# 情報構造 第十二回

集合・辞書

# 今日の予定

- 集合（Set）
  - 集合の仕様
  - 実現：ビットベクトル
  - 実現：連結リスト

- 辞書（Dictionary）
  - 辞書の仕様
  - 実現：配列へのベタ詰め法
  - 実現：ハッシュ法
    - オープンハッシュ法
    - クローズドハッシュ法

ここがメイン

# 集合

# 集合（set）

- 要素数は**有限**
- 要素の**並び順には意味はない**　{1, 2, 3}と{3, 2, 1}は同じ
- **同じ要素はひとつ**しか表れない
  - {1, 2, 2, 3, 1} は集合ではなく，**多重集合**（BAG, multiset）

- 要素間に**線形順序**（全順序）　≪ を考えることもある
- 集合S上の線形順序 ≪ はつぎのとおりである
  - ∀a, b ∈S に対して，**a≪b** または **a=b**または **b≪a** の**どれか1つ**だけ成立
    - **排中律**
  - ∀a, b, c ∈S に対して，**a≪b** かつ **b≪c** ならば **a≪c** が成立
    - **推移律**

# 集合の仕様

- **要素**：
  - 集合のすべての要素は**同じ型**Elementをもつ
  - 要素型Elementが線形順序をもちうる
- **構造**：
  - 集合は同じ型の集まり
  - 要素の重複はない
  - 要素を並べる順序に意味はない
  - 要素数は有限で，要素がないものを空集合と呼ぶ
- **操作**：

  集合に対して
  - 和・積・差の集合演算（Union, Intersection, Difference）
  - 要素の挿入・削除・所属（Insert, Delete, Member）
  - 最小要素の取り出し（Min）

  など

# 集合の実現：ビットベクトル

- 表現
  - 整数Nの値が小さいとき，0, 1,…, Nの整数を要素とする集合
  - すなわち，普遍集合（全体集合）{0, 1, 2,…N} 上の集合

  ⇒ビットベクトルで実現
- ビットベクトル
  - 表現する語・バイトの**ビット上の位置**とその**値**0または1で集合の要素を表す
  - 例えば，5が**集合要素**のとき**ビット位置5**の値が1

```
typedef char SET;        /* 普遍集合{0, 1, 2, 3, 4, 5, 6, 7} */
```

集合変数
S = {1, 5, 7}

# 実現アルゴリズム: 和集合，積集合，差集合

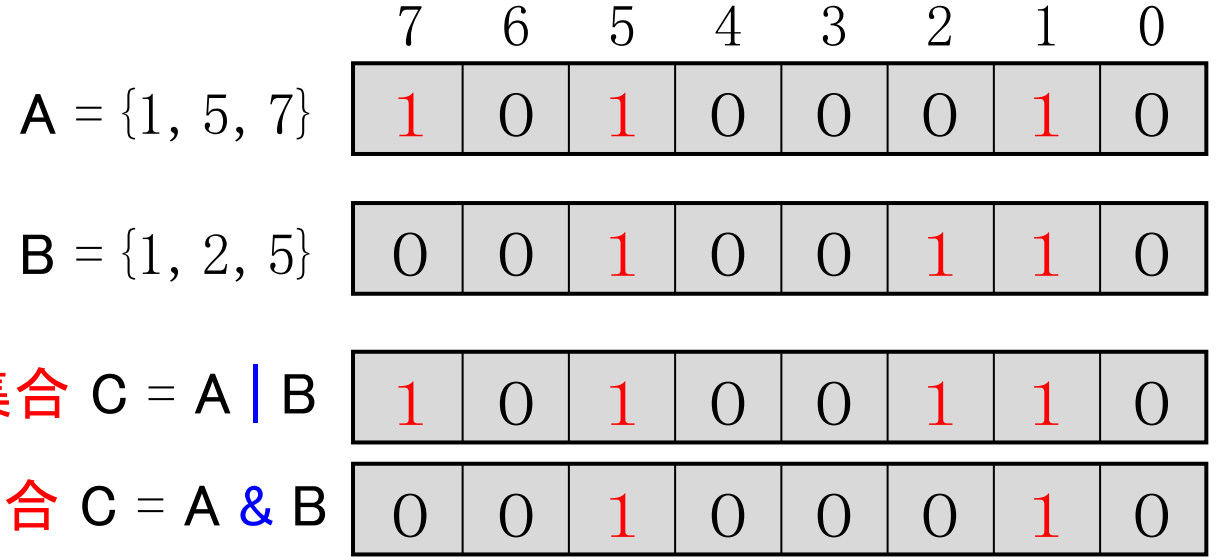
- 集合操作は以下で実現できる
  - ビット論理演算子：|, &, ~ (or, and, not)
  - シフト演算子：>>, << (右シフト，左シフト)

  集合 A の第 i 番目のビットを A[i] で表す（LSB を i=0 とする）

数値の各ビットを右または左へシフトさせる演算子

最下位ビット least significant bit（右端ビット）

- A, Bの**和集合** C = Union(A, B): C[i] = A[i] | B[i]
- A, Bの**積集合** C = Intersection(A, B): C[i] = A[i] & B[i]
- A, Bの**差集合** C = Difference(A, B): C[i] = A[i] & (~B[i])



# 実現アルゴリズム: 所属，挿入，削除

- **所属**：Member(i, A) = i∈A
  - {i} を表す整数$2^i$ と**A**の**ビット積**（$2^i$ **& A**） を結果値とする
    結果が整数**0**でないとき，すなわち $2^i$ & A != 0 のとき i∉A
- **挿入**：Insert(i, A)
  - {i} を表す整数$2^i$ と**A**の**ビット和**（$2^i$ | **A**）を結果値とする
- **削除** Delete(i, A)
  - {i} を表す整数$2^i$ の**1の補数**~$2^i$と**A**の**ビット積**（~$2^i$ **&** A）を結果値とする

Memberの例

| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| A ={1, 5, 7} | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| {5} = $2^i$ | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 5∈A? A&{5} | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |

# 実現アルゴリズム（ビットベクトル）の効率

- ビットベクトルが1語に納まるとき（**N**ビット≦1語）
  - **Member**, **Insert**, **Delete**, **Union**, **Intersection**, **Difference**すべて**一定時間**
  - …演算レジスタで**数回**の**論理演算**で実行できる
- **Nビット**に**m語**要するときは，**m倍**の**論理演算**が必要となる
- **Min**(A)は，Aの**ビット数N**の時間がかかる

# 普遍集合が列挙型

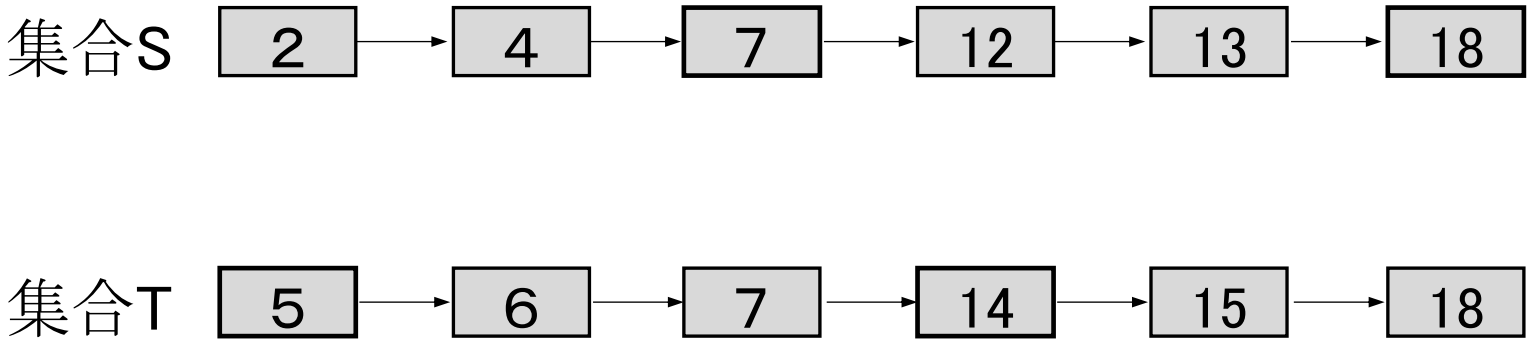
- 普遍集合を**列挙型**（スカラー型）にすることで，集合要素を**列挙値**の**識別子**で表せ，集合表現が分かりやすくなる
- 例えば，普遍集合{0, 1, 2, 3, 4, 5, 6, 7}の代わりに
  enum color {Black, White, Red, Green, Yellow, Blue, Brown, Purple}
  とした場合，列挙値Black,…Purpleは，0,…,7なので
  要素挿入 Member(2, A)は，Member(Red, A)と書ける

