ハッシュ法

比較による探索に限界が

- 2分探索法は線形探索法より効率よいが、それでも計算量が$O(\log_2 n)$である
- 線形探索も2分探索もすべてデータの値の比較に基づくものであった。結局、比較だけをもとにする探索法では$O(\log_2 n)$が計算量の限界である

格段に高速化できる方法はある！

- データのとりうる値が1から100の整数に限られているとしよう。この場合、もっとも高速な探索法は1～100の整数を添え字とする配列を用意するやり方である
- 探索は、与えられた値を添え字として、この配列の要素を調べるだけで済む
- 探索の計算量はO(1)である

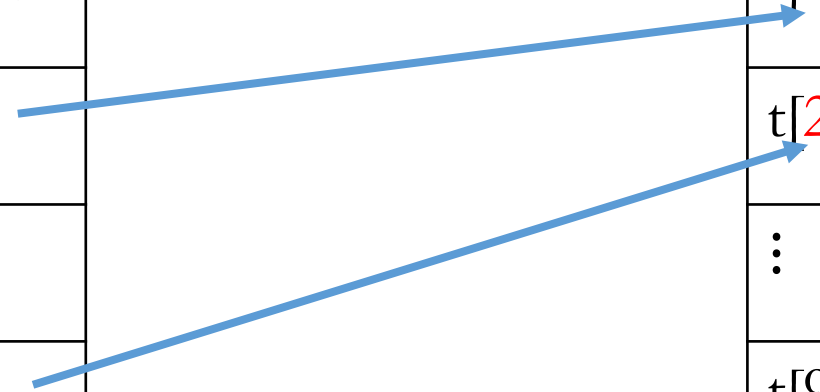
例

配列a

a[0]	50
a[1]	20
a[2]	1
⋮	
a[98]	2
a[99]	100

配列t

t[0]	
t[1]	1
t[2]	2
⋮	
t[99]	99
t[100]	100



問題点

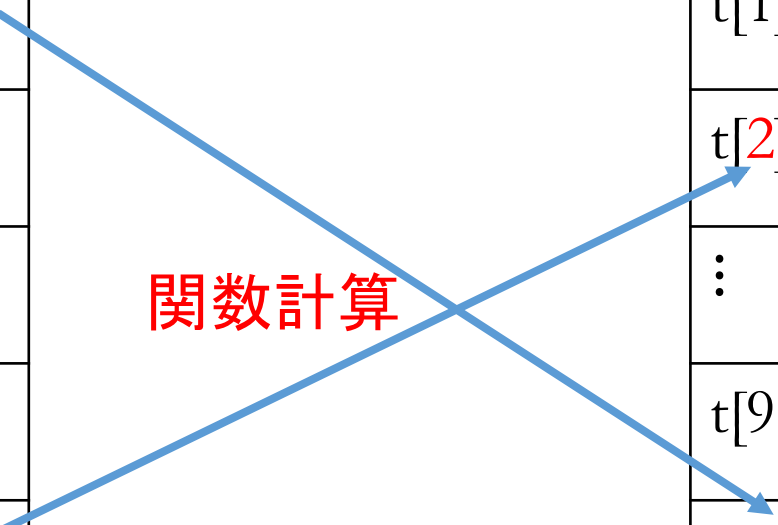
- データの値が限られた範囲の整数の場合にしか使えない
- 整数であってもデータの値が連続していないとき、配列のサイズが無駄に大きい

- 方法を少し手直しすれば、一般的な探索法とすることができる
- これがハッシュ法

例

配列a

a[0]	50
a[1]	20
a[2]	1
⋮	
a[98]	2
a[99]	100



配列t

t[0]	100
t[1]	4
t[2]	100
⋮	
t[99]	60
t[100]	20

ハッシュ法

- ハッシュ法では、データの値を引数として、ある関数の値を計算する。そして、その関数値を添え字（要素番号）として新しい配列に格納・参照する
- ここで使う関数をハッシュ関数、ハッシュ関数で求めた値をハッシュ値、新しい配列をハッシュ表と呼ぶ
- ハッシュ値を求めることをハッシュング（hashing）という
- ハッシュ関数の計算も配列の参照も、手間はnに無関係である。つまり、探索はO(1)の計算量である

