ダブル配列における動的更新の効率化アルゴリズム

森田和宏[†] 泓田正雄[†]
大野将樹[†] 青江順一[†]

トライ構造はキーの表記文字単位に構成された木構造を用いて検索するキー検索技法の 1 つであり，自然言語辞書を中心として広く用いられている．このトライ構造を実現するデータ構造として高速性とコンパクト性を満足するダブル配列法があるが，この手法は，キーの更新が頻繁に生じない検索法として確立しているため，動的検索法に比べて追加時間は高速であるとはいえず，また削除で生じる不要なノードや未使用要素により記憶量に無駄が生じていた．本論文ではこれらの問題を解決し，ダブル配列を動的検索法として確立するため，未使用要素を連結することで追加処理を高速化する手法，削除時に生じる不要ノードの削除と未使用要素の詰め直しによる圧縮法を提案する．10 万語の辞書データに対する実験結果により，追加速度については約 1,600 倍高速となることが，また大量の削除が起こった場合でも 50%以上の空間使用率を維持することが分かった．

Efficient Dynamic Update Algorithms for a Double-array Structure

Kazuhiro Morita,[†] Masao Fuketa,[†] Masaki Oono[†]
and Jun-ichi Aoe[†]

In many information retrieval applications, it is necessary to be able to adopt a trie search for looking at the input character by character. As a fast and compact data structure for a trie, a double-array is presented. However, the insertion time isn't faster than other dynamic retrieval methods because the double-array is a semi-static retrieval method that cannot treat high frequent updating. Further, the space efficiency of the double-array degrades with the number of deletions because it keeps empty elements produced by deletion. This paper presents a fast insertion algorithm by linking empty elements to find inserting positions quickly and a compression algorithm by reallocating empty elements for each deletion. From the simulation results for 100 thousands keys, it turned out that the presented method for insertion is about 1,600 times faster than the original method, and that the presented method for space efficiency keeps the activity ratio more than 50%.

1. はじめに

トライ構造[1)~3)] はキーの表記文字単位に構成された木構造を用いて検索するキー検索技法の 1 つであり，登録キーの総数に依存せず高速な検索ができること，検索失敗位置の特定が容易であること，検索文字列中の接頭辞の検出が容易であることなどの理由により，コンパイラにおける語彙解析器[4),5)]，文献検索[6)]，スペルチェッカ[7)]，形態素解析[8)~10)] などの辞書を中心として広く用いられている．

このトライ構造を実現するデータ構造として一般的な手法には，配列を用いた手法とリストを用いた手法とがある．しかし，配列を用いた手法では配列がスパースになるため記憶量に無駄が生じ，リストを用いた手法では高速検索に難がある．

これらを解決する手法に，青江[1),11),12)] の提案したダブル配列法がある．ダブル配列法は検索の高速性とコンパクト性をあわせ持つ優れたデータ構造である．しかし，ダブル配列法はあらかじめ基本的キー集合を構築しておき，後に使用者がキーを適宜追加する準静的検索法として確立されているため，追加時間は高速であるとはいえない．また，青江[11)] によって与えられた追加時間の理論的評価では，高速な追加処理への可能性を示しているが，実装方法と評価は議論されていない．さらに，不要ノードの削除や未使用要素の圧縮方法が与えられていないため，頻繁な更新処理では空間使用効率が非常に悪くなる欠点がある．

本論文ではこれらの問題を解決し，ダブル配列を動的検索法として確立するため，未使用要素を連結することで追加処理を高速化する手法，削除時に生じる不要ノードの削除と未使用要素の詰め直しによる圧縮法

† 徳島大学工学部
Faculty of Engineering, Tokushima University

を提案する．追加処理について，従来手法ではキーの追加時に発生する衝突を回避するためにダブル配列の全要素を走査するので，追加時間がキー数に依存していた．これに対し，提案手法では未使用要素のみを走査するので，キー数に依存せず高速な追加が可能となる．また，大量のキー削除が起った場合に多くなる未使用要素を抑えるために，圧縮手法では，ダブル配列の後方に位置するノードを可能な限り前方に移動し，後方に集まった未使用要素を削除する．

10 万語の辞書データに対する実験結果により，追加速度については約 1,600 倍高速となることが，また大量の削除が起こった場合でも 50%以上の空間使用率を維持することが分かった．

以下，2 章でダブル配列法とその問題点について述べる．3 章では追加の高速化アルゴリズムを提案し，4 章で削除時の記憶圧縮アルゴリズムを提案する．5 章では提案手法の理論的評価と実験による評価を与え，考察を加える．6 章では本論文のまとめと今後の課題についてふれる．

2. ダブル配列と問題点

2.1 トライ構造

トライ構造[1)~3)] はキーの共通接頭辞を併合圧縮した木構造であり，検索はトライの枝にラベル付けされた文字単位にたどるので，任意の入力文字列に対して，キーの最長一致検索や接頭辞のみが一致する検索が容易であり，遷移を $O(1)$ で検索できれば，キーの数に無関係な高速検索が実現できる．

トライの初期ノード（根）を 1 とし，トライの葉とキーを 1 対 1 に対応させるために，キー自身には含まれない終端記号 '#' をキーの最後につけてトライを構成する．ノード（節）s から t へラベル a を持つ枝が定義されている，すなわち s から t への遷移が定義されていることを関数 g で $g(s,a) = t$ と定義し，遷移が定義されていない場合は，$g(s,a) = fail$ と記述する．関数 g は文字列 X に対しても使用され，$g(s,X) = t$ と記述する．以後，枝にラベル付けされた文字または終端記号を遷移ラベルと表し，文字列 X に対する終端記号付きの文字列を $X\#$ と表す．

