An Efficient Trie Hashing Method Using a Compact Binary Trie

Masami Shishibori, Makoto Okada, Toru Sumitomo and Jun-ichi Aoe

Department of Information Science & Intelligent Systems
Faculty of Engineering
Tokushima University
2-1 Minami-Josanjima-Cho
Tokushima-Shi 770
Japan

e-mail: {bori, aoe}@is.tokushima-u.ac.jp

Abstract. In many applications, information retrieval is a very important research field. In several key strategies, the binary trie is famous as a fast access method to be able to retrieve keys in order. However, if the binary trie structure is implemented, the greater the number of the registered keys, the larger storage is required, as a result, the binary trie can not be stored into the main memory. In order to solve this problem, the method to change the binary trie into a compact bit stream have been proposed, however, searching and updating a key takes a lot of time in large key sets. This paper proposes the method to improve the time efficiency of each process by introducing a new hierarchical structure. The theoretical and experimental results show that this method provides faster access than the traditional method.

Key words: information retrieval, trie hashing, binary trie, data structures, pre-order bit stream

1 Introducion

In many natural language processing and information retrieval systems, it is necessary to be able to adopt a fast digital search, or trie search for looking at the input character by character. In digital search methods, trie method [1], [2], [3], [4] is famous as one of the fastest access methods, and trie searching is frequently used as a hash table of trie hashing [5] indices in information retrieval systems and dictionaries in natural language processing systems. Although hash and B-tree strategies are based on comparisons between keys, a trie structure can make use of their representation as a sequence of digits or alphabetic characters. A trie can search all keys made up from prefixes in an input string, in only one time scanning, since a trie advances the retrieval character by character, which makes up keys. From this reason, the trie is called the Digital Search-tree (DS-tree). Especially, DS-tree whose nodes have only two arcs labelled with 0 and 1 is called a Binary Digital Search-tree (BDS-tree) [5], [6].

In the case when the binary trie, that is BDS-tree, is implemented as the index of information retrieval application, if the key sets to be stored are large, it is too big to store into main memory. Therefore, it is very important to compress the binary trie into a compact data structure. Then, Jonge et al. [5] proposed the method to compress the binary trie into a compact bit stream, which is called the pre-order bit stream, by traversing the trie in pre-order. However, the bigger the binary trie, the longer the pre-order bit stream is, as a result, the time cost to retrieve keys located toward the end of the bit stream is high.

This paper proposes a new method able to avoid the increase of the time-cost even if the dynamic key sets become very big. The data structures compressed by this method have two distinctive features: (1) they store no pointers and require one bit per node in the worst case, and (2) they are divided into the small binary tries, and their small tries are connected by pointers.

2 A Compact Data Structure for Binary Tries

In the BDS-tree, the binary sequence, which is obtained from the translation of the characters into their binary code, is used as the value of the key, namely, the left arc is labeled with the value '0' and the right arc with the value '1'. If each of leaves in the BDS-tree points the record of only one key, the depth of the BDS-tree becomes very deep. So, each leaf has the address of the bucket, where some corresponding keys to the path are stored. We will use *B_SIZE* to denote the number of keys and their records that can be stored in one bucket. For example, let us suppose that the following key set K is inserted into the BDS-tree.

$$K = \{\text{air, art, bag, bus, tea, try, zoo}\}$$

If the binary sequence, obtained from the translation of the internal code of each character, where internal codes of a, b, c, z are 0, 1, c, 25 respectively, into binary numbers of 5 bits, is used, the corresponding bit strings to be registered are below.

air	→	0/ 8/ 17	→	00000 01000 10001
art	→	0/ 17/ 19	→	00000 10001 10011
bag	→	1/ 0/ 6	→	00001 00000 00110
bus	→	1/ 20/ 18	→	00001 10100 10010
tea	→	19/ 4/ 0	→	10011 00100 00000
try	→	19/ 17/ 24	→	10011 10001 11000
zoo	→	25/ 14/ 14	→	11001 01110 01110

If *B_SIZE* is 2, the corresponding BDS-tree for the key set K is shown in Figure 1. In order to compress the BDS-tree, we applied the particular leaf which does not have any addresses for the bucket. This leaf will be called dummy leaf. Using the dummy leaf, the following advantages are derived. First, it satisfies the property of binary trees that the number of leaves is one more than the number of internal nodes. This property underlies the search algorithm using the compact data structure. Next, if the search terminates in a dummy leaf, the search key is regarded as a key that does not belong to the BDS-tree, and no disk access at all will be needed.

