情報科学基礎論-3 2007年4月24日

情報科学基礎論

3. アルゴリズムとデータ構造

Algorithms and Data Structures

奥 乃 博

data structure（データ構造）
1. 配列
2. リスト
3. 木構造
4. ハッシング

Sorting（整列）
1. 内部整列
2. 外部整列

http://winnie.kuis.kyoto-u.ac.jp/~okuno/Lecture/07/IntroCS/

アルゴリズム（Algorithm）

入力 → アルゴリズム（有限回の計算ステップ） → 出力

- 停止性・正当性
- 計算量（complexity）
  - 時間計算量、空間計算量
- O記法（上限）、Ω記法（下限）、Θ記法（上下限）

3

データ構造（Data Structure）

- **データの有限集合、しかも、動的に要素の数が変化する集合を扱うための表現**
  - 待ち行列、スタック
- **データ構造の抽象化、演算の抽象化**
  - オブジェクトベースド：
    データ構造と演算のカプセル化
  - クラスベースド：
    クラスとインスタンスの区別
  - オブジェクト指向： クラス間の階層構造

4

データ構造：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〉)

5

データ構造：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)

6

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)
```

7

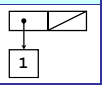
データ構造：配列（array）

- N次元にベクタを拡張
- 2次元の場合　$X=[x_{ij}]$
- 3次元の場合　$X=[x_{ijk}]$
- 基本データ構造
  - 待ち行列（キュー、queue）
    FIFO（First-In First-Out）
    enqueue, dequeue
  - スタック（stack）
    LIFO（Last-In First-Out）
    push, pop

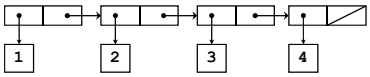
8

データ構造： リスト（list）

- 対（pair）、2元セル
- リスト、並び（sequence）
- (cons 1 nil)
- (list 1)と同じ
- nilは空リスト

	(cons 1 2)	(cons 1 nil)
ドット記法（dotted notation）	(1 . 2)	(1 . nil) または (1)
箱ポインタ記法（box-and-pointer notation）		

- (list 1 2 3 4)



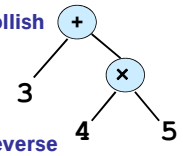
(1 2 3 4)　あるいは
(1 . (2 . (3 . (4 . nil))))

- (cons 1
  (cons 2
  (cons 3
  (cons 4 nil))))

9

前置記法（prefix notation）

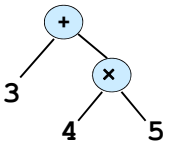
- 式 （演算子　被演算子 ...）
  operator　operands
- 式の記法
  - 前置記法 （prefix notation , Pollish notation, ポーランド記法）
    + 3 * 4 5
  - 中置記法 （infix notation）
    3 + (4 * 5)
  - 後置記法 （postfix notation, reverse Pollish notation、逆ポーランド記法）
    3 4 5 * +
- 木表現はどれも同じ



10

木の辿り方から3つの記法への変換

- 木の辿り方
  - 前順走査 （pre-order traversal）
    ノード⇒左部分木⇒右部分木
    + ⇒ 3 ⇒ * ⇒ 4 ⇒ 5
  - 間順走査 （in-order tr.）
    左部分木⇒ノード⇒右部分木
    3 ⇒ + ⇒ 4 ⇒ * ⇒ 5
  - 後順走査 （post-order tr.）
    左部分木⇒右部分木⇒ノード
    3 ⇒ 4 ⇒ 5 ⇒ * ⇒ +



11

データ構造： 木（tree）

- 木
  - 節、節点（node）、頂点（vertex ）
  - 枝（edge, branch）、辺（arc）
- 根付き木（rooted tree）
  - 根（root）、節、葉（leaf）
  - 節の深さ（depth）：
    根からその節までの節数
  - 木の高さ（height）：
    その木に含まれる最大の節の深さ

12

データ構造： 木（tree）　（続）

- 有向木（directed tree）
  - 親（parent）、子（child）、兄弟（sibling）
  - 祖先（ancestor）、子孫（decedent）
- 順序木（ordered tree）
  - 根付き木（rooted tree）かつ
  - 子（兄弟）に順序付け

13

4月24日・本日のメニュー

- data structure（データ構造）
  1. 配列
  2. リスト
  3. 木構造
  4. ハッシング
- Sorting（整列）
  1. 内部整列
  2. 外部整列

14

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。

15

Internal Sorting（内部整列）

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

16

安定整列（stable sorting）

- 同じデータ間のもともとの順序が整列後も保存されている整列のこと。
- 基数ソート（radix sort）では重要な性質。
- 辞書式順序で整列で基数ソートが使われる。

17

挿入ソート（insert sort）

1．データを前から見て行き、適切な場所に入れる。
2．すべてのデータに対して繰り返す。

例：　2、3、1、6、4

18

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 ))
```

