THE UNIVERSITY OF MICHIGAN

COMPUTING RESEARCH LABORATORY[1]

TREE REBALANCING IN OPTIMAL TIME AND SPACE

Quentin F. Stout and Bette L. Warren

CRL-TR-42-84

October 1984

Room 1079, East Engineering Building
Ann Arbor, Michigan 48109
USA
Tel: (313) 763-8000

[1]Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the funding agency.

Tree Rebalancing in Optimal Time and Space

by

Quentin F. Stout[1]

Dept. of Electrical Engineering and Computer Science

University of Michigan

Ann Arbor, MI 48109

Bette L. Warren

Department of Mathematics and Computer Science

Eastern Michigan University

Ypsilanti, MI 48197

[1]Research partially supported by National Science Foundation grant MCS-83-01019.

ABSTRACT

We give a simple algorithm which takes an arbitrary binary search tree and rebalances it to form another of minimal height, using time linear in the number of nodes and only a constant amount of additional space (beyond that used to store the initial tree). This algorithm is therefore optimal in its use of both time and space. Previous algorithms were optimal in at most one of these two measures, or were not applicable to all binary search trees. When the nodes of the tree are stored in an array, a simple addition to our algorithm results in the nodes being stored in sorted order in the initial portion of the array, again using linear time and constant space.

1. INTRODUCTION

A binary search tree is a conceptually simple, reasonably efficient, and widely used structure in which to maintain ordered data. Because the fundamental operations of insertion, deletion, and searching require accessing nodes along a single path from the root, for randomly generated trees of n nodes (using the standard insertion algorithm), the expected times to perform these operations are $\Theta(\log(n))$. Unfortunately, it is possible for a binary tree to have very long branches, and the worst case times are $\Theta(n)$.

In order to avoid the worst-case linear times is necessary to keep the tree balanced, that is, it should not be allowed to have unnecessarily long branches. This problem has been studied intensely, and there are many notions of balance and balancing strategies, such as AVL trees, weight-balanced trees, self-organizing trees, etc.[4]. Here we are concerned with perhaps the simplest strategy: periodically rebalance the entire tree to make an equivalent one of minimal height. This strategy has been discussed by many authors, and several algorithms have been offered [1,3,5]; recently Chang and Iyengar [2] surveyed this work and presented additional algorithms. Despite all of this work, no previous algorithm could be applied to an arbitrary binary search tree and perform the rebalancing in time linear in the number of nodes, while using only a fixed amount of additional space beyond that originally occupied by the tree. The main result of this paper is a simple algorithm which accomplishes this.

A tree of minimal height is sometimes called *route balanced.* Notice that a route balanced tree of n nodes has height $\lfloor lg(n) \rfloor$ ($lg(n) = \log_2(n)$). (The height of a tree of no nodes is -1, the height of a tree of 1 node is 0, and in general the height of a tree with more than 1 node is 1 more than the maximum height of each of the subtrees whose roots are children of the root of the tree.) Given the nodes to be used, route balanced

trees are not unique; for example, Figure 1 shows the 6 route balanced tree shapes with 5 nodes. A stronger requirement is that of *perfect balance,* which requires that at each node, the number of nodes in its left subtree differs by no more than one from the number of nodes in its right subtree. Every perfectly balanced tree is route balanced, but not vice versa. For example, in Figure 1, only trees b,c,d, and e are perfectly balanced.

With the exception of Day [3], previous authors concentrated on creating perfectly balanced trees. While perfect balancing fits naturally into a top-down approach, we know of no reason to prefer a perfectly balanced tree over a route balanced one, and our basic algorithm creates route balanced trees. If, for some reason, a perfectly balanced tree is needed, a modified version of our basic algorithm, still requiring only linear time and constant additional space, can be used. No previous algorithm produced a perfectly balanced tree using only constant additional space.

