A DATA STRUCTURE FOR ORTHOGONAL RANGE QUERIES

George S. Lueker+

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

+Partially supported by NSF Grant MCS77-04410

Abstract

Given a set of points in a d-dimensional space, an orthogonal range query is a request for the number of points in a specified d-dimensional box. We present a data structure and algorithm which enable one to insert and delete points and to perform orthogonal range queries. The worst-case time complexity for n operations is $O(n \log^d n)$; the space used is $O(n \log^{d-1} n)$. (O-notation here is with respect to n; the constant is allowed to depend on d.)

Next we briefly discuss decision tree bounds on the complexity of orthogonal range queries. We show that a decision tree of height $O(dn \log n)$ (where the implied constant does not depend on d or n) can be constructed to process n operations in d dimensions. This suggests that the standard decision tree model will not provide a useful method for investigating the complexity of such problems.

1. Introduction

A number of highly efficient data structures have been devised which make it possible to manipulate a set of records with totally ordered keys. For example, suppose one wishes to be able to insert, delete, or locate a given record in a set of n elements. These operations can all be performed in an average of $O(\log n)$ time per operation through the use of binary search trees [K73, AHU74]; by using 2-3 trees, AVL trees, or trees of bounded balance, the time bound can be improved to worst-case $O(\log n)$ [AL62, K73, NR73, AHU74]. These trees also allow more general operations to be performed [see C72, K73, AHU74]. For example, we may wish to perform a "range query," that is, to determine the number of keys in the range [a,b]; it is not hard to devise a worst-case $O(\log n)$ algorithm to do this using any of the three balanced schemes mentioned above, provided each node has a field containing its number of descendants.

Now suppose each key is a vector of length d; more complex queries become possible. For example, we may wish to specify a range for each component of the key and ask how many keys have all components in the desired range. This corresponds to counting the number of elements in some d-dimensional box; Knuth [K73, pp. 554-555] calls this an *orthogonal range query*; unfortunately, he observes that for this problem "No really nice data structures seem to be available."

One data structure which has been proposed for handling range queries is the *quad tree* [FB74]. A d-dimensional quad tree is much like a binary search tree, except that each node has up to 2^d children; thus when a new node is inserted, its relation to its parent can depend on the outcome of comparisons of all d components of the keys. A closely related data structure called a *multidimensional binary search tree* has been introduced [B75]; this data structure also allows one to perform orthogonal range queries. It has been shown that for either data structure, if the trees involved are balanced, an orthogonal range query can be performed in a worst case time of $O(n^{1-1/d})$ [BS75, LW77]; analysis of the average performance appears to be more difficult.

Various authors have examined related problems involving multidimensional records. Rivest [R76] lists a hierarchy of multidimensional retrieval problems. A particularly interesting special case of orthogonal range queries is partial match queries. Such a query consists of a specification of exact values for certain components of the key; the other components are called "don't cares." Rivest presents a variety of schemes which use about $O(n^{\log_2 1+\alpha})$ or $O(n^{\alpha})$ time to respond when α is the fraction of "don't cares" in the query, and n is the number of records. Each scheme involves little or no storage redundancy. He conjectures that $O(n^{\alpha})$ is the best possible average performance with binary records when no more than constant redundancy is allowed.

A number of authors have investigated related problems which are not special cases of orthogonal range queries. For example, Bentley and Shamos [BS76] have investigated various problems involving sets of points in d-space. They present an $O(n \log n)$ algorithm for finding the closest two points. An important technique used in their paper is multidimensional divide-and-conquer; they divide a problem on n points in d dimensions into two problems on n/2 points in d dimensions, and another problem on n points in d-1 dimensions. This same technique will prove useful here.

Dobkin and Lipton [DL76] investigate multidimensional searching problems. For example, they show that the problem of determining whether a point lies on any of a set of n lines in d dimensions can be solved in $O(\log n)$ steps; they assume the set of lines is fixed and that precomputation on the set is free.

