ハッシュ法の実装と応用

倪　　永茂

はじめに

ハッシュ法（Hashing）はデータの登録、探索、および削除を高速にする技法（レヴィス 1987、セッジウィック 1992）のひとつで、アルゴリズムの効率向上に欠かせない研究分野のひとつである。最適なハッシュ法は、処理対象であるデータの種類や、データ全体の大きさ、ハードウェアの制限等さまざまな要因に依存して決まる。一部の言語にはハッシュ法が実装されているが、改善余地が多く残されている。

本文でいうハッシュ法はハッシュテーブル（Hash Table）とハッシュ関数（Hash Function）によって構成される。ハッシュテーブルは多くの場合、データを格納するための一次元配列を利用する。ハッシュテーブル上の格納する位置(インデックス)はハッシュ関数の算出によって決まる。つまり、処理しようとするデータはハッシュ関数の入力になり、関数内部の計算によってハッシュ関数の値（関数の出力、以下ではハッシュ値という）が定まる。関数の入力とハッシュ値が一対一（数学でいう単射）関係にあるのがハッシュ関数の特徴である。言い換えると、A, B を相異なる集合、ハッシュ関数 f を写像 $f\colon A \to B$ だとして、A に属する任意の 2 つの要素 a_1, a_2 に対し、$a_1 \neq a_2$ ならば、$f(a_1) \neq f(a_2)$ である。さて、ハッシュ値によって、ハッシュテーブル上の位置（インデックス）が唯一決まることにすれば、データ登録はその位置にフラグを立てることで実現するし、データ探索はその位置にフラグの状態を確認することになる。また、データ削除はその位置のフラグを下ろすことで実現する。登録、探索、削除の処理時間はハッシュ値の計算時間にのみ依存するので、ハッシュ値の計算時間が $O(1)$ であれば、ハッシュ法の効率は $O(1)$ になり、効率の最大化を実現するわけである。

上記のハッシュ関数という用語は入出力が単射という特徴で暗号学でも使われている。ただ、暗号学では処理効率の追求以上に、一方向性が最も重要視されている。つまり、与えられた入力によってハッシュ値は容易に計算できるが、ハッシュ値から入力を推定することは不可能(あるいは極めて困難)という特徴も有しないといけない。しかし、一方向性は本文でいうハッシュ関数に必要ということはない。アルゴリズムによって結果的にできたハッシュ関数が一方向性を有するが、それを目的としていたわけではないことに留意しよう。

しばしば、不適当なハッシュ関数によって、異なる入力にも関わらず、ハッシュ値が同じになってしまうことがある。それをハッシュ衝突（Hash collision）という。ほとんどの応用では、ハッシュ衝突は避けるべきことのひとつである。

そして、ハッシュテーブルを実際に使用した割合を占有率というが、それを高めることはメモリ等のリソースの有効利用につながるので、望まれることのひとつである。ただし、多くのハッシュ法では、占有率が高いとハッシュ衝突が起き、ハッシュ衝突をなくすために本来必要としない処理を行うので、ハッシュ法の効率を下げることになってしまう。

以上をまとめると、よいハッシュ法とは、

① ハッシュ値の計算は極めて効率的。
② ハッシュ衝突は起きない。あるいは、起きた場合に効率よくそれを解消できる。
③ 占有率が高い。

という 3 つの要件を満たすことである。

以下では、これら 3 つの要件を満たすハッシュ法を処理対象であるデータの種類に応じて、その実装･評価、および応用について考察していく。

Ⅰ モジュロ演算法によるハッシュ法の実装と評価

ここでは、ハッシュ関数の入力をキーと呼ぶことにする。最も早く実用化され、いまでも最も多く利用されているのはモジュロ演算によるハッシュ法で

ある。

1 原理と評価

キー：任意の整数 x（値の範囲 $xmin$ ～ $xmax$）

ハッシュ関数：$y = (x + a) \bmod m$

ハッシュテーブル：大きさが m の一次元配列

上記の mod はモジュロ演算を表し、m で除算したときの余りを意味する。ハッシュ値をハッシュテーブルのインデックスに使うので、ハッシュ値 y を 0 ～ m-1 にするのは一般的である。また、オフセット a を取り組むことで、x から y へのマッピングを可能にするのである。つまり、a の値を $-xmin$ にすれば都合がよい。

