Vol. 34 No. 2　　情　報　処　理　　Feb. 1993

連載講座

キー検索技法—IV

# トライとその応用†

青江　順　一††

## 1. まえがき

今回は，最終講座としてディジタル検索法 (digital search) であるトライ (trie) 法[1)~4)]を紹介し，その特徴と応用分野について説明する．トライは，Fredkin[8)] により名付けられ，“retrieve” の真中のスペル “trie” を語源としているが，木 (tree) の読み “トリー” と区別するために “トライ (try)” と読む．

トライは，キー集合 $K$ の各キーの共通接頭辞を併合して作られる木構造であり，ハッシュ法や2文探索木などがキー全体の値を比較対象とするのに対して，トライでは，キーの文字（あるいは値）の桁単位を比較の対象とする点に大きな相違点がある．この特徴により，トライはプログラミング言語処理系[2)]や自然言語処理システム[26),27)]における文字列を対象とした語彙の検索に向いている．

以下，**2.** ではトライの概要と特徴を紹介し，**3.** ではトライを実現するデータ構造とトライの圧縮法を説明する．**4.** ではスペルチェック，形態素，かな漢字変換，自由語（カタカナ），不要語と重要語，複合語，共起辞書などへのトライの応用について述べ，実際に適用した実験結果を紹介する．**5.** では，トライの他の分野への応用とパブリック・ドメイン・ソフトについて述べる．

## 2. トライとは

### 2.1 トライの概要

以後，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端記号 (endmarker) ‘#’ をキー最後に付けて説明する[1),2)]．また，節 $n$ から節 $m$ へ枝ラベル ‘$a$’ が定義されていることを関数 $g$ を使って $g(n, a)=m$ で表す．

次のキー集合 $K1=$ {baby#, bachelor#, badge#, badger#, jar#} に対するトライを**図-1** に示す．

図-1 で文字列 “baby#” の検索を考えてみる．まず，根1から $g(1, b)=2$ なるラベル ‘$b$’ の枝が存在するから第1文字 ‘$b$’ がマッチしたことになる．この1文字単位の比較を節3，4，5へと繰り返して葉6まで到達するので，“baby#” の検索は成功する．また，“bag#” を検索すると節3以降は辿れないので，検索は失敗する．

トライの特徴の一つは，入力文字列の左端より始まる全ての接頭辞（最左部分列と呼ぶ）が1回の入力走査で探索できることである．たとえば，図-1 のトライで入力文字列 “badgers#” を探索する場合，その接頭辞 “badger” と “badge” も同時に発見できる．ただし，この探索を可能にするには，節16と17で $g(16, \#)=19$ と $g(17, \#)=18$ なる枝を確認する必要がある．

二つ目の特徴は，探索失敗の場合でも入力文字列と部分マッチする接頭辞が検出できることである．たとえば，図-1 のトライで “bagy#” を検索するとき，部分マッチする “ba” まで検索が進められ，ミスマッチの位置が3文字目であることが分かる．

三つ目の特徴は，節から出る枝の探索がその数

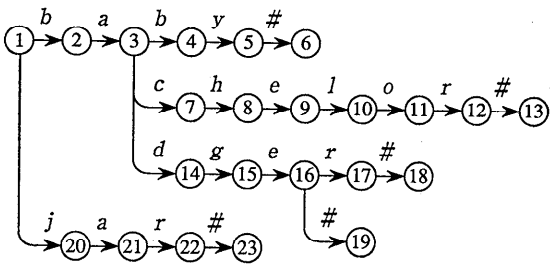



**図-1**　キー集合 $K1$ に対するトライ

† Key Search Strategies—Trie and Its Applications—by Junichi AOE (The University of Tokushima, Faculty of Engineering, Dept. of Information Science and Intelligent Systems).

