The Ubiquitous B-Tree

DOUGLAS COMER

Computer Science Department, Purdue University, West Lafayette, Indiana 47907

B-trees have become, de facto, a standard for file organization. File indexes of users, dedicated database systems, and general-purpose access methods have all been proposed and implemented using B-trees This paper reviews B-trees and shows why they have been so successful It discusses the major variations of the B-tree, especially the B$^+$-tree, contrasting the relative merits and costs of each implementation. It illustrates a general purpose access method which uses a B-tree.

Keywords and Phrases: B-tree, B*-tree, B$^+$-tree, file organization, index

CR Categories: 3.73 3.74 4.33 4 34

INTRODUCTION

The secondary storage facilities available on large computer systems allow users to store, update, and recall data from large collections of information called files. A computer must retrieve an item and place it in main memory before it can be processed. In order to make good use of the computer resources, one must organize files intelligently, making the retrieval process efficient.

The choice of a good file organization depends on the kinds of retrieval to be performed. There are two broad classes of retrieval commands which can be illustrated by the following examples:

Sequential: "From our employee file, prepare a list of all employees' names and addresses," and

Random: "From our employee file, extract the information about employee J. Smith".

We can imagine a filing cabinet with three drawers of folders, one folder for each employee. The drawers might be labeled "A–G," "H–R," and "S–Z," while the folders might be labeled with the employees' last names. A sequential request requires the searcher to examine the entire file, one folder at a time. On the other hand, a random request implies that the searcher, guided by the labels on the drawers and folders, need only extract one folder.

Associated with a large, randomly accessed file in a computer system is an *index* which, like the labels on the drawers and folders of the file cabinet, speeds retrieval by directing the searcher to the small part of the file containing the desired item. Figure 1 depicts a file and its index. An index may be physically integrated with the file, like the labels on employee folders, or physically separate, like the labels on the drawers. Usually the index itself is a file. If the index file is large, another index may be built on top of it to speed retrieval further, and so on. The resulting hierarchy is similar to the employee file, where the topmost index consists of labels on drawers, and the next level of index consists of labels on folders.

Natural hierarchies, like the one formed by considering last names as index entries, do not always produce the best perform-

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1979 ACM 0010-4892/79/0600–0121 $00 75

CONTENTS

ance when used in a computer system. Usually, a unique *key* is assigned to each item in the file, and all retrieval is requested by specifying the key. For example, each employee might be assigned a unique employee number which would identify that employee's record. Instead of labeling the drawers of the cabinet "A–G," etc., one would use ranges of employee numbers like "0001"–"3000".

Many techniques for organizing a file and its index have been proposed; Knuth [KNUT73] provides a survey of the basics. While no single scheme can be optimum for all applications, the technique of organizing a file and its index called the B-tree has become widely used. The B-tree is, de facto, the standard organization for indexes in a database system. This paper, intended for computer professionals who have heard of B-trees and want some explanation or direction for further reading, compares several variations of the B-tree, especially the B$^+$-tree, showing why it has become popular. It surveys the literature on B-trees including recent papers not mentioned in textbooks. In addition, it discusses a general purpose file access method based on the B-tree.

The starting point of our discussion is an internal storage structure called the binary search tree. In particular, we begin with balanced binary search trees because of their guaranteed low retrieval cost. For a survey of binary search trees and other internal storage mechanisms, the reader is referred to SEVE74 and NIEV74. NIEV74 also explains the graph theoretic terms "tree," "node," "edge," "root," "path," and "leaf," which will be used throughout the discussion.

The remainder of this Introduction presents a model of the retrieval process and outlines the file operations to be considered. Section 1 presents the basic B-tree as proposed by Bayer and McCreight, giving the methods for inserting, deleting, and locating items. Then for each type of operation, Section 2 examines the cost and concludes that sequential processing can be expensive. In many cases, changes in implementation can lower the costs; Section 3 shows variations of the B-tree which have been developed to do so. Extending the variations of B-trees, Section 4 reviews the problems of maintaining a B-tree in a multiple user environment and outlines solutions for concurrency and security problems. Finally, Section 5 presents IBM's general purpose file access method which is based on the B-tree.

Operations on a File

For purposes of this paper, we think of a *file* as a set of *n records*, each of the form $r_i = (k_i, \alpha_i)$, in which k_i is called the *key* for the *i*th record, and α_i the *associated information*. For example, the key for a record in an employee file might be a five-digit employee number, while the associated information might consist of the employee's name, address, salary, and number of dependents.

We assume that key k_i uniquely identifies record r_i. Furthermore, we assume that although the key is much shorter than the

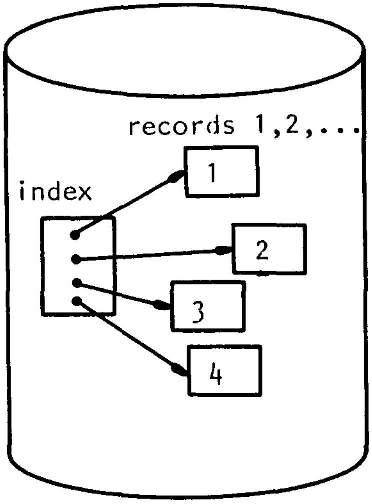

FIGURE 1. A file and its index on a secondary store.

associated information, the set of all keys is too large to fit into main memory. These assumptions imply that if records are to be retrieved randomly using the keys, it would be advantageous to construct an index to speed retrieval. Since the set of all keys does not fit in main memory, the index itself must be external. Finally, we assume that the keys have a natural order, say alphabetical, so we can refer to the *key-sequence order* of a file.

Users conduct *transactions* against a file, inserting, deleting, retrieving, and updating records. In additions, users frequently process the file sequentially, in key-sequence order, starting at a given point. Most often, that starting point is the beginning of the file. A set of *basic operations* which support such transactions are:

insert: add a new record, (k_i, α_i), checking that k_i is unique,

delete: remove record (k_i, α_i) given k_i,

find: retrieve α_i given k_i,

next: retrieve α_{i+1} given that α_i was just retrieved (i.e., process the file sequentially).

For a given file organization, there are costs associated with maintaining the index and with performing each of these operations. Since the index is intended to speed retrieval, processing time is usually taken as the primary cost measure. With current hardware technology, the time required to access secondary storage is the main component of the total time required to process the data. Furthermore, most random access devices transfer a fixed amount of data per read operation, so that the total time required is linearly related to the number of reads. Therefore, the number of secondary storage accesses serves as a reasonable cost measure for evaluating index methods. Other less important costs include the time to process data once it has been placed in main memory, the secondary storage space utilization, and the ratio of the space required by the index to the space required by the associated information.

1. THE BASIC B-TREE

The B-tree has a short but important history. In the late 1960s computer manufacturers and independent research groups competitively developed general purpose file systems and so-called "access methods" for their machines. At Sperry Univac Corporation (in conjunction with Case Western Reserve University) H. Chiat, M. Schwartz, and others developed and implemented a system which carried out insert and find operations in a manner related to the B-tree method which we will describe shortly. Independently, B. Cole, S. Radcliffe, M. Kaufman, and others developed a similar system at Control Data Corporation (in conjunction with Stanford University). R. Bayer and E. McCreight, then at Boeing Scientific Research Labs, proposed an external index mechanism with relatively low cost for most of the operations defined in the previous section; they called it a B-tree[1] [BAYE72].