[例 1] 図 1 にキー集合 $K = \{bachelor\#, back\#, badge\#, badger\#, beach\#, beta\#, bevel\#\}$ に対するトライの例を示す．図 1 において，キー "$back\#$" の検索は，初期ノード 1 から遷移ラベル 'b' によりノード 4 への遷移が定義されているので，$g(1,\text{'}b\text{'}) = 4$ となる．以後同様に $g(4,\text{'}a\text{'}) = 3$，$g(3,\text{'}c\text{'}) = 5$，$g(5,\text{'}k\text{'}) = 13$，$g(13,\text{'}\#\text{'}) = 8$ と遷移をたどることで検索でき，$g(1,\text{'}back\#\text{'}) = 8$ となる．（例終）

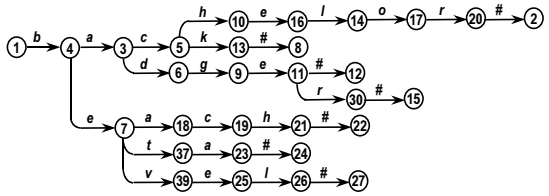

図 1 トライの例
Fig. 1 An example of a trie structure.

2.2 ダブル配列法

ダブル配列法[1),11),12)] では，2 つの配列 BASE, CHECK でトライの遷移を表現し，遷移ラベル a の内部コード値（numerical code）を $\mathrm{N}(a)$ で表すとき，$g(s,a) = t$ に対して，次を満足する．

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

すなわち，BASE[s] にはノード番号 t へのベースインデックスを格納し，$g(s,a)$ のノード番号 t は，ノード番号 s に対する BASE[s] と $\mathrm{N}(a)$ の和で計算され，CHECK[t] にはノード番号 s から引かれたことを定義する s を格納する．したがって，ダブル配列によるノードの遷移は 2 つの式を確認するだけであるため，つねに $O(1)$ で行われ，きわめて高速である．また，記憶量は，トライのノード数と未使用要素数に依存するが，未使用要素数が少なければ，非常にコンパクトになる．

ダブル配列上のインデックス番号はトライのノード番号と 1 対 1 に対応するので，以後簡単のため両者の値を一致させ，両者を同等に扱って説明する．ダブル配列の未使用要素は値 0 とする．また，通常のノードと葉ノードを区別するために葉ノードの BASE 値を負数とする．これにより，検索の成功を判定することができる．

[例 2] 図 1 のトライをダブル配列法を用いて構築した例を**図 2** に示す．遷移ラベルの内部コード値は，終端記号 '#' を 1，文字 'a'〜'z' を 2〜27 とする．たとえば，ノード番号 1 から 4 への遷移 $g(1,\text{'}b\text{'}) = 4$ は，$\mathrm{BASE}[1] + \mathrm{N}(\text{'}b\text{'}) = 1+3 = 4$，$\mathrm{CHECK}[4] = 1$ より，式 (1) を満たすので，遷移が定義されていることが分かり，$g(3,\text{'}b\text{'}) = fail$ は，$\mathrm{BASE}[3]+\mathrm{N}(\text{'}b\text{'}) = 1+3 = 4$，$\mathrm{CHECK}[4] \neq 3$ より，式 (1) を満たさないので，遷移が定義されていないことが分かる．また，ノード番号 2 や 8 は葉ノードであるので，$\mathrm{BASE}[2] = -1$，$\mathrm{BASE}[8] = -2$ のように BASE 値が負数となっている．（例終）

2.2.1 検索アルゴリズム

ダブル配列で使用されている最大のインデックス番

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
BASE	1	-1	1	1	1	1	16	-2	5	10	11	-3	7	1	-4	1	1	15	12	1
CHECK	1	20	4	1	3	3	4	13	6	5	9	11	5	16	30	10	14	7	18	17

	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
BASE	21	-5	23	-6	13	26	-7	0	0	14	0	0	0	0	0	0	21	0	19
CHECK	19	21	37	23	39	25	26	0	0	11	0	0	0	0	0	0	7	0	7

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

号を DA_SIZE とするとき，式 (1) に対して，次のノード番号 t がダブル配列のインデックスの範囲内にあるという条件を加えた，$g(s,a)$ を確認するための関数 Forward(s,a) を次に定義する．

[関数 Forward(s,a)]

begin
　$t = \mathrm{BASE}[s] + \mathrm{N}(a)$;
　if $(0 < t < DA\_SIZE + 1)$
　　and $(\mathrm{CHECK}[t] = s)$ **then** return(t)
　else return(0);
end;

（関数終）

入力キー X に対し，$g(1, X\#) = s$ なるノード s を返すダブル配列の検索アルゴリズム Trie_Search を以下に示す．ただし，$X\# = a_1 a_2 \ldots a_n a_{n+1}$, a_{n+1} ='#' と表す．

[関数 Trie_Search(X)]

手順（D-1）：{ 変数の初期化 }

ダブル配列での現在のインデックス番号を示す変数 $index$ をルートノード番号 1 に，キーの文字位置を表す変数 pos を 1 に初期化する．

手順（D-2）：{ トライの検索 }

Forward$(index, a_{pos})$ の戻り値 t が 0 であれば，検索は失敗であるので 0 を返し終了する．t が 0 でなければ，次のノードをたどるために t を $index$ にセットし，次の文字を検索するため pos をインクリメントする．

手順（D-3）：{ 終了判定 }

BASE$[index] \geq 0$ ならば手順（D-2）へ．BASE$[index] < 0$ ならば，$index$ は葉ノードとなり検索は成功するので，$index$ を返して終了する．（関数終）

