Multidimensional Data Structures: Review and Outlook

S. SITHARAMA IYENGAR
N. S. V. RAO

Department of Computer Science
Louisiana State University
Baton Rouge, Louisiana 70803

R. L. KASHYAP[†]

Department of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

V. K. VAISHNAVI

Department of Computer Information Systems
Georgia State University
University Plaza
Atlanta, Georgia 30303

[†] Partially supported by NSF under grant IST 8405052

ISBN 0-12-012127-1

1. Introduction

The design and analysis of data structures and algorithms is an important area of computer science. In recent years there have been several significant advances in this field, ranging from the binary search algorithm to the startling discovery of the polynomial algorithm for the linear programming (LP) problem. These results have produced efficient and optimal solutions to many real-life problems. As a consequence, much interest has been generated in the field of data structures and algorithms, which have blossomed into the forefront of computer science research.

Searching an ordered collection of data is a standard paradigm in the area of algorithms. The problem of search in a single dimension has been handled traditionally using the data structures of binary trees, multiway trees, hash tables, etc. In many applications arising in information retrieval, computational geometry, robotics, and statistics, the central problem is, however, one of determining the points falling in specified range limits in different dimensions. These problems have spurred an enormous interest in the design and analysis of efficient multidimensional data structures. In addition, the multidimensional data structures address themselves to a larger class of problems than single-dimensional data structures. See, for details, Horowitz and Sahni (1976). But the multidimensional data structures are not natural extensions of single-dimensional data structures. Instead, they are more complex and problem oriented. For example, consider the balanced binary tree data structure. The cost of constructing such a data structure is $O(N \log N)$, where N is the number of data points in the structure. The corresponding storage cost is $O(N)$. This data structure can answer complete match and range queries in $O(\log N)$ and $O(\log N + K_1)$ time respectively, where K_1 is the number of qualified data points that satisfy the query. There are no multidimensional data structures with similar performace.

A number of multidimensional data structures such as k-d trees, quadtrees, range trees, polygon trees, quintary trees, multiple attribute tree (MAT), balanced multidimensional and weighted trees, etc. have been studied recently for range and related search problems.

A k-dimensional search tree is a k-dimensional generalization of a search tree. It can be used for items that are k-tuples. When the (k-dimensional) data items have weights (positive real numbers) associated with them, the data items are called weighted items. A k-dimensional tree for such weighted items is a weighted tree of k-dimensions. In this paper we will also explore the relationship between weighted trees and balanced multidimensional trees.

1.1 Organization and Scope of the Paper

The focus of the initial part of this paper is to present and analyze simple and efficient balanced multidimensional tree structures that are useful for real time applications. The usefulness of such structures for very large data and concurrent environments is an important concern of future research in multidimensional data structures. A major theme of this part of the paper is the presentation of a paradigm for generalizing a balanced tree structure to multidimensions, along with a set of strategies for supporting structure violations in the generalized structures. The later part of the paper elaborates on various aspects of multiple attribute trees.

Section 2 presents an overview on performance of various multidimensional data structure. Section 3 describes balanced multidimensional and weighted trees as well as their applications in physical database organization, information retrieval, and computational geometry problems. Section 4 reviews the literature for generating a balancing concept for multidimensional trees. Section 5 describes a paradigm that models the available balanced and semi-balanced multidimensional and weighted structures. These paradigms are expected to guide future research in this area. Section 6 describes multiple attribute tree structures for partial match, complete match, and range queries. Also discussed is a comparison of the performance of the MAT structure with other multidimensional structures. Section 7 discusses paradigms for modeling the salient treatment of multiple attribute trees. Finally, Section 8 presents some conclusions.

2. What is a Multidimensional Data Structure?

A datum or data element is said to be *single-dimensional* if it is characterized by a single unique attribute or key value. Examples for single-dimensional data structures are balanced binary trees, B-trees, AVL-trees, hash tables, multiway trees, etc.

A data element is said to be *multidimensional* if it is characterized by a number of attributes. A multidimensional data element specified by k attributes can be imagined to be a "point" in a k-dimensional space. Examples for multidimensional data structures are k-d trees, quadtrees, multiple attribute trees (MAT), multidimensional B-trees, balanced multidimensional and weighted trees, etc. A multidimensional problem calls for a efficient data structure that organizes these points in such a way as to result in efficient access. The problem of retrieving all the records satisfying a query involving a multiplicity of attributes, known as *associative retrieval*, is a major issue in physical database organization. The factors which affect the performance of physical data organization are logical data characteristics, query complexity and device parameters, and it has been demonstrated that no single index structure can be optimal under all circumstances. For more on this see Kriegel (1981).

The performance of a multidimensional data structure is measured in terms of the following three quantities (We assume N points, each having k attributes, in our discussions):

- $P(N)$, *preprocessing cost*, which is the time taken for constructing the data structure;
- $S(N)$, *storage cost*, which is the amount of storage needed to store the data structure;
- $Q(N)$, *access cost*, which is the time taken for accessing the data structure as per the requirement. This can be query cost when answering a query about the data. In general, the cost embedded in this can be attributed to insertion, deletion or any other modification.

A *static* data structure is constructed only once initially. Queries can be answered in this data structure, but no insertions and deletions are supported. In a *dynamic* data structure, the insertions and deletions are supported dynamically. As will be shown in the later sections, maintenance of the data structure in the optimal form of a static one is a very difficult task. This is mainly because the insertions and deletions will, in general, destroy the balanced nature of the data structure that can be obtained in the static case. The dynamic data structures are indispensable for a large number of applications.

We now present some multidimensional data structures that arise frequently in several applications. The vast amount of literature and the varied nature of the proposed data structures prohibits an exhaustive treatment here. Therefore, this paper covers, in detail, balanced multidimensional and weighted trees, multiple attribute trees (MAT), and to a lesser extent other multidimensional data structures. Paradigms for modeling balanced multidimensional and weighted trees as well as MAT are presented in detail in section 7.

2.1 A Classification of Multidimensional Search Problems

One important class of multidimensional problems is the *search problem.* Let T_1, T_2, T_3 be sets of queries, elements, and answers respectively. A searching problem P on T_1, T_2, T_3 is a function

$$P : T_1 \times 2^{T_2} \rightarrow T_3.$$

The searching problem P takes a query object $t_1 \in T_1$ and a domain set $t_2 \in 2^{T_2}$ and produces an answer $t_3 \in T_3$. By assigning different properties to T_1, T_2, and T_3, the different classes of searching problems are obtained.

In a *member searching* problem $T_1 = T_2$ and $T_3 = \{\text{yes, no}\}$. The multidimensional searching problem can be converted to a single-dimensional one, by concatenating all attributes to form a single key. Hence, the standard approaches for single-dimensional cases, such as in Aho, Hopcroft, Ullman (1974) and Knuth (1974) can be applied to obtain $Q(N) = O(k \log N)$, $P(N) = O(kN \log N)$, $S(N) = O(N)$. The same complexity estimates can be obtained for searching for the r^{th} element or for finding the rank of an element in an ordered set as in Overmars (1984).

