A STUDY OF INDEX STRUCTURES FOR MAIN MEMORY DATABASE MANAGEMENT SYSTEMS

by

Tobin J. Lehman and Michael J. Carey

Computer Sciences Technical Report #605

July 1985

A Study of Index Structures for Main Memory Database Management Systems

Tobin J. Lehman
Michael J. Carey

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

This research was partially supported by an IBM Fellowship, an IBM Faculty Development Award, and National Science Foundation Grant Number DCR-8402818.

A Study of Index Structures for Main Memory Database Management Systems

Tobin J. Lehman
Michael J. Carey

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

ABSTRACT

One approach to achieving high performance in a database management system is to store the database in main memory rather than on disk. One can then design new data structures and algorithms oriented towards making efficient use of CPU cycles and memory space rather than minimizing disk accesses and using disk space efficiently. In this paper we present some results on index structures from an ongoing study of main memory database management systems. We propose a new index structure, the T-tree, and we compare it to existing index structures in a main memory database environment. Our results indicate that the T-tree provides good overall performance in main memory.

1. INTRODUCTION

One method for speeding up a database management system is to increase the amount of main memory in hopes of decreasing the amount of I/O traffic. There are two major approaches to using a large amount of main storage, the first approach being to use the memory to provide a large buffer pool. The goal of this approach is to make it possible for most or perhaps all of the data needed for each transaction to be retained in the buffer pool. DeWitt et al [DKO84], Shapiro [Sha85], and Elhardt et al [EIB84] have taken this approach in their work. Minimizing disk accesses still tends to be the primary performance goal for algorithm design when this approach is taken. The other major approach is to use the large amount of memory as the main store for the database. This approach requires a redesign of the database management system — the algorithms and data structures for query processing, concurrency control, and recovery must all be restructured to stress the efficient use of CPU cycles and memory rather than disk accesses and disk storage. The designs proposed by Krishnamurthy et al [AHK85] and Leland et al [LeR85] have been based on this latter approach. In this approach, disk accesses are only an issue for crash recovery purposes. (The database resides in volatile main memory, but recovery information must still reside on disk.)

This research was partially supported by an IBM Fellowship, an IBM Faculty Development Award, and National Science Foundation Grant Number DCR-8402818.

We are currently studying main memory database management systems, evaluating both old and new database algorithms to determine which ones make the best use of CPU cycles and memory in a main memory database environment. Although we assume that there is a large amount of memory available, we are not willing to assume that it is infinite. We therefore judge data structures and algorithms according to both speed and storage efficiency criteria. The first phase of our study is addressing query processing issues — although relations are memory resident, a sequential scan through a relation would be much too slow for most queries. Indexes can provide much faster retrieval, cutting the number of compares down to $O(\log_2 N)$ (via binary or tree searches) or even $O(1)$ (via hash searches). The initial part of our query processing study has addressed index structures for main memory databases, and this is the focus of the remainder of the paper.

Index structures designed for main memory are different from those designed for disk-based systems. A main memory index does not need to store actual attribute values; instead, pointers to the attribute values can be stored in the index, providing several advantages. First, a single data pointer can provide the index with access to both the attribute value of a tuple and the tuple itself. (For fixed length tuples, the address of the tuple is just a negative offset from the field address.) This saves storage, as such pointers are small, on the order of 32 bits, regardless of the length of the attributes themselves.[1] Second, this eliminates the complexity of dealing with long or variable length fields in the index. Third, moving pointers will tend to be cheaper than moving the (usually longer) attribute values. Note that the use of pointers is reasonable here because the cost formulas for using indices are different in a main memory database system. In a disk-based system, following a tuple pointer would cost a disk access, but in a main-memory-based system, the cost of a tuple fetch is simply a memory indirection.

The remainder of this paper is organized as follows: Section 2 introduces a new data structure, called the T-tree, for indexing data in main memory. Section 3 presents an experimental comparison of T-trees and a number of existing index structures. Finally, Section 4 summarizes the results of our study of main memory index structures.

[1] The pointer size can be decreased to 16 bits by storing only an offset into the relation if (segment, offset) addresses are used. This will work for up to 64K byte relations using byte offsets or up to 256K byte relations using word offsets.

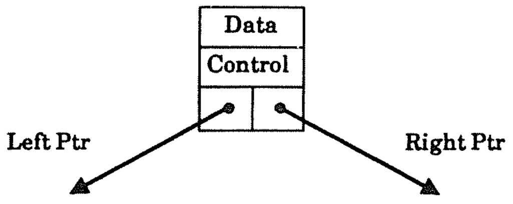
2. THE T-TREE INDEX STRUCTURE

2.1. Motivation — AVL Trees versus B-Trees

The AVL tree is a well-known internal memory data structure [AHU74]. Searching such a balanced binary tree is efficient because the binary search is intrinsic to the data structure; there is no time wasted on arithmetic index calculations. The AVL tree is wasteful of space, however, because there is a large amount of storage overhead associated with each data value, as shown in Figure 1. Each node holds a data value, a left child pointer, a right child pointer, and balance information.

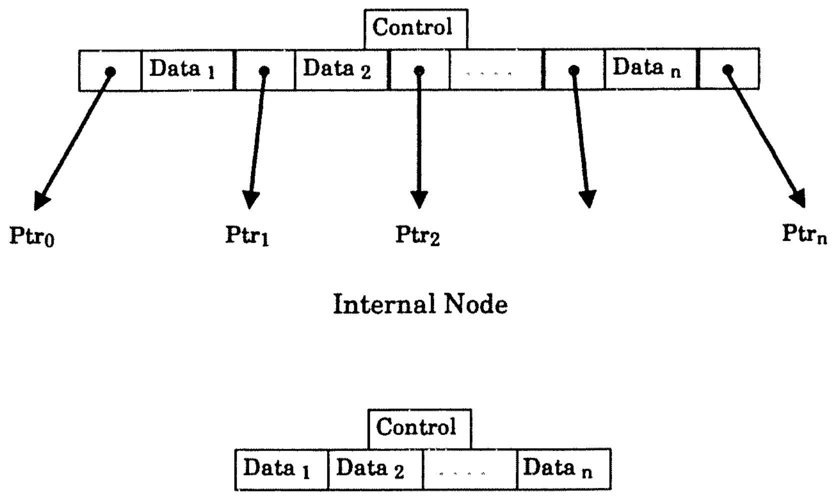
The B-tree is a well-known external memory data structure [Com79]. Its most notable characteristic is that the node fanout can be large, and thus, few levels of the tree need to be searched to retrieve an item from among many elements. Searching a B-tree usually entails a binary search of one node per tree level, although linear node searching is feasible if the nodes are small. The B-tree is much more space efficient than the AVL tree because the leaves contain only data, and the leaves comprise the majority of the nodes in the tree. (The exact percentage depends on the fanout of the tree). Figure 2 shows the internal and leaf node structure for a B-tree.

Having considered the search and storage characteristics of AVL and B trees, we now consider their update characteristics. The B-tree insertion and deletion algorithms keep the tree balanced by splitting or merging nodes and by moving data within or between nodes; in the case of a main memory B-tree index, only data pointers are moved. In contrast, the AVL tree insertion and deletion algorithms rebalance the tree by rearranging nodes in subtrees (called rotations); an AVL tree is considered to be balanced if, for every



The structure of an AVL Node.

Figure 1



The structure of a B Tree internal node and leaf node.

