クローズドハッシュによる *Member*(*x, A*) および *Delete*(*x, A*) のプログラム

そろそろ C 言語を忘れた頃であろうから, 以下では, C 言語によるプログラムを作ってみよう.

まず, バケツが empty であるか deleted であるかを示すフラッグ (識別欄) を各バケツに付け, バケツのデータ型を次のような構造体としよう.

```
typedef  struct  {
                elementtype     element ;
                short int       flag ;
        }  Bucket  ;
```

ここで, *element* は集合の要素を入れる欄であり, *flag* は 0, 1, あるいは –1 のいずれかの値をとる欄で, *flag* = 0 であれば, バケツに要素がまだ入れられたことがない (empty) ことを, *flag* = 1 であれば, バケツに要素が入っていることを, *flag* = –1 であれば, バケツに要素が入れられた後取り除かれ, 現在入っていない (deleted) ことを表す. 以下では次のように定義してあるものとする.

```
#define       empty   0
#define       deleted –1
```

このような Bucket 型のバケツを *B* 個並べてハッシュ表を作り, ハッシュ表 (バケツの配列: *HashTable*) とその長さ *B* の組をクローズドハッシュのためのデータ構造としよう.

```
typedef  struct  {
                Bucket      HashTable[maxlength]  ;
                int         B  ;                    /*  B  :  バケツの個数  */
                                                    /*  1≦B≦maxlength  */
        }  ClosedHash  ;
```

そうすると, *Insert, Member, Delete* などを実行する集合 *A* はこのような構造体として定義できる.

```
ClosedHash      A  ;
```

集合 *A* をこのような構造体で定義したので, *Member*(*x, A*) や *Delete*(*x, A*) のプログラムにおいて, パラメタ *A* は配列ではないため, C 言語では *A* へのポインタを渡すことになる. 従って, これらを呼び出す側では,

Member(*x, &A*) や *Delete*(*x, &A*)

とする.

このとき, 集合 *A* を空にする *Makenull*(*A*) のプログラムは次のように書ける.

```
#define   empty   0
void   Makenull( ClosedHash＊ ptr_A )
        /* 集合 A を空にする */
{
        int   b ;
        for ( b = 0 ;  b < (＊ptr_A).B ;  b++ )          /* 0≦b < (＊ptr_A).B なる各 b */
                ((＊ptr_A).HashTable[b]).flag = empty ;
                /* 各バケツの flag 欄に空である印(empty)を入れる */
} /* Makenull */
```

このようにせず, 集合 *A* を

```
Bucket          A[B] ;
```

と宣言し, その長さ *B* を定数としておくこともできる. この場合には, *A* をそのままパラメタとして書いても C 言語ではポインタが受け渡されるので, *Makenull*(*A*) のプログラムは次のように書くことができる.

```
#define   empty   0
void   Makenull( Bucket A[ ] )
        /* 集合 A を空にする */
{
        int   b ;
        for ( b = 0 ;  b < B ;  b++ )          A[b].flag = empty ;
                /* 各バケツの flag 欄に空である印(empty)を入れる */
} /* Makenull */
```

次に, ハッシュ関数 $h(x)$ および再ハッシュ関数 $h_i(x)$ を計算するプログラムを考えねばならないが, これは elementtype の型によって変わるものであるから, 次のような形で与えられるものとしよう.

```
int   Hashing( elementtype  x,    int  i, B  )
        /*  入力 ： x ： elementtype の要素  */
        /*       ： i ： 0≦i≦B–1 なる整数  */
        /*            この関数が i 回目の再ハッシュ関数であることを示す  */
        /*            i = 0 であれば, この関数はハッシュ関数  */
        /*       ： B ： バケツの個数を表す整数  */
        /*  出力 ： Hashing ： 0≦i≦B–1 なる整数（ハッシュ値 h_i( x ) ）  */
{
} /*  Hashing  */
```

また, 2 つの要素 x, y が同じ要素か否かを判定する関数 *Same* (x, y) が存在するものとする.

```
boolean   Same( elementtype  x,  y  )
        /*  入力 ： x, y ： elementtype の要素  */
        /*  出力 ： Same ： x と y が同じ要素であれば true, さもなくば false  */
{
} /*  Same  */
```

さらに,

```
#define   false   0  ;
#define   true    1  ;
```

と定義してあるものとする.

