ABSTRACT: A B-tree is *compact* if it is minimal in number of nodes, hence has optimal space utilization, among equally capacious B-trees of the same order. The space utilization of compact B-trees is analyzed and is compared with that of noncompact B-trees and of (node)-visit-optimal B-trees, which minimize the expected number of nodes visited per key access. Compact B-trees can be as much as a *factor* of 2.5 more space-efficient than visit-optimal B-trees; and the node-visit cost of a compact tree is never more than 1 + the node-visit cost of an optimal tree. The utility of initializing a B-tree to be compact (which initialization can be done in time linear in the number of keys if the keys are pre-sorted) is demonstrated by comparing the space utilization of a compact tree that has been augmented by random insertions with that of a tree that has been grown entirely by random insertions. Even after increasing the number of keys by a modest amount, the effects of compact initialization are still felt. Once the tree has grown so large that these effects are no longer discernible, the tree can be expeditiously compacted in place using an algorithm presented here; and the benefits of compactness resume.

Time- and Space-Optimality in B-Trees*

Arnold L. Rosenberg†

Lawrence Snyder

Research Report #167, August 1979

*Material included in this paper was presented under the title of "Compact B-Trees" at the 1979 ACM-SIGMOD International Conference on Management of Data. The work was supported in part by National Science Foundation grant MCS78-04749.

†Mathematical Sciences Department, IBM T. J. Watson Research Center, Yorktown Heights, NY 10598.

1. *Introduction*

The B-tree data structure of Bayer and McCreight [1;3§6.2.4] is an effective method of organizing an *external* file when the operations of searching, insertion and deletion must be supported since these operations on B-trees require time at most logarithmic in the size of the file. This logarithmic performance is achievable [6] because the B-tree definition is underconstrained: one file has numerous legal B-tree representations. Specifically, a finite rooted tree must satisfy two conditions to be a *B-tree of order M*:

(1.1) each internal node must have at least $\lceil M \div 2 \rceil$ descendants (except the root which can have as few as two), and at most M descendants,

(1.2) all root-to-leaf paths must be of equal length.

To illustrate that these conditions do not constrain B-trees to a unique structure, let $\mathcal{T}_k^M$ denote the set of all order M B-trees with a capacity of k keys. Then $\mathcal{T}_{15}^3$ contains ten (structurally) distinct B-trees of order 3 [also called 2,3-trees] each capable of containing fifteen keys. Figure 1 exhibits these ten trees.

The first purpose of this paper is to characterize the optimally space efficient trees in the forest $\mathcal{T}_k^M$. Space will be measured by the total number of internal nodes. (The leaves may be ignored since all trees in $\mathcal{T}_k^M$ have k+1 leaves.) Our use of the number of internal nodes as a cost measure for externally stored B-trees is motivated by the usual policy of allocating for each node a fixed-size block of storage (page, track, etc.) capable of holding the maximum of M descendant pointers and M-1 keys. Should a node actually have fewer than the maximum

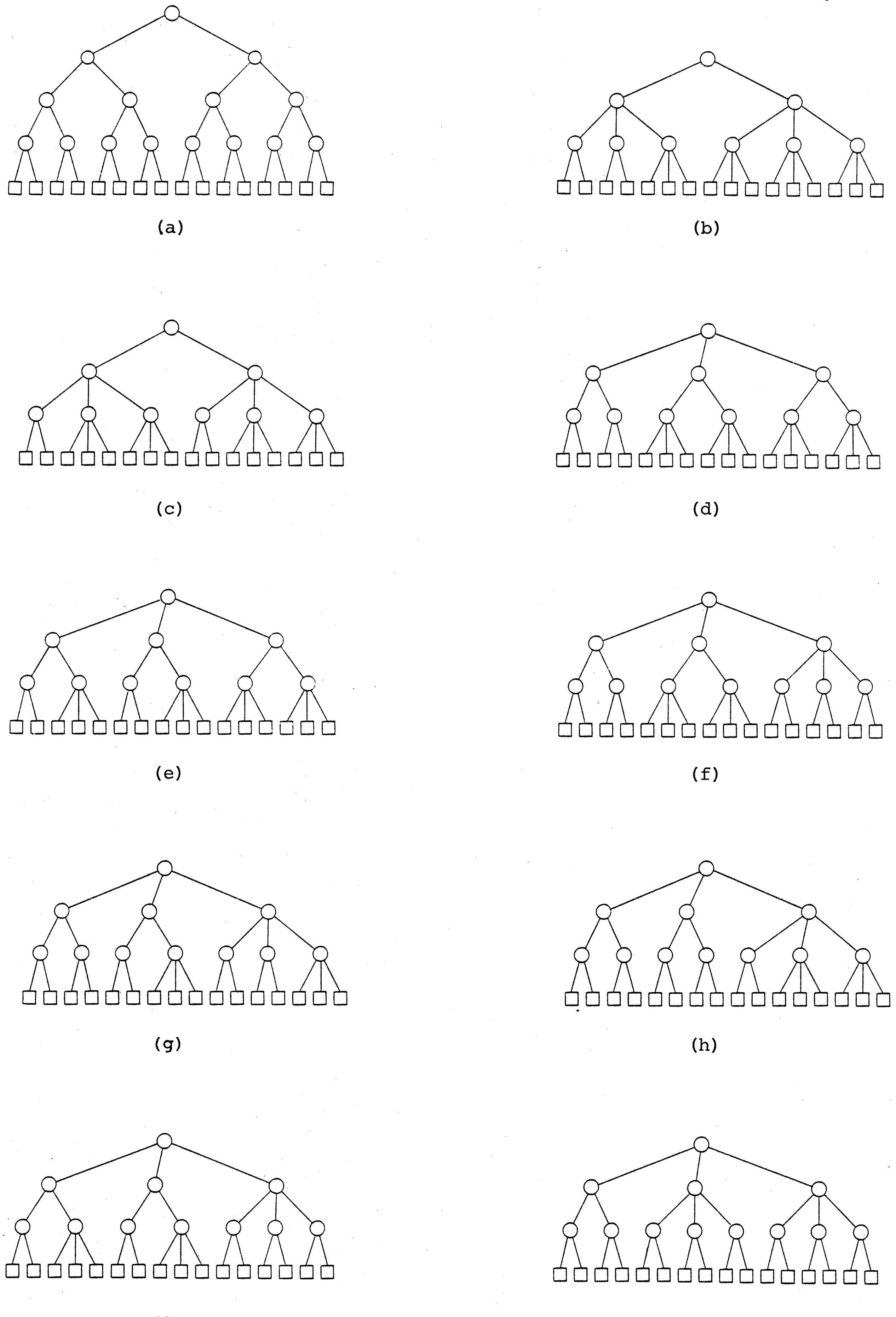

Figure 1: The ten structurally distinct B-trees of T^3_{15}.

descendants and keys, then the remaining space is unused, and therefore, wasted. Thus, for a given number of keys, fewer internal nodes implies a greater average number of keys per node and hence less wasted space. Table 1 illustrates this point for the trees of T^3_{15}.

