情報構造 第七回

リスト

第6回資料の訂正

クイックソート（第１版）C言語

```
void qsort(int L, int R){                    A[L], …A[R]の再帰的分割手続き
    int i, j; item w;
    i=L; j=R;
    x=A[(L+R)/2].key;                        軸のキーを設定
    do{
        while(A[i].key<x) i++;               左から走査
        while(x<A[j].key) j--;               右から走査
        if(i>j  i<=j){                       走査直後の比較
            w=A[i]; A[i]=A[j]; A[j]=w;       交換
            i++; j--; }}                     走査インデックスを進める
    while (i<=j);                            交換直後の比較
    if(L<i) qsort(L, j);                     各部分列を再帰的に再分割
    if(i<R) qsort(i, R); }                   （Lとj, Rとi の比較はここだけ）
void quicksort(){                            メイン
    qsort(0, n-1); }                         配列全体を与える
```

今日の予定

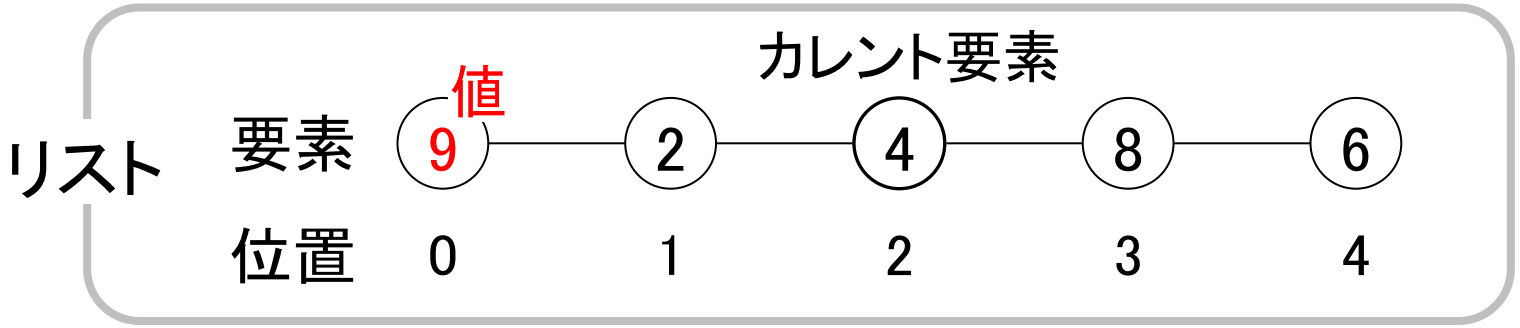
- 基本データ構造：リスト
- リストとは：リストの仕様
- リストのいろいろな実現
    - 表現
    - 実現アルゴリズム
    - 効率：計算量

データ構造：抽象データ型

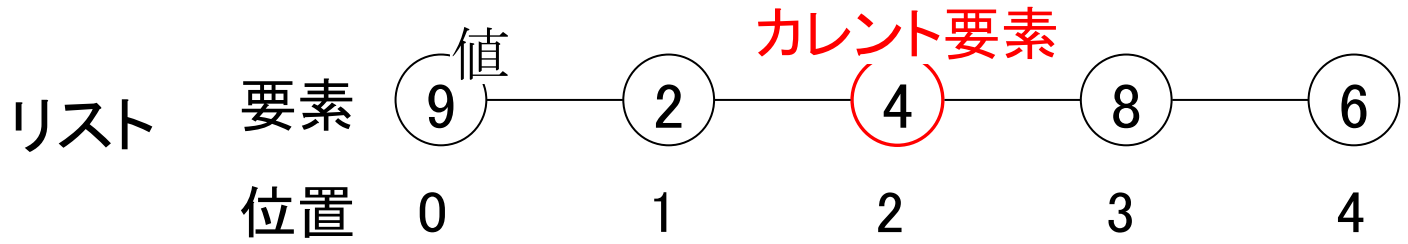
- 抽象データ型
  - 基本データ構造
    - リスト，スタック，待ち行列，順序木，2分木，集合，辞書
  - 高度なデータ構造
    - 2分探索木，AVL木，平衡木
- 仕様
  - 要素や構造を記述
  - 操作は「事前条件ー事後条件」で提示
- 実現
  - 既定義のデータ構造で定義：C言語などで既に定義されているデータ型
    - 配列，構造体，レコード，ポインタなど

リスト（list）

- 同じ型の要素／有限の個数／一列の並び
- 要素の挿入，削除，更新，参照などが行える
- リストの長さ（個数）は実行時に変化する
- 要素の位置：先頭要素から順に，整数値0, 1,…を割り当てる



リスト（list）



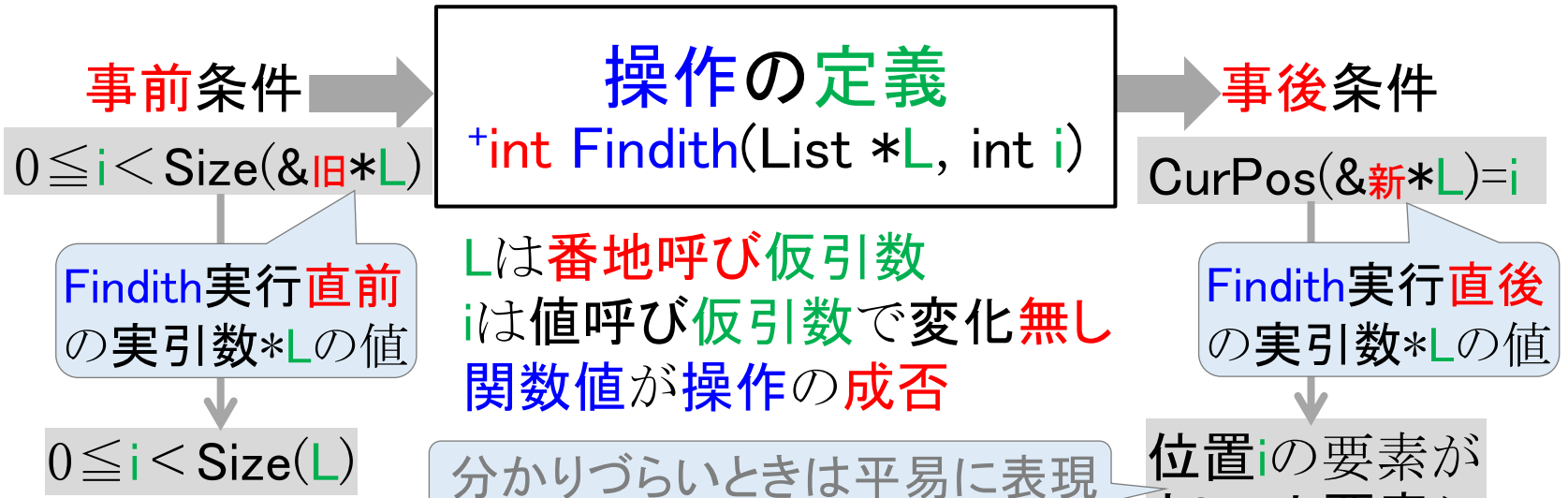
- リストの長さ：要素の個数（0以上の整数）
  - 長さ0のリスト => 空リスト
- カレント要素：リスト中の着目要素
- カレント要素に対する操作：
  - カレント要素の隣に新しい要素を挿入
  - カレント要素の値を削除
  - カレント要素の値を参照・更新
  - カレント要素の隣の要素を新たにカレント要素にする

リストの仕様

- 要素：リストの要素は「同じ型」
  - 値の等価判定とコピーの操作ができる型
- 構造：要素間の関係は線形
  - 先頭要素は，後続要素（successor）をひとつだけ持つ
  - 末尾要素は，先行要素（predecessor）をひとつだけ持つ
  - 他の要素は，先行要素と後続要素をひとつずつ持つ
    - 要素は整数値と位置をもつ
    - 先頭要素の位置は0
    - 位置kの要素の後続要素は位置k+1
- 操作：カレント要素に対して次の基本操作
  - 隣に新しい要素の挿入
  - 削除
  - 値の参照・更新
  - 隣の要素をカレント要素にする
- 要素数0のときは空リストとよび，カレント要素はない

操作の定義と意味

- 「操作の意味」を「定義」「事前条件」「事後条件」を併記して表す
  - 条件は，定義側環境（操作の仮引数）を用いて記述する
- 操作は関数で表す
  - ⁺がついた関数は次の値を返す
    - 操作が行われ事後条件が満たされたとき：真（1）
    - 満たされないとき，または実現の制約で操作が行われなかったとき：偽（0）



操作の呼び出し時の意味

- 操作の「呼び出し時」の意味は，前記「操作の定義の意味」を用いて，操作の呼び出し，事前条件（Pre-condition）と事後条件（Post-Condition）で記述できる．このとき定義側環境は呼出側環境に置き換える

リスト操作の定義

- 操作はC言語の関数の形式で表す
- 用いる抽象データ型リストの型や変数
  - リスト型（List），要素型（Element），位置の型（int）
  - リスト型変数（L），要素型データ（e），要素型変数（v），位置型データ（i）

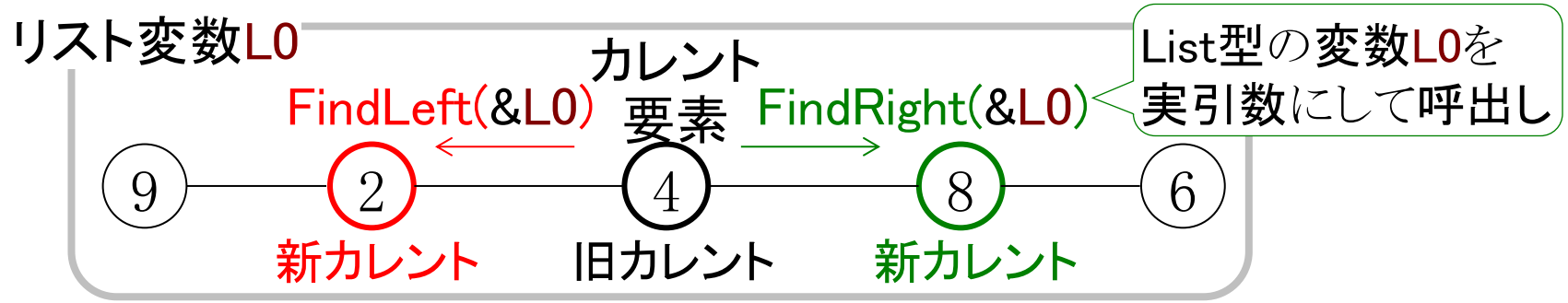
- リストの状態を確かめる操作：Size / CurPos
  - int Size (List *L)　　Lは番地呼びの仮引数
    - Post: 関数Sizeの値は，リスト*Lの要素数
  - int CurPos (List *L)
    - Post: 関数CurPosの値は，
      - カレント要素が設定されているときは，カレント要素の値
      - そうでなければ（空リストまたはカレントの末尾要素削除直後のとき）：-1
- リストを生成する操作：Create
  - void Create (List *L)
    - Post: リスト*L は空リストに設定

リスト操作の定義

(+: 関数値が操作の成否を表す)

- カレント要素を設定する操作：Findith/FindRight/FindLeft
  - +int Findith (List *L, int i)
    - Pre: 0≦i≦Size(L)-1
    - Post 位置iの要素が新カレント要素，関数値は真（1）
  - +int FindRight (List *L)
    - Pre: 0≦CurPos(L)<Size(L)-1
    - Post: 旧カレント要素の右隣が新カレント要素，関数値は真（1）
  - +int FindLeft (List *L)
    - Pre: 0<CurPos(L)≦Size(L)-1
    - Post: 旧カレント要素の左隣が新カレント要素，関数値は真（1）

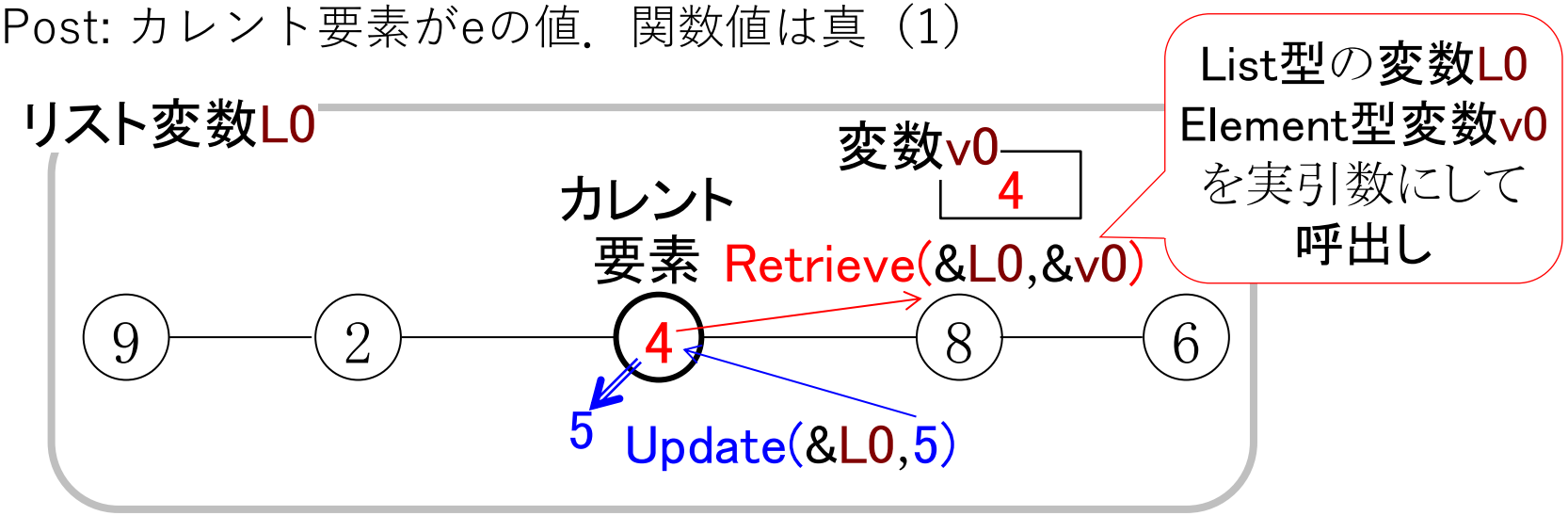
事前条件不成立のとき
関数値は偽(0)となり
実行前と同じ状態



リスト操作の定義

(+: 関数値が操作の成否を表す)

- カレント要素に処理を施す操作：
  Retrieve / Update / InsertLeft / InsertRight / Delete
  - +int Retrieve (List *L, Element *v)
    - Pre: CurPos (L) ≠ -1
    - Post: 変数 *v にカレント要素を設定．関数値は真（１）
  - +int Update (List *L, Element e)
    - Pre: CurPos (L) ≠ -1
    - Post: カレント要素がeの値．関数値は真（1）

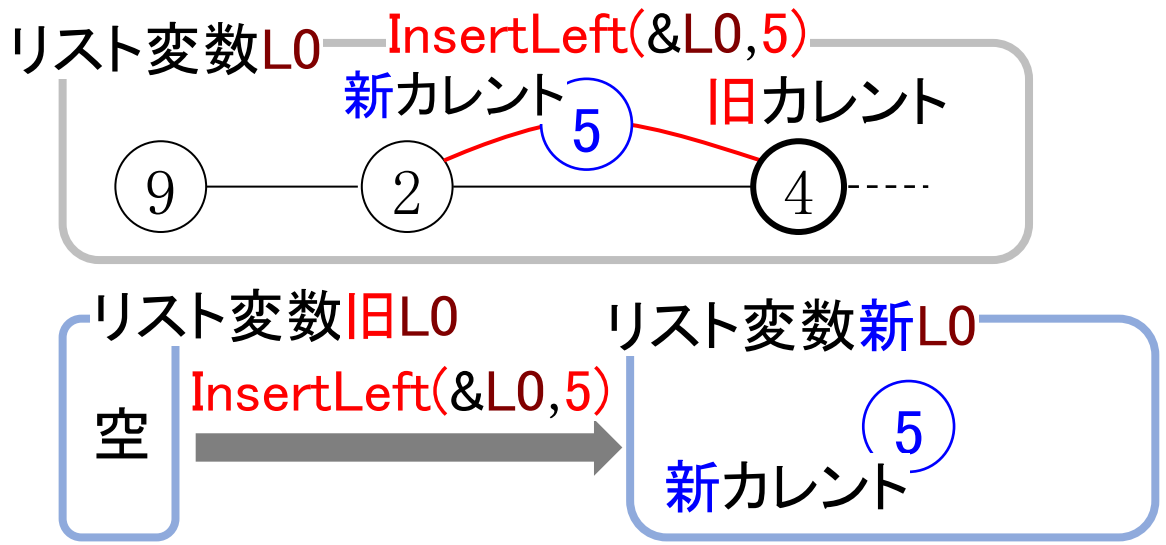


リスト操作の定義

- カレント要素に処理を施す操作：

  Retrieve / Update / InsertLeft / InsertRight / Delete

  - +int InsertLeft (List *L, int i)
    - Pre: CurPos(L) ≠ -1 または Size(L)= 0
    - Post:
      - *Lが空でないとき：eは旧カレント要素の先行要素として挿入され，新カレント要素に
      - *Lが空のとき：唯一の要素として挿入され，新カレント要素に
      - 関数値は真（1）

