データベースシステム　100－14
（１９９４．１０．１４）

拡張ハッシュ法による検索技法の拡張
－部分文字列検索と順検索への拡張－

望月　久稔†　　清原　聡†　　森本　勝士†　　青江　順一†

†徳島大学　工学部

拡張ハッシュ法は動的なキー集合に対して，高速な検索を実現する手法である．拡張ハッシュ法でキーの順検索を可能にするコンパクト２進木法がJongeらにより提案されているが，キーの検索時間が大きい．本稿では，キーの順検索特性を保存して従来の手法を改良することにより，キーの検索時間の改善方法を提案する．本手法を用いることで、検索速度が58倍高速になることが実験結果より確かめられた。

また，これまでの提案方法では、与えられた任意の文字列を含むキーの検索（部分文字列検索）を効率的に行うことはできなかった．本稿では、部分文字列検索を効率的に行うため，特徴ベクトルと呼ばれるビット列を用いる方法を提案する．また，通常の拡張ハッシュ法では、複数のバケットを参照する必要が生じる。本稿では更に、その参照数の抑制のためにディスクリプタを利用してキーの処理時間の改善方法も提案する．本手法の有効性を確認するため実験を行った結果、通常の拡張ハッシュ法に比べ，40倍以上高速に検索できることがわかった。また、検索キーの長さが長くなるほど、高速に検索できることが確認できた。

Improvements of Extensible Hashing Schemes
- Substring and Order Searching -

Hisatoshi MOCHIDUKI †, Satoshi KIYOHARA †
Katsushi MORIMOTO †, Jun-ichi AOE †

† The University of Tokushima
Minami Josanjima-Cho 2-1. Tokushima-Shi 770, JAPAN

Extensible hashing schemes keep fast key retrieval for dynamic key sets. Although Jonge et al. proposed the compact binary tree (CB-Tree) method that enables order preserved searching by extensible hashing schemes, searching and updating a key takes a lot of time for a large set of keys. This article proposes a method for dividing the CB-Tree, in order to improve the time cost. The simulation results shows that the method presented is 58 times faster than the traditional CB-Tree.

Extensible hashing schemes can not detect substrings of a given string without accessing all buckets. In order to realize effective substring searching, this article proposes a new idea that uses signature vectors as a hash value and that uses a descriptor for each bucket like a multidimensional extensible hashing method. The results of the simulations show that the method presented is 40 times faster than the traditional extensible hashing scheme.

1．はじめに

拡張ハッシュ法は静的ハッシュ法の欠点を解消し，キー集合の性質が予測できない分野においても高速な検索を維持することができる．

Jonge[5]らはハッシュ表をコンパクト２進木（compact binary tree, CB-Tree）で表現し，拡張ハッシュ法によりキーの順検索を実現する方法を提案しているが，木構造が大きくなるとキーの検索，更新時間が長くなる．本稿では，このCB-Treeの順検索特性を保存して，時間的低下を改善する方法を提案する．

また，拡張ハッシュ法で不可能であった部分文字列検索を実現するために，特徴ベクトル(signature vector)[4]をハッシュ関数に導入する．部分文字列検索では，一件の検索要求に対して複数のバケットを参照する必要が生じるので，多次元拡張ハッシュ法で用いられているディスクリプタ(descriptor)[7]と呼ばれるビット列を使用し，検索の高速化を図る．

以上，拡張ハッシュ法に対する二つの改善方法を提案し，実験結果による評価を与える．

2．拡張ハッシュ法[2],[3]

拡張ハッシュ法はハッシュ表を動的に伸縮することでバケット(bucket)のアクセスを１回でできる高速な検索法である[2]．拡張ハッシュ法では，キーKを十分に大きいm長のビット列に写像する関数

$$H(k)=b_{m-1}\ \ldots\ b_2b_1b_0$$

を定義し，H(K)に対して末尾のi個のビット列を抽出したビット列を関数

$$h_i(K)=b_{i-1}\ \ldots\ b_2b_1b_0$$

（b_i（$0\leqq i\leqq m-1$）は，0または1のビットを表す）として定義する．

例えば，都府県名１５個をローマ字表記したキーを構成する文字の内部コード値（a, b, c, …, zの内部コード値はそれぞれ1, 2, …, 26とする）の総和w(K)を利用する[2]．簡単のために，関数H(K)をw(K)の後尾６ビット列とし（表１），H(K)の末尾の１ビットに対応する関数$h_1(K)$を選び，キーakita, okayama, hiroshimaを格納したハッシュ表を図１(a)に示す．但し，バケット中に格納可能なキーの数BUCKET_SIZEは2とする．

