アルゴリズム論
（第10回）

岩手県立大学
Iwate Prefectural University

佐々木研（情報システム構築学講座）
講師　山田敬三
k-yamada@iwate-pu.ac.jp

ハッシュ法

岩手県立大学
Iwate Prefectural University

ハッシュ テーブル

- 対象データを非常に効率よく格納し、検索する方法
- 対象データはレコード
  - レコードは互いに異なるキーを含む
- キーを適当な範囲の自然数に変換

ハッシュ テーブル

- キーの値が、レコードの位置に対応していれば、検索が速い

  a[0], a[1], ・・・, a[n-2], a[n-1]

  に，レコードのキーiに対して，a[i]にデータを格納していればよい

- しかし，iが配列のサイズよりも大きい場合や，キーが自然数でない場合は，この方法は不可能

- そこで、キーを適当な自然数に変換する関数を考える
  ⇒ハッシュ関数

ハッシュ テーブル

- ハッシュ関数Hを適用し、一次インデックス値を得る。
  - 一次インデックスは要素数Nよりも小さい非負の整数値
- 一次インデックスとキーは1対1であることが望ましいが，ここでは問わない（難しい）
- H(k1)=H(k2) となる状況を衝突(collision)というP.86のプログラムを参照
  - ここでは、先頭文字と最終文字と長さに依存しているため，同じハッシュ値になるものは簡単に見つかる
  - “ABC” と “AZC”は同じハッシュ値になる

ハッシュ テーブル

- 同じハッシュに対応するための方法
  - オープンアドレス法と連結法
- ここでは、オープンアドレス法を用いる
  - 与えられた配列にすべてのデータを格納する
  - 連結法では、レコードを多数の線形リストとして格納する
- 衝突した場合、
  - つまり、同じ場所にすでにデータが入っているので、データを格納できない
  - 次の配列番号の配列要素にデータを格納する

