Programming Techniques and Data Structures

John Bruno Editor

Tree Rebalancing in Optimal Time and Space

QUENTIN F. STOUT and BETTE L. WARREN

ABSTRACT: A simple algorithm is given which takes an arbitrary binary search tree and rebalances it to form another of optimal shape, using time linear in the number of nodes and only a constant amount of 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 this 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 an efficient and widely used structure 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 time to perform each of these operations is $\Theta(\log(n))$ [5]. Unfortunately, it is possible for a binary tree to have very long branches, and the worst-case time is $\Theta(n)$. Further, there is experimental evidence that if a tree is grown as a long intermixed sequence of random insertions and deletions, as opposed to just insertions, then the expected time is worse than logarithmic [4].

This research was partially supported by National Science Foundation grants MCS-83-01019 and DCR-8507851.

To avoid the worst-case linear time it is necessary to keep the tree balanced, that is, the tree 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. [5]. Here we are concerned with perhaps the simplest strategy; periodically rebalance the entire tree into an equivalent tree of optimal shape. This strategy has been discussed by many authors, and several algorithms have been presented [1, 3, 6]; recently Chang and Iyengar [2] surveyed this work and presented additional algorithms. No previous algorithm could rebalance an arbitrary binary search tree 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 article is a simple algorithm which accomplishes this.

One notion of "optimal shape" used in rebalancing trees is that of *perfect balance*, which requires that at each node p, the number of nodes in p's left subtree differs by no more than 1 from the number of nodes in p's right subtree. It is easy to see that in a perfectly balanced tree of n nodes the maximum depth of the nodes is $\lfloor \lg(n) \rfloor$, and for each depth $0 \leq d < \lfloor \lg(n) \rfloor$ there are exactly 2^d nodes at depth d. (lg denotes $\log_2$ and $\lfloor x \rfloor$ denotes the largest integer no larger than x. The *depth* of a node is the number of links which must be traversed in traveling from the root to the node. The depth of the root is 0, and the

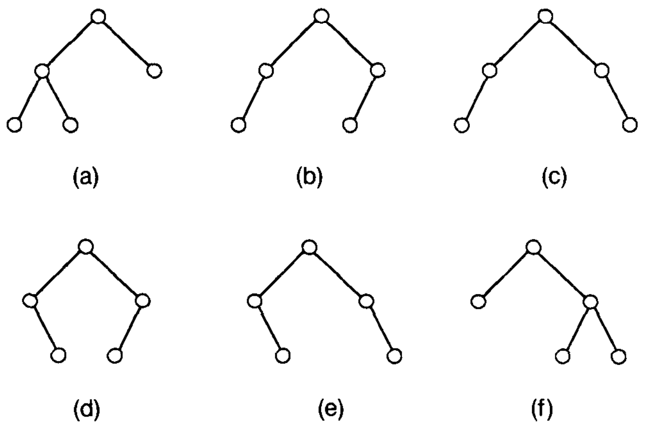

FIGURE 1. All Route-Balanced Trees of Five Nodes

children of a node of depth d have depth $d + 1$.) Using these properties, it is also easy to show that, among all binary trees of n nodes, perfectly balanced trees minimize the maximum depth of the nodes and minimize the average depth of the nodes. Therefore perfectly balanced trees have the best possible worst-case time and the best possible expected case time for each standard tree operation.

However, perfectly balanced trees are not the largest class of trees with all these properties. A binary tree with n nodes, where all nodes are at depth $\lfloor \lg(n) \rfloor$ or less, and where there are exactly 2^d nodes at depth d for each depth $0 \le d < \lfloor \lg(n) \rfloor$, will be called *route balanced.* Route balanced trees are precisely those binary trees which minimize the maximum depth of the nodes and minimize the average depth of the nodes. 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, but all six are route balanced. With the exception of Day [3], previous authors concentrated on creating perfectly balanced trees. Although perfect balancing fits naturally into a top-down approach, we know of no reason to prefer a perfectly balanced tree over a route balanced tree, and our basic algorithm creates route balanced trees. If for some reason a perfectly balanced tree is needed, then a modified version of our basic algorithm, still requiring only linear time and constant additional space, can produce it. No previous algorithm produces a perfectly balanced tree using only constant additional space.

Our algorithm proceeds in two phases. The binary tree is first transformed into a "vine" in which each parent node has only a right child and the nodes are in sorted order. The vine is then transformed into a route balanced tree. This strategy is the same as in Day [3], but he 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 is needed.)

