COMPACT 0-COMPLETE TREES

Ratko Orlandic
John L. Pfaltz

Department of Computer Science
University of Virginia
Charlottesville, VA 22903

Abstract:

In this paper, a novel approach to ordered retrieval in very large files is developed. The method employs a B-tree like search algorithm that is independent of key type or key length because all keys in index blocks are encoded by a $\log_2 M$ bit surrogate, where M is the maximal key length. For example, keys of length less than 32 bytes can be represented by a single byte in the index. This ensures a maximal possible fan out and greatly reduces the storage overhead of maintaining access indices.

Initially, retrieval in a binary *trie* structure is developed. With the aid of a fairly complex recurrence relation, the rather scraggly binary trie is transformed into a compact multi-way search tree. Then the recurrence relation itself is replaced by an unusually simple search algorithm. Finally, a specific access scheme, appropriate for secondary indexing, is presented.

1. Introduction

The principle *Sapere aude* (get courage to learn) potentially favors things whose essence is hidden, implying that they are more attractive and more challenging. If viewed through the spirit of this principle our paper, which introduces a new method of accessing and maintaining large files, would appear to have an aesthetic flaw. It first reveals the essence of the access mechanism and only then examines its surface appearance and characteristics. In this case, the essence of the method will be described in terms of a special 0-complete binary trie in section 2. The irregularity of these trees should not discourage the reader; in section 3, a recurrence transformation will open the possibility of a much more compact equivalent. These *compact* C_0-trees will support a simple and efficient search algorithm described in section 4. Finally, insertion and deletion operations on C_0-trees will be discussed in section 5. Throughout the paper the structure is contrasted to B-trees [BaM72].

The admitted generality of B-trees [Com79], their effectiveness and widespread use, has tended to stifle further research in the field of data access. The psychological barrier imposed by the question "will anyone care, since there is already a generally accepted satisfactory solution" must loom over any research. But research interest is still there. Several schemes [Fae79, Lar80, Lit80, LiL87] that provide better access performance have been published in the last decade. Unfortunately, all of them lack at least one of the important advantages of B-trees. B-trees provide relatively fast, balanced access, ensure good storage utilization (at least 50%), support gradual expansion and shrinking, allow ordered sequential access to data items, provide insurance against the catastrophic behavior, are relatively simple to implement, etc. Perhaps more important than any of these, at least in comparison to extensible hashing techniques, is that B-tree retrieval does not depend upon storage in a particular location. Consequently, B-trees can be used to support secondary access paths.

Since B-trees provide both satisfactory primary access and good secondary access to data sets, they naturally tend to be the method of choice. This is unfortunate since, unlike either extensible hashing or our compact 0-complete trees, every indexed attribute value* must be replicated in the index itself. The cumulative effect of replicating many secondary index values is to create indices which exceed, in many cases, the size of the database itself. This overhead can force database designers to reject potentially useful access paths. And, inclusion of search values within blocks of the B-tree significantly

* Actually every value but one.

This work was supported in part by JPL Contract #957721

decreases the block fan out, and increases tree depth and retrieval time.

C_0-trees eliminate search values from secondary indices altogether. They are replaced with small surrogates whose typical 8 bit length will be adequate for most practical key lengths (of less than 32 bytes). Thus actual values are confined to the main file containing principal records, leaving the secondary indices to be just hierarchical collections of (surrogate, pointer) pairs. This organization reduces the size of the secondary indexes (50% - 80%), and increases branching factor of the trees, thus providing a reduction of number of disk accesses per exact match query.

It is important to note that this has been accomplished while still retaining most of the merits of B-trees. In contrast to many proposals aimed to improve on fan-out of B-trees [BaU77, dTv87, Lom81], the entries in C_0-trees have fixed size, are insensitive to types and lengths of search values and require no structural difference between lower and upper levels of the trees. These three properties provide considerable software simplification. In addition, while the efficient in-core search, which requires just the comparison of small integers, is inherently sequential, it does not suffer from presence of variable length index entries [BaU77, Lom81], nor does the scheme impose any artificial bound on the data set size by requiring that a part of the structure be always in-core resident [dTv87, Lit81]. What can be considered as a drawback of the scheme, in contrast to B-trees, is the failure to guarantee even splitting of index blocks, and the presence of an estimated 18% dummy index entries in the lower level of the tree. Although our experiments do not indicate that these problems are significant, a way they can be minimized is discussed in the last section of the paper.

2. 0-complete Trees

A *trie* is a multi-branching edge labeled tree in which items to be retrieved (they may be a single item or a page of items) are stored at its leaves. Re*trie*val [Fre60] is achieved by successively comparing symbols in the search key with edge labels and following the indicated path to the desired leaf. Every arc in a binary tree can be labeled with either a 0 or a 1. Thus any binary tree can be regarded as a binary *trie*. A node (interior or leaf) is called a **0-node** (**0-leaf**) if its entering arc is labeled 0. Similarly, we define a **1-node** (**1-leaf**) to be a node whose entering arc is labeled 1. With arc labels, every node *n* of a binary tree can be uniquely identified by its access path from the root; which we define to be a binary string obtained by concatenating labels of edges traversed from the root down to the node. This string we call **path(n)**. The length of *path* (*n*) is the **depth** of the node *n*.

