関数と再帰呼び出し

再帰呼び出しに必要な知識

- いろいろな変数
  - 局所変数
  - 大域変数
  - 静的変数
- 有効範囲（scope)と有効期限（extent)
- 引数の値渡し

いろいろな変数

- 局所変数 (local variable)
  - 関数の中に宣言されているもの
  - 関数の定義に書かれたパラメータ変数もその１つ
- 大域変数 (global variable)
  - 関数の外に宣言されているもの
- 静的変数 (static variable)

局所変数

- 局所変数
  - 関数の中に宣言されているもの
  - 関数の定義に書かれたパラメータ変数もその１つ
- 関数には、いろいろなプログラムが書ける
  - 局所変数は、作業領域としてつかう。
- 局所変数の値は、関数の実行が終わると無効になる。
- 違う関数の中の局所変数は、使えない

大域変数

- 大域変数
  - 関数の外に宣言されているもの
  - 関数の外に書くことができる
- 関数の間で共有するデータをいれておくもの。

大域変数の悪い例

```
int t;

int foo(int x,int y)
{
     t = x + y;
     return t;
}
```

```
int goo(int x, int y)
{
        t = x-y;
        z = foo(x,y);
         … ;
        return t;
}
```

副作用(side effect)

局所的しか使わない一時的なデータにはできるだけ局所変数を使い、
必要なときだけ大域変数を使うようにしましょう。

変数の有効範囲と有効期限

- 有効範囲　ー　どこで使えるか
  - 局所変数
  - 大域変数
- 参照できる範囲　スコープ　（Scope) という
- 有効期限　ー　値がいつまで有効か
- 保持される期間　有効期限（extent) という

変数の有効期限 (extent)

- 局所変数は、その宣言された関数が終了すると値が無効になる
  - 変数のメモリが解放される
  - 一時的に用いるのにつかう
- 大域変数は関数が終わっても値は保持される
  - 関数が終わってもなにか値を保持する必要がある場合には大域変数にいれておく
- 関数から返すことのできる値は１つなので、たくさんの値を返したい場合には、大域変数を用意してそこにいれて、関数からも戻ったときにその変数を参照する

静的変数　static variable

- 局所変数にstaticをつける
- 有効範囲は、局所変数と同じ
- 有効期限は、大域変数と同じ

```
foo(){
    static int i;
    …
    i = 10;
    …
    x = i;
}
```

引数の値渡し

- 関数呼び出しでは、引数に書かれた式を実行した結果の値が使われる
- これを値渡し　call by value という

引数の値渡し

```
main() {
    int i;
    i = 10;
    j = foo(i);
    printf("i=%d¥n",i);
  }

int foo(int i)
{
     i = 100;
     return i;
}
```

関数の再帰呼び出し

- ある関数が、自分自身を呼び出すことを
  再帰呼び出し(recursive call)

```
int factorial(int n)
{
     if(n == 0) return 1;
    else return n*factorial(n-1);
}
```

関数の再帰呼び出し

- ある関数が、自分自身を呼び出すことを
  再帰呼び出し(recursive call)

- 再帰呼び出しの考え方は情報科学のなかでもっとも重要なことの一つ

- この考え方を身につけることによって、いろいろな問題が簡単に解ける強力な考え方です。

再帰の考え方

- あるnについて問題を解くときに、n-1についての解答でとくことができるときに使うことができます。

- プログラムを考えるときに、特定のケース、特定の数についてプログラムを考えるだけでなく、それを一般的な入力、一般的なnについて考えることはプログラミングについての非常に重要な考え方です。

ハノイの塔

- ハノイの塔とは、３本の柱があり、一番左の柱に下から大きな盤、その上に順に小さな盤が乗っているというもの
- 問題は以下の条件で、左の柱から、右の柱に移すというパズル
  - 一度に１枚の盤しか移すことができない（片手しか使えない）
  - 小さな盤の上に大きな盤を乗せてはいけない（壊れてしまう）
- 中間的な置き場として真ん中の柱を使うことができますが、上の条件は満たしていなくてはなりません。

ハノイの塔

- まず、大小２枚の場合を考えてみる。
  - まず小さい盤を真ん中に移して、大きな盤を左に移し、最後に小さい盤を左に移せばよいことになります。
  - では３枚ではどうでしょうか。この場合、まず、２枚の手順を使って、上２枚を真ん中に移します。それで、一番下にある盤を右の柱に移し、また２枚の手順を使って、真ん中から右に移せばいいということなります。

- これを一般化して、n枚の場合には、
  - n-1枚をあいているところに移す。
  - １番下の番を目的のところ（右の柱）に移す。
  - n-1枚を目的のところに移す。

- n枚の盤を移すという関数を定義して、その関数に再帰呼び出しを使うことで、エレガントにプログラミングすることができます。

数字のプリント

- 任意の整数を一文字の出力関数 (putchar)を使って、プリントアウトするという問題です。例えば、123という数字の場合は”1”,”2”,”3”と出力するという問題
  - 一桁の場合であれば、単にその数字を’0’に加えることによって、出力します。
  - では２桁はどうでしょう。この場合はまず、10で割って、その商が２桁目なので、その数字を印刷します。あまりを１桁にしてプリントすればいいわけです。

- これを一般化してn桁の数とすると、数dに対し、
  - まず、一桁、つまり10よりも小さければ、putchar(d+’0’)
  - 最下位以外のn-1桁について、プリントする。つまり、d/10を印刷。
  - 最下位の桁、d%10を印刷。

数字のプリント

```
void print_number(int d)
{
   if(d >=10) print_number(d/10);
                    /* 上の桁を出力 */
   putchar('0'+d%10);
                    /* 最下位の桁を出力 */
}
```

ナップサック問題

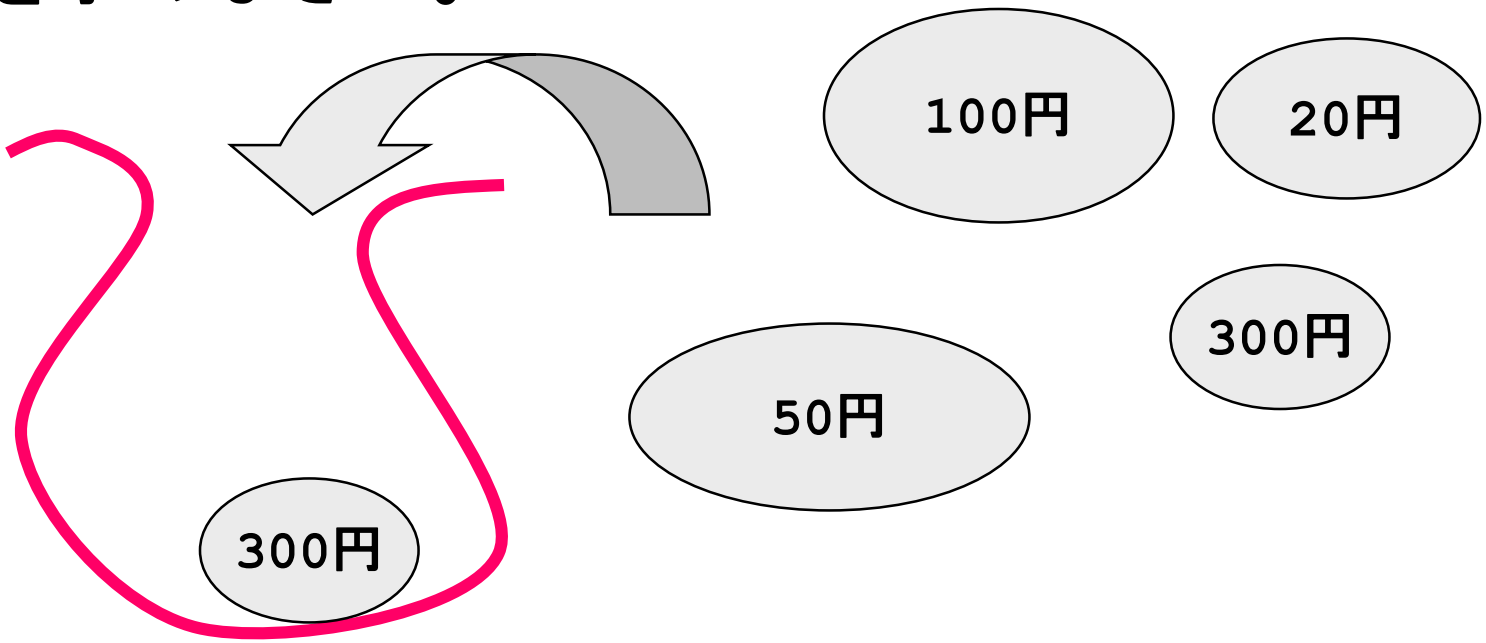
- いくつかの荷物を袋に最大の値段になるように袋に詰める組み合わせを求める問題

- N個の荷物があり、個々の荷物の重さをw_i、値段をp_iとする。袋(knapsack)には最大Wの重さまでいれることができる。このとき、袋にいれることができる荷物の組み合わせを求め、そのときの値段を求めなさい。

