A Dichromatic Framework For Balanced Trees

Leo J. Guibas
Xerox Palo Alto Research Center,
Palo Alto, California, and
Carnegie-Mellon University

and

Robert Sedgewick*
Program in Computer Science
Brown University
Providence, R. I.

ABSTRACT

In this paper we present a uniform framework for the implementation and study of balanced tree algorithms. We show how to imbed in this framework the best known balanced tree techniques and then use the framework to develop new algorithms which perform the update and rebalancing in one pass, on the way down towards a leaf. We conclude with a study of performance issues and concurrent updating.

0. Introduction

Balanced trees are among the oldest and most widely used data structures for searching. These trees allow a wide variety of operations, such as search, insertion, deletion, merging, and splitting to be performed in time $O(\lg N)$, where N denotes the size of the tree [AHU], [Kn]. (Throughout the paper lg will denote log to the base 2.) A number of different types of balanced trees have been proposed, and while the related algorithms are often conceptually simple, they have proven cumbersome to implement in practice. Also, the variety of such trees and the lack of good analytic results describing their performance has made it difficult to decide which is best in a given situation.

In this paper we present a uniform framework for the implementation and study of balanced tree algorithms. The framework deals exclusively with binary trees which contain two kinds of nodes: internal and external. Each internal node contains a key (chosen from a linear order) and has two links to other nodes (internal or external). External nodes contain no keys and have null links. If such a tree is traversed in symmetric order [Kn] then the internal nodes will be visited in increasing order of their keys. A second defining feature of the framework is that it allows one bit per node, called the *color* of the node, to store balance information. We will use *red* and *black* as the two colors. In section 1 we further elaborate upon this dichromatic framework and show how to imbed in it the best known balanced tree algorithms. In doing so, we will discover suprising new and efficient implementations of these techniques.

In section 2 we use the framework to develop new balanced tree algorithms which perform the update and rebalancing in one pass, on the way down towards a leaf. As we will see, this has a number of significant advantages over the older methods. We shall examine a number of variations on a common theme and exhibit full implementations which are notable for their brevity. One implementation is examined carefully, and some properties about its behavior are proved.

In both sections 1 and 2 particular attention is paid to practical implementation issues, and complete implementations are given for all of the important algorithms. This is significant because one measure under which balanced tree algorithms can differ greatly is the amount of code required to actually implement them.

Section 3 deals with the analysis of the algorithms. New results are given for the worst case performance, and a technique for studying the average case is described. While no balanced tree algorithm has yet satisfactorily submitted to an average case analysis, empirical results are given which show that the various algorithms differ only slightly in performance. One implication of this is that the top-down algorithms of section 2 can be recommended for most applications because of their simplicity.

Finally, in section 4, we discuss some other properties of the trees. In particular, a one-pass top down deletion algorithm is presented. In addition, we consider how to decouple the balancing from the updating operations and we explore parallel updating.

1. The Uniform Framework

In this section we present a uniform framework for describing balanced trees. We show how to embed in this framework the most widely used balanced tree schemes, namely B-trees [BaMc], and AVL trees [AVL]. In fact, this embedding will give us interesting and novel implementations of these two schemes.

We consider rebalancing transformations which maintain the symmetric order of the keys and which are local to a small portion of the tree for obvious efficiency reasons. These transformations will change the structure of the tree in the same way as the single and double rotations used by AVL trees [Kn]. The difference between the various algorithms we discuss arises in the decision of *when* to rotate, and in the manipulation of the node colors.

For our first example, let us consider the implementation of 2-3 trees, the simplest type of B-tree. Recall that a 2-3 tree consists of 2-nodes, which have one key and two sons, 3-nodes, which have two

* This work was done in part while this author was a Visiting Scientist at the Xerox Palo Alto Research Center and in part under support from the National Science Foundation, grant no. MCS75-23738.

CH1397-9/78/0000-0008$00.75

keys and three sons, and external nodes which have no keys and no sons. Inserting a new key into a 2-3 tree involves first doing an unsuccessful search which terminates at an external node, then inserting the new key into the father of that node. If this is a 3-node, it must be split into two 2-nodes, and the overflow key inserted into its father, and so on. The "balance" in the tree comes from the fact that all paths starting at an internal node and ending at an external node have the same length.

A natural way to represent a 2-3 tree as a binary tree is to make the explicit links of the tree *black* and to "binarize" the 3-nodes by connecting their two keys with a *red* link, as shown in Fig. 1. (It is more convenient to draw colored links than colored nodes, so we establish the convention that the color of a node is equivalent to the color of the link which points to it. In our figures heavy lines will indicate red links.) Note that this correponds to keeping track of the total number of key comparisons involved when traversing such a node.

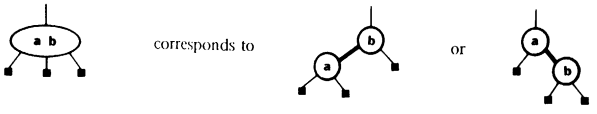

Figure 1. The binarization of a 3-node

To describe the dynamic aspects of the implementation, it is sufficient to consider insertion into 2- and 3-nodes. Insertion into a 2-node gives a 3-node in the obvious way, as shown in Fig. 2. Insertion into a 3-node requires more work, as diagrammed in Fig. 3. (This diagram assumes that the 3-node "leans" to the right: the other case is obviously symmetric.) The three cases here are quite different. With respect to the structure of the tree, the first case is a simple insertion, the second is a so-called "single rotation", and the third is a "double rotation". In all three cases, the color of the top node is eventually changed to red, with its sons and grandsons all black. This corresponds to splitting the 3-node into two 2-nodes; the red link is the message that an insertion needs to be performed into the node above, using the same method.

Fig. 4. shows the sequence of 2-3 trees that results when this method is used to insert the keys 1, 9, 2, 8, 3, 7, 4, 6, and 5 in this order into an initially empty tree. (This sequence is a well-known example which produces a completely unbalanced "zigzag" tree if no balancing at all is done.) Note that inserting the 2 causes a double rotation, the 3 causes a single rotation, the 4 causes two double rotations, and the 5 causes a single rotation.

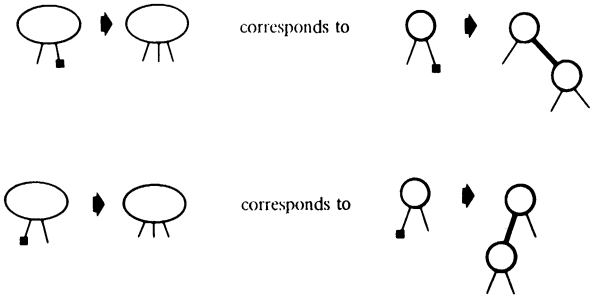

Figure 2. Insertion into a 2-node

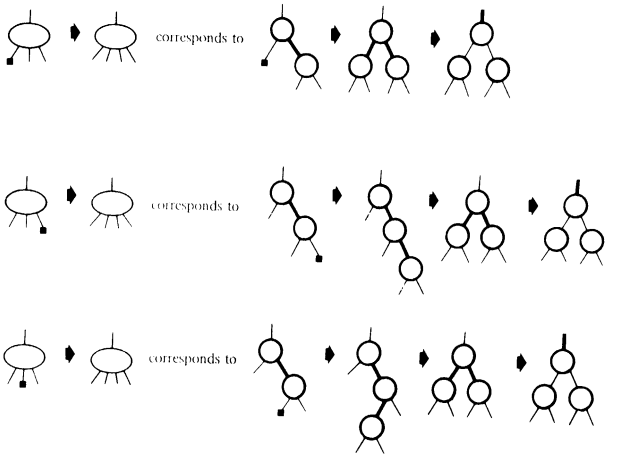

Figure 3. Insertion into a 3-node

An implementation of search and insertion for 2-3 trees within this framework is given in Program 1. For convenience, this implementation uses the normal technique of including two artificial nodes: a "head" node (h) whose right link points to the root of the tree, and a special node (z) which represents all external nodes. The key being sought is first stored in z, so that the search always terminates successfully. If the search terminates at z, then the search was really unsuccessful, and an insertion is performed. A stack is used to keep track of the path traversed on the way down the tree. (The stack could be eliminated by remembering the last 2-node encountered, but we shall see better methods of doing so later.) The single and double rotations are handled with a single procedure called "balance", which is given in Program 2. This procedure takes as input links to four consecutive nodes down the path and changes the structure of the tree to locally balance the bottom three of these nodes, as shown in Fig. 5. (Only the two cases with f as a right link are diagrammed: the cases with f as a left link are symmetric.) Also, the colors of the nodes pointed to by the g and f links after the structural transformations are interchanged. It is easily checked that if the f and x links are both red then this procedure performs exactly the second and third transformations of Fig. 3. (The program is more general than seems necessary because we shall have occasion to use it within several other algorithms.)