Figure 1: An Example of the BDS-tree.

When the BDS-tree is implemented, the larger the number of the registered keys, the greater the number of the nodes in the tree is, and more storage space is required. So, Jonge et al. [5] proposed the method to compress the BDS-tree into a very compact bit stream. This bit stream is called pre-order bit stream. The pre-order bit stream consists of 3 elements: *treemap*, *leafmap* and *B_TBL*. The *treemap* represents the state of the tree and can be obtained by a pre-order tree traversal, emitting a '0' for every internal node visited and a '1' for every bucket visited. The *leafmap* represents the state (dummy or not) of each leaf and by traversing in pre-order the corresponding bit is set to a '0' if the leaf is dummy, otherwise the bit is set to a '1'. The *B_TBL* stores the addresses of each bucket. Figure 2 shows the pre-order bit stream corresponding to the BDS-tree of Figure 1. Then, in order to understand the relation between the BDS-tree and the pre-order bit stream easily, we indicate above the *treemap* the corresponding internal node and leaf number (in the case of the dummy leaf, the symbol is a "d") within the round "()" and square "[ ]" brackets, respectively.

The search using the pre-order bit stream proceeds bit by bit from the first bit of *treemap*, so that the search is traversed the BDS-tree in pre-order. The search algorithm using the pre-order bit stream is presented below, where it uses the following variables and functions:
s_key: The bit string of the key to be searched.
keypos: A pointer to the current position in *s_key*.
treepos: A pointer to the current position in *treemap*.
leafpos: A pointer to the current position in *leafmap*.
bucketnum: The corresponding bucket number.
SKIP_COUNT(): Skips the left partial tree, and returns the number of the leaf within the partial tree.
FIND_BUCKET(): Returns the corresponding bucket number of *s_key*.
[An Algorithm to search in the BDS-tree]
Input: *s_key*;
Output: If *s_key* can be found, then the output is TRUE, otherwise FALSE;

```
Step(S-1): {Initialization}
    $keypos \leftarrow 1$, $treepos \leftarrow 1$, $leafpos \leftarrow 1$;
```

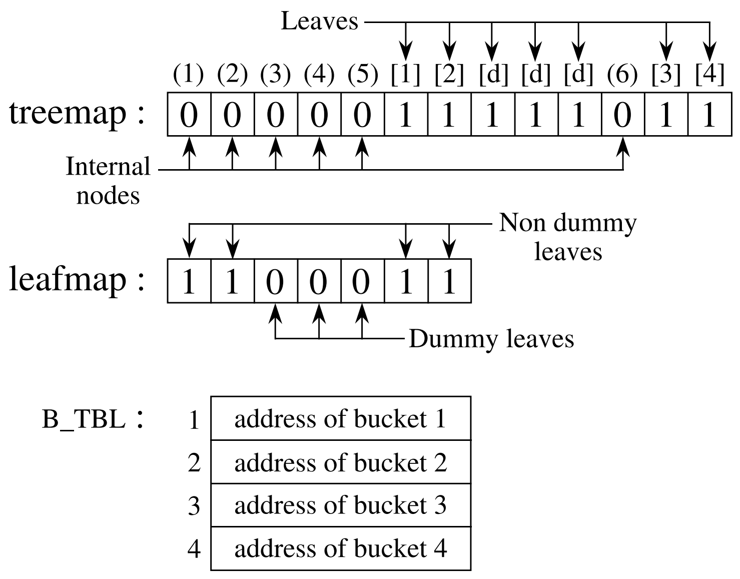


Figure 2: An example of the pre-order bit stream.

