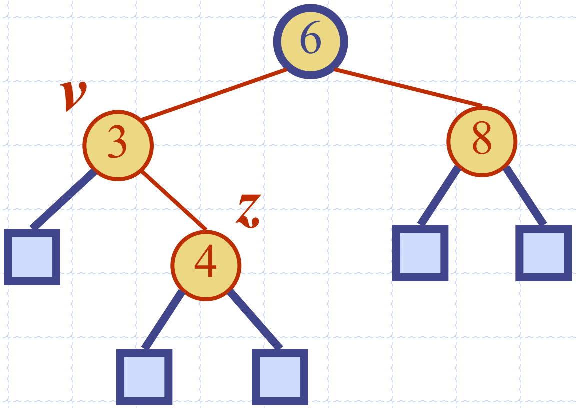
Presentation for use with the textbook Algorithm Design and Applications, by M. T. Goodrich and R. Tamassia, Wiley, 2015

Weak AVL Trees

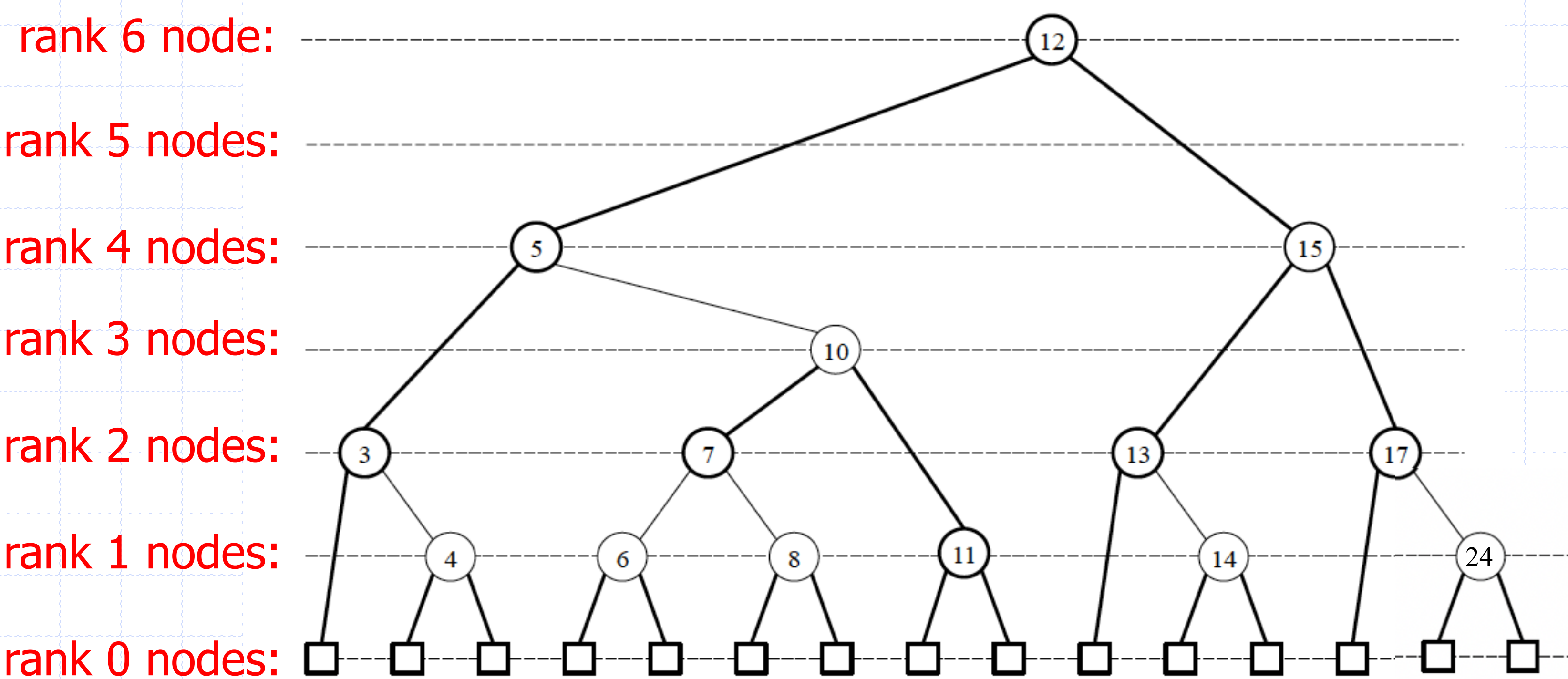


WAVL Tree Definition

- For a tree with ranks on its nodes, for each node v in T other than the root, we define the **rank difference** of v as the difference between the rank of v and the rank of v's parent.
  - An internal node is a 1,1-node if its children each have rank difference 1.
  - An internal node is a 2,2-node if its children each have rank difference 2.
  - An internal node is a 1,2-node if it has one child with rank difference 1 and one child with rank difference 2.
- A tree is a **weak AVL (wavl) tree** if the ranks assigned to its nodes satisfy the following properties:
  - Rank-difference Property: the rank difference of any non-root node is 1 or 2.
  - External-node Property: every external node (leaf) has rank 0.
  - Internal-node Property: An internal node with two external-node children cannot be a 2,2-node.

Example WAVL Tree

- A tree is a **weak AVL (wavl) tree** if the ranks satisfy the following:
  - Rank-difference Property: the rank difference of any non-root node is 1 or 2.
  - External-node Property: every external node (leaf) has rank 0.
  - Internal-node Property: An internal node with two external-node children cannot be a 2,2-node.



Height of a WAVL Tree

Theorem: The height of a wavl tree storing n keys is O(log n).

Proof: Let n_r denote the minimum number of internal nodes in a wavl tree whose root has rank r. Then, by the rules for ranks in a wavl tree,

$$\begin{aligned} n_0 &= 0 \\ n_1 &= 1 \\ n_2 &= 2 \\ n_r &= 1 + 2n_{r-2}, \text{ for } r \geq 3. \end{aligned}$$

This implies that $n_r \geq 2^{r/2} - 1$, that is, $r \leq 2\log(n_r + 1)$. Thus, by the definition of n_r, $r \leq 2\log(n+1)$. That is, the rank of the root is at most $2\log(n+1)$, which implies that the height of the tree is bounded by $2\log(n+1)$, since the height of a wavl tree is never more than the rank of its root. ■

- Thus wavl trees are balanced binary search trees.

Relationship to AVL Trees

Theorem: Every AVL Tree is a weak AVL Tree.

Proof: Suppose we are given an AVL tree, T, with a rank assignment, r(v), for the nodes of T, so that r(v) is equal to the height of v in T. Then:

- Every external node in T has rank 0.
- By the height-balance property for AVL trees, every internal node is either a 1,1-node or 1,2-node.

Hence, the rank assignment, r(v), for an AVL tree implies T is a weak AVL tree.

- Thus, an AVL tree is a weak AVL tree with no 2,2-nodes, which motivates the name "weak AVL tree."

WAVL Trees are Red-Black Trees

Theorem: Every wavl tree can be colored as a red-black tree.

