SC1007
Data Structures and Algorithms

Tree Balancing Problem

College of Engineering

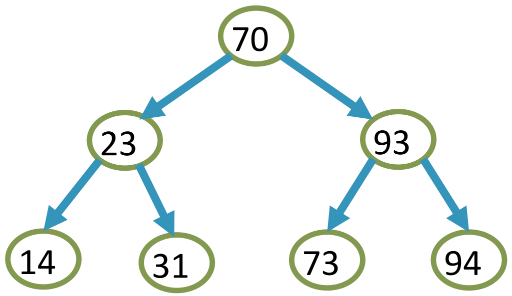
School of Computer Science and Engineering

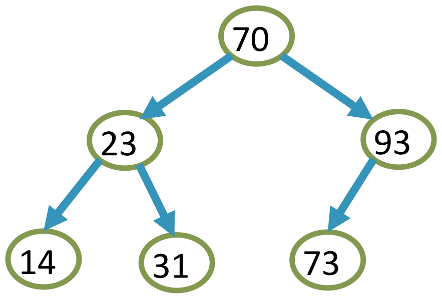
Dr. Loke Yuan Ren

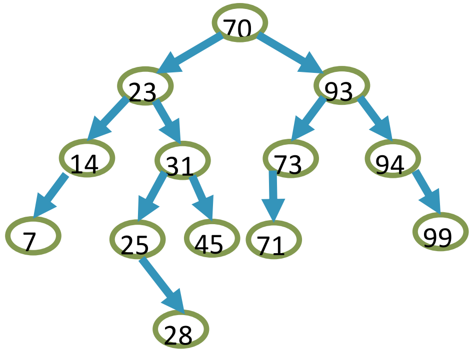
Lecturer

yrloke@ntu.edu.sg

Terminology

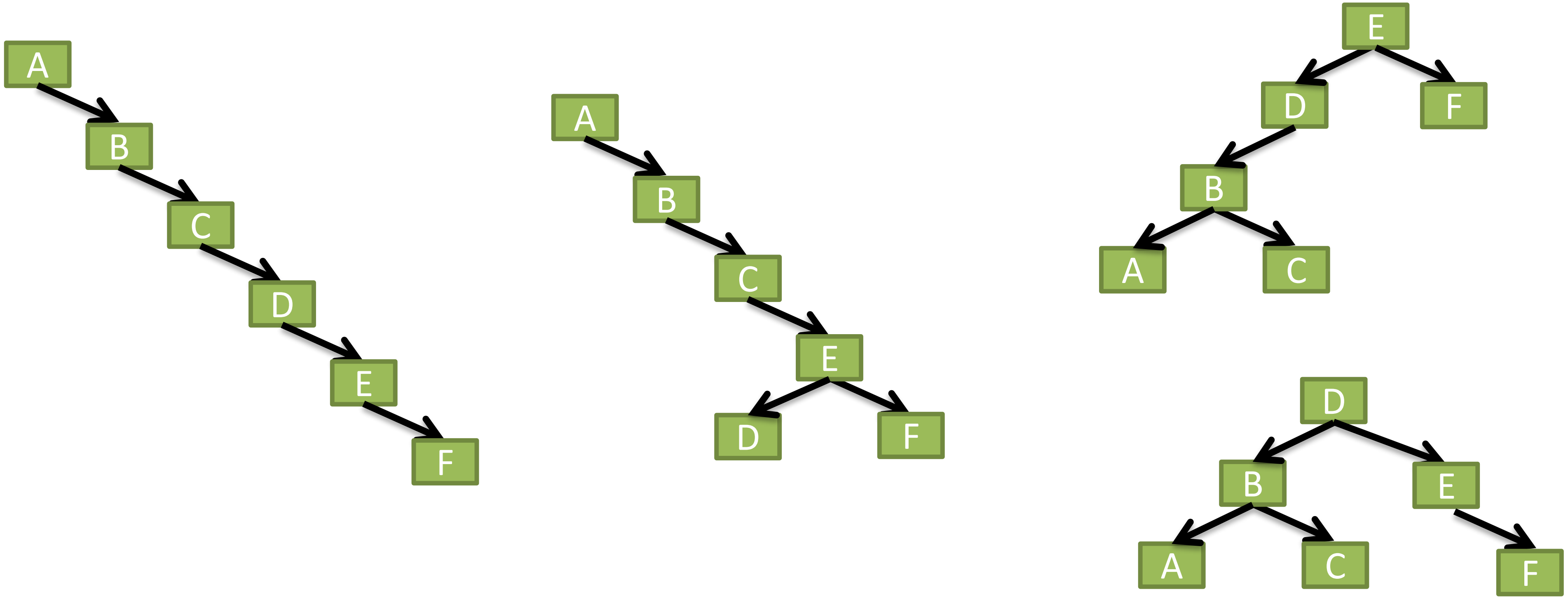






- The Height of a tree: The number of **edges** on the longest path from the root to a leaf
- The Depth/Level of a node: The number of edges from the node to the root of its tree.
- Empty Binary Tree: A binary tree with no nodes. It is still considered as a tree.
- Full Binary Tree: A binary tree of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