†† 徳島大学工学部知能情報工学科

に関係なく一定時間で検索できるデータ構造が用いられるならば，キー検索に対する最大時間計算量（worst-case time complexity）はキーの総数に関係なく，キーの長さに比例することである．すなわち，トライでは非常に高速な検索が可能となる．

静的 (static) なキー集合 $K$ に対するトライの構成法は，ストリングパターンマッチングマシンの構成手法[5)~7)]を参照するとよい．なお，トライを動的に更新する手法は **4.** で紹介する．

## 3. トライのデータ構造と圧縮法

### 3.1 配列構造

図-1 のトライに対する配列構造を**図-2** に示す．

図-2 では，根1から出るラベル '$b$' の枝は，配列の行1と '$b$' に対応する要素2（次に進むべき節）で確認でき，もしこの要素が未定義ならば探索は失敗する．

配列構造では，任意の枝の探索に対する最大時間計算量は節から出る枝の数に関係なく 0(1) となり，キーの長さに比例した高速な検索，追加，削除が実現できる．しかし，節数 $n$，文字種類数を $m$ とするとき，大きさが $n\times m$ に比例するので，キーの数が増加すると記憶量が多くなる欠点がある．これは，配列要素がほとんど未使用（疎，スパース (sparse) と呼ばれる）であることに起因する．配列を圧縮する場合は，配列構造の高速性を保存した静的スパース行列の圧縮法[11)]や動的スパース行列の圧縮法[9),10)]などを利用するとよい．そのほかの圧縮法は文献 16)，18)，20) を参照されたい．

### 3.2 2進木構造

2進木構造は，トライの枝の定義のみをポインタで表現した構造であり，図-1 のトライは**図-3** の2進木構造として表される．

2進木構造の節は，枝ラベル，枝を辿るための左リンク，左リンクで対応する枝が見つからない場合に別の枝を辿るための右リンクで構成されている．たとえば，"badger#" は，節 1，2，3，4 までの左リンクで辿って "ba" の検索は成功するが，次の文字 '$d$' は，節4と7の右リンクを辿って節14（ラベルが '$d$' である）まで探索を進める必要がある．

2進木構造は配列構造よりコンパクトになるが，キーが増加して節から出る枝の数が多くなると，右リンクの探索回数が多くなり，検索効率が低下する欠点がある．しかし，キーの追加と削除は配列構造と同様に簡単であり，広く利用されているデータ構造である[2),24),25)]．

### 3.3 ダブル配列構造

ダブル配列法[12)~15)]では，二つの1次元配列 BASE，CHECK を使用して枝 $g(n,a)=m$ を次の

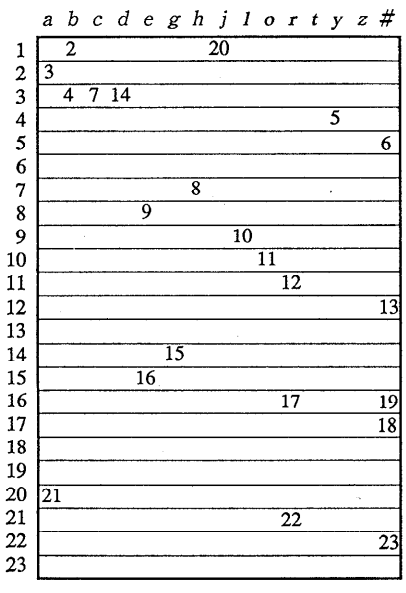



**図-2** キー集合 $K1$ に対するトライの配列構造表現

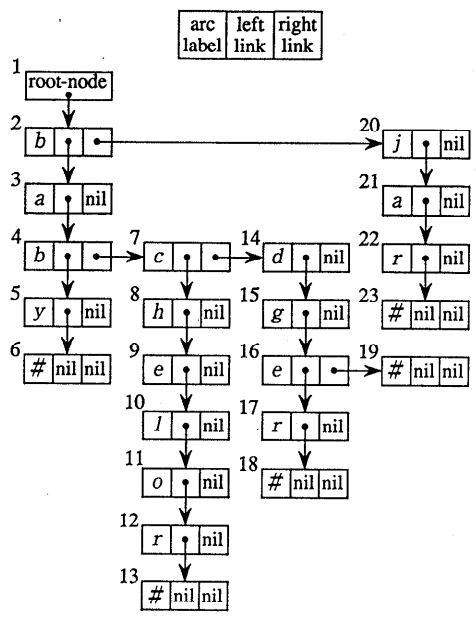



**図-3** キー集合 $K1$ に対するトライの2進木構造表現

ように表現する．

$m$=BASE[$n$]+$a$, CHECK[$m$]=$n$

すなわち，節 $n$ から出るラベル '$a$' の枝は BASE[$n$] に記号 '$a$' の内部コード値を加えた値 $m$ に対応する CHECK[$m$] に写像され，CHECK[$m$] にはこの枝が節 $n$ から出てきたことを表すために，$n$ が格納される．したがって，$g(n,a)=m$ の検索は次のように行われる．