In [KLP75] the problem of determining which elements of a set of n d-dimensional records are maximal under the natural partial ordering is considered. For $d \geq 3$, they present an $O(n \log^{d-2} n)$ algorithm. They also investigated lower bounds, and were able to show that $\log_2(n!)$ comparisons were required for $d \geq 2$. (In section 4 we will provide a partial explanation of the difficulty of proving stronger lower bounds.)

In section 2 we discuss a data structure for orthogonal range queries by which a string of n insert, delete, and query instructions (starting with an empty set) can be performed in average time of $O(n \log^d n)$. In section 3 this data structure is improved so that the worst-case time for n operations is $O(n \log^d n)$; this is obtained through a combination of a multidimensional divide-and-conquer approach and bounded-balance tree schemes. Section 4 suggests that it will be difficult to use decision trees to prove good lower bounds on the complexity of orthogonal range queries.

2. A simple algorithm with fast average behavior

Define a d-fold binary search tree, on a set of n d-dimensional keys, inductively as follows. If d=1, a d-fold binary search tree is an ordinary binary search tree; since d=1, keys have only one component, and the tree is ordered according to this component. If d>1, then a d-fold binary search tree on n records is a binary search tree organized according to the dth components of keys; in addition, however, each node x contains a field AUX which points to a (d-1)-fold binary search tree containing all records which descend from x, organized according to the first d-1 components of the keys. The n nodes in the binary tree organized according to the dth component are called primary nodes; the nodes in all of the other trees are called secondary nodes. An algorithm for inserting in such a tree is shown below. (The set of descendants of x is considered to include x; it does not include any of the nodes in AUX(x). Similar remarks apply to the term ancestor.)

```
procedure INSERT(d,k,T);
begin
  comment insert a key k into a d-fold binary
    search tree T;
  create a new node x for key k and insert it
    into T according to component d, thinking of
    T as a binary search tree;
  if d>1 then
    for each ancestor y of x do
      INSERT(d-1,k,AUX(y));
end;
```

This tree T may be used as the basis of an algorithm to respond to an orthogonal range query as follows. First, find the set of subtrees in T which corresponds to records whose dth component satisfies the query; let R be the set of roots of these trees. If T is well balanced, $O(\log n)$ is a bound on the size of R and the time to compute R.

Now sum the results of orthogonal range queries on the remaining d-1 dimensions for the auxiliary trees of all of the nodes in R. If T is well-balanced, an easy induction shows that this algorithm runs in $O(\log^d n)$ time.

On the average, T will be well balanced, and can be computed efficiently. Assume the components of keys are chosen independently and randomly. Let $T(n,d)$ be the expected time to insert n keys into a d-fold binary search tree.

Theorem 1. $T(n,d)=O(n \log^d n)$.

Proof. We use induction on d. For d=1, clearly $T(n,d)=O(n \log n)$. Now let d>1. Since the expected internal path length of T is $O(n \log n)$, the expectation of the total number of records inserted in auxiliary trees of primary nodes of T is $O(n \log n)$. By the induction hypothesis, the time to insert k keys in one of these auxiliary trees is $O(k \log^{d-1} k)$, which is $O(k \log^{d-1} n)$. Thus the time for all of the $O(n \log n)$ insertions must be $O(n \log^d n)$. []

3. An algorithm with good worst-case behavior

In this section we will introduce a data structure which makes it possible to perform a sequence of n insertions, deletions, and orthogonal range queries in worst-case $O(n \log^d n)$ total time. The basic idea is to use a balanced tree scheme. A problem that arises is that rebalancing a node may require the associated auxiliary tree to be completely restructured; this can be expensive for large trees. We would like to guarantee that this will happen only very rarely for trees with many nodes. Below we will show how trees of bounded balance [NR73] can be used to produce the desired data structure.

Let the rank of a node x, written rank(x), be one more than the number of nodes which descend from x. 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 that, assuming $\alpha \lesssim 1-\sqrt{2}/2$, 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.