Another important searching problem is the *range search.* In this problem, T_2 contains points from k-dimensional space, and each $q \in T_1$ specifies a range for each dimension. T_3 is a subset of T_2 and contains the points that satisfy q. The point quadtree proposed by Finkel and Bentley (1974) and Lee and Wong (1977) has the complexity estimates of $Q(N) = O(N^{1-1/k} + K_1)$, $P(N) = O(N \log N)$, $S(N) = O(N)$, where K_1 is the number of points that satisfy the query. A point quadtree for a given set of points is a 2^k-ary tree, which recursively divides the space into 2^k quadrants. Although the worst-case time is not very impressive, in practice the quadtree is shown to be efficient in many practical problems. Details can be found in Finkel and Bentley (1974), Bentley and Stanat (1975) and Alt *et al.* (1981).

An important data structure for solving the range searching problem is called the *k-d tree.* This data structure is introduced by Bentley (1975). A k-d tree is a type of binary tree built as follows: A data point is taken as the root node; the data space is divided into two subsets based on the first attribute value; each of these sets is represented as a subtree of the root; each subtree is split based on the second attribute value. This process is carried out recursively on the subsets. After splitting with respect to the k^{th} attribute, the first attribute is considered again. Bentley (1975) and Lee and Wong (1977) solve the range searching problem with the following performance measures: $Q(N) = O(N^{1-1/k} + K_1)$, $P(N) = O(N \log N)$, $S(N) = O(N)$. Mehlhorn (1984b) discusses dd-trees which are very similar to k-d trees. At each step the data space is divided into three subsets, in contrast with two for the k-d tree. The first subset contains all the points whose corresponding attribute value is less than that of the root. The second subset contains all the points that have the same attribute value as the root. The third subset contains all the points that have the attribute value greater than that of the root. Both structures have similar characteristics.

There is another data structure for the range query which has a better worst-case query cost at the expense of high preprocessing and storage costs. This data structure has been independently proposed by various authors, and each differs from the other only slightly in the structural aspects. The performance measures of this data structure are $Q(N) = O(\log^k N + K_1)$, $P(N) = O(N \log^{k-1} N)$, $S(N) = O(N \log^{k-1} N)$. The basic data structure that underlies all these versions, referred to here as a *range tree*, can be defined as follows: For the single dimensional case the range tree reduces to a balanced binary leaf search tree in which the points are stored in the leaf nodes. Moreover, the leaves are linked in that order. To perform a range query, we search for the limits. Then, the points that lie within the range limits are retrieved using the linked list structure. A k-dimensional range tree consists of a balanced binary leaf search tree T in which points are stored and ordered according to the first attribute. To each internal node α, a $(k-1)$-dimensional range tree T_α is associated. This subtree rooted at α takes into account only the second to kth attributes for all points of the data. The basic underlying data structure here is a single-dimensional binary search tree. Conceptually, one can think of using the other single dimensional data structures such as the AVL-tree, B-tree, 2-3 tree, BB[α] tree, etc, as the underlying data structure. The order estimates for the range query remain the same, but the exact number of steps for a particular query differs. Willard (1979b) uses B-tree, and the corresponding multidimensional data structure is called the *super B-tree.* Lueker (1978) uses BB[α] tree as the underlying data structure, and the multidimensional version is called the *d-fold tree.* The *quintary tree* of Lee and Wong (1980) also falls into this type. They use a five-way tree as the underlying structure. The other single-dimensional data structures can also be used to obtain new multidimensional data structures. However, such extensions do not seem to promise more efficient structures than the existing ones. Willard (1985) and Hart (1981) tighten the query cost to $Q(N) = O(\log^{k-1} N + K_1)$ using a clever technique. Willard (1985) refers to this data structure as a *k-fold tree.* A number of other data structures such as *overlapping k-ranges, non-overlapping k-ranges*, etc. are proposed. These structures illustrate the different trade-offs between the query time, preprocessing time, and the storage cost. The performance measures for the overlapping k-ranges are: $Q(N) = O(\log N + K_1)$, $P(N) = O(N^{1+\epsilon})$, and $S(N) = O(N^{1+\epsilon})$, for $\epsilon > 0$. The non-overlapping k-ranges have performance measures as follows: $Q(N) = O(N^{\epsilon} + K_1)$, $P(N) = O(N \log N)$, and $S(N) = O(N)$. The reduction in one of the costs is reflected as an increase in the other costs. Bentley and Friedman (1979) gives an excellent survey on the data structures for the range search. More on the range and related issues can be found in Overmars (1984), Mehlhorn (1984b), Willard (1985) and Willard and Lueker (1985).

Another often encountered searching problem is the *nearest neighbor searching* problem. This problem consists of finding a point in T_2 that is the nearest to a query point $x \in T_1$, with respect to some metric. This problem frequently arises in database systems. This problem is shown to have the solution given by $Q(N) = O(\log N)$, $P(N) = O(N \log N)$, and $S(N) = O(N)$ for the two-dimensional case. The details can be found in Overmars (1984), Shamos and Hoey (1975), Shamos (1978) and Kirkpatrick (1983).

The multidimensional data structures such as the k-d trees, quadtrees, range trees, etc., are comparison-based data structures. There is another class of multidimensional tree structures based on the digital search of nodes in the tree, (Knuth (1974)). The nodes in the comparison-based structures are formed based on the result of a comparison between two elements. In the digital-based tree structures, the nodes are formed based on the values of the elements. The class of multidimensional tree structures based on the digital techniques includes *modified doubly chained trees* of Cardenas and Sagamang (1974), *binary search tree complexes* (BST-complex) of Lien *et al.* (1975), *multidimensional B-tree* (MDBT) of Ouksel and Scheuermann (1981), *kB-trees* of Gotlieb, and Gotlieb (1978), the *kB^+-tree* of Kriegel and Vaishnavi (1981a), and the *multiple attribute trees* (MAT) of Kashyap *et al.* (1977), Gopalakrishna and Veni Madhavan (1980), and Rao *et al.* (1984). The main difference between the comparison and the digital multidimensional tree data structures lies in the way the data is processed in building the data structure. For comparison-based methods the multidimensional space is recursively divided into smaller regions of the same dimensionality. In the digital-based structures the dimensionality is recursively reduced by one. This distinction is not applicable in the single-dimensional case, but is a vital factor in deciding the performance of multidimensional data structures. In the comparison data structures the search is recursively carried out in the smaller regions of the same dimensionality, whereas in the digital data structures the search is recursively carried out in the space of reduced dimensionality. This means that there could be applications that are naturally suited to one of the two types.