整数の探索

- 例：a[]={55, 43, 60, 1, 6}
- ハッシュ関数：hash(x)=x mod m
  - m: ハッシュ表のサイズ。つまり、ハッシュ表用の配列の添え字は0からm-1の範囲内である。ここでm=11とする

hash(55)=55%11=0 → 55 → t[0]

hash(43)=10 → 43 → t[10]

hash(60)=5 → 60 → t[5]

hash(1)=1 → 1 → t[1]

hash(6)=6 → 6 → t[6]

整数の探索

- 以下のハッシュ表が作成される

t[0]	55
t[1]	1
⋮	
t[5]	60
t[6]	6
⋮	
t[10]	43

例えばx=43を探索する場合、
hash(43)=10なので、直接t[10]を参照すればよい
つまり、x=t[10]かどうかをチェックすればよい

文字列の探索

- 例：s[][64]={“TANAKA”, “NAKAYAMA”, “TAKAJIMA”, “SUZUKI”, “YAMAMOTO”}
- ハッシュ関数：$\mathrm{hash}(s) = \sum_{i=0}^{k-1} c^i \times \mathrm{code}(s[i]) \bmod m$
  - k: 文字列内の文字数
  - i: 文字列内の何番目の文字
  - code(s[i]): 文字s[i]の文字コード
  - c: 定数。ここで$c = 2$とする
  - m: ハッシュ表サイズ。ここでm=101とする

アスキーであれば

A-Z: 65-90

a-z: 97-122

0-9: 48-57

- c^iで文字の位置情報を与える。“ee”のような文字列の場合、2つの‘e’のハッシュ値が異なる。実際の実装時、位置情報を無視した方法（つまり$c = 1$）も取られている

文字列の探索

- たとえば

  hash(“SUZUKI”)=(2^0*83+2^1*85+2^2*90+2^3*85+2^4*75+2^5*73)%101=64
  → “SUZUKI” → t[64]

  ⋮

- このようにハッシュ表が作成される
- たとえば“SUZUKI”の探索は、hash(“SUZUKI”)=64をもとめ、t[64]を直接参照すればよい

ハッシュ関数

- ハッシュ法においてはよいハッシュ関数を用いるのが重要
- データが整数の場合は、h(x)=x mod mとするのが大低の場合よい。あるいはh(x)=(a*x+b) mod m）。ただし、一般的にa, b, mは素数を取るのがよい
- データが文字列の場合は

$$\mathrm{hash}(s) = \sum_{i=0}^{k-1} c^i \times \mathrm{code}(s[i]) \bmod m$$

とするのがよい

素数

- 素数とは、1以外の数で1と自分自身しか約数がない整数のことをいう。例えば、2, 3, 5, 7…などが素数である

Question

- なぜ剰余演算？
- なぜmは素数がよい？

演習

- 任意の与えられた整数について、それが素数かを判定するプログラムを作成しなさい。
- 実行例:

```
./a.out
input a number: 10
```

素数でない

```
./a.out
input a number: 5
```

素数

ハッシュ表のサイズmを求めるアルゴリズム

入力：元配列に格納されているデータの個数nと値α（=m/n）。
　　ただし、α>1

出力：ハッシュ表のサイズm（値αn以上のもっとも小さい素数）

補助: i, j, fg

衝突

- 普通、データのとりうる値の種類に比べて、配列の添え字として使える値の範囲が小さい
- そのため、異なるデータが同一ハッシュ値を持つことがある。例えばハッシュ関数hash(x)=x mod 11を用いた場合、11を割り切れる番号（11, 22, 33, ...）がみんな0のハッシュ値を持つ
- このような事態を「衝突」と呼ぶ

