プログラミング通論'19 # 12 – 整数整列と探索アルゴリズム

久野 靖 (電気通信大学)

2019.4.20

今回から B 課題はありません。今回は次のことが目標となります。

- 整数のための整列アルゴリズムについて知る
- 探索の概念と主要な探索アルゴリズムについて知る

1 整数のための整列アルゴリズム

1.1 ビンソート (分配計数法)

前回までは、整列に最してキーに求められる要件は「比較により大/小/ 等しいが定まる」ことだけでした。しかし実際には、キーとして整数が使われることは多く、そのときは整数の性質を使うことで前回の $O(n \log n)$ を上回る性能を達成することもできます。今回はそのような手法を見てみます。

ビンソート (binsort) は、バケットソート (bucket sort) や分配計数法 (distribution counting) とも呼ばれ、「どのキー値がいくつあるか」数えることで整列をおこないます。すなわち、キーの最大値が m だとして、サイズ $m+1$ の配列は 0〜m の添字でアクセスできるので、初期値を 0 としてから整列データ中に「各キー値が何回現れるか」を数えることができますね。そうしたらあとは、添字の小さい方から順に「各キーを現れた個数ずつ並べていけば」整列が完了します (図 1)。

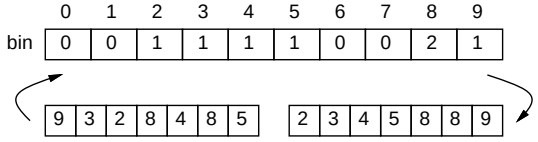

図 1: ビンソートの原理

さっそく、ビンソートのコードを見てみましょう。キーの最大値 max を受け取り、そのサイズの配列を割り当てて計数に使うようになっています (これまでのテストプログラムでは 4 桁の整数でやっていたので、最大値は 9999 ですね)。値を戻すときに「bin[i] 回繰り返す」のところを「while(--bin[i] >= 0) ...」としていますが、これも (その変数の値を壊してしまってのであれば) C 言語で便利に使える書き方です。

```
// binsort.c --- binsort with specified maximum
#include <stdlib.h>
void binsort(int n, int *a, int max) {
  int i, k = 0, *bin = (int*)malloc((max+1) * sizeof(int));
  for(i = 0; i <= max; ++i) { bin[i] = 0; }
  for(i = 0; i < n; ++i) { ++bin[a[i]]; }
  for(i = 0; i <= max; ++i) {
    while(--bin[i] >= 0) { a[k++] = i; }
  }
```

```
  free(bin);
}
```

このコードはどう見ても n に比例する仕事しかせず、$O(n)$ ということになります。ただし、キーの最大値 E が大きくなければですが。E が大きいと、そちらの影響がより大きくなるので、正確には $O(n+E)$ ということになるでしょうか。そしてあまり E が大きいと、そんな巨大な配列は取れませんということになるので、そもそも使えなくなります。

演習 1 ビンソートのプログラムについて、動作および時間を確認しなさい。そのうえで、次のことをやってみなさい。すべて単体テストすること。今回は全般に単体テストのコードは、前々回の `expect_sort_iarray` などを参考に、工夫して作る必要がある。

a. 「整数のみ」の整列なら上のコードでよいが、実際にはレコードの列を整列し、そのキーが整数である、ということが普通である。そのようなプログラムを作れ。(ヒント: 個々の `bin[i]` から始まる単連結リストにレコードを追加していくなどする。)
注意: 実装するとき、「安定な」整列にすること。少し気をつけるだけで安定にできるので、どうせなら非安定は避けた方がよい。

b. キーの最大値 E が大きい場合、その多くの値は使われない。そこで、E 要素の配列を作る代わりに D 個ずつまとめて $\frac{E}{D}$ 要素の配列を作り、個々の要素に「実際はどのキーがいくつ」という情報を記録させることで対応できる。そのようなプログラムを作れ。またその弱点を検討しなさい。

c. 上記の a と b の両方を行うプログラムを作れ。

1.2 基数ソート

基数ソート (radix sort) とは、易しく言えば「複数の整列の繰り返し」です。たとえば十進表現で 3 桁の数であれば、まず百の位が 0〜9 の順になるように並べ、その中でそれぞれ十の位が 0〜9 の順になるように並べ、さらにその中で一の位が 0〜9 の順になるように並べれば、整列が完了します。数値を何進法で (基数いくつで) 考えかに依存するため、基数ソートと呼ぶわけです。