- 簡単に考えれれば、2^Nの組み合わせがある。

ナップサック問題

- N個の荷物があり、個々の荷物の重さをw_i、値段をp_iとする。袋(knapsack)には最大Wの重さまでいれることができる。このとき、袋にいれることができる荷物の組み合わせを求め、そのときの値段を求めなさい。



考え方

- i番目の荷物を入れるときに
  - あとどのくらいいれることができるのか　ー　M
  - i番目までいれたときの、全体の値段　ー　cp

- もしも、i番目の荷物（値段p[i], 重量w[i]) がはいるのであれば,　つぎは
  - 値段が cp + p[i]
  - 残りの重さは、M-w[i]

  になる。

- いれないのであれば、Mとcpは変わらず。

```
int knap_search(int i,int cp, int M)
{
  int Opt;
  int l,r;
  if (i < N && M > 0){
     l = knap_seach(i+1,cp+P[i],M-W[i]);
     r = knap_serach(i+1,cp,M);
    if(l > r) Opt = l;
      else Opt = r;
  } else  Opt = cp;
  return(Opt);
}
```

```
int knap_search(int i,int cp, int M)
{
  int Opt;
  int l,r;
  if (i < N && M > 0){
    if(M >= W[i]){
     l = knap_seach(i+1,cp+P[i],M-W[i]);
     r = knap_serach(i+1,cp,M);
       if(l > r) Opt = l;
       else Opt = r;
    } else
       Opt = knap_search(i+1,cp,M);
  } else Opt = cp;
  return(Opt);
}
```

```
int N;
int Cap;
int W[MAX_N];
int P[MAX_N];

int main()
{
    データの読み込み…
    opt = knap_search(n,0,Cap);

}
```

ここまでの内容

- 変数の有効範囲　scope
- 変数の有効期間　extent

- 引数の値渡し　call by value

- 再帰呼び出し

- 教科書
  - ８．６
  - ８．７
  - ８．３

複雑なデータ構造とポインタ（もう一度）

これまでのおさらい（ポインタと構造体）

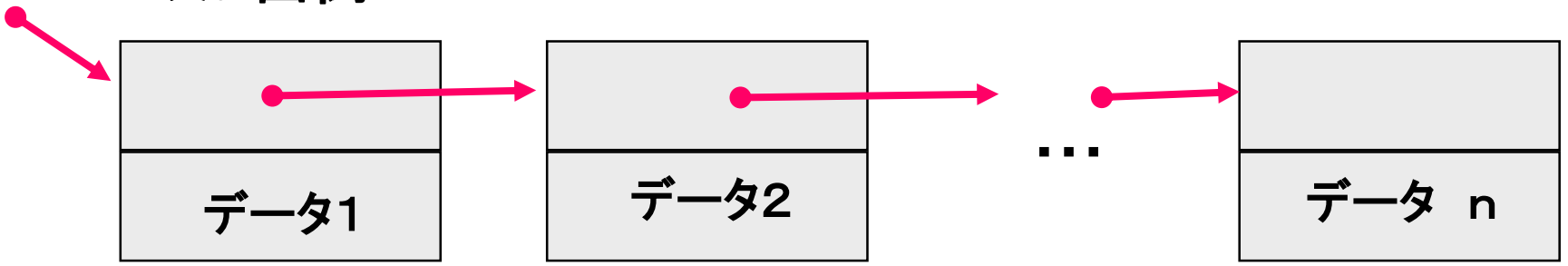
- ポインタ
  - ポインターってなに？
  - ポインター変数
  - 関数の引数とポインター
  - 配列とポインターの関係
  - データ型
- 構造体
  - 構造体ってなに？
  - データ型と構造体
  - Typedef
  - ポインターを使った構造体の参照
- 動的なメモリ割り当て

リスト構造
＝
構造体
＋
ポインタ
＋
動的な
メモリ割り当て

（線形）リスト構造とは

- 一列に並んだデータをあらわすデータ構造
  - 集合、順序がついたものの並び
  - 必要に応じて、データを確保

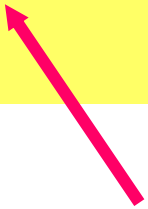
- 配列だと
  - データの数の上限をあらかじめ決めておかなくてはならない。
  - 途中にデータを挿入したり、途中のデータを削除するのが面倒



リスト構造の定義

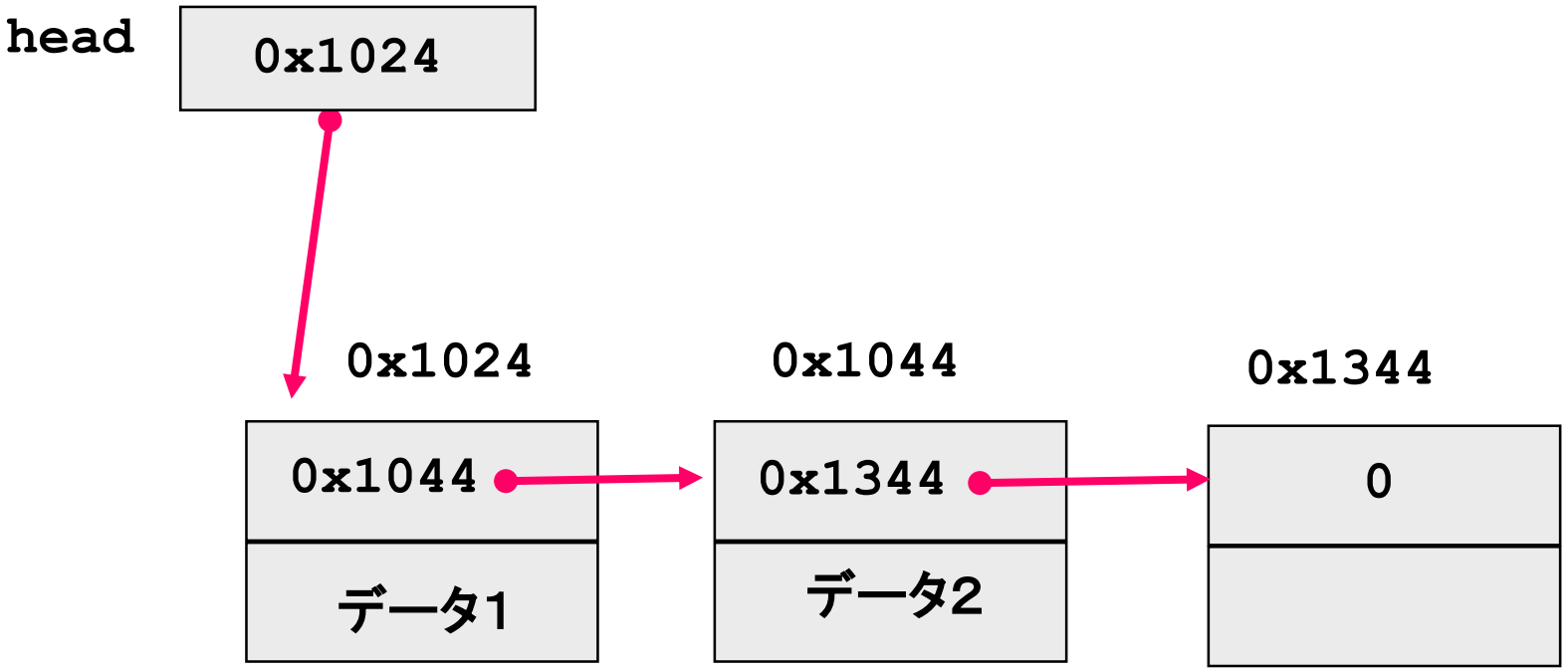
- 自分のデータ型を参照するポインタのメンバーを持つ

```
struct List {
    struct List *next;
    int data;
};
```



