自 然 言 語 処 理 154－1
（２００３．３．６）

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併せも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削除法を提案した．しかし，この手法は全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も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悪化させない効率的なキー削除手法を提案する．10 万語の英単語に対する実験より，削除を連続した場合でも，ほぼ 100%の空間使用率を維持することが，また，森田らの削除手法より約 300 倍高速に削除できることが実証された．

キーワード：キー検索，トライ構造，ダブル配列構造，削除法

An Efficient Key Deletion Method for a Double-Array Structure

Masaki OONO,　Kazuhiro MORITA,　Masao FUKETA　and　Jun-ichi AOE

A trie is a well known method for various natural language dictionaries. A double-array structure is an efficient data structure combining fast access of a matrix form with compactness of a list form. Morita presented a key deletion method of a double-array by 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 100,000 keys, it turned out that the presented method is faster 300 times than Morita's method and keeps space efficiency ratio roughly 100%.

Keywords: Key search, Trie structure, Double-array structure, Deletion method

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伴い，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]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，スペルチェックチェッカ，形態素解析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併せもつダブル配列構造[2]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[3]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．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，全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も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手法を提案する．

10 万語の英単語に対する実験より，削除を連続した場合でも，ほぼ 100%の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 300 倍高速になることがわかった．

† 徳島大学工学部知能情報工学科，〒770 徳島市南常三島町２－１
Faculty of Engineering, University of Tokushima, Tokushima-shi, Japan
{oono,kam,fuketa,aoe}@is.tokushima-u.ac.jp

２．ダブル配列

トライの節（node）s から節 t へ文字 a による遷移が定義されていることを g(s,a)=t と表す．ダブル配列は，二つの配列 BASE，CHECK を使用し，遷移 g(s,a)=t を次式で定義する．

t=BASE[s]+a; CHECK[t]=s;　(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 a の内部表現値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，極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]≠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，検索の成功を判別することができる．

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一対一に対応するので，両者を区別せずに説明する．

[例 1] キー集合 K={babe#, bad#, badge#, be#}に対するトライの例を**図 1** に示す．文字'#'はトライの葉とキーを一対一に対応させるための終端記号である．図 1 のトライをダブル配列により構築した例を**図 2** に示す．なお，遷移文字の内部表現値は，終端記号'#'を 1，文字'a'～'z'を 2～27 とする．g(1,'b')=4 は，BASE[1]+'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わかる．要素 13 は BASE[13]=CHECK[13]=0 より未使用である．葉 2 は BASE[2]=−1<0 となっている．　（例終）

キー $X=a_1a_2\ldots a_na_{n+1}$, a_{n+1}=#に対するダブル配列の検索アルゴリズム SEARCH(X)を**図 3** に示す．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．

[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]+'b'=4 を得る．行 R2 では 4<MAX=15 かつ CHECK[4]=1=s より g(1,'b')=4 なる遷移が確認され，行 R3 で s=4 とする．行 R4 では BASE[4]=9>0 より節 4 は内部節であるので探索を続ける．以下同様に g(4,'e')=15，g(15,'#')=3 を確認し，BASE[3]=−4<0 より検索が成功する．　（例終）

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 BASE[s]<0 なる要素 s を BASE[s]=CHECK[s]=0 と未使用に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．森田らの削除アルゴリズム（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おさえる手法である．削除法 M は，検索アルゴリズムの行 R5 で削除キーの存在を確認した後，**図 4** の手続き DELETE(s)を呼び出すことで実現する．

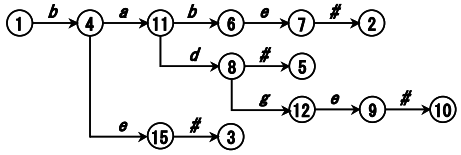

図 1　キー集合 K に対するトライ
Fig.1　A trie structure for key set K.

