プログラミング第一同演習

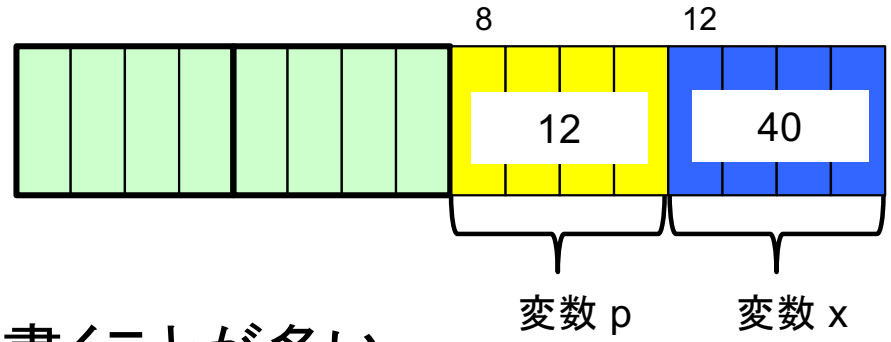
慶應義塾大学 理工学部 情報工学科
講義担当: 河野 健二
演習担当: 杉浦 裕太

本日の内容

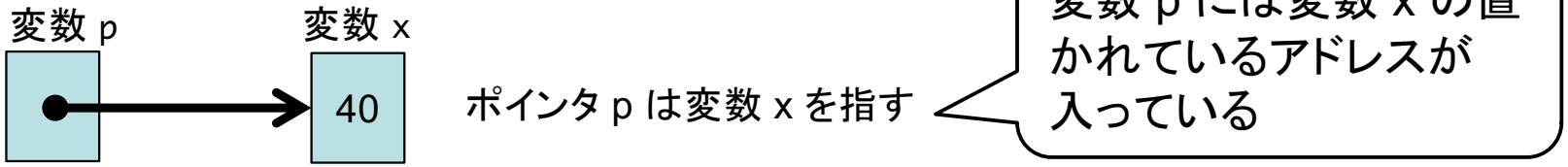
- 構造体と構造体へのポインタの利用例
  - 双方向リストについて学ぶ
  - これが理解できれば，構造体へのポインタは OK
  - 詳細は「アルゴリズム」の講義で学ぶ
    - 他にも構造体とポインタを利用した
      様々なデータ構造とアルゴリズムを学ぶ

矢印によるポインタの表現

- 次のプログラムを実行すると，メモリの様子は下図のようになる
  - `int x = 40, *p;`
    `p = &x;`
  - 変数 p に入っている値はアドレスで，そのアドレスには変数 x が置かれている



- この様子を矢印で書くことが多い



簡単な例題を考えよう

sslab system software

- ウェブを使ったアンケートシステム
  - 例:
    犬好きと猫好きのどちらが多いか調べるアンケート
  - ウェブを使って次の情報を入力してもらう
    - 犬が好きか，猫が好きか (dog_or_cat)
    - 年齢 (age)
    - 性別 (sex)
  - アンケート期間が過ぎたらデータを集計する
    - 犬好き，猫好きのどっちが多い？
    - 男性に限るとどっちが多い？
    - 女性に限るとどっちが多い？
    - 20代ではどっちが多い？
      . . .

アンケート結果を覚えておく方法は？

- まずは，適切なデータ構造を考える
  - データ構造が不適切だと，あとあと，いろいろと困る
- １つのアンケート結果を格納する方法は？
  - 構造体を使うのが自然．次のように定義する

```
#define DOG       1
#define CAT       2

struct result {
    int dog_or_cat; // DOG なら犬，CAT なら猫
    int age;        // 年齢
    char sex;       // 性別．'M' なら男性，'F' なら女性
};
```

アンケート結果を覚えておく方法は？

- 多数のアンケート結果を覚えておくには？

（これまでに習った知識だと）

- 配列を使えばよさそう
  - アンケート結果の最大数を決めておく
  - C の配列は最初に要素数を決めておかなければならない

```
#define MAX_RESULTS         10000    // アンケートの最大件数

/* 構造体の配列を使う*/
struct result results[MAX_RESULTS];
```

配列は好ましくない

sslab system software

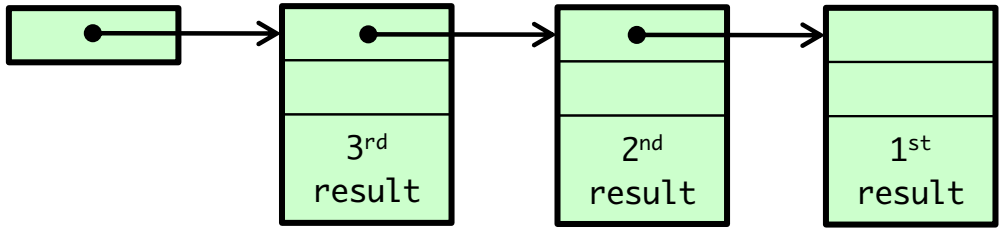
- `struct result results[MAX_RESULTS];`

- MAX_RESULTS より多くの結果を記録できない！
  - あらかじめ上限を決めておかなければならない
  - 上限が小さすぎると，アンケート結果を記録しきれない
  - 上限が大きすぎると，メモリの無駄遣い

- どうしたいか？
  - 上限を決めずに済むようにしたい
  - アンケート件数が少ない時は，あまりメモリを使わない
  - アンケート件数が増えると，徐々に使うメモリが増える

リスト構造

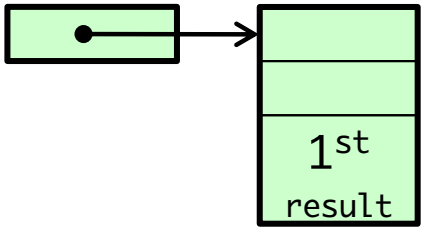
- 構造体のポインタを用いたデータ構造のひとつ
  - データ構造については「アルゴリズム」で詳しく学ぶ
- アンケート結果の構造体をポインタでつないでいく