ここのデータは場合によっていろいろなものをつけることができる！

```
strut List *head;
```



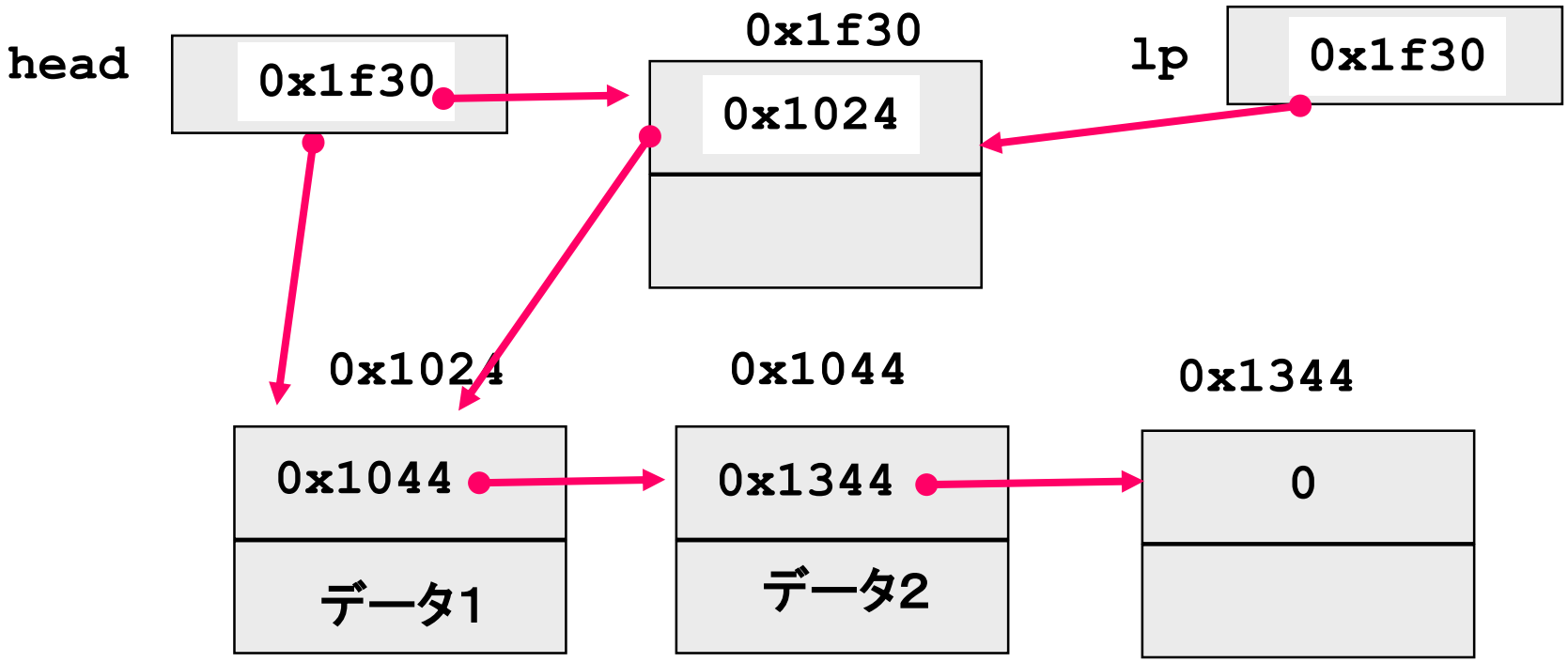
リストへのデータの追加

- 大域変数として、リストを持っている変数head
- データを追加する関数addList

```
struct List *head = NULL;

void addList(int x){
  struct List *lp;
  lp = (struct List *)malloc(sizeof(struct List));
  if(lp == NULL){
     printf("no more memory¥n");
     exit(1);
  }
  lp->next = head;
  lp->data = x;
  head = lp;
}
```

```
struct List *head;      struct List *lp;
```




```
lp->next = head; lp->data = x;    head = lp;
```

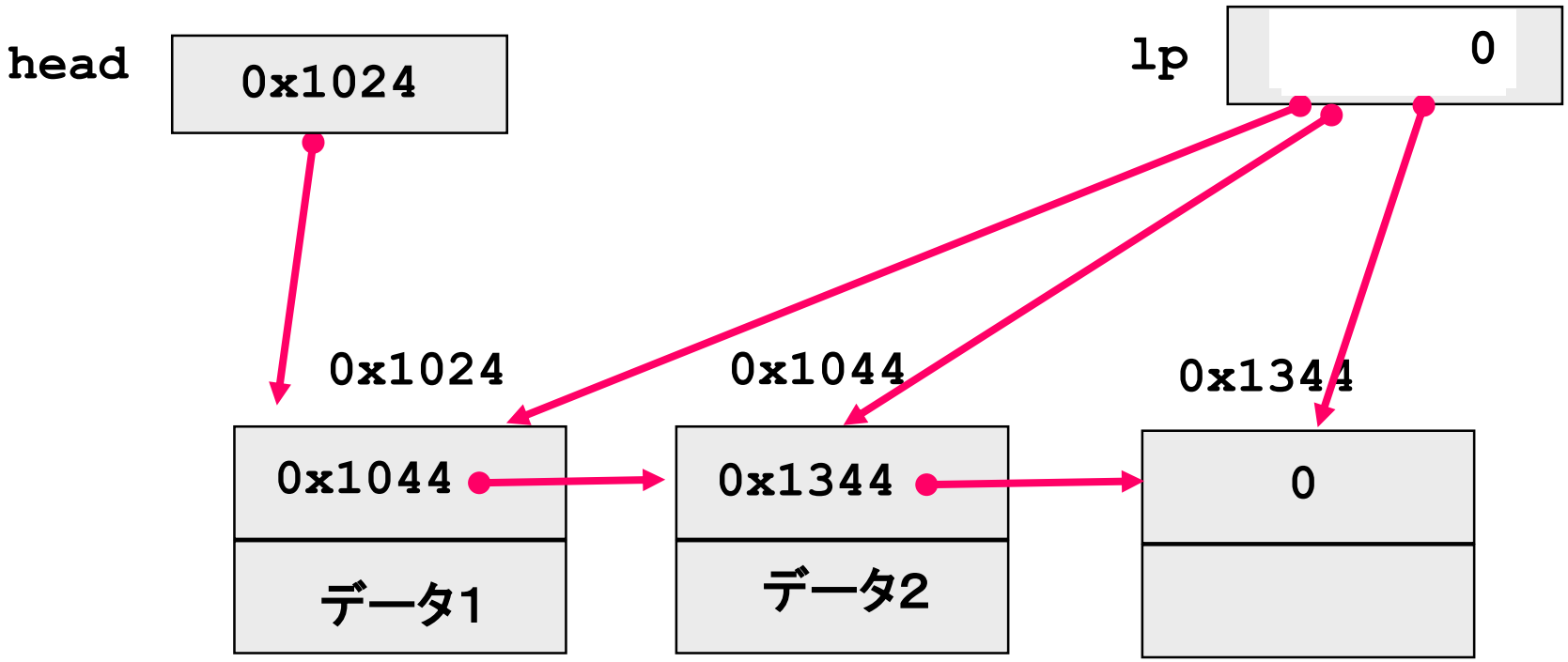
リストにあるデータの検索

- リストにデータがあったら、１、なかったら０を返す関数 isExist

```
void isExist(int x){
   struct List *lp;
   for(lp = head; lp != NULL; lp = lp->next){
      if(lp->data == x) return 1;
   }
   return 0;
}
```

```
strut List *head;
```

```
struct List *lp;
```



