解説 基般

圧縮したまま検索

— Succinct Data Structure —

岡野原 大輔　（株）プリファードインフラストラクチャー

近年，高い圧縮率と高速な問合せを両立するデータ構造として，
簡潔データ構造（succinct data structure）が注目されている．
簡潔データ構造は理論的な発展とともに，実用化に向けた改善が進んでおり，
今後さまざまな分野でその用途がさらに広がっていくと期待される．
本稿では，簡潔データ構造の仕組みと，その利用事例について解説する．

新たなデータ構造のパラダイム

あらゆる分野で大量のデータを処理する必要性が増している．これに応じて大量のデータを高速に処理しリソース量を削減するためのさまざまなアルゴリズム，データ構造，システムが提案されている．

この高速化とリソース量削減という 2 つの課題は実は深く関係している．なぜなら，記憶階層間の速度差は埋まらないどころか広がっており，高速化の最大の課題はすべての処理を高速な上位の記憶階層で処理できるかどうかであるからである．そのため，多少計算量が大きくなったとしても必要なリソース量を減らすことができれば，高速な主記憶上で計算ができ，全体として大幅な高速化を達成できる．

簡潔データ構造はデータ圧縮と索引の 2 つの技術を組み合わせることで，データを小さく保存したまま，復元せずに各種操作を高速に実現することが可能なデータ構造である．

データ圧縮は，データの特徴を分析し，適切な表現方法と符号方法を用いることでデータをコンパクトに格納する．簡潔データ構造でも，それぞれのデータの性質に合わせた適切な表現方法を利用することで，そのデータ構造の情報理論的限界に近いサイズで格納している．

大量のデータを扱う上でもう 1 つの鍵となるのが索引である．データをあらかじめ分析しておき，処理に必要な情報を索引として格納しておく．これにより，データ全体を毎回一から分析しなくても，索引を参照することで高速な操作が実現できる．簡潔データ構造でも，データの種類と操作に応じた索引を構築し，高速な操作を実現する．

これまでにさまざまな種類の簡潔データ構造が提案されている．**表 -1** に代表的な簡潔データ構造を示す．

簡潔データ構造を利用することで，これまで考えられなかったような大きなデータを主記憶上で操作

簡潔データ構造	説明
Rank/Select 辞書	Rank/Select 操作を備えたビット列．より複雑な簡潔データ構造の実現に必要．
簡潔木	木の簡潔な表現．節点 1 つあたり約 2 ビットで木に対する操作を実現．
ウェーブレット木	一般の文字列に対する簡潔データ構造．圧縮全文索引や 2 次元情報の処理に利用．
グラフ	頂点と枝情報を格納し，頂点間の関係などを高速に格納できる．
関数（連想配列）	任意の入力と出力の関係を保持．最小完全ハッシュ関数を利用する場合が多い．
圧縮全文索引	文書中の任意の文字列の出現回数，位置を返す．索引サイズは元の文書サイズより小さい．

表 -1　代表的な簡潔データ構造とその特徴

することができる．たとえば数十億頂点から構成される木構造を主記憶上に保持し，その木を圧縮格納したまま，木を辿ったり，木の性質を調べるといったことが可能となる．

本稿では，簡潔データ構造をいくつか紹介し，簡潔データ構造の概念を理解してもらうとともに，その利用例を紹介する．

簡潔データ構造の仕組み

簡潔データ構造の目標はコンパクトかつ高速処理が可能なデータ構造である．

多くのデータ構造はそのままの表現よりも非常に小さく格納することができる．たとえば順序木の簡潔データ構造は1節点あたり2ビットであり，従来のポインタベースの格納方法（1節点あたり32ビット〜数百ビット）と比べてはるかに小さい．

また，索引を構築することで，高速な処理を実現する．索引は操作に必要な情報（たとえば操作の途中までの計算結果）をあらかじめ求めておき，それを記録したものである．一般に索引の大きさと計算時間はトレードオフの関係にある．より詳細な情報を索引に記録すれば計算量は減らせるが，索引に必要なサイズは増えていく．簡潔データ構造は索引をうまく間引くことによって，計算時間を増やさずに索引をコンパクトに保持することを目指す．

この章では簡潔データ構造を簡単なものから複雑なものまで順に解説する．

■ Rank/Select 辞書：集合に対する簡潔データ構造

最初に，長さ n のビット列 $B[0, n-1]$，$B[i] \in \{0, 1\}$ に対する簡潔データ構造を説明する．このビット列 B に対し，次の2つの操作を考える．

- $\mathbf{rank}_b(B, x)$ $B[0, x-1]$ 中の $b \in \{0, 1\}$ の出現数を返す
- $\mathbf{select}_b(B, i)$ B の先頭から i 番目の $b \in \{0, 1\}$ の位置を返す

これらの操作を備えたビット列を Rank/Select 辞書と呼ぶ．このデータ構造は最も単純なデータ構造であるが，多くの簡潔データ構造で利用される重要なデータ構造である．

