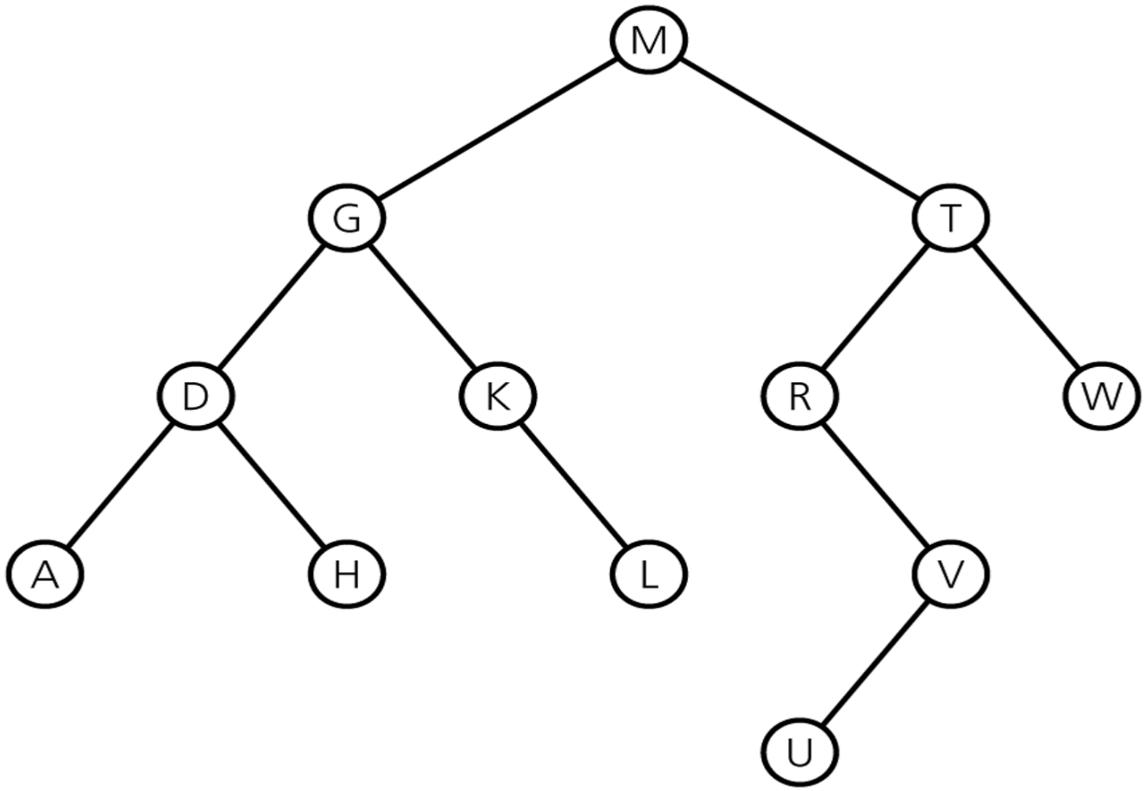
Exercises

1. What are preorder, postorder and inorder traversals of the following binary tree.
2. Is it a BST



2. Assume that the *inorder* traversal of a binary tree is

C G A H F D E I B J

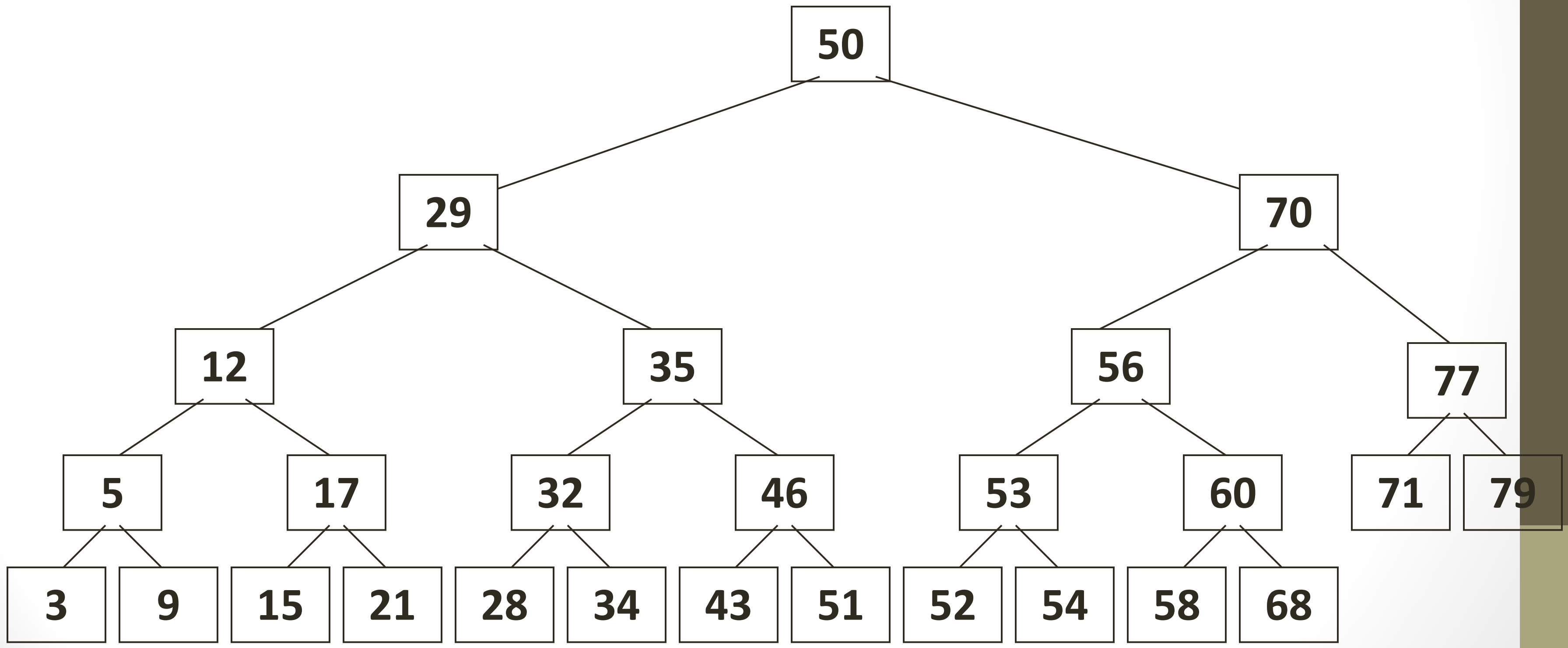
and its *postorder* traversal is

G C H F A I E J B D

Draw this binary tree.

Question?

- Is this a binary tree search tree?



Convert the infix to pre and post fix

- ((A + B) * (C + D)) – Has paranthesis
- (((A + B) * C) - ((D + E) / F))

Identifyme

```
int IM (Node u)
    if (u == nil)
        return -1;
    return 1 + max(IM(u.left), IM(u.right));
```
	```
	int IM (Node u)
	    if (u == nil)
	        return 0;
	    return 1 + IM(u.left) + IM(u.right);
	```

```
TRANSPLANT(T, u, v)
    if u.p == NIL
        T.root = v
    elseif u == u.p.left
        u.p.left = v
    else u.p.right = v
    if v != NIL
        v.p = u.p
```
	```
	int Identifyme(Node u)
	        int d = 0;
	        while (u != r)
	                u = u.parent;
	                d++;
	        return d;
	```

Non recursive functions : 1. Size 2. Height 3. Minimum and Maximum in BST	1. Traverse 2. Search 3. Successor and Predessor 4. Delete item in tree using transplant

Fill in the blanks

- A perfect binary tree of height h has   nodes
- A perfect binary tree with n nodes has height
- A perfect binary tree with height h has   leaf nodes
- The height of a complete binary tree with n nodes is
- A perfect n-ary tree of height h has nodes

$$n = \sum_{k=0}^{h} N^k = \frac{N^{h+1} - 1}{N - 1}$$

- A perfect n-ary tree with n nodes has height

$$h = \log_N \big(n(N-1)+1\big) - 1$$

Background

Run times depend on the height of the trees

As was noted in the previous section:

- The best case height is $\Theta(\ln(n))$
- The worst case height is $\Theta(n)$

The average height of a randomly generated binary search tree is actually $\Theta(\ln(n))$

Requirement for Balance

We want to ensure that the run times never fall into $\mathrm{O}(\ln(n))$

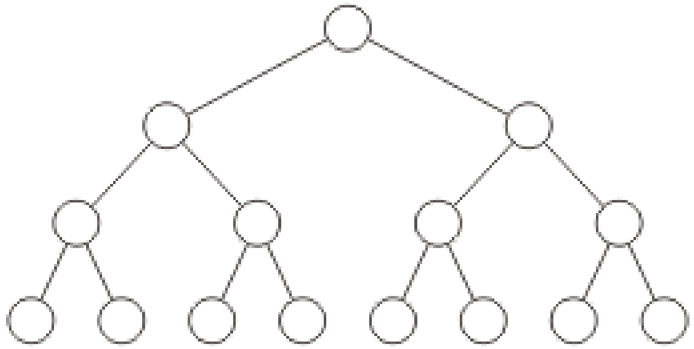
Requirement:

- We must maintain a height which is $\Theta(\ln(n))$

To do this, we will define an idea of balance

Examples

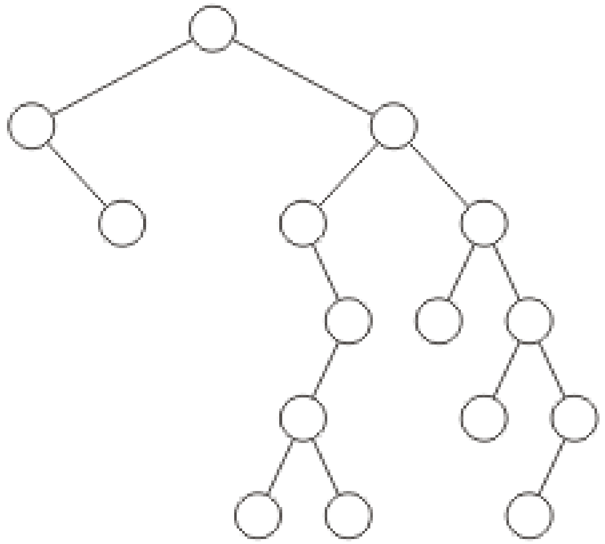
For a perfect tree, all nodes have the same number of descendants on each side



Perfect binary trees are balanced while linked lists are not

Examples

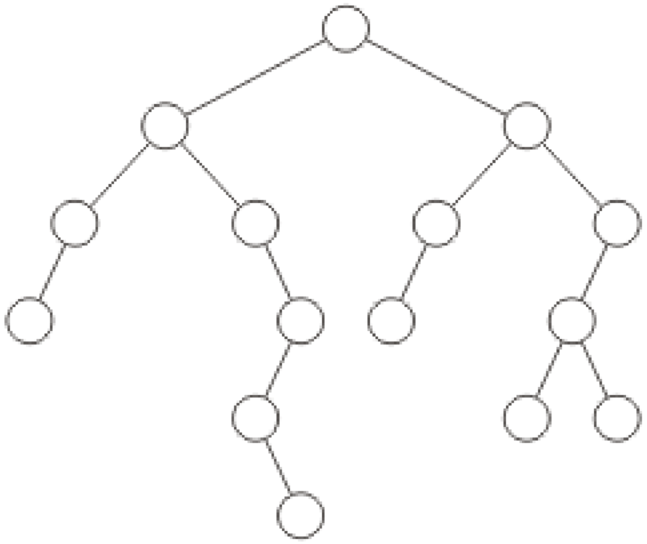
This binary tree would also probably not be considered to be “balanced” at the root node



Examples

How about this example?

- The root seems balanced, but what about the left sub-tree?



Definition for Balance

We must develop a quantitative definition of *balance* which can be applied

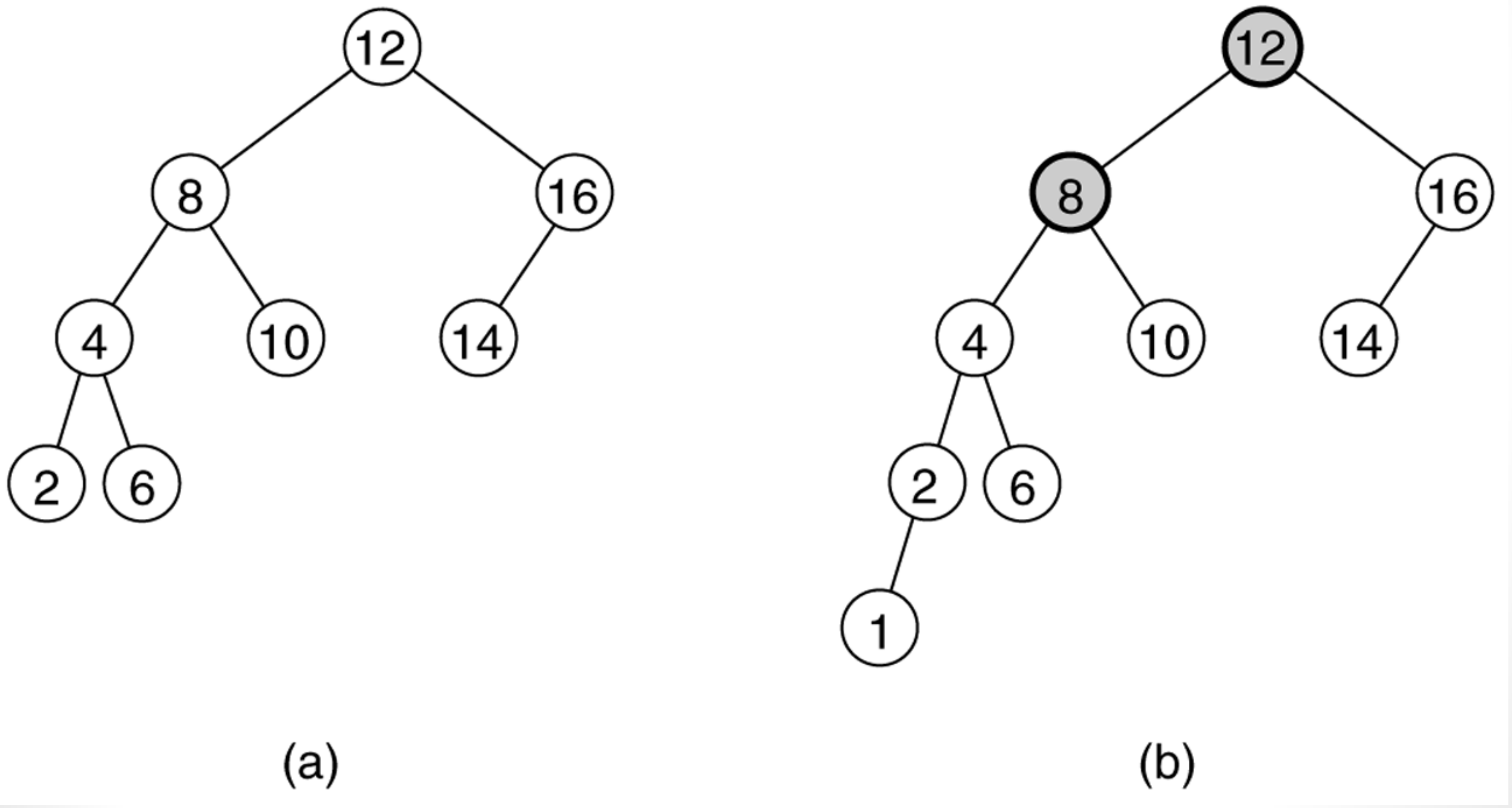
Balanced may be defined by:

- *Height balancing*: comparing the heights of the two sub trees
- *Null-path-length balancing*: comparing the null-path-length of each of the two sub-trees (the length to the closest null sub-tree/empty node)
- *Weight balancing*: comparing the number of null sub-trees in each of the two sub trees

