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

2006－FI－82（1）
2006－DD－54（1）
2006／3／22

遷移先節点集合を導入したトライ構造における更新手法の実現

中 村 康 正† 野 村 優† 望 月 久 稔†

自然言語処理システム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，2 つの 1 次元配列を用い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つ有効なデータ構造であるが，動的検索法に比べ更新処理が高速であるとはいえない．そのため現在では，未使用要素を双方向リストとして連結する手法が知られているが，遷移先節点探索の計算回数を抑制する手法ではない．そこで本論文では，更新処理を高速化するため，ダブル配列法に遷移情報を格納する 1 次元配列を新たに用いた手法を提案する．キー集合 20 万語に対する実験を行った結果，提案手法は双方向リストを用いた手法より，追加処理は約 2.4 倍，削除処理は約 2.1 倍高速となった．

Implementation of Updation Techniques for the Trie Structure Based on the Set of Terminal Node

Yasumasa NAKAMURA,† Yu NOMURA† 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 doubly list is known now, it is not the technique of controlling the complexity of terminal node search. In this paper, we presents a fast insertion and deletion algorithms by connecting empty elements as doubly list and reduction algorithm of deletion time. From the simulation results for 200 thousands keys, it turned out that the presented method for insertion is about 2.4 times faster than the doubly list method, and deletion is about 2.1 times faster than the doubly list method.

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トライ木で表現される．そのため，探索オーダーがキー数に依存せず，キー数が膨大な自然言語処理などに用いられている．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知られている[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た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，2 つの 1 次元配列を用いた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有効なデータ構造である．しかし，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ず，未使用要素を単方向にリストとして連結する手法が知られている[6,10]．また，この手法は削除処理が低速となるため，未使用要素リストを双方向へ拡張した手法が現在提案されている[7,9]．しかしながら，ダブル配列法で頻繁に実行される遷移先節点探索の計算回数を抑制する手法ではない．また，グラフ構造の遷移を高速に検出できるトリプル配列法がある[5]．

そこで本論文では，更新処理を高速化するため，遷移先節点情報を格納した 1 次元配列を用いてトライ木を実現するデータ構造と更新処理を提案する．

以下，2 節でダブル配列法を，3 節で提案手法の更新処理を説明する．4 節で提案手法に対して実験による評価を与え，5 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 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，トライ木の始点 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[3,4]．つまり，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 はトライ木の親子関係を一意に決定するために用いる．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 TAIL には，各キーにおける他のキーと共有してい

† 大阪教育大学
Osaka Kyoiku University

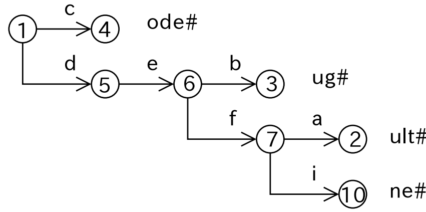

図 1　トライ木の例
Fig. 1　An example of trie tree.

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

	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	#	#	#	#	#

図 2　ダブル配列の例
Fig. 2　An example of double-array structure.

ない遷移を文字列として格納する．これにより，トライ木の節点数を抑制する[3]．以下，配列 TAIL の pos 番目の要素を $T[pos]$ とする．

下記に示す通り，ダブル配列上の未使用要素を双方向リストとして連結する[9]．ここで，ダブル配列上の未使用要素数を m とし，未使用要素の要素番号を昇順に $e_1, e_2, \cdots, e_m$ 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使用済み要素と区別する．また，ダブル配列の要素 $0(e_0)$ をダミー要素として用いる[9]．

後方のリスト

$$B[e_i] = e_{i+1} \qquad 0 \leq i \leq m-1 \tag{3}$$

$$B[e_m] = e_1 \tag{4}$$

前方のリスト

$$C[e_i] = -e_{i-1} \qquad 1 \leq i \leq m \tag{5}$$

$$C[e_0] = -e_m \tag{6}$$

キー集合 $K =${ code#，debug#，default#，define# } に対するトライ木とダブル配列をそれぞれ図 1，図 2 に示す．ここで，'#' は終端記号とする．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　追加アルゴリズム