The area of computational geometry is rich in multidimensional problems and data structures. This emerging area has found many applications in various disciplines such as pattern recognition, operations research, computer graphics, robotics, etc. Computational geometry principally deals with the computational complexity of geometric problems. The five major areas in this field are convex hull, intersections, searching, proximity finding, and combinatorial optimization. The search problems discussed in this paper are related to the search and proximity problems of computational geometry. Several generic algorithms, such as divide and conquer, incremental construction, plane sweep, locus, geometric transformations etc. have been developed for various problems. For details on computational geometry problems see Mehlhorn (1984b), Edelsbrunner (1983, 1984), Overmars (1984), Preparata and Shamos (1985), and Shamos (1978).

In the area of information retrieval and management, a number of multidimensional data structures have been studied. The *multidimensional trees* and the *multidimensional directory* of Orenstein (1982) are some of the important ones. The structures such as *inverted lists* and *multi lists* of Liou and Yao (1977) are examples of extensions of the concepts of lists to multiple dimensions. The *grid file* of Nievergelt *et al.* (1984) is another multidimensional data structure that supports multikey access. This structure is proved to be superior to conventional structures such as inverted files and multi lists in terms of the adaptation to the dynamic environments. Unlike the earlier mentioned data structures, the grid file deals with the storage of the directory on the secondary memory. There is also a pedigree of data structures for the management of the directory on the secondary memory. The generalization of hash functions by Rothnie *et al.* (1974) describes a multikey hash file structure. But finding a hash function that optimizes the query, insertion, deletion, collision resolution, and storage costs is very difficult (see Lum (1970)) and problem dependent. Lum (1970) describes *combined indices* that are formed by concatenating several attribute values. The *compressed bit maps* are based on the bit-encoded inverted files. The *transposed file* organization of Batory (1979) and Barnes and Collens (1973) also makes use of compression techniques in providing multikey access. The generalization of *tries* to support multikey access can be found in Orenstein (1982) and Tamminen (1982). Casey (1973) describes a complex-tree-based multikey access structure, which uses *superimposed* coding to characterize the records below the nodes of the tree. This approach is complex, and a more practical approach based on similar concepts can be found in Pealtz *et al.* (1980). The multidimensional directories suggested by Liou and Yao (1977) provide an efficient way of multikey access, using priority ordering of the attributes. The *interpolation*-based index maintenance of Burkhard (1983) is a multidimensional extension of the linear hashing of Litwin (1980), and is related to the extendible hashing of Fagin *et al.* (1979). These multikey access methods for the data stored in the secondary memory provide efficient solutions for many practical applications. Since many of these approaches rest on identifying a suitable hash function, they are not very amenable to rigorous theoretical analysis.

One important aspect of the multidimensional data structures is *dynamization.* Ideally, the term "dynamization" refers to the process of maintaining the data structure to ensure the performance of the static data structure under the dynamic environment of insertions and deletions of records. Normally, the dynamization of the multidimensional data structures is much more involved than that in single dimensional data structures. This problem becomes further complicated when the data directory resides on the secondary memory. This is because, in many cases, the insertions and deletions destroy the clustering of the data into secondary memory pages. Many data structures are dynamic when implemented in memory that supports random access. But the considerations become markedly different when the data structure is stored on the secondary memory. Again, the requirements of dynamization vary considerably depending on the application. The general framework for dynamization is introduced by Bentley (1979a, 1980) for a certain class of problems called *decomposable* problems. In Bentley and Saxe (1980) the general concepts of static-to-dynamic transformation of data structures are discussed. The notions of *local, partial,* and *global* rebuilding of data structures are discussed by Overmars (1984) for the purpose of dynamization. The varied nature of the dynamization makes the generalized treatment difficult. For details of a simple dynamization method see Rao, Vaishnavi and Iyengar (1987).

3. Balanced Multidimensional and Weighted Trees

3.1 Background

Vaishnavi (1984, 1987) recently initiated a new direction of research for developing balanced self-organizing and multidimensional search trees that support various operations through algorithms that are relatively simple to understand and implement. This research involves generalizing balanced search trees using a new paradigm for supporting a structure violation by an EQSON-subtree.

In this section, we will discuss certain basic concepts and issues as well as review related real-time applications for balanced multidimensional and weighted trees. The paradigms described in this section provide a methodology for generalizing balanced search tree structures to multidimensions.

3.1.1 k-dimensional (Search) Trees

A *k-dimensional (search) tree*, as illustrated in Fig. 1, is a natural data structure for *k*-dimensional data. It is related to TRIEs developed by Fredkin (1960), digital search trees by Knuth (1974), and lexicographic trees by Sleator and Tarjan (1985). It can also be interpreted as a *k-dimensional* generalization

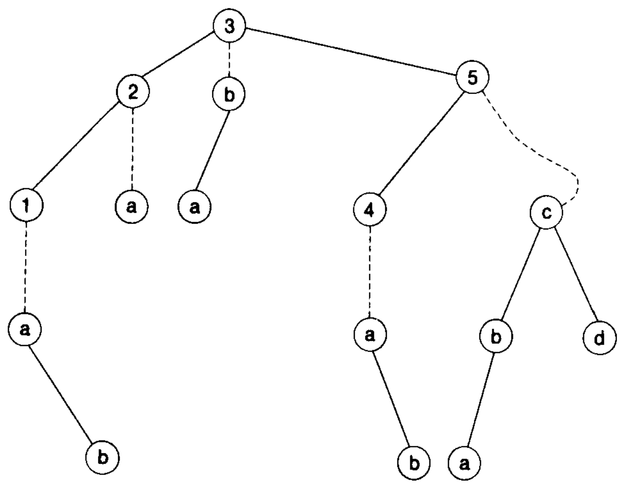


FIG. 1. A 2-dimensional search tree for the following set of 2-dimensional data items: {(1, a), (1, b), (2, a), (3, a), (3, b), (4, a), (4, b), (5, a), (5, b), (5, c), (5, d)}. For the data item (1, a), the first attribute is "a" and the second attribute is "1".

of a (one-dimensional) search tree. In such a (ternary) tree, as we have the following (Bentley and Saxe (1979): The root stores x_k, a k-dimensional key, the k^{th} attribute of a data item. The left structure of the root is a k-dimensional tree for those data items whose k^{th} attributes are less than x_k. The right subtree of the root is a k-dimensional tree for those data items whose k^{th} attributes are greater than x_k. The middle subtree of the root, an *EQSON-subtree*, is a $(k - 1)$-dimensional tree for the $(k - 1)$-dimensional projection data items obtained by omitting x_k, the identical k^{th} attribute of data items. A k-dimensional (binary) leaf tree (useful for sequential processing) and a k-dimensional multiway tree (useful for very large data) can be similarly defined. The worst-case performance per operation of a k-dimensional tree is rather poor unless its structure is properly constrained. A *balanced k-dimensional (search) tree* is a constrained tree structure such that its height is at most logarithmic, i.e., $O(\log n + k)$, which is optimal as shown by Bentley and Saxe (1979) and Hirschberg (1980), and is a flexible enough tree structure such that it can be updated in logarithmic worst-case time per operation.