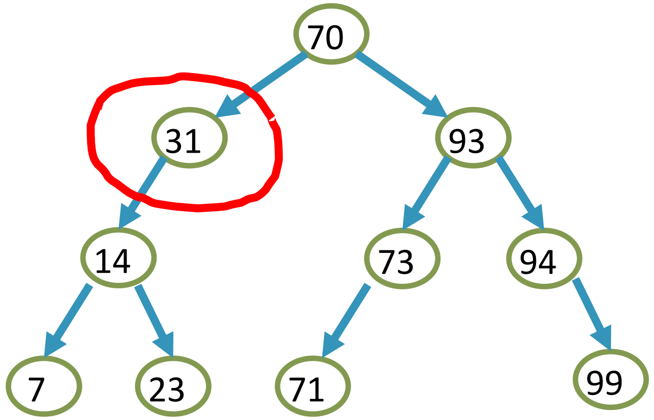
The ‘Good’ and ‘Bad’ Binary Search Trees



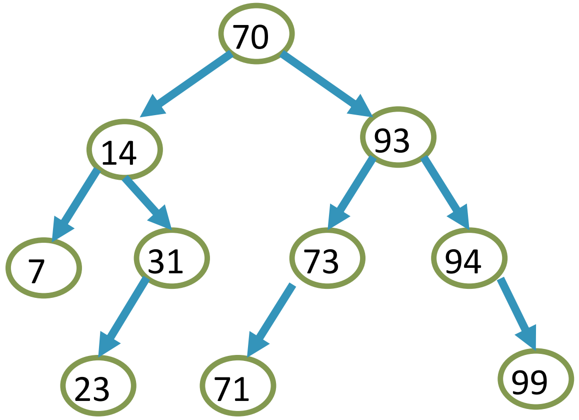
Tree Balancing

- 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

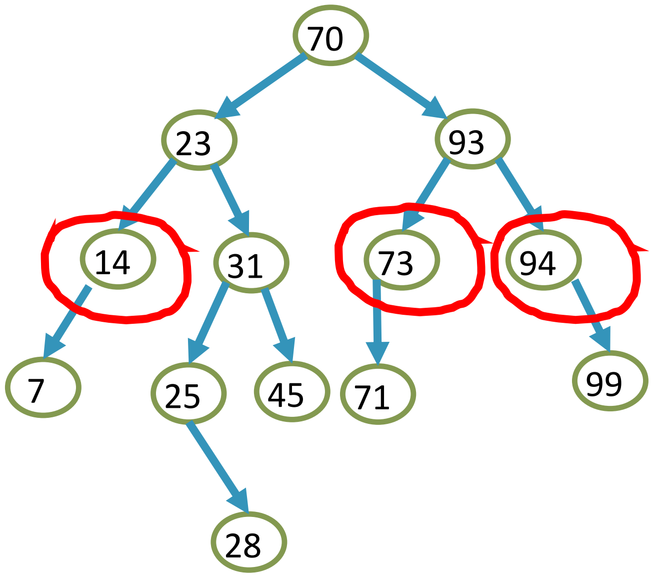
Unbalanced BST



Most Balanced BST



Balanced BST

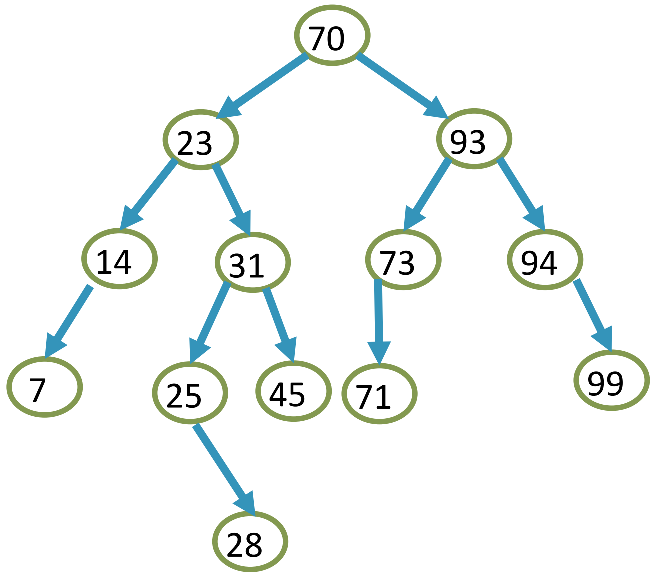


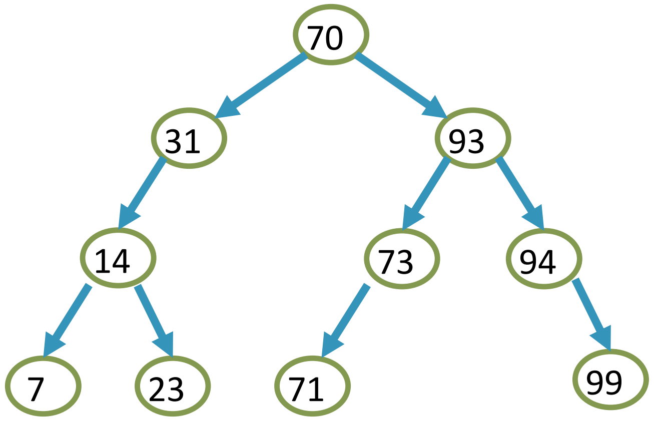
Most Balanced BST: Each tree node has exactly two child nodes except for the bottom 2 levels