Step(S-2): {Skipping the left subtree}
If the bit of *s_key* pointed to by *keypos* is a '1',
then *leafpos* $\leftarrow$ *leafpos*+SKIP_COUNT();

Step(S-3): {Advance to the right subtree}
keypos $\leftarrow$ *keypos*+1; *treepos* $\leftarrow$ *treepos*+1;

Step(S-4): {Loop invariant until reaching the leaf}
If the bit of *treemap* pointed to by *treepos* is a '0', return to Step(S-2);

Step(S-5): {Verification of *leafmap*}
If the bit of *leafmap* pointed to by *leafpos* is a '0', FALSE is returned;

Step(S-6): {Verification of *B_TBL*}
bucketnum $\leftarrow$ FIND_BUCKET();
If the bucket indicated by *bucketnum* contains the key, return TRUE, otherwise return FALSE;

Regarding the above algorithm, since a left subtree in *treemap* is represented following the 0 bit of its parent node, when advancing to the left subtree, the Step(S-2) is not executed, however when advancing to the right subtree, the Step(S-2) to skip the left subtree is added. This skipping process utilizes the binary tree's property that the number of leaves is one more than the number of internal nodes in any binary subtree. Using this property, the function SKIP_COUNT() can search for the end position of the left subtree and get the number of leaves in the left subtree. Namely, this function advances *treepos* until the number of 1 bits is one more than the number of 0 bits, and returns the number of 1 bits (leaves). Moreover, the value obtained by counting the number of 1 bits in *leafmap* from the first bit to the one pointed to by *leafpos* indicates which slot in *B_TBL* contains the required bucket address.

For example, to retrieve key="zoo" (*s_key*="11c") in Figure 2, the following steps are performed:

Step(S-1): *keypos*=*treepos*=*leafpos*=1; Since the first bit of *s_key* is a '1', the subtree whose root is node 2 is skipped by SKIP_COUNT().

Step(S-2): *leafpos*=*leafpos*+SKIP_COUNT()=6;

Step(S-3): *keypos*=2; *treepos*=11;

Step(S-4): Since the 11-th bit of *treemap* is a '0', return to Step(S-2);

Step(S-2‘4): Since the 2-th of *s_key* is a '1', the subtree whose root is node 6 is skipped; *leafpos*=*leafpos*+SKIP_COUNT()=7; *treepos*=13;

Step(S-5): Since the 7-th bit of *leafmap* is a '1', *B_TBL* is verified;

Step(S-6): Since key "zoo" is stored in the bucket 4, TRUE is returned;

3 Improvement by Using Hierarchical Structures

The BDS-tree represented by the pre-order bit stream is a very compact binary trie, however, the more keys are stored in the tree, the longer the bit strings (*treemap* and *leafmap*) are. As a result, the time-cost for each process is high. For example, as for the retrieval, the worst case is when search process is done toward the rightmost leaf in the BDS-tree as shown in Figure 3. In this case, if the rightmost leaf keeps the address of the bucket of the searching key, all bits in *treemap* (*leafmap* also) of the pre-order bit stream must be scanned. Similarly, in the case when an arbitrary key is inserted in the bucket corresponding to the leftmost leaf, suppose the bucket is divided and merge, all bits after the bit corresponding to the leftmost leaf in *treemap* of the pre-order bit stream have to be shifted. In this paper, the method to solve the problem stated above is proposed.

This method separates the BDS-tree into smaller BDS-trees of a certain depth. This depth is called the *separation depth*, and these small trees are called *separated trees*. These separated trees are numbered and connected by pointers. The BDS-tree separated in this way is called a Hierarchical Binary Digital Search tree (HBDS-tree). The HBDS-tree obtained based on the BDS-tree of Figure 4 -(a), with a separation depth of 2, is shown in Figure 4 -(b). In this case when rightmost leaf is searched, if we use the BDS-tree as shown in Figure 4 -(a), all internal nodes and leaves must be scanned in pre-order traversal. On the other hand, in the case of the HBDS-tree as shown in Figure 4 -(b), we can search the rightmost leaf by scanning all nodes and leaves of the only separated tree 1.

