圧縮ディジタル探索木における辞書情報更新の高速化手法

中村康正[†] 望月久稔[†]

ディジタル探索は，自然言語処理システムの辞書情報構築を中心に広く用いられている検索技法である．また，ディジタル探索木には共通接頭辞のみを格納した圧縮ディジタル探索木がある．圧縮ディジタル探索木のデータ構造として高速性とコンパクト性をあわせ持つダブル配列法があるが，他の動的検索法に比べ更新速度が高速であるとはいえない．そこで本論文では，ダブル配列法に対する更新処理の高速化手法を提案する．提案手法では，圧縮ディジタル探索木上の新規節点を管理することにより追加処理を高速化し，高速化にともなう削除時間の増加を抑制する．20 万語の辞書データに対する実験を行った結果，従来法と比べ追加処理は高速となり，削除処理は同等となることが分かった．

Fast Computation of Updating Method of a Dictionary for Compression Digital Search Tree

Yasumasa Nakamura[†] and Hisatoshi Mochizuki[†]

Digital search is a information retrieval application used widely, such as dictionary information construction of natural language processing system. Digital search tree has compression digital search tree stored only common prefix. As a fast and compact data structure for compression digital search tree, a double-array is presented. However, the updating processing is not faster than other dynamic retrieval methods. In this paper, we presents a faster method of updates for double-array. In the proposed method, we make insertion algorithm fast by managing a new node position on a compression digital search tree, and reduction algorithm of deletion time. The simulation results for 200 thousands keys turned out that the presented method for insertion processing is faster than original method, and deletion processing is equivalent to original method.

1. はじめに

ディジタル探索は，キー自体で構成される他の検索技法とは異なり，キーの表記記号を遷移として持つディジタル探索木（以下，DS 木）で表現される．そのため，自然言語処理システムの辞書情報構築を中心に広く用いられている[4]．また，DS 木を構成する節点はキー数とともに増加するため，共通接頭辞のみを DS 木上の節点として節点数を抑制した圧縮 DS 木がある．

DS 木を実現する構造としてトライ構造があり，配列を用いた手法とリストを用いた手法がよく知られている[5]．前者は，配列の特性から $O(1)$ で遷移可能であるが，大きな空間を必要とする．後者は，不必要な遷移情報を持たないため小さな空間ですむが，$O(1)$ で遷移できない．

そこで，配列の高速性とリストのコンパクト性をあわせ持つダブル配列法がある[4),5)]．ダブル配列法は，圧縮 DS 木を記憶する 2 つの 1 次元配列 BASE，CHECK と共通接頭辞以降を遷移列として記憶する 1 次元配列 TAIL を用いて圧縮 DS 木を実現する[4]．しかし，動的検索法に比べてキーの更新処理が高速であるとはいえない．

圧縮 DS 木ではなく DS 木として実現したダブル配列法の追加処理に対する高速化について，DS 木上の節点として使用されていない配列 BASE，CHECK の要素（以下，未使用要素）を単方向リストにより管理した手法があるが，削除処理に多くの計算量を必要とする[9]．そこで，未使用要素リストを双方向へ拡張した手法が提案されている[6]．

本論文では，ダブル配列法を動的検索法として確立するために追加処理の高速化手法を提案する．加えて，追加処理の高速化にともなう削除時間の増加に対する解決手法を提案する．

以下，2 章でダブル配列法とその問題点を述べ，3 章で追加処理の高速化手法と削除時間の増加に対する抑制手法を提案する．4 章で提案手法の評価を与え，

† 大阪教育大学
Osaka Kyoiku University

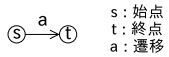

図 1 DS 木の遷移関係
Fig. 1 Transition of digital search tree.

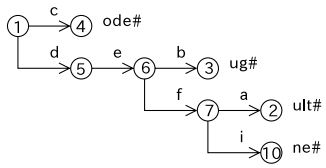

図 2 圧縮 DS 木の例
Fig. 2 An example of compression digital search tree.

5 章で本論文のまとめと今後の課題についてふれる．

2. ダブル配列法

本章では，ダブル配列法における圧縮 DS 木の実現方法を述べ，詳細なアルゴリズムに対してキー集合 K ={“code#”，“debug#”，“default#”，“define#”} を例にとり具体的に説明する．ここで，表記記号 ‘#’ を終端記号とし，表記記号 ‘#’，‘a’，···，‘z’ の内部表現値を 0，1，···，26 とする．また，本章で用いた例は後章でも対応させて使用する．

2.1 ダブル配列法のデータ構造

ダブル配列法のデータ構造は，2 つの一次元配列を用いて実現する．配列 BASE と配列 CHECK によって，**図 1** に示す DS 木の始点 s から終点 t へ表記記号 ‘a’ で遷移する関係を表す．表記記号 ‘a’ の内部表現値を単に a で表すとき，その関係を式 (1)，(2) で定義する[4]．すなわち，配列 BASE は行置換関数を表し，遷移の基底位置を与え，配列 CHECK は DS 木の親子関係を一意に決定する．以下，ダブル配列上の要素 x に関する BASE 値を $B[x]$，CHECK 値を $C[x]$ とする．

$$B[s] + a = t \quad (1)$$
$$C[t] = s \quad (2)$$

圧縮 DS 木を実現するために，各キーにおける他のキーと共有していない遷移を配列 TAIL に設定し，DS 木上の節点数を抑制する[4),5)]．以下，配列 TAIL の pos 番目の要素を $T[pos]$ とし，pos 番目から始まる遷移列を $T + pos$ とする．

例 1：キー集合 K に対する圧縮 DS 木を**図 2** に，ダブル配列を**図 3** に示す．また，初期値を配列 BASE と配列 CHECK は 0，配列 TAIL は ‘#’ とする．(例終了)

以下，図 2 における節点 5 のような出次数が 1 である節点をシングル節点と呼ぶ．また，節点 3 のような葉は，配列 TAIL の要素番号を BASE 値に保持し，

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

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
TAIL	o	d	e	#	u	g	#	g	#	u	l	t	#	#	n	e	#

図 3 ダブル配列の例
Fig. 3 An example of double-array structure.

Function:Search(key**)**
(S1) $t = 1$;
(S2) $pos = 0$;
(S3) **while:**節点 t が SP 節点ではない
(S4) $s = t$;
(S5) $t = B[s] + key[pos]$;
(S6) **if:**$C[t]$ と s が異なる
(S7) return FALSE;
(S8) $pos = pos + 1$;
(S9) return StrCmp($T + (-B[t])$, $key + pos$);

図 4 関数：Search
Fig. 4 Function: Search.

