システムプログラミング序論
第5回
構造体，動的なメモリ割り当て，リスト構造

大山恵弘

構造体

構造体（structure）

- 複数の要素を持つデータを表現するためのデータ型
  - 複数のデータをセットにして扱うのに役立つ
- 先に例を見よう

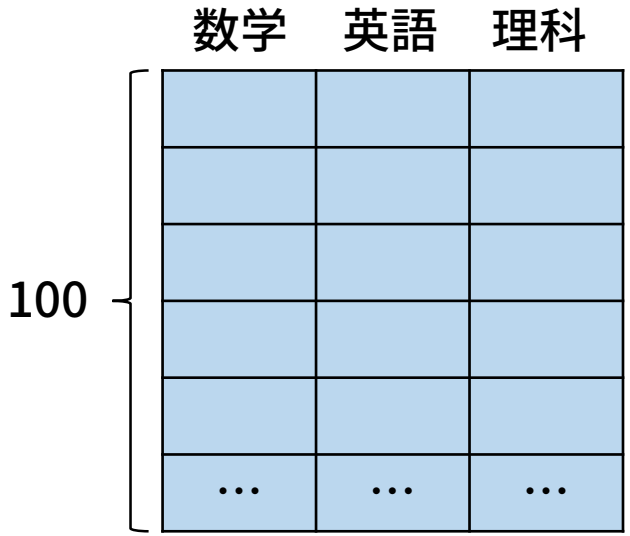
```
struct point {
  double x;
  double y;
  double z;
}
```

```
struct birthday {
  int year;
  int month;
  int day;
}
```

```
struct profile {
  char name[256];
  double height;
  double weight;
  int age;
}
```

例：成績表を集計するプログラムの作成

- 仮定
  - 数学，英語，理科の3科目
  - 学生は最大100人
- 解決
  - 人数を縦，点数を横にとる表を作る
  - 表を2次元配列で表す
    int score[100][3];
    - 1次元目が人，2次元目が科目（例：数学が0，英語が1，理科が2）
      - 例えば15番目の人の英語の成績は score[14][1]
- わかりやすいか？
  - 英語が1などの取り決めを忘れやすい



構造体を使うと

- 数学，英語，理科の成績をひとまとめにしたデータ型に名前をつけることができる

```
struct score {
  int math;
  int english;
  int science;
}
```

- これを使って，表を作る

```
struct score score_table[100];
```

構造体データ型（構造体型）の宣言

- 宣言

```
struct 構造体の名前 {
  データ型1 メンバ名1;
  データ型2 メンバ名2;
  …
}
```

- 構造体の要素をメンバ（またはフィールド）と言う
  - 任意のデータ型を使える

```
struct person {
  char name[64];
  char address[128];
  int tel[11];
}
```

構造体型の変数，メンバの参照

- 構造体型の変数の宣言

  `struct` 構造体の名前　変数名；

- メンバの参照

  構造体を表す式.メンバ名

- メンバへの代入

  構造体を表す式.メンバ名 = 式

例

```
int main(void)
{
  struct point { /*構造体の型の宣言 */
    double x;
    double y;
  };
  struct point p; /* 構造体型の変数の宣言 */
  double t;
  …
  t = p.x; /* メンバの参照 */
  p.x = 123.4; /* メンバへの代入 */
  …
}
```

融合して

```
struct point {
  double x;
  double y;
} p;
```

とも書ける

構造体の配列

- 宣言
  `struct` 構造体の名前 配列名`[`サイズ`];`
  - 例：7個の点の配列
    ```
    struct point bigdipper[7];
    ```
- メンバの参照
  - 例：3番目の点のx座標
    ```
    bigdipper[2].x
    ```
- メンバの代入
  - 例：7番目の点のy座標
    ```
    bigdipper[6].y = 2.38;
    ```

構造体そのものの代入

- 同じデータ型であれば，代入できる
  - 中のメンバ全部がコピーされる

```
struct point {
  double x;
  double y;
};
struct point a, b;
b.x = 1.0;
b.y = 2.0;
a = b; /* x and y are copied */
```

- 加減乗除などの演算はできない

```
c = a - b; /* wrong */
d = a++; /* wrong */
```

構造体の引数

- 関数は構造体を引数として受け取ることもできる
  - 中身がコピーされて渡される
  - 呼び出し側の構造体そのものが渡されるわけではない
  - これを値渡し（call by value）と言う

```
double distance(struct point a, struct point b)
{
  …
}

int main(void)
{
  struct point p1, p2;
  double d;
  …
  d = distance(p1, p2);
  …
}
```

