4.4 Weak AVL Trees

In this section, we discuss a type of rank-balanced trees, known as ***weak AVL trees*** or ***wavl trees***, that have features of both AVL trees and red-black trees. That is, as we will show, every AVL tree is a weak AVL tree, and every weak AVL tree can be colored as a red-black tree. Weak AVL trees achieve balance by enforcing rules about integer ranks assigned to nodes, which are a kind of "stand in" for height rather than having nodes have ranks exactly equal to their heights, as in AVL trees. Also, by using such ranks, wavl trees are able to achieve efficiency and have simple update methods.

In a ***weak AVL*** tree, T, we assign an integer rank, $r(v)$, to each node, v in T, such that the rank of any node is less than the rank of its parent. For each node v in T, the ***rank difference*** for v is 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. It is a $2,2$-***node*** if its children each have rank difference 2. And it is a $1,2$-***node*** if it has one child with rank difference 1 and one child with rank difference 2. The ranks assigned to the nodes in a wavl tree must 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 has rank 0.

***Internal-Node Property*:** An internal node with two external-node children cannot be a $2,2$-node.

Note that just by the rank-difference and external-node properties, the height of a wavl tree, T, is less than or equal to the rank of the root of T. (See Figure 4.12.)

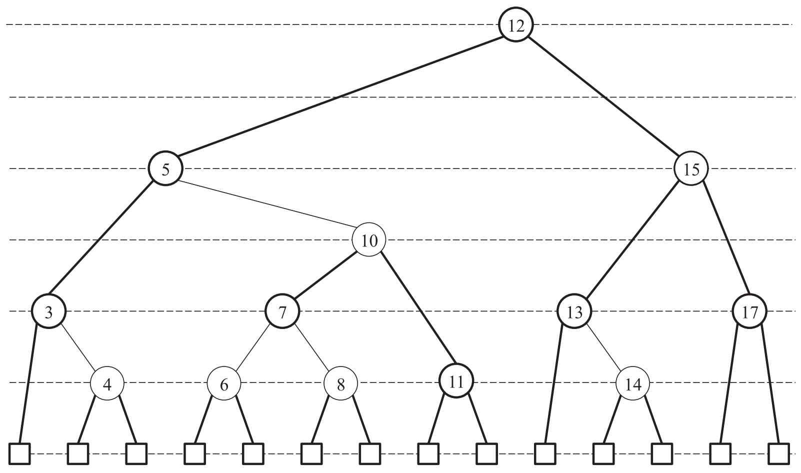


Figure 4.12: Example of a wavl tree. The nodes are placed at y-coordinates equal to their ranks.

Theorem 4.6: *The height of a wavl tree storing n items is at most $2\log(n+1)$.*

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. We also have the following.

Theorem 4.7: *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, as described in Section 4.2, so that $r(v)$ is equal to the height of v in T. Then, every external node in T has rank 0. In addition, by the height-balance property for AVL trees, every internal node is either a $1,1$-node or $1,2$-node; that is, there are no $2,2$-nodes. Thus, the standard rank assignment, $r(v)$, for an AVL tree implies T is a weak AVL tree. ■

Put another way, an AVL tree is just a weak AVL tree with no $2,2$-nodes, which motivates the name, "weak AVL tree." We also get a relationship that goes from wavl trees to red-black trees.

Theorem 4.8: *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 (e.g., see Exercise R-4.16). In particular, the internal-node property distinguishes wavl trees from red-black trees. Without this property, the two types of trees are equivalent, as Theorem 4.8 and the following theorem show.

Theorem 4.9: *Every red-black tree can be given a rank assignment that satisfies the rank-difference and external-node properties.*

Proof: Let T be a red-black tree. For each black node in v, let $r(v)$ be 2 times the black height of v. If v is red, on the other hand, let $r(v)$ be 1 more than the (same) rank of its black children. Then the rank of every external node is 0 and the rank difference for any node is 1 or 2. ■

Insertion

The insertion algorithm for weak AVL trees is essentially the same as for AVL trees, except that we use node ranks for wavl trees instead of node ranks that are exactly equal to node heights. Let q denote the node in a wavl tree 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. We then proceed according to the following rebalancing operation:

- If q has rank difference 1 after promotion, or if q is the root, then we are done with the rebalancing operation. Otherwise, if q now has rank-difference 0, with its parent, p, then we proceed according to the following two cases.
  1. q's sibling has rank-difference 1. In this case, we promote q's parent, p. See Figure 4.13. This fixes the rank-difference property for q, but it may cause a violation for p. So, we replace q by p and we repeat the rebalancing operation for this new version of q.
  2. q's sibling has rank-difference 2. Let t denote a child of q that has rank-difference 1. By induction, such a child always exists. We perform a trinode restructuring operation at t by calling the method, restructure(t), described in Algorithm 4.2. See Figure 4.14. We then set the rank of the node replacing p (i.e., the one temporarily labeled b) with the old rank p and we set the ranks of its children so that they both have rank-difference 1. This terminates the rebalancing operation, since it repairs all rule violations without creating any new ones.

