担当：　富井尚志　(tommy@ynu.ac.jp)

「アルゴリズムとデータ構造」講義日程

~~1. 基本的データ型~~
~~2. 基本的制御構造~~
~~3. 変数のスコープルール，関数~~
~~4. 配列を扱うアルゴリズムの基礎(1)．最大値，最小値~~
~~5. 配列を扱うアルゴリズムの基礎(2)．重複除去，集合演算，ポインタ~~
~~6. ファイルの扱い~~
~~7. 整列(1)．単純挿入整列・単純選択整列・単純交換整列~~
~~8. 整列(2)．マージ整列・クイック整列~~
~~9. 再帰的アルゴリズムの基礎．再帰におけるスコープ．ハノイの塔など．~~
~~10. バックトラックアルゴリズム．8 王妃問題など．~~
11. 線形リストを扱うアルゴリズム　← 本日の内容
12. 木構造を扱うアルゴリズム(1) 基礎
13. 木構造を扱うアルゴリズム(2) 挿入，削除，バランスなど．
14. ハッシング
15. その他のアルゴリズム

第 11 回「動的データ構造 ～線形リスト～ 」

☆ 静的データ構造と動的データ構造

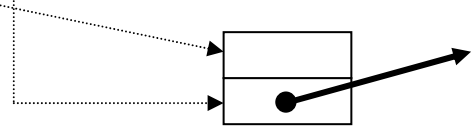
◎ 静的データ構造

変数 … **データ型**で指定
→　変数の値域や記憶のされ方（記憶の大きさ，使われ方）が固定される
→　「静的なデータ構造」

◎ 動的データ構造

・一つの変数（基本型，配列など）で表現されるデータよりも複雑な情報構造
・その情報構造が計算過程において変化

※ 成分は，「静的データ構造」 ＋ 「他の静的データ構造へのポインタ」
固定部分　ある固定部分を別の固定部分につなげる「のりしろ」



・再帰的データ型によって特徴付けられる．

再帰的データ型は以下の構造からなる型．
a) ある基本的データ構造による成分
b) その再帰的データ型に属す 1 個以上の成分（ ← 再帰部分）

☆ 記憶の動的割り当てとポインタ

◎ 再帰的データ構造は，（全体として）大きさが変化する.

→　一定の大きさの記憶領域を割り当てることはできない.

◎ 記憶の動的割り当て

・プログラムの実行時にシステムから記憶を割り当ててもらう.

・C 言語における記憶の動的割り当て

→　ライブラリ関数 malloc, free が用意されている.

```
void *malloc( size_t size )
size で指定した大きさの領域を確保
その領域の先頭番地(ポインタ)を返す
```

```
void free( void *ptr )
ポインタ変数 ptr の値で指定した番地の
領域を解放する(大きさはシステムが計算)
```

◎ 再帰的データ型におけるポインタの役割

・再帰型データ型の構成の b)（前頁）の成分

- **あるデータを直接埋め込む方法** ×

  埋め込むデータの大きさが可変なので，一つの（再帰的データ構造型）データの記憶領域の大きさが決まらない

- **あるデータへのポインタとする方法** ○

  ポインタ（を値に持つ変数）自身の大きさは固定なので，一つの（再帰的データ構造型）データの記憶領域の大きさは決まる.

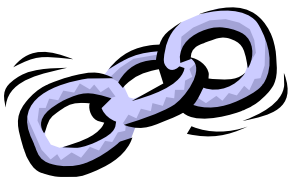
・一つのデータ構造内のポインタ部分に新しく得た記憶領域への番地を代入して扱う.