Paths to the nodes in a binary tree will normally be variable length. At the same time, we require search values (further called **keys**) to be strings of binary digits with arbitrary length up to some maximum value *M* in bits. Throughout this paper we will enforce numbering of bits in the search values from left to right, leftmost bit being at position 1. For simplicity, we assume key uniqueness, but it is not required by the method itself. The path concept can be extended to provide a fixed length node identifier, called its *discriminator*, as follows. The **discriminator**, D_n, of a node *n* is a binary string of fixed length *M* (maximal key length in bits) whose high order *l* bits are *path* (*n*) of length *l* and all other bits are 0's. Thus a node discriminator has an integer value $x \cdot 2^{(M-l)}$, where x is the integer equivalent of *path* (*n*) of length *l*. Consequently, if the key length is 8 bits, a node *n* with *path* (*n*) = 0101, has discriminator 01010000. Its integer equivalent is 80_{10}.

A binary tree with N leaves is said to be **0-complete** if (a) the sibling of any 0-leaf is present in the tree, and (b) there are exactly $N-1$ 1-nodes in the tree.

A binary tree is said to be **complete** if every node is either a leaf or has exactly two nonempty descendants. It is not hard to see* that every complete binary tree (as in figure 1) satisfies the two conditions for 0-completeness. By arbitrarily deleting 1-leaves whose 0-siblings are not leaves (e.g. the 1-leaves E and G in figure 1 to obtain figure 2), we preserve 0-completeness. But deletion of any 0-leaf would violate condition (b) which requires N-1 1-nodes in a tree with N leaves.

The nodes of any tree can be topologically ordered by a traversal of the tree. We may define the **pre-order traversal** of a 0-complete tree to start at the root of the tree and then iterate the following two steps until the last node has been accessed:

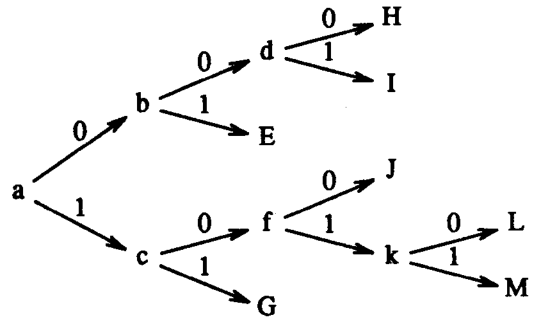


Figure 1. Complete binary tree.

* Proofs of assertions about 0-complete trees can be found in [OrP88].

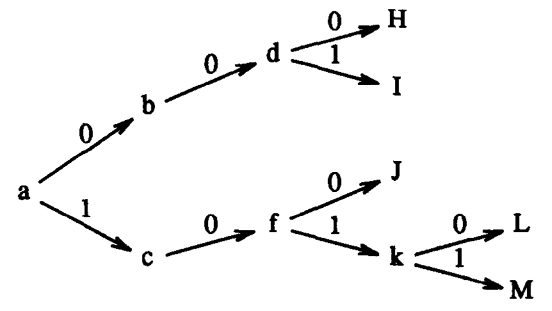


Figure 2. 0-complete binary tree.

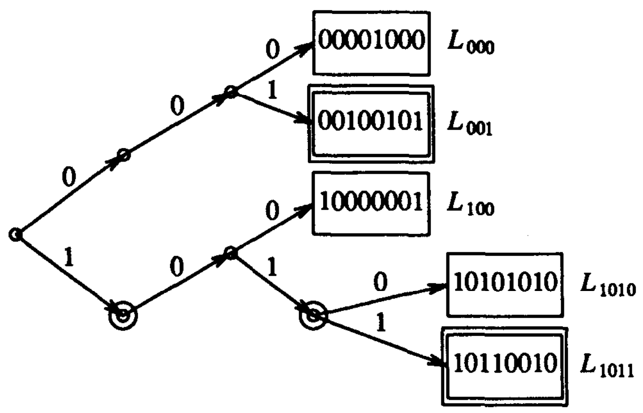


Figure 3. A 0-complete tree.

(a) if the current node n_i is an internal node then the next node n_{i+1} in the order will be its 0-son (by 0-completeness every interior node must have its 0-son);

(b) if the current node n_i is a leaf then the next node in the pre-order will be the 1-son of the node p whose 0-subtree contains n_i and whose depth is maximal.

The pre-order traversal of figure 2 is the sequence a b d H I c f J k L M , where the leaf symbols have been capitalized for emphasis. It is not hard to show that this definition is equivalent to the more usual recursive definition of pre-order traversal.

Since data items are stored at the leaves, they are of special importance in these retrieval trees. In the 0-complete tree of figure 3 they have been represented as rectangles; each contains the key* of a representative item and each has been labeled by its path. Successor nodes to leaf nodes in the pre-order traversal of a 0-complete tree are also of special importance; we call the successor of a leaf node its **bounding node.** In figure 3, the bounding nodes, which may or may not be leaf nodes, have been emphasized with double lines. Since bounding nodes are defined in terms of the pre-order traversal it follows that each leaf, except the last one, has its own unique bounding node. From part (b) of the definition of the pre-order traversal it follows that every bounding node is a 1-node. Moreover, it can be shown that in a 0-complete tree every 1-node is also a bounding node of a unique leaf.

Discriminators and bounding nodes can now be used to establish a **key interval** of the key space that corresponds to each leaf in the 0-complete tree. This interval is formally defined to be the key range between the leaf's discriminator (inclusively) and the discriminator of its bounding node (non-inclusively). The exception is again the last leaf in the pre-order traversal. The upper bound of its interval is always known in advance and consists of all 1 bits: $11...1 = 2^M-1$.

* For clarity in this paper we use only 8 bit keys. But they can be unbounded. Notice, we require only one data item per leaf, but this too can be relaxed.