This section presents the basic B-tree data structure and maintenance algorithms as a generalization of the binary search tree in which more than two paths leave a given node; the next section discusses costs for each operation. Other general introductions may be found in HORO76, KNUT73, and WIRT76.

[1] The origin of "B-tree" has never been explained by the authors. As we shall see, "balanced," "broad," or "bushy" might apply. Others suggest that the "B" stands for Boeing. Because of his contributions, however, it seems appropriate to think of B-trees as "Bayer"-trees.

Recall that in a binary search tree the branch taken at a node depends on the outcome of a comparison of the *query key* and the key stored at the node. If the query is less than the stored key, the left branch is taken; if it is greater, the right branch is followed. Figure 2 shows part of such a tree used to store employee numbers, and the path taken for the query "15."

Now consider Figure 3 which shows a modified search tree with two keys stored in each node. Searching proceeds by choosing one of three paths at each node. In the figure, the query, 15, is less than 42 so the leftmost would be taken at the root. For those queries between 42 and 81 the center path would be selected, while the rightmost path would be followed for queries greater than 81. The decision procedure is repeated at each node until an exact match occurs (success) or a leaf is encountered (failure).

In general, each node in a *B-tree of order d* contains at most 2*d* keys and 2*d* + 1 pointers, as shown in Figure 4. Actually, the number of keys may vary from node to node, but each must have at least *d* keys and *d* + 1 pointers. As a result, each node is at least ½ full. In the usual implementation a node forms one record of the index file, has a fixed length capable of accommodating 2*d* keys and 2*d* pointers, and contains additional information telling how many keys correctly reside in the node.

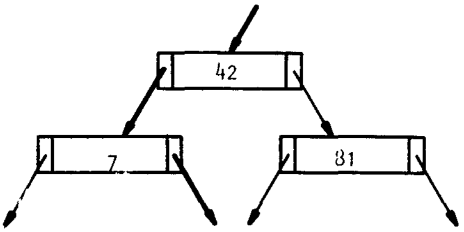

FIGURE 2. Part of a binary search tree for employee numbers The path taken for query "15" is darkened.

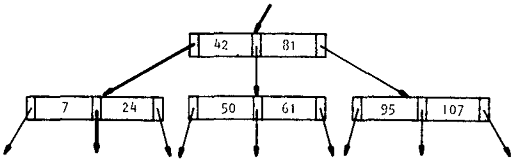

FIGURE 3. A search tree with 2 keys and 3 branches per node. The path taken for query "15" is darkened.

Usually, large, multikey nodes cannot be kept in main memory and require an access to secondary storage each time they are to be inspected. Later, we will see how, under our cost criterion, maintaining more than one key per node lowers the cost of find, insert, and delete operations.

Balancing

The beauty of B-trees lies in the methods for inserting and deleting records that always leave the tree balanced. As in the case of binary search trees, random insertions of records into a file can leave a tree unbalanced. While an unbalanced tree, like the one shown in Figure 5a has some long paths and some short ones, a balanced tree, like the one shown in Figure 5b, has all leaves at the same depth. Intuitively, B-trees have a shape as shown in Figure 6. The longest path in a B-tree of *n* keys contains at most about $\log_d n$ nodes, *d* being the order of the B-tree. A *find* operation may visit *n* nodes

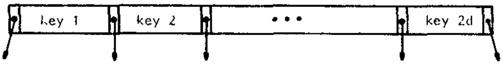

FIGURE 4. A node in a B-tree of order *d* with 2*d* keys and 2*d* + 1 pointers.

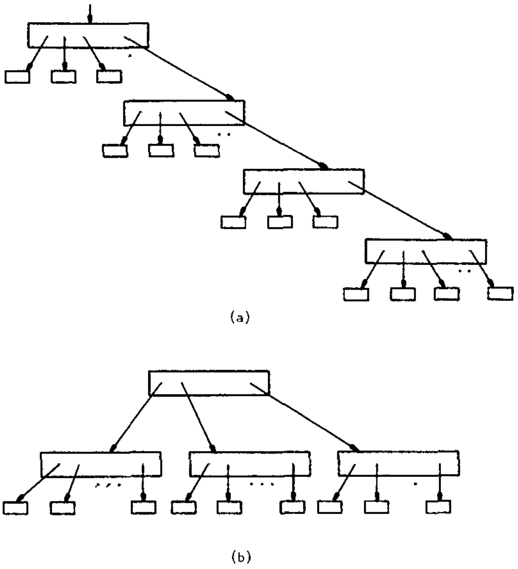

FIGURE 5. (a) An unbalanced tree with many long paths, and (b) a balanced tree with all paths to leaves exactly the same length.

in an unbalanced tree indexing a file of n records, but it never visits more than $1 + \log_d n$ nodes in a B-tree of order d for such a file. Because each visit requires a secondary storage access, balancing the tree has large potential savings. Many schemes to balance trees have been proposed (see NIEV74, FOST65, KARL76 for examples). Each scheme requires some computation time to perform the balancing, so the savings during retrieval operations must be greater than the cost of balancing itself. The B-tree balancing scheme restricts changes in the tree to a single path from a leaf to the root, so it cannot introduce "runaway" overhead. Furthermore, the balancing mechanism uses extra storage to lower the balancing costs (presumably, secondary storage is inexpensive compared to retrieval time). Hence, B-trees gain the advantages of balanced tree schemes while avoiding some of the time-consuming maintenance.

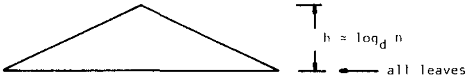

FIGURE 6. The shape of a B-tree of order d indexing a file of n records.

Insertion

To see how balance is maintained during insertion, consider Figure 7a which shows a B-tree of order 2. Since each node in a B-tree of order d contains between d and $2d$ keys, each node in the example has between 2 and 4 keys. Some indicator which is not depicted must be present in each node to mark the current number of keys. Insertion of a new key requires a two-step process. First, a find proceeds from the root to locate the proper leaf for insertion. Then the insertion is performed, and balance is restored by a procedure which moves from the leaf back toward the root. Referring to Figure 7a, one can see that when inserting