It is mathematically proved that if a tree satisfies the definition of balance, its height is $\Theta(\ln(n))$

Height balancing trees

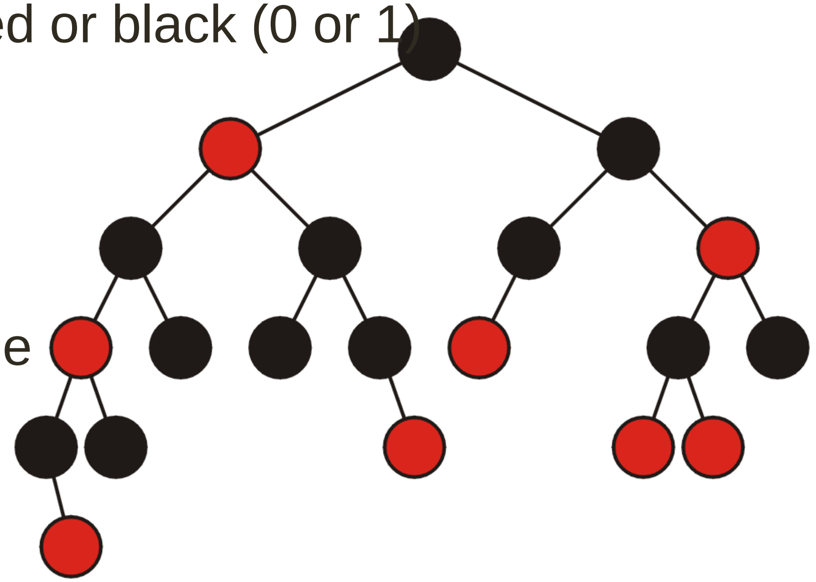
AVL trees



Red-Black Trees

Red-black trees maintain balance by

- All nodes are *colored* red or black (0 or 1)

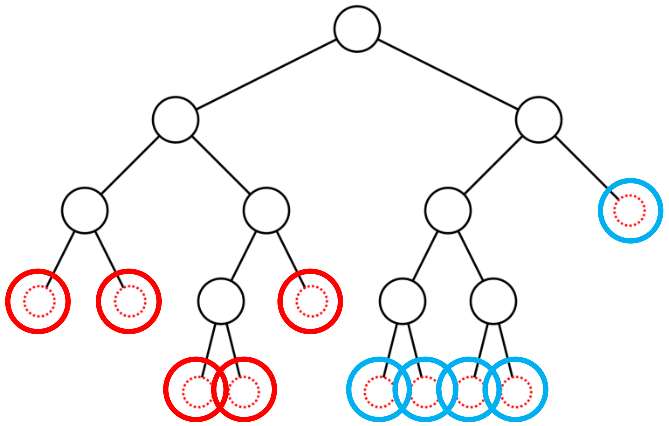


Requirements:

- The root must be black
- All children of a red node must be black
- Any path from the root to an empty node must have the same number of black nodes

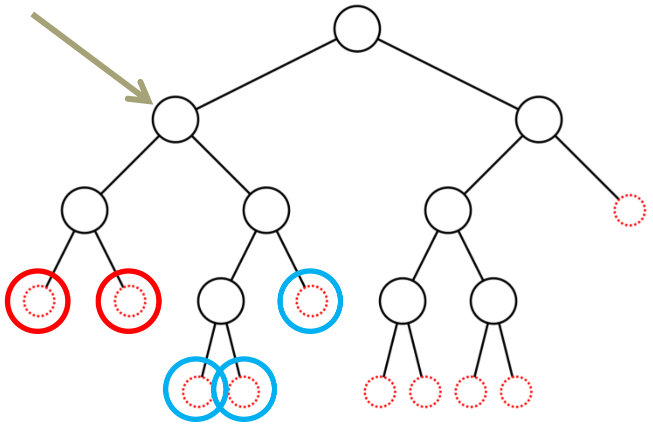
Weight-Balanced Trees

The ratios of the empty nodes at the root node are 5/10 and 5/10



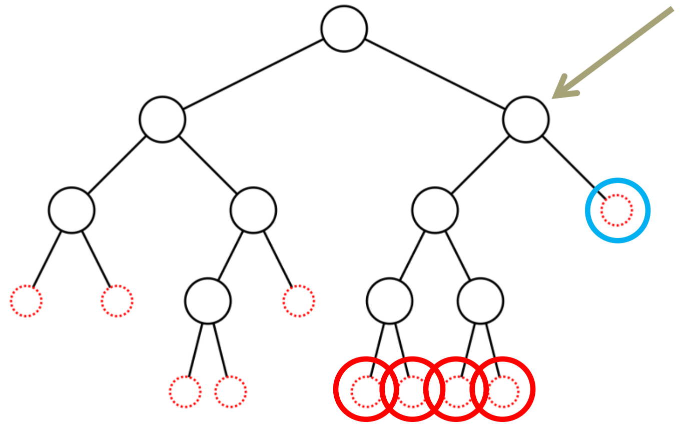
Weight-Balanced Trees

The ratios of the empty nodes at this node are 2/5 and 3/5



Weight-Balanced Trees

The ratios of the empty nodes at this node, however, are 4/5 and 1/5



AVL Trees

AVL Trees

- An AVL tree is a binary search tree with a *balance* condition.
- AVL is named for its inventors: **A**del'son-**V**el'skii and **L**andis
- AVL tree *approximates* the ideal tree (completely balanced tree).
- AVL Tree maintains a height close to the minimum.

Definition:

An AVL tree is a binary search tree such that

for any node in the tree, the height of the left and

right subtrees can differ by at most 1.

Figure 19.21
Two binary search trees: (a) an AVL tree; (b) not an AVL tree (unbalanced nodes are darkened)

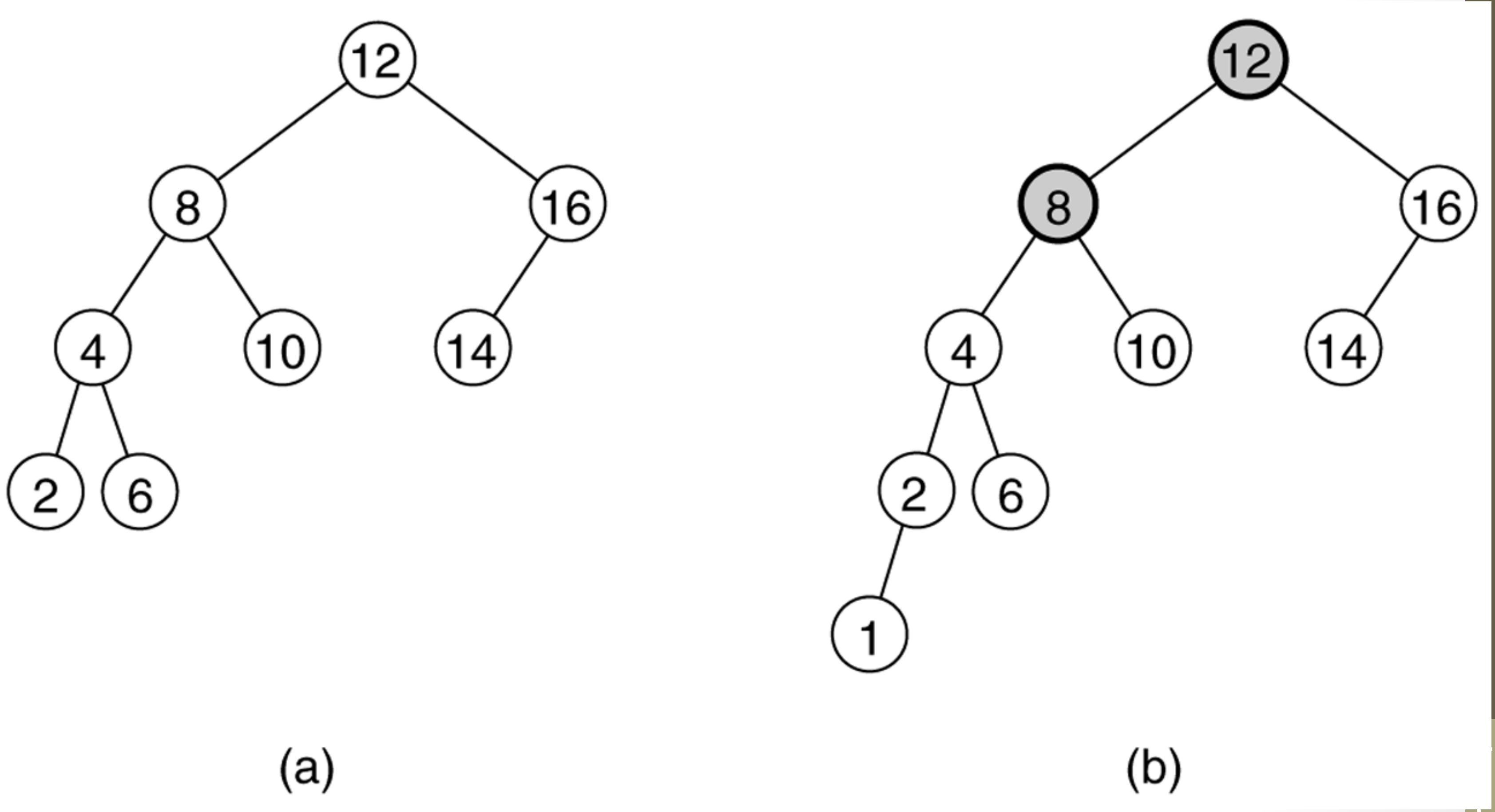
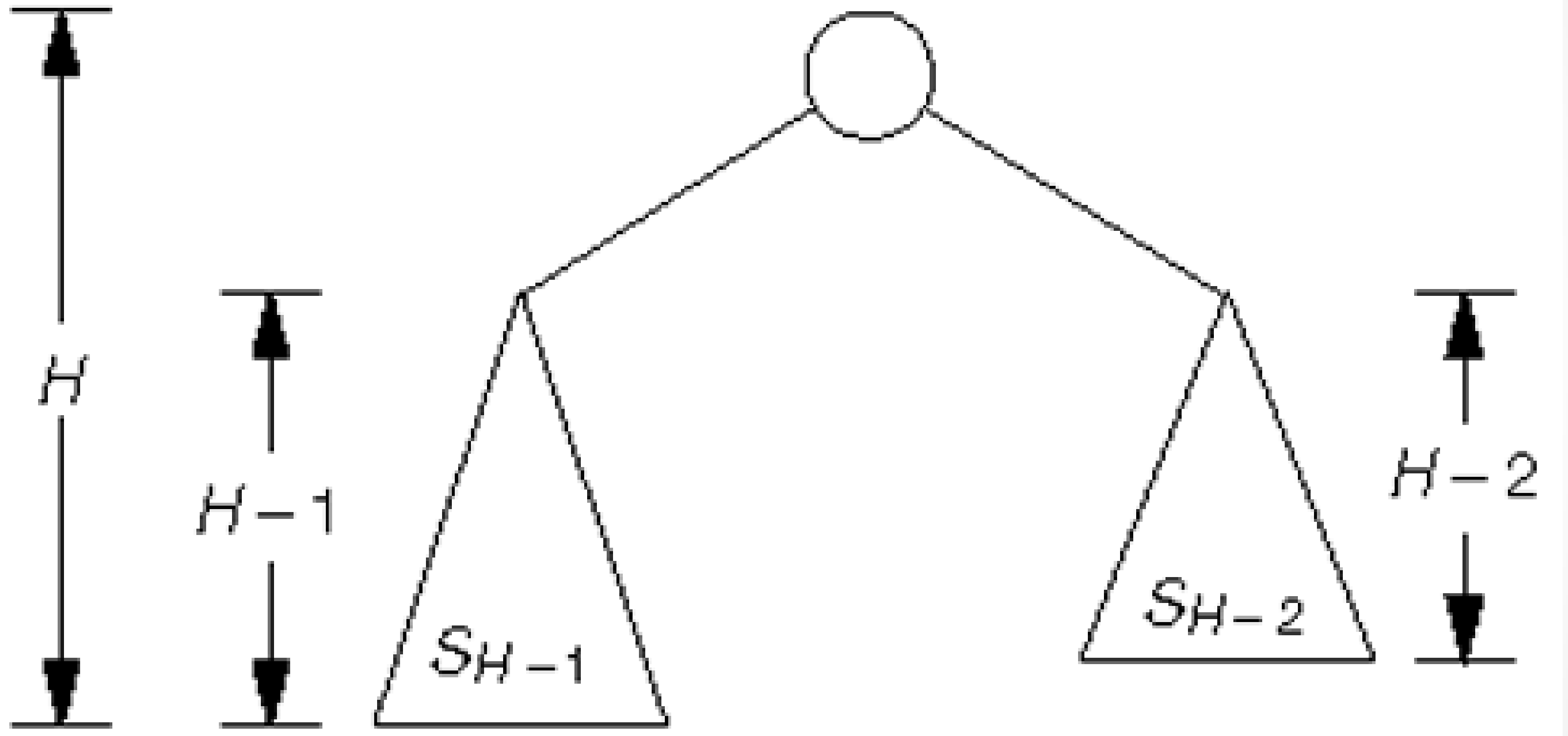
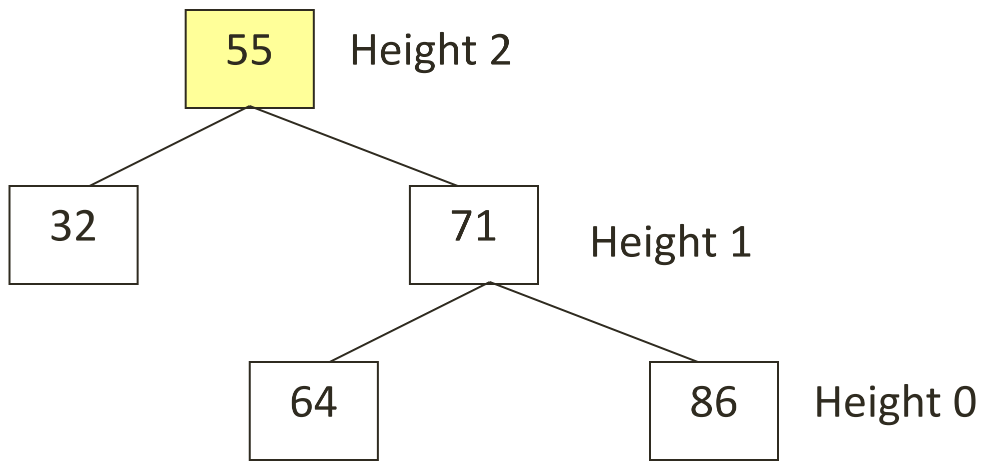


Figure 19.22
Minimum tree of height *H*



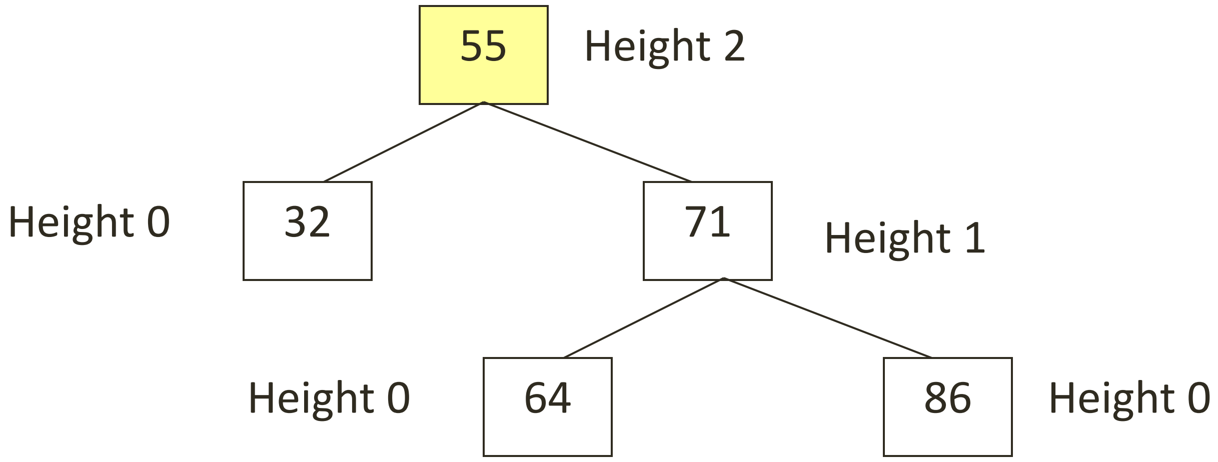
AVL Properties

- An AVL tree is a balanced binary tree
- To understand balance we need to understand the notion of **Tree Height**



