[6] L. J. Guibas, E. M. McCreight, M. F. Plass, and J. R. Roberts "A new representation for linear lists," *Proc. Ninth Annual ACM Symposium on Theory of Computing*, (1977), pp.49-60.

[7] L. J. Guibas, R. Sedgewick, "A dichromatic framework for balanced trees," *Proc. Nineteenth Annual Symposium on Foundations of Computer Science*, (1978), pp. 8-21.

[8] S. Huddleston and K. Mehlhorn, "A new data structure for representing linear lists," *Acta Informatica* 17 (1982), pp. 157-184.

[9] S. R. Kosaraju, "Localized search in sorted lists," *Proc. Thirteenth Annual ACM Symposium on Theory of Computing*, 1981, pp. 62-69.

[10] D. Maier and C. Salveter, "Hysterical B-trees," *Inform. Proc. Lett.*, 12 (1981), pp. 199-202.

[11] K. Mehlhorn, *Data Structures and Efficient Algorithms, Volume 1: Sorting and Searching* Springer-Verlag, Berlin, 1984.

[12] D. D. Sleator, Personal communication.

[13] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *Journal of the ACM*, 32:3 (1985), pp. 652-686.

[14] R. E. Tarjan, "Amortized computational complexity," *Siam J. Alg. Disc. Methods*, 6, No. 2, (1985), pp. 306-318.

[15] R. E. Tarjan and C. J. Van Wyk, "An $O(n \log \log n)$-time algorithm for triangulating a simple polygon," *SIAM J. Comput.*, 17 (1988), pp. 143-178.

[16] A. K. Tsakalidis, "AVL-trees for localized search," *Lecture Notes in Computer Science 172*, J. Paredaens, ed., Springer-Verlag, Berlin, 1984, pp. 473-485.

3 Remarks

Finger trees have a reputation for being complicated and impractical, hard to understand and hard to implement. If one considers only the operations insert, delete, and access, then the finger tree implementations given here are not significantly more complex than those of regular balanced trees. Concatenating and splitting are more complex and allowing add value operations adds still more details.

In this variant the path nodes serve only to link together a bunch of red-black subtrees and to impose regulations on their sizes so that the search time is optimal. A natural question is whether this observation can be generalized to create a simpler data structure with the same kind of performance. It is not necessary to think of the set of subtrees plus the linking paths as a tree itself, especially since the linking nodes already have to be treated differently from the subtree nodes. However, treating the whole structure as a tree allows us to use standard tree rebalancing operations (i.e., rotations) in order to maintain the balance condition in the entire structure. Splay trees, a form of self-adjusting search tree introduced by Sleator and Tarjan [13], are more elegant and easier to implement. It is not known if they perform as well, however.

References

[1] A. Aho, J. Hopcroft, and J. Ullman, *Design and Analysis of Computer Algorithms,* Addison-Wesley, 1974.

[2] G. Adelson-Velskii and E. Landis, "On an information organization algorithm," *Doklady Akademica Nauk SSSR*, 146 (1962), pp.263-266.

[3] R. Bayer, "Symmetric binary B-trees: data structure and maintenance algorithms," *Acta Informatica*, 1 (1972), pp. 290-306.

[4] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indices," *Acta Informatica*, 1 (1972), pp. 173-179.

[5] M.R. Brown and R. E. Tarjan, "Design and analysis of a data structure for representing sorted lists," *SIAM J. Comput.*, 9 (1980) pp. 594–614.

subtree. If the item is in the right subtree the algorithm requires $O(\log n)$ time, but using the reasoning used in Lemma 2 we can show that this is also $O(1 + \log d)$ in this case.

To add a constant k to the *key1* values of all external nodes, add k to the $\Delta maxkey$ values of all non-path children of path nodes. Given two external nodes x and y and their access paths, we can also add k to the values of all external nodes between x and y. Both these operations require $O(\log n)$ time on an n-node tree. The first operation is slower than for the Tarjan and Van Wyk finger trees in which adding a constant to all nodes of the tree requires constant time. However, by storing a *cost offset* value with the root of each modified finger tree, we can perform *add value* operations in $O(1)$ amortized time. Adding a constant k to this value increases by k the costs associated with all leaves [12]. Such a scheme entails more work in maintaining data after a concatenation or a split operation, but this adds only a constant factor to the running time.

It is easier to maintain the auxiliary data in this modified finger tree. Since no data is associated with the path nodes, the reversed pointers do not need to be taken into account. In contrast, with the finger trees described previously, rotating at the root does not require updating data of nodes on the left and right paths.

Since the subtree containing a particular external node is a regular red-black tree, updating data after an insertion or deletion is handled exactly as described in Section 2.2. Creating or destroying a finger also invalidates data. Updating data after such an operation is straightforward. If we wish to add a finger by reversing all the pointers along the left or right path of a tree, first traverse the path once to compute *maxkey* values for the path nodes and their siblings. For each sibling replace $\Delta maxkey$ by *maxkey*. To destroy a finger again traverse the path once computing and storing *maxkey* values (with respect to downward pointers) in place of $\Delta maxkey$ values for the nodes on the path and their siblings. Then traverse the path once more to compute new $\Delta maxkey$ values for all these nodes. Store *maxkey* with the root. In addition to reversing the pointers and updating data, when deleting a finger we must also delete the corresponding special external node and when adding a finger we must insert a null external node.

stored in difference form.

Using this data structure we can perform the same operations. With respect to searching it is similar to a finger tree with only a left finger. Accessing an item requires $O(1 + \log d)$ time if the item found is the dth. Again, insertion, deletion, and splitting require $O(1 + \log(\min\{d, n - d\}))$ time at the dth external node in an n-node tree. Concatenation requires time proportional to the rank of the smaller tree. Adding a constant to the values of all external nodes (or to the values of a consecutive subsection of the external nodes) requires $O(\log n)$ time in an n-node tree.

Implementation of insertion, deletion, concatenation, and splitting is the almost exactly the same as for regular finger trees. The only difference lies in the data updating and searching procedures.

Searching is slower since it is no longer possible to detect in constant time whether the desired item is reachable from the left finger. To access an item, first search up the left path until finding a subtree containing the item or until reaching the root. If no subtree is found, search down the right path (which first requires walking up the entire path) until reaching a subtree containing the desired item or until reaching the right finger. If we reach the right finger, then the item does not exist. Otherwise, search for the item in the found subtree as in a regular red-black tree.

In the search for the leftmost item with key at least k, we find the leftmost path node whose non-path child u has $\Delta maxkey \geq k$ and then search down within u's subtree as described in Section 2.2.

To find the leftmost item with cumulative value at least k we again maintain the value $maxcumkey$. To update this value when going from a node u on the left path to its left child, v (if v is not the root), add $totkey(right(v))$. If $maxcumkey(u) < k$ for all nodes u on the left path, then the search continues from u, the right child of the root, starting with $maxcumkey$ incremented by $totkey(left(u))$. To update $maxcumkey$ when going from node on the right path to its predecessor v, add $totkey(left(v))$. The leftmost node with $maxcumkey \geq k$ is contained in the non-path subtree of the leftmost path node having $maxcumkey \geq k$. The procedure for searching within this subtree was described in Section 2.2.