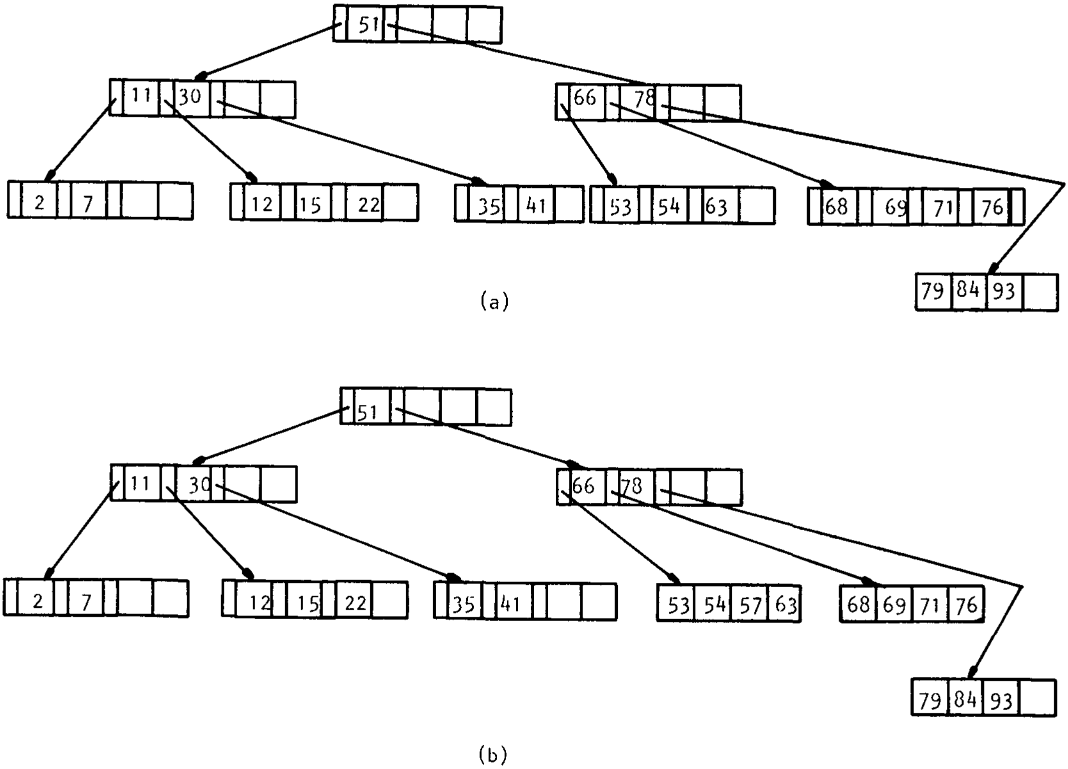

FIGURE 7. (a) A B-tree of order 2, and (b) the same tree after insertion of key "57". Note that the number of keys in the root node may be less than d, the order of the B-tree All other nodes have at least d keys in them.

the key "57" the find terminates unsuccessfully at the fourth leaf. Since the leaf can accommodate another key, the new key is simply inserted, yielding the tree shown in Figure 7b. If the key "72" were inserted, however, complications would arise because the appropriate leaf is already full. Whenever a key needs to be inserted in a node that is already full, a *split* occurs: the node is divided as shown in Figure 8. Of the $2d + 1$ keys, the smallest d are placed in one node, the largest d are placed in another node, and the remaining value is promoted to the parent node where it serves as a separator. Usually the parent node will accommodate an additional key and the insertion process terminates. If the parent node happens to be full too, then the same splitting process is applied again. In the worst case, splitting propagates all the way to the root and the tree increases in height by one level. In fact, a B-tree only increases in height because of a split at the root.

Deletion

Deletion in a B-tree also requires a find operation to locate the proper node. There are then two possibilities: the key to be deleted resides in a leaf, or the key resides in a nonleaf node. A nonleaf deletion requires that an adjacent key be found and swapped into the vacated position so that it finds work correctly. To locate an adjacent key in key-sequence order, one merely searches for the leftmost leaf in the right subtree of the now empty slot. As in a binary search tree, the needed value always resides in a leaf. Figure 9 demonstrates these relationships.

Once the empty slot has been "moved" to a leaf, we must check to see that at least d keys remain. If less than d keys occupy the leaf, then an *underflow* is said to occur, and redistribution of the keys becomes necessary. To restore balance (and the B-tree property that each node has at least d keys) only one key is needed—it could be obtained by borrowing from a neighboring leaf. But since the operation requires at least two accesses to secondary storage, a better redistribution would evenly divide the remaining keys between the two neighboring nodes, lowering the cost of successive deletions from the same node. Redistribution is illustrated by Figure 10.

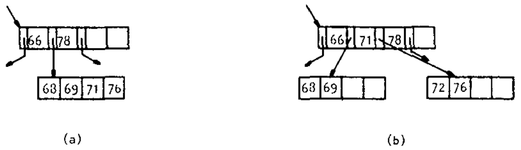

FIGURE 8. (a) a leaf and its ancestor in a B-tree, and (b) the same subtree after insertion of key "72". Each node retains between 2 and 4 keys (d and $2d$).

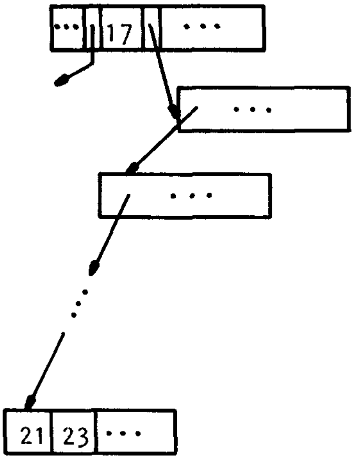

FIGURE 9. Deletion of key "17" requires that the next sequential key, "21" be found and swapped into the vacant position. The next sequential key always resides in the leftmost leaf of the subtree given by the right pointer of the empty position.

Of course, the distribution of keys among two neighbors will suffice only if there are at least $2d$ keys to distribute. When less than $2d$ values remain, a *concatenation* must occur. During a concatenation, the keys are simply combined into one of the nodes, and the other is discarded (note that concatenation is the inverse of splitting). Since only one node remains, the key separating the two nodes in the ancestor is no longer necessary; it too is added to the single remaining leaf. Figure 11 shows an example of concatenation and the final location of the separator key.

When some node loses a separator key due to concatenation of two of its children, it too may underflow and require redistribution from one of its neighbors. The process of concatenating may force concate-

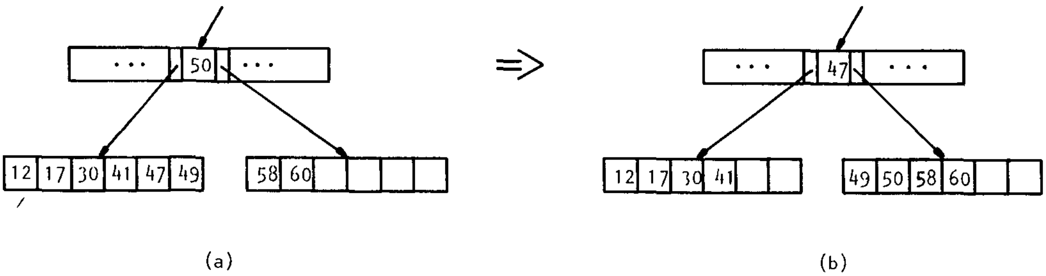

FIGURE 10. (a) Part of a B-tree before, and (b) after redistribution of keys among two neighbors. Note the final position of the separator key, "50". Redistribution into equal size nodes helps avoid underflow on successive deletions.

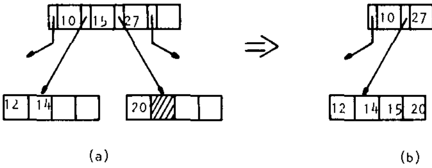

FIGURE 11. (a) A deletion causing concatenation, and (b) the rebalanced tree.

nating at the next higher level, and so on, to the root level. Finally, if the descendants of the root are concatenated, they form a new root, decreasing the height of the B-tree by 1.

Algorithms for insertion and deletion may be found in BAYE72. Simple examples programmed in PASCAL are provided by Wirth [WIRT76].

2. THE COST OF OPERATIONS

Since visiting a node in a B-tree requires an access to secondary storage, the number of nodes visited during an operation provides a measure of its cost. Bayer and McCreight [BAYE72] give a precise analysis of the costs of insertion, deletion, and retrieval. They also provide comprehensive experimental results which relate the theoretical bounds to actual devices. Knuth [KNUT73] also derives bounds for the cost of operations in a B-tree using a slightly different definition. The next section gives a simple explanation of the asymptotic bound on costs.

