CHAPTER IV

PARTIAL REBUILDING

4.1. Introduction.

When the available data structures for some searching problem seem not to allow for dynamization by means of the local rebuilding technique described in Chapter 3 (as e.g. quad-trees and k-d trees) or when the local changes needed for balancing are very complex (as in e.g. super B-trees) another technique, named PARTIAL REBUILDING, can be useful. Generally speaking, the idea of partial rebuilding is the following. Assume that the (static) data structure for the searching problem in mind is a tree structure (possible augmented by associating substructures to nodes) and that there is some kind of balance criterion that each internal node should satisfy. The insertion or deletion of a point in the structure might cause some nodes, normally located on the path towards the inserted or deleted point, to become "out of balance". As we do not have a local rebuilding technique for rebalancing these nodes, we use a brute force technique: we rebuild the complete subtree at the highest node that is "out of balance" as a "perfectly balanced" subtree. This will sometimes take a lot of work, especially when the root of the tree is out of balance. But one can often prove that bigger subtrees have to be rebuilt less often than smaller ones and that the average update time will remain low. Hence, the partial rebuilding technique, in general, will yield only good average update time bounds. It is hard to give general conditions for data structures that enable the use of partial rebuilding as a dynamization method. Applications normally require (i) a lowerbound on the number of updates performed in the subtree at some internal node β before it can go out of balance and (ii) an upperbound on the cost for rebuilding the subtree at a node that has gone out of balance into some perfectly balanced tree. The analysis normally proceeds by charging the costs for rebuilding a subtree to the updates that made it go out of balance and bounding the total charge accumulated by every update that was performed on the tree. The chapter will show a number of examples to demonstrate how the technique can be used.

We will first apply the partial rebuilding technique to BB[α]-trees to make the basic idea behind the method clear. Next we will follow Lueker [Lu1] in applying the technique to super B-trees, yielding simpler (understandable and implementable) maintenance algorithms with the same update time bounds as the very complex methods that use the local rebuilding technique, but with average rather than worst-case bounds. In Sections 4.4. and 4.5. we will give the most important applications of the method, namely the maintenance of structures similar to quad- and k-d trees.

4.2. BB[α]-trees.

In a BB[α]-tree (some α, $0<\alpha<\frac{1}{2}$) it is required that for each internal node β, the number of points below the leftson of β (denoted as $n_{lson(\beta)}$) divided by the total number of points below β (i.e., n_β) must lie between α and $1-\alpha$ (see Section 3.2.2.). We will only consider leaf-search BB[α]-trees. We call a node β in perfect balance if $n_{lson(\beta)}$ and $n_{rson(\beta)}$ differ by at most 1. We call a tree perfectly balanced if each internal node is in perfect balance. Disregarding some lowest levels from consideration, a perfectly balanced tree is a BB[α]-tree for any $0<\alpha<\frac{1}{2}$.

Lemma 4.2.1. Given a node β in a BB[α]-tree that is in perfect balance. Let n_β be the number of points below β at the moment it gets out of balance. Then, there must have been $\Omega(n_\beta)$ updates in the subtree rooted at β.

Proof

Updates that are not performed in the subtree rooted at β clearly do not influence the balance at β. Let n'_β, $n'_{lson(\beta)}$ and $n'_{rson(\beta)}$ denote the number of points below β, the leftson and the rightson of β, respectively, at the moment β is in perfect balance. Assume $n'_{lson(\beta)} \leq n'_{rson(\beta)}$. Then β will become out of balance the fastest when deletions are performed below lson(β) and insertions are performed below rson(β). Let there have been N_d deletions below lson(β) and N_i insertions below rson(β) before β goes out of balance. In this case $n_\beta = n'_\beta + N_i - N_d$ and $n_{lson(\beta)} = n'_{lson(\beta)} - N_d = \lfloor \frac{1}{2} n'_\beta \rfloor - N_d$. Since β is now out of balance, we have

$$\frac{n_{lson(\beta)}}{n_\beta} < \alpha$$

$$\Rightarrow \frac{\lfloor \frac{1}{2} n'_\beta \rfloor - N_d}{n_\beta} < \alpha$$

$$\Rightarrow \tfrac{1}{2}(n_\beta - N_i + N_d) - 1 - N_d < \alpha n_\beta$$

$$\Rightarrow -\tfrac{1}{2}N_i - \tfrac{1}{2}N_d < (\alpha - \tfrac{1}{2})n_\beta + 1$$

$$\Rightarrow N_i + N_d > 2(\tfrac{1}{2} - \alpha)n_\beta - 2$$

This is $\Omega(n_\beta)$ because $\alpha<\frac{1}{2}$.

□

Lemma 4.2.2. Given an ordered set of n points, a perfectly balanced binary tree of them can be built in O(n) time.

Proof

This can be achieved by building the tree levelwise, starting at the leaves.

□