プログラミング通論'19 ＃9 – 抽象データ型

久野 靖 (電気通信大学)

2020.2.21

今回は次のことが目標となります。

- 抽象データ型の考え方を理解する
- 機能は同一でも時間計算量が異なる場合があることを理解する

抽象データ型

抽象データ型とその必要性

ここまで散々使ってきた「型」ないし「データ型」とは、データの種類を表す言葉でした。たとえば整数型であれば32ビットの符号つき2進表現で、構造体型であれば構造体定義に書かれたフィールドの集まり、という具合です。そしてその構造と機能(整数なら整数演算ができ、構造体なら持っているフィールドの値が読み書きできる)とは一体でした。

これに対し、**抽象データ型**(abstract data type, ADT)とは、「機能は定めるが、内部の構造や機能の実装は隠されている(隠蔽されている)」ような型を言います。

抽象化(abstraction)とは「不必要な細部を見ないで重要なことだけを考える」という意味ですが、プログラムでは機能が動作しさえすれば役に立つので、まさに機能が「重要なこと」であり、中がどのようにできているかは「見なくてよい細部」なのです。

抽象データ型とその必要性

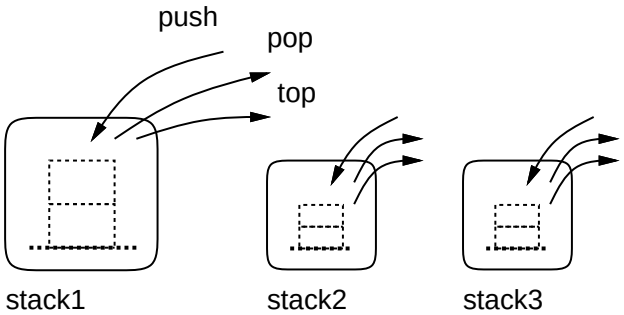

図 1: 抽象データ型としてのスタック

実はこれまでに、抽象データ型の考え方は繰り返し使っています。具体的には、構造体のポインタ型を使った構造の隠蔽を使っているときがそうです (等差数列、スタック、キュー、エディタバッファなど多数ありました)。これらでは、ヘッダファイルを取り込んで使う側は機能 (関数呼び出し) だけしか使わず、内部がどうなっているかは一切分かりません。それでいながら、複数の等差数列やスタックを自由に作れるので、確かに「データの種類」つまり「型」なわけです (図 1)。

抽象データ型とその必要性

それでは、抽象データ型の何がよいのでしょう? 実は、プログラムは大きさが2倍になると、作る手間は2倍よりずっと大きくなります。たとえば、Aというプログラムと Bというプログラムを組み合わせた (サイズ A+B の) プログラムの場合、Aを作る手間と Bを作る手間だけでは済みません。Aから Bの一部を使ったり、Bから Aの一部を使ったりする「組み合せの」手間があるからです。ですから、プログラムが大きくなると、開発の手間は指数関数的に増大してしまいます。

それを避けるには、Aの部分と Bの部分を「できるだけ独立にして」「組合せの部分をなくす」ことです。そうすれば、プログラムの大きさに対してほぼ比例の手間で済むようになります。その最たるものが抽象データ型です。なにしろ、Bは A(抽象データ型の部分) に対して「機能を呼び出す」ことしかできず、中のデータがどうであるかはそもそも分からないからです (Aは最初から使われるだけの側で、Bの機能を呼んだりしません)。

そしてこの独立性のおかげで、Aの中身 (実装) はプログラムの他の部分に影響されることなく、自由に変更できます (スタックも「配列を使った実装」「単連結リストを使った実装」の2種類あることを見てきました)。これは、プログラムの改良などに威力を発揮します。

