A HASHING TECHNIQUE USING SEPARATE BINARY TREE

Md. Mehedi Masud[1*], Gopal Chandra Das[3], Md. Anisur Rahman[2], and Arunashis Ghose[4]

[*1]*School of Information Technology and Engineering, University of Ottawa, Canada*

Email: mmasud@site.uottawa.ca

[2]*Computer Science and Engineering, Khulna University, Khulna, Bangladesh*
[3]*Computer Science and Automation, Indian Institute of Science (IISc), Bangalore, India*
[4]*Department of Telecommunication and Signal Processing, Blekinge Institute of Technology, Karlskrona, Sweden.*

ABSTRACT

It is always a major demand to provide efficient retrieving and storing of data and information in a large database system. For this purpose, many file organization techniques have already been developed, and much additional research is still going on. Hashing is one developed technique. In this paper we propose an enhanced hashing technique that uses a hash table combined with a binary tree, searching on the binary representation of a portion the primary key of records that is associated with each index of the hash table. The paper contains numerous examples to describe the technique. The technique shows significant improvements in searching, insertion, and deletion for systems with huge amounts of data. The paper also presents the mathematical analysis of the proposed technique and comparative results.

Keywords: Database, Hashing, Information retrieval

1 INTRODUCTION

In a database file organization, records can be organized in various ways. The principal goal of these different organizations is to simplify the complexities of the operations on database files. In a small database, complexity in performing database operations is not a major concern. But when the database grows larger, we need efficient mechanisms for performing database operations that access database records.

Accessing a record from a file requires some sort of unique identification that differentiates that record from other records. To achieve this unique identification, each record is associated with a unique key. In hashing, a hash function is used to generate a unique hash key based on where the record is stored in memory, termed a bucket (Silberschatz, Korth, & Sudarshan, 1997) According to different hash functions and bucket structures, different hash techniques have been developed. In this paper, we review some existing hash techniques, s*eparate chaining* (Dietrich & DeLillo, 2003) and *dynamic hashing* (Enbody & Du, 1988), and propose an enhanced technique that shows significant improvements over the reviewed ones. The proposed technique builds the advantages of and eliminates the disadvantages of those techniques. We illustrate different operations, for example, retrieval, insertion, and deletion of records to validate the performance of the technique.

The remainder of the paper is organized as follows. In Section 2, we briefly review the basics of hashing and summarize s*eparate chaining* and *dynamic hashing* techniques. Section 3 introduces the data structure used in the proposed technique. Section 3 also discusses the proposed technique and presents several examples to illustrate retrieval, insertion, and deletion mechanisms. In Section 4, we give a mathematical analysis of the algorithm and its validation using experimental results. Section 5 presents the experimental results and comparison with other techniques. Finally, we summarize our conclusions in Section 6.

2 OVERVIEW OF HASHING

Hashing is a file organization technique that arranges the records of a file in some special way so that they can be retrieved quickly and efficiently, minimizing unnecessary comparisons (Folk, Zoellick, & Riccardi, 1999). Basically, hashing is divided into two parts: (1) calculation of the hash function and (2) resolution of collision. The function that transforms a key into an index table is called a *hash function*. If *h* is a hash function and *key* is the identification

key of a record, *h(key)* is called the *hash of key* and is the index which indicates the location where the record with the key should be placed. If *r* is a record whose key hashes into *hr, hr* is called the *hash key* of *r*.

The principal criteria in selecting a good hash function are: 1) the function should produce as few hash *clashes* as possible; that is, it should spread the keys uniformly over the possible hash table indices (Cormen, Leiserson, & Rivest, 1999). Of course, unless the keys are known in advance, it cannot be determined whether a particular hash function disperses them properly. 2) A hash function should depend on every single bit of the key, so that two keys that differ in only one bit or one group of bits hash into different values. Thus a hash function that simply extracts a portion of a key is not suitable. 3) Similarly, if two keys are simply digit or character permutations of each other (such as 139 and 319 or *meal* and *lame*), they should also hash into different values. The reason for this is that key sets frequently have clusters or permutations that might otherwise result in collisions. Many hash functions are available in literature, and each has its own advantages and disadvantages depending on the set of keys to be hashed. One important consideration in choosing a hash function is efficiency of calculation.

2.1 Classification of hashing

Hashing can be classified into two categories: static hashing and dynamic hashing. In static hashing, the bucket size and storage capacity are predefined, which may be done using an array. Hence, this approach may waste memory if the number of records are less then the defined array size, or the out of memory situation occurs before allocation of all records (Langsam, Augenstein, & Tenenbaum, 1997). This method can be implemented for a small range of applications, as it is quite easy to do.

In dynamic hashing, the bucket size as well as the storage capacity is allocated dynamically (Kruse, 1996). This action may be achieved through a linked list, a binary tree, a B tree, or a B+ tree (Horowitz & Sahani, 1996) (Dale & Lilly, 1997). Hence, memory waste can be avoided. Also, insertion of new records has no theoretical bound. Dynamic hashing can be implemented for a large number of applications, and its complexity is much higher than that of static hashing. If a hash table is maintained in external storage such as on a disk or some other direct access device, it can be called hashing in external storage. When a record is requested, its key is hashed, and the hash table (now in internal memory) is used to locate the external storage address of the appropriate bucket.

Existing dynamic hashing techniques include linear hashing with overflow handling by linear probing (Larson, 1982), linear hashing with partial expansions, and linear hashing with chaining (Larson, 1985). Any of these techniques can be implemented for real world problems, and each has some particularly good features. Among these, linear hashing with separator finds records very fast, while separate chaining has the most efficient memory utilization. Each of the existing hashing techniques tries to resolve hash collisions in different ways. In the following subsections, two hashing techniques, *separate chaining* and *dynamic hashing*, are described. These two methods lay the ground work for the proposed extended technique.

2.1.1 Separate chaining

Let us suppose the hash routine produces values between 0 and *tablesize* – 1. The technique s*eparate chaining* declares an array of buckets of size *tablesize*, which is called a *hash table*. Each of the indexes (or buckets) of the hash table acts as the header node of a distinct linear linked list. *Bucket[i]* points to the list of all records whose keys hash into *i* (i.e., *h(key) = i*). While searching for a record, if the hash key value for the record is *j*, the list pointed to by *bucket[j]* will be accessed and then traversed. During insertion, if the record is not found, it is inserted at the end of the list. During deletion, if the record is found, the corresponding node is removed, and the links are reordered accordingly.

Some important points about this method that should be mentioned are:

- In this method, a very small amount of memory space is wasted, and there are no empty nodes. Only those indices in the hash table that containing NULL value are wasted. The margin of wastage is very low.
- The records in the list remain in unsorted order. As a result, the cost of searching, insertion, and deletion is very high.

- Moreover, the cost of searching may increase drastically if the hash function is not uniform. A non-uniform hash function may cause many records to hash into a single chain. The choice of a uniform hash function can be very difficult to make.

2.1.2 Dynamic hashing

Basically, dynamic hashing is a set of related and similar techniques. One type, which was introduced in the 1980's, is discussed below (Enbody & Du, 1988).

