アルゴリズムとデータ構造

第5回

ポインタによるリスト、

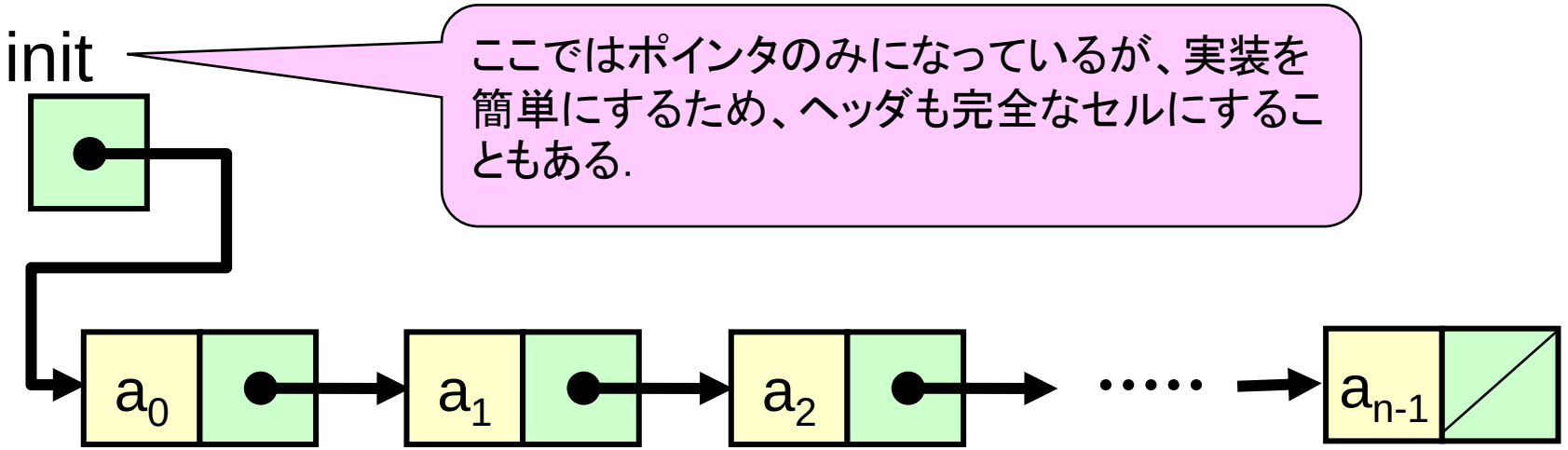
循環・重連結リスト

前回の復習

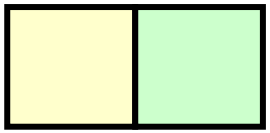
- 多次元配列
  - 2次元配列
  - 3次元配列
  - 応用例：年内の経過日数
- 構造体
  - typedef宣言
  - 構造体のメンバーの参照
  - 構造体の配列
- 配列によるリスト
  - スタックとキュー
  - リストの実現に使用できるデータ構造
  - リストを操作する代表的な関数8つ
  - 配列による線形リストの実現

ポインタによる線形リスト

- ポインタによってリストのデータ構造を表現したものを連結リスト（リンクトリスト，linked list）と呼ぶ
- 「要素」と「次のセルを指すポインタ」で構成される連結リストは，特に，**単方向リスト**，**一方向リスト**などと呼ばれる
- 実現方法
  - 本来のデータと、次のノードを示すポインタを用意
    - 自分自身と同じ構造体型を指すポインタを含む構造体：自己参照構造体
  - データが追加される時点で動的にデータ格納用構造体を確保
    - 確保した構造体を、次のノードを示すポインタで指す

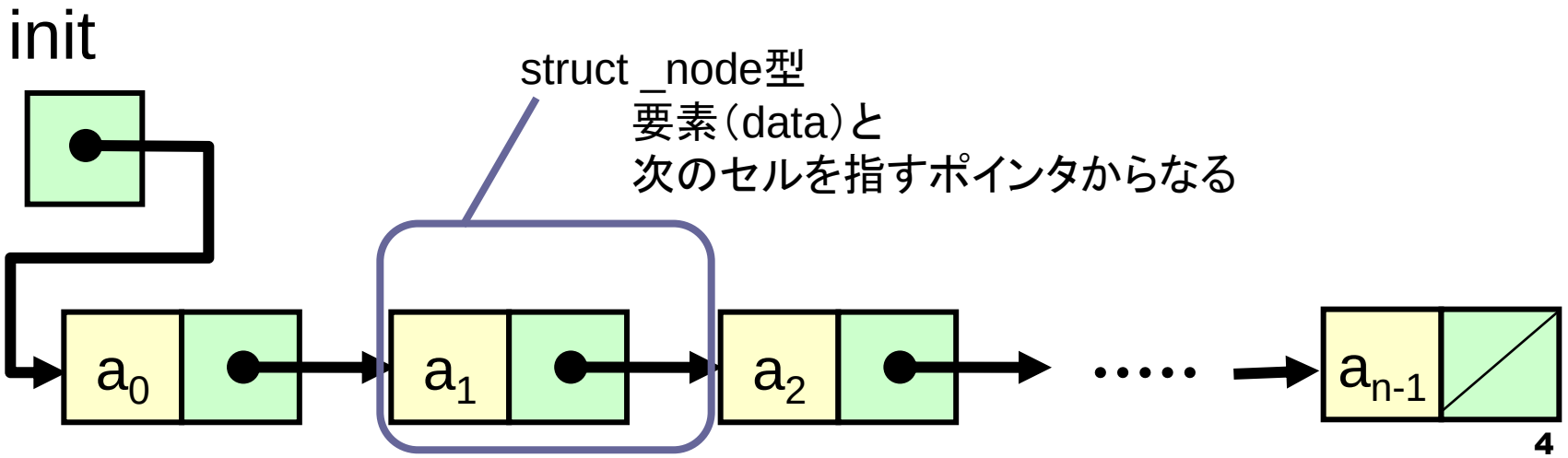


ポインタによる線形リスト

- 自己参照構造体によるノード 

```
typedef struct _node {
    Menber data;                /* データを格納する構造体 */
    struct _node *next;         /* 後続ノードへのポインタ */
} Node;
```