In order to use the notion of bounded balance in our context, we say a d-fold binary search tree is a d-fold $BB(\alpha)$ tree if the balance of each primary or secondary node is in $[\alpha, 1-\alpha]$. We begin by observing that we can build a d-fold BB(1/3) tree efficiently using the procedure BUILD, shown below. The procedure uses a multidimensional divide-and-conquer approach like that described in [BS76].

```
procedure BUILD(L,d);
begin
  comment L is a list of n keys, in ascending
    order according to component d. The routine
    returns a d-fold BB(1/3) tree for the keys in
    L. Also, if d>1, L is resorted according to
    component d-1;
  n ← the number of elements in L;
  if n=0
    then return the empty tree;
  m ← ⌈n/2⌉;
  LL ← the first m-1 elements of L;
  k ← the mth element of L;
  LR ← the last n-m elements of L;

  x ← a new node;
  KEY(x) ← component d of k;
  LEFT(x) ← BUILD(LL,d);
  RIGHT(x) ← BUILD(LR,d);
  if d=1
    then AUX(x) ← the empty tree
    else
      begin
        comment the above calls to BUILD have
          ordered LL and LR into ascending order
          according to component d-1;
        merge LL, LR, and k into a list sorted
          according to component d-1, and store
          the result in L;
        AUX(x) ← BUILD(L,d-1);
      end;
end;
```

Theorem 2. Given a list of n keys, we may form them into a d-fold BB(1/3) tree in $O(n \log^{\min(1,d-1)} n)$ time.

Proof. First sort the keys into ascending order according to the dth component; then call algorithm BUILD. That each tree constructed is a binary search tree on the appropriate component can be seen by an easy inductive proof, using the fact that the lists are ordered. Further, the balance of each node is of the form $\lfloor n/2 \rfloor/(n+1)$, which lies in $[1/3, 1/2]$.

Let $T(n,d)$ be the worst-case time used by procedure BUILD. $T(n,1)$ is easily seen to be $O(n)$, and for $d>1$ we readily establish the recurrence

$$T(n,d) = 2T(n/2,d) + T(n,d-1) + O(n).$$

This same recurrence arose in [BS76]; the solution is

$$T(n,d) = O(n \log^{d-1} n).$$

Recalling the extra time for the initial sort we obtain the bound stated in the theorem. []

Next we discuss the effect of insertions and deletions.

Lemma 1. Insertion or deletion of a node in a binary search tree on k records can change the balance of the root by only $O(1/k)$.

The proof is easy and is omitted.

The algorithm INSERTB for insertion in a d-fold $BB(\alpha)$ tree is given below; deletion can be handled similarly. (We henceforth assume $\alpha \le 1-\sqrt{2}/2$.)

Theorem 3. In the worst case, INSERTB uses $O(n \log^d n)$ time for n insertions.

Proof. Let $T(n,d)$ be the worst-case time required to perform n insertions on a d-fold $BB(\alpha)$ tree. It is clear that

$$T(n,1) = O(n \log n).$$

Now suppose $d>1$. Aside from the time spent in the block labeled REBALANCE, the time bound is easy. We will use an accounting argument to establish the time bound for the REBALANCE block. We begin with a few definitions. For any primary or secondary node x, let $\beta(x)$ be the distance on the real line

```
procedure INSERTB(d,k,T);
begin
  comment insert key k into a d-fold BB($\alpha$) tree
    T, rebalancing as needed to maintain the
    BB($\alpha$) condition;
  if d=1
    then insert k in T using the standard bounded
      balance insertion procedure
    else
      begin
        insert a new node x for key k into T
          according to component d of k, thinking
          of T as a binary search tree;
        for each ancestor y of x do
          INSERTB(d-1,k,AUX(y));
        if some node on the path from the root of
          T to x has balance outside of [$\alpha$,1-$\alpha$]
          then
            REBALANCE: begin
              locate the first node y on the path
                from the root of T to x for which
                $\rho$(y) $\notin$ [$\alpha$,1-$\alpha$];
              let L be a list of all keys in the
                subtree rooted at y;
              sort L into increasing order
                according to component d;
              replace the subtree rooted at y by
                BUILD(L,d);
            end;
      end;
end;
```

