Red-Black Balanced Trie Hashing

Otoo, E. J. and **Effah, S.**
School of Computer Science
Carleton University
Ottawa, Ontario
Canada, K1S 5B6
email otoo@scs.carleton.ca
email effah@scs.carleton.ca

February 23, 1995

Abstract

Trie hashing is a scheme, proposed by Litwin, for indexing records with very long alphanumeric keys. The records are grouped into buckets of capacity b records per bucket and maintained on secondary storage. To retrieve a record, the memory resident trie is traversed from the root to a leaf node where the address of the target bucket is found. Using the address found, the data bucket is read into memory and searched to determine the presence or absence of the record. The scheme, for all practical purposes, locates a record in one or two disk accesses. Unlike a trie, the scheme suffers from: i) potential degeneracy when the keys inserted are ordered, ii) expensive reconstruction cost if a system failure occurs during a session. We present a new approach to implementing *Trie Hashing* that resolves the problem of potential degeneracy. Our approach combines the basic *trie hashing* algorithm with the balancing techniques of the *Red-Black Binary Search Tree,* to produce a relatively balanced trie hashing scheme. As a result we ensure that the trie is of height $O(log\ n_p)$ where n_p is the number of buckets and we achieve an average data storage utilization of 67% that is reminiscent of a bucket splitting storage organization. Our method improves considerably upon the performance of the trie hashing scheme.

1 Introduction

We consider the organization of a file F, of N records such that *exact match, range, prefix-key*, and other queries can be processed efficiently. Each record r_i, $0 \leq i \leq N$, consists of a key, and some related information, i.e., $r_i = \langle k_i,\ info_i \rangle$. In our case the keys are assumed to be long alphanumeric characters and the file is highly volatile. A file is said to be volatile if it is subjected to frequent insertions and deletions of records. A file organization scheme is said to be *dynamic* if it tolerates frequent insertions and deletions without performance degradation. Given a file consisting of records with long alphanumeric keys, an exact-match query requests the retrieval of a record that matches exactly a given key K. A range query requests all records that lie in some specified key range $[K_l, K_u]$, and a prefix-key query requests the retrieval of all records whose key prefixes match some specified string. The generalization of the prefix query is one where the specified key value has a sequence of "don't care" substrings either as a prefix, substring or suffix. The prefix query is considered in the class of queries addressed in this paper.

The above problem, which can rightly be stated as an implementation of a dynamic order preserving storage scheme for long alphanumeric keys, is readily achieved by the *Prefix-B-tree* of Bayer and Unterauer [3]. An alternative scheme for implementing dynamic bucket hashing with keys of long alphabetic strings, has been proposed by Litwin [13]. The method, which he refers to as *Trie Hashing,* dynamically constructs a *weak order preserving* hashing scheme. Let Σ is a finite set of collated sequence of alphabets and let Σ^* denote the set of strings formed by the alphabets in Σ. Then trie hashing organizes the storage of records $r_i = \langle K_i, Info_i \rangle$ where $K_i \in \Sigma^*$. In the sequel, Σ is taken to be the standard ASCII character set.

The trie hashing scheme involves two levels of addressing: a binary trie T, that is resident in primary memory, and a set of data buckets $B = \{B_0, B_1, \ldots, B_{n-1}\}$, that is resident on secondary storage. The buckets hold the actual records each of which consists of the key and any related information such as the rest of the record information or a pointer (bucket address) to the location where the entire record is stored. We assume only the storage of keys in our subsequent discussions. Let "$\leq_\Sigma$" denote the precedence relation under the collating sequence of the alphabets. Then the buckets are *weak order preserving* in the sense that for any two buckets B_i, B_j, such that the keys $K_i \in B_i$ and $K_j \in B_j$, we have $K_i <_\Sigma K_j$ $\forall K_i \in B_i$ and $\forall K_j \in B_j$.

The retrieval of a record with key K_i, requires only one access to secondary storage. This is because the address of the bucket to search is determined at the leaf node of the memory resident binary trie when it is traversed in a manner yet to be specified. The traversal of the trie in memory is considered as a computation of a hash function, hence the name *Trie Hashing.* For any interval of keys $[K_l, K_u]$ let $leaf_{K_l}$ and $leaf_{K_u}$ be the terminal nodes reached by traversing the trie T with K_l and K_u respectively. The bucket addresses of the leaf nodes obtained by traversing the trie in inorder from $leaf_{K_l}$ to $leaf_{K_u}$ give the buckets that contain records in the response of the range query.

Other dynamic hashing schemes that achieve single disk access, but with different trie organizations, have been proposed. The dynamic hashing method of Larson [12] uses a forest of binary digital trees while the method of Coffman and Eve [5] retains a digital trie in primary memory called a *Prefix Tree.* These two methods avoid the problem of degeneracy by first randomizing the bits of each hash key. Randomization has the advantage that, for n buckets, a trie of height $O(\lg n)$[1] is constructed. Unfortunately, we loose the order preserving property of the file organization.

As a dynamic hashing scheme, trie hashing is comparable to a number of hashing methods with tree organized indexes that have been proposed,[12, 21]. The *PATRICIA* index [19, 11], is another trie organization that uses the distinct alphabets of the key to guide its search. It is particularly efficient for searching on long alphanumeric keys. However, since only selected alphabets of the keys may be checked during the search, it is inappropriate for

[1] $\lg n = \log_2 n$

ordered retrieval on highly volatile files when the bucket capacity $b > 1$. Patricia index can still be applied to organize *static files* that satisfy the requirement of efficient exact-match, range and prefix queries (see [9, 16, 17, 23]). In comparing trie hashing with other storage schemes, for organizing ordered dynamic files with long alphabetic keys, the Prefix B-tree [3] appears to be the most competitive alternative. Simulation study to compare the trie hashing scheme with Prefix-B-Tree and a number of different storage schemes, in particular dynamic variant of Patricia [18] is still a subject of our future work.

Trie hashing has been shown to give good storage utilization for random key insertions, [13, 22]. However, the scheme, has two major drawbacks.

1. Insertions of a pre-sorted sequence of keys give a highly degenerate trie with extremely long average external path length.
2. After system failure with loss of volatile memory, the trie must be rebuilt from the original data. For a large database this is often unacceptable. Hence one must cope with organizing the trie on secondary storage for persistence.

Since trie hashing was proposed, a number of improvements have been proposed to address the problems of degeneracy, improved storage utilization and persistence [8, 14, 20, 22]. We present yet another variation of the trie-hashing scheme that avoids the problem of degeneracy of the trie. This new method combines the *Red-Black Tree* balancing algorithm of a binary search tree with the trie hashing construction algorithm, to obtain a guaranteed or O(log n) main memory traversal of the trie. We refer to the new storage scheme as a *Red-Black Balanced Trie Hashing.* The red-black tree algorithms are discussed thoroughly in [2, 7, 10].

Some of the drawbacks in the original proposal of trie hashing have been addressed by Torenvliet and Van Emde Boas [22]. They showed that the binary trie constructed in Litwin algorithm can be transformed into an equivalent formal trie and then subsequently reconstruct it as a weight balanced binary trie. Our approach is different from that of Torenvliet and Van Emde Boas in that we construct a locally height balanced binary trie on the fly. First we eliminate single paths i.e., sequence of internal nodes with only single non nil pointers in the Litwin's basic trie. We call this a *Compact Trie Hashing scheme (CTH).* This is then balanced by applying the red-black tree balancing techniques to connected subtries having the same digit number. The explanation of a digit number is given in the next section.

Another problem concerns what happens when the trie exceeds the capacity of primary memory. Litwin, Zegour and Levy address this problem in [15], The solution proposed organizes the trie into multilevel blocks on secondary storage. In this paper we consider only the case when the trie is retained entirely in memory. However the essential ideas reported in this paper are easily extended to paged secondary storage representation of the trie using techniques proposed in [24, 4]. An elegant technique for paged memory organization is presented also in [8].

The rest of the paper is organized as follows. We introduce the basic concepts of trie hashing in the next section. In section 3, we present the details of constructing a corresponding compact binary trie. The development of the *red-black balanced trie* is presented in section 4. An illustrative examples of the scheme is also presented. In section 5 we discuss the results of an empirical study of the scheme, compared with compact trie hashing and present some further details for practical implementation of the algorithms. We present some concluding remarks and direction for future work in section 6.

