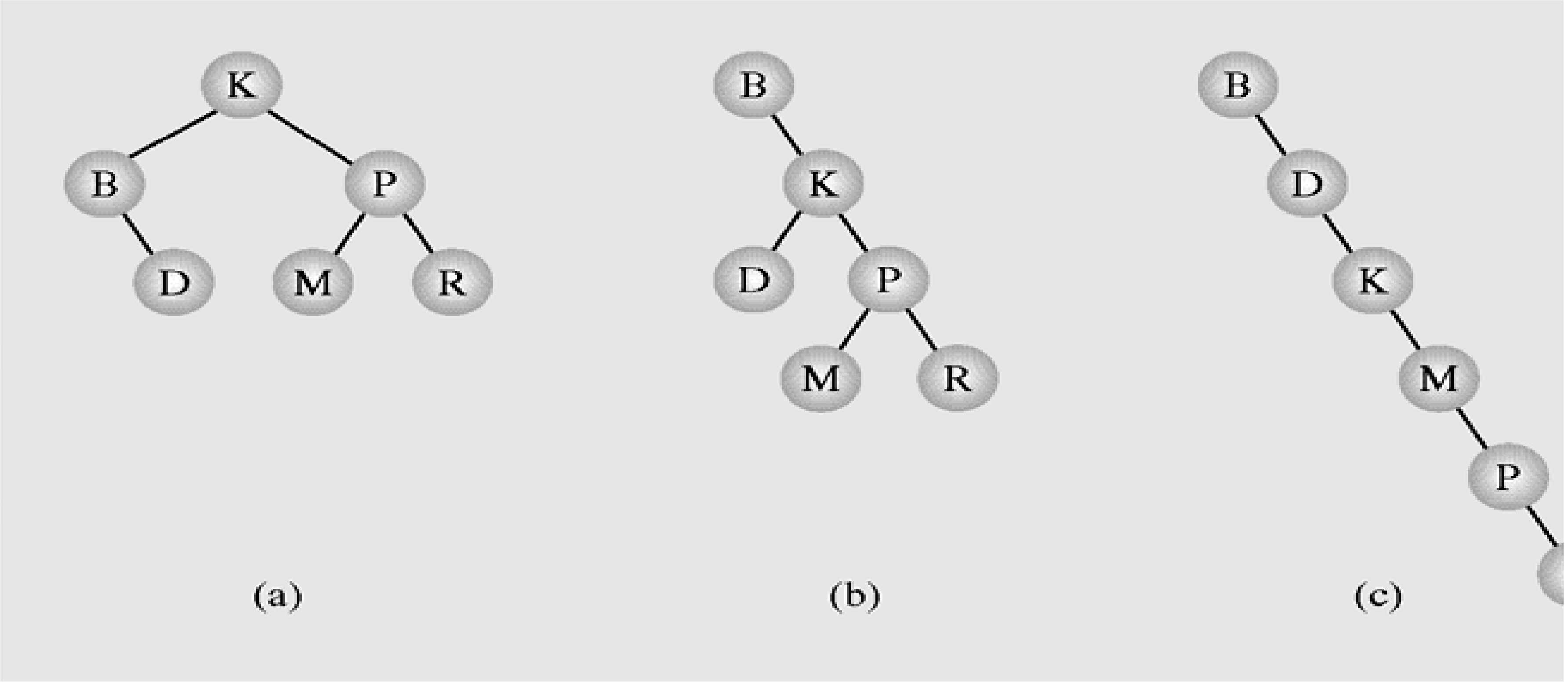
Introduction to AVL Trees

Time Complexity of Basic BST Operations:

- Search, Insert, Delete
  - These operations visit the nodes along a root-to-leaf path
  - The number of nodes encountered on unique path depends on *the shape of the* tree *and the position of the node in the tree*

Different Shapes of Tree

RE 6.34 Different binary search trees with the same information.

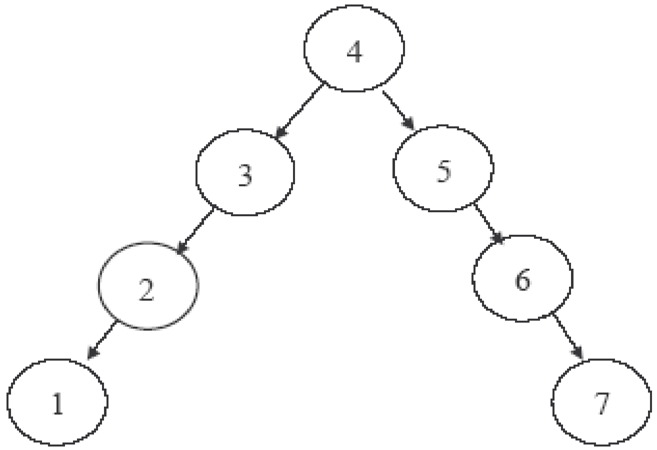


Balanced BST Can Do better

- Construct a BST for given keys:
  - 30, 40, 10, 50, 20, 5, 35.
  - 50, 40, 35, 30, 20, 10, 5.
- BSTs are limited because of their bad worst-case performance O(n). A BST with this worst-case structure is no more efficient than a regular linked list
- Balanced search trees are trees whose heights in the worst case is O(lg n)

What Does it Mean to Balance a BST?

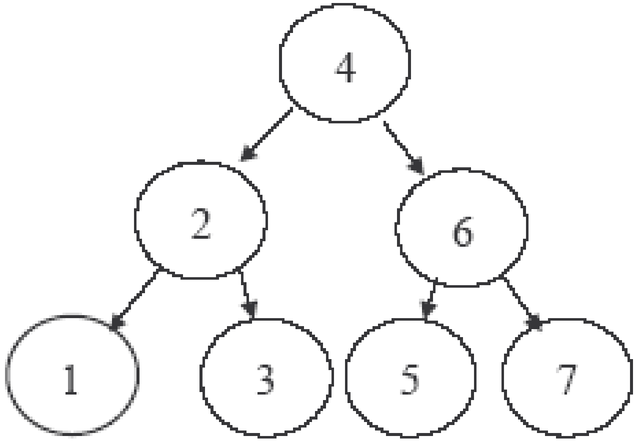
- Tentative Rule
  - Require that the left and right subtrees of the root node have the same height



We can do better

What Does it Mean to Balance a BST? (cont'd)

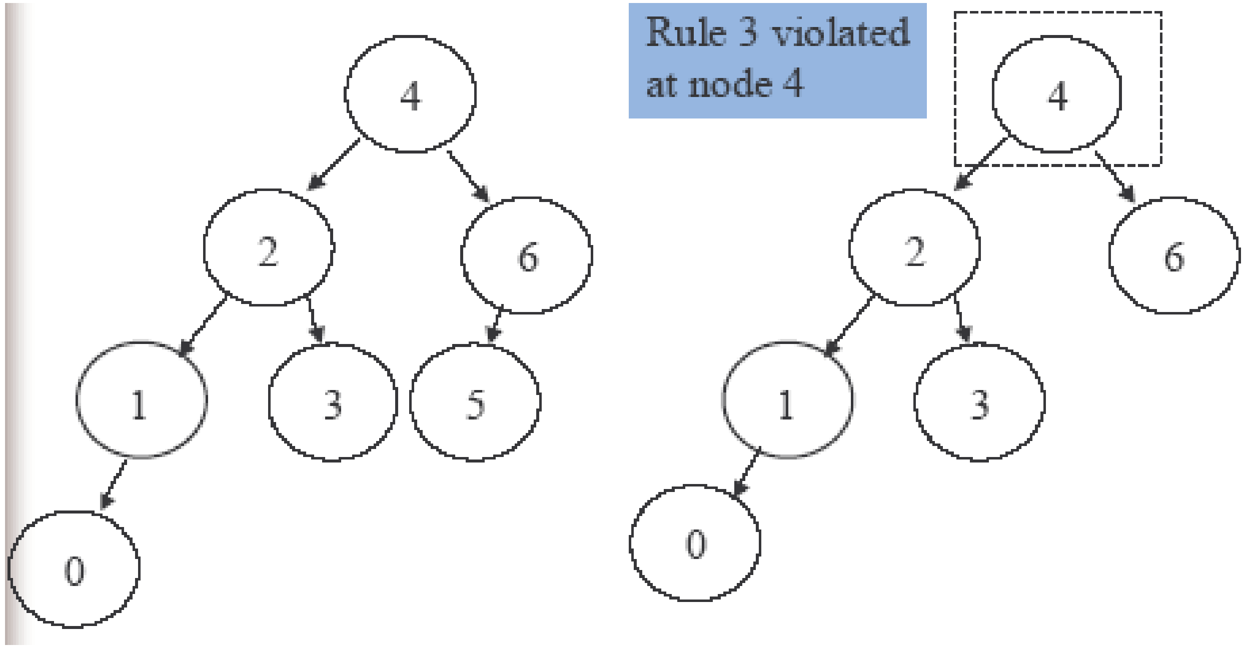
- Another Tentative Rule
  - Require that every node have left and right subtrees of the same height



Too restrictive → only perfectly balanced trees of $2^k - 1$ nodes would satisfy this criterion

What Does it Mean to Balance BST? (cont'd)

- The Rule
  - Require that, for every node, the height of the left and right subtrees can differ by at most one



Balancing a BST

- There are a number of techniques to properly balance a binary tree
  - Approach 1: take all the elements, place them in an array, sort them, and then reconstruct the tree (global) (example, and algorithm)
  - Approach 2: constantly restructuring the tree when new elements arrive or elements are deleted and lead to an unbalanced tree (i.e., self-balancing trees)

Approach 1: Reorder Data and Build BST

- When all data arrive, store all data in an array, sort the array. What is the root of the tree?
  - the middle element of the array
- Designate for the root of the BST the middle element of the array (i.e., the middle element of the array is the first element inserted into the BST)
- Continue inserting recursively on the left and right subarrays until all elements in the array have been inserted into the BST
- 1 2 3 4 5 6 7 (construct the tree)

Approach 1: Reorder Data and Build BST (cont'd)

- This approach has one serious drawback
  - All data must be put in an array before the BST can be created
  - Unsuitable or very inefficient when the BST has to be used while the data to be included in the BST are still coming.

Dictionary Implementations

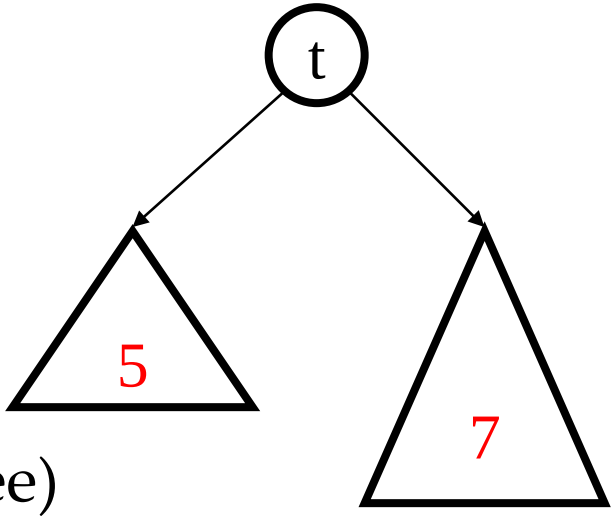
	unsorted array	sorted Array	linked list	BST
insert	find + O(n)	O(n)	find + O(1)	O(Depth)
find	O(n)	O(log n)	O(n)	O(Depth)
delete	find + O(1)	O(n)	find + O(1)	O(Depth)

BST's looking good for shallow trees, *i.e.* the depth D is small (log n), otherwise as bad as a linked list!

Balanced binary tree

- The disadvantage of a binary search tree is that its height can be as large as N-1
- This means that the time needed to perform insertion and deletion and many other operations can be O(N) in the worst case
- We want a tree with small height
- A binary tree with N node has height at least $\Theta(\log N)$
- Thus, our goal is to keep the height of a binary search tree O(log N)
- Such trees are called balanced binary search trees. Examples are AVL tree, red-black tree.

Balance



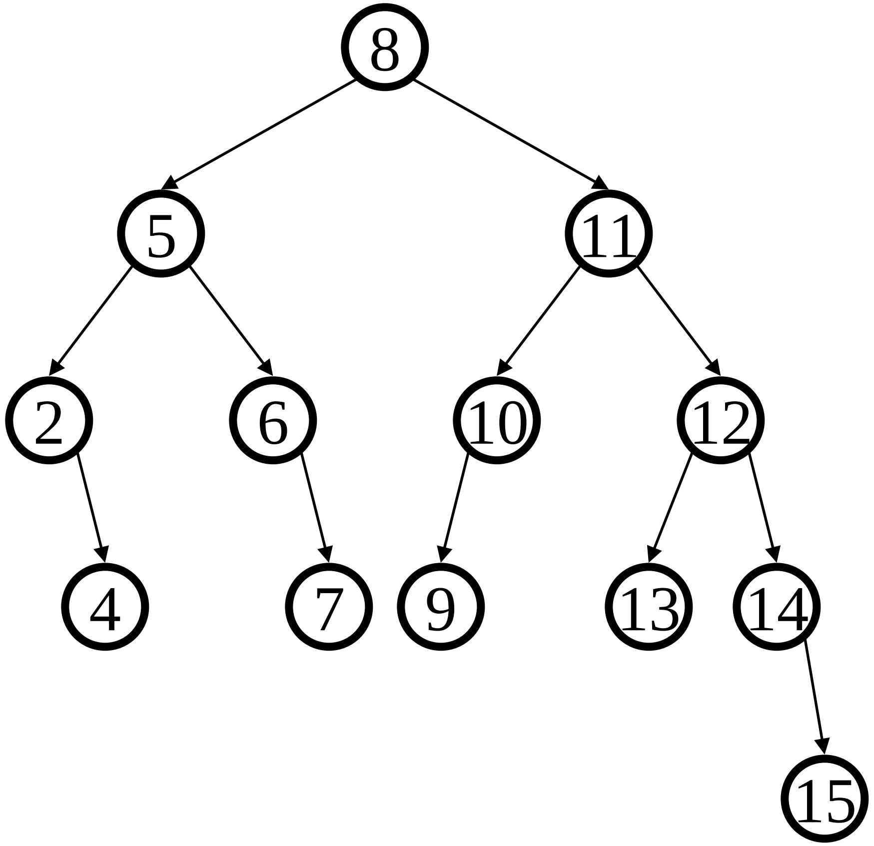
- Balance
  - height(left subtree) - height(right subtree)
  - zero everywhere $\Rightarrow$ perfectly balanced
  - small everywhere $\Rightarrow$ balanced enough

