DEWS2007 C1-1

ダブル配列によるパトリシアを拡張した基数探索法の提案

中村 康正† 望月 久稔†

† 大阪教育大学教育学部 〒 582-8582 大阪府柏原市旭ヶ丘 689-1
E-mail: j059610@ex.osaka-kyoiku.ac.jp, motizuki@cc.osaka-kyoiku.ac.jp

あらまし 木構造で表現される基数探索法は，共通接頭辞探索などが容易であるため，自然言語処理などを中心に広く用いられている．基数探索法の探索処理を高速化するため，木構造において遷移が 1 つしか存在しない分岐を圧縮したパトリシアや，木構造の遷移数を抑制するために多分木としたマルチウェイ基数探索法がある．また，マルチウェイ基数探索法のデータ構造として，節点間の遷移を定数時間で決定できる高速性をもつダブル配列がある．本論文では，ダブル配列によりパトリシアを拡張した基数探索法を提案する．評価実験の結果，提案手法は探索処理や更新処理において比較手法よりも有効であるとわかった．

キーワード 情報検索，ダブル配列，パトリシア，トライ

Proposal of Radix Search Method Extended Patricia based on the Double-array

Yasumasa NAKAMURA† and Hisatoshi MOCHIZUKI†

† Osaka Kyoiku University Asahigaoka 689–1 Kashiwara-shi, Osaka, 582-8582 Japan
E-mail: j059610@ex.osaka-kyoiku.ac.jp, motizuki@cc.osaka-kyoiku.ac.jp

Abstract Radix search method is used widely, such as dictionary information construction of natural language processing system. Patricia and multiway radix search method is proposed in order to accelerate search processing. The double-array structure is an efficient data structure combining fast access with compactness. In this paper, we presents radix search method based on the double-array structure. The simulation results turned out that the presented method is more effective than the original methods.

Key words Information retrieval, Double-array, Patricia, Trie

1. まえがき

検索技法である基数探索法（radix search）は，キーのある小さな部分を各遷移とする木構造で表現される．よって，キー数が膨大であっても探索処理は高々キー長である．また，共通接頭辞探索（prefix searching），範囲検索（range searching），類似検索（proximity searching）などが容易である特徴をもつ [2]．そのため，自然言語処理システム，パターンマッチングなどの辞書構築を中心に広く用いられる．基数探索法の手法として，離散探索法（digital search）とマルチウェイ基数探索法（multiway radix search）がある．

離散探索法はキーの各 1 ビットを遷移とした二分木で表現する．そのため，木構造上に遷移先がただ 1 つである一方向分岐が多く存在する．そこで，木構造上の遷移数を抑制するため，一方向分岐を削除したパトリシア（patricia）がある [3].

マルチウェイ基数探索法はキーの複数ビットを遷移とした多分木で表現するため，離散探索法に比べて任意のキーに対する遷移数が少ない．この手法を実現する有効なデータ構造として，高速性とコンパクト性をあわせもつダブル配列がある [1]．ダブル配列は，準静的なキー集合に対して提案された手法であるため，近年では更新処理に対する提案がなされている [4], [6]．また，二分木のパトリシアをダブル配列で表現し，使用空間を抑制した手法 [5] がある．ここで，半無限文字列を対象とした辞書について，マルチウェイ基数探索法で構築した場合でも木構造上に一方向分岐が多く存在する．

そこで本論文では，マルチウェイ基数探索法における木構造上の遷移数を抑制して探索処理を高速化するため，ダブル配列を用いてパトリシアを多分木に拡張した基数探索法を提案する．評価実験を行った結果，提案手法は探索処理や更新処理において比較手法よりも有効であるとわかった．

以下，2. で基数探索法とダブル配列を説明し，3. でパトリシアを拡張した基数探索法を提案する．4. で提案手法の評価を与え，5. で本論文のまとめと今後の課題についてふれる．

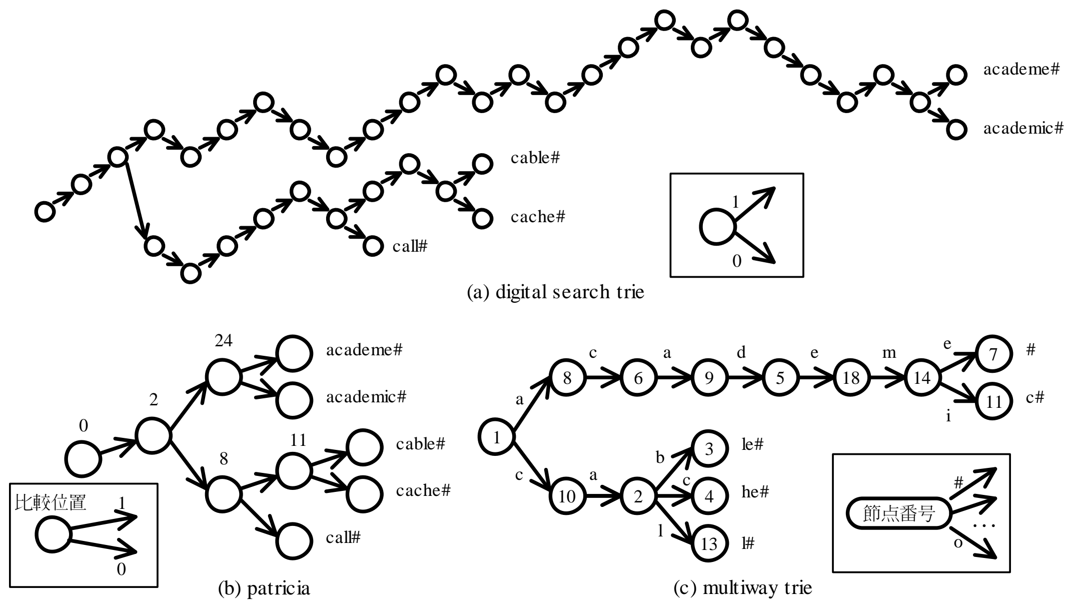


図 1　トライの例
Fig. 1　The example of trie.