例題: 整数の集合

今回は抽象データ型の例として「0以上の整数の集合」を扱います。ヘッダファイルを示しましょう。

```
// iset.h --- set of non-negative integers.
#include <stdbool.h>
struct iset;
typedef struct iset *isetp;
isetp iset_new();                    // create empty set
void iset_free(isetp s);             // free memory
bool iset_isempty(isetp s);          // test emptiness
bool iset_isin(isetp s, int e);      // e included in s?
int iset_max(isetp s);               // return max value
void iset_addelt(isetp s, int e); // add e to s
void iset_subelt(isetp s, int e); // remove e from s
isetp iset_union(isetp s, isetp q); // set union
```

例題: 整数の集合

今回は領域を大量に使うので、使い終わったら解放するようにします。ということで操作としては、集合を作る、解放する、空集合かどうか調べる、整数を指定して含まれるかどうかを調べる、最大値 (空集合のときは 0 とします)、整数を追加する、取り除く、そして 2 つの集合から和集合を作る、まで用意しました。

ではデモとして、2 つ集合を読み込んで和集合を作って表示する、というのをやってみましょう。読み込みと出力を関数として作っているので、その分量が少し多くなります。読み込む時はまず空の終業を用意し、負の値がくるまで次々に整数を読み込んで集合に追加していき、負の値が来たらそこでやめて集合を返します。打ち出す時はまず最大を調べ、0 から最大まで順に調べて集合に入っている値だったら出力します。

例題: 整数の集合

```
// isetdemo1.c --- iset demonstration.
#include <stdio.h>
#include <stdbool.h>
#include "iset.h"

isetp readiset(void) {
  isetp s = iset_new();
  while(true) {
    int i; printf("i> "); scanf("%d", &i);
    if(i < 0) { return s; } else { iset_addelt(s, i); }
  }
}
void printiset(isetp s) {
  printf("{");
  for(int max = iset_max(s), i = 0; i <= max; ++i) {
    if(iset_isin(s, i)) { printf(" %d", i); }
```

```
  }
  printf(" }\n");
}
int main(void) {
  isetp s = readiset(); printiset(s);
  isetp t = readiset(); printiset(t);
  isetp u = iset_union(s, t); printiset(u);
  iset_free(s); iset_free(t); iset_free(u);
  return 0;
}
```

例題: 整数の集合

main本体では、2つ読み込みそれぞれ表示したあと、和集合を生成してそれも印刷する、というだけです。最後に領域を解放しています。実行例を示しましょう。

```
% gcc8 isetdemo.c iset_1.c
% ./a.out
i> 1
i> 3
i> 5
i> -1
{ 1 3 5 }
i> 2
i> 3
i> 4
i> -1
{ 2 3 4 }
{ 1 2 3 4 5 }
```

例題: 整数の集合

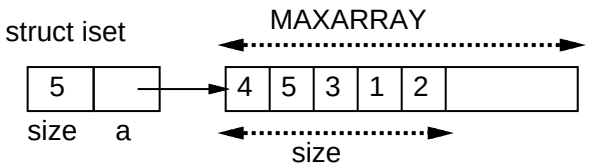

図 2: 整数の集合の実装

実装のファイルを `iset_1.c` としているのは、これからいくつも別の実装を作るからです。とりあえず、最初の実装を見てみましょう。この実装は次の方針で作ってあります (図 2)。

- 固定サイズの配列をレコードと別に持ち、そこに集合に含まれる整数を保持する。
- 整数の格納順は任意とする。

2 番目についてですが、新しい整数は末尾に追加していきますが、`iset_subelt` で要素を取り除いた時はその箇所に末尾にあった値を移してきて埋めるので、常に入れた順になっているとは限りません。ソースを示します。

