PERGAMON

Information Processing and Management 38 (2002) 221–236

INFORMATION PROCESSING & MANAGEMENT

www.elsevier.com/locate/infoproman

A dynamic construction algorithm for the Compact Patricia trie using the hierarchical structure

Minsoo Jung *, Masami Shishibori, Yasuhiro Tanaka, Jun-ichi Aoe

Department of Information Science and Intelligent Systems, Faculty of Engineering, Tokushima University, 2-1 Minami Josanjima-Cho, Tokushima-Shi 770-8506, Japan

Received 11 October 2000; accepted 22 March 2001

Abstract

We need to access objective information efficiently and arbitrary strings in the text at high speed. In several key retrieval strategies, we often use the binary trie for supporting fast access method in order. Especially, the Patricia trie (Pat tree) is famous as the fastest access method in binary tries, because it has the shallowest tree structure. However, the Pat tree requires many good physician storage spaces in memory, if key set registered is large. Thereby, an expense problem happens when storing this trie to the main storage unit. We already proposed a method that use compact bit stream and compress a Pat tree to solve this problem. This is called Compact Patricia trie (CPat tree). This CPat tree needs capacity of only a very few memory device. However, if a size of key set increases, the time expense that search, update key increases gradually. This paper proposes a new structure of the CPat tree to avoid that it takes much time in search and update about much key set, and a method to construct a new CPat tree dynamically and efficiently. This method divides a CPat tree consisting of bit string to fixed depth. In addition, it compose been divided CPAT tree hierarchically. A construction algorithm that proves this update time requires alteration of only one tree among whole trees that is divided. From experimental result that use 120,000 English substantives and 70,000 Japanese substantives, we prove an update time that is faster more than 40 times than the traditional method. Moreover, a space efficiency of memory increases about 35% only than the traditional method. © 2001 Published by Elsevier Science Ltd.

Keywords: Information retrieval; Access method; Data structure; Patricia trie; Hierarchical structure

* Corresponding author. Tel.: +81-88-656-7495; fax: +81-88-623-2761.

E-mail addresses: minsoo@is.tokushima-u.ac.jp (M. Jung), bori@is.tokushima-u.ac.jp (M. Shishibori), aoe@is.tokushima-u.ac.jp (J.-i. Aoe).

1. Introduction

The use of web pages, on-line document, electronic books, newspapers and so on, which we can retrieve directly from the internet using computers, is increasing more and more every day. In order to make the best use of the potential information which these texts contain, we need to access objective information efficiently (Sato, Sugihara, & Yoshitake, 1999). Moreover, in many natural language processing and information retrieval systems, it is necessary to be able to adopt a fast digital retrieval or trie retrieval for looking at the input character by character.

Among the digital search methods, a trie method is famous as one of the fastest access method (Aho, Hopcroft, & Ullman, 1983; Aoe, 1989; Aoe, 1991; Gonnet, 1984). Trie retrieval uses natural language processing systems (Aoe, Morimoto, Shishibori, & Park, 1996), information retrieval systems (Blumer, Blumer, Haussler, & Mcconnel, 1987) and index of dictionaries often as hashing a trie's hash table (Jonge, Tanenbaum, & Reit, 1987; Litwin, Roussopolulos, Levy, & Hong, 1991). Although a hashing and B tree strategies base in comparisons between keys, a trie structure is using their indications as sequence of digits or alphabetic characters. A trie can search all keys that were made from prefixes of string by scanning only once, since a trie advances the retrieve character by character.

A trie whose nodes have only two arcs (links) labelled with '0' and '1' are called a binary trie (Gonnet, Baeza-Yates, & Snider, 1992). In this binary trie, the binary sequence obtained by transforming each character of the key string into the binary code is utilised as the key to be registered into the trie. In the case that the binary trie is implemented as the index of information retrieval applications, if the key sets to be stored become large, it is too big to store into the main memory. Then, Jonge et al. (1987) proposed the method to compress the binary trie into a compact bit stream, which is called the Compact Binary trie (CB tree). However, when the binary trie is compressed into the CB tree, in order to guarantee that all nodes have two leaves, this method adopts the imaginary leaf, which is called a dummy leaf, for the nodes that have only one leaf. The number of dummy leaves has a bad influence on the size of the CB tree. That is, there are a number of many dummy leaves, a capacity of memory which CB tree occupies increases.

In order to solve this problem, we focussed on the Patricia trie (Pat tree). Pat tree does not have any single descendant nodes, which have only one arc in binary tree, this tree can compress by *Treemap* that dummy leaf does not exist. Manber and co-workers (Gonnet et al., 1992; Manber & Baeza-Yates, 1991; Manber & Myers, 1990) proposed the method to compress the Patricia BDS-tree into one array structure, which is called a PAT array. This PAT array is created by mapping the number of each external node of the Patricia BDS-tree into the array in order, and the key is searched by doing the binary search on the PAT array. Certainly, the PAT array is a compact data structure, however many disk accesses are required for the key retrieval. In addition, we proposed algorithm that changes to Pat tree from CB tree. This Pat tree does not have dummy leaves, and call this, that is, Compact Patricia trie (CPat tree). The CPat tree has very compact tree structure. However, if bit length of this CPat tree becomes long, it requires much time expense that search keys situated on part of end of bit stream. At operation of insertion or delete key, it needs much time because much of a large quantity of bit strings for next time operation should be moved.

This paper proposes modified a Pat tree's structure that can avoid the fact that time efficiency is worsened even if key set is established very greatly. This method separates from a CPat tree to small tree and composes hierarchically. The speed of operation in each small tree detached is fast.

On this account, it can extract unnecessary scanning for each tree detached in CPAT tree. This new tree structure is called the Hierarchical Compact Patricia trie (HCPat tree). For the dynamic key set, a composition method of HCPat tree is different from hierarchical structure of formality CB tree. Usually, adding new node on last part of node recognises key insertion of formality CB tree. On the other hand, after information that corresponds to eliminate node is stored in internal node of Pat tree, if eliminated node's information is changed by addition of new key, all child nodes of internal node that correspond to eliminated node should be modified. Because such method requires much expense of time, we propose a method that improves operation time of Pat tree to solve this problem.