$m$←BASE[$n$]+$a$;

**if**(CHECK[$m$]=$n$) **then** $g(n,a)$ は定義されており次に進むべき節番号は $m$ である

**else** 枝 $g(n,a)$ は定義されていない;

**図-4** の(a)は，図-1 のトライに対応するダブル配列を示すが，ダブル配列では節番号が変更されるので，**図-4** の(b)にダブル配列のインデックスに対応する節番号をもつトライを示す．また，記号 '#', '$a$', '$b$', …, '$z$' の内部コードは 1, 2, 3, …, 27 にそれぞれ対応する．ダブル配列の要素 0 は未使用要素である．端記号 # を経て遷移した節（図-4 の節 11）は次の遷移を定義する必要がないので，その節に対応する BASE には各キーに対応する情報を格納できる．ここでは，便宜上 "baby#", "bachelor#", "badge#", "badger#", "jar#" に対応する値 −1, −2, −3, −4, −5 をそれぞれ格納する．負の値にするのは，この BASE の値により遷移の終りを確認するためである．

文字列 "badge#" の最初の文字 '$b$' の検索を行ってみる．節 1 に対する BASE[1]=1 に '$b$' の内部コード 3 を加えると $m$=BASE[1]+$b$=4 となり，さらに CHECK[$m$]=CHECK[4]=1 となるので，枝 $g(1,b)=4$ の定義が確認できる．

ダブル配列では，節から出る枝数に関係なく枝を検索する最大時間計算量は常に 0(1) となる．文献 12), 13), 14), 15) には，ダブル配列に対する更新手法とその実用性を理論的かつ具体的に評価してある．また，記憶量も，種々のキー集合に対する実験により 2 進木構造と同等かそれ以下になることが示されている．しかし，ダブル配列の欠点は，枝の頻繁な削除があると未使用要素が容易に解放できない点にある．したがって，ダブル配列は追加頻度が削除頻度より高く，増進的に辞書項目が増える分野に適している．

### 3.4 最小接頭辞トライ

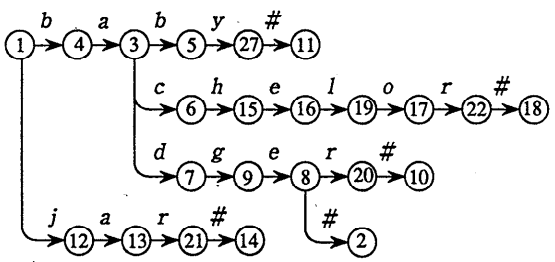
大きいキー集合では，トライの節が非常に多くなる．たとえば，英単語約 3 万語に対して節数は 10 万個近くになり，節数を減少させる必要がある[12),14),15)]．

図-1 のトライで分岐のないパス（たとえば，節 7 から始まり節 13 で終わる文字列 "helor#" に対応するパス）に気付いた読者が多いと思われるが，本節ではこのパス上のラベルを単に文字列として記憶する圧縮法を考える．このパスが始まる節をセパレート節（SP 節）と呼び，SP 節 $n$ 以降の枝ラベルより構成されるストリングを SP ストリング（str[$n$] と書く）と定義し，この SP ストリングを除いたトライを最小接頭辞（minimal-prefix, MP）トライと呼ぶ[19)]．図-1 のトライに対する MP トライを**図-5** に示す．ただし，str[19]=null は空記号列 (empty string) を表す．

図-5 上の "baby#" の探索では，SP 節 4（文字列 "bab"）まで進み，次に SP ストリング str[4]

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| BASE | 1 | -3 | 2 | 1 | 1 | 6 | 1 | 1 | 2 | -4 | -1 | 11 | 2 | -5 | 10 | 6 | 3 | -2 | 1 | 9 | 13 | 17 | 0 | 0 | 0 | 0 | 10 |
| CHECK | 0 | 8 | 4 | 1 | 3 | 3 | 3 | 9 | 7 | 20 | 27 | 1 | 12 | 21 | 6 | 15 | 19 | 22 | 16 | 8 | 13 | 17 | 0 | 0 | 0 | 0 | 5 |