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[4]．新規追加処理は以下に示す探索失敗時に行う．

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

前者は，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 が使用済み要素であれば，$B[s]$ を変更することにより t とは異なる未使用要素 t' を終点とし，SP 節点を作成する．この一連の処理は関数 TransNode により実現する．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シングル節点を作成する．その後，異なる二つの遷移が可能である BASE 値をもつ節点を作成し，それぞれの SP 節点を作成する．

ダブル配列法における追加処理は，前者における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．そこで，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．ここで，関数 NewBase(S_L) は，遷移集合 S_L の全要素が遷移できる BASE 値を決定する．

関数 InsNode は，節点 $node$ を作成するため未使用要素リストから $node$ を削除し，$C[node]$ に遷移元節点 s を設定する．

関数 InsNode($node$，s)

手順 1(IN-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 $prev$ に $-C[node]$ を設定する．

手順 2(IN-2):未使用要素リストからの削除
$B[prev]$ に $B[node]$ を，$C[B[node]]$ に $C[node]$ を設定し，未使用要素リストを連結しなおす．

手順 3(IN-3):節点作成
$C[node]$ に s を設定する．

関数 DelNode は、節点 $node$ を削除すため未使用要素リストの最終要素として $node$ を追加する．

関数 DelNode($node$)

手順 1(DN-1):前未使用要素の設定
$prev$ に未使用要素リストの最終要素 $-C[0]$ を設定する．

手順 2(DN-2):未使用要素リストへの追加
$B[node]$ に $B[prev]$ を，$B[prev]$ に $node$ を設定し，前方へリストを連結させる．$C[node]$ に $-prev$ を，$C[B[node]]$ に $-node$ を設定し，後方へリストを連結させる．

関数 SearchTnode は，始点 s の遷移先節点番号を S_T へ，遷移先節点への遷移を S_L へ，遷移先節点の BASE 値を S_B へそれぞれ格納し，遷移先節点を削除する．ここで，l_{min} は最小遷移，l_{max} は最大遷移の内部表現値とする．

関数 SearchTnode(s，$base$，S_T，S_L，S_B)

手順 1(ST-1):遷移先節点候補の初期化
遷移先節点候補 t に，節点 s の最小遷移先節点候補 $base + l_{min}$ を設定する．

手順 2(ST-2):遷移先節点の探索
$C[t]$ が s と等しければ S_T に t を，S_L に $t - base$ を，S_B に $B[t]$ を追加する．

手順 3(ST-3):遷移先節点候補の更新
t に次の遷移先節点候補 $t+1$ を設定する．t が s の最大遷移先節点候補 $base + l_{max}$ より大きければ終了し，そうでなければ手順 2 へ．

関数 TansNode における手順 1 の削除処理は，手順 2 で

行う BASE 値決定を容易に行うためである．また BASE 値の再決定後，手順 3 で始点 s の遷移先節点を移動させ，移動した節点の遷移先節点における CHECK 値を関数 RenewalCheck により更新する．このとき，異なる遷移 a，b による遷移先節点を t_a，t_b，BASE 値再決定による移動後の遷移先節点を t_a'，t_b' とすると，t_a' が t_b と等しい場合があり，t_b の遷移先節点を探索すると，t_b の遷移先節点だけでなく CHECK 値を更新した t_a' の遷移先節点が含まれる．そのため関数 RenewalCheck で，すべての遷移先節点候補に対して，S_U が与える更新済みの節点かどうか判断する (以下，判断処理と呼ぶ)．ここで，変数 i は S_T，S_L，S_B の要素番号である．

関数 TransNode(s，$label$)

手順 1(TN-1):遷移先節点情報の待避

関数 SearchTnode(s，$B[s]$，S_T，S_L，S_B) を呼び出し，S_T の全要素に対して関数 DelNode($S_T[i]$) を呼び出す．

手順 2(TN-2):BASE 値の設定

S_L と $label$ のすべてが遷移可能な BASE 値である関数 NewBase($S_L \cup label$) の戻り値を $B[s]$ に設定する．

手順 3(TN-3):節点の移動

S_L の全要素に対して以下を行い終了．s から $S_L[i]$ で遷移する節点 t に $B[s] + S_L[i]$ を設定する．関数 InsNode(t，s) を呼び出し，$B[t]$ に $S_B[i]$ を設定する．このとき $B[t] > 0$ であれば t は遷移先節点を保持するので，関数 RenewalCheck($S_T[i]$，t，$S_B[i]$，S_U) を呼び出す．