As mentioned above, this insertion algorithm is essentially the same as that for AVL trees. Moreover, it doesn't create any 2, 2-nodes. Thus, if we construct a weak AVL tree, T, via a sequence of n insertions and don't perform any deletions, then the height of T is the same as that for a similarly constructed AVL tree, namely, at most $1.441 \log{(n+1)}$. (See Exercise C-4.4.)

In addition, note that we perform at most $O(1)$ restructuring operations for any insertion in a weak AVL tree. Interestingly, unlike AVL trees, this same efficiency also holds for deletion in a wavl tree.

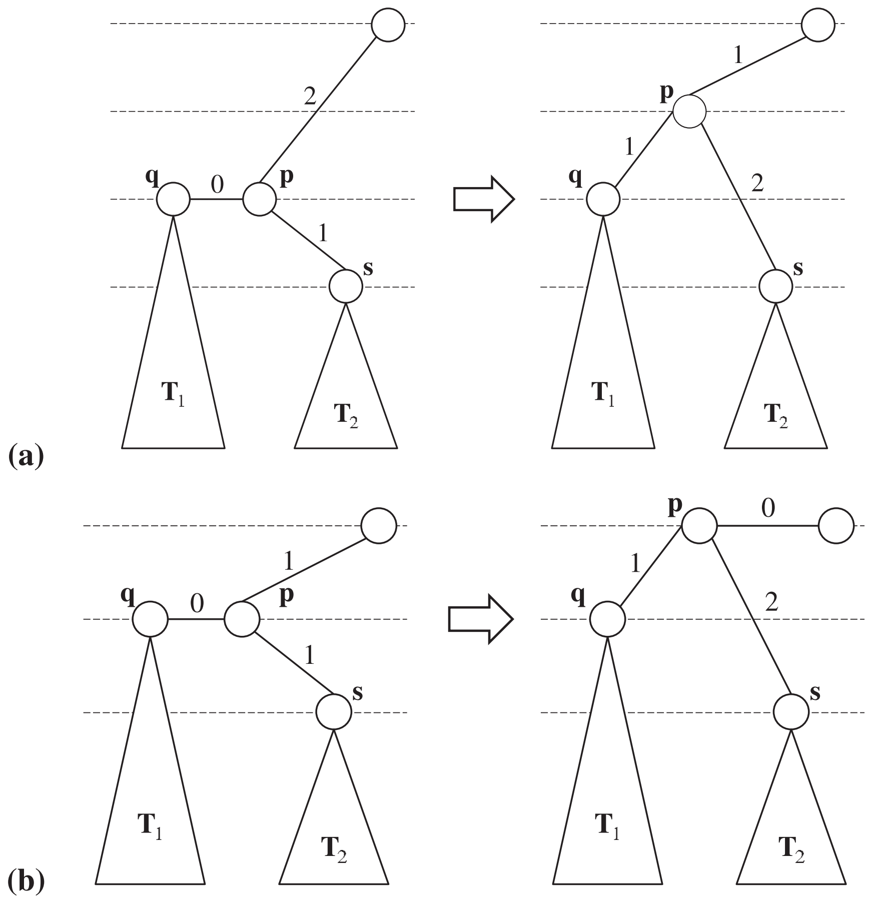
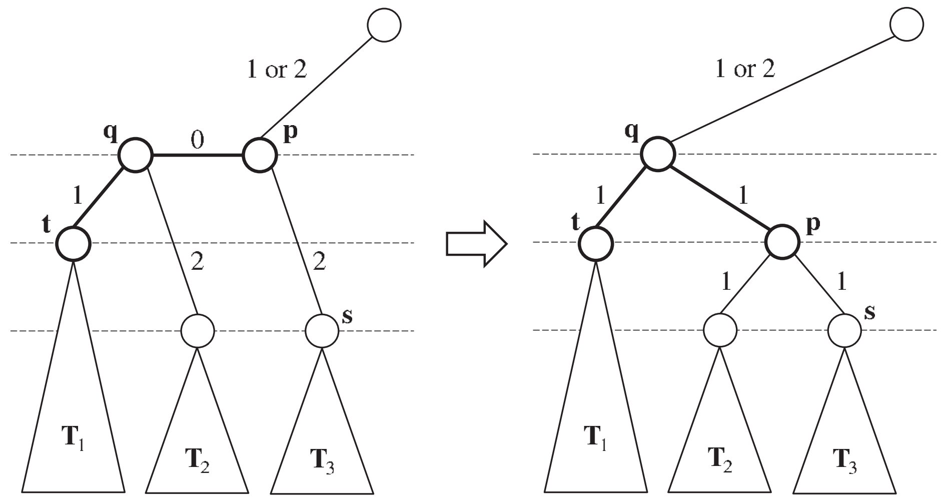