3. Compact Representation of 0-complete Trees

In the figure 4(a) the key intervals of the 0-complete tree from the figure 3 are listed in lexicographic order. (The lexicographic order coincides with the sequencing of corresponding leaves as they are accessed in the pre-order traversal.) Observe that the (non-inclusive) upper bound of any leaf's key interval is the discriminator of its bounding node, which is, in turn, the (inclusive) lower bound of

```
00000000 - 00100000     00100000 - 3
00100000 - 10000000     10000000 - 1
10000000 - 10100000     10100000 - 3
10100000 - 10110000     10110000 - 4
10110000 - 11111111     11111111 - 0

        (a)                  (b)
```

Figure 4.
Key intervals of leaves in the 0-complete tree of figure 3.

the following key interval. Thus, knowledge of bounding node discriminators is sufficient to identify the appropriate key interval of any data item with any given key. A B-tree like search procedure, which examines the bounding discriminators of the tree in their pre-order traversal sequence, will find the correct key interval when the first discriminator D_i greater than the search key K is found. D_i will be the non-inclusive upper bound of the interval; D_{i-1} is its inclusive lower bound.

The implicit switch of roles here is important. The leaf's own discriminator is not used to represent either it or its key; but rather the discriminator of its bounding node. In general, node discriminators need not be unique, but it can be shown that in a 0-complete tree all bounding node discriminators are distinct.

Figure 4(b) contains the bounding discriminators listed in order. Along with each entry there is a number denoting the depth of the bounding node with that discriminator. For the last entry, which has no bounding node, we have chosen an imaginary discriminator with assigned depth 0. Henceforth we will assume that the bounding node of the last leaf always has depth 0. There is one apparent regularity in the relationship between the discriminators of a set of bounding nodes and their depths. If the depth of a bounding node is d, then by the definition of a discriminator, the d^{th} bit of the corresponding discriminator is set to 1. We will use this to develop a correspondence between the entire upper bound discriminator and the depth of its associated bounding node.

Let D_i denote the discriminator of the i^{th} bounding node in the pre-order traversal. Let the key length be M, let an initial dummy discriminator D_0 be 0, and let the depths of the bounding nodes be the ordered set L = < d_i >, i = 1, N-1, where N denotes the number of leaves in the tree. Then D_i can be obtained from D_{i-1} by:

(1) setting the d_i^{th} bit in D_{i-1} to 1; and

(2) setting all subsequent (lower order) bits to 0.

The algorithm can be more precisely expressed with a recurrence relation:

$$D_i = \begin{cases} 0 & \text{if } i = 0 \\ \left(\left\lfloor D_{i-1} / 2^{(M-d_i)} \right\rfloor + 1\right) \cdot 2^{(M-d_i)} & \text{if } 0 < i < \text{N} \end{cases} \quad (1)$$

where $\lfloor x \rfloor$ denotes the floor of x. For i = N the discriminator D_N is $2^M - 1$, as agreed before. Proof that the recurrence relation does indeed produce the ordered set of bounding discriminators can be found in [OrP88]. Given only the depths of the bounding nodes in a 0-complete tree, *in their pre-order sequence*, it is possible to reconstruct the corresponding bounding node discriminators, and hence the corresponding key intervals.

In a new structure, which we call a **compact representation**, or C_0-tree, the *depths* of the bounding nodes of each leaf (not the depth of the leaf itself) are stored in *pre-order* sequence, along with pointers to the leaf pages. *Pointer*$_i$ points to the leaf L_i preceding the i^{th} bounding node in the pre-order. Only these (depth, pointer) entries are kept in a C_0-tree (see figure 5). Every leaf is just an abstraction of a single record, along with its key. Thus, physically, records may be stored anywhere in arbitrary order. But they can be accessed in lexicographic order through the C_0-tree. Keeping in mind the use of C_0-tees for secondary retrieval this means that both records and their keys are entirely eliminated from secondary indices. This is the source of real storage savings, offered by the scheme.

If one assumes that the maximal length of any key is 31 bytes (= 248 bits), then a short 8 bit integer can be used to record depth, since depth value cannot exceed the length of the key field ($0 \leq$ depth $\leq 2^8$-1 = 255). It saves storage and increases fan out. And longer key lengths are exceedingly rare. The actual constraint is that the surrogate, depth, must be coded as an integer of $\log_2(M)$ bits, where M denotes the maximal key length. Let 3 bytes (= 24 bits) serve as a pointer. Then, for large files (of up to 2^{24} = 16.6M items) the total length of an index entry (depth and pointer) need not be longer than 4 bytes. A C_0-tree index page of 1K bytes will support a branching factor of 256. Assuming 9 byte keys (e.g. social security numbers), a corresponding B-tree implementation would have fan-out of only 85.

4. Access in C_0 - trees

The preceding paragraphs, together with the results of section 3, indicate that an effective search procedure using C_0-indexes is feasible. Given a sequence of depths, the recurrence relation above could be used to reconstruct

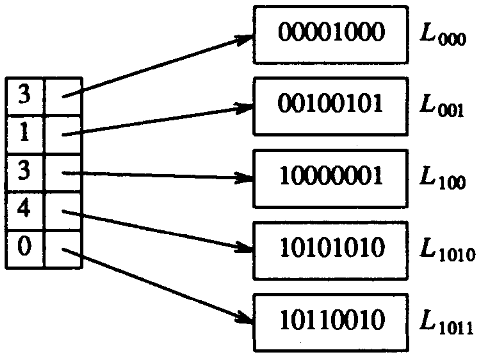


Figure 5. Compact representation of Figure 3.

the discriminator that can be used as the bounding value in the search algorithm. But any procedure implementing the recurrence (1) must be slow; it requires repeated divisions and shifting. Surprisingly, the recurrence can be replaced with the simple search algorithm given in figure 6.

