AVL Tree

DEFINITION:

AVL Tree is:

- A Binary Search Tree,
- in which the heights of the left and right subtrees of the root differ by at most 1, and
- the left and right subtrees are again AVL trees.

AVL Tree

The name comes from the discoverers of this method, G.M.Adel'son-Vel'skii and E.M.Landis.

The method dates from 1962.

Balance factor

Balance factor:

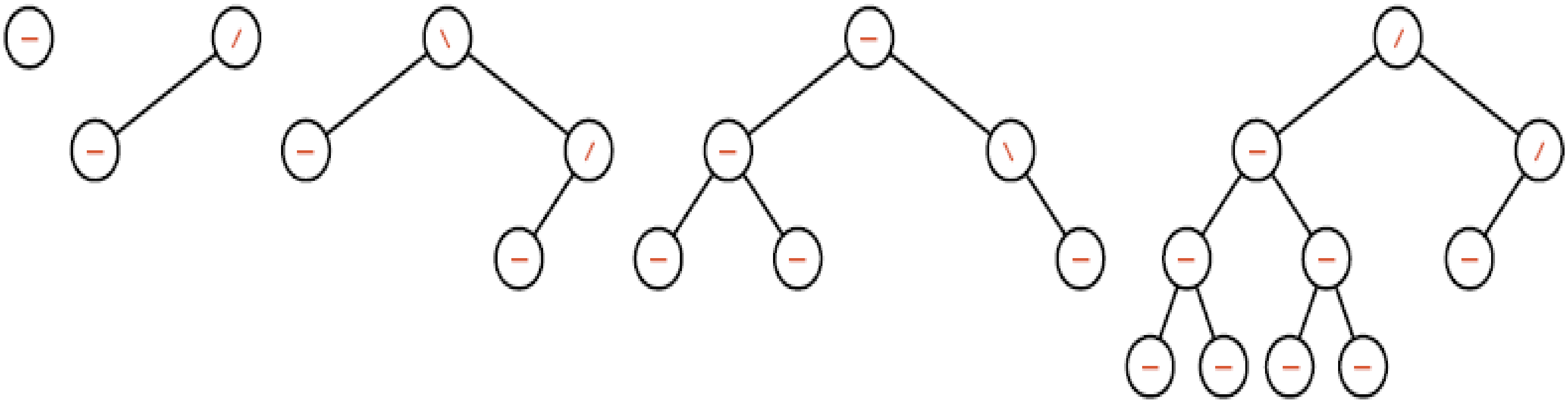
- left_higher: $H_L = H_R + 1$
- equal_height: $H_L = H_R$
- right_higher: $H_R = H_L + 1$

(H_L , H_R : the height of left and right subtree)

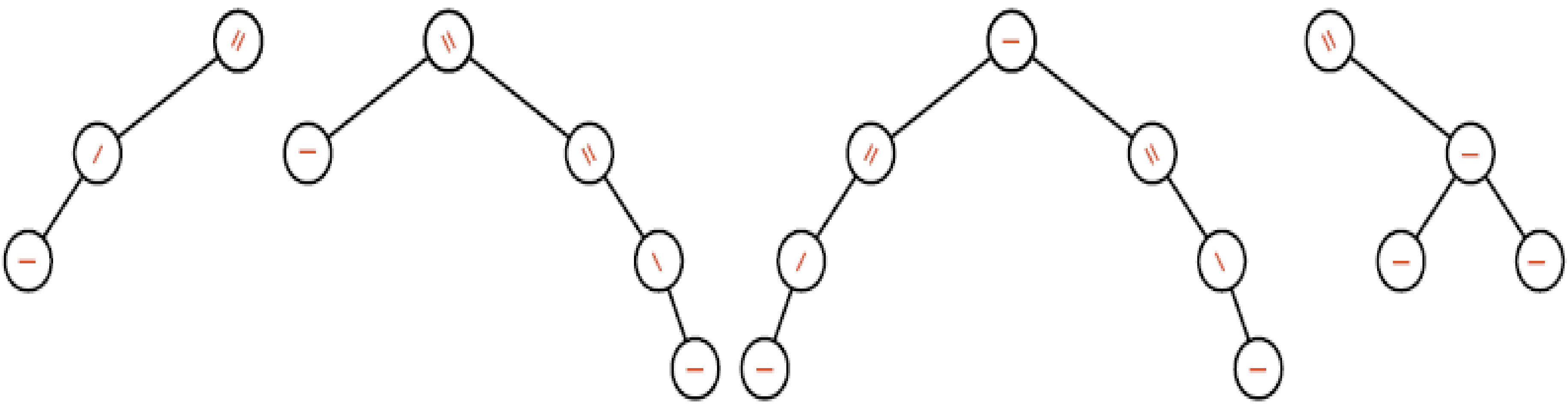
In C++:

enum Balance_factor {*left_higher, equal_height, right_higher*};

AVL Trees and non-AVL Trees



AVL trees



non-AVL trees

Linked AVL Tree

AVL_Node

data <DataType>

left <pointer>

right <pointer>

balance <Balance_factor>

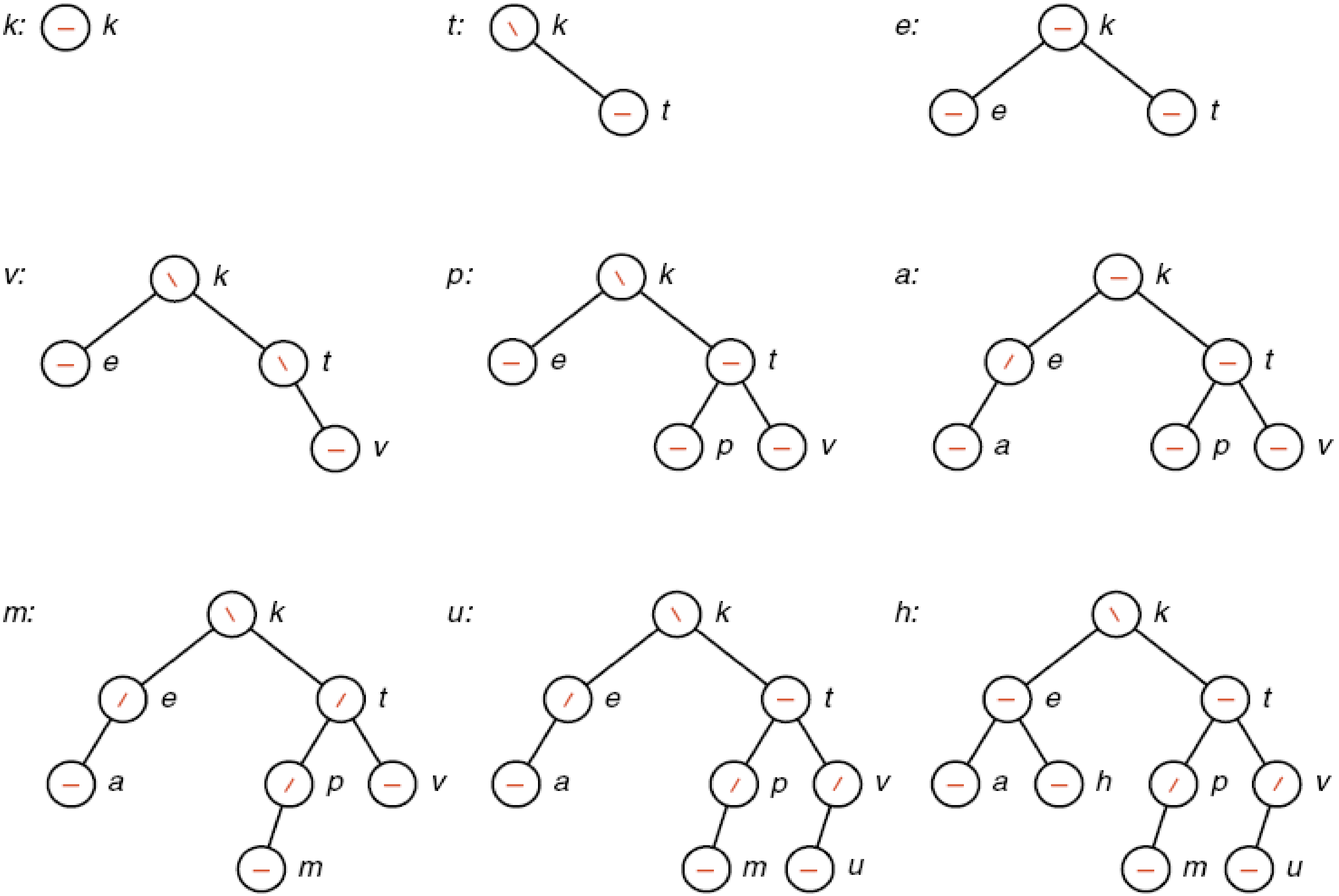
End AVL_Node

AVL_Tree

root <pointer>

End AVL_Tree

Insertion into an AVL tree



Insertion into an AVL tree

- Follow the usual BST insertion algorithm: insert the new node into the empty left or right subtree of a parent node as appropriate.

- We use a reference parameter *taller* of the recursive_Insert function to show if the height of a subtree, for which the recursive function is called, has been increased.

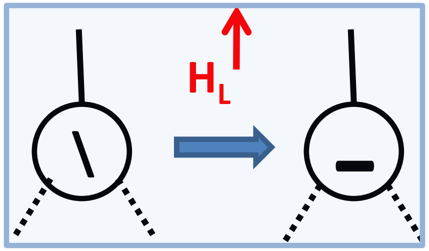
- At the stopping case of recursive, the empty subtree becomes a tree with one node for new data, taller is set to TRUE.

Insertion into an AVL tree

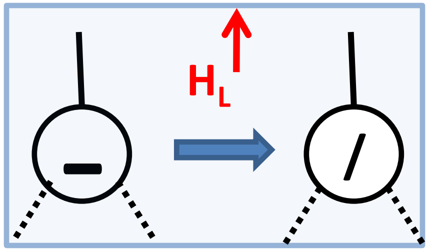
Consider the subtree, for which the recursive function is called,

- While *taller* is TRUE, for each node on the path from the subtree's parent to the root of the tree, do the following steps.

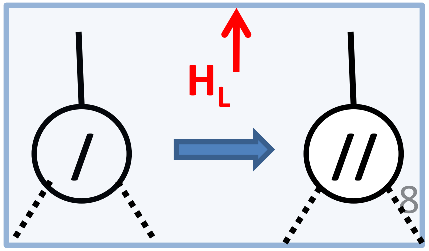
  a) If the subtree was the shorter: its parent's balance factor must be changed, but the height of parent tree is unchanged.
  taller becomes FALSE.



  b) If two subtree had the same height, its parent's balance factor must be changed, the height of parent tree increases by 1.
  taller remains TRUE.



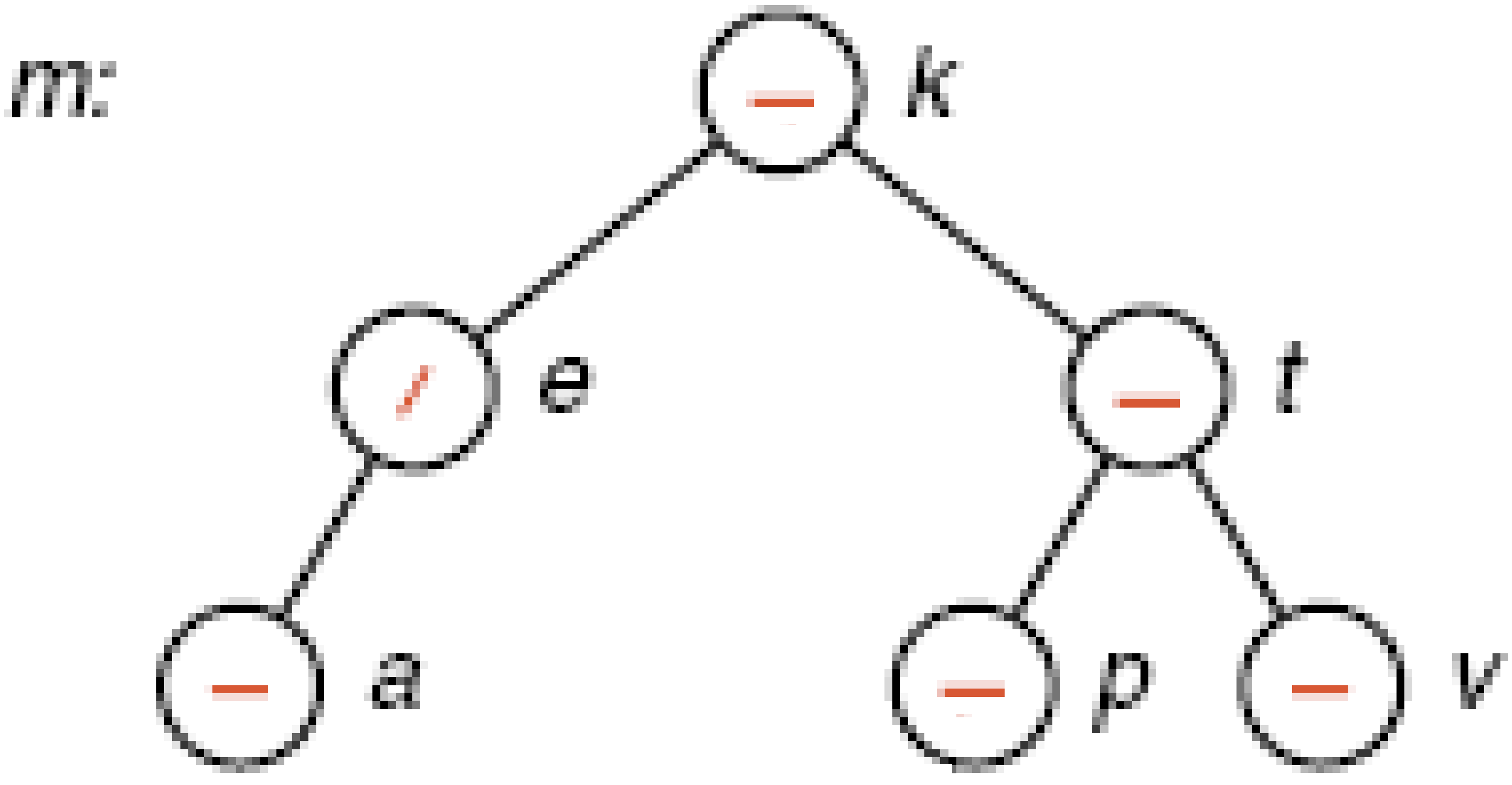
  c) If the subtree was the higher subtree: only in this case, the definition of AVL is violated at the parent node, rebalancing must be done.
  taller becomes FALSE



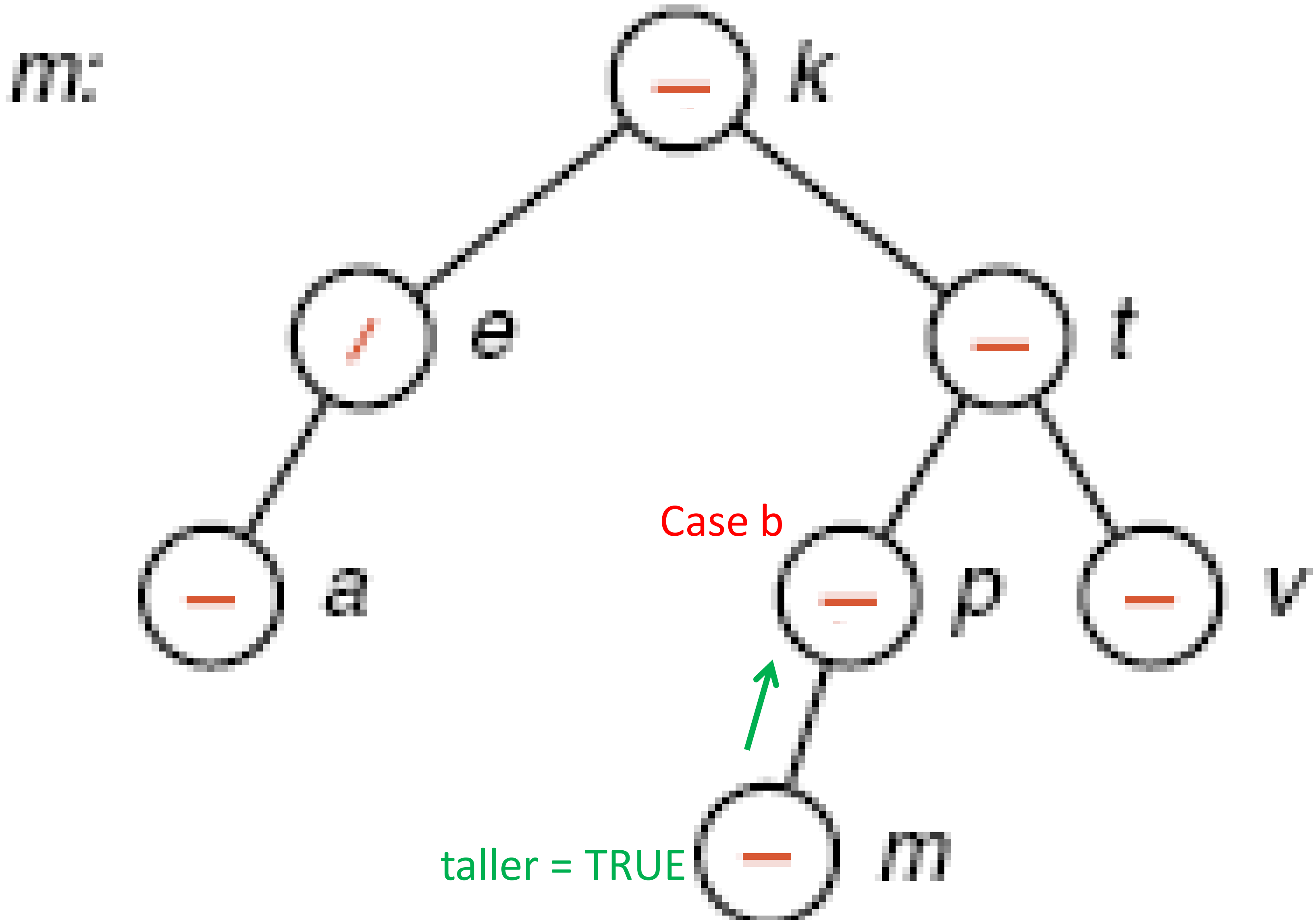
Insertion into an AVL tree

- When *taller* becomes FALSE, the algorithm terminates.
- When rebalancing must be done, the height of the subtree always returned to its original value, so *taller* always becomes FALSE!