Proof: Suppose we are given a wavl tree, T, with a rank assignment, $r(v)$. For each node v in T, assign a new rank, $r'(v)$, to each node v as follows:

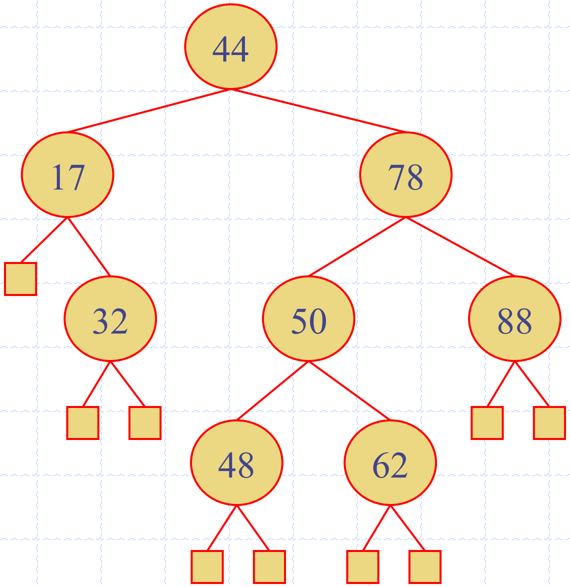
$$r'(v) = \lfloor r(v)/2 \rfloor.$$

Then each external node still has rank 0 and the rank difference for any node is 0 or 1. In addition, note that the rank difference, in the $r(v)$ rank assignment, between a node and its grandparent must be at least 2; hence, in the $r'(v)$ rank assignment, the parent of any node with rank difference 0 must have rank difference 1. Thus, the $r'(v)$ rank assignment is red-black-equivalent; hence, by Theorem 4.5, T can be colored as a red-black tree. ■

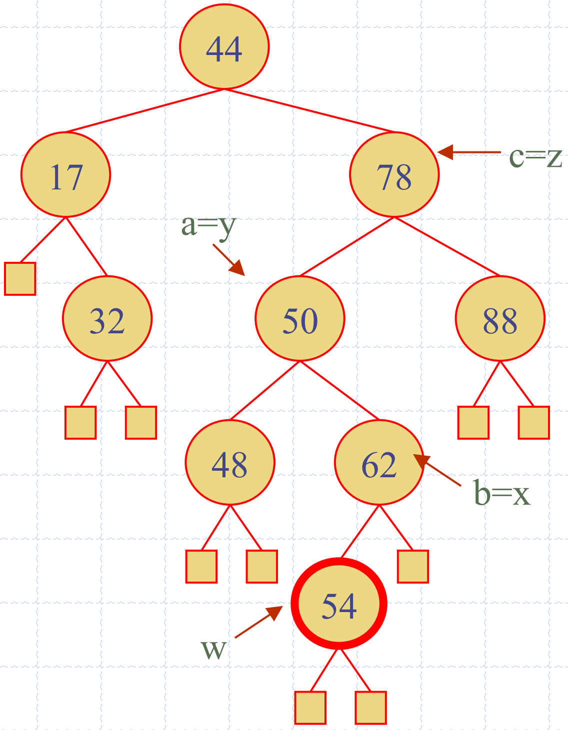
Nevertheless, the relationship does not go the other way, as there are some red-black trees that cannot be given rank assignments to make them be wavl trees.

Insertion

- Insertion is as in a binary search tree
- Always done by expanding an external node.
- Example:



