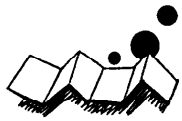
解　説

基本データ構造とアルゴリズム†



野　下　浩　平††

1. はじめに

データ構造の工夫が数多くの高能率な算法（アルゴリズム）のエッセンスになっている．本稿では，例を用いて，データ構造の工夫が自明でない高速算法にどう結びつくかをみていこう．データ構造とアルゴリズムの研究については，数多くの優れた本（たとえば参考文献1),18),19),20),30)）がでているので，この分野全体の様子は，そちらで調べられたい．

本稿の構成は次の通り．第2章では，普通の計算機で各種のデータ構造を表現する方法のごく常識的なものを概説する．第3章では，高速算法の設計に役立つデータ構造の1例として，最近発表されたフィボナッチ・ヒープをやや詳しく紹介する．第4章では，組み合わせ論的算法のうち，ソートやサーチなどに関する最近の結果の1部をながめるとともに，特にグラフの最短経路問題のための高速算法を調べる．第3章のデータ構造は，ここで応用される．第5章では，まとめとしてデータ構造研究の動向の中で，3つの考え方を紹介する．

2. 線型リスト・木・グラフ

本章では，次章以降の準備をかねて，基本的なデータ構造をざっと復習しておこう．

構造をもったデータといえる一番素朴なものは，一方向の線型リストであろう．これは，箱（レコード）の集りを表すのに，たとえば，図-1のようにポインタを用いて各箱を順次繋いだものである．線型リストの参照の仕方に制限をつけて，片方の端を固定して，他方の端でのみ箱の追加 *insert* や削除 *delete* の操作を行うのがスタックであり，片方の端で *insert*，他方の端で *delete* を行うのがキューである．両方の端で *insert* と *delete* を許すのがデクである．（もちろんこれら3つのものは，配列に連続的につめる方が時間的にも記憶領域的にも有利なことが多い．）なお，コーディングの都合を考えると，普通“ヘッダ”とよばれるダミーの箱（番兵）を1つ追加して，空の線型リストが箱1つあるように実現する方が便利である．また，リストの終端を **nil** にしないで，全体を環状に考える方が好都合なことが多い．これは，特に環状リスト（の一種）とよばれる．

一般の線型リストでは，リストの途中を参照することが想定されるが，一方向リストでは，着目している箱を指している箱をみつけたり，その箱を削除したりするのは容易でない．普通はリスト全体の大きさに比例する時間がかかる．それで各箱に手前の箱を指すポインタを追加して，前後を対称的に取り扱えるようにしたのが両方向の線型リストである（図-2）．これで，各箱にポインタ1個分ふやす犠牲を払って，一方向リストの不便さが解消できる．なお，コーディング上は，ヘッダの設定がやはり大切であろう．なお，2つの前後のポインタを1個に圧縮する技法でマジックリストと呼ばれるものがある[12]．不便なところもあるが，アイデアは面白い．

線型リストは，次にみるように木やグラフの表現に直接応用されているが，それ自身にも面白い応用例が数多くある（たとえば参考文献18) 参照）．もっとも，線型リストは，データ構造的な発想があまりなかった時代からよく使われてきている．たとえば，一方向リストは，1パスのアセンブラでチェイン技法に使われている．これは，まだ定義されていない記号番地を線型リストで繋いでおいて，後で定義が現れて番地が確定すると，それをたどって番地を書き込むというものである．

次に基本的なデータ構造としては，木と2分木であろう．2分木の表現法は，その定義に従って，各箱に左右の子を指す2つのポインタを入れるというのが基本的である．また，木あるいは木の集合（森）は，2分木で表現するのが，ポインタによる方法のなかで最

† Data Structures for Efficient Combinatorial Algorithms by Kohei NOSHITA (Department of Computer Science, Denkitusin University).

†† 電気通信大学計算機科学科

も標準的である．図-3 に例を示す．これで，木に関するさまざまな操作が2分木の操作として表現される．(なお，いくつかの本では2分木は木の特殊なものと定義されているが，Knuth 流に，2分木と木は別のものと定義する方が都合の良いことが多い[18]．)

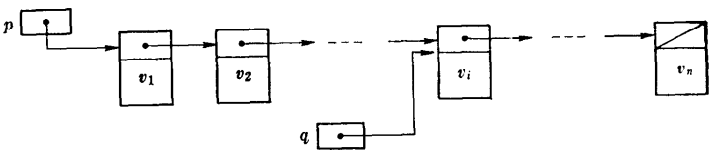


図-1 一方向の線型リスト (▨ は nil)