3.1.2 Weighted Trees

When the (k-dimensional) data items have *weights* (positive real numbers) associated with them, the data items are called *weighted items*. A k-dimensional tree for such weighted items is a *weighted (k-dimensional) tree*. An obvious interpretation of the weight of a data item is that it represents the probability with which the item is accessed. A weighted tree, with weights representing access probabilities, that dynamically adapts itself to the (possibly non-uniform) access pattern is a *self-organizing tree.* There can be other interpretations of weights too, such as the one used in the application of the D-trees developed by Mehlhorn (1984a) for the dynamization of interval trees (see Edelsbrunner (1980a), McCreight (1980)) and segment trees (see Bentley (1977)).

In a self-organizing tree, two additional operations, namely *promote* (increase the weight of an item by a certain amount) and *demote* (decrease the weight of an item by a certain amount), become relevant. The best that can be expected from a self-organizing (k-dimensional) tree is that its worst-case access time is "logarithmic," i.e., $O(\log W/w_i + k)$, where w_i is the weight of item i at a certain instant of time and W is the sum of the weights of all items at that time. For a broader treatment on this see Gueting and Kriegel (1981) and Mehlhorn (1984a). The worst-case time for other operations can at best be logarithmic. A *balanced weighted (k-dimensional) tree* can be accessed and updated in logarithmic worst-case time per operation.

3.2 Multidimensional Trees and Weighted Trees

Weighted trees and multidimensional trees are related to each other. One can start from an efficient weighted one-dimensional tree structure and use it for constructing efficient multidimensional or even efficient weighted multidimensional tree structures by using it appropriately for implementing nodes of TRIEs developed by Fredkin (1960). Alternatively, one can start from an efficient $(k + 1)$-dimensional tree structure and use it for constructing an efficient weighted k-dimensional tree structure by letting the lowest level subtrees used as virtual subtrees representing weights of items (see Gueting and Kriegel (1981)).

3.2.1 Worst-Case vs. Amortized Behavior

Many applications require the use of a sequence of operations. In such cases, it is not important that the *worst-case* time per operation be small; it suffices to have the operation times *amortized* over a worst-case sequence of operations to be small (see Bent *et al.* (1980) and Huddleston and Mehlhorn (1982)). The amortized time per operation is also a good measure of the average performance of a data structure as shown by Bentley and McGeoch (1985). There are situations, however, where it is important that the worst-case time per operation be small. A real-time application is an example of such a situation. Also, a small worst-case time per operation sometimes gets translated into a still smaller amortized time per operation. Additionally, a good worst-case performance of a data structure that is the basis for a more complex data structure can sometimes be important for a good amortized behavior of the latter (see, for example, dynamic interval and segment trees based on D-trees Mehlhorn (1984b)). Here, we will mainly focus on balanced multidimensional and weighted trees, which are worst-case efficient data structures. The Splay tree, presented by Sleator and Tarjan (1983a), is an elegant data structure which guarantees amortized-case efficient behavior for access as well as update operations.

3.3 Applications

Balanced multidimensional and weighted trees have applications in such areas as (a) physical database organization, (b) information retrieval, (c) self-organizing file structures, and (d) computational geometry. They may be used as independent data structures (see Bent *et al.* (1985), Gueting and Kriegel (1980) and Gueting and Kriegel (1981)), or as underlying schemes for complex data structures (Mehlhorn (1984b), Sleator and Tarjan (1983b), or may be used in a suitably enhanced form in complex applications (Kriegel (1984), Kriegel and Vaishnavi (1981c), and Scheuermann and Ouksel (1982)). Let us elaborate on these applications.

3.3.1 Physical Database Organization

The problem of retrieving all the records satisfying a query involving a multiplicity of attributes, known as multikey retrieval or associative retrieval, is a major concern in physical database organization (Kriegel (1984)). The expanding use of database systems for executive decision-making demands a physical organization that can support fast responses to associative queries as well as efficient updates. The available file structures (e.g., the inverted file of Knuth (1974), the grid file of Nievergelt *et al.* (1984), the MDBT organization of Scheuermann and Ouksel (1982), the *k*B-trees of Gueting and Kriegel (1980) and Kriegel (1984)), suffer from certain deficiencies, especially if the database is highly dynamic, and associative queries such as range queries and partial-range queries are important.

A balanced multidimensional multiway tree structure, suitably enhanced with additional pointers as in the MDBT organization by Scheuermann and Ouksel (1982), can serve as a dynamic file organization for associative retrieval. The performance of *k*B-trees of Gueting and Kriegel (1980) and Kriegel (1984), a balanced multidimensional multiway tree structure (further discussed in Section 5), has been compared (see Kriegel (1984)) with the inverted file (Knuth (1974)), the grid file (Nievergelt et al. (1973)), and the MDBT-organization (Scheuermann and Ouksel (1982)). The inverted file has been found to have the worst performance; while the grid file has been found to have the best performance in certain respects, *k*B-trees have been found to have the best performance in certain other respects.

3.3.2 Information Retrieval

A balanced multidimensional (search) tree can serve as a space-efficient implementation of a TRIE developed by Fredkin (1960) or digital search tree of Knuth (1974), a data structure commonly used in information retrieval (see Fig. 2). If each node of a TRIE for a set of English words is implemented as an array, then a word can be retrieved, inserted, or deleted in time proportional to the length of the word. But this implementation is quite wasteful in storage. One the other hand, an implementation based on a balanced multidimensional tree structure occupies storage proportional to the total number of characters in words and supports retrieval and update in time proportional to the length of the word plus log *n* where *n* is the total number of words. This performance is optimal as demonstrated by Hirschberg (1980) if comparison between characters is a unit operation.

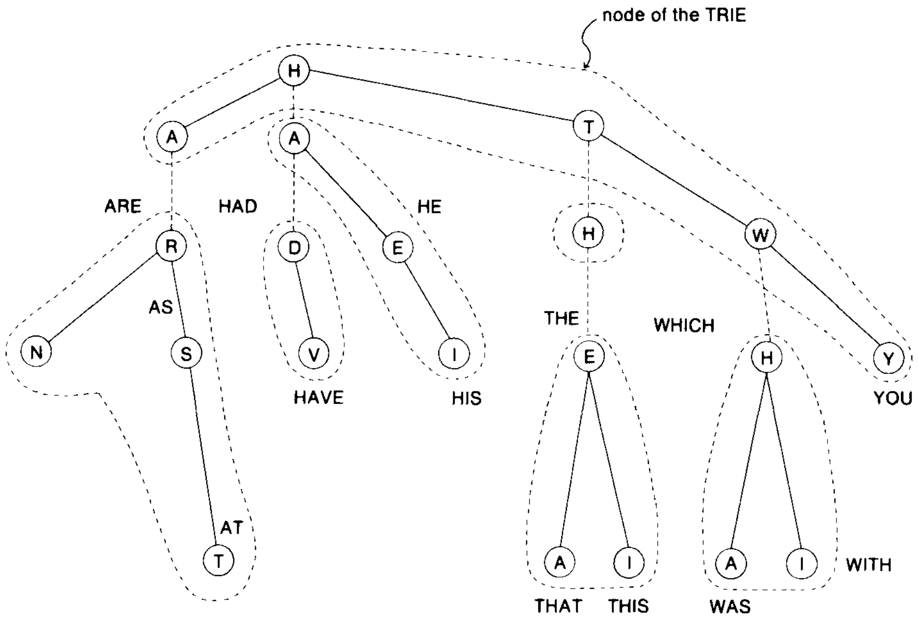


FIG. 2. A TRIE for the English words {AN, ARE, AS, AT, HAD, HAVE, HE, HIS, THAT, THE, THIS, WAS, WHICH, WITH, YOU}.

