1　ヒープ

ヒープとは、優先度つき待ち行列、ヒープソートなどで用いられる抽象データ型（ADT）であり、たいていのアルゴリズムの教科書に載っている。

ヒープの構造は**図 1**のようなツリーであり、葉は同じ高さで左側から順に詰められる。あるノードの子の値は、自分よりも大きい値をもつ。したがって、根は常に最小値をもつノードである。

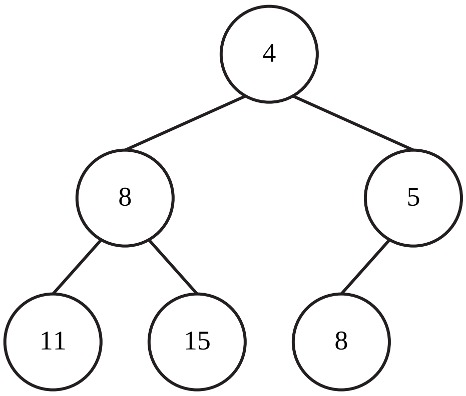

図 1　ヒープの例。

特徴としては、ノードの挿入（Insert）、最小値をもつノード（根）の取り出し（TakeMin）という 2 つの操作が、最悪でも $O(\log n)$ で実行できるということである。しかし、任意のノードを削除する操作（Delete）については、削除後のヒープの再構成が困難であるため、ヒープには適さない。

1.1　ノードの挿入

ヒープへ新しいノードを挿入する場合、まず挿入点（IP）に追加する。IP とは、**図 2**のように、そのツリーで最も深い葉で、最も右側にある葉の右隣の位置である。しかし、その位置がツリーからはみ出す場合は、最も左側にある葉の、左側の子の位置である。

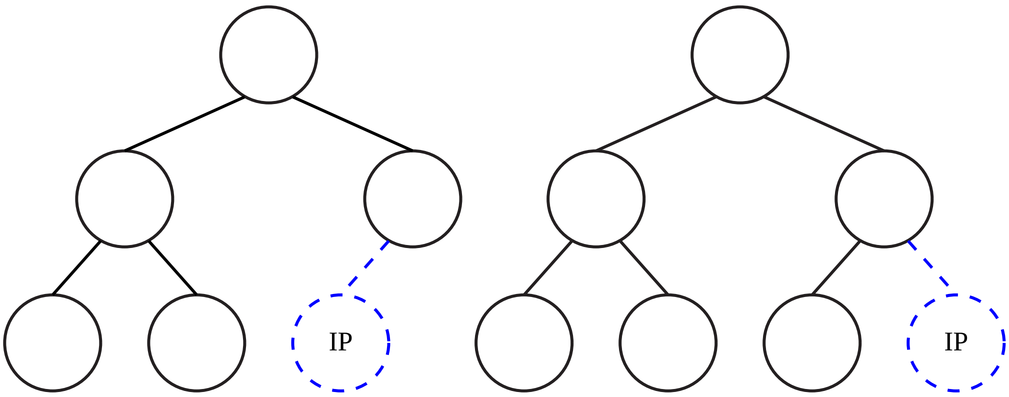

図 2　IP の位置の例。

挿入後はヒープの再構成をする必要がある。まず IP の位置に挿入されたノードがもつ値と、その親のノードがもつ値を比較し、順序が逆であれば、位置を交換する。この操作を再帰的に繰り返し、交換が起きない場合か、根と交換した場合に、ヒープの再構成は終了する。

1.2　最小値をもつノードの取り出し

ヒープから最小値を取り出す場合、まず根を取り出すと、それが最小値をもつノードである。

取り出し後もヒープの再構成をする必要がある。最初に、最終ノード（LN）を根の位置に移動する。LN とは、**図 3**のように、そのツリーで最も深い葉のうち、最も右側にある葉である。次に、根の位置に挿入されたノードがもつ値と、その子のノードもつ値を比較し、順序が逆であれば、位置を交換する。ただし、ノードが子を 2 つもつ場合は、一方より小さい値をもつ子を比較の対象とする。この操作を再帰的に繰り返し、交換が起きない場合か、葉と交換した場合に、ヒープの再構成は終了する。

