary tree of height *H* with no missing nodes. All leaves are at level *H* and all other nodes each have two children
- **Complete Binary Tree**: A binary tree of height H that is full to level H-1 and has level H filled in from left to right
- **Balanced Binary Tree**: A binary tree in which the left and right subtrees of any node have heights that differ by at most 1







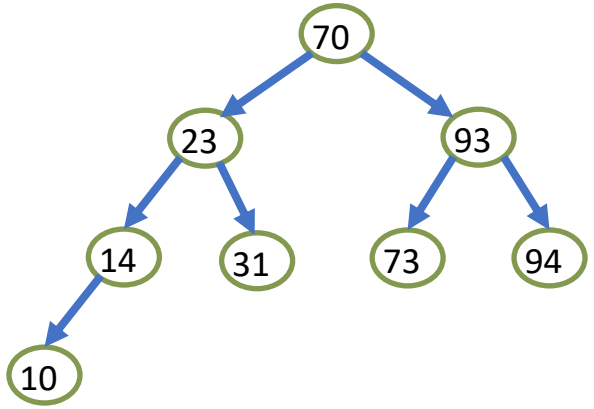
Terminalogy

- The Height of a tree: The number of **edges** on the longest path from the root to a leaf
- The Depth/Level of a node: The number of **edges** from the node to the root of its tree.
- A **complete binary tree** of height H has a number of nodes bounded by:

  $2^H - 1 < n \leq 2^{H+1} - 1$, where n is the number of nodes in the tree and H is the height of the tree.

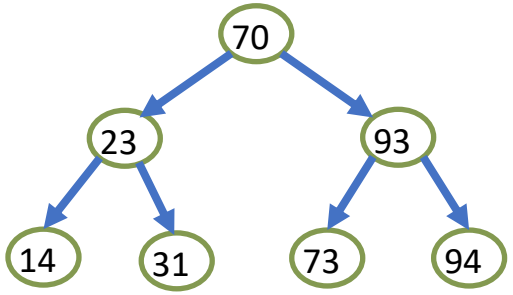
  $2^H \leq n < 2^{H+1}$

- **Minimum Number of Nodes (2^H)**
  - A **complete binary tree** of height H must have at least 2^H nodes.
  - This happens when the last level is partially filled.
  - Example: H=3, the minimum nodes required is $2^3 = 8$



Terminology

- **Maximum Number of Nodes ($2^{H+1}-1$)**
  - A **complete binary tree** of height H can have **at most $2^{H+1}-1$** nodes.
  - This happens when all levels are completely full.
  - Example: H=3, the minimum nodes required is **$2^{3+1}-1 = 15$**
  - Thus, the total number of nodes falls in the range: **$2^{H}-1 < n \leq 2^{H+1}-1$**
  - For example, if **H =3**, then: **$8 \leq n \leq 15$**
  - which means n can be any value from 8 to 15.



Terminology

- A **complete binary tree** of height H has a number of nodes bounded by:

  $2^H - 1 < n \leq 2^{H+1} - 1$ where n is the number of nodes in the tree and H is the height of the tree.

- Thus, the total number of nodes falls in the range:

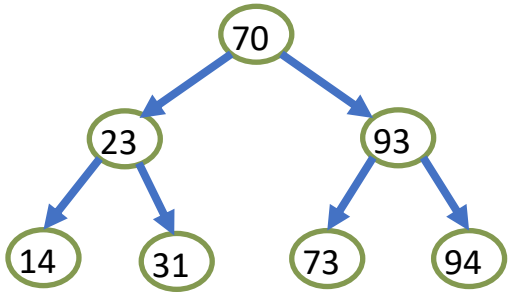
  $2^H \leq n < 2^{H+1}$ **(eg. 7<n ≤15 ≡ 8 ≤n<16)**

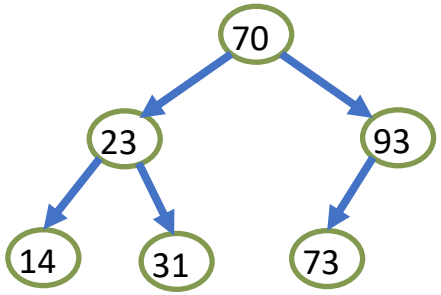
- Taking logarithm (base 2) on both sides:

  $H \leq \log_2 n < H+1$

- If H is an integer, H+1 must be the next integer.

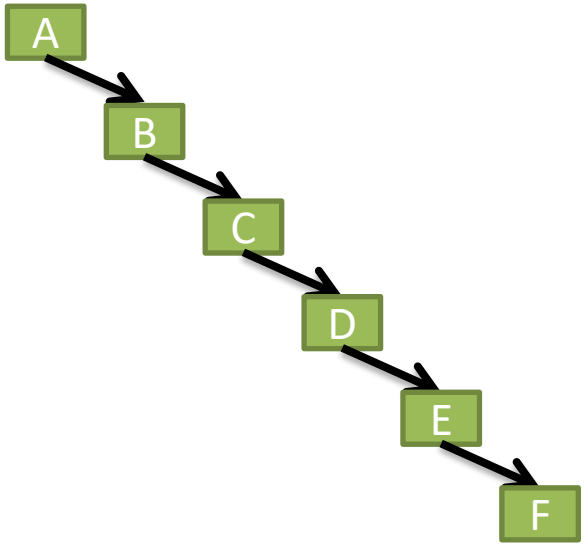
- **Minimal Height =** $\lfloor \log_2 n \rfloor$

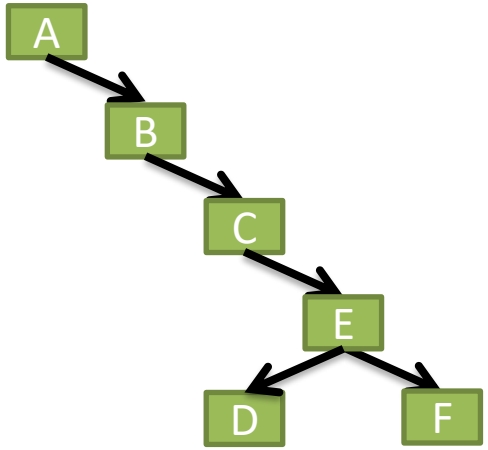


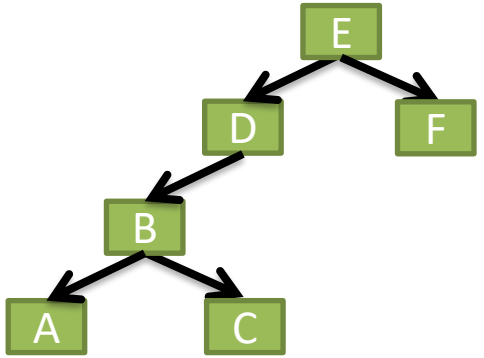


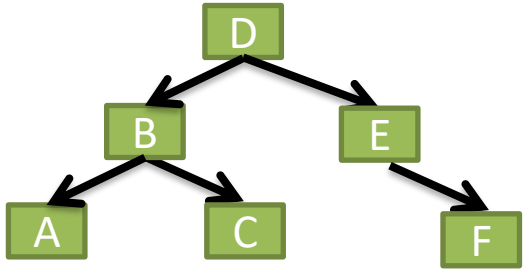
The 'Good' and 'Bad' Binary Search Trees