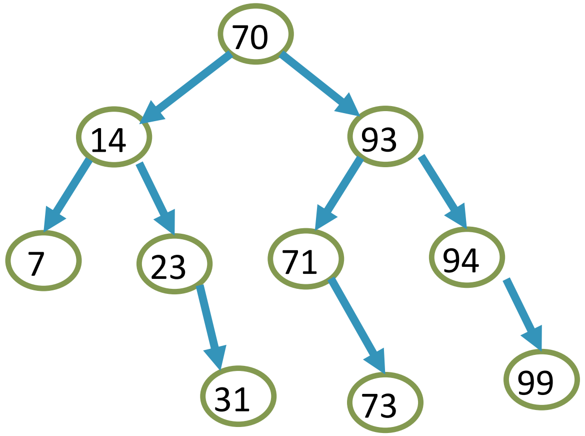
Tree Balancing

- How do you 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

AVL Tree







- Sort all the data in an array 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 an array

3. Take the middle element of the array as the root of the tree: 70

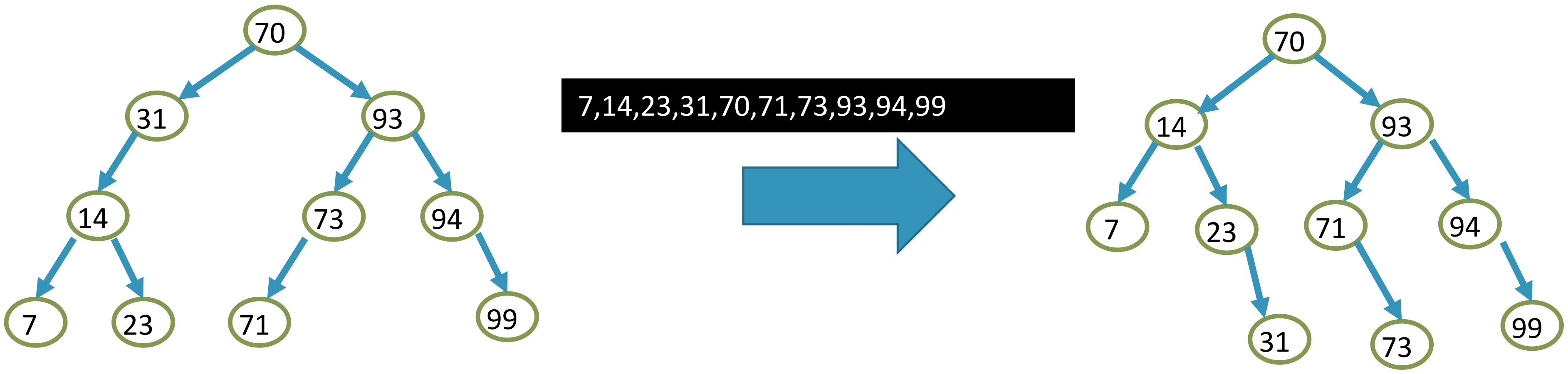
4. The first half of the array is used to build the left subtree of 70

   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.



Most Balanced BST: Each tree node has exactly two child nodes except for the bottom 2 levels

```
void treeBalance(int data[], int first, int last,
BTNode *root)
{
   if(last>=first){
     int middle = (first+last)/2;
     insertBSTNode(root,data[middle]);
    treeBalance(data,first,middle-1,root->left);
     treeBalance(data,middle+1,last,root->right);
     }
}
```

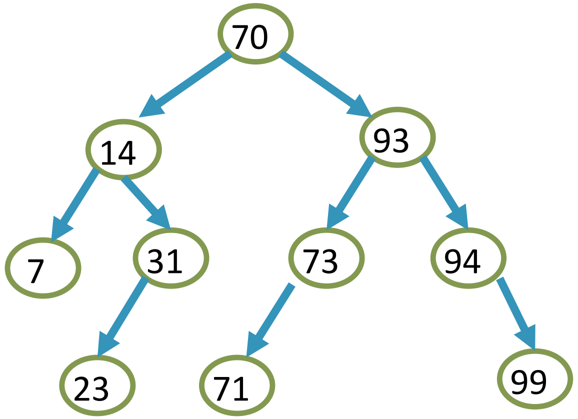
1) Need extra array to store sorted data
2) Rebuild the whole tree