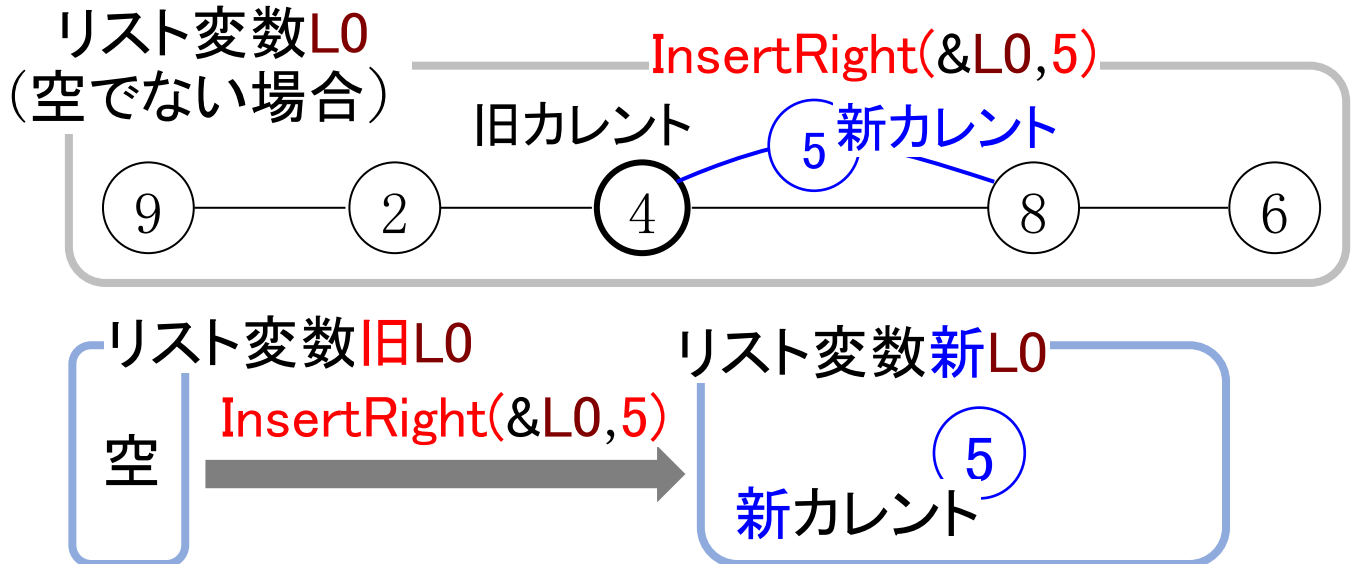


リスト操作の定義

- カレント要素に処理を施す操作：

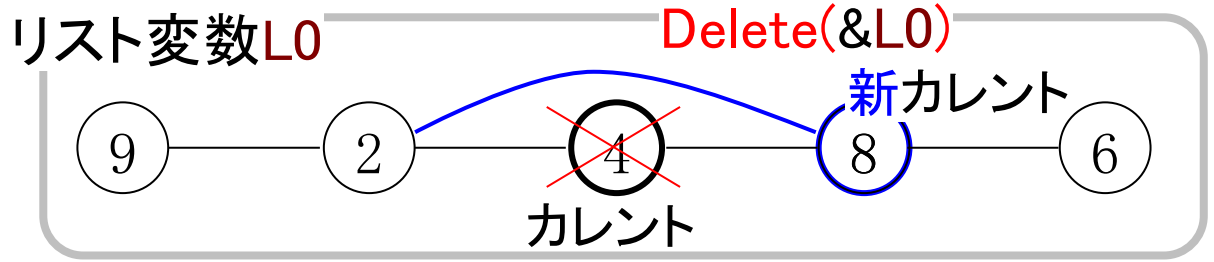
  Retrieve / Update / InsertLeft / InsertRight / Delete

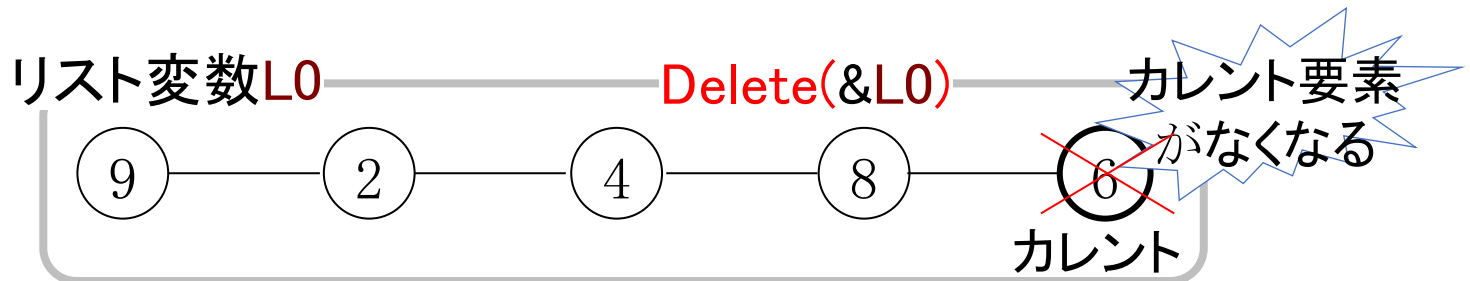
  - +int InsertRight (List *L, int i)
    - Pre: CurPos(L) ≠ -1 または Size(L) = 0
    - Post:
      - *Lが空でないとき：eは旧カレント要素の後続要素として挿入され，新カレント要素に
      - *Lが空のとき：唯一の要素として挿入され，新カレント要素に
      - 関数値は真（1）



リスト操作の定義

- カレント要素に処理を施す操作：
  Retrieve / Update / InsertLeft / InsertRight / Delete
  - +int Delete (List *L)
    - Pre: CurPos(L) ≠ -1
    - Post: 旧カレント要素が削除される
      - 削除される要素の**後続要素**が新カレント要素になる
      - 後続要素がないとき，新カレント要素は設定されない
      - 関数値は真（1）





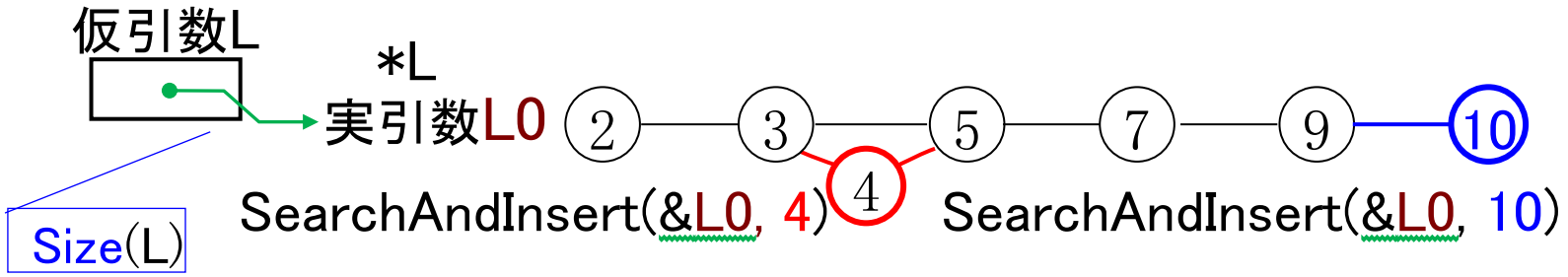
リスト操作の定義（以上で定義した操作の利用）

- リストをコピーする操作
  - void Copy(List *L1, List *L2)
    - Post:
      - リストL1と同じ構造と要素のリストL2を作成する
      - カレント要素の位置もコピーする
- リストを比較する操作
  - int Equal(List *L1, List *L2)
    - Post: リストL1とL2が同じ構造と要素の時，関数値が真（1）
      - L1とL2のカレント要素は異なってても良い

操作の例：整列要素の昇降順リストに要素を挿入

```
int SearchAndInsert (List *L, Element e){
    Element v;
    if( Size(L) == 0){                          /* 空リストのとき */
        InsertRight(L, e); return(1);           /* Lにeを追加 */
    } else {                                    /* 非空リストのとき */
        Findith(L, 0);                          /* 位置0をカレント要素にする */
        while( Retrieve(L, &v), v<e ){          /* vをカレント要素に，カレント数 < 挿入値e？ */
            if( !FindRight(L) )                 /* カレント要素を右に，要素がないとき */
                {InsertRight(L, e); return(1);}}  /* カレント要素の右にeを追加 */
        InsertLeft(L, e):                       /* カレント要素の左にeを追加 */
        return(1);
    }
}
```

仮引数Lは実引数を指すポインタなので、
他関数の呼び出し時は&無しのLを用いる



リストの実現

- 配列へのベタ詰めによる実現

- 構造体とポインタによる実現
  - 一方向連結リスト（第1版）
  - 一方向連結リスト（第2版）
  - 双方向連結リスト

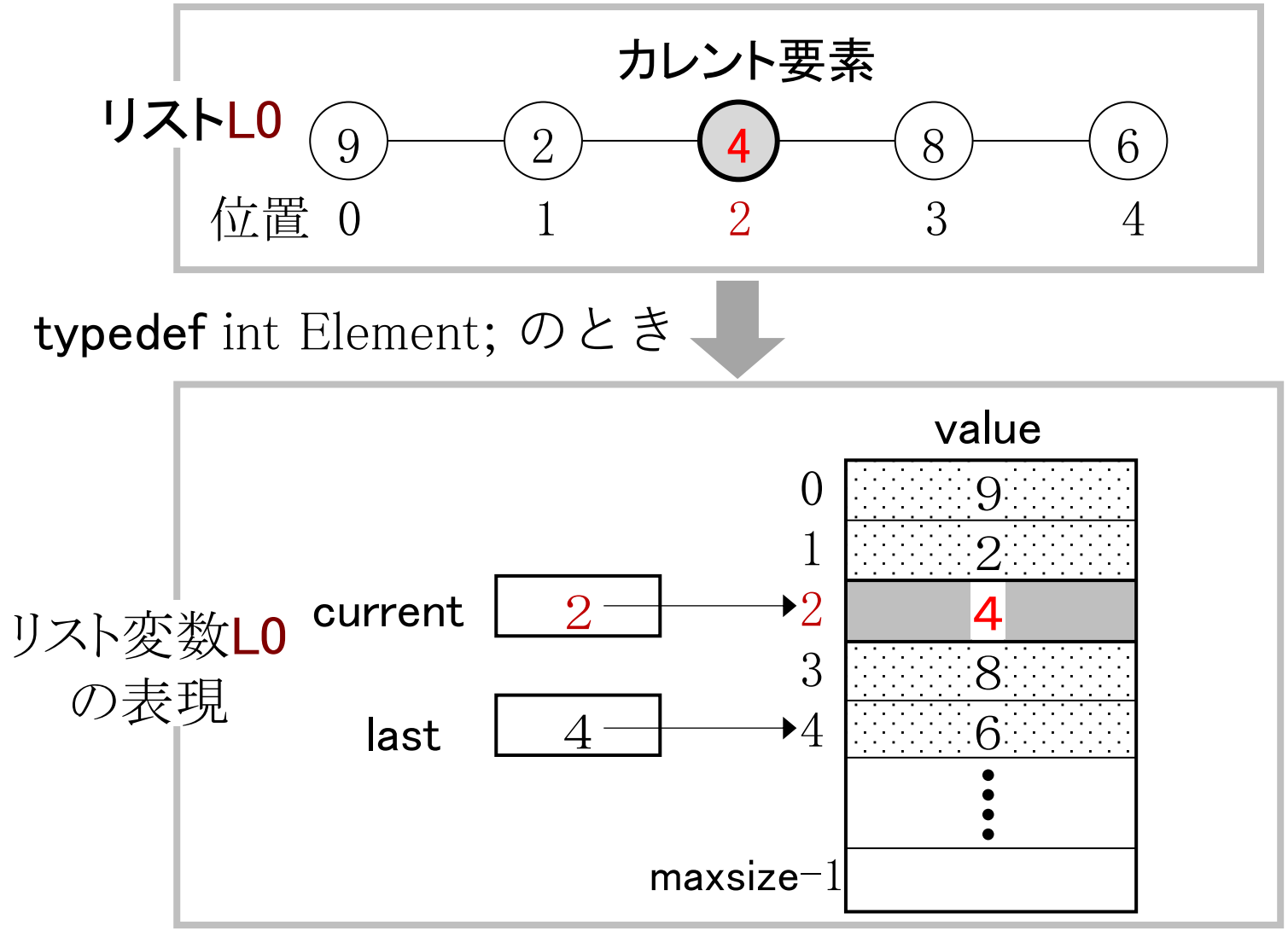
- 連結リストの配列とインデックスによる実現

リスト：配列へのベタ詰め：表現

- 表現：
  - リスト要素を並びの順で，配列の先頭からベタ詰め
  - リスト要素の位置は，配列のインデックスに対応

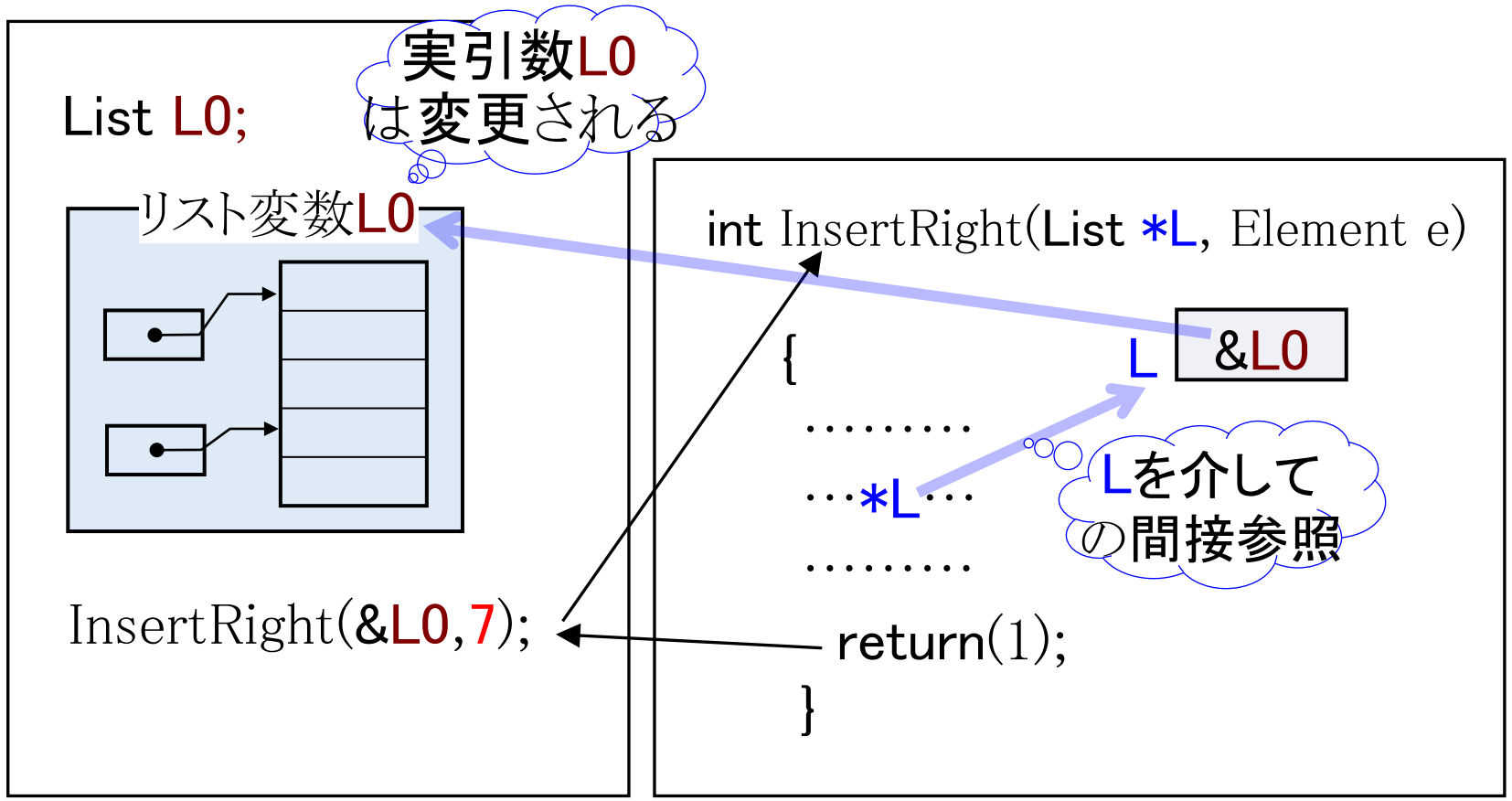
```
#define maxsize 1000                    実現の制約：リストの最大長
typedef struct{
    Element value[maxsize];
    int current;                        カレント要素の位置
    int last;                           末尾要素の位置
} List;
List L0;                                リスト変数 L0
```

リスト：配列へのベタ詰め：表現



リスト：配列へのベタ詰め：実現アルゴリズム

- リストL0 (構造体) は，番地呼びで引き渡す



リスト：配列へのベタ詰め：実現アルゴリズム

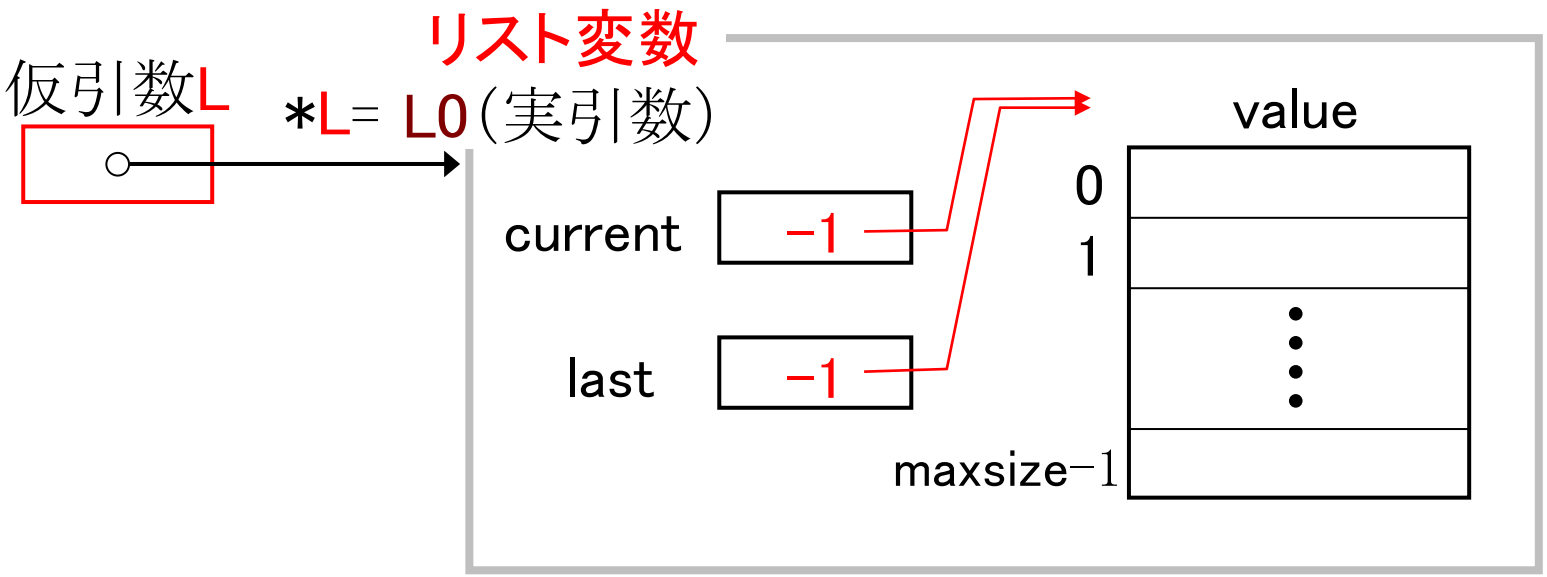
- リスト L0 (実引数) を空リストにする呼び出し：Create

```
void Create( List *L ){
    L->current = -1;
    L->last = -1;
}
```

