RD-001

ダブル配列の遷移集合管理による追加 · 削除処理の高速化
Managing Transitions of Double-Array for Fast Insertion and Deletion

重越　秀美[†]　　蔵満　琢麻[†]　　望月　久稔[†]
Hidemi SHIGEKOSHI　Takuma KURAMITSU　Hisatoshi MOCHIZUKI

1．　はじめに

トライは，キー自体を節点に保持する 2 分探索木と異なり，キーを構成する要素（以下，遷移種）により遷移する木構造であり，キーの探索時間がキー数ではなくキー長に依存するため，キー数が膨大な自然言語処理の辞書などに広く用いられる [1)]．

トライのデータ構造の 1 つに，探索処理が高速で，記憶領域がコンパクトなダブル配列 [2)] がある．しかし，ダブル配列は，遷移を定義する際に，遷移集合の取得や配列内の未使用である要素の探索が必要であり，キーの追加 · 削除処理が高速であるとは言えず，データベースの牽引など動的な更新が頻繁に発生する場合に不向きである．現在では，配列内の未使用である要素を双方向リスト（以下，未使用要素リスト）として管理することで，未使用要素の探索時間を抑制する手法 [3, 4)] が提案されているが，遷移集合の取得を高速化する手法ではない．

大野 [5)]，矢田 [6)] らは，遷移先が 1 つである節点にフラグを設けて，削除処理の時間計算量を抑制する手法を提案した．これらの手法は，追加処理を高速化する手法ではない．中村ら [7)] は，1 次元配列を新たに設けて，共通の遷移元を持つ節点を循環リストにより管理することで，遷移集合の取得に要する時間を抑制する手法を提案した．しかし，遷移集合を取得するために，遷移先の節点を 1 つ取得するまでダブル配列上を線形に走査する必要がある．

本論文では，ダブル配列に新たな 2 本の 1 次元配列を設けて，各節点における遷移集合を管理し，遷移定義に要する時間を抑制することで，キーの追加 · 削除処理を高速化する手法を提案する．また，遷移集合の要素数によって未使用要素リストの走査方法を変更することで，追加処理の更なる効率化を図る．

以下，2 章でダブル配列の追加 · 削除処理について述べ，3 章で，追加 · 削除処理の高速化手法を提案する．4 章で提案手法に理論的 · 実験的評価を与え，5 章で本論文の総括と今後の課題を述べる．

2．　ダブル配列の追加 · 削除処理

本章では，まずダブル配列の概要と追加 · 削除処理について説明し，関連研究による改善について述べる．

2.1　ダブル配列の概要

ダブル配列は 2 本の 1 次元配列 Base と Check を用いて，節点 s から遷移種 a による節点 t への遷移を式 (1) により定義する [1, 2)]．節点 s の Base 値には遷移先節点集合を定義するための基底値，遷移先節点 t の Check 値には遷移元の節点（以下，親節点）を格納する．以下，キーを構成する遷移種の集合を Σ と表記し，節点 s における遷移種 a と遷移先節点 t の組を $(a,\ t)$，遷移集合 $\{(a_1,\ t_1),\ (a_2,\ t_2),\ \cdots,\ (a_k,\ t_k);\ 1 \leq k \leq |\ \Sigma\ |\}$ を $\mathrm{R}(s)$，遷移種集合 $\{a_1,\ a_2,\ \cdots,\ a_k\}$ を $\mathrm{L}(s)$，遷移先節点集合 $\{t_1,\ t_2,\ \cdots,\ t_k\}$ を $\mathrm{T}(s)$ と表記する．また，集合 A の要素数を $|A|$ と表記する．

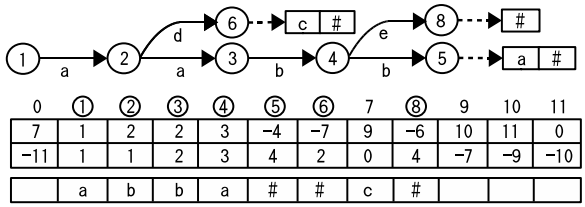
$$\begin{cases} t = \mathrm{Base}[s] + a \\ \mathrm{Check}[t] = s \end{cases} \tag{1}$$

ダブル配列中でトライの節点として使用中の要素を使用要素，それ以外の要素を未使用要素と呼ぶ．未使用要素は，Check 値を 0 以下にすることで他の節点と区別し，未使用要素リストにより管理する [3, 4)]．未使用要素における Base 値をリストの後方へのリンク，Check 値をリストの前方へのリンクとして用いることで双方向リストを実現する．以下，ダブル配列上で最も後方の使用要素を最大使用要素と呼び，最大使用要素までの範囲内における使用要素の割合をダブル配列における空間効率とする．