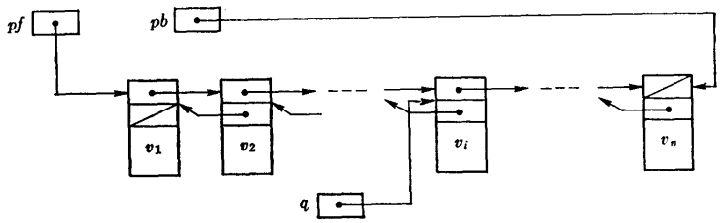


図-2 両方向の線型リスト

ここでもやはり線型リストと同様なことが考えられる．たとえば，2分木において，右下がりにたどる点の集合は，木ではある点の子の集合になっている．特に，2分木の根に対する右下がりの点列は，木の根の集合である．これらの集合は，一方向リストで表されていると考えてよいので，これを両方向リストにすることが思いつく．このように，木または2分木に対して施される操作の種類によっては，標準的な表現法に対して，いろいろな追加や変形が考えられる．ここでは，木の集合の表現法の例として，図-3 左を図-4 のように表現するものを示しておく．ここで，各点には，このための前後のポインタ b, f の他，親へのポインタ p，子のどれかへのポインタ s を入れておく．これだけ用意しておけば，たとえば，親をみつける操作，子の集合から1つの点とその子孫を独立させて，新しい木とするという操作などが一定時間で実行できる (第3章参照)．木と2分木の表現法と操作については，ゴミ集めなども含めて話題は豊富にあるが，参考文献 18) に任せよう．

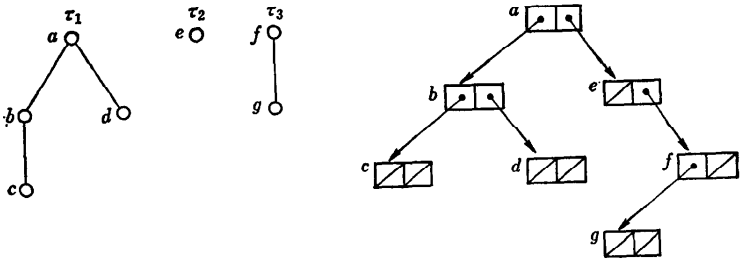


図-3 木の集合 $T=\{\tau_1, \tau_2, \tau_3\}$ と2分木表現

次は，木もその一種であるが，一般のグラフの表現法をおさらいしよう (たとえば参考文献 1))．いま，有向グラフ $G=(V, E)$ が (平行辺も自己ループもない) 単純なものとする．今後，n で点の個数 $|V|$, m で辺の個数 $|E|$ を表す．さて，最も素朴な表現法は，隣接行列とよばれるもので，$n \times n$ の配列 a を用意して，

$(i, j) \in E$ ならば $a[i, j]=1$,
$(i, j) \notin E$ ならば $a[i, j]=0$

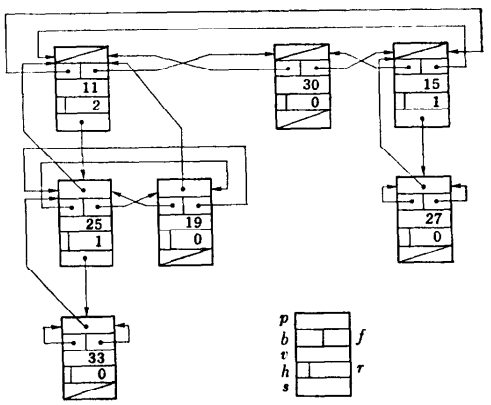


図-4 木の集合 T の表現法 (フィールド p, b, f, s は各々親，直前，直後，子の1つを指す．v, h, r は第3章で使用．)

と約束するものである．この表現法にも良いところがあるが，一般に，$\Omega(n^2)$ の記憶領域が必要な上に，たとえば，ある点に隣接する点の集合をみつけるのに，

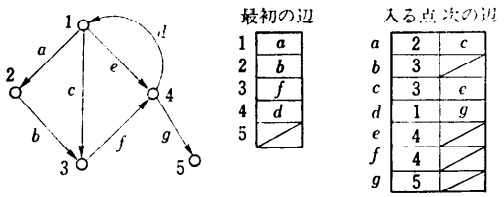


図-5 有向グラフとそのリスト表現

普通 $\Omega(n)$ 時間かかる．それで，大きいグラフ，特にそれが疎であるような場合，普通，この表現法より次のリスト構造によるほうが良い．

この方法では，各点ごとに，そこから出ている辺の集合を線型リストで表し，各辺には，それが入る点を憶えることにする．たとえば，図-5 左のグラフは，図-5 右のようになる．ここでは，ポインタは，配列の添字で表されている．これで，グラフの構造を無駄なく表現したことになり，たとえば，各点に対してそれに隣接している点の集合をみつけるのに，1つの点当り一定時間しかかからないことになる．また記憶領域の大きさについても，$O(m+n)$ ですむ．

