AVL Trees

Problem

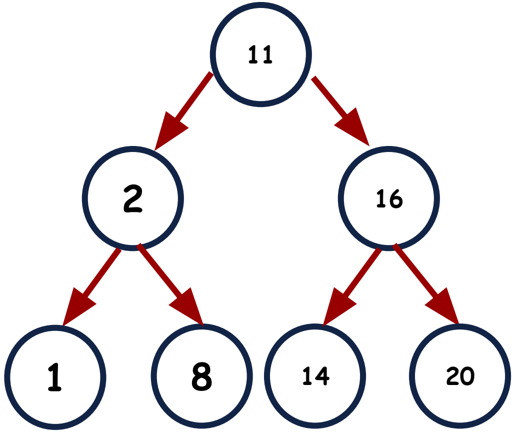
Let's create a Binary Search Tree from the following input.

Input: [11, 2, 16, 20, 14, 1, 8]

Binary Search Tree

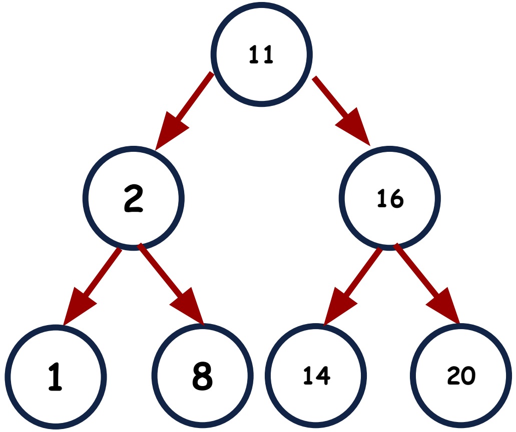
Let's create a **Binary Search Tree** from the following input.

Input: [11, 2, 16, 20, 14, 1, 8]



Binary Search Tree: Time Complexity?

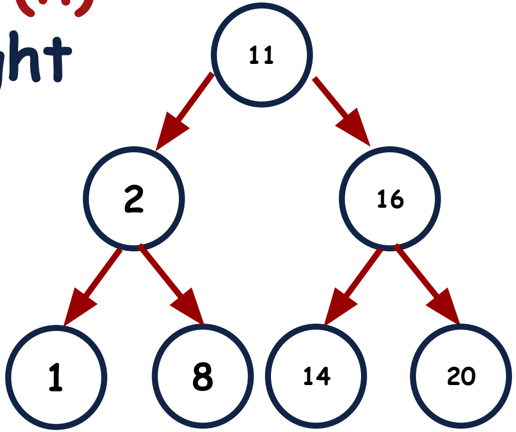
What will be the worst Time Complexity of searching in the Binary Search Tree?



Binary Search Tree: Time Complexity?

What will be the worst Time Complexity of searching in the Binary Search Tree?

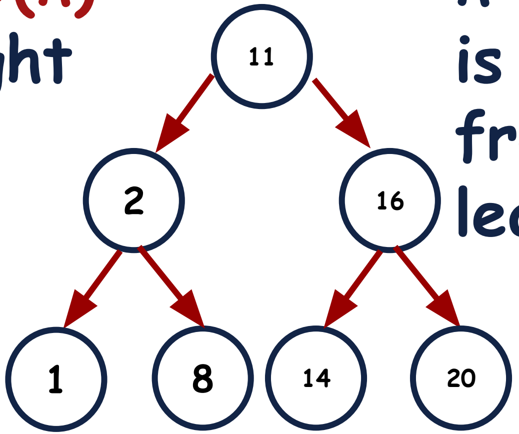
Time Complexity: O(h)
where h is the height of the tree



Binary Search Tree: Time Complexity?

What will be the worst Time Complexity of searching in the Binary Search Tree?

Time Complexity: O(h)
where h is the height of the tree



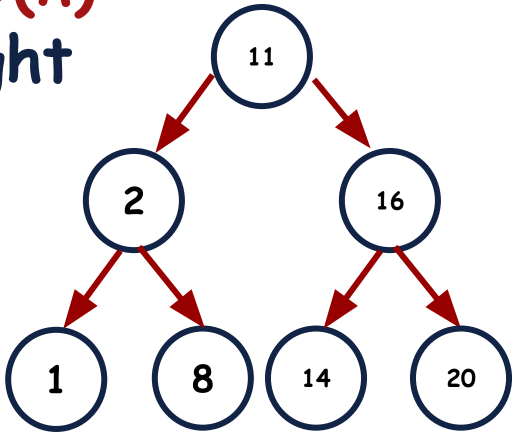
h = height of the tree is the longest path from the root to a leaf node.

Binary Search Tree: Time Complexity?

What will be the worst Time Complexity of searching in the Binary Search Tree?

Time Complexity: O(h)
where h is the height of the tree

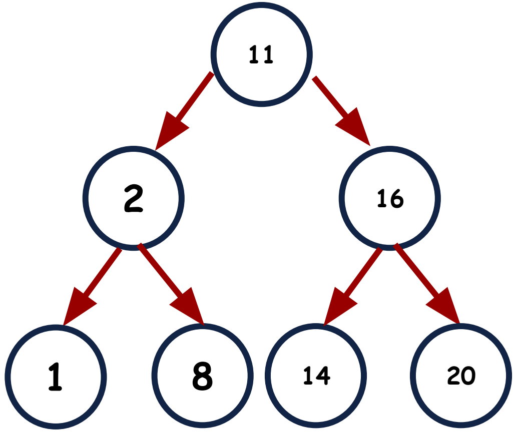
$h = \log_2(n)$



Binary Search Tree: Time Complexity?

What will be the **worst Time Complexity** of searching in the Binary Search Tree?

Time Complexity:
$O(\log_2(n))$



Binary Search Tree

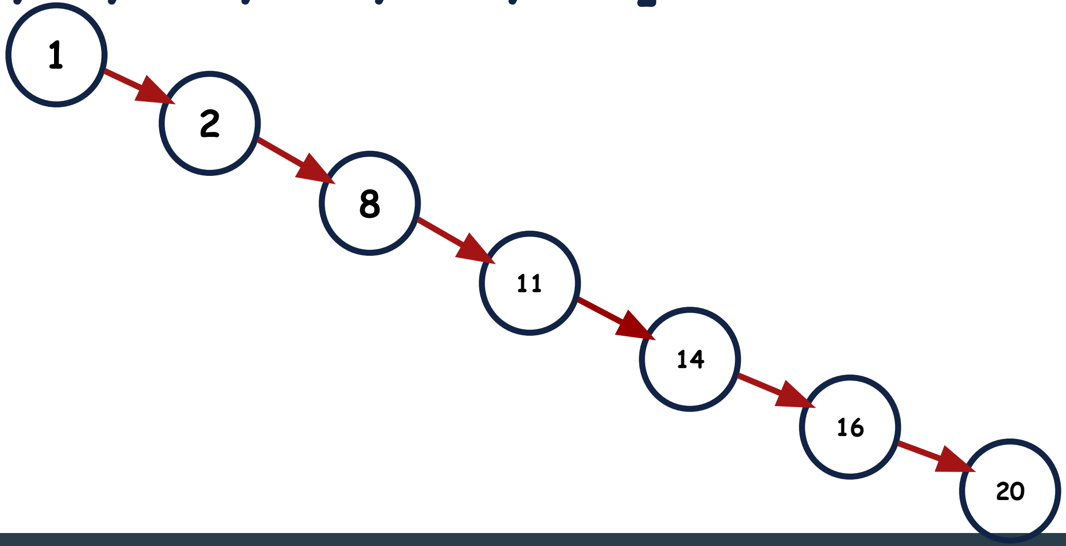
Let's create a Binary Search Tree from the following input.

Input: [1, 2, 8, 11, 14, 16, 20]

Binary Search Tree

Let's create a Binary Search Tree from the following input.

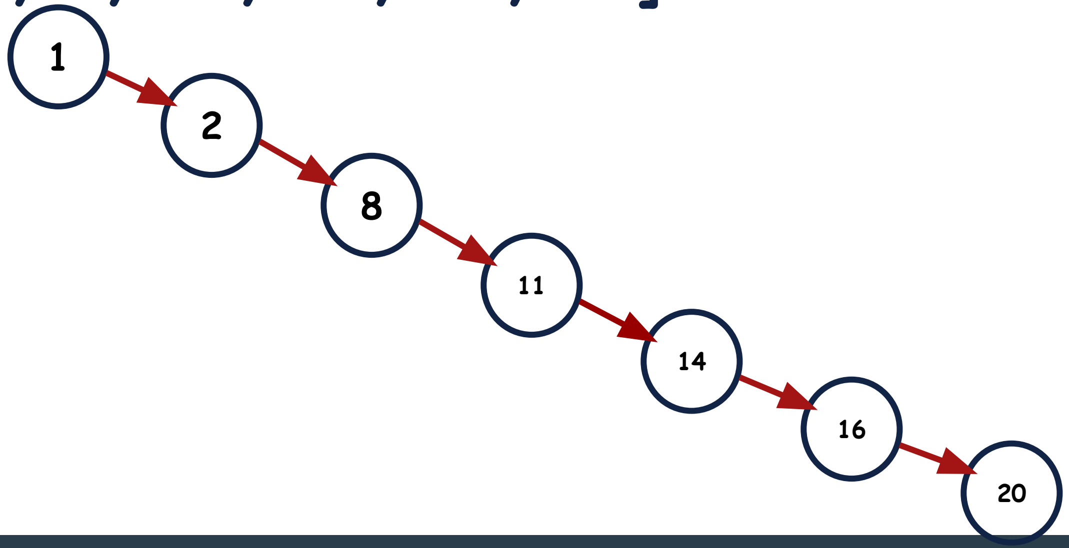
Input: [1, 2, 8, 11, 14, 16, 20]



Binary Search Tree: Time Complexity?

What will be the worst Time Complexity of searching in this Binary Search Tree?

