社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2005－FI－80 (17)
2005－NL－169 (17)
2005／9／30

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

中村康正[†] 望月久稔[†]

トライ法は，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．このトライ法のデータ構造として，青江らが提案したダブル配列法がある．ダブル配列法は高速性とコンパクト性をあわせもっており有効なデータ構造であるが，動的検索法に比べデータの更新処理が高速であるとはいえない．そこで現在では未使用要素を単方向リストとして連結する手法が知られているが，トライ木の節点を追加および削除する際に大きなコストを必要とする．そこで本論文では，未使用要素を双方向リストとして連結することにより追加処理を高速化し，さらに削除時間を抑えるアルゴリズムを提案する．10 万語の辞書データに対する実験を行った結果，追加速度は単方向リストよりも約 1.5 倍，削除時間は未使用要素リストを用いない従来法と同等となることが判った．

Effective Update Algorithms of Dictionary Information in Natural Language Processing

YASUMASA NAKAMURA[†] and HISATOSHI MOCHIZUKI[†]

A trie is used widely, such as dictionary information construction of natural language processing system. As a data structure of trie, there is the double-array structure which Aoe and others proposed. A double-array structure is an efficient data structure combining fast access with compactness. However, the updating processing is not faster than other dynamic retrieval methods. Then, although the technique of connecting empty elements as linked list is known now, big cost is needed in the node of a trie tree is inserted and deleted. In this paper, we presents a fast insertion algorithm by connecting empty elements as doubly list and reduction algorithm of deletion time. From the simulation results for 100 thousands keys, it turned out that the presented method for insertion is about 1.5 times faster than the linked list method, and deletion time is equivalent to original method which is not used linked list.

1. はじめに

トライ法は，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もつ木構造で表現される．そのため，トライ法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．

トライ法のデータ構造として，配列を用いた手法とリストを用いた手法がある[4]．前者は，配列の特性から遷移を $O(1)$ でたどることができるが，大きな空間を必要とする．また後者は，不必要な遷移情報をもつ必要がないので小さな空間ですむが，遷移を $O(1)$ でたどることができない．これらを解決するために，ダブル配列法がある[3][4]．ダブル配列法は，配列の高速性とリストのコンパクト性をあわせもつが，動的検索法に比べてキーの追加処理が高速であるとはいえない．そこで現在では，未使用要素を単方向リストとして連結する手法がある[6][8]．

本論文では，ダブル配列を動的検索法として確立するため，未使用要素を双方向リストとして連結することにより，追加処理を高速化する手法を提案する．また，未使用要素をリストとして連結することによる削除時間増加に対する解決手法を提案する．

† 大阪教育大学
Osaka Kyoiku University

以下，2 節では，ダブル配列とその問題点を述べる．3 節では，単方向リストを用いたダブル配列とその問題点を述べる．4 節では，追加処理の高速化手法と削除時間増加に対する抑制手法を提案する．5 節では，これらの提案手法に対して実験による評価を与え，6 節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

2.1 ダブル配列法のデータ構造

ダブル配列のデータ構造は，3 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，トライの始点 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[2][3][4]．以下，節点 $node$ に関する BASE 値を $B[node]$，CHECK 値を $C[node]$ とする．

$$B[s] + 'a' = t \quad (1)$$

$$s = C[t] \quad (2)$$

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 はトライの親子関係を一意に決定するために用いる．また，配列 TAIL には，各キーにおける他のキーと共有していない遷移を文字列として格納し，トライ上の節点を抑制する[3]．以下，この手法を normal 法と呼ぶ．

キー集合 K ={trie, do, doing, search} に対する，トライと normal 法によるダブル配列をそれぞれ図 1，図

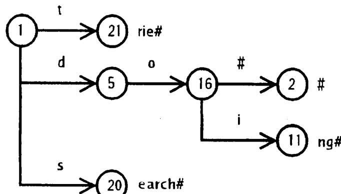

図 1　トライの例
Fig. 1　An example of trie structure.