集合値S={Black,Red,Purple}

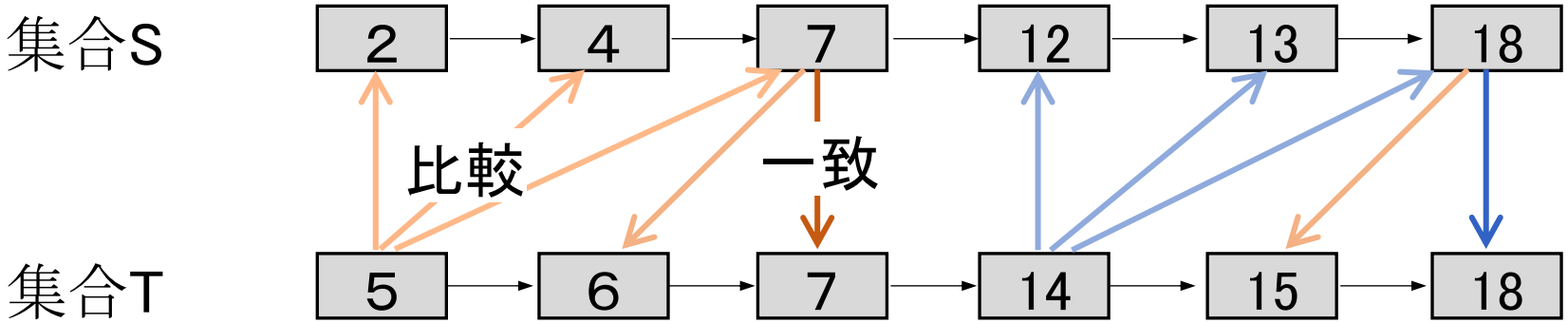
| Purple | Brown | Blue | Yellow | Green | Red | White | Black |
|---|---|---|---|---|---|---|---|
| =7 | =6 | =5 | =4 | =3 | =2 | =1 | =0 |
| 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |

# 集合の実現：連結リスト

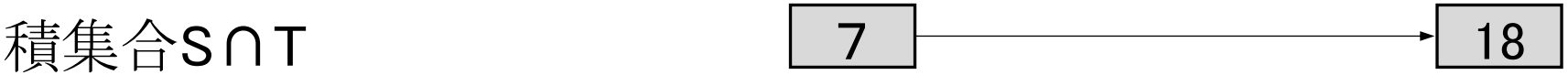
- 表現
  - ひとつの**連結リスト**でひとつの**集合**を表す
  - 連結リスト中の各構造体で，集合の各要素を表す
  - 操作の効率を考えて，集合の要素間の**線形順序**を用い，連結リスト中の**構造体**を**ソート**しておく



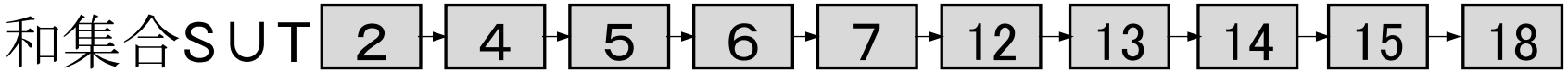
# 連結リストによる集合演算




一致した要素を出力




以下の要素を出力



# 実現アルゴリズム（連結リスト）の効率

- Union, Intersection Differenceの効率
  - 連結リストの構造体が**ソートされている**場合
    …二つのリストの，**長さの和**

  - 連結リストの構造体が**ソートされていない**場合
    …二つのリストの，**長さの積**

# 辞書

# 辞書（Dictionary）

- **操作 Insert, Delete, Member**を備えた**集合**を**辞書**とよぶ

=> 操作Union, Intersection, Differenceを除いた抽象データ型

# 辞書の仕様

- 要素・構造（辞書の要素と構造は**集合と同じ**）
  - 集合のすべての要素は**同じ型**Elementをもつ
  - Elementは**線形順序**を持ち得る
  - 要素の重複はない
  - 要素を並べる順序に意味はない
  - 要素数は有限で，要素がないものを空の辞書と呼ぶ

例：$\{1,2,3\}=\{3,2,1\}\neq\{3,3,2,1\}$

- 操作（辞書の操作は抽象データ型**集合の部分集合**）
  - 要素の挿入，削除，所属（探索）（Insert, Delete, Member）
  - など

# 辞書の操作

辞書型 Dictionary　　要素型 Element

辞書型変数 D　　要素型データ x

- int Member(Element x, Dictionary D)
  - Post: 関数値は **x∈D** ならば真（1）さもなければ偽（0）
- Dictionary Insert (Element x Dictionary D)
  - Post: 関数値は **D ∪ {x}**
- Dictionary Delete(Element x Dictionary D)
  - Post: 関数値は **D - {x}**
- Dictionary Create (void)
  - Post: 関数値は，**空の辞書**
- Element Min(Dictionary D)
  - Pre: **D≠空の辞書**
  - Post: 関数値は**D**中の**最小要素の値**

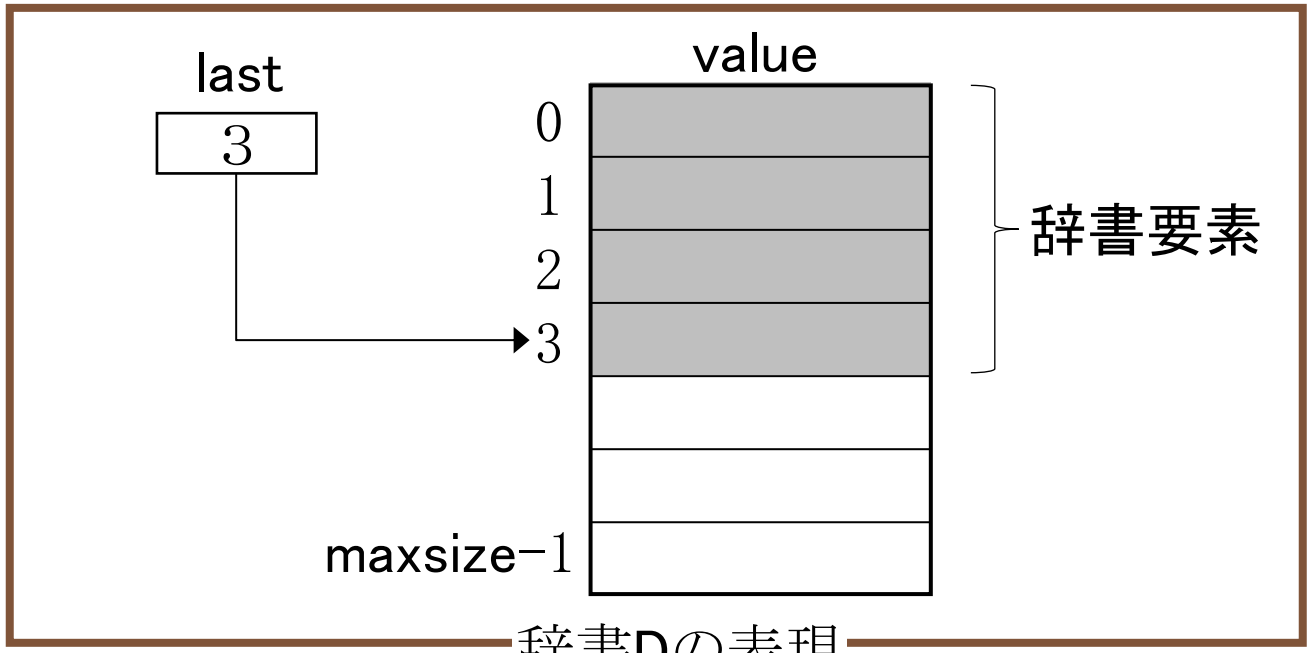
# 辞書の実現

- 辞書の**配列**上での実現方法
  - ベタ詰め法
  - ハッシュ法

# 表現：辞書の実現 配列へのベタ詰め

```
#define maxsize 1000          /*辞書の要素の最大数*/
typedef struct{
        Element value[maxsize];
        int last;             /*辞書要素の最後，-1の時は空*/
 }Dictionary;
```

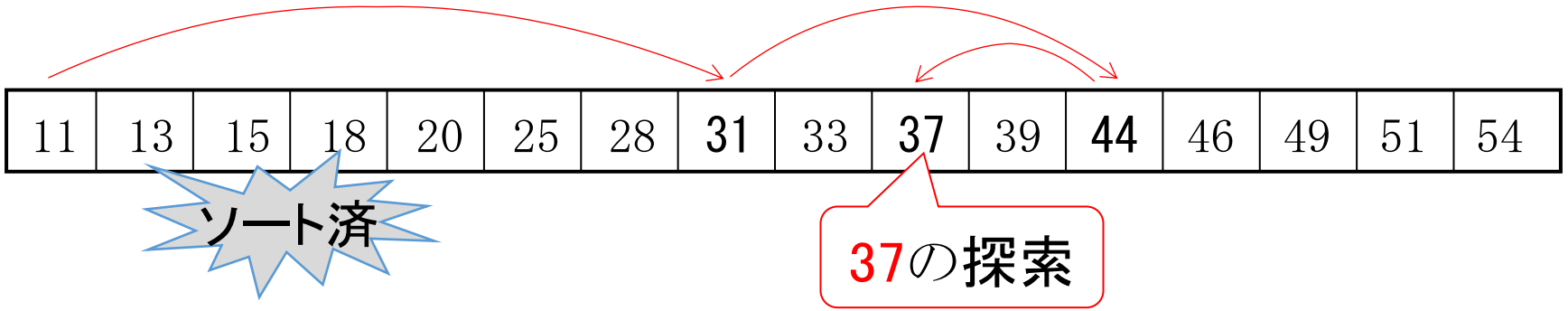
0からlastが辞書要素




辞書Dの表現

# 効率：辞書の実現 配列へのベタ詰め

- 辞書の要素を**ソートしない**とき
  - Insert, Delete, Memberは，最悪N（辞書の要素数）かかる
- 辞書の要素を**ソートした**とき
  - Member, Insert, Deleteは，探索に最悪$\log_2 N$かかる