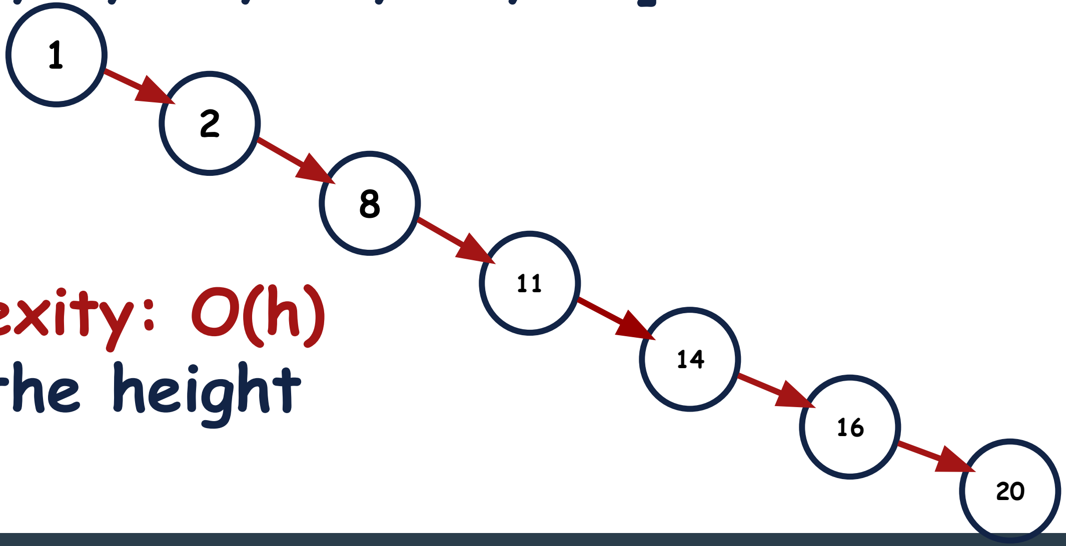
Input: [1, 2, 8, 11, 14, 16, 20]



Binary Search Tree: Time Complexity?

What will be the worst Time Complexity of searching in this Binary Search Tree?

Input: [1, 2, 8, 11, 14, 16, 20]

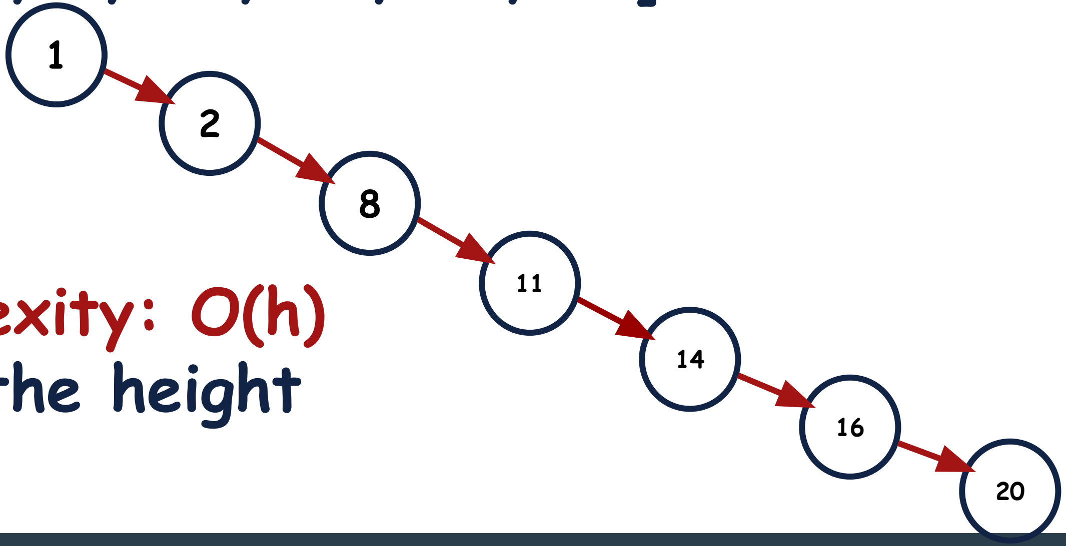


Time Complexity: O(h)
where h is the height of the tree

Binary Search Tree: Time Complexity?

What will be the worst Time Complexity of searching in this Binary Search Tree?

Input: [1, 2, 8, 11, 14, 16, 20]



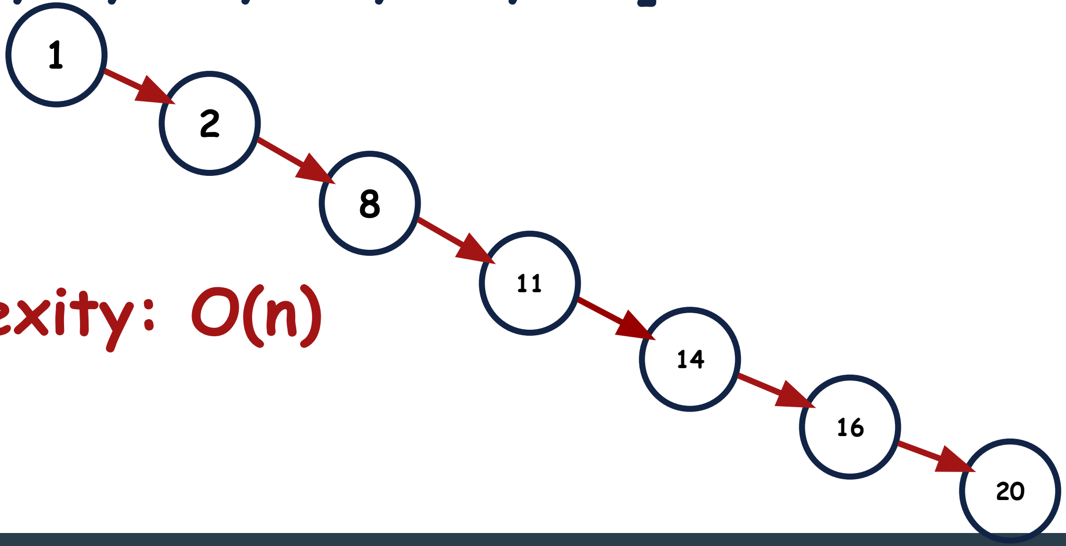
$h = n$

Time Complexity: O(h)
where h is the height of the tree

Binary Search Tree: Time Complexity?

What will be the **worst Time Complexity** of searching in this Binary Search Tree?

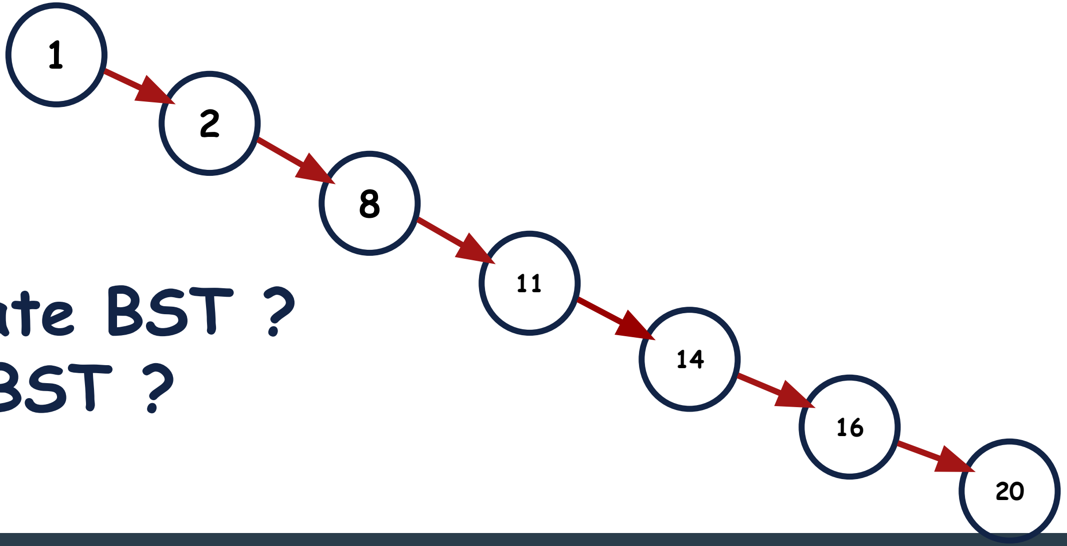
Input: [1, 2, 8, 11, 14, 16, 20]



Time Complexity: O(n)

Binary Search Tree: Food for Thought?

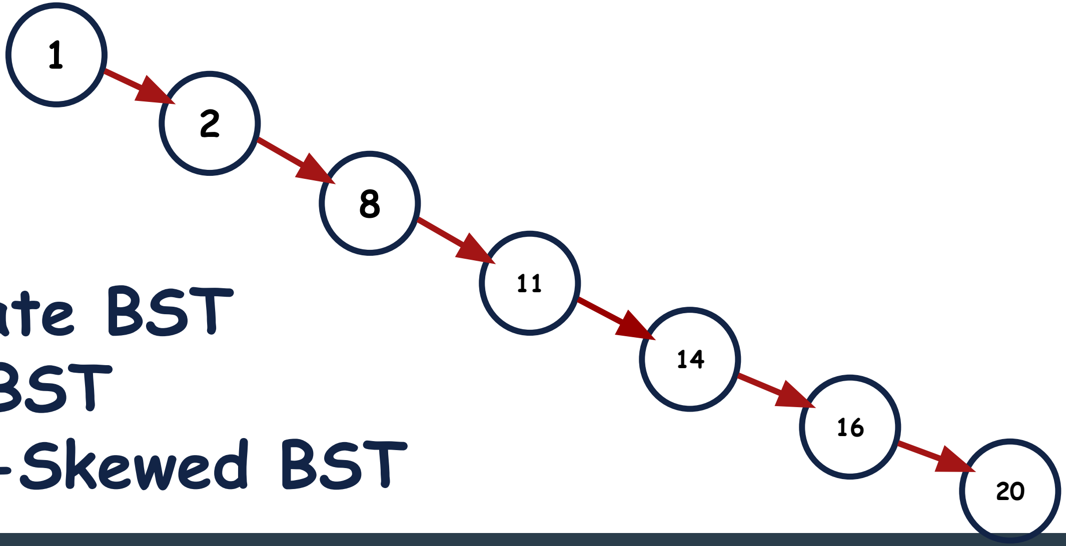
What is the special name of this Binary Search Tree?



a) Degenerate BST ?
b) Skewed BST ?

Binary Search Tree: Food for Thought?

What is the special name of this Binary Search Tree?

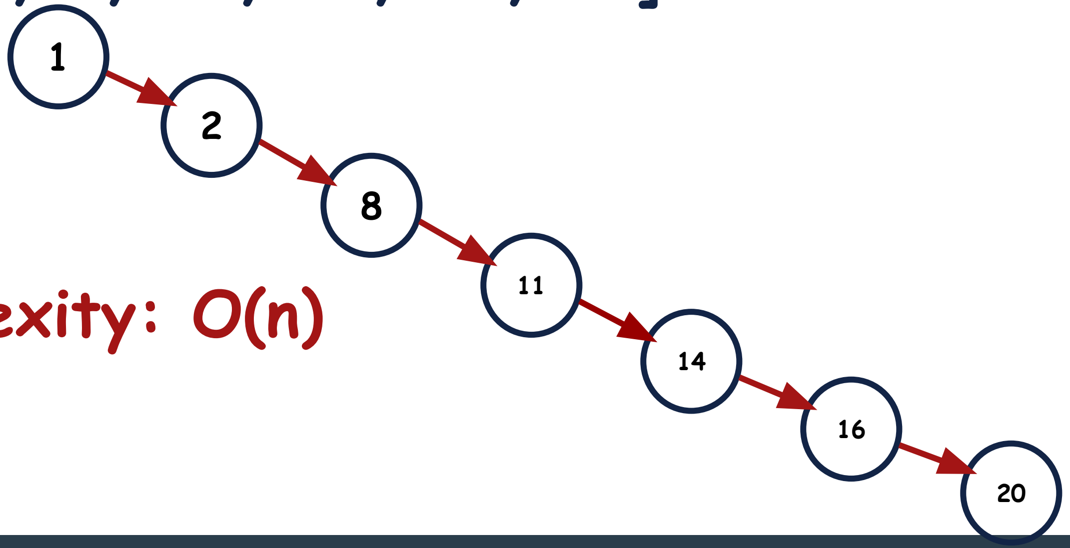


a) Degenerate BST
b) Skewed BST
Right-Skewed BST

Binary Search Tree: Linked List?

