3　リスト (List) 構造の設計と実装 (続き)

課題 3 (10/21 出題). リストを用いて、次々と入力した整数値を大きさの順に並べ替えて表示するプログラム list2.c を作成せよ。

- 入力した整数を値とする新節点を、リストの適切な場所に挿入し、作成したリストの各節点の値が大きさの順になるようにする。
- 0 が入力されたら終了。(0 を値とする節点は作らない。)
- 先頭からリストを辿って全ての値を順に表示する。

解答例 3.1. 新節点を挿入する関数 insert_Node() と main() とだけ示す。

```
/*   list2.c   2008-10-21   */
int main( int argc, char **argv )
{
    int a;
    Node head, *target;

    head.next = NULL;
    while( scanf("%d",&a), a != 0 )
    {
        target = &head;
        while( target->next != NULL && target->next->val < a)
        {
            target = target->next;
        }
        insert_Node(target,a);
    }
    print_list(&head);

    return 0;
}

void insert_Node( Node *p , int a )
{
    Node *t;

    t = p->next;
    p->next = (Node *)malloc(sizeof(Node));
    p->next->val = a;
    p->next->next = t;

    return;
}
```

この問題の意外に厄介な所は、新節点を挿入する場所を決める為の while loop の終了条件が

- 入力値より大きな値に辿り着く
- 末端 (NULL) まで到達する

の 2 通りあること。
NULL を辿ってしまうとエラーになるので、辿るのをやめる判定条件は
「NULL か、さもなくば次の値が大きい」
であるが、while 文の中に書く条件式は辿り続ける条件なので、この否定で
「NULL でなくて、かつ次の値が小さい」
となる。ここで && を使ってみた。論理積演算子 && は、
「左から順に評価し、真の時のみ右の式を評価する」
と決まっているので、この書き方でも NULL を辿ることはない。

while の条件式を末端判定だけにして、入力値より大きな値に辿り着いたら break で loop から出る、という手もある。

演習 7. 種々の便利の為に、「新節点の領域を確保し、要素の値を設定して、次を NULL に初期化して、その節点へのポインタを返す」関数 new_Node() を作っておく、という考え方もある。これを実装せよ。

演習 8. 入力値に制限がある場合 (例えば $0 \leq a \leq 100$) には、「あり得ないほど大きい値」(例えば 101) を要素とする節点を、末尾に予めダミーで作っておく、という手もある。(こうすると while loop の終了判定は「入力値より大きな値に辿り着く」だけで良い。)

このような要素は**番兵 (sentinel)** と呼ばれる。

演習 9. 実際には入力値を順次受け取りながら整列したリストを直接作るよりも、既にリストなどの形で保持しているデータに対して、それを整列したリストを作るという方が汎用性があろう。(その前の処理結果について適用することが普通だろうし、入力値も一旦受け取って事前のエラー処理などをする方が良かろうから。) そこで、必ずしも整列していないデータのリストを引数に取って、それを整列したリストを作る関数 sort_list() を作成してみよう。

プロトタイプの設計には、何通りかの考え方があり得るだろう。

さて、この方法で n 個のデータを整列させるのに、どの程度の手間が掛かるだろうか。新節点を作る回数はデータ数と同じ n 回で固定なので、問題は挿入する場所を決める為の比較回数である。最も都合の良い場合は、データが逆順で来て常に先頭に挿入される場合で、それなら毎回先頭より小さいことを確かめるだけなので n 回で済む。しかし、理論上重要なのは「最悪でもどの程度で済むか (どの程度で必ず出来るか)」であり、実際上重要なのは「平均してどの程度で済むか」「充分高い確率でどの程度で済むか」である。このような量を**計算量 (complexity)** と呼ぶ。

最悪なのは毎回先頭から末尾まで辿る場合で、k 番目のデータに対して $(k-1)$ 回の比較が必要なので、$\sum_{k=1}^{n}(k-1) = \frac{n(n-1)}{2}$ 回となる。平均的には毎回半分くらいと見てこの半分の $\frac{n(n-1)}{4}$ 回となる。計算量は **Landau の O 記号**で表すことが多い。定数倍の違いは計算機の処理速度の違いに吸収されるからである。すると、このアルゴリズムの計算量は $O(n^2)$ と表される。

$$g(n) = O(f(n))$$
$$\overset{\leftarrow}{\Longleftrightarrow} \frac{g(n)}{f(n)} \text{が有界}$$
$$\Longleftrightarrow \exists C : |g(n)| \leq C|f(n)|$$

上記は慣習だが、馴染めない人は、
$O(f) := \{g|\exists C : |g(n)| \leq C|f(n)|\}$
だと思って、
$g \in O(f)$
と読めば良いだろう。

考察 3.0.3. 木を用いた整列法の計算量は如何程か。最も都合の良い場合・最悪の場合・平均を考えよ。

4　木 (tree) 構造 (続き)

4–5　演算木 (式木)