関数 RenewalCheck(old，new，$base$，S_U)

手順 1(RC-1):遷移先節点候補の初期化

遷移先節点候補 t に，節点 old の最小遷移先節点候補 $base + l_{min}$ を設定する．

手順 2(RC-2):遷移先節点の探索と判断処理

$C[t]$ が old と等しければ，t が更新済み節点集合 S_U に含まれるか判断する．含まれる場合は手順 4 へ，そうでなければ手順 3 へ．

手順 3(RC-3):遷移先節点における遷移確立

$C[t]$ に new を設定し，S_U に t を追加する．

手順 4(RC-4):遷移先節点候補の更新

t に次の遷移先節点候補 $t+1$ を設定する．t が old の最大遷移先節点候補 $base + l_{max}$ より大きければ終了し，そうでなければ手順 2 へ．

2.3 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．キーの削除にともない，他のキーを構成する遷移および節点を配列 TAIL に写像する処理が必要となる．関数 Delete とともに，ある節点がシングル節点かどうかを判断する関数 IsSingleNode を以下に示す．ここで，FALSE は任意の値であり，$maxPos$ は配列 TAIL における次に使用可能な要素番号とする．

関数 IsSingleNode(s)

手順 1(IS-1):遷移先節点候補の初期化

遷移先節点候補 t に，節点 s の最小遷移先節点候補 $B[s] + l_{min}$ を設定する．また，遷移先節点 $tNode$ に FALSE を設定する．

手順 2(IS-2):遷移先節点の探索

$C[t]$ が s と等しければ，手順 3 へ．そうでなければ手順 4 へ．

手順 3(IS-3):遷移先節点の更新

$tNode$ が FALSE であれば $tNode$ に t を設定し，そうでなければ FALSE を返して終了する．

手順 4(IS-4):遷移先節点候補の更新

t に次の遷移先節点候補 $t+1$ を設定する．t が s の最大遷移先節点候補 $B[s] + l_{max}$ より大きければ $tNode$ を返して終了し，そうでなければ手順 2 へ．

関数 Delete($SPnode$)

手順 1(DE-1):SP 節点の削除

$SPnode$ の遷移元節点 s に $C[SPnode]$ を設定し，関数 DelNode($SPnode$) を呼び出す．

手順 2(DE-2):SP ストリングの初期化

関数 IsSingleNode(s) を呼び出し，その戻り値を t に設定する．t が FALSE であるか $B[t] > 0$ であれば終了し，そうでなければ SP ストリングを格納する配列 str[ ] に $T[-B[t]]$ から始まる SP ストリングを設定する．

手順 3(DE-3):SP ストリングへ写像

s から t への遷移 $t - B[s]$ を str[ ] に追加し，関数 DelNode(t) を呼び出す．

手順 4(DE-4):写像の判断

s に $C[s]$ を設定し，関数 IsSingleNode(s) を呼び出し，その戻り値を t に設定する．t が FALSE でなければ手順 3 へ．

手順 5(DE-5):SP 節点の作成

$B[t]$ に $-maxPos$ を設定し，$T[maxPos]$ から str[ ] を格納する．

2.4 ダブル配列法の問題点

双方向未使用要素リストを用いることにより，BASE 値再決定における計算量を抑制でき更新処理が高速となる．しかしながら，計算回数が遷移種数回必要である遷移先節点探索は頻繁に行われ，多くの計算回数を必要とする．

3. トリプル配列法

3.1 データ構造

遷移先節点探索について，追加処理で用いられる関数 SearchTnode，関数 RenewalCheck は常に遷移種数回の計算回数が必要である，削除処理で用いられる関数 IsSingleNode は，内部表現値が最小な第 1 遷移と次に小さな第 2 遷移による節点だけを探索するだけでよいが，第 2 遷移が存在しない場合は遷移種数回の計算回数が必要である．

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

図 3 トリプル配列の例
Fig. 3 An example of triple-array structure.