☆ 動的データ構造の例 --- 線形リスト

◎ 線形リスト --- 数が変化する項目の並び（cf. 配列）

(a) 「空リスト」はリスト

(b) 一つの「項目」とリストをつないだものもリスト

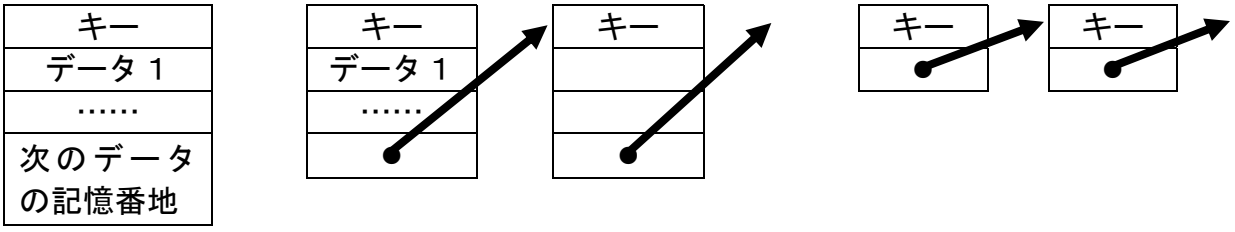


◎ データ構造

```
struct list {
   int key;
     ...
   struct list *next;
}
```

← 識別キー（なくてもよい）
← 一つの項目に保持すべきいくつかのデータ
← 次の項目へのポインタ.
　（次の項目が記憶されている領域の番地）

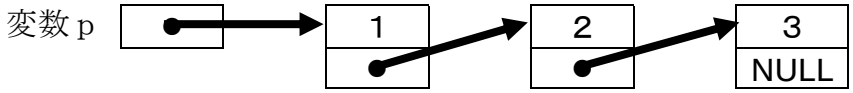
- 図示すると，**図(a)**のようになる．
- 図上で次のデータを指し示したい場合には，直観的には番地を示すよりもそれと同等の意味を持つ矢印の方がわかりやすい（**図(b)**）．
- 以後，簡単のために，「識別キー」「次の項目へのポインタ」以外の項目内データは扱わないことにする．本質は変わらない．これを図示すると**図(c)**となる．



図(a)　図(b)　図(c)

◎ 線形リストの例

・ 項目 1, 2, 3（識別キーの値）がこの順番で並んでいるリストがあり，そのリストへのポインタを値に持つ変数によりそのリストが保持されている場合．



- 最後の NULL は再帰的データ構造の終端を示す特別な値 ＝ **空リスト**である．

・ C 言語における**「構造体へのポインタ」の値参照**

p が指す構造体の key フィールドの参照を意味する

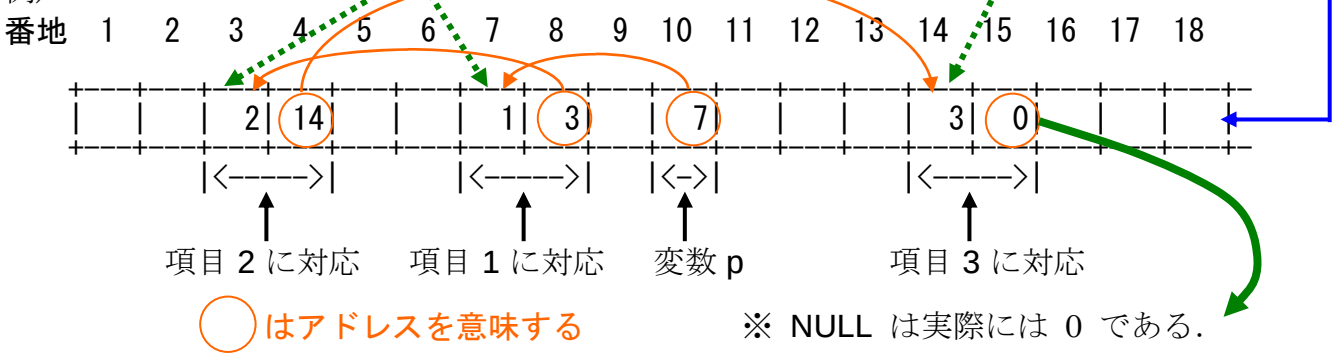
```
(*p).key == 1,   (*((*p).next)).key == 2,   (*((*((*p).next)).next)).key == 3
```

