アルゴリズムとデータ構造入門-14 2006年1月17日

アルゴリズムとデータ構造入門
Sorting（整列）
Hashing（ハッシュ法）

奥 乃 博

11年前の今朝
阪神・淡路大震災
犠牲者死者だけで6434人

天災は忘れた頃にやってくる （寺田寅彦）
天才は忘れた頃にやってくる （奥乃博）

1

1月17日・本日のメニュー

1. vector（ベクタ）
2. Sorting（整列）
3. hashing（ハッシュ法）
4. アンケート（最後の15分）

- 配布する用紙に名前を記入して下さい。
- 回収は学生同士で。
- 教員は一切タッチしません。（退場！）

2

Sorting（整列）

- 内部整列（internal sorting）
  - データはすべて主記憶上に置いて整列
  1. 作業領域を極力減らす。
  2. 比較回数を極力減らす。
- 外部整列（external sorting）
  - 外部の記憶装置を用いて整列
  3. 主記憶と補助記憶との間でのデータ転送回数を極力減らす。

3

Internal Sorting（内部整列）

- 逐次入力型
  - *挿入ソート（insertion sort）*
  - **ヒープソート（heap sort）**
- バッチ型
  - *クイックソート（quick sort）*
  - **バブルソート（bubble sort）**
- その他（外部整列と共通）
  - **マージソート（merge sort、併合ソート）**

4

クイックソートの補足

- pivotのとり方は工夫が必要
- すでに並んでいる場合には、最小あるいは最大要素を pivot に取ると最悪。
- 適切な pivot を取れば、 $\Theta(n \log n)$
- リスト使用の場合は、pivot 選択がリスト辿りが必要。コストが重い。
- ベクタ使用の場合には、コストが軽い。

5

新しいデータ型：ベクタ（vector）

- データの並びを表現するデータ型
- #(《要素$_0$》…《要素$_{n-1}$》)
- インデックス（index）は0から始まる（*0-origin*）
- 《要素 》 は任意のデータ
- いわゆる1次元配列（*array*）
- Constructor（構築子）
  - (make-vector 〈サイズ〉 [〈データ〉])
  - (vector 〈データ$_0$〉 ･･･ 〈データ$_{n-1}$〉)
  - (list->vector 〈リスト〉)

6

新しいデータ型：ベクタ(vector)(続)

- Selectors(選択子)
  - `(vector-ref 〈ベクタ〉〈インデックス〉)`
- その他
  - `(vector-length 〈ベクタ〉)`

    サイズを返す
  - `(vector-set!`

    `〈ベクタ〉〈インデックス〉〈データ〉)`
  - `(vector->list 〈ベクタ〉)`
  - 入出力は

    `#(1 2 3 4 5)`

7

ベクタ(vector)の例

- `(define y (make-vector 5))`
- `(define x #(1 2 3 4 5))`
- `(vector-length x)`
- `(vector-ref x 2)`
- `(vector-set! x 3 128)`
- `x`
- `(vector-set! x 0 'foo)`

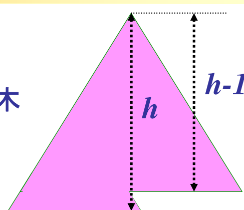
8

ヒープ(heap)というデータ構造

- ヒープとは2分木の特殊形
- ヒープの高さを*h*とすると
  1. 高さ*h-1*までは完全2分木
  2. 高さ*h*の葉は左詰
  3. 親ノードの値 v_p と子ノードの値 v_c とすると

     v_p *pred* v_c

     が成立。*pred* は比較
- ベクタで実現する

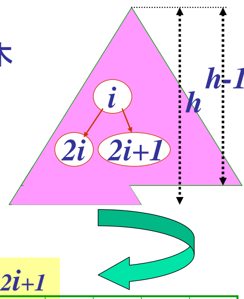


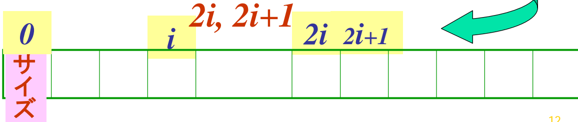
10

ヒープ(heap)のベクタ表現

■ヒープの高さをhとすると
- 高さh-1までは完全2分木
- 高さhの葉は左詰

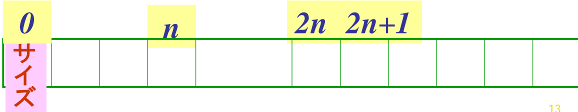
■ベクタで実現する
- 親のインデックスをi
- 子のインデックスは $2i, 2i+1$