- To get a good binary search tree
  - To make the tree as a complete binary tree





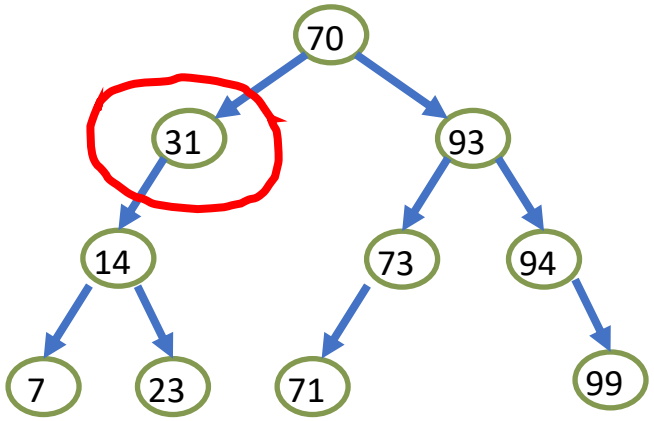




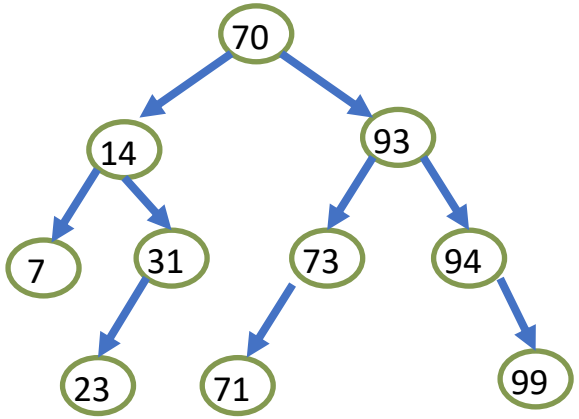
Tree Balancing

- A BST is height-balanced or balanced if the difference in height of both subtrees of any node in the tree is either zero or one.
- Most Balanced BST: Each tree node has exactly two child nodes except for the bottom 2 levels

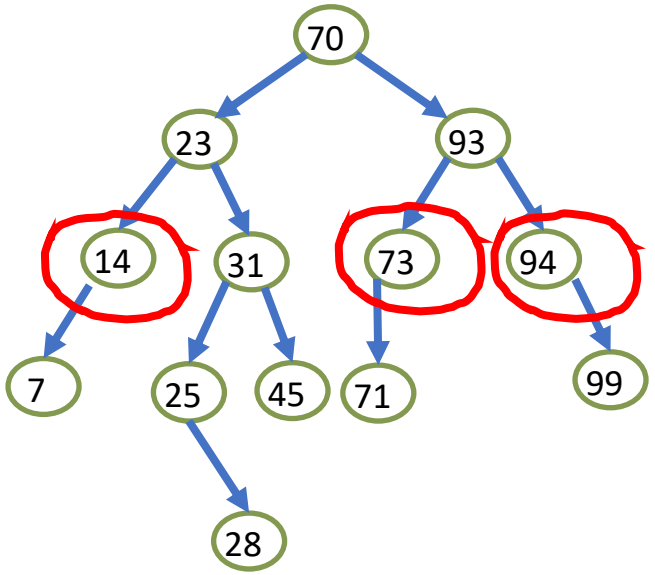
Unbalanced BST



Most Balanced BST



Balanced BST

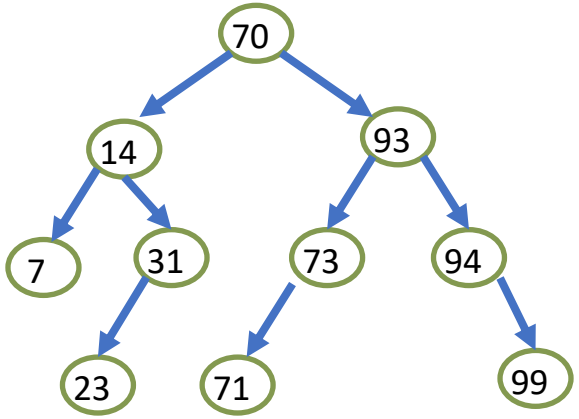


Tree Balancing

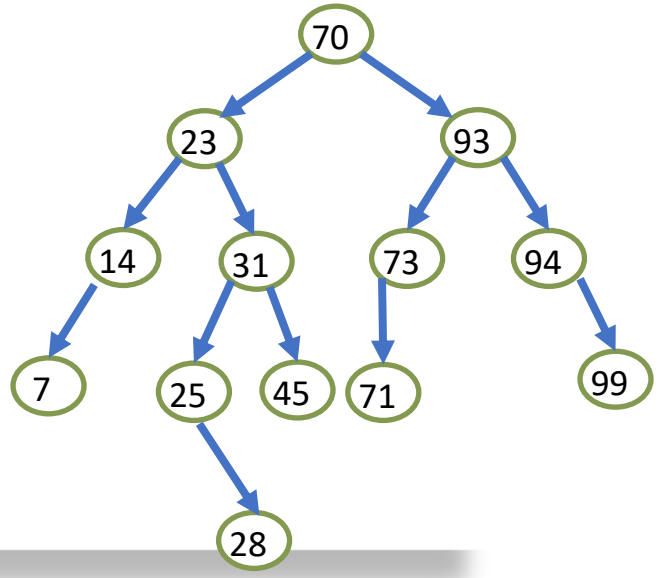
- Most Balanced BST: Each tree node has exactly two child nodes except for the bottom 2 levels
- How do we balance a **binary search tree**?