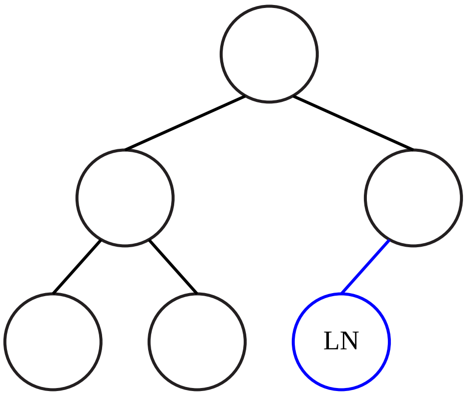

図 3　LN の例。

2　配列によるヒープの実現

教科書にあげられているヒープの実装は、たいてい配列によるものである。配列を用いる場合、ヒープは次のような`Heap`型で表現できる。

```
typedef struct {
    int n_nodes;
    Cell node[MAX_NODE_NUM];
} Heap;
```

ここで、ノードに格納されるデータ型を`Cell`型としておく。例えば、文字列のソートを扱う場合ならば、`Cell`型は次のように定義されるだろう。

```
typedef char * Cell;
```

また、`MAX_NODE_NUM`は十分に大きい数であるとする。

Heap 型へのポインタ h を用いると、そのノードの個数は h->n_nodes であり、IP, LN はそれぞれ、h->node[h->n_nodes], h->node[h->n_nodes - 1] である。配列を用いる場合に特徴的なことであるが、h->node[n] の親, 左の子, 右の子はそれぞれ、h->node[$(n-1)/2$], h->node[$2n+1$], h->node[$2n+2$] である、という関係が使用できる。これらの関係をまとめると、**図 4**のようになる。

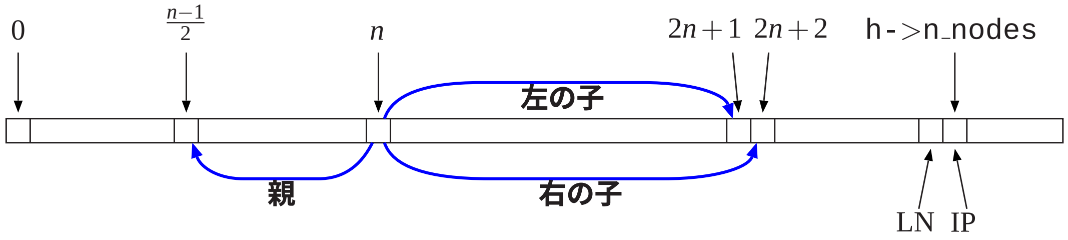

図 4　配列で表現されたヒープ。

2.1 Insert

たいていの教科書にも載っているはずであるが、ここでも Insert を次のように実装してみる。

```
void
Insert(Heap *h, Cell c)
{
    int n, m;

    h->node[h->n_nodes] = c; /* IP への挿入 */
    for (n = h->n_nodes; (m = Parent(n)) < n
             && CellCmp(h->node[n], h->node[m]) < 0; n = m) {
        c = h->node[n];
        h->node[n] = h->node[m];
        h->node[m] = c;
    }
    ++(h->n_nodes);
}
```

ただし、CellCmp は Cell 型の大小を比較する関数で、引数と返り値の関係は、strcmp と同様の関係とする。また、Parent は次のように定義できる。

```
#define Parent(n) (((n) - 1) / 2)
```

2.2 TakeMin

同様に、TakeMin を次のように実装してみる。

```
Cell
TakeMin(Heap *h)
{
    int n, m;
    Cell r = h->node[0], c;

    h->node[0] = h->node[h->n_nodes - 1]; /* LN を根に代入 */
    for (n = 0; (m = MinChild(h, n)) > 0
```

```
            && CellCmp(h->node[n], h->node[m]) > 0; n = m) {
        c = h->node[n];
        h->node[n] = h->node[m];
        h->node[m] = c;
    }
    --(h->n_nodes);
    return (r);
}
```

ここで、関数MinChildは、指定された番号のノードが子を持つ場合、小さい値をもつ方の子の番号を返し、子をもたない場合は0を返すものとする。MinChildは次のように定義できる。

```
#define Left(n) (2 * (n) + 1)
#define Right(n) (2 * (n) + 2)

int
MinChild(Heap *h, int n)
{
    if (h->n_nodes <= Left(n))
        return (0);
    else if (h->n_nodes == Right(n))
        return (Left(n));
    else if (CellCmp(h->node[Left(n)], h->node[Right(n)]) < 0)
        return (Left(n));
    return (Right(n));
}
```

