# 3 リスト (List) 構造の設計と実装 (続き)

**課題 2 (10/14 出題).** 先の例 (実習 3.1) で、リストの走査 (表示) の部分を関数 print_list() として関数化せよ。

関数 print_list() を呼出す側も含めて、正しく動作するプログラムを完成し、この実習時間中にレポートとして提出せよ。

**解答例 2.1.** 関数 print_list() の部分のみ示す。

```
/*   list0.c   2008-10-14   */
/*   print_list() ver.1 */

void print_list(Node *t)      /* printing list */
{
    while( t->next != NULL )
    {
        t = t->next;
        printf(" -> %d", t->val);
    }
    printf(".\n");

    return;
}
```

返値を void(値を返さない) とせず、例えば int 返しとして、表示した個数 (つまりリストの長さ) を返させることなども考えられる。返させておいて使わなくても別に構わない。実際、実は例えば printf() も int を返している。(返している値は何か?)

**解答例 2.2.** 前ページ下方の方針で、次のようにすることも出来る。

```
/*   list0.c   2008-10-14   */
/*   print_list() ver.2 */

void print_list(Node *t)      /* printing list */
{
    while( t != NULL )
    {
        printf(" -> %d", t->val);
        t = t->next;
    }
    printf(".\n");

    return;
}
```

それぞれの場合について、呼出し側で何を引数として呼べば良いか。

## 3–4　リストからの節点の削除

リスト構造の大きな利点は、リストの途中に新たな節点を**挿入**したり、リストの途中の節点を**削除**したり、という操作が便利にできることである。

既存のリストから節点を削除するには、その手前の節点とその次の節点とを繋いでしまえばよろしい。その要領は次の通り。

(1) 削除したい節点の**手前の節点**に、追跡用ポインタを向ける。
(2) 追跡用ポインタの指す節点の次の節点 (つまり削除したい節点) の次の節点に、追跡用ポインタの指す節点 (つまり削除したい節点の手前の節点) の次へのリンクを向ける。

実はこれは完全版ではない。削除した節点のメモリ領域が確保されたまま残ってしまっている (ばかりか、それなのに決してアクセスできない状態になっている) からである。領域確保と削除とを頻繁に行なう場合には致命的な欠陥となり得る。これを**メモリリーク (memory leak)** という。その対処も含めた完全版は後述。

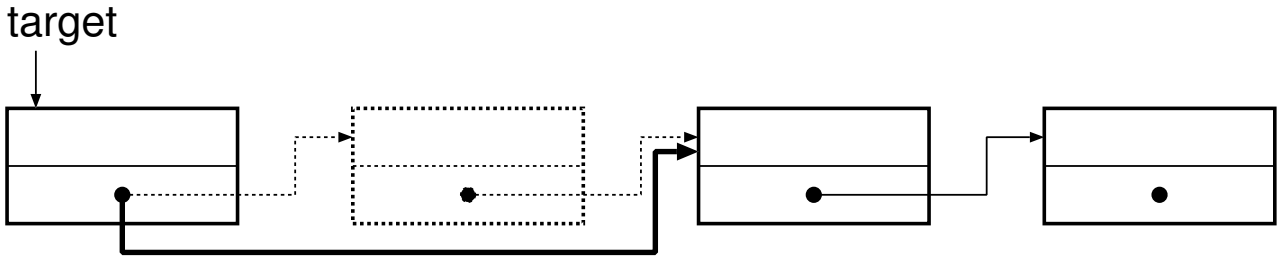


図 3: リストからの節点の削除

**例．**節点 `*p` の**次の**節点を削除する関数 `delete_Node()` の実装の暫定版。

これだけだと関数にするまでもないが、後でメモリリークの対処まで含めたものにする予定で、関数化しておく。

```
void delete_Node( Node *p )
{
    p->next = p->next->next;

    return;
}
```