例題: 整数の集合

```
// iset_1.c --- iset impl w/ unsorted array of fixed size.
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "iset.h"
#define MAXARRAY 10000
struct iset { int size, *a; };

isetp iset_new() {
  isetp s = (isetp)malloc(sizeof(struct iset));
  s->size = 0; s->a = (int*)malloc(sizeof(int)*MAXARRAY);
  return s;
}
void iset_free(isetp s) { free(s->a); free(s); }
static int isin1(isetp s, int e) {
  for(int i = 0; i < s->size; ++i) {
```

```
    if(s->a[i] == e) { return i; }
  }
  return -1;
}
bool iset_isempty(isetp s) { return s->size == 0; }
bool iset_isin(isetp s, int e) { return isin1(s, e) >= 0; }
static int max2(int a, int b) { return (a > b) ? a : b; }
int iset_max(isetp s) {
  int max = 0;
  for(int i = 0; i < s->size; ++i) { max = max2(max, s->a[i]); }
  return max;
}
void iset_addelt(isetp s, int e) {
  if(iset_isin(s, e) || s->size >= MAXARRAY-1) { return; }
  s->a[(s->size)++] = e;
}
void iset_subelt(isetp s, int e) {
```

```
  int i = isin1(s, e); if(i < 0) { return; }
  s->a[i] = s->a[--(s->size)];
}
isetp iset_union(isetp s, isetp t) {
  isetp u = iset_new();
  for(int i = 0; i < s->size; ++i) { iset_addelt(u, s->a[i]); }
  for(int i = 0; i < t->size; ++i) {
    if(!iset_isin(s, t->a[i])) { iset_addelt(u, t->a[i]); }
  }
  return u;
}
```

static指定の関数はファイルの外からは参照できず、ファイル内での下請け専用です。

例題: 整数の集合

では単体テストについて検討しましょう。個別の整数を取り出したりしてチェックするのだと使いづらそうなので、集合全体をまとめてチェックする expect_iset を作ることにします。パラメタは「集合、テストする値の最大値、含まれているべき要素の個数、含まれている各要素を昇順に並べた配列、メッセージ」となっています。含まれているべき要素を配列 a で受け取り、その中の次に見るべき位置を変数 p で覚え、i は 0〜m で変化させて行きますが、i が a[p] と等しいときは「含まれるべき値」、それ以外は「含まれるべきでない値」となります (等しいのが見つかったら p は 1 つ先に進める)。

例題: 整数の集合

```
void expect_iset(isetp s, int m, int n, int a[], char *msg) {
  bool ok = true; int p = 0;
  for(int i = 0; i <= n; ++i) {
    if(p < n && i == a[p]) {
      ++p;
      if(!iset_isin(s, i)) { printf(" NG: %d not in set.\n", i); ok = fals
    } else {
      if(iset_isin(s, i)) { printf(" NG: %d in set.\n", i); ok = false; }
    }
  }
  printf("%s %s\n", ok?"OK":"NG", msg);
}
```

これを使った単体テストの例です。aはテストデータを入れた配列ですが、expect_isetに渡す時にどこを先頭にするか、何個を指定するかを変えることで3通りに使っています。

例題: 整数の集合

```
// test_iset.c --- unit test for iset.
#include <stdbool.h>
#include <stdio.h>
#include "iset.h"
(expect_iset here)
int main(void) {
  int a[] = { 1, 3, 5, 7 };
  isetp s = iset_new();
  iset_addelt(s, 1); iset_addelt(s, 3); iset_addelt(s, 5);
  expect_iset(s, 9, 3, a, "initial: { 1 3 5 }"); iset_subelt(s, 1);
  expect_iset(s, 9, 2, a+1, "- { 1 }: { 3 5 }");
  isetp q = iset_new(); iset_addelt(s, 7); iset_addelt(s, 5);
  isetp r = iset_union(s, q);
  expect_iset(r, 9, 3, a+1, "+ { 7 5 }: { 3 5 7 }");
  iset_free(s); iset_free(q); iset_free(r);
  return 0;
```

```
}
```

動かしたようすは次の通り。

```
% gcc8 test_iset.c iset_1.c
% ./a.out
OK initial: { 1 3 5 }
OK - { 1 }: { 3 5}
OK + { 7 5 }: { 3 5 7 }
```

