A TRANSFORMATION FOR ADDING RANGE RESTRICTION CAPABILITY TO DYNAMIC DATA STRUCTURES FOR DECOMPOSABLE SEARCHING PROBLEMS

by

George S. Lueker[+]

Department of Information and Computer Science
University of California, Irvine
Irvine, California 92717

Technical Report #129
February 1979

Keywords and phrases:
Range queries
Decomposable searching problems
File organization
Trees of bounded balance
Data base management
Data structures
Searching
Empirical cumulative distribution function
Maxima searching

CR categories:
3.73,3.74,3.8,4.33,
4.34,5.25

[+]Partially supported by NSF Grant MCS77-04410

Abstract

A data base is said to allow range restrictions if we may request that only records with some specified field in a specified range be considered when answering a given query. We present a transformation which enables range restrictions to be added to an arbitrary dynamic data structure on n elements, provided that the problem satisfies a certain decomposability condition, and that we are willing to allow increases of a factor of log n in the worst-case time for an operation and in the space used. This is a generalization of a known transformation which works for static structures. We then use this transformation to produce a data structure for range queries in k dimensions with worst-case times of $O(\log^k n)$ for each insertion, deletion, or query operation.

(Similar results were achieved independently by Dan Willard. See the remarks at the end of section 1.)

1. Decomposable Problems and Transformations

We begin by reviewing some results of Bentley and Saxe [B78]. A searching problem may be viewed as a set F of records on which we allow queries to be performed. Suppose there exists a function f and a commutative associative operator $\square$ with identity, such that the response to a query q may be written as $\bigsquare_{x \in F} f(q,x)$, where $\bigsquare_{x \in F}$ denotes repeated application of $\square$. Then this searching problem is said to be <u>decomposable</u>. We will always assume that f and $\square$ are computable in O(1) time. See [B78] for numerous examples of such problems. Henceforth, we only consider decomposable searching problems.

An interesting special case of a decomposable searching problem is the following [B78]. Suppose each element x of F has a k-tuple called KEY(x) and some field VALUE(x). Suppose that each query q is a k-tuple of intervals, say

$$q = \left([\ell_1, u_1], [\ell_2, u_2], \ldots, [\ell_k, u_k] \right).$$

Finally, suppose

$$f(q,x) = \begin{cases} \text{VALUE}(x) & \text{if KEY}(x) \text{ is} \\ & \text{in } [\ell_1, u_1] \times [\ell_2, u_2] \times \ldots \times [\ell_k, u_k] \\ \text{the identity} & \\ \text{for } \square & \text{otherwise} \end{cases}$$

Then this problem is called a k-dimensional orthogonal range query, or simply a range query. An example of such a problem is the determination of the number of points lying in some k-dimensional box. See [K73 pp.554-555, BS77, B78, BF78].

Decomposable searching problems have some very nice properties which have been investigated by Bentley and Saxe, among others; some of their results are reported in [B78]. In particular, several transformation schemes have been discovered which enable one to modify the characteristics of a data structure for a decomposable problem [B78]. Let U(n) and Q(n) denote the worst case time for a single update (i.e., insertion or deletion) or query operation, respectively, when n elements are present. Sometimes an individual update may take a long time to perform but we can guarantee the average time over all updates; let $\overline{U}(n)$ denote this average, where n is the maximum number of elements present at any time. So far we have considered dynamic data structures; some data structures are static in the sense that they require all records to be presented initially and preprocessed before any queries take place. In this case we let P(n) denote the total preprocessing time, where n is the number of records presented. In either the static or dynamic case, let S(n) be the amount of storage used; assume S has at least a linear growth rate.

Now we consider some examples of transformations from [B78]. One example is the addition of a range specification. For example, suppose we have a data structure containing information about recent research, which enables us to determine quickly the total number

of papers written on a given topic. We might wish to convert this to a data structure which allows us to specify a range of dates; then we could ask, for example, how many papers on data bases were written between 1965 and 1970.

Theorem [B78]. Suppose we are given a static data structure, with complexity measures P, Q, and S as above. Then we may produce a corresponding static data structure with range specification, whose complexities are described by the primed functions below.

$$P'(n) = O(P(n) \log n)$$
$$Q'(n) = O(Q(n) \log n)$$
$$S'(n) = O(S(n) \log n)$$

In [BS79] an Alphard form which implements this transformation is presented and formally proved correct.

Another example is the construction of a dynamic structure from a static one.

Theorem [B78]. If a static data structure has complexities described by P, Q, and S as above, we may produce a corresponding dynamic data structure (with insertions and queries but without deletions) with complexities

$$\overline{U}'(n) = O(P(n) \log n/n)$$
$$Q'(n) = O(Q(n) \log n)$$
$$S'(n) = O(S(n))$$

Deletions can also be processed in the dynamic structure, with the same time bound, provided that the operator $\square$ admits inverses [Bp]. We merely maintain two data structures, one contain-

ing all inserted records, and one containing all deleted records. To respond to a query, we use the "difference" under $\square$ of the responses of each data structure.

