LINEARIZING THE DIRECTORY GROWTH IN ORDER PRESERVING EXTENDIBLE HASHING

by Ekow J. Otoo

SCS-TR-77

Revised June 1987

School of Computer Science
Carleton University
Ottawa, Ontario
CANADA K1S 5B6

This research was supported in part by the Natural Sciences and Engineering Research Council of Canada.

Linearizing the Directory Growth in Order Preserving Extendible Hashing

Ekow J. Otoo
School of Computer Science
Carleton University
Ottawa, Ont., Canada K1S 5B6

June 16,1987

Abstract

We propose a method of implementing an order preserving extendible hashing scheme using a balanced hirarchical directory. The directory is implemented as a balanced m-way tree where $m = 2^\theta$ for some predefined constant θ. This approach gives an almost linear growth in the directory size for both uniform and nonuniform key distributions at the expense of possible one extra disk. Given records whose pseudo-keys are w-bit nonnegative integers, each of value $K' < M = 2^w$, such that the records are grouped into pages of capacity C records, a record retrieval is achieved in at most $\lambda = (w - \log_2 C)/\theta$ disk accesses.

1 Introduction

A hashing scheme [8] generally provides a fast method for a random storage and lookup of records in a file. The method of hashing generally involves a set of buckets, or pages, each of which can contain C records, addressed from 0 to n-1, and a hash function $\mathcal{G}$ that maps the keys K_i to the page addresses, i.e., $\mathcal{G}(K_i) \rightarrow \{0, 1, \ldots n-1\}$. A collision is said to occur when more that C records hash into the same bucket. Conventional hashing methods, initialize the number of allocated buckets according to an estimate of the maximum data storage space required and subsequently resort to some collision resolution methods to resolve oveflows.

Recently a number of hashing techniques have been proposed that obviates the necessity for estimating the maximum storage space required. The storage expands and contracts with the number of record insertions and deletions. These methods are called dynamic hashing and some well studied schemes are *Dynamic Hashing* of Larson [9] *Linear Hashing* of Litwin [10], and *Extendible Hashing* (which we abbreviate as ExHash), of Fagin et al. [6]. In linear hashing, the address of the data pages are computed from the keys using a dynamically changing function.

One major disadvantage of the scheme is that it requires the use of a good collision resolution method in order to avoid poor record retrieval times. Details of the scheme appear in [10].

The latter dynamic hashing scheme ExHash, determines the data page in which the record is stored via a directory implemented in the form of a flattened trie. For each key K_i, a corresponding hash code $K_i' = \beta_0\beta_1 b \ldots \beta_{w-1}$, of a string of binary digits $\beta_i \in \{0,1\}$, is generated. The encoded representation K_i', is termed the pseudo-key. The pseudo-key K_i' is derived using a ramdomizing function ψ, i.e., $K_i' = \psi(K_i)$. A sequence of h prefix or suffix bits of K_i' is used to compute the address of a dynamically changing directory D containing 2^H entries. The value H is termed the *global file depth.* Each directory entry D_i contains the address of a data page in which the records that hash to it are stored. Details of implementing extendible hashing can be found in [3], [6], [13], and [16]. Two significant characteristic of extendible hashing are: a) no overflow records are organized in a special overflow area and b) a record retrieval is guaranteed in at most two disk accesses. We examine the use of extendible hashing when the natural key order must be preserved.

A disadvantage of extendible hashing is that the directory size can grow exponentially in the number of record insertions unless the generated pseudo-keys are uniformly distributed over the directory space. Hence the choice of a good randomizing function ψ, that distributes the pseudo-key uniformly over the directory address space is crucial. Unfortunately, the uniformity condition is satisfied at the expense of breaking the natural ordering of the key values.

Often there is the need to maintain the records in the natural key order for efficient query processing and other data processing activities. To highlight the significance of the order preserving requirement, consider as the access paradigm of the hashing scheme the following file operations:

i. **Find(K):** retrieve the record with key value K.

ii. **Insert(K):** insert the record with key value K.

iii. **Delete(K):** delete the record with key value K.

iv. **FindNext(K):** retrieve the record that occurs in the file but whose key value $K_i > K$.

v. **RangeFind(K_l, K_u):** retrieve all records whose key value K_i satsify $K_l \leq K_i \leq K_u$.

The randomizing function ψ leads to inefficient execution of the operations iv and v, unless ψ is order preserving in the sense that if $K_i \leq K_j$, then $\psi(K_i) \leq \psi(K_j)$. Note that all the above operations can be efficiently supported by a B-tree [1] and its variants $B^+ - tree$ [8] and Prefix B-tree [2].

We address the problem of implementing an order preserving extendible hashing that can efficiently support all of the above operations. The scheme we propose, *Balanced Extendible Hash Tree* (or BEH-tree), is a variant of extendible hashing in which the directory is structured as a balanced m-ary tree. The directory consists of nodes that splits and grows in a manner reminiscent of the B-Tree. Within each node, the number of directory entries grows by doubling as in the basic ExHash scheme. We study the performance of the BEH-tree scheme and compare the directory size generated with that of ExHash under both uniform and nonuniform key distributions. Order preserving extendible hashing has previously been studied [16] with some concern expressed on the potential explosion of the directory size.

2 Extendible Hashing

2.1 The Basic Scheme

We briefly outline the basic ideas of an order preserving extendible hashing. The scheme has two levels of organization: a first level of a directory which we denote by D that consist of a linear array of page pointers and a second level of data pages which hold the records or pairs of key and record pointers. The directory D, is headed by a value $D.H$ called the global depth of the file. This value varies with the directory space expansion and contraction.

Given a record r_i with key K_i, the target page for the record is derived as follows. The pseudo-key $K_i' = \psi(K_i)$ is first generated. Then the $(D.H)$-prefix bits of K_i', considered as an integer, provides an index q of a directory element D_q. Let $\mathcal{G}$ denote the address generation function. Then the index q is given by $\mathcal{G}$ where $\mathcal{G}$ is defined as

$$\mathcal{G}(H, K_i') = \sum_{j=0}^{H} \beta_j \, 2^{H-j-1}.$$

The directory element D_q contains a page pointer, denoted by $D_q.P$, which gives the page address into which the record r_i must be stored or looked up. The Figure 1.a illustrates some initial configuration of the directory and data pages when the global depth $D.H = 2$. There are 4 valid addressable elements in the directory. Besides the storage of records and/or keys,

each data page P_i, retains a value $P_i.h$ called the local depth. This value represents the number of prefix-bits which the keys in page P_i agree on. For any page P_i let $\phi_i = D.H - P_i.h$. Then we have the following properties of the scheme.

1. $P_i.h \leq D.H$ always.

2. There are 2^{ϕ_i} directory pointers to page P_i.

3. The directory size $n_d = 2^{D.H}$ elements

Consider the insertion of the record r_i whose pseudo-key is $K_i' = \langle 1001101 \rangle$ in Figure 1.a. The two prefix-bits "10" give the value of 2, and the page pointer $D_2.p = P_2$. As a result, the record is inserted in page P_2 if there is room. If the page is full, it is split. To split the page, the local depth is increased from 1 to 2. A new page P_3 say, having a local depth $D_3.h = D_2.h = 2$, is allocated. The page pointer of the directory element D_3 is set to point to P_3, i.e., $D_3.p \leftarrow P_3$, and the records previously in page P_2 are rehashed to be distributed between P_2 and P_3. Assuming some records get moved to page P_3, then the new record can now be stored in page P_2. The new configuration is shown in Figure 1.b.

Suppose the pseudo-key was $K_i' = \langle 0101101 \rangle$. Then the directory entry address given by "01" is $q = 1$, and the target page $D_1.p = P_1$. The local depth $P_i.h = D.H = 2$. Assuming that the page P_1 is found to be full in an attempt to insert r_i, then on increasing the local depth D_1 from 2 to 3, this exceeds the global depth $D.H = 2$. As a result, the global depth $D.H$ is also increased from 2 to 3 and the directory size is doubled from 4 to 8. The new configuration is shown in Figure 1.c. To address an element of the directory, 3 prefix-bits must now be used instead of 2. The two directory elements given by "010" and "011" obtained from concatenating '0' and '1' in turn, to the string "01" are called *buddies*. The pointer of the directory element D_3 is reset to point to page P_3, and the records of page P_2 are rehashed as before. It is possible that the records of the page being split will be rehashed to the same page, in which case the splitting process is repeated.

To delete a record with pseudo-key K_i', the page where the record should reside is first determined using the basic search algorithm as in an insertion. If the record exists in the page located, it is deleted. Let the page from which a deletion occurred be P_i and let the directory element with a page pointer to P_i be D_i. Let D_i' denote the buddy of D_i at the

Figure 1a: Extendible Hashing scheme when H = 2

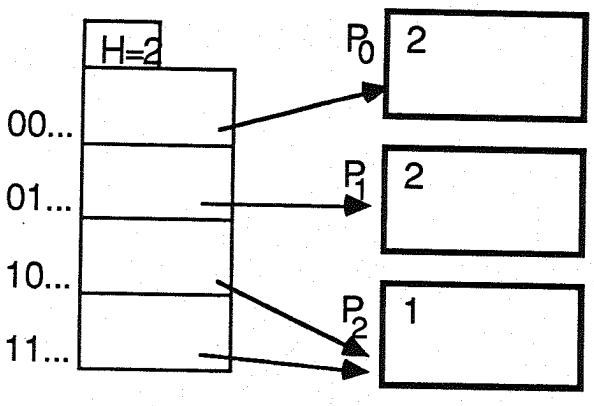
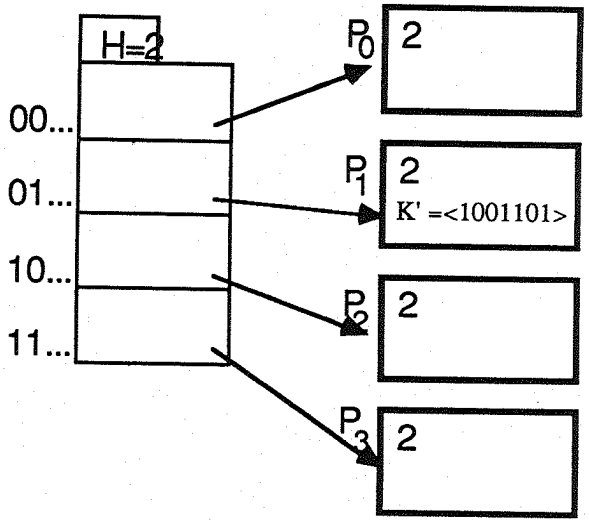