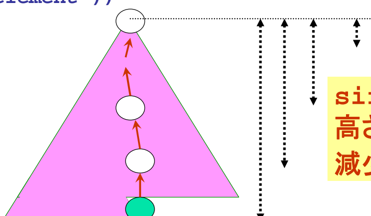
ヒープの諸演算・手続き

```
(define (make-heap maxsize)
  (let ((heap (make-vector (+ maxsize 1))))
     (vector-set! heap 0 0)
     heap )))
(define (heap-size heap) (vector-ref heap 0))
(define (heap-left-child n) (* n 2))
(define (heap-right-child n) (+ (* n 2) 1))
(define (heap-parent n) (quotient n 2))
(define (heap-top heap) (vector-ref heap 1))
(define (heap-size-set! heap n)
   (vector-set! heap 0 n) )
```



insert-heap (ヒープに要素挿入)

```
(define (insert-heap heap element pred)
  (let ((n (+ (heap-size heap) 1)))
    (vector-set! heap n element)
    (heap-size-set! heap n)
    (sift-up heap n element pred)
    element ))
```



sift-upでは、高さが1つずつ減少。

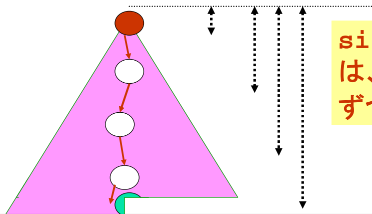
sift-up（heap修復）

```
(define (sift-up heap from element pred)
  (if (<= from 1)
      element
      (let ((parent (heap-parent from)))
        (let ((value (vector-ref heap parent)))
          (cond
             ((pred value element) element)
             (else
               (vector-set! heap from value)
               (vector-set! heap parent element)
               (sift-up heap parent element
                        pred )))))))
```

- 1回繰り返すと高さが1減少
- 要素数を n とすると計算量は $\Theta(\log n)$

最大要素の抽出とheap 修復

```
(define (heap-extract-top heap pred)
  (let* ((value (heap-top heap))
         (n (heap-size heap))
         (element (vector-ref heap n)) )
    (heap-size-set! heap (- n 1))
    (vector-set! heap 1 element)
    (sift-down heap 1 (- n 1) element pred)
    value ))
```



sift-downでは、高さが1つずつ増加。

16

sift-down（heap修復）

```
(define (sift-down heap from to element pred)
  (let ((left-child (heap-left-child from))
        (right-child (heap-right-child from)) )
    (if (or (>= from to) (> left-child to))
        element
        (let ((max-child
                (if (> right-child to)
                    left-child
                    (if (pred
                          (vector-ref heap left-child)
                          (vector-ref heap right-child) )
                        left-child
                        right-child ))))
          (let ((max-child-value
                  (vector-ref heap max-child)))
            (cond ((pred element max-child-value)
                   element)
                  (else
                   (vector-set! heap from
                                max-child-value)
                   (vector-set! heap max-child element)
                   (sift-down heap max-child to
                              element pred ))))))))
```

17

ヒープソート（heap-sort）

```
(define (heap-sort records . args)
  (let ((pred (if (null? args) >= (car args)))
        (heap (make-heap 100))
        (result ()) )
    (for-each
      (lambda (i) (insert-heap heap i pred))
      records)
    (do ((i (length records) (- i 1))
         (result nil) )
        ((<= i 0) (reverse result))
      (set! result
        (cons (heap-extract-top heap pred)
              result )))))
```

- ヒープソートの計算量 $\Theta(n \log n)$

19

ヒープソート（heap-sort）の正しさ

- ベクタ x のヒープ成立条件 *heap(m,n)*

  $\forall i \in [m+1, n] \quad x[i/2] \le x[i]$

- sift-down では
  - 実行前： *heap(1,n)* の成立は？
  - 実行後： *heap(1,3)* が成立。
- i回目の sift-down では
  - 実行前：*heap(1,k)*は成立, *heap(k,n)* ？
  - 実行後：*heap(k,2k+1)* が成立。
  - つまり, *heap(1,2k+1)* が成立。

20

ヒープソート（heap-sort）の正しさ (2)

- ベクタ x のヒープ成立条件 *heap(m,n)*

  $\forall i \in [m+1, n] \quad x[i/2] \le x[i]$

- sift-up では
  - 実行前：*heap(1,n)* 成立, *heap((n+1)/2,n+1)* だけが？
  - 実行後：*heap((n+1)/4,(n+1)/2)* は？
- i回目の sift-up では
  - 実行前：*heap(k/2,k)*？ 他は成立。
  - 実行後：*heap(k/2,n)* 成立, *heap(k/4,k/2)* ?