そこで，遷移先節点探索を高速化するため，ダブル配列法に遷移先節点情報を格納した 1 次元配列 SIBLING を用いることを提案する．以下，この手法をトリプル配列法と呼び，節点 $node$ に関する配列 SIBLING の値を $S[node]$ とする．始点 s からの終点を $t_0, t_1, \cdots, t_k$ とするとき，以下に示すように配列 SILBING を用いて遷移先節点を循環リストとして連結させる．ここで，遷移先節点リストがただ 1 つの要素 t からなる場合，$S[t]$ に t を格納する．

$$S[t_i] = t_{i+1} \qquad 0 \leq i \leq k-1 \tag{7}$$

$$S[t_k] = t_0 \tag{8}$$

節点を追加および削除する際，配列 SIBLING を更新する必要がある．そのため，遷移先節点リストに要素を追加する関数 InsSibling を関数 InsNode へ，削除する関数 DelSibling を関数 DelNode へ追加することにより，遷移先節点リストの更新が実現する．そこで，以下にこれらの関数を示す．ここで関数 MinTnode$(s, base)$ は，要素 $base$ から節点 s の最小遷移先節点番号を返す．

関数 InsSibling(s, t)

手順 1(IS-1):最小遷移先節点の探索
最小遷移先節点 min に関数 MinTnode(s) の戻り値を設定する．

手順 2(IS-2):遷移先節点リストへの追加
$S[t]$ に $S[min]$ を，$S[min]$ に t を設定する．

関数 DelSibling(t)

手順 1(DS-1):直前遷移先節点の初期化
遷移先節点リストにおける t の直前遷移先節点 $prev$ に $S[t]$ を設定する．

手順 2(DS-2):直前遷移先節点の探索
$S[prev]$ が t と等しくなるまで $prev$ に $S[prev]$ を設定し，手順 2 を繰り返す．

手順 3(DS-3):遷移先節点リストから削除
$S[prev]$ に $S[t]$ を，$S[t]$ に t を設定する．

例 1: キー集合 K に対するトリプル配列を図 3 に示す．始点 6 について，遷移先節点 3 の SIBLING 値 $S[3] = 7$ より他遷移先節点が 7 であり，$S[7] = 3$ であるので始点 6 の遷移先節点は 3 と 7 だけである．(例終了)

3.2 追加アルゴリズム

配列 Sibling により拡張した関数 SearchTnode を関数 EX_SearchTnode とし，以下に示す．

関数 EX_SearchTnode$(s, base, S_T, S_L, S_B)$

手順 1(EST-1):最小遷移先節点の探索
最小遷移先節点 min および遷移先節点 t に，関数 MinTnode$(s, base)$ の戻り値を設定する．

手順 2(EST-2):遷移先節点情報の格納
S_T に t を，S_L に $t - base$ を，S_B に $B[t]$ を追加する．

手順 3(EST-3):遷移先節点候補の更新
t に次の遷移先節点 $S[t]$ を設定する．t が min と等しければ終了し，そうでなければ手順 2 へ．

判断処理に関して，CHECK 値が等しい節点でも更新前の節点と更新済みの節点は異なる遷移先節点リストに連結されている．よって，最小遷移先節点に対して判断処理を行い，更新済み節点でなければ，配列 SIBLING により連結された節点は更新済み節点でないことが保障される．つまり，すべての節点に対して判断処理を行う必要がない．関数 RenewalCheck を拡張した関数 EX_RenewalCheck を以下に示す．

関数 EX_RenewalCheck$(old, new, base, S_U)$

手順 1(ERC-1):判断処理
old の遷移先節点候補 min に関数 MinTnode$(old, base)$ の戻り値を設定し，min が更新済み節点集合 S_U に含まれるか判断する．含まれる場合は手順 1 を繰り返し，そうでなければ遷移先節点 t に min を設定し手順 2 へ．

手順 2(ERC-2):遷移先節点における遷移確立
$C[t]$ に new を設定し，S_U に t を追加する．

手順 3(ERC-3):遷移先節点の更新
t に次の遷移先節点 $S[t]$ を設定する．t が min と等しければ終了し，そうでなければ手順 2 へ．

例 2: 図 3 にキー"decode"を追加する．このとき，式 (1)，(2) を満たしながら節点 6 まで遷移し，'c' による遷移先節点 t が $B[6]$+c$= 4$ について $C[4] \neq 6$ であるので，節点 4 は使用済み要素である．よって，関数 TransNode(6, c) を呼び出す．