3.3.3 *Self-Organizing File Structures*

If the weights of data items are interpreted as frequencies with which they are accessed, then a balanced weighted tree structure is a dynamic self-organizing file structure which adapts itself to usage. It stays in a nearly optimal form with respect to the frequencies of access, without needing *a-priori* information on these frequencies and without using counters, etc., for recording them. A balanced weighted multidimensional tree structure can thus be used as a self-organizing file structure in databases (see Kriegel and Vaishnavi (1981c)) or in information retrieval.

3.3.4 *Computational Geometry*

Computational geometry deals with algorithmic aspects of geometrical problems. The computational problems arising in two- and three-dimensional computer graphics and in VLSI design have generated much interest in this field.

Interval trees (Edelsbrunner (1980a), McCreight (1980)) and segment trees (Bentley (1977)) are some of the new data structures that have been discovered for developing efficient algorithms for problems related to computational geometry. They are, however, static data structures; their underlying data structure is a completely balanced binary tree. They can, however, be made dynamic by using a suitable balanced weighted tree structure as the underlying data structure and by interpreting the weights appropriately. D-trees (Mehlhorn (1979)) (further discussed in Section 4), a balanced weighted tree structure that has a rather strong bound on its update time, has been successfully used for the dynamization of interval and segment trees. However, D-trees are not convenient to use because they require complex restructuring operations in their update algorithms.

4. Balanced Multidimensional and Weighted Trees: A Close Look at the Literature

Besides balanced multidimensional and weighted tree structures, we include in this review *semi-balanced* multidimensional and weighted tree structures— tree structures that can be accessed in logarithmic time in the worst-case but can be updated in logarithmic time only in the amortized case. Two approaches have been used for discovering such data structures:

(a) extending a balanced tree structure, and
(b) generalizing a balancing concept.

The data structures developed, even using the same approach, differ in their basic strengths, performance, and the simplicity of their update algorithms. Let us elaborate on these approaches and the corresponding tree structures that have been developed.

4.1 Extending a Balanced Tree Structure

In this approach, Mehlhorn (1979) extended a balanced one-dimensional tree structure to handle weighted one-dimensional data. The basic idea is that a data item with a large weight is represented by a number of leaf nodes so as to maintain the balancing constraint. The only balanced tree structure that has been successfully extended using this approach is a weight-balanced tree (see Nievergelt and Reingold (1973)), resulting in D-trees (Mehlhorn (1979)). D-trees support access, insertion, deletion, promotion, and demotion in logarithmic time per operation in the worst case. They, however, require non-linear storage or else require the use of complicated restructuring operations. On the positive side, they inherit rather strong bounds on their update cost from weight-balanced trees, which makes them useful as underlying data structures for dynamic interval and segment trees (Mehlhorn (1984b)).

4.2 Generalizing a Balancing Concept

This approach involves generalizing balanced (one-dimensional) trees to the multidimensional or the weighted-item case. Major balanced tree structures such as B-trees (see Bayer and McCreight (1972)), weight-balanced trees (see Nievergelt and Reingold (1973)) and AVL-trees (Adel'son-Velskij and Landis (1962)) have been generalized. However, because of the various ways a balancing concept can be generalized, a variety of tree structures have been proposed and investigated (see Bent (1982), Bent, Sleator and Tarjan (1980, 1985), Feigenbaum and Tarjan (1983), Gueting and Kriegel (1980), Kriegel and Vaishnavi (1981a), Vaishnavi (1984, 1987)). Fig. 3 summarizes these tree structures. The algorithms for kB-trees (Gueting and Kriegel (1980), Gueting and Kriegel (1981), Kriegel and Vaishnavi (1981b) and kB^+-trees (Kriegel and Vaishnavi (1981a)) involves a multiplicity of cases and such complex restructuring operations as "Balancing with an indirect brother" and "Collapsing with an indirect brother" (Gueting and Kriegel (1980)). On the other hand, an important attribute of multidimensional AVL trees (Vaishnavi (1983b, 1984)), is that it inherits a number of important properties of the base data structure, i.e., AVL-trees. Its update algorithms involve almost the same restructuring operations as those used in AVL-trees. It has relatively simple algorithms for insertion and deletion. Like AVL-trees, it has the property that insertion of a data item requires the application of at most one restructuring operation. The

Tree structures	Balancing concepts of B-trees	Balancing concepts of weight-balanced trees	Balancing concepts of AVL-trees	Direct algorithms for insert/delete	Direct algorithms for join/split
kB-trees	X			yes; logarithmic worst-case but complex	
kB^+-trees	X			same as above	
Globally biased 2, *b* trees	X				yes; logarithmic worst-case
Globally biased binary trees	X				same as above
Globally biased *a*, *b* trees	X				same as above
Biased 2-3 trees	X				yes; logarithmic amortized
Biased 2, *b* trees	X				same as above
Biased binary trees	X				same as above
Locally biased *a*, *b* trees	X				same as above
Biased weight-balanced trees		X			same as above
Pseudo-weight-balanced trees		X			same as above
Multidimensional AVL-trees			X	yes, logarithmic worst-case	
Weighted leaf AVL-trees			X	same as above	

FIG. 3. Summary of tree structures that generalize balancing concepts and require logarithmic worst-case access time.

update simplicity of this tree structure is not at any significant expense of its access time (Vaishnavi (1986)). Weighted leaf AVL-trees (Vaishnavi (1987)), too are characterized by rather simple update algorithms that mirror corresponding AVL-tree algorithms.

5. Paradigms for Modeling Balanced Multidimensional and Weighted Trees

In this section, we describe a paradigm that models the available balanced and semi-balanced multidimensional and weighted structures. It is hoped that the paradigm will guide future research in this area.

The paradigm consists of a strategy used for generalizing a balanced tree structure to higher dimensions, along with a set of strategies for supporting "structure violations" in the generalized structure. We will now describe and illustrate the paradigm.

5.1 The Generalization Strategy Paradigm

In the generalized structure, the same balancing concept is used as in the base structure except that "structure violations" that are "supported" (see Vaishnavi (1984)) by certain specified EQSON-subtree(s) are allowed. The last part of the strategy, i.e. the meaning of "support" and the specification of EQSON-subtree(s) that must support a structure violation, is made concrete in such a way that the worst-case height of the generalized structure is logarithmic and logarithmic update algorithms can be designed.

