アルゴリズム・データ構造 I 第10回
ハッシングの手法

名城大学理工学部情報工学科
山本修身

文字の番号を取得する

- 計算機内部では文字は数と結びつけられている．アルファベット文字と7ビット整数との標準的な結びつきとしてASCII (American Standard Code for Information Interchange) と呼ばれるコード体系がある．

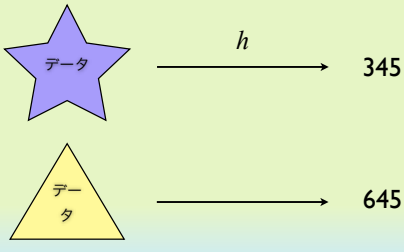
```
function charCode(str){
   var m = []
   for (var i = 0; i < str.length; i++){
     code = str.charCodeAt(i)
     m.push(code.toString(16))
   }
   puts(str)
   puts(m.join("-"))
}

charCode("Meijo University")
```

```
Meijo University
4d-65-69-6a-6f-20-55-6e-69-76-65-72-73-69-74-79
```

データから整数への写像

- データを一つ指定するとそれに対してある範囲の整数を計算する関数を考える．関数値はなんでも良いが，同じデータを入れたときは同じ値が返ってくる必要がある．
- このような関数を**ハッシュ関数 (hash function)** と呼ぶ．



文字列データのハッシュ関数

- 文字列データを数に変換する関数を作ってみる．

```
function h(s){
   var n = 0
   for (var i = 0; i < s.length; i++){
      n = (n * 234 + s.charCodeAt(i)) % 103
   }
   return n
}

var strs = ["meijo", "university", "abc",
    "shiogama","yagoto", "ueda", "hara",
    "irinaka", "yagoto-nisseki", "kanayama"]
for (var i = 0; i < strs.length; i++){
    puts(strs[i] + " : " + h(strs[i]))
}
```

```
meijo : 7
university : 12
abc : 96
shiogama : 2
yagoto : 44
ueda : 69
hara : 40
irinaka : 6
yagoto-nisseki : 83
kanayama : 57
```

ハッシュ関数を使ってデータを管理する (1)

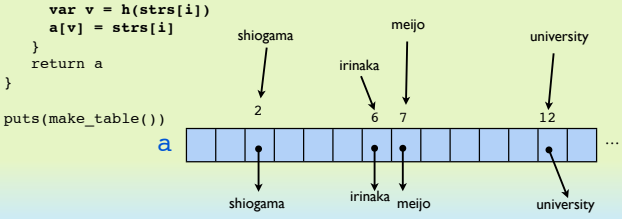
- 配列のハッシュ関数が指し示す場所にデータを入れる

```
var strs = ["meijo", "university", "abc",
    "shiogama","yagoto", "ueda", "hara",
    "irinaka", "yagoto-nisseki", "kanayama"]

function make_table(){
   var a = []
   for (var i = 0; i < strs.length; i++){
     var v = h(strs[i])
     a[v] = strs[i]
   }
   return a
}

puts(make_table())
```

```
,,shiogama,,,,irinaka,meijo,,,
,,university,,,,,,,,,,,,,,,,,,,,,,,,h
ara,,,,yagoto,,,,,,,,,,,,kanaya
ma,,,,,,,,,,,,ueda,,,,,,,,,,,,yago
to-nisseki,,,,,,,,,,,,abc
```



ハッシュ関数を使ってデータを管理する (2)

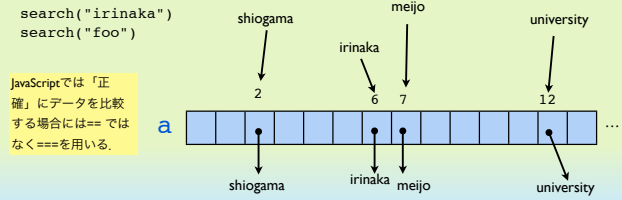
- 配列の中からデータを探し出す

```
var a = make_table()
function search(s){
  m = a[h(s)]
  if (m === undefined) puts(s + " is undefined")
  else puts(m)   ↑未定義値
}

search("irinaka")
search("foo")
```

```
irinaka
foo is undefined
```

JavaScriptでは「正確」にデータを比較する場合には== ではなく===を用いる．



ハッシュ関数を使ってデータを管理する (3)