(a) ダブル配列




(b) ダブル配列に対応するトライ

**図-4** キー集合 $K1$ に対するトライのダブル配列構造表現

=y# と残りの入力文字列 "y#" が一致するので，検索は成功する．

MP トライへのキーの追加を考えてみる．キー "baby#" だけが存在するときに（**図-6** の(a)），キー "bachelor#" の追加を行う．まず最初の文字 '*b*' を検索した後，残り文字列 "achelor#" と SP ストリング str[2]=aby# の比較で2番目の文字 '*b*' と '*c*' が異なる．ここで，残りの入力文字列と SP ストリングの共通接頭辞（文字 '*a*' に対する枝 $g(2,a)=3$ を作り，次に二つのキーを区別するための文字 '*b*' と '*c*' に対して，枝 $g(3,b)=4$ と $g(3,c)=5$ を作り，SP ストリング str[4]=y# と str[5]=helor# を定義する（**図-6** の(b)）．

次のキー "badge#" の追加では検索が節3で失敗するので，残りの文字列 "dge#" の最初の文字 '*d*' に対する枝 $g(3,d)=6$ を作り，SP ストリング str[6]=*ge*# を定義する（**図-6** の(c)）．

以上より，キーの追加は次のようにまとめられる．ただし，$\alpha$, $\beta$, $\gamma$ を文字列とし，*a*, *b* を文字とする．

(1) SP ストリング上でのミスマッチに対して節 $n$ に対する SP ストリング str[$n$]=$\alpha a\beta$# と残りの入力文字列を "$\alpha b\gamma$#" の比較で，'*a*' と '*b*' の文字でマッチングが失敗したと仮定するとき，まず節 $n$ より文字列 $y$ に対応する枝列を新しく作る．この枝列の最後の節を $m$ とするとき，新しい枝

$g(m,a)=k$, $g(m,b)=h$

を作り，str[$k$]=$\beta$#, str[$h$]=$\gamma$# なる SP ストリングを格納する．

(2) MP トライ上でのミスマッチに対して節 $n$ でミスマッチが起こり，残り入力文字列が "$a\alpha$#" であるとき，新しい節 $m$ と枝 $g(n,a)=m$ を作り，SP ストリング str[$m$]=$\alpha$# を格納する．

次に削除を考えてみる．図-6 の(c)に対してキー "baby#" を削除する場合，"baby#" を他のキーと区別する枝（キーに対応する SP 節に至る枝）$g(3,b)=4$ を削除すればよい．

MP トライをダブル配列で構成するアルゴリズムは，文献 12), 14), 15) を参照されたい．また，SP 節以前に存在する枝を圧縮する MP トライの構成法に関しては，文献 19) を参照されたい．

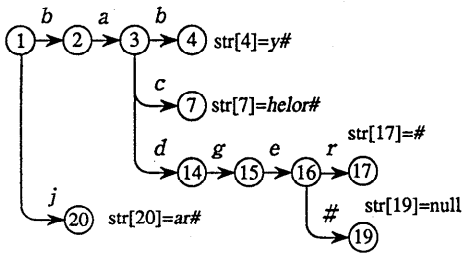



図-5 キー集合 $K1$ に対する MP トライ

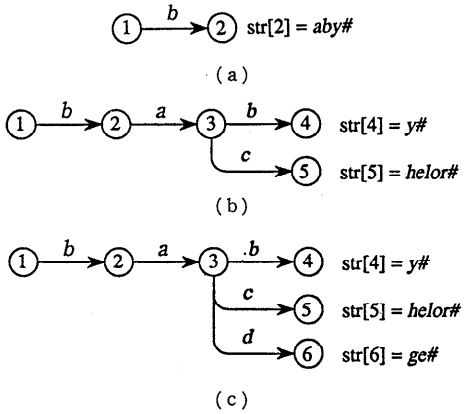



図-6 MP トライへのキーの追加

## 4. 自然言語辞書検索への応用

本章では，トライを自然言語辞書に応用する例としてスペルチェック，形態素，かな漢字変換辞書を説明する．また，トライの接尾辞を圧縮して，自由語（カタカナ），重要語，不要語辞書を検索する手法と複合語，共起辞書に利用するための手法についても触れる．