Balance between -1 and 1 everywhere $\Rightarrow$
maximum height of 1.44 log n

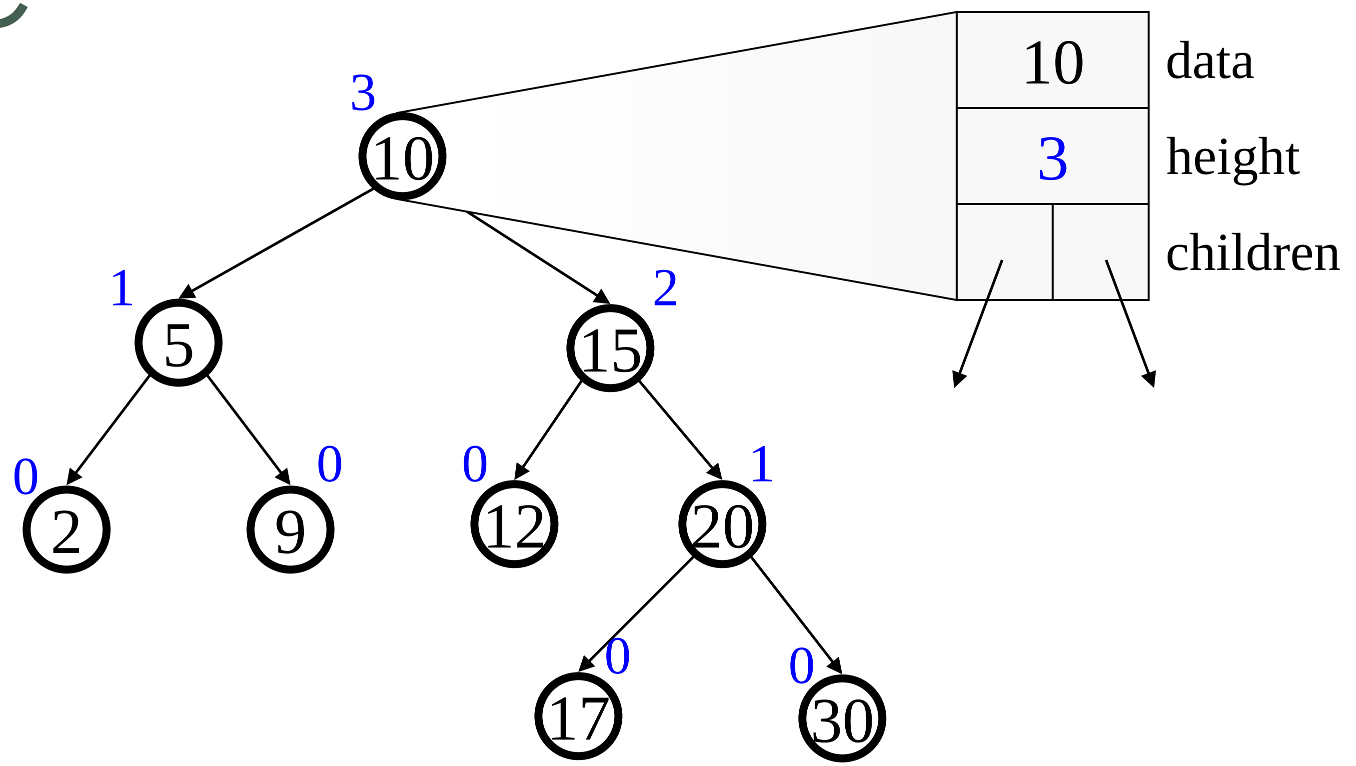
AVL Tree :

- Named after **Adelson-Velskii and Landis.**
- Binary search tree properties
  - binary tree property
  - search tree property
- Balance property
  - balance of every node is:

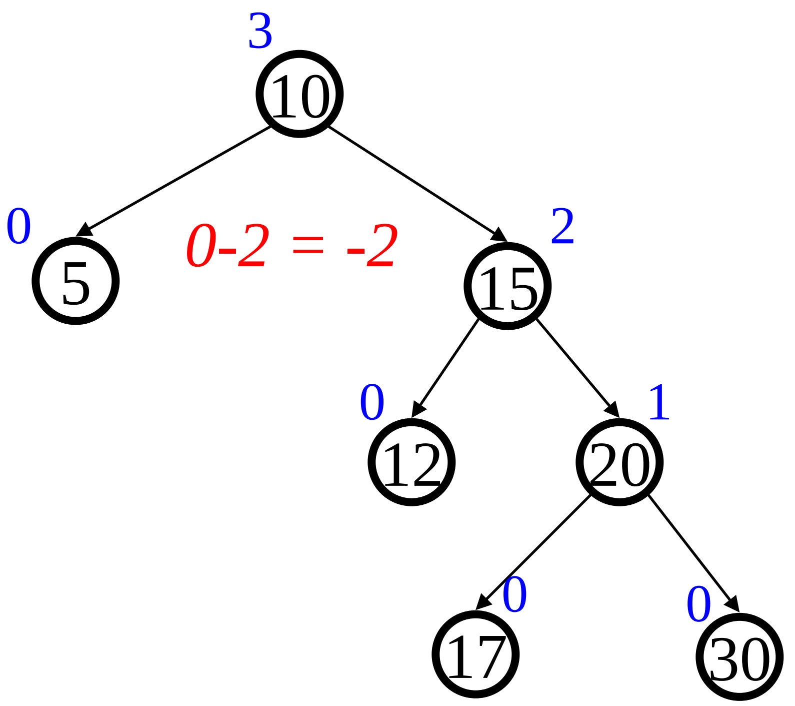
    $-1 \le b \le 1$
  - result:
    - depth is $\Theta(\log n)$

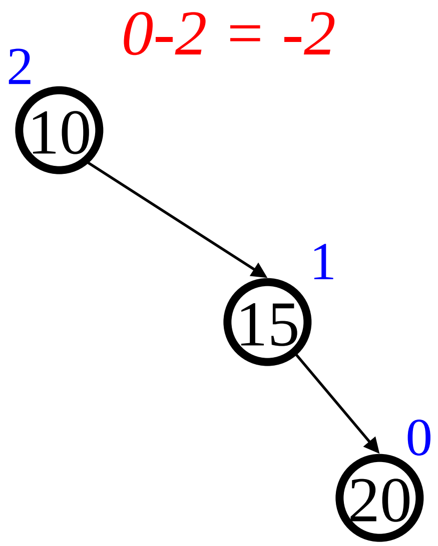


An AVL Tree



Not AVL Trees





AVL Trees

Let us call the node that must be rebalanced α. Since any node has at most two children, and a height imbalance requires that α's two subtrees' heights differ by two, it is easy to see that a violation might occur in four cases:

1. An insertion into the left subtree of the left child of α
2. An insertion into the right subtree of the left child of α
3. An insertion into the left subtree of the right child of α
4. An insertion into the right subtree of the right child of α

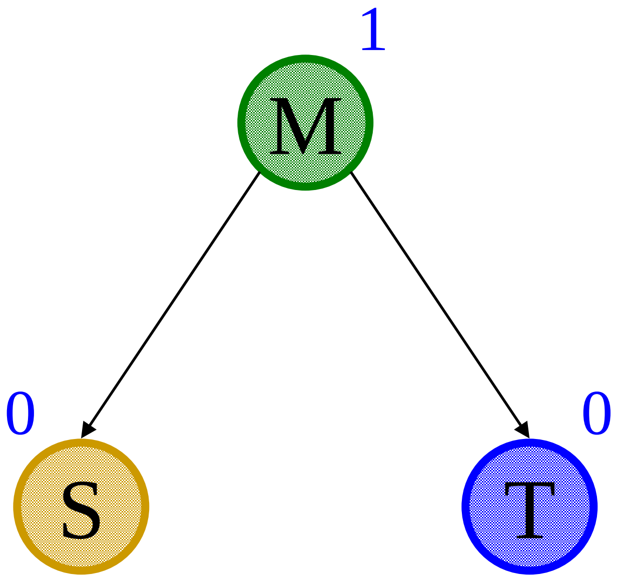
Staying Balanced

Good case: inserting small, tall and middle.

Insert(middle)

Insert(small)

Insert(tall)

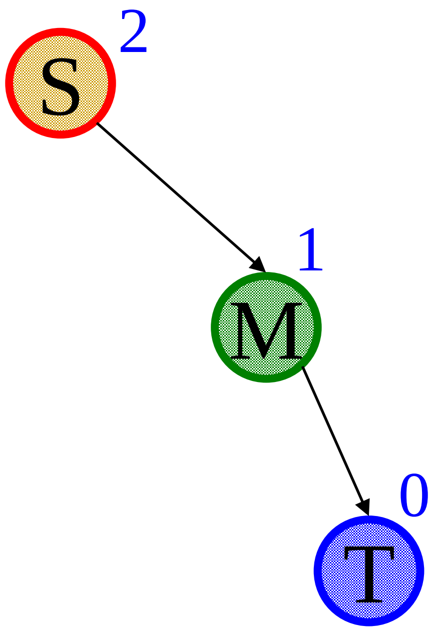


Bad Case #1

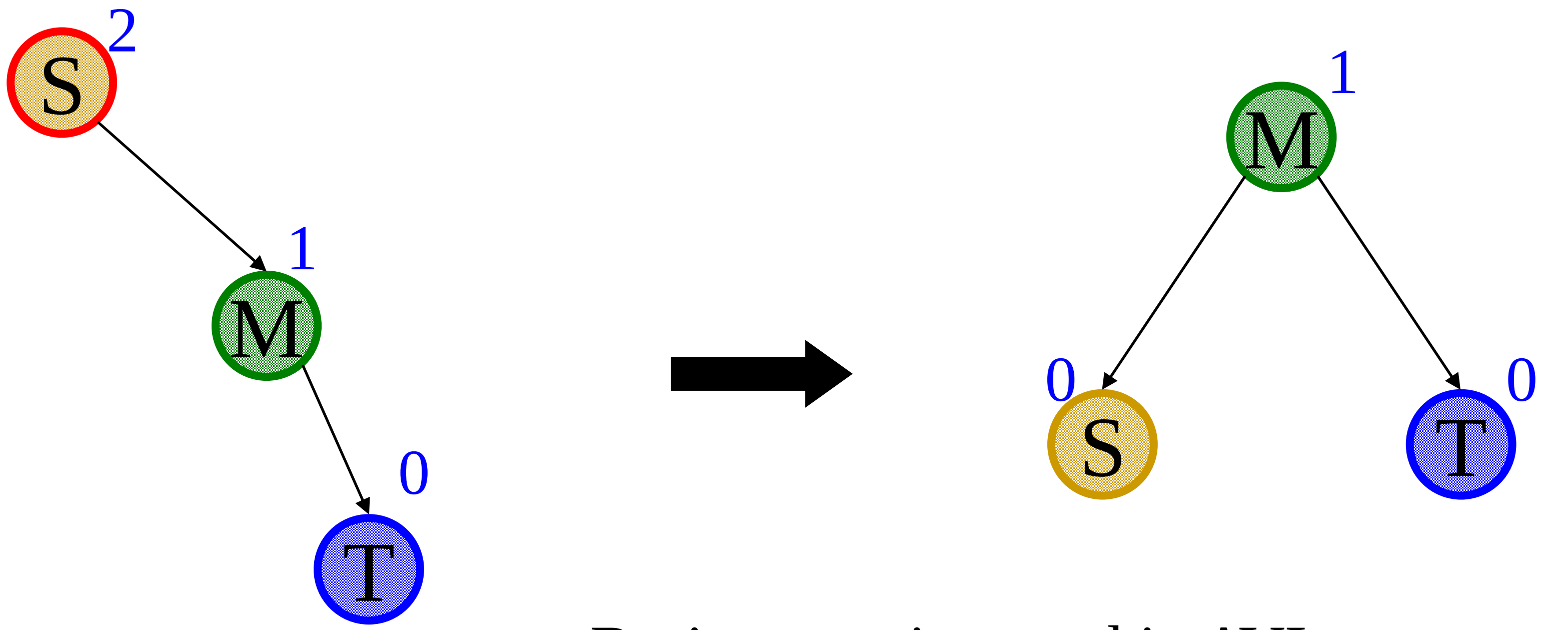
Insert(small)

Insert(middle)

Insert(tall)



Single Rotation



Basic operation used in AVL trees:

A right child could legally have its parent as its left child.