ダブル配列は，トライ上の葉にあたる節点（以下，葉節点）とキーを 1 対 1 対応させるために，キーの末尾に終端記号 ‘#’（‘#’ $\notin \Sigma$）を付加する．また，記憶領域を抑制するために，キー判別に必要な最終分岐を含む遷移までをトライ構造として構築し，それ以降の遷移は文字列として 1 次元配列 Tail に格納する [2)]．葉節点の Base 値には，配列 Tail に格納した文字列の位置を負値にして格納する．

遷移種 ‘#’, ‘a’, ‘b’, ···, ‘e’ の内部表現値をそれぞれ 0, 1, ···, 5 とし，キー集合 {“aabba#”, “adc#”, “aabe#”} を登録したダブル配列を図 1 に示す．トライの節点として使用していない未使用要素節点 7，9，10，11 は，節点 0 をヘッダとする未使用要素リストにより管理する．

共通の親節点を持つ節点を兄弟節点と呼び，兄弟節点が存在しない節点をシングル節点と呼ぶ．例えば，節点 3 の兄弟節点は共通の親節点 2 を持つ節点 6 であり，兄弟節点が存在しない節点 2 はシングル節点である．



	0	①	②	③	④	⑤	⑥	7	⑧	9	10	11
Base	7	1	2	2	3	−4	−7	9	−6	10	11	0
Check	−11	1	1	2	3	4	2	0	4	−7	−9	−10
Tail		a	b	b	a	#	#	c	#			

図 1　ダブル配列

[†] 大阪教育大学, Osaka Kyoiku University

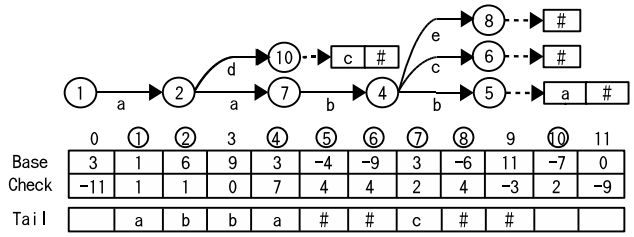
2.2 追加処理の問題点

ダブル配列は，キーの追加において節点 s に遷移種 a による遷移先節点 t を新たに定義する場合，節点 t =Base[s] + a が未使用要素であれば遷移を定義できるが，既にトライの節点として使用されていれば衝突が発生する [2]．この衝突を回避するには，節点 s（または節点 t の親節点 Check[t]）の Base 値を変更して遷移先節点集合 T(s)（または T(Check[t])）の格納場所を変更し，遷移集合 R(s)（または R(Check[t])）を再定義する必要がある．衝突に該当する遷移集合の要素数が少ない程，遷移集合の再定義に要する時間を抑制できるため，|R(s)| < |R(Check[t])| であれば節点 s の Base 値を，そうでなければ節点 Check[t] の Base 値を変更する．よって，ダブル配列の追加処理では，節点 s における遷移集合 R(s) を取得する処理が発生し，R(s) を取得するためには，ダブル配列上を Base[s] から Base[s]+ | Σ | まで線形に走査し，Check 値が s である節点 s の遷移先節点をすべて見つける必要がある．

ダブル配列は，節点 s の遷移集合 R(s) を再定義する際，まず未使用要素リストを走査して $\{newBase + x;\ x \in \mathrm{L}(s)\}$ がすべて未使用要素となる基底値 $newBase$ を求め，節点 s の遷移先節点集合 T(s) を新規節点集合 $\{newBase + x;\ x \in \mathrm{L}(s)\}$ へ移動する．ここで，T(s) の要素 t を新規節点集合の要素 t' に移動する場合，遷移式 (1) を満たすために，T(t) における各要素の Check 値を t' に変更する．その後 T(s) をトライ上から削除する．

