カーソルによるﾘｽﾄの実現

ここでは, ヘッダセルを用いず, 最後のセルの位置を示す *last* も用いない場合の例を示す.

ヘッダセルや *last* を用いれば, プログラムはもっと簡単になる. プログラムの練習のため, このようにしているので, 是非, C 言語でプログラムを作り, 実際に走らせてみるとよい.
教科書でも, ここでも,

- 先頭の位置は, 0 で表し,
- ﾘｽﾄ *L* が空ﾘｽﾄであることも, *L* = 0 で表すことになっている.

```
typedef  struct {
                elementtype     elements[1..maxlength] ;
                int             next ;
         }  CELL ;

typedef    int    position ;
typedef    int    LIST ;

CELL    CellSpace[1..maxlength] ;     /*  配列 CellSpace の各要素は CELL 型のデータ  */
LIST    L, available ;                /*  CellSpace, available は大域的変数  */

/*  以下では, カーソルが 0 であることが, ポインタが NULL であることに相当する  */
```

教科書にも示した下記のセル移動関数を用いる.
これは, *p* が指すセルを *q* が指すセルの直前に移動し, *p* は *p* が指していたセルの次のセルを, *q* は *p* が指していたセルを指すように変える関数である.

```
boolean  Displace(  p, q  )
         入力:    p, q   :        position (名前呼び)      /*  C 言語では p, q へのポインタを渡す  */
         出力:    p, q   :        position
                  Displace :       boolean
position    tmp ;                /**  position データの退避場所  */
{
         if (  p = 0  ) {                                  /*  移動すべきセルが無い  */
                  エラー表示の処理などを実行 ;
                  return    false ;
         }
         else {
                  tmp  :=  q ;
                  q  :=  p ;
                  p  :=  CellSpace[p].next ;
                  CellSpace[q].next  :=  tmp ;
                  return    true ;
         }
} /* Displace */
```

上記の関数 *Displace*(*p, q*) は, $q = 0$, すなわち空ﾘｽﾄへの移動も正しく実行されることを確認しておくと良い.

ただし, *Displace*(*p, q*) では, カーソル *p, q* の検査, すなわち *p, q* が正当なセルを指しているかの検査はしていない. 従って, *Displace* の呼び出し側で, カーソルの正当性を保証しなければならない.

その意味で, 教科書の *Insert*(*x, p, L*) および *Locate*(*x, L*) は不十分とも言える.