カレント要素はなし

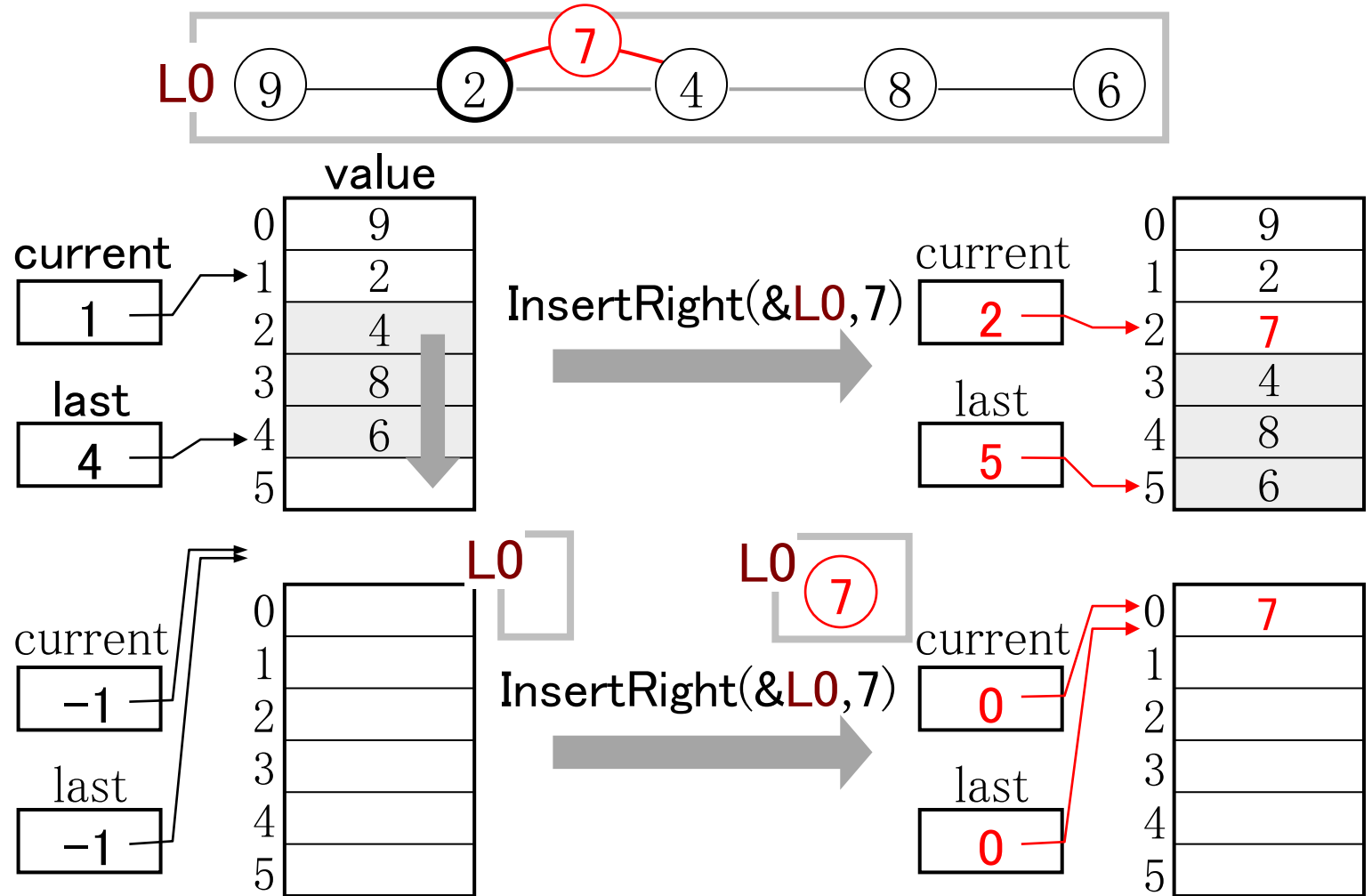
要素数0を表す

(*L).last
≡L->last



リスト：配列へのベタ詰め：実現アルゴリズム

- リスト L0 のカレント要素の右側に 7 を挿入 InsertRight(&L0, 7)



リスト：配列へのベタ詰め：実現アルゴリズム

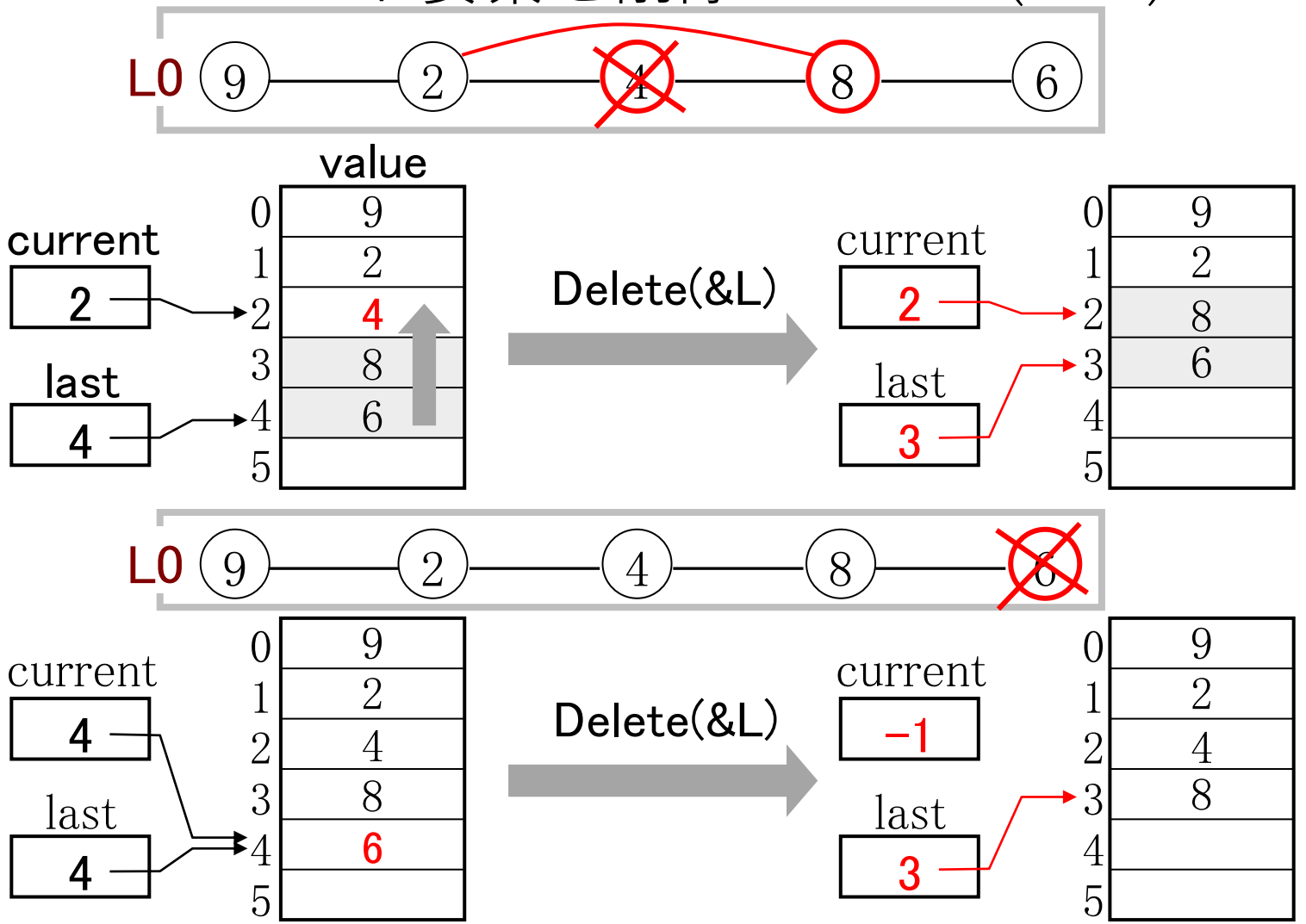
- リスト L0 のカレント要素の右側に e を挿入 InsertRight(&L0, e)

```
int InsertRight(List *L, Element e){
    int i;
    if(L->last >= maxsize-1) ERROR("List is full");
    else if(L->last == -1){                                  /* 空リストのとき */
        L->last = 0; L->current = 0; L->value[0] = e;}
    else if(L->current == -1) return(0);                     /* 非空リストでカレントがないとき */
    else if(L->current == L->last){                          /* 非空リストで末尾がカレント */
        L->current = L->last = L->last+1;                        /* カレント（末尾）を右にずらしてから挿入 */
        L->value[L->current] = e; }
    else{                                                    /* 非空でカレントあり */
        L->last = L->last+1                                    /* カレント以降を右にずらしてから挿入 */
        for(i=L->last; i>=L->current+2; i--)
            L->value[i] = L->value[i-1];
        L->current = L->current  + 1;  L->value[L->current] = e;}
    return(1)
}
```

事前条件を満たさない!

リスト：配列へのベタ詰め：実現アルゴリズム

- リスト L0 のカレント要素を削除：Delete(&L0)



リスト：配列へのベタ詰め：実現アルゴリズム

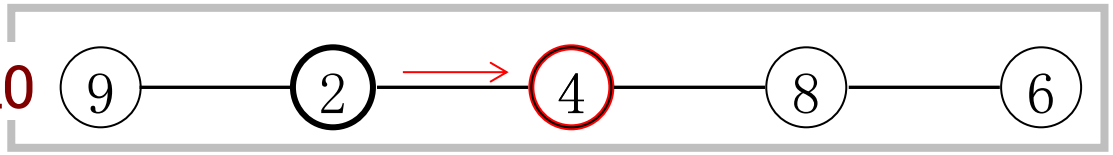
- リスト L0 のカレント要素を削除：Delete(&L0)

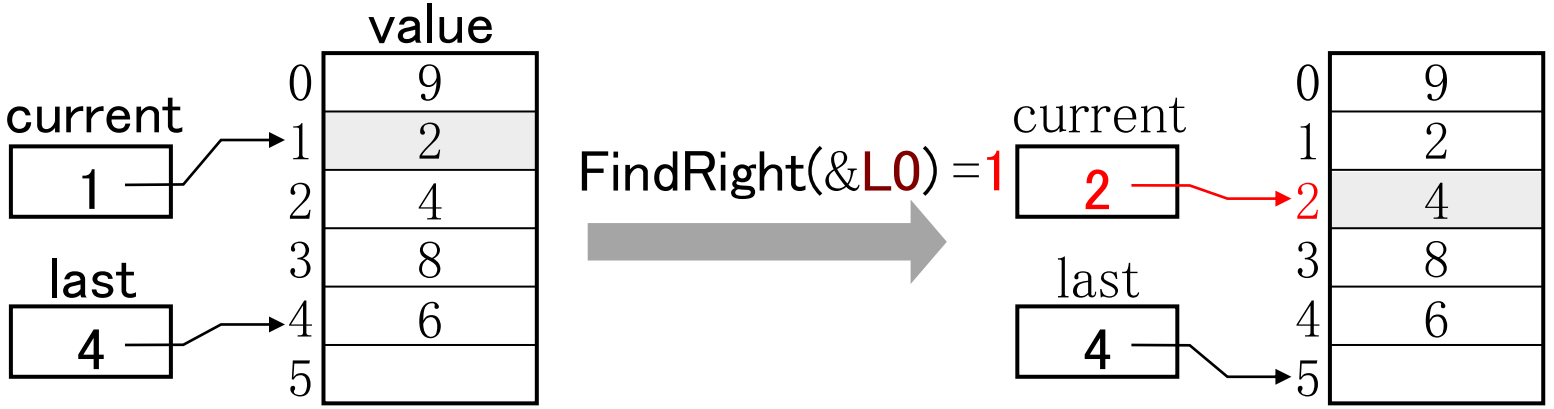
事前条件を満たさない!

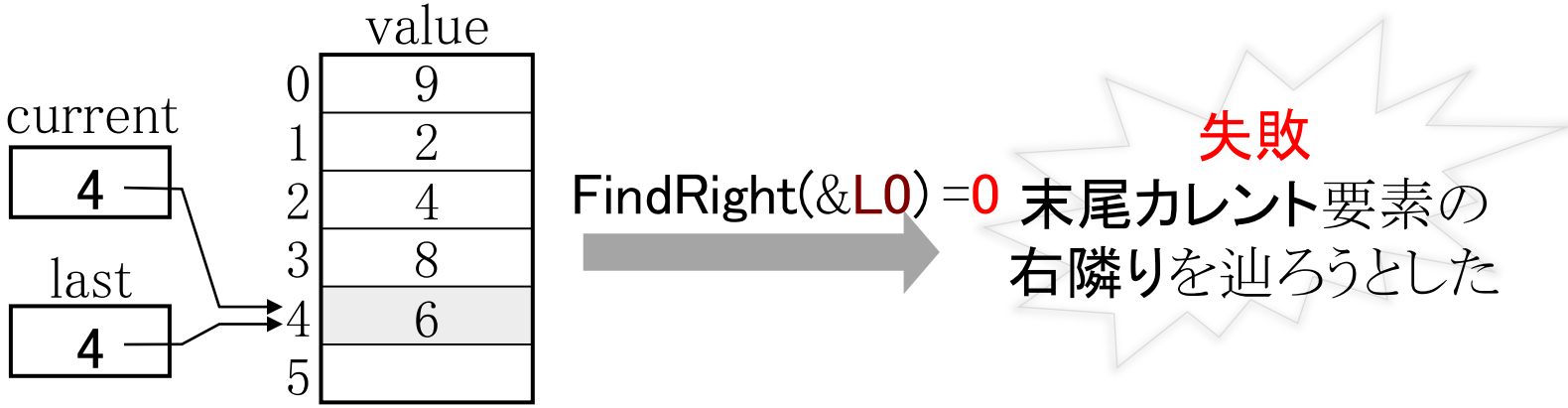
```
int Delete(List *L){
    int i;
    if(L->current == -1) return(0);              /* カレント要素がないとき */
    else if(L->current != L->last){              /* カレント要素が末尾ではないのとき */
        for(i=L->current+1; i<=L->last; i++)     /* カレント要素以降をずらす */
            L->value[i-1] = L->value[i];
        L->last = L->last -1; return(1); }
    else{                                        /* カレント要素が末尾の時*/
        L->last = L->last – 1;                   /* 末尾要素を削除 */
        L->current = -1;                         /* カレント要素を末尾に */
        return(1);}
}
```

リスト：配列へのベタ詰め：実現アルゴリズム

- リスト L0 のカレント要素を右隣に：FindRight(&L0)







リスト：配列へのベタ詰め：実現アルゴリズム

- リスト L0 のカレント要素を右隣に：FindRight(&L0)

```
int FindeRight(List *L){
    int i;
    if(L-> current == -1)                  /* カレント要素がないとき */
        return(0);
    else if(L->current >= L->last)         /* カレント要素が末尾のとき */
        return(0);
    else{                                  /* それ以外 */
        L->current >= L->current+1;            /* カレント要素を右に */
        return(1);
    }
}
```

事前条件を満たさない!

リスト：配列へのベタ詰め
実現アルゴリズムの効率

- 要素の挿入（InsertRight, InsertLeft），要素の削除（Delete）
  - 末尾の要素に対して：一定時間
  - 途中の要素に対して：それ以降の要素をずらすので
    - 要素nに比例する時間