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

	13	14	15	16	17	18	19	20	21	22
BASE	0	0	0	2	0	0	0	-10	-1	0
CHECK	0	0	0	5	0	0	0	1	1	0

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
TAIL	r	i	e	#	#	#	n	g	#	e	a	r	c	h	#	#

図 2　normal 法によるダブル配列の例
Fig. 2　An example of double-array structure of the normal method.

2 に示す．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，配列 BASE および配列 CHECK の初期値を 0，配列 TAIL の初期値を終端記号'#' とする．

以下，図 1 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．葉は，配列 TAIL への写像位置を BASE 値に保持し，探索対象となる葉以下の文字列を一意に決定する．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他の節点と区別する．SP 節点が指す配列 TAIL に格納されている文字列を SP ストリングと呼び，図 1 では SP 節点の右側に示す[4]．

2.2　normal 法の追加アルゴリズム

ダブル配列における探索成功条件は，式 (1)，(2) を満足させながらトライ上を遷移し，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[4]．新規追加処理は以下に示す探索失敗時に行われる．

- 節点 *node* から *label* で遷移する終点が存在しない．
- SP ストリングとキーが異なる．

前者は，終点が未使用であれば，終点となる要素 $B[node]+label$ に SP 節点を作成する．終点が使用済みであれば，関数 TransNode を呼び出し，後で述べる関数 NewBase により，すでに存在する *node* からの遷移と *label* による遷移とが可能である値に $B[node]$ を変更する．BASE 値の変更にともない *node* の子節点を移動し，式 (2) を満足させるため，移動した節点が保持する子節点の CHECK 値を変更する．その後，キーに対する SP 節点を作成する．後者は，SP ストリングとキーを比較し，共通する文字を遷移とした節点をダブル配列上に作成する．その後，それぞれの SP 節点を作成するため，関数 NewBase により異なる二つの遷移が可能である BASE 値を算出する．

関数 TransNode とともに，この関数に呼び出される関数を以下に示す．

関数 GetLabel(*node*, *L_SET*)**:**　節点 *node* を始点とする遷移すべてを遷移集合 *L_SET* にセットする．

関数 GetBase(*parent*, *L_SET*, *B_SET*)**:**　遷移集合 *L_SET* から節点 *parent* を親とする節点 *child* を算出し，$B[child]$ を BASE 値集合 *B_SET* にセットする．さらに関数 DelNode(*child*) を呼び出す．

関数 InsNode(*node*, *parent*)**:**　$C[node]$ に *parent* をセットし，節点 *node* を作成する．

関数 DelNode(*node*)**:**　$B[node]$，$C[node]$ を初期化し，節点 *node* を削除する．

関数 NewBase は，遷移集合 *L_SET* の全要素が遷移できる BASE 値を算出する．よって，*L_SET* の要素数が増加することで計算量が増加する．ここで，遷移集合 *L_SET* の要素を $L_SET[i]$ で表し，内部表現値により昇順に格納されているものとする．また，変数 i は *L_SET*，*B_SET* の要素番号とする．

関数 NewBase(*L_SET*)

手順 1(NB-1):初期化

BASE 値候補を表す変数 *base* に 1 をセットする．

手順 2(NB-2):BASE 値の算出

すべての $L_SET[i]$ について，$C[base+L_SET[i]]=0$ を満足する場合，*base* を返して終了．そうでなければ，*base* をインクリメントし手順 2 へ．

関数 TransNode(*node*, *label*)

手順 1(TN-1):BASE 値の設定

BASE 値待避変数 *oldBase* に $B[node]$ をセットし，関数 GetLabel(*node*)，関数 GetBase(*node*, *L_SET*, *B_SET*) を呼び出す．関数 NewBase($L_SET \cup label$) により，*L_SET* と *label* のすべてが遷移可能である BASE 値を $B[node]$ にセットする．