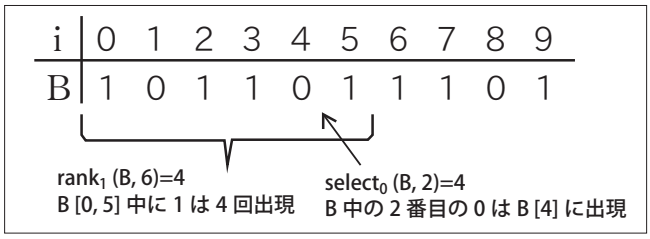


図 -1 Rank/Select 辞書の例

図 -1 に Rank/Select 辞書の例を挙げる，たとえば $\mathbf{rank}_1(B, 6)=4$ であり，$\mathbf{select}_0(B, 2)=4$ である．

これらの操作を $n+o(n)$ ビット[☆1]の記憶領域を用いて定数時間で行う方法を説明する．

配列 B を長さ $l := \log^2 n$ ビットの大ブロックに分割する．次に各大ブロックを長さ $s := 1/2 \log n$ ビットの小ブロックに分割する．

表 $L[0, n/l]$ に，各大ブロックの先頭の $\mathbf{rank}_1(B, il)$ の結果を格納し，$L[i]=\mathbf{rank}_1(B, il)$ とする．また表 $S[0, n/s]$ に，各小ブロックの先頭の $\mathbf{rank}_1$ の結果と，それが所属する大ブロックの先頭からの **rank** の差分 $S[i]=\mathbf{rank}_1(B, is)-L[is/l]$ を格納する．また，表 P に長さ s のブロック内で起き得るすべてのビットパターンに対するすべての位置の答えをあらかじめ求めておき格納しておく．

これらの表はいずれも，$o(n)$ ビットである．たとえば，表 L は長さは n/l で，それぞれの要素が $\log n$ ビットで格納できるので全体で $n \log n/(\log^2 n) = n/\log n$ ビットである．

これにより，$\mathbf{rank}_1(B, x)$ は $L[x/l]+S[x/s]$ と残り s 未満の長さのビット中の1の出現数の和で求まる．これはすべて表 L，S，P の表引きで求まるので定数時間で求めることが示せた．

また，**select** についても表 L と表 S を利用して二分探索を行えば，$O(\log n)$ 時間で求められる．また，**rank** よりも複雑だが，$o(n)$ ビットの索引を利用することで定数時間で **select** 操作は実現できることが知られている．

☆1 $o(n)$ は n が大きくなるにつれて n に対し暫近的に無視できる項を意味する．

ビット列で重要となるのは1の数が0の数に比べて非常に少ない疎なビット列の場合である．このようなビット配列は元のビット列よりはるかに小さいサイズで保持することができ，その場合でも **rank**，**select** 操作を定数に近い計算量で行えることが知られている[4]．

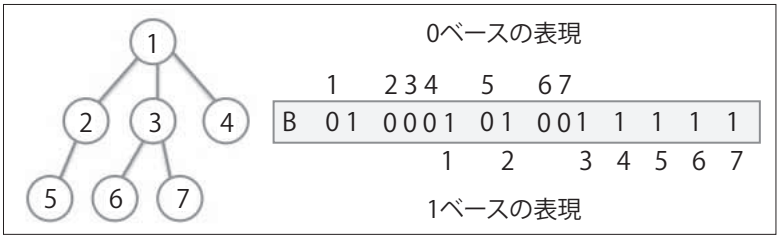


図 -2　順序木の簡潔表現である LOUDS の例

■ Rank/Select 辞書の利用例

Rank/Select 辞書を利用することでさまざまな情報を表現することができる．

- $U=\{0, 1, ..., n-1\}$ 中の部分集合 V を $x \in V$ ならば，$B[x]=1$，そうでないならば $B[x]=0$ としたビット列 B で表現することができる．たとえば，V の i 番目の要素は **select** (B, i) で得られる．
- $P[0, m-1]$，$P[i] \geq 0$ であるような正整数列 P をコンパクトに表すことができる．正整数 v の単進符号は v 個0を並べた後に1つだけ1を後続したビット列である．たとえば，正整数0，1，2の単進符号はそれぞれ1，01，001である．正整数列 $P[0, m-1]$ が与えられたとき，各要素 $P[i]$ を単進符号で符号化し，これらをつなげて得られたビット配列を $B[0, n-1]$ とする．このとき，$P[i]$ = **select**$_1(B, i)$ − **select**$_1(B, i-1)-1$ として求められる☆2．このビット列 B は m 個の1と $\sum_i P[i]$ 個の0からなり，要素数が多い場合でも合計値が押さえられているのであれば，正整数列をコンパクトに保存することができる．

また，任意の連続する部分配列の合計も $\sum_{i=s}^{e} P[i]$ = **select**$_1(B, e+1)$ − **select**$_1(B, s+1)-1$ として定数時間で求められる．その逆に任意の整数 $0 \leq x \leq \sum_i P[i]$ について，$\sum_{i=0}^{k} P[i] \leq x < \sum_{i=0}^{k+1} P[i]$ となるような k を $k=$ **rank**$_1(B,$ **select**$_0(B, x))$ として定数時間で求めることができる．