構造体の返り値

- 関数は構造体を返り値として返すこともできる
  - やはり中身がコピーされて渡される

```
struct point midpoint(struct point a, struct point b)
{
  struct point mp;
  mp.x = (a.x + b.x) / 2.0;
  mp.y = (a.y + b.y) / 2.0;
  return mp;
}

int main(void)
{
  struct p1, p2, p3;
  …
  p3 = midpoint(p1, p2);
  …
}
```

構造体のサイズ

- `sizeof(...)`：引数に与えられたデータやデータ型のサイズ（単位はバイト）を返す演算子
  - 引数は型の名前でも変数の名前でも定数でも良い
  - サイズはコンパイラや OS や CPU に依存
  - 例：
    - `sizeof(char)` → 普通は1
    - `sizeof(int)` → 4とか8とか
    - `sizeof(double)` → 普通は8
    - `char a; … sizeof(a)` → 1
    - `int buf[100]; … sizeof(buf)` → 400とか800とか
    - `sizeof("Tsukubadaigaku")` → 15
  - 問題：何が表示されるか？

```
struct point {
  double x;
  double y;
};
printf("%d¥n", sizeof(struct point));
```

typedef 宣言

- データ型に自分の名前を付けることができる
  `typedef T データ型名;`
  - あるデータ型に対して適切な名前を付けておくと，プログラムがわかりやすく，書きやすく，保守しやすくなる

```
typedef int price_t;
price_t apple, orange, melon;

typedef char * string_t;
string_t s1, s2;
...
strcpy(s1, s2);

typedef double speed_t;
speed_t jet, car, bike;
```

おさらい

- 構造体
  - 複数のデータをひとまとめにしたデータである
  - 構造体内の個々のメンバを参照，更新できる
  - 構造体全体を変数に代入したり，関数の引数として与えたり，関数の返り値として返したりできる
- **`sizeof`**： データやデータ型のサイズを返す演算子
- **`typedef`**： データ型に自分の名前を付けるための指定子

構造体を引数に与えることの問題

- 構造体を引数に与えると，構造体はコピーされて関数に渡されるため，
  - 構造体が大きい場合に大量のデータコピーが発生する
  - 関数を呼ぶ側と呼ばれる側で構造体を共有できない
    - 「引数に与えた構造体のメンバを更新する関数」を書けない

```
struct drawing_figure {
  int x_beg[1000000];
  int x_end[1000000];
  int y_beg[1000000];
  int y_end[1000000];
};
```

```
void show_figure(struct drawing_figure fig)
{
  ...
}

int main(void)
{
  struct drawing_figure myfig;
  ...
  show_figure(myfig);
  ...
}
```

ポインタによる構造体の受け渡し

- 構造体ではなく，構造体を指すポインタ（構造体の場所の情報）を関数に渡す
  - その関数向けにコピーされるのはアドレスだけ
    - 高々4バイトや8バイト
  - 高速でメモリ消費量が少ないプログラムになる

重要

```
void show_figure2(struct drawing_figure *fig)
{
   ...
}

int main(void)
{
   struct drawing_figure myfig;
   ...
   show_figure2(&myfig);
   ...
}
```