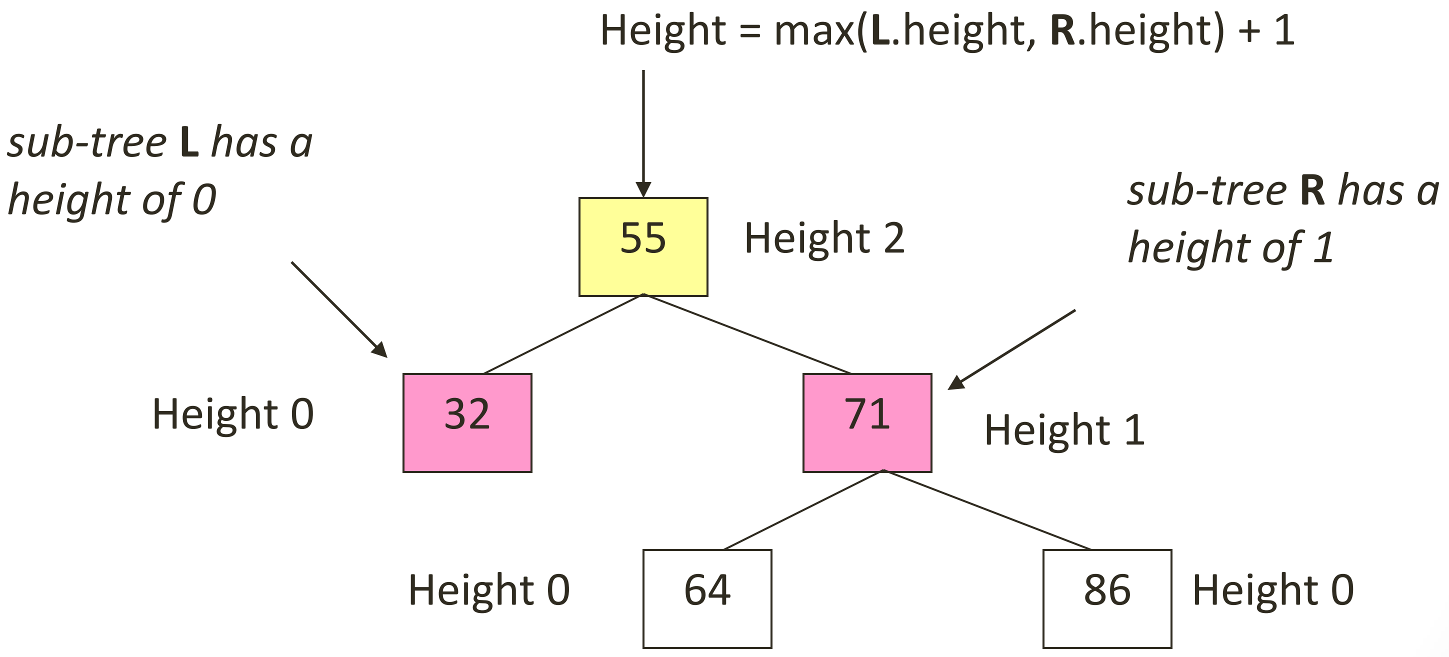
AVL Properties

- By default, nodes with no children have a height of 0.



AVL Properties

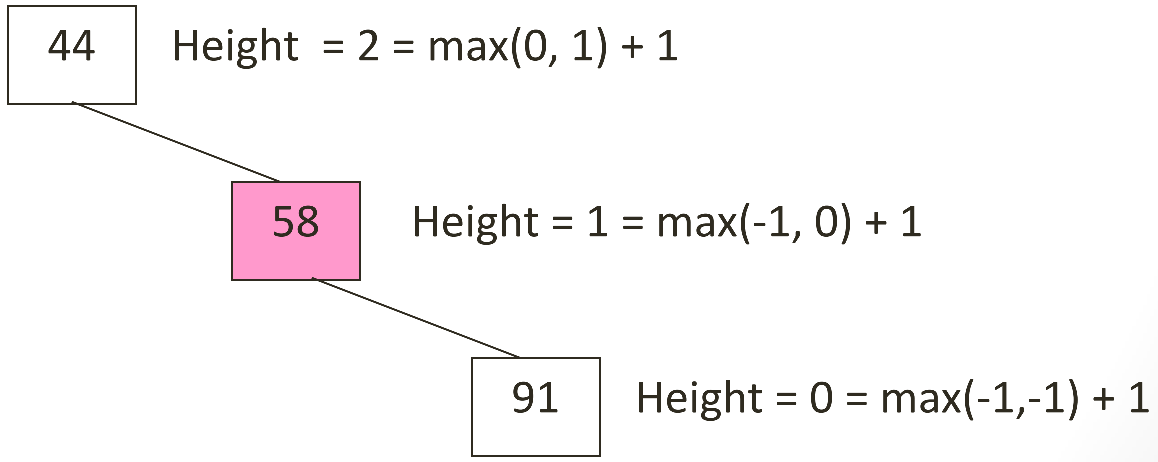
- But, we must also understand the concept of Sub-trees


Height = max(**L**.height, **R**.height) + 1
sub-tree **L** *has a height of 0*
sub-tree **R** *has a height of 1*
55
Height 2
Height 0
32
71
Height 1
Height 0
64
86
Height 0

AVL Properties

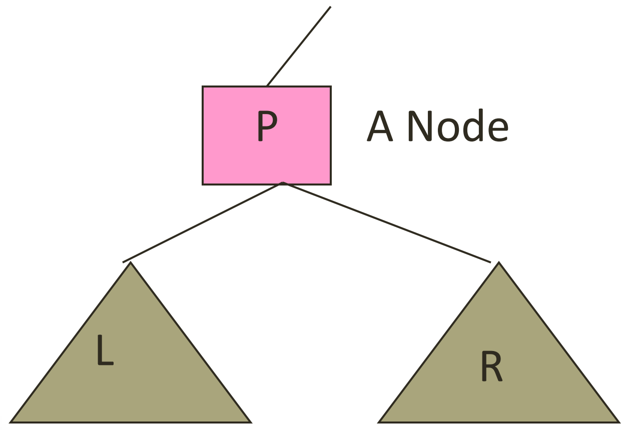
- Also empty sub-trees have a Height of -1

Height = max(**L**.height, **R**.height) + 1


44
Height = 2 = max(0, 1) + 1
58
Height = 1 = max(-1, 0) + 1
91
Height = 0 = max(-1,-1) + 1

AVL Properties

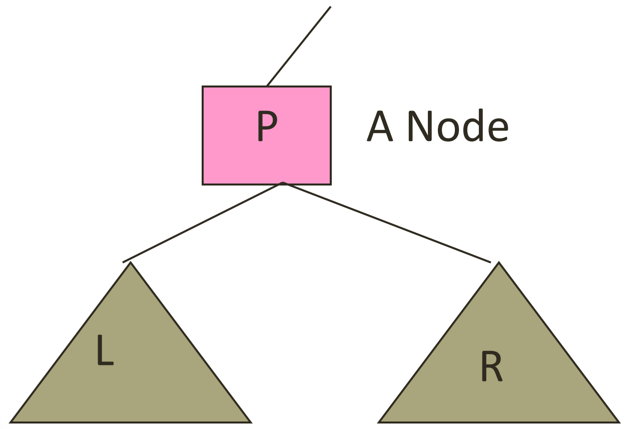
- Anyway, the AVL Balance Property is as follows...
- For ALL nodes, the Height of the Left and Right Sub-trees can only differ by 1.



$$|L.height - R.height| \leq 1$$

AVL Properties

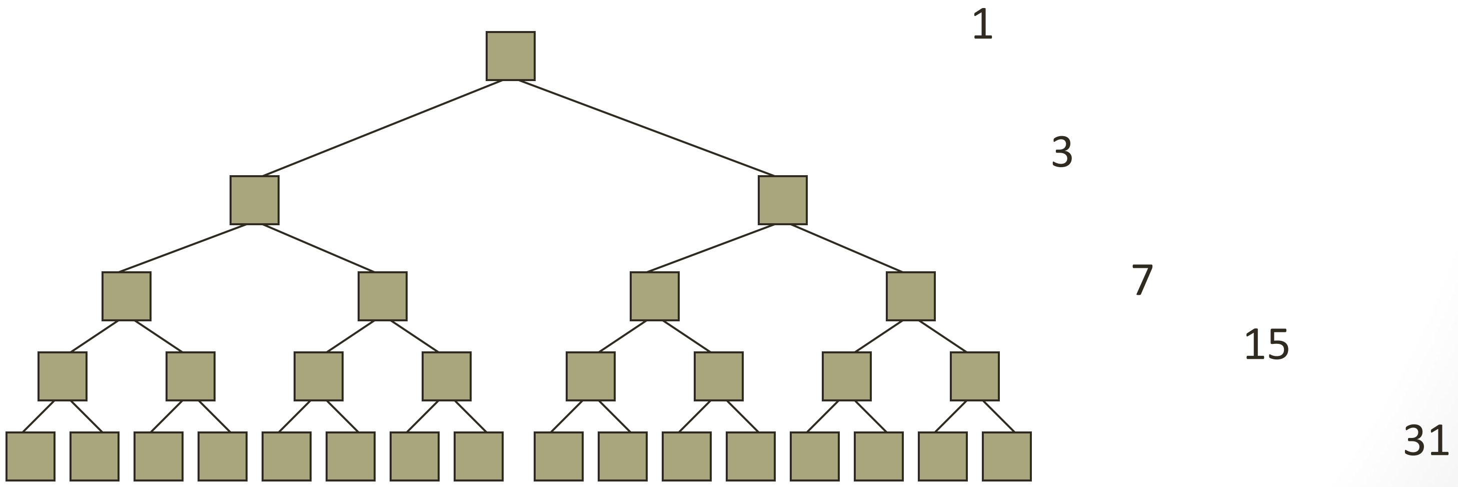
- Wouldn’t this be a better Balance property?
- For ALL nodes, the Height of the Left and Right Sub-trees **must be equal!**



$$|L.height - R.height| = 0$$

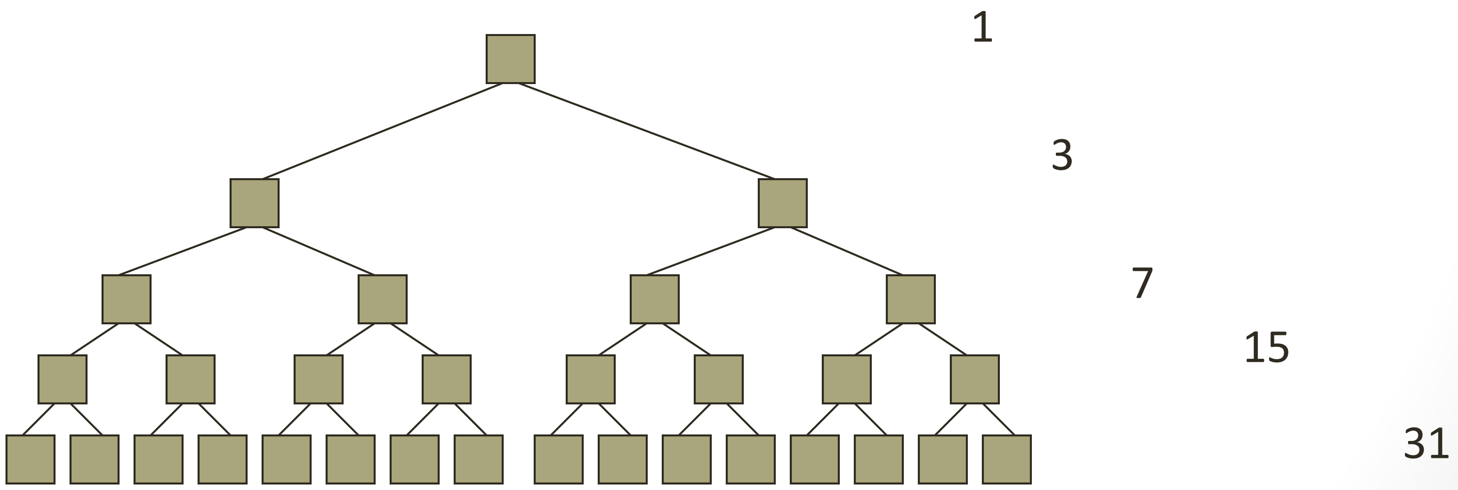
AVL Properties

- For ALL nodes, the Height of the Left and Right Sub-trees **must be equal!**



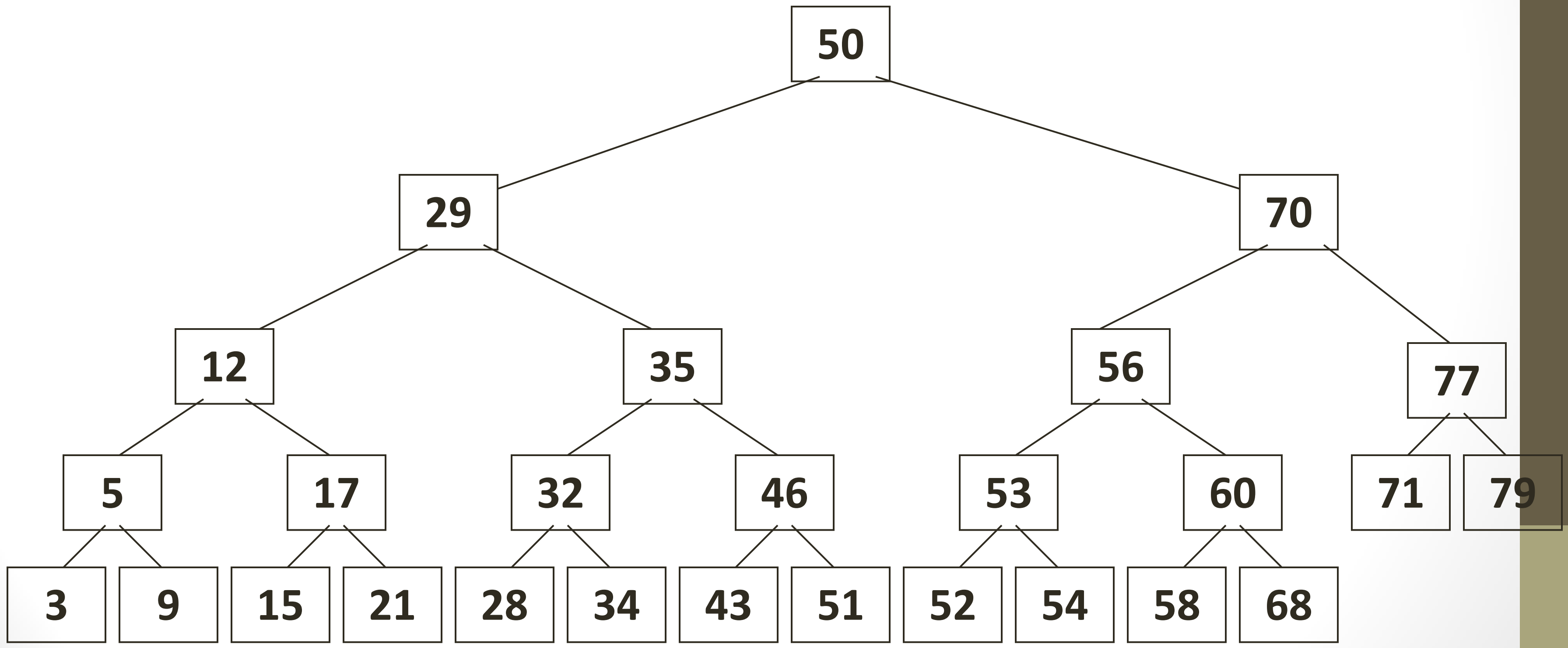
AVL Properties

- For ALL nodes, the Height of the Left and Right Sub-trees **must be equal!**
- **Strict Balance Conditions are too restrictive.**