Note that production of a data structure for range queries in k dimensions is now easy. By the first theorem we immediately can construct a static structure with [B78]

$$P(n) = O(n \log^k n)$$
$$Q(n) = O(\log^k n)$$
$$S(n) = O(n \log^{k-1} n)$$

By a trick called presorting, [B78] shows how the preprocessing time may be reduced to

$$P(n) = O(n \log^{k-1} n).$$

(This result was obtained in the special case of counting keys in the desired range in [BS77].) If dynamic structure is desired, the second theorem now enables us to immediately produce one with

$$\overline{U}(n) = O(\log^k n)$$
$$Q(n) = O(\log^{k+1} n)$$
$$S(n) = O(n \log^{k-1} n).$$

In [L78] an ad hoc construction is produced which gives time bounds of

$$\bar{U}(n) = O(\log^k n)$$
$$Q(n) = O(\log^k n)$$
$$S(n) = O(n \log^{k-1} n)$$

In addition to having a slightly faster Q, this data structure enables deletions to be handled quickly even when inverses for $\square$ do not exist.

If the operations to be performed are known to be predominantly updates or predominantly queries, other data structures may be preferable. For example, in [BM78], for any $\epsilon>0$, static data structures with

$$P(n) = O(n^{1+\epsilon})$$
$$Q(n) = O(\log n)$$
$$S(n) = O(n^{1+\epsilon})$$

or with

$$P(n) = O(n \log n)$$
$$Q(n) = O(n^{\epsilon})$$
$$S(n) = O(n)$$

are described, where ϵ is any positive constant. See [BF78] for a survey of a wide variety of approaches.

Lee and Wong [LW78] have also described a structure which enables range queries to be performed in $O(\log^k n)$ time; as in [L78], they employ weight-balanced concepts. In addition, they describe efficient algorithms for exact match queries, partial match queries, and partial region queries based on their structure. Their data structure is primarily a static one, however; they do not show how the balance condition may be maintained as insertions and deletions are performed.

In this paper we describe a transformation which adds range restriction capabilities and can be applied directly to dynamic structures. It adds a factor of log n to the processing time for updates and queries, and to the space. It thus immediately leads to a data structure for range queries with time bounds of

$$U(n) = O(\log^k n)$$
$$Q(n) = O(\log^k n)$$
$$S(n) = O(n \log^{k-1} n).$$

This gives the same average complexities as in [L78], but additionally guarantees worst-case times per operation. Unlike [L78], the approach in this paper is based on transformations like those used in [B78], and hence immediately generalizes to data structures for any decomposable problem.

Dan Willard [W78], working independently, achieved results very similar to those in this paper and in [L78]. We both first publicly claimed these results in 1978; he did so several months before I did. While we both based our approach on bounded balance trees, the final algorithms we produced are somewhat different; the algorithm to be described here appears to be significantly easier. More recently, Willard has obtained some further results; for example, he has shown how to improve the worst-case query time [Wp].

In section 2 we discuss a simplified version of the transformation which gives good total update times but does not guarantee worst-case times for each individual update. In section 3 we modify this transformation so as to guarantee worst-case times for each operation. Section 4 mentions some applications.

2. A Transformation with Good Total Update Times

As in [L78], we will use a bounded balance tree scheme [NR73]. The argument to be used to prove the more powerful results presented here will be substantially more delicate, however. We begin by reviewing some fundamental facts about bounded balance trees.

Let the rank of a node x, written rank(x), be one more than the number of nodes which descend from x. (We will always consider a node to be a descendant of itself.) The balance of a node x, written $\rho(x)$, is the ratio of the rank of the left child of x to rank(x). A node x is α-balanced if $\rho(x) \in [\alpha, 1-\alpha]$. In [NR73] it is shown that if a tree on n nodes is α-balanced for some positive α, then the height of the tree is O(log n). It is also shown how (assuming $\alpha < 1-\sqrt{2}/2$) balance in a tree may be maintained through the use of some simple rebalancing operations, so that insertions and deletions can be done in O(log n) worst-case time per operation. The rebalancing operations are illustrated in Figure 1. Note that in a single rotation, the node x remains the root of the subtree. However, in order to make sure that keys are correctly ordered, the data records corresponding to x and y must be interchanged. Similar remarks apply to double rotation. To rebalance a node x, use the following procedure. For now we will leave the parameter γ unspecified; its choice will be explained later.