Figure 1b: Configuration after page P_2 splits



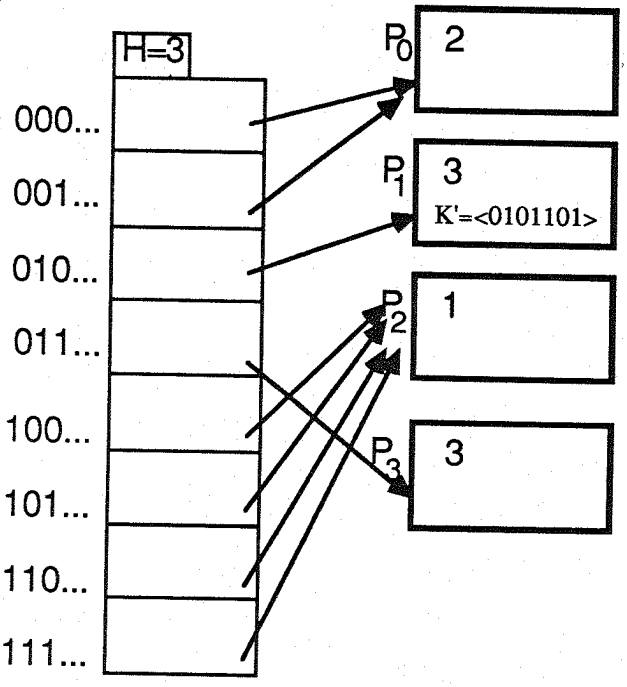


Figure 1c:
The directory of Figure 1a doubble after splitting page P_1 .

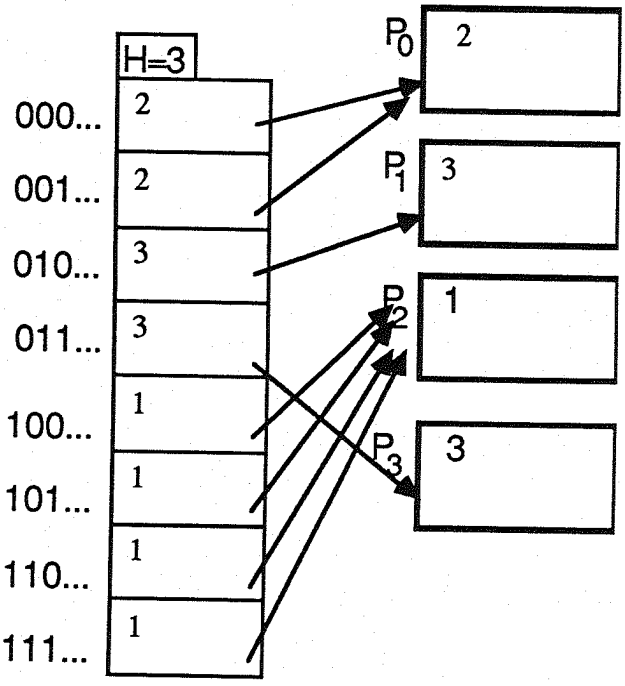


Figure 1d:
Organization of extendible hashing with local depth stored in the directory.

current directory depth, in the sense that if the global depth is $D.H$, D_i and D_i' are addressed by H-bits that differ only on the H-th bit. Also let the page pointer $D_i'.p = P_i'$. Then following the record deletion from P_i, the content of page P_i' is examined. If the content of P_i and P_i' can be contained in one page, the records in page P_i' are added to P_i, the pointer $D_i'.p$ is set to P_i, and the local depth $P_i.h$ decreased by 1.

Suppose the local depth $P_i.h$ is decreased by 1. Then if every local depth is less than the global depth, the global depth is decreased also by 1 and the directory is halved. Detecting when every page has a local depth less than the global depth is easily done using a linear array indexed by the value of the local depth. Let this array be denoted by ℓ. Then ℓ_i gives the number of pages whose local depth equals i. The directory is halved whenever $\ell_H = 0$.

The scheme described above using a one-level directory, has been studied studied extensively in the literature both analytically and through simulation. Given N random uniformly distributed keys stored in pages of capacity C, it has been shown in [5] [7], [13], [15] that:

- the average global depth is $O(1 + 1/C)\log_2 N$.
- the average directory size is $O(((1 + 1/C)\log_2 N)N^{1+1/C})$.

2.2 Order Preserving Extendible Hashing (OPEH)

The use of the prefix bits of the pseudo-keys to hash into the directory provides a scheme which is order preserving on the pseudo-keys. To achieve order preservation in the natural order of the keys, one can do two things. Either an order preserving pseudo-key encoding function is used or the pseudo-key generation phase is abandoned in place of the binary representation of the keys. In [3] the performance of an OPEH for the case where the pseudo-keys correspond to the packed and unpacked EBCDIC code are studied.

In general one can consider the binary digit encoding of the keys simply as the pseudo-key. For instance, the binary representation of integers can be used for integer keys. Alphanumeric keys can be represented in ASCII, EBCDIC or RADIX-50 codes. Implementing an order preserving extendible hashing raises some concern about the potential exponential growth of the directory size. The reasons being that:

1. The key distribution for most practical data sets are nonuniform.

2. Most data obtained on the fly exhibit some *noise effect* whereby short bursts of keys inserted in succession differ only in their low order bits.

These properties can cause the generation of large directory sizes after only a few insertions. For instance for w-bit integer keys $K < 2^w$, stored in pages of capacity C, the worst case directory size after only C+1 insertions is $O(2^w/C)$. The cost of inserting a record then is $O(2^w/C)$ disk accesses. The concern for such directory explosion has led to some recommendations such as the use of a hierarchical directory [12], [16], and the use of very large data pages [13]. Mendelson [13] proposes some guidelines for choosing a page size given that the directory size must not exceed a predefined size.

We propose then a scheme called *Balanced Extendible Hash Tree*, that builds a balanced hierarchical directory with controlled directory expansion. The directory is tree-like with fixed size nodes that grow by node doubling until they eventually split. The node splitting always begin from the leaf nodes towards the root in a manner similar to the B-tree structure [1], [2], [8]. The main advantage of the scheme over the one-level directory in extendible hashing is that the directory size grows almost linearly with the number of record insertions.

3 The Balanced Extendible Hash Tree

3.1 Basic Structure

To achieve a controlled directory growth in extendible hashing, we first make the modification that the local depth be retained as a field in the directory element instead of in the data page. This means that the local depth of a page will be replicated. However it has the advantage that an empty data page can be deleted and its storage space reclaimed. Figure 1.d illustrates the equivalent structure to Figure 1.c.

The directory of the balanced extendible hash tree (or BEH-tree), consists of fixed size nodes of at most $m = 2^\theta$ elements. For instance if we set $\theta = 6$ then the number of elements in each node doubles at each expansion step from 1 to $2^6 = 64$. Any further expansion after this requires that we split the node. To facilitate the understanding of the scheme, we suggest the reader examines Figure 4 to be familiar with the structure of BEH-tree directory. Let us define the level l_p of a node p as the path length from the node to the data page. The data pages are at level 0, the leaf nodes of the directory are at level 1 and the root node, denoted

by R is at the highest level l_R.

Let $D^{l,j}$ denote the j^{th} node at level l. Then we define the following notation:

$D^{l,j}.H$ denotes the global depth of node $D^{l,j}$;

$D_i^{l,j}$ denotes the i^{th} element of the node $D^{l,j}$;

$D_i^{l,j}.h$ denotes the local depth in the element $D_i^{l,j}$;

$D_i^{l,j}.p$ denotes the pointer in the element $D_i^{l,j}$. A pointer gives the address of a directory node or a data page at a lower level in the structure.

The BEH-tree directory is constructed in the following manner. Beginning with one node $D^{1,0}$ which is initialized with one entry say, the number of entries in the node doubles at each expansion step, just as in the one-level directory of ExHash, until the size becomes 2^θ, i.e., $D^{1,0}.H = \theta$. Suppose further directory expansion is required. Then instead of doubling the size of the directory to $2^{\theta+1}$ as in ExHash, the node $D^{1,0}$ is split into two nodes $D^{1,0}$ and $D^{1,1}$ each of size 2^θ. A third node $D^{2,0}$ with two entries $D_0^{2,0}$ and $D_1^{2,0}$, is created at level 2. The parameters and entries of node $D^{2,0}$ are initialized as follows: $D^{2,0}.H \leftarrow D_0^{2,0}.h \leftarrow D_1^{2,0}.h \leftarrow 1$, $D_0^{2,0}.p \leftarrow D^{1,0}$, $D_1^{2,0}.p \leftarrow D^{1,0}$. Suppose the initial directory node $D^{1,0}$ is as shown Figure 2. Then the Figure 3 illustrates the effect of splitting the node.

There is one special detail to which we call the readers careful attention. Suppose the splitting of page P_i into pages P_i and P_i' caused the node $D^{1,0}$ to be split as explained previously. Before the split, let $D_i^{1,0}$ be the entry such that $D_i^{1,0}.p = P_i$. Then, after the split the pointers to the pages P_i and P_i' are in consecutive directory entries, both in either $D^{1,0}$ or $D^{1,1}$. Let us assume that these entries are respectively $D_i^{1,0}$ and $D_{i'}^{1,0}$. During the split, the local depths in all the entries of the two nodes $D^{1,0}$ and $D^{1,1}$ are decreased by one except the two entries $D_i^{1,0}$ and $D_{i'}^{1,0}$ that contain the pointers to the two pages P_i and P_i'.

