データ構造とアルゴリズム
（第6回）

モバイルコンピューティング研究室
柴田史久

1

本日の講義内容

- 探索（１）
  - 探索とは
  - 線形探索と二分探索
  - ハッシュ法
    - ハッシュ法の原理
    - チェイン法
    - オープンアドレス法
  - ハッシュ関数，他

2

教科書 第7章（pp.183～185）

探索とは？
線形探索･二分探索

3

探索とは

- 表の中からある特定の値を持つデータを探すこと
  - Ex.：名前をもとに個人データを探す
  - 通常は「与えられた値と一致する」という条件が対象
- 探索は以下の機能から構成される
  - 挿入：　データを表に登録する
  - 探索：　与えられた値をキーにもつデータを探す
  - 削除：　与えられた値をキーに持つデータを削除する
- これらの機能を備えた抽象データ型は「辞書」

4

線形探索法

```
Entry[] table = new Entry[MAX];
int n = 0;  // n は登録データ数

public Object search(int key)
{
  int i = 0;                    // (1)
  while (i < n) {               // (2)
    if (table[i].key == key)    // (3)
      return (table[i].data);   // (4)
    i++;                        // (5)
  }
  return null;                  // (6)
}
```

```
class Entry {
  int    key ;
  Object data ;
}
```

i = 0　i = 1　i = 2　i = 3　i = 4

配列table の添字	0	1	2	3	4
key	1	10	2	4	6
data	One	Ten	Two	Four	Six

5

二分探索法(1)

```
Entry[] table = new Entry[MAX];
int n = 0;  // n は登録データ数

public Object search(int key)
{
  int low = 0;                          // (1)
  int high = n - 1;                     // (2)
  while (low <= high) {                 // (3)
    int middle = (low + high) / 2;      // (4)
    if (key == table[middle].key)       // (5)
      return table[middle].data;        // (6)
    else if (key < table[middle].key)   // (7)
      high = middle - 1;                // (8)
    else  // key > table[middle].key であある
      low = middle + 1;                 // (9)
  }
  return null;                          // (10)
}
```

```
class Entry {
  int    key ;
  Object data ;
}
```

	key	
low →	1	table[0]
	3	table[1]
	4	table[2]
	8	table[3]
middle →	13	table[4] < 14
	14	table[5]
	18	table[6]
	20	table[7]
	21	table[8]
high →	25	table[9]

key = 14 のデータを二分探索法で探す
low = 0, high = 9, middle = (0+9) / 2 = 4
着目範囲を後半へ

6

二分探索法(2)

```
Entry[] table = new Entry[MAX];
int n = 0;  // n は登録データ数

public Object search(int key)
{
  int low = 0;                          // (1)
  int high = n - 1;                     // (2)
  while (low <= high) {                 // (3)
    int middle = (low + high) / 2;      // (4)
    if (key == table[middle].key)       // (5)
      return table[middle].data;        // (6)
    else if (key < table[middle].key)   // (7)
      high = middle - 1;                // (8)
    else  // key > table[middle].key であある
      low = middle + 1;                 // (9)
  }
  return null;                          // (10)
}
```

```
class Entry {
  int    key ;
  Object data ;
}
```

	key	
	1	table[0]
	3	table[1]
	4	table[2]
	8	table[3]
	13	table[4]
low →	14	table[5]
	18	table[6]
middle →	20	table[7] > 14
	21	table[8]
high →	25	table[9]

low = 5, high = 9, middle = (5+9) / 2 = 7

着目範囲を前半へ

7

二分探索法(3)

```
Entry[] table = new Entry[MAX];
int n = 0;  // n は登録データ数

public Object search(int key)
{
  int low = 0;                          // (1)
  int high = n - 1;                     // (2)
  while (low <= high) {                 // (3)
    int middle = (low + high) / 2;      // (4)
    if (key == table[middle].key)       // (5)
      return table[middle].data;        // (6)
    else if (key < table[middle].key)   // (7)
      high = middle - 1;                // (8)
    else  // key > table[middle].key であある
      low = middle + 1;                 // (9)
  }
  return null;                          // (10)
}
```

```
class Entry {
  int    key ;
  Object data ;
}
```

	key	
	1	table[0]
	3	table[1]
	4	table[2]
	8	table[3]
	13	table[4]
low, middle →	14	table[5] = 14
high →	18	table[6]
	20	table[7]
	21	table[8]
	25	table[9]

low = 5, high = 6, middle = (5+6) / 2 = 5

key = 14 と table[5] が一致　→　データが見つかった

8

線形探索法と二分探索法

計算量（1回あたり）	線形探索法	二分探索法
挿入	$O(1)$	$O(n)$
探索	$O(n)$	$O(\log n)$
削除	$O(n)$	$O(n)$

9

教科書 第8章（pp.186～190）

ハッシュ法

ハッシュ法の原理

10

ハッシュ法(hashing)

