Vol. 47 No. 6 情報処理学会論文誌 June 2006

接頭辞ダブル配列における空間効率を低下させないキー削除法

矢田 晋[†] 大野将樹[††] 森田和宏[†]
泓田正雄[†] 吉成友子[†] 青江順一[†]

接頭辞ダブル配列はトライを高速かつコンパクトに実現するデータ構造である．しかし，キーの削除によって配列中に未使用の要素が蓄積し，空間効率が低下するという欠点がある．また，更新時間が未使用要素の数に依存するため，削除による空間効率の低下は更新時間の悪化にもつながる．本稿では，未使用要素を増加させることなく接頭辞ダブル配列からキーを削除する手法を提案する．EDR 電子化辞書の日英単語各 10 万件に対する実験により，提案法は従来法と比べて約 17–460 倍高速であり，高い空間効率を維持することが実証された．

A Deletion Method for Minimal Prefix Double-array without Increasing Empty Elements

Susumu Yata,[†] Masaki Oono,[††] Kazuhiro Morita,[†]
Masao Fuketa,[†] Tomoko Yoshinari[†] and Jun-ichi Aoe[†]

Minimal Prefix (MP) double-array represents a trie with two advantages — a fast retrieval and a compact dictionary. However, a key deletion produces empty elements and degrades the space efficiency of MP double-array. In addition, the deletion speed of MP double-array is degraded by the key deletion because the deletion time depends on the number of empty elements. This paper presents an efficient deletion method for MP double-array. The method dynamically removes keys from MP double-array without increasing empty elements. From experimental results for the key set which consists of 100,000 keys, it turned out that the presented method is about 17–460 times faster than the conventional method and maintains high space efficiency.

1. はじめに

トライ法[1] は，キーを構成する文字を分岐条件とする木構造（トライ）を用いたキー検索法である．トライ法でキーを検索する場合には，ルートから葉に向かって，検索キーの構成文字と対応する遷移を先頭から順番にたどる．そのため，遷移にかかる計算量が $O(1)$ であれば，検索時間は検索キーの長さのみに依存し，登録キーの総数には影響を受けないという利点がある．さらに，トライ法は共通接頭辞を容易に検出できるという，ハッシュ法や B-Tree[2] にはない特徴をあわせ持つため，スペルチェックや索引検索，形態素解析[3] などの幅広い分野で利用されている．

トライの代表的なデータ構造としては，配列構造とリスト構造がある．配列構造は，遷移を $O(1)$ で実現できるため，登録キーの総数に依存しない高速な検索が可能となる．しかし，配列がスパースになり，空間効率が悪いという欠点がある．一方，リスト構造は，配列構造よりも少ない空間使用量でトライを実現できるものの，遷移にかかる計算量が遷移元のノードから出る遷移の数に依存するため，検索時間が登録キーの総数に影響を受けるという欠点がある．

青江によって提案されたダブル配列[4] は，配列構造の高速性とリスト構造のコンパクト性をあわせ持つ効率的なデータ構造であり，各要素がノードと対応する一次元配列を用いてトライを実現する．ダブル配列において，対応するノードが存在しない要素のことを未使用要素という．未使用要素はトライの表現に不要なため，未使用要素が多くなるとダブル配列の空間効率は低下する．ダブル配列の欠点は，キーの削除によって配列中に未使用要素が蓄積し，空間効率が低下することである．この欠点は，学習によって自然言語辞書を動的に調整する場合や，データベースの索引を動的に更新する場合において問題となる．

† 徳島大学工学部
Faculty of Engineering, Tokushima University
†† 慶應義塾大学理工学部
Faculty of Science and Technology, Keio University

そこで，森田らは，キーを削除した後で未使用要素を除去する削除法を提案した[5]．森田らの手法では，最後方の使用要素（最終要素）とその兄弟要素を前方の未使用要素に移動することで未使用要素を除去するが，未使用要素の半数以上が除去されずに蓄積するという問題がある．この問題に対し，大野らは，最終要素の移動先候補に移動の容易なシングル要素（兄弟要素を持たない要素）を加えることで，高い空間効率を維持できるように改良した[6]．しかし，大野らの手法はシングル要素の多数性を利用するため，接頭辞ダブル配列のようなシングル要素を削減した構造に対して，空間効率を維持できないという問題がある．

本稿では，最終要素の兄弟要素が多いほど未使用要素の除去が難しくなることに着目し，接頭辞ダブル配列から未使用要素を効率的に除去できる削除法を提案する．EDR 電子化辞書の日英単語各 10 万件をキー集合とした実験により，提案法は従来法と比べて約 17–460 倍高速であり，高い空間効率を維持することが実証された．

2. ダブル配列

2.1 ダブル配列の概要

ダブル配列とは，BASE，CHECK という 2 つの整数を持つ要素の一次元配列でトライを表現したものであり，各要素は基本的にトライのノードと対応する．そのため，各ノードは対応する要素のインデックスで特定される．また，BASE，CHECK の符号により，各ノードは**表 1** に示すように分類される．以降，要素 s はダブル配列の s 番目の要素，ノード s は要素 s と対応するノードを表し，要素 s の BASE，CHECK はそれぞれ BASE[s]，CHECK[s] で表すものとする．