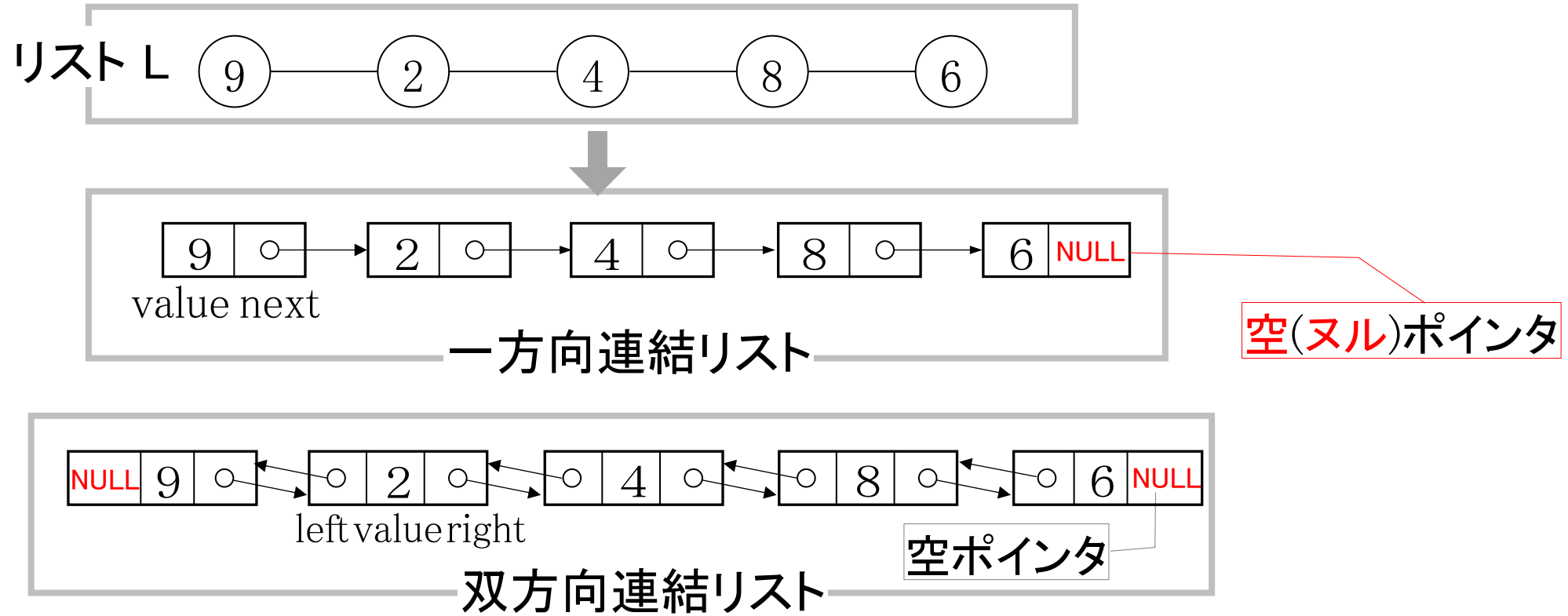
- ほかの操作
  - 一定時間

構造体とポインタによる実現

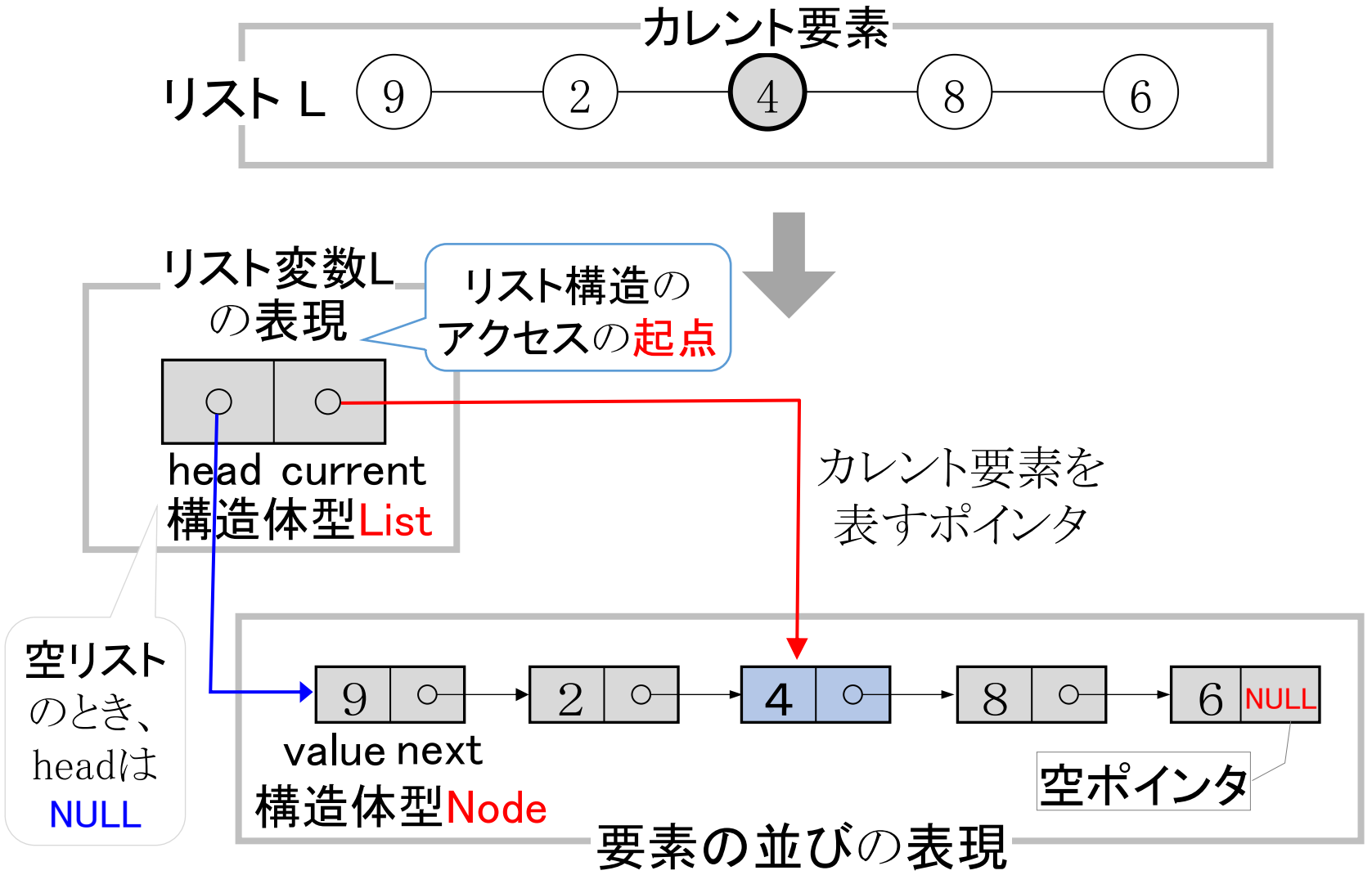
- 一方向連結リスト（第１版）
- 一方向連結リスト（第２版）
- 双向連結リスト

構造体とポインタによる実現

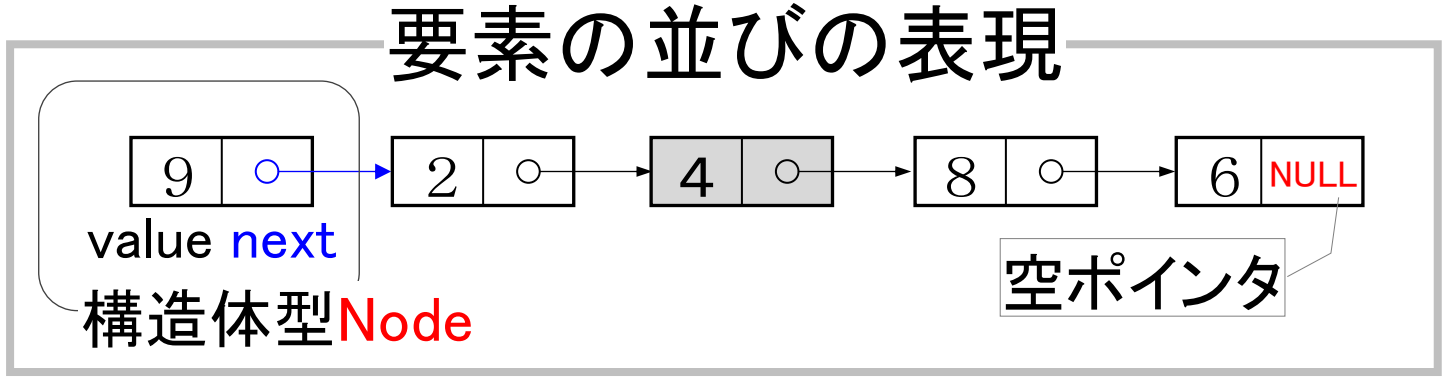
- 連結リスト（linked list）とは
  - リスト要素を構造体で表す
  - リスト要素の並びの順に，要素を表す構造体をポインタでつなげたもの



一方向連結リスト（第１版）：表現



一方向連結リスト（第１版）：表現



- 構造体による表現：不完全型構造体

```
typedef struct node_tag *NodePointer;
typedef struct node_tag{
    Element value;
    NodePointer next;} Node;
```

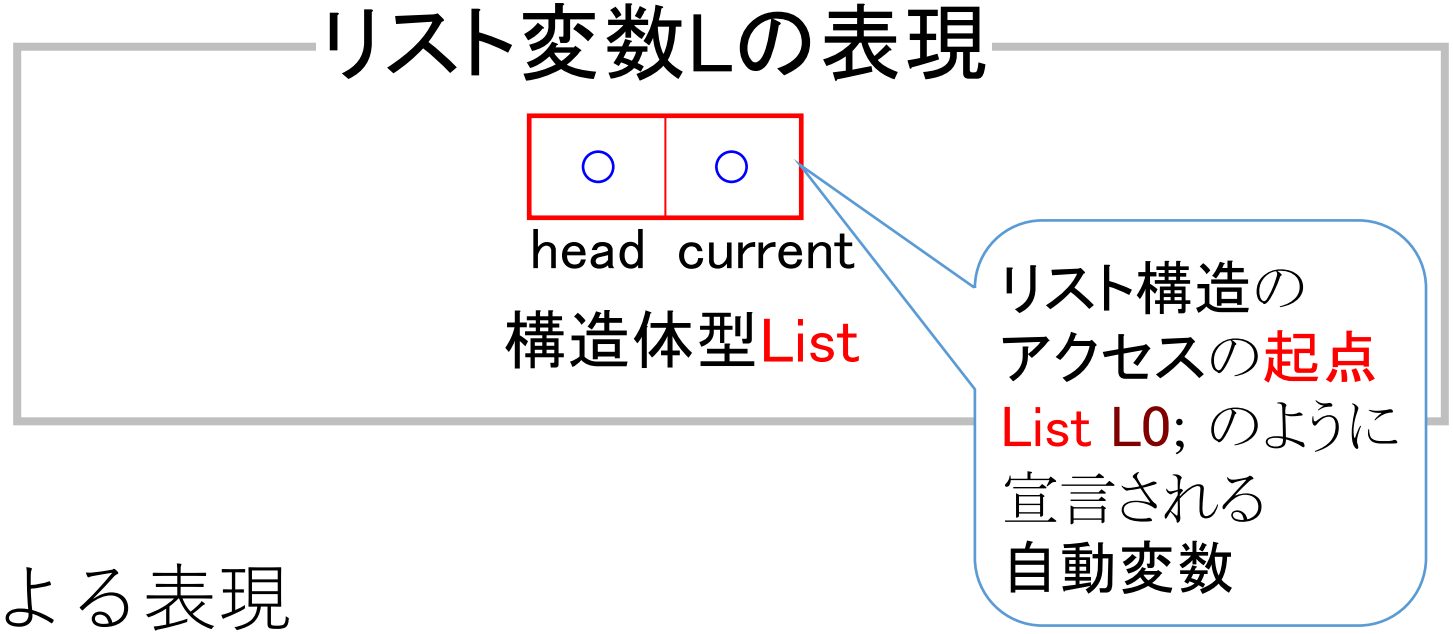
  - NodePointerは，まだ定義されていないnode_tag構造体へのポインタ型なので，不完全型とよぶ

- 構造体による表現：自己参照構造体

```
typedef struct node_tag{
    Element value;
    struct node_tag *next;} Node;
typedef Node *NodePointer;
```

  - 構造体struct node_tagは，nextフィールドで自己参照

一方向連結リスト（第１版）：表現



- 構造体による表現

```
typedef struct{
    NodePointer head;
    NodePointer current;  } List;
```

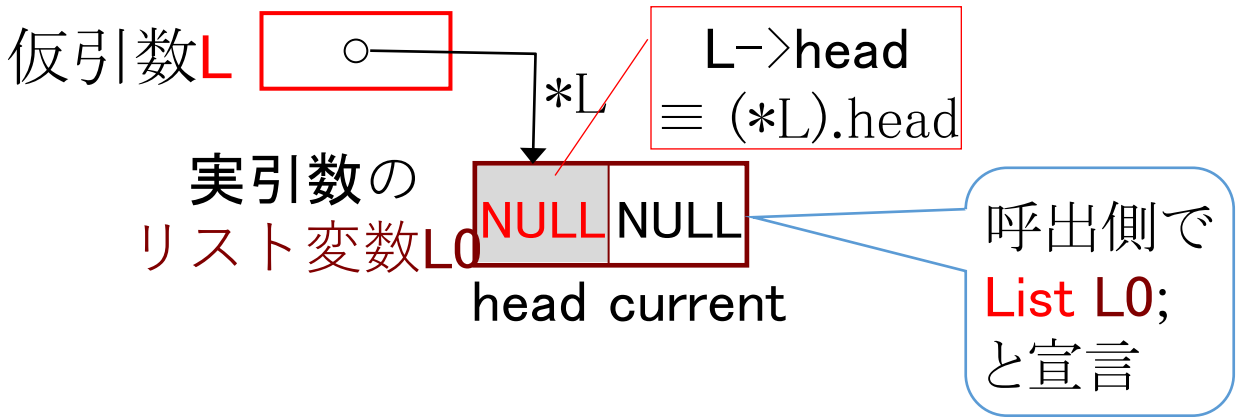
- リスト要素の並びはここからアクセス

一方向連結リスト（第１版）：実現アルゴリズム

- リスト L0 (実引数) を空リストにする呼び出し：Create(&L0)

```
void Create(List *L){          /* 仮引数Lは実引数L0の番地を持つ */
   L->head = NULL;             /* 要素数が0を表す */
   L->current = NULL; }        /* カレント要素はなし */
```

- 注意：リスト変数は番地呼びなので，実引数L0を渡す呼び出しはCreate(&L0)



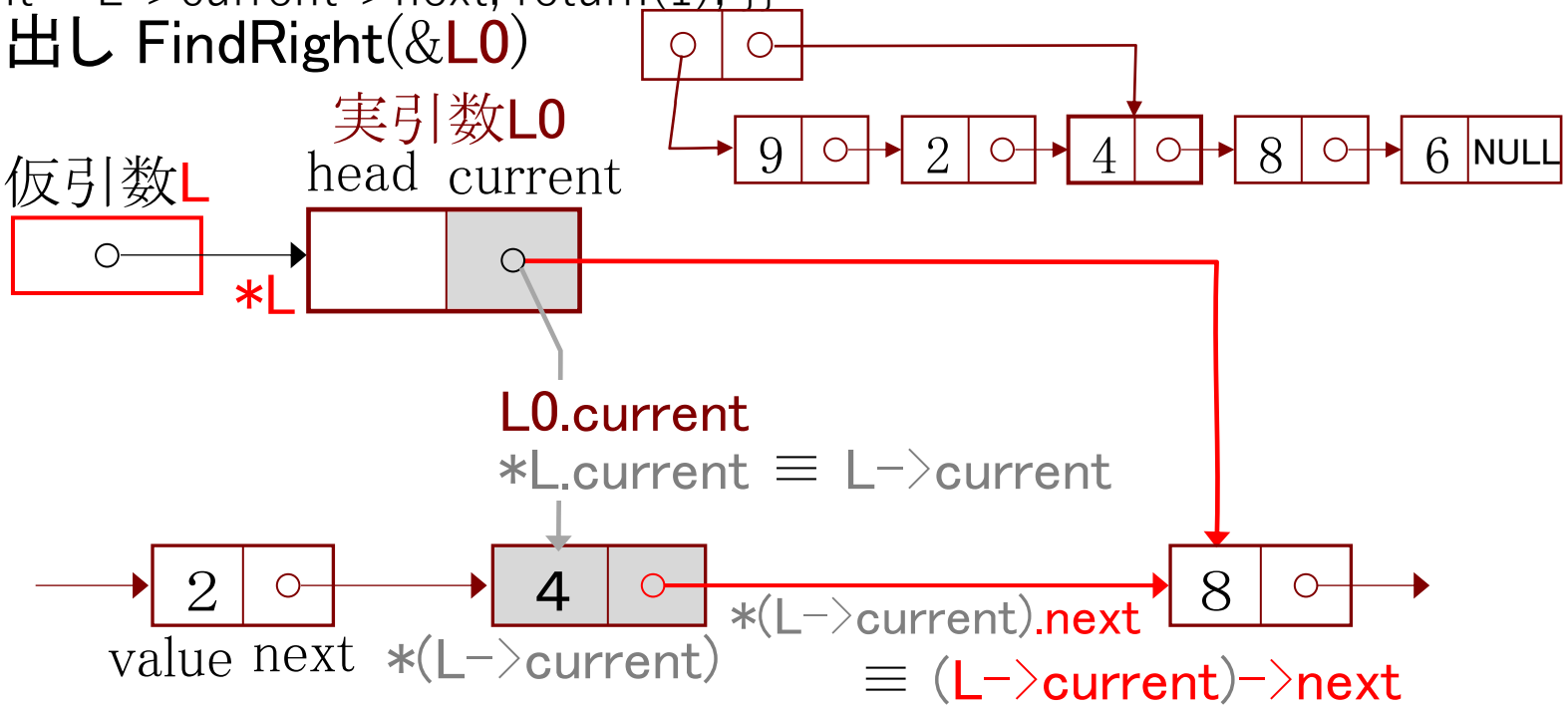
一方向連結リスト（第１版）：実現アルゴリズム

- リスト L0 のカレント要素を右隣にする：FindRight(&L0)

```
int FindRight(List *L){
   if(L->current == NULL) return(0);                    /* カレント要素がない */
   else if(L->current->next == NULL) return(0);         /* カレント要素が末尾 */
   else{
      L->current = L->current->next; return(1); }}
```

事前条件を満たさない!

呼出し FindRight(&L0)

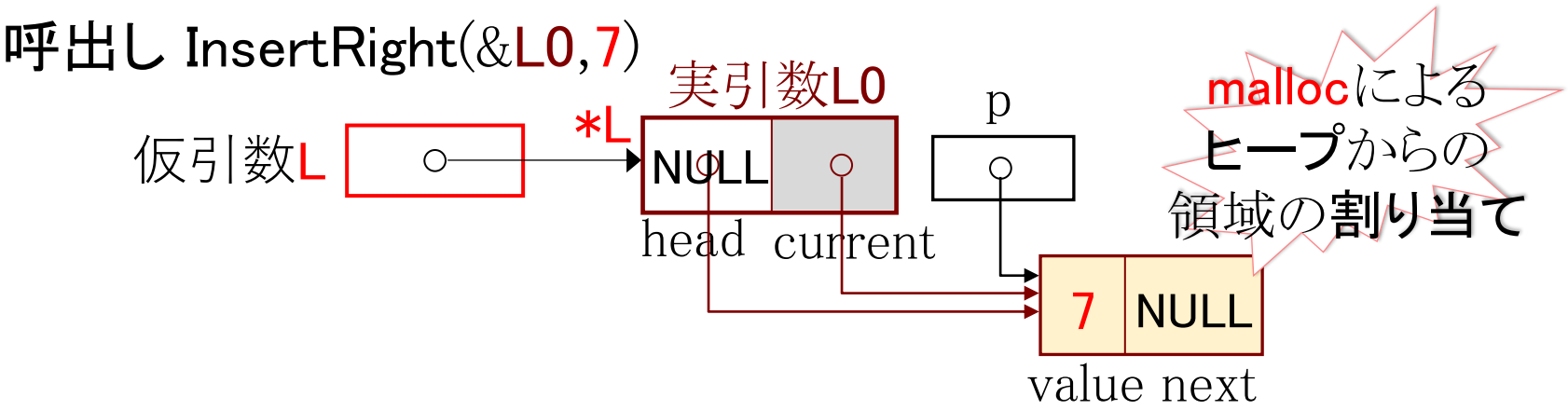


一方向連結リスト（第１版）：実現アルゴリズム

- リストL0のカレント要素の右側に7を挿入：InsertRight(&L0, 7)

```
int InsertRight(List *L, Element e){
    NodePointer p;
    if(L->head == NULL){                          /* 空リストの時 */
        p = malloc(sizeof(Node));                  /* 領域割り当て */
        p->value = e;  p->next = NULL;             /* 要素を生成 */
        L->head = p; L->current = p; return(1);  }  /* 要素の挿入 */
…つづく…
```

ヒープ領域

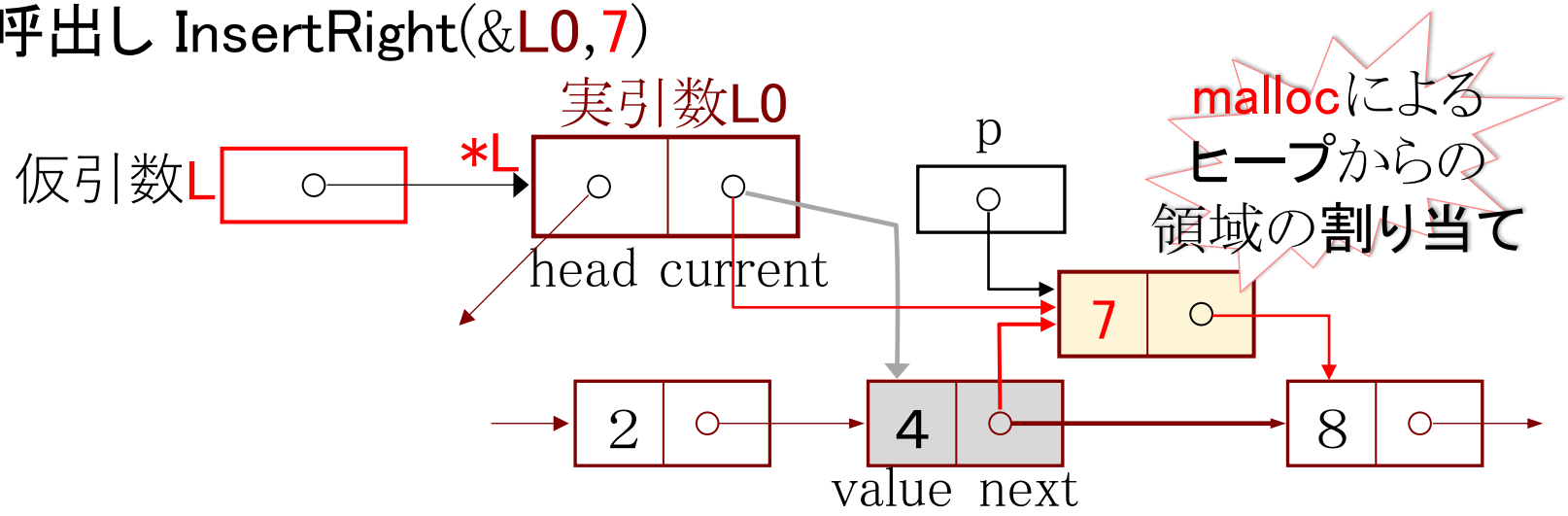


一方向連結リスト（第１版）：実現アルゴリズム

- リストL0のカレント要素の右側に7を挿入：InsertRight(&L0, 7)

