プログラミング2　第6回

講義計画（暫定）

- ~~第1回(11/28(金))　ガイダンス、Cプログラミングの基礎（データ型、変数、四則演算、制御構文）~~
- ~~第2回(12/05(金))　ビット演算、配列、関数~~
- ~~第3回(12/12(金))　配列と関数、文字列、ソート~~
- ~~第4回(12/19(金))　再帰関数、ポインタ(1)~~
- ~~第5回(12/26(金))　ポインタ(2)~~
- 第6回(01/ 9(金))　構造体・共用体、リスト
- 第7回(01/23(金))　Cプログラミング実用上必須のもの（defineマクロ、ファイル操作、複数ソースコンパイルなど）
- 第8回(01/30(金))　期末試験（**出席必須！**）

期末試験について

- 必ず現地で参加してください
  - 当日やむを得ない理由で来られなくなった場合は連絡してください
- その場でプログラムを書いてもらいます
  - プログラムを実行できる環境を用意してください
- Web検索無し、講義資料の閲覧もなし
- これまでの演習で作ったプログラムは閲覧OK
  - 演習と似た問題を一問以上出します
- テストの満点は60点だが、50点を超えた分は切り捨てて、成績への反映は最大50点とする
  - 演習も同様に最大50点。足し合わせた点数が最終成績になる
    - ただし、成績(A+, A, B, C, C-）への変換は、標準的な換算方法**ではなく**、相対評価で行う
      - 一昨年度実績： A+: 100, A: 85-99, B: 70-84, C: 65-69, C-: 60-64
      - 昨年度実績：標準的な換算法と同じ

構造体（Structure）

- 構造体の定義

```
struct タグ名 {
  型　メンバ1;
  型　メンバ2;
...
  型　メンバN;
};
```

```
struct student {
  char name[40];
  int id;
  int grade;
};
```

- メンバの型が全て同じである必要性はない
- 配列や構造体もメンバにすることができる

- 構造体の変数宣言

```
struct タグ名 変数リスト;
```

```
struct student taro, hanako;
```

型のように使う

構造体の使い方

```
#include <stdio.h>

struct student {
  char name[40];
  int id;
  int grade;
};

int main(void)
{
  struct student taro, hanako;
  taro.grade = 1;
  …
}
```

構造体の定義

この段階では鋳型（テンプレート）を作っているのみ。まだメモリ上にstruct student型の変数は存在しない。

構造体型の変数宣言

変数宣言することで、構造体型変数が生成される。（ここではじめてメモリ上にstruct student型の変数が確保される。）

構造体の使い方

```c
#include <stdio.h>

struct student {
  char name[40];
  int id;
  int grade;
};

int main(void)
{
  struct student taro, hanako;
  taro.grade = 1;
  …
}
```

構造体メンバへのアクセス

メンバへのアクセスはドット演算子(.)

```c
taro.grade = 1;
```

変数名　ドット演算子(ピリオド)　メンバ名

```c
printf("%c", taro.name[2]);
```

メンバが配列の場合はメンバ名に添え字をつける

```c
scanf("%d", &taro.id);
```

アドレス演算子は構造体の変数名の前につける

構造体の定義方法いろいろ

- 構造体定義の後に変数名を書くと、同時に変数宣言できる

```
struct student {
  char name[40];
  int id;
  int grade;
} taro, hanako;
```

定義と同時に変数 taroとhanakoを宣言

- 関数ローカルな構造体宣言もできる

```
int main(void)
{
  struct student {
    char name[40];
    int id;
    int grade;
  } taro, hanako;
  taro.grade = 1;
  …
}
```

typedef 指定子による別名定義

- typedef 指定子で構造体の別名を定義することができる
  - 構造体型のstruct ～を省略できる

```
struct student {
  char name[40];
  int id;
  int grade;
};
```

既存のデータ型　新しい別名

```
typedef struct student Student;
```

```
typedef struct student {
  char name[40];
  int id;
  int grade;
} Student;
```

構造体型と別名の定義を同時に行うこともできる

  - 定義後は別名をデータ型のように使える

```
Student cs[20];
```

構造体の配列

- 構造体の配列宣言
  - 通常の変数と同様に配列を作れる

```
struct student cs[100];
```

- 構造体の配列におけるメンバ変数へのアクセス
  - 変数名に添え字をつけて、その後にドット演算子とメンバ名

```
cs[10].id = 10001;
```

  - さらにメンバの配列要素にアクセスする例

```
printf("%c", cs[20].name[0]);
```

構造体へのポインタ

- 構造体のポインタ変数
  - 通常の変数と同様にポインタ変数を宣言できる

```
struct student nu;
struct student *cs;
cs = &nu;
```

- 構造体のポインタ変数におけるメンバ変数へのアクセス
  1. 間接演算子を使う

```
(*cs).id = 20001;
```

  2. アロー演算子（->）を使う
     - 1は面倒なため、便法として用意されていると思われる

```
cs->id = 20001;
```

構造体配列の動的なメモリ割り当て

- 構造体もmalloc()で動的に作成することが可能

```
struct student *cs;
cs = malloc(n*sizeof(struct student));


...
free(cs);
```

使い終わったらfree()で解放するのを忘れずに！

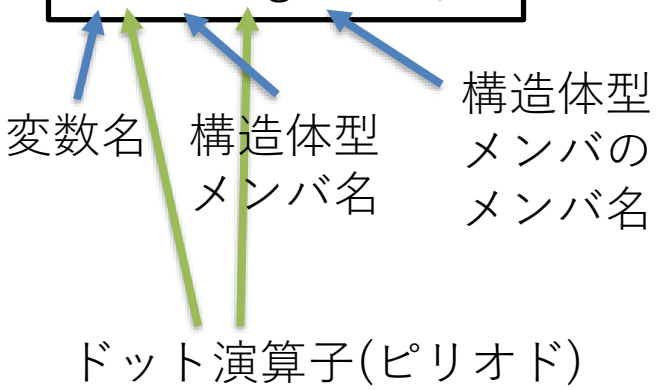
構造体のネスト（Nested structure）

- 構造体メンバの型として、別の構造体を使うことができる

```
struct personal {
  char name[40];
  char gender;
  int age;
};

struct student {
  struct personal data;
  int id;
  int grade;
};
```

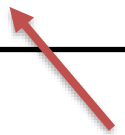
```
struct student cs;
cs.data.age = 20;
```



メンバ変数に自分自身の型を使う

- 自分自身の構造体型を使うこともできる

```
struct list {
  int value;
  struct list *next;
};
```



ポインタもメンバにできる

構造体の配列 と 配列の構造体

- 構造体の配列 と 配列の構造体 は、メモリ上のデータ配置が異なる

構造体の配列の例

```
struct element {
  int id;
  int val;
};

struct element item[N];
```

配列の構造体の例

```
struct element {
  int id[N];
  int val[N];
};

struct element item;
```

構造体の配列のメモリ配置イメージ



配列の構造体のメモリ配置イメージ



- どちらもC言語として自然な実装であり問題なく利用することができる
- 書きやすさ、データの扱いやすさ、処理速度などを考えて選ぶ
  - 使いやすさの観点では、構造体の配列が使われることが多い
  - 処理速度重視の場合、最近のプロセッサ向けの実装では、配列の構造体が使われることが多い

構造体のコピー

- 配列と違い、構造体はメンバごとコピーすることができる

```
int a[2] = {1,2};
int b[2];
b = a;
```

これはダメ

構造体のコピーはOK

```
#include <stdio.h>
#include <string.h>

int main(void)
{
  struct student {
    char name[40];
    int id;
    int grade;
  };
  struct student taro, jiro;
  strcpy(taro.name, "taro");
  taro.id = 1;
  taro.grade = 1;
  strcpy(jiro.name, "jiro");
  jiro.id = 2;
  jiro.grade = 2;
  printf("before: name=%s id=%d grade=%d\n", jiro.name, jiro.id, jiro.grade);
  jiro = taro;
  printf("after: name=%s id=%d grade=%d\n", jiro.name, jiro.id, jiro.grade);

  return 0;
}
```

構造体に含まれる配列メンバまでコピーされる

```
$ ./a.out
before: name=jiro id=2 grade=2
after: name=taro id=1 grade=1
```

メンバにポインタを持つ構造体のコピーは要注意

nameをポインタに

mallocでメモリ確保

構造体のコピーは同じ

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(void)
{
  struct student {
    char *name;
    int id;
    int grade;
  };
  struct student taro, jiro;
  taro.name=malloc(sizeof(char)*40);
  strcpy(taro.name,"taro");
  taro.id = 1;
  taro.grade = 1;
  jiro.name=malloc(sizeof(char)*40);
  strcpy(jiro.name,"jiro");
  jiro.id = 2;
  jiro.grade = 2;
  printf("before: name=%s id=%d grade=%d\n", jiro.name, jiro.id, jiro.grade);
  jiro = taro;
  printf("after1: name=%s id=%d grade=%d\n", jiro.name, jiro.id, jiro.grade);
  taro.name[0] = 'm';
  printf("after2: name=%s id=%d grade=%d\n", jiro.name, jiro.id, jiro.grade);

  return 0;
}
```

taroのメンバを書き換えて、jiroのメンバを出力

```
$ ./a.out
before: name=jiro id=2 grade=2
after1: name=taro id=1 grade=1
after2: name=maro id=1 grade=1
```

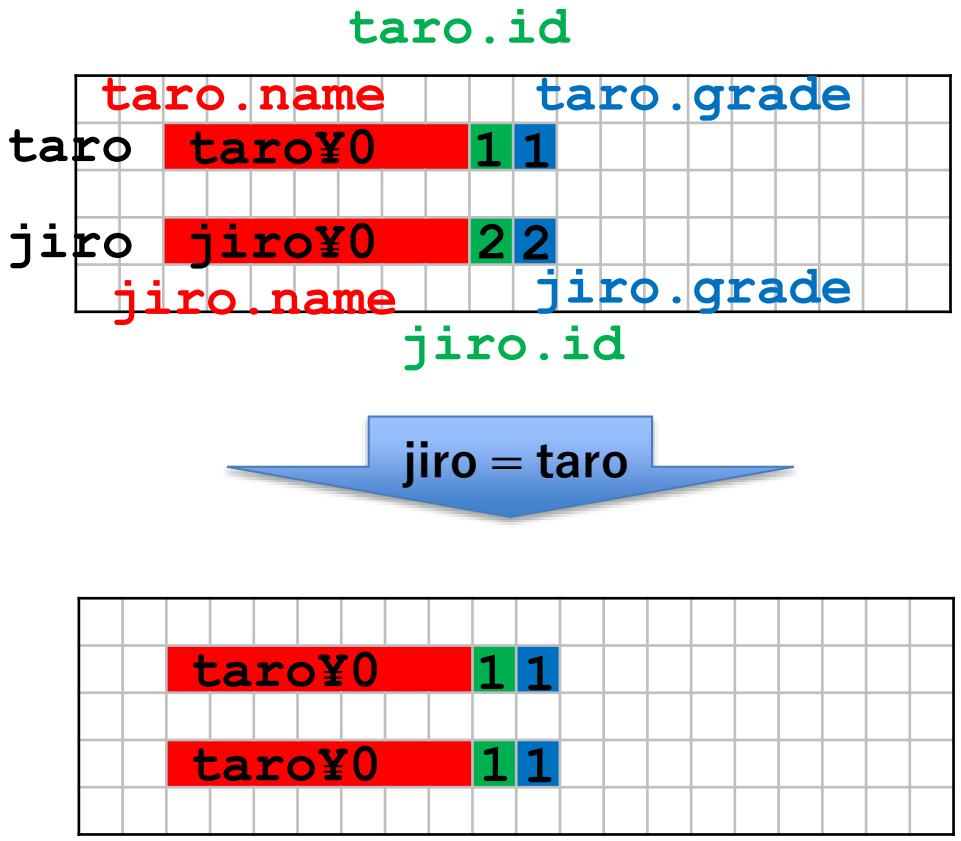
jiroも書き変わってる

構造体コピー時のメモリイメージ

- メンバが配列の時

```
struct student {
  char name[40];
  int id;
  int grade;
};
struct student taro, jiro;
…
jiro = taro;
```

メモリ空間のイメージ

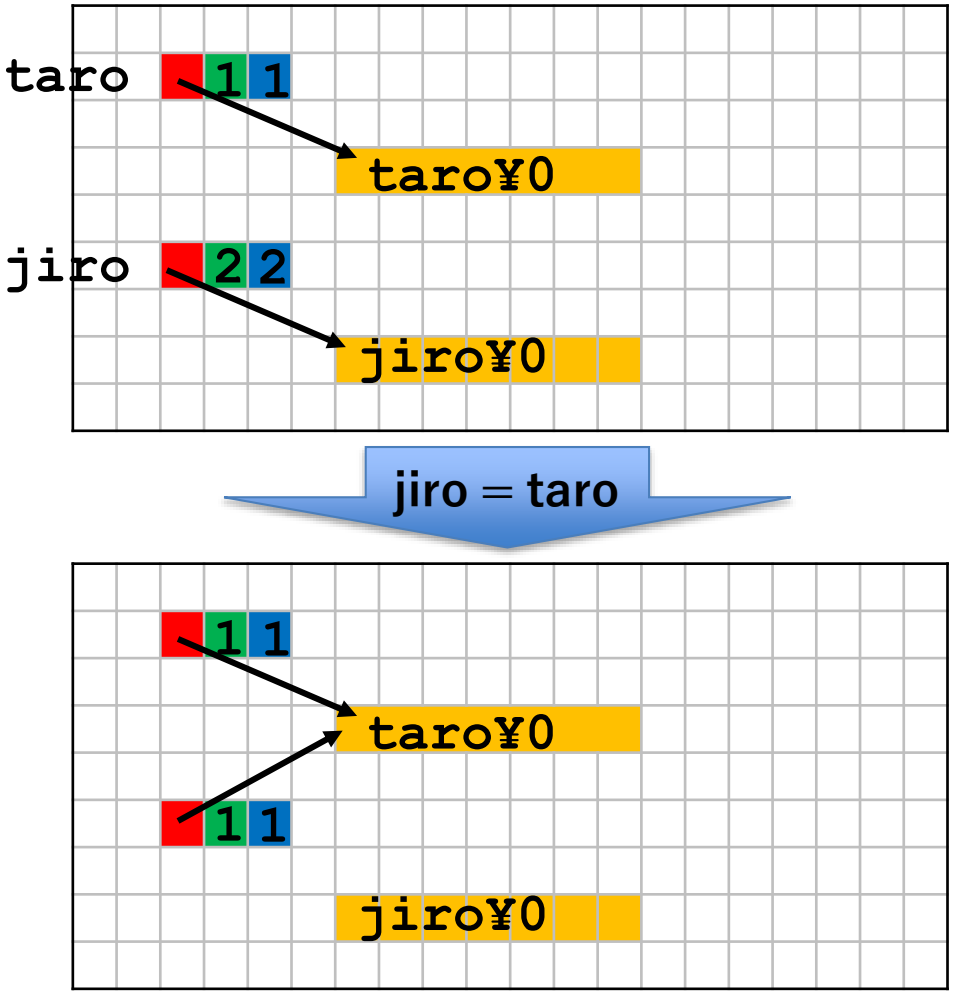


構造体コピー時のメモリイメージ

- メンバがポインタの時

```
struct student {
  char *name;
  int id;
  int grade;
};
struct student taro, jiro;
taro.name=malloc(sizeof(char)*40);
…
jiro.name=malloc(sizeof(char)*40);
…
jiro = taro;
taro.name[0] = 'm';
```

メモリ空間のイメージ



taroのポインタメンバに保存されているアドレスがコピーされるので、同じ領域を指すことになる。よって、”taro¥0”を”maro¥0”に書き換えるとjiro側にも反映されてしまう。

関数の引数に構造体を使う

関数を跨いで使うので、構造体の定義をグローバルに

使い方は通常の変数と同じ

通常の変数と同じく、仮引数にはコピーが渡る

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

struct student {
  char *name;
  int id;
  int grade;
};

void foo(struct student s);
int main(void)
{
  struct student taro;
  taro.name=malloc(sizeof(char)*40);
  strcpy(taro.name,"taro");
  taro.id = 1;
  taro.grade = 1;
  printf("before: name=%s id=%d grade=%d\n", taro.name, taro.id, taro.grade);
  foo(taro);
  printf("after: name=%s id=%d grade=%d\n", taro.name, taro.id, taro.grade);
  return 0;
}

void foo(struct student s){
  s.id = 2;
  s.name[0] = 'm';
}
```

コピーの構造体メンバを書き換えてるので、呼び出しもとに影響なし

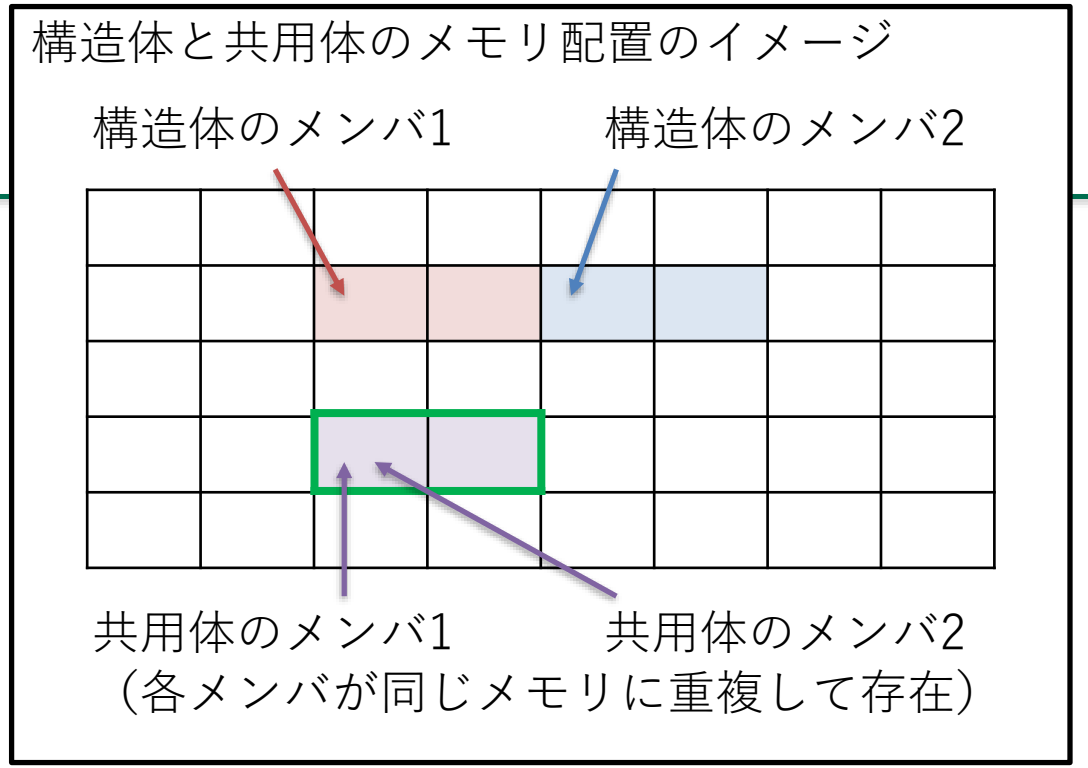
s.nameがtaro.nameと同じアドレスを指すため、呼び出し元も書き換わる

```
$ ./a.out
before: name=taro id=1 grade=1
after: name=maro id=1 grade=1
```