手順 2(TN-2):節点の移動

L_SET における全要素に対して以下を行い終了．節点 *node* から $L_SET[i]$ で遷移する節点 *next* に $B[node]+L_SET[i]$ をセットする．関数 InsNode(*next*, *node*) を呼び出し，$B[next]$ に $B_SET[i]$ をセットする．このとき $B[next]>0$ であれば節点 *next* は子節点を保持するので，手順 3 へ．

手順 3(TN-3):子節点における遷移確立

変数 *oldNode* に $oldBase+L_SET[i]$ をセットし，$C[child]=oldNode$ となるすべての子節点 *child* の CHECK 値を *next* に再定義する．

2.3　normal 法の削除アルゴリズム

削除処理は，探索成功時に SP 節点を入力とする関数 Delete を呼び出すことで実現する．

関数 Delete では，まず入力である SP 節点 *node* を削除する．その後，節点 *node* の親節点であった *parent* が保持する子節点が SP 節点 1 つだけであるかどうかを判断する．上記が偽である場合，他に削除する節点が存在しないので終了する．上記が真であることは，その SP 節点よりも根

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

	13	14	15	16	17	18	19	20	21	22
BASE	0	0	0	2	0	0	0	-10	-1	0
CHECK	-14	-15	-17	5	-18	-19	-22	1	1	-3

図 3 list 法によるダブル配列の例
Fig. 3 An example of double-array structure of the list method.

に近い遷移が SP ストリングとして配列 TAIL に写像できることを意味している．よって，この遷移を配列 TAIL に写像し，SP 節点を削除する．次に *parent* の親節点がシングル節点であるかどうか判断する．上記が真である場合，*parent* の親節点から *parent* への遷移も SP ストリングへ写像できるので，*parent* を削除する．配列 TAIL に写像できる遷移が存在しなくなるまでこれらを繰り返す．最後に，上記によって更新した SP ストリングを指す SP 節点を作成する．節点削除には，関数 DelNode を用いる．

2.4 normal 法の問題点

ダブル配列の要素数を n とすると，normal 法における関数 NewBase の計算量は $O(n)$ となる．一般的に n は非常に多く，関数 NewBase に依存している追加処理の計算量は多い[3]．これは，動的検索法として重大な問題となる．

3. 単方向未使用要素リストを用いたダブル配列法

ここでは，normal 法の問題点を解消する単方向未使用要素リストを用いた追加アルゴリズム[5][8] を説明し，この手法の問題点を提示する．以下，この手法を list 法と呼ぶ．

3.1 未使用要素リスト法

式 (3)，(4) に示す通り，未使用要素を単方向リストとして連結する．ここで，未使用要素数を m とし，ダブル配列における未使用要素の要素番号を昇順に $e_1, e_2, \cdots, e_m$ とする．また，未使用要素の CHECK 値をマイナスとすることで使用済み要素と区別する．

$$C[e_i] = -e_{i+1} \quad 1 \le i \le m-1 \tag{3}$$

$$C[e_m] = -e_1 \tag{4}$$

キー集合 K に対する list 法によるダブル配列を図 3 に示す．

3.2 list 法のアルゴリズム

上記の拡張により，節点を追加および削除する際にリストを再連結する処理を必要とする．そこで，関数 InsNode，関数 DelNode を拡張する．拡張後の関数をそれぞれ関数 L_InsNode，関数 L_DelNode とし，これらに呼び出される関数 L_PreNode とともに以下に示す．関数 L_PreNode は，入力節点 *node* の直前未使用要素を探索する．ここで，変数 *eTail* は未使用要素リストの最終要素 e_m をもつ．

関数 L_PreNode(*node*)

手順 1(LPN-1):初期化

未使用要素を表す変数 e に $-C[eTail]$ をセットする．

手順 2(LPN-2):直前未使用要素の探索

$-C[e] > node$ であれば探索成功として，e を返して終了．そうでなければ e に $-C[e]$ をセットし，手順 2 を繰り返す．

関数 L_InsNode(*node*, *parent*)