探索対象となる葉以下の遷移を一意に決定する．この意味で葉をセパレート節点（以下，SP 節点）と呼ぶ．また，この BASE 値をマイナス値とし SP 節点と他の節点とを区別する．SP 節点が指す配列 TAIL に設定されている遷移列を SP ストリングと呼び[5)]，図 2 では SP 節点の右側に示す．SP 節点 3 が指す SP ストリングは，図 3 における $B[3]$ がマイナス値で示す $T[5]$ から始まる遷移列 “ug#”である．

2.2 探索アルゴリズム

ダブル配列法におけるキー探索の成功条件は，式 (1)，(2) を満足させながら根から SP 節点まで遷移し，SP 節点が指す SP ストリングと残りのキーが一致することである[5)]．また，探索が失敗する場合は下記に示す (I)，(II) である．

(I) 始点 s から $label$ で遷移する終点 t が存在しない．
(II) SP ストリングと残りのキーが異なる．

キー key を探索するアルゴリズムである関数 Search を**図 4** に示す．以下，key の pos 番目の要素を $key[pos]$ とし，key の pos 番目から始まる文字列を $key + pos$ とする．図 4 中の S9 で用いる関数 StrCmp($x + p$, $y + q$) は，文字列 $x + p$ と $y + q$ が等しければ TRUE を，そうではなければ FALSE を返す．ここで，探索失敗 (I) は S7 で，(II) は S9 で FALSE を返す．

例 2：図 3 から “default#”を探索する．図 4 中の S1 で終点 t に根 1 を，S2 で key =“default#”の文字位置 pos に先頭文字位置 0 を設定する．S3 で $B[1] < 0$ ではないので，節点 1 は SP 節点ではないことが分かる．よって，S4 から S7 で始点 1 から $key[0]$ =‘d’ で遷移する終点を調べる．まず S4 で始点 s に 1 を退避

させ，S5 で t に $B[1]$+d= $1+4=5$ を設定する．次に S6 で $C[5]=1$ と $s=1$ が等しいので，始点 1 から ‘d’ で遷移する終点が 5 であることが分かる．

S8 で pos を 1 とし，次は節点 5 から $key[1]$ =‘e’ で遷移する節点を調べる．同様に S3 から始まる while 文を繰り返し，圧縮 DS 木上の節点を 1，5，6，7，2 と遷移し，S8 で $pos=4$ とする．ここで $B[2]<0$ より節点 2 は SP 節点であることが分かる．よって S9 で関数 StrCmp により，SP 節点 2 の BASE 値 $B[2]=-10$ が指す $T[10]$ から始まる “ult#” と $key[4]$ から始まる “ult#” とを比較し，等しいので探索成功として TRUE を返す．

次に “decode#” を探索する．上記と同様に，圧縮 DS 木上の節点を 1，5，6 と遷移し，$pos=2$ のときに始点 6 から $key[2]$ =‘c’ で遷移する終点を調べる．S4 で s に 6 を，S5 で t に $B[6]$+c= $1+3=4$ を設定する．S6 で $C[4]=1$ と $s=6$ が異なるので，始点 6 から ‘c’ で遷移する終点は存在せず，S7 で探索失敗として FALSE を返す．(例終了)

2.3 追加アルゴリズム

新規追加処理は，前述した探索失敗 (I)，(II) でそれぞれ異なった処理を行う．

(I) の場合，終点 $t=B[s]+label$ が未使用要素であれば，t に SP 節点を作成する．t が圧縮 DS 木の節点として使用されている要素（以下，使用済み要素）であれば，関数 TransNode を呼び出す．関数 TransNode では，すでに存在する s からの遷移と新規に作成する $label$ が可能な値に $B[s]$ を変更し，t とは異なる未使用要素 t' を $label$ による終点とする．ここで，式 (1) を満たすためにすでに存在していた s の終点を移動させ，式 (2) を満たすために移動した終点における終点の CHECK 値を更新する．関数 TransNode が終了後，t' に SP 節点を作成する．

(II) の場合，SP ストリングとキーを比較し，共通する表記記号を遷移としたシングル節点を作成する．その後，異なる 2 つの遷移が可能な BASE 値を持つ節点を作成し，それぞれの SP 節点を作成する．

関数 TransNode を**図 5** に示し，この関数で使用する記号を以下に示す．

$oldBase$	：変更前の BASE 値
S_L	：遷移集合
S_B	：終点の BASE 値集合
i	：S_L，S_B の要素番号
$size$	：S_L，S_B の要素数
old	：移動前の節点番号

```
Function:TransNode(s, label)
(TN1)   oldBase = B[s];
(TN2)   size =SearchTnode(s, S_L, S_B);
(TN3)   B[s] =NewBase(S_L ∪ label, size + 1);
(TN4)   i = 0;
(TN5)   while:i が size より小さい
(TN6)       t = B[s] + S_L[i];
(TN7)       InsNode(t, s);
(TN8)       B[t] = S_B[i];
(TN9)       if:節点 t が SP 節点ではない
(TN10)          old = oldBase + S_L[i];
(TN11)          RenewalCheck(old, t);
(TN12)      i = i + 1;
(TN13)  return TRUE;
```

図 5 関数：TransNode
Fig. 5 Function: TransNode.

```
Function:DelNode(t)
(DN1)   B[t] = 0;
(DN2)   C[t] = 0;
(DN3)   return t;
```

図 6 関数：DelNode
Fig. 6 Function: DelNode.

```
Function:NewBase(S_L, size)
(NB1)   base = 1;
(NB2)   while:
(NB3)       i = 0;
(NB4)       while:i が size より小さい
(NB5)           if:要素 (base + S_L[i]) が使用済み要素である
(NB6)               break;
(NB7)           i = i + 1;
(NB8)       if:i が size と等しい
(NB9)           return base;
(NB10)      base = base + 1;
(NB11)  return FALSE;
```

図 7 関数：NewBase
Fig. 7 Function: NewBase.

関数 TransNode は，図 5 中の TN2 で関数 SearchTnode（s，S_L，S_B）により，s からの遷移集合 S_L と終点の BASE 値集合 S_B を得る．関数 SearchTnode 内では，**図 6** に示す関数 DelNode を呼び出し，s の終点を削除する．また，戻り値は s の終点数であり，S_L と S_B の要素は遷移の内部表現値により昇順に設定されている．ここで，関数 DelNode(t) は節点 t を圧縮 DS 木上から削除する．

TN3 で**図 7** に示す関数 NewBase により，S_L と新規に作成する $label$ とが共存可能な値に $B[s]$ を変更する．これにともない TN6 から TN8 で，**図 8** に示す関数 InsNode により s の終点を再度作成する．ここで関数 NewBase は，図 7 中の NB4 から始まる while 文で S_L の全要素が BASE 値 $base$ によって遷移可能か判断し，遷移可能であれば NB9 で $base$ を返す．

Function:InsNode(t, s)
(IN1) $C[t] = s$;
(IN2) return t;

図 8 関数：InsNode
Fig. 8 Function: InsNode.