■ 順序木

順序木はキーの格納や，階層がある情報の格納など広く使われているデータ構造である．順序木は子の間に順序関係が付けられている木構造である．木中の節点には一意に特定できる番号が付けられているとする☆3．

順序木に対し，次の操作を考える．

- **parent** (v) 節点 v の親を返す．
- **first_child** (v) 節点 v の最初の子を返す．
- **sibling** (v) 節点 v の次の兄弟を返す．

それぞれ該当する節点が存在しない場合は -1 を返すものとする．

各節点ごとに操作の結果である節点のポインタを直接記録する場合，節点数が n のとき，節点番号を1つ格納するのに $\log n$ ビット必要☆4 であり，各節点について，**parent**，**first_child**，**sibling** のポインタを格納する必要があるので，全体で $3n \log n$ ビット必要となる．たとえば $n=2^{64}$ の場合，1節点あたり192ビット必要となる．

これに対し，n 個の節点からなる順序木の種類数はカタラン数として知られ高々 4^n 個しかないことが知られている．よって n 個からなる順序木は $\log_2 4^n=2n$ ビットで格納できるはずであり，これは先ほどのポインタベースの表現と比べてはるかに小さい．

ここでは，順序木の簡潔データ構造の1つである LOUDS (Level Ordered Unary Degree Sequence) を紹介する．**図 -2** に順序木の LOUDS 表現の例を示す．

まず各節点を幅優先探索で辿り，それぞれに1から順に番号を振っていく．また，長さ n の配列 C を用意し，$C[i]$ に i 番目の節点の子の数を記録する．

この配列 C は正整数列であるので，これを Rank/Select 辞書の利用例で紹介した方法で単進符

☆2　**select**$_1(B, 0)=-1$ と定義する．

☆3　本稿では葉も子の数が0個の節点と考える．

☆4　本稿では log は2を底とする．

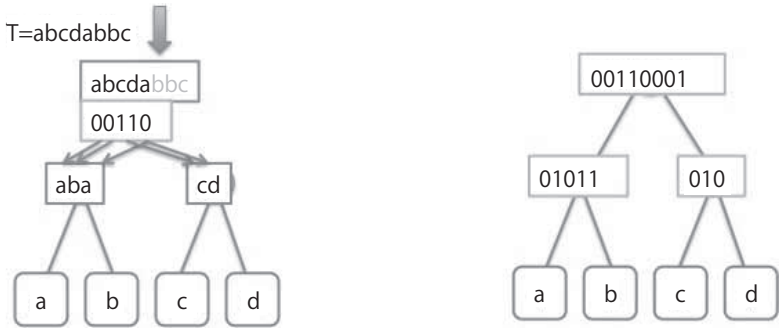


図 -3　*T*=abcdabbc に対するウェーブレット木の例

```
u ← root
while u は内部節点 do
  if c が u の左側の子孫 then
    u ← u.left, x ← rank_0 (B_u, x);
  else
    u ← u.right, x ← rank_1 (B_u, x);
  end if
end while
return x
```

図 -4　**rank**$_c$ (*T*, *x*) の例

号で格納し，その配列を B とする．ただし，先頭にスーパールートと呼ばれる番人に対応するビット列 01 を追加する．各節点の子の数の合計は，総節点数と一致するので，B は長さ $2n+1$ のビット配列であり，$n+1$ 個の 1 と n 個の 0 からなる．

LOUDS 表現では，各節点は子として指されているときの 0 と，自分の子の数を記録しているときの単進表現の末尾の 1 として 2 回表現されている．前者を 0 ベースの表現，後者を 1 ベースの表現と呼ぶことにする．たとえば図 -2 中の節点番号 3 の場合，0 ベースの表現では根の中で 2 番目の子として表現されており，1 ベースの表現では，2 つの子を持つ節点の末尾として表現されている．これらの表現は幅優先探索順なので，対応する節点番号の順序は同じであり，**rank**，**select** 操作を利用してお互いの値に変換することができる．

この表現を用いて木を実際に辿ってみる．はじめに 1 ベースの表現で p 番目の節点に対する **parent** (p) を考える．幅優先探索で節点を辿っているので p 番目の 1 に対応する 0 は，**select**$_0$ (B, p) として求めることができる．よって，**parent** (B, p)=**select**$_0$ (B, p) である．

0 ベースで p 番目の節点の場合，**first_child** (p) は，**first_child** (p)=**select**$_1$ (B, p) である．

また，0 ベースの表現での **sibling** は現在の位置の次の位置なので **sibling** $(p)=p+1$ として表される．これ以外にも多くの操作が定数時間で実現できることが知られている．

順序木を表すにはこのほかにも Baranced Parenthesis 表現，Depth First Unary Degree Sequene 表現，Range Min-Max Tree などが知られている．簡潔木の比較については文献 1) に詳しい．