$h_1(k)$が１ビットなので，ハッシュ表の大きさは2であり，$h_1(k)$=0に対応するバケット1にはキーakitaとhiroshimaが，$h_1(k)$=1に対応するバケット2にはokayamaが格納される．ここで，ハッシュ表の全域深さは，ハッシュ表全体の番地計算に必要なビット数を表し，各バケットの局所深さはそのバケットを他のバケットと区別するために必要な番地計算のビット数を表す．

図１(a)に対して，キーtokyo(010110), h_1(tokyo)=0を追加するとき，バケット1であふれが生じるので，２ビット分の関数$h_2(k)$により，大きさを２倍にする（図１(b)）．図１(b)では，番地00と10は唯一のバケットに対応するが，バケット2には複数の番地01と11が対応する（局所深さ1が全域深さ2より小さい）．次のキーtokushima(110101)とokinawa(001010)の追加では，キーtokushimaはバケット2に追加できるが，キーokinawaはバケット3に追加できない．しかも，バケッ

表１　ハッシュ関数の例

k	w(k)	H(k)	k	w(k)	H(k)
akita	42	101010	kumamoto	109	101101
okayama	67	000011	kochi	46	101110
hiroshima	100	100100	fukui	68	000100
tokyo	86	010110	nara	34	100010
tokushima	117	110101	nagasaki	63	111111
okinawa	74	001010	mie	27	011011
kagawa	44	101100	niigata	61	111101
osaka	47	101111			

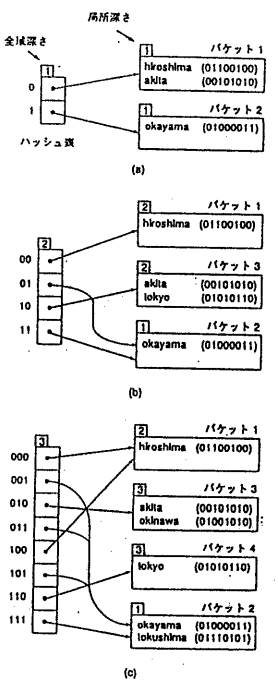

図１　拡張ハッシュ表の例

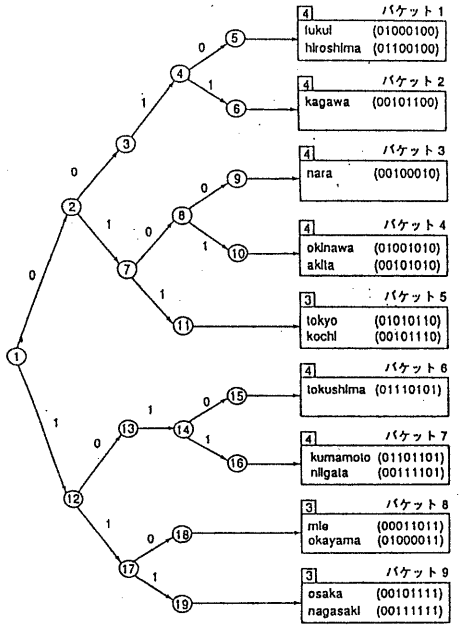

図2　2進木（トライ）表現の例

ト3の局所深さと全域深さは2であるので，3ビット分のハッシュ関数$h_2(k)$によりハッシュ表を成長させて，キーokinawaを追加する（図1(c)）．

ところが，ハッシュ表の拡張とともに各局所的深さにばらつきが生じ，拡張したハッシュ表の多くの番地が局所深さの浅いバケットに集中するので，ハッシュ表も冗長になる．従って，ハッシュ表は，2進木構造で表現する場合が多い[8]．