Let b_1 denote the position of the first 1-bit in a key K. Let b_2 denote the position of the second, and b_k the position of the k^{th} 1-bit. So if $K = 10011010$, then $b_1 = 1$, $b_2 = 4$, $b_3 = 5$, and $b_4 = 7$. Let B denote the sequence of all b_i's in K, sorted in their ascending order. This sequence is augmented with a final value equal to $M + 1$. Thus, $B_K = < 1, 4, 5, 7, 9 >$. Any search key K can thus be written as

$$\sum_{k=1}^{l} 2^{(M-b_k)},$$

where l is the cardinality of B. Notice that the final value adds the constant fraction $2^{-1} = 1/2$ to K when it is interpreted as an integer. The use of the parameter *first* will become clear later. For now we assume that *first* is always equal to 1.

While we have not yet shown how to create C_0-trees such as figure 5, the reader should use the algorithm to search for items whose keys are $K_1 = 10000001$ and $K_2 = 01010001$. Note that although no item with key K_2 exists in the file, the accessed leaf lies in the correct key interval.

```
procedure search
input:  [1] An array  B = <b[k]> of sorted
        1-bit positions in the search
        key, appended with the value M+1.
        [2] An integer first, denoting
        position in array B (usually 1)
        at which to begin comparison.
        [3] A sequence L = <d[j]> of
        depths of the bounding nodes
        in a 0-complete tree.
output: The index j of the entry whose
        interval contains key K.
begin
j ← 1;
k ← first;
while b[k] ≤ d[j] do
        begin
        if b[k] = d[j]
           then k ← k+1;
        j ← j+1
        end
return j
end;
```

Figure 6. Search algorithm.

We conceptually subdivide the execution of the algorithm into $s \geq 1$ iterations. For $k < s$ every k^{th} iteration ends when the search procedure encounters the entry i_k, such that $d_{i_k} = b_k$. The next iteration starts from the $(i_k + 1)^{st}$ entry in the index block. The last, i.e. s^{th}, iteration discovers the entry i_s such that $d_{i_s} < b_s$ and the procedure stops. Thus, $1 \leq i_1 < i_2 < \cdots < i_s$, where $s \geq 1$. The number of iterations s must be less than or equal to $l = |B|$.

Lemma 4.1: For every entry i_k in a C_0-index, such that $1 \leq k < s$, the corresponding discriminator is:

$$D_{i_k} = \sum_{j=1}^{k} 2^{(M-b_j)} < K.$$

Proof: Let D_j be the discriminator value after j steps of the recurrence procedure (1). Let $k = 1$, corresponds to the first iteration of the procedure given in figure 6. We know that $d_{i_1} = b_1$ and for all $j < i_1$ we have $d_j > b_1$. The i_1^{th} step of the recurrence procedure (1) sets the b_1^{th} bit of the discriminator D_{i_1-1} to 1 and erases all 1-bits whose position is greater than b_1, to obtain D_{i_1}. Therefore, D_{i_1} can have only one 1-bit since none of the previous steps could set any bit at position less or equal b_1. Thus,

$$D_{i_1} = 2^{(M-b_1)} < K.$$

Let $k = n-1 < s-1$ and let

$$D_{i_{n-1}} = \sum_{i=1}^{n-1} 2^{(M-b_i)}.$$

Then, for $k = n$ we have that $d_{i_n} = b_n$ and for all entries j in the C_0-index, such that $i_{n-1} < j < i_n$ we have $d_j > b_n$. Hence, recurrence procedure (1) will erase all the changes on the discriminator recorded between steps i_{n-1} and i_n and new 1-bit will be set at position b_n. Thus, for $k = n$, the discriminator at the entry i_n becomes

$$D_{i_n} = D_{i_{n-1}} + 2^{(M-b_n)} = \sum_{i=1}^{n} 2^{(M-b_i)},$$

since $b_n > b_i$ for all $i < n$. Obviously, D_{i_n} is less than the search key K since $n < s \leq l$, where s and l are as above. □

Theorem 4.2: The search procedure of figure 6 stops at the first entry i_s in the C_0-index whose corresponding discriminator D_{i_s} is greater than the search key K.

Proof: By lemma 4.1 after the $(s-1)^{st}$ iteration of the search procedure, the encountered entry i_{s-1} has discriminator $D_{i_{s-1}}$:

$$D_{i_{s-1}} = \begin{cases} 0 & \text{if } s = 1 \\ \sum_{k=1}^{s-1} 2^{(M-b_k)} & \text{if } s > 1 \end{cases}$$

In both cases $D_{i_{s-1}} < K$. After the s^th^ iteration we know that $d_{i_s} < b_s$ and that every entry k, such that $i_{s-1} < k < i_s$, has depth $d_k > b_s$. Therefore, none of the discriminators $D_{i_{s-1}+1}$ through D_{i_s-1} can be greater than K, since even if all bit positions greater than b_s are set to 1 we have that

$$K \geq D_{i_{s-1}} + 2^{(M-b_s)} > D_{i_{s-1}} + \sum_{k=b_s+1}^{M} 2^{(M-k)}.$$

