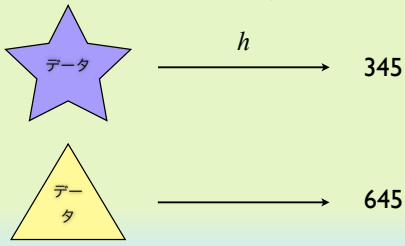
アルゴリズム・データ構造 I 第11回
色々なデータをハッシュで管理する

名城大学理工学部情報工学科
山本修身

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．



ハッシュ関数を使ってデータを管理する (1)

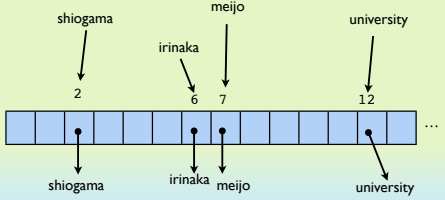
- 配列のハッシュ関数が指し示す場所にデータを入れる

```
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,,,,,,,,,,,,,,,,,,,,hara,,,,yagoto,,,,,,,,,,,,kanayama,,,,,,,,,,,,ueda,,,,,,,,,,,,yagoto-nisseki,,,,,,,,,,,,abc



ハッシュ関数を使ってデータを管理する (2)

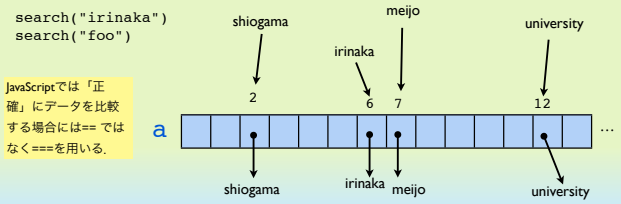
- 配列の中からデータを探し出す

```
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)

復習 7

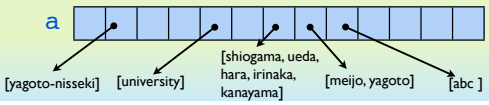
- この問題点はテーブルにそのまま要素を書くのではなくて，その中の配列に書くことによって解決する．

```
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



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

チェイン法 (3)

復習 8

- 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 $E[x_i] = \alpha$ となる．

$$E[x_1 + x_2 + \cdots + x_m] = \underbrace{E[x_1] + E[x_2] + \cdots + E[x_m]}_{\text{単純一様ハッシュの仮定からすべて等しい}} = E[n] = n$$

$$= mE[x_i]$$

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

チェイン法によるキーのアクセス時間（1）

復習 9

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）

復習 10

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関数値を計算して，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）

復習 11

- 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)

復習 12

- 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．

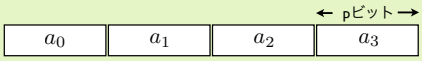
$$h(k) = k \bmod m$$

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)

復習

- 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)

復習

- 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

文字列データのハッシュ関数の定義

- 文字列データを数に変換する関数をもう一度作ってみる．

```
function h(s){
   var n = 0
   for (var i = 0; i < s.length; i++){
      n = (n * 234 + s.charCodeAt(i)) % 31
   }
   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 : 2
university : 11
abc : 7
shiogama : 16
yagoto : 7
ueda : 5
hara : 8
irinaka : 23
yagoto-nisseki : 6
kanayama : 11

オープンアドレッシング (1)

- チェイン法と並んでもう１つのハッシュアルゴリズムとしてオープンアドレッシング法ある．
- オープンアドレッシング法では衝突が起こった場合，適当なアルゴリズムによって別の場所にデータを入れる．
- この場合，ロードファクタ＜１

```
function add_data(table, data){
   var v = h(data)
   while (table[v] !== undefined){
     v = (v + 1) % 31
   }
   table[v] = data
}
```

線形探索

table

```
0-->
1-->
2-->meijo
3-->
4-->
5-->ueda
6-->yagoto-nisseki
7-->abc
8-->yagoto
9-->hara
10-->
11-->university
12-->kanayama
13-->
14-->
15-->
16-->shiogama
17-->
18-->
19-->
20-->
21-->
22-->
23-->irinaka
```

この方法はクラスターが出来易い

主クラスター化

meijo : 2
university : 11
abc : 7
shiogama : 16
yagoto : 7
ueda : 5
hara : 8
irinaka : 23
yagoto-nisseki : 6
kanayama : 11