### 4.1 スペルチェック

トライは，スペルチェック辞書の高速検索技法としてもちろん有効であるが，ミススペル単語の訂正候補の検出に対しても有効である．

ミススペルである単語は，Peterson[27] によれば，

(a) 正しい2文字の置換
(b) 正しい1文字の欠落
(c) 余分な1文字の挿入
(d) 1文字の打ち間違い

に起因するものが約 80% を占める．トライではミススペル単語に対して，どの位置まで正しく検索されたのかが分かるので，上記のミススペルに関係する訂正候補語が容易に検出できる．

たとえば，図-1 のトライで "bay#" を検索すると，文字 '*y*' に対する枝が存在しないので，ここでミススペルの原因を「正しい1文字の欠落」と

仮定すると，節3から出る枝ラベル‘$b$’，‘$c$’，‘$d$’が欠落文字の候補となるが，次の文字‘$y$’を検索すると候補文字は‘$b$’となり，一つの訂正候補単語として“baby#”が検出できる．

しかし，キーの数が多くなると，訂正候補は非常に多く検出されるので，上記の単純な方法に加えて接辞情報を利用した訂正候補の絞り込みが必要となる．

### 4.2 形態素解析，かな漢字変換

形態素解析[28]は，入力文を形態素（言語学的に意味のある最小言語単位）に分割することである．しかし，日本語はべた書き文であるので，形態素解析辞書の検索では，入力文字列の任意の位置から最左部分列を切り出す作業が頻繁に行われる．

たとえば，「未完成交響曲」の解析では，まず最左部分列「未完」，「未」が検索され，長いほうの「未完」を採用して解析を進め，次に残りの文字列「成交響曲」から最左部分列を切り出し，「成」を得る．しかし，この形態素を動詞の語幹とすれば，付属語がマッチしないし，また単漢字扱いとしても接尾辞にはならないので，「未完」でスタートした解析をあきらめる．次に，「未」を採用し，文字列「完成交響曲」の最左部分列を検索し，解析を進める．

極端な場合として，一般的ハッシュ法で「未完成交響曲」の最左部分列を検索しようとすると，「未完成交響曲」，「未完成交響」，「未完成交」，「未完成」，「未完」，「未」なる6個のキーに対して検索を行う必要がある．しかし，トライではこの最左部分列が1回の検索で実現するので，形態素解析辞書の検索法として有効である．可変長ハッシュ法[31]は，長さを固定しないキーの検索を可能にするが，自然言語辞書のような大きなキー集合に利用できる一般性はない．

かな漢字変換の処理でも，仮名読み「くるまではこをはこぶ」に対して，

車で（来るまで）／箱を／運ぶ

車では（来るまでは）／子を／運ぶ

の変換候補を得るために，各文節区切りで最左部分列の検索が必要であるので，かな漢字変換辞書に対してもトライは有効な検索技法となる．

トライ以外の方法で，この最左部分列を検索する方法として，順序ハッシュ法[33]，拡張B木法[32]がある．しかし，この方法はキーの順検索を利用して同じ接頭辞のキー（たとえば，「未」，「未完」など）を同一ブロック内でまとめて格納するものであって，本質的なディジタル検索ではない．

### 4.3 自由語（カタカナ），重要語，不要語辞書

トライの共通の接尾辞を併合したDAWG (Directed Acyclic Word-Graph) を構成し，トライの節と枝数を軽減する手法が考えられている[17),18)]．

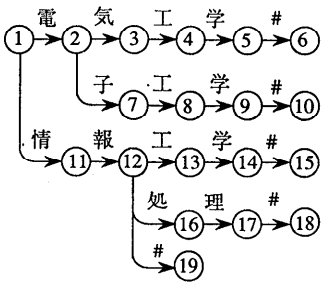
**図-7** にキー集合 $K2$={電気工学 #，電子工学 #，情報 #，情報工学 #，情報処理 #} に対するトライと DAWG の例を示す．ただし，DAWG では，キーに付加した端記号‘#’を除くので，検索成功は◎の節で表す．