However, at entry i_s the value d_{i_s} is at most $b_s - 1$ and hence:

$$D_{i_s} \geq D_{i_{s-1}} + 2^{(M-b_s+1)} > D_{i_{s-1}} + \sum_{k=b_s}^{M} 2^{(M-k)} \geq K.$$

With the observation that the entry i_s, such that $d_{i_s} < b_s$, must be present in a C_0-index (recall that the last entry has depth 0 which is less than any bit position b_i), we conclude the proof. □

It should be apparent why M+1 was appended to the sequence $< b_k >$. Let the search key K be 01010000. So that, $B = < 2, 4 >$. Suppose that some entry i has discriminator D_i = 01010000. Entry $i+1$ has corresponding discriminator greater than K. But, the search procedure will be confused since both $b_1 = 2$ and $b_2 = 4$ have already been recognized and there is no b_3. Appending the value $M + 1$ (= 8 + 1 = 9) to the sequence resolves the problem by creating a search key K which is effectively 01010000.1_2.

The search procedure reduces to comparisons of small integers, no matter what the key length or type is. Thus, the key comparison time for any access is significantly reduced.

5. Operations on C_0-trees

The preceding section described a retrieval algorithm for C_0-trees without indicating how they might be grown. In this section, we show how the operations of item insertion, item deletion, index block splitting, and index block merging take place. Throughout, we will enforce a concept of consistency, where a C_0-tree structure will be called **consistent** if it faithfully represents a corresponding 0-complete binary tree.

Conceptually, data items are stored in the leaves of a 0-complete tree and the retrieval process involves following a path of labeled edges to the desired data item. In practice, actual retrieval, based on the depths of bounding nodes in the conceptual tree, is quite different. Similarly, item insertion and deletion can be viewed as the addition, or deletion, of leaves to a 0-complete binary tree in such a manner that the path to the leaf is a prefix of the item's key—while at the same time preserving the 0-complete property. Again, the actual procedures defined on a compact C_0-tree representation are quite different. But at all times we require the results of any such insertion or deletion operation to be consistent. The idea of consistency then is a kind of mirror, reflecting the compact C_0-representation onto its conceptual 0-complete model, which we use as a criterion of correctness and for an explanation of the procedures. In this section we will use the corresponding conceptual 0-complete tree only to explain the behavior of the processes. Proofs of consistency can be found in [OrP88].

In the following let R_i denote a data item, or record, with key K_i. Let K be any search key, and let $bit_K(b)$ denote the b^th^ bit in the key K. Let $e_i = (d_i, p_i)$ denote the i^th^ entry in an index block, in which d_i denotes the depth of a bounding node in a conceptual 0-complete tree and p_i is a pointer to either a leaf (data item, R_i, in storage) or an index block (0-complete subtree). Then $e_{i-1} = (d_{i-1}, p_{i-1})$ and $e_{i+1} = (d_{i+1}, p_{i+1})$ will denote respectively the immediate predecessor and successor of e_i in the C_0-index, and hence predecessor and successor (in the pre-order traversal) of the bounding node in the conceptual 0-complete tree.

5.1. Insertion

A record R with key K has been stored in location p. An entry, (d, p), is to be inserted in the C_0-index; R is to become a leaf L of the 0-complete tree. Using the key K a search is performed to locate a leaf L_i, such that K belongs to its key interval. Let $e_i = (d_i, p_i)$ be the last index entry encountered in this search, with p_i pointing to L_i.

We will see below that it is possible to create dummy leaves whose index pointer p is NIL. If e_i is such a dummy entry ($p_i = NIL$), then set p_i to p. The insertion is completed.

Otherwise, p_i in e_i points to an actual leaf (or record) L_i with key K_i. The two leaves L and L_i both belong in the same key interval. To determine their correct placement relative to each other we need to know the depth l_i of L_i in the conceptual 0-complete tree. This is not recorded in the compact C_0-representation. Only the depths of bounding nodes are recorded. But the depths of leaves can be determined by the following rule:

(a) If e_i is the first entry in an index block then L_i is a 0-leaf and $l_i = d_i$.
(b) If $d_i > d_{i-1}$ then L_i is a 0-leaf and $l_i = d_i$.
(c) If $d_i < d_{i-1}$ then L_i is a 1-leaf and $l_i = d_{i-1}$.

Let l_i denote the conceptual depth of L_i. Now, compare the two keys, K and K_i, to determine the first bit position, b', at which they differ (i.e. where $bit_K(b') = 0$ and $bit_{K_i}(b') = 1$, or vice versa). If $b' < l_i$, then L will always follow L_i in the pre-order sequence. Change e_i to (d_i, p) and insert (b', p_i) in the C_0-tree immediately before e_i.

If $l_i < b'$ then both L and L_i must be moved deeper in the conceptual tree in order to preserve a distinction between their access paths. To ensure that the conceptual tree is still 0-complete, it may be necessary to add dummy 0-leaves. For every 1-bit position b_j, if any, in K such that $l_i < b_j < b'$ inject a dummy entry (b_j, NIL) immediately before e_i. These dummy entries are inserted in the increasing order of their b_j values. Now, if $K < K_i$ then L should precede L_i in the pre-order traversal, insert the entry (b', p) immediately before e_i. If $K > K_i$, insert (b', p_i) before e_i and change e_i to (d_i, p).

Since the index blocks of a C_0-tree are of fixed size, an index block can eventually overfill. They are split in a manner similar to that of B-trees, except that they are not necessarily split exactly in half. A minimal partition of the index is generated by finding that entry (other than the last) in the block which has minimal depth, $d_{\min}$. Let d_{last} denote the depth of the last entry in the block. The index block, I, is split immediately *following* the minimal entry into two blocks I_0 and I_1.

