An Implementation Method of Trie Structure Using Xorshift

Kazuhiro Morita, Shunsuke Kanda, and Masao Fuketa

***Abstract*—Key retrieval is the most basic and important technology. The trie is one of the key retrieval methods that can be retrieved without depending on the number of registered keys. The development of implementation method of a trie is indispensable for use in application fields. The double array is an excellent implementation method of a trie with high speed retrieval and compactness. However, there is room for the improvement in comparison with the implementation method given priority to memory efficiency because the size for two integer types per one node is necessary. This paper presents a novel implementation method for a trie by using the idea of the randomized algorithm. From experimental observations, it was shown that the proposal method promotes efficiency of a trade-off of memory size and the speed well.**

***Index Terms*—Trie, double array, randomized algorithm, pseudorandom number generator.**

I. Introduction

Key retrieval is the most basic and important technology. Because key retrieval is daily used by all scenes with the spread of the Internet, more efficient technological development is always demanded. The trie [1] is one of key retrieval methods, and has the feature that can be retrieved without depending on the number of registered keys. Therefore, a trie is used in various application fields such as natural language processing [2], a lexical analyzer of a compiler [3], a bibliographic search [4], and so on.

The development of implementation methods of a trie that is a tree structure is indispensable for use in application fields. The double array [5] is an excellent implementation method of a trie with high speed retrieval and compactness. Even though a double array is a memory size almost proportional to the number of nodes of tries, the retrieval speed that depends only on the length of a key is achieved. However, there is room for the improvement in comparison with the implementation method given priority to memory efficiency because the size for two integer types per one node is necessary. Therefore, a lot of techniques for improving the storage efficiency of a double array are proposed [6], [7]. Still, further improvements are difficult because base positions to next nodes storing in order to traverse a trie tree become a bottleneck.

This paper proposes an implementation method for a trie by the new approach. The proposal method decides the next node using the idea of the randomized algorithm. Accurately, the next node is decided by using the techniques of a pseudorandom number generator. Therefore, a storage like storing base positions in a double array is unnecessary. It only has to store information that confirms the definition of nodes, and the improvement of the space efficiency can be expected. From experimental observations, the effectiveness of the proposal method is shown.

Manuscript received October 14, 2016; revised December 29, 2016. This work was supported in part by the Kayamori Foundation of Informational Science Advancement.

Kazuhiro Morita and Masao Fuketa are with the Department of Computer Science, Graduate School of Science and Technology, Tokushima University, Tokushima, Japan (e-mail: kam@is.tokushima-u.ac.jp, fuketa@is.tokushima-u.ac.jp).

Shunsuke Kanda is with the Department of Information Science and Intelligent Systems, Graduate School of Advanced Technology and Science, Tokushima University, Tokushima, Japan (e-mail: shnsk.knd@gmail.com).

II. Tries and Implementation

A. Tries

The trie is a tree structure with labels in edges. To enable key retrieval by traversing the tree from the root along labels, a trie is used in various applications. Fig. 1 shows the trie to key set K={"be," "boy," "by," "bye," "ebb," "eye," "obey"}. '#' in the figure is a special endmarker to distinguish the key. For example, the key "bye#" is able to be retrieved by traversing the node 1, 3, 8, 13 and 21 sequentially. Therefore, the high-speed retrieval without depending on the number of registered keys is possible. For the following explanations, the function g ($g(s, a)=t$) indicates that the edge with label a is defined from the node s to t. When the edge is not defined, it becomes $g(s, a)=fail$. To retrieve the key "bye#," it is only to confirm g(1, 'b')=3, g(3, 'y')=8, g(8, 'e')=13 and g(13,'#')=21 sequentially.