図 1 のダブル配列に，キー “aabc#” を追加する例について考える．遷移種 ‘a’, ‘a’, ‘b’ により節点 4 まで遷移した後，‘c’ による遷移が定義されていないため，節点 4 から ‘c’ による遷移を定義する必要がある．しかし，Base[4]=3 に遷移種 ‘c’ の内部表現値 3 を加えた節点 6 は既に使用要素であるため衝突が発生する．衝突処理では，まずダブル配列を走査して R(4)={(‘b’, 5), (‘e’, 8)} と R(Check[6])=R(2)={(‘a’, 3), (‘d’, 6)} を取得し，要素数 |R(4)| と |R(2)| を比較する．それぞれの要素数が等しいため，既存節点 6 の親節点 2 の Base 値を変更する．未使用要素リストを線形に走査して，$newBase$ に L(2) の各要素を加えた位置がすべて未使用要素になる $newBase$ 6 を取得し，R(2) を再定義するために，T(2) の要素である節点 3，節点 6 をそれぞれ新規節点集合の要素である節点 7(6+‘a’)，節点 10(6+‘d’) へ移動する．このとき，T(2) の遷移先節点集合 T(3)={4}，T(6)=ϕ を取得し，節点 4 の Check 値を 7 に更新する．その後，T(2) をトライ上から削除し，節点 6 を遷移種 ‘c’ による節点 4 からの遷移先として定義する．

図 1 のダブル配列にキー “aabc#” を追加したダブル配列を図 2 に示し，未使用要素リストの問題点について説明する．図 2 に示すように，トライ上から削除した未使用要素（以下，削除要素）節点 3 の周辺には使用要素が多く存在し，削除要素は複数の要素を持つ新規節点集合として再利用しづらい特徴を持つ．従来のダブル配列は未使用要素リストの先頭に削除要素を追加する [3, 6] が，リスト前方に新規節点集合として再利用しづらい削除要素が蓄積すると，遷移集合を再定義する際に未使用要素リストを走査する回数が増加する．



	0	①	②	3	④	⑤	⑥	⑦	⑧	9	⑩	11
Base	3	1	6	9	3	−4	−9	3	−6	11	−7	0
Check	−11	1	1	0	7	4	4	2	4	−3	2	−9
Tail		a	b	b	a	#	#	c	#	#		

図 2　衝突処理後のダブル配列

2.3 削除処理の問題点

ダブル配列におけるキーの削除は，削除対象のキーを検索してダブル配列に登録済みであることを確認し，到達した葉節点を削除することで実現する [2]．ここで，葉節点の削除により生じた不要な節点をすべて除去する．

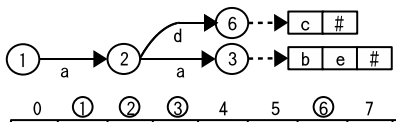
例えば，図 1 のダブル配列からキー “aabba#” を削除するため，葉節点 5 を削除する場合，節点 3 が存在すればキー “aabe#” を判別できるため，シングル節点 4, 8 が不要になる．ダブル配列は削除した葉節点の兄弟節点を求め，兄弟節点が葉節点かつシングル節点である場合，キー判別に必要な節点以降のシングル節点をトライ上から削除し，配列 Tail を更新することで削除処理を実現する．

図 1 のダブル配列からキー “aabba#” を削除したダブル配列を図 3 に示す．キーの削除により節点 3 は葉節点となり，キー “aabe#” の接尾辞 “be#” を配列 Tail 上に新たに定義する．

削除処理は，削除した葉節点の兄弟節点を求める処理や，参照する節点がシングル節点か否かを判定する（以下，シングル判定）処理において，追加処理と同様にダブル配列を走査する必要がある．また，削除処理を繰り返すと最大使用要素までに存在する未使用要素数が増加し，空間効率が低下する [5, 6]．

2.4 関連研究

大野 [5]，矢田ら [6] はシングル節点の親節点における Check 値を負値で扱うことにより，シングル判定を O(1) で行う手法を提案した．また，削除処理を繰り返すとダブル配列の空間効率が低下する問題を解決するために，最大



	0	①	②	③	4	5	⑥	7	8	9	10	11
Base	4	1	2	−9	8	7	−7	9	5	10	11	0
Check	−11	1	1	2	0	−8	2	−5	−4	−7	−9	−10
Tail		a	b	b	a	#	#	c	#	b	e	#

図 3　削除処理後のダブル配列

使用要素とその兄弟節点を配列前方の未使用要素に移動することで，ダブル配列のガベージコレクションを行う手法を提案した．これらの論文では，遷移集合の取得についての議論は行われていない．

中村らは遷移集合の取得に要する時間計算量を抑制するために配列 Sibling を新たに設け，兄弟節点を式 (2) に示す循環リストにより管理する手法を提案した [7]．この手法は，ある節点における兄弟節点を O(1) で取得できるため，構築処理と削除処理を高速化できるが，遷移先節点を 1 つ取得するまではダブル配列上を線形に走査する必要がある．

$$\begin{cases} \text{Sibling}[\text{Base}[s]+a_i] = \text{Base}[s]+a_{i+1} \\ \text{Sibling}[\text{Base}[s]+a_k] = \text{Base}[s]+a_1 \\ \qquad\qquad (1 \le i \le k-1) \end{cases} \quad (2)$$