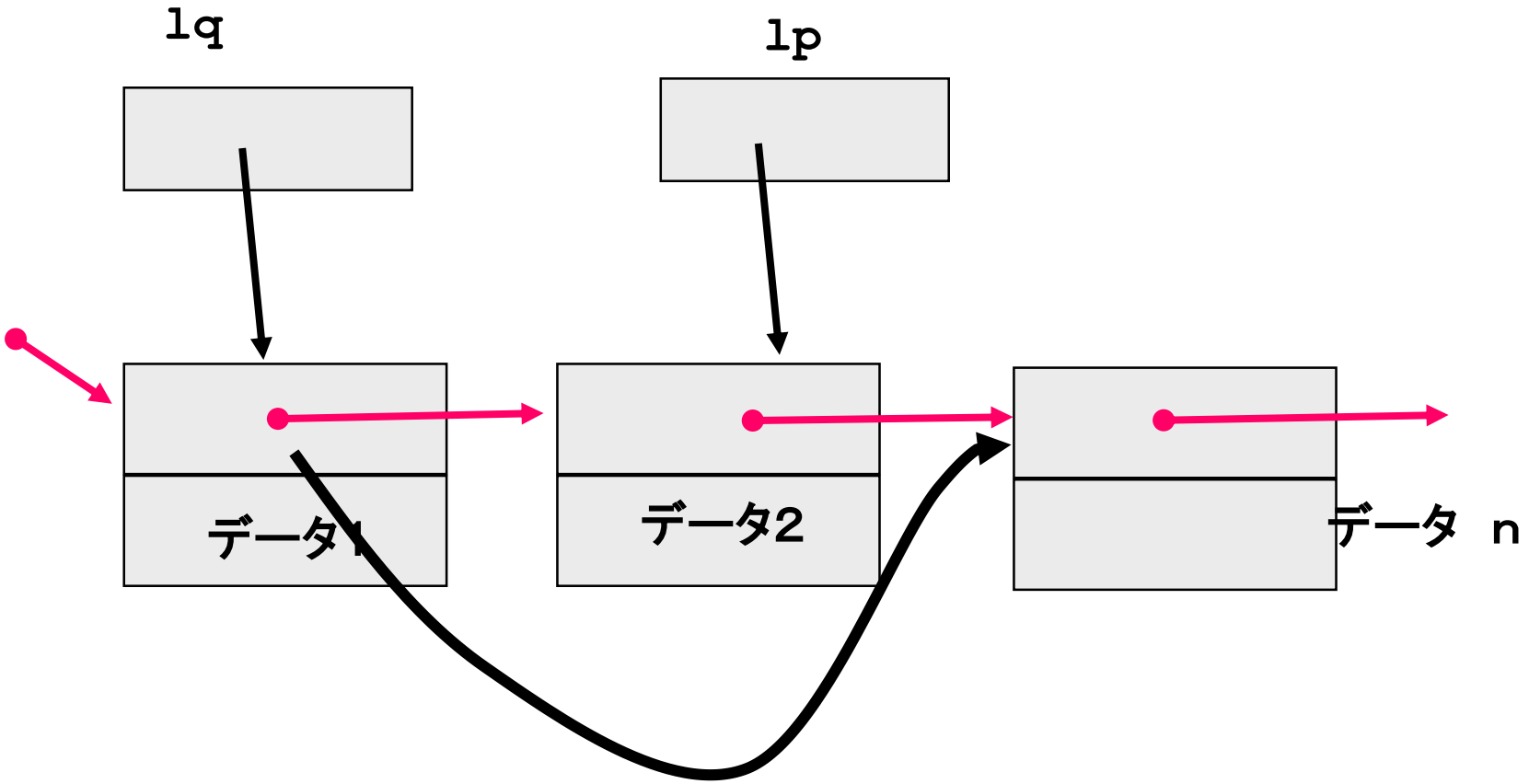

```
for(lp = head; lp != NULL; lp = lp->next){
```

リストにあるデータの削除

- リストにデータがあったら、削除する removeList

```
void removeList(int x){
  struct List *lp,*lq;
  lq = NULL;
  for(lp = head; lp!= NULL; lp = lp->next){
    if(lp->data == x){
       if(lq == NULL) head = lp->next;
       else lq->next = lp->next;
       free(lp);
       return;
    }
    lq = lp;
  }
}
```

リストからの削除

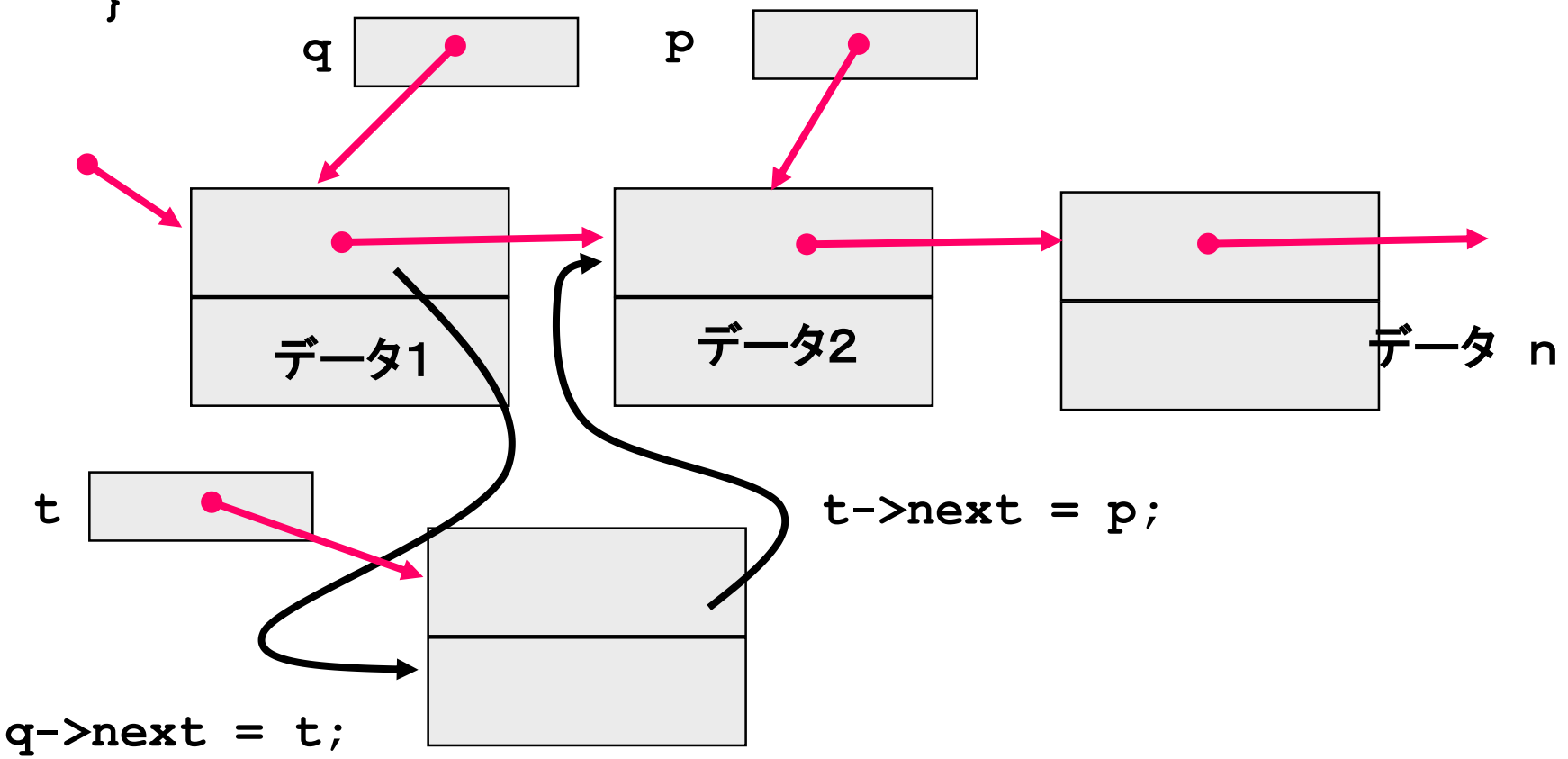



```
lq->next = lp->next;
```

```
void insert_list(char *name, int x)
{
  struct record *p, *q, *t;
  t = (struct record *) malloc(sizeof(struct record));
  if (t == NULL) {
    printf("Out of memory¥n");
    exit(1);
  }
  strcpy(t->name, name);
  t->point = x;
  q = NULL;
  for(p = head; p != NULL; p = p->next){
    if(p->point >= x) break;
    q = p;
  }
  if(q != NULL)
    q->next = t;
  else
    head = t;
  t->next = p;
}
```