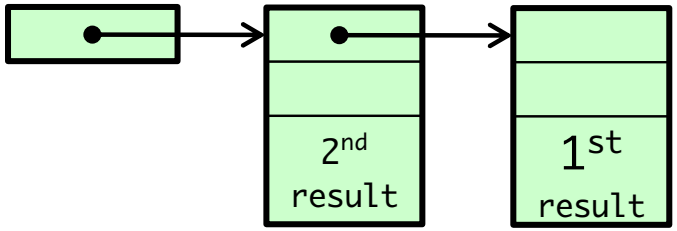
  - 新しい結果を記録するとき，新しく構造体を割り当てる
    - 前回，学んだ malloc を利用する
  - アンケート結果が増えると，割り当てられる構造体が増える

リスト構造のイメージ

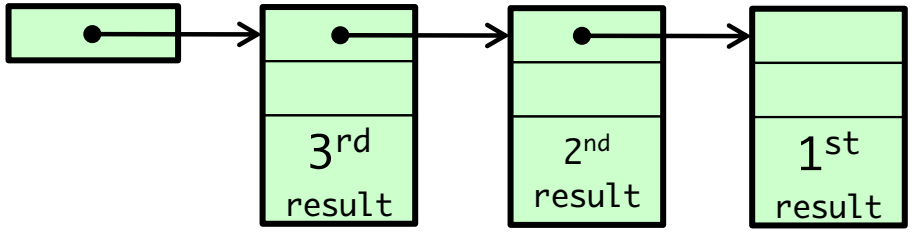
- 最初のアンケート結果が来たら，
  その結果を覚える構造体を malloc する
  - この構造体をリスト構造の “要素” とよぶ
  - “要素” をポインタでつないだものがリスト構造



- 次のアンケート結果が来たら，
  その結果を覚える構造体を malloc し，
  さらに前の構造体とポインタでつなぐ



- さらに次のアンケート結果が来たら，
  その結果を覚える構造体を malloc し，
  さらにポインタでつなぐ



リスト構造のための構造体

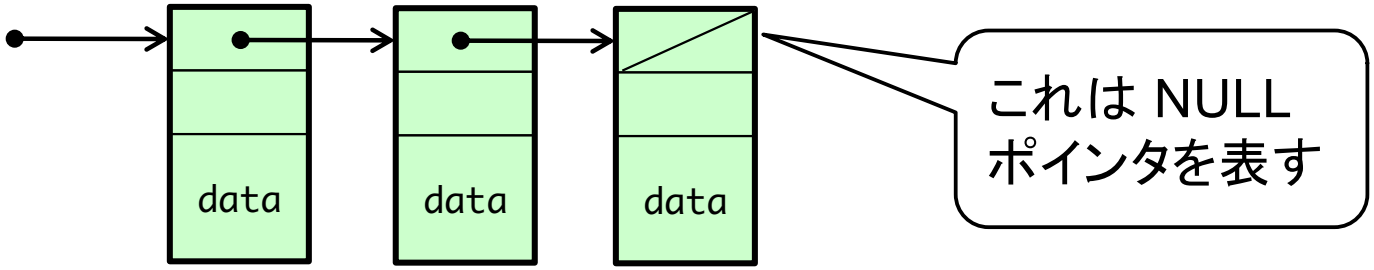
- “次の要素”を指すポインタを保持する

```
#define DOG         1
#define CAT         2

struct result {
    struct result *next;    // 次の要素を指すポインタ
    int dog_or_cat;         // DOG なら犬，CAT なら猫
    int age;                // 年齢
    char sex;               // 性別．M なら男性，F なら女性
};
```

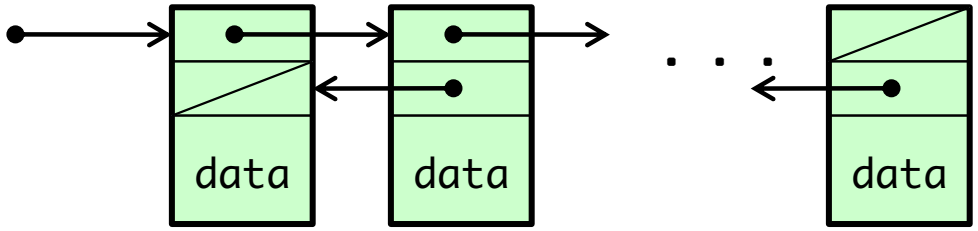
- struct result *next の意味
  - 構造体のメンバ next にはアドレスが入っている
  - そのアドレスには構造体 result がおかれている

2 種類のリスト構造

- 線形リスト (linear linked-list)
  - それぞれの要素が次の要素へのポインタだけを持つ



- 双方向リスト (double linked-list)
  - それぞれの要素が次の要素へのポインタに加え，
    ひとつ前の要素へのポインタを持つ



双方向リスト

sslab system software

- 双方向リストのほうがプログラミングが簡単
  - 線形リストのほうが単純に見える
  - 実際には，要素を削除する処理がトリッキーになる
- そこで，双方向リストを使ったサンプルを示す
  - 構造体へのポインタ
  - `malloc` / `free` の役割

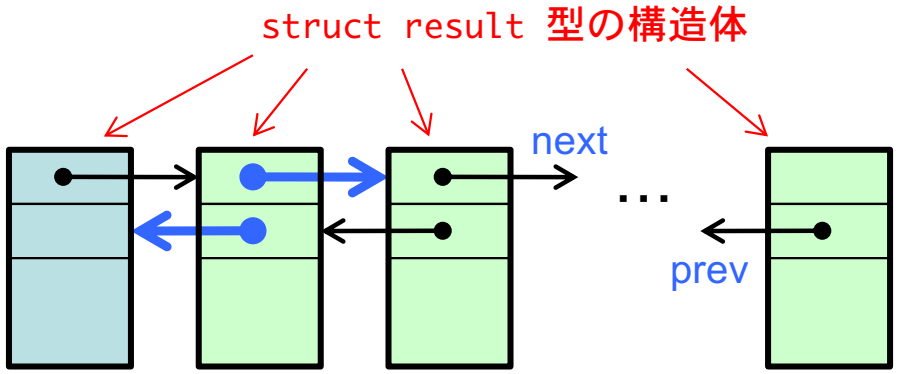
  をしっかりと理解すること

struct result の定義

・次の要素を指すポインタ next と、前の要素を指すポインタ prev をメンバーに加える
・ポインタの指す先には struct result 型の構造体がある.
　したがって, ポインタの型は struct result * となる