この問題に対し，著者らは，遷移先節点へのリンクを管理する配列を設けることで，遷移集合の取得に要する時間計算量をさらに抑制する手法を提案した [8]．

未使用要素リストの効率化に関して，中村らは，新規節点集合として再利用しづらい削除要素を未使用要素リストの最後尾に追加する手法を提案した [4]．以下，削除要素を未使用要素リストの先頭に追加する方式を LIFO 方式，最後尾に追加する方式を FIFO 方式と呼ぶ．FIFO 方式では，遷移集合を再定義するために未使用要素リストを走査する際，新規節点集合として再利用しづらい削除要素を走査しないため，LIFO 方式に比べ追加処理が高速になる．しかし，削除節点が再利用されずに未使用要素リスト後方に蓄積し，ダブル配列の空間効率が低下する．

矢田らは，ダブル配列を一定区間に分割して各ブロックごとに未使用要素を管理し，新規節点集合の探索に数多く失敗するブロックの利用を制限することで，未使用要素の蓄積による 1 キーあたりの追加時間の増加を抑制する手法を提案した [9]．登録キー数が増加しても，空間効率と更新時間のいずれかが極端に低下しないことが報告されているが，ブロックごとに未使用要素数や新規節点集合の探索に失敗した回数などの情報を管理するための記憶領域と，各情報を更新する処理が必要になる．

3． 遷移定義の高速化

本章では，まず，配列 Sibling と Child を新たに設けて各節点の遷移集合を管理し，キーの追加処理と削除処理の高速化を図る手法を提案する．次に，条件を設けて削除要素を再利用することで，空間効率を低下させずに追加処理の高速化を図る手法を提案する．

3.1 遷移集合の管理法

本節では，まず配列 Sibling による兄弟節点の取得と，配列 Child による遷移先節点の取得について説明し，その後，遷移集合の取得方法について説明する．次に，削除処理におけるシングル判定方法について説明する．

まず，提案手法の配列 Sibling について説明する．式 (1) より，節点 s の遷移先節点 t は，節点 s の Base 値と節点 t への遷移種があれば特定できる．そこで，配列 Sibling に兄弟節点の節点番号ではなく，兄弟節点への遷移種を格納する手法を提案する．遷移種を表現するためのバイト幅は，配列の全要素を表現するバイト幅よりも小さいため，節点番号を格納するよりも配列 Sibling の 1 要素あたりの領域を抑制できる．節点 s の遷移種集合 $\text{L}(s)$ を，式 (3) に示す循環リストにより管理する．

$$\begin{cases} \text{Sibling}[\text{Base}[s]+a_i] = a_{i+1} \\ \text{Sibling}[\text{Base}[s]+a_k] = a_1 \\ \qquad\qquad (1 \le i \le k-1) \end{cases} \quad (3)$$

節点 t の親節点 s は Check[t] を参照することで取得でき，節点 t の兄弟節点への遷移種は Sibling[t] を参照することで取得できるため，節点 t の兄弟節点 v は，式 (4) を用いて取得できる．

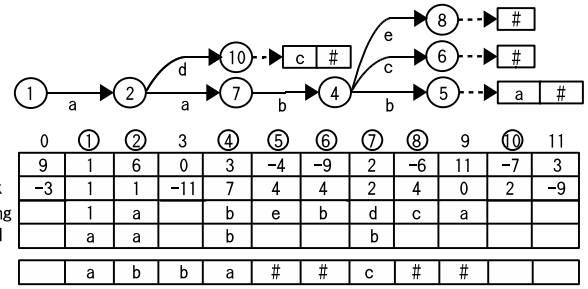
$$v = \text{Base}[\text{Check}[t]] + \text{Sibling}[t] \quad (4)$$

次に，配列 Child について説明する．遷移集合を取得する際，兄弟節点の管理のみでは，遷移先節点を 1 つ取得するまでダブル配列上を線形に走査する必要がある．そこで，配列 Child を新たに設けて遷移先節点への遷移種を格納することで，遷移先節点へのリンクを管理する手法を提案する．

図 2 のキー集合を登録した提案手法のダブル配列を図 4 に示す．節点 2 の Child 値には L(2)={‘a’, ‘d’} の要素 ‘a’ を格納し，節点 7 の Sibling 値には兄弟節点 10 への遷移種 ‘d’ を格納する．