なお、先の説明では「上の桁から」並べていましたが、「下の桁から」並べることもできます。その場合 (十進 3 桁なら)、まず一の位が 0〜9 の順になるように並べ、次に「キーが同じなら元の順番を崩さずに」十の位が 0〜9 の順になるように並べ、次に「キーが同じなら元の順番を崩さずに」百の位が 0〜9 の順になるように並べます。この「…」はつまり安定な整列をするということですね。

では「2 進表現で」「上の桁から」扱うコードを例に示します。2 進であれば各桁は「0 か 1」なので、クイックソートの時と同じようにして「ある桁が 0 の値を配列の前半に集める」ことができます。外から呼ばれる `radixsort2` は整列範囲と「どのビット」を表すマスク値 (整数の 32 ビットのうち 1 ビットだけが 1 であとが 0 の値、負の数は扱わないとしたので符号ビットの次の 30 ビット目が 1 の値から始めます) を指定して再帰手続き `rs` を呼ぶだけです。

```
// radixsort.c --- radixsort (upper->lower) from specified mask
static void swap(int *a, int i, int j) {
  int x = a[i]; a[i] = a[j]; a[j] = x;
}
static void rs(int *a, int i, int j, int mask) {
  if(i >= j || mask == 0) { return; }
  int k, s;
  for(s = k = i; k <= j; ++k) {
    if((a[k]&mask) == 0) { swap(a, s++, k); }
  }
```

```
  rs(a, i, s-1, mask/2); rs(a, s, j, mask/2);
}
void radixsort(int n, int *a) { rs(a, 0, n-1, 0x40000000); }
```

rs ですが、範囲の長さが 1 以下か、マスクが 0 (最下位まで到達) なら何もせず戻ります。次は変数 s を左端の添字とし、配列 a の範囲内の k について、a[k] とマスクのビット毎 AND 演算をおこない、0 であれば swap を使って a[k] を s の位置に置き、s を 1 増やします。そうすれば、最後には s の手前が「その桁が 0 の値」、以後が「その桁が 1 の値」となります。その後、「その桁が 0」「その桁が 1」それぞれの範囲について、さらに 1 つ下の桁での並べ替えを自分自身を再帰呼び出しして行わせます。1 つ下の桁ということは、マスクを 2 で割ればよいわけです。

演習 2 基数ソートの例題を動かし、動作と時間を確認しなさい。最大が 9999 ならマスクをもっと小さな値からにできるので、それも検討すること。その後、次のことをしてみなさい。すべて単体テストすること。

a. 2 進表現で「下の桁から」並べるプログラムを作ってみる (例題の方法で前半と後半に分けた場合は安定にはならないので注意)。

b. 単語リスト /usr/share/dict/words から長さ 10 文字の文字列を抜き出し [1]、それを基数ソートで (つまりこの場合「数字」は a〜z となる) 整列してみなさい。上の桁からやっても下の桁からやってもよい。

c. 同様だが大文字と小文字の両方が含まれる場合でやりなさい。大小順は「$a < A < b < B < \cdots$」となること。[2]

d. 同様だが長さ 1 文字以上任意でやりなさい [3]。上の桁から/下の桁からのいずれでもよい。両方試せるとなおよい (後ろが共通の語を調べるニーズもあるので下の桁からも有用)。

2 探索と探索アルゴリズム

2.1 表と探索の定式化

コンピュータ科学的には、**表** (table) という用語にはおおむね 2 通りの意味があります。1 つ目は日常生活でいう表に近いもので、次のように定式化されます。

- 表は**レコード**の集まりで、各レコードは複数の**フィールド**から成る。1 つの表の中ではすべてのレコードは同じフィールド群を持つ。

そしてもう 1 つは、上記をより抽象化したもので、次のように定式化されます。

- 表は**鍵** (key) とそれ対応する**値** (value) を格納する抽象データ型であり、鍵と値を指定して値を格納する操作と、鍵を指定して格納した値を取り出す (または格納されていなければそのことを知らせる) 操作を持つ。

