AVL Trees

Manolis Koubarakis

AVL Trees

- We will now introduce **AVL trees** that have the property that they are kept almost balanced but not completely balanced. In this way we have $O(\log n)$ search time but also $O(\log n)$ insertion and deletion time in the worst case.

- AVL trees have been named after their inventors, Russian mathematicians Adelson-Velskii and Landis.

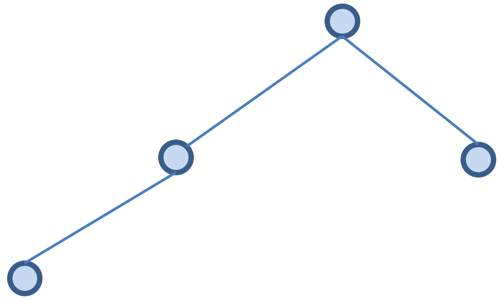
Definitions - Reminder

- We define the **height** of a binary search tree to be the length of the longest path from the root to some leaf.
- The height of a tree with **only one node** is 0. The height of the **empty tree** is defined to be -1.

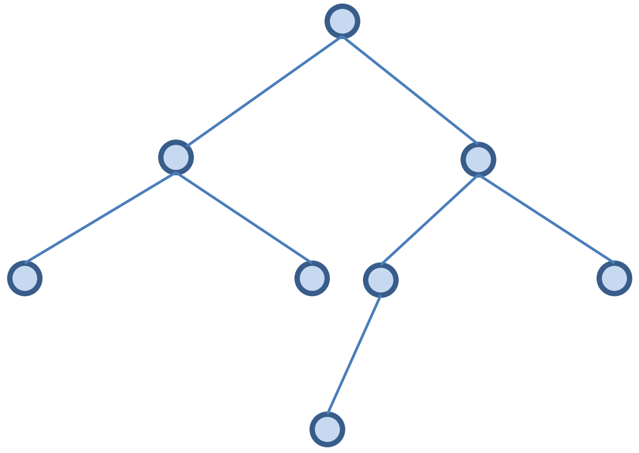
Definitions – AVL Trees

- If *N* is a node in a binary search tree *T*, then we say that *N* has the **AVL property** if the heights of the left and right subtrees of *N* are either equal or they differ by 1 (equivalently, they differ by at most 1).
- The AVL property is called **height-balance property (ιδιότητα ισορροπίας ύψους)** by some authors.
- An **AVL tree** is a binary search tree in which each node has the AVL property.

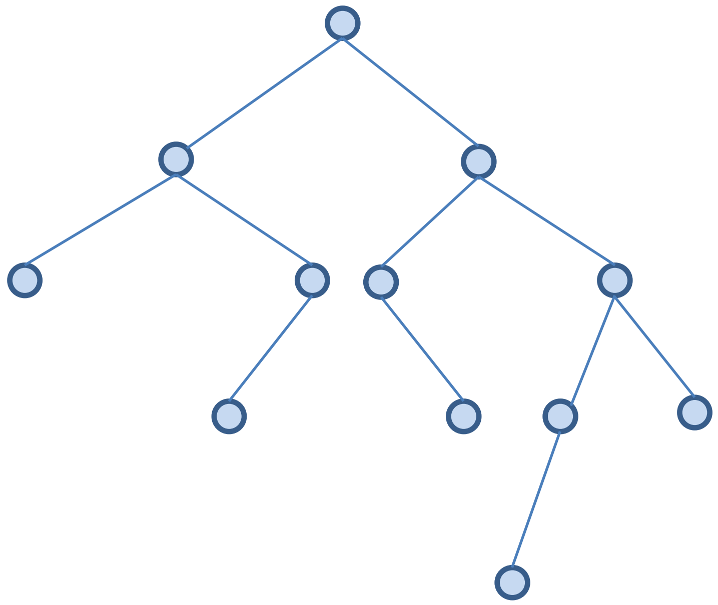
Example – AVL tree



Example – AVL tree



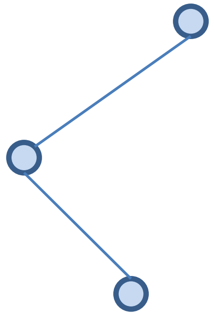
Example – AVL tree



Fact

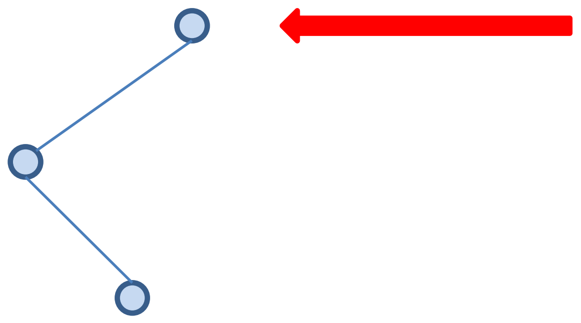
- It is easy to see that all the subtrees of an AVL tree are AVL trees.

Example – Non-AVL tree

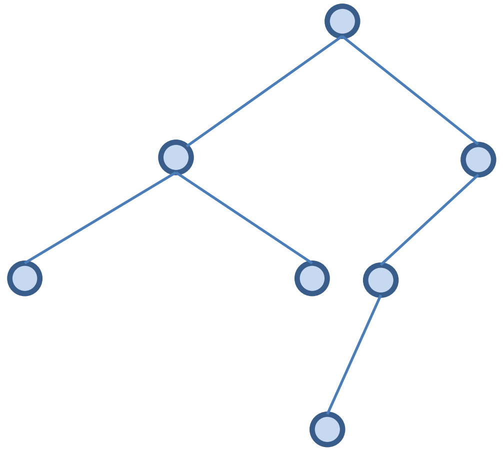


- Which nodes violate the AVL property?

Example (cont'd)

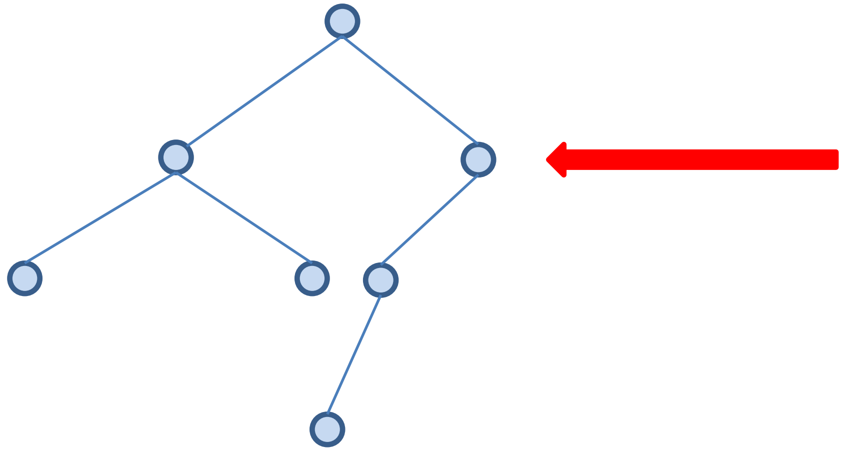


Example – Non-AVL tree

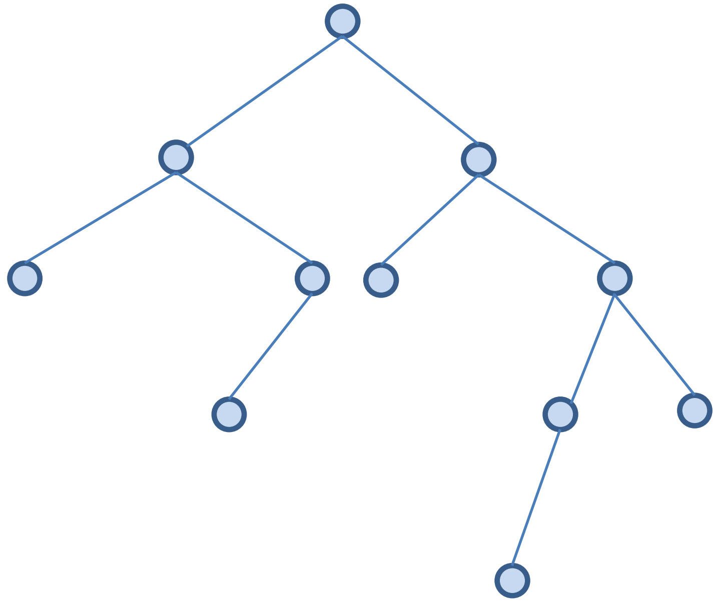


- Which nodes violate the AVL property?

Example (cont’d)

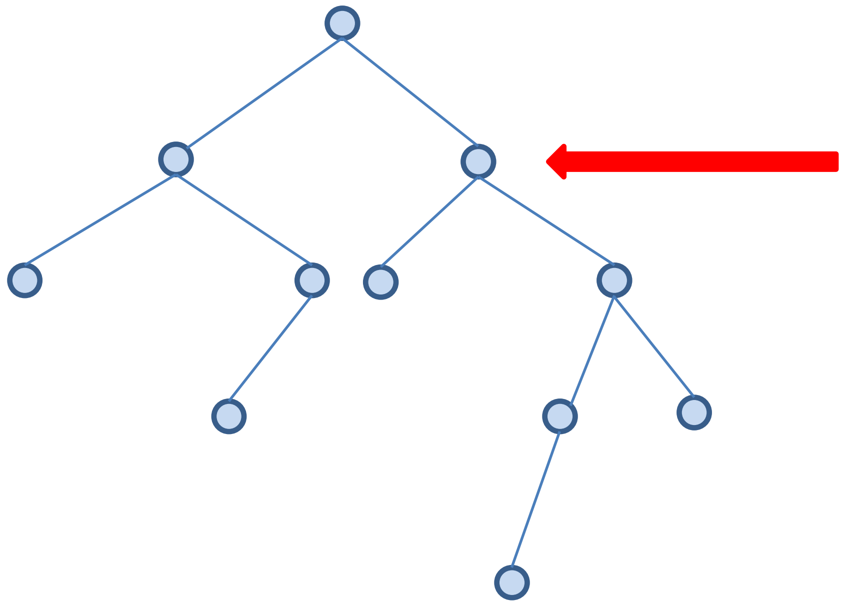


Example – Non-AVL tree



- Which nodes violate the AVL property?

Example (cont’d)

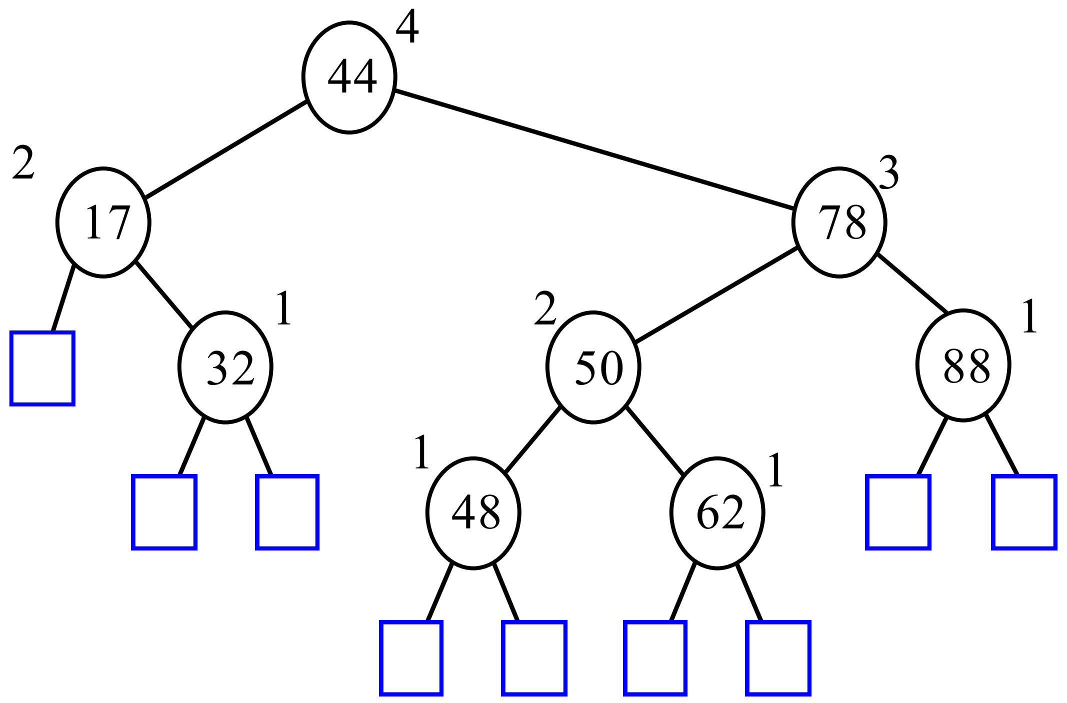


Extended AVL Trees

- If we consider trees in their **extended form** then **the AVL property needs to hold for internal nodes**.

- In the insertion and deletion algorithms given below, we will not show the trees in their extended form. It is easy to modify the algorithms to apply to that case.

Example



- In each node, we also show the height of the subtree rooted at that node.

Proposition

- The height of an AVL tree storing n keys is $O(\log n)$.
- In other words, the AVL property has the important consequence of **keeping the height small of an AVL tree small!**
- Proof?

Proof

- In the proof we assume that the tree is in its **extended form**.
- Instead of trying to find an upper bound for the height of an AVL tree directly, we will find a **lower bound on the minimum number of internal nodes $\boldsymbol{n(h)}$** of an AVL tree with height h. From this, it will be easy to derive our result.
- Notice that $n(1) = 1$ because an AVL tree of height 1 must have at least one internal node.
- Similarly, $n(2) = 2$ because an AVL tree of height 2 must have at least two internal nodes.

Proof (cont'd)

- An AVL tree of height $h \geq 3$ with the minimum number of internal nodes is such that both subtrees of the root are AVL trees with the minimum number of internal nodes: one with height $h-1$ and one with height $h-2$.
- Taking the root into account, we obtain the following formula:

$$n(h) = 1 + n(h-1) + n(h-2).$$

Proof (cont'd)

- The previous formula implies that $n(h)$ is a strictly increasing function of h.
- Thus, we know that $n(h-1) > n(h-2)$.
- Replacing $n(h-1)$ with $n(h-2)$ in the formula of the previous slide and dropping the 1, we get that for $h \geq 3$,

$$n(h) > 2\,n(h-2).$$

Proof (cont’d)

- The previous formula shows that $n(h)$ at least doubles each time h increases by 2, which intuitively means that $\boldsymbol{n(h)}$ **grows exponentially.**
- To show this formally, we apply the formula of the previous slide repeatedly, yielding the following series of inequalities:

$$\begin{aligned} n(h) &> 2\, n(h-2) \\ &> 4\, n(h-4) \\ &> 8\, n(h-6) \\ &\vdots \\ &> 2^i\, n(h-2i) \end{aligned}$$

Proof (cont'd)

- That is, $n(h) > 2^i n(h-2i)$, for any integer i such that $h - 2i \geq 1$.
- Since we already know the values of $n(1)$ and $n(2)$, we pick i so that $h - 2i$ is equal to either 1 or 2. That is, we pick $i = \left\lceil \frac{h}{2} \right\rceil - 1$.
- By substituting the value of i in the formula above, we obtain, for $h \geq 3$,

$$\begin{aligned} n(h) &> 2^{\left\lceil \frac{h}{2} \right\rceil - 1} n\left(h - 2\left\lceil \frac{h}{2} \right\rceil + 2\right) \\ &\geq 2^{\left\lceil \frac{h}{2} \right\rceil - 1} n(1) \\ &\geq 2^{\frac{h}{2} - 1}. \end{aligned}$$

Proof (cont'd)

- By taking logarithms of both sides of the previous formula, we obtain

$$\log n(h) > \frac{h}{2} - 1.$$

- This is equivalent to $h < 2\ \log n(h) + 2$.
- This implies that an AVL tree storing n keys has height at most $2 \log n + 2$.