共用体

- 同じメモリ領域を複数の変数で共有できる

```
union タグ名 {
  型　メンバ1;
...
  型　メンバN;
} 変数リスト;
```



- 例
  - 同じ8バイトであるlongとdoubleで同じメモリ領域を共有させ、整数型しか対応していないシフト演算をdouble型変数に行う

```
union dl {
  long l;
  double d;
};
```

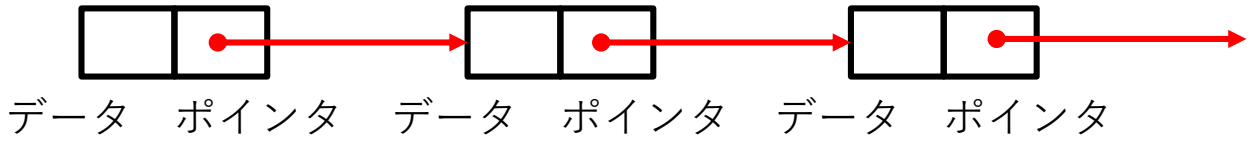
```
union dl data;
data.d = 123.456;
data.l = data.l >> 2;
data.l = data.l << 2;
```

  - メンバへのアクセス方法は構造体と同じ

- ただし、この例では具体的に嬉しいことはない
- メモリ上の配置を理解した上で強制的に処理したいとき、特定の型の変数しか受け付けない関数をだましたいときなどには有用

リスト

- リスト
  - よく知られたデータ構造の1つに連結リスト(linked list)がある。連結リストでは、データを持つノードがあり、さらに各ノードは他のノードへの参照（link）を含んでいる。ここでは最も簡単な単方向リスト（singly-linked list）について考える。
  - Pythonではリストは基本的な機能として提供されているが、C言語でリストを使いたい場合は自分で実装する必要がある。



- 配列との違い
  - あらかじめサイズが分かっている（上限を決められる）場合は静的配列を使う方法が良い
  - 実行時にサイズが分かる場合は動的配列を使えるが、効率的なデータの追加・削除が困難
  - データへのアクセス効率は劣るが、データの追加・削除が容易

（参考）Pythonのリスト

- Pythonでは括弧（[ ]）とカンマ区切りで複数のオブジェクトを囲むことでリストを作ることができる
  - あらかじめ用意されており自分で実装する必要はない
- Pythonにおけるリストの主なメソッド
  - append(): リストの末尾に要素を追加
  - insert(): 指定位置に要素を追加
  - pop(): 指定位置の要素を削除
  - remove(): 値が一致する要素を削除
  - index(): 値が一致する要素の位置を取得
  - count(): 値が一致する要素数を取得
  - sort(): 要素の並び替え

単方向リストとappend()

- 以下の構造体slistが定義されているとき、リストの末尾に要素を追加する関数slist_append()を考える
  - 構造体struct listの定義（別名slist）

```
typedef struct list {
  int value;
  struct list *next;
} slist;
```

int型のデータ

次の要素へのポインタ

構造体の定義とmain()

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct list {
  int value;
  struct list *next;
} slist;



int main(void)
{
  slist *head = NULL;
  slist_append(&head, 2);
  slist_append(&head, 1);
  slist_append(&head, 3);
  slist_display(head);
  slist_free(&head);
  return 0;
}
```

構造体struct listを定義し、 同時にtypedefで別名slistを定義

リストの先頭へのポインタを持つ変数head

末尾に要素を追加するappend()を3回呼び出し

リストの値を表示

リストの解放

末尾に要素を追加するslist_append()

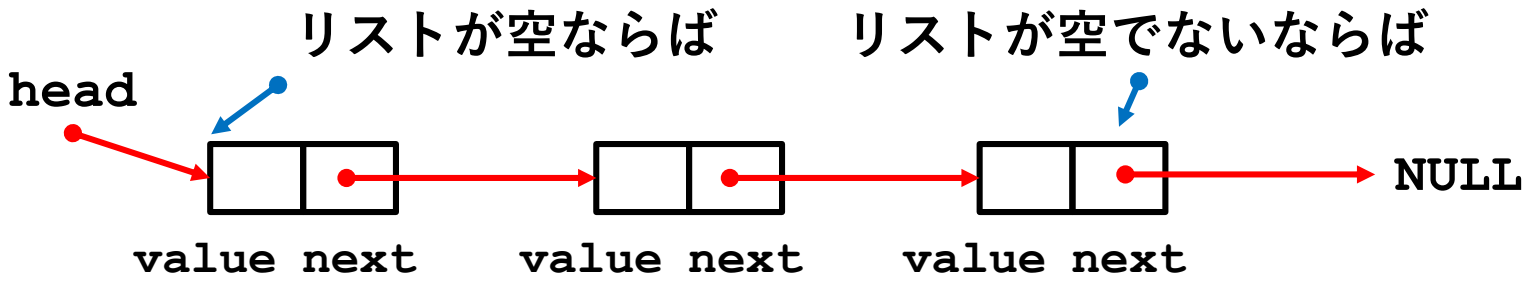
```
void slist_append(slist **head, int value)
{
  slist *node = malloc(sizeof(slist));
  node->value = value;
  node->next = NULL;
  if(*head == NULL){
    *head = node;
  }else{
    slist *p = *head;
    while(p->next != NULL){
      p = p->next;
    }
    p->next = node;
  }
}
```

先頭へのポインタのポインタを受け取る

新しい要素を作成

リストが空の場合、新しい要素を接続する

空ではない場合、末尾の要素まで移動し、新しい要素を追加する

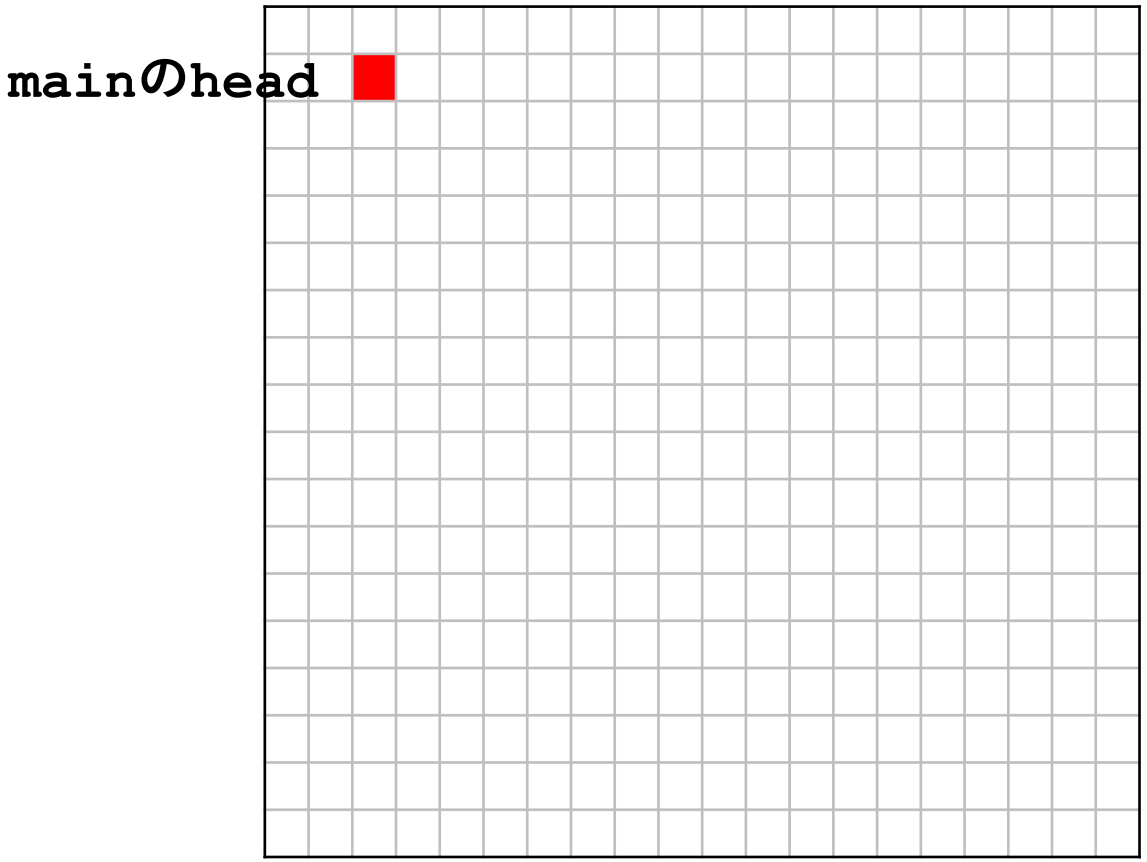


slist_append()のメモリ空間のイメージ

```
    :
  slist *head = NULL;
  slist_append(&head, 2);
  slist_append(&head, 1);
    :
```

```
void slist_append(slist **head, int value)
{
  slist *node = malloc(sizeof(slist));
  node->value = value;
  node->next = NULL;
  if(*head == NULL){
    *head = node;
  }else{
    slist *p = *head;
    while(p->next != NULL){
      p = p->next;
    }
    p->next = node;
  }
}
```

メモリ空間のイメージ

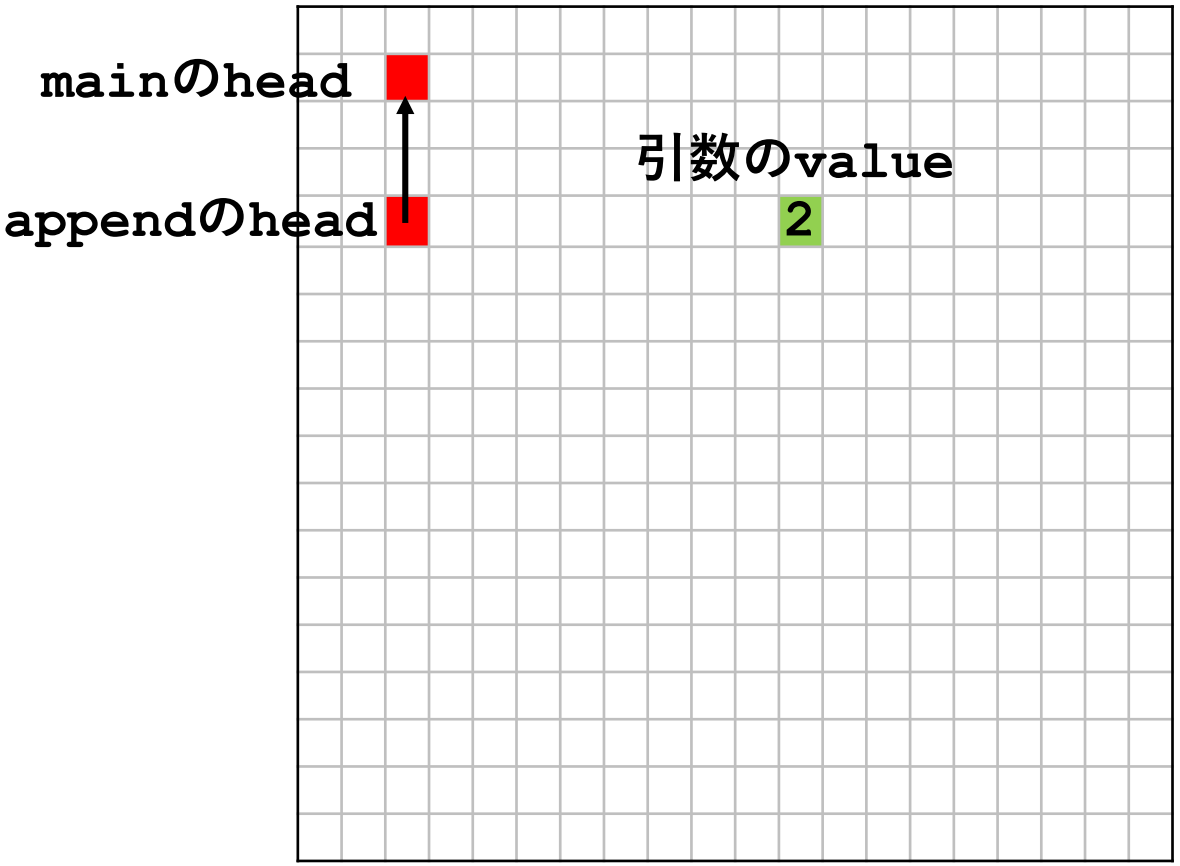


slist_append()のメモリ空間のイメージ

```
    :
  slist *head = NULL;
  slist_append(&head, 2);
  slist_append(&head, 1);
    :
```

```
void slist_append(slist **head, int value)
{
  slist *node = malloc(sizeof(slist));
  node->value = value;
  node->next = NULL;
  if(*head == NULL){
    *head = node;
  }else{
    slist *p = *head;
    while(p->next != NULL){
      p = p->next;
    }
    p->next = node;
  }
}
```

メモリ空間のイメージ

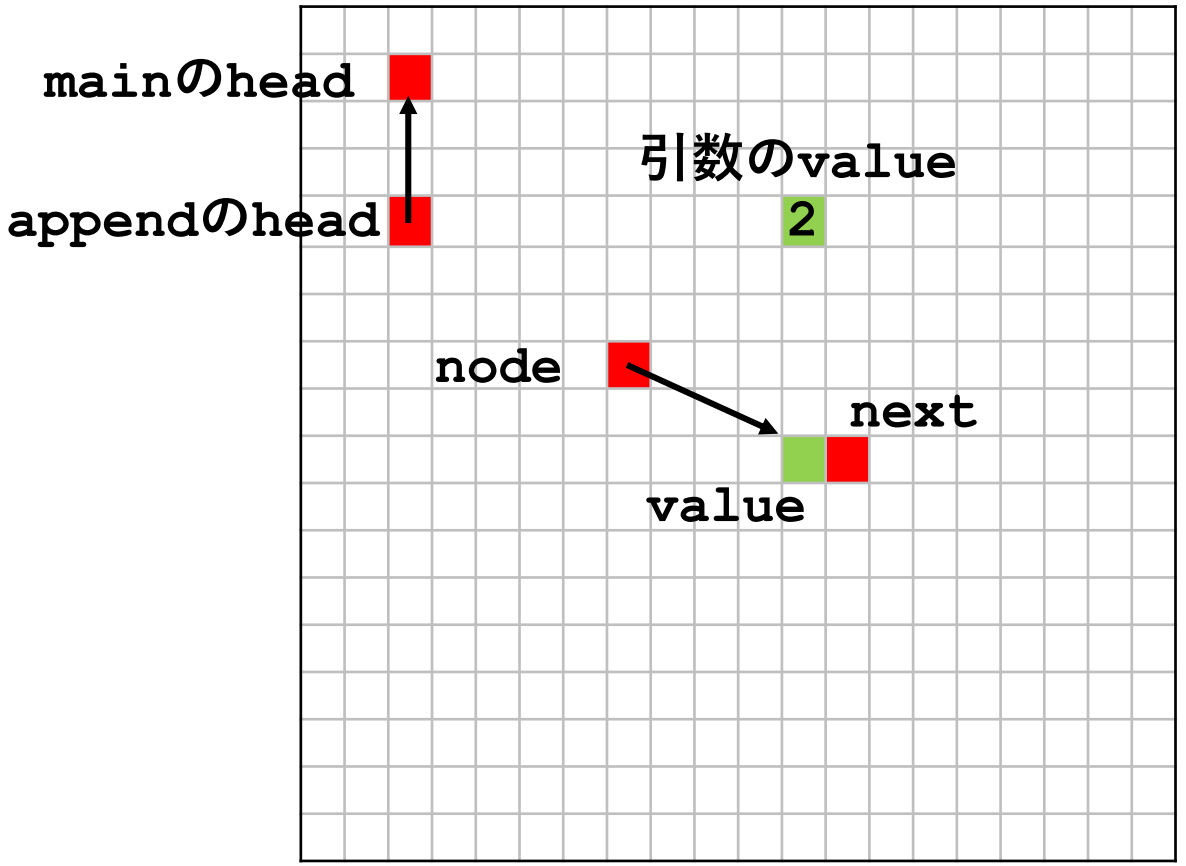


slist_append()のメモリ空間のイメージ

```
    :
  slist *head = NULL;
  slist_append(&head, 2);
  slist_append(&head, 1);
    :
```

```
void slist_append(slist **head, int value)
{
  slist *node = malloc(sizeof(slist));
  node->value = value;
  node->next = NULL;
  if(*head == NULL){
    *head = node;
  }else{
    slist *p = *head;
    while(p->next != NULL){
      p = p->next;
    }
    p->next = node;
  }
}
```

メモリ空間のイメージ

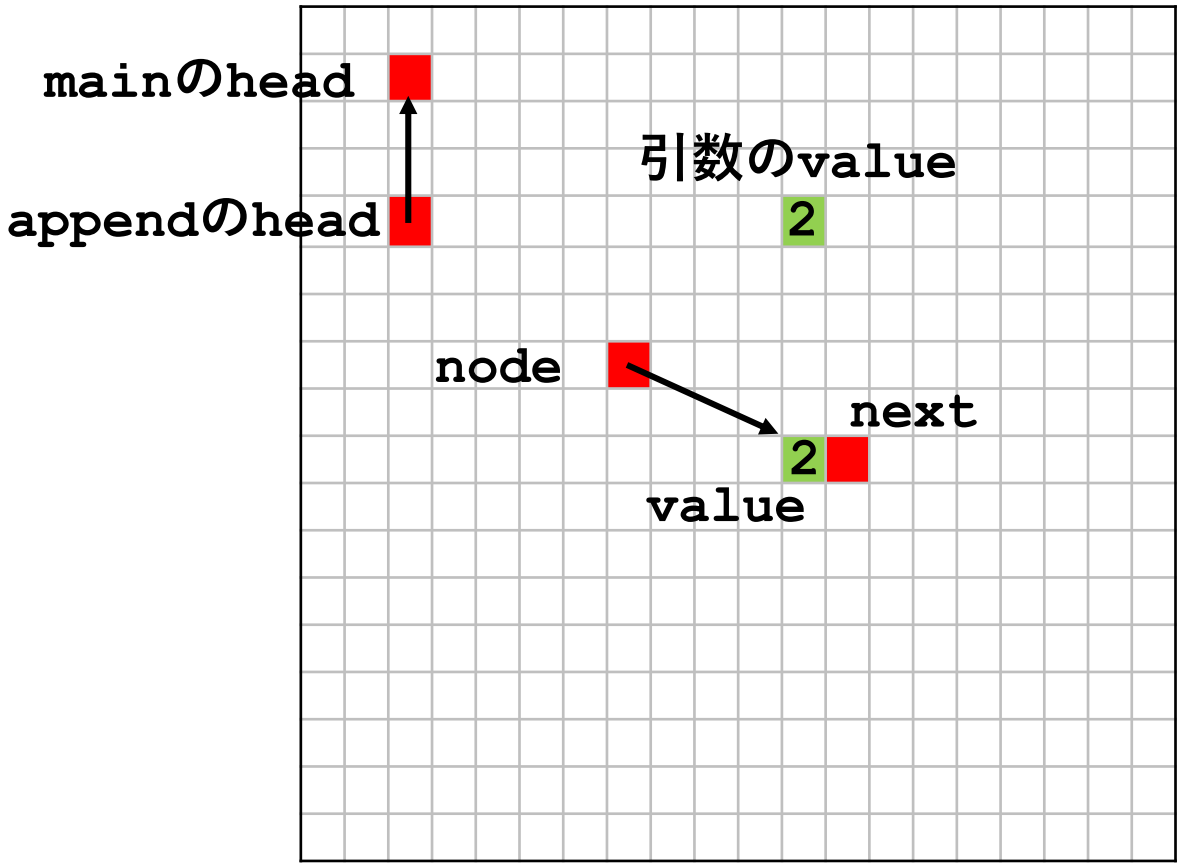


slist_append()のメモリ空間のイメージ