We describe a new data structure that improves in next time sections in detail. Section 2 describes in detail about the organisation method of Pat tree. After that, Section 3 compress method from Pat tree to CPat tree, indeed in other words, explains method that compose of CPat tree that ungues dummy leaf. Section 4 explains about composition method of HCPat tree and insertion algorithm that improve, and Section 5, estimates insertion algorithm that establish and improve various key sets. Finally, summary of our result and about the future works are discussed in Section 6.

2. The Patricia trie

In order to retrieve the key correctly in the CB tree, the dummy leaves must be adopted for nodes that have only one leaf (Jonge et al., 1987). These dummy leaves have no influence on the space efficiency of the secondary memory because they do not have any pointers to the bucket, however, they have a bad influence on the binary trie. In order to solve this problem, we have proposed an algorithm, to apply the compression method proposed by Jonge to the Pat tree (Morrison, 1968). The Pat tree is all nodes that have one node are eliminated, and, to avoid false matches, each node of the Pat tree must have either the counter for the number of eliminated nodes or a pointer of record storing eliminated symbols. In the case that the number of eliminated nodes is stored into each node, every time each node is traversed to retrieve a key, the binary sequence of the key must be skipped over by stored number before the next inspection. Then again, after the retrieval terminates at the leaf, the comparison of the bit pattern between the key string and the string pointed to by the pointer in the leaf must be made. On the other hand, in the case that each node has the pointer to the eliminated symbols, the comparison between the corresponding binary sequence and binary values pointed to by the pointer must be made every time each node is passed, hereafter we suppose that each node stores the number of eliminated nodes as the secondary information. Table 1 shows each binary sequence of the key set K. These binary sequences are registered into the Pat tree. Fig. 1 indicates the corresponding Pat tree to the key set K.

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

3. The Patricia trie represented by a Compact Binary tree

All nodes in the Pat tree are always guaranteed to have two leaves without the dummy leaf and this Pat tree can be transformed into the CB tree of bit string base. As a result, *Leafmap* of the CB

Table 1
Binary sequences of a key set K

Keys	Internal codes	Binary sequences
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

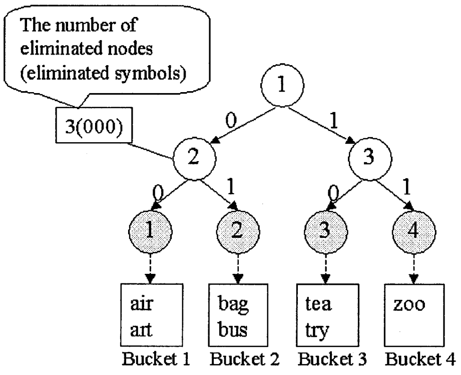

Fig. 1. Pat tree based on the binary trie.

tree becomes unnecessary, furthermore also the number of bits of *Treemap* decreases by the double number of dummy leaves in the CB tree. As for the time efficiency, time-cost of each process also becomes better than the CB tree's one, since the size of the tree becomes small by applying the CB tree to the Pat tree. The information about eliminated nodes included into the Pat tree, however, must be represented as the compact data structure. This method can represent this information as the compact bit stream.

A new CB tree made up by this method consists of *Treemap*, B_TBL and *Nodemap* instead of *Leafmap*. *Treemap* and B_TBL are obtained in the same way as Jonge's method, however, the bit length of *Treemap* is shorter by double the number of dummy leaves. *Nodemap* indicates whether the corresponding node includes the number of eliminated nodes or not. *Nodemap* is obtained by traversing the tree in pre-order, then if the internal node stores the number of eliminated nodes, emit '1' the same times as the number of eliminated nodes, after that emit '0' one time. The new CB tree created by the above method is called a CPat tree.

Fig. 2 shows the corresponding CPat tree to the Pat tree of Fig. 1. In order to show the state of each internal node easily, the corresponding internal node number and the eliminated node sign that is the symbol 'e' are shown within the round '( )' above *Nodemap*. As for *Treemap*, the important thing to note is that the length of *Treemap* is double the number of dummy leaves shorter than the one of the CB tree, namely, the more dummy leaves are included into the CB tree, the shorter the *Treemap* of the CPat tree is. As for *Nodemap*, since the node 1 is the root node that

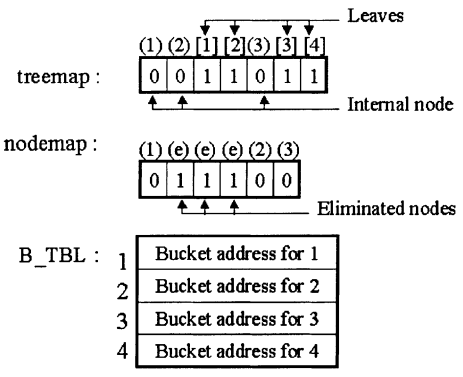

Fig. 2. The CPat tree based on the Pat tree of Fig. 1.

does not have any eliminated nodes, the first bit of *Nodemap* becomes '0'. Next, since the node 2 has the number of eliminated nodes and the number is 3, bits from 2nd to 4th are set to '111'.