…つづき…

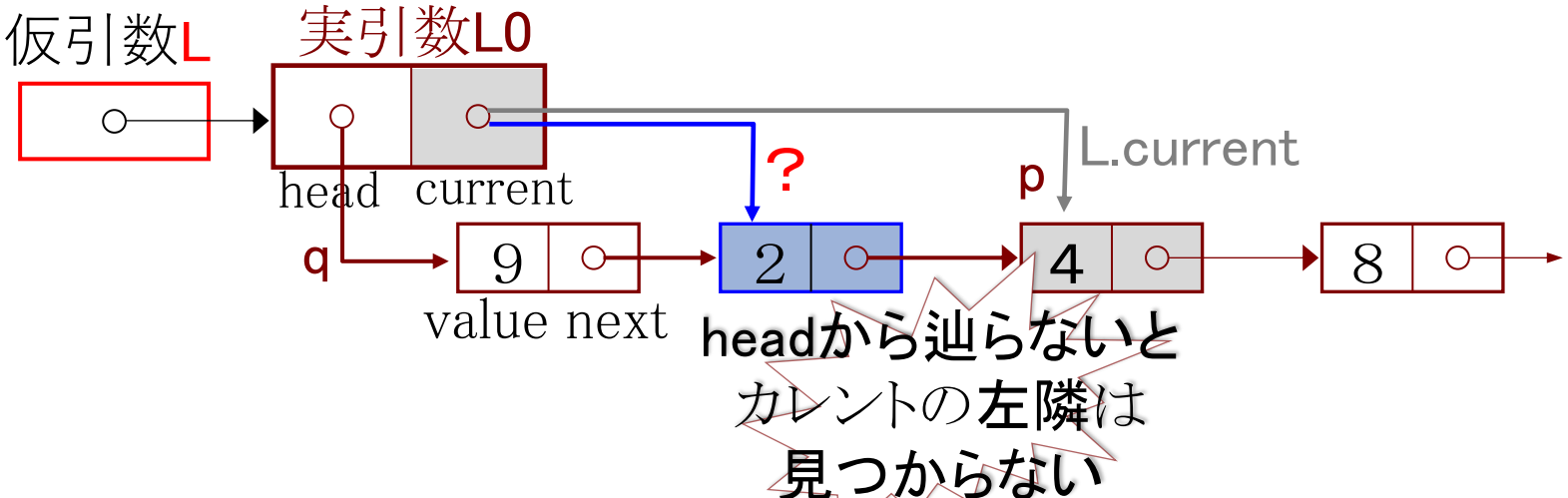
```
    else if(L->current == NULL) return(0);          /* カレントがないとき */
    else{                                           /* 空リストでなく，カレントがあるとき */
        p = malloc(sizeof(Node));  p->value = e;                /* 領域割り当て 要素の値を設定 */
        p->next = L->current->next; L-> current->next = p;    /* 要素の挿入 */
        L->current = p; return(1); }
}
```



一方向連結リスト（第１版）：実現アルゴリズム

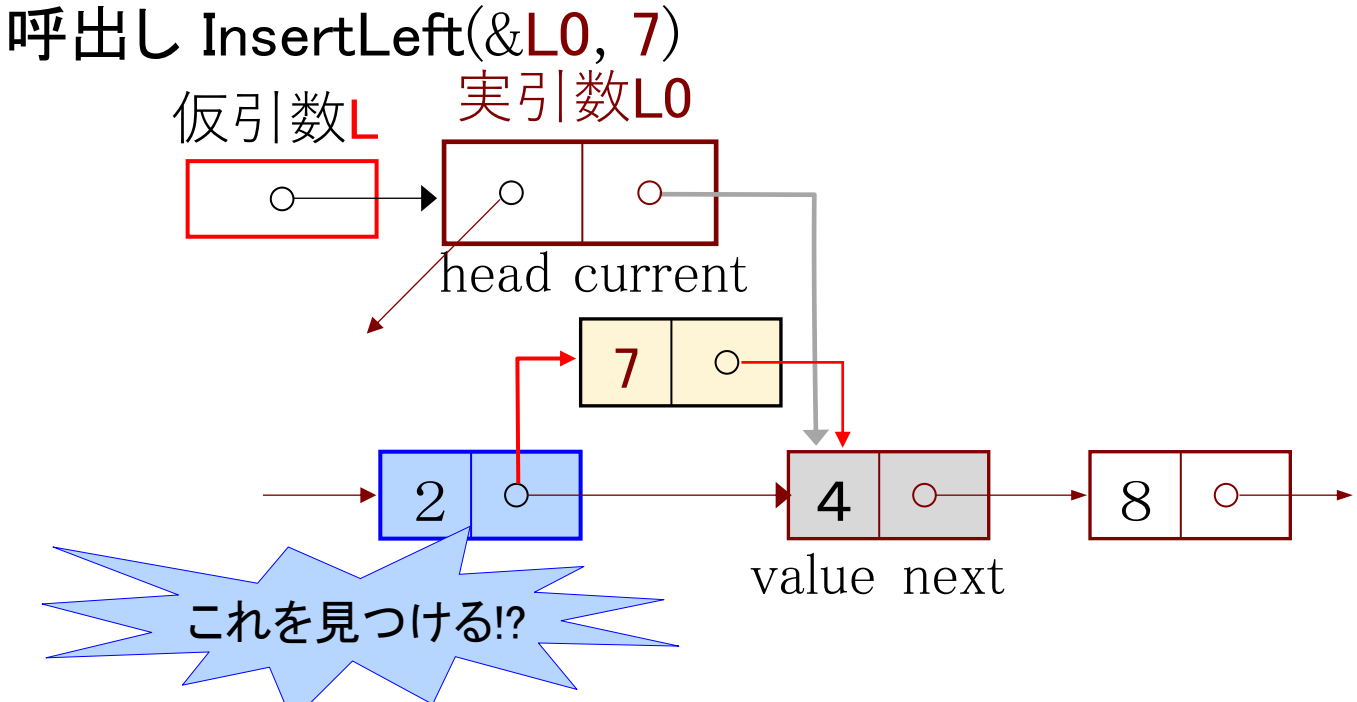
- リストL0のカレント要素を左隣にする：FindeLeft (&L0, 7)

```
int FindLeft(List *L){
    NodePointer p, q;
    if(L->current == NULL) return(0);          /* カレント要素がない */
    else if(L->current == L->head) return(0);  /* 先頭の要素がカレント */
    else{                                      /* カレント要素があり，先頭ではない */
        p = L->current; q = L->head;            /* カレントの左側を探す */
        while(q->next != p) q = q->next;
        L->current = q;  return(1);  }
}
```



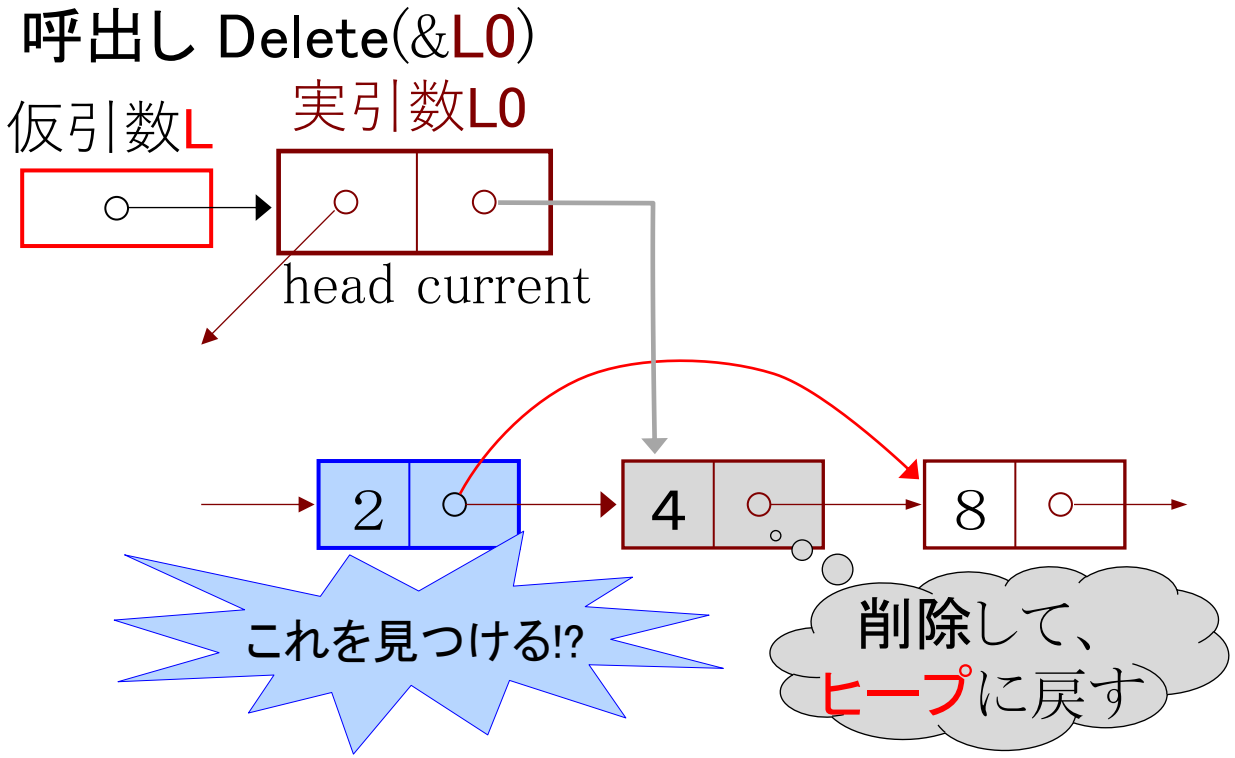
一方向連結リスト（第１版）：実現アルゴリズム

- リストL0のカレント要素の左隣にeを挿入する：InsertLeft (&L0, e)
- 挿入構造体はカレント要素の左隣の構造体が指す
  =>カレント要素の左隣の構造体を見つける
  …簡単にはたどれない



一方向連結リスト（第１版）：実現アルゴリズム

- リストL0のカレント要素を削除：Delete (&L0)
- 削除要素（カレント要素）の左側の構造体をみつける
  …簡単にはたどれない



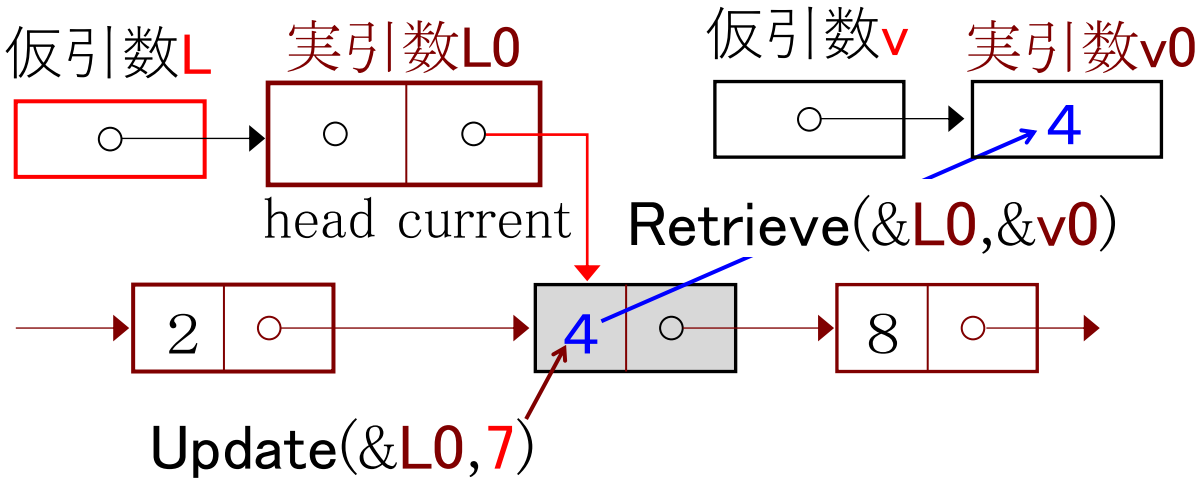
一方向連結リスト（第１版）：実現アルゴリズム

- リストLのカレント要素の値を求める：Retrieve(&L0, &v0)

```
int Retrieve(List *L, Element *v){
    if(L->current == NULL) return(0);   /* カレント要素なし */
    else{ *v = L->current->value; return(1) }}
```

- リストLのカレント要素の値を置き換える：Update(&L0, e)

```
int Update(List *L, Element e){
    if(L->current == NULL) return(0);   /* カレント要素なし */
    else{ L->current->value = e; return(1) }}
```



一方向連結リスト（第１版）：実現アルゴリズムの効率

- カレント要素の右隣に要素を挿入（InsertRight）
- カレント要素の右隣をカレント要素に（FindRight）
  - 一定時間

- カレント要素の左隣に要素を挿入（InsertLeft）
- カレント要素を削除（Delete）
- カレント要素の左隣をカレント要素に（FindLeft）
  - 要素数nに比例する時間 => これらはいまいち 改良へ

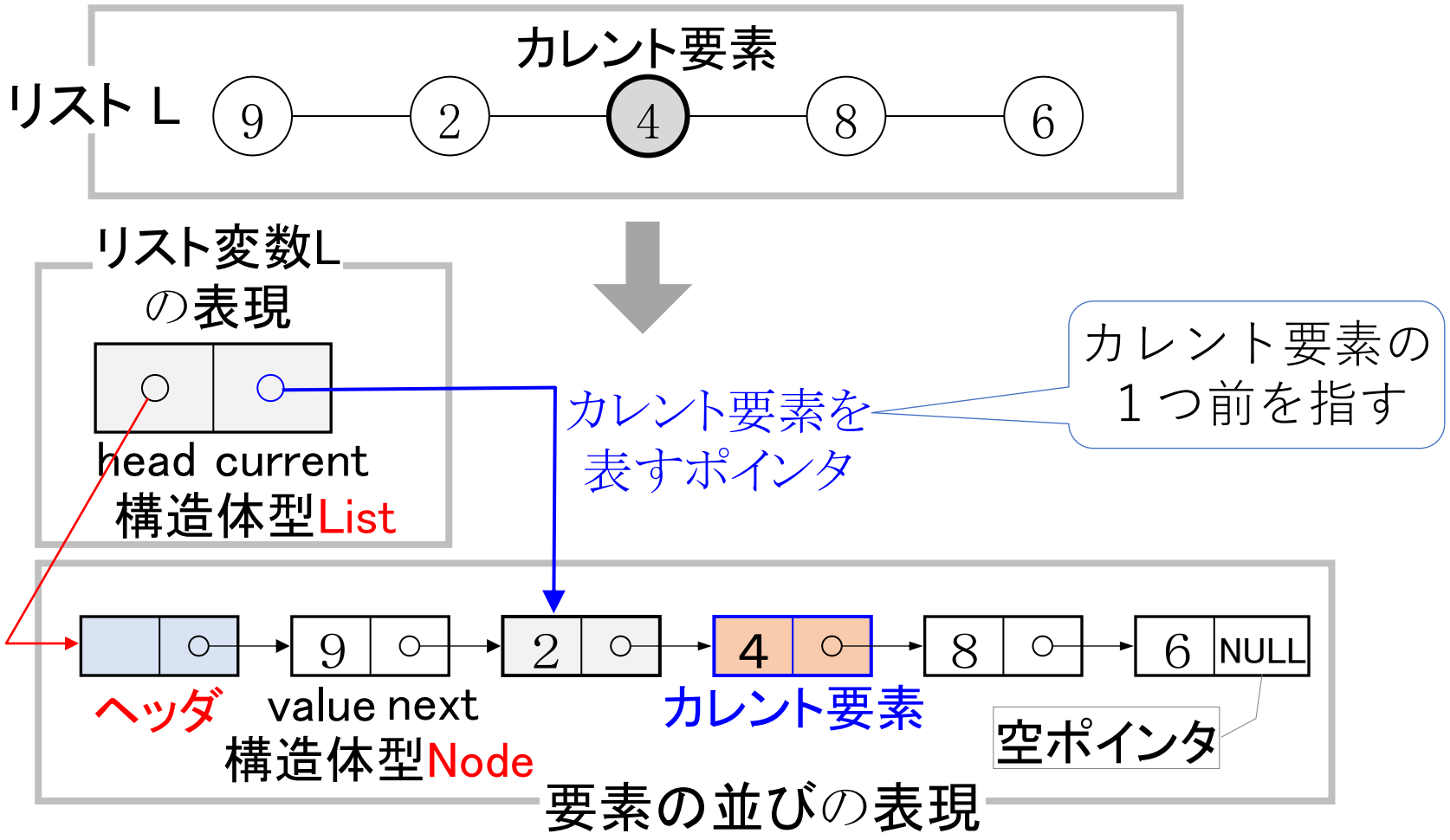
- Findith以外のほかの捜査は一定時間

一方向連結リスト（第２版）

- 第１版の欠点
  - カレント要素の左隣に要素を挿入（InsertLeft）
  - カレント要素の削除（Delete）
  - カレント要素の左隣をカレントにする（FindLeft）

  =>リスト長の時間がかかる：効率が悪い
- 改良
  - カレント要素へのポインタ
    - カレント要素を指す => カレント要素の左側を指す（カレント要素の左に注目）
- 表現
  - 「要素の並びの表現」は，先頭にヘッダと呼ばれるダミー要素をつなげる

