COMPACT O-COMPLETE TREES: A NEW METHOD FOR SEARCHING LARGE FILES

Ratko Orlandic
John L. Pfaltz

IPC-TR-88-001
January 26, 1988

Institute for Parallel Computation
School of Engineering and Applied Science
University of Virginia
Charlottesville, VA 22903

This research was supported in part by JPL under contract #957721.

COMPACT 0-COMPLETE TREES:
A NEW METHOD FOR SEARCHING LARGE FILES

Ratko Orlandic
John L. Pfaltz

Department of Computer Science
University of Virginia
Charlottesville, VA 22903

Abstract:

In this report, 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 1 byte surrogate. The replacement of actual key sequences by the 1 byte surrogate ensures a maximal possible fan out and greately 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. Then implementation details and empirical performance results are presented.

Reduction of index size by 50%-75% opens up the possibility of replicating system-wide indices for parallel access in distributed databases.

This work was supported in part by JPL Contract #957721

1. Introduction and Background

There are four generally accepted ways of directly accessing large files of records. They are (1) B-tree, (2) extensible hashing, (3) binary tree, and (4) trie retrieval. If the data file need not be stored in any kind of sort order, e.g. for range searching or sequential processing, and if only exact match retrieval is to be supported then extensible hashing [Fae79] or any of several variants [LiL87, Lom83, RLT83] are appropriate. The great appeal of extensible hashing methods is that normally no more than 2 disk reads are required to access any record of the file. But hashing methods will not support range searching, and is of limited value for secondary key retrieval.

An n-ary tree retrieval structure with sufficiently large fan out will generate broad search trees of minimal depth. The unique ability to ensure a minimal bound on the fan out in B-tree search guarantees retrieval performance of no more than 4 disk accesses for all but the largest files. Indeed, B-trees have so many desirable properties [Com79, ScO82]. that they are commonly regarded as the file access method of choice *whenever range searching is desired.* The major drawback of B-tree access is that the retrieval keys themselves must be stored in the index blocks. This leads to high storage overhead and makes both the access software and the degree of fan out dependent on key length.

Binary tree access is commonly ignored in practice. Access paths of length greater than 14 in files exceeding 30K records is intolerable. However, one can attain large fan out n-ary trees in "tries", which label the edges with search key substrings [Bur76, Fla81]. Finding an effective set of edge labels (it must be a reasonably large disjoint set of key subsegments that appear in most keys) that is appropriate in a variety of applications is difficult. Consequently, "trie" retrieval tends to be reserved for special purpose applications.

The access method described in this paper actually draws on elements of all four of these access methods. Its conceptual underpinnings come from that of binary "tries". These are represented by a compact index structure which is very similar to extensible hashing algorithms;

that is the data blocks are split and recombined in a similar fashion. But the keys are not hashed. Items may be accessed in sort order of their keys using a variant of the B-tree search algorithm. Consequently, range searching is supported. But unlike B-tree search, at most one byte will be used to represent the entire key value in the index blocks, regardless of key length.

The presentation of our access method is some what unusual. It begins, in this section, by describing access using a special 0-complete paginated binary "trie". In section 2, we will carefully define search algorithms and item insertion algorithms (complete with block splitting). But we will never really use a 0-complete paginated binary "trie" as the access structure itself. Access in a 2-way "trie" is far inferior to even regular binary tree search. These search trees are only conceptual, as are the attendant search and tree maintenance procedures. What we will then show, in section 3, is how to very efficiently emulate the access method defined for 0-complete paginated binary "trie"s with a compact B-tree like index structure that, for files of < 10M records, will bound exact match search with a maximum of 3 disk accesses.

Section 4 addresses the issue of item deletion, while in section 5 we present empirical storage utilization results for large files with uniformly distributed key values and for non-uniformly distributed key values.

1.1. Basic Terminology

A fundamental concept in this paper is that of a key. For our purposes a **key** will simply denote a string of binary digits, 0's and 1's, with arbitrary but fixed length *m*. We assume that the highest order key bit is at the first position from left. Keys may be lexicographically ordered so that we can compare any two of them for the properties of being greater, equal or less than. A **key prefix** of length *l* consists of the *l* leading bits of the key. If a prefix of two keys K_i and K_j is the same, we say that K_i and K_j have a **common prefix**.

Keys are always integral parts of records, and can be used to differentiate between them. For the purposes of this paper we assume key uniqueness, but it is not required by the method

itself. In practice, keys may be integers, reals, or characters, etc. In order to satisfy our definition we only expect that the bit string representation preserve the original ordering of the key domain (as is usually the case). For the search method we assume that records, along with their keys, are kept in fixed size units called **data pages,** or **blocks,** whose capacity *B* (blocking factor) is the maximal number of records each of them can contain. These data pages are the basic units of transfer between memory and secondary storage.

A tree is a **full binary tree** if every node either is a leaf or has exactly two nonempty descendants. Every arc in a binary tree can be labeled with either a 0 or a 1, making it a *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. Every node can be uniquely identified by its **path** which we define recursively. Let *u* be any node and *w* its parent. Then

$$path(u) = \begin{cases} \varepsilon & \text{if } u \textit{ is a root} \\ path(w)0 & \text{if } u \textit{ is } 0\textit{–node} \\ path(w)1 & \text{if } u \textit{ is } 1\textit{–node} \end{cases}$$

where ε denotes an empty string. The length of $path(u)$ is the **depth** of the node *u*.

The **discriminator,** D_u, of a node *u* is a binary string of fixed length *m* (maximal key length) whose high order *l* bits are $path(u)$ 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 $path(u)$ of length *l*. Consequently, if the key length is 8 bits, a node *u* with $path(u)$ = 0101, has discriminator 01010000. Its integer equivalent is 80_{10}. A node *y* with $path(y)$ = 0010 has discriminator 00100000 = 32_{10}. A binary node discriminator,*D*, can be used to denote both a binary string and its integer equivalent. In this paper we use juxtaposition, e.g. *xy*, to denote string concatenation and use the · or / operators to denote integer multiplication/division.

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

Proposition 1.1: Every full binary tree is a 0-complete binary tree.

Proof: Let T be a full binary tree with N leaves and (N-1) interior nodes. Every node (including 0-leaves) has its sibling, and since T is full, N/2 of the leaves are 1 nodes and (N-2)/2 = N/2 - 1 of the internal nodes are 1 nodes. (The root is neither.) Hence there are N/2 + N/2 - 1 = N-1 1-nodes. □

For the reader's convenience we give an informal meaning of the definition. Begin with any full binary tree (as in figure 1-a). By proposition 1.1, it is 0-complete. Arbitrarily delete 1-leaves whose 0-siblings are not leaves, e.g. the 1-leaves E and G in figure 1(a) to obtain figure 1(b). It is still a 0-complete tree. Note that deletion of any 0-leaf from a full tree will violate invariant (b) that requires N-l 1-nodes with N leaves.

The nodes of any tree can be totally ordered by a traversal of the tree. Since we are restricting our attention to 0-complete binary trees, we can give a slightly different definition of the familiar pre-order tree traversal. The **pre-order traversal** of a 0-complete tree starts at the root of the tree and iterates the following two steps until the last node has been accessed:

(a) if the current node u_i is an internal node then the next node u_{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 u_i is a leaf then the next node in the pre-order will be the 1-son of the node w whose 0-subtree contains u_i and whose depth is maximal.

One can easily verify that this sequence just described is the usual "pre-order traversal" by establishing that (1) every node is accessed, and (2) if node p precedes node q on any path from the root, then p is accessed before q. The pre-order traversal of figure 1-b is the sequence

---

(a) (b)

Figure 1.
0-complete binary trees.

---

a b d H I c f J k L M

where the leaf symbols have been capitalized for emphasis. Later, we will see that their order in the traversal is of special significance.

It is important to see how paths and node discriminators change during the pre-order traversal of a 0-complete tree. From our definition of the pre-order traversal it follows that

Proposition 1.2:

(a) **If a node u_i is an internal node with $path(u_i) = x$ of length l and discriminator $D_i = x \cdot 2^{(m-l)}$, then the path to the next node, u_{i+1}, in the pre-order is x0 and its discriminator is $D_{i+1} = 2 \cdot x \cdot 2^{m-(l+1)} = x \cdot 2^{(m-l)}$.**

(b) **Let u_i be a leaf and let u_{i+1} be its successor in the pre-order traversal. Let w be the parent of u_{i+1} with $path(w) = y$ of length $l-1$. Then $path(u_i)$ is either $yz0$ or $yz1$, where $length(z) > 0$ if u_i is 1-leaf or $length(z) = 0$ if u_i is 0-leaf, and $path(u_{i+1})$ is $y1$. Thus, the discriminator of u_{i+1} is $(2 \cdot y + 1) \cdot 2^{(m-l)}$.**

Proof: Follows virtually as a corollary of the definitions of pre-order traversal in a 0-complete tree and discriminator. □

The sequence of nodes in the traversal of figure 1-b, together with their corresponding discriminators of length 8 are shown in figure 2. Let L_{i-1} and L_i be two consecutive leaves encountered in the pre-order traversal, such as leaves I and J in the figure 2. Then L_{i-1} is **predecessor** of L_i and L_i is **successor** of L_{i-1}.

1.2. Paginated Binary Trees

A binary tree is said to be a **paginated binary tree** (**PB-tree**) if only leaves are data pages. Figure 3 illustrates a paginated binary tree with five leaves. Notice that, in general, a PB-tree need not be a full binary tree, nor a 0-complete tree. Figure 3 is neither.

Almost all computer representations of the PB-trees are based on pointers that explicitly convey the hierarchical relationship. Thus, each internal node of such a binary tree consists of two pointers. One of them points to its 0-subtree and the other one to the 1-subtree of the node. Either of them, but not both, can be set to NIL denoting an empty subtree. Access to a leaf in a paginated binary tree is based on the bitwise comparison of the given key, navigating through the

a	00000000
b	00000000
d	00000000
H	00000000
I	00010000
c	10000000
f	10000000
J	10000000
k	10100000
L	10100000
M	10110000

Figure 2.
Discriminators of nodes in Figure 1-b
in pre-order sequence

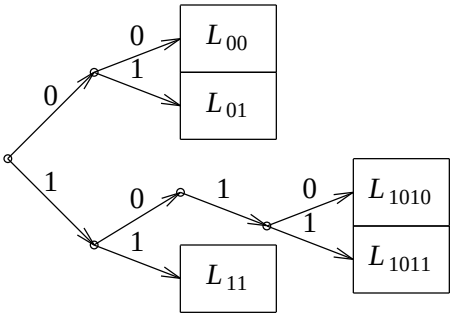


Figure 3.
Example of a paginated binary tree.

tree from the root node down. If at a traversed node the corresponding bit of the key is 0 then follow the zero pointer; if not, then follow the one pointer. The process is repeated until either a leaf page or a NIL pointer is encountered. In the later case (NIL pointer) the search is unsuccessful. If a leaf is found then the sequential examination of the keys stored in the leaf will reveal whether the desired record is actually present or not. This is a standard *trie* search, in which the search is controlled by the successive edge labels [Fre60, Knu73].