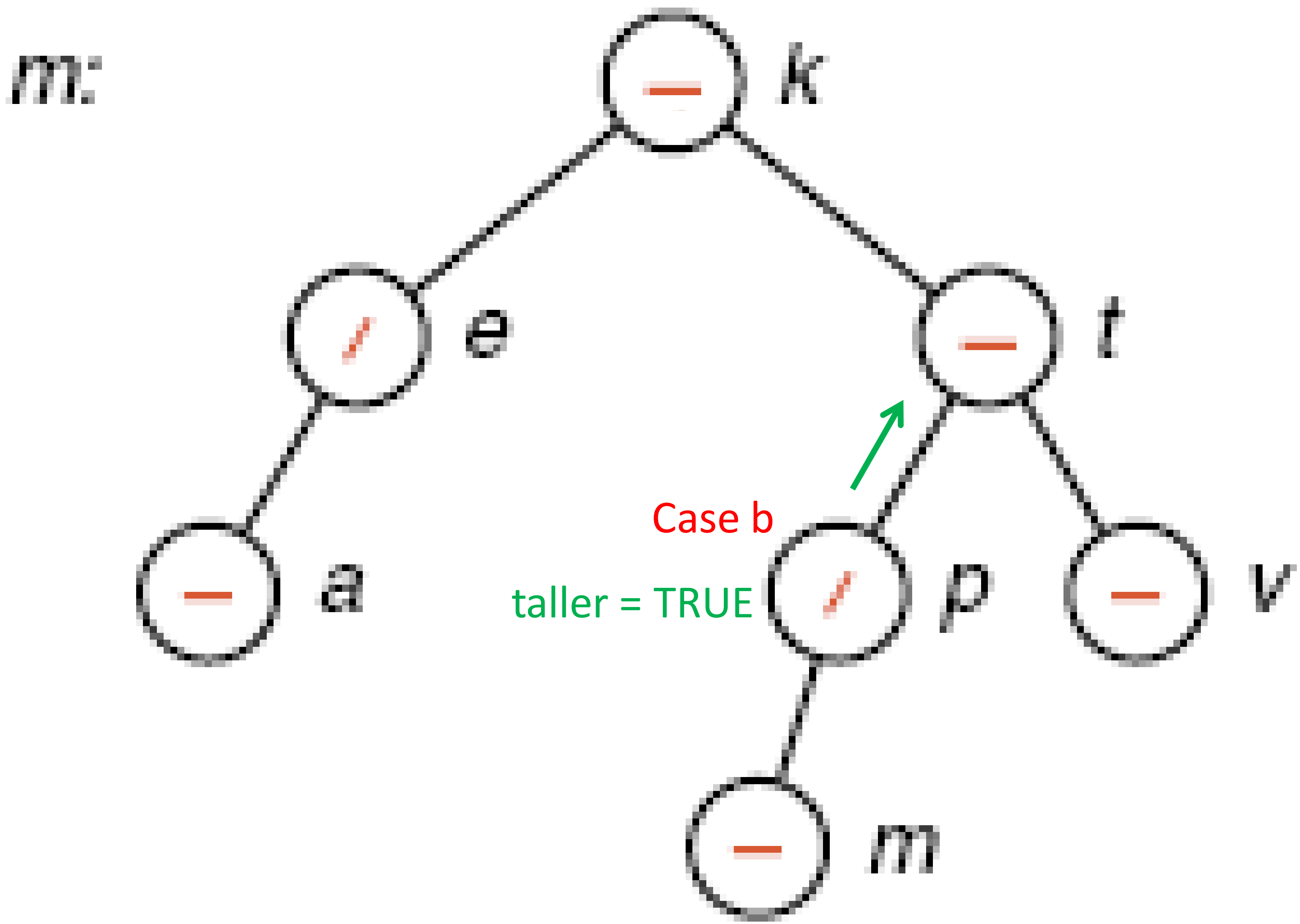
Insertion into an AVL tree



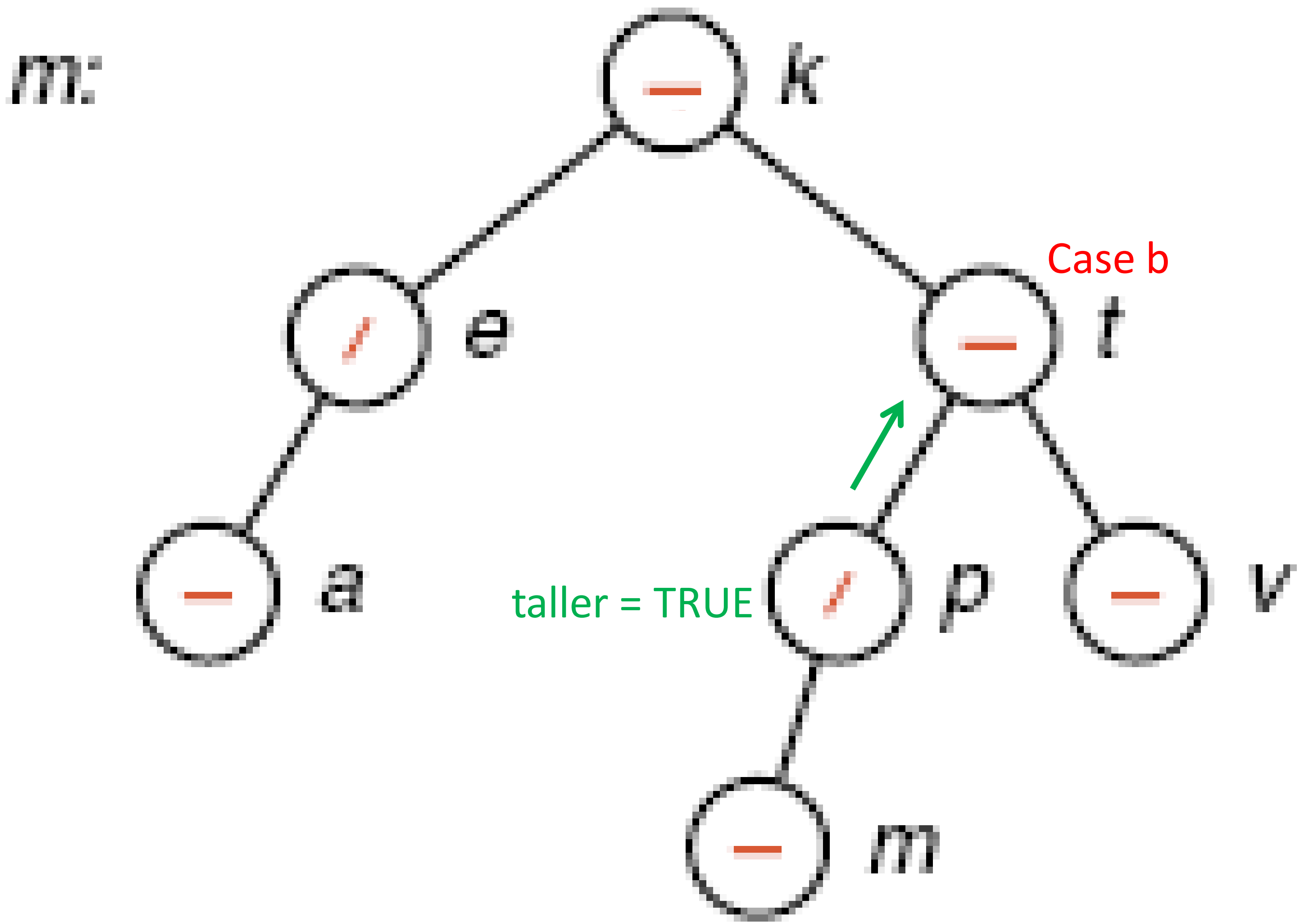
Insertion into an AVL tree



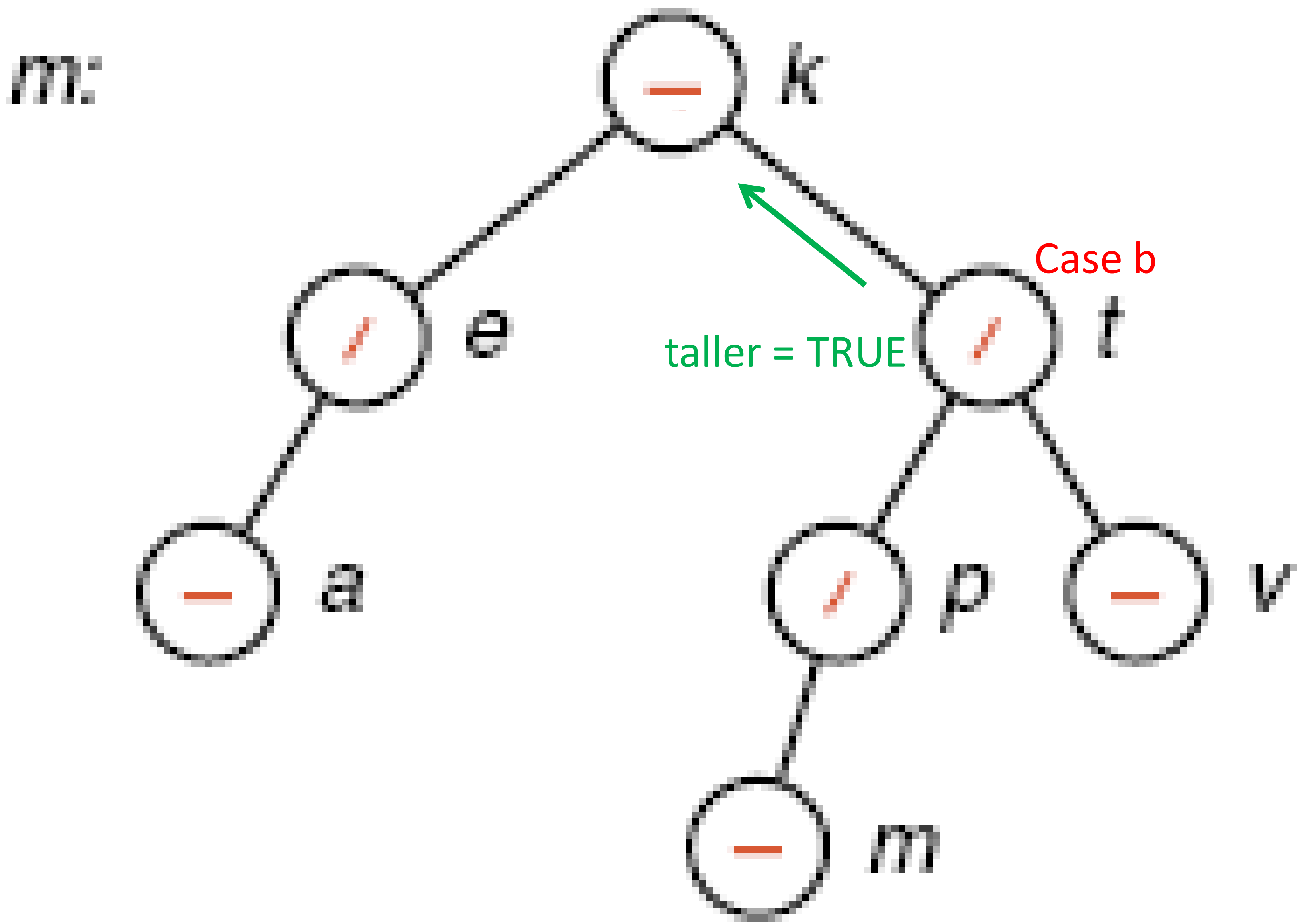
Insertion into an AVL tree



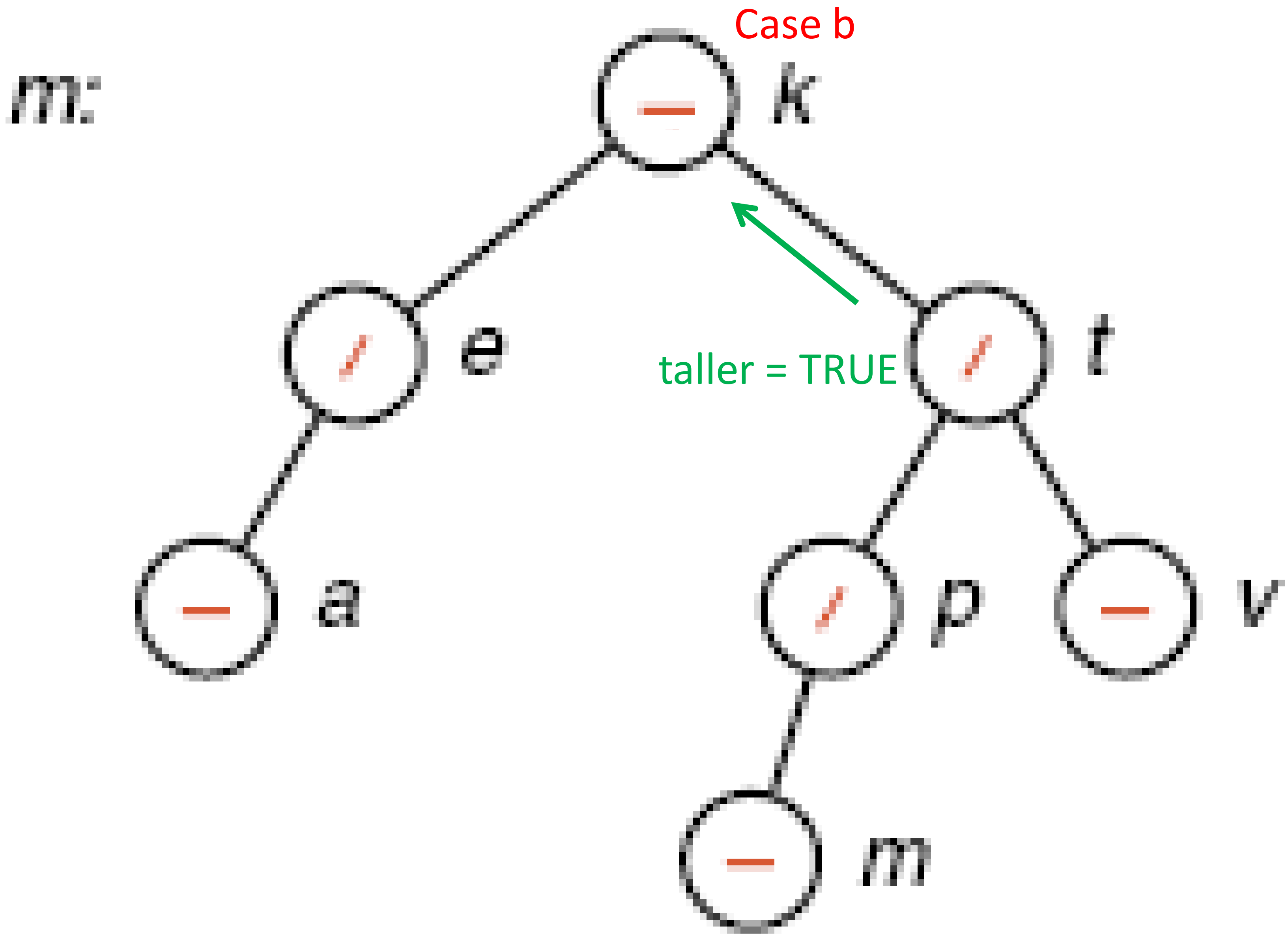
Insertion into an AVL tree



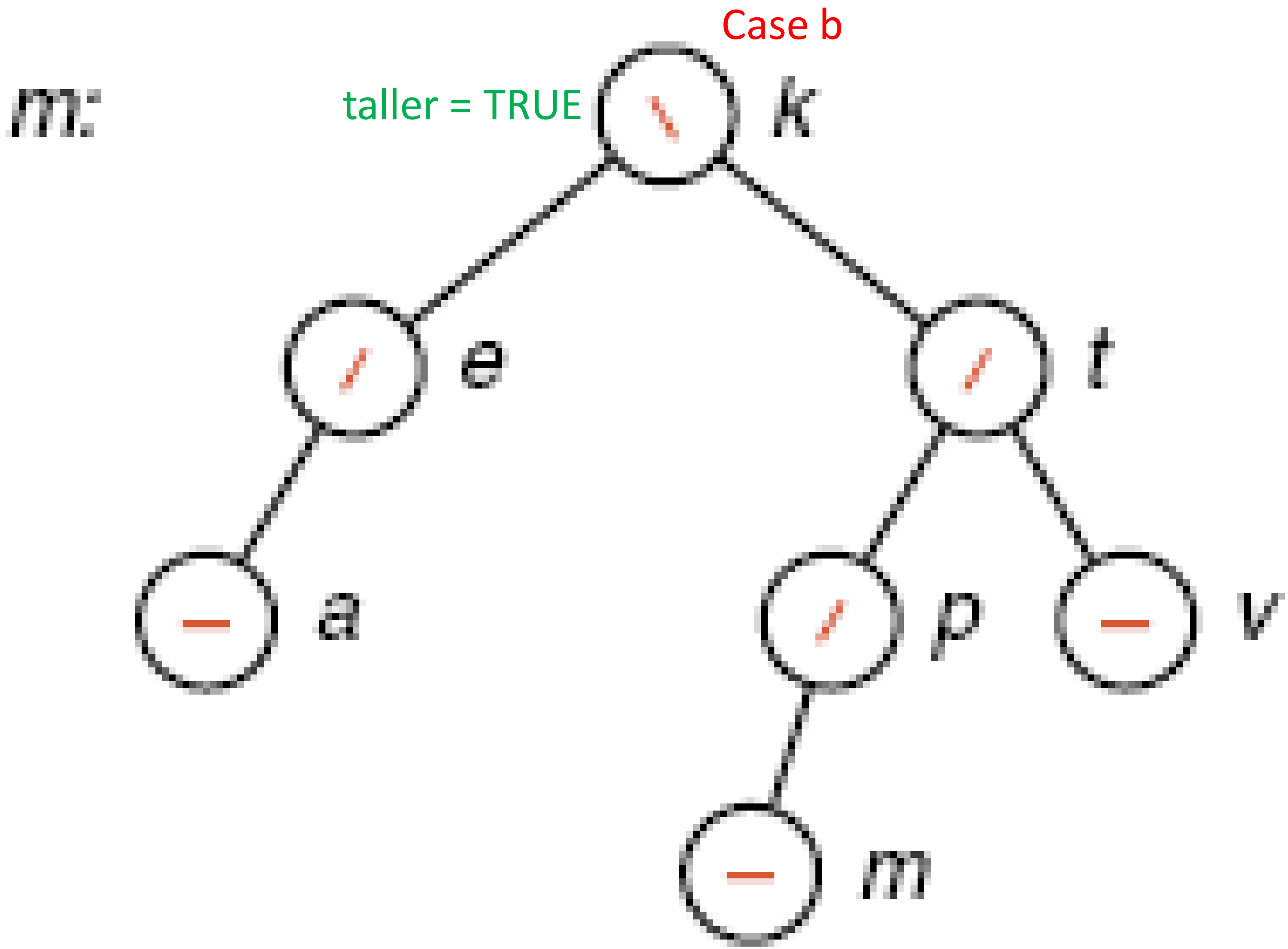