Unlike the ExHash, the value of the local depths in BEH-tree is particularly significant in computing the path from the root to the correct destination page. The BEH-tree structure combines the concept of a number of data organization schemes. In some respect, the scheme resembles a balanced m-ary prefix hash tree [4] and an m-ary radix search tree [8]. However the BEH-tree tree traversal algorithms differ from these. The directory tree is balanced in the sense that the path length of a every data page is the same. This is achieved by node

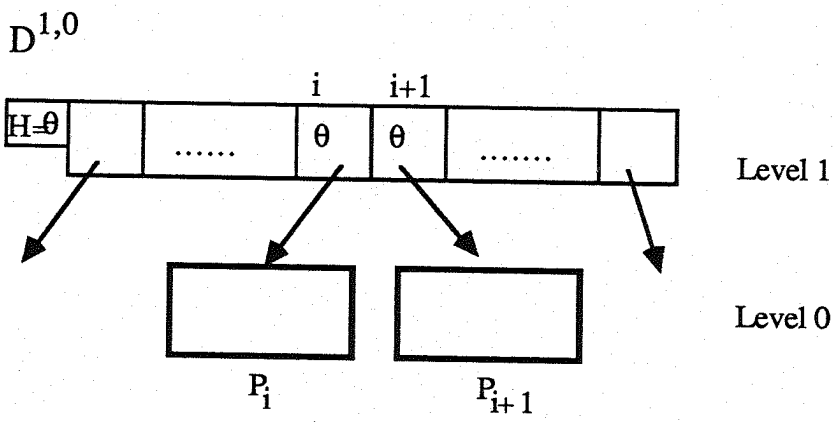


Figure 2: Initial configuration of BEH-tree with one node at level 1

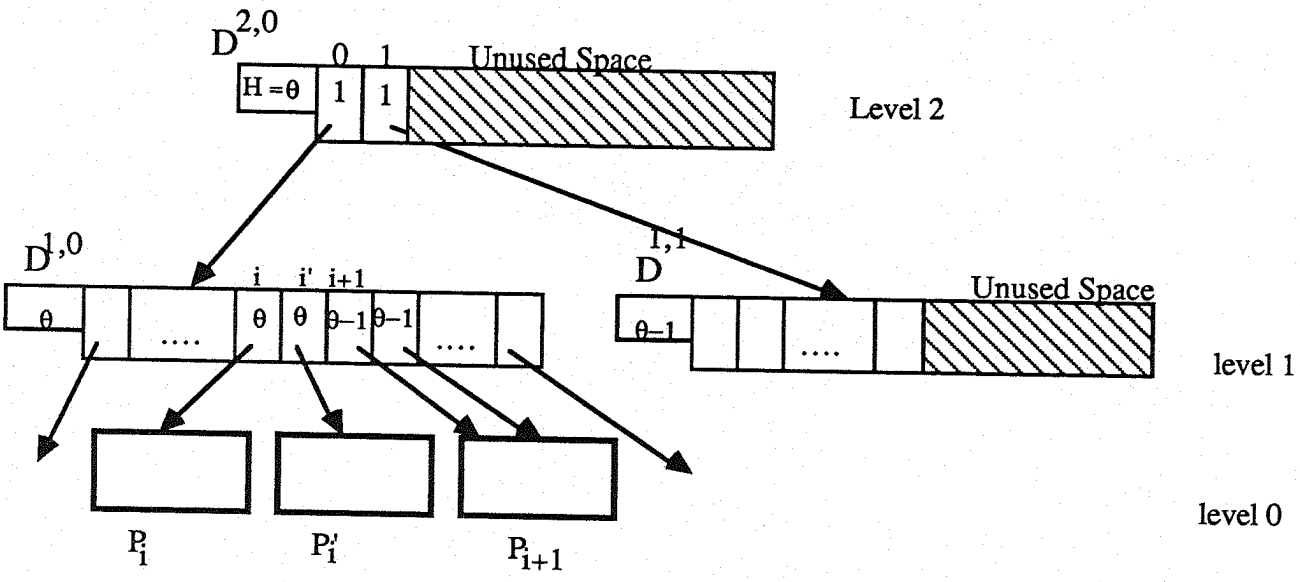


Figure 3: Effect of splitting page P_i when the initial directory node in Figure 2 is full.

splitting in a manner similar to that of B-tree [1], Prefix B-tree [2] and $B^+ - tree$ [8], where node splitting is always initiated from the leaf nodes of the directory and propagated bottom up until possible the root node is split. To facilitate the sequential processing in using the scheme, the terminal nodes at level $l = 1$, are double linked in both the forward and backward directions. Within each page, the stored records are ordererd on their natural keys. The above characteristics leads to a formal definition of a BEH-tree as follows.

Definition

A BEH-tree data organization of order $m = 2^\theta$ and data page capacity $C \geq 1$ is an extendible hashing scheme in which the directory is constructed as a balanced m-ary tree such that

1. all terminal nodes appear on the same level and are linked in key order.
2. every directory node $D^{l,j}$ at level l is headed by a value $D^{l,j}.H$ for $0 \leq H \leq \theta$ and has $k = 2^H$ entries.
3. each entry $D_i^{l,j}$ for $0 \leq i \leq 2^H - 1$ consists of
   - a pointer $D_i^{l,j}.p$ that may not necessarily be unique which points to a directory node or a data page at level $l - 1$, and
   - a local depth $D_i^{l,j}.h$ for $0 \leq h \leq H$.
4. if a node $D^{l,j}$, has a global depth H then for each entry $D_i^{l,j}$ with local depth h there are 2^{H-h} entries of the node with the same values of local depth and page pointers.
5. the terminal nodes of the directory, i.e., nodes at level $l = 1$, contain either data page pointers or are null.

Implicit in the characterization of the BEH-tree is the compression of nodes even when a split occurs. Since after splitting a node, the local depth in each entry of the affected nodes is decreased by one except for two, one will often encounter a situation where every local depth is less than the global depth. In this case the node is immediately compressed by setting the global depth to the maximum local depth in the node and reducing the number of entries accordingly.

3.2 An Example

Given the characterization of the BEH-tree in the preceding section, we can illustrate the construction of the scheme with the 27 keys shown in Table 1. We will assume that the keys correspond to the records and the pseudo-keys are the order preserving encoding of the keys. We take the data page capacity to be 4, i.e., $C = 4$, and set the value of $\theta = 2$. Therefore $m = 2^2 = 4$, and each directory node can contain at most 4 elements.

Table 1: A set of Keys and their Pseudo-Keys

Key	*Pseudo-key*	*Key*	*Pseudo-key*
the	10100 01000 00101	his	01000 01001 10011
of	01111 00110 00000	he	01000 00101 00000
and	00001 01110 00100	be	00010 00101 00000
to	10100 01111 00000	not	01110 01111 10100
a	00001 00000 00000	by	00010 11001 00000
in	01001 01110 00000	but	00010 10101 00000
that	10100 01000 00001	have	01000 00001 10110
is	01001 10011 00000	you	11001 01111 10101
i	01001 00000 00000	which	10111 01000 01001
it	01001 10100 00000	are	00001 10010 00101
for	00110 01111 10010	on	01111 01110 00000
as	00001 10011 00000	or	01111 10010 00000
with	10111 01001 10100	her	01000 00101 10010
was	10111 00001 10011		

Beginning with one directory node $D^{1,0}$ having 2 elements, and 2 pages P_0 and P_1, the key insertions are carried out one after the other until the final configuration shown in Figure 4 is obtained.

To illustrate the significance of the local depth in the address computation, we trace the traversal of the directory nodes for the key "his = $\langle 010000100110011 \rangle$". At the root node $D^{3,0}$, we have $D^{3,0}.H = 1$ hence we use the first bit "0" to derive the index 0. The node element $D_0^{3,0}$ has pointer $D_0^{3,0}.p = D^{2,0}$ and a local depth $D_0^{3,0}.h = 1$. We read in the node $D^{2,0}$ and strip off the first bit "0" in the key to obtain $\langle .10000100110011 \rangle$. The global depth $D^{2,0}.H = 2$. Therefore we use the two bits "10" to derive the index value 2. The pointer $D_2^{2,0}.p$ found is that of another node $D^{1,3}$ and $D_2^{2,0}.h = 2$. We strip off the next two bits "10", leaving the pseudo-key $\langle ...000100110011 \rangle$. At node $D^{1,3}$ we find the gobal depth $D^{1,3}.H = 2$. The index

derived is 0 and the pointer $D_0^{1,3}.p$ is that of a data page P_2. Hence we read in the page P_2 and lookup the key "his" in this page.

Clearly, the BEH-tree is very distinct from the Prefix B-tree [2] and the Digital B-tree [12]. For instance while traversal in Prefix B-tree is done by comparing prefix strings of the keys with substring stored in the nodes, the, traversal in BEH-tree is done by using prefix strings to calculate the index positions in a node. For any page P_i pointed to from an entry $D_i^{1,j}$ say, in a terminal node, the sum of the local depths along the path from the root node to $D_i^{1,j}$ gives the number of prefix bits the keys in the data page P_i agree on. For example, the keys in page P_3 agree on the first 3 bits and those in page P_2 on the first 4 bits.

3.3 A Simple Extendible Hash Tree (EH-Tree)

An immediate question on first encountering the BEH-tree is whether a straight forward implementation of a multilevel directory, in a manner suggested in [6] and [16] would result in a controlled directory growth. We consider such a directory structure here as an alternative scheme to the balanced extendible hash scheme and refer to this as a *Simple Extendible Hash Tree (EH-tree)* .

The EH-tree is essentially m-ary prefix hash tree [4] having page capacity $C \geq 1$, with one essential difference. Each node in the EH-tree is organized as a bounded one level directory of ExHash. That is each node allocated is of fixed size capable of holding 2^θ elements, but the number of elements progressively increase from 2 to 2^θ, where θ is some predefined constant. Unlike the BEH-tree, the extendible hash tree grows from the root towards the leaves. The structure of each node is organized as in the BEH-tree.

To elaborate on the extendible hash tree concept, consider a one level directory $D^{1,0}$ which is initialized as the root node with one entry $D_0^{1,0}$. As insertions are made, the number of entries doubles at each expansion step until there is 2^θ entries. Suppose the next record to be inserted hashes to the entry position i with page pointer P_i, i.e., $D_i^{1,0}.p = P_i$. Assuming that the page is full and that $D_i^{1,0}.h = D^{1,0}.H = \theta$. Then the page P_i must be split. Splitting this page requires that the node size be doubled. This implies that we must now consider the use of $(\theta + 1)$ bits in addressing the directory node.