・後続ノードがない場合、nextはNULL



ポインタによる線形リスト

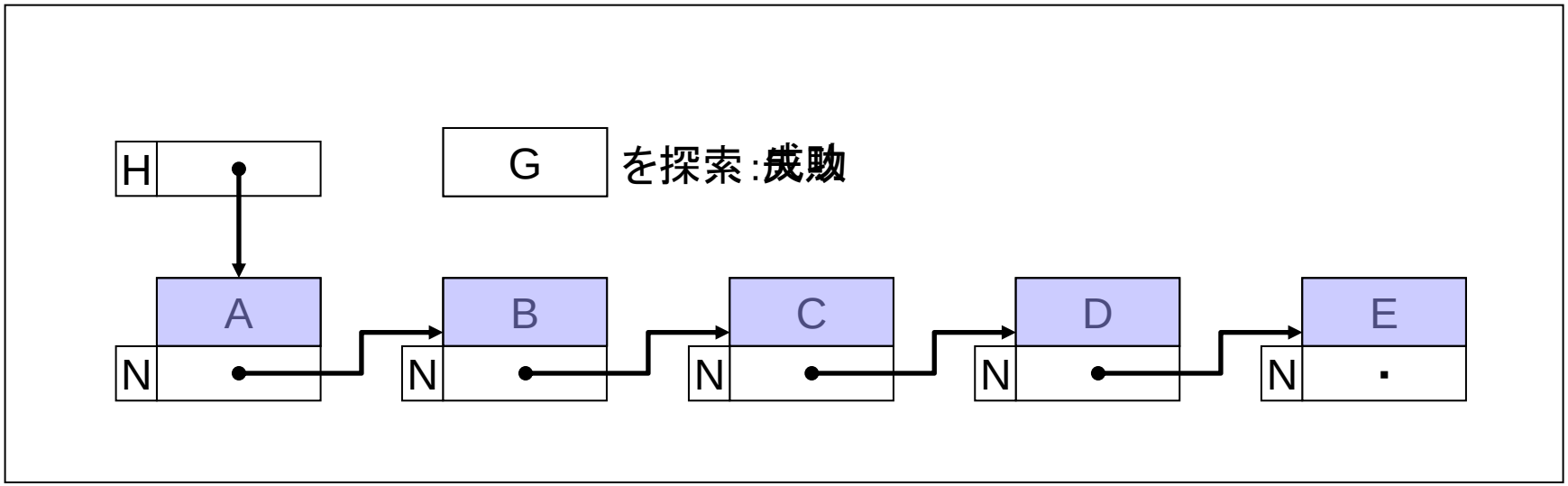
- List型構造体

```
typedef struct {
    Node *head;         /*先頭ノードへのポインタ */
    Node *crnt;         /* 現在着目中のノードへのポインタ */
} List;
```

・headは必須、データがない場合NULL
・crntは便宜上用意、なくてもよい

ポインタによる線形リスト

- ノードの探索
  - 線形探索でデータを探索
    - 先頭ノードから目的値を持つノードを探索
    - 探索すべき値と等しい要素を持つノードを見つけたら探索成功
    - 探索すべき値が見つからず末尾までいったら探索失敗