5.2 Supporting Structure Violation Strategy Paradigm

We have identified three distinct strategies that may be used for supporting structure violations and that model (along with the general paradigm) the available structure. These strategies may be called the "local-support strategy" (see, for example, Bent (1982), Bent et al. (1980, 1985), the "global-support strategy" (see Bent et al. (1985), Feigenbaum and Tarjan (1983), Gueting and Kriegel (1981)), and the "neighbor-support strategy" (see, for example, Vaishnavi (1984, 1987). The local support strategy leads to update algorithms that are logarithmic in amortized time per operation. The other two strategies lead to update algorithms that are logarithmic in worst-case time per operation.

In each of the above three strategies, a "structure violation" must be "supported" by certain "next-lower-dimensional subtree(s)". The strategies differ in specifying the subtree(s) which must "support" the structure violation. In the local-support strategy, the structure violation must be "supported" by *each* of the (next-lower-dimensional) *sibling* subtrees of the nodes causing the structure violation. In the *global-support strategy*, the structure violation must be "supported" by *each* of the *adjoining* subtrees of the node(s) causing the structure violation. In the *neighbor-support strategy*, the structure violation must be "supported" by *at least one of the "adjoining"* subtrees of the nodes(s) causing the structure violation.

We shall now briefly describe the structure of kB-trees (see Gueting and Kriegel (1980)), and k-dimensional AVL-trees (Vaishnavi (1984)), in order to illustrate the generalization strategy described above along with the global-support and neighbor-support strategies, respectively, for supporting structure violations.

5.3 Global-Support Strategy Paradigm for kB-Trees

kB-trees of order m, $m \geq 1$, are higher-order generalizations of B-trees of order m. Thus 1B-trees of order m are the same as B-trees of order m. In a B-tree of order m, every node except the root contains s keys, $d \leq s \leq 2d$. In a kB-tree, however, there may be nodes which have fewer than d keys (constituting *structure violations*) provided that the structure violations are "supported" by *each* of the "next-lower-dimensional EQSON-subtree which adjoin" the nodes causing the structure violations. A structure violation caused by a node, N, at height h is *supported* by a subtree if the height of the subtree is h or more.

Fig. 4(a) shows a 2B-tree of order 1, i.e., a 2-dimensional 2-3 tree. The tree stores the data items:

(a,1), (a,3), (a,7), (a,9), (b,2), (b,4), (b,5), (c,1), (c,2), (d,1), and (d,3).

As in a 2-3 tree, a node, N, storing r keys has $r + 1$ subtrees, all of which store keys of the same dimension as those stored in N. In addition, each two-dimensional key in a node has a one-dimensional EQSON-subtree. Thus (in Fig. 4a) the subtrees rooted at B and C are EQSON-subtrees of keys a and b, respectively. Observe that the left and middle subtrees of the node, A, are empty.

Fig. 4(b) shows the same 2-dimensional 2-3 tree as shown in Fig. 4a except that each empty left, middle, or right subtree of a node N of height h is represented by a chain of h-1 unary nodes storing zero keys (shown in the figure as crossed nodes). Observe in Fig. 4b that the subtree rooted at B is an "adjoining subtree of the next-lower dimension" of nodes P, Q, R, and S. Similarly, the subtree rooted at C is an "adjoining subtree of the next-lower dimension" of nodes R, S, D, and T. The structure violation caused by nodes P, Q, R, S, T, U, and V are allowed because each of these structure violations is supported by each of the corresponding adjoining EQSON-subtrees of the next-lower dimension.

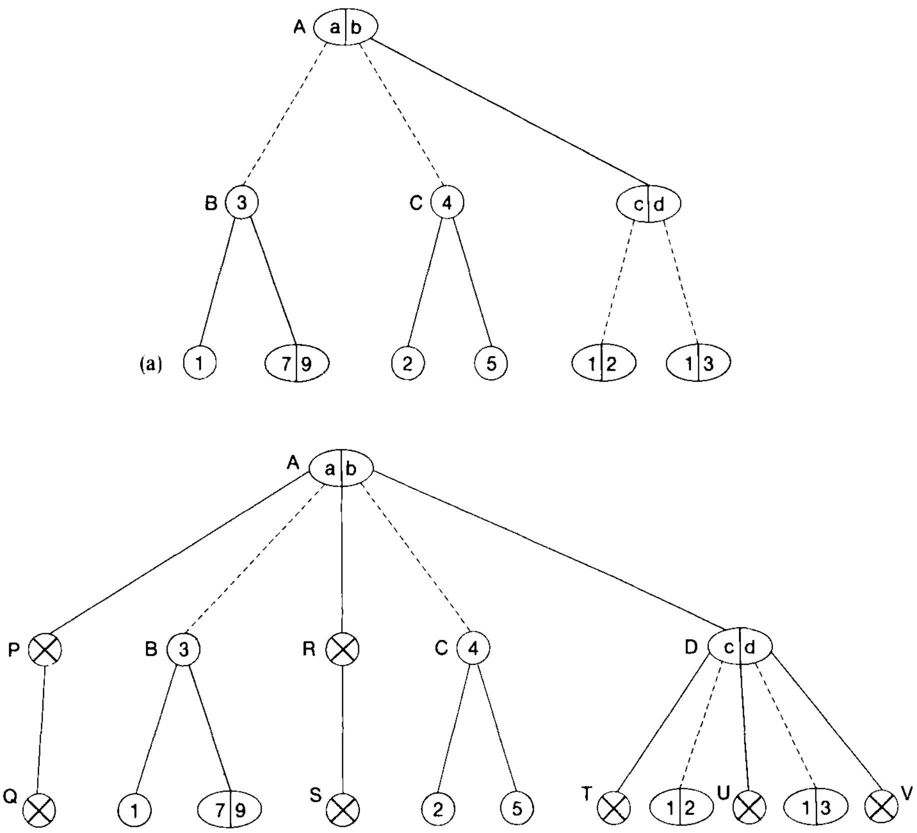
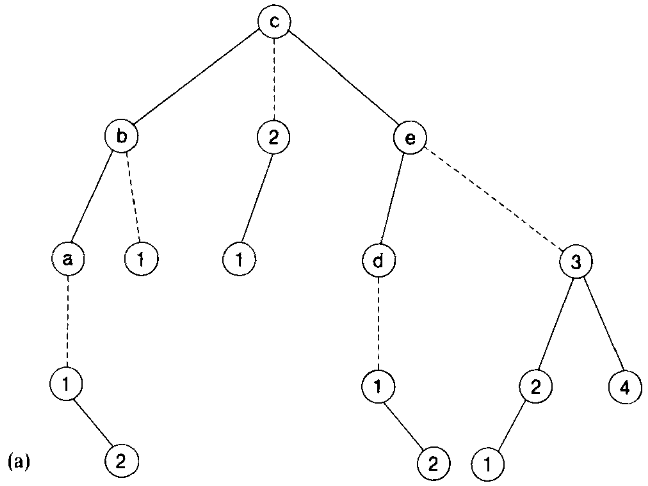


FIG. 4. (a) shows a 2B-tree of order 1, i.e., a 2-dimensional 2–3 tree; (b) shows the same 2-dimensional 2–3 tree as shown in Figure 4(a) except that each empty left, middle, or right subtree of a node N of height h is represented by a chain of h-1 unary nodes storing zero keys (shown in the figure as crossed nodes).