4. The hierarchical Patricia trie represented by Compact Binary tree

4.1. An improvement of the Patricia trie by using the hierarchical structures

We know already know of Pat array and CPat tree which has compact data structure among Patricia family. PAT array is created by mapping the number of each external node of the Patricia BDS-tree into the array in order, and the key is searched by doing the binary search on the PAT array. Certainly, the PAT array is a compact data structure, however many disk accesses are required for the key retrieval. It turns out that it is not necessary to simulate the Patricia tree for searching, and it obtains an algorithm which is $O(\log n)$ instead of $O(\log^2 n)$ for retrieval operations ($n =$ number of external node). Actually, searching becomes more uniform. It can be implemented by doing an indirect binary search over the array with the results of the comparisons being less than, equal to (or included in the case of range searching), and greater than. In this way, the searching takes at most $2\log_2 n - 1$ comparisons and $4\log_2 n$ disk accesses (Gonnet et al., 1992). Moreover, CPat tree has been consisting of compact data structure of binary tree base. Trie in some does not have dummy leaves, but it occurs as problem that length of bit string becomes overlong in case much key set is registered. As a result, efficiency is grown worse at process that retrieval, registration is executed in case location of key is situated on last of bit string. Much time is cost because it examines all locations of bit arrangement at the same time search of bit to move to last of bit string. Much time is cost because it must verify all locations of bit arrangement at the same time search of bit to move to last of bit string.

We propose a new Pat tree that proves time-cost to reduce this time expense. This Pat tree has been consisting of tree separated of fixed depth, and each tree detached has been linked to each other by pointer. Fig. 3 shows a Pat tree that uses key set K and composes hierarchically. To appear briefly, a depth of tree separated is two and size of bucket is limited by 1. Pat tree composed of this that initiates HPat tree to do. This Fig. 3 can acquire from the binary sequences of

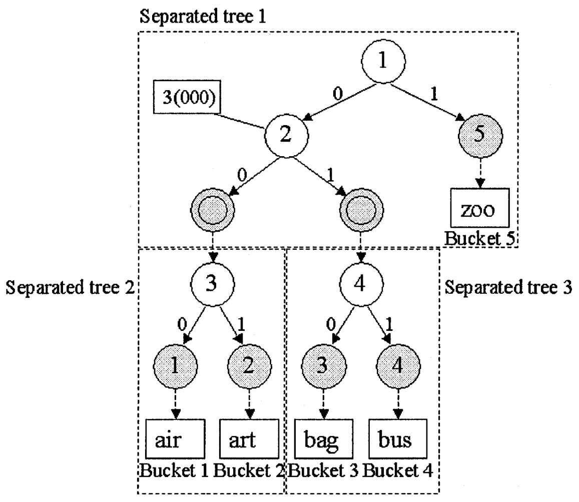

Fig. 3. The hierarchical Pat tree based on the Pat tree of Fig. 1.

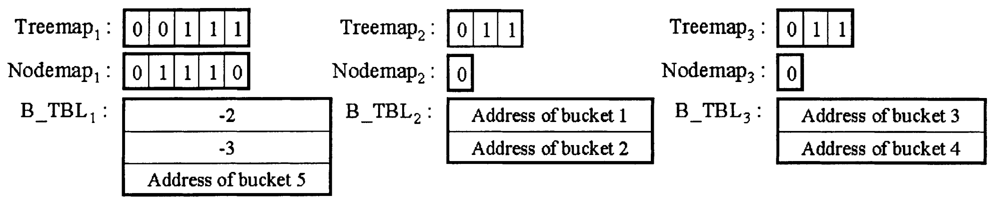
Treemap1 : 0 0 1 1 1
Nodemap1 : 0 1 1 1 0
B_TBL1 : -2 / -3 / Address of bucket 5
Treemap2 : 0 1 1
Nodemap2 : 0
B_TBL2 : Address of bucket 1 / Address of bucket 2
Treemap3 : 0 1 1
Nodemap3 : 0
B_TBL3 : Address of bucket 3 / Address of bucket 4

Fig. 4. The hierarchical Pat tree by the CB tree.

Table 1 and can be expressed by CB tree. This Hierarchical Compact Pat tree is called HCPat tree. Fig. 4 shows a HCPat tree that composes HPat tree of Fig. 3 to basis.

4.2. A retrieval algorithm for the HCPat tree

HCPat tree's retrieval algorithm uses a method such as hierarchic CB tree. Only, HCPat tree is consisted of tree that depth is thinner than hierarchical CB tree, because HCPat tree does not have dummy leaf. Next processing is retrieval order of the HCPat tree.

num: the number of trees detached that is during current retrieval operation.
keypos: the current position in bit strings of key.
treepos: the current position in $Treemap_{num}$
nodepos: the current position in $Nodemap_{num}$

```
Step 1: {initialising}
    keypos → 1, treepos → 1, nodepos → 1, num → 1;
Step 2: {verification of the bit value of the key}
    If the bit of the key pointed to by keypos is '1', proceed to step 3, otherwise to step 5;
Step 3: {skipping the left sub tree in $Treemap_{num}$}
    Advance treepos until the number of '1' bits in $Treemap_{num}$ is one more than the number of '0'
    bits, and proceed to step 4;
```

Step 4: {skipping the left sub tree in $Nodemap_{num}$}
Advance *nodepos* until ‘0’ bits in $Nodemap_{num}$ are passed the same times as the number of ‘0’ bits skipped in $Treemap_{num}$ at step 3, and process to step 5;

Step 5: {traversing an arc}
Advance *treepos* by one, and if the bit of $Treemap_{num}$ pointed to by *treepos* is ‘0’ (internal node), return to Step 6 after advancing *keypos* by 1, otherwise proceed to step 7;

Step 6: {verification whether the corresponding node stores eliminated nodes or not}
If the bit of *Nodemap* pointed to by *nodepos* is ‘1’, advance *nodepos* and *keypos* until the bit of $Nodemap_{num}$ pointed to by *nodepos* is ‘0’, and return to step 2, otherwise only return to step 2;

Step 7: {acquisition of the corresponding bucket address from B_TBL}
Count the number of 1 bits in $Treemap_{num}$ from the first bit to *treepos*, and obtain the bucket address from the slot of B_TBL indicated by the counted value;

Step 8: {Verification of an acquired bucket address}
If address of bucket that acquires is a negative number, proceed to step 9. Otherwise, proceed to step 10.

Step 9: {Acquisition of address of separated tree for next proceed}
To advance in separated tree next proceed, exchange a value of address that acquires to plus value, and store “num”. And, after do to initialise *treepos* and *nodepos* by 1, turn back around by step 2.