So, it means if the data is given in sorted order then there is not difference in BST and Linked List.

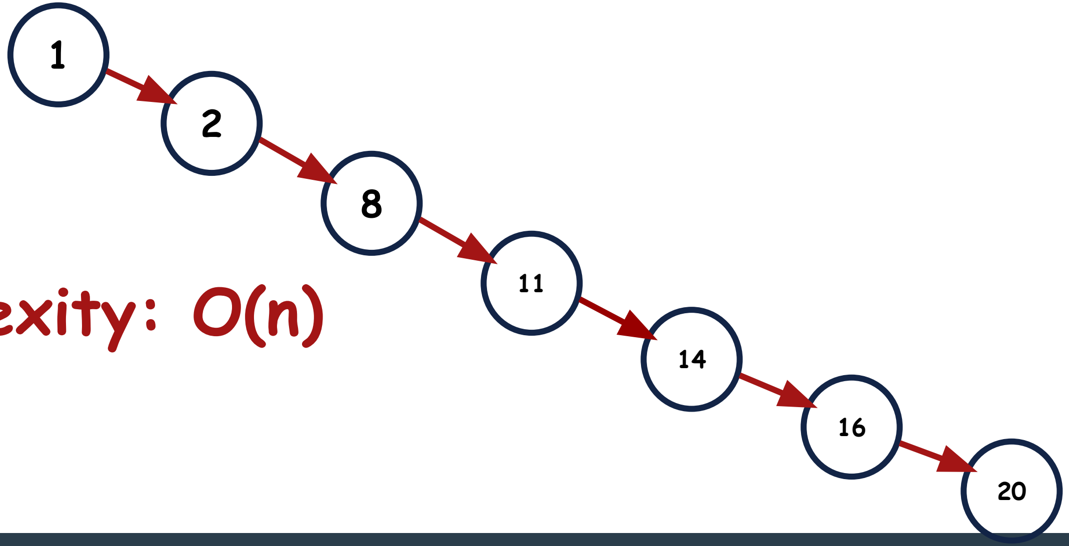
Input: [1, 2, 8, 11, 14, 16, 20]



Time Complexity: O(n)

Binary Search Tree: Linked List?

Now, the question is what is the benefit of Binary Search Tree when the worst case of Searching in a Binary Search Tree and Linked List is the same?

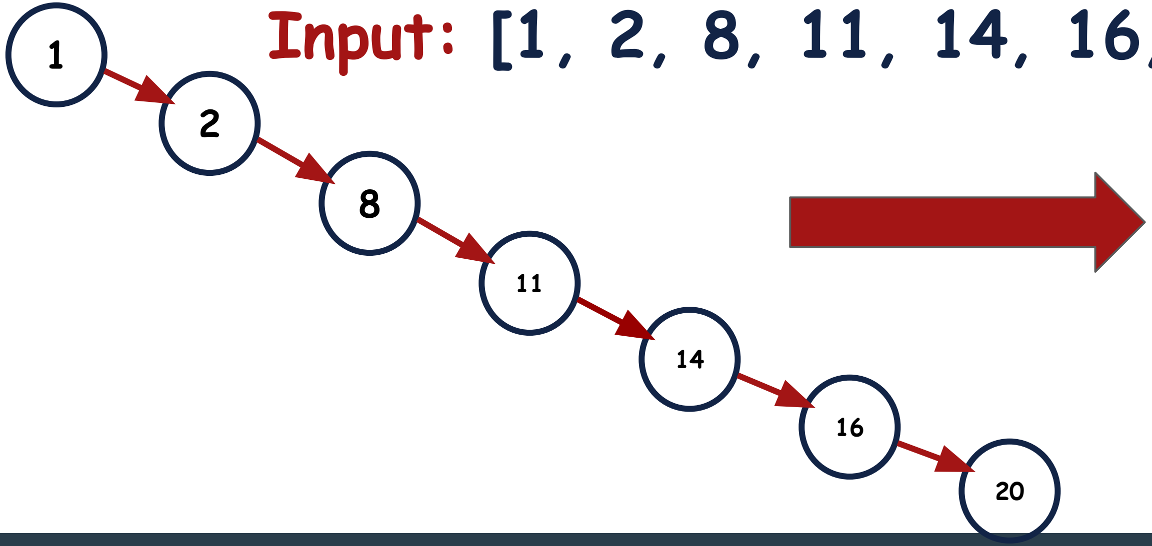


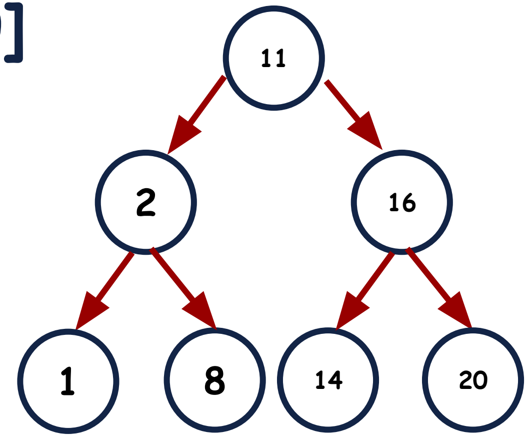
Time Complexity: O(n)

Binary Search Tree

There is no benefit of binary search trees unless we make a binary search tree whose height is always $\log_2(n)$ when the input data is given in any order.

Input: [1, 2, 8, 11, 14, 16, 20]

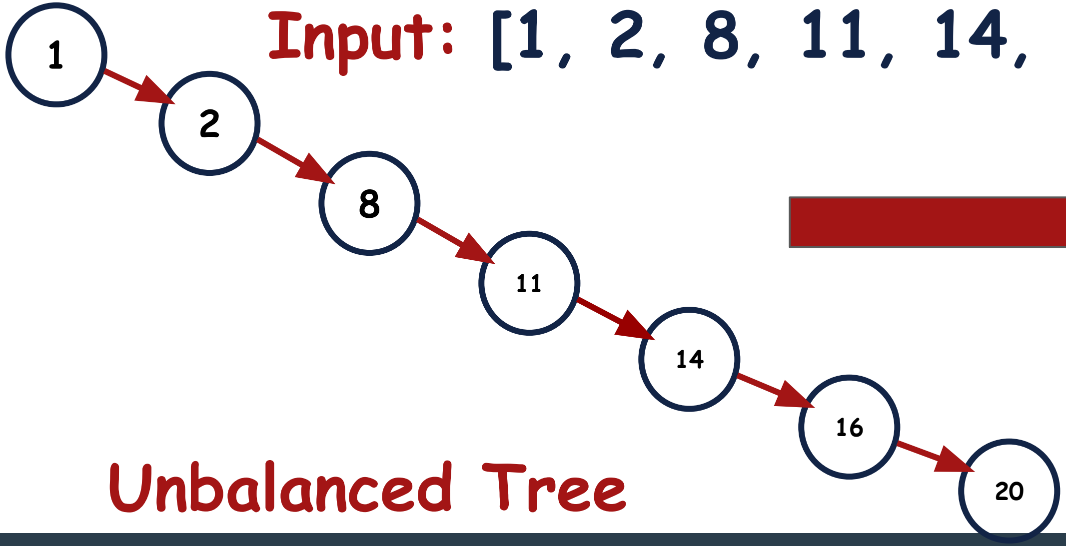




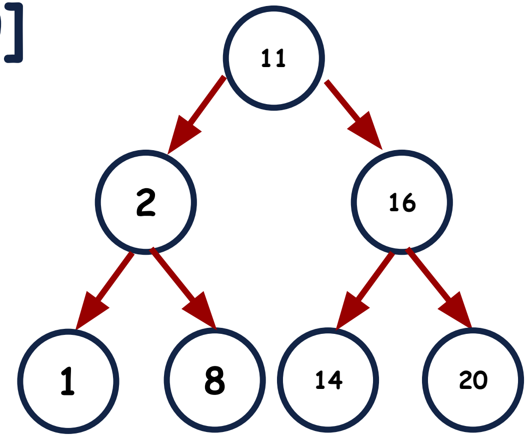
Binary Search Tree

There is no benefit of binary search trees unless we make a binary search tree whose height is always $\log_2(n)$ when the input data is given in any order.

Input: [1, 2, 8, 11, 14, 16, 20]



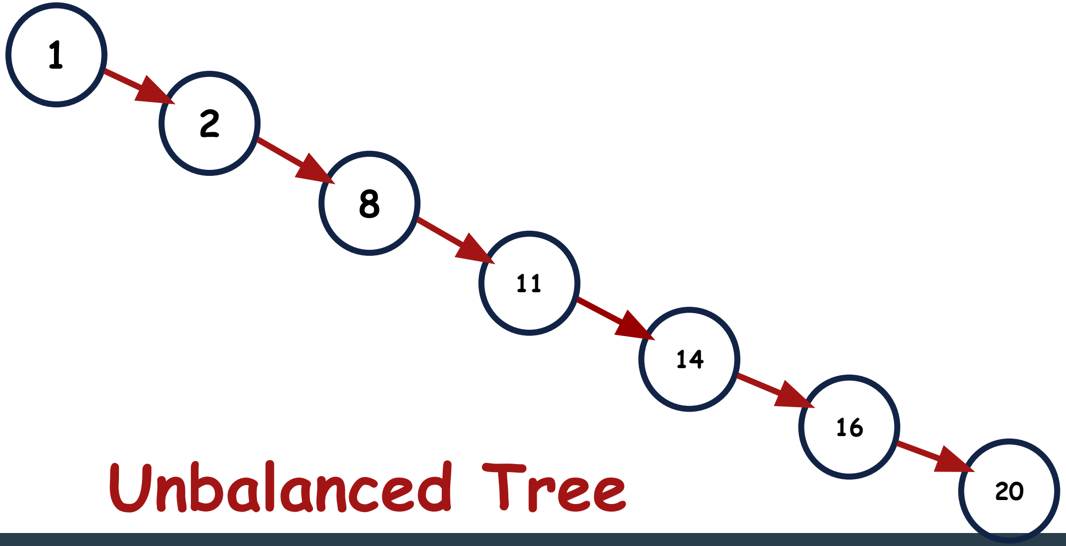
Unbalanced Tree



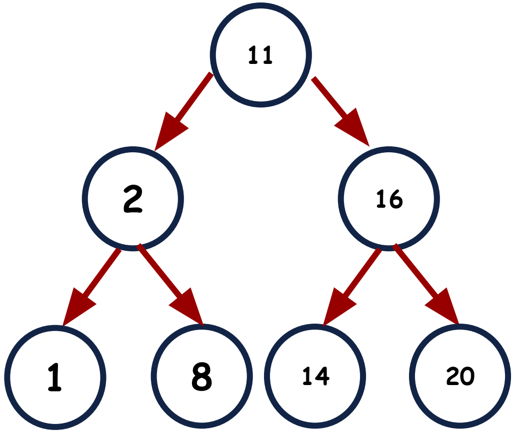
Balanced Tree

Balanced Binary Search Tree

Why is this tree a Balanced Binary Search Tree?