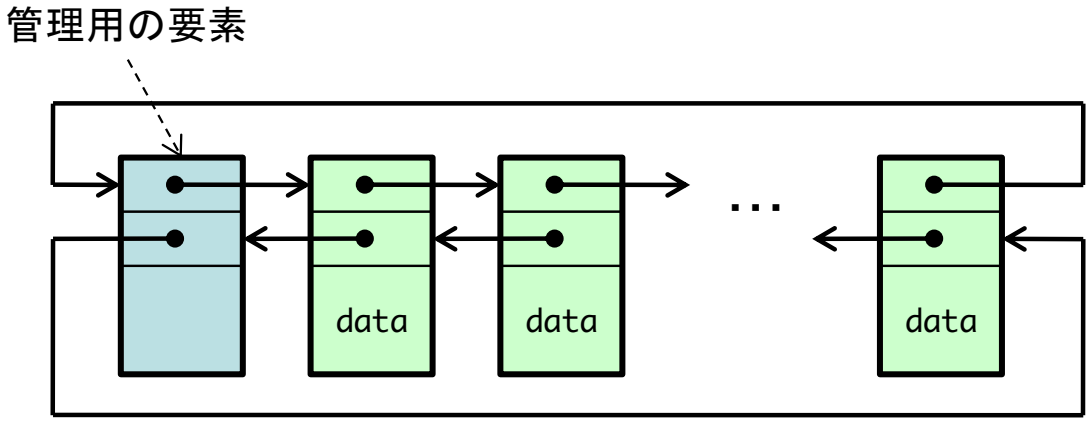
```
struct result {
    struct result *next, *prev;  // forward pointer & backward pointer
    int dog_or_cat; // DOG なら犬，CAT なら猫
    int age;        // 年齢
    char sex;       // 性別．M なら男性，F なら女性
};
```

ポインタの指す先は
struct result 型の構造体



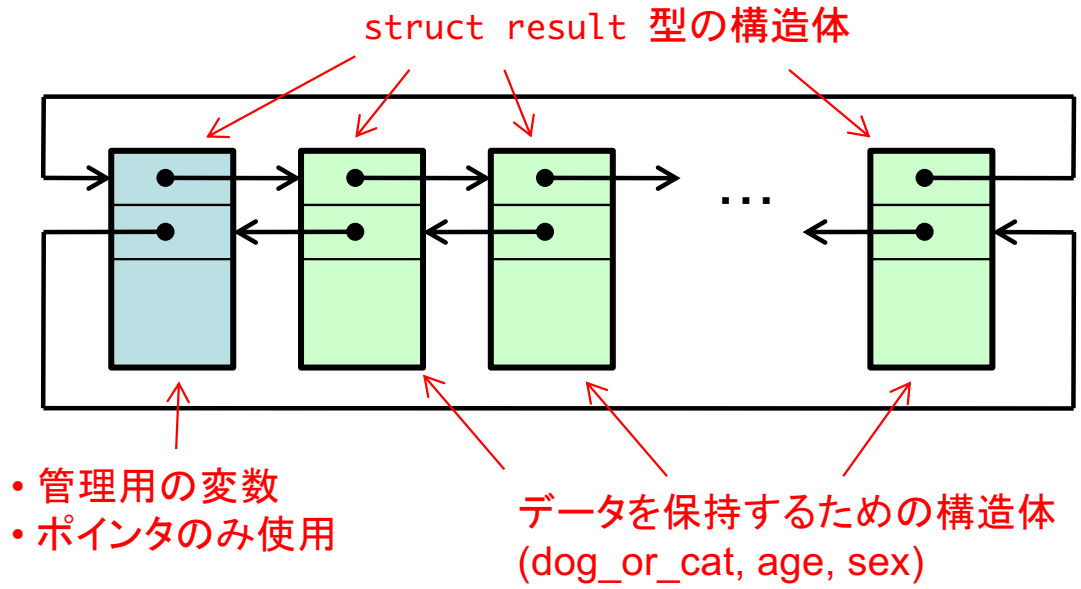
双方向リスト構造 (1)

- プログラミングを簡単にするため，次のようにする
  - リストの先頭は管理用の要素とする
    - アンケート結果は保持しない
    - 先頭以外の要素にアンケート結果を入れる
  - ポインタはぐるりと一周するようにしておく
    - 最後の要素の “次” の要素は管理用の要素
    - 管理用の要素の “ひとつ前” の要素は最後の要素



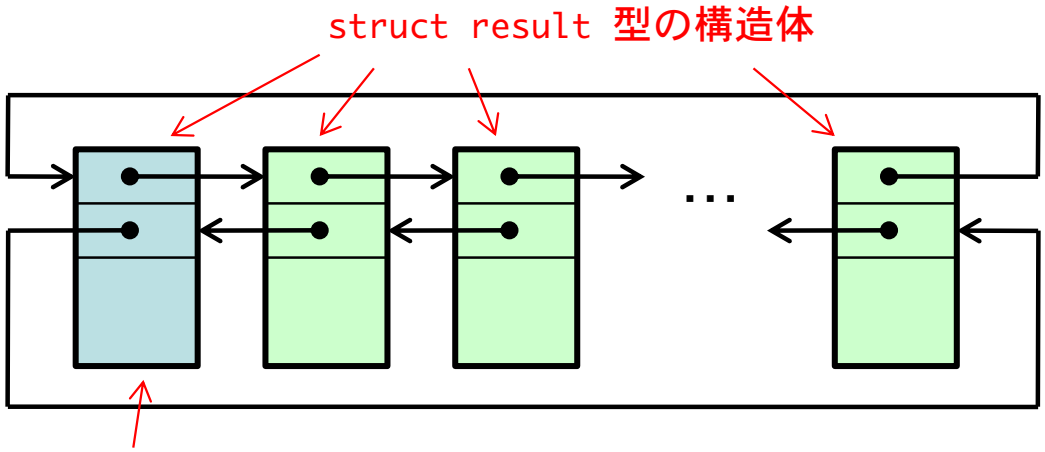
双方向リスト構造 (2)

- 管理用の要素
  - 他の要素と同じ型
  - 前後へのポインタのみを利用しその他のメンバは使用しない
- 例: `struct result` 型の双方向リストの場合