配列 Sibling と Child を用いた遷移集合の取得方法について説明する．節点 s における遷移集合 $\text{R}(s)$ は，遷移種集合 $\text{L}(s)$ から算出できる．図 5 に示す関数 GetLabel は，$\text{L}(s)$ を取得するために，節点 s における Base 値と Child 値から遷移先節点を取得し，節点 s の遷移先節点集合 $\text{T}(s)$ を走査して，$\text{L}(s)$ の要素を集合 *label* に格納する．

関数 GetLabel を用いて，図 4 に示すダブル配列から節点 2 の遷移種集合 L(2) を取得する例を考える．手順 1 で，Base[2]+Child[2]=6+‘a’=7 より節点 2 の遷移先節点 7 を求め，手順 2 で兄弟節点 10 への遷移種 Sibling[7]=‘d’ を取得する．その後 base[2]+Sibling[7]=6+‘d’=10 より遷移先節点 10 を求め，兄弟節点 7 への遷移種 Sibling[10]=‘a’ を取得する．ここで，Sibling[10]=‘a’ と Child[2]=‘a’ が一致し，すべての遷移先節点を走査し終え，L(2)={‘a’,



	0	①	②	3	④	⑤	⑥	⑦	⑧	9	⑩	11
Base	9	1	6	0	3	-4	-9	2	-6	11	-7	3
Check	-3	1	1	-11	7	4	4	2	4	0	2	-9
Sibling		1	a		b	e	b	d	c	a		
Child		a	a		b			b				
Tail		a	b	b	a	#	#	c	#	#		

図 4　提案手法のダブル配列

関数 GetLabel
引数： 節点 s

手順 1:遷移先節点の取得
集合 $label$=ϕ とし，遷移先節点 t に 1 つめの遷移先節点 Base[s]+Child[s] をセットする．
手順 2:遷移種集合の取得
集合 $label$ に兄弟節点への遷移種 Sibling[t] を追加する．Sibling[t] と 1 つめの遷移先節点への遷移種 Child[s] が同じであれば終了し，そうでなければ t に兄弟節点 Base[s]+Sibling[t] をセットして手順 2 を繰り返す．

関数 IsSingle
引数： 節点 t

節点 t への遷移種 t − Base[Check[t]] と Sibling[t] が一致すれば TRUE を返し，そうでなければ FALSE を返す．

図 5　関数 GetLabel，関数 IsSingle

'd'} を取得する．関数 GetLabel を用いることで，遷移集合の要素数回の走査で遷移集合を取得できるため，追加処理の高速化を図れる．

次に，配列 Sibling と Child を用いた削除処理におけるシングル判定方法について説明する．式 (1) より，節点 t への遷移種 a は t − Base[Check[t]] により取得できる．また，兄弟節点を式 (3) に示す循環リストにより管理するため，節点 t がシングル節点であれば，Sibling[t] には節点 t への遷移種 b を格納する．よって，a と b が一致した場合，節点 t はシングル節点である．図 5 に示す関数 IsSingle は，節点 t への遷移種から O(1) でシングル判定を行い，節点 t がシングル節点であれば TRUE を返す．

3.2　削除要素の管理法

本節では，FIFO 方式の未使用要素リストに追加した削除要素を効率的に再利用し，空間効率を低下させずに追加処理の高速化を図る手法を提案する．

ダブル配列の要素 0 を h_0，ダブル配列中で最も後方の未使用要素を h_{max} と定義する．h_0，h_{max} を除く未使用要素の数を n とし，未使用要素を e_1，e_2，$\cdots$，e_m，e_{m+1}，$\cdots$，e_n $(m \leq n)$ としたときの未使用要素リストを図 6 に示す．ただし，h_{max} 以降の e_{m+1}，$\cdots$，e_n は削除要素であり，ダブル配列上の順番と未使用要素の順番は関連しない．未使用要素リストに削除要素が存在しない場合は，Check[h_0] が $-h_{max}$ になる．なお，h_0, と h_{max} は未使用要素リストのヘッダと終端に使用する要素であるため，節点の定義先に用いない．

ダブル配列では，シングル節点を作成する際，新規節点を未使用要素リストから一意に取得できる [5]．そこで，削除要素をシングル節点の新規節点として再利用する．シングル節点の新規節点 $Snode$ を取得する関数 Get と，未使用要素リストの最後尾に削除要素 $Enode$ を追加する関数 Put を図 7 に示す．

