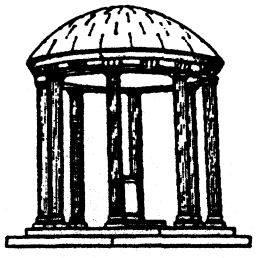
A Parallel Architecture for *k-d* Trees

TR88-026

May 1988

Geoffrey A. Frank
Donald F. Stanat



The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

A Parallel Architecture for k-d Trees

Geoffrey A. Frank
Research Triangle Institute, Research Triangle Park, N.C. 27709
Donald F. Stanat
Department of Computer Science, University of North Carolina
Chapel Hill, N.C. 27514

ABSTRACT

We describe a special purpose computer architecture for the parallel processing of queries, including associative searches, in a dynamic file. The architecture is a highly-parallel network of small processors of two types connected in a full binary tree network. Records are stored in the leaves of the tree; each leaf processor is responsible for records occurring within a rectangular solid part of the space. Queries and record updates are fed into the root of the tree. Internal nodes selectively direct each query and update to leaves so that each leaf sees only the information geometrically close to the records for which it is responsible. File updates cause a reorganization of the tree, which is accomplished in a manner that can accommodate either incremental or massive changes.

The architecture can be viewed as a hardware implementation of Bentley's k-d trees. The design is extensible and well-suited to implementation in VLSI.

1. INTRODUCTION

We address the problem of rapid searches and updates of a dynamic file with multidimensional keys. Examples of search operations of interest include range queries and nearest neighbor searches in multidimensional spaces. File update operations can include additions and deletions of records as well as changes of key values. Our approach is appropriate for applications in which queries and updates can be batched, so that the basic processing cycle consists of two phases:

Phase 1: Process queries

Phase 2: Update the file

We are interested in applications for which the processing required for each query is sufficiently demanding that processing must be done in parallel for many records, and where there is limited time available for updating the file.

We propose a special purpose massively parallel machine architecture whose degree of parallelism can be adjusted to a level appropriate to the particular problem instance. The architecture can be viewed as a hardware implementation of a dynamic k-d tree [Be75] for the appropriate dimension. These trees have been shown to be an excellent data structure for dealing with many problems in multidimensional space. Our architecture stores the records of a file in the leaves of a binary tree network. Queries and updates enter through the root and are directed to the proper subset of leaf nodes by the interior nodes of the tree network; the processing of queries and updates is performed in the leaves of the tree. This organization facilitates pipelining of queries and updates through the tree, and makes each leaf node responsible only for processing the information most likely to be relevant to its record.

The remainder of this report consists of the following sections. In Section 2 we describe k-d trees as proposed by Bentley [Be75], and a modification in which records are stored

This research was supported in part by the Office of Naval Research, contract N00014-86-K-0680.

in leaf nodes and the leaf nodes are all the same distance from the root. In Section 3 we describe the tree machine design, starting with a hardware implementation of a *k-d* tree for a static data set and culminating with an implementation that permits the tree to be dynamically reconfigured. In Section 4 we describe an application of the machine to the problem of tracking objects in 3-space and analyze the performance of the tree machine for this application. In Section 5 we discuss related work, and in Section 6 we give conclusions and describe future work.

2. K-D TREES

Bentley[Be75] proposed *k-d* trees as a generalization of binary search trees appropriate for multidimensional problems. Each record of a file is assumed to include *k* keys, any subset of which can be used in querying the file. Each record is stored in a (unique) node (interior or leaf) of a binary tree.

We will describe and use a variant of Bentley's design, where the records are stored only at the leaves. In this variant, each leaf contains a record R, and each internal node T of the *k-d* tree contains the following:

a. Pointers T.left to the left child and T.right to the right child of the node T. (Either T.left or T.right may be null.)

b. An integer T.axis in the range [*0...k-1*] specifying which of the *k* keys (axes) is to be used to direct queries in a search.

c. a *discriminator* T.disc in the key space of the key T.axis. Comparison of a search key with T.disc is used to direct queries in a search to the left subtree or the right subtree.