■ ウェーブレット木

Rank/Select 辞書の対象をビット列から，3 種類以上の文字から構成される文字列に拡張する．

文字列 $T[0, n-1]$，$T[i] \in \Sigma$，$\sigma=|\Sigma|$ を考える．この配列に対し次の操作を考える．

- **rank**$_c$ (B, x) $B[0, x-1]$ 中の $c \in \Sigma$ の数を返す
- **select**$_c$ (B, i) B の先頭から i 番目の $c \in \Sigma$ の位置を返す

先ほどの Rank/Select 辞書と同じように T をブロックごとに分割して表を作ってみることを考えることもできるが，各表のエントリで **rank** の結果を σ 個格納した場合のサイズは元の文字列と比べて無視できないほど大きい．

ここではウェーブレット木を利用して **rank**，**select** 操作を実現する．ウェーブレット木は，完全二分木からなる．葉が各アルファベットに対応し，根がアルファベット全体に対応する (**図 -3**)．

ウェーブレット木は T 中の文字の順序を保ったまま，各文字を，対応する葉がある子に振り分けていく．各節点 v にはビット配列 B_v が付随し，現在の文字が左側の子に振り分けられたら 0，右側の子に振り分けられたら 1 を B_v の末尾に追加する（図 -3 左）．これを根に対して適応し，次に左の子，右の子について同じ操作を繰り返していく．結果として各節点にビット列が付随するデータ構造が得られる (図 -3 右)．これがウェーブレット木である．

次に **rank**$_c$ (T, x) を求める疑似コードを**図 -4** に

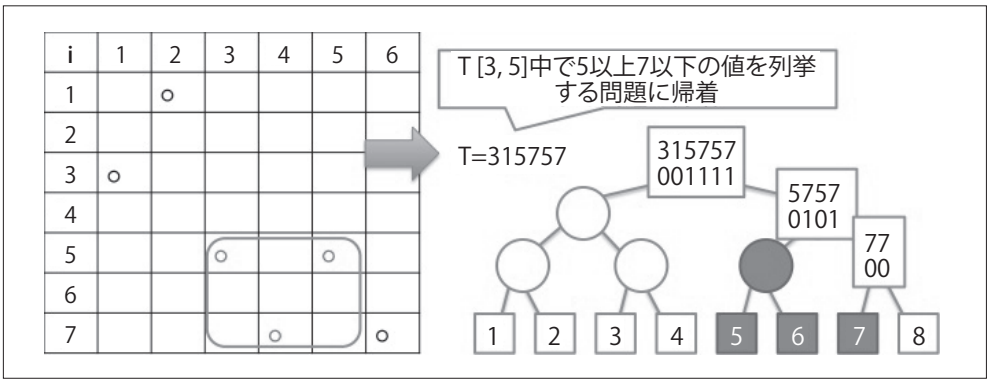


図 -5　2 次元中の点集合に対する $3 \leq x_i \leq 5$, $5 \leq y_i \leq 7$ を満たす点の列挙

示す．はじめに u を木の根とおく．文字 c に対応する葉が根の左側の子の子孫であるならば，構築の手順から根のビット配列 B_u 中では 0 に対応するので，x :=**rank**$_0(B_u, x)$ を求め，左側の子に移動する．このときの x は，根の左側の子の子孫にある葉に対応する文字集合に対応する．この操作を葉に到達するまで繰り返す．葉に到達した時点での x は $T[0, x-1]$ 中に存在する文字 c の個数に一致する．

逆に **select**$_c(T, x)$ の場合は葉から順番に根に向かって **select** 操作を繰り返していく．

ウェーブレット木の深さは $O(\log \sigma)$ であり，各節点での **rank**，**select** は定数時間で計算できるので，**rank**，**select** 操作は全体で $O(\log \sigma)$ 時間で計算できる．また，各文字ごとに $\log \sigma$ ビットで記録されるのでビット配列の長さの合計は $n \log \sigma$ ビットである．全体のサイズは $n \log \sigma + o(n \log \sigma)$ ビットである．

ウェーブレット木はほかにも非常に多くの操作を効率的にサポートでき，たとえば 2 次元データに対する矩形問合せを高速に行うことができる．

2 次元中の点集合 $V=\{v_i=(x_i, y_i)\}$，$0 \leq x_i < n$，$0 \leq y_i < m$ を考える．座標値は整数とする．また，すべての x_i は異なり，ちょうど 1 回ずつ出現するとする[☆5]．このとき，文字列 T を $T[x_i]=y_i$ のように定義すると，クエリである長方形 $(x_{sp} \leq x \leq x_{ep}, y_{sp} \leq y \leq y_{ep})$ の中に含まれる点をすべて列挙する問題は $T[x_{sp}, x_{ep}]$ 中に含まれる y_{sp} 以上，y_{ep} 以下の文字をすべて列挙する問題に還元できる（**図 -5**）．