Since the node is limited to size 2^θ, a new directory node, capable of holding 2^θ elements, is created at level 2 below it. Note that this time we consider the levels to increase downwards

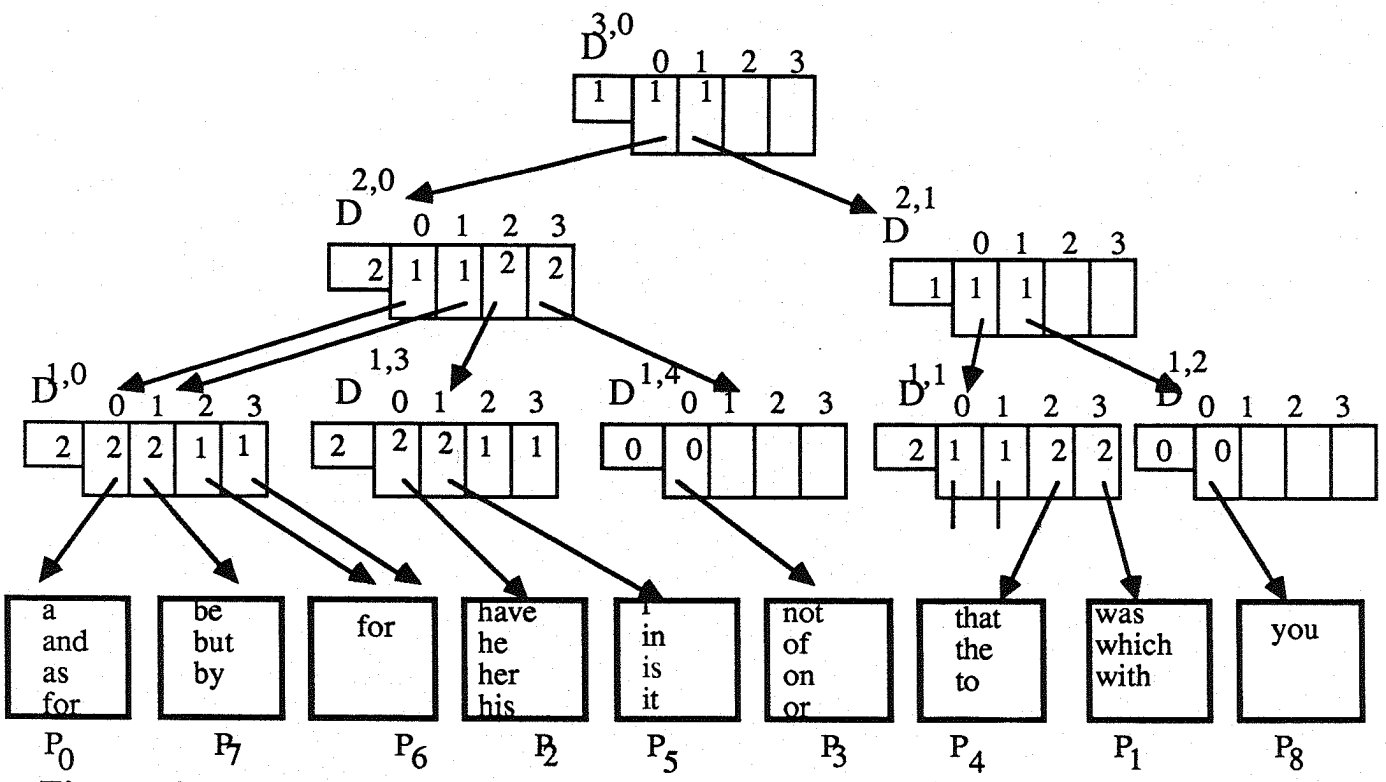
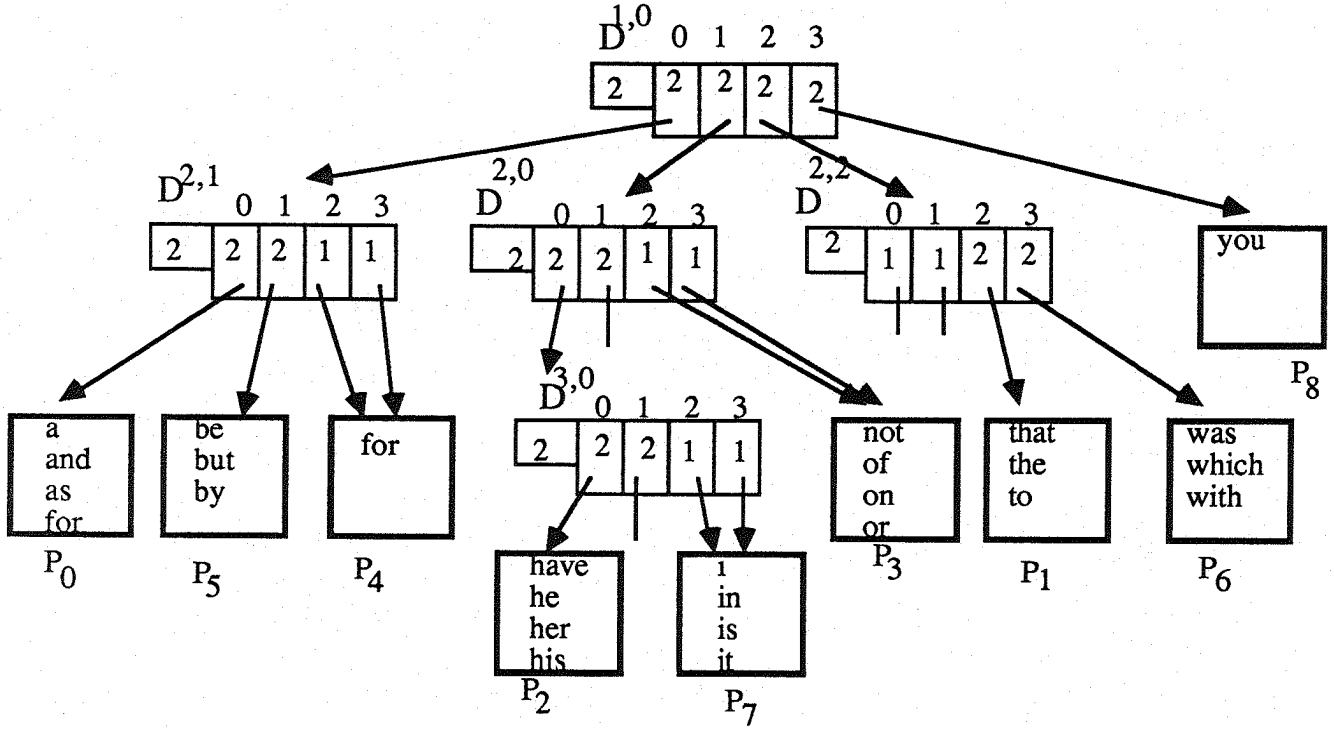


Figure 4 : The final Configuration of the BEH-tree resulting from inserting all the keys in Table 1.

Figure 5: Configuration of a Simple Extendible Hash Tree for the keys in Table 1.



towards the leaves. Let this be denoted by $D^{2,t}$. The node $D^{2,t}$ is initialized to contain two entries and the pointer $D_i^{1,0}.p$ is set to point to it,i.e., $D_i^1.p \leftarrow D^{2,t}$. A new data page P_i' is allocated and the parameters in the new node are set as follows: $D^{2,t}.H \leftarrow D_0^{2,t}.h \leftarrow D_1^{2,t}.h \leftarrow 1$, $D_0^{2,t}.p \leftarrow P_i$ and $D_1^{2,t}.p \leftarrow P_i'$. Essentially the page P_i is replaced by the directory node $D^{2,t}$ and an entry in this node has a page pointer to P_i. The C+1 records that previously got assigned to page P_i are rehashed as follows. The first θ bits of the pseudo-keys are used to hash into the entry $D_i^{2,0}$ in the root level node. On encountering the directory pointer to node $D^{2,t}$, the next bit in the $(\theta+1)$ position is used to hash into either entry $D_0^{2,t}$ or $D_0^{2,t}$ according to whether the bit is 0 or 1. The Figure 5 shows the configuration of an extendible hash tree organization with parameters $\theta = 2$ and $C = 4$ using the keys in Table 1.

Comparing the EH-tree in Figure 5 with the BEH-tree in Figure 4, it may first appear that the EH-tree organization is more conservative of storage than the the BEH-tree. This is generally true for $\theta \leq 2$. When $\theta > 2$, the BEH-tree utilizes less storage for the directory even for uniform distributed keys. To appreciate the effect of θ on the storage utilization, consider the situation when the global depth of the file is θ and the $(\theta + 1)$-bit is being considered in determining the directory address. The Table 2 below summarizes the maximum directory size generated when the address computation uses $\theta + 1$-bits.

Table 2 Directory Sizes when $\theta + 1$ bits are used and the Key Distribution is Uniform

Directory Type and size		θ				
		1	2	3	4	5
ExHash	$2^{\theta+1}$	4	8	16	32	64
BEH-tree	$3 * 2^{\theta}$	6	12	24	48	96
EH-tree	$2^{2\theta}$	4	16	64	256	1024

4 Searching Algorithms

We examine algorithm for accessing records in the BEH-tree scheme given the access paradigm mentioned in section 1. We formally present the insertion algorithm in the next section.

4.1 Random Search

To determine the record corresponding a given key value K, we use the order preserving binary encoded value $K' = \psi(K)$ and traverse the directory nodes to a terminal node where a data page pointer is located. If a *null* pointer is encountered we know the key does not exist. Otherwise the page is read into primary memory and then searched for the record with a matching key value. A sequential or binary search may be used to search within a page. An algorithm for performing a random search for a record of an arbitrary key value is specified as the procedure **Find()** below. The procedure **Find**(K, v, R) takes as parameters, the key K, its encoded key v $= \psi(K)$, a pointer to the root node R, and then reads into memory the page P_i where the record should reside. On output, the pointer R gives the address of the terminal node accessed that contains an element with a page pointer P_i and v is modified. The algorithm makes use of the following data structures and routines:

$\mathcal{G}(K', H)$: Computes the index of a key $K' = \beta_0\beta_1 \ldots \beta_w$ in a node and is defined by

$$i = \sum_{j}^{H} \beta_j 2^{H-j}.$$

Get_Node(p): Reads a node p into memory.

Get_Page(p): Reads a page p into memory.

Shif_Left(v, x): Shifts the number of bits in v, x places to the left.