# 辞書の実現：ハッシュ法

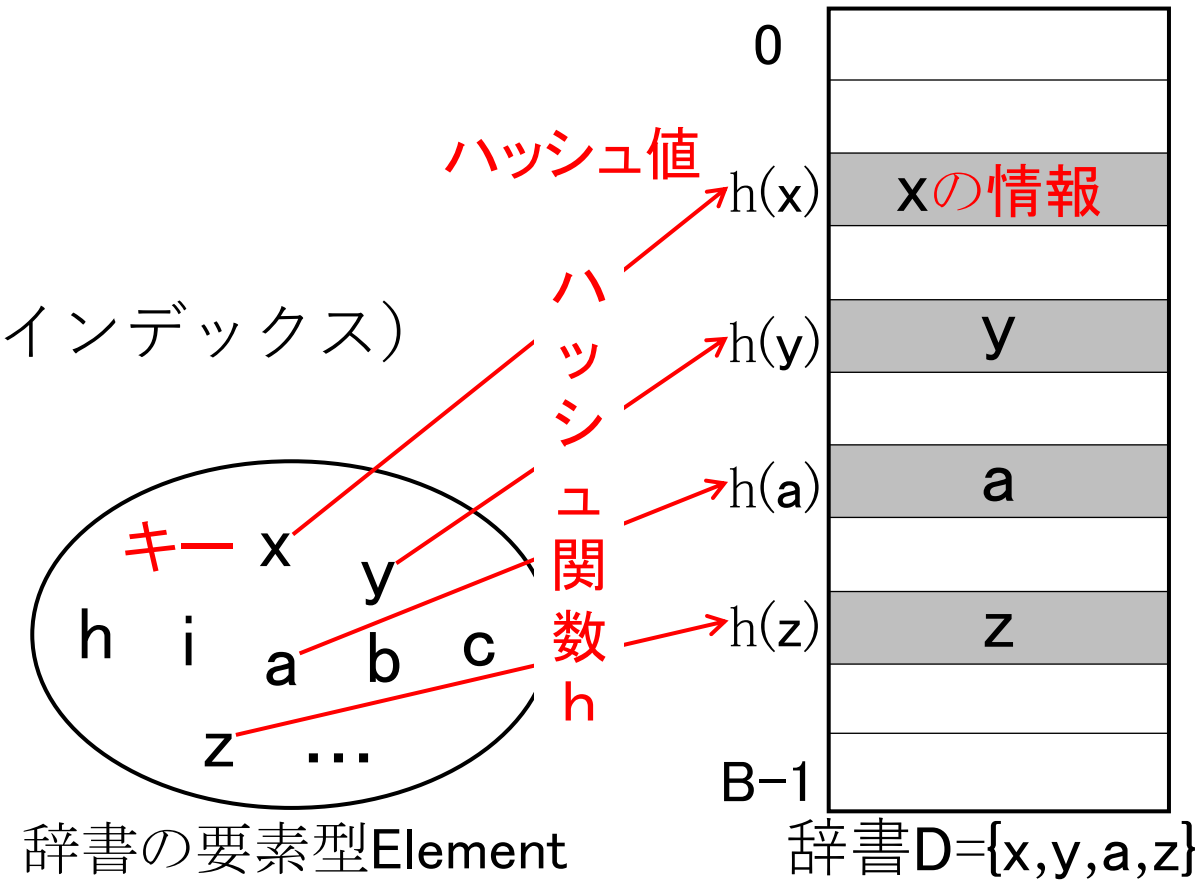
- 辞書の**配列へのベタ詰め**の実現は
  - 操作は最悪で**登録要素数N**（ソートすれば**$\log_2 N$**）のオーダーの時間計算量

- ハッシュ法
  - 辞書を配列（ハッシュ表）で実現
  - 各操作の**平均の時間計算量を一定**にする

# 辞書要素とハッシュ表の対応

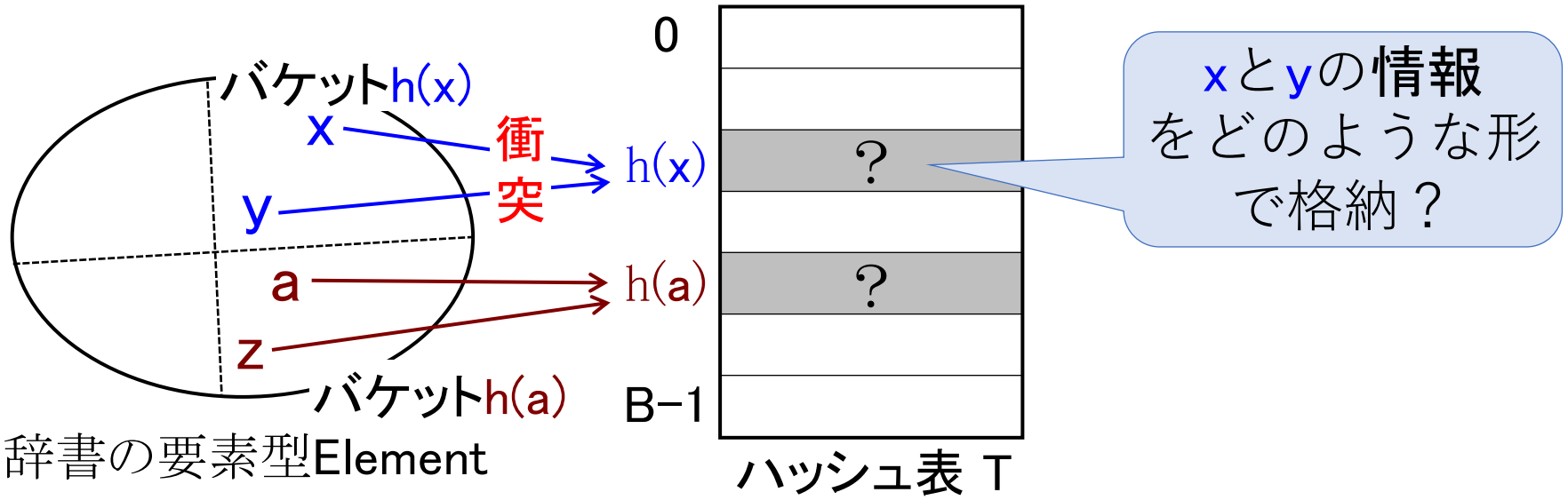
- 辞書の要素型Element
- 辞書要素を格納する配列T（ハッシュ表）…辞書の表現
- Tのインデックスが0からB-1

- 次の関数hを考える
  - h: Element→[0…B-1]（要素からインデックス）
  - hを**ハッシュ関数**と呼ぶ
  - Element型の要素xをキー
  - h(x)をキーxの**ハッシュ値**と呼ぶ



# 衝突（collision）

- Elementすべての要素のハッシュ値が異なる（**h**が単斜）なら**辞書要素（キー）xの情報は配列Tの第h(x)番目**に格納できる？
- ハッシュ表を非常に大きくしなければ，一意の関数を見つけるのは**難しい**！




- いくつかの要素が**同じハッシュ値**を持つ
  - この**クラス**を**バケット**，そのハッシュ値を**バケット番号**という
- 同じバケット内の異なる辞書要素（キー）xとyは，**衝突**するという

# ハッシュ法の種類

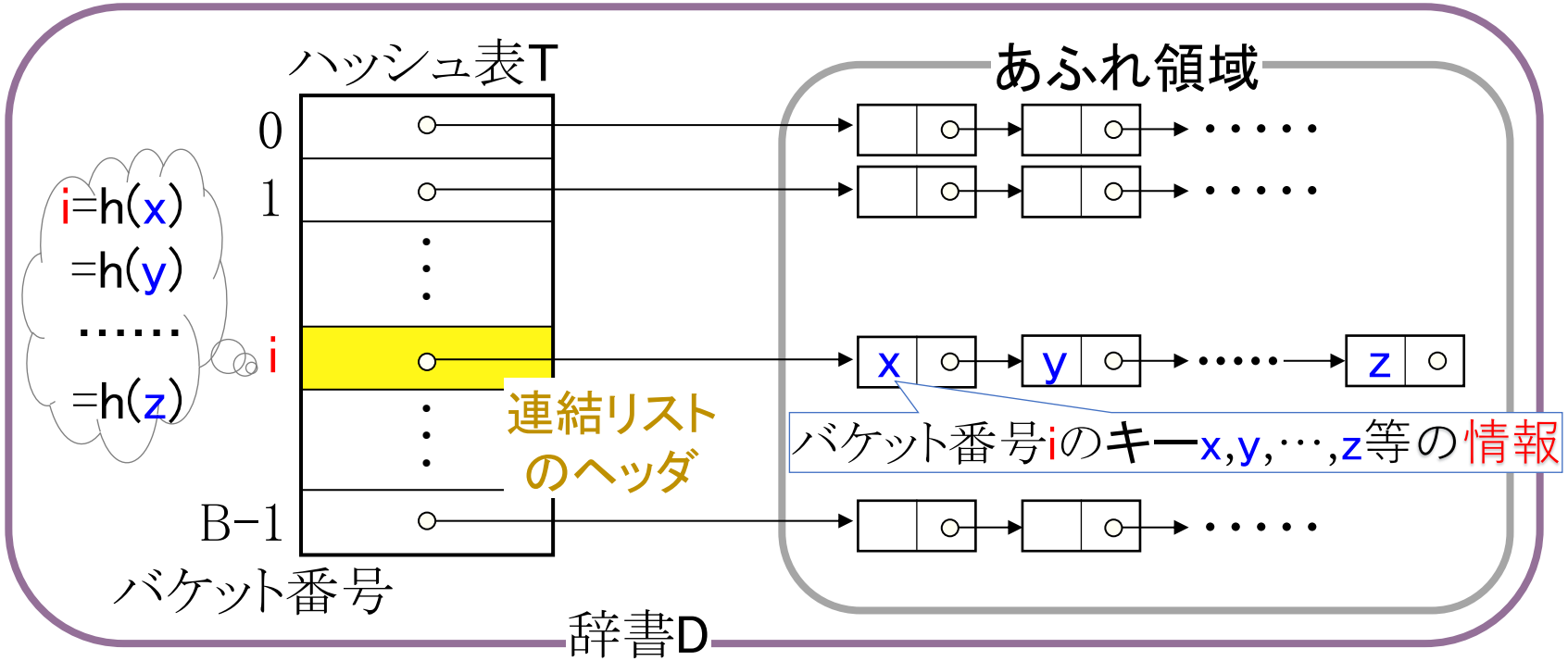
- 要素の衝突に多する処理はいろいろあるが，ここでは二つの方法を採り上げる
  1. **オープンハッシュ法**（チェイン法）
  2. **クローズドハッシュ法**（オープンアドレス法）