A second purpose of this paper is to measure the time performance of space-minimal trees and the space utilization of time-efficient trees in T^M_k. Time costs are measured by the expected root-to-key path length for equally likely keys. This measure was used in [4] where it was called "node-visit-cost." It is motivated by the fact that external reference to a node usually requires an expensive disk fetch. Thus, a smaller number of nodes visited to reference an arbitrarily selected key, implies a smaller expected searching time. See Table 1 for the node-visit-costs of elements T^3_{15}.

Tree from Figure 1	Time Cost (node-visit)	Space Cost (internal nodes)
a	3.27	15
b	2.60	9
c	2.60	9
d	2.53	10
e	2.53	10
f	2.47	11
g	2.47	11
h	2.47	11
i	2.47	11
j	2.40	12

Table 1: Comparison of Time and Space costs for trees of Figure 1.

It is evident from Table 1 that in T^3_{15} no single 15 key tree minimizes both space and time costs, although for other numbers of keys it does occasionally happen. In general, as Table 1 suggests, there is a trade-off between time and space. Our study of this trade-off will indicate that

(1.3) space-optimal trees are nearly time-optimal, but time-optimal trees are nearly space-*pessimal*.

Thus, the trade-off is strongly biased in favor of space-minimizing B-trees. This observation is important because a "heuristic" has, we are told, gained reasonably wide acceptance among B-tree users† as a means of constructing "better" B-trees: keep the nodes near the root filled, " ... thereby increasing the branching degree of the nodes near the root, where a high branching degree is most beneficial" [2, p.13]. In light of (1.3) this "heuristic" is of dubious merit if not wrong, since keeping nodes near the root filled *reduces* space efficiency.

Although any element of T^M_k might be created during the dynamic operation of the B-tree algorithms, we can arrange matters so that the more efficient trees are more likely to be chosen. This is accomplished by acknowledging certain pragmatic considerations in the use of B-trees. For example, very large B-trees are often quite stable, tending to change only by a small percentage each day. Furthermore, B-tree files are periodically "backed-up" for error recovery or archival purposes. These two considerations suggest that if the tree were compactified, i.e. made space-minimal, during a daily "back-up" operation then the

†Private communication.

benefits indicated by (1.3) would be realized at the beginning of each day. As the tree is modified the magnitude of the benefits would diminish as the insertions and deletions cause it to depart further and further from its compact profile. But by our assumption of only modest modifications in a given day, this decay would not be rapid.

To implement this approach we present a linear time (in the size of the file) in-place compactification algorithm. Also, we give an analysis of the expected departure from optimal performance of a compact tree as a function of the number of random insertions into that compact tree. From the analysis and a few statistics about the use of a B-tree, one can easily compute the benefits of the "compactify during back-up" approach.

We shall use the following agenda:

Section 2 - characterization of space- and time-minimal B-trees;

Section 3 - analysis of time/space trade-offs;

Section 4 - analysis of "rate of decay" of a compact tree under random insertions;

Section 5 - in place-compactification algorithm;

Section 6 - summary and discussion.

2. *Characterizations of Space- and Time-Minimal B-trees*

Given the use of B-trees as search trees, one can identify at least three natural measures of the efficiency of a B-tree, namely, the expected number of nodes visited per access, the expected number of key-comparisons per access, and the space requirements of the tree. The node-visit cost of B-trees is studied at some length in [4], where the visit-optimal B-trees are characterized. The comparison cost of 2,3-trees is studied in [5], where the comparison-optimal trees are characterized and are compared with their visit-optimal forest mates. This section is devoted to studying the space cost of B-trees and to reviewing the results from [4] on the time cost of B-trees.

An *order M B-tree* ($M \geq 3$, an integer) is a finite, rooted tree satisfying (1.1) and 1.2). The maximum degree of a node is M; we will denote the minimum degree by $m = \lceil M \div 2 \rceil$. The root of a B-tree is said to be at *level* 0 of the tree, and, in general, the sons of a level ℓ node are said to reside at level $\ell+1$. The (common) level of the tree's leaves is the *depth* of the tree. (Recall from the introduction that the number of leaves equals the key capacity plus one.)

An order M B-tree is used as a search tree by loading its nonleaf nodes with *keys* from a totally ordered set in such a way that (a) each s-son node contains s-1 keys; and (b) if the s-son node N contains keys

$$k_1 < k_2 < \ldots < k_{s-1}$$

then

$$(\text{all keys in subtree 1 of N}) < k_1,$$

$$k_{s-1} < (\text{all keys in subtree s of N}),$$

$$k_{i-1} < (\text{all keys in subtree i of N}) < k_i$$

for $1<i<s$. Knuth [3, §6.2.4] describes in some detail procedures for accessing, inserting, and deleting keys in an order M B-tree in time bounded above by a small multiple of $\log_m$ (number of keys in T).

Two descriptors of B-trees will facilitate the following exposition. The *profile* of a depth d B-tree $T \in \mathcal{T}_k^M$ is a length d+1 integer sequence

$$\Pi(T) = \nu_0, \nu_1, \ldots, \nu_d$$

where each ν_ℓ is the number of nodes at level ℓ of T. Thus, $\nu_0 = 1$ and $\nu_d = k+1$. The *detailed profile* of T is the length d sequence of integer (M-1)-tuples

$$\Delta(T) = \langle\sigma_0^2, \sigma_0^3, \ldots, \sigma_0^M\rangle, \ \ldots \ , \langle\sigma_{d-1}^2, \sigma_{d-1}^3, \ldots, \sigma_{d-1}^M\rangle$$

where each σ_ℓ^s is the number of s-son nodes at level ℓ of T. Note that if $\ell > 1$, all $\sigma_\ell^s = 0$ for $s < m$. These notions and the notation for them originate in [4].

