Vol. 27 No. 2　　情　報　処　理　　Feb. 1986

解　説

手続き型言語におけるデータ構造†

和　田　英　一††

1. はじめに

データ構造[1]のなんであるかはもうかなり常識的になっているとは思うけれども，筆者なりのデータ構造の定義からはじめよう．

処理しようとするデータがさらに子の要素のデータから構成されていると考える場合，要素間，または要素と親のデータ間の位置関係，それらに対する基本的操作やその記法，内部表現や実現法を論ずるのがデータ構造である．たとえばAlgol 68のcomplという型(Algol 68ではmodeという)は図-1のようにreフィールドとimフィールドからなるレコード型のデータ構造で，それぞれのフィールドの型はrealである．このreal型はPascal[2]では基本の型だが，仮数部(mantissa)と指数部(exponent)からできているとも考えられる．そのそれぞれはさらにビットの列にまで分解できる．complはこのように分解できるが，普通のユーザからはひとつのデータ型と見られ，代入や四則演算などの基本操作を施すことができる．integerのような基本型の演算はハードウェアに組み込んであるのが普通である．スタックのような多少複雑なデータ構造と考えられていたものも，最近のハードウェアでは基本的と考えて組み込むことが多くなった．配列の演算もスーパ・コンピュータではかなりハードウェアの助けを借りて行うようになった．ハードウェアに組み込む程には基本的でないが，その次の程度に重要なものはプログラム言語でソフトウェアとして用意する．complなどはまだこのレベルであろう．もっと一般的なものは，ユーザが自分でプログラムすることになる．ここではプログラム言語に組み込まれているデータ構造についてサーベイしてみる．

個々のサーベイに入る前にデータ構造を簡単に分類しておく．

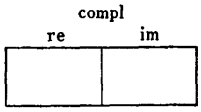

図-1　Compl 型

アクセスの仕方：

ランダムアクセス　どの要素にも同一時間でアクセスできる．

シーケンシャルアクセス　“隣”の関係をたどって次々とアクセス．

要素の総数：

固定　配列，レコード…

可変

一次元の端だけで可変　スタック，キュー

もっと一般に可変　リスト，木構造

内部表現：

リニア　“隣”の関係は記憶場所の関係も確定(別名べた詰め)

リンク　“隣”の関係はポインタにより可変（別名いも蔓）

2. 配　　列

同一の配列要素型の一定数の要素からなる構造．計算機の記憶装置が一次元の配列ということもあってデータ構造の中では最も基本的である．プログラム記憶方式の計算機のプログラムが威力を発揮できたのはプログラムをループにし，一連の記憶領域の作業場所に対して順次に処理が行えたことによる．プログラム言語が開発されると，ほとんど最初ともいえるFortranから配列は用意してあった．配列は他のデータ構造を実現するのにも使われる．

最初に「同一の配列要素型の…」といったのは，FortranやPascalなどで，配列要素の形をしている式の型を，添字の値と独立に知りたいためで，Lispのように型がどうでもよいような言語では別に「同一の

† Data Stvuctuves in Proceolural Programming Languages by Eiiti WADA (Faculty of Engineering, University of Tokyo).

†† 東京大学工学部計数工学科

配列要素型」である必要はない.

要素型がまた配列だと二次元，三次元…の配列が作れるから，一次元の配列だけが基本的である．C[3] や BCPL[4] のように一次元の配列しか宣言できない言語もあるが，これはこれで立派な見識である.

話を一次元に限れば n 個の要素をもつ配列を宣言したとき，下限が1になるもの (Fortran)，0になるもの (BCPL や C)，宣言で自由にきめられるもの (Pascal) などがあるが，筆者はC流の添字0から $n-1$ までの場所がとれるのが好きである．BCPL はちょうど n 個の場所をとるのに vec $n-1$ と宣言しなければならないのが気持ちがわるい.

Pascal や Ada[5),6)] のように添字を列挙型（数えあげ型）にできるようにした言語もある.

```
type dayofweek=(mon, tue, wed, thu, fri, sat, sun);
var i1: array [dayofweek] of integer;
```

と宣言し，i1 [tue] のようにアクセスする．しかし可読性は増すかも知れないがそう本質的とは思えない.

配列型に対する基本的操作はまず配列型変数への代入である．これは他のデータ構造全体についていえるけれども，共有による代入と複写による代入の両方が考えられ，実際にそれぞれの方式の言語が存在する.

Pascal は複写による代入を行う．Pop-2 や BCPL はデータ構造の値は構造へのポインタであって，ポインタを代入するから共有による代入になる．共有によるものは Lisp で (setq dest (copy source)) とやるようにコピィも簡単にできるのが望ましい．代入は手続き呼出しのパラメタ渡しでも起きるが Ada では配列型，レコード型はコピィによってもよくアドレス渡しによってもよいと規定している (Ada 文法書 6.2)．それならプログラマは，処理系がいずれのメカニズムかを知らなければならないかというと，メカニズムに依存するようなプログラムはエラーとするという大変な規定もついている．ずい分昔のことになるが Algol Nにはこの両方の代入があり，:=と←とで区別していたと記憶する.