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

Throughout, n will denote the number of nodes in the tree. The 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;
```

Although we use this standard pointer implementation of trees, our algorithms require no special properties of pointers (nor of Pascal) and can be easily 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 the algorithms, each vine will have a pseudoroot which contains no data,

Rebalance Algorithm

```
procedure rebalance(var root: nodeptr);
{rebalance the binary search tree with
 root "root↑", with the result also
 rooted at "root↑". Uses the tree_to_vine
 and vine_to_tree procedures.}
  var pseudo_root: nodeptr;
      size: integer;
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}
```

where the pseudoroot's right pointer points to the real root.

Tree_to_Vine
This 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 pseudoroot via a path of right links retains this property after the rotation. Further, after the rotation, the node that was initially pointed to by remainder↑.left is also reachable via right links. Since each rotation increases by 1 the number of nodes reachable from the pseudoroot via right links, at most $n - 1$ rotations can occur (note that the root is reachable initially). Therefore the while-loop will be executed at most $2n - 1$ times, and at least n times, so tree_to_vine runs in $\Theta(n)$ time.

Tree_to_Vine Algorithm

```
procedure tree_to_vine (root: nodeptr;
                        var size:
                        integer);
{transform the tree with pseudo-root
 "root↑" into a vine with pseudo-root
 node "root↑", and store the number of
 nodes in "size"}

  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}
```

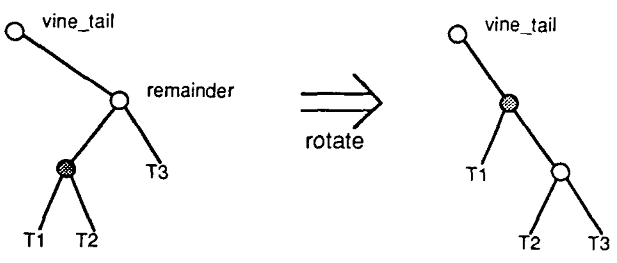

FIGURE 2. A Tree_to_Vine Rotation

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 ordered binary tree from an ordered vine with $2^m - 1$ nodes, for some positive integer m. (A *complete* binary tree is a route balanced binary tree of $2^m - 1$ nodes, for some positive integer m. Such a tree has 2^{m-1} nodes at depth $m - 1$, and is unique.) The kth step of this algorithm is illustrated in Figure 3. Each triangle represents a complete binary tree of $2^k - 1$ nodes, and each of the $2^j - 1$ circles represents a *spine node*, where $j + k = m$. 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 $2^{j-1} - 1$ spine nodes and 2^{j-1} complete subtrees of $2^{k+1} - 1$ nodes each. We call this operation a *compression*. Performing compression $m - 1$ times produces an ordered complete binary tree.

When $n + 1$ is not an integral power of 2 we alter

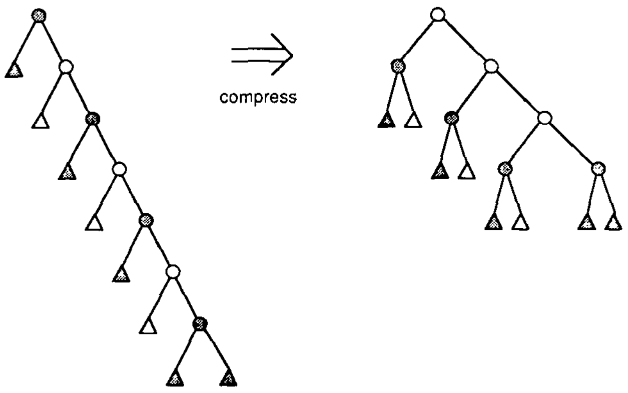

FIGURE 3. Compression

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. Compression is then performed as before $\lfloor \lg(n) \rfloor - 1$ times, producing a route balanced tree regardless of which nodes are reattached in the first step.

The 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 toward 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 in 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 complete tree, and the even numbered nodes would form the spine after the first compression. The complete tree would have $l = 2^{\lceil \lg(n+1) \rceil - 1}$ leaves. The actual tree will have $h = (2^{\lceil \lg(n+1) \rceil} - 1) - n$ holes where the conceptual tree had leaves. The ith hole with be at leaf position $\lfloor i*(l/h) \rfloor$. Note that $l \geq h$, so different holes will be at different leaf positions.

To see that the final tree will be perfectly balanced, identify the jth leaf of the imagined tree with the real interval $[j, j+1)$. The leaf positions associated with the left and right subtree of any node correspond to disjoint half-open intervals of the same length. Since the rational numbers $1*(l/h)$, $2*(l/h), \ldots, h*(l/h) = l$ 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 (p. 906). Since perfect_leaves goes sequentially through the vine, it runs in linear time. Vine_to_tree uses only a constant amount of extra

Vine_to_Tree Algorithm

```
procedure vine_to_tree (root: nodeptr; size: integer);
{convert the vine with "size" nodes and pseudo-root node "root↑" into a balanced
 tree}
  var leaf_count: integer;

  procedure compression (root: nodeptr; count: integer);
  {compress "count" spine nodes in the tree with pseudo-root "root↑"}

    var scanner, child: 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 := size + 1 - 2^⌊lg(size+1)⌋;
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}
```

Perfect_Leaves Algorithm