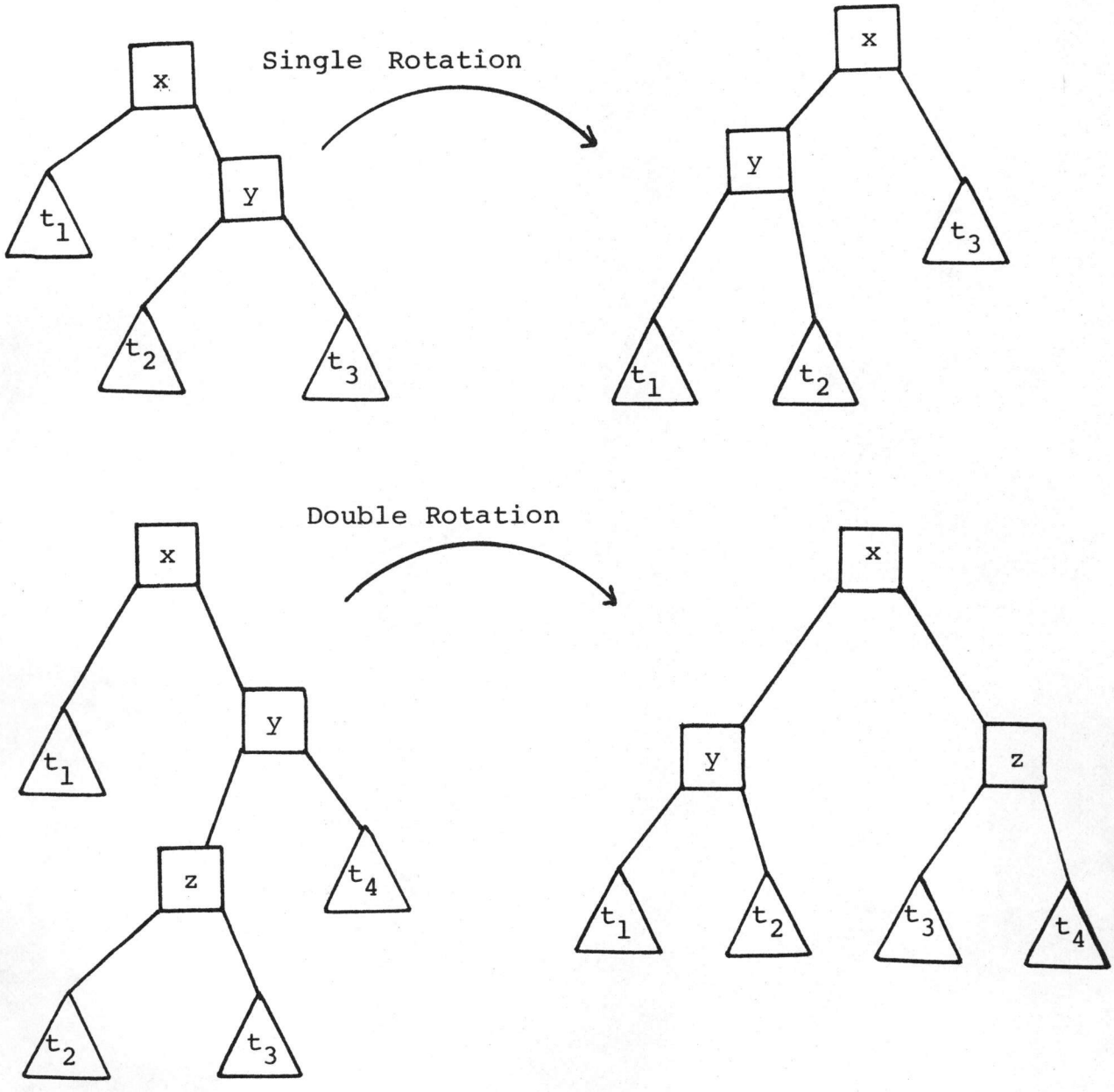

Figure 1. Rebalancing operations for trees of bounded balance [NR73].

```
procedure BALANCE(x);
  begin comment assuming ρ(x) ∉ [α,1-α],
     modify the tree so that ρ(x) ∈ [α,1-α];
   without loss of generality assume ρ(x)<α;
   let y be the right child of x;
   if ρ(y)<γ
       then perform the single rotation of Figure 1
       else perform the double rotation of Figure 1
  end;
```

We will say that nodes x, y, and z of Figure 1 actively participate in the rebalancing. From now on, whenever we use the term "binary search tree" or "bounded balance tree," we will assume that records are stored at leaves only; moreover, each internal node has precisely two children. It is easy to modify the algorithms for binary search trees or bounded balance trees to reflect these changes; we need to maintain a field at each internal node x, telling the largest leaf which descends from x, to guide searches in the tree.

Now assume we have a data structure $\mathcal{D}$ for some decomposable problem, and we wish to add the capability of range restriction; we will call that component of the record which is used in the range restriction the range component.

Let U, Q, and S be the complexity functions for $\mathcal{D}$. We will assume that S grows at least linearly. Let F be the current set of records. We will use a special bounded balance tree T to represent F. The tree T will be arranged according to the range component of the keys in F; however, each node x will in addition have a field AUX(x) which points to an instance of the original data structure $\mathcal{D}$, containing all records which correspond to leaves descending from x. This structure is, so far, much like that used in [B78], except that we use bounded balance trees instead of the

much more restrictive class of trees in [B78].

Lemma 1. The space complexity of this structure is O(S(n) log n).

Proof. Note that at any given depth in the tree, the set of records in the auxiliary data structures is disjoint. Thus the total space for all of them is O(S(n)), by fact that S is at least linear. Since there are O(log n) levels, we obtain the bound of the lemma. ■

The procedure QUERY below will perform a range-restricted query on such a data structure.

```
procedure QUERY(ℓ,u,x,T)
 begin comment return the value of the query q for the
    subfile consisting of all records whose range component
    is in [ℓ,u];

        R ← a set of roots of disjoint subtrees whose union
            is all nodes whose range component is in [ℓ,u];

        QUERY ← ⊔_{r ∈ R}   (the response of AUX(r) to query q);

 end;
```

Lemma 2. The time for an execution of QUERY is O(Q(n) log n).

Proof. Note that $|R| = O(\log n)$ and that the time to construct R is O(log n) by standard tree searching techniques. The bound of the lemma follows immediately. ■

Again, Lemmas 1 and 2 are similar to results in [B78]. The problem of performing an insertion in such a tree is, however, a more difficult one, and will require us to take a more careful look at bounded balance trees. In particular, we wish to make sure that rotations are not done too frequently. A trick similar to that used in [GMPR77] is useful here; we will define several

degrees of balance, and guarantee that when a node is rebalanced it becomes sufficiently well balanced that it does not need to be considered for a while. (In [GMPR77] B-trees were used; for us trees of bounded balance will be more useful.) The following lemma will make this possible.