配列要素へのアクセスの記法にもいろいろある．Fortran は A1(I) のように丸かっこを使う．Pascal や Algol 系では丸かっこは関数呼出しと同じようでいやだと考えて a1[i] のように角かっこを使う．一風変わっているのは BCPL で，a1!i のように書く．つまりここでは a1 の値は配列 (BCPL ではベクタ) a1 へのポインタ，a1[0] のアドレスであり，a1[i] のアドレスはしたがって a1+i で得られる．そこで a1[i] は一回間接アドレスにして !(a1+i) と書くのだが，これはわずらわしい．そこでこう書くかわりに a1!i とするのである．もちろん i!a1 としても OK.

a1(i) とすると関数呼出しのようだから配列のアクセスは a1[i] とすると書いたが，配列のアクセスに関数呼出しを使っているプログラム言語もある．Pop-2[7] がそれで，配列 a1 の第 i 要素を読み出すには a1(i) と書く．一方，その要素に0を代入したければ 0→a1(i) と書く．(Pop-2 では代入される変数は右向き矢印の右に書く.) つまり同じ a1(i) でも式の中に現れたときと，矢印の右に現れたときでは別の関数を呼び出す．もう少しくわしくのべると a1 という一パラメタの関数は，実は双子の関数 (doublet) で selector と updator からなる．式の中で使うと，このうち selector の方を呼び出し，右辺で使うと updator を呼び出す．a1 を双子の関数にするには a1 の参照関数 a1sel と更新用関数 a1upd を用意しておき,

```
a1sel→selector (a1)
a1upd→updator (a1)
```

のようにする．このとき右辺の selector と updator もそれぞれ双子の関数だけれど，ともに updator の方が使われている．a1 の selector はどんな関数だったか知りたくなれば

```
selector (a1)
```

のように selector の selector を使えばよい．この updator と selector を使う関係は BCPL の left-value と right-value の関係[8] に似ている.

```
0→a1(b1(j))
```

と Pop-2 で書くと b1 は selector, a1 は updator が使われるが BCPL でも

```
a1!(b1!j):=0
```

では b1!j は rightvalue で評価し，a1!(b1!j) は leftvalue で評価する.

FranzLisp では a1 という配列をとると，a1 の要素を読み出すのは (a1 i) だが，更新には (store (a1 i) 0) のようにしなければならない．一方 UtiLisp の配列 (UtiLisp ではベクタという) では参照も更新も特にそのための関数があり

```
参照 (vref a1 i)
更新 (vset a1 i 0)
```

と書く.

	AU 平均距離	離心率	° 軌道傾斜	年 公転周期	km 赤道半径	比 重	衛星数
水 星	0.387	0.206	7.0	0.24	2420	5.60	0
金 星	0.723	0.007	3.4	0.61	6100	5.14	0
地 球	1.000	0.017	—	1.00	6380	5.52	1
火 星	1.524	0.093	1.9	1.88	3390	3.97	2
……							

図-2 惑星データベース

3. レコード

レコードは必ずしも同一とは限らない型の一定数のフィールドからなるデータ構造で，各フィールドにはフィールド名をつけ，フィールド名を使ってフィールドにアクセスする．レコードが登場したのは Cobol あたりが早い方かと思う．たとえば Pascal で

```
var z: record re, im: real end
```

とするとzは re, im ふたつの実数型フィールドのレコード型変数となる．また

```
type mo=(jan, feb, mar, …, dec);
date=record y: integer; m: mo; d: 1..31
end; var a: date;
```

とするとaは date 型の変数となり，date は y, m, d のフィールドからなるレコード型であり，y フィールドは整数型，mフィールドは mo型(つまり jan, feb, mar…などの値をとる)，d フィールドは 1..31 の部分範囲型である．aの各フィールドに値を入れるには a.y:=1985; a.m:=oct; a.d:=10 のようにピリオドとフィールド名（ピリオドとフィールド名でセレクタという）でフィールドを示す．しかしaのそれぞれのフィールドにいちいち a. をつけるのは馬鹿らしいというので Pascal には with 文がある．すなわち

```
with a do begin y:=1985; m:=oct; d:=10 end
```

とすればよい．with 文の中の式にさらにレコード型が現れるとしたら with a, b do… のような with 文も書けることになっている．

レコードで忘れてはならないのは a.m のようにセレクタを書いた場合，このフィールド名のところは変数ではいけないということである．つまり a.v のようにして，v の値が y, m, d のどれかになるとしてもコンパイル時には a.v がどのフィールドになるかわからない，ひいては a.v 型がきまらないのである．

レコード型変数の代入にも共有によるもの，複写によるものがある．配列と同様である．Pascal にはないが，レコード型に基本的な操作のひとつは，Cobol の対応転記 (corresponding move) であろう．

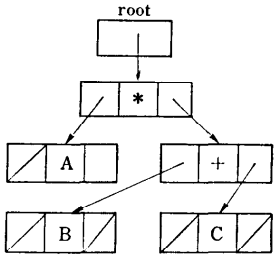

図-3 二進木

レコード型では配列のように添字の式の値を変えながらデータを順になめることはできないが，フィールド名で一挙にアクセスできるから，いろいろとオプティマイズできるのがうれしい．compl の例でいえば，実部と虚部を z[0], z[1] で表すより z.re, z.im の方がわかりやすいのはいうまでもない．