データベースやデータ処理では 1 番目の意味での表が使われます。ただし、2 番目の定義であっても、その「値」が複数のフィールドを持つレコード値であってよいので、2 番目の定義のもとにデータ処理を扱うことも何ら問題なくできます。そして 2 番目にある「鍵を指定して値を格納/取り出す」操作のことを**表の探索** (table lookup) と呼び、多くのアルゴリズムが研究されています。以下ではこれらについて取り上げて行きます。

ところで、1 番目の定義の場合、探索はどうなのでしょうか。もちろん、データが多量にある場合、その効率的な処理は重要です。そこで、フィールドのうちで探索に使うものについては、2 番目の意

[1] 「egrep ^[a-z]{10}$ ファイル >出力ファイル」でできます。
[2] 「egrep ^[a-zA-Z]{10}$ ファイル >出力ファイル」
[3] 「egrep ^[a-zA-Z]+$ ファイル >出力ファイル」

味でいう探索アルゴリズムを実装するデータ構造を追加します。これを**索引** (index) と呼びます。複数のフィールドに対して索引をつけることもあります。ということで、探索アルゴリズムはいずれの場合でも重要なわけです。

2.2 線形探索

ここでもスタック等のときと同じように構造体を使って情報隠蔽で実装コードを記述することにして、まず呼び出し API を定めます。ここでは簡単のため、鍵も値も整数値であるものとします。外から呼び出す操作は最初に構造を割り当てる `itbl_new`、鍵を指定して値を格納する `itbl_put`、鍵を指定して検索し値を返す `itbl_get` の 3 つです (検索して見付からなかったときは -1 を返すことにします)。

```
// itbl.h --- int table api.
struct itbl;
typedef struct itbl *itblp;
itblp itbl_new();                       // allocate new tbl
void itbl_put(itblp t, int k, int v); // store value
int itbl_get(itblp t, int k);         // obtain value
```

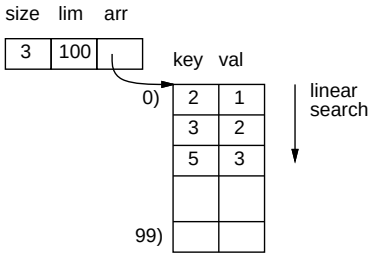

図 2: 線形探索の表

では一番シンプルな実装として、表の 1 つの項目を構造体で表し、そのフィールドとして鍵と値が格納されている、という形のものを作ります (図 2)。新しい値を追加するときは配列の末尾に入れます (配列が満杯になったら倍の大きさの配列を作って値をコピーします)。値の検索は上から順に鍵の一致する項目を探します。これを**線形探索** (linear search) と呼びます。探したり配列を拡張する作業は下請けの関数を呼ぶようにすることで、本体の関数が複雑にならないようにしています。

```
// itbl.c --- itbl impl with arry of records.
#include <stdlib.h>
#include "itbl.h"
typedef struct ent { int key, val; } * entp;
struct itbl { int size, lim; entp arr; };
itblp itbl_new() {
  itblp p = (itblp)malloc(sizeof(struct itbl));
  p->arr = (entp)malloc(100 * sizeof(struct ent));
  p->size = 0; p->lim = 100; return p;
}
static void enlarge(itblp p) {
  entp a = (entp)malloc(p->lim * 2 * sizeof(struct ent));
  for(int i = 0; i < p->size; ++i) { a[i] = p->arr[i]; }
  free(p->arr); p->arr = a; p->lim *= 2;
}
static entp lookup(itblp p, int k) {
```

```
  for(int i = 0; i < p->size; ++i) {
    if(p->arr[i].key == k) { return p->arr + i; }
  }
  return NULL;
}
void itbl_put(itblp p, int k, int v) {
  entp e = lookup(p, k);
  if(e != NULL) { e->val = v; return; }
  if(p->size + 1 >= p->lim) { enlarge(p); }
  p->arr[p->size].key = k; p->arr[p->size++].val = v;
}
int itbl_get(itblp p, int k) {
  entp e = lookup(p, k); return e == NULL ? -1 : e->val;
}
```