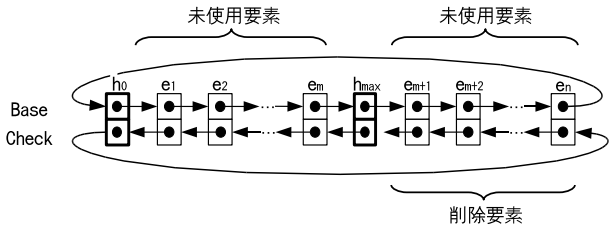


図 6　未使用要素リスト

関数 Get

Check[h_0]$\neq -h_{max}$ で削除要素が存在すれば，$Snode$ に削除要素 $-$Check[h_0] をセットする．そうでなければ，$Snode$ に未使用要素リストの先頭要素である Base[h_0] をセットする．未使用要素リストを更新するため，Base[$-$Check[$Snode$]] に Base[$Snode$] を，Check[Base[$Snode$]] に $-$Check[$Snode$] を格納する．その後，$Snode$ を返して終了する．

関数 Put
引数： 削除要素 $Enode$

$next$ に未使用要素リストの最後尾の要素である $-$Check[h_0] をセットする．その後，未使用要素リスト後方へのリンクを連結するため，Base[$Enode$] に h_0 を，Base[$next$] に $Enode$ を格納する．前方へのリンクを連結するため，Check[$Enode$] に $-next$ を，Check[h_0] に $-Enode$ を格納する．

図 7　関数 Get，関数 Put

関数 Get はシングル節点の新規節点を取得するときのみ用いる．関数 Get は，未使用要素リストに削除要素が存在するか否かを判断し，存在する場合は最後尾の削除要素を再利用し，存在しない場合は未使用要素リストの先頭要素を返す．

提案手法では，関数 Put を用いて未使用要素リストの最後尾に削除要素を追加することで，未使用要素リスト前方に削除要素が集中する LIFO 方式の問題点を緩和し，新規節点集合の取得に要する時間計算量の抑制を図る．また，関数 Get を用いて削除要素をシングル節点の作成時に再利用することで，未使用要素リスト後方に削除要素が蓄積する FIFO 方式の問題点を緩和し，空間効率低下の抑制を図る．

4．　評価

矢田らにより提案された手法 [6]（以下，手法 Y）と中村らにより提案された手法 [7]（以下，手法 N）を比較手法とし，追加時間と削除時間，キー集合を登録するために必要な記憶領域について評価する．

4.1　理論的評価

まず，追加時間について評価する．以下，追加するキーの長さを l，遷移種の数を $|\Sigma|$，最大使用要素までの未使用要素の数を e とする．

トライ構造であるダブル配列の追加処理は，衝突が発生しなければ O(l) である．しかし，衝突が発生した場合は，遷移集合の取得と遷移集合の再定義時に行う未使用要素リストの走査に時間を要する．以下，遷移集合の取得時にダブル配列を走査する回数を SL 回数，遷移集合の再定義時に未使用要素リストを走査する回数を SF 回数とし，それぞれの回数を評価する．

節点 x の遷移種集合 L(x) に含まれる遷移種における内部表現値の最小値を A とし，節点 x の遷移集合 R(x) を取得する際の SL 回数を評価する．手法 Y では，R(x) の取得時に，ダブル配列上に遷移種数分存在する遷移先節点の候補すべてを線形に調べるため，SL 回数は | Σ | 回である．兄弟節点の管理を行う手法 N では，手法 Y より SL 回数を抑制するが，遷移先節点を 1 つ取得するまでダブル配列を線形に調べた後，遷移先節点集合 T(x) を走査するため，SL 回数は $A + 1 +$ |R(x)| 回である．これに対し，提案手法は，遷移先節点を一意に取得し，T(x) を走査して R(x) を取得するため，SL 回数は |R(x)| 回になり，他手法に比べ SL 回数を抑制する．

手法 Y の SF 回数は最悪で O(e) である [4] が，提案手法と手法 N は，新規節点集合として用いづらい削除要素を未使用要素リストの最後尾へ追加するため，手法 Y よりも SF 回数を抑制する可能性がある．提案手法と手法 N の未使用要素リストの効果は，次節で実験により評価する．SL 回数と SF 回数を抑制する提案手法は追加処理が他手法よりも高速であると言える．

次に，削除時間について評価する．ダブル配列の削除処理は O(l) の探索処理に加え，キー判別に不要な節点を除去するために兄弟節点の取得とシングル判定が必要である．以下，兄弟節点を取得する際にダブル配列を走査する回数を SS 回数とし，シングル判定の時間計算量と SS 回数について評価する．

手法 Y では，親節点の Check 値を利用することで，提案手法と手法 N では，配列 Sibling を利用することで，シングル判定の時間計算量を O(1) にする．