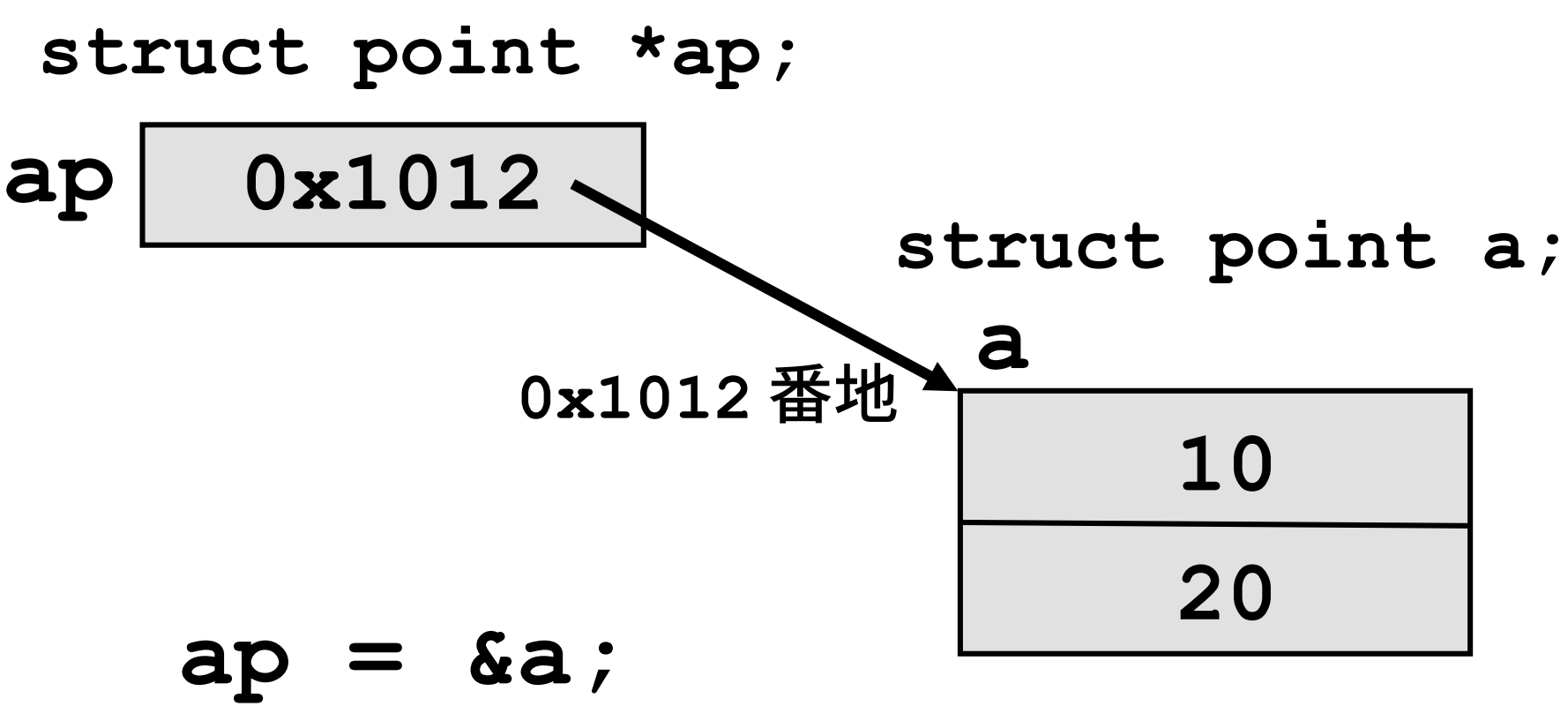
重要

構造体を指すポインタからのメンバの参照

- -> 演算子を使う

```
struct point {
  int x;
  int y;
} *ap; /* ap は構造体を指すポインタ */
...
t = ap->x + 1; /* メンバ x を参照 */
ap->x = 123; /* メンバ x へ代入 */
```

ap->x ⇔ (*ap).x


struct point *ap;
ap
0x1012
struct point a;
a
0x1012 番地
10
20
ap = &a;

動的なメモリ割り当て

実行中にメモリが欲しくなったら…

- メモリを確保する（「動的に」割り当てる）関数を呼び出す
  - `malloc(`確保するメモリのバイト数`)`
    - 確保したメモリ領域へのポインタが返り値として返される
    - 確保できなかった場合には `NULL` が返される
  - 使用例：
    - `p = malloc(4096);`
    - `q = malloc(sizeof(struct point));`
- 使うメモリ分の変数を常に関数内で定義できるならば，実行中にメモリを確保する必要はないが，そうはいかない
  - 使うメモリのサイズが，プログラム実行前には決まらず，実行中に決まる場合がある

型のキャスト

- ある型のデータを別の型のデータに変換すること
  - データの前に (型の名前) を付ける

```
int x = 1;
double y;
y = (double)x; /* int 型から double 型へのキャスト */
```

  - malloc と共に出てくることが多い
    - malloc の返り値の型は void *
      （任意の型のデータを指すポインタ）

```
q = (struct point *)malloc(sizeof(struct point));
```

メモリの解放

- **free** 関数を使う
  free(mallocでもらったポインタ**)**
  例えば
  free(p);
- 解放されたメモリ領域は OS やライブラリに返却され，後の **malloc** などで再利用される
  - メモリを解放せずに続けて確保し続けると，最終的には，他のプログラムや自身のプログラムで使えるメモリが枯渇する
  - 枯渇したら，**malloc** が **NULL** を返すようになったり，OS がいくつかのプログラムを強制終了させるなどの対策を実行する
- **malloc**，**free**を使う際には，**stdlib.h** を include する必要がある

malloc で確保したメモリはいつ解放されるのか？

- free を実行したとき
- プログラム（正確に言えばプロセス）が終了したとき
  - main 関数からの return，exit 関数の実行，不正なメモリのアクセスによる強制終了，KILL シグナルの受信，...
  - そのプログラムが確保した全メモリ領域が解放され，OS に返される