不可能であれば，NB10 で $base$ を変更し NB2 から始まる while 文を繰り返す．

TN11 で関数 RenewalCheck(x, y) により，ダブル配列上の要素番号が x から y へ移動した始点について，その終点の CHECK 値を x から y に再設定する．以下，変数 i を用いて S_L，S_B の要素を $S_L[i]$, $S_B[i]$ とする．

例3：図 3 に “decode#”を追加する．例 2 より，始点 6 から ‘c’ で遷移する終点は存在しなかった．また終点となる要素 4 は CHECK 値が初期値 0 ではないので，使用済み要素である．よって，関数 TransNode(6, ‘c’) を呼び出す．

TN1 で変更前の BASE 値 $oldBase$ に $B[6] = 1$ を退避させ，TN2 で関数 SearchTnode($6, S_L, S_B$) により，S_L に { ‘b’, ‘f’ } を，S_B に { −5, 1 } を，終点数 $size$ に 2 を設定する．

TN3 で関数 NewBase({ ‘b’, ‘c’, ‘f’ }, 2 + 1) を呼び出す．NB1 で BASE 値 $base$ に 1 を設定し，NB4 から始まる while 文で S_L の全要素が $base$ により遷移可能か調べる．まず $S_L[0]$ =‘b’ について，NB5 で終点 $base$+b= 3 は CHECK 値が初期値 0 であり未使用要素なので遷移可能である．次に $S_L[1]$ =‘c’ について，終点 $base$+c= 4 は CHECK 値が 1 であり使用済み要素である．よって，$base = 1$ では遷移できない．

次に，NB10 で $base = 2$ とする．この場合も S_L の全要素が遷移可能な BASE 値ではない．同様に $base$ を増加させ，$base = 5$ のときにすべての遷移が可能となる．よって，NB9 で関数 TransNode に戻り，TN3 で $B[6]$ に 5 を設定する．

これにともない，TN6 から TN8 で $t = B[6]$+b= 7, 関数 InsNode(7, 6)，$B[7] = S_B[0] = -5$ とし，始点 6 における BASE 値変更前の終点 3 を 7 へ移動する．ここで，$B[7] < 0$ であるから節点 7 は SP 節点である．よって TN9 から始まる if 文は実行しない．

同様に始点 6 の終点 7 を 11 へ移動する．ここで，$B[11] > 0$ であり，節点 11 は SP 節点ではないので終点を持つ．よって，TN10 と TN11 で節点 11 における終点の CHECK 値である $C[2]$ と $C[10]$ を 11 に再設定する．

以上で関数 TransNode を終了し，最後に要素 $B[6]$+$c = 8$ に SP 節点を作成する．追加処理終了後の圧縮 DS 木を**図 9** に，ダブル配列を**図 10** に示す.(例終了)

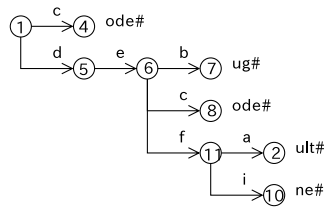

図 9 追加処理後の圧縮 DS 木
Fig. 9 Compression digital search tree after insertion processing.

	1	2	3	4	5	6	7	8	9	10	11	12
BASE	1	-10	0	-1	1	5	-5	-18	0	-15	1	0
CHECK	1	11	0	1	1	5	6	6	0	11	6	0

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
TAIL	o	d	e	#	u	g	#	g	#	u	l	t	#	#	n	e	#	o	d	e	#

図 10 追加処理後のダブル配列
Fig. 10 Double-array structure after insertion processing.

Function:Delete$(SPnode)$
(DE1) $s = C[SPnode]$;
(DE2) DelNode($SPnode$);
(DE3) t =IsSingleNode(s);
(DE4) **if:**節点 s がシングル節点であり, 節点 t が SP 節点である
(DE5) StrCpy($tail$[KSIZE], $T + (-B[t])$);
(DE6) $pos = 1$;
(DE7) **while:**IsSingleNode(s) ≠FALSE
(DE8) $tail$[KSIZE−pos] = $t - B[s]$;
(DE9) DelNode(t);
(DE10) $t = s$;
(DE11) $s = C[t]$;
(DE12) $pos = pos + 1$;
(DE13) $B[t] = -maxTail$;
(DE14) StrCpy($T[maxTail]$, $tail$ +(KSIZE−pos+1));
(DE15) return TRUE;

図 11 関数：Delete
Fig. 11 Function: Delete.

ダブル配列法の追加処理は，探索失敗 (I) の終点が使用済み要素である場合に多くの計算量を要する．

2.4 削除アルゴリズム

削除処理は，削除キーを探索して到達した SP 節点 $SPnode$ を削除することで実現する．これにともない，削除キーとは異なるキーに対する圧縮 DS 木上の遷移列が一意に決定する場合がある．これは，削除節点における始点の終点がただ 1 つとなる場合である．ここで圧縮 DS 木であることを保持するために，一意に決定する遷移列を抽出し，配列 TAIL に設定する．

上記を実現する関数 Delete を**図 11** に示し，この関数で使用する記号を以下に示す．

$SPnode$ ：削除キーに関する SP 節点番号
$tail$ ：一意に決定する遷移列
pos ：$tail$ の要素番号
$maxTail$ ：次に使用可能な配列 TAIL の要素番号

まず図 11 中の DE2 で，SP 節点 $SPnode$ を削除する．次に，DE3，DE4 で $SPnode$ における始点 s の終点 t が SP 節点 1 つだけかを判断する．これが偽である場合，他に削除する節点が存在しないので終了する．真である場合，s から t への遷移を SP ストリングとして配列 TAIL に設定する．よって，DE5 で t が指す SP ストリングを一意な遷移列 $tail$ に退避する．さらに DE8 で s から t への遷移を key に設定し，DE9 で t を削除する．

ここで，関数 StrCpy($x[p], y+q$) は文字列 $(y+q)$ を配列 x の p 番目から設定する．また，関数 IsSingleNode(s) は，始点 s がシングル節点であればその終点を，そうではなければ FALSE を返す．

次に，s の始点がシングル節点であるか判断する．これが真である場合，s の始点から s への遷移も一意に決定するので $tail$ に設定し，s を削除する．一意な遷移が存在しなくなるまで根方向へ遷移しながらこれらを繰り返す．

最後に，DE13，DE14 で $tail$ を SP ストリングとした SP 節点を作成する．ここで，$tail$ の要素数は最大キー長 KSIZE の 2 倍であり，$maxTail$ は次に使用可能な配列 TAIL の要素番号である．

例 4：図 3 から “default#” を削除する．例 2 より SP 節点 2 を得て関数 Delete(2) を呼び出す．

DE1 で s に $C[2]=7$ を設定し，DE2 で関数 DelNode(2) により節点 2 を削除する．DE3 で t に関数 IsSingleNode(7) の戻り値 10 を設定し，DE4 で節点 10 は，始点 7 における唯一の終点であり $B[10]<0$ より SP 節点だと分かる．よって，DE5 で $tail$ に SP 節点 10 が指す SP ストリング “ne#” を設定し，さらに，DE8 で始点 7 から終点 10 への遷移 $10-B[7]=9=$‘i’ を $tail$ に設定し “ine#” とする．その後，DE9 で終点 10 を削除し，DE11 と DE12 で t に 7 を，s に 6 を設定する．

DE7 に戻り，関数 IsSingleNode(6) により，始点 6 がシングル節点ではないことが分かる．よって，DE13 で $B[7]$ に $maxTail=18$ をマイナス値として設定し，DE14 で $tail=$“ine#” を $T[18]$ から設定することにより節点 7 を SP 節点とする．

以上で削除処理を終了する．処理後の圧縮 DS 木を**図 12** に，ダブル配列を**図 13** に示す．(例終了)

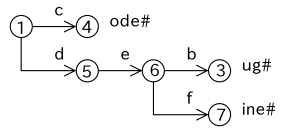

図 12 削除処理後の圧縮 DS 木
Fig. 12 Compression digital search tree after deletion processing.