21

バブルソート(bubble sort)

```
(define (bubble-sort records . args)
  (let ((pred (if (null? args) >= (car args)))
        (size (vector-length records)) )
    (do ((i 0 (+ i 1)))
        ((>= i size) records)
      (do ((j (- size 1) (- j 1))
           (data nil) )
          ((<= j i))
        (set! data (vector-ref records j))
        (cond ((pred data
                     (vector-ref records (- j 1)) )
               (vector-set! records j
                     (vector-ref records (- j 1)) )
               (vector-set! records (- j 1)
                     data) ))))))
```

22

バブルソート(bubble sort)実行トレース

```
9 1 8 2 5 3 0 7 4
9|8 1 7 2 5 3 0 4
9 8|7 1 5 2 4 3 0
9 8 7|1 5 2 4 3 0
9 8 7 5|1 4 2 3 0
9 8 7 5 4|1 3 2 0
9 8 7 5 4 3|1 2 0
9 8 7 5 4 3 2|1 0
```

23

バブルソート(bubble sort)の計算量

1回ごとに敷居(|)が1つずつ減る

$$\sum_{i=1}^{n}(i-1) = \frac{1}{2}n(n-1)$$

$$\Theta(n^2)$$

24

シェルソート(Shell sort)

- バブルソート: 隣接データを比較
  - $h=1$
- 飛び飛び(h)に比較
  - $h_k = 3h_{k-1} + 1, \ldots, 1$ の時
  - e.g., 40, 13, 4, 1

$$\Theta(n^{1.25})$$

25

マージソート(併合ソート、merge sort)

- ソート済みのデータを前からマージ(併合)
- リスト版
- ベクタ版
- 計算量は両方のデータのスキャンのみ
- m 個のデータと n 個のデータとすると

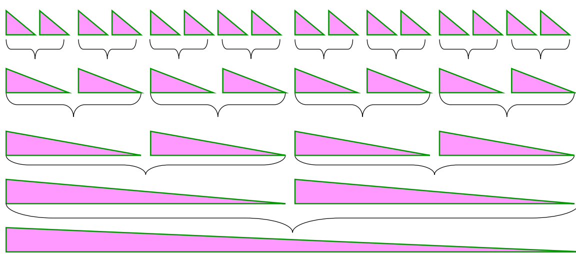
$$\Theta(m+n)$$

- 空間計算量も余分に $\Theta(m+n)$

27

内部マージソート(In-place merge sort)

- 分割統治型



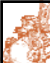
ラン(run)と言う

$$\Theta(n \log n)$$

Sorting（整列）のまとめ

- 選択ソート (selection sort) $\Theta(n^2)$
- 挿入ソート (insertion sort) $\Theta(n^2)$ 定数小
- シェルソート (Shell sort) $\Theta(n^{1.25})$
  - $h_k = 3h_{k-1} + 1, \ldots, 1$ の時
- クイックソート (quick sort), 分割統治法 (devide and conquer)
  - 平均 $\Theta(n \log n)$ 最悪 $\Theta(n^2)$
- ヒープソート (heap sort) 常時 $\Theta(n \log n)$
- マージソート (merge sort) 常時 $\Theta(n \log n)$

30



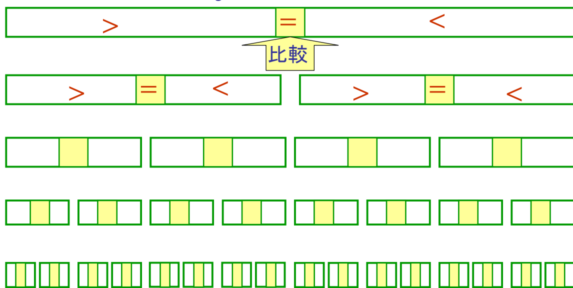
Sorting（整列）のデモ

- Java のデモプログラム
- http://winnie.kuis.kyoto-u.ac.jp/~okuno/Lecture/05/IntroAlgDs/

31

Sort（整列）済ベクタの探索

- 2分探索（binary search）



- 2分探索の計算量 $\Theta(\log n)$

33

ハッシュ法(hashing)