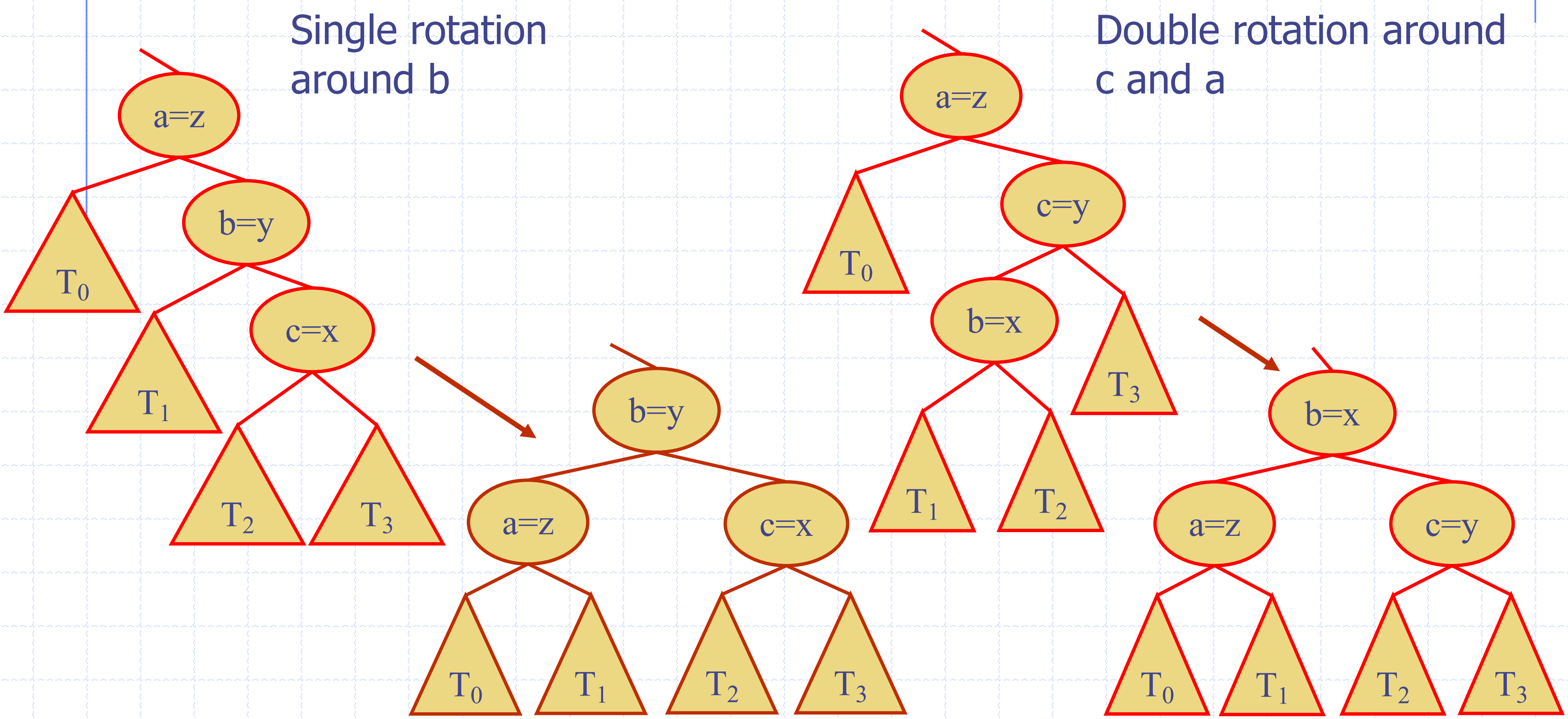
before insertion



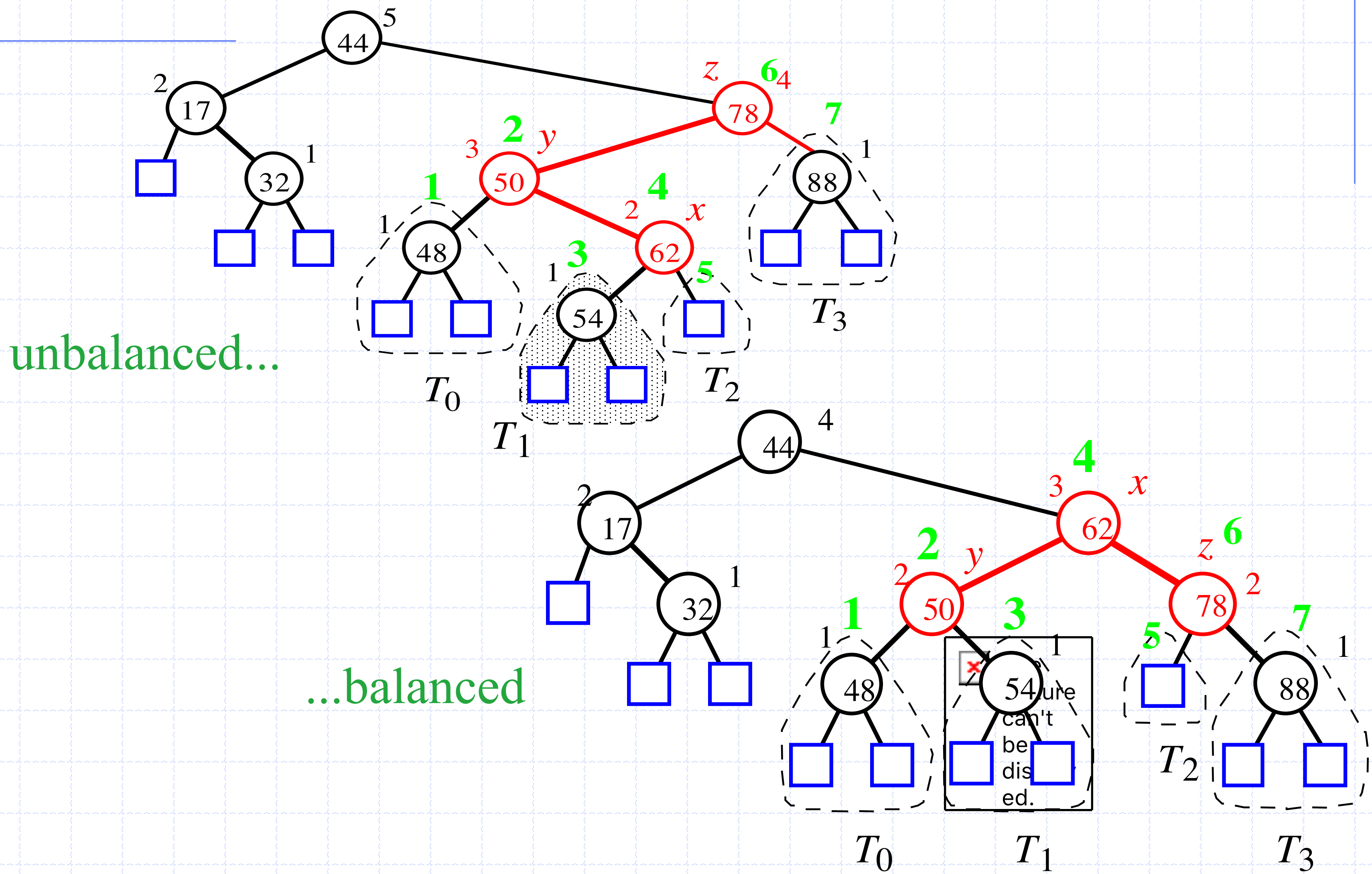
after insertion

Trinode Restructuring

- Let (a,b,c) be the inorder listing of x, y, z
- Perform the rotations needed to make b the topmost node of the three

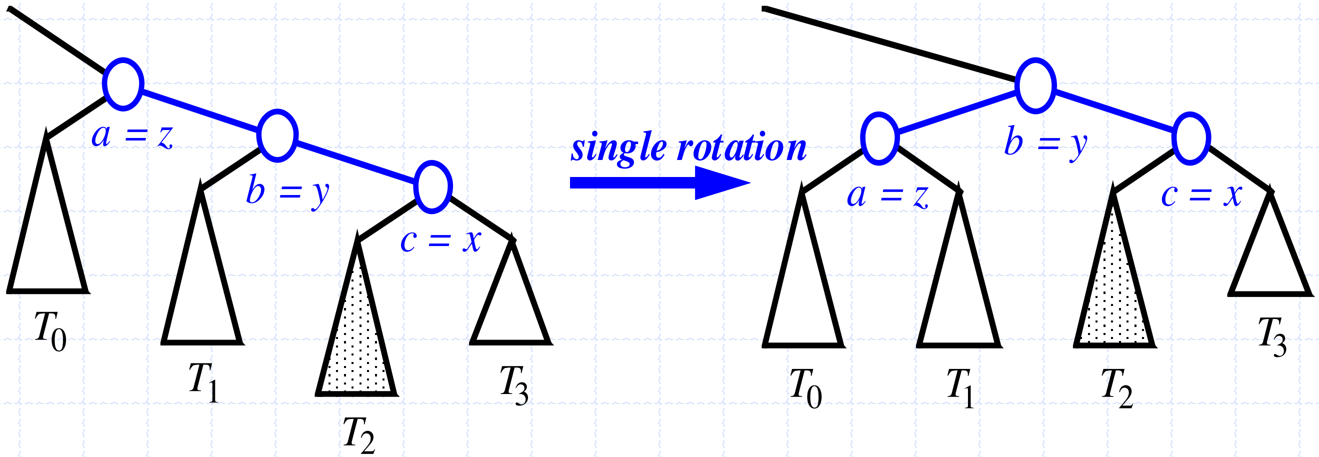


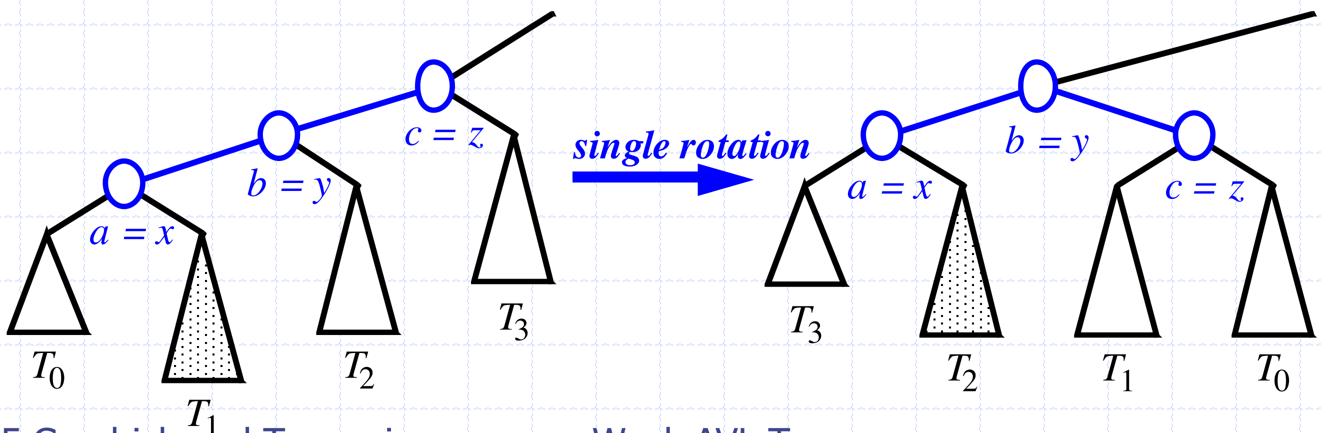
Insertion Example, continued



Restructuring (as Single Rotations)