関係データベースはレコードの配列である．

```
var 惑星データベース:
array [(水星, 金星, 地球, …冥王星)] of
record 平均距離, 離心率, 軌道傾斜, 公転周期,
赤道半径, 比重: real;
衛星数: integer end
```

とすれば図-2 のような惑星データベースができる．

最近レコード型が脚光を浴びているのは算体主導型 (object oriented) 言語でのインスタンス変数である．知識ベースでいうフレームもレコード型で実現するのが自然のようである．

しかしレコード型が重宝なのは，これとポインタ型を使ってリンクによる構造を実現するときである．たとえば (A*(B+C)) のような二進木を作るには図-3 のようにする．それにはそれぞれのセルをレコード型に型宣言する．

```
type link=↑cell;
     cell=record left: link; node: alfa;
     right: link end;
```

```
var root : link ;
```

と宣言したとすれば，図-3 の構造なら

```
new (root); {*の入っている cell をとり，そこへのポインタを root に入れる．root は link 型だし，link は cell 型へのポインタだから cell 型の変数をとればよいことがわかる}
root↑.node:='*'
new (root↑.left);
root↑.left↑.node:='A';
root↑.left↑.left:=nil;
root↑.left↑.right:=nil;
```

このようにして次々と図-3 のデータ構造ができる．

上の例で，変数 root はちゃんと変数宣言で名前つきで場所がとられるいわゆる静的変数である．この宣言のあるブロックに制御が移ってくると自動的にスタック上に変数領域をとる．静的変数は宣言のときに大きさを決めておかなければならないから，もしこの機能しかなければ，いつでも十分に大き目の場所をとっておくことになるのだが，それはそれでまた不経済である．そこで Pascal では，スタックの他にヒープ領域が用意してあって，標準手続き new でその領域に動的変数がとれるようになっている．動的変数には名前はなく，静的変数からポインタをたぐってアクセスしなければならない．上の例の cell 型はすべて動的変数である．動的変数が使えるようになるとデータ構造の実現はきわめて自由になる．

レコードには可変部 (variant) といって，構造の後方を異なるフィールド構成にできたりもするが，その話は省略しよう．

4. セットとビット演算

セットのデータ構造をもっているプログラム言語は筆者は Pascal 以外にほとんど知らない．言語の名前から見て SETL はセットだらけかと思うがよくわからない．理研の Flats マシンの最後の s もはじめはセットの s だといっていた．

Pascal のセットは要するにビット演算である．話の一方の端からの各ビットに基底型 (Base type) のとる範囲の値を順に対応させてゆく．たとえば基底型が mo, つまり (jan, feb, mar, …, dec) だと set of mo のセット型の語は，第 0 ビットが jan, 第 1 ビットが feb, …第 11 ビットが dec に対応している．セットの表記 [jan, jun, jul, aug] は第 0, 5, 6, 7 ビットだけが 1，あとが 0 になっている語である．ふたつのセットの和はビットごとの or をとればよい．ある要素がセット中にあるかどうかを見るにはビットテストをやればよい．要するにせっかくハードウェアにビット演算がいろいろ用意してあるのに，高水準言語ではそれをあまり利用しないから，なんとか使おうと考えだしたのがセットだと思われる．

しかしセット型のなるほどと思う例はなかなか見当たらない．エラトステネスの篩なんか，正にこれかと思うけれども，ビット演算はレジスタで行うから，ハードウェアのレジスタ長までのセットしか扱えない．Pascal の CDC 版コンパイラは，入力される文字の上のセットを用意し，英字のところだけ 1 になっているようなセット型定数をもっており，入ってきた文字が英字かどうかのテストをやっていた．これは CDC のレジスタが 60 ビットもあったからできたのであった．

筆者もセット型を使ったことは皆無ではないが，そう魅力あるデータ構造とは思わない．他の人もそう思うのか，Pascal の後継言語のような顔をしている Ada からもセットは消えてしまった．

セットというのは前述のようにビットに名前がつくというところに最大の特徴があった．ビットに名前をつけずにビット演算のできる言語は BCPL や C など実存する．C でいまだに混乱するのは & と && の使い分けだがあれはなんとかならないだろうか．(BCPL の logand, logor というオペレータも気持ちが悪いがこれは BCPL では and を宣言のつなぎに，or をテストに使ってしまったからである)

1 ビットごとのビットセット，ビットテストなら UtiLisp にも備わっている．筆者は Macintosh にインプリメントした UtiLisp を使い，ビット操作の機能を用いてエラトステネスの篩のプログラムを書いた．そして篩がちょうどビットマップディスプレイにうつるように作業場所を用意し，合成数が次々と篩い落ちてゆくのを眺めて楽しんだことがある．

5. リ ス ト

Lisp の話がでたら次はリストということになる．リストは，なにものかの一次元の並びである．「リスト」は和訳のとき「並び」にすることが多いから，この定義は訳みたいでもあるが，一次元の配列と根本的に違う点はリストの要素は添字と直接には対応していないということである．リストは

```
(a b c d)
```