from the point $\rho(x)$ to the set $[1/3,2/3]$; that is,

$$\beta(x) = \max(0, \rho(x)-2/3, 1/3-\rho(x))$$

Let $d(x)$ be the dimension of the subtree of which x is a root. Now we define the imbalance $I(T)$ of a tree T to be the sum, over all primary and secondary nodes x in T, of

$$\beta(x)\ \mathrm{rank}(x) \log^{d(x)-1} n$$

We will charge each insertion operation an amount equal to the increase it produces in the imbalance of T, before any rebalancing is performed. In this paragraph we prove by induction on d that a single insertion can increase $I(T)$ only by $O(\log^d n)$. First note that by Lemma 1, $\beta(x)\ \mathrm{rank}(x)$ can be seen to increase by at most $O(1)$ per insertion for any node in the tree. Thus for d=1, the increase in $I(T)$ during an insertion is $O(\log n)$, since only nodes on the path from the root to the inserted node are affected. Now we prove the assertion for dimension d, assuming it holds for lower dimensions. The total increase of $\beta(x)$ rank(x) for primary nodes in T is O(log n) by an argument like that in the basis, so the change in I(T) due to these nodes is $O(\log n \cdot \log^{d-1} n)$ or $O(\log^d n)$. However, we also insert the new key into O(log n) auxiliary trees, each of dimension d-1. By the induction hypothesis, the increase in I(T) for each such insertion is $O(\log^{d-1} n)$, for a total of $O(\log^d n)$. Thus the total increase in I(T) for a d-fold BB(α) tree due to a single insertion is $O(\log^d n)$.

Next we show that the decrease in I(T) due to a REBALANCE operation at node y is sufficient to cover the cost of the operation. Note that after the operation, β is 0 for all primary or secondary nodes in the new subtree. On the other hand, we are performing the operation since node y had a balance outside $[\alpha, 1-\alpha]$. Thus, initially, we had

$$\beta(y) > \alpha - 1/3.$$

Let r = rank(y) and d = d(y). Then the decrease in I(T) is at least

$$(\alpha-1/3)\ r \log^{d-1} n.$$

On the other hand, by Theorem 2, the cost of the rebuild is

$$O(r \log^{d-1} r).$$

Thus by choice of suitable constants in the accounting argument, we can guarantee that the decrease in imbalance covers the cost.

Since the imbalance of an initial empty tree is 0, and never goes negative, we have shown that the amount charged to the operations covers the rebalancing costs. This completes the proof. []

4. Decision tree bounds

In this section we discuss the complexity of orthogonal range queries under the decision tree model. We will show that an upper bound for processing n operations in d dimensions is O(d n log n). At first this might seem to be a desirable result, since it is much lower than the time bound for the data structure of the preceding section. Actually, however, we feel that the main impact of this result is to show that decision trees do not accurately model the practical difficulty of processing orthogonal range queries. Let R denote the set of real numbers. A map T from R to R is *strictly monotonic* if T preserves strict inequalities. A map from R^d to R^d is strictly monotonic if it is strictly monotonic in each component.

Lemma 2. Let F be a function defined on tuples of vectors chosen from R^d. Suppose F is invariant under strictly monotonic transformations of the input. Then there is a decision tree to compute F which has height O(d n log n), where n is the number of vectors in the input. The implied constant in the O-notation is independent of d and n.

Proof. Construct a decision tree which determines for each i,

a) the permutation which is required to place the ith component of the vectors into ascending order, and

b) which vectors are equal in the ith component.

Clearly a decision tree of height O(d n log n) can be constructed to do this.

Now it is easy to see that two tuples of vectors will lead to the same leaf if and only if one can be mapped into the other by a strictly monotonic transformation. Thus when we reach a leaf we have enough information to deduce the value of F. []

Corollary 1. The problem of determining which of a set of n d-dimensional vectors are maximal [KLP75] is solvable by a decision tree of height O(d n log n).

Corollary 2. The problem of processing a sequence of n insert, delete, and orthogonal range query operations in d dimensions is solvable by a decision tree of height O(d n log n).

Note that the problem of determining whether a point lies on any of a given set of lines, investigated in [DL76], does not satisfy the invariance condition required by the lemma.