The key observation is that every leaf can be labeled by its path from the root. The traversed arc labels correspond to the prefix of all keys stored in the leaf. Thus, in the figure 3 the leaf L_{1010} contains all keys whose prefix is 1010, for example the keys 10100100 and 10101100. This common prefix of all keys stored in a leaf P of a PB-tree we call **leaf prefix** (**page prefix**) of P.

1.3. 0-complete Paginated Binary Trees

Since we are using paginated binary trees as a retrieval structure, our interest is in 0-complete PB-trees. As noted before, the PB-tree of the figure 3 is not 0-complete; it lacks a 0-leaf. To assure 0-completion, we must allow the addition of "empty" leaves, or pages, as needed. For example, to make the tree of fig. 3 0-complete we would have to add an empty leaf L_{100}, even though no item with key prefix 100 has been entered. We will say that such an empty 0-leaf **conceptually exists.** We actually allocate a data page only for leaves that are not empty. However, when growing a 0-complete PB-tree we will have to denote all of the empty 0-leaves in their appropriate positions in the tree. This will not be the case with empty 1-leaves.

From the implementation point of view there will be nontrivial difference between empty 0-leaves and empty 1-leaves. Entry of an item with a key which should be stored in an empty 0-leaf will force the insertion algorithm to allocate space for its page and link it at its place in the structure. The 0-leaf will "actually exist" now. Unfortunately, use of an entire page to store a single item is wasteful. With the empty 1-leaves we can be a bit more cavalier. The insertion algorithm will employ the following rule: items with keys whose prefix is the same as the path to an empty 1-leaf will be stored in the immediately preceding leaf page according to the pre-order traversal. In figure 4, curved arcs indicate where the items of empty 1-leaves would be located. Notice that an item with key 0100 0110 has been entered into the leaf L_{001} in this figure.

The **bounding node** of a leaf L in a 0-complete PB-tree is its successor node in the pre-order traversal of the tree.

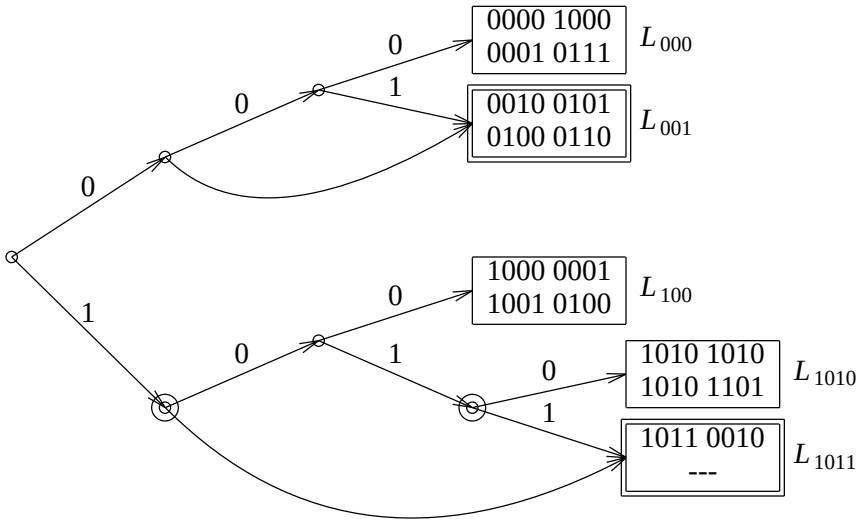


Figure 4.
A 0-complete PB tree with sample keys entered into the leaves

In the figure 4, bounding nodes are those which are circled. Since they are defined in terms of the pre-order traversal it follows that in a 0-complete tree each leaf, except the last one, has its own unique bounding node. Moreover, we have the following theorem:

Theorem 1.3: A node in a 0-complete tree is bounding if and only if it is a 1-node.

Proof: Every bounding node is a 1-node follows directly from part (b) in the definition of the pre-order traversal.

To see that every 1-node is bounding, observe that in a 0-complete tree there can be no 1-node which follows an interior node in the pre-order traversal (part (a) of the definition). Therefore, every 1-node must follow a leaf and thus is bounding. □

With the concept of bounding node we can now formally define what keys should be allocated to a particular leaf page. A leaf *L* in a 0-complete PB-tree contains precisely those keys which are greater or equal than the leaf's discriminator and less than the discriminator of its bounding node. Thus, the bounding node discriminator is the upper bound of the **key interval**

covering the leaf corresponding to that bounding node. This is precisely why we call them "bounding" nodes. 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.

2. Growth of a 0-complete Paginated Binary Tree

In this section we define the following primitive operations on 0-complete PB-trees:

search(K, t):	find item with key *K* in the tree *t*.
insert(K, i, t):	insert item *i* with key *K* in the tree *t*.
partition(t):	partition *t* into three 0-complete trees t_0, t_1 and t_2.

Deletions will be discussed latter. Our primary interest in this section will be to develop fundamental principles underlying the final structure.

2.1. Search (K, t)

Our search procedure is slightly different from the usual search in a PB-tree. As before, it uses the prefix of the given key to generate a search path through the tree until one of the following three conditions is satisfied:

(1) An existing leaf has been accessed, or
(2) An "empty" 0-leaf is encountered, or
(3) An "empty" 1-leaf is encountered.

In the first case of a leaf with data records, a sequential examination of each record in the leaf page will reveal whether any items actually satisfy the complete search key. In the second case, the search is immediately known to be unsuccessful. Finally, in the third case the search procedure examines the leaf page immediately preceding the empty 1-leaf in the pre-order traversal. With the following insertion algorithm some items that should have been stored in 1-leaves may be out of place!

2.2. Insertion (K, i, t)

The insertion algorithm starts with the search procedure to find the appropriate leaf page, *L*, for the given item with key K. Once the search for the leaf has been completed four cases can occur:

(1) the leaf is an empty 0-leaf;

(2) the leaf is an empty 1-leaf;
(3) the leaf is partially full; or
(4) the leaf is full.

The effect of the first two options have been generally described above. If the item belongs in an empty 0-leaf, a page must be allocated for L, attached to the tree, and the item is inserted. If the item belongs in an empty 1-leaf, it is inserted into the leaf page L' immediately preceding L in the ordering. L' must satisfy either of cases (3) or (4). In the third case the item is simply inserted into the leaf page L (or L').

In the fourth case the overfilled leaf must be split. There are two distinct ways of splitting which we call **growth in width** and **growth in height**. Before we explain these splitting options in more detail, notice that the structure may have become somewhat degenerate due to way that records which belong to empty 1-leaves are inserted. We can not assert, as in the case of PB-trees, that a common prefix of all keys stored in a leaf is exactly the path to the leaf. Thus, in the figure 4 the leaf L_{001} may contain keys with the prefix 01, such as 0100 0110. Those keys of a leaf L whose prefix of length l is not the same as the $path(L)$ are called **intruder keys**. Those keys whose prefix is the same as $path(L)$ are called **proper keys**. In a sort (ordering) of the keys of a leaf page, all proper keys must precede all intruder keys (if any). Either there exist items with intruder keys in the full leaf page, or not.

Notice that depth l of the leaf L is essential parameter used to recognize presence of intruder keys in L. We require all leaves, L, to have at least one proper key. Since all proper keys precede intruder keys, one can effectively identify the existence of intruder keys by just comparing the first key, K_1, and the last key, K_n in the leaf L. Let $bit_K(d)$ denote d^{th} bit of key K. Let d denote the first bit position from left where $bit_{K_1}(d) = 0$ and $bit_{K_n}(d) = 1$. If $d \le l$ then there exists at least one intruder key in L. If $d > l$ then all keys in L are proper keys.

If the full leaf page to be split, for example L_{001}, contains intruder keys, such as 0100 0110, it is natural to group all items with intruder keys together into a newly allocated leaf page L_{01}.

(Note that all intruder keys must have this common prefix.) Items with proper keys of L_{001} can remain in the original leaf page.

The tree has only grown **in width** since the path to the newly allocated leaf page must be shorter than that to the split leaf page. And readily, the 0-complete property has been preserved, since by adding a new 1-leaf to an 0-complete PB-tree we maintain the invariant of N leaves and N-1 1-nodes (see figure 5). But, we cannot assure an even splitting of leaves; there may be many, or few, items with intruder keys in the leaf, *L*.

Now suppose that the leaf page to be split has only proper keys, such as L_{1010} of the 0-complete PB-tree in figure 5. The insertion procedure first finds the longest common prefix of all keys in the leaf and then splits the page on the basis of the next bit. The common prefix will be the path to the parent of the two newly created leaves. For example, assume L_{1010} contains only proper keys whose common prefix is 10101. Two new leaves L_{101010} and L_{101011} would be

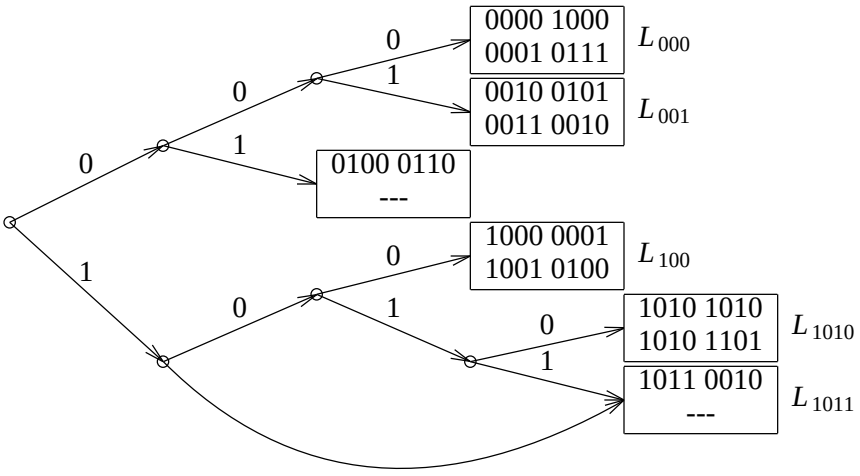


Figure 5.
Leaf L_{001} of figure 4 is split
so that the tree grows in width

created. The paths to the resultant pair of split nodes are always longer than the path to *L*. The tree has grown **in height.**

After splitting the tree would appear as in figure 6. However, the resulting tree is not 0-complete; there is a missing 0-leaf. To be 0-complete the empty 0-leaf must "conceptually exist". We can simply include it in the drawing of the tree as a NIL pointer, as has been done in figure 6. This can happen whenever the longest common prefix of all keys in the leaf to be split is longer than the path to the leaf. Formally, let *d* be the first bit position where some keys in a leaf *L* have 0 and some have 1. Let *l* be the depth of the leaf *L*. Since all keys in L are proper, $d > l$. Then, when splitting of the leaf occurs, for every 1-bit position *b* (if any) such that $l < b < d$ we include a conceptually existing 0-leaf in the appropriate position at depth *b*. At depth *d* two new leaves will be created in the 0-complete tree. Notice, any conceptually existing