TN-1 で関数 EX_SearchTnode(6，1，S_T，S_L，S_B) を呼び出す．EST-1 で min および t に関数 MinTnode(6, 1) の戻り値である 3 を設定し，EST-2 で S_T に 3 を，S_L に $3 - 1 = 2$ つまり b を，S_B に $B[3] = -5$ を追加する．ETS-3 で t に $S[t] = 7$ を設定し，同様に処理を行うことで $S_T = \{3, 7\}$，$S_L = \{$b,f$\}$，$S_B = \{-5, 1\}$ とし，関数 EX_SearchTnode を終了する．

その後，TN-1 で節点 3，7 を削除し，TN-2 で関数 NewBase より $B[6]$ に 5 を設定する．TN-3 で t に $B[6]$+b$= 7$ を設定し，関数 InsNode(7，6) および $B[7]$ に -5 を設定することで，節点 3 を節点 7 へ移動する．同様に節点 7 を節点 11 へ移動する．このとき，$B[11] > 0$ なので関数 EX_RenewalCheck(7，11，1，S_U) を呼び出し，ERC-1 で min に関数 MinTnode(6，1) の戻り値である 2 を設定する．このとき，S_U の要素に 2 が存在しないことを確認し，t に 2 を設定する．ERC-2 で $C[2]$ を 11 に再定義し，S_U に 2 を追加する．その後，ERC-3 で t に $S[2] = 10$ を設定し，$C[10]$ に 11 を設定し，S_U に 10 を追加する．同様に ERC-3 で t に $S[10] = 2$ を設定するが，ここで min が t と等しいので関数 EX_RenewalCheck を終了する．こ

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

	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 4 追加処理後のトリプル配列

Fig. 4 A triple-array structure after insertion processing.

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

	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 5 削除処理後のトリプル配列

Fig. 5 A triple-array structure after deletion processing.

れにより関数 TransNode を終了し，最後に節点 8 に SP 節点を作成することで追加処理を終了する．終了後のトリプル配列を図 4 に示す．(例終了)

3.3 削除アルゴリズム

関数 IsSingleNode では，節点 s がシングル節点であるときはすべての遷移先節点候補を調べる必要があった．しかし，遷移先節点リストを循環にすることで，シングル節点であるかの判断は最小遷移先節点における SIBLING 値が自身であるかを調べるだけでよい．拡張後の関数 IsSingleNode を関数 EX_IsSingleNode とし，以下に示す．

関数 EX_IsSingleNode(s)

手順 1(EIS-1):遷移先節点候補の初期化

最小遷移先節点 min に関数 MinTnode(s，$B[s]$) の戻り値を設定する．

手順 2(EIS-2):遷移先節点の探索

$S[min]$ が min と等しければ min を．そうでなければ FALSE を返し終了する．

例 3: 図 3 からキー"default"を削除する．このとき，式 (1)，(2) を満たしながら SP 節点 2 まで遷移し，関数 Delete(2) を呼び出す．DE-1 で s に $C[2]=7$ を設定し，関数 DelNode(2) を呼び出す．関数 DelNode では，未使用要素リストの最終要素として 2 を追加し，関数 DelSibling(2) を呼び出す．

DS-1 で $prev$ に $S[2]=10$ を設定し，DS-2 で $S[prev]=2$ と $t=2$ が等しいので DS-3 を行う．ここで，$S[10]$ に $S[2]=10$ を設定することにより遷移先節点リストから 2 を削除し，$S[2]$ に 2 を設定することで $S[2]$ を初期化する．

DE-2 で関数 EX_IsSingleNode(7) を呼び出し，EIS-1 で関数 MinTnode(7，$B[7]=1$) の戻り値 10 を min に設定する．EIS-2 により，$S[10]$ と 10 が等しいので，節点 7 の遷移先節点リストは 10 のみで構成されており，節点 7 はシングル節点であるとわかる．よって，ただ 1 つの遷移先節点 10 を関数 Delete に返し，それを t に設定する．