[例 3] 図 2 のダブル配列において，キー "*beta#*" を検索する例を示す．まず，手順（D-1）で，$index$ に 1 を，pos に 1 をセットする．次に，手順（D-2）で，Forward(1,'*b*') において，$t = \mathrm{BASE}[1] + \mathrm{N}(\text{'}b\text{'}) = 1 + 3 = 4$, $\mathrm{CHECK}[4] = 1 = s$ より，$g(1, \text{'}b\text{'}) = 4$ が定義され，$index = 4$, $pos = 2$ となる．手順（D-3）で，BASE$[index]$ = BASE[4] $= 1 > 0$ であるので手順（D-2）へ．以下，同様に，Forward$(4, \text{'}e\text{'}) = 7$, Forward$(7, \text{'}t\text{'}) = 37$, Forward$(37, \text{'}a\,box\text{'}) = 23$, Forward$(23, \text{'}\#\text{'}) = 24$ となり，$\mathrm{BASE}[24] = -6 < 0$ より検索が成功し，葉ノード番号 24 を返す．（例終）

2.2.2 追加アルゴリズム

ダブル配列の追加アルゴリズムは，検索アルゴリズムの手順（D-2）で，検索失敗時に関数 INSERT$(index, pos)$ を呼び出すことで実現する．ノード $index$ に新しい遷移ラベル b を追加する場合，$t = \mathrm{BASE}[index] + \mathrm{N}(b)$ なるインデックス t の CHECK$[t]$ が未使用ならば追加が可能であるが，使用されている場合は衝突が起こるので，ノード $index$ から出ているすべての遷移ラベルの集合 R に新しいラベル b を加えたすべての遷移が格納できる新しい BASE$[index]$ を決定し，すべての遷移ラベルを新しい場所に移動して格納する．この処理は関数 MODIFY$(index, R, b)$ で実行され，新しい BASE 値の決定は関数 X_CHECK$(R \cup \{b\})$ で実行される．

以下で使用する関数を，配列 BASE と CHECK に値を格納する関数 W_BASE と W_CHECK とともに示す．

W_BASE$(index, val)$： BASE$[index]$ に val を格納する．ただし，$index > DA\_SIZE$ のときの DA_SIZE を $index$ まで拡張する．

W_CHECK$(index, val)$： CHECK$[index]$ に val を格納する．ただし，$index > DA\_SIZE$ のときの DA_SIZE を $index$ まで拡張する．

GET_LABEL$(index)$： ノード $index$ から出る遷移ラベルを要素とする集合 R を返す．

関数 X_CHECK(A) は $c \in A$ なるラベル c すべてが $\mathrm{CHECK}[q + \mathrm{N}(c)] = 0$ を満足する最小のインデックス q (> 1) を返す関数であり，インデックス q に対し，$\mathrm{CHECK}[q + \mathrm{N}(c)] = 0$ を満足するか調べ（手順（X-2）），満足しなければ q を次の調査対象に移動する（手順（X-3））．

[関数 X_CHECK(A)]

手順（X-1）：{ 初期化 }

ダブル配列のインデックスを格納する変数 q に 1 をセットする．

手順（X-2）：{ インデックス検索 }

$c \in A$ なるラベル c すべてに対し，$\mathrm{CHECK}[q + \mathrm{N}(c)] = 0$ を満足すれば，q を返し終了する．満足しなければ，q をインクリメントし，手順（X-3）へ．

手順（X-3）：{ 終了判定 }

$q \leq DA\_SIZE$ であれば，手順（X-2）へ．$q > DA\_SIZE$ であれば，q を返し，終了する．

（関数終）

以下，関数 INSERT と MODIFY を示す．

関数 INSERT は，キー追加時に衝突が起こった場合，ノードの移動によって衝突を回避し（手順（I-1）），キーの残りの文字列を追加する（手順（I-3））．

[関数 INSERT($index, pos$)]

手順（I-1）：{ 衝突が起こった場合の処理 }

CHECK[BASE[$index$] + N(a_{pos})] > 0 であれば，衝突が起こっているので，$index$ から出る遷移ラベルの集合を GET_LABEL($index$) により変数 R にセットし，関数 MODIFY($index, R, a_{pos}$) により，ノードの移動処理を行う．

手順（I-2）：{ 遷移先の定義 }

t = BASE[$index$] + N(a_{pos}) を得，W_CHECK(t, $index$) により遷移を定義し，残りの文字列を格納するために t を $index$ にセットし，pos をインクリメントする．

手順（I-3）：{ 残りの文字列の追加 }

$newbase$ = X_CHECK({a_{pos}}) により新しい BASE 値 $newbase$ を決定し，W_BASE($index$, $newbase$) により BASE[$index$] に $newbase$ を格納する．遷移を定義するため t = BASE[$index$] + N(a_{pos}) を得，W_CHECK($t, index$) により CHECK[t] に $index$ を格納し，次の遷移を定義するために t を $index$ にセットし，pos をインクリメントする．

これを pos が $n+1$ になるまで繰り返す．ただし，$pos = n+1$ の場合は葉ノードであるので，W_BASE($t, -1$) とする．（関数終）

関数 MODIFY は，ノード $index$ と $index$ から出る遷移ラベルの集合 R，追加するラベル b を引数にとり，$R \cup \{b\}$ に対する新しい遷移先を決定し（手順（M-1）），$index$ から出るノードを新しい遷移先に移動する（手順（M-2））．さらに，移動したノードから遷移するノードがあれば，その CHECK 値を再定義する（手順（M-3））．

[関数 MODIFY($index, R, b$)]