2 The basic trie hashing scheme

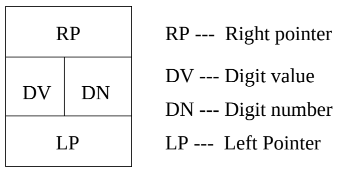
Trie hashing, unlike other hashing schemes, stores records in weak order preserving manner. It is an access method for maintaining records of dynamic files. Searching for a record requires only one disk access, since the trie itself is memory resident. The key/record pairs are kept in buckets on secondary storage. If records are inserted randomly, the storage utilization of the buckets is O(lg 2). The structure grows by bucket splitting and the insertions of branch and terminal nodes in the trie. The trie structure is organized as a binary tree. An internal node, P, of the trie consists of the following defined fields:

DN(P): A digit number. This specifies the position at which the digit value found at node P must be concatenated to a comparator string u. The comparator u is formed as the tree is traversed from root to a leaf node.

DV(P): The digit value found at node P;

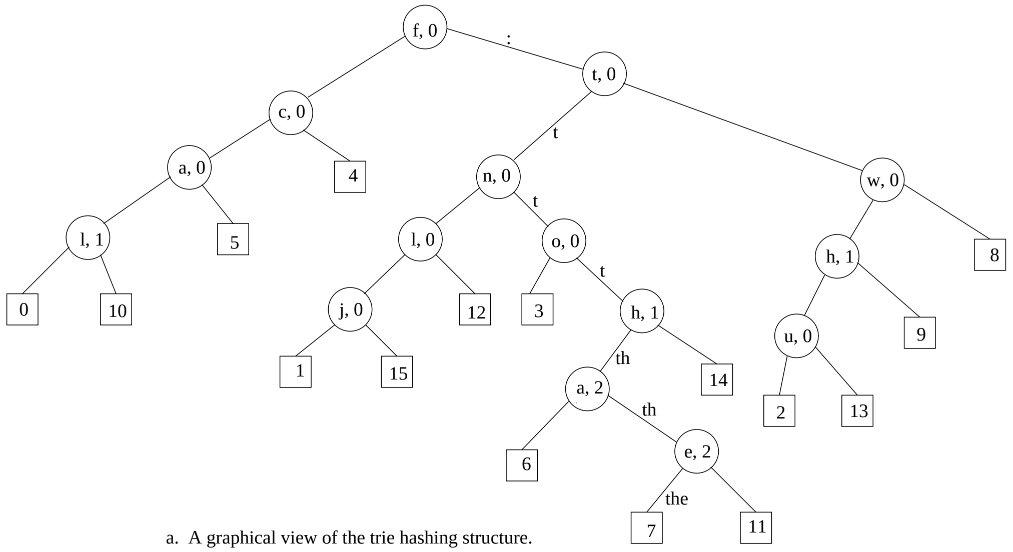
LP(P), RP(P): The respective left and right children pointers of node P.

An internal node of a trie structure is depicted in the following figure.



The pair (DV, DN) is sometimes referred to as the digit field, DF. An internal node has its pointers pointing to other nodes, while an external node has no digit field, but stores the address of a disk resident bucket. A bucket has a capacity, b, $b > 1$. Figure 1 is an example of a trie, in which Figure 1b is the data of the first 46 most frequently used English words which have three or more letters (ordered by frequency). The italicized words are the words whose insertions caused collision (or bucket overflow). Figure 1c is the buckets' contents after the insertions of the data in Figure 1b in which the bucket capacity is set at $b = 4$. The trie structure is shown in part (a). Detailed, search, insertion and deletion algorithms of the structure are presented subsequently.

If the trie is empty, then only one bucket may be allocated. Assuming that the trie is not empty then to search for a key, K, within the trie, the procedure is as follows. Two strings, u and v, initially set empty, are maintained. The string u, is termed the *comparator*. It is important to note that at any node P, the key K is compared with the comparator string u that constructs a logical path to P. If the key K is less than or equal to the comparator at P, the left pointer is followed otherwise the right pointer is followed. The logical path u_p of any node P, is defined recursively as follows:



a. A graphical view of the trie hashing structure.

the, and, for, that, *you,* this, *with,* not, *have,* are, from, can, but, will, *all,* was, there, *one, they, what, would,* any, which, *about,* get, your, use, some, *then,* name, *like,* out, *when,* time, other, more, only, just, end, also, *know,* how, new, should, been, than.

b. Key insertion order.

about all also	get have how just	use	one only other out	for from end	been but can	should some that than	the then there they	you your	will with would	and any are	this	more name new not	was what when which	time	know like
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

c. The buckets after the 46 keys have been inserted.

Figure 1: An example of a trie hashing structure

Definition 2.1

If P is the root, $u_p = \$$, where '\$' is terminal digit larger than any other digit of the alphabets. Otherwise

let f be the father of P,

if P is the right child of f then $u_p = u_f$

else $u_p = (u_f)_{DN(f)-1}DV(f)$.

The quantity $(u_f)_{DN(f)-1}$ is empty if $DN(f) - 1$ is negative. The logical path from root to bucket 7 is illustrated in Figure 1a by the letters on the edges of the trie. Consider the search, where the node P is the current node reached. The comparator string formed, $u = u_0u_1 \cdots u_{DN(P)-1}DV(P)$ where $u_0u_1 \cdots u_{DN(P)-1}$ is a prefix of the logical path to P. At P u is is compared with a substring of the key K, given by $K' = k_0k_1 \cdots k_{DN(P)}$. If u, the comparator, is less than K', in the collating sequence, then the right pointer of P is followed, otherwise the left pointer of P is followed. Whenever the right pointer is followed, the string u computed at P is discarded and u is set back to logical path of the parent node. When a left pointer is followed, the logical path at P is retained. This is done by updating the value of v.

The search terminates if an terminal node, or a nil pointer is encountered. If the external node is not nil, the bucket whose address is stored at that node is read into memory and examined to see if K, the search key is present. The search algorithm is presented formally in algorithm of Figure 2.

```
A search algorithm for trie hashing
Begin
    Let the trie be rooted at $T$.
    Let $K$ be the search key.
    $u = v = \phi$;                                   Initialize u and v as empty strings.
    $P = T$;
    while $P$ is an internal node and $P \neq nil$
        if $DN(P) > 0$, then
            $u = u_0 u_1 \cdots u_{DN(P)-1} DV(P)$;     Concatenate $u^2$ with DV(P).
        else $u = DV(P)$;                               Copy DV(P) into u.
        $K' = k_0 k_1 \cdots k_{DN(P)}$;                 Compute $K'$.
        if $u < K'$ then $P = RP(P); u = v$;             Move right, discard u at P.
            else $P = LP(P); v = u$;                     Move left, retain u.
    endwhile;
    if $P = null$
        return message for unsuccessful search;         No disk access in this case.
    else return bucket address;
End
```

Figure 2: The search algorithm of the trie hashing.

The structure of the search algorithm guarantees that either successful or an unsuccessful search takes one disk access. Suppose we are searching for the key "*when*" in the example trie in Figure 1. We would start at the root of the trie where $DN(f,0) = 0$, therefore $\mathbf{u} =$ "f" and $K' =$ "w". Since "f" $<$ "w", we follow the right pointer and P, the current node is now $(t,0)$, $\mathbf{u} = \mathbf{v} = \phi$. At $(t,0)$, $DN = 0$ and we have $\mathbf{u} =$ "t" and $K' =$ "w". Comparing "u" and "K"' suggests that we follow the right pointer to $P = (w,0)$. The comparator $\mathbf{u}$ is set to the value of $\mathbf{v}$ which is still empty. At $(w,0), DN = 0$, $\mathbf{u} =$ "w" and $K' =$ "w" giving $\mathbf{u} = K'$, so the left pointer is taken and $\mathbf{v}$ is set to the value of $\mathbf{u}$ = "w". At $(h,1), DN > 0$, and so $\mathbf{u} =$ "wh", $K' =$ "wh". Once again $\mathbf{u} = K' =$ "wh" and the left pointer is followed to node $(u,0)$ where $DN = 0$, and the variable $\mathbf{u}$ equals the constant "u", $K' =$ "w". Since $\mathbf{u} < K'$, the right pointer is followed. The node now visited is an external node, holding the address of bucket 13. Bucket 13 is then retrieved from disk and examined in memory for the key "*when*", which is found. The search in this case is successful.

[2] u truncated at the DN(P)th digit.

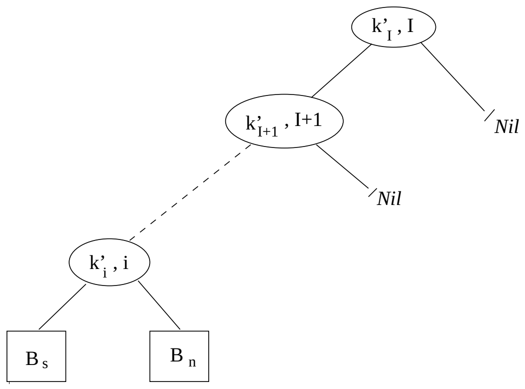


Figure 3: An illustration of how an insertion is accomplished in the trie hashing.

