Vol. 44 No. 5 情報処理学会論文誌 May 2003

ダブル配列におけるキー削除の効率化手法

大野 将樹[†] 森田 和宏[†]
泓田 正雄[†] 青江 順一[†]

トライ法は自然言語処理システムの辞書を中心として広く用いられているキー検索技法であり，トライを実現するデータ構造に検索の高速性と記憶量のコンパクト性をあわせ持つダブル配列構造がある．ダブル配列構造の欠点は，キーの削除によって生じる未使用要素により空間効率が低下する点である．これに対し森田らはダブル配列を詰め直すことにより未使用要素を除去するキー削除法を提案した．しかし，この手法はすべての未使用要素を除去できないため高い空間効率を維持できず，また削除コストが未使用要素数に依存するので，削除を連続するほど削除速度が低下するという問題がある．本論文では，トライの節のうち兄弟を持たない節が多くの割合を占めること，また，これらの節の遷移は容易に変更できるという特徴を利用し，削除を連続した場合でも空間使用率と削除速度を低下させない効率的なキー削除法を提案する．EDR 日英単語辞書，WordNet 英単語辞書，日本の郵便番号リスト，各 5 万件に対する実験より，提案法は削除を連続した場合でもきわめて高い空間使用率を維持することが，また，森田らの削除法より約 50〜200 倍高速に削除できることが実証された．

An Efficient Key Deletion Method for a Double-array

Masaki Oono,[†] Kazuhiro Morita,[†] Masao Fuketa[†]
and Jun-ichi Aoe[†]

A trie is a well known method for various dictionaries, such as spelling check and morphological analysis. A double-array structure is an efficient data structure combining fast access of a matrix form with compactness of a list form. The drawback of the double-array is that the space efficiency becomes worse by empty elements produced in key deletion. Therefore, Morita presented a key deletion method eliminating empty elements. However, the space efficiency of this method is low for high frequent deletion. Further, the deletion takes a lot of time because the cost depends on the number of empty elements. In this paper, a fast and compact deletion method is presented by using a property of nodes having no brothers. From simulation results for 50,000 keys, it turned out that the presented method is faster 50 to 200 times than Morita's method and keeps high space efficiency.

1. はじめに

近年，計算機の処理能力の向上および記憶装置の大容量化にともない，従来では考えられないような大量のデータを扱う必要性が急速に高まっている．このような環境において，高速かつコンパクトな検索技術の重要性が増している．

キーの表記文字を分岐条件とする木構造によりキー集合を表現するトライ法[1),2)]は，検索失敗位置の特定が容易であること，最左部分列の検出が容易であることなどの理由から，スペルチェッカ[3)]，形態素解析[4)]などの辞書を中心として広く用いられている．

トライを実現するデータ構造に，検索の高速性と記憶量のコンパクト性をあわせ持つダブル配列構造[5),6)]がある．ダブル配列構造の欠点は，キー削除によって生じる未使用要素を除去する手段が与えられていないため，削除キー数に比例して空間効率が低下する点である．この欠点に対し，森田らは，ダブル配列を詰め直すことにより未使用要素を除去するキー削除法を提案した[7)]．森田らの削除法は従来法よりも高い空間効率を実現するものの，すべての未使用要素を除去できないので，圧縮効率は不十分である．また，削除コストが未使用要素数に依存するため，削除が連続すると削除速度が大きく低下するという欠点がある．

本論文では，トライの節のうち兄弟を持たない節が多くの割合を占めること，これらの節の遷移は容易に変更できるという特徴に着目し，削除を連続した場合でも，空間使用率と削除速度を低下させない効率的なキー削除法を提案する．EDR 日英単語辞書，Word-

† 徳島大学工学部
Faculty of Engineering, Tokushima University

Net 英単語辞書，日本の郵便番号リスト，各 5 万件に対する実験より，削除を連続した場合でもきわめて高い空間使用率を維持することが，また，森田らの削除法より約 50〜200 倍高速になることが分かった．

以下，2 章でダブル配列の概要を説明する．3 章では森田らの削除法とその問題点を述べ，4 章で兄弟を持たない節を利用した効率的な削除法を提案する．5 章では提案法の理論的評価と実験による評価を行う．6 章では本論文のまとめと今後の課題についてふれる．

2. ダブル配列

トライの節（node）s から節 t（s, t は節番号）へ文字 a による遷移が定義されていることを $g(s,a)=t$ と表し，遷移が定義されていないことを $g(s,a)=fail$ と表す．ダブル配列は，2 つの配列 BASE，CHECK を使用し，$g(s,a)=t$ なる遷移を次式で定義する．

$$t=\mathrm{BASE}[s]+\mathrm{N}(a);\quad \mathrm{CHECK}[t]=s; \qquad (1)$$

このように，ダブル配列法は，節 s から文字 a による遷移先 t を BASE[s] と遷移文字 a の内部表現値 N(a) の和によって計算し，t が s からの遷移であることを，CHECK[t] に s を格納することで定義する．ダブル配列による節の遷移確認は式 (1) の計算量 $O(1)$ となり，きわめて高速な検索が実現できる．

BASE[r] $=0$ かつ CHECK[r] $=0$ ならば BASE[r] と CHECK[r] は未使用である．BASE[r] $\neq 0$ ならば BASE[r] と CHECK[r] はトライの定義に使用されている．ただし，節 r がトライの葉（leaf）のとき BASE[r] <0 とすることで内部節と区別される．これにより，検索の成功を判別できる．

以後，BASE[r]，CHECK[r] をまとめて要素 r と略記する．また，ダブル配列のインデックスとトライの節番号は 1 対 1 に対応するので，両者を区別せずに説明する．