そこで T に対し，ウェーブレット木を構築し，根から順にクエリ長方形に対応する範囲を調べていく．もし左側の子，右側の子だけであるなら，存在する節点のみを調べ，両方に含まれるのであれば，両方辿る．高々 $O(\log \sigma)$ 個の節点がクエリを含む節点になるので，それらに含まれる結果を報告すればよい．長方形に含まれる点数は常に $O(\log \sigma)$ 時間で求めることができ，それらの点の列挙も各点につき $O(\log \sigma)$ 時間で求められる．

■ 圧縮全文索引

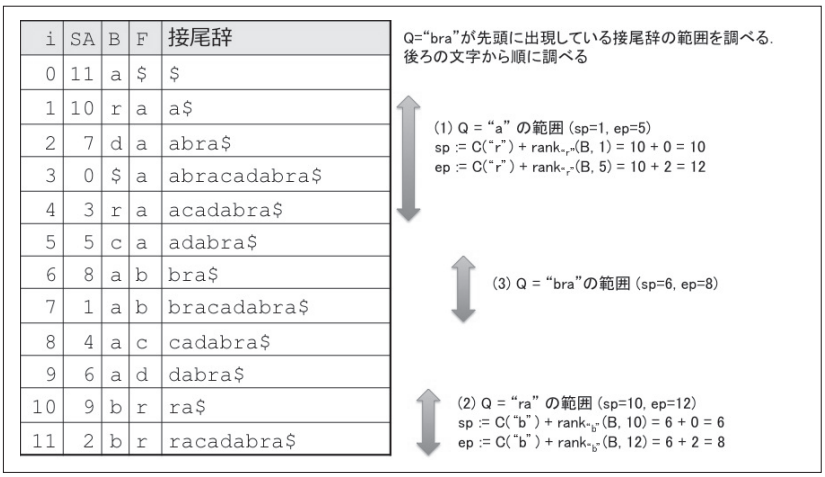
文書中の任意の文字列の出現回数，出現位置を求めることができる索引を全文索引と呼ぶ．全文索引は元の文字列の情報と同じ情報を持っているため[☆6]，基本的には元の文字列と同じかそれ以上のサイズを必要とする．

この全文索引を元の文字列と同じサイズまたはそれ以下に圧縮したのが圧縮全文索引である．しかも索引のみから元の文字列を復元することができる性質を持ち (self-index)，元の文字列を保持する必要がない．圧縮全文索引はこれまで圧縮接尾辞配列，LZ-index などが提案されている．ここでは FM-index と呼ばれる方法について紹介する．

はじめに FM-index を説明するにあたって接尾辞配列と Burrows Wheeler 変換を紹介する．検索対象文字列を $T[0, n-1]$ とし，各文字 $T[i]$ は全順序関係が与えられているとする．また，文字列末尾 $T[n-1]$=\$ はほかの位置に出現するどの文字よりも小さいと仮定する．T から先頭の文字を何文字か取り除いた部分文字列 $S_i=T[i, n-1]$ を T の接尾辞と

☆5　この仮定は X 軸方向に Rank/Select 辞書を作ることで外すことができる．

☆6　あらゆる文字列をクエリとして調べて，その出現位置をつなげれば元の文字列は復元できる．



i	SA	B	F	接尾辞
0	11	a	$	$
1	10	r	a	a$
2	7	d	a	abra$
3	0	$	a	abracadabra$
4	3	r	a	acadabra$
5	5	c	a	adabra$
6	8	a	b	bra$
7	1	a	b	bracadabra$
8	4	a	c	cadabra$
9	6	a	d	dabra$
10	9	b	r	ra$
11	2	b	r	racadabra$

図 -6　*T* = *abracadabra$* に対する接尾辞配列（列 SA）と Burrows Wheeler 変換（列 B）

$sp \leftarrow 0$; $ep \leftarrow n$;
for $i = m-1 \rightarrow 0$ **do**
  $sp \leftarrow C(P[i]) + \mathbf{rank}_{P[i]}(B, sp)$
  $ep \leftarrow C(P[i]) + \mathbf{rank}_{P[i]}(B, ep)$
  if $sp \geq ep$ **then**
    return NOTFOUND
  end if
end for
return (sp, ep)

図 -7　文字列 *P* [0, *m*−1] を BWT 配列 *B* を用いて検索する

呼ぶ．T のすべての接尾辞を辞書式順序でソートし，それらの添字番号を記録した配列 $SA[0, n-1]$ を T の接尾辞配列と呼ぶ．配列 SA は $[0, n-1]$ の並び替えであり，$S_{SA[0]} < S_{SA[1]} < \ldots < S_{SA[n-1]}$ である（**図 -6** 中の列 SA）．また，各接尾辞配列の直前の文字をつなげた文字列 $B[i]=T[SA[i]-1]$[☆7] を得る操作を T の Burrows Wheeler 変換 (BWT) と呼ぶ（図 -6 中の列 B）．