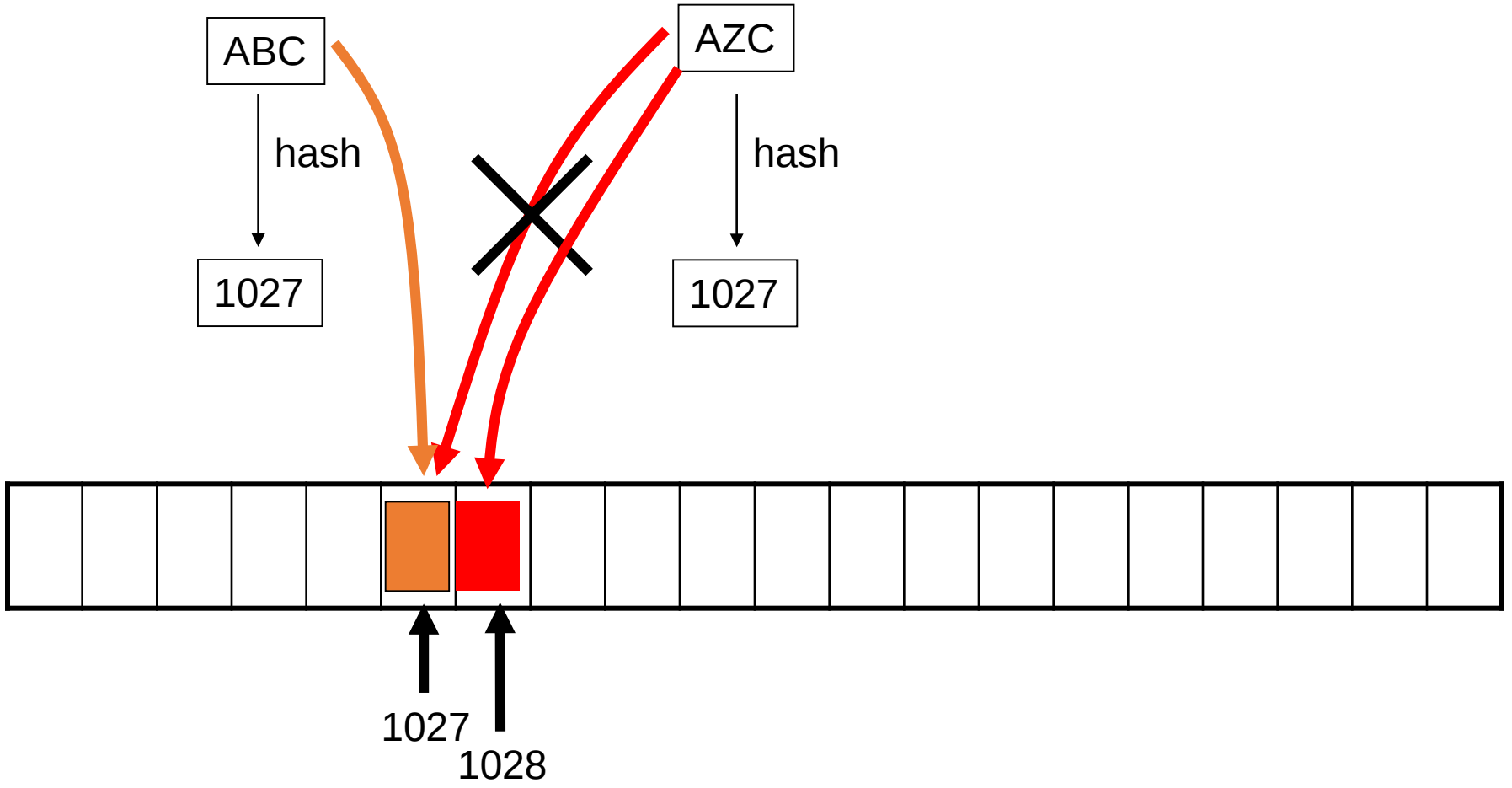
    i+1 (i<Nのとき)または

    0 (i=N-1のとき)

