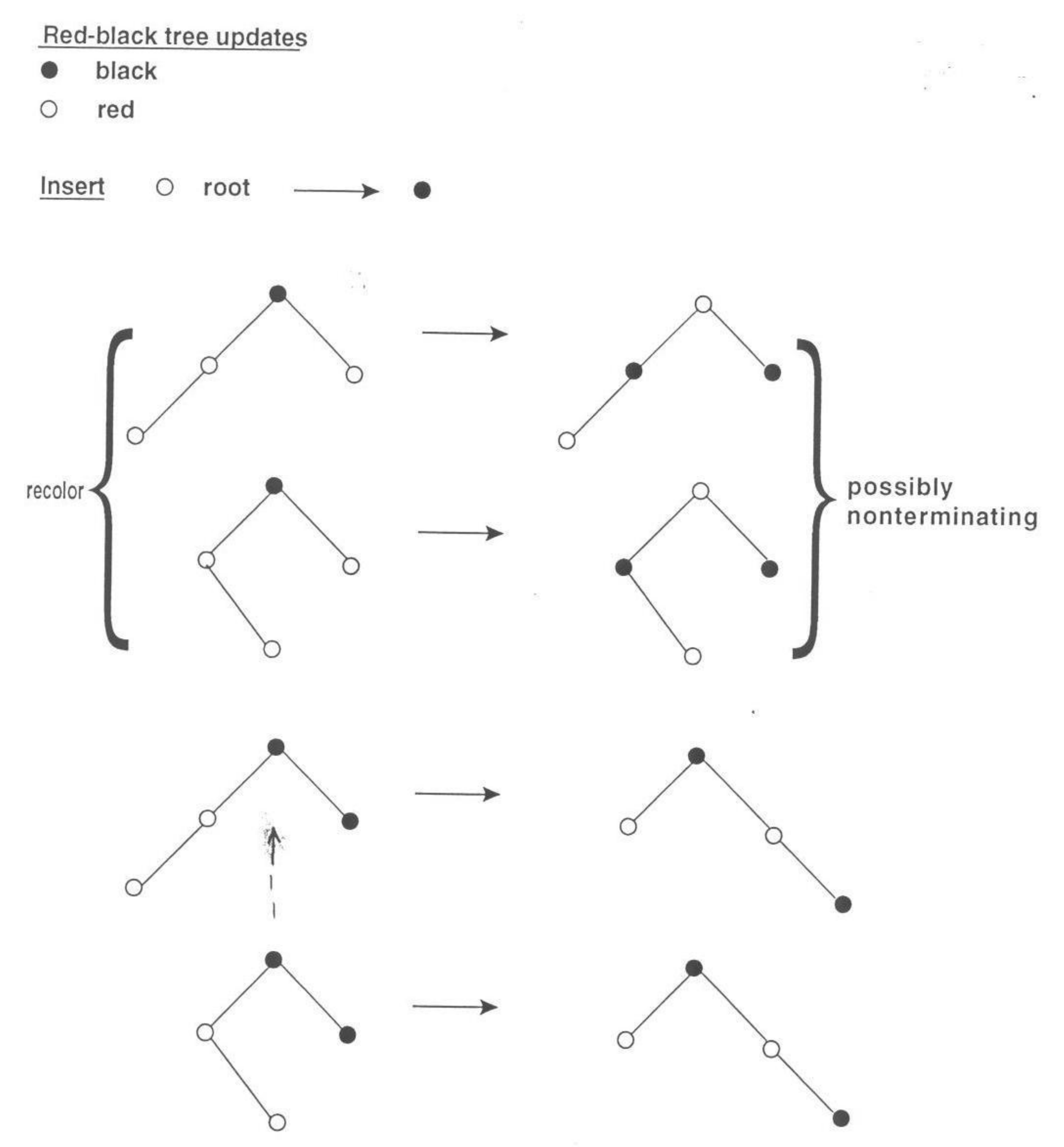