2. 基数探索法とダブル配列

本章では，基数探索法である離散探索法とマルチウェイ基数探索法について述べる．さらに，マルチウェイ基数探索法を実現するダブル配列を説明する．また，例として表記記号 ‘#’ を終端記号としたキー集合 K ={“academe#”，“academic#”，“cable#”，“cache#”，“call#”} を用い，後の章でも対応させて使用する．ここで，表記記号 ‘#’，‘a’，···，‘o’ の内部表現値を 0，1，···，15 とし，各内部表現値は 4 ビットで表現可能とする．

2.1　基数探索法

検索技法である基数探索法は，キーのある小さな部分を遷移とする木構造で表現される．基数探索法に対して，ハッシュ法や二分探索木など他の検索技法はキー全体を比較対象とする．つまり，基数探索法の探索処理は高々キー長であり，他の検索技法はキー数とキー長に依存する．

基数探索法のうち，木構造の葉で探索キーと葉に対応したキーを比較するものをトライ（trie）という．また，基数探索法には，キー全体をトライとして実現する方法と，他のキーと共有している遷移列（以下，接頭辞）をトライとして実現する方法がある．後者の探索処理は，まずトライ上を根から葉まで遷移し，次にトライ上で実現されていない遷移列（以下，接尾辞）とトライ上で遷移を確認していない残りの探索キーを比較する．そのため，葉に接尾辞を格納する．本論文では，接頭辞をトライとして実現する方法について述べる．

基数探索法を実現する手法として，離散探索法とマルチウェイ基数探索法がある．前者はキーの各 1 ビットを遷移とした二分木を用い，後者はキーの複数ビットを遷移とした多分木を用いて実現する．そのため，離散探索法は各節点に不要な遷移情報が少なく，実現も容易である．しかし，任意のキーに対する二分木上の遷移数は多分木上の遷移数よりも多い．一方，マルチウェイ基数探索法は二分木よりも遷移数が少ないが，任意の節点からすべての遷移種に対する遷移が存在するわけではないので不要な遷移情報が多い．

例 1: キー集合 K に対する離散探索法とマルチウェイ基数探索法を表現したトライをそれぞれ図 1(a)，(c) に示す．ここで各トライの遷移について，離散探索法では内部表現値を 4 ビットの二進数で表現したものに対して 1 ビット毎とし，マルチウェイ基数探索法では表記記号毎とした．また，接尾辞をトライ上の葉の右側に示すが，離散探索法に対してはキーを示す．(例終了)

図 1(a) からも分かるように，離散探索法には一方向分岐が多く存在する．パトリシアは，一方向分岐を作成せず，各節点にキーの比較位置情報を格納した手法である [3]．つまり，キー間で異なる遷移のみで木を構築する．よって，根を除くすべての節点には不要な遷移情報が存在せず，空間効率が良い．また，一方向分岐による遷移を抑制しているため，離散探索法に比べて任意のキーに対するトライ上の遷移数が少ない．

パトリシア上の葉には，接尾辞ではなく葉に対応したキーを格納する．これは，離散探索法やマルチウェイ基数探索法を実現したトライとは異なり，パトリシアがすべての接頭辞を表現していないためである．よって，パトリシアの探索処理は，パトリシア上を根から葉まで遷移し，葉に対応するキーと探索キーを比較する．

例 2: キー集合 K に対するパトリシアを図 1(b) に示す．ここで，節点の上に示す値はキーの比較位置である．

図 1 について，キー “academic#” の探索における遷移数を比較する．離散探索法，パトリシア，マルチウェイ基数探索法におけるトライ上の遷移数は，それぞれ 25，3，7 である．また，キー “cable#” を探索する場合はそれぞれ 12，4，3 である．よって，離散探索法とマルチウェイ基数探索法では多分木とした後者の遷移数が少なく，離散探索法の遷移数を抑制した

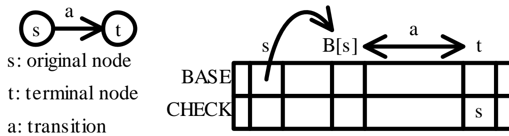


図 2　ダブル配列によるトライの遷移関係

Fig. 2　Transition of the trie by the double-array.

パトリシアは，離散探索法よりも遷移数が少ない．ここで，マルチウェイ基数探索法はキー長に，パトリシアはキー数に依存する．

使用空間について，離散探索法，パトリシア，マルチウェイ基数探索法におけるトライ上の節点数は，それぞれ 39，10，14 である．よって，パトリシアは離散探索法より節点数が少なく，トライ上の使用空間が小さい．また，マルチウェイ基数探索法は離散探索法より節点数は少ないが，遷移の決定を高速化するためには遷移情報を多くもつ必要がある．よって，節点毎の使用空間は離散探索法よりも大きい．

また，離散探索法とマルチウェイ基数探索法が接尾辞のみを葉に格納するのに対し，パトリシアはキー全体を葉に格納する．よって，キー情報に関する使用空間はパトリシアが大きい．

(例終了)