We will denote the *k* keys of a record R by R.*0* ... R.*k-1*.

The *k-d* tree is organized to satisfy the following invariant. (Here and elsewhere we will not distinguish between a node T, the subtree of which T is the root, and a pointer to T.)

Invariant: For every leaf node R and interior node T,

a. if R is in the subtree T.left, then R.(T.axis) < T.disc.

b. if R is in the subtree T.right, then R.(T.axis) > T.disc.

For the present, we assume that discriminators will be chosen to lie between the relevant key values of records; thus, equality cannot hold.

Although the restrictions are unnecessary, we will assume, as Bentley did, that if T is the root, then T.axis = 0 and if an internal node Q is a child of node P then Q.axis = P.axis + 1 *mod k*. This implies that all the nodes at any level in the tree discriminate along the same axis, that is, if P and Q are the same distance from the root, then P.axis = Q.axis.

Just as ordinary binary search trees effectively associate a segment of the linear key space with each subtree, *k-d* trees associate a *k*-dimensional rectangular region with each subtree. In a 2-dimensional key space, for example, the root node divides the space into a right and left part. The left child of the root will then divide the left part into a top and a bottom part, and the right child will divide the right part similarly. Note that the discriminator value used to divide the right part need not be the same as that used to divide the left part. Each of the four grandchildren of the root will then divide each of the four parts into two disjoint left and right parts. Thus each subtree in a *k-d* tree is naturally associated with a *k*-dimensional 'rectangular region' (possibly semi-infinite) of the key space. See Figure 1.

Note that ordinary binary search trees can be viewed as *1-d* trees; that is, all nodes

have the same value for T.axis. In general, *k-d* trees share the virtues and vices of binary search trees: the expected height of randomly built trees is logarithmic in the number of records in the file, so expected search and insertion times are logarithmic. On the other hand, deletions are awkward, and building trees that are guaranteed to be well-balanced is costly, since it requires determining an approximate median value along the appropriate axis for the subfile stored in each subtree. So while a tree structure has many signal virtues, it poses a number of problems for dealing with a dynamic file. These problems are handled in our machine architecture by devoting a part of each machine cycle to a reorganization of the tree; this approach is attractive because much of the reorganization can be done in parallel.

The architecture we propose is a hardware implementation of a special class of *k-d* trees. We require that all the leaves of the *k-d* tree be the same distance from the root, that each internal node with only one son stores the appropriate key value as its discriminator, and that all nodes with only one son occur at the bottom of the tree; these requirements are reflected in the following additional invariants:

Invariant: For every leaf node R and interior node T, 3. The depth of R is equal to the height of the tree.

4. if the subtree T has a single leaf node R, then T.disc = R.(T.axis).

5. if T has a single child, then every descendant of T is either a leaf or has a single child.

3. THE TREE MACHINE

3.1 The Machine Architecture

The tree machine (TM) is a full binary tree of small processors of only two types; see Fig. 2. We will call the interior processors T cells (which we will identify with the interior T cells of a *k-d* tree) and the leaf processors R cells. The TM has the nodes of a reconfigurable *k-d* tree embedded in its nodes. This makes it possible to direct queries only to a subset of the leaf processors of the machine. Each cell of the TM is associated with a rectangular region of the key space; this rectangular region contains all the records stored in leaf cell descendants of the cell. Moreover, the cells at each level of the tree partition the key space into disjoint rectangular regions. At the leaf level, each cell contains no records or a single record.

Each T cell is a small processor that is capable of receiving packets containing records, update information and queries from its parent node and child cells, performing simple comparisons and computations involving these packets, and then forwarding the packets to its child and parent cells. In addition t being able to send query responses from the R cells up into the tree, each T cell must also have the capacity to store two records so that records can be sent up into the tree by the R cells, sorted by the T cells, and arrive at the root in either increasing or decreasing order of a specified key. In order to play its role as an interior node of a *k-d* tree, each T cell also has storage for

a. An integer T.axis in the range [*0...k-1*] specifying which of the *k* keys (axes) is to be used to direct queries in a search.

b. a *discriminator* T.disc in the key space of the key T.axis.