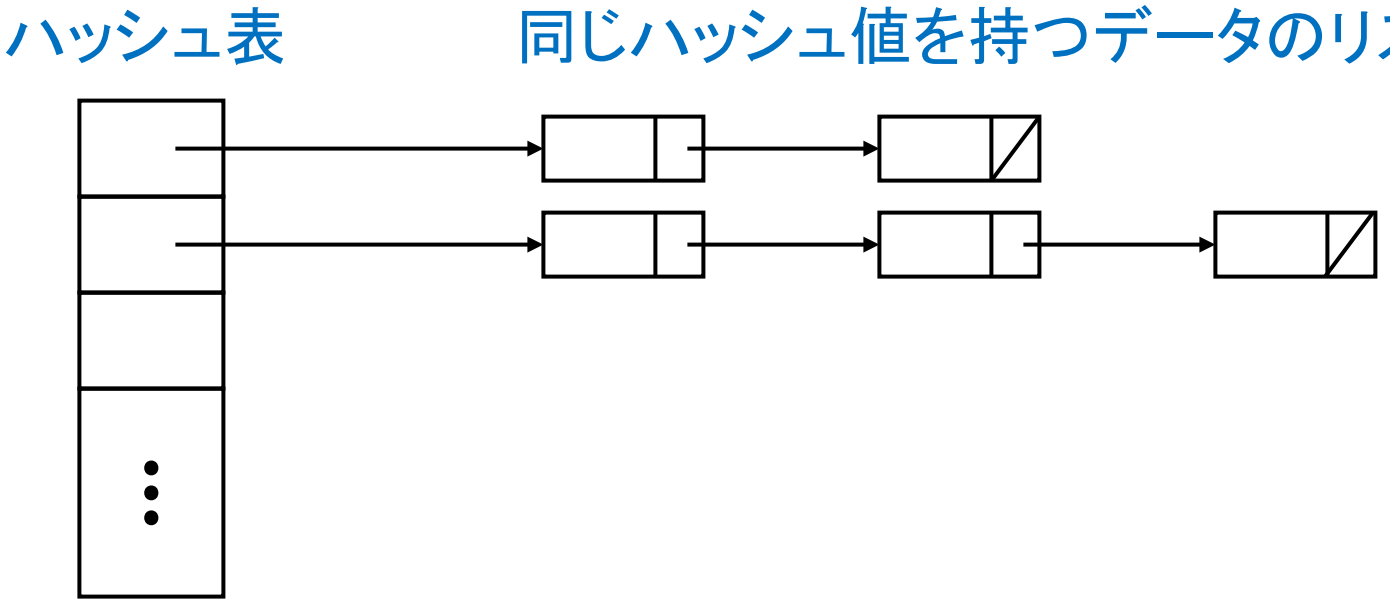
衝突の処理

処理法としては次の二つの方法がよく使われる

1. 同じハッシュ値を持つデータをリストでつなぐ方法（チェイン法）
2. 衝突が起きた場合、別のハッシュ関数を使って再度ハッシュする方法（オープンアドレス法、あるいは開番地法）
   - この処理を再ハッシュ（rehasing）という

チェイン法

- ハッシュ表の各要素に、このリストの先頭を指すポインターを入れる。ハッシュ関数を計算して特定のリストを選んでから、リストの上で探索を行う
- この方法の欠点は、同じハッシュ値を持つキーがたくさんあって（つまり特定のチェインが非常に長い）、ハッシュ法の意味がなくなる



オープンアドレス法（開番地法）

- 一般的には、m個のハッシュ関数h1, h2, ..., hmを用意し、衝突が起きないまでそれらの関数を用いて順に調べる方法
- このうち、もっとも簡単な方法として、線形走査法がある

線形走査法

- これは最初の場所の隣、その隣、さらにその隣と、配列番号を1つずつ増やしながら調べていく方法である。つまり、

$$h1(x)=h(x)$$

$$h2(x)=h(x)+1$$

$$h3(x)=h(x)+2$$

$$\vdots$$

とした場合に相当する

線形走査法について

- 簡単で確実な方法
  - すべての場所を探せるので、確実
- 表がいっぱいになり始めると「クラスター」と呼ばれる現象が生じやすくなり、再ハッシュの回数が増える可能性がある
- クラスター現象