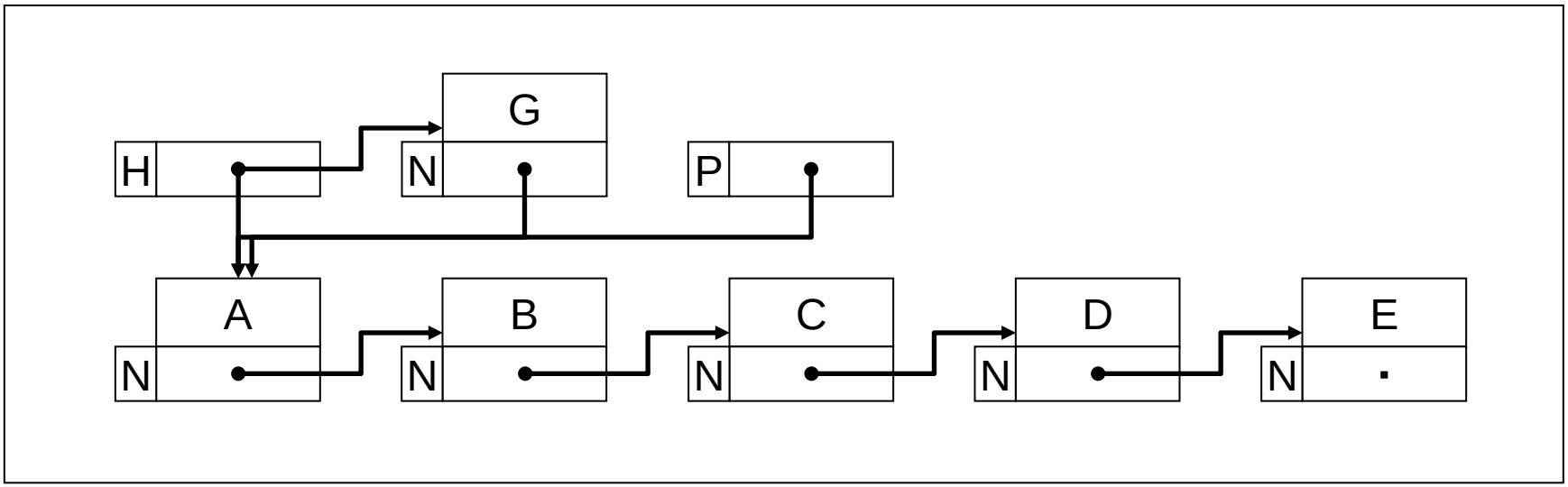


ポインタによる線形リスト:ノードの探索

```
/*--- 関数compareによってxと一致すると判定されるノードを探索 ---*/
Node *Search(List *list, const Member *x,
             int compare(const Member *x, const Member *y))
{
    Node *ptr = list->head;                                    1

    while (ptr != NULL) {
        if (compare(&ptr->data, x) == 0) {  /* キー値が一致 */
            list->crnt = ptr;
3           return ptr;                     /* 探索成功 */     2
        }
4       ptr = ptr->next;                    /* 後続ノードに着目 */
    }
    return NULL;                            /* 探索失敗 */     5
}
```

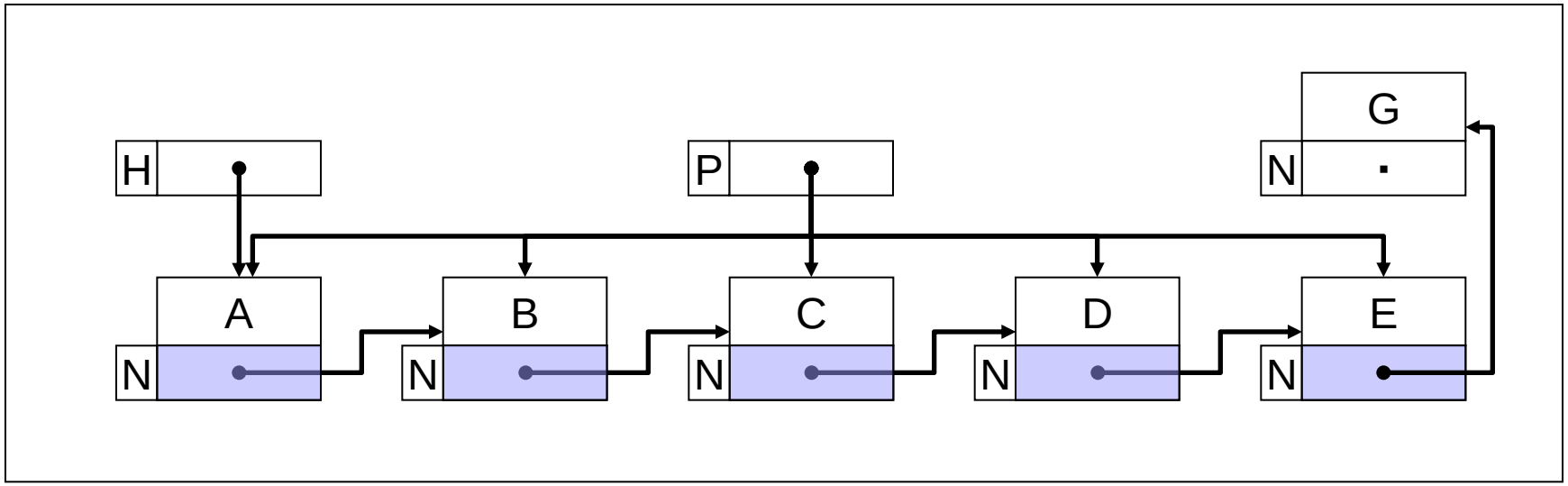
ポインタによる線形リスト

- 先頭へのノードの挿入
  - 新規ノードを生成後、ポインタの付け替え
    1. 現先頭ノードのポインタを保存
    2. 新規ノードを先頭ノードへ
    3. 新規ノードの後続ノードを、保存してあったポインタで置き換え



ポインタによる線形リスト

- 末尾へのノードの挿入
  - 新規ノードを生成後、ポインタの付け替え
    1. headがNULL（データなし）なら先頭にノード挿入
    2. headから、後続ノード（next）がない（NULL）ノードまで探索
    3. 新規ノードを生成し、探索したノードの後続ノードに接続