- 探したいデータの範囲膨大
- 例: 最大10文字の単語
- 50文字とすると組合わせの数は 50^{10}

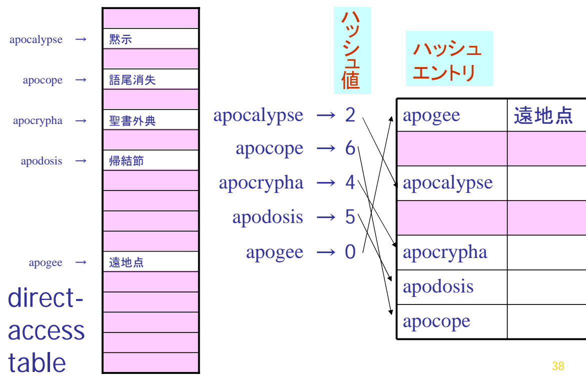
  $\log(50^{10}) = 10(2 - \log 2) \cong 17$ 10^{17}
- ところが実際の単語数は高々 10^{6}
- ベクタ(配列)で単語を管理すると疎

  すかすかの配列
- ハッシュ法(hasing)を使用

37

辞書とハッシュ表

empty(空)



38

ハヤシライスを知っていますか

- hashed beaf
- 語源は同じ

39

ハッシュ法(hashing)

- キーの値の探索なしにアクセス
- ハッシュ関数(hash function)
  キー ⇒ ハッシュ値(整数)
- ハッシュ表(hash table)、サイズ M
- 占有率(load factor) α 、データ量 N

$$\alpha = N / M$$

- 異なるキーに対してハッシュ値が同じ
  ハッシュ値の衝突(collision)

ハッシュ関数(hash function)

- 設計の指針: ランダム性を有するもの。
- キー: $x = a_1 a_2 \cdots a_n$ $key(x) = m$
- 例1:キーから $h_1(x) \equiv m \mod M$
- 例2: 文字列から整数への写像

$$h_2(x) \equiv \sum_{i=1}^{n} code(a_i) \mod M$$

- 例3: m^2の中央部分の $\log M$ 桁分を使用

$$h_3(x) \equiv \left\lfloor m^2 / K \right\rfloor \mod M$$

where constant K such that $MK^2 \cong N^2$

ハッシュ法の基本手続き

- 挿入(insert)
  hash表にkeyを持つデータを挿入
- 検索(member)
  hash表からkeyでデータを検索
- 削除(delete)
  hash表からkeyを持つデータを削除

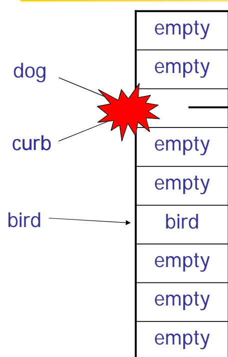
ハッシュ値衝突(collision)対処法

- チェイン法(chaining, separate chaining, 連鎖法、内部ハッシュ法)
- 開番地法(open addressing, オープン法、外部ハッシュ法)
  1. 線形走査法(linear probing)
  2. 万能ハッシュ法(universal hashing)
  3. 2重ハッシュ法(double hashing)
     h, gとすると、
     $h(x), h(x)+g(x), h(x)+2g(x), h(x)+3g(x), \ldots$

43

チェイン法

- ハッシュ値の衝突
  Backet(バケツ)を作り、つないで行く。



- チェイン法の
  平均最悪計算量

1. 挿入 $\Theta(N)$
2. 検索 $\Theta(N)$
3. 削除 $\Theta(N)$

44

内部ハッシュ法(internal hashing)

- ハッシュ関数 h_i
- 占有率 α $\alpha = N/M < 1$
- エントリに状態を導入
  empty/deleted/key(データ)

1. 挿入: empty/deleted というフラグのあるエントリに入れる
2. 検索:
   empty/deletedまで探す。
3. 削除:
   deletedというフラグを立てる。

empty	
empty	
empty	
empty	
deleted	
empty	
empty	
キー1	データ1
empty	
empty	
empty	
キー2	データ2
empty	
empty	
empty	
empty	

45

線形走査法（linear probing）

- ハッシュ関数　h
- 衝突発生時
  $h_i \equiv h+i \bmod M$
- 挿入
  - empty/deleted というフラグのあるエントリに入れる
- 検索
  - empty/deletedまで探す。
- 削除
  - deletedというフラグを立てる。

empty	
empty	
empty	
empty	
empty	
empty	
empty	
empty	
empty	
empty	
empty	
empty	
empty	
empty	
empty	
empty	

