An Efficient Digital Search Algorithm by Using a Double-Array Structure

JUN-ICHI AOE, MEMBER, IEEE

Abstract—**An efficient digital search algorithm is presented in this paper by introducing a new internal array structure, called a *double-array*, that combines the fast access of a matrix form with the compactness of a list form. Each arc of a digital search tree, called a *DS-tree*, can be computed from the double-array in $O(1)$ time; that is to say, the worst-case time complexity for retrieving a key becomes $O(k)$ for the length k of that key. The double-array is modified to make the size compact while maintaining fast access and algorithms for retrieval, insertion and deletion are presented. Suppose that the size of the double-array is $n + cm$, n being the number of nodes of the DS-tree, m being the number of input symbols, and c a constant depending on each double-array. Then it is theoretically proved that the worst-case times of deletion and insertion are proportional to cm and cm^2, respectively, independent of n. From the experimental results of building the double-array incrementally for various sets of keys, it is shown that the constant c has an extremely small value, ranging from 0.17 to 1.13.**

Index Terms—**Database systems, data structure, dictionary, digital search, dynamic internal storage, key retrieval algorithm.**

I. Introduction

In many information retrieval applications, it is necessary to be able to adopt a fast digital search, or trie search [2], [21], [23], [25], [30], for looking at the input character by character. Examples include a lexical analyzer [3], [24] and a local code optimizer [3] of a compiler, a bibliographic search [1], a spell checker [26], a filter for the most frequently used words [27], and a morphological analyzer [12] in natural language processing, and so on. It is important that a dictionary of natural language processing be able to be built incrementally, because one needs to append additional words to the established basic vocabulary from time to time. The algorithm presented here is suitable for information retrieval systems in which the frequency of appending keys is higher than that of deleting keys, allowing the redundant space created by the deletion to be exhausted by the subsequent insertion.

Key retrieval strategies can be roughly classified into two categories, depending on whether a set of keys is variant or invariant. The first category, called the "*dynamic method*" permits the retrieval table to be modified. Several dynamic methods are known, including hashing [2], [23], binary trees [23], [30], B^+-trees [2], [30], extensible hashing [20], and trie hashing [19]. The second, the "*static method*," does not allow this modification. Perfect hashing [14]–[16], [18], [27], sparse table representation [5], [9], [22], [29], and a compressed trie [25] are included in this category. When using the static method, we can concentrate our efforts on the retrieval speed and compactness of the data structure, but when using the dynamic method, extra space must be taken into consideration in order to update the items quickly. The retrieval method presented here can be categorized as falling between the static and dynamic methods, so it is called the "*weak static method*." Extending the static method to the weak static method while maintaining the useful points of the former is ideal but difficult task. The extension of perfect hashing has been presented by Cormack *et al.* [16], but they could not determine the upper bound of the insertion time. The aim of this paper is to present a digital search algorithm incorporating the speed and compactness of the static method with the updating ability of the dynamic method.

Instead of basing a search method on comparisons between keys, a digital search [19], [23] makes use of their representation as a sequence of digits or alphabetic characters. Each node of the DS-tree on level h represents the set of all keys that begin with a certain sequence of h characters; the node specifies branches, depending on the $(h + 1)$st character. The basic concept of this paper is to compress a trie structure [2], [21], [23], [30] representing the DS-tree into two one-dimensional arrays BASE and CHECK, called a *double-array*, and to present the updating algorithm. Each node of the trie is an array indexed by the next "item." The element indexed is a final-state flag, plus a pointer to a new node (or a null pointer). Retrieval, deletion and insertion on the trie are very fast, but it takes lots of space because most of the nodes in the trie are empty; that is to say, the trie is sparse. So we shall try to compress node r into CHECK by giving a mapping from locations in r to locations in CHECK such that no two nonempty locations in each node are mapped to the same position in CHECK. Our mapping is defined by BASE[r] for each node r.

In the following sections, we will describe our ideas in detail. In Section II, we formalize the DS-tree as a string pattern matching machine and define the double-array

Manuscript received December 16, 1987; revised February 7, 1989. Recommended by T. Murata. This work was supported in part by a subsidy from the Japanese Ministry of Education for promoting scientific research.

The author is with the Department of Information Science and Systems Engineering, Faculty of Engineering, University of Tokusima, Minami-Josanjima-Cho, Tokusima-Shi 770, Japan.

IEEE Log Number 8929446.

structure computing an arc in $O(1)$ time. In order to apply the double-array to a large set of keys, the double-array is modified while maintaining the fast access, and a retrieval algorithm for the modified double-array is provided. The principal innovation is to store into the double-array only as much of the prefix in the DS-tree as is necessary to disambiguate the key, and to store the tail of the key not needed for further disambiguation in a compact form in a string array. Algorithms for insertion and deletion are discussed and illustrated through an example in Section III. When an insertion of a new nonempty location of node r is blocked by a nonempty location of another node k in CHECK, an insertion algorithm solves the conflict by redefining BASE[r] or BASE[k]. In this selection of an r or k node, priority is given to the one with fewest nonempty locations, to reduce space and time. In Section IV each worst-case time complexity of the presented algorithms is theoretically determined and supported by results of experimentation with various sets of keys. Partial matching and a key-order search [19] by the double-array are also discussed. Finally, in Section V we summarize our results.

II. Retrieval Algorithm Using the Double-Array

A. Formalization of a Digital Search Tree

In the DS-tree, each path from the root to a leaf corresponds to one key in the presented set. This way, the nodes of the DS-tree correspond to the prefixes of keys in the set. To avoid confusion between words like THE and THEN, let us add a special *endmarker* symbol, #, to the ends of all keys, so no prefix of a key can be a key itself. In this paper, a set of these keys is denoted as K. There is a remarkable relationship between the DS-tree and the string pattern matching machine [1], [7] [8], [10], which locates all occurrences of a finite number of keys in a text string. Hence, the DS-tree will be defined in terms of the matching machine in this paper.

The DS-tree for K is formally defined by a 5-tuple (S, I, g, s_1, A), where

1) S is a finite set of *nodes*.

2) I is a finite set of *input symbols*.

2) g is a function from $S \times I$ to $S \cup \{\text{fail}\}$ called a *goto function*.

3) s_1 is the *initial node, or the root* in S.

4) A is a finite set of *accepting nodes*.

In other words, s_r is in A if and only if a path from s_1 to s_r spells out some $x\#$ in K. The arc labeled a in I from s_r to s_t indicates that $g(s_r, a) = s_t$. The absence of an arc indicates *fail*. The following (usual) conventions hold in this paper:

$$a, b, c, d, e \in I \cup \{\#\}; x, y, z \in (I \cup \{\#\})^*.$$

Let ϵ be the empty string. The notation of the goto function for K is extended to strings by the conditions

$$g(s_r, \epsilon) = s_r, \; g(s_r, ax) = g(g(s_r, a), x).$$

B. A Double-Array Structure and Its Modification

A triple-array structure using one-dimensional arrays BASE, CHECK, and NEXT was defined by S. C. Johnson [17] (the details are found in Aho *et al.* [3]) as a static implementation scheme for transition tables of YACC [3], [17] and LEX [3], [24]. Aoe *et al.* [7], [10] improved this structure by restricting the applicable transition table to that of a static string pattern matching machine. The improved structure, or a double-array, uses only two arrays BASE and CHECK, indexed by node numbers, and maintains the configuration represented in Fig. 1 below.

For s_r and s_t in S, $g(s_r, a) = s_t$ if and only if the double-array for K holds

$$t = \text{BASE}[r] + a \text{ and } \text{CHECK}[t] = r.$$

Note that each node s_r in S corresponds to an *index* r, or *node number*, of the double-array and that each input symbol a is treated as a numerical value. In the double-array structure, the subsequent governing node number t can be computed by BASE[r] and the current input symbol a, so the array NEXT can be removed from the triple-array. One of the valuable features in this paper is to make the triple-array structure reduce to two arrays, but it is a more significant originality to apply the double-array to a compact and fast key retrieval table; and to extend the static use with the double-array inherited from the triple-array to the dynamic use.