Step 10: {retrieval of the key within the corresponding bucket}
If the bucket indicated by the obtained bucket address contains the key, this process is a success, otherwise it is found that the key is unregistered.

Next processing is an example of retrieval for “art” key. As shown in Fig. 4, “art” key is able to retrieve through two-separated tree (tree 1,tree 3) of HCPat tree.

[A retrieval order of key “art”]

1. *Step* 1: Num = 1, *treepos* = 1, *nodepos* = 1, *keypos* = 1.
2. *Step* 2: Advance by step 5 because bit value that corresponds to *keypos*(= 1) in “Art” is 0.
3. *Step* 5: Advance by step 6 because $Treemap_1$’s bit value that corresponds to *treepos*(= 1) is 0. On this time, *Treepos* increases by 2, and *keypos* increases by 2.
4. *Step* 6: Return by step 2 because $Nodemap_1$’s bit value that corresponds to *nodepos*(= 1) is 0. On this time, *nodepos* increases by 2.
5. *Step* 2: Advance by step 5 because bit value that corresponds to *keypos*(= 2) in “Art” is 0.
6. *Step* 5: Advance by step 6 because $Treemap_1$’s bit value that corresponds to *treepos*(= 2) is 0. On this time, *Treepos* increases by 3, and *keypos* increases by 3.
7. *Step* 6: Return by step 2 because $Nodemap_1$’s bit value that corresponds to *nodepos*(= 2) is 1. On this time, *nodepos* increases by 5, and *Keypos* increases by 5.
8. *Step* 2: Advance by step 5 because bit value that corresponds to *keypos*(= 5) in “Art” is 0.
9. *Step* 5: Advance by step 7 because $Treemap_1$’s bit value that corresponds to *treepos*(= 3) is 1. On this time, *Treepos* increases by 4, and *keypos* increases by 6.
10. *Step* 7: *Treemap* 1 from 1 to *treepos*(= 3), “1” bit one exist.
11. *Step* 8: Value of first bucket “B_TBL_1 [1]” is 2, and move by step 9.
12. *Step* 9: Index of tree that is demarcated next time is 2 (Num = 2), because value of bucket is minus. And, return by step 2 after is initialised (*treepos* = 1, *nodepos* = 1).

13. *Step* 2: Advance by step 3 because bit value that corresponds to *keypos*(= 6) in "Art" is 1.
14. *Step* 3: *Treepos* amounts to 3 because skip left sub-tree. Advance by step 4.
15. *Step* 4: *Nodepos* amounts to 1 because skip left sub-tree. Advance by step 5.
16. *Step* 5: Advance by step 7 because *Treemap*$_2$'s bit value that corresponds to *treepos*(= 3) is 1. On this time, *Treepos* increases by 4, and *keypos* increases by 7.
17. *Step* 7: *Treemap*$_2$ from 2 to *treepos*(= 4), "1" bit two exist.
18. *Step* 8: Value of second bucket "B_TBL$_2$ [2]" is 2, and Advance by step 10.
19. *Step* 10: 'art' key has been registered to bucket B_TBL$_2$ [2], therefore enquiry is success.

The retrieval and deletion algorithms of the HCPat tree are very simple, and they are almost same algorithms as the hierarchical CB tree's one. However, the insertion algorithm has some problems. In the next section, we will discuss this insertion algorithm's problem.

4.3. An insertion algorithm for the HCPat tree

4.3.1. An insertion algorithm of the hierarchical CB tree

HCPat tree has a problem such as lower part when insertion is executed. When a new key is inserted, a new node is added in external node only in hierarchical CB tree. But, in the case of HCPat tree, case that new node is added to interior node happens. To solve this insertion problem, we compared insertion algorithm of HCPat tree and hierarchic CB tree. First, in hierarchic CB tree, case of next two happens when new key is added. These two cases happen only in external node.

1. The case that the bucket is partially full.
2. The case that the bucket is full.

In the case of first, the corresponding bucket to the insertion key is searched justly, and this bucket has smaller size than the maximum size of bucket. In this case, it is simple that this does to register new key to required bucket. In addition, in the case of second, the corresponding bucket to the insertion key is searched justly, and the size of the bucket has the maximum size of bucket. In this case, because space that can do to register new key is lacking, overflow happens in the bucket. When there is an overflow in the required bucket, the following processes are repeated until the overflow of the bucket does not happen. First, the full bucket is changed into a tree (this tree is called a unit tree), which consists of a node and two dummy buckets. Next, all the keys in the full bucket and an insertion key are distributed between the two dummy buckets of the unit tree.

Let us examine the process that inserts a new key in Fig. 5's class CB tree. Fig. 5 composes hierarchical CB tree to key set {air, tea, zoo}. Moreover, depth of tree detached is two, and size of bucket has been fixed by "1". For example, let us add new key "bag" in this Fig. 5's class CB tree. This insertion process engenders overflow. A key "air" includes already to bucket that corresponds to a new key "bag" and size of bucket is 1. New node is added in external node to solve this overflow, because new key "bag" and a key "air" have same bit string to 4th ($= 0000$), this breeds three dummy nodes. CB tree is added, two-separated tree by these dummy nodes. This two-separated tree is linked from external node to pointer. Fig. 6 displays great kindness hierarchic binary tree and HCB tree that insert new key.

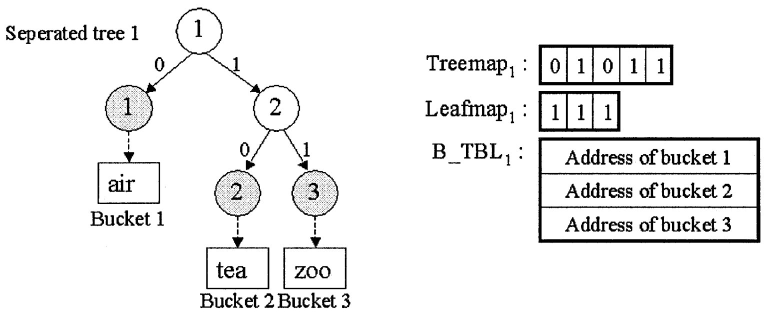

Fig. 5. Hierarchical CB tree.

