4　木 (tree) 構造 (続き)

課題 5 (10/28 出題). 二分木を用いて、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 tree.c を作成せよ。

- 木のどの節点についても次の条件が成り立つように、入力した整数を値とする新節点を、木の適切な先端に追加する。
  - ⋆ その左にある部分木のどの要素の値も、その節点の要素の値より小さい。
  - ⋆ その右にある部分木のどの要素の値も、その節点の要素の値より大きい。
- 0 が入力されたら終了。(0 を値とする節点は作らない。)
- 先頭から木を辿って全ての値を順に表示する。

注 . ありがちな誤答例として次を挙げる。節点を追加する (つもりなのに出来ていない) 関数 xappend() のみ示す。型定義など、全体については後の解答例を参照されたい。

```
void xappend( Tree p, int a )          /* wrong!! */
{
    while( p != NULL )
    {
        if ( a < p->val )
        {
            p = p->left;
        }
        else
        {
            p = p->right;
        }
    }
    p = new_Node(a);

    return;
}
```

型定義は、

```
typedef struct node {
    int val;
    struct node *left;
    struct node *right;
} *Tree;
```

ポインタを操作しているので勘違いし易いが、要は、例えば p->left の先に新しい節点を追加したい (即ち、値が NULL だった変数 p->left の値を書換えたい) のに、別のポインタ変数 t を用いて、
t=p->left; t=New_Node(a);
とやってるのと同じで、これなら変数 p->left の値が書換わっていないのは一目瞭然。

解答例 5.1. ちょっとすっきりしないが、関数 append() をこう書けば、前頁の注で示した誤りは回避できる。木を表示する関数 print_tree() の工夫については別に述べる。

```
/*   tree0.c   2008-10-28   */
#include <stdio.h>
#include <stdlib.h>

typedef struct node {
    int val;
    struct node *left;
    struct node *right;
} *Tree;

Tree new_Node( int );
void append( Tree, int );
void print_tree( Tree );

int main( int argc, char **argv )
{
    int a;
    Tree root, target;

    root = new_Node(0);        /* dummy node */
    while( scanf("%d",&a), a != 0 )
    {
        append(root,a);
    }
    print_tree(root);

    return 0;
}

Tree new_Node( int a )
{
    Tree t;

    t = (Tree)malloc(sizeof(struct node));
    t->val = a;
    t->left = NULL;
    t->right = NULL;

    return t;
}
```

```
void append( Tree p, int a )
{
    while( p != NULL )
    {
        if ( a < p->val )
        {
            if ( p->left == NULL )
            {
                p->left = new_Node(a);
                return;
            }
            else
            {
                p = p->left;
            }
        }
        else
        {
            if ( p->right == NULL )
            {
                p->right = new_Node(a);
                return;
            }
            else
            {
                p = p->right;
            }
        }
    }
}
```

勿論、関数 print_tree() の本体がここに必要

解答例 5.2. 木の走査順に単に表示するだけでは、表示結果から木構造が読みとれないので、木構造が幾らか判るように工夫してみた。また、基本的には再帰呼出だが改行は最後の一回で良い、などのこともあるので、二段構えにしてみた。

```
void print_tree0( Tree p )
{
    printf(" [");
    if ( p != NULL )
    {
        print_tree0(p->left);
        printf("%d", p->val);
        print_tree0(p->right);
    }
    printf("] ");

    return;
}

void print_tree( Tree p )  /* print except the dummy node, and then ".\n" */
{
    print_tree0(p->right);
    printf(".\n");

    return;
}
```

バグ取りの段階では、これ位やっておくのも有効だろう。

この場合、外 (ユーザ) から見えるのは `print_tree()` だけで良い。

実行例 . `main()` 内の `print_tree(root);` を `while` loop 内に移すと、木が育っていく様子が観察できる。

```
=> cat infile
123 45 678 90 12 345 6789 0
=> ./tree0 < infile
 [ [] 123 [] ] .
 [ [ [] 45 [] ] 123 [] ] .
 [ [ [] 45 [] ] 123 [ [] 678 [] ] ] .
 [ [ [] 45 [ [] 90 [] ] ] 123 [ [] 678 [] ] ] .
 [ [ [ [] 12 [] ] 45 [ [] 90 [] ] ] 123 [ [] 678 [] ] ] .
 [ [ [ [] 12 [] ] 45 [ [] 90 [] ] ] 123 [ [ [] 345 [] ] 678 [] ] ] .
 [ [ [ [] 12 [] ] 45 [ [] 90 [] ] ] 123 [ [ [] 345 [] ] 678 [ [] 6789 [] ] ] ] .
```

解答例 5.3. 先に述べた誤答例を避ける為に、ポインタ変数へのポインタを使うのも素直な考え方で、全体の記述はすっきりする。しかし、若干記述がややこしくなるようだ。