- データ量によらず挿入，探索，削除の計算量が $O(1)$
- キーの値をデータの格納位置に直接関連付ける
  - データの格納位置＝配列の添字

11

ハッシュ法の原理(1)

- 一連のデータ
  - キーの範囲を制限：0～99
  - キーの重複はない
  - データは例えば文字列（別になんでもいい）
- このデータを表に登録

```
class Entry {
  int    key ;
  Object data ;
}
```

key	data
0	Zero
1	One
2	Two
3	Three
4	Four
5	Five
6	Six
7	Seven
8	Eight
9	Nine
⋮	⋮

- 大きさ100の配列xを準備
  - `Entry[] x = new Entry[100];`
- キーの値がnのデータはx[n]に登録
- 挿入，探索，削除は当然 $O(1)$

12

ハッシュ法の原理(2)

- データがセットされているか否かを区別するには？
  - フラグを持たせる方式
    - boolean[] occupied = new boolean[100];
    - MyData[] data = new MyData[100];
    - occupied[n] が true なら data[n] にデータあり
  - データとして現れない値をフラグに
    - 参照型なら null を利用
    - double 型なら Double.NaN を利用

13

ハッシュ法の原理(3)

- キーの範囲の制限をなくすには？
  - キーの値を配列の添字の範囲に写像する関数h(x)を導入
  - 関数h(x)をハッシュ関数（hash function）と呼ぶ
  - ハッシュ関数が返す値はハッシュ値（hash value）
- ハッシュ関数を利用して配列にデータを格納
  - キーの値がaならば，table[h(a)] にデータを格納
  - ハッシュ関数の計算量が $O(1)$ ならば，挿入・探索・削除も $O(1)$
- 用語
  - データを格納する配列：ハッシュ表（hash table）
  - ハッシュ表の各要素：バケット（bucket）

14

衝突

- 異なるキーに同じハッシュ値を生成する可能性
- 例
  - 文字列をキーとするデータを大きさ100の配列に格納
  - ハッシュ関数は文字列中の文字コードを加算し100で剰余

```
static int hash(String s)
{
  int sum =  0;
  for (int i = 0; i < s.length(); i++) {
    sum += (int)s.charAt(i);
  }
  return sum % 100;
}
```

文字列	ハッシュ値
one	22
two	46
three	36
four	44
five	26
six	40
seven	45
eight	29
nine	26
ten	27

  - fiveとnineのハッシュ値が同じ：衝突
- 衝突の対策が必要

15

衝突の対策

- 同じハッシュ値を持つデータを連結リストに格納
  - チェイン法，連鎖法（chaining）
  - 直接チェイン法（direct chaining）
  - オープンハッシュ法（open hashing）
- 別のバケットにデータを格納
  - オープンアドレス法，開番地法（open addressing）
  - クローズドハッシュ法（closed hashing）

16

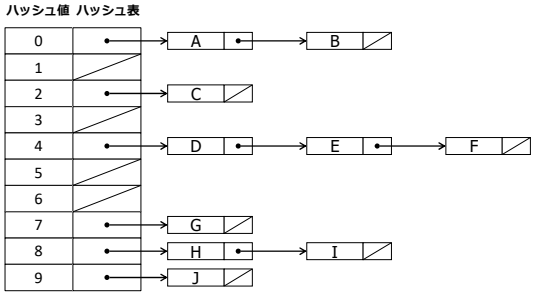
教科書 第8章（pp.190～200）

ハッシュ法
チェイン法

17

チェイン法

- 同じハッシュ値を持つデータを連結リストに格納
  - 登録：ハッシュ値からバケットを確定し，リストに登録
  - 探索：ハッシュ値からバケットを確定し，線形探索



18

チェイン法の実装(1)

- チェイン法でハッシュを実現するクラスHashC
  - 連結リストのセルを内部クラスCellとして定義
    - キーはMyKeyクラスとして定義
  - Cellクラスの配列tableがハッシュ表