2.2　ダブル配列のデータ構造

ダブル配列は，BASE と CHECK という 2 つの一次元配列を用いてトライの節点を実現する．配列 BASE は遷移の基底位置を与え，配列 CHECK はトライ上の親子関係を決定する．つまり，図 2 に示す始点 s から遷移 a での終点 t を式 (1)，式 (2) で定義する [1]．また，トライ上の節点番号とダブル配列上の要素番号は対応している．ここで，表記記号の内部表現値を単に a とし，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．

$$B[s] + a = t \tag{1}$$

$$C[t] = s \tag{2}$$

トライ上の節点として使用していないダブル配列上の要素（以下，未使用要素）について，更新処理を高速化するために循環双方向リスト（以下，E-LINK）として連結する [4], [6]．

トライ上の葉について，ダブル配列では接尾辞を一次元配列 TAIL に格納する [1]．この際，葉の BASE 値を配列 TAIL の要素番号に設定する．さらに，他の節点と区別するために BASE 値を負値とする．以下，配列 TAIL の pos 番目の要素を $T[pos]$ とし，$T[pos]$ から始まる遷移列を $T + pos$ とする．

例 3: キー集合 K に対するダブル配列を図 3 に示す．図 1(c) の節点番号は図 3 の要素番号と対応している．また，本論文では混乱を避けるため，E-LINK として使用する未使用要素の BASE 値と CHECK 値を空白とする．

(例終了)

3.　パトリシアを拡張した基数探索法

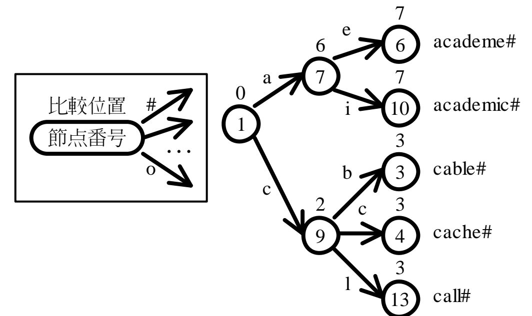
多分木で表現されるマルチウェイ基数探索法は，トライ上に

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

	7	8	12	15	18
TAIL	#	c#	le#	he#	l#

図 3　ダブル配列の例

Fig. 3　The example of the double-array.



	1	2	3	4	5	6	7	8	9	10	11	12	13
BASE	6		-18	-24		-1	1		1	-9			-30
CHECK			9	9		7	1		1	7			9
POS	0		3	3		7	6		2	7			3

	1	9	18	24	30
TAIL	academe#	academic#	cable#	cache#	call#

図 4　提案手法におけるトライとダブル配列の例

Fig. 4　The example of the trie and the double-array in the proposal technique.

一方向分岐が存在する．そこで，探索速度を向上させるため，二分木で表現されるパトリシアを多分木に拡張する．また，効率良く拡張するため，ダブル配列にキーの比較位置情報を付加することを提案する．

以下，提案手法のデータ構造と探索手法を説明する．加えて，提案手法の更新処理について述べる．

3.1　データ構造

ダブル配列にキーの比較位置を格納する配列 POS を新たに用いる．キー集合 K に対して，提案手法が実現するトライとダブル配列を図 4 に示す．

配列 POS を用いることにより，キー key に対する始点 s からの終点 t は式 (3) となる．以下，ダブル配列上の要素 x に関する POS 値を $P[x]$ とする．また，key の pos 番目の要素を $key[pos]$ とし，$key[pos]$ から始まる遷移列を $key + pos$ とする．

$$B[s] + key[P[s]] = t \tag{3}$$

また，提案手法が実現するトライはダブル配列とは異なり接頭辞をすべて実現していない．よってパトリシアと同様に，配列 TAIL にキー全体を格納する．

例 4: 図 4 における使用空間について，提案手法が実現するトライの節点数は 8 であり，他の基数探索法よりも少ない．ここで，提案手法は各節点にキーの比較位置情報を付加したため，ダブル配列よりも節点毎の使用空間が大きい．また，葉に対応した遷移列に関する使用空間はパトリシアと同等である．

Function: Search(t, key)

```
(S1)  while: 節点 t が葉ではない
(S2)      s = t;
(S3)      t = B[s] + key[P[s]];
(S4)      if: C[t] と s が異なる
(S5)          return FALSE;
(S6)  return EqualKey(T + (−B[t]), key);
```

図 5　関数：Search
Fig. 5　Function: Search.

(例終了)

3.2　探索アルゴリズム

提案手法におけるキーの探索が成功する条件は，式 (3) と式 (2) を満足させながらトライ上の根から葉まで遷移し，葉に対応したキーと探索キーが一致することである．

節点 t を根とする部分木から key を探索する関数 Search を図 5 に示す．まず図 5 の S3 で，key に対する始点 s からの終点を配列 POS を用いて決定する．次に，S4 では式 (2) により遷移が存在するかを判断し，存在すれば S1 から始まる while 文により葉まで遷移する．その後，S6 で遷移列 $T+(-B[t])$ とキー key が一致するかを関数 EqualKey により判断する．関数 EqualKey(x, y) は，遷移列 x と y が等しければ TRUE を，等しくなければ異なる最小の比較位置を返す．

例 5: 図 4 から key として “academic#” を探索する．まず終点 t を根 1 とし，図 5 の S1 で t が葉でないことを確認する．次に S2 で始点 s を $t=1$ に設定し，S3 で s から比較位置 $P[1]=0$ である ‘a’ で遷移する終点 t を決定する．つまり，式 (3) より t を $B[s]+a=6+1=7$ に設定する．その後，式 (2) を満足させるため，S4 で $C[t]=1$ が $s=1$ と等しいことを確認する．これらにより，始点 1 から ‘a’ での終点が 7 であることがわかった．