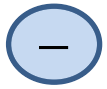
Definitions

- We will say that a node of an AVL tree is **left-higher** if the height of its left subtree is 1 plus the height of its right subtree.
- We will say that a node of an AVL tree is **right-higher** if the height of its right subtree is 1 plus the height of its left subtree.

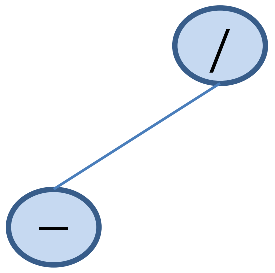
Notation

- In drawing trees, we shall show a **left-higher** node by “/”, a node whose balance factor (παράγοντας ισοζύγισης) is **equal** by “—”, and a **right-higher** node by “\”.
- We will use notation “//” or “\\” for nodes that **do not have the AVL property,** and they have longer paths on the left or right respectively.

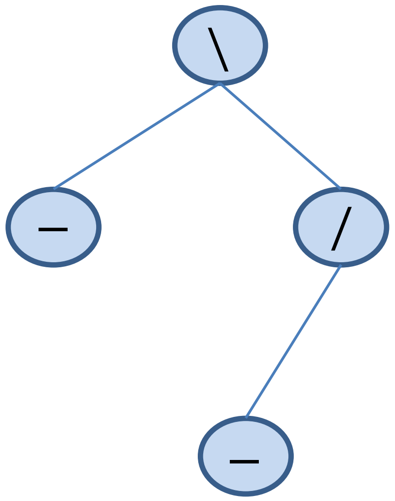
Example AVL Tree



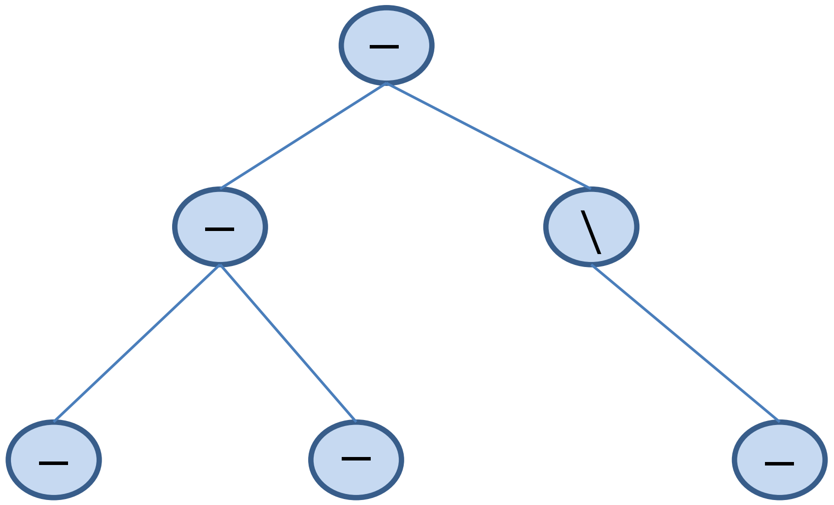
Example AVL Tree



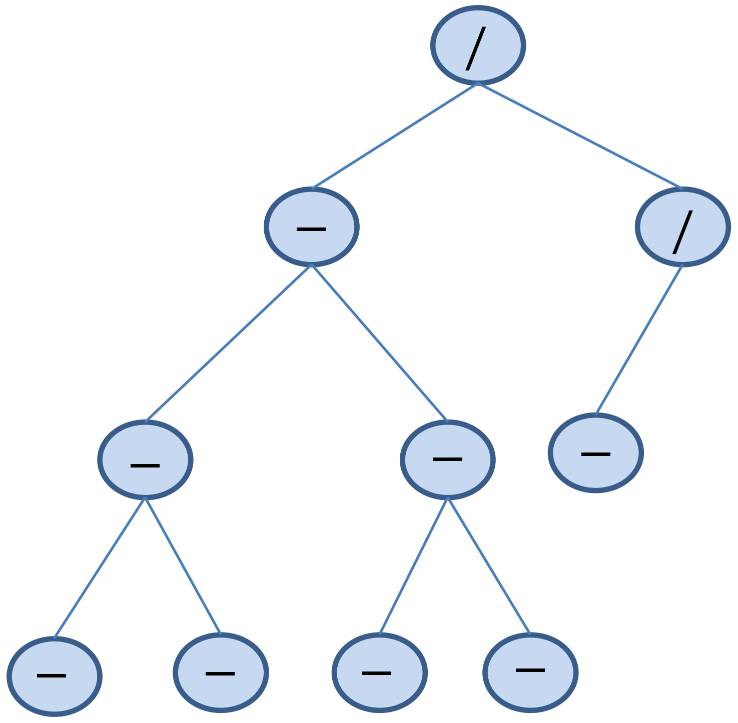
Example AVL Tree



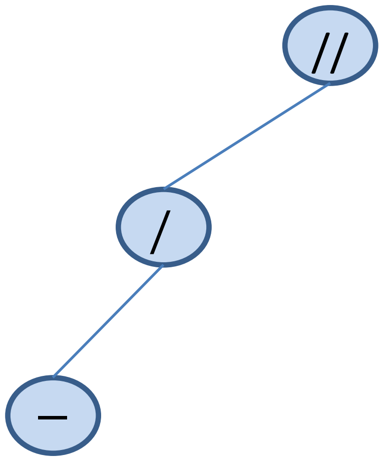
Example AVL Tree



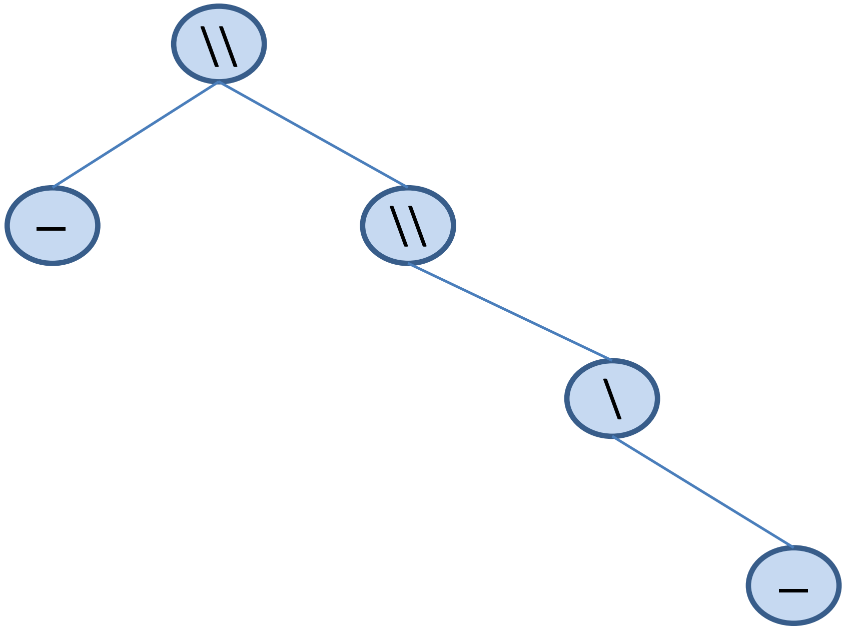
Example AVL Tree



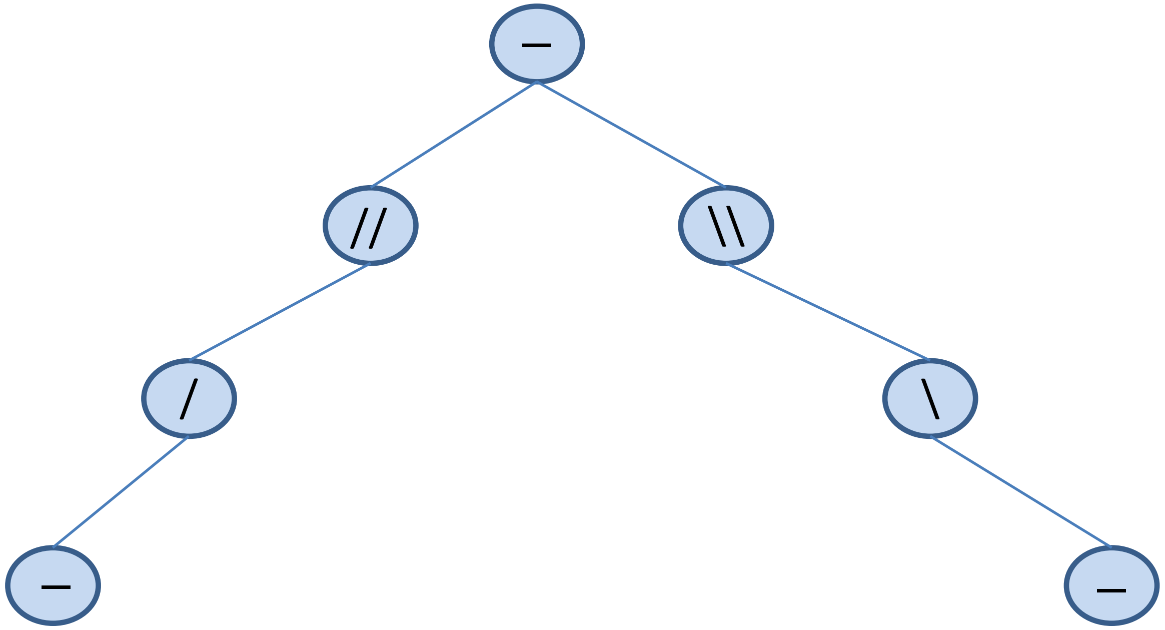
Example Non-AVL Tree



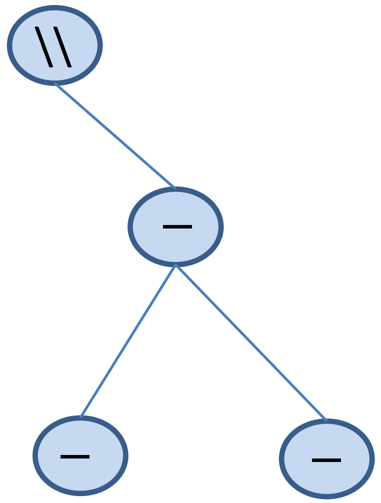
Example Non-AVL Tree



Example Non-AVL Tree



Example Non-AVL Tree



Keeping Track of Balance Factors

- By adding a new member to each node of an AVL tree, we can keep track of whether the left and right subtree are of equal height, or whether one is higher than the other.

```
typedef enum {LeftHigh, Equal, RightHigh} BalanceFactor;

typedef struct AVLTreeNodeTag {
            BalanceFactor BF;
            KeyType       Key;
            struct AVLTreeNodeTag *LLink;
            struct AVLTreeNodeTag *RLink;
        } AVLTreeNode;
```

Rebalancing an AVL Tree

- In the insertion and deletion algorithm for AVL trees that we will present now, it is possible that the AVL property will be lost at some point.
- In this case we apply to the tree some shape-changing transformations to restore the AVL property. These transformations are the **rotations** we have introduced in the previous lecture.

Insertion in an AVL Tree

- If a node N is “-” and we insert a new node in its left or right subtree then the AVL tree property at node N is not lost and N becomes “/” or “\” respectively.
- If a node N is “/” and we insert a node in its right subtree (i.e., its shorter subtree) then the AVL tree property at node N is not lost and N becomes “-”.
- If a node N is “\” and we insert a node in its left subtree (i.e., its shorter subtree) then the AVL tree property at node N is not lost and N becomes “-”.

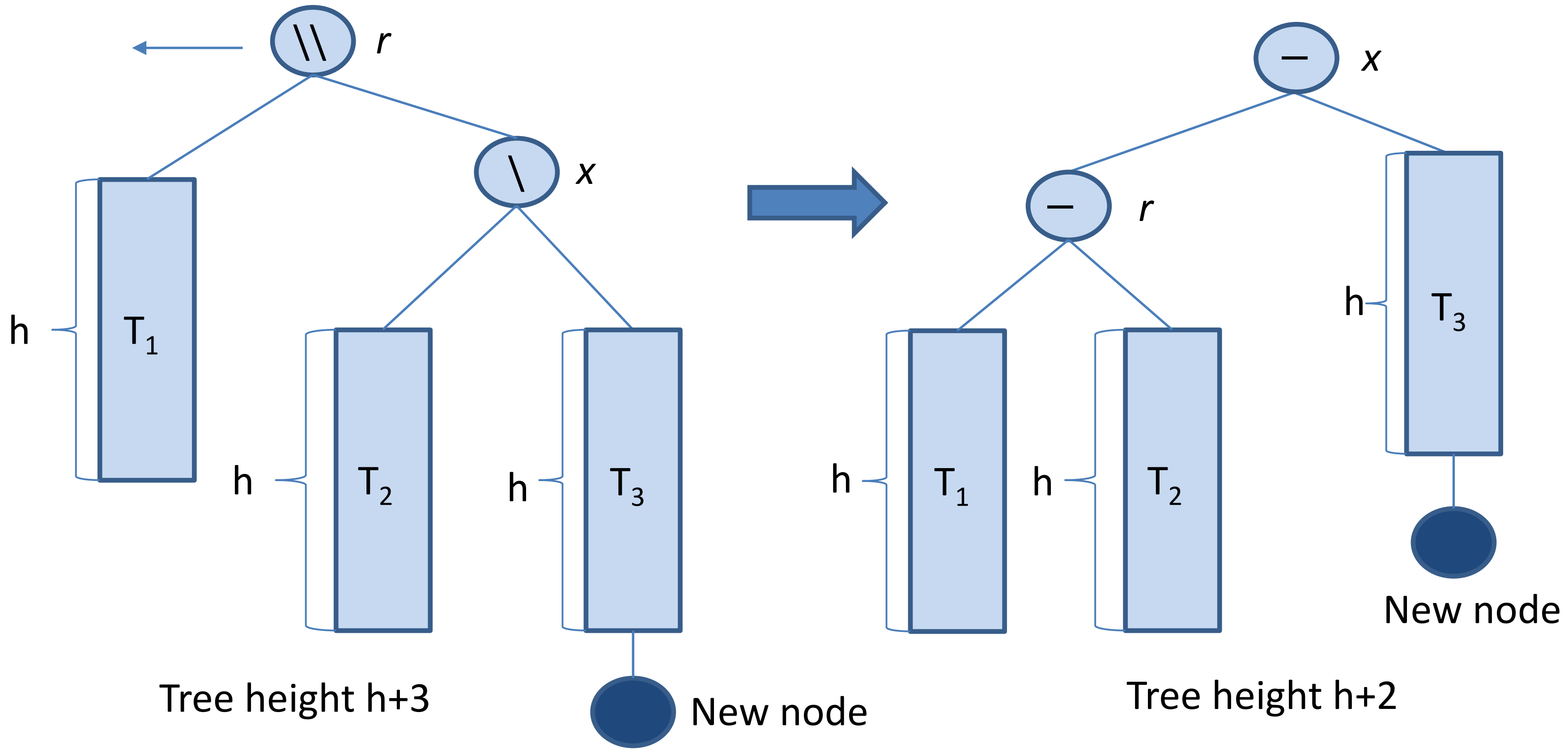
Insertion (cont’d)

- Let us consider the case when a new node has been inserted into the **taller subtree** of a node and its height has increased, so that now one subtree has height 2 more than the other, and the node no longer satisfies the AVL property.
- Let us assume **we have inserted the new node into the right subtree** of node *r*, its height has increased, and *r* previously was **right higher** (so now it will become “\\”).
- So, *r* is the node where the AVL property was lost and let *x* be the root of its right subtree. Then there are **three cases** to consider depending on the balance factor of *x*.