```
    :
  slist *head = NULL;
  slist_append(&head, 2);
  slist_append(&head, 1);
    :
```

```
void slist_append(slist **head, int value)
{
  slist *node = malloc(sizeof(slist));
  node->value = value;
  node->next = NULL;
  if(*head == NULL){
    *head = node;
  }else{
    slist *p = *head;
    while(p->next != NULL){
      p = p->next;
    }
    p->next = node;
  }
}
```

メモリ空間のイメージ

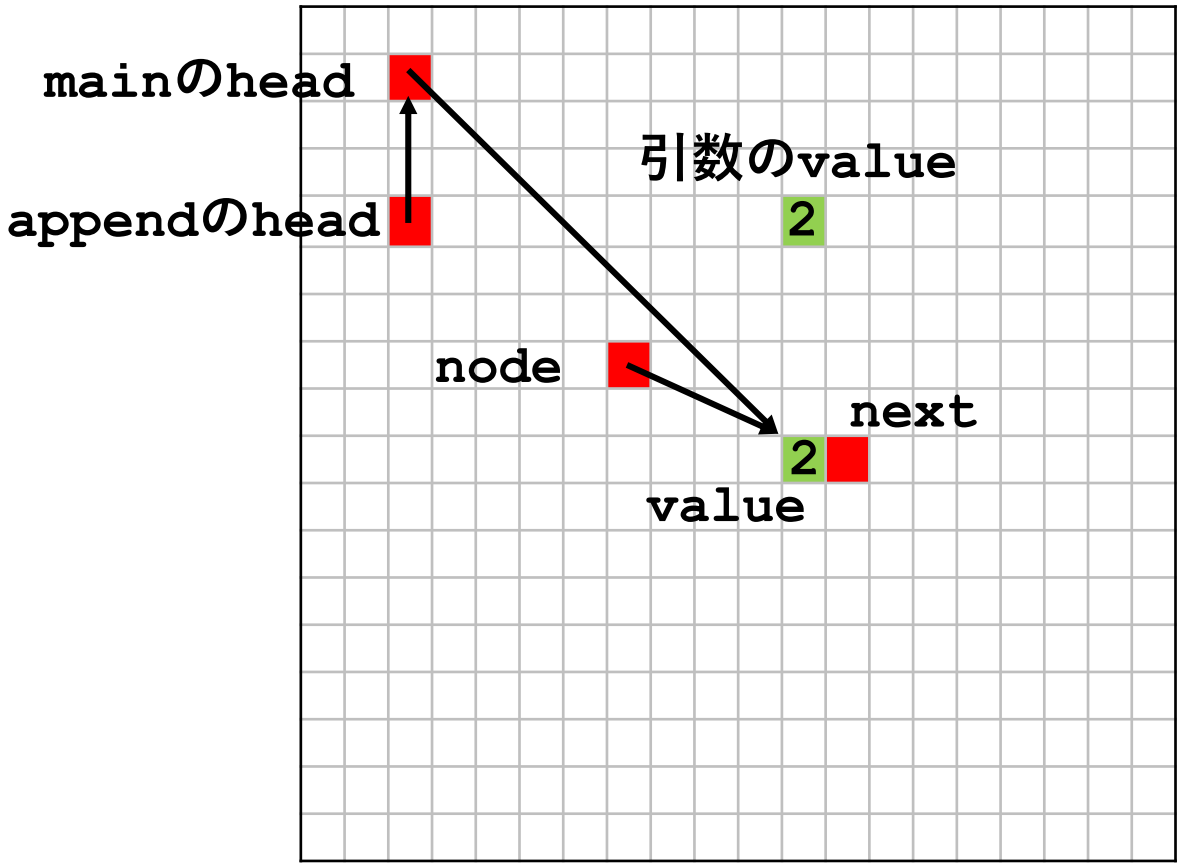


slist_append()のメモリ空間のイメージ

```
    :
  slist *head = NULL;
  slist_append(&head, 2);
  slist_append(&head, 1);
    :
```

```
void slist_append(slist **head, int value)
{
  slist *node = malloc(sizeof(slist));
  node->value = value;
  node->next = NULL;
  if(*head == NULL){
    *head = node;
  }else{
    slist *p = *head;
    while(p->next != NULL){
      p = p->next;
    }
    p->next = node;
  }
}
```

メモリ空間のイメージ

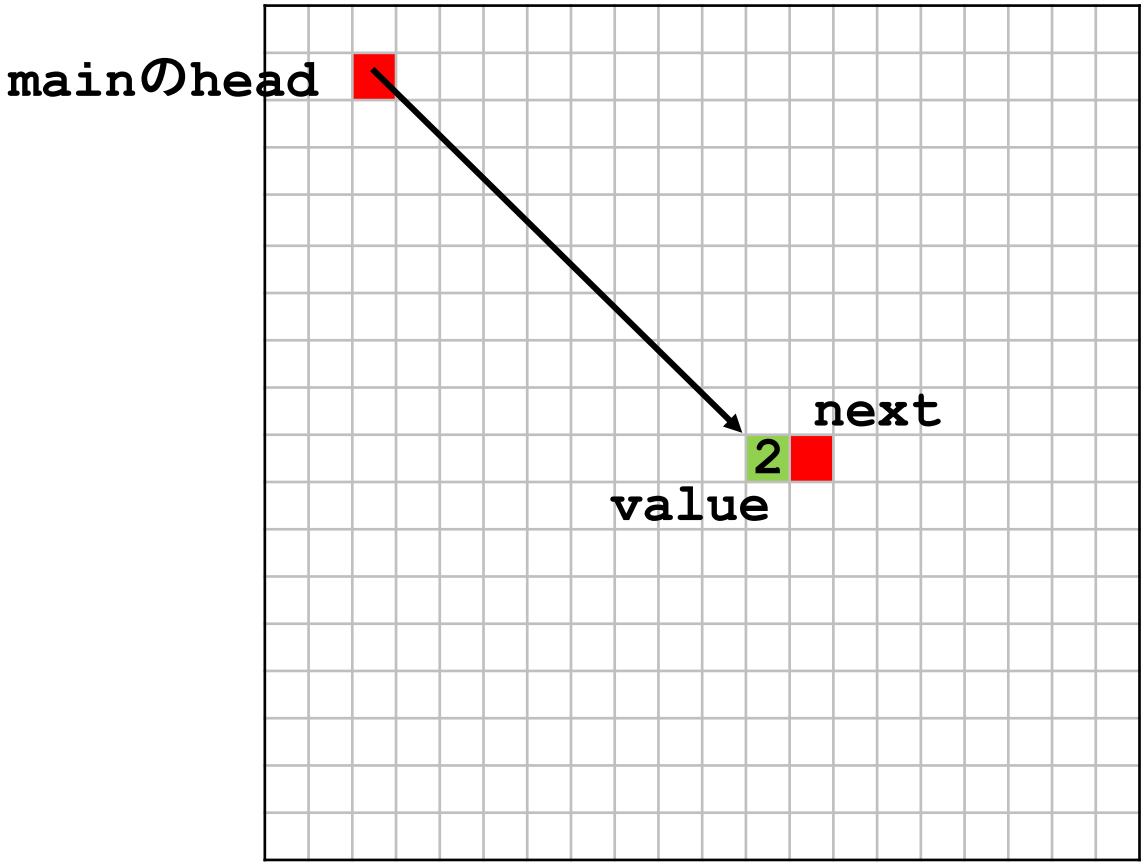


slist_append()のメモリ空間のイメージ

```
    :
  slist *head = NULL;
  slist_append(&head, 2);
  slist_append(&head, 1);
    :
```

```
void slist_append(slist **head, int value)
{
  slist *node = malloc(sizeof(slist));
  node->value = value;
  node->next = NULL;
  if(*head == NULL){
    *head = node;
  }else{
    slist *p = *head;
    while(p->next != NULL){
      p = p->next;
    }
    p->next = node;
  }
}
```

メモリ空間のイメージ



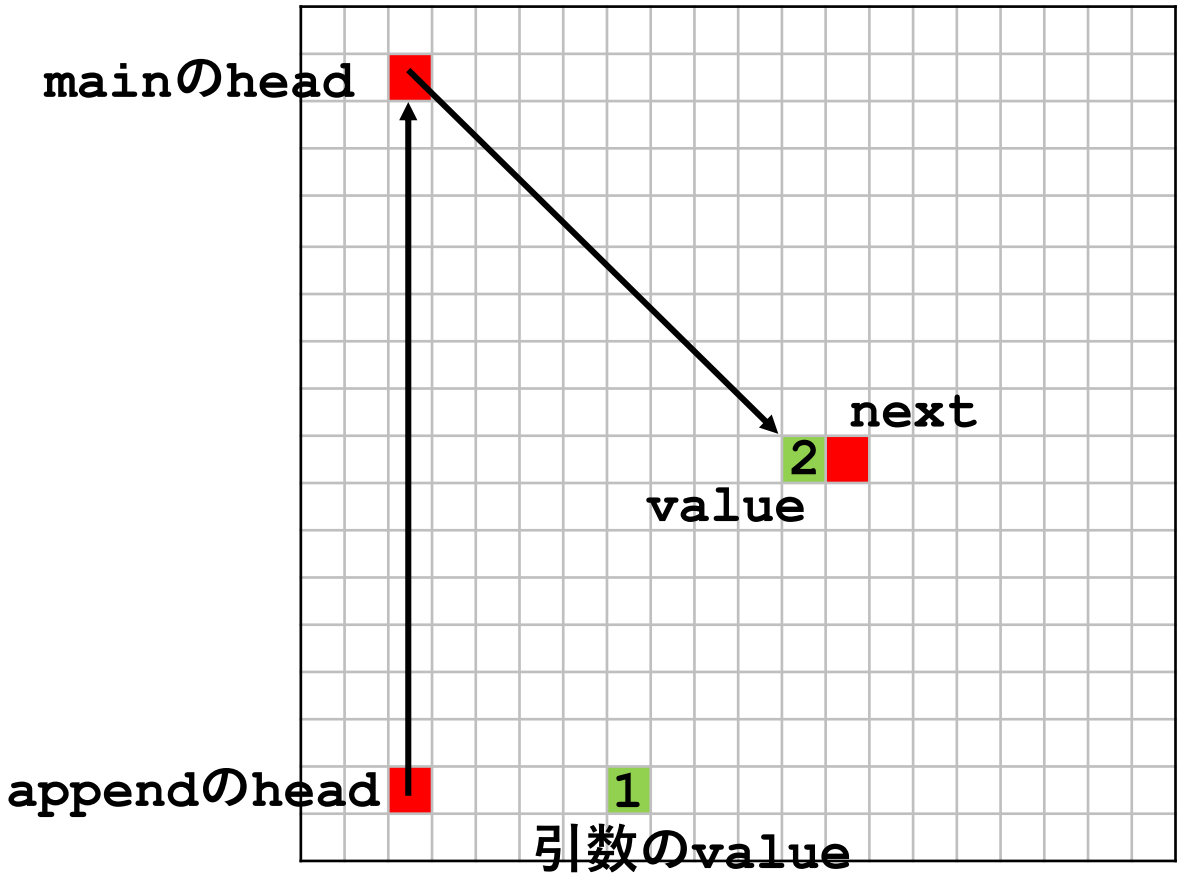
関数を抜ける時にローカル変数が消え、↑が残る

slist_append()のメモリ空間のイメージ

```
    :
  slist *head = NULL;
  slist_append(&head, 2);
  slist_append(&head, 1);
    :
```

```
void slist_append(slist **head, int value)
{
  slist *node = malloc(sizeof(slist));
  node->value = value;
  node->next = NULL;
  if(*head == NULL){
    *head = node;
  }else{
    slist *p = *head;
    while(p->next != NULL){
      p = p->next;
    }
    p->next = node;
  }
}
```

メモリ空間のイメージ

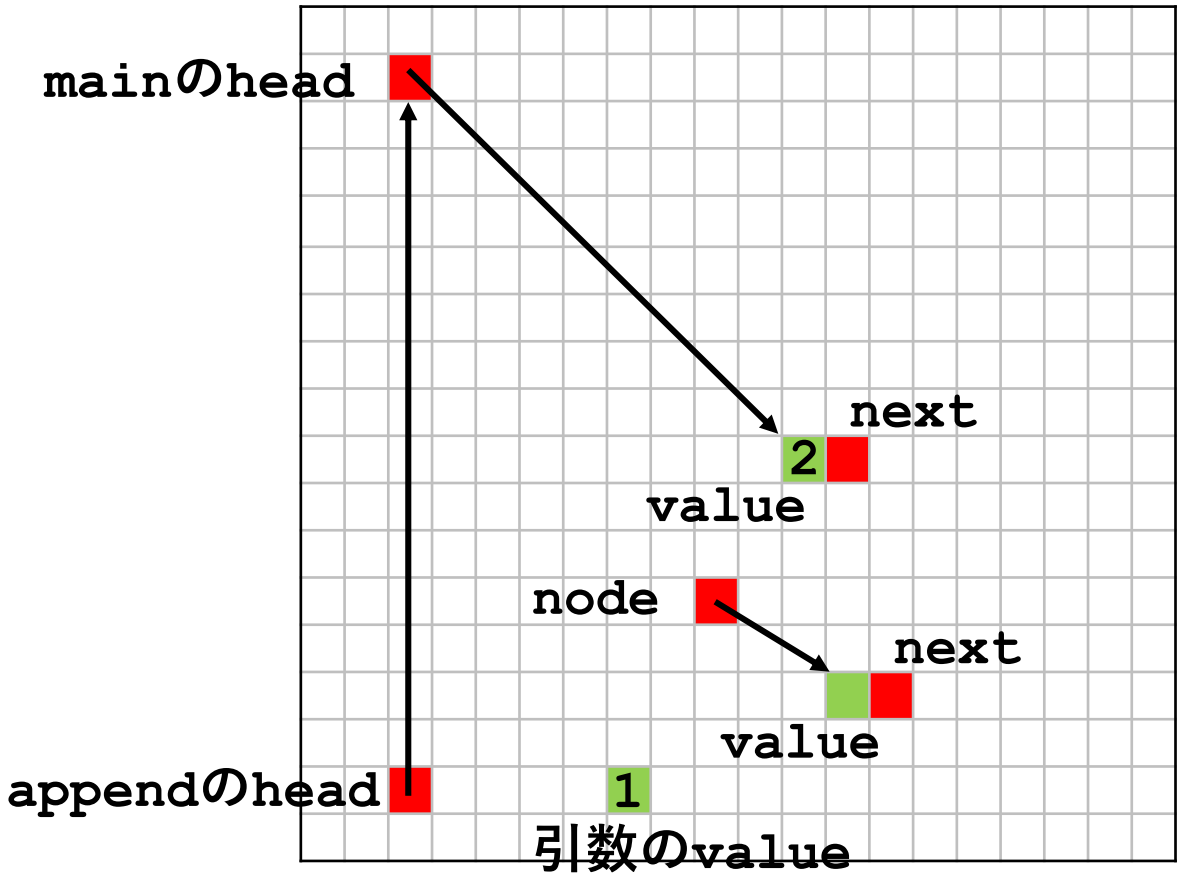


slist_append()のメモリ空間のイメージ

```
    :
  slist *head = NULL;
  slist_append(&head, 2);
  slist_append(&head, 1);
    :
```

```
void slist_append(slist **head, int value)
{
  slist *node = malloc(sizeof(slist));
  node->value = value;
  node->next = NULL;
  if(*head == NULL){
    *head = node;
  }else{
    slist *p = *head;
    while(p->next != NULL){
      p = p->next;
    }
    p->next = node;
  }
}
```

メモリ空間のイメージ

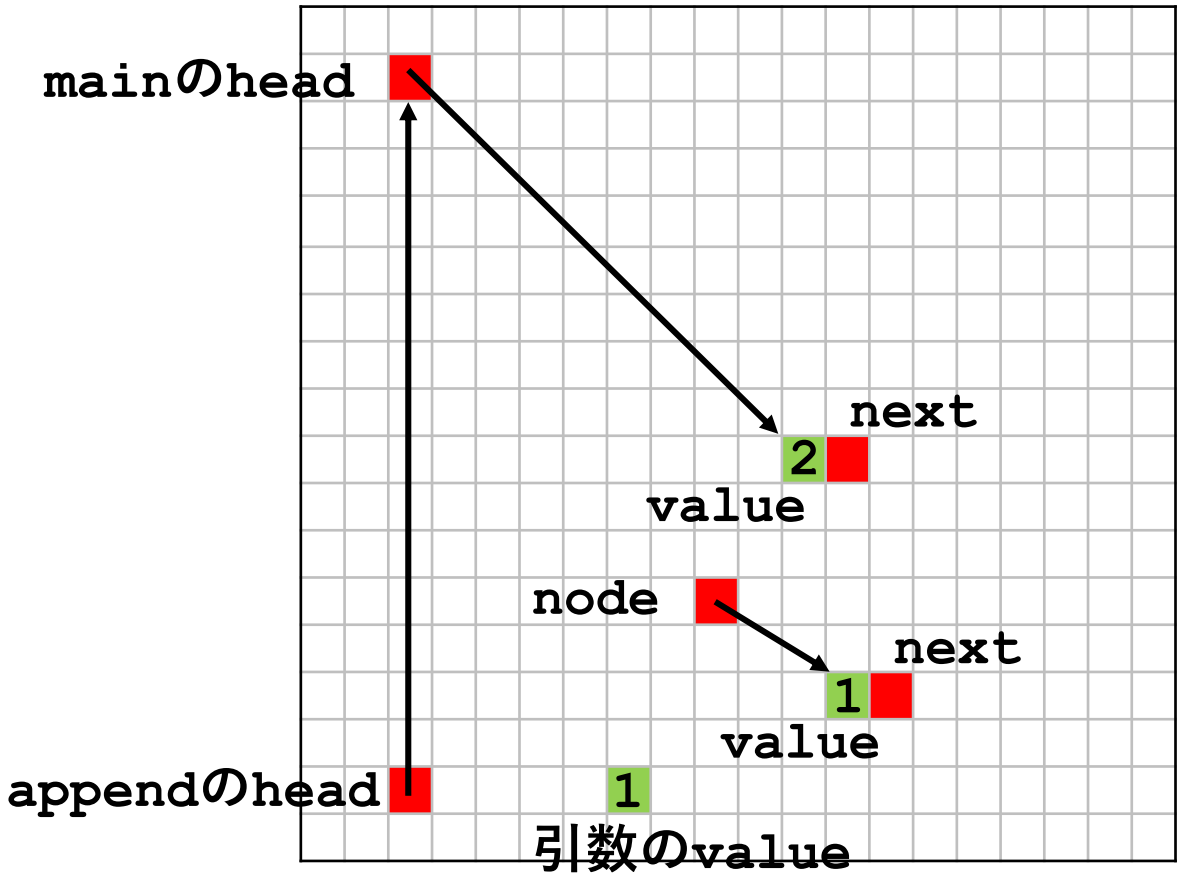


slist_append()のメモリ空間のイメージ

```
    :
  slist *head = NULL;
  slist_append(&head, 2);
  slist_append(&head, 1);
    :
```