    を繰り返す。

  ⇒線形探査法(linear probing)
- P.88のプログラムを参照

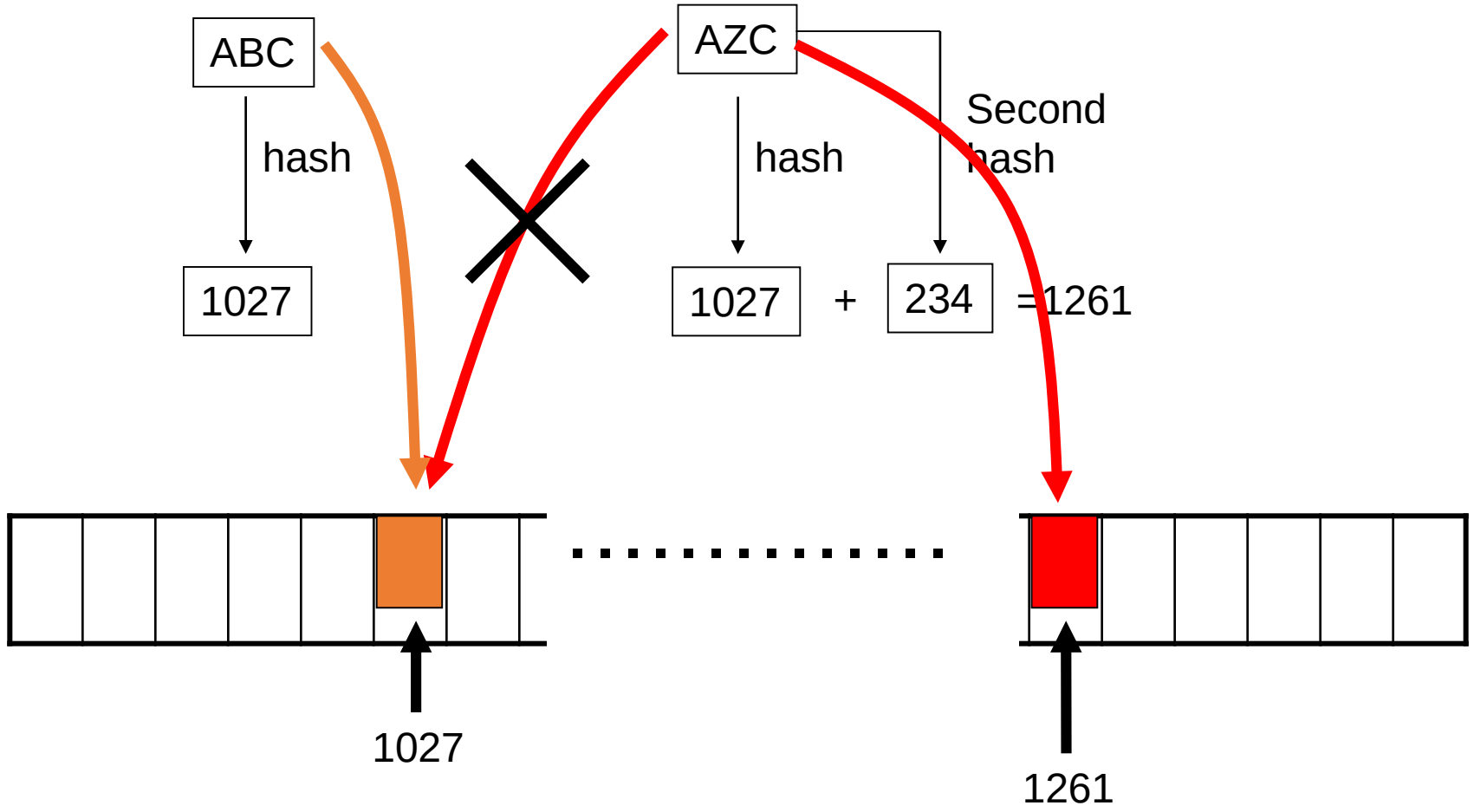
線形探査法



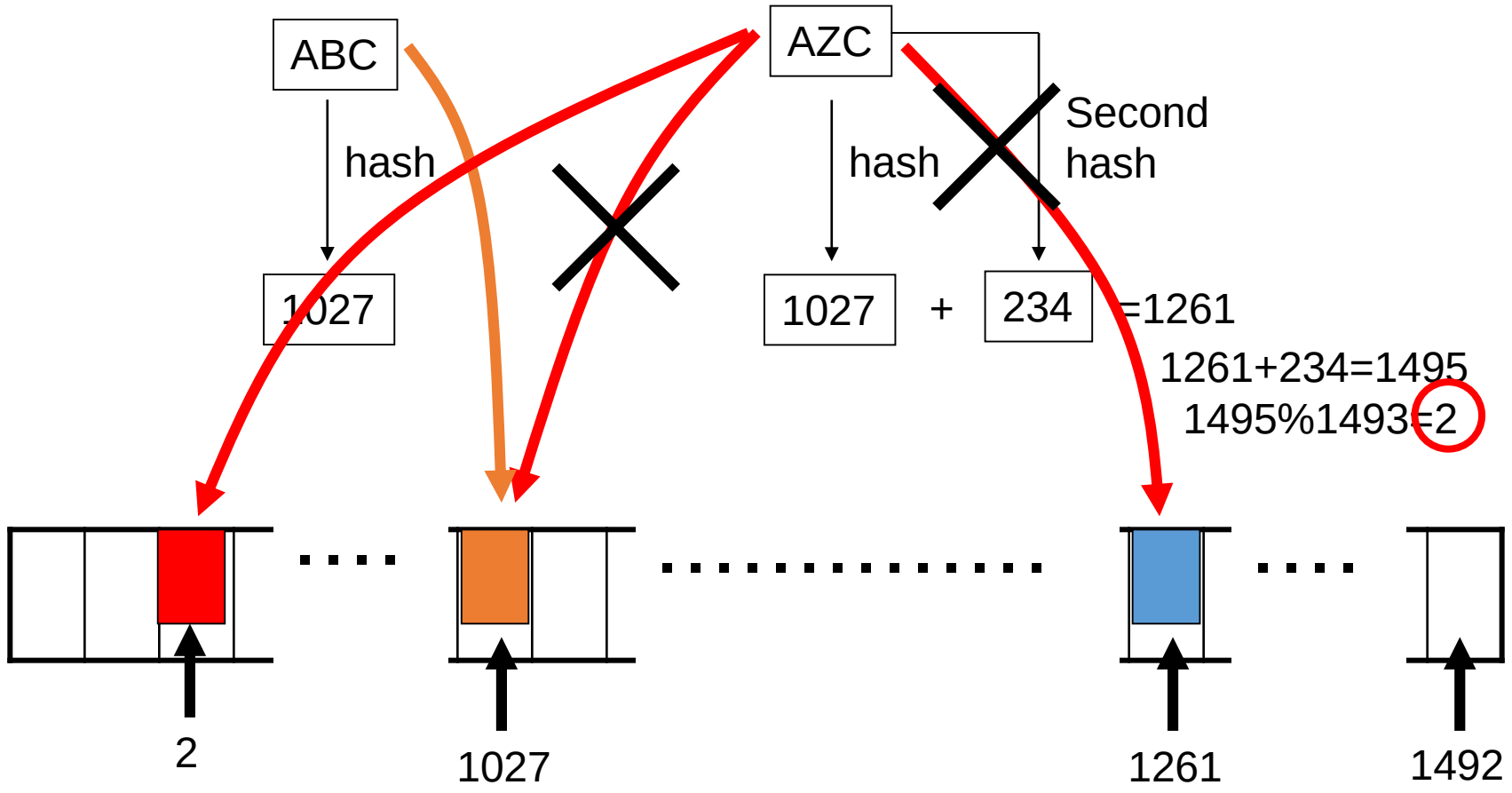
ハッシュ テーブル

- 線形探査法は単純で魅力的！
- しかし、同じハッシュが続くと効率が悪くなる。