Insertion (cont’d)

- **Case 1: *x* is right higher**. Therefore, the new node was inserted in the right subtree of *x*. Then, we can do a **single left rotation** that restores the AVL property as shown on the next slide.
- We have rotated the node x upward to the root, dropping *r* down into the left subtree of *x.* The subtree T_2 of nodes with keys between those of *r* and *x* now becomes the right subtree of *r*.
- Note that in the tallest subtree we had height h+2, then height h+3 when the new node was inserted, then height h+2 again when the AVL property was restored. Thus, **there are no further height increases in the tree** that would force us to examine nodes other than *r*.
- Note that r was the closest ancestor of the inserted node where the AVL property was lost. We do not need to consider any other nodes higher than *r*.

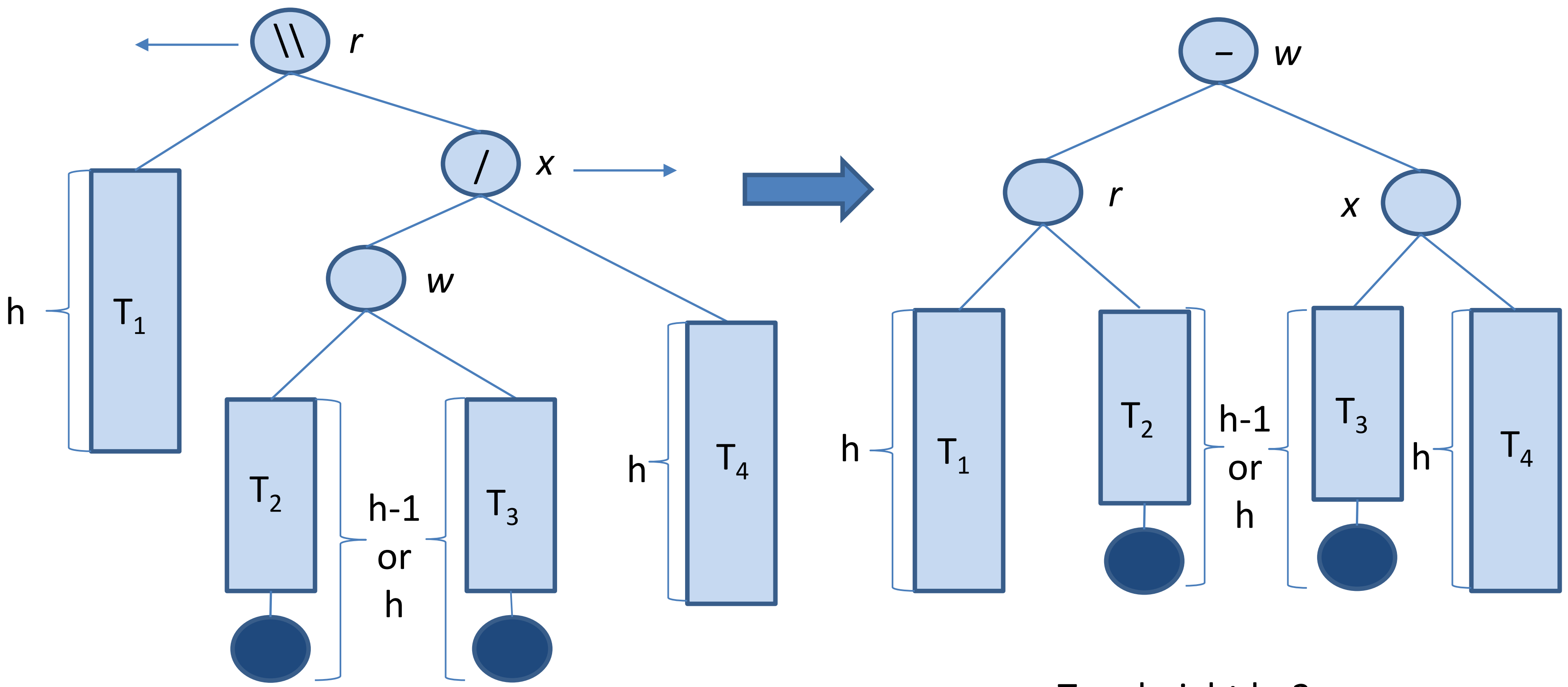
Single Left Rotation at *r*



Insertion (cont'd)

- **Case 2: *x* is left higher**. Therefore, the new node was inserted in the left subtree of *x*. In this case, we have to move down two levels to the node *w* that roots the left subtree of *x*, to find the new root of the local tree where the rotation will take place.
- This is called **double right-left rotation** because the transformation can be obtained in two steps by first rotating the subtree with root *x* to the right (so that *w* becomes the root), and then rotating the tree with root *r* to the left (moving w up to become the new root).
- Note that **after the rotation the heights have been restored** to h+2 as they were before the rotation, **so no other nodes of the tree need to be considered.**
- Some authors call this rotation double left rotation. The term double right-left that we use is more informative.

Double Right-Left Rotation at *x* and *r*



One of T_2 or T_3 has the new node and height h
Tree height h+3

Tree height h+2

Insertion (cont'd)

- In this case, the new balance factors of *r* and *x* depend on the balance factor of *w* after the node was inserted. The diagram shows the subtrees of *w* as having equal heights, but it is possible that *w* may be either left or right higher. The resulting balance factors are as follows:

Old *w*	New *r*	New *x*
–	–	–
/	–	\
\	/	–

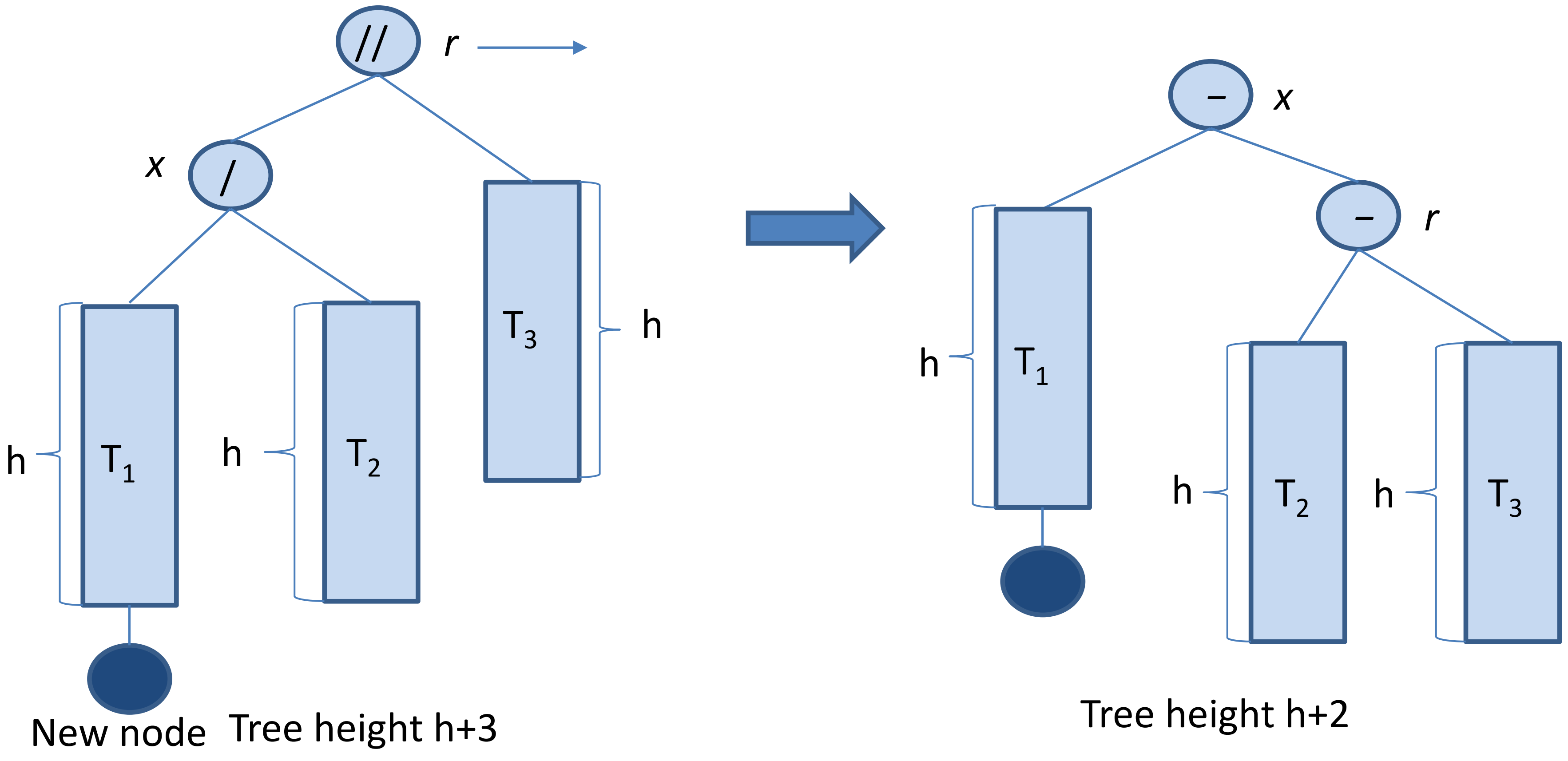
Insertion (cont’d)

- **Case 3: Equal Height**. This case cannot happen.
- Remember that we have just inserted a new node into the subtree rooted at *x*, and this subtree now has height 2 more than the left subtree of the root. The new node went either into the left or the right subtree of *x*. Hence its insertion increased the height of only one subtree of *x*. If these subtrees had equal heights after the insertion, then the height of the full subtree rooted at *x* was not changed by the insertion, contrary to what we already know.

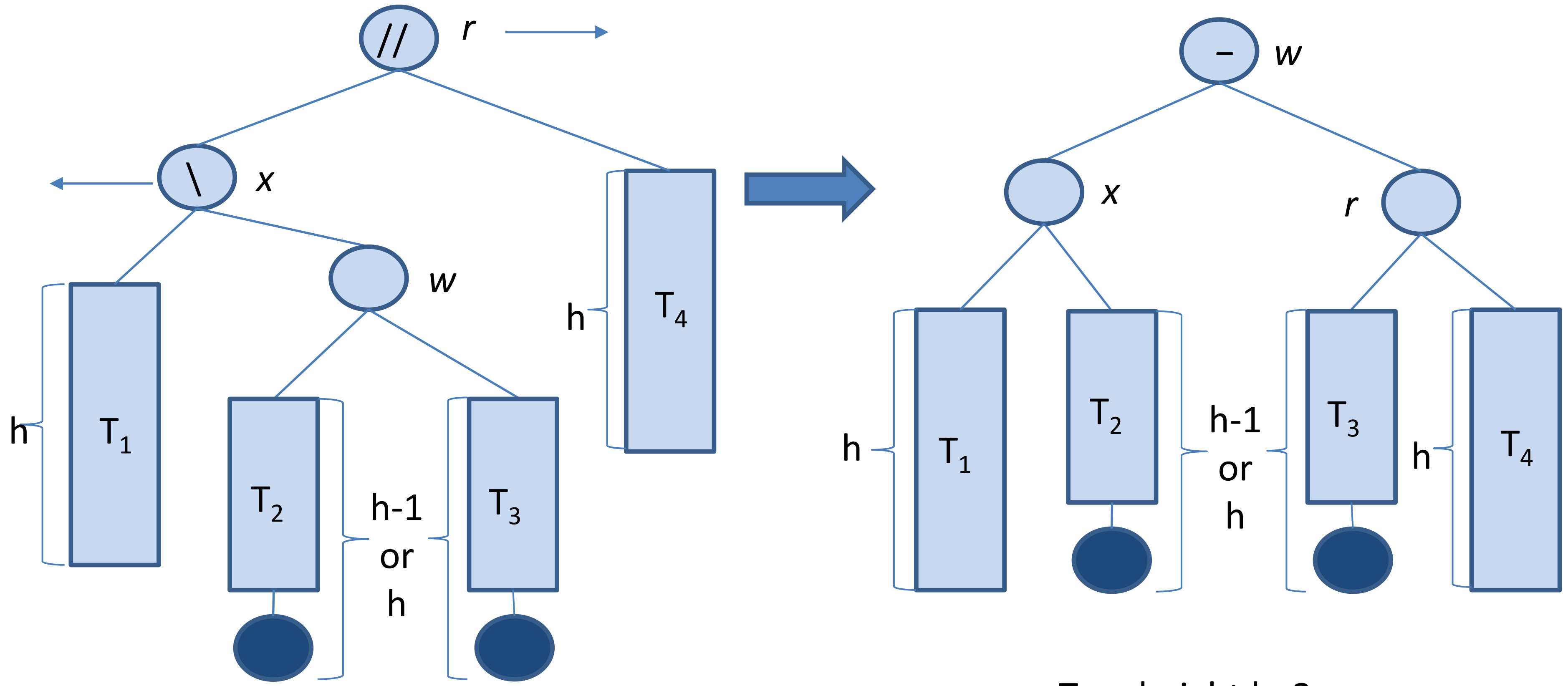
Insertion (cont'd)

- Let us now consider the case **symmetric** to the one we considered so far: *r* was left higher and we introduced the new node in the left subtree of *r*.
- In this case we will use **single right rotation** and **double left-right rotation** to restore the AVL property.