1. **Sort all the data in an array and reconstruct the tree**
2. **The AVL Tree:**
    - It is a locally balanced tree:
    - Heights of left vs right subtrees differ by at most 1
    - invented by Adel'son-Velskii and Landis in 1962

Most Balanced BST



AVL Tree



Sort all data in a list and reconstruct the tree

1. In-order traversal visits every node in the given BST. We obtain the sorted data:

   7,14,23,31,70,71,73,93,94,99

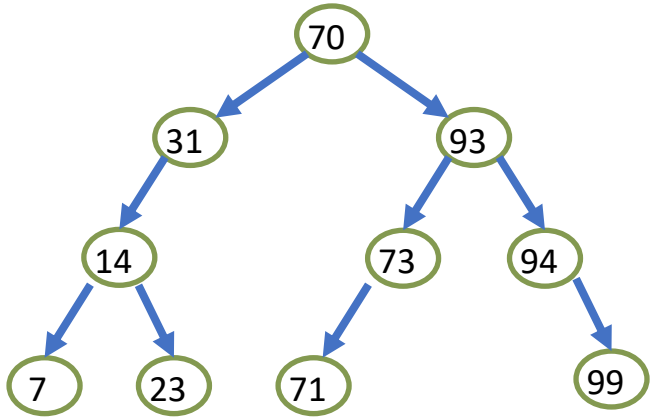
2. Storage it in a list
3. Take the middle element of the array as the root of the tree: 70
4. The first half of the array is used to build the left subtree of 70

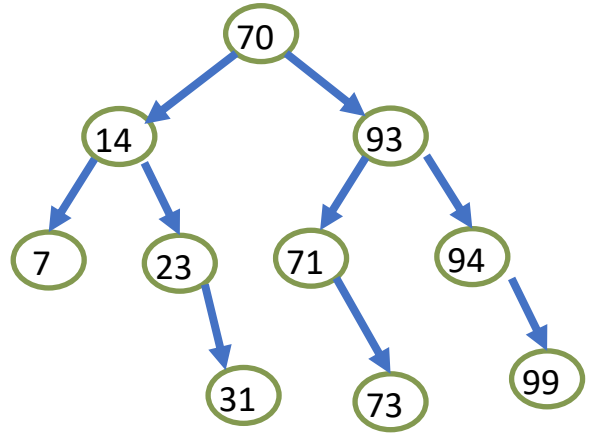
   7,14,23,31

5. The second half of the array is used to build the right subtree

   71,73,93,94,99

6. Step 4 and Step 5 recursively repeat the step 3-5.

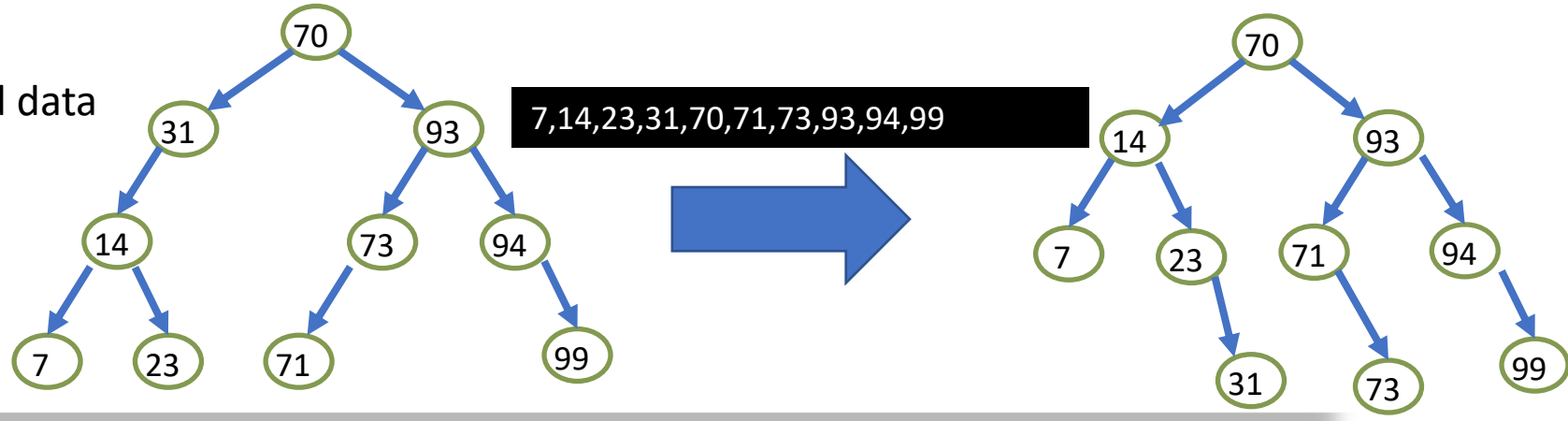




Sort all data in a list and reconstruct the tree

```
def tree_balance(data, first, last):
    if last >= first:
        middle = (first + last) // 2 # // is rounded down to the nearest integer.
        root = Node(data[middle])
        root.left = tree_balance(data, first, middle - 1)
        root.right =tree_balance(data, middle + 1, last)
        return root
    return None
```

- Need an extra list to store sorted data
- Rebuild the whole tree



AVL Tree History

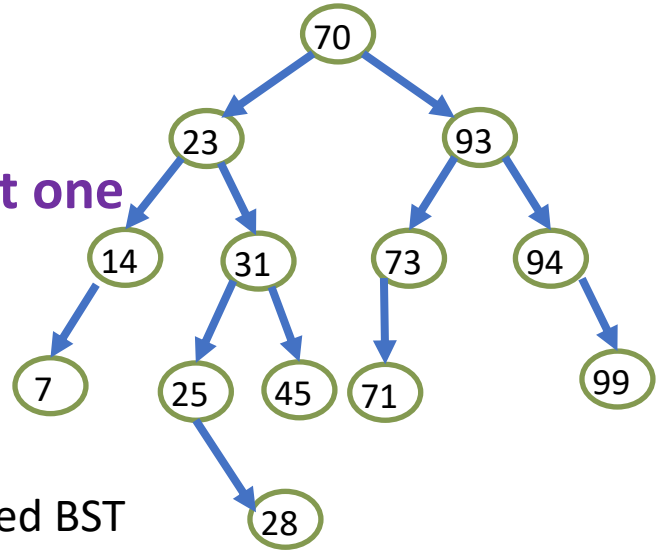
- AVL:
  - Invented in 1962
  - Gregory Maximovich **A**delson-**V**elskii
  - Evgenii Mikhailovich **L**andis

Adelson-Velskii

Landis

AVL Tree approach

- Add/ remove only one node to/ from a BST
- The BST may become unbalance after insertion or removal
- Instead of reconstructing the BST via sorting data, the BST can be locally balanced
- It is known as AVL Tree
- **The height of left and right subtrees of every node differ by at most one**