DE-2 で $B[10]<0$ であることを確認し，$str=$\{ ne# \} とする．DE-3 で 7 から 10 への遷移 $10-B[7]=9$，つまり i を str に追加し，$str=$\{ ine# \} とし関数 DelNode(10) を呼び出す．DE-4 で s に $C[7]=6$ を設定し，関数 EX_IsSingleNode(6) を呼び出す．関数 EX_IsSingleNode は min に 3 が設定され，$S[3]$ が 6 であるので FALSE を返し，DE-5 で SP 節点を作成し，削除処理を終了する．

終了後のトリプル配列を図 5 に示す．(例終了)

4. 実験による評価

提案手法の有効性を示すため，青江が提案したダブル配列法 (以下，対象手法 A)，森田らが提案した単方向未使用リストを用いた手法 (対象手法 B)，大野らが提案した双方向未使用リストを用いた手法 (対象手法 C) との比較実験を Intel Pentium 4 2.8GHz，Fedora Core4 上で行った．実験では，日本語キー集合として日本語単語辞書約 40 万語[1)8)]，英語キー集合として英単語辞書約 40 万語[2)]，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 20 万語をランダムに抽出したものを用いた．ここで，各キーの内部表現値は EUC コードとした．

表 1 に，各キー集合の特徴として，平均キー長，追加処理後のトライ木における内部節点を対象として算出した平均遷移数，トライ木における節点数およびシングル節点数を示す．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 2byte とする．さらに各手法の性能として，使用 byte 数，追加処理と追加処理における遷移先節点探索および判断処理の計算回数，追加および削除における処理時間を示す．また，日本語キー集合より 1 万語から 20 万語まで 1 万語ずつランダムに抽出した部分キー集合に対して，対象手法 B，対象手法 C，提案手法の追加処理における時間と遷移先節点探索と判断処理の合計計算回数を，対象手法 A，対象手法 C，提案手法の削除処理における削除時間と遷移先節点探索の計算回数をそれぞれ図 6，図 7 に示す．

追加処理に関して，表 1，図 6 より日本語キー集合に対して，提案手法の処理速度は対象手法 A の約 112.52 倍，対象手法 B の約 3.23 倍，対象手法 C の約 2.43 倍高速であった．この要因は，図 6 から判るように処理時間が遷移先節点探索および判断処理の計算回数に依存しており，それを削減できたことである．また英語および URI キー集合に対しては，日本語キー集合より提案手法に大きな向上はみられなかった．これは，英語および URI キー集合における平均遷移数が少なく判断処理で用いられる S_U の要素数が少なくなったためである．

削除処理に関して，表 1，図 7 より，日本語キー集合に対して，提案手法の処理速度は対象手法 A の約 0.941 倍と低速になったが，対象手法 B，対象手法 C に対しては約 1631.20 倍，約 2.11 倍と高速になった．対象手法 A に対しての結果は，シングル節点の割合が少なく関数 IsSingleNode の計算回数をあまり削減できず，さらに平均遷移数が多く関数 DelSibling の計算回数が多いことが原因である．しかしながら，シングル節点が多く，平均遷移数が

表 1　キー 20 万語に対する実験結果
Table 1　The simulation results in 200,000 words.

	日本語キー集合	英語キー集合	URI キー集合
平均キー長 (byte)	7.157	9.462	57.462
平均遷移数			
対象手法 A，提案手法	2.629	2.130	1.264
対象手法 B，対象手法 C	1.326	1.304	1.047
節点数			
対象手法 A，提案手法	122,824	177,044	757,567
対象手法 B，対象手法 C	612,801	656,468	4,257,606
シングル節点数			
対象手法 A，提案手法	41,889	70,288	643,584
対象手法 B，対象手法 C	531,867	549,712	4,143,623
使用 byte 数			
対象手法 A	3,745,136	4,113,563	14,361,654
対象手法 B	6,503,608	6,852,368	35,660,848
対象手法 C	8,657,032	7,868,808	35,927,808
提案手法	8,825,832	7,455,199	19,039,958
追加時間 (秒)			
対象手法 A	167.504	271.656	2,750.258
対象手法 B	2.778	1.007	1.322
対象手法 C	2.301	0.899	1.276
提案手法	0.760	0.438	0.712
SearchTnode の計算回数			
対象手法 A	88,454,934	72,636,392	60,511,790
対象手法 B	185,458,601	132,640,236	109,402,605
対象手法 C	150,181,578	118,480,187	107,893,906
提案手法	62,002,498	28,830,369	27,483,980
RenewalCheck の計算回数			
対象手法 A	230,585,152	10,586,346	3,688,522
対象手法 B	421,927,537	8,168,829	1,344,946
対象手法 C	476,888,362	7,746,927	1,934,058
提案手法	18,724,534	1,150,982	444,538
削除時間 (秒)			
対象手法 A	0.589	0.639	1.503
対象手法 B	1,431.171	1,554701	50,078.551
対象手法 C	0.975	1.000	4.683
提案手法	0.583	0.522	1.058
IsSingleNode の計算回数			
対象手法 A	86,007,177	89,152,938	233,960,472
対象手法 B	187,064,030	186,811,623	1,220,350,780
対象手法 C	186,451,293	186,154,530	1,106,093,337
提案手法	60,008,811	43,932,538	91,262,104