のように書く．a が 0 番の要素だとすると，b はその

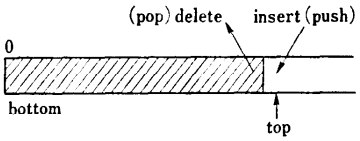

図-4 リニアアロケーション スタック

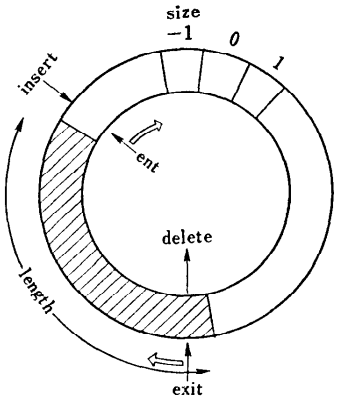

図-5 リニアアロケーション キュー

直後の要素だから1番だけれども，aとbの間に別の要素を挿入すると，bはたちまち2番になってしまう．

リストはこのように各要素の位置が固定していないから並びの両端または途中で要素の挿入削除が自由にできる．

リストは両端がまったく対等で，両方とも頭と思えるものや，一方が頭，他方が尾で機能が非対称のものがある．Lispのリストは非対称のものの例である．頭の方だけで挿入削除するのはスタック，尾の先に挿入し頭から削除するのがキューなことはいうまでもあるまい．

挿入削除が簡単なためには二進木でも使ったリンクによるのがよいが，工夫次第ではリニアアロケーションでももちろんうまくゆく．リンクにするとリンクを置く場所が馬鹿にならないので，ミニコン，マイコンではリニアでできればそうしたいのが人情である．特に両端だけで挿入削除するならリニアで十分な場合が多い．現実にスタックとキューは図-4,図-5のように

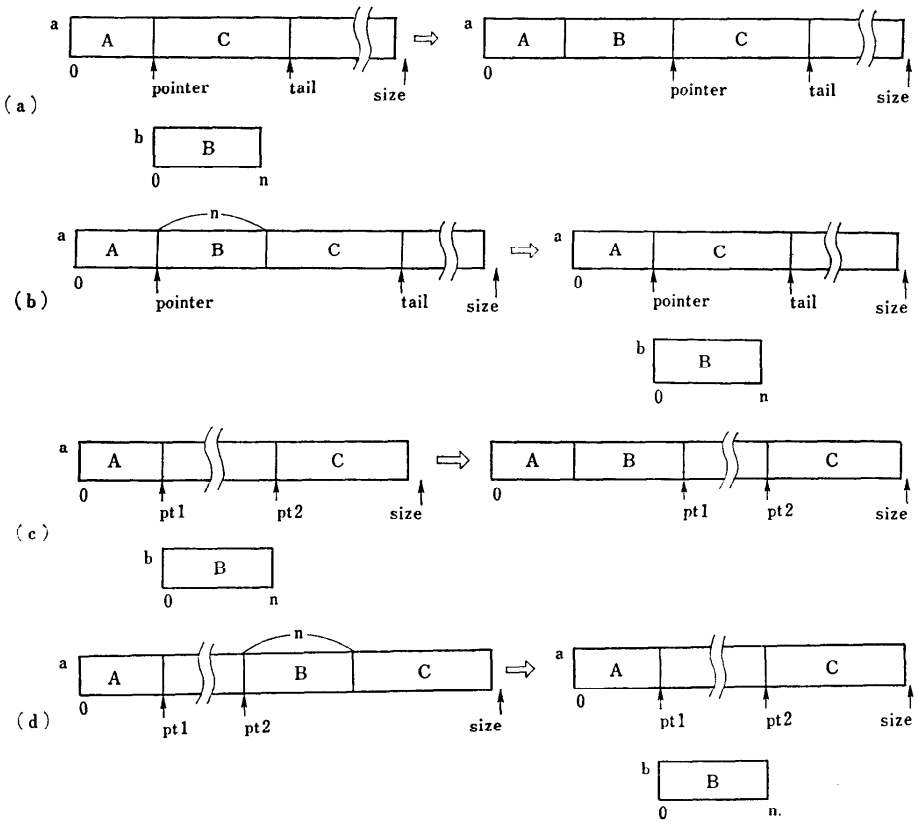

図-6 リニアアロケーション 途中の挿入削除

作る．スタックの方は半無限のバッファはとれないからスタックがオーバフローしないようなチェックが必要だが，これを毎回やるのは結構負担が多い．キューの方はサーキュラバッファというけれど，実際にこういう丸いメモリは存在しないからポインタの ent と exit が size-1 から0へ戻るチェックをやらなければならない．また空のキューから取り出そうとか，満杯のキューに追加しようとかするのも困るから，そのチェックもしなければならない．ent=exit では空か満杯かいずれかだということしかわからないので，ent を進めて exit に一致すれば full，exit を進めて ent に一致すれば empty とフラグをセットしたり，挿入削除のたびにポインタを進める他に length を調節し，length=0 が empty のこと，length=size なら full のこととしたりする．

もっと一般的にリストの途中に挿入削除がある場合でもベタ詰めでインプリメントすることがある．途中に操作するときは，その場所を示すポインタが要る．**図-6**(a)はベタ詰めのバッファ a の pointer のところにBを挿入，(b)は pointer のところから長さ n のBを削除する場合の絵である．プログラムは示さないが奇麗に対称にかける．

途中に挿入削除するリストをベタ詰めにインプリメントするのに，ポインタのところでデータを分離する方法もよく使われる[9]．図-6 の(c)と(d)にその概略を示す．ポインタをポインタより前の部分の最後を示す pt1 とポインタより後の部分の先頭を示す pt2 とにし，その間があき地である．