これまでの議論からすぐ思いつくことであろうが，この表現法はもっと一般的にできる．まず，各点に対して，出る辺の集合だけでなく，入る辺の集合も表現することが考えられる．さらに，辺の削除や挿入を容易にするために，線型リストを両方向にする．このようにすれば，点については，出る辺の集合の先頭と末尾の辺2つ（環状リストなら辺1つ），入る方の2つ，合計4つのポインタ，一方，辺については，関与する2つの集合の前後ポインタ4つ，入る点と出る点の2つ，合計6つのポインタが必要となるが，全体としては，やはり $O(m+n)$ の記憶領域ですむ．

無向グラフについては，1つの辺を往復2つの有向辺とみなし，一旦有向グラフにして，上述の方法で表現する手がある．ただし，どの2つの辺がもとのグラフで同一であるかを知る手当てを施しておくことにする．

3. 木の応用例：フィボナッチ・ヒープ

本章では，木構造の例として，最近発表されたフィボナッチ・ヒープ (Fibonacci Heap, 略してFヒープ)[11] をやや詳しく解説しよう．

実数値の集合（一般的には多重集合）S が空からはじめて，次の操作が繰返し適用されるというサーチ問題の一種，順位キュー（ヒープ）問題を考えよう．

（1） *insert*(x)：S に新しい要素 x を追加する．
（2） *deletemin*：S からその最小値を削除する．
（3） *decrease*(w, y)：S の要素 x を小さい $y(y \leqslant x)$ で置き換える．ただし，S 中での x の位置 w が与えられているものとする．

このような操作を考える意味については，第4章で説明することにして，この問題を能率良く解くためのデータ構造を考えよう．ここでの目標は，次の定理を示すことである．

定理 (Fredman-Tarjan (1984))
$S=\phi$ ではじめて，n 回の操作を施したときの全計算時間は，最悪の場合でも

$$O(n+k\log_2 n)$$

である．ここで，k は *deletemin* の回数である．

さて，集合 S は，図-4 のように木の集合として表すことにするが，各点には，4つのポインタの他に，値そのもの v，子の数を示すランクとよぶ整数値 r，フラグ1ビット h を追加している．そして，木の集合全体として，次のヒープ条件を満たすものとする：

任意の2点 x, y に対して，$x=y\cdot p$ ならば，$x\cdot v \leqslant y\cdot v$，つまり（親の値）≦（子の値）である．ここで '$a\cdot b$' の書き方は，'$a$ の b' と読む．さらに，S の最小値の点は，*root* で表し，いつもその位置がわかっているようにする（図-4 では値 11 の点が *root*）．

このようなデータ構造をFヒープとよぶことにするが，Fヒープに対する操作の算法を図-6 に示す．ここで，R は木の根の集合を表す．なお，奇妙な数 '$1.45\log n$' については，後で説明する（対数の底は省略すると2と約束する）．この算法の理解の助けのために，図-7, 8 に例を示す（図-8 で，値 15 の点のフラグが *true* とする）．

各操作は，次のような考えを実現している．

insert(x) は，値 x の根1つでなる木を R に追加するだけである．

deletemin は，最小値をもつ根を取り除いた後，残った木の集合に対して，結合という操作を繰り返す．結合は，同じランク（子の数）をもつ根の間でのみ起こり，ヒープ条件をみたすように，値の大きい方の根が小さい方の根の子になる．

decrease(w, x) は，値が減った点を（親から切り離すことにより）新しく木として独立させる．この際，（根でない）点は，自分のいくつかの子のうち，丁度2つの子が木として独立すると，親である自分も木として独立するという約束事に従う．これを実現す

```
insert(x):
  begin 値 x の新しい点を作り，R に追加する；
        x<root•v ならば，root を改める
  end
deletemin:
  begin R に root の子の集合を追加して，root を削除する；
    {結合ステップ：同じランクの根が2つ以上ある限り，根の結合を繰
    り返す．最初，root は仮に値 ∞ の点としておく．}
    for i:=1 to ⌈1.45 log n⌉ do P[i] を空にする；
    for R の各点 w do
      begin
        while P[w•r] に他の点 w' がある do
          begin w•v と w'•v の値を比較して，小さい方の点を wm，
            大きい方の点を wM とよぶ；
            wm を R に残し，wM を wm の子の集合に追加する；
            wm•r:=wm•r+1；wM•h:=false；
            P[w•r] を空にする；wm を w とよぶ
          end；
        w•v<root•v ならば，root を改める；
        P[w•r] に w をいれる
      end
  end
decrease(w, z){z≦w•v}:
  begin {減少} w•v:=z；
    z<root•v ならば，root を改める；
    点 w が根ならば (w∈R)，これで終了する；
    w をその親 u から切り離し，R に w を追加する；
    u•r:=u•r−1；
    while u•h∧(u∉R) do
      begin {切断} u を改めて w とおく；
        w をその親 u から切り離し，R に w を追加する；
        u•r:=u•r−1
      end；
    u•h:=true
  end
```