Additionally, in order to participate in the balancing of the tree, T cell must contain

c. an integer T.balance equal to the difference between the number of records in the left subtree and the right subtree.

The embedding of a *k-d* tree in the tree of processors is done by first building a *k-d* tree of the height of the hardware tree, with all records stored at the leaves and all leaves the

same distance from the root. The tree is then mapped onto the hardware in the obvious way, preserving the left-right and parent-child relationships. In particular, each interior node T of the *k-d* tree is mapped to a T cell of the TM, and each interior node of the TM contains the appropriate *k-d* tree axis index T.axis and key value discriminator T.disc. All records that lie below a node T of the *k-d* tree are stored at R cells that are descendants of the corresponding T cell of the TM; those that have key values less than the discriminator are left descendants, while those with key values greater than or equal to the discriminator are right descendants.

3.2 Search Algorithms

The algorithms for searches in the TM are straightforward modifications of the corresponding searches in a *k-d* tree, except that the TM allows a search to proceed in parallel in different subtrees. Exact matches are easily handled; they cause a query to follow a unique path from the root to an R cell (if the search does not fail at a T cell). Partial matches are handled similarly except that interior nodes that have key values that are not being matched send the query to both sons. A range query (a search for all records with keys in specified ranges) requires that a query be sent into each subtree that has a non-empty intersection with the rectangular solid that is described by the set of key ranges. Each interior node that receives this query from its parent node may forward it to one, both or none of its child nodes. All leaf cells that contain records in the space (and possibly others) will receive the query; their responses can be pipelined up the tree.

The partitioning of the space makes it possible for each leaf processor to see only those packets that are located in nearby volumes of the partitioned space. If the space is partitioned in a fixed way, however, insertions, deletions and key changes of records in the space could lead to imbalance in the form of too many records being assigned to a subtree. By making the partitioning dynamic, it is possible to make each subtree responsible for only a limited number of records, and each R cell responsible for only one. This requires that the portion of space assigned to a subtree be able to grow or shrink or move, i.e., that the tree be reorganized. Reorganization of the tree requires changing the discriminator values of the interior nodes and moving records to maintain the invariant.

3.3 Update Algorithms

The basic pattern for file update processing is first to update the set of records, possibly violating the *k-d* tree invariant or the tree balance. The second step is to reorganize the tree so that the resulting tree satisfies the invariants and is balanced. We will treat three ways of updating a file: deletions, insertions, and changes of keys.

Deletions are easily handled at any time. We will call the child of a T cell *active* if that child has one or more R cell descendants that contain records. Thus, the left (right) child of an T cell is active if T.left (T.right) of the corresponding node of the *k-d* tree is non-null. When the record in an R cell is deleted, the R cell signals its parent and becomes inactive. When an T cell with only one active child receives such a signal, it propagates the signal upward; that is, if both children are now inactive, it sends the same signal to its parent. In any case, a distinct signal propagates from the R cell in which a record has been deleted to the root, modifying the balance factor T.balance of each T cell along the path. Note that deletions do not require a reorganization of the tree, although they affect the balance of the tree.

Adding records to a file can be done without reorganizing the tree only when empty R cells exist in locations that will permit the addition of the new records using the current discriminator values. In general, insertions will be a part of a larger file-update procedure that will require the interruption of query processing.

When file updates are batched, the first step of the update is to process deletions as

described above. The next step is to process changes of keys of records currently in the tree; these changes are made locally, possibly violating the invariant. The last step is to insert records to be added into the tree. Each new record is sent to an arbitrary empty R cell; this is done by the T cells using their balance factor T.balance so that the insertions will not adversely affect (and may improve) the balance of the tree. The next step is to restore the invariant by reorganizing the tree. In general, reorganization can be initiated any time the *k-d* tree invariant is violated or when the imbalance of parts of the tree becomes excessive.