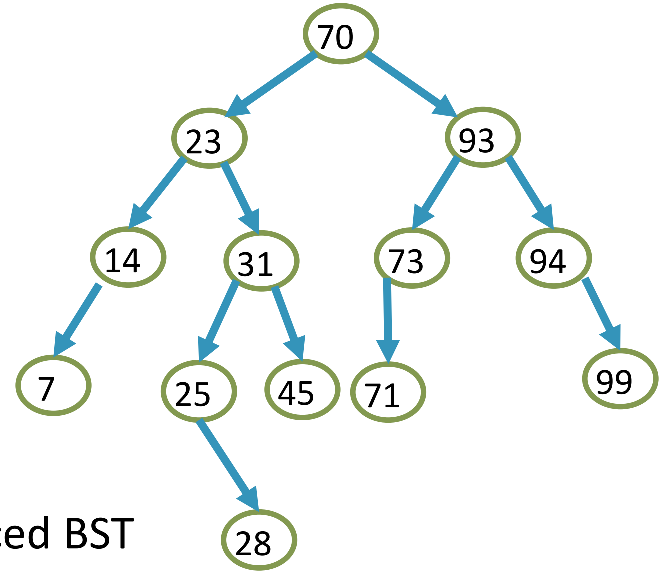
Time complexity: Θ(nlogn)
Space complexity: Θ(n) (additional array)

AVL Tree

- 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

Most Balanced BST



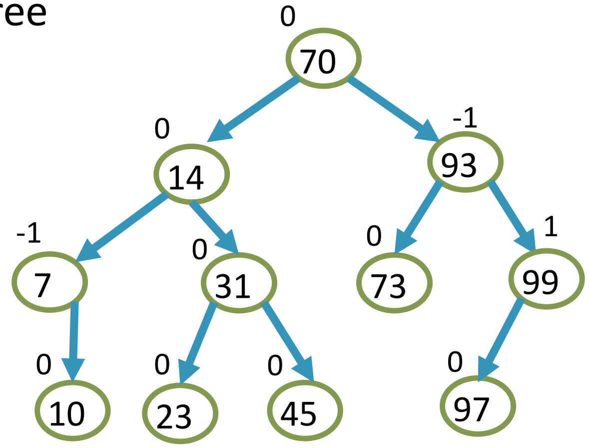


Balanced BST

Balance Factor

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

```
typedef struct _btnode{
    int item;
    struct _btnode *left;
    struct _btnode *right;
    int height;
} BTNode;
```



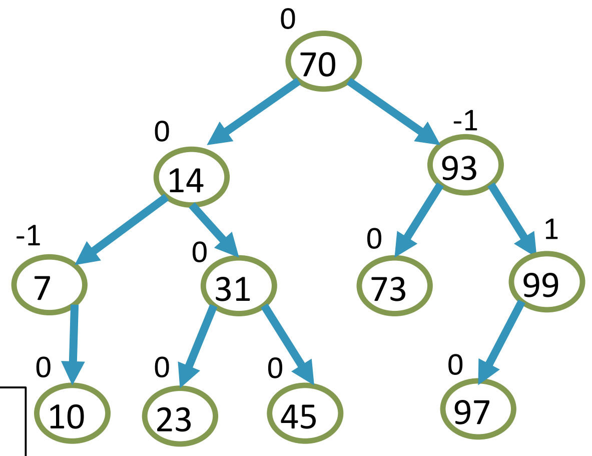
- An AVL tree: The height of left and right subtrees of every node differ by at most one.
  - Balance Factor of each node in an AVL tree can only be -1, 0 or 1.
  - Node insertion or node removal from the tree may change the balance factor of its ancestors (from parent, grandparent, grand-grandparent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

Balance Factor

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

```
typedef struct _btnode{
    int item;
    struct _btnode *left;
    struct _btnode *right;
    int height;
} BTNode;
```

```
int getHeight(BTNode *cur)
{
    if(cur==NULL) return -1;
    return cur->height;
}
int balanceFactor(BTNode *cur)
{
    if(cur==NULL) return 0;
    return getHeight(cur->left)-getHeight(cur->right);
}
```



Node Insertion

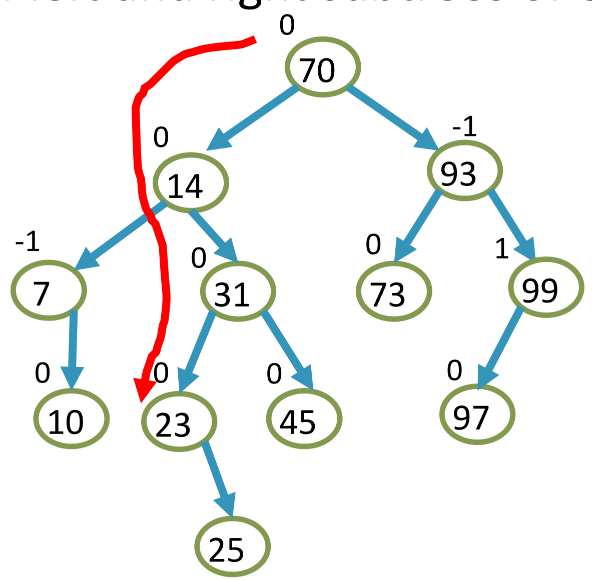
- Case 1: Balance factor of nodes along the insertion path from the root to the expectant parent node is zero.
- Case 2: Balance factor of nodes along the insertion path from the root to the expectant parent node is non-zero (1 or -1) but a new node is inserted at the shorter subtree.
- Case 3: Balance 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: 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 height of left and right subtrees of every node still differ by at most one.
- No balancing issue.

