SOFIWARE—PRACTICE AND EXPERIENCE, VOL. 22(9), 695–721 (SEPTEMBER 1992)

An Efficient Implementation of Trie Structures

JUN-ICHI AOE AND KATSUSHI MORIMOTO
Department of Information Science and Intelligent Systems, University of Tokushima, Minami-Josanjima- Cho, Tokushima-Shi 770, Japan

AND

TAKASHI SATO
Department of Arts and Sciences, Osaka Kyoiku University, 3-1-1, Ikeda 563, Japan

SUMMARY

A new internal array structure, called a *double-array,* implementing a *trie* structure is presented. The double-array combines the fast access of a matrix form with the compactness of a *list* form. The algorithms for retrieval, insertion and deletion are introduced through examples. Although insertion is rather slow, it is still practical, and both the deletion and the retrieval time can be improved from the list form. From the comparison with the list for various large sets of keys, it is shown that the size of the double-array can be about 17 per cent smaller than that of the list, and that the retrieval speed of the double-array can be from 3·1 to 5·1 times faster than that of the list.

KEY WORDS Dictionary Information retrieval Key retrieval strategies Natural language processing

INTRODUCTION

In many information retrieval applications, it is necessary to be able to adopt a *trie* search [1,2] for looking at the input character by character. Examples include a lexical analyser of a compiler, a bibliographic search [3,5] a spelling checker, [6] and morphological dictionaries [7] in natural language processing, and so on. 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 either a new node or a null pointer. Figure 1 gives an example of an array-structured trie for the set *K* = {baby, bachelor, badge, jar}. Retrieval, deletion and insertion on the trie are very fast, but it takes lots of space because the space complexity is proportional to the product of the number of nodes and the number of characters. A well-known strategy for compressing the trie is to list the arcs out of each node, with the null pointer at the end of the list. Figure 2 shows an example of a list-structured trie for the set *K*. The list-structured trie enables us to save the space by use of null pointers of the array-structured trie, but the retrieval becomes slow if there are many arcs leaving each node.

This paper presents a technique of compressing the trie into two one-dimensional arrays BASE and CHECK called a double-array. In the double-array, non-empty locations of node *n* are mapped, by the array BASE, into the array CHECK such that

0038–0644/92/090695–27$18.50

Received 27 February 1990
Revised 14 April 1992

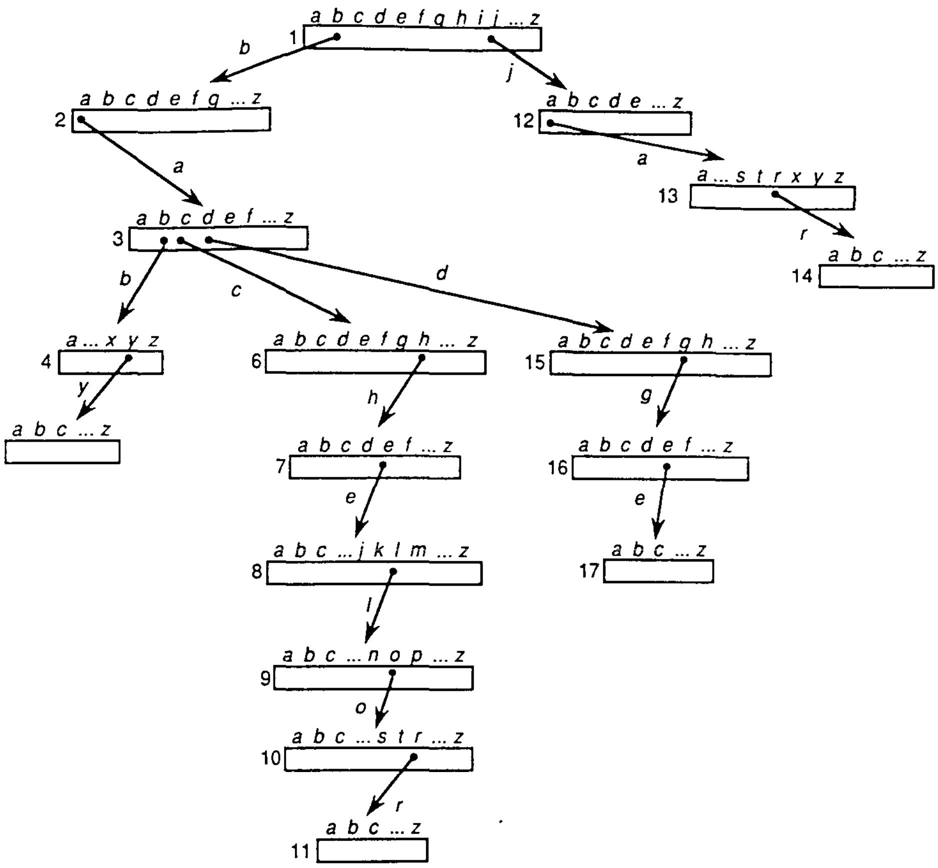


Figure 1. An array-structured trie for bachelor, baby, badge, jar

no two non-empty locations in each node are mapped to the same position in CHECK. Each arc of the trie can be retrieved from the double-array in $O(1)$ time, that is, the worst-case time complexity for retrieving a key becomes $O(k)$ for the length k of that key. The trie has many nodes for a large set of keys, so it is important to make the double-array compact. In order to implement the trie for a large set of keys, the double-array stores only as much of the prefix in the trie as is necessary to disambiguate the key, and the tail of the key not needed for further disambiguation is stored in a string array, denoted as TAIL.

REPRESENTATION OF A TRIE

A trie is a tree structure in which each path from the root to a leaf corresponds to one key in the represented set. The paths in the trie correspond to the characters of the keys in the set. To avoid confusion between words like 'the' and 'then', a special endmarker symbol, #, is used at the end of each word in the set.

The following definitions will be used in the explanations that follow. K is the set of keys represented by the trie. The trie is formed of nodes and arcs. Arc labels

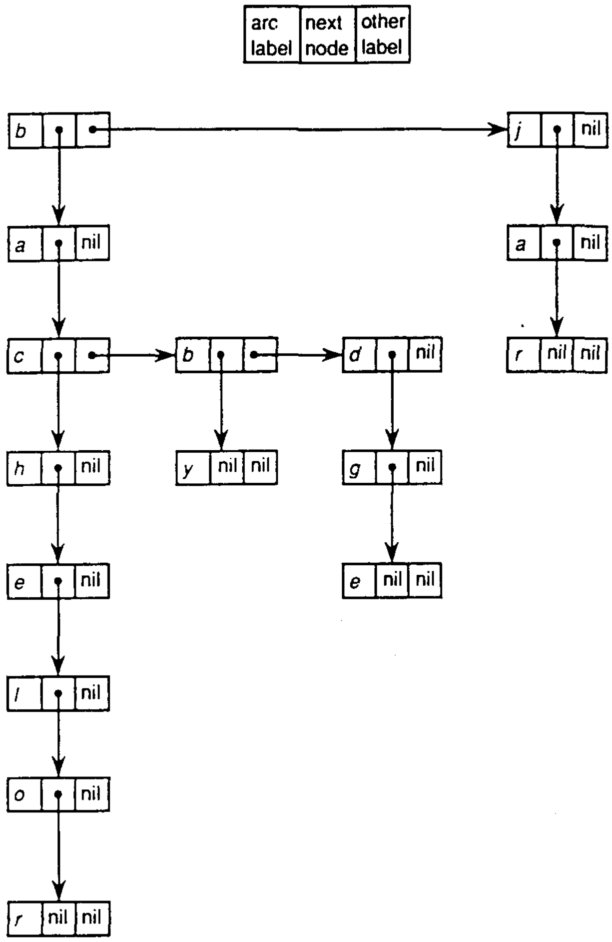


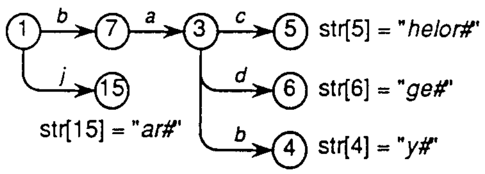
Figure 2. A list-structured trie for bachelor, baby, badge, jar

consist of symbols, called characters. An arc labeled *a* from node *n* to *m* is represented by the notation $g(n, a)=m$.

