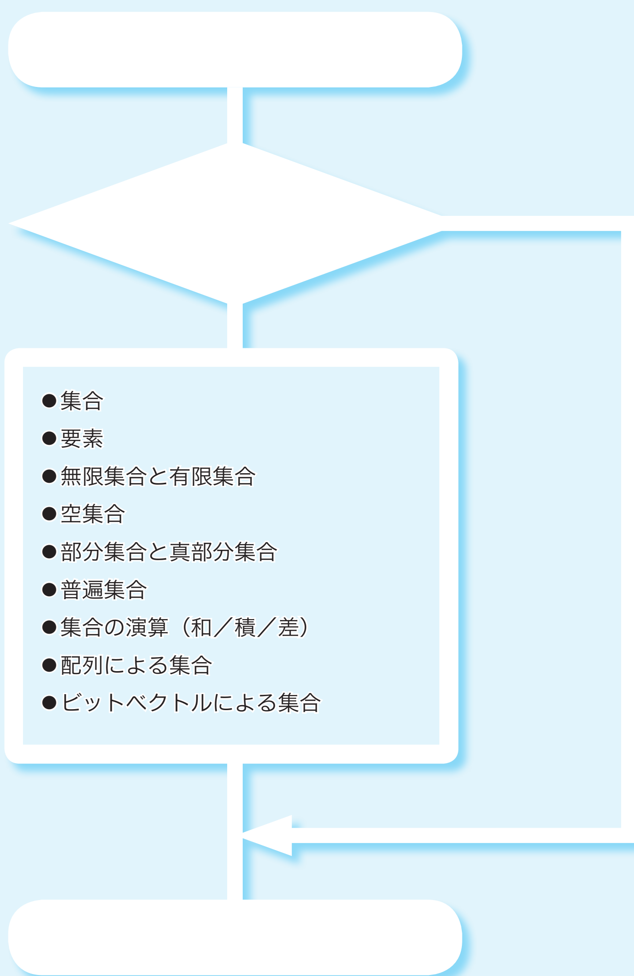
第７章

集　合



7-1 集合とは

《もの》の集まりを表すのが集合です。本節では、集合の基本を学習します。

集合と要素

集合（*set*）とは、**客観的に範囲が規定された《もの》の集まり**であり、その集合中の個々の《もの》が要素（*element*）です。

Fig.7-1 に『九州の県』の集合を示します。《福岡県》、《佐賀県》などが『九州の県』の集合の要素です。

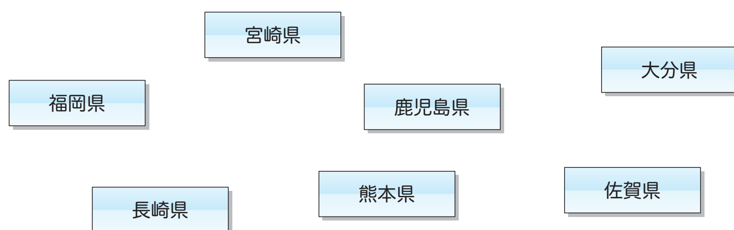


Fig.7-1　九州の県の集合

集合 X の要素が 1 と 5 であることは、以下のように表記します。

$$X = \{1, 5\}$$

ただし、集合内の要素には**順序がありません**ので、

$$X = \{5, 1\}$$

とも表記できます。

また、一般に、

$$N = \{1, 2, 3, 4, \dots\}$$ 自然数の集合

と自然数の集合を N で表し、

$$Z = \left\{ \begin{array}{l} 1,\ 2,\ 3,\ 4,\ \dots \\ 0 \\ -1,\ -2,\ -3,\ -4,\ \dots \end{array} \right\}$$ 整数の集合

と整数全体の集合を Z で表します。

▶　N は natural number（自然数）の頭文字、Z はドイツ語の Zahl（数）の頭文字に由来します。

集合の要素は、それ以上分解できない要素、すなわち**アトム**（*atom*）であっても、集合であっても構いません。

ただし、すべての要素は**互いに異なるものでなければなりません**。すなわち、{1, 5, 1}といった集合はあり得ません。

▶ 要素が互いに異ならない集合を**多重集合**と呼んで区別します。

a が集合 X の要素であることを、『a は X に**入っている**』、あるいは『a は X に**属する**』といいます。その表記が、

$a \in X$　または　$X \ni a$　　a は X に入っている

です。一方、b が集合 X の要素でなければ、

$b \notin X$　または　$X \not\ni b$　　b は X に入っていない

と書き表します。

＊

二つの集合 X と Y が同じ要素から構成されるとき、『**X と Y は等しい**』といい、

$X = Y$　または　$Y = X$　　X と Y は等しい

と表記します。一方、同じ要素で構成されない場合は、『**X と Y は等しくない**』といい、

$X \neq Y$　または　$Y \neq X$　　X と Y は等しくない

と表記します。

＊

整数の集合のように、要素数が無限大の集合は**無限集合**と呼ばれます。それに対して、要素数が有限の集合は**有限集合**です。

有限集合 X の要素数が n であるとき、

$|X| = n$　　X の要素数は n

と表します。ただし、X が無限集合であれば、

$|X| = \infty$　　X は無限集合

と書き表します。

さて、集合は《もの》の集まりであると最初に説明しました。日本語の「**集まり**」には複数のものを表すイメージがありますが、集合の要素数は 1 個であっても構いません。すなわち $|X| = 1$ である X も集合です。

さらに、$|X| = \emptyset$ である X も集合とみなされます。このような、要素が 1 個もない集合を**空集合**（*empty set*）と呼び、ϕ で表します。

部分集合と真部分集合

他の集合に含まれる集合は、部分集合あるいは真部分集合です。

部分集合

集合Aのすべての要素が集合Bの要素となっているとき、AはBの**部分集合**（*subset*）であり、『**AはBに含まれる**』といいます（**Fig.7-2**）。

この関係は $A \subset B$ あるいは $B \supset A$ と表します。

AとBが等しければ、互いに部分集合であり $A \subset B$ かつ $B \subset A$ です。

例　$A = \{1, 3\}$ で $B = \{1, 3, 5\}$ のとき $A \subset B$。

例　$A = \{1, 3, 5\}$ で $B = \{1, 3, 5\}$ のとき $A \subset B$ かつ $B \subset A$。

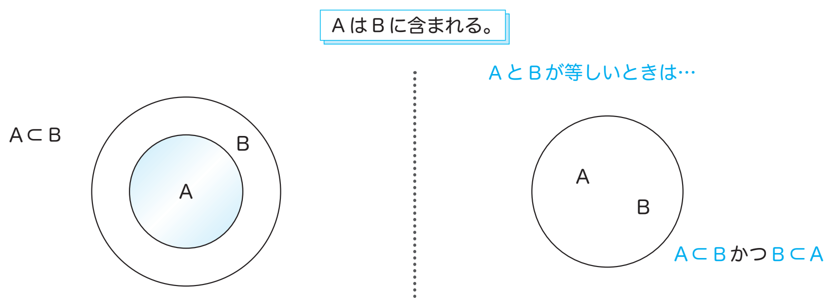


Fig.7-2　部分集合

真部分集合

集合Aのすべての要素が集合Bの要素であって、集合Aと集合Bが等しく**な**いとき、すなわち $A \subset B$ かつ $A \neq B$ であるとき、AはBの**真部分集合**（*proper subset*）です。