同様に，式 (3)，式 (2) を確認しながら，始点 7 から $key[P[7]=6]$ である ‘i’ で終点 10 に遷移する．終点 10 は葉であるので，S1 から始まる while 文を終了し，S6 で葉 $t=10$ に対応した遷移列 $T+(-B[t])=T+9=$“academic#” と $key=$“academic#” を比較する．葉に対応した遷移列と key が等しいので，探索成功として TRUE を返す．以上より，“academic#” を探索した場合におけるトライ上の遷移数は 2 である．よって，他の基数探索法よりも提案手法の遷移数が少ない．

同様に，提案手法において “cable#” を探索した場合におけるトライ上の遷移数は 2 である．この場合も，他の基数探索法より提案手法の遷移数が少ない．

次に “caching#” を探索する．先程と同様に，トライ上を根 1 から 9，4 に遷移し，葉 4 に対応する遷移列 $T+(-B[4])=$“cache#” と $key=$“caching#” を比較する．比較位置 4 で異なるので探索失敗として 4 を返す．この場合，トライは 0 から 2 までの異なる比較位置を表現する．つまり，従来のマルチウェイ基数探索法と同様に接尾辞での探索失敗となる．

また “analysis#” を探索する．先程と同様に，トライ上を根 1 から 7，10 に遷移し，葉 10 に対応する遷移列 $T+(-B[10])=$“academic#” と $key=$“analysis#” を比較する．比較位置 1 で異なるので探索失敗として 1 を返す．この場合，トライは 0 から 6 までの異なる比較位置を表現する．つまり，従来のマルチウェイ基数探索法ではトライ内の比較位置 1 で探索が失敗する．

最後に “account#” を探索する．先程と同様に，トライ上を根 1 から 7 に遷移し，式 (3) より始点 7 から $key[P[7]=6]$ である ‘t’ での終点 $B[7]+t=26$ を決定する．その後，S4 で未定義である $C[26]$ が $s=7$ とは異なることがわかる．よって，始点 7 から ‘t’ での終点は存在せず，S5 で探索失敗として FALSE を返す．この場合，トライが表現しているキーに対して，比較位置 1 から 6 までに key とは異なる比較位置が存在する．

(例終了)

3.3　追加アルゴリズム

キー key に対する追加処理は，key の探索が失敗した場合に行う．提案手法においてキー探索が失敗するのは，図 5 の S5 で始点 s から $key[P[s]]$ での終点 t が存在しない場合と，S6 で葉に対応するキーと key が一致しない場合である．追加処理で使用する変数と関数を以下に示す．

$maxTailPos$	：配列 TAIL の使用済み最大要素番号
$\lvert A\rvert$	：集合 A の要素数
L，L_s，$L_{s'}$	：遷移集合

関数 GetLeaf(s)
始点 s を根とする部分木における葉を返す．

関数 GetLabels(s, L)
始点 s からの遷移を遷移集合 L に格納する．

関数 InsNode(x)
節点 x を作成するため，E-LINK から x を削除する．加えて，E-LINK の拡大を行う．

関数 MakeLeaf(t, key)
節点 t を葉とするために，$B[t]$ を $-(maxTailPos+1)$ に設定し，$T[maxTailPos+1]$ からキー key を設定する．

関数 NewBase(L)
E-LINK を走査し，L の全要素が遷移可能な BASE 値 $base$ を返す．遷移 a が可能となるかは，終点 $base+a$ がダブル配列上の未使用要素であることと同値である．

関数 TransNode(s, $newBase$, L)
始点 s の BASE 値を新たな BASE 値 $newBase$ に変更し，式 (1) を満足させるために s の終点を移動する．また，式 (2) を満足させるため，関数 RenewalCheck により移動した節点における終点の CHECK 値を更新する．

関数 RenewalCheck(new, old)
ダブル配列上の要素番号が old から new へ移動した節点の終点に対して，その CHECK 値を old から new に更新する．

提案手法は，図 6 に示す関数 Insert により追加処理を実現する．関数 Insert の入力は，関数 Search により遷移が成功した節点 $terminal$ と追加キー key である．$terminal$ は，S5 で探索が失敗した場合は s であり，S6 の場合は t である．

Function: Insert($terminal$, key**)**

(I1) **if:** 節点 $terminal$ が葉である
(I2) $leaf = terminal$;
(I3) **else:**
(I4) $leaf$ =GetLeaf($terminal$);
(I5) pos =EqualKey($T + (-B[leaf])$, key);
(I6) s をトライ上の根に設定する;
(I7) **while:** 節点 s が $terminal$ とは異なる
(I8) $t = B[s] + key[P[s]]$;
(I9) **if:** pos が $P[t]$ より小さい
(I10) **return** InsCommon(s, key, $T + (-B[leaf])$, $P[s]$, pos);
(I11) $s = t$;
(I12) **if:** 節点 s が葉である
(I13) InsCommon(s, key, $T + (-B[leaf])$, $P[s]$, pos);
(I14) **else:**
(I15) $t = B[s] + key[P[s]]$;
(I16) InsLeaf(s, t, key, pos);
(I17) **return**;

図 6 関数：Insert
Fig. 6 Function: Insert.

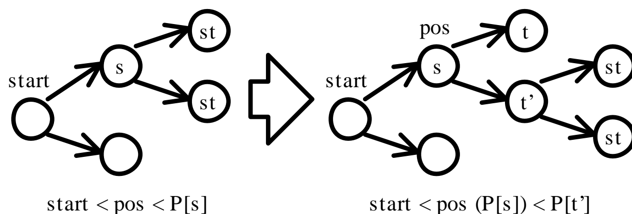


図 7 関数 InsCommon の概要
Fig. 7 The illustration of Function: InsCommon.