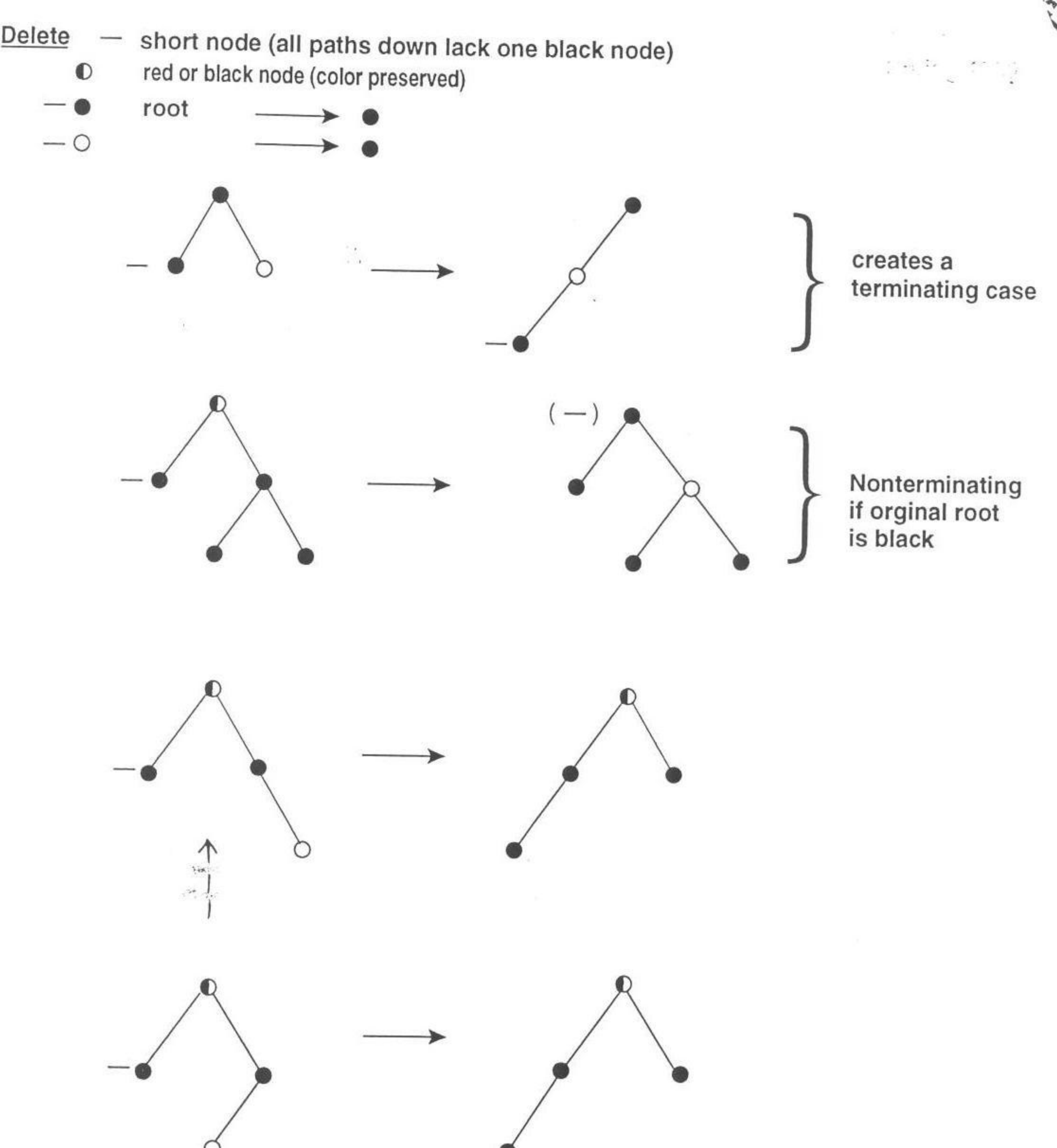