	1	2	3	4	5	6	7	8	9	10	11	12
BASE	1	0	-5	-1	1	1	-18	0	0	0	0	0
CHECK	1	0	6	1	1	5	6	0	0	0	0	0

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
TAIL	o	d	e	#	u	g	#	g	#	u	l	t	#	#	n	e	#	i	n	e	#

図 13 削除処理後のダブル配列
Fig. 13 Double-array structure after deletion processing.

2.5 ダブル配列法の問題点

ダブル配列法の追加処理について，関数 NewBase の計算量はダブル配列の要素数と遷移種数に依存する．一般的にダブル配列法の要素数は非常に多く，関数 NewBase に依存する追加処理の計算量は多い[4),9)]．

3. 更新アルゴリズムの高速化

関数 NewBase では，BASE 値を増加させながら BASE 値を決定していたため，多くの計算量を必要としていた．しかし，任意の未使用要素 e が既知である場合，任意の遷移 a が可能な BASE 値 $base$ を式 (3) により決定できる．

$$base = e - a \tag{3}$$

そこで追加処理の高速化するため，未使用要素を管理し利用することを提案する．

3.1 双方向未使用要素リスト

関数 NewBase の計算量を削減するため，ダブル配列上の未使用要素をリストにより管理する．このとき，未使用要素リストから要素を削除する際に，その要素の直前未使用要素が指す次の要素を変更する．ここで，未使用要素リストが単方向である場合は直前未使用要素を探索する必要がある[9)]．

そこで式 (4) から (7) に示すとおり，未使用要素リストを双方向に拡張することを提案する[8)]．これにより，直前未使用要素を $O(1)$ で決定できる．式 (4) は先頭から終端方向への，式 (6) は終端から先頭方向へのリストを実現している．加えて，式 (5)，(7) により，未使用要素リストを循環リストとする．ここで，ダブル配列上の未使用要素数を m とし，未使用要素番号を出現順に e_1，e_2，$\cdots$，e_m とする．さらに，未使用要素の CHECK 値をマイナスとすることで未使用要素と使用済み要素とを区別する．また，ダブル配列の要素 $0(e_0)$ をダミーとして用いる．

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

図 14 双方向未使用要素リストを用いたダブル配列の例
Fig. 14 An example of double-array structure for the doubly list.

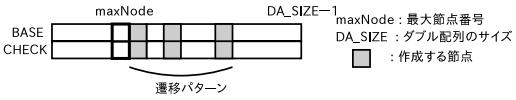

図 15 提案手法における節点作成の概要
Fig. 15 An illustration of node creation.

終端方向へのリスト

$$B[e_i] = e_{i+1} \qquad 0 \le i \le m-1 \qquad (4)$$

$$B[e_m] = e_1 \qquad (5)$$

先頭方向へのリスト

$$C[e_i] = -e_{i-1} \qquad 1 \le i \le m \qquad (6)$$

$$C[e_0] = -e_m \qquad (7)$$

例 5：キー集合 K に対する双方向未使用要素リストを用いたダブル配列を**図 14** に示す．

図 14 において，終端方向への未使用要素リストとして，要素 0 における次の未使用要素は $B[0] = 8$ により 8 となる．同様に要素 0 から 8，9，11，12，0 と循環リストとして未使用要素を連結している．

先頭方向への未使用要素リストとして，要素 0 における次の未使用要素は $C[0] = -12$ により 12 となる．同様に要素 0 から 12，11，9，8，0 と循環リストとして未使用要素を連結している．(例終了)

3.2 追加アルゴリズム

未使用要素を利用することにより，遷移パターンと未使用要素パターンの照合を行う関数 NewBase の計算量を削減する．すなわち，連続した未使用要素を利用して遷移パターンに関係なく BASE 値を決定する．また，遷移パターンに対して十分に未使用要素が連続していない場合においても，未使用要素リストにより関数 NewBase がダブル配列の要素数ではなく未使用要素数に依存する．

そこで，ダブル配列のサイズ DA_SIZE を設定し，**図 15** のようにダブル配列の後方に節点を作成する BASE 値を設定する．また，未使用要素リストを用いるのは節点番号が DA_SIZE を超える場合である．

拡張した関数 NewBase を関数 EX_NewBase とし，**図 16** に示す．拡張部分は，節点の作成位置を決定する図 16 中の ENB1 から ENB3 と，BASE 値を決定する ENB5 から始まる while 文である．

ダブル配列の後方へ節点を作成するため，ENB1,

Function:EX_NewBase$(S_L, size)$
(ENB1) $base = (maxNode + 1) - S_L[0]$;
(ENB2) **if:**$1 \le base <$DA_SIZE$-S_L[size-1]$
(ENB3) return $base$;
(ENB4) $e = B[0]$;
(ENB5) **while:**e がダミー要素ではない
(ENB6) $base = e - S_L[0]$;
(ENB7) **if:**$base$ が 1 より大きい
(ENB8) $i = 1$;
(ENB9) **while:**i が $size$ より小さい
(ENB10) **if:**要素 $S_L[i] + base$ が使用済み要素である
(ENB11) break;
(ENB12) $i = i + 1$;
(ENB13) **if:**i が $size$ と等しい
(ENB14) return $base$;
(ENB15) $e = B[e]$;
(ENB16) return FALSE;

図 16 関数：EX_NewBase
Fig. 16 Function: EX_NewBase.

Function:EX_InsNode(t, s)
(EIN1) $B[-C[t]] = B[t]$;
(EIN2) $C[B[t]] = C[t]$;
(EIN3) $C[t] = s$;
(EIN4) return t;

図 17 関数：EX_InsNode
Fig. 17 Function: EX_InsNode.

ENB2 で式 (3) を用いて BASE 値 $base$ を決定し，ENB3 で $base$ を返す．ここで，$maxNode$ は最大節点番号である．

節点番号が DA_SIZE を超える場合，ENB4 から ENB15 で未使用要素リストを用いて BASE 値を決定する．ENB6 で $S_L[0]$ が遷移可能である $base$ を式 (3) により決定する．$base$ で S_L のすべての要素が遷移可能でなければ，ENB15 で未使用要素リストから次の未使用要素を決定し ENB6 で $base$ を変更する．ここで，ENB8 が NB3 と異なっているのは，$S_L[0]$ が遷移可能であることが既知であるからである．

未使用要素リストを用いることにより，節点 t を作成する際に未使用要素リストから t を削除する．そこで，拡張した関数 InsNode を関数 EX_InsNode とし，**図 17** に示す．関数 InsNode では $B[t]$，$C[t]$ を初期化したが，関数 EX_InsNode では図 17 中の EIN1, EIN2 で未使用要素リストから t を削除している．

また 2 章で示した探索失敗 (II) の場合，SP ストリングを圧縮 DS 木上の遷移とし節点を作成するため，SP ストリングが短縮される．このとき，従来では配列 TAIL に短縮した SP ストリングを再設定していた[4]．この処理を高速化するため，配列 TAIL ではなく SP 節点の BASE 値を再設定する．

Function:EX_DelNode(t)
(EDN1) $prev = -C[0]$;
(EDN2) $B[prev] = t$;
(EDN3) $B[t] = 0$;
(EDN4) $C[0] = -t$;
(EDN5) $C[t] = -prev$;
(EDN6) return t;

図 18 関数：EX_DelNode
Fig. 18 Function: EX_DelNode.