Lemma 3. There exist α, α' and γ, with $\alpha' < \alpha$, which make the following true. Let T be a subtree, rooted at a node x, in which all nodes are α'-balanced. Suppose that x is not α-balanced. Then after the call to BALANCE(x), all nodes in T which actively participated in the rebalancing will be α-balanced.

Proof sketch. Some values of α, α', and γ which work are shown in Table 1. These can be verified by a tedious analysis of inequalities for the balances of nodes x, y, and z; see the appendix for details. ■

α'	α	γ
0.05000	0.05249	0.72500
0.10000	0.10989	0.70000
0.15000	0.17191	0.67500
0.20000	0.23809	0.65000
0.25000	0.28077	0.60961

Table 1. Acceptable values for α, α', and γ.

Henceforth we assume we have chosen specific values for α, α', and γ. As a measure of the balance of a node, it is useful to define

$$\beta(x) = 2(\alpha-\alpha')^{-1}\max(0,\rho(x)-(1-\alpha),\alpha-\rho(x))$$

Thus a node is α'-balanced iff $\beta(x) \leq 2$, and α-balanced iff $\beta(x)=0$.

Lemma 4. There is a constant ξ such that if a deletion or insertion is made in a tree, and no rebalancing is done, then for each node in the tree the change in $\beta(x)$ rank(x) is bounded above by ξ.

The proof is easy and is omitted.

We can now describe an insertion procedure for our data structure.

```
procedure INSERT(w,T);
  begin comment insert node w in T;
    insert node w into T according to
      its range component, thinking of T as a binary search
      tree;
    for each ancestor x of w do
      insert the record represented by w into AUX(x);
    if any node x on the path from w to the root
      has β(x) > 2 then
        PANIC: begin
          let x be the highest node with β(x) > 2;
          rebuild the tree rooted at x into a BB(1/3) tree;
          rebuild all the auxiliary structures in the
            tree rooted at x;
        end;
    for each node x on the path from w to the root do
      if β(x) > 1 then
        ROTATE: begin
          BALANCE(x);
          for each node y whose set of descendants
            has been changed by this call to
            BALANCE do
              AUX(y) := a structure of type 𝒟 for all
                records corresponding to descendants of y;
        end;
  end;
```

This algorithm tries to maintain $\beta(x) \leq 2$, for all x in T, by means of rotation operations. If $\beta(x)$ ever becomes greater than 2, an emergency rebalancing takes place in block PANIC. (Note that it is too late for rotation, since the conditions of Lemma 3 are violated.)

Lemma 5. Suppose the average time required per insertion in $\mathcal{D}$ is $\overline{U}(n)$, where n is the maximum number of records ever present. Then the average time required by procedure INSERT is $O(\overline{U}(n) \log n)$.

Proof. Outside of the blocks labeled PANIC and ROTATE, the time bound is clearly satisfied. For block PANIC, note that a single insertion must have changed $\beta(x)$ from 1 or less to more than 2. Thus, the rank of x must, by Lemma 4, be bounded by O(1), so the entire block is doable in time O(1).

For the time spent in block ROTATE, we use an accounting argument much like that of [L78]. Let I(T) be the sum, over all nodes x in T, of

$$\overline{U}(n)\ \beta(x)\ \text{rank}(x)$$

Initially I(T) is 0, since the tree is empty. Note that a single insertion increases I(T) by at most $O(\overline{U}(n) \log n)$ since $\beta(x)$ rank(x) changes for only O(log n) nodes, and by Lemma 4 each change is O(1). On the other hand, a ROTATE operation at x decreases $\beta(x)$ from more than 1 to 0, by Lemma 3 and the de-

finition of β; hence I(T) decreases by at least $\overline{U}(n)$ rank(x); this is (up to constant factors) a bound on the amount of time required for the ROTATE block. Thus on the average each INSERT must cost at most $O(\overline{U}(n) \log n)$. ■

If the original data structure $\mathcal{D}$ supported deletions, a similar procedure and lemma may be produced for deletions in T. Thus we have:

Theorem 1. Let $\mathcal{D}$ be a dynamic data structure with complexity measures $\overline{U}$, Q, and S. Then we may produce a new structure $\mathcal{D}'$ with range restriction capability whose complexity is described by the primed functions below.

$$\begin{aligned} \overline{U}'(n) &= O(\overline{U}(n) \log n) \\ Q'(n) &= O(Q(n) \log n) \\ S'(n) &= O(S(n) \log n) \end{aligned}$$

If $\mathcal{D}$ supported deletions, $\mathcal{D}'$ also will.

3. A Transformation with Guaranteed Time Bounds per Operation

Note that for nodes y and z in Figure 1, the set of descendants changes. Thus the auxilary structures need to be rebuilt. Since we now wish to guarantee worst case times for individual operations, we will not necessarily be able to do this all at once; we will sometimes have to do it a little bit at a time. Until this is complete, we must have some means for responding to queries.

This problem is somewhat reminiscent of that tackled in [FR68]. Let A be an alphabet of 2k symbols, corresponding to positive and negative unit vectors in each of k dimensions. Let L be the language consisting of all strings over A which contain equal numbers of positive and negative symbols in each dimension. (L can be viewed as the set of walks in k-space, starting at the origin, which end at the origin.) In [FR68] it is shown that this language can be recognized in real time by one-tape Turing machine. The construction involves maintenance of counters which may not always be fully updated, but which are guaranteed to become correct as the origin is approached.