If the split index block was the root block in the C_0-index, then a new root block must be created with just two entries. The first entry will be $(d_{\min}, I_0)$, the second entry will be (d_{last}, I_1). If the split index block I was not the root of the C_0-hierarchy, then it was referenced by an entry (d_{last}, I). This entry is changed to (d_{last}, I_1) and an entry $(d_{\min}, I_0)$ is inserted immediately before it. This may, of course, force an additional index splitting. Notice, this way of splitting enforces a stronger consistency interpretation in which every index block corresponds to a different 0-complete tree. However, special care must be taken when I_1 will have only one entry, since it invalidates the rule (a) for determining the true depth of the conceptual leaf. Keeping a trailing variable, while descending the tree from the root down, will resolve the problem.

Let us illustrate these three insertion steps with a running example in which we will assume at most 5 entries per index block. Begin with the conceptual 0-complete tree shown in figure 3 and its corresponding C_0-representation in figure 5. An item with key 01000110 is to be inserted. The resulting conceptual 0-complete tree will be that shown in figure 7.

Application of the search algorithm establishes that 01000110 falls in the key interval of the second entry, $e_2 = (1, L_{001})$. In this case, $b' = 2 < 3 = l_2$. Hence the single entry e_2 is replaced by the sequence $(2, L_{001})$ $(1, L_{01})$. The block will have to be split after the entry which has minimal depth, $d_{\min} = 1$. Following splitting, we get the hierarchical structure shown in figure 8.

At this point the reader should verify that the search algorithm given in figure 6 still works on a hierarchical C_0-structure. Let the search key K be 10101010, so that B = <1, 3, 5, 7, 9>. Begin searching the root index with *first* = 1. It will exit with k = 2 and j = 2. Follow pointer p_2 to its index block and continue scanning with *first* = 2.

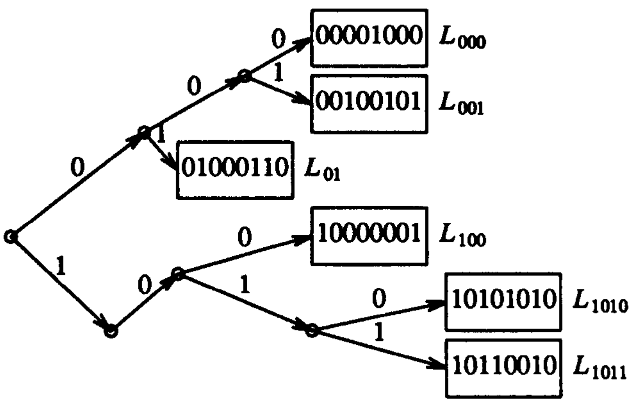


Figure 7.
The 0-complete tree of figure 3 after inserting 01000110

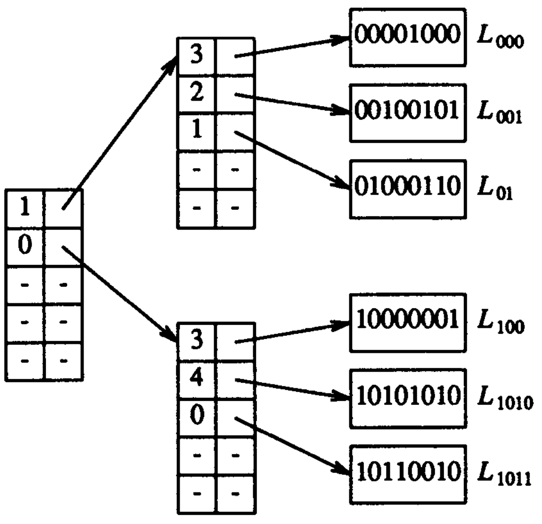


Figure 8.
The hierarchical C_0-representation of figure 7.

Entry of a new item with key, K = 10101100, will lead to a different growth pattern. After entry, the conceptual tree must look like figure 9. Notice that there is an empty leaf (indicated by the dashed lines) corresponding to L_{10100}. The long path 10101 must be present in this tree

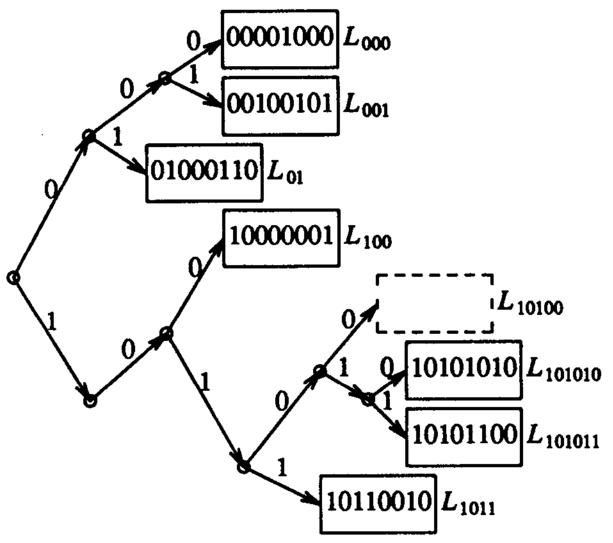


Figure 9.
The 0-complete tree after entry of 10101100

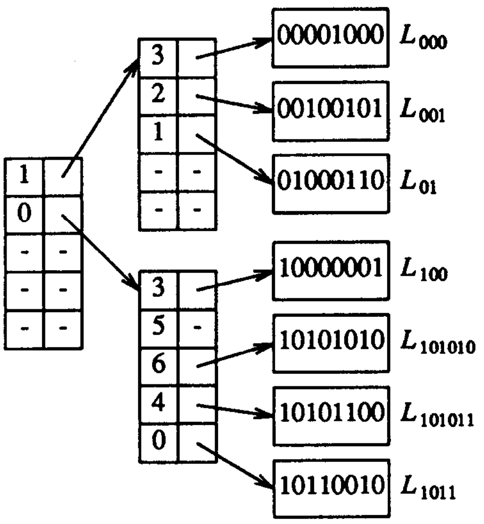


Figure 10.
The C_0-representation after entry of 10101100.

to differentiate between the keys 10101010 and 10101100. So if there were no leaf L_{10100}, the tree would not be 0-complete.