Reorganization proceeds as follows:

Records in the left subtree are pipelined to the root, queued in decreasing order of the key R.*0*. Records in the right subtree are sent up to the root in increasing order of R.*0*. The root cell first re-establishes the balance of the tree by removing the proper number of records from one subtree and inserting them into the other, where they are then sent to leaf cells. Records are removed from the over-loaded subtree in their sorted order; that is, the order in which they arrive at the root.

Reorganization continues by next re-establishing the property that all records in the left subtree have R.*0* values less than all records in the right subtree. This is done by exchanging the two records at the heads of the queues at the root if the value R.*0* of the record in the left subtree is greater than that of the record in the right subtree. Records that were exchanged are then sent to R cells. Note that records are queued at the root in the order in which they will be transferred from one subtree to the other; the root exchanges records as long as necessary to re-establish that all records in the left subtree of the root have R.*0* values less than all records in the right subtree. When the root is finished exchanging records between its subtrees it can reset its discriminator to the value midway between the largest key value in the left subtree and the smallest key value in the right; this re-establishes the invariant for the root cell.

Finally, the remaining queued up records are sent back down the tree to arbitrary R cells, with each T cell directing the records so as to achieve the best balance possible. This completes the tree reorganization at the root level.

The tree reorganization continues by repeating the process (in parallel) for each of the sons of the root, except using the key R.*1*. This continues until all levels of the *k-d* tree have been reorganized, using the proper key at each level, and the discriminators of all internal nodes have been re-set.

This completes the machine cycle; a new set of queries can now be processed.

4. AN APPLICATION

4.1 Tracking Multiple Objects

An application of particular importance is multi-target tracking in 3 dimensions. In this application, a file of records called *tracks* is maintained. The file is dynamic, since the position of each track will probably change over time, and tracks may be added to or deleted from the file. The keys for the records are the predicted positions of the tracks; for a 3 dimensional tracking problem, each record has 3 keys. The tracks are updated at at regular intervals (which we shall denote by *t*, *t+1*, etc.) with a new set of observations of the objects being tracked. Each new set of observations contains update information that is used to correct the predicted positions of the tracks and to create predictions for the next time interval. The response time for the system is critical; each set of observations must be processed and all tracks must be updated before the arrival of a new set of observations. This application is most naturally handled by the TM by batching the updates so that the tree is reorganized once for each set of observations. Note that the reorganizations will not

be massive because the records will move slowly through the key space.

We assume that the updating of any track t requires only information about the observation that is nearest the predicted position for t. For the purposes of this paper, we will assume that the observation nearest to a predicted track location will lie within a cube of size *2r* centered at the position of the observation.

We now describe informally the operation of the machine. We assume the tracks from time *t* are stored in the leaves, and that predictions of their position at time *t+1* have been made. The machine cycle proceeds as follows:

1. Merge observations with the predicted track positions.

   Observations come into the tree through the root and are directed into subtrees by means of the discriminators stored in the internal nodes. Observations within distance *r* of a discriminator T.disc will be directed into both subtrees of a cell T. Under the assumptions we have made, merging a track with its associated observation can be done by its leaf cell, choosing the observation that is closest to the predicted position of its track. Although easily implemented, this criterion will not be sufficiently sophisticated in practice, where observations from sensors must be matched with tracks or used to create a new track, possibly with some confidence indicator. In a practical situation, in which observations may be matched with more than one track, and some tracks may not be matched with any observation, there must be some way of detecting and resolving conflicts among the local decisions made by the leaf cells, and the resolution must be in subtrees rather than leaves.

2. Update track descriptions.

   The information in each track is updated based on the information from new observations and associated with the track in the previous step. A corrected track position is computed, and predictions for the position of the track at the next time interval are made. New tracks are created for observations that did not match a track, and tracks which no longer represent an object in the field of view are eliminated.

3. Reorganize the tree.

   This part of the machine cycle is devoted to re-establishing the *k-d* tree property and rebalancing the tree, as described in Section 3.3.

   This completes the machine cycle; a new set of observations can now be processed.