まず，図 6 の I5 で新規節点の比較位置 pos を得るため，key と比較する他のキーを調べる．そこで，$terminal$ が葉であれば $terminal$ から他のキーを得て，$terminal$ が葉でなければ $termnal$ を根とする部分木における葉 $leaf$ から他のキーを得る．

次に，I6 から I11 でトライ上の根から $terminal$ まで遷移しながら，新規節点の追加位置を決定する．ここで，提案手法は一方向分岐に対応する節点を作成していないため，始点 s から終点 t に至る遷移列内に新規節点の比較位置 pos が存在する場合がある．これは，pos が t における比較位置 $P[t]$ よりも小さい場合であり，I10 で図 8 に示す関数 InsCommon により s と t の間に節点を作成する．

新規節点の追加位置が既存の節点間ではない場合，$terminal$ までの遷移が正しいことが保証されている．よって，$terminal$ が葉であれば I13 で関数 InsCommon を呼び出し，$terminal$ から遷移 $key[pos]$ での終点が存在しなければ I16 で図 9 に示す関数 InsLeaf を呼び出す.

関数 InsCommon は，葉から接尾辞に対する節点を作成する場合と，既存の節点間に節点を作成する場合に用いる．後者はダブル配列とは異なるため，概要を図 7 に示す.

前者の場合，始点 s は葉である．まず図 8 の IC1 で $oldBase$ を，葉 s が指す遷移列 $tail$ を格納した配列 TAIL の要素番号 $B[s]$ に設定する．次に IC4，IC5 で s から $key[pos]$ による終点

Function: InsCommon(s, key, $tail$, pos**)**

(IC1) $oldBase = B[s]$; $oldPos = P[s]$;
(IC2) $P[s] = pos$;
(IC3) $B[s]$ =NewBase($\{key[pos], tail[pos]\}$);
(IC4) $t = B[s] + key[pos]$;
(IC5) $t' = B[s] + tail[pos]$; $B[t'] = oldBase$;
(IC6) **if:** t' が葉となる
(IC7) $P[t'] = pos + 1$;
(IC8) **else:**
(IC9) $P[t'] = oldPos$;
(IC10) RenewalCheck(t', s);
(IC11) InsNode(t); $C[t] = s$; $P[t] = pos + 1$;
(IC12) MakeLeaf(t, key);
(IC13) InsNode(t'); $C[t'] = s$;
(IC14) **return**;

図 8 関数：InsCommon
Fig. 8 Function: InsCommon.

t と，$tail[pos]$ による終点 t' を決定する．その後，IC11，IC13 で t，t' を作成し，IC12 で t を key に対応した葉とする．ここで，s からの遷移に対する比較位置は pos であるため，IC2 で $P[s]$ を pos に設定する．また，t，t' の POS 値は $pos+1$ であり，IC13 で $B[t']$ を $oldBase$ に設定しなおす.

後者の場合，まず IC1 で $oldBase$ を s の終点に対する BASE 値 $B[s]$ に，$oldPos$ を $P[s]$ に設定する．次に始点 s が葉である場合と同様に t，t' を作成し，$P[s]$ を pos に，$P[t]$ を $pos+1$ に設定する．ここで，節点 t' には比較位置 $oldPos$ での終点が存在するため，IC9 で $P[t']$ を $oldPos$ に設定し，IC10 で t' の終点における CHECK 値を関数 RenewalCheck により更新する.

関数 InsLeaf では，始点 s から $key[pos]$ での終点 t が未使用要素であるか未使用要素ではないかで処理が異なる．そこで，図 9 の IL1 で始点 s から $key[pos]$ での終点 t が未使用要素かを判断する．t が未使用要素である場合，IL11 で t を作成し，IL12 で t を葉とする.

t が未使用要素ではない場合，t に節点を作成できないために遷移を定義できない．これを衝突といい，回避処理には s の BASE 値を変更して t とは異なる未使用要素 t' を終点とする方法と，t の始点 s' の BASE 値を変更して t を未使用要素とする方法がある．これらの計算量は s，s' からの遷移種数に依存するため，IL4 で遷移種数が少ない方法を選択して追加処理の効率化をはかる [1]．衝突を回避した後の IL12 で，前者は t' に，後者は t に葉を作成する.

衝突の回避を行う IL2 から IL10 について，まず IL2 で s' を t の始点に設定し，IL3 で関数 GetLabels により s，s' からの遷移を遷移集合 L_s，$L_{s'}$ に得る．次に，IL4 で L_s，$L_{s'}$ の要素数を比較して衝突回避処理を選択し，関数 TransNode によりトライ上の節点を移動することで衝突を回避する.

3.4 削除アルゴリズム

提案手法の削除処理は，削除キーに対応する葉を削除する処理と，葉の削除により出現した一方向分岐を削除する処理で構成する．よって，提案手法は最大 2 つの節点を削除する．一方，ダブル配列の削除処理は最大で葉から根までの節点を削除する.

Function: InsLeaf(s, t, key, pos)

(IL1) **if:** t が未使用要素ではない
(IL2) $s' = C[t]$;
(IL3) GetLabels(s, L_s); GetLabels(s', $L_{s'}$);
(IL4) **if:** $|L_s| + 1 < |L_{s'}|$
(IL5) $base$ =NewBase($L_s \cup key[pos]$);
(IL6) TransNode(s, $base$, L_s);
(IL7) **else:**
(IL8) $base$ =NewBase($L_{s'}$);
(IL9) TransNode(s', $base$, $L_{s'}$);
(IL10) $t = B[s] + key[pos]$;
(IL11) InsNode(t); $C[t] = s$; $P[t] = pos + 1$;
(IL12) MakeLeaf(t, key);
(IL13) **return**;

図 9　関数：InsLeaf
Fig. 9　Function: InsLeaf.