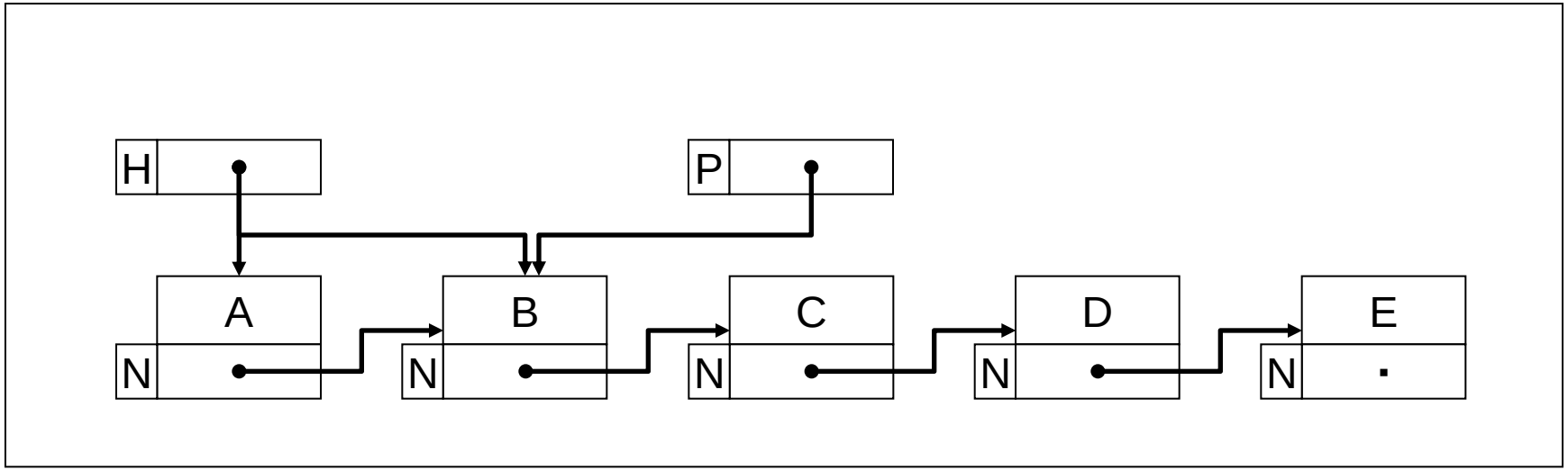
先頭へのノードの挿入
末尾へのノードの挿入

```
/*--- 先頭にノードを挿入 ---*/
void InsertFront(List *list, const Member *x)
{
    Node *ptr = list->head;                          ←1
    list->head = list->crnt = AllocNode();           ←2
    SetNode(list->head, x, ptr);                     ←3
}

/*--- 末尾にノードを挿入 ---*/
void InsertRear(List *list, const Member *x)
{
    if (list->head == NULL)                  /* 空であれば */
        InsertFront(list, x);                /* 先頭に挿入 */
    else {
        Node *ptr = list->head;
    4   while (ptr->next != NULL)
            ptr = ptr->next;
        ptr->next = list->crnt = AllocNode();
    5   SetNode(ptr->next, x, NULL);
    }
}
```

while文終了時、ptrは末尾ノードへのポインタとなる。

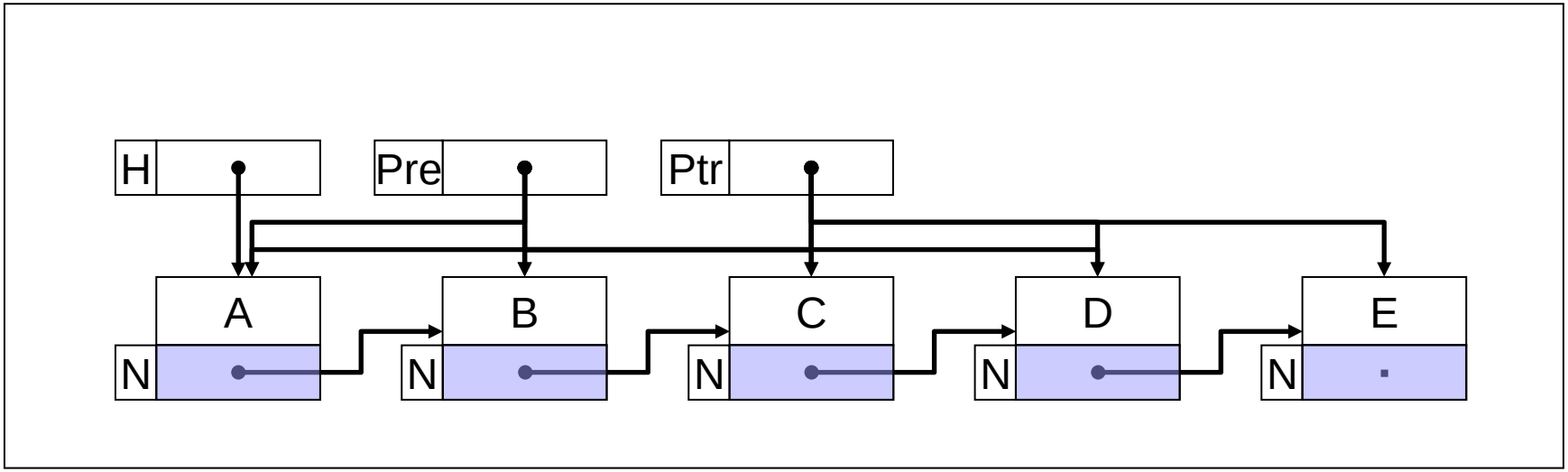
ポインタによる線形リスト

- 先頭ノードの削除
  - 先頭ノードを、先頭の後続ノードへ
    1. 現先頭ノードの後続ノードへのポインタを保存
    2. 先頭ノードを削除
    3. 保存してあったポインタを先頭ノードとして置き換え



ポインタによる線形リスト

- 末尾ノードの削除
  - 末尾ノードの先行ノードに、後続ノードがない状態に
    1. ノードが1つだけなら先頭ノードの削除処理
    2. 末尾から2番目のノードを探索
    3. 末尾ノードを削除
    4. 末尾から2番目の後続ノード（next）をなし（NULL）に更新