The implementation makes obvious an extension to 2-3-4 trees, which also fit nicely into the dichromatic framework. The idea is to allow 4-nodes (nodes with three keys and four sons), which are represented as in Fig. 6. Now splitting only has to be done upon insertion into a 4-node, and the split corresponds to simply complementing the colors of the three nodes involved, as shown in Fig. 7. (We shall refer to this operation as a "color flip".) Insertion into a 3-node may require a single or double rotation to convert it into a 4-node: the necessary transformations are exactly the transformations of Fig. 3 without the color flips. Fig. 8 shows the sequence of 2-3-4 trees corresponding to Fig. 4. The implementation of this method corresponds to moving the test for "balance" out of the splitting loop, as shown in Program 3. Insertion into a 2-3-4 tree involves a sequence of color flips and at most one rotation; insertion into a 2-3 tree may involve many rotations. These trees have been called "symmetric binary B-trees" by R. Bayer [Ba].

Let us now make a few observations about these implementations. First, note that if we start with an internal node and follow any two

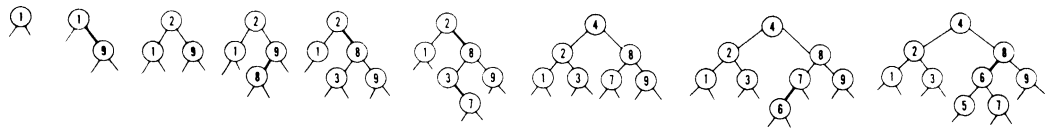

Figure 4. Constructing a 2-3 tree

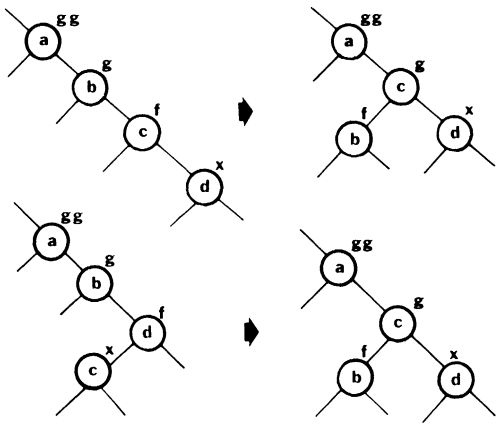

Figure 5. Local balancing transformations

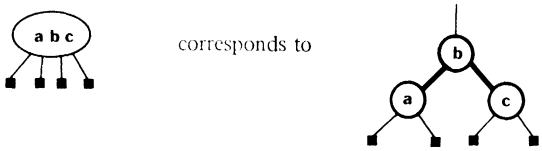

Figure 6. The binarization of a 4-node

paths to external nodes, then the number of black arcs on these two paths will be the same. (This follows from the defining property of the trees.) Next observe that two consecutive red links never appear on any path. (This follows from the implementation which keeps the red links "inside" the internal nodes.) In fact, an alternate way to view the algorithms is that they make exactly the transformations necessary to maintain both these properties.

A variety of balanced tree algorithms can be described in terms of these simple transformations on dichromatic binary trees. In general, we shall consider families of trees which satisfy the following two conditions:

1. External nodes are black; internal nodes may be red or black.

2. For each internal node, all paths from it to external nodes contain the same number of black arcs.

There will further be a third condition, depending on the family of trees we consider, which expresses the "balance" property. In essence this condition will restrict the size of connected red subtrees that can arise. Such conditions can often be expressed in many equivalent forms. For 2-3-4 trees, as we have seen, an appropriate condition is

```
record node (color m; reference (node) l, r; integer k);
reference (node) h, z;
reference (node) array p(0::50); integer i;

procedure initialize;
    begin z ← node(black, , , ); h ← node(black, z, z, -∞); end;

procedure search and insert (integer value v; reference (node) h);
    begin reference (node) x; logical success;
    x ← h; k(z) ← v; i ← 0; success ← true;
    loop until v = k(x):
        p(i) ← x; i ← i+1; if v < k(x) then x ← l(x) else x ← r(x) endif;
    repeat;
    if x = z
    then p(i) ← x ← node(black, z, z, v); m(z) ← red; success ← false;
        if v < k(p(i-1)) then l(p(i-1)) ← x else r(p(i-1)) ← x endif;
        loop while m(l(p(i))) = m(r(p(i))) = red:
            m(l(p(i))) ← m(r(p(i))) ← black; m(p(i)) ← red; i ← i−2;
            if m(p(i+1)) = red then balance(p(i−1), p(i), p(i+1), p(i+2))
                                   else i ← i+1 endif;
        repeat;
    endif;
    m(r(h)) ← black;
    return(x, success);
    end "search and insert"
```

Program 1. 2-3 trees.

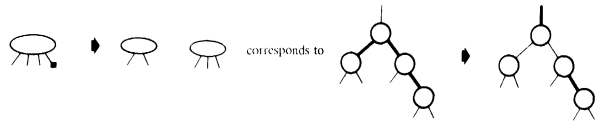

Figure 7. Insertion into a 4-node

3. (2-3-4) No path from an internal node to an external node contains two red links in a row.

Another way to express this condition is

3'. (2-3-4) The only allowed connected red subtrees are those shown in Fig. 9.

It is straightforward to check that any binary tree satisfying conditions 1, 2, and 3 (or 3') can be uniquely "decoded" into a 2-3-4 tree. (However, not all such trees can be produced by Program 3.)

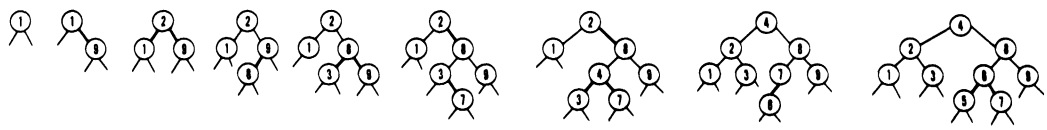

Figure 8. Constructing a 2-3-4 tree

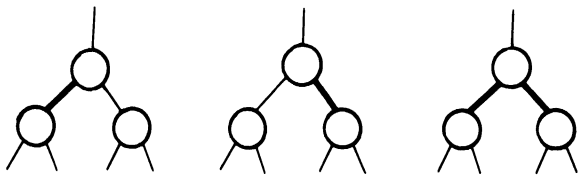

Figure 9. The allowed red subtrees in a 2-3-4 tree

```
procedure balance (reference (node) value result gg, g, f, x);
    begin
    if (v < k(g)) ≠ (v < k(f))
    then if v < k(f) then l(f) ← r(x); r(x) ← f else r(f) ← l(x); l(x) ← f endif;
        f ← x;
    endif;
    if v < k(g) then l(g) ← r(f); r(f) ← g else r(g) ← l(f); l(f) ← g endif;
    g ← f; m(f) ↔ m(g);
    if v < k(gg) then l(gg) ← g else r(gg) ← g endif;
    end "balance"
```

Program 2. Local balancing.

```
procedure initialize;
    begin y ← node(red, , , ); z ← node(black,y,y, );
        p(-1) ← h ← node(black, z, z, -∞); end;
```

```
    loop while m(l(p(i))) = m(r(p(i))) = red:
        m(l(p(i))) ← m(r(p(i))) ← black; m(p(i)) ← red; i ← i−2;
    repeat;
    if m(p(i+1)) = m(p(i+2)) then balance(p(i−1), p(i), p(i+1), p(i+2)) endif;
```

Program 3. 2-3-4 trees (new initialization and balancing loop for Program 1).

Note that since no two reds can appear in a row, the "binarized" 2-3-4 tree satisfies the property that from any node the ratio of the shortest to the longest path is at most 2 (take one all black path and another alternating between red and black.) From this it follows immediately that if the tree has N internal nodes, then the length of the longest path is $O(\lg N)$ (in fact $\leq 2\lg N$), and so a search or insertion takes logarithmic time in the worst case. All of the algorithms that we discuss have this property.