Figure 2

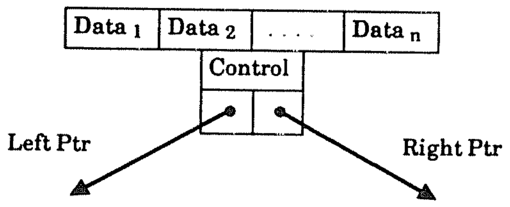
node, the depths of the node's subtrees are within one level of each other (rather than equal as in B-trees). An AVL tree rotation requires between 3 and 5 pointer reassignments, depending on the type of rebalance rotation performed. Despite the fact that an AVL tree rebalance requires reassigning just a few pointers, however, it is likely to be more costly than balancing for a B-tree. In an AVL tree, inserts and deletes are done to leaves, as in a B-tree, but every node in the path from the root to the leaf must have its balance information examined and possibly updated. The B-tree's movement of small blocks of data (or data pointers) is most probably cheaper than traversing an entire path in a tree and examining the nodes, especially since microprogrammed computers usually have a single instruction to quickly move an entire block of data in memory.

2.2. The T-Tree

Based on the preceding discussion, a combination of the AVL tree and B-tree would seem likely to give the best overall performance for a main memory database environment. In particular, it would be beneficial to have a data structure that has the search characteristics of the AVL tree coupled with the storage efficiency and update characteristics of the B-tree. The T-tree, which we introduce in this section, has just such properties.

As mentioned previously (and shown in Figure 1), an AVL tree node has two child pointers plus balance information (probably one byte), but it holds only one data item. The pointer to data ratio is thus two to one. By changing the node so that it holds many items, we can reduce this ratio considerably. We call this type of node a T-node, shown in Figure 3, and we call a tree composed of these nodes a T-tree. The T-tree is a binary tree, so it retains the intrinsic binary search nature of the AVL tree. Data movement is required for insertion and deletion, but it is usually needed only within a single node. Rebalancing is done using AVL tree style rotations, but it is done much less often than in an AVL tree due to the possibility of intra-node data movement.

To aid in our discussion of T-trees, we begin by introducing some helpful terminology. There are three different types of T-nodes, as shown in Figure 4. A T-node that has two subtrees is called an *internal node*. A T-node that has one NIL child pointer and one non-NIL child pointer is called a *half-leaf*. A node that has two NIL child pointers is called a *leaf*. For each internal node A, there is a corresponding leaf (or



The structure of a T Node.

Figure 3

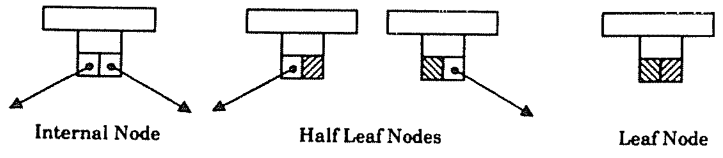


Figure 4

half-leaf) that holds the data value that is the predecessor to the minimum value in A, and there is also a leaf (or half-leaf) that holds the successor to the maximum value in A. The predecessor value is called the *greatest lower bound* of the internal node A, and the successor value is called the *least upper bound* of A, as shown in Figure 5. For a node N and a value X, if X lies between the minimum element of N and the maximum element of N (inclusive), then we say that node N *bounds* the value X. Since the data in a T-node is kept in sorted order, its leftmost element is the smallest element in the node and its rightmost element is the largest.

Associated with a T-tree is a minimum count and a maximum count. Internal nodes and half-leaf nodes keep their occupancy (*i.e.* the number of data items in the node) in this range. The minimum and maximum counts will usually differ by just a small amount, on the order of one or two items, which turns out to be enough to significantly reduce the need for rotations. With a mix of inserts and deletes, this little bit of extra room reduces the amount of data passed down to leaves due to insert overflows, and it also reduces the amount of data borrowed from leaves due to delete underflows. Thus, having flexibility in the occupancy of internal nodes allows storage utilization and insert/delete time to be traded off to some extent. Leaf nodes have an occupancy ranging from zero to the maximum count.

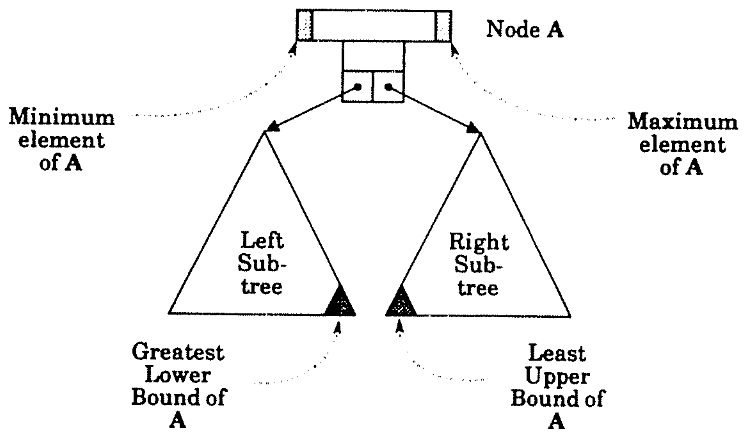


Figure 5

2.3. Search, Insert, and Delete Operations

In this section we present the algorithms for the search, insert and delete operations for the T-tree index structure.

2.3.1. Search Algorithm

Searching in a T-tree is similar to searching in a binary tree. The main difference is that comparisons are made with the minimum and maximum values of the node rather than a single value as in a binary tree node. The algorithm works as follows:

(1) The search always starts at the root of the tree.

(2) If the search value is less than the minimum value of the node, then search down the subtree pointed to by the left-child pointer. Else, if the search value is greater than the maximum value of the node, then search down the subtree pointed to by the right-child pointer. Else, search the current node.

The search fails either when a node is searched but the item is not found or when a node that bounds the search value cannot be found.

2.3.2. Insert Algorithm

The insert operation begins with a search, but the path from the root to the leaf is kept on a stack. Then, if the T-tree is unbalanced as a result of the insertion (or deletion, for that matter), the appropriate rebalancing operation checks the nodes in the path between the root and the leaf where the insertion or deletion took place. Insertion works as follows, although we postpone our discussion of the rebalancing rotations until Section 2.3.4:

(1) Search for the appropriate node, saving the node and the direction (either left or right) taken at each level on the stack.

(2) If a node is found, then see if it has room for another entry. If the insert value will indeed fit, then insert it into this node and stop. Else, remove the minimum element from the node, insert the old insert value, and make the minimum element the new insert value. Proceed from here as in (1). (In

the latter case, the minimum element will be inserted into a leaf and become the new greatest lower bound value for this node.)

(3) If the search exhausts the tree and no node bounds the insert value, then insert the value into the last node on the search path (which is a leaf or a half-leaf). If the insert value fits, then it becomes the new minimum or maximum value for the node. Otherwise, create a new leaf (so the insert value becomes the first element in the new leaf).

(4) If a new leaf was added, then check the tree for balance by following the path saved on the stack. For each node in the search path (going from leaf to root), if the two subtrees of a node differ in depth by more than one level, then a rotation must be performed (see Section 2.3.4). Once one rotation has been done, the tree is rebalanced and processing stops.

A design note is in order here: When an internal node overflows, and thus its minimum value is removed and passed down to a leaf, the insert into the leaf requires no data movement because the value becomes the leaf's rightmost entry. If instead the maximum value had been removed from the internal node, it would have to be inserted as the leftmost entry in the leaf, requiring intra-node data movement. Hence, removing the minimum value instead of the maximum value avoids this data movement.

2.3.3. Delete Algorithm

The deletion algorithm is similar to the insertion algorithm in the sense that the element to be deleted is searched for, the search path is stacked, and then rebalancing is done if necessary. The algorithm works as follows:

(1) Search for the node that bounds the delete value, saving the search path on the stack. Search for the delete value within this node, reporting an error and stopping if it is not found.

(2) If the delete will not cause an underflow (*i.e.* if the node has more than the minimum allowable number of entries prior to the delete), then simply delete the value and stop. Else, if this is an internal node, then delete the value and borrow the greatest lower bound of this node from a leaf or half-leaf to bring this node's element count back up to the minimum; push the search path on the stack while searching

for the greatest lower bound. Else, this is a leaf or a half-leaf, so just delete the element. (Leaves are permitted to underflow, and half-leaves are handled in step (3).)