The DS-tree has many nodes for a large set of keys, so it is important to make the double-array more compact. In the interest of space saving, we define the following terms on the DS-tree.

Definition 1: The following nodes are defined on the DS-tree.

1) For key xay in K, we define node s_r such that $g(s_1, xa) = s_r$ as a *separate node* if a is a sufficient symbol for distinguishing the key xay from all others in K.

2) Each node in a path from the initial node to the separate node is called a *multinode*.

3) Each node in a path from the separate node to the accepting node is called a *single-node*.

Let S_P, S_M, and S_I be sets of separate nodes, multinodes and single-nodes, respectively. Note that $S_P = S_M \cap S_I$.

Definition 2: A string x such that $g(s_r, x) = s_t$ for s_r in S_P and for s_t in A is called a *single-string* for the separate node s_r, denoted by STR[s_r].

We propose that the arcs from $S_M \times (I \cup (\#))$ to S_M be stored in the double-array and that those from $S_I \times (I \cup (\#))$ to S_I be stored as the single-string in a string memory, called a *TAIL*. There is, however, a problem in how to determine the following:

1) Whether a given node is a separate node or not.

2) A location for taking a single-string from TAIL.

This problem can be easily solved by using additional arrays, but it can take up too much space. The following modified double-array and TAIL enable us to solve the problem without using extra space.

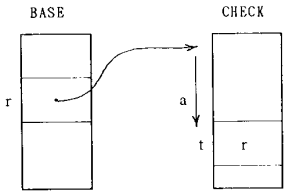

Fig. 1. A double-array structure for $g(s_r, a) = s_t$.

Definition 3: We define the double-array and TAIL as being valid for K if the following conditions are satisfied for the DS-tree.

1) For s_r, s_t in S_M, $g(s_r, a) = s_t$ if and only if BASE[r] + a = t and CHECK[t] = r.

2) For s_r in S_P, suppose that STR[s_r] = $b_1 b_2 \cdots b_m$ ($0 \leqq m$). Then, BASE[r] < 0 and for $p = -$BASE[r],

$$\text{TAIL}[p] = b_1, \text{TAIL}[p+1] = b_2, \cdots, \text{TAIL}[p+m-1] = b_m.$$

In this elaborate structure as depicted in Fig. 2, BASE[r] has two values to indicate a separate node number and locate the single-string in TAIL. Generally, the double-array can be used as an internal retrieval table, and TAIL may include the essential single-string and a record associated with the retrieval key, or alternatively a pointer locating the record in a file. Thus, in TAIL we use the symbol "$" as a one-byte information item concerning the record and the symbol "?" as a garbage symbol.

Example 1: Consider a set K' = {*baby#*, *bachelor#*, *back#*, *badge#*, *badger#*, *badness#*, *bcs#*), where bcs means BCS(Bachelor of Computer Science). Figs. 3 and 4 show the DS-tree, and the double-array and TAIL for K', respectively. In Fig. 3 the multinodes are $s_1 \sim s_{20}$ and the single-nodes s_4, s_9, s_{10}, s_{13}, s_{15} and $s_{19} \sim s_{35}$. The separate nodes and single-strings are as follows:

$$\text{STR}[s_4] = y\#, \quad \text{STR}[s_{10}] = \text{elor}\#, \quad \text{STR}[s_{13}] = \#,$$

$$\text{STR}[s_{20}] = \epsilon, \quad \text{STR}[s_{19}] = \#, \quad \text{STR}[s_{15}] = \text{ess}\#,$$

$$\text{STR}[s_9] = s\#.$$

In this example, the numerical values for $a, b, c, \cdots, r$ are regarded as 1, 2, 3, $\cdots$ 18, respectively. # is treated as 19. For the double-array and TAIL shown in Fig. 4, arc $g(s_7, b) = s_4$, and separate node s_4, satisfy Definition 3 as follows.

$$\text{BASE}[7] + b = 2 + 2 = 4 = t \text{ and CHECK}[4] = 7,$$

$$\text{BASE}[4] = -17 < 0, \text{ and VALUE}[17] = y$$

$$\text{and VALUE}[18] = \#.$$

C. Retrieval Algorithm

In order to keep a node number from exceeding the maximum index of the double-array, we define the size, denoted by *DA-SIZE*, of the double-array as the maximum index of the nonzero entries of CHECK, and DA-SIZE is stored in CHECK[1] for the dynamic double-array. Algorithm 1 of retrieving a key is summarized below, where it utilizes the following functions.

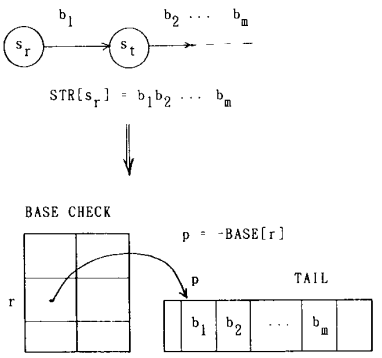

Fig. 2. The double-array and TAIL.

FETCH_STR(p): Return a string TAIL[p]TAIL[p + 1] $\cdots$ TAIL[p + k] such that TAIL[p + k] = #($0 \leqq k$).

STR_CMP(x, y): Return -1 if a string x is equal to string y, otherwise return the length of the longest prefix of x and y.

Algorithm 1. Retrieval Algorithm.

Input: A string $x = a_1 a_2 \cdots a_n a_{n+1}$, a_{n+1} = #; and BASE, CHECK and TAIL for K.

Output: If $x \in K$, then the output is TRUE, otherwise FALSE.

Method:

```
begin
    Initialize the current node number r and the
    input position h to 1 and 0, respectively;
    repeat
      h := h + 1;
      Set the next node number t to BASE[r]+a_h;
(1-1) if t exceeds DA-SIZE or CHECK[t] is
        unequal to r then return(FALSE)
        /* g(s_r, a_h) = s_t is undefined*/
      else Set r to t
    until BASE[r] < 0; /* s_t is a separate node*/
(1-2) if h reaches the last position n+1
    then return(TRUE)
    else Set S_TEMP to the single-string
         FETCH_STR(-BASE[r]);
(1-3) if STR_CMP(a_{h+1} ··· a_n a_{n+1}, S_TEMP) = -1
    then return(TRUE)
         /* a_{h+1} ··· a_n a_{n+1} is the remaining input
           string*/
    else return(FALSE)
end
```

In Algorithm 1, line(1-1) returns FALSE when a mismatch is detected on the double-array and line (1-2) returns FALSE for a mismatch on TAIL. The repeat-loop terminates if BASE[r] is negative, that is, s_r is a separate node. The remaining input string is compared with S_TEMP corresponding to the single-string STR[s_r] to determine whether x is in K or not. Only if a_h = # at line

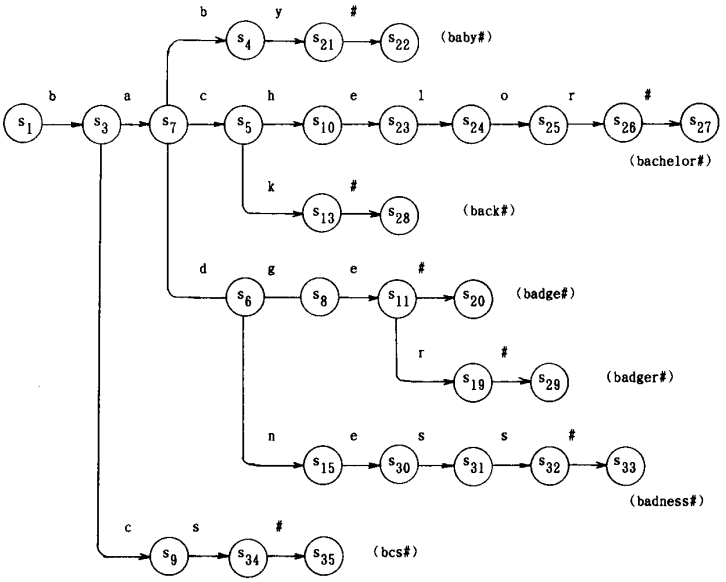

Fig. 3. The DS-tree for K'.