※ (*p).key は p->key と書くことができる．

```
p->key == 1,   p->next->key == 2,   p->next->next->key == 3
```

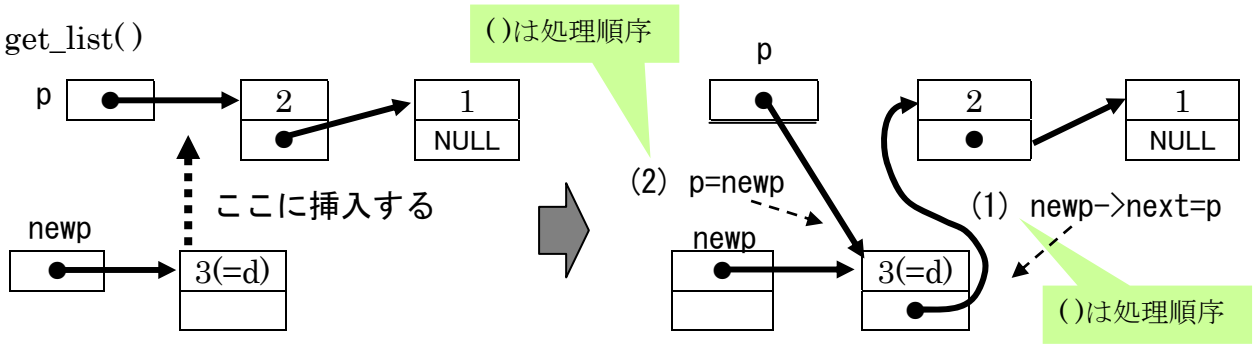
◎実際の記憶領域上では

例)