そうすると, 関数 *Member*(x, A) および手続き *Delete*(x, A) は次のように書ける.

```
boolean  Member( elementtype x,  ClosedHash* ptr_A )
    /* 要素 x が集合 A に含まれるか否かを判定する関数 */
    /* x ∈ A であれば true, さもなくば false を返す */
{
        int  i, b ;
        for ( i = 0 ; i < (*ptr_A).B ; i++ ) {   /* 0≦i<(*ptr_A).B なる各 i に対して */
                b = Hashing( x, i, (*ptr_A).B ) ;        /* b は x のハッシュ値 */
                if ( ((*ptr_A).HashTable[b]).flag == empty )
                        return false ;                   /* x は集合 A に含まれない */
                else if ( ((*ptr_A).HashTable[b]).flag != deleted  &&
                                        Same( x, ((*ptr_A).HashTable[b]).element ) )
                        /* flag が deleted でなく, かつ element が x, と同じならば */
                        return true ;                    /* x は集合 A に含まれる */
        }
        /* 必要であれば, ここにハッシュ表が満杯であることを警告する文を入れる */
        return false ;                   /* x は集合 A に含まれない */
} /* Member */
```

```
void  Delete( elementtype x,  ClosedHash* ptr_A )
    /* 要素 x を集合 A から取り除く手続き */
{
        int  i, b ;
        for ( i = 0 ; i < (*ptr_A).B ; i++ ) {   /* 0≦i<(*ptr_A).B なる各 i に対して */
                b = Hashing( x, r, (*ptr_A).B ) ;        /* b は x のハッシュ値 */
                if ( ((*ptr_A).HashTable[b]).flag == empty )
                        return ;                 /* x は A に含まれないので何もしない */
                else if ( ((*ptr_A).HashTable[b]).flag != deleted  &&
                                        Same( x, ((*ptr_A).HashTable[b]).element )
                        ((*ptr_A).HashTable[b]).flag = deleted ;
                                        /* x は A から取り除いたという印を入れる */
        }
} /* Delete */
```

ここまで読んだ懸命な諸君は, *Insert* もこのデータ構造に応じて作成(*flag* を正しくセット)しなければならないことが分かるであろう. そこで, 以下に, ここで定義したデータ構造の基, 教科書で示した *Insert* の操作を詳細化したものを示す. これを, 上に示した *Member* の操作を参考にして書き換えてみよ(文の個数を減らしてみよ).

```
#define  exist  1
void  Inseert( elementtype  x,   ClosedHash＊  ptr_A  )
    /*  要素 x を集合 A に入れる手続き  */
{
        int   i,  b  ,  b_deleted  ;
1°      b  =  Hashing( x, 0, (＊ptr_A).B )  ;              /*  b は x のハッシュ値  */
2°      b_deleted  =  –1  ;          /*  deleted の入ったバケツはまだ見つかっていない  */
3°      for (  i = 1 ;  i < (＊ptr_A).B ;  i++  )  {
3-1°            if (  ((＊ptr_A).HashTable[b]).flag  !=  empty    &&
                      Same(  x,  ((＊ptr_A).HashTable[b]).element  )      return  ;
                                  /*  x は集合 A に含まれるので, 何もしない  */
3-2°            else  if (  ((＊ptr_A).HashTable[b]).flag  =  empty  ) {
                      if (  b_deleted  = =  –1  ) {
                          /*  最初に見つかった deleted の入ったバケツである  */
                          ((＊ptr_A).HashTable[b]).element  =  x  ;
                          ((＊ptr_A).HashTable[b]).flag  =  exist  ;
                      }                          /*  x をバケツ b に入れた  */
                      else {
                          ((＊ptr_A).HashTable[b_deleted]).element  =  x  ;
                          ((＊ptr_A).HashTable[b_deleted]).flag  =  exist  ;
                      }                          /*  x をバケツ b_deleted に入れた  */
                      return  ;
                }
3-3°            else  if (  ((＊ptr_A).HashTable[b]).flag  = =  deleted    &&
                                                          b_deleted  =  –1  )
                      b_deleted  =  b  ;
3-4°            b  =  Hashing( x, r, (＊ptr_A).B )  ;       /*  b は x のハッシュ値  */
        }
4°      /*  ここにハッシュ表が満杯であることを警告する文を入れる  */
        return  ;                  /*  x を入れることができないので何もしない  */
} /*  Insert  */
```

判定の順序などを工夫すれば, 次のように文の個数を減らすことができる.

```
#define  exist  1
void  Inseert( elementtype x,  ClosedHash＊ ptr_A )
    /* 要素 x を集合 A に入れる手続き */
{
        int  i, b , b_deleted ;
        b_deleted = –1 ;            /* deleted の入ったバケツはまだ見つかっていない */
        for ( i = 0 ; i < (＊ptr_A).B ; i++ ) {
                b = Hashing( x, i, (＊ptr_A).B ) ;              /* b は x のハッシュ値 */
                if ( ((＊ptr_A).HashTable[b]).flag = empty ) {
                        if ( b_deleted = = –1 ) {
                                ((＊ptr_A).HashTable[b]).element = x ;
                                ((＊ptr_A).HashTable[b]).flag = exist ;
                        }                       /* x をバケツ b に入れた */
                        else {
                                ((＊ptr_A).HashTable[b_deleted]).element = x ;
                                ((＊ptr_A).HashTable[b_deleted]).flag = exist ;
                        }                       /* x をバケツ b_deleted に入れた */
                        return ;
                }
                else if ( ((＊ptr_A).HashTable[b]).flag = = deleted ) {
                        if ( b_deleted = –1 )           b_deleted = b ;
                }
                else {          /* バケツ b には要素が入っている */
                        if ( Same( x, ((＊ptr_A).HashTable[b]).element )    return ;
                                /* x は集合 A に含まれるので, 何もしない */
                }
        }
        /* ここにハッシュ表が満杯であることを警告する文を入れる */
        return ;                /* x を入れることができないので何もしない */
} /* Insert */
```