For another example, consider the extension of the above properties to handle general B-trees. An appropriate "balance condition" is:

3'. (B-tree of order m). The only allowed connected red subtrees are perfectly balanced trees with $\lfloor m/2 \rfloor - 1$ to $m-1$ nodes.

The implementation of a B-tree of order m involves representing a node with x sons by a perfectly balanced subtree of x leaves. As in the 2-3-4 case, insertions into such a subtree will sometimes require generalized rotations to keep it perfectly balanced. When the red subtree grows to have $m+1$ leaves, it "splits" by performing a color flip at the root, thus giving rise to two smaller red subtrees. We have not yet precisely defined the term "perfectly balanced", and this leaves some flexibility in the implementation. If we defined a "perfectly balanced" tree as one in which, for each node, the number of nodes in the left subtree differs by at most one from the number of nodes in the right subtree, then we get the standard B-tree implementation. Another plausible definition is to call a tree "perfectly balanced" if all its external nodes appear on the bottom two levels. This way requires fewer transformations to maintain, but it will not generally "split" nodes exactly in half, without an accompanying generalized rotation. In either case, an implied but weaker (equivalent when m is a power of 2) condition is

3. (B-tree of order m). No path from an internal node to an external node contains more than $\lfloor \lg m \rfloor - 1$ consecutive red links.

The usual implementation of B-trees involves storing nodes as sorted lists of keys in separate pages. This may be characterized as an implementation which avoids explicitly storing red links. (In fact the "perfectly balanced" condition becomes irrelevant in such implementations, since for each node size, only one connected red subtree is allowed, that whose shape is determined by the search strategy within the page.)

The above characterization of B-trees in the dichromatic framework requires balancing both to limit the size of the connected red subtrees which arise and to keep those subtrees locally balanced. An alternative is to not require local balancing at all. This leads to fewer transformations, but ones which are more complex. Also, the resulting trees are less balanced. For the 2-3-4 case, this corresponds to doing simple insertions to 3-nodes (thus allowing 4-nodes to be represented in three different ways, two of which have two red arcs in a row), and doing the local balance upon insertion into the 4-node.

It is much more surprising that AVL trees can also be embedded in our dichromatic framework. These are trees in which the heights of the subtrees rooted at the sons of each node differ by at most one. This balance condition appears, at first sight, to be of a quite

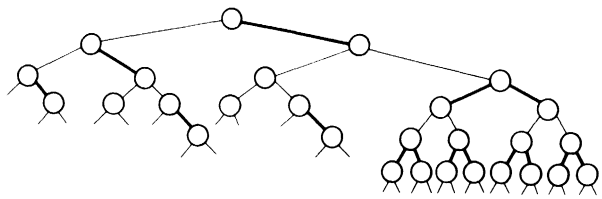

Figure 10. An AVL tree colored as a 2-3-4 tree

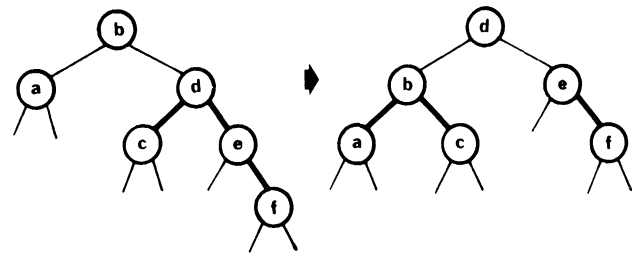

Figure 11. The AVL rotation

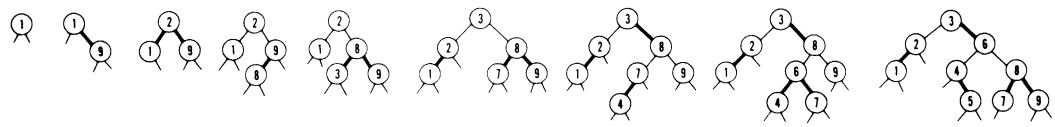

Figure 12. Constructing an AVL tree

different nature than those we have been considering. Bayer [Ba] essentially showed that every AVL tree is a 2-3-4 tree: in the dichromatic notation this can be described quite succintly. Define the *height* of a node to be the length of the longest path from that node to an external node. To make an AVL tree into a 2-3-4 tree simply color red exactly those links which go from a node at an even height to a node at an odd height. Fig. 10 shows how a Fibonacci tree [Kn] looks after coloring. Now, condition 3 above for 2-3-4 trees follows immediately. Condition 2 can easily be proven by induction on the height of the tree, using the stronger hypothesis that avery node of height h has exactly $\lfloor h/2 \rfloor$ black links on every path to an external node. (Not every AVL tree can be viewed as a 2-3 tree, however.)

Observe that a 2-node corresponds to a node at an odd height whose father is at an odd height (2 greater); a 3-node corresponds to a node at an even height with one son at an odd height (1 less) and one son at an even height (2 less); and a 4-node corresponds to a node at an even height whose sons are both at an odd height (1 less). Conversely we note that a node with two red sons is balanced; a node with one red son and one black is heavy (in the AVL sense) towards the red son, and a node with both black sons has a balance factor analogously determined by the colors of its grandsons. We need go no further, as not all grandsons can be black. (Why?)

With this correspondance in mind, we can transform our algorithm for 2-3-4 tree insertion into an algorithm for AVL tree insertion by the addition of a simple test. Insertion into a 4-node implies that the height of the two nodes on the insertion path is incremented by 1: the color flip of Fig. 7 is precisely the necessary transformation. Let us call the node newly painted red x. To maintain the AVL property, it is necessary to check the brother of x. If the brother's height is now two less that that of x, a rebalancing transformation is necessary. It turns out that one local balancing transformation, a single or double rotation as in Fig. 5, involving the three nodes above x (along with two color changes) suffices to terminate the insertion. This transformation is shown in Fig. 11. (Only one case is shown; the other three are similar.) On the other hand, if the height of the brother of x is now equal to or less than that of x, then we proceed as before: the red link from the color flip is the message that the

```
procedure initialize;
    begin y ← node(red, , , );  z ← node(black,y,y, );
        p(-3) ← p(-2) ← p(-1) ← h ← node(black, z, z, -∞); end;

        loop while m(l(p(i))) = m(r(p(i))) = red:
            m(l(p(i))) ← m(r(p(i))) ← black;  m(p(i)) ← red;  i ← i−2;
            if v < k(p(i−1) then b ← r(p(i−1) else b ← l(p(i−1)) endif;
            if m(b) = m(l(b)) = m(r(b)) = black and m(l(p(i)) = m(r(p(i))) = red
                then m(p(i+1)) ← black;  m(b) ← red;  i ← i−1 endif;
        repeat;
        if m(p(i+1)) = m(p(i+2)) then balance(p(i−1), p(i), p(i+1), p(i+2)) endif;
```

Program 4. AVL trees (new initialization and balancing loop for Program 1).

subtree below has increased in height by 1 (from even to odd) and that the height of its brother is equal or one less. With this extra condition the transformations of Figs. 2 and 3 suffice to terminate the insertion when a 2-node or 3-node is encountered. Program 4 gives an implementation for AVL trees based on these comments. (It makes use of the fact that a 4-node has height one greater than its brother if and only if both its brother and its father are black.) The tree sequence for our sample insertions is given in Fig. 12.

Another way to view this implementation is that the new statement checks to see if an overflowing 4-node has a brother which is a 2-node. In that case we do a rotation instead of a color flip and terminate the algorithm. After the rotation we have a 4-node and a 3-node, where before we had a 2-node and an overflowing 4-node. This is exactly one instance of Bayer and McCreight's suggested

improvement to the standard B-tree algorithms [BaMc]: before splitting a node, check to see if some of the mass can be passed on to a brother.

If we implement the full "brother" heuristic, that is also transform an overflowing 4-node with a brother which is a 3-node into two 4-nodes, then we obtain a different kind of tree, which might be called a *second order* AVL tree. These trees satisfy the following stronger height condition: If we consider the four subtrees at depth two below any node, then the heights of these subtrees are within one of each other (for AVL they could differ by as much as two). Thus these trees are more strongly balanced than AVL. We can show, for example, that in such a tree of N leaves the longest path is at most $1.34\lg N$ (as compared to $1.44\lg N$ for AVL). It is, however, cumbersome to implement these second order trees, as the transformation required is not always a simple single or double rotation; rather, up to nine pointers may have to be changed.