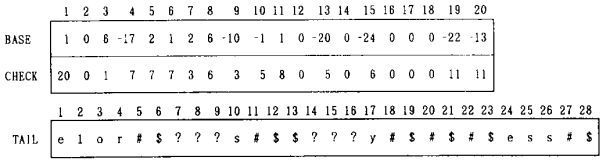

Fig. 4. A double-array for K'.

(1-2), then TRUE is immediately returned because of STR[s_r] = ϵ. Suppose that the double-array and TAIL are valid for K. From the observation and Definition 3, it is clear that Algorithm 1 returns TRUE if the input string x is in K, otherwise FALSE.

Example 2: Consider the retrieval of key *badness#* $\in K'$ by using the double-array and TAIL shown in Fig. 4. Algorithm 1 returns TRUE by the following computations

(1-1): t = BASE[r]+a_1 = BASE[1]+b = 3
t = 3 < DA-SIZE = 20 and CHECK[t] = CHECK[3] = 1
BASE[r] = BASE[3] = 6 > 0
(1-1): t = BASE[r]+a_2 = BASE[3]+a = 7
t = 7 < DA-SIZE = 20 and CHECK[t] = CHECK[7] = 3
BASE[r] = BASE[7] = 2 > 0
(1-1): t = BASE[r]+a_3 = BASE[7]+d = 6
t = 6 < DA-SIZE = 20 and CHECK[t] = CHECK[6] = 7
BASE[r] = BASE[6] = 1 > 0
(1-1): t = BASE[r]+a_4 = BASE[6]+n = 15
t = 15 < DA-SIZE = 20 and CHECK[t] = CHECK[15] = 6
BASE[r] = BASE[15] = −24 < 0
(1-2): h = 4 ≠ n+1 = 8,
S_TEMP = FETCH_STR(24) = *ess#*.
(1-3): STR_CMP(*ess#*, *ess#*) = −1

III. Algorithm of Updating the Double-Array

A. Insertion Algorithm

In order to understand an insertion algorithm, called Algorithm 2, we will illustrate the insertion process by means of the DS-tree. Suppose that Algorithm 2 uses the same input and output as Algorithm 1. Then, Algorithm 2 can be obtained by modifying return(FALSE) of the lines (1-1) and (1-3) of Algorithm 1 as follows.

a) For return(FALSE) of line (1-1):

begin
A_INSERT(r , $a_h a_{h+1} \cdots a_n a_{n+1}$) ;
return(FALSE)
end

b) For return(FALSE) of line (1-3):

begin
B_INSERT(r , $a_{h+1} \cdots a_{h+k}$, $a_{h+k+1} \cdots a_n a_{n+1}$, $b_1 \cdots b_m$) ;
return(FALSE)
end;

The parameters r and $a_h a_{h+1} \cdots a_n a_{n+1}$ of A_INSERT represent the current node number and the remaining input string, respectively. Thus, A_INSERT appends an arc $g(s_r, a_h) = s_t$ to the double-array and stores the single-string STR[s_t] = $a_{h+1} \cdots a_n a_{n+1}$ in TAIL as shown in Fig. 5. When B_INSERT is invoked there must be different symbols b_1 and a_{h+k+1} for S_TEMP = STR[s_r] = $a_{h+1} a_{h+2} \cdots a_{h+k} b_1 \cdots b_m$ ($0 \leqq k \leqq n-h+1$, $1 \leqq m$) and for the remaining input string $a_{h+1} \cdots a_{h+k} a_{h+k+1} \cdots a_n a_{n+1}$ as shown in Fig. 6(a). The first parameter r of B_INSERT stands for the current node number and the other parameters are:

- $a_{h+1} \cdots a_{h+k}$ is the longest prefix of the remaining input string and S_TEMP.

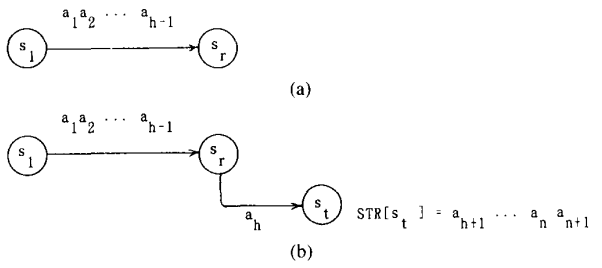

Fig. 5. Illustration of the procedure A_INSERT. (a) The DS-tree before A_INSERT. (b) The DS-tree after A_INSERT.

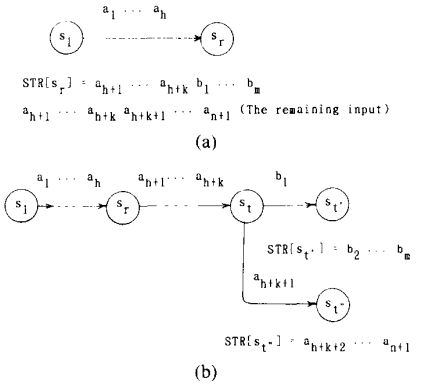

Fig. 6. Illustration of the procedure B_INSERT. (a) The DS-tree before B_INSERT. (b) The DS-tree after B_INSERT.

- $a_{h+k+1} \cdots a_n a_{n+1}$ is the string removed this prefix from that input string.
- $b_1 \cdots b_m$ is the string removed this prefix from S_TEMP.

Thus, B_INSERT appends arcs $g(s_r, a_{h+1} \cdots a_{h+k}) = s_t$, $g(s_t, b_1) = s_{t'}$ and $g(s_t, a_{h+k+1}) = s_{t''}$ to the double-array, and stores STR$(s_{t'}] = b_2 \cdots b_m$ and STR$[s_{t''}] = a_{h+k+2} \cdots a_n a_{n+1}$ into TAIL as shown in Fig. 6(b).

The double-array can be built incrementally using Algorithm 2. We assume that the initial double-array has only two nonzero entries, BASE[1] = 1 and CHECK[1] = 1, and has enough zero entries to insert a new key. Algorithm 2 utilizes the following functions and variables.

Set_LIST(r): Return a set of symbols a such that CHECK[BASE[r] + a] = r.

R-LIST, K-LIST, LIST, ADD, ORG: A subset of $I \cup \{\#\}$.

N(LIST): Return the number of entries in LIST.

POS: A global variable which indicates the minimum index of available entries of TAIL, and which has an initial value of 1.

X_CHECK(LIST): Return the minimum q such that $q > 0$ and CHECK[$q + c$] = 0 for all c in LIST.

STR_TAIL (p, y): Store the string y in the location p of TAIL and return POS incremented by the length of y if p = POS, otherwise return the original POS.

The procedures A_INSERT and B_INSERT are summarized below, but we use the same formal parameters as the actual parameters in order to aid comprehension of these insertions.

procedure A_INSERT(r, $a_h a_{h+1} \cdots a_{n+1}$)
Step (a-1). Set the next node number t to BASE[r]+a_h.
If CHECK[t] = 0, then goto Step (a-3).
/* $g(s_r, a_h) = s_t$ can be defined on the double-array*/
Otherwise change either BASE[r] or BASE[k] for k = CHECK[t] at the next Step (a-2).
/* $g(s_r, a_h) = s_t$ cannot be defined on the double-array because $g(s_k, b) = s_t$ (See Fig. 7)*/
Step (a-2). Set R-LIST and K-LIST to SET_LIST(r) and SET_LIST(k), respectively.
If N(R-LIST)+1 is less than N(K-LIST) then
Call a function MODIFY(r, r, $\{a_h\}$, R-LIST) to determine BASE[r] such that
CHECK[BASE[r]+b] = 0 for all b in R-LIST $\cup$ $\{a_h\}$.
Otherwise call MODIFY(r, k, ϕ, K-LIST) to determine BASE[k] such that
CHECK[BASE[r]+a_h] $\neq$ CHECK[BASE[k]+b] for all b in K-LIST, and set r to the returned node number by this MODIFY.
/* ϕ is the empty set*/
Step (a-3). Call INS_STR(r, $a_h a_{h+1} \cdots a_{n+1}$, POS) to define $g(s_r, a_h)$ on the double-array and to store $a_{h+1} \cdots a_{n+1}$ in TAIL.