万能ハッシュ法（universal hashing）

- ハッシュ関数　$h, \cdots$
- *ハッシュ関数をランダムに選択*
- 挿入
  - empty/deleted というフラグのあるエントリに入れる
- 検索
  - empty/deletedまで探す。
- 削除
  - deletedというフラグを立てる。

empty	
empty	
empty	
empty	
empty	
empty	
empty	
empty	
empty	
empty	
empty	
empty	
empty	
empty	
empty	
empty	

2重ハッシュ法（double hashing）

- ハッシュ関数　h, g
- 衝突発生時
  $h_i \equiv h+ig \bmod M$
- 挿入
  - empty/deleted というフラグのあるエントリに入れる
- 検索
  - empty/deletedまで探す。
- 削除
  - deletedというフラグを立てる。

empty	
empty	
empty	
empty	
empty	
empty	
empty	
empty	
empty	
empty	
empty	
empty	
empty	
empty	
empty	
empty	

線形走査法（linear probing）の例

1. h(dog)=2
2. h(Kyoto)=4
3. h(Univ)=0
4. h(Informatics)=2
5. h(SICP)=3
6. h(test)=8

0		
1	empty	
2		
3		
4		
5		
6	empty	
7	empty	
8		
9	empty	

49

線形走査法の挿入の計算量

1. n個のデータが格納、$n+1$個目のデータを挿入するときに$h_i(x)$がi回目で空いている確率

$$\frac{n}{M}\frac{n-1}{M-1}\frac{n-2}{M-2}\cdots\frac{n-i+1}{M-i+1}$$

2. 空きセルを見つけるまでの比較回数

$$1+\sum_{i=1}^{M-1}\frac{n(n-1)\cdots(n-i+1)}{M(M-1)\cdots(M-i+1)} \cong 1+\sum_{i=1}^{\infty}\left(\frac{n}{M}\right)^i = \frac{M}{M-n}$$

3. ハッシュ表にN個のデータを挿入する手間は

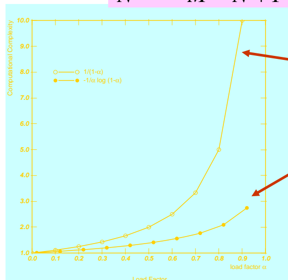
$$\sum_{n=0}^{N}\frac{M}{M-n} \cong \int_0^N \frac{M}{M-x}dx = M\log_e\frac{M}{M-N+1}$$

50

線形走査法の挿入の計算量（続）

4. 1回あたりの平均の挿入の手間は

$$\frac{M}{N}\log_e\frac{M}{M-N+1} \cong -\frac{1}{\alpha}\log_e(1-\alpha)$$



$$\frac{1}{1-\alpha}$$

$$-\frac{1}{\alpha}\log_e(1-\alpha)$$

51

線形走査法の検索の計算量

1. deleted　はないものと仮定
2. 表にキーがない時は、$n=N$ の挿入と同じ

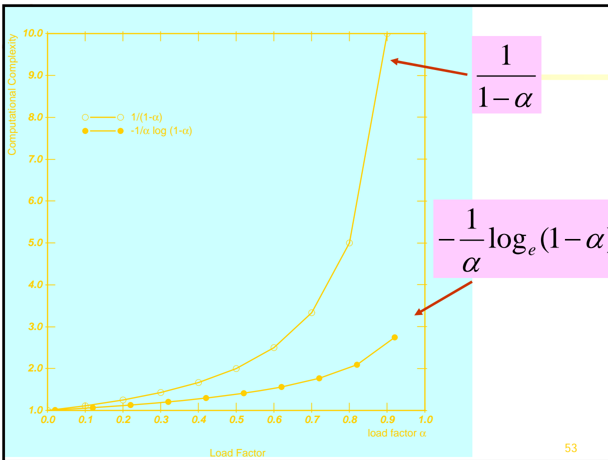
$$\frac{M}{M-N} = \frac{1}{1-\alpha}$$

3. 表にキーがある時

$$\frac{M}{N}\log_e \frac{M}{M-N+1} \cong -\frac{1}{\alpha}\log_e(1-\alpha)$$

4. 削除も検索と同じ
5. 上記の解析は、一様ハッシュ（uniform hashing）を仮定：　キーの探索列ランダム

52



$$\frac{1}{1-\alpha}$$

$$-\frac{1}{\alpha}\log_e(1-\alpha)$$

53

宿題はありません

- 期末テスト、健闘を期待します。
- 必修課題3もお忘れなく。
- 随意課題の提出でランクアップを。

テスト：　2月7日3限

DON'T PANIC!