Initially, ***m*** buckets and a hash table (or index) of size ***m*** are allocated. Assuming that ***m*** equals 2^b and assuming a hash routine ***h*** that produces hash values that are ***w>b*** bits in length, let ***h(key)*** be the integer between 0 and ***m*** represented by the first ***b*** bits of *h(key)*. Then ***h*** is used initially as the hash routine, and records are inserted into the ***m*** buckets as in ordinary external storage hashing. When a bucket overflows, the bucket is split in two, and its records are assigned to the two new buckets based on the $(b+1)^{th}$ bit of *h(key)*. If the bit is 0, the record is assigned to the left bucket; otherwise it is assigned to the right bucket. The records in each of the two new buckets use the same first *b+1* bits in their hash keys, *h(key)*. Similarly, when a bucket representing *i* bits overflows (where $b <= i <= w$), the bucket is split, and the $(i+1)^{th}$ bit of *h(key)* for each record in the bucket is used to place the record in the left or right new bucket. Both new buckets then represent *i+1* bits of the hash key. The buckets whose keys have 0 in their $(i+1)^{th}$ bit are called *0-buckets*, and the others are called *1-buckets*.

Under dynamic hashing, each of the ***m*** original index entries represents the root of a binary tree whose leaves contains a pointer to a bucket. Initially, each tree consists of only one node (a leaf node) that points to one of the ***m*** initially allocated buckets. When a bucket splits, two new leaf nodes are created to point to the new buckets. The former leaf that had pointed to the bucket being split is transformed into a non-leaf node whose left child is the leaf pointing to the *0-bucket* and whose right child is the leaf pointing to the *1-bucket*. To locate a record under dynamic hashing, ***h(key)*** has to be computed, and then the first ***b*** bits are used to locate a root node in the original index. One must use each successive bit of ***h(key)*** to move down the tree, going left if the bit is 0 and right if the bit is 1, until a leaf is reached. The pointer in the leaf is used to locate the bucket that contains the desired record if it exists. In the case of deletion, the node containing the *key* to be deleted holds a meaningless value (say, -1) after the *key* is deleted. As the node is not removed, memory for that node remains allocated after deletion.

Important points about this method are:

- The main advantage of dynamic hashing is that performance does not degrade as the file size grows.
- Buckets do not need to be reserved for future growth.
- When a record is deleted, the corresponding node remains allocated containing a meaningless value. Thus, deletion creates unused allocated space.
- As the number of nodes containing meaningless values remains in the structure, the overall cost of every operation increases.

The two existing methods discussed so far have both positive and negative aspects. The main disadvantage of separate chaining is that the cost of each operation is linear, i.e., of order ***O(n),*** where *n* is the number of records present in the database. Therefore, for large databases, this technique is almost impractical to implement. Although dynamic hashing is of order ***O(log n)***, deletion creates a problem. If deletion increases, the searching cost increases unnecessarily. To design a technique that will work efficiently for large databases, these two major problems must be solved. This is the goal of the proposed extended technique.

3 THE PROPOSED EXTENDED TECHNIQUE

The extended technique introduces a hash table such that each index of this hash table contains a separate binary tree. The hash table has fixed size, but the binary tree can be expanded as necessary and thus can accommodate large numbers of records. In the following, we introduce the structures and organization of the tables and binary trees, the calculation of the hash index, and the position of a record in the tree from the *key* value and possible database size.

3.1 Hash table structure

The Hash table is an array of pointers where each pointer points to a structure named *root_node*. Initially, the hash table is declared to be an array of an arbitrary number of pointers. Memory for each root node is allocated only when a tree needs to be constructed to store the first record under the corresponding index. In the discussion that follows, the array used here is of 100 indices, as shown in Figure 1, with *MaxIndex* equaling 100. The structure is defined below:

```
struct  root_node
{       struct  record_node  *left , *right ;
}*hash_table[MaxIndex].
```

The structure *root_node* in Figure 2 shows that each index of the array contains a pointer to a structure *root_node* that has two structure (*record_node*) pointers, *left* and *right*. The structure of *record_node* is described in Section 3.2.

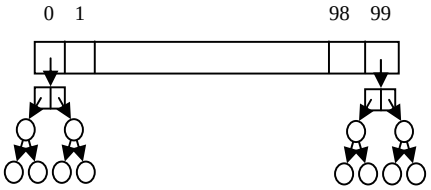


Figure 1. Hash table containing binary trees

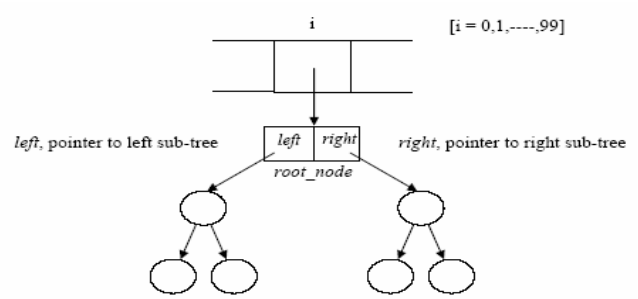


Figure 2. i[th] index pointing to the r*oot_node*

3.2 Binary tree structure

Each binary tree is constructed under a root node. The structure of the other nodes, excepting the root node, in the binary tree is shown in Figure 3. The field *key* contains the primary key of the corresponding record. The *balance_indicator* contains a positive or negative value indicating which sub-tree (left or right) of the node contains more nodes. Mathematically,

balance_indicator = number of nodes in the right sub-tree - number of nodes in the left sub-tree.

```
struct  record_node
{       long  key ;
        int  balance_indicatior ;
struct  record_node *left , *right ;
}
```

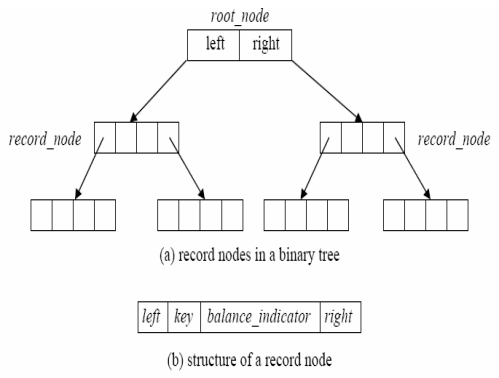


Figure 3. Binary tree structure

For example, *balance_indicator* = -2 indicates that the left sub-tree contains two more nodes than the right sub-tree. On the contrary, *balance_indicator* = 3 indicates that the right sub-tree contains three more nodes than the left sub-tree. *balance_indicator* = 0 means both the left and right sub-trees of the corresponding nodes contain the same number of nodes. When a node is created to store a record, its *balance_indicator* is assigned to zero. The value of this field helps us to keep the tree balanced, that is, to prevent as much as possible the tree from being linear. This technique helps to reduce the searching cost.

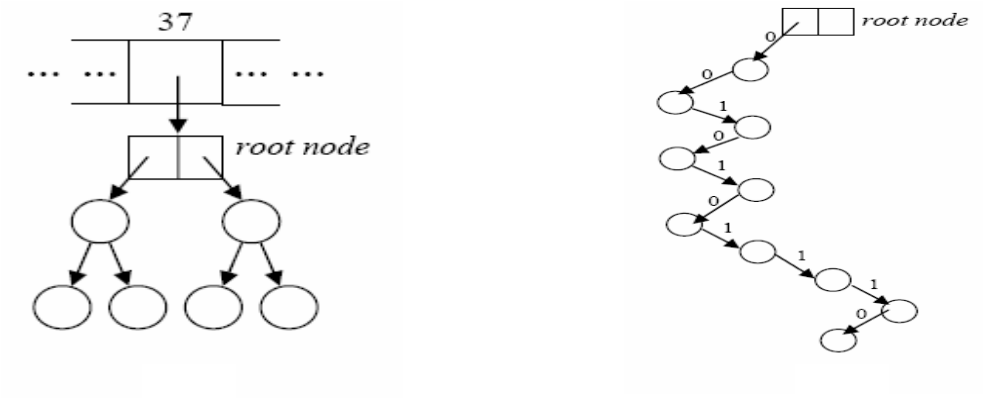
The pointers *left* and *right* point to the left and right sub-tree respectively of the corresponding node. When a node is created, its left and right pointers are assigned to NULL. The symbol Ø is used to represent NULL value.