Figure 4.13: Case 1 of rebalancing in a wavl tree after an insertion: node q has rank-difference 0 (a violation) and its sibling, s, has rank-difference 1. To resolve the rank-difference violation for node q, we promote p, the parent of q. (a) If p had rank difference 2, we are done. (b) Else (p had rank-difference 1), we now have a violation at p.

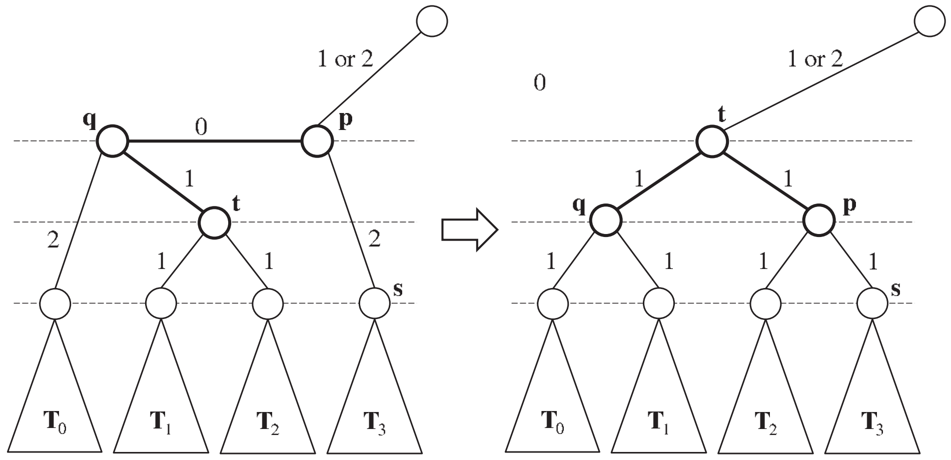
Deletion

Let us next consider how to perform a deletion in a wavl tree T. Recall that a deletion in a binary search tree operates on a node v with an external-node child u and another child q (which may also be an external node). Note that if q is an internal node, q has rank 1 and v has rank 2. Else (q is an external node), q has rank 0 and v has rank 1.

To perform the deletion, nodes u and v are removed. If v was the root, then q becomes the new root. Otherwise, let p be the former parent of v. Node q now becomes a child of p. Note that p is either null or has rank 2, 3, or 4. Still, it



(a)



(b)

Figure 4.14: Case 2 of rebalancing in a wavl tree after an insertion: node q has rank-difference 0 (a violation) and its sibling, s, has rank-difference 2. To resolve the violation for node q, we perform a trinode restructuring operation at t.

cannot be the case that q has rank 0 and p has rank 4, since that would mean v had rank 2 with two rank-0 children, which is a violation of the internal-node property. Thus, after deletion, q has rank-difference 2 or 3 unless it is the root. If q has rank-difference 3, the deletion has caused a violation of the rank-difference property. To remedy this violation we perform a rebalancing operation. Let s be the sibling of q. We consider two cases:

1. If s has rank difference 2 with p, then we reduce the rank of p by 1, which is called a ***demotion***, as shown in Figure 4.15. The demotion of p repairs the violation of the rank-difference property at q. But it may case a violation of the rank-difference property at p. So, in this case, we relabel p as q, and we repeat the rebalancing operation.
2. Else (s has rank difference 1 with p), we consider two subcases:

   (a) Both children of s have rank-difference 2. In this case, we demote both p and s, as shown in Figure 4.16. These demotions repair the rank-difference violation at q (and avoid a violation at s). But they may create a rank-difference violation at p. So, in this case, we relabel p as q, and we repeat the rebalancing operation.