**実習 3.2.** 実習 3.1 の要領で、入力した整数を各節点の値とするリストを作成・表示した後で、整数値を一つ入力し、それより大きい値を持つ節点を全てリストから削除した後、削除後のリストを辿って全ての値を順に表示するプログラム。(次頁)

```
/*   list1.c   2008-10-21   */
#include <stdio.h>
#include <stdlib.h>

typedef struct node {
    int val;
    struct node *next;
} Node;

void print_list( Node * );
void delete_Node( Node * );

int main( int argc, char **argv )
{
    int a;
    Node head, *target;

    target = &head;
    while( scanf("%d",&a), a != 0 )
    {
        target->next = (Node *)malloc(sizeof(Node));
        target = target->next;
        target->val = a;
    }
    target->next = NULL;
    print_list(&head);

    /* delete if val is greater than a */
    scanf("%d",&a);
    target = &head;
    while( target->next != NULL )
    {
        if ( target->next->val > a )
            delete_Node(target);
        else
            target = target->next;
    }
    print_list(&head);

    return 0;
}
```

関数 print_list() は各自作ったものを、delete_Node() は前頁のものを利用せよ。

while 文の条件式の所にコンマ演算子 , を使ってみた。, で区切って並べた式を左から順次評価し、最後 (一番右) に評価した式の値が、コンマ演算子で結合された式の値となる。濫用すると見難くもなるが、便利な場合が時々ある。ここでは実感に合い、自然であろう。

節点を削除した時には

```
target=target->next;
```

を実行してはいけないことを確認しよう。

「動作確認のすすめ」

- 途中を削除
- 先頭を削除
- 末尾を削除
- 全部削除
- 一つも削除しない

など、様々な状況で動作確認せよ。

## 3–5　動的確保したメモリの解放

前節の例 (実習 3.2) では、節点をリストから削除したと言っても、リストの先頭から辿って到達できなくなっただけで、その節点のメモリ領域は確保されたままで、変数領域として使用中ということになっている。従って、次にメモリの動的確保をする場合には、この領域は使われない。大量のデータを扱い、領域確保と削除とを頻繁に行なう場合には、この領域を再利用したい。その為には「この領域はもう使っていないので次の動的確保で使ってもよろしい」という**領域解放**の手続きが必要である。それには関数 `free()` を用いる。基本形は次の通り。`p` をポインタ変数とし、その指す領域が `malloc()` (など) で動的確保されているとする。

```
free(p);
```

**例．**節点 `*p` の次の節点を削除する関数 `delete_Node()` の実装で、削除した節点の領域を解放し、メモリリークの対処を含めたもの。

```
void delete_Node( Node *p )
{
    Node *t;

    t = p->next;
    p->next = t->next;
    free(t);

    return;
}
```

前の例のように先に `p->next = p->next->next` としてしまうと、解放すべき領域を指すポインタがなくなってしまい、もはやその領域にアクセスすることが出来ない (従って領域解放も出来ない)。そこでそれを一時的に覚えておく為の変数が必要となることに注意しよう。

本小節は純粋にデータ構造という点では本質的でないが、データがどのように計算機上で扱われているか、という点も含め、実際のプログラミングに於いては重要である。

プロトタイプは `stdlib.h` に記述されている。`malloc()` を使うのに呼んでいるだろうからそれでよい。

```
void free(void *)
```

ここでの `void *` 型は汎用ポインタ型で、どんな型へのポインタでも良い。

`free()` で解放する領域は動的確保されたメモリ領域に限る。例えば

```
int a;
free(&a);
```

は不可。

`free()` の引数の値が `NULL` なら何もしない。(エラーではない。)

**注．** このように見てくると、節点へのポインタ Node *型の方が頻繁に現れがちなことに気付く。そこで、節点へのポインタ型の方を、例えば Link 型として、別名定義して扱う方針にすると、次のような型定義・初期化処理になる。

先頭にダミー節点を置き、末尾を NULL にする流儀の場合。*head がダミー節点となる。

```
typedef struct node *Link;
struct node {
    int val;
    Link next;
};

int main( int argc, char **argv )
{
    Link head, target;

    head = (Link)malloc(sizeof(struct node));
    head->next = NULL;

    ...
}
```