3.3 Hash index and tree index

The hash index is the hash key generated by a hash function. The hash index of a *key* indicates the position of the hash table that holds the binary tree in which the *key* is to be inserted or is probably located. The hash function used here is given below.

$$hash\_index = key \bmod 100$$

For example, if the *key* is 46837, then the *hash_index* = 46837 mod 100 = 37. The tree index is a number that is generated using the following function *tree_index = key div 100*. In the proposed technique, the binary equivalent tree index is used to locate or place a *key* in the corresponding tree.



(a) Position of the tree (b) Possible positions in the tree

Figure 4. Positioning a key 46837

For example, for the above *key* 46837, the *tree_index* = 46837 div 100 = 468. This *tree_index* is further converted to its binary equivalent of a specific size. While implementing the technique in this example, 10 bits are used to represent the binary equivalent of the *tree_index*. Using 10 bits for the binary equivalent is not rigid, and it can be varied depending on the possible future database size. Here, the binary equivalent of *tree_index* 468 is 0111010100. Starting from the least significant bit (LSB), the tree turns left if the bit is 0 and right if the bit is 1.

3.4 Searching

In this section we describe the searching mechanism. The *key* to be searched is broken into two parts, the *hash_index* and the *tree_index*, with the functions *hash_index = key mod 100* and *tree_index = key div 100* respectively, as described above. The *hash_index* is used to locate the corresponding index in the hash table. Then it must be checked to see if the root node of the corresponding index exists. If a root node exists, then a binary tree has already been created, and the *key* is searched using *tree_index*. Absence of the root node indicates the absence of a tree. That is, no data having the *hash_index* has yet been entered. Thus the searching process terminates with failure.

If a binary tree exists already, the process turns left or right depending on the value of the binary equivalent of the *tree_index* starting from the LSB. If a 0 is found, it turns to the left; otherwise it turns to the right. At each step, the *key* of the corresponding node is checked. If a match is found, the searching process terminates with success. Otherwise, this traversal continues until the next left or right pointer contains the NULL value. The algorithm is shown in Algorithm A1.

Algorithm for Searching

Step 1 : If the hash_index contains no tree, then go to step 4.
Step 2 : If the LSB equals zero, traverse to the left child of the root node, if it exists. Otherwise, traverse to the right child of the root, if it exists. If neither exists, go to step 4.
Step 3 : For each bit of binary equivalent except the LSB, Check for the key in the corresponding node. If it is not found, depending on the Bit value (zero or one respectively), traverse to the left or right child. If it is found, go to step 5.
Step 4 : Show message "key not found" and go to step 6.
Step 5 : Show message "key found".
Step 6 : End.

Algorithm A1. Algorithm for searching

Example 1: Suppose the keys are stored with the configuration in Figure 6. The construction process of such a structure is described within the insertion process in section 3.5. Table 1 contains the *key*, *hash_index*, *tree_index* and binary equivalent of *tree_index* of the corresponding records.

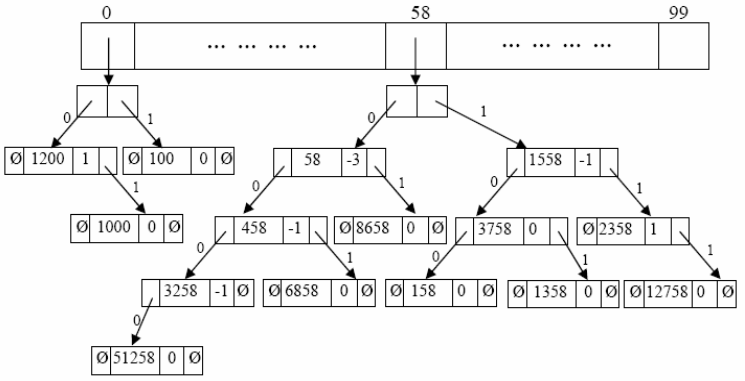


Figure 6. The trees constructed using the sample records

Table 1. A set of sample *key*s

key	*hash_index*	*tree_index*	Binary equivalent of *tree_index*
1558	58	15	0000001111
100	0	1	0000000001
58	58	0	0000000000
1200	0	12	0000001100
3758	58	37	0000100101
158	58	1	0000000001
458	58	4	0000000100
2358	58	23	0000010111
1000	0	10	0000001010
3258	58	32	0000100000
6858	58	68	0001000100
12758	58	127	0001111111
51258	58	512	1000000000
8658	58	86	0001010110
1358	58	13	0000001101

Suppose, the keys 458, 499, and 1358 are to be searched. Their corresponding *hash_index*, *tree_index* and binary equivalent of *tree_index* are given in Table 2.

Table 2. The keys to be searched

Key	*hash_index*	*tree_index*	Binary equivalent of *tree_index*
458	58	4	0000000100
499	99	4	0000000100
1358	58	13	0000001101

To search for the *key* 458, it is first broken into *hash_index* and *tree_index*. The *key* is then hashed into index 58. The process finds that there exists a root node; i.e., a binary tree has been already created. The binary equivalent of the *tree_index* is then calculated to be 0000000100. Because the LSB, i.e. the 0 bit, is 0, the process moves to the left from the root node. The *key* 58 found there is not the desired key. Thus the next bit, i.e. the 1st bit, is considered. As this bit is 0**,** it again takes a left turn and finds the desired *key*. Finally, the searching process terminates with success.

To search for the *key* 499, the *key* hashes into the 99th index according to the *hash_index* of the *key*. But the process observes the absence of the root node. Therefore, no *key* having *hash_index* 99 has been inserted yet. Thus, the search *key* is not present in the database.

To search for 1358, the *key* hashes into the 58th index of the hash table. The calculated binary equivalent is 0000001101. First it takes a right turn. As a match is not found, it then turns left. The process again fails to get a match. Because the 2nd bit is 1, the process then moves to the right, and the search *key* is found.

3.5 Insertion

This section describes the insertion mechanism. The *key* to be inserted is first broken into two parts, the *hash_index* and the *tree_index*, with the functions *hash_index = key mod 100* and *tree_index = key div 100* respectively, as performed in the searching procedure. The *hash_index* is used to locate the corresponding index in the hash table. It must be checked to see whether the root node of the corresponding index exists. If a root node exists, then it is clear that a tree has already been created. The process then inserts the *key* using the binary equivalent of the *tree_index*. On the other hand, the absence of the root node indicates that no tree has been created yet; that is, no data having this *hash_index* has been hashed before. In this case, a root node is created, and the *key* is inserted either to the left or right of the root node according to the LSB of the binary equivalent. The algorithm is shown in Algorithm A2.

Algorithm for Insertion

Step 1 : If the hash_index contains no tree, create a root node and insert the key to its left or right child after creating it, according to the LSB value.
Otherwise, traverse to the left or right according to value of the LSB.
Step 2 : For each bit of binary equivalent except LSB:
If there is no node, create a node and insert the key. If there is a node already and no duplicate is found, keep track of the node and direction and then traverse to the left or right in the same way. If a duplicate is found, go to step 4.
Step 3 : Update the balance indicator of the corresponding node and go to step 5.
Step 4 : Show message “key already exists”.
Step 5 : End.