3 配列のリストによるヒープの実現

ここまではよくある話であった。しかし、配列によるヒープの実現では、いくつか問題がある。例えば、

- ☞ MAX_NODE_NUM より大きなノードを含むヒープを扱えないので、限界を超えたときに配列を再確保するか、エラー処理を用意する必要がある。
- ☞ メモリを有効に利用していない。
- ☞ アルゴリズムやデータ構造の教科書にしては、親、左の子、右の子を表すために、算術式を用いた奇妙な（しかし都合のよい）関係を利用していて、気持ちが悪い（✌）。

などがあげられる。

そこで、配列のリストを用いてヒープを表現し、改善できないか試してみよう。**図 5**のように、ヒープのツリーを構成するノードのうち、同じ高さ k のノードの列を1つの配列として表し、それを双方向リストとして表現してみる。根では $k = 0$ で、previous はNULLを指す。また、最も高いノードの列では next はNULL 指す。

図 6のように、高さ k の左から i（i は0から数える）番目のノードの親, 左の子, 右の子はそれぞれ、高さ $k-1$ の左から $i/2$ 番目, 高さ $k+1$ の左から $2i$ 番目, 高さ $k+1$ の左から $2i+1$ 番目のノードという関係にある。

ヒープの高さが n であり、高さ n のノードの列（next がNULL を指す）にノードが m 個ある場合、そのヒープに含まれるノードの総数は $2^0 + 2^1 + ... + 2^{n-1} + m = 2^n - 1 + m$ である。

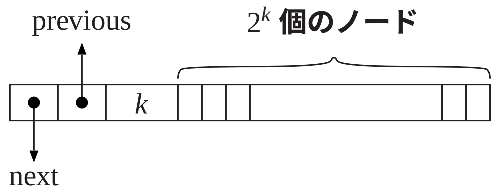

図 5　同じ高さをもつノード。

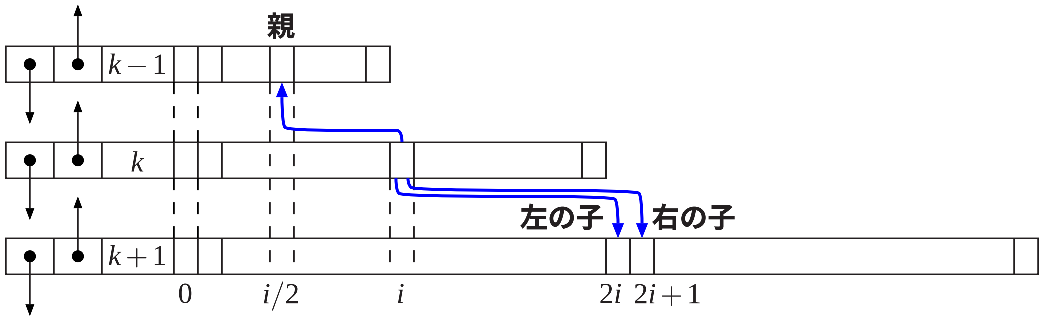

図 6　ノードの親子関係。

同じ高さをもつノードの列は次のような ADT で表現できる。

```
typedef struct SameHeightNodes {
    struct SameHeightNodes *next;
    struct SameHeightNodes *prev;
    int height;
    Cell *node; /* (1 << height) の配列を割り当てる */
} SameHeightNodes;
```

これらを用いると、ヒープは次のような ADT で表現できる。

```
typedef struct Heap {
    SameHeightNodes *root;
    SameHeightNodes *last;
    int n_nodes; /* last に含まれるノードの数 */
} Heap;
```

3.1　Insert

ノードの配列のリストで表現されたヒープに対する Insert を、次のように実装してみる。

```
#define Parent(n) ((n) / 2)

void
Insert(Heap *h, Cell c)
{
    int n, m;
    SameHeightNodes *d, *e;
```

```
    if (h->n_nodes >= (1 << h->last->height)) {
        e = CreateSameHeightNodes(h->last->height + 1);
        e->prev = h->last;
        e->next = NULL;
        h->last->next = e;
        h->last = e;
        h->n_nodes = 0;
    }
    h->last->node[h->n_nodes] = c; /* IP への挿入 */
    for (n = h->n_nodes, d = h->last; (e = d->prev) != NULL
             && ((m = Parent(n)), CellCmp(d->node[n], e->node[m]) < 0);
             n = m, d = e) {
        c = d->node[n];
        d->node[n] = e->node[m];
        e->node[m] = c;
    }
    ++(h->n_nodes);
}
```