**演習 1.** 上記のように節点へのポインタ型の方を通常扱う型として (つまり typedef struct node *Link と別名定義して) 扱う方針に変更して、今までの全てのプログラムを書き換えるとどうなるか。また、今後の課題についても適宜判断して便利な方を用いて書け。

**演習**は、期限までの提出を義務とする課題ではないが、内容が良ければ成績評価に反映させる、一種の研究課題である。積極的に取り組むことを望む。
提出の際の Subject は、Exercise1 等とすれば良い。

---

**演習 2 (Josephus の問題・継子立て).** 2 つの自然数 $n, k$ に対して、次のようなゲーム風の動作を実行して結果 (並びに途中経過) を表示するプログラム josephus.c を作成せよ。

(1) 1 番から $n$ 番までの $n$ 人が番号順に車座になる。($n$ 番の人の次が 1 番に戻る。)
(2) 1 番の人から数えていって $k$ 番目の人が抜ける。(従って $k \leq n$ なら $k$ 番の人、$k > n$ なら 2 巡目以降。) 抜けた場所は詰める。
(3) 残った人の中で、抜けた人の次の人から数えていって $k$ 番目の人が抜け、抜けた場所は詰める。
(4) 以下、これを繰返して最後まで残った 1 人が勝ち。

## 3–6　リストへの節点の挿入

既存のリストの途中への新たな節点の挿入も、ポインタの繋ぎ替えにより、次の要領で容易に出来る。

(1) 挿入したい場所の**手前の**節点に、追跡用ポインタを向ける。
(2) 追跡用ポインタの指す節点の次へのリンクが指す節点 (つまり挿入したい場所の次の節点) を一時記憶用ポインタに記憶する。
(3) 挿入する節点の領域を `malloc()` で確保し、その返値 (つまり今確保した節点へのポインタ) を、追跡用ポインタの指す節点の次へのリンクに代入する (つまり次へのリンクを今確保した新節点に向ける)。
(4) 一時記憶用ポインタが指す節点 (つまり挿入したい場所の次の節点) に、新節点の次へのリンクを向ける。

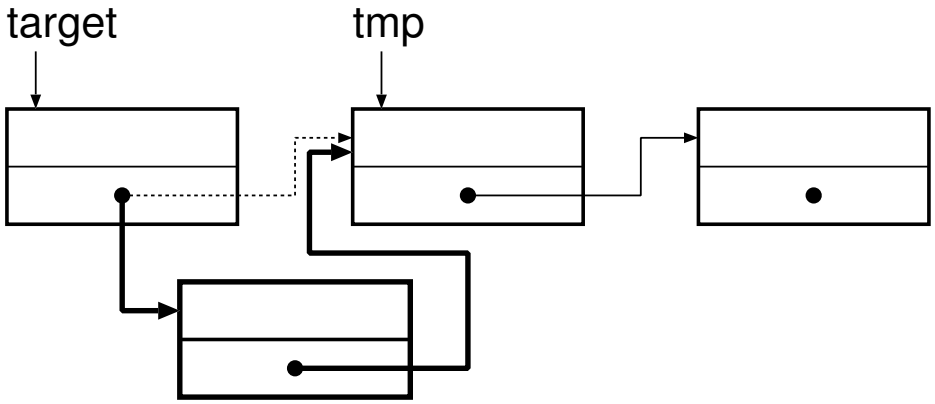


図 4: リストへの節点の挿入

この操作が挿入箇所の前後だけの処理で高速に出来る点が重要。

先頭への挿入の場合、「手前の節点」とはダミーの節点。また、末尾への挿入 (追加) の場合、「次」は `NULL` である。これらの場合に例外処理は必要であるか。

**課題 3 (〆切 11/3(月)).** 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。

今回の課題は 1 週余裕を見て次々回の授業前日までとする。

入力データ数を $n$ 個 (充分大) とする時、比較の回数は如何程と見積もれるか。

- 最大で
- 平均で