この関係は $A \subsetneq B$ あるいは $B \supsetneq A$ と表します。

例　$A = \{1, 3\}$ で $B = \{1, 3, 5\}$ のとき
　　AはBの部分集合であり、AはBの真部分集合である。

例　$A = \{1, 3, 5\}$ で $B = \{1, 3, 5\}$ のとき
　　AはBの部分集合であるが、AはBの真部分集合ではない。

▶　部分集合の関係を $A \subseteq B$ と表記して、真部分集合の関係を $A \subset B$ と表記する流儀もあります。

集合の演算

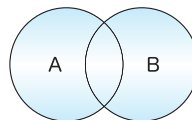
集合に対する基本的な演算は、和・積・差を求める演算です。

和

集合Aと集合Bの少なくとも一方に属している要素の集合をAとBの**和集合**と呼び、A∪Bと書き表します。

これを図で表したのが**Fig.7-3**です。

例　A={1, 3, 5}でB={1, 4, 6}のとき
　　A∪Bは{1, 3, 4, 5, 6}です。



AとBのいずれかに含まれる要素の集合

Fig.7-3　集合の和

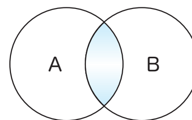
積

集合Aと集合Bの両方に属している要素の集合をAとBの**積集合**と呼び、A∩Bと書き表します。

これを図で表したのが**Fig.7-4**です。

例　A={1, 3, 5}でB={1, 4, 6}のとき
　　A∩Bは{1}です。

例　A={3, 5}でB={1, 4, 6}のとき
　　A∩Bはφ（空集合）です。



AとBの両方に含まれる要素の集合

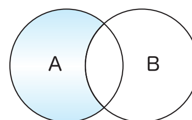
Fig.7-4　集合の積

差

集合Aの要素であって集合Bに属さない要素の集合を**差集合**と呼び、A－Bと書き表します。

これを図で表したのが**Fig.7-5**です。

例　A={1, 3, 5}でB={1, 4, 6}のとき
　　A－Bは{3, 5}です。



Aに含まれBに含まれない要素の集合

Fig.7-5　集合の差

7-2 配列による集合

同じ型のデータの集合は、配列によって表せます。本節で学習するのは、配列によって実現する集合です。

配列による集合

すべての要素が同じ型の集合は、配列によって容易に実現できます。たとえば、整数の集合 {1, 2, 3, 4, 5, 6, 7, 8} は、以下のように、要素数 8 の **int** 型配列に格納できます。

▶ ものの《集まり》が表現できればよいため、配列内での要素の順序は**任意**です。

もっとも、配列の全要素を用いて集合を表すのであれば、集合の要素数と配列の要素数とを常に一致させるための工夫が必要です。配列本体と、“現在いくつの要素が集合に入っているのか”を表す変数とを組み合わせるのが現実的です。

そこで、**int** 型を要素とする集合を、**Fig.7-6** に示す構造体 *IntSet* によって管理することにします。

▶ 配列を構造体で管理するという点では、第 4 章で学習したスタックやキューと同じ要領です。

```
typdef struct {
    int max;      /* 集合の容量 */
    int num;      /* 集合の要素数 */
    int *set;     /* 集合本体 */
} IntSet;
```

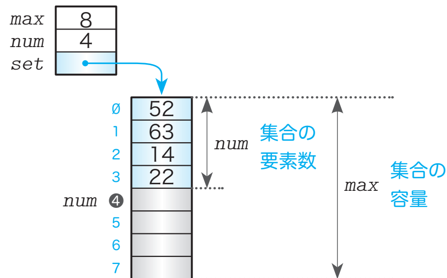


Fig.7-6　配列による集合を管理するための構造体 IntSet

構造体 *IntSet* は、三つのメンバで構成されます。

▪ *max*

集合の容量、すなわち配列の要素数を表すメンバです。ここに示す図の場合であれば、*max* の値は 8 です。

▪ *num*

集合の要素数です。集合の要素は、配列の先頭側（添字の小さいほうの要素）に格納します。すなわち、*set*[0]〜*set*[*num* - 1] の *num* 個の要素が、集合の要素です。この図の場合、*num* の値は 4 です。なお、配列が空集合であれば、この値は 0 となります。

■ *set*

集合を格納するための配列です（厳密には、配列の先頭要素へのポインタです）。

▶ 配列用記憶域の確保は、関数 *Initialize* で行います。

この構造体によって管理する **int** 型集合プログラムのヘッダ部を **List 7-1** に、ソース部を **List 7-2**（次ページ）に示します。

List 7-1 chap07/IntSet.h

```
/* int型集合IntSet（ヘッダ部） */

#ifndef ___IntSet
#define ___IntSet

/*--- int型の集合を実現する構造体 ---*/
typedef struct {
    int max;     /* 集合の容量 */
    int num;     /* 集合の要素数 */
    int *set;    /* 集合本体の配列（の先頭要素へのポインタ） */
} IntSet;

/*--- 集合の初期化 ---*/
int Initialize(IntSet *s, int max);

/*--- 集合sにnが入っているか ---*/
int IsMember(const IntSet *s, int n);

/*--- 集合sにnを追加 ---*/
void Add(IntSet *s, int n);

/*--- 集合sからnを削除 ---*/
void Remove(IntSet *s, int n);

/*--- 集合sが格納できる最大の要素数 ---*/
int Capacity(const IntSet *s);

/*--- 集合sの要素数 ---*/
int Size(const IntSet *s);

/*--- 集合s2をs1に代入 ---*/
void Assign(IntSet *s1, const IntSet *s2);

/*--- 集合s1とs2は等しいか ---*/
int Equal(const IntSet *s1, const IntSet *s2);

/*--- 集合s2とs3の和集合をs1に代入 ---*/
IntSet *Union(IntSet *s1, const IntSet *s2, const IntSet *s3);

/*--- 集合s2とs3の積集合をs1に代入 ---*/
IntSet *Intersection(IntSet *s1, const IntSet *s2, const IntSet *s3);

/*--- 集合s2からs3を引いた集合をs1に代入 ---*/
IntSet *Difference(IntSet *s1, const IntSet *s2, const IntSet *s3);

/*--- 集合sの全要素を表示 ---*/
void Print(const IntSet *s);

/*--- 集合sの全要素を表示（改行付き） ---*/
void PrintLn(const IntSet *s);

/*--- 集合の後始末 ---*/
void Terminate(IntSet *s);

#endif
```