手順（M-1）：{ 新しい BASE[$index$] の決定 }

BASE[$index$] を $oldbase$ に退避しておき，BASE[$index$] には，W_BASE($index$, X_CHECK($R \cup \{b\}$)) によって新しい BASE 値を設定する．

手順（M-2）：{ 遷移の移動 }

ノード $index$ の遷移先を移動するために，ラベル c ($\in R$) に対し，新しい遷移先 t = BASE[$index$] + N(c) を得る．ノード t に対し，W_CHECK($t, index$) により CHECK を再定義し，古い遷移先 old_t = $oldbase$ + N(c) に対し，W_BASE(t, BASE[old_t]) により古い遷移先の BASE 値をコピーする．このとき，BASE[old_t] > 0 であれば，old_t に対する遷移先も再定義が必要なので，手順（M-3）へ．最後に old_t を未使用要素に設定し，終了する．

手順（M-3）：{ 新しい遷移先の遷移先の再定義 }

CHECK[q] = old_t なるすべての q に対して，W_CHECK(q, t) により CHECK 値を再定義する．（関数終）

[例 4] 図 2 のダブル配列において，キー “*baby#*” を追加する例を示す．まず，手順（D-1），（D-2），（D-3）で，Forward(1, ‘*b*’) = 4，Forward(4, ‘*a*’) = 3 となるが，Forward(3, ‘*b*’) で CHECK[4] = 1 ≠ 3 となり，検索に失敗する．そこで，関数 INSERT が呼ばれるが，手順（I-1）で，t = BASE[3] + N(‘*b*’) = 4，CHECK[4] > 0 なので，R = GET_LABEL(3) = {‘*c*’, ‘*d*’} より，関数 MODIFY(3, {‘*c*’, ‘*d*’}, {‘*b*’}) を呼び，衝突処理を行う．手順（M-1）で，$oldbase$ = BASE[3] = 1 を退避し，X_CHECK({‘*b*’, ‘*c*’} ∪ {‘*d*’}) = 28 を BASE[3] に設定する．手順（M-2）で，t = BASE[3] + N(‘*c*’) = 32 に対し，CHECK[32] = 3 と，old_t = $oldbase$ + N(‘*c*’) = 5 から BASE[32] = BASE[old_t] = BASE[5] = 1 を設定する．また，BASE[5] > 0 なので，手順（M-3）で，CHECK[q] = 5 となる q = {10, 13} に対し，CHECK[10] = 32，CHECK[13] = 32 を設定し，BASE[5] = CHECK[5] = 0 とする．文字 ‘*d*’ についても同様に処理し，手順（I-2）で，CHECK[BASE[3] + N(‘*b*’)] = CHECK[31] = 3 により遷移を定義し，手順（I-3）で，キー “*baby#*” の残りの文字列 “*y#*” について，BASE[31] = X_CHECK({‘*y*’}) = 2，CHECK[BASE[31] + N(‘*y*’)] = CHECK[28] = 31，BASE[28] = X_CHECK({‘#’}) = 28，CHECK[BASE[28] + N(‘#’)] = CHECK[29] = 28，BASE[29] = −1 と設定し，終了する．（例終）

2.2.3 削除アルゴリズム

削除アルゴリズムは，検索アルゴリズムの手順（D-3）で，検索成功時に関数 DELETE を呼び，葉ノード $index$ を削除することで実現する．関数 DELETE ($index$) は，W_BASE($index$,0) と W_CHECK($index$,0) により，ダブル配列の要素を未使用にすることで実行される．

[例 5] 図 2 のダブル配列において，キー “*beach#*” を削除する例を示す．ます，手順（D-1），（D-2），（D-3）で，Forward(1, ‘*b*’) = 4，Forward(4, ‘*e*’) = 7，Forward(7, ‘*a*’) = 18，Forward(18, ‘*c*’) = 19，Forward(19, ‘*h*’) = 21，Forward(21, ‘#’) = 22，BASE[22] = −5 < 0 より検索が成功し，

DELETE(22) により，W_BASE(22,0)，W_CHECK (22,0) が行われ，BASE[22] = CHECK[22] = 0 となる．（例終）

2.3 ダブル配列の問題点

ダブル配列の追加における問題点は，関数 X_CHECK の処理時間である．この関数は新しい BASE 値を探すためにダブル配列上のインデックスをシーケンシャルに走査しているので，最悪の場合最大インデックス DA_SIZE に比例した計算量を必要とし，ダブル配列のサイズが大きくなると，この処理時間は非常に長くなる．

削除における問題点は，葉ノードのみを削除することに起因する．図 1 で，キー “*beach#*” を削除すると，葉ノード 22 のみが削除されるが，ノード 18，19，21 が不要ノードとして残るので，新たな追加にも再利用されず，記憶量の無駄となる．また，大量の削除が起こった場合，未使用要素の割合が増加するが，これらの要素数を少なくするための圧縮技法が提案されていないので，これも空間効率に悪影響を与えることになる．

3. 動的追加アルゴリズム

関数 X_CHECK が新しい BASE[$index$] の値を探すために必要とするのは，未使用要素のみなので，それらの要素のみ走査できればこの関数の処理時間を大幅に短縮することができる．この走査処理を効率化するために，未使用要素リスト法を提案する．

3.1 未使用要素リスト法

まず，未使用要素リストを定義する．

[定義 1] ダブル配列の未使用要素のインデックス番号を，昇順に $r_1, r_2, \ldots, r_m$ とするとき，

$\mathrm{CHECK}[r_i] = -r_{i+1} \quad (1 \le i \le m-1)$

$\mathrm{CHECK}[r_m] = -(DA\_SIZE + 1)$

なるリストを作成する．ただし，r_1 はリストの先頭を表す変数 E_HEAD に格納される．これらを未使用要素リストと呼ぶ．未使用要素の CHECK 値を負数にするのは，通常ノード番号と区別するためである．（定義終）