手順 1(LIN-1):前未使用要素の探索

関数 L_PreNode(*node*) を呼び出し，その戻り値を変数 *pre* にセットする．

手順 2(LIN-2):未使用要素リストからの削除

$C[pre]$ に $C[node]$ をセットし，未使用要素リストを連結しなおす．

手順 3(LIN-3):節点作成

$C[node]$ に *parent* をセットする．

関数 L_DelNode(*node*)

手順 1(LDN-1):前未使用要素の探索

関数 L_PreNode(*node*) を呼び出し，その戻り値を変数 *pre* にセットする．

手順 2(LDN-2):未使用要素リストへの追加

$B[node]$ に 0 をセットし，*node* を初期化する．$C[node]$ に $C[pre]$ を，$C[pre]$ に $-node$ をセットし，未使用要素リストを連結しなおす．

単方向未使用要素リストにより拡張した関数 NewBase を関数 L_NewBase として以下に示す．ここで，$L_SET[1]$ は L_SET の最小要素である．

関数 L_NewBase(*L_SET*)

手順 1(LNB-1):初期化

未使用要素を表す変数 e に $-C[eTail]$ をセットする．

手順 2(LNB-2):BASE 値の設定

BASE 値候補を表す変数 *base* に $e - L_SET[1]$ をセットする．

手順 3(LNB-3):BASE 値の算出

$base \ge 1$ かつ，すべての $L_SET[i]$ について，$C[base + L_SET[i]] \le 0$ であれば *base* を返して終了．そうでなければ手順 4 へ．

手順 4(LNB-4):未使用要素の変更

e に $-C[e]$ をセットし手順 2 へ．

関数 L_NewBase は，遷移集合 L_SET の最小要素が遷移可能である BASE 値を $O(1)$ で求めることができる．

3.3 list 法の問題点

list 法は，追加処理における関数 NewBase の計算量を大幅に削減できる[8]．しかし，関数 L_PreNode が新たに必要となる．この関数は，先頭未使用要素から入力要素の直前未使用要素を探索するため，$O(m)$ の計算量を要する．よって，normal 法の関数 InsNode および関数 DelNode よりも，拡張したこれらの関数における計算量は増加する．

4. 双方向未使用要素リストを用いたダブル配列法

4.1 提案手法の概要

ここでは，関数 L_PreNode による計算量を削減するため，単方向であった未使用要素リストを双方向にすることを提案する．以下，この手法を doubly 法と呼ぶ．

双方向未使用要素リストは式 (3)，(4) を以下に拡張することにより実現する．また，ダブル配列における要素 $0(e_0)$ をダミー要素として用いる．

	0	1	2	3	4	5	6	7	8	9	10	11	12
BASE	3	1	-5	4	6	1	7	8	9	10	12	-7	13
CHECK	-22	1	16	0	-3	1	-4	-6	-7	-8	-9	16	-10

	13	14	15	16	17	18	19	20	21	22
BASE	14	15	17	2	18	19	22	-10	-1	0
CHECK	-12	-13	-14	5	-15	-17	-18	1	1	-19

図 4 doubly 法によるダブル配列の例

Fig. 4 An example of double-array structure of the doubly method.

前方向のリスト

$$B[e_i] = e_{i+1} \qquad 0 \le i \le m-1 \qquad (5)$$

$$B[e_m] = e_1 \qquad (6)$$

後方向のリスト

$$C[e_i] = -e_{i-1} \qquad 1 \le i \le m \qquad (7)$$

$$C[e_0] = -e_m \qquad (8)$$

doubly 法は，配列 BASE を用いて list 法を拡張するので使用領域は増加しない．list 法同様，ある要素が未使用要素であるかの判定は配列 CHECK を用いて行う．

キー集合 K に対する doubly 法によるダブル配列を図 4 に示す．

4.2 doubly 法の追加アルゴリズム

上記の拡張により関数 L_InsNode を関数 D_InsNode とし，以下に示す．

関数 D_InsNode(*node*, *parent*)