List 7-2 [A] chap07/IntSet.c

```
/* int型集合IntSet（ソース部） */

#include <stdio.h>
#include <stdlib.h>
#include "IntSet.h"

/*--- 集合の初期化 ---*/
int Initialize(IntSet *s, int max)
{
    s->num = 0;
    if ((s->set = calloc(max, sizeof(int))) == NULL) {
        s->max = 0;                                 /* 配列の生成に失敗 */
        return -1;
    }
    s->max = max;
    return 0;
}

/*--- 集合sにnが入っているか ---*/
int IsMember(const IntSet *s, int n)
{
    int i;

    for (i = 0; i < s->num; i++)
        if (s->set[i] == n)
            return i;               /* 含まれる（添字を返却） */
    return -1;                      /* 含まれない */
}

/*--- 集合sにnを追加 ---*/
void Add(IntSet *s, int n)
{
    if (s->num < s->max && IsMember(s, n) == -1)    /* 含まれなかったら */
        s->set[s->num++] = n;                       /* 配列の末尾に追加 */
}

/*--- 集合sからnを削除 ---*/
void Remove(IntSet *s, int n)
{
    int idx;

    if ((idx = IsMember(s, n)) != -1) {
        s->set[idx] = s->set[--s->num];     /* 末尾の要素を削除位置に移動 */
    }
}
```

配列の初期化：Initialize

関数 *Initialize* は、集合本体用の配列を生成するなどの準備処理を行います。

初期状態の集合は空（データが 1 個もない状態）ですから、*num* の値を 0 にします。

そして、要素数が *max* である配列 *set* の本体を生成して、仮引数 *max* に受け取った《集合の容量》を、メンバ *max* にコピーします。

要素として入っているか：IsMember

関数 *IsMember* は、集合本体の配列 *set* に、値 *n* の要素が入っているかどうかを調べる関数です。利用するアルゴリズムは、3-2 節で学習した線形探索です。

Fig.7-7 に示すように、配列の先頭から走査し、探索成功時は見つけた要素の添字を返し、失敗時は `-1` を返します。図の場合は、探索に成功して 4 を返却します。

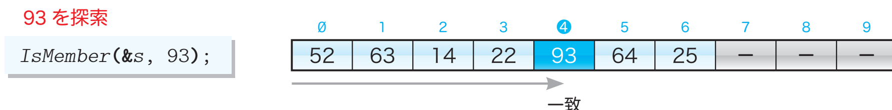


Fig.7-7 集合からの探索

要素の追加：Add

関数 *Add* は、集合に要素 *n* を追加する関数です。追加を行うのは、配列が満杯でなくて、かつ、集合に *n* が入っていないときです。

追加の手続きは単純です。**Fig.7-8** に示すように、末尾要素の次の要素である *set*[*num*] に *n* を代入して、その後 *num* をインクリメントするだけです。

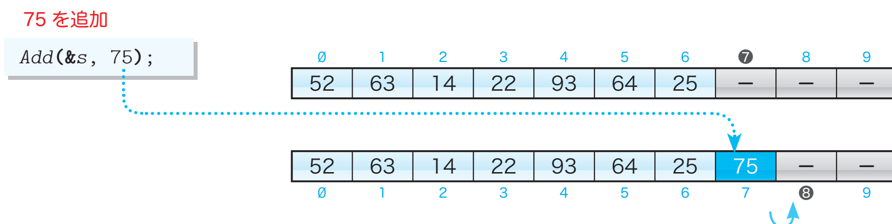


Fig.7-8 集合への追加

要素の削除：Remove

関数 *Remove* は、集合から要素 *n* を削除する関数です。削除を行うのは、集合内に *n* が含まれているときのみです。

Fig.7-9 に示すのは、集合から 22 の削除を行う手続きです。

まず 22 が入っている要素の添字 3 を関数 *IsMember* で調べて *idx* に代入します。その後、要素数 *num* を 7 から 6 へとデクリメントし、末尾要素 *set*[*num*] すなわち *set*[6] の値を、*set*[*idx*] すなわち *set*[3] にコピーすると、作業は完了です。

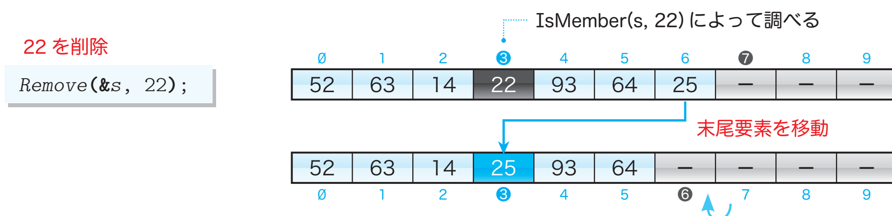


Fig.7-9 集合からの削除

List 7-2 [B]　　chap07/IntSet.c

```
/*--- 集合sが格納できる最大の要素数 ---*/
int Capacity(const IntSet *s)
{
    return s->max;
}

/*--- 集合sの要素数 ---*/
int Size(const IntSet *s)
{
    return s->num;
}

/*--- 集合s2をs1に代入 ---*/
void Assign(IntSet *s1, const IntSet *s2)
{
    int i;
    int n = (s1->max < s2->num) ? s1->max : s2->num;    /* コピーする要素数 */

    for (i = 0; i < n; i++)
        s1->set[i] = s2->set[i];
    s1->num = n;
}

/*--- 集合s1とs2は等しいか ---*/
int Equal(const IntSet *s1, const IntSet *s2)
{
    int i, j;

    if (Size(s1) != Size(s2))
        return 0;

    for (i = 0; i < s1->num; i++) {
        for (j = 0; j < s2->num; j++)
            if (s1->set[i] == s2->set[j])
                break;
        if (j == s2->num)
            return 0;
    }
    return 1;
}

/*--- 集合s2とs3の和集合をs1に代入 ---*/
IntSet *Union(IntSet *s1, const IntSet *s2, const IntSet *s3)
{
    int i;

    Assign(s1, s2);
    for (i = 0; i < s3->num; i++)
        Add(s1, s3->set[i]);

    return s1;
}
```

集合の容量を調べる：Capacity

関数 *Capacity* は、集合の容量（集合が含むことのできる最大の要素数）を返す関数です。メンバ *max* の値をそのまま返します。

集合の要素数を調べる：Size

関数 *Size* は、集合の要素数を返す関数です。メンバ *num* の値をそのまま返します。

集合の代入：Assign

関数 *Assign* は、ある集合を別の集合にコピーする関数です。コピー元は *s2* が指す集合で、コピー先は *s1* が指す集合です（**Fig.7-10**）。

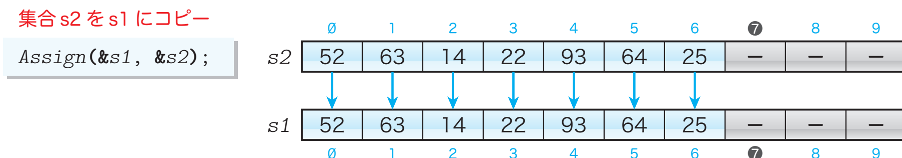


Fig.7-10　集合のコピー

なお、コピー元の要素数 *s2*->*num* がコピー先集合の容量 *s1*->*max* を超える場合は、コピー先集合の容量である *s1*->*max* 個の要素のみをコピーします。

▶　コピー元の要素数を記憶させるための変数 *n* の宣言は、以下のようにも実現できます。

```
int n = (Capacity(s1) < Size(s2)) ? Capacity(s1) : Size(s2);
```