[例 6] 図 1 のトライに対する未使用要素リスト法を用いたダブル配列の例を**図 3** に示す．$E\_HEAD = 28$ から未使用要素リストをたどることができる．（例終）

3.2 追加の高速化アルゴリズム

未使用要素リストの導入によって，関数 X_CHECK は未使用要素のみを走査でき，高速化が実現できる．以下に X_CHECK の変更を示す．ただし，A に含まれる文字 c のうち，N(c) が最小のものを c_1 とする．

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
BASE	1	-1	1	1	1	1	16	-2	5	10	11	-3	7	1	-4	1	1	15	12	1
CHECK	1	20	4	1	3	3	4	13	6	5	9	11	5	16	30	10	14	7	18	17

	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
BASE	21	-5	23	-6	13	26	-7	0	0	14	0	0	0	0	0	0	21	0	19
CHECK	19	21	37	23	39	25	26	-29	-31	11	-32	-33	-34	-35	-36	-38	7	-40	7

図 3 未使用要素リスト法の例
Fig. 3 An example of the list method of empty elements.

[関数 X_CHECK(A)]

手順（XX-1）：{ 初期化 }

ダブル配列の未使用要素を格納する変数 e_index に E_HEAD をセットする．

手順（XX-2）：{ インデックス検索 }

e_index を c_1 の遷移先として使用するために，インデックス q に $e\_index - c_1$ をセットする．$c \in A$ なるラベル c すべてに対し，$\mathrm{CHECK}[q + \mathrm{N}(c)] < 0$ を満足し，かつ $q > 1$ であれば，q を返し，終了する．満足しなければ，$e\_index = -\mathrm{CHECK}[e\_index]$ により次の未使用要素をセットし，手順（XX-3）へ．

手順（XX-3）：{ 終了判定 }

$e\_index \le DA\_SIZE$ であれば，手順（XX-2）へ．$e\_index > DA\_SIZE$☆であれば，$e\_index - c_1$ を返し，終了する．（関数終）

従来の X_CHECK との違いは，手順（XX-2）で，文字 c に対する調査が失敗したときに $e\_index = -\mathrm{CHECK}[e\_index]$ によって直接次の未使用要素に移動することで無駄な走査を省いていることである．

3.3 未使用要素リストの更新

未使用要素リストの更新は，ダブル配列の CHECK 値の変更と同時に行われるので，その新しい関数 W_CHECK を次に示す．この関数は，$index > DA\_SIZE$ の場合にダブル配列の拡張処理（手順（W-1））を行っておき，インデックス $index$ や設定する値 val によって未使用要素リストに対する処理を決定し（手順（W-2）），未使用要素リストからの削除（手順（W-3a），（W-3b）），と挿入（手順（W-4a），（W-4b））を行う．

[関数 W_CHECK($index, val$)]

手順（W-1）：{ ダブル配列のサイズが増えたときの処理 }

$index > DA\_SIZE$ の場合，$DA\_SIZE < e\_index < index$ なるインデックス e_index は未使用要素となるので，未使用要素リストに挿入し，DA_SIZE を $index$ に拡張する．また，この場合

☆ この場合は，ダブル配列のサイズを拡張してからすべての遷移を格納する．

は val が格納されるので，CHECK[$index$] に val をセットし，終了する．

手順（W-2）：{ 処理の決定 }

CHECK[$index$] が負の場合，val が格納されるので，未使用要素リストから削除するため，手順（W-3a）へ．また，val が 0 の場合，CHECK[$index$] が未使用要素になるので，未使用要素リストに挿入するため，手順（W-4a）へ．それ以外（CHECK[$index$] と val が正の場合）は，CHECK[$index$] を再定義する場合であるので，CHECK[$index$] に val をセットし，終了する．

手順（W-3a）：{ 未使用要素リストの先頭を削除 }

$index \neq E\_HEAD$ の場合，手順（W-3b）へ．$index = E\_HEAD$ の場合，未使用要素リストの先頭が使用されるので，E_HEAD を $-$CHECK[$index$] に変更して，CHECK[$index$] に val をセットし，終了する．

手順（W-3b）：{ 未使用要素リストから削除 }

未使用要素リストを再定義するため，$index = -$CHECK[$prev\_index$] なる $prev\_index$ を未使用要素リスト上で発見し，CHECK[$prev\_index$] を CHECK [$index$] に変更して，CHECK[$index$] に val をセットし，終了する．

手順（W-4a）：{ 未使用要素リストの先頭へ挿入 }

$index \geq E\_HEAD$ の場合，手順（W-3b）へ．$index < E\_HEAD$ の場合，$index$ が未使用要素リストの先頭になるので，CHECK[$index$] に $-E\_HEAD$ をセットし，E_HEAD を $index$ に変更して終了する．

手順（W-4b）：{ 未使用要素リストへ挿入 }

E_HEAD から未使用要素リストをたどり，$prev_index < index < -$CHECK[$prev\_index$] なる $prev\_index$ を発見し，CHECK[$index$] に CHECK [$prev\_index$] をセットする．また，CHECK[$prev\_index$] を $-index$ に変更し，終了する．（関数終）

W_CHECK の変更にともない，関数 W_BASE も変更する必要があるが，W_BASE については未使用要素リストを直接操作することはないので，W_CHECK の手順（W-1）の CHECK を BASE に変更した動作を行うように変更する．