☆ リストに対する操作（サンプルプログラム listop.c ）

◎リスト生成　（p が指すリストの先頭に新しい項目を挿入）

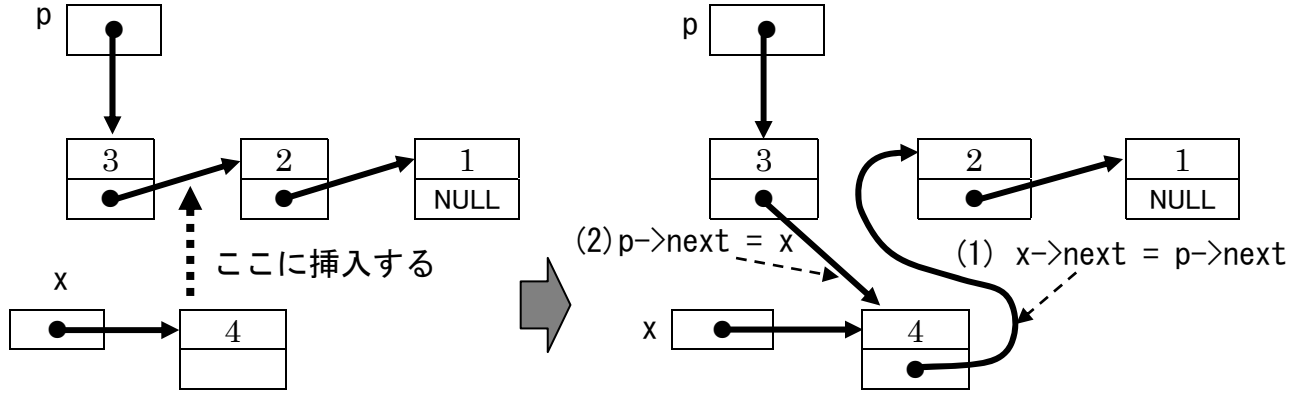


◎リストの走査

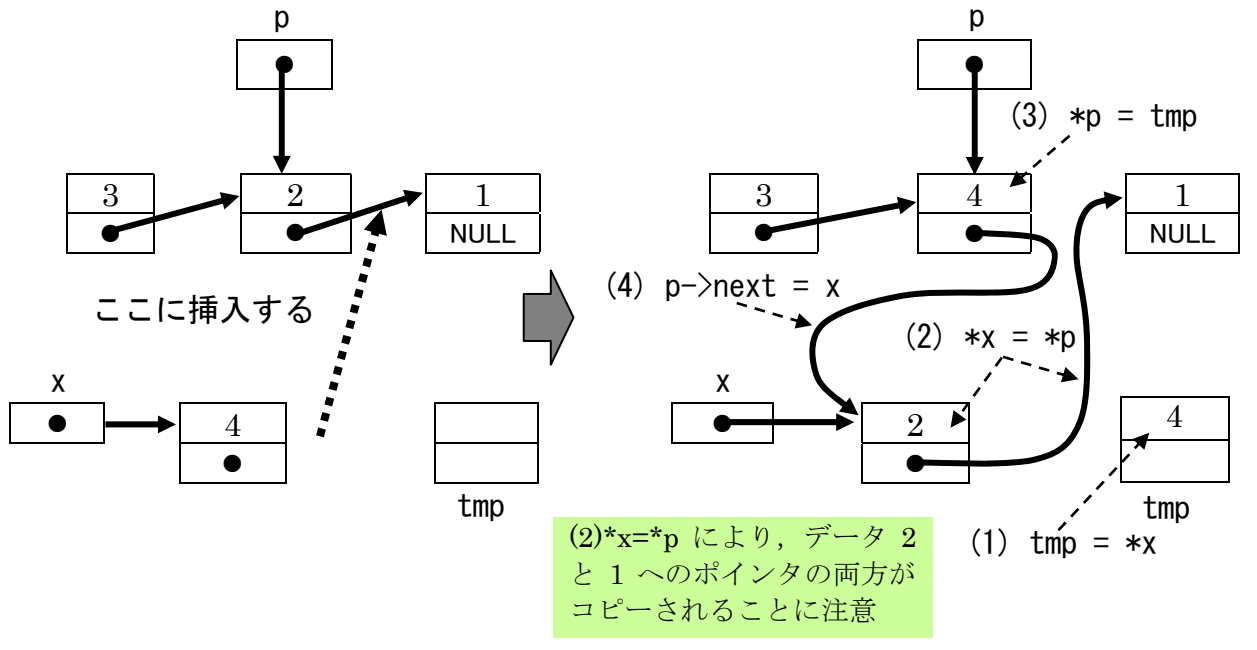
print_list( )