手順 1(DIN-1):前未使用要素の設定

節点 *node* よりも前方に存在する未使用要素を表す変数 *pre* に $-C[node]$ をセットする．

手順 2(DIN-2):未使用要素リストからの削除

$B[pre]$ に $B[node]$ を，$C[B[node]]$ に $C[node]$ をセットし，未使用要素リストを連結しなおす．

手順 3(DIN-3):節点作成

$C[node]$ に *parent* をセットする．

関数 D_InsNode では，関数 L_InsNode で呼び出される関数 L_PreNode が不要となり，$O(1)$ で直前未使用要素を決定する．

双方向未使用要素リストを用いることで，関数 L_InsNode 以外の関数も拡張する必要がある．ここでは，それぞれ適宜拡張を行うこととし，関数名は list 法と同じとする．

以下に，doubly 法における追加例について，関数 D_InsNode を中心に示す．

例 1: 図 4 におけるダブル配列にキー"dos"を追加する．"dos"を探索し，節点 16 において's' での遷移先節点は $B[16]$+'s'$= 21$ となるが，$C[21] = 1$ であるので式 (1)，(2) を満たしていない．よって探索失敗となり，関数 TransNode(16, 's') を呼び出す．TN-1 より，関数 GetLabel(16, L_SET) を呼び出し，$L_SET = \{$ '#', 'i' $\}$ とし，関数 GetBase(16, L_SET, B_SET) を呼び出す．関数 GetBase 内で，$B_SET = \{-5, -7\}$ とし，関数 L_DelNode(2) および関数 L_DelNode(11) を呼び出し，節点 2 と節点 11 を削除する．その後，関数 L_NewBase({ '#', 'i', 's' }) を呼び出し，その戻り値 3 を $B[16]$ にセットする．$B[16]$ を 2 から 3 へ変更したことにより，節点 2 を節点 3 へ，節点 11 を節点 12 へ移動させる．このため TN-2 では，関数 D_InsNode(3, 16) および関数 D_InsNode(12, 16) を呼び出す．関数 D_InsNode(12, 16) について，DIN-1 で *pre* に $-C[12] = 10$ をセットする．その後，DIN-2 において，$B[10]$ に $B[12] = 13$ を，$C[13]$ に $C[12] = -10$ をセットすることで，未使用要素リストから要素 12 を削除する．DIN-3 で $C[12]$ に 16 をセットし，関数 D_InsNode 終了後，$B[12]$ に B_SET の要素 -7 をセットすることにより，節点 11 を節点 12 へ移動し終える．同様に，節点 2 を節点 3 へ移動させる．節点 3 および節点 12 は SP 節点であるので，TN-3 は行わず関数 TransNode を終了する．その後 $B[16]$+'s'$= 22$ より，節点 16 から遷移's' で遷移する SP 節点 22 を作成し，追加処理を終了する．追加後のダブル配列を図 5 に示す．(例終)

	0	1	2	3	4	5	6	7	8	9	10	11	12
BASE	2	1	4	-5	6	1	7	8	9	10	11	12	-7
CHECK	-23	1	0	16	-2	1	-4	-6	-7	-8	-9	-10	16

	13	14	15	16	17	18	19	20	21	22	23
BASE	14	15	17	3	18	19	23	-10	-1	-16	0
CHECK	-11	-13	-14	5	-15	-17	-18	1	1	16	-19

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
TAIL	r	i	e	#	#	#	n	g	#	e	a	r	c	h	#	#	#

図 5 図 4 に対する例 1 における追加処理後のダブル配列

Fig. 5 A double-array structure after insertion processing in example 1.