Since N(R-LIST) and N(K-LIST) designate the numbers of arcs drawing out nodes s_r and s_k, respectively, changing BASE[k] instead of BASE[r] is reasonable in terms of time efficiency when N(R-LIST) + 1 > N(K-LIST), where the value 1 corresponds to $g(s_r, a_h)$. In MODIFY(r, k, ϕ, K-LIST) to change BASE[k], we must consider a special case such that BASE[k] + d = r for d in K-LIST, because the current node number r is changed along with BASE[k] as shown in Fig. 8. To solve the problem, MODIFY has the first parameter indicating the current node number r and returns the valid current node number.

function MODIFY(current_s, h ADD, ORG)
Step (m-1). Copy BASE[h] to *old_base* and determine a new BASE[h] as X_CHECK(ADD $\cup$ ORG).
Step (m-2). For each c in ORG, repeat Steps (m-3 ~ m-5).
Step (m-3). Set an old node number t to *old_base* + c.

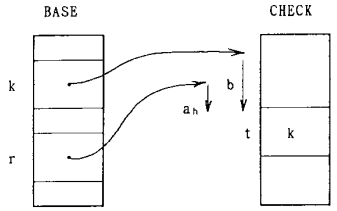

Fig. 7. A double-array for $g(s_k, b) = s_t$ and $g(s_r, a_h) = s_t$.

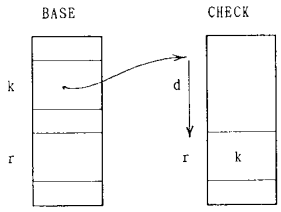

Fig. 8. A double-array such that BASE[k] + d = r.

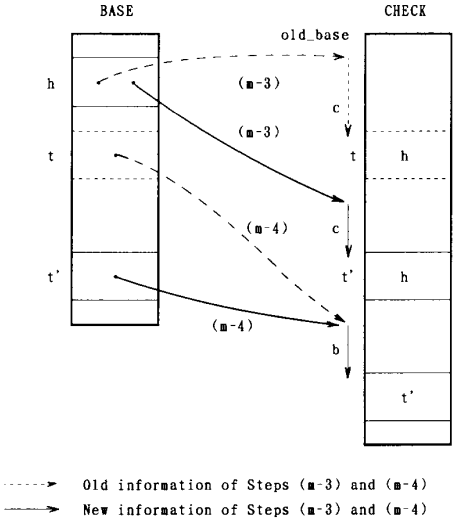

Fig. 9. Illustration of the function MODIFY.

Set a new node number t' to BASE[h] + c.
Copy the original BASE[t] into BASE[t'] and set CHECK[t'] to h.
/* $g(s_h, c) = s_t$ is redefined by $g(s_h, c) = s_{t'}$ as shown in Fig. 9 */

Step (m-4). If BASE[t] > 0, then set CHECK[q] to t' for each q such that CHECK[BASE[t]+b]=t and q = BASE[t]+b.
/* Replace an old node number t in CHECK by a new t' as shown in Fig. 9 */
Set t' to *current_s* if t = *current_s*.
/*The situation described in Fig. 8 happens.*/

Step (m-5). Initialize BASE[t] and CHECK[t] to 0, respectively.

Step (m-6). Return *current_s*.

Note that a new arc $g(s_r, a_h)$ is defined on the double-array by Step (a-3) after MODIFY.

procedure INS_STR(h, $e_1e_2 \cdots e_q$, *d_pos*)

Step (s-1). Set the next node number t to BASE[h]+e_1 ;

Step (s-2). Set BASE[t] and CHECK[t] to $-d\_pos$ and h , respectively.
/* $g(s_h, e_1) = s_t$ is defined on the double-array*/

Step (s-3). POS := STR_TAIL(*d_pos*, $e_2e_3 \cdots e_q\$$).

procedure B_INSERT(r , $a_{h+1} \cdots a_{h+k}$, $a_{h+k+1} \cdots a_na_{n+1}$, $b_1 \cdots b_m$)

Step (b-1). Copy -BASE[r] to *old_pos*

Step (b-2). Define a sequence of arcs for each a_{h+i} for $1 \leqq i \leqq k$ on the double-array by repeating the following statements:
Determine a new BASE[r] as X_CHECK($\{a_{h+i}\}$);
Set CHECK[BASE[r] + a_{h+i}] to r ;
Set the next node number r to BASE[r] + a_{h+i} ;

Step (b-3). Determine a new BASE[r] as X_CHECK($\{a_{h+k+1}, b_1\}$);

Step (b-4). INS_STR(r, $b_2 \cdots b_m$, *old_pos*);
/* $b_2 \cdots b_m$ is overwritten in the original position *old_pos* of TAIL*/

Step (b-5). INS_STR(r, $a_{h+k+2} \cdots a_na_{n+1}$, POS)
/* $a_{h+k+2} \cdots a_na_{n+1}$ is written in the new POS of TAIL*/

Theorem 1: Suppose that the double-array and TAIL are valid for K and that the input x of Algorithm 2 is not in K. Then, the modified double-array and TAIL by Algorithm 2 are valid for $K \cup \{x\}$.

Proof: For $x = a_1a_2 \cdots a_{n+1} \notin K$, consider the configurations shown in Figs. 5 and 6.

1) Procedure A_INSERT: If CHECK[t] = 0 in Step (a-1), then it is clear that Steps (s-1, s-2) of INS_STR invoked at Step (a-3) store an arc $g(s_r, a_h) = s_t$ in the double-array without changing BASE[r]. Moreover, node s_t always becomes a separate node, so BASE[t] determined at Step (s-2) and TAIL stored at Step (s-3) hold 1) and 2) of Definition 3. As for the case of CHECK[t] ≠ 0, we assume that MODIFY(r, r, $\{a_h\}$, R-LIST) was invoked at Step (a-2). It is necessary to prove the following three points.

a) Redefinition of arcs by new BASE[r]: From h = r, ORG = R-LIST, ADD = $\{a_h\}$, the function

X_CHECK of Step (m-1) determines the new BASE[r] such that

$$\text{CHECK}[\text{BASE}[r] + b] = 0$$

$$\text{for all } b \in \text{R-LIST} \cup \{a_h\},$$

so it is clear that $g(s_r, c)$ for all c in R-LIST can be redefined on the double-array by Step (m-3).

b) Influence on the other arcs (see Fig. 9): The arcs defined by the new BASE[r] influence the arcs associated with node s_t such that

$$t = \text{old\_base} + c \text{ and } \text{CHECK}[t] = r \text{ for } c \text{ in R-LIST.}$$

This old node number t is no longer used, so BASE[t] must be copied into BASE[t'] for the new node number $t' = \text{BASE}[r] + c$. As for the old arc $g(s_t, b)$, it is necessary to change CHECK[BASE[t] + b] with the old node number t into the new node number t', but it is unnecessary if s_t is a separate node, that is, BASE[t] < 0.

It is clear that these processes are carried out correctly by Steps (m-3, m-4).

c) Deletion of the old arc: From Step (m-5), the validity is straightforward.

As for MODIFY(r, k, ϕ, K-LIST), there is no confusion since a new BASE[k] is determined as X_CHECK(K-LIST) in Step (m-1) and since the current node number r can be replaced by a new number t' in Step (m-4) whenever $r = \text{BASE}[k] + d$ for d in ORG as shown in Fig. 8. When changing BASE[k], CHECK[BASE[r] + a_h] becomes available, so $g(s_r, a_h) = s_t$ can be stored in the double-array at INS_STR invoked by Step (a-3).

Hence, it is clear that the modified double-array and TAIL by A_INSERT are valid for $K \cup \{x\}$.