[例 7] 図 3 において，W_CHECK(31,7) が呼び出された場合の例を示す．まず，手順（W-1）で，$31 < DA\_SIZE = 39$ であるので，手順（W-2）で，CHECK[31] $= -32 < 0$ であるので，インデックス番号 31 を未使用要素リストから削除する．手順（W-3a）で，$31 \neq E\_HEAD = 28$ であるので，手順（W-3b）で，$31 = -$CHECK[$prev\_index$] $= -$CHECK[29] より，インデックス番号 31 の前の未使用要素，インデックス番号 29 を得，CHECK[29] = CHECK[31] $= -32$ としてインデックス番号 31 を未使用要素リストから削除し，CHECK[31] = 7 により値を設定して，終了する．（例終）

4. 動的削除アルゴリズムと記憶量の圧縮法

削除における問題点である不要ノードの削除法と未使用領域の圧縮法を示す．

4.1 不要ノード削除法

不要ノードの削除は，キーの削除時に葉ノード $index$ から遷移元を保持し（手順（K-1）），$index$ を削除後（手順（K-2）），遷移元が不要ノードであれば削除することで（手順（K-3））実現できるので，関数 DELETE を次のように変更する．なお，不要ノードは 2.3 節で説明したように，遷移の分岐を持たないノード 18，19，21 であるが，説明を簡単にするために葉ノード 22 も不要ノードとして扱う．

[関数 DELETE($index$)]

手順（K-1）：{ 遷移元の保持 }

$parent\_index =$ CHECK[$index$] により，不要ノード $index$ に対する遷移元のノード $parent\_index$ を保持する．

手順（K-2）：{ 不要ノードの削除 }

W_BASE($index$,0)，W_CHECK($index$,0) により不要ノード $index$ を未使用要素とする．

手順（K-3）：{ 遷移元が不要ノードか調査 }

ノード $parent\_index$ から出るすべての遷移ラベル a に対し，$t =$ BASE[$parent\_index$] + N(a)，CHECK[t] $= parent\_index$ なる CHECK[t] が存在しない場合，$parent\_index$ は不要ノードとなるので，$index$ に $parent\_index$ をセットし，手順（K-1）へ．CHECK[t] が存在する場合は，削除する不要ノードがないので終了する．（関数終）

[例 8] 図 3 において，キー “$beach\#$”を削除する例を示す．まず，検索アルゴリズムの手順（D-1），（D-2），（D-3）より，$index = 22$ で検索が成功するので，DELETE(22) が呼び出され，手順（K-1）で，$parent\_index$ に CHECK[$index$] = CHECK[22] = 21 を得る．手順（K-2）で，ノード 22 を未使用要素とする．手順（K-3）で，ノード $parent\_index = 21$ から遷移するノードは存在しないので，$index = parent\_index = 21$ とし，手順（K-1）へ．以後，同様にノード 21，19 を削除し，$index = 18$ において，$parent\_index =$ CHECK[$index$] = CHECK[18] = 7 を得，ノード 18 を削除する．手順（K-3）で，ノー

ド $parent\_index = 7$ から遷移するノードは 37 と 39 があるので終了する．（例終）

4.2 未使用要素圧縮法

未使用要素の除去は，関数 DELETE の終了時に次の関数 DEL_ELEMENT を実行することで実現する．この関数はダブル配列の最後方に位置するノードを前方に移動できるか否かを確認し（手順（E-1),(E-2)），移動可能であれば移動する（手順（E-3））．これにより，ダブル配列中の未使用要素は後方（インデックスの大きい方）に移動し，この未使用要素を除去することによって（手順（E-4)），ダブル配列のサイズを圧縮させる．

[関数 DEL_ELEMENT()]

手順（E-1）：{ 移動対象のノード設定 }

ダブル配列中の未使用要素でない最大のインデックス $maxindex$ に対し，移動対象のノードを保持する変数 m_index にノード $maxindex$ の遷移元 CHECK[$maxindex$] をセットする．

手順（E-2）：{ 移動後のサイズ調査 }

m_index から出る遷移ラベルの集合を GET_LABEL(m_index) により変数 R にセットする．BASE[m_index] $\leq$ X_CHECK(R) であれば，現在の格納位置より前方に移動できないので，手順（E-4）へ．BASE[m_index] $>$ X_CHECK(R) であれば，手順（E-3）へ．

手順（E-3）：{ ノードの移動 }

MODIFY(m_index, R, ϕ) によって，m_index から遷移するノードを移動する．

手順（E-4）：{ ダブル配列のサイズ再設定 }

ダブル配列中の未使用要素でない最大インデックスを DA_SIZE にセットし，終了する．（関数終）

[例 9] 例 8 においてキー “*beach#*”を削除後の関数 DEL_ELEMENT の動作を示す．まず，手順（E-1）で，未使用要素でない最大のインデックス 39 に対し，m_index = CHECK[39] = 7 により遷移元をセットする．手順（E-2）で，R = GET_LABEL(7) = {‘t’,‘v’} より遷移ラベルの集合を得る．X_CHECK({‘t’,‘v’}) = 8 < BASE[7] = 16 よりノードを移動できるので，手順（E-3）で，MODIFY(7,{‘t’,‘v’},ϕ) によってノード 7 から遷移するノード 37，39 は，ノード 29，31 に移動する．最後に手順（E-4）で，未使用要素でない最大のインデックス 31☆を DA_SIZE にセットし，終了する．図 3 において，キー “*beach#*”を削除後のトライとダブル配列の例を**図 4** に示す．図中の下線部が更新された箇所である．（例終）

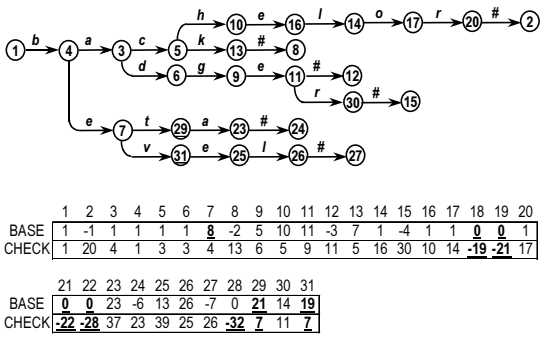

図 4 キー “*beach#*” 削除後のトライとダブル配列の例
Fig. 4 Examples of a trie and a double-array after deletion of key “*beach#*.”