Our algorithm proceeds in two phases. The binary tree is first transformed into an ordered "vine" in which each parent node has only a right child, and then the vine is converted into a route balanced tree. This strategy is the same as Day's [3]. However, Day requires that the initial tree be threaded and we do not. Threading requires extra space at each node to store a flag indicating whether a pointer points to a child or to an ancestor. (In Day's case an extra sign bit was needed.)

Chang and Iyengar [2] assume that the nodes are stored in an array, while we do not. One of their algorithms has the side benefit that when finished, the nodes are stored in sorted order. In section 3 we show how an easy addition to our algorithm will also perform sorting for nodes stored in an array, again using only linear time and constant additional space.

Throughout, we will use n to denote the number of nodes in the tree. Our algorithms do not require prior knowledge of n.

2. REBALANCING

We will use the following declarations:

```
type nodeptr = ↑node;
    node = record right, left: nodeptr;
                {other components, including the key}
                end;
```

While we use this standard pointer implementation of trees, our algorithms require no special properties of pointers (nor of Pascal) and can easily be modified for a variety of tree implementations with no loss of efficiency.

A procedure tree_to_vine reconfigures the initial tree into an increasing vine, and also returns a count of the number of nodes. Then the procedure vine_to_tree uses the vine and size information to create a balanced tree. To simplify our algorithms, each vine will have a dummy root which contains no data.

Tree_to_vine

The tree_to_vine procedure is quite straightforward. The algorithm proceeds top-down through the tree, creating an initial portion which has been transformed into a vine and a remaining portion of nodes with larger keys which may require further transformation. A pointer "vine_tail" points to the tail of the portion known to be the initial segment of the vine, and a pointer "remainder" points to the root of the portion which may need additional work. Remainder always points to vine_tail↑.right. When remainder is nil the procedure is finished. If remainder points to a node with no left child, then that node can be added to the tail of the vine. Notice that this happens exactly n times. Finally, if remainder points to a node with a left child then a rotation

is performed, as illustrated in Figure 2.

Any node initially reachable from the dummy root via a path of right links retains this property after the rotation. Further, after the rotation, the node which was initially pointed to by remainder↑.left also is reachable via right links. Since each rotation increases by 1 the number of nodes reachable from the dummy root via right links, at most $n-1$ rotations can occur (it is $n-1$, and not n, because the root is initially reachable). Therefore the while-loop can be executed at most $2n-1$ times (and must be executed at least n times), and tree_to_vine finishes in $\Theta(n)$ time.

Vine_to_tree

Two versions of vine_to_tree are given. Each modifies a restricted version of a simple algorithm of Day[3] which creates a complete binary tree from a vine with 2^m-1 nodes, for some positive integer m. The k^{th} step of this algorithm is illustrated in Figure 3. The triangles represent complete binary trees of 2^k-1 nodes, and the circles represent 2^l-1 *spine nodes,* where $l+k=m$. Each white triangle is reattached to the right side of the black spine node above and the resulting tree is attached to the left side of the white spine node below. The result is an ordered tree with $2^{l-1}-1$ spine nodes and 2^{l-1} complete binary subtrees of $2^{k-1}-1$ nodes each. We call this operation a *compression.* If compression is performed $m-1$ times it will produce a complete ordered binary tree.

When $n+1$ is not an integral power of two we alter the first step by reattaching only $n-(2^{\lfloor lg(n) \rfloor}-1)$ nodes. The result is a tree with $2^{\lfloor lg(n) \rfloor}-1$ spine nodes and $2^{\lfloor lg(n) \rfloor}$ attached subtrees with either 0 or 1 node in them. The algorithm then uses compression as before for $\lfloor lg(n) \rfloor - 1$ more steps, producing a route balanced tree regardless of which nodes are reattached in the first step.

Our basic algorithm uses the first, third, fifth, etc. nodes as the choices to reattach in the first step, producing a route balanced tree in which all of the deepest leaves are as far left as possible. This is achieved by doing a compression on an initial portion of the vine. Day's algorithm also works for vines of arbitrary length, producing trees in which the deepest leaves tend towards the right. The sole reason for our adjustment of his algorithm is to simplify the discussion for perfectly balanced trees.