Algorithm Find(K, v, R);
Get_Node(R) ;
$i \leftarrow \mathcal{G}(v, D^R.H)$;
$Q \leftarrow D_i^R.p$;
while Q is a directory node pointer **do**
Shift_Left (v, $D_i^R.h$);
$R \leftarrow Q$;
Get_Node(R) ;
$i \leftarrow \mathcal{G}(v, D^R.H)$;

```
            Q ← D_i^R.p;
        endwhile
        if Q ≠ null then
          Get_Page(Q);
          Search for record with key K in page Q;
          if found then return ("Found") endif;
        endif
        return ("Not Found");
end Find
```

The complexity of the search algorithm in number of disk accesses is l_R where l_R is the level of the root node in the BEH-tree. We assume that the root node is always retained in memory. The value of l_R is controlled by a designer's choice of the parameters θ and the data page capacity C and for most practical purposes we can choose $l_R \leq 3$. For example suppose we have $w = 32$, so that the pseudo-key encoding $K' \leq 32$ bits in an address space of 2^{32} bytes. Then for C = 64 and $\theta = 9$, we have $l_R < 3$. In general we have the following theorem:

Theorem 4.1 *In a balanced extendible hash tree with nodes of bounded global depth θ, data page capacity C, and pseudo-keys of size w bits, the number of disk accesses required to locate a record is $(w - \lg C)/\theta$.*

Proof

The above Theorem follows trivially from the way the tree is constructed. We assume that the keys are unique. For a page capacity C, the maximum number of bits used in addressing the directory tree when the pseudo-keys are of size w is $w - \lg C$. At most $w - \lg C$ bits are used in the address computation when the directory has its maximum levels l_R, since there will be a path from the root to a terminal node with a local depth of θ in each entry visited. The maximum number of levels in the directory is

$$l_R = \frac{w - \lg C}{\theta}.$$

Since the root node is always memory resident but one extra disk access is made to read in the data page, the number of disk accesses made to locate a record is l_R. □

4.2 Ordered Sequential Search

Given that a record with key K has been retrieved, one normally desires successive retrievals of the records with key values K_s (K_p) where K_s (K_p) is the next higher (lower) key value to the

one previously retrieved. Such data processing operations are typically carried out by routines as FindNext(K), and RangeFind(K_l, K_u). Using the two functions Find() and FindNext(), algorithms for RangeFind() follow immediately.

In BEH-tree, the terminal nodes are linked. But unlike $B^+ - tree$, a terminal node can contain only a null pointer, so that no data page is accessible from the node. See node $D^{1,2}$ of Figure 7a. As a result, a FindNext() operation may have to pass over a number of linked terminal nodes before encountering one with a non-null page pointer. The algorithm for locating the next record having key K_{j+1} following one with a specified key K_j in the natural ordering, is given by the function FindNext(v, R, Q). The function takes on 3 parameters, v, R and Q, where v is the pseudo-key resulting from Shift_Left operations on an initial $v = K'$ used in traversing the BEH-tree from the root node to the terminal node given in a Find() operation. The node D^R has an entry i containing a pointer (possibly null), to the page Q where the key K should reside, i.e., $D^R_i.p = Q$.

Algorithm FindNext(v, R, Q);
  if $R \neq null$ then
    Consider the records in page P to be order as
    $K_1 \leq \ldots \leq K_{j-1} \leq K_j \leq K_{j+1} \leq \ldots \leq K_L$
    where L is the number of records in Q and $K = K_j$.
    if $j \neq L$ **then**
      Return (record $r_{j+1} \in Q$ with key K_{j+1});
    endif;
  endif
  $i \leftarrow \mathcal{G}(v, D^R.H)$;
  while $R \neq null$ **do**
    $H \leftarrow D^R.H$; $h \leftarrow D^R_i.h$;
    while $i < 2^H$ **do**
      $i \leftarrow i - (i \bmod 2^{H-h}) + 2^{H-h}$;
      if $i \leq 2^H$ **then**
        $Q \leftarrow D^R_i.p$;
        if $Q \neq null$ **then**
          Return (record $r_s \in Q$ where $K_s = \min_j(K_j \in Q)$;
        endif;
      endif;
    endwhile;
    $R \leftarrow D^R.Flink$;
    $i \leftarrow 0$;
  endwhile
end FindNext

Let the keys of of records be encoded as w-bit integers, and let the records be grouped into pages of capacity C records. Further, let the records be organized using the balanced extendible hash tree scheme such that each node in the directory tree is bounded by the global depth of θ. Then we say the balanced extendible hash tree has parameters w, θ and C.

Define ξ_K as the sum of the local depth in all the entries visited along the path from the root to the terminal node but not including the local depth in the entry of the terminal node in a Find(K', v, R) execution. We can characterize the complexity of the FindNext() operation by the Theorem below.

Theorem 4.2 *In a balanced extendible hash tree with parametrs w, θ and C, the complexity of determining the successor K_s of K using the algorithm FindNext() immediately following Find() is $O(\xi_K + \xi_{K_s})$ where ξ_K is the sum of the local depths in the entries visited along the path from the root to the node on level 2, if it exists, in a Find() operation.*

Proof

Consider the two encodings of keys $K' = \psi(K) = \langle \beta_0 \beta_1 \ldots \beta_{\xi_K} \ldots \beta_{w-1} \rangle$, and $K'_s = \psi(K_s) = \langle \beta_0 \beta_1 \ldots \beta_{\xi_{K_s}} \ldots \beta_{w-1} \rangle$, where K_x is the next successor to K in the natural ordering. Then after executing Find(K, v, R), we will arrive at the terminal node given by the pointer T_K say, having utilized ξ_K-bits of K. The index i of the node D^{T_K} is computed from $\langle \beta_{\xi_K} \beta_{\xi_K+1} \ldots \beta_{w-1} \rangle$ using the first $D^{T_K}.H$ bits. There are two cases to consider after the entry index i is computed.

Case I: $D_i^{T_K}.p = Q$, i.e., the page pointer is not null.

In this case the page Q is read. If the key K is not the last in the ordering we return the next record after key K. The cost is O(1). If the key occurs as the last then the situation is as in Case II.

Case II $D_i^{T_K}.p = null$. The algorithm locates the next non-null page pointer and returns the record with the minimum key in the indicated page. If a non-null page pointer is not located within the node D^{T_K}, the forward link of the terminal page is followed until a node is found with a non-null page pointer. The cost of the FindNext() operation is $(1 + \nu)$ where ν is the number of terminal nodes accessed to get to a page containing K_s. $\nu - 1$ of these have only null page pointers. Let the terminal node where K_s is found be $D^{T_{K_s}}$. Then from the construction of the BEH-Tree, we can arrive at the $D^{T_{K_s}}$ from D^{T_K}

in the following manner. We take the psuedo-key $v = \langle \beta_0 \beta_1 \ldots \beta_{\xi_K} \ldots \beta_{w-1} \rangle$, set the bits $\beta_{\xi_K} \ldots \beta_{w-1}$ to "00 ...0' and add 2^{ξ_K}, i.e., we set $v \leftarrow v - (v \bmod 2^{w-\xi_K}) + 2^{w-\xi_K}$. This may be used as new key to locate the next terminal node to examine. Each time a terminal node is reached with no non-null page pointer, ξ_K is either decreased or increased, until we generate $K'_s = \psi(K_s) = \langle \beta_0 \beta_1 \ldots \beta_{\xi_{K_s}} 0 \ldots 0 \rangle$. In particular, ξ_K monotonically increases and then decreases. The number of terminal nodes visited is at most equal to the number of bit position $\xi_K + \xi_{K_s}$. But this is also the number of terminal nodes traversed with null page pointers before possible encountering one with non-null pointer while using the forward links of the terminal nodes. This proves the above Theorem. □.

The algorithms for Find(), FindNext() form the basis for implementing other searching strategies. The algorithm for retrieving all records whose keys lie in the interval $[K_l, K_u]$ is given by teh RangeFind() algorithm below. We assume that the order preserving pseudo-keys $K'_l = \psi(K_l)$ and $K'_u = \psi(K_u)$ are generated prior to invoking RangeFind().

Algorithm RangeFind (K_l, K_u, R);
    $v \leftarrow K'_l \leftarrow \psi(K_l)$;
    $K'_u \leftarrow \psi(K_u)$;
    Find(K_l, v, R); {Sets the pointer values R and Q where R is the
        terminal node pointer and Q is the data page pointer.
        R is initially the root. }
    while $K' \leq K'_u$ **do**
        if $Q \neq$ *null* **then**
            for all record $r_i \in Q$ having key K_i such that
            $K'_l \leq \psi(K_i) \leq K'_u$ report record r_i.
        endif;
        FindNext(v, R, Q);
    endwhile
end RangeFind;

Let $D^{T_{K_l}}$ and $D^{T_{K_u}}$ be the terminal nodes arrived at in executing Find(K'_l) and Find(K'_u) respectively. The RangeFind() processes all data pages pointed to from the entries of the terminal nodes that lie between $D^{T_{K_l}}$ and $D^{T_{K_u}}$. The complexity of the range retrieval algorithm is given by the Theorem ??.

Theorem 4.3 *The retrieval of all records with key values in the interval* $[K_l, K_u]$ *is achieved in a balanced extendible hashing scheme with parameters* w, θ *and* C *in at most*

$O(\varsigma_u - \varsigma_l)$ disk accesses, where ς_l and ς_u are the integers formed from the first $w - \lg C$ bits of the pseudo-keys K'_l and K'_u respectively.

Proof

Define ξ'_{K_l} as the sum of the global depth in all the entries visited along the path from the root node to the terminal node inclusive by a Find(K_l, v, R) operation. Let ξ'_{K_u} be defined similarly. Then the maximum entry in the terminal node arrived at using K_l is given by

$$s_1 = \varsigma_l - (\varsigma_l \mod 2^{w-lgC-\xi'_{K_l}}) + (2^{w-lgC-\xi'_{K_l}} - 1)$$

The minimum entry in the terminal node arrived at using K'_u is given by

$$s_2 = \varsigma_u - (\varsigma_u \mod 2^{w-lgC-\xi'_{K_u}})$$

The maximum number of entries accessible between these two limits is $\Delta = (s_2 - s_1)$. In the worst case, each entry has a non-null data page that is accessed. Hence the worst case number of disk access is $2 + \Delta/\theta + \Delta$ which is $O(\Delta)$. But Δ can be expressed as

$$\begin{aligned}
\Delta &= s_2 - s_2 \\
&= \varsigma_u - \varsigma_l \\
&\quad -(\varsigma_u \bmod 2^{w-lgC-\xi'_{K_u}} - \varsigma_l \bmod 2^{w-lgC-\xi'_{K_l}}) \\
&\quad -(2^{w-lgC-\xi'_{K_l}} - 1).
\end{aligned}$$

or

$$\begin{aligned}
\Delta &\leq (\varsigma_u - \varsigma_l) \\
&= O(\varsigma_u - \varsigma_l). \quad \square
\end{aligned}$$

5 Insertion and Deletion Algorithms

The example of section 4 illustrated how the insertion is carried out in a balanced extendible hashing scheme. We present here the details of the insertion algorithm. The algorithm makes use of a global stack and the following functions and procedures. An equivalent full recursive algorithm may be easily programmed in place of the explicit use

of a stack. The directory is initialized first with one node R, which is the root, having parameters $D^R.H = D_0^R.h = 0$ and $D_0^R.p = null$. Other auxiliary data structures and routines are:

STK: A global stack for retaining the nodes visited along path from the root to the terminal node during a search.