詳細は教科書 pp.191～195 List 8.2

```
public class HashC {
  private static class Cell {
    MyKey  key;
    Object data;
    Cell   next;
    private Cell(MyKey key, Object data) {
      this.key  = key;
      this.data = data;
    }
  }
  Cell[] table;       // ハッシュ表
  int    bucketSize;  // バケットの個数
  int    nElements;   // 登録されているデータ数
}
```

19

チェイン法の実装(2)

- コンストラクタ
  - バケット数＝配列の要素数を指定
  - 未指定時はデフォルトの値（50）を採用

詳細は教科書 pp.191～195 List 8.2

```
public class HashC {
  static final int DEFAULT_BUCKET_SIZE = 50;
  public HashC()
  {
    this(DEFAULT_BUCKET_SIZE);
  }
  public HashC(int bucketSize)
  {
    table = new Cell[bucketSize];
    this.bucketSize = bucketSize;
    nElements = 0;
  }
}
```

20

MyKeyクラス

- 3つのメソッド
  - equals：2つのキーが等しいかどうかを返す
  - hashCode：キーのハッシュ値を返す
  - toString：キーの内容を文字列で返す

詳細は教科書 pp.196～197 List 8.3

```
public class MyKey {
  String str;  // キーとなる文字列
  public MyKey(String s) {
    str = s;
  }
  public int hashCode() {
    int sum = 0;
    for (int i = 0; i < str.length(); i++) {
      sum += (int)str.charAt(i);
    }
    return sum;
  }
}
```

合計値をそのまま返している点に注意
バケット数で割って剰余を求めるのはHashCクラス

理由：バケット数を知っているのはHashCクラス

21

チェイン法の実装(3)

- ハッシュ値の計算
  - hashメソッドでハッシュ値を計算
  - MyKeyクラスのhashCodeメソッドで得られた値をbucketSizeで割って得られた剰余を返す

詳細は教科書 pp.191～195 List 8.2

```
public class HashC {
  private int hash(MyKey key)
  {
    return key.hashCode() % bucketSize;
  }
}
```

22

チェイン法の実装(4)

- 探索
  - 探したいキーの値をパラメータとして受け取る
  - キーが含まれるはずのバケットについてリストを探索
  - キーと一致するものがなければnullを返す

詳細は教科書 pp.191～195 List 8.2

```
public class HashC {
  public Object find(MyKey key)
  {
    for (Cell p = table[hash(key)]; p != null; p = p.next) {
      if (key.equals(p.key)) {
        return p.data;
      }
    }
    return null;
  }
}
```

23

チェイン法の実装(5)

- 挿入
  - 登録するキーとデータをパラメータとして受け取る
  - キーを探索し既に登録済みかどうかをチェック
  - 登録済みでなければ連結リストに登録

詳細は教科書 pp.191～195 List 8.2

```
public class HashC {
  public boolean insert(MyKey key, Object data)
  {
    if (find(key) != null) {
      return false;  // 既にデータが登録されているので失敗(false)
    }
    Cell p = new Cell(key, data);
    int h = hash(key);
    p.next = table[h];
    table[h] = p;
    nElements++;
    return true;
  }
}
```

24

チェイン法の実装(6)

- 削除
  - 削除するキーをパラメータとして受け取る
  - 該当するキーをもったデータが存在すれば削除
  - 先頭（境界条件）を扱う必要がある

詳細は教科書 pp.191～195 List 8.2

```
public class HashC {
  public boolean delete(MyKey key) {
  h = hash(key);
  if (table[h] == null) {  // バケットが空なら失敗
    return false;
  }
  if (key.equals(table[h].key)) {  // 削除対象が先頭の場合の処理
    // 省略
  }
  // 削除対象が2番目以降の場合の処理
  Cell p, q;
  for (q = table[h], p = q.next; p!= null; q = p, p = p.next) {
    if (key.equals(p.key)) {
      // 省略
    }
  }
}
```


チェイン法の解析

- バケット数：B，データ数：N
- ハッシュ値の生成は均等と仮定
- 探索の計算量：$O(1 + N / B)$
  - ハッシュ値の計算：$O(1)$
  - リストの探索：$O(N / B)$
- 挿入の計算量：$O(1 + N / B)$
  - 挿入：$O(1)$
  - 重複チェック：$O(N / B)$
- 削除の計算量：$O(1 + N / B)$
  - 削除：$O(1)$
  - 削除対象の探索：$O(N / B)$