(3) The stack now holds a path from the root to a leaf or a half-leaf where an item was just removed. If the node at the top of the stack (*i.e.* the last node pushed on) is a half-leaf and has an element count below the minimum count, then borrow from its only child and push the child onto the stack.

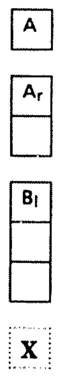
(4) If the current top-of-stack leaf is not empty, then stop. Else, free the node and proceed to step (5) to rebalance the tree.

(5) For every node along the path from the leaf up to the root, if the two subtrees of the node differ in height by more than one, perform a rotation operation (see Section 2.3.4). Since a rotation at one node may create an imbalance for a node higher up in the tree, balance checking for deletion must examine all of the nodes on the search path.

2.3.4. Rebalancing a T-Tree

The rebalancing operations for a T-tree are similar to those for an AVL tree [AHU74]. A T-tree's balance is checked whenever a leaf is added or deleted, as indicated in the sections on the insertion and deletion algorithms. The stacked search path is checked from the root to the leaf — for each node in the path, if the node's two subtrees differ in height by more than one level, a rotation operation is needed. In the case of an insertion, at most one rotation is needed to rebalance the tree, so processing stops after one rotation. A rotation on one node may trigger an imbalance for a node higher up in the tree in the case of a deletion, so processing continues all the way up the path to the root in this case. Figure 6 shows the simple LL rotation and the more complex LR rotation for the case of an insert. These are two of the four types of rotations used to rebalance an AVL tree or a T-tree. The algorithms for the RR and RL rotations are symmetrical to the LL and LR rotations, respectively, so they are not shown.

The T-tree requires one special case rotation that the AVL tree does not have. When an LR or RL rotation is done and the node C is a leaf (so both nodes A and B are half-leaves), a regular rotation would move C into an internal node position, as shown in Figure 7. If C has only one item, which is always the

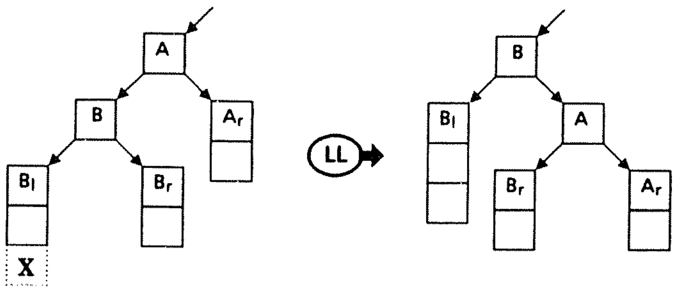


A single box denotes a single node.

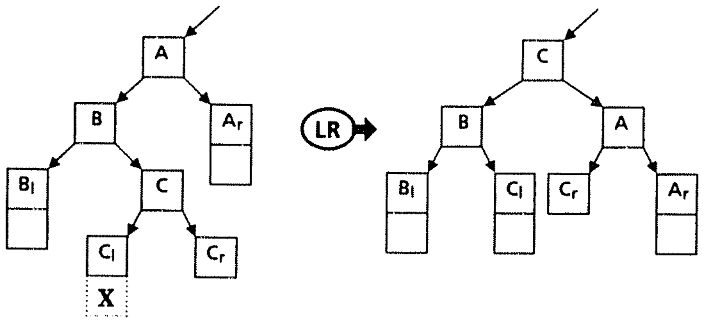
A double box denotes a subtree of depth two.

A triple box denotes a subtree of depth three.

A dotted box with an **X** denotes a level of a subtree that is about to be added or deleted from the subtree.



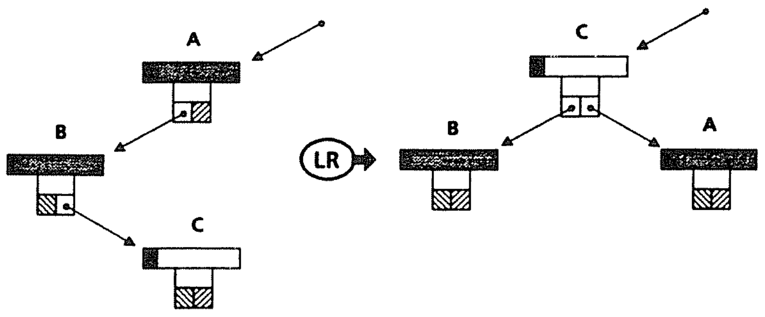
After appending to subtree B_l, the depth of subtree B is two greater than the subtree A_r. An LL rotation on node A is needed to rebalance the tree.



After appending to subtree C_l, the depth of subtree B is two greater than the subtree A_r. An LR rotation on node A is needed to rebalance the tree. Appending to C_r instead of C_l would have resulted in the same situation.

Insert - when the A->Left subtree extends two levels further than the A->Right subtree, then we must use a left rotation to rebalance the tree. We do an **LL Rotation** when the A->Left->Left subtree is longer than its sibling subtree, otherwise we do an **LR Rotation** (the A->Left->Right subtree is longer).

Figure 6



Regular LR rotation

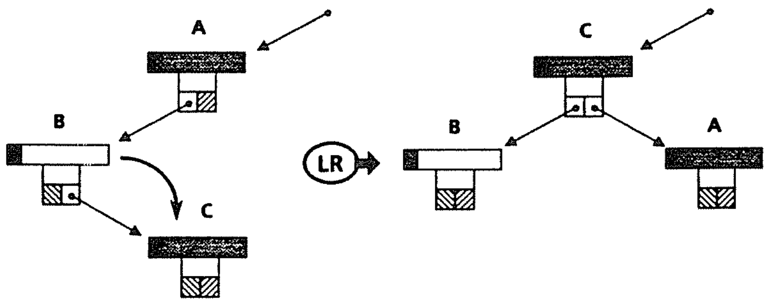
Figure 7

case during an insert, then C would *never* get to hold more than a single item as an internal node (unless it is later rotated back into a leaf position). Since this would be detrimental from a storage utilization standpoint, a special rotation operation first moves values from B to C so that, after the rotation, C is a full internal node. This special case rotation is shown in Figure 8.

Rebalancing after deletion is identical to rebalancing after insertion, but the cause of the imbalance in the tree is that a subtree has grown shorter rather than longer. Figure 9 shows the LL and LR rotations for the case of a delete operation, and the RR and RL rotations are symmetrical, as before.

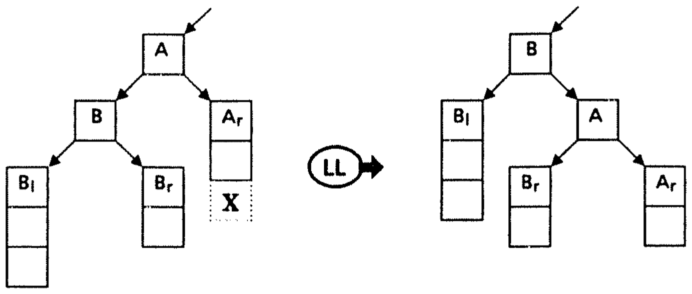
2.4. Another Version of the T-Tree

The algorithms described above perform well and yield storage characteristics similar to the B-tree, as will be seen in Section 3. In order to achieve even better storage utilization, we can modify the insert algorithm as follows, with the algorithms for searches and deletions remaining the same as before. We refer to the T-tree algorithms with these modifications as the "modified T-tree" algorithms in subsequent discussions.