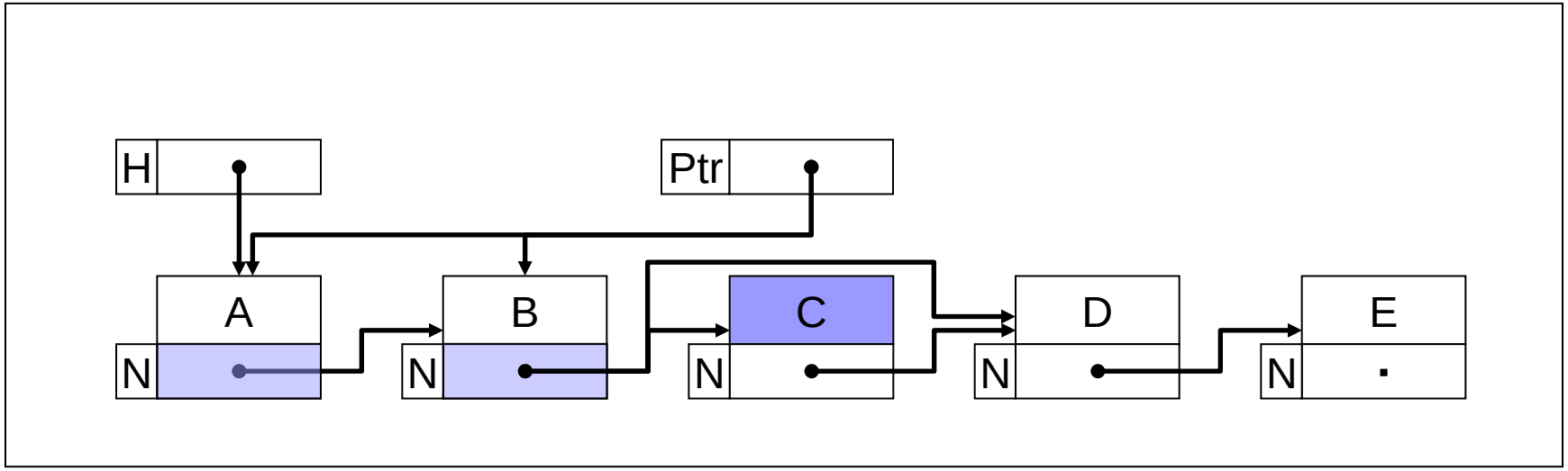
先頭ノードの削除
末尾ノードの削除

```
/*--- 先頭ノードを削除 ---*/
void RemoveFront(List *list)
{
    if (list->head != NULL) {
        Node *ptr = list->head->next;      /* ２番目のノードへのポインタ */
        free(list->head);                  /* 先頭ノードを解放 */
        list->head = list->crnt = ptr;     /* 新しい先頭ノード */
    }
}

/*--- 末尾ノードを削除 ---*/
void RemoveRear(List *list)
{
    if (list->head != NULL) {
        if ((list->head)->next == NULL)    /* ノードが一つだけであれば */
            RemoveFront(list);             /* 先頭ノードを削除 */
        else {
            Node *ptr = list->head;
            Node *pre;

1           while (ptr->next != NULL) {
                pre = ptr;
                ptr = ptr->next;
            }
            pre->next = NULL;              /* preは末尾から2番目 */
2           free(ptr);                     /* ptrは末尾 */
            list->crnt = pre;
        }
    }
}
```

while文終了時、ptrは末尾ノードを指し、preは末尾から２番目のノードを指す。

ポインタによる線形リスト

- 着目ノードの削除
  - 着目ノードの先行ノードの後続ノードを、着目ノードの後続ノードに付け替え
    1. ノードが1つだけなら先頭ノードの削除処理
    2. 着目ノードの先行ノードを探索
    3. 探索したノードの後続ノードを着目ノードの後続ノードに更新
    4. 着目ノードを削除



ポインタによる線形リスト

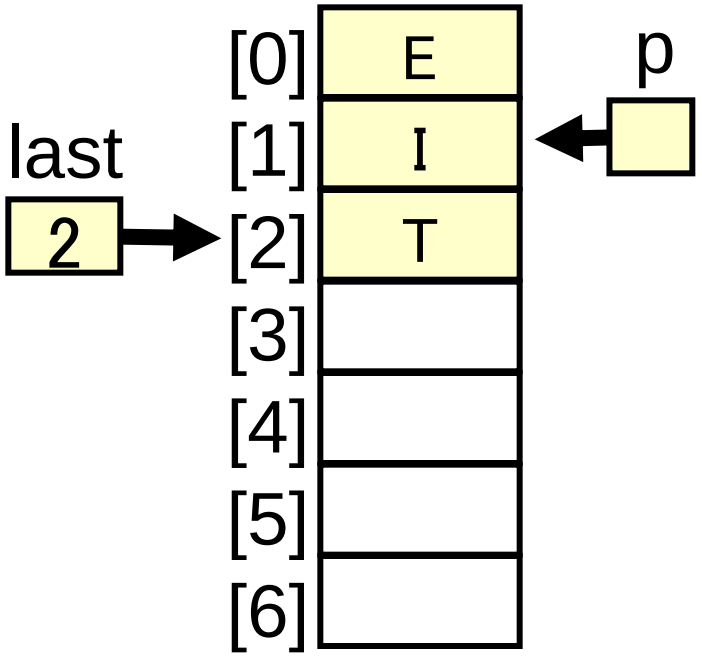
- 全ノードの削除
  - 線形リストが空になるまで先頭要素の削除の繰り返し
- 全ノードの表示
  - 先頭ノードから順に内容表示
  - 後続ノードがなくなったら終了