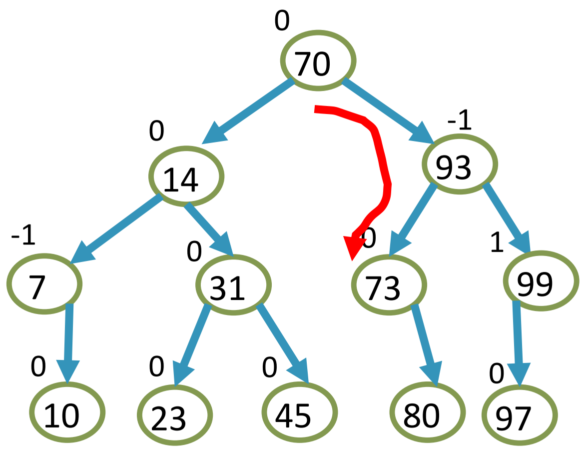


Node Insertion

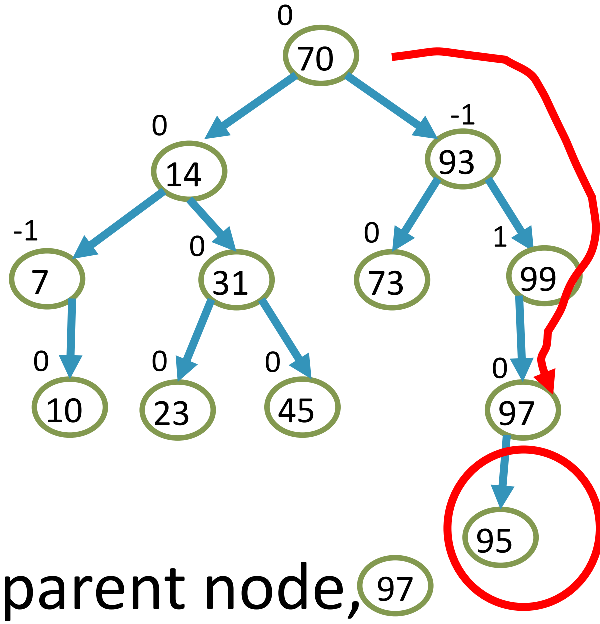
Case 2: Insert node 80

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

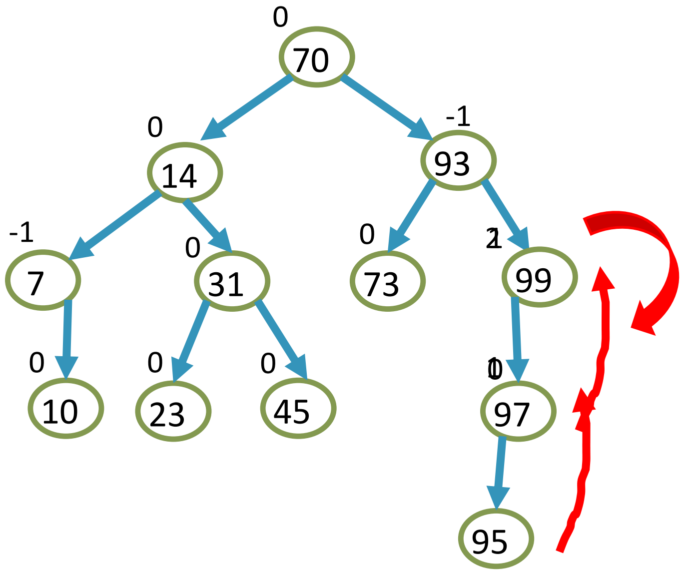


Case 3 (a) LL Case: Insert node 95

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.