```
void slist_append(slist **head, int value)
{
  slist *node = malloc(sizeof(slist));
  node->value = value;
  node->next = NULL;
  if(*head == NULL){
    *head = node;
  }else{
    slist *p = *head;
    while(p->next != NULL){
      p = p->next;
    }
    p->next = node;
  }
}
```

メモリ空間のイメージ

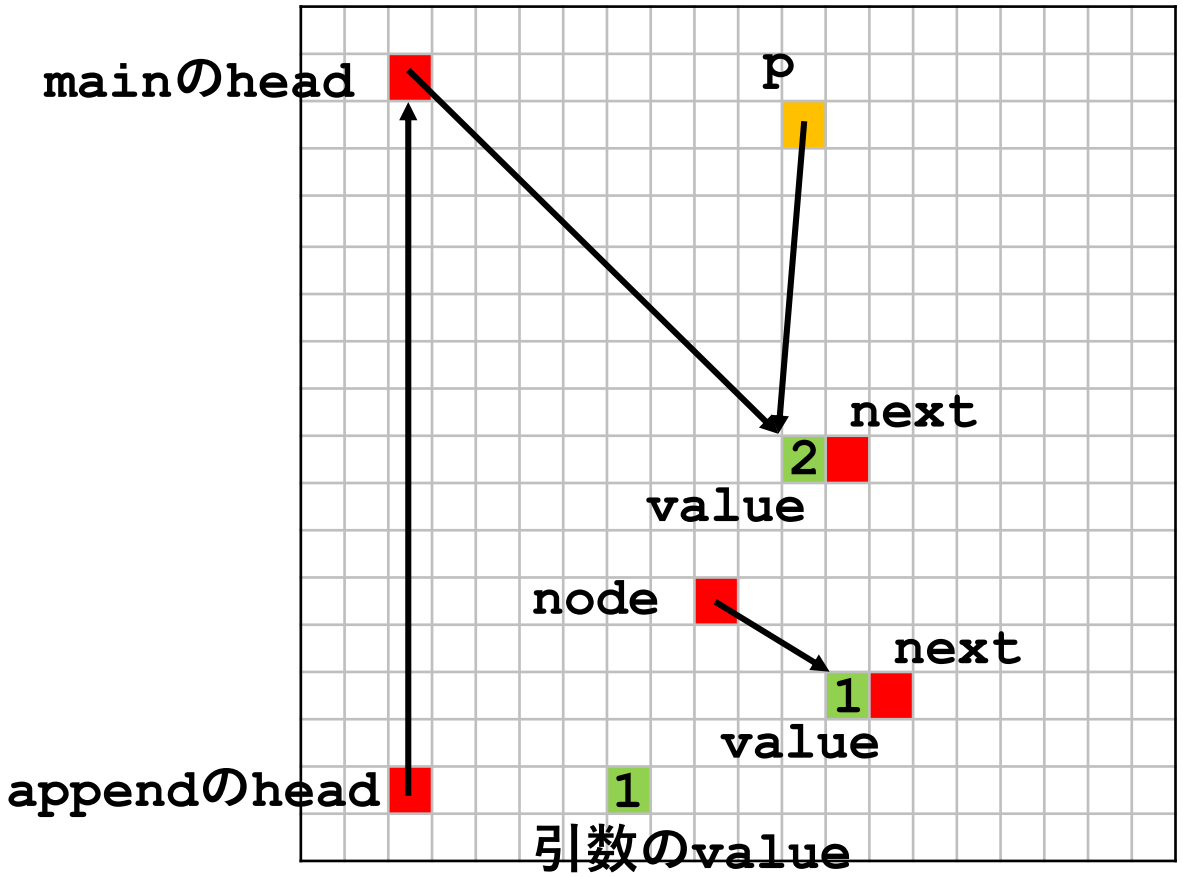


slist_append()のメモリ空間のイメージ

```
    :
  slist *head = NULL;
  slist_append(&head, 2);
  slist_append(&head, 1);
    :
```

```
void slist_append(slist **head, int value)
{
  slist *node = malloc(sizeof(slist));
  node->value = value;
  node->next = NULL;
  if(*head == NULL){
    *head = node;
  }else{
    slist *p = *head;
    while(p->next != NULL){
      p = p->next;
    }
    p->next = node;
  }
}
```

メモリ空間のイメージ



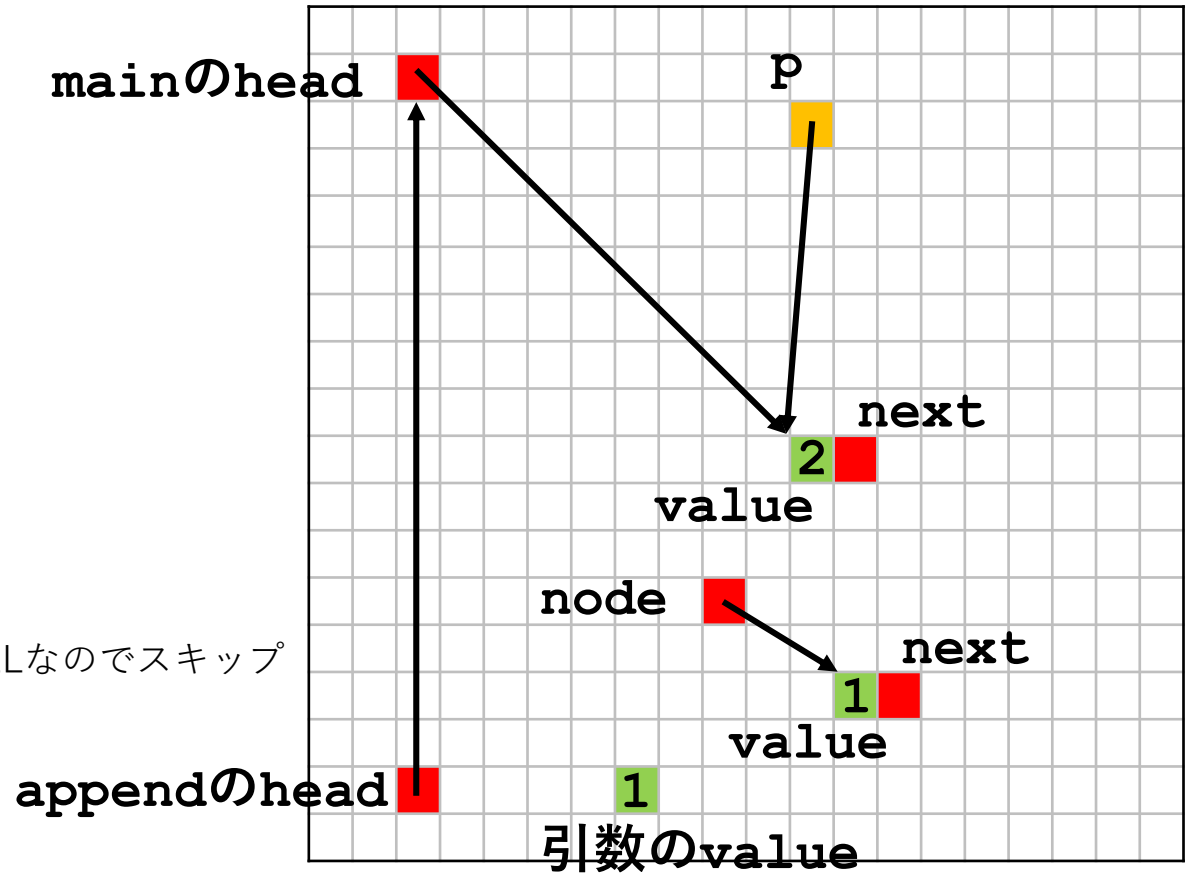
slist_append()のメモリ空間のイメージ

```
    :
  slist *head = NULL;
  slist_append(&head, 2);
  slist_append(&head, 1);
    :
```

```
void slist_append(slist **head, int value)
{
  slist *node = malloc(sizeof(slist));
  node->value = value;
  node->next = NULL;
  if(*head == NULL){
    *head = node;
  }else{
    slist *p = *head;
    while(p->next != NULL){
      p = p->next;
    }
    p->next = node;
  }
}
```

NULLなのでスキップ

メモリ空間のイメージ

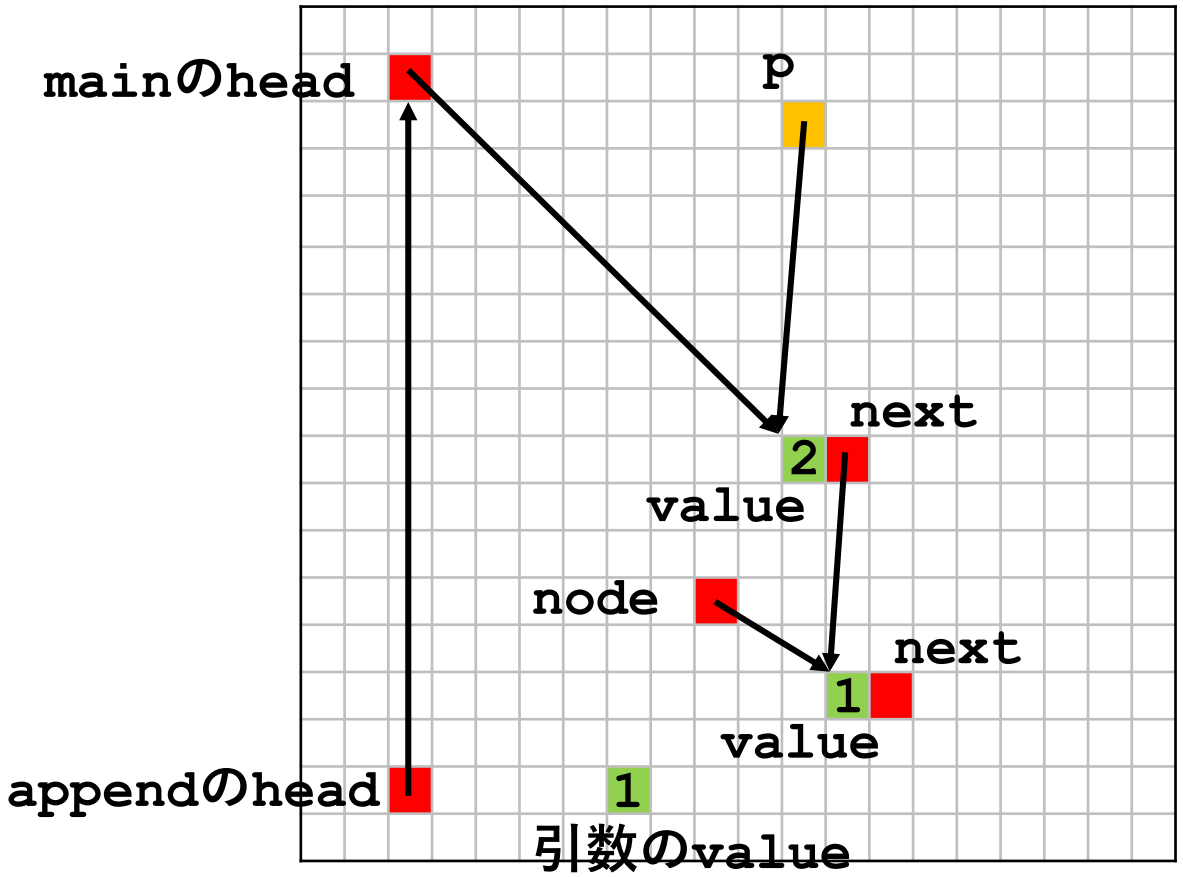


slist_append()のメモリ空間のイメージ

```
    :
  slist *head = NULL;
  slist_append(&head, 2);
  slist_append(&head, 1);
    :
```

```
void slist_append(slist **head, int value)
{
  slist *node = malloc(sizeof(slist));
  node->value = value;
  node->next = NULL;
  if(*head == NULL){
    *head = node;
  }else{
    slist *p = *head;
    while(p->next != NULL){
      p = p->next;
    }
    p->next = node;
  }
}
```

メモリ空間のイメージ

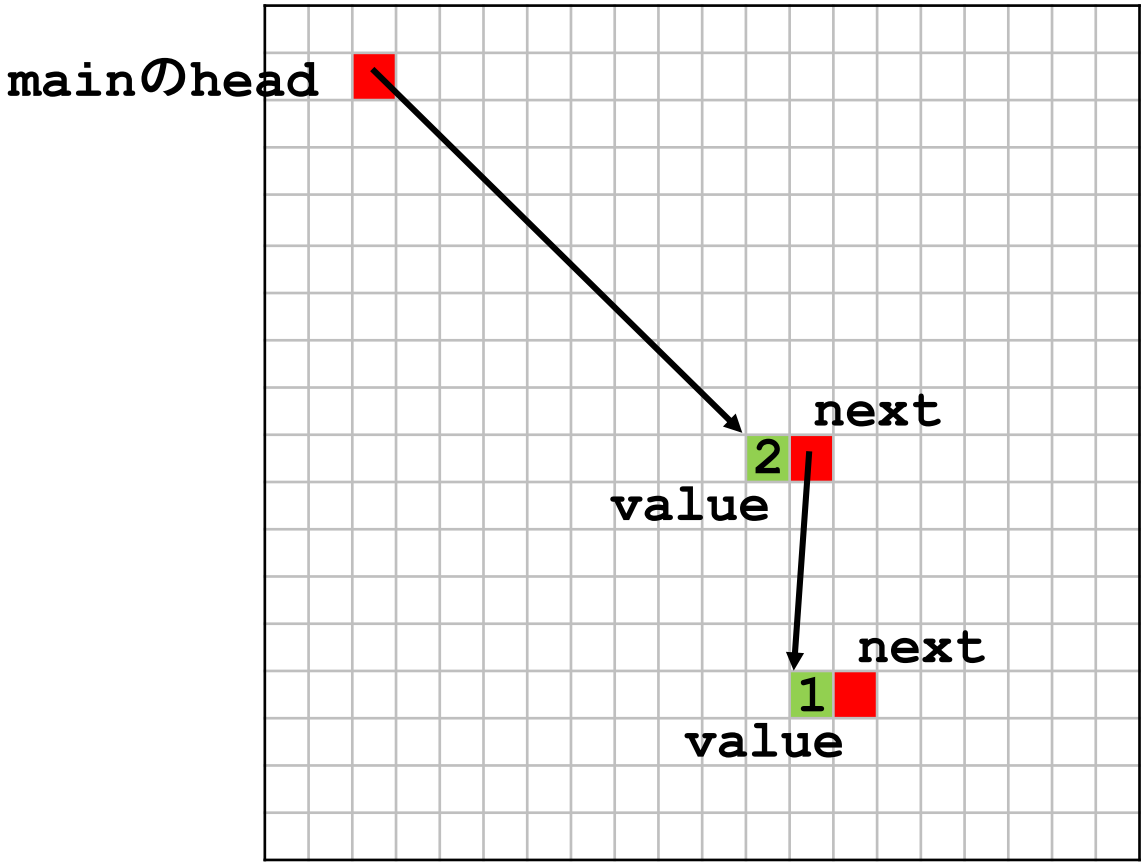


slist_append()のメモリ空間のイメージ

```
    :
  slist *head = NULL;
  slist_append(&head, 2);
  slist_append(&head, 1);
    :
```

```
void slist_append(slist **head, int value)
{
  slist *node = malloc(sizeof(slist));
  node->value = value;
  node->next = NULL;
  if(*head == NULL){
    *head = node;
  }else{
    slist *p = *head;
    while(p->next != NULL){
      p = p->next;
    }
    p->next = node;
  }
}
```

メモリ空間のイメージ



関数を抜ける時にローカル変数が消え、↑が残る

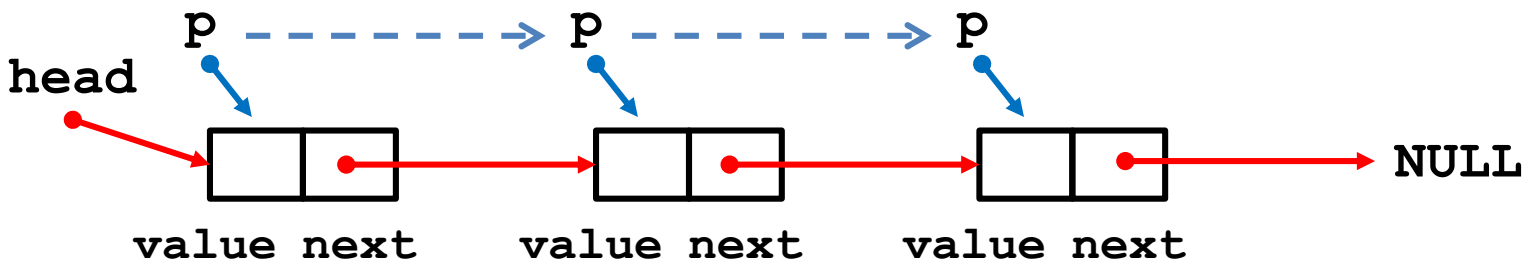
要素の値を出力するslist_display()

```
void slist_display(slist *head)
{
  slist *p = head;
  while(p != NULL){
    printf("%d ", p->value);
    p = p->next;
  }
  printf("¥n");
}
```

リストのある要素へのポインタを指す変数p（先頭headから始める）

要素がNULLになるまで繰り返す

メンバnextを代入して次の要素に移動する

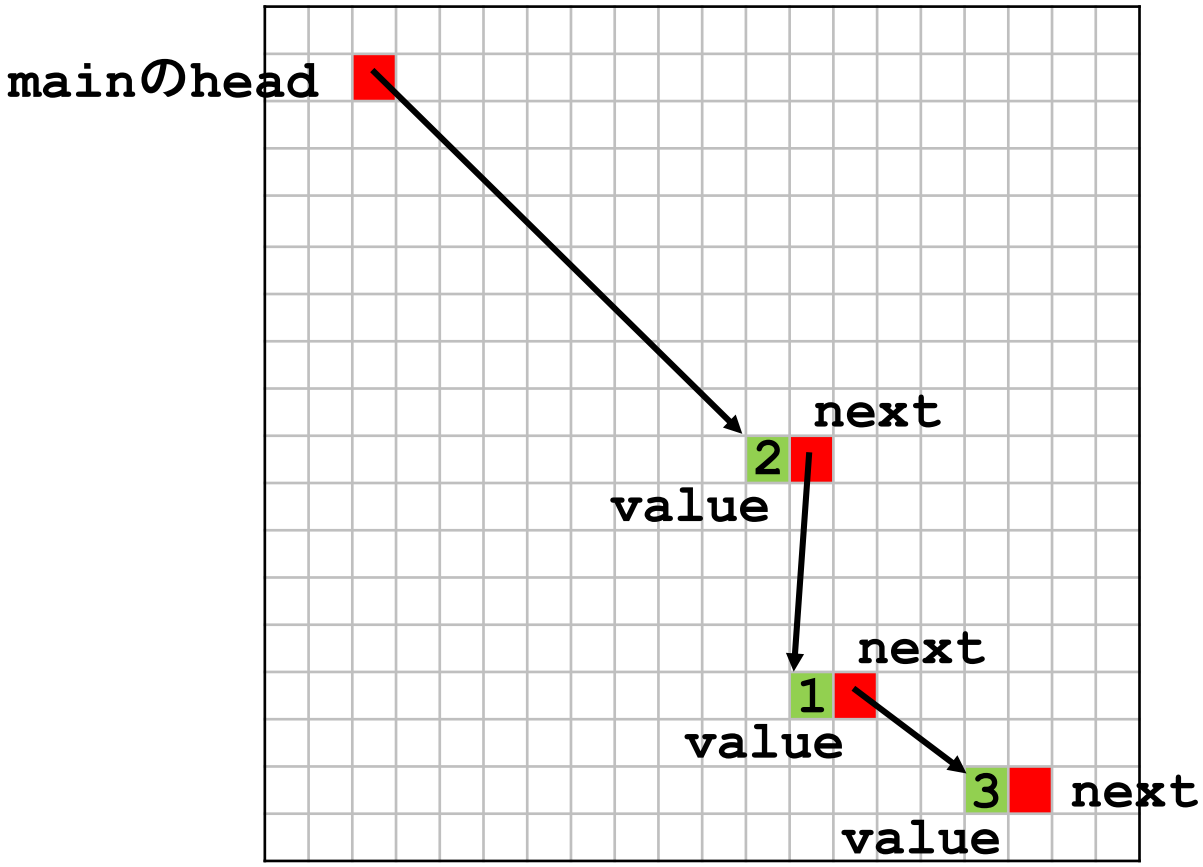


slist_display()のメモリ空間のイメージ