終了もせず，free も実行しないで malloc を繰り返し実行するプログラムでは，メモリリークの問題が生じる

```
void evil(void)
{
  while (1) {
    malloc(1000000);
  }
}
```

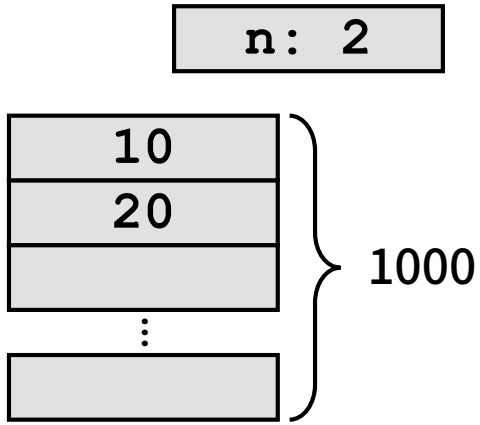
リスト構造

（線形）リスト構造

- 一列に並んだデータ群を表すデータ構造
  - 通常は，順序がついている並びを意味する
  - [1, 3, 5, 7] や ["Nougat", "Oreo", "Pie"] など
  - 必要に応じて，データを加えたり削除したりできる
- 典型的なインタフェース
  - 空のリストを作る
  - データ x をリストに加える
  - データ x をリストから削除する
  - データ x がリストに含まれているかどうか検査する
- どう実現するか？

配列によるリストの実現

リスト
[10, 20]

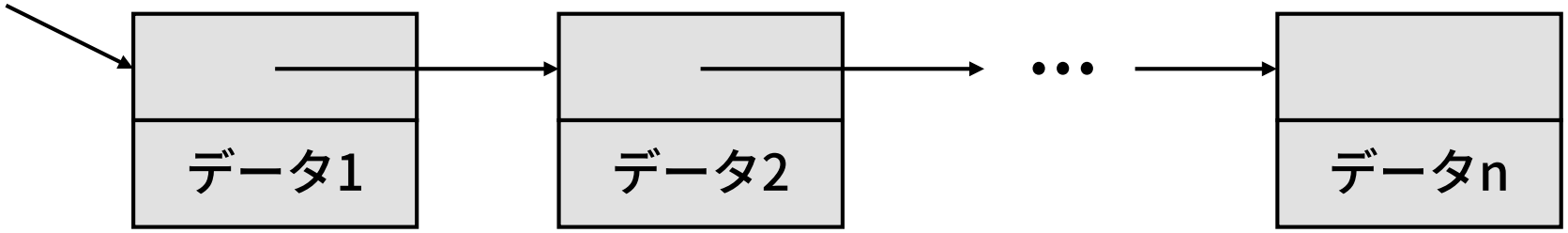


- 方法
  - 十分大きい配列を確保する
  - 配列のどの要素までデータがあるかを示す変数 n を作る
  - リストの末尾にデータを加える際には，末尾の次の空き配列要素にデータを入れ，n に1を足す
  - リストの末尾のデータを削除する場合には，n から1を引く
- 懸念
  - リストの中間や先頭にデータを加えるにはどうする？
  - リストの中間や先頭からデータを削除するにはどうする？
  - リストに多数のデータを加えた結果，配列の要素（配列のサイズ）が足りなくなったらどうする？

