Weight-balanced trees

by J. L. BAER
University of Washington
Seattle, Washington

INTRODUCTION

It is now recognized that binary search trees are structures which can be used efficiently for the organization of files and directories. The ease of insertion and deletion of nodes makes trees very appealing for directories which are often modified. By comparison with a sequential table organization, some additional memory is required for the links between nodes. From a cost-effective viewpoint, this is generally more than compensated for by the savings in searching (for a linear table) and inserting (for an ordered table).

A number of studies have been conducted in order to build trees with as small an average searching time as possible while keeping to a reasonable degree the amount of computation and of extra memory needed for the insertion algorithms. Depending on the type of application, these studies have fallen into two categories: those concerned with trees where the nodes have a uniform weight as exemplified by AVL trees,4,6 binary B-trees,9 and BB or bounded balanced trees;6 and those which consider weighted trees, the weight being for example a frequency of query.4,6,8

Our primary intention was to concentrate exclusively on weighted trees for the following reasons. First, they are more representative of common directories; second, dynamic self-optimization of weighted trees has not been considered yet; and finally, it was felt that the techniques used in non-weighted trees could be easily modified to be applied efficiently to weighted trees. Hence, we present two closely related algorithms, called weight-balanced (WB), for constructing dynamically self-optimizing binary search trees. Results of simulation experiments show that near-optimal trees are obtained.

But, if the algorithms are applied to non-weighted trees, we obtain results in the same range as those yielded by AVL and BB algorithms. This is not surprising since WB algorithms produce BB(1/4) trees in the worst case. A discussion of the relative advantages and disadvantages of the various methods is then in order. It will show the flexibility of the WB approach.

DEFINITIONS AND RELATED WORK

A binary tree is either the empty tree T₀ or the triple (T_L, r, T_R) where T_L and T_R are the left and right binary subtrees and r is a special node called the root. Given a node P in the tree and the subtree (T_L, P, T_R), its left son is the root of T_L and its right son the root of T_R. Conversely, each node (but the root) has a father for which it is a son. A node without a son is a leaf. Each node in the tree contains a key. A binary tree is a binary search tree (abbreviated b.s.t.) if for all nodes P, the keys of the left (right) subtree are less (greater) than the key of P. In a weighted binary search tree (w.b.s.t.) each node has a weight associated with it. The level of a node P in a tree is one if P is the root, or else the level of its father plus one. The height of a node is one if it is a leaf, or else the maximum height of its sons plus one.

In addition, we define the total of a node as its weight if it is a leaf, or otherwise as the sum of the total(s) of its son(s) and its own weight.

The weighted path length of a w.b.s.t. T of n nodes is

\[ \text{Path}(T) = \sum_{i=1}^{n} \text{level}(i) \cdot \text{weight}(i) \]

In terms of computer representation, each node P will contain four fields; namely LLINK(P), INFO(P), TOTAL(P), and RLINK(P). LLINK(P) points to the left son of P or equals λ if there is no left son and likewise for RLINK(P); INFO(P) is the key, and TOTAL(P) is as defined previously.

Given a set of n keys and associated weights, there exists an algorithm which produces the optimal w.b.s.t. that is the tree with minimal path length. However, this algorithm presents two disadvantages:

1. It requires a processing time of the order \(O(n^2)\) and, most importantly, additional memory requirements of \(O(n^2)\) units. Thus it becomes rapidly impractical when the number of nodes increases.
2. The weights of all nodes must be known in advance.

Circumventing these difficulties is possible if one is ready to settle for near-optimal trees instead of complete optimality. Heuristic methods, approaching the optimal solution within 2 or 3 percent, have been devised. The algorithms run in time between \(O(n \log n)\) and \(O(n^2)\) depending on the weight distribution and require only \(O(n)\) memory units. Yet, the second assumption above is still enforced.

We introduce next new algorithms which will yield also near-optimal trees with a similar requirement of \(O(n)\) mem-

---

*This work was supported by NSF Grant GJ-41164.

467

From the collection of the Computer History Museum (www.computerhistory.org)
ory units. The main advantage of our scheme is that assumption (2) above is no longer necessary. That is, the algorithms work on evolving structures corresponding to dynamic queries in a directory and "balance" the tree accordingly. In the case where all weights are known in advance, its running time is of the same order as those of the already mentioned algorithms for comparable results. This is not surprising since many heuristics lead to near-optimal trees.5

The weight-balanced algorithms that we introduce are adaptations of techniques used in the AVL and BB trees to the case of w.b.s.t.'s. Recall that an AVL tree is a (non-weighted) tree such that the height of the left son of any node never differs by more than one from the height of the right son. The algorithm to build an AVL tree consists of a top-down search to find the place of insertion followed by a bottom-up traversal of the search path with appropriate "balancing" of the tree. The data structure used is similar to the one described above with the only difference being that the TOTAL field is replaced by a smaller (2 bits only) BALANCE indicator.