双方向リスト: 検索

- 双方向リストがすでにできあがっているとしよう
  - 要素の挿入を繰り返して，いくつもの要素が追加済

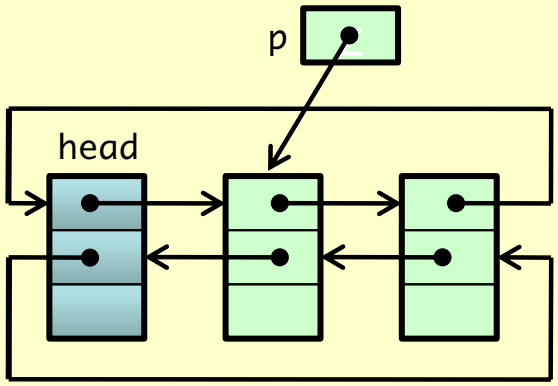


- このリスト構造の要素を順にたどってみよう
  - 例: アンケート結果から，
    犬好きと猫好きの人数を数える

双方向リストの探索 (1/8)

```
void count_dog_or_cat()
{
    int dog = 0, cat = 0;  // 犬好きと猫好きの人数．0 で初期化
    struct result *p;

    for (p = head.next; p != &head; p = p->next) {
        if (p->dog_or_cat == DOG) {
            dog++;
        } else if (p->dog_or_cat == CAT) {
            cat++;
        }
    }
}
```

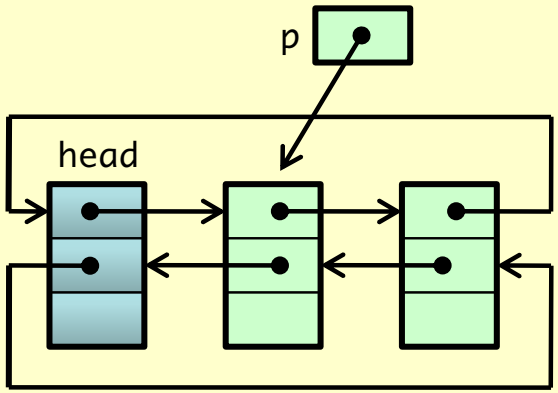
for 文中の p = head.next を実行したところ



双方向リストの探索 (2/8)

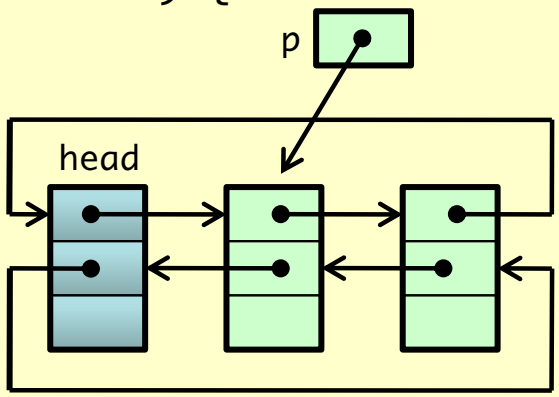
```
void count_dog_or_cat()
{
    int dog = 0, cat = 0;  // 犬好きと猫好きの人数．0 で初期化
    struct result *p;

    for (p = head.next; p != &head; p = p->next) {
        if (p->dog_or_cat == DOG) {
            dog++;
        } else if (p->dog_or_cat == CAT) {
            cat++;
        }
    }
}
```

p は head を指していない．
すなわち，p != &head が成り立つので，
for 文の本体を実行する



双方向リストの探索 (3/8)

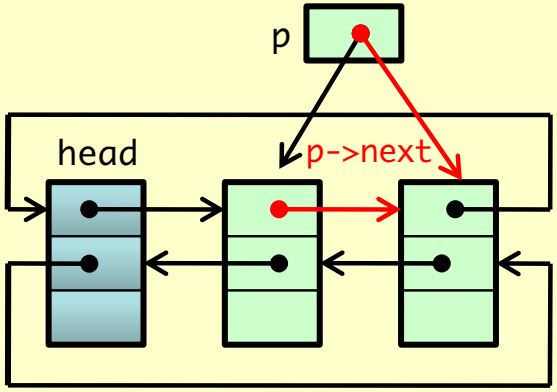
```
void count_dog_or_cat()
{
    int dog = 0, cat = 0;  // 犬好きと猫好きの人数．0 で初期化
    struct result *p;

    for (p = head.next; p != &head; p = p->next) {
        if (p->dog_or_cat == DOG) {
            dog++;
        } else if (p->dog_or_cat == CAT) {
            cat++;
        }
    }
}
```



p の指している構造体の
メンバ変数 dog_or_cat の値を
参照し，犬好き，猫好きの人数を
カウント

双方向リストの探索 (4/8)

```
void count_dog_or_cat()
{
    int dog = 0, cat = 0;  // 犬好きと猫好きの人数．0 で初期化
    struct result *p;

    for (p = head.next; p != &head; p = p->next) {
        if (p->dog_or_cat == DOG) {
            dog++;
        } else if (p->dog_or_cat == CAT) {
            cat++;
        }
    }
}
```



ポインタ p が次の要素を指すようにする．
p->next は次の要素を指す
すなわち，p->next の値は次の構造体の置かれたアドレスとなっている

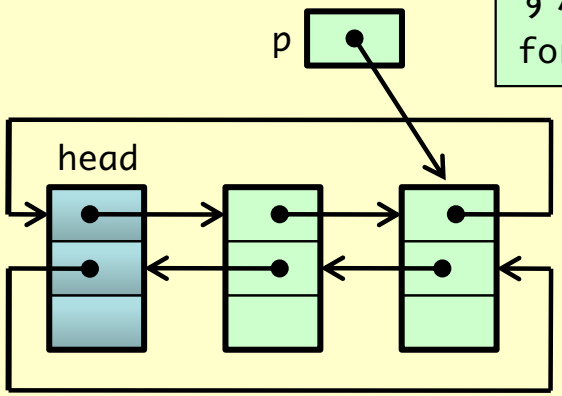
```
p = p->next
```

によって p->next の値を p に代入することで，p は次の要素を指す

双方向リストの探索 (5/8)