For a key in *K*, the node *m* with $g(n,a)=m$ is a *separate node* if *a* is a sufficient character (or arc label) for distinguishing that key from all others in *K*. The concatenation of the arc labels from separate node *m* to the terminal node is called a *single string* for *m*, denoted as STR[*m*]. The characters of key *K* remaining after the single string is deleted from *K* are called the *tail* of *K*. A tree constructed only of the arcs from the root to the separate nodes for all keys in *K is* called the *reduced trie.*

An example of a reduced trie for the set *K*= {baby#, bachelor#, badge#, jar#} is shown in Figure 3. This same reduced trie representation is also shown in Figure 3, using a double-array and an array of characters for tail storage. Question marks (?) in TAIL indicate garbage; their use will be explaining when analysing the insertion and deletion algorithms.

The following relations between the reduced trie and the double-array shown in Figure 3 exist:



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
BASE	4	0	1	-15	-1	-12	1	0	0	0	0	0	0	0	-9
CHECK	0	0	7	3	3	3	1	0	0	0	0	0	0	0	1

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
TAIL	*h*	*e*	*l*	*o*	*r*	*#*	*?*	*?*	*a*	*r*	*#*	*g*	*e*	*#*	*y*	*#*

POS=17

Figure 3. The reduced trie and the double-array for K

1. If there is an arc $g(n,a)=m$ on the reduced trie, then BASE [n] $+a=m$ and CHECK [m]= n. {For the arc labels: '#'=1, 'a'=2, 'b'=3, 'c'=4, etc. }
2. If the node m is a separate node such that the tail string STR [m] = $b_1, b_2, \ldots . b_h$ then
   (a) BASE [m]< 0
   (b) let, p =– BASE [m], TAIL [p]= b_1, TAIL [p +l]= $b_2, \ldots .$ TAIL [$p+h$ -1]= b_h

These two relations will remain throughout this paper.

Retrieval

Retrieval operations using the double array are straightforward. For example, to retrieve 'bachelor#' from the double-array shown in Figure 3, the following steps are performed:

Step 1. Store the root node at position 1 of BASE in the double-array, and start at that position. The value for the character 'b' is 3, so from relation 1 above

BASE [n]+ a =BASE [l]+ 'b'= BASE [l]+3=4+3=7

Observe that CHECK [7] = 1

Step 2. Since the value found for BASE in step 1 is positive, proceed. Use the value 7 from step 1 as the new index into BASE, and the value 2 for character 'a', so:

BASE [7]+ 'a'= BASE [7] +2=l+2=3, and CHECK [3]=7

Steps 3,4. Proceed as above, using 4 for 'c':

BASE [3]+ 'c'= BASE [3]+4=1+4=5, and CHECK [5]=3

Step 5. The value in BASE [5] is – 1. A negative value indicates that the rest of the word is located in TAIL, starting at TAIL [– BASE [5]] = TAIL [1]. The other words in the list can be retrieved using a similar technique, always starting at the root node at position 1 in BASE.

Observe that retrieval involves only direct array lookups (no searching is required) and addition, making retrievals extremely efficient in this implementation.

Insertion

Insertion into a double-array is also straightforward. During insertion, any of the four cases below arises:

1. Insertion of the new word when the double-array is empty.
2. Insertion of the new word without any collisions.
3. Insertion of the new word with a collision; in this case, additional characters must be added to the BASE and characters must be removed from the TAIL array to resolve the collision, but nothing already in the BASE array must be removed.
4. When insertion of the new word with a collision as in case 3 occurs, values in the BASE array must be moved.

A collision indicates that two different characters have the same index value within the double-array. These four insertion cases will be demonstrated by adding 'bachelor' (case 1), 'jar#' (case 2), 'badge#' (case 3) and 'baby#' (case 4) to an empty double-array structure like the one shown in Figure 4. We define the size denoted by DA_SIZE, of the double-array as the maximum index of the non-zero entries of CHECK. Note that the BASE and CHECK entries exceeding the DA_SIZE can be allocated dynamically as zero entries if needed.

Case 1: the insertion of the new word when the double-array is empty

To insert for example 'bachelor#', perform the following steps:

Step 1. Start at position 1 of BASE in the double-array. The value for the character 'b' is 3, so:

$$\text{BASE }[1]+\text{'b'} = \text{BASE }[1]+3=1+3=4, \text{ and CHECK }[4]=0 \neq 1$$

Step 2. The value 0 in CHECK [4] indicates insertion of the rest of the word. That

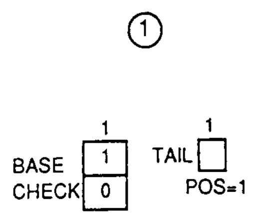


Figure 4. The reduced trie and the double-array with no data

is, ‘b’ is defined on the double-array (by the relation g(l ,‘b’) =4) so store into TAIL the remaining string, ‘achelor#’.

Step 3. Set

$$\text{BASE}\,[4] \leftarrow -\,\text{POS} = -1$$

to indicate that the rest of the word is stored into TAIL starting from position POS. And set

$$\text{CHECK}\,[4] \leftarrow 1$$

to indicate the node number it comes from, within the double-array.

Step 4. Set the pointer to TAIL

$$\text{POS} \leftarrow 9$$

which is the location for the next insertion.

Figure 5 shows the reduced trie and the double-array after inserting ‘bachelor#’.

Case 2: *insertion, when the new word is inserted without collisions*

Perform the following steps to insert ‘jar#’:

Step 1. Start at position 1 of BASE in the double-array. The value for the character ‘j’ is 11, so:

$$\text{BASE}\,[l]+\text{‘j’}= \text{BASE}\,[1]+11=1+11=12, \text{ and CHECK}\,[12]= 0 \neq l$$

Step 2. The value 0 in CHECK [12] indicates insertion of the rest of the word implying that no collision with ‘bachelor#’ occurred. Store the remaining string, ‘ar#’, into TAIL from position POS = 9.

Step 3. Set

$$\text{BASE}\,[12] \leftarrow -\text{POS} = -9$$

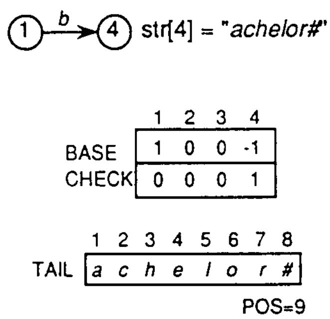


Figure 5. The reduced trie and the double-array for insertion of ‘bachelor#’

to indicate that the rest of the word is stored into TAIL starting from position POS. And set

CHECK [12] ←1

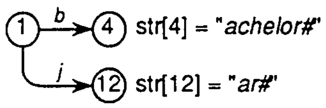
to indicate the node number it comes from, within the double-array.

Step 4. Set the pointer to TAIL

POS ←12

which is the location for the next insertion.

As can be observed, there is no difference between case 1 and case 2 insertions so their classification is only conceptual and not operational. The resulting reduced trie and double-array after inserting 'jar#' are shown in Figure 6.



	1	2	3	4	5	6	7	8	9	10	11	12
BASE	1	0	0	-1	0	0	0	0	0	0	0	-9
CHECK	0	0	0	1	0	0	0	0	0	0	0	1

	1	2	3	4	5	6	7	8	9	10	11
TAIL	*a*	*c*	*h*	*e*	*l*	*o*	*r*	*#*	*a*	*r*	*#*

POS=12

Figure 6. The reduced trie and the double-array for insertion of 'jar#'

To study the insertion cases 3 and 4, consider the function X_CHECK(LIST) which returns the minimum integer q such that $q > 0$ and CHECK [$q+c$] =0 for all c in LIST. q always starts with the value 1 and has unitary increments at analysis time.

Case 3: *insertion, when a collision occurs*

Perform the following steps for 'badge#':

Step 1. Start at position 1 of BASE in the double-array. The value for the character 'b' is 3, so:

BASE [l]+'b'= BASE [l]+3=l+3=4, and CHECK [4]=1

The non-zero value in CHECK [4] indicates that an arc definition from the node indicated by the value in CHECK [4], i.e. 1, to node 4 exists.

Step 2. Since the value found for BASE in step 1 is positive, proceed. The value 4 from step 1 is used as the new index into BASE, but:

BASE [4]=–1

This negative value indicates that searching has finished and string comparison is to be performed.