In our solution, we also allow the data structure to be only partially complete at certain points, but we guarantee that it is maintained well enough to support queries at any time. To do so, we provide each node x with some additional fields, as follows.

FLAG - This is a boolean field which is ordinarily false. When a node plays the role of y or z in Figure 1, we set FLAG to true to indicate that a valid AUX field is not available. In this case, a response to a query is to be made by adding the responses for the AUX fields of the left and right child of x. Once AUX becomes complete, we set FLAG to false.

L - This contains a list of all nodes which are represented in AUX.

Bounds on space and query time like those in Lemmas 1 and 2 will still hold. However, the alert reader will notice a potential problem with this scheme. Suppose that FLAG(x) is true. What if when we look to the children of x, we discover that these also have FLAG set to true and thus have AUX fields which are not yet rebuilt from some previous rotation? Much of the remainder of this section is devoted to this problem. Some definitions will be useful. Say a node is unified if its FLAG field is false; otherwise it is disunified. Define the near descendants of a node to be the node itself and its children, grandchildren, and great-grandchildren. Say a node is eligible for rotation, or more briefly, eligible, if all of its near descendants are unified. We will eliminate the problem discussed at the beginning of this paragraph by only performing rotations at eligible nodes.

Below we present the modified algorithm INSERT. This algorithm contains a parameter c to be chosen later. A deletion procedure may be similarly defined; we will not present the details. By a fixup operation at a node y we mean the following procedure; the procedure assumes y is disunified.

```
procedure FIXUP(y);
  begin comment work on unifying y;
        let y' and y" be the children of y;
        z ← an element of L(y') ∪ L(y") - L(y);
        add z to the data structure AUX(y);
        add z to L(y);
        if L(y') ∪ L(y") - L(y) = φ then FLAG(y) := false;
  end;
```

Lemma 6. The procedure FIXUP can be made to run in time bounded by O(1) plus the time for a single insertion in $\mathcal{D}$.

Proof. The only potential problem is the calculation of $L(y') \cup L(y") - L(y)$. To do this efficiently, we associate two new fields DIFL and DIFR with each node y; these are pointers into the lists L(y') and L(y"), manipulated in such a way that the union of the sublists to which they point is $L(y') \cup L(y") - L(y)$. When L(y) is set to ϕ, DIFL and DIFR are set to the beginning of L(y') and L(y"). Addition of a node to L(y) can be reflected by advancing DIFL or DIFR in L(y') or L(y"). The details are a straightforward link manipulation problem, and are omitted. ■

Note that nodes x with $\beta(x) > 1$ are rebalanced when eligible. If the node has not yet become eligible by the time $\beta(x)$ exceeds 2, we execute block PANIC and completely rebuild the corresponding subtree by brute force. Of course, this can be time consuming. As in the previous section, we can show that this will happen only for nodes with rank O(1); the proof is somewhat more difficult, however, and is contained in the following two lemmas. By an operation involving x we will mean an insertion or deletion of a descendant of x.

```
procedure INSERT(w,T);
  begin comment insert node w in T;
    insert node w in T according to its range component,
      thinking of T as a binary search tree;
    for each ancestor x of w do
      begin
        insert the record represented by w into AUX(x);
        insert w into L(x);
      end;
    if any node x on the path from w to the root has β(x) > 2 then
      PANIC: begin
          let x be the highest node with β(x) > 2;
          rebuild the tree rooted at x into a BB(1/3) tree;
          rebuild all of the auxiliary structures in the tree
            rooted at x;
      end;
    for each node x on the path from the root to w do
      if x is eligible and β(x) > 1 then
        ROTATE: begin
          BALANCE(x);
          for each node y whose set of descendants has been changed
            by this call to BALANCE do
            begin
              FLAG(y) ← true;
              AUX(y) ← the null structure for 𝒟;
              L(y) ← ϕ;
            end;
        end;
    for each node x on the path from the root to w do
      begin
        do fixup operations on the disunified near descendants
          of x until either x is eligible or c fixups have
          been done;
      end;
end;
```

Lemma 7. For every $\delta > 0$ we may choose c in algorithm UPDATE so that the following holds. Let x be any node in the tree, with r=rank(x). If we then do at least δr operations involving node x, and x does not actively participate in a rebalance during this time, at the end of one of these operations x must be eligible.

Proof. First we give some informal motivation for the proof. The idea is to make the constant c large enough so that enough fixups take place during the δr operations to guarantee that all relevant nodes are unified. A problem that arises is that rebalancing operations on near descendants of x can work against unification in two ways:

a) The rebalancing causes certain nodes to become disunified.

b) The rebalancing moves some nodes toward the root, so that the node x in the lemma may acquire new near descendants, which may not be unified and may soon need to be rebalanced themselves.

In order to take these problems into account, we will use an accounting argument.

Let S be the set of near descendants of x in T. For any node y in T, let DIS(y) be a measure of the disunity of y, defined by

$$DIS(y) = \left| L(y') \cup L(y'') - L(y) \right|$$

where y' and y" are the children of y. Now define the cost of x in T, denoted C(x,T), as

$$C(x,T) = \sum_{y \in S} \left(DIS(y) + W_{dist(x,y)} \; rank(y) \; \beta(y) \right)$$

where dist(x,y) is the distance from x to y and $W_0=0$, $W_1=18$, $W_2=6$, and $W_3=2$. (For later convenience, also define $W_4=0$.) Note that this takes into account the amount of reunification required for all near descendants of x, and also takes into account the imbalance of these nodes with various weighting factors.