```
procedure perfect_leaves (root: nodeptr; leaf_count, size: integer);
{position leaves in the vine with pseudo-root "root↑" and "size" nodes so that the
 final tree will be perfectly balanced}
  var scanner, leaf: nodeptr;
      counter, hole_count, next_hole, hole_index, leaf_positions: integer;
begin {perfect_leaves}
if leaf_count > 0 then begin
  leaf_positions := $2^{\lceil lg(size+1) \rceil - 1}$;
  hole_count := leaf_positions - leaf_count;
  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_positions) 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 through for}
  end {if}
end; {perfect_leaves}
```

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.

3. SORTING

Sometimes a tree is implemented as an array 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 entire 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 linear time and constant additional space. One note of caution: since sort_vine moves the data, it cannot be used safely in a pinned structure where there are additional pointers pointing at nodes. All of the other procedures can be used in such cases because they change pointers rather than locations.

Sort_vine moves the vine so that its items are stored in positions 1 . . . n. It proceeds top-down, moving data from the vine into its desired position in the array. The ith node from the vine is moved to the ith position of the array by switching data parts. It may be that position i held some other 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 data of the next vine node is to be moved, the right pointer of the previous node points only to the data's initial position in the array. The variable "alias" is used to find the current location of the data by following left pointers until a null pointer is found. The final values of left and right

Sort_Vine Algorithm

```
procedure sort_vine (var root: nodeptr; size: integer);
{move the vine with pseudo-root "nodes[nodeptr]" into positions 1 ... size of
 "nodes", retaining the sorted order, and make "nodes[size + 1]" the new pseudo-
 root. The following declarations are assumed:
  const node_array_limit = {some positive integer ≥ n};
        null = 0; {equivalent of nil for pointers}
  type  nodeptr = null .. node_array_limit;
        node = record left, right: nodeptr;
                 data: {includes everything else, including the key}
               end;
  var   nodes: array[1 .. node_array_limit] of node;
}
  var 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 remaining code sets up the pointers so that vine_to_tree can be used
 unaltered. It can be eliminated if vine_to_tree is rewritten to use the fact that
 the items are now sorted in positions 1 ... size.}
for counter := 1 to size - 1 do begin
  nodes[counter].right := counter + 1;
  nodes[counter].left := null
  end;  {for}
nodes[size].right := null;
nodes[size].left := null;
root := size + 1;
nodes[root].right := 1
end;  {sort_vine}
{There should also be some allocation procedures to simulate the "new" and
 "dispose" procedures for pointer variables. Positions size + 2 ...
 node_array_limit should be made available for reallocation.}
```

pointers are computed and assigned in a single pass through the relevant portion of the array after all data components have been moved into their final positions.

To see that the algorithm runs in linear time, note that the number of iterations of the while-loop is equal to the total number of temporary positions (other than the initial one) occupied by the nodes with the $n - 1$ largest keys. Since two nodes are exchanged only when the one with the smaller key is being moved into its final position, this number is no greater than $n - 1$.

4. SUMMARY

We have presented a simple algorithm which takes an arbitrary binary search tree and transforms it into one which has the minimal worst and expected depths of its nodes. Aside from producing an optimal tree, our algorithm is also optimal in its use of time and space, requiring only linear time and constant additional space. Previous algorithms required more time or space [2, 6], or both [1], or could not be applied to arbitrary binary search trees [3]. The basic algorithm produces a route balanced tree, which should suffice for most applications. In case there is a need for a perfectly balanced tree, we have also provided a slightly more complicated algorithm which produces one, again using only linear time and constant additional space. This is the first algorithm which produces perfectly balanced trees using only constant additional 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 sorted order in the initial portion of the array. This modification also uses only linear time and constant additional space, unlike the P^2 algorithm of Chang and Iyengar [2], that sorts and rebalances in linear time, but requires a second array.

Acknowledgments. We would like to thank the referees for several helpful comments.

REFERENCES
1. Bentley, J.L. Multidimensional binary search trees used for associative searching. *Commun. ACM 18*, 9 (Sept. 1975), 509–517.
2. Chang, H., and Iyengar, S.S. Efficient algorithms to globally balance a binary search tree. *Commun. ACM 27*, 8 (July 1984), 695–702.
3. Day, A.C. Balancing a binary tree. *Comput. J. 19*, 4 (Nov. 1976), 360–361.
4. Eppinger, J.L. An empirical study of insertion and deletion in binary search trees. *Commun. ACM 26*, 9 (Sept. 1983), 663–669.
5. Knuth, D.E. *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison-Wesley, Reading, Mass., 1973.
6. Martin, W.A., and Ness, D.N. Optimal binary trees grown with a sorting algorithm. *Commun. ACM 15*, 2 (Feb. 1972), 88–93.

CR Categories and Subject Descriptors: E.1 [**Data**]: Data Structures—*trees*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*sorting and searching*
General Terms: Algorithms
Additional Key Words and Phrases: optimal search, perfect balance, rebalancing

Received 10/84; revised 2/86; accepted 6/86

Authors' Present Addresses: Quentin F. Stout, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109. Bette L. Warren, Department of Mathematics, Eastern Michigan University, Ypsilanti, MI 48197.