図-6 3操作の算法

るには，(根でない) 点は，フラグ1ビットをもち，過去に子を1つ独立させたことがあるかどうかを憶えておけばよい．

次に計算時間を解析する．ここでも，最近 R. Tarjan を中心に盛んに研究されている'ならした計算量 (amortized complexity)'[31] の考え方が使われる．いま，n 個の操作の列 $\sigma=\sigma_1\sigma_2\cdots\sigma_n$ に対して ($n\geq 2$)，各操作 σ_i の運転 (実行) のために予算 a_i 円用意することにして，その一部は運転の実費 t_i に使い，残りを預金することにする．実費 t_i の捻出には，預金の一部を引き出すこともある．(場合によっては借金してもよい．) さて，預金0円ではじめて，列 σ の実行が終了したとき，もし預金が残っていれば，当然

$$t_1+t_2+\cdots+t_n\leqq a_1+a_2+\cdots+a_n$$

が成り立っている．このことより，予算 a_i の総和をみつもれば，実際の運転費用全体 (つまり全計算量) の上界がえられることになる．いまの定理の場合，詳しくは，*insert*, *decrease* に c 円，*deletemin* に $c\log n$ 円の予算を見積もれば (c は定数)，

$$a_1+a_2+\cdots+a_n\leqq c(n_I+n_M\log n+n_D)$$

が成り立つ．ここで，n_I, n_M, n_D はそれぞれ *insert*, *deletemin*, *decrease* の回数である ($n=n_I+n_M+n_D$)．

この意味で，各 a_i (またはその総和) は，ならしてみたときの計算量ということができる．そして，列 σ はどんなものでもよいので，'最悪の場合' のならした計算量を考えていることになる．

それでは，各操作の予算とその使い途をみていこう．

(1) *insert*：実費は1円 (一定額) ですむが，それ以外に新しくできた根のために1円預金する．予算は2円である．

(2) *deletemin*：予算は $6\log n$ 円あれば十分である．結合部分以外については，実費は1円ですむ．これは予算で充てる．また，Rに追加される子 (新しい根) の数は，(後でみるように，) 高々 $1.45\log n$ なので，この分高々 $1.45\log n$ 円は予算から預金に

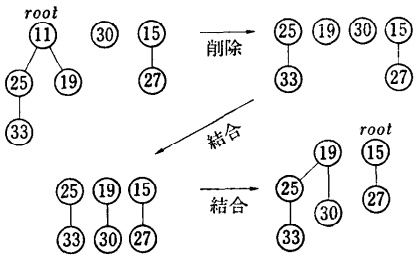


図-7 *deletemin* の実行例

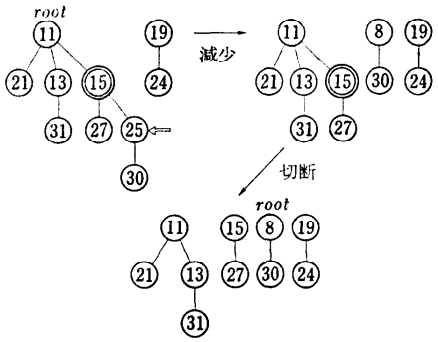


図 8 *decrease*(25, 8) の実行例

まわす．さて結合部分の第1の **for** ループは，1.45 $\log n$ 円かかるが，これは予算で充てる．第2の **for** ループは，Rの個数 $|R|$ と2つの根の結合の個数の合計を考慮すればよい．1回結合が起こると必ず根が1つ減るので，結合の回数は $|R|$ 以下である．したがって，合計 $2|R|$ 回の操作のためには，$|R|$ 円かかるとしてよいので，この実費には預金を引き出すことにする．ここでのポイントは，いつも預金に少なくとも $|R|$ 円残っていることである．最後に，結合の結果できた木の総数（新しい $|R|$）は，高々 1.45 $\log n$ なので，やはり予算から 1.45 $\log n$ 円預金しておく．

（3） *decrease*：切断部分以外については，実費は1円とみてよい．また，Rに根が追加されると，忘れずに1円預金しておく．さて切断の部分であるが，切断は，以前にフラグが立てられた（根でない）点に限って起こるので，切断の回数は，高々立っているフラグの個数である．それで，以前の *decrease* でフラグを立てるつど，2円預金されていたとすると，いまの1回の切断の実費1円をこれで充て，残り1円を根になった点の分として預金に残すことができる．それで，切断部分の最後のフラグを立てる場合，後のため，予算から2円預金しておく．結局 *decrease* では，高々4円の予算でよい．

以上併せると，*insert* と *decrease* が一定額 c 円，*deletemin* が $c \log n$ 円の予算ですむことになった．

あと残されたのは，点の子の数（ランク）がいつも 1.45 $\log n$ 以下ということである．これには，根以外の点は，結合によってのみ生まれ，結合は，同じランクの根同士でしか起こらないことにまず着目しよう．根でない点のランクは，切断によって減るが，減っても高々1だけである．それで，ある点のもつ k 個の子 $s_1, s_2, \cdots, s_k$ が子になった時刻順に並んでいるものとすると，次が成り立つ：

$$s_i \cdot r \geqslant 0, \quad s_i \cdot r \geqslant i-2 \quad (i=2, \cdots, k)$$