If the desired item is not found, then both search procedures described above will take $O(\log n)$ time in an n-node tree. If the desired item is the dth, then the search will require $O(1 + \log d)$ time. Lemma 2 implies that the length of the access path for the dth item is $O(1 + \log d)$ if it is in the left

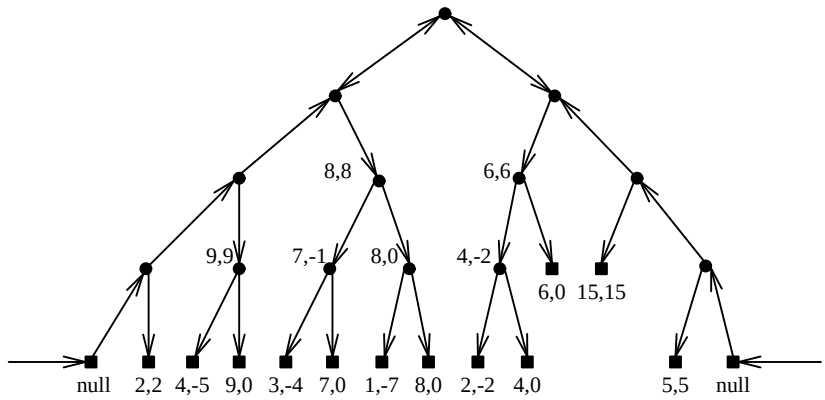


Figure 13: *An example of a finger tree with no path data. Shown with each node x are the values* maxkey(x) *and* Δmaxkey(x) *(*maxkey *values are not stored).*

fact that the rebalancing procedure reached the root implies that $O(\log n)$ work was already done, either in rebalancing (in an insertion or deletion) or in restructuring and rebalancing (in a concatenation). This implies that rebalancing still requires only $O(1)$ time in the amortized sense.

Finger search trees can also support add value operations. Again, the *maxkey* values must be stored in difference form as described in Section 2.2 for red-black trees. Maintaining this data with the reversed pointers, in particular, recomputing auxiliary data when reversing the pointers along a path, is somewhat involved. We describe here a simpler, but slightly less powerful, type of finger search tree in which data is stored only with the nodes not on the left or right paths. This type of finger tree is used for the application described in Chapter 4.

In order to represent a list having n items, create a finger search tree having $n + 2$ external nodes; the leftmost and rightmost nodes represent null items. The remaining external nodes are associated with list items in order, as before. For both null external nodes set *key1* to minus infinity and *key2* to 0. Store auxiliary data only with the non-path nodes. For each non-path node y store $totkey(y)$ and $\Delta maxkey(y)$. The latter value is $maxkey(y) - maxkey(p(y))$ if $p(y)$ is not a path node and is $maxkey(y)$ otherwise. An example illustrating the data is shown in Figure 13.

Notice that the non-path subtrees are regular red-black trees with *maxkey*

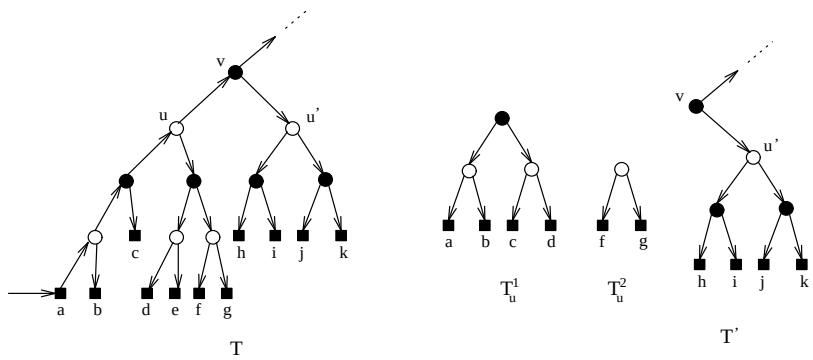


Figure 12: *Splitting finger tree T at node e. Shown at right are the trees formed during the splitting process; the trees T_u^1 and T_u^2 left after splitting T_u at e and the fragment of T left after deleting T_u.*

v requires $O(1)$ amortized time. Concatenating T' and T'_u requires $O(1 + \log d)$ time also. Thus the amortized time for splitting T at the dth node is $O(1 + \log(\min\{d, n-d\}))$.

To restore the balance condition after a tree restructuring operation (eg, insertion, deletion, or concatenation), as for regular red-black trees, we use the rebalancing transformations of Figures 2 and 3. These operations are applied to the path from the violation up to the root, ignoring the reversed pointers. In order to traverse this path without storing parent pointers, the portion (if any) that is not on the left or right path must be stored. Note that in a regular red-black tree without parent pointers the entire path from the root must be stored in a similar manner.

Auxiliary data must again be updated after rotations and restructuring operations. For a node x, the auxiliary value stored at x becomes invalid only if the set of nodes reachable from x changes. First consider how rotations affect the data.

For rotations involving only non-path nodes, the effect and subsequent recomputation is the same as in regular red-black trees; if path nodes are involved the reversed pointers must be taken into account. A rotation at the root is more complicated. Such an operation changes the set of reachable nodes for all nodes on both the right and left paths. To restore these auxiliary values requires traversing both paths top-down, recomputing the data at each node. This requires $O(\log n)$ time in an n-node tree. However, the

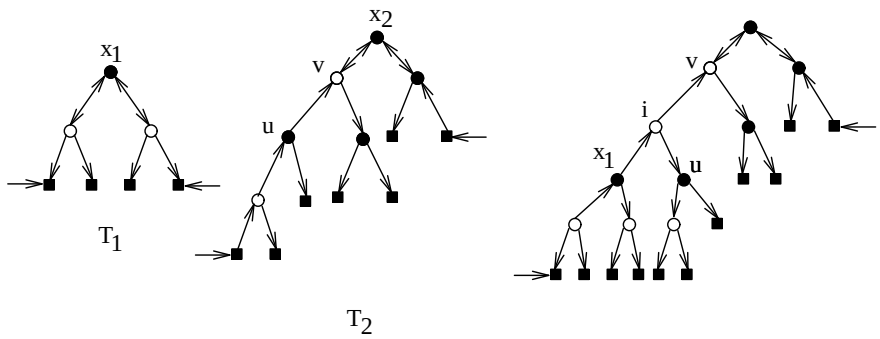


Figure 11: *Concatenating finger trees. Shown are the trees to be concatenated, T_1 and T_2, and the resulting tree T before rebalancing.*

the left path node whose non-path subtree contains x.

Let T_u be the subtree rooted at u, let v be the successor of u on the left path and let u' be the non-path child of v (see Figure 12). Delete T_u from T leaving a fragment, T', of the original tree. Make T_u a regular red-black tree by reversing the pointers along the left path and recomputing auxiliary data for nodes on this path. Use the red-black tree splitting operation to split T_u at x into T_u^1 and T_u^2. These trees are regular red-black trees; make them finger trees by reversing the pointers on the left and right paths and again recomputing the auxiliary data of these nodes. The desired tree T_1 is T_u^1.

To form T_2 we need to combine the nodes of T_u^2 and T'. Make T' a valid finger tree as follows. Reverse the pointers along the path from u' to its leftmost descendant, which is now pointed to by a left finger. Replace v by u' and its subtree. If v was black this causes a violation of the black constraint (u' is short). If so, rebalance from u' as for a deletion, updating auxiliary data as required. The auxiliary data along the path from the left finger to u' is now invalid (since the pointers have been reversed) and must be recomputed. Now use the finger tree concatenation operation to concatenate T' and T_u^2. The result is T_2.

Note that u, u', and v all have heights proportional to $O(1+\log d)$ which is also $O(1+\log(\min\{d, n-d\}))$. The work involved in finding u and deleting and creating fingers in the various trees requires time proportional to the height of u. Splitting T_u requires $O(1+\log d)$ time. Rebalancing after deleting

backwards and recomputing the auxiliary data for each node as it is accessed.

Updating the auxiliary data requires time proportional to the length of the access path, which is $O(1 + \log(\min\{d, n-d\}))$ if x is the dth node in an n-node tree. Constant time is required for the restructuring in the worst case and constant amortized time for the rebalancing. This implies that insertion at or deletion of the dth node requires $O(1 + \log(\min\{d, n-d\}))$ amortized time.