We now define precisely the costs to be studied. For an order M B-tree T with profile

$$\Pi(T) = \nu_0, \ldots, \nu_d$$

the *node-number cost* of T is

$$\mathrm{NNCOST}(T) = \sum_{i<d} \nu_i \tag{2.1}$$

This is clearly the "space" cost referred to informally. We measure space efficiency by the *node-utilization*

$$\mathrm{NU}(T) = (\nu_d - 1) \div ((M-1)\ \mathrm{NNCOST}(T)) \tag{2.2}$$

The node-utilization, NU, which never exceeds 1, gives the expected proportion of a node that contains keys. From the definition it is clear that a space-minimal tree T *may not* have NU(T) = 1 since some space may be wasted simply to honor the "equal path length" condition (1.2) on B-trees.

Next we define the "time" cost measure to be used. The *node-visit cost* of a B-tree T in T_k^M with detailed profile

$$\Delta(T) = \langle\sigma_0^2,\ldots,\sigma_0^M\rangle,\ldots,\langle\sigma_{d-1}^2,\ldots,\sigma_{d-1}^M\rangle$$

is

$$\mathrm{NVCOST}(T) = \Big(\sum_{0\le \ell<d}\ \sum_{2\le s\le M} (\ell+1)(s-1)\sigma_\ell^s\Big) \div k. \qquad (2.3)$$

Clearly,

NVCOST(T) = (expected number of nodes visited when accessing a key in T).

Although the presented definition of node-visit cost is intuitively appealing, we prefer the following less perspicuous but more useful definition.

Proposition 2.1 [4]: The node-visit-cost of the B-tree T with profile $\Pi(T) = \nu_0,\ldots,\nu_d$ is precisely

$$\mathrm{NVCOST}(T) = \Big(d\nu_d - \sum_{i<d} \nu_i\Big) \div (\nu_d - 1).$$

Having completed the definitions, we are now ready to characterize the space- and time-minimal trees. It will be convenient for comparison purposes to characterize the space- and time-maximal trees as well. These trees will be referred to by the names:

compact - node-number cost minimal,
sparse - node-number cost maximal,
bushy - node-visit cost minimal [4],
scrawny - node-visit cost maximal [4].

Theorem 2.2: An order M B-tree T with profile $\Pi(T) = \nu_0,\ldots,\nu_d$ minimizes NNCOST over all elements of T_k^M if and only if for $0 \le \ell < d$, $\nu_\ell = \lceil \nu_{\ell+1} \div M \rceil$; T maximizes NNCOST over all elements of T_k^M if and only if for $0 \le \ell < d$, $\nu_\ell = \max(1, \lfloor \nu_{\ell+1} \div m \rfloor)$.

Proof: To minimize (resp. maximize) NNCOST(T) it is necessary to minimize (resp. maximize) $\sum\limits_{\ell<d} \nu_\ell$ of T, which is equivalent, in turn, to maximizing (resp. minimizing) the branching ratios of T's nodes. The indicated profiles accomplish the desired shaping. □

Corollary 2.3: If $\Pi(T) = \nu_0,\ldots,\nu_d$ is the profile of a compact B-tree then $d = \lceil \log_M \nu_d \rceil$; if it is the profile of a sparse B-tree then $d = \lfloor \log_m (\nu_d \div 2) \rfloor + 1$.

Thus, compact B-trees are as "shallow" as possible for their size while the sparse B-trees are as "deep" as possible for their size. (For sparse B-trees, the maximum depth is not $\lfloor \log_m \nu_d \rfloor$ as might be expected, because the root is allowed to be binary.)

Using Proposition 2.1, among other facts about node-visit costs, the following computationally attractive characterization of *visit-optimal and pessimal* B-trees was established.

Theorem 2.4 [4]: The order M B-tree T with profile $\Pi(T) = \nu_0,\ldots,\nu_d$ has minimal NVCOST over all trees in T_k^M if and only if $d = \lceil \log_M \nu_d \rceil$ and $\nu_\ell = \min(M^\ell, \lfloor \nu_{\ell+1} \div m \rfloor)$ for $0 \le \ell < d$; T has maximal NVCOST over all trees in T_k^M if and only if $d = \lfloor \log_m (\nu_d \div 2) \rfloor + 1$, $\nu_0 = 1$, $\nu_1 = 2$ and $\nu_\ell = \max(m^\ell, \lceil \nu_{\ell+1} \div M \rceil)$.

Notice that both bushy and compact B-trees have a lot of M-ary nodes; but bushy trees have these nodes concentrated as close to the root as possible while compact trees have them concentrated as close to the leaves as possible. This difference is restated in the following less constructive but no less illuminating characterization of bushy B-trees, whose proof is direct from Proposition 2.1 and Theorem 2.4.

> *Proposition 2.5:* The B-tree T in T_k^M has minimal NVCOST over the class if and only if T is minimal in depth and, for that depth, maximal in number of nonleaf nodes.

Scrawny trees also have their M-ary nodes concentrated near the leaves, but unlike compact trees, they must have maximal depth. Sparse trees have their "fullest" nodes as near the root as possible, but as the following easily proved proposition indicates, they are so rare that it is difficult to say where they are concentrated.

> *Proposition 2.6:* Let T be a maximal NNCOST B-tree in T_k^M. If M is even, then T has no M-ary nodes. If M is odd, then T has at most one M-ary node per level.

This classification is summarized in Table 2.

		minimum depth	maximum depth
fullest nodes near to	root	bushy	sparse
	leaves	compact	scrawny

Table 2: Characteristics of the optimal and pessimal B-trees.

Our main interest, of course, is in the space- and time-*minimal* trees. Occasionally, the depth requirement is so restrictive that the two notions actually coincide.

Fact 2.7: For all sufficiently large k, there is a B-tree in $\mathcal{T}_k^M$ that is both visit-optimal and space-optimal precisely when $M^d - M < k+1 \le M^d$. "Sufficiently large" here means $k+1 \ge 4M$ when M is even, and $k+1 \ge 4M(1+\frac{2}{M-2})$ when M is odd.

Proof: By direct calculation using the profile-oriented characterization of the two notions of optimality. $\square$

Thus, the two notions of optimality coincide very infrequently. The major thrust of the next section is to determine *how much* they differ the rest of the time.