5. 評　価

5.1 理論的評価

ダブル配列によるトライの遷移検索の最悪時間計算量は $O(1)$ であるので，検索キーの長さを k とするとダブル配列によるトライ検索の最悪時間計算量は $O(k)$ となる[11]．

提案手法でのダブル配列への追加の最悪時間計算量は，関数 MODIFY，X_ CHECK，W_CHECK に依存する．トライの遷移ラベルの最大数を e，ダブル配列の使用要素数を n，未使用要素数を m とする☆☆とき，関数 W_CHECK については，手順（W-2b),(W-3b）で未使用要素の位置を決定するためにすべての未使用要素をたどる必要があるので $O(m)$ となる．関数 X_CHECK については，手順（XX-2),(XX-3）で 2 重ループ構造となっており，手順（XX-2）で e 回，手順（XX-3）で m 回繰り返すので，計算量は $O(m \cdot e)$ となる．関数 MODIFY については，手順（M-1）で関数 X_CHECK を使用し，手順（M-2),(M-3）で e 回の 2 重ループ構造となっており，ともにループ内で関数 W_CHECK を使用する．ただし，手順（M-3）での W_CHECK は再定義のため $O(1)$ となるので，計算量は $O(m \cdot e + m \cdot e^2)$ となる．

ここで，従来手法での追加の最悪時間計算量を考察すると，関数 W_CHECK は値の設定のみなので $O(1)$，関数 X_CHECK は，手順（X-2）で e 回，手順（X-3）で $n+m$ 回繰り返すので，$O((n+m) \cdot e)$，関数 MODIFY は，$O((n+m) \cdot e + e^2)$ となる．した

☆ 手順（E-1）での最大インデックス 39 は，移動されたため未使用要素になっている．

☆☆ すなわち，$DA\_SIZE = n + m$ となる．

表 1 実験結果
Table 1 The simulation results.

登録キー数	10,000	20,000	30,000	40,000	50,000	60,000	70,000	80,000	90,000	100,000
総ノード数 n	67,266	108,918	150,128	191,213	232,146	272,434	312,935	353,011	392,054	429,283
未使用要素数 m	1	11	5	3	5	6	1	1	4	9
追加時間 (ms)										
従来手法	8.6	15.3	21.2	26.7	32.0	37.1	42.0	47.0	51.7	55.6
提案手法	0.044	0.039	0.037	0.036	0.036	0.036	0.035	0.035	0.035	0.035
削除時間 (ms)	0.46	0.46	0.35	0.33	0.32	0.30	0.29	0.28	0.28	0.26

表 2 未使用要素数に対する追加時間の実験結果
Table 2 The simulation results for insertion depending on empty elements.

未使用要素の割合 m/DA_SIZE(%)	10	20	30	40	50	60	70	80	90
追加時間 (ms)	0.037	0.033	0.034	0.033	0.034	0.034	0.034	0.036	0.033

がって，従来手法における追加の最悪時間計算量は，e は定数となるので，DA_SIZE に依存することになる．特に，n は使用要素数，すなわちトライにおける総ノード数となるので，従来手法は追加するキー数が多くなるに従って総ノード数も増加し，追加時間が大幅に低下する．

提案手法については未使用要素数 m に依存するが，m の値が大きくなると，ダブル配列がスパースになるので，関数 X_CHECK や W_CHECK で未使用要素をすべてたどる確率が減ると考えられる．また，キー数には依存しないので，非常に高速な追加が可能となる．

削除の最悪時間計算量は，関数 DELETE，関数 DEL_ELEMENT に依存する．関数 DELETE は，1 つのノードの削除が関数 W_CHECK に依存し，手順（K-3）で遷移元の調査を e 回繰り返すので，$O(m \cdot e)$ となる．関数 DEL_ELEMENT は関数 MODIFY に依存しているので，$O(m \cdot e + m \cdot e^2)$ となる．したがって，削除の最悪時間計算量は，未使用要素数 m に依存し，追加と同様キー数に依存しなくなる．

5.2 実験による評価

本手法の構成システムは約 1,000 行の C++言語で記述されており，DELL Precision 410（OS: Windows 2000，CPU: PentiumII[400 MHz]）上で稼動している．遷移ラベルの総数 e は，1 バイトで表現できる数 256 に終端記号を加えた 257 である．また，実験に用いたデータは，EDR 電子化辞書[13] の英語単語辞書である．

本論文での提案手法は，キーの追加，削除に対する改善手法であり，これらの手法を導入することによって検索アルゴリズムに変更が加えられることはないので，検索時間の評価は行わない．また，キーの削除時間に対する評価については，従来の削除法が不要ノードや未使用要素の処理を行わないので，比較実験はできない．したがって，提案手法のみ評価を行う．

表 1 に登録キー数を変化させた場合の実験結果を示す．表中の追加時間は 1 つのキーに対する平均時間を表している．表 1 の結果から，提案手法は約 190〜1,600 倍高速に追加できることが分かる．また，従来手法の追加時間は，ダブル配列の要素数に依存するので，追加キー数（すなわち総ノード数）が増加するにつれて追加時間も増加している．しかし，未使用要素数にのみ依存する提案手法ではほぼ一定の値を示していることが分かる．