以下のプログラムでは, $0 \leqq p \leqq maxlength$ の検査をし, *p* の正当性を確認している.

```
position  Tail( L )
        入力 :  L  :     LIST (値呼び)
        出力 :  Tail  :  position (整数)
position   q ;                  /*  セルを順に調べるためのカウンタ (カーソル). 局所的変数  */
{
        q  :=  L ;
        while ( q  ≠  0 ) {
                if ( 1 ≦ q ≦ maxlength )      q  :=  CellSpace[q].next ;
                else        エラー処理関数へ飛ぶ ;
        }
        return  q ;             /*  このプログラムでは, 空ﾘｽﾄ L の Tail( L ) は 0 としている  */
} /*  Tail  */
```

```
position  Makenull( L )
        入力 :  L  :     LIST (名前呼び)
        出力 :  Tail  :  position (整数)
boolean   dummy ;               /*  関数 Displace の戻り値を入れるための変数 (値は使わない)  */
{
        if ( 0 ≦ available ≦ maxlength ) {
                while ( L  ≠  0 ) {
                        if ( 1 ≦ L ≦ maxlength )      dummy  :=  Displace( L, available ) ;
                        else        エラー処理関数へ飛ぶ ;
                }
                return  L ;
        }
        else        エラー処理関数へ飛ぶ ;
} /*  Makenull  */
```

```
position  First( L )
        入力 :  L  :     LIST (値呼び)
        出力 :  First  :  position (整数)
{
        if ( 0 ≦ L ≦ maxlength )      return  0 ;
        else        エラー処理関数へ飛ぶ ;
} /*  First  */
```

```
void  Insert( x, p, L )
        入力 :  x  :      elementtype
                p  :      position (値呼び)
                L  :      LIST (名前呼び)
        出力 :  L  :      LIST
                CellSpace   :      各要素が CELL 型の配列
                available   :      LIST
{
        if ( 0 ≦ L ≦ maxlength  and  0 ≦ available ≦ maxlength ) {
                if ( p = 0 ) {                  /* L の先頭のセルに x を入れる */
                        if ( Displace( available, L ) ) {
                                CellSpace[L].element, := x ;
                                return ;
                }       }
                else if ( 0 ≦ p ≦ maxlength ) {
                                        /* p が指すセルの直後のセルに x を入れる */
                        if ( Displace( available, CellSpace[p].next ) ) {
                                CellSpace[CellSpace[p].next].element, := x ;
                                return ;
        }       }       }
        空セルがないか, p が存在しない位置か, L が正しくないか, あるいは
                リスト available が壊れているので, エラー処理関数へ飛ぶ ;
} /* Insert */
```

```
void  Delete( p, L )
        入力 :  p  :      position (値呼び)
                L  :      LIST (名前呼び)
        出力 :  L  :      LIST
                CellSpace   :      各要素が CELL 型の配列
                available   :      LIST
{
        if ( 0 ≦ L ≦ maxlength  and  0 ≦ available ≦ maxlength ) {
                if ( p = 0 ) {                  /* L の先頭のセルを取り除く*/
                        if ( Displace( L, available ) )      return ;
                }
                else if ( 1 ≦ p ≦ maxlength ) {
                                        /* p が指すセルの直後のセルを取り除く*/
                        if ( Displace( CellSpace[p].next, available ) )      return ;
        }       }
        L が正しくないか, 空リストか, p が不正なカーソルか, あるいは
                リスト available が壊れているので, エラー処理関数へ飛ぶ ;
} /* Delete */
```

```
position  Next( p, L )
        入力 :  p  :     position (値呼び)
                L  :     LIST (値呼び)
        出力 :  Next  :  position (整数)
{
        if ( 0 ≦ L ≦ maxlength  and  0 ≦ p ≦ maxlength ) {
                if ( p = 0 )        return   L ;
                else                return   CellSpace[p].next ;
        }
        else    L が正しくないか, p が不正なカーソルなので, エラー処理関数へ飛ぶ ;
} /* Next */
```

```
position  Previous( p, L )
        入力 :  p  :     position (値呼び)
                L  :     LIST (値呼び)
        出力 :  Previous  :  position (整数)
position  q ;                /* セルを順に調べるためのカウンタ (カーソル). 局所的変数 */
{
        if ( 0 ≦ L ≦ maxlength  and  1 ≦ p ≦ maxlength ) {
                q := L ;
                while ( q ≠ 0 ) {
                        if ( 0 ≦ CellSpace[q].next ≦ maxlength ) {
                                if ( CellSpace[q].next = p )     return q ;
                                else                             q := CellSpace[q].next ;
                        }
                        else    不正なカーソルを見つけたので, エラー処理関数へ飛ぶ ;
        }       }
        L が正しくないか, 空リストか, p が見つからないか, あるいは
                p の直前のセルは無いので, エラー処理関数へ飛ぶ ;
} /* Previous */
```

```
elementtype  Retrieve( p, L )
        入力 :  p  :     position (値呼び)
                L  :     LIST (値呼び)
        出力 :  Retrieve  :  elementtype
/* ここでは, elementtype の要素を関数の戻し値とできるものとしている */
{
        if ( 1 ≦ L ≦ maxlength  and  0 ≦ p ≦ maxlength ) {
                if ( p = 0 )        return   CellSpace[L].element ;
                else if ( 1 ≦ CellSpace[p].next ≦ maxlength )
                        return   CellSpace[CellSpace[p].next].element ;
        }
        L には, 位置 p の要素が見つからないので, エラー処理関数へ飛ぶ ;
} /* Retrieve */
```

```
position  Locate( x, L )
        入力 :  x  :     elementtype
                L  :     LIST (値呼び)
        出力 :  Locate  :   position (整数)
/* ここでは, elementtype の二つの要素 x, y が同一か否かを判定する関数 Same( x, y ) */
/* が存在するものとし, それを用いている */
position  p ;                   /* 要素を順に調べるためのカウンタ (カーソル). 局所的変数 */
{
        if ( 0 ≦ L ≦ maxlength ) {
                if ( L = 0 )        return   −1 ;      /* リスト L は空リストなので, x はない */
                else  if ( Same( CellSpace[L].element, x ) )    return  0 ;
                                                        /* x はリスト L の先頭にある */
                else {
                        p := L ;
                        while ( 1 ≦ CellSpace[p].next ≦ maxlength ) {
                                if ( Same( x, Retrieve( p, L ) = false )
                                        p := CellSpace[p].next ;
                                else        return p ;
                        }
                        if ( CellSpace[p].next = 0 )      return p ;
        }       }
        L が正しくないか, 不正なカーソルが見つかったので, エラー処理関数へ飛ぶ ;
} /* Locate */
```

```
void  PrintList( L )
        入力 :  L  :     LIST (名前呼び)
        出力 :  L の要素を順に印刷したもの
/* ここでは, elementtype の要素 x を出力する関数 PrintElement( x ) が存在する */
/* ものとし, それを用いている */
position  p ;                   /* 要素を順に調べるためのカウンタ (カーソル). 局所的変数 */
{
        if ( 1 ≦ L ≦ maxlength ) {
                p := L ;
                while ( 1 ≦ CellSpace[p].next ≦ maxlength ) {
                        PrintElement ( Retrieve( p, L ) ) ;
                        p := CellSpace[p].next ;
                }
                if ( CellSpace[p].next = 0 )      return ;
        }
        L が正しくないか, 不正なカーソルが見つかったので, エラー処理関数へ飛ぶ ;
} /* PrintList */
```