Step 3. Retrieve from TAIL the string starting at position – BASE [4] = 1, i.e. 'achelor#' and compare it with the remaining part of the string to be inserted, i.e. 'adge#'. As comparison fails, that is the strings are different from each other, insert the common prefix into the double-array as indicated in steps 4, 5, and 6.

Step 4. Save the current value of – BASE [4] into a temporal variable:

TEMP ← -BASE [4]=1

Step 5. Calculate X_CHECK [{'a'}] for the prefix character 'a' common to the two strings 'adge#' and 'achelor#':

CHECK [q+a]= CHECK [l+'a'] = CHECK [l+2]= CHECK [3]=0

The value of q, i.e. 1, is the candidate for new value of BASE [4], and the 0 value of CHECK [3] indicates that node 3 is available, so:

Step 6. Store the new value for BASE [4]:

BASE [4] ← q = 1

And the new value of CHECK for the available node 3:

CHECK [BASE [4]+'a'] = CHECK [l+2] = CHECK [3] ←4

This indicates an arc definition from the node value in CHECK [3], i.e. 4, to node 3.

Note: Due to the common prefix in this example, steps 5 and 6 are not repeated, but these two steps must be performed as many times as prefix values exist.

Step 7. To store the remaining strings 'chelor#' and 'dge#', calculate the value to be stored into BASE [3] for two arc labels 'c' and 'd' according to the closest neighbour available by X_CHECK ({'c' ,'d'}) as follows.

For 'c': CHECK [q+'c'] = CHECK [l +4] = CHECK [5] =0 ⇒ available
For 'd': CHECK [q+'d'] = CHECK [l+5]= CHECK [6] =0 ⇒ available

as q= 1 is the candidate, set

BASE [3] ← 1

Step 8. Calculate the node number for the reference to 'chelor#' in BASE and CHECK taking as parameter the first character of the string:

BASE [3]+'c'=1+4=5
BASE [5] ← –TEMP= –1 and CHECK [5] ← 3

Establishing the reference to TAIL by means of BASE, and the arc definition between states 3 and 5 by means of CHECK.

Step 9. Store the rest of the string 'helor#' into TAIL starting at position BASE [5] = 1, but TAIL [7] and TAIL [8] become *garbage* in Figure 7.

Step 10. For the remaining string 'dge#':

BASE [3]+'d'=l+5=6
BASE [6] $\leftarrow -$ POS $= -12$ and CHECK [6] $\leftarrow 3$
Store 'ge#' into TAIL starting at POS .

Step 11. Finally set POS to the new insertion position, at the end of the used part of TAIL.

POS $\leftarrow$ 12+length['ge# ']=12+3=15

In summary, when a collision occurs the prefix common to the collisioned strings is extracted from TAIL and inserted into the double-array. The values within the double-array for the collisioned strings, including the new string, are moved to the nearest neighbour position available and adjusted to such new positions (see Figure 7).

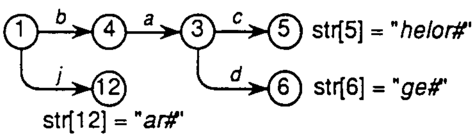
Case 4: *insertion, when a new word is inserted with a collision*

As in case 3, values in the BASE array must be moved; perform the following steps for 'baby #':

Step 1. The root node is stored at position 1 of BASE in the double-array, so start at position 1. For the first three characters the values for BASE and CHECK are:

BASE [l]+'b'= BASE [l]+3=l+3=4, and CHECK [4]=1
BASE [4]+'a'= BASE [4] +2=l+2=3, and CHECK [3]=4
BASE [3]+'b'= BASE [3]+3=l+3=4, and CHECK [4]=l $\neq$ 3

The inconsistency in CHECK [4] indicates an undefined status, this means



	1	2	3	4	5	6	7	8	9	10	11	12
BASE	1	0	1	1	-1	-12	0	0	0	0	0	-9
CHECK	0	0	4	1	3	3	0	0	0	0	0	1

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
TAIL	*h*	*e*	*l*	*o*	*r*	*#*	*?*	*?*	*a*	*r*	*#*	*g*	*e*	*#*

POS=15

Figure 7. The reduced trie and the double-array for insertion of 'badge#'

that the values for nodes 1 and 3 should be modified to allow the new insertion, thus proceed as follows.

Step 2. Set a temporal variable TEMP_NODE1 to

TEMP_NODE1 ← BASE [3]+ 'b'=1+3=4

If CHECK [4] were equal to 0 then this will imply availability, so the case would be a straightforward insertion of the string into TAIL at POS. As this is not the case, do the following.

Step 3. Store in a list, having as index number the node number where the inconsistency was found, the characters that correspond to the arcs leaving that node.

LIST [3] ←['c','d']

And in another list, having as index the last CHECK value, the characters that correspond to the arcs leaving that node.

LIST [l] ←['b','j']

Step 4. As the purpose here is to associate the new string with node 3, compare the length of the two lists incrementing the length of LIST [3] by 1. This increment is necessary to consider the new character to be added to node 3.

compare (length [LIST [3]]+ 1, length [LIST [l]])= *compare* (3,2)

If length [LIST [3]]+ 1 < length [LIST [l]] the current node, 3, would be modified. But as length [LIST [3]] + 1 $\geq$ length [LIST [l]] modify the alternative node, i.e. node 1, as follows.

Step 5. Set a temporal variable for the node referenced by BASE.

TEMP_BASE ← BASE [l]=l

and calculate a new BASE using LIST [l] according to the closest neighbour available as follows:

X_CHECK ['b']: CHECK [q+'b']
= CHECK [1+3] =CHECK [4]=l $\neq$ 0
CHECK [2+3] =CHECK [5]=–l $\neq$ 0
CHECK [3+3] =CHECK [6]=–14 $\neq$ 0
CHECK [4+3] =CHECK [7]=0 ⟹ available

and

X_CHECK ['j']: CHECK [q+'j'] = CHECK [4+11]=CHECK [15]
=0 ⟹ available

as q = 4 is the candidate, set

BASE [1] ← 4

Step 6. For 'b', store the value for the states to be modified in temporal variables:

TEMP_NODE1 ← TEMP_BASE+ 'b' =1+3=4
TEMP_NODE2 ← BASE[1]+ 'b'=4+3=7

Copy the BASE value from the original status to the new status:

BASE[TEMP_NODE2] ← BASE[TEMP_NODE1]

i.e.

BASE [7] ← BASE [4] = 1

And set the CHECK value for the new node:

CHECK [TEMP_NODE2] =CHECK [7] ← CHECK [4]=1

Step 7. As

BASE[TEMP_NODE1] = BASE [4]=1>0

That is, the BASE value for the original status is an internal pointer instead of a pointer to TAIL, calculate the offset ω to obtain the node value to be modified to point to the new node.

CHECK [BASE] TEMP_ NODE1]+ ω] =TEMP_NODE1

i.e.

CHECK [BASE [4]+ ω]=4; CHECK [1+ ω] = 4 ⇒ ω=2

and modify CHECK to point to the new status:

CHECK [BASE [4]+2] = CHECK [l+2]= CHECK [3] ← TEMP_NODE2 =7

Step 8. Initialize the old BASE and CHECK:

BASE [TEMP_NODE1] = BASE [4] ← 0
CHECK [TEMP_NODE1] = CHECK [4] ← 0

Step 9. For 'j', store the value for the states to be modified in temporal variables:

TEMP_NODE1 ← TEMP_BASE + 'j' =1+11=12
TEMP_NODE2 ← BASE [1]+ 'j' =4+11=15

Copy the BASE value from the original status to the new status:

BASE [TEMP_NODE2 ← BASE [TEMP_NODE1]

i.e.

BASE [15] ← BASE [12]=-9

And set the CHECK value for the new node:

CHECK [TEMP_NODE2] = CHECK [15] ← CHECK [12]=1

Step 10. As

BASE [TEMP_NODE1] =BASE [12]=–9<0

That is, the BASE value for the original status is a pointer to TAIL, so only initialize the old BASE and CHECK:

BASE [TEMP_NODE1] = BASE [12] ← 0
CHECK [TEMP_NODE1] = CHECK [12] ← 0

Now the conflict generated by the collision of ‘b’ from ba <u>*b*</u> y has been solved. Finally, insert the remaining part of the new string ‘by#’ into TAIL.