[例 1] キー集合 $K=\{$“*babe#*”, “*bad#*”, “*badge#*”, “*be#*”$\}$ に対するトライとダブル配列を**図 1** と**図 2** に示す．図 2 は Aoe のダブル配列構築法[6] に基づいて構成されている．文字 ‘#’ はトライの葉とキーを 1 対 1 に対応させるための終端文字である．遷移文字の内部表現値は，終端文字 ‘#’ を 1，文字 ‘*a*’〜‘*z*’ を 2〜27 とする．$g(1,\text{‘}b\text{’})=4$ は，BASE[1] + N(‘*b*’) $=1+3=4$，CHECK[4] $=1$ より，式 (1) を満たすので，遷移が定義されていることが分かる．同様に，$g(1,\text{‘}a\text{’})=fail$ は，BASE[1] + N(‘*a*’) $=1+2=3$，CHECK[3] $=15\neq 1$ により確認できる．要素 13 は BASE[13] = CHECK[13] $=0$ より未使用である．また，葉 2 は BASE[2] $=-1<0$ となっている．（例終）

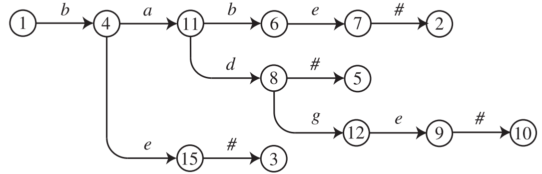

図 1 キー集合 K に対するトライ
Fig. 1 A trie structure for key set K.

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

図 2 キー集合 K に対するダブル配列
Fig. 2 A double-array structure for key set K.

```
Retrieval Algorithm
input:   Double-array D(K) for key set K,
         and retrieving key X=a1a2…anan+1, an+1=#.
output:  If X∈K, then output is TRUE, otherwise FALSE.
method:
begin
      s:=1; h:=0;
      repeat
        h:=h+1;
(R1)    t:=BASE[s]+N(ah);
(R2)    if (t>MAX)or(CHECK[t]≠s)
(R3)      then return(FALSE) else s:=t;
(R4)  until BASE[s]<0;
(R5)  return(TRUE);
end;
```

図 3 ダブル配列による検索アルゴリズム
Fig. 3 A retrieval algorithm by using double-array.

ダブル配列による検索アルゴリズムを**図 3** に示す．なお，MAX はダブル配列の最大インデックスを保持する大域変数である．行 R1 で節 s の遷移先 t を計算し，行 R2 で $g(s,a_h)=t$ なる遷移が定義されているか否かを確認する．t が MAX を超えるかあるいは遷移が未定義の場合，検索失敗なので FALSE を返す．遷移が確認できた場合，t を s にセットする．BASE[s] <0 ならば s は葉となり検索は成功するので，行 R5 で TRUE を返す．

[例 2] 図 2 のダブル配列において，キー “*be#*” を検索する例を示す．まず，行 R1 で節 $s=1$ の遷移先 $t=$ BASE[1] + N(‘*b*’) $=4$ を得る．行 R2 では $4<MAX=15$ かつ CHECK[4] $=1=s$ より $g(1,\text{‘}b\text{’})=4$ なる遷移が確認される．行 R3 では次の節をたどるため s に 4 をセットする．行 R4 では BASE[4] $=9>0$ より節 4 は葉ではないので repeat-until 文を繰り返す．以下同様に $g(4,\text{‘}e\text{’})=15$, $g(15,\text{‘}\#\text{’})=3$ が確認され，BASE[3] $=-4<0$ より節 3 はトライの葉となり検索が成功する．（例終）

```
procedure DELETE(s);
begin
      do
(D1)    if s=1 then return;
(D2)    parent:=CHECK[s];
(D3)    BASE[s]:=0; W_CHECK(s,0);
(D4)    s:=parent; d:=DEGREE(s);
(D5) while d=0;
(D6) COMPRESS;
end;
```

図 4 手続き DELETE
Fig. 4 Procedure DELETE.

```
procedure COMPRESS;
begin
(C1) cmp:=CHECK[MAX];
(C2) LABEL:=GET_LABEL(cmp);
(C3) new_base:=X_CHECK(cmp,LABEL);
(C4) if new_base≠FALSE then
(C5)   MODIFY(cmp,new_base,LABEL);
end;
```

図 5 手続き COMPRESS
Fig. 5 Procedure COMPRESS.

3. 削除アルゴリズムと問題点

3.1 削除アルゴリズム

ダブル配列の提案者である青江によるキー削除法（削除法 A とよぶ）は，検索アルゴリズムにより削除キーの存在を確認し，行 R5 で削除キーの終端文字の遷移先節 s（トライの葉 s）に対応する $\mathrm{BASE}[s] < 0$ なる要素 s を $\mathrm{BASE}[s] = \mathrm{CHECK}[s] = 0$ と未使用とし，トライから節 s を除去することで実現される．しかし，この方法は，削除キーを定義していた s 以外の節が不要な節☆として残るので，記憶量の無駄となる．また，大量にキーを削除した場合，未使用要素の増加により空間効率が低下する．

森田らの削除法（削除法 M とよぶ）は，不要な節を除去した後，未使用要素を詰め直すことで空間効率の悪化を抑える手法である．削除法 M は，検索アルゴリズムの行 R5 で，**図 4** の手続き DELETE(s) をよび出すことで実現される．

手続き DELETE で使用する関数と手続きを**図 5**, **6**, **7** に示す．また，削除法 M における要素の動きの概要を**図 8** に示す．

図 4 の手続き DELETE は，不要な節に対応する要素を未使用にすることで除去する．図 5 の手続き COMPRESS は，ダブル配列の最後方の要素 MAX と節 MAX の兄弟を定義する要素（これらを圧縮要素とよぶ）☆☆を前方の未使用要素に移動できるか否か

```
function X_CHECK(cmp,LABEL);
begin
(X1)  i:=E_HEAD;
(X2)  while i<MAX do
      begin
(X3)    j:=i-N(c1);
        {c1 is the smallest character in LABEL.}
(X4)    if j≤0 then
        begin
(X5)      i:=CHECK[i]; continue;
        end;
(X6)    if j≥BASE[cmp] then
(X7)      return(FALSE);
(X8)    flag:=TRUE;
(X9)    for each c in LABEL do
(X10)     if BASE[j+N(c)]≠0 then
(X11)       flag:=FALSE;
(X12)   if flag=TRUE then return(j);
(X13)   i:=CHECK[i];
      end;
(X14) return(FALSE);
end;
```

図 6 関数 X_CHECK
Fig. 6 Function X_CHECK.

```
procedure MODIFY(s,new_base,LABEL);
begin
(M1) old_base:=BASE[s];
(M2) BASE[s]:=new_base;
(M3) for each c in LABEL do
     begin
(M4)   t:=old_base+N(c);
(M5)   t':=new_base+N(c);
(M6)   W_CHECK(t',s); BASE[t']:=BASE[t];
(M7)   if BASE[t]>0 then
(M8)     for each d such that CHECK[d]=t
(M9)       do CHECK[d]:=t';
(M10)  BASE[t]=0; W_CHECK(t,0);
     end;
end;
```

図 7 手続き MODIFY
Fig. 7 Procedure MODIFY.