ではこれを使ってみる例として「素数を鍵、それが何番目の素数かを値」とするような表を作ってみます。isprime はおなじみ素数判定で、regist が指定された最大値までの範囲で「素数、何番目」を表に入れていきます (何個入れたかを値として返す)。main ではまず値を登録し、次にコマンド引数で渡されたそれぞれの整数について、表の検索結果を返します。

```
// itbldemo.c --- register primes with ranks.
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <math.h>
#include "itbl.h"

bool isprime(int n) {
  int lim = (int)sqrt(n);
  for(int i = 2; i <= lim; ++i) { if(n % i == 0) { return false; } }
  return true;
}
int regist(itblp t, int lim) {
  int r = 0;
  for(int i = 2; i <= lim; ++i) { if(isprime(i)) { itbl_put(t, i, ++r); } }
  return r;
}
int main(int argc, char *argv[]) {
  itblp t = itbl_new();
  printf("max rank = %d\n", regist(t, 10000));
  for(int i = 1; i < argc; ++i) {
    int k = atoi(argv[i]); printf("%d: %d\n", k, itbl_get(t, k));
  }
  return 0;
}
```

実行例を示します。3 は (もちろん)2 番目の素数、97 は 25 番目の素数と分かります。

```
% gcc8 itbldemo.c itbl.c -lm  ← sqrt のため -lm 指定必要
```

```
% ./a.out 3 97 100
max rank = 1229                ← 10000 までに素数は 1229 個
3: 2
97: 25
100: -1
```

では整列に引き続き、時間計測もおこなう単体テストを作ってみます。単体テストなのでデータはチェックの簡単な「鍵の値+1」を登録することにしました。データの件数はコマンド引数で指定します。このプログラムは itbl.h にプロトタイプ宣言が含まれている抽象データ型をテストするので、さまざまな実装と一緒にしてそのままテストできます。最初に指定された個数の乱数を配列に入れ、まず配列のそれぞれの値を鍵として値 (鍵+1) を登録し、次に検索してそれぞれの時間を計測します。最後に再度それぞれの鍵で検索して結果が合っているか確認します。

```
// test_itbl.c --- unit test for itable.
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "itbl.h"
void expect_itbl(int n) {
  struct timespec tm1, tm2, tm3;
  itblp t = itbl_new();
  int *a = (int*)malloc(n * sizeof(int));
  for(int i = 0; i < n; ++i) { a[i] = rand(); }
  clock_gettime(CLOCK_REALTIME, &tm1);
  for(int i = 0; i < n; ++i) { itbl_put(t, a[i], a[i]+1); }
  clock_gettime(CLOCK_REALTIME, &tm2);
  for(int i = 0; i < n; ++i) { itbl_get(t, a[i]); }
  clock_gettime(CLOCK_REALTIME, &tm3);
  int c = 0;
  for(int i = 0; i < n; ++i) {
    int v = itbl_get(t, a[i]);
    if(v == a[i]+1) { continue; }
    if(++c < 5) {
      printf(" NG: #\%d get[%d] == %d, expected %d\n", i, a[i], v, a[i]+1);
    } else if(c == 5) {
      printf("more wrong value omitted.\n");
    }
  }
  double dt1 = (tm2.tv_sec-tm1.tv_sec) + 1e-9*(tm2.tv_nsec-tm1.tv_nsec);
  double dt2 = (tm3.tv_sec-tm2.tv_sec) + 1e-9*(tm3.tv_nsec-tm2.tv_nsec);
  printf("%s size=%d tget=%.4f tput=%.4f %s\n", c==0?"OK":"NG", n, dt1, dt2);
  free(a);
}
int main(int argc, char *argv[]) {
  srand(time(NULL));
  for(int i = 1; i < argc; ++i) { expect_itbl(atoi(argv[i])); }
  return 0;
```

```
}
```

性能はどうでしょうか。線形探索では、表の項目数が n のとき、見付かるとすれば平均して半分の項目と鍵を比較する必要があり、見付からない場合は全部の項目と鍵を比較する必要があります。時間計算量としては $O(n)$ となりますね。ここでは n までの範囲の素数について鍵を登録した後、、その範囲の全部の整数について検索して時間を計測してみます (ということは計測回数の n 回を掛けて全体では $O(n^2)$ になるはずです。上のテストを動かしてみます。

```
% gcc8 test_itbl.c itbl2.c
% ./a.out 1000
OK size=1000 tget=0.0012 tput=0.0012 OK
% ./a.out 10000
OK size=10000 tget=0.1207 tput=0.1204 OK
% ./a.out 100000
OK size=100000 tget=12.0635 tput=12.0559 OK
```