N に対して B が十分大きければ N / B は定数
$O(1 + N / B) = O(1)$

教科書 第8章（pp.200～213）

ハッシュ法
オープンアドレス法

オープンアドレス法

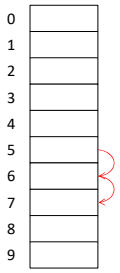
- 衝突の際にある手順で別のバケットにデータを格納
  - 手順：再ハッシュ（rehashing）
- ハッシュ表にはデータそのものを登録
  - キー x のハッシュ値 $h(x)$ のバケットが空でない場合，再ハッシュ手順にしたがって次のバケットを調べる
  - 再ハッシュを k 回行って得られるハッシュ値：$h_k(x)$
  - 再ハッシュを B 回行っても空のバケットがなければ満杯
- 探索
  - キー x のハッシュ値 $h(x)$ のバケットを調べ，キー x のデータならば探索終了
  - そうでないなら再ハッシュ手順にしたがって $h_1(x)$，$h_2(x)$，… を順に調べる

再ハッシュの手順

- キーxの所属すべきバケットから順に調べる
  - $h_k(x) = (h(x) + k) \bmod B$
- バケット数 B
  - 素数 or $2^n - 1$

オープンアドレス法の動作

- a～eの順番でハッシュ表に登録
  - バケットに空きがなければ再ハッシュ



キー	ハッシュ値
a	1
b	5
c	8
d	5
e	6
f	6

削除の手順

- データdを削除する
  - 単純に削除するだけではダメ
  - 削除済みを記録する

0	
1	a
2	
3	
4	
5	b
6	削除済み ← 見つからない
7	e ← 本当はここに
8	c
9	

キー	ハッシュ値
a	1
b	5
c	8
d	5
e	6
f	6

31

オープンアドレス法の実装(1)

- オープンアドレス法でハッシュを実現するクラス HashOA
  - バケットを内部クラスBucketとして定義
    - キーはMyKeyクラスとして定義
  - Bucketクラスの配列tableがハッシュ表

詳細は教科書 pp.205～209 List 8.4

```
public class HashOA {
  private static class Bucket {
    MyKey  key;
    Object data;
    private Bucket(MyKey key, Object data) {
      this.key  = key;
      this.data = data;
    }
  }
  Bucket[] table;       // ハッシュ表
  int      bucketSize;  // バケットの個数
  int      nElements;   // 登録されているデータ数
}
```

HashCと違い配列tableに直接データを格納

32

オープンアドレス法の実装(2)

- コンストラクタ
  - バケット数＝配列の要素数を指定
  - 未指定時はデフォルトの値（53）を採用
  - ハッシュ表を空を表すEMPTYで初期化

詳細は教科書 pp.205～209 List 8.4

```
public class HashOA {
  static final MyKey DELETED = new MyKey(null);  // 削除済みを表す
  static final MyKey EMPTY   = new MyKey(null);  // 空を表す
  static final int DEFAULT_BUCKET_SIZE = 53;     // サイズは素数が望ましい
  public HashOA() {
    this(DEFAULT_BUCKET_SIZE);
  }
  public HashOA(int bucketSize) {
    table = new Bucket[bucketSize];
    for (int i = 0; i < bucketSize; i++) {
      table[i] = new Bucket(EMPTY, null);
    }
    this.bucketSize = bucketSize;
    nElements = 0;
  }
}
```

33

オープンアドレス法の実装(3)

- MyKeyクラスは共通
- ハッシュ値の計算：チェイン法と同じ
  - hashメソッドでハッシュ値を計算
  - MyKeyクラスのhashCodeメソッドで得られた値をbucketSizeで割って得られた剰余を返す
- 再ハッシュ
  - 現在のハッシュ値に1を加えてbucketSizeで剰余演算

詳細は教科書 pp.205～209 List 8.4

```
public class HashOA {
  private int hash(MyKey key) {
    return key.hashCode() % bucketSize;
  }
  private int rehash(int h) {
    return (h + 1) % bucketSize;
  }
}
```

34

オープンアドレス法の実装(4)

- 探索
  - キーのハッシュ値で示されるバケットから探索を開始
    - 一致するキーがみつかる
    - 「空」のバケットがみつかる

まで再ハッシュしながら調べる

詳細は教科書 pp.205～209 List 8.4

```
public class HashOA {
  public Object find(MyKey key) {
    int count = 0;
    int h = hash(key);
    MyKey k;
    while ((k = table[h].key) != EMPTY) {
      if (k != DELETED && key.equals(k)) {
        return table[h].data;
      }
      if (++count > bucketSize) {
        return null;  // 全部調べても見つからなかった
      }
      h = rehash(h);
    }
    return null;
  }
}
```