Step 11. Considering the original BASE node number, i.e. 3, where the inconsistency was generated (see step 3) as pivot, store in a temporal variable the node number for the reference to the new string:

TEMP_NODE ← BASE [3]+ ‘b’= 1+3=4

Step 12. Store in the new BASE node the pointer to TAIL for the new string:

BASE [TEMP_NODE] =BASE [4] ← – POS =–15

and in CHECK the internal pointer to the referencing node

CHECK [TEMP_NODE] =CHECK [4] ← 3

Step 13. Insert the new string in TAIL:

TAIL [POS] =TAIL [15] +‘y#’

Step 14. To finish set the pointer to TAIL to its new value:

POS ← POS+ length [’y#’] = 15+2= 17

In summary, when a collision occurs the values in the double-array must be moved since there is no room for the specification of the new word, the node that collided, or the previous node, as indicated by CHECK. The node with fewer branches is then moved to another place within the double-array to give space for the specification of more nodes to the node that collided. The connecting node numbers are modified to point to the new place within the double-array where the node specification was moved. Finally, the new string is inserted (see Figure 3).

Deletion

Deletion of words from a double-array is also straightforward. Deletion has the same scanning process as in case 2 insertion. In fact, the only difference consists of resetting, for the word to be deleted, the necessary pointers to TAIL within the double-array.

For example, to delete the word 'badge#' perform the following:

Step 1. The root node is stored at position 1 of BASE in the double-array, so start at position 1. For the first three characters the values for BASE and CHECK are

BASE [1]+'b'= BASE [1]+3=4+3=7, and CHECK [7]=1
BASE [7]+'a'= BASE [7]+2=l+2=3, and CHECK [3]=7
BASE [3]+'d'=BASE [3]+5=l+5=6, and CHECK [6]=3
BASE [6] = – 12<0 ⇒ separate node

The separate node is the pointer to TAIL.

Step 2. Compare the remaining part of the string, i.e. 'ge#' with the string in TAIL at –BASE [6]=12

compare ('ge#','ge#')

Step 3. As both strings are equal, reset the corresponding pointers to TAIL within the double-array.

BASE [6] ←0
CHECK [6] ←0

Since the pointers to TAIL for 'ge#' do not exist any more, the portion of TAIL for 'ge#' becomes garbage as in Figure 8, so that space is available for future usage.

EVALUATION

Comparison with list structures

A well-known representation of the trie is to use a list form .[2-4] Although the list can efficiently compress and update the trie, its access is slow. Suppose that each node number of the reduced trie is implemented by two bytes. Each node occupies four bytes in the double-array and five bytes in the list form (based on the components arc label, pointer, next node). A reasonable compromise is to store the most frequently used nodes (such as the root node) as direct access tables in which the next node can be determined by directly indexing into the table with the current input symbol, and store the other nodes in the list form. In the simulation, and within this paper

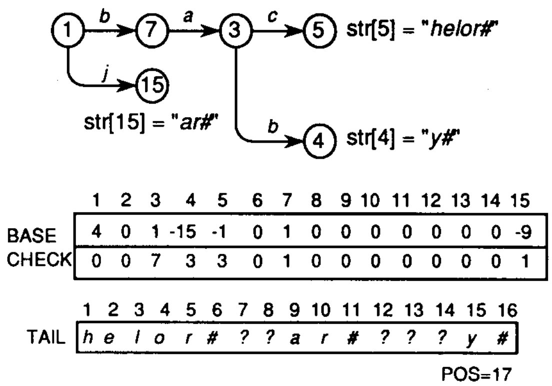


Figure 8. The reduced trie and the double-array for deletion of 'badge#'

too, this revised list form is used for the reduced trie and the single strings were stored into TAIL.

The following five kinds of sets of keys were used for the experimental observations.

KW1, KW2: The reserved words for Pascal and COBOL, respectively.
KW3: Commands in UNIX 4.2BSD.
KW4: Main city names in the world.
KW5: Katakana for a Japanese word dictionary.
KW6: Words for an English dictionary.
KW7: Kanji for a Japanese word dictionary.

Table I. The results of storage

	KW1	KW2	KW3	KW4	KW5	KW6	KW7
Numbers and length							
KEY_NUM	35	310	657	1480	23,976	32,344	65,857
TOTAL_NUM	161	1558	2781	9742	100,235	88,801	259,324
NODE_NUM	52	611	1051	2461	40,498	45,383	98,713
KEY_LEN	5·1	7·5	6·9	9·5	8·2	5·6	6·8
Storages (kilo-bytes)							
SPACE_DOUBLE	0·63	2·78	4·68	10·12	161	182	396
SPACE_LIST	0·26	3·06	5·23	12·31	202	227	494
SPACE_TAIL	0·11	0·95	173	7·28	60	44	161
SPACE_DOUBLE_TAIL	0·74	3·64	6·41	17·40	221	226	557
SPACE_LIST_TAIL	0·37	4·01	696	19·59	262	271	655
SPACE_SOURCE	0·28	2·33	4·54	14·08	197	182	671
Rate(%)							
RED_RATE	67·7	12·2	11·3	2·7	0·1	0·1	0·1

Note that the double-array for KW5–KW7 is divided into several blocks in order to keep two-byte entries of the double-array and the list form.

Space efficiency

In Table I, KEY_NUM is the number of keys; TOTAL_NUM is the number of nodes in the unreduced trie; NODE_NUM is the number of nodes for the reduced trie; and KEY_LEN is the average length of keys. SPACE_DOUBLE stands for storage of the double-array for the reduced trie; SPACE_LIST for storage of the list form for the reduced trie; SPACE_TAIL for storage of TAIL; SPACE_DOUBLE_TAIL for storage of SPACE_DOUBLE plus SPACE_TAIL; SPACE_LIST_TAIL for storage of SPACE_LIST plus SPACE_TAIL; and SPACE_SOURCE for storage of a source file of all keys with a delimiter between keys. Note that TAIL is reorganized to remove the redundant characters. RED_RATE represents the percentage of the number of redundant indexes to NODE_NUM for the final double-array, that is to say, it shows the space efficiency of the compression method presented.

From the results in Table I, for all sets except KW1, it can be seen that storage in SPACE_DOUBLE_TAIL of the double-array is about 8 to 17 per cent less than storage in SPACE_LIST_TAIL of the list form. In particular, the result of the storage shows that SPACE_DOUBLE_TAIL is just from 1.1 to 1.2 times larger than SPACE_SOURCE for large sets. This depends on the extremely small RED_RATE of the double-array for large sets.

Time efficiency

The key concept in the algorithm presented is that during case 4 insertion, only one part of the trie (the part of the trie at the collision, having fewer arcs) has to be moved to the first available blank location in BASE which is big enough to hold that piece. This makes the insertion operation expensive, but the fact that only one part is moved is the reason why the cost of insertion does not become $O(n^2)$ for n states. The worst-case time complexity of the presented insertion algorithm depends on the function X_CHECK(LIST) invoked at case 4 insertion because it travels along all the indexes of the double-array to search available positions for each symbol in LIST. The insertion can keep a practical speed for small sets of keys whose number is less than several hundred, but it becomes slow for large sets of keys. In order to avoid the overhead, the double-array with bidirectional links connecting the available, or redundant, indexes is introduced as follows.

Let $r_1, r_2, \ldots. r_m$ be the increasing order of the available indexes on the double-array:

$$\text{CHECK}[r_i] = -r_{(i+1)},\ \text{CHECK}[r_m] = -1;$$
$$\text{BASE}[r_1] = -1,\ \text{BASE}[r_{(i+1)}] = -r_i \text{ for } 1 \leq m-1$$

where the value – 1 represents each end of both links. Each head of both links is represented by global variables. Note that the available entries in the CHECK array must be confirmed by negative integers instead of zeros and that the initializing entries of the double-array require appending those entries into the links. This revision is simple and independent of the space occupied by the double-array, so the revised double-array is used in the following evaluation.

Let DOUBLE and LIST stand for the double-array and the list form, respectively. All evaluations are demonstrated by the average and worst-case times for one key, denoted as DOUBLE -Average, DOUBLE -Worst, LIST -Average and LIST -Worst. Note that all times were counted under the condition that each set of keys and the retrieval table were in the main memory. Figure 9 shows the comparison of DOUBLE and LIST for the insertion under the condition that the double-array and the list are built incrementally. The results show that the insertion of the double-array spends from 5 to 9 times more than the insertion of the list, but it is still a practical speed. The number of redundant indexes in the double-array closely reaches a very small constant value for large sets KW4–KW7, so the insertion time is also constant.