図1(c)に対して，残りの全てのキーを追加したトライによる拡張ハッシュ表を図2に示す．

キーokayama(000011)の検索は，ノード1,12,17,18と辿り，バケット8をアクセスすればよい．

3．　順検索可能なキー検索処理の高速化

3．1　CB－Treeの概要

CB-Treeではキーの値順を保存するためにハッシュ値としてキー自身の2進数表現を用いる．例えば，3文字の英単語をキーの内部コード値を各々5ビットの2進数表現にした関数H(key)を利用する．キーcatでは，H(cat)=000110000110100となる．また，バケットサイズを2とする．

図3はキー集合

K＝{cat, ear, job, pen, sea, sun, zoo}

に対する2進ディジタル探索木（binary digital search tree, BDS-tree）を示す．BDS-Treeとは，キーのビット列のトライ（trie）構造である．但し，トライ構造ではキーの増加とともにノード数が増加し，記憶量が多くなる．Jongeらはノードがアーク2本をもつことを保証するために，1本のアークしか持たないノードに対しては，ダミーバケットを挿入したコンパクトな表現CB-Treeを提案した．

図3のCB-Treeには8つのバケットがあり，バケット4，5，7はダミーバケットである．キーkがビット列00($h_2(k)$=00)で始まるレコードはバケット1，01で始まるレコードはバケット2に格納される．このように，バケットに到達するまでの経路をキーの上位ビット列で構成すれば，キーの内部コードが保存されるので，CB-Treeはキーの値順を反映でき，これを先行順走査すれば，キーの順検索が可能となる．

図3でキーpenを検索する場合，H(pen)=100000010101110より，ルートノード1からノード2，バケット1，バケット2を通過し，ノード3，4，5を経てバケット3にアクセスし，キーpenが検索される．また，順検索では，同様に先行順走査で木を辿り，バケットを通過する度にバケット中のレコードを出力すればよい．

CB-Treeはトライの状態をビット列で表現したtreemap，バケットの状態（ダミー，非ダミー）を表現したbitmap，バケット番号を格納したバケット表から構成される．

treemapは先行順走査により，○印の内部ノード通過時点でビット0を，□印のバケット通過時点でビット1を出力したビット列である．bitmapも同様にバケット

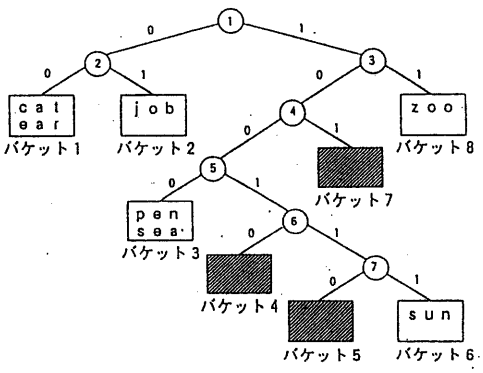

図3　CB-Tree

通過時点でバケットがダミーバケットならビット0を，非ダミーバケットならビット1を出力したビット列である．

図3のCB-Treeに対するtreemap，bitmapおよびバケット表を次に示す．

treemap	0011.000101011111
bitmap	11100101

table

1	2	3	4	5
1	2	3	6	8

3．2　改良ＣＢ－Ｔｒｅｅの概要

CB-Treeでは，全てのビット列走査を前提としているので，大きな木構造では，検索，更新時間が低下する．本節では，この改善法[10]を提案する．

図4は図3のCB-Treeを最大全域深さが2になるように分割したものである．キーsun(H(sun)=10011101010 1110)の検索では，キーsunを全域深さ以下になるように10/01/11/･･･のように分割する．まず，1つ目の分割キー10を用いて，トライ1を走査し，ノード4にトライ2へのポインタがあるので，次の分割キー01を用いて，トライ2を先行順走査する．同様に，ノード6にトライ3へのポインタがあるので，次の分割キー11を用いて，トライ3を走査し，バケット6に到達するので，キーsunが登録されているのが確認される．