Single Right Rotation at *r*



Double Left-Right Rotation at *x* and *r*



One of T_2 or T_3 has the new node and height h
Tree height h+3

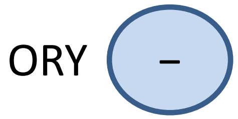
Tree height h+2

Rotations are Local

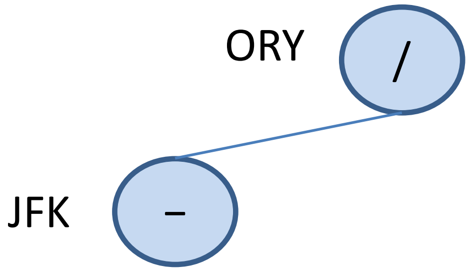
- Rotations are done only when the height of a subtree has increased. After the rotations, the increase in height has been removed so **no further rotations or changes of balance factors are done.**
- So, the AVL property is restored with a **single or a double rotation.**

Example: Building an AVL Tree

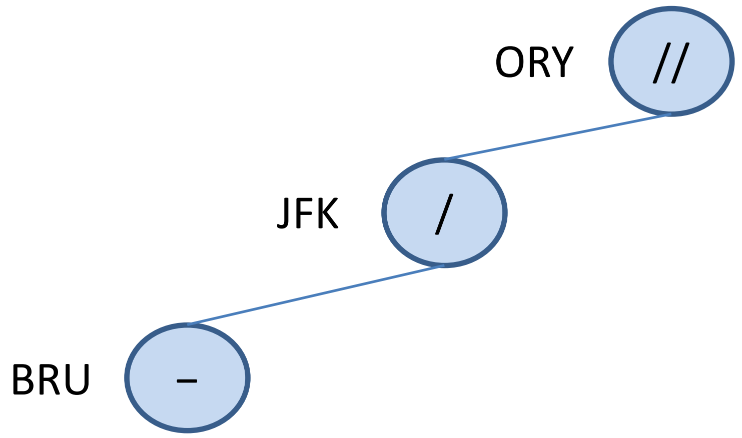
- Insert ORY



Insert JFK

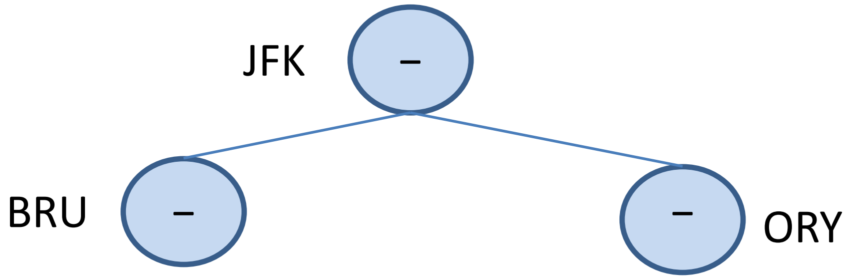


Insert BRU

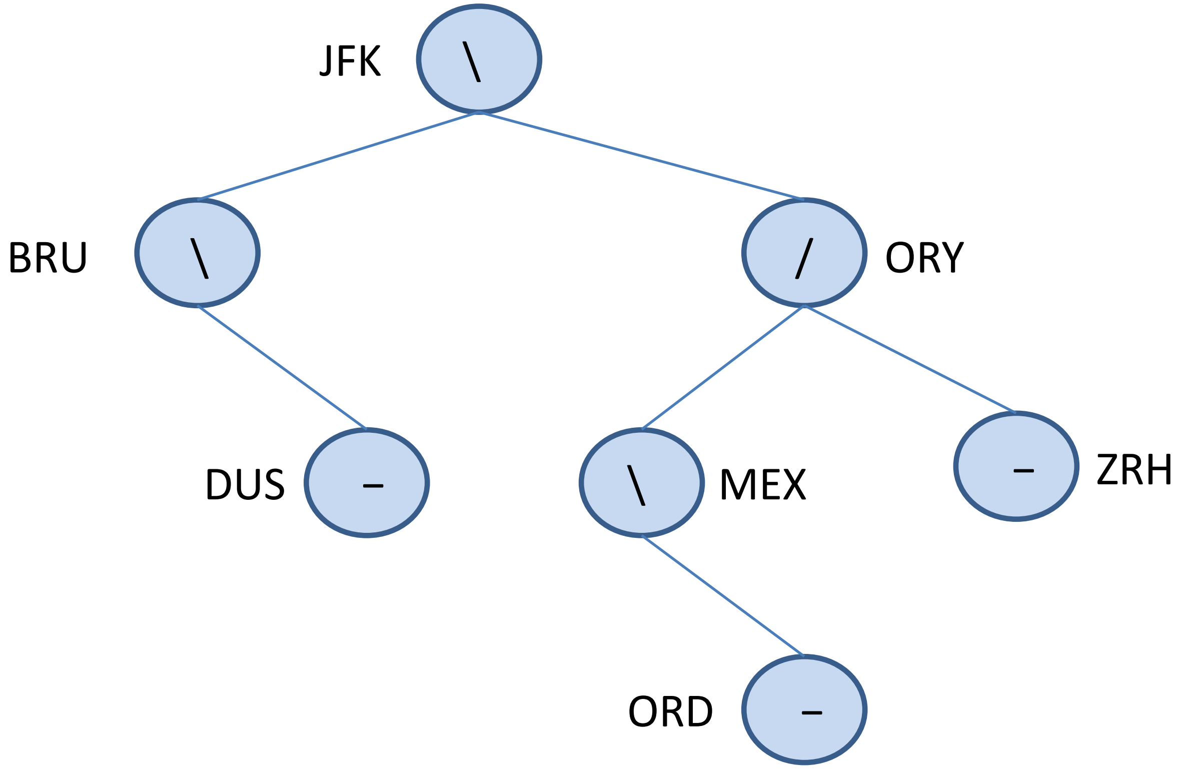


- The AVL property is violated at node ORY.

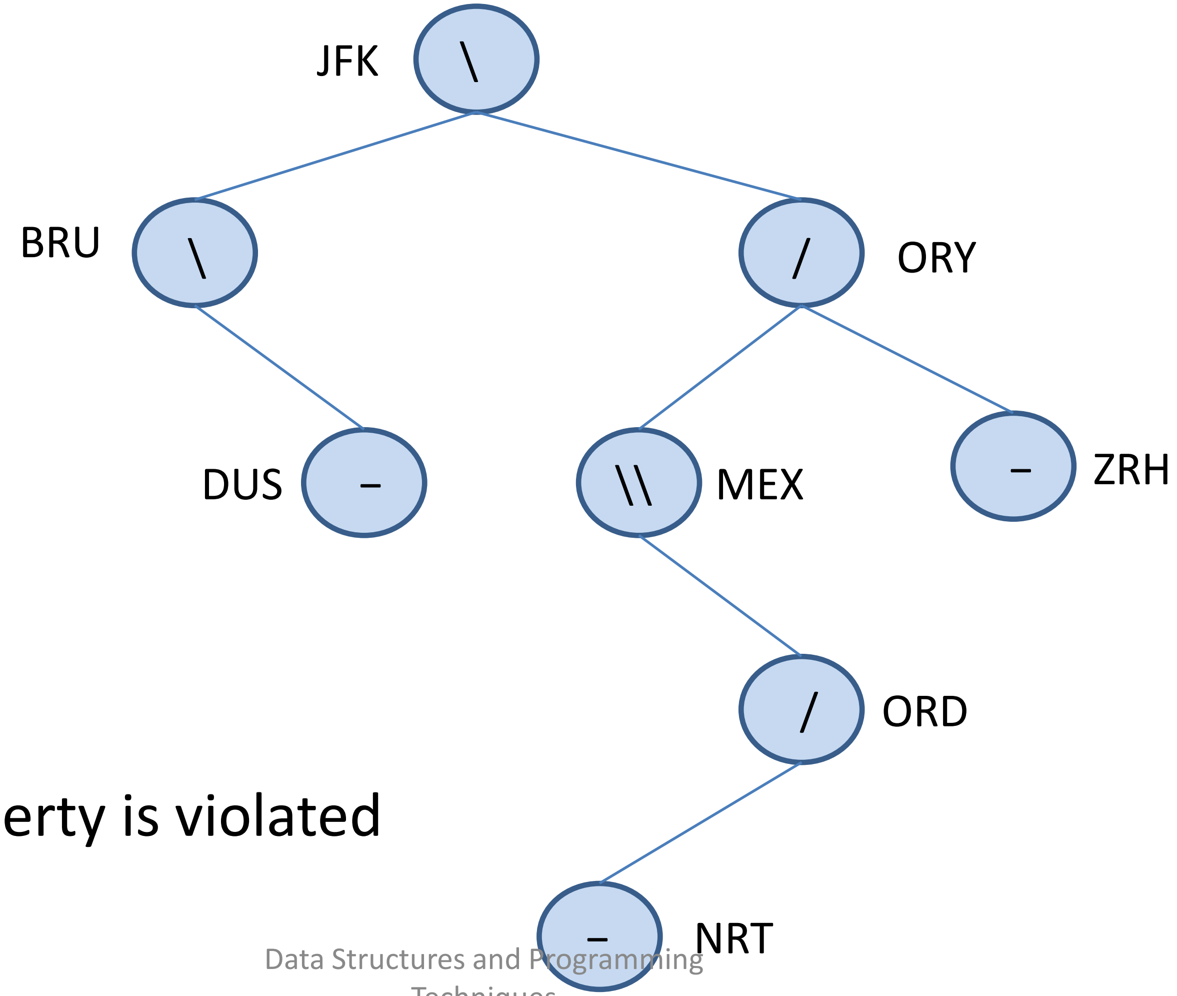
Do a Single Right Rotation at ORY



Insert DUS, ZRH, MEX and ORD

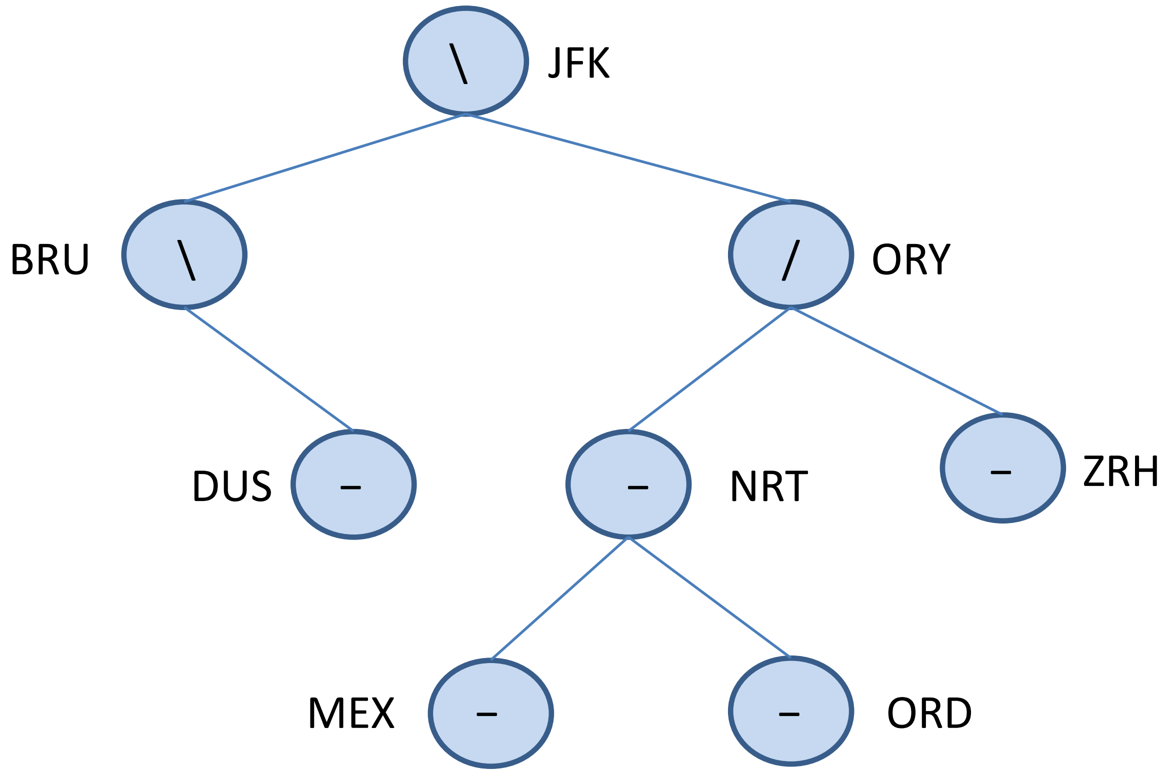


Insert NRT

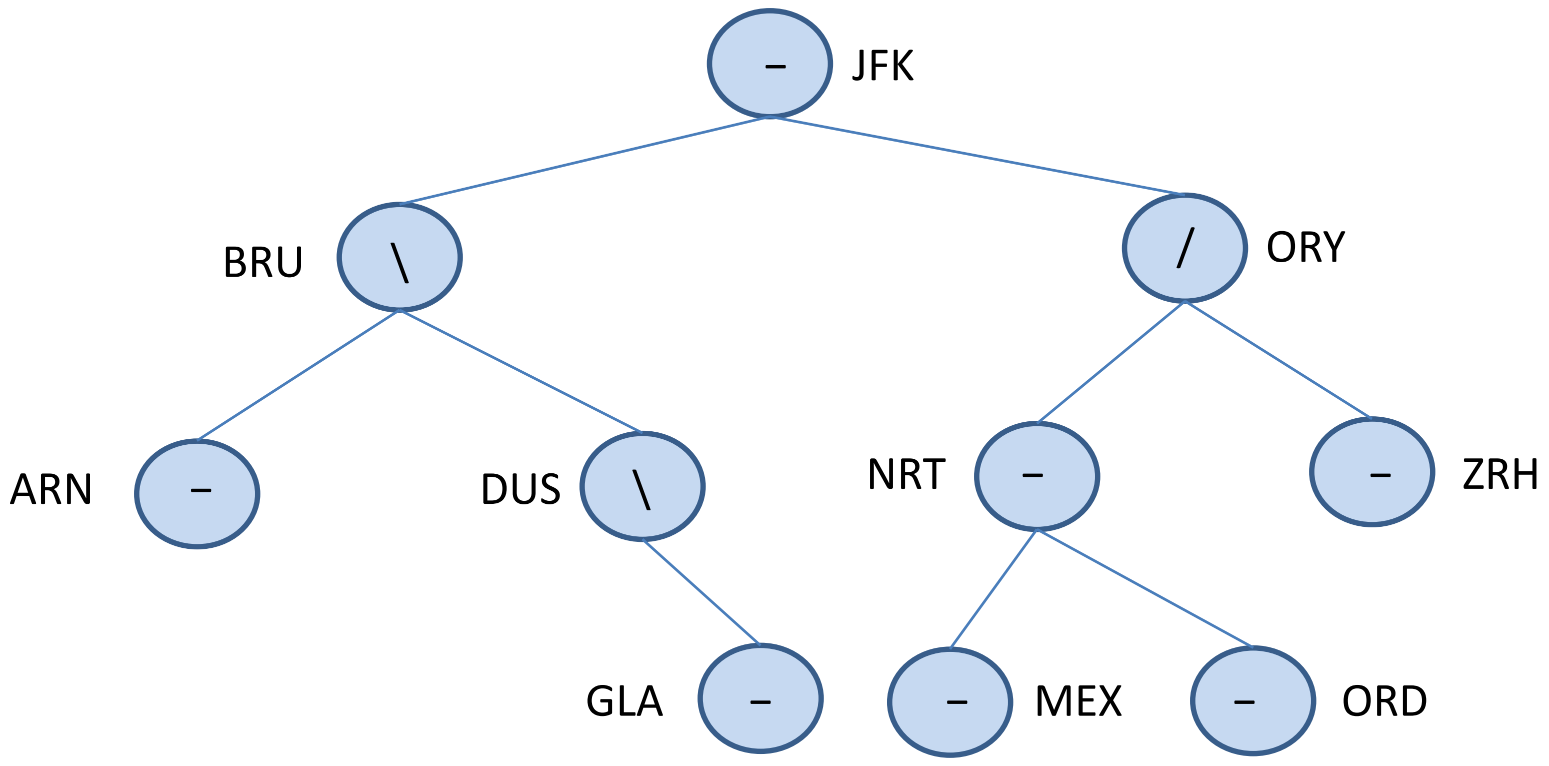


The AVL property is violated at node MEX.

