木 T の葉を順に除去する手続き用のデータ構造

木の根を示すポインタ T が与えられたとき, 木の葉を 1 個づつ取り除くことにより, 根だけにする以下のアルゴリズムに必要なデータ構造を考える.

```
void   Shrinking_Tree ( T )
   入力:   T  :   TREE (値呼び)
   出力:   T  :   TREE
        /* 木 T の葉を順に除去する手続き */
NODE   w ;
SET   L ;
{
0゜      木 T の葉の集合 L を作成する ;
1゜      while ( 木 T が根以外の点を含む ) {
1-1゜            集合 L の中の一つの葉を v とし, Delete( v, L ) を実行して,
                     L から v を取り除く ;
                     /* この操作によって, 葉 v が指定される */
1-2゜            v に対してある操作を行う ;
1-3゜            葉 v を木 T から取り除く ;
1-4゜            if ( 木 T に新たな葉 w が生まれた )        Insert( w, L ) ;
                     /* 本では, この w が v になっている. 間違いである */
        }
} /* Shrinking_Tree */
```

この手続きにおける木 T および葉の集合 L に対して実行すべき操作は, 下記のように纏められる.

1゜　木 T が根以外の点を含むか否かを判定する
1-3゜　指定された葉 v を木 T から取り除く
1-4゜　葉 v の除去によって, 木 T に新たな葉 w が生まれたか否かを判定する

0゜　木 T を探索して, 葉の集合 L を作成する
1-1゜　集合 L の中から一つの葉 v を取り除く
1-4゜　指定された葉 w を集合 L に入れる

まず, 図 3.2 に示されたデータ構造を用いた場合, これらの操作がどのような計算量で実行できるかを考えよう. 図 3.2 のデータ構造は下記のように書ける.

```
typedef     CELL↑          NODE ;        /* NODE は CELL 型データへのポインタ */

typedef  struct {
                    elementtype      element ;
                    NODE             parent, lm_child, r_sibling ;
                                          /* これらは CELL 型データへのポインタ */
         } CELL ;

typedef     NODE           TREE ;        /* TREE は CELL 型データへのポインタ */
```

このデータ構造において, ある点 v が木 T において葉になっているか否かは, $(\uparrow v)$.*lm_child* が **NULL** か否かによって判定できる. 従って, 木 T が根以外の点を含むか否か (**1°**) は, $(\uparrow T)$.*lm_child* が **NULL** か否かを調べればよい. また, 指定された葉 v の除去によって, 影響を受ける点は, v の親 w だけであるから, 木 T に新たな葉 w が生まれたか否か (**1-4°**) は, v の除去により v の親 w の *lm_child* 欄 $(\uparrow w)$.*lm_child* が **NULL** か否かで分かる. 従って, これらの操作は $O(1)$ でできる.

しかし, このデータ構造では, 指定された葉 v を木 T から取り除く操作 (**1-3°**) は, v がその親 w の長男 (*lm_child*) とは限らないので, w の全ての子を調べながら v が指すセル (CELL) を取り除かねばならない. 従って, $O(1)$ ではできない.

そこで, この操作 **1-3°** を効率的に行うことを考える.

指定された葉 v を取り除く際, v の親 w の子を全て調べなければならないのは, w の子が, *r_sibling* 欄が次のセルを示すポインタになった連結リストで覚えられており, その連結リスト内において, v が指すセルがどこにあるか分からないからである. すなわち v が指すセルの直前のセルを調べるため, w の全ての子を調べなければならないのである.

従って, 木 T が図 3.2 に示されたデータ構造で与えられる場合, すなわち木 T の各点の構造を上に示した CELL 型にしている限り, 操作 **1-3°** をより効率的に行うことはできない.

そこで, v が指すセルの直前のセルを求めなくても $O(1)$ で v が指すセルを除去できるようにす

るため, 点 *w* の子のﾘｽﾄを連結ﾘｽﾄから双方向ﾘｽﾄに変える. すなわち, 各点に対して, その点の直ぐ上の兄を示す *l_sibling* 欄を設ける.