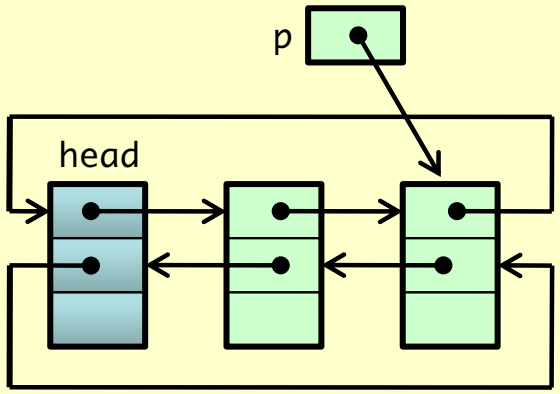
```
void count_dog_or_cat()
{
    int dog = 0, cat = 0;  // 犬好きと猫好きの人数．0 で初期化
    struct result *p;

    for (p = head.next; p != &head; p = p->next) {
        if (p->dog_or_cat == DOG) {
            dog++;
        } else if (p->dog_or_cat == CAT) {
            cat++;
        }
    }
}
```

p は head を指していない．
すなわち，p != &head が成り立つので，
for 文の本体を実行する



双方向リストの探索 (6/8)

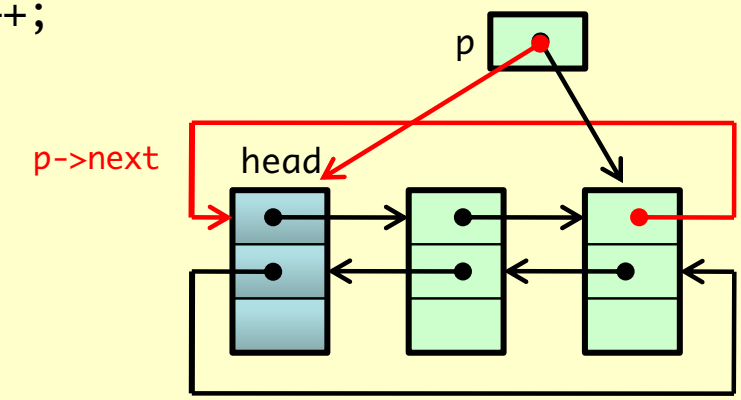
```
void count_dog_or_cat()
{
    int dog = 0, cat = 0;  // 犬好きと猫好きの人数．0 で初期化
    struct result *p;

    for (p = head.next; p != &head; p = p->next) {
        if (p->dog_or_cat == DOG) {
            dog++;
        } else if (p->dog_or_cat == CAT) {
            cat++;
        }
    }
}
```



p の指している構造体の
メンバ変数 dog_or_cat の値を
参照し，犬好き，猫好きの人数を
カウント

双方向リストの探索 (7/8)

```
void count_dog_or_cat()
{
    int dog = 0, cat = 0;  // 犬好きと猫好きの人数．0 で初期化
    struct result *p;

    for (p = head.next; p != &head; p = p->next) {
        if (p->dog_or_cat == DOG) {
            dog++;
        } else if (p->dog_or_cat == CAT) {
            cat++;
        }
    }
}
```



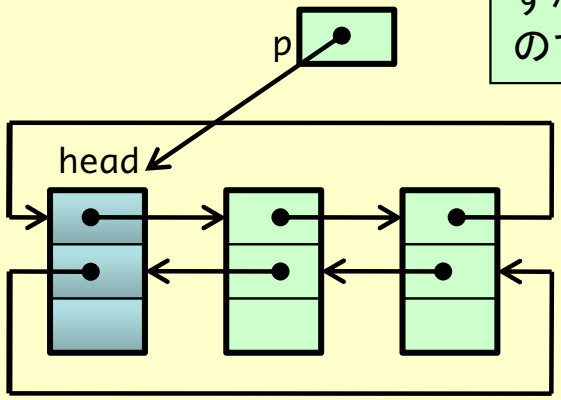
ポインタ p が次の要素を指すようにする．
p->next は次の要素を指す
すなわち，p->next の値は次の構造体の置かれたアドレスとなっている

```
p = p->next
```

によって p->next の値を p に代入することで，p は次の要素を指す

双方向リストの探索 (8/8)

```
void count_dog_or_cat()
{
    int dog = 0, cat = 0;  // 犬好きと猫好きの人数．0 で初期化
    struct result *p;

    for (p = head.next; p != &head; p = p->next) {
        if (p->dog_or_cat == DOG) {
            dog++;
        } else if (p->dog_or_cat == CAT) {
            cat++;
        }
    }
}
```

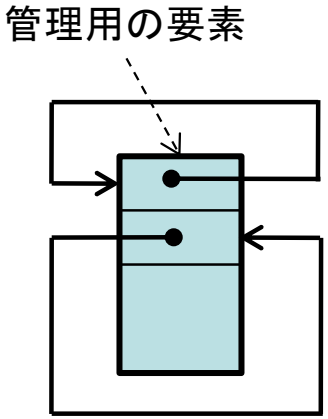


p が head を指している．
すなわち，p != &head が成り立たないので，for 文の実行を終了する

（管理用の構造体をのぞいて）
すべての要素を探索し終わった！

双方向リスト: 初期化

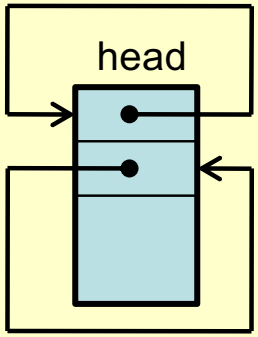
- 初期状態では，覚えているアンケート結果はない
  - リストは空っぽ
  - 管理用の構造体だけがある