Retrieval Costs

First, consider the cost of a find operation. Except for the root, each node in the B-tree has at least d direct descendants since there are between d and $2d$ keys per node; the root has at least 2 descendants. So the number of nodes at depths[2] 0, 1, 2, . . ., must be at least 2, $2d$, $2d^2$, $2d^3$, . . . All leaves lie at the same depth h so there are

$$\sum_{i=0_2}^{h} d^i = \frac{d^h - 1}{2d - 1}$$

nodes with at least d keys each. The height of a tree with n total keys is therefore constrained so that

$$2d(d^h - 1)/(d - 1) \leq n$$

with a little work one can show that

$$2d^h \leq n + 1,$$

or

$$h \leq \log_d \frac{n + 1}{2}$$

Thus, the cost of processing a find operation grows as the logarithm of the file size.

Table I shows how reasonable logarithmic cost can be, even for large files. A B-tree of order 50 which indexes a file of one million records can be searched with only 4 disk accesses in the worst case. Later we will see that this estimate is too high; simple implementation techniques lower the worst case cost to 3, and the average cost to less.

Aho et al. [AHO74] provide another perspective on the cost of finds in a B-tree. They show that for the decision-tree model of computation, one where searching is based on comparison at each node, no asymptotically faster retrieval algorithm can be devised. Of course, this model does

[2] The root of a tree lies at depth 0, sons of a node at depth $i-1$ lie at depth i.

TABLE I. UPPER BOUND ON THE NUMBER OF NODES RETRIEVED IN THE WORST CASE FOR VARIOUS NODE SIZES AND FILE SIZES.

Node size ↓	File size (records) →	10^3	10^4	10^5	10^6	10^7
	10	3	4	5	6	7
	50	2	3	3	4	4
	100	2	2	3	3	4
	150	2	2	3	3	4

rule out some methods, such as hashing [MAUE75]. Nevertheless, B-trees exhibit low retrieval costs in both a practical and theoretical sense.

Insertion and Deletion Costs

An insert or delete operation may require additional secondary storage accesses beyond the cost of a find operation as it progresses back up the tree. Overall, the costs are at most doubled, so the height of the tree still dominates the expressions for these costs. Therefore, in a B-tree of order d for a file of n records, insertion and deletion take time proportional to $\log_d n$ in the worst case.

The advantage of nodes containing a large number of keys should now be clear. As the branch factor, d, increases, the logarithmic base increases, and the costs of find, insert, and delete operations decrease. There are, however, practical limits on the size of a node: most hardware systems bound the amount of data that can be transferred with one access to secondary storage. Besides, our cost hides the constant factor which grows as the size of data transferred increases. Finally, each device has some fixed track size which must be accommodated to avoid wasting large amounts of space. So, in practice, optimum node size depends critically on the characteristics of the system and the devices on which the file is allocated.

Bayer and McCreight [BAYE72] give some loose guidelines for choosing node sizes based on rotational delay time, transfer rate, and key size. Their experiments verify that the model's optimal values perform well in practice.

Sequential Processing

So far we have considered random transactions conducted by specifying a key. Often, users wish to view the file as a sequential one, using the *next* operation to process all records in key-sequence order. In fact, one alternative to B-trees, the so called Indexed Sequential Access Method (ISAM) [GHOS69], assumes that sequential accesses occur very frequently.

Unfortunately, a B-tree may not do well in a sequential processing environment. While a simple preorder tree walk [KNUT68] extracts all the keys in order, it requires space for at least $h = \log_d(n + 1)$ nodes in main memory since it stacks the nodes along a path from the root to avoid reading them twice. Additionally, processing a next operation may require tracing a path through several nodes before reaching the desired key. For example, the smallest key is located in the leftmost leaf; finding it requires accessing all nodes along a path from the root to that leaf as shown in Figure 12.

What can be done to improve the cost of the next operation? This question and others will be answered in the next section, under the topic "B⁺-trees."

3. B-TREES VARIANTS

As with most file organizations, variations of B-trees abound. Bayer and McCreight [BAYE72] suggest several implementation alternatives in their original paper. For ex-

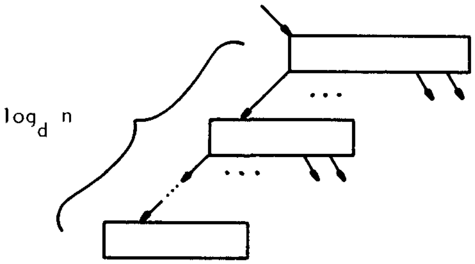

FIGURE 12. The location of the smallest key in the leftmost leaf of a B-tree. Reaching it requires $\log_d n$ accesses.

ample, the underflow condition, resulting from a deletion, is handled without concatenation by redistributing keys from neighboring nodes (unless the requisite number of keys cannot be obtained). Applying the same strategy to the overflow condition can delay splitting and eliminate the associated overhead. Thus, instead of splitting a node as soon as it fills up, keys could merely be distributed into a neighboring node, splitting only when two neighbors fill.

Other variations of B-trees have concentrated on improvements in the secondary costs. Clampet [CLAM64] considers the cost of processing a node once it has been retrieved from secondary storage. He suggests using a binary search instead of a linear lookup to locate the proper descendent pointer. Knuth [KNUT73] points out that a binary search might be useful if the node is large, while a sequential search might be best for small nodes. There is no reason to limit internal searching to sequential or binary search; any number of techniques from KNUT73 might be used. In particular, Maruyama and Smith [MARU77] mention an extrapolation technique they call the square root search.

In their general treatment of index creation for a file, Ghosh and Senko [GHOS69] consider the use of an interpolation search to eliminate a secondary storage access. The analysis presented generalizes to B-trees and indicates that it might be cost effective to eliminate some of the index levels just above the leaves. Since a search would terminate with several possible candidate leaves, the correct one would be found by an "estimate" based on the key value and the key distribution within the file. When the estimate produced the wrong leaf, a sequential search could be carried out. Although some estimates might miss, the method would pay off on the average.

Knuth [KNUT73] suggests a B-tree variation which has varying "order" at each depth. Part of the motivation comes from his observation that pointers in leaf nodes waste space and should be eliminated. It also makes sense to have a different shape for the root (which is seldom very full compared to the other nodes). Maintenance costs for this implementation seem rather high compared to the benefits, especially since secondary storage is both inexpensive and well suited to fixed length nodes.

B*-Trees

Perhaps the most misused term in B-tree literature is B*-tree.[3] Actually, Knuth [KNUT73] defines a B*-tree to be a B-tree in which each node is at least 2/3 full (instead of just 1/2 full). B*-tree insertion employs a local redistribution scheme to delay splitting until 2 sibling nodes are full. Then the 2 nodes are divided into 3, each 2/3 full. This scheme guarantees that storage utilization is at least 66%, while requiring only moderate adjustment of the maintenance algorithms. It should be pointed out that increasing storage utilization has the side effect of speeding up the search since the height of the resulting tree is smaller.

The term B*-tree has frequently been applied to another, very popular variation of B-trees also suggested by Knuth (cf. [KNUT73, WEDE74, BAYE77]). To avoid confusion, we will use the term B$^+$-tree for Knuth's unnamed implementation.

B$^+$-Trees

In a B$^+$-tree, all keys reside in the leaves. The upper levels, which are organized as a B-tree, consist only of an index, a roadmap to enable rapid location of the index and key parts. Figure 13 shows the logical separation of the index and key parts. Naturally, index nodes and leaf nodes may have different formats or even different sizes. In particular, leaf nodes are usually linked together left-to-right, as shown. The linked list of leaves is referred to as the *sequence set*. Sequence set links allow easy sequential processing.