4.2 Performance Analysis

The tree machine gets its speed from two sources: the *k-d* tree organization reduces the number of pairs of points that must be compared, and massive parallelism makes it possible to perform the many computations without a great time cost. Pipelining of values through the tree also plays an important role.

The following is an informal analysis of the expected time complexity of the machine cycle for the tracking application. We assume a set of n observations (queries) and a hardware tree of height $\log n$; thus the number of tracks is approximately the same as the number of observations. But if the number of tracks is twice the number of observations, then the height of the tree increases to $\log n + 1$, which has no effect on the asymptotic analysis.

1. The first part of the machine cycle requires $O(n)$ time to bring in n observations through the root of the tree and an additional $O(\log n)$ time for the last of these observations to reach the appropriate leaf processors. Thus query input requires $O(n+\log n)$ time.

2. The second part of the cycle updates the tracks with the information in the ob-

servations. Because the *k-d* tree sends observations only to those leaf processors that contain nearby tracks, the expected time for these computations is a function of the expected value of the number of observations processed by any leaf processor, which we assume is bounded by a constant. Thus, this stage of the computation will require an expected time of $O(1)$, although the coefficient for this part of the cycle may be much larger than that for the first part of the cycle.

3. The third part of the cycle reorganizes the tree. Full reorganization of tree each cycle as described will take place in log n steps, one for each level of the tree. The most costly step is likely to occur at the root, where in the worst case n records can be moved from one subtree to the other. But at the next level, only $n/2$ records can be moved at the root of each subtree, and in general, at distance t from the root, only $n/2^t$ exchanges can be made in each subtree. Thus the entire time for exchanges is $O(n)$. The time for sending points up to the appropriate tree node and back down is $O(\log n)$ for the reorganization of each level. Thus reorganizing the entire tree requires $O(n)$ time for point exchanges and $(\log n) * O(\log n)$ or $O((\log n)^2)$ time for sending points up and down the tree. It follows that reorganizing all the levels of the tree requires $O(n + (\log n)^2)$ time. This cost dominates the asymptotic complexity of the machine cycle.

For the tracking problem, this analysis of the reorganization time is very pessimistic, principally because the number of points to be moved between subtrees during any cycle is likely to be small.

5. RELATED WORK

The architecture we describe owes much of its character to the design of the FFP machine, a small-grain general purpose computer architecture proposed by Magó [Ma79a,Ma79b,MM84]. That machine, like the TM, is a static hardware tree network that is reconfigured during each machine cycle into a collection of subtrees appropriate to the current computational task.

Bentley and Kung [BK79] examined tree architectures for associative searching problems. For many applications, due to the cost of file updates, the TM architecture may not be better than that proposed by Bentley and Kung. Their approach also stored records in leaves and provided for the parallel processing of matched queries and records. In their design, queries are broadcast to all leaves from the root and all leaves process all inputs; thus the processors holding the records are responsible both for finding which queries match the record, and for processing the matched queries and record. In the TM, the two tasks of comparison of keys and processing queries are delegated to different processors. Because key comparisons can be pipelined and done in parallel in the TM, this design appears to have an advantage in situations in which the time to match a key is substantial in comparison to the time required to forward a packet, formulate a response to a query or update a record. Also, since the comparison of the keys is pipelined, the query processing rate in the TM can be independent of the number of keys.

6. CONCLUSIONS AND FURTHER WORK

We have described an implementation of a modification of Bentley's *k-d* trees with a special-purpose parallel processor. This architecture provides for the rapid processing of queries and can gracefully accommodate major reorganizations of the tree. A straightforward analysis of the algorithms indicates that when the number of queries processed between file updates is of the order of the number of records in the file, then the expected reorganization and the search times are both $O(n + (\log n)^2)$.

In the application of multi-target tracking in 3 dimensions, each new set of observations usually contains update information for each record (track), and rapid response and up-

dating is crucial. This application is an appropriate candidate for the TM because updates (observations) can be batched and the tree reorganized once for each set of observations, but the machine cycle must be fast enough to process the incoming update information on the set of records and reorganize the tree before new observations arrive. Note, however, that the reorganizations will not be massive because the records will move slowly through the key space.