Rotations in AVL:

Rebalancing rotation are classified as LL, LR, RR and RL

LL Rotation: Inserted node is in the left sub-tree of left sub-tree of node A

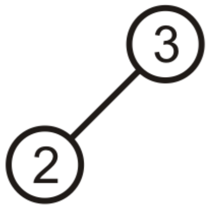
RR Rotation: Inserted node is in the right sub-tree of right sub-tree of node A

LR Rotation: Inserted node is in the right sub-tree of left sub-tree of node A

RL Rotation: Inserted node is in the left sub-tree of right sub-tree of node A

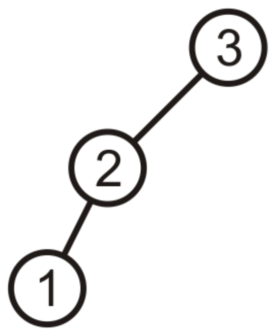
Prototypical Examples

These two examples demonstrate how we can correct for imbalances: starting with this tree, add 1:



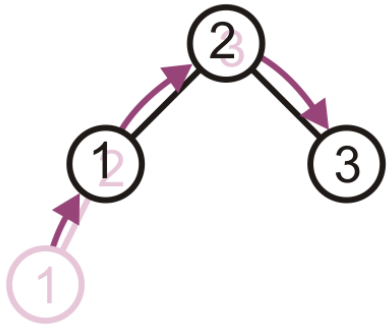
Prototypical Examples

This is more like a linked list; however, we can fix this…



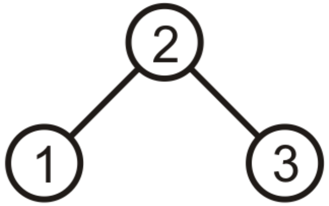
Prototypical Examples

Promote 2 to the root, demote 3 to be 2's right child, and 1 remains the left child of 2



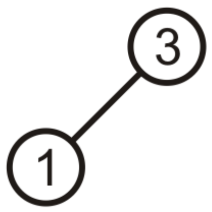
Prototypical Examples

The result is a perfect, though trivial tree



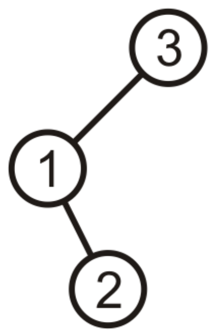
Prototypical Examples

Alternatively, given this tree, insert 2



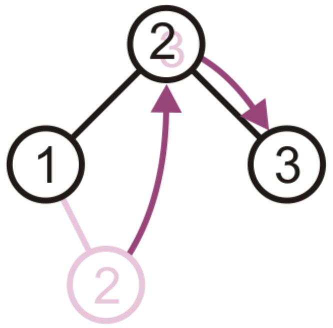
Prototypical Examples

Again, the product is a linked list; however, we can fix this, too



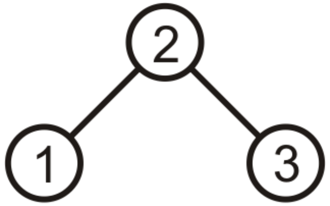
Prototypical Examples

Promote 2 to the root, and assign 1 and 3 to be its children



Prototypical Examples

The result is, again, a perfect tree



These examples may seem trivial, but they are the basis for the corrections in the next data structure we will see: AVL trees

Single Rotation

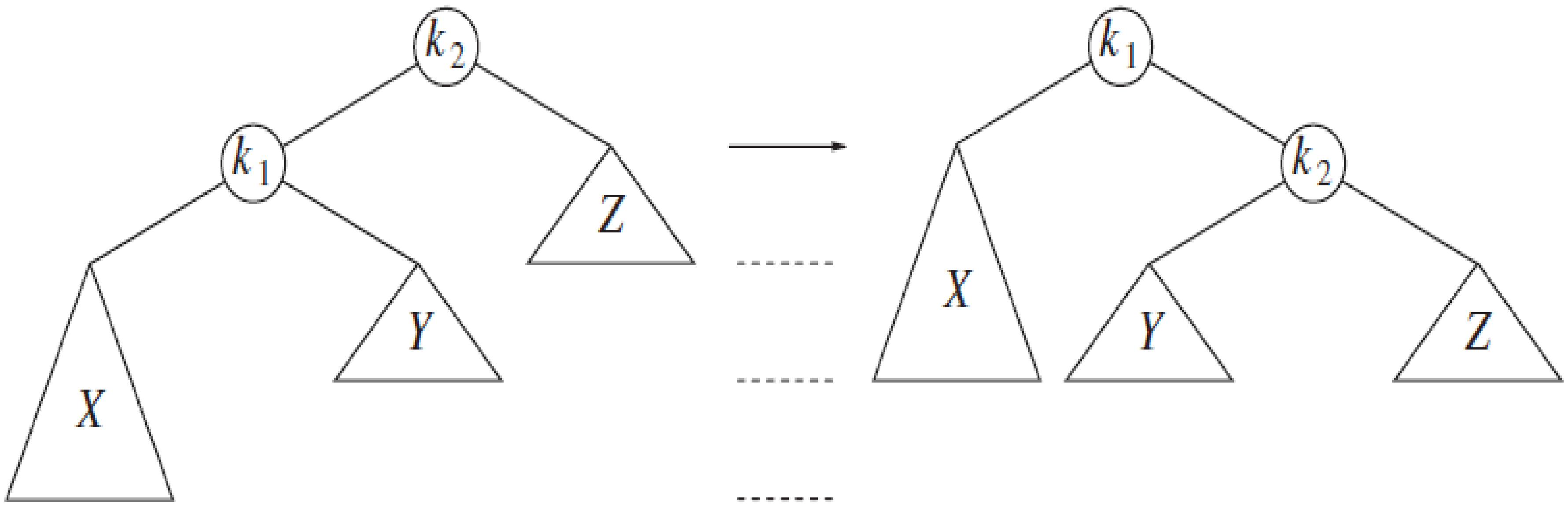


Figure 4.34 Single rotation to fix case 1

LL Rotation

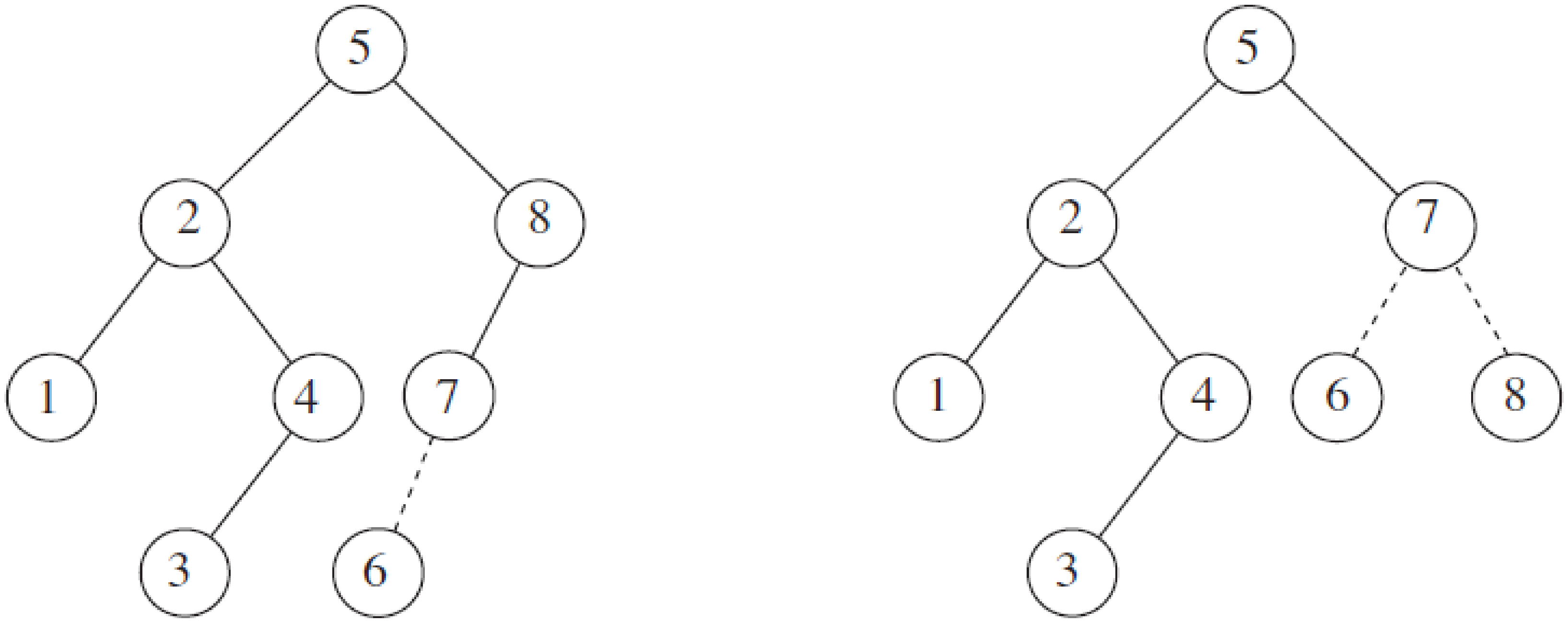


Figure 4.35 AVL property destroyed by insertion of 6, then fixed by a single rotation

RR ROTATION :

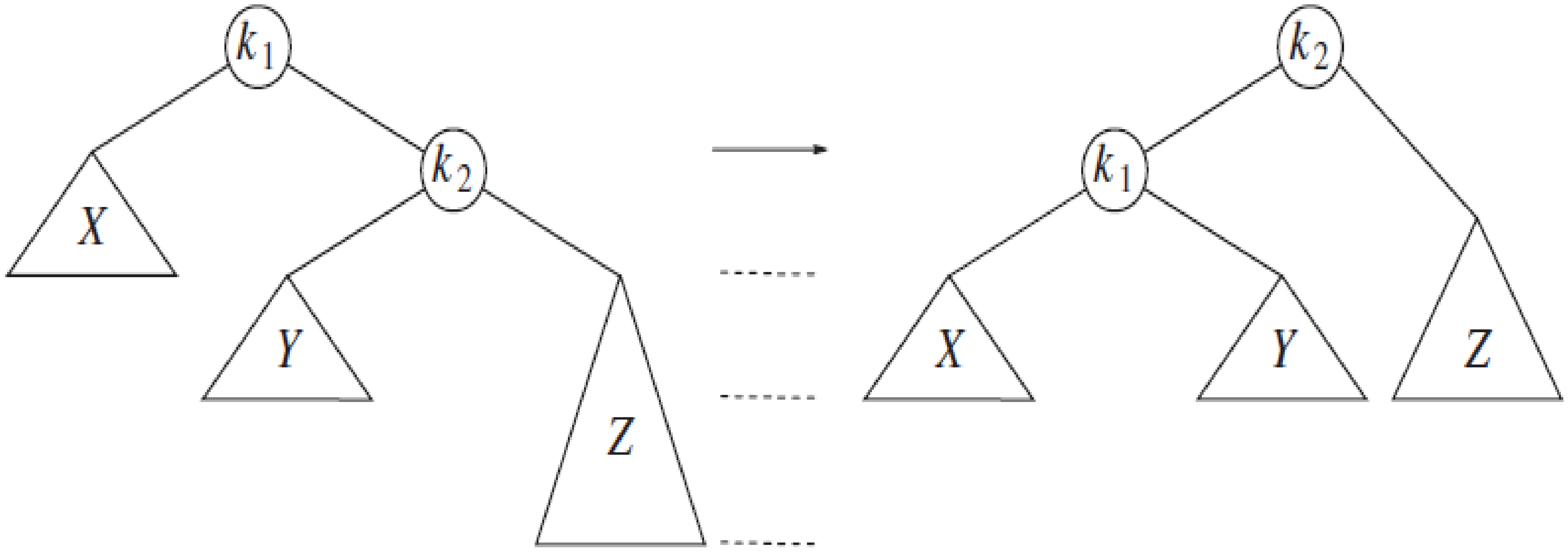
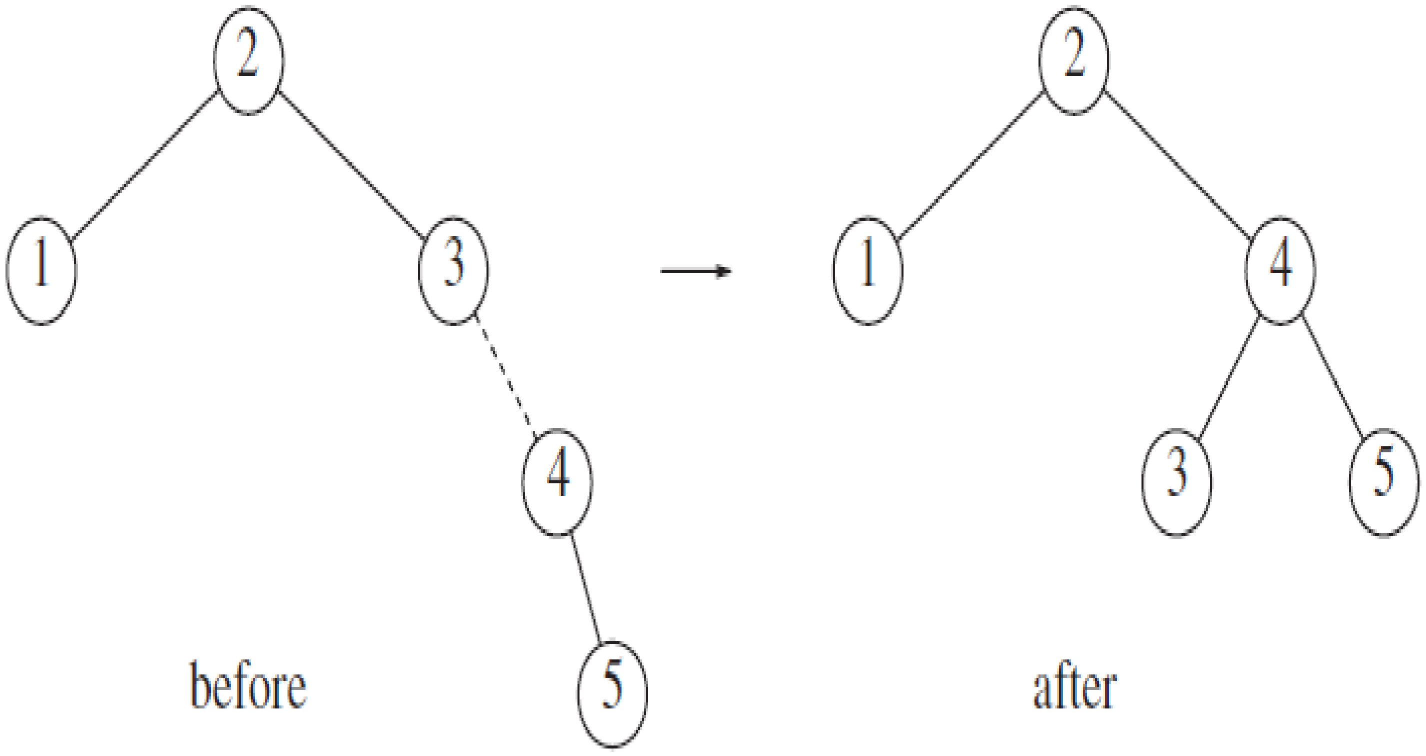