2.1 Insertion

To insert a key, K into the trie structure, a search must first be performed. Let the capacity of a bucket be b. On arriving at an external node P, the bucket whose address is in P, say B_s, is read into memory. Assuming that the search fails. Then if there are less than b keys in B_s, K is inserted into B_s at its lexicographic position and the insertion terminates. However, if B_s contains exactly b keys, i.e. the bucket is full, a collision is said to have occurred. Bucket B_s is split into two buckets.

A split key will be chosen among the $b+1$ keys. The split key is a prefix of one of the $b+1$ keys, which is usually referred to as the *middle key*. The procedure involved in choosing the split key is described in the next section. Assuming $K' = k'_0 k'_2 \cdots k'_{i-1}$, is chosen as the split key, K' is used as a search argument in the search algorithm. At node P, where the search terminates the second time around, if u, the comparator string matches the first I digits of K', then $i - I + 1$ nodes are created such that, $LP(P_j) = P_{j+1}$, $RP(P_j) = nil$, $DV(P_j) = k'_j$ and $DN(P_j) = j$, for $j = I, I+1, \ldots, i-1$. Node P is replaced by nodes $P_I, P_{I+1}, \ldots, P_{i-1}$. The node P_i will have two external nodes as its children. Let these external nodes be P_l and P_r, then $LP(P_i) = P_l$, and $RP(P_i) = P_r$. Also, the bucket addresses B_s and B_n are stored in P_l and P_r respectively, where B_n is a newly allocated bucket. All keys ($\approx (b+1)/2$ of them), that are less than or equal to K' are stored in bucket B_s and the rest are stored in B_n.

Figure 3 illustrates the splitting procedure described above. Suppose during the search to insert a key we follow a pointer that leads to a *nil* node, we allocate a new bucket and create a new node to hold the bucket's address. The key to be inserted is then inserted into the new bucket. Figure 4 shows an example of the insertion procedure. Assuming the bucket capacity $b = 3$, then bucket $b2$ in part (a) is full. If we attempt to insert the key "*there*", at node P, we examine bucket $b2$ which is full, so we compute $K' =$ "*ther*". Performing a search with "*ther*", we arrive at $P = b2$ with $u =$ "*th*". At this point we create two internal nodes. Assuming a new bucket $b5$ is allocated, then the resulting trie is shown in part (b) of Figure 4. If the next key to be inserted into the trie (in Figure 4b) is "*third*", the search fails on a pointer to a *nil* node. In this situation, a new bucket $b6$ is allocated to hold key "*third*" and an external node is created in place of the *nil* node to hold the address of bucket $b6$. The resulting buckets of the file are shown in part (c) of the Figure 4.

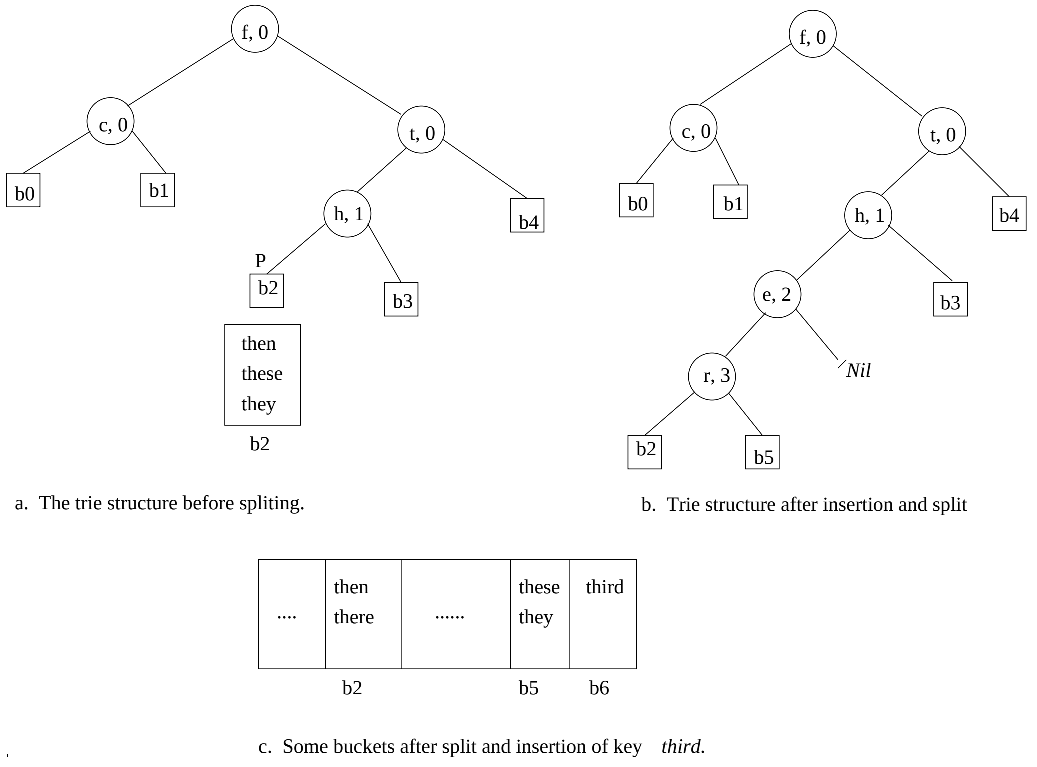

f, 0
c, 0
t, 0
b0
b1
h, 1
b4
P
b2
b3
then
these
they
b2
a. The trie structure before spliting.
f, 0
c, 0
t, 0
b0
b1
h, 1
b4
e, 2
b3
r, 3
Nil
b2
b5
b. Trie structure after insertion and split
....
then
there
......
these
they
third
b2
b5
b6
c. Some buckets after split and insertion of key third.

Figure 4: An example of trie hashing insertion.

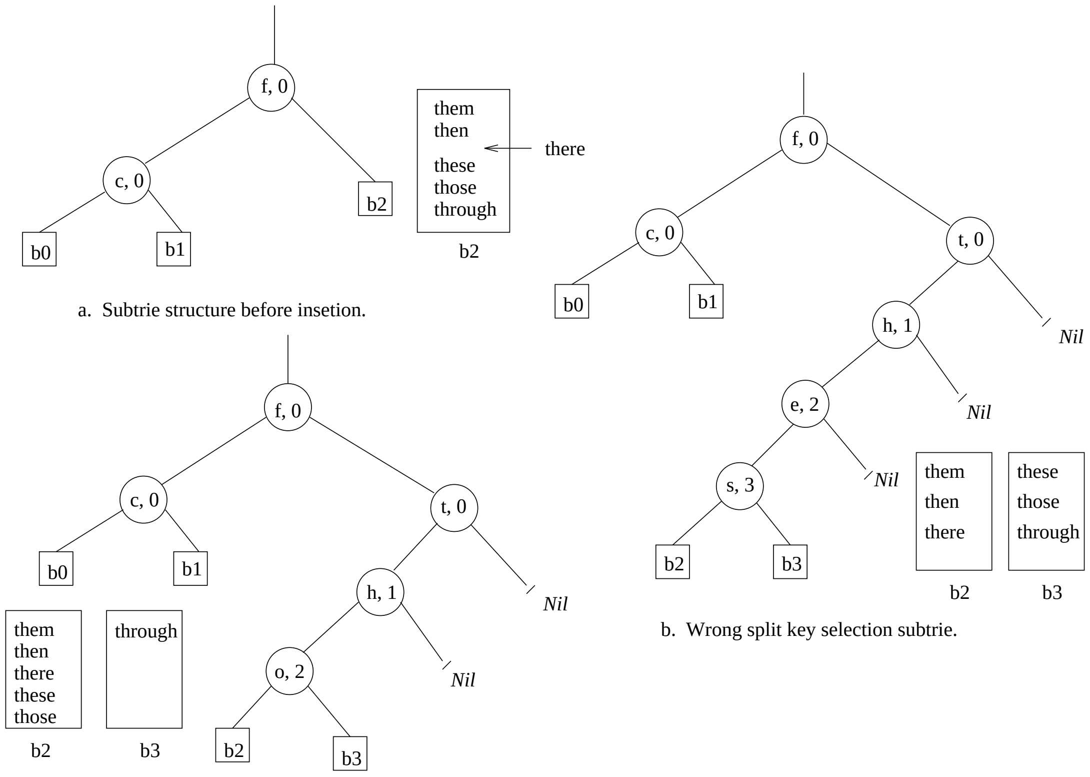
2.2 The concept of a split key

Storage utilization is of concern in file structures. Naturally we desire a split of a bucket into two equal sizes when an overflow occurs to guarantee a storage utilization of 50% (as in the case of B-trees). However, the trie structure may not behave well when the split key is chosen as the prefix of the middle key of the $b + 1$ keys.