集合が等しいかどうかを調べる：Equal

関数 *Equal* は、二つの集合 *s1* と *s2* が等しいかどうかを判定する関数です。等しければ 1 を返し、等しくなければ 0 を返します。

集合の等価性の判断は、以下のように行います。

▪ 要素数が等しくない場合

集合は等しくないと判断して 0 を返します。

▪ 要素数が等しい場合（プログラム網かけ部）

i を 0, 1, …, *num* - 1 とインクリメントして、集合 *s1* の要素 *s1*->*set*[*i*] が、集合 *s2* に含まれるかどうかを調べます。含まれない要素が見つかったら、集合は等しくないと判定します（**break** 文によって内側の **for** 文を中断した後に、**return** 文によって 0 を返します）。

中断されることなく外側の **for** 文が完了すれば、集合は等しいと判定できますので、そのことを表す 1 を返します。

二つの集合の和集合を求める：Union

関数 *Union* は、集合 *s2* と集合 *s3* の和集合を *s1* に求める関数です。

まず関数 *Assign* によって集合 *s2* を *s1* にコピーして、それから *s3* の全要素を一つずつ *s1* に追加します。これで、*s1* に和集合が格納されます。

このようにして求められた和集合 *s1*（へのポインタ）を返却します。

List 7-2 [C]　　chap07/IntSet.c

```
/*--- 集合s2とs3の積集合をs1に代入 ---*/
IntSet *Intersection(IntSet *s1, const IntSet *s2, const IntSet *s3)
{
    int i, j;

    s1->num = 0;                /* s1を空集合にする */
    for (i = 0; i < s2->num; i++)
        for (j = 0; j < s3->num; j++)
            if (s2->set[i] == s3->set[j])
                Add(s1, s2->set[i]);
    return s1;
}

/*--- 集合s2からs3を引いた集合をs1に代入 ---*/
IntSet *Difference(IntSet *s1, const IntSet *s2, const IntSet *s3)
{
    int i, j;

    s1->num = 0;                /* s1を空集合にする */
    for (i = 0; i < s2->num; i++) {
        for (j = 0; j < s3->num; j++)
            if (s2->set[i] == s3->set[j])
                break;
        if (j == s3->num)
            Add(s1, s2->set[i]);
    }
    return s1;
}

/*--- 集合sの全要素を表示 ---*/
void Print(const IntSet *s)
{
    int i;

    printf("{ ");
    for (i = 0; i < s->num; i++)
        printf("%d ", s->set[i]);
    printf("}");
}

/*--- 集合sの全要素を表示（改行付き）---*/
void PrintLn(const IntSet *s)
{
    Print(s);
    putchar('\n');
}

/*--- 集合の後始末 ---*/
void Terminate(IntSet *s)
{
    if (s->set != NULL) {
        free(s->set);                        /* 配列を破棄 */
        s->max = s->num = 0;
    }
}
```


二つの集合の積集合を求める：Intersection

関数 *Intersection* は、集合 *s2* と集合 *s3* の積集合を *s1* に求める関数です。

まず、*s1* を空集合とします。それから、集合 *s2* の全要素を走査し、その要素が *s3* に含まれていれば *s1* に追加します。走査が終了した時点で、*s1* には積集合が格納されています。

このようにして求められた積集合 *s1*（へのポインタ）を返却します。

二つの集合の差集合を求める：Difference

関数 *Difference* は、集合 *s2* と集合 *s3* の差集合を *s1* に求める関数です。

まず、*s1* を空集合とします。それから、集合 *s2* の全要素を走査し、その要素が *s3* に含まれていなければ *s1* に追加します。走査が終了した時点で、*s1* には差集合が格納されています。

全要素の表示：Print ／ PrintLn

関数 *Print* と関数 *PrintLn* は、集合の全要素を表示する関数です。

たとえば、集合 s の要素が 1，5，7 であれば、「{ 1 5 7 }」と表示します。

▶ 関数 *Print* は集合のみを表示し、関数 *PrintLn* は集合の表示後に改行文字を出力します。

後始末：Terminate

関数 *Terminate* は、集合の後始末をする関数です。集合用の配列の破棄などを行います。

演習 7-1

構造体 *IntSet* で管理する集合プログラムに対して、以下の関数を追加せよ。

- 集合が満杯（これ以上要素を追加できない状態）であるかどうかを判定する関数。満杯であれば 1 を、満杯でなければ 0 を返却する。

```
int IsFull(const IntSet *s);
```

- 集合の全要素を削除する関数。

```
void Clear(IntSet *s);
```

- 集合 *s2* と集合 *s3* の対称差（一方の集合には含まれるが両方には含まれない要素の集合）を求めて *s1* に代入する関数。

```
IntSet *SymmetricDifference(IntSet *s1, const IntSet *s2, const IntSet *s3);
```

- 集合 *s1* に対して、集合 *s2* の全要素を追加する（*s1* を *s2* との和集合に更新する）関数。

```
IntSet *ToUnion(IntSet *s1, const IntSet *s2);
```

- 集合 *s1* から、集合 *s2* に含まれない全要素を削除する（*s1* を *s2* との積集合に更新する）関数。

```
IntSet *ToIntersection(IntSet *s1, const IntSet *s2);
```

- 集合 *s1* から、集合 *s2* に含まれる全要素を削除する関数。

```
IntSet *ToDifference(IntSet *s1, const IntSet *s2);
```

※上記三つの関数は、いずれも *s1* の値をそのまま返却する。

- 集合 *s1* が集合 *s2* の部分集合であるかどうかを判定する関数。部分集合であれば 1 を、部分集合でなければ 0 を返却する。

```
int IsSubset(const IntSet *s1, const IntSet *s2);
```

- 集合 *s1* が集合 *s2* の真部分集合であるかどうかを判定する関数。真部分集合であれば 1 を、真部分集合でなければ 0 を返却する。

```
int IsProperSubset(const IntSet *s1, const IntSet *s2);
```

演習 7-2

配列内の要素を常に昇順にソートしておくように変更した集合プログラムを作成せよ。要素の探索は２分探索によって行え、要素の追加や削除も２分探索によって得られた位置に対して行える。また、他の配列と等しいかどうかの判断も効率よく行える（ただし、要素の挿入・削除の効率は悪くなる）。なお、集合を管理する構造体に与える型名は *SortedIntSet* とすること。

集合 *IntSet* を利用する二つのプログラム例を **List 7-3** と **List 7-4** に示します。

▶ いずれのプログラムのコンパイルにも、"IntSet.h" と "IntSet.c" が必要です。

List 7-3 chap07/IntSetTest1.c

```c
/* int型集合IntSetの利用例（その１） */

#include <stdio.h>
#include "IntSet.h"

int main(void)
{
    IntSet s1, s2, s3;
    Initialize(&s1, 24);
    Initialize(&s2, 24);
    Initialize(&s3, 24);

    Add(&s1, 10);        /* s1 = {10}                */
    Add(&s1, 15);        /* s1 = {10, 15}            */
    Add(&s1, 20);        /* s1 = {10, 15, 20}        */
    Add(&s1, 25);        /* s1 = {10, 15, 20, 25}    */

    Assign(&s2, &s1);    /* s2 = {10, 15, 20, 25}    */
    Add(&s2, 12);        /* s2 = {10, 15, 20, 25, 12} */
    Remove(&s2, 20);     /* s2 = {10, 15, 12, 25}    */

    Assign(&s3, &s2);    /* s3 = {10, 15, 12, 25}    */

    printf("s1 = ");   PrintLn(&s1);
    printf("s2 = ");   PrintLn(&s2);
    printf("s3 = ");   PrintLn(&s3);

    printf("集合s1に15は含まれ%s。\n", IsMember(&s1, 15) > 0 ? "る" : "ない");
    printf("集合s2に25は含まれ%s。\n", IsMember(&s2, 25) > 0 ? "る" : "ない");
    printf("集合s1とs2は等し%s。\n", Equal(&s1, &s2) ? "い" : "くない");
    printf("集合s2とs3は等し%s。\n", Equal(&s2, &s3) ? "い" : "くない");

    Terminate(&s1);
    Terminate(&s2);
    Terminate(&s3);

    return 0;
}
```