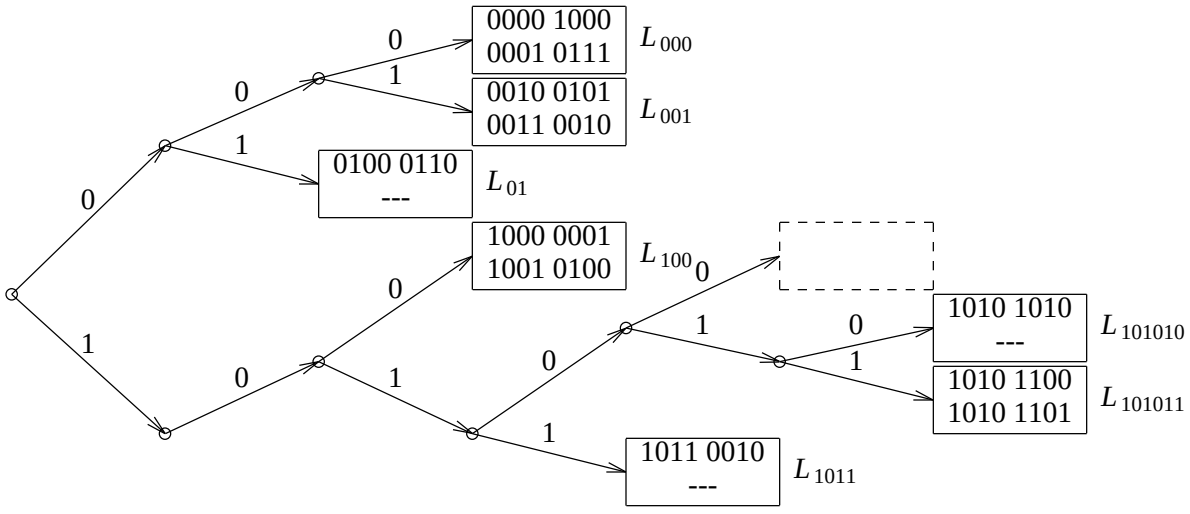


Figure 6.

Leaf L_{1010} of figure 5 is split
so that the tree grows in height.

0-leaf created by growth in height precedes the two newly created leaves in the pre-order traversal.

2.3. The Minimal Partition of 0-complete Trees

A **minimal bounding node,** is the bounding node, or 1-node, with the minimal depth.

Proposition 2.1: Given any 0-complete tree t with more than one leaf, there exists a unique minimal bounding node.

Proof: Suppose there exists a bounding node w, other than the minimal bounding node $u_{\min}$, with the same depth $d_{\min}$. By theorem 1.3, both w and $u_{\min}$ are 1-nodes with minimal depth $d_{\min}$. Since $u_{\min}$ and w are distinct nodes $path(u_{\min})$ and $path(w)$ are also distinct, so that there must exist a bit position where $path(u_{\min})$ has 0 and $path(w)$ has 1, or vice versa. This means that there must be a 1-node on the path from the root to either $u_{\min}$ or w with depth less than $d_{\min}$, which violates the minimality assumption. □

Proposition 2.2: In a 0-complete tree, there can be no 1-node on the path from the root to the minimal bounding node $u_{\min}$.

Proof: Follows directly from the proof of the preceding proposition. □

A partition of a 0-complete tree t into t_0, t_1 and t_2 is said to be the **minimal partition of** t if

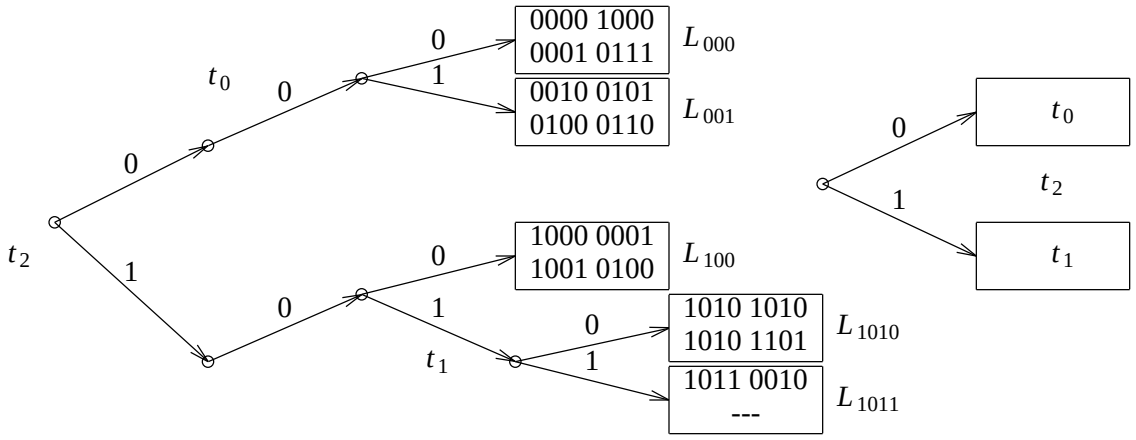
(a) t_1 is the subtree of t rooted at the minimal bounding node $u_{\min}$ of t;
(b) t_0 is the subtree of t rooted at the 0-sibling of $u_{\min}$.
(c) t_2 is the subtree produced from t by homomorphically contracting the subtrees t_0 and t_1 (see figure 7(b)).

Proposition 2.3: Trees t_0, t_1 and t_2 induced by the minimal partition of a 0-complete tree t are 0-compete trees.

Proof: From proposition 2.2 and the definition of the minimal partition it follows that t_2 has two leaves and just one 1-node (the leaf corresponding to t_1). Thus, it is 0-complete.

Let us suppose that either t_0 or t_1 is not 0-complete. By definition of 0-completeness it means that the tree has either a missing 0-leaf or a missing 1-leaf. This would, however, imply that the original tree t also has missing leaf, which contradicts our assumption. □

Note that the contracted tree t_2, together with t_0 and t_1, permit the unique reconstruction of t in the sense of [Pfa72, Pfa83].



(a) (b)

Figure 7.

Minimal partitions of 0-complete PB-trees.

2.4. Key Discrimination in 0-complete PB-trees

We turn our attention to the unambiguous identification of key intervals. This discussion will allow us to transform 0-complete trees into a much more compact representation.

In the figure 8(a) the key intervals of the 0-complete PB-tree from the figure 4 are listed in lexicographic order. We assume, as before, that the key length is 8 bits. Notice that the inclusive lower bound of a key interval is the upper bound of its preceding interval, which is, in turn, the bounding node discriminator of the preceding leaf. The fact that the discriminators of these 1-nodes (which by theorem 1.3 follow every leaf) bound the interval of keys represented in the leaf is precisely the reason they were given this name.

Given any key, to identify the interval it belongs to, it is sufficient to scan the bounding discriminators sequenced in the order they appear in the pre-order traversal of the tree. As soon as a discriminator greater than the given key is encountered it is known to be the upper bound so

```
00000000 - 00100000        00100000 - 3
00100000 - 10000000        10000000 - 1
10000000 - 10100000        10100000 - 3
10100000 - 10110000        10110000 - 4
10110000 - 11111111        11111111 - 0
      (a)                        (b)
```

Figure 8. The key intervals of leaves in a 0-complete tree.

that the key must belong to the leaf.

Figure 8(b) contains the bounding discriminators listed in order. Along with each item 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 depth 0. Henceforth we will assume that the last leaf in the order has a bounding node with depth 0.

Notice the regularity in the relationship between the discriminators and the depths of the set of bounding nodes. If the depth of a bounding node is d then the d^{th} bit of the corresponding discriminator is set to 1. We use this to develop a correspondence between the entire upper bound discriminator and the depth of its associated bounding node.

We begin with an informal description of the algorithm. 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:

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

where $\lfloor x \rfloor$ denotes the floor of x. We must still prove the assertion expressed by the recurrence relation. We begin with a lemma.

Lemma 2.4: If a leaf L_i has discriminator D_i and a bounding node u at depth d, then

$$D_u = (\lfloor D_i / 2^{(m-d)} \rfloor + 1) \cdot 2^{(m-d)},$$

where m is the key length.

Proof: Let l be the depth of L_i. By lemma 1.2(b) we have

$$path(L_i) = yz \quad \text{and} \quad path(u) = y1$$

$$D_i = (y \cdot 2^{(l-d+1)} + z) \cdot 2^{(m-l)} \tag{1}$$

$$D_u = (2 \cdot y + 1) \cdot 2^{(m-d)}. \tag{2}$$

Equation (1) uses the fact that z has length $d - l + 1$. We can write

$$2 \cdot y = \lfloor 2 \cdot (y \cdot 2^{(l-d+1)} + z) \cdot 2^{(d-l-1)} \rfloor$$

which can be transformed into

$$2 \cdot y = \lfloor (y \cdot 2^{(l-d+1)} + z) \cdot 2^{(m-l)} \cdot 2^{(d-m)} \rfloor = \lfloor D_i \cdot 2^{(d-m)} \rfloor \tag{3}$$

Substituting (3) in (2) we obtain the lemma. □

Theorem 2.5: The discriminators, D_i, of the bounding nodes of a 0-complete paginated binary tree satisfy the following recurrence relation:

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

where D_i denotes the discriminator of the i^{th} bounding node in the pre-order, d_i denotes the depth of the bounding node, m the key length and N number of leaves in the tree.

Proof: There are only three cases to consider.

Case 1. The node u_1 is the first bounding node in the pre-order traversal.

Because t is 0-complete, it cannot have a missing 0-leaf. Thus, the first leaf in the pre-

order will be the 0-leaf L_1 whose path x consists of all 0's. The discriminator D' of L_1 is just 0, which is equal to D_0. By lemma 2.4, the discriminator of the bounding node b of L_1 is

$$D_1 = (\lfloor D_0 / 2^{(m-d_1)} \rfloor + 1) \cdot 2^{(m-d_1)}.$$

Case 2. The bounding node u_{i-1} is an interior node.

Let $path(u_{i-1}) = x$. Then its discriminator is $D_{i-1} = x \cdot 2^{(m-d_{i-1})}$. The next leaf w in the pre-order traversal of the tree must be a 0-leaf whose depth d_i is greater than d_{i-1}. On the subpath from u_{i-1} to w there can be no 1-node, since by theorem 1.3 it would be a bounding node and by definition of bounding node another leaf would be interposed between u_{i-1} and w. This would violate the assumption that w is the next leaf encountered in the pre-order traversal after u_{i-1}. Thus, $path(w) = xy$ where y is a string containing $d_i - d_{i-1}$ zeros. Then the discriminator of w is:

$$D_w = x \cdot 2^{(d_i - d_{i-1})} \cdot 2^{(m-d_i)} = x \cdot 2^{(m-d_{i-1})} = D_{i-1}.$$

Since the bounding node of w is the next bounding node in the order after u_{i-1} we have:

$$D_i = (\lfloor D_{i-1} \cdot 2^{(d_i - m)} \rfloor + 1) \cdot 2^{(m-d_i)}.$$

Case 3. The bounding node u_{i-1} is a leaf.

This case straightforwardly proceeds from the lemma 2.4. □

Notice that the only input information for an algorithm which implements the recursive relation of theorem 2.5 is the set of depths of the bounding nodes in the given 0-complete tree. This can be restated as

Corollary 2.6: The ordered sequence of depths of all bounding nodes in a 0-complete tree provides sufficient information in order to access any leaf in the tree.