Insertion into an AVL tree



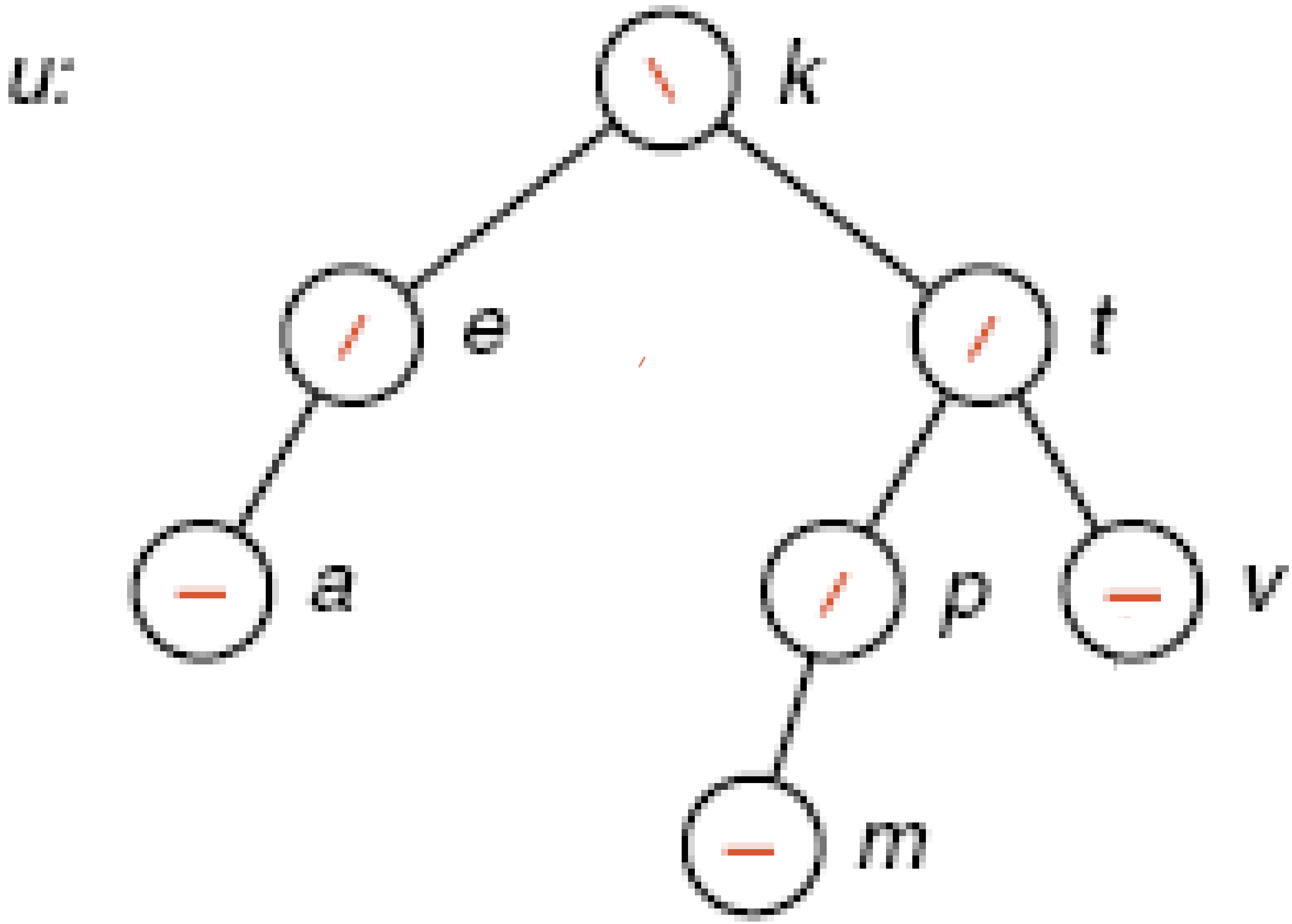
Insertion into an AVL tree



Insertion into an AVL tree

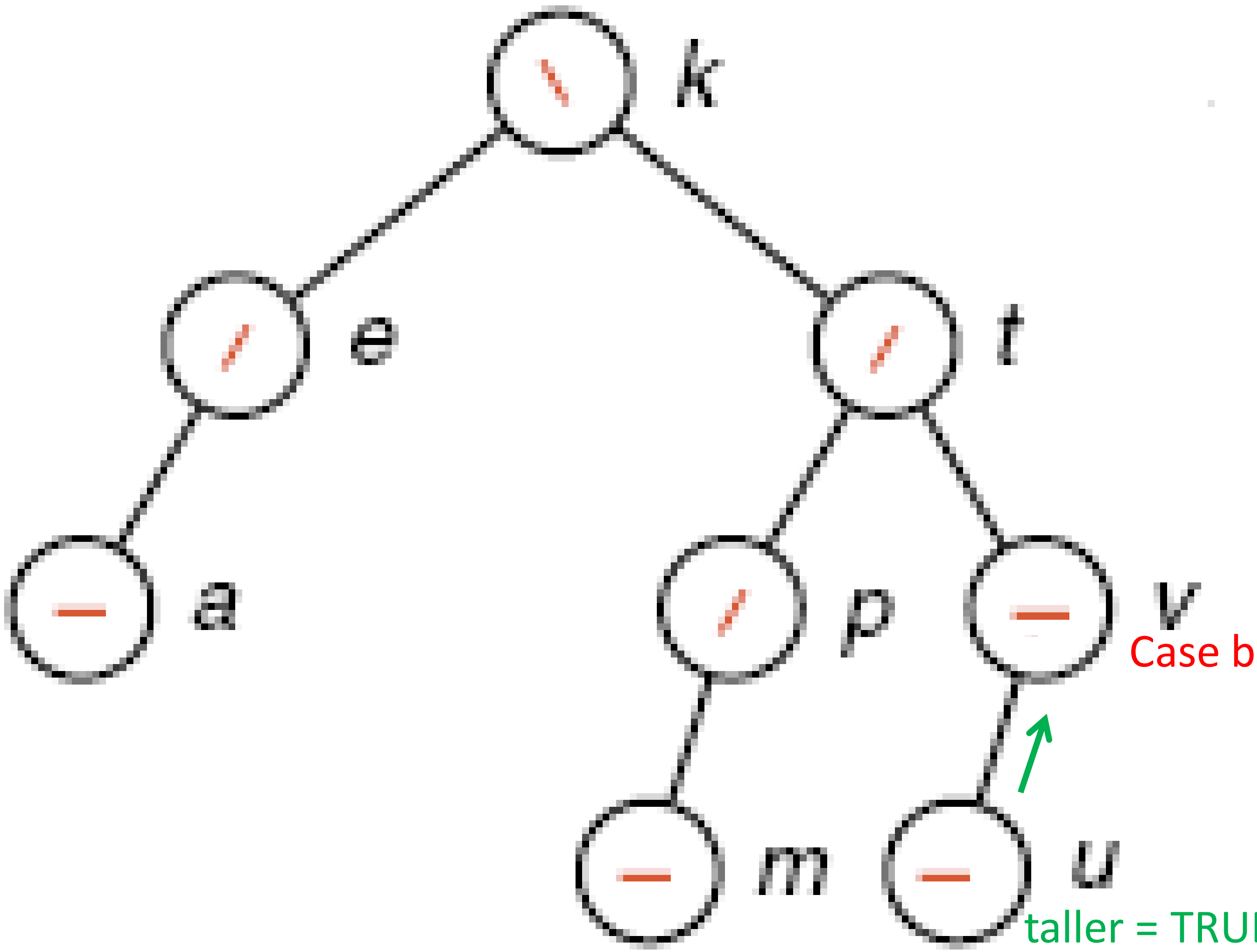


Insertion into an AVL tree

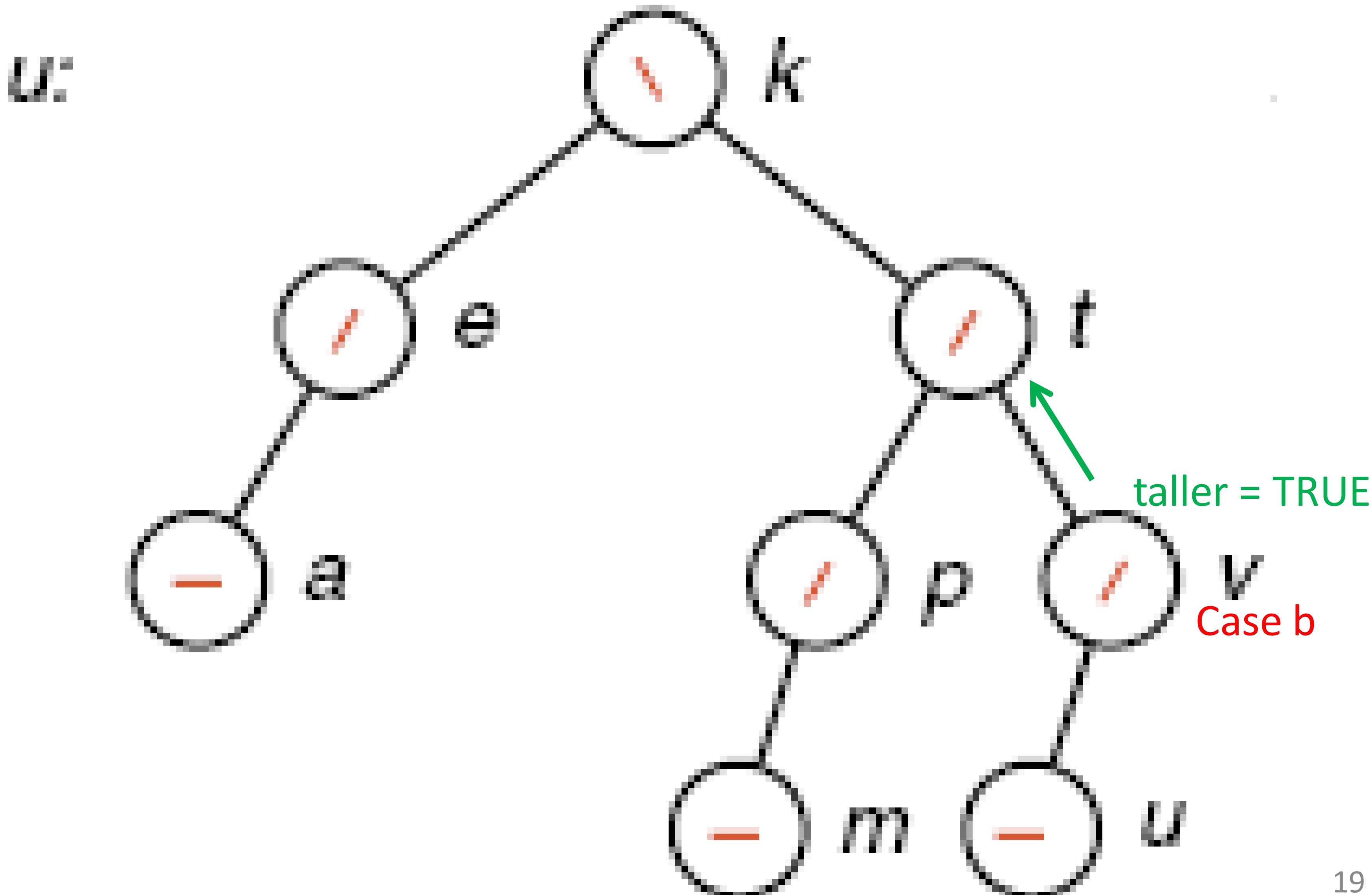


Insertion into an AVL tree

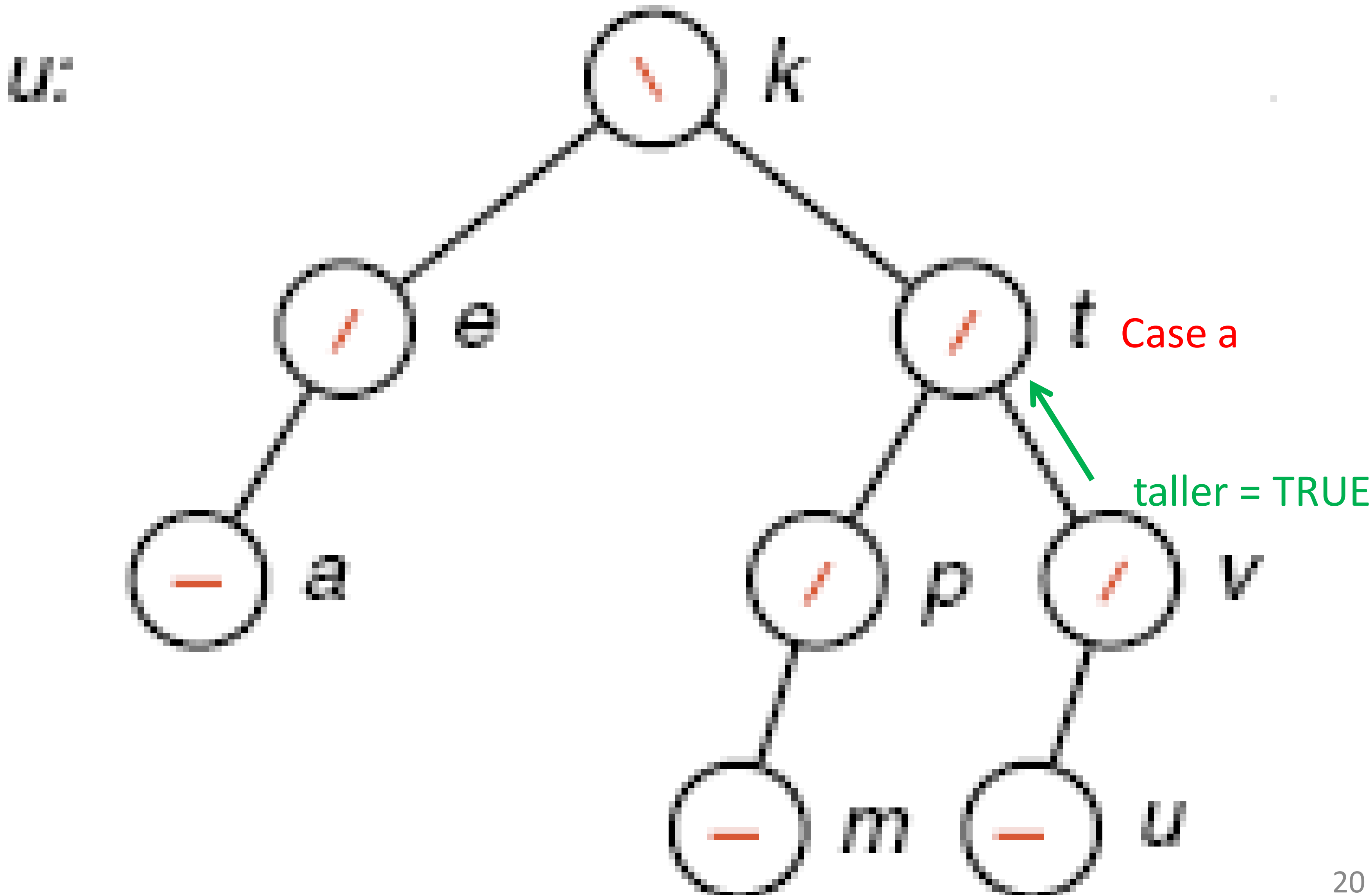
u:



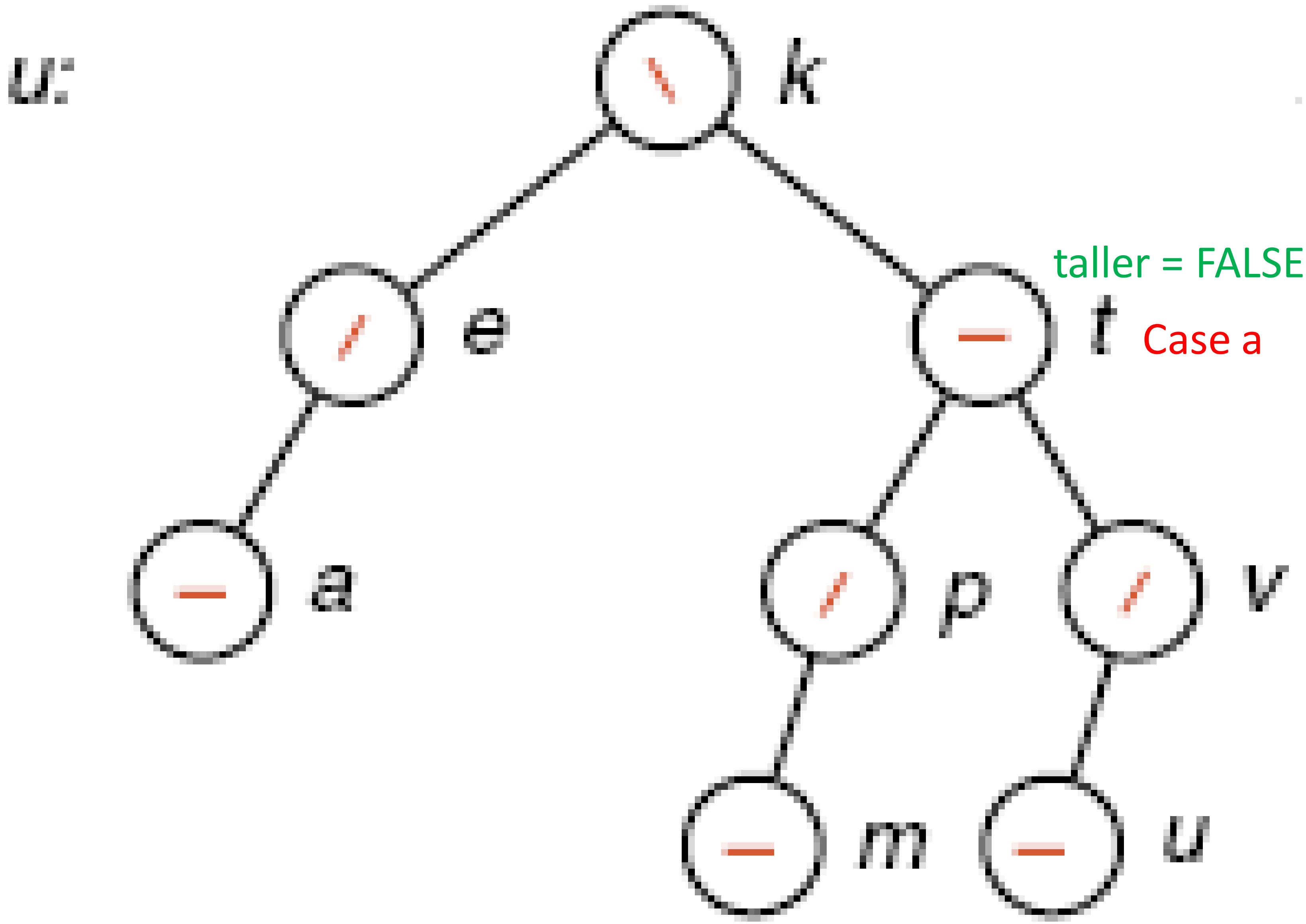
Insertion into an AVL tree



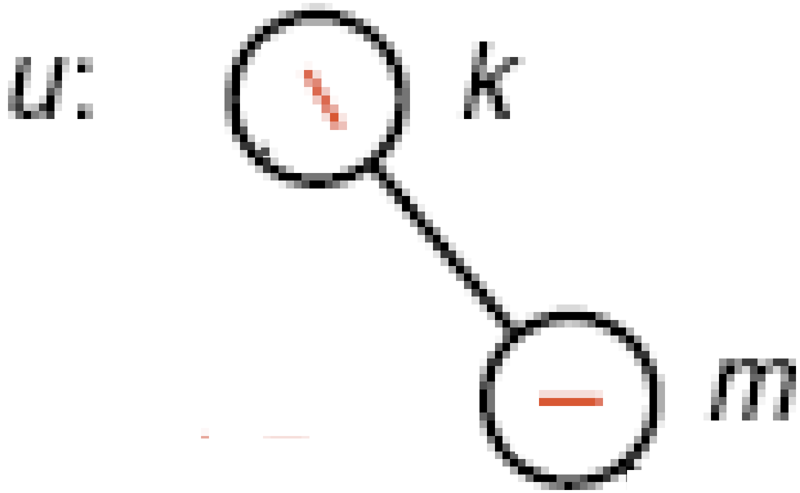
Insertion into an AVL tree



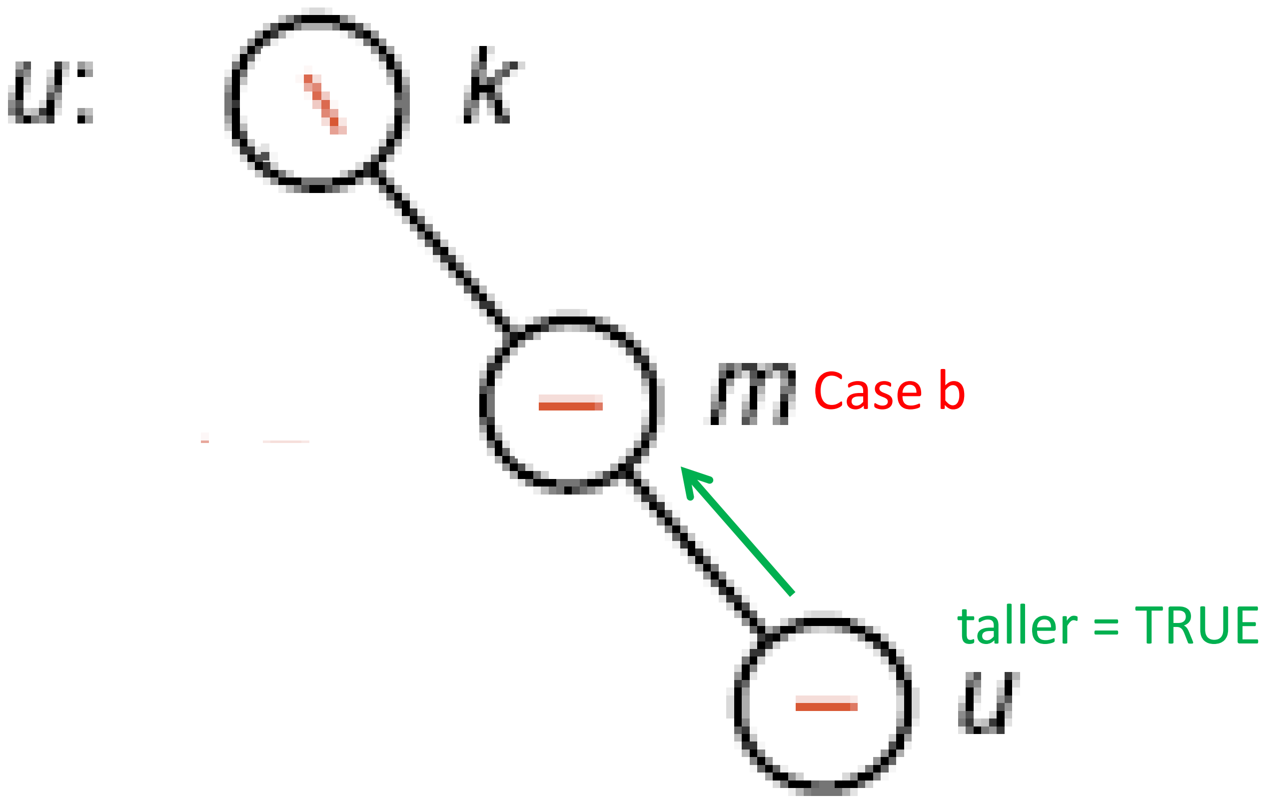
Insertion into an AVL tree



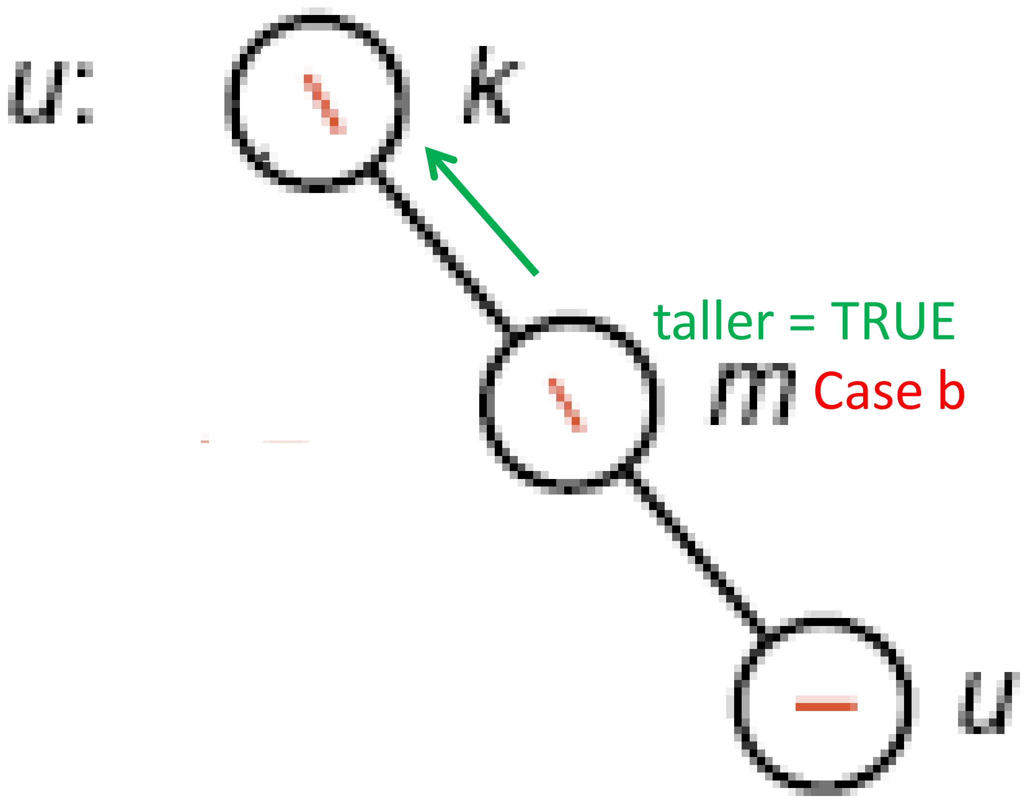
Insertion into an AVL tree



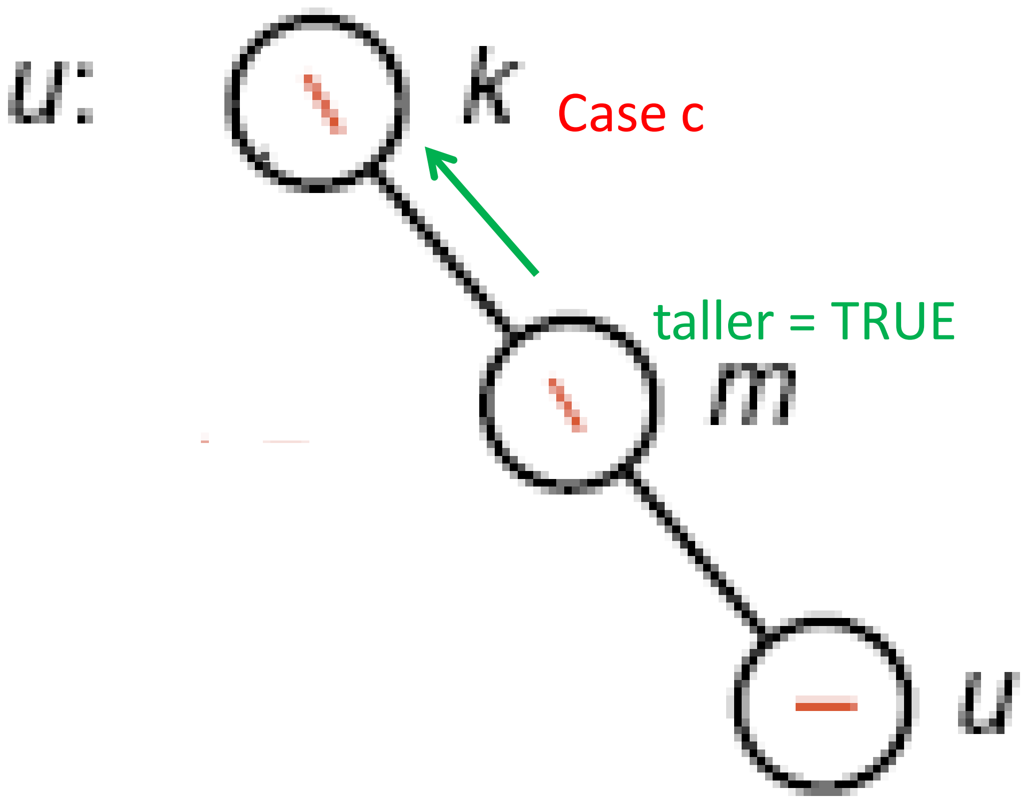
Insertion into an AVL tree



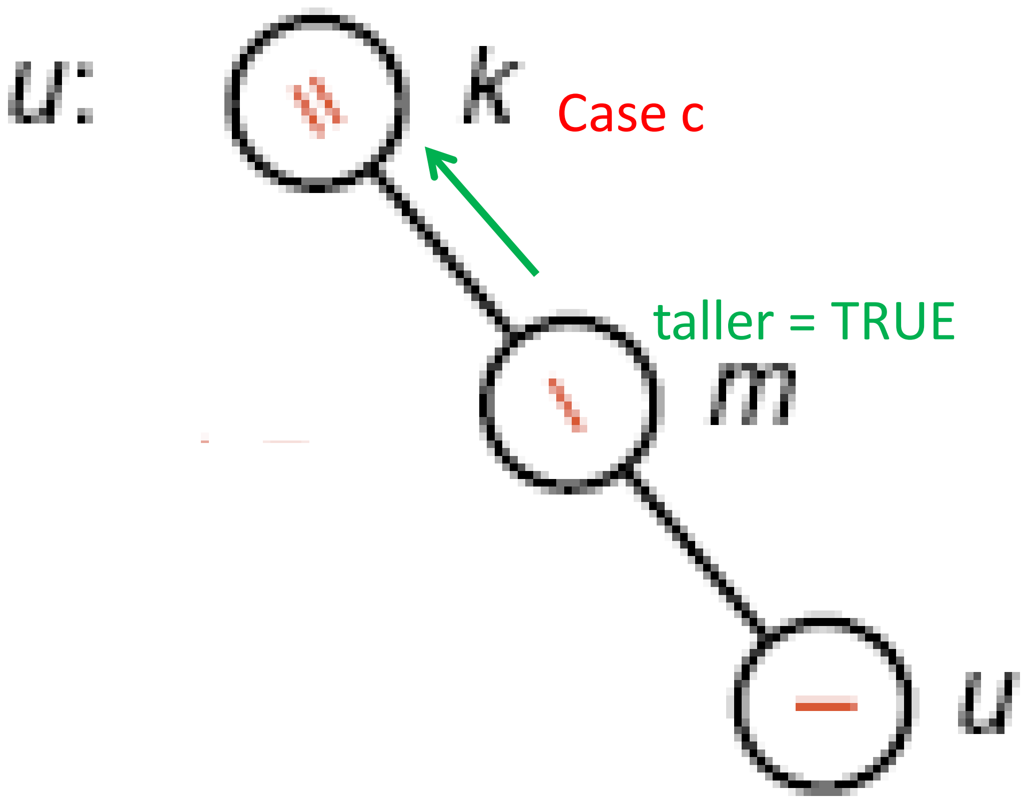
Insertion into an AVL tree



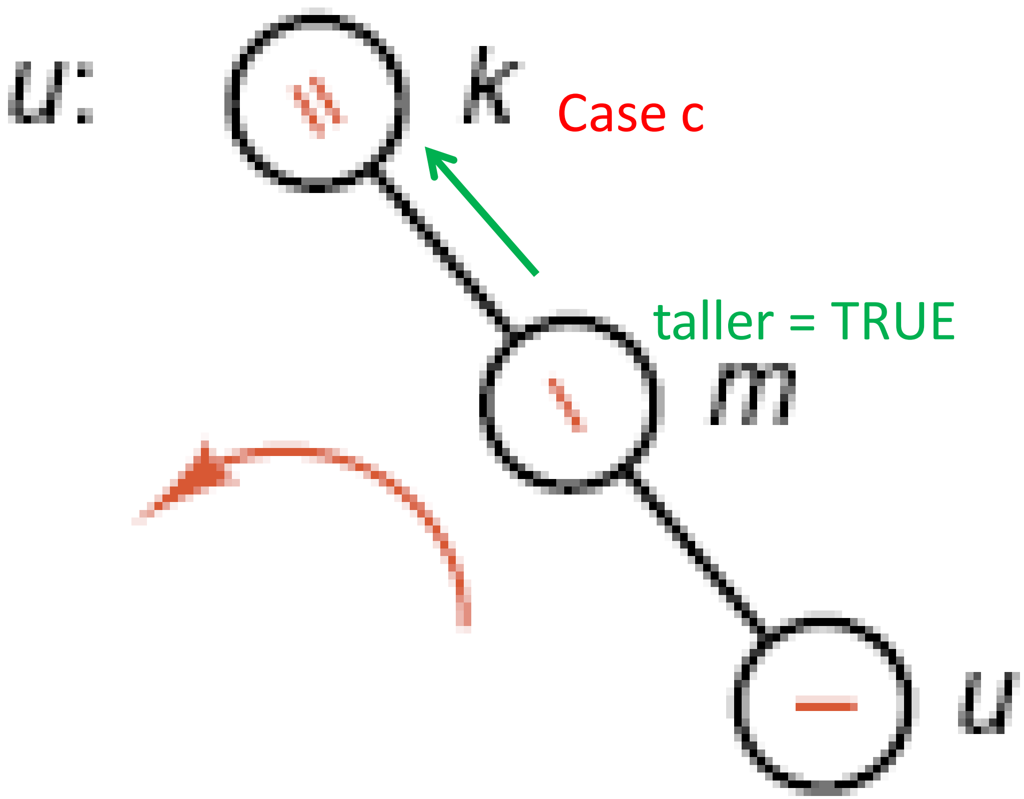
Insertion into an AVL tree



Insertion into an AVL tree

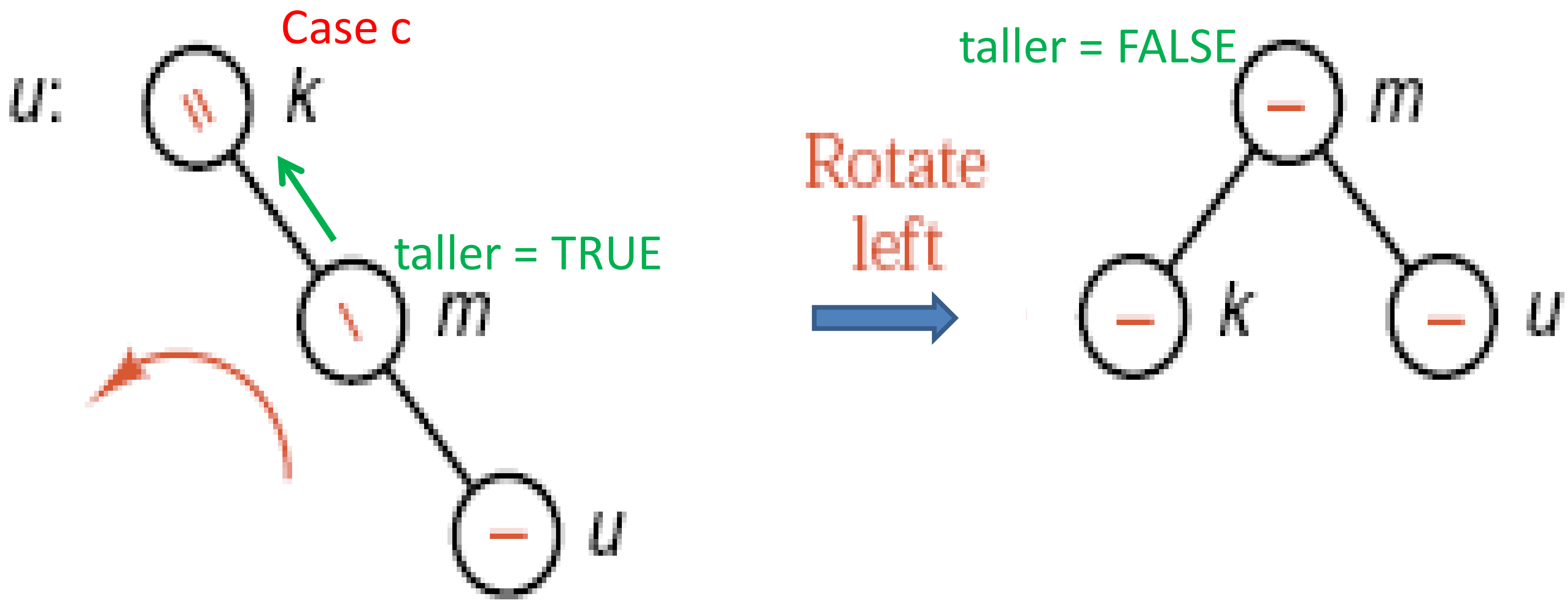


Rebalancing at the node violating AVL definition



Single Rotation

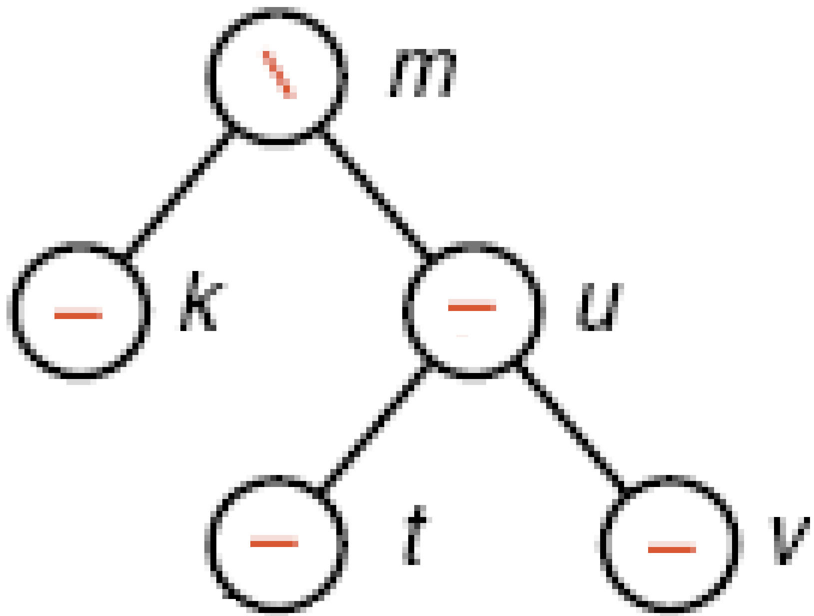
Rebalancing at the node violating AVL definition



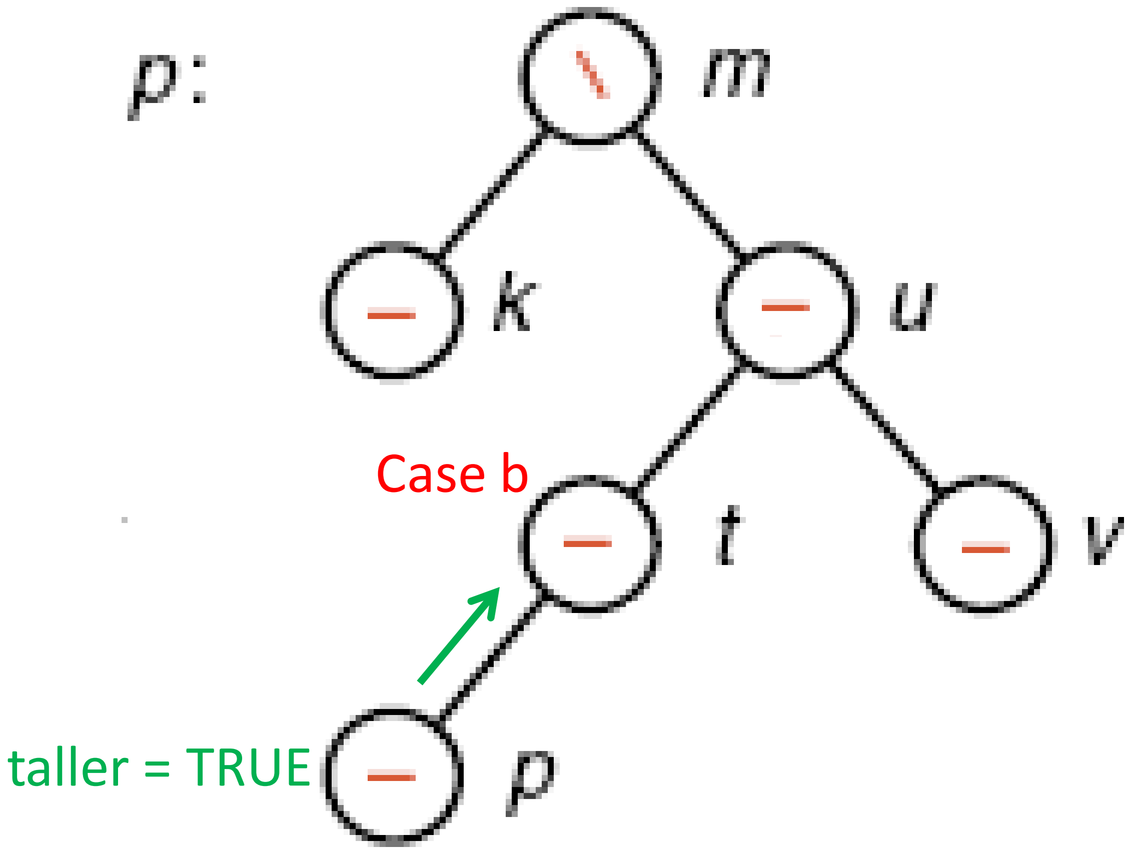
Single Rotation

Insertion into an AVL tree

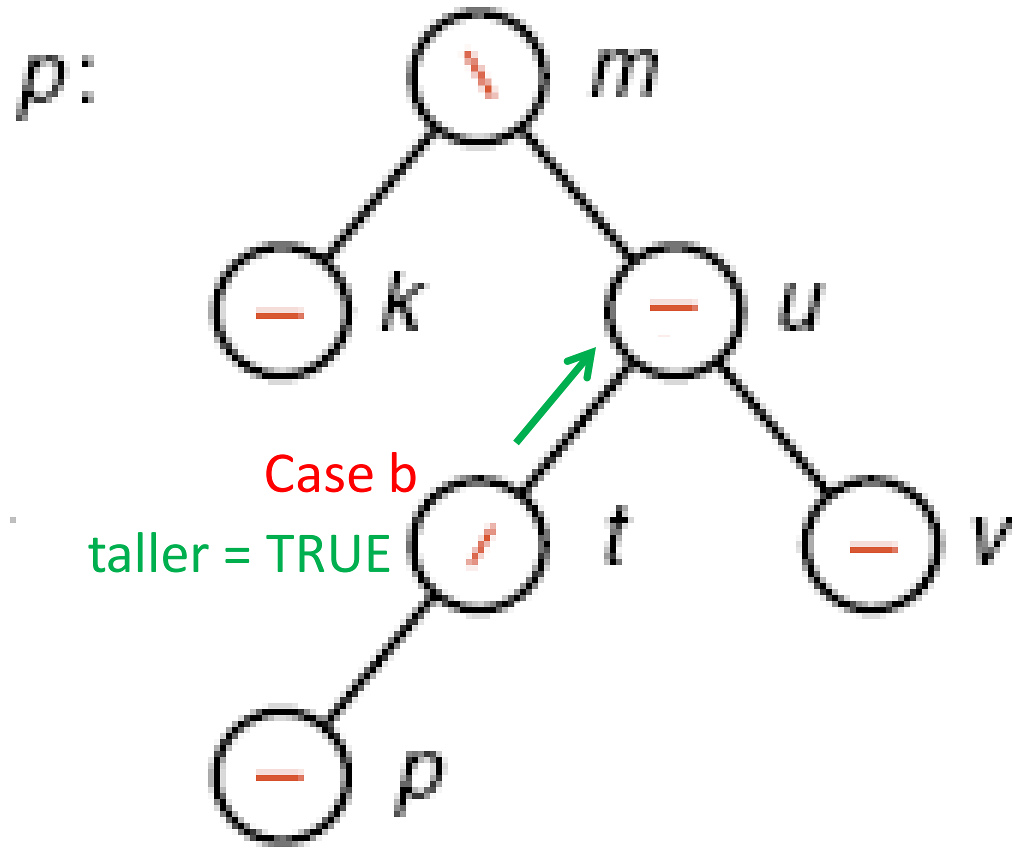
p:



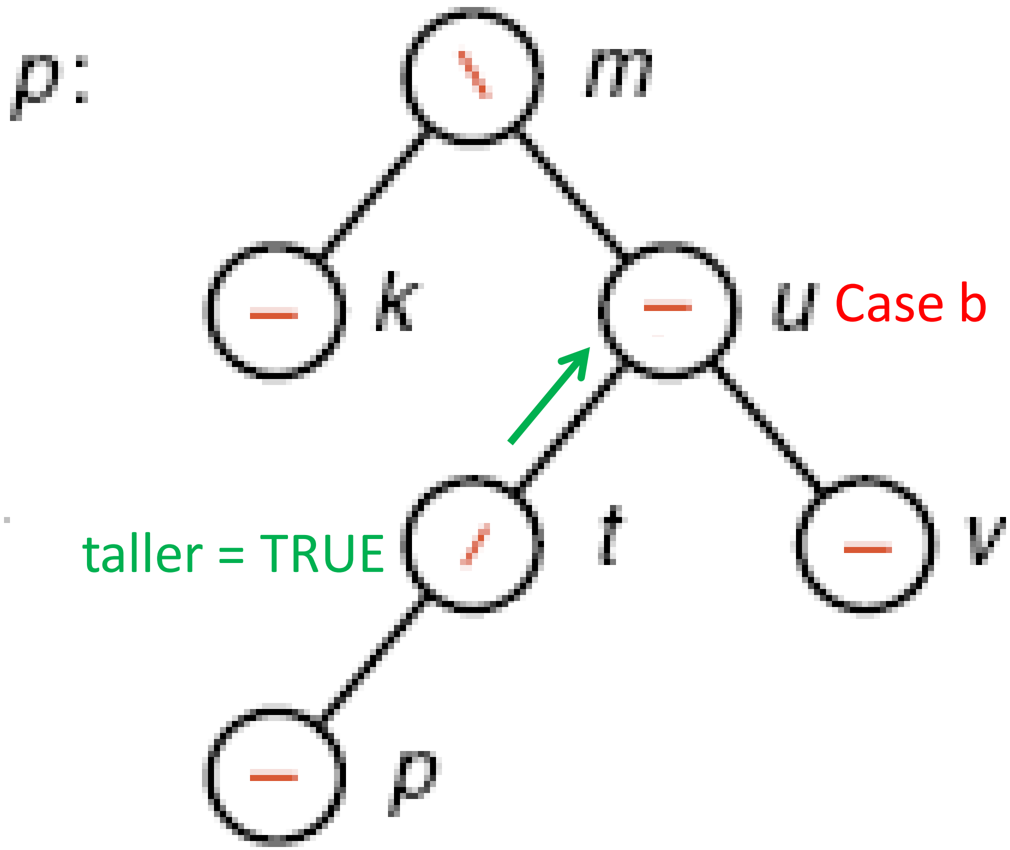
Insertion into an AVL tree



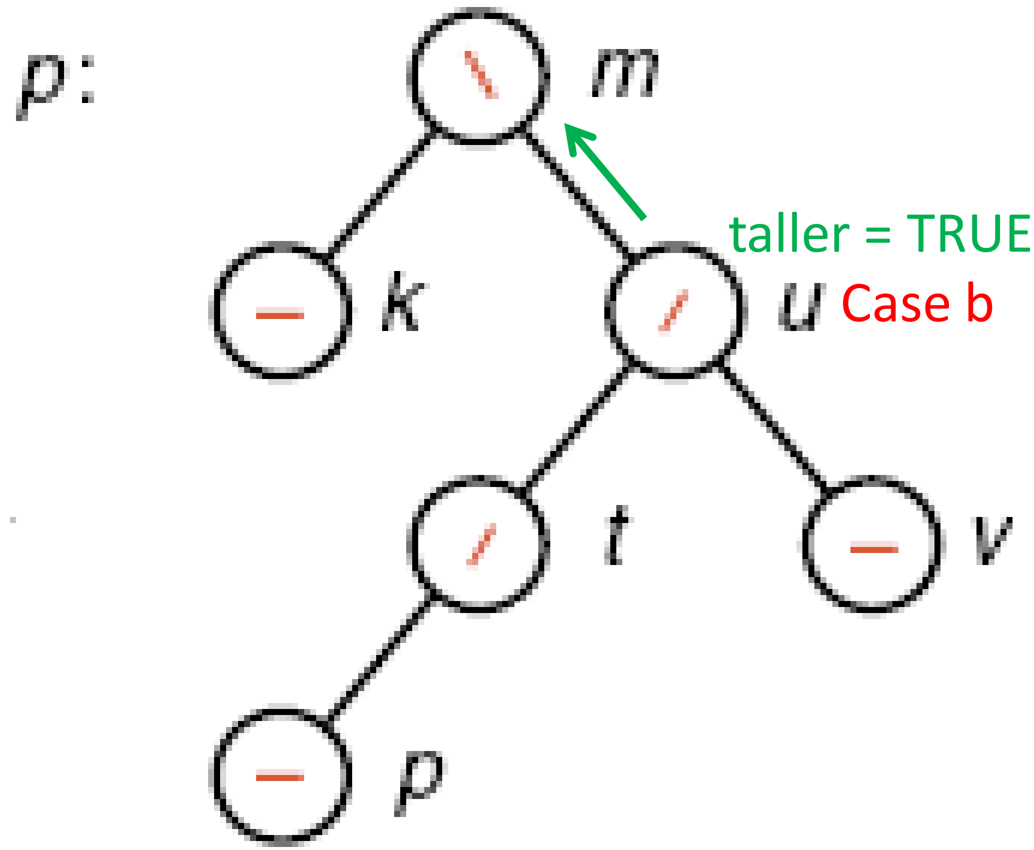
Insertion into an AVL tree



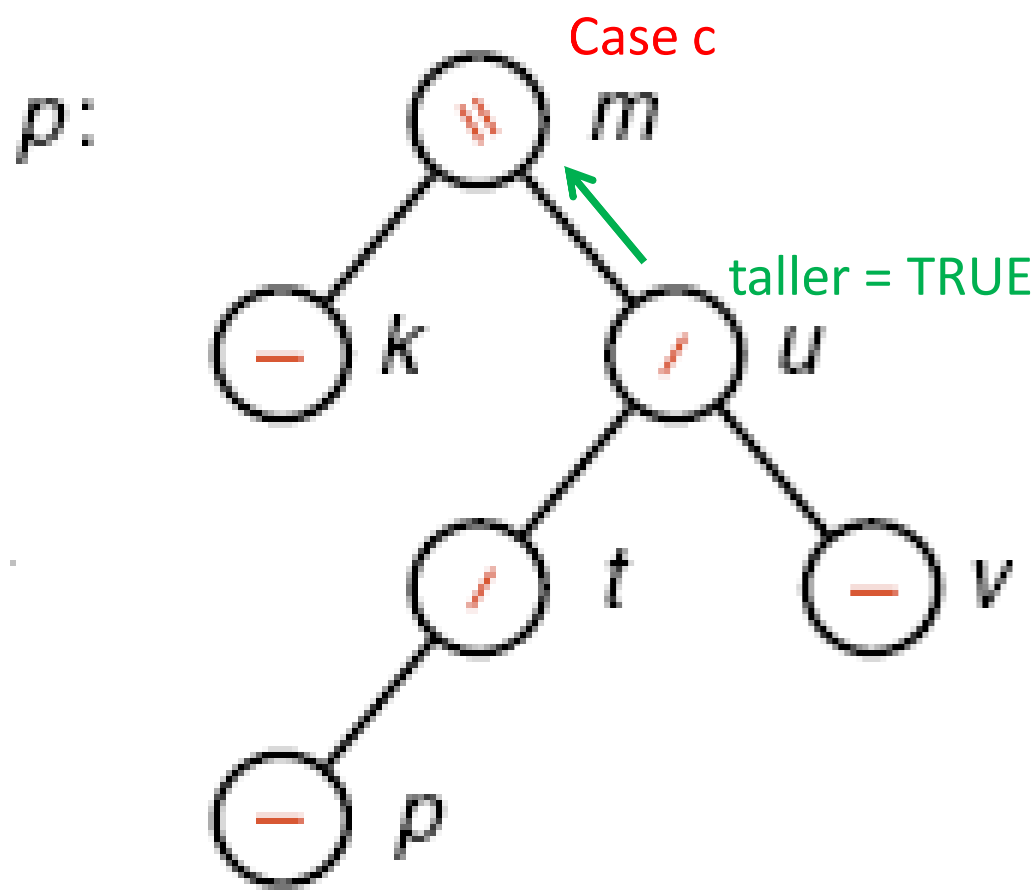
Insertion into an AVL tree



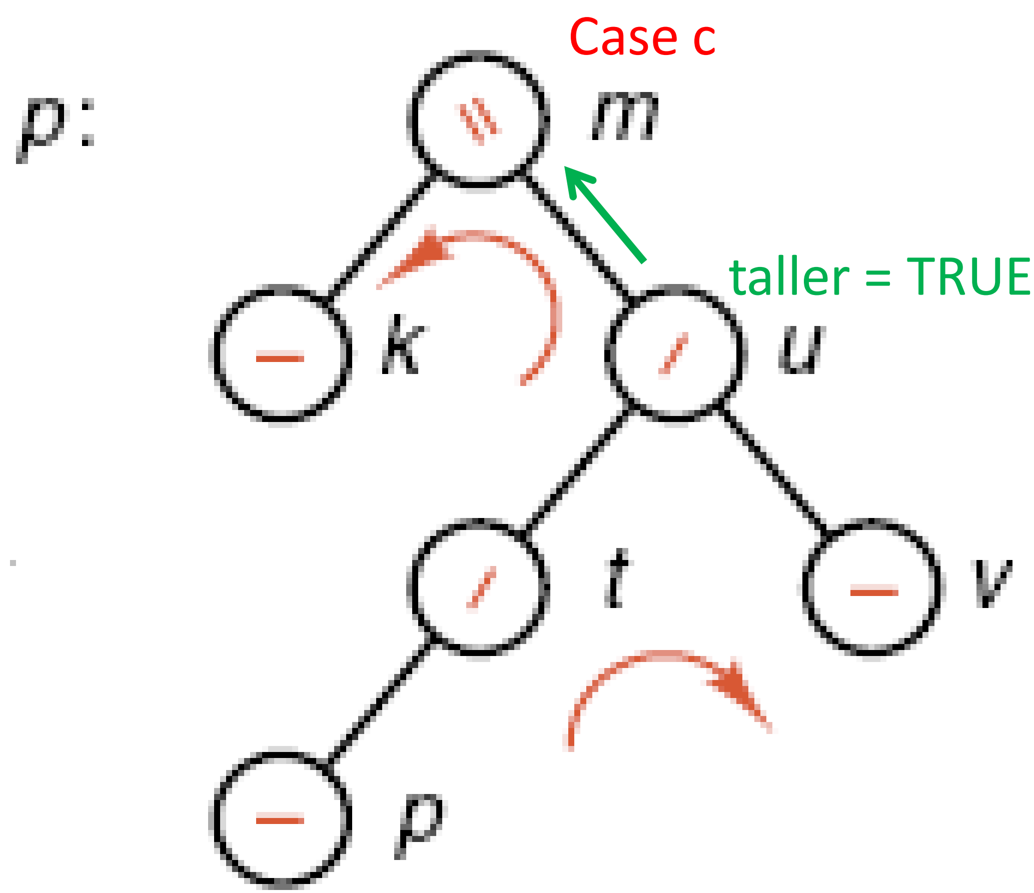
Insertion into an AVL tree



Insertion into an AVL tree

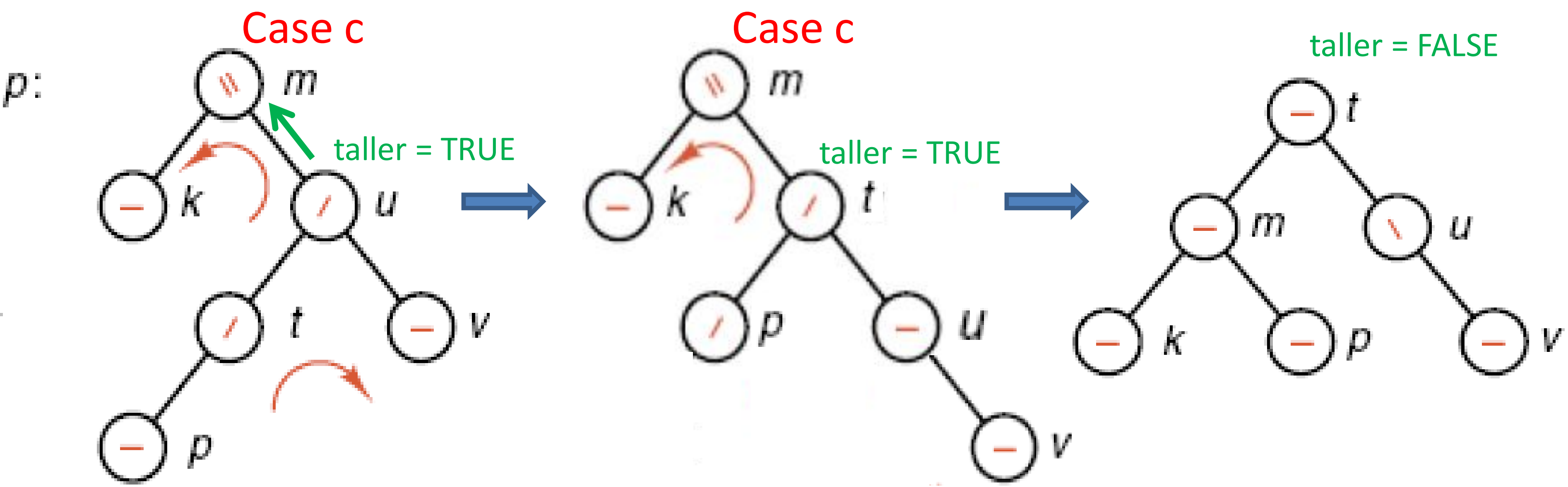


Rebalancing at the node violating AVL definition



Double Rotation

Rebalancing at the node violating AVL definition



Double Rotation

Insert Node into AVL Tree

<ErrorCode> **Insert** (val DataIn <DataType>)

Inserts a new node into an AVL tree.

Post If the key of DataIn already belongs to the AVL tree, *duplicate_error* is returned. Otherwise, DataIn is inserted into the tree in such a way that the properties of an AVL tree are preserved.

Return *duplicate_error* or *success*.

Uses recursive_Insert .

1. taller <boolean> // Has the tree grown in height?
2. return recursive_Insert (root, DataIn, taller)

End Insert

Recursive Insert

<ErrorCode> **recursive_Insert** (ref subroot <pointer>,
val DataIn <DataType>, ref taller <boolean>)

Inserts a new node into an AVL tree.

Pre subroot points to the root of a tree/ subtree.
DataIn contains data to be inserted into the subtree.

Post If the key of DataIn already belongs to the subtree, *duplicate_error* is returned. Otherwise, DataIn is inserted into the subtree in such a way that the properties of an AVL tree are preserved.
If the subtree ***is increased in height***, the parameter taller is set to TRUE; otherwise it is set to FALSE.

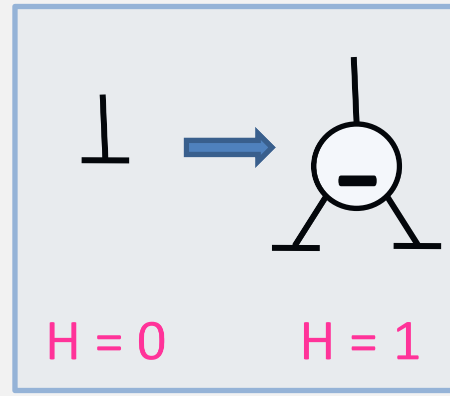
Return *duplicate_error* or *success*.

Uses recursive_Insert , left_balance, right_balance functions.

Recursive Insert (cont.)

1. result = *success*

2. **if** (subroot is NULL)
    1. Allocate subroot
    2. subroot ->data = DataIn
    3. taller = TRUE



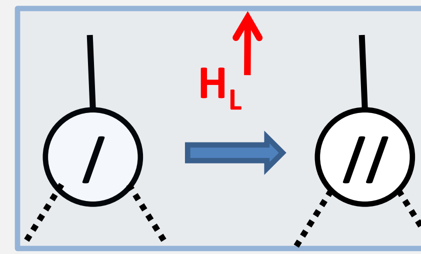
3. **else if** (DataIn = subroot ->data)
    1. result = *duplicate_error*
    2. taller = FALSE

4. **else if** (DataIn < subroot ->data)

Recursive Insert (cont.)

4. **else if** (DataIn < subroot ->data) *// Insert in the left subtree*
    1. result = recursive_Insert(subroot->left, DataIn, taller)
    2. **if** (taller = TRUE)

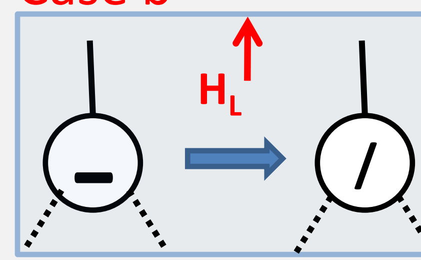
        Case c

        1. **if** (balance of subroot = *left_higher*)
            1. left_balance (subroot)
            2. taller = FALSE

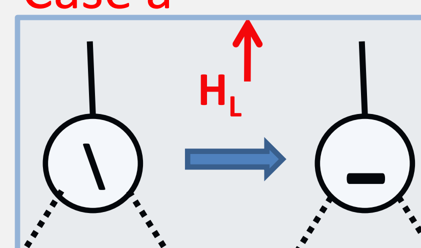
               // Rebalancing always shortens the tree.

        Case b

        2. **else if** (balance of subroot = *equal_height*)
            1. subroot->balance = *left_higher*

        Case a

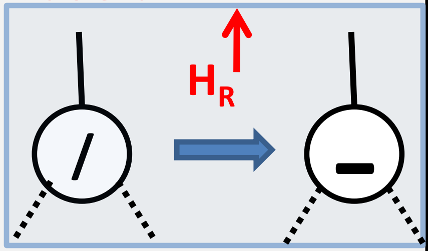
        

        3. **else if** (balance of subroot = *right_higher*)
            1. subroot->balance = *equal_height*
            2. taller = FALSE

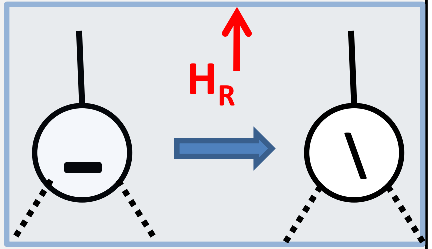
4. **else** *// (DataIn > subroot ->data) Insert in the right subtree*
   1. result = recursive_Insert(subroot->right, DataIn, taller)
   2. **if** (taller = TRUE)
      1. **if** (balance of subroot = *left_higher*)
         1. subroot->balance = *equal_height*
         2. taller = FALSE
      2. **else if** (balance of subroot = *equal_height*)
         1. subroot->balance = *right_higher*
      3. **else if** (balance of subroot = *right_higher*)
         1. right_balance (subroot)
         2. taller = FALSE

         // Rebalancing always shortens the tree.

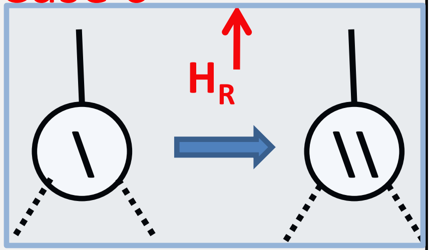
Case a



Case b



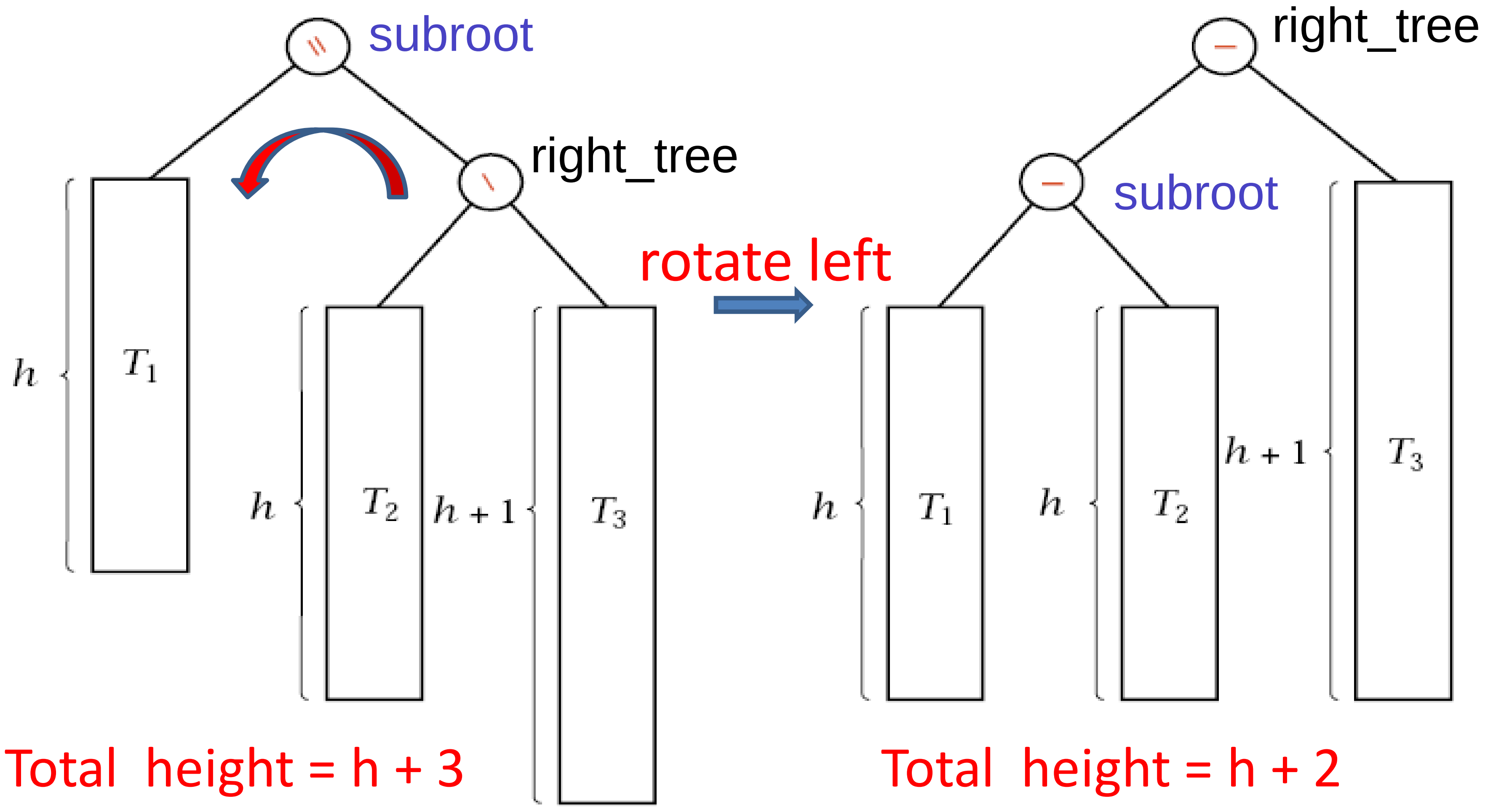
Case c



1. return result

End recursive_Insert

Rotation of an AVL Tree



1. right_tree = subroot->right
2. subroot->right = right_tree->left
3. right_tree->left = subroot
4. subroot = right_tree

Rotation of an AVL Tree

<void> **rotate_left** (ref subroot <pointer>)

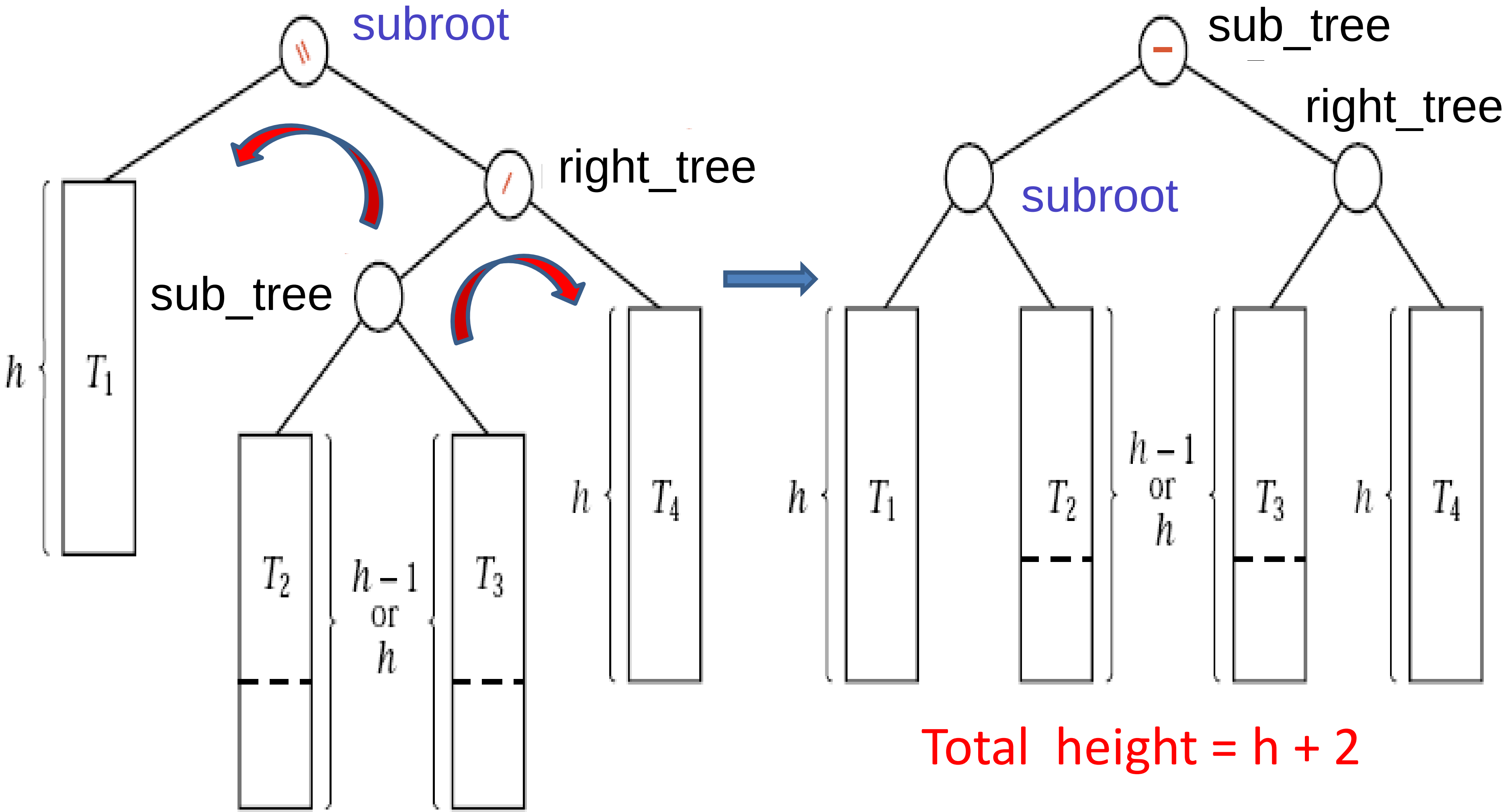
Pre subroot is not NULL and points to the subtree of the AVL tree. This subtree has a nonempty right subtree.

Post subroot is reset to point to its former right child, and the former subroot node is the left child of the new subroot node.

1. right_tree = subroot->right
2. subroot->right = right_tree->left
3. right_tree->left = subroot
4. subroot = right_tree

End rotate_left

Double Rotate



Total height = h + 2

One of T_2 or T_3 has height h.
Total height = h + 3

Double Rotate

The new balance factors for subroot and right_tree depend on the previous balance factor for subtree

old sub_tree	*new* subroot	*new* right_tree	*new* sub_tree
-	-	-	-
/	-	\	-
\	/	-	-

right_balance function

<void> **right_balance** (ref subroot <pointer>)

Pre subroot points to a subtree of an AVL tree, doubly unbalanced on the right.

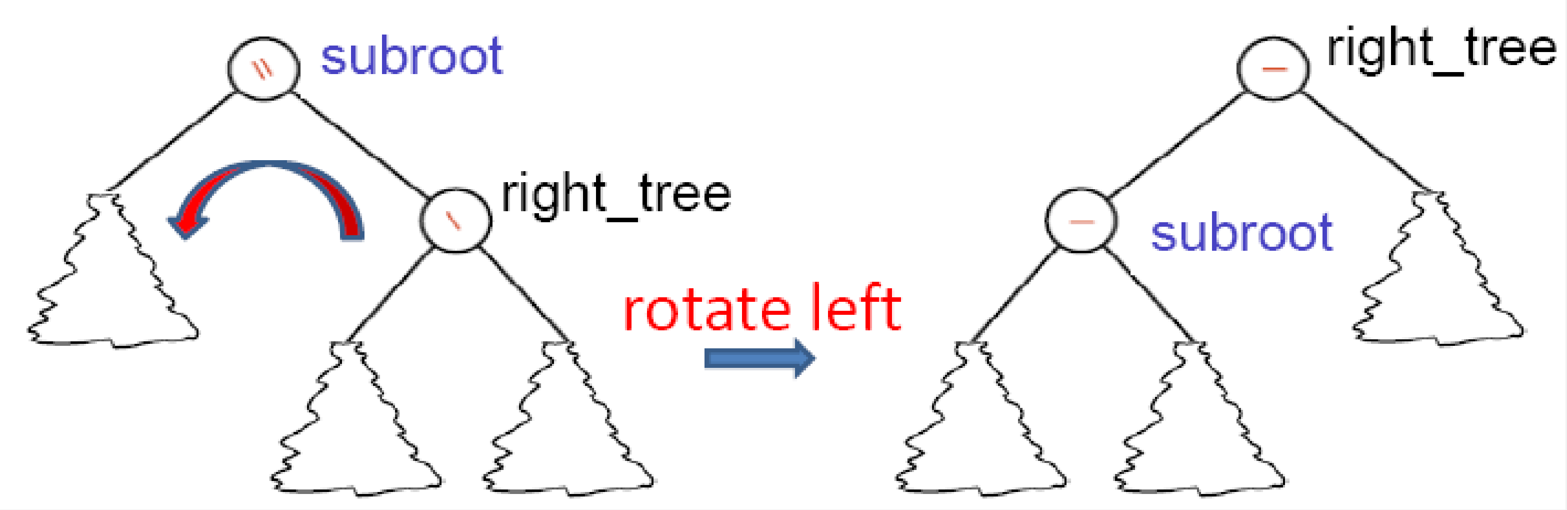
Post The AVL properties have been restored to the subtree.

Uses rotate_right, rotate_left functions.

right_balance function (cont.)