ダブル配列において，BASE[s] と CHECK[s] がともに負の場合，要素 s は未使用であるといい，ノード s が存在しないことを示す．未使用要素はトライの表現に不要なため，未使用要素が増加するとダブル配列の空間効率は低下する．未使用要素がなければ，ダブル配列の空間効率はリスト構造と同等になる．

未使用要素を循環リストとして連結したものを E-LINK といい，空間効率を維持しつつ高速にダブル配列を更新するために利用される[5]．E-LINK 上で k 番目の要素を r_k，最後の要素を r_{emax} とおくと，E-LINK は**図 1** のように表される[7]．ただし，ダブル配列における最後方の使用要素（最終要素）MAX より前方に未使用要素がまったくない状況でも E-LINK を維持するため，$r_{emax} = MAX + 1$ とする．また，新しい未使用要素は E-LINK の先頭に追加されるため，$MAX + 1$ が末尾として扱われることを除いて，ダブル配列上での順序と E-LINK 上での順序には関係がない．

表 1　BASE，CHECK の符号によるノードの判別
Table 1　Node types corresponding to signs of BASE and CHECK.

符号 BASE	符号 CHECK	ノードの種類
−	−	存在しない
−	+	葉ノード
+	−	遷移先が 1 つのノード
+	+	その他のノード

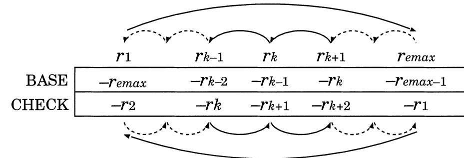


図 1　E-LINK による未使用要素の連結
Fig. 1　A linkage with empty elements on E-LINK.

葉ノードの BASE には，キーの ID を負にしたものが格納される．負にする理由は，葉ノードであるかどうかを判別するためである．

BASE が正のノードには他のノードへの遷移が存在し，ノード s からノード t への文字 c による遷移（$g(s,c) = t$ と表す）が定義されている場合，ダブル配列は次式を満足する．

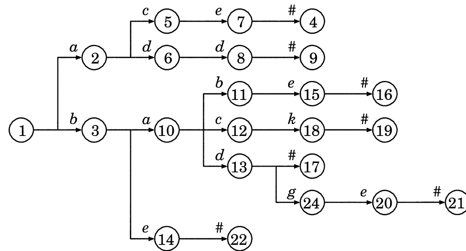
$$t = \mathrm{BASE}[s] + c, \quad |\mathrm{CHECK}[t]| = s \qquad (1)$$

すなわち，ノード s からの文字 c による遷移は，遷移先のインデックス t を BASE[s] と文字 c に割り当てられた整数（内部コード）の和で計算し，CHECK[t] の絶対値を遷移元のインデックス s と比較することで確認できる．式 (1) の計算量は $O(1)$ であるため，配列構造と同等の高速な検索が可能となる．

遷移元の確認に CHECK の絶対値を用いる理由は，遷移先が 1 つしか存在しないノードを，CHECK の符号を負にすることで区別するためである．この情報は，大野らの削除法において用いられる．また，ダブル配列において，遷移元が同じ要素は兄弟要素といい，兄弟要素を持たない要素はシングル要素という．

[例 1]　キー集合 K={“*ace*”, “*add*”, “*babe*”, “*back*”, “*bad*”, “*badge*”, “*be*”} に対するダブル配列を**図 2** に示す．文字 ‘#’ はキーの終端を表しており，“*bad*” と “*badge*” のようなキーを判別するために使用される．遷移文字の内部コードは，終端文字 ‘#’ を 0，文字 ‘*a*’〜‘*z*’ を 1〜26 としている．

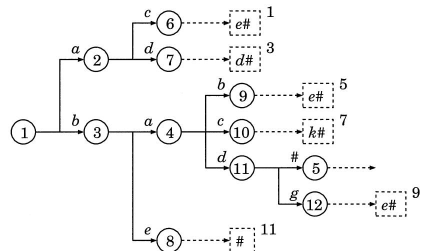
図 2 のダブル配列には 2 つの未使用要素 {23, 25} が存在しているため，$emax = 2$，$r_1 = 23$，$r_2 = 25$



	1	2	3	4	5	6	7	8	9	10	11	12	13
BASE	1	2	9	–1	2	4	4	9	–2	9	10	7	17
CHECK	0	1	1	7	–2	–2	–5	–6	8	3	–10	–10	10

	14	15	16	17	18	19	20	21	22	23	24	25
BASE	22	16	–3	–5	19	–4	21	–6	–7	–25	15	–23
CHECK	–3	–11	15	13	–12	18	–24	20	14	–25	–13	–23

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



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

	1	3	5	7	9	11
TAIL	e#	d#	e#	k#	e#	#

図 3 キー集合 K に対する接頭辞ダブル配列
Fig. 3 Minimal Prefix (MP) double-array for key set K.