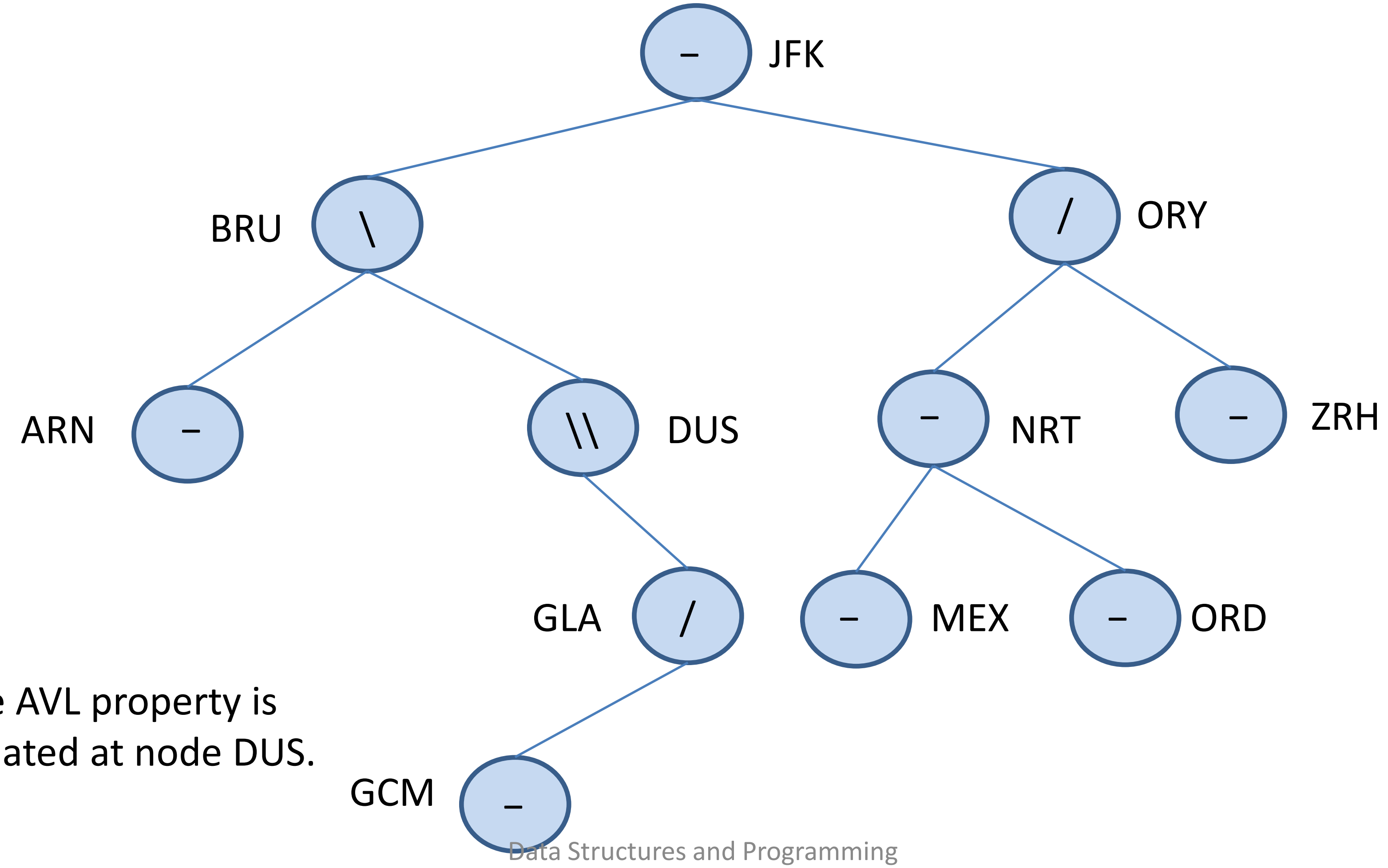
Double Right-Left Rotation at ORD and MEX



Insert ARN and GLA

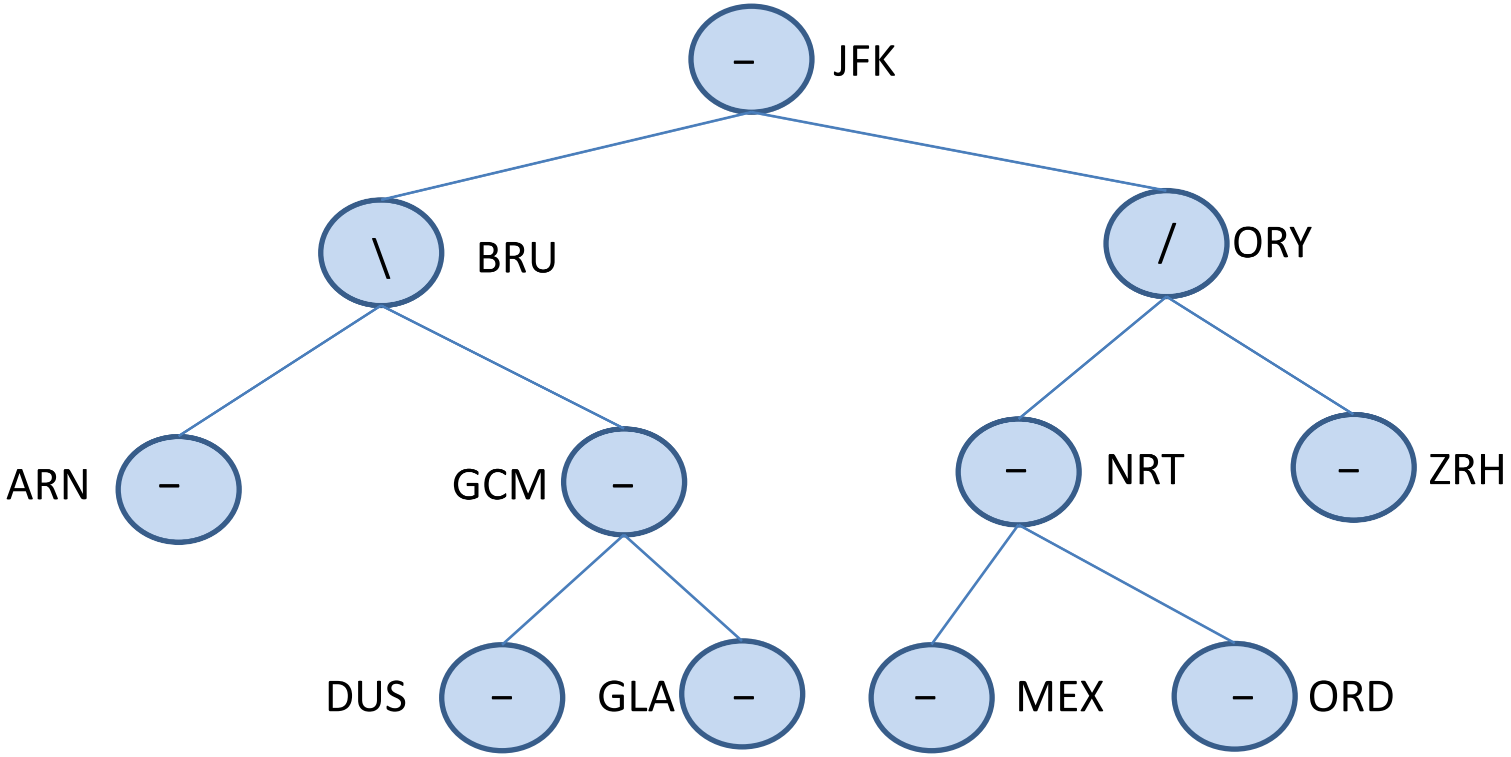


Insert GCM



The AVL property is violated at node DUS.

Double Right-Left Rotation at GLA and DUS



Deletion of a Node

- To delete a node from an AVL tree, we will use similar ideas with the ones we used for insertion.
- We will reduce the problem to the case when the node *x* to be deleted has **at most one child.**
- Suppose *x* has two children. Find the **immediate predecessor** *y* of *x* **under inorder traversal** by first taking the left child of *x*, and then moving right as far as possible to obtain *y*.
- The node *y* is guaranteed to have no right child because of the way it was found.
- Place *y* into the position in the tree occupied by *x*.
- Delete *y* from its former position by proceeding as follows.

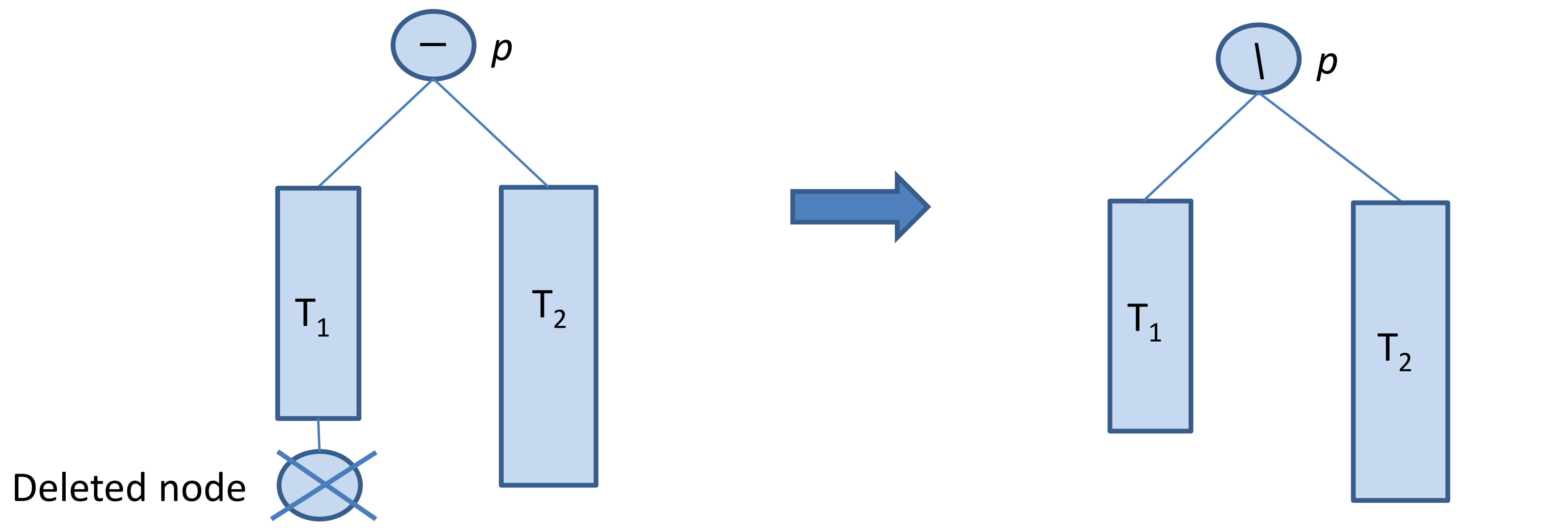
Deletion of a Node (cont'd)

- Delete node *y* from the tree. Since we know that *y* has at most one child, we delete *y* by simply linking the parent of *y* to the single child of *y* (or to `NULL`, if there is no child).
- The height of the subtree formerly rooted at *y* has been reduced by 1, and we must now trace the effects of this change on height through all the nodes on the path from *y* back to the root of the tree.
- We will use a Boolean variable `shorter` to show if the height of a subtree has been shortened. The action to be taken at each node depends on the value of `shorter`, on the balance factor of the node and sometimes on the balance factor of a child of the node.
- The Boolean variable `shorter` is initially `TRUE`. The following steps are to be done **for each node *p* on the path from the parent of *y* to the root of the tree**, provided `shorter` remains `TRUE`. When `shorter` becomes `FALSE`, then no further changes are needed, and the algorithm terminates.

Case 1: No rotation

- The current node *p* has balance factor equal. The balance factor of *p* is changed accordingly as its left or right subtree has been shortened, and `shorter` becomes `FALSE`.

Case 1 Graphically

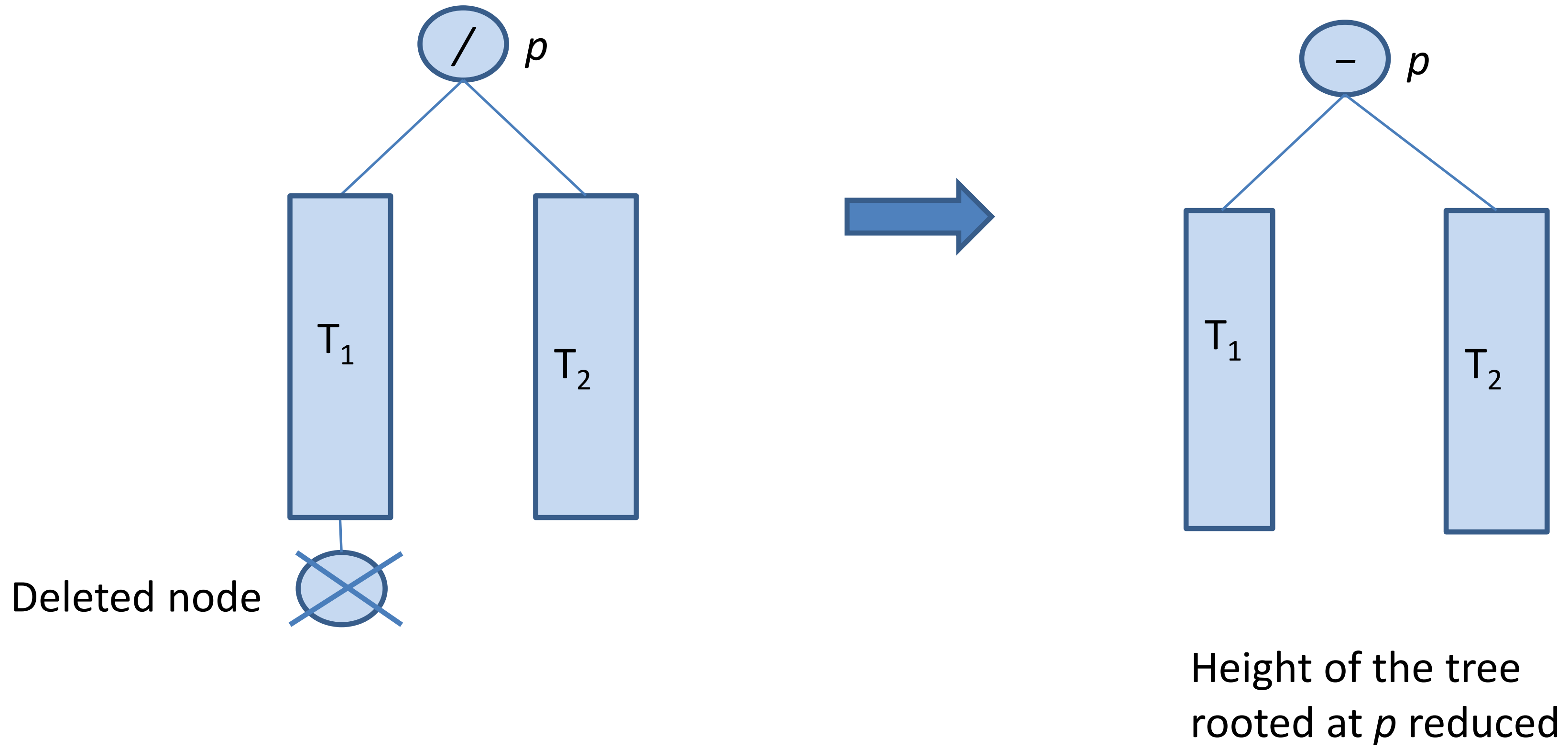


Height of tree rooted at *p* unchanged

Case 2: No rotation

- The balance factor of *p* is not equal, and the taller subtree was shortened. Change the balance factor of *p* to equal and leave `shorter` as `TRUE` because the height of tree rooted at *p* has changed.