実行結果

```
s1 = { 10 15 20 25 }
s2 = { 10 15 12 25 }
s3 = { 10 15 12 25 }
集合s1に15は含まれる。
集合s2に25は含まれる。
集合s1とs2は等しくない。
集合s2とs3は等しい。
```

▶ 以下に示すのは、右ページに示す **List 7-4** の実行例です。

実行例

```
s1 = { }
s2 = { }
(1)s1に追加 (2)s1から削除 (3)s1から探索 (4)s1←s2 (5)各種演算
(6)s2に追加 (7)s2から削除 (8)s2から探索 (9)s2←s1 (0)終了：1
追加するデータ：1
s1 = { 1 }
s2 = { }
(1)s1に追加 (2)s1から削除 (3)s1から探索 (4)s1←s2 (5)各種演算
(6)s2に追加 (7)s2から削除 (8)s2から探索 (9)s2←s1 (0)終了：1
追加するデータ：3
s1 = { 1 3 }
s2 = { }
(1)s1に追加 (2)s1から削除 (3)s1から探索 (4)s1←s2 (5)各種演算
(6)s2に追加 (7)s2から削除 (8)s2から探索 (9)s2←s1 (0)終了：9
s1 = { 1 3 }
s2 = { 1 3 }
(1)s1に追加 (2)s1から削除 (3)s1から探索 (4)s1←s2 (5)各種演算
(6)s2に追加 (7)s2から削除 (8)s2から探索 (9)s2←s1 (0)終了：1
追加するデータ：5
s1 = { 1 3 5 }
s2 = { 1 3 }
(1)s1に追加 (2)s1から削除 (3)s1から探索 (4)s1←s2 (5)各種演算
(6)s2に追加 (7)s2から削除 (8)s2から探索 (9)s2←s1 (0)終了：5
s1 == s2 = false
s1 & s2 = { 1 3 }
s1 | s2 = { 1 3 5 }
s1 - s2 = { 5 }
s1 = { 1 3 5 }
s2 = { 1 3 }
(1)s1に追加 (2)s1から削除 (3)s1から探索 (4)s1←s2 (5)各種演算
(6)s2に追加 (7)s2から削除 (8)s2から探索 (9)s2←s1 (0)終了：2
削除するデータ：1
s1 = { 5 3 }
s2 = { 1 3 }
(1)s1に追加 (2)s1から削除 (3)s1から探索 (4)s1←s2 (5)各種演算
(6)s2に追加 (7)s2から削除 (8)s2から探索 (9)s2←s1 (0)終了：3
探索するデータ：5
s1に含まれています。
s1 = { 5 3 }
s2 = { 1 3 }
(1)s1に追加 (2)s1から削除 (3)s1から探索 (4)s1←s2 (5)各種演算
(6)s2に追加 (7)s2から削除 (8)s2から探索 (9)s2←s1 (0)終了：0
```

List 7-4 chap07/IntSetTest2.c

```
/* int型集合IntSetの利用例（その２）*/

#include <stdio.h>
#include "IntSet.h"

enum { ADD, RMV, SCH };

/*--- データ読込み ---*/
int scan_data(int sw)
{
    int data;
    switch (sw) {
     case ADD: printf("追加するデータ："); break;
     case RMV: printf("削除するデータ："); break;
     case SCH: printf("探索するデータ："); break;
    }
    scanf("%d", &data);
    return data;
}

int main(void)
{
    IntSet s1, s2, temp;
    Initialize(&s1, 512);  Initialize(&s2, 512);  Initialize(&temp, 512);

    while (1) {
        int m, x, idx;

        printf("s1 = ");   PrintLn(&s1);
        printf("s2 = ");   PrintLn(&s2);
        printf("(1)s1に追加 (2)s1から削除 (3)s1から探索 (4)s1←s2 (5)各種演算\n"
               "(6)s2に追加 (7)s2から削除 (8)s2から探索 (9)s2←s1 (0)終了：");
        scanf("%d", &m);

        if (m == 0) break;

        switch (m) {
         case 1: x = scan_data(ADD);  Add(&s1, x);      break;  /* s1に追加 */
         case 2: x = scan_data(RMV);  Remove(&s1, x);   break;  /* s1から削除 */
         case 3: x = scan_data(SCH);  idx = IsMember(&s1, x);   /* s1から探索 */
                 printf("s1に含まれていま%s。\n", idx >= 0 ? "す" : "せん"); break;
         case 4: Assign(&s1, &s2);  break;      /* s2をs1に代入 */
         case 5: printf("s1 == s2 = %s\n", Equal(&s1, &s2) ? "true" : "false");
                 printf("s1 & s2 = ");  Intersection(&temp, &s1, &s2);
                                        PrintLn(&temp);
                 printf("s1 | s2 = ");  Union(&temp, &s1, &s2);
                                        PrintLn(&temp);
                 printf("s1 - s2 = ");  Difference(&temp, &s1, &s2);
                                        PrintLn(&temp);
                 break;
         case 6: x = scan_data(ADD);  Add(&s2, x);      break;  /* s2に追加 */
         case 7: x = scan_data(RMV);  Remove(&s2, x);   break;  /* s2から削除 */
         case 8: x = scan_data(SCH);  idx = IsMember(&s2, x);   /* s2から探索 */
                 printf("s2に含まれていま%s。\n", idx >= 0 ? "す" : "せん"); break;
         case 9: Assign(&s2, &s1);  break;      /* s1をs2に代入 */
        }
    }

    Terminate(&s1);  Terminate(&s2);  Terminate(&temp);

    return 0;
}
```

7-3 ビットベクトルによる集合

本節では、ビットベクトルによる配列の実現法を学習します。

ビットベクトルによる集合

ある整数値maxの値が小さければ、0, 1, …, max - 1の整数を要素とする集合は、単一の整数で表現できます。その実現法を学習していきましょう。

ここでは、**Fig.7-11**に示すように、**unsigned long**型が32ビットと仮定して話を進めます。なお、整数を構成するビットの並びは、**ビットベクトル**（*bit vector*）と呼ばれますので、今後は、この語句を使っていきます。

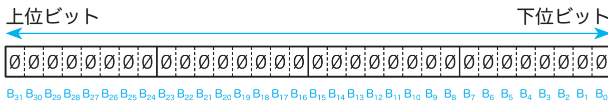


Fig.7-11　unsigned long型の内部のビット

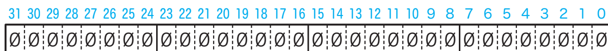
図に示すように、右側の最下位ビットから始めて、各ビットをB_0, B_1, B_2, …, B_{31}と呼ぶことにします。