図-7より，DAWG ではトライに比べて節と枝数が共に減少することが分かる．

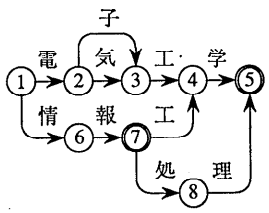
DAWG では，接尾辞の併合によりキー対応するレコードを一意に決定できないが，入力文字列がキー集合に存在するのか否かだけを検索し，キーに対応するレコード情報は辞書に格納しない分野，たとえば，スペルチェックにおける一般辞書や高頻度単語のフィルタリング辞書，自由度（カタカナ）辞書[23]，重要語や不要語の抽出用の辞書などがこの DAWG の応用範疇に入る．

### 4.4 複合語，共起辞書

DAWG はキーに対するレコードを一意に決定できないので，MP トライの SP ストリングに対




（a） キー集合 $K2$ に対するトライ




（b） キー集合 $K2$ に対する DAWG

**図-7** キー集合 $K2$ に対するトライと DAWG

表-1 トライに関する実験データ

| | KEY 1 | KEY 2 | KEY 3 | KEY 4 | KEY 5 | KEY 6 | KEY 7 |
|---|---|---|---|---|---|---|---|
| キーに関するデータ | | | | | | | |
| キーの数 | 35 | 310 | 657 | 1,480 | 23,976 | 32,344 | 65,857 |
| キーの平均長 | 5.1 | 7.5 | 6.9 | 9.5 | 8.2 | 5.6 | 6.8 |
| トライのノード数 | 161 | 1,558 | 2,781 | 9,742 | 100,235 | 88,801 | 259,324 |
| MP トライのノード数 | 52 | 611 | 1,051 | 2,461 | 40,498 | 45,383 | 98,713 |
| 記憶量の比較（キロバイト） | | | | | | | |
| ダブル配列 | 0.36 | 3.64 | 6.18 | 17.26 | 221 | 226 | 557 |
| 2進木構造 | 0.37 | 4.01 | 6.96 | 19.59 | 262 | 271 | 655 |
| ソースファイル | 0.28 | 2.33 | 4.54 | 14.08 | 197 | 182 | 671 |

してもう一つのトライ（SP トライと呼ぶ）を構成して DAWG の欠点を解決する手法を紹介する[23]．この手法では，MP トライと独立に，SP ストリングの逆順の文字列に対して SP トライを構成し，MP トライ（ダブルトライと呼ぶ）の節を連結する．MP トライはキーを一意に区別する SP 節をもつので，キーのレコードはこの SP 節に対応させて格納できる．

図-8 にキー集合 $K2$ に対するダブルトライを示す．破線の枝が両トライの節間の連結である．

“情報工学 ♯” の検索を考えてみる．まず，MP トライの SP 節 7 までの探索で “情報工” がマッチングでき，次に節 7 は SP トライの節 11 に連結されているので，残りの文字列 “学 ♯” は SP トライの枝を逆に探索することで成功する．

ダブルトライは，複合語や共起関係などの 2 単語間の関係表現にも拡張して応用できる[21),29]．たとえば，キー “情報処理” のレコードには，「接頭辞キー “情報” に関する複合語で，“情報を処理する” に分解できる」などの情報が格納できる．

SP トライでは検索時に枝を逆に辿り，SP ストリングを追加する場合は順方向に辿る必要があるので，データ構造はダブル配列が最適である．

### 4.5 実験結果

次のキー集合に対するトライの実験データを表-1 に示す．

KEY 1, KEY 2: プログラミング言語 PASCAL と COBOL の指定語の集合

KEY 3: UNIX の主要コマンド名の集合

KEY 4: 世界主要都市名の集合

KEY 5: かな漢字変換辞書のキー集合

KEY 6: 英単語の集合

KEY 7: 形態素辞書のキー集合

トライの枝ラベルは，KEY 5 がカタカナ，KEY 7 が 2 バイト日本語コードの 1 バイト分，残りのキー集合は英文字である．記憶量のデータは，MP トライに対するダブル配列と 2 進木構造表現の記憶量にそれぞれ SP ストリング（1 文字当たり 1 バイトとする）の記憶量を加えた値である．ただし，ダブル配列のインデックスと 2 進木構造のポインタがそれぞれ 2 バイトで表現できるように，キー集合 KEY 5, KEY 6, KEY 7 は部分集合に分割した．また，ソースファイルはキーを昇順に並べたもので，最も単純な線形探索のデータ構造に対応する[22),31]．表-1 から，MP トライの節数はトライの節数より大幅に減少することが分かる．また，ダブル配列は 2 進木構造よりコンパクトになることが分かる．これは，ダブル配列が 2 進木構造のように枝ラベルに対するフィールド情報を必要としないことと，ダブル配列の未使用領域が少ないことに関係する．辞書が大きくなると，2 次記憶から分割された辞書ブロックを転送することになるが，これを効率的に行うには，$B^+$ 木のように 2 次記憶上の動的ファイル管理技法[30]を導入する必要がある．