兄弟節点の取得時にダブル配列を走査する必要がある手法 Y の SS 回数は O(| Σ |) になる．一方，提案手法と手法 N では，配列 Sibling を利用して，兄弟節点を一意に取得できるため，SS 回数は 1 回になる．以上のことから，提案手法と手法 N は，手法 Y よりも削除処理が高速であると言える．

最後に，キー集合を登録するために必要な記憶領域を各配列に必要な領域の和とし，各手法について評価する．各手法とも配列 Tail に必要な記憶領域は等しいため，配列 Base，Check，Sibling，Child について評価する．以下，各配列の 1 要素あたりに必要な記憶領域と，最大使用要素までの要素数を評価する．

配列 Base と Check を 1 要素あたり 4byte の配列を用いて実現し，節点番号を格納する手法 N の配列 Sibling を 1 要素あたり 4byte，遷移種を格納する提案手法の配列 Sibling と Child を 1 要素あたり 1byte の配列を用いて実現した．よって，提案手法，手法 Y，手法 N における 1 節点あたりの記憶領域はそれぞれ 10byte，8byte，12byte である．提案手法は配列 Sibling と Child に遷移種を格納するため，配列 Sibling に節点番号を格納する手法 N より 1 節点あたりの記憶領域が小さい．しかし，提案手法は手法 Y と比較すると 1 節点あたりの記憶領域が 1.25 倍に増加する．

削除節点を再利用する提案手法と手法 Y は，最大使用要素が手法 N よりも小さいと言える．提案手法における未使用要素リストの効果は次節で実験的に評価する．

4.2 実験的評価

提案手法の有効性を示すため，他手法との比較実験を Intel Pentium Dual 1.6GHz，Fedora7 上で行った．ここで，未使用要素リストの管理方法を評価するため，提案手法の配列 Sibling と Child を実装し，LIFO 方式で未使用要素を管理した手法（以下，LIFO）をあわせて実験した．

実験では，ランダムに抽出した 20 万語の英単語（以下，キー集合 E）と 20 万語の日本語形態素（以下，キー集合 J）をキー集合として使用し，各キー集合をそれぞれランダムな順でダブル配列に登録し，その後すべてのキーを削除した．実験では遷移種を 1byte として扱い，キーの内部表現値に ASCII コードを用いた．また，配列 Tail を 1 要素あたり 1byte の配列を用いて実現した．

表 1 に，各キー集合を登録する際の衝突回数，SL 回数，SF 回数，構築時間，最大使用節点，空間効率，記憶領域，すべてのキーを削除する際の SS 回数，削除時間を示す．ここで，SL 回数，SF 回数，SS 回数を各処理ごとの平均回数とし，記憶領域を配列 Tail を除く各配列に必要な記憶領域の合計とする．

以下では，まず各手法の追加時間と削除時間について評価し，次に，ダブル配列の空間効率とキー集合を登録するために必要な記憶領域について評価する．

まず，追加時間について評価する．表 1 に示すように，提案手法は他手法より構築時間を短縮した．以下，各手法の衝突回数，SL 回数，SF 回数について評価する．表 1 より，20 万語のキー集合を登録する際に衝突が発生した確率は，提案手法においてキー集合 E で 97.0%，キー集合 J で 93.8%，手法 Y においてキー集合 E で 98.4%，キー集合 J で 99.0%，手法 N においてキー集合 E で 67.1%，キー集合 J で 66.9% であった．このように，キーを追加する際は衝突が頻繁に発生するため，衝突処理を高速化することが追加処理の高速化に有効であるといえる．ここで，手法 N は，表 1 に示すように，最大使用節点が他手法より大きく，新たな遷移を定義する節点の作成場所が未使用要素である可能性が高いため，衝突回数が少ない．

提案手法は手法 Y や手法 N に比べ，表 1 に示すように，SL 回数を 97.4%〜99.1% 抑制した．さらに，提案手法は削除要素を未使用要素リストの最後尾に追加するため，手法 Y に対して SF 回数をキー集合 E で 68.7%，キー集合

J で 74.6% 抑制した．しかし，提案手法の SF 回数は手法 N よりも多い．これは，手法 N の衝突回数が少ない理由と同じである．

次に，削除時間について評価する．提案手法と手法 N は，兄弟節点を O(1) で取得できるため，O(| Σ |) で取得する手法 Y に対して，表 1 に示すように削除時間をキー集合 E で 84.2%，キー集合 J で 82.7% 抑制した．

次に，各手法の空間効率について評価する．表 1 に示すように，FIFO 方式の未使用要素リストを用いる手法 N の空間効率は，手法 Y と比較して低下する．これに対し，提案手法の空間効率は，LIFO 方式の未使用要素リストとほぼ同等であった．これは，提案手法が削除要素を効率的に再利用できることを示す．