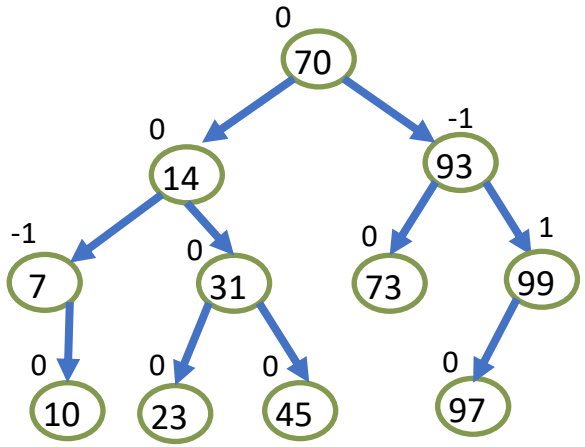


Balanced BST

Balance Factor

- **Balance Factor: Height of Left Subtree – Height of Right Subtree**
- All the leave have 0 Balance Factor

```
class AVLNode:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
        self.height = 0
```



- **An AVL tree**: The **heights** of left and right subtrees of every node differ by at **most one**.
  - **Balance Factor** of each node in an AVL tree can only be **-1, 0** or **1.**
  - Node insertion or node removal from the tree may change the balance factor of its ancestors (from parent, grandparent, grand-grandparent etc. to the root of the tree)

Balance Factor

- **Balance Factor: Height of Left Subtree – Height of Right Subtree**

```
def _get_height(self, node):
        if not node:
            return -1
        return node.height # return 1 + max(left_height, right_height)
```

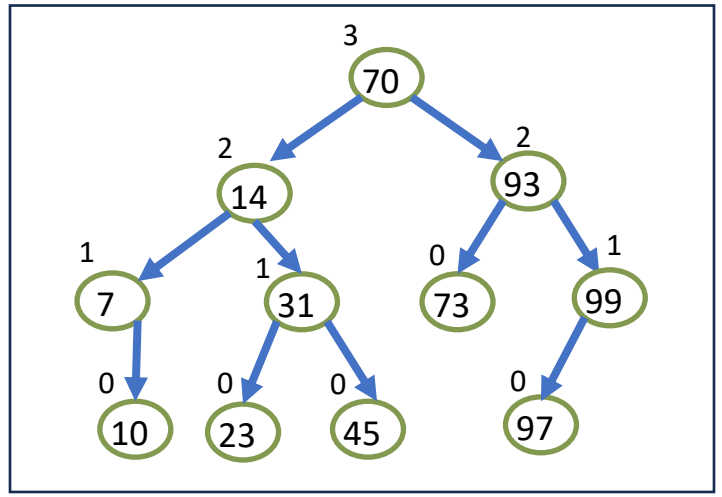
```
def _get_balance(self, node):
        if not node:
            return 0
        return self._get_height(node.left) - self._get_height(node.right)
```

Height of a tree: The number of **edges** on the longest path from the root to a leaf

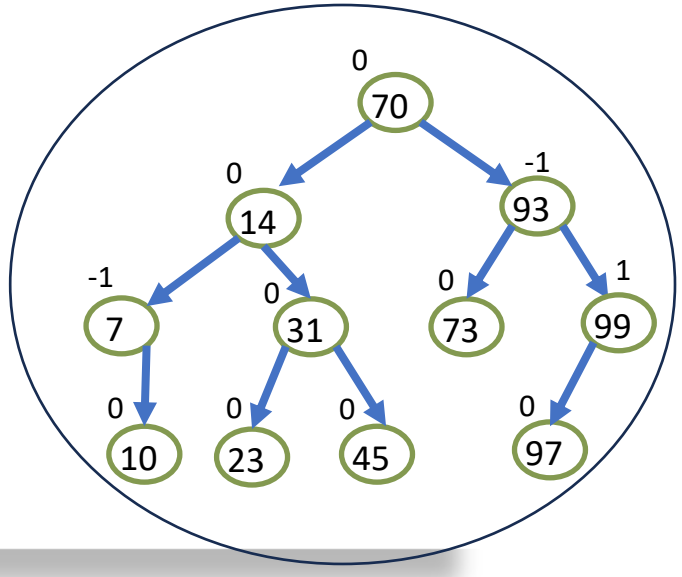
Balance Factor

- **Height of a tree**: The number of **edges** on the longest path from the root to a leaf node

```
def _get_height(self, node):
        if not node:
            return -1
        return node.height # return 1 + max(left_height, right_height)
```




```
def _get_balance(self, node):
        if not node:
            return 0
        return self._get_height(node.left) - self._get_height(node.right)
```



Balance Factor: Height of Left Subtree – Height of Right Subtree

Node Insertion

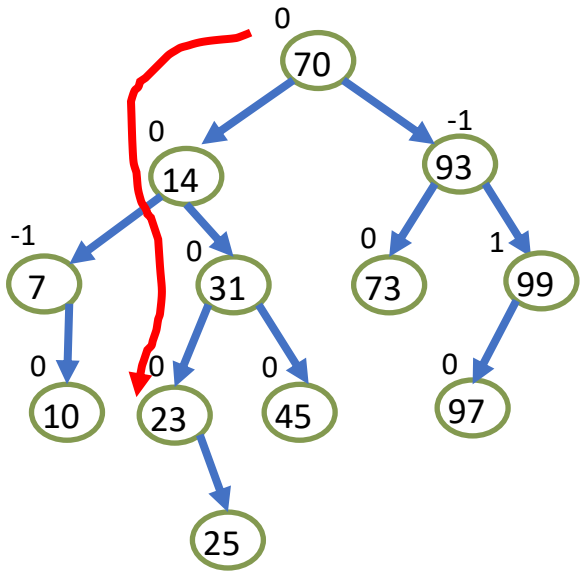
- **Case 1**: Balance factor of nodes along the insertion path from the root to the expectant parent node is zero.
- **Case 2**: Balance factor of nodes along the insertion path from the root to the expectant parent node is non-zero (1 or -1) but a new node is inserted at the shorter subtree.
- **Case 3**: Balance factor of nodes along the insertion path from the root to the expectant parent node is non-zero (1 or -1) and a new node is inserted at the higher subtree. The new node is inserted at the non-zero balance factor node's
    a) Left child's Left subtree (**LL Case**)
    b) Right child's Right subtree (**RR Case**)
    c) Left child's Right subtree (**LR Case**)
    d) Right child's Left subtree (**RL Case**)

Node Insertion: Case 1

Case 1: Insert node 25

Nodes along the insertion path from the root to the expectant parent node, 23

- Balance factor is zero before insert the node
- After insertion, the heights of left and right subtrees of every node still differ by at most one.
- No balancing issue.