19

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
```

20

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 $\sum_{i=1}^{n}(i-1)=\frac{1}{2}n(n-1)$ $O(n^2)$
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 $\sum_{i=2}^{n}1=(n-1)$ $O(n)$
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 $\sum_{i=1}^{n}\frac{1}{2}(i-1)=\frac{1}{4}n(n-1)$ $O(n^2)$

24

クイックソート(quick sort)

1. データが1個ならば整列終了。
2. データの中からpivot(枢軸)を選ぶ。
3. 残りのデータのpivotでそれより大きいグループと小さいグループに分割。
4. それぞれのグループについて整列。
5. 大きいグループの整列結果、pivot、小さいグループの整列結果を順に並べる。

例: 2、3、1、6、4

25

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 ))))
```

29

quick sortの実行トレース(まとめ)

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

分割統治法

(divide and conquor)

27

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 $\sum_{i=1}^{n}(n-i)=n^2-\frac{1}{2}n(n+1)=\frac{1}{2}n(n-1)$ $O(n^2)$
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 $\log n$ $O(n\log n)$
3. 平均の場合(*average case*)

30

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

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

31

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),…, (n-1,0)* が等確率で生ずるとすると、次の漸化式を得る

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

$$T(1) = 0$$

$$T(0) = 1$$

■ 帰納法で証明すると

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

32

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、$O(n \log n)$
- 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。

35

ピープ（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

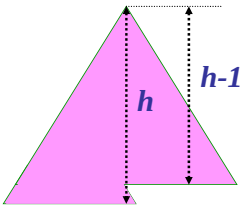
1．高さ*h-1* までは完全2分木

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

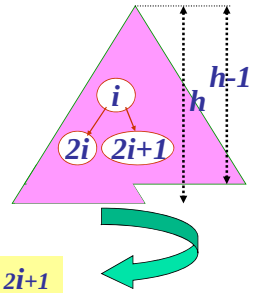
37

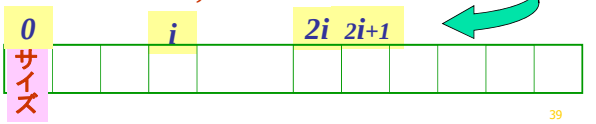
ピープ（heap）のベクタ表現

■ 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

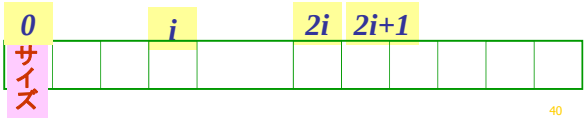




39

ピープの諸演算・手続き

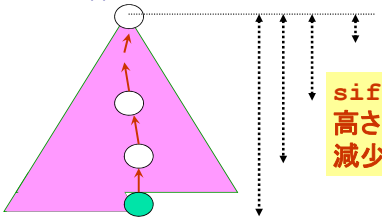
```
(define (make-heap maxsize)
  (let ((heap (make-vector (+ maxsize 1))))
    (vector-set! heap 0 0)
    heap )))
(define (heap-size heap) (vector-ref heap 0))
(define (heap-left-child n) (* n 2))
(define (heap-right-child n) (+ (* n 2) 1))
(define (heap-parent n) (quotient n 2))
(define (heap-top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



40

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。

41

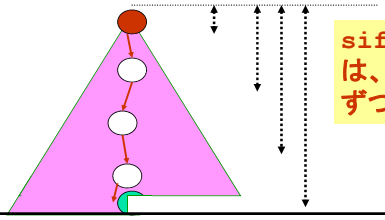
sift-up（heap修復）

```
(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 $O(\log n)$

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。

43

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 ))))))))
```

44

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 $O(n \log n)$

46

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。

47

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)* ?

48

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) ))))))
```