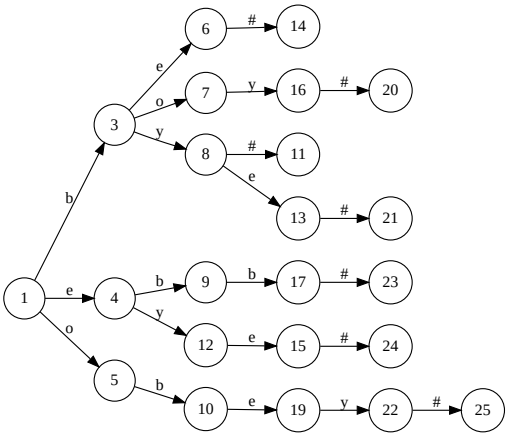


Fig. 1. Example of a trie for key set K.

B. Double Array

As for implementing the trie, various techniques are proposed from respect of the retrieval speed and memory

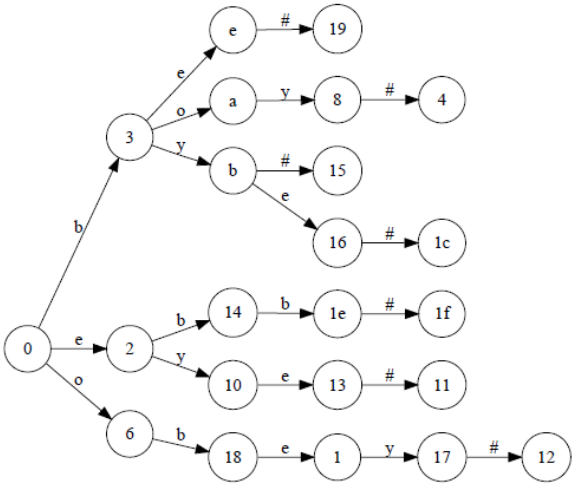
doi: 10.18178/ijfcc.2016.5.6.476

usage [5], [8]. Those techniques are essentially an implementation method of the function *g*.

	#	b	e	o	y
CODE	1	2	3	4	5

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
BASE	1		3	7	8	13	11	10	15	16	-1	12	20	-1	23	19	22		17	-1	-1	24	-1	-1	-1
CHECK	-		1	1	1	3	3	3	4	5	8	4	8	6	12	7	9		10	16	13	19	17	15	22

Fig. 2. Example of a double array for Fig. 1.



	#	b	e	o	y
CODE	1	2	3	4	5

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	10	11	12	13	14	15	16	17	18	19	1a	1b	1c	1d	1e	1f
PARITY	-	3	d	6	d		f		3		8	3			a		f	7	e	5	d	0	6	5	d	3			c		8	6
COLLISION	-	3	1	1	3		1		1		1	1			1		1	4	2	1	1	1	1	4	1	2			2		1	5

Fig. 3. Example of a trie and a xorshift array for key set *K*.

The double array is an especially excellent technique as the implementation method of the trie, and a lot of improvement techniques of a double array are proposed [9], [10]. A double array is composed of two array BASE and CHECK with the index corresponding to nodes of the trie. And the function $g(s,a)=t$ is implemented as follows:

[function $g(s, a)$]
Input: s is the parent node. a is a label.
Output: t is the next (child) node or *fail*.
begin
  t := *BASE*[s] + *CODE*(a);
  if *CHECK*[t]=s then return t;
  otherwise return *fail*;
end;

where *CODE* is numeral value for labels.

That is, the next node t is calculated as the distance from a base position to the label (the sum of the CODE for the label a and the BASE position for the node s), and it is CHECKed whether the transition s to t is defined.

Fig. 2 shows the double array representation of the trie of Fig. 1. Blank elements in the figure are unused elements. When the key "bye#" is retrieved, g(1, 'b')=3 is confirmed by *BASE*[1]+*CODE*('b') = 1+2 = 3 and *CHECK*[3] = 1. In the same manner, g(3, 'y')=8, g(8, 'e')=13 and g(13, '#')=21 are confirmed as follows:

BASE[3]+*CODE*('y') = 3+5 = 8, *CHECK*[8] = 3;
BASE[8]+*CODE*('e') = 10+3 = 13, *CHECK*[13] = 8;
BASE[13]+*CODE*('#') = 20+1 = 21, *CHECK*[21] = 13;

Because the memory size becomes wasteful when unused elements increase, it is important how to decide base positions to the construction of a double array.