We now begin the actual proof by noticing that there is a constant δ_1 such that before the sequence of δr operations,

$$C(x,T) \leq \delta_1 \, r.$$

This is true since $|S|$ is at most 15, and DIS(y) and rank(y) are bounded in size by the rank of x for all y in S.

Next we observe that any rebalancing operation on a descendant of x cannot increase C(x,T). First, it is clear that a rebalance operation of the sort done during PANIC cannot increase C(x,T). Now let d=dist(x,y) and consider the ways in which a rebalancing operation on y during statement ROTATE causes C(x,T) to change.

a) One or two children of y become disunified. This causes their DIS value to increase, which increases C(x,T) by at most rank(y).

b) Certain nodes move closer to y. Thus some nodes which were not near descendants of x will be, and some which were already near became associated with higher weighting factors. Let μ_ℓ denote the maximum number of nodes which can move closer to y during a rotation, and afterwards appear at a distance of ℓ from y. Inspection of the single and double rotations shows that

$$\mu_1 = 1 \qquad \mu_2 = 2 \qquad \mu_3 = 4.$$

This effect increases C(x,T) in two ways.

b1) Imbalanced nodes become associated with higher weighting factors. The resultant increase in C(x,T) is bounded by

$$2\ \mathrm{rank}(y) \sum_{\ell=d+1}^{3} \mu_{\ell-d}(W_\ell - W_{\ell+1})$$

b2) For each new member z of S, we must add DIS(z); this total contribution can easily seen to be bounded by rank(y).

c) Finally, $\beta(y)$ decreases by at least one, which decreases C(x,T) by at least

$$W_d\ \mathrm{rank}(y).$$

Adding all of these effects, we see that the increase in C(x,T) is bounded above by

$$\mathrm{rank}(y) \left[1+2 \sum_{\ell=d+1}^{3} \mu_{\ell-d}(W_\ell - W_{\ell+1}) + 1 - W_d \right].$$

It can readily be verified that the quantity in square brackets is equal to zero for d=1, 2, or 3. Thus rebalancing does not increase C(x,T).

Next recall that a single insertion or deletion can increase $\beta(y)$ rank(y) by at most O(1) for any node in the tree, before any rebalancing is performed. Thus each insertion or deletion increases C(x,T) by at most some constant δ_2.

Finally, assume for a contradiction that x was ineligible at the end of each of the δr operations involving x. Then during each of these operations we must have done c fixups on near descendants of x. Each fixup clearly reduces C(x,T) by 1. Putting the results obtained thus far together, we conclude that after t operations beyond the initial tree, we have

$$C(x,T) \leq \delta_1 r + \delta_2 t - ct. \qquad (1)$$

Let

$$c = \frac{\delta_1 + \delta_2 \delta}{\delta}$$

Then the right hand side of (1) becomes 0 when $t = \delta r$; this means that x must be eligible of the end of the δr operations, contradicting our assumption. ■

Lemma 8. If we choose c large enough, there is a constant M such that block PANIC is never performed for any node y with rank(y) greater than M.

Proof. Let ξ be as in Lemma 4. Choose c large enough so that Lemma 7 holds with

$$\delta = \frac{1}{2\xi + 2} \qquad (2)$$

Suppose that at some point during the algorithm, node x has to be balanced in block PANIC. Let r_1 be rank(x) at this point. Now recall back to the last point at which node x had $\beta(x) \leq 1$ at the end of an operation; for convenience call this t_0, and call the present point t_1.

(Note that x has not actively participated in a rotation since t_0.) Let r_0 be the rank of x at t_0, and let s be the number of insertions and deletions involving x since t_0; denote these s operations by op_1, op_2, ..., op_s. Since insertions and deletions change the rank by at most 1, during the entire interval from t_0 to t_1 we have

$$\text{rank}(x) \in \left[r_0 - s,\ r_0 + s \right]. \tag{3}$$

Then since insertions and deletions change $\beta(x)$ by at most $\xi/\text{rank}(x)$, and since $\beta(x)$ has changed from at most 1 to more than 2, we conclude that

$$1 < \xi(r_0 - s)^{-1}s$$

unless $s \geq r_0$. Thus in any case,

$$(\xi + 1)s > r_0. \tag{4}$$

Now since x did not actively participate in a rotation during op_2, op_3,..., op_{s-1}, it must have been ineligible at the end of op_1, op_2, ..., op_{s-2}. Thus, by Lemma 7 and our choice of c,

$$s - 2 < \delta r_0. \tag{5}$$

Now, combining (4) and (5) we obtain

$$\delta(\xi + 1)\ s > s-2\ .$$

Using (2) we thus have

$$\frac{1}{2}\ s > s-2$$

so, since s is integral,

$$s \leq 3. \tag{6}$$

Finally, by (3), (4), and (6), we deduce that at the time of PANIC,

$$\begin{aligned} \text{rank}(x) &\leq r_0 + s \\ &\leq (\xi + 1)s + s \\ &\leq 3(\xi + 2). \end{aligned}$$

Thus the lemma is established. ∎

Corollary. Each insertion and deletion can be done using O(log n) basic operations and O(log n) insertion and deletion operations on the original data structure.

We may finally summarize all of the results as follows.