| | | |
|---|---|---|
| オープンハッシュ法 | ⇔ | クローズドハッシュ法 |
| (open hashing) | | (closed hashing) |
| overflow hash | ⇔ | open hash |
| direct chaining | ⇔ | open addressing |
| chaining | ⇔ | open addressing |
| 外部ハッシュ | ⇔ | 内部ハッシュ |

オープンが逆になる他の名前があるので注意

# オープンハッシュ法：辞書の表現

- 配列T：　ハッシュ表
  インデックスがバケット番号 0, 1,…, B-1
- 同じバケット i 中の衝突要素（衝突キー）：
  T[i]をヘッダとする**連結リスト**に登録（Tの要素はキーではない）

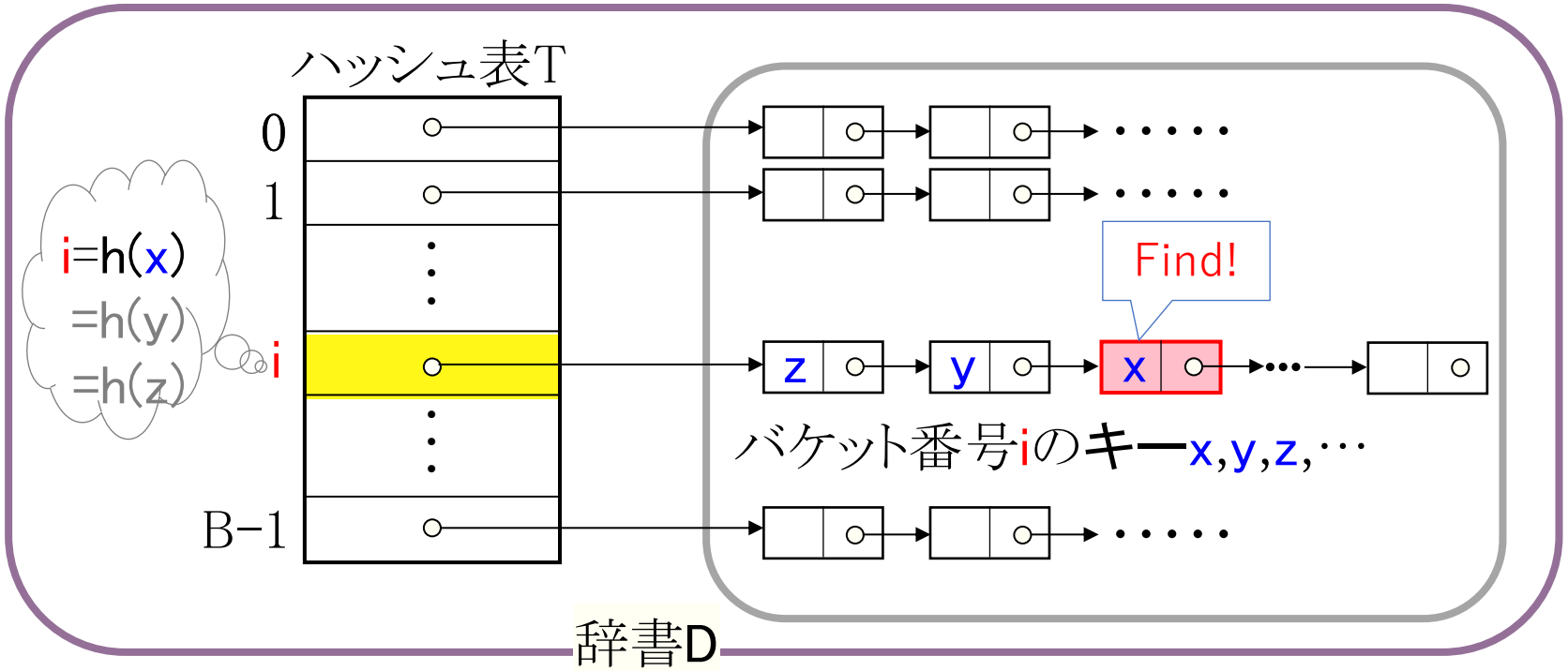


# オープンハッシュ法：操作

- 挿入：Insert(x, D)
- 削除：Delete(x, D)
- 所属・探索：Member(x, D)

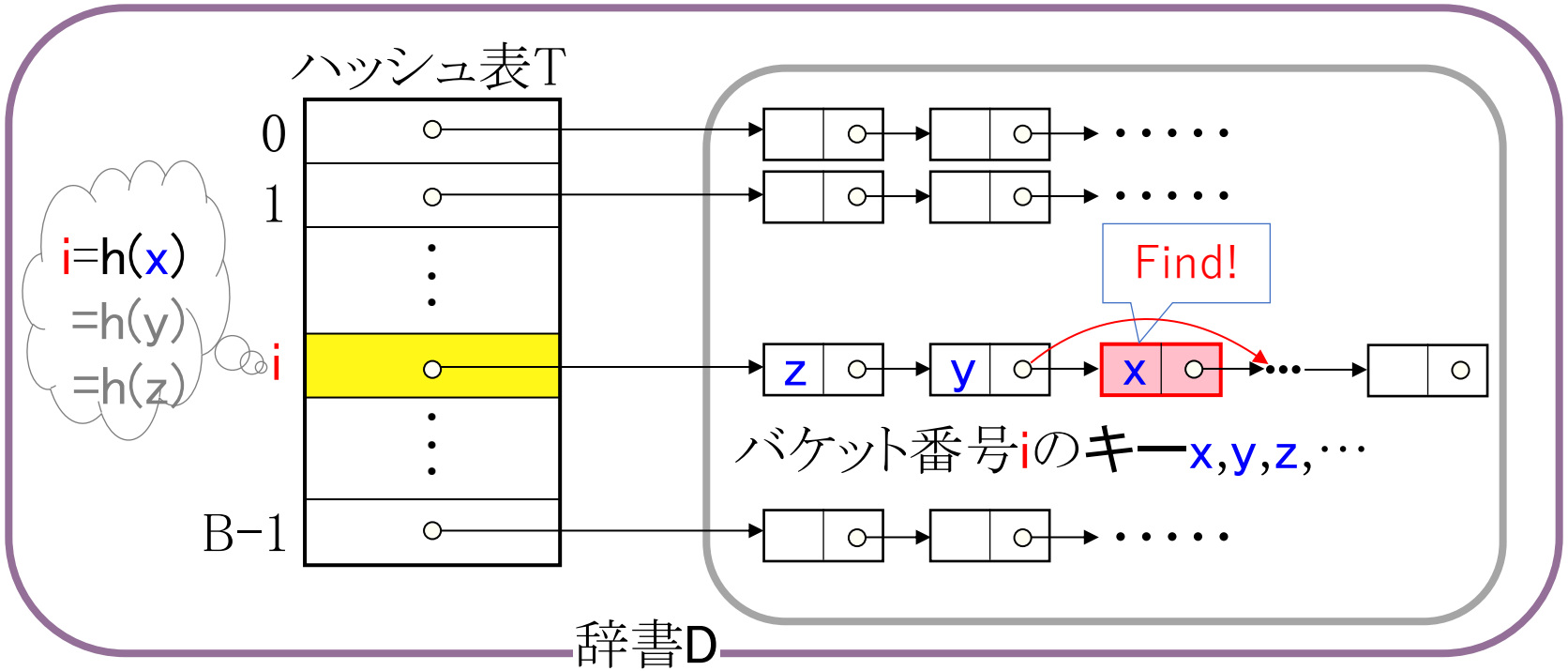
# 実現アルゴリズム: 挿入Insert (x, D)

1. **キー**（辞書要素）xの**バケット** i (=h(x))を求める
2. **ハッシュ表の要素T[i]をヘッダ**とする**連結リスト**中に**xがあるか**調べる
3. **あれば**，何もせず終了
4. **なければ**，xを**連結リストの要素**として挿入して終了