Figure 10 shows the comparison of DOUBLE and LIST for the deletion under the condition that one key is removed from the double-array and the list. From the results it can be observed that the deletion of the double-array is from 1·2 to 1·5 times in the average case, from 1·5 to 2·5 times in the worst case, faster than that of the list. This efficiency directly depends on the retrieval time because the deletion requires, the search to determine whether the key has been registered in the

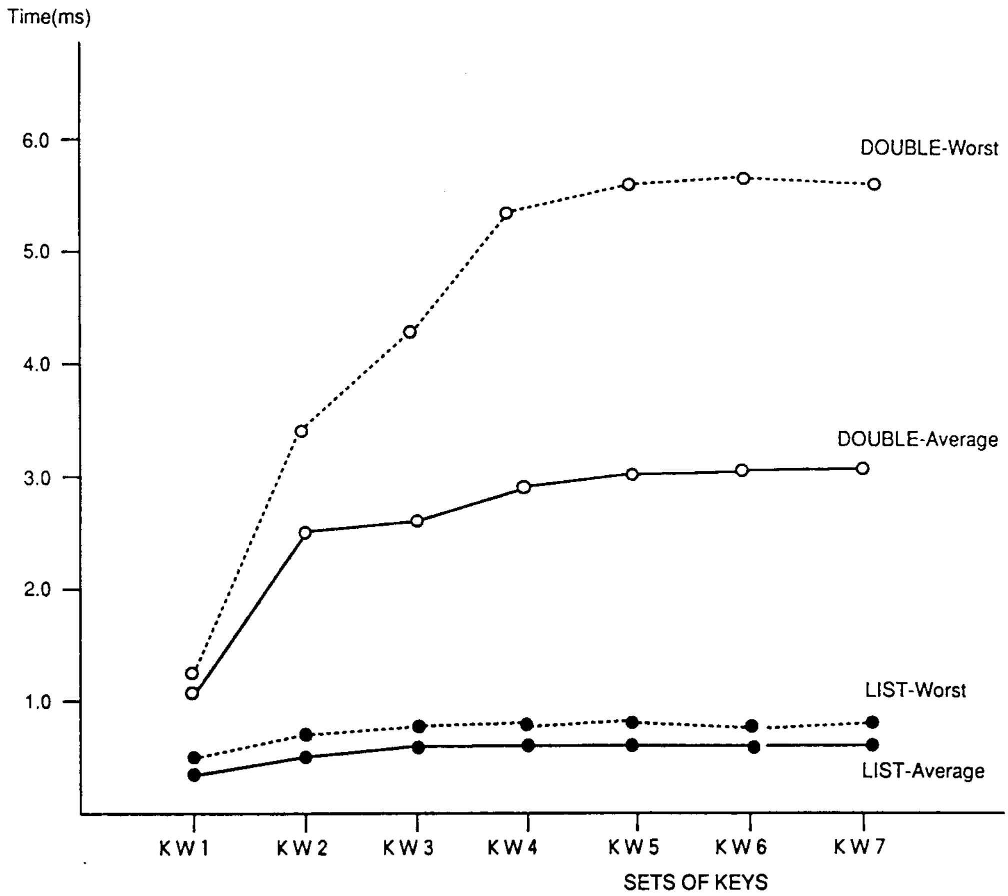


Figure 9. The results of insertion time

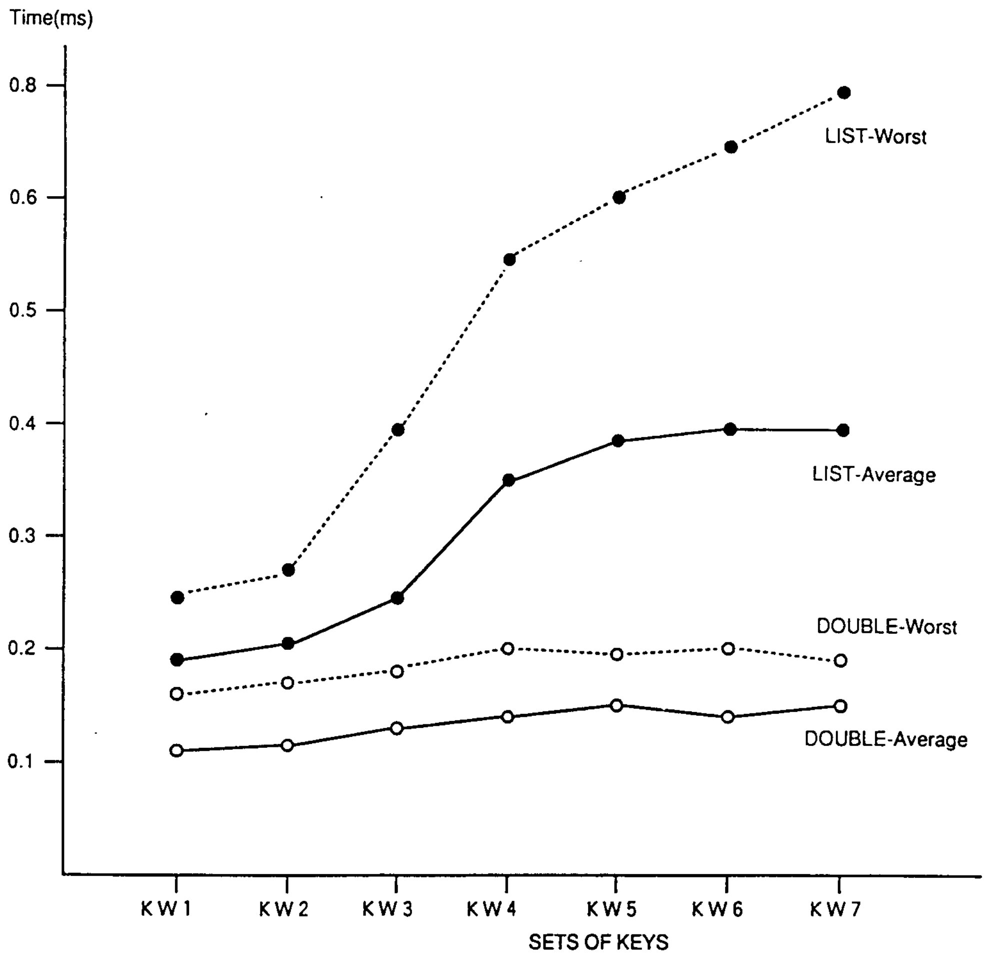


Figure 10. The results of deletion time

dictionary or not. The number of keys to be stored is restricted to keep two-byte entries, so LIST -worst for KW4–KW7 becomes constant as shown in Figure 10.

Figure 11 represents the comparison of DOUBLE and LIST for the retrieval time. Observe from the results that search within the double-array is from 1·2 to 3·1 times faster in the average case, and from 1·5 to 5·2 times faster in the worst case, than that of the list. The gap of the retrieval time between DOUBLE and LIST for KW5-KW7 grows along with the increasing of the number of keys to be stored in both the divided double-array and the list. This is because the retrieval time of DOUBLE is constant and that of LIST is proportional to the number of arcs leaving the node for the reduced trie.

There are other ways [8,9] to compress tries. However, the 'double-offset indexing' approach [8] of trie-gen in the GNU, using both the compression algorithm of a sparse matrix as well as compressed tries [9] are just tools for static sets of keys.