Case 2 Graphically



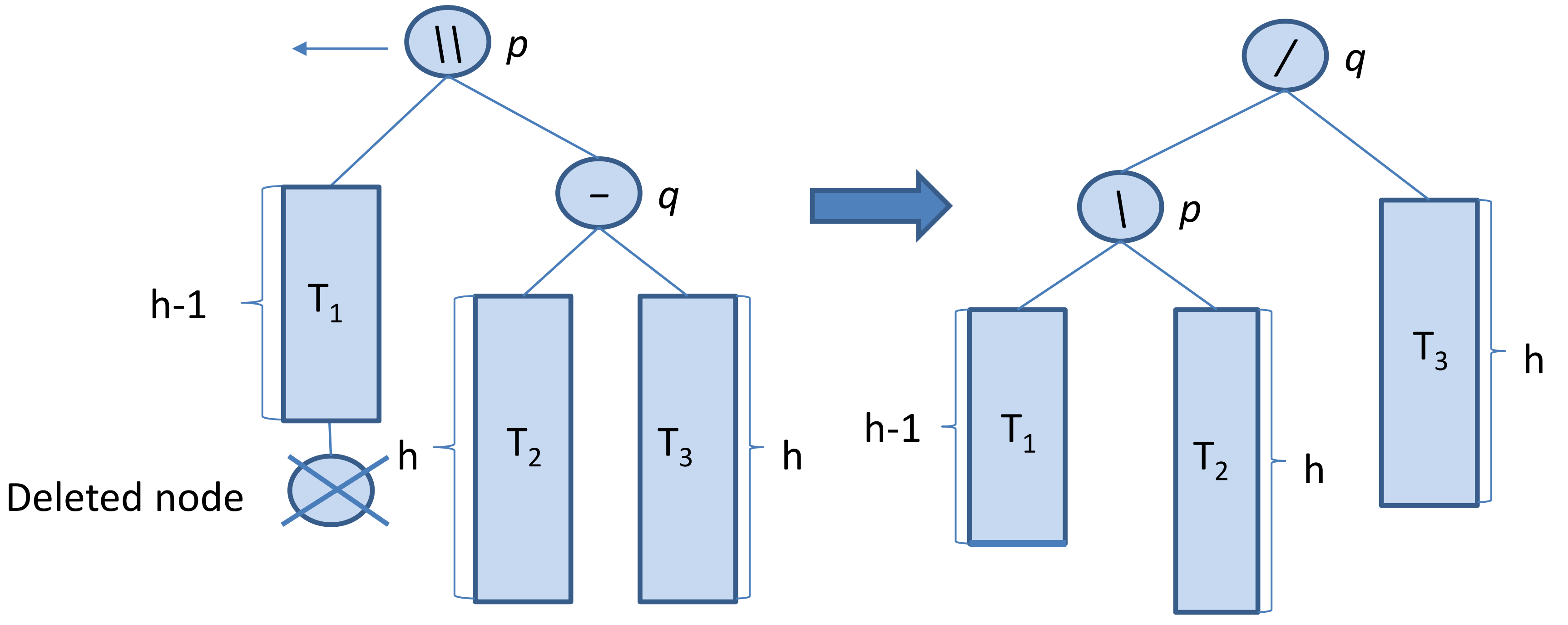
Case 3

- The balance factor of *p* is not equal (i.e., “-”) and the shorter subtree was shortened. The height requirement for an AVL tree is now violated at *p*, so we apply a rotation as follows to restore balance.
- Let *q* be the root of the taller subtree of *p* (the one not shortened). We have **three cases according to the balance factor of *q***.

Case 3a: Single left rotation

- **The balance factor of *q* is equal (i.e., “-”)**. A single left rotation at *p* (with changes to the balance factors of *p* and *q*) restores balance, and `shorter` becomes `FALSE`.

Case 3a Graphically

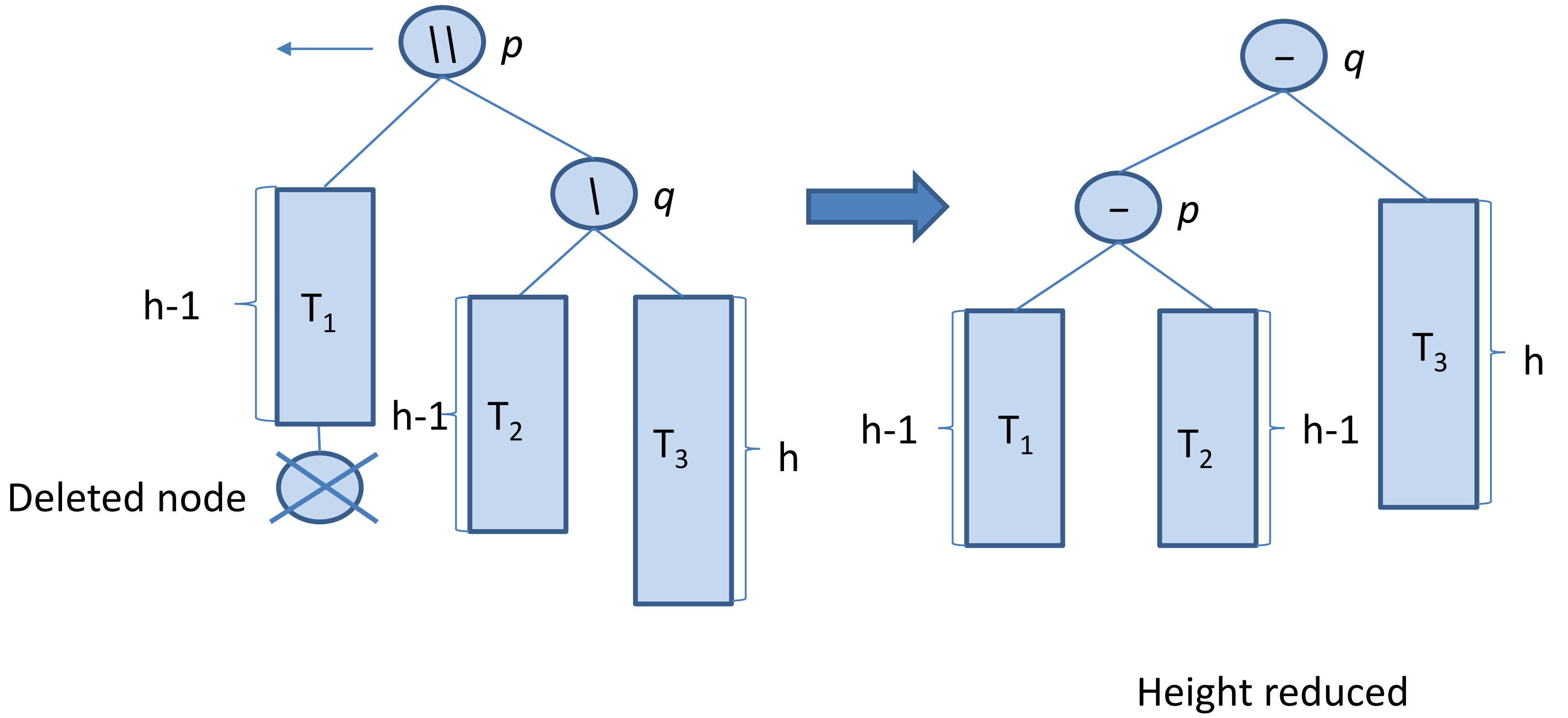


Height unchanged

Case 3b: Single left rotation

- **The balance factor of *q* is the same as that of *p***. Apply a single left rotation at *p*, set the balance factors of *p* and *q* to equal, and leave `shorter` as `TRUE`.

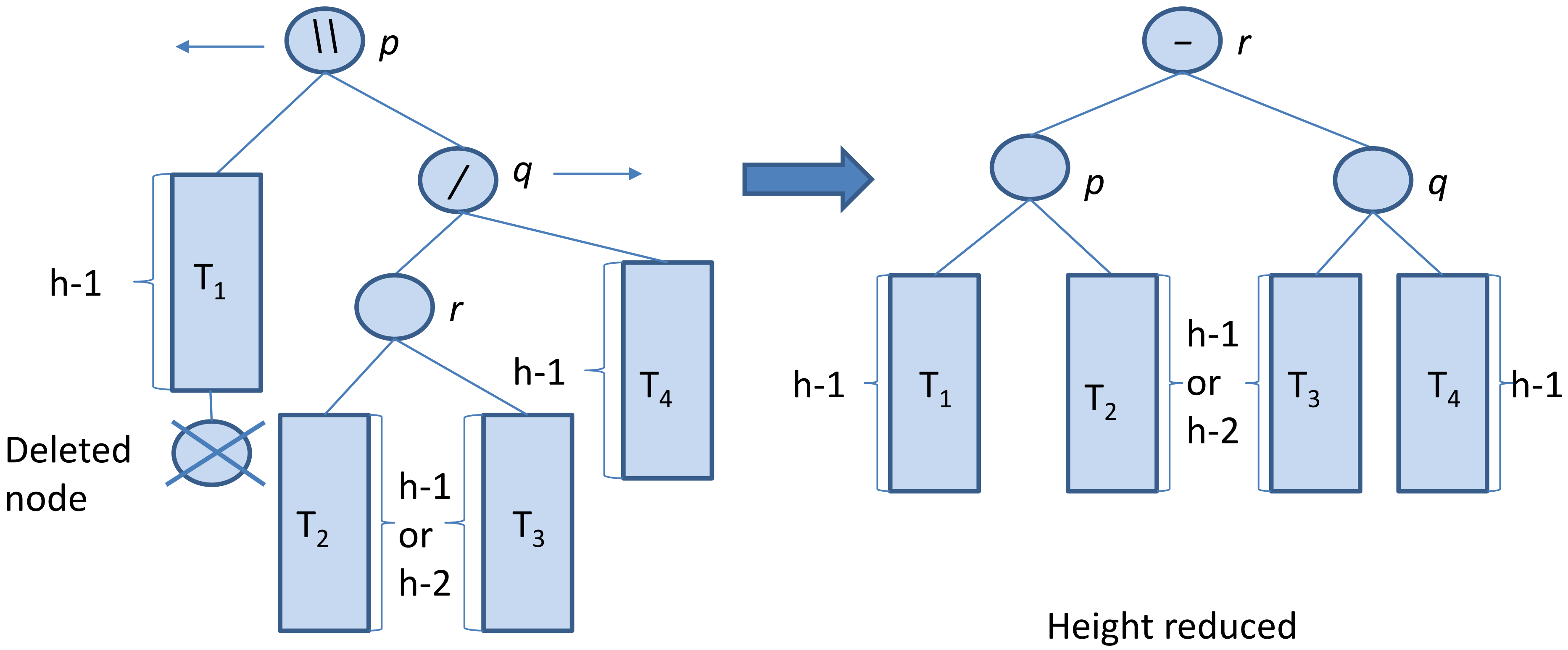
Case 3b Graphically



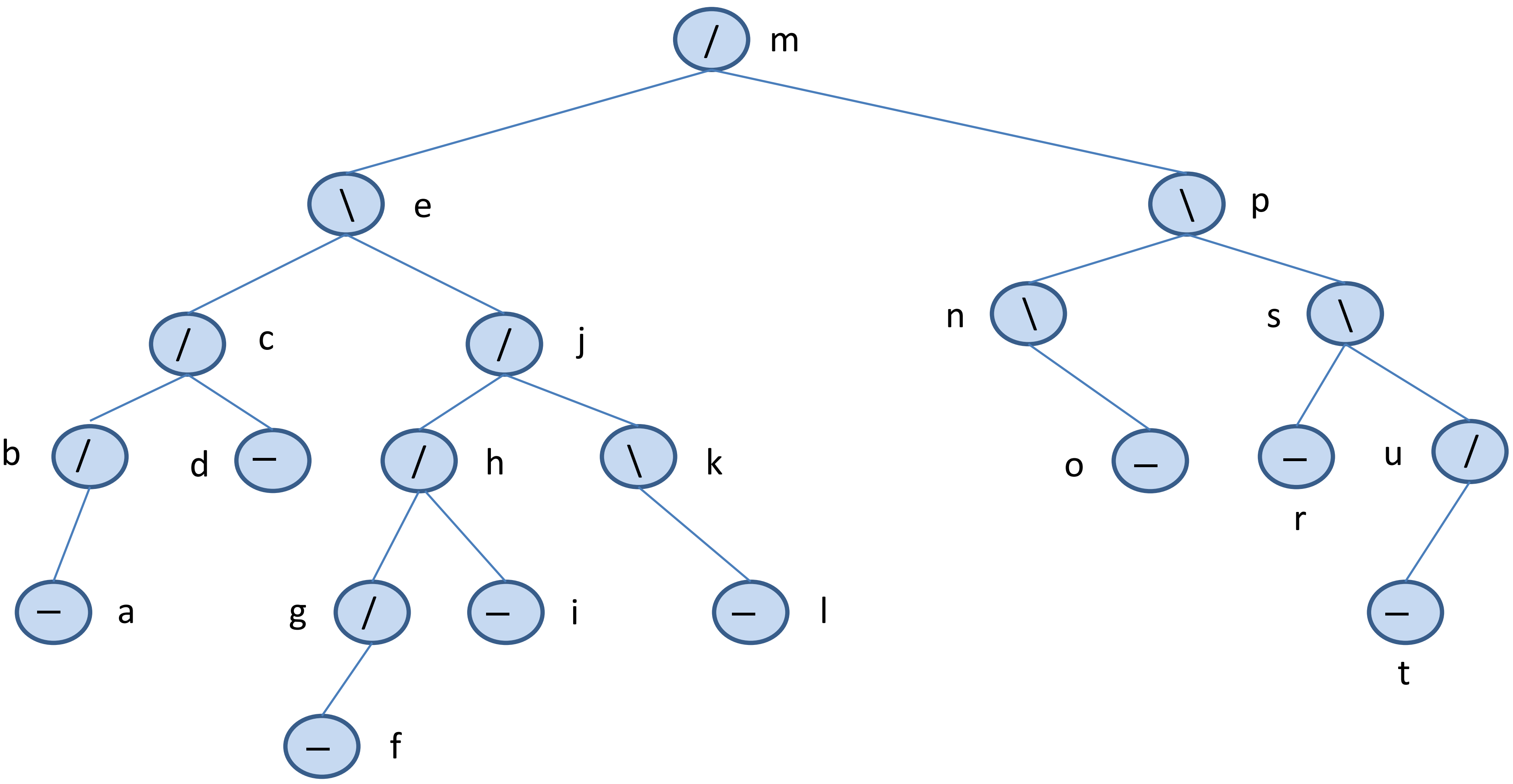
Case 3c: Double right-left rotation

- **The balance factors of *p* and *q* are opposite.** Apply a double right-left rotation (first at *q*, then at *p*), set the balance factor of the new root to equal and the other balance factors as appropriate, and leave `shorter` as `TRUE`.