よって，ランク k の根をもつ木は，少なくとも S_k 個の点をもつものとすると，$S_0=1$, $S_1=2$ で，

$$S_k \geqslant 2+S_0+S_1+\cdots+S_{k-2} \quad (k \geqslant 2)$$

が成り立つ．フィボナッチ数 F_k と見較べると，直ちに次がえられる．

$$S_k \geqslant F_{k+2} \geqslant ((1+\sqrt{5})/2)^k$$

それで，$n \geqslant S_k$ であることより，次が成り立つ：

$$k \leqslant \lceil 1.45 \log n \rceil .$$

以上で定理は証明されたが，定理の上界をもっと詳しく書けば，$O(n_I + n_M \log n_I + n_D)$ となることがわかる．

普通，順位キュー（あるいはヒープ）といわれるデータ構造は，上でみた3つの操作の他，

（4） *findmin*：Sの最小値をみつける，

（5） *delete*(w)：Sの要素 x を削除する．

ただし，S中で x の位置 w が与えられているものとする

の操作も施される．上の説明からすぐわかるように，*findmin* が $O(1)$ 時間，*delete* がならして $O(\log n)$ 時間で実行できる．*delete* は，*findmin*, *decrease*, *deletemin* を組み合わせれば実現できることに注意．なお，順位キューの集合を考えて，2つの順位キューを併合する操作 *meld* をふくめるという一般化された順位キュー問題に対しても[1]，Fヒープを容易に拡張できる（*meld* は $O(1)$ 時間でできる）[11]．

4. 組み合わせ論的算法

ソート（整列，併合，選択など）とサーチ（順位キュー，各種の辞書，UNION-FIND など）については，その高速算法はデータ構造の工夫によるところが大きい．この分野の73年頃までの研究成果は，Knuthの第3巻[19]に集大成されている（参考文献1）も参照）．それ以来も，特にサーチについては，バランス木やハッシュ法などの研究が相変わらず盛んである．しかし，この方面の基本的な常識と研究方法はかなり固まってきているようにも思える．最近の本[20]と上記の本を見較べてみると面白い．

最近の結果の中で，注目すべきものの例としては，順位キューでは，前章の F ヒープ，（広い意味での）辞書では，スプレィ木 (splay tree)[28]があげられる．整列については，Quicksort のような既知の算法の詳しい解析や改良は当然としても，その他少数であるが新しい算法も提案されている．たとえば，Smoothsort[6] は，Heapsort と同様に最悪の場合 $O(n \log n)$ 時間，作業用領域の大きさ $O(1)$ の算法であるが，さらにもともと整列されているデータについては，$O(n)$ 時間ですむ．その実際的評価や改良については，参考文献26) を参照．（ところで，Heapsort については，多くの本にもとの版[9]が紹介されているが，参考文献19) の練習問題 5.2.3.18 にある版の方が良いと思う．）また，分配分割法 (distributive partitioning) は，当時研究者の間で一寸話題になった[7),21]．これは，0-1 区間の一様分布をふくむかなり一般的な確率分布からとった n 個の実数値の整列が平均で $O(n)$ 時

```
{P は等高線が通過した点の集合，つまり各点 v(∈P) に
対して最短距離 δ*(v) が確定している．T は等高線より
外の最前線にある点の集合，つまり各点 v(∈T) に対し
て，P の中より辺1つで v に至る道のうちの最短経路の
距離 δ(v) が計算されている．}
begin P:=φ; T:={s}; δ(s):=0;
  while T≠φ do
    begin T で δ(v) の最小値をもつ点 v* を見つける;
      T:=T−{v*}; P:=P∪{v*}; δ*(v*):=δ(v*);
      for (v*,u)∈E (u∉P) となる各点 u do
        if u∈T then δ(u):=min{δ(u), δ(v*)+w((v*,u))}
        else begin T:=T∪{u}; δ(u):=δ(v*)+w((v*,u))
             end
    end
end
```

図-9 Dijkstra 法

間でできるというものである．最悪の場合は $O(n \log n)$ 時間である．ただし，この算法の一族は，いわゆる比較型の算法，すなわち比較決定木で表現できるものではなく，値の大きさの範囲を陽に利用している．

グラフやネットワーク問題へのデータ構造の応用については，73年頃までの基本的な結果が参考文献1)にまとめられている（15）も参照）．その後の進歩については，17),20),30) などをみると，その様子がよくわかる．この分野は，最近急速に発展している計算幾何学，集積回路設計などとも直接関係しており，全体として，その発展は広く速い．