例題: 整数の集合

演習 1 上の例題をそのまま動かせ。打ち込む値は変えて試してみること。OKだったら、次の機能を追加してみなさい。単体テストを作成し実行すること。

a. 積集合 (intersection) の演算を作れ。

b. 差集合 (difference) の演算を作れ。

c. 排他的和集合 (exclusive or、どちらか片方のみにある値だけを集めた集合) の演算を作れ。

d. その他あるとよいと思う機能を追加してみよ。

例題: 整数の集合

演習 2 上で見た実装は、配列のサイズが`MAXARRAY`に固定になっていて、少しの値しか扱わなければ領域が無駄だし、多くの値を扱おうとすると入り切らなくなった分が無視される。これは不便なので、次の考え方に基づいて実装を修正してみよ。単体テストを作成し実行すること。

1. 構造体のフィールドに配列の現在のサイズ`limit`を追加する。
2. 最初は小さめの(たとえば大きさ10)配列から始めて、大きさを増やそうとして入り切らなくなったら(`size`が`limit`より大きくなりそうになったら)、新しい配列を`limiit+1`の大きさで割り当て、内容をコピーしてからこちらの配列を構造体に入れる(前の配列は解放する)。

演習2のように、情報隠蔽がなされていることで、内部の構造を修正しても外からの使い方は一切変化しないで済みます。これが抽象データ型の利点ということです。

データ構造の選択と計算量の関係
疑似乱数の生成

これからデータ構造を変化させて計算量を検討しつつ時間計測を行いますが、その準備として疑似乱数の扱いから見ていきましょう。C 言語の標準ライブラリでは疑似乱数の生成には`rand`を使います。この関数は`stdlib.h`で宣言されていて、引数は0個で、0から`RAND_MAX`(これも同じヘッダファイルで定義されています)までの範囲の整数を返します。

ただし、乱数の「種」で初期化しないと毎回同じ列が返ってくるので、通常は最初に「`srad(time(NULL))`」を呼び出して種を設定します。`srand`は種を設定する関数、`time`(宣言は`time.h`)は1970.1.1 0:00からの経過秒数を返します(引数として領域の番地を渡すとそこにも値を入れますが、`NULL`を渡した場合は値のみ返します)。秒数は絶えず変わって行くので、乱数の種に適切だというわけです。

`rand`の結果を、ある範囲の整数乱数として使いたければ適当な値で剰余を取ります。また$[0, 1)$の実数にするには`RAND_MAX`で実数除算します。サンプルを見てみましょう。

疑似乱数の生成

```
// randdemo.c --- demonstration of rand.
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char *argv[]) {
  int i, n = atoi(argv[1]);
  srand(time(NULL));
  for(i = 0; i < n; ++i) { printf(" %d", rand() % 1000); }
  printf("\n");
  for(i = 0; i < n; ++i) { printf(" %.3f", rand()/(double)RAND_MAX); }
  printf("\n");
  return 0;
}
```

疑似乱数の生成

まったく上で説明した通りで、まず種を設定し、指定された個数の整数乱数 (0〜999) と、実数の乱数 (区間 $[0, 1)$) を出力しています。動かしたようすは次の通り。

```
% ./a.out 10
 774 837 508 259 653 449 448 585 551 172
 0.289 0.868 0.881 0.543 0.394 0.571 0.068 0.873 0.531 0.970
```

疑似乱数の生成

演習 3 疑似乱数を使った次のようなプログラムを作れ。

a. サイコロ (実際には 0〜5 の整数乱数に 1 を足せばよい) を 6000 回振ってそれぞれの目が何回出たかを表示する実験プログラム。

b. 100 ますのすごろく (後戻り等なし、ゴールから行きすぎても OK) でサイコロを何回振ってゴールできるかを、1000 回やってみて平均回数を求める。

c. X 座標 $[0, 1)$、Y 座標 $[0, 1)$ の範囲にランダムに点を打ち、中心 0、半径 1 の円内に入った点の比率を求めて 4 倍することで π を求めてみる。何個の点を打つかは実行時に指定する。

d. 0.6 の確率で表が出る偏ったコインで、「10 回連続して表になる」までに何回トスすればできるかを、100 回やってみて平均回数を求める。

e. その他、疑似乱数を使った好きなプログラム。

所要時間の計測

次に、所要時間の計測について知っておきましょう。C ライブラリの中で時間を返すものとして、前述の time がありますが、秒単位だと時間計測には荒すぎて不向きです。