ここで関数`CreateSameHeightNodes()`は次のようなものとする。

```
static SameHeightNodes *
CreateSameHeightNodes(int height)
{
    int n_nodes = (1 << height);
    SameHeightNodes *s;

    if ((s = (SameHeightNodes *)malloc(sizeof(SameHeightNodes))) == NULL)
        goto no_same_height_nodes;
    if ((s->node = (Cell *)malloc(sizeof(Cell) * n_nodes)) == NULL)
        goto no_node;
    s->next = NULL;
    s->prev = NULL;
    s->height = height;
    return (s);

no_node:
    free(s);
no_same_height_nodes:
    return (NULL);
}
```

3.2 TakeMin

同様に、TakeMin を次のように実装してみる。

```
Cell
TakeMin(Heap *h)
{
    int n, m;
    Cell r = h->root->node[0], c;
    SameHeightNodes *d, *e;

    h->root->node[0] = h->last->node[h->n_nodes - 1]; /* LN を根に代入 */
    for (n = 0, d = h->root; (m = MinChild(h, d, n)) >= 0
             && ((e = d->next), CellCmp(d->node[n], e->node[m]) > 0);
             n = m, d = e) {
        c = d->node[n];
```

```
        d->node[n] = e->node[m];
        e->node[m] = c;
    }
    if (--(h->n_nodes) <= 0 && h->last->prev != NULL) {
        e = h->last->prev;
        free(h->last);
        h->last = e;
        h->last->next = NULL;
        h->n_nodes = (1 << h->last->height);
    }
    return (r);
}
```

ここで、関数`MinChild`は、指定された番号のノードが子を持つ場合、小さい値をもつ方の子の番号を返し、子をもたない場合は -1 を返すものとする。`MinChild`は次のように定義できる。

```
#define Left(n) (2 * (n))
#define Right(n) (2 * (n) + 1)

int
MinChild(Heap *h, SameHeightNodes *d, int n)
{
    int total = (1 << h->last->height) - 1 + h->n_nodes;

    if ((d = d->next) == NULL || total <= (1 << d->height) - 1 + Left(n))
       return (-1);
    else if (total == (1 << d->height) - 1 + Right(n))
        return (Left(n));
    else if (CellCmp(d->node[Left(n)], d->node[Right(n)]) < 0)
        return (Left(n));
    return (Right(n));
}
```

4　配列を用いないヒープの実現

このようなヒープの実現により、ある程度問題は解決されたようにみえるが、やはり次のような問題は残されたままである。

☞ 最悪の場合、約半分のメモリは使用されていない。

☞ あいかわらず親、左の子、右の子を表すために、奇妙な関係を利用している。

そこで、ポインタだけを使用して、Insert、TakeMin がともに $O(\log n)$ であるようなヒープを実現してみよう。

そのためには、ノードが**図 7**のように表現される、ツリーと双方向リストを組み合わせた構造の ADT を用いる。この場合、あるノードの親、左の子、右の子については、$O(1)$ で求まる。例として、ノートが 2 個の場合を**図 8**に示した。