3. *Space and Time Comparisons*

In this section we establish time and space bounds on the advantage derivative from the use of optimal B-trees -- both bushy and compact. First we set forth the obvious limits within which the space utilization of B-trees can vary.

Proposition 3.1: As $k \to \infty$,

(a) for space-optimal order M B-trees,
$NU(T) \sim 1$;

(b) for space-pessimal k-key order M B-trees,
$NU(T) \sim \frac{m-1}{M-1}$;

(c) for "random*" k-key order M B-trees, and fixed, large M,
$NU(T) \approx \log_e 2 \approx .69$.

Proof: (a,b) by direct calculation from (2.2), (c) see [7]. $\square$

Proposition 3.2: Let T_o, T_p and T_r be respectively, space-optimal, space-pessimal and "random" (in Yao's sense) trees from $\mathcal{T}^M_k$. Then asympotically (as $k \to \infty$),

(a) for M even,
$NU(T_o) \sim (2 + \frac{2}{M-2}) \cdot NU(T_p)$;

(b) for M odd,
$NU(T_o) \sim 2 \cdot NU(T_p)$;

(c) for large fixed M,
$NU(T_o) \approx 1.45 \cdot NU(T_r)$.

Hence the space-optimal trees use their nodes between 2 (at odd M) and 3 (at M = 4) times more efficiently as do their space-pessimal forest-mates. For the sake of completeness it should be noted that these asymptotics take

---

*We use the word "random" here in the same sense as Yao [7]: the tree is grown from the empty tree by a sequence of k insertions, with each insertion equally likely to fall between any pair of the already-inserted keys or to either side of these keys.

hold at rather modest values of k, as Table 3 indicates.

B-tree	optimal profile $\Pi(T)$	pessimal profile $\Pi(T)$	$\frac{\text{pessimal NNCOST}}{\text{optimal NNCOST}}$
M=3, k=127	1,2,5,15,43,128	1,2,4,8,16,32,64,128	127÷66>1.9
M=4,k=15	1,4,16	1,2,4,8,16	3

Table 3

One might argue, with some justification, that the bounds of Proposition 3.2 are only of academic value in that the pessimal trees are *arbores non gratae* in the context of large trees and, hopefully, will never occur. (One should note, however, that these trees are very often *minimal* in number of comparisons per expected access [5].) In light of this criticism, we investigate also the disparities in space utilization between compact and bushy trees. We find these disparities to be only marginally less than the ones just exhibited.

Theorem 3.3: Let T_b be a space-pessimal element of all visit-optimal (i.e. bushy) trees in $\mathcal{T}_k^M$ and T_c a compact tree in $\mathcal{T}_k^M$. Then asymptotically (as $k \to \infty$),

(a) for M even,

$$NU(T_c) \sim \left(2 + \frac{2}{M-2} - \frac{M^2}{(M-2)M^{\log M}}\right) \cdot NU(T_b);$$

(b) for M odd,

$$NU(T_c) \gtrsim \left(2 + M^{1-\mu} 2^{\mu+1} (M+1)^{-\mu}\right) \cdot NU(T_b);$$

where

$$\mu = \left\lfloor \frac{\log M}{\log\left(\frac{2M}{M+1}\right)} \right\rfloor .$$

Hence, the space optimal trees use their nodes between $1\frac{5}{6}$ (at M = 3) and $2\frac{1}{2}$ (at M = 4) times as efficiently as do their visit-optimal forest-mates.

Proof: Let T_b be a visit-optimal n-leaf order M B-tree with profile

$$\nu_0, \nu_1, \ldots, \nu_{d-1}, \nu_d = n.$$

By Theorem 2.4, n lies in the range

$$M^{d-1}+1 \le n \le M^d. \tag{3.1}$$

We shall compute NU(T_b) by using Lemma A in the Appendix to compute the number $\sum_{i<d} \nu_i$ of nonleaf nodes of T_b.

(a) Say first that M is even. Let

$$\ell = \log_2 n + (1-\log_2 M)d.$$

By Lemma A, those levels of T_b numbered less than ℓ contribute

$$\frac{M^\ell-1}{M-1}$$

nonleaf nodes, and those levels of T_b numbered ℓ to d-1 contribute

$$\sum_{1\le i\le d-\ell} \lfloor n \div m^i \rfloor = \frac{2n}{M-2}\left(1 - \left(\frac{2}{M}\right)^{d-\ell}\right) + \alpha\ell$$

nonleaf nodes, where $0\le\alpha\le 1$. Now, since n lies in the range (3.1), the ratio of the number of nonleaf nodes of T_b to n, hence to the number of keys in T_b, is maximized (thus minimizing NU(T_b)) when $n = M^{d-1}+1$. In this case, we find that T_b has

$$\left(\frac{2}{M-2} - \frac{M^2}{(M-1)(M-2)} M^{-\log M}\right) n + o(n) \tag{3.2}$$

nonleaf nodes. One now invokes Proposition 3.1 and Definition (2.1) to complete the proof for the case of even M.

(b) When M is odd, one proceeds via a similar chain of reasoning, except that one must now be satisfied with asymptotic inequalities. Let

$$\ell = (\log_2 n + (1-\log_2(M+1))d)/\log_2 \left(\frac{2M}{M+1}\right) .$$

By Lemma A, the number of nonleaf nodes of T_b residing on levels 0 through ℓ-1 is given by

$$\frac{M^{\ell}-1}{M-1}$$

and the number residing on level ℓ through d-1 is

$$\sum_{1\le i\le d-\ell} \lfloor n \div m^i \rfloor = \frac{2n}{M-1}\left(1 - \left(\frac{2}{M+1}\right)^{d-\ell}\right) + \alpha\ell$$

for some $0\le\alpha\le 1$. Now as in the case of even M, one verifies easily that $NU(T_b)$ is minimized when $n = M^{d-1}+1$. In this case we find that T_b has at least

$$\frac{n}{M-1}\left(2 + M^{-\mu} - 2\left(\frac{2}{M+1}\right)^{\mu}\right) + o(n)$$

nonleaf nodes, where

$$\mu = \left\lfloor \frac{\log M}{\log \frac{2M}{M+1}} \right\rfloor$$

Once again, we invoke Definition (2.1) and Proposition 3.1 to derive the desired ratio and complete the proof of the theorem. $\square$

From Theorem 3.3 the reader can easily compute the node utilization cost of bushy trees for comparison with Proposition 3.1. However, the resulting equations are not likely to be perspicuous. Therefore, we will illustrate some sample trees that may be more enlightening.