Future work will explore the design of the TM more carefully as well as variations of the machine. This will be done in conjunction with consideration of a more realistic version of the tracking problem, where the number of objects changes, objects are endowed with different (observable) properties, etc. A matter of particular concern is coordination of decisions; in the machine as we described it, identification of new points with old ones is done independently by each R cell processor; that is, the global solution is the union of many independent local solutions. In practice, of course, this is not acceptable, but additional communication can provide the means for processors to cooperate in order to arrive at a satisfactory global solution.

Additionally, serious development would require looking into ways of overlapping various parts of the machine cycle by providing a network with sufficiently rich communication and processing paths.

There exist a number of variations on the architecture that may merit further investigation. Many of the possible variations would have a substantial effect on a working system, but would not affect the asymptotic analysis. For example, input speed could be increased by allowing internal nodes at some level in the tree to be input ports for queries. This would reduce the time required for the input by reducing the bottleneck at the root; it would also change the load balance between internal and leaf node processing. Another way of changing this balance is to increase the number of records stored in each leaf cell. Storing additional records in a leaf cell can also alleviate some difficulties associated with using this architecture for certain applications; for example, disambiguation of observations in a tracking problem requires some communication among nearby tracks. Simulations of some of these approaches should provide some insight into some of the tradeoff issues that are not addressed by our asymptotic analysis.

Other questions concern strategies for avoiding reorganizations of the tree. There are two possibilities: allowing the discriminators to become inexact (that is, loosen the invariant that characterizes the tree), and allowing the tree to become unbalanced. Both possibilities would increase the potential cost of matching queries and records, but could save time by reducing the frequency of tree reorganization.

Other application areas might utilize other variants of the architecture. It might be of particular interest in applications that involve data bases that are not passive, i.e. where the processing of matched queries and records can generate new queries. If the new queries involve records in the same locality, then the ***k-d*** tree organization could confine the processing of the new queries to subtrees, allowing more parallelism. This would be an advantage over the Bentley and Kung architecture, which requires that all new queries be sent to the root. A detailed treatment of the tracking application would make use of this opportunity for exploiting locality in processing, as do object-oriented data bases, and frames in AI applications, where a match of a query with a frame may cause a demon to be invoked.

BIBLIOGRAPHY

[Be75] Bentley, Jon L.: "Multidimensional Binary Search Trees Used for Associative Searching". Communications of the ACM, 18, No. 9, Sept., 1975.

tracking problem requires some communication among nearby

[BK79] Bentley, Jon L. and H.T. Kung: "A Tree Machine for Searching Problems". In Carnegie-Mellon Technical Report CMU-CS-79-149: "Two Papers on a Tree-Structured Parallel Computer."

[Ma79a] Magó, G.A.: "A Network of Microprocessors to Execute Reduction Languages" (Two parts). International Journal of Computer and Information Sciences 8, 5 (1979), 349-385; 8, 6 (1979), 435-471.

[MM84] Magó, G.A. and D. Middleton: "The FFP Machine- - -A Progress Report". *International Workshop on High-Level Computer Architecture 84*, Los Angeles, California, May 23-25, 1984.