Push_Node(STK, p, i): Pushes the node p into stack STK, and retains the index i used in node p.

Pop_Node(STK, p, i): Pops the top node of the stack STK and sets its address to p. The index value i of an entry in P is restored.

Compress(p): Sets the global depth $D^p.H$, to the maximum local depth of the elements in D^p and reorganizes the elements to use space $2^{D^p.H}$ entries only.

Put_Page(p): Writes the data page addressed by p onto secondary storage.

Put_Node(p): Writes the directory node addressed by p onto secondary storage.

Algorithm Insert$(r = \langle K, info \rangle, R)$; { R is initially the root node pointer }
$v \leftarrow \psi(K)$;
$x \leftarrow 0$; {x keeps track of the prefix bits used in hashing }
$STK \leftarrow \phi$; { empty stack STK }
$i \leftarrow \mathcal{G}(v, D_H^R)$; { i is the index for an entry}
while $D_i^R.p$ is a directory node pointer **do**
Shift_Left$(v, D_i^R.h)$;
$x \leftarrow x + D_i^R.h$;
Push_Node(STK, R, i);
$R \leftarrow D_i^R.p$;
$i \leftarrow \mathcal{G}(v, D_H^R)$;
endwhile

$Q \leftarrow D_i^R.p$; { Q is the page pointer found }
if $Q \neq null$ **then**
Allocate a new page Q;
Store r_i in Q ;
$\delta \leftarrow D^R.H - D_i^R.H$;
$i \leftarrow i - (i \bmod 2^\delta)$;
for $j \leftarrow i$ **to** $i + 2^\delta - 1$ **do**
set $D_j^R.p \leftarrow Q$;
endfor;
return;

```
else
  Get_page(Q);
  Search page Q for record r.
  if record r found then
    return(" Error");
  endif
  if page Q is not full then
    Store record r in Q and return;
  endif
  Allocate a new data page Q'.
  Copy content of Q to a temporal area P;
  Set Q ← φ; { empty page Q}
  for each record r ∈ Q having key K do
      if (x+1)-bit of ψ(K) = 0 then
        Store r in Q;
      else
        Store r in Q';
      endif
  endfor
  if Q = φ set Q ← φ endif;
  if Q' = φ set Q' ← φ endif;
  Put_Page (Q); Put_Page(Q');
  DidSplit ← true ;
  while STK is not empty and DidSplit do
        Pop_Node(STK, R, i);
        Let T be a temporal area with the same structure as a node
        but with size = 2^(θ+1) entries.
        Allign R in the left half of T;
        if D_i^T.h = D^T.H then
          Set D^T.H ← D^T.H + 1 and double the
          number of entries in T appropriately.
        endif;
        i ← 2 * i;
        δ ← D^R.H − D_i^R.H;
        i ← i − (i mod 2^δ);
        for j ← i to i + 2^(δ−1) − 1 do
            k ← j + 2^(δ−1);
            set D_j^T.p ← Q;
            set D_k^T.p ← Q';
            set D_j^T.h ← D_k^T.h ← D_j^T.h + 1;
        endfor
        if D^T.H ≤ θ then
          DidSplit ← false;
        else

         Allocate a new directory node R';
```

```
            for j ← i to i + 2^{D^T.H} do
                D_j^T.h ← D_j^T.h − 1;
            endfor;
            D^{R'}.H ← D_j^T.H;
            Copy left half entries of T into R';
            Compress_Node (R'); Put_Node (R');
          endif;
        endwhile
        Put_Node (R);
        if DidSplit then
          Allocate a new directory node R'';
          Set D^{R''}.H ← D_0^{R''}.h ← D_1^{R''}.h ← 1;
          D_0^{R''}.p ← R; D_1^{R''}.p ← R'' ;
          R ← R''; Put_Node(R);
        endif
        Insert (r, R);
    endif
end Insert
```

An equivalent algorithm for deletion similar to the insertion algorithm above is easily derived. Instead of splitting data pages and directory nodes we merge them and free the spaces previously occupied. The insertion and deletion algorithms have the same order of complexity. We discuss the case for insertion only.

Given that the pseudo-keys are of fixed size w-bits say, the balanced directory tree structure imposes a bound on the worst case number node splits that can occur in a record insertion. As a consequence we get a bound on the worst case number of disk accesses for an insertion (or deletion) which is considerable less than in ExHash. We state this in following Theorem.

Theorem 5.1 *In a balanced extendible hash tree with parameters w, θ and C the worst case number of disk accesses made during an insertion or a deletion is $O(\theta l_R^2)$ where $l_R = (w - \lg C)/\theta$ is the level of the root node.*

For the proof see [14]. The worst case increase in the size of the BEH-tree with parameters w, θ and C is $O(\theta l_R^2) * 2^\theta$ entries. Observe that under the same conditions of insertion the directory in ExHash has a worst case increase in size of $O(2^{w-lgC}) = O(2^w/C)$ entries.

In balance extendible hash tree replication of pseudo-keys can be admitted as long as the number of replications is less than C.

6 Some Experimental Results

The performance characteristics of the three methods of implementing the dirtrectory of extendible hashing, ExHash, BEH-tree and EH-tree, vary considerable with respect to the directory sizes depending on the record key distributions. In one extreme, where we have a perfectly uniform distributed keys, we would expect ExHash to give the least directly size. On the other extreme, where the distribution is highly non-uniform, e.g., a spiked distribution, the EH-tree, theoretical should give the minimum directory size.

For any interesting size of a file, which needs to be retained on secondary storage, the fact that the keys uniquely identify the records makes key distributions such as a spiked distribution unrealistic. What is encountered in practice is a key distribution lying between the two extremes. We investigate empirically the growth in the directory size, the search cost, the insertion and deletion cost, for all the three schemes using uniform and non-uniform distributed keys. We use a Gaussian distributed keys to illustrate the case of non-uniform keys.

The simulation experiments highlight the significance of the BEH-tree. The experiments consist of generating 32-bit pseudo-random numbers. In the first case the keys are uniformly distributed between 0 and $2^{31}-1$, i.e., $K \in [0, 2^{31}-1]$. In the second case the keys are drawn from a gaussian distribution $N(4,1)$ and mapped onto the interval $[0, 2^{31}-1]$. For each key distribution, we generate 50,000 keys randomly which are then organized using each of the respective schemes for page sizes of $C = 8, 16, 32$ and 64. For both BEH-tree and EH-tree, the value of θ is set at 7 to encourage a fast growth in the number of levels. The following performance characteristics are measured are compared:

λ : The average number of disk accesses made for a successful search.

λ' : The average number of disk accesses made for an unsuccessful search.

ρ : The average number of disk accesses made to insert a key.

α : The average storage utilization of the data pages. We define α as $N/(n_p * C)$ where N is the number of keys inserted and n_p is the number of pages allocated.

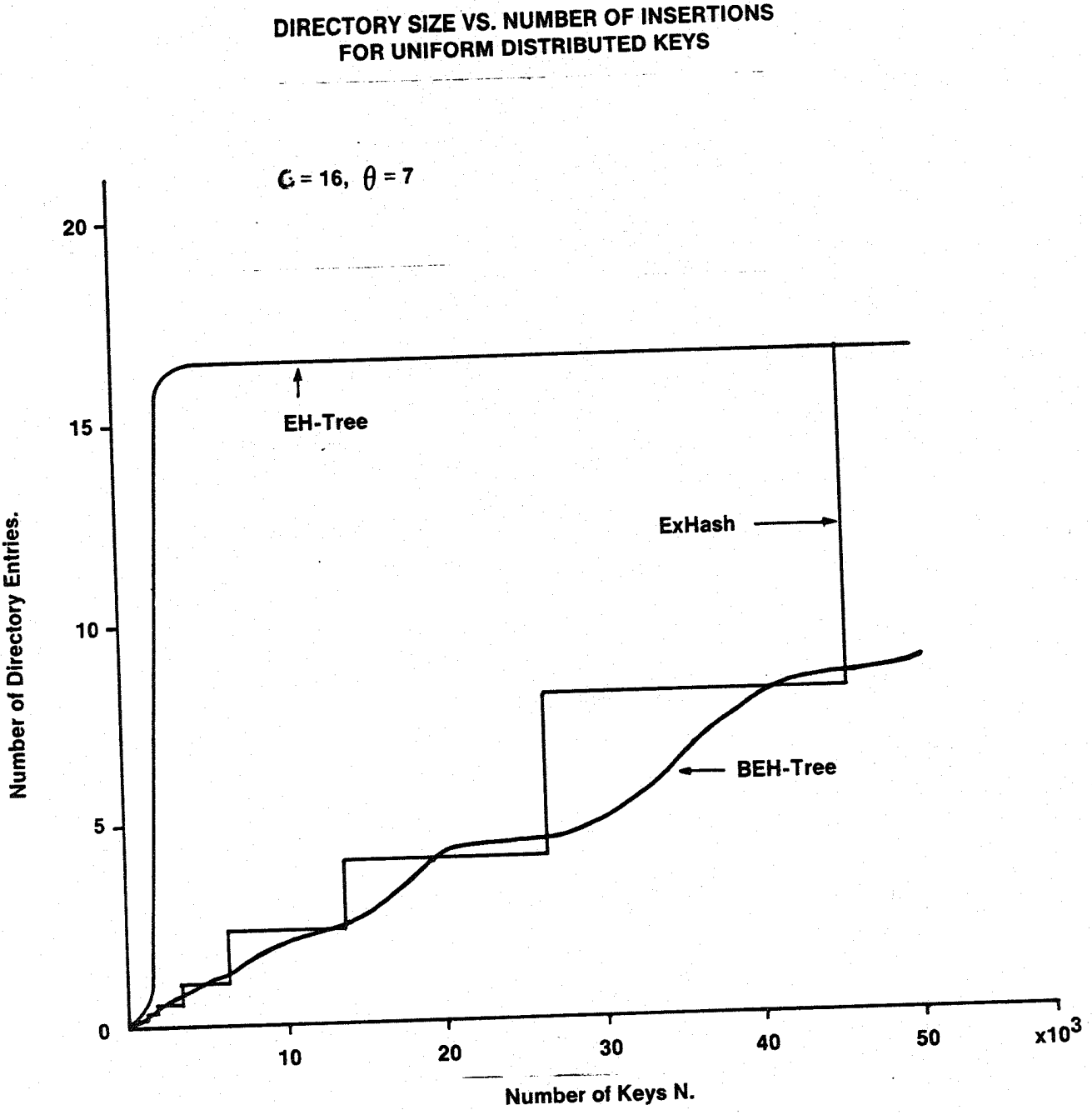
n_d : The directory size, measured by the number of entries.