この方式を徹底するなら、ダミー節点なしでも実装できる。

```
void append( Tree *p, int a )
{
    while( *p != NULL )
    {
        if ( a < (*p)->val )
        {
            p = &((*p)->left);
        }
        else
        {
            p = &((*p)->right);
        }
    }
    *p = new_Node(a);

    return;
}
```

演習 11. 上記の append() の実装を用いて、ダミー節点なしの実装で、tree0.c を書き直してみよ。

演習 12. 今までの例では、入力値に同じ値があるかもしれないことを考慮していなかった。自分のプログラムを改良して、木を作りながらそれぞれの値が何回現れたかをカウントして、値と登場回数との組を入力値の大きさ順に表示せよ。又、登場回数の多い順に並べ替えて表示せよ。

5　応用: 多項式の実装 (続き)

リスト構造を利用した多項式の実装として、まづ単純に、昇冪の順に係数のリストとして扱う (抜けている次数は係数 0 とする) ことにしてみよう。多項式の型定義は例えば次のようにすることが出来よう。

勿論、他の実装でも構わない。一長一短があろう。

```
typedef struct term {
    int coeff;
    struct term *next;
} *Poly;
```

さて、多項式の乗算はどう実装すれば良いだろう。既に上の形で係数を保持している 2 つの多項式 $f(X) = \sum_{i=0}^{m} a_i X^i, g(X) = \sum_{j=0}^{n} b_j X^j$ に対して、$f(X)g(X) = \sum_{k=0}^{m+n} c_k X^k$ とすると、$c_k = \sum_{i+j=k} a_i b_j$ であるから、これを係数とする多項式を作ろう。k を固定する毎に、$i+j=k$ を満たすように i, j を動かすのだが、それには、i を 1 増加させる (= 次の項を見る) 毎に j を 1 減少させる (= 前の項を見る) 必要がある。ところが今迄に見たリスト構造では、リンクを辿って次の項を見ることは出来ても、戻ることは出来ないのであった。

多項式の入力 (初期値設定)・表示・定数倍・加算・減算までは、多項式演算と言っても単純なベクトルの成分演算と同様であるので詳細略。但し、加減算では 2 つの多項式の次数がどちらが高いかで場合分けが必要になることもあろう。

リスト構造に於いて、このようにリンクを辿って戻りたい時には、戻る為のリンクを各節点に備えておく手がある。このようなリストを**双方向リスト (doubly linked list)** という。

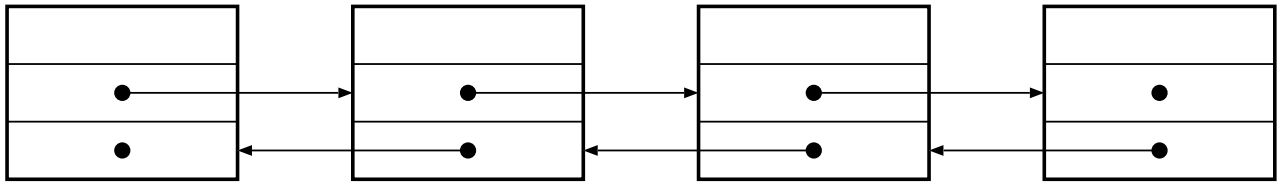

図 7: 双方向リストの概念図

逆にも辿れるのは便利であるが、その為に、各節点が一つ余計なリンクメンバを持つのでメモリ消費が増える他、挿入・削除などのリンク操作の度に繋ぎ替えるリンクが増えて手間も余計に掛かるので、無闇に使えば良い訳ではない。しかし、それを上回る必要・利点があれば、利用を考えて良かろう。

考察 5.0.3. 双方向リストでは、

- 削除する節点そのものだけを指定して、節点の削除とその前後の繋ぎ直しをすること
- 節点を指定して、その前に新節点を挿入すること
- 節点を指定して、その後に新節点を挿入すること

が出来る。どうすれば良いか。

双方向リストを用いた具体的な型定義の例は、以下のようになる。

next: 次の
previous: 前の

```
typedef struct term {
    int coeff;
    struct term *next;
    struct term *prev;
} *Poly;
```

さて、項を逆方向 (高次から低次へ) にも辿れるようになれば、割り算も普通の方法をなぞることで行なえるであろう。と言う訳で、

課題 6 (〆切 学期末迄 (詳しい期日は追って連絡する)). 多項式の型を定義して、入力 (初期値設定)・表示・定数倍・加算・減算・乗算・整除 (商と余りとを求める割り算) を行なう関数を実装し、正しく動作することが一通り確認できるように `main()` 関数でその計算を行なえ。整除については、法 (割る式) の最高次係数が係数環の単数の場合に限って良い。

Extra feature 大歓迎。