  あるデータのハッシュ値がhだとすると、ほかのデータのハッシュ値がh近辺の可能性が高く、その近辺にクラスター、つまりデータのかたまりができやすい。このような現象をクラスター現象という。クラスターができてしまうと、線形走査法では効率が落ちる

線形走査法の例

- 例：a[]={65, 66, 75, 76, 77}
- ハッシュ表のサイズを11とし、ハッシュ関数を次のように定義する。

$$h(x) = x \bmod 11$$

$$h1(x)=h(x)$$

$$h2(x)=[h(x)+1] \bmod 11$$

$$h3(x)=[h(x)+2] \bmod 11$$

⋮

線形走査法の例

- 各データに対してハッシュ値をh1(x)を用いて計算すると、

  h1(65)=10

  h1(66)=0

  h1(75)=9

  h1(76)=10　衝突が起きるのでハッシュ値を再度計算

  h2(76)=0

  h3(76)=1

  h1(77)=0　衝突が起きるのでハッシュ値を再度計算

  h2(77)=1

  h3(77)=2

2重ハッシュ法、2次ハッシュ法

- 2つのハッシュ関数hとgを用意し、以下のように求める方法を2重ハッシュ法（double hashing）という

  $h1(x)=h(x)$

  $h2(x)=[h(x)+g(x)] \bmod m$

  $h3(x)=[h(x)+2g(x)] \bmod m$

- また、$hi(x)=[x+(i-1)^2] \bmod m$ で計算する2次ハッシュ法（quadratic hashing）と呼ばれる方法もある
- これらの方法の狙いは h1, h2, …などを互いに独立なものにし、クラスター（データのかたまり）をできにくくすることである

g(x)は注意して選ぶ！

- g(x)は注意して選ばないと、プログラムが正しく機能しない
- 実際、g(x)=q-(x mod q)がよいようである。ここのqはmより小さな整数である

2重ハッシュ法の例

- 例：a[]={65, 66, 75, 76, 77}
- ハッシュ表のサイズを11とし、ハッシュ関数を次のように定義する。

$$h1(x)=h(x)$$

$$h2(x)=[h(x)+g(x)] \bmod 11$$

$$h3(x)=[h(x)+2g(x)] \bmod 11$$

⋮

$$h(x)=x \bmod m$$

$$g(x)=8-(x \bmod 8)$$

2重ハッシュ法の例

- 各データに対してハッシュ値をh1(x)を用いて計算すると、

  h1(65)=10

  h1(66)=0

  h1(75)=9

  h1(76)=10　衝突が起きるのでハッシュ値を再度計算

  h2(76)=[10+(8-76%8)]%11=3

  h1(77)=0　衝突が起きるのでハッシュ値を再度計算

  h2(77)=[0+(8-77%8)]%11=3

  h3(77)=[0+2*(8-77%8)]%11=6

演習

- 上記の例において、2次ハッシュ法を用いるとする。つまり、

  $hi(x)=[x+(i-1)^2] \bmod m$

  をハッシュ関数として用いる。ただし、$m=11$

  問1：$h1(x),h2(x),h3(x)$の式を具体的に示しなさい。

  問2：$h1$でハッシュ値を求めると、

  $x=65, h1=10$
  $x=66, h1=0$
  $x=75, h1=9$
  $x=76, h1=10$
  $x=77, h1=0$

  が得られる。76と65が衝突しているのがわかる。76の新しいハッシュ値を求めなさい。

  問3：そのとき、衝突は解消したか。理由をつけてのべなさい。

  問4：解消していないと答える人はさらに新しいハッシュ値を求めなさい。

ハッシュ表作成アルゴリズム

入力：n個のデータが格納されている配列a, ハッシュ配列tのサイズm

出力：ハッシュ表配列t

補助：

1.ハッシュ表配列tを初期化

- 正の整数の探索問題であれば負数に設定すればよい

2.i=0からn-1まで、以下を実行

2.1 ハッシュ関数でa[i]のハッシュ値vを計算する

2.2 t[v] にすでに（初期値以外の）値が入っていれば、以下を繰り返す

2.2.1 他のハッシュ関数でハッシュ値vを再度計算する

2.3 ハッシュ表配列にデータを格納する

ハッシュ探索アルゴリズム

入力：探索キーx, ハッシュ表のサイズm, ハッシュ表配列t

出力：キーxがハッシュ表の中の位置p（ない場合-1）

補助：

1. xのハッシュ値vを計算する
2. x≠t[v]かつt[v] にすでに（初期値以外の）値が入っていれば、以下を繰り返す
   2.1 xのハッシュ値vを再度計算
3. p=v or p=-1とおく