```
    :
  slist *head = NULL;
  slist_append(&head, 3);
  slist_display(head);
    :
```

```
void slist_display(slist *head)
{
  slist *p = head;
  while(p != NULL){
    printf("%d ", p->value);
    p = p->next;
  }
  printf("¥n");
}
```

メモリ空間のイメージ

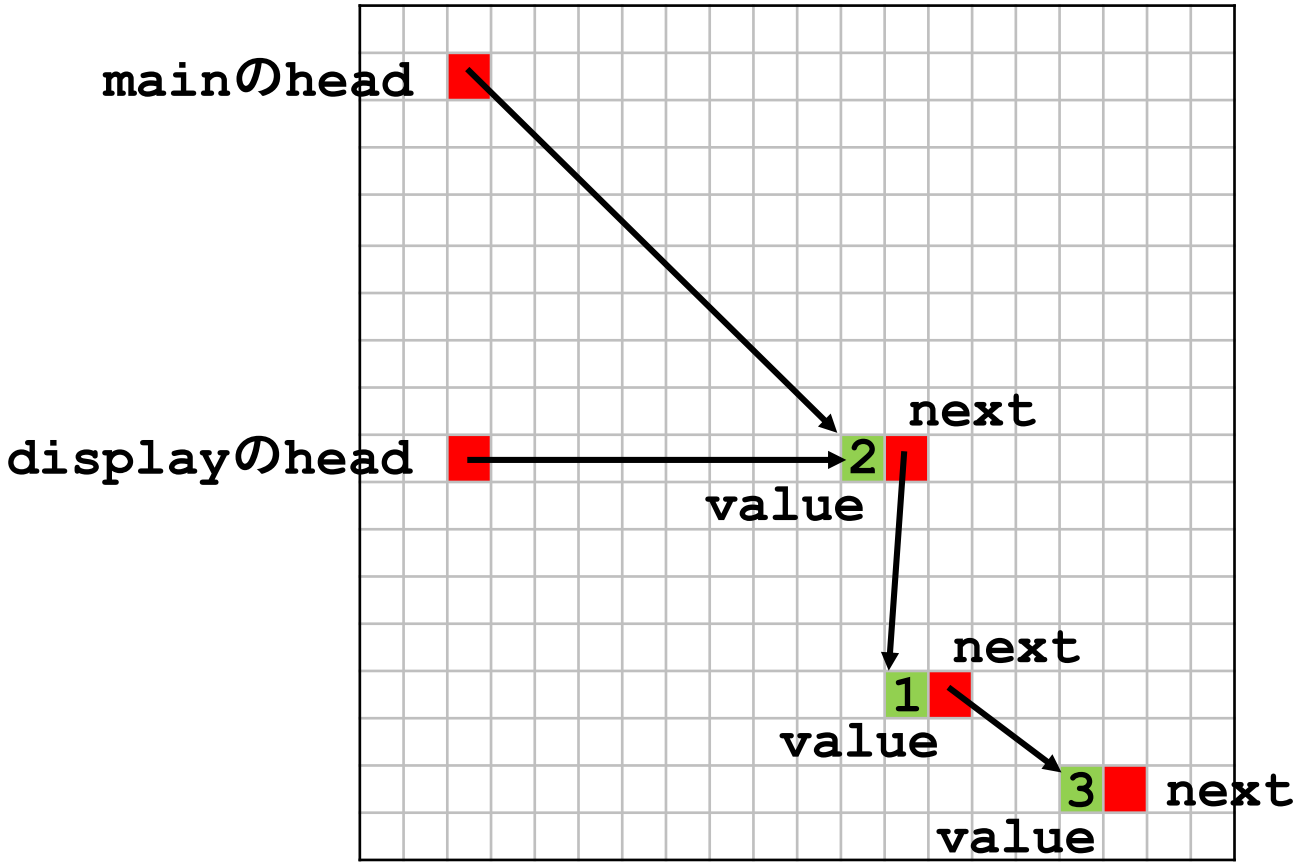


slist_display()のメモリ空間のイメージ

```
    :
  slist *head = NULL;
  slist_append(&head, 3);
  slist_display(head);
    :
```

```
void slist_display(slist *head)
{
  slist *p = head;
  while(p != NULL){
    printf("%d ", p->value);
    p = p->next;
  }
  printf("¥n");
}
```

メモリ空間のイメージ

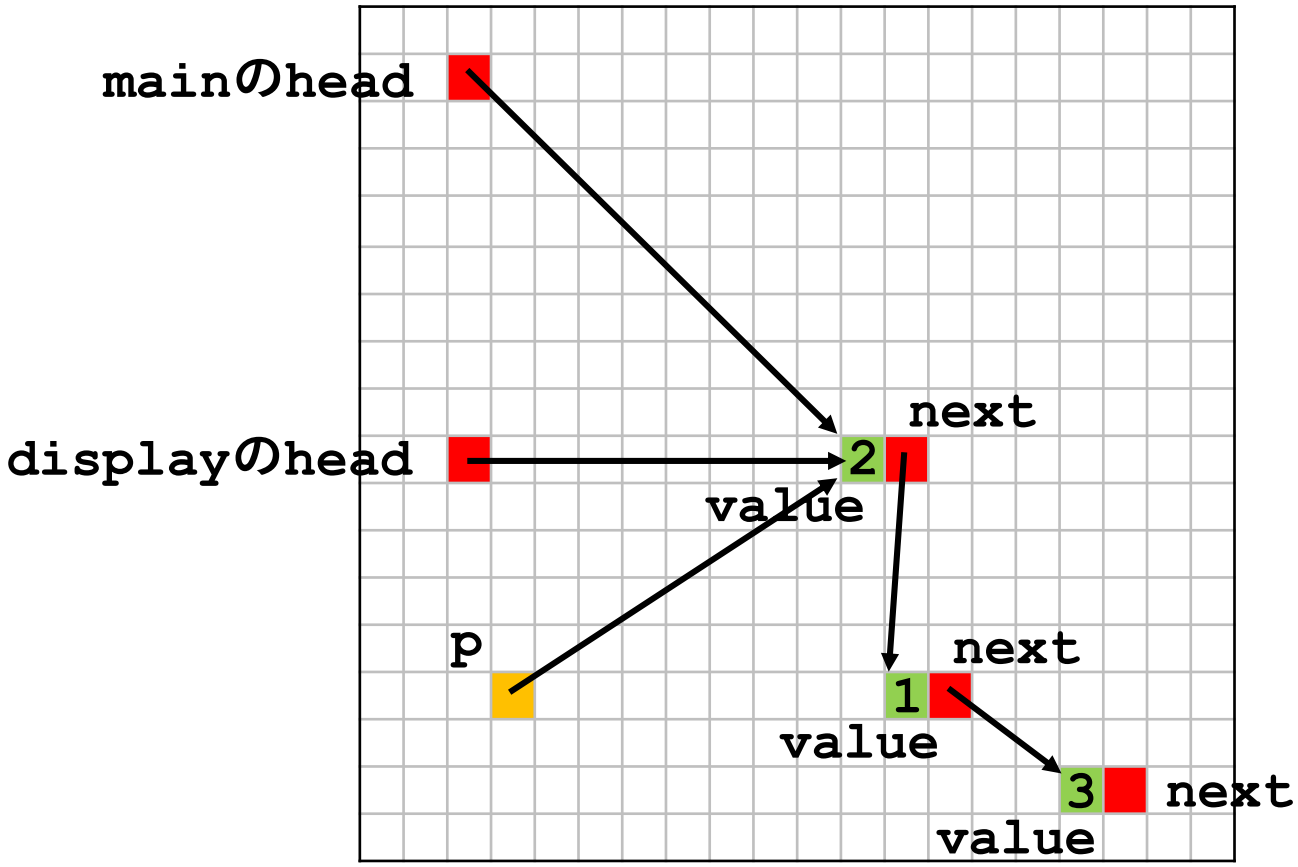


slist_display()のメモリ空間のイメージ

```
    :
  slist *head = NULL;
  slist_append(&head, 3);
  slist_display(head);
    :
```

```
void slist_display(slist *head)
{
  slist *p = head;
  while(p != NULL){
    printf("%d ", p->value);
    p = p->next;
  }
  printf("¥n");
}
```

メモリ空間のイメージ



slist_display()のメモリ空間のイメージ

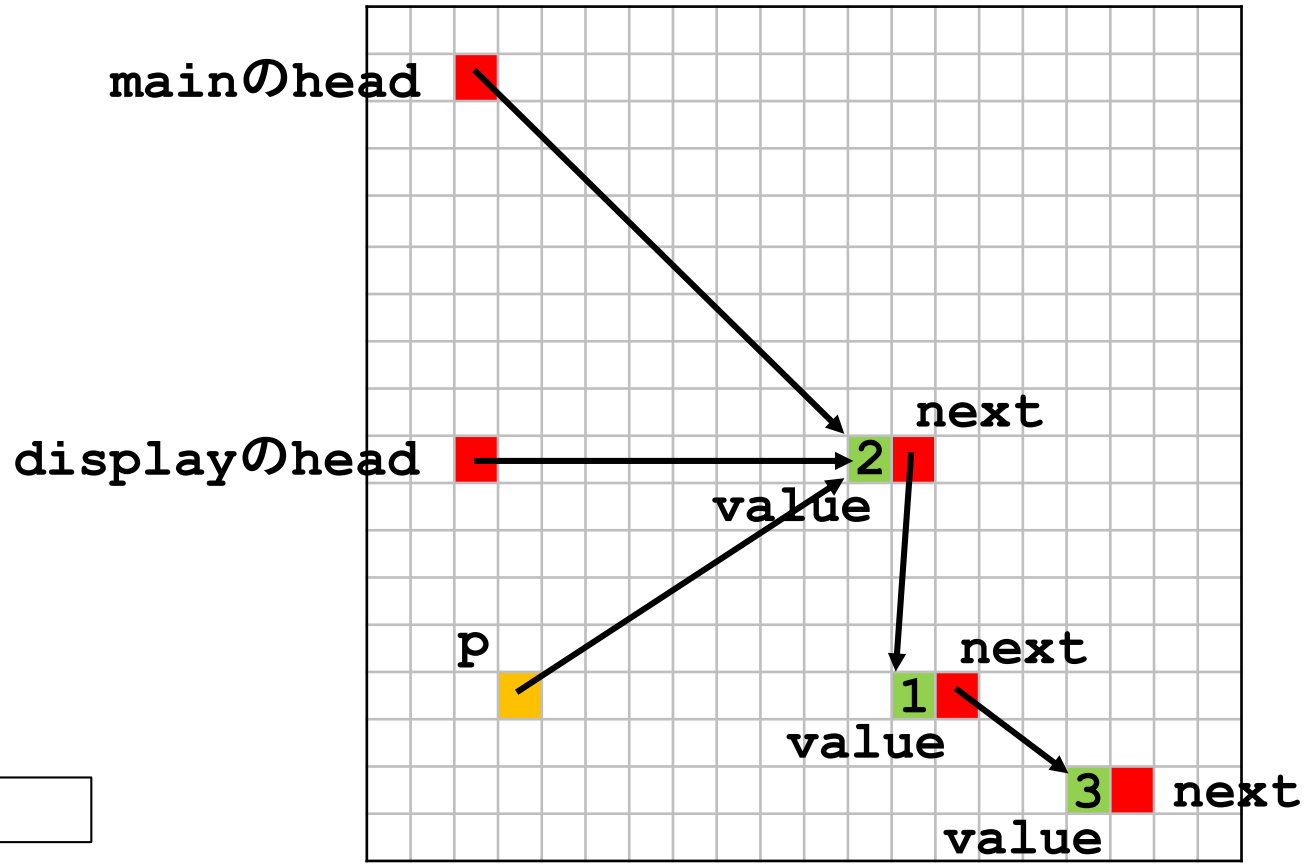
```
    :
  slist *head = NULL;
  slist_append(&head, 3);
  slist_display(head);
    :
```

```
void slist_display(slist *head)
{
  slist *p = head;
  while(p != NULL){
    printf("%d ", p->value);
    p = p->next;
  }
  printf("¥n");
}
```

出力

```
2
```

メモリ空間のイメージ



slist_display()のメモリ空間のイメージ

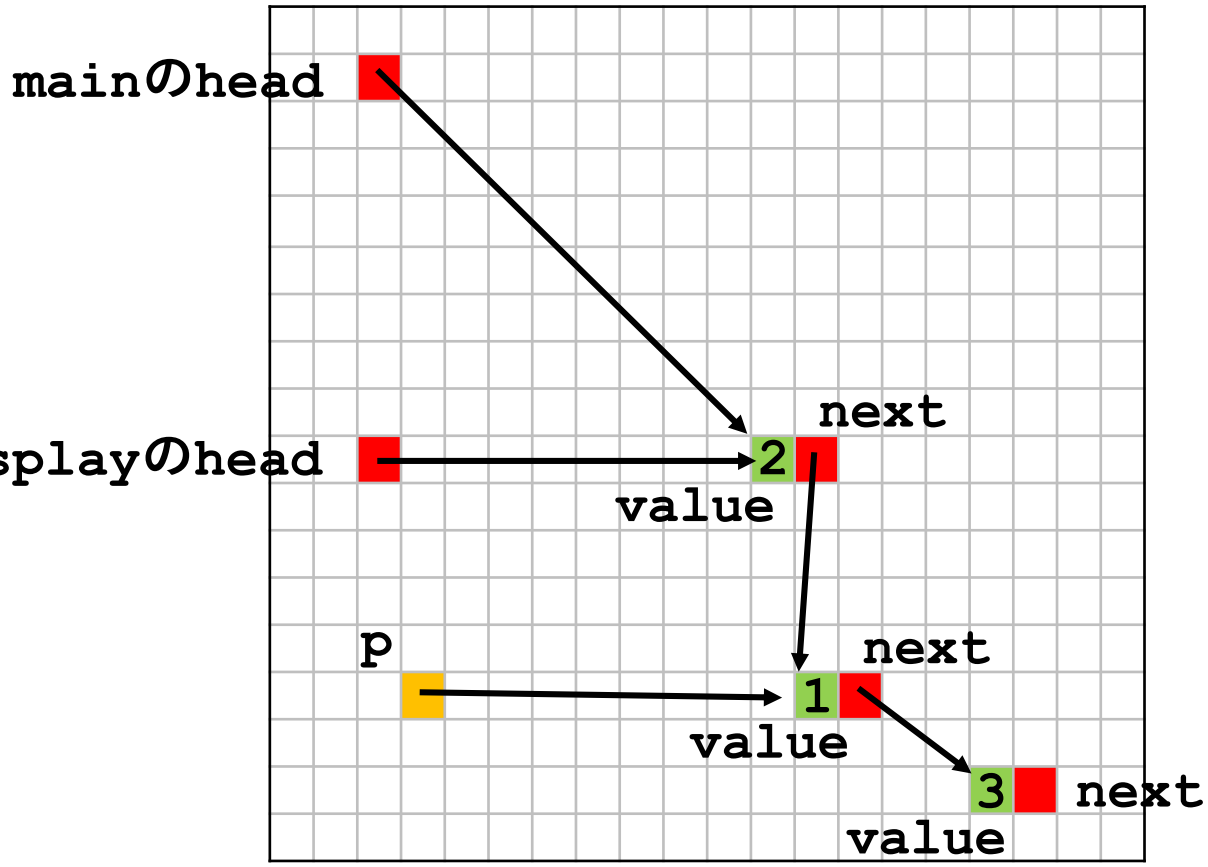
```
    :
  slist *head = NULL;
  slist_append(&head, 3);
  slist_display(head);
    :
```

```
void slist_display(slist *head)
{
  slist *p = head;
  while(p != NULL){
    printf("%d ", p->value);
    p = p->next;
  }
  printf("¥n");
}
```

出力

```
2
```

メモリ空間のイメージ



slist_display()のメモリ空間のイメージ

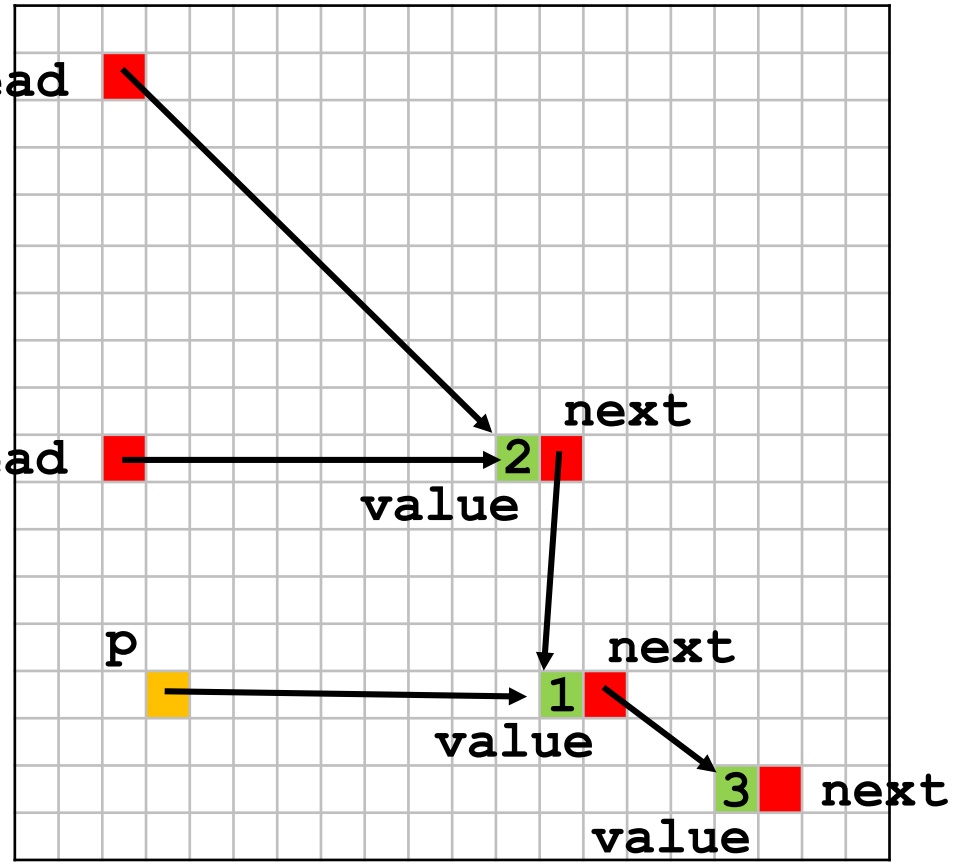
```
    :
  slist *head = NULL;
  slist_append(&head, 3);
  slist_display(head);
    :
```

```
void slist_display(slist *head)
{
  slist *p = head;
  while(p != NULL){
    printf("%d ", p->value);
    p = p->next;
  }
  printf("¥n");
}
```

出力

