アルゴリズムとデータ構造入門
ソーティング・サーチ
D.E. Knuth: *The Art of Computer Programming, Vol.3: Sorting and Searching* (邦訳あり)

奥 乃 博
大学院情報学研究科知能情報学専攻
知能メディア講座 音声メディア分野
http://winnie.kuis.kyoto-u.ac.jp/~okuno/Lecture/09/IntroAlgDs/
okuno@i.kyoto-u.ac.jp

試験は2月2日(火)3限 8号館3階大会議室

1月18日・本日のメニュー

1. Internal Sorting(内部整列)
   1. 挿入ソート(insertion sort)
   2. クイックソート(quick sort)
2. vector (ベクタ)によるSorting
   1. ヒープソート(heap sort)
   2. バブルソート(bubble sort)
   3. マージソート(merge sort)
3. Searching (探索)
   1. 二分探索(binary search)
   2. ハッシュ法(hashing)
4. アンケート(最後の15分)

- 講義コード: 91150
- 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。

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.

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)**

Javaによるデモ
http://winnie.kuis.kyoto-u.ac.jp/~okuno/Lecture/09/IntroAlgDs/

1月18日・本日のメニュー

1. Internal Sorting(内部整列)
   1. 挿入ソート(insertion sort)
   2. クイックソート(quick sort)
2. vector (ベクタ)によるSorting
   1. ヒープソート(heap sort)
   2. バブルソート(bubble sort)
   3. マージソート(merge sort)
3. Searching (探索)
   1. 二分探索(binary search)
   2. ハッシュ法(hashing)

挿入ソート(insert sort)

```
(define (insert-sort-pred pred records)
  (if (null? records)
      '()
      (insert-elem pred (car records)
         (insert-sort-pred pred (cdr records))
      )))
(define (insert-elem pred elem ordered-rec)
  (cond ((null? ordered-rec) (cons elem '()))
        ((pred elem (car ordered-rec))
         (cons elem ordered-rec) )
        (else
         (cons (car ordered-rec)
               (insert-elem pred elem
                     (cdr orderd-rec) )))))
(define (insert-sort records . args)
  (insert-sort-pred
     (if (null? args) > (car args)))
     records ))
```

挿入ソート(insert sort)の実行トレース

```
(insert-sort-pred > '(2 3 1 6 4))
                    ↑
2
3  2
1
6  3  2  1
   4  3  2  1
```


挿入ソート(insert sort)の実行トレース

```
1. (insert-sort-pred > '(2 3 1 6 4))
2. (insert-elem > 2
     (insert-sort-pred > '(3 1 6 4)) )
3. (insert-elem > 2
     (insert-elem > 3
       (insert-sort-pred > '(1 6 4)) ))
4. (insert-elem > 2
     (insert-elem > 3
       (insert-elem > 1
         (insert-sort-pred > '(6 4)) )))
5. (insert-elem > 2
     (insert-elem > 3
       (insert-elem > 1
         (insert-elem > 6
           (insert-sort-pred > '(4)) ))))
6. (insert-elem > 2
     (insert-elem > 3
       (insert-elem > 1
         (insert-elem > 6
           (insert-elem > 4
             (insert-sort-pred > ()) ))))
```


挿入ソート(insert sort)の実行トレース2

```
7. (insert-elem > 2
     (insert-elem > 3
       (insert-elem > 1
         (insert-elem > 6
           (insert-elem > 4 '()) ))))
8. (insert-elem > 2
     (insert-elem > 3
       (insert-elem > 1
         (insert-elem > 6 '(4)) )))
9. (insert-elem > 2
     (insert-elem > 3
       (insert-elem > 1 '(6 4)) ))
10. (insert-elem > 2
      (insert-elem > 3
        (cons 6 (insert-elem > 1 '(4))) ))
11. (insert-elem > 2
      (insert-elem > 3
        (cons 6 (cons 4 (insert-elem > 1 '()))) ))
12. (insert-elem > 2
      (insert-elem > 3
        (cons 6 (cons 4 '(1))) ))
```