内部構造として本質的に木構造から成るデータの例として、演算子と被演算子とから成る数式がある。

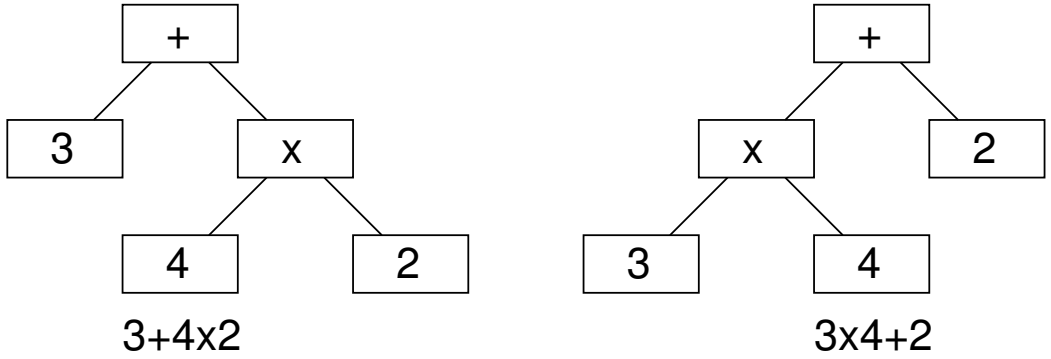

図 6: 演算木の概念図

次回までの課題は、前回出題の課題 4 (p.21) である。

- 「箱と矢印との図式」
  を描いて動作を追ってみること
- “`printf` debug” により、
  何処までは確実に動作し、
  何処で止まっているか、
  を見極めること

などを心掛けよ。

数式を理解するとは、まづ第一に、この内部的な木構造を把握することである。数学としての意味や意義はその後の話。

実習 4.2. Mathematica・Maple などの数式処理 (計算機代数) ソフトウェアの多くは、内部的に数式の木構造を保持し、それに基づいて処理している。色々な数式に対して、どのような形で保持しているか調べよ。

数式 (演算式) は本来このような木構造を成しているものであって、通常その木構造に対する走査順に、便宜的に一列に表記しているに過ぎない。左の被演算子は右の被演算子より左に書くことにすると、走査順は次の 3 通りがある。表記法が異なれば表記は変わるが、同じ実体を表していることを理解しよう。

- 中→左→右の順: 演算子が被演算子の前… **前置記法 (prefix)・ポーランド記法**
- 左→中→右の順: 演算子が被演算子の間… **中置記法 (infix)**
- 左→右→中の順: 演算子が被演算子の後… **後置記法 (postfix)・逆ポーランド記法**

ポーランドの論理学者 Łukaszewicz に因る。

小学校以来通常見慣れている記法は中置記法であるが、この記法では演算子の優先順位を括弧で指定する必要がある。しかし、前置記法・後置記法では、(各演算子が取る被演算子の個数が定められていれば) 括弧不要という著しい性質を持つ。前置記法は順番としては数学の関数記号と同じである。又、後置記法は実は日本語の本来的な語順と一致している。

中置	前置	後置	日本語
$3+4*2$	$+\ 3\ *\ 4\ 2$	$3\ 4\ 2\ *\ +$	3 に 4 に 2 を掛けたものを足したもの
$(3+4)*2$	$*\ +\ 3\ 4\ 2$	$3\ 4\ +\ 2\ *$	3 に 4 を足したものに 2 を掛けたもの
$3*4+2$	$+\ *\ 3\ 4\ 2$	$3\ 4\ *\ 2\ +$	3 に 4 を掛けたものに 2 を足したもの

後置記法は、特に、スタックを用いて式の値を計算するのに便利である。次の明解な手順で計算することが出来る。簡単の為、演算子は全て二項演算子とする。

- 数値が来たらプッシュする。
- 演算子が来たら 2 数をポップして演算を施し、その結果をプッシュする。
- 式が終わったらポップし、それでスタックが丁度空になったらその値が答え。

途中でポップする数値がなかったり、最後のポップ後にまだ数値が残っていたら文法エラー。

例えば、ページ記述言語の PostScript は後置記法による言語であり、スタックを用いて処理している。

演習 10. (提出推奨) 後置記法 (逆ポーランド記法) の式に対しスタックを用いて値を計算する、上記のアルゴリズムを実装せよ。 演算子は二項演算子 `+`, `-`, `x`, `/` とし、入力方法については、

- コマンドライン引数として空白で区切って入力し、各項を文字列 `argv[i]` として受取る。(初級)
- 実行中に文字列として一行を一括して受取り、読取りながら各項に分割 (トークン分割) する。(上級)

`*` はシェルのメタキャラクタ (ワイルドカード) であるので、掛け算記号は `x` で代用せよ。

などのいづれかの方法を取り、入力数値としては、

- 一桁の自然数 (0〜9) に限定。(初級)
- 自然数に限定、二桁以上も可。(中級)
- 実数に対応し、小数点付きの数値も受付ける。(上級)

など適当な仕様を選べ。

「逆ポーランド電卓」の実装では、読んだ文字が演算子か数字か、演算子ならどの演算子か、で場合分けをすることになる。勿論 `if 〜 else` の繰返しで書いても良いのだが、このように或る式の値によって多岐に場合分けを行なう場合には、`switch` 文を用いるのが便利かつ明快である。