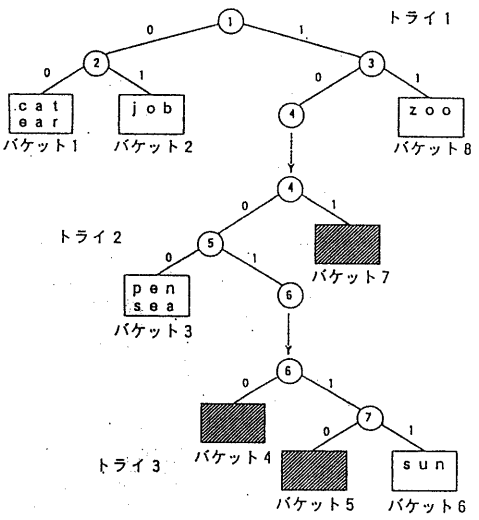

図4　改良CB-Tree

また，キーzoo(H(zoo)=111100111101111)の検索では，トライ1を走査するだけでバケット8に到達し，キーzooが検索できる．

この改良により，不必要なトライの走査が不要になるので，検索，更新時間を短縮することができる．

3．3　改良ＣＢ－Ｔｒｅｅの検索アルゴリズム

本手法では，従来のCB-Treeに対し，次の変更を行う．

(1)　次のトライへのポインタとなっている内部ノードを葉とみなし，その葉に対応するtreemapとbitmapの部分に1を立てる．

(2)　バケット表中には次のトライ番号にマイナスを付けた値を格納する．

図4のCB-Treeに対する構造は次のようになる．

	トライ　1	トライ　2	トライ　3
treemap	0011011	00111	01011
bitmap	1111	110	001

table

トライ1				トライ2		トライ3
1	2	-2	8	3	-3	6

改良CB-Treeでは各トライの最大全域深さLを設定し，キーの2進数表現をLビット毎に次のように分割する．

$H(k)=H_0(k)H_1(k)\cdots H_i(k)\cdots H_n(k)$

($H_0(k)\sim H_{n-1}(k)$：L bit，$H_n(k)$：≦ L bit)

なお，iは現在のトライの番号，keyは検索対象キー，tmapはtreemap，bmapはbitmap，tableはバケット表を表す．特に，tmap，bmap，tableにおいてi番目のトライを指すときは$tmap_i$，$bmap_i$，$table_i$と表す．また，tpos，bpos，kposは各々tmap，bmap，keyの現在処理中のビット位置を表し，bucketはバケット番号を，indexはtableの添字を表す．また，以下の関数を用いる．

g(map，pos)：mapよりposの位置のビット値を返す．

f(H(k)，pos)：H(k)よりposの位置のビット値を返す．

skipandcount()：トライを右に進むときに，左部分木をスキップする．

findleftbucket()：トライの走査でダミーバケットに到達したとき，ダミーバケットの最も左側にある非ダミーバケットのバケット番号を返す．

getindex()：bmapの先頭からbposが指すビットまで1が立っている数がtableの添字となる．

find(key，bkt)：キーkeyをバケットbkt中より探し，見つかればTRUEを，見つからなければFALSEを返す．

以上の関数の入力として，mapはtmapまたはbmap，b

ktはバケット番号，posはmapおよびkeyのビット列中の特定のビットを指定する整数を表す．

以下に本アルゴリズムの流れを示す．

```
      begin
         i:=1;  j:=0;
         for each j such that H_i(key) exsists do
           begin
             kpos:=1;  tpos:=1;  bpos:=1;
(s-1)       while g(tmap_i, tpos)=0 do
              begin
(s-2)           if f(H_i(key), kpos)=1 then
(s-3)             bpos=bpos+skipandcount();
                tpos:=tpos+1;  kpos:=kpos+1;
              end
             j:=j+1;
(s-4)        if g(bmap_i, bpos)=0 then
              begin
(s-5)           bucket:=findleftbucket();
                return(find(key, bucket));
              end
             else
              begin
(s-6)           index:=getindex();
                bucket:=table_i[index];
                if bucket<0 then
                  i:=-bucket;
                else
                  return(find(key, bucket));
              end
           end
       end;
```