BWT は可逆変換であり，BWT 後の文字列のみから元の文字列を復元できるという性質がある．また，BWT 後の文字列は同じ文字が連続しやすく，圧縮しやすいという性質がある．これは同じ文字列の直前には同じ文字が出現しやすいという性質からきている．この性質を利用して bzip2 などのデータ圧縮ソフトが作られている．

BWT を利用する上で LF-mapping と呼ばれる操作が重要である．各接尾辞配列の先頭文字をつなげた文字列を $F[i]=T[SA[i]]$ とする．定義から $B[i]$ と $F[i]$ は元の文字列 T 中では隣り合っていることに注意する．このとき，ある文字 c に着目すると，B 上で c の中で i 番目に出現した文字 c は F 上でも c の中で i 番目に出現している．つまり，$\mathbf{select}_c(B, i)$ と $\mathbf{select}_c(F, i)$ は T 上では元々同じ文字を指している．なぜなら，ある文字 c に着目した場合，それらの B 中の順位と F 中の順位を決めるのに使われている後続文字列は F の先頭 1 文字目を除いて同じであるからである．

この関係を利用すると，$c := B[i]$ に対応する文字は F 上では $C(c)+\mathbf{rank}_c(B, i)$ に出現していることが分かる．ただし $C(c)$ は T 中に出現した c より小さい文字の総出現回数である．ここで，$p := C(c)+\mathbf{rank}_c(B, i)$ とおくと，$SA[i]=SA[p]+1$ という関係が成り立つ．つまり，$i := C(B[i])+\mathbf{rank}_c(B, i)$ を繰り返すと，元の文字列 T を 1 文字ずつ前に辿っていく操作に対応する．この操作を LF-mapping と呼ぶ．

この性質を全文検索に利用する．**図 -7** に示したアルゴリズムは，クエリ文字列 $P[0, m-1]$ が出現している接尾辞配列の範囲を返す．$ep-sp$ が出現回数であり，$SA[sp, ep-1]$ が出現している位置である．

図 -6 の右側に，クエリ文字列 $Q[0, 2]$:="bra" を検索した場合の例を示す．このアルゴリズムでは後ろの文字から順に調べる．まず $Q[2]$="a" であり，$sp=C(\text{"a"})=1$，$ep=C(\text{"a"}+1)=C(\text{"b"})=5$ と初期化する．次に，$Q[1]$="r" について調べるが，sp, ep それぞれについて，$sp := C(\text{"r"})+\mathbf{rank}_{\text{"r"}}(B, sp)=10$, $ep := C(\text{"r"})+\mathbf{rank}_{\text{"r"}}(B, ep)=12$ と更新する．この新しく得られた $[sp, ep]$ は，$Q[1, 2]$="ra" に対応する接尾辞の範囲となる．次に $Q[0]$="b" についても同様の操作を繰り返すと $sp=6$，$ep=8$ となり，$[6, 8-1]=[6, 7]$ が Q="bra" が出現している接尾辞の範囲だと分かり，出現位置は $SA[6]=8$, $SA[7]=1$ となり，出現回数は $ep-sp=2$ と求められる．

この **rank** は先ほどのウェーブレット木を利用することで元の文字列長によらない $O(\log \sigma)$ 時間で調べることができる．よって，任意のクエリ文字列 $P[0, m-1]$ の出現回数を $O(m \log \sigma)$ 時間で求めら

☆7　$SA[i]=0$ のとき，$B[i]=T[n-1]$ とする．

ライブラリ名	説明
tx, ux	大量のキーをコンパクトに管理し，共通接頭辞探索などを高速に行う
dag_vector	整数列をコンパクトに圧縮格納し，ランダムアクセスなどを備える
bep, fujimap	連想配列をコンパクトに格納．最小完全ハッシュ関数を利用
wat_array	ウェーブレット木を利用し，アルファベット数が多い場合の **rank**, **select**，矩形探索をサポート

表 -2　代表的な簡潔データ構造とその特徴

れることが示された．出現位置の情報もサンプリングされた接尾辞配列の情報を利用することで復元できることが知られている．

圧縮全文索引のより詳細なサーベイについては文献 3) が詳しい．

簡潔データ構造の実装

簡潔データ構造を利用可能なライブラリを紹介する．今回紹介するソフトウェア（**表 -2**）はすべて C＋＋ で書かれたオープンソースソフトウェアであり，修正 BSD ライセンス☆8 で配布されている．

■ tx，ux

トライ木は文字列からなるキー集合を処理するためのデータ構造で，キーが辞書に含まれているかのみではなく，キーの接頭辞が含まれているかを高速に求めることができる．

tx☆9 はコンパクトなトライ木を構築し利用するためのライブラリであり．商用製品などでも利用されている安定したライブラリとなっている．

tx は作業領域量が小さいことが特徴であり，従来のトライ木の実装に比べ 1/4 ～ 1/10 のサイズで辞書を保持することができ，数億～十億キーワードなど大規模な辞書を扱うことが可能である．内部ではトライ木の木構造を LOUDS で表現し枝のラベル情報や終点情報をそれに合わせて格納している．tx は次の操作をサポートしている．