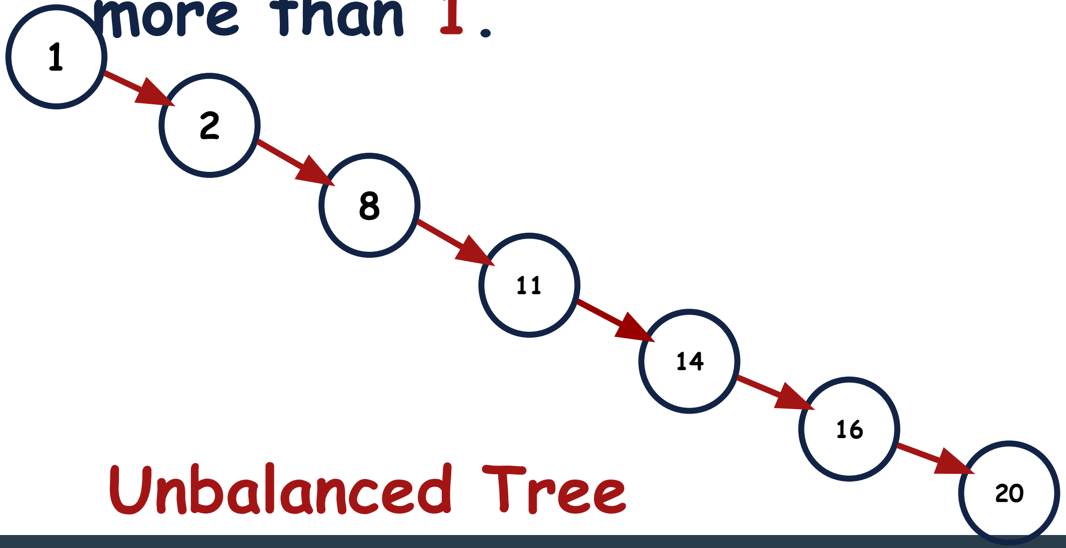
Unbalanced Tree



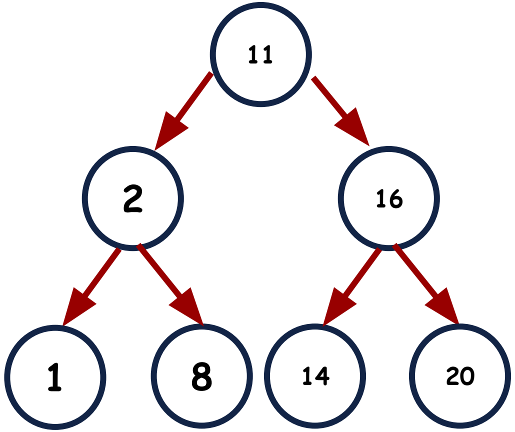
Balanced Tree

Balanced Binary Search Tree

A balanced binary search tree (height-balanced binary search tree) is defined as a tree in which the height of the left and right subtree of any node differ by not more than 1.



Unbalanced Tree



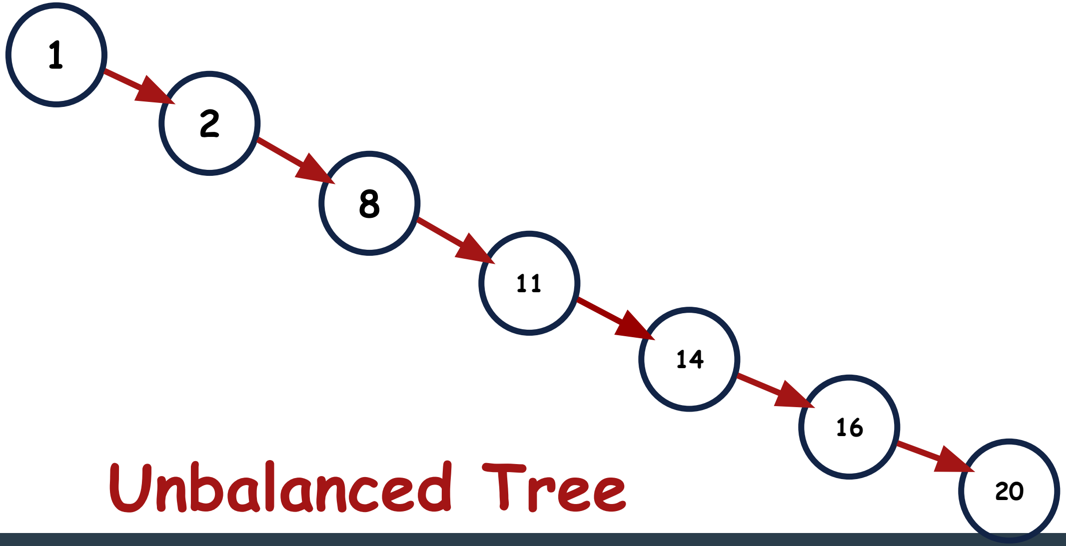
Balanced Tree

Balanced Binary Search Tree

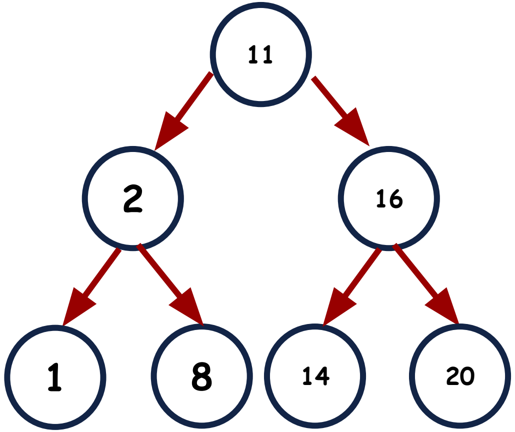
Balance Factor (BF)

=

Height of left SubTree - Height of right SubTree



Unbalanced Tree



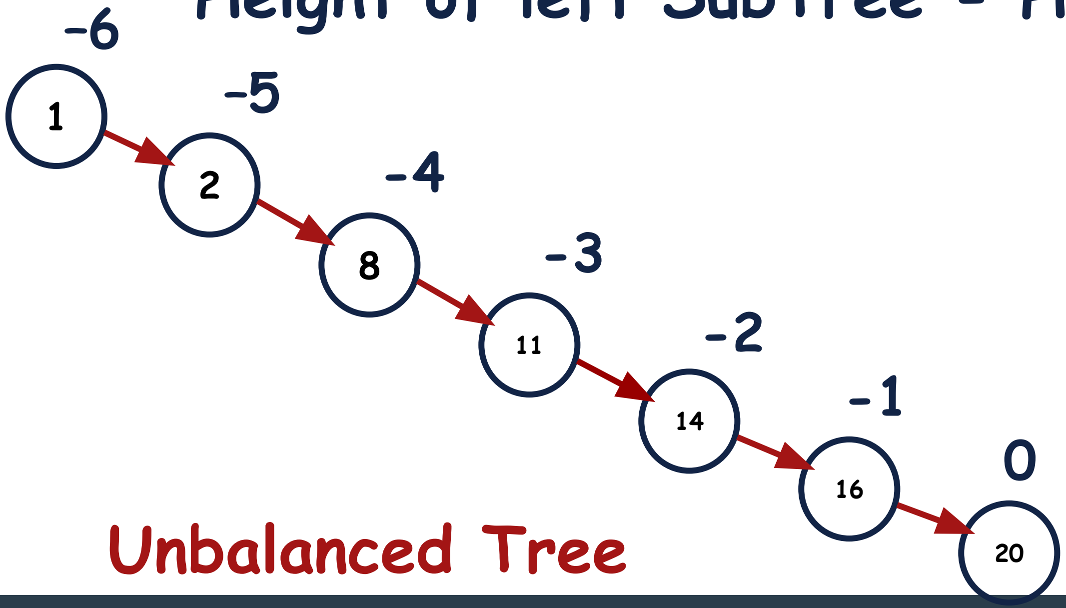
Balanced Tree

Balanced BST: Balance Factor

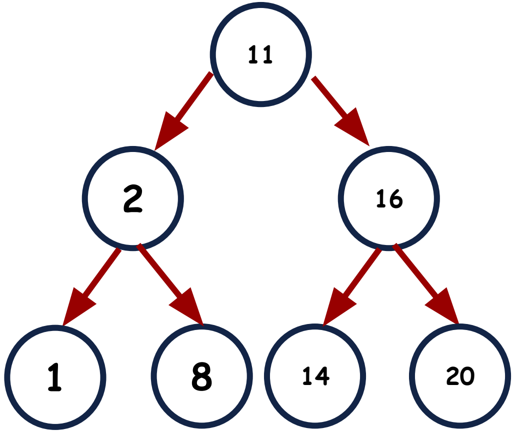
Balance Factor (BF)