» 管理用の要素だけがある
» 次の要素は自分自身
» ひとつ前の要素も自分自身

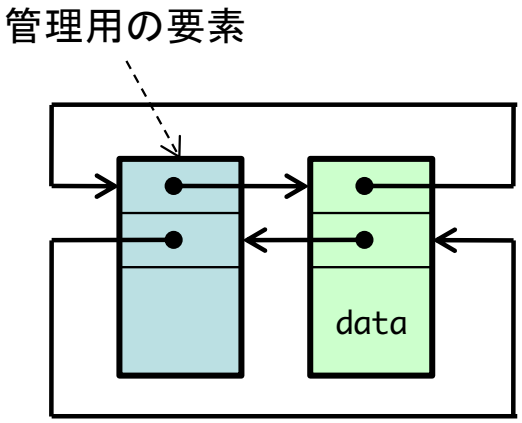
管理用の構造体と初期化

```
struct result {
    struct result *next, *prev;  // forward pointer & backward pointer
    int dog_or_cat;
    int age;
    char sex;
};

// 管理用の構造体．大域変数として定義
struct result head;

int main()
{
    // 初期化．右上の図の状態にする
    head.next = &head;  // ポインタ next の指す先を自分自身にする
    head.prev = &head;  // ポインタ prev の指す先を自分自身にする
    . . .
```



双方向リスト: 要素をひとつ追加

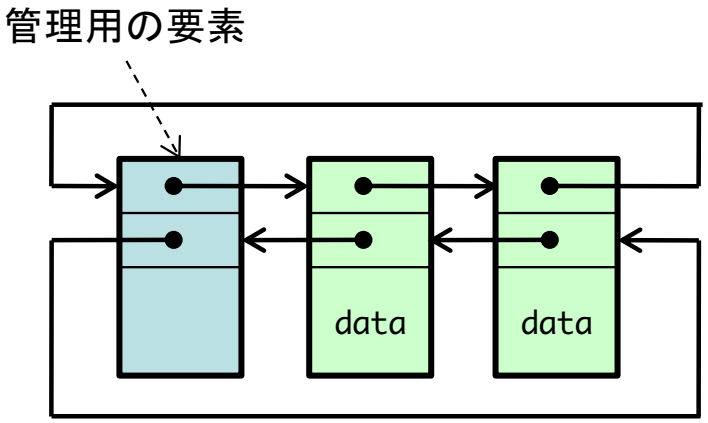
- 覚えているアンケート結果はひとつだけ
  - 結果を覚えるための“要素”がひとつ
  - それに加えて，管理用の構造体がある



  - 管理用の要素の次に新しい要素（構造体）を追加
  - ポインタがぐるりと一周するようにつなげてあることに注意

双方向リスト: 要素をさらに追加

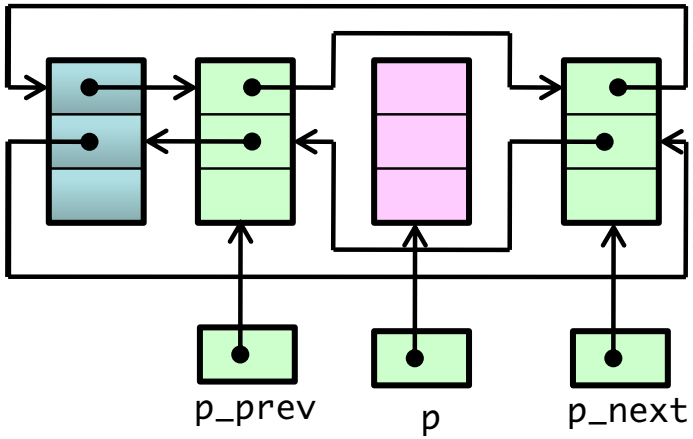
- 覚えているアンケート結果はふたつ
  - 結果を覚えるための“要素”がふたつ
  - それに加えて，管理用の構造体がある



  - 管理用の要素の次に要素（構造体）を追加
  - ポインタがぐるりと一周するようにつなげてあることに注意

双方向リストへの挿入 (1/7)

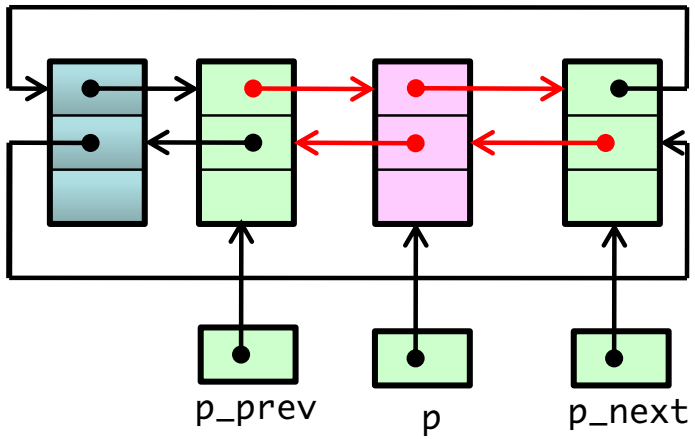
要素を挿入する前の状態



挿入したい
要素へのポインタ

p_prev の指す要素の次に
p の指す要素を新しく挿入する

要素を挿入した後の状態



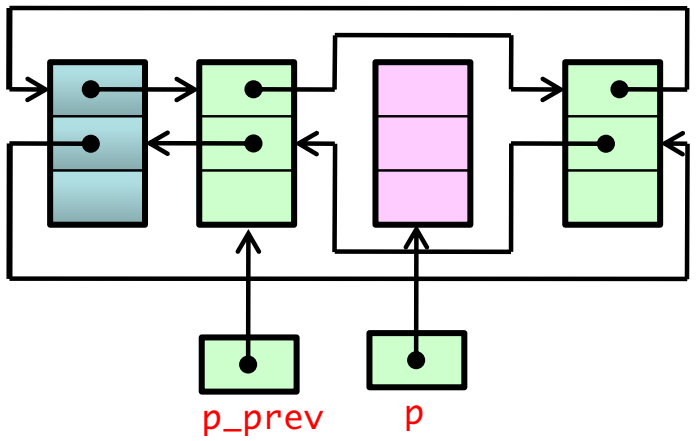
p の指す要素について:
  p のひとつ前の要素は p_prev
  p のひとつ先の要素は p_next
にする

p_prev のひとつ先の要素は p,
p_next のひとつ前の要素は p,
にする

双方向リストへの挿入 (2/7)

```
// p_prev の後に p を入れる
void insert(struct result *p_prev,
            struct result *p)
{
    struct result *p_next;
    p_next = p_prev->next;
    p->next = p_next;
    p->prev = p_prev;
    p_prev->next = p;
    p_next->prev = p;
}
```