```
2 1
```

メモリ空間のイメージ



slist_display()のメモリ空間のイメージ

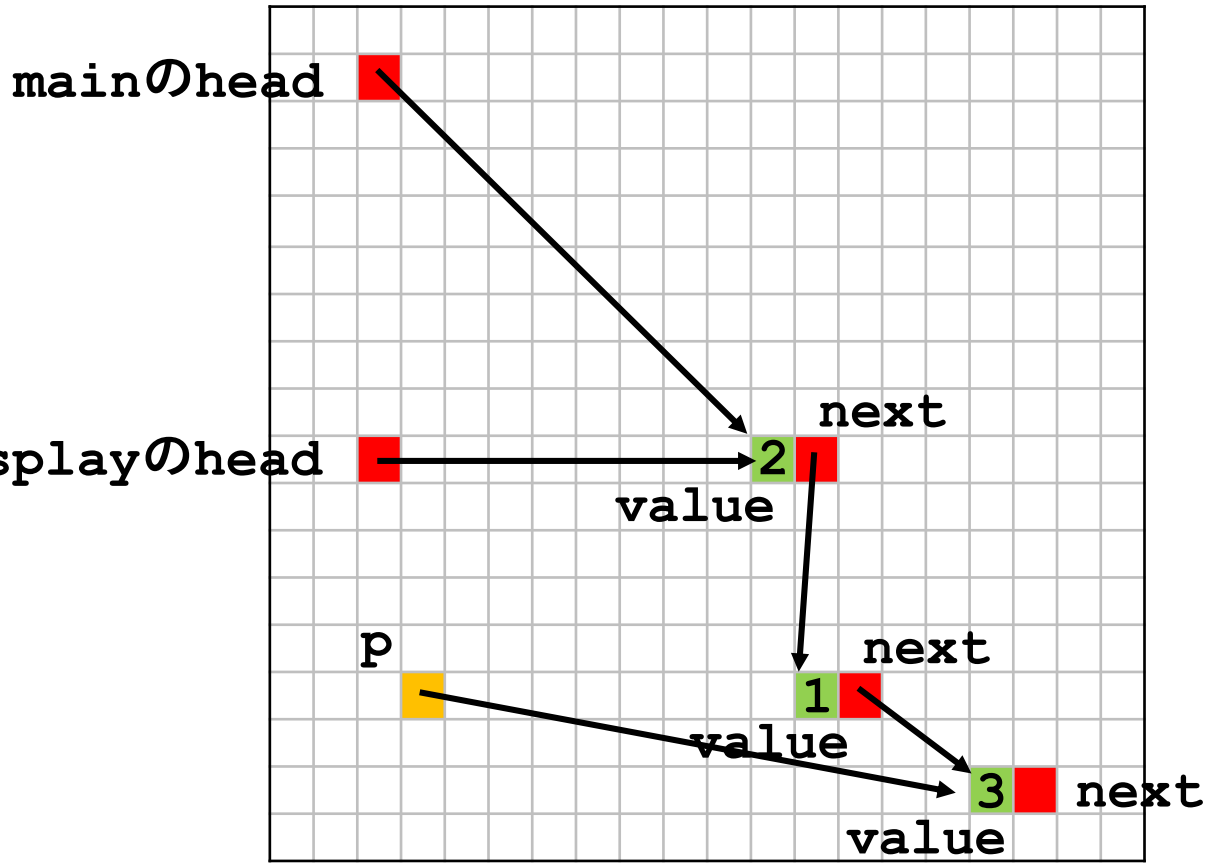
```
    :
  slist *head = NULL;
  slist_append(&head, 3);
  slist_display(head);
    :
```

```
void slist_display(slist *head)
{
  slist *p = head;
  while(p != NULL){
    printf("%d ", p->value);
    p = p->next;
  }
  printf("¥n");
}
```

出力

```
2 1
```

メモリ空間のイメージ



slist_display()のメモリ空間のイメージ

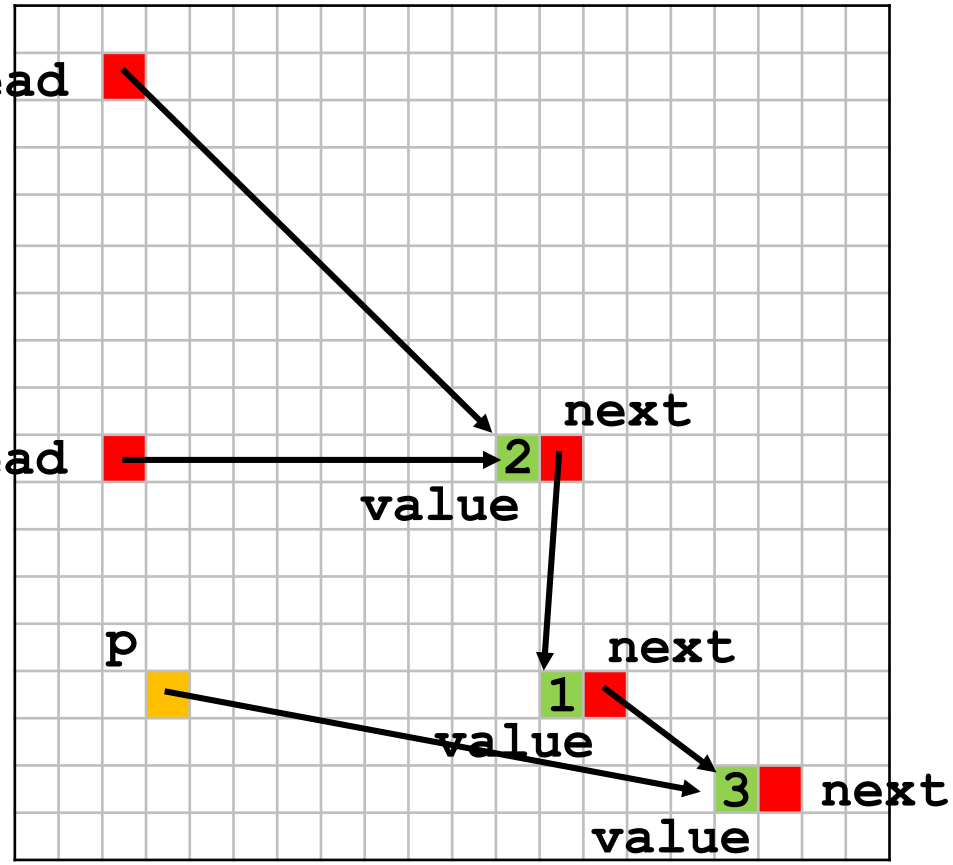
```
    :
  slist *head = NULL;
  slist_append(&head, 3);
  slist_display(head);
    :
```

```
void slist_display(slist *head)
{
  slist *p = head;
  while(p != NULL){
    printf("%d ", p->value);
    p = p->next;
  }
  printf("¥n");
}
```

出力

```
2 1 3
```

メモリ空間のイメージ



slist_display()のメモリ空間のイメージ

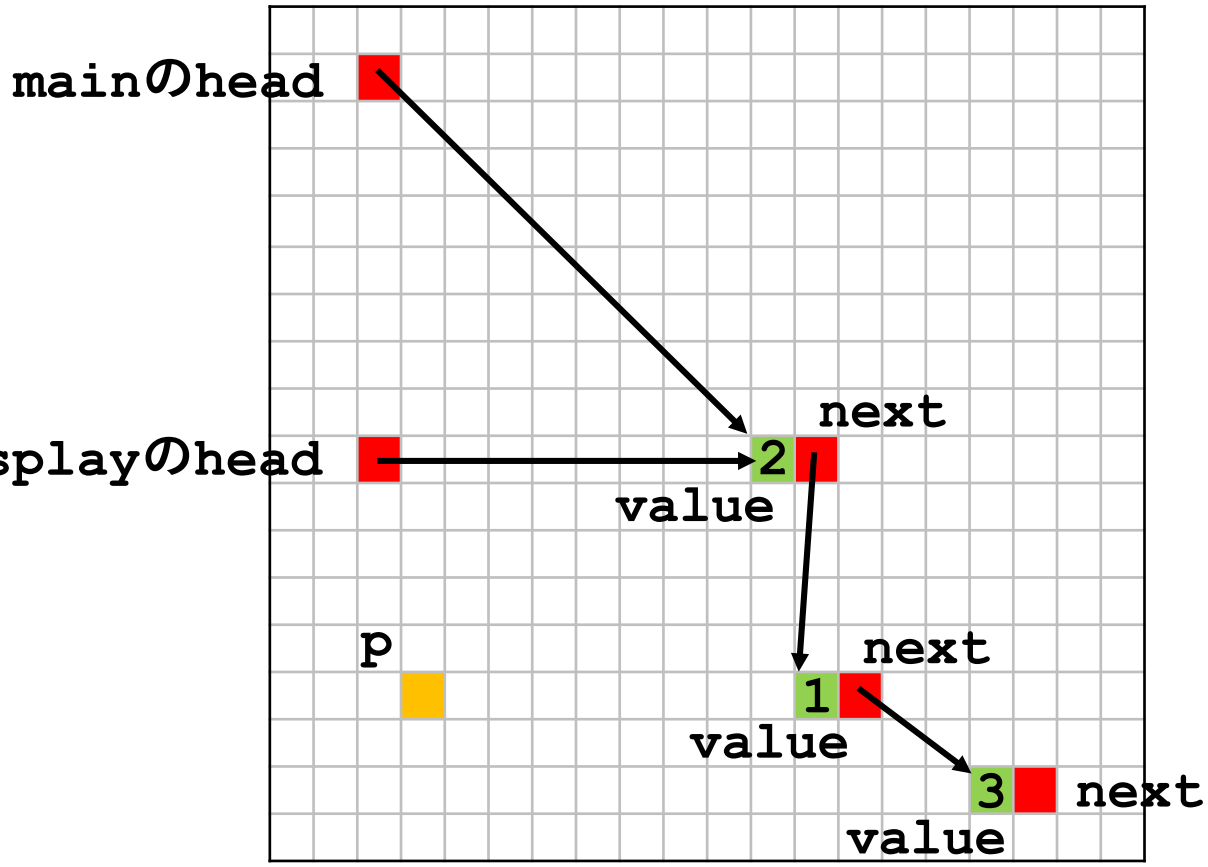
```
    :
  slist *head = NULL;
  slist_append(&head, 3);
  slist_display(head);
    :
```

```
void slist_display(slist *head)
{
  slist *p = head;
  while(p != NULL){
    printf("%d ", p->value);
    p = p->next;
  }
  printf("¥n");
}
```

出力

```
2 1 3
```

メモリ空間のイメージ



動的確保した要素を解放するslist_free()

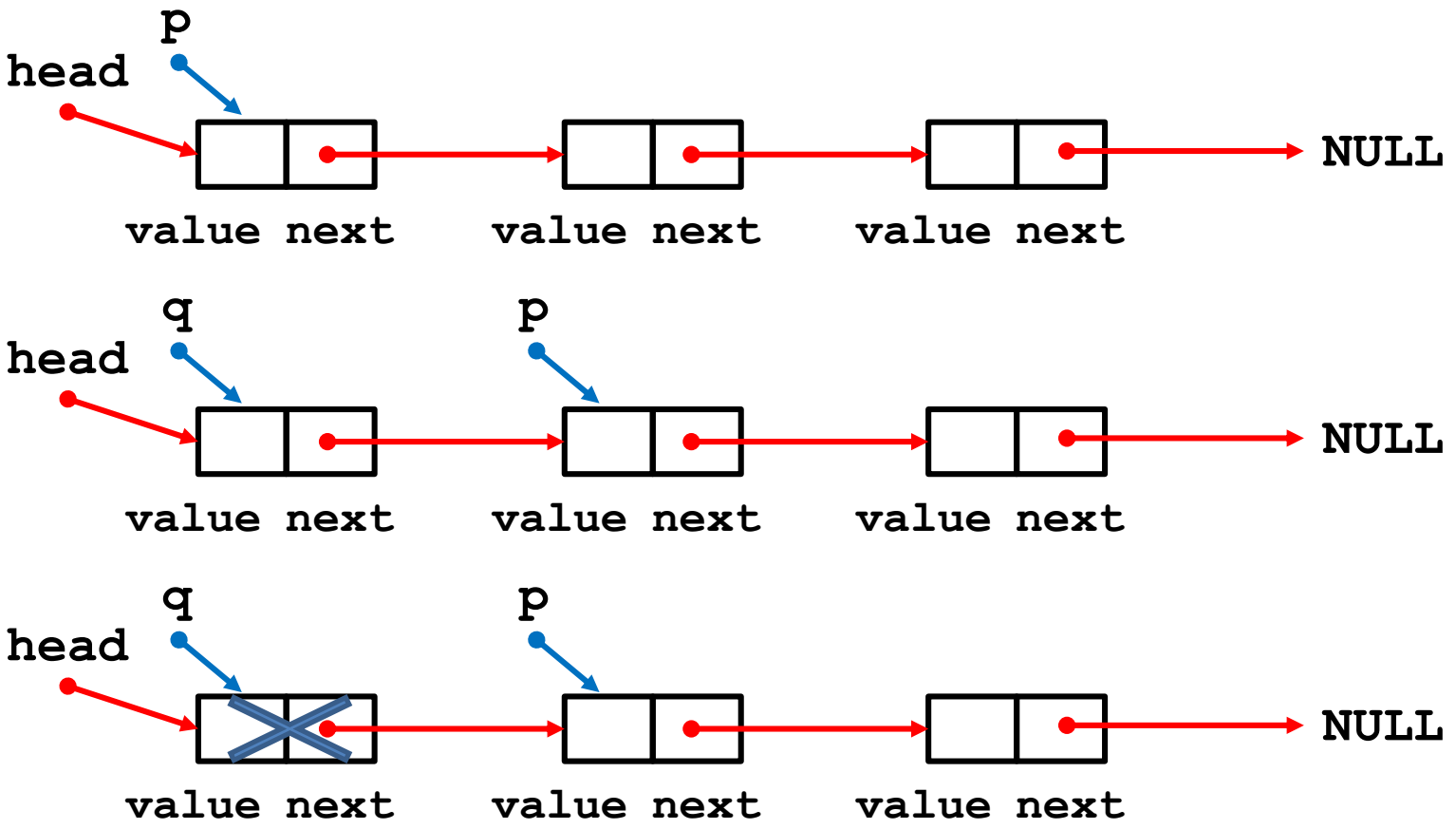
```
void slist_free(slist **head)
{
  slist *p = *head;
  slist *q;
  while(p != NULL){
    q = p;
    p = p->next;
    free(q);
    q = NULL;
  }
  *head = NULL;
}
```

リストのある要素へのポインタを指す変数p（先頭headから始める）

要素がNULLになるまで繰り返す

現在位置pをqに代入、 pには次の位置を代入、その後現在の要素を解放

headにNULLを入れて利用終了を明示（最初の状態に戻す。これがないと再度appendし直す際に躓く。）

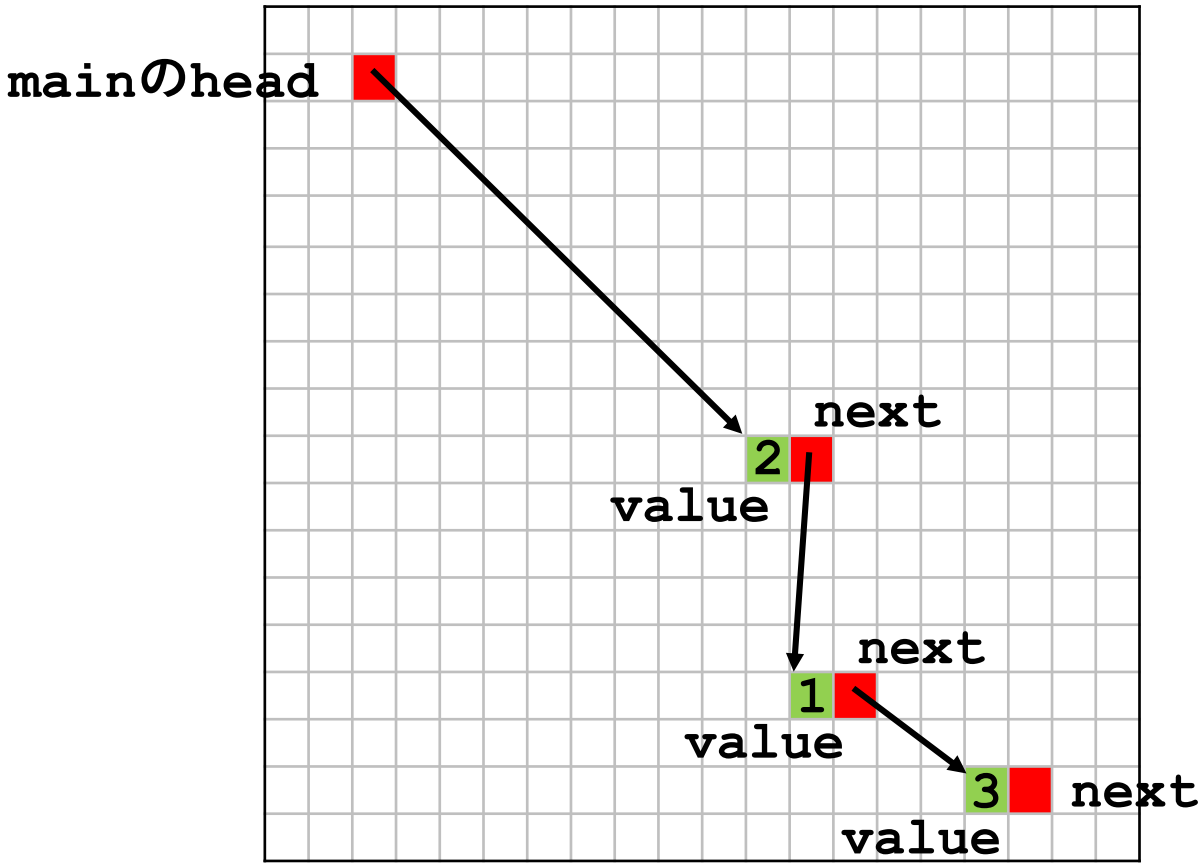


free()のメモリ空間のイメージ