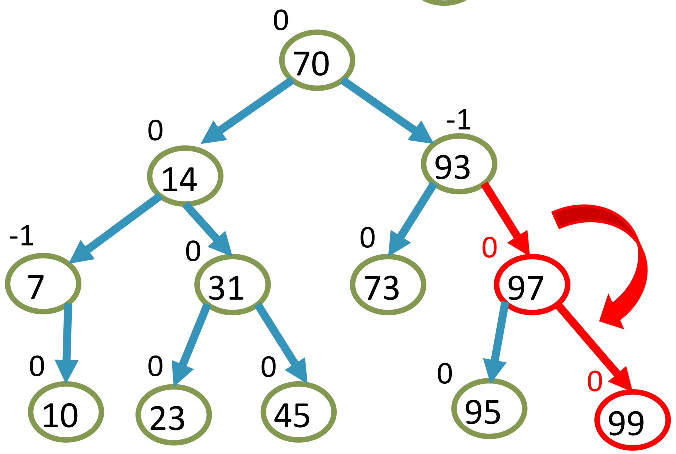
Right Rotation



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 imbalanced. BF>1

=> Some nodes in the left subtree need to shift to right subtree

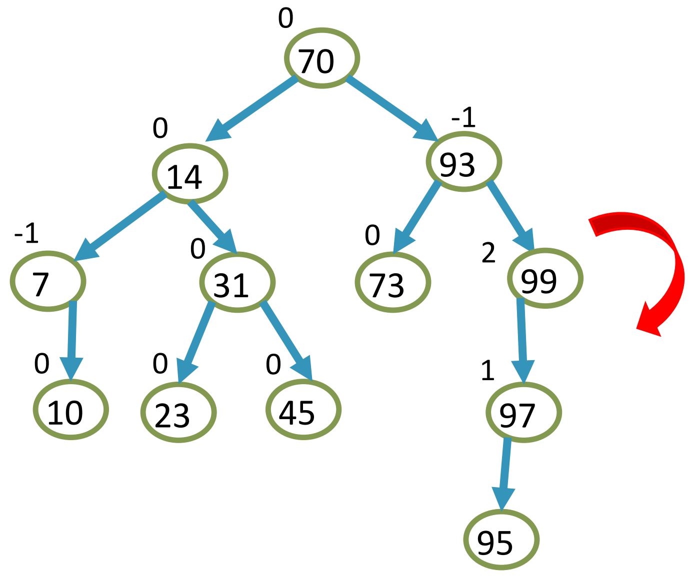


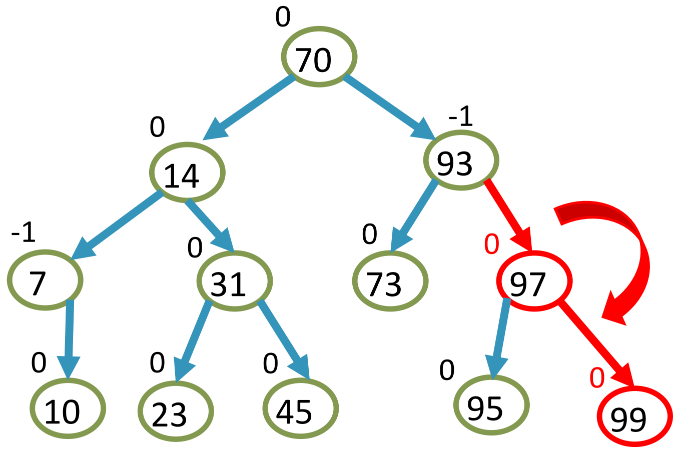
Rotate Right about Node 99

1. Let Node 97's right child be Node 99's left child (NULL 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.

```
typedef struct _btnode{
    int item;
    struct _btnode *left;
    struct _btnode *right;
    int height;
} BTNode;
```

Right Rotation





Rotate Right about Node 99

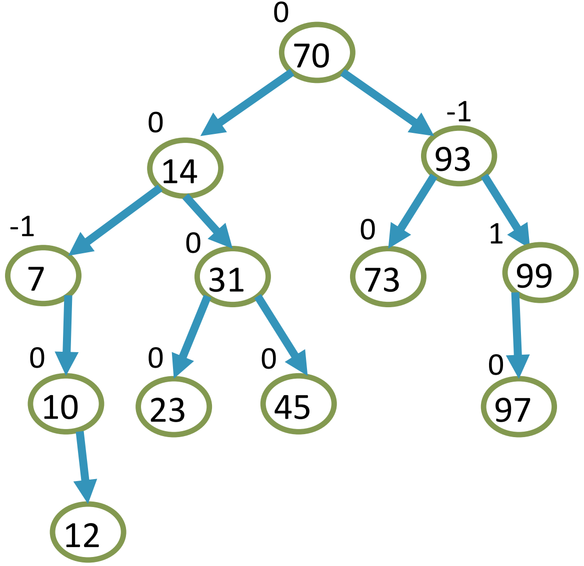
1. Let Node 97's right child be Node 99's left child (NULL in this case)

   *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.

```
BTNode* rightRotate(BTNode *cur)
{
    BTNode* x = cur->left;
    BTNode* xRChild = x->right;

    // rotation
    cur->left = xRChild; //Step 1
    x->right = cur;      //Step 2

    // Update heights
    cur->height = max(getHeight(cur->left), getHeight(cur->right))+1;
    x->height = max(getHeight(x->left), getHeight(x->right))+1;

    // Step 3: Return new root
    return x;
}
```