ここでは，特にネットワークの最も基本的な問題の一つ，最短経路問題について，少し詳しく調べてみよう．

無向グラフ $G=(V, E)$，出発点 $s(\in V)$，辺の距離 $w: E\rightarrow$(非負実数) が入力として与えられるとする．いま G は，単純で連結としてよいので，点の数 $n=|V|$，辺の数 $m=|E|$ に対して，$n-1\leqslant m\leqslant n(n-1)/2$ が成り立つ．これで出発点 s から各点 t へ至る最短経路とその距離を求める問題は，'単一出発点・非負' の問題とよばれる．この問題に対しては，Dijkstra 法とよばれる古典的な解法がよく知られている[5]．この算法は s から始めて，等高線（等距離線）をだんだん広げていくという素朴な考え方に基づいており，それを逐次的な算法として実現したものである．**図-9** に算法の概略を示す（グラフの表現は図-5 を参照）．

この算法では，最短経路を陽には求めていないが，必要ならば，各点ごとにそこへの最短距離を定める辺を憶えておけばよい．

さて，Dijkstra 法では，集合 T で δ の最小値をもつ点 v を削除する操作，T の $\delta(u)$ の値を小さい値に更新する操作，T へ u を追加する操作の繰り返しで成り立っているので，前節で調べた順位キューがぴったりである（Fヒープの各点の v には点の番号をいれること）．それで，*deletemin* が n 回，*decrease* が m-n 回，*insert* が n 回実行されることより，単一出発点・非負の問題は，最悪の場合でも

$$O(m+n \log n)$$

時間で解けることになる[11]．なお，この考察は，最短木 (minimum spanning tree) 問題に対する Prim 法[5),27)]にも適用でき，やはりこの問題も同じ計算量で解ける．最短木の問題は，算法の改良が数多く発表されてきているが[30]，最近Fヒープを用いる凝った算法によって，$O(m \log \beta(m, n))$ 時間で解けることが報告されている[11]．ここで，

$$\beta(m, n)=\min\{i \mid \log^{(i)} n \leqslant m/n\},$$

$\log^{(i)} n=\log \log^{(i-1)} n$, $\log^{(0)} n=n$ である．

さて次に，最短経路算法の工夫の歴史を簡単に振り返ってみよう．70年頃より前は，順位キューに木構造を使うことはあまり知られていなかったようである．集合 T の表現として，1次元配列に各点をそのままいれる素朴な方法を用いると，最悪の場合，$\theta(n^2)$ 時間の算法になる．また，T を距離に関してソートして，線型リストで保持するというもう1つの素朴な方法によると，最悪の場合，$\theta(mn)$ 時間になるが，グラフが疎であれば，計算実験や，実際の場での応用でかなり良い結果を生むとされていた[14]．72年頃，順位キューに木構造を使うことがほぼ同時になんか所かで発表されたが（15）参照），いまでは，たとえば，一番簡単な完全2分木のヒープを用いる算法は，練習問題になっている[1]．

このヒープによれば，上記3つの操作が全て $O(\log n)$ で実行できるので，全体として $O(m \log n)$ 時間で解ける．完全2分木を自然に完全 α 分木に拡張した α ヒープを用いて，*deletemin* の負担を $O(\alpha \log_\alpha n)$ 時間に増やし，*insert* と *decrease* を $O(\log_\alpha n)$ 時間に減らし，$\alpha=m/n$ に選ぶと，全体として，

$$O(m \log n/\log (m/n))$$

時間で解ける．この算法は，Johnson 法[16]とよばれているが，前の素朴な方法を計算量の意味ですべて含んでいて最近までの最良の結果であった．この結果は，最悪の場合のものであるが，平均的には，α を約 $\log_e (2m/n)$ に選ぶのが良いという理論的な話もある[25]．ここで，平均計算量は，辺の距離がどんな確率分布であっても，その分布から独立に選ばれていればよいという意味できわめて一般的に定義される．実際，計算

実験でも（乱数データによる平均的な場合）これはなかなか良い（$\alpha=2$ も $\alpha=m/n$ も良くなく，素朴なものはもっと悪い）．

さて，上で示した，Fredman-Tarjan の結果は，$m=n^{1+\varepsilon}$(ε は定数，$0<\varepsilon<1$）や，$m=cn$(c は定数 $c\geq 1$）の場合，オーダの意味で改良になっていないが，たとえば，$m=n\log n$ では，従来の $O(n\log^2 n/\log\log n)$ が $O(n\log n)$ に改良されている．しかし実際に，図-6 の算法をコーディングしてみると，普通の計算機で比例定数がかなり大きいことがわかる．それで α ヒープをFヒープに置き換えるのが，現時点でいつも実用的な意味で改良になるかどうかは，もう少し様子をみる必要があろう．

なお，これまでの算法はすべて距離同士の比較を演算の単位として考えてよい Dijkstra 法をもとにしているが，この型に入らない算法としては，Moore 法とよばれる算法[22]の一族やその変形・拡張が数多く発表されており（参考文献 17）参照），計算実験などにより，こちらの方向で改良を重ねた算法のほうが実用的には良いという報告もあるので[13]，現場にいる人は検討されたい．