(a)(b)の流儀と(c)(d)の流儀はどちらがよいかといえば，それは応用による．ポインタのあげさげの方がデータの挿入削除より頻繁に行われるなら(a)(b)の方が楽である．(a)(b)の方は挿入削除で大量にデータが移動する．しかし挿入削除を絶えずやるようなエディタを作ろうとすれば(c)(d)型の方がよいであろう．この方はポインタのあげさげで大量のデータが移動する．

リニアアローケーションはこのくらいにして，次は

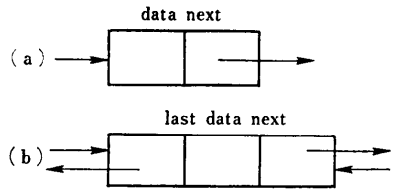

図-7　セルの形

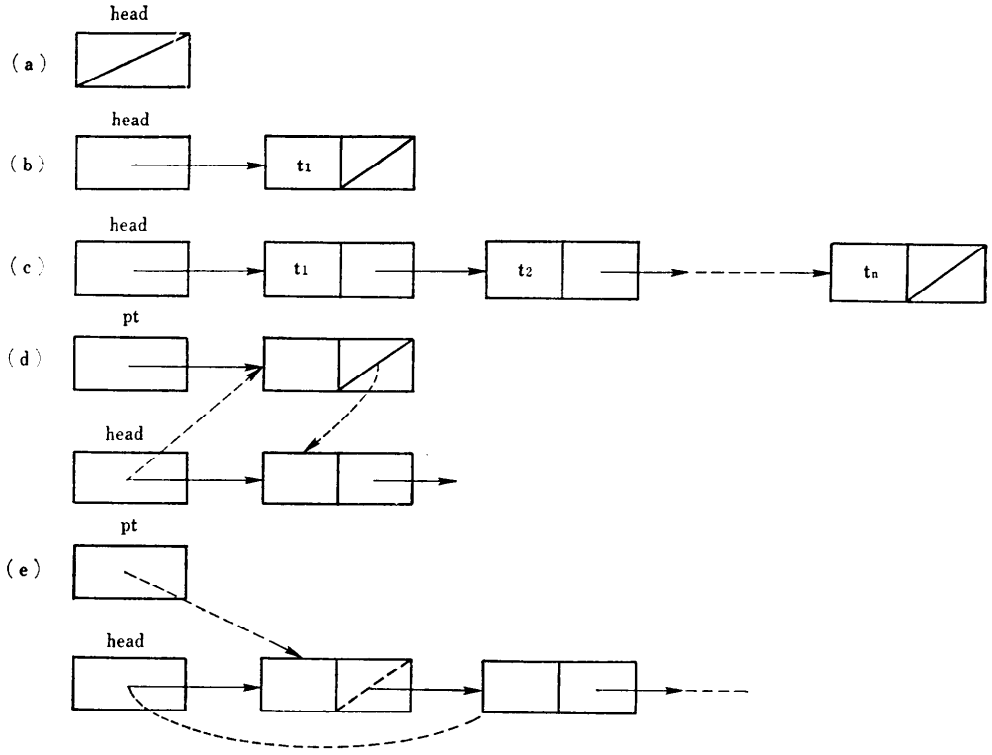

図-8

リンクによるアローケーションを復習してみる．図-7はレコードのところで二進木を表すのに使ったような(一次元リスト用の)セルである．(a)は一方向にだけ辿れるもの，(b)は両端のいずれからも辿れるものだが，しばらくは(a)だけで話を進める．データ型の定義は念のために書くと

```
type link=↑cell;
     cell=record data: T; next: link end;
```

でTはこのリストの要素の型のつもりである．リストを操作する元になる名前つきの静的変数を head としよう．var head: link;

リストが空，つまり要素がひとつもなければ図-8(a)のように head が nil になる．ひとつでも要素があれば，(b)(c)のように head にはリストの先頭の要素のセルへのポインタが入る．各セルの next フィールドには次の要素のセルへのポインタが入り，最後のセルの next フィールドは nil になって，もうこれ以上要素がないことを示す．Lisp に倣い，nil は斜線で示す．

先頭に新しい要素を挿入する insert to head は図-8(d)のようにする．すなわち pt のさすセルを head のさすセルの前へ入れるには

```
pt↑.next:=head; head:=pt
```

を行う．結果は図の破線のようになる．

先頭の要素を削除する delete from head は図-8(e)のようにする．すなわち head が nil でなかったら head のさしているセルを pt へつなぎ，head はそのセルの next と同じものをさすようにする．head が nil なら削除はできない．

```
if head=nil then error
else begin pt:=head; head:=head↑.next;
pt↑.next:=nil end
```

を行う．破線は結果の形を示す．

insert to head と delete from head があればスタックが実現できる．しかしキューを実現するには尾の方の挿入か削除が必要だが，今のままの形のリストでは毎回 head から最後のセルを探しにゆかなければならず，これは不便である．そこで尾の先をさしているもうひとつのポインタ tail を用意する(図-9)．この図の(b)のように tail はリストの最後のセルを探す．したがって空リストの場合は(a)のように head と同時に tail も nil になっている．insert to tail は図-9(d)のようにする．つまり tail のさしているセルの next を pt と同じものを探すようにし，tail を更新する．プログラムで書けば

```
if head=nil then head:=pt
else tail↑.next:=pt; tail:=pt
```