挿入ソート(insert sort)の実行トレース3

```
13. (insert-elem > 2
      (insert-elem > 3
        (cons 6 (cons 4 '(1))) ))
14. (insert-elem > 2
      (insert-elem > 3 '(6 4 1)) )
15. (insert-elem > 2
      (cons 6 (insert-elem > 3 '(4 1))) )
16. (insert-elem > 2
      (cons 6 (cons 4 (insert-elem > 3 '(1)))) )
17. (insert-elem > 2 '(6 4 3 1))
18. (cons 6 (insert-elem > 2 '(4 3 1)))
19. (cons 6 (cons 4 (insert-elem > 2 '(3 1))))
20. (cons 6 (cons 4 (cons 3 (insert-elem > 2 '(1)))))
21. (cons 6 (cons 4 (cons 3 (cons 2 '(1)))))
22. (6 4 3 2 1)
```


挿入ソート(insert sort)の計算量

1. 最悪の場合(*worst case*) (predが≧とする)
   - 大きなものから順に入ってくる
   - 比較回数は $\sum_{i=1}^{n}(i-1)=\frac{1}{2}n(n-1)$ $\Theta(n^2)$
2. 最良の場合(*best case*)
   - 小さいものから順に入ってくる
   - 比較回数は $\sum_{i=2}^{n}1=(n-1)$ $\Theta(n)$
3. 平均の場合(*average case*)
   - すでに入っているデータの半分だけ比較
   - 比較回数は $\sum_{i=1}^{n}\frac{1}{2}(i-1)=\frac{1}{4}n(n-1)$ $\Theta(n^2)$

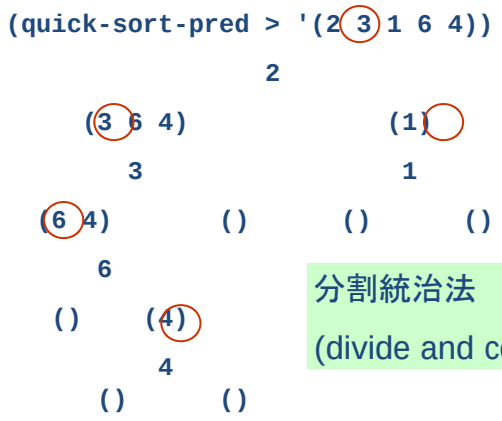
1月18日・本日のメニュー

1. Internal Sorting(内部整列)
   1. 挿入ソート(insertion sort)
   2. クイックソート(quick sort)
2. vector (ベクタ)によるSorting
   1. ヒープソート(heap sort)
   2. バブルソート(bubble sort)
   3. マージソート(merge sort)
3. Searching (探索)
   1. 二分探索(binary search)
   2. ハッシュ法(hashing)

クイックソート(quick sort)

```
(define (quick-sort records . args)
  (quick-sort-pred (if (null? args) > (car args))
                   records ))
(define (quick-sort-pred pred records)
   (if (null? records)
       '()
       (let* ((pivot (car records))
              (division (partition pred pivot
                                   (cdr records) '() '() )))
         (append (quick-sort-pred pred (car division))
                 (cons pivot
                       (quick-sort-pred pred
                                        (cdr division) ))))))
(define (partition pred pivot records left right)
  (cond ((null? records) (cons left right))
        ((pred pivot (car records))
         (partition pred pivot (cdr records) left
                    (cons (car records) right) ))
        (else
         (partition pred pivot (cdr records)
                    (cons (car records) left) right ))))
```


quick sortの実行トレース(まとめ)