ここでは clock_gettime というものを使います。その呼び出しの第 1 パラメタは取得する時間の種類で、今回のような計測では CLOCK_REALTIME(実時間) または CLOCK_VIRTUAL(CPU 消費時間) を指定します。そして第 2 パラメタには構造体 struct timespec (上記の呼び出しや定数とともに time.h で宣言) のアドレスを渡し、そこに計測値が入って戻って来ます。この構造体の内容は次のようになっています。

```
struct timespec {
  time_t  tv_sec;  /* seconds */
  long    tv_nsec; /* and nanoseconds */
};
```

time_t は秒数を入れるのに適切な整数型を typedef したものなので、要するにどちらも適当な長さの整数で、秒以上の部分と秒未満の部分 (ナノ秒単位) を分けて扱っています。

所要時間の計測

では、3^n を遅い再帰を使って計算し、その時間を測ってみましょう。`pow3n` がその遅い再帰の関数で、次の方法で計算をしています。

$$pow3n(n) = \begin{cases} 1 & (n = 0) \\ pow3n(n-1) + pow3n(n-1) + pow3n(n-1) & (otherwise) \end{cases}$$

1 つ少ない n に進むごとに 3 回ずつ自分を呼ぶので、時間計算量は $O(3^n)$ になるはずです (そもそも 1 ずつ足して 3^n になるのだからその意味でも $O(3^n)$ です)。

所要時間の計測

```
// timedemo.c --- demonstration of mesuring time.
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "iset.h"

int pow3n(int n) {
  if(n < 1) { return 1; }
  else { return pow3n(n-1)+pow3n(n-1)+pow3n(n-1); }
}
int main(int argc, char *argv[]) {
  int i, n = atoi(argv[1]);
  struct timespec tm1, tm2;
  clock_gettime(CLOCK_REALTIME, &tm1);
  int v = pow3n(n);
  clock_gettime(CLOCK_REALTIME, &tm2);
```

```
  double dt = (tm2.tv_sec-tm1.tv_sec) + 1e-9*(tm2.tv_nsec-tm1.tv_nsec);
  printf("pow3n(%d) = %d; elapsed-time = %.4f\n", n, v, dt);
  return 0;
}
```

所要時間の計測

main の方では、2 回 clock_gettime を呼んで時刻を取得し、その間で pow3n を呼びます (この部分が時間計測の対象)。終わったら、秒単位の実数に直したいので、秒の差はそのまま、ナノ秒の差は 10^{-9} を掛けて加えたものを dt とします。最後にこれらを表示しています。では動かしてみましょう。

```
% ./a.out 4
pow3n(4) = 81; elapsed-time = 0.0000
% ./a.out 10
pow3n(10) = 59049; elapsed-time = 0.0002
% ./a.out 15
pow3n(15) = 14348907; elapsed-time = 0.0560
% ./a.out 16
pow3n(16) = 43046721; elapsed-time = 0.1745
% ./a.out 17
pow3n(17) = 129140163; elapsed-time = 0.5069
```

所要時間の計測

3^4 とか 3^{10} では速すぎですが、3^{15} で 0.056 秒となり、n をもう 1 つ増やすと 3 倍、さらにもう 1 つ増やすと 3 倍で、確かに $O(3^n)$ となっているようです (注意: 個人使用のマシンであればこのように取得する時刻の種別は `CLOCK_REALTIME` でよいが、共用マシンだと他人も CPU を使っているので `CLOCK_VIRTUAL` を使う方がよいかも)。

演習 4 n を指定して動かしたときに、例題では時間計算量が $O(3^n)$ だったが、(a) $O(2^n)$、(b) $O(n^3)$、(c) $O(n^2)$、(d) $O(n)$ などとなる関数を作成して同様に計測し、実際にそのような時間になることを確認してみなさい (どれか 1 つ以上でよい)。