Node Insertion

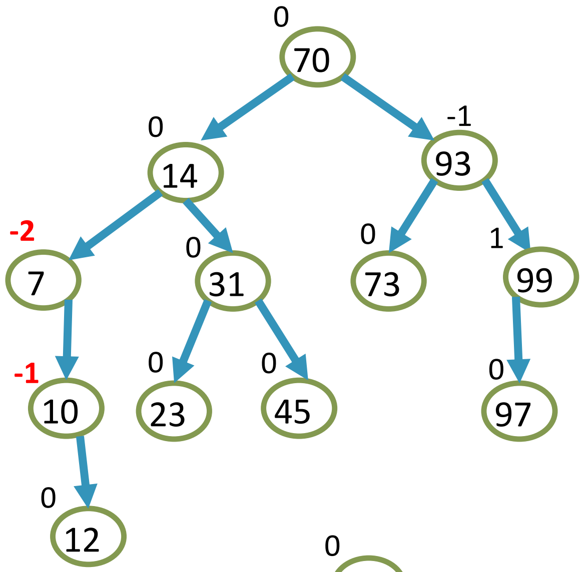


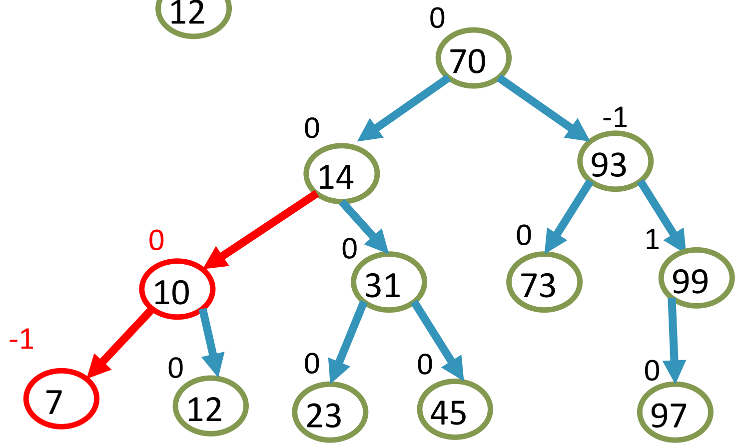
Case 3 (b) RR Case: Insert node 12

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.

Left Rotation





Rotate Left about Node 7

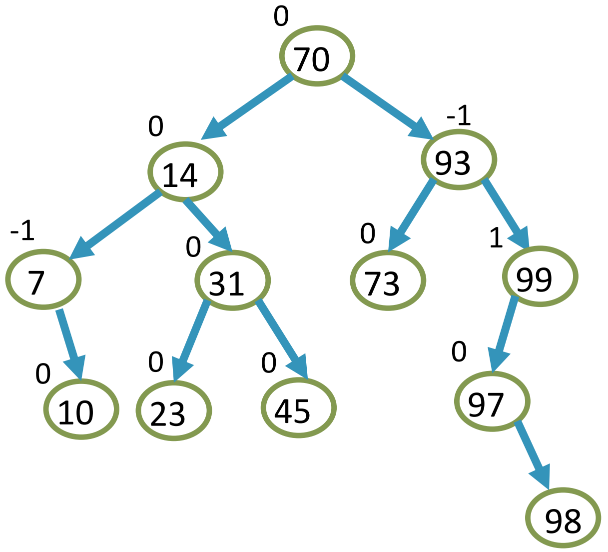
1. Let Node 10's left child be Node 7's right child (NULL in this case)
   *those nodes are between 7 and 10
2. Let Node 7 be Node 10's left child
3. Let Node 10 be Node 14's left child.

```
BTNode* leftRotate(BTNode *cur)
{
    BTNode* x = cur->right;
    BTNode* xLChild = x->left;

    // rotation
    cur->right = xLChild; //Step 1
    x->left = cur;       //Step 2

    // Update heights
    cur->height = max(getHeight(cur->left), getHeight(cur->right))+1;
    x->height = max(getHeight(x->left), getHeight(x->right))+1;

    // Step 3: Return new root
    return x;
}
```

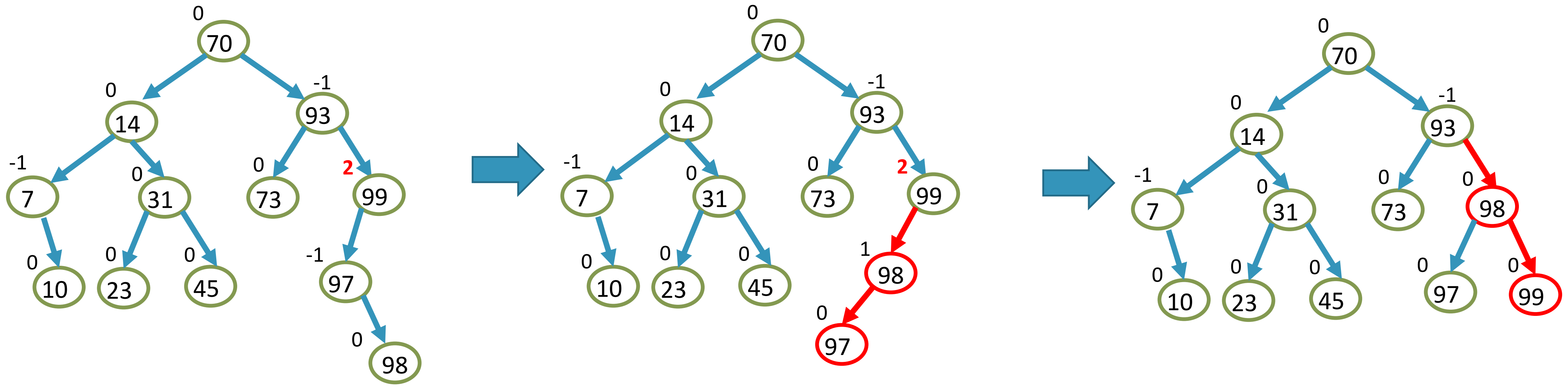
Case 3 (c) LR Case: Insert node 98

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.