- データを削除する．削除もハッシュ関数値を用いてその場所に要素があれば削除する

```
function delete_element(s){
  var val = a[h(s)]
  if (val === undefined)
    puts("cannot delete " + s + ".")
  else{
    a[h(s)] = undefined
    puts("ok. " + s + " has been deleted.")
  }
}

delete_element("irinaka")
delete_element("foobar")
search("irinaka")
```

ok. irinaka has been deleted.
cannot delete foobar.
irinaka is undefined

ハッシュ法の問題点

- ここで示した例の場合にはうまく行ったが，ハッシュ関数値が異なる要素について同じ値になるとうまくいかない．

```
function h(s){
   var n = 0
   for (var i = 0; i < s.length; i++){
      n = (n * 234 + s.charCodeAt(i)) % 13
   }
   return n
}

var strs = ["meijo", "university", "abc",
    "shiogama","yagoto", "ueda", "hara",
    "irinaka", "yagoto-nisseki", "kanayama"]
for (var i = 0; i < strs.length; i++){
    puts(strs[i] + " : " + h(strs[i]))
}
```

meijo : 7
university : 4
abc : 8
shiogama : 6
yagoto : 7
ueda : 6
hara : 6
irinaka : 6
yagoto-nisseki : 1
kanayama : 6

shiogama, ueda, hara, irinaka, kanayama は皆同じ値になってしまう．これではテーブルに書き込めない

チェイン法 (1)

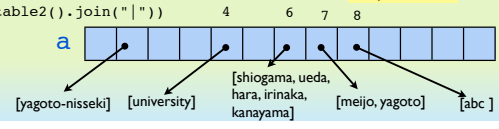
- この問題点はテーブルにそのまま要素を書くのではなくて，その中の配列に書くことによって解決する．

```
var strs = ["meijo", "university", "abc",
    "shiogama","yagoto", "ueda", "hara",
    "irinaka", "yagoto-nisseki", "kanayama"]

function make_table2(){
   var a = []
   for (var i = 0; i < strs.length; i++){
     var v = h(strs[i])
     if (a[v] === undefined) a[v] = [strs[i]]
     else a[v].push(strs[i])
   }
   return a
}
puts(make_table2().join("|"))
```

|yagoto-nisseki|||university||shiogama,ueda,hara,irinaka,kanayama|meijo,yagoto|abc

meijo : 7
university : 4
abc : 8
shiogama : 6
yagoto : 7
ueda : 6
hara : 6
irinaka : 6
yagoto-nisseki : 1
kanayama : 6



チェイン法 (2)

```
function search(s){
   var v = h(s)
   if (a[v] === undefined)
     puts(s + " is not in the table.")
   else {
      var lst = a[v]
      for (var i = 0; i < lst.length; i++){
        if (lst[i] == s){
           puts(s + " is in the table")
           return;
        }
      }
      puts(s + " is not in the table.")
   }
}
var a = make_table2()
search("shiogama")
search("irinaka")
search("motoyama")
```

セパレートチェイニング法の場合には，ハッシュ値を計算してから，さらにリストの中身を探さないといけない場合がある．その分時間がかかる．

shiogama is in the table
irinaka is in the table
motoyama is not in the table.

チェイン法 (3)

- テーブルの大きさをmとして，管理するデータの数をnとしたとき，$\alpha = n / m$ のことを **占有率 (load factor)**と呼ぶ．
- ハッシュ関数値はどの値も同じ程度の確からしさで出現するという仮定をおく（**単純一様ハッシュの仮定**）．
- **性質：** テーブルのi番目の要素に繋がっているデータの個数をx_iとしたとき，その期待値 $\mathrm{E}[x_i] = \alpha$ となる．

$$E[x_1 + x_2 + \cdots + x_m] = E[x_1] + E[x_2] + \cdots + E[x_m] = E[n] = n$$

単純一様ハッシュの仮定からすべて等しい

$$= mE[x_i]$$

$$E[x_i] = n/m = \alpha$$

チェイン法によるキーのアクセス時間（1）

性質： 衝突がチェイン法によって解決されるようなハッシュテーブルで，単純一様ハッシュを仮定すると，失敗する探索にかかる時間は平均 O(1 + α) である．

失敗する場合，ハッシュ値を計算して，その値に対応するテーブル上の場所を調べ，そこに繋がれているリストの要素をすべて調べる．リストの要素数は平均αである．ハッシュ関数値を計算するための時間も含めて探索にかかる時間は，O(1 + α) である．

チェイン法によるキーのアクセス時間（2）

性質： 衝突がチェイン法で解決されるハッシュテーブルで，単純一様ハッシュを仮定すれば，成功する探索にかかる時間の平均は O(1 + α) である．

ある値がハッシュテーブルに存在するとき，その値を見つけるにはハッシュ間数値を計算して，それに対応するリストを先頭から見て，対応するものを探せばよい．その要素をサーチするには，始めその要素が付け加えられたときのそのリストの長さの期待値 + 1となる．n個の要素がハッシュテーブルに入るとすれば，平均のアクセス回数は，

$$\frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{i-1}{m}\right)=1+\frac{1}{nm}\sum_{i=1}^{n}(i-1)=1+\frac{\alpha}{2}-\frac{1}{2m}$$