In each run of the simulation, except for the directory size which is recorded whenever there is a change, the above parameters λ, λ', ρ and α are recorded for the last 2500 keys inserted. Figures 6 and 7 show respectively, the variation of the directory size when the page capacity is 16 for the uniform and gaussian key distributions. The results for varying page capacities are summarized in Tables 3 and 4.

Desirable properties of the scheme are that the parameters λ, λ', ρ and n_d should be as low as possible and that α should be as high as possible. Comparing the three design approaches for extendible hashing, the values of λ and λ' for ExHash are constant at 2. The corresponding values BEH-tree and EH-tree for the chosen parameters vary but are less than or equal to 3. The values for BEH-tree are slightly higher than those of EH-tree for low data page capacities. Recall that a design choice for θ can always be made, given the size of the pseudo-keys, to guarantee no more than 3 or 4 disk accesses. For instance, if we hash into an exceptionally huge address space of 2^{48} bytes with 48 bits, choosing a data page capacity of C=256 and $\theta = 10$, guarantees no more than 4 disk accesses.

The averge cost per insertion in ExHash can be as high as 2 to 3 times the cost of inserting in a EH-tree or BEH-tree, particularly for non-uniform keys at low page sizes. The load factor which is a measure of the storage utilization ranges from 0.67 to 0.74.

The remarkable differences in the three schemes is in the directory sizes required to sustain them. When the keys are uniformly distributed, the EH-tree, uses twice as much storage as ExHash for the directory nodes. The directory size in BEH-tree is comparable to that of ExHash. The BEH-tree directory grows more smoothly than either ExHash or EH-tree. For non-uniform key distribution, the BEH-tree is clearly superior than ExHash or EH-tree. The directory size in BEH-tree is 4 to 8 times less expensize than in ExHash or EH-tree. Comparing the three plots of the growth of the directory sizes in Figures 6 and 7, the near linear growth of the directory size for BEH-tree is clearly demontrated irrespective of the key distribution.


DIRECTORY SIZE VS. NUMBER OF INSERTIONS
FOR UNIFORM DISTRIBUTED KEYS
G = 16, θ = 7
20
15
10
5
0
EH-Tree
ExHash
BEH-Tree
Number of Directory Entries.
10
20
30
40
50
x10^3
Number of Keys N.

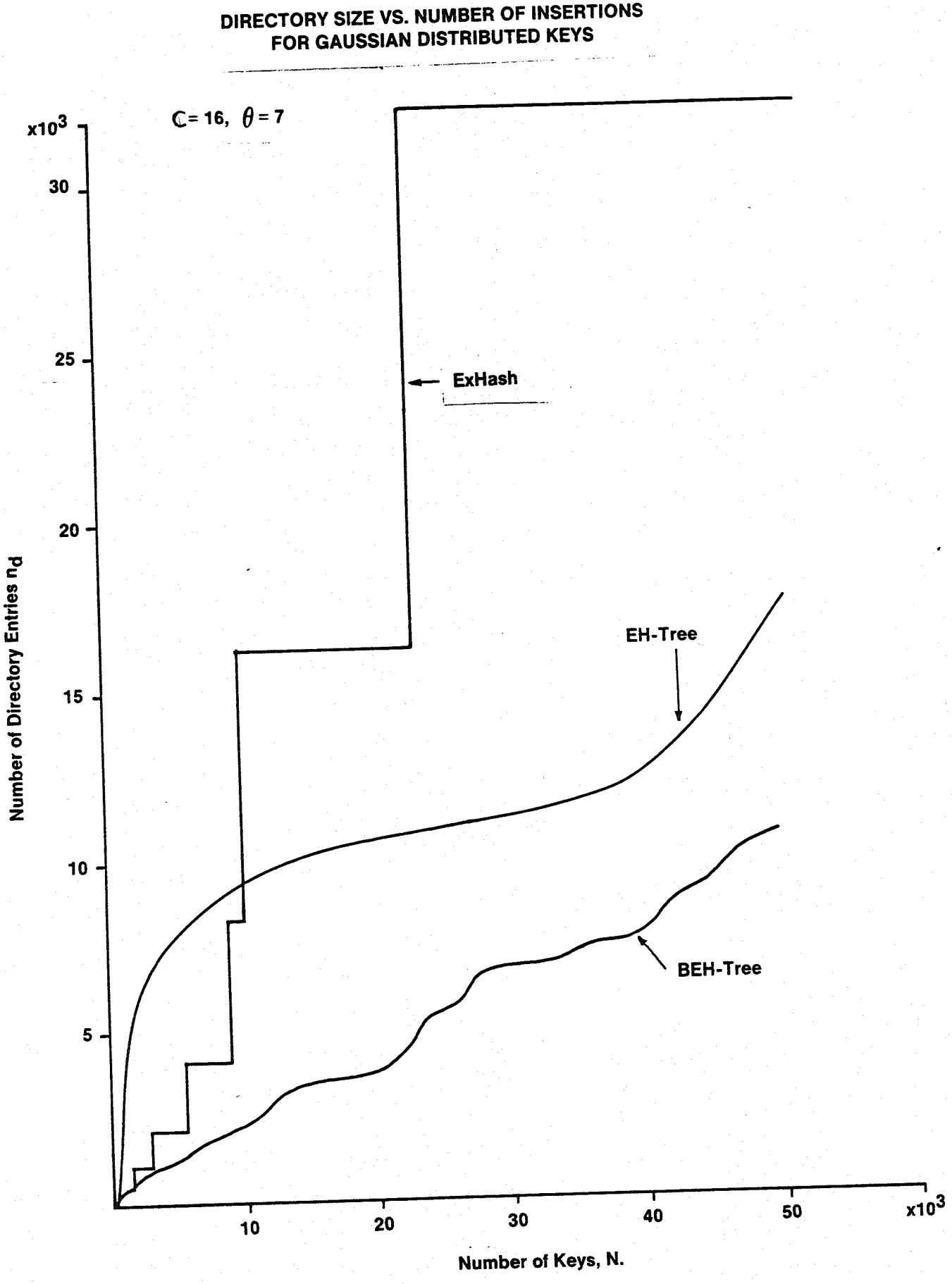

DIRECTORY SIZE VS. NUMBER OF INSERTIONS
FOR GAUSSIAN DISTRIBUTED KEYS
C= 16, θ = 7
x10^3
30
25
20
15
10
5
ExHash
EH-Tree
BEH-Tree
Number of Directory Entries n_d
10
20
30
40
50
x10^3
Number of Keys, N.

Table 3: Summary of Results for Uniform Distributed Keys.

Performance Measure	Directory Design	Page Capacity			
		8	16	32	64
Average No of Disk I/O for a Succ. Search λ	ExHash	2.000	2.000	2.000	2.000
	EH-tree	2.016	2.000	2.000	2.000
	BEH-tree	3.000	2.000	2.000	2.000
Average No of Disk I/O for an Unsucc. Search λ'	ExHash	2.000	2.000	2.000	2.000
	EH-tree	2.000	2.000	2.000	2.000
	BEH-tree	3.000	2.000	2.000	2.000
Average No of Disk I/O for an insertion ρ	Exhash	8.158	5.524	4.071	3.419
	EH-tree	5.031	3.757	3.252	3.059
	BEH-tree	7.104	3.964	3.321	3.059
Average Load factor α	ExHash	0.691	0.702	0.718	0.734
	EH-tree	0.691	0.702	0.718	0.734
	BEH-tree	0.691	0.702	0.718	0.734
Directory Size after 50,000 insertions n_d	ExHash	32,768	16,384	8192	4096
	EH-tree	26,368	16,384	16,384	16,384
	BEH-tree	24,192	8,448	4,352	2,176

Table 4: Summary of Results for Gaussian Distributed Keys.

Performance Measure	Directory Design	Page Capacity			
		8	16	32	64
Average No of Disk I/O for a Succ. Search λ	ExHash	2.000	2.000	2.000	2.000
	EH-tree	2.433	2.012	1.993	1.985
	BEH-tree	3.000	3.000	2.000	2.000
Average No of Disk I/O for an Unsucc. Search λ'	ExHash	2.000	2.000	2.000	2.000
	EH-tree	2.325	2.003	1.993	1.983
	BEH-tree	3.000	3.000	2.000	2.000
Average No of Disk I/O for an insertion ρ	Exhash	14.694	7.404	6.885	5.112
	EH-tree	5.615	3.991	3.573	3.269
	BEH-tree	6.916	5.03	3.744	3.362
Average Load factor α	ExHash	0.693	0.686	0.681	0.673
	EH-tree	0.693	0.686	0.681	0.673
	BEH-tree	0.693	0.686	0.681	0.673
Directory Size after 50,000 insertions n_d	ExHash	131,072	32,768	16,384	4,096
	EH-tree	270,080	17,408	11,008	9.984
	BEH-tree	25,216	10,496	3,712	1,920

7 Conclusion

We have shown an approach for designing a hierarchical directory for an order preseriving extendible hashing scheme without compromising the $O(1)$ record search cost. In comparison with the one level directory design of ExHash, the BEH-tree utilizes less directory storage space and further gives a linear directory growth instead of the staircase like directory growth. The proposed scheme is sufficiently robust in the sense that it maintains a uniform performance characteristic irrespective of the key distribution. This provide a practical alternative data structure to the B^+-tree for maintaining fast dynamic indexed files.

A performance comparison of the B-tree and extendible hashing has been made in the original paper on ExHash [6]. The linkage of the terminal nodes in BEH-tree to allow for fast sequential processing thus making the structure comparable more to a B^+-tree than a B-tree. In contrast to the B^+-tree, the balanced extendible hash tree does not maintain the key values either in whole or in part as part of the entries in the nodes. The key values only assist in tracing the path from the root to the data page. In a B^+-tree a sequential search or binary search in each node arrived at must be made to locate the pointer to a lower level node or data page. No searching in a BEH-tree node is performed. Each index position is computed from the bitstring of the pseudo-key. Since no key values are stored, the BEH-tree allows both fixed length and variable length keys to be handled with one common data structure. The bitstring of any pseudo-key can always be extended by zeroes if necessary.