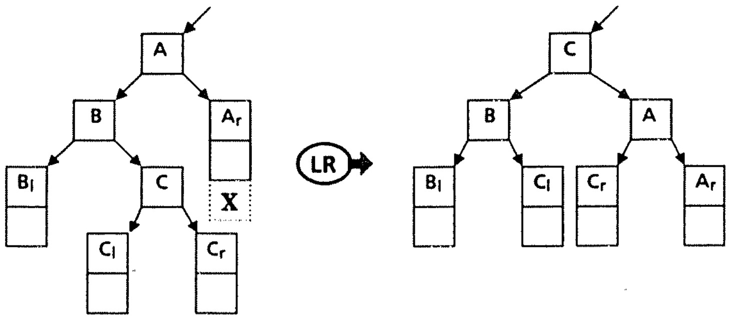


Special LR rotation

Figure 8



After deleting from subtree A_r, A's right subtree is two levels shorter than A's left subtree. An LL Rotation on node A is needed to rebalance the tree.



After deleting from subtree A_r, A's right subtree is two levels shorter than A's left subtree. An LR Rotation on node A is needed to rebalance the tree.

Delete - when A's right subtree shrinks to two levels less than A's left subtree, a left rotation is needed to rebalance the tree. If the two subtrees of A->Left are equal, or if the A->Left->Left subtree is longer, than an LL Rotation is used, otherwise an LR Rotation is used (the A->Left->Right subtree is longer).

Figure 9

Instead of creating new leaves on either side of their parent nodes, we instead allow the creation of leaves only on one side, the left side. If a leaf or half-leaf is full and a value needs to be added to its right side, then we can move the elements in the leaf or half-leaf to the left, pushing the minimum element of the leaf or half-leaf off the left side. This change is shown in Figure 10. If the node is a leaf, then a new left leaf is created. Otherwise, the node is a half-leaf and the value that was pushed off the left side is inserted into its existing child. In the rare event that this child is also full, we make an exception to the rule above, creating a new right child for the original leaf or half-leaf rather than sliding elements over in two nodes.

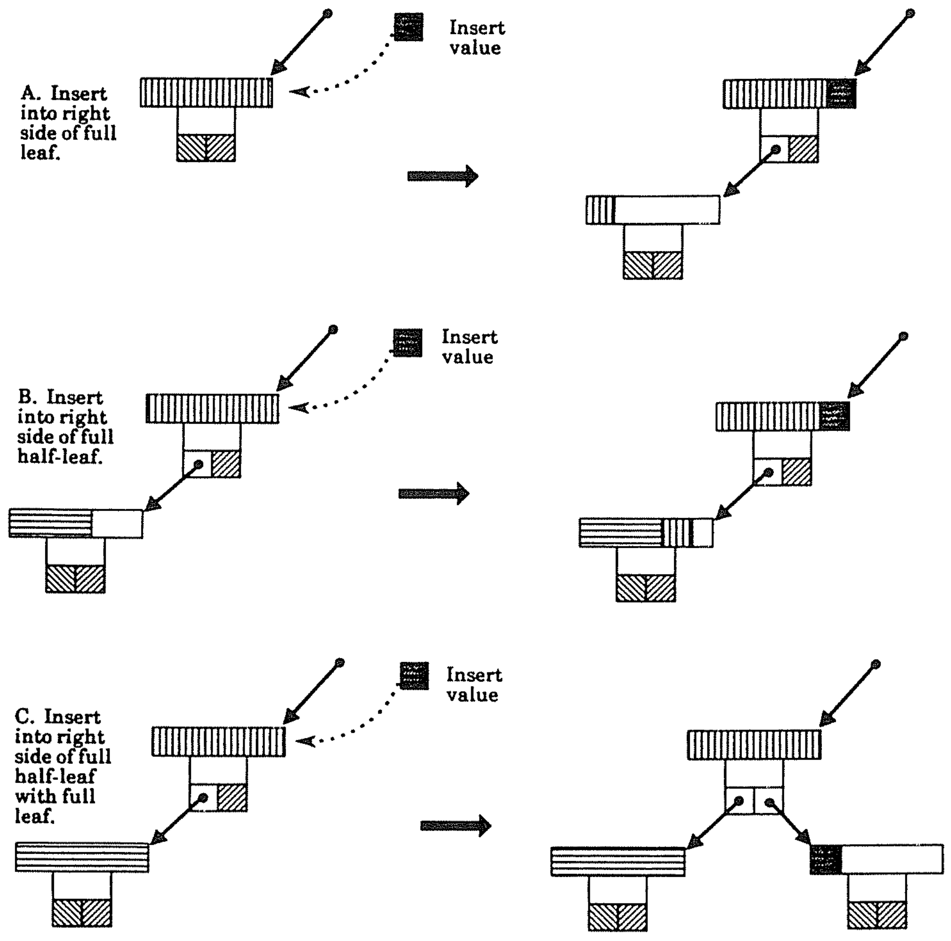

A. Insert into right side of full leaf.
Insert value
B. Insert into right side of full half-leaf.
Insert value
C. Insert into right side of full half-leaf with full leaf.
Insert value

Figure 10

This change improves storage utilization because it reduces the average number of (fairly empty) leaves. (Recall that leaves are the only T-nodes allowed to hold fewer entries than the minimum specified occupancy.)

3. A COMPARISON OF MAIN MEMORY INDICES

To determine the usefulness of the T-tree, we implemented it and tested its performance against implementations of several other candidates for main memory data structures. Since Leland et al [LeR85] use balanced binary trees in their main memory database system design, we implemented the AVL tree index structure. Krishnamurthy et al [AHK85] state that they plan to use simple arrays in their system to save space, so this is another index structure that we implemented. Since B-trees are a common index structure for database systems, we included it in our tests. Note that most database systems actually use the B+ tree, a B-tree variant where the data is held entirely in the leaves and the internal nodes hold copies of leaf values. This modification is not useful in main memory — preliminary tests showed that the B+ tree's main memory performance was the same as the B-tree, but that it used more space to hold the index than the B-tree. We therefore eliminated the B+ tree from consideration.

In addition to these data structures, which can be used for either exact match or range queries, several hashing schemes appear attractive for dealing with exact match queries in main memory database systems. We examined several dynamic hashing schemes, eliminating fixed-size hash tables from consideration since their size must be tuned to the amount of data, which is generally not known. We chose two of the more well-known dynamic hashing algorithms, though there are many others that we did not consider. We felt that extendible hashing [FNP79] and linear hashing [Lit80] would be good dynamic hashing algorithm representatives.

3.1. The Basic Tests

We ran a number of experiments on each data structure to simulate the various demands that a database management system would place on an index. The tests were run in the following order: The data structures were built, then searched for exact matches, then scanned, then searched via range queries, then subjected to two mixes of queries, and then finally half of the entries were deleted. We describe each test in more detail below.

Insert 30,000 Elements — Starting with an empty data structure, 30,000 elements were inserted. The number 30,000 was used because it was the largest number of elements that would not cause "out of memory" problems for some of the data structures in our test environment; still, extendible hashing ran out of memory when run with the smallest node sizes.

Search for 30,000 Random Elements — This test examined the exact match retrieval speed of the data structures. 30,000 different values were searched for, with each value requiring a new search. A third of the values were found, and two-thirds were not found. (This was a characteristic of our random number generator.)

Range Queries — These queries tested the ability of the data structures to retrieve a logical range of values. Rather than trying to pick two values from the structures that appeared N values apart, we searched the data structure for a random value and then retrieved an additional 10, 100, or 1000 elements (there were three range query tests in all). Since the hashed structures do not store values in any logical order, we did not run the range query tests on them; a full sequential scan would be needed for the range queries in their case. The three range query tests were as follows:

Range query 10: 30,000 queries, retrieving 10 elements per query
Range query 100: 3,000 queries, retrieving 100 elements per query
Range query 1000: 300 queries, retrieving 1000 elements per query

Sequential Scan — This tested the scanning speed of the data structures by reading every value in order. For the array and tree data structures, the values were read in key sequential order. For the hashing structures, the values were read in their physical (basically random) order.