リストへの挿入

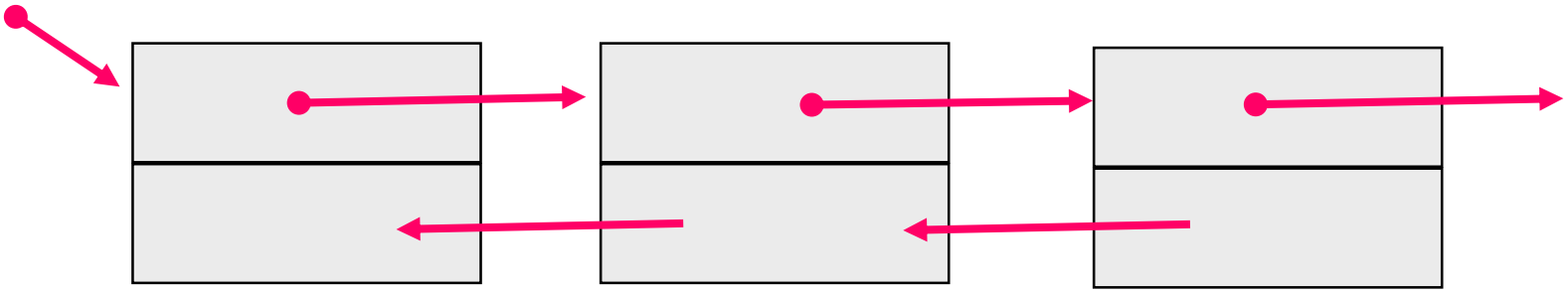
```
for(p = head; p != NULL; p = p->next){
    if(p->point >= x) break;
    q = p;
  }
```



Double linked List

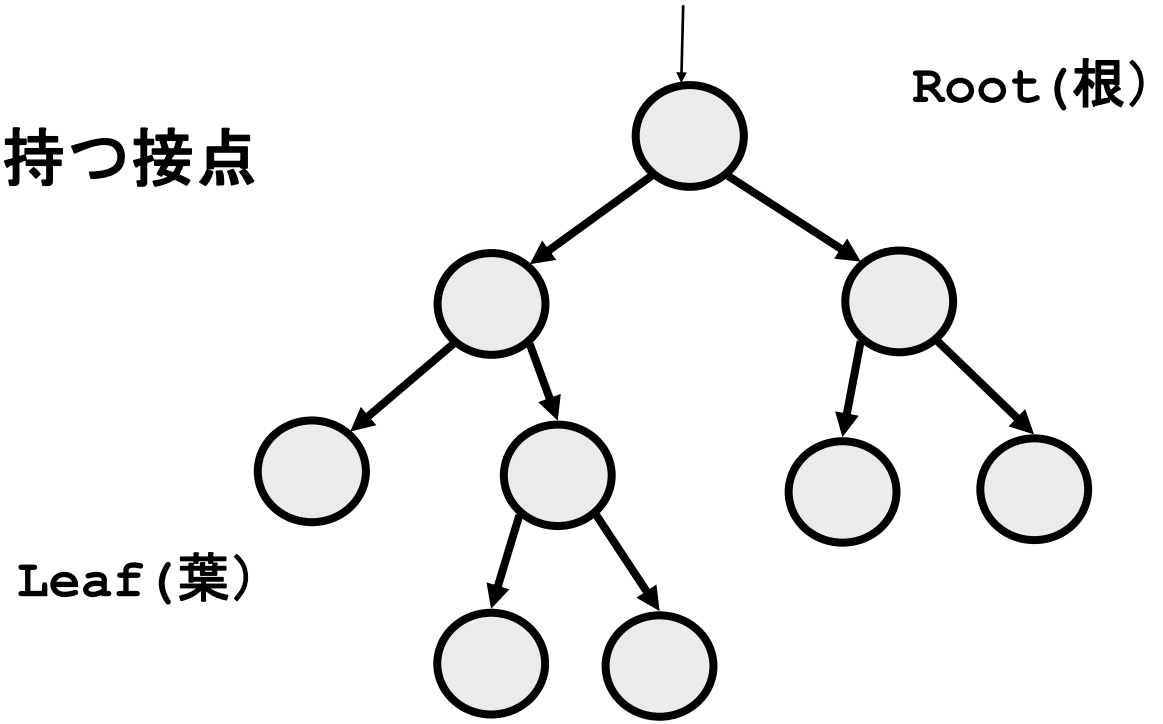
- 後ろだけでなくて、前へのポインタをもっているリスト

```
struct DoubleList {
        struct DoubleList *prev,*next;
        int data;
  };
```



木構造

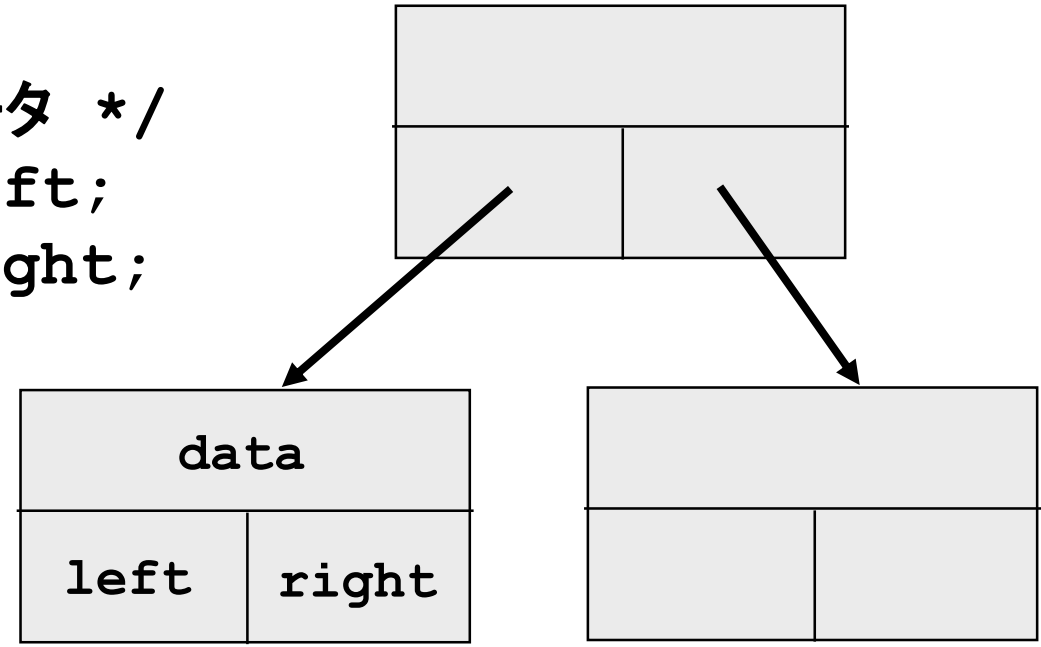
- 木構造Ｔとは
  - 空
  - Ｔの有限値の木構造（部分木）をもった型Ｔの接点
- ２分木
  - ２つの部分木を持つ接点



２分木のデータ構造

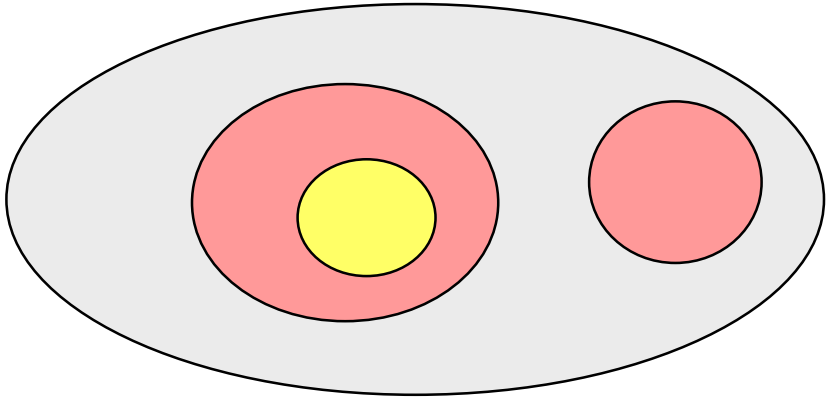
- ２つの部分木left, rightを持つ
  - 線形リストはひとつの部分木（？）を持つ構造

```
struct node {
        /* ノードに持つデータ */
        struct node *left;
        struct node *right;
};
```