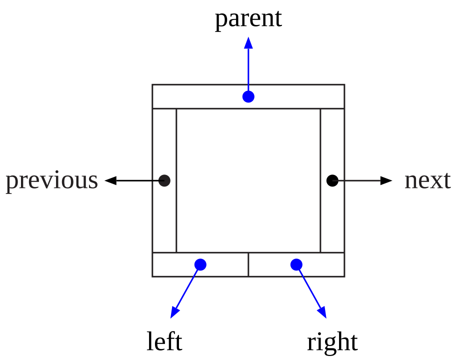

図 7　ツリー構造でもあり双方向リストでもあるノード。

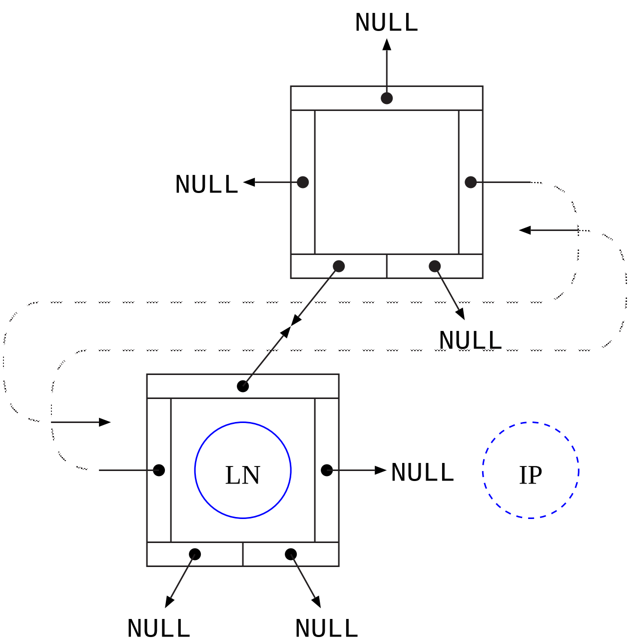

図 8　ノード 2 個の場合。

Insert, TakeMin については、LN の親を調べることによって、新しい LN の位置を求める 3 つの場合が存在する。それらは、ノードが 1 つしかない（すなわち根しかない）場合、LN が左の子の場合（**図 9**)、LN が右の子の場合（**図 10**）である。

ノードが 1 つしかない場合を除き、LN の直接の親は必ず存在する。LN の追加や削除に伴い、新しい LN の親のポインタを書き換える必要があるが、LN の親を経由することで、その処理は $O(1)$ である。