Red-black tree updates
black
red
Insert
root
recolor
possibly
nonterminating



O(log n) recolorings; 0, 1, 2, or 3 rotations

O(1) amortized recoloring time for insert/delete:

Φ = 2 for [black node with two white (red) children], 1 for [black node with two black children]

and s_2 the right child of i. We define the rank of i to be $rank(x) + 1$, thus making x and s_2 black. Then we rebalance as in an insertion, starting from i. The case $rank(s_1) < rank(s_2)$ is symmetric. Such a join takes $O(|rank(s_1) - rank(s_2)|) = O(\log n)$ time and produces a tree whose root has rank $\max\{rank(s_1), rank(s_2)\}$ or $\max\{rank(s_1, rank(s_2)\} + 1$; in the latter case both children of the root are black.

To split a balanced binary tree at an item i, we perform a sequence of joins exactly as described in §4.1, using the method above for each join. The time bounds for the joins that construct the left tree form a telescoping series summing to $O(\log n)$. This is also true for the joins that construct the right tree. More precisely, the split algorithm maintains the following invariant, where x, y, s_1 and s_2 are the current node, the previous node and the roots of the left and right trees, respectively (see the description of split in §1): If y was black in the unsplit tree, then $\max\{rank(s_1), rank(s_2)\} \leq rank(x)$, and the time spent on the split so far is $O(rank(s_1) + rank(s_2) + rank(x))$. Thus a split takes $O(\log n)$ time.

In addition to having an $O(\log n)$ time bound for each of the sorted set operations, balanced binary trees have other useful properties. All the operations have alternative top-down implementations that make concurrent search and updating easy [7]. Bottom-up rebalancing after an insertion or a deletion, in addition to needing only $O(1)$ rotations, takes $O(1)$ total time if the time is amortized over a sequence of updates [8], [9], [11]. Furthermore, time-consuming insertions and deletions are rare; in particular, operations that take time t occur with a frequency that is an exponentially decreasing function of t [8], [9]. This makes balanced binary trees ideal for use as "finger" search trees, in which we maintain one or more pointers to search starting points; the time for an access, insertion, or deletion is $O(\log d)$, where d is the number of items between the search starting point and the accessed item [8], [9], [11].

4.3. Self-adjusting binary trees. There is another way to obtain $O(\log n)$ operation times in binary search trees. If we are willing to settle for an amortized rather than worst-case time bound per operation, we do not need to maintain any explicit balance condition. Instead, we adjust the tree every time we do an access or update, using a method that depends only on the structure of the access path. Allen and Munro [2] and Bitner [5] studied such restructuring heuristics for binary trees, but none of their methods has an $O(\log n)$ amortized time bound; they are all $O(n)$. Sleator and Tarjan [15] recently discovered a heuristic that does give an $O(\log n)$ amortized time bound, which we shall study here.

The restructuring heuristic proposed by Sleator and Tarjan is the *splay* operation. To splay a binary tree at an internal node x, we begin at x and traverse the path to the root, performing a single rotation at each node. We perform the rotations in pairs, in an order that depends on the structure of the tree. The splay consists of repeating the following *splay step* until $p(x)$ is undefined (see Fig. 4.9): If x has a parent but no grandparent, we rotate at $p(x)$. If x has a grandparent and x and $p(x)$ are both left or both right children, we rotate at $p^2(x)$ and then at $p(x)$. If x has a grandparent and x is a left and $p(x)$ a right child, or vice versa, we rotate at $p(x)$