	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.

```
Function SEARCH(X);
  begin
    s:=1; h:=0;
    repeat
      h:=h+1;
(R1)  t:=BASE[s]+a_h;
(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.

DELETE で使用する関数と手続きを**図 5**，**図 6** に示す．また，削除法 M における要素の動きの概要を**図 7** 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を確認し（行 C1～C4），移動可能であれば移動する（行 C5）．圧縮要素の移動後，ダブル配列のサイズを圧縮する．削除法 M では，圧縮要素の移動先を効率的に決定するため，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．

$CHECK[r_i]=r_{(i+1)}$　$(1 \leqq i \leqq m-1)$，$CHECK[r_m]=MAX+1$

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 BASE[r]=0 により確認される．図 2 のダブル配列に対し E リンクを適用した例（**図 8**）では，E_HEAD=13 から未使用要素 13，14 を辿ることができる．

削除法 M で使用する他の関数と手続きを説明する．関数

```
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 DELTE.

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

図5 手続き COMPRESS

Fig.5 Procedure COMPRESS.

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

図6 手続き MODIFY

Fig.6 Procedure MODIFY.

DEGREE(s) は節 s から出る遷移数を返す．関数 GET_LABEL(s)は節 s から出る全ての遷移文字 a を要素とする集合を返す．X_CHECK(m_index,LABEL)は，c∈LABEL なる全ての c が BASE[j+c]=0 なる BASE[m_index]以下のインデックス j を返す関数である．条件を満足する j が存在しない場合，FALSE を返す．X_CHECK は E リンクを E_HEAD から順にたどることで j を探索する．W_CHECK(t,s)は s を CHECK[t]に格納し，E リンクを再構築する．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未使用領域とする．

以下，削除法 M を例で説明する．

[例 3] 図 8 のダブル配列に対し，キー"badge#"を削除した後のダブル配列を**図 9** に示す．図中の下線部は更新された箇所である． 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 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=DEGREE(9)=0 より節 9 は遷移先が無く不要であるので，行 D1 へ．以後同様に要素 9 を未使用とし，s=12 において parent=CHECK[12]=8 を得，要素 12 を未使用にする．行 D4 は s=8，d=DEGREE(8)=1 より節 8 は遷移先があるので，不要節の除去を終了し，行 D6 で COMPRESS を呼び出すことでダブル配列を圧縮する．

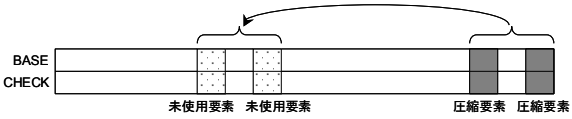

図7 削除法 M における要素の移動の概要

Fig.7 An illustration of element movement by method M.

	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

E_HEAD = 13

図8 E リンクを適用したダブル配列

Fig.8 A double-array that applied E-link.

	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9 キー"badge#"削除後のダブル配列

Fig.9 A double-array of after deletion for "badge#".

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 m_index にセットする．これにより，節 m_index=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わかる．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]の値 7 を得，new_base にセットする．行 C4 では new_base=7≠FALSE より圧縮要素 11，15 を前方の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，BASE[4]を new_base=7 に変更する．行 M3～M8 では，まず圧縮要素 11 を移動する．行 M3 で節 4 の遷移先 t=9+'a'=11 と新しい遷移先 t'=7+'a'=9 を計算し，行 M4 で W_CHECK(9,4)， BASE[9]=BASE[11]=3 とすることで g(4,'a')=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 M6，M7 で CHECK[d]=11 なる d={6,8} に対して CHECK[6]=9，CHECK[8]=9 とする．行 M8 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 M8 で最後方要素 15 が未使用と

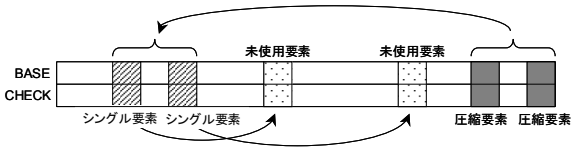

図 10 提案手法における要素の移動の概要

Fig.10 An illustraion of element movement by present 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 11 図 8 に対し定義 1 を満足させたダブル配列

Fig.11 A double-array that applied definition 1 for fig. 8.