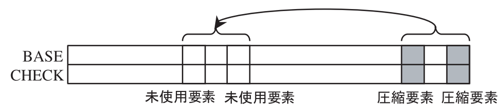

図 8 削除法 M における要素の移動の概要
Fig. 8 An illustration of element movement by method M.

☆ 図 1 から “$badge\#$” を削除する場合，節 10 のみがトライから除去されるが，節 9，12 が不要な節として残る．

☆☆ 図 2 では最後方要素 $MAX = 15$ と節 15 の兄弟に対応する要素 11 が圧縮要素である．

を確認し（行 C1〜C3），移動可能であれば移動する（行 C4，C5）．圧縮要素の移動後，ダブル配列のサイズを圧縮する．要素 MAX だけでなくその兄弟に対応する要素も移動する理由は，節 MAX とその兄弟が同じ遷移元 CHECK[MAX] からの遷移であり，共通の BASE 値によって遷移文字を定義しているためである．図 6 の関数 X_CHECK はすべての圧縮要素が移動できる新しい BASE[CHECK[MAX]] を決定し，図 7 の手続き MODIFY は圧縮要素を未使用要素へ移動する．X_CHECK は未使用要素のみをたどることで効率的に新しい BASE 値を決定する．そこで削除法 M では，昇順の未使用インデックス $r_1, r_2, \ldots, r_m$ に対する次のリンク（E リンクとよぶ）を定義している．

$$\mathrm{CHECK}[r_i] = r_{i+1} \quad (1 \leq i \leq m-1)$$
$$\mathrm{CHECK}[r_m] = MAX + 1$$

ただし，r_1 は大域変数 E_HEAD に格納され，ダブル配列中に未使用要素が存在しない場合には $E_HEAD = MAX + 1$ となる．要素 r が未使用であることは $\mathrm{BASE}[r] = 0$ により確認できる．図 2 に対し E リンクを適用した例（**図 9**）では，$E_HEAD = 13$ から未使用要素 13，14 をたどれる．

削除法 M で使用する他の関数と手続きを説明する．関数 DEGREE(s) は節 s から出る遷移数を返す．関数 GET_LABEL(s) は $g(s,a) \neq fail$ なる文字 a を要素とする集合を返す．手続き W_CHECK(t,s) は s を CHECK[t] に格納し，E リンクおよび E_HEAD を変更する．ただし，$s = 0$ かつ $t = MAX$ ならば，t でない最大の使用インデックス r を MAX に格納し，$r+1$ から t までの未使用要素を使用領域から解放する．

以下，削除法 M を例で説明する．

［例 3］ 図 9 に対し，キー “*badge#*” を削除した後のダブル配列を**図 10** に示す．図中の下線部は更新箇所である．DELETE により要素 10，9，12 が未使用となり，COMPRESS により圧縮要素 11，15 がそれぞれ未使用要素 9，13 へ移動する．後方の未使用要素

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
BASE	1	-1	-4	9	-2	1	1	4	9	-3	3	3	0	0	2
CHECK	1	7	15	1	8	11	6	11	12	9	4	8	14	16	4

図 9　E リンクを適用したダブル配列
Fig. 9　A double-array which E-linkage was applied.

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

図 10　削除法 M によりキー “*badge#*” 削除したダブル配列
Fig. 10　A double-array after deletion for “*badge#*” by method M.

14，15 は解放され，最終的にダブル配列のサイズは 13 に圧縮される．以下，この詳細を説明する．

まず，検索アルゴリズムにより $s = 10$ で検索が成功し，DELETE(10) が実行される（行 D1 は $s = 1$ の場合，不要な節がないので手続きを終了する）．行 D2 では節 10 の遷移元 CHECK[10] = 9 を $parent$ にセットし，行 D3 で要素 10 を未使用にする．行 D4 は $s = 9$，$d = \mathrm{DEGREE}(9) = 0$ より節 9 は遷移先がなく不要なので，行 D1 へ．以後同様に要素 9 を未使用とし，$s = 12$ において $parent = \mathrm{CHECK}[12] = 8$ を得，要素 12 を未使用にする．行 D4 は $s = 8$，$d = \mathrm{DEGREE}(8) = 1$ より節 8 は遷移先があるので，不要な節の除去を終了し，行 D6 の COMPRESS によりダブル配列を圧縮する．

COMPRESS の行 C1 では最後方要素に対応する節 $MAX = 15$ の遷移元 CHECK[15] = 4 を cmp にセットする．これにより，節 $cmp = 4$ の遷移先に対応する要素 11，15 が圧縮要素であることが分かる．行 C2 で GET_LABEL(4) により節 4 から出る遷移文字の集合 {‘a’,‘e’} を得る．行 C3 で X_CHECK(4, {‘a’,‘e’}) により圧縮要素を格納できる新しい BASE[4] を決定し，new_base にセットする．

X_CHECK は $c \in LABEL$ なるすべての c が $\mathrm{BASE}[j + \mathrm{N}(c)] = 0$ を満足する BASE[cmp] 以下（条件 X とよぶ）のインデックス j を返す．E リンクによりすべての未使用要素をたどっても条件 X を満足する j が存在しない場合，行 X14 で FALSE を返す．行 X1 では $i = E_HEAD = 9$ より，行 X3 で $j = 9 - \mathrm{N}(‘a’) = 7$ を得る．行 X4，X5 では $j > 0$ となるまで E リンクをたどるが，$j = 7 > 0$ なので処理をスキップする．行 X6，X7 では $j \geqq \mathrm{BASE}[cmp]$ のとき圧縮要素を前方に移動できないので FALSE を返すが，$j = 7 < \mathrm{BASE}[4] = 9$ より処理をスキップする．行 X9，X10 では $\mathrm{BASE}[7 + \mathrm{N}(‘a’)] = 0$，$\mathrm{BASE}[7{+}\mathrm{N}(‘e’)] = 0$ となり条件 X を満たすので，行 X12 で 7 を返し，COMPRESS の行 C3 の new_base に 7 がセットされる．行 C4 では $new_base = 7 \neq$ FALSE より圧縮要素 11，15 を未使用要素へ移動できるので，これらを MODIFY(4,7,{‘a’,‘e’}) により移動する．