III. Novel Implementation Method Using Xorshift

This paper proposes the implementation method of a trie by a new approach. The array corresponding to nodes is basically prepared as well as a double array. The idea of the randomized algorithm is used to calculate the next node.

A. Xorshift

The Xorshift is one of pseudorandom number generators proposed by Marsaglia [11]. An extremely uniform random numbers are generated only with the combination of simple calculations such as the bitwise xor (exclusive or) and the bit shift operation at high speed. Moreover, the same random number sequence is generated from the same seed as well as The Mersenne Twister [12]. That is, there is one of generators based on linear recurrences.

One concrete Xorshift operation is a xor of a bit vector x and the shifted copy of x either to left or to right. This operation is defined as the following function *xos*.

[function *xos*(x, b)]
Input: x is a bit vector. b is shift bits.

```
begin
    if b >= 0 then return x ^ (x << b);
    if b < 0 then return x ^ (x >> -b);
end;
```

In order to generate the next random number, Xorshift calculates *xos* three times with changing the value of *b*. Let $B=(b_1, b_2, b_3)$ be the triplet as the combination of these *b*. The function to enhance the function *xos* to *B* is defined as $XOS(x,B) = xos(\, xos(\, xos(\, x, b_1\,), b_2\,), b_3\,)$. Therefore, the best combination of *b* is important.

B. Xorshift Array

A basic strategy of the proposal technique is to decide the next node *t* at random in function *g*. However, it should be only *t* against the node *s* and label *a*. Then, Xorshift is used to implement the function *g*.

When implementing, the array PARITY corresponding to nodes of a trie is prepared. The implementation of the function *g* is shown below:

```
[function g(s, a)]
Input: s is the parent node. a is a label.
Output: t is the next (child) node or fail.
begin
    x := (s << W) | CODE(a);
    y := XOS( x, B );
    t := y >> W;
    p := y & (2^w-1);
    if PARITY[t]=p then return t;
    otherwise return fail;
end;
```

where *W* is the minimum number of bits in which all labels are expressible.

That is, the bit vector *x* that is combined the node *s* with the numeral value of the label *a* is given. The bit vector *x* is regarded as a seed, and the next random value *y* is generated by Xorshift. The value *y* is divided into the next node *t* and the parity *p*. Finally, the definition of the transition is confirmed by comparing the value of *PARITY*[*t*] with the parity *p*.

Now, consider a problem that the collision occurs to *t*. Because *XOS* is the bijective function, the value *y* is uniquely decided to *x* if *B* is fixed. However, the next node *t* that is a part of the value *y* is not necessarily unique. Then, it deals with the collision by operating Xorshift again when the collision occurs. In other words, it means that the next random number is selected. Therefore, the array COLLISION to store what number random value is selected is prepared. The final version of the function *g* in the proposal method is implemented as follows:

```
[function g(s, a)]
Input: s is the parent node. a is a label.
Output: t is the next (child) node or fail.
begin
    c := 1;
    x := (s << W) | CODE(a);
    repeat
        y := XOS( x, B );
        t := y >> W;
        p := y & (2^w-1);
        if PARITY[t]=p and COLLISION[t]=c
        then return t;
        x := y;
        c := c+1;
    until c > C;
    return fail;
end;
```

where *C* shows the upper bound of the loop frequency.

The PARITY and the COLLISION frequency are examined at the confirmation of the next node. When it is not equal, the next random numbers are examined. It becomes a failure when not found even if this is repeated to the collision frequency upper bound.