3. Compact Representation of 0-complete Trees

In multiway search techniques, of which B-tree search is the paradigm, each index block is implemented as k pointers and (k-1) key values ($k \leq M$, the maximum fan out of the tree). The core of the search algorithm is the loop

```
i ← 0
while i ≤ k do
      if search_key < value[i]
         then follow pointer[i]
         else i ← i+1
follow pointer[k]
```

Each key $value_i$ in the index block is a (non-inclusive) upper bound of all keys that can possibly be encountered by following $pointer_i$. In this section we transform the 0-complete binary *trie* into a multi-way structure in which we can use a search algorithm that is almost identical to B-tree search—except that we use a densely coded surrogate for each of the bounding $value_i$. From the surrogate the actual index key $value_i$, must be deducible. The surrogate that we use is the depth (in the conceptual 0-complete PB tree) of the bounding node with discriminator equal to $value_i$. Corollary 2.6 assures us that use of such a surrogate is possible.

In the new structure, which we call a **compact representation,** or C_0-tree, the depths of the *bounding nodes* of each leaf page (not the depth of the leaf page itself) are stored *in order* along with pointers to the leaf pages. These (depth, pointer) entries are kept in index blocks.

The depth field of an index entry denotes the length of a key prefix in bits. This depth certainly cannot exceed the length of the key field, which seldom exceeds 248 bits (= 31 bytes). Therefore, the depth can be represented by a single byte ($0 \leq \text{depth} \leq 2^8\text{-}1 = 255$). Thus, for rather large files (of up to 2^{24} = 16.6M pages) the total length of an index entry (depth and pointer) need not be longer than 4 bytes. A standard 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.

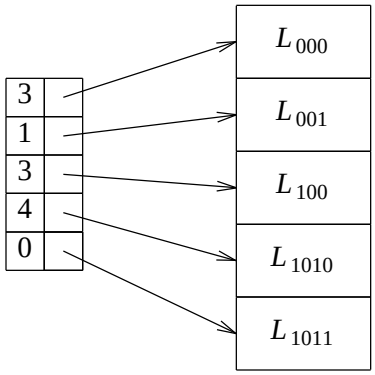


Figure 9.

Compact representation of Figure 4.

Key compression obtained by replacing bounding search values with the depth of the bounding node in a 0-complete tree yields significant efficiencies, both in terms of storage overhead and total blocks accessed. But equally important is the resulting software simplification. For all keys, for any key length, the layout of an index block and the search algorithm is fixed. They are identical, no matter if the keys are character strings, dates, integers, or real values. Common software is used for all access.

3.1. Access in C_0 - trees

The preceding paragraphs, together with the results of section 2, indicate that an effective search procedure using depth directories is feasible. Given a sequence of depths, theorem 2.5 demonstrates how to reconstruct the discriminator that will be used as the bounding value in the search algorithm. But the algorithm of theorem 2.5 is a "messy" one; it requires repeated divisions and shifting. The remainder of this section will be devoted to showing how the method of theorem 2.5 can be replaced with a very simple discriminator reconstruction algorithm.

We begin with five straightforward facts about 0-complete trees.

Proposition 3.1: If two bounding nodes u_i and u_j belong to the same path then the one with the shortest depth will appear first in the pre-order traversal.

Proof: Follows directly from the definition of the pre-order traversal. □

Proposition 3.2: If a bounding node u_i is at depth d_i then no bounding node u_j, appearing before u_i in the pre-order traversal with depth $d_j > d_i$, can belong to the same path with u_i.

Proof: Follows from proposition 3.1. □

Proposition 3.3: Let u_i and u_k, $i < k$ be two consecutive bounding nodes on some path x. Then all bounding nodes, u_j, appearing between them in the pre-order traversal must have the depth greater than the depths of either u_i or u_k (e.g. $d_j > d_i > d_k$).

Proof: Since on the path x there are no more bounding nodes between u_i and u_k the path from u_i to u_k must have the form 000...01. Consequently, all the bounding nodes that appear in the pre-order traversal after u_i and before u_k must be in the 0-subtree of the parent of u_k. The sibling w_k of u_k must be a 0-node and therefore, by theorem 1.3, it cannot be a bounding node. All the other nodes in the subtree have depths greater than the depth of w_k and hence, all possible bounding nodes in the subtree rooted at w_k must have depths greater than the depth of u_k. By the proposition 3.1, they also must be greater than the depth of u_i. □

Proposition 3.4: If a bounding node u_k is the first bounding node on a path x from the root down, then all bounding nodes u_j appearing before u_k in the pre-order traversal must have depths greater than the depth of u_1.

Proof: Similar to the proof of the proposition 3.3 with the root playing the role of u_i. □

Proposition 3.5: Positions of 1-bits in a discriminator of a bounding node u_i are the depths of all bounding nodes on the path to the bounding node u_i inclusively.

Proof: Follows directly from the definition of the node discriminator. □

Let L be the sequence of depths d_i of the bounding nodes in a 0-complete tree sorted by the pre-order traversal. In addition, let us have B = $[b_j \mid j \geq 1]$ denote the sequence of 1-bit positions of an arbitrary discriminator D. Then we can find the entry in the ordered set which corresponds to the bounding node u_n whose discriminator $D_n = D$. We give here an informal description of the algorithm.

Take the highest order 1-bit position b_1 of D_1 to guide the search. By proposition 3.4, starting from the first entry of the ordered set L and and skipping all entries *i*, if any, whose depth $d_i > b_1$, we will find an entry *j*, such that $d_j = b_1$. The first bounding node has been identified. By the proposition 3.1, the bounding node with the depth $d_k = b_2$ must appear after the j^{th} entry. So, we continue the procedure with b_2 determining the search. By the proposition 3.3, an entry *k* with $d_k = b_2$ will be found, after possibly skipping some more entries *i* with $d_i > b_2$. The search continues until every given 1-bit position has been recognized and stops on the entry corresponding to the bounding node u_n whose discriminator is $D_n = D$.

The preceding informal algorithm assumed *a priori* knowledge of the target discriminator D. This is unrealistic. But suppose a search key *K* has been specified. Recall that by the definition of a key interval covering a leaf page in a 0-complete tree, its upper bound is the discriminator of the bounding node of the leaf. Its inclusive lower bound is the discriminator of the leaf. If the bounding node has the depth d_i then by the proposition 1.2, both of the interval bounds have the same prefix of length $d_i - 1$ and they should differ in the d_i^{th} bit position. Namely, we have the following proposition.

Proposition 3.6: If a key *K* falls in the interval bounded above by the discriminator D_i of a bounding node with the depth d_i then

(a) both *K* and D_i have a common prefix of $d_i - 1$ bits length, and
(b) $bit_K(d_i)$ = 0 and $bit_D(d_i)$ = 1

where $bit_s(d_i)$ denotes the d_i^{th} bit of the string s.

An immediate consequence of the proposition is that the key *K* contains some $s \le d_i - 1$ higher order 1-bits in the same positions as the bounding node discriminator D_i and that the 1-bit position d_i is less than the next 1-bit position b_{s+1} in the key *K*. It is the positions of 1-bits in the key K and the depths d_i that are crucial. Moreover, we can base a search procedure on just these positions, b_s, of one bits in the key, K, and the depths, d_i, of bounding nodes. (To make sure that b_{s+1} exists in the sequence of 1-bit positions of a given key we append some large value to the sequence such as $m + 1$ where *m* is the key length.)

Needless to say, we know in advance neither the value of s nor d_i. Nevertheless, to identify the entry with the bounding node discriminator D_i, we actually do not need these values. Recall that the search procedure of the depth structure is looking for the first entry whose corresponding discriminator is greater than the given key. Proposition 3.2 tells us that as soon as we find an entry j with depth d_j, which is less than the depth b_k we are looking for, we know that the bounding nodes with depths d_j and b_k cannot belong to the same path and hence, we have exhausted all s bounding nodes defining the 1-bits that are common to the key K and the discriminator D_i. Since b_k turned out to be greater than d_j the key K must be less than the bounding node discriminator corresponding to the entry j. The whole search algorithm can be rewritten very compactly as in figure 10.

Notice, that we eventually must find the appropriate entry, since the last recorded depth is $0 < b_i$ for all i. Essentially, the whole search procedure reduces to the comparison of short integers, no matter what the key length is. This significantly reduces the computational time of

---

```
procedure search
given: A sequence B = b[j] of 1-bit positions in the search
       key appended with the value m+1.
       A sequence L = d[k] of depths of the bounding
       nodes in a 0-complete PB-tree.
begin
j ← 1
k ← 1
while b[j] ≤ d[k] do
       begin
       if b[j] = d[k]
          then j ← j+1
       k ← k+1
       end
return k
end
```

Figure 10.

Search algorithm.

---

the search procedure.

3.2. Growth of C_0 - trees

In the process of inserting new elements, the structure is allowed to grow gradually. The only requirement to be obeyed is that the structure should always retain the properties of 0-complete trees. Thus, at any instance of time we can find a 0-complete tree corresponding to the current state of the given C_0-tree. In other words, the structure should simulate the growth of the 0-complete tree. For any new bounding node appearing in the 0-complete tree we include a new entry in the index block in the proper position, so that the pre-order traversal is preserved.

Some bounding nodes in a 0-complete tree correspond to empty 0-leaves. No data page is allocated for these leaves, so that their bounding node entries in the index of the C_0-tree will have pointers set to NIL. Thus, these entries are dummies, but they still must be present in the structure in order to assure the 0-completeness property. An empty 1-leaf and its non-empty predecessor share the same bounding node and their proper keys are merged into one leaf. Thus, just one entry in the index is sufficient for an existing leaf and its empty successor leaves (if any) in the pre-order. Dummy entries are unnecessary in this case. Readily, this distinction between empty 0-leaves and empty 1-leaves allows us to reduce the number of dummy NIL pointers by approximately 50%.

In section 2 we used the parameter *l*, denoting the true depth of a leaf *L*, to identify intruder keys in *L*. However, this information is not directly available in a C_0-tree. It must be established in the process of accessing the leaf.

Lemma 3.7: Depth of any 0-leaf in a 0-complete tree is equal to the depth of its bounding node.

Proof: The bounding node of any 0-leaf is its sibling. □

Lemma 3.8: Depth of any 1-leaf in a 0-complete tree is equal to the depth of the bounding node of its predecessor in the pre-order traversal.

Proof: By theorem 1.3 any 1-leaf L_i is the bounding node of its predecessor leaf L_{i-1}. □

Lemma 3.9: Let u_{i-1} and u_i be two consecutive bounding nodes encountered in pre-order traversal of a 0-complete tree. Let d_{i-1} and d_i be their depths, respectively. Then

- **(a)** $d_i > d_{i-1}$ **if and only if u_i is the bounding node of a 0-leaf,**
- **(b)** $d_i < d_{i-1}$ **if and only if u_i is the bounding node of a 1-leaf.**

Proof: Suppose u_{i-1} and u_i belong to the same path. Then by the proof of the proposition 3.3, sibling of u_i is a 0-node w. If w were an interior node, then by 0-completeness it would have at least one descending 0-leaf whose sibling u_k would be its bounding node. By the definition of the pre-order traversal, u_k would appear after u_{i-1} and before u_i, violating the assumption that u_{i-1} and u_i are two consecutive bounding nodes encountered in the pre-order traversal. Thus, w has to be a 0-leaf and u_i is its bounding node. At the same time, since u_{i-1} and u_i belong to the same path, by proposition 3.1, $d_i > d_{i-1}$.