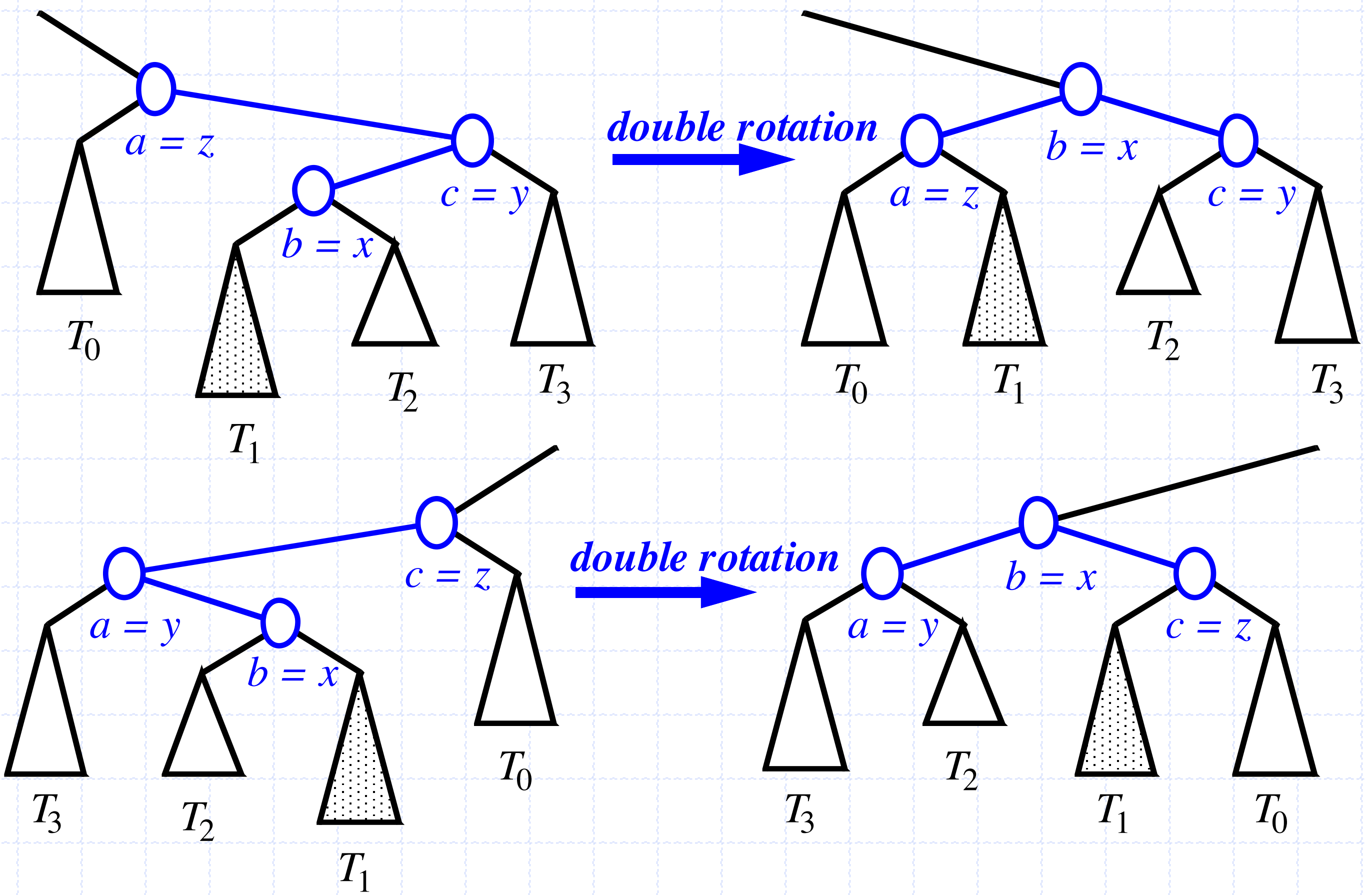
- Single Rotations:





Restructuring (as Double Rotations)

- double rotations:



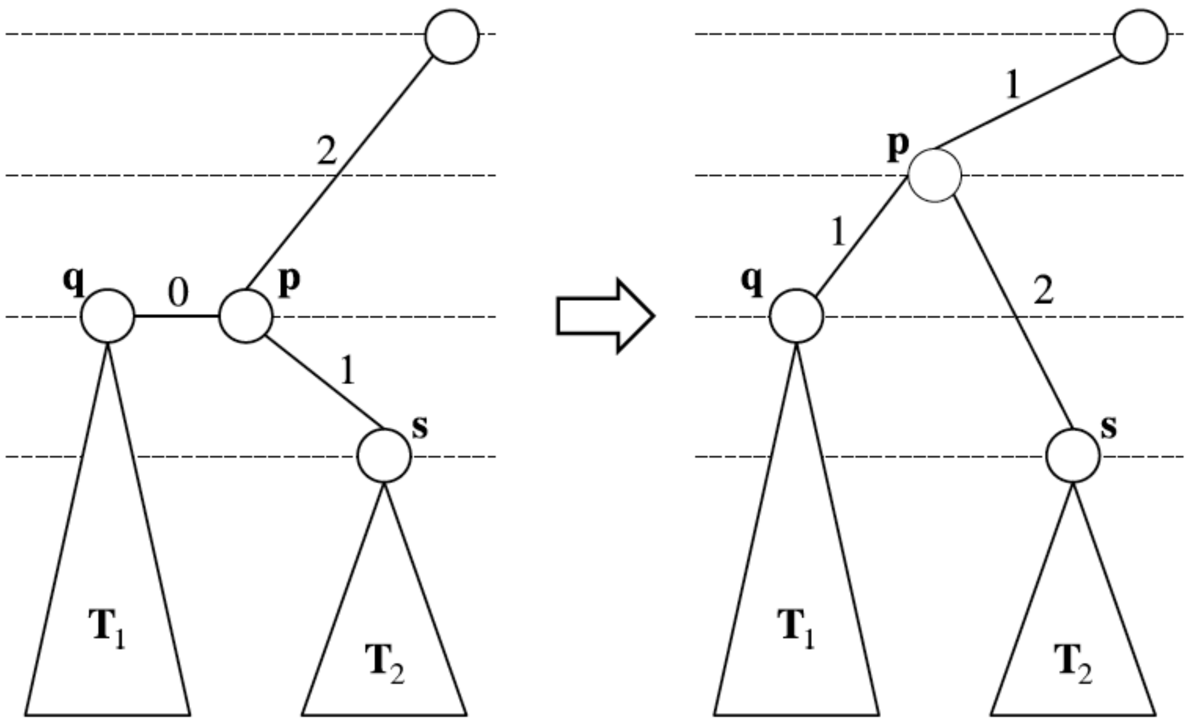
Rebalancing after Insertion

Let q be the node where we just performed an insertion, and note that q previously was an external node. Now q has two external-node children; hence, we increase the rank of q by 1, which in an action called a **promotion** at q.