4.3 doubly 法の削除アルゴリズム

関数 DelNode は，直前未使用要素を求める必要がなく，$O(1)$ で節点を削除することができる．しかし，関数 L_DelNode は関数 L_PreNode を必要とする．関数 L_PreNode は未使用要素数に依存するので，未使用要素が多くなるほど削除処理が遅くなる．これは，追加処理を行わず，繰り返しキーを削除する場合，多くの計算量を要することを示し，動的検索法として重大な問題となる．

ここでは，未使用要素が多い場合でも，関数 L_PreNode の計算量増加を抑制する手法を提案する．関数 L_PreNode を改善した関数 D_PreNode を以下に示す．

関数 D_PreNode は，要素 *node* からダブル配列の先頭方向へ使用済み要素をたどり，未使用要素を探索する．

関数 D_PreNode(*node*)

手順 1(DPN-1):探索要素の設定

node をデクリメントする．

手順 2(DPN-2):未使用要素の判定

$C[node] \le 0$ であれば要素 *node* は未使用であるので，*node* を返して終了．そうでなければ手順 1 へ．

双方向未使用要素リストを用いることにより，関数 L_DelNode を関数 D_DelNode とし，以下に示す．

関数 D_DelNode(*node*)

手順 1(DDN-1):前未使用要素の探索

関数 D_PreNode(*node*) を呼び出し，その戻り値を変数 *pre* にセットする．

手順 2(DDN-2):未使用要素リストへの追加

$B[node]$ に $B[pre]$ を，$B[pre]$ に *node* をセットし，

	0	1	2	3	4	5	6	7	8	9	10	11	12
BASE	2	1	3	4	6	-16	7	8	9	10	11	12	13
CHECK	-22	1	0	-2	-3	1	-4	-6	-7	-8	-9	-10	-11

	13	14	15	16	17	18	19	20	21	22
BASE	14	15	16	17	18	19	22	-10	-1	0
CHECK	-12	-13	-14	-15	-16	-17	-18	1	1	-19

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
TAIL	r	i	e	#	#	#	n	g	#	e	a	r	c	h	#	o	#	#

図 6　図 4 に対する例 2 における削除処理後のダブル配列

Fig. 6　A double-array structure after insertion processing in example 2.

前方へリストを連結させる．$C[node]$ に $-pre$ を，$C[B[node]]$ に $-node$ をセットし，後方へリストを連結させる．

以下に，doubly 法における削除例について，関数 D_PreNode および関数 D_DelNode を中心に示す．

例 2: 図 4 におけるダブル配列からキー"doing"を削除する．まず，"doing"を探索し，SP 節点 11 を入力とする関数 Delete を呼び出す．関数 Delete では，まず SP 節点 11 を削除するために関数 D_DelNode(11) を呼び出す．DDN-1 より関数 D_PreNode(11) を呼び出し，DPN-1 より $node = 10$ とする．DPN-2 で $C[10] \leq 0$ を満たすので，関数 D_PreNode を終了し，*pre* に戻り値である 10 をセットする．DDN-2 において，$B[11]$ に $B[10] = 12$，$B[10]$ に 11，$C[11]$ に -10，$C[12]$ に -11 をセットし，要素 11 を未使用要素リストに追加し，関数 D_DelNode(11) を終了する．SP 節点 11 の親節点であった節点 16 について，子節点が SP 節点 2 のみである．よって，SP 節点 2 よりも根に近い遷移が SP ストリングとなる．SP 節点 2 に関する SP ストリングを変数 *string* とし，*string* ="#"とする．節点 16 から SP 節点 12 への遷移 '#' を *string* に追加し，*string* ="##"とする．その後，SP 節点 2 を削除するために関数 D_DelNode(2) を呼び出し，DPN-1 より $node = 1$ とする．DPN-2 では $C[1] \leq 0$ を満たさないので，DPN-1 より $node = 0$ とする．ここで，$C[0] \leq 0$ を満たすので，関数 D_PreNode を終了し，*pre* に戻り値である 0 をセットする．DDN-2 において，要素 2 を未使用要素リストに追加し，関数 D_DelNode(2) を終了する．節点 16 の親節点 5 はシングル節点であるので，関数 D_DelNode(16) により節点 16 を削除し，*string* ="o##"とする．節点 5 の親節点 1 はシングル節点でないので，節点 5 を SP ストリング"o#"を指す SP 節点とし，関数 Delete を終了する．削除後のダブル配列を図 6 に示す．(例終)