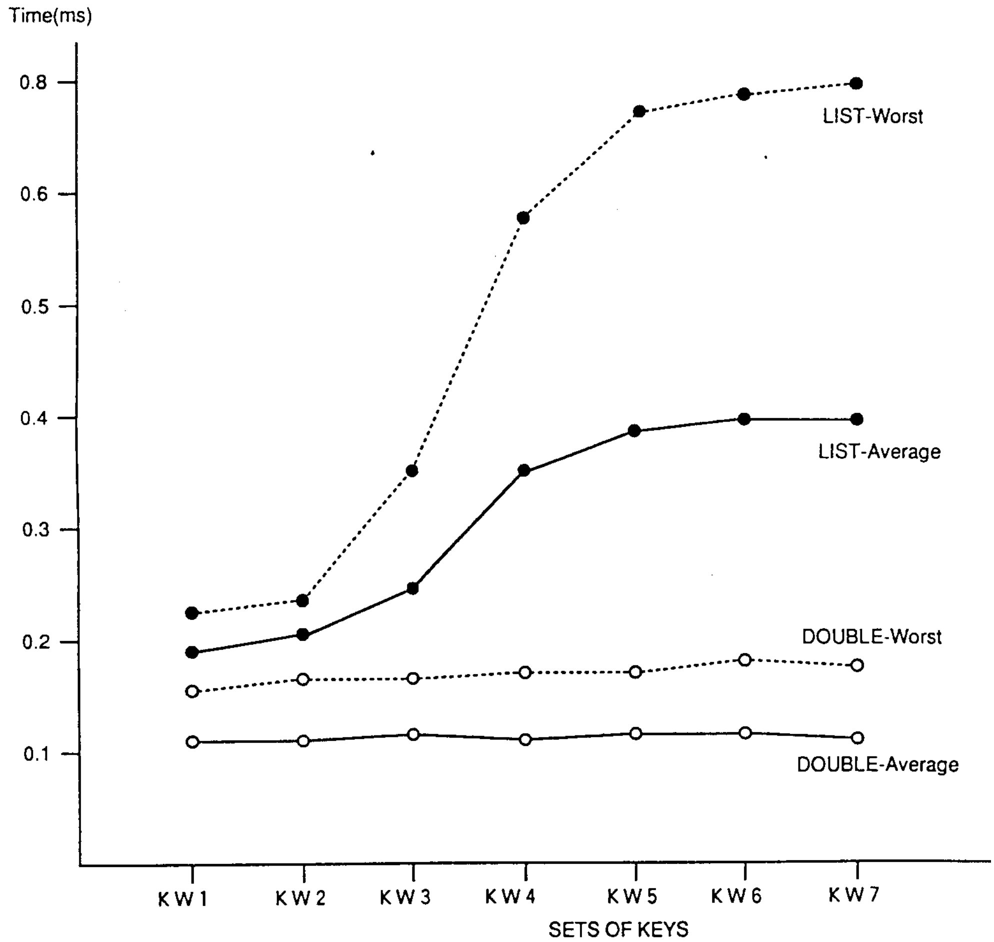


Figure 11. The results of retrieval time

CONCLUSION

A double-array structure representing a trie structure has been presented. For a variety of keys, the space ‘and time efficiencies of the double-array have been evaluated by comparing the list form. The double-array can build a fast and compact dictionary, but the insertion is slower than the list form when many keys are stored into the double-array. In the actual implementation for a large set of keys, the double-array should be divided into appropriate blocks and stored into auxiliary memory. This division is useful for application software limited to a work area in main memory, and enables the realization of fast insertion because each divided double-array keeps a reasonable number of keys.

It is important that a dictionary for natural language processing be built incrementally, because it might be necessary 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.

The presented double-array has been used in about 40 sets of dynamic keys (i.e. dynamic command interpreters, a bibliographic search [3,4,5] Japanese and English morphological dictionaries for a machine translation system, [7] filtering of highly frequent English words [6] and a spelling checker [6] emitting correction candidates, etc.). The double-array can efficiently manipulate the longest applicable match based on the trie, 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; [5] the reduction of static sparse matrices [9,10] by using a row displacement; a finite state machine associated with a parsing table; and so on.

APPENDIX: CODE IN C LANGUAGE FOR RETRIEVAL, INSERTION, AND DELETION

The trie based on the double-array structure is implemented using about 300 lines of C on a Sun-Microsystems Sun/4, IBM6100RT-PC, and various personal computers.

Global variables

The following is constant values and global variables to be used.

```
#define  INSERT_MODE  1 /* Indicate the insertion mode */
#define  SEARCH_MODE 2 /* Indicate the search, or retrieval, mode */
#define  DELETE_MODE 3 / * Indicate the deletion mode */
#define  DUMP_MODE    4 /* Indicate the dump mode */
#define  END_MODE     5 /* Indicate the end of program */
#define  MIN_CODE     1 /* Minimum numerical code */
#define  MAX_CODE   255 /* Maximum numerical code */
#define  BC_INC      10 /* Increment of the double-array */
#define  TAIL_INC    10 /* Increment of TAIL */
#define  KEY_INC      5 /* Increment of the double-array */
#define  TEMP_INC     5 /* Increment of TAIL */
#define  TRUE        -1
#define  FALSE        0
#define  NILL        -1

FILE  *KEY FILE;       /* Key dictionary file */
char  *KEY;            /* Key */
char  *TAIL;           /* TAIL */
char  *TEMP ;          /* Buffer */
int   *BC;             / * BASE and CHECK*/
int   MODE ;           /* Flag indicating insertion, search,
                          deletion, dump and end */
int   BC_POS ;   /* Current maximum index of the double-array */
int   TAIL_POS; /* The current maximum index of TAIL */

int   BC_MAX ;   /* Maximum size of BASE and CHECK */
int   TAIL_MAX; /* Maximum size of TAIL */
int   KEY_MAX;  /* Maximum size of KEY */
int   TEMP_MAX; /* Maximum size of TEMP */
```

```
void BC_INSERT(), TAIL_INSERT(), SEPARATE() , WRITE_TAIL() ,
     W_BASE() W_CHECK();
char *MEM_STR() ;
```

A one-dimensional array BC represents the arrays BASE and CHECK, so the following functions BASE(n) and CHECK(n) enable the description of the double-array access. In the following context, the notations of BASE[*n*] and CHECK[*n*] are conventionally used too:

```
int BASE (n)    /* BASE[n] */
int n;

        if (n > BC_POS) return(0) ;
   else return (BC[2*n] ) ;
}
int CHECK (n)   /* CHECK[n] */
int n;

        if (n > BC_POS) return (0) ;
   else return (BC[2*n+1]) ;
}


/*W_BASE (n, node) and W_CHECK (n, node) store node into BASE [n] and CHECK [n]
respectively*/

void W _ BASE (n, node) /* BASE [n] <= node*/
int      n, node;

         while(n >= BC_MAX) REALLOC_BC() ;
         if(n > BC_POS) BC_POS = n;
         BC[2*n] = node;
}
void W_CHECK(n, node) /*CHECK[n] <= node */
int      n, node;

         while(n >= BC_MAX) REALLOC_BC() ;
         if(n > BC_POS) BC_POS = n;
         BC(2*n + 1] = node;
}

char   * MEM_STR(area_name, max, int)
char   *area_name;
int    *max, int;

       char     *area;

       *max= int;
       if( (area = (char *)malloc (sizeof(char) * *max)) == NULL)
       {
                printf("%s malloc error! !\n", area_name) ;
       }
       memset(area, sizeof(area), '\0');
       return(area) ;
}

void REALLOC_BC()
{
   int  i, pre_bc;
```

```
    pre_bc = BC_MAX;
    BC_MAX += BC_INC;
    if((BC = (int *)realloc(BC, sizeof(int) *2*BC_MAX) ) == NULL) {
              fprintf(stderr,  "BC reallot error! !");
              exit(-1);
     }
     for(i = 2*pret_bc: i < 2*BC_NAX; i++) BC[i] = 0;
     fprintf(stderr,  "*** BC realloc ***\n");
}


char   *REALLOC_STR(area_name, area, max, int)
char   *area_name, *area;
int    *max,  inc;
{
      int      i,pre_size;

      pre_size=*max;
      *max+=int;
      if( (area= (char*) realloc(area, sizeof(char) * *max))==NULL) {
              printf("%s reallot error! !\n",area_name);
            exit(-1) ;
      }
      for(i=pre_size; i<*max; i++) area[i]=='\0';
      fprintf(stderr, "***%s realloc ***\n", area name) ;
      return(area) ;
```

/* READ_TAIL() copies the requested single-string from TAIL*/

```
void READ_TAIL(p)
int  p;
{
   int     i=0;

   while(TAIL[p] != '#') TEMP[i++] = TAIL[P++];
   temp[i++l = '#'; TEMP[i] = '/0';
}
```