- If q has rank difference 1 after promotion, or if q is the root, then we are done.
- Otherwise, if q now has rank-difference 0, with its parent, p, then we have two cases.

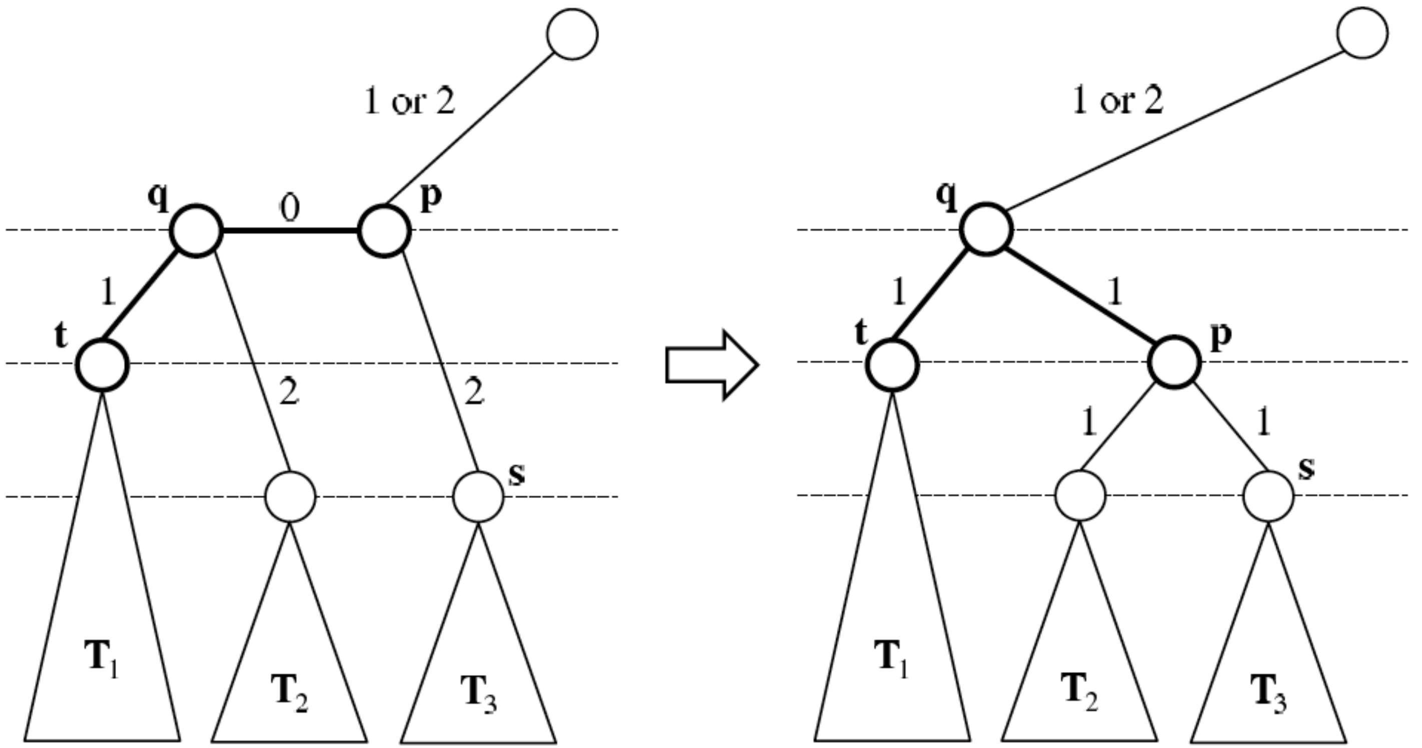
Rebalancing Operation, Case 1a

- Case 1: q's sibling has rank-difference 1.
- In this case, we promote q's parent, p. This fixes the rank-difference property for q, and if there is no violation for p, then we are done.



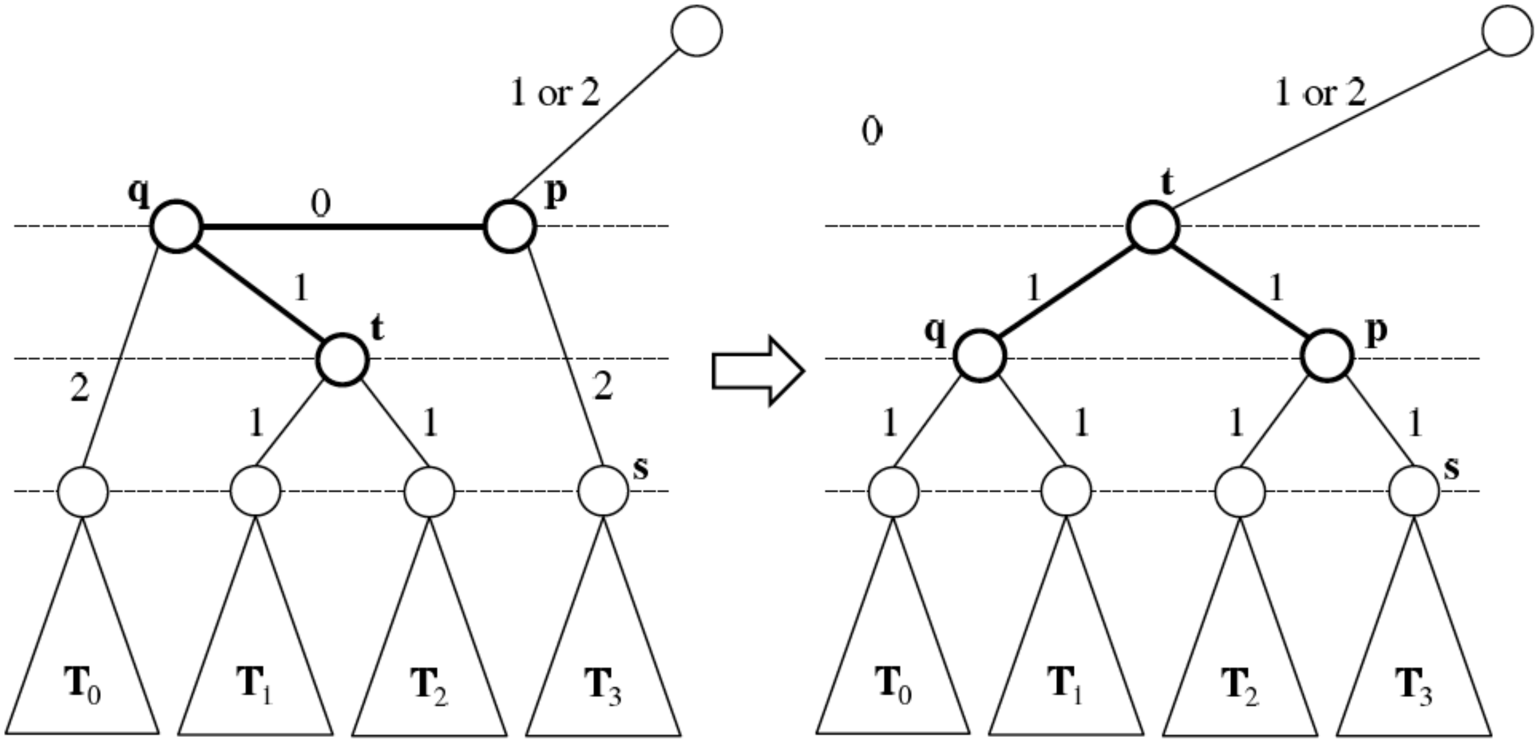
Rebalancing Operation, Case 2a

- Case 2: q's sibling has rank-difference 2.
- Let t denote a child of q that has rank-difference 1. We perform a trinode restructuring operation at t. Single-rotation rank updates:



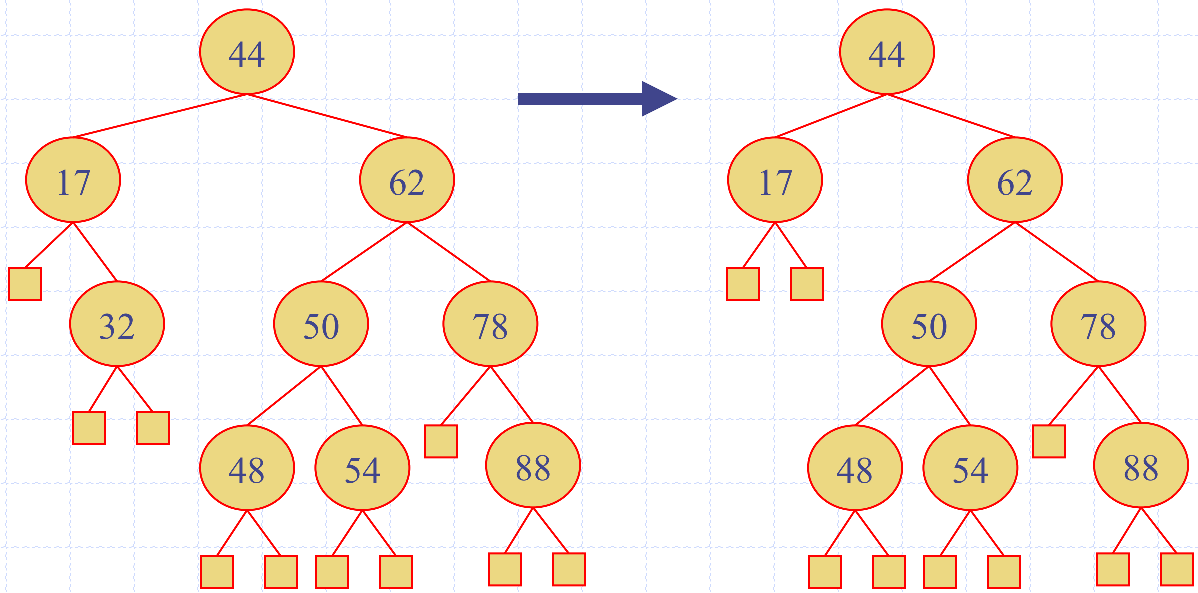
Rebalancing Operation, Case 2b

- Case 2: q's sibling has rank-difference 2.
- Let t denote a child of q that has rank-difference 1. We perform a trinode restructuring operation at t. Double-rotation rank updates:



Deletion

- Deletion begins as in a binary search tree, which means the node removed will become an empty external node. Its parent, q, may cause an imbalance.
- Example:



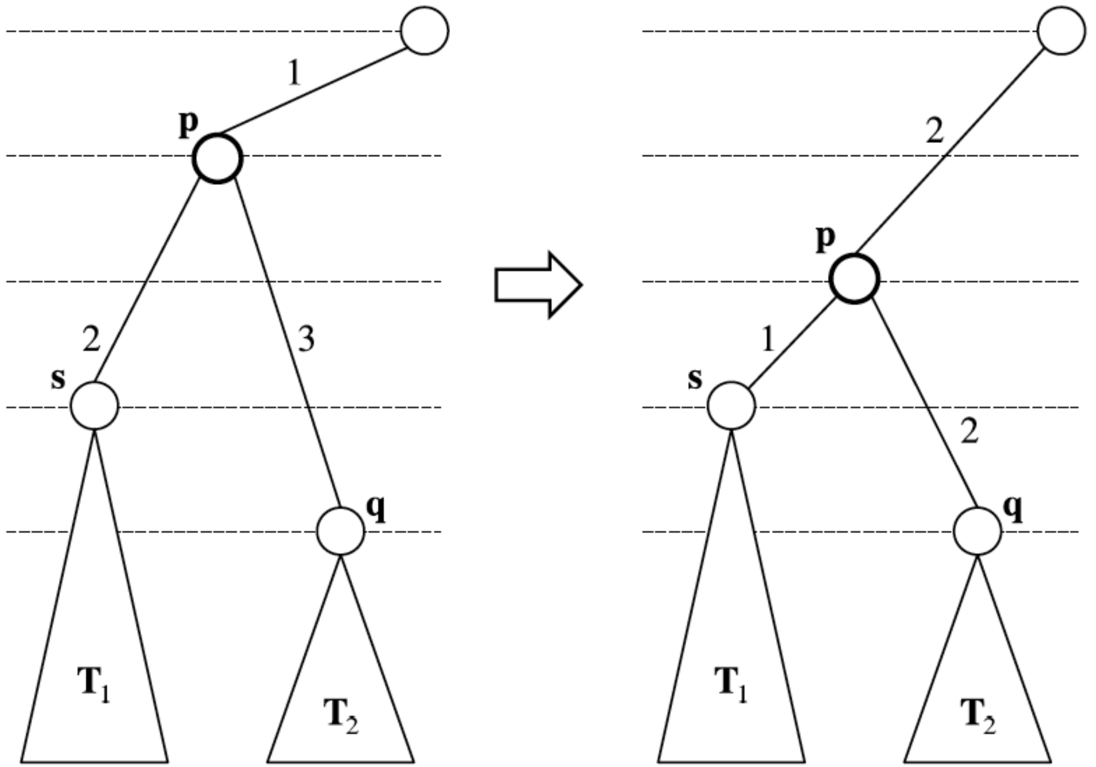
before deletion of 32 after deletion

Rebalancing after a Deletion

- Let p be the former parent of the deleted node, so now q becomes a child of p.
- Note that p is either null or has rank 2, 3, or 4 (although we cannot have q with rank 0 and p with rank 4).
- Thus, q has rank-difference 2 or 3 unless it is now the root.
- If q now has rank-difference 3, then we have a violation of the rank-difference property. Let s be the sibling of q.
- We consider two cases.