The algorithm to retrieve a key in the HBDS-tree uses the pre-order bit stream. The binary sequence H(k) of the key is divided into the following binary sequence:

$$H(k) = H_1(k)\ H_2(k)\ c\ H_j(k)\ c\ H_n(k)$$

Supposing that the separation depth is denoted by L, the lengths of $H_1(k)$‘$H_{n-1}(k)$ are L bits and the length of $H_n(k)$ is less than L bits. The HBDS-tree can be compressed into a very compact data structure named the pre-order bit stream as well as the BDS-tree. The pre-order bit stream is created and controlled for each of the separated trees. The pre-order bit stream that corresponds to the *i*-th separated tree in the

Figure 3: Retrieval of the BDS-tree in the worst-case.

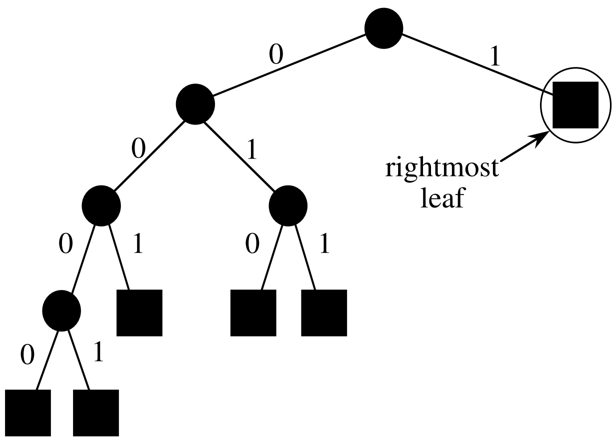
HBDS-tree consists of $treemap_i$, $leafmap_i$ and B_TBL_i, but the leaf which becomes the pointer to the next separated tree is regarded as a especial leaf and B_TBL_i contains the number of the next separated tree preceded by a minus sign in the slot corresponding to the leaf. The HBDS-tree obtained based on the BDS-tree of Figure 1, with a separated depth of 2, is shown in Figure 5, and the pre-order bit stream for the HBDS-tree of Figure 5 is shown in Figure 6, where, as can be seen above the *treemap*, the leaves which became the pointer to the separated tree are marked by "$\langle\rangle$". By using this improved method, each process can be sped up, because unnecessary scanning of the pre-order bit stream for each separated tree can be omitted.

The algorithm for retrieval of the HBDS-tree represented by the pre-order bit stream is shown below, where it uses the following variables:
i: The current separated tree number.
s_key: The key to be searched.
$keypos$: A pointer to the current position in s_key.
$treepos$: A pointer to the current position in $treemap_i$.
$leafpos$: A pointer to the current position in $leafmap_i$.
$bucketnum$: The corresponding bucket number.
Moreover, each of the functions performs the same process as the functions explained in Section 2 toward the i-th separated tree, when i is initialized with 1.

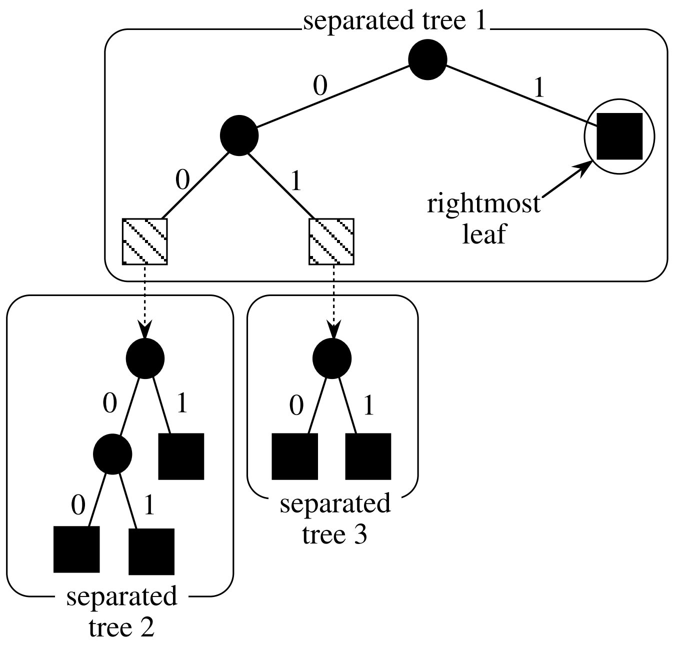
[An Algorithm to search in the HBDS-tree]
Input: s_key;
Output: If s_key can be found, then the output is TRUE, otherwise FALSE;