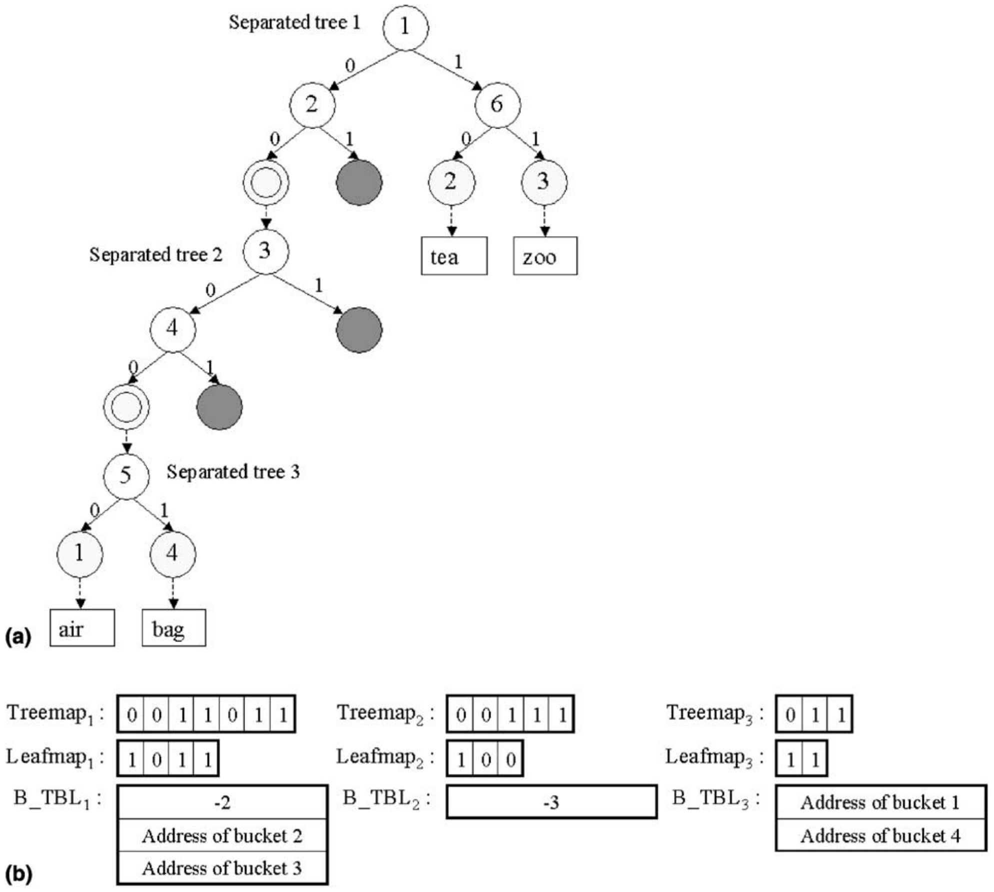

Fig. 6. After adding new key "bag" in hierarchical CB tree.

4.3.2. *A problem of insertion algorithms for the HCPat tree*

In the case of insertion algorithm of HCPat tree, a process is little different with hierarchic CB tree, because it happens to be a case that new node is added at external node and interior node. In case there is composure to bucket and case overflow happens, a process is similar hierarchic CB tree. This addition is achieved at all external nodes. But, HCPat tree needs additional operation because addition is achieved at internal node in false drop's case when appropriate bucket does not exist. The following problems occurred when a new key is inserted at the HCPat tree.

1. The case that the bucket is partially full.
2. The case that the bucket is full.
3. The case that the key cannot belong to the bucket, that is, the key happens to be the false drop.

In first case, use algorithm such as hierarchic CB tree. Find appropriate bucket by search key, because size of the bucket is smaller than size of maximum bucket, it is simple that do to register new key. In second case, it is a little different from hierarchic CB tree. HCPat tree is no dummy node. Change a node that overflow happens to internal node, after add new external node in internal node that changes, and registers new key. And in the case of last, this is when false drop happens by new key. We must add new node in internal node so that false drop happens and modifies eliminated node's number, because a number of dummy nodes has been stored only to a eliminated node. But, if internal node modified this, a problem should modify both the low child nodes happens.

Fig. 7 is example of HCPat tree composed by Key Set {air,art, bag, bus}. A separated depth of tree is two, and size of bucket fixed by one. Let us examine the process that adds new key in this HCPat tree. Priority, we can find bucket that corresponds to new key "eat" (= 00100 00000 10011). However, this current bucket are already different has a registered key "air". We can know that false drop happens if we compare bit string of this key "air" and bit string of new key "eat". In the present case, we must add new node in internal node so that false drop (= internal node 2) happens. Then, this causes problem that must update all low child nodes of internal node (= internal node 3, 4, external node 1, 2, 3, 4). We propose insertion algorithm that improves to avoid these time expense problem that happens.

4.3.3. Improved insertion algorithm of the HCPat tree

If except case that false drop happens, insertion algorithm of the HCPat tree is similar to the hierarchical CB tree. Internal node that false drop happens requires modifying of all child nodes, and this request is bad because spends much time expense. We propose method that avoids

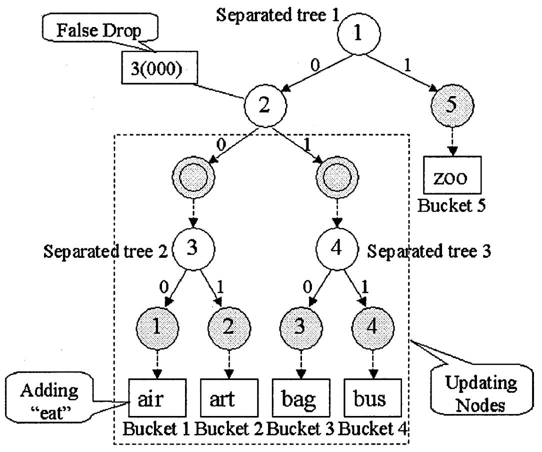

Fig. 7. Insertion problem in the HCPat tree.