Algorithm A2. Algorithm for insertion

If a binary tree already exists, the process turns left or right depending on the value of the binary equivalent of the *tree_index*, starting from the LSB. If a 0 is found, it turns to left. Otherwise it turns to the right. Each time the process checks whether the *key* of the corresponding node matches. If a match is found, the insertion process terminates after generating the message “the key already exists”. Otherwise, this traversal continues until a leaf node is reached. According to the corresponding bit value, a new record is then created to store the *key* either to the left or the right of that leaf node.

While traversing the tree, in order to find the appropriate position of the *key* to be inserted or to check whether it already exists, the process has to keep track of the path. If the *key* is inserted, then the value of the *balance_indicator* of those nodes that are encountered along the path of traversal is updated. For example, consider Figure 7.

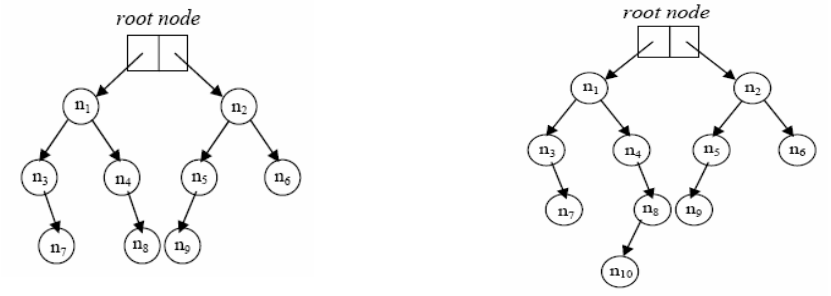


Figure 7. Operations: (a) before insertion (b) after insertion

Consider that before insertion, the tree contains 9 nodes, $n_1, n_2, \ldots, n_9$. Here, the node $\mathbf{n_i}$ has the *balance_indicator* value b_i, where $i = 1, 2, \ldots, 9$. Suppose a new node n_{10} is to be inserted, and the insertion procedure finds its position of insertion to the left of the node n_8. To reach the position of insertion, the process has to traverse along the path that contains the nodes n_1, n_4, n_8. Therefore, the addresses of these nodes must be stored. In addition, for each of the nodes n_1, n_4, and n_8, a separate integer value is stored which is added to the corresponding *balance_indicator*. During traversal, if the process turns left, –1 is stored, and if it turns right, 1 is stored. Thus, corresponding integer values for the nodes n_1, n_4, and n_8 are 1, 1, and -1 respectively. Therefore, after insertion, the *balance_indicator* of the nodes n_1, n_4 and n_8 are b_1+1, b_4+1, and b_8-1 respectively.

Example 2: Consider the database structure in Figure 6 in which some new keys 2858 and 599 are to be inserted. The corresponding *hash_index*, *tree_index*, and the binary equivalent of these keys are given in the Table 3.

Table 3. The keys to be inserted

Key	*Hash_index*	*tree_index*	Binary equivalent of *tree_index*
2858	58	28	0000011100
599	99	5	0000000101

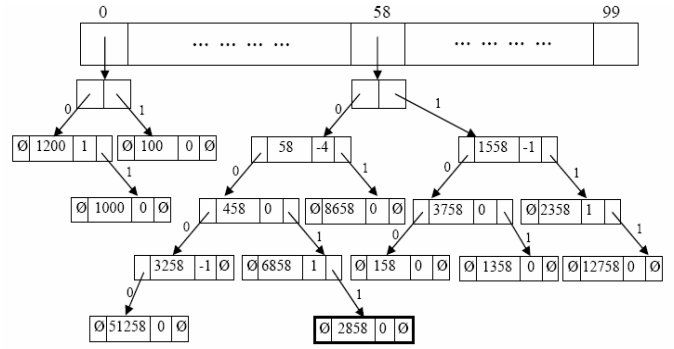


Figure 8. The tree structure after inserting the *key* 2858

To insert the *key* 2858, the process first hashes into the 58th index of the hash table. Its *tree_index* is 28, and the binary equivalent is 0000011100. To find the appropriate position of insertion, the process traverses the tree in the same manner as the searching process. During traversal, each node is checked on the way to be sure that the *key* doesn't exist already. If the *key* already exists, a message is generated, and the process terminates. Because the LSB, i.e. the 0 bit, is 0, the process turns left from the root node and finds the *key* 58. As the 1st bit is 0, it has to turn left again, and at the same time the address of this node and a value –1 against this node are stored. Then the process reaches the node holding the *key* 458, which doesn't match the *key* 2858. It turns right as the 2nd bit is 1. The address

of the node containing 458 and a value 1 against it are stored. Then it reaches the node containing the *key* 6858, also no match. The 3rd bit is 1, and it has to turn right. The value 1 against the address of the node containing 6858 is stored. Because this is a leaf node, a new node to the right of this node is created, and the *key* 2858 is stored there. Now the *balance_indicator* values of the stored nodes are updated by adding the values stored against them. After the insertion of *key* 2858, the database structure is shown in Figure 8.

Table 4 shows the value of the balance_indicator of the stored nodes before and after insertion of the key 2858.

Table 4. Changes of *balance_indicator* after insertion

Key values of the stored nodes	Value of balance_indicator	
	Before insertion	After Insertion
58	-3	-3 + (-1) = -4
458	-1	-1 + 1 = 0
6858	0	0 + 1 = 1

Now the *key* 599 is inserted. This *key* hashes to the 99th index of the hash table which does not have a root node. The root node must be created first. The binary equivalent is 000000010. A record node is created to the right of the root node, as the LSB is 1. Its *balance_indicator* is initialized to zero.

Figure 9 shows the structure after inserting the *key* 599.

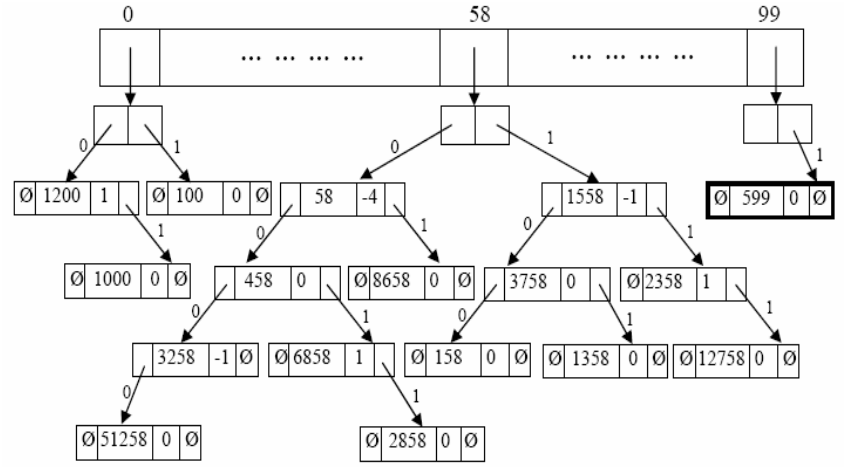


Figure 9. The tree structure after inserting the *key* 599

3.6 Deletion

This section describes the deletion technique. The first step is to calculate the ***hash_index, tree_index*** and binary equivalent as done in the searching procedure. The ***hash_index*** is used to locate the corresponding index in the hash table. The next step is to see whether or not the root node of the corresponding index exists. The presence of a root node indicates that a tree already exists. Then the process finds the *key* for deletion using the binary equivalent of the *tree_index*. On the other hand, the absence of the root node indicates that no tree is present; that is, no data having this *hash_index* has been entered yet. In this case, the deletion process terminates, generating a message "the key is not found".