=

Height of left SubTree - Height of right SubTree



Unbalanced Tree



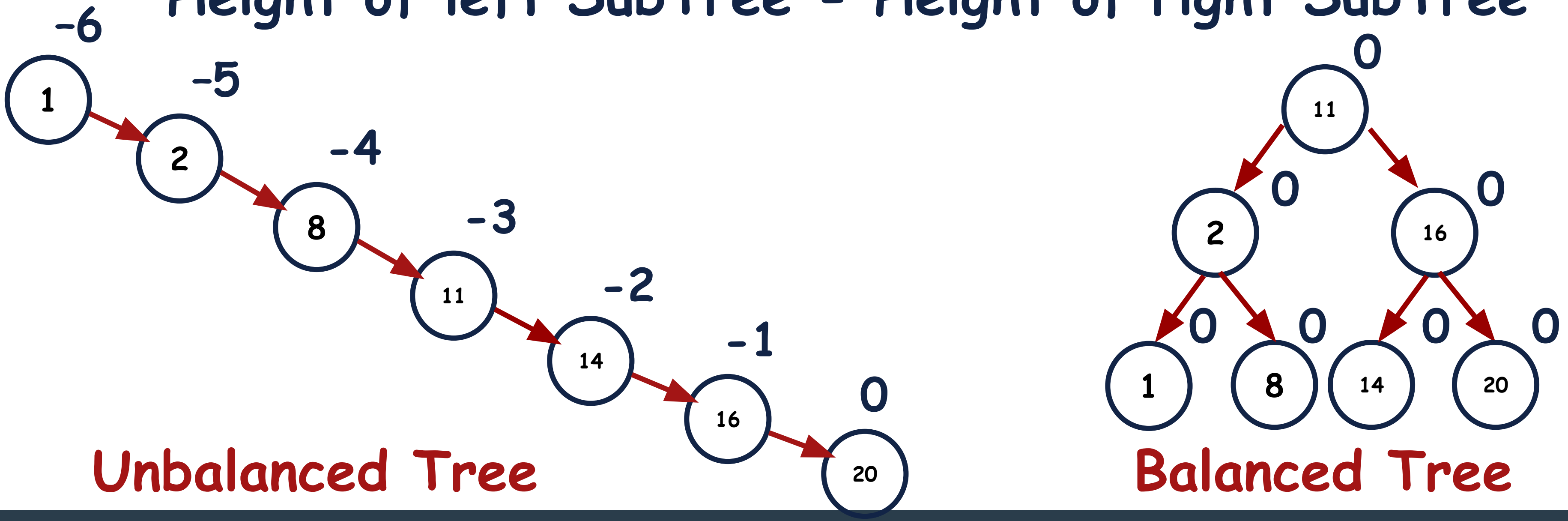
Balanced Tree

Balanced BST: Balance Factor

Balance Factor (BF)

=

Height of left SubTree - Height of right SubTree



Unbalanced Tree

Balanced Tree

Balanced BST

Now, the question is How to Create a Balanced Binary Search Tree?

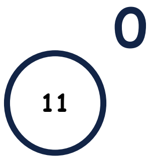
Balanced BST

Let's start inserting the input values in the Tree and see what happens when the balance factor is greater than 1 or less than -1.

Balanced BST

Input: 11

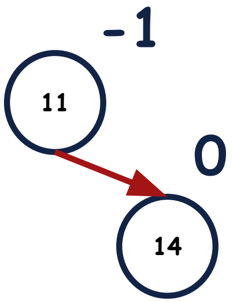
Inserted the Node and calculated the Balance Factor of the node.



Balanced BST

Input: 11, 14

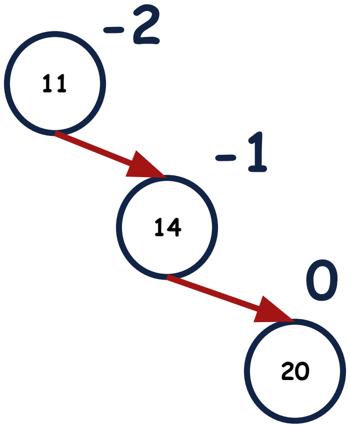
Inserted the Node and calculated the Balance Factor of the node.



Balanced BST

Input: 11, 14, 20

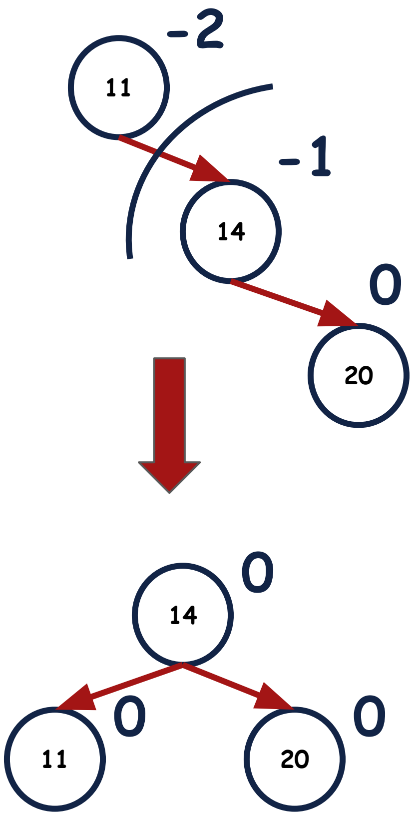
Inserted the Node and calculated the Balance Factor of the node.



Balanced BST

Input: 11, 14, 20

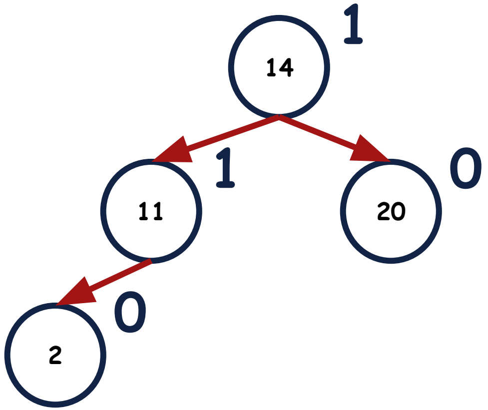
If the balance factor is less than -1 and if the node is inserted in the right subtree of the right child then do the **Left Rotation.**



Balanced BST

Input: 11, 14, 20, 2

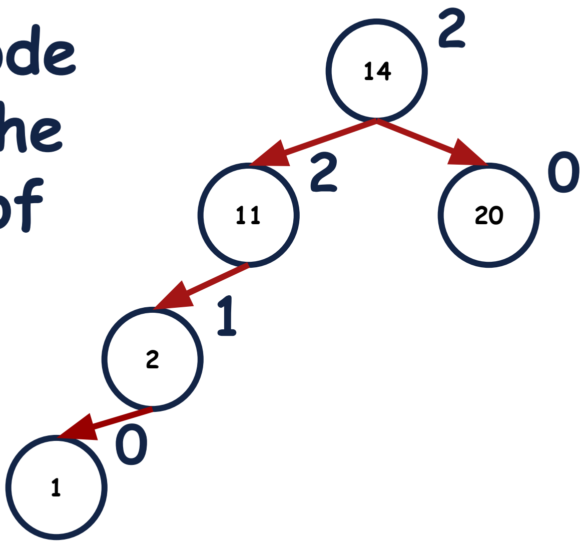
Inserted the Node and calculated the Balance Factor of the node.



Balanced BST

Input: 11, 14, 20, 2, 1

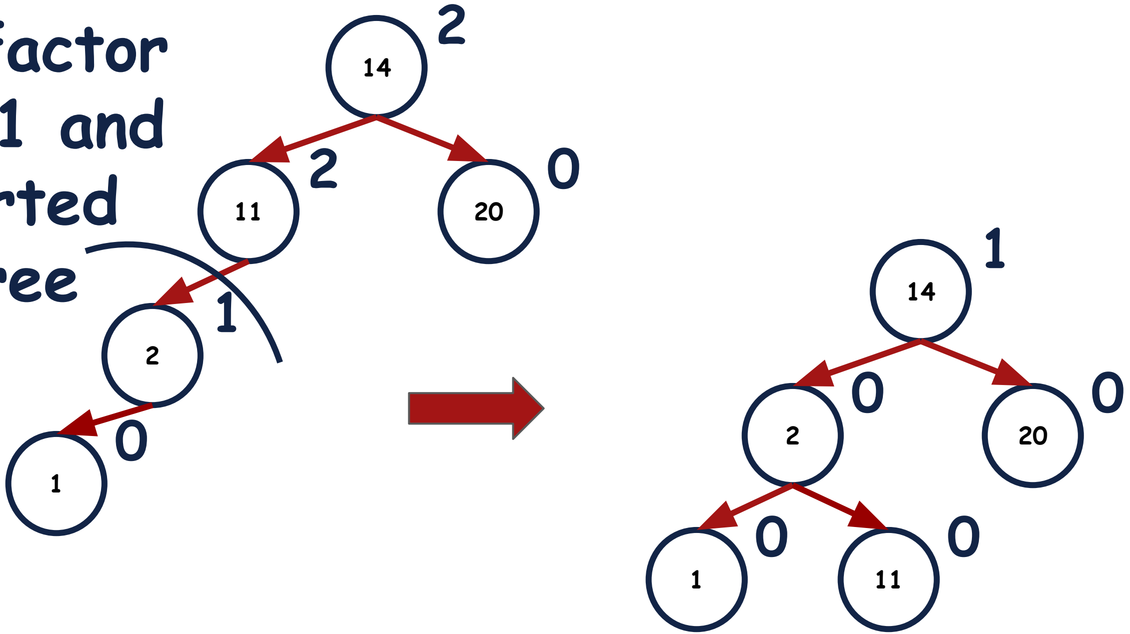
Inserted the Node and calculated the Balance Factor of the node.



Balanced BST

Input: 11, 14, 20, 2, 1

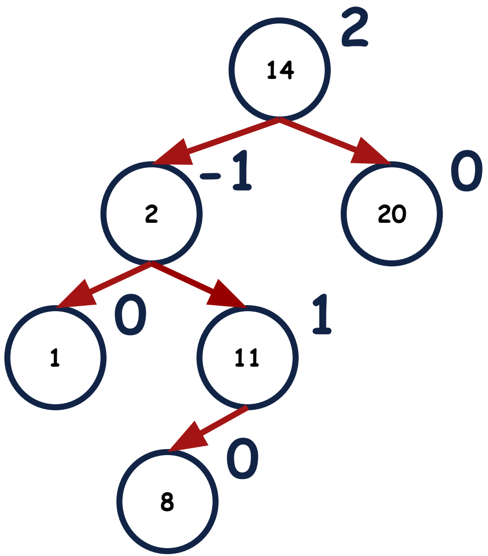
If the balance factor is greater than 1 and the node is inserted in the left subtree of the left child then apply **Right Rotation**



Balanced BST

Input: 11, 14, 20, 2, 1, 8

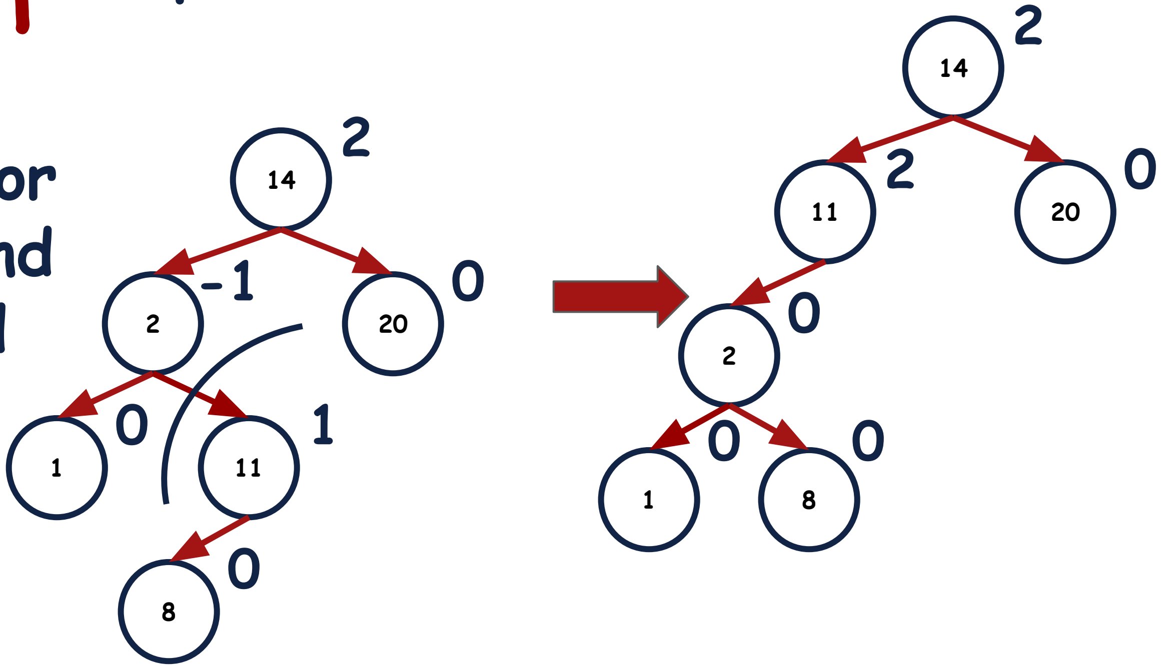
Inserted the Node and calculated the Balance Factor of the nodes.