分割統治法

(divide and conquor)

quick sortの実行トレース1

```
1.  (quick-sort-pred > '(2 3 1 6 4))
2.  (partition > 2 '(3 1 6 4) '() '())
      ((3 6 4) (1))
3.  (append (quick-sort-pred > '(3 6 4))
        (cons 2 (quick-sort-pred > '(1)) )

3-1. (partition > 3 '(6 4) '() '())
       ((6 4) ())
3-2. (append (quick-sort-pred > '(6 4))
         (cons 3 ()) )
4-1.    (partition > 6 '(4) '() '())
          (() (4))
4-2.    (append ()
            (cons 6 (quick-sort-pred > '(4))) )
5-1.       (partition > 4 '() '() '())
             (() ())
4-3.    (append () (cons 6 (append () (cons 4 ()))))
          (6 4)
4-4.    (6 4)
```


quick sortの実行トレース2

```
3-1.   (partition > 3 '(6 4) '() '())
         ((6 4) ())
3-2.   (append (quick-sort-pred > '(6 4))
           (cons 3 ()) )
3-3.   (append '(6 4) (3))
         (6 4 3)
3-4. (append '(6 4 3)
           (cons 2 (quick-sort-pred > '(1)) )
4-1.    (partition > 1 '() '() '())
4-2. (append '(6 4 3)
           (cons 2 '(1)) )
     (6 4 3 2 1)
```


クイックソート(quick sort)の計算量

1. 最悪の場合(*worst case*) (predが≧とする)
   - 小さいものから順に入ってくる
   - partitionでの走査回数は

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

2. 最良の場合(*best case*)
   - 分割がバランスしている
   - partitionの呼ばれる回数は

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

3. 平均の場合(*average case*)

クイックソート(quick sort)の計算量

3. 平均の場合(*average case*)
   - データはすべて異なる。あらゆる順列が等確率。
   - 途中での分割でも同様の仮定が成立するとする。
- *n*要素のquick sort に要する時間: $T(n)$ とする
- 左右の結果の統合に要する時間: cn (c: 定数)
- *n*要素が*i*要素と*n-i-1*要素に分割されたとすると

$T(n) \leqq T(i)+T(n-i-1)+cn$

$$T(n) \le T(i)+T(n-i-1)+cn$$

クイックソート(quick sort)の計算量

- n要素の分割、$(0,n-1), (1, n-2),\ldots, (n-1,0)$ が等確率で生ずるとすると、次の漸化式を得る

$$T(n) = cn + \frac{1}{n}\sum_{i=0}^{n-1} T(i) \qquad (n \geq 2)$$

$$T(0) = 0$$

$$T(1) = 1$$

- 帰納法で証明すると

$$T(n) \approx 2n \log n \qquad \Theta(n \log n)$$

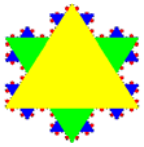
1月18日・本日のメニュー

1. Internal Sorting(内部整列)
   1. 挿入ソート(insertion sort)
   2. クイックソート(quick sort)
2. vector (ベクタ)によるSorting
   1. ヒープソート(heap sort)
   2. バブルソート(bubble sort)
   3. マージソート(merge sort)
3. Searching (探索)
   1. 二分探索(binary search)
   2. ハッシュ法(hashing)

Jakldで色を付ける

```
(define (wind-mill painter colors)
  (lambda (frame)
    (set-color (car colors))
    (painter frame)
    (set-color (cadr colors))
    ((rotate90 painter) frame)
    (set-color (caddr colors))
    ((rotate180 painter) frame)
    (set-color (cadddr colors))
    ((rotate270 painter) frame) ))

(define (moulin painter colors)
  (lambda (frame)
    (cond ((pair? colors)
      (set-color (car colors))
      (painter frame)
      ((moulin (rotate90 painter) (cdr colors))
   frame) ))))
(moulin color-lambda '(red blue green yellow))
```