For M = 3, the most benign of the orders since its bushy trees waste the least space, we have a bushy 3^{12}-key tree with profile

$$\Pi(T_b) = 1,3,9,27,81,243,729,2187,6561,19683,59049,132860, 265721,531442$$

and

$$NU(T_b) = .545455.$$

The compact tree with 3^{12}-key capacity has profile

$$\Pi(T_c) = 1,2,4,10,28,82,244,730,2188,6562,19684,59050,177148, 531442$$

and

$$NU(T_c) = .99995.$$

Thus, there is a premium of 1.8332 which is rather close to the asymptotic value of $1\frac{5}{6}$.

For M = 4, the most malevolent of orders: a visit-optimal B-tree with 4^{10} keys has the profile

$$\Pi(T_b) = 1,4,16,64,256,1024,4096,16384,65536,262144,524288, 1048577,$$

and

$$NU(T_b) = .4000002.$$

The compact tree of order 4 has profile

$$\Pi(T_c) = 1,2,5,17,65,257,1025,4097,16385,65537,262145,1048577$$

and

$$NU(T_c) = .99997.$$

The ratio is thus 2.4999 which approaches the asymptotic value of 2.5.

Although our comparison results have emphasized the cases most favorable to space minimality, Table 4 illustrates that the situation arises throughout $\mathcal{T}_k^M$.

order	leaves	premium as $d \to \infty$
M = 3	$3^{d-1}+1$	$1\frac{5}{6}$
	$4 \cdot 3^{d-2}$	$1\frac{3}{4}$
	$2 \cdot 3^{d-1}$	$1\frac{1}{2}$
M = 4	$4^{d-1}+1$	$2\frac{1}{2}$
	$2 \cdot 4^{d-1}$	2
	$3 \cdot 4^{d-1}$	$1\frac{5}{6}$

Table 4

To complete the comparison, we next consider the node-visit cost of bushy and compact trees.

Proposition 3.4: Let T_b be a bushy order M B-tree and let T_c be a compact order M B-tree each with n leaves. As $n \to \infty$,

$$\mathrm{NVCOST}(T_b) \gtrsim \lceil \log_3 n \rceil - \frac{2}{M-2} + \frac{M^2}{(M-1)(M-2)} M^{-\log M},$$

$$\mathrm{NVCOST}(T_c) \sim \lceil \log_3 n \rceil - \frac{1}{M-1} \quad .$$

In particular, independent of M, as $n \to \infty$,

$$\mathrm{NVCOST}(T_c) - \mathrm{NVCOST}(T_b) \lesssim 1.$$

Proof: The bound on $\mathrm{NVCOST}(T_b)$ is immediate by Proposition 2.1 and the derivation, in the proof of Theorem 3.3, of the expression (3.2) for the number of nonleaf nodes in an even order B-tree. The expression for $\mathrm{NVCOST}(T_c)$ follows directly from Propositions 2.1 and 2.4. □

Both visit-optimal and space-optimal B-trees have minimum depth for a given number of keys (Corollary 2.3 and Theorem 2.4, respectively); however, given this depth, the visit-optimal trees are maximal in the number of nonleaf nodes (Propositions 2.1, 2.5) while the space-optimal trees are minimal in number of nonleaf nodes (Definition 2.1)). What has emerged in this section is that, notwithstanding their minimal depths, visit-optimal trees can be very wasteful of space (Theorem 3.3); indeed, as the maximum branching, or order, of the trees grows without bound, visit-optimal trees can waste space with a profligacy approximating that of space-pessimal trees (Proposition 3.2 and Theorem 3.3). In sharp contrast, notwithstanding their node-visit pessimality given their depths (Corollary 2.3), space-optimal trees have virtually the same NVCOST as do visit-optimal trees (Proposition 3.4): depth is the prime determinant of NVCOST, with the number of nonleaf nodes playing only a secondary role.

The moral of the tale seems, thus, to be that compact B-trees are the preferred ones when one is initially loading a large data base into a B-tree. But how long will the tree remain compact?

4. *The Persistance of Compactness*

The three notions of optimality that have been studied in [4,5] and the present paper are static notions, and fragile ones at that: insertion or deletion in an optimal B-tree is likely to destroy its optimality. One can easily verify that space-optimal B-trees are statistically more fragile for insertions than are visit-optimal B-trees, which in turn are more fragile for deletions. (This is due to the space-optimal trees' preference for dense nodes at high-numbered levels and the visit-optimal trees' preference for sparse nodes at those levels.) It is the purpose of this section to quantify this statistical fragility of space-optimal trees. The major conclusion of the section will be that compact B-trees are always desirable when one initializes a data base (this follows from the previous section); they are reasonable to maintain for a stable data base (say one with no more than a 10% insertion rate between "backup" reorganizations); but they are too fragile to use directly with a volatile data base.

Since the analysis we present here is only suggestive and not definitive, we shall content ourselves with an analysis under simplified conditions. Specifically, we shall consider only order 3 B-trees (i.e. 2,3-trees), and we shall carry out only a "first-order" analysis (in the sense of Yao [7]) of these trees.

(4.1) Define the real-valued function ENU with domain $N \times N$ (N denotes the positive integers) as follows: for $n_0 < n$, ENU(n, n_0) is the expected NU of an n-leaf 2,3-tree that is obtained from a space-optimal n_0-leaf 2,3-tree by a sequence of $n - n_0$ "random" insertions. Here as in [7] an insertion into a k-key 2,3-tree is termed "random" if it is equally likely to fall in any of the $k + 1$ intervals defined by the order on the keys.

The competition for our compact-initialized B-trees are the purely random B-trees studied by Yao.

Proposition 4.1 [7]: As $n \to \infty$, $0.64 \le \mathrm{ENU}(n,1) \le 0.72$.

To evaluate the effect of compact initialization, we contrast Proposition 4.1 with the following.

Theorem 4.2: For $0<\alpha\le 1$, as $n \to \infty$, $\frac{21}{18-4\alpha^7} \le \mathrm{ENU}(n,\alpha n) \le \frac{14}{9-2\alpha^7}$.