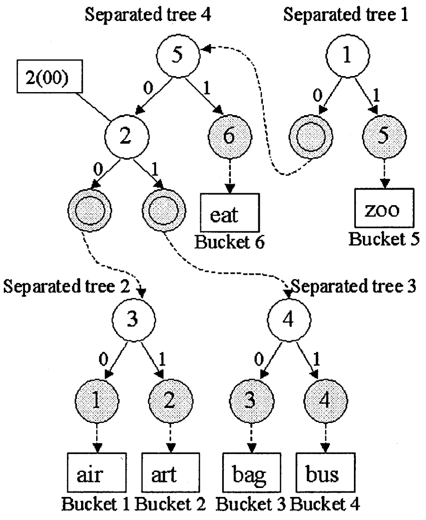

Fig. 8. The HPat tree of after inserting new key "eat" in Fig. 7's tree.

modifying of low rank node. Fig. 8 is example of method that does not modify low child nodes. Like Fig. 8's example, we moved node that false drop happens in new tree, and attached original node and new tree to pointer. This requires only two modifications of tree detached. Next operation is insertion algorithm process of that false drop happened.

- *Step* 1: Store the position of the false drop node
  Store the position of *keypos*, *treepos*, *nodepos*, num in the false drop node.
- *Step* 2: Product the new separated trie
  Product the new $Treemap_{num}$, $Nodemap_{num}$, and B_TBL_{num} that have the lowest num to empty.
- *Step* 3: Transpose the $Treemap_{num}$ to the new separated trie
  Obtain the bits from false drop node until below child node in $Treemap_{num}$, and these bits are transposed to the *Treemap* of the new separated trie, and replace moved bits of $Treemap_{num}$ by '1'.
- *Step* 4: Transpose the $Nodemap_{num}$ to the new separated trie
  Obtain the bits from false drop node until below child node in $Nodemap_{num}$, and these bits are transposed to the *Nodemap* of the new separated trie.
- *Step* 5: Verification of the bit value of the key
  If the bit of the key pointed to by *keypos* is '0', proceed to step 6, otherwise to step 7.
- *Step* 6: Appending the external node to left subtree
  Replace the first bit '0' of *Treemap* in the new separated trie by '010', and proceed to step 8.
- *Step* 7: Appending the external node to right sub-tree
  Replace the first bit '0' of *Treemap* in the new separated trie by '001', and proceed to step 8.
- *Step* 8: Modification of the *Nodemap* of new separated trie
  Replace the bits of false drop position in *Nodemap* of the new separated trie by '0'.
- *Step* 9: Transpose the B_TBL_{num} to the new separated trie
  Transpose the buckets to the new separated trie, these buckets are included below child node of the false drop node.

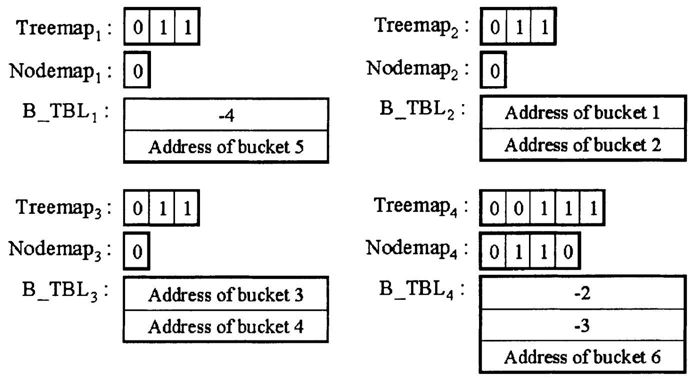
Treemap$_1$: 0 1 1
Nodemap$_1$: 0
B_TBL$_1$: -4 / Address of bucket 5
Treemap$_2$: 0 1 1
Nodemap$_2$: 0
B_TBL$_2$: Address of bucket 1 / Address of bucket 2
Treemap$_3$: 0 1 1
Nodemap$_3$: 0
B_TBL$_3$: Address of bucket 3 / Address of bucket 4
Treemap$_4$: 0 0 1 1 1
Nodemap$_4$: 0 1 1 0
B_TBL$_4$: -2 / -3 / Address of bucket 6

Fig. 9. The HCPat tree of after inserting new key "eat" in Fig. 7's tree.

- *Step* 10: Linkage to the new separated trie
  Append the pointer of the new separated trie to B_TBL$_{num}$ by minus.
- *Step* 11: Appending the new key to the new separated trie
  Append a new key to the B_TBL$_{num}$ of the new separated trie.

Fig. 9 is HCPat tree after insert new key "eat" in hierarchic Pat tree of Fig. 7. Next progress is example that solves false drop when inserting new key "eat" in HCPat tree.

[an insertion process of key 'eat']

1. *Step* 1: *keypos* $=2$, *treepos* $=1$, *nodepos* $=2$, num $=1$.
2. *Step* 2: Product the *Treemap*$_4$,*Nodemap*$_4$, B_TBL$_4$.
3. *Step* 3: Transpose the bits '011' to *Treemap*$_4$ of the new separated tree 4 from the bits '00111'of *Treemap*$_1$.
4. *Step* 4: Transpose the bits '1110' to *Nodemap*$_4$ of the new separated tree from the bits '01110'of *Nodemap*$_1$.
5. *Step* 5: The bit value 'eat' is '1' at *keypos* $(=2)$, go to step 7.
6. *Step* 7: Replace the bits '011' of *Treemap*$_4$ by '00111' in the new separated trie 4.
7. *Step* 8: Replace the bits '1110' of *Nodemap*$_4$ by '0110' in the new separated trie 4.
8. *Step* 9: Transpose the B_TBL$_1$ [1], [2] to B_TBL$_4$ of the new separated trie 4.
9. *Step* 10: Append the bucket address '-4' to B_TBL$_1$ [1] of the separated trie 1.
10. *Step* 11: Append the new key 'eat' to B_TBL$_4$ [3] of the separated trie 4.