BB trees of balance α, are such that at every node P:

$$\alpha \leq \frac{\text{TOTAL}(\text{LLINK}(P)) + 1}{\text{TOTAL}(P) + 1} \leq 1 - \alpha, \quad 0 < \alpha \leq \frac{1}{2}$$

A pure top-down algorithm to insert an item in a BB(α) tree for any α in the range $0 \leq \alpha \leq 1 - \sqrt{2}/2$ is known.4 As in the AVL trees, the balancings are of the two types shown in Figure 1. The data structure is the same as the one already presented.

SELF-OPTIMIZING WEIGHTED BINARY SEARCH TREES

Self-optimizing weighted structures have been investigated only in the case of linear lists without frequency counters under the assumption that the extra cost incurred by the space reserved for the counters is not effective. Two heuristics have been tested with success: "move up front", which moves the queried entry at the head of the list, and "transposition", which moves it up one notch unconditionally. The latter performs always as well as the former, and in addition, can also be easily implemented with a sequential allocation technique.

In the case of b.s.t.'s, dynamic optimality is impractical for the reasons mentioned previously. Direct extensions of the above heuristics imply restructuring of the tree at every insertion with no assurance of good results. In fact, one can construct realistic cases giving a worst case behavior. Therefore we need a more selective criterion for balancing.

The heuristic that we propose has for its goal to minimize locally the weighted path length on the path of insertion. More specifically, turning our attention to Figure 1, with $p, q, r$ and $s$ being the TOTAL's of the respective sons of A, B and C:

A single rotation will be performed if (Figure 1a):

$$\text{weight}(A) + p > \text{weight}(B) + r$$

(or $\text{TOTAL}(A) - q > \text{TOTAL}(B) - \text{TOTAL}(A)$),

thus minimizing locally the path length of the subtree of new root A.

A double rotation will be performed, after checking that a single rotation did not apply, if (Figure 1b):

$$2 \cdot \text{weight}(B) + p + q > \text{weight}(C) + s$$

(or $2 \cdot \text{TOTAL}(B) - p - q > \text{TOTAL}(C) - \text{TOTAL}(A)$)
thus minimizing again locally the path length of the subtree of new root B. Hence we define as weight-balanced trees (WB trees) those trees obtained dynamically by the application of the above balancing criterion.

The algorithms that we introduce are modelled after the bottom-up AVL construction on one hand and after the top-down BB algorithm on the other hand. (Computer programs can be obtained from the author.) Both can be used with incremental weights corresponding to individual queries or with total weights (or fractions of them) for the periodic rebalancing of directories.

Intuitively it appears that the bottom-up algorithm having more information about weights on the insertion path should perform slightly better. For example, if the node A of weight 3 were introduced at the left of B in the tree of Figure 2a, then the bottom-up algorithm would yield the tree of Figure 2c, and the top-down algorithm the tree of Figure 2d resulting in an advantage of one unit for the former. However, as seen later, the performance of the bottom-up algorithm is only marginally better. Yet, it should not be discarded despite its extra space or pointer manipulation requirement since it can sometimes be used more conveniently as seen in the next section.

Analytical results on w.b.s.t.'s are scarce. While using heuristics care must be taken in their development since some will tend asymptotically to construct random trees (e.g., insert in order of decreasing weights). The difference between random trees and optimal trees being on the order of 40 percent on the average, one can understand why a category of heuristics yielding "balanced" trees is needed. By examining the results of some simulation experiments, we can safely assume that the one we propose falls into that latter class.

### Simulation results

In order to test our hypothesis we performed a series of experiments on the following three trees:

- Tree 1: 31 nodes; total queries~85000.
- Tree 2: 35 nodes; total queries~1300.
- Tree 3: 27 leaves carried the information (i.e., a tree of 53 nodes); total queries = 1000.

The results of the simulation are shown in Table I for the bottom-up algorithm and Table II for the top-down. Ten sample runs were used for lines 2, 5, 6 and 7. Lines 1 through 4 show the average search times in the optimal (1) and random (2) cases, with insertion of keys (total weight at once) in decreasing weight order without balancing (3), and with balancing (4). In lines 5 to 7 each query is treated individually and is selected randomly assuming a uniform distribution on the remaining queries. Balancing is performed on either all queries (5), the first 10 percent (6) or the first 1 percent (7).