Suppose there is no path x such that both u_{i-1} and u_i belong to the path. In that case, u_{i-1} is a leaf for the same reason above the node w could not be an interior node. By theorem 1.3, u_{i-1} must be a 1-leaf. By proposition 1.2(b), the next bounding node u_i in the pre-order traversal is a 1-node whose path is shorter than the path to u_{i-1}. Therefore, u_i is a bounding node of a 1-leaf, and $d_i < d_{i-1}$. □

Theorem 3.10: Let I denote the index block of a C_0-tree. Let d_{i-1} and d_i be depth fields of two entries e_{i-1} and e_i in I, which point to leaf pages P_{i-1} and P_i corresponding to leaves L_{i-1} and L_i of the appropriate 0-complete tree. Let l denote the true depth of L_i.

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

Proof: Part (a) follows from the 0-completeness property and the lemma 3.7. Parts (b) and (c) follow directly from lemmas 3.7, 3.8 and 3.9. □

Theorem 3.10 provides a way to identify the true depth of a leaf L in a 0-complete tree without explicitly storing it in the C_0-tree. Thus the same procedures given in section 2 to grow a 0-complete PB tree can be used to split data pages in a C_0-tree.

Since an index block of a C_0-tree is of fixed size, the index itself can overfill. Splitting the index corresponds to partitioning the 0-complete tree t into three subtrees t_0, t_1 and t_2. To find the minimal partition it is sufficient to locate that entry in the index page which has minimal depth (not counting the last entry with artificial depth 0). The index page is split immediately

following the minimal entry into two pages I_0 and I_1. Thus I_0 corresponds to the t_0 subtree and I_1 to t_1. In addition, a new root block corresponding to the partition t_2 is allocated. Since t_2 has only two leaves and one among them is a bounding node whose depth is the minimal depth $d_{\min}$ in the old index block, we allocate two entries in the root page; the first one with depth $d_{\min}$, pointing to the L_0, and the second one with the depth 0, pointing to L_1. Splitting of index pages below the root requires just injection of a new entry in the upper level index page.

Let us illustrate the discussion with a more comprehensive example. We start with the C_0-tree of figure 9 repeated (with sample key entries) as figure 11. We assume maximal number of entries per index block to be 5, maximal number of entries per data block 2 and the key length 8. The corresponding 0-complete tree is shown in figure 4.

Insertion of the key 0011 0010, whose set of 1-bit positions (augmented with 8+1 = 9) is B = {3, 4, 7, 9}, starts with the search for the appropriate block. The first bit $b_1 = 3$ is matched by the first entry in the index block. However, trying to match $b_2 = 4$ we encounter the second entry whose depth $d_2 = 1$ is less than b_2, so that the pointer of the entry leads us to the data block L_{001}

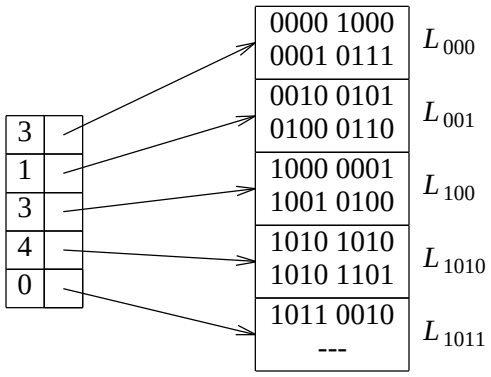


Figure 11.

A C_0-tree representing
the 0-complete tree of figure 4.

of the structure. Since the block is full it must split. Inclusion of a new entry into the index block forces it to split and the new root page is created. Figure 12 reflects the new state of the structure. It corresponds exactly to the 0-complete tree of figure 5 after the same leaf was split.

Now, insert the key 1010 1100, with B = {1, 3, 5, 6, 9}. We recognize $b_1 = 1$ at the root page. The second index entry points to a lower level index block. We match $b_2 = 3$ on the first entry of the index block, but trying to find $b_3 = 5$ we encounter the second entry in the block with $d_2 = 4 < b_3$, and the pointer of that entry leads us to the data block L_{1010} of the structure. The block splits and two new entries are inserted in the block: one is dummy entry, corresponding to the fifth 1-bit common to all keys in the leaf L_{101010} and the second one points to the newly created block. Figure 13, which corresponds to the 0-complete tree of figure 7, shows the structure after the insertion.

Finally, entry of the key 1000 1100 will force splitting of the data block L_{100} (in figure 13 and figure 6) and hence of the higher index block. A new item is included in the root page (see

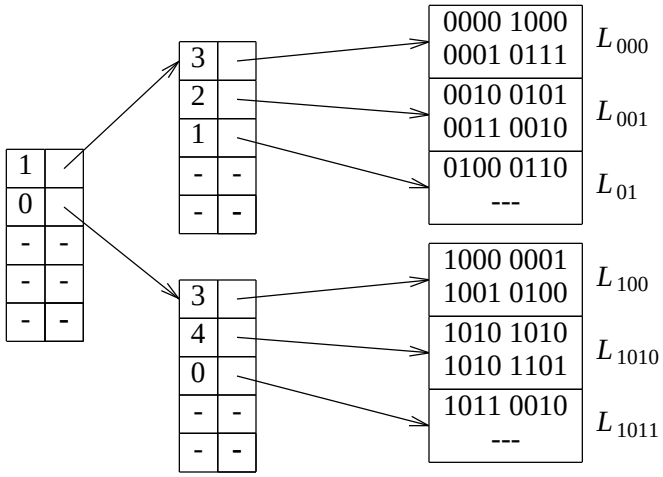


Figure 12.

The C_0-tree after insertion of key 0011 0010.

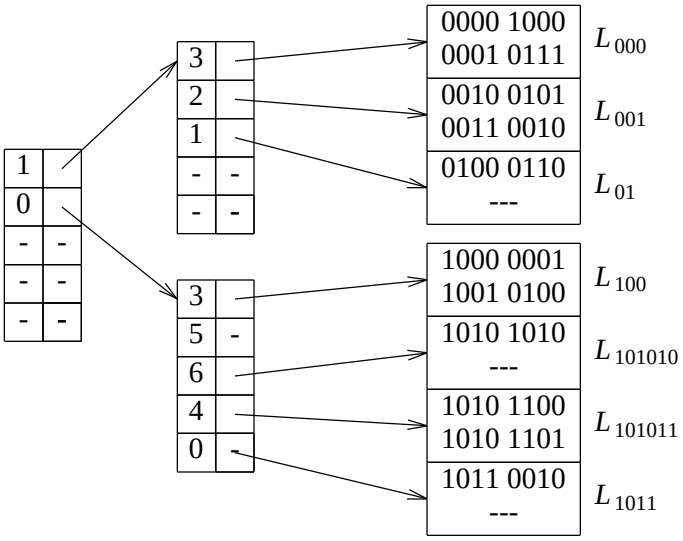


Figure 13.

The C_0-tree after insertion of 1010 1100.

figure 14).

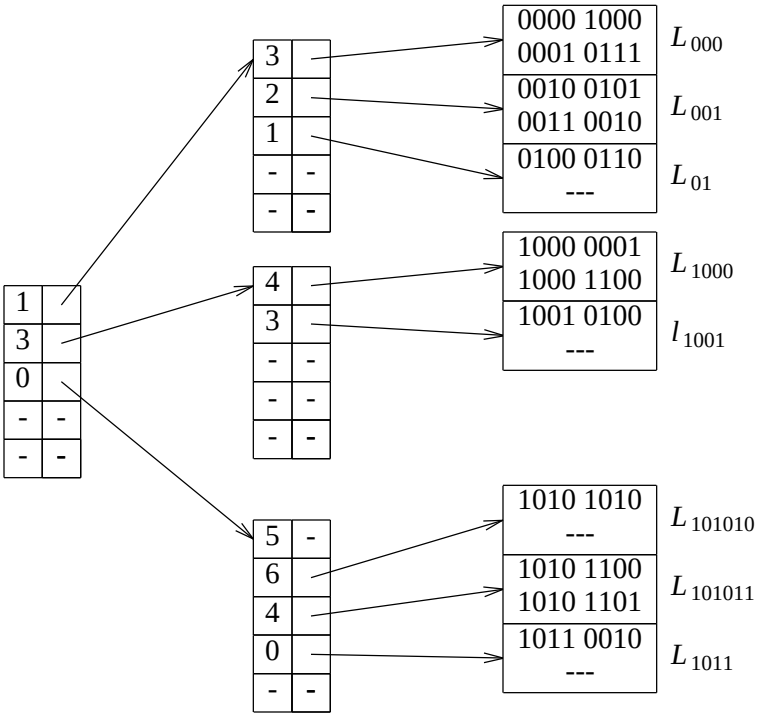


Figure 14.

The C_0-tree after insertion of 1000 1100.

4. Deletion

The deletion procedure begins with the search for the item with the given key. Once the item has been located it is, in general, simply removed from the leaf page. There are exceptions, however. Recall that, in section 2, insertion required that every leaf page have at least one proper key prior to its splitting. The reader can verify that this is essential by using the algorithm of section 2.2 to insert 0100 1100 into L_{001} of figure 15. In a 0-complete tree (or its compact representation) that only grows through splitting, every leaf must have at least one proper key. But if deletions are allowed this need no longer be true.

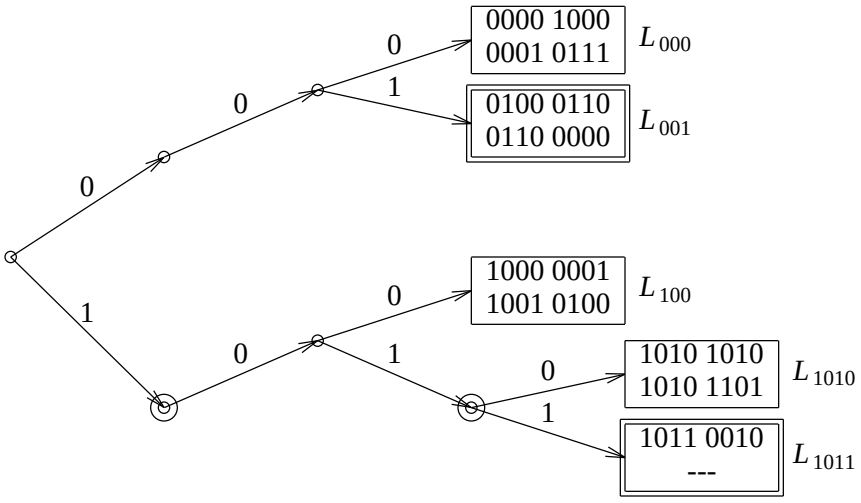


Figure 15.

A 0-complete tree whose second leaf consists of only intruder keys.