Concatenating is a little more complicated than in the regular case. Let T_1 and T_2 be two finger trees with roots x_1 and x_2, and ranks r_1 and r_2. Assume that $r_1 \leq r_2$ and that T_1 is to be concatenated to the left of T_2. The other cases are analogous.

If $r_1 = r_2$, walk up the left paths of T_1 and T_2 until reaching the roots. Create new node i having x_1 and x_2 as children and make x_1 and x_2 point to i. Make i red if both x_1 and x_2 are black and red otherwise.

If $r_1 < r_2$, then the procedure is as follows. Delete the pointers from x_1 to its children. Walk up the left path of T_1 to the root, x_1. If x_1 is red, make it black. Let r_1' denote the resulting rank. Walk up from the left finger of T_2 until reaching a black node u such that $rank(u) = r_1{}'$. Let v be the left child of u (the successor on the left path). Create new node i. Make v and u the left and right children of i, respectively and make i the left child of x_1 (see Figure 11). Make i red and rebalance using the transformations given in Figure 2 if this causes a violation of the red constraint with v (i.e., if v is red).

The tree formed is now a valid red-black tree but is not a proper finger tree since the pointers on the right path of T_1 and on the path from the old left finger of T_2 to u (or x_2 if the ranks are equal) point up instead of down. To fix this, traverse both these paths reversing the pointers and recomputing auxiliary data values. The left finger for the resulting tree is the left finger of T_1 and the right finger is the right finger of T_2. Finally, recompute the auxiliary data for nodes on the path from the left finger to i.

The time required for finding u and updating pointers and data is $O(r_1)$, which is $O(1 + \log d)$ if the tree had d external nodes, and the time required for rebalancing is $O(1)$ in the amortized sense.

Splitting is more complicated still. Let T be a finger tree and let x be the dth external node of T. Splitting at x yields finger trees T_1 containing the external nodes preceding x and T_2 containing the nodes succeeding x. We assume that x is in the left subtree; the other case is symmetric. Let u be

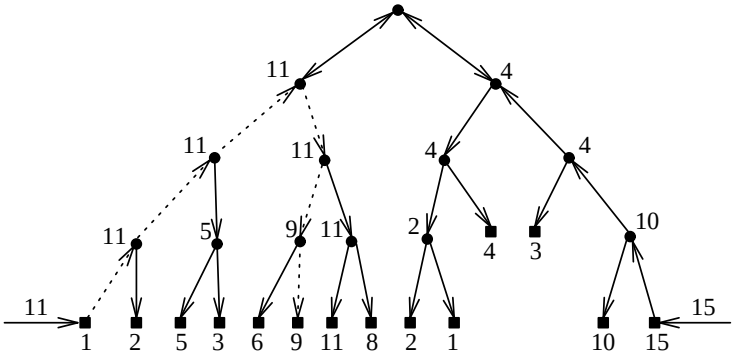
including x is at least k is similar. Such a node exists if and only if the sum of the *totkey* values of the left and right fingers is at least k. As in Section 2.2, the search is guided by *maxcumkey* values, where *maxcumkey*(u) is the maximum *cumkey* value of an external node in u's subtree. This may not be the same as the maximum *cumkey* value of an external node reachable from u.

If the *totkey* value of the left finger is at least k, set *maxcumkey* $= 0$ and walk up the left path, maintaining *maxcumkey*, until *maxcumkey* $\geq k$. To update this value when going from node u to its successor on the left path, add *totkey*(v) where v is the right child of u. Next search down in the non-path subtree as in a regular red-black tree. If the item is not in the left subtree, search from the right finger as follows. Set *maxcumkey* to be the sum of the *totkey* values of the left and right fingers. Walk up the right path maintaining *maxcumkey* until reaching the first node u whose successor has *maxcumkey* value less than k. To update *maxcumkey* when going from a node v to its successor, subtract the *totkey* value of v's left child. Then search for the leftmost node with *cumkey* at least k in the left subtree of u. See Figure 10(b) for an example.

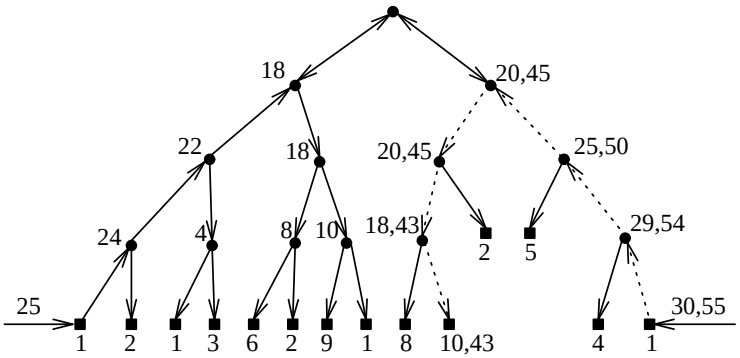
In both cases, accessing a node requires time proportional to the length of the access path, which is $O(1 + \log \min\{d, n-d\}))$ for the dth external node in an n-node tree.

As described in [15], it is also possible to access a node by searching simultaneously from both fingers. Here a node is always accessed via the access path (i.e., the path from the "nearer" finger), even though the other path may be shorter by as much as a constant factor. Which method one should use depends on the application. If after accessing an external node x we wish to insert after x or to delete x, we need to know the access path in order to update the auxiliary data. Similarly, splitting at x requires knowledge of the access path, as we will see. For our purposes it is simpler to search as described above.

Performing insertions and deletions in finger trees is almost the same as in regular red-black trees. Consider inserting after or before an external node x or deleting x. It is again necessary to know the access path to x. If x and its parent are both non-path nodes, then the restructuring is the same as before. Otherwise the reversed pointers have to be taken into account. Both of these operations potentially invalidate the data of all nodes on the access path to x. Updating is done as in regular trees by traversing the access path



(a) k = 8



(b) k = 40

Figure 10: *Searching in finger trees. A tree with* maxkey *values is shown in Figure (a). The dashed edges comprise the access path for a search on* maxkey *with* $k = 8$. *The same tree is shown in Figure (b) with the* totkey *value associated with each node. Also shown are the* maxcumkey *values maintained along the access path (which is dashed) during the search for the leftmost node with* cumkey $\geq k$, *for* $k = 40$.

Lemma 2 *The length of the access path to the dth external node is $O(1 + \log(\min\{d, n-d\}))$.*

Proof: Let x be the dth external node in an n-node tree. Assume x is in the left subtree; the other case is symmetric. We will show that the length of the left access path is $O(1 + \log d)$ and that this is also $O(1 + \log(n-d))$. The case when x is in the right subtree is analogous.

Let u be the left path node whose non-path subtree contains x. The left access path consists of the path nodes from the left finger up to u and the path from u down to x. Its length is $O(rank(u))$. If $d = 1$ then the path has length one. Otherwise let v be the predecessor of u on the left path (ie, u is the left child of v). Since v has fewer than d external descendants, we know that v has rank at most $\log d + 1$. This implies that the rank of u is at most $\log d + 2$ and thus that the length of the left access path is $O(1 + \log d)$.

Given that x is in the left subtree we can also show that this value is $O(1+\log(n-d))$. Let z be the right child of the root. The number of external descendants of z is at most $n-d$. Thus the rank of z is at most $\log(n-d)+1$. The rank of z is at least $r-1$, where r is the rank of the root of the tree. This is at least $(\log n - 1)/2$, which implies that $\log n \leq 2\log(n-d) + 3$. Since $d \leq n$, this implies that that $1 + \log d$ is $O(1 + \log(n-d))$, as claimed. ∎

Both types of queries described in the previous section can be performed on finger trees as well. Both searches proceed by finding the node on the left or right path whose non-path subtree contains the item and then searching down in the indicated non-path subtree (which is a regular red-black tree) for the desired external node.