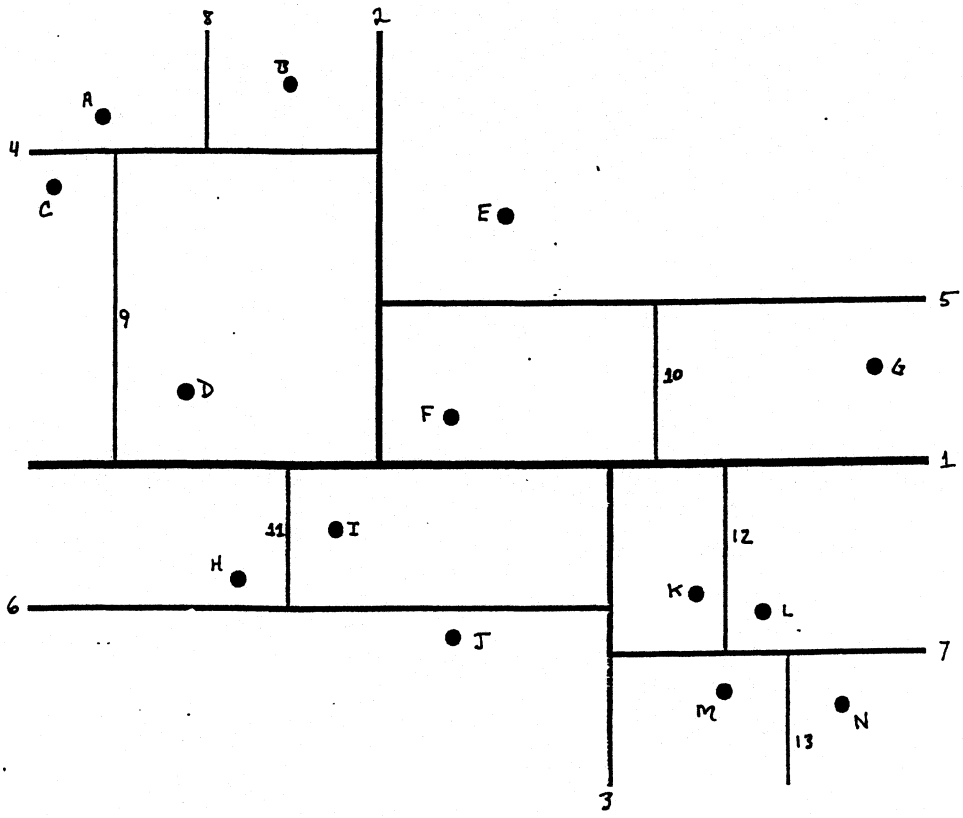


Figure 1: A partitioning of a two dimensional key space by a *k-d* tree

The file has fourteen records. Each discriminator of the *k-d* tree is represented by a horizontal or vertical line segment. The boldness of the line segment indicates its depth in the *k-d* tree, with the boldest line being the discrimination made by the root node. Discriminator lines are indexed for association with the nodes of the machine shown in the following figure. Note that the size of the regions reflects the local density of records in the key space.

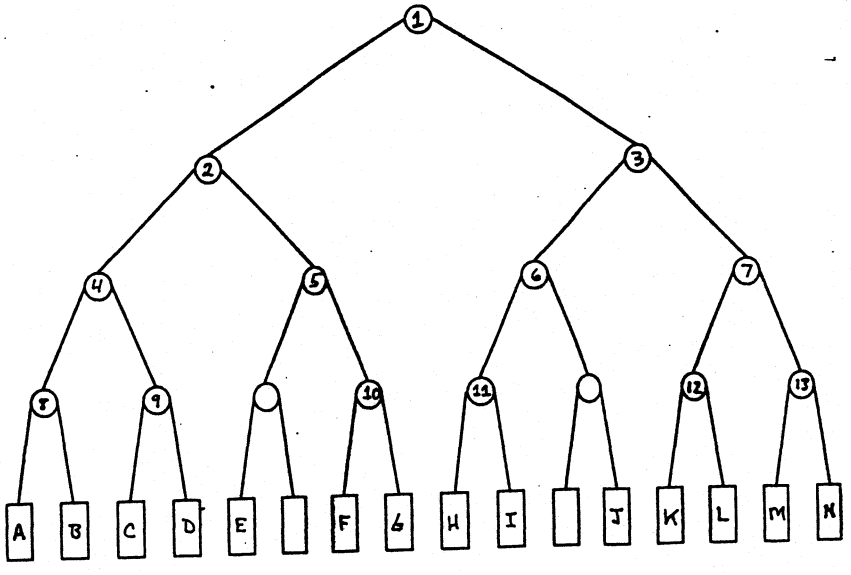


Figure 2: A TM (Tree Machine)

The leaf nodes of the tree are L cells; the non-leaf cells are T cells. The nodes are labeled by according to the previous figure, with discriminators being associated with T cells and records with L cells.