ビットベクトル中のビットを各要素の有無に対応させ、集合に*x*が含まれていればB*x*を1とし、含まれていなければB*x*を0とすると、32ビットの符号無し整数1個で、{0, 1, …, 31}を**普遍集合**（*universal set*）とする、整数の集合が表現できます。

▶ 普遍集合は、考慮の対象となる要素すべてを含んだ集合のことであり、**全体集合**あるいは**全集合**とも呼ばれます。ビットベクトルで表現できる集合は、普遍集合である{0, 1, …, 31}の部分集合に限られます。

たとえば、要素が一つもない空集合ϕは**Fig.7-12** aのように表現されます。また、集合{1, 5, 7, 17, 23}は図bのように表現されます。

a 空集合



b 集合{1, 5, 7, 17, 23}

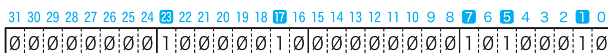


Fig.7-12　ビットベクトルによる集合の表現の一例

それでは、**unsigned long** 型で表すビットベクトルで実現する集合のプログラムを作成していきましょう。そのプログラムのヘッダ部が **List 7-5** です。

List 7-5 chap07/BitSet.h

```
/* ビットベクトル集合BitSetライブラリ（ヘッダ部）*/

#ifndef ___BitSet
#define ___BitSet

typedef unsigned long  BitSet;                /* 集合を表す型 */

#define BitSetNull      (BitSet)0             /* 空集合 */
#define BitSetBits      32                    /* 有効なビット数 */

#define SetOf(no)   ((BitSet)1 << (no))       /* 集合{no} */

/*--- 集合sにnが入っているか ---*/
int IsMember(BitSet s, int n);

/*--- 集合sにnを追加 ---*/
void Add(BitSet *s, int n);

/*--- 集合sからnを削除 ---*/
void Remove(BitSet *s, int n);

/*--- 集合sの要素数 ---*/
int Size(BitSet s);

/*--- 集合sの全要素を表示 ---*/
void Print(BitSet s);

/*--- 集合sの全要素を表示（改行付き）---*/
void PrintLn(BitSet s);

#endif
```

定義されている型やマクロなどを理解していきます。

■ *BitSet* 型

BitSet は、ビットベクトルを表す型です。**typedef** 宣言によって、**unsigned long** 型の同義語として定義されています。

■ *BitSetNull*

オブジェクト形式マクロ *BitSetNull* は、空集合のビットベクトルを表す定数です。すべてのビットが 0 である **unsigned long** 型の定数値です。

■ *BitSetBits*

オブジェクト形式マクロ *BitSetBits* は、ビットベクトルで有効なビット数を表す定数です。本プログラムでは、**unsigned long** 型のビットの下位 32 ビットのみを利用する仕様としているため、この値は 32 と定義されています。

▶ C言語の定義により、**long** 型は、少なくとも 32 ビットであることが保証されます（それ以上のビット数で **long** 型を表現する処理系であっても、本プログラムで利用するのは下位 32 ビットのみです）。

▪ *SetOf(no)*

関数形式マクロ *SetOf* は、唯一の要素が *no* である集合 {*no*} を表すビットベクトルを生成します（以下に、定義を再掲します）。

```
#define SetOf(no)   ((BitSet)1 << (no))          /* 集合{no} */
```

Fig.7-13 に示すのは、*SetOf*(5) が {5} を生成する様子です。最下位ビットのみが 1 である {0} のビットベクトルを、左に *no* ビットシフトすることで、集合 {*no*} を生成します。

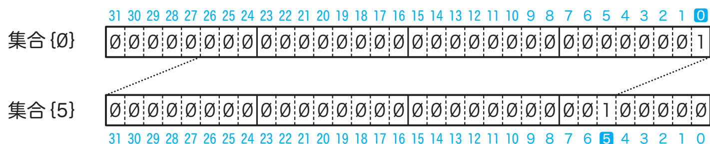


Fig.7-13 SetOf(no) による集合 {no} の生成

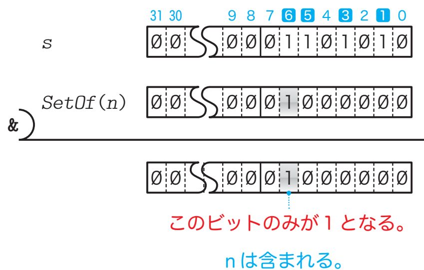
List 7-6 に示すのが、ビットベクトルのプログラムのソース部です。各関数を理解していきましょう。

要素として入っているか：IsMember

関数 *IsMember* は、集合 *s* に、値 *n* の要素が入っているかどうかを調べる関数です。集合 *s* のビットベクトルと、{*n*} のビットベクトルの論理積を取るだけで判定を行います。

判定の様子を示したのが、**Fig.7-14** です。図**a**のように、集合に *n* が入っている場合は、ビット単位の論理積によって得られるのは {*n*} です。また、図**b**のように、集合に *n* が入っていない場合は、論理積によって得られるのは空集合です。

a 集合{1, 3, 5, 6}に6は入っているか？



b 集合{1, 3, 5, 6}に4は入っているか？

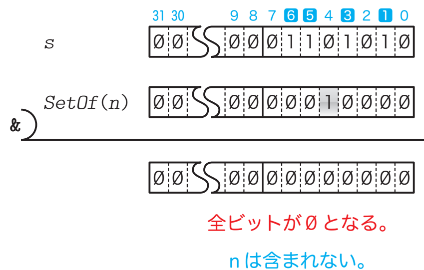


Fig.7-14 関数 IsMember の働き

List 7-6 [A]　　chap07/BitSet.c

```
/* ビットベクトルによる整数集合演算（ソース部）*/

#include <stdio.h>
#include "BitSet.h"

/*--- 集合sにnが入っているか ---*/
int IsMember(BitSet s, int n)
{
    return s & SetOf(n);
}

/*--- 集合sにnを追加 ---*/
void Add(BitSet *s, int n)
{
    *s |= SetOf(n);
}
```

すなわち、論理積の演算結果は、集合 s に n が入っていれば 0 以外の値となり、入っていなければ 0 となります。この関数は、この演算結果をそのまま返却します。

▶　C 言語では、0 は偽と見なされ、0 以外の値は真とみなされます（**Column 1-6**：p.29）ので、以下のように簡潔な式での判定が行えます。

```
if (IsMember(&s, n))
    /* 集合sにnが含まれているときに実行される文 */
```

要素の追加：Add

関数 *Add* は、集合 s に要素 n を追加する関数です。

Fig.7-15 に示すように、集合 s のビットベクトルを、*SetOf*(n) のビットベクトルとのビット単位の論理和に更新するだけで、追加の処理を実現しています。