木構造であらわされるもの

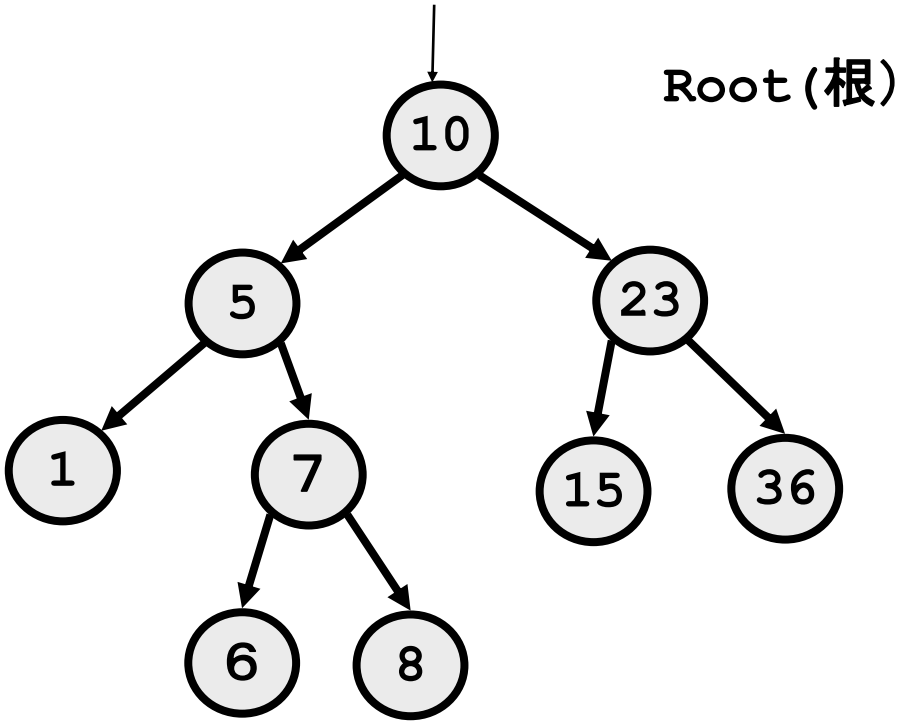
- 階層的な構造
  - 集合
  - 式
  - …



1+2*3+4

木構造を用いたソート

- 小さいものは左に、大きいものは右に



考え方（アルゴリズム）

- ①ノードのデータが、入力するデータよりも大きいならば、右の木を対象とする
- ②ノードのデータが、入力するデータよりも小さいきいならば、左の木を対象とする
- ③もしも、そこに木がなければ、ノードを割り当てて、挿入する