4.1. Local Rotations

Ideally, we would like to detect the situation when only intruder keys are present in a leaf page and than place the leaf in its proper position. In our example leaf L_{001} would become L_{01}. This would, however, leave the leaf L_{000} without its sibling, which would violate the definition of the 0-complete trees. The hanging 0-leaf L_{000} must be moved closer to the root in order to ensure that it has its sibling (see figure 16). We call this movement a **local rotation.**

The same operation performed in C_0-trees is simpler. Just one entry in an index block needs to be affected. Because all 0-leaves have siblings, only 1-leaves can have intruder keys. A rotation of a 1-leaf L_i must move it one or more levels closer to the root, linking it at the place of a previously "non-existing" 1-leaf. The bounding node of L_i will remain the same. On the other hand, L_i was also a bounding node for its predecessor L_{i-1}. Thus, moving L_{001} means that the bounding node of L_{i-1} has changed its depth. No other bounding node needs to be affected.

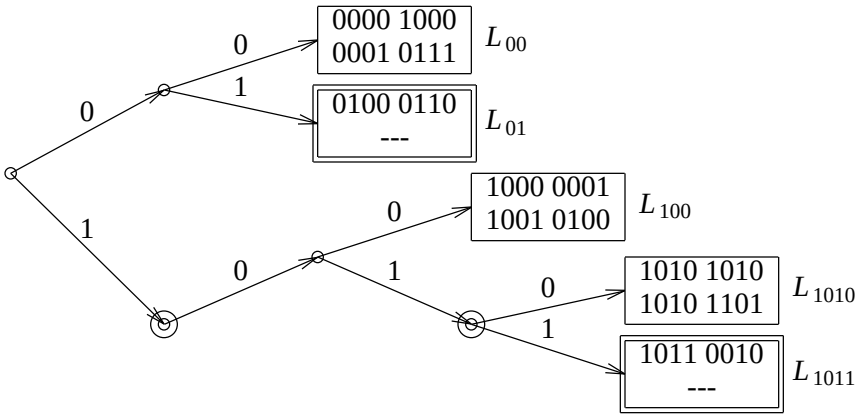


Figure 16.

The 0-complete tree after the local rotation.

Formally, let e_i be an entry in an index block I, pointing to a data page P_i which has to be rotated. Let e_{i-1} with the depth field d_{i-1} be immediate predecessor of e_i in the block I. (Notice that e_{i-1} must be in the same block I, since (1) only 1-leaves force the rotation, and (2) by Theorem 3.10 e_1 must denote a 0-leaf.) If the leaf L_i of a 0-complete tree, corresponding to the data page P_i in the C_0-tree, is rotated to the level l, d_{i-1} will be set to l. Nothing else needs to be changed. The ordered sequence of depths of all bounding nodes in the 0-complete tree of figure 15 is [3, 1, 3, 4, 0]. After the rotation it has become [2, 1, 3, 4, 0] corresponding to the depths of the bounding nodes in the 0-complete tree of figure 16.

One problem remains to be resolved. How can we identify that a page in a C_0-tree has only intruder keys. Observe two facts. First, a data page with such property can be created only by deleting all proper keys from the page. None of the other operations can create a page consisting of only intruder keys. Second, all intruder keys will reside after the proper keys, provided that sequential order within the page is maintained. Consider the situation immediately before a data page P has become occupied only by intruder keys. The page had exactly one proper key, resid-

ing in the first position, and all the subsequent ones were intruder keys. Deleting the proper key from the leaf will leave only intruder keys present. Therefore, it is appropriate to perform a check for remaining proper keys only when deleting the first record in the data block. In that case, if the second key in the page is the intruder key then all the subsequent one's are too. Let K_1 and K_2 be the first two keys in a data page P. Assume K_1 is to be deleted, and that the current true depth of the page P is l. Let b be the first bit position such that $bit_{K_1} = 0$ and $bit_{K_2} = 1$. If $b < l$ then the key K_1 is a singleton proper key, and after it is deleted local rotation is performed (i.e. d_{i-1}, defined above, is set to b). Only after deletion of a singleton proper key is local rotation ever needed.

4.2. Merging Procedure

If many leaves 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. Let us first analyze merging in 0-complete trees. To decide what leaves can be merged we have to keep in mind the following:

(a) Only two adjacent leaves can be merged; otherwise the result will not be a binary tree, and

(b) After merging two leaves the resulting tree must be 0-complete.

In operational access systems, whether B-tree or C_0-tree, one does not usually bother to merge underfilled blocks; but it is possible, as we indicate in considerable detail below.

Let L_i be a partially filled leaf eligible for merging. By (a), L_i can be merged either with its predecessor L_{i-1} or with its successor L_{i+1}. Let d_i, d_{i-1} and d_{i+1} be the depths of bounding nodes of the leaves L_i, L_{i-1} and L_{i+1}, respectively. Then at least one of the following situations will occur (the reader should pay special attention to the discussion on depths d_i, d_{i-1} and d_{i+1}).

[1] **Both L_i and L_{i-1} are 1-leaves.** By theorem 1.3, leaf L_i is also a bounding node of L_{i-1}. By lemma 3.8 depth l of L_i is equal to d_{i-1}. By lemma 3.9 $d_i < l$. Therefore, $d_i < d_{i-1}$.

[2] L_i **is 1-leaf and** L_{i-1} **a 0-leaf.** By the definition of 0-completeness, L_i and L_{i-1} are sibling leaves. L_i must be the bounding node of L_{i-1}. As before, by lemma 3.8 depth l of L_i is equal to d_{i-1} and by lemma 3.9 $d_i < l$. Thus, again $d_i < d_{i-1}$.

[3] **Both** L_i **and** L_{i+1} **are 1-leaves.** This case is analogous to case 1. Therefore, $d_{i+1} < d_i$.

[4] L_i **is 0-leaf and** L_{i+1} **is a 1-leaf.** This case is analogous to case 2. Hence, $d_{i+1} < d_i$.

[5] **Both** L_i **and** L_{i-1} **are 0-leaves.** By lemma 3.9 $d_i > d_{i-1}$.

[6] L_i **is 0-leaf and** L_{i-1} **is a 1-leaf.** Let u_k and u_i be two consecutive bounding nodes encountered in the pre-order traversal. Let u_i be the bounding node of L_i. Hence, u_k is the bounding node of L_{i-1}. Since L_i is a 0-leaf, by the proof of lemma 3.9 u_k and u_i must belong to a common path from the root down and by proposition 3.1 we have that d_k is less than d_i. Since $d_k = d_{i-1}$ we have that $d_i > d_{i-1}$.

[7] **Both** L_i **and** L_{i+1} **are 0-leaves.** This case is analogous to case 5. Therefore, $d_{i+1} > d_i$.

[8] L_i **is 1-leaf and** L_{i+1} **a 0-leaf.** This case is analogous to case 6. Hence, $d_{i+1} > d_i$.

Notice that situations 5-8 cannot lead to merging since there is no way to ensure that all records will reside in their proper intervals after the merging. Then, provided that careful merging has been conducted, only cases 1-4 need be considered. Observe that these cases can be easily identified in a C_0-tree by considering only d_i, d_{i-1} and d_{i+1}. Namely, two adjacent underfilled data pages P_i and P_{i-1} or P_i and P_{i+1}, can be merged only if one of the following conditions holds: $d_i < d_{i-1}$ or $d_i > d_{i+1}$, respectively. Otherwise, the configuration corresponds to one of cases 5-8 and merging is impossible. The complete algorithm for merging is given in figure 17. Notice, the algorithm assumes that it will be invoked after deleting a record from a data page P_i which leaves the page with a number of records below some threshold.

The check for a dummy entry corresponds to the case 2 when L_{i-1} is only a "conceptually existing" leaf. If such a situation occurs, no merging is performed, but the index node entry e_{i-1} is just removed. A leaf can become empty, despite the above rules for merging. In that case we

```
flag ← false
if d[i-1] > d[i]
    then begin
         if not ( p[i-1] = NIL )
             then begin
                  if rec's of P[i] and P[i-1] can be placed in one block
                    then begin
                      Replace all records from P[i] into P[i-1]
                      Release page P[i]
                      d[i-1] ← d[i]
                      Remove entry e[i] from index block
                      flag ← true
                      end
                 end
             else remove entry e[i-1] from index block
       end
if ( d[i] > d[i+1] ) and ( flag = false )
    then begin
         if records of P[i] and P[i+1] can be placed in one block
              then begin
                   Replace all records from P[i+1] into P[i]
                   Release page P[i+1]
                   d[i] ← d[i+1]
                   Remove entry e[i+1] from index block
                   end
         end
    else no merging is possible
```

Figure 17.
Merging Algorithm.

deallocate its page, but only in the case of an empty 0-leaf, satisfying none of the above conditions, must we retain a NIL pointer to the "conceptually existing" page.

Merging two index blocks into one follows the pattern of the algorithm in the figure 17. The only difference is that upper level index blocks (above those directly pointing to the data pages) do not have NIL pointers. Thus, the procedure is simplified by not having to check for existence of such pointers. If, as a consequence of merging index blocks, just one entry (with pointer p_1) remains in the root page we deallocate the page and the new root becomes the page whose address is p_1. It is not hard to see that continuous merging can contract the whole

structure into its initial state, corresponding to the case when the structure contains just one data block.

We must still show that the merging algorithm on C_0-trees preserves 0-completeness of the appropriate 0-complete tree *t*. Let P_i be the data page of a C_0-tree whose underfilling forces the merge procedure to be invoked. There are several cases to consider. First, let P_i be merged with the page P_{i-1}. Let L_i and L_{i-1} be the leaves of the appropriate 0-complete tree *t*, corresponding to P_i and P_{i-1}, respectively. Then, just before merging, as we have seen above, either both leaves L_i and L_{i-1} were 1-leaves **or** L_i was a 1-leaf and L_{i-1} was a 0-leaf.

(a) **Both L_i and L_{i-1} are 1-leaves.**

Merging will destroy the leaf L_i. Bounding node of L_i becomes the bounding node of L_{i-1}. Since L_i was both a leaf and a bounding node, after its destruction the invariant of 0-complete trees is preserved. Also, it is easy to see that the sibling of L_i, prior to merge, was an interior node. Otherwise, L_{i-1} would be the sibling of L_i, but it is not. Thus, by destruction of L_i no 0-leaf is left without its sibling which proves the second part of the definition of 0-completeness.

(b) **L_i is a 1-leaf and L_{i-1} is a 0-leaf.**

Merging again destroys L_i and since it was both 1-leaf and a bounding node the invariant of 0-completeness is satisfied. However, we have to show that L_{i-1} was not left hanging. Notice, after the destruction of L_i its bounding node with depth d_i has become the bounding node of L_{i-1}. By changing d_{i-1} to d_i we have implicitly moved L_{i-1} closer to the root. Theorem 3.10 can tell us what is the new depth of L_{i-1}. It can be either d_i or d_{i-2}, where d_{i-2} is the depth of the bounding node of the L_{i-1}'s predecessor L_{i-2}. If it is d_i then L_{i-1} has remained to be a 0-leaf, but now its sibling is its new bounding node. If it is d_{i-2} then L_{i-1} has become the sibling of L_{i-2}, thus a 1-leaf. In both cases, after the merging, L_{i-1} has its sibling. This proves the second part of the definition of 0-completeness.