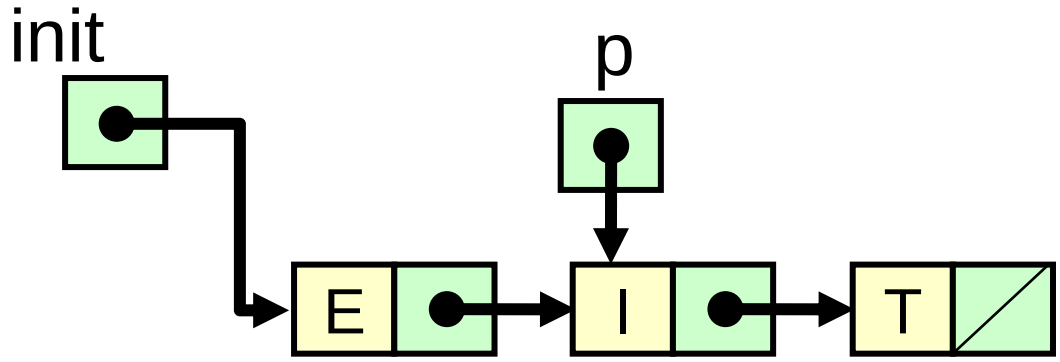
着目ノードの削除
全ノードの削除・表示

```
/*--- 着目ノードを削除 ---*/
void RemoveCurrent(List *list)
{
    if (list->head != NULL) {
        if (list->crnt == list->head)    /* 先頭ノードに着目していれば */
            RemoveFront(list);           /* 先頭ノードを削除 */
        else {
            Node *ptr = list->head;
            while (ptr->next != list->crnt)
                ptr = ptr->next;
            ptr->next = list->crnt->next;
            free(list->crnt);
            list->crnt = ptr;
        }
    }
}

/*--- 全ノードを削除 ---*/
void Clear(List *list)
{
    while (list->head != NULL)           /* 空になるまで */
        RemoveFront(list);               /* 先頭ノードを削除 */
    list->crnt = NULL;
}

/*--- 着目ノードのデータを表示 ---*/
void PrintCurrent(const List *list)
{
    if (list->crnt == NULL)
        printf("着目ノードはありません。");
    else
        PrintMember(&list->crnt->data);
}

/*--- 着目ノードのデータを表示（改行付き）---*/
void PrintLnCurrent(const List *list)
{
    PrintCurrent(list);
    putchar('\n');
}
```

計算量の比較

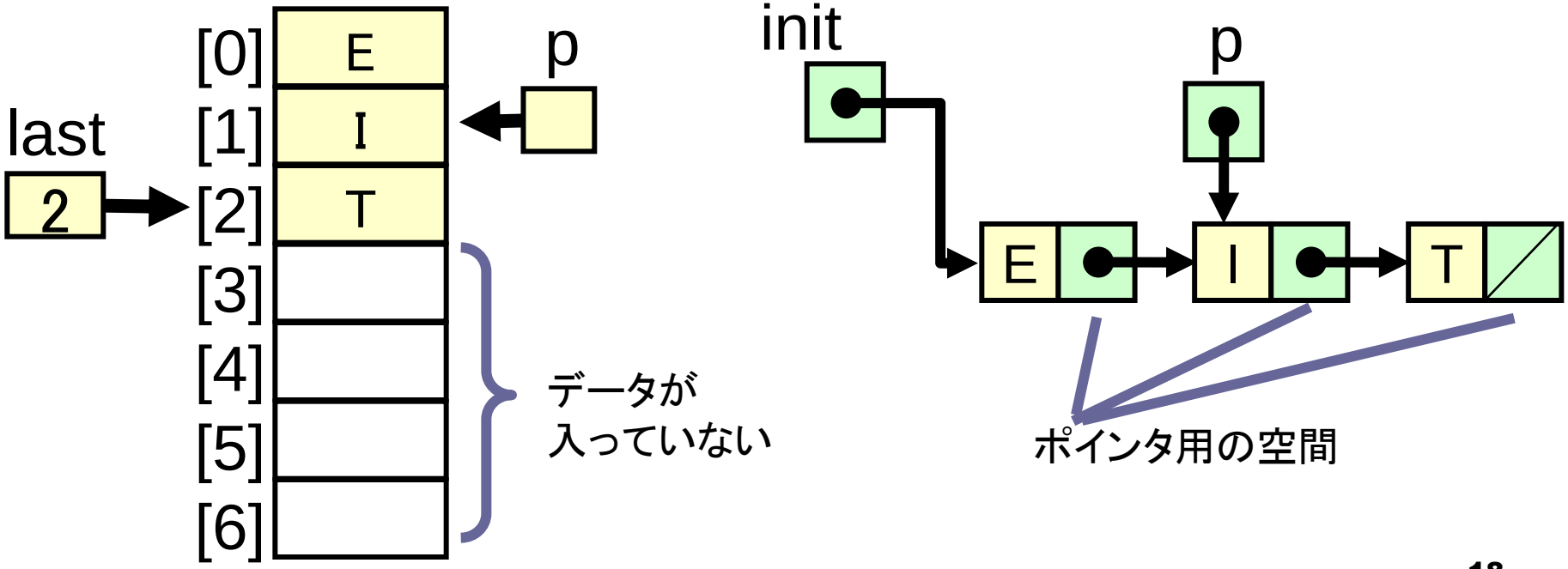
データ構造	INSERT, DELETE	FIND, LAST, PREVIOUS
配列	要素数に比例$O(n)$	一定時間$O(1)$
単方向リスト	一定時間$O(1)$	要素数に比例$O(n)$