- PrefixSearch：クエリ文字列の接頭辞に最長一致する文字列の ID を返す．
- Lookup：ID を用いてキーを復元する．
- CommonPrexSearch：クエリ文字列の接頭辞に一致する文字列をすべて列挙する．
- PredictiveSearch：接頭辞がクエリ文字列に一致するキーをすべて列挙する．

その後，tx の圧縮率向上を目指し ux が開発された☆10．

トライ木の表現では，共通する接頭辞は共有化されるが，接尾辞は独立に格納されている．最後の節点から，枝分かれのないパスの部分を tail と呼ぶ．この tail を共有化するために，ux では tail を抜き出し，tail の文字列を逆向にした文字列からなるトライ木を作り tail 情報を格納している．これにより，tail の中で共通する接尾辞部分が共有化され，効率的に格納される．ux は tx と比べ約半分の辞書サイズを達成している．

■ dag_vector

次に紹介するのが，配列に関する簡潔データ構造である dag_vector☆11 である．このライブラリは配列を圧縮して格納したまま定数時間でのランダムアクセスが可能である．また，すべて要素の末尾追加が可能である．

dag_vector のライブラリ群を以下に示す．

- dag_vector：直接アクセス可能なガンマ符号．ガンマ符号は，正整数 $x \geq 0$ を $2\log_2(x+1)+1$ ビットで格納することができ，小さい値が多い場合に効率的に格納できる．
- sparse_set：疎なビット配列を効率よく保存する．長さが n で 1 が m 個存在するビット配列を $2m+m\log_2(n/m)$ ビットで格納することができる．
- comp_vector：任意のオブジェクト配列について，頻出する要素に対しては短い符号，そうで

☆8　三条項 BSD ライセンスとも呼ばれる．著作権表示，ライセンス条文，無保証の 3 点をドキュメント等に記載しておけば，複製・改変して作成した場合でもソースコードを公開せずに頒布できる．
☆9　http://code.google.com/p/tx-trie/
☆10　http://code.google.com/p/ux-trie/
☆11　https://github.com/pfi/dag_vector/

ない要素に対しては長い符号を割り当てることで，全体長を圧縮する．

■ bep，fujimap

Bep[☆12]は大規模な連想配列を扱うためのライブラリである．連想配列は文字列からなるキーを利用して任意のオブジェクトを登録・参照できるデータ構造である．Bep は内部に最小完全ハッシュ関数[☆13]を利用し，従来の連想配列の実装に比べて作業サイズは非常に少ない．具体的には，キー 1 つあたりのサイズは約 3 ビットである．

fujimap[☆14]は False Positive を許した上でキー自体も保存しない連想配列である．登録済みのキーについては $[0, n-1]$ 中の値を割り当て，登録されていないキーについては一定の確率で未登録として報告することができる．

■ wat-array

ウェーブレット木を実装してさまざま操作を備えたライブラリが wat-array である[☆15]．たとえば，任意の連続する範囲中の，任意番目に大きい値などを，範囲，順位によらず高速に調べることができる．

たとえば，配列 $A[0, n-1]$ に対して wat-array を構築した場合，次の操作を配列長，クエリの範囲長によらず $O(\log \sigma)$ 時間で実行することができる．

wat-array のサイズは入力データと同じであり，また元の配列を wat-array だけから復元することもできる．

- 文字 $A[i]$ の復元，任意の文字 c の i 番目の出現位置 ($\mathbf{select}_c(T, i)$) を返す．
- 任意の連続した範囲内にある最大値／最小値／k 番目に大きい値，またそれらの出現位置，頻度を返す．
- 任意の連続した範囲内にある指定した文字 “c” の出現回数，“c” 未満／“c” より大きい文字の出現回数を求める．
- 任意の連続した範囲中で最も多く出現した文字を返す．計算時間はほとんどの場合 $O(\log \sigma)$ 時間であり最悪 $O(\sigma)$ 時間必要である．

アプリケーション例

これまで紹介してきた簡潔データ構造は実際の現場でも多く利用されている．ここでは簡潔データ構造を利用したアプリケーション例をいくつか紹介する．

■ 検索エンジン

筆者が所属している (株) Preferred Infrastructure[☆16]では簡潔データ構造を，検索エンジン Sedue を中心にさまざまな部分で利用している．

たとえば，Sedue は全文検索用の索引や，文書の特徴情報抽出に利用するキーワード辞書，また属性情報なども簡潔データ構造を利用して効率的に格納している．これにより数十万から多い場合は数千万キーワードからなる辞書をメモリ上に格納し，文書からのキーワード抽出や特徴情報抽出を高速に行うことが可能となっている．

■ データ管理

大量の文字列情報や，ログ情報などを簡潔データ構造を利用して効率的に保持することができる．

たとえば，Google IME はかな漢字変換用の辞書情報を LOUDS 表現を利用してコンパクトに格納し，大規模辞書を利用したかな漢字変換を実現している[5]．