ハッシュ衝突が起きないために、また、占有率を高めるために、m の値を慎重に決める必要がある。m の値の設定に関して、2 つの意見が対立している。すなわち、ひとつは従来の教科書（たとえば、Cormen et al. 2001、レヴィス 1987）が説明しているように、m の値を（$xmax - xmin + 1$）以上の最小の素数にする意見と、もうひとつは m の値を 2 のべき乗にすべきだという意見、の 2 つである。

2 のべき乗にする意見の根拠は m の値を 2 のべき乗にすれば、モジュロ演算はビット AND（ビット論理積）演算で実現するので、大幅な効率向上が期待される（倪 2018）。つまり、ハッシュ関数の計算 は$y = (x + a)$ & $(m - 1)$ に変わるからである。& 記号は C 言語のビット AND 演算を意味する。

表 1 は 2 のべき乗と、2 のべき乗以上の最も小さい素数の一部である。

では、モジュロ演算法について評価しよう。ハッシュ値の計算効率は $O(1)$ であることは自明であろう。ハッシュ衝突が起きないこともモジュロ演算の性質から容易に証明できる。占有率は 100% ではないにしても、最小限とする大きさを僅かに超える素数を選べば、容認できるレベルだと思われる。

このように、キーの範囲は数千万以内であれば、モジュロ演算法は極めてよいハッシュ法といえよう。こういった、キーに対して一意的なハッシュ値が得られ、ハッシュ衝突も発生しないハッシュ関数を完全ハッシュ関数とも呼ばれる。

表 1　2 のべき乗とそれ以上の最小素数

n	2^n	素数
12	4096	4099
13	8192	8209
14	16384	16411
15	32768	32771
16	65536	65537
17	131072	131101
18	262144	262147
19	524288	524309
20	1048576	1048583
21	2097152	2097169
22	4194304	4194319
23	8388608	8388617
24	16777216	16777259
25	33554432	33554467
26	67108864	67108879
27	134217728	134217757

しかし、モジュロ演算法がこのままではうまく機能しない場合、すなわち、キーの大きさが数千万のレベルを大きく超えた場合に起きるハッシュ衝突への対策が求められる。

2 ハッシュ衝突への対策

モジュロ演算法では、ハッシュ衝突は異なるキーがハッシュ関数によって、同一ハッシュ値に算出された場合に起きる。たとえば、相違なるキー x_1 と x_2 が、$x_2 = x_1 + m$ のときに、それぞれのハッシュ値 y_1 と y_2 が $y_1 = y_2$ になる。

そこで、キーの値を x とすると、相違なる x の数が数百万以内という条件のもとで、ハッシュ衝突への対策として、以下に示すオープンアドレス法を用いる。

ハッシュ衝突への対策：$y = (y + p) \bmod m$

m が素数であれば、p の値は m の倍数以外のどの値でもよい。

3 C 言語による実装

ハッシュ衝突を考慮した実装は C 言語を用いて以下の通り記述する。

```
#define HASHSIZ 67108864 // 67108879
typedef struct { int key, a; } DATA;
typedef struct {DATA data; char f;}
HASH;
HASH hash[HASHSIZ];
HASH *hashend = hash + HASHSIZ;
```

```
// つぎの 2 行はハッシュ関数：2 のべき乗と素数
#define HASHFUNC(x) (hash+(x)&(HASHSIZ-1))
//#define HASHFUNC (x) (hash+ (x) %HASHSIZ)
void init_hash(void) { // 初期処理
  memset(hash, 0, sizeof(hash));
  hashend = hash + HASHSIZ;
}
int insert(DATA data) { // データ登録
  HASH *p = HASHFUNC(data.key);
  while (p->f) {
    if (p->data.key == data.key) {
      if (p->f != 1) break;
      return 0; // 重複登録
    }
    // つぎの 1 行はハッシュ衝突対策
    if (++p == hashend) p = hash;
  }
  p->data = data, p->f = 1;
  return 1;
}
int lookup(DATA data) { // データ探索
  HASH *p = HASHFUNC(data.key);
  while (p->f) {
    if (p->data.key == data.key)
      return p->f == 1; // 登録済
    if (++p == hashend) p = hash;
  }
  return 0;  // 登録無
}
int delete(DATA data) { // データ削除
  HASH *p = HASHFUNC(data.key);
  while (p->f) {
    if (p->data.key == data.key) {
      if (p->f != 1) break;
      p->f = 2;
      return 1; // 正常終了
    }
    if (++p == hashend) p = hash;
  }
  return 0;  // 登録無
}
```