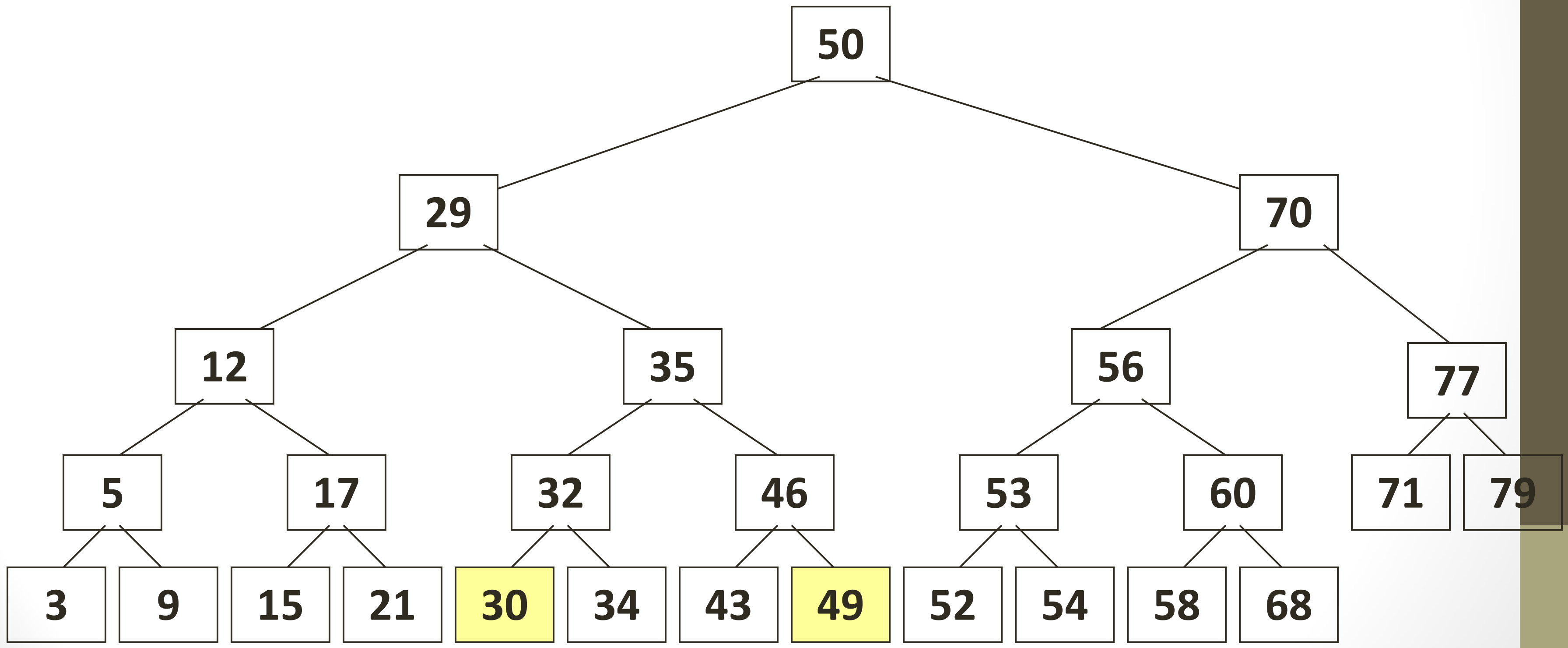
Question?

- Is this an AVL Tree?



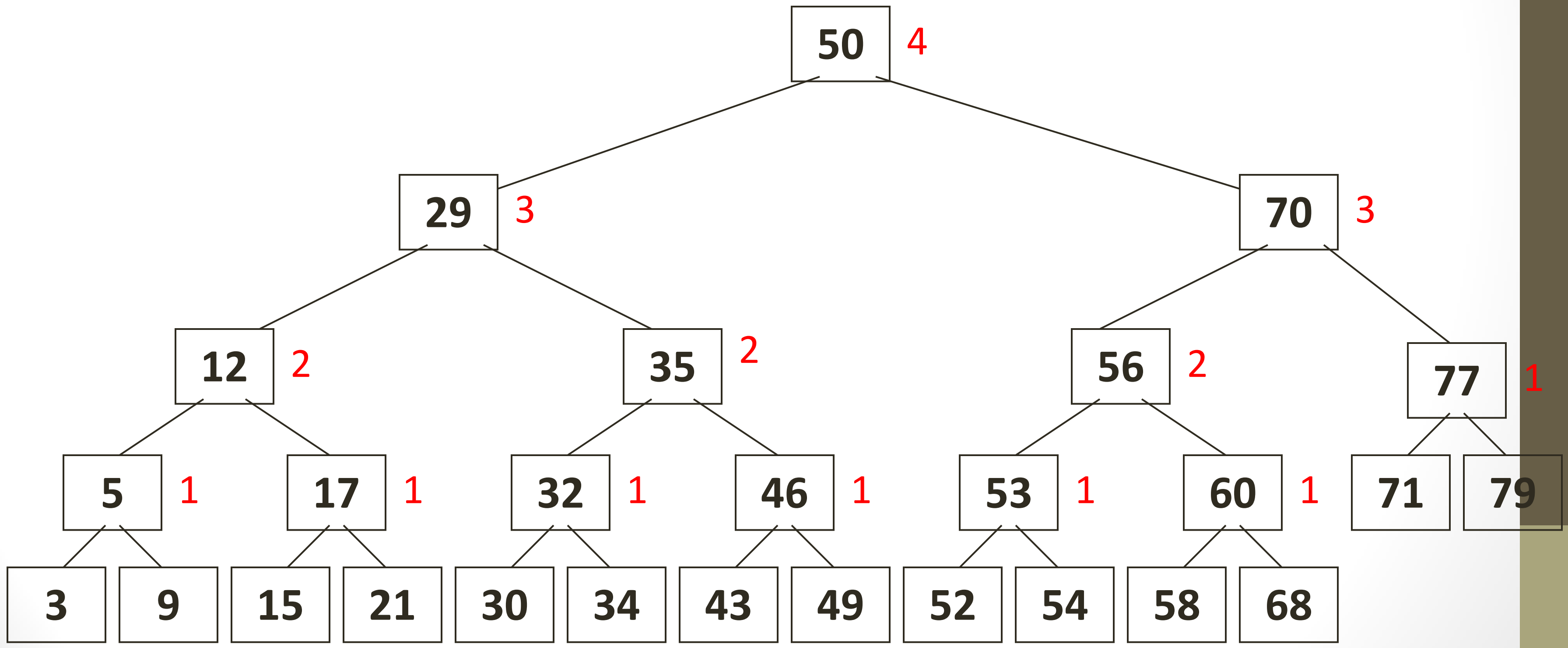
Question?

- Is this an AVL Tree?



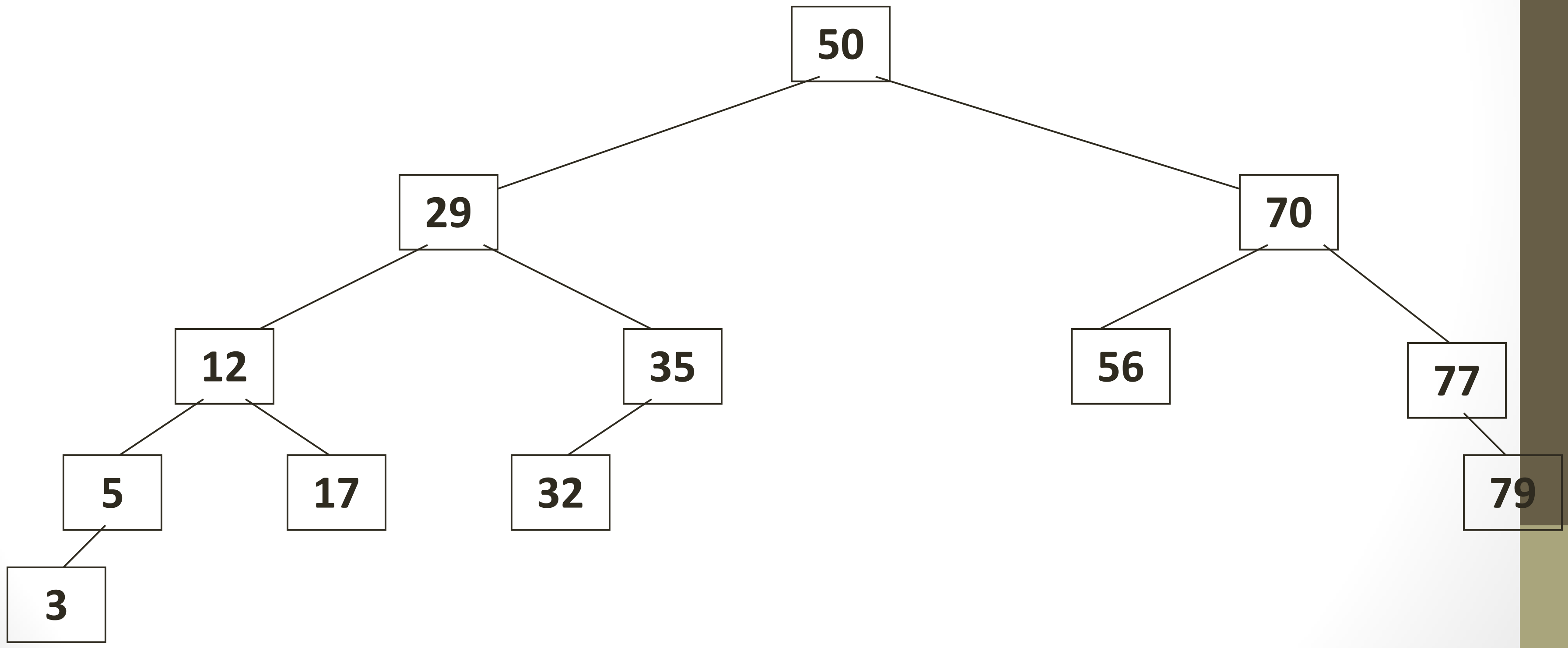
Question?

- Is this an AVL Tree?



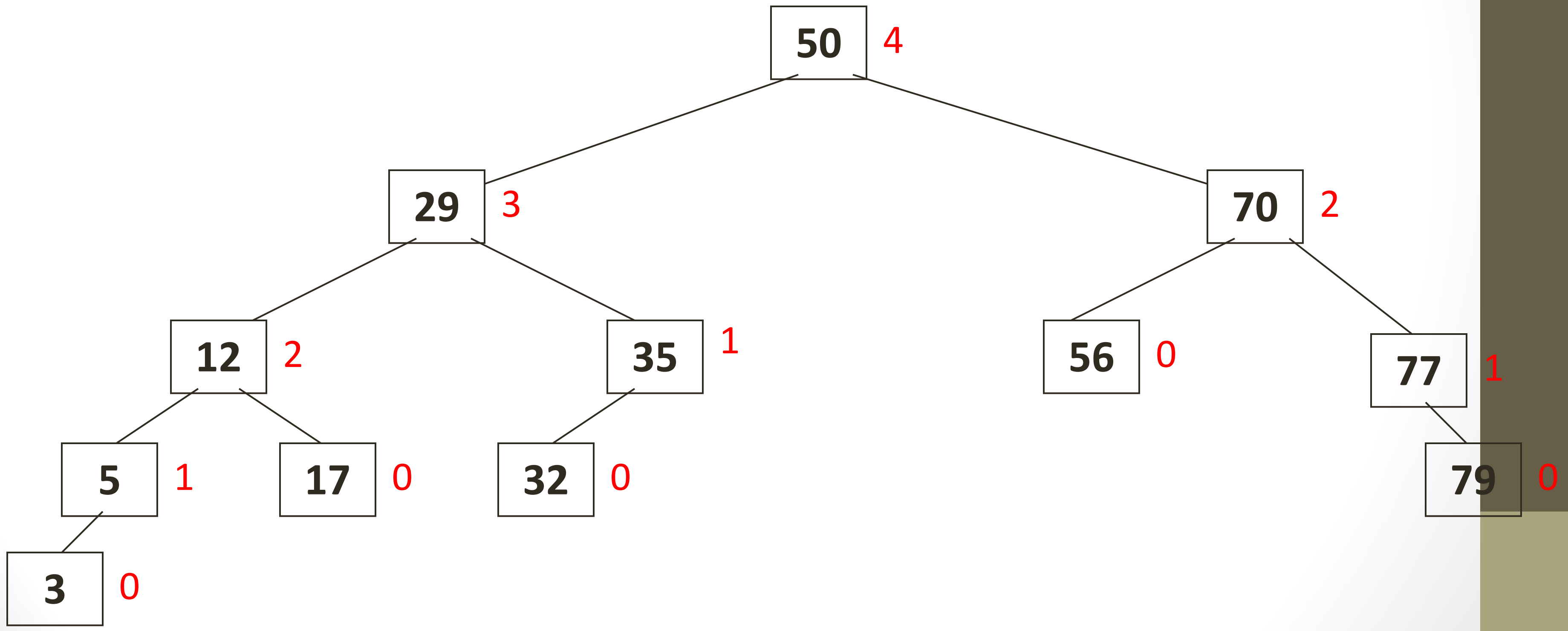
Question?

- Is this an AVL Tree?



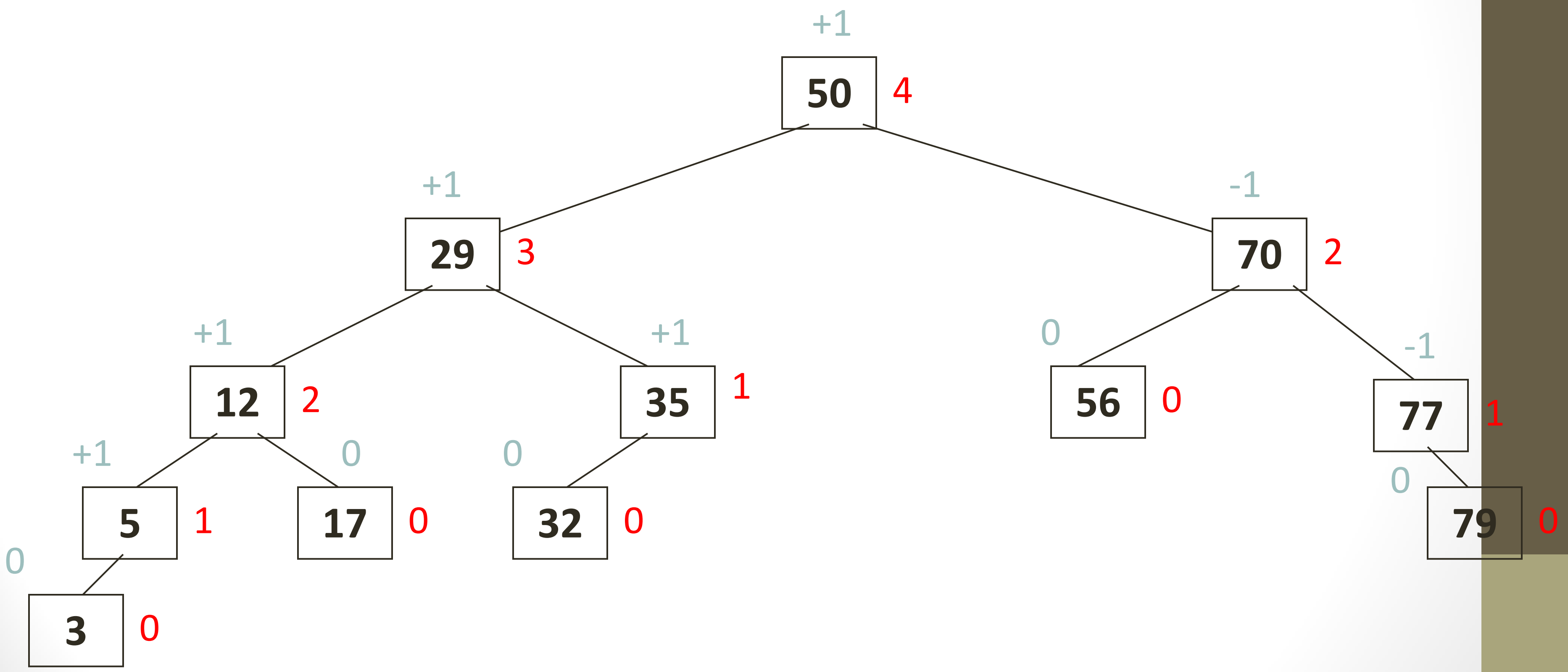
Question?