Theorem 4.2 dictates, and Table 5 illustrates for various values of α, the range in which the expected node utilization will fall in the presence of random insertions.

% random increase	α	expected node utilization $n \to \infty$
1%	$\frac{100}{101}$	$0.74 \le \mathrm{ENU}(n,\alpha n) \le 0.98$
2.5%	$\frac{40}{41}$	$0.72 \le \mathrm{ENU}(n,\alpha n) \le 0.96$
5%	$\frac{20}{21}$	$0.69 \le \mathrm{ENU}(n,\alpha n) \le 0.92$
10%	$\frac{10}{11}$	$0.66 \le \mathrm{ENU}(n,\alpha n) \le 0.88$

Table 5

Thus even after moderate growth of the data base, the expected benefits of compact initialization are still discernible.

The remainder of this section is devoted to proving Theorem 4.2.

Proof: The proof follows the strategy of Yao's proof of his Theorem 2.6 [7], the first-order version of Proposition 4.1.

Lemma 4.3 [7]: If the 2,3-tree T has profile $\Pi(T) = \nu_0,\nu_1,\ldots,\nu_{d-1},\nu_d$, then the number of nonleaf nodes of T satifies

$$\frac{3}{2}\nu_{d-1} - \frac{1}{2} \le \text{NNCOST}(T) \le 2\nu_{d-1} - 1.$$

Define the following quantities:

For $i = 1,2$, let

$A_i(n,n_0) =_{def}$ the expected number of $(i+1)$-son nodes at level $d-1$ of the tree constructed from an n_0-leaf space-optimal tree via a sequence of $n-n_0$ random insertions.

Let $N^*(n,n_0) =_{def}$ the expected NNCOST of the tree just described.

Lemma 4.4 [7]: Letting $A(n,n_0) = A_1(n,n_0) + A_2(n,n_0)$,

$$\frac{3}{2}A(n,n_0) - \frac{1}{2} \le N^*(n,n_0) \le 2A(n,n_0) - 1.$$

Proof: Lemma 4.3. □

Lemma 4.5: For $0<\alpha\le 1$, as $n \to \infty$, $A_1(n,\alpha n) = \frac{2}{7}n(1-\alpha^7) + O(1)$; and $A_2(n,\alpha n) = n(\frac{1}{7} + \frac{4}{21}\alpha^7) + O(1)$.

Proof: Yao, in his Lemma 2.6 [7], shows that when $n > \alpha n$,

$$A_1(n,\alpha n) = (1 - \frac{6}{n})A_1(n-1,\alpha n) + 2.$$

Since the αn-leaf tree we start with is space-optimal, we have the initial condition $A_1(\alpha n,\alpha n) \le 2$. We find, therefore, as n grows without bound,

$$A_1(n,\alpha n) = 2 \sum_{6\le i\le n-\alpha n} \prod_{0\le k\le 5} (1 - \frac{i}{n-k}) + O(1).$$

This simple form is found by just expanding the recurrence and noting the resulting cancellations. This sum is easily seen to evaluate to

$$A_1(n,\alpha n) = 2n^{-6} \sum_{6\le i\le n-\alpha n} (n-i)^6 + O(1) ,$$

whence the asserted value of A_1. The value of $A_2(n,\alpha n)$ now follows directly from the obvious equation $2A_1(n,\alpha n) + 3A_2(n,\alpha n) = n$. □

Lemmas 4.4 and 4.5 combine to yield

Lemma 4.6: For $0<\alpha\leq 1$, as $n \to \infty$, $n\left(\frac{9}{14} - \frac{1}{7}\alpha^7\right) \leq N^*(n,\alpha n) \leq n\left(\frac{6}{7} - \frac{4}{21}\alpha^7\right)$.

The theorem is now immediate by definition of ENU. □

Thus, if a compact tree grows (by random insertions) by less than 2½%, the expectation of its space utilization is strictly greater than that for randomly grown trees. In order to use compact trees for these stable files, we present in the next section an *in situ* compactification algorithm that can be used as a "back-up" procedure for essentially no additional cost over the naive "back-up" operation.

5. *Algorithms*

The import of Section 3 is that compact B-trees minimize space and nearly minimize time. The import of Section 4 is to quantify the robustness of compact B-trees in the presence of insertions. An obvious way to apply these results in practice would be (1) to initialize a B-tree to compact form, (2) to monitor the NVCOST and NU functions as the file is modified, and (3) when NU begins to stray too far from unity, to recompact the entire file. An alternative approach would be to dispense with the monitoring operation and simply recompact during routine "back-up" of the file. This latter approach depends on the (likely) condition that large data bases do not grow so rapidly that recompactification would be required more often than the frequency of the "back-up" operation.

It is, perhaps, worthwhile to emphasize the importance of taking special precautions when initializing B-trees from sorted or "nearly" sorted files. In [4] it was observed that construction of a B-tree by repeated insertion from a sorted file frequently results in a scrawny tree. In fact, if the file has size $2m^d$ (for odd M), the repeated insertion produces a tree with a binary root whose descedants are both complete *m-ary* trees. This result is *both* sparse and scrawny and is thus space- *and* time-pessimal.

The purpose of this section is to address the algorithmic issues involved in initialization, monitoring and compactification. (All complexity bounds are for a uniform cost random-access machine.)

Initialization Algorithm: In [4] a linear-time (in the size of the file) algorithm was presented for constructing a 2,3-tree given a (detailed) profile for the tree and a sorted list of records.* Conversion of that

*This algorithm was used in conjunction with Theorem 2.4 to construct visit-optimal 2,3-trees.

algorithm to operate on B-trees of arbitrary order is a straightforward matter requiring no further discussion. Construction of the proper compact profile can be performed in logarithmic time using the characterization Theorem 2.2. From the profile, a detailed profile can be constructed by observing that at level ℓ the ν_ℓ vertices must have $\nu_{\ell+1}$ outedges and a vertex containing s keys has s+1 outedges. In the absence of any other criterion, we simply distribute the keys "evenly" throughout the ν_ℓ vertices by assigning s keys to r of the vertices and s+1 to the remaining vertices. Thus, $s = \left\lfloor \frac{\nu_{\ell+1}}{\nu_\ell} \right\rfloor - 1$ and $r = \nu_\ell - \nu_{\ell+1} \bmod \nu_\ell$. Finally, the records can be sorted by any external sorting algorithm.