配列による実現は，それほど簡単ではない！

セルをポインタでつないだデータ構造によるリストの実現

- セル： リストを構成する各データを格納するための小さいデータ構造



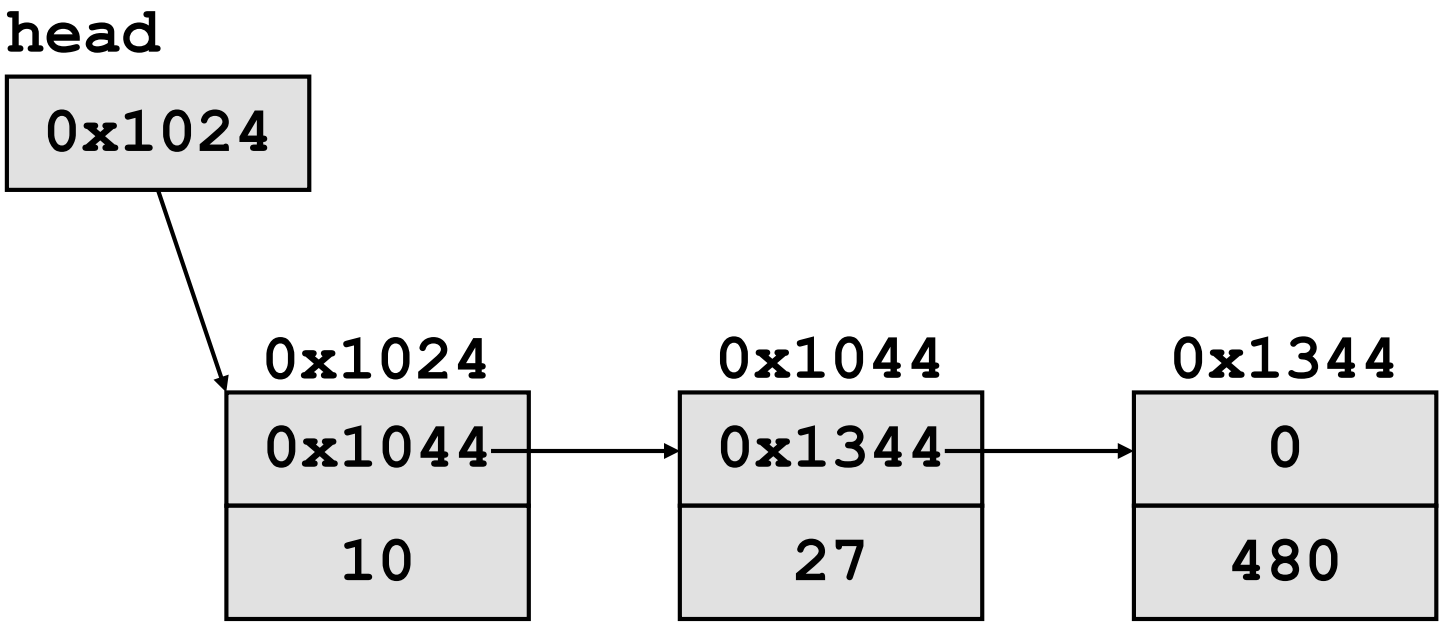

```
struct List {
   struct List *next;
   int data;
};
```