となる．このことは，$r_{emax} = MAX + 1 = 25$ であり，E-LINK を r_{emax} から順方向に CHECK[25] $=$ -23，CHECK[23] $= -25$ とたどれること，逆方向に BASE[25] $= -23$，BASE[23] $= -25$ とたどれることから分かる．また，要素 2 にとって同じ遷移元 1 を持つ要素 3 は兄弟要素であり，兄弟要素を持たない要素 4，7，8，9 などはシングル要素である．

図 2 に示すダブル配列からキー "*be*" を検索する場合について考える．まず，ルートであるノード 1 からの，検索キーの一文字目 '*b*' による遷移を確認する．この例では，BASE[1]+ '*b*' $= 1 + 2 = 3$，CHECK[3] $= 1$ となり，$g(1,\text{'}b\text{'}) = 3$ が定義されていることが分かる．同様にして，$g(3,\text{'}e\text{'}) = 14$，$g(14,\text{'}\#\text{'}) = 22$ と確認できるため，キー "*be*" が登録されていることが分かる．（例終）

2.2 接頭辞ダブル配列

トライの空間使用量を削減する手法として，キーの判別に不要なノードを除去する Minimal Prefix（MP）トライがある．MP トライでは，キーを判別できる最前方のノードを分岐ノードと定義し☆，分岐ノードより後方のノードを除去する．除去対象となるノードについては，対応する遷移文字を連結した文字列を保存しておき，分岐ノードからリンクすることで解決する．分岐ノード以降のノードは文字列として保存されるため，空間使用量が削減される．

MP トライに対するダブル配列のことを接頭辞ダブル配列という．接頭辞ダブル配列では，分岐ノード以降の遷移文字を連結した文字列を，新たに導入する TAIL 配列に格納する．分岐ノードから TAIL 配列へのリンクは，分岐ノードの BASE を TAIL 配列におけるオフセットとして用いることで実現する．また，通常のダブル配列における葉ノードと同様に，オフセットを負の値にして BASE に格納することで他のノードと区別する．

［例 2］ キー集合 K に対する接頭辞ダブル配列を **図 3** に示し☆☆，キー "*badge*" を検索する場合について考える．通常のダブル配列における検索と同様に，ルートから検索キーの構成文字による遷移を確認すると，$g(1,\text{'}b\text{'}) = 3$，$g(3,\text{'}a\text{'}) = 4$，$g(4,\text{'}d\text{'}) = 11$，$g(11,\text{'}g\text{'}) = 12$ となり，BASE[12] $= -9 < 0$ より分岐ノードに到達したことが分かる．遷移の確認において照合していない検索キーの残りの文字列 "*e*" が TAIL[9] に格納されている文字列と等しいため，キー "*badge*" が登録されていることが分かる．なお，キー "*bad*" を検索する場合は，遷移の確認において終端文字 '#' も照合されるので，TAIL 配列との照合は不要である．そのため，BASE[5] は符号のみが利用されることになる．（例終）

3. 削除法と問題点

3.1 削 除 法

ダブル配列における削除は，削除キーのみと対応し，他のキーには影響しないノードを除去することであり，除去対象のノードと対応する要素を未使用にすることで実現される．接頭辞ダブル配列については，削除によって分岐ノードが変更される場合があり，新たな分岐ノード以降のノードも除去する必要がある．しかし，

☆ 図 2 のダブル配列では，ノード 5，6，11，12，14，17，24 が分岐ノードとなる．

☆☆ 図 3 の下線部は分岐ノードから TAIL 配列へのリンクを表している．

ノードを除去するだけでは，削除キー数に比例して未使用要素が増加し，空間効率が低下するという問題がある．

この問題に対し，森田らは，ノードを除去した後で未使用要素を除去する削除法を提案した．森田らの手法は，最終要素とその兄弟要素を併せて圧縮要素とし，圧縮要素を前方の未使用要素に移動することで，ダブル配列中の未使用要素を除去する．しかし，移動先候補が少ないと圧縮要素の移動が困難になるため，未使用要素が足りないと未使用要素を除去できないというジレンマに陥り，高い空間効率を保てないという問題がある．そこで，大野らは，他の要素に依存することなく移動可能である，シングル要素を移動先候補に加えた削除法を提案した（以降，シングル要素移動法と表記する）．圧縮要素の移動先として採用したシングル要素は，ダブル配列の末尾に退避しておき，圧縮要素を移動した後であらためて前方の未使用要素に移動する．シングル要素移動法は，未使用要素のみを移動先候補とする場合と比較して，高い空間効率を維持しつつ，高速にキーを削除することができる．シングル要素移動法における，未使用要素を除去する手続き Compress を以下に示す．

[手続き Compress()]

```
define SIZE = MAX + 1
begin
(c-1)  while ( BASE[SIZE] ≠ −SIZE ) do
       begin
(c-2)      prev := |CHECK[MAX]|;
(c-3)      labels := GetLabel( prev );
(c-4)      base := XCheck( prev, labels );
(c-5)      if ( base ≥ BASE[prev] ) then
               return;
(c-6)      Shelter( base, labels );
(c-7)      Move( prev, base, labels );
       end;
end;
```