To produce a perfectly balanced tree it is necessary to skip over some nodes at the first step, creating somewhat evenly spaced conceptual holes in the lowest level of the final tree. Imagine a vine with $2^{\lceil lg(n+1)\rceil}-1$ nodes. In such a vine the odd numbered nodes would be the leaves in the final perfectly balanced tree, and the even numbered nodes would form the spine after the first step. The tree created would have $l=2*2^{\lfloor lg(n)\rfloor}$ leaves. The actual tree we will build has $h=(2^{\lceil lg(n+1)\rceil}-1)-n$ holes where the imagined tree had leaves. We place the i^{th} hole at the $\lfloor i*(l/h)\rfloor$ leaf position. (Note that $l \geq h$, so different holes will be placed at different leaf positions.)

To see that the final tree will be perfectly balanced, identify the j^{th} leaf of the imagined tree with the interval $[j,j+1)$. The leaf positions associated with the left and right subtrees of any node correspond to two disjoint half-open intervals of the same length. Since the rational numbers $1*(l/h)$, $2*(l/h)$, . . . , $h*(l/h)=l$ are evenly spaced, the number of rational numbers falling into one of the half-open intervals cannot differ by more than one from the number falling into the other; consequently, the number of holes in the two subtrees cannot differ by more than one.

The algorithm for producing perfectly balanced trees is obtained from the basic algorithm by replacing the first call to compression with a call to perfect_leaves. Since perfect_leaves goes sequentially through the vine, it runs in linear time. Vine_to_tree

uses only a constant amount of extra space, and runs in linear time, regardless of which version is used, because each call to compression runs in time linear in the number of spine nodes, and at each step after the first, the number of spine nodes after compression is less than half the number before it.

```
procedure rebalance (var root: nodeptr);
        var pseudo-root: nodeptr; size: integer;

        procedure tree_to_vine (var root: nodeptr; var size: integer);
                var vine_tail,remainder,tempptr: nodeptr;
        begin {tree_to_vine}
                vine_tail:=root;
                remainder:=vine_tail↑.right:
                size:=0;
                while remainder≠nil do
                        if remainder↑.left=nil
                                then begin {move vine_tail down one}
                                        vine_tail:=remainder;
                                        remainder:=remainder↑.right;
                                        size:=size+1
                                        end {then}
                                else begin {rotate}
                                        tempptr:=remainder↑.left;
                                        remainder↑.left:=tempptr↑.right;
                                        tempptr↑.right:=remainder;
                                        remainder:=tempptr;
                                        vine_tail↑.right:=tempptr
                                        end; {else}
        end; {tree_to_vine}
```

```
procedure vine_to_tree (var root: nodeptr; size: integer);
        var temp: nodeptr; leaf_count: integer;

        procedure compression (var root: nodeptr; count: integer);
                var scanner: nodeptr; i: integer;
        begin {compression}
                scanner:=root;
                for i:=1 to count do begin
                        child:=scanner↑.right;
                        scanner↑.right:=child↑.right;
                        scanner:=scanner↑.right;
                        child↑.right:=scanner↑.left;
                        scanner↑.left:=child;
                        end; {for}
        end; {compression}
begin {vine_to_tree}
        leaf_count:=2^⌈lg(size+1)⌉ -1 -size;
        compression (root, leaf_count); {create deepest leaves}
        size:=size-leaf_count;
        while size > 1 do begin
                compression (root, size div 2);
                size:=size div 2;
                end; {while}
end; {vine_to_tree}
```

```
begin {rebalance}
new (pseudo_root);
pseudo_root↑.right:=root;
tree_to_vine (pseudo_root, size);
vine_to_tree (pseudo_root, size);
root:=pseudo_root↑.right;
dispose (pseudo_root);
end; {rebalance}
```

```
procedure perfect_leaves (var root: nodeptr; leaf_count: integer; size: integer);
    var scanner,leaf: nodeptr;
        counter, hole_count, next_hole, hole_index, leaf_positions: integer;
begin {perfect_leaves}
    leaf_positions:=2^⌈lg(size+1)⌉-1;
    hole_count:=leaf_positions-leaf_count;
    if hole_count>0 then begin
        hole_index:=1;
        next_hole:=leaf_positions div hole_count;
        scanner:=root;
        for counter:=1 to leaf_positions-1 do
            {The upper limit is leaf_positions-1, and not leaf_positions,
            because the last position is always a hole.}
            if counter=next_hole
                then begin
                    scanner:=scanner↑.right;
                    hole_index:=hole_index+1;
                    next_hole:=(hole_index*leaf_position) div hole_count;
                    end {then}
                else begin
                    leaf:=scanner↑.right;
                    scanner↑.right:=leaf↑.right;
                    scanner:=scanner↑.right;
                    scanner↑.left:=leaf;
                    leaf↑.right:=nil;
```

```
                    end; {else}

            end; {if}

end; {perfect_leaves}
```