An example best illustrates this and is crucial to the correct behavior of the algorithm. In Figure 5 we take the capacity b, of a bucket as 5. On inserting the 6^{th} key "*there*" into the bucket in part (a), we encounter an overflow. If we choose "*thes*" (the prefix of the middle key "*these*") as the split key the resulting trie is shown in Figure 5b. Suppose we now search for the key "*those*". We will arrive at the node $P_l = (e, 2)$, with a comparator string $u =$ "*the*" $<$ "*tho*" $= k'$. Therefore, the right pointer is followed, and $RP(P_l) = nil$. The search is unsuccessful, and the choice of split key leads to a wrongly constructed trie. This problem was first pointed out in [20]. The split key may be formally defined as follows:

Definition 2.2 *In a set of b ordered keys K_i, $0 \leq i \leq b-1$, such that each is terminated be a blank symbol "$\sqcup$", the middle key $K_j = k'_0 k'_1 \ldots k'_{l_j-1}$, $j < b-1$, is one such that if the next key in the ordered sequence is $K_{j+1} = k_0 k_1 \cdots k_{l_{j+1}-1}$, then for some $l' \leq min(l_j, l_{j+1})$ we have:*

1. *$k'_s = k_s$ for $s < l'$, and $k_{l'} \neq k'_{l'}$.*
2. *l' is minimum over all j satisfying (1) above.*
3. *$|\lfloor b/2 \rfloor - j|$ is minimum for all j satisfying (1) and (2) above.*



a. Subtrie structure before insetion.

b. Wrong split key selection subtrie.

c. Correct split key selection subtrie.

Figure 5: An example of how the split key is selected in trie hashing.

In our example, the above definition gives the split key as the prefix of the key "*those*" which is "*tho*". The correct splitting of bucket $b2$ in Figure 5a is shown in Figure 5c. The above definition forces the split key to be chosen such that it is close to the middle key as much as possible. It is obvious then that buckets splitting in the basic trie hashing do not guarantee a minimum storage utilization of 50%.

2.3 Deletion

Deletion from the trie hashing structure is simple. To delete a key, the search algorithm is first applied, if the key to be deleted is found, it is simply removed from the bucket. Let the external node holding the address of the bucket from which the deletion occurred be say P, and let the sibling of P be Q. If after deletion, both P and Q are external nodes and have a total of less than b keys. We merge buckets given by P and Q to form the left external node. If a single path results this is removed systematically up the trie until a a node is encountered that can maintain two external non-nil pointers.

3 Variants of trie hashing

3.1 Compact binary trie

In [20], Otoo introduced a scheme called the *compact binary trie* that eliminates the nil pointers that appear in the basic trie structures. Compact binary tries enable the actual median keys to be selected as a separator in the event of a bucket split. This automatically guarantees a load factor of at least 50%. Furthermore the data buckets are double linked to facilitate processing of both sequential and range queries. In the compact binary trie, the entire separator s, obtained after trimming the logical path to node P, is stored as the digit value. The digit number of P specifies the position of the comparator at which the concatenation of the digit value of P is done.

The implementation of compact binary trie does not require variable length digit value fields in the memory resident trie structure. A fixed length pointer is stored in place of the digit value. This then points to the memory address of the digit value string. An example of the compact trie is shown in Figure 7. This is constructed from the KWIC index of page 13. The resulting buckets of the trie are the same as those shown in Figure 10. In the compact trie, if the comparator u matches the first I digits of the split key $K' = k'_0k'_1 \ldots k'_{i-1}$, at node P where the split key is to be inserted, then the value DV(P) $= k'_Ik'_{I+1}k'_{I+2} \cdots k'_{i-1}$ and DN(P) $= I$.

4 rbTrie

To understand the ideas embodied in *red-black balanced trie hashing,* we consider first a companion structure called *balanced compact trie hashing.* The balancing method in this scheme makes use of the AVL height [1], balancing methods.

4.1 Balanced compact trie hashing

Balanced compact trie hashing (BCTH) is a compact trie that is locally balanced with respect to connected subtrie of nodes that have the same digit numbers. This follows from some observations of the preceding structure. Let us formally define the following concepts.

Definition 4.1 *The level number of a node P with respect to a digit number κ of P in a subtrie rooted at T, is the path length from the node T to the node P, where all the nodes along the path have the same digit number κ.*

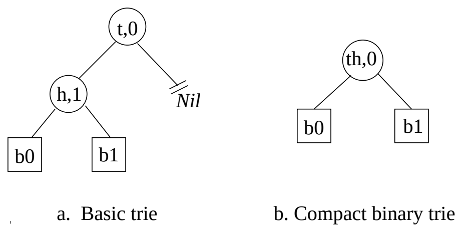


Figure 6: Difference between the basic trie and the compact binary trie during a split.

Definition 4.2 *The height of a node P with respect to digit number κ, in a binary trie is the largest level number achieved by a node of the trie that terminates a path of the same digit number and is denoted by $h(P, \kappa)$.*

Definition 4.3 *Let P be a node of a connected subtrie T_s having a common digit number κ. We say a binary trie is locally height balanced if $\forall P \in T_s$ such that $DN(P) = \kappa$, we have $|h_R(P, \kappa) - h_L(P, \kappa)| \leq 1$ where $h_R(P, \kappa)$ and $h_L(P, \kappa)$ are the heights of the right and left subtries with respect to the common digit number κ.*

The balancing strategy is the same as in the AVL-trees except that single and double rotations (see [1, 7, 23]), are carried out only on nodes with common digit numbers. When a new node P is inserted, and its digit number is less than the digit number of the parent node, the BCTH algorithm 'pushes' the node up the trie. The node is said to migrate towards the root until p becomes the son of a node Q where, DN(q) $\leq$ DN(P). If the migration causes nodes in the trie with common digit number to be unbalanced, the BCTH balancing scheme is invoked to balance the corresponding subtrie using left and right rotations as in the AVL-trees. The migration in the trie affects only the right spine of the trie.

Definition 4.4 *The left (right) spine of a binary trie rooted at R is the maximal path traversed, starting from R, by following the left(right) pointers only.*

The entire trie is said to be locally height balanced if for each subtrie T_i having a common digit number, and for every node $p \in T_i$ we have $|h_l(p) - h_r(p)| \leq 1$, where $h(P)$ is the height of a trie rooted at P, $h_l(p)$ and $h_r(p)$ are the heights of the left and right subtries of node p respectively. Figure 8 illustrates how a migration in BCTH is performed. In both parts (a) and (b), the node with value R migrates by one step. Migration guarantees that along any path from the root to a leaf node, the digit numbers occur in a non-decreasing order. It is worthwhile to note that the *RL-migration* is equivalent to *left rotation* in the red-black tree and the *LR-migration* is equivalent to *right rotation* in the red-black tree.

If the left (right) spine of the trie has non-decreasing digit numbers, then LR-migration (RL-migration) will not be required. Migration and rotation of nodes maintain the sequence order and leave the trie hashing function invariant. Thus the basic trie search algorithm still applies to the BCTH structure. This is formally stated as follows.

Proposition 4.1 *Let P be a node in the right spine of a compact binary trie T, such that P is not a root node and it is not in a non-decreasing sequence order of the digit numbers. Then the RL-migration of the node always restores the sequence order and maintains the trie hashing function invariant.*

The proof of the above proposition is given in [20].

4.2 Red-Black Balanced Tries

The close similarity between the AVL-tree and the red-black-tree [2, 10, 7, 23], and the similarity in the migrations and red-black tree rotations, motivated the work to investigate the use of red-black tree balancing technique to the compact trie hashing algorithm. The integration of the red-black-tree balancing algorithms and the compact trie hashing is the main result advocated in this paper. This presents a better solution to the problem of trie hashing degeneracy.

4.2.1 Red-black tree

The balancing algorithm in an AVL-trees may require as many as $\Theta(\log_2 n)$ rotations after an insertion is made to maintain the height balance. The red-black tree balancing technique performs constant number of rotations after an insertion.

Briefly, the red-black tree colors each node in the binary search tree either red or black, such that the following properties are satisfied.

1. Every node is either red or black.
2. Every extension or *nil* node is black.
3. If a node is red, then both its children are black.
4. Every simple path from a node to a descendant leaf contains the same number of black nodes.