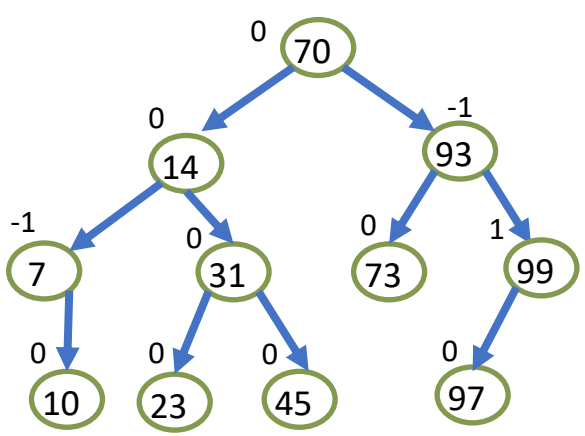


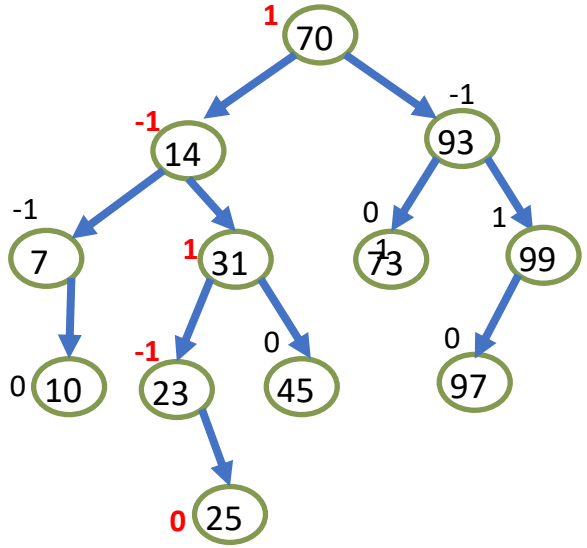
Node Insertion: Case 1

Case 1: Insert node 25

Nodes along the insertion path from the root to the expectant parent node, 23

- Balance factor is zero before insert the node
- After insertion, the heights of left and right subtrees of every node still differ by at most one.
- No balancing issue.



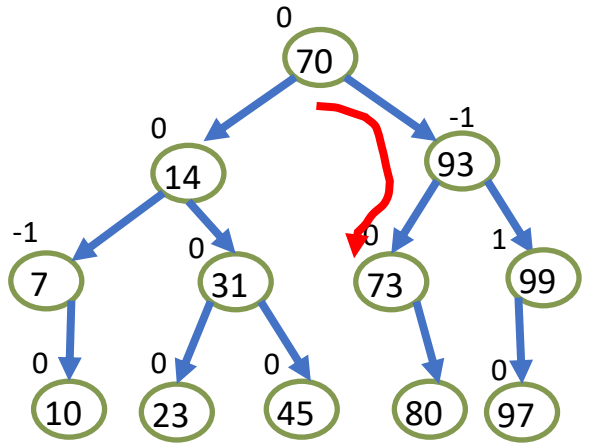


Node Insertion: Case 2

Case 2: Insert node 80

Nodes along the insertion path from the root to the expectant parent node, 73

- Balance factor of 93 is -1 before insert the node
- Insertion occurs at 93 's shorter subtree
- After insertion, the height of left and right subtrees of every node still differ by at most one.
- No balancing issue.

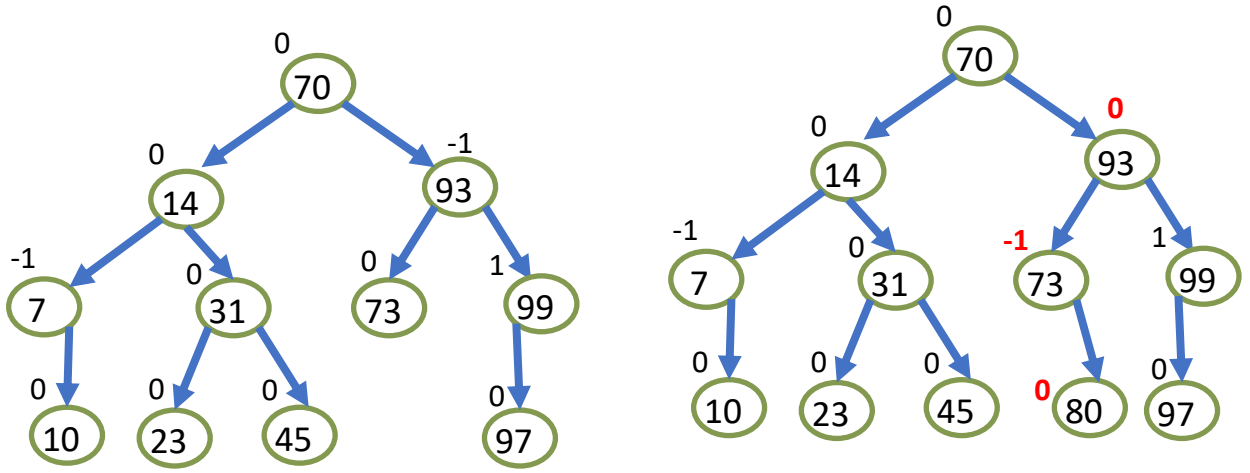


Node Insertion: Case 2

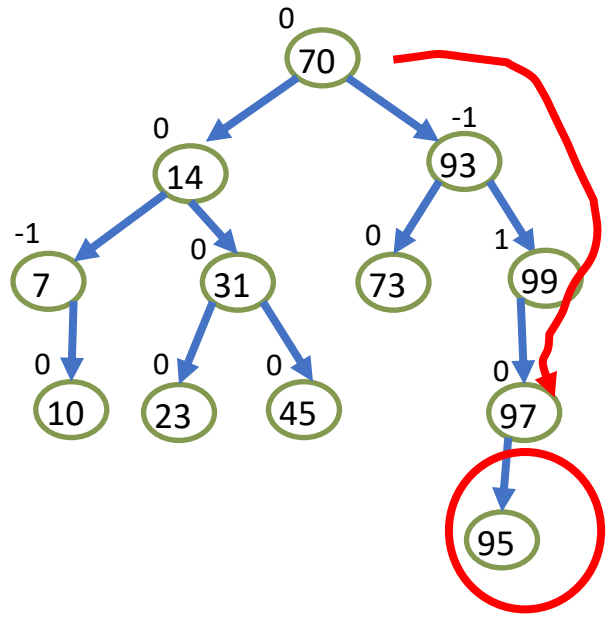
Case 2: Insert node 80

Nodes along the insertion path from the root to the expectant parent node, 73

- Balance factor of 93 is -1 before insert the node
- Insertion occurs at 93 's shorter subtree
- After insertion, the height of left and right subtrees of every node still differ by at most one.
- No balancing issue.