探索数をチェックし，全部調べたら終了

35

オープンアドレス法の実装(5)

コードは割愛，詳細は教科書 pp.205～209 List 8.4

- 挿入
  - キーのハッシュ値で示されるバケットから開始し，再ハッシュしながら順に調べる
    - 「空」か「削除済み」のバケットがみつかる → データを登録
    - 一致するキーがみつかる → エラー（falseを返す）
  - 再ハッシュ回数がバケット数を超えたら例外処理
- 削除
  - キーのハッシュ値で示されるバケットから開始し，再ハッシュしながら順に調べる
    - 一致するキーがみつかる → DELETEDにして「削除済み」に
    - 「空」のバケットがみつかる → 見つからなかった
  - 再ハッシュ回数がバケット数を超えたら見つからなかった

36

再ハッシュ手順

- 衝突が発生すると連続したバケットが埋まる
- 解決案１：バケットを c 個おきに調べる
  - $h_k(x) = (h(x) + ck) \bmod B$
  - c 個おきのバケットが連続的に埋まるで解決ではない
- 解決案２：再ハッシュのたびにランダムに選択
  - $1 \sim B-1$ までに数が１回だけランダムに出現する数列 $n_1, n_2, \ldots n_{B-1}$ を準備
  - $h_k(x) = (h(x) + n_k) \bmod B$

n_k：6, 1, 4, 7, 2, 9, 8, 3, 5

ハッシュ値3：9, 4, 7, 0, 5, 2, 1, 6, 8

ハッシュ値1：7, 2, 5, 8, 3, 0, 9, 4, 6

37

オープンアドレス法の解析

- α：ハッシュ表の利用率

$$\alpha = \frac{\text{登録したデータ数}}{\text{全バケット数}}$$

	(A) 順にバケットを調べる	(B)ランダムにバケットを調べる
探索成功	$\frac{1-\alpha/2}{1-\alpha}$	$\frac{1}{\alpha}\log_e\frac{1}{1-\alpha}$
探索失敗	$\frac{1}{2}+\frac{1}{2(1-\alpha)^2}$	$\frac{1}{1-\alpha}$
$\alpha = 80\%$（探索成功）	3.0個	2.0個
$\alpha = 90\%$（探索成功）	5.5個	2.6個

- (A)で80%，(B)90%が最大使用率の目安

38

教科書 第8章（pp.214～217）

ハッシュ法
ハッシュ関数，他

39

ハッシュ関数

- 均等なハッシュ値が発生することを前提
  - 実際のキーに対して事前に実験したほうがよい
- できるだけキーのすべてのビットがハッシュ値に影響を与えるようにする
  - Ex. 文字列の１文字目だけより文字コードの和を使う
- キーの値を２乗してその中央をとる（平方採中法：middle-square method）の結果は良好
- JavaではhashCodeメソッドでハッシュ値を計算
  - 詳細は，教科書第3章の「hashCodeメソッドを定義する」

40

ハッシュ法の応用

- 大きなデータを大量に比較する場合
  - 各データのハッシュ値を予め計算
  - ハッシュ値を比較し等しくなければ次の処理へ
- 通信データやファイルの改ざんを防止
  - 送信側と受信側でハッシュ値を求めて比較
  - 一方向ハッシュ関数を利用

詳細は教科書 8.6節 pp.215～216

41

ハッシュ法の性質

- ハッシュ法の高速性はハッシュ表の大きさによる
- データの個数
  - チェイン法ではバケットの個数の数倍程度
  - オープンアドレス法ではバケットの個数の80～90%程度
- メモリ使用量：1000個のデータを扱うなら
  - チェイン法：400～500個のバケット
    - 「基本料金＋従量制料金」
  - オープンアドレス法：1176個（85%と仮定）
    - 「完全固定料金制」

42

まとめ

- 探索とは
- 線形探索と二分探索
- ハッシュ法
  - ハッシュ法の原理
  - チェイン法
  - オープンアドレス法
- ハッシュ関数，他

参考文献

- 定本 Javaプログラマのための
  アルゴリズムとデータ構造（近藤嘉雪）
- 新・明解 Javaで学ぶ
  アルゴリズムとデータ構造（柴田望洋）
- 岩波講座ソフトウェア科学 3
  アルゴリズムとデータ構造（石畑清）
- Javaで学ぶアルゴリズムとデータ構造
  Robert Lafore（著）・岩谷 宏（翻訳）
- Java アルゴリズム+データ構造完全制覇
  オングス (著)・杉山 貴章・後藤 大地 (監修)