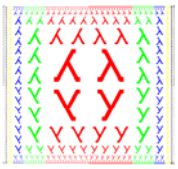
Square-limit に色を付ける(未完)

```
(define (right-split painter n colors)
  (lambda (frame)
    (if (= n 0)
        (painter frame)
        (let ((smaller (right-split painter (- n 1) (cdr colors))) )
          (if (pair? colors) (set-color (car colors)))
          ((beside painter below smaller smaller)) frame) ))))

(define (corner-split painter n colors)
  (lambda (frame)
    (if (= n 0)
        (painter frame)
        (let ((up (up-split painter (- n 1)))
              (right (right-split painter (- n 1) (cdr colors))) )
          (let ((top-left (beside up up))
                (bottom-right (below right right))
                (corner (corner-split painter (- n 1) (cdr colors))) )
            (if (pair? colors) (set-color (car colors)))
            ((beside (below painter top-left)
                     (below bottom-right corner) )
             frame ))))))

(define (square-limit painter n colors)
  (let ((quarter (corner-split painter n colors)))
    (let ((half (beside flip-horiz quarter) quarter)))
      (below (flip-vert half) half)) ))

((square-limit color-letterlambda 5
     '(red green blue yellow black) )
```



reverseの活用例:回文(palindrome)

```
(define (palindrome? chars)
  (equal? chars (reverse chars) ) )

(define (make-dalindrome chars)
  (define (make-even-palindrome chars)
      (append chars (reverse chars)) )
  (define (make-odd-palindrome chars)
      (append chars (cdr (reverse chars))) )

 (palindrome? '(shi n bu n shi))
 (palindrome? '(ta ke ya bu ya ke ta))
 (palindrome? '(M A D A M I M A D A M))

(define (last-but-one items)
  (reverse (cdr (reverse items))) )
(last-but-one '(shi n bu n shi))
   ⇒ (shi n bu n)
```


1月18日・本日のメニュー

1. Internal Sorting(内部整列)
   1. 挿入ソート(insertion sort)
   2. クイックソート(quick sort)
2. vector (ベクタ)によるSorting
   1. ヒープソート(heap sort)
   2. バブルソート(bubble sort)
   3. マージソート(merge sort)
3. Searching (探索)
   1. 二分探索(binary search)
   2. ハッシュ法(hashing)

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〉)

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)

ベクタ(vector)の例

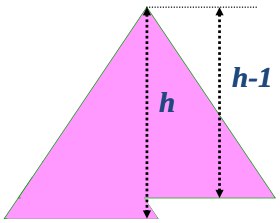
```
(define y (make-vector 5))
   #(() () () () ())
(define x #(1 2 3 4 5))
   #(1 2 3 4 5)
(vector-length x)      5
(vector-ref x 2)       3
(vector-set! x 3 128)
   #(1 2 3 128 5)
x                #(1 2 3 128 5)
(vector-set! x 0 'foo)
   #(foo 2 3 128 5)
```


1月18日・本日のメニュー

1. Internal Sorting(内部整列)
   1. 挿入ソート(insertion sort)
   2. クイックソート(quick sort)
2. vector (ベクタ)によるSorting
   1. ヒープソート(heap sort)
   2. バブルソート(bubble sort)
   3. マージソート(merge sort)
3. Searching (探索)
   1. 二分探索(binary search)
   2. ハッシュ法(hashing)