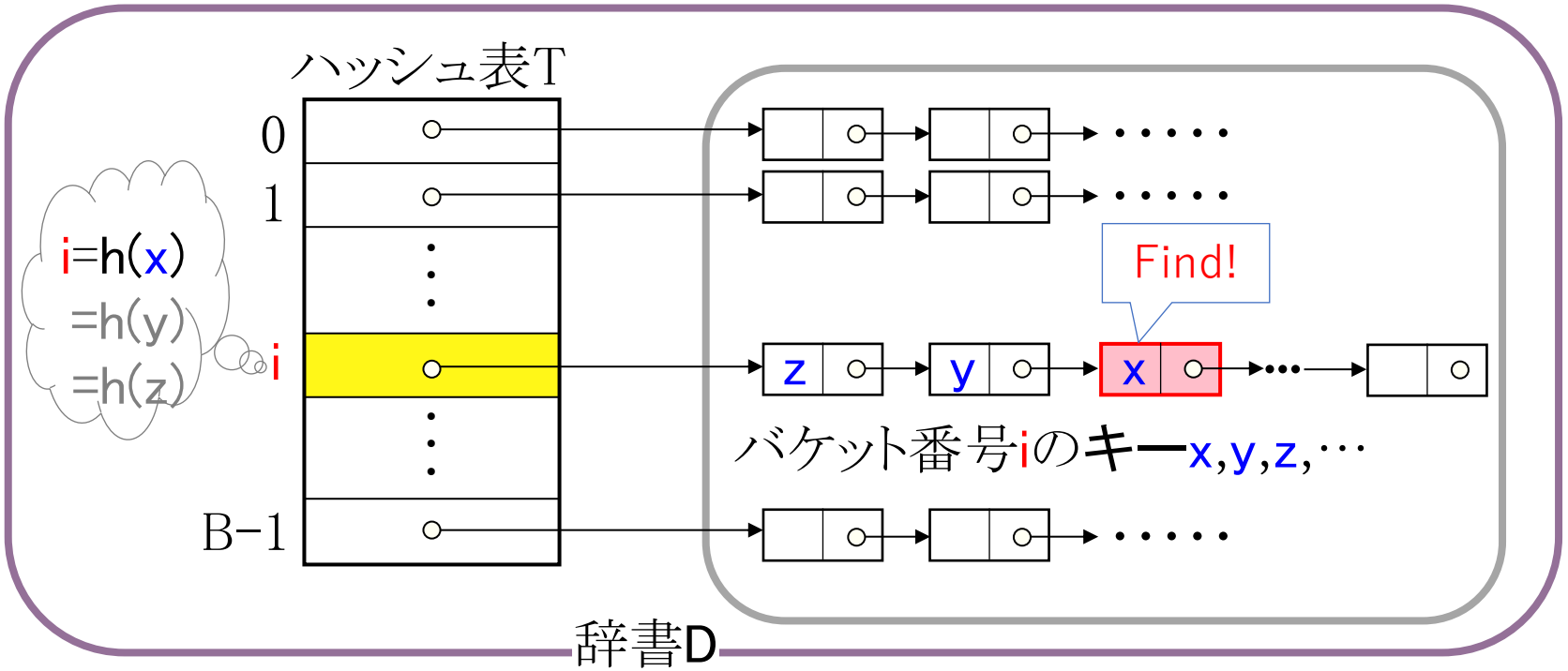
# 実現アルゴリズム: 削除Delete (x, D)

1. **キーx**のバケット i (=h(x))を求める
2. **ハッシュ表の要素T[i]をヘッダ**とする**連結リスト**中に**xがあるか**調べる
3. **あれば**，**連結リスト**から**削除**して終了
4. **なければ**，何もせず終了



# 実現アルゴリズム: 探索Member (x, D)

1. **キーx**のバケット i (=h(x))を求める
2. **ハッシュ表の要素T[i]をヘッダ**とする**連結リスト**中に**xがあるか**調べ，その**結果を関数値**として終了



# ハッシュ関数の選定

- **キー**（辞書要素）を各バケットに**均一**に**割り当てる**ハッシュ関数が好ましい！
- **キーx**が**文字列**の時は，**数値**に変換してから，バケットに割り当てる

# ハッシュ関数の例

- **キーx**: **文字列$c_{n-1}\cdots c_1\ c_0$**
- **バケット数B**: 50
- **ハッシュ値**：「各文字の**ASCIIコードの和**の**B**による**剰余**」

  $h(c_{n-1}\cdots c_1 c_0) = (c_{n-1} + \cdots + c_1 + c_0)\ \%\ B$

  $h(\text{“A00”}) = (65 + 48 + 48)\ \%\ 50 = 11$

**C言語**では、文字の和は、文字の**ASCIIコード**の和

- キー100個：A00, A01, …, A99を
  50個のバケットに割り当てる

=> キーが**均一**に**割り当てられない**

- キー集合{“A00”,…, “A99”}, h(x) = 「文字コードの和のBの剰余」

| | | |
|---|---|---|
| バケット11 | 1個 | A00 |
| **バケット12** | **2個** | A01 A10 |
| バケット13 | 3個 | A02 A11 A20 |
| バケット14 | 4個 | A03 A12 A21 A30 |
| バケット15 | 5個 | A04 A13 A22 A31 A40 |
| バケット16 | 6個 | A05 A14 A23 A32 A41 A50 |
| バケット17 | 7個 | A06 A15 A24 A33 A42 A51 A60 |
| バケット18 | 8個 | A07 A16 A25 A34 A43 A52 A61 A70 |
| バケット19 | 9個 | A08 A17 A26 A35 A44 A53 A62 A71 A80 |
| **バケット20** | **10個** | A09 A18 A27 A36 A45 A54 A63 A72 A81 A90 |
| バケット21 | 9個 | A19 A28 A37 A46 A55 A64 A73 A82 A91 |
| バケット22 | 8個 | A29 A38 A47 A56 A65 A74 A83 A92 |
| バケット23 | 7個 | A39 A48 A57 A66 A75 A84 A93 |
| バケット24 | 6個 | A49 A58 A67 A76 A85 A94 |
| バケット25 | 5個 | A59 A68 A77 A86 A95 |
| バケット26 | 4個 | A69 A78 A87 A96 |
| バケット27 | 3個 | A79 A88 A97 |
| **バケット28** | **2個** | A89 A98 |
| バケット29 | 1個 | A99 |

バケット0から10のキーは0個

バケット30から49のキーは0個

| | |
|---|---|
| 平均 | 2 |
| 分散 | 9.4 |

# ハッシュ関数の例

- **キーx**: **文字列$c_{n-1}\cdots c_1\ c_0$**
- **バケット数B**: 50
- **ハッシュ値**：「n個の文字列をASCIIコードで128進数n桁の数値」の**B**による**剰余**

  $h(c_{n-1}\cdots c_1 c_0) = (c_{n-1} * 128^{n-1} + \cdots + c_1 * 128^1 + c_0 * 128^0)\ \%\ B$

  例：$h(A49) = (\text{‘A’} * 128^2 + \text{‘4’} * 128^1 + \text{‘9’} * 128^0)\ \%\ B$

  $= (65 * 16381 + 52 * 128 + 57 * 1)\ \%B$

  $= 1071673\ \%\ 50$

  $= 23$

- キー100個：A00, A01, …, A99を　50個のバケットに割り当てる

=> キーがほぼ**均一**に**割り当てられる**

128進数3桁ならば，$128^3$(=2,097,152)の大きさのハッシュ表を用意すれば**ハッシュ値は単斜**（衝突は起こらない）

- キー集合{“A00”,…, “A99”}, h(x) = 「128進数のBの剰余」
  - すべてのバケットで，キーは1個から3個

| | | | | | |
|---|---|---|---|---|---|
| **バケット0** | **2個** | A58 A72 | バケット25 | 1個 | A65 |
| **バケット1** | **2個** | A59 A73 | **バケット26** | **2個** | A66 A8 |
| **バケット2** | **2個** | A00 A74 | **バケット27** | **2個** | A67 A81 |
| **バケット3** | **2個** | A01 A75 | **バケット28** | **2個** | A68 A82 |
| バケット4 | 3個 | A02 A76 A90 | **バケット29** | **2個** | A69 A83 |
| バケット5 | 3個 | A03 A77 A91 | **バケット30** | **2個** | A10 A84 |
| バケット6 | 3個 | A04 A78 A92 | **バケット31** | **2個** | A11 A85 |
| バケット7 | 3個 | A05 A79 A93 | **バケット32** | **2個** | A12 A86 |
| バケット8 | 3個 | A06 A20 A94 | **バケット33** | **2個** | A13 A87 |
| バケット9 | 3個 | A07 A21 A95 | **バケット34** | **2個** | A14 A88 |
| バケット10 | 3個 | A08 A22 A96 | **バケット35** | **2個** | A15 A89 |
| バケット11 | 3個 | A09 A23 A97 | **バケット36** | **2個** | A16 A30 |
| **バケット12** | **2個** | A24 A98 | **バケット37** | **2個** | A17 A31 |
| **バケット13** | **2個** | A25 A99 | **バケット38** | **2個** | A18 A32 |
| **バケット14** | **2個** | A26 A40 | **バケット39** | **2個** | A19 A33 |
| **バケット15** | **2個** | A27 A41 | バケット40 | 1個 | A34 |
| **バケット16** | **2個** | A28 A42 | バケット41 | 1個 | A35 |
| **バケット17** | **2個** | A29 A43 | **バケット42** | **2個** | A36 A50 |
| バケット18 | 1個 | A44 | **バケット43** | **2個** | A37 A51 |
| バケット19 | 1個 | A45 | **バケット44** | **2個** | A38 A52 |
| **バケット20** | **2個** | A46 A60 | **バケット45** | **2個** | A39 A53 |
| **バケット21** | **2個** | A47 A61 | バケット46 | 1個 | A54 |
| **バケット22** | **2個** | A48 A62 | バケット47 | 1個 | A55 |
| **バケット23** | **2個** | A49 A63 | **バケット48** | **2個** | A56 A70 |
| バケット24 | 1個 | A64 | **バケット49** | **2個** | A57 A71 |

| | |
|---|---|
| 平均 | 2 |
| 分散 | 0.32 |

# ハッシュ関数の作り方

- **除算法**　　h(x) = キーxを数値化 % B
- **平方採中法（乗算法，中央2乗法）**

  $h(x) = (n^2 / C) \% B$

  擬似乱数の生成方法

  - **文字キーx**を変換した**数値n**が**0から**Kとする
  - $BC^2 \fallingdotseq K^2$となるように**整数C**で，数値**nの2乗の中央値**をとることができる
  - 原理は，次のようにキーの数値54321の2乗の**中央値077**をとることである．
    これは，**中央の桁**が**全桁に依存**することによる