To fully appreciate a B$^+$-tree, one must understand the implications of having an independent index and sequence set. Consider for a moment the find operation.

[3] An amusing case is the "B* tree search algorithm," which is about a tree-search algorithm named B* [BERL78].

Searching proceeds from the root of a B$^+$-tree through the index to a leaf. Since all keys reside in the leaves, it does not matter what values are encountered as the search progresses as long as the path leads to the correct leaf.

During deletion in a B$^+$-tree, the ability to leave non-key values in the index part as separators simplifies processing. The key to be deleted must always reside in a leaf so its removal is simple. As long as the leaf remains at least half full, the index need not be changed, even if a copy of the key had been propagated up into it. Figure 14 shows how the copy of a deleted key can still direct searches to the correct leaf. Of course, if an underflow condition arises, the redistribution or concatenation procedures may require adjusting values in the index as well as in the leaves.

Insertion and find operations in a B$^+$-tree are processed almost identically to insertion and find operations in a B-tree. When a leaf splits in two, instead of promoting the middle key, the algorithm promotes a copy of the key, retaining the actual key in the right leaf. Find operations differ from those in a B-tree in that searching does not stop if a key in the index equals the query value. Instead, the nearest right pointer is followed, and the search proceeds all the way to a leaf.

We have seen that B-trees, which support low-cost find, insert, and delete operations, may require $\log_d n$ accesses to secondary storage to process a next operation. The B$^+$-tree implementation retains the logarithmic cost properties for operations by key, but gains the advantage of requiring at most 1 access to satisfy a next operation. Moreover, during the sequential processing of a file, no node will be accessed more than once, so space for only 1 node need be available in main memory. Thus, B$^+$-trees are well suited to applications which entail both random and sequential processing.

Prefix B$^+$-Trees

The separation of the index and sequence set in a B$^+$-tree is intuitively appealing. Recall that the index part serves merely as a roadmap to guide the search to the correct leaf; it need not contain actual keys at all. When keys consist of a string of characters there is good reason not to use actual keys

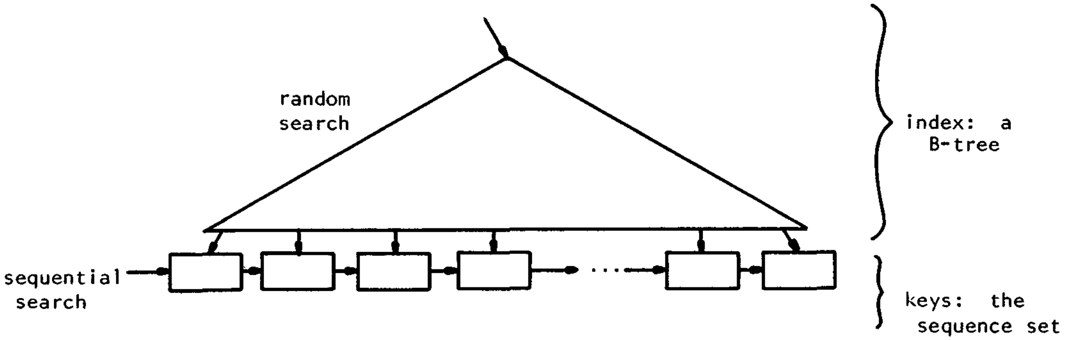

FIGURE 13. A B$^+$-tree with separate index and key parts. Operations "by key" begin at the root as in a B-tree, sequential processing begins at the leftmost leaf.

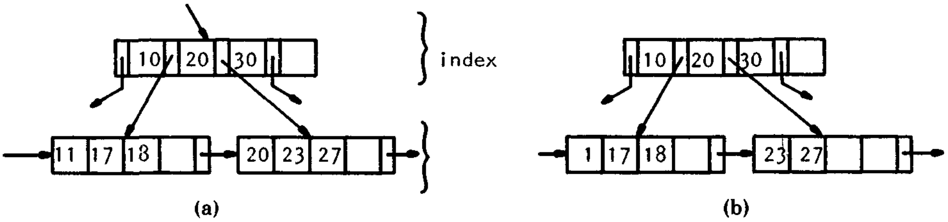

FIGURE 14. (a) A B$^+$-tree and (b) the B$^+$-tree after deletion of the key "20". Even after its removal, key "20" still serves as a separator value in the index part.

as separators: actual keys require too much space. Bayer and Unterauer [BAYE77] consider an alternative, the Prefix B^+-tree.

Suppose that the sequence of alphabetic keys "binary," "compiler," "computer," "electronic," "program," and "system" were allocated in a B-tree as shown in Figure 15. The separator value in the index between the keys "computer," "electronic" need not be either of them; any string between suffices. For example, any of the strings "elec," "e," or "d" would do nicely. Since it makes no difference during retrieval, the shortest such separator should be used to save space. As space requirements become smaller, more keys can be placed in each node, the branching factor increases, and the height of the tree decreases. Since shorter trees cost less to search, using shorter separators will decrease access time as well as save space.

The simple technique of choosing the shortest unique prefix of the key to serve as a separator works well. In the example, the shortest prefix of "electronic" which distinguishes it from "computer" is "e." Sometimes, however, the prefix technique does not perform well: choosing the shortest prefix of "programmers" which distinguishes it from "programmer" results in no savings at all. In such cases, Bayer and Unterauer suggest scanning a small neighborhood of keys to obtain a good pair for the separation algorithm. While this may leave the nodes unevenly loaded, having a few extra keys in one of the nodes will not affect the overall costs.

Virtual B-Trees

Many modern computer systems employ a memory management scheme which provides each user with a large virtual memory. The address space of a user's virtual memory is divided into pages which are saved on secondary storage and loaded into main memory automatically when they are referenced. This technique, called *demand paging*, multiplexes real memory among users and, at the same time, affords protection to insure that one user will not interfere with the data of another. Furthermore, special purpose hardware handles the paging so that transfers to and from secondary storage are performed at high speed.

The availability of demand paging hardware suggests an interesting implementation of B-trees. Through careful allocation, each node of the B-tree can be mapped into one page of the virtual address space. Then the user treats the B-tree as if it were in memory. Accesses to nodes (pages) which are not in memory cause the system to "page-in" the node from secondary storage.

Most paging algorithms choose to remove the least recently used (LRU) page when making room for a new one. In terms of a B-tree, the most active nodes are those close to the root; these tend to stay in memory. In fact, Bayer and McCreight [BAYE72] and Knuth [KNUT73] both suggest a LRU mechanism for B-trees even when not using paging hardware. At least, the root should remain in main memory since it is accessed for each search.

Thus, virtual B-trees have the following advantages:

1) The special hardware performs transfers at high speed,
2) The memory protection mechanism isolates other users, and
3) Frequently accessed parts of the tree will remain in memory.

Compression

Several other implementation techniques have been suggested to improve the performance of B-trees. Wagner [WAGN73] summarizes several of them, including the notions of compressed keys and compressed pointers.[4]

Pointers can be compressed using a base/displacement form of node address rather than an absolute address value. A node with compressed pointers has the form shown in Figure 16, where the base address is stored once in the node, and an offset value, or displacement beyond the base, replaces each pointer. To reconstruct an actual pointer value, the base is added to the displacement for that pointer. Compressed pointer techniques are particularly appropriate for virtual B-trees where pointers take on large address values.

Keys, or separator values, can be com-

[4] See also AUER76.