MODIFY の行 M1 では BASE[4] = 9 を old_base に退避し，行 M2 は BASE[4] を $new_base = 7$ に変更する．行 M3 の for 文内では圧縮要素を移動する．まず，圧縮要素 11 について，行 M4 で節 4 の遷移先 $t = 9 + \mathrm{N}(‘a’) = 11$，行 M5 で新しい遷移先 $t’ = 7 + N(‘a’) = 9$ を得，行 M6 で W_CHECK(9,4)，

BASE[9] = BASE[11] = 3 とすることで $g(4, \text{‘}a\text{’}) = 9$ なる遷移を定義する．これにより，節 11 の遷移先の CHECK 値を 9 に変更する必要があるので，行 M8, M9 で CHECK[d] = 11 なる $d = \{6, 8\}$ に対して CHECK[6] = 9，CHECK[8] = 9 とする．行 M10 では BASE[11] = 0，W_CHECK(11,0) により要素 11 を未使用にする．圧縮要素 15 についても同様に処理し，BASE[13] = 2，CHECK[13] = 4，CHECK[3] = 13 となるが，行 M10 で最後方要素 15 が未使用となるので，W_CHECK により要素 14，15 が使用領域から解放され，ダブル配列の最大インデックスは 13 に圧縮される．　（例終）

3.2　森田らのキー削除法 M の問題点

削除法 M は，削除法 A よりも高い空間効率を実現できるが，次のような問題点がある．

［問題点 1］ COMPRESS において，すべての圧縮要素を移動できる未使用要素が存在しない場合，ダブル配列を圧縮できない．また，未使用要素の数が少ないほど圧縮要素は移動し難くなる．したがって，削除法 M は高い空間効率を保つことが困難となっている．

［問題点 2］ 圧縮要素の数 b よりも未使用要素数 m が多ければ，COMPRESS において $m-b$ 個の未使用要素が残存する．また，圧縮要素を格納していた *MAX* 以外の要素は未使用となり残る．したがって，頻繁な削除により未使用要素が蓄積し，空間効率が悪化する．

［例 4］ 図 9 からキー “*bad#*” を削除する場合，DELETE(5) によって要素 5 が未使用となり，この時点で要素 5，13，14 が未使用となる．次に COMPRESS で圧縮要素 11，15 を移動するが，これらを移動できる未使用要素がないので圧縮できず，空間効率が悪化する（問題点 1）．

また，図 9 からキー “*badge#*” を削除する場合，DELETE(10) によって要素 10，9，12 が未使用となり，COMPRESS を実行する時点での未使用要素は 9，10，12，13，14 の 5 つになる．COMPRESS により，圧縮要素 11，15 はそれぞれ未使用要素 9，13 に移動するが，要素 10，12 は未使用のまま残存する（14 は解放される）．さらに，要素 11 は未使用となり，空間効率が悪化する（問題点 2）．　（例終）

4．シングル節を利用した効率的削除法

4.1　提案法の概要

図 1 のトライの節 2 などのように兄弟を持たない節をシングル節とよび，兄弟を持つ節 15 などをマルチ節とよぶ．後の実験で示すが，シングル節はトライの節の多くの割合を占める（多数性）．また，$g(s, c) = r$ なる節 r がシングル節であるとき，要素 r（シングル要素とよぶ）はインデックス N(c) 以上の未使用要素に必ず移動できる（機動性）．

提案法は，シングル節の多数性と機動性を利用することにより，手続き COMPRESS を効率化する．削除法 M では，COMPRESS における圧縮要素の移動先候補は未使用要素のみであったが，提案法では数多く存在するシングル要素を候補に加えることで，圧縮要素を移動可能な未使用要素が存在しない場合でも，高い確率で圧縮できるようになる．さらに，圧縮要素の移動先となったシングル要素は容易に未使用要素へ移動できるので，未使用要素を効率的に使用することができる（**図 11**）（問題点 1 の解決）．

また，削除法 M では圧縮要素の移動と最後方要素 *MAX* の解放を 1 回のみ行っていたが，提案法では，この操作を COMPRESS 実行前の未使用要素数だけ繰り返す．これにより，すべての未使用要素を除去することができる（問題点 2 の解決）．

ある節 s がシングル節であることを確認するためには，DEGREE(CHECK[s]) により節 s の遷移元 CHECK[s] からの遷移数が 1 であるか否かを調べればよいが，CHECK[r] = CHECK[s] なるすべてのインデックス r をダブル配列中から探索する必要があり，多くの処理時間を要する．そこで，シングル節の確認を高速にするため，次を定義する．

［定義 1］ CHECK[|CHECK[s]|] > 0☆なる節 s はシングル節である．CHECK[|CHECK[s]|] < 0 ならば節 s はマルチ節である．　（定義終）

［例 5］ 図 9 に対し，定義 1 を満足させた例を**図 12** に示す．定義 1 によりインデックス 4，8，11 の CHECK 値が負数となる．たとえば，節 2 は CHECK[|CHECK[2]|] = 6 > 0 よりシングル節であ

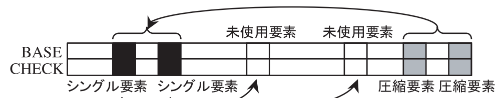

図 11　提案法における要素の移動の概要
Fig. 11　An illustration of element movement by new method.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
BASE	1	-1	-4	9	-2	1	1	4	9	-3	3	3	0	0	2
CHECK	1	7	15	-1	8	11	6	-11	12	9	-4	8	14	16	4

図 12　定義 1 を満足するダブル配列
Fig. 12　A double-array which satisfies definition 1.