Monitoring algorithm: As insertions and deletions change the structure of the B-tree, the profile and/or detailed profile can be updated easily in time proportional to that of insertion or deletion. For example, if an insertion changes t internal levels in the tree, then t+1 entries in the profile and 2t entries in the detailed profile must be changed. Maintenance of these profiles gives a current description of the structure of the tree.

If, however, only the NU and NVCOST are to be monitored, then these values can be updated in *constant* time since by Proposition 2.1 and equation (2.1) they both depend on k, the number of keys, the current depth, and $\sum_{i<d} \nu_i$, the number of internal nodes. If I is the current number of internal nodes and an insertion or deletion changes t internal levels then the number of internal nodes in the new tree is $I \pm (t-1)$ where the sign depends on whether the operation is an insertion (+) or a deletion (-).

Compactification Algorithm: Obviously, compactification can be accomplished simply by using the initializing algorithm with the existing tree as input. However, this approach requires two copies of the tree to exist simultaneously, and if space were that plentiful, compact trees would have only limited benefit. So we present an *in situ* compactification scheme that requires at most d free nodes. These d nodes need not all be available when the algorithm starts; it is sufficient, after having compacted n keys, to have freed $\lceil \log_M n \rceil + 1$ nodes.

In order to simplify the presentation, we give the compactification algorithm only for 2,3-trees, assuming that the generalization to arbitrary M is obvious. The algorithm takes three inputs: a 2,3-tree T, a depth d, and a detailed profile for the desired compact tree. Since 2,3-trees have such simple detailed profiles $<\sigma_0^2, \sigma_0^3>$, ... , $<\sigma_{d-1}^2, \sigma_{d-1}^3>$, it is sufficient to know only one component of each pair. Accordingly, the array variable *sigma3*[0:d-1] gives the number of *ternary* nodes at each level of the new compact tree. The output from *compact* is a reference (i.e. pointer) to the root of the compacted tree.

The details of the tree represetnation are purposely left obscure except that we follow the initialization algorithm of [4] in assuming that each node has fields: *left, key1, middle, key2, right* with the obvious meanings. By convention, binary nodes have NIL as values in the last two fields. Also, by convention, the resulting compact tree will have all ternary nodes "left justified" on a level. Clearly, other distributional disciplines can be implemented easily.

The overall structure of the algorithm can be viewed abstractly as a "producer/consumer" solution. That is, two procedures are used: one to fetch keys from the existing tree (producer) and one to construct a compact tree given its keys one-at-a-time (consumer). These two abstract operations could be implemented by means of recursive co-routines, but such exotic control structures are unnecessary. Instead, the producer operation is implemented explicitly by a recursive depth-first traversal procedure (*dft*) and at each point where a key is "produced" a nonrecursive subroutine (*build*) is called which implicitly implements the consumer operation.

The only subtlety in this approach is that the *build* routine must keep track of the subtrees currently "under construction." Say that a subtree is *under construction at level i* if a subtree rooted at level i has at least one but not all of its immediate descendant subtrees completed. For instance, if the portion of the tree shown in bold in Figure 2 has so far been constructed, then subtrees at levels 1 and 3 are under construction. In order to keep track of this progress, then, a variable UC[0:d-1] (mnemonic for under **Construction**) is used to record a reference to each node under construction. Initially, UC contains only NIL's, indicating that no nodes are under construction.

```
Algorithm - input:  T = a 2,3-tree to be compacted
                    d = depth of the new compact tree
                    sigma3[0:d-1] = a vector of the number of ternary
                         nodes at each level in the new compact tree
           output:  a reference to the root of the compact tree
```

```
function compact(T,d,sigma3);
   tree T; integer d;
   integer array sigma3;
   begin integer level;                           index into UC
      ref n,                                      node currently under construction
         desc;                                    descendant of node currently under const.
      ref array UC[0:d-1];                        table of all nodes under construction
      procedure dft(node);                        recursively performs depth-first traversal of T
         ref node;                                  starting at "node." This is the
            begin                                   producer operation.
             if node.left = NIL                   is this the lowest nonleaf level?
               then {build(node.key1);              yes, produce key for consumer
                  if node.key2 ≠ NIL                is there a second key?
                     then build(node.key2)}         yes, produce key for consumer
               else                               this is an interior node
                  {dft(node.left);                traverse to lower depth, and return
                  build(node.key1);               back now, produce key for consumer
                  dft(node.middle);               traverse to lower depth, and return
                  if node.key2 ≠ NIL              is this a ternary node
                     then {build(node.key2);      yes, produce key for consumer
                        dft(node.right)}};        traverse to lower depth, and return
            free(node)                            release the space
            end
      procedure build(key);                     incrementally constructs tree, adding
         integer key;                             "key" at each call.  This is the
         begin                                    consumer operation.
         comment global var. "level" is set at the
            level of last key placement; initially, -1;
         while level ≠ d-1 do                     set all levels below last key placement
            {level ← level + 1;                     to be not under construction
              UC[level] ← NIL};
         desc ← NIL;                              descendant of node under. const. is NIL
         while key ≠ NIL ∧ level ≥ 0 do           perform the following til key placed or tree done
            {if UC[level] = NIL                   are we beginning construction at this level?
               then n ← UC[level] ← getspace        yes, then get a clean record
               else n ← UC[level] ;                 no, then get the recond we were working on
            cond                                  there are 4 possible cases
               n.key1 = NIL                         (1) clean record, no keys present
                  ⇒ {n.left ← desc;                      record ref. to left   subtree
                     n.key1 ← key;                       place key
                     key ← NIL}                          signal that placement has been done
               n.key2 = NIL ∧ sigma3[level] > 0       (2)  one key placed in rec. of ternary node
                  ⇒ {n.middle ← desc;                    record ref. to middle subtree
                     n.key2 ← key;                       place key
                     key ← NIL}                          signal that placement has been done
               n.key2 = NIL ∧ sigma3[level] = 0       (3)  one key placed in rec., of binary node
                  ⇒ {n.middle ← desc;                    record ref. to middle subtree
                     desc ← n;                           this binary node done, save ref. for parent
                     level ← level - 1}                  move to next higher level
               n.key1 ≠ NIL ∧ n.key2 ≠ NIL            (4)  both keys placed in ternary node
                  ⇒ {n.right ← desc;                    record ref. to right subtree
                        desc ← n;                       this ternary node done, save ref. for parent
                        sigma3[level] ← sigma3[level] - 1;   record completion of ternary node
                        level ← level - 1}}             move next to higher level
         end;
         comment declaration completed;
         level ← - 1;
         dft(root(T));                              traverse entire tree
         build(1);                                  dummy call to link pointers on right spine
         compact ← UC[0]                            return reference to root.
         end
```