(c) L_i **is a 1-leaf and** L_{i-1} **is a conceptually existing 0-leaf.**

Notice that after removing the entry e_{i-1} from an index page the resulting C_0-tree is the same as if we were dealing with the case when L_{i-1} is a fully existing leaf. Thus, pretending that all records from L_i were borrowed to L_{i-1} and then moving L_{i-1} to its new position in the 0-complete tree, we obtain analogous situation as in the case [b]. Therefore, the same proof applies here.

Now, consider the situation when pages P_i and P_{i+1} of a C_0-tree are merged. Then, either both L_i and L_{i+1} are 1-leaves **or** L_i is a 0-leaf and L_{i+1} is a 1-leaf. These cases are analogous to cases [a] and [b] in that now L_i now plays the role of L_{i-1} and L_{i+1} stands for L_i. Thus, proofs [a] and [b] apply here as well. Notice that in both cases L_{i+1} is a 1-leaf, so that the situation when L_{i+1} was a conceptually existing leaf cannot occur. Since there are no more cases to consider we conclude the proof that merge algorithm indeed preserves 0-completeness.

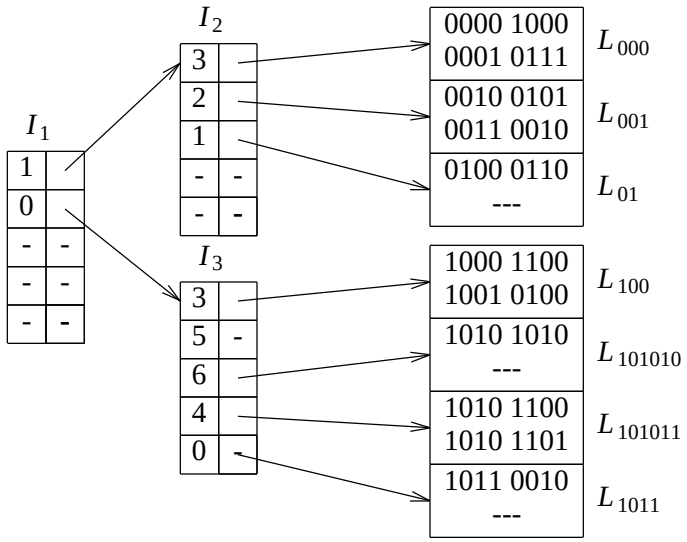


Figure 18.

C_0-tree of figure 14 after deletion of key 1000 0001.

Let us now continue our example from section 3. Suppose we delete key 1000 0001 from the block L_{1000} in figure 14. The block becomes candidate for merging since it is now half full. We examine index block I_3. The entry $e_{3,1}$ is the one pointing to the data block L_{1000}. We examine $e_{3,1}$ and $e_{3,2}$ and conclude that they can be merged since $d_{3,1} = 4 > d_{3,2} = 3$ and the data block pointed to by the index entry $e_{3,2}$ is also half full. So, data blocks L_{1000} and L_{1001} are collapsed and one index entry is removed from I_3. Now index block I_3 becomes eligible for merging, so we examine the root page. It appears that I_3 cannot be merged with I_2 since the entries $e_{1,1}$ and $e_{1,2}$ in the root page pointing to the index blocks contain the depth values such that $d_{1,1} = 1 < d_{1,2} = 3$. However, I_3 can be merged with I_4 since $d_{1,2} > d_{1,3}$. The resulting tree is shown in the figure 18.

Now, deleting the key 1010 1010 from the block L_{101010} will force blocks L_{101010} and L_{101011} to merge and subsequently to discard an entry from the block I_3. The resulting tree is shown in figure 19. Deletion of the key 1010 1100 from block L_{10101} in figure 19 will reveal that

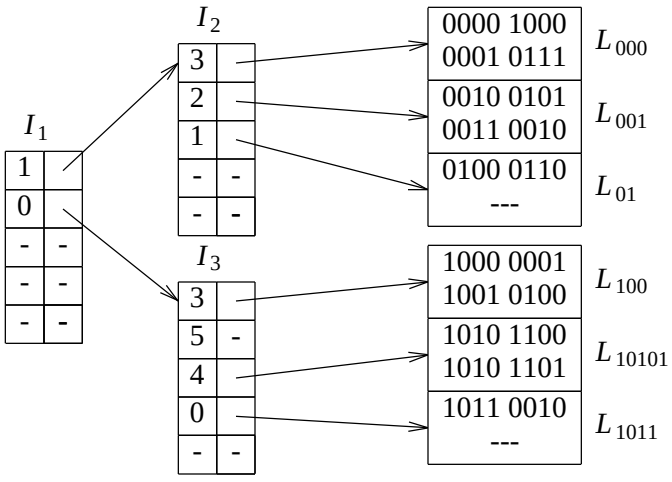


Figure 19.

The C_0-tree after deletion of key 1010 1010.

that we can simply remove the adjacent NIL entry from I_3. The structure becomes the one of the figure 20.

Finally, if we delete the key 0100 0110 from L_{01} the blocks L_{001} and L_{01} are merged. Removing one entry from the index block I_2 will force merging I_2 and I_3 in the figure 20. Subsequently, removing one entry from the root I_1 will leave the root with just one entry. In this case we deallocate the root page. The final structure is shown in figure 21.

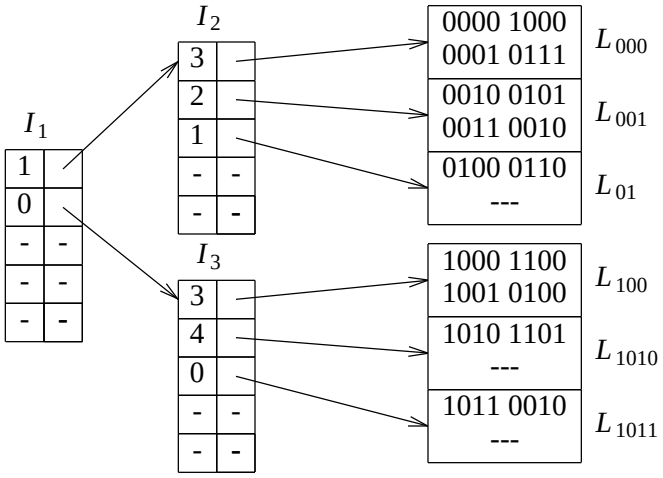


Figure 20.

The C_0-tree after deletion of key 1010 1100.

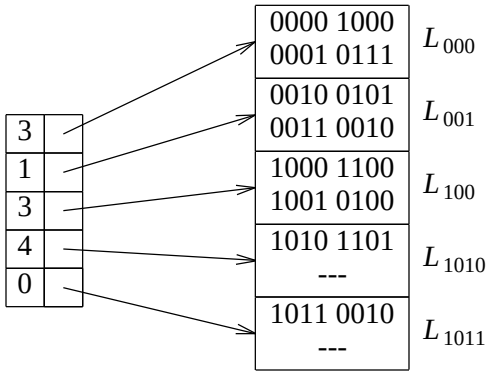


Figure 21.

The C_0-tree after deletion of key 0100 0101.

5. Analysis and Discussion

Using a derivation, similar to one in [Fae79], we can show that under a Poisson distribution the expected number of data blocks *B*, in the structure with N records and the blocking factor per data page *M*, tends asymptotically to

$$B(N,M) = N/(M \cdot \ln 2) .$$

The term $\ln 2 \approx 0.693$, can be regarded to as the average data page utilization. For the sake of a rough estimation of expected path length let us assume that the same utilization can be expected for index blocks. (This assumption, however, is not justifiable for the root index block.) Then we can compute the expected depth of a C_0-tree to be

$$D(N,M,k) = 1 + \log_{k \cdot \ln 2}\left\lceil \frac{N}{M \cdot \ln 2} \right\rceil ,$$

where *k* is maximal number of entries per index block. Accordingly, for a block size of 4K bytes, with 4 byte index entries (1 byte depth surrogate, 3 byte pointers), and $M = 100$, we would have $k \approx 1{,}000$. Using the expression above, we can expect to retrieve any one of 30,000,000 records

C_0-tree in no more than 3 disk accesses.

5.1. Experimental Results

The algorithms outlined in the previous sections have been written in C and run on a VAX-11/780 under 4.2BSD UNIX. The experiments were run for up to 20 000 records. The following structure parameters were chosen. The block size was chosen to be 512 bytes, blocking factor per data page was varied between 20 and 40, and the maximal number of entries per index page was set to 125. Keys were coded as unsigned integers and character strings.

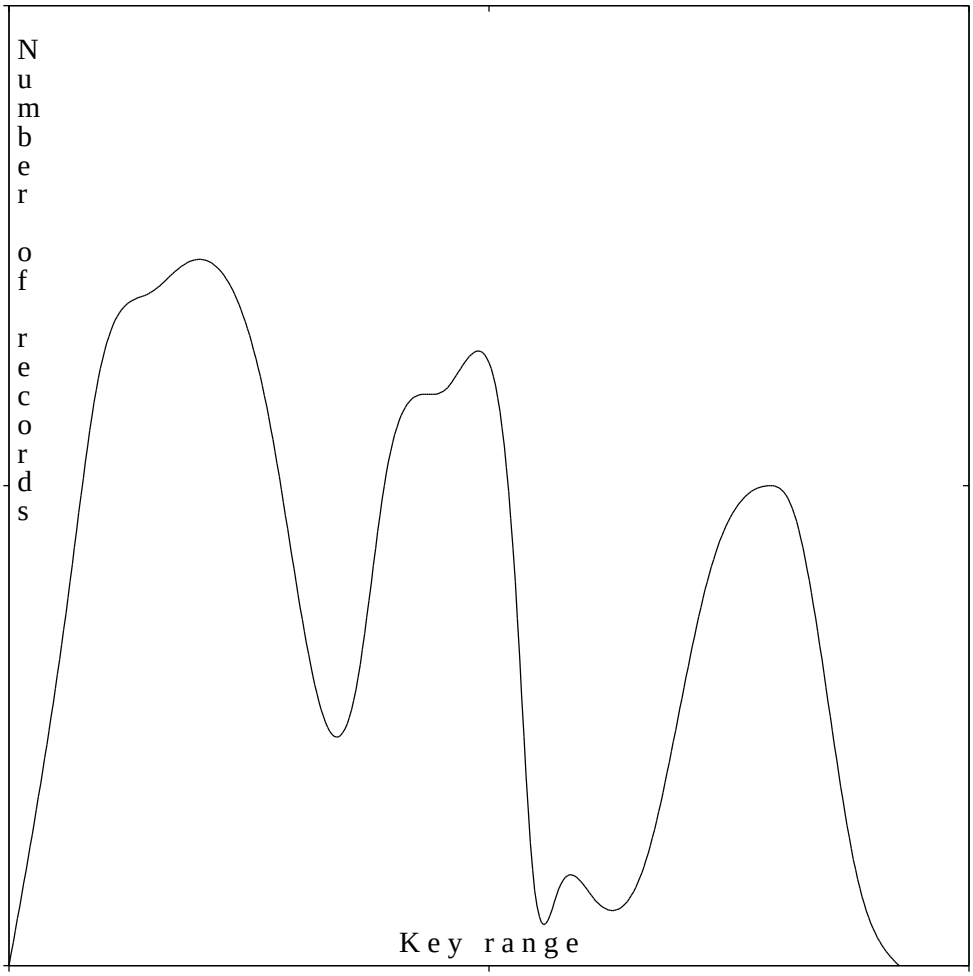
The measurements can be divided into two major groups. First, some experiments were conducted to measure performance under the uniform distribution of keys. Second, we have simulated nonuniform key distribution and measured the structure under these conditions. As indicated before, access and storage performance of the structure heavily depend upon the average load factor of data blocks, which was the major criterion of our interest.