そうすると, 木 *T* のデータ構造は次のようになる. なお, この双方向ﾘｽﾄは巡回ﾘｽﾄとしてもよいが, ここでは巡回ﾘｽﾄとはしないことにしよう. このことは下記の記述では表せないので, ｺﾒﾝﾄとして残す.

```
typedef     N_CELL↑      NODE  ;    /*  NODE  は  N_CELL  型データへのポインタ  */

typedef  struct {
                    elementtype       element  ;
                    NODE              parent, lm_child, r_sibling, l_sibling  ;
          } N_CELL ;
     /*  parent  および  lm_child  はそれぞれ親および長男へのポインタ  */
     /*  r_sibling  および  l_sibling  はそれぞれ次弟および直ぐ上の兄へのポインタ  */
     /*  長男の  l_sibling  および末弟の  r_sibling  はそれぞれ  NULL  としておく  */

typedef     NODE         TREE  ;    /*  TREE  は  N_CELL  型データへのポインタ  */
```

次に, 集合 *L* のデータ構造を決定しなければならないが, 操作 **1-1˚** において葉 *v* がどのように指定されるのか明確に定められていない. 葉 *v* が何らかの方法で指定されるものとしてもよいが, ここでは, これは自由に決めることができるものとし, 集合 *L* に入れられた順に葉 *v* が取り出されるものとすると, 集合 *L* は QUEUE になる.

こうすると, 下記の操作それぞれは, **0˚** は木 *T* を深さ優先探索しながら, 葉 *v* を見出す度に, QUEUE *L* に *Enqueue* すればよく, **1-1˚** は *Enqueue*, **1-4˚** は *Dequeue* すればよい.

0˚ 木 *T* を探索して, 葉の集合 *L* を作成する
1-1˚ 集合 *L* の中から一つの葉 *v* を取り除く
1-4˚ 指定された葉 *w* を集合 *L* に入れる

以上の考察の基に, 手続き *Shrinking_Tree* (*T*) は下記のように詳細化できる.

なお, 木を深さ優先探索しながら, 集合 *L* を初期化する手続き *DFS* を先に示す. これを呼ぶため, 操作 **0˚** は次のようにする. ここで, *Makenull*(*L*) は QUEUE *L* の初期化である.

0-1˚ *Makenull*(*L*) ;
0-2˚ *DFS*(*T, L*) ;

```
void   DFS ( v, L )
   入力:   v  :   NODE (値呼び)
           L  :   QUEUE (名前呼び)
   出力:   v  :   点 v を根とする部分木
           L  :   QUEUE (名前呼び)
        /* v を根とする部分木に対する深さ優先探索 */
NODE   w ;
{
        w := (↑v).lm_child ;
        if ( w = NULL ) {
                Enqueue( v, L ) ;
                return ;
        }
        else {
                while ( w≠NULL ) {
                        DFS( w, L ) ;
                        w := (↑w).r_sibling ;
        }       }
} /* DFS */
```

```
void   Shrinking_Tree ( T )
   入力:   T  :   TREE (値呼び)
   出力:   T  :   TREE
        /* 木 T の葉を順に除去する手続き */
NODE   w ;
QUEUE   L ;
{
0-1゜     Makenull( L ) ;
0-2゜     DFS( T, L ) ;
1゜       while ( (↑T).lm_child≠NULL ) {
1-1゜             v := Dequeue( L ) ;
1-2゜             v に対してある操作を行う ;
1-3゜             w := (↑v).parent ;
                  /* 葉 v を親 w の子のリストから取り除く */
                  if ( (↑w).lm_child = v ) {
(1-4゜)                     if ( (↑v).r_sibling = NULL ) {
                                  Enqueue( w, L ) ;
                                  (↑w).lm_child := NULL ;
                            }
                            else {
                                  (↑w).lm_child := (↑v).r_sibling ;
                                  (↑(↑w).lm_child).l_sibling := NULL ;
                  }         }
                  else {
                            (↑(↑v).l_sibling).r_sibling := (↑v).r_sibling ;
                            (↑(↑v).r_sibling).l_sibling := (↑v).l_sibling ;
                  }
                  free( v ) ;
        }
} /* Shrinking_Tree */
```