3.3 削除アルゴリズム

未使用要素リストを用いることにより，節点 t を削除する際に未使用要素リストに t を追加する．ここで，要素番号について未使用要素リストを整列させる場合，削除要素の直前未使用要素を探索する必要がある[9]．そこで削除処理を高速化するため，未使用要素リストの最終要素として削除要素を追加する．

拡張した関数 DelNode を関数 EX_DelNode とし，**図 18** に示す．拡張部分は，図 18 中の EDN1 から EDN5 であり，EDN1 では t の前未使用要素となる $prev$ に未使用要素リスト上の最終要素 $-C[0]$ を設定し，EDN2 から EDN5 で未使用要素リストを更新する．

例 6：双方向未使用要素リストを用いたダブル配列に対して “decode#”を追加する．

まず，DA_SIZE= 17 である**図 19** について，例 3 同様，TN2 で関数 SearchTnode($6, S_L, S_B$) を呼び出す．この関数内で，始点 6 から ‘b’ で遷移する終点 3 を削除するために EX_DelNode(3) を呼び出す．

EDN1 で削除要素 3 を未使用要素リストの最終要素とするため，$prev$ に $-C[0] = 16$ を設定する．まず，終端方向へのリストとして要素 16 とダミー要素 0 との間に要素 3 を追加するため，EDN2 で $B[16]$ に 3 を，EDN3 で $B[3]$ に 0 を設定する．次に，先頭方向へのリストとしてダミー要素 0 と要素 16 との間に要素 3 を追加するため，EDN4 で $C[0]$ に -3 を，EDN5 で $C[3]$ に -16 を設定する．これらにより未使用要素リストに削除要素 3 を追加する．同様に，始点 6 から ‘f’ で遷移する終点 7 に対して EX_DelNode(7) を呼び出し，未使用要素リストに削除要素 7 を追加する．

TN3 で関数 EX_NewBase({ ‘b’, ‘c’, ‘f’ }, 3) を呼び出す．ここで $maxNode$ =10，$S_L[0]$ =‘b’ であるので，ENB1 で $base$ に $10 + 1 - b = 9$ を設定する．このとき，$base$ は ENB2 の条件を満たすので，関数 TransNode に戻り $B[6]$ に 9 を設定する．

$B[6]$ を変更したことにより，TN6 で t に $B[6]+b = 11$ を設定し，TN7 で関数 EX_InsNode(11, 6) を呼び出す．まず，終端方向へのリストとして要素 9 と要素 12 との間から要素 11 を削除するために，EIN1 で $B[9]$ に $B[11] = 12$ を設定する．次に，先頭方向へのリストとして要素 12 と要素 9 との間から要素 11 を削除するために，EIN2 で $C[12]$ に $C[11] = -9$ を設定する．これらにより未使用要素リストから要素 11 を削除する．EIN3 で $C[11]$ に 6 を設定し，始点 6 の終点として節点 11 を作成する．同様に関数 EX_InsNode(15, 6) を呼び出し，終点 15 を作成する．

以下，例 3 同様に処理し，最後に要素 $B[6]+c = 12$ に SP 節点を作成するために関数 EX_InsNode(12, 6) を呼び出す．処理後の圧縮 DS 木を**図 20** (a) に，ダブル配列を**図 21** (a) に示す．

次に，DA_SIZE= 13 である図 14 について，上記同様，TN3 で関数 EX_NewBase({ ‘b’, ‘c’, ‘f’ }, 3) を呼び出し，ENB1 で $base$ に 9 を設定する．このとき $base$ は ENB2 の条件を満たさないので，ENB4 で未使用要素 e に $B[0] = 8$ を，ENB6 で $base$ に $8 - b = 6$ を設定する．ENB9 から始まる while 文で，BASE 値 6 で S_L の全要素が遷移可能であると分かるので，ENB14 で関数 TransNode に戻り $B[6]$ に 6

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

図 19 DA_SIZE が 17 である双方向未使用要素リストを用いたダブル配列の例
Fig. 19 An example of double-array structure for the doubly list (DA_SIZE: 17).

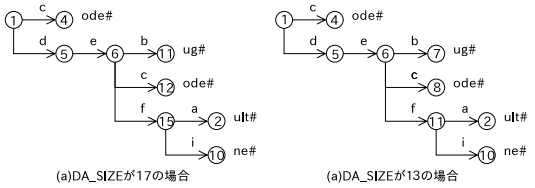

図 20 追加処理後の圧縮 DS 木
Fig. 20 Compression digital search tree for the doubly list after insertion processing.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
BASE	8	1	-11	7	-1	1	9	0	9	13	-15	-7	-18	14	16	1	3
CHECK	-7	1	15	-16	1	1	5	-3	0	-8	15	6	6	-9	-13	6	-14

(a)DA_SIZE が 17 の場合

	0	1	2	3	4	5	6	7	8	9	10	11	12
BASE	9	1	-11	0	-1	1	5	-7	-18	12	-15	1	3
CHECK	-3	1	11	-12	1	1	5	6	6	0	11	6	-9

(b)DA_SIZE が 13 の場合

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
TAIL	o	d	e	#	e	b	u	g	#	a	u	l	t	#	n	e	#	o	d	e	#

図 21 追加処理後の双方向未使用要素リストを用いたダブル配列
Fig. 21 A double-array structure for the doubly list after insertion processing.

を設定する．

以下，例 3 同様に処理する．処理後の圧縮 DS 木を図 20 (b) に，ダブル配列を図 21 (b) に示す．

ここで図 10 と図 21 とを比較すると，$B[2]$ と $T[5]$ から $T[8]$ が異なる．これは，2 章で示した探索失敗が (Ⅱ) の場合に，配列 TAIL ではなく，SP 節点の BASE 値を再設定したからである．(例終了)

4. 評　　価

提案手法の有効性を示すため，ダブル配列法を対象手法 A[4]，森田らが提案した単方向未使用要素リストを用いた手法を対象手法 B[9]，大野らが提案した双方向未使用要素リストを用いた手法を対象手法 C[6]，ダブル配列法とは異なる辞書構築手法である Suffix array を対象手法 D[2] とし，探索，追加，削除処理に対して比較評価を行う．