記述内容について簡単に説明する。HASHSIZ はハッシュテーブルの大きさであり、以下の検証実験では 2 のべき乗か、素数か（コメントアウトした行）の比較を行う。構造体 DATA は処理対象であるデータを表すもので、必要に応じて構造体のメンバーを増やせばよい。構造体 HASH はハッシュテーブルの要素であり、処理対象であるデータを格納する部分を確保することに加えて、データの格納状態を示すフラグ f を取り入れた。f の値はそれぞれ、0 が未格納、1 が格納済、2 が削除済を表す。f の値として、0、1 以外に、2 を使ったのは、ハッシュ衝突対策としてのオープンアドレス法を機能させるための工夫である。

insert () はデータ登録用の関数であり、データが重複登録しようとすると、関数のリターン値が 0 になる。正常終了時の関数リターン値は 1 である。

lookup () はデータ探索用の関数であり、データが登録されていれば関数のリターン値が 1、登録されていなければリターン値は 0 になる。

delete () は登録済データを削除するための関数であり、そもそもデータが登録されていなければ関数のリターン値が 0 である。正常に削除処理が終了した場合にリターン値は 1 になる。

3 つの関数はいずれも処理が正常に終了したことを示す関数のリターン値として 1 に統一してある。

なお、関数 init_hash () はハッシュテーブルの初期設定を行うもので、ハッシュ法を利用する際に呼び出すことにしよう。また、ハッシュ衝突対策のための正の整数として p =1 にしてある。

4 検証比較実験

実験では、データの個数を 100 万から 1 千万まで、キーの値の範囲を符号なし 31 ビット（0 ～ 2^{31}–1）として、それぞれのデータを擬似乱数アルゴリズム Xorshift（Marsaglia 2003）を利用して生成する。ただし、重複するキーについては取り除く。

実験プログラムの流れはつぎの通りである。生成されたすべてのデータをひとつひとつハッシュテーブルに登録した後、これらのデータがデータ探索によってすべて登録されたことを確認する。最後に、これらのデータをひとつひとつハッシュテーブルから削除する。つまり、データ生成、データ登録、データ探索、データ削除の順番で処理を行う。

表 2（そのグラフ図 1 ～図 3）では処理時間、ハッシュ衝突の発生回数、占有率を実測した。

表２　モジュロ演算法に関する処理時間、ハッシュ衝突回数と占有率の実測結果

登録・検索・削除データ数	ハッシュテーブルの大きさ					
	67108864（2 のべき乗）			67108879（素数）		
	処理時間（sec）	ハッシュ衝突数	占有率（%）	処理時間（sec）	ハッシュ衝突数	占有率（%）
1000000	0.173	7124	1.49	0.189	7358	1.49
2000000	0.259	29558	2.98	0.283	30093	2.98
3000000	0.352	67804	4.47	0.383	68489	4.47
4000000	0.444	123026	5.96	0.486	123448	5.96
5000000	0.541	195207	7.45	0.602	195557	7.45
6000000	0.634	286158	8.94	0.697	285719	8.94
7000000	0.737	395714	10.43	0.802	395604	10.43
8000000	0.823	525585	11.92	0.905	525822	11.92
9000000	0.919	676533	13.41	1.011	675714	13.41
10000000	1.016	849706	14.90	1.125	848207	14.90