If a binary tree already exists, the process traverses the tree to find the node containing the *key* to be deleted as in the searching procedure. While traversing, it keeps track of all the record nodes it encounters on the path to the node containing the *key* to be deleted, if it exists, from the root node. It stores the addresses of the nodes encountered along the path, and in association with each address a distinct integer value is also stored.

Algorithm for Deletion

Step 1 : If the hash_index contains no tree, then go to step 5.
Step 2 : If the LSB equals zero, traverse to the left child of the root. **Otherwise,** traverse to the right child.
Step 3 : For each bit of the binary equivalent except the LSB:
If there is no node, go to step 5. If the key is found and the corresponding node is a leaf node, delete it; Otherwise, call the reordering scheme (replace by a leaf).
If the key is not found, keep track of the node and direction and then traverse to the Left or right in the same way.
Step 4 : Update the balance indicator of the corresponding node and go to step 6.
Step 5 : Show message "key not found".
Step 6 : End.

Algorithm A3: Algorithm for deletion

If the process fails to find the *key* value, it terminates with a failure. If the *key* exists and the node containing it is a leaf node, it then deletes the node by freeing the memory allocated by it. In the case where the *key* is contained by a non-leaf node, it tries to find a leaf node in either the right or left sub-tree of the node containing the *key*. If the left sub-tree contains n_1 number of nodes and the right sub-tree contains n_2 number of nodes and $n_1 \geq n_2$, then a leaf node is chosen from the left sub-tree. If $n_1 < n_2$, the leaf node is chosen from right sub-tree. This choice is made with the help of the value of the *balance_indicator*. If the *balance_indicator* is less than or equal to zero, the process turns left; otherwise it proceeds right. In this way, while the process searches for a leaf node, the addresses of each node on the path including the node containing the *key* to be deleted and an integer value for each node are stored. The integer value is either 1 or –1 depending on the direction of the traversal. –1 is stored for a left turn, and 1 is stored for a right turn.

When the process finds a leaf node, it first replaces the *key* to be deleted by the *key* contained by this leaf node. Then it deletes this leaf node by freeing its respective allocated memory. After that, using the corresponding stored integer value**,** the values of the *balance_indicator* of the nodes whose addresses are already stored is updated. An integer value, either 1 or –1, is subtracted from the *balance_indicator* of the corresponding node. Consider the tree structure in the Figure 10. The process of deleting a non-leaf node is explained further using this structure.

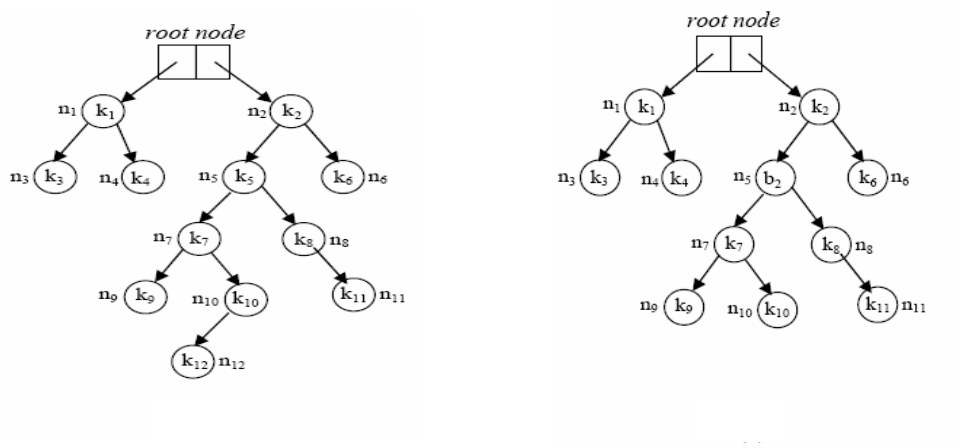


Figure 10. (a) Before deletion (b) After deletion

Suppose before deletion the tree contains nodes $n_1, n_2, \ldots, n_{12}$. Each node n_i contains the *key* k_i for $i = 1, 2, \ldots, 12$. Also, the node n_i has the *balance_indicator* value b_i, where $i = 1, 2, \ldots, 12$. From Figure 10(a), we get $b_1 = 0$, $b_2 = -6$, $b_3 = 0$, $b_4 = 0$, $b_5 = -2$, $b_6 = 0$, $b_7 = 1$, $b_8 = 1$, $b_9 = 0$, $b_{10} = -1$, $b_{11} = 0$, and $b_{12} = 0$.

Consider that the *key* k_5 is selected for deletion. To reach the node containing k_5, the process has to turn right from the root node and then turn left from node n_2. The address of node n_2 and an integer value –1 against it are stored. The node n_5 containing the *key* k_5 is a non-leaf node. Therefore, a leaf node from either the left or right sub-tree of n_5 has to be found. Because $b_5 < 0$, the process turns left, and the address of node n_5 and a value –1 is stored. The process reaches n_7 and from n_7 has to turn right as n_7 is not a leaf node and $b_7 > 0$. At this point, the address of n_7 and a value 1 against it are stored. Then it reaches the node n_{10}, which is also not a leaf node. From n_{10}, it turns left as $b_{10} < 0$. Here, the process stores the address of n_{10} and a value –1 against it. After that it arrives at the node n_{12} , a leaf node.

Now the *key* k_5, the target for deletion, is replaced by the *key* k_{12} contained by the node n_{12}. Then the node n_{12} is removed. After that the values of the *balance_indicators* of all the nodes whose addresses are already stored are updated by subtracting the corresponding stored value from them. Here, b_2, b_5, b_7, b_{10} are updated as shown in the Table 5.

Table 5. Changes of *balance_indicator* after deletion

Balance_indicator	Before deletion	After deletion
b_2	-6	-6 – (-1) = -5
b_5	-2	-2 – (-1) = -1
b_7	1	1 – 1 =0
b_{10}	-1	-1 – (-1) = 0

Example 3: Consider the database structure in Figure 11 from which the key 458 is to be deleted. Table 2 shows its *hash_index*, *tree_index* and the binary equivalents of the key.

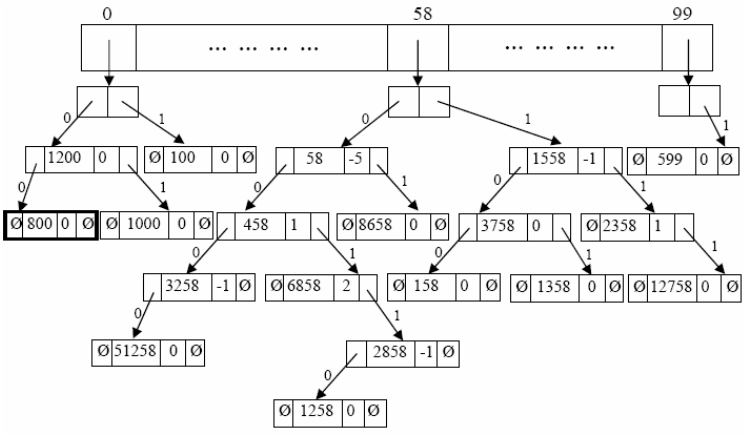


Figure 11. A database structure

To delete the *key* 458, the process hashes into the 58th index of the hash table. Its *tree_index* is 4, and the binary equivalent is 0000000100. Prior to deletion, it is necessary to find the node containing 458 as in the searching process. As the LSB, the 0 bit, is 0, the process turns left from the root node and finds the *key* 58. Since the 1st bit is 0, it has to turn left again, and at the same time, the address of this node and a value –1 against this node are stored. It then reaches the node holding the *key* 458 and deletes it.