手続き Compress は，行 c-1 から開始するループにおいて，未使用要素がなくなるか移動先がなくなるまで圧縮要素の移動を繰り返す．まず，行 c-2 で最終要素 MAX の遷移元 $prev$ を取得する．次に，行 c-3 において関数 GetLabel を呼び出し，$prev$ から圧縮要素への遷移文字の集合を $labels$ に格納する．行 c-4 では，関数 XCheck により圧縮要素の移動先をオフセット $base$ として取得する．前方に圧縮要素を移動できない場合，行 c-5 において手続き Compress は終了する．移動先が見つかった場合，行 c-6 において移動先に含まれるシングル要素を MAX の後方に退避する．行 c-7 で手続き Move により圧縮要素を実際に移動し，行 c-1 に戻る．手続き Move は，$labels$ のすべての遷移文字 c に対して，要素 BASE[$prev$] + c を未使用要素 $base + c$ に移動する．なお，後方に退避したシングル要素は，次のループにおいて，新たな圧縮要素として未使用要素に移動されることになる．

行 c-4 の関数 XCheck は，E-LINK を走査して圧縮要素の移動先を求める関数である．ただし，つねに E-LINK の先頭から走査を開始すると，E-LINK の前方の要素付近にシングル要素以外が集中し，探索効率の低下につながる．そこで，探索位置を格納する大域変数 $XPOS$ を導入し，探索が終了した位置を次回の探索開始位置とする．

[関数 XCheck(*index*, *labels*)]

```
define labels_1 is the smallest letter in labels.
begin
(x-1)  start := XPOS;
(x-2)  old := BASE[index];
       repeat
(x-3)      base := XPOS − labels_1;
(x-4)      XPOS := −CHECK[XPOS];
(x-5)      if ( 0 < base < old ) then
           begin
(x-6)          for each c in labels do
               begin
(x-7)              flag := IsTarget( base + c );
(x-8)              if ( flag = FALSE ) then
                       break;
               end;
(x-9)          if ( flag = TRUE ) then
                   return base;
           end;
(x-10) until ( XPOS ≠ start );
(x-11) return old;
end;
```

関数 XCheck は，行 x-1，x-2 で探索開始位置と元の BASE を保存する．行 x-3 において未使用要素 $XPOS$ に $labels$ の最小の遷移文字 $labels_1$ が対応するようなオフセット $base$ を計算し，行 x-4 で探索位置を E-LINK 上で次の要素に設定する．行 x-5 では，$base$ が移動先として利用可能な範囲にあるかどうかを確認する．行 x-6 からのループは，$labels$ のすべての遷移文字 c に対して，$base+c$ が移動先候補であるかどうかを検証する．なお，行 x-7 の関数 IsTarget は，受け取った要素が移動先候補かどうかを判定する関数であ

る．すべての $base + c$ が移動先候補であれば，行 x-9 において移動先のオフセット $base$ を返す．E-LINK を一巡しても移動先が見つからなければ，行 x-11 において移動元のオフセット old を返す．

シングル要素移動法では，未使用要素とシングル要素を移動先候補とするため，要素 $index$ が移動先候補かどうかを判定する関数 IsTarget は以下のように与えられる．

[関数 IsTarget($index$)]

begin
(t-1) $prev$:= |CHECK[$index$]|;
(t-2) **if** (CHECK[$prev$] < 0) **then**
    return TRUE;
(t-3) **return** FALSE;
end;

要素 $index$ が未使用要素の場合，要素 $prev$ は次の未使用要素に対応するので，CHECK[$prev$] は明らかに負である．そうでなければ，$prev$ は $index$ の遷移元に対応するので，$index$ に兄弟が存在しない場合にのみ CHECK[$prev$] が負となる．そのため，$index$ が未使用要素もしくはシングル要素であれば TRUE が返り，それ以外であれば FALSE が返る．

[手続き Shelter($base$, $labels$)]

begin
(s-1) **for each** c **in** $labels$ **do**
  begin
(s-2) $index := base + c$;
(s-3)   **if** (BASE[$index$] > 0 **or**
      CHECK[$index$] > 0) **then**
    begin
(s-4) $prev$:= |CHECK[$index$]|;
(s-5) $labels_mv$:= GetLabel($prev$);
(s-6) $dest := SIZE - labels_mv_1$;
(s-7)   Move($prev$, $dest$, $labels_mv$);
    end;
  end;
end;

手続き Shelter は，$labels$ のすべての遷移文字 c に対して，要素 $base + c$ が未使用でなければダブル配列の末尾に移動する．未使用かどうかの判定は行 s-3 で行われ，要素の移動は行 s-4 から s-7 において行われる．

[例 3] 図 2 に示すダブル配列から，シングル要素移動法を用いてキー "add" を削除する場合，削除キー以外には影響を与えない要素 {6, 8, 9} を未使用にした後，手続き Compress によって未使用要素を除去する．この例では，{17, 24} を {8, 15} へ，{25} を {9} へ，{22} を {17} へ，{21} を {6} へという順で圧縮要素を移動することにより，最終的に未使用要素がなくなり手続き Compress は終了する．詳細を以下で説明する．