自分のデータ型のデータを参照するポインタ

リストに何を入れるかによって色々なものがここに来る

作られるデータ構造の例

```
/* リストの先頭を保持する大域変数 */
struct List *head;
```



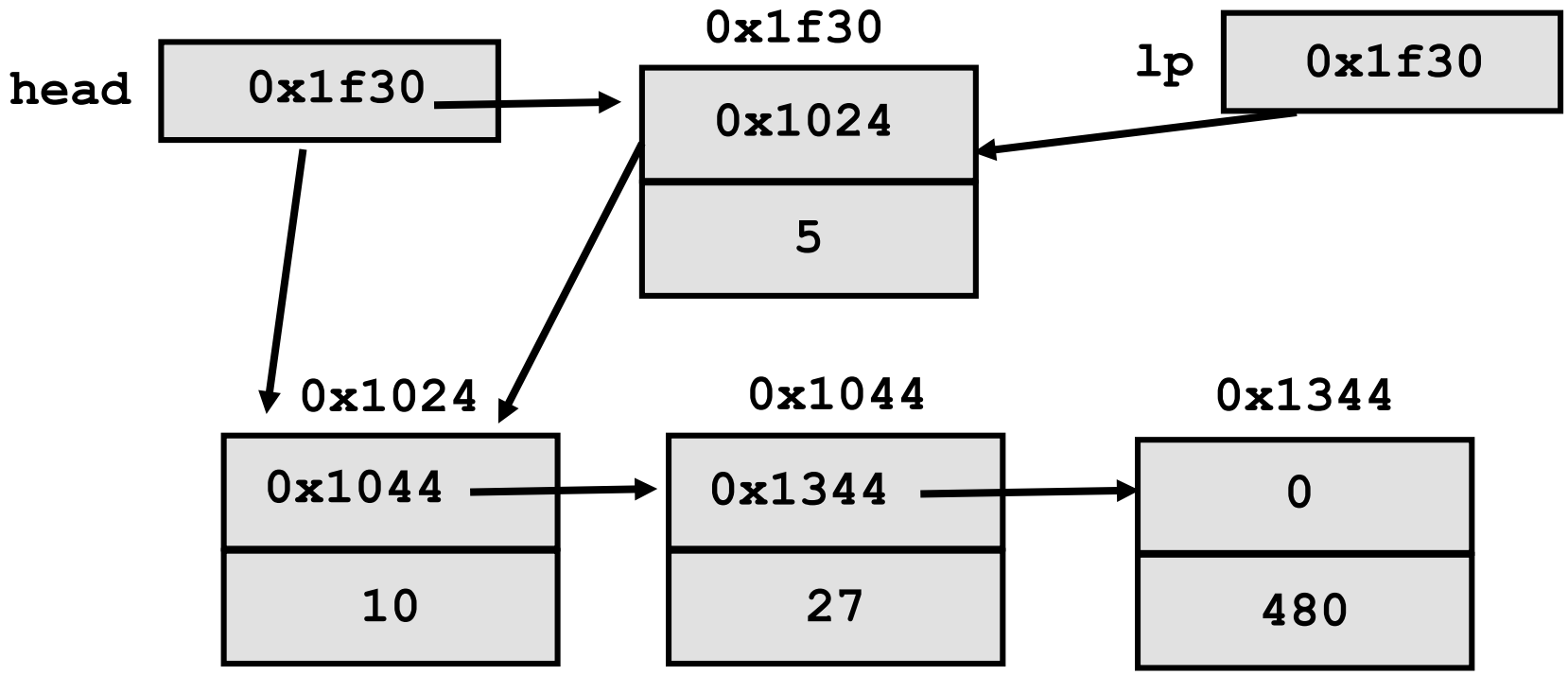
リストへのデータの追加

- head が持つリストに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) {
    fprintf(stderr, "No more memory¥n");
    exit(1);
  }
  lp->next = head;
  lp->data = x;
  head = lp;
}
```

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



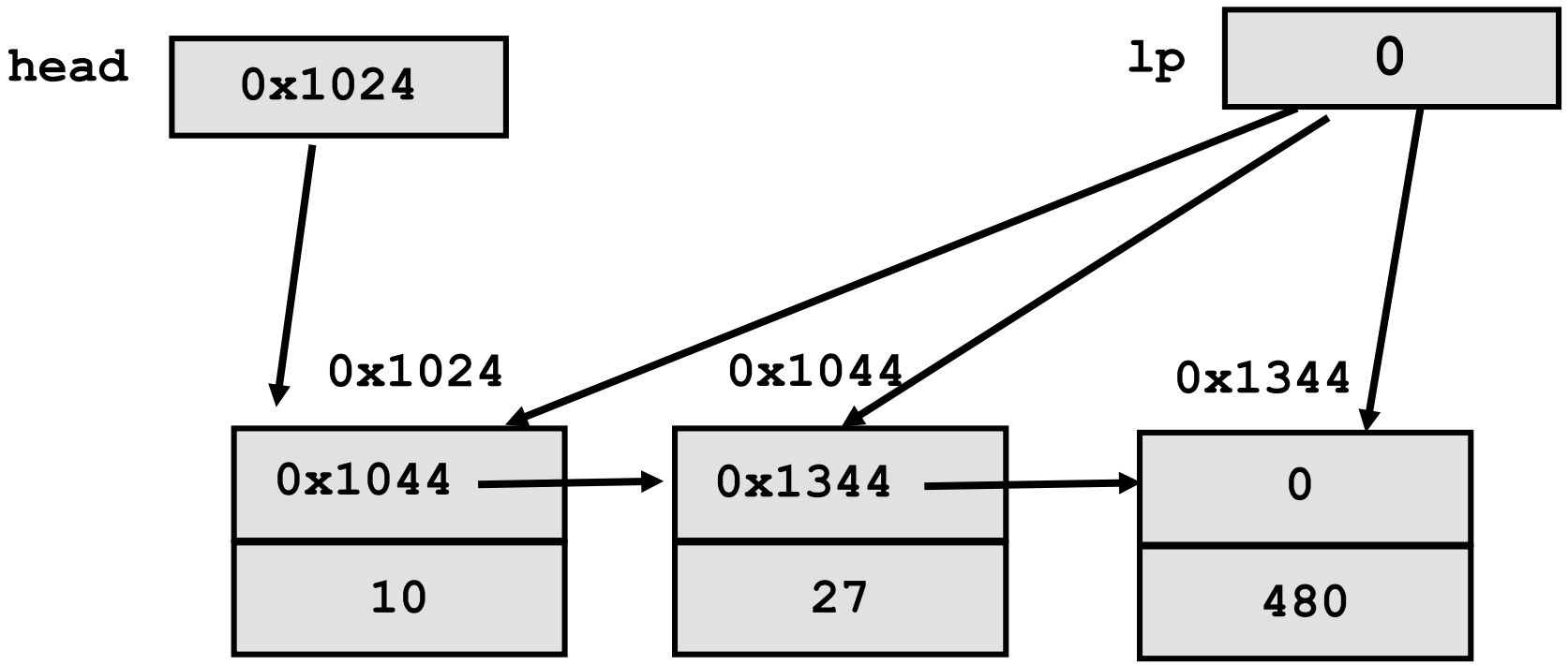