Step(S'-1)∼Step(S'-5): The same procedures as the Step(S-1)‘Step(S-5) are performed, however their $treemap$, $leafmap$ are changed into $treemap_i$, $leafmap_i$;

Step(S'-6): {Verification of $bucketnum$}



(a) **An example of the BDS-tree.**



(b) **An example of the HBDS-tree.**

Figure 4: Improvement of the BDS-tree by using hierarchical structures.

```
    $bucketnum \leftarrow$ FIND_BUCKET($i$);
    If $bucketnum \leq 0$, proceed to Step(S'-7), otherwise proceed to Step(S'-8);

Step(S'-7): {Obtaining the separated tree number}
    $i = -1 \times bucketnum$; Return to Step(S'-1);

Step(S'-8): {Verification of $B\_TBL$}
    If the bucket indicated by $bucketnum$ contains the key, return TRUE, otherwise
    return FALSE;
```

For example, in the case of retrieval the key = zoo (s_key = "11c") in the pre-order bit stream of the HBDS-tree as shown in Figure 6, s_key can be retrieved in the HBDS-tree by using the pre-order bit stream of the only separated tree 1, so that the time-cost of retrieval becomes better than the case by using the BDS-tree's one.

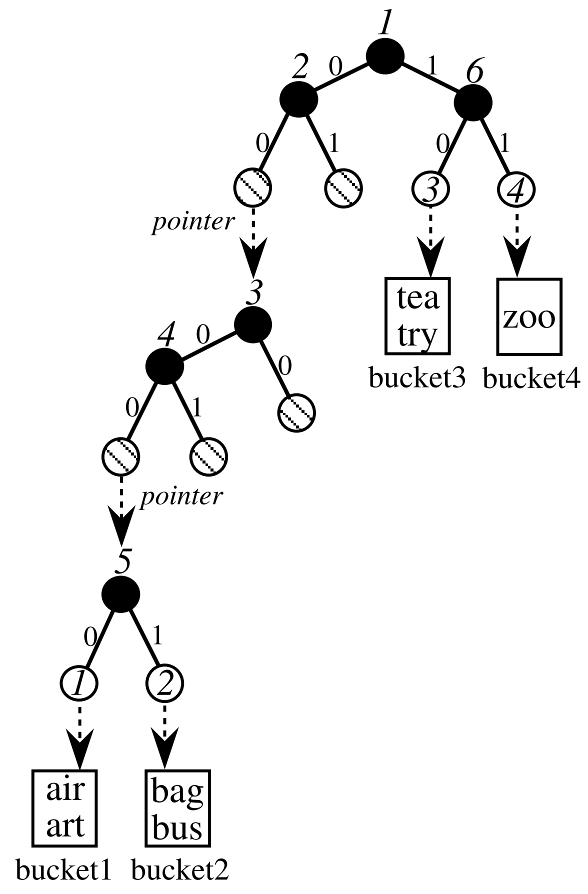


Figure 5: HBDS-tree based of the BDS-tree of Figure 1.

4 An Insertion Algorithm

The method for inserting the new key into the HBDS-tree is divided into the following three cases as well as the BDS-tree.
1) the required bucket is partially filled.
2) the required bucket is a dummy bucket.
3) the required bucket is full.
In this chapter, the third case, when the required bucket is full, that is, the method for dividing the full bucket into the new two buckets is explained. An explanation of the other cases is omitted, because they are very simple.

When there is an overflow in the required bucket, in the BDS-tree, the following processes are repeated until the overflow of the bucket does not happen. First, the corresponding leaf to the full bucket is changed into a tree which consists of a node and two dummy leaves. This tree is called a unit tree. Next, all the keys in the full bucket and an insertion key are distributed between the corresponding two buckets to dummy leaves of the unit tree. On the HBDS-tree, when the unit tree is made, a new separated tree must be created every time the depth of each separated tree exceeds the separation depth. As for the insertion process which uses the pre-order bit stream, a bit line "011", which represents the unit tree in *treemap*, and a bit line "00", which represents the two dummy leaves of the unit tree in *leafmap*, are inserted into *treemap* and *leafmap* respectively.