	1	2	3	4	5	6	7	8	9	10	11	12	13
BASE	1	2	9	–1	2	–25	4	–6	–8	9	10	7	17
CHECK	0	–1	1	7	–2	–8	–5	–9	–23	3	–10	–10	10

	14	15	16	17	18	19	20	21	22	23	24	25
BASE	22	16	–3	–5	19	–4	21	–6	–7	–9	15	–23
CHECK	–3	–11	15	13	–12	18	–24	20	14	–25	–13	–6

	1	2	3	4	5
E–LINK	6	8	9	23	25

(a) Immediately after deletion for key "add"

	1	2	3	4	5	6	7	8	9	10	11	12	13
BASE	1	2	9	–1	2	–24	4	–5	–6	9	20	7	8
CHECK	0	–1	1	7	–2	–9	–5	13	–23	3	–10	–10	10

	14	15	16	17	18	19	20	21	22	23	24	25	26
BASE	22	15	–3	–26	19	–4	21	–6	–7	–9	–17	16	–23
CHECK	–3	–13	15	–24	–12	18	–15	20	14	–26	–6	–11	–17

	1	2	3	4	5	6
E–LINK	17	24	6	9	23	26

(b) After the first loop of procedure Compress

	1	2	3	4	5	6	7	8	9	10	11	12	13
BASE	1	2	9	–1	2	–6	4	–5	16	9	4	7	8
CHECK	0	–1	1	7	–2	20	–5	13	–11	3	–10	–10	10

	14	15	16	17	18	19	20	21
BASE	17	15	–3	–7	19	–4	6	–21
CHECK	–3	–13	9	14	–12	18	–15	–21

	1
E–LINK	21

(c) The end of procedure Compress

図 4 シングル要素移動法を用いたキー "add" の削除
Fig. 4 Deletion for key "add" by using single elements.

図 4 は，キー "add" 削除直後 (a)，圧縮要素 {17, 24} を移動した後 (b)，手続き Compress が終了した後 (c) のダブル配列を示している．図 4 において，BASE，CHECK に対する下線は更新箇所を表しており，E-LINK に対する下線は $XPOS$ を表している．ただし，削除前の状態で $XPOS = 23$ であるものとする．

まず，削除キーと対応する葉ノード 9 から逆順に遷移をたどりつつ，要素 {9, 8, 6} を未使用にする．新たな未使用要素は E-LINK の先頭に追加されるため，E-LINK は {6, 8, 9, 23, 25} の順になる（図 4 (a)）．次に，手続き Compress が呼び出される．

手続き Compress における最初のループでは，行 c-1 において，BASE[$SIZE$] = BASE[25] $\neq$ –25 より，最終要素の前方に未使用要素があることが分

かる．行 c-2 では，最終要素 $MAX = 24$ の遷移元 $|$CHECK[24]$| = 13$ が $prev$ に設定される．行 c-3 では，関数 XCheck により得られる，要素 $prev$ から圧縮要素への遷移文字の集合 {‘#’, ‘g’} = {0, 7} を $labels$ に格納する．行 c-4 で関数 XCheck を呼び出し，圧縮要素の移動先をオフセット $base$ として求める．

関数 XCheck では，E-LINK{6, 8, 9, 23, 25} を $XPOS = 23$ から走査する．この例では，移動元のオフセット old は BASE[13] $= 17$ で，$labels$ の中で最小の遷移文字 $labels_1$ は ‘#’ $= 0$ で与えられる．初めに確認される $XPOS = 23$，25 については，$base = 23$，$25 > old = 17$ となるので，行 x-5 において不適切と判断される．$XPOS = 6$ のときは，$0 < base < old$ を満足するものの，要素 $base +$ ‘g’ $= 6+7 = 13$ が未使用要素でもシングル要素でもないため，行 x-7 の関数 IsTarget で $|$CHECK[13]$| = 10$，CHECK[10] $= 3 > 0$ となり，不適切と判断される．$XPOS = 8$ のとき，$labels$ のすべての遷移文字 c に対して，要素 $base + c$ が未使用要素もしくはシングル要素となるので，関数 XCheck は $base = 8$ を返して終了する．このとき，圧縮要素の移動先は {8, 15}，$XPOS = 9$ となる．

次に，行 c-6 の手続き Shelter によって，移動先に含まれる使用要素をダブル配列の後方に退避する．要素 8 については，未使用要素であるため何もしない．要素 15 については，行 s-4 で遷移元 $|$CHECK[15]$| = 11$ を $prev$ に設定する．行 s-5 で要素 $prev$ から出る遷移文字 ‘e’ を取得し，行 s-6 で最終要素の後方へのオフセット $dest$ を $SIZE -$ ‘e’ $= 25 - 5 = 20$ により求めた後，行 s-7 で実際に要素 15 を要素 25 へと移動する．手続き Move では，関連する遷移の再定義とともに E-LINK の更新が行われる．最後に，行 c-7 において圧縮要素 {17, 24} を前方の未使用要素 {8, 15} に移動することで，手続き Compress の最初のループは終了する（図 4 (b)）．