First we describe how to find the leftmost node with key at least k. Detecting whether such a key exists can be done in constant time; a tree contains an item with $key1 \geq k$ if and only if one of the fingers has $maxkey$ value at least k. If the maxkey value of the left finger is at least k (the item is reachable from the left finger) then walk up the left path until reaching the first node u with non-path child v such that $maxkey(v) \geq k$. Then search down in v's subtree for the leftmost external node with key at least k. Otherwise, walk up the right path until reaching the last node u such that $maxkey(u) \geq k$. Then search down in u's non-path subtree for the leftmost external node with key at least k. This is illustrated in Figure 10 (a).

Finding the leftmost external node x such that the sum of keys up to and

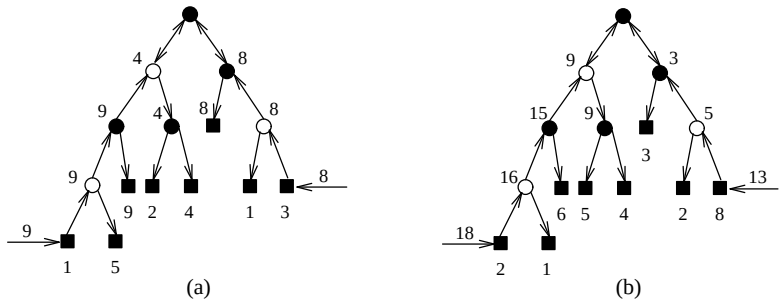


Figure 9: *Figure (a) illustrates a finger tree shown with* key1 *and* maxkey *values. In Figure (b) the same tree with* key2 *and* totkey *values shown.*

the left or right path is the subtree rooted at the non-path child of u.

A list of items is again represented by associating the items with the external nodes, in order. We assume that each item has two associated real values, *key1* and *key2*. For each non-root node u in the tree let $maxkey(u)$ be $\max\{key1(x)|x$ is an external node reachable from $u\}$, and let $totkey(u)$ be $\sum\{key2(x)|x$ is an external node reachable from $u\}$. Note that these values are defined in terms of the reversed pointers along the left and right paths. In order to perform the search operations described previously we store these values with each node except for the root and with each finger. Figure 9 contains an example illustrating these values.

A node is accessed by traversing the path from one of the fingers. There are two such paths; one passes through the root and one does not. A node is always accessed by the path which does not pass through the root, which is called the *access path*. Although this path may be longer than the other one, it will be longer by at most a factor of 2. The finger from which the access path starts is called the *nearer* finger. The advantage of finger search trees over red-black trees is that the time required to access a node is proportional to the distance from a finger rather than on the distance to the root; this distance is $O(1 + \log(\min\{d, n-d\}))$ if the node is the dth and the tree has n external nodes altogether, as shown below.

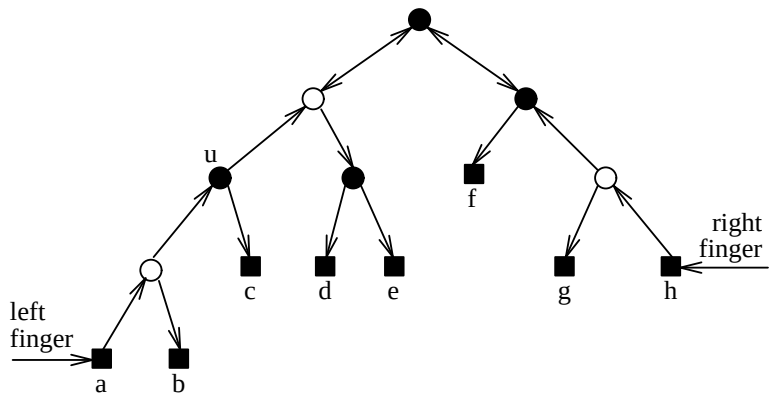


Figure 8: *A finger tree.*

In a regular red-black tree, the *left path* is the path from the root to the leftmost external node and the *right path* is the path from the root to the rightmost external node. To convert a red-black tree into a finger tree we reverse the pointers along the left and right paths and add pointers from the root to its two children.

For non-root nodes on the left path in the finger tree the left pointer now points to the parent in the red-black tree. For non-root nodes on the right path the right pointer points to the parent. The pointers from the root are unchanged. The tree is accessed via pointers to the leftmost and rightmost external nodes. These are called the *left finger* and *right finger*, respectively. See Figure 8 for an example.

For such a finger tree with n external nodes, inserting, deleting, or splitting at the dth node requires $O(1+\log(\min\{d, n-d\}))$ time, as does accessing the dth node. Concatenating requires time proportional to the rank of the smaller tree. All these time bounds are in the amortized sense.

The nodes *reachable* from a node u in the finger tree are those nodes connected to u by a path of pointers that does not pass through the root. It will later be clear why we exclude paths which pass through the root. The *subtree* of u is the set of nodes which would be reachable if the pointers were not reversed. For example, in Figure 8, c, d, and e are the external nodes reachable from u and a, b, and c are the external nodes contained in the subtree of u. A non-root node is called a *path* node if it is on the left or right path and a *non-path* node otherwise. The *non-path subtree* of a node u on

x to the root as long as the old and new *maxkey* values for x are known. Let *maxkey* and $\Delta maxkey$ denote the old values and $maxkey'$ and $\Delta maxkey'$ the new ones. At each node x we compute $\Delta maxkey'(x)$ and $\Delta maxkey'(sib(x))$, where $sib(x)$ denotes the sibling of x, as follows:

$$
\begin{aligned}
&maxkey(p(x)) = maxkey(x) - \Delta maxkey(x)\\
&maxkey(sib(x)) = maxkey(p(x)) + \Delta maxkey(sib(x))\\
&maxkey'(p(x)) = \max\{maxkey'(x), maxkey(sib(x))\}\\
&\Delta maxkey'(x) = maxkey'(x) - maxkey'(p(x))\\
&\Delta maxkey'(sib(x)) = maxkey(sib(x)) - maxkey'(p(x)).
\end{aligned}
$$

Now it is possible to process $p(x)$. This procedure is used for updating $\Delta maxkey$ data after an insertion, deletion, or concatenation. The time required is the length of the path traversed, and is proportional to the worst-case time required for the operation itself.

Auxiliary values may also be invalidated by rotations. To recompute these values, we maintain the *maxkey* value of the current node while rebalancing. After a rotation, this value is used to compute first the old *maxkey* values from which we compute the new *maxkey* values and new $\Delta maxkey$ values, which are then stored. This requires constant time per rotation.

To summarize, we have described a data structure that is capable of supporting any sequence of insertions, deletions, concatenations, splittings, and accesses of the form described above. Except for concatenation, all operations require $O(\log n)$ time on an n-node tree. Concatenation of two trees of ranks r_1 and r_2 requires $O(\max\{|r_1 - r_2|, 1\})$ time. In $O(\log n)$ time we can access the leftmost external node with *key* at least k or the leftmost with *cumkey* at least k and can also add a constant to the values of all external nodes in the tree, or to the values of all external nodes between two chosen ones.

2 Finger Trees

In this section we describe the finger tree data structure presented by Tarjan and Van Wyk in [15]. We will refer to these trees simply as finger trees. Also presented is a cleaner variant of this data structure which has slightly worse access time.