3. SORTING

Sometimes trees are implemented by using arrays of records, where a pointer to a node is an index into the array. (For FORTRAN-style implementations, instead of an array of records one uses parallel arrays, one for each of the record's components.) In this case, one of the algorithms in Chang and Iyengar [2] provides a fringe benefit: when finished, the tree occupies the first n positions of the array, and the items are stored in sorted order. However, their algorithm requires a significant amount of extra space, as it first copies the entry array into an auxiliary array. For such an implementation, a call to a new procedure sort_vine, made between the calls to tree_to_vine and vine_to_tree, also provides a sorted array, while still using only $\Theta(n)$ time and $\Theta(1)$ space.

Sort_vine moves the vine so that its items are stored in order in positions $1 \cdots n$. We will use the following declarations:

```
const
        node_array_limit={some positive integer};
        null=0;{equivalent of nil for pointer}
type
        nodeptr=null..node_array_limit;
        node=record data: {arbitrary, and includes the key};
                left,right:nodeptr;
                end;
        node_space=array[1..node_array_limit] of node;
var
        nodes: node_space;
```

Sort_vine proceeds top-down, moving data from the vine to its desired position in the array. The i^{th} node from the vine is moved to the i^{th} position of the array by switching data parts. It may be that position i held a node of the vine, in which case some pointer still points to i. To ensure that this data can be found later, the left pointer at position i is used to point to the position to which the data has moved. (Since the vine uses only right pointers, no pointer information is destroyed.) In general, when the next vine node is to have its data moved, the pointer to it points only to its initial position. The variable "alias" is used to find the current location of the data by following left pointers until a null one has been found.

To verify that the algorithm takes $\Theta(n)$ time, we note that any left link is traversed at most once, and hence the total number of iterations of the while-loop is at most $n-1$. (It is $n-1$, and not n, because the last left link created cannot be traversed.)

```
procedure sort_vine (var root: nodeptr; size: integer);
        var next_node,next_node_alias: nodeptr;
            counter: integer;

begin {sort_vine}
        next_node:=nodes[root].right;
        for counter:=1 to size do begin
                alias:=next_node;
                while nodes[alias].left≠null do
                        alias:=nodes[alias].left;
                switch(nodes[alias].data, nodes[counter].data);
                nodes[counter].left:=alias;
                next_node:=nodes[next_node].right;
                end {for};
```

{The remainder of this procedure merely sets up the correct left and right pointers for the vine, enabling vine_to_tree to be used unaltered. If vine_to_tree is rewritten to use the fact that the nodes are now sorted in positions $1 \cdots$ size, then the remainder of this procedure can be eliminated.}

```
        root:=size +1;
        nodes[root].right:=1;
        for counter:=1 to size-1 do begin
                nodes[counter].right:=counter +1;
                nodes[counter].left:=null;
                end {for};
```

```
        nodes[size].left:=null;
        nodes[size].right:=null;
```