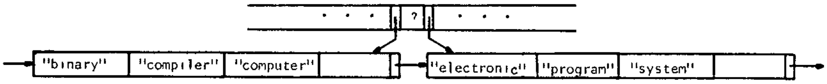

FIGURE 15. Part of a Prefix B⁺-tree. The index entry "e" is sufficient to separate "computer" from "electronic."

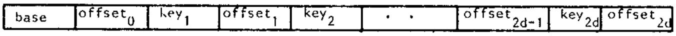

FIGURE 16. A node with compressed pointers. To obtain the ith pointer, the base value is added to the ith offset.

pressed using any one of several standard techniques for removing redundancy [RUBI76]. Both key compression and pointer compression increase the capacity of each node and, therefore, decrease the retrieval costs. The tradeoff for decreased secondary storage accesses is an increase in the CPU time necessary to search a node after it has been read. Thus, complicated compression algorithms may not always be cost effective.

It should be noted that both front and rear compression can be applied to keys. For example Bayer and Unterauer [BAYE77] consider compression of keys for prefix B⁺-trees.

Variable Length Entries

Many applications require the storage of data with variable length keys. Additionally, variable length entries result from compression techniques mentioned above. McCreight [McCR77] considers the storage of trees with variable length entries and shows how promoting shorter keys during an insertion produces a tree with better storage utilization and faster access times.

Binary B-Trees

Another variation proposed by Bayer [BAYE72a], the Binary B-tree, makes B-trees suitable for a one-level store. Essentially, a Binary B-tree is a B-tree of order 1; each node has 1 or 2 keys and 2 or 3 pointers. To avoid wasting space for nodes that are only half full, a linked representation is used as shown in Figure 17. Nodes with 1 key are represented exactly as in Figure 17a, while nodes with 2 keys are linked as in Figure 17b. Since the right pointer in a node may point to either a sibling or a descendant, one extra bit must be used to indicate its meaning.

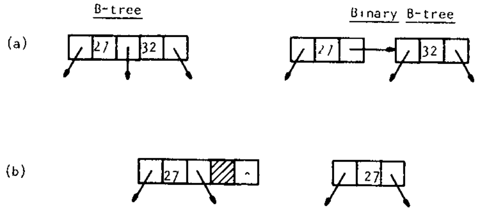

FIGURE 17. Nodes in a B-tree and the corresponding nodes in a Binary B-tree. Each right pointer in the Binary B-tree representation can point to a sibling or a descendent.

Analysis shows that insertion, deletion, and find still take only log n steps as in a B-tree, although searching the rightmost path requires twice as many nodes to be accessed as the leftmost. Using the right pointer for two purposes does complicate the insertion and deletion algorithms, however. To maintain logarithmic cost, care must be taken to insure that there are never two right links pointing to sibling nodes in a row. Detailed algorithms for a rotation process, one that prevents three or more successive sibling links, are given in BAYE72a and WIRT76.

An extension of the Binary B-tree, which allows for both left and right links to point to sibling nodes, exhibits symmetry lacking in the Binary B-tree. Hence, the name Symmetric Binary B-tree has been applied to such a data structure by Bayer [BAYE73], who also reports that Symmetric Binary B-trees contain the well-known class of AVL trees as a subclass [FOST65].

2-3 Trees and Theoretical Results

Hopcroft developed the notion of a 2-3 tree, and explored its usefulness in a one-level store. Each node in a 2-3 tree has 2 or 3 sons (because it contains 1 or 2 keys). Thus, a 2-3 tree is a B-tree of order 1, and vice versa. The small node size makes 2-3 trees impractical for external storage, but quite appropriate for an internal data structure. Rosenbaum and Snyder [ROSE78], and Miller et al. [MILL77] consider the problem of constructing optimal 2-3 trees for a given set of keys. They use the number of comparisons and the number of node accesses, respectively, as the cost criterion. In each case, a linear time algorithm is presented for constructing optimal trees from the sorted list of keys. The results in MILL77 extend to B-trees of arbitrary order.

Yao [YAO78] reports the results of analyzing 2-3 trees built from a uniformly distributed set of *n* keys. The paper gives both an upper and lower bound on the expected storage utilization. Extending the analysis to B-trees of higher order, Yao has shown that the expected storage utilization is *ln* 2 ≈ 69%.

Guibas et al. [GUIB77] consider a B-tree variant for maintaining a list of keys which have highly skewed probability of access. By maintaining a set of *fingers* which point to localities of interest, one can update items within *p* locations from a finger in $\log_d p$ time. For example, one might allocate a finger at the beginning and end of the list. As locality of activity changes, one of the fingers can be moved to the new locality.

Guibas and Sedgewick [GUIB78] present another B-tree scheme and compare the performance of several balanced tree techniques. Their important contribution shows that no upward splitting is ever required. The trick is to split nodes that are nearly full when traveling down the tree. The next section shows that eliminating bottom-up updating can be crucial to performance.

Also see BROW78 and BROW78a for related theoretical results.

4. B-TREES IN A MULTIUSER ENVIRONMENT

If B-trees are to be used in a general purpose database system, they must permit several user requests to be processed simultaneously. Unless some constraints are applied to synchronize the processes, they may interfere with each other. One process may read a node and follow one of the links while another process is changing it. To further complicate the interaction, find operations begin processing top-down in the B-tree while insertion and deletion require bottom-up access. Samadi [SAMA76] presents one solution to the concurrency problem. Held and Stonebraker [HELD78] argue that concurrency conflicts, which are resolved by giving only one process access to the tree, diminish the advantages of B-trees in a multiuser environment.

Bayer and Schkolnick [BAYE77a] show that a set of locking protocols, enforced by a supervisor process, can insure the integrity of B-tree accesses while allowing concurrent activity. In essence, a find locks, or holds, a node once it has been read so that other processes cannot interfere with it. As the search progresses to the next depth, the find processor releases its lock on the ancestor, allowing others to read it. Thus, readers lock at most two nodes at any time; other reader processes are free to explore (and lock) other parts of the tree simultaneously.

Updating in a concurrent environment presents a more complex problem, one that requires more complex protocols. Since updates may affect higher levels in the tree, an update process leaves a reservation on each node it accesses, reserving the right to lock the node. Later, the reservation may be converted to a lock if the update process determines that its change will propagate to the reserved node. Alternatively, the reservation may be cancelled if the update will not affect the reserved node. Reserved nodes may be read, since readers will always continue to a leaf, but they may not be reserved a second time until the first reservation is cancelled.

Once an update process establishes reservations on a path leading to some leaf, it may convert the reservations to absolute locks, top-down. The absolute lock guarantees that no other process will access the node. Then the update proceeds, changing only nodes on which it holds absolute locks. After all changes have been made, absolute locks are cancelled and the updated path becomes available for other processes.

Reserving an entire path from the root to a leaf prevents other updates from accessing the B-tree. Furthermore, most updates affect only a few levels—those near a leaf—

so reserving an entire path is not desirable. Yet reserving too few nodes might make it necessary to begin again at the root. Bayer and Schkolnick, therefore, propose a generalized locking protocol which represents a tradeoff between the two extremes. They provide a parameterized model and show how reservations can permit enough concurrency to utilize present technology while wasting very little time on restarting reservations.

In contrast, using the top-down splitting suggested in GUIB78 eliminates the need for all but the most simple protocols, since updaters never need to travel back up the tree at all. Thus, only one pair of nodes will ever be locked at a given time. Of course, the price for splitting nodes before they fill completely is a slight decrease in storage utilization and a corresponding increase in access time.