平均ハッシュ回数の実験結果

- 5種類10万個の（0～RAND_MAXまでの）乱数を使用（乱数は5個程度重複あり）
- ハッシュ表作成時の平均ハッシュ回数

α(m/n)	2.0	1.7	1.4	1.25	1.1	1.05
線形	1.5	1.7	2.3	2.98	5.84	11.36
2重	1.4	1.5	1.8	2.2	3.3	5.06
2次	1.4	1.6	1.9	2.18	2.9	3.52

αの値が小さくなるにつれ、各手法に性能差が大きくなる。

第4回演習課題

1. 任意の与えられた値について、それ以下のすべての素数を求めるプログラム（ex04-prime-table.c）を作成しなさい。

   実行例：

   ```
   ./a.out
   input a number: 10
   2
   3
   5
   7
   ```

第4回演習課題

2. ハッシュ表のサイズを求めるアルゴリズムを実現する関数int CalM(int n, double α)を作成し、その動作を確認できるプログラム（ex04-calm.c）を作成しなさい。ただし、求めたサイズを関数の戻り値とする。

   注意：例えば整数変数にnα=6x1.2を与えると、値が7となってしまう。このような問題は切り上げ関数を用いるとよいだろう。具体的な関数名や、関数の意味・使い方などは各自で調べよう。

第4回演習課題

3. ハッシュ表作成アルゴリズムを実現する関数int MakeHashTable(int n, int m, int a[], int t[])を作成し、その動作を確認できるプログラム（ex04-makehashtb.c）を作成しなさい。ただし、
   a. aは元データ配列，tはハッシュ表配列，nは元データの個数，mはハッシュ表のサイズである。
   b. 元データは値が1以上1000000以下の重複なしの正の整数乱数で、講義資料のWebサイトからex04-a.datをダウンロードして使うこと（n=100000）。
   c. ハッシュ表のサイズmを125003とする。
   d. 再ハッシュには線形走査法を用いる。
   e. （一つのデータがハッシュ表に格納されるまでの）最大ハッシュ回数を関数の戻り値とする。
   f. main()関数においてm, ハッシュ表（配列t）を結果ファイルex04-t.datに出力すること。
   g. m, ハッシュ表が作成された平均ハッシュ回数と最大ハッシュ回数を計算機画面上に出力すること

課題4-3の実行例

ファイルへの入出力は、前回講義資料の「ファイル入出力」機能を利用するとよい。

```
$ ./a.out
m: 125003
平均ハッシュ回数: 2.7
最大ハッシュ回数: 127
$ more ex04-t.dat                ←ファイルの中身を見る
125003                           ←m
250006                           ←t[0]
875020                           ←t[1]
250008                           ←t[2]
750021                           ←t[3]
⋮
```

第4回演習課題

4. ハッシュ探索アルゴリズムを実現する関数int HashSearch(int m, int t[], int x)を作成し、その動作を確認するプログラム（ex04-hsearch.c）を作成しなさい。ただし、
   a. xは探索データである。データの位置情報を関数の戻り値とする（見つからないとき-1）。
   b. mとハッシュ表（配列t）は課題4-3で得られたファイルex04-t.datから読み込むこと

課題4-4の実行例

```
$ ./a.out
Input the search key (end with -99): 419076
nhash: 68                                   ←nhash: 一つの探索結果が得られるまでのハッシュ回数
Search result: 44134
Input the search key (end with -99): 79
nhash: 5
Search result: 83
Input the search key (end with -99): 100
nhash: 2
Search result: -1
Input the search key (end with -99): -99 ... 終了
```