このようにヒープを実現した場合のリンクの様子は、**図 11**のようになる。

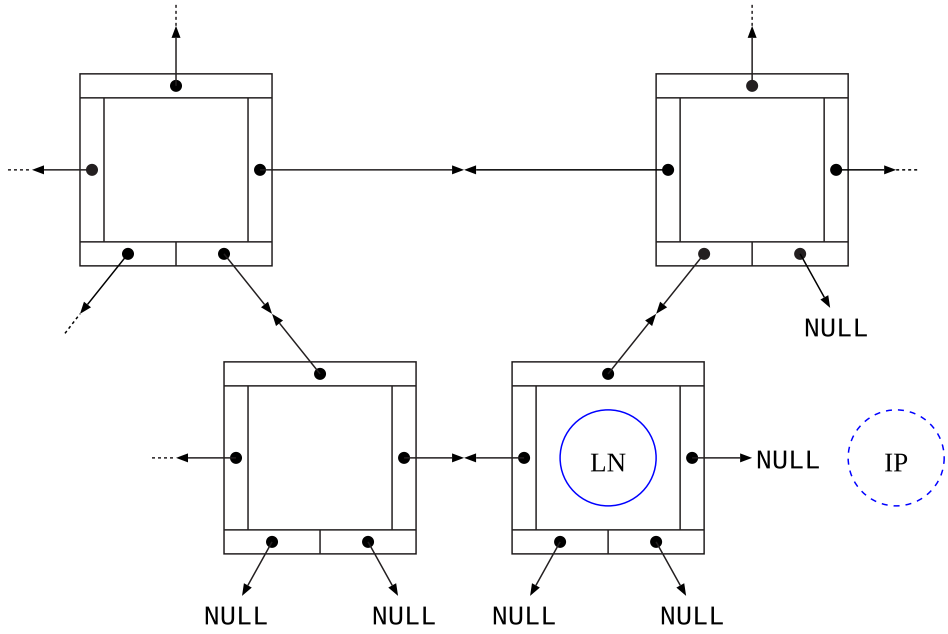

図 9　LN が左の子の場合。

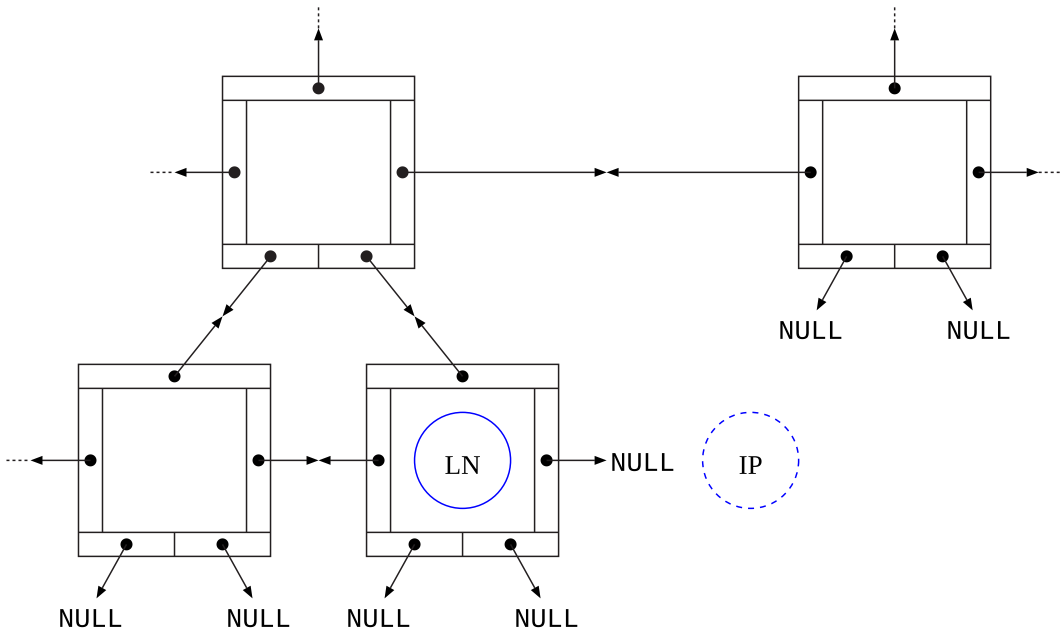

図 10　LN が右の子の場合。

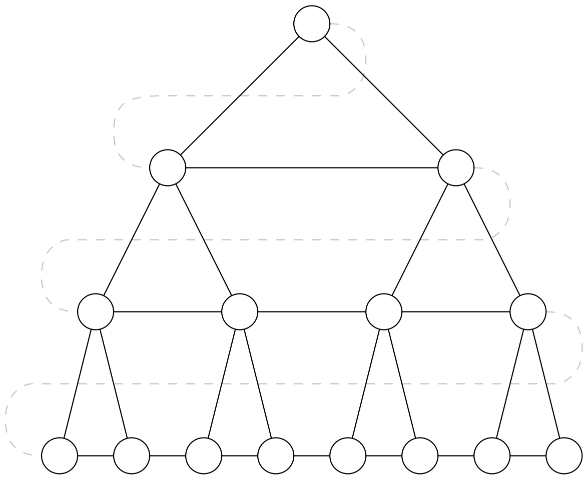

図 11　ツリーと双方向リストの混合 ADT のリンクの様子。

5　ヒープの正体

しかし、前節のようにヒープを実現した場合、ツリーの深さが増えたり減ったりするときに、その構造が正しく関係を保つのか、直感的に理解できないかもしれない。

図 11の表現をよく考えると、ヒープの正体は**図 12**のような構造であることがわかる。このため、Insert や TakeMin といった操作は、ツリーの深さとは無関係に実行でき、その親の状態のみに依存することが保証できる。

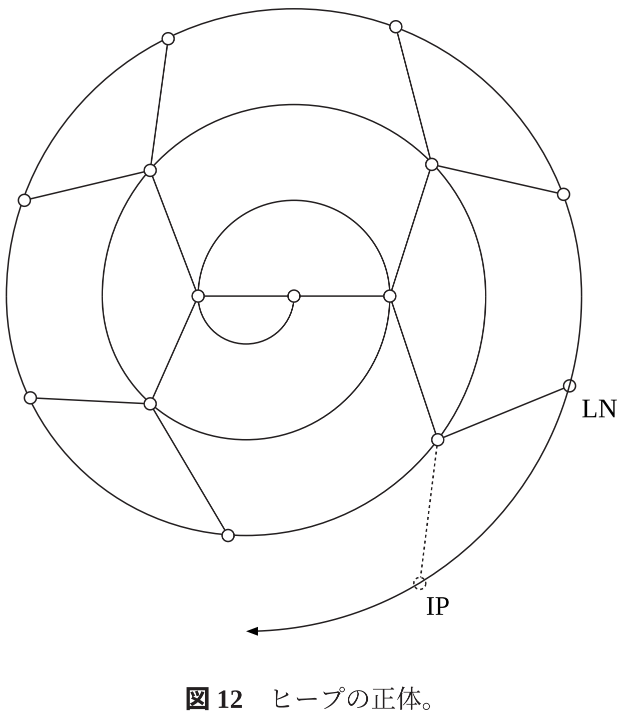

図 12　ヒープの正体。