Node Insertion: Case 3 (LL)



Case 3 (a) LL Case: Insert node 95

Nodes along the insertion path from the root to the expectant parent node, 97

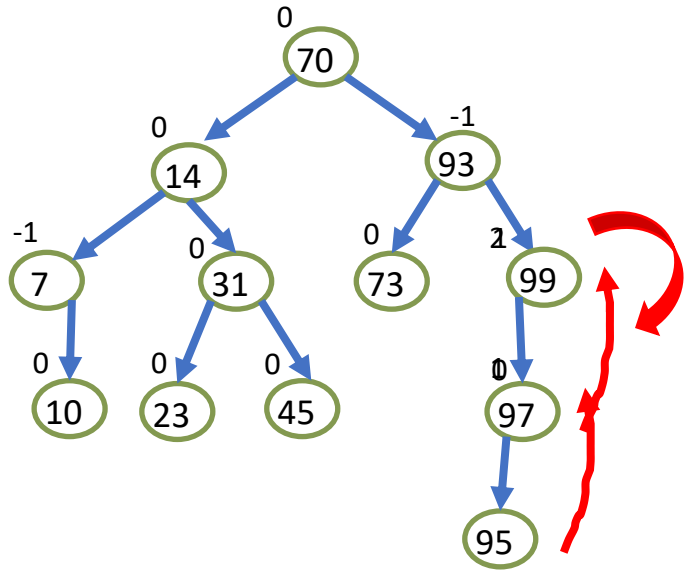
- Balance factor of 99 is 1 before insert the node
- Insertion occurs at 99 's higher subtree
- After insertion, the height of left and right subtrees of every node differ by more than one.
- **Right Rotation** about node 99 is required to rebalance the tree.

Node Insertion: Case 3 (LL) - Right Rotation

1. Update the height of node 97
2. Update the balance factor of node 97

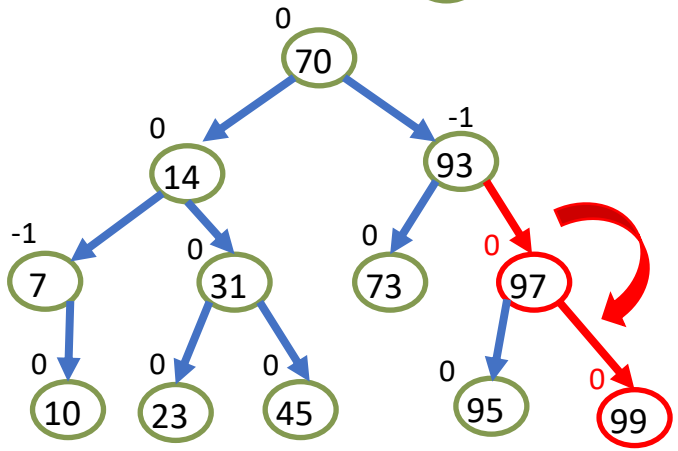
1. Update the height of node 99
2. Update the balance factor of node 99
3. Node 99 is unbalanced. BF>1

⇒ Some nodes in the left subtree need to shift to right subtree



Rotate Right about Node 99

1. Let Node 97's right child be Node 99's left child (None in this case) *those nodes are between 99 and 97
2. Let Node 99 be Node 97's right child
3. Let Node 97 be Node 93's right child.

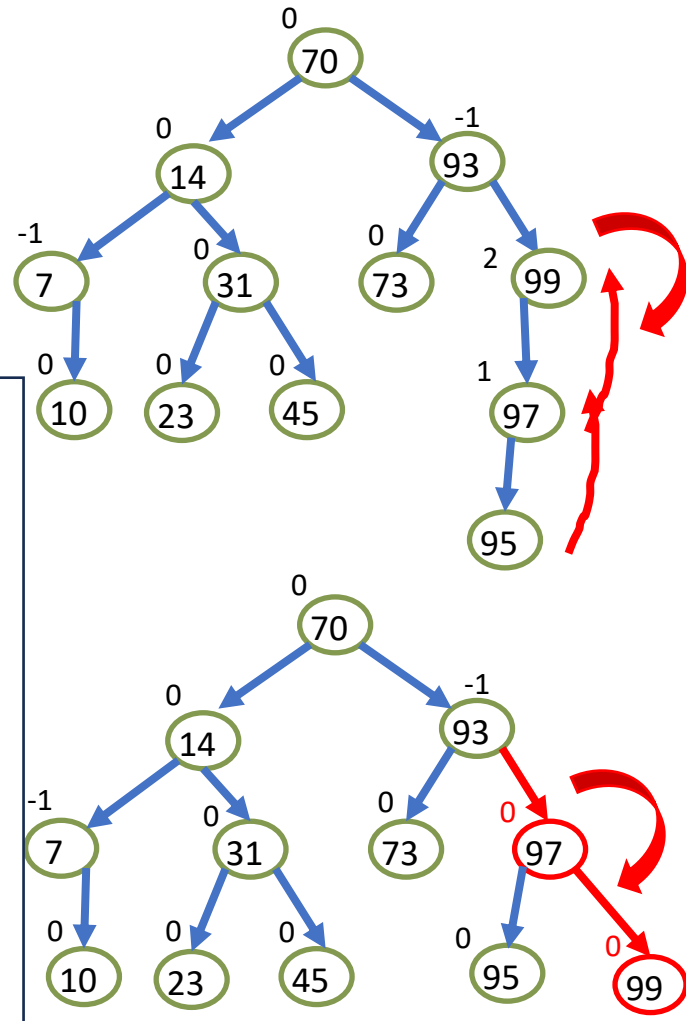


Node Insertion: Case 3 (LL) - Right Rotation

Rotate Right about Node 99

1. Let Node 97's right child be Node 99's left child (None in this case) *those nodes are between 99 and 97
2. Let Node 99 be Node 97's right child
3. Let Node 97 be Node 93's right child.

```
def right_rotate(self, cur):
        x = cur.left           # x = 97 (left child of 99) [Cur => 99]
        xRChild = x.right      # xRChild = None (97 has no right child)

        # Perform rotation
        cur.left = xRChild  # Step1:99.left = None (Assigning 97's right child to 99's left)
        x.right = cur       # Step2:97.right = 99 (99 becomes the right child of 97)

        # Update heights (99[cur.height] and 97 [x.height]
        cur.height = 1 + max(self.get_height(cur.left), self.get_height(cur.right))
        x.height = 1 + max(self.get_height(x.left), self.get_height(x.right))

        # Return the new root
        return x
```