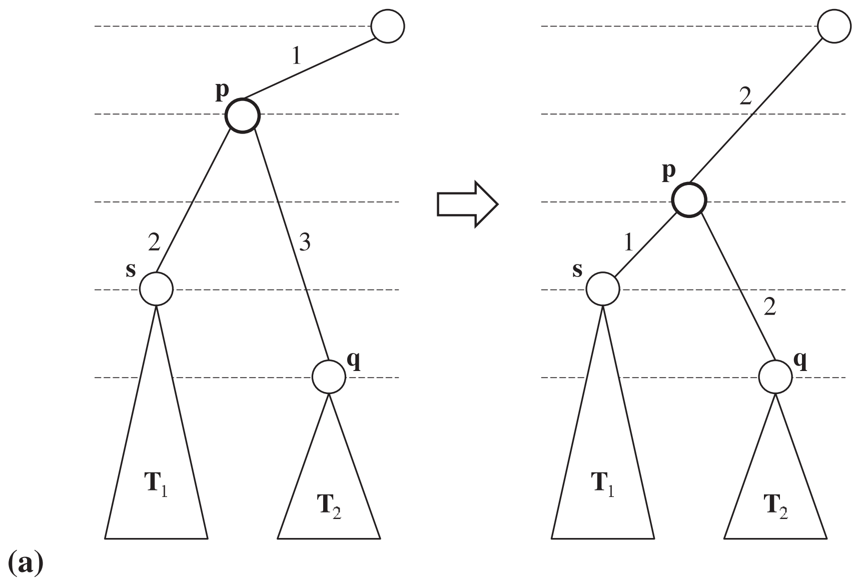
   (b) Node s has at least one child, t, with rank-difference 1. When both children of s have rank-difference 1, we pick t as follows: if s is a left child, then t is the left child of s, else t is the right child of s. In this case, we perform a trinode restructuring operation at t, by calling method, restructure(t), described in Algorithm 4.2. See Figure 4.17, which also shows how to set the ranks of p, s and t. These actions repair the rank-difference violation at q and do not create any new violations.

Thus, we can restore the balance after a deletion in a wavl tree after a sequence of at most $O(\log n)$ demotions followed by a single trinode restructuring operation.

Table 4.18 summarizes the performance of wavl trees, as compared to AVL trees and red-black trees (with black roots).

	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	$O(\log n)$	$O(\log n)$	$O(\log n)$

Table 4.18: Comparing the performance of an n-element AVL tree, red-black tree, and wavl tree, respectively. Here, $H(n)$ denotes the worst-case height of the tree, $IH(n)$ denotes the worst-case height of the tree if it is built by doing n insertions (starting from an empty tree) and no deletions, $IR(n)$ denotes the worst-case number of trinode restructuring operations that are needed after an insertion, and $DR(n)$ denotes the worst-case number of trinode restructuring operations that are needed after performing a deletion in order to restore balance. The space usage is $O(n)$ for all of these balanced binary search trees.