The proposal method will be called the xorshift array. Fig. 3 shows the trie and the xorshift array representation for key set *K*. The node number is described by the hexadecimal number for easiness. Moreover, parameters are *W*=4, *C*=5 and *B*=(3,-5,1), respectively. For example, consider a key "bye#". *g*(0x0, 'b')=0x3 is confirmed by (0x0 << 4) | *CODE*('b') = 0x02, *XOS*(0x02, (3,-5,1)) = 0x36, *PARITY*[0x3]=0x6 and *COLLISION*[0x3]=1. In the same manner, *g*(0x3, 'y') = 0xb and *g*(0xb, 'e') = 0x16 are confirmed. For *g*(0x16, '#')=0x1c, by (0x16 << 4) | *CODE*('#') = 0x161, *XOS*(0x161, (3,-5,1)) = 0xbe, then the next node and parity become 0xb and 0xe respectively. However, because *PARITY*[0xb] is not equal to 0xe, the next random number is generated by *XOS*(0xbe, (3,-5,1)) =0x1cc. Finally, the next node *t*=0x1c is confirmed by *PARITY*[0x1c]=0xc and *COLLISION*[0x1c]=2.

The xorshift array is composed of two arrays named COLLISION and PARITY as well as a double array. However, because the xorshift array can construct the size of one element with 8 bits at most while a double array is roughly the 32-bit integer type, the efficiency improvement of the memory size can be expected.

IV. Experimental Obsevation

For experimental observations, Entry words of Wordnet, IPAdic, English and Japanese Wikipedia are used for the key set. The numbers of words are 147,306, 217,550, 1,518,205 and 11,519,354, respectively. The xorshift array is compared with a double array and a level-order unary degree sequence (LOUDS) [8], [13]. These methods were implemented with C++. The PC environment used to experiment is Quad-Core Intel Xeon 2 x 2.4 GHz. Table I shows parameters of the xorshift array. The maximum collision frequency *C* and the triplet *B* were decided to minimize *C* of each key set.

The experimental results to the retrieval speed and the memory size are shown in Table II and Table III respectively. As the implementation method of a trie, LOUDS is the most compact method, and about half size of the xorshift array. However, the retrieval speed is a weak point, LOUDS is about seven times slower than the xorshift array. In the comparison with a double array, the retrieval speed is about two times faster than the xorshift array. But the memory size is about 3 times, and the xorshift array is more compact. From this result, it can be said that the xorshift array promotes efficiency of a

trade-off of memory size and the speed well.

TABLE I: PARAMETERS OF THE XORSHIFT ARRAY

Parameters	WordNet	IPAdic	Japanese Wiki	English Wiki
W		8		
C	23	42	93	43
B	(26,-5,6)	(15,-16,12)	(5,-11,7)	(7,-30,1)

TABLE II: EXPERIMENTAL RESULTS FOR RETRIEVAL SPEED

	WordNet	IPAdic	Japanese Wiki	English Wiki
Retrieval speed (μs / key)				
LOUDS	2.171	2.045	7.571	11.065
Double Array	0.139	0.133	0.655	0.786
Xorshift Array	0.302	0.308	1.780	1.757
ratio for Speed				
to LOUDS	0.139	0.151	0.235	0.159
to Double Array	2.173	2.316	2.718	2.235

TABLE III: EXPERIMENTAL RESULTS FOR MEMORY SIZE

	WordNet	IPAdic	Japanese Wiki	English Wiki
Memory Size (byte)				
LOUDS	1,109,855	1,314,822	22,843,366	150,717,827
Double Array	7,526,800	9,203,384	141,119,280	948,367,360
Xorshift Array	2,244,616	2,244,616	35,913,736	287,309,832
ratio for Size				
to LOUDS	2.022	1.707	1.572	1.906
to Double Array	0.298	0.244	0.254	0.303

V. CONCLUSION

This paper proposed the novel implementation method of a trie by using the technique of Xorshift that was the pseudorandom numbers generator. The effectiveness of the proposal technique was confirmed by the experiment.