☆ |CHECK[s]| は CHECK[s] の絶対値である．

```
procedure COMPRESS;
begin
(P1) e:=NUM_EMPTY;
(P2) for i:=1 to e do
     begin
(P3)   cmp:=|CHECK[MAX]|;
(P4)   if BASE[cmp]=1 then return;
(P5)   if CHECK[cmp]>0 then
(P6)     ret:=SINGLE(cmp)
(P7)   else ret:=MULTI(cmp);
(P8)   if (ret=FALSE)or(E_HEAD=MAX+1)
(P9)     then return;
     end;
end;
```

図 13 拡張した手続き COMPRESS
Fig. 13 Extended procedure COMPRESS by new method.

```
function SINGLE(cmp);
begin
(T1) LABEL:=GET_LABEL(cmp);
(T2) new_base:=X_CHECK(cmp,LABEL);
(T3) if new_base=FALSE then
(T4)   return(FALSE);
(T5) MODIFY(cmp,new_base,LABEL);
(T6) return(TRUE);
end;
```

図 14 関数 SINGLE
Fig. 14 Function SINGLE.

る．節 15 は CHECK[|CHECK[15]|] $= -1 < 0$ よりマルチ節であることが分かる． （例終）

DELETE による不要な節の除去にともない，マルチ節がシングル節になる場合がある．そこで，DELETE の行 D6 を次のように変更する．

手続き DELETE の行 D6 の変更：

(B1) **if** d=1 **then** CHECK[s]:=|CHECK[s]|;
(B2) COMPRESS; （変更終）

行 B1 は，節 s から出る遷移数 d が 1 である場合，s の遷移先はマルチ節からシングル節に変わっているので CHECK[s] の符号を正にする．また，キーの追加より節 s の遷移先がマルチ節となる場合，CHECK[s] の符号を負にする必要があるが，この処理は容易であるので詳細を省略する．

4.2 シングル節の利用によるキー削除アルゴリズム

提案法は，手続き COMPRESS を**図 13** のように変更することで得られる．COMPRESS で使用する関数を**図 14**，**15**，**16** に示す．

図 13 の COMPRESS は，節 MAX がシングル節の場合，要素 MAX を前方の未使用要素に移動する（行 P6）．節 MAX がマルチ節の場合，シングル節を利用して圧縮要素を移動する（行 P7）．この処理を COMPRESS 実行前の未使用要素数 NUM_EMPTY だけ繰り返し，すべての未使用要素を除去する（行 P1，P2）．圧縮要素を移動できなかったり，未使用要素がなくなった場合，圧縮を終了する（行 P8，P9）．

図 14 の関数 SINGLE はシングル要素を未使用要素へ移動する．図 15 の関数 MULTI は圧縮要素とその移動先となるシングル要素を図 11 のように移動する．SINGLE と MULTI は要素の移動に成功した場合 TRUE，失敗した場合 FALSE を返す．

図 16 の関数 EX_CHECK は圧縮要素を移動できる新しい BASE[|CHECK[MAX]|] の値 q を決定する関数であり，移動先候補を未使用要素あるいはシング

```
function MULTI(cmp);
begin
(U1)  old_max:=MAX;
(U2)  LABEL:=GET_LABEL(cmp);
(U3)  new_base:=EX_CHECK(cmp,LABEL);
(U4)  if new_base=FALSE then
(U5)    return(FALSE);
(U6)  for each c in LABEL do
      begin
(U7)     t:=new_base+N(c);
(U8)     if BASE[t]=0 then continue;
(U9)     u:=t-BASE[|CHECK[t]|];
(U10)    b:=MAX+1-u;
(U11)    MODIFY(|CHECK[t]|,b,{u'});
         {u' is a character such that N(u')=u.}
(U12)    if t=cmp then cmp:=b+u;
      end;
(U13) MODIFY(cmp,new_base,LABEL);
(U14) while MAX>old_max do
(U15)   SINGLE(|CHECK[MAX]|);
(U16) return(TRUE);
end;
```

図 15 関数 MULTI
Fig. 15 Function MULTI.

```
function EX_CHECK(cmp,LABEL);
begin
(E1)  if HEAD≥BASE[cmp] then HEAD:=1;
(E2)  q:=HEAD;
      repeat
(E3)    flag:=TRUE;
(E4)    for each c in LABEL do
(E5)      if (CHECK[|CHECK[q+N(c)]|]<0)
          or(BASE[q+N(c)]≠0)
(E6)        then flag:=FALSE;
(E7)    if flag=TRUE then
(E8)      begin HEAD:=q; return(q); end;
(E9)    q:=q+1;
(E10) until q=BASE[cmp];
(E11) HEAD:=1;
(E12) return(FALSE);
end;
```

図 16 関数 EX_CHECK
Fig. 16 Function EX_CHECK.

	1	2	3	4	5	6	7	8	9	10
BASE	1	-1	**3**	**1**	-2	**3**	**9**	4	**1**	**-4**
CHECK	1	**9**	**-4**	-1	8	**3**	**4**	**3**	**6**	**7**

図 17　提案法によりキー “*badge#*” を削除したダブル配列
Fig. 17　A double-array after deletion for “*badge#*” by new method.