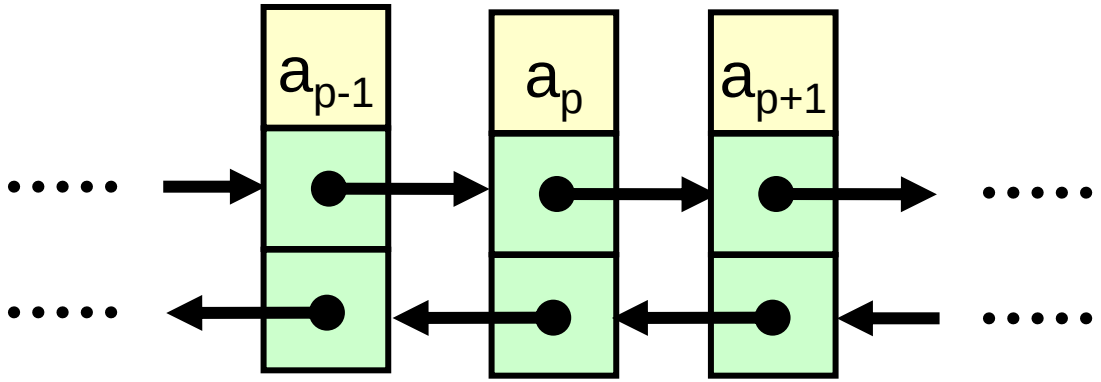
メモリの使用効率に関する比較

データ構造	リストの最大長	余分に必要になるメモリ
配列	固定	MAXLENGTH – 実際の長さ
単方向リスト	可変	ポインタ用の空間



循環・重連結リスト

- 循環リスト
  - 線形リストの末尾ノードが先頭ノードを指すリスト
- 重連結（双方向）リスト
  - 後続ノードへのポインタだけでなく、先行ノードへのポインタも備えたリスト
    - 長所：リストを前方にも後方にもたどれる
    - 短所：前のセルを指すポインタが必要になる
      単方向リストと比べ，操作が複雑になる



- 循環・重連結リスト
  - 循環リストと重連結リストの両方を併せ持つリスト

循環・重連結リスト

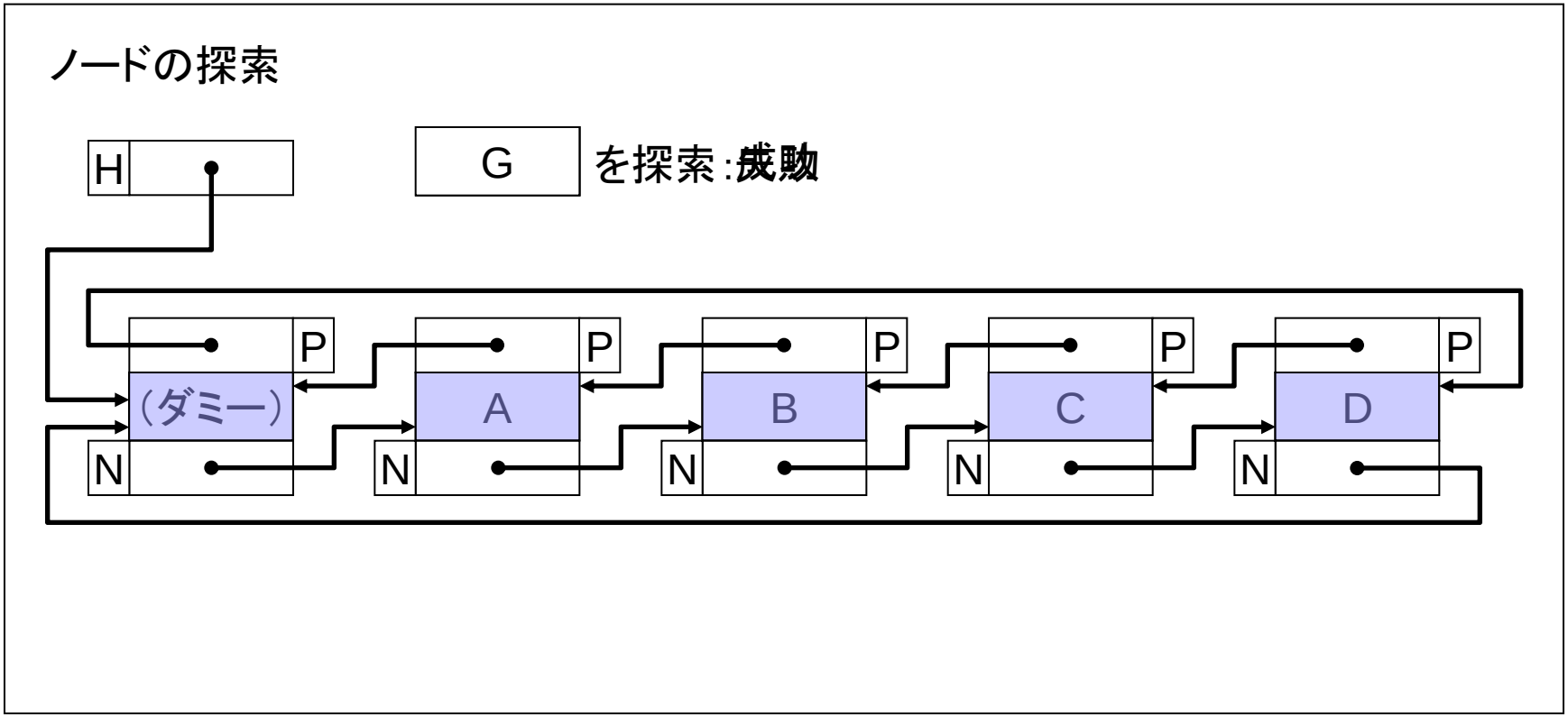
- 循環・重連結リストの実現
  - 実現方法
    - 本来のデータと、先行ノード、後続ノードを示す2つのポインタを備えたノードを用意
  - リストの初期化
    - データがなくてもダミーとして1つノードを作成
      - ノードの追加や削除を円滑に行うため

循環・重連結リスト

- 循環・重連結リストの実現
  - ノードの探索
    - 線形探索でデータを探索
      - ダミーノードの次のノードから目的値を持つノードを探索
      - 探索すべき値と等しい要素を持つノードを見つけたら探索成功
      - 探索すべき値が見つからずダミーノードまで戻ったら探索失敗