以上は，単一出発点・非負の問題であるが，辺の距離に負のものを許す場合は，あまり良い方法はない．（この問題は有向グラフに限られ，負の距離の閉路をもたないことが仮定される．この仮定は，大抵算法の副産物として検査できる．）一番良いものとされている算法の1つは，Bellman-Ford 法といわれているもので，$O(mn)$ 時間で走る[17]．これは経路をなす辺の数に関する帰納法による簡単な算法である．

もう1つの基本的な問題として，すべての点の対の間の最短経路を求める '全点対' 問題がある．これは，出発点を次々に取り換え，単一出発点の算法を n 回適用すればよいが，それ以外にこの問題特有の算法がある．辺の距離に負のものも許す場合，Warshall-Floyd 法[8]が標準的であり，$O(n^3)$ で走る．これは，$n\times n$ 距離行列を用いるもので，点の番号に関する帰納法の考え方に基づいている．もう1つの算法は，まず Bellman-Ford 法で1つの出発点を決めて単一出発点問題を解き，その結果を用いて，辺の距離が非負の問題に変換して，出発点を $(n-1)$ 回取り換えて Dijkstra 法（の最高速のもの）を使うという算法があり，これは，全体として

$$O(mn+n^2\log n)$$

時間で走る．

最短経路問題にはその他，平均的に高速な算法に関する一連の結果など面白い話題が多いが，紙数の都合で省略する．

5. おわりに

最後に，データ構造の研究に関するいくつかの新しい方向をながめてみよう．

まず，第3章で解説したような一連の操作全体に対する計算量を扱う解析手法として，ならした計算量の考え方は，今後も研究されていくであろう（参考文献 2), 29), 30), 31) なども参照）．たとえば，この考え方によって，Fヒープが開発され，最短経路問題，最短木問題，ある種のマッチング問題などの計算量の上界が一斉に下がった[11]．

また，良い算法といっても，それが単にオーダの意味で良いという場合，たとえば，問題のサイズが現在の計算機の規模と桁違いに大きくならなければ，良さがでないという場合，現在の普通の計算機で実用的に良い算法を開発するということも工学的にやはり重要であろう．たとえば，最大流問題の高速算法[28),30)]で使われているスプレィ木は，"自己調整 (self-adjusting)" という性質をもっている．これは，データ構造の形のバランス度を示すような情報を持たないですむものであり，その情報を扱う必要のない分だけ計算時間の比例定数の減ることが期待できる．たとえば，よく知られている AVL 木には，各点に2ビットのバランスをとるための情報が必要であり，この調整のために操作算法の比例定数が大きくなっているとも解釈できる．Fヒープも自己調整型でない．それで，自己調整できるデータ構造により，この方向での良い算法ができることも期待される．（参考文献 2), 29) も参照．なお，最大流問題については，理論的にも実用的にも現在のものより良い算法が作られる可能性が大きい．）

いまのところやや理論的な話としては，作業用領域の少ないソートやサーチの算法が一部の研究者の間で追究されている．これは，"べたづめデータ構造" とでもいうべきもので (implicit data structure など)，n 個の純データ以外には，$O(1)$ あるいは $O(n)$ の作業用領域しか使わないものである．たとえば，整列については，Heapsort や Smoothsort が $O(1)$, Quicksort（の変形版）が $O(\log n)$ である．サーチについては，辞書 (*member*, *insert*, *delete*) が巡回リストのアイデアにより，$O(1)$ 領域で，*member* が $O(\log n)$ 時間，*insert* と *delete* がほぼ $O(2^{\sqrt{2\log n}})$ 時間

まで改良されたが[10),23)]，最近，*member*, *insert*, *delete* が $O(\log^2 n)$ 時間というものが発表されている[24)]．しかし，これは，$O(1)$ 領域にするために，データの並び方でポインタを符号化するということを用いており，定数係数を考慮すると，実用的とはいえないであろう．サーチのうち，順位キューについては，古典的な完全2分木のヒープを用いれば，$O(1)$ 領域で，*findmin* が $O(1)$ 時間，*deletemin*, *insert*, *delete*, *decrease* が $O(\log n)$ 時間で実行できる．これは，(*meld* のない場合．) 実用的にも推奨される方法である[4)]．

ここで3つの見方をあげたが，その他，すでにその重要性が広く認識されている確率的算法などもふくめて，算法の設計・解析に関するある概念の下で高能率であるという算法が今後も追究されていこう．

なお，本稿の守備範囲外であるが，データ構造と組み合わせ論的算法については，並列計算に関する研究がこの数年特に盛んになっていることを付記しておく(たとえばソートについては参考文献3) 参照)．