したがって，ハッシュ関数の計算時間も含めて，$O(2+\alpha/2-1/2m)$ となる．

チェイン法によるキーのアクセス時間（3）

- 扱うデータの数nがmに比例する程度の個数であれば，α = n / m = 定数なので，データの挿入，探索，削除のすべての操作がO(1)で実行可能である．

ハッシュ関数の作り方 (1)

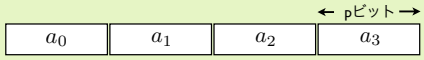
- ハッシュデータは整数であると仮定する．
- 比較的良いハッシュ関数を作る方法について考える．
- 単純な方法として**除算法 (division method)** がある．

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

mとして，2の累乗を用いてはいけない．除算法の場合，下位ビットをそのままもってくることになるので，大抵の場合，あまり良い結果が得られない．同様にして，10の累乗も避けるべきである．

ハッシュ関数の作り方 (2)

- m = 2^p - 1としたとき，除算法で文字列を扱うハッシュ関数を作ると，このとき，それぞれの文字の区切れが2^pであるとするとハッシュ関数は以下のようになる．



$$k = a_0 2^{3p} + a_1 2^{2p} + a_2 2^p + a_3$$

$h(k) = k \bmod (2^p - 1)$ と仮定すれば，

$$\begin{aligned} h(k) &= a_0 2^p 2^p 2^p + a_1 2^p 2^p + a_2 2^p + a_3 \\ &= a_0 \cdot 1 \cdot 1 \cdot 1 + a_1 \cdot 1 \cdot 1 + a_2 \cdot 1 + a_3 \\ &= a_0 + a_1 + a_2 + a_3 \end{aligned}$$

となり，文字の順番に依らず同じ値がでてしまい，良くない．

ハッシュ関数の作り方 (3)

- 実際，以下のようになる．

```
function h(s){
  var sum = 0
  var p = 255
  for (var i = 0; i < s.length; i++){
     var c = s.charCodeAt(i)
     sum = (sum * 256 + c) % p
  }
  return sum
}

function work(s){
  puts(s + ": " + h(s))
}

work("abc")
work("bca")
work("yamamoto")
work("motoyama")
```

```
abc: 39
bca: 39
yamamoto: 106
motoyama: 106
```


乗算法 (1)

ハッシュ関数として乗算法が知られている．これは，ある定数を実数値に掛けて，その小数点部分を用いてハッシュ関数値を計算する方法である．直接的には入力は適当な整数（実数）であることが必要である．

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

ただし，Aは適当な定数である．また，0〜m - 1がハッシュ関数の値域となる．D. E. Knuthによれば，A = (√5 - 1) / 2 とすると良い結果が得られる．

乗算法 (2)

- 以下のようなプログラムを実行してみる．freqには頻度が入る．100000個のデータについて，どのようにハッシュ関数値が分布するか調べると以下のようになる．

```
function work(){
    var A = (Math.sqrt(5) - 1) / 2
    var m = 50
    function h(k){
        var b = A * k
        var n = Math.floor((b - Math.floor(b)) * m)
        return n
    }
    var freq = []
    for (var i = 0; i < m; i++) freq[i] = 0
    for (var i = 0; i < 100000; i++){
        var j = h(i)
        freq[j] += 1
    }
    puts(freq)
}

work()
```

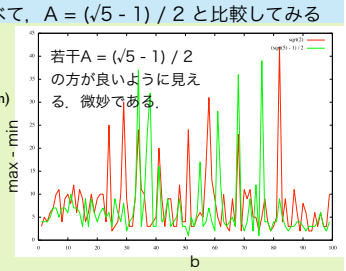
2000,2000,2000,2000,2000,2000,2001,1998,2002,1999,
2001,2000,1999,2000,2000,2000,2001,1999,2001,1998,
2000,2000,2001,2000,1999,2001,2000,1999,2000,2000,
2001,1999,2002,1998,2001,1999,2001,1999,2000,2001,
2000,2000,2001,1998,2001,2000,2000,2000,2000,2000

乗算法 (3)

A = √2 とした場合について調べて，A = (√5 - 1) / 2 と比較してみる

```
function work(A, b){
    var m = 50
    function h(k){
        var b = A * k
        var n = Math.floor((b - Math.floor(b)) * m)
        return n
    }
    var freq = []
    for (var i = 0; i < m; i++) freq[i] = 0
    for (var i = 0; i < 100000; i += b){
        var j = h(i)
        freq[j] += 1
    }
    var amax = Math.max.apply(null, freq)
    var amin = Math.min.apply(null, freq)
    return (amax - amin)
}

for (var b = 1; b < 100; b++)
  puts(b + " " + work(Math.sqrt(2), b) + " " + work((Math.sqrt(5) - 1) / 2, b))
```



若干A = (√5 - 1) / 2 の方が良いように見える．微妙である．

bごとにデータをハッシュにかけたとき，５０区分に落ちる頻度の最大値から最小値を引いた値をプロットしてみる．