- 線形探索法は1つずつ配置をずらすために効率が悪い。
- 衝突が起こったときに、増分を別のハッシュ関数で求める。
  - 増分を用いて循環的に足し算する。
    ⇒ ダブルハッシュ法
- 循環的に足し算するので、配列の大きさNが素数で無い場合は、元の場所に戻る可能性がある。
- 増分incはNよりも小いとしても一般性を失わない
  inc<N

ダブルハッシュ法



ダブルハッシュ法


ABC
hash
1027
AZC
hash
Second
hash
1027
+
234
=1261
1261+234=1495
1495%1493=2
2
1027
1261
1492

ハッシュ テーブル

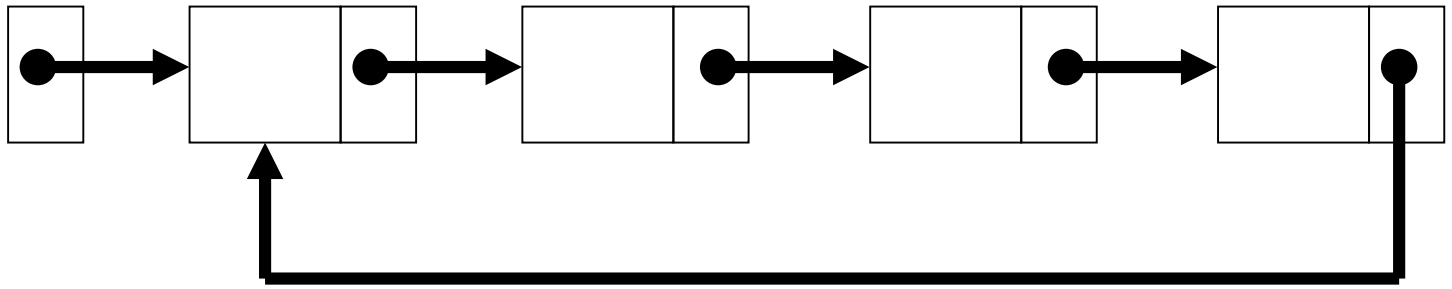
- P.90～92のプログラムを参照
- ダブルハッシュ法はデータ（レコード）の数よりも十分大きな配列（空間）を持つことが望ましい