ここで，ダブル配列上の要素数を n，未使用要素数を m とし，キー数を a，キー長を k，キーの表記記号数を t とする．

4.1 理論的評価

ダブル配列法は $O(1)$ で遷移可能であるので，探索における最悪時間計算量は $O(k)$ となる[4]．ただし，対象手法 A と提案手法は圧縮 DS 木であり，S9 で SP ストリングに対して単純な文字列照合を行う．一方，対象手法 B，C は DS 木であるので，遷移を確認する S4 から S6 を行う．また対象手法 D は，要素数 $k \cdot a$ の Suffix array に対して二分探索を行う．よって，$O(log_2 k \cdot a)$ である[2]．

追加処理は関数 TransNode に依存する[4),9)]．よって，関数 TransNode を構成する関数 SearchTnode，NewBase，InsNode，RenewalCheck と，関数 SearchTnode 内で呼び出される関数 DelNode の評価を行う．

関数 EX_DelNode の最悪時間計算量は，EDN1 から ED5 は $O(1)$ であり，関数 DelNode の DN1，DN2 も $O(1)$ であるので，それぞれ $O(1)$ となる．また，対象手法 B では，未使用要素リストをたどり直前未使用要素を探索するので $O(m)$ となる[9]．

関数 SearchTnode の最悪時間計算量は，全終点候補を調べ関数 DelNode を呼び出すので，対象手法 B は $O(m \cdot t)$ となり，他の手法は $O(t)$ となる[9]．

関数 EX_NewBase は ENB1 から ENB3 で BASE 値が決定した場合は $O(1)$ である．そうではない場合，ENB5 から始まる while 文により未使用要素リストをたどり，ENB9 から始まる while 文によりすべての遷移が可能か調べるので $O(t \cdot m)$ となる．また，関数 NewBase は NB2 から始まる while 文，NB4 から始まる while 文により $O(t \cdot n)$ となる．よって，提案手法は $O(t \cdot m)$，対象手法 A は $O(t \cdot n)$ となり，対象手法 B，C は，ENB5 から始まる while 文と同等であるので $O(t \cdot m)$ となる[6),9)]．

関数 EX_InsNode は EIN1 から EIN3 は $O(1)$，関数 InsNode も $O(1)$ である．また，対象手法 B における関数 InsNode では，未使用要素リストをたどり直前未使用要素を探索するので $O(m)$ となる[9]．

関数 RenewalCheck は，すべての終点候補を調べるので $O(t)$ となる．

関数 TransNode は，TN2，TN3 で関数 SearchTnode，NewBase を呼び出し，すべての終点について処理する TN5 から始まる while 文で関数 InsNode，RenewalCheck を呼び出す．よって，提案手法は $O(t + m \cdot t + t^2)$，対象手法 A は $O(t + n \cdot t + t^2)$，対象手法 B は $O(m \cdot t + m \cdot t + t \cdot (t + m))$，対象手法 C は $O(t + m \cdot t + t^2)$ となる．

対象手法 A 以外は m に依存するが，m が大きいとダブル配列がスパースとなり，関数 NewBase の計算回数が減少する．提案手法は対象手法 C と最悪時間計算量は等しいが，DA_SIZE が十分に大きい間は高速な追加処理が可能である．

また，対象手法 D における Suffix array の構築は基数ソートに依存するので $O(k \cdot a)$ である[2]．

削除処理である関数 Delete は，関数 DelNode，IsSingleNode に依存する[9]．関数 IsSingleNode は，すべての終点候補を調べるので $O(t)$ となる．

よって関数 Delete は，最悪 k 回処理する DE7 から始まる while 文内で関数 IsSingleNode，DelNode を呼び出すので，対象手法 B は $O(m + t + k \cdot (t + m))$，他の手法は $O(t + k \cdot t)$ となる．

4.2 実験による評価

対象手法 A は約 700 行，対象手法 B，C は約 800 行，対象手法 D は約 500 行，提案手法は約 800 行の C 言語で実装し，Intel Pentium Ⅳ 2.8 GHz，Fedora core 4 上で稼動している．

実験では，英語キー集合として英単語辞書約 40 万語[3]，日本語キー集合として日本語形態素辞書約 40 万語[1),7)]，URI キー集合としてランダムに抽出した 100 万件から，それぞれ 20 万語をランダムに抽出したものを用いた．また部分キー集合として，各キー集合より 1 万語から 20 万語まで 1 万語ずつランダムに抽出したものを用いた．加えて動的キー集合として，各キー集合より重複を含む 1 万語から 20 万語まで 1 万語ずつランダムに抽出したものを用いた．また対象手法 D に対するキー集合は，上記のキー集合に存在し

表 1 キー集合
Table 1 The set of key in 200,000 words.

	英語キー集合	URI キー集合	日本語キー集合
キー数	200,000	200,000	200,000
平均キー長 (byte)	9.46	57.46	7.16
平均遷移数			
対象手法 A，提案手法	2.13	1.26	2.63
対象手法 B，対象手法 C	1.30	1.04	1.33

表 2 ダブル配列の状況に対する実験結果
Table 2 The simulation results of a double-array in 200,000 words.

	英語キー集合	URI キー集合	日本語キー集合
節点数			
対象手法 A，提案手法	377,044	957,567	322,823
対象手法 B，対象手法 C	856,468	4,457,606	812,801
シングル節点数			
対象手法 A，提案手法	270,288	843,584	241,889
対象手法 B，対象手法 C	749,712	4,343,623	731,867
未使用要素数			
対象手法 A	529	47	27,218
対象手法 B	78	0	150
対象手法 C	127,133	33,370	269,328
提案手法	132,713	92,208	277,146
使用 byte 数			
対象手法 A	4,113,563	14,361,654	3,745,136
対象手法 B	6,852,368	35,660,848	6,503,608
対象手法 C	7,868,808	35,927,808	8,657,032
対象手法 D	10,457,535	58,457,630	8,156,750
提案手法	5,171,019	15,098,942	5,744,560

ないコードを区切り文字とし，すべてのキーを連結したテキストデータを用いた．

表 1 に，キー集合の特徴として，平均キー長，追加処理後の DS 木および圧縮 DS 木における内部節点を対象として算出した平均遷移数を示す，ただし，日本語キー集合における平均キー長は日本語 1 文字に対して 2 byte とする．

表 2 に，キー集合を追加後のダブル配列における節点数，シングル節点数，未使用要素数を，加えて実験で使用したダブル配列および配列 TAIL を，対象手法 D に対してはインデックス部分である Suffix array とテキスト部分を対象とした byte 数を示す．ここで，DA_SIZE は圧縮 DS 木が構築可能な最小値とし，以下の実験でも同様とする．

探索処理について，キー集合を追加後，同一キー集合を探索した処理時間を**表 3** に示す．また，英語部分キー集合に対する対象手法 A および提案手法，対象手法 B および C，対象手法 D の探索時間を**図 22** に示す．

表 3，図 22 より，対象手法 A，提案手法は DS 木を実現した対象手法 B，C よりも各キー集合に対して約 1.1 倍高速であった．また対象手法 D は，キー数が少ない場合は他手法よりも高速であるが，キー数が増加すると他手法よりも低速となった．これは，対象手法 D の探索処理はキー数に依存する二分探索によるもので，DS 木および圧縮 DS 木の探索処理はキー長に依存していることが要因である．

表 3 キー集合追加後における探索時間に対する実験結果
Table 3 The simulation results of search time.