The results show that under the uniform key distribution the structure behaves as theoretically predicted. For $M = 33$, after 20,000 insertions of integer keys the number of data blocks was 921 giving the average data block utilization of 0.66. Small variations of the value (0.65 - 0.73) were apparent throughout the experiment. The cumulative average storage utilization was 0.692. With the root page the number of index blocks were 9 at the end of the experiment. Thus, average index block utilization, not counting the root page was almost 92%. Dummy entries did not appear.

We have suspected that the performance could be significantly worse for nonuniform key distribution, since no insurance against block underfilling is provided in the structure. In several experiments we simulated a nonuniform distribution of keys based on the Central Limit Theorem. In these experiments integer keys have a tendency of grouping on specific subranges of the key space. We shall describe here the major experiment, which collects results and tendencies observed in the sequence of similar experiments. The same parameters as above were used for

the measurement. Key distribution was approximately as shown in figure 22.

At the end of this experiment, average storage utilization was 0.690 and cumulative average utilization had the value 0.678. The depth of the structure was 3, with 875 data blocks and 19 index blocks. No significant difference was observed at the end of the experiment with respect to strage utilization, compared with the uniform distribution. However, average load factor of the data blocks was very low, about 30%, at the beginning of the experiment when the number of entries was small. As the structure grew, the storage utilization increased with small variation

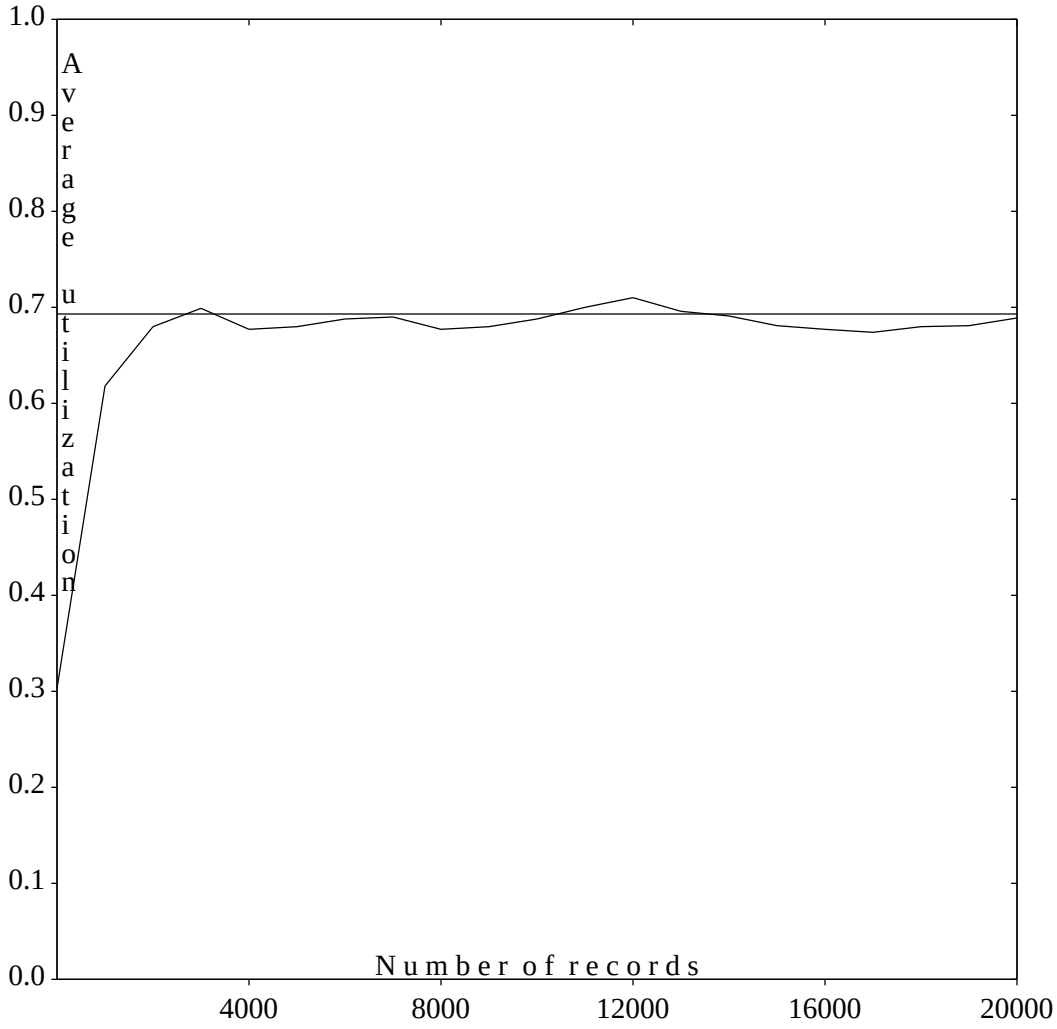


Key Distribution over the Key Range

Figure 22.

and quickly attained its theoretical behavior (see figure 23).

This result shows that large numbers have randomizing effect on the structure and that it gradually adjusts, exercising theoretically predicted behavior.



Parameters:
Maximal number of records per data block - 33
Maximal number of entries per index block - 125
Block size - 512 bytes

Average Utilization for Nonuniform Distribution

Figure 23.

A significant difference was observed in the utilization of index blocks, which was about 0.39 at the end of the experiment. Some blocks were utilized only 2%, while some others over 80%. In addition, dummy entries have appeared, although their number was insignificant. Our conjecture is that the storage overhead at each level of the index independently conforms to figure 23. Only an expected 950 level-1 index elements are needed with files of this size, and using figure 23 one would predict a storage utilization of 0.40 to 0.45.

Other experiments were conducted with random values coded as 9 or 12 byte ASCII character strings. There were no significant deviation with respect to the load factor of data pages. However, the number of dummy entries in the index blocks was significant (about 10-15%), due to the fact that four out of eight bits in an ASCII character (two of which are 1-bits) were not utilized. This imposes a demand for a special care to be paid when coding character strings. In general, the number of nonutilized 1-bits in the key values should be reduced or hopefully eliminated, since their immediate consquence is inclusion of dummy entries in the index blocks of the structure. Further, a good implementation need not allow index blocks containing just one entry, especially if it is a dummy. This requires a small modification of the algorithms outlined in the previous sections.

Yet another improvement can increase utilization of index blocks, although it will not prevent underfilling altogether. Without alterating the presented algorithms, an index block can be split immediately after the entry e_i, provided that depths of all preceding entries in the index 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 index blocks.

5.2. C_0-trees Used for Secondary Indexing

So far, we have discussed C_0-trees assuming their purpose for primary retrieval. It is equally important to see how the structure can be used for secondary indexing. Let us assume that a database file contains data records with several attribute fields, stored in the order of their

insertion. Let us call this structure **main file.** For every attribute field of the particular record type, we can build a secondary index pointing to the records in the main file. Obviously, it is important to keep the size of secondary indices as low as possible, since their cumulative space overhead can be significant.

A slight modification, which eliminates keys from data blocks of C_0-trees, makes the structure extremely suitable for this purpose. We here outline only the basic idea. Imagine a 0-complete tree whose every leaf is either conceptually existing or has exactly **one** record. Then, using corrolary 3.6 as the paradigm, we can access any record in a main file knowing only ordered sequence of depths of all bounding nodes in the 0-complete tree. Every bounding node of the tree stands for at most one record. We can, therefore, substitute every record in the data blocks of a C_0-tree with a pair (d,p), where d is the depth of its bounding node in the appropriate 0-complete tree and p is the pointer to the record stored in the main file. What we see here is a C_0-tree consisting of only densly coded surrogates, thus obtaining the same structure for both lower and upper level blocks in the tree. Maintainance algorithms for the structure are further simplified, since no separate provision for data blocks needs to be provided. We can also expect significant storage savings in comparison to other schemes used for secondary indexing.

Due to limited space, discussion on some specifics of the algorithms for building, accessing and mintaining the C_0-trees in the context of secondary retrieaval, as well as the provision for handling large number of equal keys is left out here. We just point out that handling these specifics presents no structural difficulty.

References

[Bur76] W. A. Burkhard, Hashing and Trie Algorithms for Partial Match Retrieval, *Trans. Database Systems 1*,2 (June 1976), 175-187.

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

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

[Fla81] P. Flajolet, On the Performance Evaluation of Extendible Hashing and Trie Searching, RJ3258, IBM, San Jose CA, Oct. 1981.

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

[Knu73] D. E. Knuth, *The Art of Computer Programming, Sorting and Searching*, Addison Wesley, Reading, MA, 1973. Vol. 3.

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

[Lom83] D. B. Lomet, Bounded Index Exponential Hashing, *Trans. Database Systems 8*,1 (Mar. 1983), 136-165.

[Pfa72] J. L. Pfaltz, Graph Structures, *J. ACM 19*,3 (July 1972), 411-422.

[Pfa83] J. L. Pfaltz, Transformations of Structures by Convex Homomorphisms, in *Lecture Notes in Computer Science, #153*, H. Ehrig, M. Nagl and G. Rozenberg (editors), Springer-Verlag, 1983, 297-313.

[RLT83] K. Ramamohanarao, J. W. Lloyd and J. A. Thom, Partial-match Retrieval using Hashing and Descriptors, *Trans. Database Systems 8*,4 (Dec. 1983).

[ScO82] P. Scheurermann and M. Ouksel, Multidimensional B-Trees for Asssociative Searching in Database Systems, *Inform. Systems 7*,2 (1982), 123-137.

Table of Contents

1. Introduction and Background 2
1.1. Basic Terminology 3
prefix 3
0-node, 1-node 4
depth 4
discriminator 4
0-complete 4
pre-order traversal 5
predecessor 6
successor 6
1.2. Paginated Binary Trees 6
leaf prefix 8
1.3. 0-complete Paginated Binary Trees 8
bounding node 8
key interval 10
2. Growth of a 0-complete Paginated Binary Tree 11
2.1. Search 11
2.2. Insertion 11
intruder keys 12
proper keys 12
growth in width 13
growth in height 14
2.3. The Minimal Partition of 0-complete Trees 15
minimal bounding node 15
2.4. Key Discrimination in 0-complete PB-trees 16
3. Compact Representation of 0-complete Trees 20
3.1. Access in C_0-trees 21
search algorithm 24
3.2. Growth of C_0-trees 25
4. Deletion 30
4.1. Local Rotations 31
4.2. Merging Procedure 33
5. Analysis and Discussion 40
5.1. Experimental Results 41
5.2. C_0-trees Used for Secondary Indexing 44