Function: Delete($leaf$)

(D1) $s = C[leaf]$;
(D2) DelNode($leaf$);
(D3) t =IsOneWayBranch(s);
(D4) **if:** t が FALSE ではない
(D5) $B[s] = B[t]$;
(D6) **if:** t が葉ではない
(D7) $P[s] = P[t]$;
(D8) RenewalCheck(s, t);
(D9) DelNode(t);
(D10) **return**;

図 10　関数：Delete
Fig. 10　Function: Delete.

削除処理を実現する関数 Delete を図 10 に示す．また，削除処理で使用する関数を以下に示す．

関数 DelNode(x)
トライ上から x を削除するため，E-LINK に x を追加する．加えて，E-LINK の縮小を行う．
関数 IsOneWayBranch(s)
始点 s からの遷移が一方向分岐であれば s の終点を，一方向分岐ではなければ FALSE を返す．

削除キーに対応する葉を削除する処理は，図 10 の D2 で削除キーの探索で到達した葉 $leaf$ を削除することで実現する．葉の削除により出現した一方向分岐を削除する処理は，$leaf$ の始点 s から終点 t の遷移が一方向分岐である場合に行い，終点 t のみを削除する．

まず，D3 で s から t への遷移が一方向分岐であるかを判断し，真であれば D9 で t を削除する．このとき，t が葉であれば，D5 で $B[s]$ を t が指すキーにおける配列 TAIL の要素番号 $B[t]$ に設定する．t が葉でなければ，式 (1) と式 (2) を満足させるため，D5 で $B[s]$ を $B[t]$ に，D8 で t の終点における CHECK 値を s に更新する．さらに，式 (3) を満足させるために D7 で $P[s]$ を $P[t]$ に更新する．

4. 評　　価

提案手法の有効性を示すため，パトリシア [3] とマルチウェイ基数探索法を実現したダブル配列 [1], [4], [6] を比較手法とし，探索，追加，削除処理に対して比較評価を行う．ダブル配列について，キー全体をトライ上に実現した配列 TAIL を用いない手法 [1] を比較手法 D，配列 TAIL を用いた手法を比較手法 DT とし，パトリシアを比較手法 P とする．

ここで，キー数を n，ダブル配列の全要素数を m，未使用要素数を e とし，キー長を k，キーの遷移種数を b とする．

4.1 理論的評価

探索処理について，各手法は基数探索法であるため，任意のキーに対するトライ上の遷移数は高々k である．比較手法 D は，トライ上の遷移数が k であるので $O(k)$ となる．比較手法 DT は，トライ上の遷移数と葉でのキー比較により $O(k)$ となる．また，パトリシアの高さが最良 $\log_2 n$ であるのに対し，提案手法が実現するトライの高さは最良 $\log_b n$ となる．よって，葉でのキー比較による計算量を考慮すると，比較手法 P が $O(\log_2 n + k)$ であり，提案手法が $O(\log_b n + k)$ となる．

追加処理について，比較手法 D は新規節点の追加位置までトライ上を遷移し，追加位置で関数 InsLeaf を呼び出す [1]．比較手法 DT は，追加位置までトライ上を遷移し，追加位置で関数 InsLeaf か関数 InsCommon を呼び出す．よって，以下では関数 InsLeaf と関数 InsCommon の評価を行う．

ダブル配列の関数 InsLeaf は関数 NewBase に依存する [4], [6]．関数 NewBase は，要素数 e の E-LINK を走査し，要素数 b の遷移集合による終点が作成できるかを判断する．よって，関数 NewBase は $O(e \cdot b)$ である．関数 TransNode では，要素数 b の遷移集合による終点を移動し，さらに関数 RenewalCheck で CHECK 値を更新する．そのため，関数 TransNode は $O(b \cdot b)$ である．よって，関数 InsLeaf は $O(e \cdot b + b \cdot b)$ である．

比較手法 DT で呼び出す関数 InsCommon は，関数 NewBase と一方向分岐を作成する計算量が多い．一方向分岐を作成する処理は，最大で k 個の節点を作成する．またこの処理では，要素数が 1 の遷移集合を入力とする関数 NewBase を呼び出す [4]．このとき，E-LINK により $O(1)$ で BASE 値が決定するので，一方向分岐を作成する処理は $O(k)$ である．一方，関数 InsCommon 内で呼び出す関数 NewBase は，要素数が 2 の遷移集合を入力としているので $O(e \cdot 2)$ である．よって，比較手法 DT における関数 InsCommon は $O(k + e \cdot 2)$ となる．

比較手法 P の追加処理は，パトリシア上を根から葉まで遷移し，追加キーと葉に対応したキーを比較して追加位置を決定する．その後，再度パトリシア上を根から追加位置まで遷移するので，$O(\log_2 n + k + \log_2 n)$ となる [3]．

提案手法の追加処理について，パトリシアと同様に関数 Insert でトライ上の根から葉までたどり，葉でキー比較を行う．その後，再度トライ上を根から遷移し，関数 InsLeaf か関数 InsCommon を呼び出す．よって，まずトライ上の遷移とキー比較に $O(\log_b n + k + \log_b n)$ が必要である．関数 InsLeaf はダブル配列と同様に $O(e \cdot b + b \cdot b)$ であり，関数 InsCommon は，