Theorem 2. Let $\mathcal{D}$ be a dynamic data structure with complexity measures U, Q, and S. Then we may produce a new structure $\mathcal{D}'$ with range restriction capability whose complexity is described by the primed functions below.

$$\begin{aligned} U'(n) &= O(U(n) \log n) \\ Q'(n) &= O(Q(n) \log n) \\ S'(n) &= O(S(n) \log n) \end{aligned}$$

If $\mathcal{D}$ supported deletions, $\mathcal{D}'$ also will. (Note that this theorem is identical to Theorem 1, except that we have removed the bars from the U's.)

By repeated application of this theorem we obtain the following.

Corollary. There is a data structure for orthogonal range queries in k dimensions with

$$U(n) = O(\log^k n)$$
$$Q(n) = O(\log^k n)$$
$$S(n) = O(n \log^{k-1} n).$$

4. Some applications

The empirical cumulative distribution function problem is a special case of range searching in which we ask for the number of vectors which are less than a query vector in all components. See [BS77], where a static structure for this problem is presented with complexities

$$P(n) = S(n) = O(n \log^{k-1} n)$$
$$Q(n) = O(\log^{k} n)$$

The results of the previous section enable us to solve this problem dynamically in worst-case $O((\log n)^k)$ time per operation.

A more interesting problem is the following. Given two vectors v and w, say w dominates v if $w \neq v$ and $w \geq v$, where inequalities are interpreted componentwise. Given a vector v and a set F of vectors, say v is maximal with respect to F if there is no $w \in F$ which dominates v. Let k be the dimension of the vectors, and let n be the number of vectors in F. In [KLP75] an $O(n \log^{k-2} n)$ time algorithm is presented for $k \geq 3$, which would tell which vectors in F are maximal with respect to F. (For k=1 the problem is trivially O(n), and for k=2 they give an O(n log n) algorithm.) In [B78] a static algorithm is presented, with complexities

$$P(n) = S(n) = (n \log^{k-2} n)$$
$$Q(n) = O(\log^{k-2} n),$$

which tells whether an arbitrary v is maximal with respect to F. We wish to find a dynamic structure for this task. (One could use the static-to-dynamic transformation quoted in section 1 adding a factor of O(log n) to the total time and space [B78];

here we will produce an algorithm with faster query times and guaranteed times for individual updates.) The techniques discussed already easily yield a solution with $O(\log^k n)$ time per operation. We can save a factor of $O(\log n)$ by using insights from [KLP75,B78].

Theorem 3. For $k \geq 2$, there is a dynamic structure for the maximal vector problem with complexities

$$U(n) = Q(n) = O(\log^{k-1} n)$$
$$S(n) = O(n \log^{k-2} n).$$

Proof. We use induction on k. Let HEAD(v) denote the first component of v, and let TAIL(v) denote the k-1 dimensional vector formed by removing the first component.

Basis. For k=2, note that TAIL(v) has but a single component. Then v is maximal if and only if

$$\text{TAIL}(v) \geq \max \{\text{TAIL}(w) \mid w \in F \text{ and } \text{HEAD}(w) \geq \text{HEAD}(v)\}, \text{ and}$$
$$\text{TAIL}(v) > \max \{\text{TAIL}(w) \mid w \in F \text{ and } \text{HEAD}(w) > \text{HEAD}(v)\}.$$

(Surprisingly, neither condition by itself is sufficient.) These tests are simply one-dimensional range queries, so the desired complexity bounds hold.

Inductive step. For $k > 2$, we show the theorem holds for k assuming it held for k-1. Note that v is maximal if and only if both conditions below hold:

a) TAIL(v) is maximal with respect to

$$\{\text{TAIL}(w) \mid w \in F \text{ and } \text{HEAD}(w) \geq \text{HEAD}(v)\}$$

b) HEAD(v) is greater than or equal to

$$\max\{\text{HEAD}(w) \mid w \in F \text{ and } \text{TAIL}(w) \geq \text{TAIL}(v)\}.$$

Now (a) can be performed in the desired time bound by the inductive hypothesis and Theorem 2; (b) is merely a (k-1)-dimensional range query and is thus doable in the desired time bound by the corollary to Theorem 2. ■

Conclusions

In this paper we have used trees of bounded balance to produce a data structure for orthogonal range queries which allows insertion, deletion, and query operations to be performed with a guaranteed worst-case time of $O(\log^k n)$ for each operation. More significantly, we have demonstrated that range restriction capabilities may be added to any dynamic structure with only a factor of log n increase in time and space usage.

The constants hidden in the O-notation are probably fairly large. For practical applications which do not demand the guaranteed times for individual operations, other structures discussed in [BF78, L78, W78] may well be preferable.

In the course of our analysis we have come to a much deeper understanding of the capabilities of trees of bounded balance. This may also prove helpful in designing other algorithms.

Acknowledgements

Jon Bentley has had several very useful discussions with me on this problem; in particular, he brought to my attention a number of references, and suggested section 4 of the paper. Dan Willard and Steve Stedry have also had some interesting discussions with me. Dov Harel carefully read a manuscript and made helpful suggestions.

Appendix: Proof of Lemma 3.