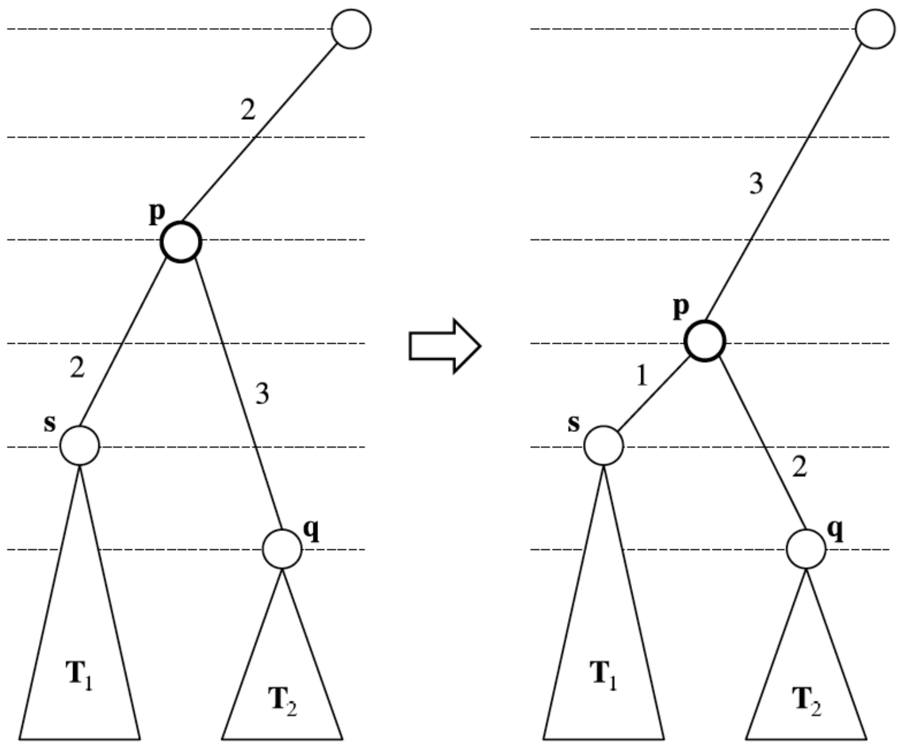
Case 1a (easy case)

- If s has rank difference 2 with p, then we reduce the rank of p by 1, which is called a **demotion**. If p still satisfies the rank-difference property, then we are done.



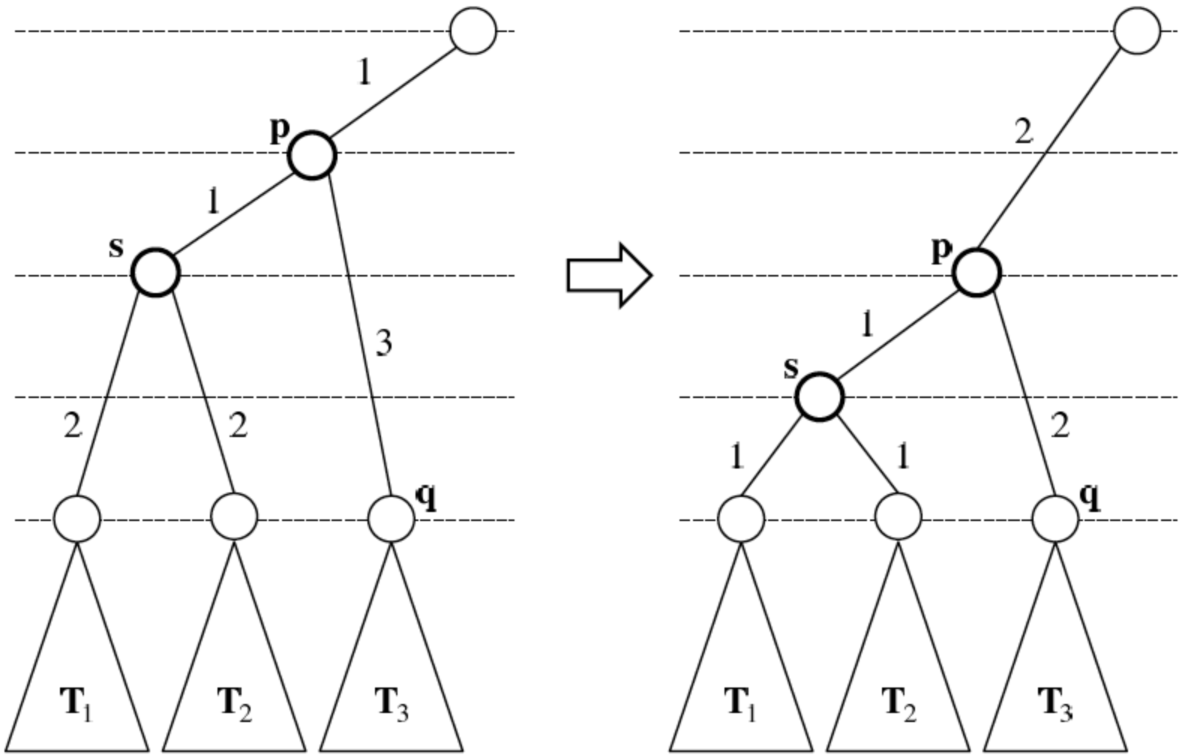
Case 1b (repeating case)

- If s has rank difference 2 with p, then we reduce the rank of p by 1, which is called a **demotion**. If p doesn't satisfy the rank-difference property, then we are repeat at p (as q).



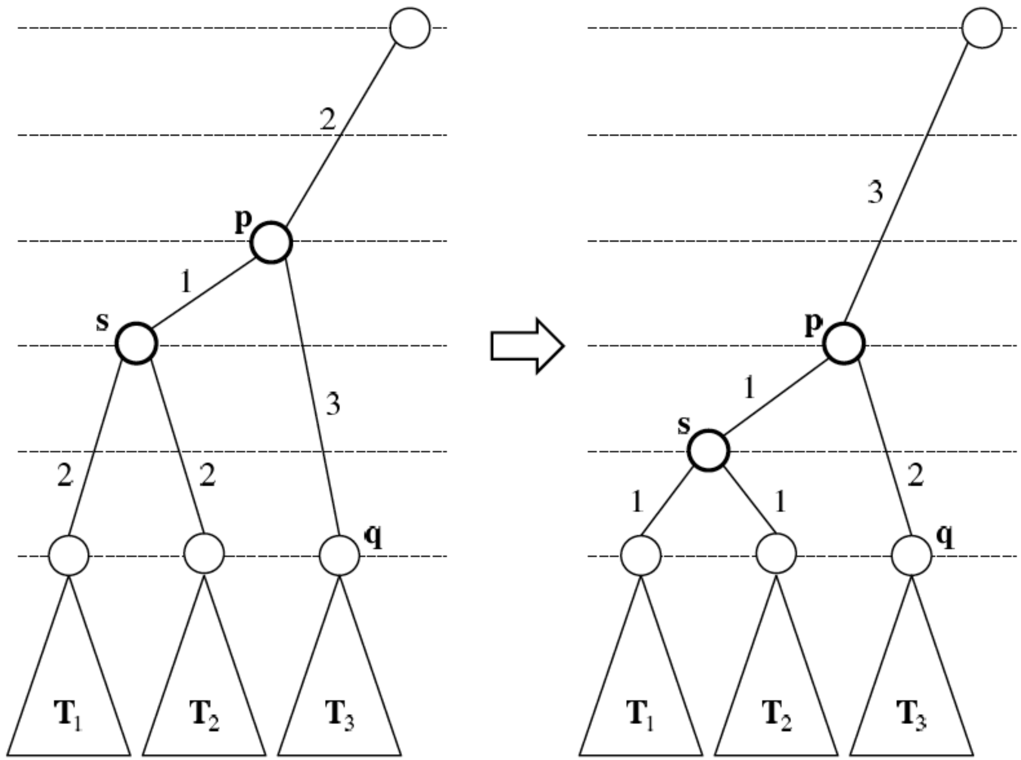
Case 2a (s has rank-diff 1 w/ p)

- If both children of s have rank-difference 2, then we demote both p and s. Easy subcase (p doesn't violate the rank-difference property and we're done):