Balanced BST

Input: 11, 14, 20, 2, 1, 8

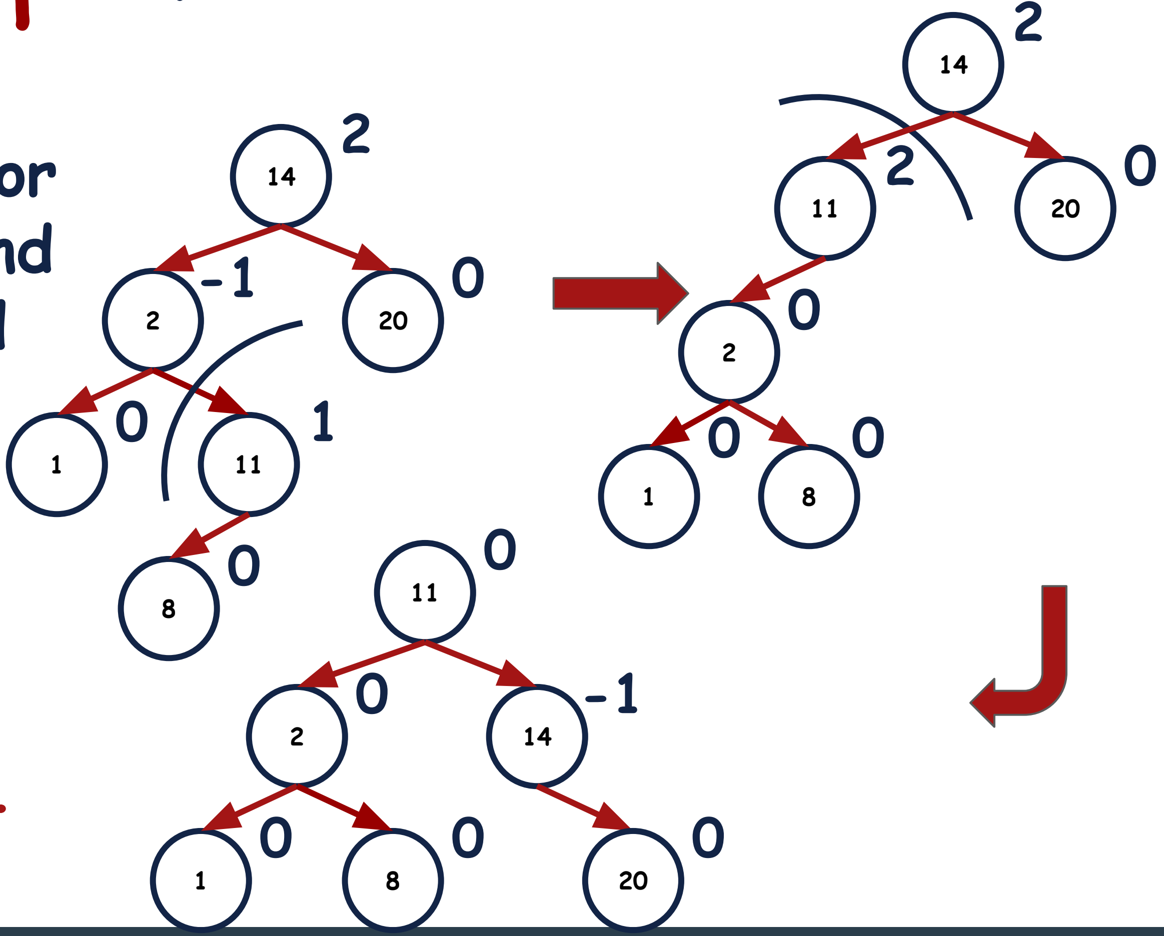
If the balance factor is greater than 1 and the node is inserted in the right subtree of
left child then apply 2 rotations.
Left Rotation
then Right Rotation.



Balanced BST

Input: 11, 14, 20, 2, 1, 8

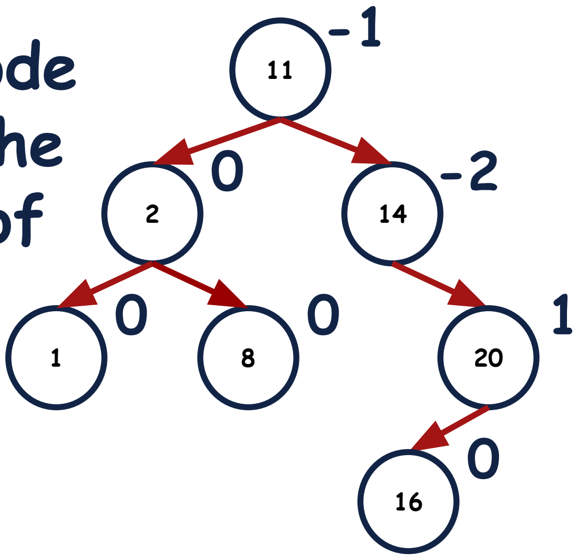
If the balance factor is greater than 1 and the node is inserted in the right subtree of left child then apply 2 rotations. Left Rotation then Right Rotation.



Balanced BST

Input: 11, 14, 20, 2, 1, 8, 16

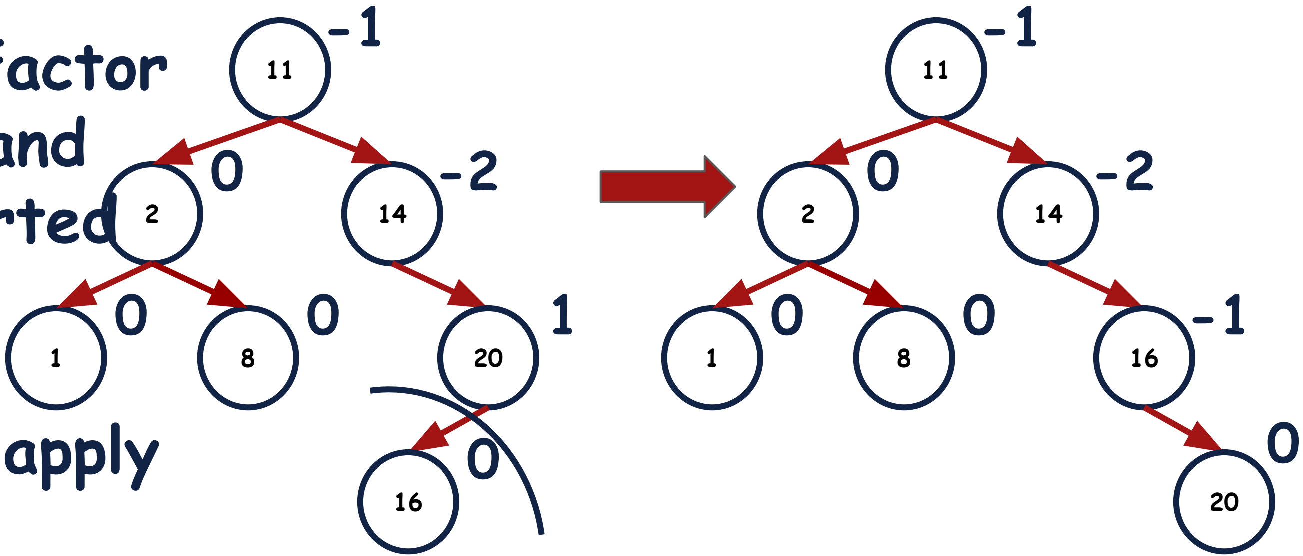
Inserted the Node and calculated the Balance Factor of the nodes.



Balanced BST

Input: 11, 14, 20, 2, 1, 8, 16

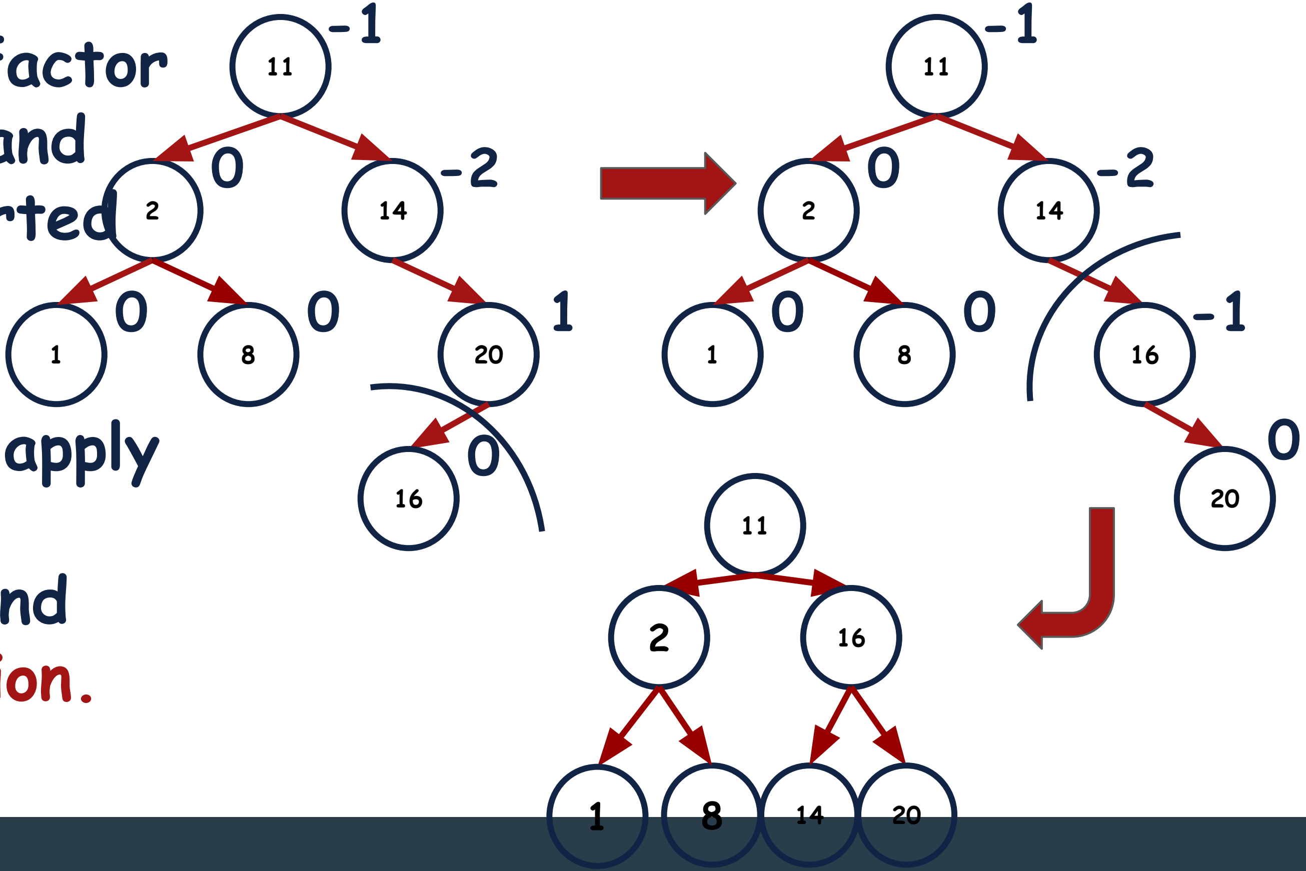
If the balance factor is less than -1 and the node is inserted in the left subtree of right child then apply 2 rotations.
Right Rotation and then Left Rotation.