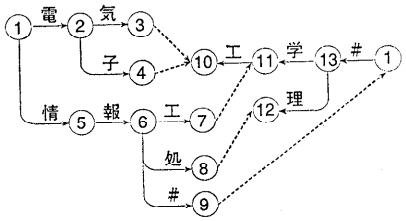



図-8 キー集合 $K2$ に対するダブルトライ

## 5. まとめ

以上，本解説ではトライの特徴を説明して，自然言語処理辞書検索への応用について述べた．本解説では触れなかったが，トライ検索は動的ハッシュ法でのトライハッシュ[24),30]の検索技法としてデータベースシステムや，テキスト検索[22]にも

利用されている.

なお，カルフォルニア大学 (Irvine 分校) のコンピュータ科学科の Douglas C. Schmidt 博士 (E-Mail: schmidt@ics.uci.edu) による MP トライの検索ソフトウェア trie-gen は，GNU の trie-gen-1.1.tar.Z として入手でき，またダブル配列による MP トライの検索ソフトウェア Double は，著者より入手可能である．trie-gen は，配列構造を利用したものであり，必要ならば Tarjan[11] のスパース行列の圧縮が可能であるが，多くのキー集合を動的に管理するには不便である．これに対して，Double は多くのキー集合を動的に管理でき，しかも文字種類の多い日本語にも対応している．

**謝辞** 有益なコメントをいただいた閲読者に感謝いたします.

## 参 考 文 献

[入門用文献]

1) Aho, A. V., Hopcroft, J. E. and Ullman, J. D.: Data Structures and Algorithms, Addison-Wesley, Reading Mass., pp. 163-169 (1983). 大野訳：培風館，pp. 143-148 (1987).
2) Knuth, D. E.: The Art of Computer Programming, Vol. 3, Sorting and Searching, Ch. 6 (1973).
3) Thomas, S. A.: Data Structure Techniques, Addison-Wesley, Reading Mass. Ch. 3 (1980).

[トライ関係の文献と分類]

4) Aoe, J.: Computer Algorithms—Key Search Strategies, IEEE Computer Society Press (1991).

[トライの一般的構成法]

5) Aho, A. V. and Corasick, M. J.: Efficient String Matching: An Aid to Bibliographic Search, Commun. ACM, Vol. 18, No. 6, pp. 333-340 (1975).
6) Aoe, J., Yamamoto, Y, and Shimada, R.: A Method for Improving String Pattern Matching Machines, IEEE Trans. Softw. Eng., Vol. SE-10, No. 1, pp. 116-120 (1984).
7) Aoe, J.: An Efficient Implementation of Static String Pattern Matching Machines, IEEE Trans. Softw. Eng., Vol. SE-15, No. 8, pp. 1010-1016 (1989).
8) Fredkin, E.: Trie Memory, Commun. ACM, Vol. 3, No. 9, pp. 490-500 (1960).

[配列構造と圧縮法]

9) 青江，安留：行置換による動的スパース行列の縮小アルゴリズム，信学論 (D)，Vol. J 71-D, No. 12, pp. 2508-2516 (1988).
10) Aoe, J.: A Practical Method for Compressing Sparse Matrices with Variant Entries, Int. J. Comput. Math., Vol. 36, No. 3, pp. 163-173 (1990).
11) Tarjan, R. E. and Yao, A. C.: Storing a Sparse Table, Commun. ACM, Vol. 22, No. 11, pp. 606-611 (1979).