参 考 文 献

1) Aho, A. V., Hopcroft J. E. and Ullman, J. D.: The Design and Analysis of Computer Algorithms, Addison-Wesley (1974) (訳，アルゴリズムの設計と解析，I，II，サイエンス社)．
2) Bentley, J. L. and McGeoch, C. C.: Amortized Analyses of Self-Organizing Sequential Search Heuristics, CACM, 28, 4, pp. 404-411 (1985)．
3) Bitton, D., Dewitt, D. J., Hsiao, D. K. and Menon, J.: A Taxonomy of Parallel Sorting, ACM Comp. Surveys, 16, 3, pp. 287-318 (1984)．
4) Brown, M. R.: Implementation and Analysis of Binomial Queue Algorithms, SIAM J. Comput., 7, pp. 298-319 (1978)．
5) Dijkstra, E. W.: A Note on Two Problems in Connexion with Graphs, Numer. Math., 1, pp. 269-271 (1959)．
6) Dijkstra, E. W.: Smoothsort, an Alternative for Sorting in Situ, Sci. of Comp. Prog., 1, pp. 223-233 (1982)．
7) Dobosiewicz, W.: Sorting by Distributive Partitioning, Info. Proc. Lett., 7, 1, pp. 1-6 (1978)．
8) Floyd, R. W.: Algorithm 97, Shortest Path, CACM, 5, 6, p. 345 (1962)．
9) Floyd, R. W.: Algorithm 245, Treesort 3, CACM, 7, 12, p. 701 (1964)．
10) Frederickson, G.: Implicit Data Structures for the Dictionary Problem, JACM, 30, 1, pp. 80-94 (1983)．
11) Fredmam, M. and Tarjan, R. E.: Fibonacci Heaps and Its Uses in Improved Network Optimization Algorithms, 25th Annual Symp. of FOCS pp. 338-346 (1984)．
12) 古川康一：マジック・リスト，情報処理，Vol. 12, No. 4, p. 248 (1971)．
13) Imai, H. and Iri, M.: Practical Efficiencies of Existing Shortest-Path Algorithms and a New Bucket Algorithm, J. ORSJ, 27, 1, pp. 43-57 (1984)．
14) 伊理正夫編：ネットワーク構造を有するオペレーションズ・リサーチ問題の電算機処理に関する基礎研究，日本 OR 学会，報文シリーズ，T73-1 (1973)．
15) 伊理正夫，中森真理雄：算法の最近の進歩，電子通信学会誌，58, 4, pp. 433-445 (1975)．
16) Johnson, D. B.: Efficient Algorithms for Shortest Paths in Sparse Networks, JACM, 24, 1, pp. 1-13 (1977)．
17) Klee, V.: Combinatorial Optimization, What is the State of the Art, Math. of Oper, Res., 5, 1, pp. 1-26 (1980)．
18) Knuth, D. E.: The Art of Computer Programming, Vol. 1 (Fundamental Algorithms), Addison-Wesley (1968, 1973 (second edition)), (訳，基本算法，I，II，サイエンス社)．
19) Knuth, D. E.: The Art of Computer Programming, Vol. 3 (Sorting and Searching), Addison-Wesley (1973, 1975 (second printing))．
20) Mehlhorn, K.: Data Structures and Algorithms, 1 (Sorting and Searching), 2 (Graph Algorithms and NP-completeness), 3 (Multi-dimensional Searching and Computational Geometry), Springer-Verlag (1984)．
21) Meijer, H. and Akl, S. G.: The Design and Analysis of a New Hybrid Sorting Algorithm, Info. Proc. Lett., 10, 4-5, pp. 213-218 (1980)．
22) Moore, E. F.: The Shortest Path through a Maze, Proc. International Symp. on the Theory of Switching, Part II (1957)．
23) Munro, J. I. and Suwanda, H.: Implicit Data Structures for Fast Search and Update, JCSS, 21, pp. 236-250 (1980)．
24) Munro, J. I.: An Implicit Data Structure for the Dictionary Problem that Runs in Polylog Time, 25th Annual Symp. of FOCS pp. 369-374 (1984)．
25) Noshita, K.: A Theorem on the Expected Complexity of Dijkstra's Shortest Path Algorithm, J. Algorithms, 6, pp. 400-408 (1985)．
26) Noshita, K. and Nakatani, Y.: On the Nested Heap Structure in Smoothsort, 電子通信学会研究会資料，AL 84-10, pp. 1-10 (1984)．
27) Prim, R. C.: Shortest Connection Networks

and Some Generalizations, BSTJ, 36, pp. 1389-1401 (1957).
28) Sleator, D.D. and Tarjan, R.E.: Self-Adjusting Binary Search Trees, JACM, 32, 3, pp. 652-686 (1985).
29) Sleator, D.D. and Tarjan, R.E.: Amortized Efficiency of List Update and Paging Rules, CACM, 28, 2, pp. 202-208 (1985).
30) Tarjan, R.E.: Data Structures and Network Algorithms, SIAM (1983).
31) Tarjan, R.E.: Amortized Computational Complexity, SIAM J, Alg. Disc. Meth., 6, 2, pp. 306-318 (1985).

(昭和60年9月30日受付)