Balanced BST

Input: 11, 14, 20, 2, 1, 8, 16

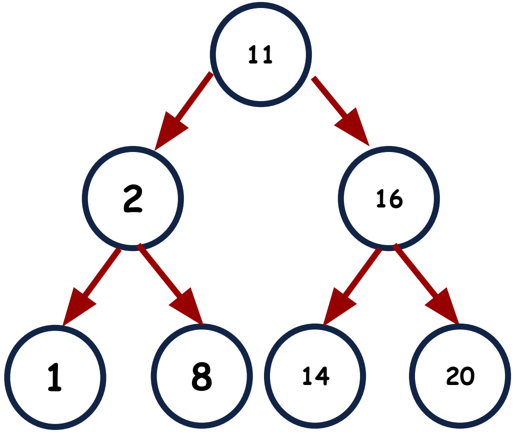
If the balance factor is less than -1 and the node is inserted in the left subtree of right child then apply 2 rotations.
Right Rotation and then Left Rotation.



Self Balancing BST

Input: 11, 14, 20, 2, 1, 8, 16

Self-Balancing Binary Search Trees are **height-balanced** binary search trees that automatically keeps height as small as possible when **insertion** and **deletion** operations are performed on the tree

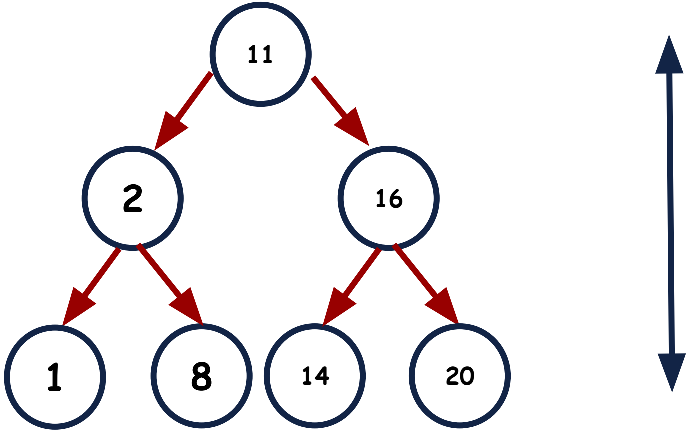


$h = \log_2(n)$

AVL Trees

Input: 11, 14, 20, 2, 1, 8, 16

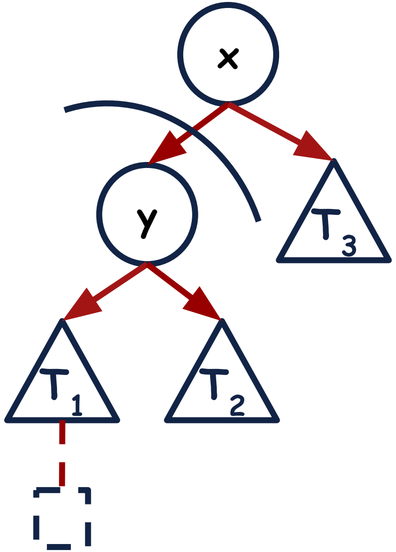
Such Self Balancing BST in which the heights of the two child subtrees of any node differ by at most one are known as AVL trees (named after inventors Adelson-Velsky and Landis)