- Is this an AVL Tree?



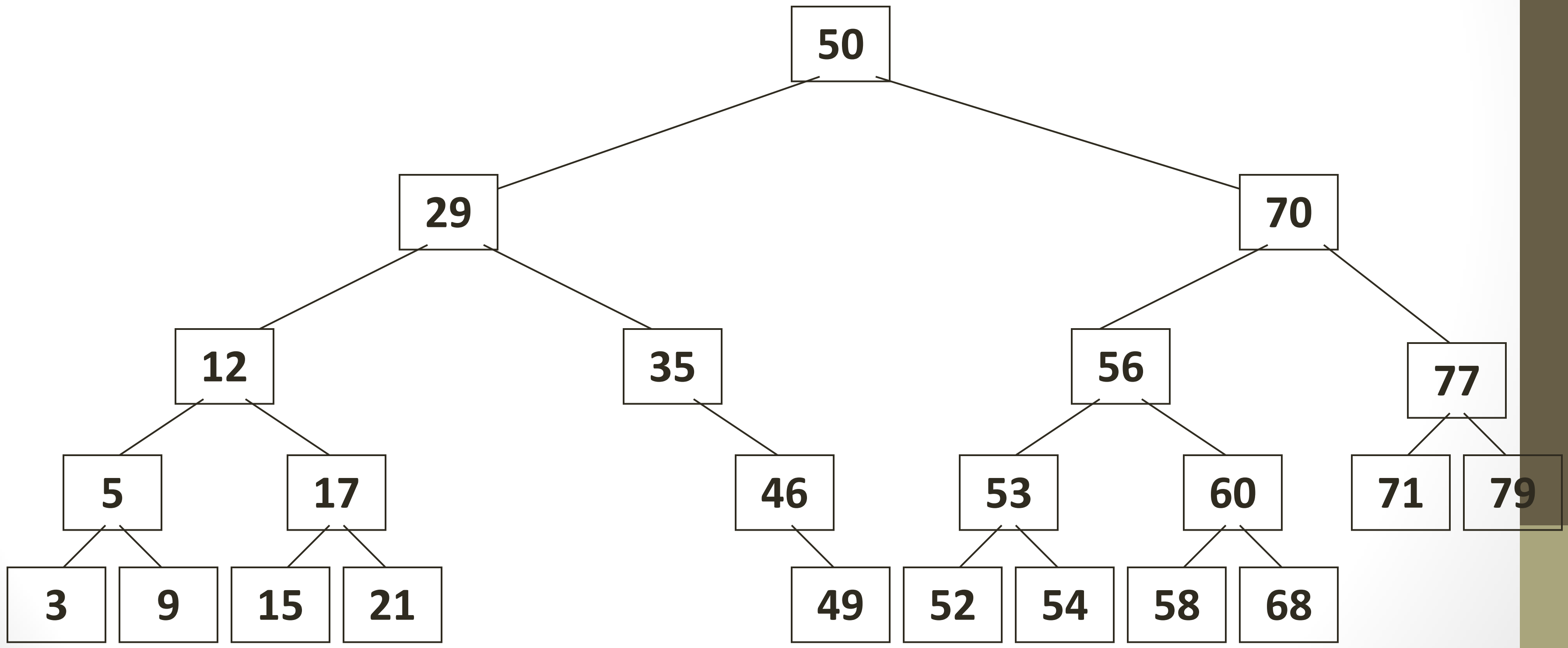
Question?

- Is this an AVL Tree?



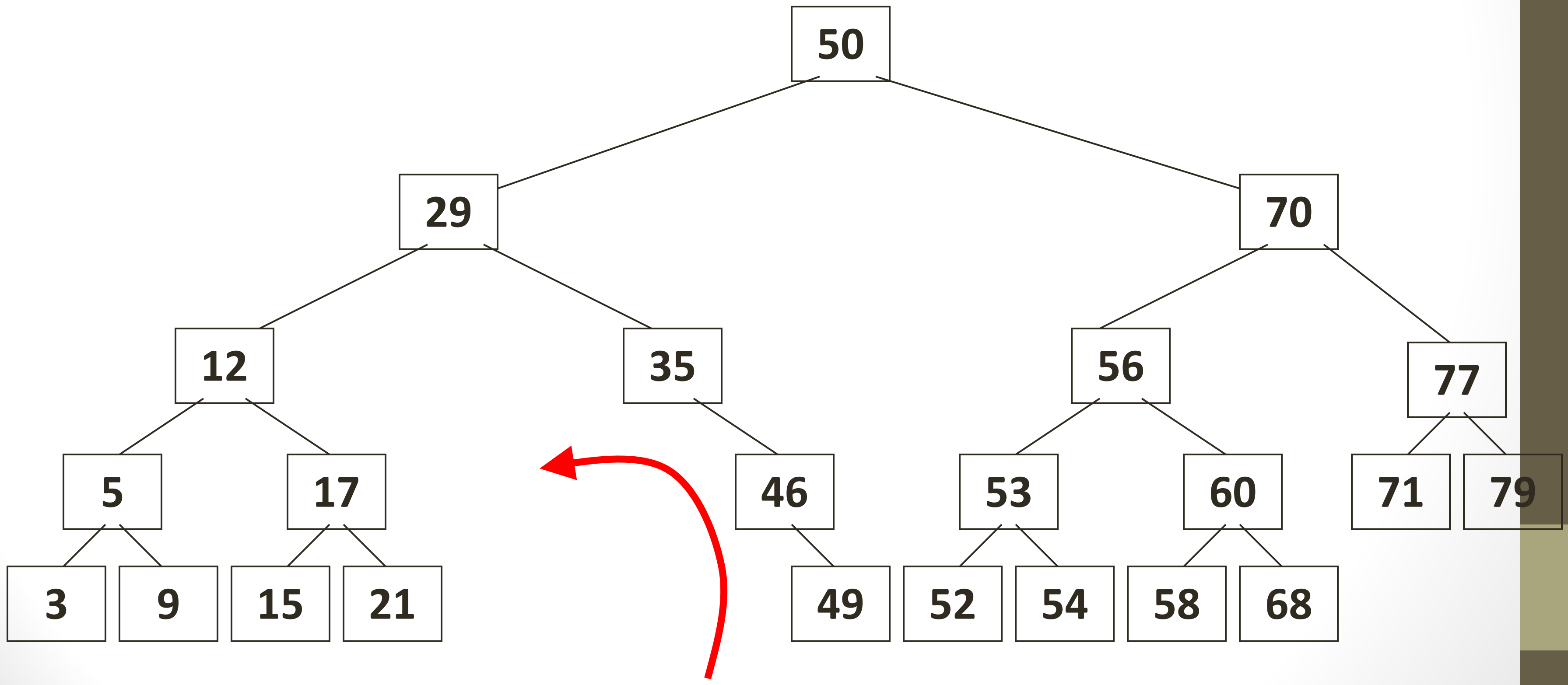
Question?

- Is this an AVL Tree?



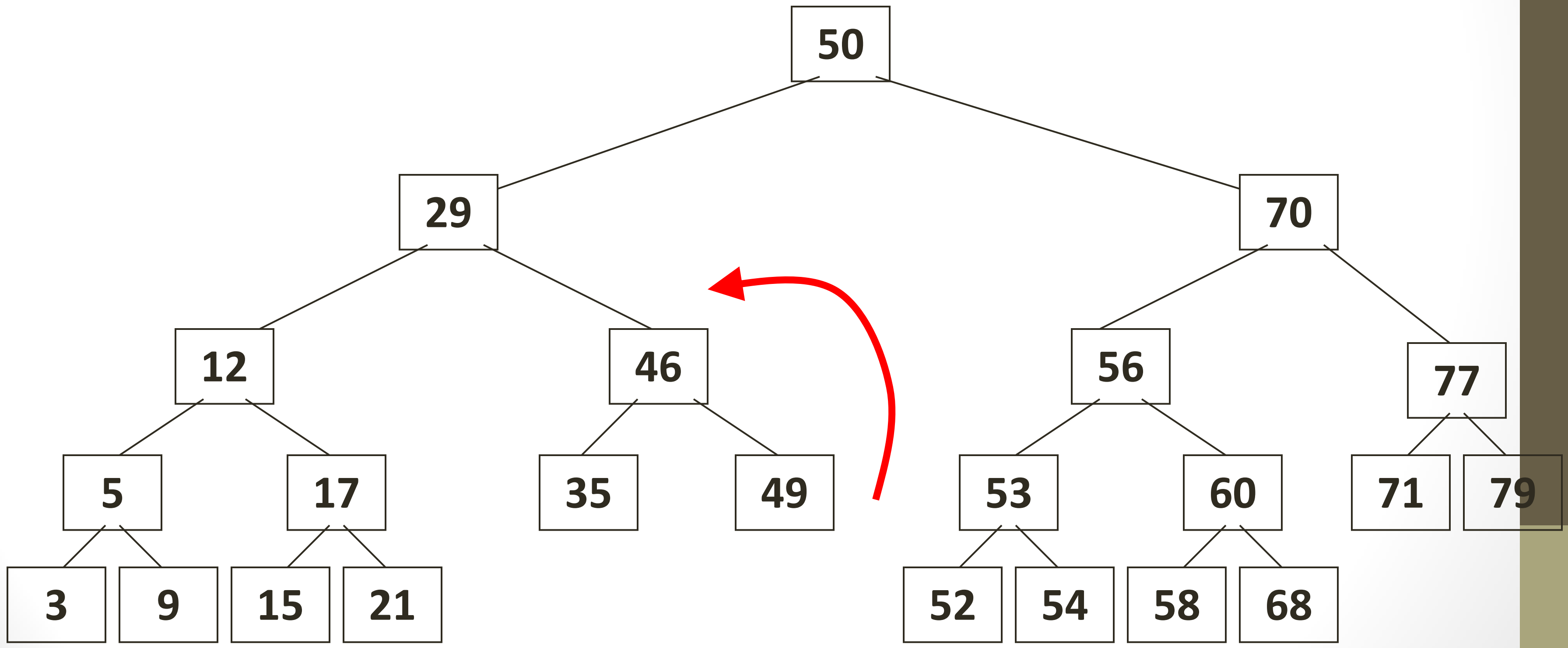
Question?

- No



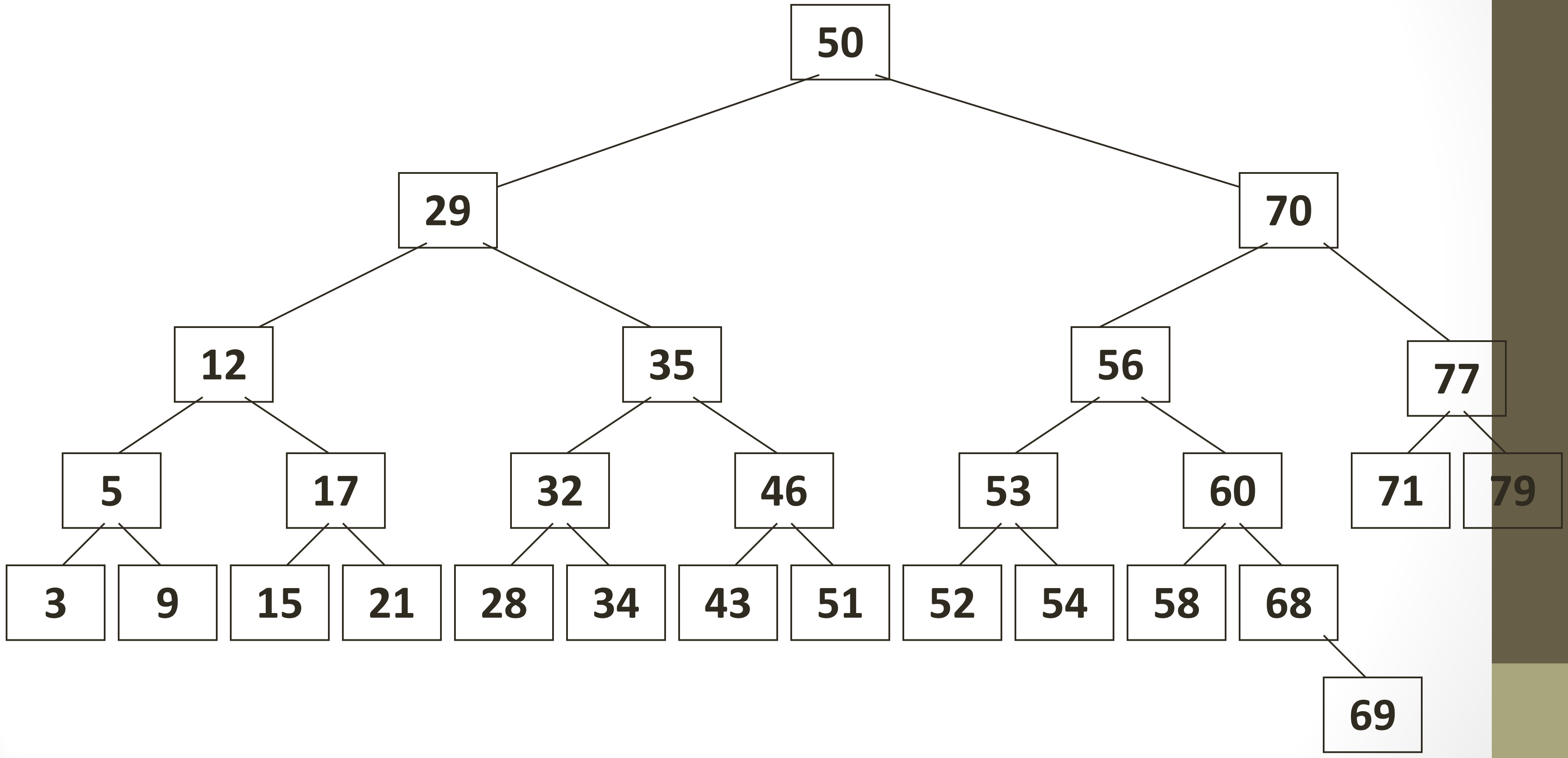
Question?

- Did this fix the problem?