Query Mixes — The query mix tests were intended to test the data structures in a "normal" index environment. We ran two query mixes, one composed of eighty percent searches, ten percent inserts, and ten percent deletes, and the other composed of sixty percent searches, twenty percent inserts, and twenty percent deletes. We picked equal percentages of inserts and deletes so that the structures would remain at a constant size, making the results easier to evaluate. To ensure that the delete operations were always successful, we used a modified delete procedure for these tests — if the item was not found, the next closest value was deleted instead. Inserts were always successful (another characteristic of the random number generator). The query mix tests contained the following numbers of index operations:

Query mix 1: 24,000 (80%) searches, 3000 (10%) inserts, 3000 (10%) deletes
Query mix 2: 18,000 (60%) searches, 6000 (20%) inserts, 6000 (20%) deletes

Delete 15,000 Elements — This test examined the deletion cost of the various data structures. We again used our modified delete procedure to guarantee a successful delete each time.

Storage Costs — In addition to measuring the elapsed times for all of the tests detailed above, we also computed the storage cost for each of each of the data structures after building the index with thirty thousand elements. The pointer size used for both data pointers and index node pointers was four bytes. Our cost computations assumed that byte (as opposed to word) alignment is permissible, and we assumed that the smallest unit of storage allocation is a byte. The computations were based on the number (and size) of nodes of various types in existence at the end of the index construction test.

3.2. Reducing the Number of Test Parameters

B-trees, extendible hashing, linear hashing, and T-trees each have several parameters that can be varied. To reduce the potentially large number of tests needed, we ran some preliminary experiments to determine which of the parameters could reasonably be held constant. For each structure, we reduced the number of variable parameters to one, the node size. We did this as follows for the various data structures:

The B-tree has a variable node size, a variable leaf size, and two possible node/leaf search methods, linear and binary search. Preliminary tests showed that varying the leaf and node sizes separately did not affect the results, so we made them the same size for the tests reported here. The linear search times were good for small nodes, but small nodes lead to much worse storage utilizations than larger nodes, so binary search was used for our tests. The T-tree has variable minimum and maximum node sizes, plus it has the same two possible node searching methods as the B-tree. For the same reasons as for the B-tree, a binary node search was used for our tests. As for the minimum node size (*i.e.* the minimum occupancy for non-leaf nodes), we set it to the maximum size minus two — we found that most of the performance gains due to having a variable size were achieved at this setting, and that the effect of allowing still fewer elements was mainly to make the storage overhead worse. Extendible hashing has a variable node size and three different possible node search techniques: linear search, binary search, and hash search. A hash search on each page would have added more page structure, thus wasting space. A binary search would have required keeping the page sorted, thus adding data movement costs for insertion and deletion. Since linear search seemed to be the simplest and quickest method, this was used for the tests. Linear hashing has a variable node size, a variable overflow bucket size, a variable storage utilization factor, and the same three possible node/bucket search techniques. For the same reasons as stated for extendible hashing, linear search was used to search the nodes and the overflow buckets. An overflow bucket that was half the size of the node was found to give reasonable overall performance, and a storage utilization factor of 70 percent gave reasonable results. (80 percent gave slightly better storage efficiency, but reduced performance, while 60 percent increased performance slightly, but worsened the storage efficiency, so a storage utilization factor of 70 percent was used for our tests.) Table 1 summarizes the parameter combinations that our preliminary tests led us to settle on for the data structure comparisons reported here.

3.3. The Test Results

We coded the algorithms for each data structure in the C programming language. The code to move bytes from one location to another in memory was optimized into a single VAX *movc3* instruction, a "move character long" instruction that runs much faster than the equivalent optimized loop. The tests were all run on a bare VAX 11/750 computer with two megabytes of real memory; no virtual memory mechanism was in use at the time of the tests. The timing measurements were taken using "getrusage", a timing facility taken from UNIX 4.2 BSD. The resulting performance thus reflects that of a main memory computer with a processing power of about 0.5 MIPS.

We tested each data structure with a range of node sizes (where appropriate). Each data structure had its best overall performance for a particular node size or set of node sizes. Different tests led to different optimal node sizes for the data structures, however, as there are tradeoffs (*e.g.* space-time tradeoffs) for many of the structures. Table 2 gives the performance of the data structures for each test for a particular

Data Structure	*Parameters*
Array	no parameters
AVL Tree	no parameters
B-Tree	binary search, variable node size
T-Tree	binary search, minimum node size of two less than the maximum node size, variable (maximum) node size
Extendible Hashing	linear search, variable node size
Linear Hashing	linear search, overflow bucket size of half the node size, 70% storage utilization factor, variable node size

Table 1

node size, the node size that we selected as being the overall "optimal" size for the data structure. (Node sizes are specified as the maximum number of entries throughout the paper; the two values for linear hashing are the maximum node size and the corresponding maximum bucket size.) In selecting these optimal node sizes, we weighted the decision in favor of search and query mix performance. The storage ratio figures given in Table 2 are the overall amount of storage used divided by the amount needed to store the data alone. We now consider the results of each of the individual tests in turn.

Insert 30,000 Elements — Graph 1 shows the results of the insertion test. The cost of a series of inserts depends on the search time and the amount of data movement needed to insert each element. The insert cost for the array index is not shown here because it is over ten times larger than the rest of the numbers (see Table 2); an insert into the array requires a great deal of data movement.[2] The hash table indices require no data movement because the new item is simply appended to the end of the heap of values in the node. The search time for these indices is thus constant and dependent on the node size. A larger node size means a longer search time in the node, so only small sizes of nodes are practical. For the tree structures, the search

Test	Array	AVL Tree	B Tree	Extendible Hashing	Linear Hashing	T Tree	T Tree (mod)
Insert 30,000	1338.18	22.97	22.84	22.98	23.49	18.11	20.79
Search 30,000	11.13	8.77	16.79	8.06	8.63	9.50	9.32
Scan 30,000	0.36	2.09	0.58	1.90	0.64	0.56	0.55
Range 10	15.11	37.86	24.27	na	na	20.88	20.66
Range 100	5.78	24.52	9.07	na	na	8.51	8.42
Range 1,000	4.68	22.56	7.37	na	na	7.09	7.02
Query Mix 80	253.43	12.20	18.24	10.72	14.92	12.33	12.37
Query Mix 60	600.56	15.22	19.53	12.77	21.66	15.12	15.45
Delete 15,000	280.47	11.73	12.02	8.16	13.19	12.64	12.74
Node Size	na	na	40	8	6/3	26	26
Storage in bytes	120000	390000	182190	314944	183837	182721	168530
Storage ratio	1.0	3.25	1.52	2.62	1.53	1.52	1.40