Security

The protection of information in a multiuser environment poses another problem for database designers. Earlier, under the topic Virtual B-Tree, it was indicated that isolation of users could be obtained from the memory protection mechanism of paging. When the contents of a file must be protected outside of the system, some encryption technique must be used. Bayer and Metzger [BAYE76] consider encipherment schemes and possible security threats. They show that encipherment has a relatively high cost unless implemented via hardware. On the other hand, changes to the B-tree maintenance algorithms to accommodate encoded files are minor, especially if the encipherment can be done "on the fly" during data transmission.

5. A GENERAL PURPOSE ACCESS METHOD USING B$^+$-TREES

This section presents an example of the use of B$^+$-trees—IBM's general purpose B-tree based access method, VSAM [IBM1, IBM2, KEEH72, WAGN73]. Intended to serve in a wide variety of applications, VSAM is designed to support sequential searching as well as logarithmic cost insertion, deletion, and find operations. Compared to the conventional indexed-sequential organization, the B$^+$-trees offer the following advantages: dynamic allocation and release of storage, guaranteed storage utilization of 50%, and no need for periodic "reorganization" of the entire file.

Since VSAM must handle the storage of both keys and associated information, a VSAM file is represented as in Figure 18. The top two sections of the VSAM tree form a B$^+$-tree index and sequence set as described earlier; the leaves contain actual *data records*. In VSAM terminology, a leaf is called a *control interval*, and forms the basic unit of data transferred in one I/O operation. Each control interval contains one or more data records as well as *control information* describing the format of the interval. Figure 19 illustrates the fields of a control interval.

Performance Enhancements

Although VSAM presents a logical, or machine-independent, view of data to the user, the file organization must accommodate the underlying devices if transactions are to be conducted efficiently. Therefore, the maximum size of a control interval is limited by the largest unit of data that the hardware can transfer in one operation. In addition, the set of all control intervals associated with one sequence set node (called a *control area*) must fit on one cylinder of the particular disk storage unit used to store the file. These restrictions improve performance and permit even further enhancements described below.

Since all the descendants of a sequence set node are allocated on one cylinder, performance can be improved by allocating the sequence set node on the same cylinder. Then, once the sequence set node has been retrieved, items in the control area can be retrieved without disk arm movement. An extension to the contiguous sequence set node allocation is demonstrated in Figure 20 which shows how the sequence set node can be replicated on one track of the cylinder. Replication reduces disk seek time. VSAM attempts to improve performance in several other ways. Pointers are compressed using the base/displacement method described above, keys are compressed in both the forward (prefix) and

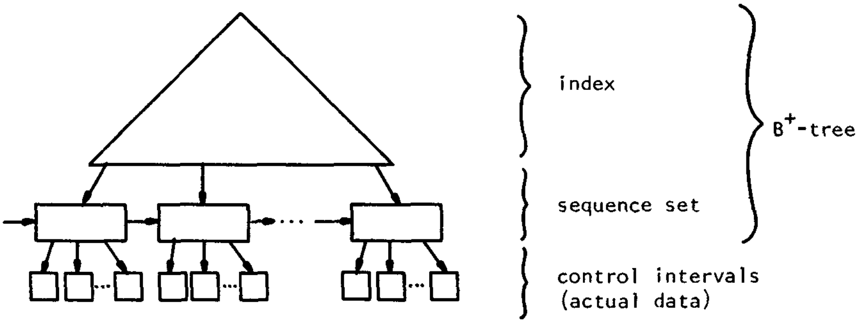

FIGURE 18. A VSAM file with actual data (associated information) stored in the leaves.

Data record 1	Data record 2	• • •	Data record r	control information about data	control information about control interval

FIGURE 19. The format of a control interval. The control fields describe the control interval itself, and the format of the data fields.

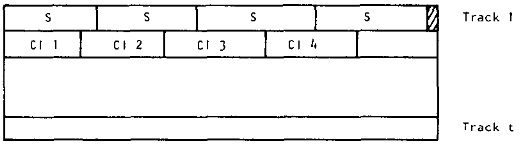

FIGURE 20. The format of a control area with the sequence set node, S, replicated on the first track to minimize latency time.

backward (suffix) directions, index records can be replicated, and the index can be allocated on a separate device to allow concurrent access of index and data. Finally, VSAM allows the index part to be a virtual B-tree, using the virtual memory hardware to retrieve it.

Tree-Structured File Directory

Perhaps the most novel idea in the VSAM implementation is that one data format should be used throughout the system. For example, those routines which maintain a directory of all VSAM files in the system keep the information in a VSAM file, the *master catalog.* Figure 21 shows the master catalog which contains an entry for each VSAM file (or VSAM data set). Since all VSAM files must be entered into the catalog, the system can locate any file automatically given its name. Of course, the catalog is a VSAM data set so it contains an entry describing itself.

If several processes access the master catalog simultaneously, contention occurs, and all but one will have to wait. To avoid lengthy delays caused by such contention, each user can define a local catalog with entries for his VSAM files. The user catalogs, which are VSAM files, must be entered into the master catalog. Once a user catalog has been located by searching the master catalog, further references to files indexed by that catalog do not entail searching the master catalog. The resulting multilevel, tree-structured catalog scheme has a flavor similar to the MULTICS file system [ORGA72].

Other VSAM Facilities

Many facets of VSAM have not surfaced in our brief discussion—the reader is warned that we have only given a quick overview. For example, the VSAM files we discussed are called *key-sequenced.* Another form, the *entry-sequenced* VSAM files allow efficient sequential processing when no key accompanies a record (i.e., no operations are to be performed using the key). Entry-sequenced VSAM files require no index so they are less expensive to maintain.

In addition to the VSAM file maintenance and retrieval procedures, the system provides a mechanism for defining and loading a VSAM file. One must decide how to distribute free space within the file: if the user anticipates many insertions, then

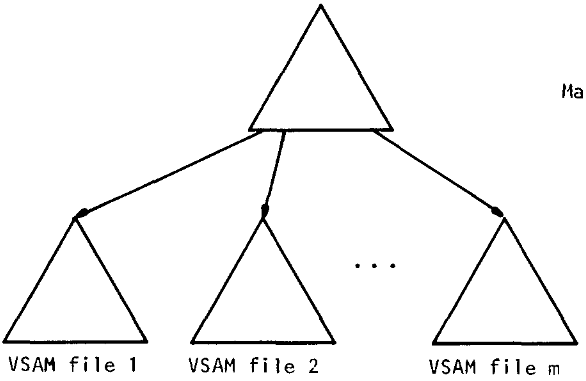
Master catalog: A VSAM file with information for all VSAM files
VSAM file 1
VSAM file 2
. . .
VSAM file m

FIGURE 21. The VSAM Master catalog, which serves as a directory for all VSAM files, is itself a VSAM file.

the file should probably not be loaded with each node 100% full or the initial insertions will be expensive. On the other hand, if the file will remain relatively stable, loading the nodes to only 50% capacity wastes storage. The VSAM file definition facility provides assistance by loading the file according to the parameters chosen.

Finally, VSAM supplies facilities for efficient insertion of a large contiguous set of records, protection of data, file backup, and error recovery, all of which are necessary in a production environment.

SUMMARY

A balanced, multiway, external file organization, the B-tree, is efficient, versatile, simple, and easily maintained. One variation, the B^+-tree, allows efficient sequential processing of the file while retaining the desirable logarithmic cost for find, insert, and delete operations. B-tree schemes guarantee 50% storage utilization while allocating and releasing space as the file grows or shrinks. Moreover, B-trees grow and shrink in exactly the opposite manner; massive file "reorganization" is never necessary even after heavy transaction traffic.