まず，行(s-1)のループはi番目のトライのtreemapのtposが指すビットが0である度に，行(s-2)でkposの指すビットが0ならトライを左に進み，1なら右に進む．トライを左に進む場合は，tposを1ビット右に進め，右に進む場合は左部分木をスキップする．CB-Treeは葉の数が内部ノードの数より1多い性質より，行(s-3)の関数skipandcountで(ビット1を通過する数:A)=(ビット0を通過する数:B)+1となるまでtposを進め，bposをスキップした左部分木の葉の数Aだけ進める．行(s-1)でtposの指すビットが1になるとループを抜け，行(s-4)でi番目のトライのbitmapのbposが指すビットを調べる．このビットが0ならばダミーバケットに到達したことになるので，行(s-5)の関数findleftbucketでダミーバケットの最も左側にある非ダミーバケット番号を見つけ，keyの検索を行う．ビットが1ならば非ダミーバケットか次のトライを指しているノードに到達したことを意味するので，行(s-6)の関数getindexでi番目のトライのテーブルの添字indexを調べ，テーブルのindex番目の値bucketを見る．この値が負の値ならばそれは次のトライへのポインタなので，iに-bucketを格納して行(s-1)に戻り，上記の処理を繰り返す．正の値の場合は非ダミーバケットなので，そのバケット中でkeyの検索を行う．

表2　順検索の実験結果

	CB−Tree	改良CB−Tree
時間（ミリ秒）		
検索	29.24	0.5
追加	68	7
削除	32.46	2.3
記憶量（kバイト）		
treemap	8.5	9.3
bitmap	4.2	5.1
テーブル	8.9	22.6

3．4　評価

表2にトライの最大全域深さを5，バケットサイズを16とした場合のランダムな日本語名詞50,000語に対する実験結果を示す．本手法の有効性を確認するため，従来のCB-Tree法との比較を行った．検索，追加，削除時間は最初に全ての語を登録しておき，登録済みの全ての語の検索・削除と，未登録の1,000語の追加を行った場合の1語当たりに要する時間である．バケットの総数は6,849，トライの総数は6,849，内部ノード数は33,913，トライの最大深さは176である．

実験結果から，本手法は従来法より，検索で58倍，追加で10倍，削除で14倍高速になることが分かった．また，記憶量は改良前に比べ，treemapで1.1倍，bitmapで1.2倍，テーブルで2.5倍程増加しているが，CB-Treeが元来非常にコンパクトであるのでこれだけの増加でも十分実用的な値である．

4．部分文字列検索

4．1　特徴ベクトル

拡張ハッシュに対する部分文字列検索のための，新しいハッシュ関数を導入する．

文字列strに対するビット長mの特徴ベクトルは，関数s(str)より得られ，$s(str)=b_{m-1}\cdots b_1 b_0$は，str中の全ての隣接2文字abに対し，i=hash(ab)なるiを計算し，$b_i=1$とすることで得られる（hash(ab)は2文字abから[0,m-1]なる整数を返す）．

二つの文字列str_1, str_2に対する特徴ベクトルをそれぞれ$sign_1=b_{m-1}\cdots b_1 b_0$, $sign_2=c_{m-1}\cdots c_1 c_0$とするとき，以下が成立する．

a) $b_i=1$なる全てのiに対し，$c_i=1$であるとき，
　str_1はstr_2の部分文字列である可能性がある．

b) それ以外のとき，
　str_1はstr_2の部分文字列でない．

表３　特徴ベクトルの例

key	signature vector
cat	0000010010
check	0100011010
consume	1100011001
educate	1000110010

a)が満たされる場合はstr_1がstr_2の部分文字列であるか否かを実際に検証する必要がある．

関数COMP_VECT(sign1_, sign_2)は，特徴ベクトルsign_1, sign_2がa), b)のいずれを満たすかを判定し，a)が満たされる場合はTRUEを，b)が満たされる場合はFALSEを返す．

［例１］ビット長10の特徴ベクトルを生成する関数s:(a+b)%10により，キーcat, educate, consume, checkに対する特徴ベクトル(m=10)を表３に示す．ただし，演算子%は剰余を表す．