Definition 4.5 *The* black-height *of a node p, bh(p), is the number of black nodes along the path from (but not including) node p to a leaf. The black-height of a red-black tree is the black height of its root.*

The red-black tree has a bit field in a node to store the color, which is either red (0) or black (1). When a new node is inserted into the red-black tree, it is colored red. The algorithm then proceeds to rebalance the tree by recoloring and performing rotations to ensure that the tree after the insertion conforms to the red-black properties enumerated above. The red-black tree is balanced if property 4 above is satisfied. Red-black tries or rbTries result from applying the balancing strategy of the red-black trees to the compact trie hashing structure. In rbTrie we will require, the following modifications to the compact trie.

1. In addition to the content of a node in the basic trie, the rbTrie will have an additional field for the node's color. For simplicity in stating the algorithm (although this is not required in a practice), we also add backward pointer from a node to its father node.
2. The red-black condition of black height will be modified as follows:
    - Every simple path from a node to a descendant leaf contains **approximately** the same number of black nodes.
    - every connected subtrie, having the same digit number, is black height balanced

Apart from condition 4 of the red-black trees which is modified above, all the other conditions apply to the rbTrie. The trie is perceived as formed from connected sub-tries such that each sub-trie has the same digit number. For example a leaf node of a sub-trie with digit number 0 becomes the root node of a subtrie with digit number 1. Some leaf node of the subtries with digit number 1 is the root node of a sub-trie with say digit number 2. By maintaining that the root node of every sub-trie be black, we ensure that each sub-trie adheres strictly to the properties of the red-black tree.

Rotations in rbTrie are used in two ways. A rotation either forces a node with with a smaller digit number than its parent's to move up the trie or balances a connected subtrie with the same digit number. Rotation during balancing, of a subtrie structure may not involve nodes with different digit numbers. In particular, a rotation should not change the role of a node x and its parent $f(x)$ if $DN(x) > DN(p(x))$. If this *trie rotation condition* is not adhered to, the trie structure will be modified and the search algorithm will fail to locate some keys.

4.3 Insertion

When constructed, the rbTrie uses the same search algorithm as the basic trie. Similarly, to insert a record into the rbTrie, we first apply the search algorithm of the basic trie to locate the bucket in which the key must reside. The algorithm of Figure 9 is the insertion algorithm for the rbTrie. We adopt the notations for the red-black tree algorithm in [7] Note the difference in the notation for $left(P)$ and $right(P)$ of a node P in place of $LP(P)$ and $RP(P)$ respectively. To insert a key, the basic trie hashing insertion algorithm is used. It then proceeds to color the inserted internal node red (the external nodes, which contains the bucket addresses are always colored black). After the coloring, the algorithm of figure 9 proceeds to examine which of the conditions of the rbTrie are violated. If any of the conditions are violated, the algorithm fixes it.

Conditions 1 and 2 certainly continue to hold since every node is colored and the external nodes are all black. Modified condition 4 also holds since the newly inserted node takes the place of an external node which was black. Since we color the newly inserted internal node red, and its external nodes black, the number of black nodes on the path is unaffected.

Following an insertion, the only condition that is likely to be violated in the red-black tree properties is condition 3, which says that a red node cannot have a red child (or a red father). The *while* loop in the algorithm of Figure 9 iterates to move the violation of condition 3 up the trie while ensuring that condition 4 is not violated in the process. Rotations are not allowed to take place in the rbTrie if they will cause the trie structure to fail. So if condition 3 is violated, original condition 4 is marginally modified. While condition 3 is violated, if rotations will affect the trie structure, the nodes are recolored.

4.4 Deletion

In most cases when the item to be deleted is located in a bucket, it is simply removed. Let P be the external node pointing to the bucket B_P and let Q be its sibling node. If the bucket from which an item is deleted becomes less than half the capacity b, and Q is an external node, then an attempt is made to merge its content with the bucket given by Q in the rbTrie. If this is possible then the two buckets are merged into one, say B_P. The father node of P is deleted and the pointer of the grand father of node P made to point to P.

On the other hand if the sibling node Q is an internal node then then the contents of the bucket may be merged with the predecessor bucket or the successor bucket according to whether Q is left or right sibling, respectively, of P. The algorithm is slightly more complicated in this situation and involves, possible path compression and node recoloring. We do not pursue this further. The interested reader may consult [8]. A complete rbTrie is presented in Figure 10, the black-height of the trie is 2. The figure is constructed from the KWIC (Key-Word-In-Content) index that appeared in [11, 20]. The KWIC index in

1. part	13. methods	25. problems
2. their	14. as	26. from
3. solve	15. obtaining	27. note
4. for	16. a	28. and
5. to	17. its	29. solution
6. an	18. solutions	30. which
7. some	19. use	31. with
8. certain	20. notes	32. method
9. new	21. by	33. computations
10. equation	22. computation	34. the
11. problems	23. in	35. equation
12. on	24. of	

Table 1: The KWIC index of size 35.

the order of insertion is presented in Table 1. The bucket layout of Figure 10 is the same generated in the illustration of the trie of Figure 7.

5 Performance Evaluation and Practical Considerations

This section compares the performance parameters such as height, maximum path length, size of separators and data page utilization of the rbtrie data structure developed against their counterparts such as the compact trie. Some observations are also drawn from rbTrie and the other trie balancing strategies discussed in earlier sections. Our sample data is the set of words formed from the characters of the English alphabets (both lower and upper cases). In particular, the keys that are used in our experiments are drawn from a dictionary D, consisting of about 50000 words. All the algorithms are implemented in C on the Sun SparcStation.

5.1 Comparison of rbTrie with compact tries

The rbTrie is a locally balanced (compact) trie hashing structure using the technique for balancing in the red-black tree. The trie rotation condition however inhibits the rbTrie to be completely balanced unlike its red-black binary tree counterpart. In ascending ordered insertion, the rbTrie yields a trie that is red-black tree balanced. In our experiment performed, the maximum difference of the black heights in the rbTrie was 4 (see Section 5.2). That is, there is a path from the root to an external node that has 4 more black nodes than another. The load factor of the data pages in the rbTrie, like the basic trie hashing is about 70% for random order insertion and 50% for ascending order insertion.

Compact trie was chosen for the experiments because of its good performance over the basic Litwin trie, and since rbTrie improves upon compact trie, it is appropriate to compare these two schemes. Various sizes of words were drawn from the key set D and inserted both in random order and in ascending order. In addition, the buckets size b, of the rbTrie and compact trie files were set at 10 and 20 to see the impact of the size of b on the performance of the rbTrie and compact trie.

#Keys (x1000)	Average height CT	rbT	Maximum CT	rbT
1	8.60	8.50	12	10
2	9.53	9.67	13	12
5	10.90	11.22	13	14
10	12.17	12.62	16	16
15	12.76	13.65	16	19
20	15.08	13.92	25	19
25	14.28	14.17	26	19
30	15.06	14.65	27	20

a. Random Insertion, utilization = 70%

#Keys (x1000)	Average height CT	rbT	Maximum CT	rbT
1	87.56	8.78	172	13
2	169.63	9.73	338	15
5	426.25	11.25	851	18
10	846.92	12.25	1694	20
15	1271.10	12.95	2540	21
20	1697.49	13.25	3386	22
25	2117.36	13.62	4233	23
30	2544.69	13.96	5082	23

b. Sorted Insertion, utilization = 59%

Table 2: Compact trie and rbTrie comparison for $b = 10$.

#Keys (x 1000)	Average height CT	rbT	Maximum CT	rbT
1	7.34	7.28	9	8
2	8.31	8.32	10	9
5	9.67	9.86	12	12
10	10.69	10.96	13	13
15	11.44	11.60	14	15
20	13.12	12.47	21	16
25	12.77	12.35	23	15
30	13.26	12.57	22	16

a. Random Insertion, utilization = 70%

#Keys (x 1000)	Average height CT	rbT	Maximum CT	rbT
1	47.06	7.76	91	11
2	92.59	8.79	183	13
5	231.87	10.24	459	16
10	459.61	11.24	917	18
15	690.77	11.80	1377	19
20	918.40	12.24	1833	20
25	1155.20	12.87	2299	21
30	1378.88	12.80	2756	21

b. Sorted Insertion, utilization = 55%

Table 3: Compact trie and rbTrie comparison for $b = 20$.

The split key in each case (for $b = 10$ and $b = 20$) was selected to vary from the middle key by 1. That is, 3 separators, about the middle of the keys that are involved in the splitting, are computed during each split, the shortest of which is chosen as the separator. Because the same split behavior is adapted for both the rbTrie and compact trie, their bucket utilizations compare equally. It is typical for key values chosen randomly to have a utilization factor of 70% and storage utilization of 50% [6]. This storage utilization holds true for any block splitting storage organization.