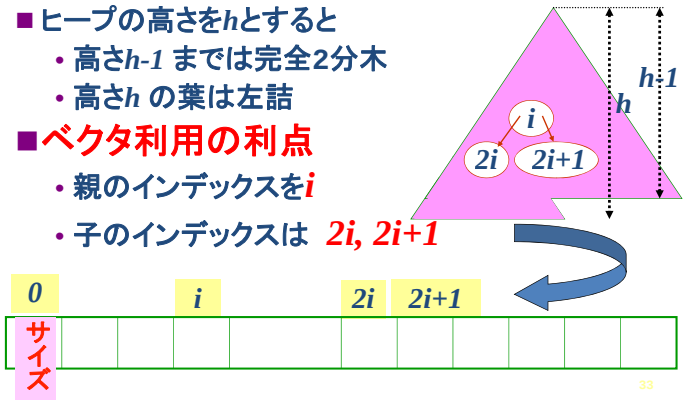
ヒープ(heap)というデータ構造

- 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



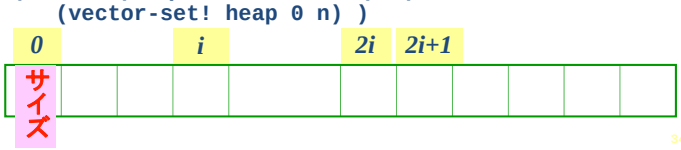
ヒープ(heap)のベクタ表現

- 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
- ベクタ利用の利点
  - 親のインデックスを*i*
  - 子のインデックスは *2i, 2i+1*



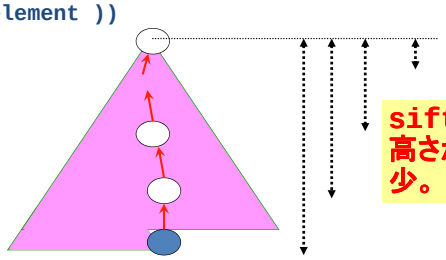
ピープの諸演算・手続き

```
(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（ピープに要素挿入）

```
(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。

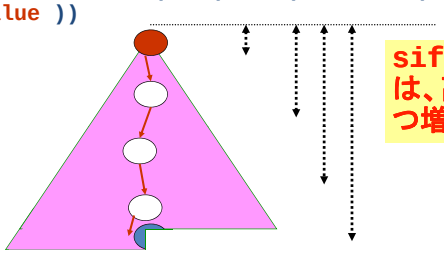
sift-up(heap修復)

```
(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。

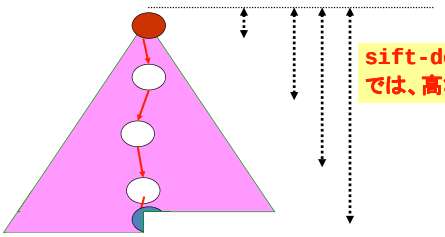
sift-down(heap修復)

```
(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))))))))
```


heap-sort の計算量

- 1回繰り返すと高さが1増加
- 要素数を n とすると計算量は

$$\Theta(\log n)$$



sift-downward では、高さが1ずつ増加。

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 $\Theta(n \log n)$

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)*

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

- `sift-down` では
  - 実行前: *heap(1,n)* の成立は？
  - 実行後: *heap(1,3)* が成立。
- *k*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。

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)*

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

- `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？
- *k*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)* ?

1月18日・本日のメニュー

1. Internal Sorting(内部整列)
   1. 挿入ソート(insertion sort)
   2. クイックソート(quick sort)
2. vector (ベクタ)によるSorting
   1. ヒープソート(heap sort)
   2. バブルソート(bubble sort)
   3. マージソート(merge sort)
3. Searching (探索)
   1. 二分探索(binary search)
   2. ハッシュ法(hashing)

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) ))))))
```