2) Procedure B_INSERT: From Steps (b-1 ~ b-3), it is clear that B_INSERT defines $g(s_r, a_{h+1} \cdots a_{h+k}) = s_t$, $g(s_t, b_1) = s_{t'}$ and $g(s_t, a_{h+k+1}) = s_{t''}$ to the double-array. By using Steps (s-2, s-3) in INS_STR invoked at Step (b-4), the STR[$s_{t''}$] $= b_2 \cdots b_m$ is overwritten in the original position *old_pos* (copied by Step (b-1)) of TAIL. By using Steps (s-2, s-3) in INS_STR invoked at Step (b-5), the single-string STR[$s_{t'}$] $= a_{h+k+2} \cdots a_n a_{n+1}$ for the remaining input string is written in the first available position POS of TAIL. In this process, it is clear that the positions *old_pos* and POS are stored by Step (s-2) as BASE[t'] $= -$*old_pos* and BASE[t''] $= -$POS, respectively. Hence, the modified double-array and TAIL by B_INSERT are valid for $K \cup \{x\}$. Therefore, the theorem is proved. Q.E.D.

Example 3: Consider each insertion of keys *bachelor#, bcs#, badge#, baby#, back#, badger#, badness#* for the initialized double-array. For the first four keys, the results of the double-array and TAIL, and the corresponding DS-trees are shown in Figs. 10 and 11, respectively. These final results have been shown in Figs. 4 and 3, respectively.

(a) bachelor#.
From t=BASE[1]+ b=3 and CHECK[3]=0 $\neq$ 1, A_INSERT(1, *bachelor#*) is computed as follows.

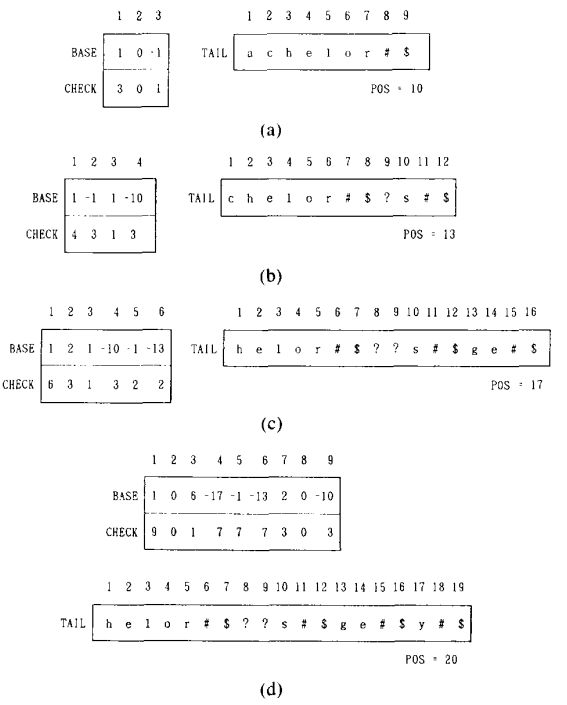

Fig. 10. An example of insertion process on the double-array. (a) The result by A_INSERT (1, *bachelor#*) for *bachelor#*. (b) The result by B_INSERT (3, ϵ, *cs#*, *achelor#*) for *bcs#*. (c) The result by B_INSERT (2, ϵ, *dge#*, *chelor#*) for *badge#*. (d) The result by A_INSERT (2, *by#*) for *baby#*.

(a-1): t = BASE[1] + b = 3, CHECK[3] = 0,
(a-3): INS_STR(1, *bachelor#* , 1),
(s-1,s-2): CHECK[3] = 1, BASE[3] = -1,
(s-3): POS = STR_TAIL(1, *achelor#$*) = 10.

(b) *bcs#*.
The retrieval is unsuccessful because
t = BASE[1] + b = 3, CHECK[3] = 1,
BASE[3] = $-1 < 0$, $h = 1 \neq n + 1 = 3$,
S_TEMP =FETCH_STR(1) = *achelor#*,
STR_CMP(*cs#*, *achelor#*) = 0.
Then, B_INSERT(3, ϵ , *cs#*, *achelor#*) is computed as follows.
(b-1): *old_pos* = $-$BASE[3] = 1,
(b-3): BASE[3] = X_CHECK({*c*,*a*}) =1,
(b-4): INS_STR(3, *achelor#*, 1),
(s-1,s-2): CHECK[2] = 3, BASE[2] = -1,
(s-3): POS = STR_TAIL(1, *chelor#$*) = 10,
(b-5): INS_STR(3, *cs#*, 10),
(s-1,s-2): CHECK[4] = 3, BASE[4] = -10,
(s-3): POS = STR_TAIL(10, *s#$*) = 13

(c) *badge#*
The retrieval is unsuccessful because
t = BASE[1] + b = 3, CHECK[3] = 1,
t = BASE[3] + a = 2, CHECK[2] = 3,
BASE[2] = $-1 < 0$, $h = 2 \neq n+1 = 3$,
S_TEMP = FETCH_STR(1) = *chelor#*,
STR_CMP(*dge#*, *chelor#*) = 0.
Then, B_INSERT(2, ϵ , *dge#*, *chelor#*) = 0 is computed as follows.

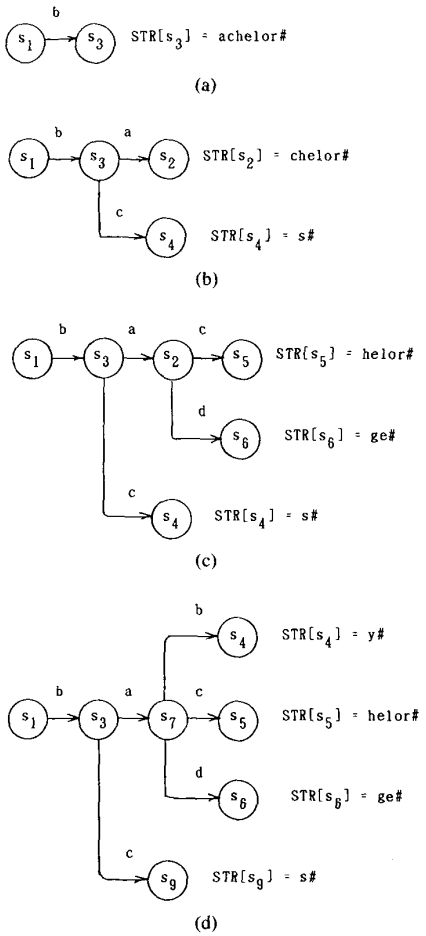

Fig. 11. An example of insertion process on the DS-tree. (a) The DS-tree for *bachelor#*. (b) The DS-tree for *bachelor#* and *bcs#*. (c) The DS-tree for *bachelor#*, *bcs#*, and *badge#*. (d) The DS-tree for *bachelor#*, *bcs#*, *badge#*, and *baby#*.