少ない英語および URI キー集合は，対象手法 A に対してもそれぞれ約 1.20 倍，約 1.48 倍高速となり，他手法に関しても日本語キー集合よりも向上が大きかった．また内部表現値の分布も大きな要因の 1 つであり，各キーの内部表現値を ECU コードとしたため，英語および URI キー集合は日本語キー集合より小さな内部表現値が多く存在する．つまり，内部表現値の分布に依存する最小遷移先節点探索の計算回数が日本語キー集合よりも少なくなる．

表 1 より，提案手法の使用領域は配列 SIBLING を用いることで，対象手法 A よりも日本語キー集合では約 2.36 倍，英語キー集合では約 1.81 倍，URI キー集合では約 1.32 倍となり増加したが，配列 TAIL を用いない対象手法 B，対象手法 C とは同等かそれ以下となった．よって，提案手法は各対象手法よりも有効であるといえる．

5. おわりに

本論文では，ダブル配列法に遷移先節点情報を格納した配列 SIBLING を導入し，更新処理を高速化する手法を提案し，実験により有効性を示した．今後の課題として，増加した使用領域を抑制することがあげられる．

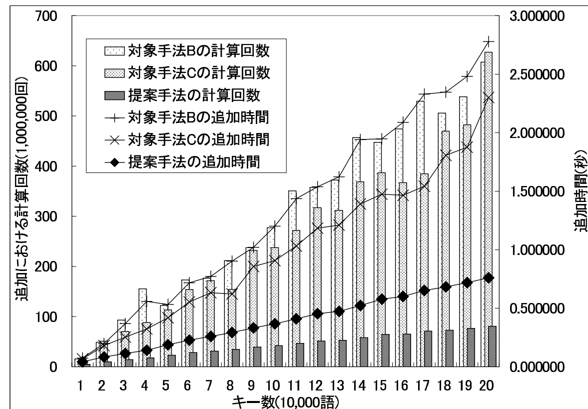

図 6　追加処理に対する実験結果
Fig. 6　The simulation results of the insertion processing.

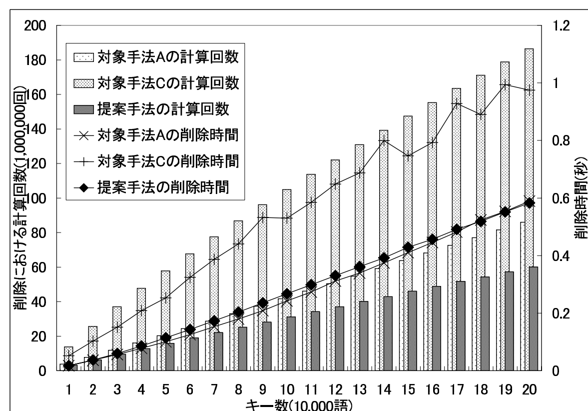

図 7　20 万語追加後における削除処理に対する実験結果
Fig. 7　The simulation results of the deletion processing after 200,000 words insertion.

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) Kevin Atkinson，Ispell English Word Lists，http://wordlist.sourceforge.net/.
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，Vol.21，No.6，pp.36-44，1989.
4) 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.
5) 入口浩一，津田和彦，獅子堀正幹，青江順一，グラフ構造に対する効率的記憶検索法，電子情報通信学会論文誌 D-1，Vol.j79-D-I，No.8，pp.502-510，1996.
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.
8) 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/.
9) 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.
10) 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.