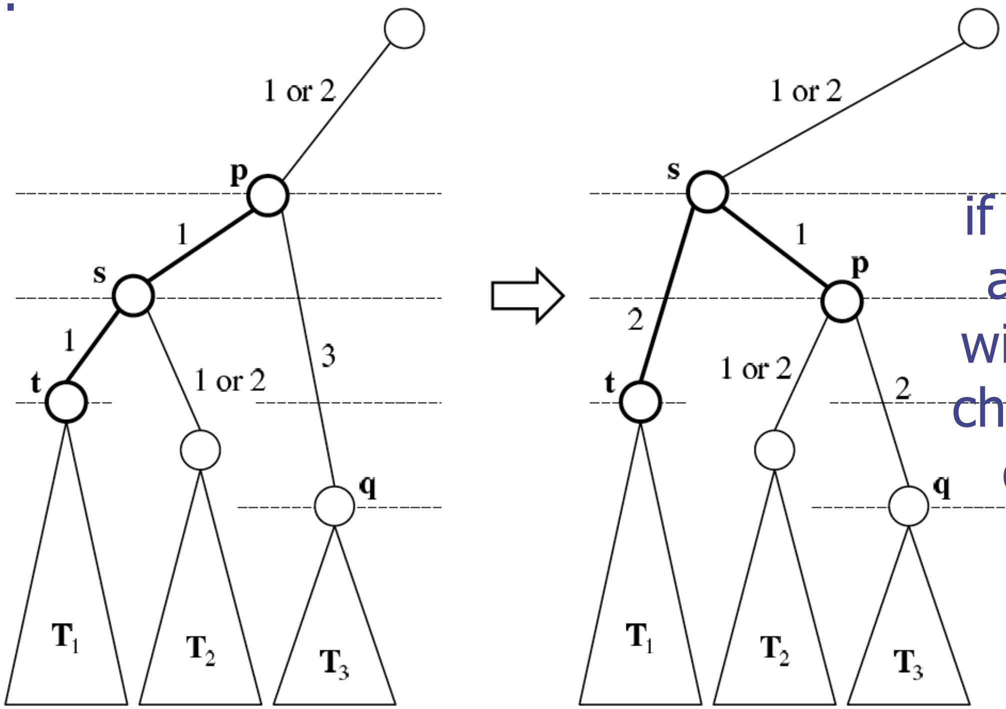
Case 2a' (s has rank-diff 1 w/ p)

- If both children of s have rank-difference 2, then we demote both p and s. Repeating subcase (p violates the rank-difference property and we repeat at p, as q):



Case 2b (s has rank-diff 1 w/ p)

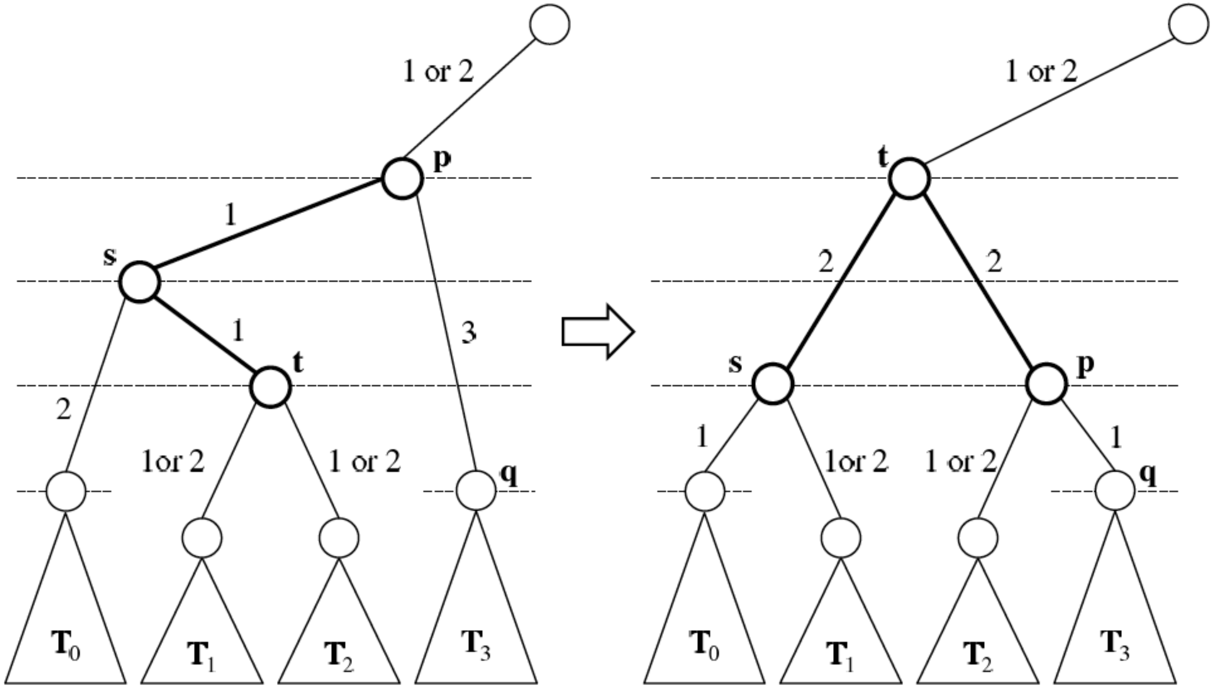
- If s has a child, t, with rank-difference 1 (when both children of s have rank-difference 1, we pick t to that if s is a left child, then t is the left child of s, else t is the right child of s). We then perform a trinode restructure at t, and reset the ranks as appropriate. Single rotation subcase:



if p becomes a 2,2-node with external children, then demote p

Case 2b' (s has rank-diff 1 w/ p)

- If s has a child, t, with rank-difference 1 (when both children of s have rank-difference 1, we pick t to that if s is a left child, then t is the left child of s, else t is the right child of s). We then perform a trinode restructure at t, and reset the ranks as appropriate. Double rotation subcase:



Weak AVL Tree Performance

- AVL tree storing n items
  - The data structure uses O(n) space
  - A single restructuring takes O(1) time
    - using a linked-structure binary tree
  - Searching takes O(log n) time
    - height of tree is O(log n), no restructures needed
  - Insertion takes O(log n) time
    - initial find is O(log n)
    - restructuring up the tree, maintaining ranks is O(log n); O(1) restructures
  - Removal takes O(log n) time
    - initial find is O(log n)
    - restructuring up the tree, maintaining ranks is O(log n); O(1) restructures

Comparison with Other Trees

- H(n) denotes worst-case height
- IH(n) denote worst-case height if built with insertions only
- IR(n) denote the worst-case number of restructures after an insertion
- DR(n) denote the worst-case number of restructures after a deletion.

	AVL trees	**red-black trees**	**wavl trees**
$H(n)$	$1.441 \log(n+1)$	$2 \log(n+1)$	$2 \log(n+1)$
$IH(n)$	$1.441 \log(n+1)$	$2 \log(n+1)$	$1.441 \log(n+1)$
$IR(n)$	1	1	1
$DR(n)$	$O(\log n)$	2	1
search time	$O(\log n)$	$O(\log n)$	$O(\log n)$
insertion time	$O(\log n)$	$O(\log n)$	$O(\log n)$
deletion time	$O(\log n)$	$O(\log n)$	$O(\log n)$