未使用要素がなくなるまで手続き Compress は継続するものの，初回以降は最終要素がシングル要素になり，行 c-4 の関数 XCheck はほぼ E-LINK の先頭の要素を採用する，行 c-6 の手続き Shelter は何もしないなど，動作が非常に単純になるので，以降の説明は省略する．　（例終）

3.2　削除法の問題点

シングル要素移動法の問題点は，シングル要素の多数性を利用するため，シングル要素の少ない接頭辞ダブル配列において，高い空間効率を維持できないことである．また，関数 XCheck の計算量は未使用要素の数に比例するため，空間効率の低下が削除速度の低下につながるという問題もある．

	1	2	3	4	5	6	7	8	9	10	11	12
BASE	1	3	3	7	<u>–12</u>	–1	–3	–11	–5	–7	<u>–12</u>	<u>–5</u>
CHECK	0	1	1	3	<u>–12</u>	2	2	3	4	4	<u>4</u>	<u>–5</u>

	1	2
E–LINK	5	<u>12</u>

図 5　キー “$badge$” 削除後の接頭辞ダブル配列
Fig. 5　MP double-array after deletion for key “$badge$”.

[例 4]　図 3 に示す接頭辞ダブル配列から，シングル要素移動法を用いてキー “$badge$” を削除する場合について考える．まず，削除キーのみと対応するノード 12 を除去する．このとき，キー “bad” と対応する分岐ノードがノード 11 になるので，ノード 5 も除去する．要素 {5, 12} を未使用にしたダブル配列は**図 5** に示すようになる．図 5 において，BASE，CHECK に対する下線は更新箇所を表しており，E-LINK に対する下線は $XPOS$ を表している．次に手続き Compress による圧縮要素の移動を試みるものの，移動先候補 {5} に対して圧縮要素 {9, 10, 11} の方が多く，圧縮要素の移動は明らかに不可能である．そのため，未使用要素を除去することなく手続き Compress は終了する．　（例終）

4.　拡張削除法

4.1　拡張削除法の概要

シングル要素移動法の問題点は，圧縮要素が多くなるほど移動が難しくなるにもかかわらず，移動先候補の条件が固定されていることである．この問題により，圧縮要素が多い状況で移動先候補が不足し，未使用要素をすべて除去できずに圧縮が終了してしまう．そこで，移動先候補の条件を圧縮要素の数に応じて変更する適応型要素移動法を提案する．

適応型要素移動法では，兄弟要素が少ない要素ほど移動が容易であることを考慮し，最終要素と比べて兄弟要素が少ない要素を移動先候補とする．これにより，圧縮要素が多いほど移動先候補を大量に確保できるため，圧縮要素の移動に失敗しにくくなる．

4.2　拡張削除法のアルゴリズム

適応型要素移動法では，要素 $index$ が移動先候補かどうかを判定する関数 IsTarget が以下のように拡張される．新たな引数 num には，圧縮要素の数を受け取る．

[関数 IsTarget($index$, num) の拡張]

<u>**define**</u> $|labels|$ is the number of letters in $labels$.
<u>**begin**</u>

```
(t'-1)  prev := |CHECK[index]|;
(t'-2)  if ( CHECK[prev] < 0 ) then
            return TRUE;
(t'-3)  else if ( num = 2 ) then
            return FALSE;
(t'-4)  labels := GetLabel( prev );
(t'-5)  if ( |labels| < num ) then
            return TRUE;
(t'-6)  return FALSE;
end;
```

行 t'-1，t'-2 は，拡張前と同様に，要素 $index$ が未使用要素かシングル要素であれば TRUE を返す．行 t'-3 は，移動先候補の条件が未使用要素とシングル要素のみの場合に FALSE を返す．行 t'-4 で要素 $prev$ から出る遷移文字の集合 $labels$ を求め，$labels$ に含まれる遷移文字の数が圧縮要素の数よりも少なければ行 t'-5 で TRUE を返す．そうでなければ，行 t'-6 において FALSE を返す．

なお，関数 IsTarget に圧縮要素の数を受け取るための引数 num が追加されるので，関数 XCheck の行 x-7 は以下のように変更される．

[関数 XCheck(*index*, *labels*) の変更]

```
(x'-7)  flag := IsTarget( base + c, |labels| );
```

[例 5]　図 2 に示すダブル配列から，適応型要素移動法を用いてキー "*add*" を削除する場合，シングル要素移動法を用いた場合と同じ結果になる．これは，圧縮要素が 2 つ以下の場合，各手法による移動先候補の条件が同じになるからである．

図 3 に示す接頭辞ダブル配列から，適応型要素移動法によりキー "*badge*" を削除する場合について考える．シングル要素移動法と同様，要素 {5, 12} を未使用にし（図 5），手続き Compress によって圧縮要素を前方に移動する．ただし，適応型要素移動法では，圧縮要素 {9, 10, 11} に対して，兄弟要素が 1 つ以下の要素 {2 ～ 8} が移動先候補となる．これにより，{9, 10, 11} を {5, 6, 7} へ，{12, 13} を {9, 10} へという順で圧縮要素が移動され，最終的に未使用要素がなくなり手続き Compress は終了する．