    $54321^2 = 2950771041$

- 折り込み法

  12345678 = 0001|2345|678

  ⇒1000 + 2345 + 867 = 4221

- 平方採中法 キー集合{0,…, 499}, h(x) =(n² / C) % B,  K = 499, C = 71

| | | | | | |
|---|---|---|---|---|---|
| バケット0: | 14個 | バケット17: | 9個 | **バケット34:** | **10個** |
| バケット1: | 16個 | バケット18: | 8個 | バケット35: | 8個 |
| **バケット2:** | **17個** | バケット19: | 12個 | バケット36: | 9個 |
| バケット3: | 8個 | バケット20: | 7個 | バケット37: | 12個 |
| バケット4: | 12個 | バケット21: | 12個 | バケット38: | 11個 |
| バケット5: | 9個 | バケット22: | 5個 | バケット39: | 8個 |
| バケット6: | 13個 | バケット23: | 8個 | バケット40: | 7個 |
| バケット7: | 9個 | **バケット24:** | **10個** | バケット41: | 9個 |
| バケット8: | 11個 | **バケット25:** | **10個** | バケット42: | 14個 |
| バケット9: | 13個 | バケット26: | 16個 | バケット43: | 5個 |
| バケット10: | 8個 | **バケット27:** | **10個** | **バケット44:** | **10個** |
| バケット11: | 11個 | バケット28: | 8個 | バケット45: | 12個 |
| バケット12: | 5個 | バケット29: | 8個 | バケット46: | 9個 |
| バケット13: | 8個 | バケット30: | 11個 | バケット47: | 11個 |
| バケット14: | 13個 | **バケット31:** | **10個** | **バケット48:** | **3個** |
| バケット15: | 14個 | バケット32: | 6個 | **バケット49:** | **10個** |
| バケット16: | 12個 | バケット33: | 9個 | | |

| | |
|---|---|
| 平均 | 10 |
| 分散 | 8.7 |

- 平方採中法 キー集合{“A00”,…, “A99”}, h(x) = ($n^2$ / C) % B
  - n=x[0]*$128^2$+x[1]*128+x[2], K=1072313("A99"-1), C=153187

| | | |
|---|---|---|
| バケット0: 1個 | **バケット18: 2個** | バケット36: 1個 |
| バケット1: 4個 | バケット19: 3個 | バケット37: 3個 |
| **バケット2: 2個** | バケット20: 1個 | **バケット38: 2個** |
| **バケット3: 0個** | バケット21: 4個 | バケット39: 1個 |
| **バケット4: 2個** | バケット22: 1個 | バケット40: 1個 |
| バケット5: 3個 | バケット23: 3個 | バケット41: 3個 |
| バケット6: 1個 | **バケット24: 2個** | バケット42: 1個 |
| バケット7: 4個 | バケット25: 1個 | **バケット43: 5個** |
| バケット8: 1個 | バケット26: 1個 | バケット44: 1個 |
| **バケット9: 2個** | バケット27: 4個 | バケット45: 1個 |
| **バケット10: 2個** | バケット28: 1個 | **バケット46: 2個** |
| **バケット11: 2個** | バケット29: 4個 | **バケット47: 2個** |
| **バケット12: 0個** | バケット30: 1個 | **バケット48: 0個** |
| バケット13: 4個 | バケット31: 1個 | バケット49: 4個 |
| バケット14: 1個 | **バケット32: 2個** | |
| バケット15: 4個 | **バケット33: 2個** | |
| バケット16: 1個 | **バケット34: 0個** | |
| バケット17: 1個 | **バケット35: 5個** | |

| | |
|---|---|
| 平均 | 2 |
| 分散 | 1.8 |

- 平方採中法 キー集合{“A00”,…, “A99”}, h(x) = ($n^2$ / C) % B
  - n=x[0]*$128^2$+x[1]*128+x[2], K=1161("A99"-"A00"), C=165

| | | |
|---|---|---|
| バケット0: 3個 | バケット18: 1個 | **バケット36: 2個** |
| バケット1: 1個 | バケット19: 1個 | **バケット37: 2個** |
| バケット2: 1個 | バケット20: 4個 | バケット38: 3個 |
| **バケット3: 2個** | バケット21: 3個 | バケット39: 3個 |
| **バケット4: 0個** | **バケット22: 0個** | **バケット40: 0個** |
| バケット5: 5個 | **バケット23: 2個** | バケット41: 1個 |
| **バケット6: 0個** | バケット24: 3個 | **バケット42: 0個** |
| **バケット7: 0個** | バケット25: 3個 | **バケット43: 0個** |
| バケット8: 2個 | **バケット26: 2個** | バケット44: 1個 |
| バケット9: 1個 | バケット27: 3個 | バケット45: 3個 |
| **バケット10: 2個** | バケット28: 3個 | バケット46: 3個 |
| バケット11: 4個 | バケット29: 1個 | バケット47: 3個 |
| バケット12: 4個 | **バケット30: 2個** | バケット48: 1個 |
| **バケット13: 2個** | **バケット31: 2個** | **バケット49: 0個** |
| **バケット14: 2個** | **バケット32: 7個** | |
| **バケット15: 2個** | **バケット33: 0個** | |
| バケット16: 4個 | バケット34: 1個 | |
| **バケット17: 2個** | バケット35: 3個 | |

| | |
|---|---|
| 平均 | 2 |
| 分散 | 2.16 |

- 平方採中法 キー集合{“A00”,…, “A99”}, h(x) = (n² / C) % B
  - n=x[0]*128²+x[1]*128+x[2], K=2097157(128進数3桁), C=299593

| | | |
|---|---|---|
| バケット0: 3個 | **バケット18: 2個** | バケット36: 4個 |
| バケット1: 3個 | バケット19: 1個 | **バケット37: 2個** |
| **バケット2: 2個** | **バケット20: 2個** | **バケット38: 0個** |
| **バケット3: 2個** | **バケット21: 2個** | バケット39: 3個 |
| **バケット4: 2個** | バケット22: 1個 | バケット40: 1個 |
| バケット5: 1個 | バケット23: 3個 | バケット41: 3個 |
| バケット6: 1個 | バケット24: 1個 | **バケット42: 0個** |
| バケット7: 3個 | **バケット25: 2個** | **バケット43: 5個** |
| **バケット8: 2個** | バケット26: 3個 | **バケット44: 2個** |
| バケット9: 3個 | **バケット27: 2個** | バケット45: 1個 |
| バケット10: 3個 | バケット28: 3個 | バケット46: 3個 |
| **バケット11: 2個** | バケット29: 1個 | **バケット47: 2個** |
| バケット12: 1個 | バケット30: 3個 | **バケット48: 2個** |
| **バケット13: 2個** | **バケット31: 2個** | **バケット49: 0個** |
| **バケット14: 2個** | バケット32: 1個 | |
| **バケット15: 2個** | バケット33: 1個 | |
| バケット16: 3個 | バケット34: 3個 | |
| **バケット17: 2個** | **バケット35: 0個** | |

| | |
|---|---|
| 平均 | 2 |
| 分散 | 1.08 |

# オープンハッシュの平均時間計算量

- バケット数B，登録要素数N
  ⇒1バケットあたりの**平均要素数**は N/B
- **Member, Insert, Delete**の操作1回に要する平均の時間計算量は
  O(1 + N/B)
  ⇩
  B≒N ⇒ **平均時間計算量一定**