キーcat, consumeに対し，COMP_VECT(s(cat), s(consume))=FALSEとなるので，キーconsumeはcatを部分文字列として含まないことが分かる．一方，キーcat, educateに対し，COMP_VECT(s(cat), s(educate))=TRUEとなり，キーcat, checkに対しても，COMP_VECT(s(cat), s(check))=TRUEとなる．したがって，キーcatの特徴ベクトルに対し，関数COMP_VECTがTRUEとなる特徴ベクトルを持つキーはeducate, checkである．この二つのキーを検証すると，catを部分文字列に含むのはeducateのみであることがわかる．

４．２　ディスクリプタ

本手法による部分文字列検索では複数のバケットを参照する必要があるため，ディスクリプタを採用し，読み出しバケット数を抑制する．

キーkeyに対するディスクリプタk_discは関数sと同様の関数GENE_DISCより生成し，バケットの持つディスクリプタb_discは，次の重ね合わせ関数SET_DISCにより計算される．

$$SET\_DISC(BUCKET) = GENE\_DISC(key_1) \mid GENE\_DISC(key_2) \mid \cdots \mid GENE\_DISC(key_n)$$

ただし，BUCKETは該当バケットのバケット番号，$K = \{key_1, key_2, \cdots, key_n\}$ をバケットBUCKET中のキーの集合とし，|は隣接２項間のビット毎の論理和を表す．

［例２］ビット長８のハッシュ値を生成する関数s:(a+2b)%8を用いて（演算子%は剰余演算子），表１のキー集合を登録したものを図５に示す．各キーに対す

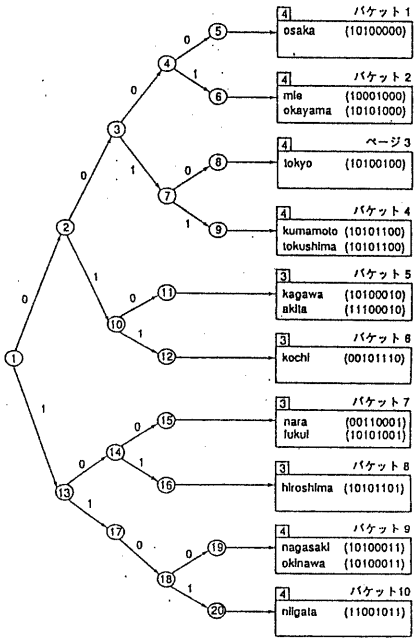

図５　特徴ベクトルによる拡張ハッシュ

表４　キーに対するディスクリプタ

k	k_disc	k	k_disc
akita	0000001001001100	kumamoto	1111000100000001
okayama	0001010100000001	kochi	0010000000000001
hiroshima	0010001100000011	fukui	0100100100000100
tokyo	1000000100000100	nara	1000100000000010
tokushima	1110010100000000	nagasaki	1100000001000100
okinawa	1100000100100100	mie	0000000000000001
kagawa	0100010000000100	niigata	0100111010010000
osaka	0000010001000101		

表５　バケットに対するディスクリプタ

bucketi	b_disc
1	0000010001000101
2	0001010100000001
3	1000000100000100
4	1111010100000001
5	0100011001001100
6	0010000000000001
7	1100100100000110
8	0010001100000011
9	1100000101100100
10	0100111010010000

るディスクリプタk_disc（表４）は関数GENE_DISC:(7a+13b)%16によって生成されている．また，図５のディスクリプタb_discを表５に示す．　例えば，バケット5に含まれるキーkagawa,akitaのディスクリプタはそれぞれ0100010000000100,0000001001001100となっており，バケット5のディスクリプタはこれらを重ね合わせた0100011001001100となっている．