The standard implementation of AVL trees involves keeping two bits per node to encode the three cases (i) the node is balanced, (ii) the left subtree is one deeper, and (iii) the right subtree is one deeper. The necessity of maintaining two bits per node has been viewed as disadvantage, and some researchers have dealt with modifying the basic properties of the trees in order to implement them with one one bit per node [Ko], [KK]. Program 4 gives a direct implementation using only bit per node. M. Brown [Br2] has remarked that this can also be done in a more straightforward way, by pushing the two bits of the balance factor down, one to each son. This corresponds to an alternative coloring of the trees: a node is marked red if its height is one greater than that of its brother. In this coloring, if red links are given weight 2 (and black links weight 1), then, from any internal node, all paths to external nodes have the same weight. (In our framework, red links are given weight 0.) It is also possible to color 2-3 and 2-3-4 trees with black and (double-weight) red links to give a constant weighted path length from each node: color both sons of each 2-node and the "upper" son of each 3-node red. This leads to an alternate dichromatic framework to the one we have been discussing. We have chosen to use zero weight links because the algorithms appear to be somewhat simpler.

All of the algorithms described here have two features which make them cumbersome to implement. First, there are two loops: one controlling the search (going down the tree), and one controlling the insertion (normally going up the tree). Second, the code for the balance procedure is rather cumbersome as it has to handle the four cases of left and right single and double rotations. In the next section, we will see new algorithms which avoid both of these difficulties.

2. Top-Down Algorithms

The new algorithms, which also are conveniently embedded within the dichromatic framework, are based on the common theme that the rebalancing transformations are applied on the way down the tree during an update operation. Thus, when an insertion search encounters an external node, the record being inserted can be attached right there, and the operation is complete. The algorithms need not maintain a stack, since no portion of the search path need be traversed again to restore the balance condition. In this respect, the algorithms are similar to the *weight-balanced* trees introduced by Reingold [RND]. Unfortunately those trees seem to require considerably more than one bit of balance information per node.

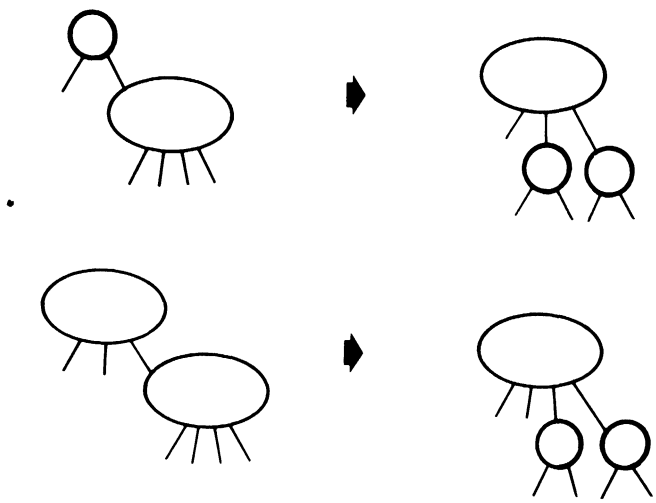

Figure 13. Top-down 2-3-4 tree transformations

The benefits of rebalancing on the way down will become more apparent in subsequent sections where we discuss performance issues. For the moment suffice it to mention that we can at least hope for code which is simple, efficient, and elegant since only one loop is necessary. Top-down schemes will also have inhererent advantages for parallel updating, as each writer will need to lock only a bounded context around itself in the tree.

Perhaps the easiest such algorithm to explain is a top-down insertion algorithm for 2-3-4 trees. Such an algorithm can be build out of exactly the same transformations that were used in the more traditional bottom-up implementation presented in the previous section. The general idea is quite easy to explain, even for a general B-tree. As we go down a path, we split an encountered node if it is full, and insert the splitting key into the father. Note that the father cannot itself be full, so the splitting will not propagate.

Fig. 13 shows the transformations involved for the 2-3-4 case: a 2-node attached to a 4-node becomes a 3-node attached to two 2-nodes, and a 3-node attached to a 4-node becomes a 4-node attached to two 2-nodes. The transformations required for the colored binary tree are exactly those of Figs. 7, 4, and 2.

An implementation for 2-3-4 trees with rebalancing done on the way down is given in Program 5. It is interesting to compare this implementation with the standard bottom up implementation of Program 3. Each does a color flip when the current node's sons are both red and then a rotation if the current node's father is also red. The top-down implementation manages to perform all the necessary transformations on the way down the tree. In order top perform the rotations, it is necessary to keep hold of the great-grandfather (gg), grandfather (g), and father (f) of the current node. The test for the actual insertion has been moved out of the inner loop by the artifact of making the universal external node (z) have two red sons. The sequence of trees produced for our sample keys is that of Fig. 8. It is possible to implement single and double rotations with somewhat less code than the balancing procedure of Program 2. The idea is to separate the two single rotations that Program 2 does to implement the double rotation. After the first rotation, the "current" node pointer is set high enough in the tree to set up the next rotation. The

```
record node (color m; reference (node) l, r; integer k);
reference (node) h, y, z;

procedure initialize;
    begin y ← node(red, , , ); z ← node(black,y,y, ); h ← node(black, z, z, -∞); end;

procedure search and insert (integer value v; reference (node) h);
    begin reference (node) x, gg, g, f; logical success;
    x ← h; k(z) ← v; success ← true;
    loop until v = k(x):
        gg ← g; g ← f; f ← x;
        if v < k(x) then x ← l(x) else x ← r(x) endif;
        if m(l(x)) = m(r(x)) = red then
            if x = z then x ← node(black, z, z, v); success ← false;
                                if v < k(f) then l(f) ← x else r(f) ← x endif;
            endif;
            m(l(x)) ← m(r(x)) ← black; m(x) ← red;
            if m(f) = red then balance(gg, g, f, x); x ← g endif;
        endif;
    repeat;
    m(r(h)) ← black;
    return(x, success);
    end "search and insert"
```

Program 5. Top-down 2-3-4 trees.

```
if m(l(x)) = m(r(x)) = red or m(x) = m(l(x)) = red or m(x) = m(r(x)) = red then
    if x = z then x ← node(black, z, z, v); success ← false endif;
    if m(x) = black then m(l(x)) ← m(r(x)) ← black; m(x) ← red
                    else m(x) ← black; m(f) ← red endif;
    if m(x) = black or (m(f) = red and (v < k(g)) ≠ (v < k(f))) then
        if v < k(f) then l(f) ← r(x); r(x) ← f; f ← g
                    else r(f) ← l(x); l(x) ← f; f ← g endif
    endif;
    if v < k(f) then l(f) ← x else r(f) ← x endif;
    if m(x) = black and (m(f) = red or f = g) then x ← gg endif;
endif;
```

Program 6. Top-down 2-3-4 trees (alternate balancing code for Program 5).

trick is to maintain the colors properly. The first part of the double rotation involves no color changes: both nodes involved are red before and after the rotation. The single rotation (which is also the second part of the double rotation) requires that the colors of the two nodes involved be switched. The algorithm proceeds as follows when a color flip causes a node to become red, and the father of that node is also red. First, a single rotation is performed if necessary to make the two red links go in the same direction. Second, the current pointer is set to its great-grandfather node. Third, a single rotation is performed when the two reds in a row are encountered, to complete the balancing operation. This requires an extra test within the inner loop, but the resulting code sequence is quite compact, as shown in Program 6.

It is also possible to implement AVL trees from the top down, by adding the "brother" transformation of Fig. 11 to transform a 4-node whose brother is a 2-node into a 3-node whose brother is a 3-node, or a 2-node whose brother is a 4-node. However, 2-3 trees are not easily handled, because the splitting occurs when a node is full, not when it has overflowed. Thus a 3-node would have to split into a 2-node and a 1-node, which leads to obvious complications.