◎リストへ新しい項目を挿入（p で指定した項目の直後，直前）

insert_after( )： p の直後

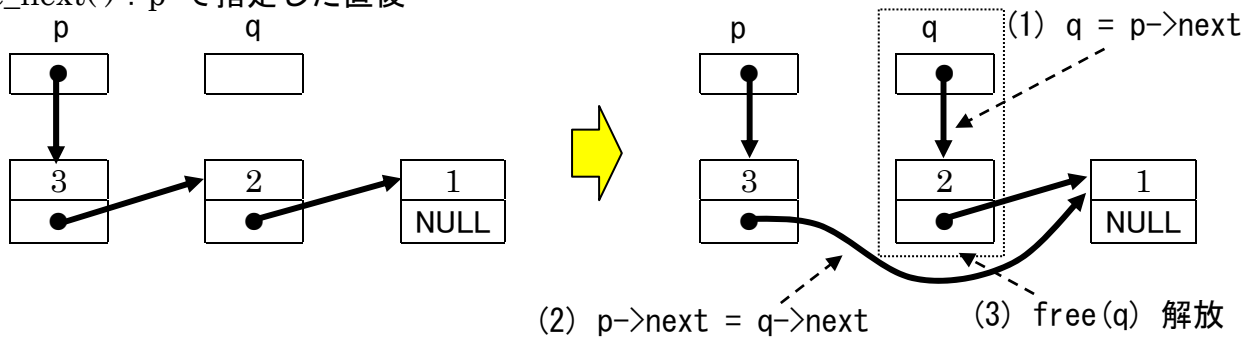


insert_before( )： p の直前

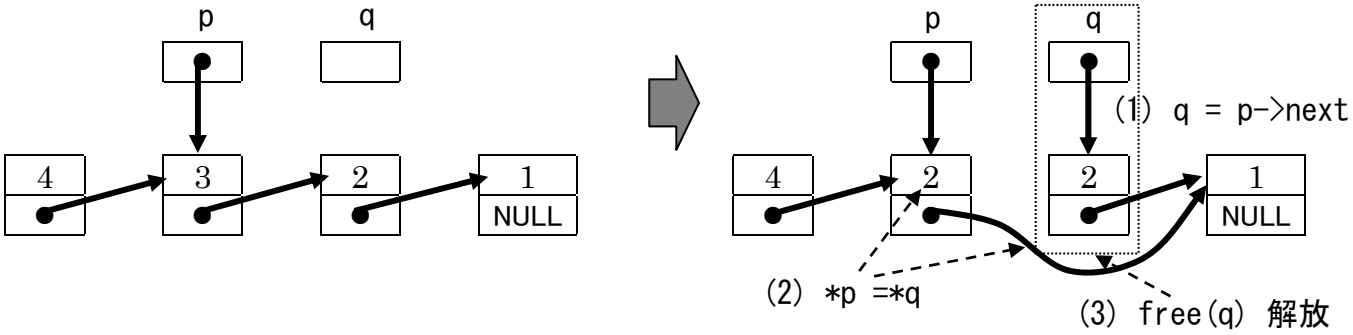


◎リストから項目を削除（ p で指定した項目の直後，それ自身）

delete_next( ) : p で指定した直後



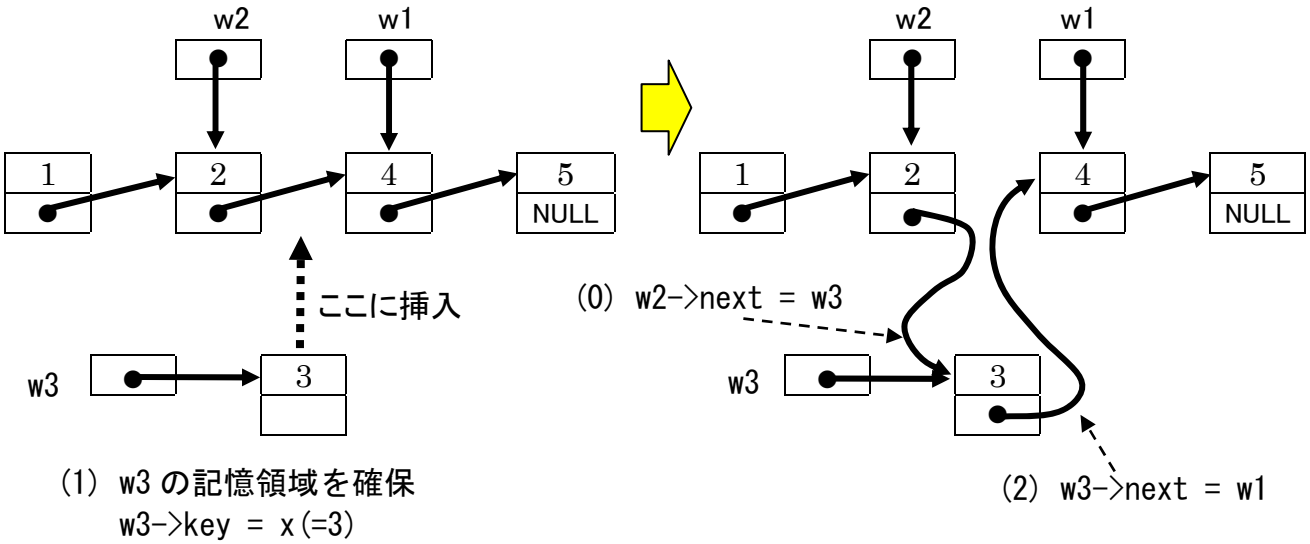
delete_it( ) : p で指定したそれ自身（最後の項目は対象外とする）