でよい．head が nil なら tail も nil で tail↑.next はとれないから，そのときは head を pt にする．

尾の方からひとつずつ削除するのは tail ポインタがあっても困難である．それは尾から先頭へ向かうリンクがないことによる．そこでキューは尾の方に挿入，頭から削除という組で実現する．

Prolog の方では差分リスト[10]がよく使われる．これもリストを head と tail のふたつのポインタで表すが，tail は最後のセルをさすのではなく，最後のセ

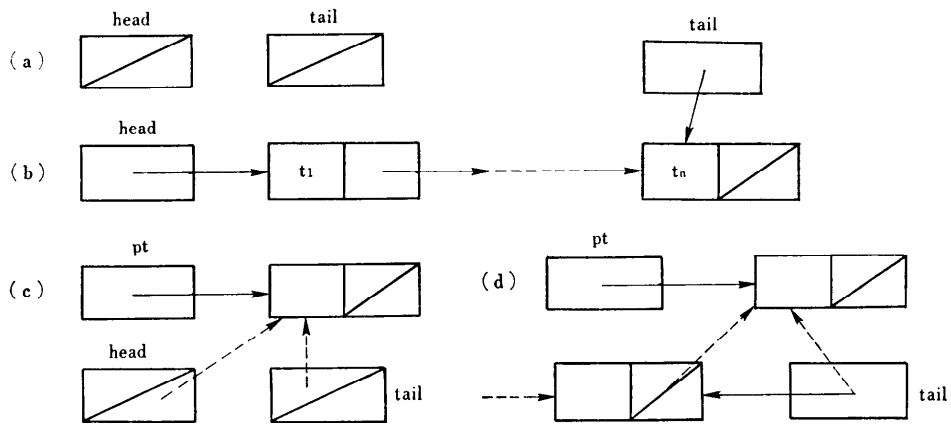

図-9

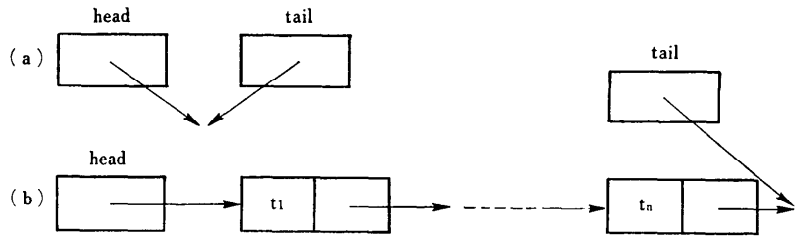

図-10 差分リスト

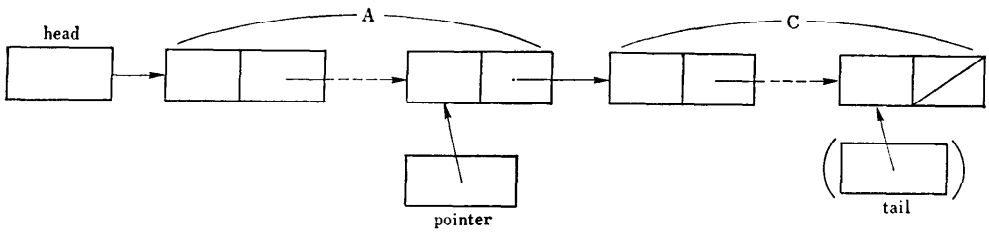

図-11

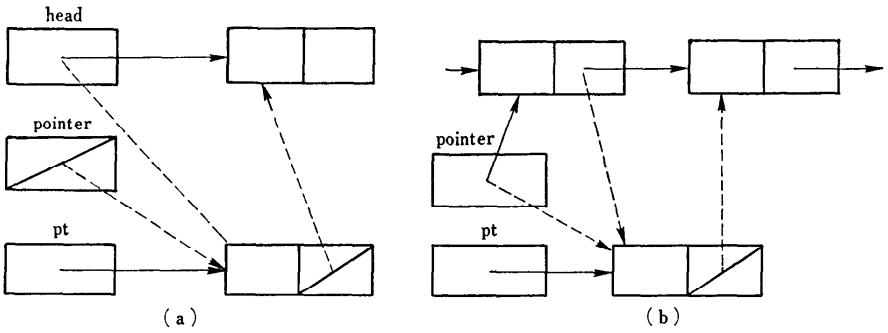

図-12

ルの next フィールドと同じものをさす（図-10）．この方式ではリストの最後の next フィールドは nil ではなく，どこかのアドレスでもよい．とにかくリンクの値が tail と同じなら nil だったと考える．差分リストを使うと，長いリストの一部を別に切りださずに通常のリストとして扱うことができるが，欠点はどのリストにも tail ポインタが必要になること，最後の検出が nil が入っているのより多少面倒になることである．