確かに、登録数が 10 倍になると所要時間は登録も検索も 100 倍になっています。

2.3 2 分探索

上の表では鍵 (素数) が小さい方から並んでいるため、端から順に探さなくても、中央にある値と比べることで、求める鍵が表の前半分にあるか後ろ半分にあるかが分かります。これを繰り返すことで探す範囲を半分ずつにしていき、最後は範囲の長さが 1 になって見付かる (か、または表に入っていないことが分かる) はずです。このアルゴリズムを **2 分探索** (binary search) と呼びます (図 3)。

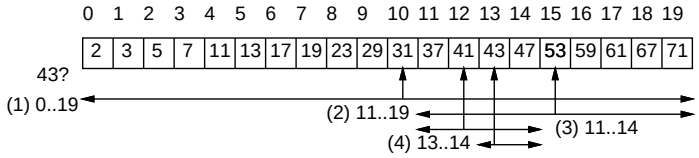

図 3: 2 分探索

2 分探索では、範囲 n を半分ずつにしていって長さ 1 になるまでに探索が終わるので、1 回の探索に掛かる時間計算量は $O(\log n)$ になります。ただし、項目が鍵の昇順に並んでいるという条件が必要になるので、その手間も考慮すべきです。表が完成したら変更しないというのであれば、これまでに学んで来た $O(n \log n)$ のアルゴリズムで整列すればよいのですが、1 個値を追加するごとに適切な位置に挿入すると考えるなら、それは挿入ソートと同じであり、値の追加にも $O(n)$ が必要となります。この様子を整理した表を示しておきます。一般には挿入回数より検索回数が多いので、挿入にコストを掛けても引き合うことは多いです。

	線形探索の表	2 分探索の表
挿入操作	$O(1)$	$O(n)$
検索操作	$O(n)$	$O(\log n)$

では実装を見てみましょう。ヘッダファイルは変更せず、実装だけ差し替えています。`itbl_new` や `enlarge` も変更しません。`lookup` を 2 分探索を用いた `lookup2` に変更しますが、こちらは範囲を指定するパラメタが必要です。また、`itbl_put` で値を追加するときは、末尾に追加したあと `shiftup` を呼んで適切な位置までその項目を移動しています。たまたま鍵の昇順に追加していけば、`shiftup` のループは実行されず、$O(1)$ で追加が終了します。

```
// itbl2.c --- itbl impl with sorted arry of records.
```

```
(途中略)
static entp lookup2(itblp p, int k, int i, int j) {
  if(i > j) { return NULL; }
  int m = (i + j) / 2;
  if(p->arr[m].key == k)     { return p->arr + m; }
  else if(p->arr[m].key > k) { return lookup2(p, k, i, m-1); }
  else                       { return lookup2(p, k, m+1, j); }
}
void shiftup(itblp p) {
  entp a = p->arr;
  for(int i = p->size-1; i > 0 && a[i-1].key > a[i].key; --i) {
    struct ent x = a[i-1]; a[i-1] = a[i]; a[i] = x;
  }
}
void itbl_put(itblp p, int k, int v) {
  entp e = lookup2(p, k, 0, p->size-1);
  if(e != NULL) { e->val = v; return; }
  if(p->size + 1 >= p->lim) { enlarge(p); }
  p->arr[p->size].key = k; p->arr[p->size++].val = v; shiftup(p);
}
int itbl_get(itblp p, int k) {
  entp e = lookup2(p, k, 0, p->size-1);
  return e == NULL ? -1 : e->val;
}
```

先と同じ単体テストをやってみましょう。次のように、登録時間はやはり $O(n^2)$ ですが、検索は線形探索よりもずっと高速です。

```
% gcc8 test_itbl.c itbl2.c
% ./a.out 1000
OK size=1000 tget=0.0014 tput=0.0002 OK
% ./a.out 10000
OK size=10000 tget=0.1193 tput=0.0018 OK
% ./a.out 100000
OK size=100000 tget=11.6172 tput=0.0274 OK
```