There are many variants of Program 5 which work on the same basic theme: *"on the way down the tree, if a node with two red sons is encountered do a color flip, and if two red links in a row are encountered, balance the tree nodes they connect."* A remarkable variety of different tree structures can result depending upon: (i) which node involved in a transformation is the one causing it to happen, (ii) which transformation is given preference when more than one is applicable, and (iii) whether the application of a transformation ought to disable another from happening immediately. Program 5 corresponds to (i) having the color flip done by the top node involved and the balance done by the the bottom node involved, (ii) ensuring that only one transformation is applicable at each node, and (iii) disabling the color flip after the balance but enabling the balance after the color flip.

One tempting variant is to (i) have both the color flip and the balance done by the top node involved, (ii) still ensure that only one transformation is applicable at each node, and (iii) not disable any transformations. The sequence of trees produced by such an algorithm for our sample keys is shown in Fig. 14. This algorithm corresponds to 2-3-4 trees whose 4-nodes are represented by three nodes connected with two red links in any orientation. Fewer balancing transformations are involved, but the trees are less balanced, and the algorithm is difficult to implement cleanly, since a node may have to examine boths sons and two grandsons.

Another possibility is to change Program 6 to do a color flip after each balance (but then disable further balances which would involve going further back up the tree). Fig. 15 shows a particularly bad sequence of keys for this variant (the initial stages are omitted). Not only is the number of possible red subtrees greatly increased, but also three reds in a row could occur! (Consider, for example, what happens if the last tree in Fig. 15 is connected to some red node and a 12 is inserted. Then *two* red links are passed up, resulting in three reds in a row.) This example illustrates that some caution must be exercised if good balanced trees are to be produced.

However, there is still a great deal of flexibility left in designing top-down algorithms within this framework. As we shall see in the next section, the algorithms that we have been considering have essentially the same average case performance, so we should look for an algorithm which is easily implemented. One goal might be to find an algorithm which doesn't do any "double" rotations on the way down the tree. It turns out that such an algorithm is easily derived from Program 6, by simply removing all references to *gg*. The result, given as Program 7, is a method which allows two reds in a row to be encountered on way down the tree, but only if they are oriented in the same direction. Fig. 16 shows the operation of this algorithm on our sample keys. The allowed connected red subtrees are shown in Fig. 17 (only one from each symmetric pair is included). The meaning of the labels on those trees will be discussed below.

The example above indicates that we must be careful to prove that Program 7 operates in the way that we expect. In particular, we need

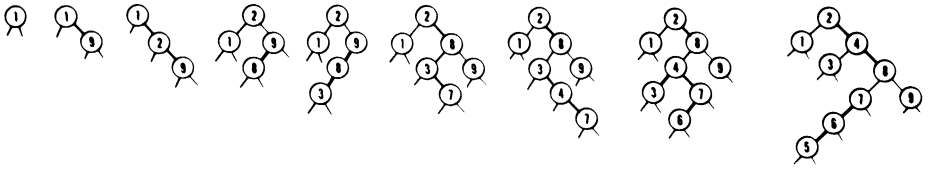

Figure 14. Constructing a top-down tree (first variant)

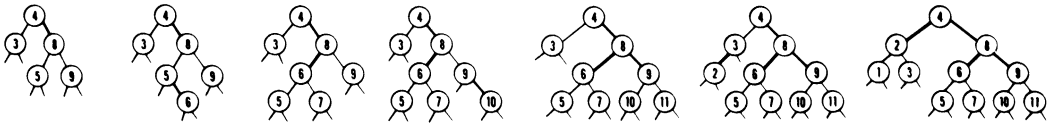

Figure 15. Constructing a top-down tree (second variant)

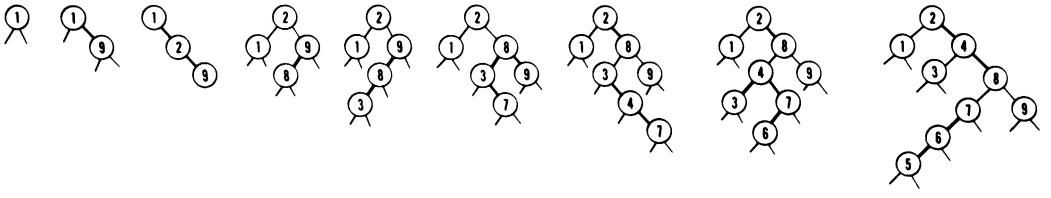

Figure 16. Constructing a top-down single rotation tree

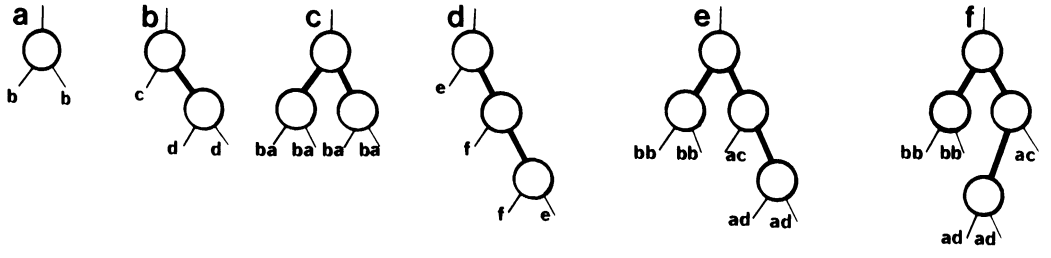

Figure 17. The allowed red subtrees for program 7.

to prove that the list L of allowed connected red subtrees is "closed" under the insertion operation. We can think of the algorithm as a sequence of traversals of the trees of L, each of which may cause one tree in L to be transformed into another. Although each tree has a black link into the root and black links at the leaves, the situation is complicated by the fact that the link into the root of the tree labelled c may become red. From the point of view of the subtree above that link, one of its bottom links will become red. We shall refer to this phenomenon as "passing a red up". This of course can also happen when the insertion terminates and an external node is replaced by a new red node. The situation is more completely described by the following lemmas:

LEMMA 2.1. Suppose that a red subtree in L is traversed top-down during the execution of Program 7, and that the subtree is exited on black link t. Then the following facts are true:

(i) link t may become red either by insertion of a new node or because it points to the root of a type c subtree;

(ii) if link t becomes red, then the links below it will become black, and no subsequent transformation during the current insertion will change their color;

(iii) whether link t becomes red or not, any connected red subtree resulting from transformations on the tree being traversed is in L.

LEMMA 2.2. Each of the trees in L does in fact arise.

These lemmas are easily proven by case analysis from Fig. 17. The letters on each of the black links leaving the trees denote the trees in L that will be formed if the subtree in question is traversed during execution of Program 7, the subtree is exited at that black link, and that black link turns red.

Since two reds is a row are allowed, the ratio of the length of the longest path to the shortes path in the tree is now 3 (consider the situation when the keys inserted are in increasing order), so the length of the longest path in a tree of N nodes is bounded by $3 \lg N$.

```
record node (color m; reference (node) l, r; integer k);
reference (node) h, y, z;

procedure initialize;
    begin y ← node(red, , , ); z ← node(black,y,y, ); h ← node(black, z, z, -∞); end;

procedure search and insert (integer value v; reference (node) h);
    begin reference (node) x, g, f; logical success;
    x ← h; k(z) ← v; success ← true;
    loop until v = k(x):
        g ← f; f ← x; if v < k(x) then x ← l(x) else x ← r(x) endif;
        if m(l(x)) = m(r(x)) = red or m(x) = m(l(x)) = red or m(x) = m(r(x)) = red then
            if x = z then x ← node(black, z, z, v); success ← false endif;
            if m(x) = black then m(l(x)) ← m(r(x)) ← black; m(x) ← red
                            else m(x) ← black; m(f) ← red endif;
            if m(x) = black or (m(f) = red and (v < k(g)) ≠ (v < k(f))) then
                if v < k(f) then l(f) ← r(x); r(x) ← f; f ← g
                            else r(f) ← l(x); l(x) ← f; f ← g endif
            endif;
            if v < k(f) then l(f) ← x else r(f) ← x endif;
        endif;
    repeat;
    m(r(h)) ← black;
    return(x, success);
    end "search and insert"
```

Program 7. Top-down single rotation trees (2-3-4-5 trees).

The implementation in Program 7 is notable for its brevity: it requires only about 60% as much code as the classical AVL and 2-3 algorithms. The following section shows that it can also be expected to perform as well as these algorithms in a dynamic sense.