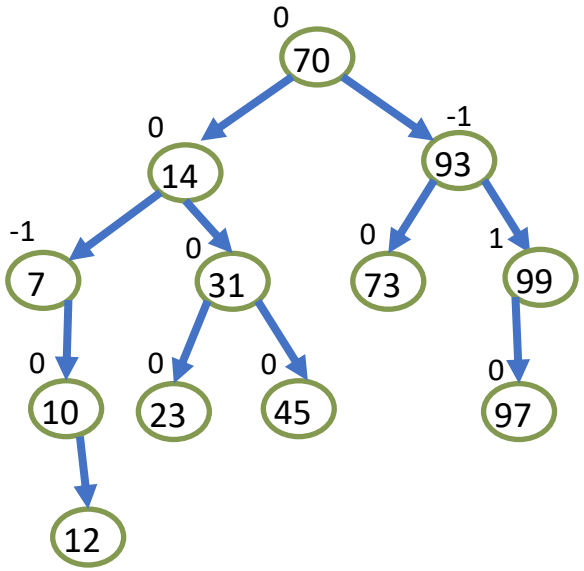


Node Insertion: Case 3 (RR)

Case 3 (b) RR Case: Insert node 12

Nodes along the insertion path from the root to the expectant parent node, 10

- Balance factor of 7 is -1 before insert the node
- Insertion occurs at 7 's higher subtree
- After insertion, the height of left and right subtrees of every node still differ by more than one.
- Left Rotation about node 7 is required to rebalance the tree.

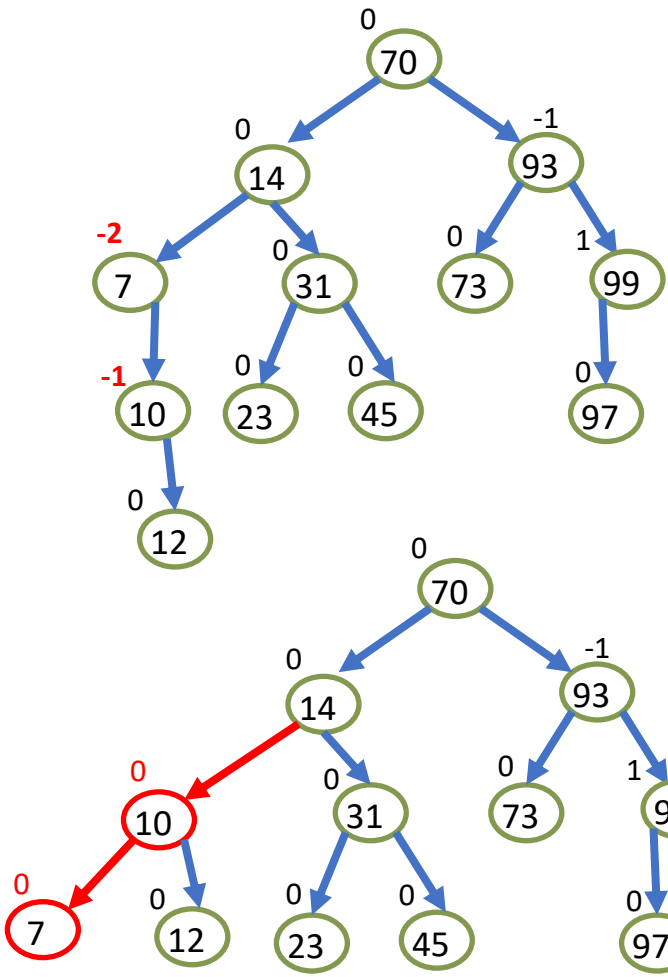


Node Insertion: Case 3 (RR)-Left Rotation

Rotate Left about Node 7

1. Let Node 10's left child be Node 7's right child (None in this case)
   *those nodes are between 7 and 10
2. Let Node 7 be Node 10's left child
3. Let Node 10 be Node 14's left child.

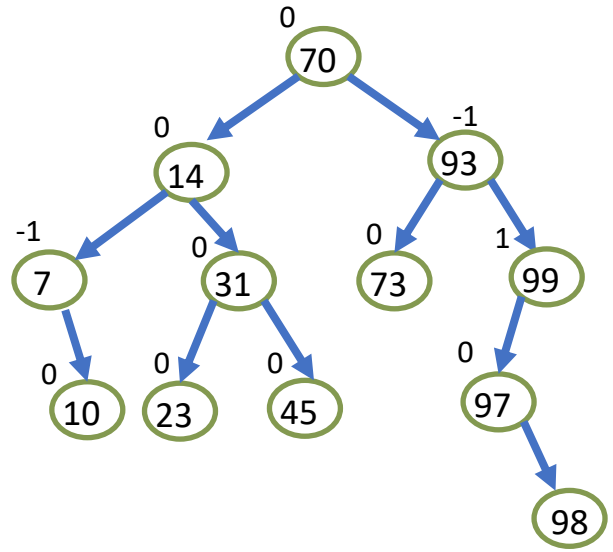
```
def left_rotate(self, cur):
        x = cur.right      # x = 10 (right child of 7) [Cur => 7]
        xLChild = x.left   # xLChild = None (10 has no left child)

        # Perform rotation
        cur.right = xLChild #Step1: 7.right = None (Assigning 10's left child to 7's right)
        x.left = cur        #Step2: 10.left = 7 (7 becomes the left child of 10)

        # Update heights (7[cur.height] and 10 [x.height]
        cur.height = 1 + max(self.get_height(cur.left), self.get_height(cur.right))
        x.height = 1 + max(self.get_height(x.left), self.get_height(x.right))

        # Return the new root
        return x
```



Node Insertion: Case 3 (LR)