The search algorithm determines that 10101100 belongs in the key interval of the second entry $e_2 = (4, L_{1010})$ of the second index block. Since $4 > 3 = d_1$, $l_2 = 4$ and $b' = 6 > 4$. Since $l_2 = 4 < b_3 = 5 < b' = 6$, a dummy entry $(5, NIL)$ is entered. Then since $K > K_2$, e_2 is changed to $(4, L_{101011})$ and $(6, L_{101010})$ is inserted immediately before it to obtain figure 10.

5.2. Deletion

Assume that an item with key K has been deleted from storage. Its reference must be deleted from the index. The deletion procedure begins with the search for the index entry e_i corresponding to the given key. The easiest and fastest way to perform deletion in the C_0-tree is to set its pointer p_i in e_i to NIL. This will be sufficient in an environment where deletions are rare or at least less frequent than insertions.

We present in figure 11 a full deletion algorithm whose effect is precisely the opposite of insertion. With it, for example, the removal of the record with $K = 10101100$ from the tree in the figure 10 would yield the structure of figure 8 once again. In this case, $d_{i-1} = 6 > 4 = d_i$. Consequently, d_{i-1} becomes 4 and the entry pointing to the deleted record is removed. Subsequently, the preceding dummy entry, whose depth is greater than 4, is also removed from the index block.

If many blocks are underfilled it may be possible to increase storage utilization by merging two partially filled nodes in a manner that is nearly the inverse of node splitting. The complete algorithm for merging is given in

```
if d[i-1] > d[i]
  then begin
       d[i-1] ← d[i];
       Remove entry e[i];
       Remove all dummy entries j
       immediately preceding e[i-1]
       such that new d[i-1] < d[j];
       end
  else if d[i] > d[i+1]
         then begin
              Remove entry e[i];
              Remove all dummy entries
              j immediately preceding
              the new entry e[i] such
              that d[i] < d[j];
              end
         else set p[i] to NIL;
```

Figure 11. Deletion Algorithm

figure 12. Its pattern is similar to that of deletion of individual index entries. Notice, the procedure will be invoked after deleting one or more entries from an index page I_i which leaves the page with a number of entries below some threshold. Then the higher level entry e_i pointing to I_i, is consulted.

Merging can contract the root page to the point when it contains just one entry (with pointer p_1). In that case, the root page is deallocated and the new root becomes the page whose address is p_1. For example, this will happen if we delete the record with key 01000110 from the structure in figure 8. The entry e_3, pointing to the deleted record, will be removed which results in an underfilled index page. Consulting the root page we can see that the second merging condition is satisfied and the two lower level index blocks are combined together. An appropriate entry from the root page is deleted and the root is left with just one entry. This will force deallocation of the root page yielding figure 5.

```
flag ← false;
if d[i-1] > d[i]
   then if entries of I[i] and I[i-1]
           can be combined together
           then begin
              Replace all entries
              from I[i] into I[i-1];
              Release page I[i];
              d[i-1] ← d[i];
              Remove entry e[i] from
              the index block I[j];
              flag ← true
              end;
if d[i] > d[i+1] and flag = false
   then begin
        if entries of I[i] and I[i+1]
           can be combined together
           then begin
              Replace all entries
              from I[i+1] into I[i];
              Release page I[i+1];
              d[i] ← d[i+1];
              Remove entry e[i+1] from
              the index block I[j]
              end
        end
   else no merging is possible
          or it has already been done;
```

Figure 12. Merging Algorithm.

6. Results

Key compression obtained by replacing bounding search values with small, fixed size surrogates yields significant efficiencies in terms of storage overhead, total blocks accessed, and software simplification. The larger branching factors obtained ensure that the depths of C_0-trees will almost never exceed three levels. The storage overhead of C_0-tree indices can be expected to be 50%-80% less than the overhead of equivalent B-trees. Software simplification is equally important. For all keys, of any length or type, the layout of an index block and the search procedure are fixed. Precisely the same steps are performed regardless of whether the keys are character strings, dates, integers, reals or some combination of these.