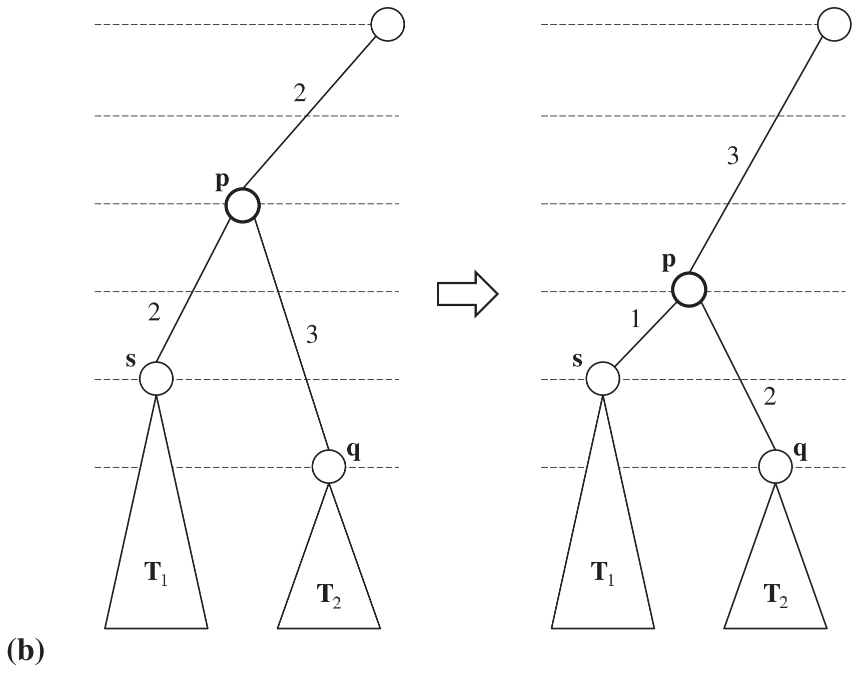


Figure 4.15: Case 1 of rebalancing in a wavl tree after a deletion: node q has rank-difference 3 (a violation) and its sibling, s, has rank-difference 2. To resolve the rank-difference violation for node q, we demote p, the parent of q. (a) If p had rank difference 1, we are done. (b) Else (p had rank-difference 2), we now have a violation at p.

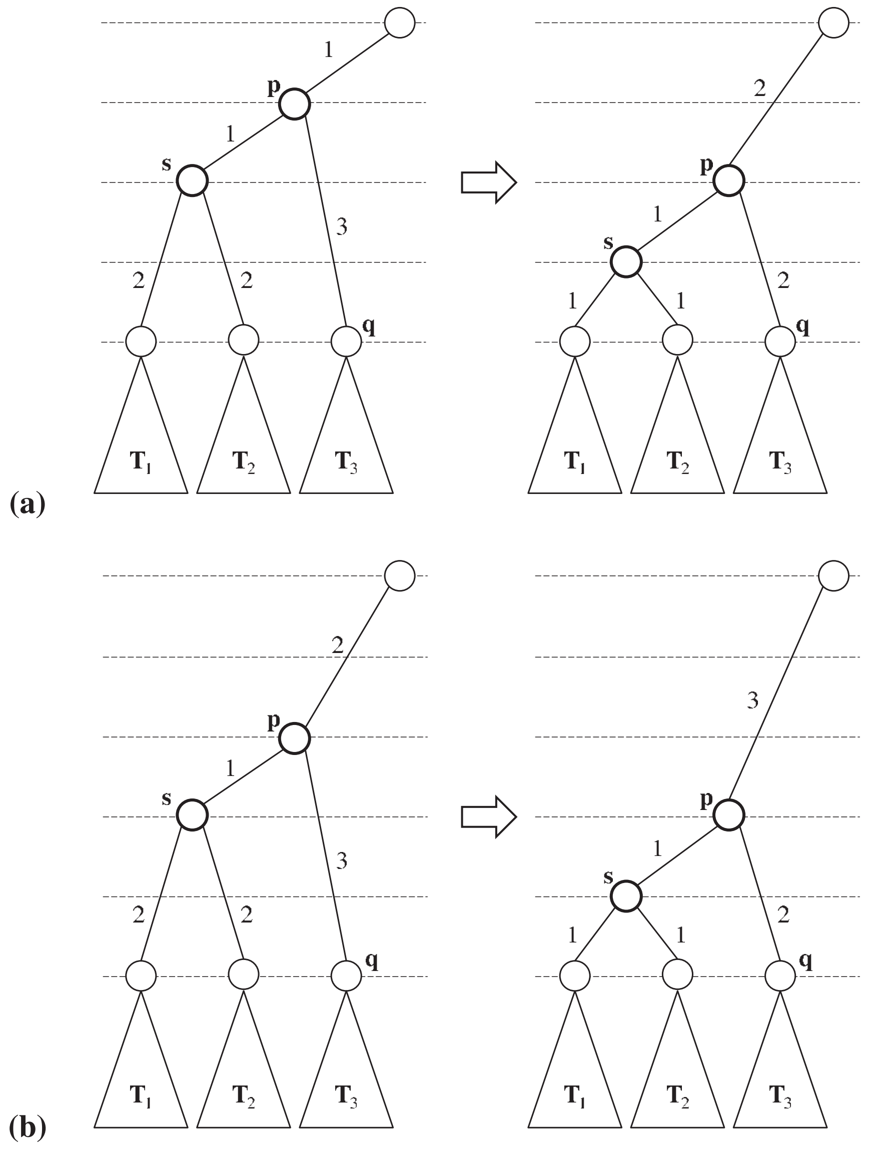


Figure 4.16: Case 2.a of rebalancing in a wavl tree after a deletion: node q has rank-difference 3 (a violation), its sibling, s, has rank-difference 1, and both children of s have rank-difference 2. To resolve the rank-difference violation for node q, we demote both s and p, the parent of q. (a) If p had rank difference 1, we are done. (b) Else (p had rank-difference 2), we now have a violation at p.