Other parameters measured are the longest and average external path lengths of the structures. The path lengths are calculated by traversing the trie for each key inserted. The maximum path length is the longest path from the root to any external node, and the average path length is the average of all external path lengths. Let λ_i be an external path in a trie corresponding to key k_i, and N the number of keys inserted into the pTrie, then the average path length $\hat{\lambda} = \frac{1}{N}\sum_1^N \lambda_i$.

The results from our experimental runs are shown in Tables 2 and 3. The graphs representing the tables are shown in Figures 11 to 15. Note that the scale of the y-axis of the compact trie and rbTrie for sorted keys insertion differ by a factor of 100.

As can be seen in the cases $b = 10$ and $b = 20$, compact trie has a slight advantage over rbTrie in random insertion for a small file size (about $N < 17000$). As N increases, the height of both rbTrie and compact trie increases marginally with that of compact trie exceeding rbTrie. Therefore for large files, rbTrie's search time for random insertion is better than that of compact trie. In the case of ascending ordered insertion, considering rbTrie, the performance matches that of the random insertion. The average case of smaller buckets (see Table 2) appears outperforms its random counterpart sometimes. This is not the case for ordered insertions in the compact trie. The height of the compact trie (both external maximum and average path lengths) deteriorate drastically as the number of keys inserted increases, the increase averages linearly about $800/c$ per 1000 records (approximately 84 per 1000 records for $b = 10$ and 46 per 1000 records for $b = 20$). Ascending order insertion of rbTrie therefore outperforms that of the compact trie by a factor of 10 to 100.

5.2 Black height of rbTrie

As shown in Table 4, ascending ordered insertion generates a trie that is perfectly red-black balanced where the original condition 4 of red-black holds, i.e., every simple path from a node to a descendant leaf contains the same number of black nodes. As can be seen in the table, there is no difference between the minimum and maximum black height of the trie in the case of ascending ordered insertion.

The case of random insertion produces variable black heights for some nodes in the rbTrie. The heights of the rbTries as shown in Tables 2 and 3 compare favorably with that of compact trie and hence the basic Litwin trie. In our experiments the largest difference between the maximum and minimum black heights is 4, which occurred for $N = 20000$.

5.3 Adopting the rbTrie in Applications

There three major concerns that need be addressed in when adopting the scheme in practice.

1. The first relates to technique to handle the trie in limited memory;
2. The second relates to maintaining the trie on durable storage;
3. The third relates to fast approach to construct an index for either a very large database containing very long variable length keys or a text database.

#Keys (x 1K)	Black height Max	Min	Maximum Path	Deviation (Max-Min)
1	4	4	8	0
2	4	3	9	1
5	6	4	12	2
10	7	5	13	2
15	7	4	15	3
20	9	5	16	4
25	8	5	15	3
30	8	5	16	3

a. Random Insertion.

#Keys (x 1K)	Black height Max	Min	Maximum Path
1	5	5	11
2	6	6	13
5	8	8	16
10	9	9	18
15	9	9	19
20	10	10	20
25	10	10	21
30	10	10	20

b. Ascending Order Insertion.

Table 4: The height of rbTrie, for $b = 20$.

Consider a file of N records grouped in buckets of capacity b. The number of buckets is $n = N/(b\ lg\ 2)$. If a node utilizes 14 bytes, then the size of memory required is $14(2n-1)$ bytes. Assuming $n = 10^6$. Then the memory required is about 28 Megabytes. For a page size of 2 Kilobytes, this implies that 28 megabytes of index is able to address about 2 Gigabytes of data. Under constraint memory, we should adopt a page binary tree [11, 24, 4, 15]. In this case, the index nodes are blocked into pages of about 4 to 8 Kilobytes. A node pointer, rather than being 4 bytes, may now occupy 2 bytes since they reference nodes in the same page. Using 8 Kilobytes per node block and a memory capacity that can hold at least 3 index node blocks at a time, we can address 2 Gigabytes of data with at most 2 disk access instead of one. We assume here that the root node will always remain in memory during a session.

The trie can also be constructed by initial sorts of the data that are then grouped into buckets. The separating key strings of each bucket are then used to construct the trie bottom up rather than by repeated insertions.

6 Conclusion

The trie hashing scheme proposed by Litwin gives another alternative to the design of a dynamic hashing scheme. In particular it proposes a data structure comparable to the PATRICIA and Prefix-B-Tree for indexing very large databases of long variable length keys and also for indexing large text databases. Unlike PATRICIA, trie hashing is capable of efficient ordered dynamic access when keys are grouped in buckets of capacity $b > 1$. PATRICIA index allows ordered organization only in situations where $b = 1$ and in the case of $b > 1$, this can be achieved with preprocessing.

We have shown how some of the drawbacks of the original trie hashing have been addressed in [20, 22, 15]. The rbTrie proposes a much improved and elegant method. The empirical results show that for random insertions, the rbTrie scheme is much better and guarantees an O(log N) trie structure in memory. The rbTrie trie gives a factor of 100 improvement in the average external path lengths. In the absence of any knowledge of the order of key arrivals, it is worthwhile to incorporate balancing in trie hashing.

In comparing the rbTrie with BCTH, we find that the rbTrie takes a constant number of rotations to balance the trie after an insertion is made. BCTH takes an average of $\Theta(\log_2 n_t)$ rotations to balance a subtrie with n_t nodes. Some future works on the use of trie hashing include the use of a simple parallel processor for constructing and manipulating the rbTrie structure and using the trie hashing scheme to index very large text and other non structured databases.

Acknowledgment

This work is supported in part by grants from the Natural Sciences and Engineering Research Council of Canada.

References

[1] G. M. Adel'son-Vel'skii and E. M. Landis. An algorithm for organization of information. *Soviet.Math. Dokl.*, 3:1259 – 1263, 1963.

[2] R. Bayer. Symmetric binary b-trees: Data structure and maintenance algorithm. *Acta Informatica*, 1:290 – 306, 1972.

[3] R. Bayer and K. Unterauer. Prefix b-trees. *ACM Trans. Database Syst.*, 2(1):11 –26, Mar. 1977.

[4] F. Cesarini and G. Soda. Biinary trees paging. *Inform. Syst.*, 7(4):337 – 344, 1982.

[5] E. G. Coffman and J. Eve. File structures using hashing function. *Comm. of ACM*, 13(7):427 – 436, Jul. 1970.

[6] D. Comer. Heuristics for trie index minimization. *ACM Trans. on Database Syst.*, 4(3):383 – 395, Sept. 1979.

[7] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, Mass., 1989. *.

[8] S. Effah. ptrie: Paginated trie hashing. Master's thesis, School of Computer Science, Carleton University, Aug. 1993.

[9] W. B. Frakes and R. Baeza-Yates, editors. *Information Retrieval: Data Structures and Algorithms.* Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[10] L. J. Guibas and R. Sedgewick. A diochromatic framework for balanced trees. In *Proc. 19th Annual Symp. on Foundations of Comp. Sc.*, pages 8 – 21, IEEE Comp. Soc., 1978.

[11] D. Knuth. *The art of computer programming, Vol 3,: Sorting and searching.*, volume 3. Addison-Wesley, Reading, Mass., 1 edition, 1973.

[12] P. Larson. Dynamic hashing. *BIT*, 18:184 – 201, 1978.

[13] W. Litwin. Trie hashing. In *Proc. ACM SIGMOD Conf. Ann Arbor, Michigan*, pages 19 – 29., 1981.

[14] W. Litwin, N. Roussopoulous, G. Levy, and W. Hong. Trie hashing with controlled load. *IEEE Trans. on Soft. Eng.*, 17(7), Jul. 1991.

[15] W. Litwin, D. Zegour, and G. Levy. Multilevel trie hashing. Preliminary report, INRIA, France, 1986.

[16] U. Manber and R. Baeza-Yates. An algorithm for string matching with sequence of don't cares. *Inform. Proc. Lett.*, 37:133 – 136, 1991.

[17] U. Manber and G. Myers. Suffix arrays: A new method fo on-line string searches. In *Ist ACM-SIAM Symp. on Discrete Algorithms*, pages 319 – 327, San Fancisco, 1990.

[18] T. H. Merrett and B. Fayerman. Dynamic patricia. In *Proc. Int'l. Conf. on Foundations of Data Organization*, pages 13 – 20, 1985.

[19] D. R. Morrison. Patricia: Practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514 – 534, Oct. 1968.

[20] E. J. Otoo. Linearizing the directory growth of of extendible hashing. In *Proc. Int'l Conf. on Data Engineering*, Los Angeles California, 1988. IEEE.