1. right_tree = subroot->right

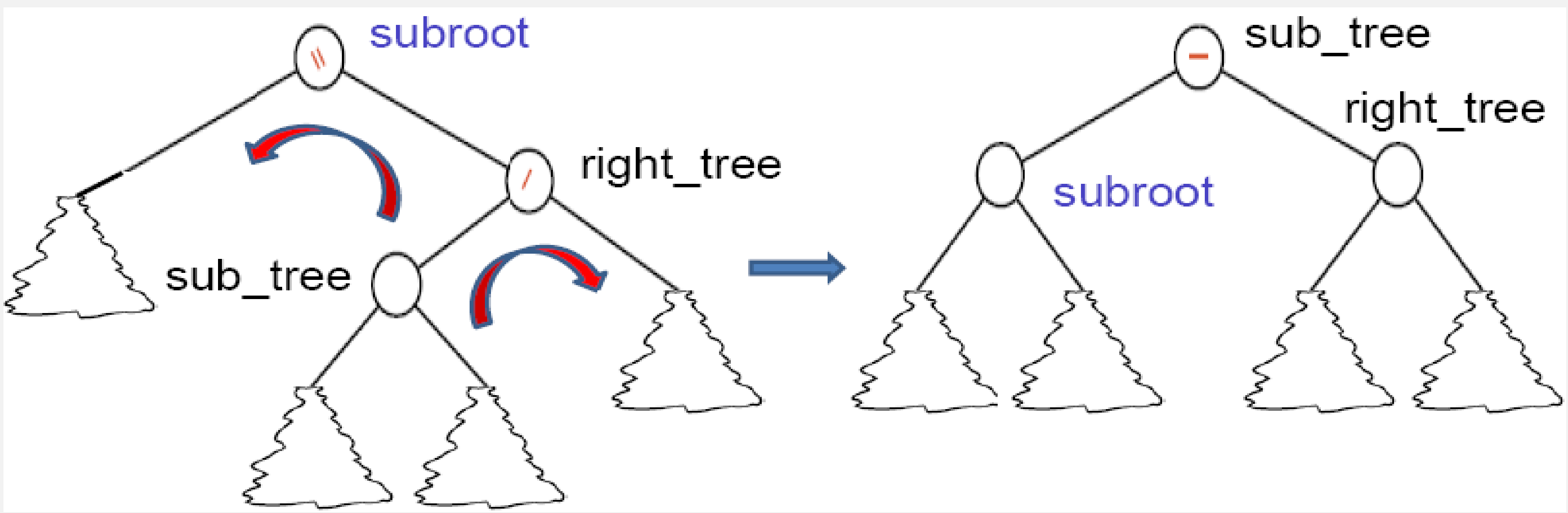
2. if (balance of right_tree = *right_higher*)
    1. subroot->balance = *equal_height*
    2. right_tree->balance = *equal_height*
    3. rotate_left (subroot)



3. if (balance of right_tree = *equal_height*) *// impossible case*

right_balance function (cont.)

4. **if** (balance of right_tree = *left_higher*)
    1. subtree = right_tree->left
    2. subtree->balance = *equal_height*
    3. rotate_right (right_tree)
    4. rotate_left (subroot)



right_balance function (cont.)

5. **if** (balance of subtree = *equal_height*)
    1. subroot->balance = *equal_height*
    2. right_tree->balance = *equal_height*
6. **else if** (balance of subtree = *left_higher*)
    1. subroot->balance = *equal_height*
    2. right_tree->balance = *right_higher*
7. **else** *// (balance of subtree = right_higher)*
    1. subroot->balance = *left_higher*
    2. right_tree->balance = *equal_height*

End right_balance

old sub_tree	*new* subroot	*new* right_tree	*new* sub_tree
-	-	-	-
/	-	\	-
\	/	-	-

Removal of a node

- Reduce the problem to the case when the node x to be removed has at most one child.

- We use a parameter shorter to show if the height of a subtree has been shortened.

- While shorter is TRUE, do the following steps for each **node p** on the path from the parent of x to the root of the tree.

- When shorter becomes FALSE, the algorithm terminates.

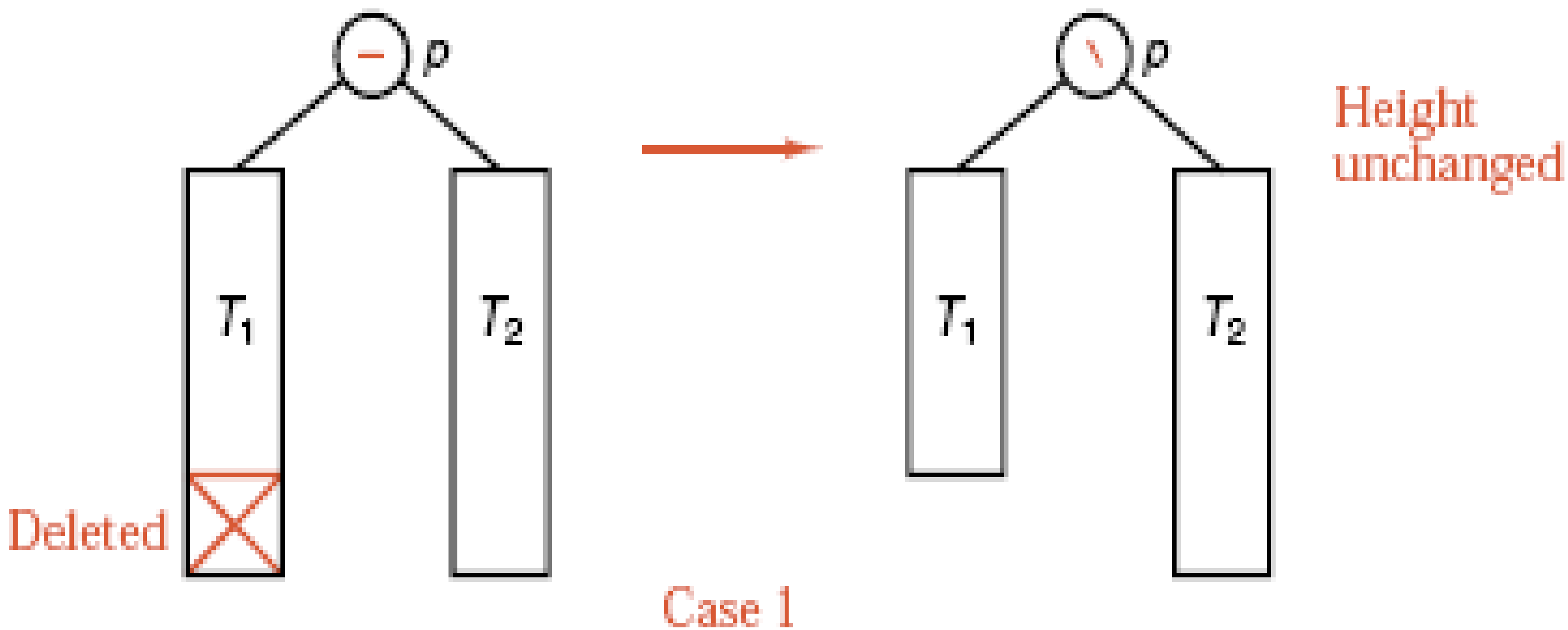
Removal of a node

- **Case 1**: Node p has balance factor equal.

  So only this balance factor must be changed.

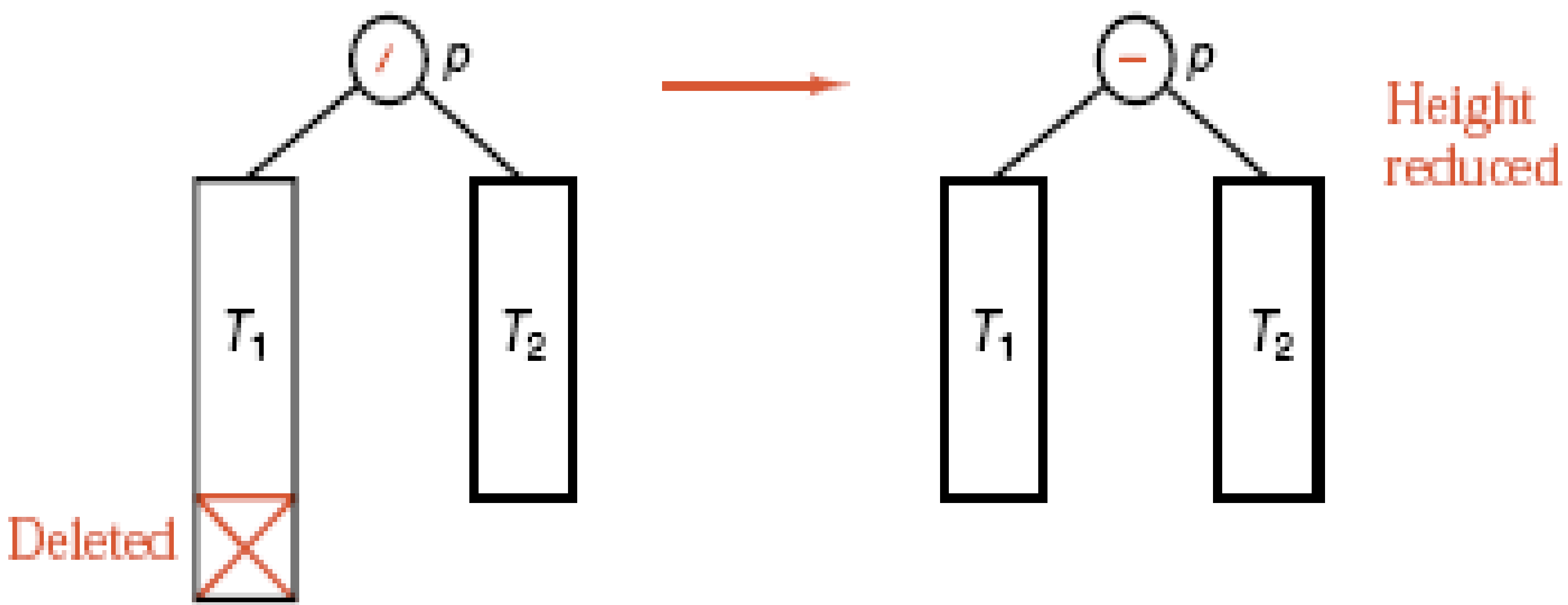
  The height of p is unchanged.

  shorter becomes FALSE.



Removal of a node

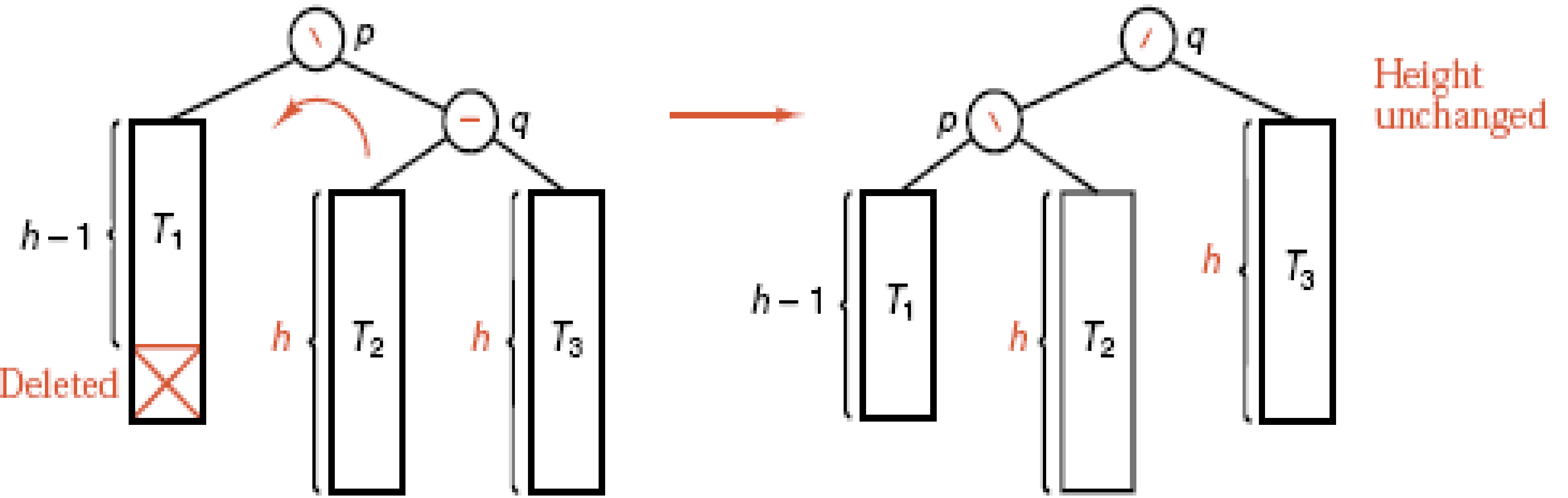
- **Case 2**: The balance factor of p is not equal, the taller subtree was shortened.
  So the balance factor must be changed.
  The height of p is decreased.
  shorter remains TRUE.



Removal of a node

- **Case 3**: The balance factor of p is not equal, the shorter subtree was shortened.
  So *AVL definition is violated at p.* Rebalancing must be done.
  Let q be the root of the taller subtree of p.

Case 3a: The balance factor of q is equal.
  So single rotation needs to do. shorter becomes FALSE.



Case 3a

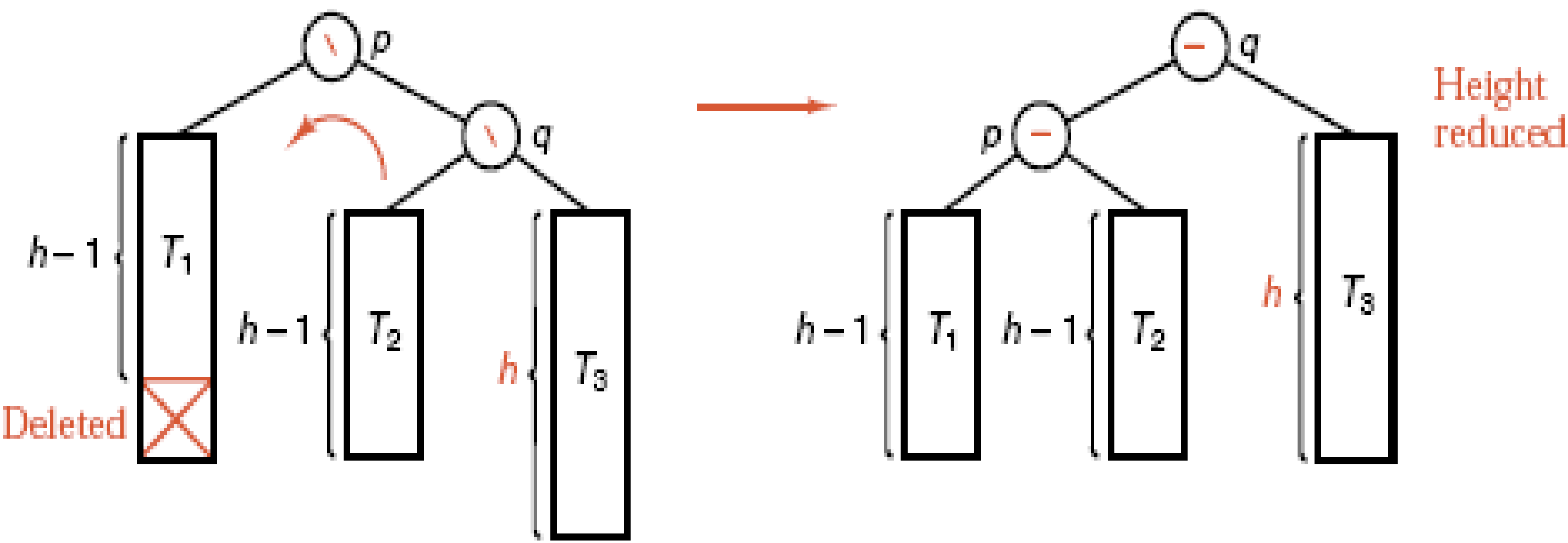
Removal of a node

Case 3b: The balance factor of q is the same as that of p.

So single rotation needs to do.

Balance factors of p and q become equal.

shorter remains TRUE.



Case 3b

Removal of a node

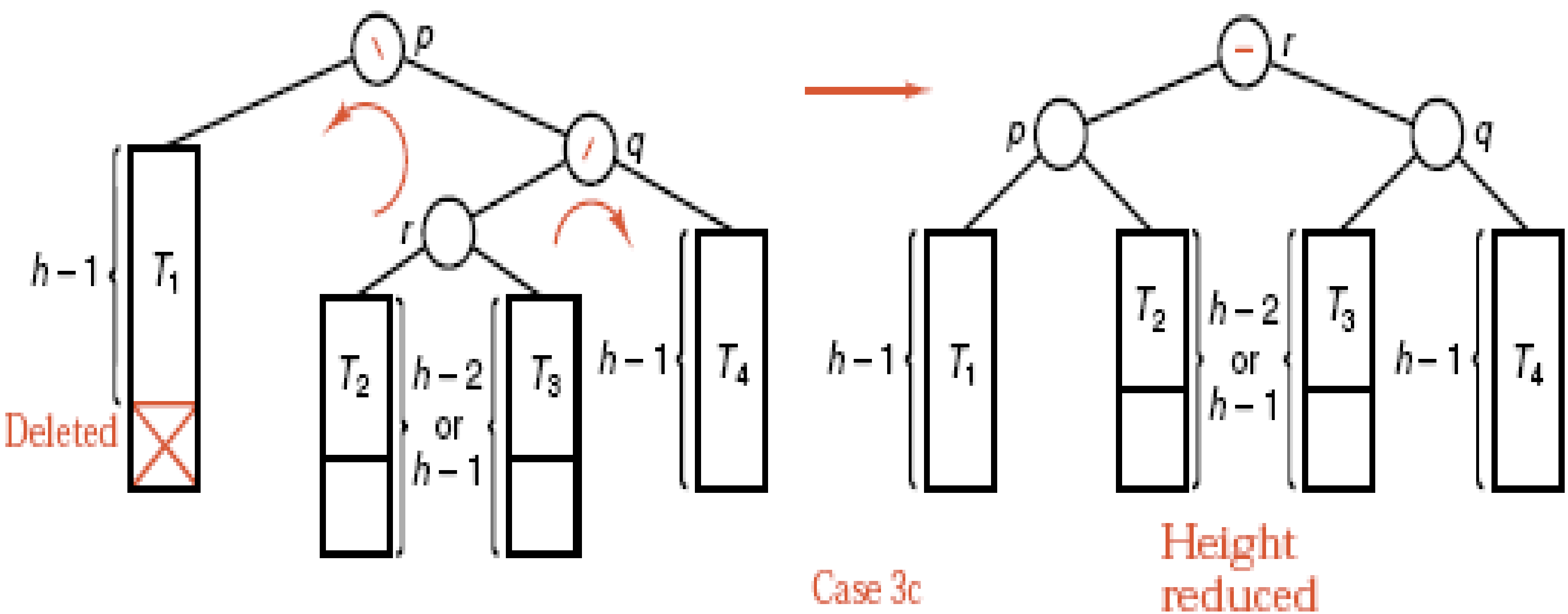
Case 3c: The balance factor of q and p are opposite.

Double rotation must be done (first around q, then around p).

The balance factor of the new root is equal.

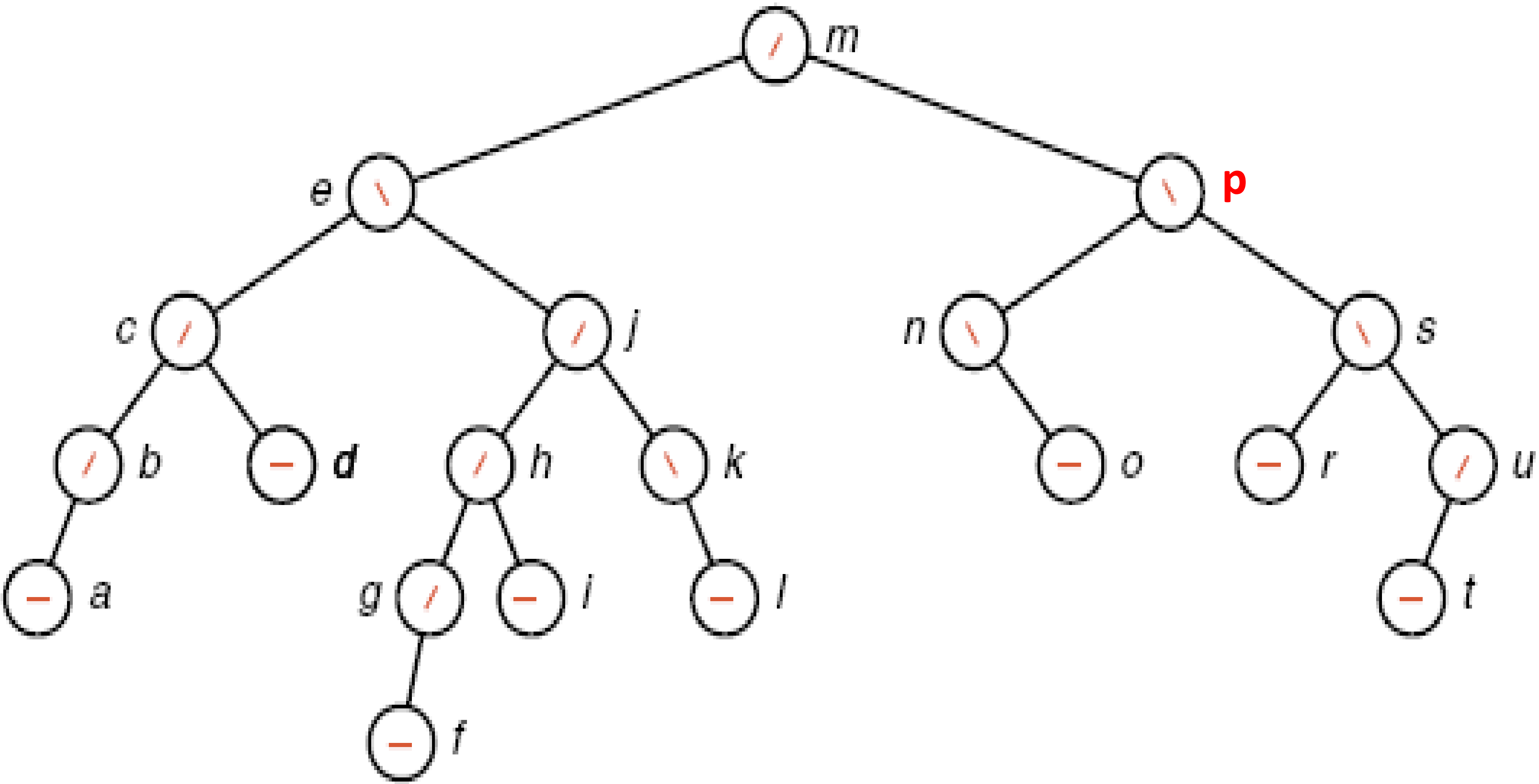
Other balance factors are set as appropriate.

shorter remains TRUE.