循環・重連結リスト

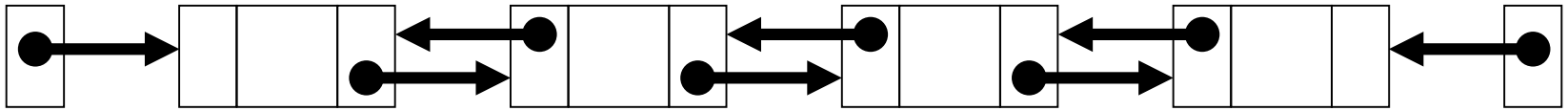
岩手県立大学
Iwate Prefectural University

循環リストと重連結リスト

- 循環リスト（circular list）

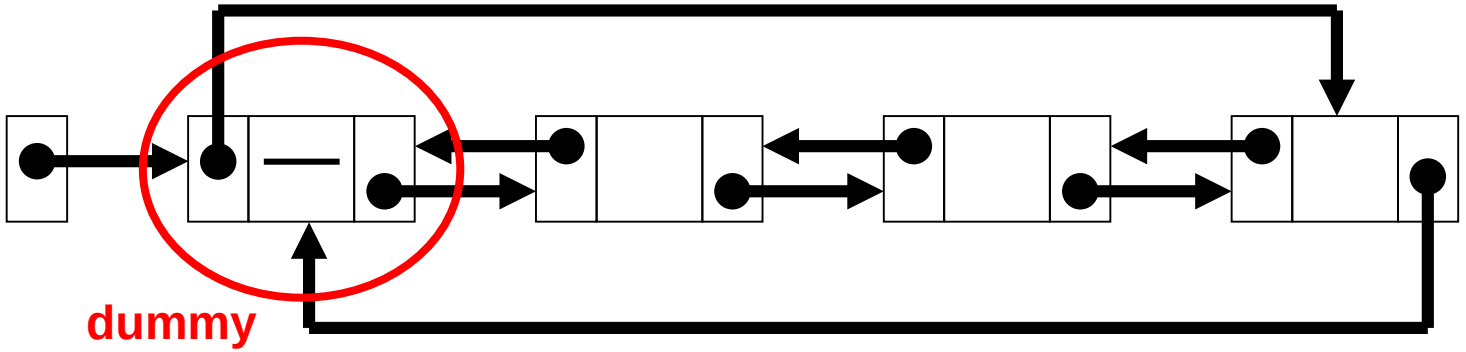


- 重連結リスト（doubly linked list）

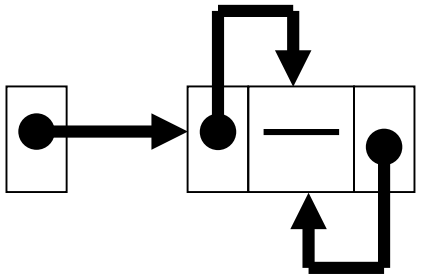


循環リストと重連結リスト

- 循環・重連結リスト

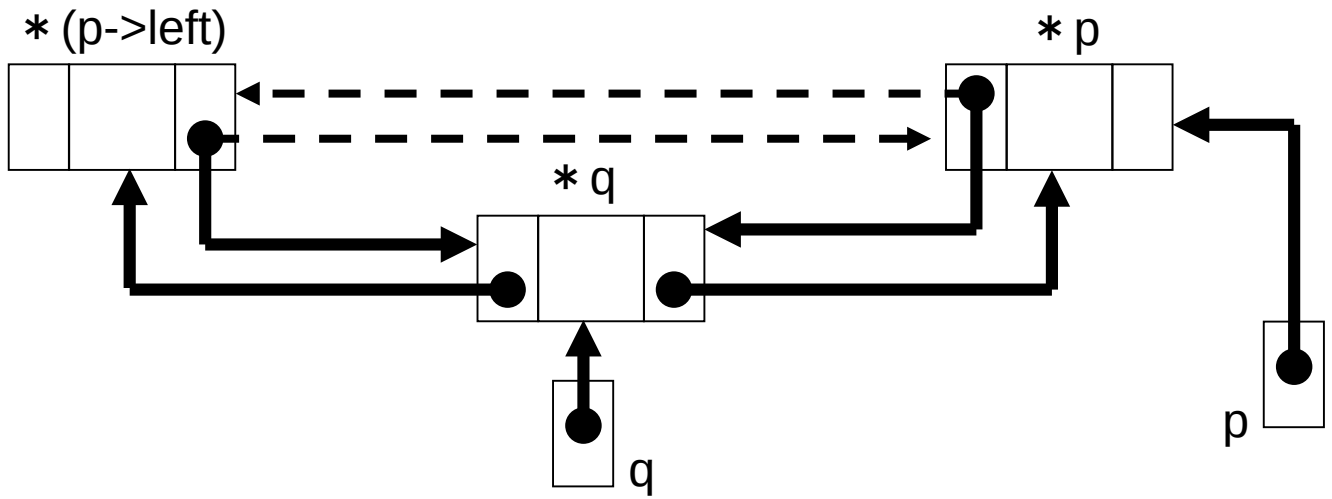


- 空の循環・重連結リスト

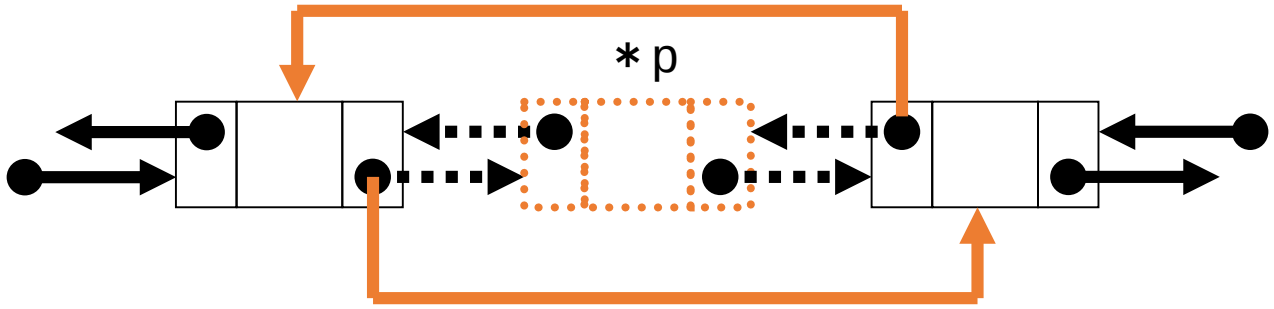


循環リストと重連結リスト

- ノードの挿入



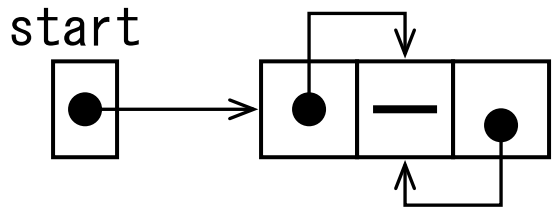
- ノードの削除



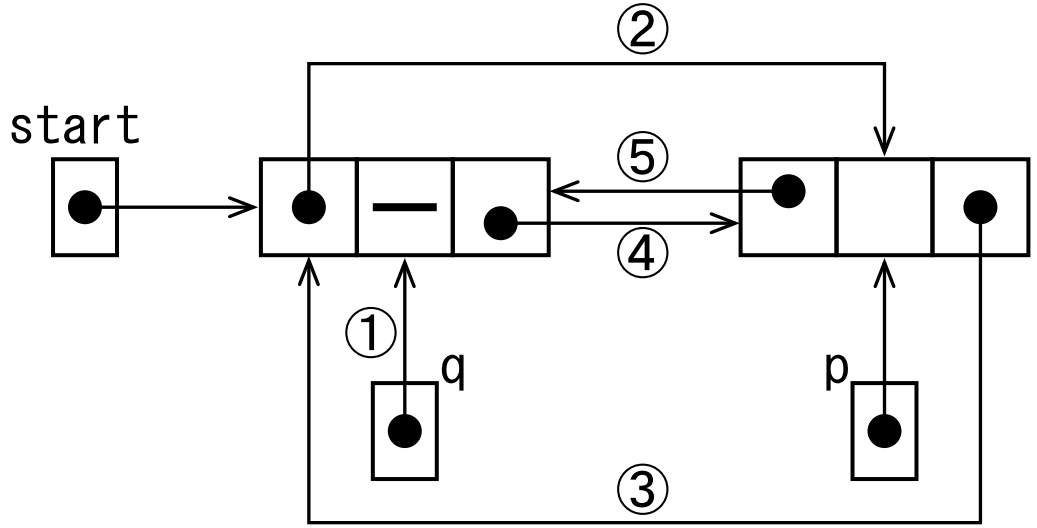
循環・重連結リスト

```
typedef struct Node {
    int num;
    struct Node *left, *right;
} node;

int insert_cdl_list(int x)
{
    node *q, *p = (node*)malloc(sizeof(node));
    if (p == NULL)
        return 0;
    p->num = x;
 ① q = start->left;
 ② start->left = p;
 ③ p->right = start;
 ④ q->right = p;
 ⑤ p->left = q;
    return 1;
}

int insert_left(node *p, int x)
{
    node *q = (node*)malloc(sizeof(node));
    if (q == NULL)
        return 0;
 ⑥ q->left = p->left;
    q->num = x;
 ⑦ q->right = p;
 ⑧ p->left->right = q;
 ⑨ p->left = q;
    return 1;
}
```