[21] M. Scholl. A new file organization based on dynamic hashing. *ACM Trans. on Database Syst.*, 6(1):194 – 211, Mar. 1981.

[22] L Torenvliet and P. van Emde Boas. The reconstruction and optimization of trie hashing functions. In *Proc. 9th Int'l. Conf. on Very Large Databases*, pages 142 – 156, Florence, Italy, Nov. 1983.

[23] D. Wood. *Data Structures, Algorithms, and Performance.* Addison Wesley Publishing Co., Reading, Massachusetts, 1993.

[24] W. E. Wright. Binary search tree in secondary memory. *Acta Info.*, 15:3 –17, 1981.

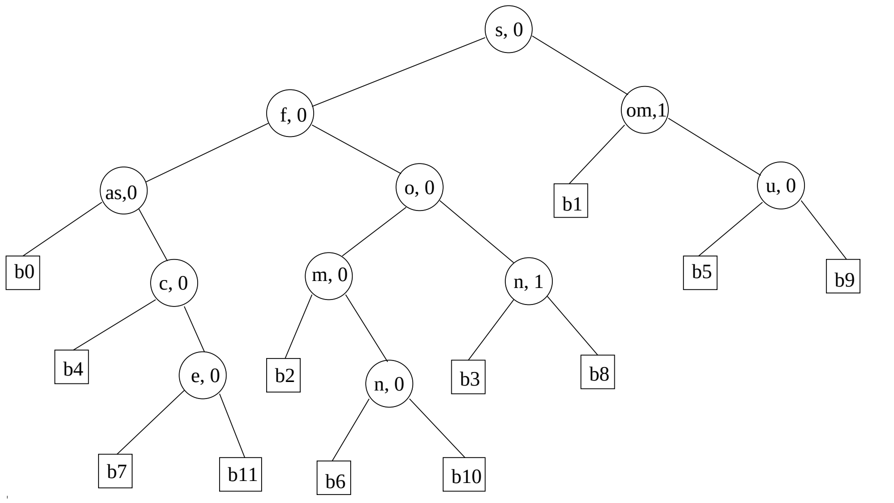


Figure 7: An example of a compact binary trie.

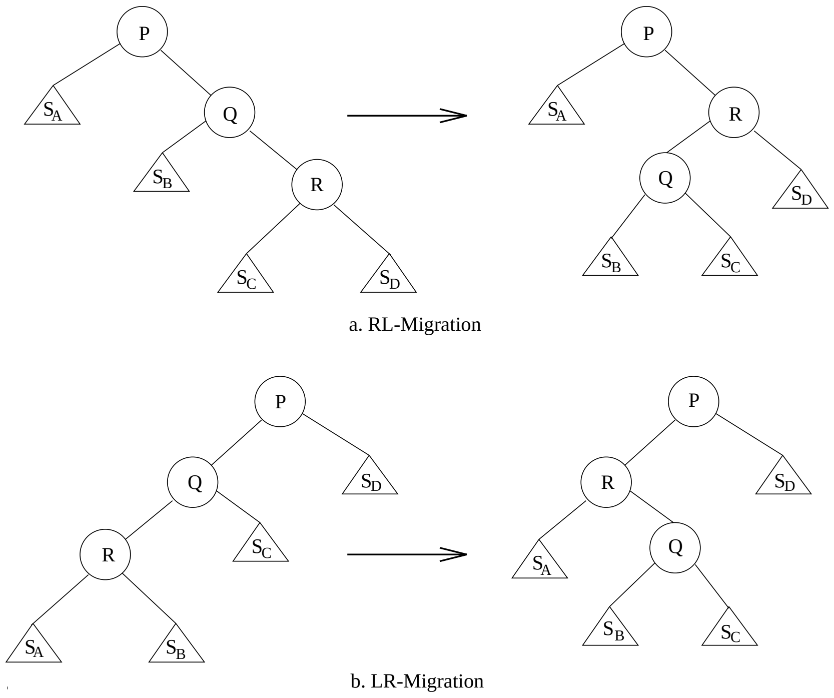


Figure 8: Illustration of RL and LR migrations of the BCTH.

The rbTrie Insertion Algorithm

Assumption: The algorithm invokes the Trie-Insert() insertion algorithm of the basic trie.

```
Begin
Let the trie be rooted $T$, and let $k$ be the key to be inserted.
Trie-Insert($T, k$);
if bucket split does not occur, return;
color inserted internal node $x$ red;
while $x \neq root[T]$ and ($color[f(x)]$ = red )
    let $gf(x) = f(f(x))$;
        if (DN[x] < DN[f(x)]) {
            if (x = left(f(x))
                Right-Rotate(T, f(x))
            else
                Left-Rotate(T, f(x))
        }
        else {
            if (DN[x] > DN[f(x)]) {
                colour[x] ← black;
                x ← f(x) ;
            }
            else {
                if (f(x) = left(gf(x)) {
                    y ← right(gf(x));
                    if (color[y] = red) {
                        color[f(x)] ← black;
                        color[y] ← black;
                        color[gf(x)] ← red;
                        x ← gf(x);
                    }
                    else {
                        if $x$ = right($f(x)$) {
                            $x \leftarrow f(x)$;                          Case 2.
                            Left-Rotate($T, x$);
                        }
                        color[$f(x)$] ← black;                           Case 3.
                        color[$f(f(x))$] ← red;
                        Right-Rotate($T, f(f(x))$);
                    }
                else {(same as the if clause with                 Cases 4, 5, 6.
                        "right" and "left" exchanged)
                }
            }
        }
endwhile;
color[$root(T)$] ← black;
End
```

Figure 9: The insertion algorithm of the rbTrie

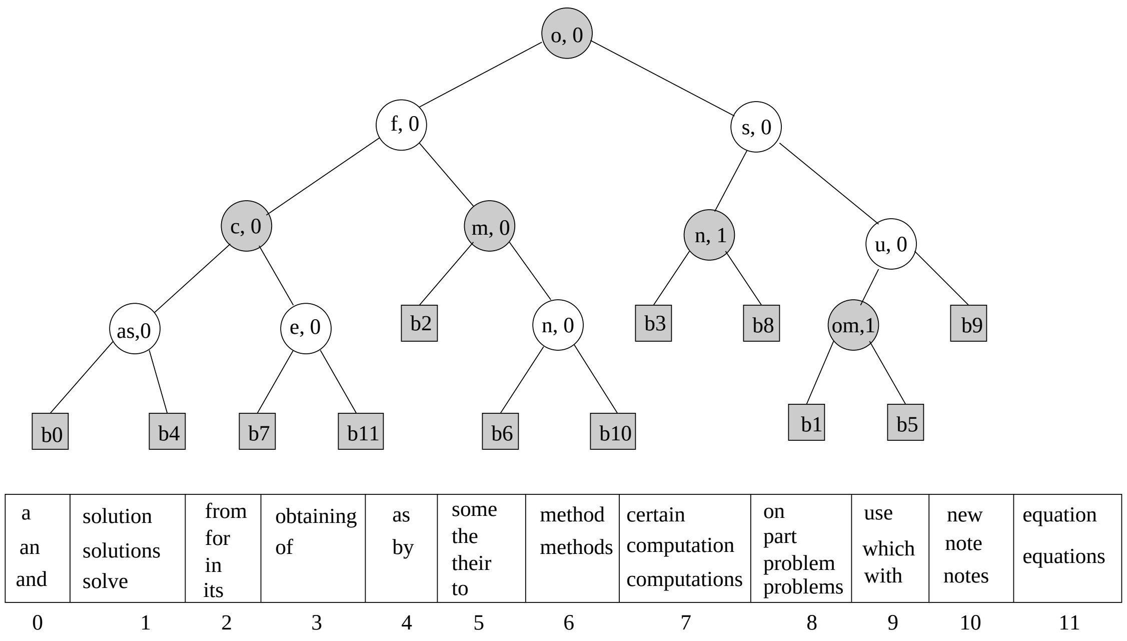


Figure 10: An example of a rbTrie from the insertion of random keys.