```
lp->next = head; lp->data = x;    head = lp;
```

リスト内のデータの検索

- head が持つリストに引数のデータがあれば1，なければ0を返す関数 existList

```
int existList(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;
```



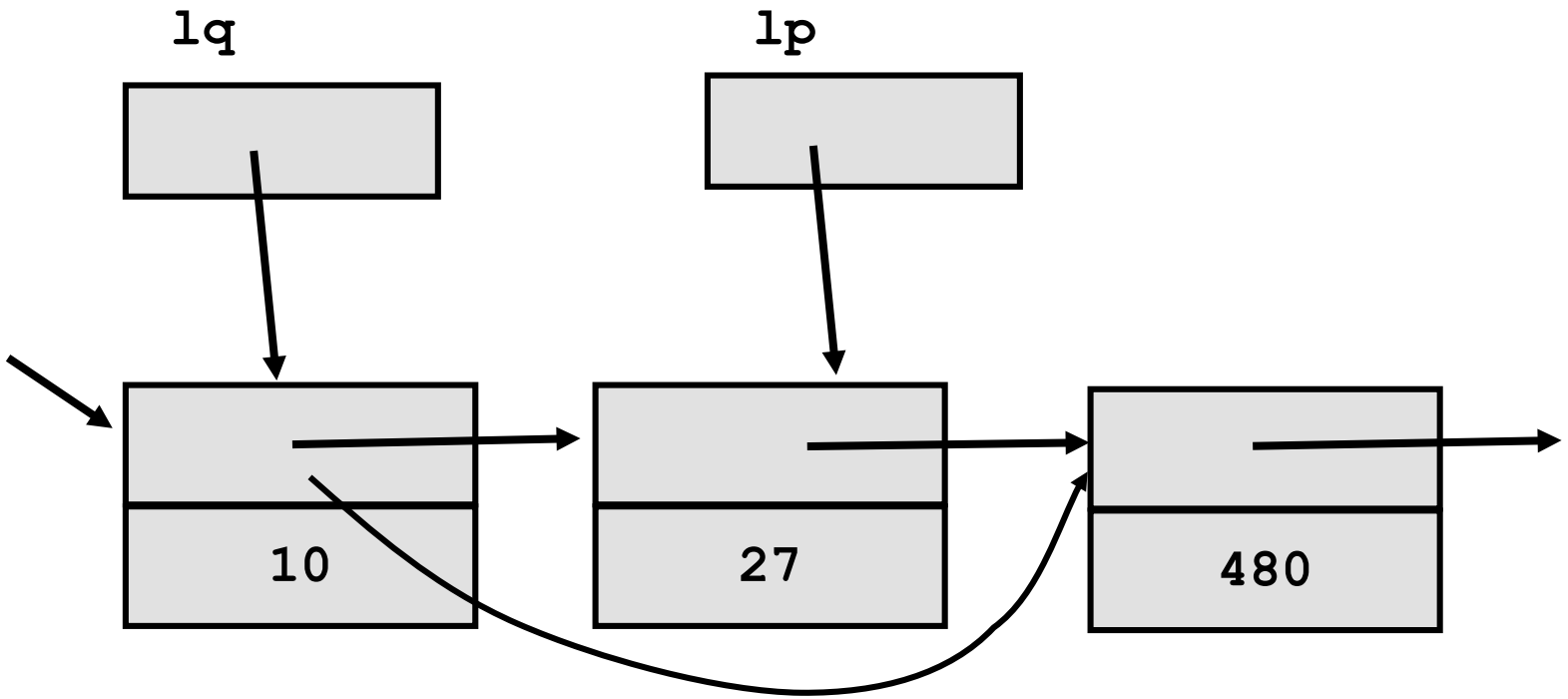

```
for (lp = head; lp != NULL; lp = lp->next) {
```

リストからのデータの削除

- **head** が持つリストに引数のデータがあれば削除する **removeList**