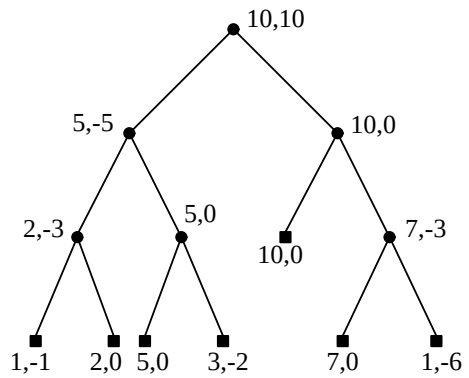


Figure 7: *Data required for performing an* add value *operation. Shown with each node u is* $\mathrm{maxkey}(u), \Delta\mathrm{maxkey}(u)$. *For each external node u,* $\mathrm{maxkey}(u) = \mathrm{key}(u)$.

node x. This requires time proportional to the depth of the tree (assuming that the path from the root to x is given) and is accomplished by traversing this path, at each node u adding k to $\Delta maxkey(sib(u))$ if $sib(u)$ is a left child. An analogous procedure can be used to add value to all items contained in external nodes to the right of x. Combining the two procedures allows us also to add a value to the keys of all items between two specified external nodes.

If $\Delta maxkey$ values are stored instead of $maxkey$ values, then extra work is required in the splitting operation. In order to split at node x we need to compute $maxkey(x)$ and to maintain the $maxkey$ value of the current node as the path up from x is traversed. When a subtree of this path is cut off, the $\Delta maxkey$ value of the root of the subtree is replaced by the $maxkey$ value of the root. Taken by itself, each subtree hanging off the path now has correct $\Delta maxkey$ values and the splitting algorithm can proceed as described above. This extra computation adds only a constant factor to the running time.

Maintaining $\Delta maxkey$ values is harder than maintaining $maxkey$ or $totkey$. Again values may be invalidated by structural changes and by rotations. First consider the changes caused by modifying the tree. Modifying the contents of the subtree rooted at node x may invalidate the $maxkey$ values of the nodes on the path from x to the root and the $\Delta maxkey$ values of these nodes and their siblings. These values can be recomputed by traversing the path from

in an n node red-black tree.

It is also necessary to describe how to compute and maintain the auxiliary data. Computing the auxiliary data initially requires $O(n)$ time for an n-node tree since both *maxkey* and *totkey* have recursive definitions and can be computed bottom-up. Note that tree restructuring such can invalidate some auxiliary data values. These values can be updated quickly as long as the data is defined recursively; that is, the value associated with a node can be computed from the values associated with its children.

Insertion or deletion at an external node x may invalidate the auxiliary data of the ancestors of x only. Concatenating may invalidate the data of all nodes along a path from the new node (i in Figure 4) to the root. This data can be recomputed by processing the affected path bottom-up. This will take at most $O(\log n)$ time after an insertion into or a deletion from an n-node tree, making the amortized (as well as the worst-case) time $O(\log n)$. Updating data after a concatenation adds only a constant factor to the running time.

The auxiliary data is also affected by rotations occurring during rebalancing. A rotation affects the data values of only a constant number of nodes. It is easy to see that this data can be recomputed (bottom-up) in constant time per rotation.

Another useful operation that can be executed is *add value*. Given a value k and a node u we can add the value k to the keys of all items in u's subtree in constant time. This operation is compatible only with the search on *maxkey* values. That is, we cannot perform *add value* operations and also search on cumulative keys in logarithmic time. We will assume from now on that each item has two keys, *key1* and *key2*. The *maxkey* values are defined relative to *key1* values and *totkey* and *cumkey* relative to *key2*. Add value operations affect *key1* values only.

In order to support *add value* we store the *maxkey* data in difference form. With each non-root node x we define $\Delta maxkey(u) = maxkey(u) - maxkey(p(u))$. If u is the root, $\Delta maxkey(u) = maxkey(u)$. This is illustrated in Figure 7. Note that for a node u, $maxkey(u) = \sum\{\Delta maxkey(v) | v$ is an ancestor of $u\}$. If $\Delta maxkey$ values are stored and we are given the *maxkey* value of a node u then we can compute the *maxkey* values for the parent or children of u in constant time.

In order to perform *add value*(k, u) to add the value k to the keys of all external descendants of u, simply add k to $\Delta maxkey(u)$. It is also possible to add a value to the keys of all external nodes to the left of a given external

here. First we describe, for given value k, how to find the leftmost item with key at least k. This does not assume any ordering among the keys.

To perform this type of search store with each internal node u the value $maxkey(u)$ which is the maximum key of an external descendant of u. The desired item is found by searching down the path from the root. If u is the current node on the path and v is its left child, then go left if $maxkey(v) \geq k$ and go right otherwise. The external node reached is the leftmost external node with key at least k.

It is easy to modify this in order to find the rightmost item with key at least k. By storing *minkey* values instead of *maxkey* we can find the leftmost (rightmost) item with key at most k.

We can also search on *cumulative* key values, if the keys are all non-negative. That is, given value k, find the leftmost item x such that the sum of the keys of items up to and including x is at least k. For example, if all keys are 1, this would find the kth item.

For external node x let $cumkey(x)$ denote the sum of keys of external nodes up to and including x. One way to perform the above search would be to consider the *cumkey* values as the keys and to perform the search given above. In order to achieve this we would have to store with each node u the value $maxcumkey(u)$, which is the maximum *cumkey* value of an external descendant of u. Note that this equivalent to the *maxcumkey* value of the rightmost external descendant of u. If data is stored in this manner, then operations modify the tree (ie, insertions and deletions) may potentially invalidate many *maxcumkey* values. For example, deletion of external node x invalidates the *cumkey* data of all external nodes succeeding x which is likely to invalidate the *maxcumkey* data of all ancestors of these nodes.

Instead we store with each internal node u the value $totkey(u)$ which is the sum of the keys of all the external descendants of u. From these values we can compute *maxcumkey* values as they are needed. Again the search traverses the path from the root to the desired node. Let u be the current location of the search and let v be the left child of u and w the right child. Initially u is the root and $maxcumkey(u) = totkey(u)$. At a general step compute $maxcumkey(v) = maxcumkey(u) - totkey(w)$. If this value is greater than or equal to k go left; otherwise go right and set $maxcumkey(w) = maxcumkey(u)$.

In accessing an item both of the search operations described above traverse the path from the root to the desired item. This requires $O(\log n)$ time

both cases, both children of x_{i-1} are black. The children of x_i and of x_{i-1} all have rank $k-1$. Since the invariant holds for x_{i-2} we know that the ranks of L_{i-2} and R_{i-2} are at most $k-1$ and that $cost_{i-2} \leq 4(k-1)+r(L_{i-2})+r(R_{i-2})$.

Case 2a : x_{i-2} is a left child (see Figure 6(b)).
On the $(i-1)^{st}$ step, R_{i-1} is formed by concatenating R_{i-2} and T_{i-1}. The rank of R_{i-1} is at most k since the ranks of the two constituent trees are at most $k-1$. The rank of R_i is also at most k, since $R_i = R_{i-1}$. On the ith step, L_i is formed by concatenating T_i and L_{i-1}, which is the same as L_{i-2}. Again, both constituent trees have rank at most $k-1$, making the rank of L_i at most k.

The cost at stage i is at most $cost_{i-2} + 1 + \max\{|r(T_i) - r(L_{i-1})|, 1\} + 1 + \max\{|r(T_{i-1}) - r(R_{i-2})|, 1\}$. The techniques used in Case 1 above can be used here to show that $cost_i \leq 4k + r(L_i) + r(R_i)$, which completes the proof that the invariant is satisfied in this case.