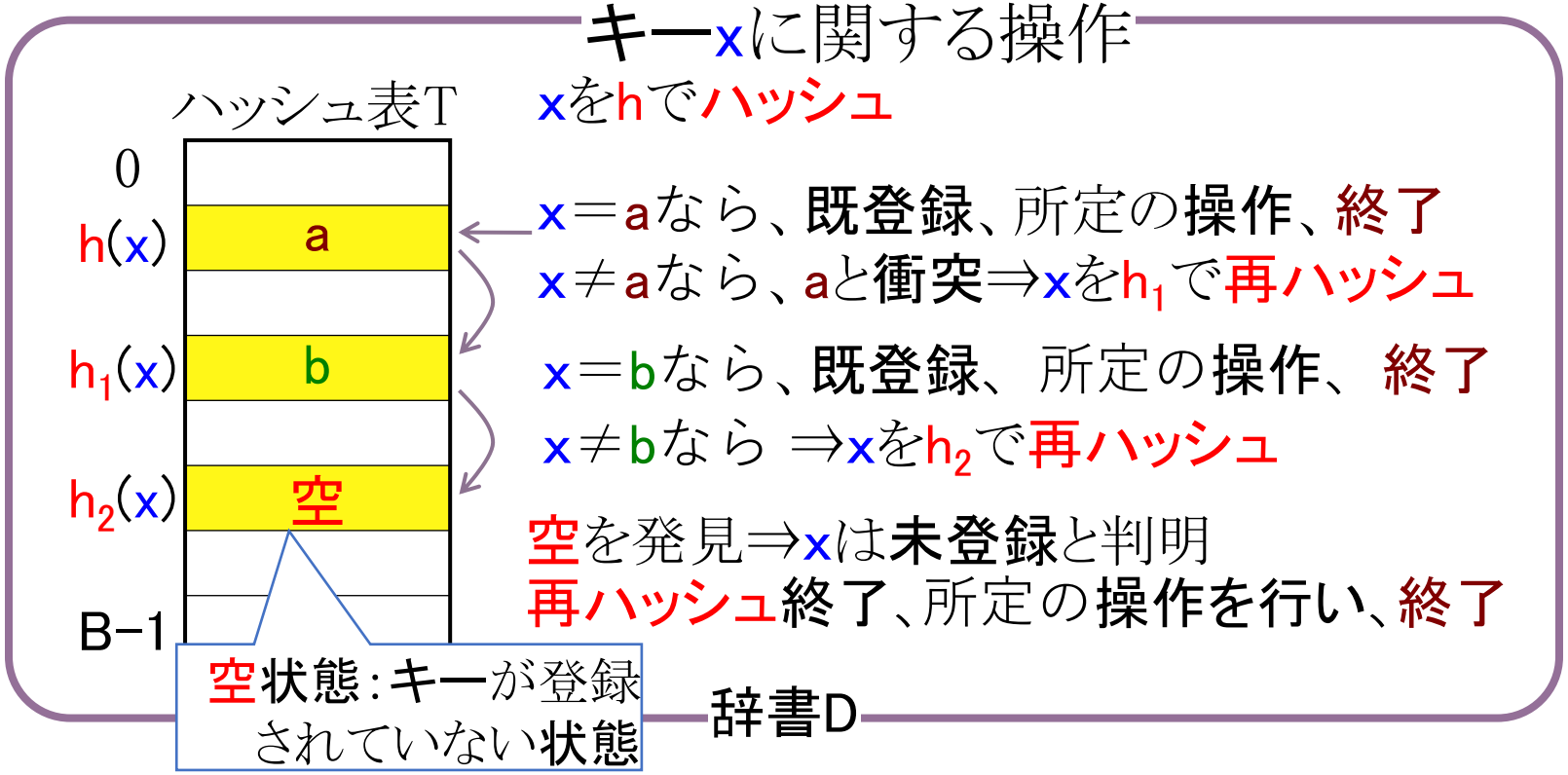
一つの要素にバケット一つ

- **表の再構成**
  - 登録要素数がバケット数Bの2倍以上になったとき，バケット数が2倍の新しいハッシュ表を作る

  ⇒いつも **(N/B)<2**となるので，操作の平均効率は**3未満**
  ⇒**表の大きさB**は**登録要素数N**の2倍程度
- **最小の要素**を取り出す操作**Min**の効率のよいアルゴリズムはない

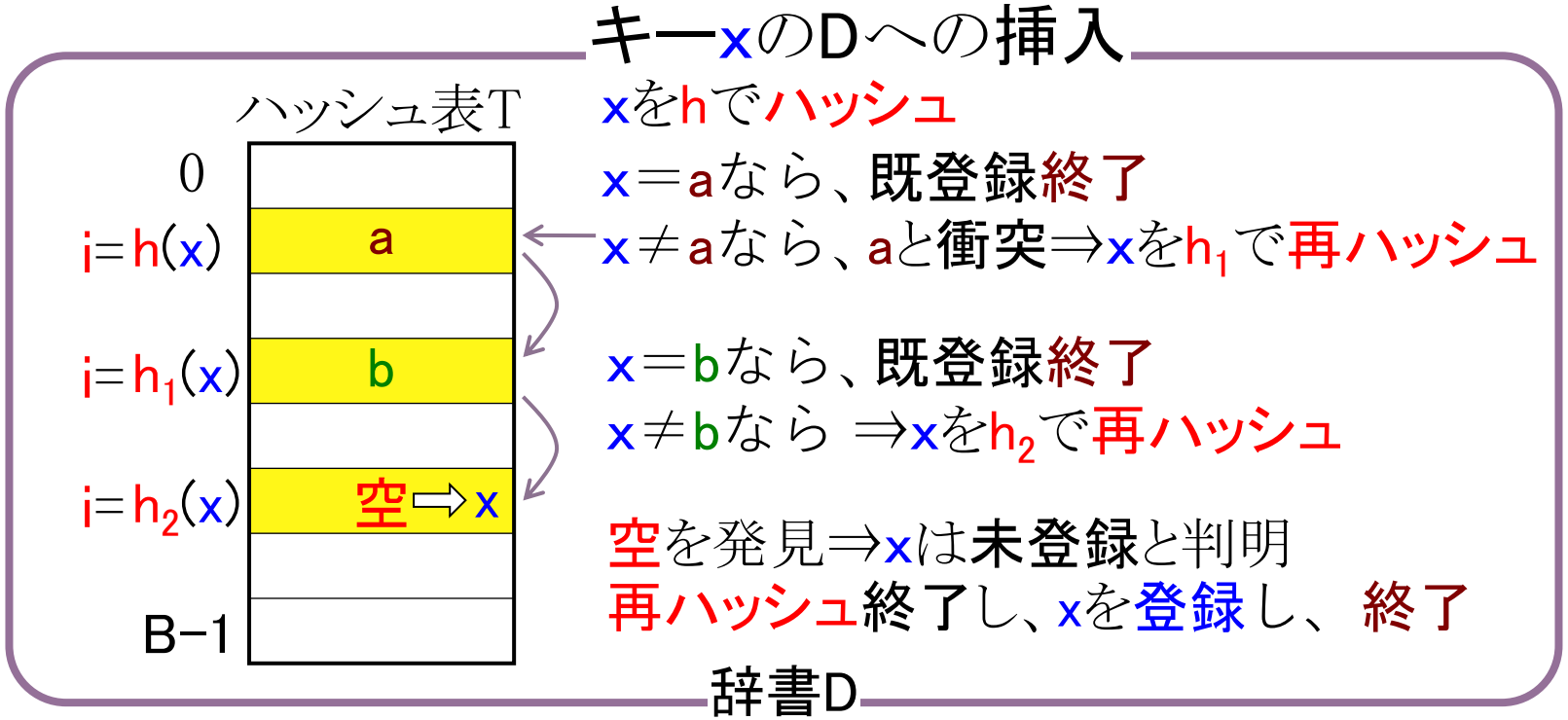
# クローズドハッシュ法：辞書の表現

- **ハッシュ表T**に，直接**キー（辞書要素）そのもの**を格納
- **ハッシュ表T**に，**ハッシュ関数h以外**に，**再ハッシュ関数**h1, h2, h3,…（高々B-1個）を使う



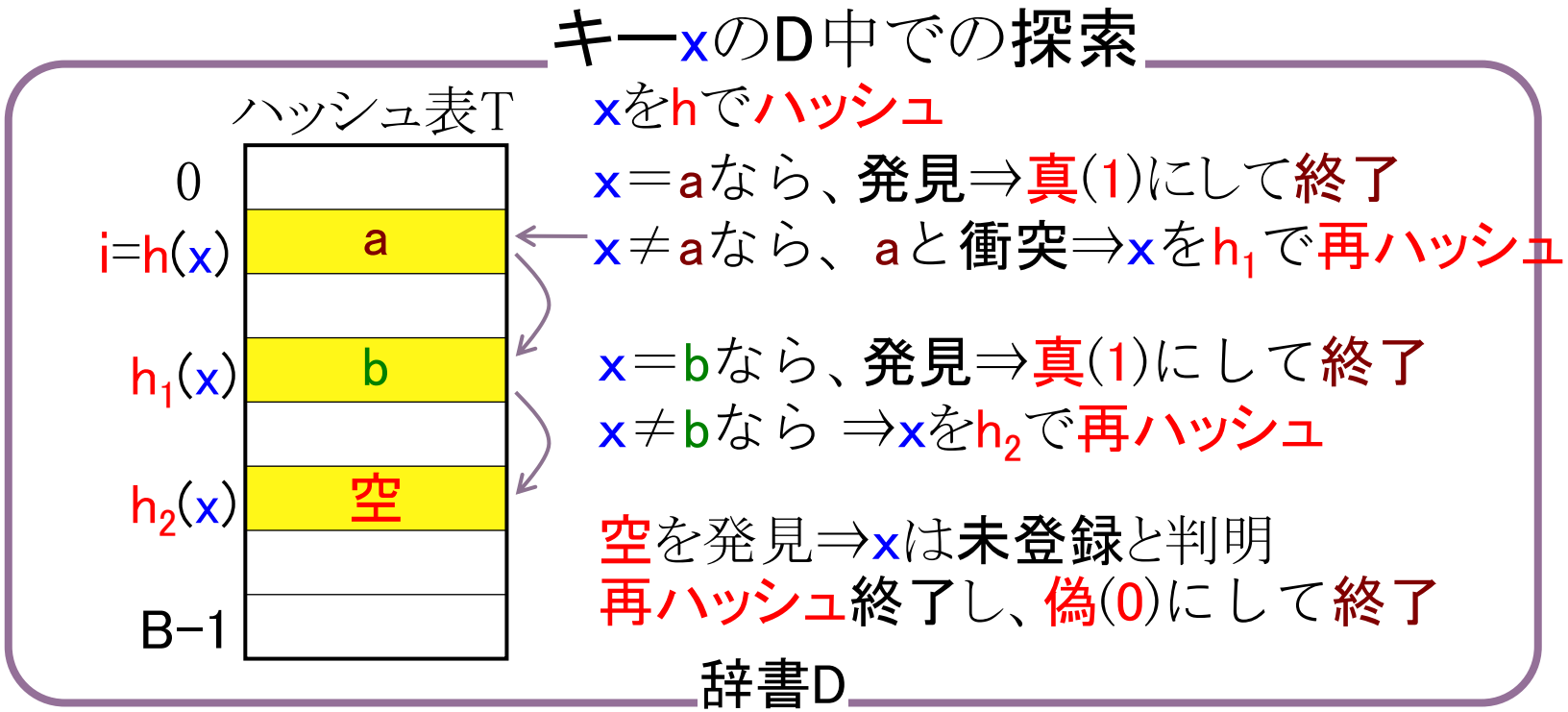
# 削除操作のない場合の挿入Insert(x, D)