演習 3 線形探索と 2 分探索の例題をひととおり動かし、動作を確認しなさい。動いたら、次のことをやってみなさい。すべて単体テストすること。

a. 例題では 2 分探索が再帰関数で実現されているが、再帰を使わない版に書き換えて動作を確認しなさい。
b. より多くのフィールドを持つ構造体の配列に対して探索が行えるような API を設計し実装しなさい。
c. その他、自分独自の (特定の使い方をした時に有利な) 改良をおこない、その効果を確認しなさい。

2.4 ハッシュ表

前節まで見て来たように、2 分探索でも探索には $O(\log n)$ を要していましたが、もっと速い $O(1)$ の方法があります。どうするかというと、たとえば鍵が整数で値が $0 \sim 9999$ であれば、図 4 左のよう

に、大きさ 10000 の配列を作って「鍵 i を i 番に入れる」ようにすればよいのです (i 番の場所に i に対応する値を入れるのなら、鍵を格納するフィールドは実際はなくてもよい)。実はビンソートもまさにこれと同様のことをやっています。

ただし問題は、この方法は鍵の範囲が広いと使えない、ということです (ビンソートも同じ問題がありました)。今日のコンピュータではサイズ百万の配列は作れますが、もう 1 桁多いとだいぶ無理があります。

ここで、鍵の範囲が広い場合でも、その範囲の全部の鍵が使われることはない (むしろ範囲が広くても、扱う実際に扱うデータの数はそれよりずっと少ないのが普通)、ということに着目します。そこで、鍵の値 k を受け取る関数 $h(k)$ を定義し、その結果があまり大きくない (たとえば $0 \sim 9999$) 範囲にします。そして、その範囲の配列を作り、鍵と値を入れるわけです。

put でも get でも、同じ k を指定すれば $h(k)$ も同じになりますから、その値の場所をアクセスすればよいわけです (図 4 右)。これが**ハッシュ表** (hash table) の原理で、関数 $h(k)$ のことを**ハッシュ関数** (hash function) と呼びます。ハッシュ表も上の方法と同様、ハッシュ関数の計算、配列アクセスとも一定時間ですから、$O(1)$ で登録や検索が行えます。

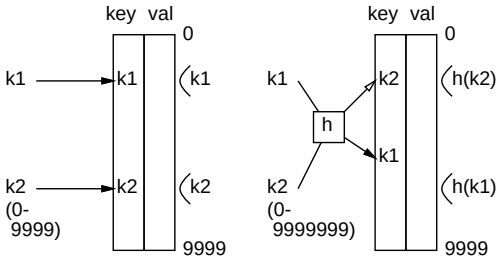

図 4: 配列の直接使用とハッシュ表

ただし今度は、広い範囲の鍵を一定範囲に「折り畳む」ため、異なる鍵 k_1、k_2 について $h(k_1) = h(k_2)$ となってしまう場合があります。これを**衝突** (collision) と呼びます。衝突を減らすためには、ハッシュ関数はできるだけ「ランダムに」値域内に値をばらまくことが望まれますが、それでも衝突は避けられません。

衝突に対応する方法は次の 2 つの方向があります。

- 連結リストなどを使い、1 つの場所に複数の値が入れられるようにする。
- 衝突が起きた場合、片方を別の場所に入れる。

ここでは余分なデータ構造がいらないという利点を持つ後者について取り上げます。「別の場所」をどのように定めるかがポイントです。たとえば「次の場所に入れる」だと、衝突が増えると「鍵が入っている場所のかたまり」ができてしまい、そこに別の鍵が衝突しやすくなります。

では、定数 d を決めて d だけ先にしてはどうでしょうか。そうしても、$h(k)$ が同じになる (衝突する) 鍵が複数あると、それが全部この「d 飛びの場所」に並ぶので、衝突が増えるとその連鎖が長くなります。

そこで、もう 1 つ別のハッシュ関数 $h2(k)$ を用意し、衝突が起きたら $d = h2(k)$ として d だけ先の場所に入れるのはどうでしょうか。別の関数なので、最初のハッシュ関数で衝突が起きても、$h2(k)$ は異なる値になるため、「次の場所」はそれぞれ異なるのですぐに入れる場所が見つかります。これを**ランダムリハッシュ**(random rehash) と呼びます。