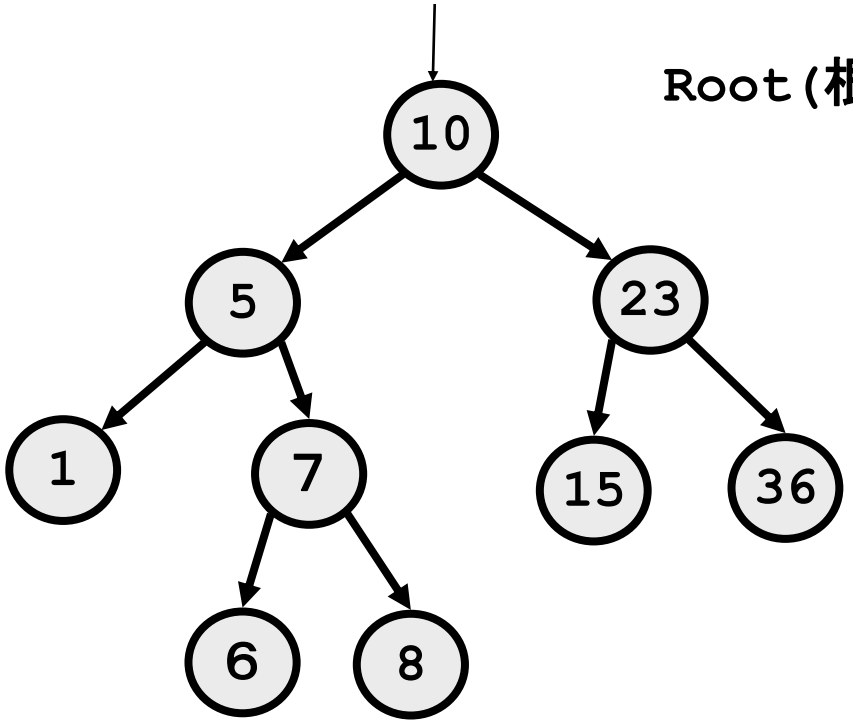
再帰的に考える！！！

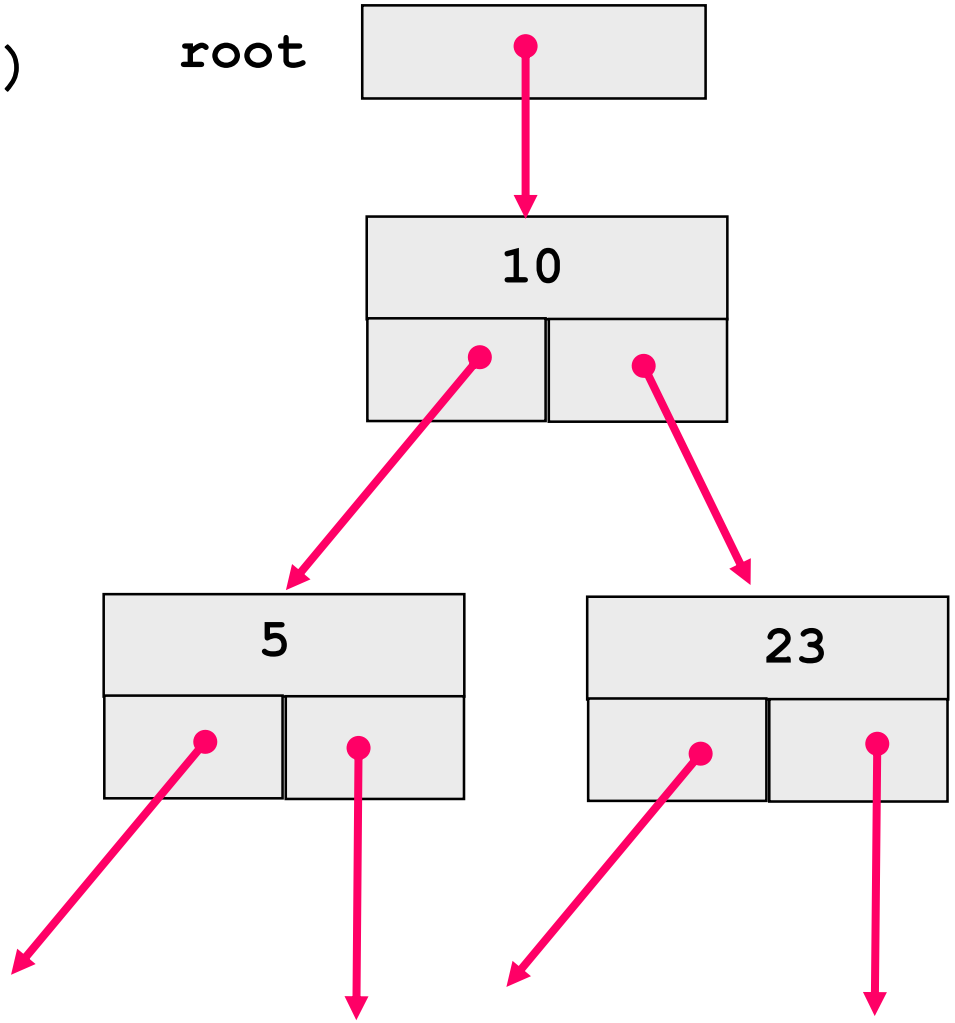
- 実際は、③を最初に行う。

２つのバージョン

- 挿入するところをポインターで渡す
  - ポインターのポインタをつかう
- 挿入したものをポインタで返す
- メインプログラムは、リストのときとほぼ、同じ

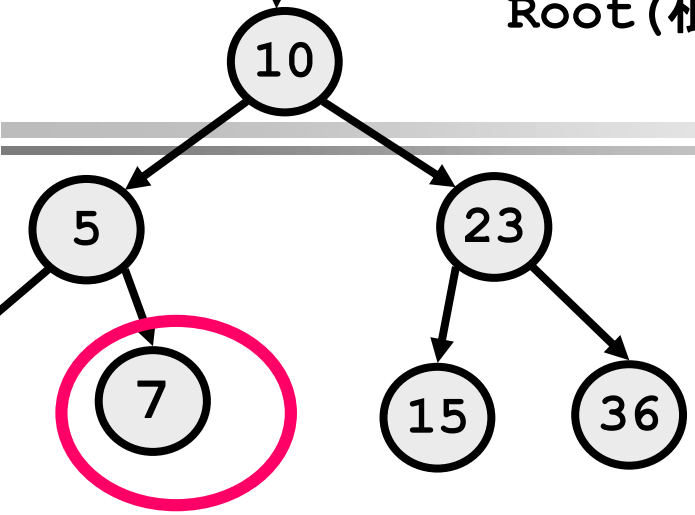
```
void insert_data(char *name, int x, struct node **pp)
{
  struct node *p, *t;
  p = *pp;
  if(p == NULL){
    t = (struct node *) malloc(sizeof(struct node));
    if (t == NULL) {
      printf("Out of memory¥n");
      exit(1);
    }
    strcpy(t->name, name);
    t->point = x;
    *pp = t;
    return;
  }
  if(x <= p->point)
    insert_data(name,x,&p->left);
  else
    insert_data(name,x,&p->right);
}
```


Root(根)
10
5
23
1
7
15
36
6
8


root
10
5
23

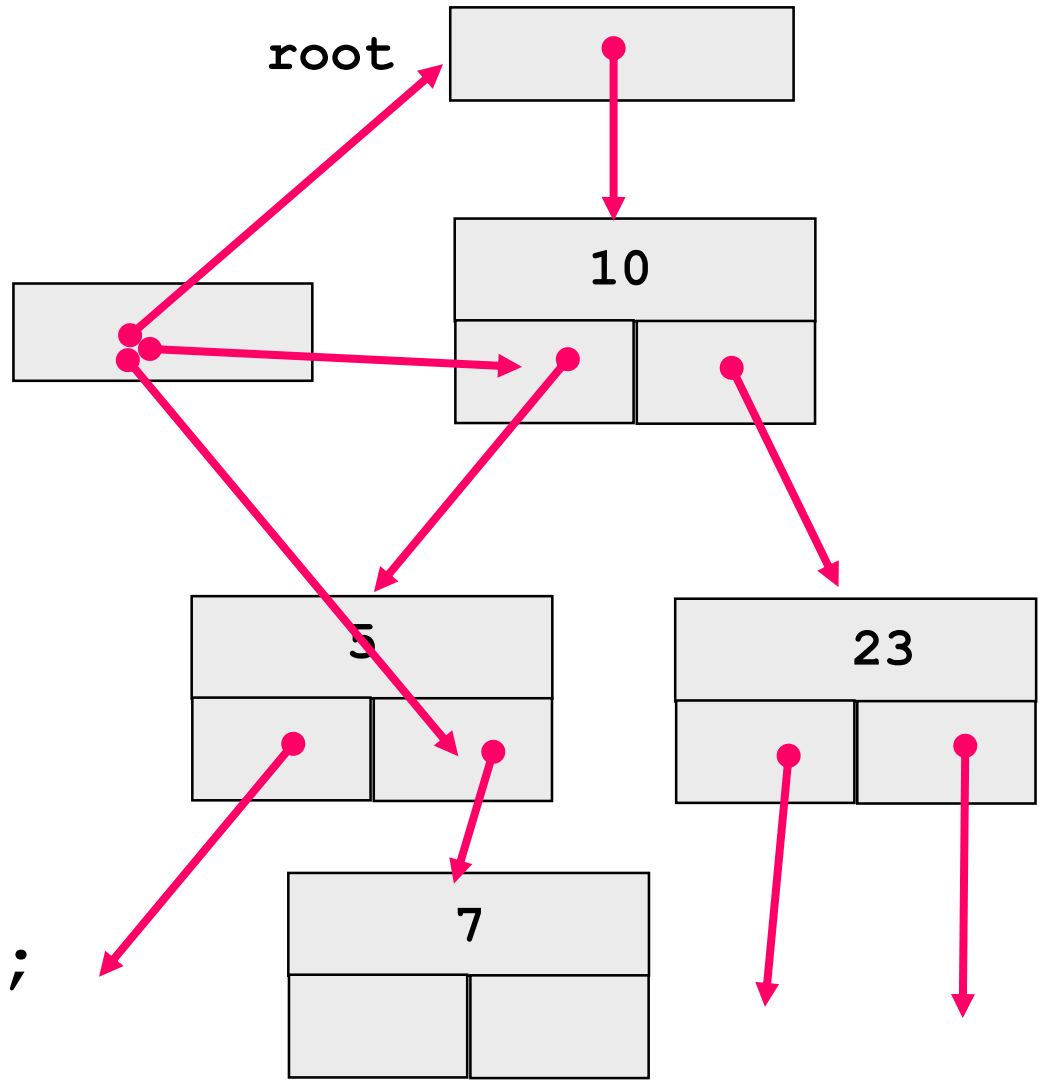
例 1

```
void insert_data(char *name
  int x, struct node **pp)
```



7を挿入する場合

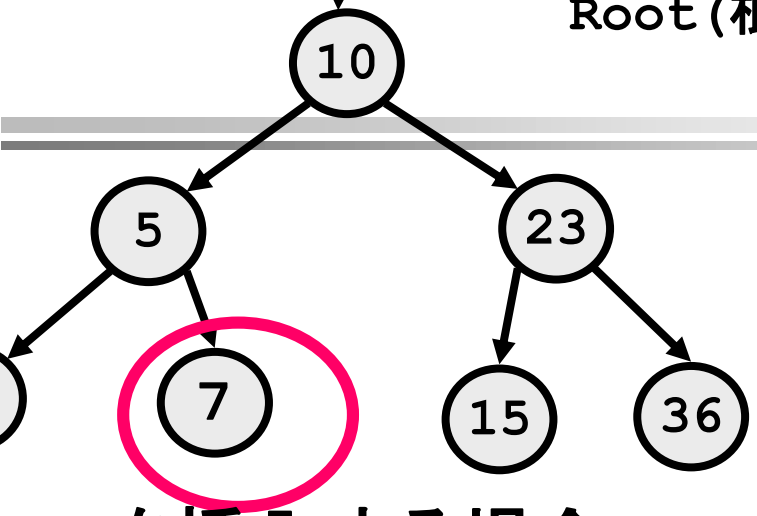
```
if(x <= p->point)
 insert_data(name,x,
       &p->left);
else
   insert_data(name,x,
       &p->right);
```




```
*pp = t;
```

```
struct node *insert_data(char *name, int x, struct node *p)
{
  if(p == NULL){
    p = (struct node *) malloc(sizeof(struct node));
    if (p == NULL) {
      printf("Out of memory¥n");
      exit(1);
    }
    strcpy(p->name, name);
    p->point = x;
    return p;
  }
  if(x <= p->point)
    p->left = insert_data(name,x,p->left);
  else
    p->right = insert_data(name,x,p->right);
  return p;
}
```