Different B-tree implementation techniques provide enhanced performance, generality, and the ability to use B-trees in a multiuser environment. Compression of keys and pointers, careful allocation (and replication) of nodes on secondary storage, and local redistribution of keys during insertion or deletion all improve performance and make B-trees viable in a production environment, while locking protocols, virtual memory protection, and data encryption provide security and mutual exclusion necessary when a B-tree must be shared by several users.

IBM's VSAM demonstrates that it is reasonable to construct a general purpose file access method based on B-trees. In addition to user's B-tree files, the system itself uses a B-tree file to catalog the name and location of all available VSAM files. Using a B^+-tree implementation to permit efficient sequential processing, VSAM incorporates many of the techniques available for performance enhancement and protection of data.

ACKNOWLEDGMENTS

The author thanks the referees, especially for providing contacts regarding the history of B-trees, and IBM Corporation for cheerfully making available detailed information on its B-tree based access method when none of its competitors would reveal theirs.

REFERENCES

AHO74 AHO, A., HOPCROFT, J., AND ULLMAN, J. *The desıgn and analysıs of computer algorıthms,* Addıson Wesley, Publ. Co., Reading, Mass., 1974.

AUER76 AUER, R. *Schlusselkompressıonen ın B*-baumen,* Diplomarbeit, Tech. Unıversitat, Munich, 1976.

BAYE72 BAYER, R, AND MCCREIGHT, C "Organızation and maintenance of large ordered ındexes," *Acta Inf.* 1, 3 (1972), 173-189.

BAYE72a BAYER, R. "Bınary B-trees for virtual memory," in *Proc 1971 ACM SIGFIDET Workshop,* ACM, New York, 219-235.

BAYE73 BAYER, R. "Symmetric binary B-trees. data structure and maıntenance algorithms," *Acta Inf.* 1, 4 (1972), 290-306

BAYE76 Bayer, R., and Metzger, J. "On encipherment of search trees and random access files," *ACM Trans. Database Syst.* **1,** 1 (March 1976), 37-52.

BAYE77 Bayer, R , and Unterauer, K. "Prefix B-trees," *ACM Trans. Database Syst.* **2,** 1 (March 1977), 11-26.

BAYE77a Bayer, R., and Schkolnick, M. "Concurrency of operations on B-trees," *Acta Inf.* **9,** 1 (1977), 1-21.

BERL78 Berliner, H. *The B*-tree search algorithm: a best-first proof procedure,* Tech. Rep. CMU-CA-78-112, Computer Science Dept , Carnegie-Mellon Univ., Pittsburgh, 1978.

BROW78 Brown, M "A storage scheme for height-balanced trees," *Inf. Process. Lett.* **7,** 5 (Aug. 1978), 231-232.

BROW78a Brown, M. "A partial analysis of height-balanced trees," *SIAM J. Comput.*, to appear.

CLAM64 Clampett, H. "Randomized binary searching with tree structures," *Commun. ACM* **7,** 3 (March 1964), 163-165.

FOST65 Foster, C. "Information storage and retrieval using AVL trees," in *Proc. ACM 20th National Conf.*, ACM, New York, 1965, 192-205.

GHOS69 Ghosh, S , and Senko, M. "File organization: on the selection of random access index points for sequential files," *J ACM* **16,** 4 (Oct. 1969), 569-579.

GUIB77 Guibus, L., McCreight, E., Plass, M., and Roberts, J. "A new representation for linear lists," in *Proc. 9th ACM Symp. Theory of Computing,* ACM, New York, 1977, 49-60.

GUIB78 Guibas, L., and Sedgewick, R. "A dichromatic framework for balanced trees," in *Proc 19th Symp. Foundations of Computer Science*, 1978, 8-21.

HELD78 Held, G., and Stonebraker, M. "B-trees reexamined," *Commun. ACM* **21,** 2 (Feb. 1978), 139-143.

HORO76 Horowitz, E., and Sahni, S. *Fundamentals of data structures*, Computer Science Press, Inc., Woodland Hills, Calif., 1976.

IBM1 *OS/VS Virtual Storage Access Method (VSAM) planning guide*, Order No. GC26-3799, IBM, Armonk, N.Y.

IBM2 *OS/VS Virtual Storage Access Method (VSAM) logic*, Order No. SY26-3841, IBM, Armonk, N.Y.

KARL76 Karlton, P., Fuller, S., Scroggs, R., and Kachler, E. "Performance of height balanced trees," *Commun. ACM* **19,** 1 (Jan. 1976), 23-28.

KEEH74 Keehn, D., and Lacy, J. "VSAM data set design parameters," *IBM Syst. J.* **3,** (1974), 186-212.

KNUT68 Knuth, D. *The art of computer programming, Vol. 1. fundamental algorithms*, Addison-Wesley Publ. Co., Reading, Mass , 1968.

KNUT73 Knuth, D. *The art of computer programming, Vol. 3: sorting and searching,* Addison-Wesley Publ. Co , Reading, Mass., 1973.

MARU77 Maruyama, K , and Smith, S. "Analysis of design alternatives for virtual memory indexes," *Commun. ACM* **20,** 4 (April 1977), 245-254.

MAUE75 Mauer, W., and Lewis, T. "Hash table methods," *Comput. Surv.* **7,** 1 (March 1975), 5-19.

McCR77 McCreight, E. "Pagination of B*-trees with variable-length records," *Commun. ACM* **20,** 9 (Sept. 1977), 670-674

MILL77 Miller, R., Pippenger, N., Rosenberg, A., and Snyder, L. *Optimal 2-3 trees,* IBM Research Rep. RC 6505, IBM Research Lab , Yorktown Heights, N.Y., 1977.

NIEV74 Nievergelt, J. "Binary search trees and file organization," *Comput. Surv.* **6,** 3 (Sept. 1973), 195-207.

ORGA72 Organick, E. *The Multics system: an examination of its structure,* MIT Press, Cambridge, Mass., 1972.

ROSE78 Rosenberg, A , and Snyder, L. "Minimal comparison 2-3 trees," SIAM *J. Comput* **7,** 4 (Nov. 1978), 465-480.

RUBI76 Rubin, F "Experiments in text file compression," *Commun. ACM* **19,** 11 (Nov 1976), 617-623.

SAMA76 Samadi, B. "B-trees in a system with multiple views," *Inf. Process. Lett.* **5,** 4 (Oct. 1976), 107-112

SEVE74 Severence, D. "Identifier search mechnisms. a survey and generalized model," *Comput. Surv* **6,** 3 (Sept. 1974), 175-194.

WAGN73 Wagner, R. "Indexing design considerations," *IBM Syst. J.* **4,** (1973), 351-367.

WEDE74 Wedekind, H. "On the selection of access paths in a database system," in *Data base management* (*Proc. IFIP Working Conf. Data Base Management*) J Klimbie and K. Koffeman (Eds.), Elsevier/North-Holland Publishing Co., New York, 1974, 385-397.

WIRT76 Wirth, N. *Algorithms + data structures = programs,* Prentice-Hall Inc., Englewood Cliffs, N J., 1976.

YAO78 Yao, A. "On random 2-3 trees," *Acta Inf.* **9,** 2 (1978), 159-170.

RECEIVED AUGUST 1978; FINAL REVISION ACCEPTED DECEMBER 1978