5. 実験による評価

提案手法の有効性を示すため，normal 法，list 法，doubly 法の比較実験を行った．normal 法は約 600 行，list 法は約 800 行，doubly 法は約 800 行の C 言語で実装し，Intel Pentium 4 2.8GHz，Fedora Core3 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 12 万語，日本語キー集合として日本語単語辞書約 40 万語[1)7)]，URI キー集合としてランダムに 100 万件を抽出したものから，それぞれ 10 万語をランダムに抽出したものを用いた．

表 1　キー 10 万語における空間効率に対する実験結果

Table 1　The simulation results of the space efficiency in 100,000 words.

	英語キー集合	日本語キー集合	URI キー集合
平均キー長 (byte)	9.859	8.614	67.190
要素数	192,207	171,627	315,995
使用済み要素数	191,894	155,596	315,905
未使用要素数	313	16,031	90
追加処理における相対速度			
normal 法	0.009	0.369	0.002
list 法	1.000	1.000	1.000
doubly 法	1.512	2.854	1.250
削除処理における相対速度			
normal 法	1.000	1.000	1.000
list 法	0.004	0.005	0.003
doubly 法	0.942	0.965	0.955

表 1 に，上記のキー集合における平均キー長，およびそれらを追加したダブル配列の要素使用状況を示す．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 2byte とする．また，list 法の追加時間を基準値 1 とした normal 法および doubly 法の相対速度，normal 法の削除時間を基準値 1 とした list 法および doubly 法の相対速度をそれぞれ示す．さらに，list 法および doubly 法に関して，追加時間の実験結果を図 7 に示す．上記の実験では，英語キー集合より 1 万語から 1 万語ずつ 10 万語までランダムにキーを抽出したキー集合を用いた．表 1，図 7 より，英語キー集合において，doubly 法の追加処理は normal 法の約 170 倍，list 法の約 1.512 倍高速であった．また，10 万語以下のキー集合を追加した場合においても list 法より約 1.5 倍高速であることが判る．すなわち，list 法において関数 L_InsNode で呼び出される関数 L_PreNode の計算量が全体の約 1/3 であったといえる．また，日本語キー集合に対して，追加速度が list 法よりも約 2.854 倍高速であり，英語キー集合を追加した場合よりも大きく向上していることが判る．これは表 1 より，英語キー集合よりも未使用要素が多いからである．一方，URI キー集合に対する追加速度の向上が小さいのは，未使用要素が少ないからである．これは，関数 L_PreNode が未使用要素数に依存していることを示している．以上より，未使用要素数が全要素数の 0.03% 未満という少ない URI キー集合に対しても提案手法が有効であることが判った．また，未使用要素が存在しない場合でも list 法と等しい追加性能が得られる．

表 2 に，normal 法，list 法，doubly 法の削除時間を示す．上記は，追加したキーをすべて削除する実験を行い，英語キー集合より 2 万語から 2 万語ずつ 10 万語までランダムにキーを抽出したキー集合を用いた．図 8 に，英語キー集合 10 万語を追加後，同一キー集合に対して関数 L_PreNode および関数 D_PreNode を用いた削除処理の実験結果を示す．上記は，追加した順にキーを削除し，1 万語毎の削除時間を計測した．また，各語数における削除時間計測開始状態の未使用要素数を示す．表 1 および表 2 より，関数 L_PreNode を用いた list 法の削除処理が非常に遅いことが判る．これは図 8 より，削除処理によって未使用要素数が増加し，関数 L_PreNode の計算量が増加した

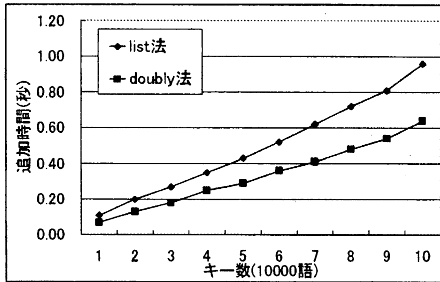

図 7　追加時間に対する実験結果
Fig. 7　The simulation results of the insertion time.