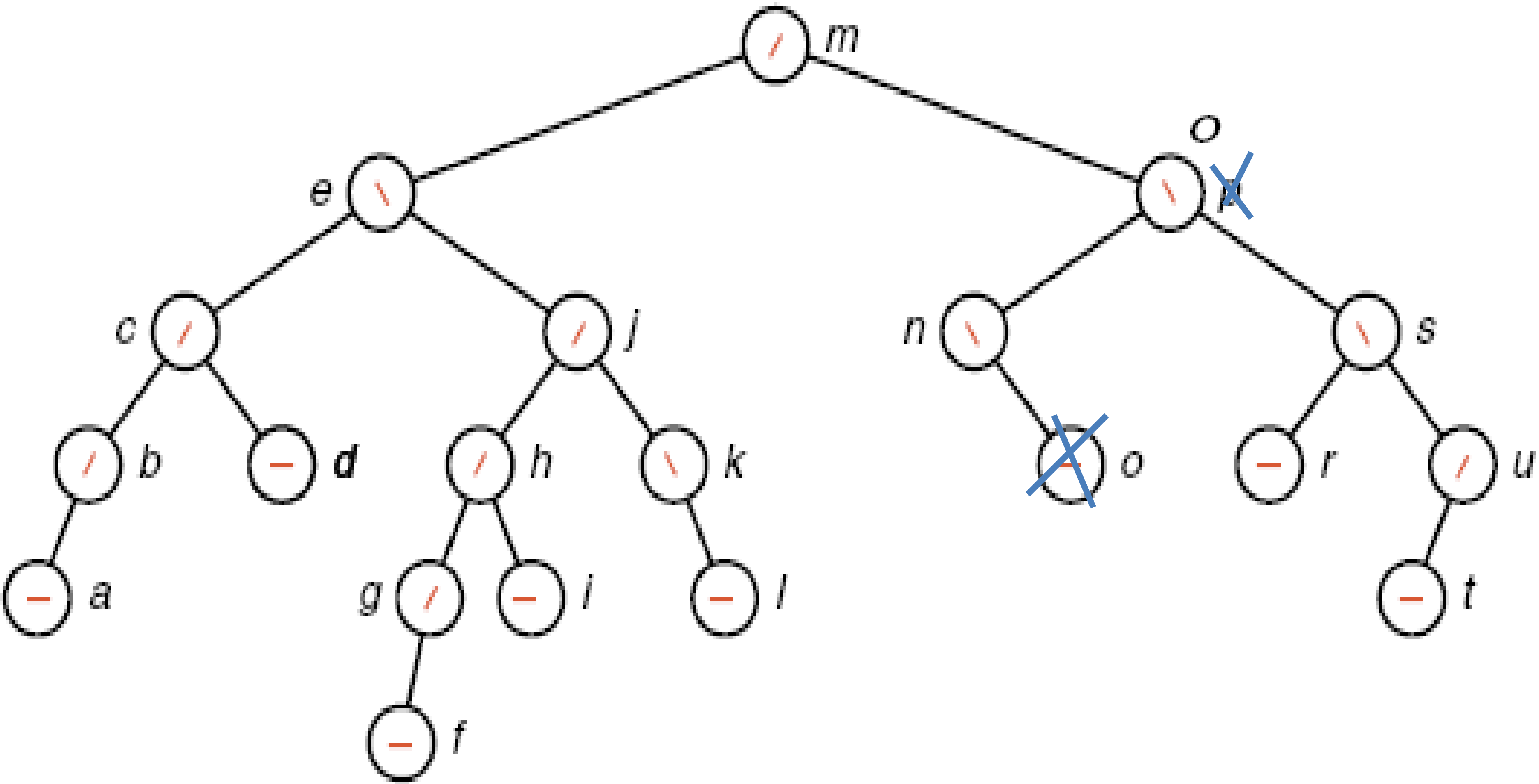
Removal of a node

Delete p



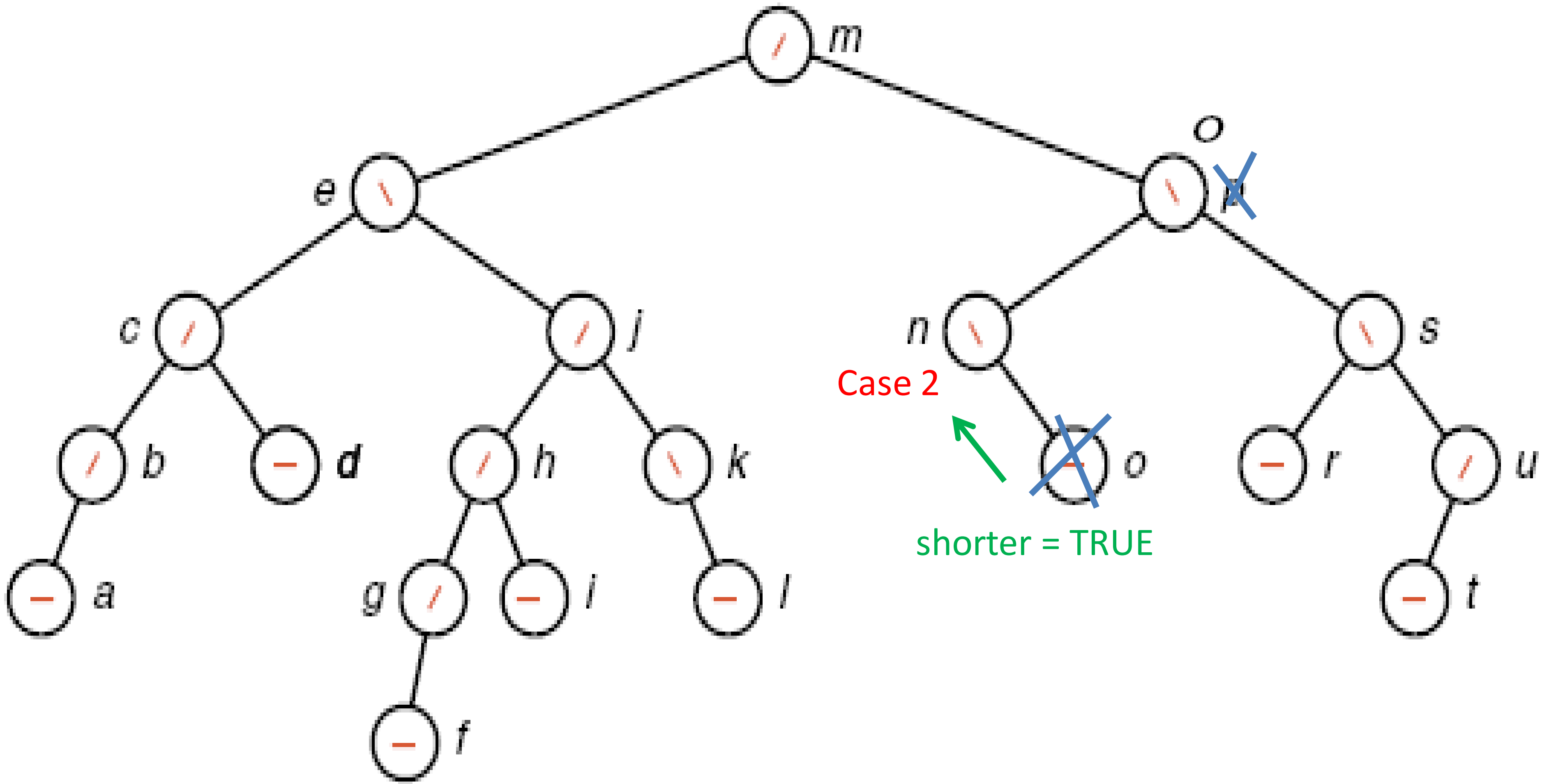
Removal of a node

Delete p



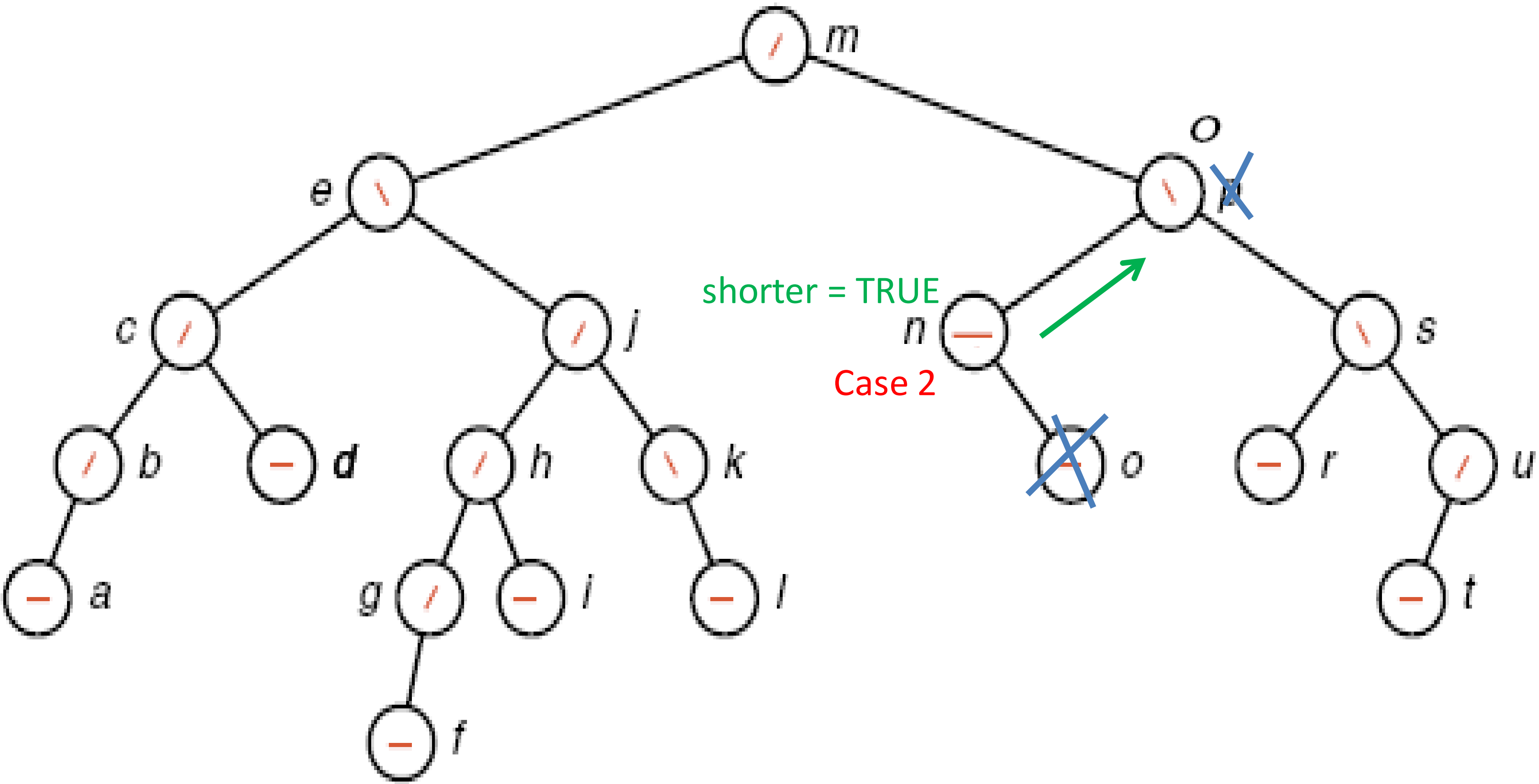
Removal of a node

Delete p



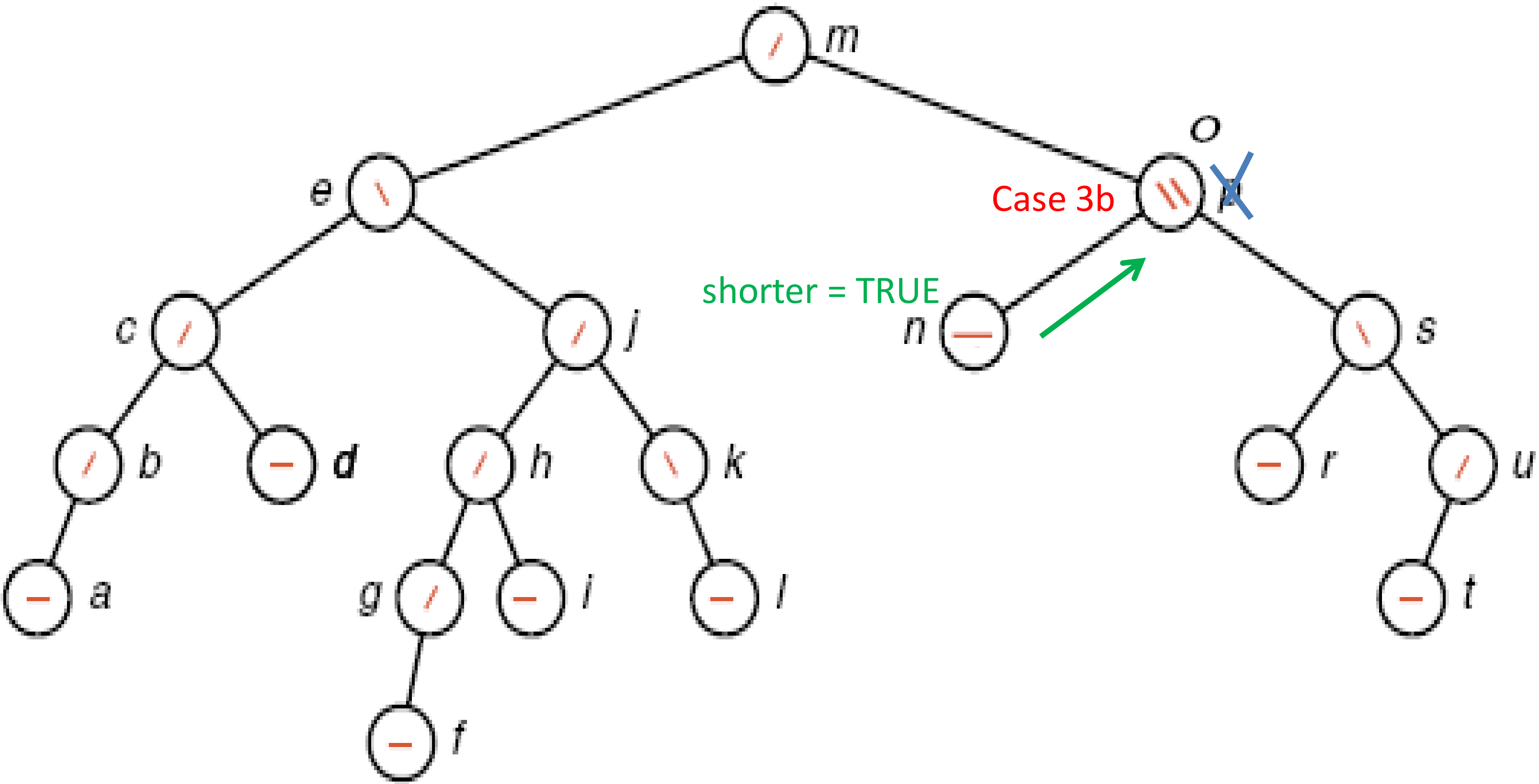
Removal of a node

Delete p



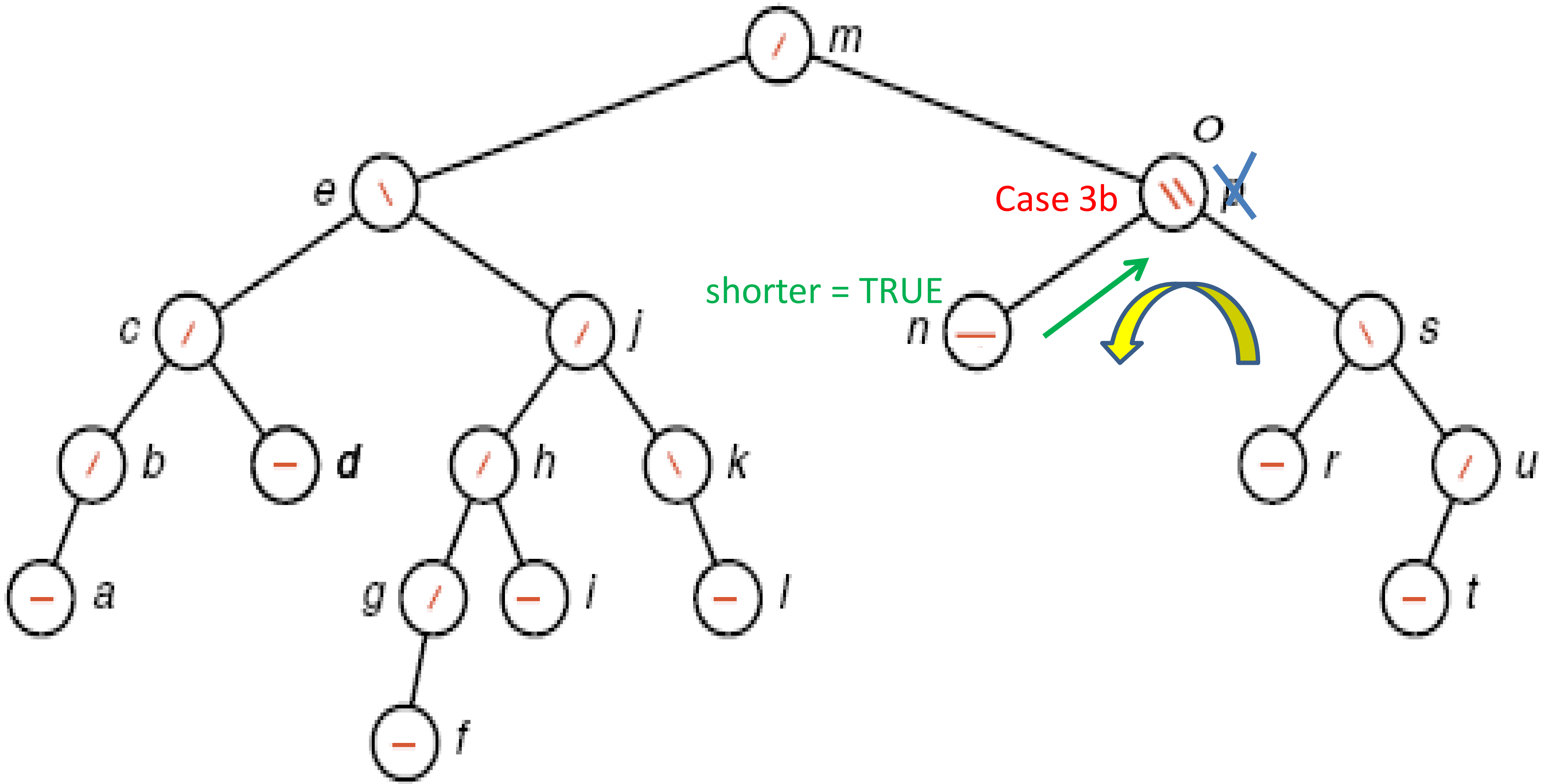
Removal of a node

Delete p



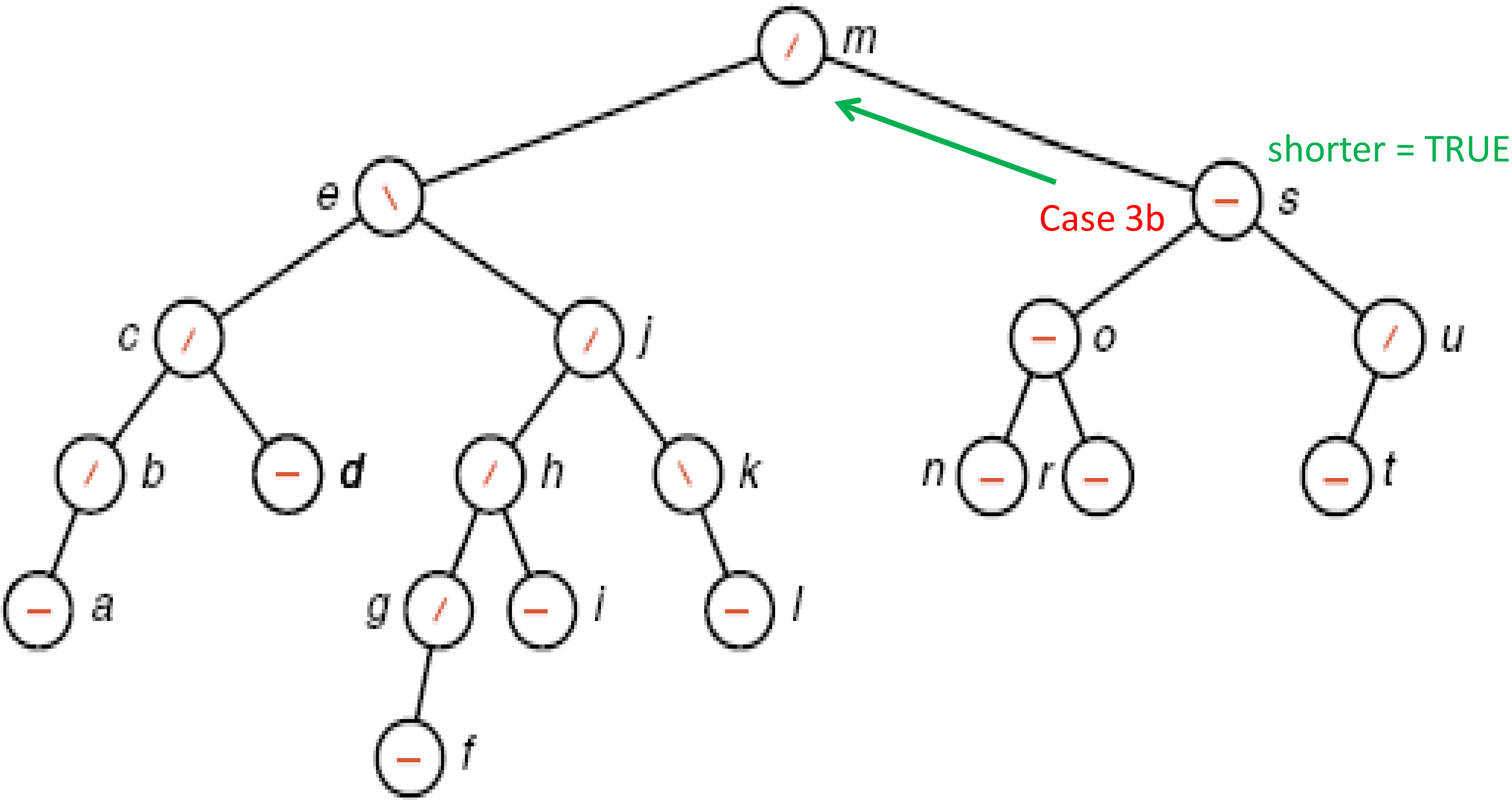
Removal of a node

Delete p



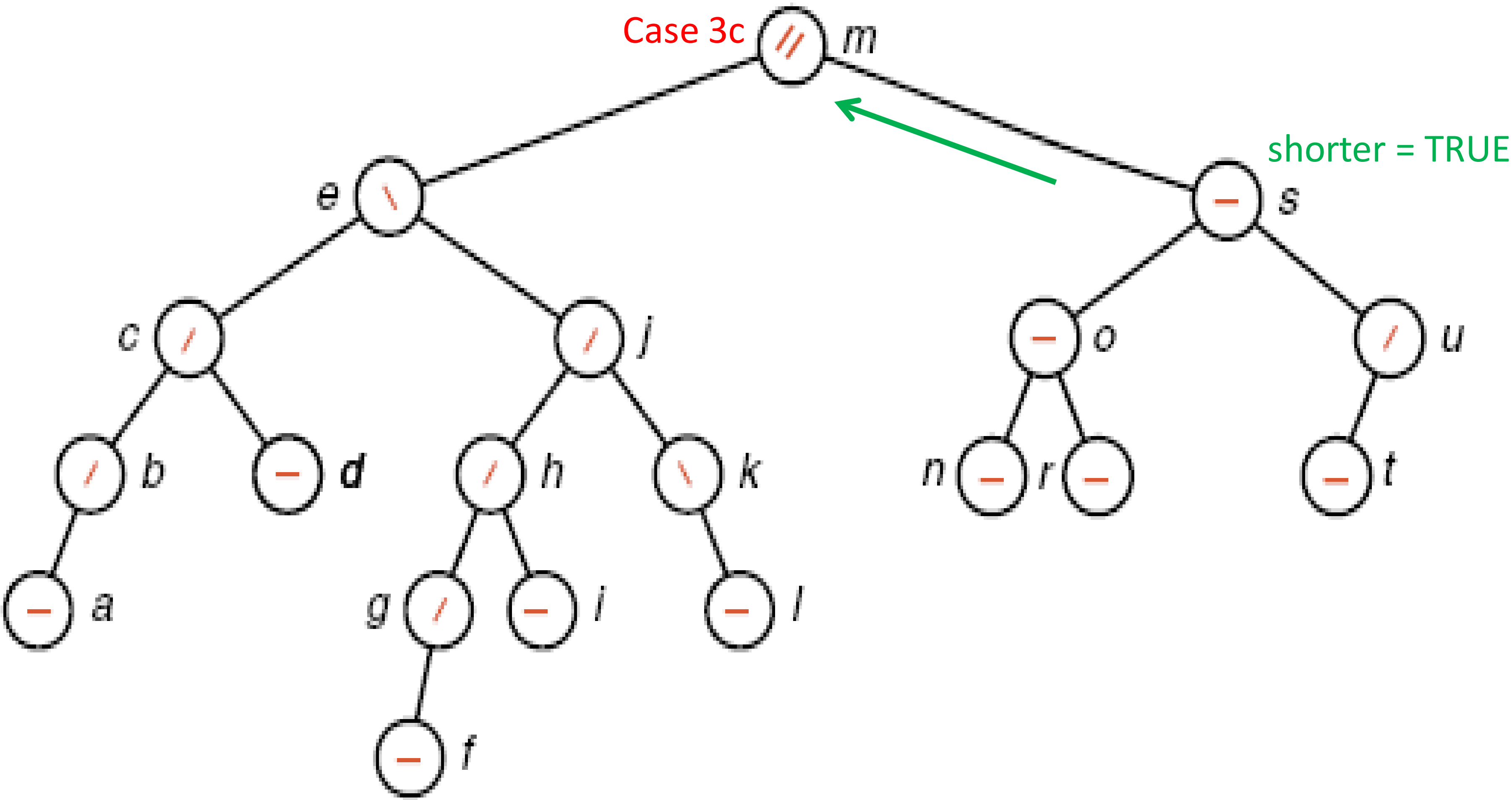
Removal of a node

Delete p



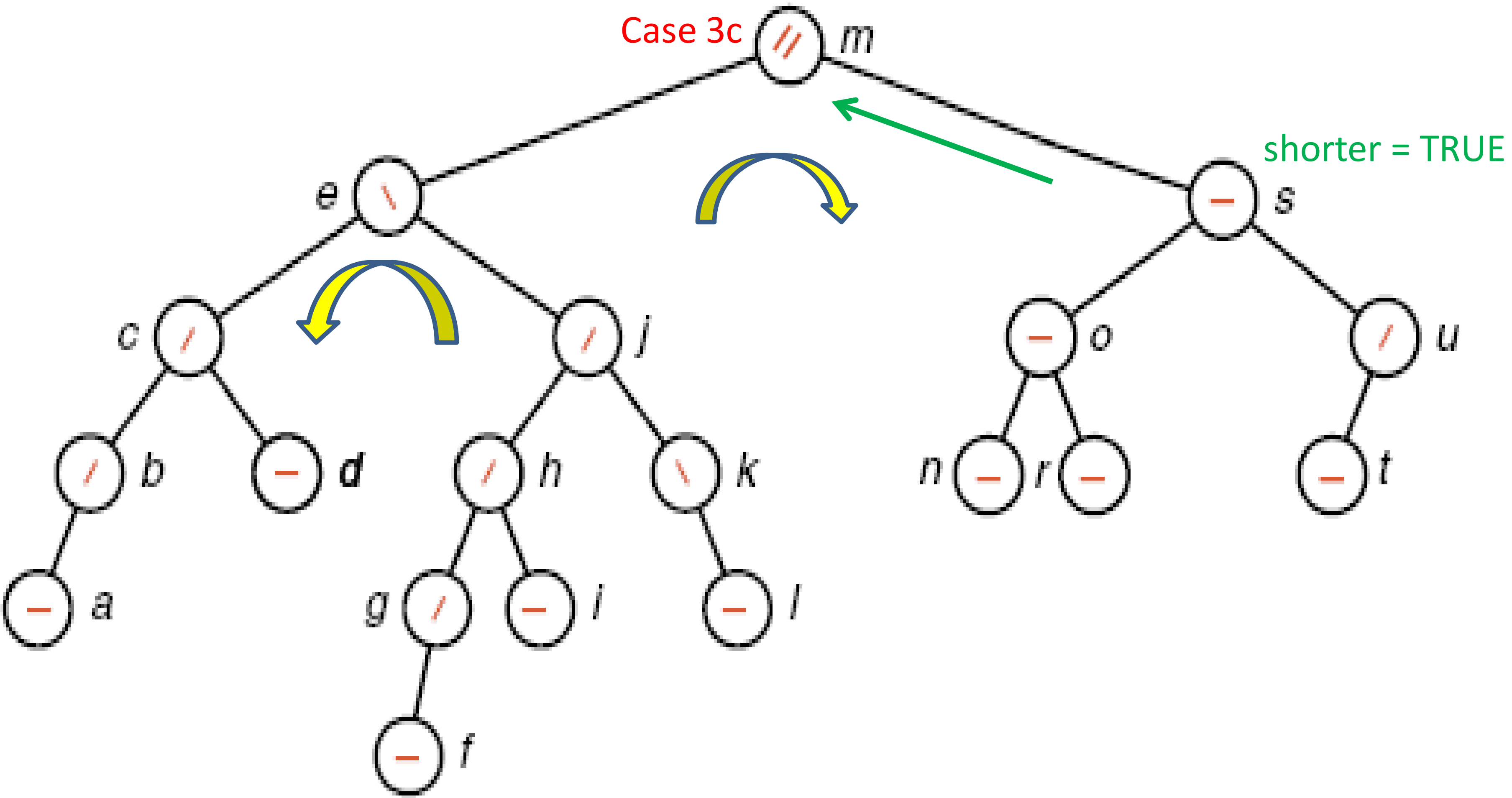
Removal of a node

Delete p



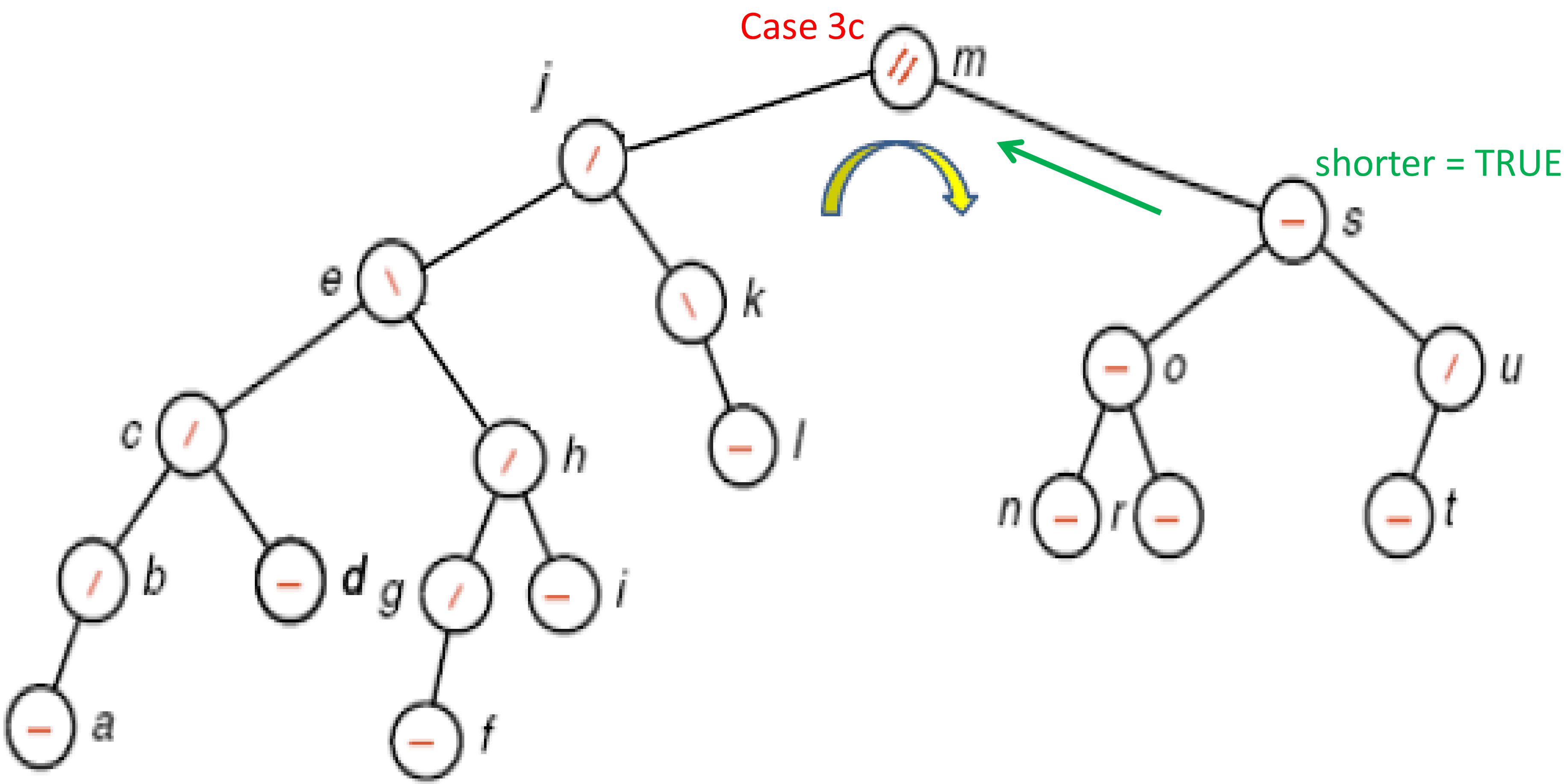
Removal of a node

Delete p



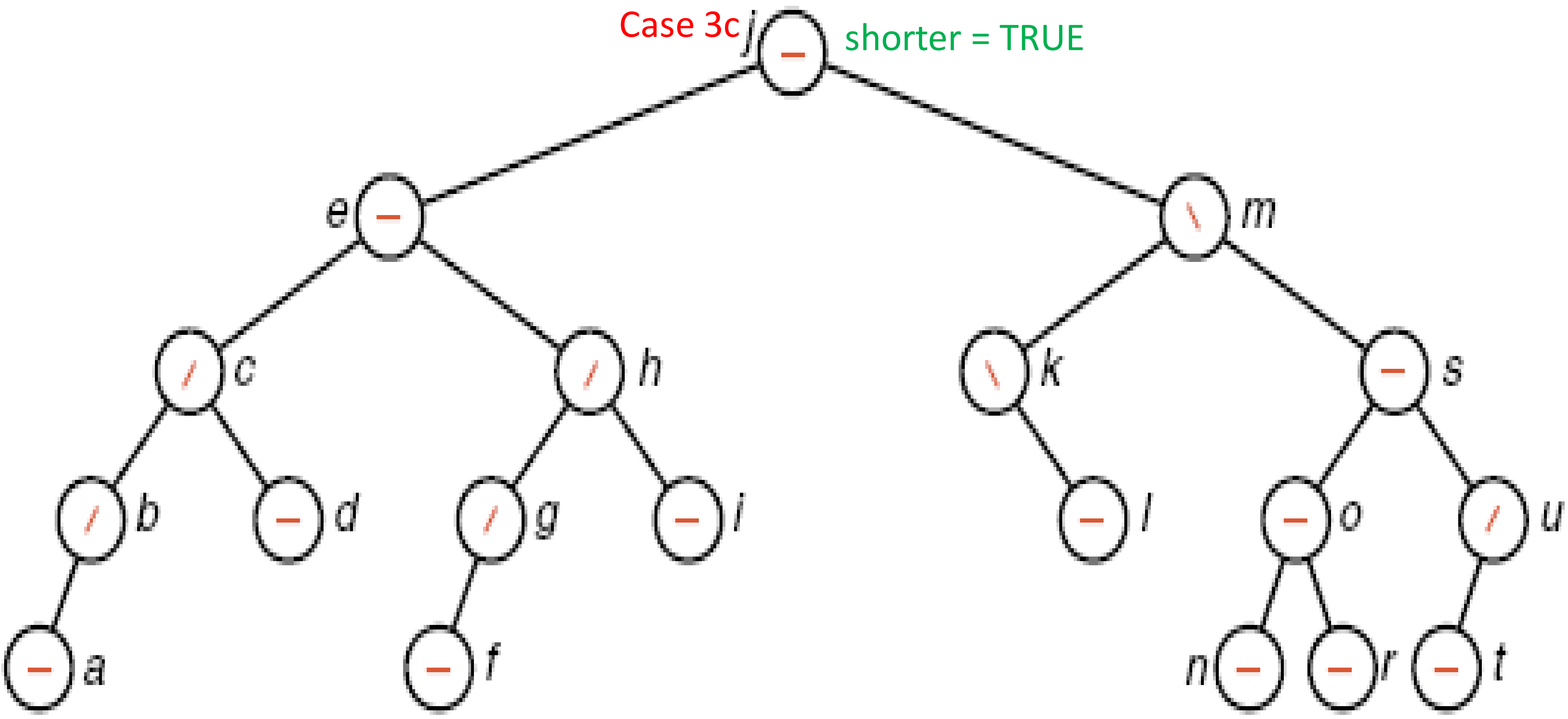
Removal of a node

Delete p



Removal of a node

Delete p



Analysis of AVL Tree

- The number of recursive calls to insert a new node can be as large as the height of the tree.

- At most one (single or double) rotation will be done per insertion.

- A rotation improves the balance of the tree, so later insertions are less likely to require rotations.

Analysis of AVL Tree

- It is very difficult to find the height of the average AVL tree, but the worst case is much easier.

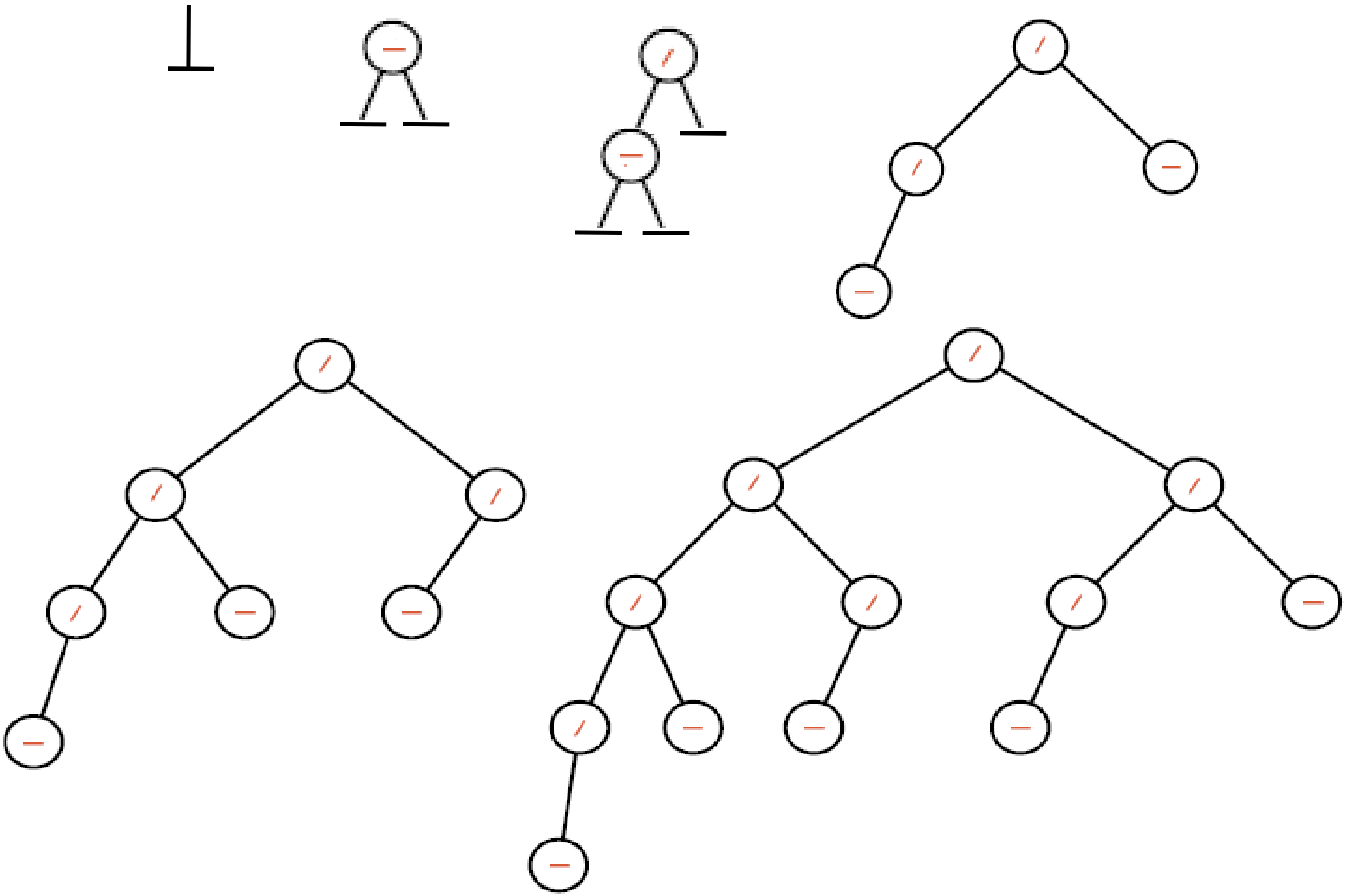
- The worst-case behavior of AVL trees is essentially no worse than the behaviour of random BST.