3. Performance Comparisons

Since balanced trees are suitable for a wide variety of applications, there are a number of different measures which could be used to compare the various algorithms we have been discussing. In the previous sections we have dealt with some static issues such as program size and overhead required. In this section we shall concentrate on the dynamic statistics of the various algorithms. There are essentially two costs of interest. One is the search cost, when a tree built by one of the algorithms is used for searches only. The other is the insertion, or balancing cost. The first measures the balance quality of the trees built by the algorithm; the second the effort consumed in achieving this balance. We have already seen examples which supprt the intuitive notion that search cost may be traded for insertion cost and vice versa.

The dichromatic framework makes the task of comparing the algorithms somewhat simpler, since the properties of the binary trees produced can be studied in a color-blind manner. (As mentioned above, this corresponds to explicitly counting node-internal comparisons for 2-3 and 2-3-4 trees.) In what follows, we shall concentrate on the cost of unsuccessful searches: the length of the path traversed to an external node. (The cost of successful searches can be derived from this in a standard way [Kn].) In particular, we shall consider three different measures.

One is the *worst-case path* cost, which is the length of the longest path among all trees of N keys built by the algorithm. The second, and perhaps more representative, is the *worst-case path length* cost, which is the average search cost for the tree of maximal external path length, among all trees built by the algorithm. And finally we have the *average* cost, which is the average search cost for a *random* tree built by the algorithm, under the usual model that the $N!$ possible permutations of the N keys used in building the tree are equally likely. Note also that for a given class of trees, the average, worst-case path length, and worst-case path costs form a non-decreasing sequence of numbers.

For a perfectly balanced tree of N keys the worst-case path, worst-case path length, and average cost are all essentially $\lg N$, so this will form our *de facto* standard of comparison. Define the *fractional* cost to be the supremum, as N gets large, of the ratio of the cost in question to $\lg N$. Thus the fractional worst-case path, worst-case path length, and average cost for perfectly balanced trees are all trivially 1. For trees produced by our algorithms, the fractional costs will be ≥ 1.

The situation for worst-case path cost is the simplest to analyze. It is well-known that for AVL trees the fractional cost is $1/\lg\varphi = 1.44...$, which is achieved by the Fibonacci trees [Kn]. (A Fibonacci tree of height n is constructed by putting a Fibonacci tree of height $n-2$ to the left of the root and one of height $n-1$ to the right of the root. The tree of height 0 is a single external node; the tree of height 1 is an internal node with two external sons.) For 2-3 or 2-3-4 trees, a tree which is entirely 2-nodes except for one path of 3-nodes gives a fractional cost of 2 (which is clearly the highest possible). Similarly, from the comments in the previous section, the fractional cost for the trees generated by Program 7 is 3.

For the fractional worst-case path length cost the situation is more difficult and interesting. We have been able to improve on a number of previously known bounds. A common misconception is that Fibonacci trees maximize path length among all AVL trees of the same size. This would be nice, since the fractional cost for Fibonacci trees under this metric is quite low. A Fibonacci tree of height n has F_{n+2} external nodes, and its external path length is defined by the recurrence

$$E_n = E_{n-1} + E_{n-2} + F_{n+2}, \qquad n \geq 2,$$

with $E_0 = 0$, $E_1 = 1$. The solution to this recurrence is

$$E_n = 4n/5\ F_{n+1} + (3n+3)/5\ F_n,$$

so the fractional path length cost for a Fibonacci tree of height n is

$$\text{limsup } (4nF_{n+1}/5 + (3n+3)F_n/5)/F_{n+2}\lg F_{n+2}) = (4/(5\varphi) + 3/(5\varphi^2))/\lg\varphi = 1.04...$$

(Recall that $F_n = \varphi^n/5^{1/2}$ rounded to the nearest integer, where $\varphi = (5^{1/2}+1)/2$ is the "golden ratio".) Fibonacci trees are only about 4% worse than optimal under this metric.

However, it is possible to construct AVL trees which are much worse. Given a Fibonacci tree of height n and some positive integer k, $k < n$, we can construct an "overweight" Fibonacci tree by replacing the rightmost (bottommost) Fibonacci subtree of height k by a complete binary tree with 2^k external nodes. Fig. 10 shows such a tree with $n = 6$ and $k = 4$. By appropriately choosing k, we can get a tree in which asymptotically all paths have the maximal possible length. Specifically, the fractional path length for the overweight Fibonacci tree is

$$\text{limsup } (E_n - E_k + k2^k + (n-k)(2^k - F_{k+2})) / (F_n - F_k + 2^k)\lg(F_n - F_k + 2^k)).$$

If k is chosen so that 2^k is about nF_n, then this limit equals

$$\text{limsup } n^2F_n/nF_n\lg F_n,$$

which approaches $1/\lg\varphi$.

For 2-3-4 trees, a similar construction leads to a fractional worst-case path length of 2. The situation for 2-3 trees is less clear. (This problem has been studied by Rosenberg and Snyder [RS].) We can easily upper bound this cost by 2, and and an analogous construction to the above yields 2-3 trees with fractional cost of $2 - 1/3\lg 3$. We start the construction by bulding the already considered *scrawny* trees which have maximum height for their number of leaves. (In the sequel heights will always refer to the dichromatic framework representation of these B-trees.) In the 2-3 or 2-3-4 case such trees are clearly composed of a single path of 3-nodes with everyone else a 2-node. (A 2-node is also allowed at the root, if the height is odd.) These scrawny trees naturally correspond to the Fibonacci trees of the AVL case. Without loss of generality, we can assume that the rightmost chain is the one consisting of 3-nodes. To make these trees overweight, we choose a k and replace the righmost scrawny tree of height k by, in each case, the *bushiest* possible tree of height k. This bushy tree consists entirely of 3-nodes in the 2-3 case, and entirely of 4-nodes in the 2-3-4 case. An appropriate choice of k now as a function of n, the total tree height, completes the argument. We only present some of the details of the 2-3 argument, as the 2-3-4 argument is somewhat simpler. The fractional worst-case path length cost for the 2-3 tree just constructed is

$$\text{limsup } (5k3^{k/2}/6 + (n-k)3^{k/2} + n2^{n/2}) / (3^{k/2} + 2\times 2^{n/2})\lg(3^{k/2} + 2\times 2^{n/2})).$$

We now let $k = n/\lg 3 + \lg n$. It is then easy to check that the above limsup is $2 - 1/3\lg 3$.

Although we have not carried out the construction for the trees of Program 7, it is reasonable to conjecture that a fractional cost of 3 can be obtained.

No balanced tree algorithm has yet been completely analyzed under the average cost metric. The classical bottom up algorithms are extremely difficult to analyze because they do not "preserve" randomness: given a random tree, its subtrees are not random trees. On the other hand, it is possible that the top-down algorithms may submit to analysis, because they perform their transformations blindly in a consistent fashion. However, even under the most generous randomness assumptions, the recurrences that arise in the analysis seem intractable. The question of whether any of these algorithms are truly asymptotic to $\lg N$ on the average is the most fundamental open question in the analysis of balanced trees.

It is possible to do a *fringe* analysis, of the average case behavior assuming that the rebalancing transformations occur only at the "bottom" of the tree. Yao [Y] showed how to compute the average number of 2-nodes and 3-nodes at the bottom levels of 2-3 trees, and Brown [Br?] gave some similar results for AVL trees. Neither gave any results concerning path lengths, but these can be derived with the help of the following lemma for (arbitrary) binary search trees.

LEMMA 3. Given any binary search tree with n keys, let the average unsuccessful search cost be C_n. Then the average unsuccessful search cost after a random insertion is

$$C_{n+1} = C_n + 2/(n+2).$$

Proof: The external path length of the tree is $(n+1)C_n$. Each external node is equally likely to receive the insertion, with probability $1/(n+1)$. Notice that if the insertions is at level i, then the external path length increases by $i+2$ (two new external nodes are created at level $i+1$, less the one at level i). Therefore, the average increase in external path length is

$$1/(n+1)\sum(\text{level}(x)+2) = C_n + 2,$$

where the sum is taken over all external nodes x. This leads immediately to the recurrence

$$(n+2)C_{n+1} = (n+1)C_n + C_n + 2,$$

which proves the lemma. ∎

This lemma has a number of interesting consequences. By telescoping the recurrence, we get

$$C_N = C_n + 2H_{N+1} - 2H_{n+1}, \qquad \text{for } N > n,$$

where H_N denotes the N-th harmonic number [Kn]. (In particular, taking $n = 0$ and $C_0 = 0$ gives the well-known average unsuccessful search cost for random trees, $C_N = 2H_{N+1} - 2$, which says that the fractional cost for such trees is $2\ln 2 = 1.38...$, since $H_N = \ln N + O(1)$.) If we start with a "seed" tree which is perfectly balanced, $C_n = \lg n$, then we get

$$C_N = \lg n + 2\lg(N/n) + O(1),$$

and by taking n large enough, say $n = O(N/\lg N)$, then we have trees with an optimum fractional cost,

$$C_N = \lg N + O(\log\log N).$$

This means that no balancing need be done at all if it can be ensured that the tree is perfectly balanced after a a sufficient number of keys have been inserted.