では実装を見てみましょう。本体が構造体の配列なのはこれまでと変わりませんが、最初にすべてのエントリの鍵を -1(空いているという印) で初期化します。

```
// hashtbl.c --- itbl impl w/ random rehashing.
#include <stdlib.h>
```

```
#include "itbl.h"
typedef struct ent { int key, val; } *entp;
struct itbl { int size; entp arr; };
#define INITSIZE 999983
#define REHASH 97
itblp itbl_new() {
  itblp p = (itblp)malloc(sizeof(struct itbl));
  p->arr = (entp)malloc(INITSIZE * sizeof(struct ent));
  p->size = INITSIZE;
  for(int i = 0; i < p->size; ++i) { p->arr[i].key = -1; }
  return p;
}
```

ではハッシュ関数から見ていきます。h1 は単に表のサイズで剰余を取るだけです。後で示す理由により、表のサイズは素数でなければいけません。h2 は適当な素数で剰余を取ってから 3 を足しています。この 3 を足す理由ですが、運が悪くて h2 の結果が 0 になると「次の場所」に行けなくなるため、正の数になるようにしています。

```
static int h1(itblp p, int n) { return n % p->size; }
static int h2(int n) { return n % REHASH + 3; }
void itbl_put(itblp p, int k, int v) {
  int i = h1(p, k), d = h2(k), c = 0;
  while(p->arr[i].key != k && p->arr[i].key != -1) {
    if(++c > p->size) { return; }
    i = (i + d) % p->size;
  }
  p->arr[i].key = k; p->arr[i].val = v;
}
int itbl_get(itblp p, int k) {
  int i = h1(p, k), d = h2(k), c = 0;
  while(p->arr[i].key != k) {
    if(++c > p->size || p->arr[i].key == -1) { return -1; }
    i = (i + d) % p->size;
  }
  return p->arr[i].val;
}
```

次は、put から見てみましょう。まず h1 で場所を計算し、そこが「そのキーの場所であるか、空いている」ならすぐ入れればよいですが、そうでなければ次の場所へ行くので「キーが一致せず -1 でもない間繰り返す」ループになっています。ループの中でカウンタを増やしてチェックしていますが、これは表が満杯のときはいくら探しても一致も空きもないので、表のサイズ回やってだめならあきらめるためです。しかし、d 飛びに見ていったら空きがあっても元の場所に戻って以後同じ場所を繰り返し見るだけにならないでしょうか? それは、表のサイズを素数にしておけば d との最大公約数が 1 なので全部の場所を見終わるまでは元の位置に戻ることがな、というふうにして防ぐわけです。get の方も基本的に同様ですが、こちらは -1 があったら「無い」と分かるという点が違っています。

では、計測をしてみましょう。確かに $O(1)$ な感じです (しかも速い)。ただし、最後の 1000000 は少し遅くなっています。それは下の演習で。

```
% gcc8 test_itbl.c hashtbl.c
```

```
% ./a.out 1000
OK size=1000 tget=0.0000 tput=0.0000 OK
% ./a.out 10000
OK size=10000 tget=0.0003 tput=0.0003 OK
% ./a.out 100000
OK size=100000 tget=0.0037 tput=0.0030 OK
% ./a.out 1000000
OK size=1000000 tget=0.2262 tput=0.2108 OK
```

演習 4 ハッシュ表のコードを動かし、動作を確認しなさい。動いたら、次のことをやってみなさい。

a. 例題のコードでは「削除」の機能が無い。鍵を削除できるようにしてみなさい。(ただ鍵を -1 に戻すだけではまずい。その理由も検討し、対策を考え実装すること。)

b. 表のサイズが素数でないと表にあきがあるのに登録ができなくなることがある。その現象を確認しなさい。

c. ランダムリハッシュ法では、表が満杯に近付くほど登録も検索も遅くなる。その現象を確認しなさい。

d. 上記の問題を解決するには、表の充填率が一定値 (8 割とか) を超えたら表のサイズを大きくして作り直す方法がある。この方法を実装し、動作を確認しなさい。

本日の課題 12A

「演習 1」〜「演習 4」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) に提出してください。

1. sol または CED 環境で「`/home3/staff/ka002689/prog19upload 12a` ファイル名」で以下の内容を提出。
2. 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。
3. プログラムどれか 1 つのソースと「簡単な」説明。
4. 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。
5. 以下のアンケートの回答。

   Q1. 整数の性質を利用した整列方法について分かりましたか。

   Q2. 線形探索、2 分探索、ハッシュ表について理解しましたか。

   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。