/* WRITE_TAIL() stores the single-string into the requested address p of TAIL*/

```
void WRITE_TAIL(temp, p)
char     *temp;
int       p;
{
    int    i = 0, tail index;

    tail_index = p;
    while((p + strlen(temp)) >= TAIL_MAX-1)
              TAIL = REALLOC_STR("TAIL", TAIL, &TAIL_INC) ;
    while(*(temp+i) != '\0')
              TAIL [tail_index++] = *(temp+i++);
    if( (p + i + 1) > TAIL_POS) TAIL_POS = p + i;
}
```

Selection of modes

```
main()
{
int c, i, count;
```

```
INITIALIZE() ;
for(;;)
{
       SELECTION() ;
       count = 1; i=0
       while( (c = getc(KEY_FILE) ) !=EOF) {
          if(c !='\n'){
                        while(i >= KEY_MAX-2) {
                          KEY = REALLOC_STR("KEY",KEY, &KEY_MAX, KEY_INC);
                          TEMP = REALLOC_STR("TEMP", TEMP, &TEMP_MAX, TEMP_INC);
                             {
                                                         KEY[i++] = c;
                                                         continue;
                                               }
                        KEY[i] = '\0'; i = 0;

                        switch(MODE) {
            case INSERT_MODE:
                if(SEARCH() == FALSE)
                printf("%d:%s is inserted\n", count++, KEY);
                else printf("%d:%s is already in your dictionary\n",
                            count++, KEY) ;
                break;
             case SEARCH_MODE:
                if(SEARCH() == TRUE)
                printf("%d:%s is searched\n", count++, KEY);
                else printf("%d:%s is mismatch\n", count++, KEY) ;
                break;
                case DELETE MODE:
                   if(SEARCH() == TRUE)
                   printf("%d:%s is deleted\n", count++, KEY);
                   else printf("%d:%s is not in your dictionary\n",
                              count++, KEY) ;
                   break;
                case DUMP_MODE: break;
                default: exit(0);
                     }/switch*/
                }/*while*/
        INFO(count) ;
        fclose(KEY_FILE) ;
     }/*for*/
  }
  void SELECTION()
  {
  char  key_name[30];
         printf(" 1. Insert   2. Search 3. Delete  4. Dump  5. End \n");
         scanf("%d%*c", &MODE) ;
         if(MODE == END_MODE) exit(0) ;
         if(MODE != DUMP_MODE) {
               printf("key_file = ");
               scanf("%s%*c", key_name) ;
               KEY_FILE = fopen(key_name, "r") ;
               if(KEY FILE == NULL){
                    printf("\nkey_dic can't open\n");
                    exit(0);
                  }
        }
  }
```

/* INITIALIZE() initializes the entries of BASE and CHECK by zero, and TAIL by '\0', but, particularly, BASE[l] is initialized by 1. Note that BASE[0], CHECK[0], CHECK[l] and TAIL[0] are not used in this implementation*/

```
void INITIALIZE()
{
int i;
      BC_MAX = BC_INC; BC_POS = 1; TAIL_POS = 1;
      if(BC = (int *)malloc (sizeof (int) *2*BC_MAX) ) == NULL) {
                          fprint(stderr,  "BC malloc error!!") ;
                          exit(-1);
            }
      memset(BC, sizeof(BC), 0);
      W_BASE(1,1); BC_POS = 1;
             TAIL = MEM STR("TAIL",  &TAIL_MAX,  TAIL_INC);
             TAIL_POS =-1; TAIL[0] = '#';
          TEMP  = MEM_STR("TEMP",  &TEMP_MAX, TEMP_INC);
             KEY = MEM_STR("KEY", &KEY_MAX, KEY_INC) ;
}
```

/* INFO() displays the sizes of the doubIe-array and TAIL, and all values of the double-array and TAIL are included if DUMP_MODE is selected.*/.

```
void INFO(count)
int count;
{
   int   it bc_empty = 0;

   for(i=0; i <= BC POS; ++i){
       if(BASE(i) == 0 && CHECK(i) ==0) bc_empty++;
      }
   if(MODE == DUMP_MODE) {
           printf("\n");
           printf("Index  |   BASE   | CHECK\n");
   for(i=0; i <= BC_POS; ++i)
      printf("%7d |%7d | %7d|\n", i, BASE(i), CHECK(i));
   for(i=0; i <= TAIL_POS; ++i) printf("%d%c|", i, TAIL[i]);
      print("\n");
    }

    printf("Total number of keys=%d\n",count-1);
    printf("BC_POS=%d\n",  BC_POS);
    printf("bc_empty+%d\n",  bc_empty);
    printf("TAIL_POS=%d\n",  TAIL_POS);
}
```

Search routine

Suppose that MODE is SEARCH_MODE. The following function SEARCH() continues the do-while loop manipulating traversal on the reduced trie until BASE(t) becomes negative at line (s-4), that is to say, t is a separate node number. Then, the remaining input string (KEY+ h+ 1) is compared with temp (the single-string STR[t]) computed by READTAIL() in order to determine whether KEY is registered or not. But only if *(KEY+h) is equal to '#', then TRUE is immediately returned without accessing TAIL because STR[t] is the empty, or the null string. If KEY is not registered in the dictionary, then line (s-1) detects the mismatch of KEY on the double-array, and line (s-6) detects the mismatch on TAIL. Then, BC_INSERT() and TAIL_INSERT() are invoked for these detections, respectively, so each FALSE of lines (s-3, s-11) indicates that KEY was registered and TRUE of line (s-9) indicates that KEY has been already registered. Suppose that KEY is in the dictionary and that MODE is DELETE_MODE. The deletion of KEY is performed at line (s-8) after confirming the existence of KEY at line (s-6).

Suppose that MODE is DELETE_MODE and that KEY has been registered in the dictionary. Then, lines (s-7, s-8) deletes the arc prior to a separate node t. The entries initialized by the deletion are available for the next insertion.

```
int SEARCH ()
      {
             unsigned char ch;
      int    h=-1,  s=1,  t;

          strcat (KEY, "#") ;
          do {
               ++h ;
               ch = KEY[h];
               t = BASE(s) + ch;
/*(s-1)*/      if (CHECK(t) != s) (
/*(s-2)*/              if(MODE == INSERT_MODE) BC_INSERT(s, KEY+h) ;
/*(s-3)*/              return(FALSE) ;
                   }
/*(s-4)*/      if(BASE(t) < 0) break;
               s = t ;
            }while(TRUE);
/*(s-5)*/   if(*(KEY+h) != '#') READ_TAIL ((-1) *BASE(t)) ;
/*(s-6)*/   if(*(KEY+h) == '#' || !strcmp(TEMP, (KEY+h+1))) {
/*(s-7)*/      if(MODE == DELETE_MODE) {
/*(s-8)*/                     W_BASE(t, 0);   W_CHECK(t, 0);
                      }
/*(s-9)*/      return(TRUE) ;
            }else{
/*(s-10)*/      if(MODE == INSERT_MODE && BASE(t) != 0)

TAIL INSERT(t, TEMP, KEY+h+1);
/*(s=11)*/      return(FALSE) ;
            }
      }
```

Insertion routine

As mentioned in the above search routine, there are two kinds of the functions BC_INSERT() and TAIL_INSERT() to be invoked in the INSERT_MODE. Consider first the following function BC_INSERT():

```
void BC_INSERT(s, b)
int      s;
char     *b;
{
 int     t;
 char   list_s[MAX_CODE-MIN_CODE+1],  list_t[MAX_CODE-MIN_CODE+1  ],
        *SET_LIST() ;

            t = BASE(s) + (unsigned char) *b;
 /*(b-1)*/  if(CHECK(t) != 0) {
 /*(b-2)*/          strcpy(list_s,  SET_LIST(s));
 /*(b-3)*/          strcpy(list t, SET_LIST (CHECK(t))) ;
 /*(b-4)*/          if(strlen(list_s)   +1<strlen(list_t))
 /*(b-5)*/              s = CHANGE_BC(s, s, list_s, *b) ;
 /*(b-6)*/           else  s = CHANGE_BC(s, CHECK(t), list_t, '\0');
                    }
 /*(b-7)*/ SEPARATE(s, b, TAIL_POS);
}
```