Returning to the fringe analysis, let us consider how to calculate the average search cost for 2-3 trees under the assumption that rebalancing is only done at the bottom. Yao showed that the ratio of

2-nodes to 3-nodes on the bottom level is 2:1, so any particular external node belongs to a 3-node with probability 3/7 and a rotation is done on insertion with probability 2/7. If a rotation is done for an insertion on level i, then the external path length is increased by only $i+1$, and Lemma 3 is easily modified to take this into account, with the result that the fractional average cost for such trees is 12/7 ln2 = 1.188... . The result is the same for AVL and 2-3-4 trees.

In general, each of the algorithms that we have considered has a set of allowable connected red subtrees, L. For each tree t in L, the fringe analysis will give us the probability that a random insertion will strike one of the external nodes of t. The average external path length in a tree with N nodes ignoring rotations above the bottom level, is

$$\left(2 - \sum p_t \sum \Delta_x\right) H_{N+1} - 2 .$$

In this formula the first sum is over all t in L, and the second over all external nodes x of t. In addition, p_t denotes the probability of hitting a tree of type t, and Δ_x the saving in path length due to the rotation if one is done and 0 otherwise. (If the external node is at level i this is the difference between $i+2$ and the increase in the external path length *after* the rotation.) Note that rotations at the fringe always reduce the path length; however, this need not be so for rotations higher up in the tree. In fact all the algorithms we have considered can be forced to do rotations that will increase the path length. This is another reason why a complete average case analysis is non-trivial.

As an application, consider the fringe analysis for Program 7, the top-down single rotation trees. Let a, b, c, d, e, f also represent the probabilities that a random insertion strikes an external node in a tree labelled a, b, c, d, e, f respectively if Fig. 17. (Then $2a$, $3b$, $4c$, $4d$, $5e$, $5f$ are the probabilites that the respective trees themselves occur). For simplicity, assume that these probabilities reach steady-state after a sufficient number of nodes have been inserted (Yao shows how to make this precise). Then, from Fig. 17, we can write down the equations

$$\begin{aligned} a &= -2a + 4c + 3e + 3f \\ b &= 2a - 3b + 4c + 4e + 4f \\ c &= b - 4c + e + f \\ d &= 2b - 4d + 2e + 2f \\ e &= 2d - 5e \\ f &= 2d - 5f . \end{aligned}$$

We also have the normalization condition

$$2a + 3b + 4c + 4d + 5e + 5f = 1.$$

The solution to this set of simultaneous equations (there is one redundant equation) is

$$a = 8/105, \quad b = 11/105, \quad c = 3/105,$$
$$d = 6/105, \quad e = 2/105, \quad f = 2/105.$$

Now, the only insertions for which Δ is non-zero are the rightmost three in tree d. The first saves 1, the other two save 2, so the average external path length in a tree with N nodes is

$$(2 - 5d)(H_{N+1} - 1) = 12/7 (H_{N+1} - 1),$$

the same as for 2-3 or AVL trees! (There are easier ways to prove this result; the intention here was to illustrate a general technique for the fringe analysis of any such algorithm.)

The fringe analysis can be extended upwards by considering the set of possible subtrees of red height 2, etc., but these sets tend to be large, and the calculations quickly become prohibitively difficult. (For AVL trees, M. Brown has remarked that the fringe analysis does not seem to extend in this fashion, because the transformation on a tree depends on the shape of the tree rooted at its brother.) It does appear that the fractional average cost quickly approaches 1. On the average, most of the rotations occur at the bottom levels: those higher up present a bad worst case.

The reader is again reminded that these fringe analyses rigorously prove nothing about the average case performance of the algorithms in the previous sections. We are unable to prove that for a given algorithm (e.g., 2-3 trees) the average case behavior of the fringe variant is an upper bound on the average case behavior of the real algorithm, though we conjecture this to be true. However, they can be taken as analytic evidence that the algorithms perform very well on the average.

From a practical standpoint, simulation studies of balanced tree algorithms consistently show that the fractional average case cost is very close to 1. (See [KFSK], [Kn].) Table 1 gives the results of simulations for the five implementations that we have on five different files of 20,000 nodes each. The main empirical observation that can be made from this table is that *on the average all the algorithms have essentially the same behavior.* Furthermore, the performance of all the methods seems to be extremely insensitive to the input. Since the external path length of a perfectly balanced 20,000 node tree is 287248, this data may be interpreted as showing that the average-case fractional cost of these algorithms is approximately 1.02... . Unfortuantely, even for such large N, the value of $\lg N$ is so small that the same data is also consistent with the hypothesis that the fractional cost is 1 (or, in other words, the average external path length is about $\lg N + 0.3$). Though the simulations do not help resolve this theoretical question, they do indicate that the trees are extremely well balanced, since they are within 2% of optimal.

Another point worth noting is that the insertion cost for all of the algorithms is very low. The number of rotations or color flips to be expected is about one every two trips down the tree. Program 7 uses fewer rotations at the expense of a slightly less balanced tree. It is possible to get by with even fewer rotations at the expense of more imbalance: some of the variants mentioned in the previous section have this property. Finally, although one might expect the top-down algorithms to do significantly more rotations than their bottom-up counterparts, the table show this not to be the case. A direct comparison between the top-down and bottom-up 2-3-4 tree algorithms shows their performance statistics to be extremely similar.

Since the algorithms are so similar in performance, it is wise to pay careful attention to the implementation, which can have a very significant effect on the performance. The empirical studies show that the "inner loop" of the algorithms is the search loop, which must therefore be carefully implemented. If searches are to be done much more often than insertions, it may be advisable to have a separate search procedure, then call "search and insert" if the search was unsuccessful. However, for most applications this is probably not worth the trouble, since the extra overhead in the inner loop for all the balanced tree algorithms is so small. The inner loops of the top-down algorithms can be "unwound" so that they involve only one

External path length					
Program	File 1	File 2	File 3	File 4	File 5
2-3 trees (Prog. 1)	292457	292725	291960	292124	292269
2-3-4 trees (Prog. 3)	293315	292680	293727	293010	292464
AVL trees (Prog. 4)	291708	292364	293479	292712	292433
top-down 2-3-4 (Prog. 5)	292816	292364	293479	292712	292433
single rotation (Prog. 7)	294422	294331	294197	294137	294753

single rotations					
Program	File 1	File 2	File 3	File 4	File 5
2-3 trees (Prog. 1)	12537	12569	12543	12437	12534
2-3-4 trees (Prog. 3)	11643	11571	11558	11563	11567
AVL trees (Prog. 4)	14003	13875	14035	13965	13860
top-down 2-3-4 (Prog. 5)	11852	11758	11769	11726	11783
single rotation (Prog. 7)	11365	11040	11398	11306	11209

color flips					
Program	File 1	File 2	File 3	File 4	File 5
2-3 trees (Prog. 1)	14912	14970	14938	14922	14848
2-3-4 trees (Prog. 3)	10280	10231	10238	10346	10280
AVL trees (Prog. 4)	9541	9492	9532	9509	9524
top-down 2-3-4 (Prog. 5)	11419	11393	11339	11439	11380
single rotation (Prog. 7)	9931	9967	9998	9948	10022

Table 1. Empirical data for five programs on five random 20,000 node files

more test than a straight search precedure. This compares favorably with the overhead required to maintain the stack (or remember where to start rebalancing) for the bottom-up algorithms. The test for Programs 6 and 7 is slightly more expensive than that for program 5, but for most applications this is probably worthwhile in view of the simplicity of those algorithms. If search speed is essential or more bits per node are available, then there are other alternatives to consider. For example, on some computers it might be easier to keep the color bits with the links, rather than the nodes. This makes the extra test in the inner loop of the top-down algorithms even simpler to implement.