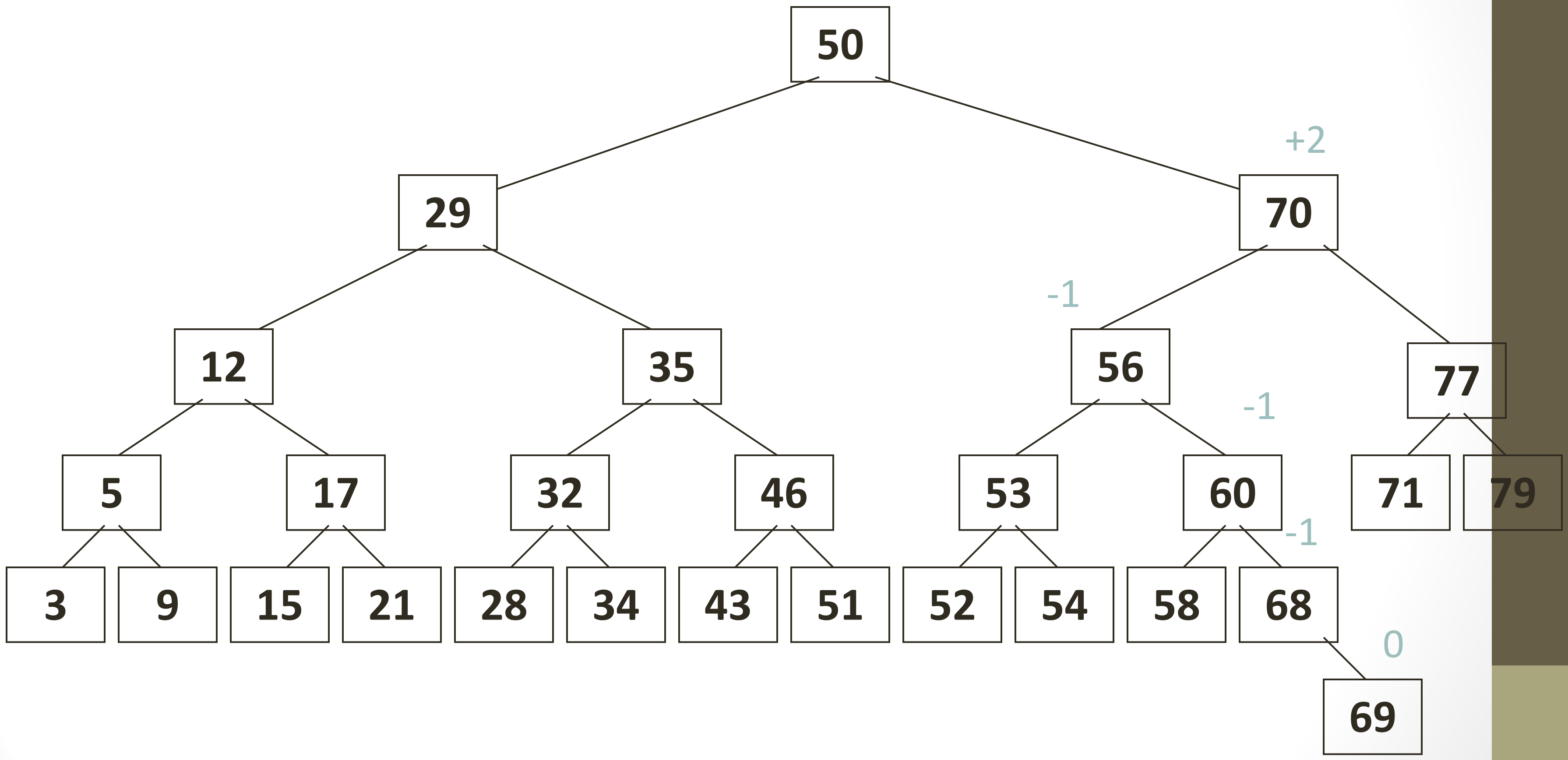
Question?

- Is this an AVL Tree?



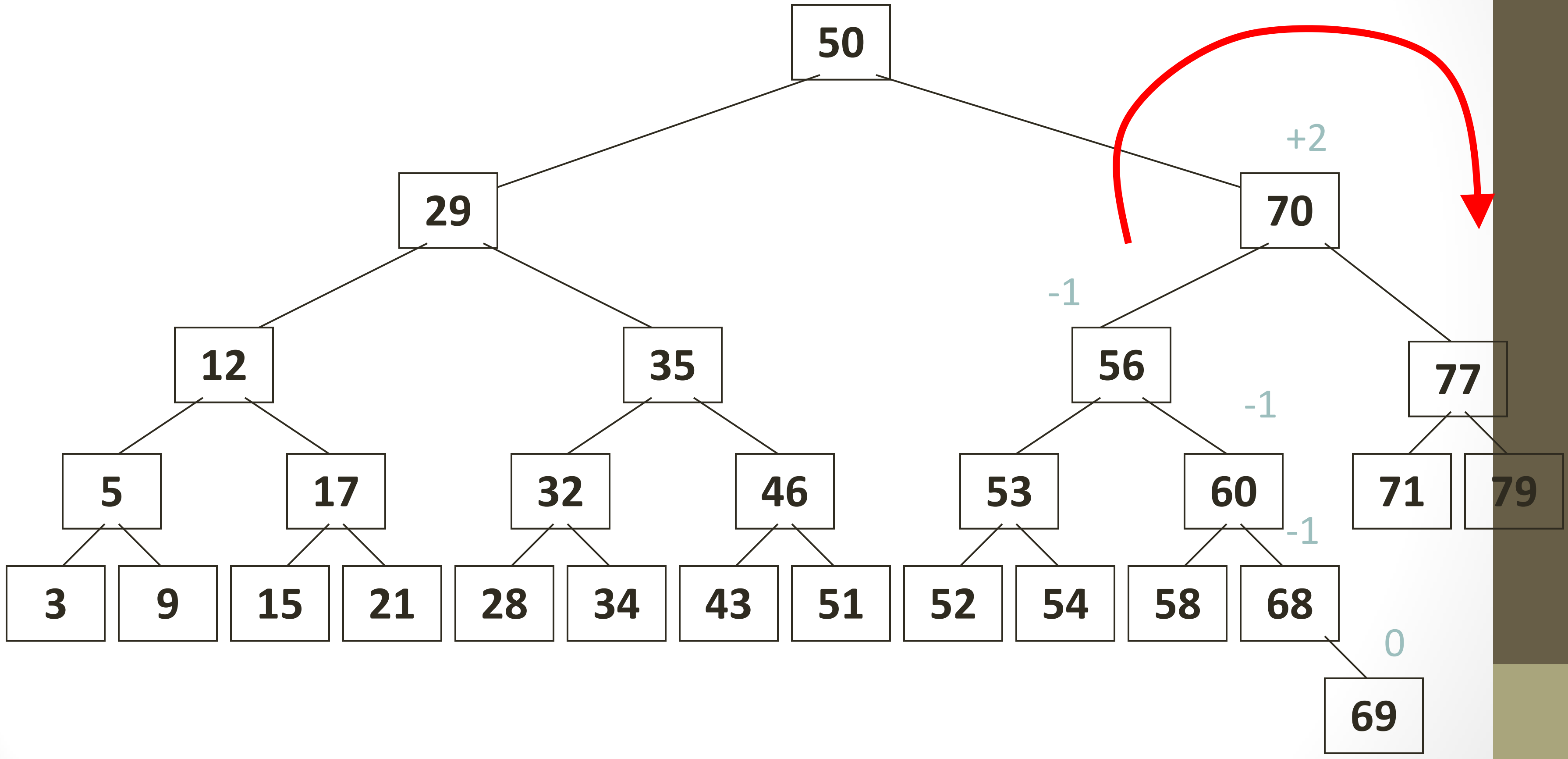
Question?

- Is this an AVL Tree?



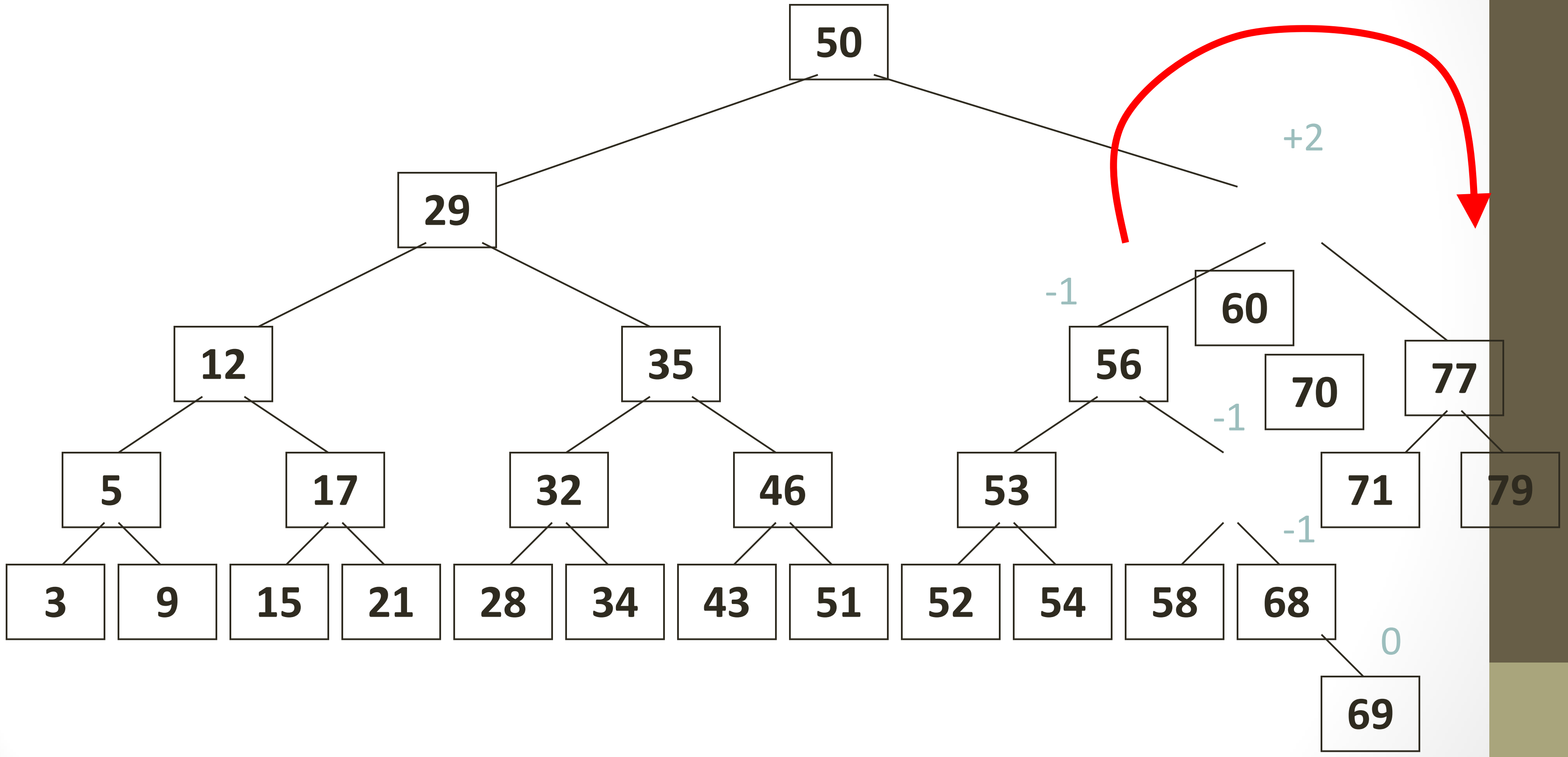
Question?

- Is this an AVL Tree?



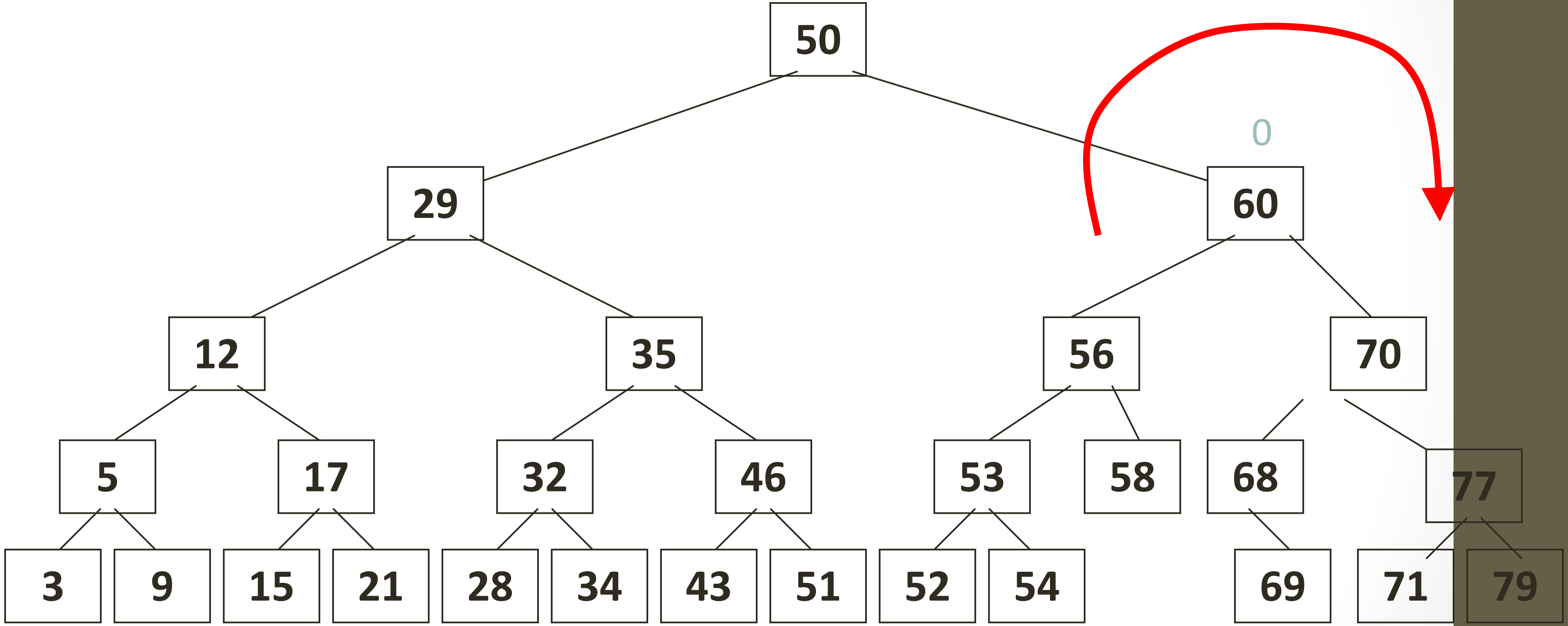
Question?

- Is this an AVL Tree?



Question?

- Is this an AVL Tree?



Correcting Imbalance

1. After every insertion
2. Check to see if an imbalance was created.
    - All you have to do backtrack up the tree
3. If you find an imbalance, correct it.
4. As long as the original tree is an AVL tree, there are only 4 types of imbalances that can occur.

Properties

- The depth of a typical node in an AVL tree is very close to the optimal *log N*.
- Consequently, all searching operations in an AVL tree have logarithmic worst-case bounds.
- An update (insert or remove) in an AVL tree could destroy the balance. It must then be rebalanced before the operation can be considered complete.
- After an insertion, only nodes that are on the path from the insertion point to the root can have their balances altered.

Rebalancing

- Suppose the node to be rebalanced is X. There are 4 cases that we might have to fix (two are the mirror images of the other two):
  1. An insertion in the left subtree of the left child of X,
  2. An insertion in the right subtree of the left child of X,
  3. An insertion in the left subtree of the right child of X, or
  4. An insertion in the right subtree of the right child of X.
- Balance is restored by tree *rotations*.

Balancing Operations: Rotations

- Case 1 and case 4 are symmetric and requires the same operation for balance.
  - Cases 1,4 are handled by *single rotation.*
- Case 2 and case 3 are symmetric and requires the same operation for balance.
  - Cases 2,3 are handled by *double rotation.*

Single Rotation

- A single rotation switches the roles of the parent and child while maintaining the search order.
- Single rotation handles the outside cases (i.e. 1 and 4).
- We rotate between a node and its child.
  - Child becomes parent. Parent becomes right child in case 1, left child in case 4.
- The result is a binary search tree that satisfies the AVL property.