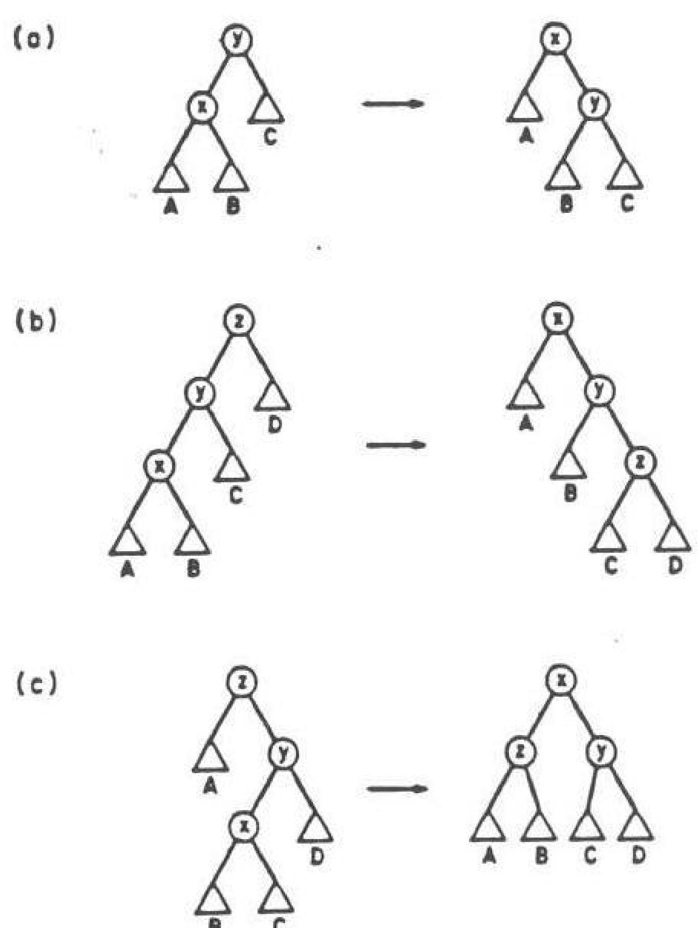


FIG. 4.9. *Cases of splay step. Each case has a symmetric variant* (*not shown*). (a) *Terminating single rotation.* (b) *Two single rotations.* (c) *Double rotation.*

and then at the new parent of x (the old grandparent of x). The overall effect of the splay is to move x to the root of the tree while rearranging the rest of the original path to x. (See Fig. 4.10.)

To make binary search trees self-adjusting, we perform a splay during each access or update operation other than a join. More precisely, after accessing or inserting an item i we splay at i. After deleting an item i we splay at its parent just before the deletion (this is the original parent of i's predecessor if i and the predecessor were swapped). Before splitting at an item i we splay at i. This makes the split easy, since it moves i to the root: we merely return the left and right subtrees of the root. In each case the time for the operation is proportional to the length of the path along which the splay proceeds, which we call the *splay path*. Although we have described splay as a bottom-up operation, an appropriate variant will work top-down [15]. Thus for example when accessing or inserting an item i we can splay at i while we search for i.

Splaying is reminiscent of path compression and even more of path halving (see Chapter 2) in the way it changes the tree structure. Although the techniques that Tarjan and van Leeuwen [17] used to analyze path halving also apply to splaying,

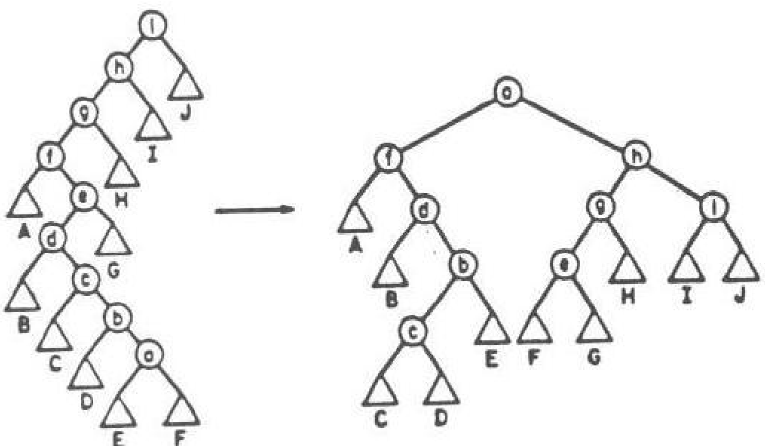


FIG. 4.10. *Splay at node a.*

another method gives a simpler analysis. In the remainder of this section we shall derive Sleator and Tarjan's $O(m \log n)$ bound on the total time required for a sequence of m sorted set operations beginning with no sorted sets, where n is the number of insert and join operations (items).

We use a credit-debit accounting scheme (see Chapter 2), keeping track only of credits. We assign to each item i an *individual weight* $iw(i)$. The individual weights are arbitrary positive real numbers, whose value we shall choose later. We define the *total weight* $tw(x)$ of a node x in a binary search tree to be the sum of the individual weights of all descendants of x, including x itself. Finally, we define the *rank* of a node x to be $rank(x) = \lfloor \lg tw(x) \rfloor$. We maintain the following *credit invariant:* any internal node x holds $rank(x)$ credits.

LEMMA 4.2. *Splaying a tree with root v at a node x while maintaining the credit invariant takes $3(rank(v) - rank(x)) + 1$ credits.*

Proof. Consider a splay step involving the nodes x, $y = p(x)$, and $z = p^2(x)$, where p denotes the parent function before the step. Let rank and rank′, tw and tw′ denote the rank and total weight functions before and after the step respectively. To this step we allocate $3(rank'(x) - rank(x))$ credits and one additional credit if this is the last step. There are three cases (see Fig. 4.9).