2 分探索

整数の集合に話を戻しますが、先に示した実装では要素の整数は配列内に「おおむね入れた順で」並んでいました (ランダム順と言っていいでしょう)。この方法だと、ある指定した整数 e が含まれているかを調べるのに、先頭から順に調べて行く必要があるため、集合の要素数が n のとき、`iset_isin(`e`)` の時間計算量が $O(n)$ になります。

別の方法として、要素を整列して昇順に並べておくとしましょう。そのときは 2 分探索が使えます。次のコードを見てください関数 `bsearch` は、配列 `a` の添字 `i`〜`j` の範囲で、値 `e` があればその添字、無ければ `-1` を返します。配列 `a` は昇順に整列されているものとします。

2分探索

```
int bsearch(int *a, int e, int i, int j) {
  if(i > j) { return -1; }
  int k = (i + j) / 2;
  if(a[k] == e) {
    return k;
  } else if(a[k] > e) {
    return bsearch(a, e, i, k-1);
  } else {
    return bsearch(a, e, k+1, j);
  }
}
```

2 分探索

まず、`i`が`j`より大きければ範囲が空なので、`-1`を返します。そうでないときは、`i`と`j`の中間の位置`k`を計算します。`a[k]`が`e`に等しければ、ちょうど見付かったので、`k`を返します。そうでない場合は、`a[k]`と`e`の大小関係に応じて、`e`がある範囲が`i`〜`k-1`か`k+1`〜`j`のどちらかになりますから(昇順に整列されているため)、それに応じて自分自身を再帰呼び出しします。

このアルゴリズムは「繰り返し半分に分けて扱う」ことから分割統治アルゴリズムに分類できます。また、コードを見ると再帰は末尾再帰のみなので、簡単に再帰を除去してループのみのコードに変換できます。それでは、このアルゴリズムの時間計算量はどうでしょうか。長さnの配列に対して、半分ずつにしていって長さ0になれば終わります。ということは、半分にする回数は$\log_2 n$なので、時間計算量は$O(\log n)$です。

2 分探索

ただしもちろん、常に配列を整列された状態に保つためのコストが必要です。前の実装では、集合にまだ無い要素を追加するのは、最後に入れるだけだったので $O(1)$ です。しかし今度は、追加する値がちょうどいい位置になるように、後ろの方を 1 つずつずらす必要があります。平均して n 個の要素のうち半分ずらすだろうと期待できますから、$O(n)$ になりますね。また、要素を削除する場合も、前の実装では空いた場所に最後の要素を移して来るだけでしたので $O(1)$ でしたが、整列を維持する場合は空いた要素の後ろを前に詰めることになり、やはり $O(n)$ になります。

2 分探索

ただし! 上の議論は「要素の位置が分かった後」の話ですよね。そもそもランダム版では位置を探すのに $O(n)$ 掛かるので、その後が $O(1)$ でも結局全体としては $O(n)$ になります。これらを整理して表 1 に示しました (3 番目の方法は後述)。整列してあれば最大値はそもそも端っこの値を取るだけなので $O(1)$ になります。このように、機能によって得意不得意がかなり異なることが分かります。

表 1: 整数集合のランダム順版と昇順版の操作の比較

操作	ランダム	昇順	ビットマップ
`iset_isin`	$O(n)$	$O(\log n)$	$O(1)$
`iset_addelt`	$O(n)$	$O(n)$	$O(1)$
`iset_subelt`	$O(n)$	$O(n)$	$O(1)$
`iset_max`	$O(n)$	$O(1)$	$O(1)$
`iset_union`	$O(n^2)$	$O(n)$	$O(1)$

2 分探索

演習 5 2 分探索を大きな配列に対して実行して時間計測し、大きさ n に対する時間計算量が $O(\log n)$ になっているか確認しなさい。線形探索 (端から順に探す) と比較するとなおよい。(ヒント: 1 回の探索はあっという間に終わってしまうので、同じ探索を 100 回とか 1000 回ループ実行してその時間を計測するのがよい。)