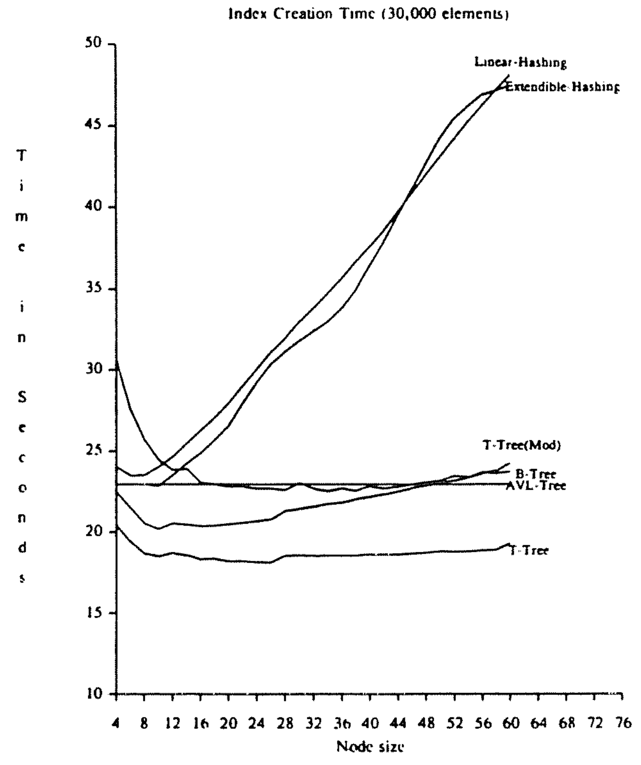
Best times for each data structure

Table 2

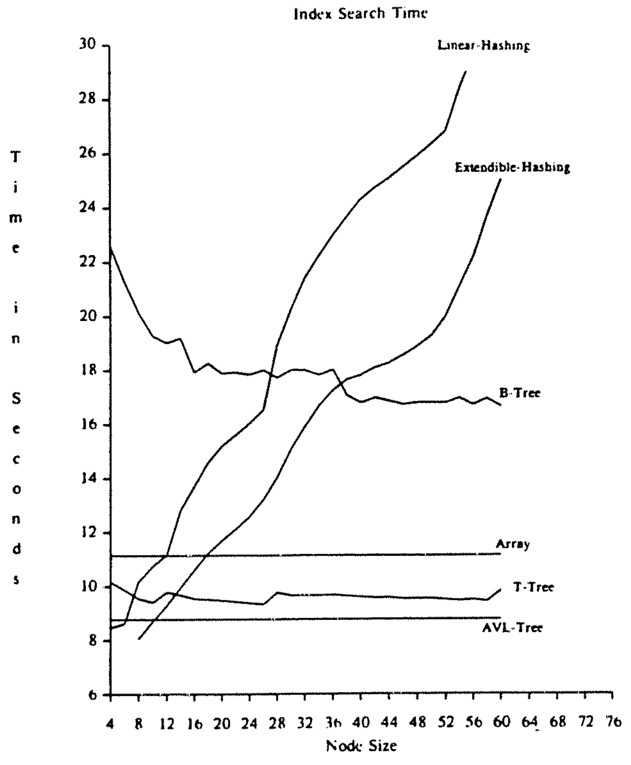
[2] If the array were used as an index, a practical way of building it would be to do a bulk insert into the array as a heap and then quicksort it.

cost is $O(log_2 N)$, so it increases as the number of elements grows (but the increase is slow). The B-tree and T-tree require data movement because an insert into a node requires moving half the items in the node (on the average).[3] AVL tree searches are faster than B-tree searches, but the B-tree can do the actual element insertion and rebalancing faster than the AVL tree; thus, their times for this test are nearly the same. The T-tree has the insert costs of the B-tree (since it can slide data within a node) and the search costs of the AVL tree (due to its intrinsic binary search nature). The T-tree is therefore the clear winner in the index creation test. The modified T-tree (presented in Section 2.4) suffers due to increasing data movement as the node size gets large, but it does perform second best out to a node size of 50 or so.

Search for 30,000 Random Elements — Graph 2 shows the results of the search test. The array uses a pure binary search. The overhead of the arithmetic calculation and movement of pointers is noticeable when compared to the "hardwired" binary search of a binary tree. In contrast, the AVL tree needs no arithmetic



Graph 1



Graph 2

[3] Without the VAX 11/750 block move instruction, the T-tree and B-tree times would be about ten percent slower for node sizes of twenty and even worse for larger nodes.

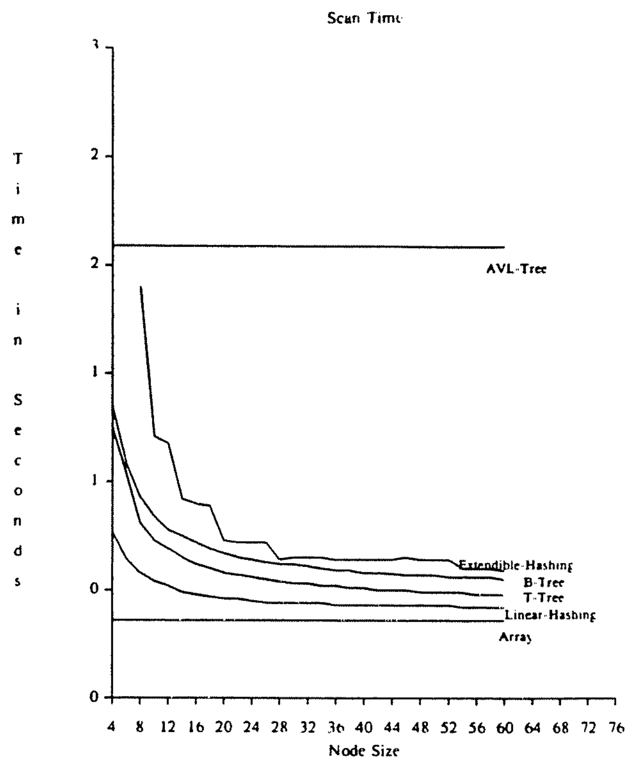
calculations, as it just does one compare and then follows a pointer. The T-tree does the majority of its search in a manner similar to that of the AVL tree. Then, when it locates the correct node, it switches to a binary search of that node. Thus, the cost of the T-tree search is slightly more than the AVL tree search cost, as some time is lost in binary searching the final node. The hashing schemes have a fixed cost for the hash function computation plus the cost of a linear search of the node and any associated overflow buckets. There is a space-time trade-off with the hashing schemes: Small nodes allow the search to make fewer compares, but also require a large amount of space. Large nodes are more space efficient, but the larger sets of values require more time to search through.

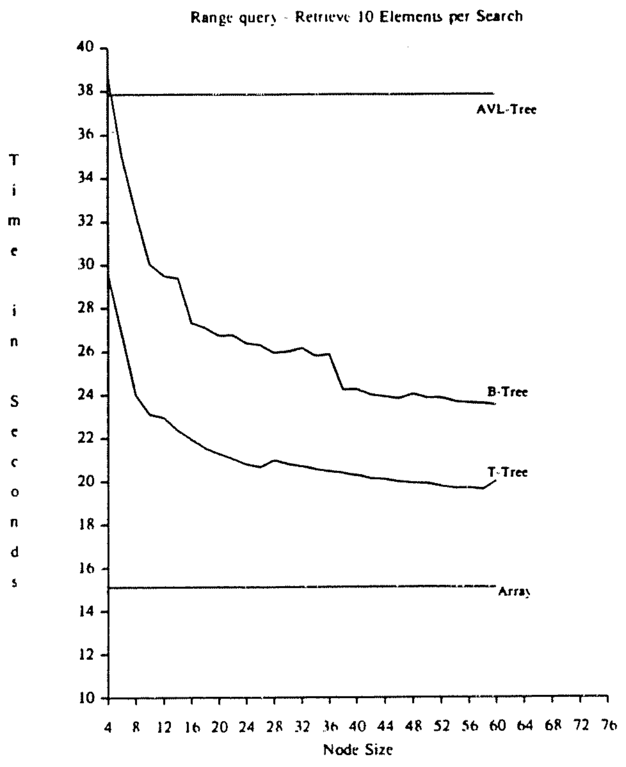
Using a bucket size of eight, extendible hashing has the best search time. A smaller bucket size probably would have resulted in a faster search time, but we did not have sufficient memory available to run that test. The AVL tree, linear hashing, and the T-tree are just a bit slower. The search time of the B-tree will eventually approach that of the array as its node size gets very large — it uses a binary search on each node, but suffers the search startup and termination overheads for each node in the search path.