図**a**のように、もともと集合に n が入っている場合は、集合 s は更新されません。また、図**b**のように、集合に n が入っていない場合は、演算によって n に対応するビットが 0 から 1 に変更されます。

a 集合{1, 3, 5, 6}に6を追加

b 集合{1, 3, 5, 6}に4を追加

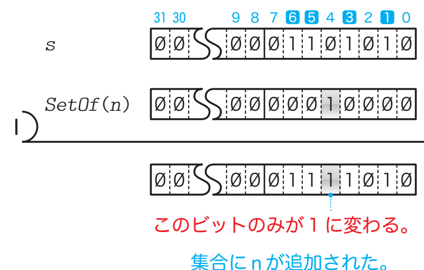


Fig.7-15　関数 Add の働き

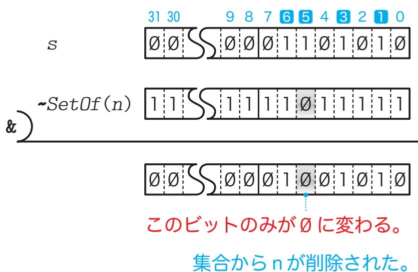
List 7-6 [B] chap07/BitSet.c

```
/*--- 集合sからnを削除 ---*/
void Remove(BitSet *s, int n)
{
    *s &= ~SetOf(n);
}

/*--- 集合sの要素数 ---*/
int Size(BitSet s)
{
    int n = 0;

    for ( ; s != BitSetNull; n++)
        s &= s - 1;
    return n;
}

/*--- 集合sの全要素を表示 ---*/
void Print(BitSet s)
{
    int i;

    printf("{ ");
    for (i = 0; i < BitSetBits; i++)
        if (IsMember(s, i))
            printf("%d ", i);
    printf("}");
}

/*--- 集合sの全要素を表示（改行付き）---*/
void PrintLn(BitSet s)
{
    Print(s);
    putchar('\n');
}
```


要素の削除：Remove

関数 *Remove* は、集合 *s* から要素 *n* を削除する関数です。

削除の具体例を示したのが、**Fig.7-16** です。集合 *s* のビットベクトルを、*SetOf*(*n*) の補数（B_n のみが 0 で、それ以外の全ビットが 1 である整数）との論理積に更新します。



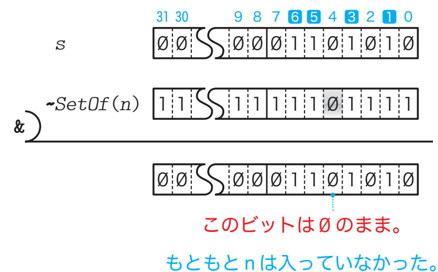


Fig.7-16 関数 Remove による要素の削除

▶ 図**a**は、集合{1, 3, 5, 6}から5を削除する例です。B_5が1からØに変更されます。また、図**b**は、集合{1, 3, 5, 6}から4を削除する例です。要素4は集合に含まれないため、*s*は変化しません。

集合の要素数を調べる：Size

関数 *Size* は、要素数を調べる関数です。

集合の要素数は、ビットベクトル中の“1”であるビットの数と一致します。そこで、1のビットをカウントして要素数を求めます。

*

Fig.7-17 を見ながら、ビットをカウントする手続きを理解していきましょう。なお、ここに示す例は、集合{1, 3, 5, 6}の要素数を求める手順です。

図**a**に示すように、*s*を、それから1を引いた*s* - 1との論理積に更新します。その結果、*s*中の最も下位側の1であるビットB_1が1からØに変化します。

▶ すなわち、集合{1, 3, 5, 6}から、最も小さい要素1が削除されます。

更新された*s*に対して同じ演算を行ったのが、図**b**です。ここでも、最も下位側の1であるビットB_3が1からØに変化します。

▶ すなわち、集合{3, 5, 6}から、最も小さい要素3が削除されます。

図**c**と図**d**も同様の演算であり、図**d**の演算後は、すべてのビットがØとなります。

演算を行うたびに要素が1個ずつ削除されるのですから、演算回数が、集合の要素数を表します。

▶ ビット数を格納するのが、変数*n*です。繰返しの前にØで初期化され、繰返しを行うたびにインクリメントされます。

Fig.7-17 要素数のカウント

全要素の表示：Print／PrintLn

関数 *Print* と *PrintLn* は、集合の全要素を表示する関数です。

たとえば、集合*s*の要素が1, 5, 7であれば、「{ 1 5 7 }」と表示します。

▶ 関数 *Print* は集合の表示を行い、関数 *PrintLn* は集合の表示後に改行文字を出力します。

「ビットベクトルによる集合」のプログラムで作られた関数は、前節で学習した「配列による集合」のプログラムに比べると、数が少なくなっています。

というのも、わざわざ関数を作成しなくても、各種の演算が実行できるからです。ここでは、以下のように宣言された集合 *s1* と *s2* を例に、集合に対する演算について考えていきましょう。

```
BitSet s1, s2;
```

等価性の判定

二つの集合 *s1* と *s2* が等しいか、あるいは、等しくないかの判定は、等価演算子 **==** と **!=** によって実現できます。

```
s1 == s2            /* 集合s1と集合s2は等しいか？ */
s1 != s2            /* 集合s1と集合s2は等しくないか？ */
```

というのも、二つの集合が等しければ、すべてのビットの状態が等しい（すなわち同一の整数値である）からです。

なお、いずれの判定も、判定が成立すれば **int** 型の 1 となり、成立しなければ **int** 型の 0 となります。

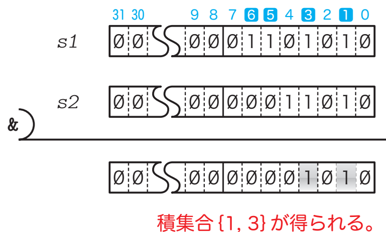
積集合を求める

集合 *s1* と *s2* の積集合は、ビット単位の論理積演算子 **&** によって求められます。

```
s1 & s2             /* 集合s1と集合s2の積集合 */
```

演算の一例を示したのが、**Fig.7-18** です。

a 集合{1, 3, 5, 6}と{1, 3, 4}の積集合を求める



b 集合{1, 3, 5, 6}と{2, 4, 7}の積集合を求める

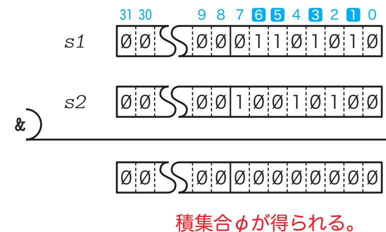


Fig.7-18　二つの集合の積集合を求める

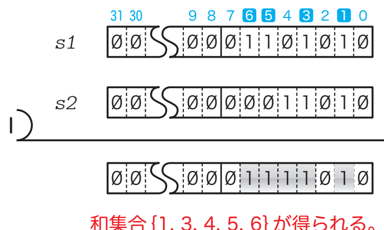
和集合を求める

集合 *s1* と *s2* の和集合は、ビット単位の論理和演算子 | によって求められます。

```
s1 | s2                 /* 集合s1と集合s2の和集合 */
```

演算の一例を示したのが、**Fig.7-19** です。

a 集合{1, 3, 5, 6}と{1, 3, 4}の和集合を求める



b 集合{1, 3, 5, 6}と{2, 4, 7}の和集合を求める

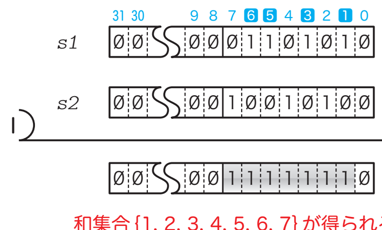


Fig.7-19　二つの集合の和集合を求める

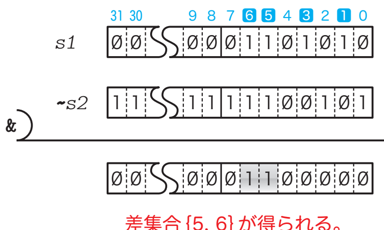
差集合を求める

集合 *s1* と *s2* の差集合は、補数演算子 ~ と、ビット単位の論理積演算子 & との組合せで求められます。

```
s1 & ~s2                /* 集合s1と集合s2の差集合 */
```