Figure 4.36 Single rotation fixes case 4

AVL Tree rotations

- 3,2,1,4,5,6,7


2
1
3
4
5
before
2
1
4
3
5
after

Double Rotation LR

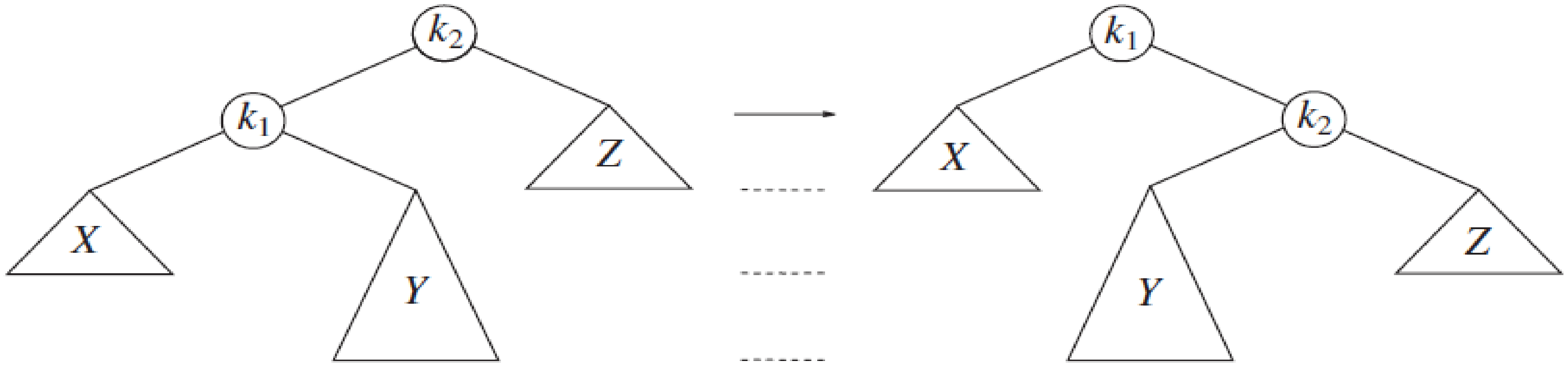


Figure 4.37 Single rotation fails to fix case 2

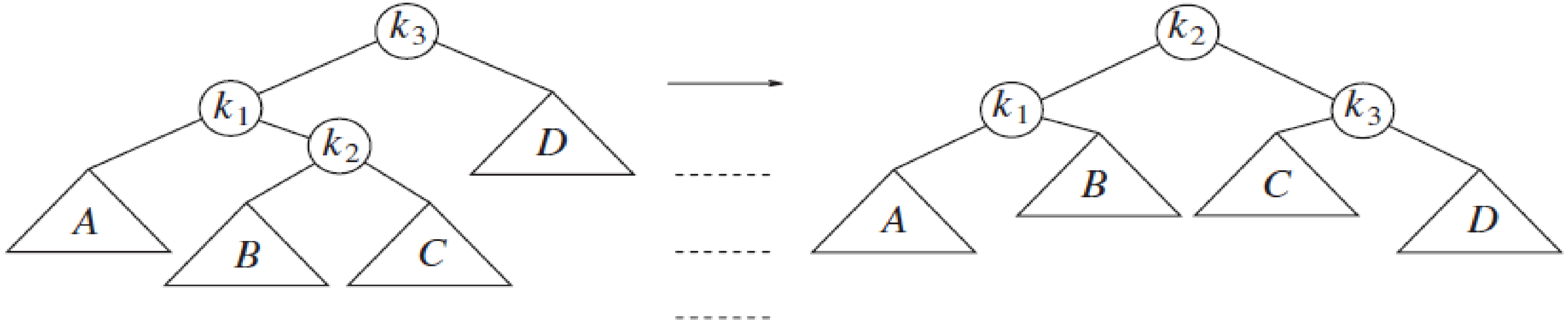


Figure 4.38 Left–right double rotation to fix case 2

Double Rotation RL

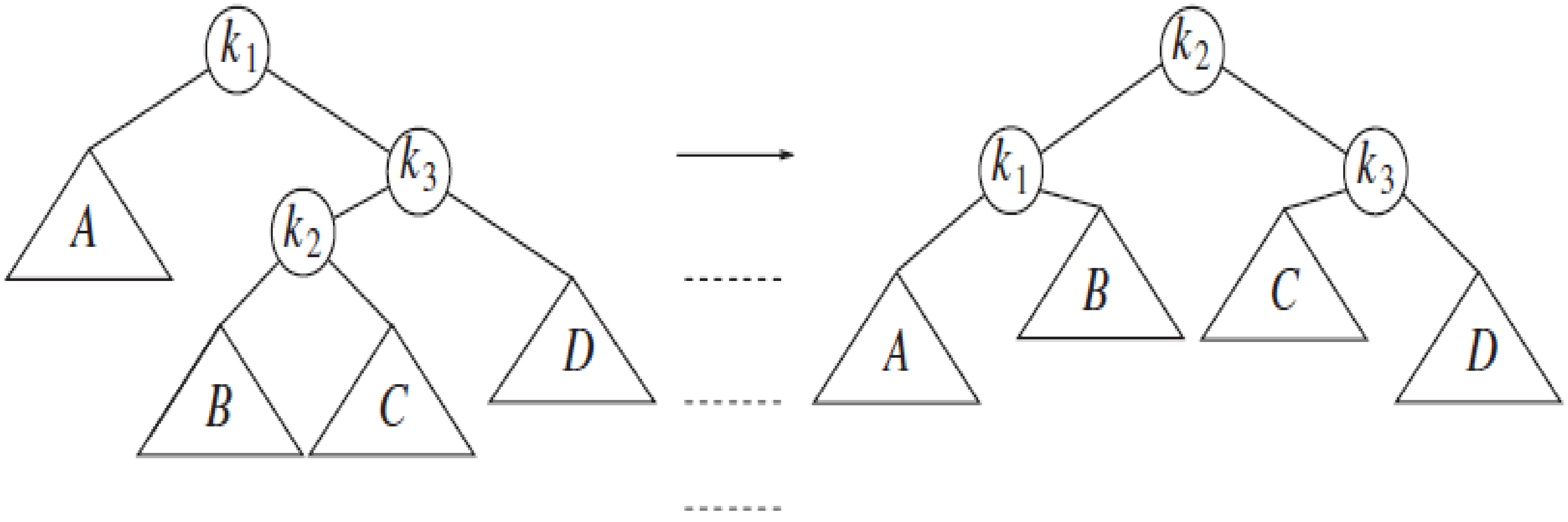
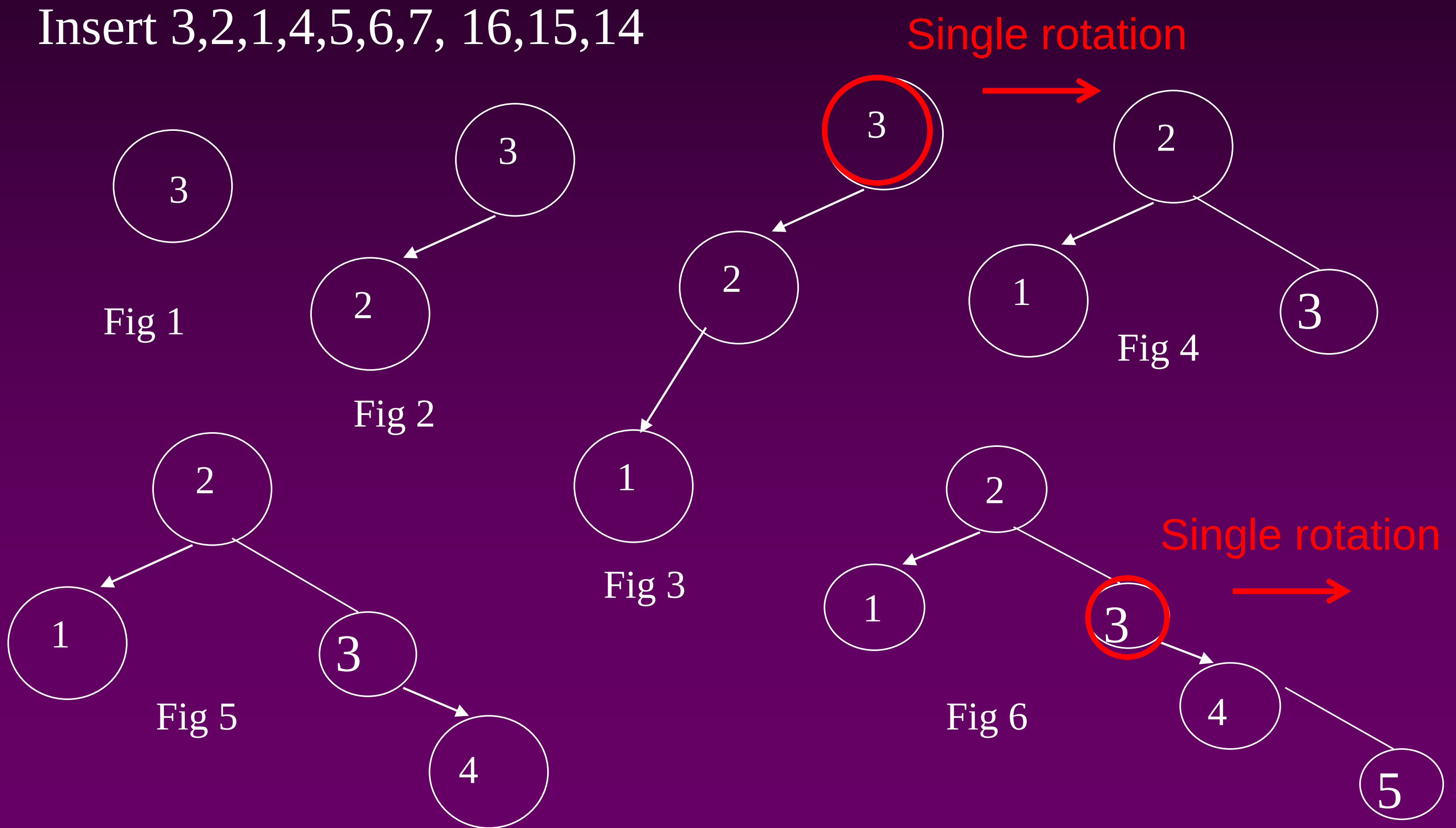