5.4 Neighbor-Support Strategy Paradigm for *k*-dimensional AVL-Trees

k-dimensional AVL-trees (kAVL-trees) are higher-order generalizations of AVL-trees. Figure 5a shows a 2AVL-tree. For the sake of explanation, assume that if a node, M, at height h_1 has a left or right child at height, N, at height h_2, $h_2 < h_i - 1$, then there is a chain of $h_1 - h_2 - 1$ unary nodes, each storing zero keys, between M and N. In particular, assume that each empty left or right subtree of a node at height h is represented by a chain of h-1 unary nodes. Fig. 5b shows the same tree as in Fig. 5a except for the change in its representation.



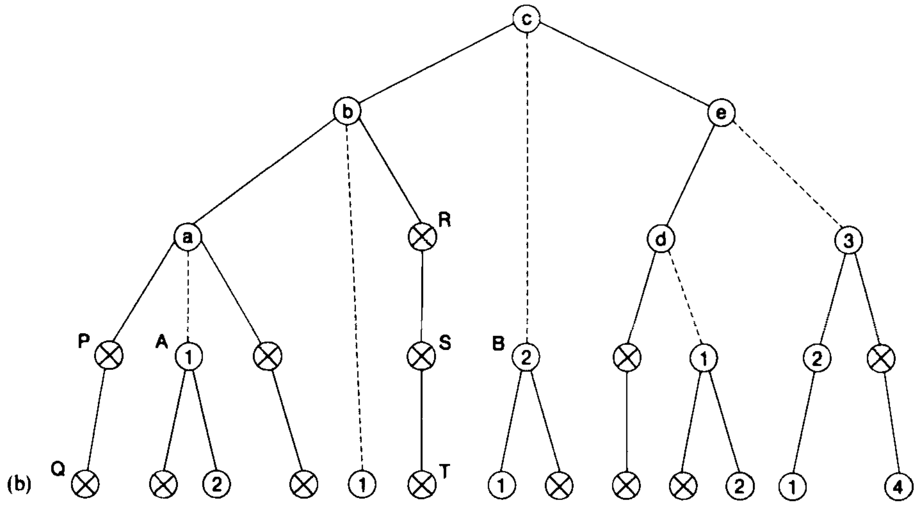


FIG. 5. (a) shows a 2AVL-tree; (b) shows the same tree as in (a) except for the change in its representation.

A chain of *two or more* unary nodes constitute a *structure violation* in a kAVL-tree. A structure violation is allowed in a kAVL-tree provided that it is "supported" by at least one of the "next-lower dimensional EQSON-subtrees that are adjoining" to the nodes in the chain constituting the structure violation.

Let h be the height of the highest node in a chain of unary nodes constituting a structure violation. The structure violation is *supported* by a subtree if the height of the subtree is h-1 or more.

In Fig. 5b, the structure violation caused by the chain of nodes P and Q is supported by the EQSON-subtree rooted at A, a one-dimensional EQSON-subtree adjoining to the nodes P and Q. Similarly, the structure violation caused by the chain of nodes R, S, and T is supported by the EQSON-subtree rooted at B. Other structure violations in the tree are similarly supported by appropriate EQSON-subtrees.

Observe that there cannot be a structure violation in a 1AVL-tree, i.e., an AVL-tree, because it is not possible to support it appropriately since the heights of all next-lower-dimensional EQSON-subtrees are zero.

6. Multiple Attribute Trees (MAT)

6.1 Preliminaries

The basic abstract data structure underlying the MAT is used in different applications in different forms. An early effort is due to Cardenas and Sagamang (1974) who have extended the concept of the doubly chained tree (DCT) to multidimensional data. Kashyap, Subas, and Yao (1977) and Subas and Kashyap (1975) use a similar notion, but their data structure differs from the abstract data structure in the ranking of the attributes according to the decreasing order of their probabilities of occurrence in a query. There are other structures such as the multidimensional B-trees (MDBT) of Ouksel and Scheuermann (1981), the kB trees of Gueting and Kriegal (1980), and the $k\mathrm{B}^+$ tree, and $(k + 1)B^+$ trees of Kriegel and Vaishnavi (1981a) and Kriegel (1984)—all these data structures are based on the same abstract data structure. We present a variation of this abstract data structure, which we subsequently call the *Multiple Attribute Tree* (MAT). The data structure, MAT, as used in this paper, is the same as the above mentioned abstract data structure with the additional property that each filial set is ordered according to the attribute values. This is the same as the MAT of Kashyap, Subas and Yao (1977) and Subas and Kashyap (1975), without the attribute ranking constraint. We retain the name because in both structures the most important property, namely the clustering of tree nodes that have "nearby" attribute values, is preserved.

The MAT will be shown to be as efficient as other variants, but is particularly suited for large data structures that are maintained on secondary storage. The details of these structures are deferred to Section 6.2. We formally define the MAT as follows:

Definition of MAT. A k-dimensional MAT on k-attributes, $A_1, A_2, \ldots, A_k$ for a set (file) of records is defined as a tree of depth k, with the following properties:

(i) It has a root at level O.
(ii) Each child of the root is a (k-1)-dimensional MAT, on (k-1) attributes, $A_2, A_3, \ldots, A_K$, for the subset of the records that have the same A_1 value. This value of A_1 is the value of the root of the corresponding (k-1)-dimensional MAT.
(iii) The child nodes of the root are in the ascending order of their values. This set of child nodes is called the *filial set.* ■

Fig. 6a shows a set of records, and the corresponding MAT is shown in Fig. 6c. We note that there is a root node at level 0, which does not have any value. Every node of level i, i = 1, 2, ..., k corresponds to a value of attribute A_i and this value is the value of the node. Thus, the attributes $A_1, A_2, \ldots, A_k$ form the hierarchy of levels 1 through k. The important property of the MAT is that every filial set is ordered. This is the result of the property that the sorting of a set of records on all the attributes naturally induces a tree structure, as illustrated in Fig. 6b. Hence the abstract MAT can be imagined to have been constructed as follows:

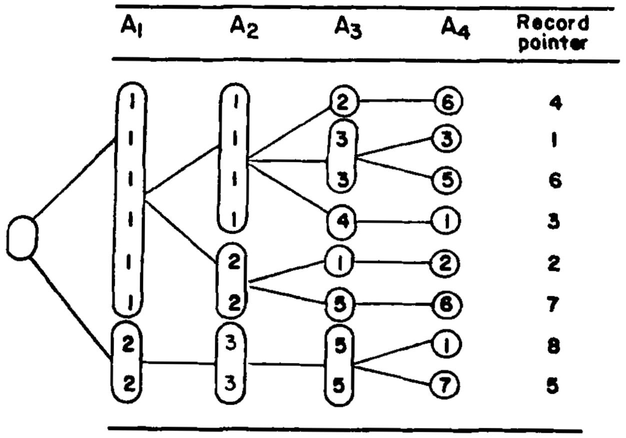
a) Records are sorted in ascending order on all attributes.
b) Starting from the first attribute, all the elements of an attribute having the same attribute value are combined into a node. This process is recursively carried out for each subset of records that have the same first attribute value.