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



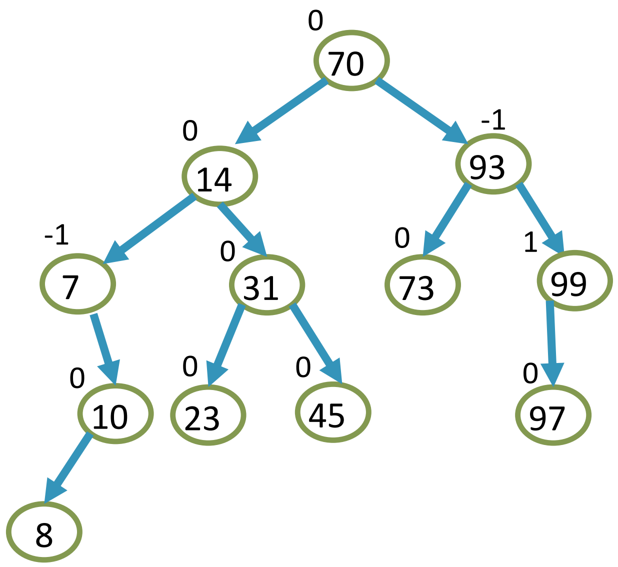
Left Rotation about node 97

Right Rotation about node 99

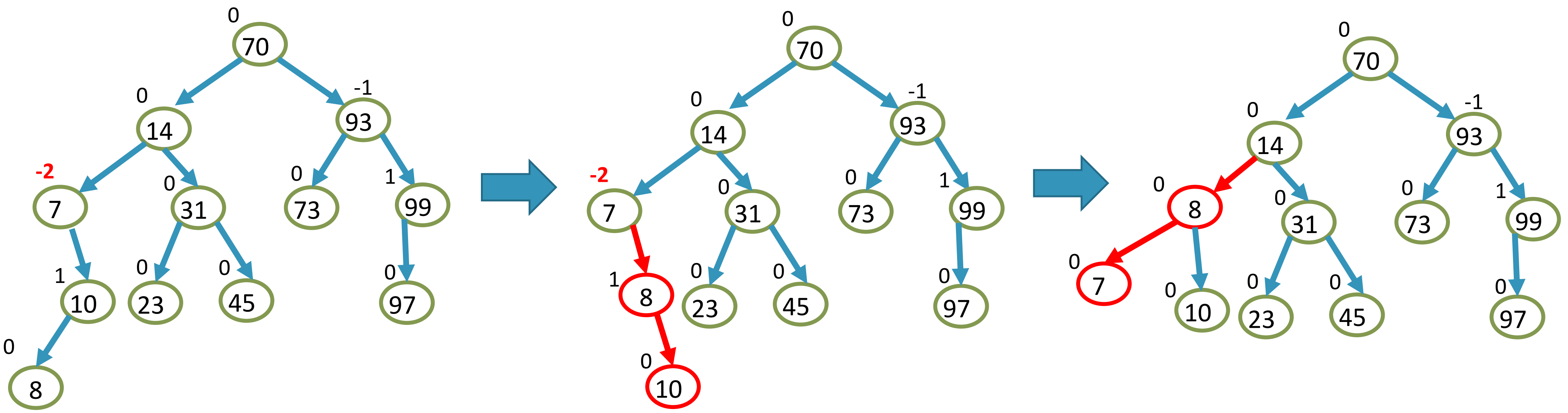
Case 3 (d) RL Case: Insert node 8

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.



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



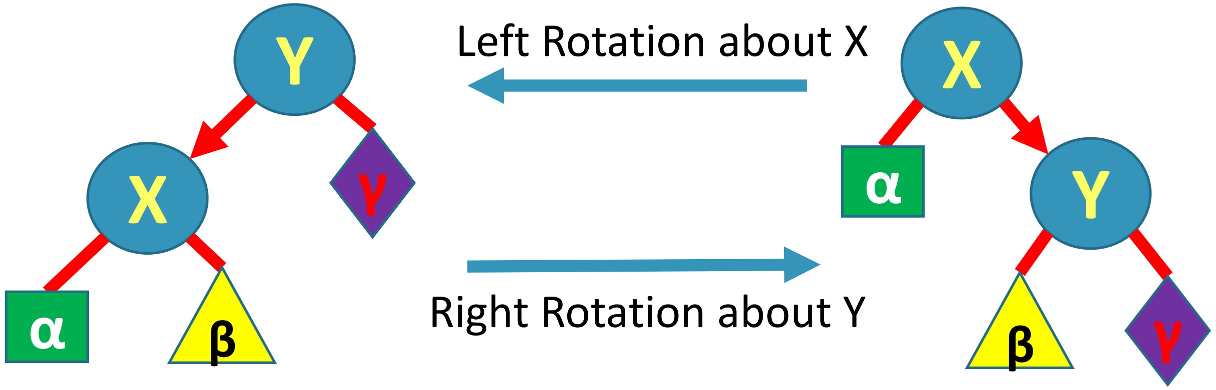
Right Rotation about node 10

Left Rotation about node 7

Other Balanced Search Trees

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

Summary



1. Binary Search Tree
   1. **Node's value is greater than all values in its left subtree.**
   2. **Node's value is less than all values in its right subtree.**
   3. **Both subtrees of the node are also binary search trees.**
2. Node Insertion for BST
3. Node Removal for BST (lab/tutorial questions)
4. Height of a Tree
5. Tree Balancing
6. AVL (balancing issue on insertion operation)
   1. Balanced Factor
   2. Rotation operation
   3. Understand/ learn how people analyze the problem
      1. How many cases are there?
      2. How to resolve each case?
      3. Is there any similar operation? Etc.