To see that the algorithm halts we observe first that the basic control flow is determined by *dft* and amounts to a simple depth-first traversal of a 2,3-tree that obviously halts if its subparts do. Secondly, we note that the two other loops ([29-31] and [33-54] of the *build* procedure) can not possibly cycle forever for positive d. The [29-31] while-loop always terminates since the initial value of *level* is always less than or equal to d-1 upon entry to the loop. The [33-54] loop clearly terminates since in each clause of the conditional some change is made that effects the disjunction controlling the loop.

The tree that results from *build*, (if it is a tree at all) will evidently be compact since binary and ternary nodes are built according to the dictates of the profile. To see that a valid 2,3-tree results from *build*, say that in a tree containing keys $k_1 < k_2 < \ldots < k_n$ and leaves $\ell_1, \ell_2, \ldots, \ell_{n+1}$ (numbered left to right), that the *left leaf of* k_i is ℓ_i. Then *build* is called n times by the *dft* routine and on the i^{th} call, *build* (1) begins by moving to the left leaf of k_i (2) creates all edges from ℓ_i to the vertex containing k_i, (3) "completes" the vertices between ℓ_i and the vertex containing k_i according as they are either binary [46-49] or ternary [50-54], and (4) places k_i in the proper position in the vertex according to whether it is in the first key position [39-41] or the second key position [42-45]. Thus, the calls from *dft* complete all portions of the tree except the edges on the right spine. The dummy call from *compact* [59] completes this feature and the algorithm halts.

Evidently, *compact* operates in linear time on a uniform-cost random-access machine since it is simply the composition of two depth-first tree traversals. The total number of tree records that are required in excess of the number freed is d (i.e. if a compact tree were compacted, d extra cells would be needed for the algorithm to operate.)

6. *Summary and Discussion*

In the set $\mathcal{T}_k^M$ of all order M B-trees with capacity k, we have characterized the space-minimal elements. We have bounded the time performance of these space-minimal B-trees as well as bounding the space utilization of the time-minimal B-trees. The main result of this analysis is that

> the space-minimal trees are nearly time-minimal, but
> the time-minimal trees are nearly *space-maximal*.

This bias in favor of using space-minimal trees when possible motivated our analysis of the robustness of space-minimal trees in the presence of random insertions. Though space-minimal B-trees are only modestly robust, pragmatic considerations such as file stability suggest that they could be beneficial with periodic recompactification, e.g. during routine file "back-up." This compactification can be accomplished inexpensively using the linear time algorithm presented here.

Many issues remain to be investigated. For example, are there variants on the B-tree theme that enhance the robustness of space efficient trees? How robust are compact B-trees under "random" insertions? How do variable length keys affect the analysis presented here -- is it better to have smaller nodes when the keys are variable in length? What are the time and space characteristics of random B-trees created by random insertions and deletions?

Acknowledgement: We are indebted to Mark R. Brown of Yale for his helpful comments on an earlier version of this paper. We also wish to thank Michele Boucher for her patience and her skillful preparation of this document and the innumerable drafts that preceded it.

7. *References*

[1] R. Bayer and E. McCreight.
Organization and maintenance of large ordered indexes.
Acta Informatica 1(3):173-189, 1972.

[2] R. Bayer and K. Unterauer.
Prefix B-trees.
Transactions on Database Systems 2(1):11-26, 1977.

[3] D. E. Knuth.
The Art of Computer Programming, Volume 3.
Addison-Wesley, 1973.

[4] R. E. Miller, N. Pippenger, A. L. Rosenberg, and L. Snyder.
Optimal 2,3-trees.
SIAM Journal on Computing 8(1):42-59, 1979.

[5] A. L. Rosenberg and L. Snyder.
Minimal comparison 2,3-trees.
SIAM Journal on Computing 7(4):465-480, 1978.

[6] L. Snyder.
On uniquely represented data structures.
Proceedings of the 18th Annual Conference on the Foundations of Computer Science, 412-417, 1977.

[7] A. C. Yao.
On random 2,3-trees.
Acta Informatica 9(3):159-170, 1978.

APPENDIX A

Lemma A: Let T be a visit optimal n-leaf order M B-tree with profile

$$\Pi(T) = \nu_0, \nu_1, \ldots, \nu_d.$$

If M is even, then

$$\nu_\ell = M^\ell$$

for all

$$\ell \le \log_2 n + (1-\log_2 M)\, d - 1, \qquad \text{(A.1)}$$

and

$$\nu_\ell = \lfloor n \div m^{d-\ell} \rfloor$$

for all

$$\ell \ge \log_2 n + (1-\log_2 M)\, d. \qquad \text{(A.2)}$$

If M is odd, then

$$\nu_\ell = M^\ell$$

for all

$$\ell \le (\log_2 n + (1-\log_2 (M+1))d)/\log_2 (\tfrac{2M}{M+1}) + o(1)$$

as $n \to \infty$; and

$$\nu_\ell = \lfloor n \div m^{d-\ell} \rfloor$$

for all

$$\ell \ge (\log_2 n + (1-\log_2 (M+1))d)/\log_2 (\tfrac{2M}{M+1}).$$

Proof: Assume first that M is even. Inequality (A.1) on ℓ yields

$$\ell \cdot \log_2 M \le \log_2 n - (d-\ell)(\log_2 M-1) - 1$$

so that

$$M^\ell \le \lfloor n \div (M \div 2)^{d-\ell} \rfloor,$$
$$= \lfloor n \div m^{d-\ell} \rfloor$$

and the lemma follows by Theorem 2.4. By similar calculations, inequality (A.2) on ℓ yields

$$M^\ell \ge \lfloor n \div m^{d-\ell} \rfloor,$$

so again the lemma follows by Theorem 2.4.

The case of odd M follows by similar reasoning, using the fact that $m = (M+1)/2$ when M is odd. $\square$