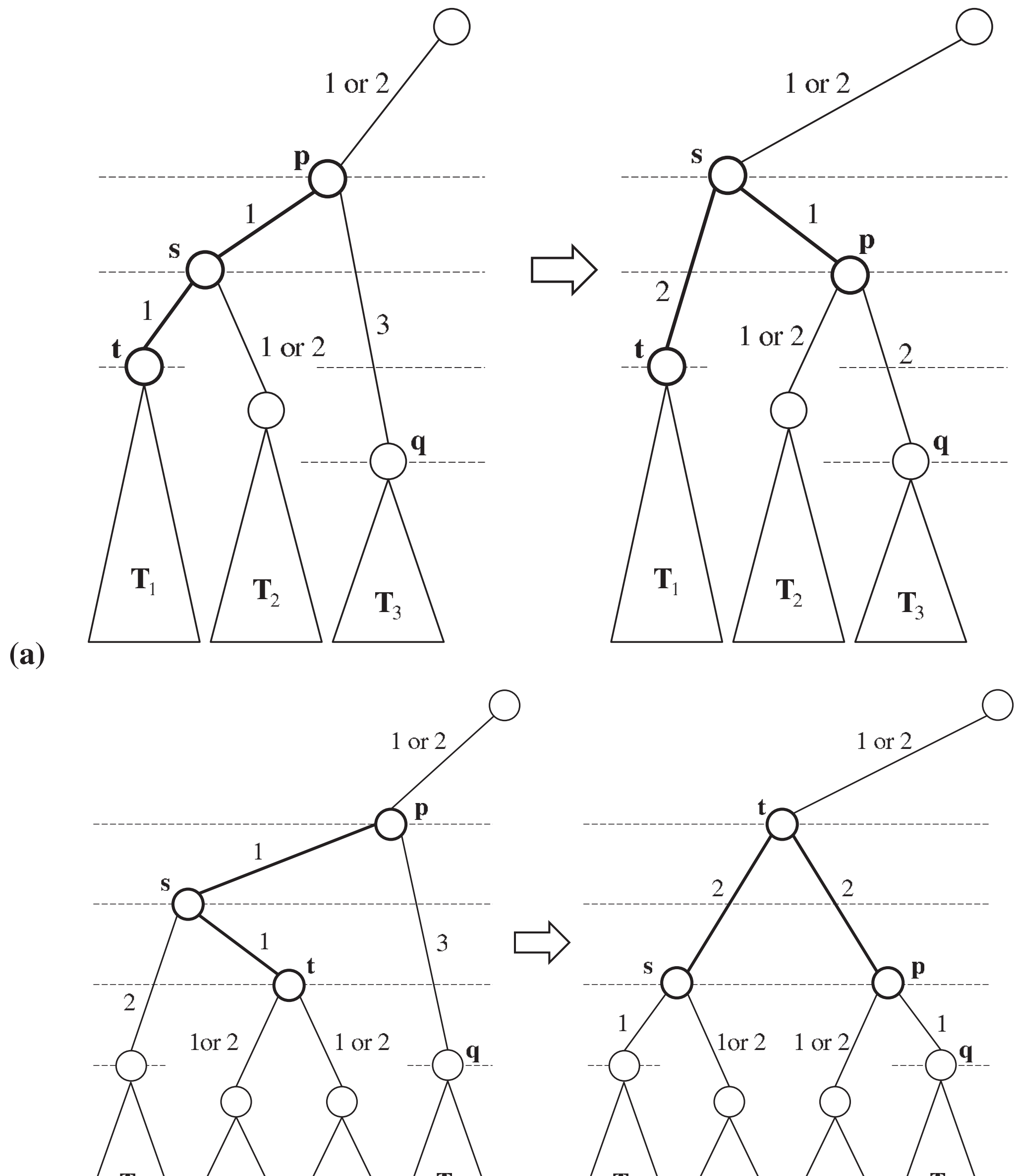


Figure 4.17: Case 2.b of rebalancing in a wavl tree after a deletion: node q has rank-difference 3 (a violation), its sibling, s, has rank-difference 1, and a child, t, of s has rank-difference 1. To resolve the violation for node q, we perform a trinode restructuring operation at t.