From this series of experiments we observe that:

1. The bottom-up algorithm performs slightly better than the top-down.
2. The random trees are (on the average) between 17 and 43 percent worse than the optimal, and the "decreasing weight" heuristic is not good, thus confirming the theoretical analysis.
3. The "decreasing weight with balancing" performs extremely well (within 2 percent of optimality).
4. The self-optimizing feature is very efficient. Near-optimal trees are obtained with a small number of rebalancings. Under our assumption of uniform distribution of queries, this is quite encouraging, since it

### Table I—Bottom-up Algorithm on Weighted Trees.

<table>
<thead>
<tr>
<th></th>
<th>Tree 1</th>
<th>Tree 2</th>
<th>Tree 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Optimal</td>
<td>3.437</td>
<td>2.992</td>
<td>5.200</td>
</tr>
<tr>
<td>2. Random</td>
<td>4.804 (43%)</td>
<td>3.820 (28%)</td>
<td>6.074 (17%)</td>
</tr>
<tr>
<td>3. Decreasing weights</td>
<td>4.042 (18%)</td>
<td>3.142 (8%)</td>
<td>5.726 (10%)</td>
</tr>
<tr>
<td>4. Decreasing weights</td>
<td>3.437 (--)</td>
<td>2.992 (--)</td>
<td>5.212 (2.2%)</td>
</tr>
<tr>
<td>5. Self-optimizing</td>
<td>3.523 (2.2%)</td>
<td>3.907 (5.5%)</td>
<td>5.241 (8.8%)</td>
</tr>
<tr>
<td>100%</td>
<td>88 rotations</td>
<td>48 rotations</td>
<td>31 rotations</td>
</tr>
<tr>
<td>6. Self-optimizing</td>
<td>3.526 (2.4%)</td>
<td>3.059 (2.2%)</td>
<td>5.314 (2.2%)</td>
</tr>
<tr>
<td>10%</td>
<td>61 rotations</td>
<td>21 rotations</td>
<td>20 rotations</td>
</tr>
<tr>
<td>7. Self-optimizing</td>
<td>3.550 (3.3%)</td>
<td>55 rotations</td>
<td>--</td>
</tr>
</tbody>
</table>

### Table II—Top-down Algorithm on Weighted Trees.

<table>
<thead>
<tr>
<th></th>
<th>Tree 1</th>
<th>Tree 2</th>
<th>Tree 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Optimal</td>
<td>3.437</td>
<td>2.992</td>
<td>5.200</td>
</tr>
<tr>
<td>2. Random</td>
<td>4.804 (43%)</td>
<td>3.820 (28%)</td>
<td>6.074 (17%)</td>
</tr>
<tr>
<td>3. Decreasing weights</td>
<td>4.042 (18%)</td>
<td>3.142 (8%)</td>
<td>5.726 (10%)</td>
</tr>
<tr>
<td>4. Decreasing weights</td>
<td>3.502 (2%)</td>
<td>2.992 (--)</td>
<td>5.233 (7%)</td>
</tr>
<tr>
<td>5. Self-optimizing</td>
<td>3.563 (3.6%)</td>
<td>3.060 (2%)</td>
<td>5.265 (1.2%)</td>
</tr>
<tr>
<td>100%</td>
<td>35 rotations</td>
<td>27 rotations</td>
<td>26 rotations</td>
</tr>
<tr>
<td>6. Self-optimizing</td>
<td>3.569 (3.8%)</td>
<td>3.110 (4%)</td>
<td>5.337 (2.6%)</td>
</tr>
<tr>
<td>10%</td>
<td>29 rotations</td>
<td>17 rotations</td>
<td>19 rotations</td>
</tr>
<tr>
<td>7. Self-optimizing</td>
<td>3.575 (4%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1%</td>
<td>23 rotations</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>
implies that balancing could be performed only at the
creation of the directory on each individual query and
then periodically later on.

These simulation results show that WB trees lead to effi­
cient heuristics, yielding near optimal trees. The amount
of space needed is minimal and the extra computation time not
significant if one limits the restructuring to the initial creation
time of the directory.

BALANCING NON-WEIGHTED TREES

The algorithms presented in the previous section were not
intended to be used for non-weighted trees. Yet, when applied
to those structures they yield results quite comparable in
efficiency and flexibility with those obtained through the
AVL and BB techniques.