最後に，キー集合を登録するために必要な記憶領域について評価する．提案手法は，1 節点あたりの領域が手法 Y より大きいため，手法 Y に対して記憶領域がキー集合 E で 20.7%，キー集合 J で 22.0% 増加する．しかし，提案手法より 1 節点あたりの領域が大きく，空間効率が低下する手法 N に対しては，記憶領域をキー集合 E で 32.7%，キー集合 J で 31.4% 削減した．

以上，提案手法が構築処理と削除処理の高速化に有効であることを示した．手法 Y に対しては，記憶領域がキー集合 E で 20.7%，キー集合 J で 22.0% 増加したが，構築時間をキー集合 E で 92.8%，キー集合 J で 92.3% 削減し，削除時間をキー集合 E で 84.2%，キー集合 J で 82.7% 削減した．また，手法 N に対しては，記憶領域をキー集合 E で 32.7%，キー集合 J で 31.4% 削減し，構築時間をキー集合 E で 45.3%，キー集合 J で 54.8% 削減した．

表 1　実験の結果

	提案	手法 Y	手法 N	LIFO
キー集合 E				
衝突回数 (k)	194.08	196.79	134.28	195.27
SL 回数	2.41	256.00	91.32	2.45
SF 回数	8.01	25.58	2.43	23.60
構築時間 (s)	0.29	4.00	0.53	0.34
最大使用節点 (k)	377.71	376.74	493.42	376.93
空間効率 (%)	99.70	99.96	76.32	99.91
記憶領域 (MB)	4.43	3.67	6.58	4.43
SS 回数	1.00	90.52	1.00	1.00
削除時間 (s)	0.21	1.33	0.21	0.21
キー集合 J				
衝突回数 (k)	187.65	197.95	133.76	197.38
SL 回数	2.74	256.00	144.53	2.80
SF 回数	15.29	60.18	3.49	63.73
構築時間 (s)	0.33	4.26	0.73	0.52
最大使用節点 (k)	385.35	375.10	501.14	375.32
空間効率 (%)	97.29	99.95	74.81	99.89
記憶領域 (MB)	4.72	3.87	6.88	4.62
SS 回数	1.00	149.63	1.00	1.00
削除時間 (s)	0.23	1.33	0.22	0.22

5．おわりに

本論文では，2 本の配列 Sibling と Child により遷移種集合を管理し，遷移集合の取得や兄弟節点の取得，シングル判定の時間計算量を抑制することで，追加処理と削除処理を高速化する手法を提案した．また，遷移集合の要素数によって未使用要素リストの走査方法を変更することで，空間効率を維持しつつ追加処理を高速化する手法を提案した．日，英単語 20 万語における実験結果より，提案手法は，矢田らや中村らの手法と比較して構築時間を 46.0%〜92.8% 削減し，削除時間を 31.4%〜83.9% 削減した．

遷移集合を管理する手法は，矢田ら [6] の提案したガベージコレクションにも利用できる．今後の課題として，配列 Sibling と Child を利用して，配列 Tail を含めた効率的なガベージコレクションを実装し，実システムに応用することが挙げられる．

参考文献

1) 青江順一．キー検索技法-IV トライとその応用．情報処理, Vol. 34, No. 2, pp. 244–251, 1993.
2) Jun'ichi Aoe. An efficient digital search algorithm by using a double-array structure. *IEEE Transactions on Software Engineering*, Vol. 15, No. 9, pp. 1066–1077, 1989.
3) 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.
4) 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), pp. 117–122, 2005.
5) 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.
6) 矢田普, 大野将樹, 森田和宏, 泓田正雄, 吉成友子, 青江順一．接頭辞ダブル配列における空間効率を低下させないキー削除法．情報処理学会論文誌, Vol. 47, No. 6, pp. 1894–1902, 2006.
7) 中村康正, 野村優, 望月久稔．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．情報処理学会研究報告 (FI-82/DD-54), No. 33, pp. 1–6, 2006.
8) 重越秀美, 芳本貴男, 中川智之, 蔵満琢麻, 望月久稔．遷移先節点集合の管理によるダブル配列の更新手法．第 7 回情報科学技術フォーラム (FIT2008), pp. 109–110, 2008.
9) 矢田晋, 田村雅浩, 森田和宏, 泓田正雄, 青江順一．ダブル配列による動的辞書の構成と評価．情報処理学会第 71 回全国大会, pp. 263–264, 2009.