{Note: in actual use, there would also be some allocation procedures which obtain and release nodes, just as **new** and **dispose** are used for pointer variables. Positions size+2 · · · node_array_limit should be made available for future allocation}

end {sort_vine};

4. SUMMARY

We have presented a simple algorithm which takes an arbitrary binary search tree and transforms it into one of minimal height, using linear time and constant additional space. Previous algorithms required more time or space[2,5], or both[1], or could not be applied to arbitrary binary search trees [3]. The basic algorithm produces a tree of minimal height, which should be an optimal tree for any application. In case there is some need to produce a perfectly balanced tree, we also provide a slightly more complicated algorithm which accomplishes this, again using optimal time and space.

Finally, our last modification can be used when the nodes are stored in an array. The tree is rebalanced, and the nodes are stored in order in the initial portion of the array. This modification uses only linear time and constant space, unlike the P2 algorithm of Chang and Iyengar[2], which also sorts and rebalances in linear time, but needs a second array.

REFERENCES

1. Bentley, J.L., Multidimensional binary search trees used for associative searching, Comm. ACM 18 (1975), 509-517.

2. Chang, H. and Iyengar, S.S., Efficient algorithms to globally balance a binary search tree, Comm. ACM 27 (1984), 695-702.

3. Day, A.C., Balancing a binary tree, Computer J. 19 (1976), 360-361.

4. Knuth, D.E., The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison-Wesley, Reading, MA, 1973.

5. Martin, W.A. and Ness, D.N., Optimal binary trees grown with a sorting algorithm, Comm. ACM 15 (1972), 88-93.

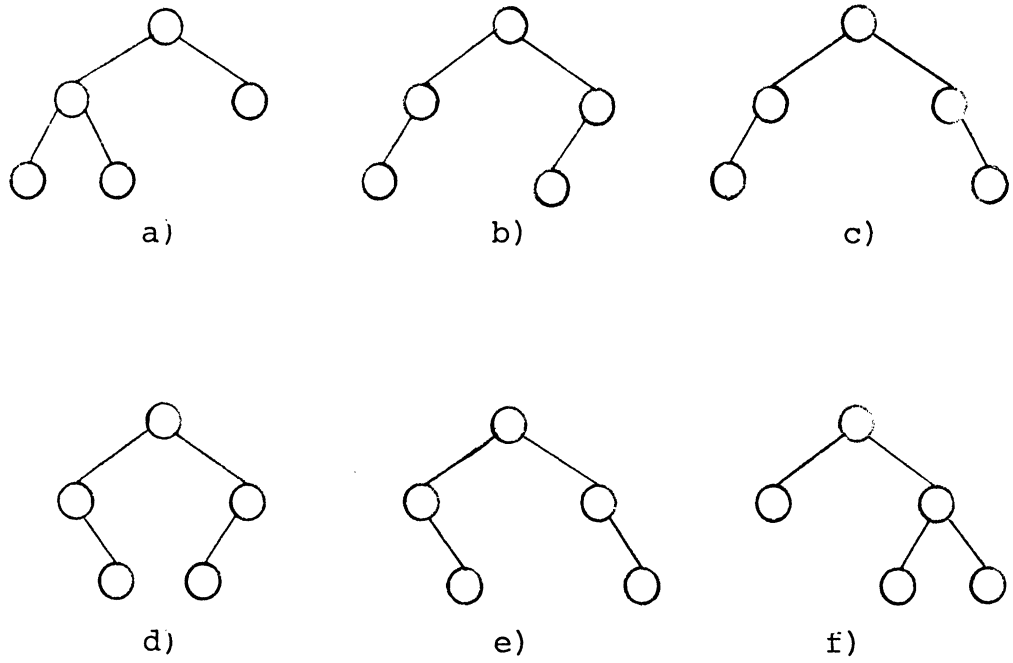

Figure 1.