(a)

A_1	A_2	A_3	A_4	Record pointer
1	1	3	3	1
1	2	1	2	2
1	1	4	1	3
1	1	2	6	4
2	3	5	7	5
1	1	3	5	6
1	2	5	6	7
2	3	5	1	8

FIG. 6. (a) Input data file; (b) Sorted input file, and induction of a natural tree structure; (c) MAT representation for data of (a).

(b)



(c)

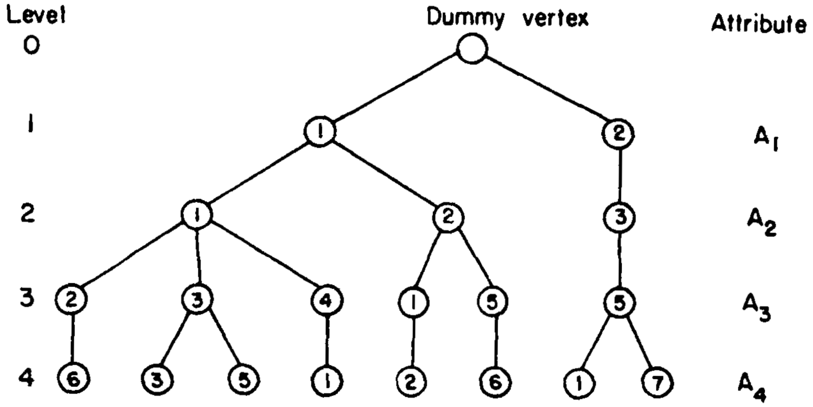


FIG. 6. (*Continued*)

The construction of the MAT can be visualized using geometric notions. The sub-MAT on $A_2, A_3, \ldots, A_k$, generated as in *ii*) of the definition above, corresponds to the set of points that lie on a hyperplane in the data space. The equation of this hyperplane is $A_1 = a_1$ is the value of A_1 corresponding to the root of the sub MAT. Since a hyperplane in a space of dimensionality k corresponds to a space of dimensionality $(k - 1)$, each recursive generation of a sub-MAT is carried out on a space of lesser dimensionality. The same effect will be found in the search algorithms, which recursively search on spaces of lesser dimensionality.

Another important property is that each record or point is represented by a *unique* path from the root to the corresponding terminal node. Because of this property the profile of the MAT, for a set of records, completely depends on the occurrence of various attribute values in the records. In general, the sizes of various filial sets depend on the exact set of input records, and the corresponding attribute values. As a consequence of this behavior the sizes of filial-sets are bound to be random. Hence, we characterize the sizes of the filial-sets of the MAT by average measures. This type of analysis is done for the MAT by Rao and Iyengar (1986) and by Ouksel and Scheuermann (1981) for the multidimensional B-trees. These measures may correspond to the expected values of filial-set sizes, if the probability distribution of the various attributes is known. The disparity of the actual filial set size from this average value can be taken into account by using dispersion measures such as variance, third moment, etc.

We use the following notation in our subsequent discussion:

N: number of records

M: number of nodes of the MAT

k: number of attributes

s_j: average (expected) size of the filial-set at level j, $\quad j = 1, 2, \ldots, k$

σ_j: standard deviation (square root of variance) of the filial set at level $j = 1, 2, \ldots, k$

d_j: size of the set from which values of attribute A_j can be chosen, $\quad j = 1, 2, \ldots, k$

The properties of the MAT can be summarized as follows:

$$s_0 = 1$$

$$M \leq (1 + kN)$$

$$N = s_1 s_2 \cdots s_k$$

$$N = \prod_{i=1}^{k} s_i \tag{6.1}$$

$$M = \sum_{j=1}^{k} \prod_{i=1}^{j} s_i \tag{6.2}$$

When the distribution of data is symmetric over attributes, we characterize the MAT as the *symmetric* version, where the average filial set sizes of all levels are the same, i.e., $s_1 = s_2 = \cdots = s_k = s$, $N = s^k$. This characterization is most appropriate when the distribution of each attribute values is independent of the distribution of other values, and uniform with the same mean. For example, let each A_i correspond to an independent and uniform distribution with probability of occurrence of each value to be p. Then for a symmetric MAT we have $s = pd$, and $d_1 = d_2 = \cdots = d_k = d$. For a symmetric MAT we have

$$s_j = O(N^{1/k}) \tag{6.3}$$

$$\prod_{j=1}^{k-1} s_j = N/s_k = O(N^{1-1/k}) \tag{6.4}$$

In the following sections, we use the symmetric MAT for our analysis. Without loss of generality, the expressions (6.3) and (6.4) are used to estimate the complexity of various algorithms. A MAT for any data can be imagined to be equivalent to a symmetric version with the order estimates satisfying (6.3) and (6.4). For many practical situations, this approximation gives reasonably good analytical measures, as reported in Veni Madhavan (1984), and Rao and Iyengar (1986). However, it is to be noted that there could be certain applications in which this approximation is not completely accurate. But this is the approach that is used mostly in the literature as in Kashyap et al. (1977) and Gopalakrishna and Veni Madhavan (1980). A completely generalized analysis seems very difficult at this stage.

6.2 MAT-Related Multidimensional Data Structures

There are many variants of the MAT data structure. Most of these data structures differ from one another in the organization of either pointers or filial sets.

The data structure proposed by Cardinas and Sagamang (1974) is a refinement of the doubly chained tree (DCT) of Sussenguth (1963). In the modified DCT each level corresponds to an attribute, and Fig. 7 shows the basic idea of the doubly chained tree for the data of Fig. 6a. In this data structure, more pointers are used to facilitate the search and other operations. One disadvantage of the DCT is that the length of the path corresponding to a single record can at most be $N + k$, as against k for the other data structures such as the MAT, etc. In the worst case, the search path for a complete match query can contain as many as $N + k$ nodes. Kashyap et al (1977) have shown the superiority of the MAT over the modified DCT.

The other structures such as the MDBT, kB-tree, etc. are analogous to the MAT in the basic organization of data. But they differ in the organization of filial sets. The organization of filial-sets plays an important role in the performance of the data structure. The binary search tree complex (BST-complex) of Lien *et al.* (1975) is one of the early structures of this kind. The