Now a leaf node from either the left or right sub-tree of the node containing 458 has to found. As the *balance_indicator* of the node containing 458 is 1, which is greater than zero, the process proceeds towards the right form of this node. At the same time, the address of this node and a value 1 against it are stored. Next it arrives at the node containing 6858, which is not a leaf node. Since the *balance_indicator* of this node is 2, the process must turn right. Before that, it stores the address of this node containing 6858 and a value 1 against it. Then the process reaches the node holding *key* 2858, which is also non-leaf node. Its *balance_indicator* value is –1. Therefore, it turns left from here after storing the address of this node and a value –1 against it. Then the node, containing *key* 1258, is reached and is a leaf node.

Because the process has found the appropriate leaf node, it first replaces the *key* 458 by 1258 and then deletes the node containing the *key* 1258. After that, it updates the **v**alues of the *balance_indicators* of the nodes whose addresses have been stored using the corresponding stored integer values, either 1 or –1. Here, the integer value is subtracted from the *balance_indicator*. Table 6 shows the changes of the *balance_indicator* values.

Table 6. Changes of *balance_indicator* after deletion

Key values of the stored nodes	Value of balance_indicator	
	Before insertion	After Insertion
58	-5	-5 + (-1) = -4
458/1258*	1	1 – 1 =0
6858	2	2 – 1 = 1
2858	-1	-1 – (-1) = 0

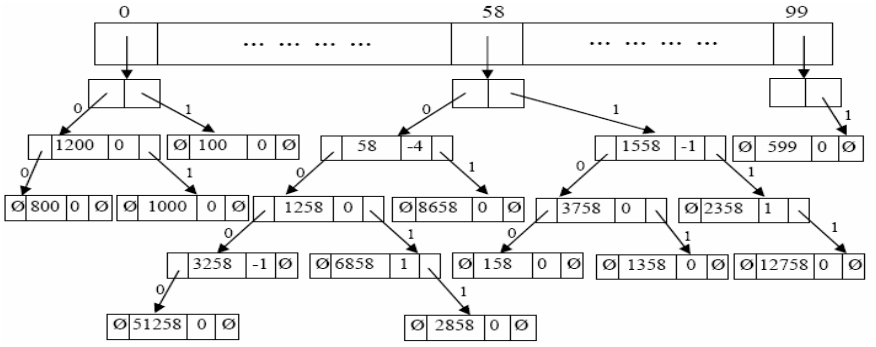


Figure 12. After deletion of key 458

The database structure after deleting the *key* 458 is shown in Figure 12.

4 MATHEMATICAL ANALYSIS

4.1 Successful search

To analyze this method mathematically, let us assume that the records are uniformly distributed among the tree and each tree is also constructed uniformly.

For a successful search, let,

r = number of records in the database,
i = number of indices in the hash table,
p = number of records in a tree,
b = number of bits to represent p,
q = number of internal nodes,
e = number of external nodes, and
I = internal path length (total path length to reach all the internal nodes from the root node).

Here, $p = 2^b$ as ***b*** is the number of bits required to represent ***p***. Therefore, $b = \log_2 p$.
Because ***b*** must be a positive integer, $b = \lceil \log_2 p \rceil$.
The total number of nodes in a tree, $p = r/$; thus $b = \lceil \log_2 (r/i) \rceil$.

The $(b-1)^{th}$ and b^{th} levels consist of $(2^{b-1}-2)$ and 2 leaf nodes respectively. Then the total number of external nodes,

$$e = 2^{b-1}-2+2$$
$$= 2^{b-1}, \text{ where } b \geq 2.$$

The rest of the nodes are internal, and the total number of internal nodes,

$$q = p - e$$
$$= 2^b - 2^{b-1}$$
$$= 2^{b-1}(2-1)$$
$$= 2^{b-1}, \quad \text{where } b \geq 2.$$

For b bits, $$I = \sum_{n=1}^{b-2} (n*2^n) + 2*(b-1), \text{ where } b \geq 2.$$

The following steps are perfomed to obtain the average number of comparisons S_{avg} in a successful search

1. Divide **I** by the number of internal nodes, q;
2. Add **1** for a leaf node or **0** for a non-leaf node;
3. Multiply that result by 3 because of the following three comparisons:
   - One for checking the target key,
   - One for making the decision to turn right or left, and
   - One for determining if a child node is required at each internal node.
4. Add 3 because of the following three comparisons:
   - One checking if a tree exists in the corresponding hash index, and
   - Two for making the decision to turn right or left from the root node (one for checking the binary equivalent and the other for checking the NULL value); and
5. Subtract 2 for the one comparison done at the node where the target is found (compare to the three comparisons required in step 3).

The average number of comparisons for a successful search is

$$S_{avg} = (I/q+1)*3+3-2$$
$$= (I/q)*3+4, \quad \text{where } b \geq 2.$$

For $b = 0$ and 1, there is no internal path, that is, $I = 0$. In this case,

$$S_{avg} = 4.$$

Finally,

$$S_{avg} = \begin{cases} 4, & 0 \leq b \leq 1 \\ (I/q) * 3 + 4, & b \geq 2 \end{cases}$$

For Figure 12, $S_{avg} = (16/8)*3+4 = 10$.

Table 7. Comparison between mathematical analysis and experimental result

Number of records	Successful Search	
	Mathematical Analysis (No. of probes)	Experimental result (No. of probes)
100	4	4
200	4	4
400	7	6
800	9	7
1600	10	9
3200	12	11
6400	14	14
12800	17	17
25600	19	20
51200	22	22
102400	25	25
204800	28	28

Table 8. Comparison between mathematical analysis and experimental result

Number of records	Unsuccessful Search	
	Mathematical Analysis (No. of probes)	Experimental result (No. of probes)
100	5	4
200	6	6
400	9	8
800	11	10
1600	13	12
3200	15	15
6400	18	18
12800	21	21
25600	24	24
51200	27	27
102400	30	30
204800	33	31

4.2 Unsuccessful search

An unsuccessful search ends at a leaf node.

Let

x = Total path length to reach the leaf nodes at $(b-1)^{th}$ level from the root node,
y = Total path length to reach the leaf nodes at b^{th} level from the root node,
m = Total number of comparisons for the leaf nodes at $(b-1)^{th}$ level, and
n = Total number of comparisons for the leaf nodes at b^{th} level.