なるので，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，全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，ダブル配列を圧縮することができない．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，頻繁なキー削除により未使用要素が蓄積し，空間効率が悪化する．

４．シングル節を利用した効率的削除法

４．１ 提案手法の概要

図 1 のトライの節 2 などのように兄弟をもたない節をシングル節とよび，兄弟をもつ節 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 c 以上の未使用要素に必ず移動できる（機動性）．

提案手法は，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 10）**（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，全ての未使用要素を除去できる（問題点 2 の解決）．

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］ 節 s がシングル節ならば CHECK[|CHECK[s]|]>0，マルチ節ならば CHECK[|CHECK[s]|]<0 が成り立つ．（定義終）

［例 4］ 図 8 のダブル配列に対し，定義 1 を満足させた例を **図 11** 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．例えば，節 2 は CHECK[|CHECK[2]|]=6>0 よりシングル節である．節 15 は CHECK[|CHECK[15]|]=−1<0 よりマルチ節であることがわかる． （例終）

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，COMPRESS を **図 12** 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 13～15** に示す．

図 12 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だけでよいので移動する（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，全ての未使用要素を除去する（行 P1，P2）． 図 13 の関数 SINGLE はシングル要素を未使用要素へ移動する．図 14 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 10 のように移動する．図 15 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，移動先を未使用要素あるいはシングル要素とする点が 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 を常に 1 とした場合，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り，探索時間が悪化してしまう．そこで，HEAD を前回実行された 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押さえる．

以下，提案法を例で説明する．

［例 5］ 図 11 のダブル配列に対し，キー”badge#”を削除した後のダブル配列を **図 16** に示す．図中の下線部は更新された箇所である．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，CHECK[8]=|CHECK[8]|=11 で正数とし，行 B2 の COMPRESS でダブル配列を圧縮する．

拡張された COMPRESS は，要素 9，10，12，13，14 が未使用なので行 P1 で e=NUM_EMPTY=5 とし，行 P3 は最後方要素に対応する節 MAX=15 の遷移元 CHECK[15]=4 を m_index にセットする．行 P5 は 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 では最大インデックス MAX=15 を old_pos

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

図12 拡張した手続き COMPRESS

Fig.12 Extended procedure COMPRESS by present method.

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

図13 関数 SINGLE

Fig.13 Function SINGLE.

に退避し，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 ∈ {‘a’,’e’} なる全ての c が CHECK[|CHECK[q+c]|]>0 あるいは BASE[q+c]=0 を満足する BASE[4]=9 以下（条件 E とよぶ）のインデックス q を返す．行 E2 は q に HEAD=1 をセットする．行 E4，E5 では，q が条件 E を満足するか否かを調査する．この例の場合，文字’a’に対し CHECK[|CHECK[1+’a’]|]=CHECK[15]=4>0，文字’e’に対し CHECK[|CHECK[1+’e’]|]=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 HEAD=q=1 を設定し，q=1 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 の行 U5～U11 では，圧縮要素 11，15 の移動先となるシングル要素を old_max=15 以降に退避する．行 U6 では圧縮要素 11 の移動先となるシングル要素 t=1+’a’=3 を得る．行 U8 では 3－BASE[|CHECK[3]|]=3－2=1 により節 3 への遷移文字’#’を u にセットし，行 U9 で節 3 の遷移元 CHECK[3]=15 の新しい BASE 値 b=MAX+1-u=15+1-1=15 を得る．行 U10 では MODIFY(15,15,{‘#’})により要素 3 を要素 16 へ移動し，要素 3 は未使用となる．圧縮要素 15 の移動先となるシングル要素 t=1+’e’=7 についても同様の処理を行い，要素 7 が要素 17 に移動される．行 U12 では MODIFY(4,1,{‘a’,’e’})により圧縮

```
function MULTI(m_index);
   begin
(U1)  old_max:=MAX;
(U2)  LABEL:=GET_LABEL(m_index);
(U3)  new_base:=EX_CHECK(LABEL);
(U4)  if new_base=FALSE then return(FALSE);
(U5)  for each c in LABEL do
        begin
(U6)      t:=new_base+c;
(U7)      if BASE[t]=0 then continue;
(U8)      u:=t-BASE[|CHECK[t]|];
(U9)      b:=MAX+1-u;
(U10)     MODIFY(|CHECK[t]|,b,{u});
(U11)     if t=m_index then m_index:=b+u;
        end;
(U12) MODIFY(m_index,new_base,LABEL);
(U13) while MAX>old_max do
(U14)   SINGLE(|CHECK[MAX]|);
(U15) return(TRUE);
   end;
```

図14 関数 MULTI

Fig.14 Function MULTI.