演算の一例を示したのが、**Fig.7-20** です。

a 集合{1, 3, 5, 6}を{1, 3, 4}との差集合に更新



b 集合{1, 3, 5, 6}を{2, 4, 7}との差集合に更新

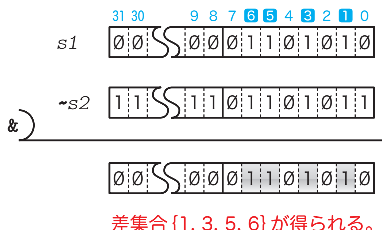


Fig.7-20　二つの集合の差集合を求める

集合 *BitSet* を利用するプログラム例を **List 7-7** に示します。二つの集合 *s1* と *s2* とに対して対話的に演算を行います。

▶ 本プログラムのコンパイルには、"BitSet.h" と "BitSet.c" が必要です。

List 7-7 chap07/BitSetTest.c

```
/* ビットベクトルによる整数集合の利用例 */

#include <stdio.h>
#include "BitSet.h"

enum { ADD, RMV, SCH };

/*--- データ読込み ---*/
int scan_data(int sw)
{
    int data;
    switch (sw) {
     case ADD: printf("追加するデータ："); break;
     case RMV: printf("削除するデータ："); break;
     case SCH: printf("探索するデータ："); break;
    }
    scanf("%d", &data);
    return data;
}

int main(void)
{
    BitSet s1 = BitSetNull;
    BitSet s2 = BitSetNull;

    while (1) {
        int m, x, idx;

        printf("s1 = ");   PrintLn(s1);
        printf("s2 = ");   PrintLn(s2);
        printf("(1)s1に追加 (2)s1から削除 (3)s1から探索 (4)s1←s2 (5)各種演算\n"
               "(6)s2に追加 (7)s2から削除 (8)s2から探索 (9)s2←s1 (0)終了：");
        scanf("%d", &m);

        if (m == 0) break;

        switch (m) {
         case 1: x = scan_data(ADD);  Add(&s1, x);       break;  /* s1に追加 */
         case 2: x = scan_data(RMV);  Remove(&s1, x);    break;  /* s1から削除 */
         case 3: x = scan_data(SCH);  idx = IsMember(s1, x);     /* s1から探索*/
                 printf("s1に含まれていま%s。\n", idx != 0 ? "す" : "せん"); break;
         case 4: s1 = s2;   break;       /* s2をs1に代入 */
         case 5: printf("s1 == s2 = %s\n", s1 == s2 ? "true" : "false");
                 printf("s1 & s2 = ");  PrintLn(s1 & s2);
                 printf("s1 | s2 = ");  PrintLn(s1 | s2);
                 printf("s1 - s2 = ");  PrintLn(s1 & ~s2);
                 break;
         case 6: x = scan_data(ADD);  Add(&s2, x);       break;  /* s2に追加 */
         case 7: x = scan_data(RMV);  Remove(&s2, x);    break;  /* s2から削除 */
         case 8: x = scan_data(SCH);  idx = IsMember(s2, x);     /* s2から探索 */
                 printf("s2に含まれていま%s。\n", idx != 0 ? "す" : "せん"); break;
         case 9: s2 = s1;   break;       /* s1をs2に代入 */
        }
    }

    return 0;
}
```

実行例

```
s1 = { }
s2 = { }
(1)s1に追加 (2)s1から削除 (3)s1から探索 (4)s1←s2 (5)各種演算
(6)s2に追加 (7)s2から削除 (8)s2から探索 (9)s2←s1 (0)終了：1↵
追加するデータ：1↵
s1 = { 1 }
s2 = { }
(1)s1に追加 (2)s1から削除 (3)s1から探索 (4)s1←s2 (5)各種演算
(6)s2に追加 (7)s2から削除 (8)s2から探索 (9)s2←s1 (0)終了：9↵
s1 = { 1 }
s2 = { 1 }
(1)s1に追加 (2)s1から削除 (3)s1から探索 (4)s1←s2 (5)各種演算
(6)s2に追加 (7)s2から削除 (8)s2から探索 (9)s2←s1 (0)終了：1↵
追加するデータ：3↵
s1 = { 1 3 }
s2 = { 1 }
(1)s1に追加 (2)s1から削除 (3)s1から探索 (4)s1←s2 (5)各種演算
(6)s2に追加 (7)s2から削除 (8)s2から探索 (9)s2←s1 (0)終了：6↵
追加するデータ：2↵
s1 = { 1 3 }
s2 = { 1 2 }
(1)s1に追加 (2)s1から削除 (3)s1から探索 (4)s1←s2 (5)各種演算
(6)s2に追加 (7)s2から削除 (8)s2から探索 (9)s2←s1 (0)終了：3↵
探索するデータ：2↵
s1に含まれていません。
s1 = { 1 3 }
s2 = { 1 2 }
(1)s1に追加 (2)s1から削除 (3)s1から探索 (4)s1←s2 (5)各種演算
(6)s2に追加 (7)s2から削除 (8)s2から探索 (9)s2←s1 (0)終了：8↵
探索するデータ：1↵
s2に含まれています。
s1 = { 1 3 }
s2 = { 1 2 }
(1)s1に追加 (2)s1から削除 (3)s1から探索 (4)s1←s2 (5)各種演算
(6)s2に追加 (7)s2から削除 (8)s2から探索 (9)s2←s1 (0)終了：5↵
s1 == s2 = false
s1 &  s2 = { 1 }
s1 |  s2 = { 1 2 3 }
s1 -  s2 = { 3 }
s1 = { 1 3 }
s2 = { 1 2 }
(1)s1に追加 (2)s1から削除 (3)s1から探索 (4)s1←s2 (5)各種演算
(6)s2に追加 (7)s2から削除 (8)s2から探索 (9)s2←s1 (0)終了：2↵
削除するデータ：1↵
s1 = { 3 }
s2 = { 1 2 }
(1)s1に追加 (2)s1から削除 (3)s1から探索 (4)s1←s2 (5)各種演算
(6)s2に追加 (7)s2から削除 (8)s2から探索 (9)s2←s1 (0)終了：5↵
s1 == s2 = false
s1 &  s2 = { }
s1 |  s2 = { 1 2 3 }
s1 -  s2 = { 3 }
s1 = { 3 }
s2 = { 1 2 }
(1)s1に追加 (2)s1から削除 (3)s1から探索 (4)s1←s2 (5)各種演算
(6)s2に追加 (7)s2から削除 (8)s2から探索 (9)s2←s1 (0)終了：0↵
```

演習 7–3

List 7-7 のプログラムの『(5) 各種演算』の箇所に、集合 *s1* と *s2* の対称差を求めて表示する機能を追加せよ。

▶本章には、章末問題はありません。