Figure 19.23

Single rotation to fix case 1: Rotate right

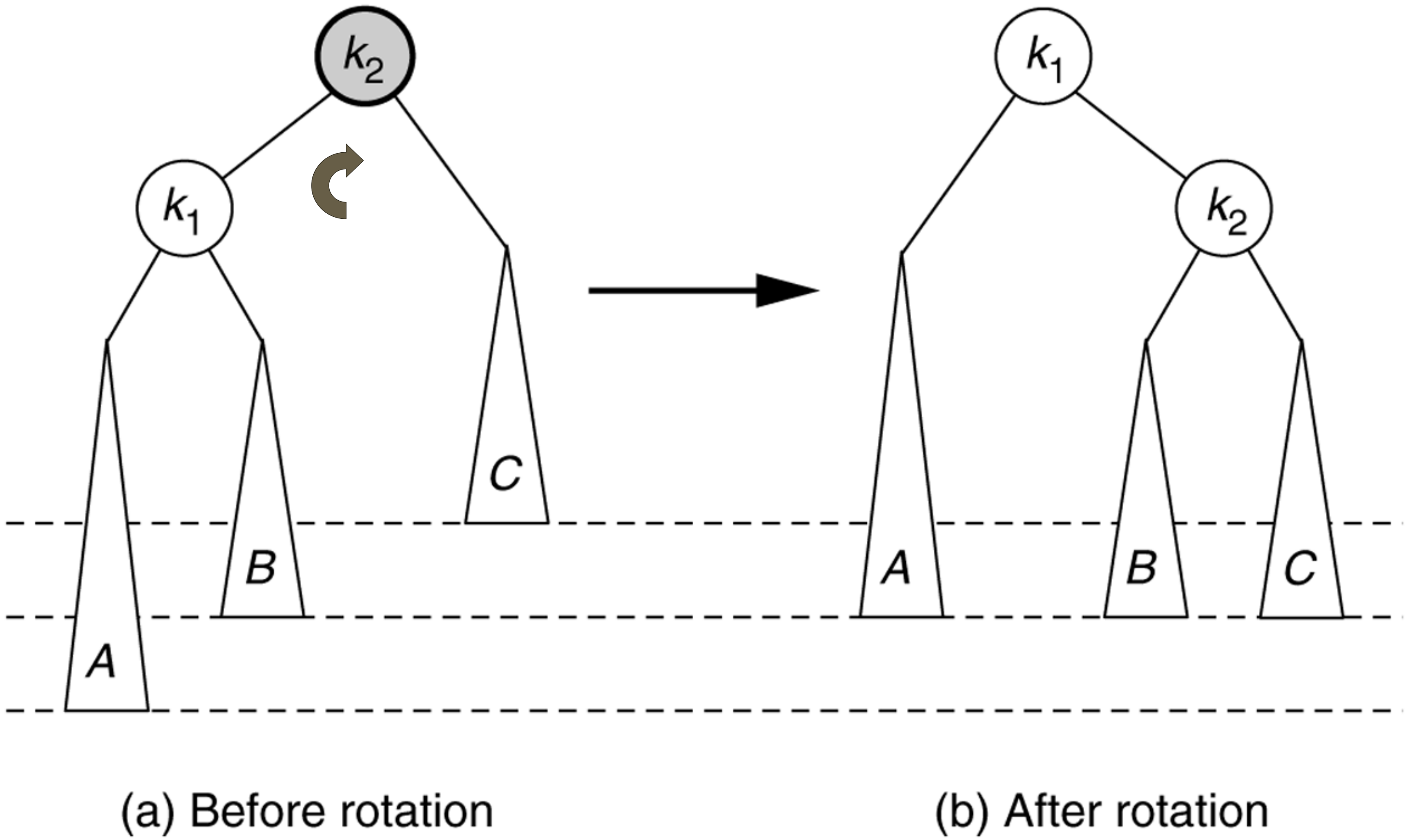


Figure 19.26

Symmetric single rotation to fix case 4 : Rotate left

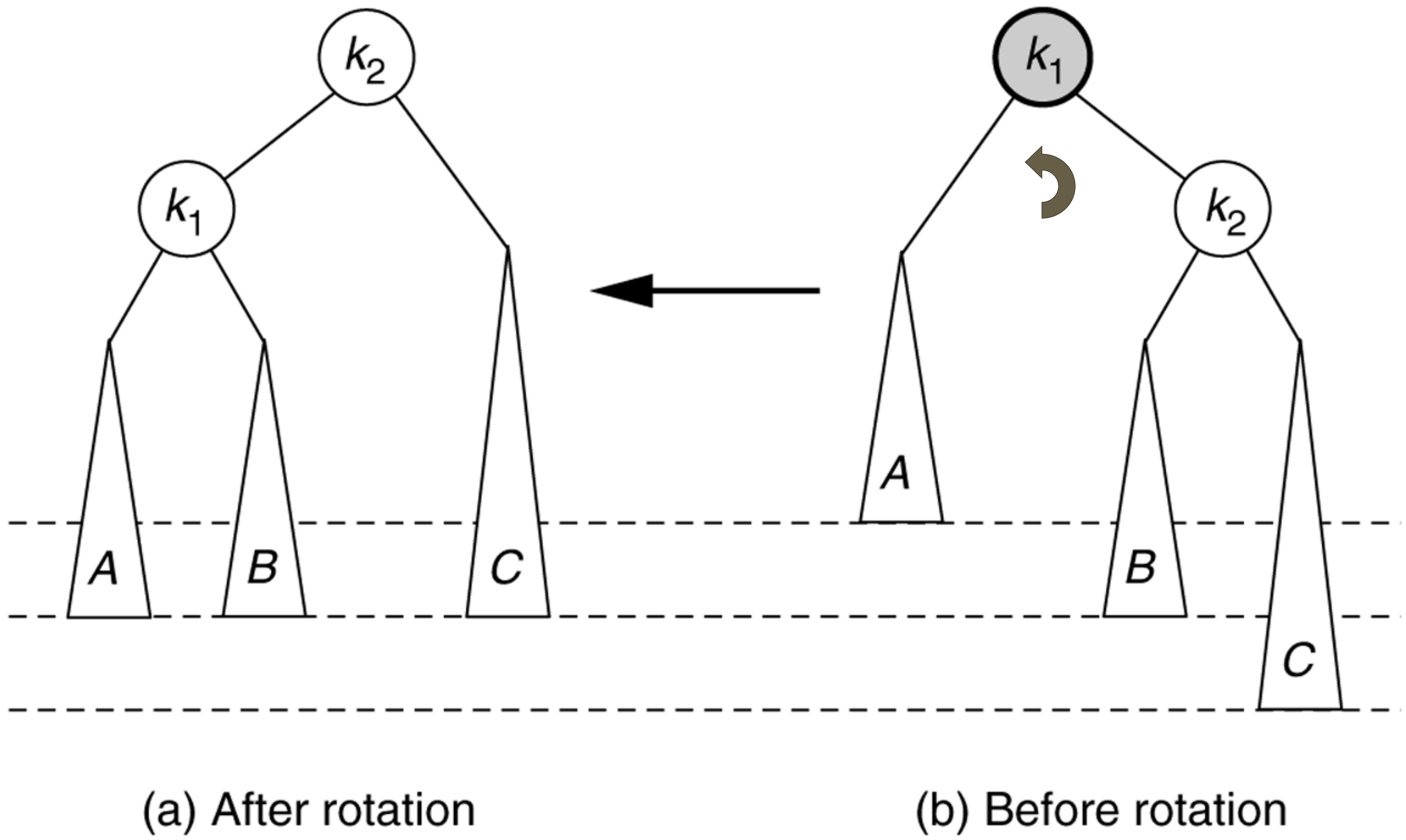
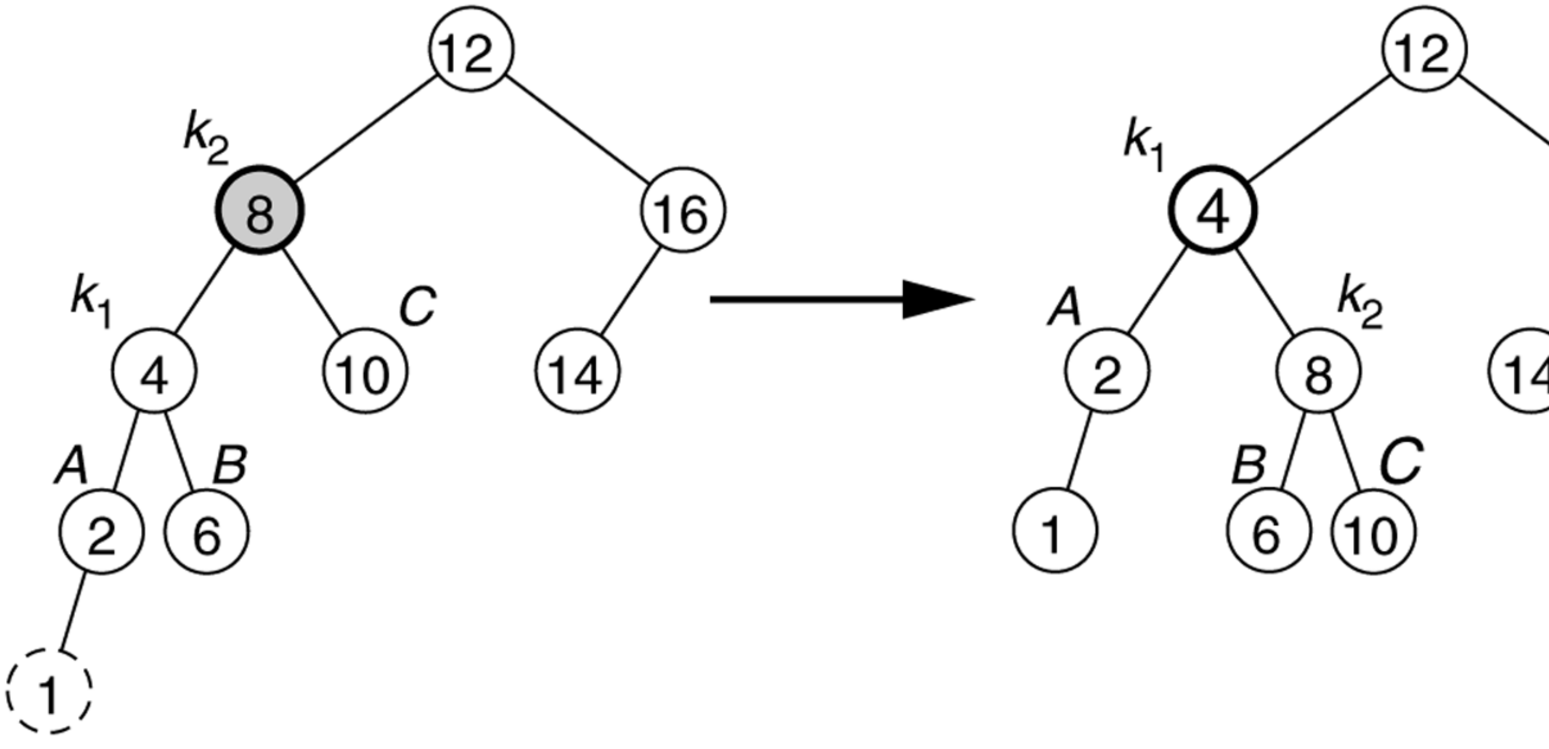


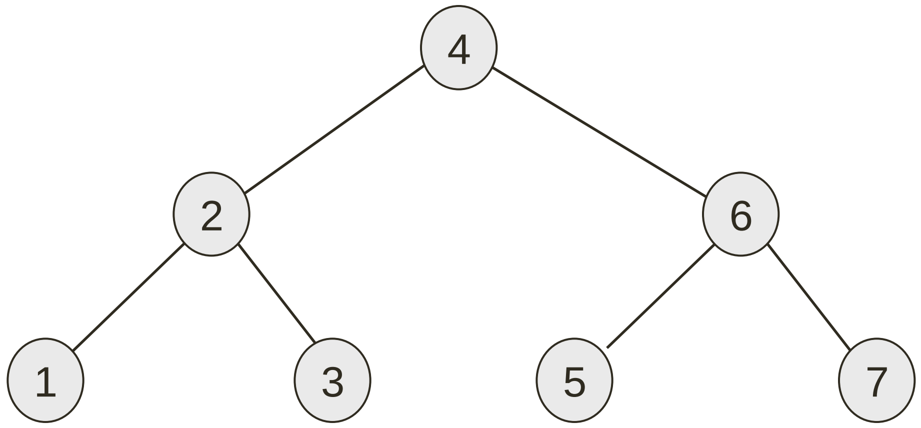
Figure 19.25
Single rotation fixes an AVL tree after insertion of 1.



(a) Before rotation (b) After rotation

Example

- Start with an empty AVL tree and insert the items 3,2,1, and then 4 through 7 in sequential order.
- Answer:



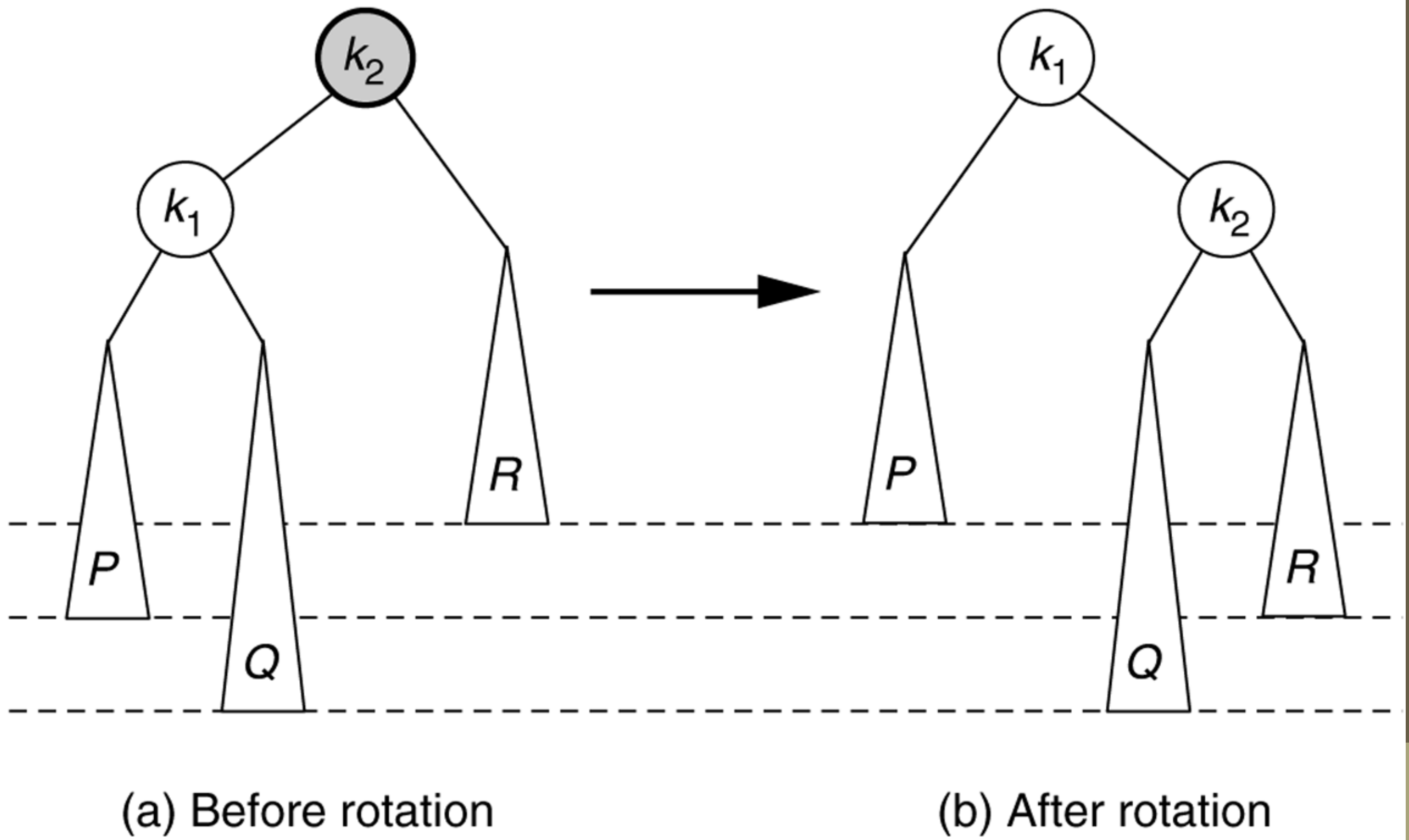
Analysis

- One rotation suffices to fix cases 1 and 4.
- Single rotation preserves the original height:
  - The new height of the entire subtree is exactly the same as the height of the original subtree before the insertion.
- Therefore it is enough to do rotation only at the first node, where imbalance exists, on the path from inserted node to root.
- Thus the rotation takes O(1) time.
- Hence insertion is O(logN)