- The average behaviour of AVL trees is much better than that of random BST, almost as good as that which could be obtained from a perfectly balanced tree.

Analysis of AVL Tree

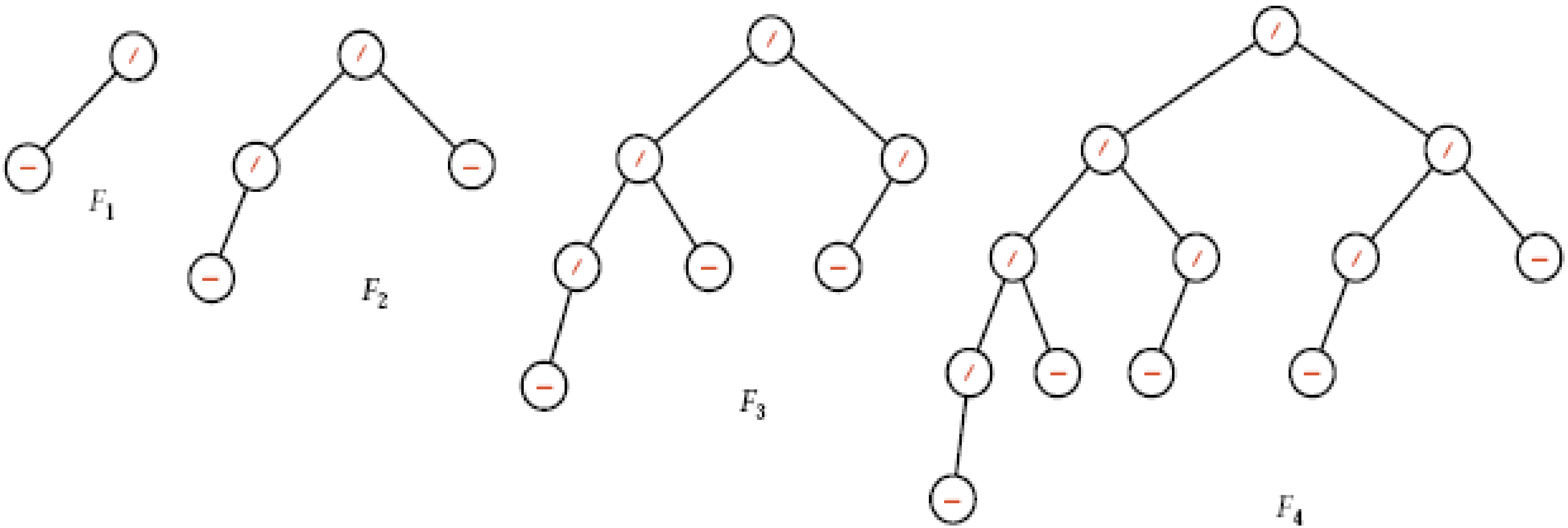
- To find the maximum height of AVL tree with n nodes, we instead find the minimum number of nodes that an AVL tree of height h can have.
  - **F_h**: an AVL tree of height h with minimum number of nodes.
  - **F_L**: a left subtree of height $h_L = h-1$ with minimum number of nodes.
  - **F_R**: a right subtree of height $h_R = h-2$ with minimum number of nodes.

Built sparse AVL trees



Fibonacci trees

- Trees, as sparse as possible for AVL tree, are call Fibonacci trees.



Analysis of AVL Tree

If |T| is the number of nodes in tree T, we have:

$$|F_h| = |F_{h-1}| + |F_{h-2}| + 1,$$

where $|F_0| = 1$ and $|F_1| = 2.$

And we can calculate

$$h \approx 1.44 \lg |F_h|$$

Analysis of AVL Tree

- The sparsest possible AVL tree with n nodes has height about 1.44 lg n compared to:
  - A perfectly balanced BST with n nodes has height about lg n.
  - A random BST, on average, has height about 1.39 lg n.
  - A degenerate BST has height as large as n.

Analysis of AVL Tree

- Hence the algorithm for manipulating AVL trees are guaranteed to take no more than about 44 percent more time than the optimum.

- In practice, AVL trees do much better than this on average, perhaps as small as lg n + 0.25.