循環・重連結リスト

- 循環・重連結リストの実現



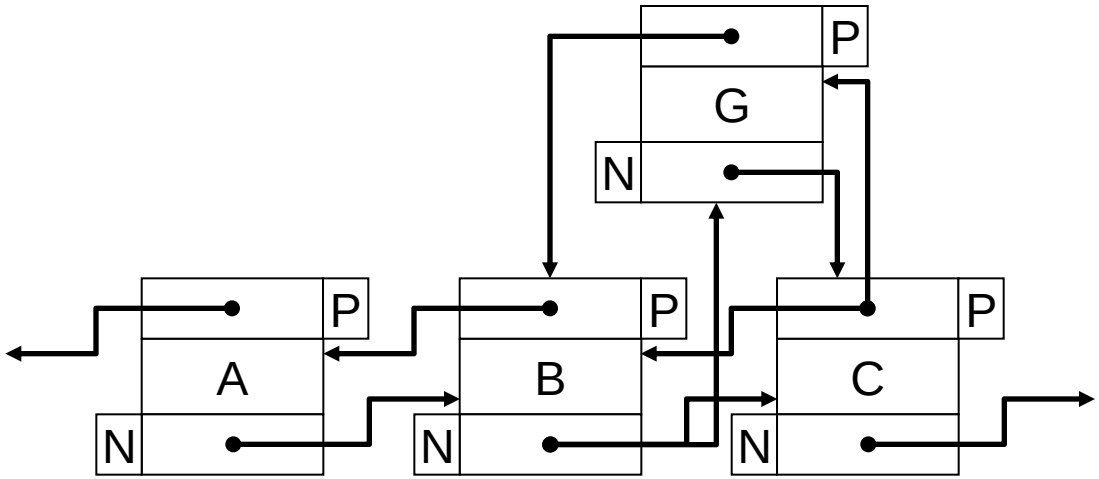
循環・重連結リスト

- 循環・重連結リストの実現
  - ノードの挿入
    - 新規ノードと、挿入すべき前後のノードでポインタの付け替え（4つ）
  - 先頭へのノードの挿入
    - ダミーノードの直後へノードを挿入
  - 末尾へのノードの挿入
    - ダミーノードの直前へノードを挿入

循環・重連結リスト

- 循環・重連結リストの実現

ノードの挿入



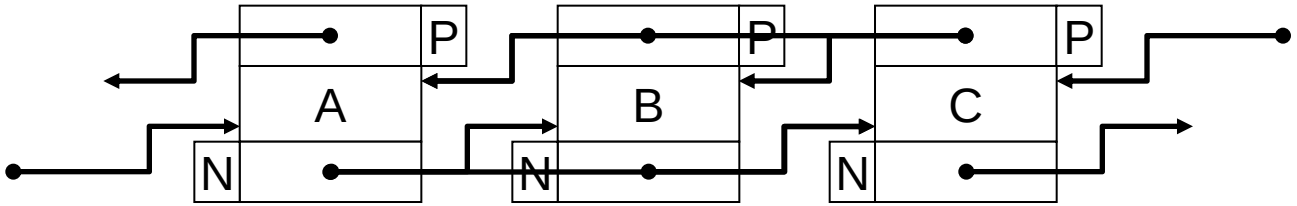
循環・重連結リスト

- 循環・重連結リストの実現
  - ノードの削除
    - 削除するノードの記憶域を開放し、前後のポインタを適宜付け替え
  - 先頭ノードの削除
    - ダミーノードの直後のノードを削除
  - 末尾ノードの削除
    - ダミーノードの直前のノードを削除

循環・重連結リスト

- 循環・重連結リストの実現

ノードの削除

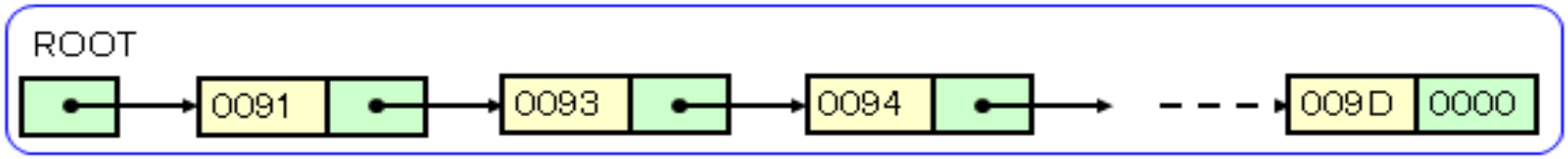


まとめ

- 抽象データ型としての「リスト」
  - 一定の型の要素を0個以上一列に並べたもの
  - リスト中のどの位置でも自由に参照，挿入(Insert)，削除(Delete)の操作を行える
- リストの実現方法
  - 配列を用いる方法：「要素の配列」と「最後の要素の位置を示す変数」で実現
  - ポインタを用いる方法：「要素」と「次のセルを指すポインタ」でセルを構成し，セルを順次つなぐことで，連結リストを作成し，実現

演習問題

のように連結されている時、下の番地表での連結を矢印で示せ。
一つのセルは連続した2つの番地に入っているとする。



ROOT

番地	内容
⋮	⋮
00FF	0100
0100	0091
0101	0106
0102	009C
0103	0104
0104	009D
0105	0000

番地	内容
0106	0093
0107	0108
0108	
0109	010E
010A	009A
010B	010C
010C	009B
010D	0102

番地	内容
010E	0095
010F	010A
0110	0096
0111	0112
0112	0098
0113	0114
0114	0099
0115	0000