BC_INSERT(s, b) defines arc g(s, b[0]) on the double-array by invoking the function SEPARATE() at line (b-7) if CHECK(t) = 0 at line (b-1). SEPARATE(s, b, tail_pos) shown below defines arc g(s, b[0]) = t in CHECK[t] at line (e-1); stores position tail_pos of the single-string and minus sign indicating the separate node in BASE[t] at line (e-2); and stores the single-string into TAIL at line (e-3).

```
void SEPARATE (s, b, tail_pos)
char      *b ;
int       s, tail_pos;
{
  int      t;

              t = BASE(s) + (unsigned char) *b;   b++;
 /*(e-1)*/    W_CHECK (t , s ) ;
 /*(e-2)*/    W_BASE (t, (-1) *tail_pos) ;
 /*(e-3)*/    WRITE_TAIL (b, tail_pos) ;
}
```

If CHECK(t) has been already occupied at line (b-1) in BC_insert(), the BASE entries must be modified. In this situation, since an insertion of g(s, b[0]) is blocked by an arc of another node k = CHECK(t), an insertion algorithm solves the conflict by redefining BASE [s] or BASE[k]. In this selection of an s or k node, priority is given to the one with the fewest arcs, in order to reduce space and time. This operation uses the following functions SET_ LIST(), CHANGE_BC() and X_ CHECK() at lines from (b-2) to (b-6).

```
char *SET_LIST (s)
int    s;
{
  char      list [MAX_CODE-MIN_CODE+1 ] ;
  int       i , j = 0 , t ;

  for (i = MIN_CODE; i < MAX_CODE-1; i++) {
       t = BASE (s)+i;
       if (CHECK(t)  == s) list [j++] = (char)i;
      }
      list [j] = '\0';
       return (list) ;
}

CHANGE_BC (current, s, list, ch)
int    current, s;
char   *list, ch;
{
  int    i, k, old node, new_node, old_base;
  char   a_list [MAX_CODE-MIN_CODE ] ;

 /*(c-1)*/ old_base = BASE(s);
 /*(c-2)*/ if(ch != '\0'){
                   strcpy(a_list, list) ; i = strlen (a_list) ;
                   a_list [i] = ch;       a_list[i+1] = '\0';
                  }
 /* (c-3) */ W_BASE (s, X_CHECK(a list) ) ;
              i = 0 ;
             do {
```

```
/*(c-4)*/       old_node = old_base + (unsigned char) (*list) ;
/*(c-5)*/       new_node = BASE(s) + (unsigned char) (*list) ;
/*(c-6)*/       W_BASE(new_node, BASE(old_node));
/*(c-7)*/       W_CHECK(new_node, s) ;
/*(c-8)*/       if(BASE(old_node) > 0) {
                    k = BASE(old_node)+1;
/*(c-9)*/           while (k-BASE(old_node) <MAX_CODE-MIN_CODE||k<BC_POS) {
/*(c-10)*/                if(CHECK(k) == old_node) W_CHECK(k, new_node) ;
                          ++k ;
                      }
                  }
  /*(c-11)*/    if(current != s && old_node == current) current = new_node;
  /*(c-12)*/    W_BASE(old node, 0);  W_CHECK(old_node, 0) ;  list++;
            }while(*list != '\0');
/*(c-13)*/  return(current);
}
```

CHANGE_BC(current, s, list, ch) modifies BASE[s] by calling the following function X_CHECK(list) at line (c-3). X_CHECK() determines the minimum index such that base_pos> 1 and CHECK(base_pos + 'c')=0 for all entries 'c' in list.

```
int X_CHECK(list)
char *list;
{
  int      i, base_pos = 1, check_pos;
  unsigned char    sch;

        i = 0 ;
        do{
              ch = list[i++];
/*(x-1)*/     check_pos = base_pos + sch;
/*(x-2)*/     if (CHECK(check_pos) != 0) {
              base_pos++; i= 0;
              continue;

        }while(list[i] != '\0');
        return(base_pos) ;
}
```

Modifying BASE[s] in CHANGE_BC() involves the modification of the following arcs g(s, 'a') = old_node for ¢'a' in list and the arcs leaving old_node as follows.

1. For arcs leaving node s, the BASE and CHECK entries concerning old_node are replaced by using new_node at lines from (c-4) to (c-7).
2. For arcs leaving node old_node, the entries such that CHECK(k) = old_node are replaced by new_node at lines form (c-8) to (c-10).
3. As a special case of (2), replacing old_node by new_node involves the modification of current node s in BC_INSERT() when CHANGE_BC() has been invoked at line (b-6), so line (c-11) checks this situation and line (c-13) returns the modified current node.

Another function TAIL_INSERT() for the insertion is shown below. For the current node number s and for the remaining input string b, the function TAIL_INSERT(s, a, b) defines arc g(s, b[0] = t on the double-array and stores the single-string STR[t]=b[1]b[2]... in TAIL. The while-loop at lines from (t-3) to (t-6) appends a sequence of arcs for the longest prefix a[0]a[1]. .a[length-1] of strings a and b. Lines (t-8, t-9) defines arcs labelled a[length] and b[length] are stored in the double-array. The function SEPARATE() invoked at line (t-8, t-9) defines arcs such that g(s,a[length] = t and g(s,b[length]) = t ′ on the double-array, and stores the both remaining strings a+length+ 1 and b+length+ 1 for separate nodes t and t′ into TAIL.

```
void TAIL_INSERT (s, a, b)
char      *at *b;
int       s;
{
 char     list [3];
 unsigned char ch;
 int      i = 0, length = 0, t, old_tail_pos;

  /*(t-1)*/  old_tail_pos = (-1) *BASE(s) ;
  /*(t-2)*/  while (a [length] == b [length] ) length++;
  /*(t -3)*/  while (i < length) (
                    ch = a[i++];
                    list [0] = ch;    list [l] = '\ 0';
  /*(t -4)*/        W_BASE (s, X_CHECK (list, '\0'));
  /*(t-5)*/         t–= BASE(s) + ch;
  /*(t-6)*/         W_CHECK (t, S) ;
                    s = t ;
              }
 /*(t-7)*/    list [0= a [length];   list [1= b[length];   list [2] = ‘\ 0’;
 /*(t-8)*/   W_BASE (s X_CHECK(list, '\0';
 /*(t-9)*/   SEPARATE (s a+length, old_tail_pos) ;
 /*(t-10/   SEPARATE(S, b+length, TAIL_POS);
}
```

The number of redundant entries of the double-array grows for small sets of keys, but the number for large sets can keep an extremely small value. In order to build a more compact dictionary for small sets of keys, the remapping of characters on the basis of their frequency, statistically, becomes necessary. In this implementation, other kinds of characters (Katakana, Chinese, etc.) can be used. Nevertheless, it is better to treat a multi-byte character as one-byte by one-byte due to an offset based on a large numerical value makes the size of the double-array grow, that is to say, the double-array has many available, or redundant, entries.

REFERENCES

1. E. Fredkin, ‘Trie memory’, *Comm. ACM.*, **3,** (9), 490–500 (1960).
2. D. E. Knuth, *The Art of Computer Programming, vol. I, Fundamental Algorithms,* pp. 295–304; *vol. III, Sorting and Searching,* pp. 481–505, Addison-Wesley, Reading, Mass., 1973.
3. J. Aoe, Y. Yamamoto and R. Shimada, ‘A method for improving string pattern matching machines’, *IEEE Trans. Softw. Eng.,* **SE-10,** (l), 116–120 (1984).
4. John A. Dundas, III, ‘Implementing dynamic minimal-prefix tries ‘, ‘*Software—Practice and Experience,* **21,** (10), 1027–1040 (1991).
5. J. Aoe, Y. Yamamoto and R. Shimada, ‘An efficient implementation of static string pattern matching machines’, *Proc. First Int. Conf. on Supercomputing,* December 1985, pp. 491–498.
6. J. L. Peterson, *Computer Programs for Spelling Correction,* Lecture Notes in Comput. Sci., Springer-Verlag, New York, 1980.
7. J. Aoe and M. Fujikawa, ‘An efficient representation of hierarchical semantic primitives—an aid to machine translation systems’, *Proc. Second Int. Conf. on Supercomputing,* May 1987, pp. 361–370.
8. K. Maly, ‘Compressed tries’, *Commun. ACM.,* **19,** (7), 409–415 (1976).
9. R. E. Tarjan and A. C. Yao, ‘Storing a sparse table’, *Comm. ACM,* **22,** (11), 606–611 (1979).
10. J. Aoe, ‘A practical method for compressing sparse matrices with variant entries’, *Int. J. Comput. Math.,* **12,** (11), 97–111 (1982).