The expected storage utilization of C_0-trees is $\ln 2 = 0.693$, as in the case of B-trees [Yao78]. However, due to the presence of dummy entries, the number of index entries L at the lowest level of the tree exceeds the actual number of records N. For random keys, this value asymptotically tends to

$$L = \mathrm{N}\cdot(log_2 e + 1)/2 \approx 1.221\cdot N.$$

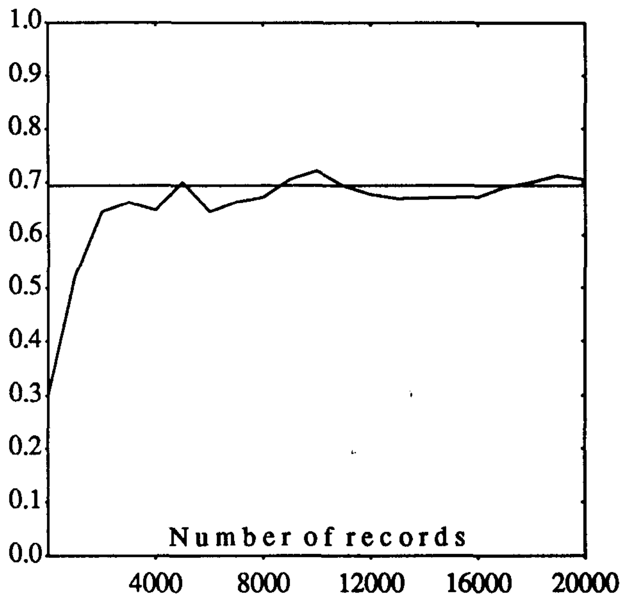
The number of dummy entries is

$$D = \mathrm{N}\cdot(log e - 1)/2 \approx 0.221\cdot\mathrm{N},$$

so that $(log e - 1)/(log e+1) \approx 0.181$ of entries at the lowest tree level will be nil, just to ensure consistency with an underlying 0-complete tree. Subsequently, counting only "useful" entries, the effective storage utilization at this level of a C_0-tree drops to about 0.567. Upper levels have no nil entries. Measurements in many large experimental databases confirm all of these "expected" values.

With these figures we can continue our comparative example from section 3. With parameters as before, a 3-level B-tree secondary index can be expected to support up to 295,885 records, while a 3 level C_0-tree index would normally support 6,570,240 records. Thus, despite a drop in effective utilization, the number of items accessible with the same retrieval cost (number of disk accesses per exact match query) is dramatically increased.

We have suspected that the storage utilization could deteriorate when non-uniformly distributed keys are maintained. In several experiments we have simulated skewed distributions of integer keys, as well as numbers coded as character strings, based on the Central Limit Theorem. In these experiments 20,000 keys have a tendency of grouping on specific subranges of the key space. The obtained results show relative insensitivity of C_0-trees to the distribution. After 20,000 insertions storage utilization was about 0.69 in all experiments. However, load factor was low (about 0.3) at the very beginning of these experiments when a small number of records were present (see figure 13). As the structure grew, the storage utilization showed a steady growth with small variations observed, and quickly attained its theoretical behavior. Surprisingly, in all experiments with skewed distribution the number of dummy



Parameters:
Maximal number of entries per index block - 124
Block size - 512 bytes

Figure 13.
Storage Utilization with Non-uniform Distribution

entries decreased when compared to the simulations with perfectly random keys. Recall that nil entries are less likely to appear when keys contain more 0's than 1's. Our non-uniform distributions were accidently biased towards such keys. Random keys tend to have approximately the same number of 0's and 1's.

A simple heuristic can increase storage utilization, although it will not prevent underfilling altogether. Without altering the algorithms given in this paper, an index block can be split immediately after the entry e_i, provided that depths of all preceding entries in the block are greater than the depth value of e_i. If we chose the e_i closest to the middle of the block we can obtain much closer approximation of even splitting of blocks in a tree. Another simple heuristic passes all keys through a filter that eliminates "unnecessary" 1-bits, as in the higher order bits of ASCII codes. This will minimize dummy entries.

References

[BaM72] R. Bayer and E. McCreight, Organization and Maintenance of Large Ordered Indexes, *Acta Informatica*, , 1972, 173-189.

[BaU77] R. Bayer and K. Unterauer, Prefix B-trees, *Trans. Database Systems 2*, 1 (Mar. 1977), 11-26.

[Com79] D. Comer, The Ubiquitous B-Tree, *Computing Surveys 11* , 2 (June 1979), 121-137.

[dTv87] W. de Jonge, A. S. Tanenbaum and R. P. van de Riet, Two Access Methods Using Compact Binary Trees, *IEEE Trans. on Software Eng. SE-13*, 7 (July 1987), 799-809.

[Fae79] R. Fagin and et.al., Extendible Hashing---A Fast Access Method for Dynamic Files , *Trans. Database Systems 4*, 3 (Sep. 1979), 315-344.

[Fre60] E. Fredkin, Many-way Information Retrieval, *Comm. of the ACM 3*, (1960), 490-500.

[Lar80] P. Larson, Linear Hashing with Partial Expansions, *Proc. 6th Conf. on VLDB*, Montreal, Canada, Oct. 1980, 224-232.

[Lit80] W. Litwin, Linear Virtual Hashing: A New Tool For Files and Tables Implementation, *Proc. 6th Conf. on VLDB*, Montreal, Canada, Oct. 1980, 212-223.

[Lit81] W. Litwin, Trie Hashing, *Proc. ACM-SIGMOD Conf. on Management of Data*, Ann Arbor, USA, 1981, 19-29.

[LiL87] W. Litwin and D. B. Lomet, A New Method for Fast Data Searches with Keys, *IEEE Software*, , Mar. 1987, 16-24.

[Lom81] D. B. Lomet, Digital B-trees, *Proc. 7th Conf. on VLDB*, Cannes, France, Oct. 1981, 333-344.

[OrP88] R. Orlandic and J. L. Pfaltz, Compact 0-Complete Trees: A New Method for Searching Large Files, IPC Tech. Rep.-88-001, Institute for Parallel Computation, Univ. of Virginia, Jan. 1988.

[Yao78] A. C. Yao, Random 3-2 Trees, *Acta Informatica*, , 1978, 159-170.