ル要素とする点が X_CHECK と異なる．図中の変数 *HEAD* は q の探索開始インデックスを保持する大域変数であり初期値を 1 とする．なお，*HEAD* をつねに 1 とした場合，q は圧縮要素を移動できる最小の BASE 値となるが，MULTI において q を BASE 値とした要素に圧縮要素を移動するため，キー削除を繰り返すほどダブル配列の前方にマルチ節に対応する要素（マルチ要素とよぶ）が集中することになる．つまり，キー削除を繰り返すほどダブル配列の前方で q が存在し難くなり，q の探索時間が悪化してしまう．そこで，前回実行された EX_CHECK における q の探索終了インデックスを *HEAD* とすることでマルチ要素を分散させ，q の探索時間の悪化を抑える．

以下，提案法を例で説明する．

［例 6］ 図 12 に対し，キー “*badge#*” を削除した後のダブル配列を**図 17** に示す．図中の下線部は更新された箇所である．まず，DELETE により要素 10，9，12 が未使用となる．削除キーと 1 対 1 に対応するトライの葉が除去されるので，削除キーに対する検索は失敗するようになる．次に COMPRESS により圧縮要素 11，15 がそれぞれシングル要素 3，7 へ移動し，移動先となる要素 3 は未使用要素 10 へ，要素 7 は未使用要素 9 へ移動する．後方の未使用要素 11〜15 は解放され，ダブル配列のサイズは 10 に圧縮される．この時点で未使用要素はなくなるので圧縮を終了する．この流れを具体的に説明する．

まず，検索アルゴリズムにより $s = 10$ で検索が成功し，DELETE(10) によって要素 10，9，12 が未使用となるが，節 12 の除去により節 5 がシングル節となるので，$s = \mathrm{CHECK}[12] = 8$ より，行 B1 で $\mathrm{CHECK}[8] = |\mathrm{CHECK}[8]| = 11$ とし，行 B2 の COMPRESS でダブル配列を圧縮する．

拡張された COMPRESS は，行 P1 で要素 9，10，12，13，14 が未使用なので $e = \mathit{NUM_EMPTY} = 5$ とし，行 P2 の for 文によって圧縮要素の移動を $e = 5$ 回繰り返すことですべての未使用要素を除去する．行 P3 は最後方要素に対応する節 $\mathit{MAX} = 15$ の遷移元 $\mathrm{CHECK}[15] = 4$ を *cmp* にセットする．これにより，節 $\mathit{cmp} = 4$ の遷移先に対応する要素 11，15 が圧縮要素であることが分かる．行 P4 では，$\mathrm{BASE}[\mathit{cmp}] = 1$ の場合，圧縮要素を前方に移動できないので手続きを終了するが，$\mathrm{BASE}[4] = 9$ より処理を続ける．行 P5 は $\mathrm{CHECK}[4] = -1 < 0$ より節 15 はマルチ節であるので，行 P7 でシングル節の利用により圧縮要素を移動する関数 MULTI(4) を実行する．

MULTI の行 U1 では $\mathit{MAX} = 15$ を *old_max* に退避し，行 U2 で GET_LABEL(4) により節 4 から出る遷移文字の集合 {‘*a*’,‘*e*’} を得る．行 U3 では EX_CHECK(4,{‘*a*’,‘*e*’}) により圧縮要素 11，15 を移動可能な新しい BASE[4] を決定し，*new_base* にセットする．EX_CHECK は $c \in \mathit{LABEL}$ なるすべての c が $\mathrm{CHECK}[|\mathrm{CHECK}[q + \mathrm{N}(c)]|] > 0$ あるいは $\mathrm{BASE}[q + \mathrm{N}(c)] = 0$ を満足する $\mathrm{BASE}[\mathit{cmp}]$ 以下（条件 E とよぶ）のインデックス q を返す．行 E1 では，q の探索開始インデックス *HEAD* がすでに $\mathrm{BASE}[\mathit{cmp}]$ 以上のとき *HEAD* を 1 に初期化するが，$\mathit{HEAD} = 1 < \mathrm{BASE}[4] = 9$ より処理をスキップする．行 E2 は q に $\mathit{HEAD} = 1$ をセットする．行 E4〜E6 では，q が条件 E を満足するか否かを調査する．条件 E に適合しない場合，q をインクリメントし，$q = \mathrm{BASE}[\mathit{cmp}]$ となるまで探索する（行 E9，E10）．*HEAD* から $\mathrm{BASE}[\mathit{cmp}]$ までを探索しても条件 E を満足する q が存在しない場合，*HEAD* を 1 に初期化し，FALSE を返して関数を終了する（行 E11，E12）．この例の場合，文字 ‘*a*’，‘*e*’ に対し $\mathrm{CHECK}[|\mathrm{CHECK}[1 + \mathrm{N}(‘a’)]|] = \mathrm{CHECK}[15] = 4 > 0$，$\mathrm{CHECK}[|\mathrm{CHECK}[1 + \mathrm{N}(‘e’)]|] = \mathrm{CHECK}[6] = 11 > 0$ より条件 E を満たすので，行 E8 で次回の探索開始インデックス $\mathit{HEAD} = q = 1$ を設定し，q を返して関数を終了する．したがって，MULTI の行 U3 の *new_base* に 1 がセットされる．

MULTI の for 文内では，圧縮要素 11，15 の移動先となるシングル要素を $\mathit{old_max} = 15$ 以降に退避する．これは，圧縮要素の移動後に生じる未使用要素 11，15 を，退避したシングル要素の移動先として使用できるようにするための処理である．なお，ダブル配列のサイズの伸張および変数 *MAX* の更新は行 U11 の MODIFY 内で実行される W_CHECK が行うものとする．行 U7 では圧縮要素 11 の移動先となるシングル要素 $t = 1 + \mathrm{N}(‘a’) = 3$ を得る．行 U9 では $t - \mathrm{BASE}[|\mathrm{CHECK}[3]|] = 3 - 2 = 1 = \mathrm{N}(‘\#’)$ より 1 を u にセットし，行 U10 で節 3 の遷移元 $\mathrm{CHECK}[3] = 15$ の新しい BASE 値 $b = \mathit{MAX} + 1 - u = 15 + 1 - 1 = 15$ を得る．行 U11 は MODIFY(15,15,{‘#’}) によりシングル要素 3 を要素 16 へ退避する．これにより $\mathrm{BASE}[15] = 15$，

BASE[16] $= -4$，CHECK[16] $= 15$，要素 3 は未使用となる．行 U12 では，行 U11 の MODIFY によって t の節番号は $b+u$ に変更されるので，$t = cmp$ のとき cmp を $b+u$ に更新する．この例の場合，$t = 3 \neq cmp = 4$ より処理をスキップする．圧縮要素 15 の移動先となるシングル要素 $t = 1 + \mathrm{N}(`e')= 7$ についても同様の処理を行い，要素 7 が要素 17 に移動され，BASE[6] $= 11$，BASE[17] $= 1$，CHECK[17] $= 6$，CHECK[2] $= 17$，要素 7 は未使用となる．行 U13 では MODIFY(4,1,{'a','e'}) により圧縮要素 11，15 が要素 3，7 へ移動され，BASE[4] $= 1$，BASE[3] $= 3$，CHECK[3] $= -4$，CHECK[6] $= 3$，BASE[7] $= 15$，CHECK[7] $= 4$，CHECK[16] $= 7$，要素 11，15 は未使用となる．行 U14，U15 は $old_max = 15$ 以降に退避したシングル要素 16，17 を関数 SINGLE によって未使用要素へ移動する．