insert を呼び出した時の様子

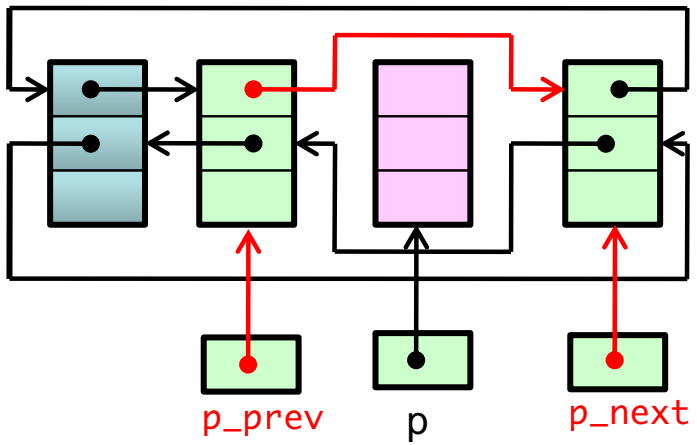


p_prev の指す要素の次に
p の指す要素を新しく挿入する

双方向リストへの挿入 (3/7)

```
// p_prev の後に p を入れる
void insert(struct result *p_prev,
            struct result *p)
{
    struct result *p_next;
    p_next = p_prev->next;
    p->next = p_next;
    p->prev = p_prev;
    p_prev->next = p;
    p_next->prev = p;
}
```

赤字の行を実行した直後の様子

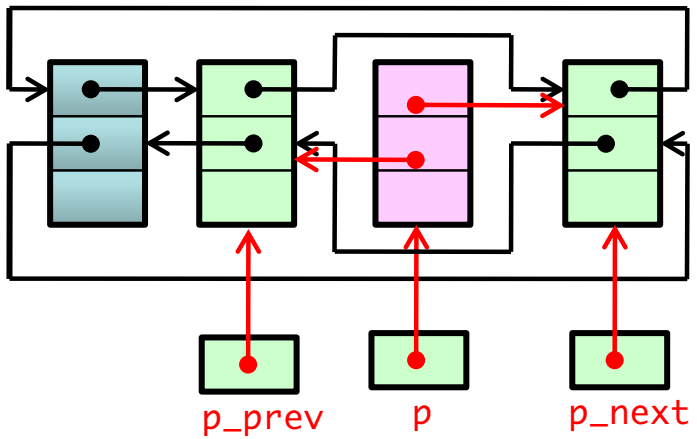


p_next は，p の次に来る要素．
現時点では，p_prev の指している要素の次の要素になっている

双方向リストへの挿入 (4/7)

```
// p_prev の後に p を入れる
void insert(struct result *p_prev,
            struct result *p)
{
    struct result *p_next;
    p_next = p_prev->next;
    p->next = p_next;
    p->prev = p_prev;
    p_prev->next = p;
    p_next->prev = p;
}
```

赤字の行を実行した直後の様子

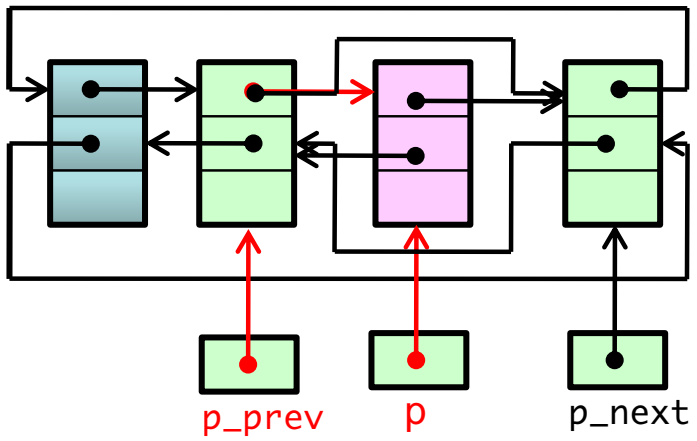


p のひとつ先の要素を p_next に，
p のひとつ前の要素を p_prev にしている

双方向リストへの挿入 (5/7)

```
// p_prev の後に p を入れる
void insert(struct result *p_prev,
            struct result *p)
{
    struct result *p_next;
    p_next = p_prev->next;
    p->next = p_next;
    p->prev = p_prev;
    p_prev->next = p;
    p_next->prev = p;
}
```

赤字の行を実行した直後の様子

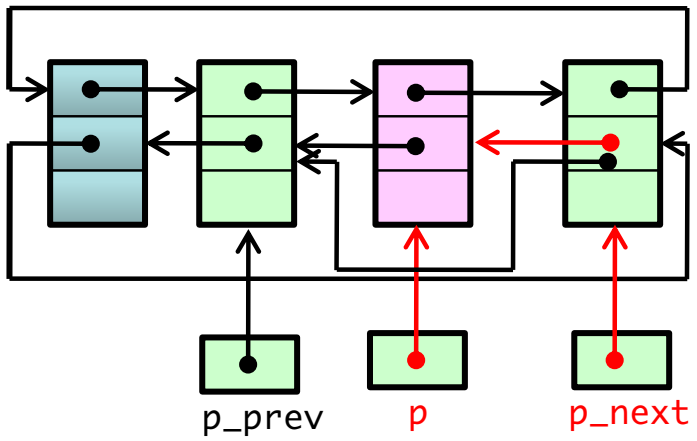


p_prev のひとつ先の要素を p にしている

双方向リストへの挿入 (6/7)

```
// p_prev の後に p を入れる
void insert(struct result *p_prev,
            struct result *p)
{
    struct result *p_next;
    p_next = p_prev->next;
    p->next = p_next;
    p->prev = p_prev;
    p_prev->next = p;
    p_next->prev = p;
}
```

赤字の行を実行した直後の様子



p_next のひとつ前の要素を p にしている

双方向リストへの挿入 (7/7)

sslab system software

- 新しい要素を作ってから insert を呼び出す

```
struct result *p;
...
// 新しく追加する構造体を malloc する
p = (struct result*)malloc(sizeof(struct result));
if (p == NULL) {
    エラー処理
}
// メンバ変数を初期化
p->dog_or_cat = DOG;
p->age = 30;
p->sex = 'M';

// 管理用の構造体の次に要素を追加
insert(&head, p);
...
```