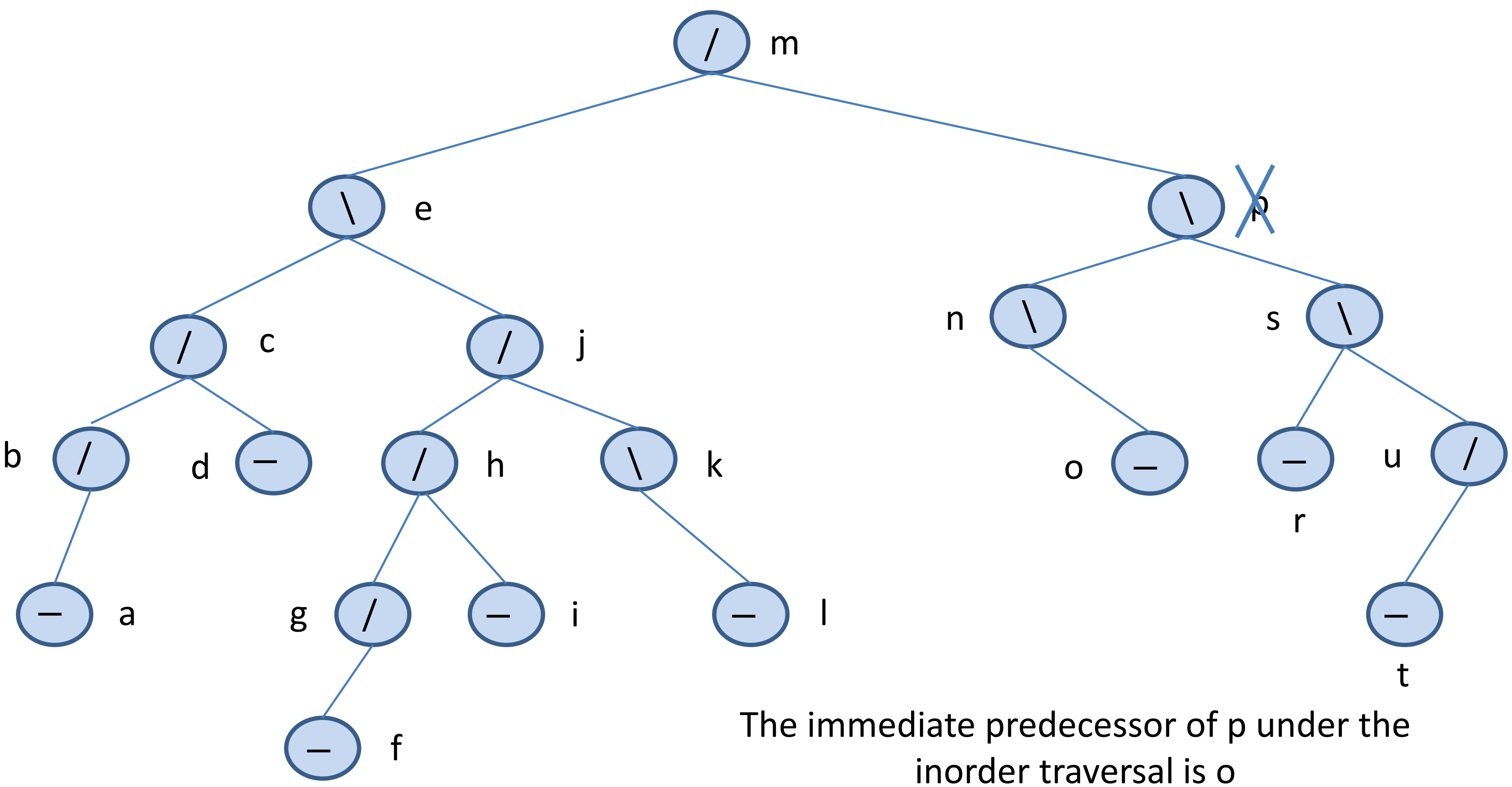
Case 3c Graphically



Example of Deletion in an AVL Tree

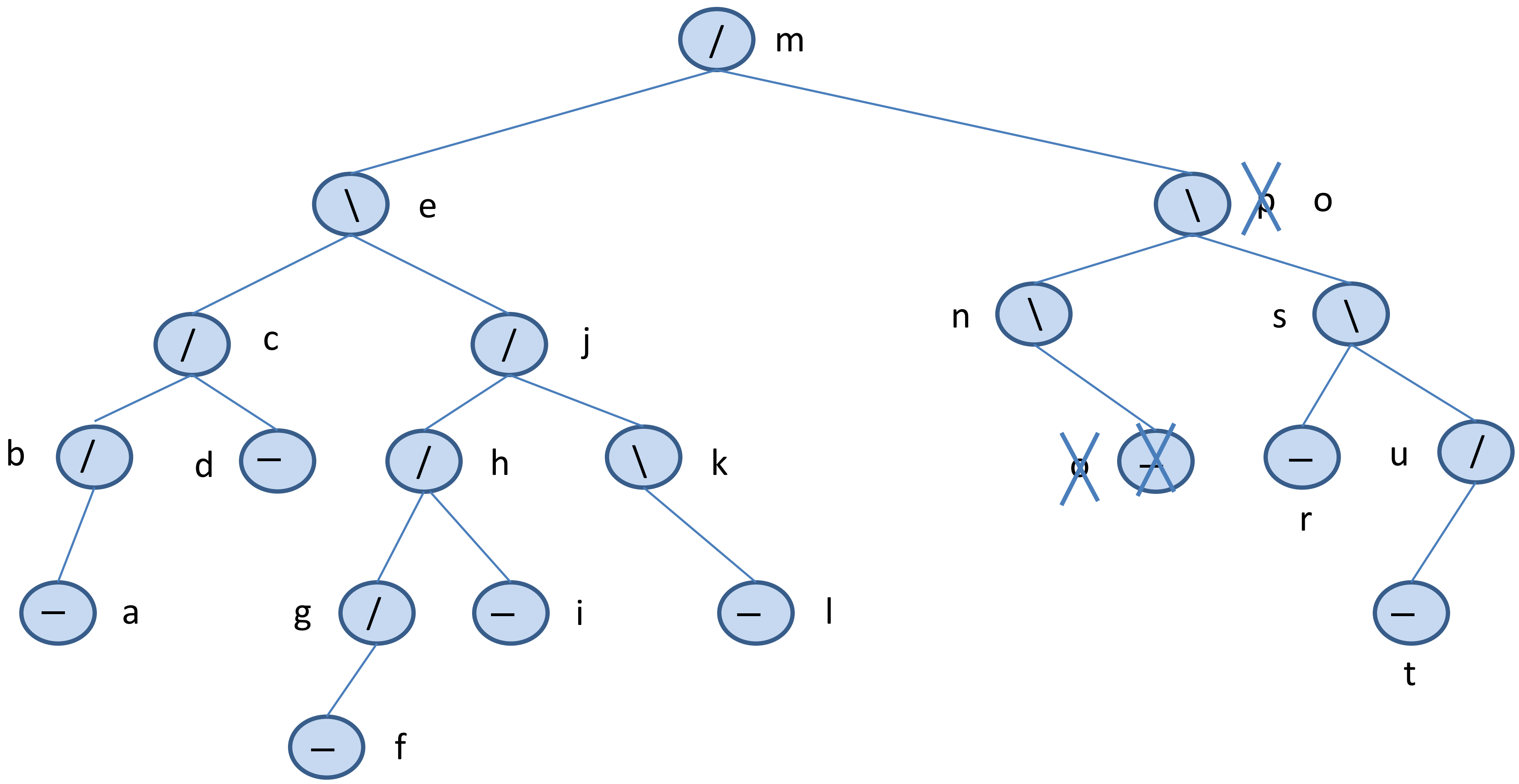


Delete p

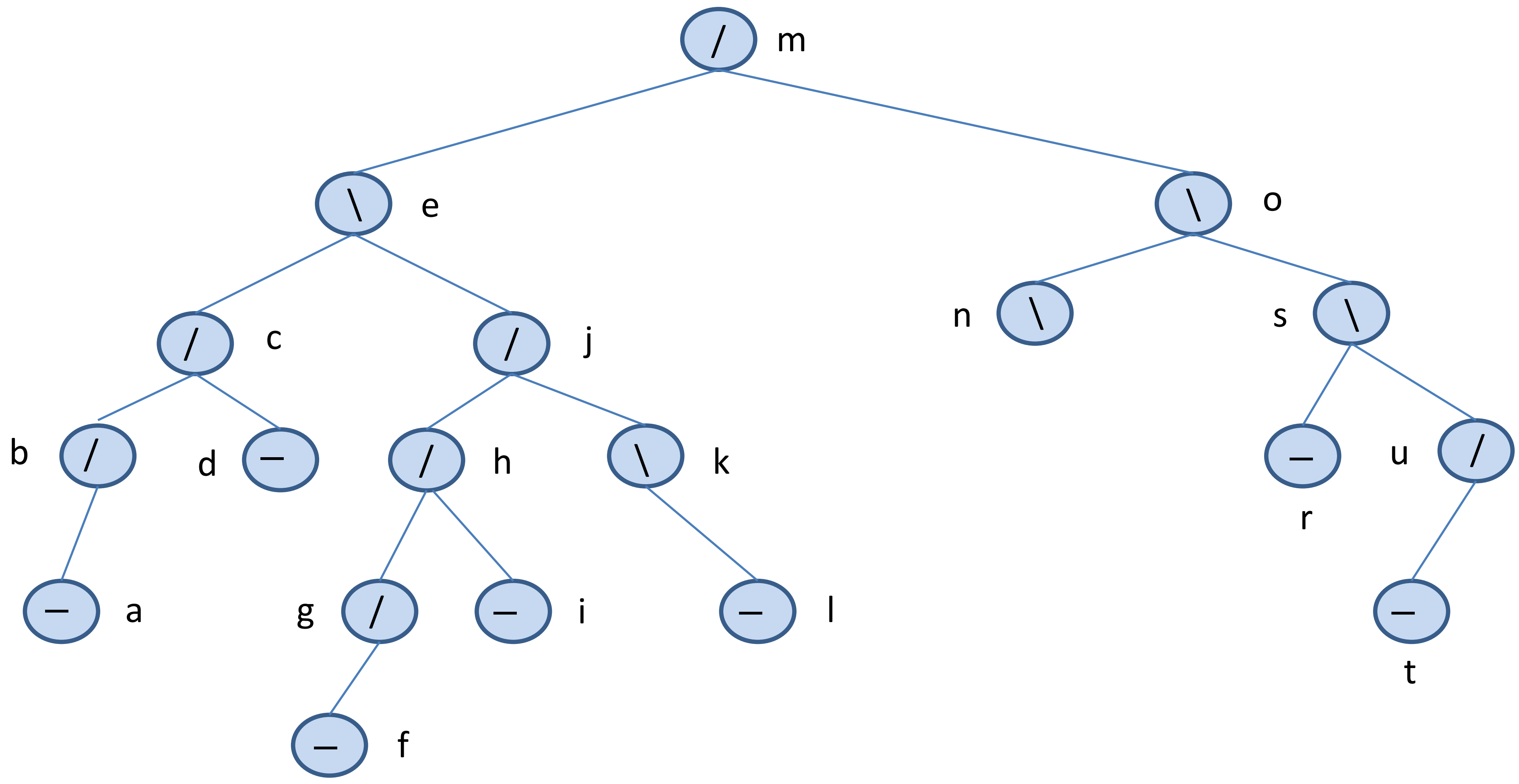


The immediate predecessor of p under the inorder traversal is o

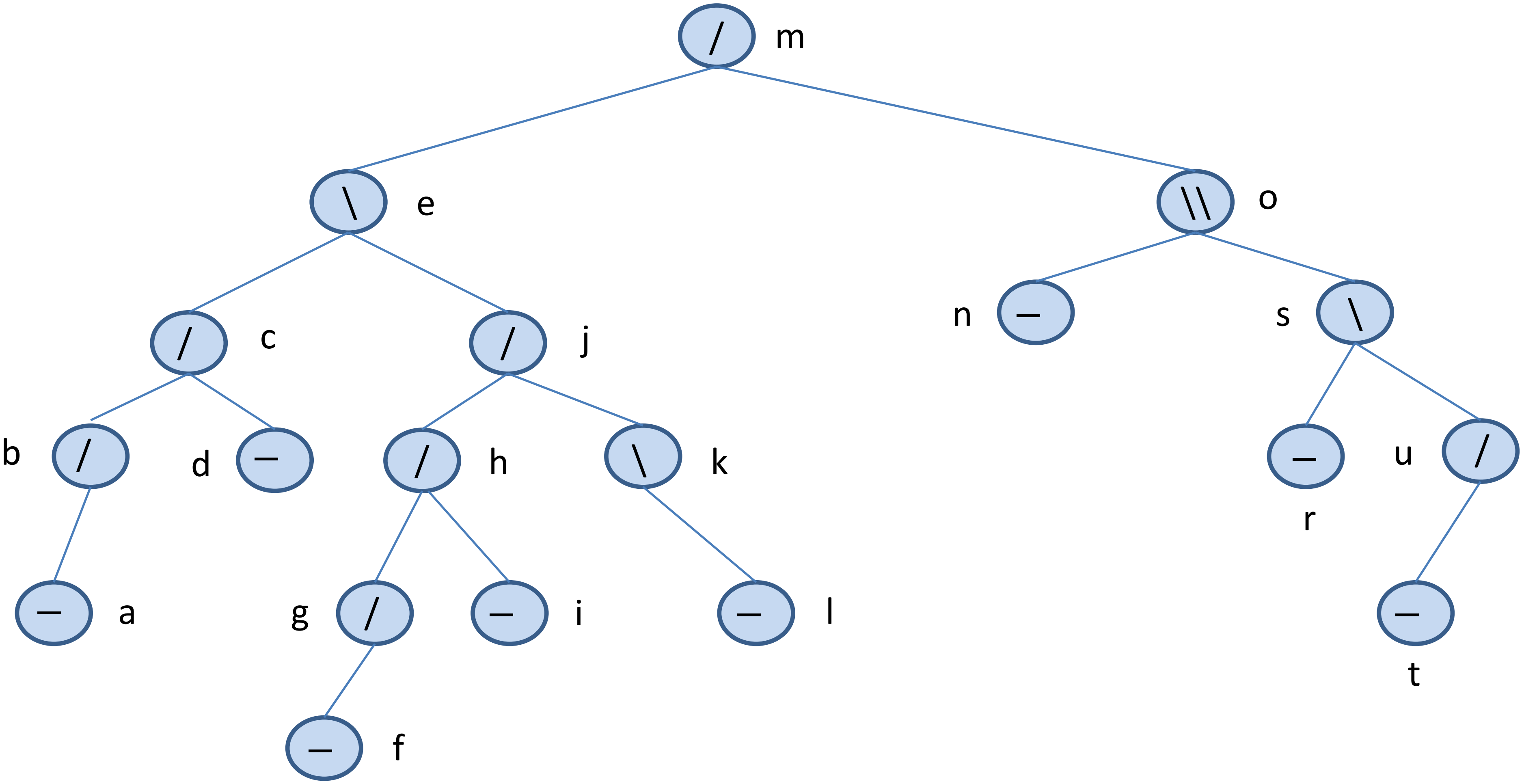
Replace p with o and Delete o



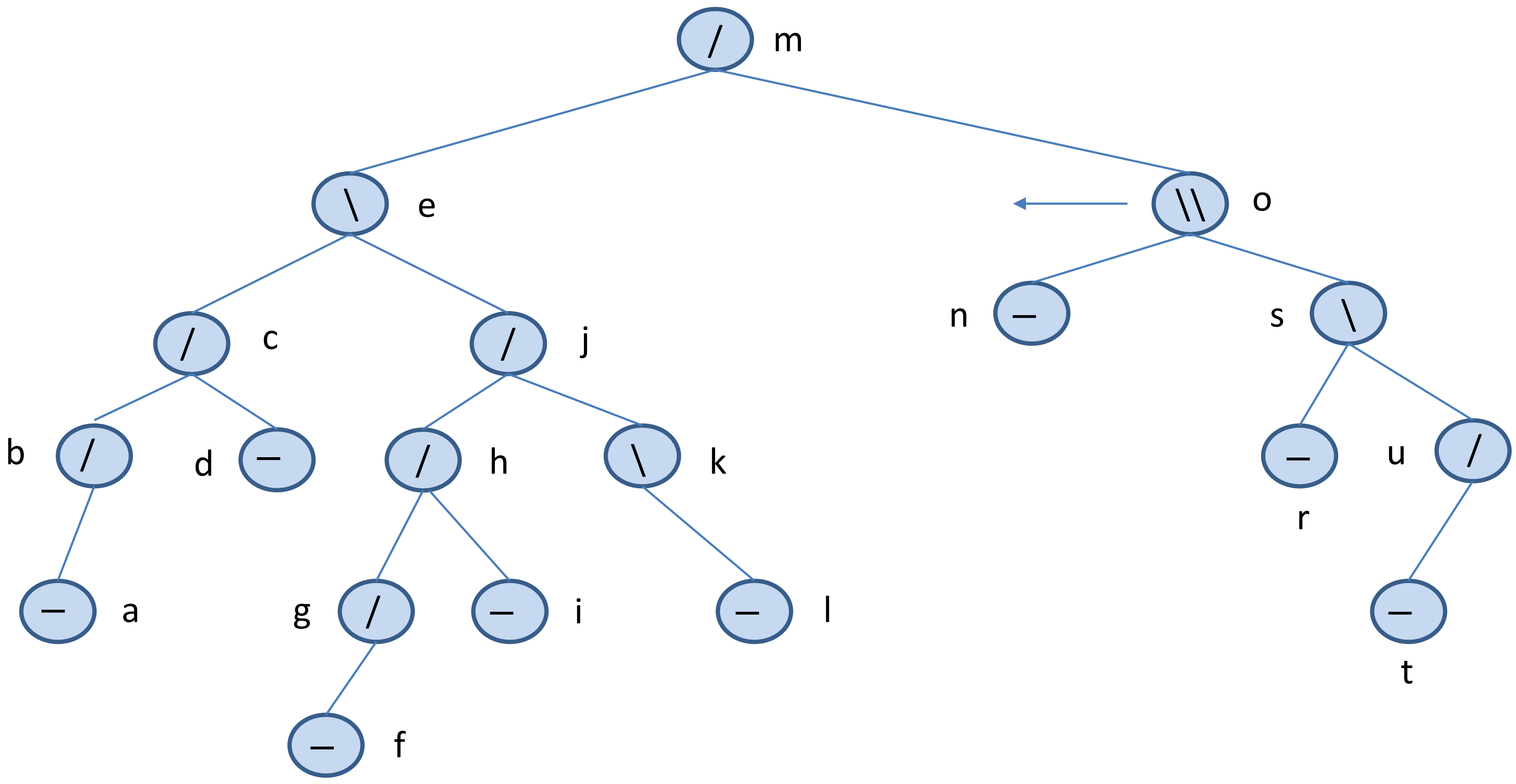
Adjust Balance Factors



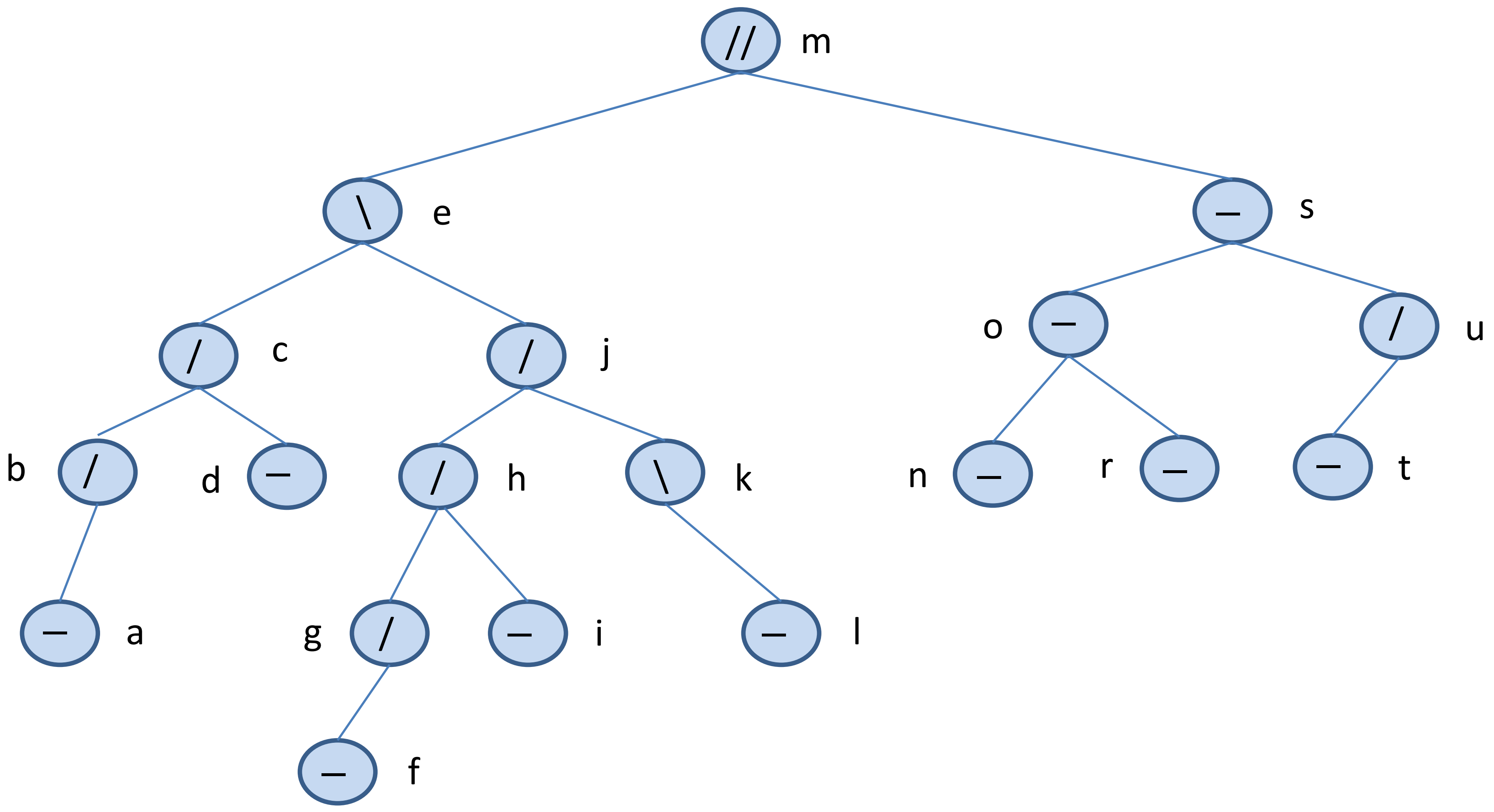
Balance Factors Adjusted



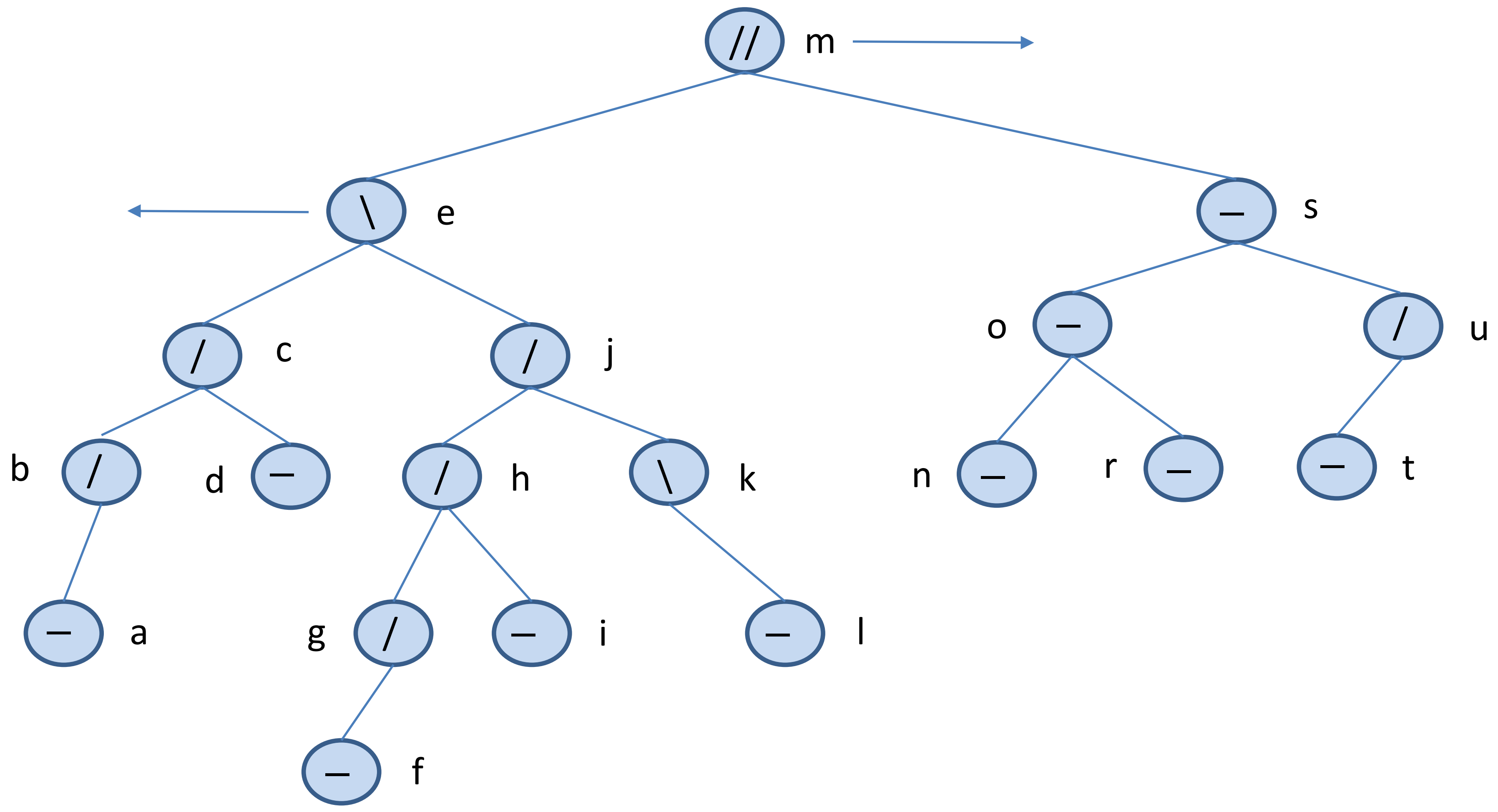
Rotate Left at o



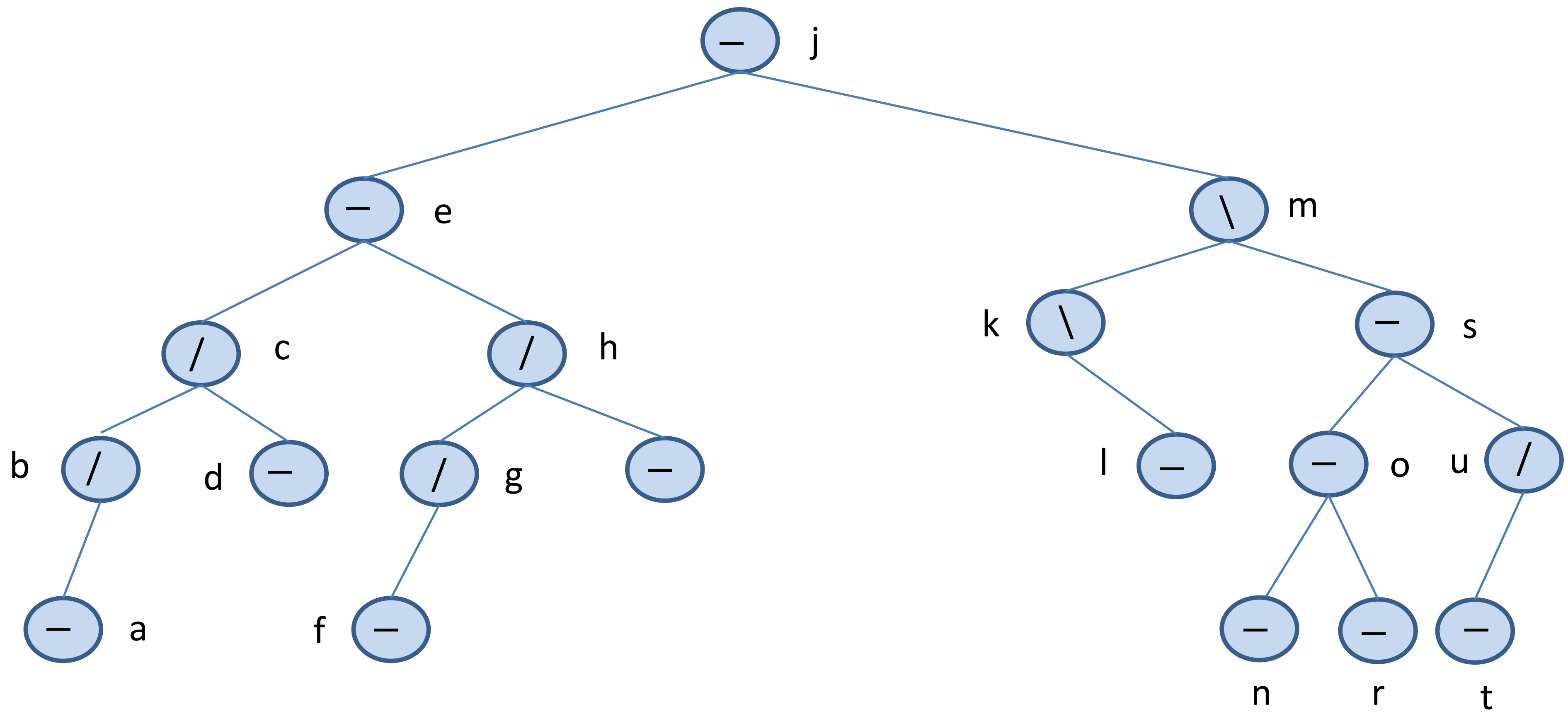
Result of Left Rotation



Double Rotate Left-Right at e and m



Result of Double Rotation



Search in an AVL Tree

- Since an AVL tree is a binary search tree, the search algorithm is as in the case of binary search trees.

Complexity of Operations on AVL Trees

- The operations of search, insertion and deletion in an AVL tree visit the nodes **along a root-to-leaf path** of the tree, plus, possibly, their siblings.
- There is a **going-down phase** which typically involves search, and a **going-up phase** which involves rotations.
- The complexity of the work done at each node is $O(1)$.
- Thus, the worst-case complexity for **search**, **insertion** and **deletion** in an AVL tree with height h and n nodes is $\boldsymbol{O(h) = O(\log n)}$.

Readings

- T. A. Standish. *Data Structures, Algorithms and Software Principles in C*.
  – Chapter 9. Section 9.8.
- R. Kruse, C.L. Tondo and B.Leung. *Data Structures and Program Design in C.*
  – Chapter 9. Section 9.4.
- M. T. Goodrich, R. Tamassia and Michael H. Goldwasser. *Data Structures and Algorithms in Java. 6th* edition. John Wiley and Sons, 2014.
  – Section 11.3