比較手法 DT とは異なり一方向分岐を作成しないので $O(e \cdot 2)$ である．

削除処理について，比較手法 D，DT は，削除キーに対応する葉から根へ遷移しながら，関数 IsOneWayBranch により一方向分岐であるかを確認する．よって，$O(b \cdot k)$ である．

比較手法 P の削除処理は，まず削除キーを探索し，削除キーに対応する葉と，場合によっては他の一つの節点を削除する．二分木であるために削除する他の節点は $O(1)$ で決定できるので，$O(\log_2 n + k)$ である [3]．

提案手法の削除処理は，パトリシアと同様，削除キーの探索後に葉と他の一つの節点を削除する．葉以外の節点を削除するかは関数 IsOneWayBranch により判断し，削除する場合は関数 RenewalCheck を呼び出す．よって，提案手法は $O(\log_b n + k + b + b)$ となる．

4.2 実験的評価

比較手法 P は約 400 行，比較手法 D，DT と提案手法は約 700 行の C 言語で実装し，Intel Pentium 4 2.8GHz，Fedora core 4 上で実験を行った．実験では，多分木としてトライを実現した場合でも一方向分岐が多いキー集合として，1,000 万件をランダムに抽出した URI を母集合 S とした．S におけるキーの平均長は 58.47 バイトであり，文字コードは ASCII としたため，内部表現値は 0 から 255 である．比較手法 P における遷移を 1 ビット，比較手法 D，DT および提案手法における遷移を 1 バイト（8 ビット）とする．

比較手法 P について，Morrison の文献 [3] に従って作成し，キーは提案手法と同様に配列 TAIL に格納した．比較手法 P の節点は，キーの比較位置，終点への二つのリンク，キーを指す配列 TAIL の要素番号をもつ [3]．

探索処理に対する実験について，比較手法 P と提案手法はトライを構築するキー数に依存する．よって，キー数を変動させた S の部分集合 S_1 を用いて探索実験を行った．S_1 は，S より 50 万件から 500 万件まで 50 万件ずつランダムに抽出したものを用いた．ここで，探索を行ったキー集合は 50 万件の S_1 とし，すべて成功探索とした．

探索実験の結果を表 1 と図 11 に示す．表 1 には，500 万件の S_1 に対するトライ上の探索時間と配列 TAIL 上の探索時間，およびそれらの合計時間，トライ上の遷移数を示す．図 11 には，S_1 により構築されたトライに対する探索時間を示す．表 1 と図 11 に示した値は，1 つのキーに対する平均値である．

図 11 より，トライを構築するキー数が増加するにつれて，各手法の探索時間も増加していることが確認できた．また表 1 より，提案手法は比較手法 D より約 1.97 倍，比較手法 DT より約 1.88 倍，比較手法 P より約 2.13 倍高速である．これは，表 1 からもわかるように，他の手法と比べて，提案手法はトライ上の遷移数が少なく，トライ上の探索が高速となったためである．

以上より，提案手法はトライ上の遷移数を抑制し，探索処理を高速化できたことがわかる．

更新処理に対する実験について，初期状態として 50 万件の S_1 を追加し，その後ランダムにキーを追加および削除する実験を行った．更新実験では，更新するキー数を変動させ，追加

表 1　探索処理に対する実験結果

Table 1　The simulation results of search processing.

	探索時間（μs）			トライ上の遷移数
	トライ上	配列 TAIL 上	合計	
比較手法 D	4.02	0.00	4.02	58.48
比較手法 DT	3.68	0.18	3.86	48.67
比較手法 P	3.89	0.47	4.36	52.21
提案手法	1.60	0.44	2.04	15.97

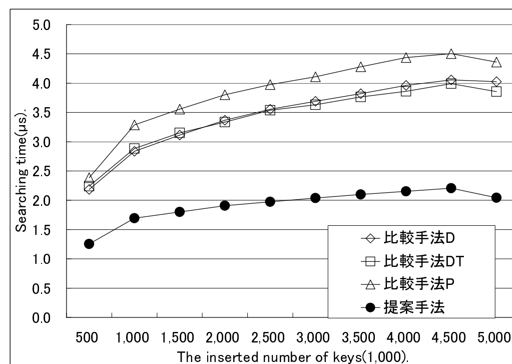


図 11　追加キー数変動による探索時間の実験結果

Fig. 11　The simulation results of searching time.

時間，削除時間，使用空間について評価を行う．ここで，追加および削除するキー集合を S_2 とし，S からランダムに抽出したものを用いた．S_2 の追加キー数と削除キー数はほぼ同じとし，ランダムに追加処理と削除処理を行うので，トライに登録されるキー数は更新処理中もほぼ 50 万件である．

S_2 のうち 100 万件，500 万件，1,000 万件に対する更新実験の結果を表 2，表 3 に示す．表 2 には，更新処理における追加時間，削除時間を示し，あわせて，関数 NewBase における計算回数，トライ上の節点数，ダブル配列上の未使用要素数を示す．表 2 に示した計測時間は 1 つのキーに対する平均値であり，関数 NewBase における計算回数は一回の衝突で必要となった平均値である．

表 2 より，1,000 万件の S_2 において提案手法の追加速度は比較手法 P の約 0.79 倍と遅く，比較手法 D より約 1.41 倍，比較手法 DT より約 1.64 倍と高速となった．比較手法 P が高速となったのは，ダブル配列とは異なるデータ構造であり，関数 InsLeaf のような計算量の多い処理を必要としないためである [3]．

比較手法 DT と提案手法を比較した場合，更新キー数の増加とともに提案手法が高速となっている．これは，未使用要素が増加したことにより，関数 NewBase の計算回数が減少したためである．関数 NewBase は，E-LINK を走査しながら遷移集合の全要素による終点が未使用要素であるかを判断する．つまり，未使用要素が多いと E-LINK の走査における計算量が多くなるが，終点が未使用要素と対応しやすい．

一方，比較手法 D は未使用要素がほとんど存在せず E-LINK の走査における計算量が少ない．よって，関数 NewBase の計算回数は少ないが，追加処理全体の時間が提案手法よりも多い．

表 2　更新処理に対する実験結果
Table 2　The simulation results of updating processing.