Double Rotation

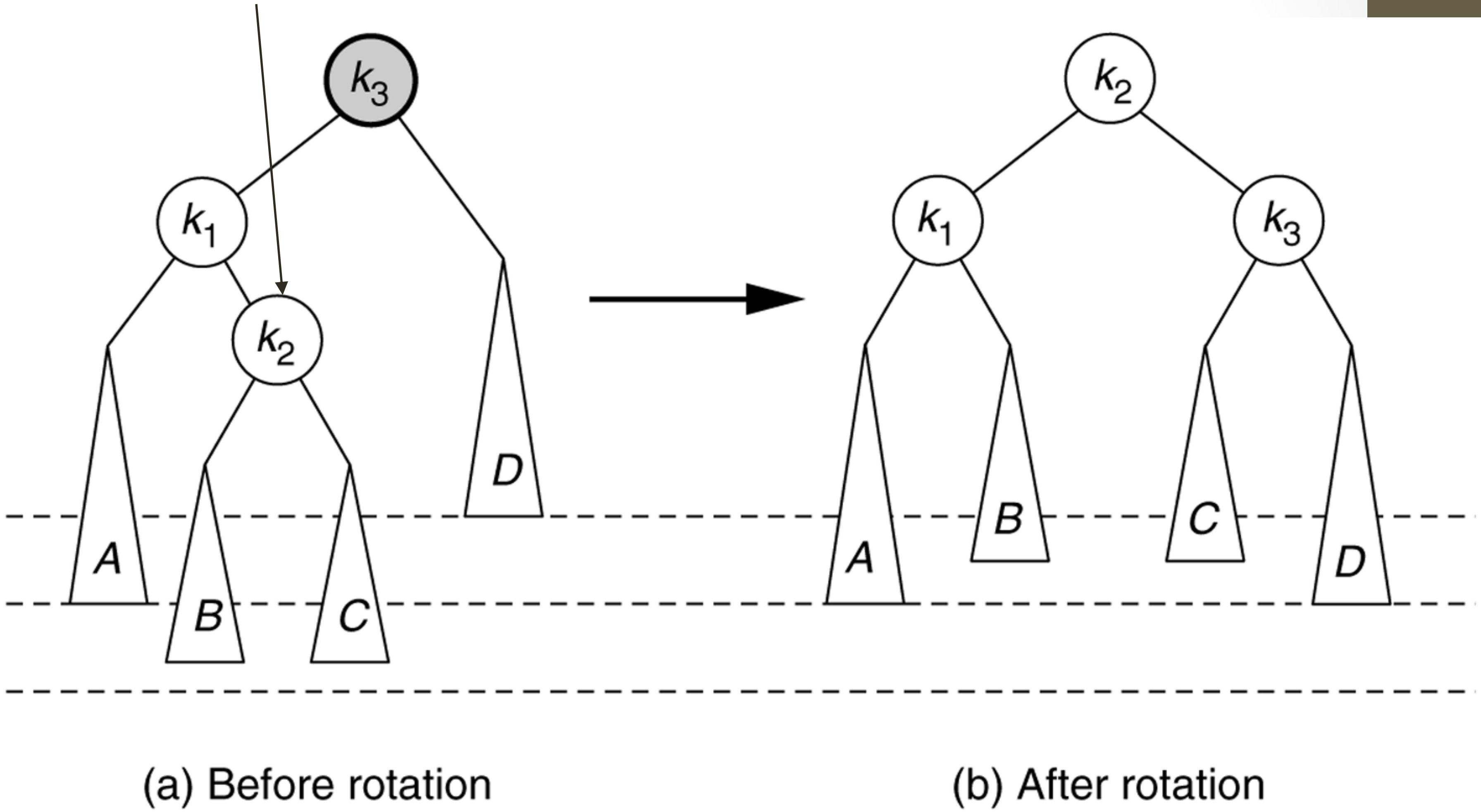
- Single rotation does not fix the inside cases (2 and 3).
- These cases require a *double* rotation, involving three nodes and four subtrees.

Figure 19.28
Single rotation **does not** fix case 2.



Left–right double rotation to fix case 2

Lift this up:
first rotate left between (k_1,k_2),
then rotate right betwen (k_3,k_2)

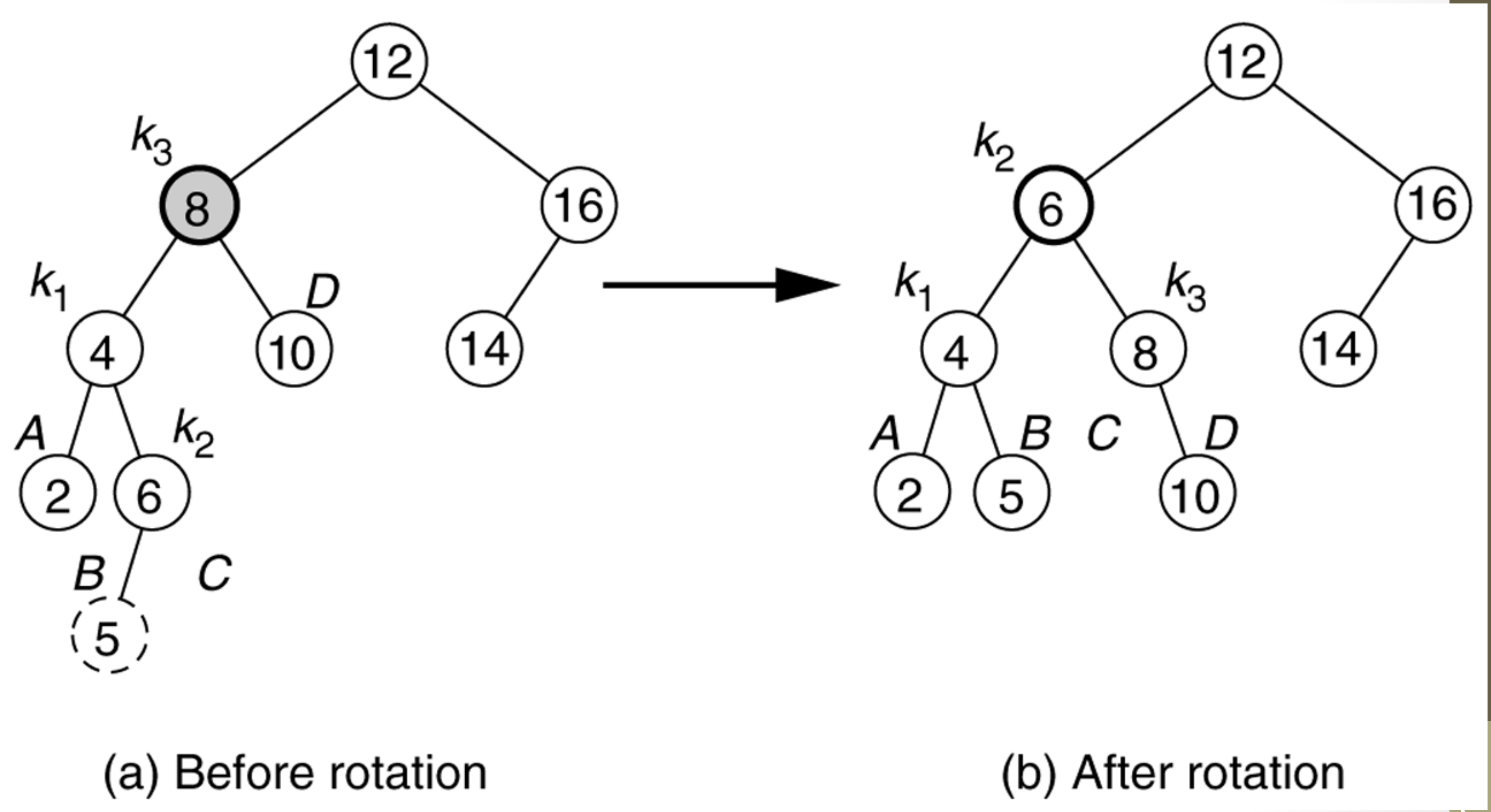


Left-Right Double Rotation

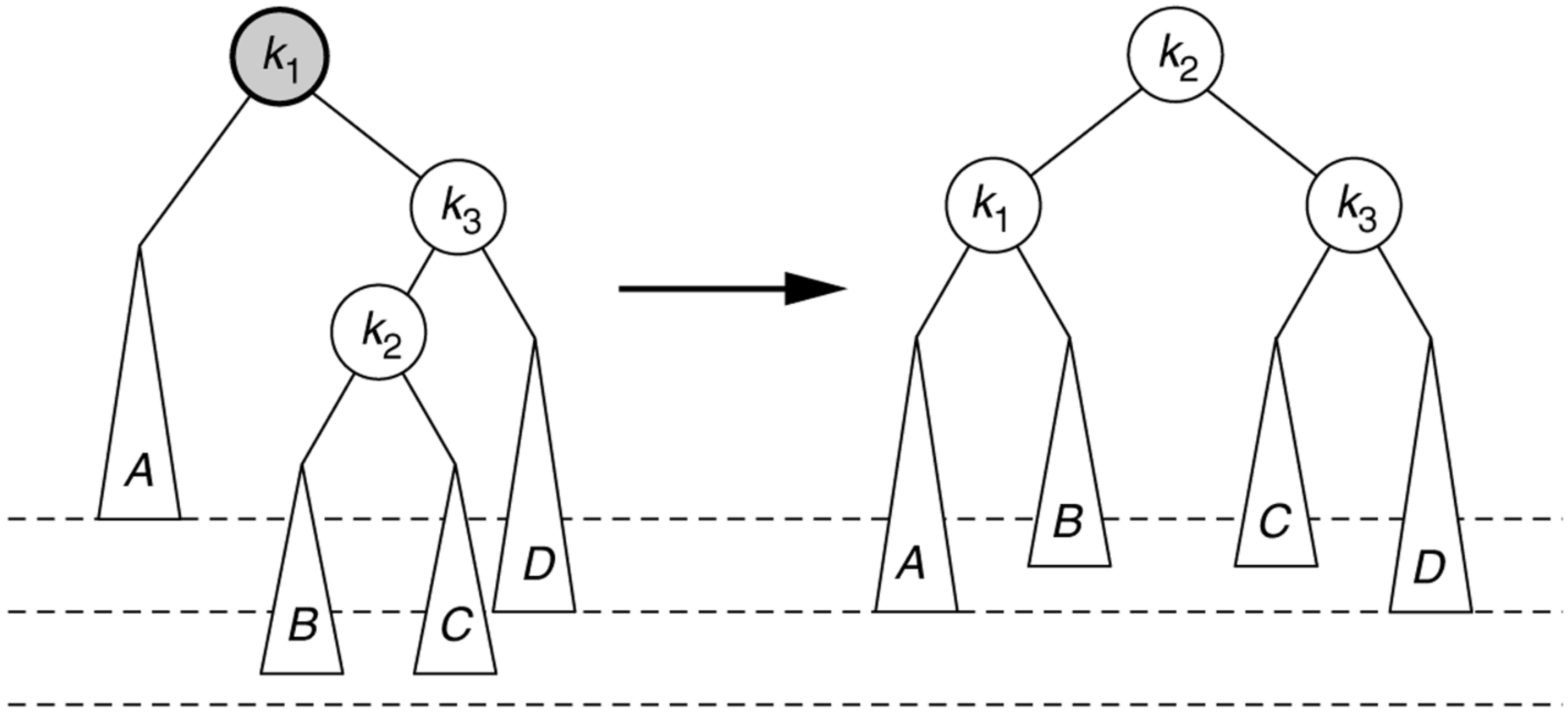
- A left-right double rotation is equivalent to a sequence of two single rotations:
  - 1st rotation on the original tree:
    a *left* rotation between X's left-child and grandchild
  - 2nd rotation on the new tree:
    a *right* rotation between X and its new left child.

Figure 19.30

Double rotation fixes AVL tree after the insertion of 5.



Right–Left double rotation to fix case 3.



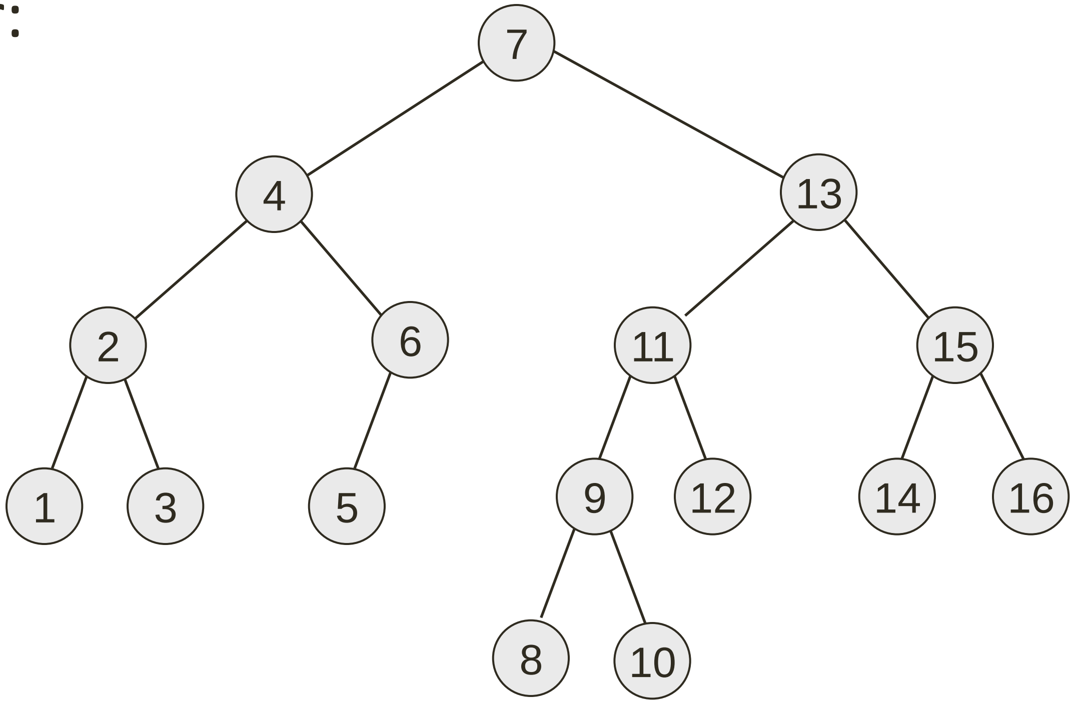
(a) Before rotation

(b) After rotation

Example

Insert 16, 15, 14, 13, 12, 11, 10, and 8, and 9 to the previous tree obtained in the previous single rotation example.

Answer:


7
4
13
2
6
11
15
1
3
5
9
12
14
16
8
10