$h = \log_2(n)$

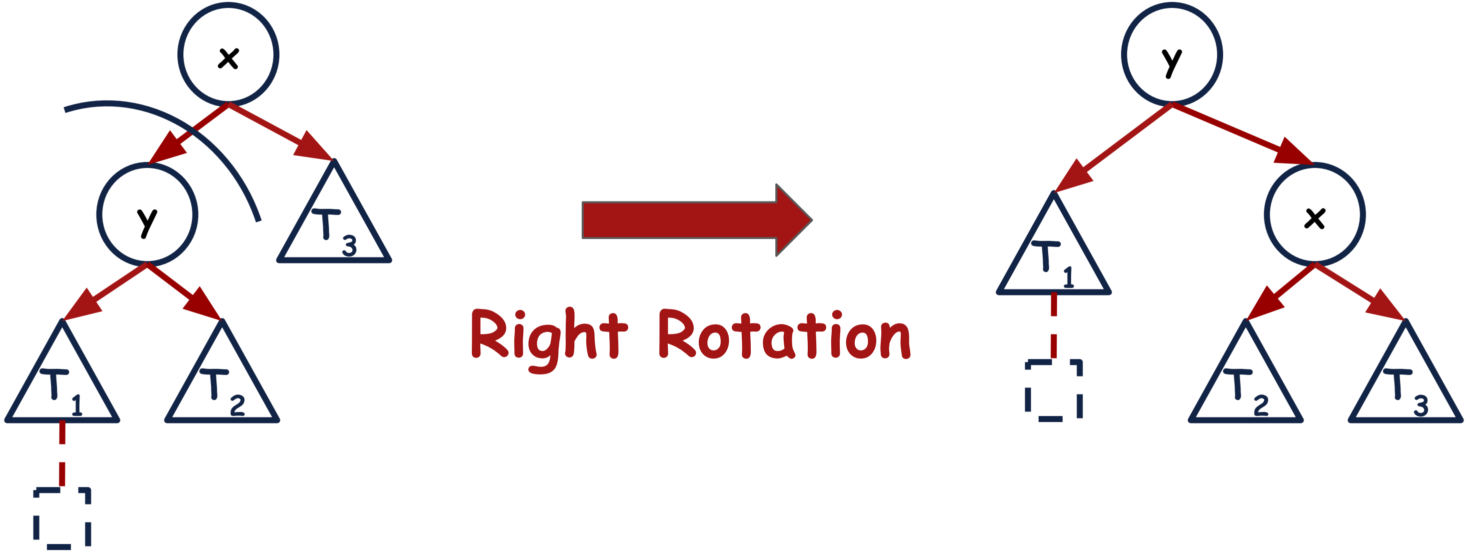
AVL Trees: 4 Types of Rotations

Case 1 (LL Case): Insertion into **left** subtree of **left** child of node x



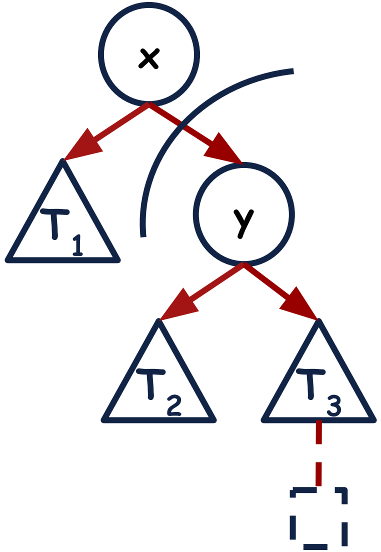
AVL Trees: 4 Types of Rotations

Case 1 (LL Case): Insertion into left subtree of left child of node x



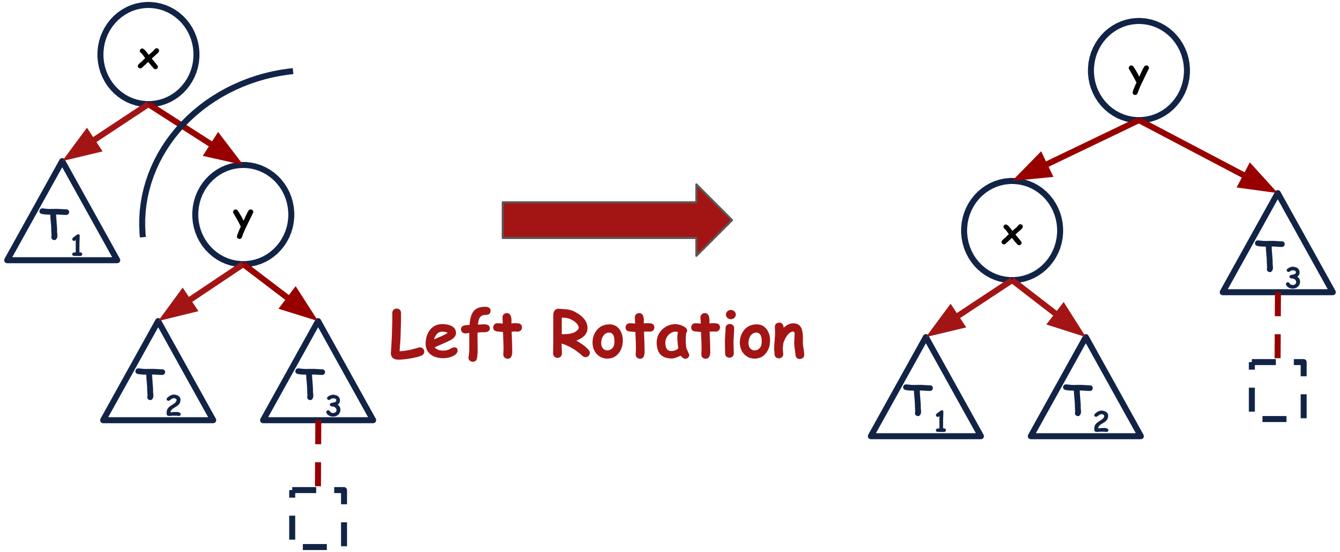
AVL Trees: 4 Types of Rotations

Case 2 (RR Case): Insertion into right subtree of right child of node x



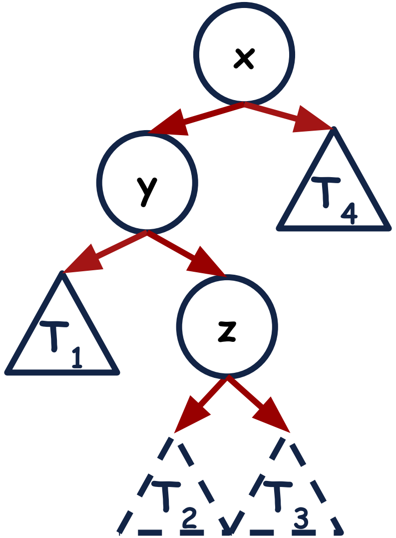
AVL Trees: 4 Types of Rotations

Case 2 (RR Case): Insertion into right subtree of right child of node x



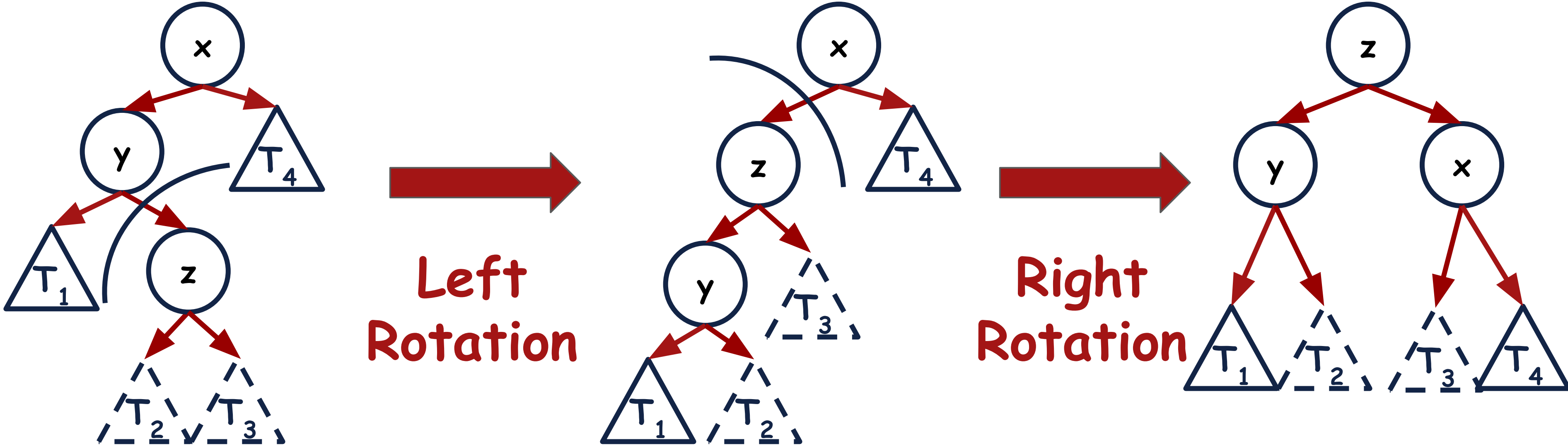
AVL Trees: 4 Types of Rotations

Case 3 (LR Case): Insertion into right subtree of left child of node x



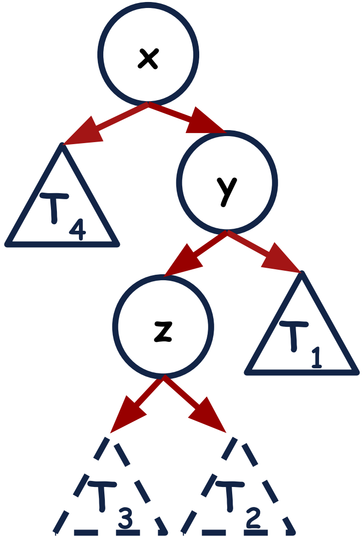
AVL Trees: 4 Types of Rotations

Case 3 (LR Case): Insertion into right subtree of left child of node x



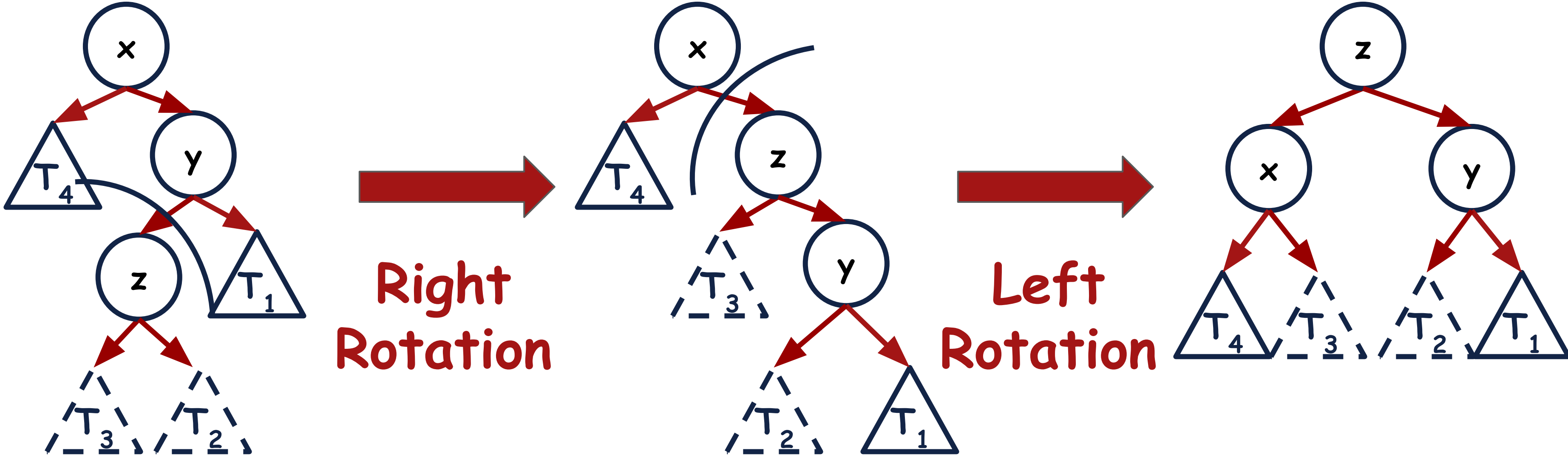
AVL Trees: 4 Types of Rotations

Case 4 (RL Case): Insertion into left subtree of right child of node x



AVL Trees: 4 Types of Rotations

Case 4 (RL Case): Insertion into left subtree of right child of node x



AVL Trees: Working Example

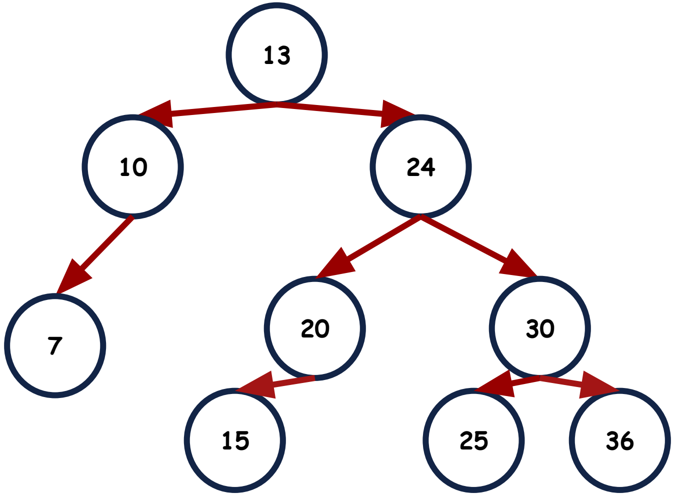
Build an AVL tree with the following values.

Input: 15, 20, 24, 10, 13, 7, 30, 36, 25

AVL Trees: Working Example

Build an AVL tree with the following values.

Input: 15, 20, 24, 10, 13, 7, 30, 36, 25

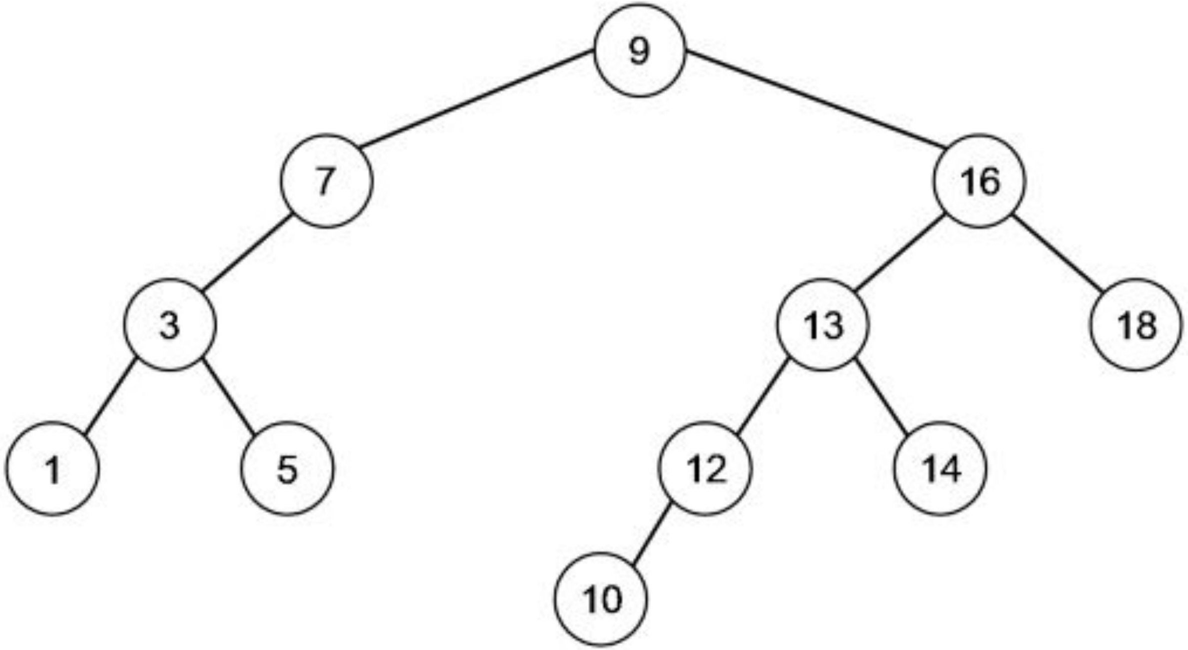


Learning Objective

Students should be able to insert elements in the AVL trees.

Self Assessment

In the following tree, write the Balance Factor of each node.



Self Assessment

Draw all the rotations that you must perform and the final AVL tree after the following elements are inserted in the given order starting from an empty tree.

Input:

1, 10, 5, 7, 3, 13, 6, 4, 8, 9