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

図15 関数 EX_CHECK

Fig.15 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16 提案法によりキー”badge#”を削除したダブル配列

Fig.16 A double-array of after deletion for “badge#” by present method.

要素 11，15 が要素 3，7 へ移動し，要素 11，15 は未使用となる．行 U13，U14 は old_max=15 以降に退避したシングル要素 16，17 を関数 SINGLE によって前方の未使用要素へ移動する．これにより，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．COMPRESS の行 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)

表1　実験結果

Table 1　The simulation results.

削除キー数	10,000	30,000	50,000	70,000	90,000
使用要素数n	372,069	305,708	233,830	153,982	60,509
未使用要素数m					
削除法M	31,481	97,823	167,595	184,229	168,277
提案法	0	0	0	0	0
空間使用率 [%]					
削除法M	92	76	58	46	26
提案法	100	100	100	100	100
削除時間 [sec]					
削除法M	26	289	819	1,635	2,199
提案法	0.9	2.5	3.9	5.2	6.6
RATE:single [%]	62	64	67	70	75
AVG:n+m	5.8	5.6	5.4	5.4	5.5

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，理論的評価は全て最悪の場合とする．

5．1　理論的評価

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 は，E リンクを再構築する操作に依存し O(m)，X_CHECK は E リンクを先頭から走査するので O(m)となる[3]．MODIFY の計算量は，行 M2，M6 で e に関するループ構造となっており，行 M4, M8 で W_CHECK を実行するので, O(e(m+e+m))=O(m)となる（e は定数のため省略する）．以上より，削除法 M の計算量は, 削除キーの長さを k とするとき 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．

次に，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 T4 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 を実行するので O(n+m)となる．以上より，提案法の計算量は，DELETE の do-while 文で k 回 W_CHECK を実行し，COMPRESS の行 P7 の MULTI を for 文によって m 回繰り返すので，O(km+m(n+m))となる．

提案手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（例えばトライの根 1 以外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のランダム性が高いキー集合で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．また，削除法 M は削除を連続すると m が増加し削除速度が低下するが，提案法はシングル節の利用により m を少なく保つことができるため高速に削除できる．以上 2 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

提案手法の構成システムは約 500 行の C++言語で記述されており，Apple PowerMacG4（OS:MacOS9，CPU:733MHz）上で稼働している．実験に用いたデータは EDR 電子辞書[4]の英単語辞書であり，遷移文字の最大数 e は 256 である．

表 1 に英単語 10 万語をダブル配列に登録後，徐々に削除していった場合の実験結果を示す．表中の空間使用率は，ダブル配列の全要素数 n+m に対する使用要素数 n の割合である．AVG:n+m は EX_CHECK におけるダブル配列の平均走査回数である．RATE:single は n に対するシングル要素の割合を表すが，削除により登録キー数が変化した場合でも 60～75%がシングル要素であることから，その多数性が実証される．

表 1 の結果から，削除法 M は削除を連続するほど m が増加し空間使用率が低下するが，提案法は 100%の空間使用率を維持していることがわかる．また，提案法は削除法 M より約 30～300 倍高速に削除できている．これは提案法が m を非常に少なく保てていること，また AVG:n+m が全要素数 n+m に関係なく，たかだか 5 回程度であることに起因する．

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．

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，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手法を提案した．本手法により, ダブル配列法の応用分野が更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]　青江順一: キー検索技法ートライとその応用, 情報処理, Vol.34, pp.1244-1251 (1993).

[2]　Aoe, J.: An efficient digital search algorithm by using a double-array structure, IEEE Transactions on Software Engineering, Vol.SE-15, No.9, pp.1066-1077 (1989).

[3]　森田, 泓田, 大野, 青江: 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).

[4]　日本電子化辞書研究所: EDR 電子化辞書 (1996).