(b-1): $old\_pos = -\text{BASE}[2] = 1$,
(b-3): $\text{BASE}[2] = \text{X\_CHECK}(\{d, c\}) = 2$,
(b-4): INS_STR(2, *chelor#*, 1)
(s-1,s-2): $\text{CHECK}[5] = 2, \text{BASE}[5] = -1$,
(s-4): $\text{POS} = \text{STR\_TAIL}(1, helor\#\$) = 13$,
(b-5): INS_STR(2, *dge#*, 13),
(s-1,s-2): $\text{CHECK}[6] = 2, \text{BASE}[6] = -13$,
(s-3): $\text{POS} = \text{STR\_TAIL}(13, ge\#\$) = 17$
(d) *baby#*.
The retrieval is unsuccessful because
$t = \text{BASE}[1]+b = 3, \text{CHECK}[3] = 1$,
$t = \text{BASE}[3]+a = 2, \text{CHECK}[2] = 3$,
$t = \text{BASE}[2]+b = 4, \text{CHECK}[4] = 3 \neq 2$.
Then, A_INSERT(2, *by#*) is invoked as follows.
(a-1): $t = \text{BASE}[2] + b = 4, \text{CHECK}[4] = 3 \neq 0$
(a-2): $\text{R-LIST} = \text{SET\_LIST}(2) = \{c, d\}$,
$\text{K-LIST} = \text{SET\_LIST}(3) = \{a, c\}$,
$\text{N(R-LIST)} + 1 = 3 > \text{N(K-LIST)} = 2$,
MODIFY(2, 3, ϕ, $\{a, c\}$) is called.
(m-1): $old\_base = \text{BASE}[3] = 1$
$\text{BASE}[3] = \text{X\_CHECK}(\{a, c\}) = 6$
For a in ORG,
(m-3): $t = \text{old\_base} + a = 2$,
$t' = \text{BASE}[3] + a = 7$,
$\text{CHECK}[7] = 3, \text{BASE}[7] = \text{BASE}[2] = 2$,
(m-4): $\text{BASE}[t] = \text{BASE}[2] = 2 > 0$,
$\text{CHECK}[5] = \text{CHECK}[6] = 7$ because of
$\text{CHECK}[5] = \text{CHECK}[6] = 2$.
$current\_s = 2$ is changed by $t' = 7$ because of $current\_s = t = 2$.
(m-5): $\text{BASE}[2] = 0, \text{CHECK}[2] = 0$.
For c in LIST,
(m-3): $t = \text{old\_base} + c = 4$, $t' = \text{BASE}[3]+c = 9$,
$\text{CHECK}[9] = 3, \text{BASE}[9] = \text{BASE}[4] = -10$,
(m-4): $\text{BASE}[4] = -5 < 0$,
(m-5): $\text{BASE}[4] = 0, \text{CHECK}[4] = 0$.
(m-6): Returns $current\_s = 7$ as the new current node number.
(a-3): INS_STR(7, *by#*, −17)
(s-3): $\text{POS} = \text{STR\_TAIL}(17, y\#\$) = 20$.

B. Deletion

Suppose that a deletion algorithm, called Algorithm 3, uses the same input and output as Algorithm 1. Then, Algorithm 3 can be obtained by modifying two instructions return(TRUE) of Algorithm 1.

Modification of return(TRUE)
begin
BASE[r] = 0; CHECK[r] = 0;
return(TRUE)
end

Algorithm 3 deletes the arc prior to a separate node s_r and consequently the single-string STR[s_r] of TAIL becomes garbage symbol "?".

Theorem 2: Suppose that the double-array and TAIL are valid for K and that the input x of Algorithm 2 is in K. Then, the modified double-array and TAIL by Algorithm 3 are valid for $K - (x)$.

Proof: From CHECK[r] = 0, it is clear that the arc $g(s_t, a_h) = s_r$ entering into a separate node s_r is undefined. A mapping from K to S_P is a bijection (one-to-one correspondence), so the resulting double-array never define $g(s_1, a_1a_2 \cdots a_h)$ for $1 \leqq h \leqq n + 1$. Therefore, the theorem is proved. Q.E.D.

While the entries initialized by the deletion are available for the next insertion, it is difficult to remove them dynamically. Because of this, we can say that the presented scheme is well suitable for a weak static method as mentioned in Section I.

IV. Evaluation

A. Theoretical Observations

The worst-case time complexity of the presented insertion algorithm depends on the procedure MODIFY invoked in A_INSERT. Moreover, the time complexity of MODIFY is related to the size, or DA-SIZE, of the double-array, because the function X_CHECK invoked in

MODIFY travels on the all indexes of the double-array. Since the computation of X_CHECK is similar to that of the row displacements proposed by Tarjan *et al.* [29], it is difficult to evaluate theoretically DA-SIZE by using external parameters n (the number of nodes) and m (the number of input symbols). However, because of the sparse relations on the nodes and input symbols of the goto function, we can assume that DA-SIZE is $n + cm$ for a constant c concerning each double-array, where cm is equal to *the number of available indexes on the double-array*. The value c will be determined through empirical observations.

From the above observation, it is clear that the worst-case time complexity of X_CHECK(LIST) is proportional to the product $nm + cm^2$ of DA-SIZE $n + cm$ and the maximum number m of entries in LIST, but we do not expect to include n in this complexity because n is proportional to the number of keys. Thus, we show that n can be removed from this complexity by modifying the structure of the array CHECK.

Definition 4: Let $r_1, r_2, \cdots r_{cm}$ be the increasing order of the available indexes on the double-array. We define a link, called a G-link, such that

$$\text{CHECK}[r_i] = -r_{(i+1)}; \quad 1 \leqq i \leqq cm-1,$$
$$\text{CHECK}[r_{cm}] = -1,$$

where the value -1 represents the tail of the G-link, but the head of the G-link is represented by G_HEAD. Note that available entries in CHECK must be confirmed by negative integers instead of zeros.

By traveling the G-link in X_CHECK, it is clear that the worst-case time complexity of X_CHECK becomes $O(cm \cdot m) = O(cm^2)$. From two loops concerning "for each c in ORG" in Step (m-2) and "for each q" in Step (m-4), the worst-case time complexity of Steps (m-2 ~ m-5) in MODIFY becomes $O(m^2)$. Hence, the worst-case time complexity of the insertion algorithm becomes $O(cm^2 + m^2) = O(cm^2)$. The double-array realizes random storing of the nodes for the DS-tree, so there is little effect concerning different orders of inserting keys.

Consider the other time complexities. The worst-case time complexity of the deletion algorithm becomes $O(1)$, but the index of the initialized entry in CHECK must be appended to the G-link, so the precise time complexity is $O(cm)$. It is clear that the worst-case time complexity of the retrieval algorithm is $O(k)$ for the length k of a given key.

As for TAIL, the garbage symbol "?" created by the deletion algorithm and the procedure INS_STR invoked in Step (b-4) can be removed by shifting all symbols in TAIL and rewriting the BASE entries for the separate nodes. This time is proportional to $h + (n + cm)$, h being the total length of TAIL. If keys are inserted in alphabetical order, or in numerical value order, then TAIL can be built without creating any garbage symbols because:

1) the single-string x to be rewritten by INS_STR of Step (b-4) must be shorter than the previous single-string y, and

2) the single-string y of 1) always exists in the tail of TAIL.

To do this, the number of created garbage symbols must be subtracted from POS obtained by Step (s-3).

Here, consider a partial matching problem by the double-array. For a key $x * y\#$ with a dummy symbol "*" indicating one unknown symbol, there are at most m arcs $g(s_r, b) = s_t$ for s_r such that $g(s_1, x) = s_r$, and all arcs $g(s_t, y\#) = s_{t'}$ for each s_t must be confirmed on the DS-tree. Since each arc on the DS-tree can be retrieved from the double-array in $O(1)$ time, the worst-case time complexity of this partial matching is proportional to hm for the length h of $y\#$. Thus, for a key with e dummy symbols. the worst-case time complexity of partial matching by the double-array becomes $O(m^e)$.

Essentially, the DS-tree has a key-order preserving property [19] which enables sequential access to keys. In this search on the DS-tree, all arcs $g(s_r, a) = s_t$ for each node s_r existing in range of the requested key-order should be confirmed. Thus, the worst-case time complexity of a sequential search by the double-array becomes $O(dm)$ for the total number d of nodes to be traveled by this search. This time complexity can be greatly reduced by introducing buckets such as a B^+-tree [2], [23], [30]. This modification is shown in Fig. 12.

B. Empirical Observations

A retrieval system, called a *DOUBLE*, based on the presented method, was written using about 2300 lines of C, and implemented on the following workstations: DEC Micro VAX-II, Sun Microsystems Sun-3, SONY NEWS, TOSBAC UX-700, IBM6100, and various personal computers. DOUBLE has the following two routines plus the main routine based on the presented algorithm:

1) design routine,
2) memory management routine.

In routine 1), we can select the sets of the input symbols (i.e., ASCII, EBCDIC, Katakana, or Hiragana in Japanese, Chinese characters, etc.) and design a form of the record plus the essential single-string in TAIL. For example, a bibliographic search [1] and a spell checker [26] require only the single-strings. A local code optimizer [3] of a compiler has a sequence of codes to be replaced. In a machine translation system [12], the basic morphological and syntax data are stored in TAIL and the detailed knowledge translation data is stored in the other file accessible by the pointer in TAIL.