４．３　部分文字列検索

本稿における部分文字列検索の関数substr_searchでは，以下の変数，関数を用いる．

iは特徴ベクトルの現在処理中のビット位置；NODEは現在処理中のノード番号（NODE=1は根を表す）；BUCKETは葉であるノードNODEに対するバケット番号；DETECTは目的のキーが見つかったか否かを示す；keyは検索部分文字列；sign(=$b_{m-1}\cdots b_1 b_0$)はkeyの特徴ベクトル；discはkeyのディスクリプタ；b_discはBUCKETのディスクリプタ；GENE_SIGN(key)は特徴ベクトルの項で述べたsig関数s；advance()はノードNODEを起点としてsignのビット列を用いてトライの走査を行い，葉に到達できたか否かを調べる関数；replace()：ビット0による遷移をビット1による遷移に置き換えられるか否かを調べる関数；bucket(node)はノードnodeに対するバケット番号を返す関数；g(node,c)はノードnodeからcによる遷移先のノード番号を返す関数；遷移先のノードが存在しない場合はg(node,c)=failと定義する．substr(str,bkt)は文字列strを含むキーをバケットbkt中より探し，見つかればTRUEを，見つからなければFALSEを返す関数．また，これらの各関数（手続き）の入力はbkt：バケット番号；str：文字列；node：ノード番号；c：ビット値（0または1）である．

【関数substr_search(key)】

```
     begin
       DETECT:=FALSE;  sign:=GENE_SIGN(key);
       disc:=GENE_DISC(key);  NODE:=1;  i:=0;
       repeat
(ss-1)    if advance()=TRUE then
             begin
(ss-2)         BUCKET:=bucket(NODE);
(ss-3)         if COMP_VECT(disc,b_disc)=TRUE then
                  begin
(ss-4)               if substr(key,BUCKET)=TRUE
                       then DETECT:=TRUE;
                  end
             end
(ss-5) until replace()=FALSE
       return(DETECT)
     end;
```

［例３］図５を用いてsubstr_search(shima)の実行を考える．

文字列shimaのsgin，discは，

sign:=GENE_SIGN(shima)=s_1(shima)=10001100,

disc:=GENE_DISC(shima)=0010000100000000

と計算され，行(ss-1)で関数advanceにより特徴ベクトルsignが走査される．advanceはノード1,2,3,7,9と辿りTRUEを返す．行(ss-2)でBUCKET:=bucket(9)が実行されBUCKET=4となる．バケット4のディスクリプタb_disc=1111010100000001であるので，行(ss-3)でCOMP_VECT(disc,b_disc)=TRUEとなる．行(ss-4)のsubstr(shima,4)の実行で，バケット4からshimaを部分文字列として含むキーtokushimaが検出されsubstrはTRUEを返し，該当キーが見つかったのでDETECTはTRUEに変更される．g(7,1)=9,g(3,1)=7,g(2,0)=3,g(2,1)=10≠failであるので，行(ss-5)の関数replaceはノード9から7,3,2と戻り，ノード2からビット1によってノード10へ進みTRUEを返し行(ss-1)に戻る（このときNODE=10,i=2となっている）．同様な処理を続けsubstr_searchのrepeat-until ループを抜けるまでにキーhiroshimaが検出される．repeat-until ループを終了するとTRUEが返される．

４．４　評価

実験に用いたキー集合は日本語（103,336語），英単語（80,425語），ディスクリプタのビット長は共に６４である．実験では日本語，英単語に対してそれぞれトライの最大深さは日本語名詞で２９，英単語対象で１９となった．トライの総ノード数は日本語名詞で22,463，英単語で16,211，使用バケット数は日本語名詞11,020，英単語8,076である．

表６は，本手法での部分文字列の検索結果および，以下で説明する通常の拡張ハッシュ法による部分文字列検索を想定した実験の結果を示す．

通常の拡張ハッシュ法による部分文字列検索を想定した実験は，本手法で生成された全バケットを逐次的に読み込み，バケット中のキーを対象として部分文字列検索を行った．参考のため，ディスクリプタ比較による読み込みバケット絞り込み行った場合も比較対象とし，表６ではこれらの実験をそれぞれ「ディスクリプタなし」，「ディスクリプタあり」としている．