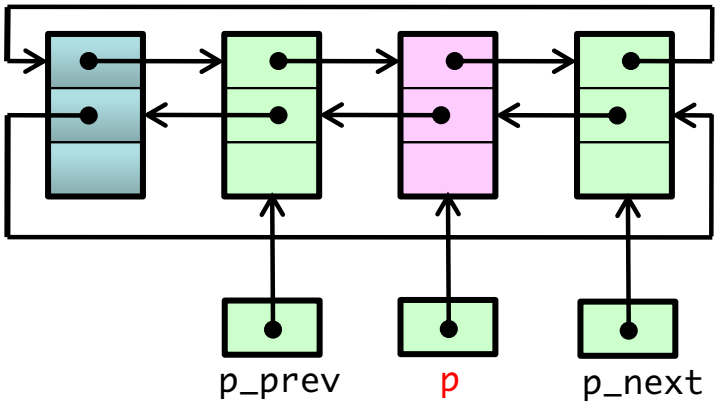
構造体を malloc する

構造体を初期化する

頭から順に追加

双方向リストからの削除 (1/3)

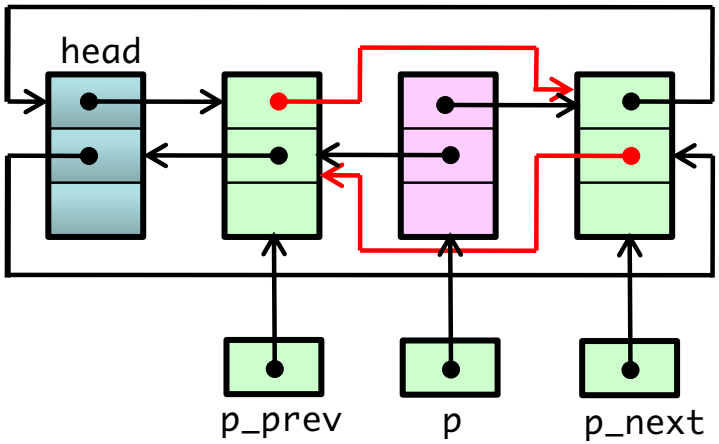
要素を削除する前の状態



削除したい
要素へのポインタ

p のひとつ前の要素を p_prev,
p のひとつ先の要素を p_next,
とする

要素を削除して終わった状態

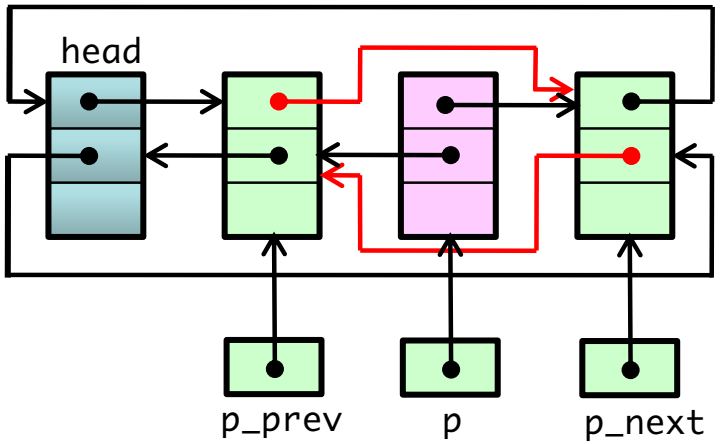


p_prev の次の要素を p_next,
p_next の前の要素を p_prev,
にすればよい

p の指している要素は head からたどれなくなっている．つまり，削除されている

双方向リストからの削除 (2/3)

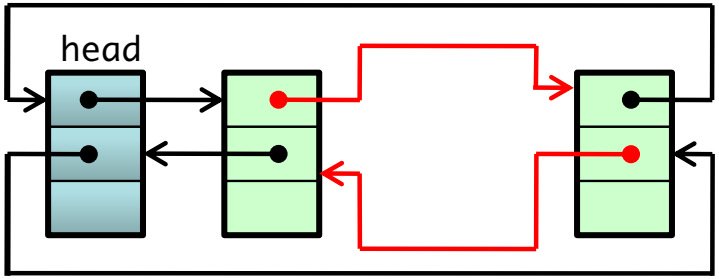
要素を削除して終わった状態



p の指している構造体はもう不要．よって，
free(p)
でメモリを解放する

free(p) をした後の状態

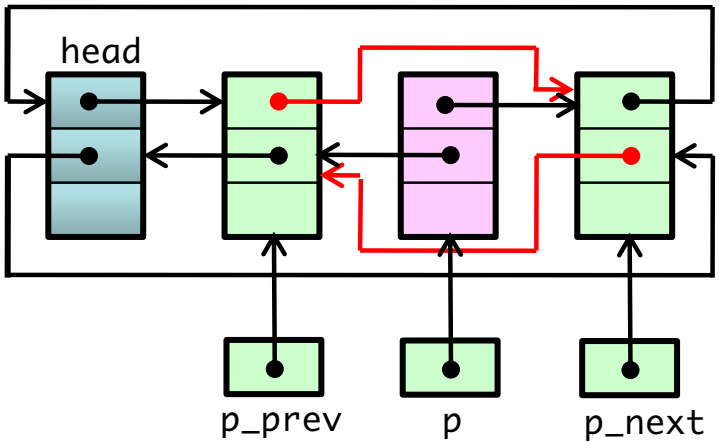
free(p) の後は p の指す先は存在しない
決して，参照しないこと



双方向リストからの削除 (3/3)

```
void remove(struct result *p)
{
    struct result *p_prev, *p_next;
    p_prev = p->prev;
    p_next = p->next;
    p_prev->next = p_next;  // (1)
    p_next->prev = p_prev;  // (2)
    free(p);
}
```

要素を削除して終わった状態



p_prev の次の要素を p_next,
p_next の前の要素を p_prev,
にすればよい

まとめ

sslab
system software

- 構造体と構造体へのポインタ応用編
  - リスト構造として，双方向リストを紹介
  - 双方向リストの探索
  - 双方向リストへの要素の挿入
  - 双方向リストからの要素の削除

- 背後にある理論や便利な使い方などは，
  「アルゴリズム」の講義で学んでください