50

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
```

51

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

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

$$O(n^2)$$

52

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

$$O(n^{1.25})$$

53

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

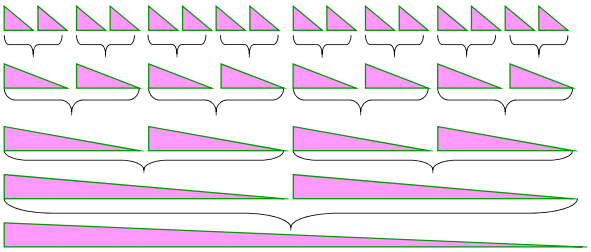
$$O(m+n)$$

■ 空間計算量も余分に $O(m+n)$

54

内部マージソート（In-place merge sort）

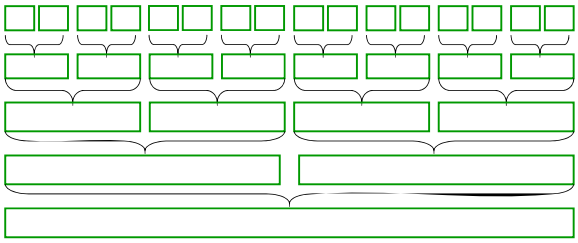
- 分割統治型



ラン（run）と言う　$O(n \log n)$

内部マージソート（In-place merge sort）

- 分割統治型（ベクタの場合）



$O(n \log n)$

56

Sorting（整列）のまとめ

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

57

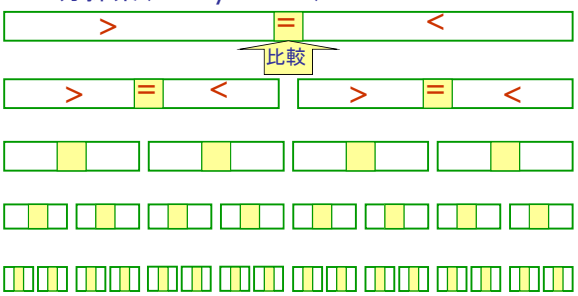
Sorting（整列）のデモ

- Java のデモプログラム
- http://winnie.kuis.kyoto-u.ac.jp/~okuno/Lecture/07/IntroAlgDs/

58

Sort（整列）済ベクタの探索

- 2分探索（binary search）



- 2分探索の計算量　$O(\log n)$

60

4月24日・本日のメニュー

- **Sorting（整列）**
  1. 内部整列
  2. 外部整列
- **data structure（データ構造）**
  1. 配列
  2. リスト
  3. 木構造
  4. ハッシング

63

ハッシュ法（hashing）

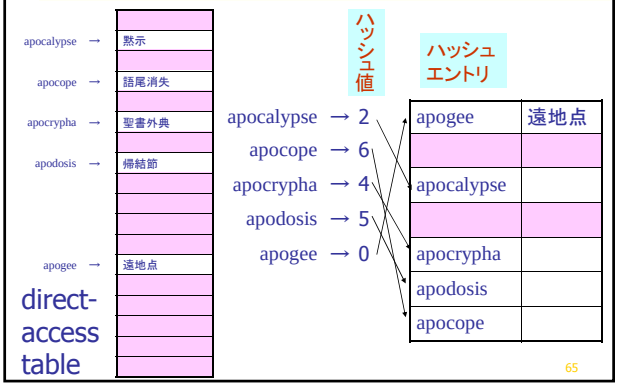
- 探したいデータの範囲膨大
- 例： 最大10文字の単語
- 50文字とすると組合わせの数は 50^{10}

  $\log(50^{10}) = 10(2 - \log 2) \cong 17$ 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

64

辞書とハッシュ表

empty（空）



65

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）

67

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 $MK^2 \cong N^2$

68

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

69

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）

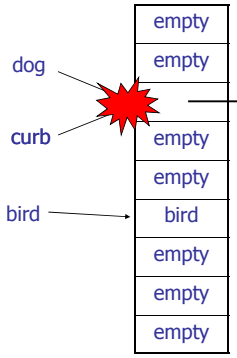
     h, gとすると、

     $h(x), h(x)+g(x), h(x)+2g(x), h(x)+3g(x), \ldots$

70

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 $\Theta(N)$

71

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	

72

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	

73

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	

74

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	

75

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

76

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

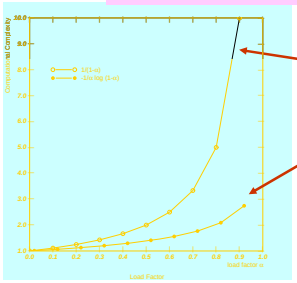
$$\sum_{n=0}^{N}\frac{M}{M-n}\cong\int_0^N\frac{M}{M-x}dx=M\log_e\frac{M}{M-N+1}$$

77

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

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



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

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

78

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

79

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

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

81

ハッシュ表というデータ構造

```
(define (quick-sort records . args)
  (quick-sort-pred (if (null? args) > (car args))
                   records ))
(define (quick-sort-pred pred records)
   (if (null? records)
       nil
       (let* ((pivot (car records))
              (division (partition pred pivot (cdr records) nil nil)) )
         (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 ))))
(define (insert-sort records . args)
  (insert-sort-pred
     (if (null? args) > (car args)))
    records ))
```

82

レポート課題：5月21日午後5時締切

どれか1つアルゴリズムを選び、次の課題を行え。

1. アルゴリズムを説明せよ。
   - 今日習ったアルゴリズム
   - ご自身の研究に関係するアルゴリズム
2. アルゴリズムの挙動を表示（可視化）するプログラムを作成せよ。
   MATLAB、Javaなどで。

レポートとプログラムはメイルで提出のこと

okuno@i.kyoto-u.ac.jp

83