☆ 整列リストの探索/生成（サンプルファイル sortedlist.c ）

・挿入場所は挿入項目より大きなキーが現れないとわからない．
　　→ 走査中のポインタの「前」が挿入場所

・先ほどと別の方法を使ってみる．走査するポインタを二つ使う．



```
/*************************************************************
     「アルゴリズムとデータ構造」
     サンプルプログラム listop.c
     <<動的データ構造の例: 線形リストの作成, 走査>>
     copyright (c)  T.Mori <mori@forest.dnj.ynu.ac.jp>
*************************************************************/
#include <stdio.h>
#include <stdlib.h>
/* 線形リストのための再帰的データ型の定義 */
struct list {
    int key;
    struct list *next;
};

struct list *get_list(void);
void print_list(struct list *p);
void insert_after(struct list *x, struct list *p);
void insert_before(struct list *x, struct list *p);
void delete_next(struct list *p);
void delete_it(struct list *p);
int get_data(void);

#define EOD -1
/* 初期データリスト */
int a[ ] = { 1, 2, 3, 4, 5, 6, EOD };
-------------------------------------------------------------
int main(void)
{
    struct list *listptr, *new1, *new2;

    listptr = get_list( );
    print_list( listptr );

    new1 = (struct list *)malloc(sizeof(struct list));
    new1->key = 111;
    insert_after( new1, listptr->next->next );
    print_list( listptr );

    new2 = (struct list *)malloc(sizeof(struct list));
    new2->key = 222;
    insert_before( new2, listptr->next->next );
    print_list( listptr );

    delete_next( listptr->next->next );
    print_list( listptr );

    delete_it( listptr->next->next );
    print_list( listptr );
    return 0;
```

list, key, next は勝手につけた名前

EOD : End Of Data の意

関数の切れ目に後から入れた線

struct list（構造体）へのポインタ

main 関数

```
50  }
51  ----------------------------------------------------------------
52  /* リストの生成例 */
53  struct list * get_list( void )
54  {
55      int d;
56      struct list *p,*newp;
57
58      p = NULL;      /* NULL は空リストを表す */
59      while( ( d = get_data( ) ) != EOD) {
60          newp = (struct list *)malloc(sizeof(struct list));
61                                          /* ↑新しい項目のメモリ確保 */
62          newp->key = d;    /* 新しい項目の取得 */
63          newp->next = p;   p = newp; /* 古いリストの先頭に新しい項目を挿入 */
64      }
65      return p;
66  }
67  ----------------------------------------------------------------
68  /* データ取得 */
69  int get_data(void)
70  {
71      static int i=0;
72
73      return a[i++];
74  }
75  ----------------------------------------------------------------
76  /* リストの走査の例: 印刷 */
77  void print_list(struct list *p)
78  {
79      while (p != NULL) {
80          printf("<%d> ", p->key);
81          p = p->next;
82      }
83      printf("¥n");
84  }
85  ----------------------------------------------------------------
86  /* リストへの挿入: 指定要素の直後 */
87  /* p で示される構造体の次に x で示される構造体を挿入 */
88  void insert_after(struct list *x, struct list *p)
89  {
90      x->next = p->next;
91      p->next = x;
92  }
93  ----------------------------------------------------------------
94  /* リストへの挿入: 指定要素の直前 */
95  /* p で示される構造体の前に x で示される構造体を挿入 */
96  void insert_before(struct list *x, struct list *p)
97  {
98      struct list tmp;
```

struct list（構造体）へのポインタ

関数 get_list

static は静的変数の意．最初に呼ばれたときに i=0 と初期化され，以降，この関数を離れても存在し続ける

関数 get_data

i++は，i の値を使ってから i の値を 1 を加える．
(++i)は 1 を加えてから i の値を使う．
a[0](=1), a[1](=2), a[2](=3),...と，呼ばれるたびに 1 つずつ項目番号が増える．