Acknowledgement

The implementation of the balanced extendible hash tree by Nguyan Thong, to be used as a generalized file indexing scheme, and their subsequent investigation of concurrent manipulation of the scheme is very much appreciated. This work is supported in part by grant No 8102A from the Natural Sciences and Engineering Council of Canada.

References

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

[2] Bayer, R. and Unterauer, K. Prefix B-tree. *ACM Trans. on Database Syst. 2, 1 (1977), 11–26.*

[3] Bechtold, U. and Kuspert, K. On the use of extendible hashing without hashing *Inform. Process. Lett., 19 (1984), 21 – 26.*

[4] Coffman, E. G. and Eve, J. File structure using hashing functions. *Comm. ACM 13, 7 (1970), 427–436.*

[5] Devroye, L. A probabilistic analysis of the height of tries and the complexity of triesort. *Acta Informatica 21 (1984), 229–237.*

[6] Fagin, R., Nievergelt, J., Pippenger, N. and Strong, H. R. Extendible hashing: a fast access method for dynamic files. *ACM Trans. on Database Syst. 4, 3, (1979), 315–344.*

[7] Flajolet, P. On the performance evaluation of extendible hashing and trie searching. *Acta Informatica 20 (1983), 345–369.*

[8] Knuth, D. E. The art of computer programming, vol 3: sorting and searching. *Addison Wesley Publ. Reading, Massachusetts, (1973).*

[9] Larson, P. A. Dynamic hashing. *BIT 18 (1978), 184–200.*

[10] Litwin, W. Linear hashing: a new tool for table and file addressing. *Proc. 6th Int'l. Conf. on Very Large Databases, Montreal (1980), 212–223.*

[11] Litwin, W. Trie hashing. *Proc. ACM SIGMOD Conf., Ann Arbor, Michigan (1981), 19–29.*

[12] Lomet, D. B. Digital B-trees. *Proc. 7th Int'l. Conf. on Very Large Databases, Cannes, France (1981), 333–344.*

[13] Mendelson, H. Analysis of extendible hashing. *IEEE Trans. on Soft. Eng. SE-8, 6 (1982), 611–619.*

[14] Otoo, Ekow J. Balanced extendible hash tree. *Technical Report, School of Computer Science, Caleton University, (June 1987)*

[15] Regnier, M. On the average height of tries in digital searching and dynamic hashing. *Inform. Process. Lett., 13, 3 (1981), 64–66.*

[16] Tamminen, M. Order preserving extendible hashing and bucket tries. *BIT 21 (1981), 419–435.*

Carleton University, School of Computer Science
Bibliography of Technical Reports
Publications List (1985 -->)

School of Computer Science
Carleton University
Ottawa, Ontario, Canada
KIS 5B6

SCS-TR-66 **On the Futility of Arbitrarily Increasing Memory Capabilities of Stochastic Learning Automata**
B.J. Oommen, October 1984. Revised May 1985.

SCS-TR-67 **Heaps in Heaps**
T. Strothotte, J.-R. Sack, November 1984. Revised April 1985.

SCS-TR-68 **Partial Orders and Comparison Problems**
out-of-print M.D. Atkinson, November 1984. See Congressus Numerantium 47 ('86), 77-88

SCS-TR-69 **On the Expected Communication Complexity of Distributed Selection**
N. Santoro, J.B. Sidney, S.J. Sidney, February 1985.

SCS-TR-70 **Features of Fifth Generation Languages: A Panoramic View**
Wilf R. LaLonde, John R. Pugh, March 1985.

SCS-TR-71 **Actra: The Design of an Industrial Fifth Generation Smalltalk System**
David A. Thomas, Wilf R. LaLonde, April 1985.

SCS-TR-72 **Minmaxheaps, Orderstatisticstrees and their Application to the Coursemarks Problem**
M.D. Atkinson, J.-R. Sack, N. Santoro, T. Strothotte, March 1985.

SCS-TR-73 **Designing Communities of Data Types**
Wilf R. LaLonde, May 1985.
Replaced by SCS-TR-108

SCS-TR-74 **Absorbing and Ergodic Discretized Two Action Learning Automata**
out-of-print B. John Oommen, May 1985. See IEEE Trans. on Systems, Man and Cybernetics, March/April 1986, pp. 282-293.

SCS-TR-75 **Optimal Parallel Merging Without Memory Conflicts**
Selim Akl and Nicola Santoro, May 1985

SCS-TR-76 **List Organizing Strategies Using Stochastic Move-to-Front and Stochastic Move-to-Rear Operations**
B. John Oommen, May 1985.

SCS-TR-77 **Linearizing the Directory Growth in Order Preserving Extendible Hashing**
E.J. Otoo, July 1985.

SCS-TR-78 **Improving Semijoin Evaluation in Distributed Query Processing**
E.J. Otoo, N. Santoro, D. Rotem, July 1985.

Carleton University, School of Computer Science
Bibliography of Technical Reports

SCS-TR-79 _____
On the Problem of Translating an Elliptic Object Through a Workspace of Elliptic Obstacles
B.J. Oommen, I. Reichstein, July 1985.

SCS-TR-80 _____
Smalltalk - Discovering the System
W. LaLonde, J. Pugh, D. Thomas, October 1985.

SCS-TR-81 _____
A Learning Automation Solution to the Stochastic Minimum Spanning Circle Problem
B.J. Oommen, October 1985.

SCS-TR-82 _____
Separability of Sets of Polygons
Frank Dehne, Jörg-R. Sack, October 1985.

SCS-TR-83 out-of-print
Extensions of Partial Orders of Bounded Width
M.D. Atkinson and H.W. Chang, November 1985. See Congressus Numerantium, Vol. 52 (May 1986), pp. 21-35.

SCS-TR-84 _____
Deterministic Learning Automata Solutions to the Object Partitioning Problem
B. John Oommen, D.C.Y. Ma, November 1985

SCS-TR-85 out-of-print
Selecting Subsets of the Correct Density
M.D. Atkinson, December 1985. To appear in Congressus Numerantium, Proceedings of the 1986 South-Eastern conference on Graph theory, combinatorics and Computing.

SCS-TR-86 _____
Robot Navigation in Unknown Terrains Using Learned Visiblity Graphs. Part I: The Disjoint Convex Obstacles Case
B. J. Oommen, S.S. Iyengar, S.V.N. Rao, R.L. Kashyap, February 1986

SCS-TR-87 _____
Breaking Symmetry in Synchronous Networks
Greg N. Frederickson, Nicola Santoro, April 1986

SCS-TR-88 _____
Data Structures and Data Types: An Object-Oriented Approach
John R. Pugh, Wilf R. LaLonde and David A. Thomas, April 1986

SCS-TR-89 _____
Ergodic Learning Automata Capable of Incorporating Apriori Information
B. J. Oommen, May 1986

SCS-TR-90 _____
Iterative Decomposition of Digital Systems and Its Applications
Vaclav Dvorak, May 1986.

SCS-TR-91 _____
Actors in a Smalltalk Multiprocessor: A Case for Limited Parallelism
Wilf R. LaLonde, Dave A. Thomas and John R. Pugh, May 1986

SCS-TR-92 _____
ACTRA - A Multitasking/Multiprocessing Smalltalk
David A. Thomas, Wilf R. LaLonde, and John R. Pugh, May 1986

SCS-TR-93 _____
Why Exemplars are Better Than Classes
Wilf R. LaLonde, May 1986

SCS-TR-94 _____
An Exemplar Based Smalltalk
Wilf R. LaLonde, Dave A. Thomas and John R. Pugh, May 1986

SCS-TR-95 _____
Recognition of Noisy Subsequences Using Constrained Edit Distances
B. John Oommen, June 1986

Carleton University, School of Computer Science
Bibliography of Technical Reports

SCS-TR-96 **Guessing Games and Distributed Computations in Synchronous Networks**
J. van Leeuwen, N. Santoro, J. Urrutia and S. Zaks, June 1986.

SCS-TR-97 **Bit vs. Time Tradeoffs for Distributed Elections in Synchronous Rings**
M. Overmars and N. Santoro, June 1986.

SCS-TR-98 **Reduction Techniques for Distributed Selection**
N. Santoro and E. Suen, June 1986.

SCS-TR-99 **A Note on Lower Bounds for Min-Max Heaps**
A. Hasham and J.-R. Sack, June 1986.

SCS-TR-100 **Sums of Lexicographically Ordered Sets**
M.D. Atkinson, A. Negro, and N. Santoro, May 1987.

SCS-TR-102 **Computing on a Systolic Screen: Hulls, Contours, and Applications**
F. Dehne, J.-R. Sack and N. Santoro, October 1986.

SCS-TR-103 **Stochastic Automata Solutions to the Object Partitioning Problem**
B.J. Oommen and D.C.Y. Ma, November 1986.

SCS-TR-104 **Parallel Computational Geometry and Clustering Methods**
F. Dehne, December 1986.

SCS-TR-105 **On Adding *Constraint Accumulation* to Prolog**
Wilf R. LaLonde, January 1987.

SCS-TR-107 **On the Problem of Multiple Mobile Robots Cluttering a Workspace**
B. J. Oommen and I. Reichstein, January 1987.

SCS-TR-108 **Designing Families of Data Types Using Exemplars**
Wilf R. LaLonde, February 1987.

SCS-TR-109 **From Rings to Complete Graphs - Θ(n logn) to Θ(n) Distributed Leader Election**
Hagit Attiya, Nicola Santoro and Shmuel Zaks, March 1987.

SCS-TR-110 **A Transputer Based Adaptable Pipeline**
Anirban Basu, March 1987.

SCS-TR-111 **Impact of Prediction Accuracy on the Performance of a Pipeline Computer**
Anirban Basu, March 1987.

SCS-TR-112 **ε-Optimal Discretized Linear Reward-Penalty Learning Automata**
B.J. Oommen and J.P.R. Christensen, May 1987.

SCS-TR-113 **Angle Orders, Regular n-gon Orders and the Crossing Number of a Partial Order**
N. Santoro and J. Urrutia, June 1987.

SCS-TR-115 **Time is Not a Healer: Impossibility of Distributed Agreement in Synchronous Systems with Random Omissions**
N. Santoro, June 1987.