Sequential Scan — Graph 3 shows the results of the sequential scan test. For most of the structures, scanning is quite fast. The main exception is the AVL tree, as it must traverse a pointer for every data item referenced. Pointer traversal dictates that smaller node sizes lead to larger scan times and vice versa.

Range Queries — Graphs 4 to 6 show the results of the range query tests. Range queries consist of two parts, a search for the initial value, and then a scan of the data structure until the last value in the range is encountered. The scanning cost appears to be the predominant factor in this test. The array is the winner here, with the T-tree coming in second, the B-tree third, and the AVL tree last. The reasons for these results are similar to those for the scan test. (We remind the reader here that since hash tables do not preserve the logical ordering of the values, they were not tested for range queries.)

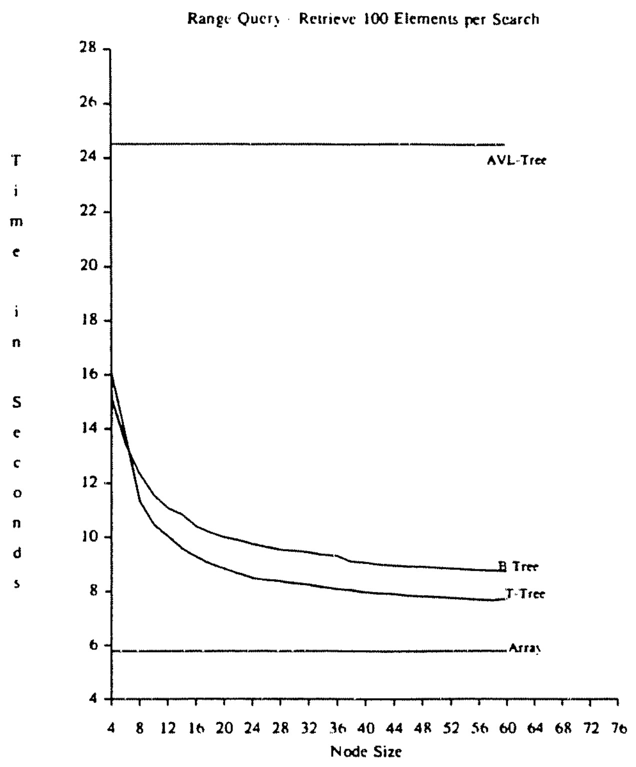
Query Mixes — Graphs 7 and 8 show the results of the query mix tests, the tests that we view as the "main event" for comparing overall index performance. Two different mixes were used, one composed of 80 percent searches, 10 percent inserts and 10 percent deletes, and another mix composed of 60 percent searches, 20 percent inserts and 20 percent deletes. The inserts and deletes were uniformly distributed

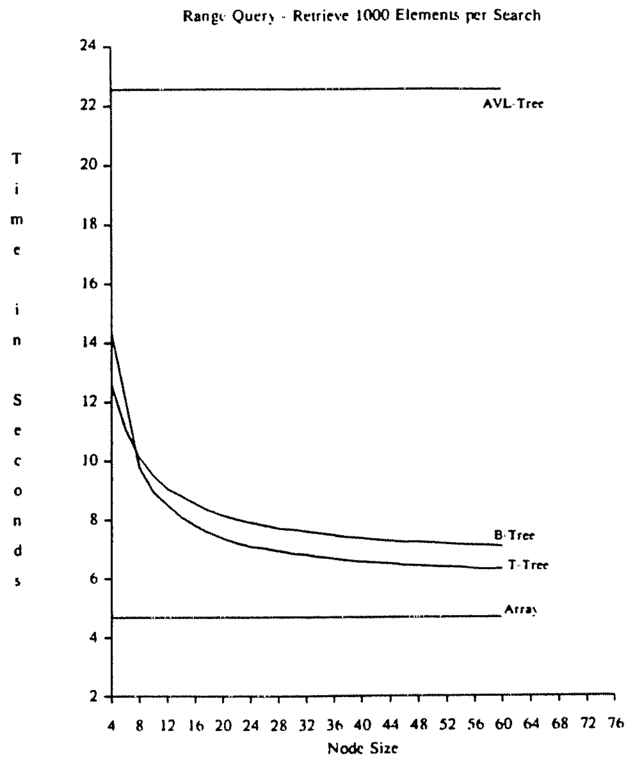




Graph 3

Graph 4





Graph 5

Graph 6

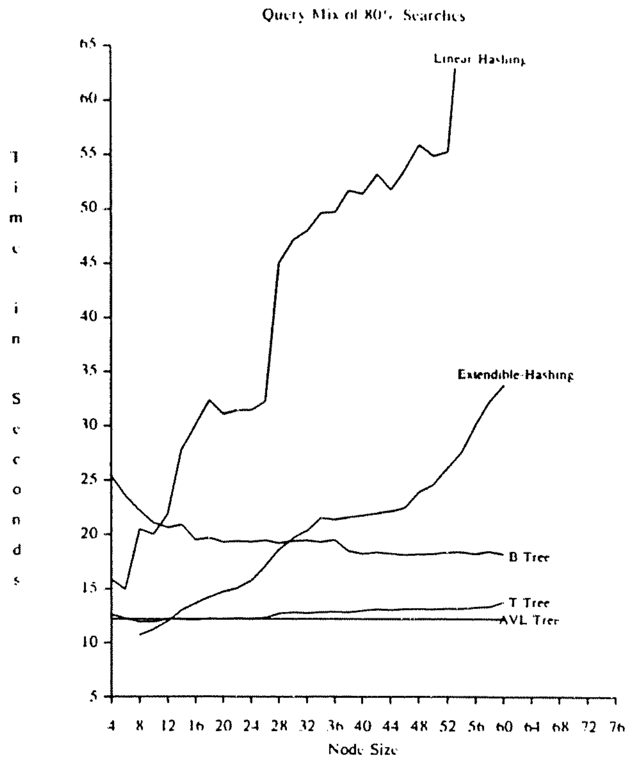
throughout the duration of the tests so that the data structures would remain at an average size of 30,000 elements. The AVL tree, the T-tree, and extendible hashing provided the best performance for each of the mixes.

Delete 15,000 Elements — Graph 9 shows the results of the deletion test, where half of each data structure was deleted. The hashing structures have the best delete times. The T-tree and B-tree have similar times at their assumed optimum node sizes of 26. (Recall that the optimum node size is a function of both overall performance and storage utilization.) We are unsure exactly why it is that the B-tree times decrease for larger node sizes while the T-tree times increase for larger node sizes.

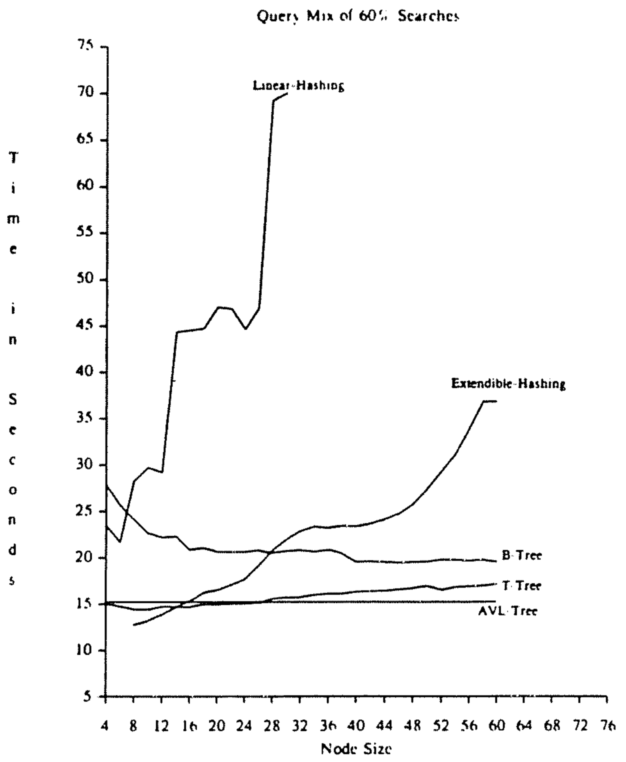
Storage Costs — Finally, Graph 10 shows the average case total storage cost (in bytes) for each of the data structures. The total storage cost is simply the total number of bytes needed to hold the data structure. The AVL tree and the array serve as constant upper and lower storage cost bounds, respectively. The B-tree and T-tree storage costs are almost identical. The modified T-tree uses 10 to 15 percent less space than the regular T-tree. Linear hashing has quite good storage characteristics for medium and large size nodes, but it has good performance only for small size nodes. Extendible hashing has very poor storage utilization for small nodes. (In fact, we could not run its tests for node sizes smaller than eight, as the hash table grew so large that it exhausted the two megabytes of available memory.)