When the maximum node number exceeds a two-byte integer for a large number of keys, the double-array should be divided for appropriate subsets of K if one needs to maintain space efficiency. This division is also useful for a computer with a small main memory, such as a personal computer, because it enables us to read the one appropriate divided double-array from the auxiliary memory. In routine 1), we can set up the number of the divisions and determine whether the double-array and/or TAIL are placed on a main or an auxiliary memory. The routine 2)

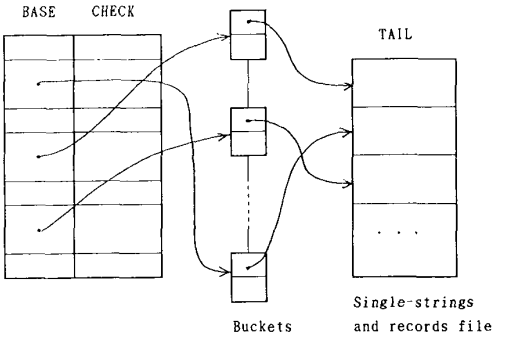

Fig. 12. A double-array with buckets.

manages dynamic storage of the divided double-array and TAIL by means of a first-fit strategy [2], [23], [30].

Table I represents the experimental results of building the double-array and TAIL incrementally for the following sets of keys.

K1, K2: The reserved words for Pascal and Cobol, respectively.

K3: Main city names in the world.

K4: Words for an English dictionary.

K5: Katakana for a Japanese word dictionary.

Note that, in each case, the number m of the input symbols equals 65 and that each double-array for *K*4 and *K*5 is divided into five. In order to see the effect on space and time of different orders of inserting the keys, the double-array was incrementally built in alphabetical order and in three kinds of random orders. Thus, the value specified by "(ordered)" stands for the result of alphabetical order and that specified by "(random)" for the worst-case result of three random orders. Each single-string in TAIL constructed by random order is shifted to reduce space. These times were about nine and six seconds for all keys in *K*4 and *K*5, respectively. It more important to evaluate empirically space and time of the double-array because the single-strings in TAIL should be controlled, together with the records, by a memory management routine in actual information retrieval systems. In Table I, KNUM, MNUM, INUM, and TNUM represent each number of keys, nodes in $S_M - S_P$, nodes in $S_I - S_P$, and all nodes, respectively. That is, TNUM = MNUM + INUM + KNUM because the KNUM is equal to the number of the separate nodes. KLEN is the average length of keys. SBCT stands for storage of both the double-array and TAIL; SBC for storage of the double-array; STAIL for storage of TAIL; and SSOURCE for storage of a source file of all keys with a delimiter between keys. Note that TAIL has no symbol "$" indicating one-byte record information. C-VALUE represents c for the size $n + cm$ of the final double-array, but each C-VALUE for *K*4 and *K*5 is the average value for the divided double-array. As may readily be seen in the results, there is no difference between C-VALUE(ordered) and C-VALUE(random). It depends on the random storing of node numbers in the double-array. From the results, it turns out that the size of the double-array is independent of different orders of inserting the keys.

TABLE I
SIMULATION RESULTS

	K1	K2	K3	K4	K5
Numbers and Length					
KNUM	35	310	1,480	23,976	32,344
MNUM	17	301	981	16,518	13,039
INUM	109	947	7,281	59,741	43,436
TNUM	161	1,558	9,742	100,233	88,801
KLEN	5.1	7.5	9.5	8.2	5.6
Storages(kilo-bytes)					
SBCT	0.36	3.63	17.17	221	226
SBC	0.25	2.69	9.89	161	182
STAIL	0.11	0.95	7.28	60	44
SSOURCE	0.18	2.33	14.08	196	183
C-VALUE(ordered)	0.17	0.94	0.32	0.23	1.13
C-VALUE(random)	0.19	0.96	0.34	0.24	1.14
Counts					
AVE-cm(ordered)	2.4(6)	6.2(48)	4.8(46)	3.6(64)	11.3(93)
AVE-m^2(ordered)	16.9(65)	97.5(325)	117.2(520)	91.3(650)	136.5(975)
AVE-cm^2(ordered)	1.1(3)	4.7(31)	3.6(37)	3.1(44)	19.5(255)
AVE-cm(random)	2.3(6)	5.9(49)	4.9(47)	3.5(63)	12.7(99)
AVE-m^2(random)	16.3(64)	99.3(398)	125.3(531)	96.7(668)	145.9(997)
AVE-cm^2(random)	1.1(3)	4.9(33)	3.7(39)	3.9(48)	20.7(256)
Times(milli-second)					
TSEARCH	0.38(0.40)	0.39(0.41)	0.42(0.43)	0.39(0.45)	0.38(0.43)
TINSERT	31.4(44.4)	30.6(53.7)	29.1(49.1)	36.7(78.6)	35.5(77.1)

From the results in Table I, it turns out that TAIL is useful for the space-saving of the double-array since INUM occupies 50 ~ 70 percent to TNUM, 60 ~ 70 percent if *K*5 is ignored. The result of the storage shows that SBCT is just 1.1 ~ 1.2 times larger than SSOURCE for large sets *K*3, *K*4, and *K*5. This depends on the extremely small C-VALUE and DS-tree structure being allowed to merge the common prefixes into the same arc.

In order to estimate the length cm of the G-link, the time cost m^2 of Steps (m-2 ~ m-5), and the time cost cm^2 of X_CHECK, the terms of Counts are provided as being the results of all key insertions from *K*1 to *K*5. AVE-cm, AVE-m^2, and AVE-cm^2, respectively, represent the average length of the G-link; the product of the average number of entries of ORG in Step (m-2) and the time cost m in Step (m-4); and the average count of confirming whether the CHECK's entry in X_CHECK is available or not. The values in ( ) represent the maximum value of each count. From these results, it turns out that the length of G-link, the time cost spent at Steps (m-2 ~ m-5) and the time cost spent by X_CHECK are reasonable. From the values specified by "(ordered)" and "(random)," it turns out that the building time of the double-array is independent of different orders of inserting the keys.

Let us compare the space of the double-array with those of an open hash [2], [23] and a binary tree [23], [30]. Since the keys are of varying length, the cells of the buckets of the hash and the node of the tree should not contain the keys themselves. Let p be a pointer to the beginning of a key in the key-file and let q be a pointer to link the cells of the bucket or the nodes. Thus, the cell of the bucket has pointers p and q, and the nodes of the binary

tree have one pointer p and two pointers q. We assume that q and p are two-byte pointers for the same condition as the two-byte entries of the double-array and that the hash table has 0.5 × KNUM two-byte entries. Then, for $K3$, $K4$, and $K5$, the total sizes of the hash method become from 1.7 to 2.2 times larger than SSOURCE and the total sizes of the binary tree become from 1.8 to 2.4 times larger than SSOURCE. So, it is evident that the double-array can be very compact.

TSEARCH and TINSERT respectively represent average retrieval and insertion times by SONY NEWS (2.3MIPS), for the double-array and TAIL in the main memory. The values in ( ) represent the worst-case times based on the longest key for the retrieval, and the key with the maximum cost of AVE-m^2 plus AVE-cm^2 for the insertion, respectively. As may readily be seen in the results, it turns out that the retrieval is very fast and that the insertion is about one hundred times in the case of average value, about two hundred times in the worst-case, slower than the retrieval, but it is still a reasonable speed. The worst-case retrieval of the above hash method requires searching two keys even if conflicts on the hash table are uniform, and that of the binary tree requires searching $\log_2 x$ keys for x = KNUM even if the tree is balanced. So, it is evident that the retrieval by the double-array is very fast. There are no appropriate empirical data to compare the insertion time of the double-array and those of the two methods under the same condition, but it seems that the double-array is faster than the two methods for a large static set of keys that keeps increasing because of the worst-case time complexity $O(cm^2)$, independent of x, of the double-array and those $O(x)$ of the two methods.