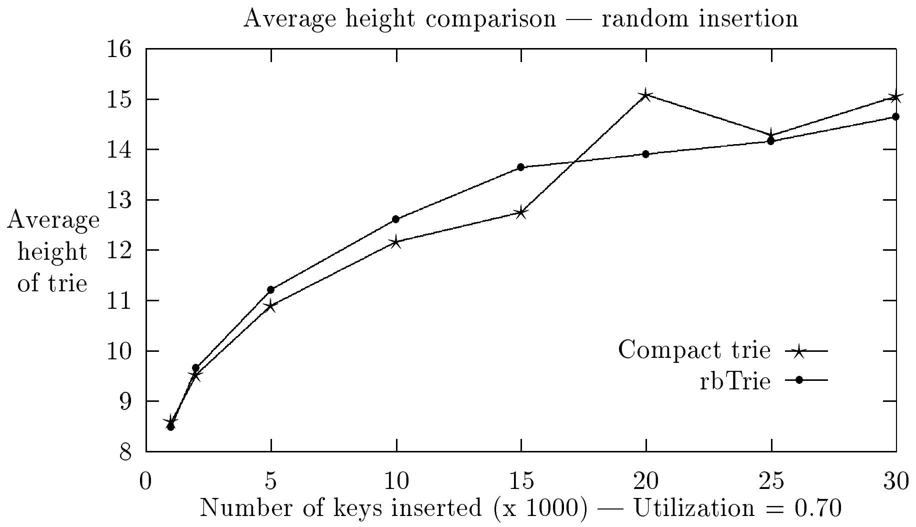


Figure 11: Average height comparison $b = 10$, random insertion.

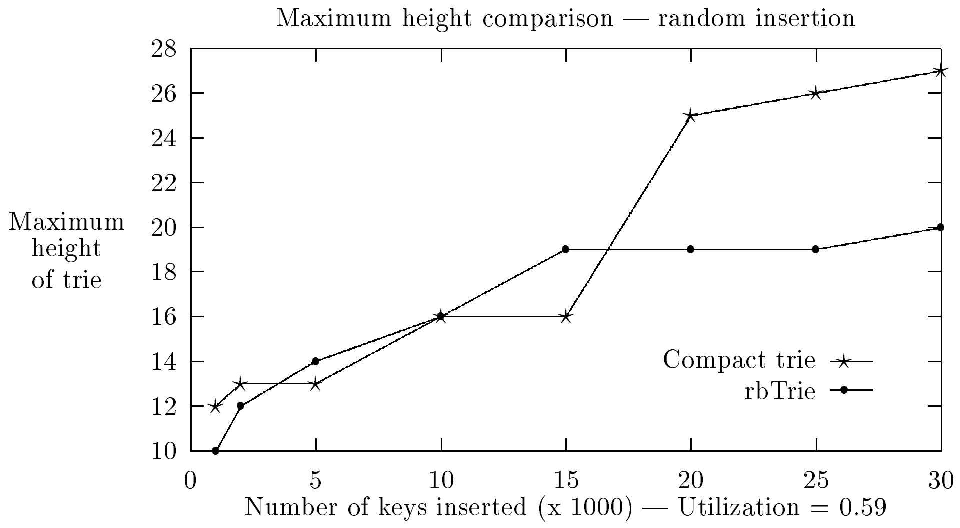


Figure 12: Maximum height comparison $b = 10$, random insertion.

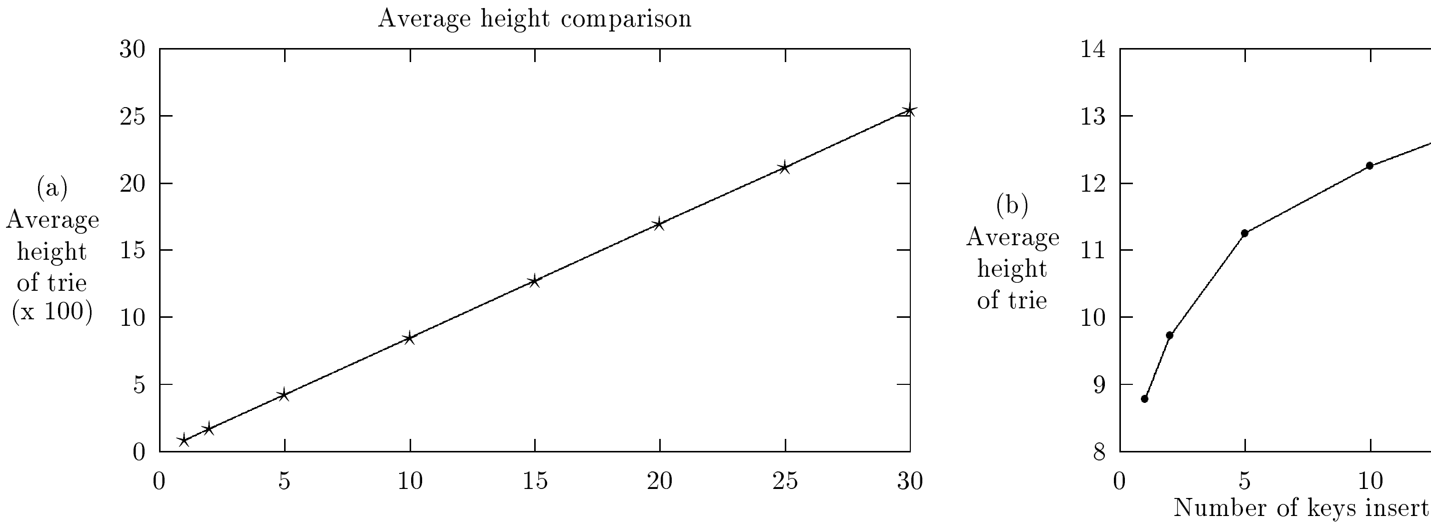


Figure 13: Average height comparison $b = 10$, sorted insertion.

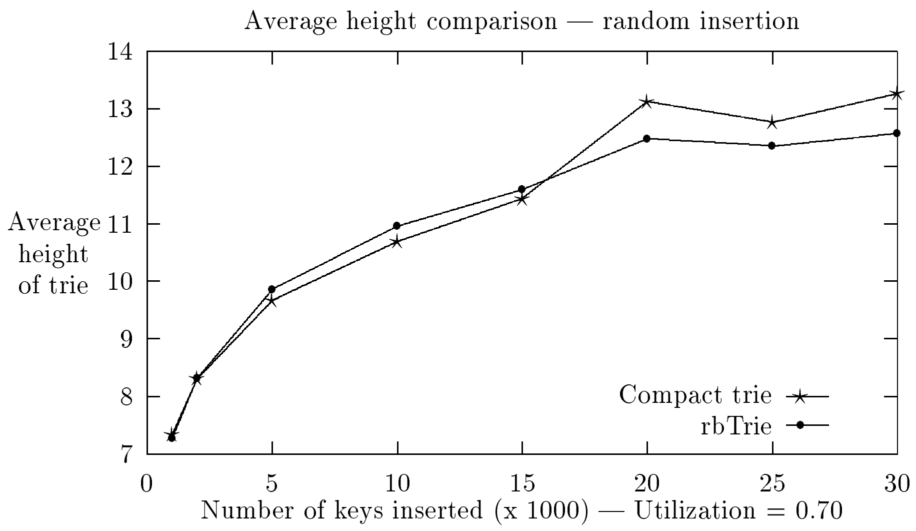


Figure 14: Average height comparison $b = 20$, random insertion.

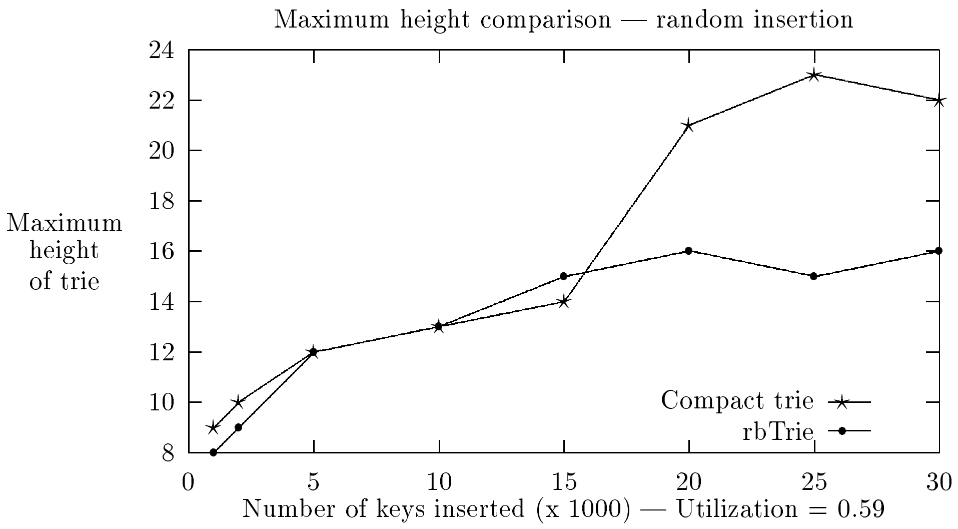


Figure 15: Maximum height comparison $b = 20$, random insertion.