3.4. Comparing Apples and Oranges?

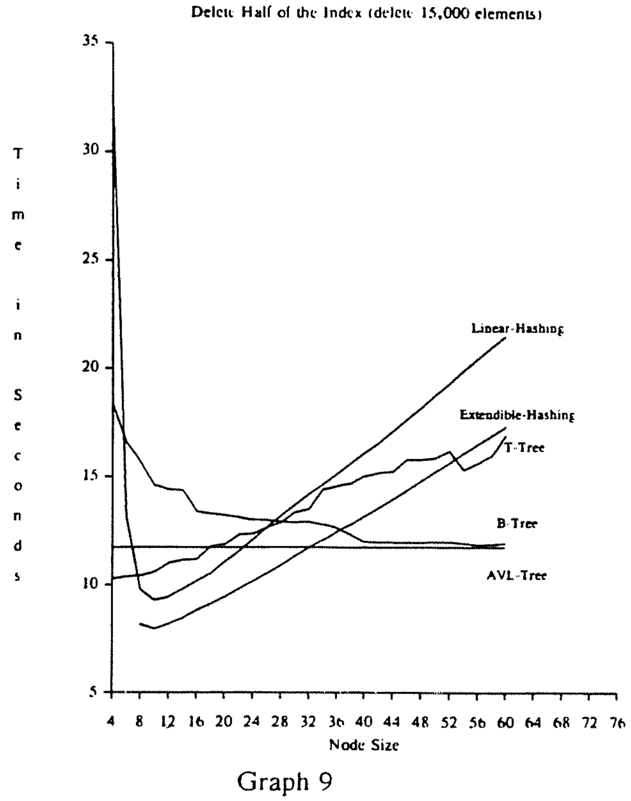
A major problem with the type of comparison reported here is the issue of *fairness*. Since the results are close in some cases, it might seem that any number of constant factors could come into play and alter the results. We did our best to implement each of the index algorithms equally well, so we feel that the results do indeed provide a reasonably fair comparison of the algorithms. Still, some algorithms had small advantages because of the test environment. In particular, the data used for all the tests was generated by a random number generator and the resulting randomness helped the hashing schemes. This aided in randomly distributing keys in these schemes, preventing data "clumping", and it allowed us to use a simple hash function to get the desired results. A more complicated hash function would cost more to compute, so it may be that our hash index results are slightly optimistic. In order to provide for a fairer and less VAX-specific comparison



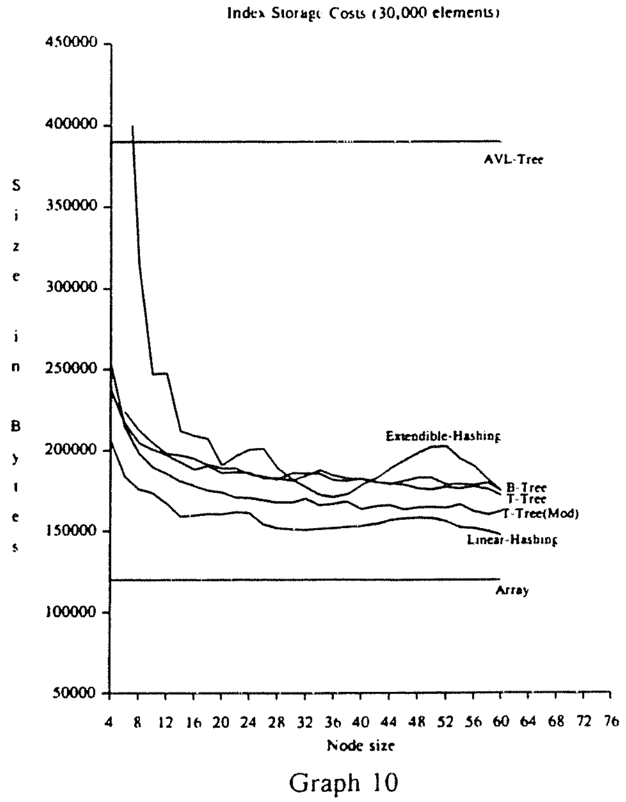
Graph 7



Graph 8



Graph 9



Graph 10

of the index structures, we plan to mathematically analyze the various algorithms in detail in the future. The metrics of interest will include the number of key comparisons, the number of block moves and their average block size, the number of pointer copies, the number of hash function calculations, and the storage overhead of the algorithms. (The results will thus be less VAX-specific in the sense that the component costs can be weighted in various ways, rather than according to the relative costs for a VAX 11/750, when computing the overall costs for various index operations.)

4. CONCLUSIONS

In this paper we introduced a new main memory index structure, the T-tree. We compared the T-tree structure against AVL trees, simple arrays, B-trees, extendible hashing, and linear hashing. Our results indicate that the T tree provides excellent overall performance for mixes of searches, inserts, and deletes, and that it does so at a relatively low cost in storage space. We plan to use the T-tree as the primary access method for our prototype implementation of a main memory database management system, perhaps with a variant of linear hashing available for indexing relations for which only exact match queries are relevant.

5. ACKNOWLEDGEMENTS

We thank Joel Richardson for helping us to implement the hashing index structures and for his helpful comments. We thank Bruce Bullis for his comments and suggestions. Also, the NSF-sponsored Crystal multicomputer project at the University of Wisconsin provided the many VAX 11/750 CPU-hours that were required for the index structure performance study.

6. REFERENCES

[AHU74] A. Aho, J. Hopcroft and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company, 1974.

[AHK85] A. Ammann, M. Hanrahan and R. Krishnamurthy, "Design of a Memory Resident DBMS", *IEEE COMPCON Proceedings*, February 1985.

[Com79] D. Comer, "The Ubiquitous B-Tree", *ACM Computing Surveys* 11(2), June 1979.

[DKO84] D. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker and D. Wood, "Implementation Techniques for Main Memory Database Systems", *Proceedings of the 1984 SIGMOD Conference on Management of Data*, June 1984.

[ElB84] K. Elhardt and R. Bayer, "A Database Cache for High Performance and Fast Restart in Database Systems", *ACM Transactions on Database Systems* 9(4), December 1984.

[FNP79] R. Fagin, J. Nievergelt, N. Pipenger and H. Strong, "Extendible Hashing — A Fast Access Method for Dynamic Files", *ACM Transactions on Database Systems* 4(3), September 1979.

[LeR85] M. Leland and W. Roome, *The Silicon Database Machine*, working paper, AT&T Bell Laboratories, 1985.

[Lit80] W. Litwin, "Linear Hashing: A New Tool For File and Table Addressing", *Proceedings of the 6th VLDB Conference*, October 1980.

[Sha85] L. D. Shapiro, "Join Processing in Database Systems with Large Main Memories", submitted to *ACM Transactions on Database Systems*, 1985.