Case 1. *Node z is undefined.* This is the last step of the splay, and the extra credit allocated to the step pays for it. We have $rank'(x) = rank(y)$. Thus the number of credits that must be added to the tree to maintain the invariant is $rank'(y) - rank(x) \le rank'(x) - rank(x)$, which is one third of the remaining credits on hand.

Case 2. *Node z is defined and x and y are both left or both right children.* We have $rank'(x) = rank(z)$. The number of credits that must be added to the tree to maintain the invariant is $rank'(y) + rank'(z) - rank(y) - rank(x) \le 2(rank'(x) - rank(x))$, which is two thirds of the credits on hand. If $rank'(x) > rank(x)$, there is at least one extra credit on hand to pay for the step. Otherwise, $rank'(x) = rank(x) = rank(y) = rank(z)$. Furthermore $tw(x) + tw'(z) \le tw'(x)$, which implies that $rank'(z) < rank'(x)$. (If

$rank'(z) = rank'(x)$, $tw'(x) > 2^{rank(z)} + 2^{rank'(z)} \geq 2^{rank'(x)+1}$, a contradiction.) Maintaining the invariant thus frees $rank(y) + rank(x) - rank'(y) - rank'(z) \geq 1$ credits, one of which will pay for the step.

Case 3. *Node z is defined and x is a left and y is a right child, or vice versa.* The credit accounting in this case is the same as in Case 2, with the following exception: If $rank'(x) = rank(x)$, we have $\min\{rank'(y), rank'(z)\} < rank'(x)$ since $tw'(y) + tw'(z) \leq tw'(x)$, and maintaining the invariant frees $rank(y) + rank(x) - rank'(y) - rank'(z) = 2\ rank'(x) - rank'(y) - rank'(z) \geq 1$ credit to pay for the step.

Summing over all steps of a splay, we find that the total number of credits used is $3(rank'(x) - rank(x)) + 1 = 3(rank(v) - rank(x)) + 1$, where $rank$ and $rank'$ denote the rank function before and after the entire splay. □

In order to complete the analysis, we must consider the effects of insertion, deletion, join and split on the ranks of nodes. To simplify matters, let us define the individual weight of every item to be 1. Then every node has a rank in the range $[0 .. \lfloor \lg n \rfloor]$, and Lemma 2 gives a bound of $3\lfloor \lg n \rfloor + 1$ credits for splaying. Inserting a new item i without splaying takes at most $\lfloor \lg n \rfloor + 1$ credits, since only nodes along the path from i to the root of the tree gain in total weight, and of these only the ones with an old total weight of $2^k - 1$ for some $k \in [0 .. \lfloor \lg n \rfloor]$ gain (by one) in rank. Deleting an item i without splaying causes either a net decrease or no change in the number of credits in the tree. Joining two trees requires placing at most $\lfloor \lg n \rfloor$ credits on the new root, and splitting a tree frees the credits (if any) on the old root. Thus we have the following theorem:

THEOREM 4.1. *The total time required for a sequence of m sorted set operations using self-adjusting trees, starting with no sorted sets, is $O(m \log n)$, where n is the number of insert and join operations (items).*

A more refined analysis using weights other than one shows that self-adjusting search trees are as efficient in an amortized sense as static optimum search trees, to within a constant factor [15]. Self-adjusting trees have other remarkable properties [15]. In the next chapter we shall study a problem in which the use of self-adjusting search trees gives an asymptotically more efficient algorithm than seems possible with any kind of balanced search trees.

References

[1] G. M. ADEL'SON-VEL'SKII AND E. M. LANDIS, *An algorithm for the organization of information*, Soviet Math. Dokl., 3 (1962), pp. 1259–1262.

[2] B. ALLEN AND I. MUNRO, *Self-organizing search trees*, J. Assoc. Comput. Mach. 25 (1978), pp. 526–535.

[3] R. BAYER, *Symmetric binary B-trees: Data structure and maintenance algorithms*, Acta Inform., 1 (1972), pp. 290–306.

[4] R. BAYER AND E. MCCREIGHT, *Organization of large ordered indexes*, Acta Inform., 1 (1972), pp. 173–189.

[5] J. R. BITNER, *Heuristics that dynamically organize data structures*, SIAM J. Comput., 8 (1979), pp. 82–110.

[6] C. A. CRANE, *Linear lists and priority queues as balanced binary trees*, Tech. Rep. STAN-CS-72-259, Computer Science Dept., Stanford Univ., Stanford, CA, 1972.