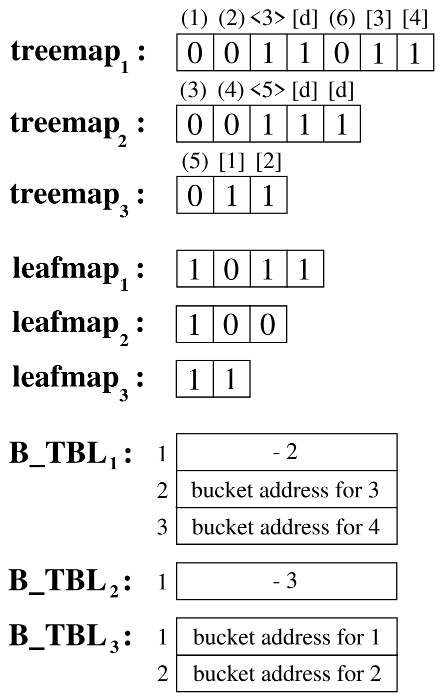


Figure 6: The pre-order bit stream for the HBDS-tree of Figure 5.

5 Evaluation

5.1 Theoretical Evaluation

In this section, the worst-case time complexities of each algorithm for the BDS-tree and HBDS-tree are theoretically analyzed. And the space complexities of each pre-order bit stream for the BDS-tree and HBDS-tree also are calculated. Let the tree structure to be analyzed be the complete tree. The following parameters are used:
n: The depth of the complete tree;
m: The separation depth;
α: The number of layers in the HBDS-tree. It is obtained by $[n/m]$, where $[n/m]$ indicates the minimum integer greater than or equal to n/m;

As for the time complexity, the worst-case time complexity for retrieval for the BDS-tree is $O(2^n)$, because the whole of the complete tree must be scanned. However, for the HBDS-tree it is $O(\alpha 2^m)$, since only α separate trees are scanned. Regarding the insertion and deletion, the worst case is when each process is done toward the leftmost bucket in the tree. In this case, suppose the bucket is divided and merged, the BDS-tree has a time complexity $O(2^n - n)$, because all bits after the bit corresponding to the bucket in the pre-order bit stream have to be shifted, however for the HBDS-tree it is $O(2^m - m)$, because the same operations are performed toward only one separated tree. Generally, for $n \ll 2^n$ and $m \ll 2^m$, the worst-case time complexity for insertion and deletion in the BDS-tree is $O(2^n)$ and for the HBDS-tree it is $O(2^m)$.

As for the space complexity, on the BDS-tree, the number of bits used for the *treemap* is equal to the total number of nodes (internal nodes and leaves) of the complete tree, that is, it is $2^{n+1} - 1$. And the leafmap needs 2^n bits which is the number of leaves in the complete tree. As for the sizes of the *treemap* and $leafmap$

for the HBDS-tree, they are calculated as shown below :
Number of bits required for *treemap*

$$= \text{(number of all nodes of the separated tree)} \times \text{(number of the separated trees within the complete tree)}$$
$$= \sum_{k=0}^{m} 2^k \times \sum_{k=1}^{\alpha} 2^{m(k-1)} = (2^{m+1}-1)\frac{2^{m\alpha}-1}{2^m-1}$$
$$= \{2(2^m-1)+1\}\frac{2^{m\alpha}-1}{2^m-1} = (2^{m\alpha+1}-2) + \frac{2^{m\alpha}-1}{2^m-1}$$
$$= (2^{n+1}-1) + \frac{2^n-1}{2^m-1} - 1$$

Number of bits required for *leafmap*

$$= \text{(number of leaves of the separated tree)} \times \text{(number of the separated trees within the complete tree)}$$
$$= 2^m \times \sum_{k=1}^{\alpha} 2^{m(k-1)} = 2^m \frac{2^{m\alpha}-1}{2^m-1}$$
$$= (2^m-1+1)\frac{2^{m\alpha}-1}{2^m-1} = (2^{m\alpha}-1) + \frac{2^{m\alpha}-1}{2^m-1}$$
$$= 2^n + \frac{2^n-1}{2^m-1} - 1$$

From the above results, if the BDS-tree is separated, the storage requirement for both the treemap and the leafmap increases only $(2n-1)/(2m-1)-1$ bits.