As shown in the experimental results, the storage SBC of the double-array (except TAIL) occupies at most 160 ~ 180K bytes for $K4$ with 23 976 keys and $K5$ with 32 344 keys; thus the presented method enables us to build a compact and fast internal retrieval table if the double-array is placed in a main memory and if TAIL is placed in the auxiliary memory along with the single-strings and the records. That is to say, it enables us to retrieve a key in only one disk access. If one megabyte is available for the double-array, then about 100 000 keys can be stored in the internal retrieval table.

VI. Conclusions

A new internal array structure for a digital search tree, a double-array, has been introduced and the updating algorithm presented. The presented method contains the following features.

1) Any key can be retrieved and deleted in constant time.

2) There is a practical upper bound of the insertion time.

3) Space efficiency is very high if the frequency of insertions is greater than that of deletions.

4) A retrieval table can be constructed for arbitrary sets of keys.

The presented double-array has been used for about 50 sets of static keys (i.e., a lexical analyzer like LEX [24] and a local code optimizer of a compiler, Roman–Hiragana conversion routine of a Japanese word processor [12], etc.), and for about 40 sets of weak static and dynamic keys (i.e., dynamic command interpreters, a bibliographic search [7], [8], [10], Japanese and English morphological dictionaries for a machine translation system [12], filtering of highly frequent English words, and a spell checker emitting correctable candidates, etc.). The double-array can efficiently manipulate the longest applicable match based on a digital search, so it is well suited for the analysis of Japanese sentences without a delimiter between words.

It would be a very interesting study to apply the algorithm presented here for updating the double-array to a pattern matching machine with failure function [1], [7], [10], the reduction of static sparse matrices by using a row displacement [9], [29], the compression of a directed graph [2], a finite state machine [4], [11] associated with a parsing table [3], [6] like YACC [17], and so on.

The developed system will be provided on request to any readers. Please feel free to contact me.

Acknowledgment

The author is grateful to Mr. S. Yasutome for the implementation of the information retrieval system presented.

References

[1] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid bibliographic search," *Commun. ACM*, vol. 18, pp. 333–340, June 1975.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*. Reading, MA: Addison-Wesley, 1983.

[3] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers Principles, Techniques, and Tools*. Reading MA: Addison-Wesley, 1986, ch. 2.

[4] J. Aoe, Y. Yamamoto, and R. Shimada, "An efficient method for storing and retrieving finite state machines" (in Japanese), *IECE Trans.*, vol. J65-D, pp. 1235–1242, Oct. 1982; (in English), *Electronica Japonica*, vol. 13.

[5] ——, "A practical method for reducing sparse matrices with invariant entries," *Int. J. Comput. Math.*, vol. 12, pp. 97–111, Nov. 1982.

[6] ——, "A method for reducing weak precedence parsers," *IEEE Trans. Software Eng.*, vol. SE-10, pp. 25–30, Jan. 1983.

[7] ——, "An efficient method for storing and retrieving pattern matching machines" (in Japanese), *Trans. IPS Japan*, vol. 24, pp. 414–420, July 1983.

[8] ——, "A method for improving string pattern matching machines," *IEEE Trans. Software Eng.*, vol. SE-10, pp. 116–120, Jan. 1984.

[9] ——, "An efficient algorithm of reducing sparse matrices by row displacements" (in Japanese), *Trans. IPS Japan*, vol. 26, pp. 211–218, Mar. 1985.

[10] ——, "An efficient implementation of static string pattern matching machines," in *Proc. First Int. Conf. Supercomputing*, Dec. 1985, pp. 491–498; also in *IEEE Trans. Software Eng.*, vol. 15, pp. 1010–1016, Aug. 1989.

[11] J. Aoe, "An efficient implementation of finite state machines using a double-array structure" (in Japanese), *IECE Trans.*, vol. J70-D, pp. 653–662, Apr. 1987.

[12] J. Aoe and M. Fujikawa, "An efficient representation of hierarchical semantic primitives—An aid to machine translation systems," in *Proc. Second Int. Conf. Supercomputing*, May 1987, pp. 361–370.

[13] J. Aoe, S. Yasutome, and T. Sato, "An efficient digital search algorithm by using a double-array structure," in *Proc. Twelfth Int. Computer Software and Applications Conf.*, Oct. 1988, pp. 472–479.

[14] F. Berman, E. Bock, E. Dittert, M. J. O'Donnelland, and D. Plank,

"Collections of functions for perfect hashing," *SIAM J. Comput.*, vol. 15, pp. 604-618, Feb. 1986.
[15] R. J. Cichelli, "Minimal perfect functions made simple," *Commun. ACM*, vol. 23, pp. 17-19, Jan. 1980.
[16] G. V. Cormack, R. N. S. Horspool, and M. Kaiserswerth," Practical perfect hashing," *Comput. J.*, vol. 28, pp. 54-58, Jan. 1985.
[17] S. C. Johnson, "YACC—Yet another compiler-compiler," Bell Lab., NJ, Comput. Sci. Tech. Rep. 32, pp. 1-34, 1975.
[18] G. Jaeschke, "Reciprocal hashing: A method for generating minimal perfect hashing functions," *Commun. ACM*, vol. 24, pp. 829-833, Dec. 1981.
[19] W. D. Jonge, A. S. Tanenbaum, and R. P. Reit, "Two access methods using compact binary trees," *IEEE Trans. Software Eng.*, vol. SE-13, pp. 799-810, July 1987.
[20] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, "Extensible hashing—A fast access method for dynamic files," *ACM Trans. Database Syst.*, vol. 4, pp. 315-344, Sept. 1979.
[21] E. Fredkin, "Trie memory," *Commun. ACM*, vol. 3, pp. 490-500, Sept. 1960.
[22] M. L. Fredman, J. Komlos, and E. Szemeredi, "String a sparse table with $O(1)$ worst case access time," *J. ACM*, vol. 31, no. 3, pp. 538-544, Mar. 1984.
[23] D. E. Knuth, *The Art of Computer Programming*, vol. I, *Fundamental Algorithms*, vol. III, *Sorting and Searching*. Reading, MA: Addison-Wesley, 1973, pp. 295-304, 481-505.
[24] M. E. Lesk, "Lex—A lexical analyzer generator," Bell Lab. NJ, Comput. Sci. Tech. Rep. 39, pp. 1-13, Oct. 1975.
[25] K. Maly, "Compressed tries," *Commun. ACM*, vol. 19, no. 7, pp. 409-415, July 1976.
[26] J. L. Peterson, *Computer Programs for Spelling Correction* (Lecture Notes in Computer Science). New York: Springer-Verlag, 1980.
[27] B. A. Sheil, "Median split trees: A fast lookup techniques for frequency occurring keys," *Commun. ACM*, vol. 21, pp. 947-959, Nov. 1978.
[28] R. Sprugnoli, "Perfect hashing functions: A single probe retrieving method for static sets," *Commun. ACM*, vol. 20, pp. 841-850, Nov. 1977.
[29] R. E. Tarjan and A. C. Yao, "Storing a sparse table," *Commun. ACM*, vol. 22, pp. 606-611, Nov. 1979.
[30] T. A. Standish, *Data Structure Techniques*. Reading *MA: Addison-Wesley, 1980.*

Jun-ichi Aoe (M'87) received the B.E. and M.E. degrees in electronic engineering from the University of Tokushima, Tokushima, Japan, in 1974 and 1976, respectively, and the Ph.D. degree in communication engineering from the University of Osaka, Japan, in 1980.

Since 1976 he has been with the University of Tokushima. He is currently an Associate Professor in Information Science and Systems Engineering. He is the author of about 35 scientific papers. His research interests include software engineering and natural language processing.

Dr. Aoe is a member of the Association for Computing Machinery, the American Association for Artificial Intelligence, the Association for Computational Linguistics, the Institute of Electronics, Information, and Communication Engineers of Japan, the Japan Society for Software Science and Technology, the Information Processing Society of Japan, and the Japanese Society for Artificial Intelligence.