リンクによるリストの途中での挿入削除も操作すべき点を示すポインタ（ユーザが指定するためのもの）—pointer という名のポインタを用意しておけば，そうむずかしくはない（図-11）．この図のリストは pointer のところでAとCのふたつの部分に分かれている．実際には pointer は（tail のように）Aのリストの最後のセルをさしているが，これはAとCの間になにか挿入したり，pointer のところ，つまりCの先頭を削除したりするとAの最後のセルの next フィールドを処理しなければならないからである．そうだとするとAの部分 pointer より前が空リストになったとき pointer はどこをさすか．head はセルでないから head をさすことはできない．そこで pointer を nil にする．一方Cの部分が空リストなら pointer は tail と同じにする．

pointer のところに pt のさしているセルを挿入するには図-12 のようにやればよい．

```
if pointer=nil then
begin pt↑.next:=head; head:=pt end
else begin pt↑.next:=pointer↑.next;
pointer↑.next:=pt end; pointer:=pt
```

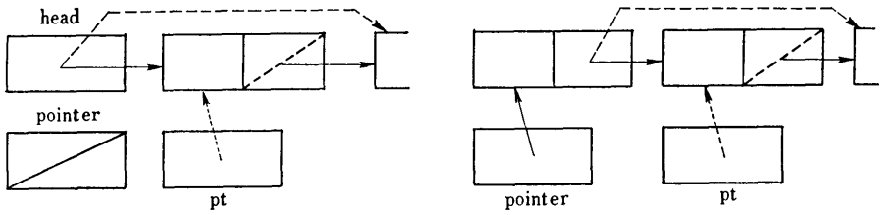

図-13

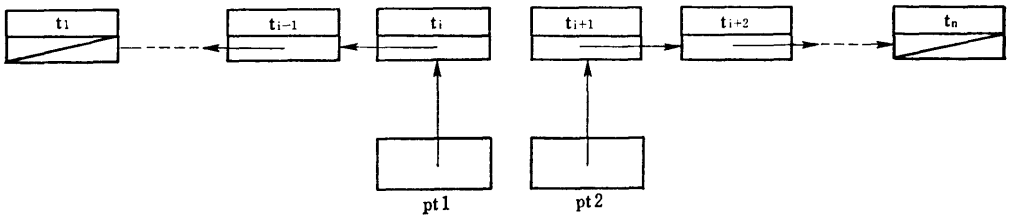

図-14

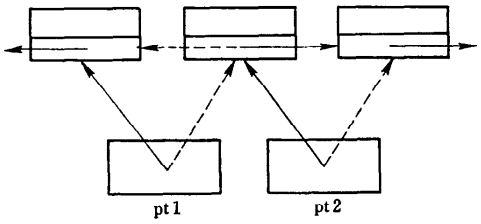

図-15

削除の方は図-13 の通り.

```
if pointer=nil then
if head=nil then error
else begin pt:=head ; head:=pt↑. next end
else if pointer↑. next=nil then error
else begin pt=pointer↑. next ;
pointer↑. next:=pt↑. next end ; pt↑. next:=nil
```

リスト処理の話が長びいているが，最後にもうひとつ図-14 をご覧いただきたい．これはリンクによるリストだが，図-11 がべた詰めの図-6 の(a)に似ているとすれば図-14 は図-6 の(c)に似ているものである．つまり pointer のところでリンクを両端に向けて切りはなしている．pointer も図-6(c)のように pt1, pt2 としてある．図-11 のようなリストでは pointer は head の方から反対側に向けてしか動かなかったのに対し，図-14 のは pointer はどちらへでも動いてゆく．ただ pointer の移動は一方向だけだけれども図-11 のものの方が楽である．図-14 のものでは pointer を右へひとつずらすだけでも図-15 に示すように

```
t:=pt2 ; pt 2:=pt2↑. next ; pt2↑. next:=pt1 ;
pt1:=t
```

のようにポインタのつなぎかえをしなければならない．このうち t は作業用の場所であって，要するに三つのポインタが pt1→pt2↑. next→pt2→pt1 のように一斉におきかわったのである．ポインタの操作にはこのように三つを回転させることが多いので，数年前の Comm ACM に鈴木則久君が提案していたような rotate (p1, p2, p 3) というオペレーション[11]があるとプログラムも読みやすくなり，インプリメンテーションの能率もあがるかもしれない．

図-11 のような一方向きのリンクでバックもしようと思えば，どこからきたかをスタックに入れておくということになるだろうが，ポインタを逆転すればスタックは不要になる．これを二進木のトラバースに用いたのが Deutsch, Show, Waite のポインタ逆転のトラバースだが，これは二進木の話題だし，ますますリストが長びくので割愛しよう．(Kunth の第一巻[12] p. 417)

これまでのリストの話は言語に備わっているリスト用の機能というより，レコードとポインタ型，あるいは配列を使ってインプリメントすることであった．プログラム言語にリストが備わっているのはなんといっても Lisp, それと Pop-2 であろう．Lisp については基本的には insert to head の cons, delete from head の car と cdr だけがあるようだが，リストを辿る nthcdr とか，リストの外科手術をする rplaca, rplacd もあって実に自由に操作できる．リストの要素をひと

つずつ評価するには (list a b c) のように；どれも評価しないためには (quote (a b c)) または，(a b c) とするのも常識であろう．しかし評価するのとしないのと混っていたらどうするか．最近の Lisp にはバッククォート (`) のマクロがあって `(a ,b c) のように書く．するとクォート (') と同じようにふつうには評価しないが，カンマ (,) のつぎの要素だけは評価するという約束になっていて便利である．Pop-2 のリストは [a, b, c] だが，これは要素を評価しない．評価したいときは装飾リスト (decorated list) にして [% a, b, c %] のように書く．