演習 6 整数の集合を要素が昇順に並んでいて 2 分探索を使用できるように修正してみなさい。単体テストを作成し実行すること。

整数集合の文字配列表現

実は、時間計算量がもっと小さく簡単な方法があります。それは…配列 a を用意し、整数 i が集合に含まれていれば a[i] を 1、そうでなければ 0 にしておく、という方法です。簡単でしょう? しかも、配列のどこかをアクセスする時間は一定なので、基本的な操作は全部 $O(1)$(定数時間) です (これは次に出て来るビットマップでも同様)。

ただ、この方法だとメモリが沢山必要になります。できるだけ節約するとして、1 要素のビット数が一番小さいデータ型は 1 要素が 1 バイト (8 ビット) の文字型ですから、char の配列を使うとしても、0/1 を入れるだけなら 1 ビットで必要なところ、7 ビットの無駄な領域がついて来ます。それでも、最大の値があまり大きくないなら、十分役に立ちます。

演習 7 上記の考え方に基づく整数集合の実装を作れ。最大整数は固定でもいいが、集合に入れられる最大値に応じて大きさが変化できると使いやすい。単体テストを作成し実行すること。

整数集合のビットマップ表現

上で出て来たように、0/1を使って集合に入っているかどうかを表すのなら、ビットをそのまま使うのが自然な方法だと言えます。整数型でビット数が多いのは`unsigned long`なので(64ビット。符号つきの`long`でも同じなのですが、マイナスが出てこない方がよさそうです)、これを使って0〜63の整数集合を表現できます。この方法を、ビットの有無の「地図」で表す、という意味で集合の**ビットマップ**(bitmap)表現と言います。

64ビット中の`e`ビット目を調べるにはどうしたらいいのでしょうか。それには「`1L << e`」という式を使います。これは、64ビットの一番下のビットが1である語(整数定数の後に`L`がついていると`long`として扱います)を、左シフト演算`<<`によって`e`ビット左にずらすと、右から`e`ビット目が1の語ができます。それと集合を表す64ビットとを`&`演算(ビット毎AND)すると、集合の右から`e`ビット目が0であれば全体として0になり(図3上左)、0でなければそのビットだけが立った(0ではない)語になりますから(図3上右)、それで判別できます(図では見やすさのため8ビットにしています)。

整数集合のビットマップ表現

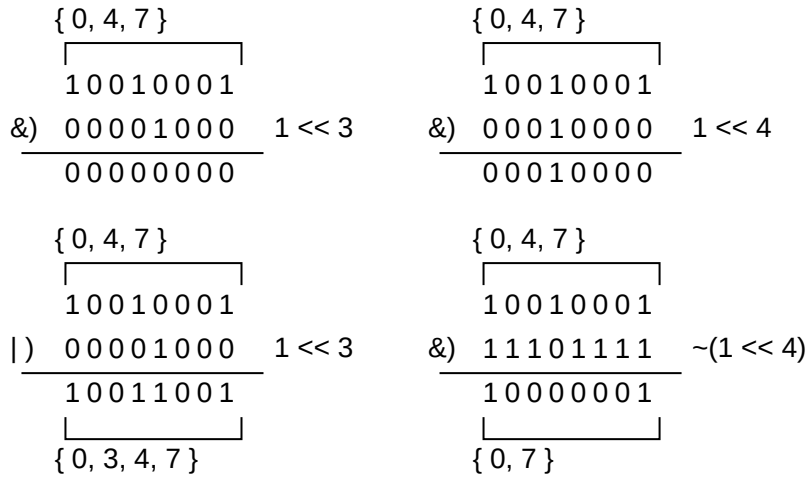

図 3: ビットマップによる整数集合の操作

整数集合のビットマップ表現

では、要素を追加したり取り除くのはどうでしょうか。追加は簡単で、|演算(ビット毎OR)を使えばそのビット位置を「1」にしたビット列が作れます(図3下左)。取り除くのはちょっとややこしいのですが、シフトの後で~(ビット毎反転)演算を使うと、そのビット位置だけが0、残りが1のビット列になりますから、それと集合のビット列の&を取れば当該ビットを「0」にしたビット列が作れます(図3下右)。あと、図にはありませんが和集合や積集合も|や&で作れます(排他的論理和演算はC言語では^です)。