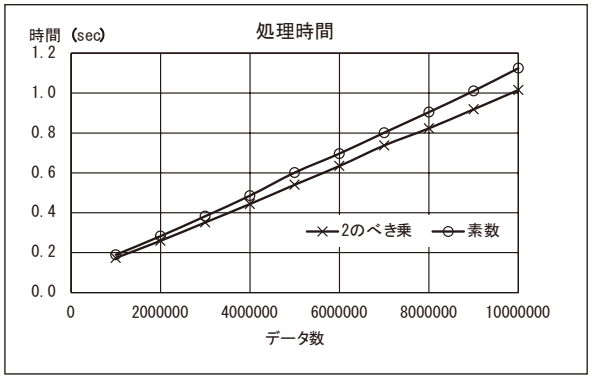

図１　データ数と処理時間との関係

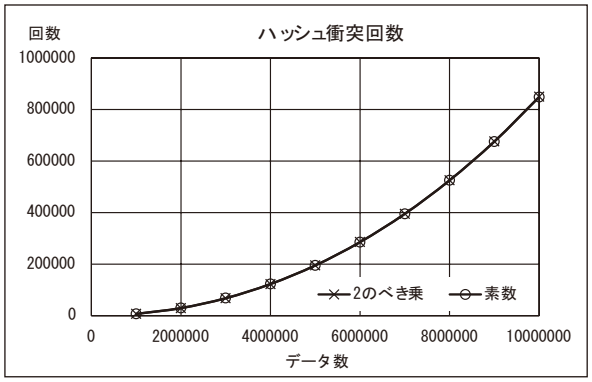

図２　データ数とハッシュ衝突回数との関係

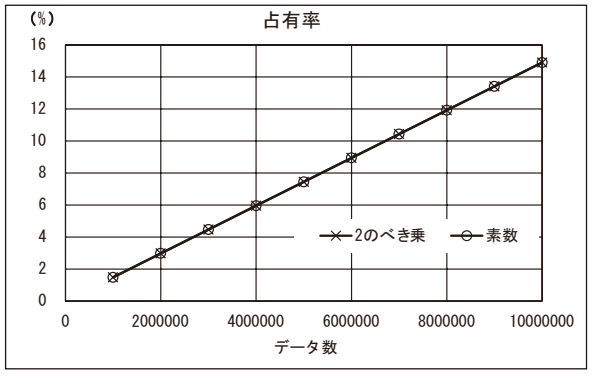

図３　データ数と占有率との関係

実験において、ハッシュテーブルの大きさをそれぞれ、2 のべき乗（67108864 = 2^{26}）と素数 67108879 にした。実験結果から、擬似乱数法によって生成されるデータに対して、ハッシュテーブルの大きさを 2 のべき乗にしたほうが処理効率が 10% 前後あがり、ハッシュ衝突回数や占有率についての差はほとんどなかったことがわかった。

Ⅱ 文字列のハッシュ化

数十種のアルファベットで作られた文字列は文字種の少なさから、長い文字列でも工夫次第で、ハッシュ値にすることが可能である。

文字列を S、特定の文字列（照合パターン、または探索パターンという）を T とすると、S に T が含まれているかどうかの判定、さらに含まれた場合にそのインデックス（複数含まれた場合に、それぞれのインデックス）の算出はハッシュ値を使えば効率的にできる。なぜなら、2 つの文字列が一致するかどうかの照合は、そのハッシュ値の比較で済むので、大幅な効率向上が得られるからである。

1 ローリングハッシュ

長さ n の文字列 S を次のような計算で得られたハッシュ値をローリングハッシュ（Rolling Hash）という。

$$H(0,n) = \sum_{i=1}^{n-1} S_i \times B^{n-i-1} \bmod M$$

ただし、B、M は定数であり、また B は基数といい、多進数における基数と同じ役割を演じる。定義から、つぎの性質が直ちに得られる。長さ l, r の文字列に対応するローリングハッシュ $H(0, l)$、$H(0, r)$ がそれぞれ与えられたとき、部分文字列 $[l, r - 1)$（インデックス l から r - 1 までで、文字列の先頭インデックスが 0）のローリングハッシュは