```
    :
  slist_display(head);
  slist_free(&head);
  return 0;
    :
```

```
void slist_free(slist **head)
{
  slist *p = *head;
  slist *q;
  while(p != NULL){
    q = p;
    p = p->next;
    free(q);
    q = NULL;
  }
  *head = NULL;
}
```

メモリ空間のイメージ

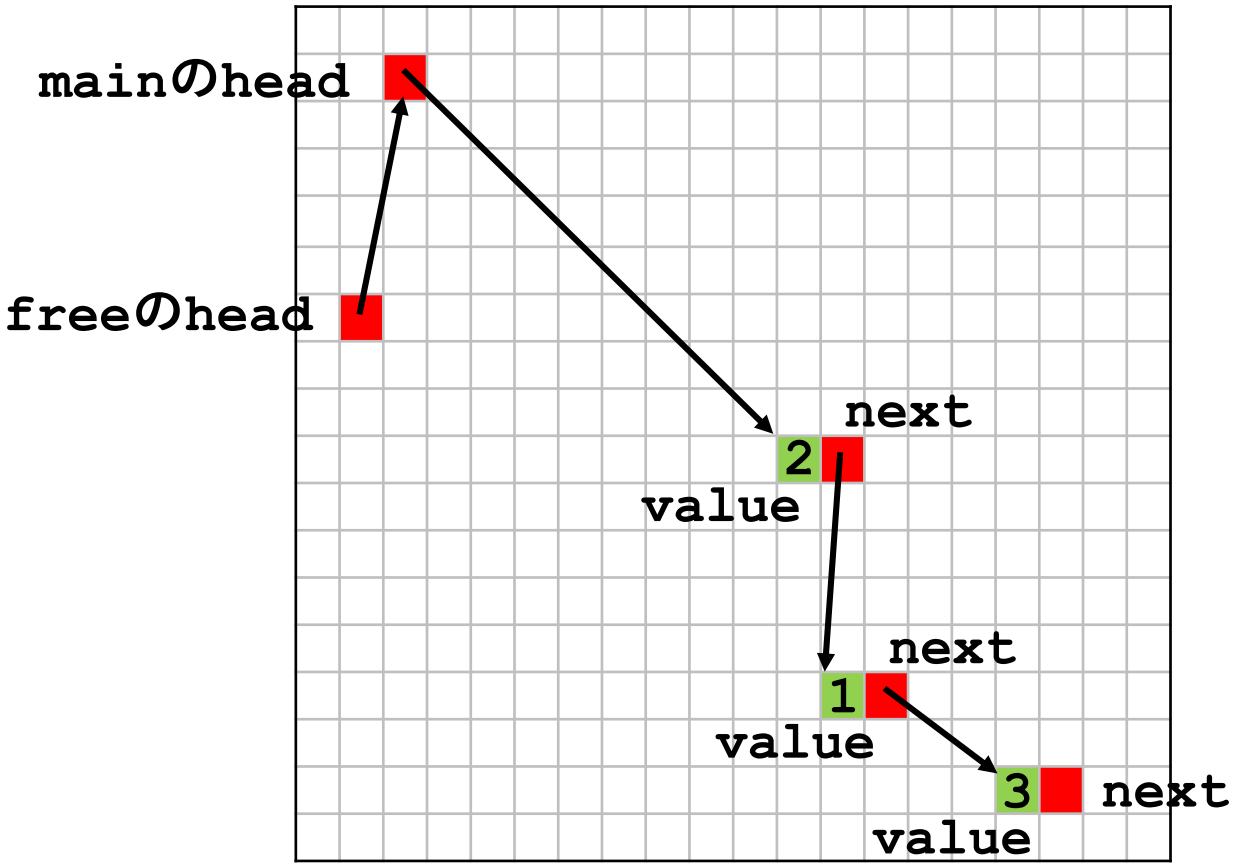


free()のメモリ空間のイメージ

```
    :
  slist_display(head);
  slist_free(&head);
  return 0;
    :
```

```
void slist_free(slist **head)
{
  slist *p = *head;
  slist *q;
  while(p != NULL){
    q = p;
    p = p->next;
    free(q);
    q = NULL;
  }
  *head = NULL;
}
```

メモリ空間のイメージ

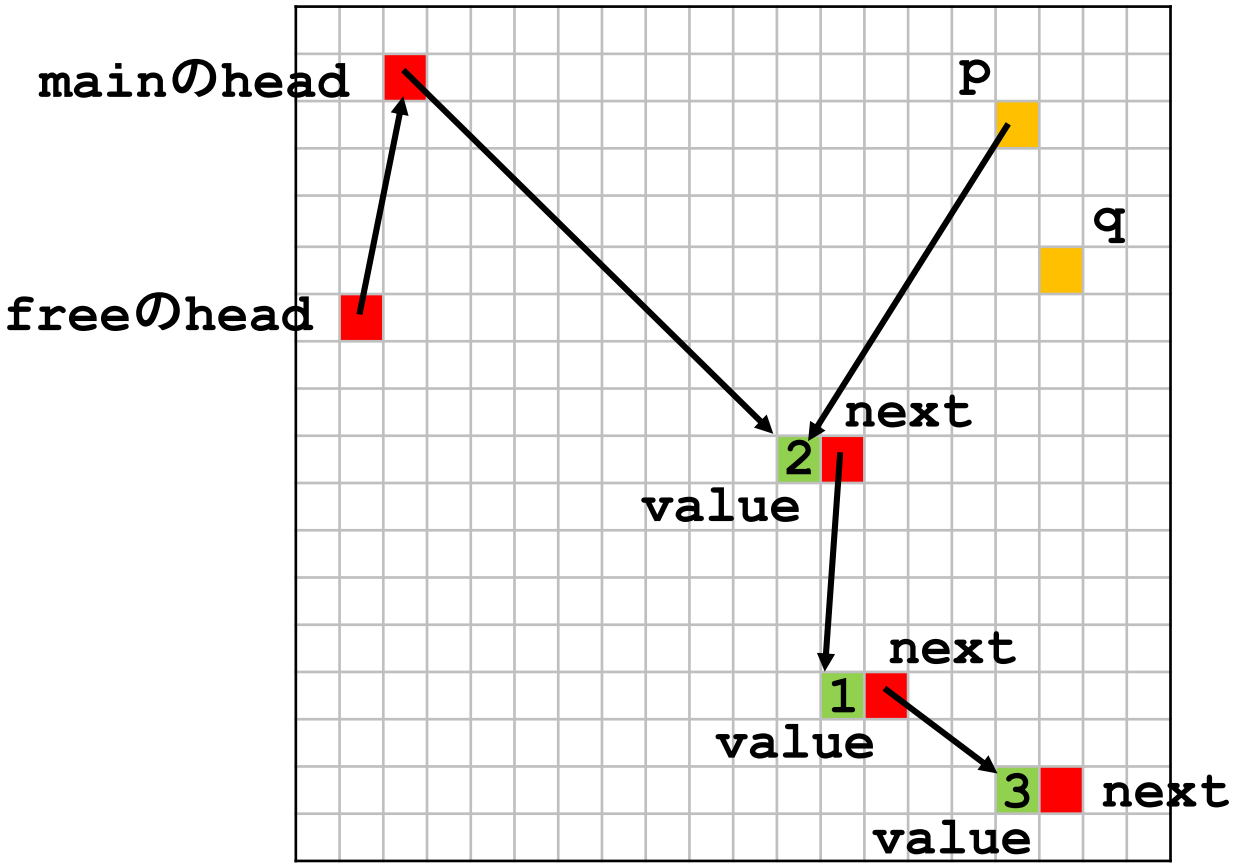


free()のメモリ空間のイメージ

```
    :
  slist_display(head);
  slist_free(&head);
  return 0;
    :
```

```
void slist_free(slist **head)
{
  slist *p = *head;
  slist *q;
  while(p != NULL){
    q = p;
    p = p->next;
    free(q);
    q = NULL;
  }
  *head = NULL;
}
```

メモリ空間のイメージ

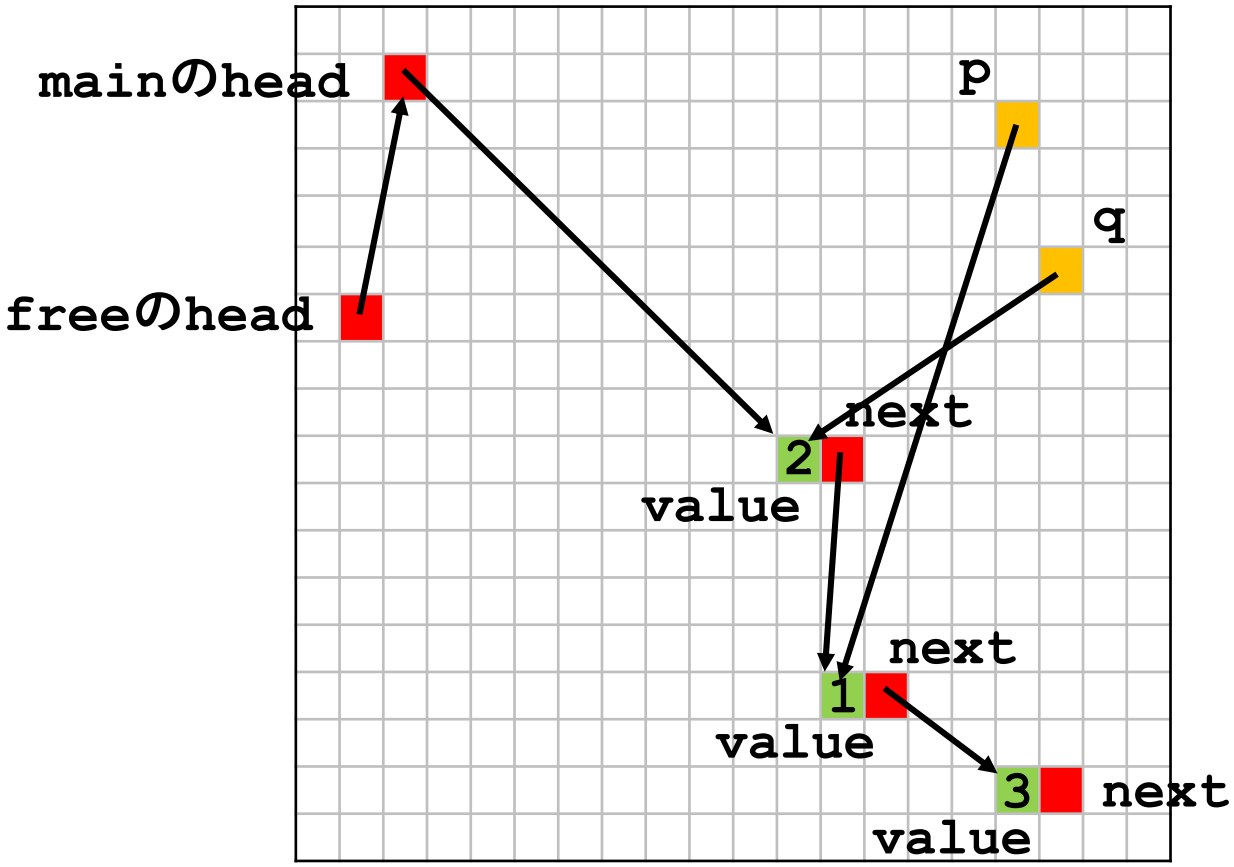


free()のメモリ空間のイメージ

```
    :
  slist_display(head);
  slist_free(&head);
  return 0;
    :
```

```
void slist_free(slist **head)
{
  slist *p = *head;
  slist *q;
  while(p != NULL){
    q = p;
    p = p->next;
    free(q);
    q = NULL;
  }
  *head = NULL;
}
```

メモリ空間のイメージ

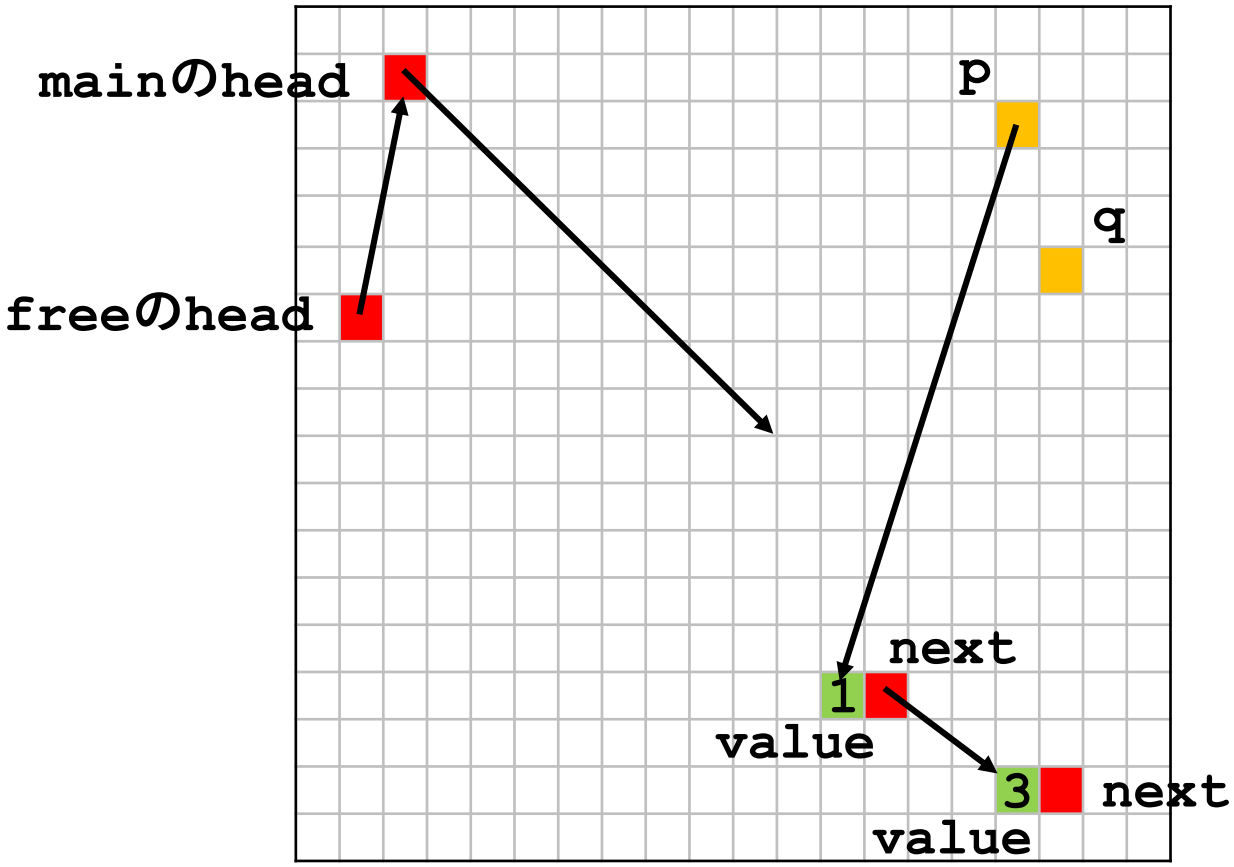


free()のメモリ空間のイメージ

```
    :
  slist_display(head);
  slist_free(&head);
  return 0;
    :
```

```
void slist_free(slist **head)
{
  slist *p = *head;
  slist *q;
  while(p != NULL){
    q = p;
    p = p->next;
    free(q);
    q = NULL;
  }
  *head = NULL;
}
```

メモリ空間のイメージ

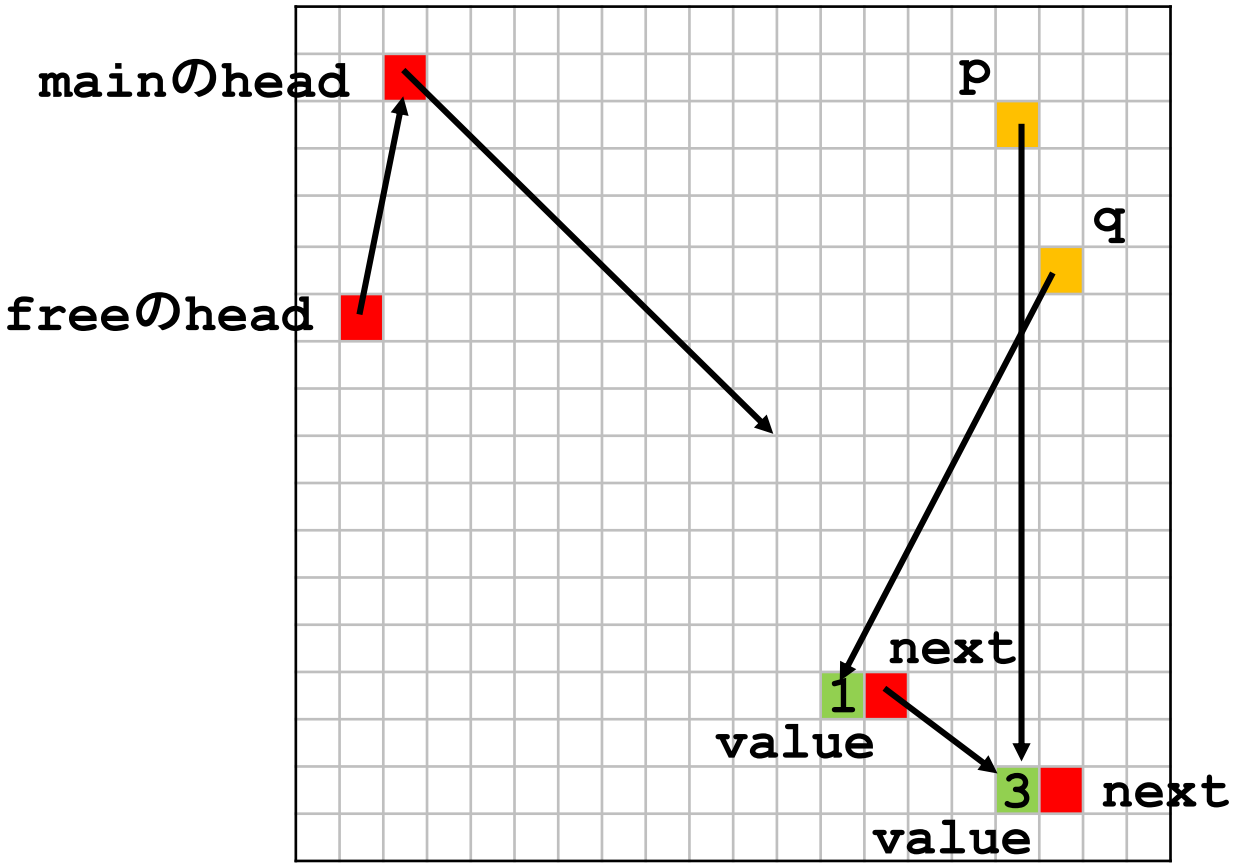


free()のメモリ空間のイメージ

```
    :
  slist_display(head);
  slist_free(&head);
  return 0;
    :
```

```
void slist_free(slist **head)
{
  slist *p = *head;
  slist *q;
  while(p != NULL){
    q = p;
    p = p->next;
    free(q);
    q = NULL;
  }
  *head = NULL;
}
```

メモリ空間のイメージ

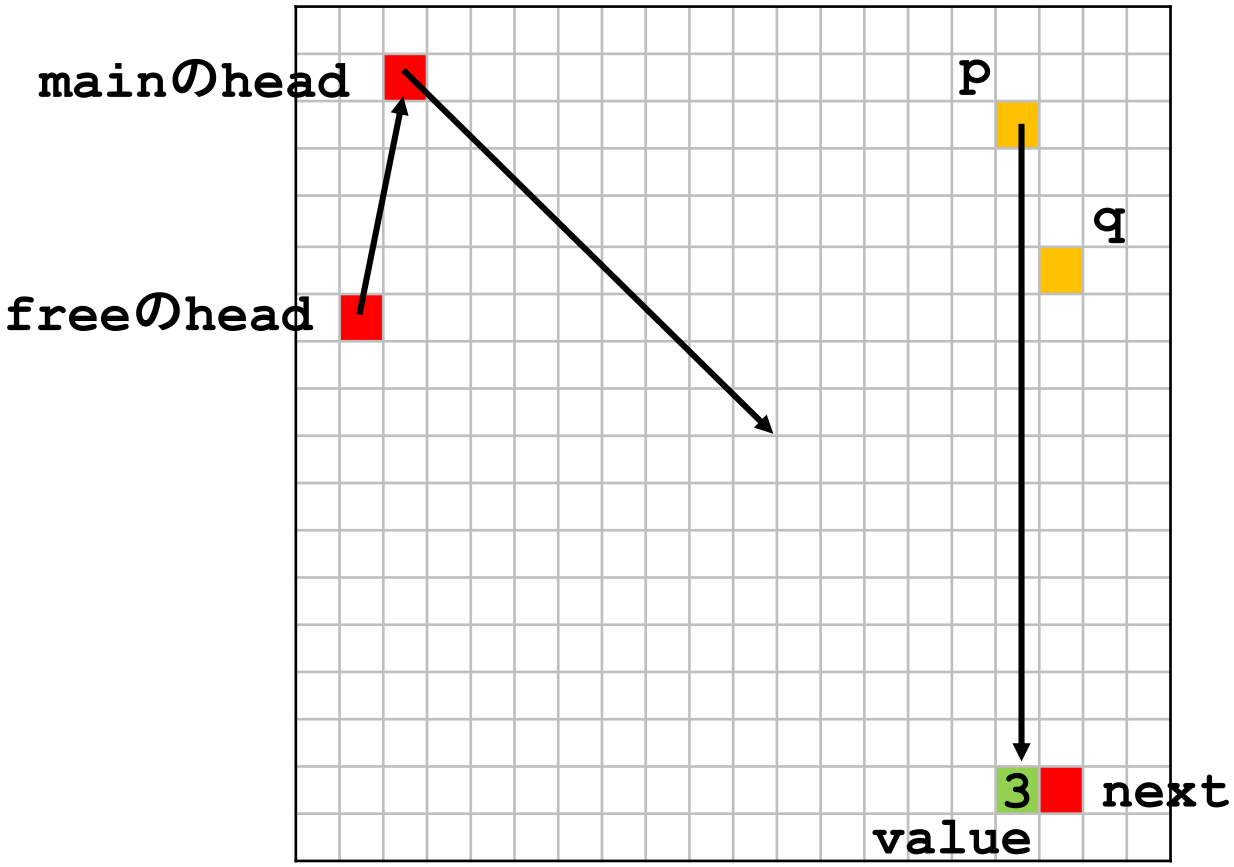


free()のメモリ空間のイメージ