There are $(2^{b-1}-2)$ leaf nodes at the $(b-1)^{th}$ level, and the path length for each of these nodes from the root node is $(b-1)$. The total path length for these nodes is

$$x = (2^{b-1}-2)*(b-1), \text{where } b \geq 2.$$

Because at each node except the root node (as described in step 3) 3 comparisons are needed and at the root node (as described in step 4) 3 and 2 comparisons are needed, the total number of comparisons for the nodes at $(b-1)^{th}$ level is

$$m = x*3+(2^{b-1}-2)*2, \text{ where } b \geq 2.$$

Again, there are 2 leaf nodes at b^{th} level and the path length is b for each of these nodes from the root node. The total path length for these nodes is

$$y = 2*b, \text{ where } b \geq 2.$$

Because at each node 3 and at the root node 2 comparisons are needed, the total number of comparisons for these nodes of b^{th} level is

$$n = y*3+2*2, \text{ where } b \geq 2.$$

The average number of comparisons for an unsuccessful search is determined in the following way:

1) Adding m and n,
2) Dividing the result by the number of external nodes e , and
3) Adding this result to **1** as one comparison is needed to check whether the hash index contains a tree or not.

Therefore, the average number of comparisons for an unsuccessful search is

$U_{avg} = [(m+n)/e]+1$, where $b \geq 2$.

For $b = 0$, five comparisons are required for one side and two for other side. In doing this, we find the average number of comparisons to be 3.5. One comparison is added to this average for checking the existence of a tree. Therefore, $U_{avg} = 4.5$.

For $b = 1$, five comparisons are required for both sides, and we find the average number of comparisons to be 5. Similarly, one comparison is added to this average. Therefore, $U_{avg} = 6$.

In summary,

$$U_{avg} = \begin{cases} 4.5, & b = 0 \\ 6, & b = 1 \\ [(m+n)/e]+1, & b \geq 2 \end{cases}$$

4.3 Deletion

To delete a node, first a search for it must be executed. Then the node is checked to see whether or not it is a leaf node. If it is a leaf node, the node is simply removed; otherwise a reordering scheme is needed to replace the *key* to be deleted with a *key* of its leaf node. As the number of internal nodes and the number of external nodes are equal, i.e., 2^{b-1}, for a tree having b levels $(b \geq 2)$, the probability of the necessity of a reordering scheme for a deletion of a node is 0.5.

The average cost of deletion = Cost of a successful search
+ 0.5 * Cost of checking a leaf node
+ 0.5 * Cost of reordering scheme.

That is, $D_{avg} = S_{avg} + 0.5 * 2 + 0.5 * R_{avg.}$
$= S_{avg} + 0.5 * R_{avg} + 1$, where $b \geq 2$
And D_{avg} = Average cost of deletion and
R_{avg} = Cost of reordering scheme.

While reordering the tree, the traversal starts from the node that is to be deleted and must end at a leaf node.

For a uniform distribution of the records in the tree, leaves are placed at the $(b-1)^{th}$ and b^{th} levels. As $(2^{b-1}-2)$ leaves are placed at the $(b-1)^{th}$ level, the probability of finding a leaf node at this level is

$$P_l(b-1) = (2^{b-1}-2)/2^{b-1}$$
$$= (1- 2/2^{b-1})$$
$$= (1-2^{2-b}).$$

Similarly, as 2 leaves are placed at the b^{th} level, the probability of finding a leaf node at this level is

$$P_l(b) = 2/2^{b-1}$$
$$= 2^{2-b}.$$

A node at i^{th} level, which finds the leaf node at the $(b-1)^{th}$ level, needs to traverse $(b-i-1)$ internal nodes and a leaf node. At each internal node, 3 comparisons are required:

1. one to check whether the left pointer is null or not,
2. one to check whether the right pointer is null or not, and
3. one to check the balance indicator value.

At the leaf node, 2 comparisons are required:

1. one to check whether the left pointer is null or not, and

2. one to check whether the right pointer is null or not.

Therefore, the cost for a non-leaf node of i^{th} **level** which finds its leaf node at $(b-1)^{th}$ level is $C_i(b-1) = P_{nl}(i) * P_l(b-1) * [(b-i-1)*3 + 2]$.

A node at the i^{th} level which finds the leaf node at the b^{th} level needs to traverse $(b-i)$ internal nodes and a leaf node. Thus, the cost for a non-leaf node at the i^{th} level which finds its leaf node at the b^{th} level is

$$C_i(b) = P_{nl}(i) * P_l(b) * [(b-i)*3 + 2].$$

For the 2 non-leaf nodes at the $(b-1)^{th}$ level, the traversal must end at a leaf node of b^{th} level. Here, 5 comparisons are required:

1. 3 for the non-leaf node and
2. 2 for the leaf node.

Therefore, a cost $C_{b-1}(b) = P_{nl}(b-1) * 5$ is added.

Finally, we get the cost of reordering scheme:

$$R_{avg} = \sum_{i=1}^{b-2} (C_i(b-1) + C_i(b)) + 5 * 2^{2-b}, \text{ where } b \geq 2.$$

For $b = 0$ and 1, two comparisons are necessary for leaf node checking.

Therefore, the cost of deletion of a record is

$$D_{avg} = \begin{cases} S_{avg} + 2, & 0 \leq b \leq 1 \\ S_{avg} + 0.5 * R_{avg} + 1, & b \geq 2 \end{cases}$$

Table 9. Comparison between mathematical analysis and experimental result

Number of records	Deletion	
	Mathematical Analysis (No. of probes)	* Experimental result (No. of probes)
100	6	6
200	6	6
400	11	9
800	12	11
1600	14	14
3200	16	16
6400	19	19
12800	21	22
25600	24	25
51200	27	28
102400	30	31
204800	33	34

The node containing a record in the proposed technique requires a little more memory than separate chaining and dynamic hashing. The extra field *balance_indicator*, however, helps to maintain proper balance in the tree and thus increases efficiency for searching, insertion, and deletion. The facility of using the binary equivalent is that it does not depend on the order in which *key*s arrive, whereas for a usual binary tree, its balance depends on the order of *key*s. Moreover, the association of a binary tree with each hash index enables the technique to handle large numbers of records efficiently. The experimental results in the next section explore the performance of the method.

5 EXPERIMENTAL RESULTS

Analyses have been performed on several sets of data. First, a set of 100 records is inserted, and then an additional 10 records are inserted. The total numbers of comparisons are evaluated; from this, the average number of comparisons for an insertion is found. After this, the same experiment is performed on additional sets of data. The last set contains 204800 records. Experimental results for searching and deletion are also tested in the same manner. The same experiments are performed on the same sets of data after deleting 10% of the records.

Table 10. No. comparisons for successful search

Number of records	Successful Search		
	Separate Chaining	Dynamic Hashing	Proposed Technique
100	2	3	4
200	3	5	4
400	4	6.	6
800	6	8.	7
1600	9	11	9
3200	18	14	11
6400	33	17	14
12800	65	19	17
25600	130	22	20
51200	258	25	22
102400	514	28	25
204800	1026	31	28

Table 11. No. comparisons for unsuccessful search

Number of records	Unsuccessful Search		
	Separate Chaining	Dynamic Hashing	Proposed Technique
100	2	5	4
200	3.	7	6
400	5	9	8
800	9	12	10
1600	17	15	12
3200	33	18	15
6400	65	20	18
12800	129	26	21
25600	257	27	24
51200	513	29	27
102400	1025	32	30
204800	2049	35	31

Table 11. Comparisons for insertion

Number of records	Insertion		
	Separate Chaining	Dynamic Hashing	Proposed Technique
100	2	4	5
200	3	6	6
400	6	9	8
800	11	11	10
1600	19	14	12
3200	36	17	15
6400	71	20	18
12800	142	23	21
25600	283	26	24
51200	564	29	27
102400	1076	32	30
204800	2151	35	33