Case 2b : x_{i-2} is a right child (see Figure 6(c)).
Here R_i is the same as R_{i-2} and has rank at most $k-1$ which is less than $r(x_i)$. Consider L_i. In the $(i-1)^{st}$ stage L_{i-1} is formed by concatenating T_{i-1} and L_{i-2}. The resulting tree has rank k or $k-1$. By the concatenation property given previously, the rank of L_{i-1} will be $k-1$ or k; if it is k then the root of L_{i-1} must be black. If L_{i-1} has rank $k-1$ then $r(L_i) \leq k$ since T_i has rank $k-1$. If L_{i-1} has rank k, then we are concatenating a tree with rank k and black root with a tree of rank $k-1$. The resulting tree, L_i, has rank k.

The cost at time i is

$$cost_{i-2} + 2 + \max\{|r(T_{i-1} - r(L_{i-2})|, 1\} + \max\{|r(T_i) - r(L_{i-1})|, 1\}.$$

The reasoning used in the previous cases can be used to show that this is at most $4k - 1 + r(R_i) + r(L_i) + \max\{|r(L_{i-1}) - r(T_i)|, 1\}$. Since the value of the last term is at most 1, $cost_i$ also satisfies the invariant. ∎

Searching in a list represented by a balanced tree requires logarithmic time. Recall that an n-element list is represented by a red-black tree with n external nodes, in which each item and its key is associated with an external node such that order in the list corresponds to left-to-right order among the external nodes.

There are two general types of search operations which we will describe

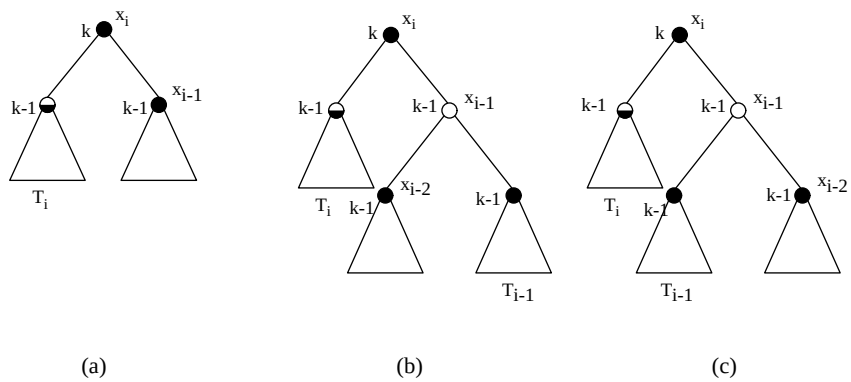


Figure 6: *The cases of the proof of Lemma 1. The half-darkened nodes may be either black or red.*

$r(R_i) \leq r(x_i)$, and $cost_i \leq 4r(x_i) + r(L_i) + r(R_i)$. When $i = l$ this implies that $cost_l$, which is the time required to execute the entire splitting process, is $O(\log n)$, since the ranks of x_l, L_l and R_l are all $O(\log n)$.

When $i = 1$, the invariant is satisfied since $r(x_1) = 1$, L_1 and R_1 are empty and have rank 0 and $cost_1 = 0$.

We will show that the invariant holds at stage i if x_i is black. Assume that the invariant holds for all black nodes x_j such that $j < i$. Assume also that x_{i-1} is a right child; the other case is symmetric. Let k denote the rank of x_i. Both children of x_i have rank $k - 1$.

Case 1 : x_{i-1} is black. (See Figure 6(a).)
By the inductive hypothesis, the invariant holds for x_{i-1}. This implies that the ranks of L_{i-1} and R_{i-1} are at most $k - 1$ and that $cost_{i-1}$ is at most $4r(x_{i-1}) + r(L_{i-1}) + r(R_{i-1})$.

The invariant holds with respect to R_i since $R_i = R_{i-1}$ which has rank at most $k-1$. Next consider L_i, which is formed by concatenating T_i and L_{i-1}. Since both these trees have rank at most $k-1$, L_i has rank at most k. Finally, consider $cost_i$, which is $cost_{i-1}+1+\max\{|r(T_i)-r(L_{i-1})|, 1\}$. Using the facts that $r(T_i) \geq r(L_{i-1})$ and that for all i the rank of L_i (R_i) is no less than the rank of L_{i-1} (R_{i-1}) gives: $cost_i \leq 4k-4+r(R_i)+1+\max\{r(T_i), r(L_{i-1})+1\}$. Substituting $r(L_i) + 1$ for the last term gives the desired result.

Case 2 : x_{i-1} is red.
There are two sub-cases depending on whether x_{i-2} is a left or a right child. In

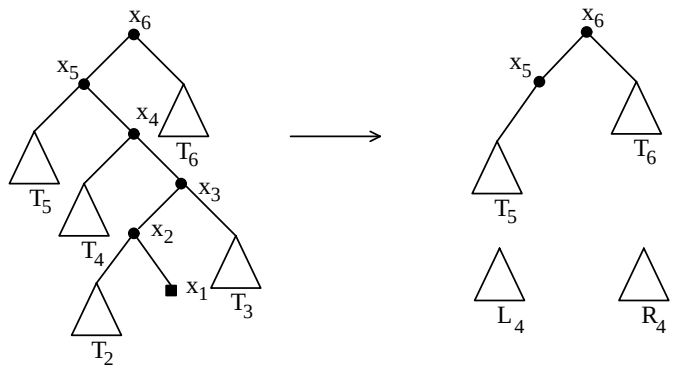


Figure 5: *Splitting a red-black tree. After four stages, L_4 is the result of concatenating T_2 and T_4 and R_4 is T_3.*

is the root. Let T_i, $2 \leq i \leq l$, be the subtree of x_i that does not contain x_{i-1}. Let L_i be the tree formed by concatenating those left subtrees of $x_2, \ldots, x_i$ that do not contain x_1 and let R_i be defined analogously for the right subtrees. This is illustrated in Figure 5.

Splitting is accomplished as follows. Initially L_1 and R_1 are empty. At time i, compute L_i and R_i. If x_{i-1} is a left child, then $L_i = L_{i-1}$ and R_i is formed by concatenating R_{i-1} and T_i. If x_{i-1} is a right child, then $R_i = R_{i-1}$ and L_i is formed by concatenating T_i and L_{i-1}. The final trees L_l and R_l are T_l and T_r.

We will show that the time required for splitting an n-node tree is $O(\log n)$. The cost is essentially a telescoping series, but the analysis is involved. Let $cost_i$ denote the time required for executing the first i stages as described above. Note that $cost_i = cost_{i-1} + 1+$ the cost of performing the ith concatenation. Recall that the worst-case time to concatenate two trees of rank r_1 and r_2 is $O(\max\{|r_1 - r_2|, 1\})$. For simplicity we will assume here that the constant is 1; this affects only the constant of the running time of the splitting algorithm.

Lemma 1 *The time required to split an n-node red-black tree at an external node is $O(\log n)$.*

Proof: We show by induction on the black nodes of the path that the following invariant is maintained: at the ith stage, if x_i is black, then $r(L_i) \leq r(x_i)$,

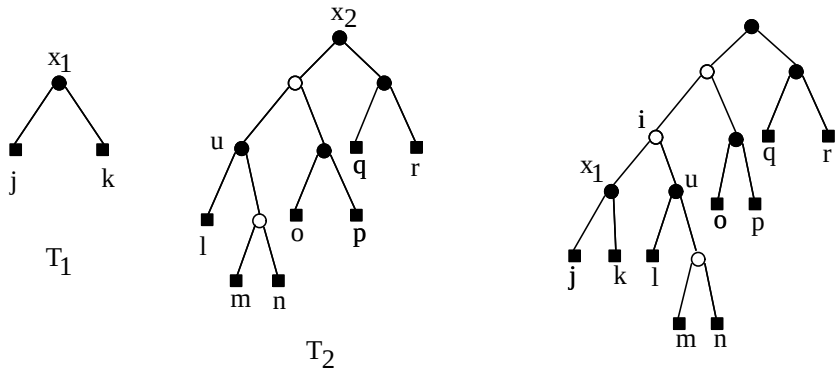


Figure 4: *Concatenating two red-black trees. Shown are the original trees, T_1 and T_2, and the result before rebalancing.*

from such a concatenation: if $r_2 = r_1$ and both roots were originally black or if $r_2 > r_1$ and x_2 was black then r is r_2; otherwise r is r_2 or $r_2 + 1$. Further, if the rank of the resulting tree is $r_2 + 1$, then its root is black.