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 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
```

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

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

$$\Theta(n^2)$$

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

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

挿入ソート(insert sort)の計算量

1. 最悪の場合(*worst case*) (predが≧とする)
   - 小さいものから順に入ってくる
   - 比較回数は $\sum_{i=1}^{n}(i-1)=\frac{1}{2}n(n-1)$ $\Theta(n^2)$
2. 最良の場合(*best case*)
   - 大きなものから順に入ってくる
   - 比較回数は $\sum_{i=2}^{n}1=(n-1)$ $\Theta(n)$
3. 平均の場合(*average case*)
   - すでに入っているデータの半分だけ比較
   - 比較回数は $\sum_{i=1}^{n}\frac{1}{2}(i-1)=\frac{1}{4}n(n-1)$ $\Theta(n^2)$

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

$$\Theta(m+n)$$

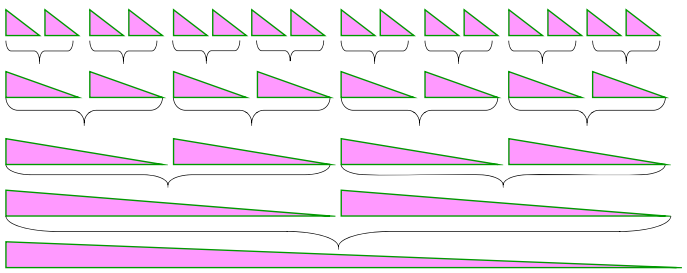
- 空間計算量も余分に $\Theta(m+n)$

1月18日・本日のメニュー

1. Internal Sorting(内部整列)
   1. 挿入ソート(insertion sort)
   2. クイックソート(quick sort)
2. vector (ベクタ)によるSorting
   1. ヒープソート(heap sort)
   2. バブルソート(bubble sort)
   3. マージソート(merge sort)
3. Searching (探索)
   1. 二分探索(binary search)
   2. ハッシュ法(hashing)

内部マージソート(In-place merge sort)

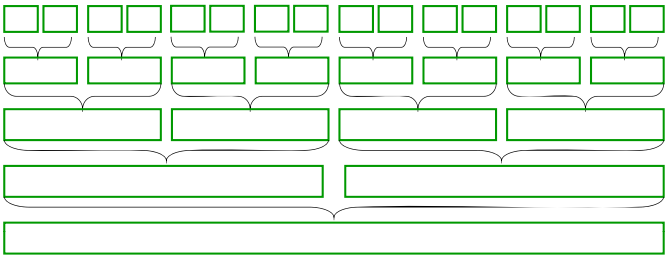
分割統治型



ラン(run)と言う $\Theta(n\log n)$

内部マージソート(In-place merge sort)

分割統治型(ベクタの場合)



$\Theta(n \log n)$

52

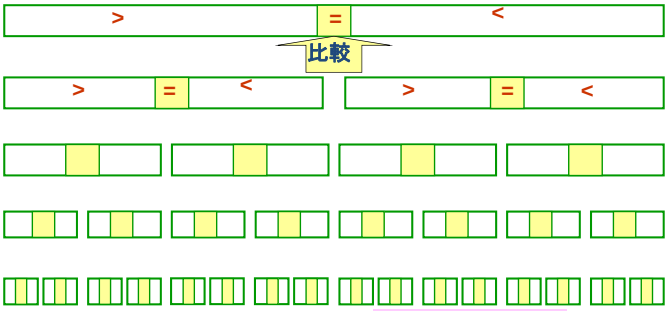
Sorting(整列)のまとめ

- 選択ソート (selection sort) $\Theta(n^2)$
- 挿入ソート (insertion sort) 定$\Theta(n^2)$
- シェルソート (Shell sort)
  - $h_k = 3h_{k-1} + 1, \ldots, 1$ の時 $\Theta(n^{1.25})$
- クイックソート (quick sort),分割統治法(divide 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 $\Theta(n \log n)$

53

Sort(整列)済ベクタの探索

二分探索(binary search)



二分探索の計算量 $\Theta(\log n)$

56

1月18日・本日のメニュー

1. Internal Sorting(内部整列)
   1. 挿入ソート(insertion sort)
   2. クイックソート(quick sort)
2. vector (ベクタ)によるSorting
   1. ヒープソート(heap sort)
   2. バブルソート(bubble sort)
   3. マージソート(merge sort)
3. Searching (探索)
   1. 二分探索(binary search)
   2. ハッシュ法(hashing)

ハッシュ法(hashing)

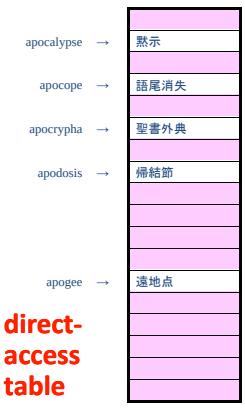
- 探したいデータの範囲膨大
- 例: 最大10文字の単語
- 50文字とすると組合わせの数は 50^{10}

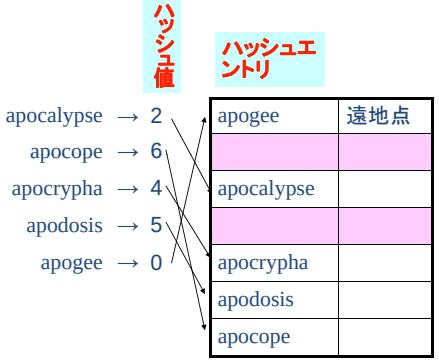
  $\log(50^{10}) = 10(2 - \log 2) \cong 17$ 10^{17}
- ところが実際の単語数は高々 10^6
- ベクタ(配列)で単語を管理すると疎
- すかすかの配列
- ハッシュ法(hashing)を使用

59

辞書とハッシュ表

empty(空)





60

ハヤシライスを知っていますか

- hashed beaf
- 語源は同じ

61

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)