まず，SINGLE(6) により最後方要素 $MAX = 17$ を未使用要素へ移動する．行 T1 は GET_LABEL(6) により節 6 から節 17 への遷移文字 'e' を $LABEL$ にセットし，行 T2 で $new_base =$ X_CHECK(6, {'e'}) $= 3$ を得る．行 T5 は MODIFY(6,3,{'e'}) により最後方要素 17 を未使用要素 9 へ移動する．これにより，各要素は BASE[6] $= 3$，BASE[3] $= 1$，CHECK[3] $= 6$，CHECK[2] $= 9$ となり，最後方要素 17 は未使用となるので使用領域から解放され $MAX = 16$ となる．要素 16 に対しても同様の処理を行い，最後方要素 16 が未使用要素 10 へ移動され，BASE[7] $= 9$，BASE[10] $= -4$，CHECK[10] $= 7$ となり，要素 11〜16 が未使用となるので，ダブル配列のサイズは $MAX = 10$ に圧縮される☆．MULTI の行 U14 では $MAX = 10 < old_max = 15$ より退避したすべてのシングル要素を移動し終えたので，行 U16 で TRUE を返す．したがって，COMPRESS の行 P7 の ret に TRUE がセットされる．行 P8 では $E_HEAD = MAX + 1 = 11$ となりダブル配列中に未使用要素が存在しないので圧縮を終了する．（例終）

5. 評 価

本章では，ダブル配列の使用要素数を n，未使用要素数を m，遷移文字の最大数を e として説明する．また，理論的評価はすべて最悪の場合とする．

5.1 理論的評価

ダブル配列の領域計算量は $O(n+m)$ となる．削除法 M では，削除を連続すると未使用要素数 m が増加するが，提案法ではシングル節を利用することで m を非常に少なく保つことができる．この点については，次節の実験により実証する．

検索に対する時間計算量は，検索キーの長さ k に依存し，$O(k)$ となる．

削除法 M の時間計算量を得るために，まず，W_CHECK，X_CHECK，MODIFY の計算量を求める．W_CHECK の計算量は，未使用インデックスを E リンクに追加する操作に依存し，$O(m)$ となる[7]．X_CHECK の計算量は，行 X2 の while 文を m 回，行 X9 の for 文を e 回繰り返すので $O(me)$ となるが，定数 e を省略し $O(m)$ となる（以下，e に関するループを無視する）．MODIFY の計算量は，行 M6，M10 で W_CHECK を実行するので，$O(m+m) = O(m)$ となる．以上より，削除法 M の計算量は，DELETE の do-while 文で k 回 W_CHECK を実行し，COMPRESS の行 C3 で X_CHECK を実行し，行 C5 で MODIFY を実行するので，$O(km + m + m) = O(km + m)$ となる．

提案法の時間計算量を得るために，EX_CHECK，SINGLE，MULTI の計算量を求める．EX_CHECK の計算量は，repeat-until 文を $n + m$ 回繰り返すので $O(n + m)$ となる．SINGLE の計算量は，行 T2 で X_CHECK，行 T5 で MODIFY を実行するので $O(m)$ となる．MULTI の計算量は，行 U3 で EX_CHECK を実行し，行 U11 および行 U13 で MODIFY を実行し，行 U15 で SINGLE を実行するので $O(n + m)$ となる．以上より，提案法の計算量は，DELETE で k 回 W_CHECK を実行し，拡張された COMPRESS の行 P7 の MULTI を m 回繰り返すので，$O(km + m(n + m))$ となる．

提案法の時間計算量は削除法 M より大きくなるが，これは EX_CHECK において圧縮要素の移動先を決定するために最大 $n+m$（$= MAX$）回，ダブル配列の要素を走査することに起因する．ただし，$n+m$ 回走査することは，ダブル配列中のシングル節が極端に少ない場合にのみ起こる稀なケースである（たとえばトライの根 1 と葉以外のすべての節がマルチ節となるような場合）．自然言語のように文字の並びや長さが不規則なキー集合にはシングル節が数多く存在するので，実際の走査回数は $n + m$ よりはるかに少なくなる（提案法の時間計算量は $O(km + m)$ と見なせる）．また，削除法 M は削除を連続すると未使用要素が増加し削除速度が悪化するが（削除法 M の時間計算量 $O(km + m)$ の m が増大），提案法は m を少なく保

☆ 退避したシングル要素数 $\leq$ 圧縮要素数のため，行 U15 では関数 SINGLE の返り値の検証を省略している．

表 1 削除時間に対する実験結果
Table 1 The simulation results for deletion time.

	EDR 英辞書	EDR 日辞書	WordNet	郵便番号
初期状態				
キーの総数	50,000	50,000	50,000	50,000
節の平均遷移数	2.8	3.6	2.9	5.3
使用要素数 n	234,122	262,986	349,828	113,032
未使用要素数 m	8	1,666	59	4,516
シングル要素数	157,164	193,740	274,816	51,496
マルチ要素数	76,958	69,246	75,012	61,536
削除時間［秒］				
削除法 M	1,096	2,018	1,960	263
提案法	10	11	14	5
EXCH_SCAN	5.7	9.8	5.8	11.5

てるので高速に削除できる．以上の点については次の実験により実証する．

5.2 実験による評価

提案法は約 500 行の C++言語で実装されており，Intel Pentium III 1 GHz，Microsoft Windows2000 上で稼働している．実験に用いたキー集合は EDR [8] 英語形態素辞書，EDR 日本語形態素辞書，WordNet [9] の英語名詞辞書，日本の郵便番号であり，キー数はいずれも 5 万件である．特に，郵便番号はシングル要素が少なく，提案法にとって条件が悪くなるものとして採用した．