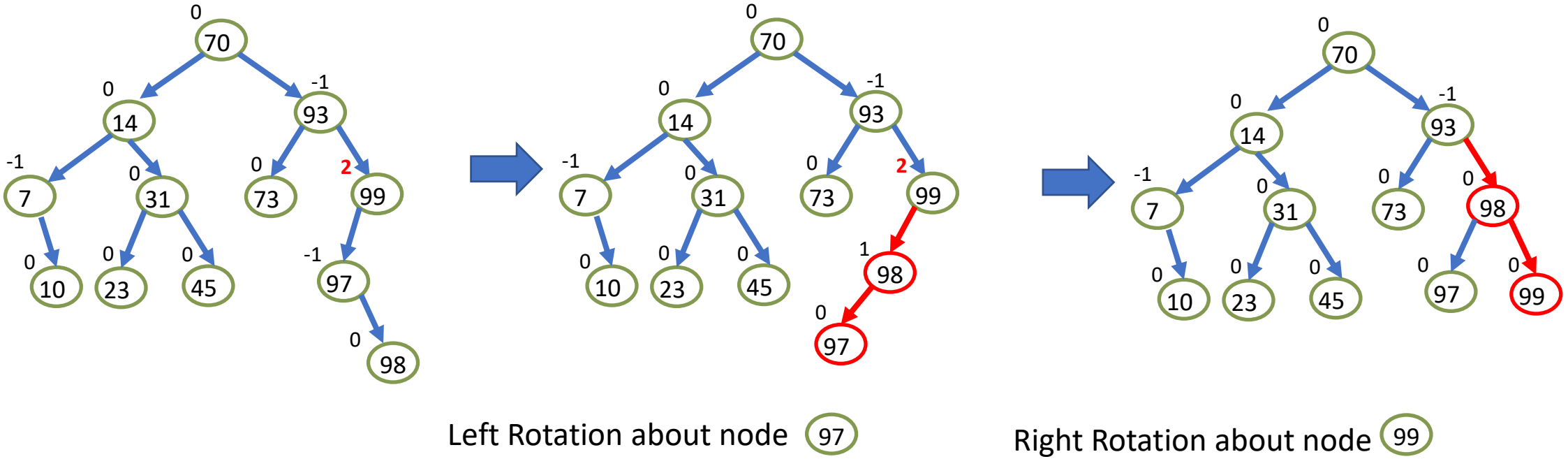
Nodes along the insertion path from the root to the expectant parent node, 97

- Balance factor of 99 is 1 before insert the node
- Insertion occurs at 99 's higher subtree
- After insertion, the height of left and right subtrees of every node still differ by more than one.
- Two rotations are required:
- **Left Rotation** about node 97 .
- **Right Rotation** about node 99 .



Node Insertion: Case 3 (LR)- Two Rotation

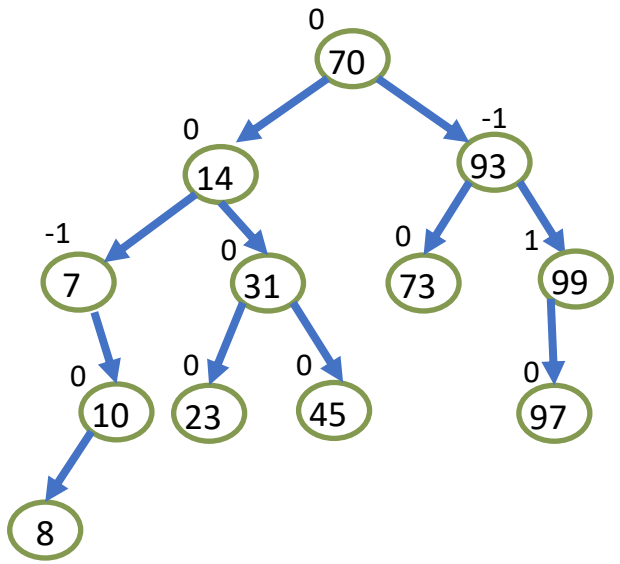
- Two rotations are required:
  - **Left Rotation about node** 97.
  - **Right Rotation about node** 99.



Node Insertion: Case 3 (RL)

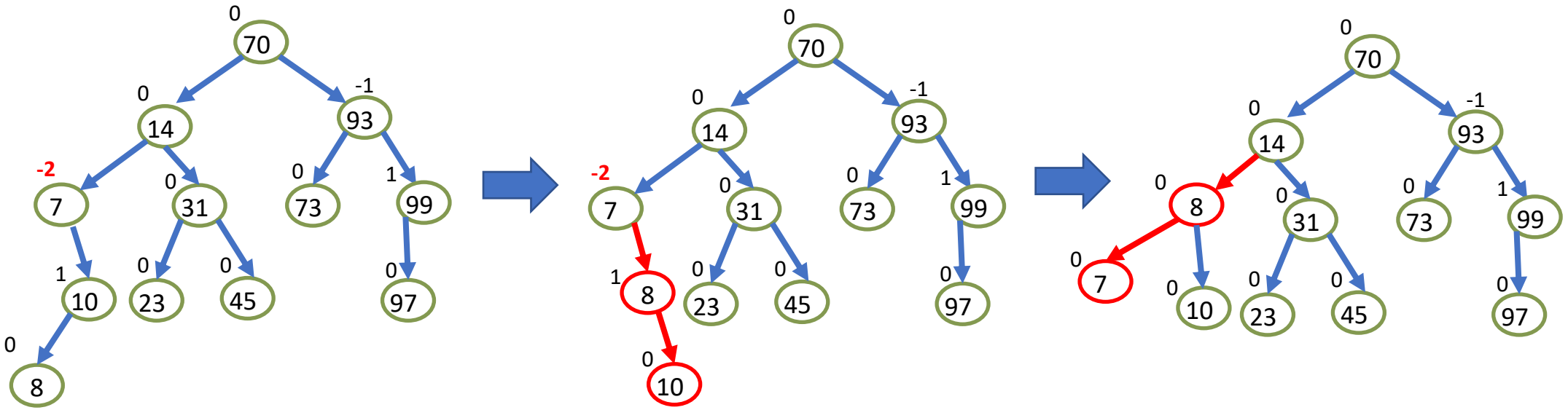
Nodes along the insertion path from the root to the expectant parent node, 10

- Balance factor of 7 is -1 before insert the node
- Insertion occurs at 7 's higher subtree
- After insertion, the height of left and right subtrees of every node still differ by more than one.
- Two rotations are required:
- **Right Rotation** about node 10
- **Left Rotation** about node 7



Node Insertion: Case 3 (RL)- Two Rotation

- Two rotations are required:
  - **Right Rotation** about node 10.
  - **Left Rotation** about node 7.



Other Balanced Search Trees

- Red-Black Trees
- Splay Trees
- Scapegoat Trees
- B-Trees
- Treaps
- etc.

Other Balanced Search Trees

1. Height of a Tree
2. Tree Balancing
3. AVL (balancing issue on insertion operation)
    1. Balanced Factor
    2. Rotation operation
    3. Understand/ learn how people analyze the problem
        - How many cases are there?
        - How to resolve each case?
        - Is there any similar operation? Etc.