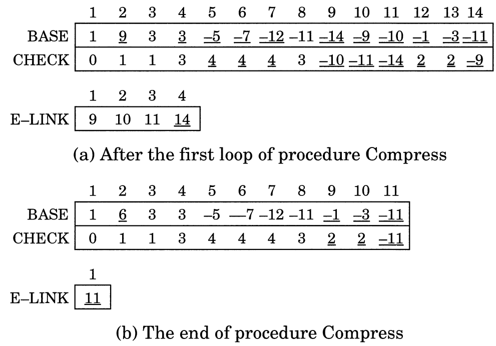


図 6　適応型要素移動法を用いた "*badge*" の削除
Fig. 6　Deletion for key "*badge*" by the new method.

図 6 は，圧縮要素 {9, 10, 11} を移動した後 (a) と手続き Compress が終了した後 (b) のダブル配列を示している．図 6 において，BASE，CHECK に対する下線は更新箇所を表しており，E-LINK に対する下線は $XPOS$ を表している．圧縮要素 {9, 10, 11} の移動に関する詳細を以下で説明する．

手続き Compress は，行 c-2 で最終要素 $MAX = 11$ の遷移元 $prev = 4$ を取得し，行 c-3 で圧縮要素への遷移文字の集合 $labels = \{$'b', 'c', 'd'$\}$ を取得する．行 c-4 では，関数 XCheck により圧縮要素の移動先を求める．

関数 XCheck では，E-LINK{5, 12} を $XPOS = 12$ から走査する．$XPOS = 12$ については，移動元のオフセット $old = \mathrm{BASE}[4] = 7$ と最小の遷移文字 $labels_1 =$ 'b' $= 2$ より，$base = 10 > old$ となるので行 x-5 において不適切と判断される．$XPOS = 5$ のときは，$0 < base < old$ を満足し，$labels$ のすべての遷移文字 c に対して，移動先となる要素 $base + c$ が兄弟要素を 2 つ以上持たないので，関数 XCheck は $base = 3$ を返す．

手続き Compress の行 c-6 で呼び出される手続き Shelter は，圧縮要素の移動先 {5, 6, 7} に含まれる使用要素をダブル配列の末尾に移動する．まず，遷移文字 'b' について評価すると，要素 $base +$ 'b' $= 5$ が未使用要素であることが分かるため，何もしない．遷移文字 'c' については，要素 $base +$ 'c' $= 6$ が使用要素であるため，行 s-4 から s-7 で移動される．このとき，要素 6 の遷移元は $|\mathrm{CHECK}[6]| = 2$ であり，要素 2 から出る遷移文字は $labels_mv = \{$'c', 'd'$\}$ であるので，要素 6 の兄弟要素 7 も同時に移動されることになる．結果的に，要素 {6, 7} がダブル配列の末尾 {12, 13} へと移動される．遷移文字 'd' に関しては，要素 $base +$ 'd' $= 7$ が要素 6 の兄弟要素として移動済みである．

最後に，手続き Compress の行 c-7 において，圧縮要素 {9, 10, 11} が前方の未使用要素 {5, 6, 7} に移動される．以降のループでは，最終要素が兄弟要素を 2 つ以上持たないため，シングル要素移動法と同様の操作が行われる．　　（例終）

5. 評 価

シングル要素移動法と適応型要素移動法をそれぞれ約450行のC言語により実装し，DELL OPTIPLEX GX270（CPU: Pentium4 3.2 GHz）上で比較実験を行った．実験では，EDR電子化辞書[8]から無作為に選択した日英単語各10万件をキー集合として使用した．各キー集合に対する接頭辞ダブル配列の初期状態を**表2**に示す．未使用要素数は，最終要素より前方の未使用要素の数を表している．空間使用量は，BASE, CHECKおよびTAILに割り当てられた領域の合計を表している．BASE，CHECKにはそれぞれ4 bytesの整数を使用したので，TAIL配列の長さを$TLEN$とおくと，空間使用量は$8 \cdot SIZE + TLEN$ bytesで算出される．

実験では，接頭辞ダブル配列からランダム順にキーを削除し，一定数のキーを削除した後の未使用要素数，要素使用率，空間使用量，および各区間における削除時間を求めた．**表3**に示す実験結果は10回の試行の平均を示しており，要素使用率は全要素に対して使用要素の占める割合，削除時間はキーを1つ削除するために要した平均時間を表している．なお，構築や検索にかかる時間およびTAILの長さは，どちらの手法を用いても等しくなるので省略した．

表3より，シングル要素移動法を用いた場合，削除キー数に比例して未使用要素が増加し，要素使用率が約50–60%まで低下していることが分かる．表2におけるシングル要素の割合を考慮すると，全要素の50%以上を移動先候補として確保できなければ，圧縮要素を安定して移動することは難しいということも分かる．一方，適応型要素移動法は未使用要素がなくても移動先候補を十分に確保できるため，要素使用率がほぼ100%という理想的な状態を維持している．その結果，適応型要素移動法はシングル要素移動法と比べて，TAIL配列を含めた空間使用量を最大で約30–40%削減できている．