プログラム言語に備わっているリスト処理ではないが，Lisp の構造エディタもリスト処理では忘れることができない．ちょっとだけそのことにも触れておきたい．Lisp の構造エディタ[13] は InterLisp が元祖かと思うが，今では FranzLisp や UtiLisp や多くの Lisp に用意してある．リストの要素に左から順に番号をふる．例によって筆者は 0 からふるのが好きだから

(a b c d e f)

のようなリストでは a が 0 番，f が 5 番である．これらの要素はもちろんまたリストでよい．今このリストを対象に編集していて，b のリストの編集へ進みたければ，b は 1 番から 1 というコマンドをだす．反対に親のリストへ戻りたければ (InterLisp などでは 0 とやるのだが，ここでは 0 は使えないから up のつもりで) コマンド u をだす．こういうふうにしてリストの中に入ったり外へ出たりする．リストの挿入削除にもいろいろあるが，(n) とやるとそのときの n 番の要素を削除する．(n s_1 $s_2 \cdots s_k$) とすると n 番の要素の前へ s_1 $s_2 \cdots s_k$ を挿入するというのが筆者のもっている超簡単な構造エディタである．単一操作で置きかえはできない；一度削除してから挿入することになっている．(a b c d e f) というリストがあったとき，これを (a (b c d) e f) のように形をかえたいということもよくある．これは 1 番から 3 番までの両側にカッコを挿入するというので (bi 1 3) (bi は both in の意) というコマンドを使う．反対に (a (b c d) e f) を一列にするには第 1 番の要素の両側のカッコをとるというので (bo 1) (bo は both out の意) を使う．これだけのコマンドのエディタがあれば，最低の仕事はできる．

動的リストのことで終りにしよう．これは Pop-2 にある機能である．普通はリストの要素の値同士のあいだにはあまり関係がないから，これまで述べてきたような静的リスト，つまりリストを記憶装置にそのまま入れておくという形をとる．しかし要素のあいだに規則があれば，その規則を表す関数の形でリストを憶えておくことができる．これが動的リストである．たとえば 0, 1, 2, … というリストは，たとえ無限に長くても add1 という関数で表すことができる．これを Pop-2 では次のように fntolist という関数を動的リストに変換する関数を使って実現している．

```
vars n ; −1 → n ;
function suc ; n+1 →n ; n end ;
fntolist (suc) → y ;
element (y, 20) ⇒
**19
```

最初に n を変数宣言し，初期値を −1 にする．次に suc を関数宣言する．これは n に 1 をたし，その n を返すものである．最後にこれを fntolist して動的リストを y とする．element (y, 20) で y の 20 番目の要素をとると 19 が得られるという仕掛けである．日本ではあまり知られなかったが，Pop-2 はプログラム言語としては結構面白いものであった．

6. おわりに

ずい分リスト偏重の解説になってしまった．二進木についても書くことは同じくらいあったのだが，あとはすべて省略しなければならない．

まだこの他にバッグとかシーケンスとかストリームとか奇抜なデータ構造があるが，それらもマイナーということで割愛せざるを得ない．バッグというのはセットのように構造に入れた順には関係なく，入っているかいないかだけが問題なのだが，2 回入れたものは 2 回入ったということも覚えていて，2 回とり出さないとなくならないものである．シーケンスはリストと似ているが FP のような関数プログラムによく出てくるもので，オペランドや関数の列である．オペランドのすべての要素にある関数を mapcar してみたり，ある述語で filter したり，また反対にオペランドに関数の列をザーっと作用させたり，これも結構面白い．ストリームというのは入出力がかっていて，動的な文字リストという気分で使うもの．筑波大学の Stella とか，UtiLisp のストリームとか OS 6 のストリーム関数とか，各所に神出鬼没しているが，これにももう鬼没してもらうことにしたい．

参 考 文 献

1) Hoare, C. H. A.: データ構造化序論(野下他訳: 構造化プログラミングの第2章) サイエンス社 (1975).
2) 石畑清他: Pascal の標準化——ISO 規格全訳とその解説——, bit 別冊 (1984年9月).
3) B. M. カーニハン, D. W. リッチー (石田晴久訳): プログラミング言語C, 共立出版 (1981).
4) Richards, M. and Whitby-stevens, C.: BCPL-the Language and its Compiler, Cambridge University Press (1979).
5) 米田信夫編: プログラム言語 Ada, bit 臨時増刊, 1981年10月.
6) 情報処理振興事業協会編: 最新 Ada 基準文法書, bit 別冊 (1984年8月).
7) Burstall, R. M., Collins, J. S. and Popplestone, R. J.: Programming in Pop-2, Edinburgh University Press (1971).
8) System-5: 右の値と左の値, 数学セミナー (1983年8月).
9) 和田英一: マイクロエディタ, bit 臨時増刊 (1978年2月).
10) 上田和紀: リストと差分リスト, bit (1983年5月).
11) Suzuki, N.: Analysis of Pointer "Rotation", Comm ACM, Vol. 25, No. 5, pp. 330-335 (May 1982).
12) Knuth, D. E.: The Art of Computer Programming, Vol. 1, Addison Wesley.
13) System-5: Lisp 礼讃, 数学セミナー (1985年4月).

(昭和60年11月26日受付)