Figure 4.39 Right–left double rotation to fix case 3

An Extended Example

Insert 3,2,1,4,5,6,7, 16,15,14



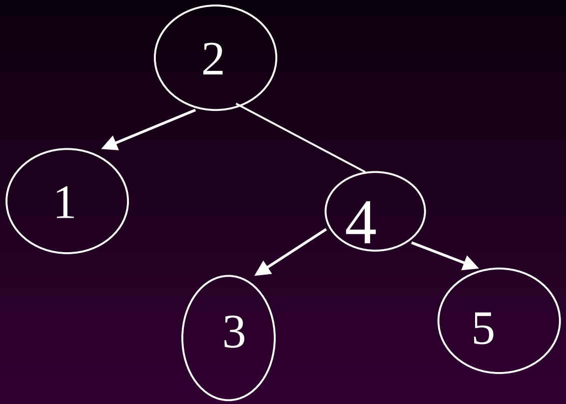


Fig 7

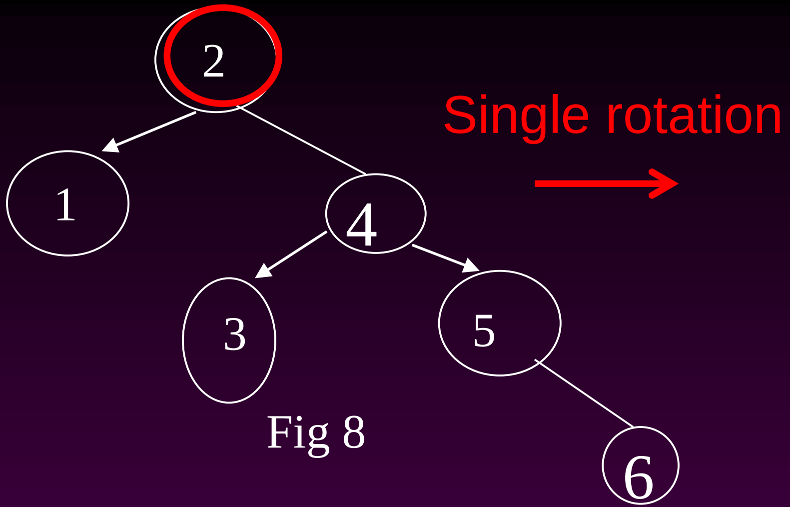


Fig 8

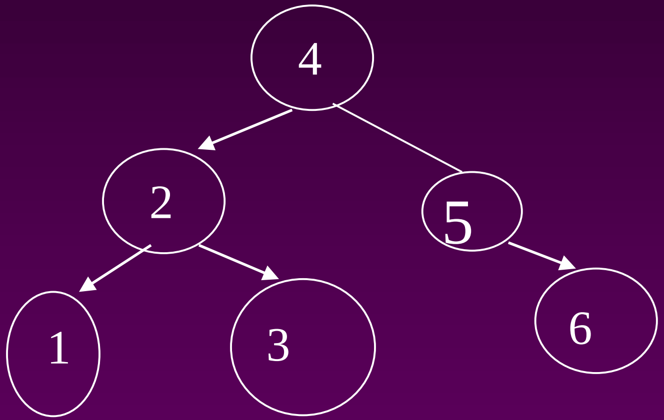


Fig 9

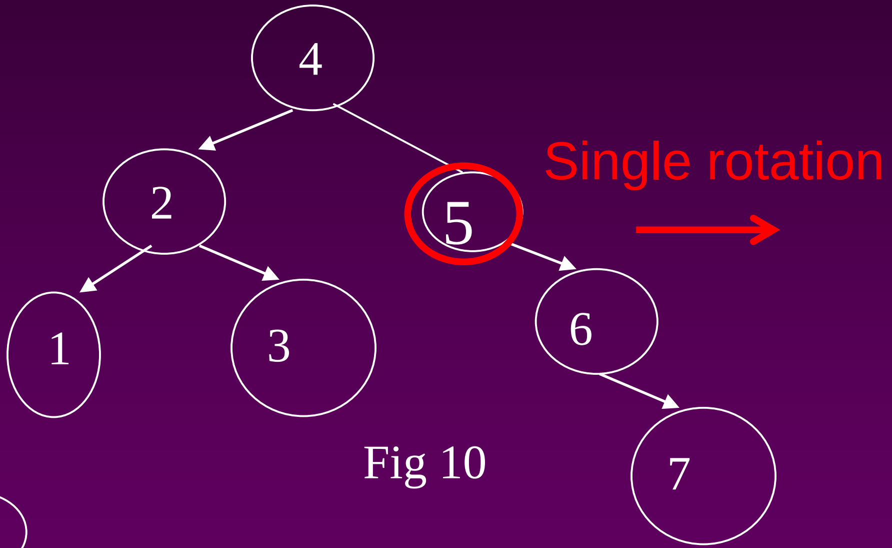


Fig 10

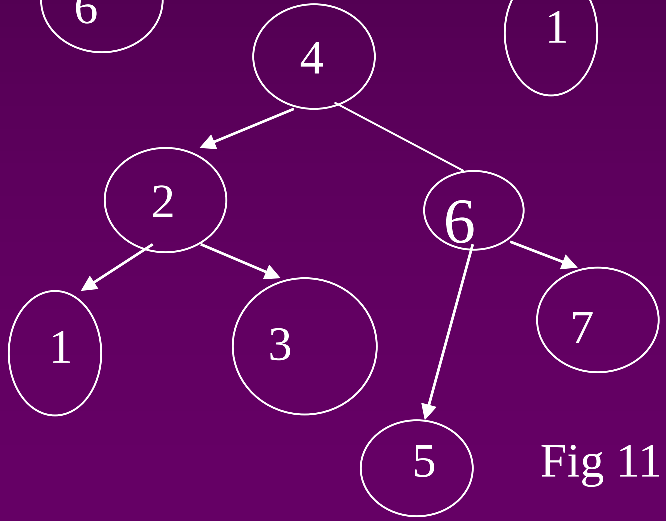


Fig 11

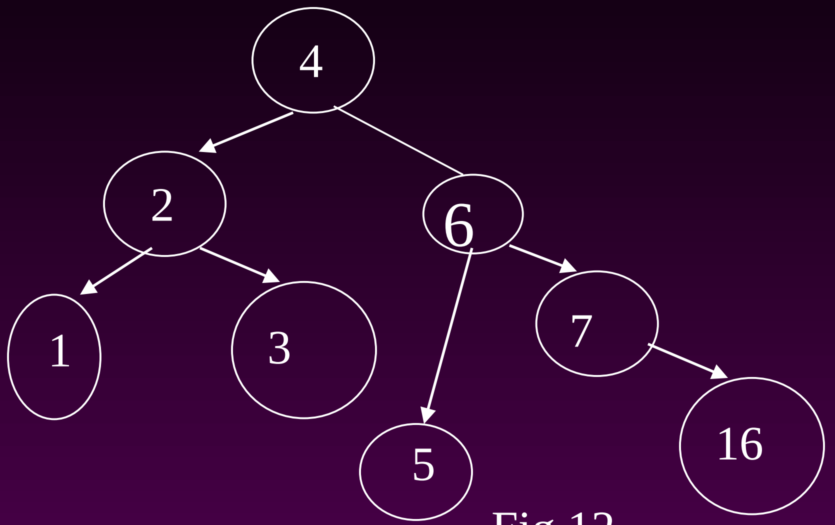


Fig 12

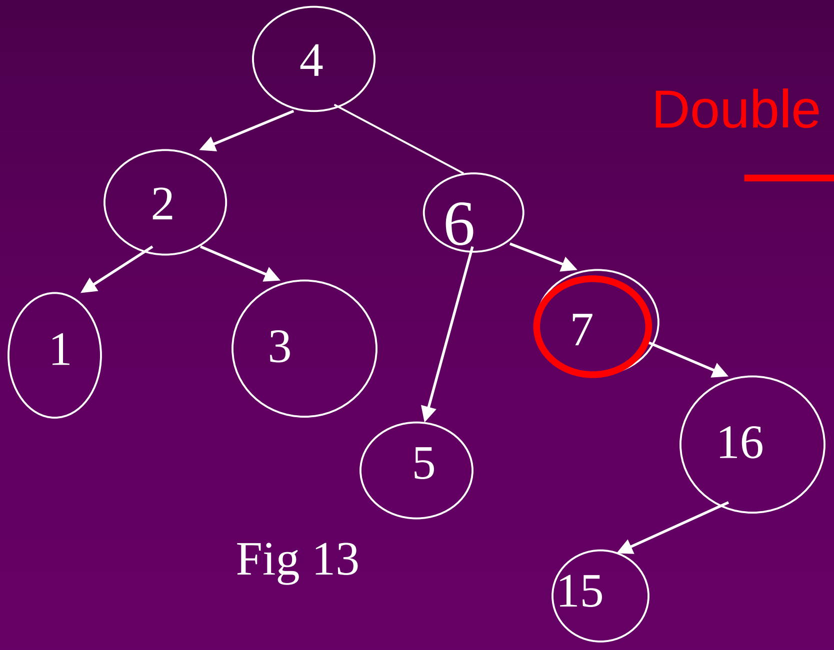


Fig 13

Double rotation

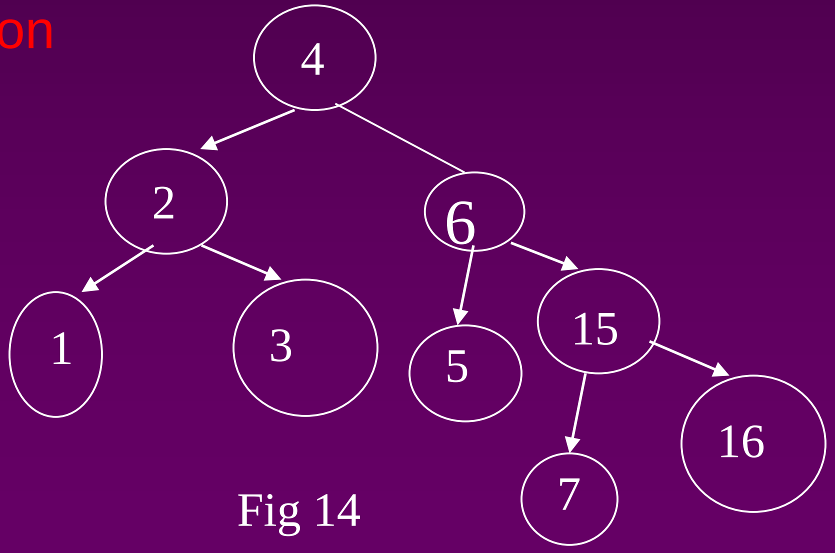


Fig 14

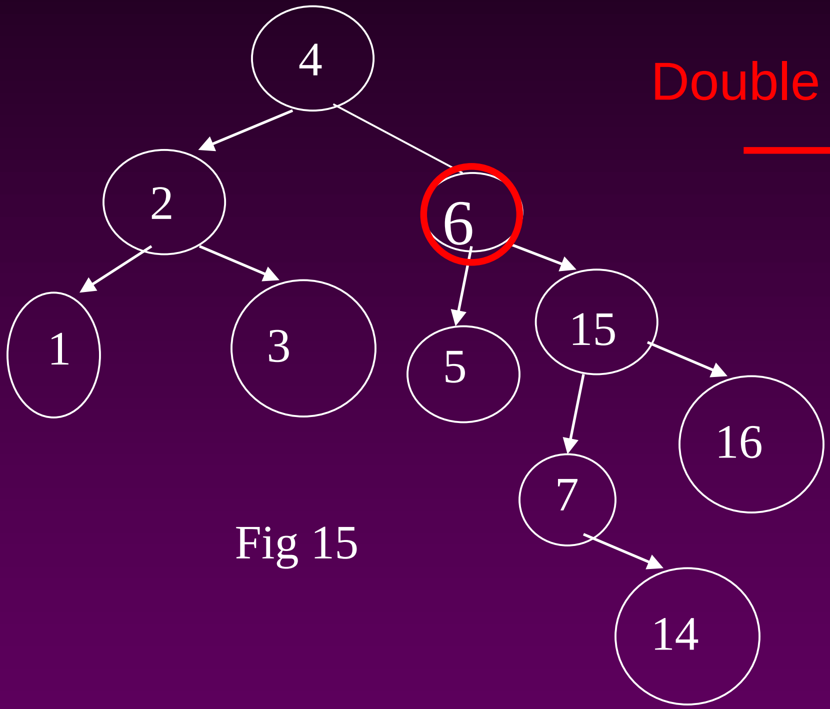


Fig 15

Double rotation

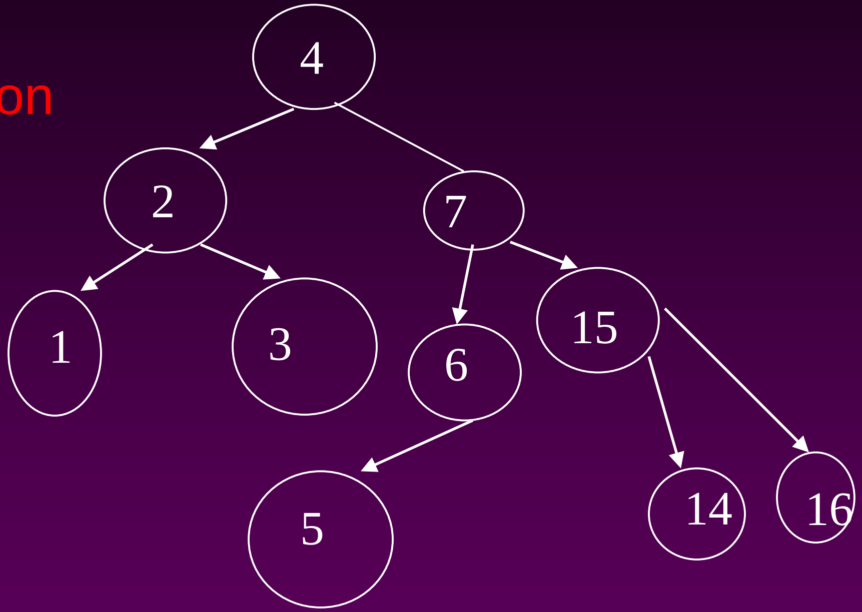
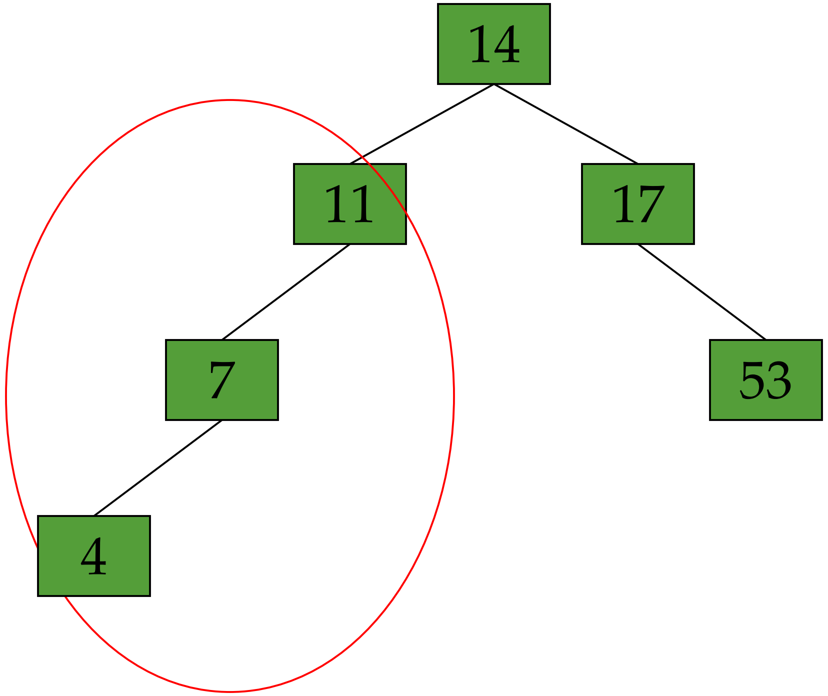


Fig 16

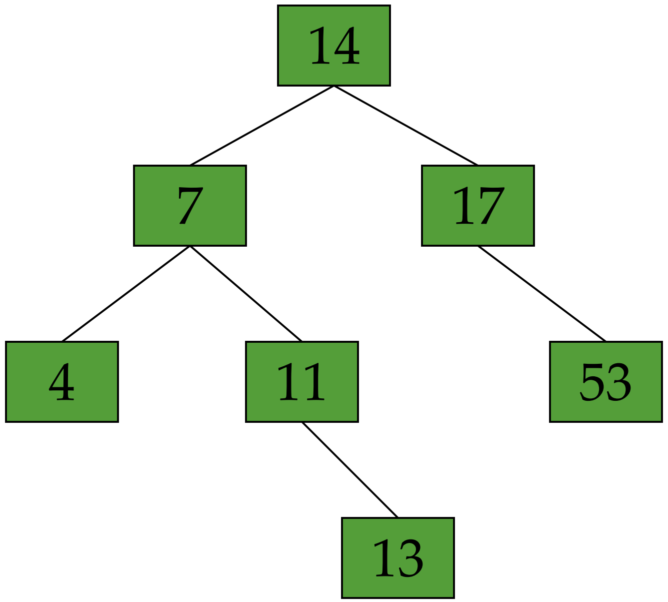
AVL Tree Example:

- **Insert 14, 17, 11, 7, 53, 4, 13 into an empty AVL tree**



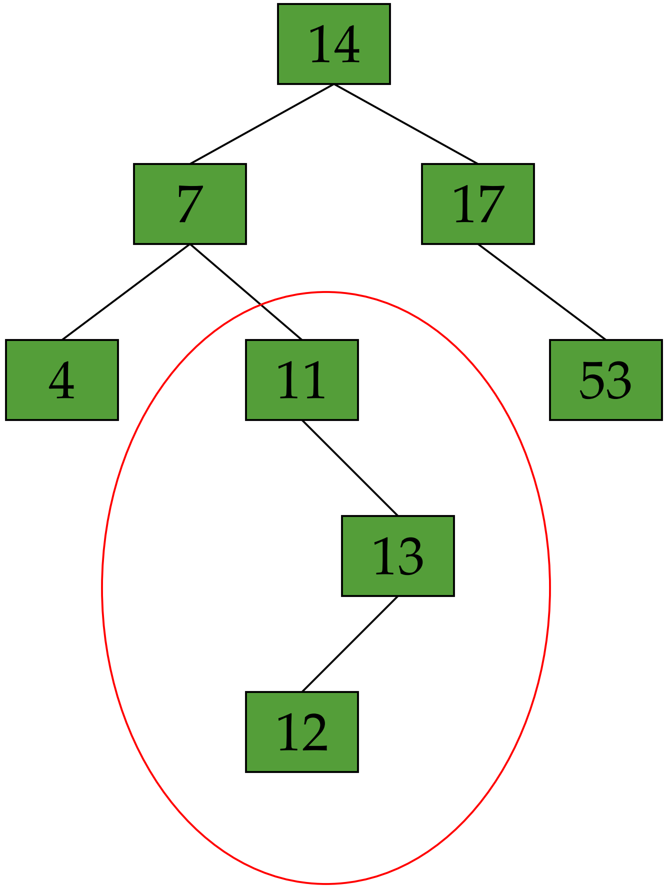
AVL Tree Example:

- **Insert 14, 17, 11, 7, 53, 4, 13 into an empty AVL tree**



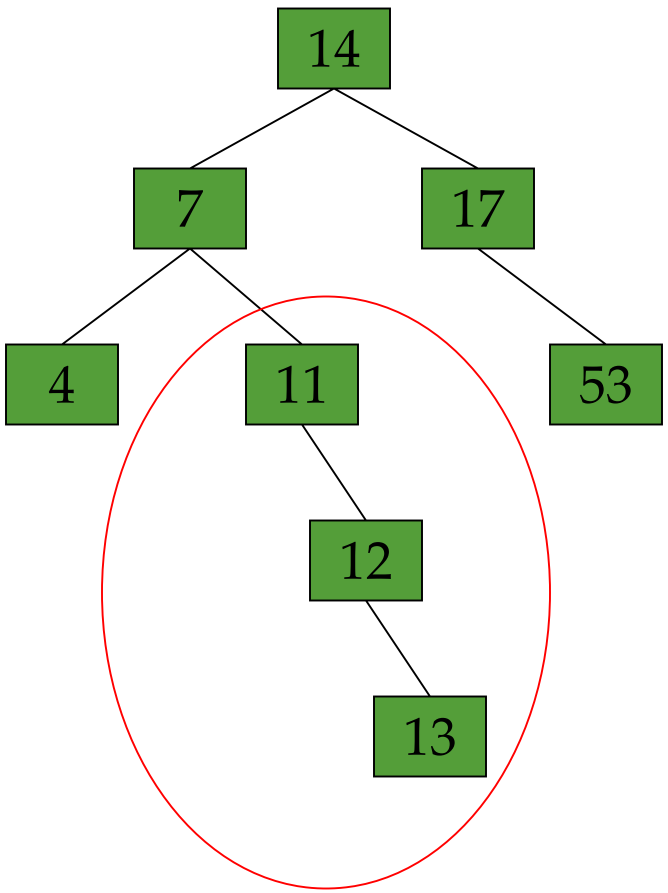
AVL Tree Example:

- Now insert 12



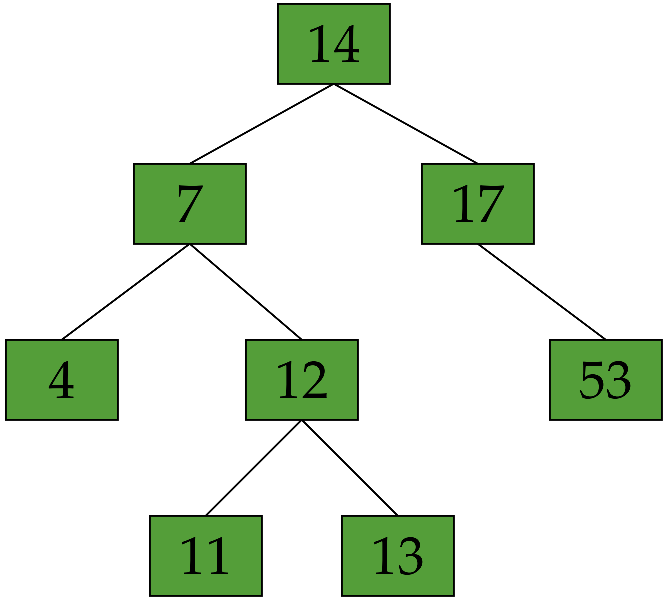
AVL Tree Example:

- **Now insert 12**



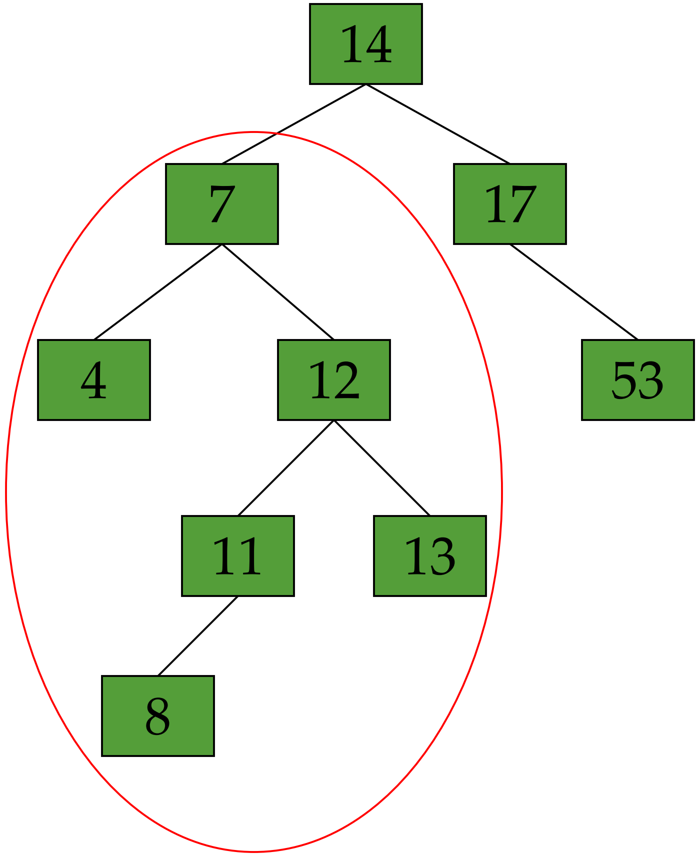
AVL Tree Example:

- **Now the AVL tree is balanced.**



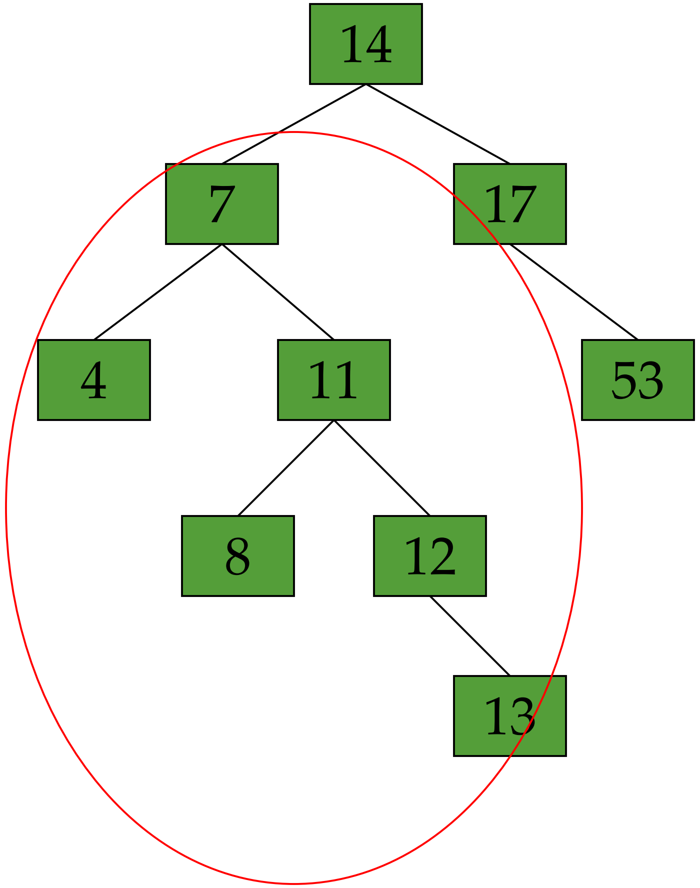
AVL Tree Example:

- Now insert 8



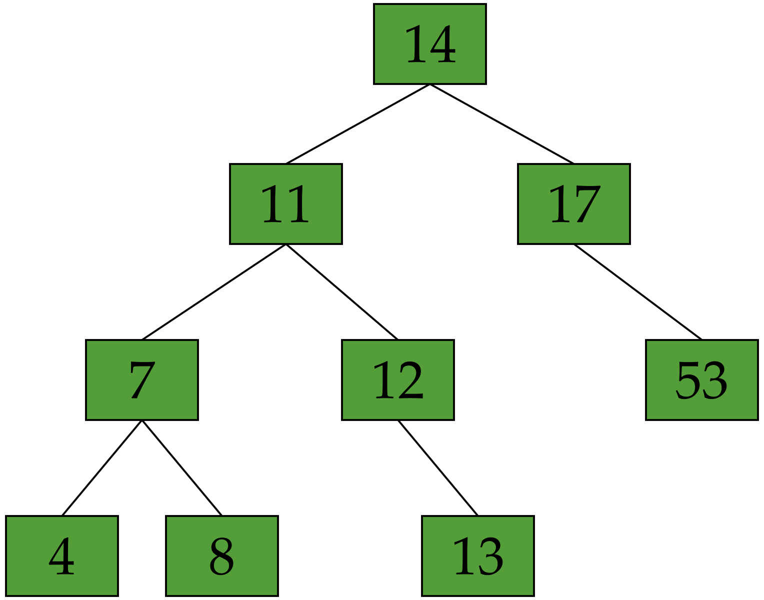
AVL Tree Example:

- **Now insert 8**



AVL Tree Example:

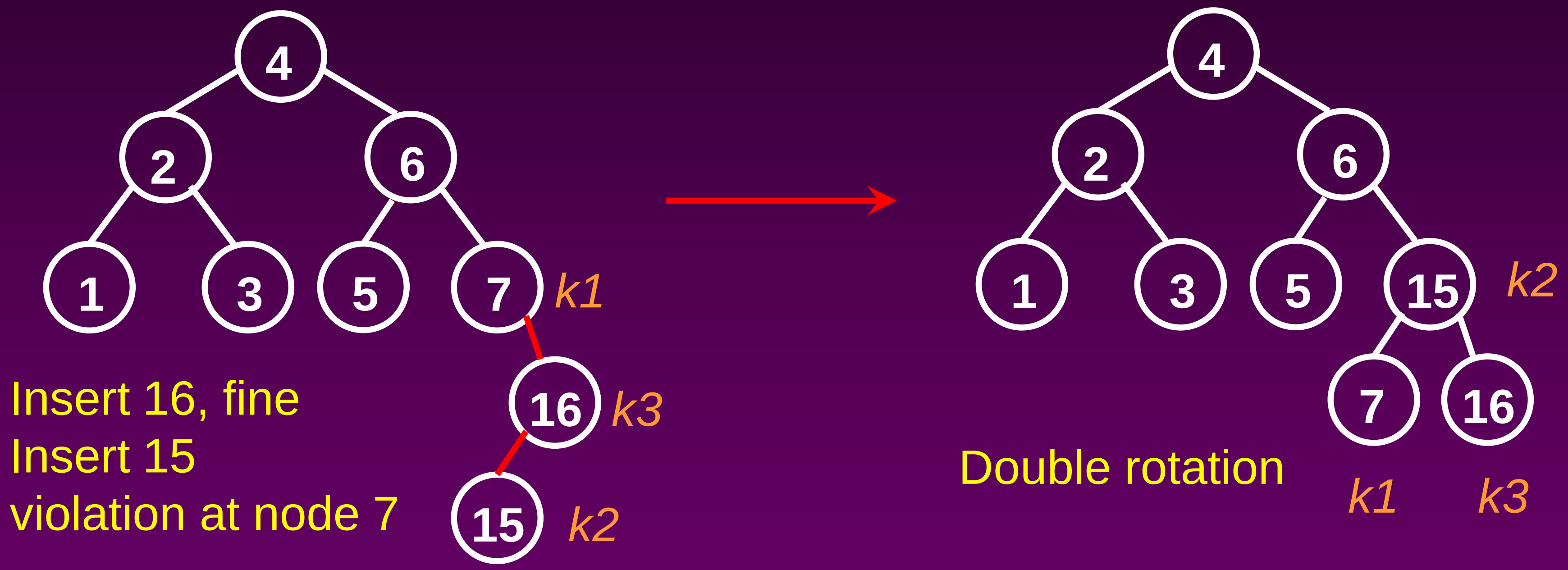
- **Now the AVL tree is balanced.**

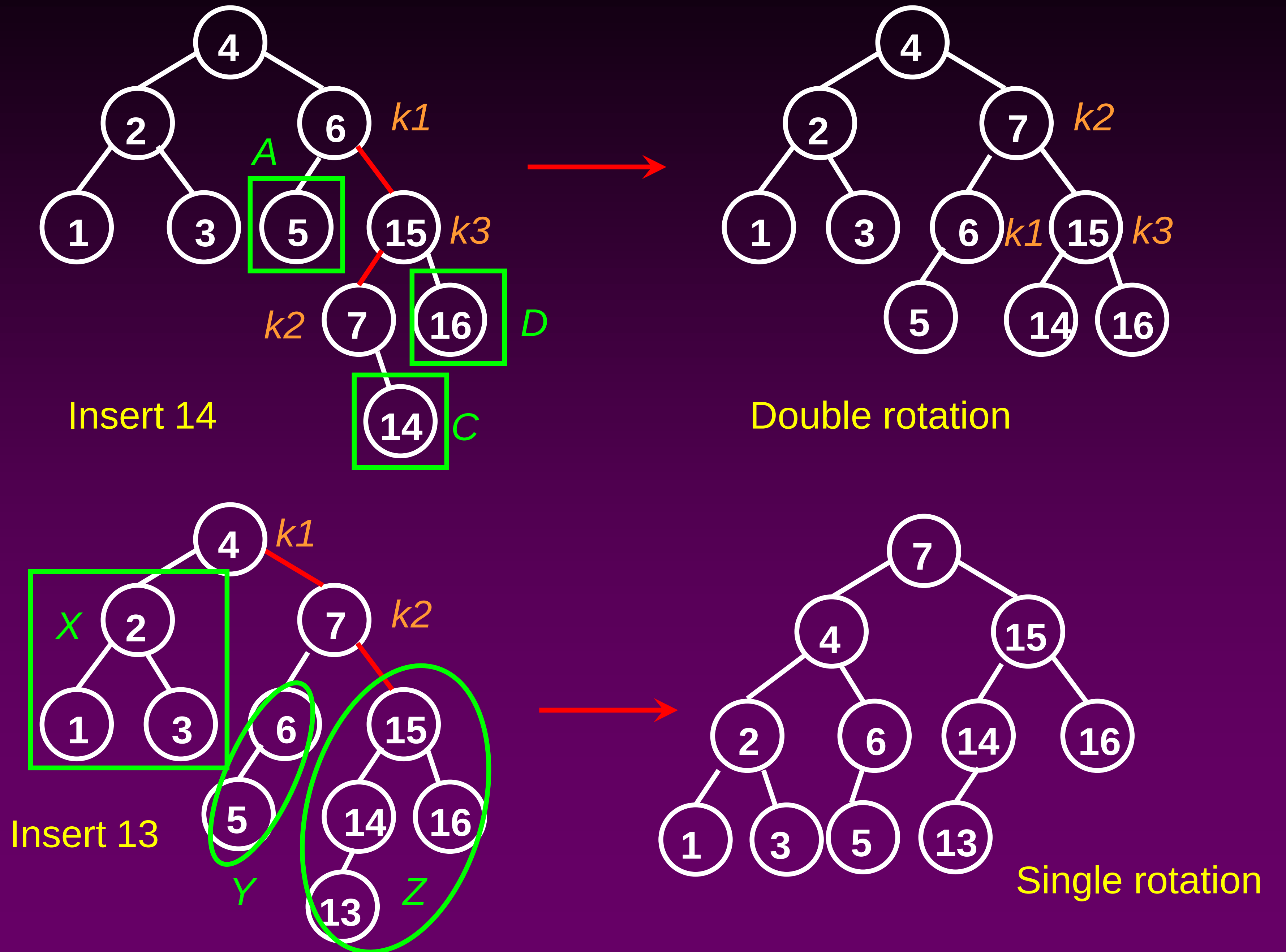


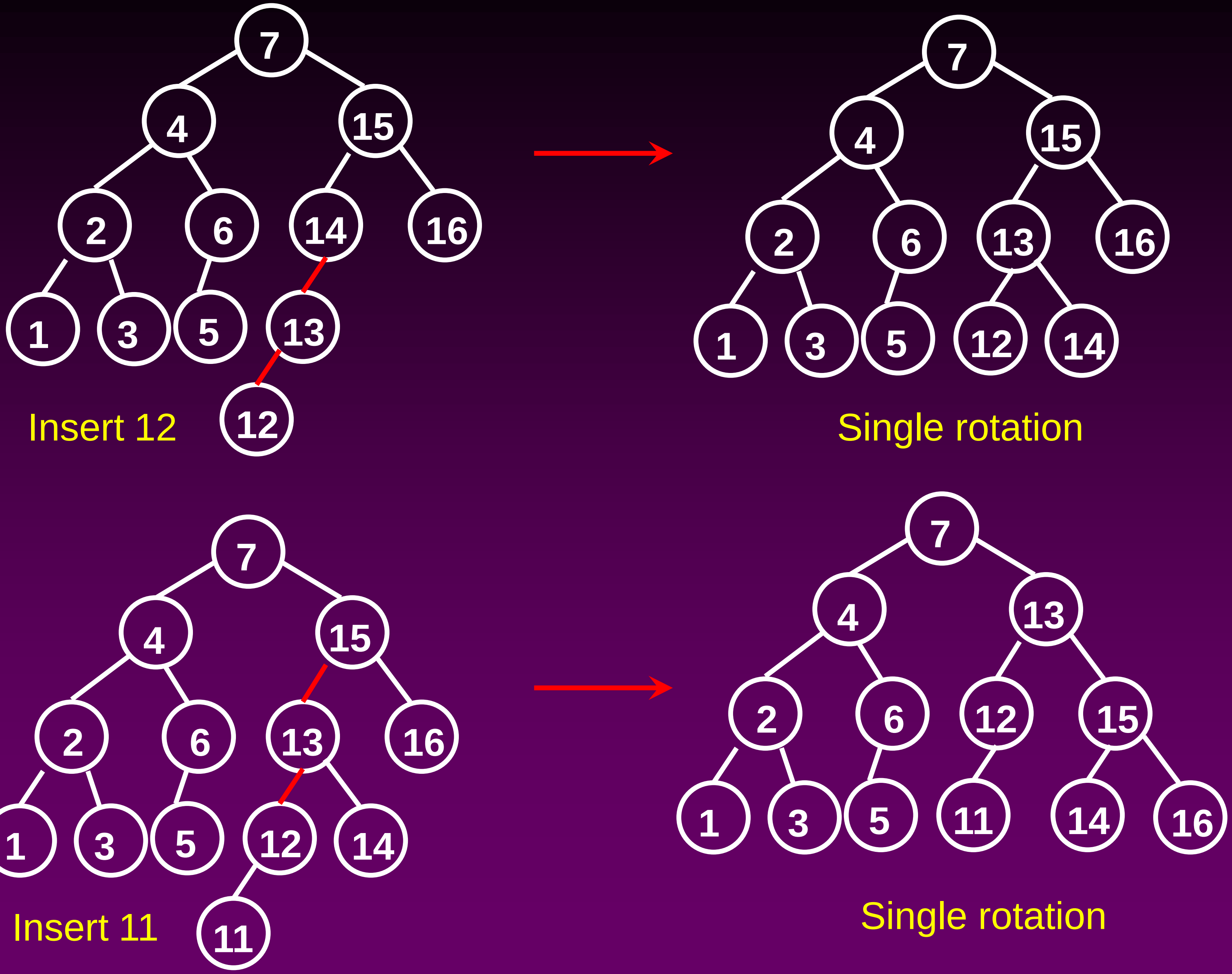
- Construct the AVL tree for given nodes:
  - 40, 20, 10, 25, 30, 22, 50.
  - 3,2,1,4,5,6,7,16,15,14,13,12,11,10,8,9.
  - 1, 2, 3, 6, 5, -2, -5, -8.

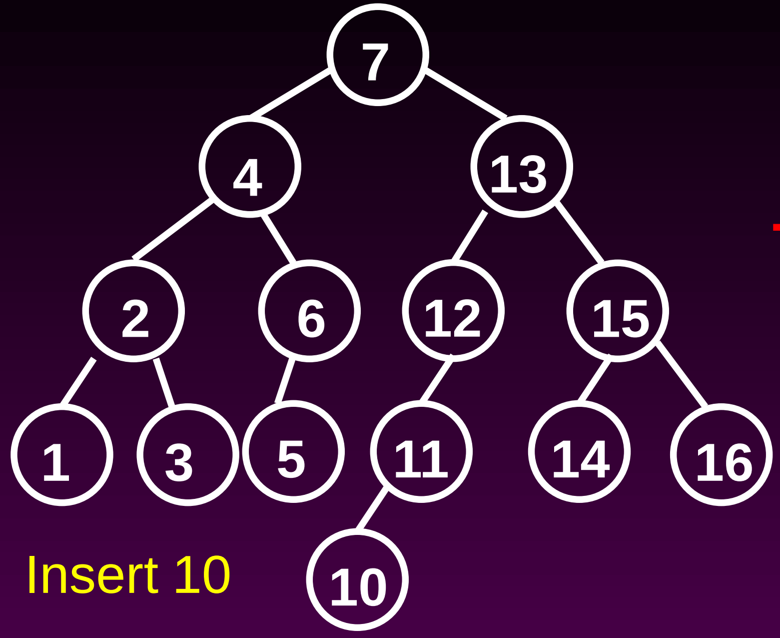
We’ve inserted 3, 2, 1, 4, 5, 6, 7

We’ll insert16, 15, 14, 13, 12, 11, 10, 8, 9

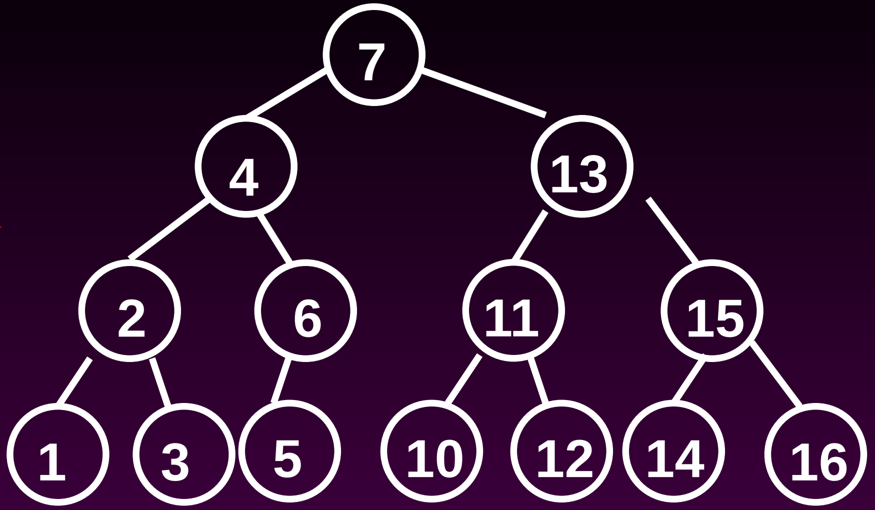



k1
A
k3
k2
D
C
Insert 14
k2
k1
k3
Double rotation
k1
X
k2
Insert 13
Y
Z
Single rotation