実験は日本語，英語とも検索部分文字列の長さ別に１００件ずつ部分文字列検索を行っている．

表６では，トライ走査によって到達したバケットの数を示し，読み込みバケット数は読み込みを行ったバケットの数を示す．表中の値はいずれも部分文字列検

表6　部分文字列検索の実験結果

検索部分文字列の長さ	本手法			ディスクリプタあり		ディスクリプタなし	
	到達バケット数（読込バケット数）	走査ノード数	時間（秒）	読込バケット数	時間（秒）	読込バケット数	時間（秒）
日本語							
2文字	567.87(170.49)	1,430.72	0.31	1,121.71	2.00	11,020	13.01
3文字	202.47(49.51)	588.71	0.08	425.06	0.85	11,020	12.90
4文字	90.03(21.71)	288.53	0.03	224.20	0.39	11,020	12.84
5文字	53.91(13.88)	177.28	0.02	126.47	0.21	11,020	12.85
6文字	30.17(9.41)	111.66	0.01	79.72	0.13	11,020	12.84
英　語							
3文字	1,874.64(823.82)	4,179.81	1.54	2,211.24	3.34	8,076	10.53
4文字	1,017.53(343.31)	2,455.58	0.65	1,340.83	2.19	8,076	10.45
6文字	353.88(103.65)	974.56	0.19	589.68	1.07	8,076	10.32
8文字	135.77(32.09)	429.56	0.05	247.84	0.46	8,076	10.21
10文字	89.27(22.69)	303.22	0.04	175.28	0.32	8,076	10.14
12文字	46.51(11.62)	180.06	0.02	116.21	0.20	8,076	10.10

索1件当たりの平均値を示している．

実験結果より，本手法は「ディスクリプタなし」と比較して7～千倍程度と非常に高速となっており，「ディスクリプタあり」と比較しても2～10倍程高速であることが分かった．

本手法の走査ノード数はトライの全ノード数の1/4から1/200程度に抑えられ，走査しないアーク決定の効果が表れている．

5．おわりに

本稿では，コンパクト2進木が順検索は可能であるが大きな木構造ではキーの処理時間が大きくなることに着目し，トライ構造を分割することにより，処理時間を改善する手法を提案した．今後は，あふれ処理の効率化を考案し，より深い評価を行う予定である．

また，拡張ハッシュ法において部分文字列検索を効率的に行うために，特徴ベクトルを生成する関数をハッシュ関数として用いることを提案した．今後の課題としては，ディスクリプタ比較による読み込みバケット絞り込みをより効率的にするための，特徴ベクトル生成関数の精選が挙げられる．

参考文献

(1) 青江順一：”静的ハッシュ法とその応用”，情報処理，33,11,pp.1359-1366 (1992-11).

(2) 青江順一：”動的ハッシュ法とその応用”，情報処理，33,12,pp.1465-1472 (1992-12).

(3) Enbody R.J. and Du H.C. : "Dynamic Hasing Scheems", ACM Computing Surveys, 20,2, pp.85-113 (1988). 遠山元道 訳：Bit別冊，共立出版，p43-68(1990).

(4) Harrison M.C. : "Implementation of the Substring Test by Hashung", Commun.ACM, 14,12,pp.777-779 (Dec. 1971).

(5) Kelly K.L. and Rusinkiewicz M.R. : "Multikey,Extensible Hashing for Relational Databases", IEEE Software, pp.77-85 (Jul. 1988).

(6) Mehlhorn, K.:Dynamic Binary Search Tree, SIAM J. Comput., Vol.8, No.2, pp.175-198 (1979).

(7) Ramamohanarao K., Lloyd J.W and Thom J.A. : "Patial-Mathc Retrieval Using Hasing and Descriptors", ACM Trans.Database Syst., 8,4, pp.552-576 (Dec. 1983).

(8) R.J.Enbody, et al.:Dynamic Hashing Schemes, Comput. Surveys, Vol.20, No.2, pp.85-113, (1990).

(9) 佐藤，青江：拡張ハッシィングにおけるディリクトリの圧縮アルゴリズム：情報処理学会第45回全国大会，4-259，(1992).

(10) W.D.Jonge, et al.:Two Access methods using Compact Binary Trees, IEEE Trans. Softw. Eng., Vol.SE-13, No.7, pp.799-810, (1987).