キー数	100 万	500 万	1,000 万
追加時間（μs）			
比較手法 D	10.41	12.94	14.22
比較手法 DT	8.54	17.55	16.51
比較手法 P	7.79	7.93	7.94
提案手法	13.56	12.12	10.07
削除時間（μs）			
比較手法 D	21.12	25.96	29.34
比較手法 DT	11.72	13.17	13.58
比較手法 P	6.44	6.60	6.62
提案手法	5.31	5.43	5.43
関数 NewBase の計算回数			
比較手法 D	1.01	5.80	35.83
比較手法 DT	1.05	32.09	29.44
提案手法	1.01	27.37	20.79
トライの節点数			
比較手法 D	7,597,277	9,504,186	9,511,421
比較手法 DT	2,166,326	2,253,162	2,287,317
比較手法 P	432,624	500,300	500,538
提案手法	676,635	781,673	781,808
未使用要素数			
比較手法 D	108	576	84
比較手法 DT	82,164	83,638	118,719
提案手法	183,950	194,253	217,733

これは，追加する節点数が多いためである．

ここで，提案手法が比較手法 D，DT よりも未使用要素が多いのは，関数 NewBase において終点と未使用要素の対応が容易ではなく，未使用要素がそのまま存在するためである．これは，提案手法における節点からの遷移が必ず二つ以上存在するためである．一方，比較手法 D，DT は一方向分岐による終点が容易に未使用要素と対応するため，未使用要素が減少する．

削除処理について，1,000 万件の S_2 において提案手法の削除速度は，比較手法 D より約 5.40 倍，比較手法より約 2.50 倍，比較手法 P より約 1.46 倍高速であった．これは，提案手法が最大二つの節点を削除するのに対し，比較手法 D，DT は最大キー長個の節点を削除するためである．また，提案手法が比較手法 P よりも高速となったのは，表 1 より削除キーの探索処理に差があるためである．

表 3 に，トライと配列 TAIL における使用空間を示す．使用空間は更新処理後に算出し，比較手法 P は節点の各要素を，比較手法 D，DT は配列 BASE，CHECK を，提案手法は配列 BASE，CHECK，POS を対象とした．

使用空間について，提案手法におけるトライと配列 TAIL での合計使用空間は，比較手法 D の約 2 分の 1，比較手法 DT の約 1.57 倍，比較手法 P の約 1.10 倍であり，提案手法の使用空間は大きい．しかし，提案手法におけるトライの使用空間は，比較手法 D の約 10 分の 1，比較手法 DT の約 2 分の 1，比較手法 P の約 1.56 倍であった．比較手法 D，DT に対しては，配列 POS を付加した提案手法においても空間が小さくなった．

表 3　使用空間に対する実験結果
Table 3　The simulation results of used area.

キー数	100 万	500 万	1,000 万
トライに関する使用空間（MB）			
比較手法 D	60.78	76.03	76.09
比較手法 DT	15.27	17.36	17.35
比較手法 P	5.19	6.00	6.01
提案手法	8.12	9.38	9.38
配列 TAIL の使用空間（MB）			
比較手法 DT	6.54	7.35	7.35
比較手法 P，提案手法	25.31	29.27	29.28
合計使用空間（MB）			
比較手法 D	60.78	76.03	76.09
比較手法 DT	21.81	24.70	24.69
比較手法 P	30.50	35.27	35.29
提案手法	33.43	38.65	38.66

よって，比較手法 DT よりも合計使用空間が大きくなったのは，キー全体を配列 TAIL に格納したことにより配列 TAIL の使用空間が増加したためである．提案手法の配列 TAIL に対する使用空間は比較手法 DT の約 3.98 倍であった．

以上より，ダブル配列に対して提案手法の使用空間は大きいが，更新処理が有効であるといえる．また，パトリシアと比較した場合，提案手法により探索速度が向上したにも関わらず使用空間がほぼ同等であり有効であるといえる．

5. おわりに

本論文では，一方向分岐が多いキー集合に対して探索処理を高速化するため，ダブル配列を用いてパトリシアを拡張した基数探索法を提案し，実験により有効性を示した．今後の課題として，半無限文字列のようなキーが長いデータに対して実験を行い，詳細に評価することが挙げられる．

文　献

[1] J. Aoe, K. Morimoto, M. Shishibori, and K-H. Park. A trie compaction algorithm for a large set of keys. *IEEE Trans. Knowledge and Data Engineering*, Vol. 8, No. 3, pp. 476–491, 1996.

[2] W. Frakes. *Information Retrieval: Data Structures & Algorithms*. Prentice Hall, 1992.

[3] D. R. Morrison. Patricia practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, Vol. 15, pp. 514–534, 1968.

[4] 中村康正, 望月久稔. 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. 情報処理学会：データベース, Vol. 47, SIG 13（TOD 31）, pp. 16–27, 2006.

[5] 山本一徳, 獅々堀正幹, 柘植覚, 北研二. パトリシアトライの１次元配列構造への圧縮手法. 言語処理学会第 11 回年次大会, pp. 688–690, 2005.

[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.