キー数	手法	英語キー集合	URI キー集合	日本語キー集合
1 万	対象手法 A，提案手法	0.03	0.11	0.03
	対象手法 B，C	0.04	0.16	0.04
	対象手法 D	0.01	0.02	0.01
10 万	対象手法 A，提案手法	0.12	0.34	0.09
	対象手法 B，C	0.13	0.38	0.12
	対象手法 D	0.24	0.39	0.20
20 万	対象手法 A，提案手法	0.15	0.40	0.13
	対象手法 B，C	0.17	0.45	0.14
	対象手法 D	0.58	0.93	0.49

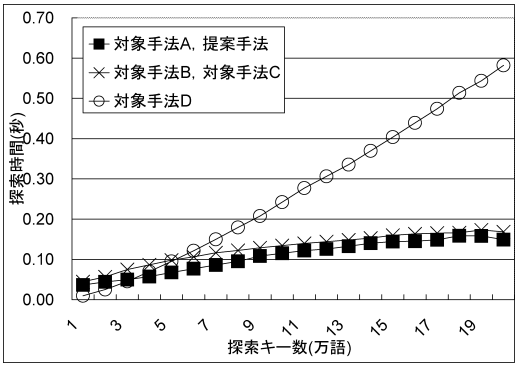

図 22 キー集合追加後における探索時間に対する実験結果
Fig. 22 The simulation results of search time.

表 4 追加時間および計算回数に対する実験結果
Table 4 The simulation results of insertion time and number of comparison in 200,000 words.

	英語キー集合	URI キー集合	日本語キー集合
追加時間 (秒)			
対象手法 A	271.66	2,750.26	167.50
対象手法 B	1.01	1.32	2.78
対象手法 C	0.90	1.28	2.30
対象手法 D	4.19	34.72	4.04
提案手法	0.50	0.70	1.59
NewBase の計算回数 (千回)			
対象手法 A	38,831,625	192,207,672	22,937,940
対象手法 B	1,739	4,638	8,195
対象手法 C	1,090	4,544	2,423
提案手法	7,984	2,552	16,325

追加処理について，まず，提案手法と各対象手法との比較を行う．**表 4** に，キー集合を追加した処理時間と関数 NewBase の計算回数を，対象手法 D は追加時間のみを示す．また，英語部分キー集合に対する対象手法 B，C，提案手法の追加時間と関数 NewBase の計算回数を**図 23** に示す．

表 4，図 23 より，英語キー集合における提案手法の追加処理速度は，対象手法 A より約 539.8 倍，対象手法 B より約 2.0 倍，対象手法 C より約 1.8 倍，対象手法 D より約 8.4 倍高速であった．対象手法 A に

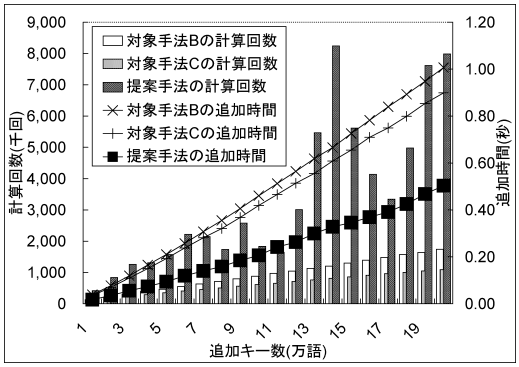

図 23 追加キー数を変化させた追加時間および計算回数に対する実験結果
Fig. 23 The simulation results of insertion time and number of comparison.

対する高速化の要因は，関数 NewBase の計算回数を約 0.02%と大幅に削減したことである．しかし，DS 木を実現した対象手法 B，C に対してはそれぞれ約 458.9%，約 732.5%と多くなっており，高速となった要因ではない．両対象手法に対しては，表 2 より作成する節点が多いことが要因であり，さらに対象手法 B に対しては節点削除時の直前未使用要素探索も要因の 1 つである．ここで両対象手法における関数 NewBase の計算回数が少ないのは，節点の約 90%をシングル節点が占め，未使用要素をなくしたからである．

また対象手法 A に対して，提案手法は URI キー集合においては約 3,928.9 倍，日本語キー集合においては約 105.3 倍高速となった．表 2 より，英語キー集合に対して，URI キー集合における処理速度が大きく向上したのは未使用要素が多いからである，一方，日本語キー集合における処理速度の向上が小さいのは遷移数が多いからである．

よって，関数 EX_NewBase が未使用要素数に依存していることが分かる．

次に，提案手法における追加処理の性能は，4.1 節から分かるように DA_SIZE である n が関係している．そこで，DA_SIZE による提案手法における追加処理の性能比較を行うため，DA_SIZE を圧縮 DS 木が構築可能な最小値から，関数 EX_NewBase の計算回数が一定となる最大値まで増加させて追加時間および関数 EX_NewBase の計算回数を計測した．**表 5** に，DA_SIZE の最小値，中央値，最大値に対する実験結果を示す．また**図 24** に，英語キー集合に対する追加時間および関数 EX_NewBase の計算回数を示す．英語キー集合に対しては，DA_SIZE を最小値 50 万要素から最大値 400 万要素まで 10 万要素ずつ増加させた．

表 5，図 24 より，英語キー集合における提案手法

表 5 DA_SIZE の変化に対する追加時間および計算回数に対する実験結果
Table 5 The simulation results of insertion time and number of comparison.

		英語キー集合	URI キー集合	日本語キー集合
最小値	DA_SIZE(万要素)	50	105	60
	追加時間 (秒)	0.50	0.70	1.59
	計算回数 (千回)	7,984	2,552	16,325
中央値	DA_SIZE(万要素)	230	160	370
	追加時間 (秒)	0.34	0.65	0.62
	計算回数 (千回)	4,990	2,027	3,707
最大値	DA_SIZE(万要素)	400	210	680
	追加時間 (秒)	0.28	0.63	0.30
	計算回数 (千回)	403	810	134

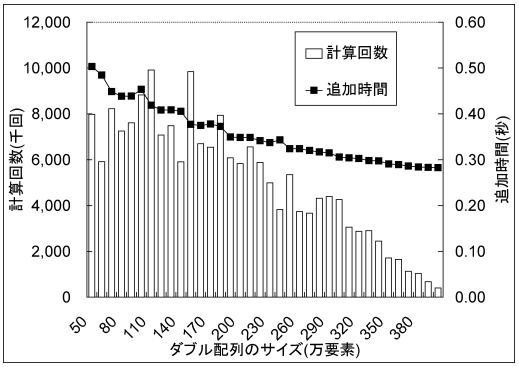

図 24 DA_SIZE の変化に対する追加時間および計算回数に対する実験結果
Fig. 24 The simulation results of insertion time and number of comparison.