Worst case behavior

It is known that Fibonacci trees are the worst case of
AVL trees, i.e., the highest level is bounded by 1.44log2
(n+1) for a tree of n nodes. In the case of BB (a) trees, it has been shown1 that the worst case is (log2(n+1)−1)/
log2(1/(1−a)).

In order to obtain a bound on the worst case for WB trees,
we show that they are BB(1/4). The following lemma is true
for BB trees constructed either by the top-down or bottom-up algorithms. The proofs assume a bottom-up al­
gorithm. The method is to show that some trees cannot arise
when the BB algorithms are executed. Because of space
limitations we show only part of the proof in detail.

Lemma: In a WB tree each node P is such that
\[ |T_L^p| \leq 3 \] and \[ |T_R^p| + 1, \] and \[ |T_L^p| \leq 3 \] \[ |T_R^p| + 1. \]

Proof: (By induction on the number of nodes n in the tree.)

The lemma is true for n=1, 2, 3 as can be easily seen by
construction. Assume it is true for n (hypothesis H1). Let us
insert a node in the tree and show that the BB algorithm
results in a tree with H1 verified for n+1. We only have to
check on the path of insertion that the lemma’s property is
conserved. Let k’ be the height of the node C on the path of
insertion. The lemma’s property is still true for k’=1, 2 as
can be shown directly by construction. Assume that it is true
for k’=k−1 (H2). We show that it is still true for k’=k.

We distinguish 3 cases:

A. No rotation

Before insertion we had

The only difficulty is in showing that \( p+q+1 \leq 3r+1, \) or

B. Single rotation

Before insertion we had

The (easy) proof stems from H1 and a possible contradiction
on H2.

C. Double rotation

Before insertion we had

The difficult part of the proof is to show that \( s \leq 3p+1 \) (or
\( r \leq 2q+1 \). We use the same contradictory approach in as­
suming \( s = 3p+2 \). Then before the rotation we had:

either

\[
\begin{align*}
3p+2 & \\
p-1 & \\
2p+2 &
\end{align*}
\]

(next insertion on left of B1)

or

\[
\begin{align*}
3p+2 & \\
p & \\
p+1 & \quad \text{(next insertion on right of B2)}
\end{align*}
\]
TABLE III—AVL, BB and WB Algorithms

<table>
<thead>
<tr>
<th># of nodes</th>
<th>AVL</th>
<th>BB [1 - √3/2]</th>
<th>WB (top down)</th>
<th>WB (bottom up)</th>
<th>Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>8.98</td>
<td>9.207 (.46)</td>
<td>9.161 (.51)</td>
<td>9.155 (.54)</td>
<td>11.981</td>
</tr>
<tr>
<td>750</td>
<td>8.66</td>
<td>8.790 (.47)</td>
<td>8.740 (.51)</td>
<td>8.741 (.52)</td>
<td>11.157</td>
</tr>
<tr>
<td>500</td>
<td>7.99</td>
<td>8.193 (.47)</td>
<td>8.157 (.51)</td>
<td>8.150 (.52)</td>
<td>10.665</td>
</tr>
</tbody>
</table>

We show that obtention of the subtrees of root Bl is not possible. Evidently this is true for \( p=1 \). Assume it is not possible for \( p-1 \) (hypothesis H3). We prove then that it is not possible for \( p \). We detail the proof only for the first subtree.

The last insertion to obtain the subtree of root Bl cannot come from the right of C since we would have had either

\[
\begin{align*}
\text{or } & \quad B1 \\
\text{or } & \quad B2
\end{align*}
\]

(forbidden by H1)

Evidently H3 forbids that the latest insertion comes from the left of B1. The only two situations left open are:

\[
\begin{align*}
\text{or } & \quad C \\
\text{or } & \quad B1
\end{align*}
\]

This construction can be carried on further, but we can see that the \( p+2 \) keys \( K \) such that \( \text{INFO}(B1) < K < \text{INFO}(B2) \) are never introduced. Therefore, either of H1 or H3 will soon be contradicted, thus showing the impossibility that \( s = 3p+2 \).

Theorem: The maximum level of a WB tree is \( 0(\log_2 n) \)

where \( n \) is the number of nodes in the tree.

Proof: The lemma shows that

\[
\frac{1}{4} \leq \frac{\text{TOTAL}(\text{LLINK}(P)) + 1}{\text{TOTAL}(P) + 1} \leq \frac{3}{4}
\]

i.e., WB trees are BB(1/4); the highest level is hence \( \sim 2.32 \) \((\log_2(n+1) - 1)\), or \( 0(\log_2 n) \).