Suppose we are about to perform a rotation as in Figure 1. Let

$$\rho_1 = \rho(x)$$
$$\rho_2 = \rho(y)$$
$$\rho_3 = \rho(z)$$

(where ρ_3 is relevant only if a double rotation is to be performed). By the assumptions of lemma, we have

$$\left.\begin{array}{lll} \rho_1 \in [\alpha', \alpha] & & \\ \rho_2 \in [\alpha', \gamma] & \text{(single rotation)} & \\ \rho_2 \in [\alpha, 1-\alpha'] & \text{(double rotation)} & \\ \rho_3 \in [\alpha', 1-\alpha'] & & \end{array}\right\} \quad \text{(A1)}$$

After the rotation, the new balances (call them ρ_1', ρ_2', and ρ_3') are as shown in Table 2 [NR73]. It is not hard to verify that each quantity is monotonic in ρ_1, ρ_2, and ρ_3 over the domain of interest. The nature of the monotonicity is also shown in the table; "+" denotes monotonic increasing,"-" denotes monotonic decreasing, and "O" denotes independence from the ρ_i in question. On the basis of these monotonicities and the inequalities in (A1), we obtain the last two columns of the table which tell lower and upper bounds on ρ_1', ρ_2', and ρ_3'.

It is now a matter of simple calculation to verify that, for each of the choices for α, α', and γ in Table 1, each quantity in the last two columns of Table 2 lies in $[\alpha, 1-\alpha]$. (The values in

Rotation Type	Variable	Value	Monotonicity ρ_1	Monotonicity ρ_2	Monotonicity ρ_3	Bounds Lower	Bounds Upper
Single	ρ_1'	$\rho_1 + (1-\rho_1)\rho_2$	+	+	0	$(2-\alpha')\alpha'$	$\alpha + (1-\alpha)\gamma$
	ρ_2'	$\frac{\rho_1}{\rho_1 + (1-\rho_1)\rho_2}$	+	-	0	$\frac{\alpha'}{\alpha' + (1-\alpha')\gamma}$	$\frac{\alpha}{\alpha + (1-\alpha)\alpha'}$
Double	ρ_1'	$\rho_1 + (1-\rho_1)\rho_2\rho_3$	+	+	+	$\alpha' + (1-\alpha')\gamma\alpha'$	$\alpha + (1-\alpha)(1-\alpha')^2$
	ρ_2'	$\frac{\rho_1}{\rho_1 + (1-\rho_1)\rho_2\rho_3}$	+	-	-	$\frac{\alpha'}{\alpha' + (1-\alpha')^3}$	$\frac{\alpha}{\alpha + (1-\alpha)\gamma\alpha'}$
	ρ_3'	$\frac{\rho_2(1-\rho_3)}{1 - \rho_2\rho_3}$	0	+	-	$\frac{\gamma\alpha'}{1 - (1-\alpha')\gamma}$	$\frac{(1-\alpha')^2}{1 - (1-\alpha')\alpha'}$

Table 2. Description of balances after a rotation.

Table 1 were calculated by a computer program which tried to determine the maximum allowable value of α for a given value of α'.) ■

References

√[B78] Bentley, J. L., "Decomposable Searching Problems," Computer Science Report CMU-CS-78-145, Carnegie-Mellon University, 1978; to appear, Information Processing Letters.

[Bp] Bentley, J. L., private communication, August 1978.

√[BF78] Bentley, J. L., and Friedman, J. H. "A Survey of Algorithms and Data Structures for Range Searching," Proc. of the Computer Science and Statistics: Eleventh Annual Symposium on the Interface, (March 1978), pp. 297-307.

[BM78] Bentley, J. L., and Maurer, H. A., "Efficient Worst-Case Data Structures for Range Searching," draft, 1978.

[BS77] Bentley, J. L. and Shamos, M. I., "A Problem in Multivariate Statistics: Algorithm, Data Structure, and Applications," Proceedings of the Fifteenth Allerton Conference on Communication, Control, and Computing, (September 1977), pp. 193-201.

[BS79] Bentley, J. L., and Shaw, M., "An Alphard Specification of a Correct and Efficient Transformation on Data Structures," to be presented at IEEE Conference on Specifications of Reliable Software, April 1979.

[FR68] Fischer, M. J., and Rosenberg, A. L., "Real-Time Solutions of the Origin-Crossing Problem," Math. Systems Theory, 2:3 (1968), pp. 257-263.

[GMPR77] Guibas, L. J., McCreight, E. M., Plass, M. F., and Roberts, J. R., "A New Representation for Linear Lists," Proc. Ninth Annual Acm Symposium an Theory of Computing, (May 1977), pp. 49-60.

[K73] Knuth, D., The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison-Wesley, Reading, Mass., 1973.

[KLP75] Kung, H. T., Luccio, F., and Preparata, F. P., "On Finding the Maxima of a Set of Vectors," JACM 22:4 (October 1975), pp. 469-476.

[LW78] Lee, D. T., and Wong, C. K., "Quintary Tree: A File Structure for Multi-dimensional Database System," IBM Research Report RC 7285 (#31364), August 28, 1978.

[L78] Lueker, G. S., "A Data Structure for Orthogonal Range Queries," Proc. 19th Annual Symposium on Foundations of Computer Science," (October 1978), pp. 28-34.

[NR73] Nievergelt, J., Reingold, E. M., "Binary Search Trees of Bounded Balance," SIAM J. Comput., 2:1 (1973), pp. 33-43.

[W78] Willard, D. E., Predicate-Oriented Database Search Algorithms, Ph.D. thesis, Aiken Computation Laboratory, Harvard University, 1978; available as technical report TR-20-78.

[Wp] Willard, D. E., private communication, January 1979.