それでは、この方法を使った整数集合の実装を見てみましょう。扱える値の最大は上述のように63ですが、とくにチェックはしていません。あと、&=とか|=とか見慣れない演算が出てきますが、これらはたとえば「`x &= 1`」であれば「`x = x & 1`」と同じ意味になります。

整数集合のビットマップ表現

```
// iset_4.c --- iset impl w/ unsiged long bitmap.
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "iset.h"
#define MAXARRAY 10
struct iset { unsigned long bits; };
isetp iset_new() {
  isetp s = (isetp)malloc(sizeof(struct iset));
  s->bits = 0L; return s;
}
void iset_free(isetp s) { free(s); }
bool iset_isempty(isetp s) { return s->bits == 0L; }
bool iset_isin(isetp s, int e) { return (s->bits & 1L<<e) != 0; }
int iset_max(isetp s) {
  int i;
```

```
  printf("%lx\n", s->bits);
  for(i = 63; i > 0; --i) {
    if(iset_isin(s, i)) { return i; }
  }
  return 0;
}
void iset_addelt(isetp s, int e) { s->bits |= 1L<<e; }
void iset_subelt(isetp s, int e) { s->bits &= ~(1L << e); }
isetp iset_union(isetp s, isetp t) {
  isetp u = iset_new(); u->bits = s->bits | t->bits;
  return u;
}
```

整数集合のビットマップ表現

演習 8 ビットマップによる整数集合の実装を最初に出て来た `isetdemo1.c` と組み合わせて動かし、同じに動作することを確認しなさい。OK なら、次の機能を追加してみなさい。単体テストを作成し実行すること。

a. 積集合 (intersection) の演算を作れ。

b. 差集合 (difference) の演算を作れ。

c. 排他的和集合 (exclusive or) の演算を作れ。

d. その他あるとよいと思う機能を追加してみよ。

演習 9 `unsined long` が 1 語では 63 までしか扱えないが、`unsigned long` の配列を使うことでもっと多くの値まで扱える。この場合、値 `e` は配列の `e / 64` 要素目の `e % 64` ビット目の 0/1 で表されることになる。この考え方で整数集合を実装してみなさい。最大値は固定でもよいが、必要に応じて自動的に増えるとなおよい。単体テストを作成し実行すること。

本日の課題 9a

「演習 1」～「演習 9」で動かしたプログラム 1 つを含むレポートを本日中 (授業日の 23:59 まで) に久野までに提出してください。

1. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、提出日時。名前の行は先頭に「@@@」を付けることを勧める。
2. プログラムどれか 1 つのソースと「簡単な」説明。
3. レビュー課題。提出プログラムに対する他人 (ペア以外) からの簡単な (ただしプログラムの内容に関する) コメント。
4. 以下のアンケートの回答。

   Q1. 抽象データ型の概念と整数集合の実装方法を最低 1 つ理解しましたか。

   Q2. 乱数の生成方法と時間の計測方法が分かりましたか。

   Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

次回までの課題 9b

「演習 1」～「演習 9」(ただし 9a で提出したものは除外、以後も同様) の (小) 課題から選択して 2 つ以上プログラムを作り、レポートを提出しなさい。できるだけ複数の演習から選ぶこと。レポートは次回授業前日 23:69 を期限とします。

1. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、提出日時。名前の行は先頭に「@@@」を付けることを勧める。
2. 1 つ目の課題の再掲 (どの課題をやったか分かればよい)、プログラムのソースと「丁寧な」説明、および考察 (課題をやってみて分かったこと、分析、疑問点など)。
3. 2 つ目の課題についても同様。
4. 以下のアンケートの回答。

   Q1. ビットマップを用いた整数集合の実現方法を理解しましたか。
   Q2. 必要に応じて配列サイズを拡張できるようになりましたか。
   Q3. リフレクション (今回の課題で分かったこと) ・感想・要望をどうぞ。