62

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

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

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

63

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

64

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), …

65

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 $\Theta(N)$

66

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	

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)

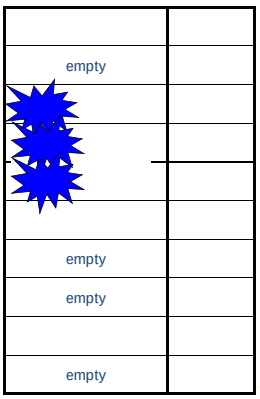
- ハッシュ関数 h, g
- 衝突発生時
  $h_i \equiv h+ig \bmod M$
- 挿入
  - empty/deleted というフラグのあるエントリに入れる
- 検索
  - empty/deletedまで探す。
- 削除
  - deletedというフラグを立てる。

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

h(dog)=2
h(Kyoto)=4
h(Univ)=0
h(Informatics)=2
h(SICP)=3
h(test)=8

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



線形走査法の挿入の計算量

1. n個のデータが格納済、$n+1$個目のデータ挿入時に$h_i(x)$がi-1回目まですべて詰まっている確率

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

2. 空きセルを見つけるまでの比較回数

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

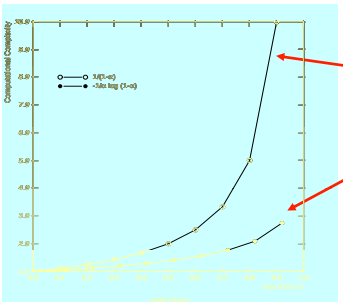
3. ハッシュ表にN個のデータを挿入する手間は

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

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

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



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

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

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

線形走査法の削除の計算量

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

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

期末テスト, 必修課題, 随意課題

- 期末テスト、健闘を期待します。
- 必修課題2の締切は2月11日13時。
- 随意課題の提出でランクアップを。

期末テスト:1月25日(火)3限 3階大会議室

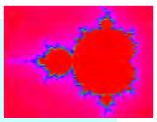
随意課題: 2月28日午後24時締切

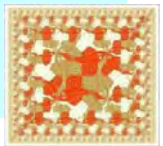
挑戦的な作品を期待します

- 図形, フラクタル, 音楽, Lego
- アルゴリズム可視化,

プログラムはメイルで

okuno@i.kyoto-u.ac.jp





アンケート 講義コード: 91150

- 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.

17. Intermissionには興味が持てましたか.
18. 講義HPは毎回見ましたか.
19. 講義HPは役に立ちましたか.
20. TAのページは毎回見ましたか.
21. TA掲示板は活用されましたか.
22. TAのページ・掲示板は役に立ちましたか.
23. 2名のTAは役に立ちましたか.

アンケート　講義コード：91150

24. KULASISのページは毎回見ましたか.
25. KULASISのページは役に立ちましたか.
26. 階乗の再帰/繰り返しプログラムは書けますか.
27. Fibonacci数の再帰/繰り返しプログラムは?
28. Sequence と accumulate, filter, map, enumerate は理解できましたか.
29. 手続き抽象化は理解できましたか.
30. データ抽象化は理解できましたか.