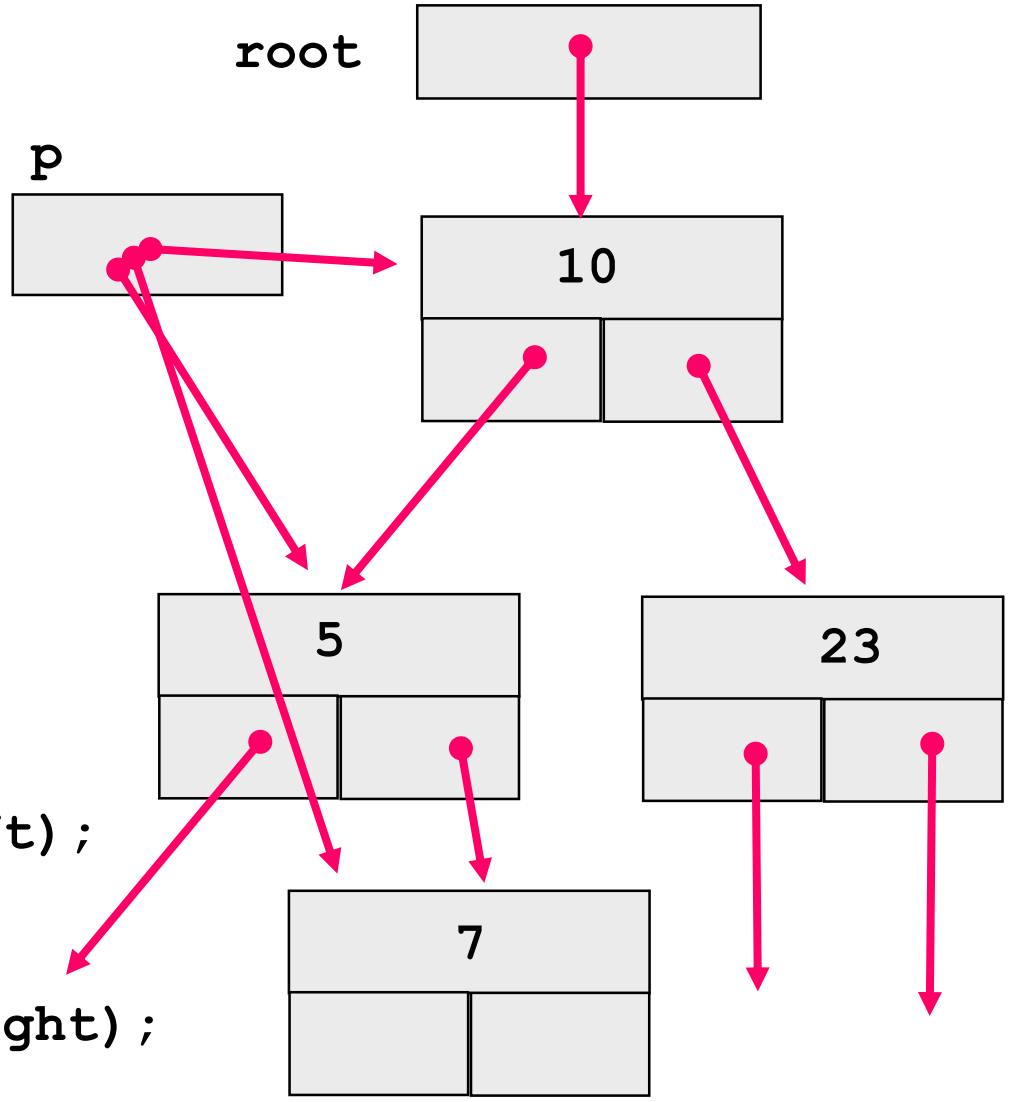
Root(根) その２ insert_data(…, root)



7を挿入する場合

```
struct node *insert_data(
  char *name,
  int x, struct node *p)


if(x <= p->point)
 p->left =
 insert_data(name,x,p->left);
else
 p->right =
  insert_data(name,x,p->right);
```



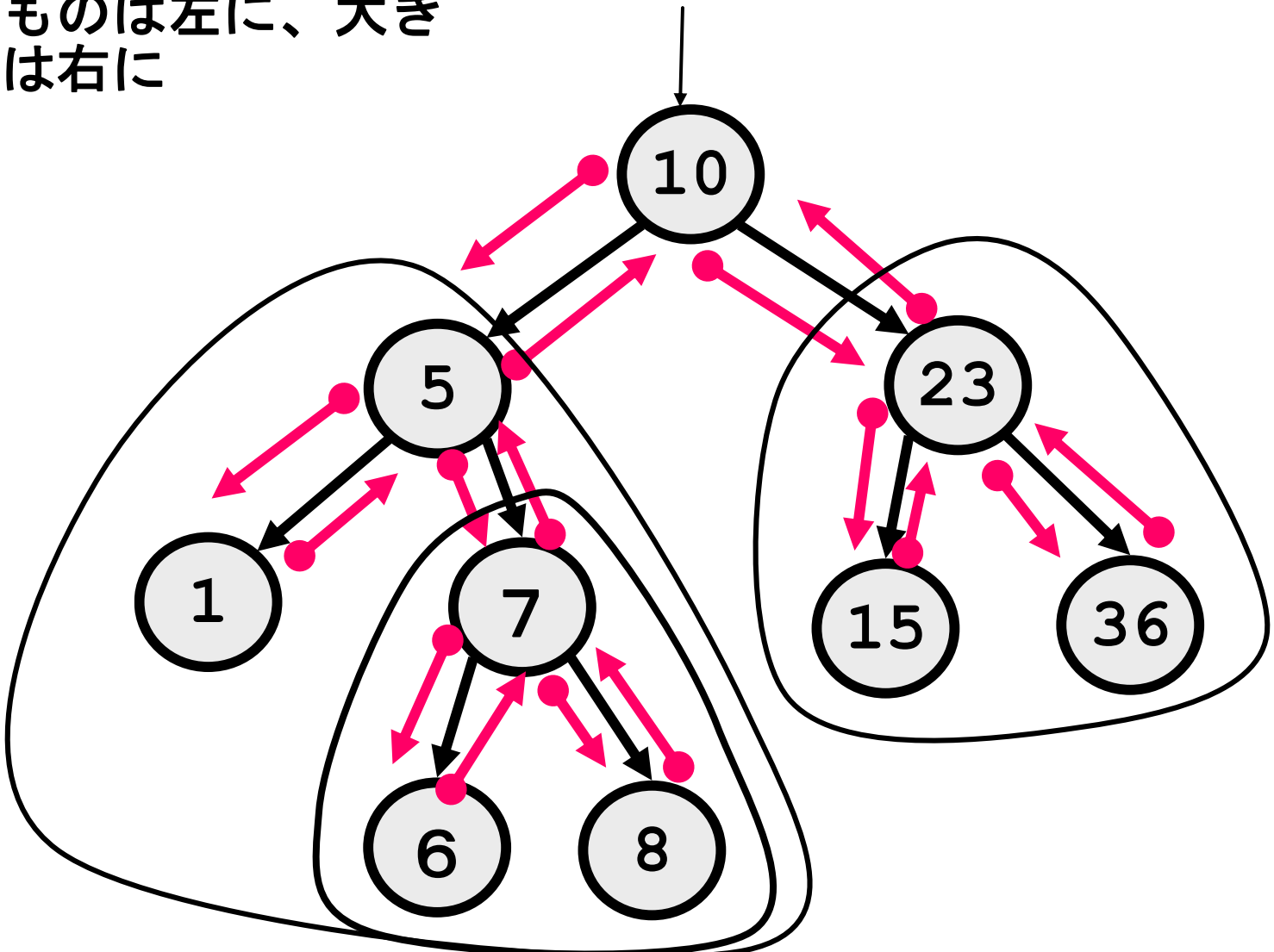
ソート結果の表示

- 左⇒現在のノード⇒右の順でたどればいい。

```
void printtree(struct node *p)
{
   if(p == NULL) return;
   printtree(p->left);
   printf("%s %d¥n", p->name, p->point);
   printtree(p->right);
}
```

木構造のトラバース

- 小さいものは左に、大きいものは右に



考えてみましょう

- あるデータを検索したい場合はどうすればいいのか？
- 線形リストに比べて、木構造の方が早い！なぜか？
- 木構造からデータを削除する場合はどうすればいいか？