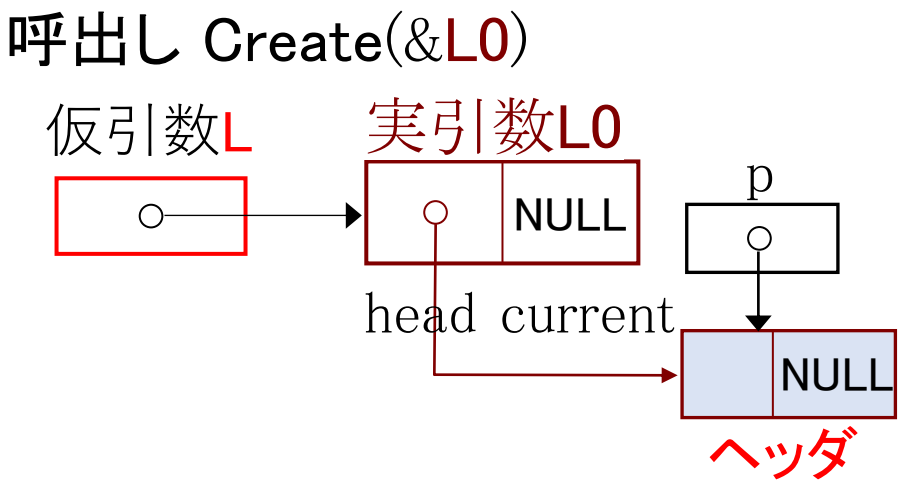
一方向連結リスト（第２版）：表現



一方向連結リスト（第２版）：実現アルゴリズム

- リストL0を空リストにする：Create(&L0)

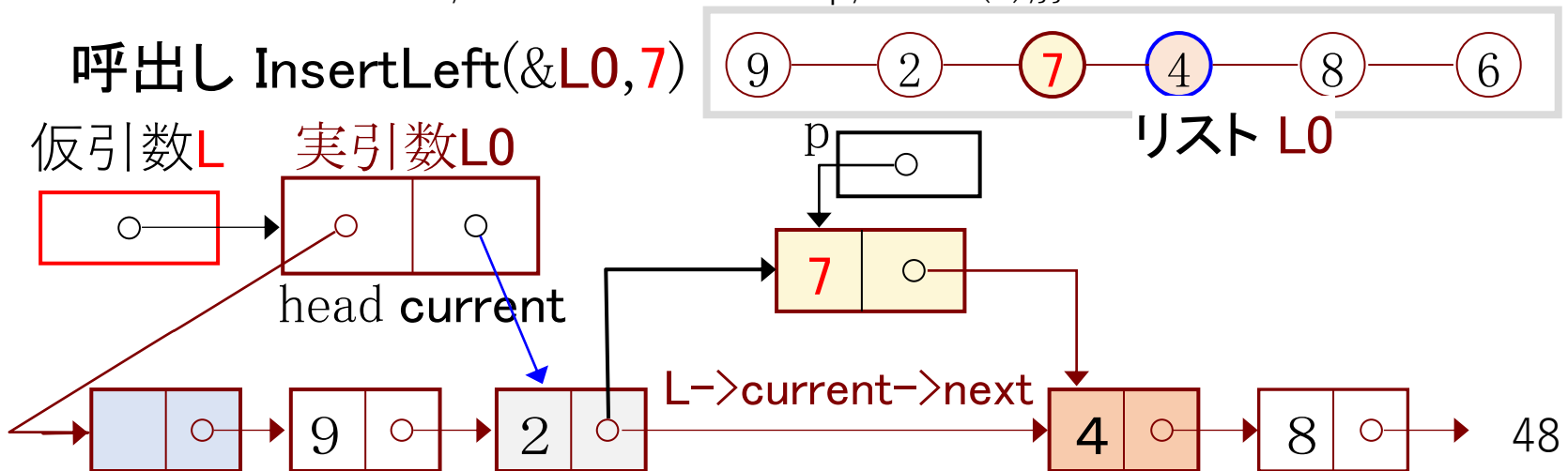
```
void Create(List *L){
    NodePointer p;
    p = malloc(sizeof(Node));
    p->next = NULL;                /* ヘッダだけからなる空リスト */
    L->head = p;
    L->current = NULL;}            /* 空リストはcurrentがNULL */
```



一方向連結リスト（第２版）：実現アルゴリズム

- リストL0をカレント要素の左隣にを挿入：InsertLeft(&L0, 7)

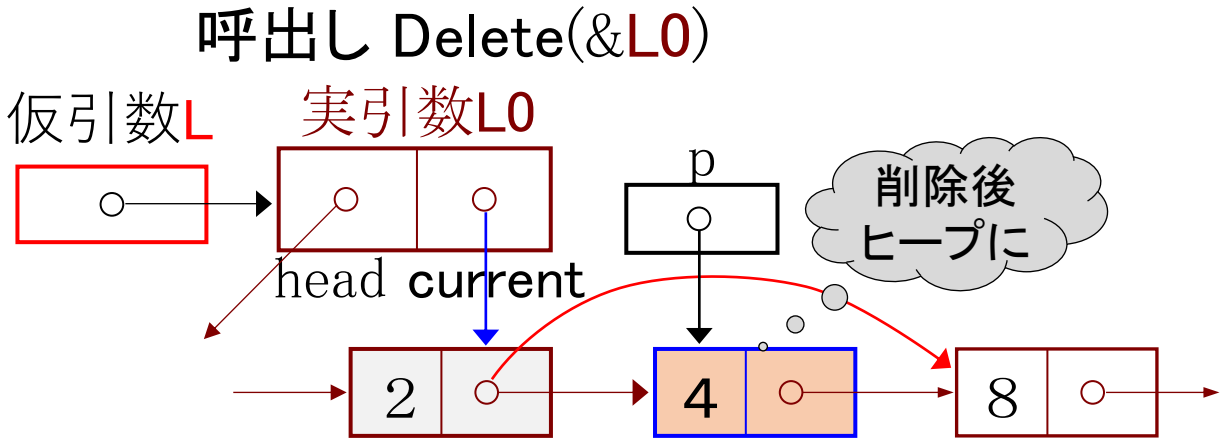
```
int InsertLeft(List *L, Element e){
   Node Pointer p;
   if (L->current == NULL){
      if(L->head->next != NULL) return(0);          /* カレント要素なし */
      else{                                         /* 空リストに挿入*/
         p = malloc(sizeof(Node));  p->value = e;
         p->next = NULL; L-> head->next = p;
         L->current = L->head; return(1);    }
   else{                                            /* 非空リストに挿入 */
      p = malloc(sizeof(Node)); p->value = e;
      p->next = L->current->next; L->current->next = p; return(1);}}
```



一方向連結リスト（第２版）：実現アルゴリズム

- リストL0のカレント要素を削除：Delete (&L0)

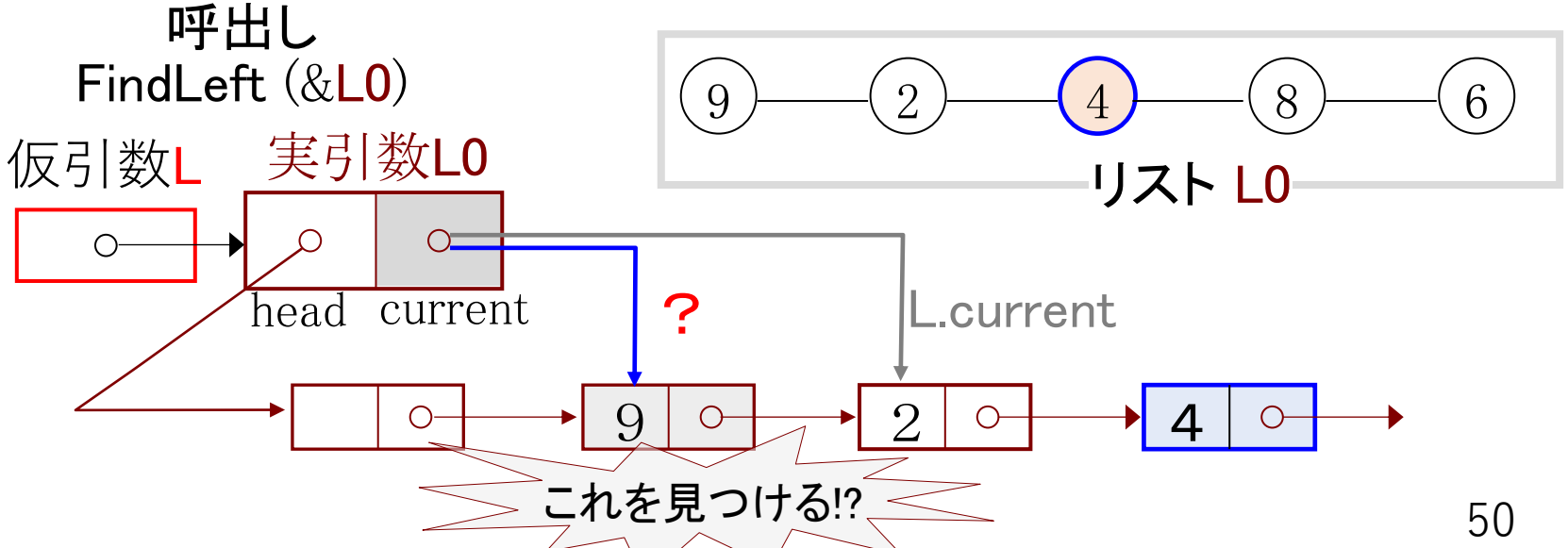
```
int Delete(List *L){
    NodePointer p;
    if(L->current == NULL) return(0);       /* カレント要素なし */
    else{
        p = L->current->next;
        L->current->next = p->next;         /* カレントを削除 */
        free(p);                            /* 削除要素をヒープへ戻す */
        if(L->current->next ==NULL)
            L->current = NULL;              /* カレント要素がなくなる */
        return(1);}
```



一方向連結リスト（第２版）：実現アルゴリズム

- リストL0のカレント要素を左隣にする：FindLeft (&L0)

```
int FindLeft(List *L){
    NodePointer p, q;
        if(L->current == NULL) return(0);          /* カレント要素なし */
        else if(L->current == L->head) return(0);  /* 先頭がカレント */
        else{
            p = L->current; q = L->head;
            while (q->next != p) q = q->next;      /* カレントの左を探す */
            L->current = q; return(1);  }}
```



一方向連結リスト（第２版）：効率

- カレント要素の隣に要素を挿入（InsertRight, InsertLeft）
- カレント要素の削除（Delete）
- カレント要素の右側を要素にする（FindRight）
  - 一定時間

- カレント要素の左側をカレント要素にする（FindLeft）
  - 要素数nに比例する時間 => 改良できないか？

- Findith以外のほかの捜査は一定時間

双方向連結リスト

- 一方向連結リスト（第２版）
  - カレント要素を左隣へ移動（FIndLeft）以外は効率がよい

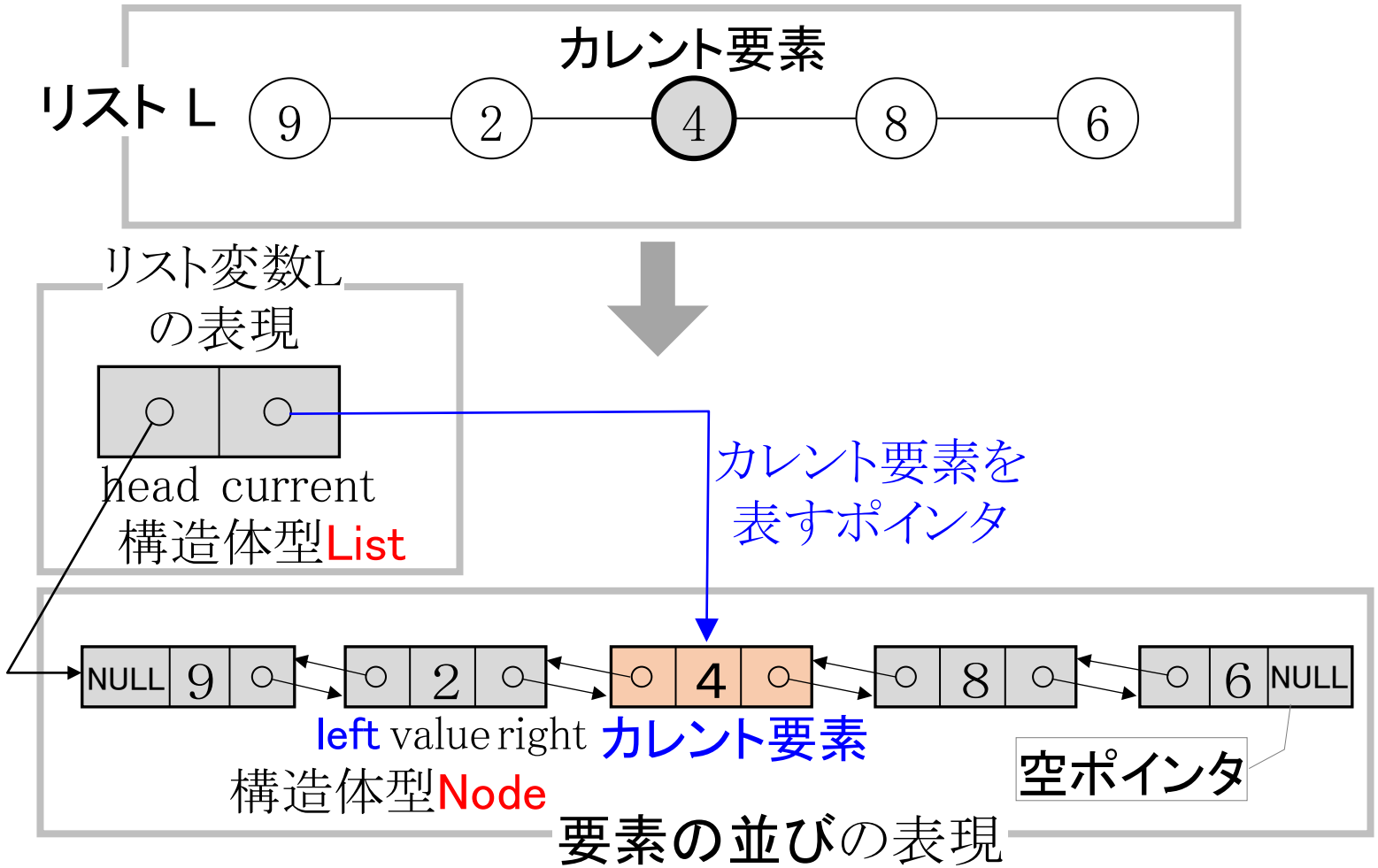
  ⇒FindLeftも一定時間へできないか？

  ⇒左側の要素を指すポインタの導入

- 表現

```
typedef struct node_tag{
    Element value;
    struct node_tag *left, *right; } Node;
typedef Node *NodePointer;
```

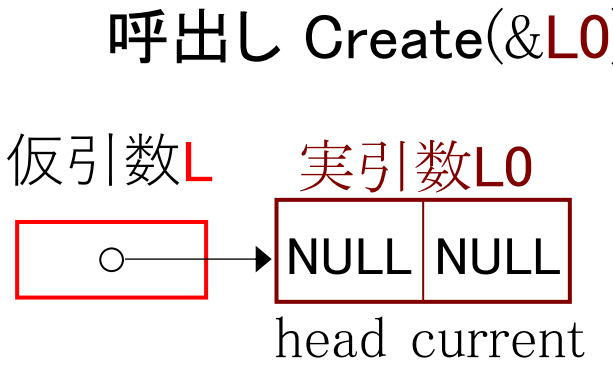
双方向連結リスト：表現


カレント要素
リスト L
9
2
4
8
6
リスト変数L
の表現
head current
構造体型List
カレント要素を
表すポインタ
NULL
9
2
4
8
6
NULL
left value right
カレント要素
構造体型Node
空ポインタ
要素の並びの表現

双方向連結リスト：実現アルゴリズム

- リストL0を空リストにする：Create(&L0)

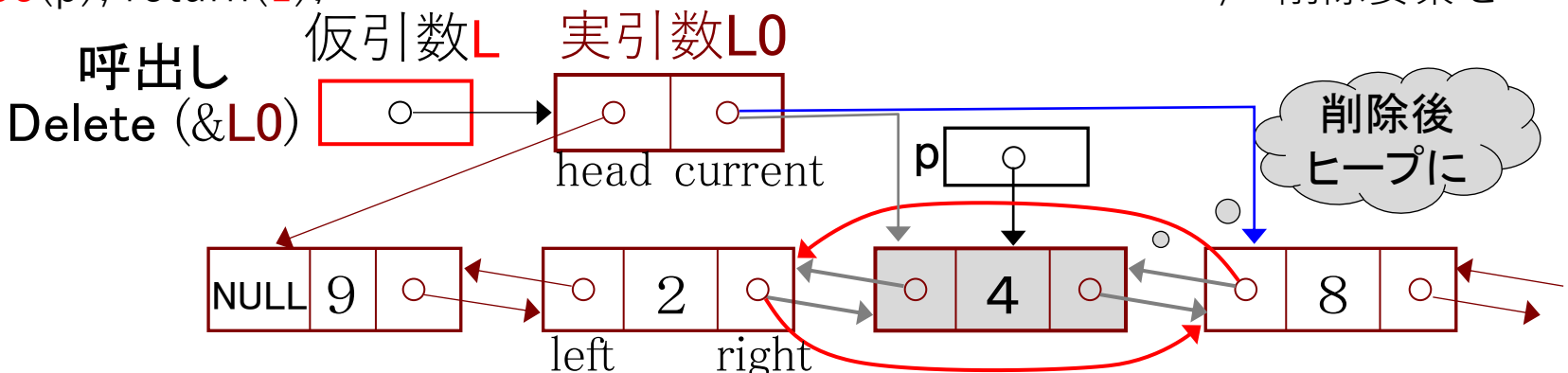
```
void Create(List +L){
    L->head = NULL;                 /* 要素数が0を表す */
    L-> current = NULL; /* カレント要素はない */
}
```



双方向連結リスト：実現アルゴリズム

- リストL0のカレント要素を削除：Delete(&L0)

```
int Delete(List *L){
    NodePointer p, q, r;
    if(L->current == NULL) return(0);                    /*カレント要素なし*/
    else if(L->head->right = NULL){                      /*ただひとつの要素を削除*/
        p= L->current; L->head = L->current =NULL;}
    else {                                               /*要素が2個以上*/
        p = L->current; q = L->current->left; r = L->current->right;
        if(q == NULL){                                   /*先頭要素の削除*/
            r->left = NULL; L->head = L->current = r;}
        else if(r == NULL){                              /*末尾要素の柵状*/
            q->right = NULL; L->current = NULL;}
        else{ q->right = r; r->left = q; L->current = r; } }
    free(p); return(1);                                  /* 削除要素をヒープに戻す */
```