$$H(l,r) = (H(0,r) - H(0,l) \times B^{r-l}) \bmod M$$

になる。あるいは、元の文字列 X に、新たに文字列 Y を後ろにつないで、得られる文字列 $X+Y$ のローリングハッシュは

$$H(0,|X|+|Y|) = (H(0,|X|) \times B^{|Y|} + H(0,|Y|)) \bmod M$$

である。ただし、$|Z|$ は文字列 Z の長さを表す。

2 ペア（B, M）の選定

ローリングハッシュにおいて、ハッシュ衝突対策が大きな課題になる。文字列の長さが何万、何十万（文字）ということは珍しくないので、照合対象である文字列をすべてハッシュテーブルに登録することは一般的にできないからである。その結果、ローリングハッシュが一致すれば、無条件に元の文字列が一致すると判定するか、探索パターンと文字列をもう一度文字ごとに照合することを行う。

では、ローリングハッシュにおいて、（B, M）の選定をどのようにすればよいか。

M について、大きな素数をランダムに選ぶのがよいとされている。ただし、2^{32} を超える素数を使うときに、計算のオーバーフローに気をつけるべきで、必要であれば 128 ビット以上の整数が計算できる言語環境や多倍長整数ライブラリーを利用しよう。また、64 ビット言語環境（C、C++、Java 等）下では M として、符号なしの 64 ビット整数を抑える 2^{64} を選ぶのもよい。オーバーフローになる心配はないし、モジュロ演算も省略でき、計算の効率が多少向上する。

また、基数 B の選定について原始根を選ぶのがよいだろう。原始根とは、素数 M に対し、B mod M, B^2 mod M, B^3 mod M, …, B^{M-1} mod M の値がすべて異なる B のことをいう。表 3 ではいくつかの大きな素数 M に対する原始根の一部を示した。解決しようとする課題における文字種(10 進数列のみか、アルファベット大文字のみか、小文字のみか、大小文字の両方か、アルファベットに加えて数字も記号も処理対象になるか等）に応じて適切な原始根を選べばよい。たとえば、10 進数列については 10 より大きい原始根、アルファベット 52 文字については 52 より大きい原始根を選ぼう。

表 3　素数に対する原始根の一部

素数		対応する原始根の一部
1000000007	=10^9 +7	5, 13, 37, 53, 107, 139, 263
1000000009	=10^9 +9	13, 29, 113, 139, 263
1073741827	=2^30 +3	7, 13, 31, 59, 131, 269
2147483647	=2^31 -1	7, 11, 31, 53, 103, 131, 263
4294967291	=2^32 -5	19, 29, 71, 97, 101, 103, 127, 269

いずれにしても、ペア数を複数用意すれば、誤判定の確率が下がる。2 ペアがあれば、誤判定の確率が 0.01%（1 万回の判定で 1 回の間違い）以下になることは検証実験によって確認した。

3 C 言語による実装

ローリングハッシュについて、C 言語を用いて以下の通り実装する。

```
typedef long long ll;
unsigned B, M; //（B, M）ペアを選定しておく
ll pat_hash（char *pat, int plen）{
  ll phash = 0;
  for（int i = 0; i < plen; ++i）
    phash =（phash*B + pat[i]）% M;
  return phash;
}
int text_idx（char *text, int tlen,
          int plen, ll phash, int *idx）
{
  int i, k = 0;
  ll thash = 0, BM = 1;
  if（tlen < plen）return 0;
  for（i = 0; i < plen; ++i）{
    thash =（thash*B + text[i]）% M;
    BM = BM*B % M;
  }
  if（thash==phash）idx[k++] = i-plen;
  while（i < tlen）{
    thash =（thash*B-text[i-plen]*BM
          +text[i]）% M;
    if（thash < 0）thash += M;
    ++i;
    if（thash==phash）idx[k++]=i-plen;
  }
  return k;
}
```

関数を利用するまえに、（B, M）ペアを先に選定しておく必要がある。上記の実装では、文字種に関する仮定はなく、シングルバイト文字であれば、B は 256 以上の原始根がよいだろう。

関数 pat_hash () は引数が 2 つ、すなわち照合パターン T の文字列と長さである。関数のリターン値として、照合パターンのハッシュ値が得られる。処理効率は T の長さに比例し、$O(|T|)$ となる。