The future works is to discover an appropriate triplet *B* that decreases the collision frequency, and to reduce in an unused element.

REFERENCES

[1] E. Fredkin, "Trie memory," *Communications of the ACM*, vol. 3, no. 9, pp. 490–500, 1960.

[2] L. Yang, L. Xu, and Z. Shi, "An enhanced dynamic hash TRIE algorithm for lexicon search," *Enterprise Information Systems*, vol. 6, no. 4, pp. 419-432, 2012.

[3] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools* (2Nd Edition), Addison-Wesley Longman Publishing Co., Inc., 2006.

[4] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Communications of the ACM*, vol. 18, no. 6, pp. 333-340, 1975.

[5] J. Aoe, "An efficient digital search algorithm by using a double-array structure," *IEEE Transactions on Software Engineering*, vol. 15, no. 9, pp. 1066–1077, 1989.

[6] K. Morita, E.-S. Atlam, M. Fuketa, K. Tsuda, and J. Aoe, "Fast and compact updating algorithms of a double-array structure," *Information Sciences*, vol. 159, no. 1-2, pp. 53-67, 2004.

[7] S. Yata, M. Oono, K. Morita, M. Fuketa, and J. Aoe, "A compact static double-array keeping character codes," *Information Processing & Management*, vol. 43, no. 1, pp. 237-247, 2007.

[8] G. Jacobson, "Space-efficient static trees and graphs," *IEEE Symposium on Foundations of Computer Science*, 1989, pp.549-554.

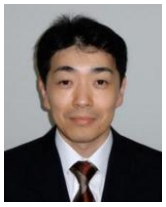
[9] M. Fuketa, H. Kitagawa, T. Ogawa, K. Morita, and J. Aoe, "Compression of double array structures for fixed length keywords," *Information Processing & Management*, vol. 50, no. 5, pp. 796-806, 2014.

[10] S. Kanda, M. Fuketa, K. Morita, and J. Aoe, "A compression method of double-array structures using linear functions," *Knowledge and Information Systems*, vol. 48, no.1, pp. 55-80, 2016.

[11] G. Marsaglia, "Xorshift RNGs," *Journal of Statistical Software*, vol. 8, no. 14, pp. 1-6, 2003.

[12] M. Matsumoto and T. Nishimura, "Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. Model. Comput. Simul.*, vol. 8, no. 1, pp. 3–30, 1998.

[13] O. Delpratt, N. Rahman, and R. Raman, "Engineering the louds succinct tree representation," in *Proc. WEA 2006*, 2006, pp. 134-145.



Kazuhiro Morita received the B.Sc., M.Sc. and Ph.D. degrees in information science and intelligent systems from Tokushima University, Japan, in 1995, 1997 and 2000, respectively. He had been a research assistant from 2000 to 2006 in information science and intelligent systems, Tokushima University, Japan. He is currently an associate professor in the Department of Information Sci. and Intelligent Systems, Tokushima University, Japan. His research interests are sentence retrieval from huge text databases, double-array structures and binary search tree.

Shunsuke Kanda received the B.Sc. and M.Sc. degrees in information science and intelligent systems from Tokushima University, Japan, in 2014 and 2016, respectively. He is currently a Ph.D. student at Tokushima University. He is a student member of the Information Processing Society in Japan. His research interests are data structures for string processing and indexing.

Masao Fuketa received the B.Sc., M.Sc. and Ph.D. degrees in information science and intelligent systems from Tokushima University, Japan, in 1993, 1995 and 1998, respectively. He had been a research assistant and an associate professor from 1998 to 2000 and from 2000 to 2015 in information science and intelligent systems, Tokushima University, Japan, respectively. He is currently a professor in the Department of Information Science and Intelligent Systems, Tokushima University, Japan. He is a member of the Information Processing Society in Japan and the Association for Natural Language Processing of Japan. His research interests are information retrieval and natural language processing.