```
void removeList(int x)
{
  struct List *lp, *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;
```

練習問題

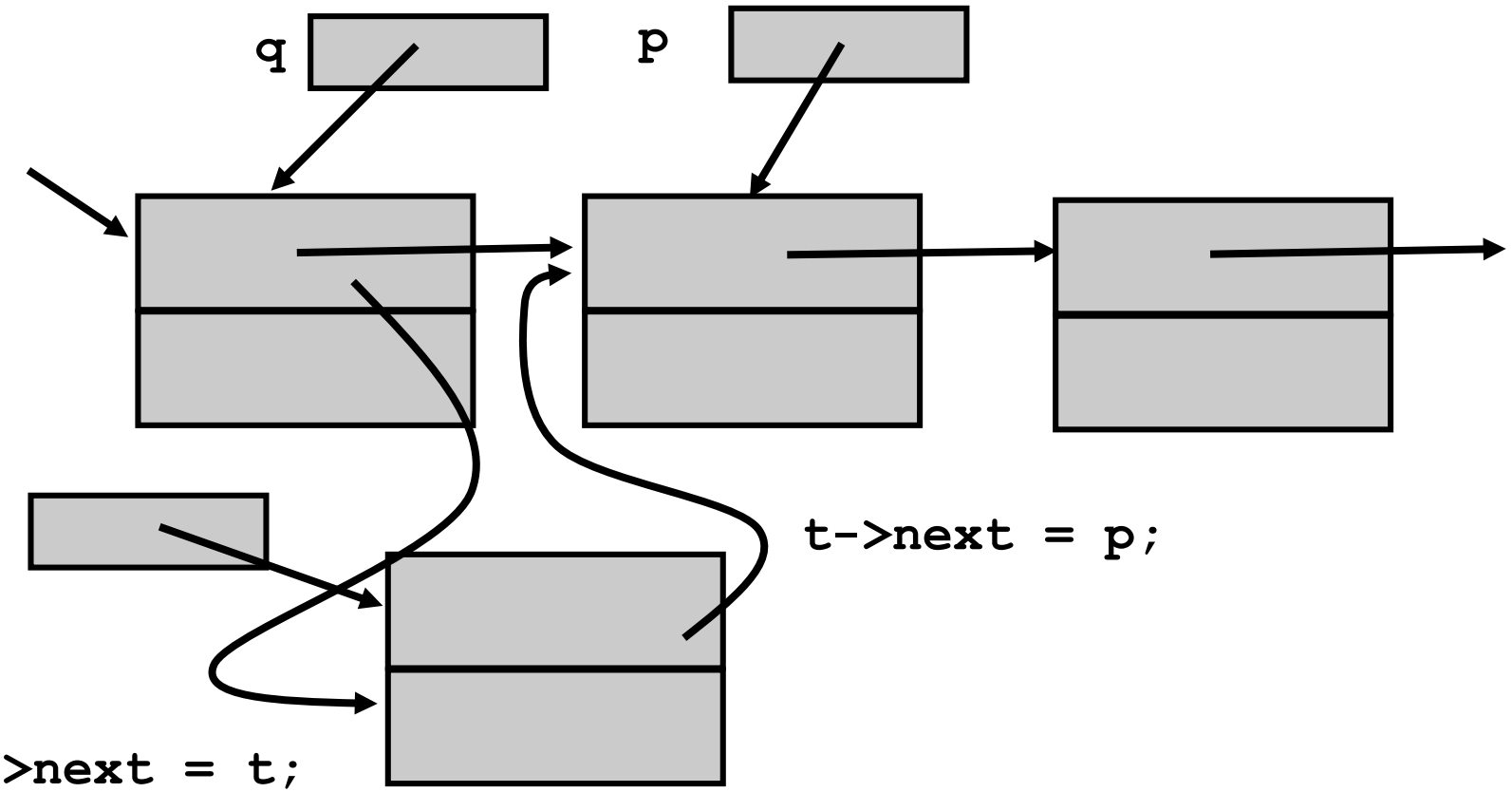
- リスト内に整数が昇順に並んでいる
- 昇順を維持するように，このリストの途中に整数を挿入する方法を考えなさい

```
void insert_list(int no, char *name, int x)
{
  struct record *p, *q, *t;
  t = (struct record *)malloc(sizeof(struct record));
  if (t == NULL) {
    fprintf(stderr, "Out of memory¥n");
    exit(1);
  }
  t->no = no;
  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;
}
```

```
struct record {
  int no;
  char name[512];
  int point;
  struct record *next;
};
struct record *head = NULL;
```

リストへの挿入

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



双方向リスト
Doubly Linked List

- セルが，次のセルへのポインタだけなく，前のセルへのポインタも持っているリスト
  - 前のセルの向きへの操作も高速に行える

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