衝突

衝突回数：3回

オープンアドレッシング (2)

- 線形探索はクラスターの問題があり，それを解決するために，2次関数的つぎの場所を探す2次関数探索 (quadratic probing) が知られている．

一般には `c * i + d * i * i`

```
function add_data(table, data){
   var i = 0
   while (true){
     var v = (h(data) + i + i * i) % 31
     if (table[v] === undefined){
        table[v] = data
        return
     }
     i = i + 1
   }
}
```

2次関数探索

```
0-->
1-->
2-->meijo
3-->
4-->
5-->ueda
6-->yagoto-nisseki
7-->abc
8-->hara
9-->yagoto
10-->
11-->university
12-->
13-->kanayama
14-->
15-->
16-->shiogama
17-->
18-->
19-->
20-->
21-->
22-->
23-->irinaka
```

線形探索よりもクラスターが出来にくい．

この状態からたとえばハッシュ関数値が7の要素をもう１つ加えると "abc", "yagoto"と同じ動きをする（副クラスター化）

meijo : 2
university : 11
abc : 7
shiogama : 16
yagoto : 7
ueda : 5
hara : 8
irinaka : 23
yagoto-nisseki : 6
kanayama : 11

衝突

衝突回数：2回

オープンアドレッシング (3)

- オープンアドレッシングで最良の方法として知られているのはダブルハッシュ法である．この方法では，元のハッシュ関数と衝突が起きたときにずらす量を決めるハッシュ関数を独立に利用する．

```
function add_data(table, data){
   var i = 0
   while (true){
     var v = (h(data) + i * h2(data)) % 31
     if (table[v] === undefined){
        table[v] = data
        return
     }
     i = i + 1
   }
}
```

```
function h2(s){
   var n = 0
   for (var i = 0; i < s.length; i++){
      n = (n * 22 + s.charCodeAt(i)) % 7
   }
   return n
}
```

オープンアドレッシング (4)

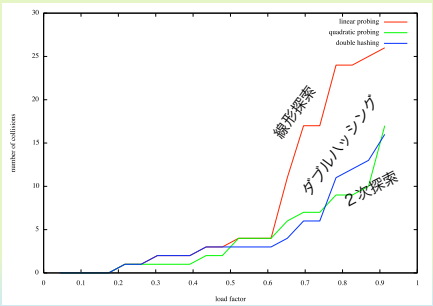
ダブルハッシュ法を用いると，データはかなり散らばる．右の例ではクラスターと呼べるものがあまり顕著ではない．

```
0-->
1-->
2-->meijo
3-->
4-->
5-->ueda
6-->yagoto-nisseki
7-->abc
8-->yagoto
9-->
10-->
11-->university
12-->
13-->kanayama
14-->hara
15-->
16-->shiogama
17-->
18-->
19-->
20-->
21-->
22-->
23-->irinaka
```

衝突回数：3回

オープンアドレッシング (5)

ロードファクターを大きくして試してみる．必ずしもダブルハッシングが良いわけではないが，ロードファクターが大きくなると線形探索は明らかに良くない．



オープンアドレッシングにおける要素の削除 (1)

- オープンアドレッシングは要素を追加することと，要素を探すことだけを行うのであれば，うまく動く．
- 要素を削除する必要がある場合，工夫が必要となる．

データの挿入

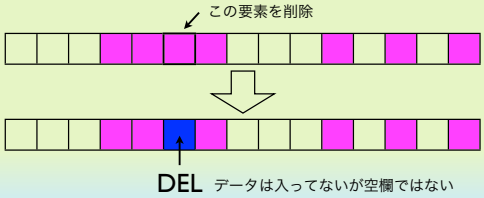
```
function add_data(table, data){
   var i = 0
   while (true){
     var v = h(data, i)
     if (table[v] === undefined){
        table[v] = data
        return
     }
     i = i + 1
     if (i >= m) break;
   }
   puts("table is full")
}
```

データの探索

```
function search_data(table, data){
   var i = 0
   while (true){
     var v = h(data, i)
     if (table[v] === data){
        return v
     }
     i = i + 1
     if (i >= m ||
   table[v] === undefined) break;
   }
   return null
}
```