表 1 に各キー集合に対するダブル配列の初期状態と削除時間の実験結果を示す．**表 2** は空間効率に対する実験結果である．表 2 から，削除法 M はキーを削除するほど未使用要素数 m が増加し（括弧内は 1 万件削除間の最大未使用要素数），空間効率が大きく低下するのに対し，提案法は未使用要素がほとんど残留せず，非常に高い空間効率を維持していることが分かる．これは，ダブル配列の使用要素数 n のうちシングル要素が約 75%（郵便番号以外の自然言語によるキー集合の場合）と大きな割合を占めていることに起因する．また，郵便番号に対する実験結果から，シングル要素の割合が約 45%まで低下した状態でも提案法は有効であることが分かる．

表 1 から，提案法は削除法 M よりも高速にキーを削除できていることが分かる．これは，提案法は m を非常に少なく保てること，また EX_CHECK におけるダブル配列の平均走査回数（表中の *EXCH_SCAN*）がたかだか 5〜11 回程度であり，ダブル配列のサイズ（$n+m$）よりはるかに少ないことに起因する．

以上より，提案法はダブル配列の削除アルゴリズムとして有効であるといえる．

表 2 空間効率に対する実験結果
Table 2 The simulation results for space efficiency.

削除キー数	10,000	20,000	30,000	40,000	50,000
EDR 英辞書					
使用要素数 n	195,815	154,396	109,872	60,780	0
未使用要素数 m					
削除法 M	38,274	79,693	96,021	106,046	0
提案法（最大）	0（0）	0（0）	0（0）	0（1）	0（9）
シングル要素数	134,432	108,583	79,584	45,883	0
EDR 日辞書					
使用要素数 n	215,922	166,470	114,989	60,887	0
未使用要素数 m					
削除法 M	48,708	97,949	149,430	203,531	0
提案法（最大）	0（1）	0（2）	0（4）	0（1）	0（91）
シングル要素数	160,720	125,226	87,625	47,328	0
WordNet					
使用要素数 n	287,404	223,300	155,124	83,004	0
未使用要素数 m					
削除法 M	60,312	103,408	145,197	177,977	0
提案法（最大）	0（1）	0（0）	0（1）	0（1）	0（52）
シングル要素数	227,472	178,482	125,273	681,43	0
郵便番号					
使用要素数 n	92,879	72,617	51,857	29,479	0
未使用要素数 m					
削除法 M	24,614	34,853	20,803	144,74	0
提案法（最大）	0（0）	0（0）	0（2）	0（1）	0（54）
シングル要素数	41,716	32,311	23,492	14,646	0

6. おわりに

本論文では，トライの節のうちシングル節が多くの割合を占めること，また，シングル節の遷移は容易に変更できるという特徴に着目し，削除を連続した場合でも空間使用率と削除時間を悪化させない効率的なキー削除法を提案した．また，実験による評価により，本手法の有効性を実証した．本手法により，ダブル配列法の応用分野がさらに広がるものと思われる．

今後の課題は，提案法を実用システムに適用し，有効性を確認することである．

参 考 文 献

1) Fredkin, E.: Trie Memory, *Comm.ACM*, Vol.3, No.9, pp.490–500 (1960).
2) 青江順一：キー検索技法—トライとその応用，情報処理学会論文誌，Vol.34, No.2, pp.244–251 (1993).
3) Peterson, J.L.: Computer Programs for Spelling Correction, *Proc. 10th ACM Symposium on the Theory of Computing*, pp.59–65 (1980).
4) 田中穂積：自然言語解析の基礎，産業図書 (1989).
5) 青江順一：ダブル配列による高速ディジタル検索アルゴリズム，電子情報通信学会論文誌 D，No.9,

pp.1592–1600 (1988).
6) Aoe, J.: An Efficient Digital Search Algorithm by Using a Double-Array Structure, *IEEE Trans. Softw. Eng.*, Vol.15, No.9, pp.1066–1077 (1989).
7) 森田和宏，泓田正雄，大野将樹，青江順一：ダブル配列における動的更新の効率化アルゴリズム，情報処理学会論文誌，Vol.42, No.9, pp.2229–2238 (2001).
8) 日本電子化辞書研究所：EDR 電子化辞書 (1996).
9) Fellbaum, C.: *Wordnet: An Electronic Lexical Database*, MIT Press (1998).

(平成 14 年 4 月 11 日受付)
(平成 15 年 3 月 4 日採録)

大野　将樹（学生会員）

昭和 52 年生．平成 12 年徳島大学工学部知能情報工学科卒業．平成 14 年同大学院博士前期課程修了．現在同大学院博士後期課程在学中．情報検索，自然言語処理の研究に従事．

森田　和宏（正会員）

昭和 47 年生．平成 7 年徳島大学工学部知能情報工学科卒業．平成 9 年同大学院博士前期課程修了．平成 12 年同大学院博士後期課程修了．博士（工学）．同年徳島大学工学部知能情報工学科助手，現在に至る．情報検索，自然言語処理の研究に従事．

泓田　正雄（正会員）

昭和 46 年生．平成 5 年徳島大学工学部知能情報工学科卒業．平成 7 年同大学院博士前期課程修了．平成 10 年同大学院博士後期課程修了．博士（工学）．同年徳島大学工学部知能情報工学科助手．平成 12 年同大学工学部知能情報工学科講師，現在に至る．情報検索，自然言語処理の研究に従事．

青江　順一（正会員）

昭和 26 年生．昭和 49 年徳島大学工学部電子工学科卒業．昭和 51 年同大学院修士課程修了．同年同大学工学部情報工学科助手．現在同大学工学部知能情報工学科教授．この間コンパイラ生成系，パターンマッチングアルゴリズムの効率化の研究に従事．最近，自然言語処理，特に理解システムの開発に興味を持つ．著書「Computer Algorithms—Key Search Strategies」，「Computer Algorithms—String Matching Strategies」（IEEE CS press）．平成 4 年度情報処理学会 Best Author 賞受賞．工学博士．電子情報通信学会，人工知能学会，日本認知科学会，日本機械翻訳協会，IEEE，ACM，AAAI，ACL 各会員．