Table 12. Comparisons for deletion

Number of records	Deletion		
	Separate Chaining	Dynamic Hashing	Proposed Technique
100	2	3	6
200	3	4	6
400	3	6	9
800	6	8	11
1600	10	11	14
3200	17	14	16
6400	34	17	19
12800	66	19	22
25600	130	22	25
51200	257	25	28
102400	512	28	31
204800	1021	31	34

Table 13. No. of comparisons for an unsuccessful search after 10% deletion

Number of records	Successful Search after 10% deletion		
	Separate Chaining	Dynamic Hashing	Proposed Technique
100	2	3	4
200	3	5	4
400	3	6	6
800	5	8	6
1600	9	11	9
3200	16	14	11
6400	30	17	14
12800	59	19	16
25600	117	22	19
51200	232	25	22
102400	463	28	25
204800	926	31	28

Table 14. No. of comparisons for an unsuccessful search after 10% deletion

Number of records	Unsuccessful Search after 10% deletion		
	Separate Chaining	Dynamic Hashing	Proposed Technique
100	2	5	3
200	3	7	6
400	5	9	7
800	8	12	9
1600	15	15	12
3200	30	18	15
6400	59	20	17
12800	116	24	20
25600	231	27	23
51200	462	29	26
102400	923	32	29
204800	1845	35	32

Table 15. No. of comparisons for insertion after 10% deletion

Number of records	Insertion after 10% deletion		
	Separate Chaining	Dynamic Hashing	Proposed Technique
100	2	4	4
200	3	6	6
400	6	9	7
800	10	12	10
1600	17	14	12
3200	33	17	15
6400	65	20	17
12800	129	23	20
25600	257	26	23
51200	513	29	26
102400	1025	32	29
204800	2048	35	32

Table 16. No. of comparisons for deletion after 10% deletion

Number of records	Deletion after 10% deletion		
	Separate Chaining	Dynamic Hashing	Proposed Technique
100	2	3	6
200	3	4	6
400	4	6	9
800	6	8	11
1600	9	11	14
3200	16	14	16
6400	30	17	19
12800	59	19	22
25600	117	22	25
51200	232	25.	28
102400	460	28	31
204800	922	31	34

5.1 Memory utilization

In the proposed technique, each record node requires total 10 bytes, whereas in dynamic hashing, each node requires 8 bytes. After deletion, dynamic hashing does not destroy the node containing the key to be deleted, but the proposed technique does. For x records in a database, the proposed technique requires $10x$ bytes; whereas dynamic hashing requires $8x$ bytes. After the deletion of p% records, memory requirements for both techniques are the same. Therefore,

$$8x = 10x - 10 * \frac{px}{100}$$

$$\Rightarrow \frac{px}{10} = 2x. \quad \text{Therefore,} \quad p = 20.$$

That is, after 20% deletion, the memory requirements for both techniques are same. If more than 20% deletion occurs, the proposed technique requires less memory than dynamic hashing.

Table 17. Memory comparison

Number of records	Separate chaining (in KB)	Dynamic hashing (in KB)	Proposed technique (in KB)
500	3.20	4.13	5.40
1000	6.20	8.13	10.40
2000	12.20	16.13	20.40
5000	30.20	40.13	50.40

Table 18. Memory comparison after 20% deletion

Number of records	Separate chaining (in KB)	Dynamic hashing (in KB)	Proposed technique (in KB)
400	2.60	4.13	4.40
800	5.00	8.13	8.40
1600	9.80	16.13	16.40
4000	24.20	40.13	40.40

Table 19. Memory comparison after 30% deletion

Number of records	Separate chaining (in KB)	Dynamic hashing (in KB)	Proposed technique (in KB)
350	2.32	4.13	3.90
700	4.40	8.13	7.40
1400	8.60	16.13	14.40
3500	21.20	40.13	35.40

6 DISCUSSION AND CONCLUSION

From the experimental results, it is obvious that separate chaining works better for small numbers of records. When the number of records increases, however, its performance degrades drastically. In dynamic hashing, the main problem is that each deletion creates an unused node, thus increasing the cost for accessing a record. In the proposed technique, when a record is deleted, the key is replaced by another key located in a leaf node, and that leaf node is destroyed. Thus, no unused node is ever created in the tree. Also in the proposed technique, an extra field is added to each node in order to prevent the tree from becoming unbalanced when a record is deleted. As a result, the memory

required for the technique is greater than for separate chaining and dynamic hashing. Today, sufficient memory is available, and in the proposed technique, reduction in comparison costs has been given greater importance.

From the experimental results, it is also clear that the proposed technique works better in some cases than in others. In the deletion operation, the proposed technique costs more in comparisons than dynamic hashing due to the reordering scheme. But after deletion, dynamic hashing performs worse than the proposed technique during the insertion and searching operations. The main advantage of the proposed technique over dynamic hashing is that after deletion, the former works better in searching and insertion. But in the case of deletion, dynamic hashing performs better.

From the experimental results, it is also observed that the proposed technique performs well while searching and inserting a record. Except in a few cases, it shows better results than separate chaining and dynamic hashing. When database size increases, the proposed technique's cost of comparison increases at a not very high rate. Though memory utilization in the proposed technique is better than in dynamic hashing, the former requires more memory space. For a large database distributed uniformly, there is no allocated but unused node in this technique. Thus, it can be said that though the proposed technique requires more memory, it shows significant improvements in several cases.

We have assumed the input data to be uniform. However, if the input records are not uniformly distributed, performance of the proposed technique degrades. For example, the greater the difference between the numbers of records having odd and even key values, the more the corresponding tree becomes one-sided and the more the performance degrades. In the worst case, a tree may become linear. The proposed technique has no efficient way to handle sequential key order traversal, which is sometimes important in order to retrieve a range of records.

7 REFERENCES

Langsam, Y., Augenstein, M. J., & Tenenbaum, A. M. (1997) *Data Structures Using C and C++. Second Edition,* India: Prentice-Hall of India Private Limited.

Silberschatz, A., Korth, H. F., & Sudarshan, S. (1997) *Database System Concepts. Third Edition*, Singapore: The McGraw-Hill Companies, Inc..

Folk, J. M., Zoellick, B., & Riccardi, G. (1999) *File Structure – An Object Oriented Approach with C++*, India: Addison-Wesley.

Horowitz, E., & Sahni, S. (1996) *Fundamentals of Data Structures.*, India: Galgotia Book Source.

Cormen, T. H., Leiserson, C. E., & Rivest, R. L (1999) *Introduction to Algorithms*. India: Prentice-Hall of India Private Ltd..

Dale, N., & Lilly, S. C. (1997) *Pascal Plus Data Structures, Algorithms and Advanced Programming. Third Edition*, India: Galgotia Publications Pvt. Ltd.

Kruse, R. L. (1996) *Data Structures and Program Design. Third Edition*, India: Prentice-Hall of India Private Ltd.

.

Larson, P.-Å. (1982) Performance Analysis of Linear Hashing with Partial Expansions. *ACM Transactions on Database Systems* 7 (4), 566-587.

Larson, P.-Å. (1985) Linear Hashing with Overflow-Handling by Linear Probing. *ACM Transactions on Database Systems* 10 (1), 75-89.

Enbody, R. J., & Du, H. C. (1988) Dynamic hashing schemes**.** *ACM Computing Surveys (CSUR*), 20 (2): 50 – 113.

Dietrich, T. M., & DeLillo, N. J. (2003) Implementing Hashing Using Separate Chaining in Java. *CSIS Technical Report*, Pace University, New York, NY, USA.