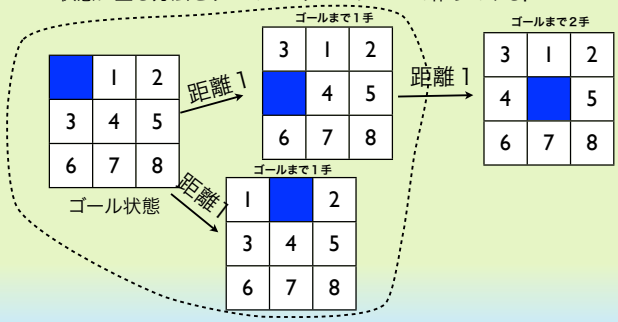

オープンアドレッシングにおける要素の削除 (2)

- クラスターになっているところからデータを削除する場合には，空欄にするのではなくて，削除されたことを表す記号（たとえばDELなど）を置くことによって，クラスターの接続性を保証する．
- この場合，データの探索や挿入も複雑になるので，むしろチェイン法を用いる方が楽になる．



8パズル再び (1)

- 木構造を根から順に構築することによって，最少回数でゴール状態に至る方法をすべてのパターンについて作ってみる．



8パズル再び (2)

- まず，８パズルのパターンを管理するハッシュ表を用意する．

```
/*
 * A hash function which outputs values
 * less than 20,011
 */
function h(pat){
    var s = 0
    for (var i = 0; i < 9; i++){
        s = (s  * 3444 + pat[i]) % 20011
    }
    return s
}

function eq_pat(pat1, pat2){
    for (var i = 0; i < 9; i++){
        if (pat1[i] != pat2[i])
            return false
    }
    return true
}
```

```
var pat_table = []

function add_pat(pat, dist){
    var v = h(pat)
    if (pat_table[v] === undefined){
        pat_table[v] = [[pat, dist]]
        return 1
    } else{
        lst = pat_table[v]
        for (var i = 0; i < lst.length; i++)
            if (eq_pat(lst[i][0], pat)) return 0
        pat_table[v].push([pat, dist])
        return 1
    }
}
function find_pat(pat){
    var v = h(pat)
    if (pat_table[v] === undefined) return -1
    else {
        var lst = pat_table[v]
        for (var i = 0; i < lst.length; i++)
            if (eq_pat(lst[i][0], pat)) return
lst[i][1]
        return -1
    }
}
```

8パズル再び (3)

幅優先探索でパターンを走査する．
最終的にハッシュテーブルに入った個数を表示する．

```
function find_zero(pat){
    for (var i = 0; i < 9; i++){
        if (pat[i] == 0) return i
    }
    return -1
}

function work(){
    var init_pat = [0, 1, 2, 3, 4, 5, 6, 7, 8]
    var queue = [[init_pat, 0]]
    var counter = 0
    while (queue.length > 0){
        var [pat, dist] = queue.shift()
        var ddist = find_pat(pat)
        if (ddist < 0){
            add_pat(pat, dist)
            counter += 1
            var p = find_zero(pat)
            var px = p % 3
            var py = Math.floor(p / 3)
            右へ動かす
            if (px > 0){
                ppat = pat.slice(0)
                ppat[p] = ppat[p - 1]
                ppat[p - 1] = 0
                queue.push([ppat, dist + 1])
            }
            左へ動かす
            if (px < 2){
                ppat = pat.slice(0)
                ppat[p] = ppat[p + 1]
                ppat[p + 1] = 0
                queue.push([ppat, dist + 1])
            }
            上へ動かす
            if (py > 0){
                ppat = pat.slice(0)
                ppat[p] = ppat[p - 3]
                ppat[p - 3] = 0
                queue.push([ppat, dist + 1])
            }
            下へ動かす
            if (py < 2){
                ppat = pat.slice(0)
                ppat[p] = ppat[p + 3]
                ppat[p + 3] = 0
                queue.push([ppat, dist + 1])
            }
        }
    }
    puts(counter)
}
```


8パズル再び (4)

- ハッシュテーブルを使うことによっていくつかの事柄がわかる．
- work() を実行すると，ハッシュテーブルに蓄えられたパターン数が表示される．→ 181,440通り
- 最後にキューに付加されたパターンを調べると，ゴール状態から最も遠いパターンが分かる → [8,7,6,0,4,1,2,5,3]（ゴール状態まで31手）