次に，提案手法についてのみ，未使用要素数を変化させた場合の追加時間の評価を行った．**表 2** の実験結果は，10 万件のキーを登録後，未使用要素の割合が 10%〜90%になるまでキーを削除し，その後 1 万件のキーを追加したときの 1 件あたりの追加時間を示している．10 万件のキーを登録したときの総ノード数は表 1 と同じ 429,283 であり，キーの削除は未使用要素数を増やすために関数 DELETE のみを使用し，関数 DEL_ELEMENT による圧縮は行っていない．表 2 の結果から，未使用要素の割合に関係なく一定の値を示しており，非常に高速に追加されていることが分かる．これは，ダブル配列がスパースになることによって関数 X_CHECK や W_CHECK で未使用要素をすべてたどる確率が減ったためである．

表 1 に示した削除時間は，登録キー数を変化させた場合のキー登録後，1,000 件のキーを削除した場合の 1 件あたりの時間である．表 1 の結果から，提案手法は登録キー数に関係なく一定の値を示していることが分かる．また，削除時間についても 1 件あたり約 0.35 ms であり高速であることが分かる．

次に，多量の削除が生じた場合の空間使用率の変化を従来手法と比較する．**図 5** に 10 万件登録後，徐々

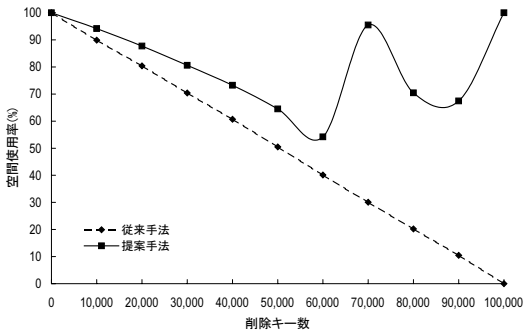

図 5 削除時の記憶量に対する実験結果
Fig. 5 The simulation results for storages after deletion.

に削除していった場合のダブル配列の空間使用率の実験結果を示す．図中の空間使用率はダブル配列の全要素数に対する使用要素数の割合である．図 5 の結果から，従来手法では削除キーに比例して空間効率は悪くなるが，提案手法では大量の削除が起こった場合でも 50%以上の使用率を維持していることが分かる．ここで，削除キー数が 6〜7 万件と，9〜10 万件のときに空間使用率が大幅に増加している点について考察する．削除時の圧縮は，ダブル配列の後方に位置するノードが前方に移動可能であれば，前方に移動することによって圧縮されるが，前方への移動が不可能であれば圧縮が行われずに空間使用率が低下する．逆に，この状態が長く続き，空間使用率が大きく低下すると，ダブル配列がスパースになり，後方に位置するノードが前方に移動しやすくなる．このため，ダブル配列のサイズが一気に圧縮され，空間使用率が大幅に増加することになる．

以上より，提案手法はダブル配列の動的更新アルゴリズムとして有効であるといえる．

6. おわりに

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問題となっていた追加時間を高速化する手法を提案し，削除時に生ずる不要ノードや未使用要素の無駄を解消する手法を提案した．また，実験による評価により，本手法の有効性を実証した．

今後の課題としては，動的検索を必要とする実用システムに組み込み，有効性を確認することである．

参考文献

1) 青江順一：キー検索技法—トライ法とその応用，情報処理学会誌，Vol.34, No.2, pp.244–251 (1993).
2) Fredkin, E.: Trie Memory, *Comm. ACM.*, Vol.3, No.9, pp.490–500 (1960).
3) Knuth, D.E.: *The Art of Computer Programming*, Vol.3, Sorting and Searching, ch.6, Addison-Wesley, Reading, MA (1973).
4) Aho, A.V., Sethi, R. and Ullman, J.D.: *Compilers—Principles, Techniques, and Tools*, chapter 3, 4, Addison-Wesley, Reading Mass. (1986).
5) Lesk, M.E.: Lex—A lexical analyzer generator, *CSTR 39*, pp.1–13, Bell Lab., N.J. (1975).
6) Aho, A. and Corasick, M.: Efficient String Matching: An Aid to Bibliographic Search, *Comm. ACM*, Vol.18, No.6, pp.333–340 (1975).
7) Peterson, J.L.: *Computer Programs for Spelling Correction*, Lecture Notes in Comput. Sci., Springer-Verlag, New York (1980).
8) 永田昌明：岩波講座言語の科学〈3〉単語と辞書，chapter 2, 岩波書店 (1997).
9) 吉村賢治：自然言語処理の基礎，chapter 6, サイエンス社 (2000).
10) Aoe, J., Morimoto, K., Shishibori, M. and Park, K.-H.: A Trie Compaction Algorithm for a Large Set of Keys, *IEEE Trans. Knowledge and Data Engineering*, Vol.8, No.3, pp.476–491 (1996).
11) 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.4, pp.1592–1600 (1988).
12) Aoe, J.: An efficient digital search algorithm by using a double-array structure, *IEEE Trans. Softw. Eng.*, Vol.SE-15, No.9, pp.1066–1077 (1989).
13) 日本電子化辞書研究所：EDR 電子化辞書 (1996).

(平成 12 年 8 月 21 日受付)
(平成 13 年 6 月 19 日採録)

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．

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．自然言語処理学会会員．

大野　将樹（学生会員）

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．現在同大学院博士前期課程在学中．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．同年徳島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，人工知能学会，日本認知科学会，日本機械翻訳協会，IEEE，ACM，AAAI，ACL 各会員．