If the trees are concatenated by creating a new root with x_1 and x_2 as children, then an analysis of the possible cases above shows that the property holds. Otherwise the resulting tree is formed by modifying the structure of T_2, leaving the root intact, and then rebalancing as for an insertion if necessary. The rank of this tree is the same as the rank of T_2 unless rebalancing causes the rank of the root to increase. Inspection of the relevant rebalancing transformations (see Figure 2) reveals that this can occur only if x_2 was originally red. This implies that for $r_2 > r_1'$, the rank of the resulting tree is r_2 if x_2 was originally black, as claimed. Note that the rank of the root can increase by at most one; in all such cases the new root is black.

The inverse of concatenating is *splitting*. Splitting a tree T at node x yields trees T_l containing the items to the left of x and T_r containing the items to the right of x, in order. In the implementation described here x is not contained in either resulting tree. It is easy to modify this implementation so that x is included either in T_l or in T_r, as desired. We assume that the path from x to the root of T is known. Splitting is accomplished by cutting off all subtrees of this path and then concatenating all the left ones in order to form T_l and then all right ones to form T_r.

More formally, let the nodes on the path be $x_1, \ldots x_l$ where $x_1 = x$ and x_l

this, we define the *potential* of a red-black tree to be the number of black nodes with two black children plus twice the number of black nodes with two red children. We define the *actual time* of an insertion or deletion to be one plus the number of transformations applied and the *amortized* time to be the actual time plus the net increase in potential caused by the tree restructuring. Each application of the (non-terminal) transformations of figures 2(a) or 3(a) reduces the potential by at least one. An application of any of the other transformations either decreases the potential or increases it by at most two. This implies that the amortized time of an insertion or deletion is $O(1)$, since there are at most a constant number of applications of the latter type of transformation. If we begin with an empty tree and perform a sequence of k external insertions and deletions, the total time for all the tree restructuring is $O(k)$, because the amortized time is $O(k)$, the actual time is the amortized time minus the net increase in potential over the sequence, and the net increase in potential is nonnegative (since the initial potential is zero and the final potential is nonnegative.)

On a list represented by red-black trees we can also perform the more complicated operations *concatenate* and *split* in logarithmic time.

Concatenate is the simpler operation. Let T_1 and T_2 be two red-black trees with ranks r_1 and r_2 respectively. Let x_1 be the root of T_1 and x_2 the root of T_2. We will assume that $r_1 \leq r_2$ and that we wish to concatenate T_1 on the left of T_2; the other cases are analogous.

If $r_1 = r_2$, create new root i, having x_1 as left child and x_2 as right child. If both x_1 and x_2 are black, make i red and otherwise make it black. Note that although the balance constraints can be satisfied by making the new root black in either case, this would unnecessarily increase the rank of the resulting tree, which would adversely affect the splitting operation described below.

Otherwise $r_2 > r_1$. If x_1 is red, make it black. Let $r_1{}'$ denote the current rank of x_1; $r_1{}'$ is r_1 if x_1 was black to start with and is $r_1 + 1$ otherwise. Walk down the left path of T_2 until reaching first black node u having rank equal to $r_1{}'$. Create new node i having x_1 as left child and u as right child. Replace u by i and make i red. If this causes a violation to the red constraint (if the parent of i is red), then rebalance as for an insertion. Notice that both children of i are black. This procedure (illustrated in Figure 4) requires $O(\max\{|r_1 - r_2|, 1\})$ time in the worst-case.

We will need the following property about the rank, r, of the tree resulting

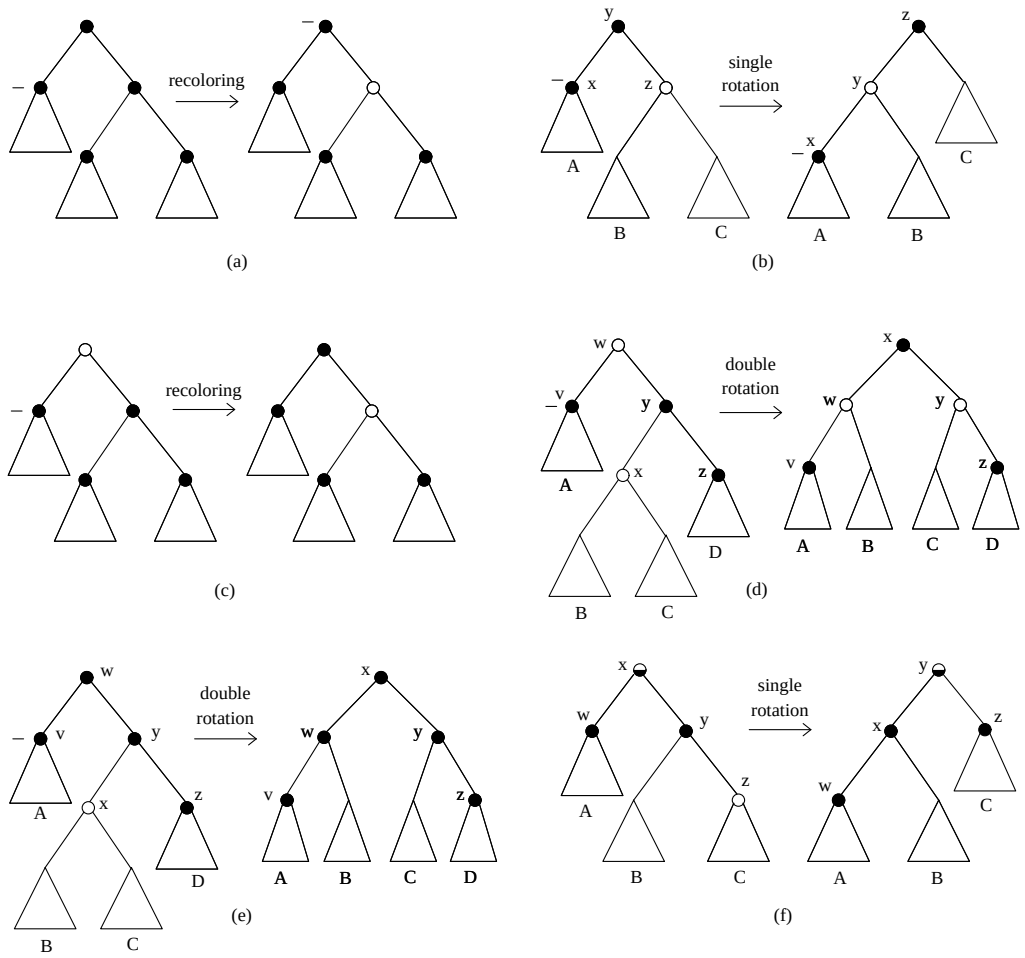


Figure 3: *Rebalancing transformations for deletions. The half-darkened nodes in (f) are nodes which can be either red or black, but they must both be the same color. A "-" by a node indicates that it is short.*