Average searching time

As for all b.s.t.'s the average searching time in a WB tree is \( 0(\log_2 n) \). To evaluate the efficiency of the WB algorithms, we performed the following experiment. Sets of 500, 750 and 1000 different keys were generated randomly and we applied the AVL, BB(I-V2/2) and WB algorithms for their insertions in a binary search tree. The averages of 25 sample runs are shown in Table III. As can be seen, the WB trees are slightly more performant (.5 percent to 1 percent), but certainly not significantly. The average number of rotations/insertion is also indicated (figures in parentheses). The WB algorithms require between 5 percent and 10 percent more rotations than the other two methods.

This larger number of rotations is compensated by the following advantages. First, in comparing with the BB algorithm, there is no need for division which is a time consuming operation. Second, in comparing with the AVL technique, we can use a top-down algorithm so that we do not have to go back up the path. Finally, the fact that we can choose between two algorithms allows us to face easily

TABLE IV—AVL, BB and WB Flexibility (1000 nodes)

<table>
<thead>
<tr>
<th>( s )</th>
<th>AVL</th>
<th>BB</th>
<th>WB top down</th>
<th>WB bottom up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9.207 (.46)</td>
<td>9.267 (.40)</td>
<td>9.161 (.51)</td>
<td>9.155 (.54)</td>
</tr>
<tr>
<td>3</td>
<td>9.665 (.12)</td>
<td>9.765 (.16)</td>
<td>9.344 (.18)</td>
<td>9.333 (.18)</td>
</tr>
<tr>
<td>4</td>
<td>9.925 (.07)</td>
<td>9.427 (.13)</td>
<td>9.422 (.13)</td>
<td>9.499 (.11)</td>
</tr>
<tr>
<td>5</td>
<td>10.23 (.05)</td>
<td>9.505 (.11)</td>
<td>9.505 (.11)</td>
<td>9.505 (.11)</td>
</tr>
</tbody>
</table>
the contradictory situations: the key to be inserted is certainly not present, hence use a top-down algorithm; and, the key to be inserted might be present, hence use a bottom-up algorithm and cancel the balancing if the key were present.

Space required

The amount of space required is of the same order for the three algorithms. As observed in Reference 4, the little savings in AVL trees is compensated by the need of the TOTAL field (or equivalent) if one wants to use the structure efficiently as a linear list.

Flexibility of the algorithm

It is interesting to see how the algorithms behave if one relaxes the balancing criteria in order to reduce the number of rotations. In terms of AVL trees this is akin to letting the difference in heights be some \( \delta \) greater than 1 and for BB trees to having \( \alpha \) be less than \( 1-v/2 \). For WB trees this means that we will have a single rotation if

\[
\text{weight}(A)+p>\text{weight}(B)+r+\gamma \quad \text{(Figure 1a)}
\]

and a double rotation if

\[
2\cdot\text{weight}(B)+p+q>\text{weight}(C)+s+\gamma \quad \text{(Figure 1b)}
\]

The above sets of 1000 keys were used again to construct AVL trees with \( \delta \) varying from 1 to 5, BB trees with \( \alpha \) being \( 1-v/2, 0.25 \) and 0.20, and WB trees with \( \gamma \) varying from 0 to 4. As before, the number of rotations was monitored. The results summarized in Table IV show that WB algorithms provide a fine tuning of the average search time with decreases in rotations (proportionally) as great as for AVL trees.

At this time it does not seem possible to rate the “goodness” of the different algorithms since there is no evident connection between the parameters \( \alpha, \gamma \) and \( \delta \). The only fair assessment that one can make relative to WB trees is that both the worst case and average searching times (for trees of approximately \( 10^9 \) nodes) are of the same order of magnitude as those obtained by using AVL and BB trees. Furthermore, the parameter \( \delta \) can be used very efficiently for tuning the algorithm. Mainly, it should be emphasized that since the WB technique can also be applied to weighted trees and since there is a choice between a top-down and a bottom-up algorithm, WB trees provide a very flexible tool.

CONCLUSION

In this paper we have presented a new technique for balancing search trees in order to minimize the average searching time of a key in the tree. The technique is based on the concept of weight balance and is applicable to both weighted and non-weighted trees. In the former case, near-optimal trees can be obtained dynamically in a self-optimizing fashion. In the latter, the technique gives results of quality comparable to those obtained by AVL or BB methods.

It would be interesting to test further the technique in different environments, as for example paging systems, multiprocessing and in connection with B-trees.

REFERENCES