の追加処理速度について，DA_SIZE が最小値の場合より最大値の場合が約 1.8 倍高速であった．これは関数 EX_NewBase の計算回数が減少していることが要因である．

他のキー集合に対しても同様に，DA_SIZE が増加するとともに追加処理速度が高速となった．また，4.1 節から分かるように，各対象手法は DA_SIZE によって追加処理の性能は変化しない．よって，DA_SIZE に最小値を設定した提案手法と各対象手法との実験結果である表 4 と表 5 より，DA_SIZE にかかわらず提案手法が各対象手法よりも高速であることが分かる．

削除処理について，キー集合を追加後のダブル配列および Suffix array に対して，同一キー集合を削除した処理時間を**表 6** に示す．また，英語部分キー集合に対する対象手法 A，C，提案手法の削除時間を**図 25** に示す．ここで対象手法 D の削除処理は，削除キーの探索によって該当する Suffix array の削除要素を決定し，以降の要素を前方へ移動させるものとする．

表 6，図 25 より，英語キー集合における提案手法の削除処理速度は，対象手法 A の約 0.99 倍と同等であり，対象手法 B より約 2,477.2 倍，対象手法 C よ

表 6 キー集合追加後における削除時間に対する実験結果
Table 6 The simulation results of deletion time in 200,000 words.

	英語キー集合	URI キー集合	日本語キー集合
対象手法 A	0.63	1.47	0.59
対象手法 B	1,554.70	50,078.55	1,431.17
対象手法 C	1.00	4.68	0.96
対象手法 D	1,281.67	6,593.45	1,061.17
提案手法	0.64	1.50	0.61

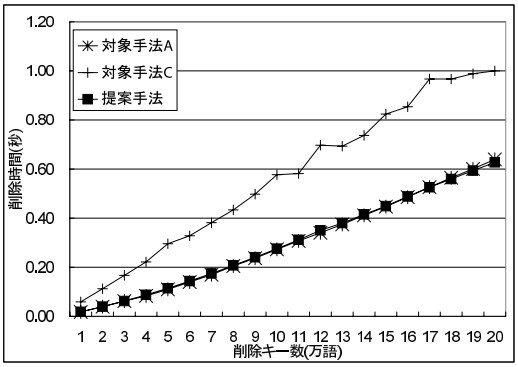

図 25 削除キー数を変化させた削除時間に対する実験結果
Fig. 25 The simulation results of deletion time.

表 7 動的更新時間に対する実験結果
Table 7 The simulation results of dynamic updation time in 200,000 words.

	英語キー集合	URI キー集合	日本語キー集合
追加キー数	118,708	101,193	100,354
削除キー数	81,292	98,807	99,646
動的更新時間 (秒)			
対象手法 A	133.057	849.891	46.010
対象手法 B	2.053	5.575	39.411
対象手法 C	1.480	3.318	1.804
対象手法 D	643.475	3,355.107	543.754
提案手法	1.129	1.707	1.301

り約 1.6 倍，対象手法 D より約 2,002.6 倍高速であった．DS 木を実現した対象手法 B，C より高速となった要因は，表 2 より削除する節点が多いことである．対象手法 B に対しては，節点削除時の直前未使用要素探索が大きな要因となっている．よって提案手法は，未使用要素を双方向に連結し管理することにより削除時間を抑制することができた．

動的更新実験について，部分キー集合 10 万語を追加した後，動的キー集合 20 万語に対して，格納済みのキーであれば削除処理を，そうではなければ追加処理を行った合計処理時間を**表 7** に示す．加えて，追加，削除したキー数を示す．また，英語キー集合における動的更新実験について，動的キー集合に対する対象手法 B，C，提案手法の更新時間を**図 26** に示す．ここで，対象手法 D の追加アルゴリズムはキー 1 語ごとに追加するものではない．そこで動的更新実験における対象手法 D の追加処理は，追加キーを探索するこ

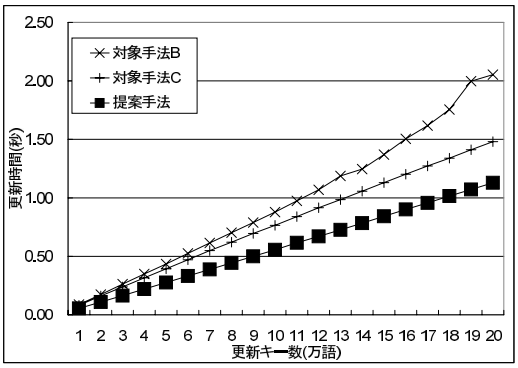

図 26 更新キー数を変化させた動的更新時間に対する実験結果
Fig. 26 The simulation results of dynamic updation time.

とで適切な Suffix array 上の追加位置を決定し，その位置以降の要素を後方へ移動させるものとする．

表 7，図 26 より，動的更新に対して，提案手法は対象手法 A の約 117.9 倍，対象手法 B の約 1.8 倍，対象手法 C の約 1.3 倍，対象手法 D の約 570.0 倍高速となった．ここで，対象手法 B が削除処理実験ほど低速にならなかったのは，追加処理により未使用要素が増加しなかったからである．

以上より，DS 木および圧縮 DS 木におけるダブル配列法や Suffix array に対して，提案手法は動的更新アルゴリズムとして有効であるといえる．

5. おわりに

本論文では圧縮 DS 木におけるダブル配列を動的検索法として確立するため，追加処理を高速化する手法とともに削除時間を抑制する手法を提案し，実験により有効性を示した．今後の課題として，内部表現値の最適化を行いダブル配列内のスパースを減少させること，ダブル配列法に対して高速なガーベージコレクション手法を考案することがあげられる．

参考文献

1) (財) 新世代コンピュータ技術開発機構：ICOT 形態素辞書．http://ftp.icot.or.jp/
2) Juha, K. and Peter, S.: Simple Linear Work Suffix Array Construction, *ICALP 2003*, LNCS 2719, pp.943–955 (2003).
3) Atkinson, K.: Ispell English Word Lists. http://wordlist.sourceforge.net/
4) 青江順一：自然言語辞書の検索—ダブル配列による高速ディジタル検索アルゴリズム，*bit*, Vol.21, No.6, pp.36–44 (1989).
5) 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).

6) 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処理辞書の高速更新手法，言語処理学会第 11 回年次大会予稿集，pp.745–748 (2005).
7) 情報処理振興事業協会技術センター：IPA 日本語辞書．http://www.ipa.go.jp/
8) 中村康正，望月久稔：自然言語処理における効果的な辞書情報更新アルゴリズム，情報処理学会研究報告，FI-80/NL-169, pp117–122 (2005).
9) 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，Vol42, No.9, pp.2229–2238 (2001).

(平成 18 年 3 月 20 日受付)
(平成 18 年 7 月 15 日採録)

（担当編集委員　高須 淳宏）

中村　康正（学生会員）

昭和 58 年生．平成 17 年大阪教育大学教育学部教養学科情報科学専攻卒業．現在同大学大学院修士課程在学中．自然言語処理に関する研究に従事．

望月　久稔（正会員）

昭和 44 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）．同年大阪府立工業高等専門学校電子情報工学科講師．平成 15 年大阪教育大学教育学部教養学科情報科学講座講師，現在に至る．情報検索，自然言語処理，知識表現の研究に従事．電子情報通信学会，人工知能学会，自然言語処理学会各会員．