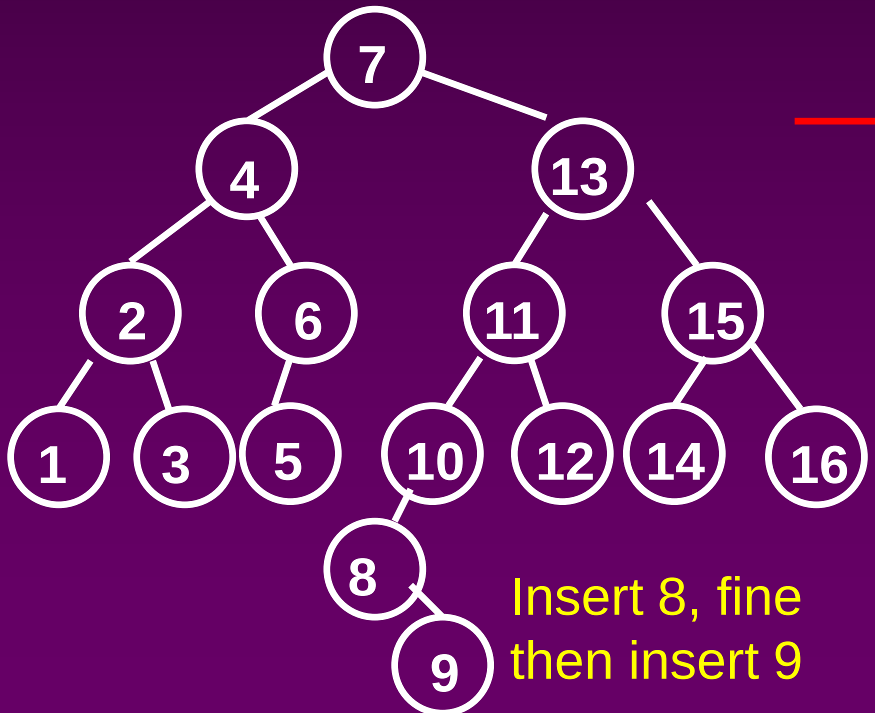

Insert 12
Single rotation
Insert 11
Single rotation



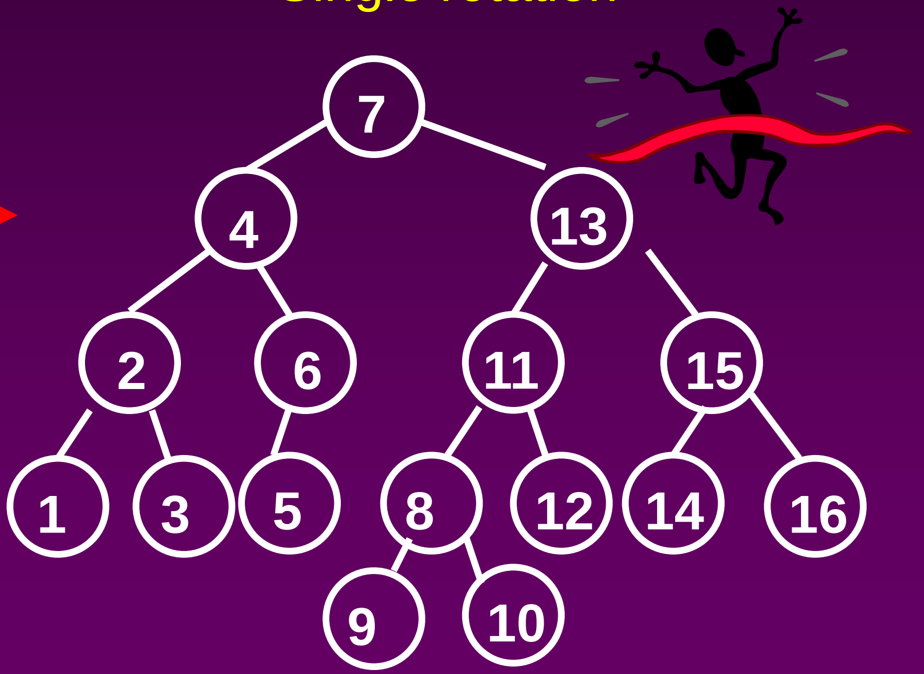
Insert 10



Single rotation

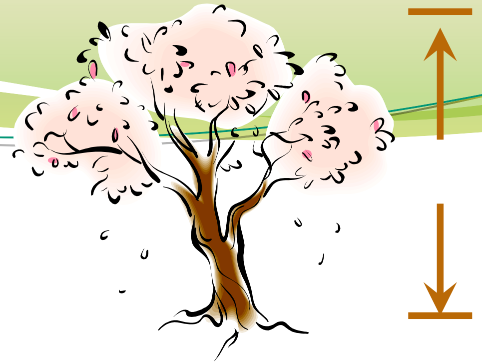


Insert 8, fine
then insert 9



Double rotation

Insertion Analysis



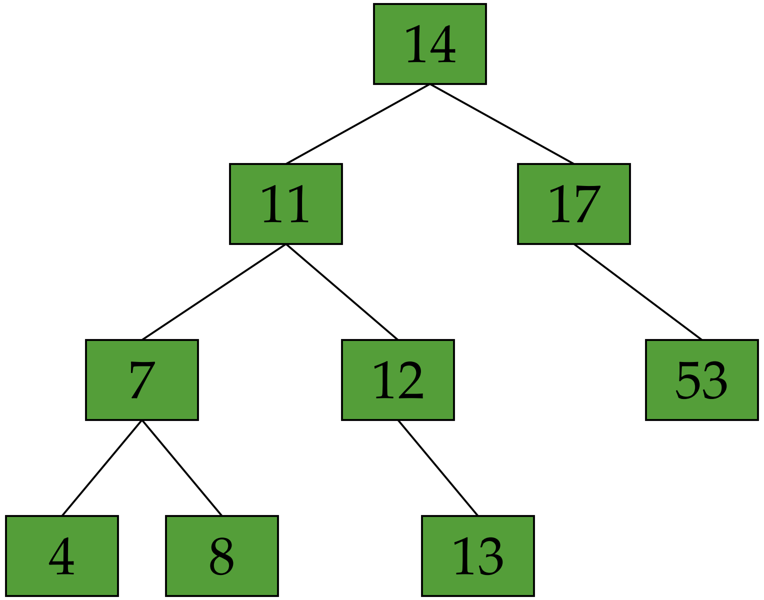
- Insert the new key as a new leaf just as in ordinary binary search tree: O(logN)
- Then trace the path from the new leaf towards the root, for each node x encountered: O(logN)
  - Check height difference: O(1)
  - If satisfies AVL property, proceed to next node: O(1)
  - If not, perform a rotation: O(1)
- The insertion stops when
  - A rotation is performed
  - Or, we've checked all nodes in the path
- Time complexity for insertion O(logN)

Deletion in AVL tree

- While deleting a node from the AVL tree follows the deletion of Binary Search Tree.
- After deleting check the balance factor of the nodes. If tree is imbalance then make it balance one.

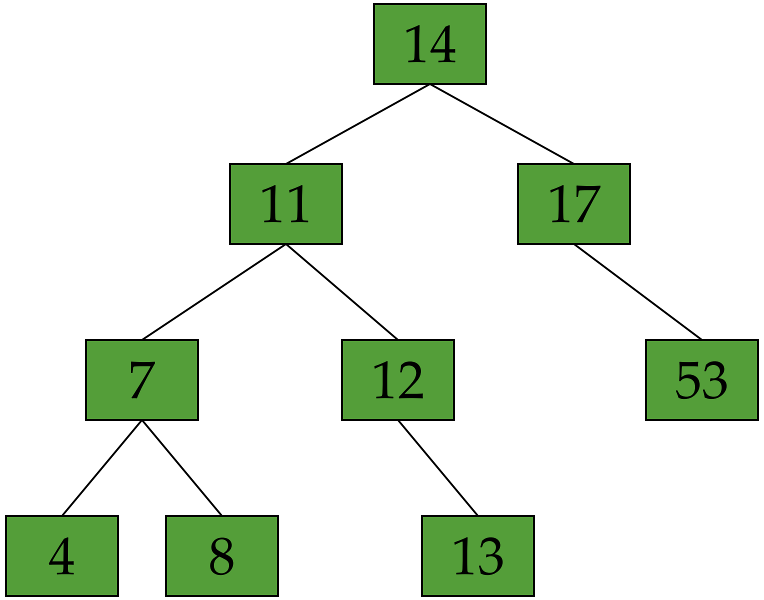
AVL Tree Example:

- **Now the AVL tree is balanced.**



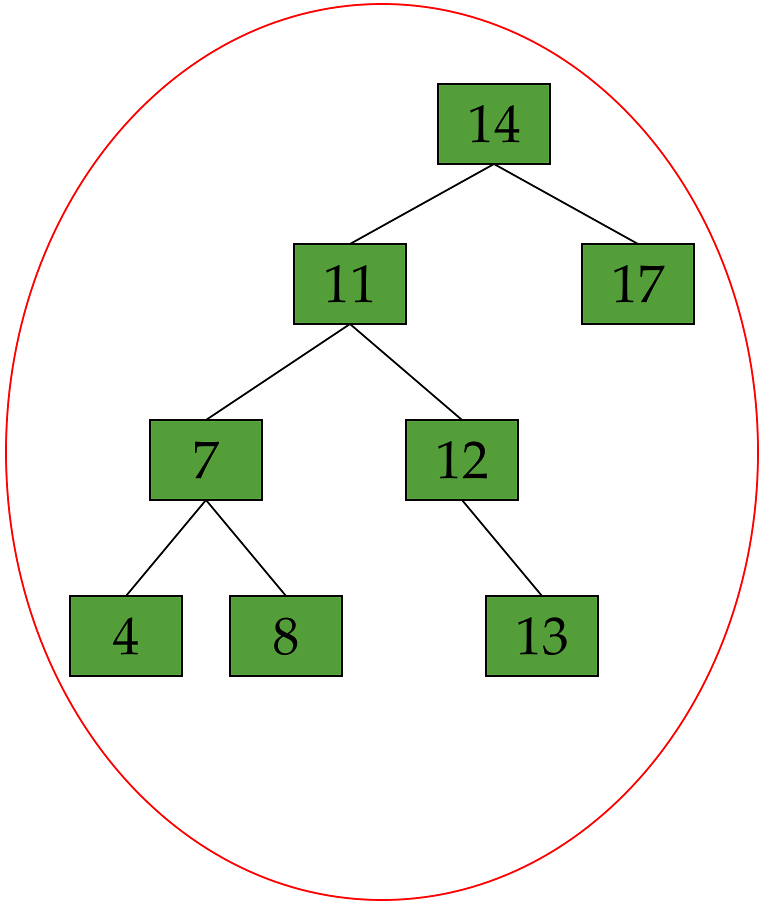
AVL Tree Example:

- **Now remove 53**



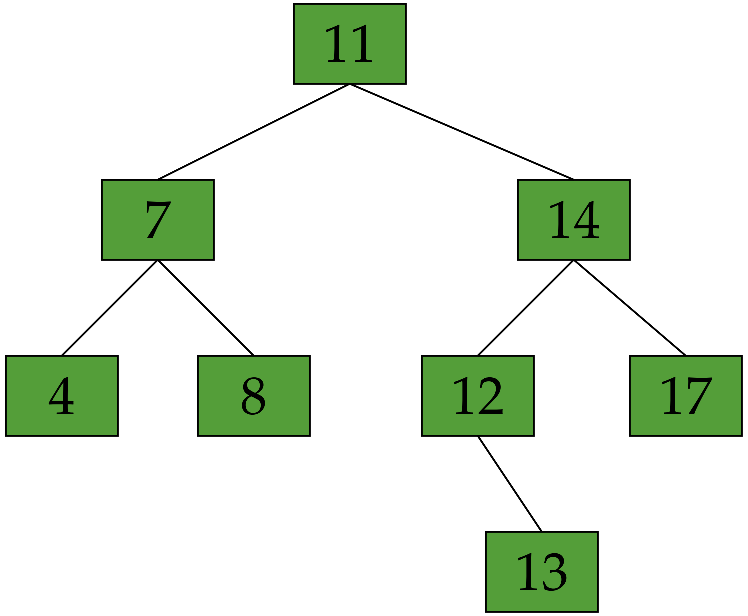
AVL Tree Example:

- **Now remove 53, unbalanced**



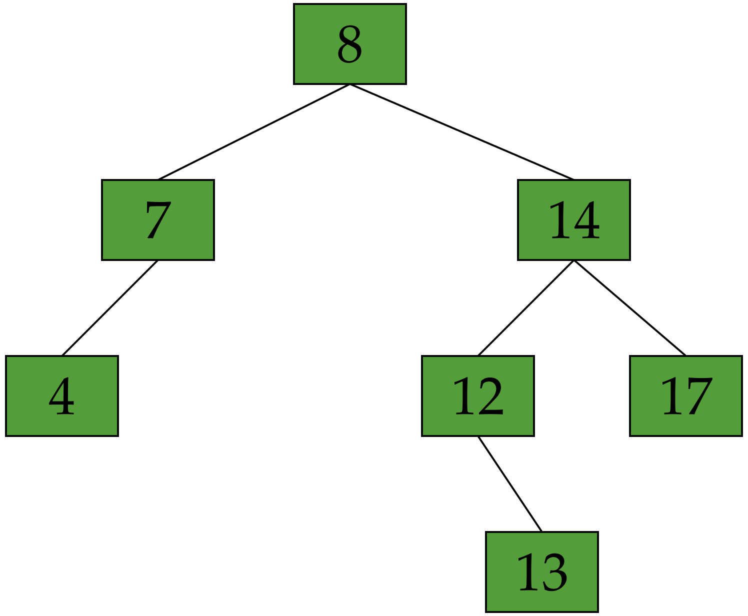
AVL Tree Example:

- **Balanced!   Remove 11**



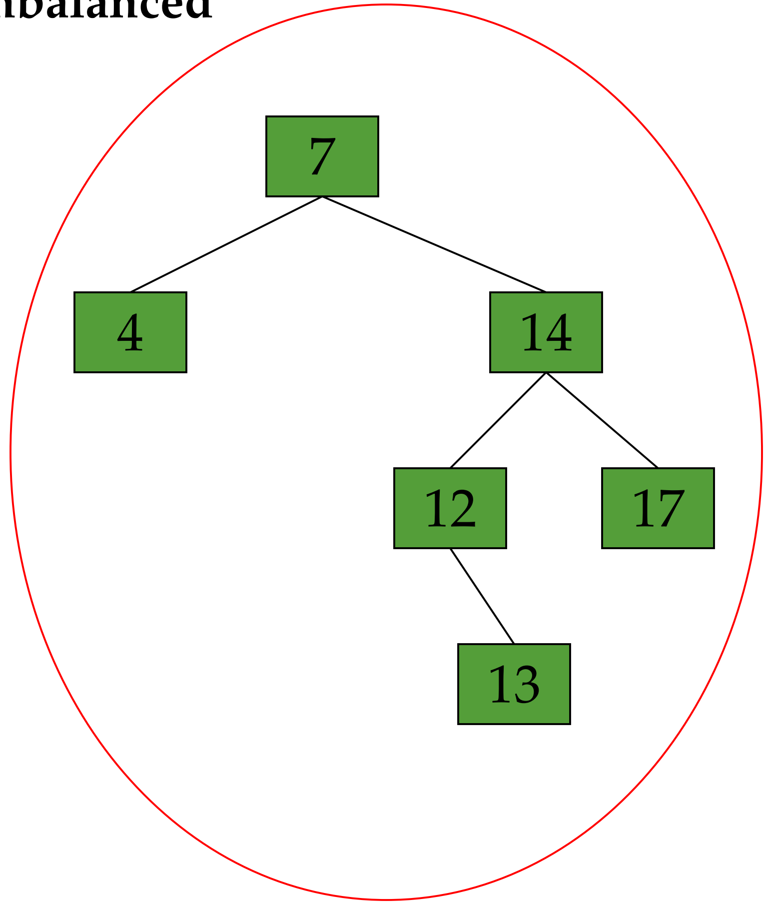
AVL Tree Example:

- **Remove 11, replace it with the largest in its left branch**



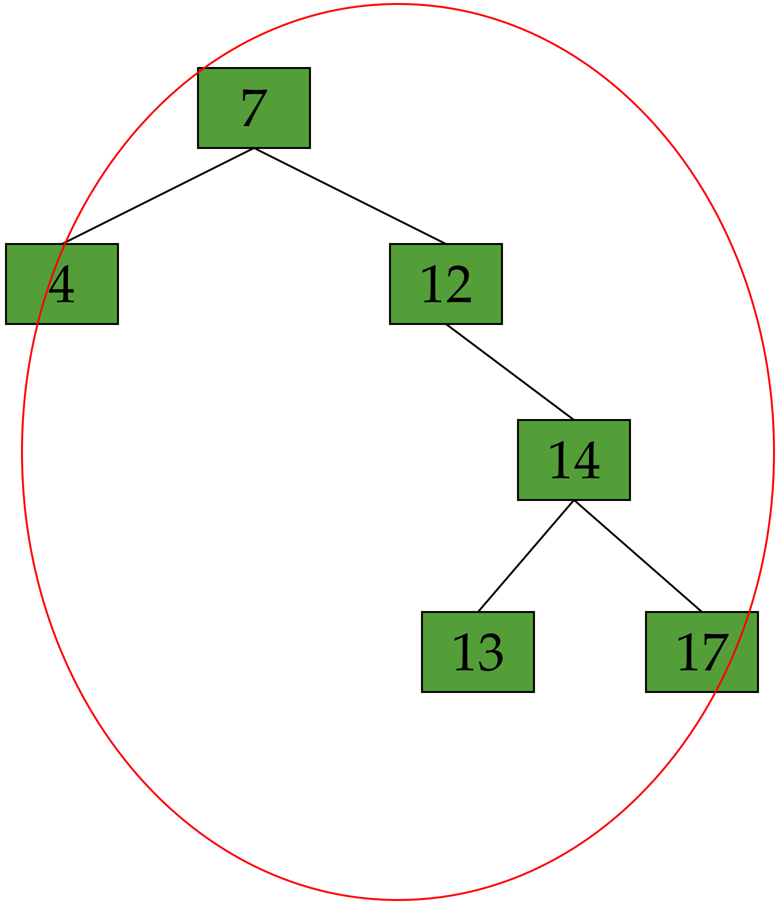
AVL Tree Example:

- **Remove 8, unbalanced**



AVL Tree Example:

- **Remove 8, unbalanced**



AVL Tree Example:

- **Balanced!!**