5.2 Experimental Evaluation

This method was written in about 2,000 lines of code in C, and implemented on a Sun Microsystems Sparc Station 2 (28 MIPS).

Key sets	Japanese nouns		English words	
Kinds of trees	BDS-tree	HBDS-tree	BDS-tree	HBDS-tree
Number of				
non_dummy leaves	6,002		6,159	
dummy leaves	3,649		8,411	
Internal nodes	9,650		14,569	
depth	82		70	
separated tree	2,060		2,940	
Time (Second)				
Registration	870	146	1875	164
Time (Milli-Second)				
Retrieval	8.68	0.48	11.26	0.56
Insertion	38.00	3.00	37.50	3.28
Storage (K-byte)				
treemap	2.41	2.67	3.64	4.00
leafmap	1.21	1.46	1.82	2.19
B_TBL	12.00	16.12	12.32	18.20

Table 1: Experimental results.

In order to observe the effect of this method, we compare the cost time of each process and storage requirement for the BDS-tree and the HBDS-tree. 50,000 nouns in Japanese and 50,000 English words with an average length of 6 and 9 bytes respectively are used as the key sets. Table 1 shows the experimental results for the each of key sets, where the separation depth is 5 and *B_SIZE* is 16. Retrieval time is the average time required for a key when all registered keys are searched and deleted, respectively. Insertion time is the average time required for a key when 1000 unregistered keys are added to the key set. Storage in Table 1 shows the memory required for the registration of the each key set.

From the experimental results, the retrieval in the HBDS-tree is 18‘20 times faster than in the BDS-tree, the insertion is 11‘13 times faster. Thus, it can be concluded that the time each of the processes requires is significantly less when using this method. As for the storage space required by the HBDS-tree, the sizes of *treemap*, *leafmap* and *B_TBL* are 1.11, 1.21 and 1.34 times the size of the ones used by the BDS-tree. However, by nature, the pre-order bit stream is very compact in size, thus their sizes are good enough for practical applications. Moreover, for the BDS-tree and the HBDS-tree, both represented by the pre-order bit stream, the storage requirement to register one key is of 2.50 and 3.24 bits, respectively. Thus, these methods can be operated with more compact storage than the B-tree, B^+-tree, etc.

6 Conclusions

The Binary trie represented by the pre-order bit stream can search a key in order, however, the time-cost of each process becomes high for large key sets. So, the method for solving the above problem by separating the tree structure has been presented in this paper. The time and space efficiency of the proposed method is theoretically discussed, and the validity of this method has been supported by empirical observations. As future improvements, an efficient method to improve the space efficiency of the bucket should be designed.

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, “Data Structures and Algorithms”, Reading, Mass.: Addison-Wesley, pp. 163–169, 1983.

[2] J. Aoe, “An Efficient Digital Search Algorithm by Using a Double-Array Structure”, *IEEE Trans. Software. Eng.*, Vol. 15, No. 9, pp. 1,066–1,077, 1989.

[3] J. Aoe, “Computer Algorithms-Key Search Strategies”, IEEE Comput. Society Press, 1991.

[4] G.H. Gonnet, “Handbook of Algorithms and Data Structures”, Addison-Wesley, Reading Mass. Ch. 3 (Searching Algorithms), pp. 25–147, 1984.

[5] W. D. Jonge, A.S. Tanenbaum and R.P. Reit, “Two Access Methods Using Compact Binary Trees”, *IEEE Trans. Software. Eng.*, Vol. 13, No. 7, pp. 799–809, 1987.

[6] M. Shishibori, S. Kiyohara and J. Aoe, “Improvement of Binary Digital Search (BDS)-Trees Using Hierarchical Structures”, in Japanese, *Trans. IEICE*, Vol. J79-D-I, No. 2, pp. 79–87, 1996.