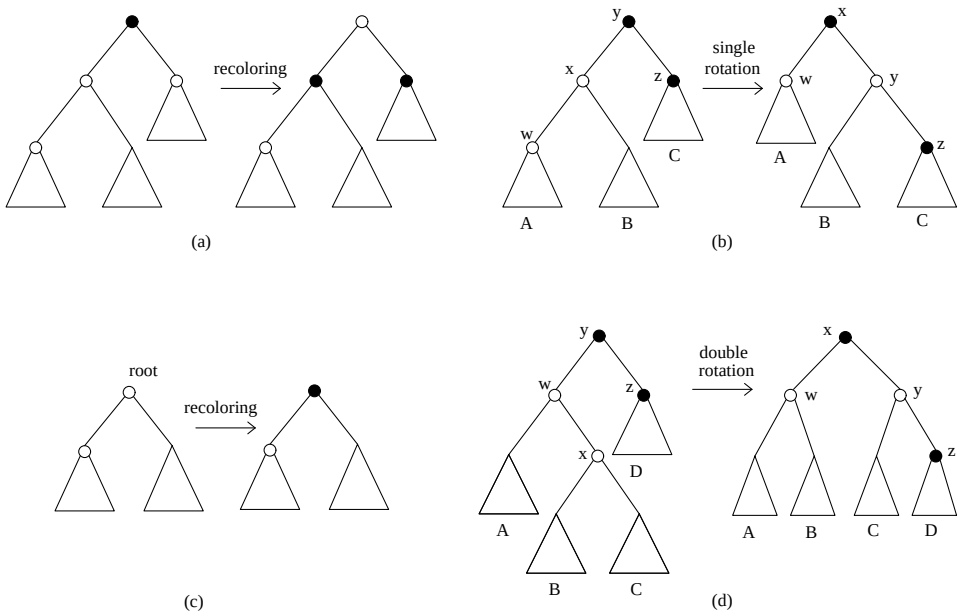


Figure 2: *Rebalancing transformations for insertions. There are symmetric cases not shown.*

if necessary applying one of the transformations in Figures 2(b),(c), or (d). (Transformations (b) and (d) change the tree structure as well as the node colors; transformation (b) does one rotation; transformation (d), two.) Note that there are symmetric cases which are not shown.

Deleting an item is similar but slightly more complicated. To delete an external node x with sibling y, replace $p(x)$ by y. If the deleted internal node was black, this violates the black constraint. That is, y is *short*, i.e., paths down from it contain one too few black nodes. The violation is corrected by walking up the tree path from the violation, repeatedly applying the transformation of Figure 3(a) until it no longer applies, followed if necessary by one application of Figure 3(b) and then possibly one application of Figure 3(c), (d), (e), or (f). Altogether at most three rotations are done.

The worst-case time needed to insert or delete an external node in a red-black tree is $O(\log n)$, but the amortized time[1] is only $O(1)$. To prove

[1]By *amortized time* we mean roughly the time of an operation averaged over a worst-case sequence of operations. See [14].

The *rank* of a node u is defined to be the number of black nodes along any path from u to an external node in its subtree. See Figure 1 for an example of a red-black tree with rank values shown. The *height* of node u is the maximum number of nodes along the longest path from u to one of its external descendants. A node with rank r has height between r and $2r$ and has between 2^{r-1} and 2^{2r-1} external descendants. Thus a red-black tree with n external nodes has depth $O(\log n)$. We will refer to the rank of the root of a tree as the rank of the tree.

For a node x we use $p(x)$ to denote the parent of x, $left(x)$ and $right(x)$ to denote the left and right children of x, and $sib(x)$ to denote the sibling of x.

There are two ways to represent a list using red-black trees. The items can be stored either in the internal nodes, so that order in the list corresponds to symmetric order in the tree, or in the external nodes, so that order in the list corresponds to left-to-right order among the external nodes. We will use the latter representation. We will also assume that the keys associated with list items are real numbers. Each red-black tree node will have pointers to its left and right children. The rank of each tree is stored with its root.

Given a list represented as a red-black tree T we may wish to access a particular item of the list, for example, the item with key k or perhaps the kth item in the list. Performing any kind of access requires storing auxiliary data in the internal nodes. For example, in order to find the node with key k we store with each internal node u the maximum key associated with an external node in u's subtree. We will describe two broad types of search operations and how they are executed. Both types of search are used in the algorithm presented in Chapter 4. First we describe how the list operations *insert*, *delete*, *concatenate*, and *split* are implemented in a red-black tree.

Inserting into and deleting from red-black trees is accomplished by the addition and deletion of external nodes. Such updates require rebalancing the tree, which is done by recoloring nodes and performing *rotations*, which locally change the tree structure.

To insert a new item after a specified external node x, replace x by a new internal node having x and a new external node containing the new item as children. The new internal node is colored red. If its parent is also red, this violates the red constraint (iii). Such a violation is corrected by walking up the tree path from the violation, repeatedly applying the recoloring transformation of Figure 2(a) until it no longer applies, and then

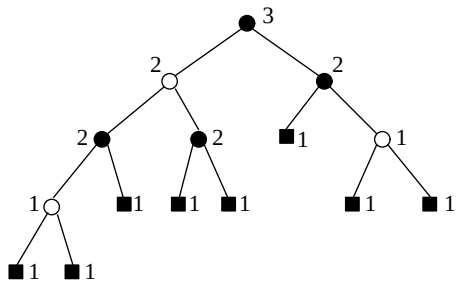


Figure 1: *A red-black tree. The darkened nodes are black nodes. The external nodes are denoted by squares. Shown with each node is its rank.*

Wyk give another, simpler, implementation of finger trees. They describe a finger data structure which is a modification of red-black trees, but other forms of balanced trees could be used as a basis for the structure.

The two problems presented in Chapters 3 and 4 rely on the use of red-black and finger trees respectively. In this chapter we give a fairly complete overview of red-black trees, of the finger trees introduced by Tarjan and Van Wyk, and of a variant of these which we use in Chapter 4. The material here is intended to be comprehensive and useful as an introduction to these two types of data structures.

1 Red-Black Trees

A *red-black tree* is a full binary tree in which each node is assigned a color, either *red* or *black*. The leaves are called *external* nodes and the non-leaves are called *internal*. The node colors satisfy the following constraints:

(i) All external nodes are black.

(ii) All paths from the root to an external node contain the same number of black nodes.

(iii) Any red node with a parent has a black parent.

March 31, 1992

An Overview of Red-Black and Finger Trees

Chapter 2 of "Some Fast Algorithms on Graphs and Trees", Technical Report CS-TR-296-90, Princeton University, 1990 (dissertation).

Heather D. Booth,

Department of Computer Science
University of Tennessee
Knoxville, TN 37996–1301
USA

Balanced trees are a well-known solution to the problem of maintaining an ordered list of items so as to be able to quickly perform the operations *insert*, *delete*, and *access*. These operations can be executed in $O(\log n)$ time on a list of n elements if the list is represented by a balanced tree. Several types of balanced tree structures have been studied in the literature, including AVL-trees [2, 1], B-trees [3, 4], and red-black trees [7]. With both red-black and B-trees, the more complicated list operations *split* and *concatenate*, can also be performed in logarithmic time.

A more recently developed data structure is the *finger tree*, which performs better than balanced trees when the tree operations exhibit some locality of reference. In finger trees we access data through pointers to a fixed number of leaves ("fingers") rather than through the root. This makes the access time dependent on the distance to the fingers rather than on the number of items in the list.

Finger trees were introduced by Guibas, McCreight, Plass, and Roberts [6], as a variant of B-trees. Huddleston and Mehlhorn refined this work in [8], still using B-trees. Tsakalidis has presented a finger tree data structure based on AVL trees [16]. Kosaraju [9] presents a more general structure which also has similar properties. In the appendix of [15], Tarjan and Van