・初期状態



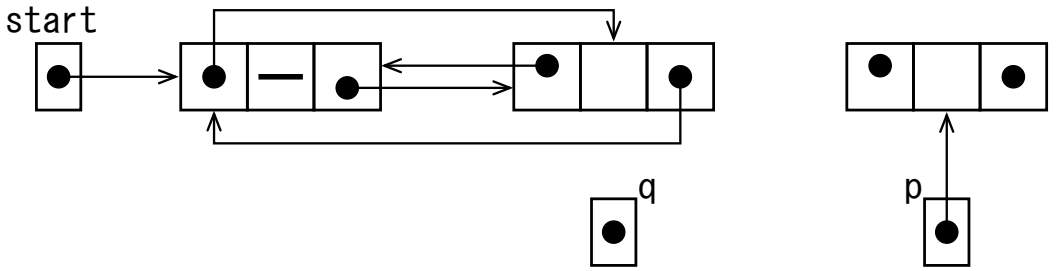
・関数 insert_cdl_list を実行



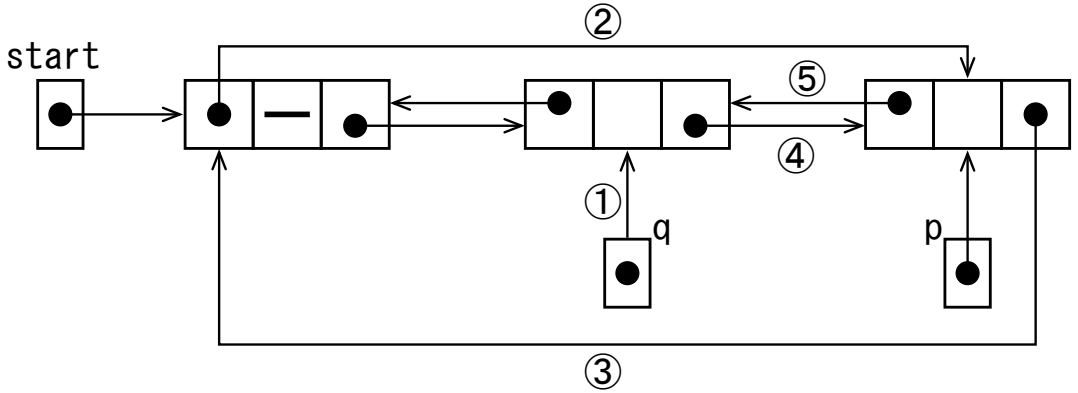
問題1

```
typedef struct Node {
    int num;
    struct Node *left, *right;
} node;

int insert_cdl_list(int x)
{
    node *q, *p = (node*)malloc(sizeof(node));
    if (p == NULL)
        return 0;
    p->num = x;
 ① q = start->left;
 ② start->left = p;
 ③ p->right = start;
 ④ q->right = p;
 ⑤ p->left = q;
    return 1;
}

int insert_left(node *p, int x)
{
    node *q = (node*)malloc(sizeof(node));
    if (q == NULL)
        return 0;
 ⑥ q->left = p->left;
    q->num = x;
 ⑦ q->right = p;
 ⑧ p->left->right = q;
 ⑨ p->left = q;
    return 1;
}
```

- 再度，関数 insert_cdl_list を実行した結果を答えなさい.



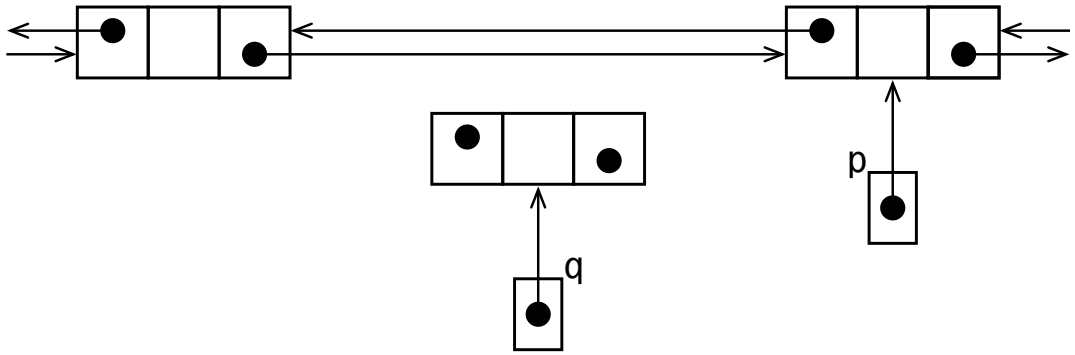
- 解答



問題2

```
typedef struct Node {
    int num;
    struct Node *left, *right;
} node;

int insert_cdl_list(int x)
{
    node *q, *p = (node*)malloc(sizeof(node));
    if (p == NULL)
        return 0;
    p->num = x;
 ① q = start->left;
 ② start->left = p;
 ③ p->right = start;
 ④ q->right = p;
 ⑤ p->left = q;
    return 1;
}

int insert_left(node *p, int x)
{
    node *q = (node*)malloc(sizeof(node));
    if (q == NULL)
        return 0;
 ⑥ q->left = p->left;
    q->num = x;
 ⑦ q->right = p;
 ⑧ p->left->right = q;
 ⑨ p->left = q;
    return 1;
}
```

- 関数 insert_left を実行した結果を答えなさい.



- 解答