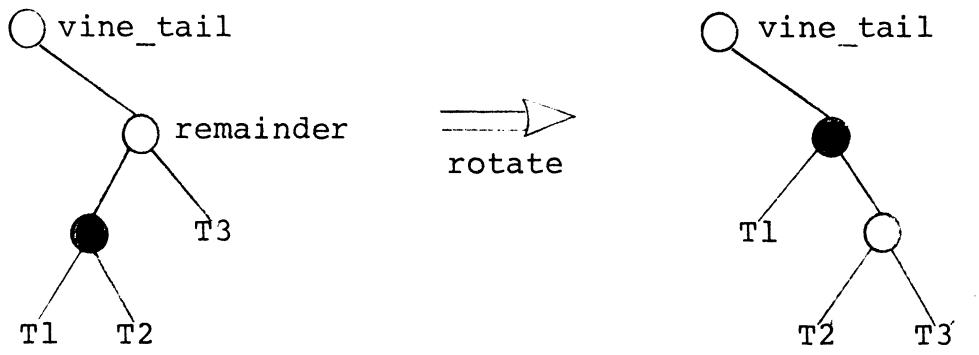

Figure 2.

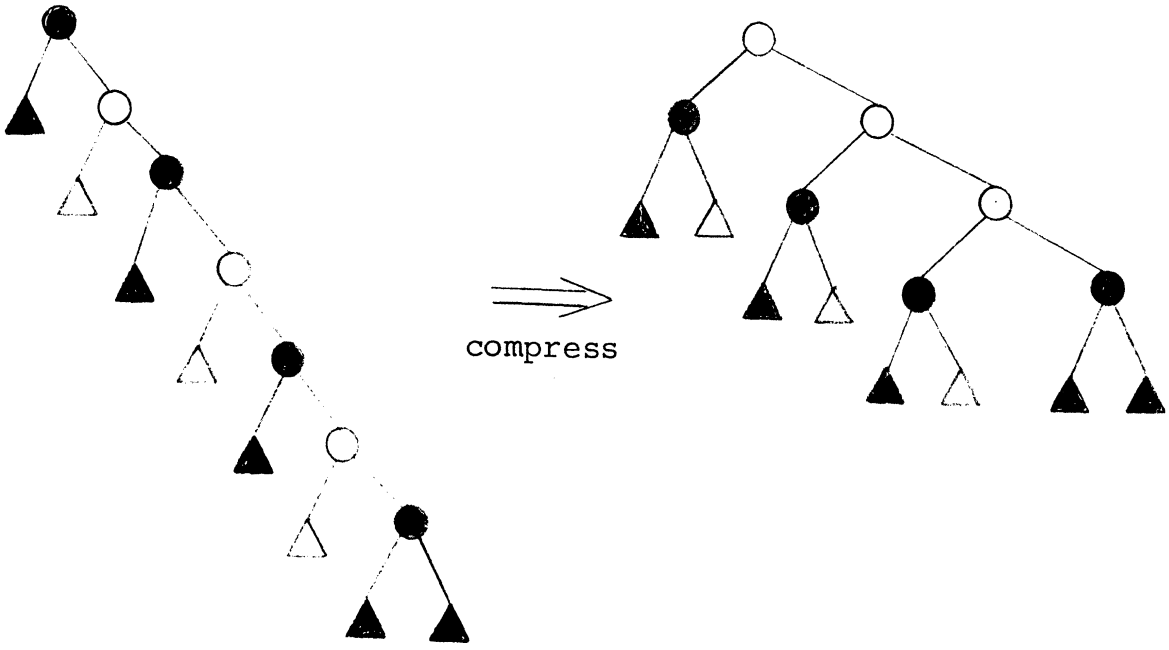

Figure 3.