4. Further Topics

Balanced trees have utility in a wide variety of applications. Besides search and insertion, many other operations are commonly required of such data structures. Some examples are deletion, splitting, concatenation, and selection. A full discussion of these and other problems is given by Knuth [Kn]. Due to lack of space, all of these problems cannot be considered here, but rather we shall atempt to illustrate some of the machinery involved by considering in detail the deletion problem.

For the classical balanced tree deletion algorithms, deletion is generally considered to be harder than insertion. Fortunately, the dichromatic framework and the top-down viewpoint can lead to deletion methods which are not much more complex than insertion. This is illustrated by Program 8, which completes the deletion operation for 2-3-4 trees in one top-down pass. It is well known that it suffices to consider the case that the node to be deleted is on the bottom level (has external sons). This is accomplished by doing a search for the node to be deleted, saving its position in t when it is encountered, and continuing until an external node is hit. Then the father of the external node is the successor to the node to be deleted. The deletion is completed by deleting this father after saving its key in the node pointed to by t. Now, if the bottom level node to be deleted is red, it may simply be removed: the difficulty is when a black node must be deleted. Program 8 ensures that this will never be necessary, by pushing a red down from the root to the bottom. The transformations involved are essentially those of Fig. 13 in reverse, with two additions: (i) 3-nodes are rotated, if necessary, so that the red (bottom) node is traversed, and (ii) if a 2-node is encountered which has a 3- or 4-node for a brother, a balance transformation is performed which makes the node being traversed a 3-node. The various transformations are diagrammed in Fig. 18. The sequence of trees resulting from deleting our sample set of keys, in reverse order from insertion, is shown in Fig. 19. Since Program 8 works for any 2-3-4 tree, it may be used on trees built by either Program 3 or Programs 5 and 6. A similar algorithm is available for Program 7, but 2-3 trees and AVL trees are still somewhat more difficult to handle.

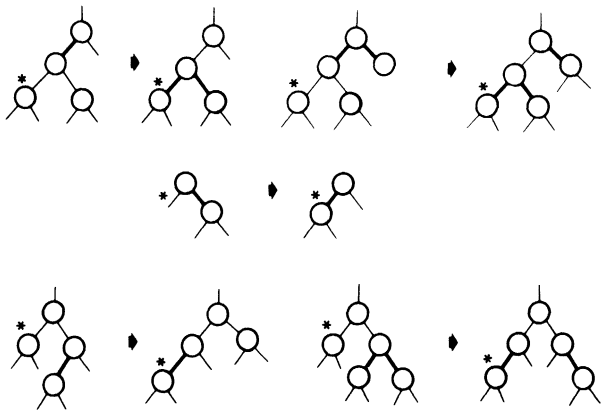

Figure 18. Top-down 2-3-4 deletion transformations

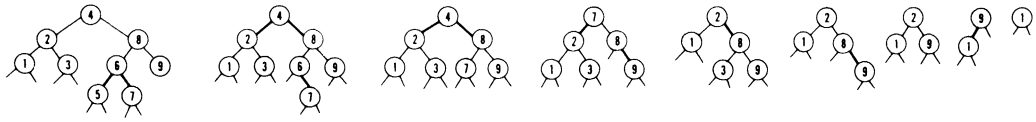

Figure 19. Destroying a 2-3-4 tree

One nice feature of the dichromatic framework is that it allows us to decouple the job of maintaining the tree balanced from the operations of insertion and deletion. We can design a *balancer* which works on the basis of local context only, without having to gather tree-wide information. Such a balancer traverses the tree and uses our standard tranformations: two reds on a path cause a rotation, a black with two reds below causes a color flip. With careful traversal design the balancer can be shown to have the following property. If we start with any red-black tree satisfying conditions 1. and 2. of section 1 then, after the balancer has made $O(\lg N)$ passes over the tree, the resulting tree will be balanced, in the sense of satisfying condition 3 for (one of) our algorithms. (Note that condtions 1. and 2. allow extremely unbalanced trees, for instance ones where all internal nodes are on one red linear chain.) This implies we can run the balancer asynchronously with the tree updaters, and if we guarantee that it receives enough cycles, then we know that the tree will remain well balanced. The simplicity of the rebalancing decisions and transformations makes it attractive to consider putting such a balancer into microcode and/or hardware.

The previous paragraph raised some issues about concurrent access to our trees. As we have already mentioned, the top-down approach implies that inserters and readers do not interfere as long as they lock a small bounded context in the tree around themselves. In fact, it is possible to do the rebalancing in the "shadow" of the real tree, with the result that readers are never locked out at all. The only penalty is that writers will then have to lock a slightly wider context. Deleters are ssomewhat more difficult to handle. The only difficulty is the dangling reference t in the middle of the tree. One then can either lock the search path below t, or else rotate t to the bottom of the tree. (This can be done by a sequence of rotations which maintain the defining balance properties.) Many other ramifications for parallel execution of the top-down approach remain to be explored and we hope to undertake them in a future report. For a discussion of similar issues see the work of Kung and Lehman [KuLe].

In this paper, we have exhibited a framework suitable for studying the implementation and performance of a variety of balanced tree algorithms. Within this framework, we were able to develop new algorithms which perform as well but are significantly simpler than the classical algorithms. The dichromatic framework not only has sufficient flexibility to aid in developing new techniques, but also it is simple enough to perhaps lead to a complete analysis of some balanced tree algorithm.

Acknowledgements: The authors wish to thank Lyle Ramshaw for many helpful discussions and especially for his contributions to section 3. Clark Thompson and Mark Brown offered valuable comments on both the form and the content of the manuscript. Finally, the authors wish to acknowledge the use of the MACSYMA system at MIT for checking some of the calculations in section 3.

5. References

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

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

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

```
procedure delete (integer value v; reference (node) h);
    begin reference (node) x, g, f, t, b, bb;
    x ← h;  k(z) ← -∞;  t ← nil;
    if m(l(r(h))) = m(r(r(h))) = black then m(r(h)) ← red endif;
    loop until x = z:
        g ← f;  f ← x;
        if v < k(x) then x ← l(x);  b ← r(x);  bb ← r(b)
                    else x ← r(x);  b ← l(x);  bb ← l(b) endif;
        if v = k(x) then t ← x endif;
        if m(x) = m(f) = black and m(b) = red then balance(g, f, b, bb) endif;
        if m(x) = m(l(x)) = m(r(x)) = black then
            m(x) ← red;  m(f) ← black;
            if m(b) = m(l(b)) = m(r(b)) = black then m(b) ← red
            elseif m(l(b)) = red then balance(g, f, b, l(b))
            elseif m(r(b)) = red then balance(g, f, b, r(b))
            endif;
        endif;
    repeat;
    m(r(h)) ← m(z) ← black;
    if t = nil then if v < k(g) then l(g) ← z else r(g) ← z endif;
                    k(t) ← k(f) endif;
    return(t);
    end "delete"
```

Program 8. Deletion for 2-3-4 trees.

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

[Br1] Brown M., *A partial analysis of height-balanced trees*, SIAM J. Comp., (to appear)

[Br2] Brown M., *A storage scheme for height-balanced trees*, IPL, (to appear)

[KFSK] Karlton P., Fuller S., Scroggs R., and Kaehler T., *Performance of height-balanced trees*, CACM 19 (1976), 23-28

[Kn] Knuth D., **The Art of Computer Programming**, Vol. III, *Sorting and Searching*, Addison-Wesley, 1973

[KuLe] Kung, H.T. and Lehman, P.L., *A concurrent data-base manipulation problem: binary search trees*, to appear in the proceedings of the 1978 Large Data-Base Conference, Berlin

[RND] Reingold E., Nievergelt J., Deo N. , **Combinatorial Algorithms: Theory and Practice**, Prentice-Hall, 1977

[RS] Rosenberg A., and Snyder L., *Minimal-comparison 2,3-trees*, SIAM J. Comp., (to appear)

[Y] Yao A., *On random 2-3 trees*, Acta Informatica, 9 (1978), p. 159-170