```
    :
  slist_display(head);
  slist_free(&head);
  return 0;
    :
```

```
void slist_free(slist **head)
{
  slist *p = *head;
  slist *q;
  while(p != NULL){
    q = p;
    p = p->next;
    free(q);
    q = NULL;
  }
  *head = NULL;
}
```

メモリ空間のイメージ

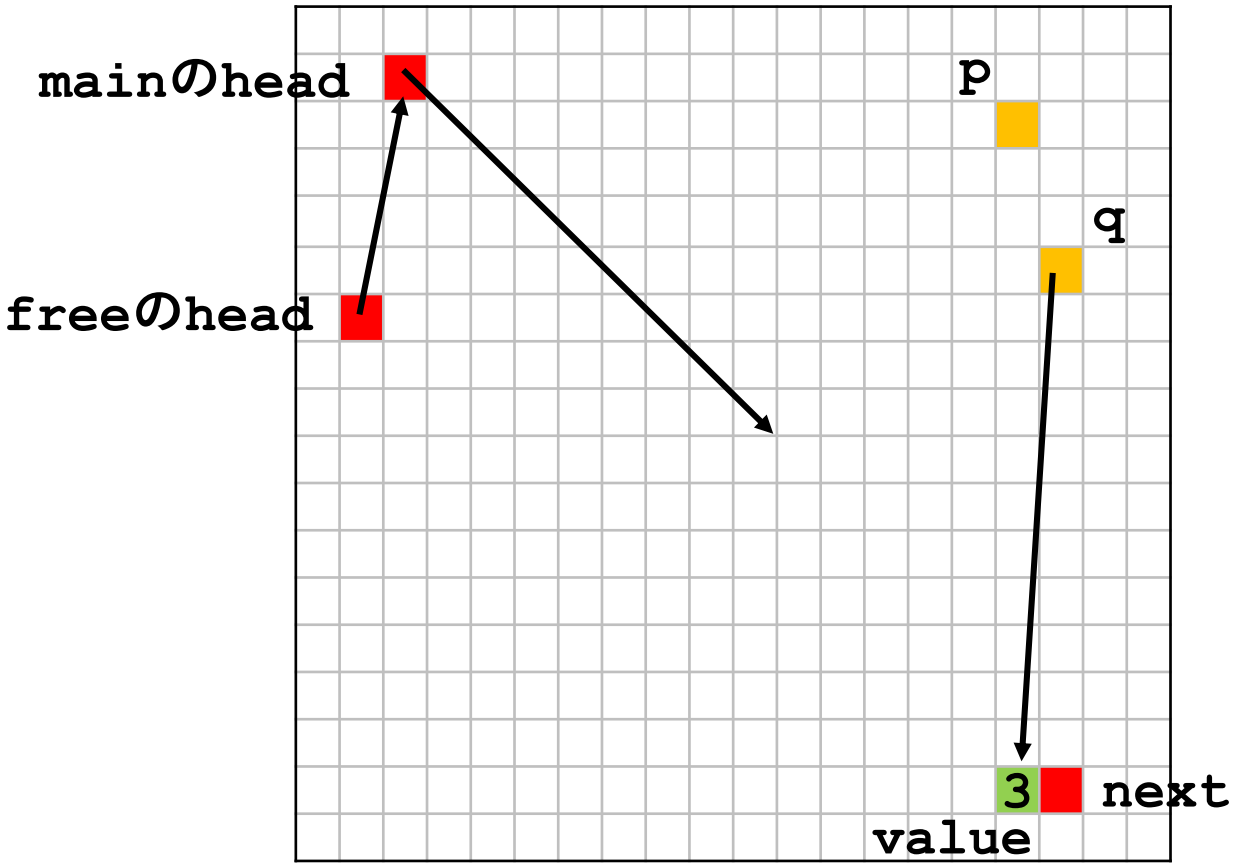


free()のメモリ空間のイメージ

```
    :
  slist_display(head);
  slist_free(&head);
  return 0;
    :
```

```
void slist_free(slist **head)
{
  slist *p = *head;
  slist *q;
  while(p != NULL){
    q = p;
    p = p->next;
    free(q);
    q = NULL;
  }
  *head = NULL;
}
```

メモリ空間のイメージ

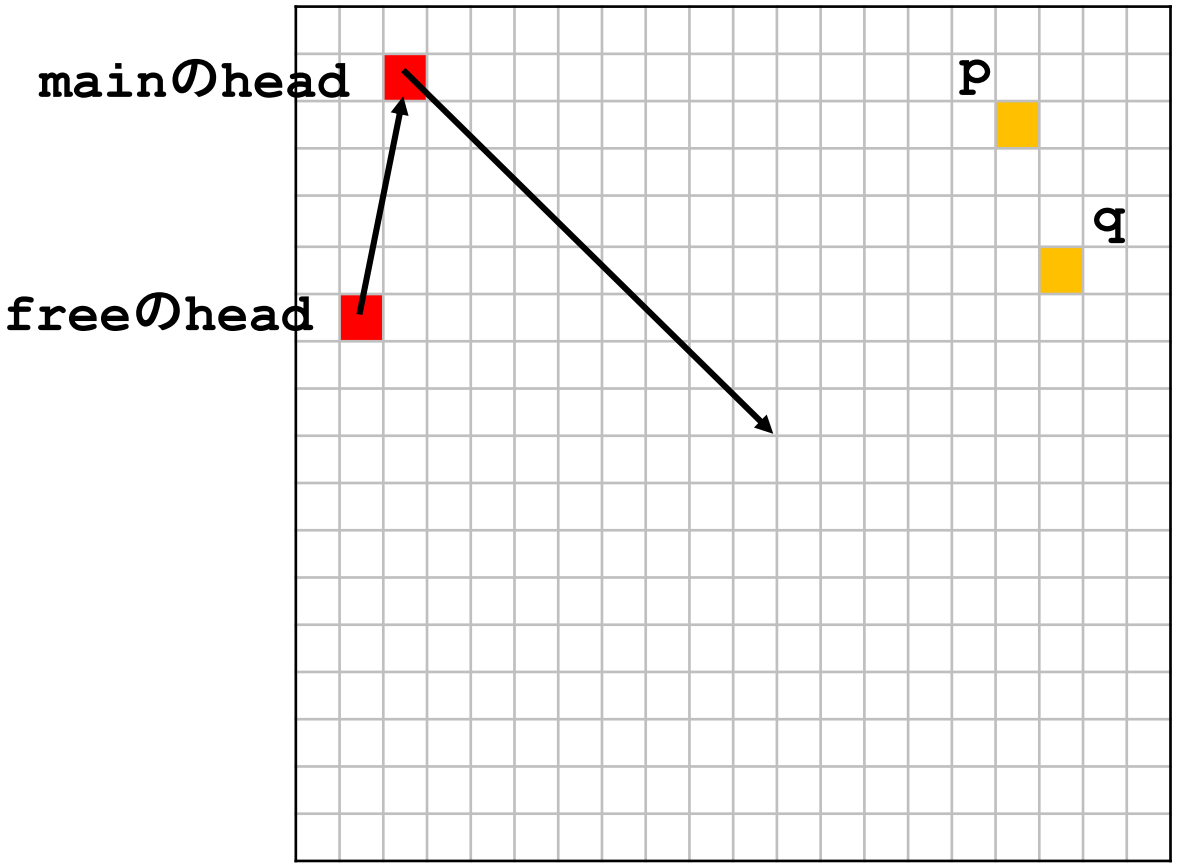


free()のメモリ空間のイメージ

```
    :
  slist_display(head);
  slist_free(&head);
  return 0;
    :
```

```
void slist_free(slist **head)
{
  slist *p = *head;
  slist *q;
  while(p != NULL){
    q = p;
    p = p->next;
    free(q);
    q = NULL;
  }
  *head = NULL;
}
```

メモリ空間のイメージ

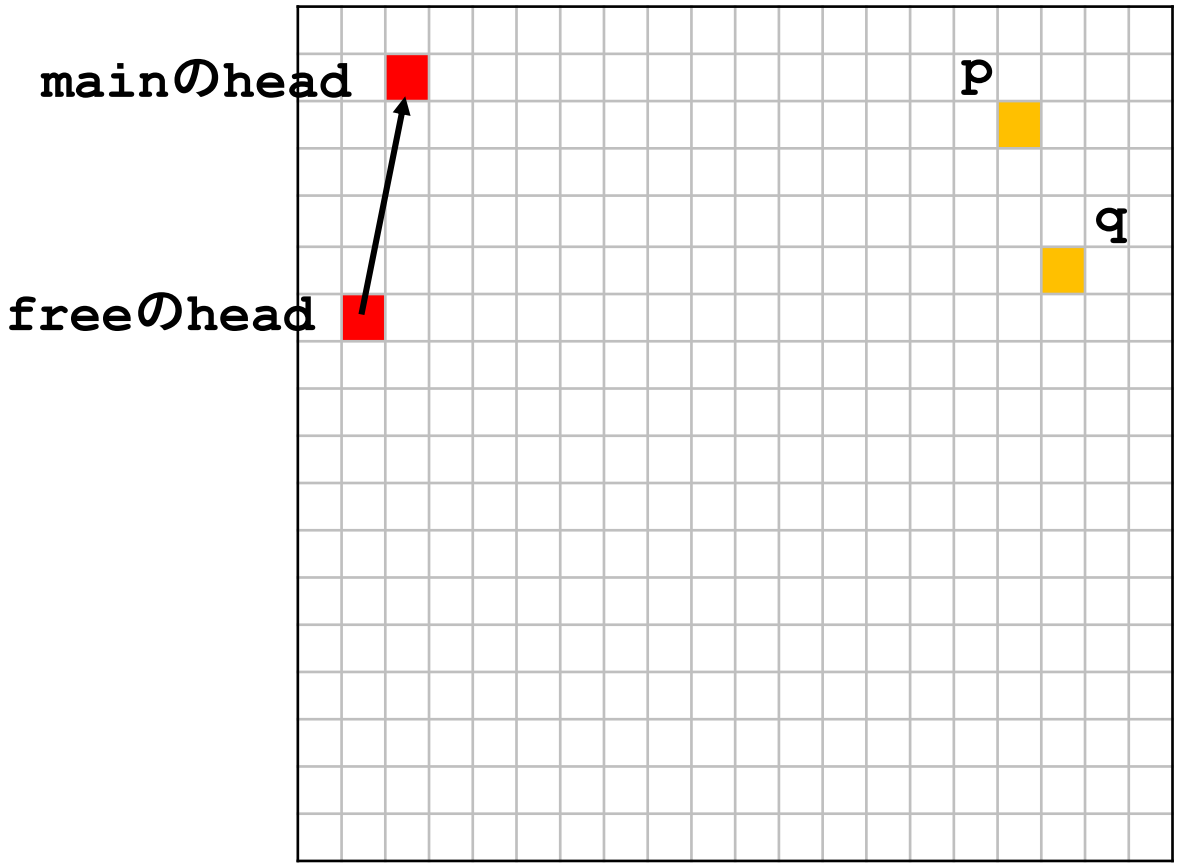


free()のメモリ空間のイメージ

```
    :
  slist_display(head);
  slist_free(&head);
  return 0;
    :
```

```
void slist_free(slist **head)
{
  slist *p = *head;
  slist *q;
  while(p != NULL){
    q = p;
    p = p->next;
    free(q);
    q = NULL;
  }
  *head = NULL;
}
```

メモリ空間のイメージ

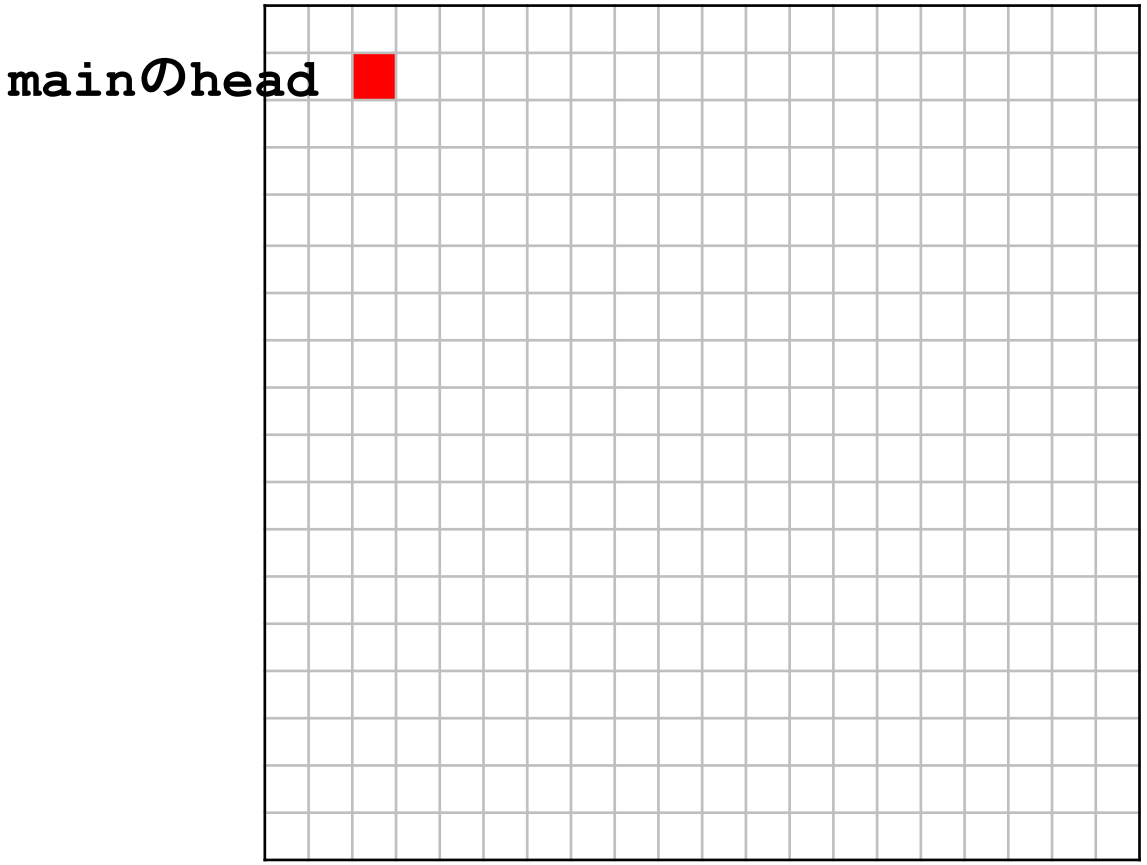


free()のメモリ空間のイメージ

```
    :
  slist_display(head);
  slist_free(&head);
  return 0;
    :
```

```
void slist_free(slist **head)
{
  slist *p = *head;
  slist *q;
  while(p != NULL){
    q = p;
    p = p->next;
    free(q);
    q = NULL;
  }
  *head = NULL;
}
```

メモリ空間のイメージ



第 6 回 演習

EX-6.1

- scanf関数で入力された文字列を、リストに辞書順に追加やリストから削除するプログラムを作成する
  - 具体的には、add_word関数、remove_word関数を作成する
    - 通常のappendのようにリスト末尾ではなく、単語を辞書順に追加するので注意
  - scanfで読み込まれた文字列が、
    - “a:XXXXX” なら、add_word関数で文字列XXXXXを辞書順にリストに追加
    - “r:YYYYY” なら、remove_word関数で文字列YYYYYと一致するノードをリストから削除
      - 一致するノードが複数ある場合、消すのは一つで良い
    - “e” ならプログラム終了
  - 文字列の比較にはstrcmp関数を使う
    - strcmp(s1,s2)と呼び出したときの戻り値は、文字列s1が文字列s2よりアルファベット順で後ならば正の値（並び替え対象）、同じならば0となる。

ひな形と実行例

- ひな形はTACTにあります
- scanfの入力は、最初の一文字が’a’, ‘r’, ‘e’のどれかであることを仮定していて、3文字目からが登録or削除する単語であると仮定した実装になってます
- display_listとfree_listは実装済みです

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct word_list{
 char word[40];
 struct word_list *next;
};

void display_list(struct word_list *dictionary);
void add_word(struct word_list **dictionary, char* word);
void remove_word(struct word_list **dictionary, char* word);
void free_list(struct word_list **dictionary);

int main(){

 struct word_list *dictionary = NULL;
 char word[42], op;
 while(1){
  printf("input op & word\n");
  scanf("%s",word);
  op = word[0];
  if(op == 'a'){
   add_word(&dictionary, &word[2]);
  }else if(op == 'r'){
   remove_word(&dictionary, &word[2]);
  }else if(op == 'e'){
   break;
  }else{
   printf("unknown operation\n");
  }
  display_list(dictionary);
 }
 free_list(&dictionary);

 return 0;
}
```

実行例

- 先頭に追加や削除を行う場合にバグりやすいので注意
- 先頭・途中・末尾で正しいかデバッグしてみると良い

```
$ $ ./a.out
input op & word
a:pen

dictionary: pen
input op & word
a:pineapple

dictionary: pen pineapple
input op & word
a:apple

dictionary: apple pen pineapple
input op & word
a:pen

dictionary: apple pen pen pineapple
input op & word
r:pen

dictionary: apple pen pineapple
input op & word
r:apple

dictionary: pen pineapple
input op & word
r:pineapple

dictionary: pen
input op & word
r:pen

dictionary:
input op & word
e
```

入力の省力化

- （全ての環境でできるかわからないが）scanfに入力する文字列をあらかじめファイルに書いておき、実行時に読み込ませることができる

input.txt

```
a:pen
a:pineapple
a:apple
a:pen
r:pen
r:apple
r:pineapple
r:pen
e
```

```
$ ./a.out < input.txt
input op & word


dictionary: pen
input op & word


dictionary: pen pineapple
input op & word


dictionary: apple pen pineapple
input op & word


dictionary: apple pen pen pineapple
input op & word


dictionary: apple pen pineapple
input op & word

…
```

提出

- 演習課題EX-6.1
- TACTでソースファイルと実行結果（入力+画面に出力されたものをコピー＆ペーストしたテキストファイル）を提出
  - ソースファイルの名前は　学籍番号-6-{課題番号}.c　とすること
    - 例：252301111-6-1.c
  - 結果ファイルの名前は　学籍番号-6-{課題番号}.txt　とすること
    - 例：252301111-6-1.txt
  - 実行可能ファイル（gccにより生成されたファイル）の提出は不要
  - 1課題分計2ファイル提出
- 提出期限
  - **次々回（1/30）講義開始前まで**