今回紹介していない種類のデータ構造に対する簡潔データ構造も大規模データを処理するために利用されている．たとえば，グラフ構造も簡潔データ構造で表現することで，数十億頂点からなるグラフ情報も主記憶上で格納でき，グラフを辿ることができることが知られている．

☆12 http://www-tsujii.is.s.u-tokyo.ac.jp/511hillbig/bep-j.htm
☆13 ハッシュ関数が完全であるとは，与えられたキー集合に対し，すべてのハッシュ関数値が異なることが保証されていることであり，最小とはキーの種類数が n のときハッシュ関数の値域が $[0, n-1]$ であるような場合である．
☆14 http://code.google.com/p/fujimap/
☆15 http://code.google.com/p/wat-array/
☆16 http://www.preferred.jp

■ データ圧縮

ゲノム配列やソースコードレポジトリ，機器のログデータは，同じ文字列の繰り返しが多く非常に冗長性が高い．こうした冗長性が高い文字列は適切な表現方法を利用すれば高い圧縮率が期待できる．そのように圧縮したまま様々な処理が行えるようなデータ構造が近年盛んに研究されている．

LZ 法は，このような冗長性が高いデータを効率的に表現できる方法である．たとえばLZ77 と呼ばれる方法では，繰り返し文字列を（前回出現した相対位置，一致長）に置き換えることで効率的に表現する．これらのLZ 法は現在，データ全体を圧縮，復元するだけでなく，圧縮したまま高速な探索を実現できるようになっている．

また，従来のデータ圧縮手法は，データを利用するときにはデータを復元する必要があった．データ中のほんの一部分だけを扱う場合でも，すべてのデータをまず復元し，それから必要な部分を取り出さなければならかった．

簡潔データ構造を利用することにより，全体を復元しなくても任意の文字列に定数時間でアクセスできるようになるデータ圧縮を作ることができる．このようなデータ圧縮は，圧縮しないときとまったく同じようにデータを透過的に扱えるようになるため，透過的データ圧縮と呼ばれる．

たとえば，LZEnd を利用した手法[2)]では圧縮した後の文字列を任意の位置から復元した文字列長のオーダーで復元することができる．

今後の展望

簡潔データ構造は，高速な処理と省スペースでの格納という 2 つの目標を同時に達成するデータ構造である．この簡潔データ構造を利用することで，これまで考えられなかった大規模データを高速な主記憶上で処理することができるようになる．

今後，データ処理は大きく 2 つの方向に向かうと考えられる．1 つはクラウド型のデータ処理で，すべてのデータをデータセンタなど 1 カ所にまとめ，その後にまとめて計算を行う方法である．近年のGoogle や Facebook，Amazon など世界的な Web サービス企業を中心にこうした流れは今後も加速していくと考えられる．

もう 1 つはモバイルデバイスや端末などで，データが生成された場所に近いところで処理を行い，上の階層には必要な情報のみ伝えるような処理である．今後データの生成量が増えるにつれてネットワークの回線容量がボトルネックとなり，すべてのデータが集められなくなってくる．こうした場合，PC クラスタと比べて非力な環境でのデータ処理が求められる．

どちらの場合でも，マシンの性能に比べて，処理しなければならないデータ量は膨大である．こうした場合にデータをより効率的に扱うことができる簡潔データ構造は重要となる．

現在の簡潔データ構造の大きな課題は動的更新のサポートである．データの追加や削除，変更を行える簡潔データ構造も存在するが，まだ実用化はされていない．動的更新をサポートしていない簡潔データ構造と比べても作業領域量と計算量が同じような簡潔データ構造をいかに作るかが重要である．

今後広い分野や問題での簡潔データ構造の利用が望まれる．

参考文献

1) Arroyuelo, D., Canovas, R., Navarro, G. and Sadakane, K. : Succinct Trees in Practice, Proc. 11th Workshop on Algorithm Engineering and Experiments (ALENEX'10), SIAM Press, pp.84-97 (2010).
2) Kreft, S. and Navarro, G. : LZ77-like Compression with Fast Random Access, Proc. 20th Data Compression Conference (DCC), pp.239-248 (2010).
3) Navarro, G. and Mäkinen, V. : Compressed Full Text Indexes, ACM Computing Surveys, Vol.39 (2007).
4) Okanohara, D. and Sadakane, K. : Practical Entropy-Compressed Rank/Select Dictionary, Proc. ALENEX (2007).
5) 花岡俊行，田畑悠介，向井　淳，小松弘幸，工藤　拓：辞書と言語モデルの効率のよい圧縮とかな漢字変換への応用，言語処理学会第 17 回年次大会 (2011).

(2011 年 10 月 21 日受付)

■岡野原 大輔（正会員） daisuke.okanohara@gmail.com
1982 年生．2006 年（株）プリファードインフラストラクチャーを共同で創業．2010 年東京大学情報理工学研究科コンピュータ科学博士課程修了．現在プリファードインフラストラクチャー特別研究員．統計的自然言語処理，機械学習，データ構造に関する研究開発に従事．情報理工学博士．