5. Conclusions

This paper presents a data structure for orthogonal range queries by which a string of n insert, delete, and query operations on d-dimensional keys may be performed in $O(n \log^d n)$ operations. The previous lack of highly efficient data structures for such queries, combined with their practical importance, makes this an interesting result. Unfortunately, the direct usefulness of this data structure is impaired by two problems:

a) The $O(n \log^d n)$ time complexity, though fast in n, increases rapidly in d.

b) The number $O(\log^{d-1} n)$ of nodes per key in the data structure is also exponential in d.

Nonetheless, for fixed d, this data structure has a time-space product which appears to be asymptotically much lower than that of any previous approach. It would be very interesting to find an algorithm which retains the desirable behavior in n without degrading so rapidly as the dimension becomes large.

It would also be interesting to prove that a data structure must use this much time to process these queries. It appears that the standard decision tree model will not be of much value, however, since there exist decision trees of height $O(d\ n \log n)$ to perform this task; unfortunately, they do not appear to correspond to practical algorithms.

It should be noted that although we used trees of bounded balance, the rebalancing operations discussed in [NR73] are not essential to the data structure. When rebalancing a node of rank greater than one, we completely restructure the tree; further, for trees of rank one, we could use any balanced tree scheme which guarantees logarithmic behavior.

Finally, it should be noted that although we have only considered the problem of counting the number of records in some box, the approach discussed here could easily be used to calculate the sum of some value over the selected records; in fact, the value of any associative and commutative operation over the selected keys could readily be computed.

References

[AL62] G. M. Adel'son-Vel'skiĭ and E. M. Landis, "An Algorithm for the Organization of Information," Sov. Math. Dokl., 3 (1962), pp. 1259-1262.

[AHU74] Alfred Aho, John Hopcroft, and Jeffrey Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Mass., 1974.

[B75] Jon L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," CACM, 18:9 (1975), pp. 509-517.

[BS75] Jon L. Bentley and Donald F. Stanat, "Analysis of Range Searches in Quad Trees," Inf. Proc. Letters, 3:6 (1975), pp. 170-173.

[BS76] Jon L. Bentley and Michael I. Shamos, "Divide-and-Conquer in Multidimensional Space," Proc. 8th Annual ACM Symposium on Theory of Computing, 1976, pp. 220-230.

[C72] C. A. Crane, "Linear Lists and Priority Queues as Balanced Binary Trees," Ph. D. Thesis, Stanford University, 1972.

[DL76] David Dobkin and Richard Lipton, "Multidimensional Searching Problems," SIAM J. Comput., 5:2 (1976), pp. 181-186.

[FB74] R. A. Finkel and J. L. Bentley, "Quad Trees: A Data Structure for Retrieval on Composite Keys," Acta Inf., 4 (1974), pp. 1-9.

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

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

[LW77] D. T. Lee and C. K. Wong, "Worst-Case Analysis for Region and Partial Region Searches in Multidimensional Binary Search Trees and Balanced Quad Trees," Acta Inf., 9 (1977), pp. 23-29.

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

[R76] Ronald Rivest, "Partial Match Retrieval Algorithms," SIAM J. Comput., 5:1 (1976), pp. 19-50.

Note Added August 18, 1978:

After submitting the final version of this paper, the author became aware of the references below which substantially overlap the results of this paper. In particular they discuss an idea very much like the BUILD procedure of Section 3, and also discuss considerably more general problems and approaches in a very elegant framework. The notion of bounded balance [NR73], applied in the present paper, seems to yield a better dynamic structure for range queries; it permits intermingled insertion and queries with query times faster by a factor of log n. Also, the present approach is more amenable to deletions.

References

Jon Bentley, Private Communication, August 1978.

Jon Bentley and Jerome Friedman, "Algorithms and Data Structures for Range Searching," Proceedings of the Computer Science and Statistics: 11th Annual Symposium on the Interface, (March 1978), pp. 297-307.

Jon Bentley and Michael Shamos, "A Problem in Multi-variate Statistics: Algorithm, Data Structure, and Applications," Proceedings of the 15th Allerton Conference on Communication, Control, and Computing, (Sept. 1977), pp. 193-201.