関数 print_list

記号<，p が指す key の値，記号>の順番に画面に表示する．

関数 insert_after

関数 insert_before

```
99
100     tmp = *x;
101     *x = *p;
102     *p = tmp;
103     p->next = x;
104 }
105 ----------------------------------------------------------------
106 /* リストからの削除: 指定要素の直後 */
107 /* p で示される構造体の直後の構造体が削除される */
108 void delete_next(struct list *p)
109 {
110     struct list *q;
111
112     q = p->next;
113     p->next = q->next;
114     free(q);
115 }
116 ----------------------------------------------------------------
117 /* リストからの削除: 指定要素 */
118 /* p で示される構造体がリストから削除される */
119 /* 一番最後の項目（p->next == NULL）は対象外 */
120 void delete_it(struct list *p)
121 {
122     struct list *q;
123
124     if ( p->next != NULL ){
125         q = p->next;
126         *p = *q;
127         free(q);
128     }
129 }
130 ----------------------------------------------------------------
131
```

関数 delete_next

関数 delete_it

【実行結果】

```
<6> <5> <4> <3> <2> <1>
<6> <5> <4> <111> <3> <2> <1>          (→ <4> の後に <111> を入れた)
<6> <5> <222> <4> <111> <3> <2> <1>    (→ <4> の前に <222> を入れた)
<6> <5> <222> <111> <3> <2> <1>        (→ <222> の次を消去した)
<6> <5> <111> <3> <2> <1>              (→ <222> を消した)
```

```
/*************************************************************
     「アルゴリズムとデータ構造」
     サンプルプログラム sortedlist.c
     <<動的データ構造の例: 整列リストの探索>>
     copyright (c)  T.Mori <mori@forest.dnj.ynu.ac.jp>
*************************************************************/
#include <stdio.h>
#include <stdlib.h>

struct list {
     int key;
     int count;
     struct list *next;
};
void print_list(struct list *p);
void search(int d, struct list *root);

#define EOD -1
/* 初期データリスト */
int a[ ] = { 2, 1, 5, 1, 3, 4, 6, 1, 5, 6, EOD };
/* 番人 */
struct list *sentinel;
```

大域変数（至る所で参照・代入可能）

```
int main(void)
{
     int d;
     struct list *root;

     root = (struct list *)malloc(sizeof(struct list));
     sentinel = (struct list *)malloc(sizeof(struct list));
     root->next = sentinel;

     while((d=get_data( )) != EOD)
          search( d,root );

     print_list( root );
     return 0;
}
```

main 関数

```
/* リストの探索ならびに挿入 */
/* キーが既にリストに存在すならば出現回数を増やし */
/* 存在しないならばキーの昇順で整列した位置に挿入する */
void search( int x, struct list *root )
{
     struct list *w1,*w2, *w3;

     w2 = root;
     w1 = w2->next;
     sentinel->key = x;
```

関数 search

```
50
51      while (w1->key < x) {
52          w2 = w1;
53          w1 = w2->next;
54      }
55      if (w1->key == x && w1 != sentinel)
56          w1->count++;
57      else {
58          w3 = (struct list *)malloc(sizeof(struct list));
59          w3->key = x;
60          w3->count = 1;
61          w3->next = w1;
62          w2->next = w3;
63      }
64  }
65  ----------------------------------------------------------------
66  /* リストの印刷 */
67  void print_list(struct list *p)
68  {
69      while ((p = p->next) != sentinel)
70          printf("<%d,%d 回> ", p->key,p->count);
71      printf("¥n");
72  }
73  ----------------------------------------------------------------
74  /* データ取得 */
75  int get_data(void)
76  {
77      static int i=0;
78
79      return a[i++];
80  }
81  ----------------------------------------------------------------
82
83  【実行結果】
84  <1,3 回> <2,1 回> <3,1 回> <4,1 回> <5,2 回> <6,2 回>
```

関数 print_list

関数 get_data