関数 text_idx () は引数が 5 つ、すなわち照合対象である文字列 S と長さ、照合パターン T の長さ、T のハッシュ値、および、T が S に含まれる場合に S におけるインデックスに関するリスト（配列）で

ある。インデックスリスト `idx[]` を得ることが関数 `text_idx`()の目的である。関数のリターン値として、インデックスリストの長さが得られる。リターン値が 0 であれば、T が含まれていないことを意味する。処理効率は S の長さに比例し、$O(|S|)$ となる。

なお、繰り返しになるが、誤判定を避けるためには、複数の（B, M）ペアを使うべきであろう。それぞれのペアで得られたインデックスリストが一致することを最終確認する。

よって、全体の処理効率は $O(k*(|S| + |T|)$ である。ただし、k は（B, M）のペア数を表す。

4 正規表現による探索への対応

正規表現とは、探索パターンにおいて、任意の文字または不特定長さの文字列を、形式的な記述（メタ文字という）で表現する方法である。任意の一文字を意味するメタ文字「?」、任意の長さの文字列を意味するメタ文字「*」等がよく知られている。

ここでは、これら 2 つのメタ文字に対し、それぞれローリングハッシュを応用してみる。

（1）メタ文字「?」（任意の 1 文字）

探索パターン T のローリングハッシュ:メタ文字「?」（そのインデックスを i とする）に対応する文字コードを 0 にする。

文字列 S のローリングハッシュ：インデックス i に対応する文字コードをローリングハッシュから以下のように削除し、比較すればよい。

$$(H(0,n) - S_i \times B^{n-i-1}) \bmod M$$

（2）メタ文字「*」（任意長の文字列）

メタ文字「*」直前までの探索パターンと、メタ文字「*」以降の探索パターンに 2 つに分割して、前半部分に対する照合と、後半部分に対する照合をそれぞれ行う。無論、それぞれの探索パターンのなかにさらにメタ文字「*」が含まれていれば、再帰的に処理すべきである。

終わりに

ハッシュ法はデータの登録、探索、および削除を $O(1)$ で行い、理論上最も高速なアルゴリズムのひとつである。そのため、理論的にも実践的にも与えられた課題に最適なハッシュ法が数多くいままで提案されてきた。

本文は処理対象であるデータに関するキーを数値と文字に分けて、それぞれのハッシュ法およびその実装を考察した。数値に対してはモジュロ演算法を説明し、処理効率、ハッシュ衝突回数、占有率という観点で、ハッシュテーブルの長さを 2 のべき乗と素数にした際の比較実験を行った。文字対してはローリングハッシュを説明し、ハッシュ衝突の可能性を減らす対策を説明した。さらに、正規表現に対する対応法に言及した。

ハッシュ法は古くて新しい研究テーマである。ハードウェアの進化や新しい課題の出現によって最適なハッシュ法が常に変わるからである。

参考文献

セッジウィック, R（野下浩平他訳）（1982）、『アルゴリズム第 2 巻＝探索・文字列・計算幾何』近代科学社

倪 永茂（2018）、「C 言語プログラミング実践教育におけるビット演算」宇都宮大学国際学部研究論集（46）、87-94

レヴィス , T. G、スミス , M. Z（浦昭二他訳）（1987）、『データ構造』培風館

Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L. and Stein, Clifford（2001）, *Introduction to Algorithms, Second edition*, The MIT Press.

Marsaglia, George（2003）, *Xorshift RNGs*, Journal of Statistical Software Vol. 8（14）pp. 1-6.

Implementation and Applications of Hashing

NI Yongmao

Abstract

The hashing is composed of a hash table and a hash function. The hashing is one of the fastest algorithms in theory, which uses $O(1)$ to insert, fetch, or delete the data and related information.

In this article, the keys related to the data were divided into numerical values and string, and each hashing and its implementation were considered. For numerical values, we explained the modulo operation method, and conducted a comparative experiment when the length of the hash table was made a power of 2 and a prime number in terms of processing efficiency, possibility of hash collisions, and occupancy ratio. For string, we explained rolling hashes and indicated how to reduce the possibility of hash collisions. Furthermore, we mentioned how to deal with regular expressions.

（2019 年 10 月 30 日受理）