表3の削除時間より，適応型要素移動法はシングル要素移動法と比べて約17–460倍高速にキーを削除できていることが分かる．また，削除時間がほとんど変化しない適応型要素移動法に対し，シングル要素移動法は未使用要素の増加とともに削除時間が長くなっている．これは，圧縮要素の移動先を求めるとき，関数XCheckにおいてE-LINKを走査するからである．

以上のことから，適応型要素移動法はコンパクトかつ高速なキー削除法であり，接頭辞ダブル配列の動的更新において有効であるといえる．

表2 接頭辞ダブル配列の初期状態
Table 2 MP double-arrays before deletion.

言語	英語	日本語
キー数	100,000	100,000
平均キー長 (bytes)	8.86	7.64
最大キー長 (bytes)	28	54
要素数	192,430	160,819
未使用要素数	141	689
シングル要素数	38,439 (20%)	20,324 (13%)
空間使用量 (bytes)	1,861,722	1,672,859

表3 実験結果
Table 3 The simulation results.

削除キー数	10,000	30,000	50,000	70,000	90,000
英語					
未使用要素数					
従来法	18,916	51,505	61,054	32,832	12,453
提案法	60	47	53	66	63
要素使用率 (%)					
従来法	90.16	72.31	61.30	63.11	58.45
提案法	99.97	99.96	99.94	99.98	99.54
空間使用量 (kb)					
従来法	1,877	1,865	1,667	1,169	745
提案法	1,726	1,454	1,179	907	646
削除時間 (ms)					
従来法	0.762	3.261	5.544	3.171	0.887
提案法	0.012	0.012	0.012	0.012	0.012
日本語					
未使用要素数					
従来法	17,182	49,608	80,513	40,403	18,450
提案法	94	79	77	95	101
要素使用率 (%)					
従来法	89.31	69.07	49.27	53.44	45.53
提案法	99.95	99.92	99.87	99.78	99.40
空間使用量 (kb)					
従来法	1,690	1,721	1,743	1,202	814
提案法	1,553	1,325	1,100	880	667
削除時間 (ms)					
従来法	0.536	2.521	6.228	6.680	1.387
提案法	0.031	0.026	0.024	0.023	0.023

6. おわりに

本稿では，圧縮要素によって移動先候補の配置条件が変化することに着目し，接頭辞ダブル配列に対して高い空間効率を維持する削除法を提案した．また，実験によって，提案法は空間効率において優れているだけでなく，削除を高速化できることも実証した．提案法により，接頭辞ダブル配列の応用は，動的に更新するデータベースの索引や自動学習を考慮した自然言語辞書など，効率的な削除が望まれる分野へと拡大される．また，様々な検索システムにおいて，現在用いている静的な辞書や他のキー検索法を接頭辞ダブル配列に置き換えることで，利便性や検索機能の向上が期待される．

本稿では未使用要素が蓄積するという問題に着目したが，接頭辞ダブル配列における削除には，TAIL 配列の空間効率が低下するという問題も存在する．この問題を効率良く動的に解決する手法を提案することが今後の課題としてあげられる．

参考文献

1) Fredkin, E.: Trie Memory, *Comm.ACM*, Vol.3, No.9, pp.490–500 (1960).
2) 石畑　清：アルゴリズムとデータ構造，岩波講座ソフトウェア科学 3，岩波書店 (1989).
3) 松本裕治：形態素解析システム茶筌 (2004). http://chasen.naist.jp/hiki/ChaSen/
4) 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，Vol.J71-D, No.9, pp.1592–1600 (1988).
5) 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).
6) 大野将樹，森田和宏，泓田正雄，青江順一：ダブル配列におけるキー削除の効率化手法，情報処理学会論文誌，Vol.44, No.5, pp.1311–1320 (2003).
7) 大野将樹，森田和宏，泓田正雄，青江順一：ダブル配列による自然言語辞書の高速更新法，言語処理学会第 11 回年次大会発表論文集，pp.745–748 (2005).
8) 日本電子化辞書研究所：EDR 電子化辞書 (1996).

(平成 17 年 8 月 8 日受付)
(平成 18 年 3 月 2 日採録)

矢田　　晋（学生会員）

昭和 57 年生．平成 16 年徳島大学工学部知能情報工学科卒業．平成 18 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．

大野　将樹（正会員）

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．平成 16 年同大学院博士後期課程修了．同年徳島大学工学部研究員．平成 18 年慶應義塾大学理工学部情報工学科助手，現在に至る．博士（工学）．情報検索，自然言語処理の研究に従事．電子情報通信学会会員．

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．IEEE 会員．

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．電子情報通信学会，言語処理学会，IEEE CS 各会員．

吉成　友子

昭和 55 年生．平成 15 年徳島大学工学部知能情報工学科卒業．平成 17 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．

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，言語処理学会，IEEE 各会員．