5. Evaluations

This algorithm uses memory capacity to be a few, and has fast process time about big key set that you set. This algorithm is implemented by C++ and we tested in console mode of Windows 2000 that have an Intel 300 MHz CPU. The used words of evaluation are 70,000 nouns in Japanese with a length of 6–100 byte (1,596,650 byte) and 120,000 words in English with a length of 1–50 byte (1,414,389 byte). Priority, we must find values that the best result appears of depth of tree and size of bucket. Much differences show on registration time and retrieval time by this value. The following may compare traditional method and this method that proposes using cost that find.

5.1. Evaluations on the tree depth

Fig. 10 shows the transformation of registration time and retrieval time for total keys in key set, and Fig. 11 shows the transformation of *Treemap*'s size and *Nodemap*'s size by increase of the tree depth. Difference can show little according to order of keyword that experiment result registers. As shown Figs. 10 and 11, if depth of tree deepens, retrieval time increases gradually, and the total number of separated trees is decreased. However, size of tree detached increases gradually. As the results, transfer time between trees detached is less more long than transfer time of bit in one tree detached. And, if depth of tree deepens, the capacity of *Treemap* and *Nodemap* decreases some. Based on Figs. 10 and 11, we could get result that best efficiency is good when a tree of depth is two.

5.2. Evaluations on the bucket size

Fig. 12 shows change of registration time and retrieval time by bucket size, and Fig. 13 shows change in *Treemap*'s size and *Nodemap*'s size by augmentation of bucket size. As shown in Fig. 12

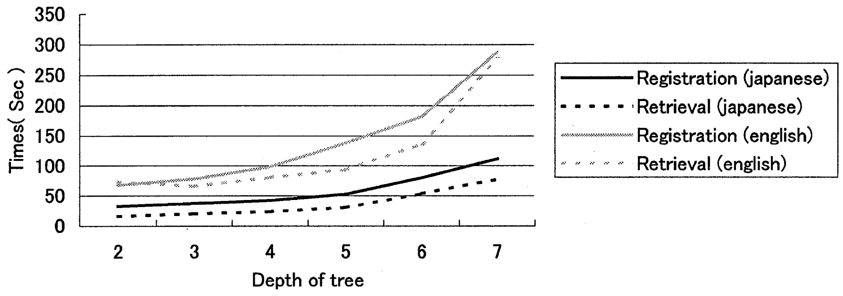

Fig. 10. Transformation of registration time and retrieval time by the depth of tree.

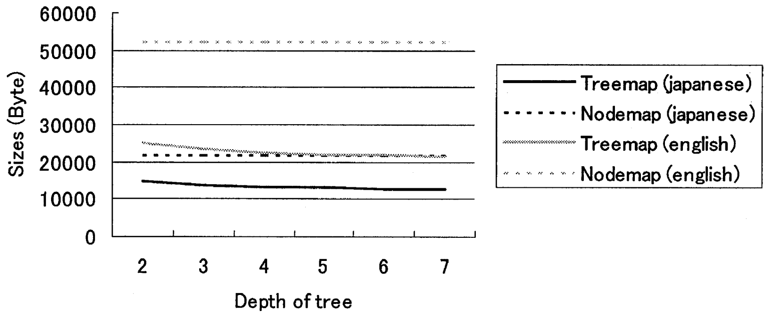

Fig. 11. Transformation of the *Treemap*'s size and *Nodemap*'s size by the depth of tree.

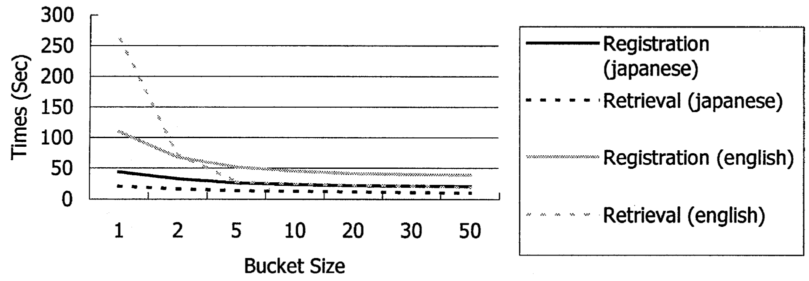

Fig. 12. Transformation of registration time and retrieval time by the bucket size.

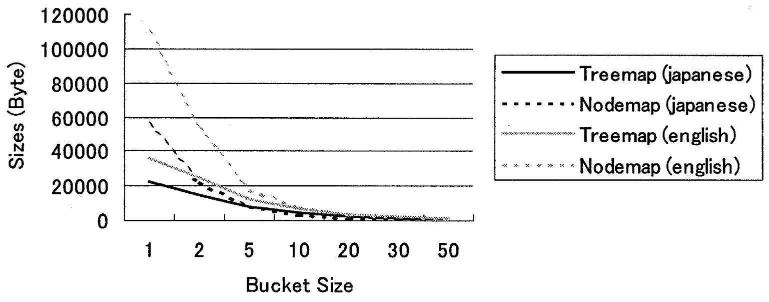

Fig. 13. Transformation of the *Treemap*'s size and *Nodemap*'s size by the bucket size.

and 13, if size of bucket increases, registration time and retrieval time decrease. In addition, *Treemap*'s whole size and *Nodemap*'s complete size decrease too. Because the size of bucket increases, the number of separated trees decreases. However, if the bucket size is too large, it will summon the waste of disk, and it takes a lot of time when you retrieve the keyword in a bucket. Based on Figs. 12 and 13, we could get a result that best efficiency is good when the bucket size is about 10.