表 2　削除時間に対する実験結果
Table 2　The simulation results of the deletion time.

キー数	20,000	40,000	60,000	80,000	100,000
normal 法	0.05	0.10	0.17	0.25	0.33
list 法	2.72	11.80	27.53	50.02	79.00
doubly 法	0.05	0.11	0.18	0.26	0.35

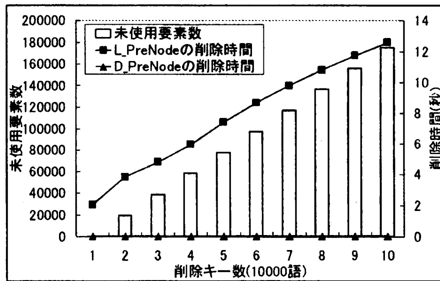

図 8　10 万語追加後における削除処理に対する実験結果
Fig. 8　The simulation results of the deletion processing after 100,000 words insertion.

からであることが判る．しかしながら，関数 D_PreNode を用いた doubly 法の削除時間は normal 法と同等となった．よって，繰り返しキーを削除した場合，提案手法は未使用リストを用いることによる削除時間増加を抑制できる．

図 9 に，関数 L_PreNode と関数 D_PreNode を用いた doubly 法における追加処理の実験結果を示す．上記は，英語キー集合に対して 1 万語毎に追加時間を計測した．また，各語数における追加時間計測開始状態の使用済み要素数を示す．図 9 より，関数 D_PreNode は使用済み要素数に依存し，使用済み要素が多い追加処理では処理速度が遅いことが判る．一方，関数 L_PreNode は，使用済み要素数に関係なく一定の処理速度を保っている．以上から，追加処理，削除処理に関係なく，未使用要素が多い場合は関数 D_PreNode を，使用済み要素が多い場合は関数 L_PreNode を用いることが効果的である．

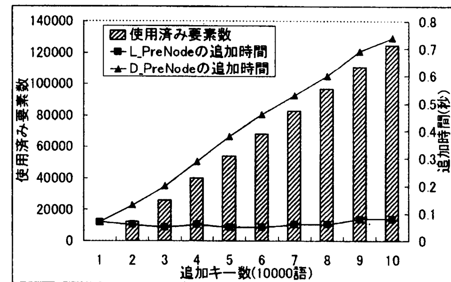

図 9　10 万語追加時における追加時間と使用済み要素数に対する実験結果
Fig. 9　The simulation results of the insertion time and the number of the used elements at the time of 100,000 words insertion.

以上より，提案手法は list 法よりも追加処理および削除処理において有効であるといえる．また，未使用要素数および使用済み要素数に応じて関数 L_PreNode と関数 D_PreENode を使いわけることにより，更新処理はさらに高速となる．

6. おわりに

本論文では，ダブル配列を動的検索法として確立するために，追加処理を高速化する手法および削除時間増加の抑制手法を提案し，実験によりそれらの有効性を示した．今後の課題として，他の動的検索法と提案手法とを比較すること，提案手法を実際の自然言語処理システムに実装して詳細に評価することが挙げられる．

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) Aoe Jun-ichi, Morimoto Katsushi, Shishibori Masami, Park Ki-Hong: A Trie Compaction Algorithm for a Large Set of Key, IEEE Trans.Knowledge and Data Engineering, vol-8, No.-3, pp.476-491, 1996.
3) 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(共立出版)，vol-21, No.-6, pp.36-44, 1989.
4) 青江順一:キー検索技法 IV-トライとその応用-，情報処理学会，vol-34, No.-2, pp.244-251, 1993.
5) 青江順一，森本勝士:ダブル配列法によるトライ検索の実現法，自然言語処理学会研究会資料，NL-85-2, pp.9-16, 1991.
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) 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) 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.