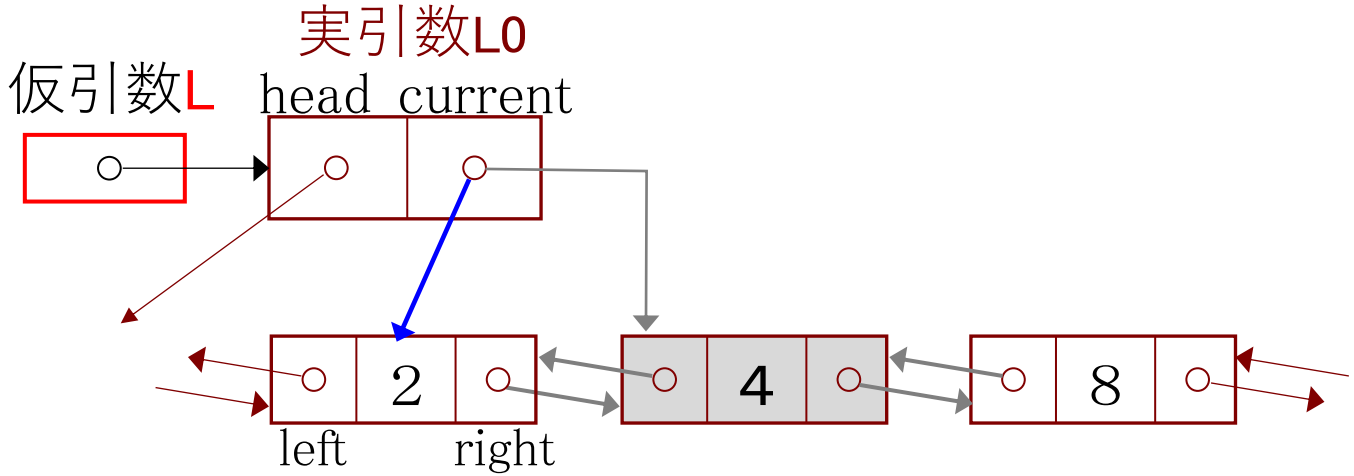


双方向連結リスト：実現アルゴリズム

- リストL0のカレント要素を左隣にする：FindLeft (&L0)

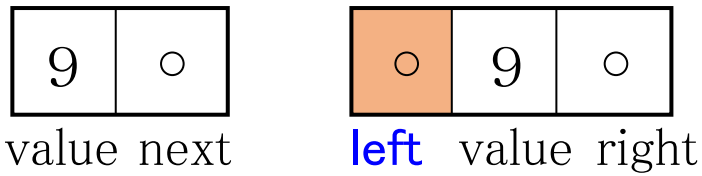
```
int FindLeft(List *L){
    if(L->current == NULL) return(0);              /*カレント要素はなし*/
    else if(L->current == L->head) return(0);      /*カレント要素は先頭要素*/
    else{
        L-> current = L ->left; return(1); }
}
```

呼出し FindLeft(&L0)



双方向連結リスト：効率

- 左要素への移動（FindLeft）も一定時間
- すべての操作（Findith以外）が一定時間
- 時間計算量は，双方向連結リストによる実現が優れている
- 領域計算量は，要素を表す構造体が2つのポインタを持つため，一番効率が悪い



リストの実現：効率比較

- 時間計算量

	FindRight	FindLeft	InsertRight	InsertLeft	Delete	Findith	そのた
ベタ詰め	一定	一定	nに比例	nに比例	nに比例	一定	一定
一方向1版	一定	nに比例	一定	nに比例	nに比例	nに比例	一定
一方向2版	一定	nに比例	一定	一定	一定	nに比例	一定
双方向	一定	一定	一定	一定	一定	nに比例	一定

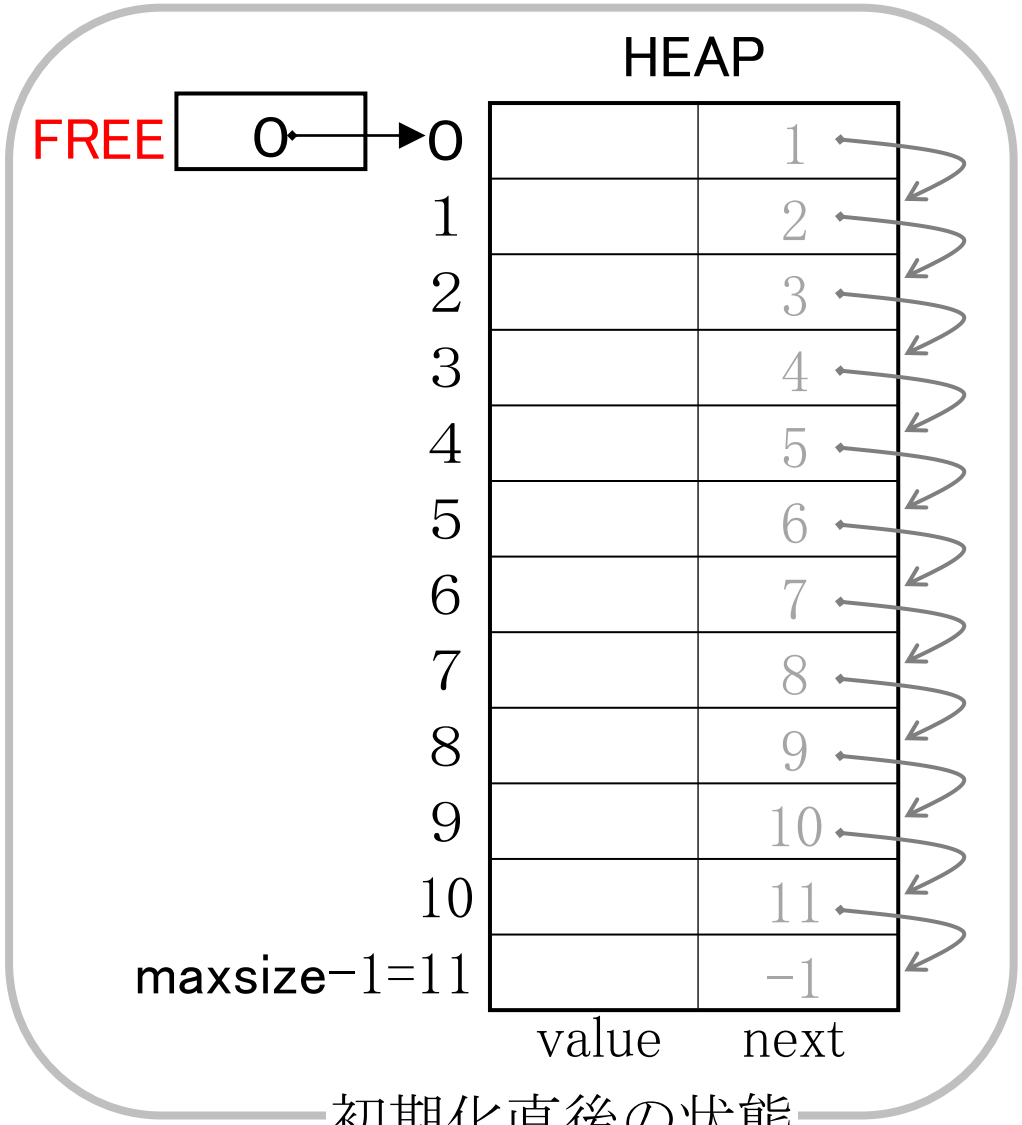
- 領域計算量
  - 配列へのベタ詰め以外は n に比例する
  - 双方向連結リストの場合，ポインタフィールドが2つあり効率が悪い

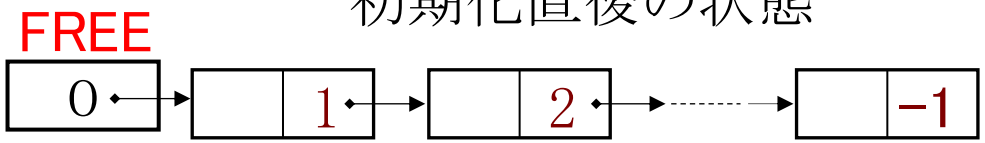
連結リストの配列とindexによる実現

- 配列なのでポインタを持たない
- ヒープを効率よく実現していないプログラミング言語で有効
  ⇒配列のindexを使ってポインタを実現
  ⇒配列を用いて，ヒープを模倣できる
- 表現
  - ヒープ領域：配列（配列HEAPという）
  - 連結リスト要素：配列の構造体
    - 値フィールド：value
    - ポインタのフィールド：next
  - ポインタ：index 0からmaxsize-1
  - 空ポインタ値NULL: -1（整数値）

空き領域リスト

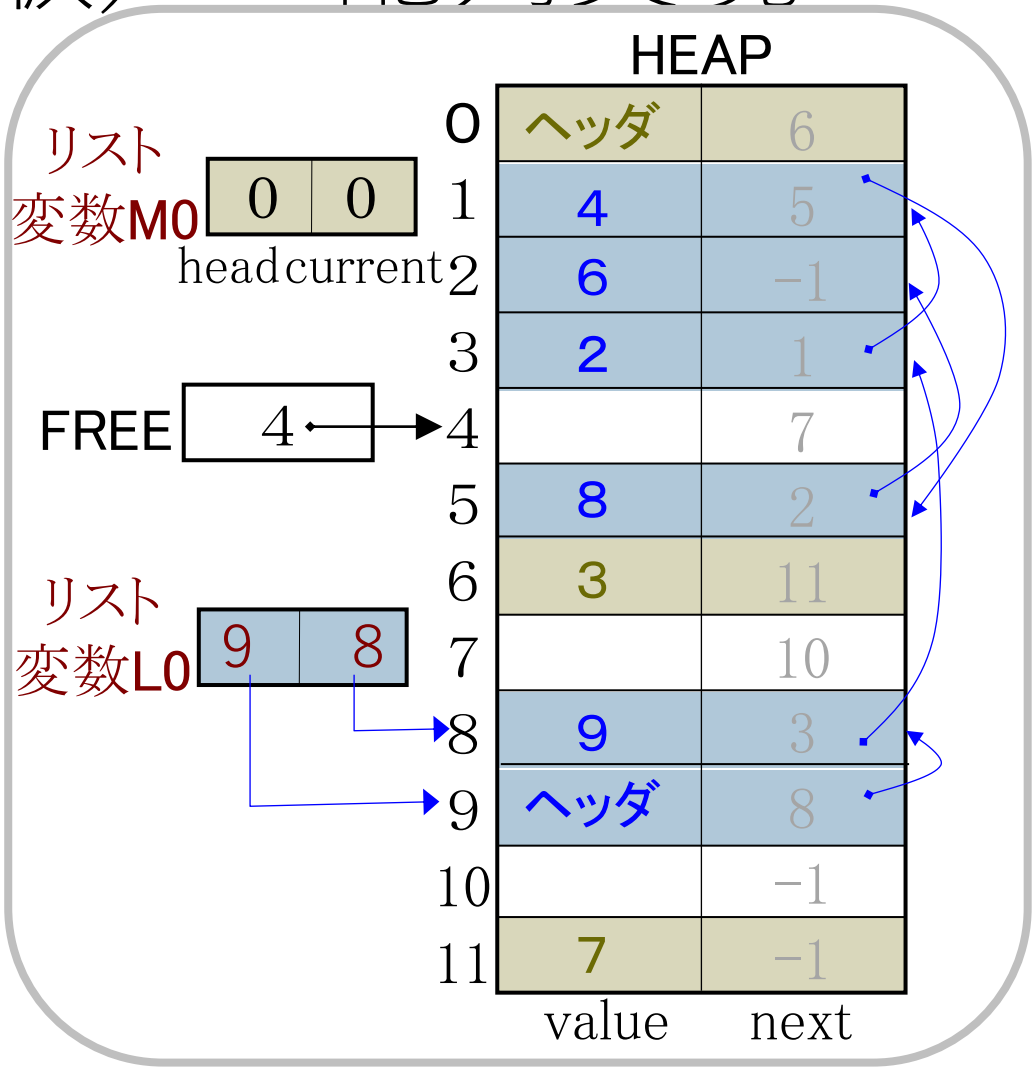
- ヒープの実現 HEAP
- 空き領域リスト
  - ヒープの未使用領域
- 配列HEAP
  - すべての要素を１つにつないだ連結リストに初期化
- 先頭要素をポインタ変数FREEが指す
- 新しい要素が必要になったときここから持ってくる
- 削除して不要になった要素は，再利用のためここにつなげる