5.3. Compare of Pat tree, CPat tree and HCPat tree

Usually, a Pat tree is with very fast process time, and CPat tree is less memory expense by large key set. Because all nodes have been linked by pointer, an operation of Pat tree is fast available, and CPat tree uses memory which is less because all nodes have been consisting of bit array. To get strong points of Pat tree and CPat tree, we composed tree by dividing in several trees that have fixed length. One tree detached consisted bit string, and these separated trees are linked to pointer each other. Because HCPat tree has such a structure, we can get fast operation result into small memory capacity. In evaluations, the depth of HCPat tree is three, and the size of bucket is 10. Such tree depth and bucket size are values that get estimation result of Fig. 5 to basis. The HCPat tree, Pat tree and CPat tree use in estimation has all same bucket sizes and same tree depth.

As see in Tables 2 and 3, HCPat tree shows fast access time, and this uses memory that is less than traditional method. In the case of Japanese word, a registration time of CPat tree is

Table 2
Experiment result of the Pat tree, CPat tree and HCPat tree that use Japanese word

	Pat tree	CPat tree	HCPat tree
Information of tree			
Number of nodes	25,857	25,857	30,696
Internal nodes	12,928	12,928	12,928
External nodes	12,929	12,929	17,768
Times			
Registration (s)	10.3	928.0	26.7
Retrieval (ms)	0.084	12.711	0.219
Insertion (ms)	0.147	13.257	0.382
Storage			
Treemap (Kbytes)	103.4	3.2	3.8
Nodemap (Kbytes)	51.7	3.4	3.4
Bucket table (Kbytes)	0.0	51.7	71.0

Table 3
Experiment result of the Pat tree, CPat tree and HCPat tree that use English word

	Pat tree	CPat tree	HCPat tree
Information of tree			
Number of nodes	44,517	44,517	52,976
Internal nodes	22,258	22,258	22,258
External nodes	22,259	22,259	30,718
Times			
Registration (s)	23.5	3981.4	52.6
Retrieval (ms)	0.109	30.294	0.245
Insertion (ms)	0.195	33.178	0.438
Storage			
Treemap (Kbytes)	178.1	5.6	6.6
Nodemap (Kbytes)	89.0	7.9	7.9
Bucket table (Kbytes)	0.0	89.0	122.8

928 s, but one of HCPat tree is 26 s only. Moreover, a process time of retrieval is about 40 times faster than CPat tree, and this difference is great the more there are many keys registered. Only, about 35% of memory capacity is big if compared with CPat tree consisted of pure bit string. This memory vacancy is the pointer space for separated tree. HCPat tree needs such additional space to connect tree detached. Of course, it is Pat tree that has the fastest processing speed, but this requires memory of much capacity for very large key set. As shown in Table 2 or Table 3, HCPat tree requires 1/3 memory capacity of Pat tree's and this difference becomes bigger by the increase of key set. Consequently, HCPat tree has the 40 times fast processing speed than traditional CPat tree, and it requires just 1/3 memory space than traditional Pat tree.

6. Conclusion

A CPat tree is the compact data structure, and it is able to retrieve keys in order, but this has problem of high time expense about key set of large. This treatise proposes a HCPat tree that uses less memory about big key set, and has fast access time. This tree has been separated to fixed length, and the detached has coupled structure to pointer. And we improved registration algorithm so that it can be composed dynamically. For evaluations, we compare and evaluated HCPat tree with traditional Pat tree and CPat tree. In process time efficiency, HCPat tree is faster about 40 times than CPat tree. In memory efficiency aspect, it requires 1/3 memory space of Pat tree that is consisted of pointer structure. As for the future works, we will propose *Treemap*'s compression method that increases more memory efficiency.

References

Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1983). *Data structures and algorithms*. Reading, MA: Addison-Wesley.

Aoe, J. (1989). An efficient digital search algorithm by using a double-array structure. *IEEE Transactions on Software Engineering*, *15*(9), 1066–1077.

Aoe, J. (1991). *Computer algorithms-key search strategies*. Silver Spring, MD: IEEE Computer Society Press.

Aoe, J., Morimoto, K., Shishibori, M., & Park, K. (1996). A trie compaction algorithm for a large set of keys. *IEEE Transactions on Knowledge and Data Engineering*, *8*(3), 476–491.

Blumer, A., Blumer, J., Haussler, D., & Mcconnel, R. (1987). Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, *34*(3), 578–595.

Gonnet, G. H. (1984). Searching algorithms. In *Handbook of algorithms and data structures* (pp. 25–147). Reading, MA: Addison-Wesley.

Gonnet, G. H., Baeza-Yates, R. A., & Snider, T. (1992). New indices for text: Pat trees and Pat arrays. In *Information retrieval-data structures and algorithms* (pp. 66–82). NJ: Prentice-Hall.

Jonge, W. D., Tanenbaum, A. S., & Reit, R. P. (1987). Two access methods using compact binary trees. *IEEE Transactions on Software Engineering*, *13*(7), 799–809.

Litwin, W. A., Roussopolulos, N., Levy, G., & Hong, W. (1991). Trie hashing with controlled load. *IEEE Transactions on Software Engineering*, *17*(7), 678–691.

Morrison, D. R. (1968). PATRICIA: practical algorithm to retrieve information coded in alpha-numeric. *Journal of ACM*, *14*(4), 514–534.

Sato, T., Sugihara, K., & Yoshitake, S. (1999). Coding grams of variable length into fixed bytes for fast full text retrieval with compact indices. *IR-L Digest*, *16*(37), 473.

Manber, U., & Baeza-Yates, R. (1991). An algorithm for string matching with a sequence of don't cares. *Information Processing Letters*, *37*, 133–136.

Manber, U., & Myers, G., (1990). Suffix arrays: a new method for on-line string searches. In *1st ACM-SIAM symposium on discrete algorithms* (pp. 319–327).