[ダブル配列構造]

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) 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，信学論 (D)，Vol. J 71-D, No. 9, pp. 1592-1600 (1988).
14) 青江順一：自然言語辞書の検索—ダブル配列による高速検索アルゴリズム，bit (共立出版)，Vol. 21, No. 6, pp. 776-784 (1990).
15) Aoe, J., Morimoto, K. and Sato. T.: An Efficient Implementation of Trie Structures, Softw. Prac. & Exper., Vol. 22, No. 9, pp. 695-721 (1992).

[トライの変形 (圧縮)]

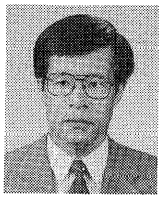
16) Al-Suwaiyel, M. and Horowitz, E.: Algorithms for Trie Compaction, ACM Trans., Database Syst., Vol. 9, No. 2, pp. 243-263 (1984).
17) 青江，森本，長谷：トライ構造における共通接尾辞の圧縮法，信学論(D)，Vol. J 75-D, No. 4, pp. 770-779 (1992).
18) Appel, A. W. and Jacobson, G. J.: The World's Fastest Scrabble Program, Commun, ACM, Vol. 31, No. 5, pp. 572-578 (1988).
19) Dundas, J. A.: Implementing Dynamic Minimal-Prefix Tries, Softw. Prac. & Exper., Vol. 21, No. 10, pp. 1027-1040 (1991).
20) Kurt, M.: Compressed Tries, Commun, ACM, Vol. 19, No. 7, pp. 409-415 (1976).
21) 森本，青江：トライ構造による形態素辞書と共起辞書の統合法，情報処理学会研究会資料，NL 85-3 (1991).

[トライの応用]

22) Blumer, A., Haussler, J. D. and McConnell, R.: Complete Inverded Filters for Efficient Text Retrieval and Analysis, J, ACM., Vol. 34, No. 3, pp. 578-595 (1987).
23) 川口，加藤，藤澤，畠，藤縄：自由語検索のための高速文字列検索方式，第 39 回情報処理学会全国大会，2 N-8 (1989).
24) Litwin, W. A., Roussopoulos, N., Levy, G. and Hong, W.: Trie Hashing with Controlled Load, IEEE Trans. Softw, Eng, SE-17, No. 7, pp. 687-691 (1991).
25) 中嶋，杉村：TRIE 構造とグラフスタックをもちいた日本語形態素解析，第 39 回情報処理学会全国大会，IF-4 (1989).
26) 長尾，辻井，山上，建部：国語辞書の記憶と日本語文の自動分割，情報処理，Vol. 19, No. 6 (1978).
27) Peterson, J. L.: Computer Programs for Spelling Correction, Lecture Notes in Comput. Sci., Springer-Verlag, N. Y., pp. 57-67 (1980).
28) 田中穂積：自然言語解析の基礎，産業図書，3 章，5 章 (1989).
29) 上脇，田中：辞書の TRIE 構造化と熟語処理，Proc. of LPC '85, ICOT, pp. 329-340 (1985).

[その他関連文献]
30) Enbody, R. J. and Du. H. C.: Dynamic Hashing Schemes, ACM Comput. Surveys, Vol. 20, No. 2, pp. 85-113 (1988). 遠山訳: bit 別冊 (共立出版), pp. 43-67 (1990).
31) Pearson, P.K.: Fast Hashing of Variable-Length Text Strings, Commun. ACM, Vol. 33, No. 6, pp. 677-680 (1990).
32) 日高, 稲永, 吉田: 拡張 B-Tree と日本語単語辞書への応用, 信学論(D), Vol. J 67-D, No. 4, pp. 399-404 (1984).
33) 横山, 元吉, 井佐: 二次記憶上の大規模語彙を用いる自然言語処理システム, 情報処理学会論文誌, Vol. 29, No. 6, pp. 570-580 (1988).

(平成4年8月26日受付)



**青江 順一** (正会員)

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. 同年同大学工学部助手(情報工学科), 現在知能情報工学科助教授. 工学博士. この間, コンパイラ自動生成系, 自然言語処理と理解, 知識工学に関する研究に従事し, パーサ, 情報検索, ストリングパターンマッチング, データ圧縮に関するアルゴリズムの効率化に興味をもつ. 著書「Computer Algorithms-Key Search Strategies」IEEE CS Press など. 電子情報通信学会, 日本人工知能学会, 日本ソフトウェア科学会, 日本機械翻訳協会, IEEE, ACM, AAA, ACL 各会員.