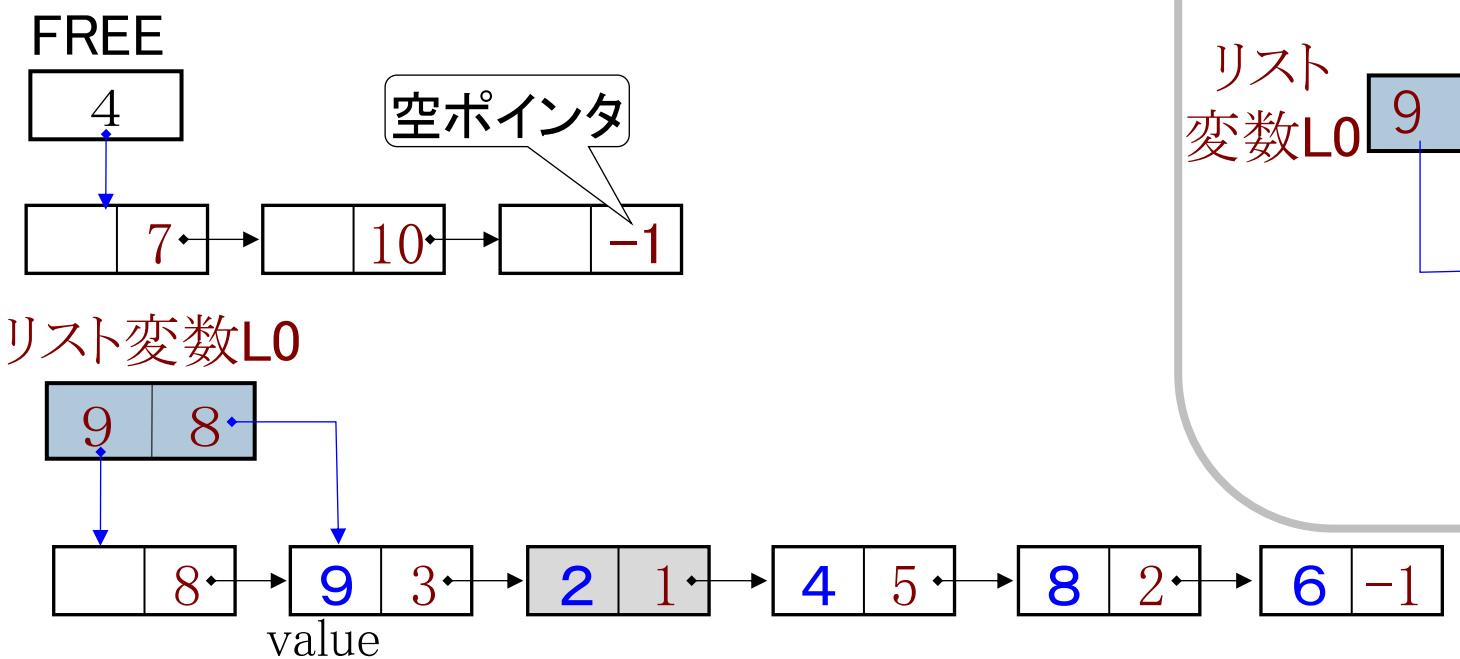




一方向連結リスト（第２版）の配列実現

- リストはヘッダを表す配列要素で始まる
- 各要素がポインタの代わりにindexでつながれる
- 最後の要素をにはNULLを表す-1が入る

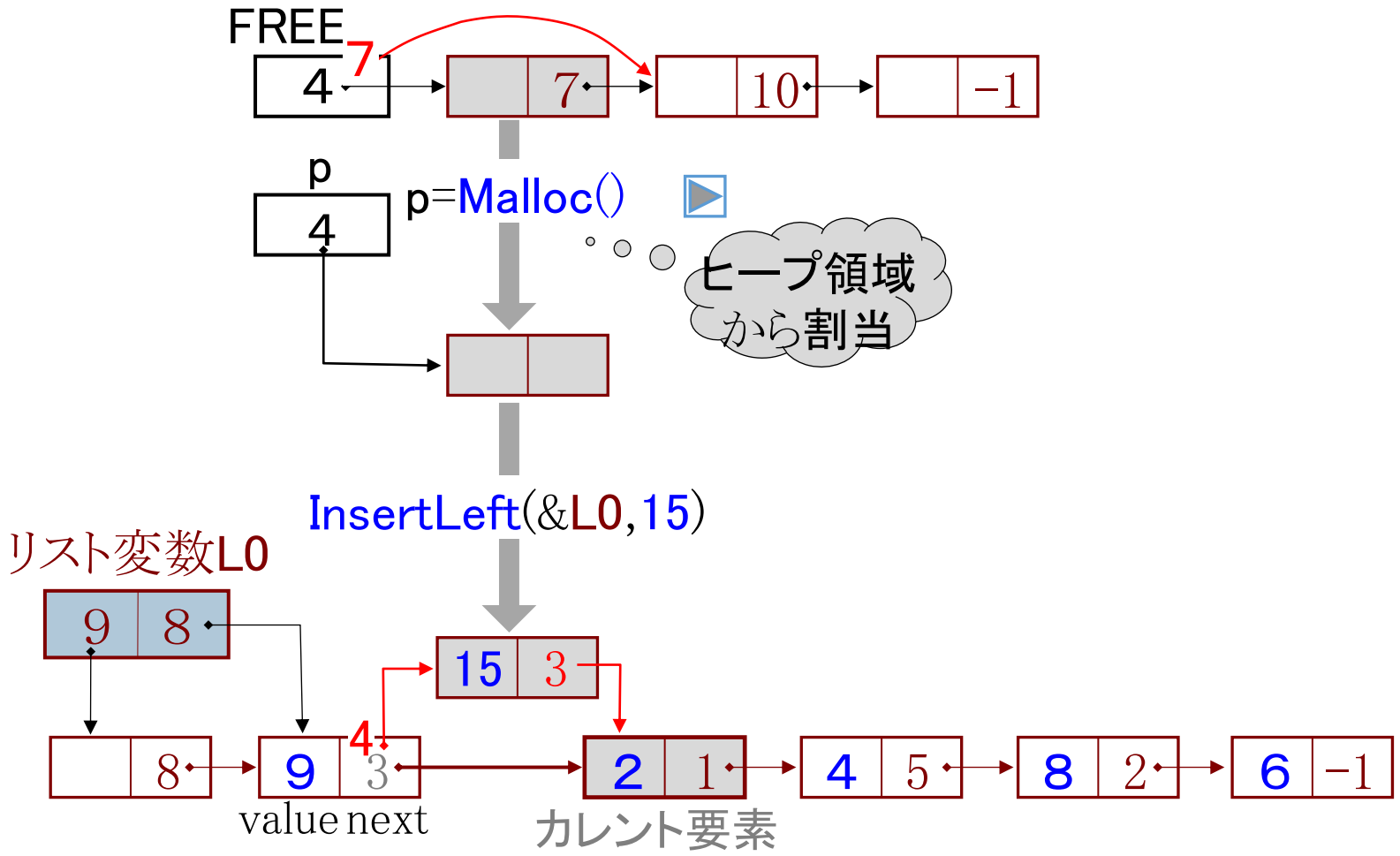




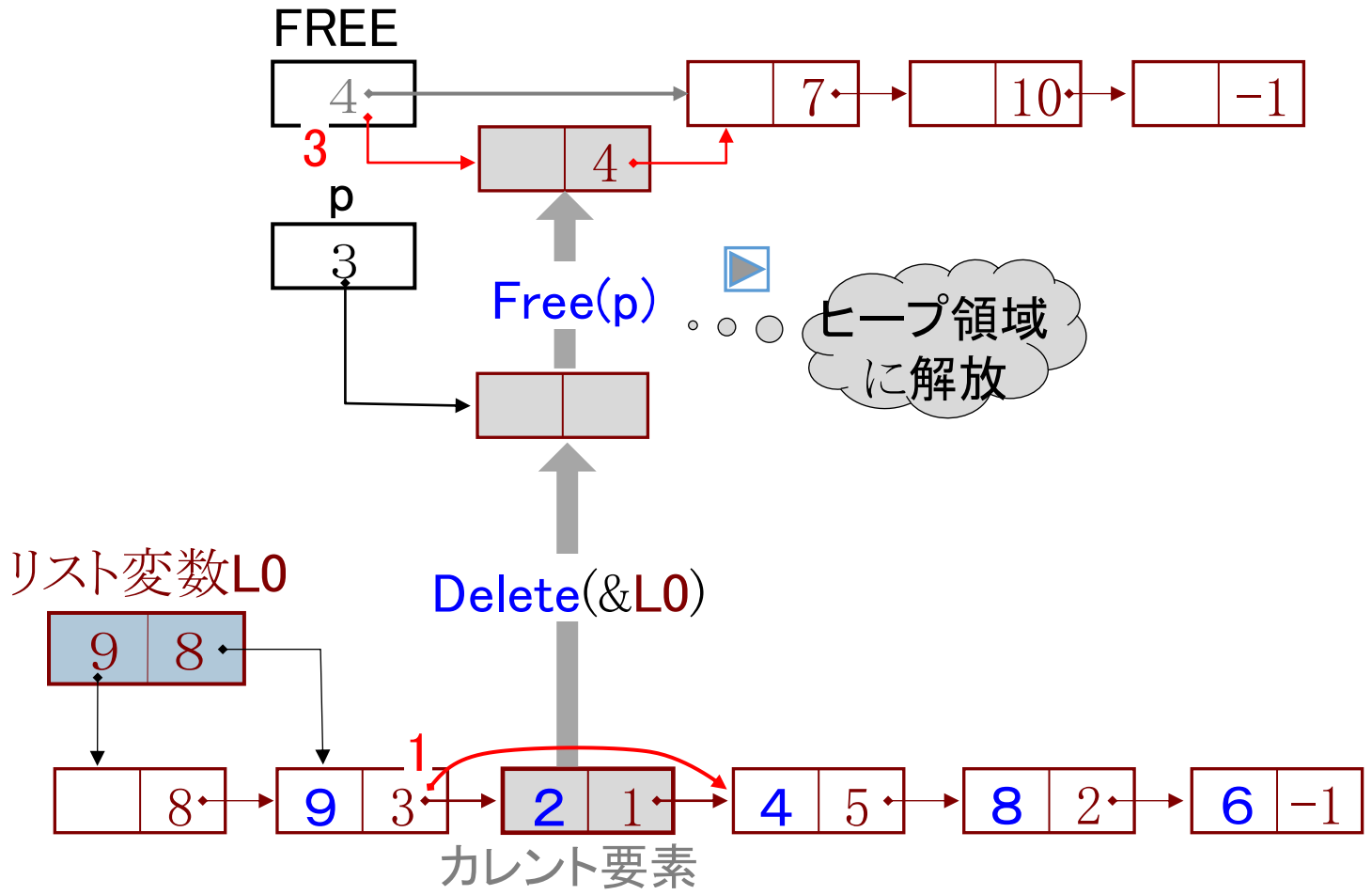
空き領域リストとリストの表現

```
#define NULL -1                    空ポインタ値（システムではNULLが規定のため）
#define maxsize 12                 ヒープの最大要素数
typedef int NodePointer;           リスト要素を指すポインタの型
typedef int Element;               リストの要素型
typedef struct{                    リスト変数を表す型
    NodePointer head;
    NodePointer current; } List;
typedef struct {                   リスト要素を表す型
    Element value;
    NodePointer next; } Node;
Node Heap[maxsize];                ヒープを表す配列名
NodePointer FREE;                  空き領域の先頭要素を指す変数
List L0, M0;                       利用者が使うリスト変数
```

利用者リストと空き領域リスト（ヒープ）のやりとり：Malloc, InsertLeft



利用者リストと空き領域リスト（ヒープ）のやりとり：Free, Delete



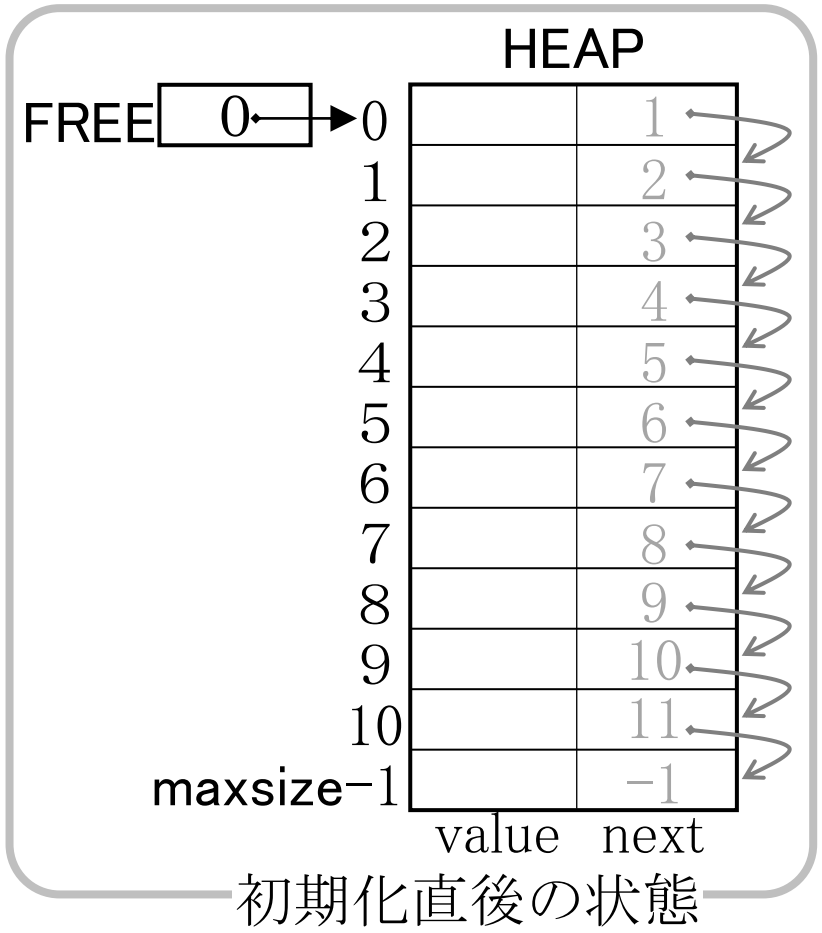
ヒープ実現アルゴリズム

- 配列でヒープを実現するための手続き
  - 配列HEAPを空き領域リストに初期化する：HeapInit
  - 空き領域リストから新しい要素を取り出す：Malloc
  - 空き領域に不要要素を戻す：Free

ヒープ実現アルゴリズム：HeapInit

- 配列HEAPを空き領域に初期化

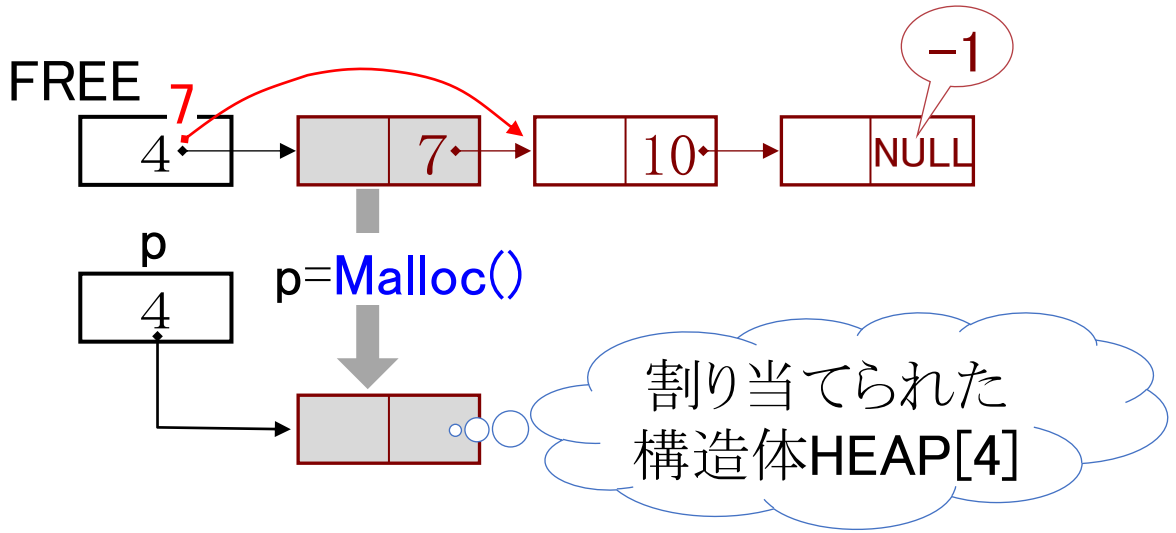
```
void HeapInit(){
    NodePointer i;
    FREE = 0;
    for(i=0; i<maxsize-1; i++)
        HEAP[i].next = i+1;
    HEAP[maxsize-1].next = -1;
}
```



ヒープ実現アルゴリズム：Malloc

- 空き領域リストから新しい要素を取り出す

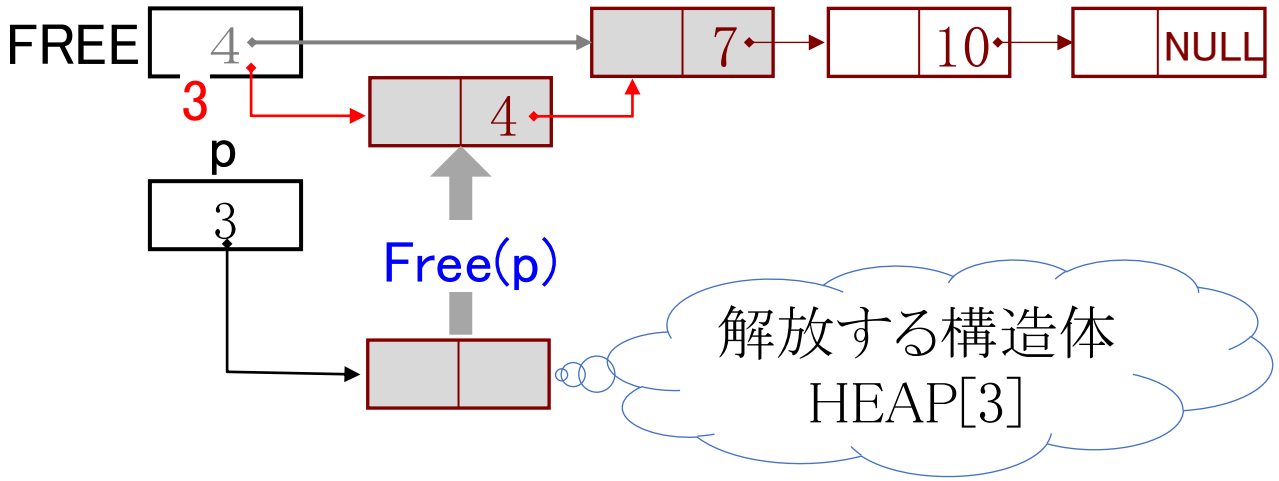
```
int Malloc(void){
    NodePointer p;
    if(FREE == NULL) ERROR(“空き領域がない”);
    else {p = FREE; FREE = HEAP[FREE].next; }
    return(p);
}
```



ヒープ実現アルゴリズム：Free

- 不要要素を空き領域リストに戻す

```
void Free(NodePointer p){      /* 空き領域用変数FREEとは異なる */
    Heap[p].next = FREE;
    FREE = p;
}
```



リスト操作の実現アルゴリズム

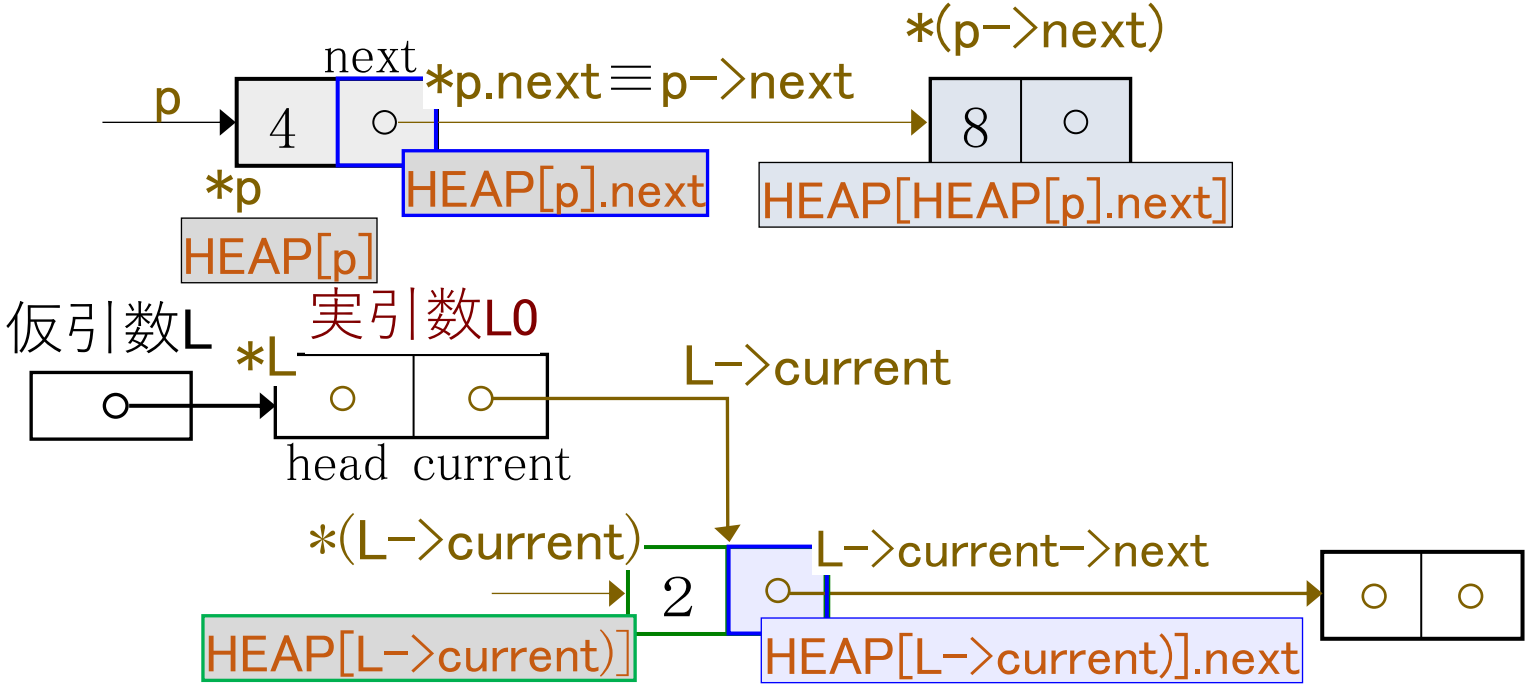
- リスト操作の実現アルゴリズムはヒープ操作（HeapInt, Malloc, Free）を用いた記述
- 連結リストによる実現アルゴリズムとほぼ同じ

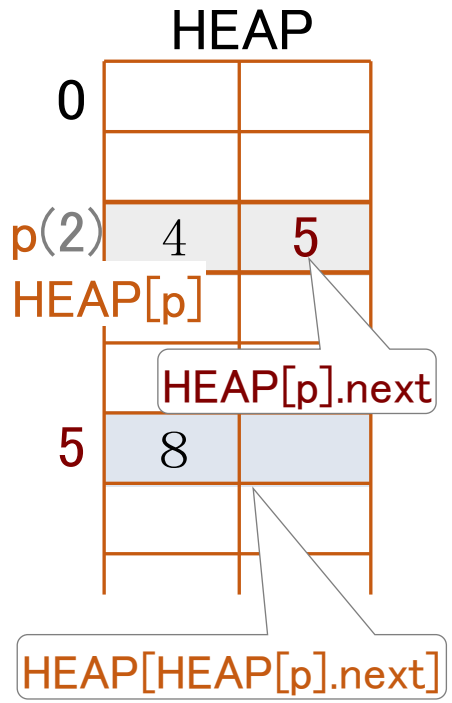
書き換え

```
int InsertLeft(List *L, Element e){
    NodePointer p;
    if(L->current == NULL){
        if( L->head->next != NULL) return(0);          /* カレント要素がない*/
          HEAP[L->head].next
        else{                                                  /* 空リストに挿入 */
            p =malloc(sizeof(Node)); p->value =e; p->next = NULL; L->head->next = p;
                    Malloc()        HEAP[p].value  HEAP[p].next     HEAP[L->head].next
            L->current = L->head; return(1); }
    else{                                                  /* 非空リストに挿入 */
        p = malloc(sizeof(Node)); p->value = e;  p->next  =  L->current ->next;
                    Malloc()        HEAP[p].value  HEAP[p].next   HEAP[L->current].next
        L->current->next = p; return(1); } }
     HEAP[L->current].next
```

実現アルゴリズムの効率

- ヒープの管理操作以外は
  - リストのポインタ表現「ポインタpで指される変数*p」を「インデックスpの配列要素の変数HEAP[p]」で置き換え
    *(L->current) ⇒ HEAP[L->current}
    L->current->next ⇒ HEAP[L->current].next
    ≡ *(L->current).next
- 実現の操作計算時間は，連結リストの形に依存





まとめ

- リストとは
- 実現アルゴリズム
  - 配列ベタ詰め
  - 構造体とポインタ
    - 一方向連結リスト（第1版）
    - 一方向連結リスト（第2版）
    - 双方向連結リスト
  - 配列とインデックスを用いた連結リスト（一方向第2版）

演習課題

- C言語でリストをつくってみよう
- どのような実現アルゴリズムで作ってもらってもかまいません
- 最低１つは作ってください
  - いろいろな種類を作ると勉強になります
  - 以下のもの以外の実現でもOK

  - 配列ベタ詰め
  - 構造体とポインタ
    - 一方向連結リスト（第1版）
    - 一方向連結リスト（第2版）
    - 双方向連結リスト
  - 配列とインデックスを用いた連結リスト（一方向第2版）

提出について

- LETUSにて
- 提出物：ソースコードのファイル
- 2023/6/5 10:30まで