1. **キーx**をハッシュとするハッシュ値 i =h(x) を求める
2. ハッシュ表の要素**T**[i]が
   - xならば，**既登録**より**終了**
   - **空**の状態ならば，**未登録**なので，ここにxを**登録**し**終了**
   - x**以外**のキーならば，**再ハッシュ**し再ハッシュ値 i を求め2.を繰り返す



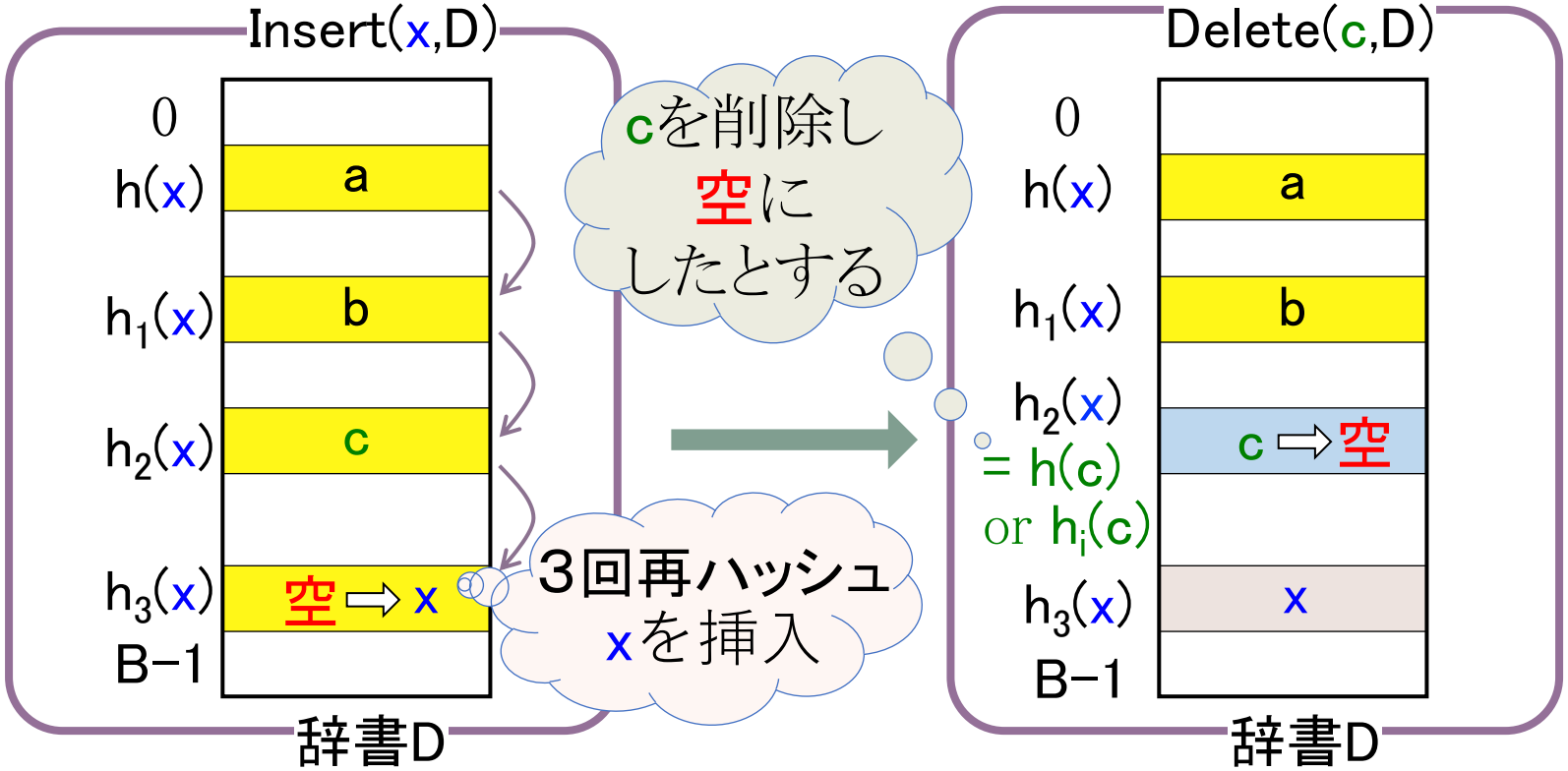
# 削除操作のない場合の所属Member (x, D)

1. **キーx**をハッシュとするハッシュ値 i =h(x) を求める
2. ハッシュ表の要素**T**[i]が
   - xならば，Member値を**真（1）**にして，**終了**
   - **空**の状態ならば，**未登録**なので，Member値を**偽（0）**にし**終了**
   - **x以外**のキーならば，**再ハッシュ**し再ハッシュ値 i を求め2.を繰り返す



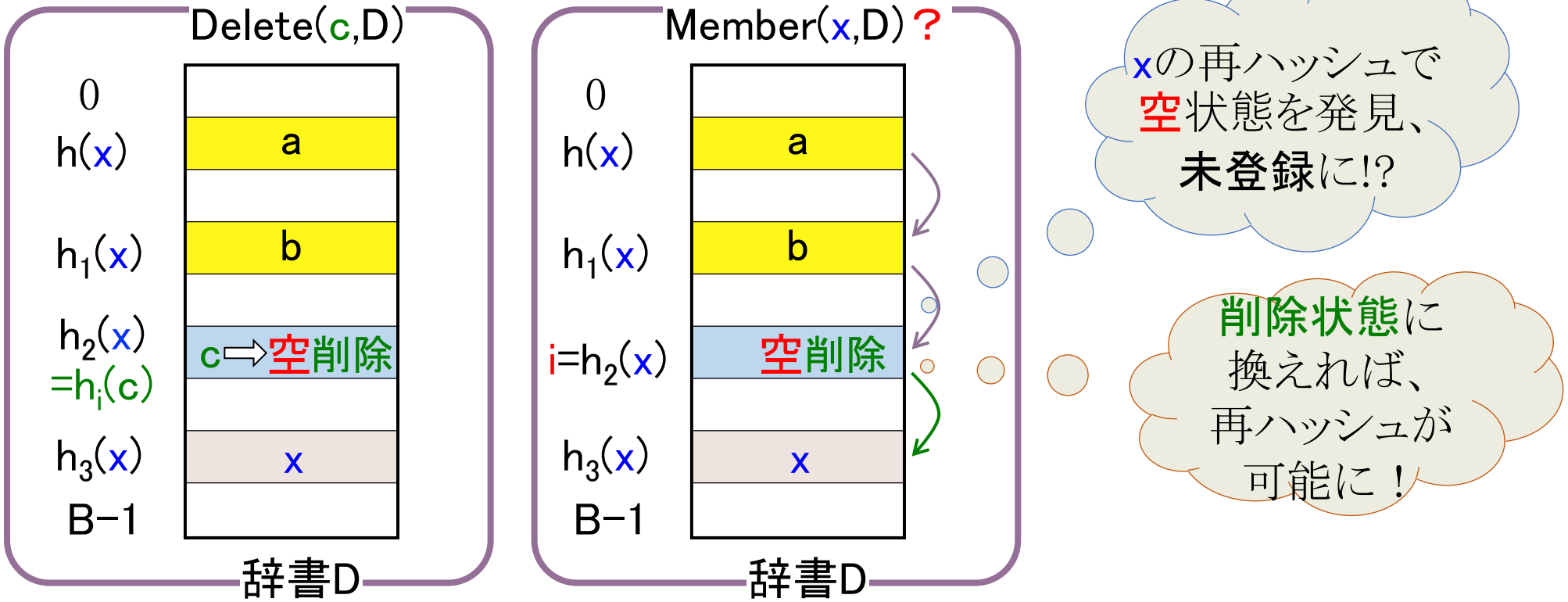
# 削除操作がある場合

- ハッシュ表の要素の状態
  - キー（辞書要素）でふさがっている**状態**
  - 一度でもキーでふさがったことのない**空状態**
  - 操作Deleteでキーが削除された**削除状態**



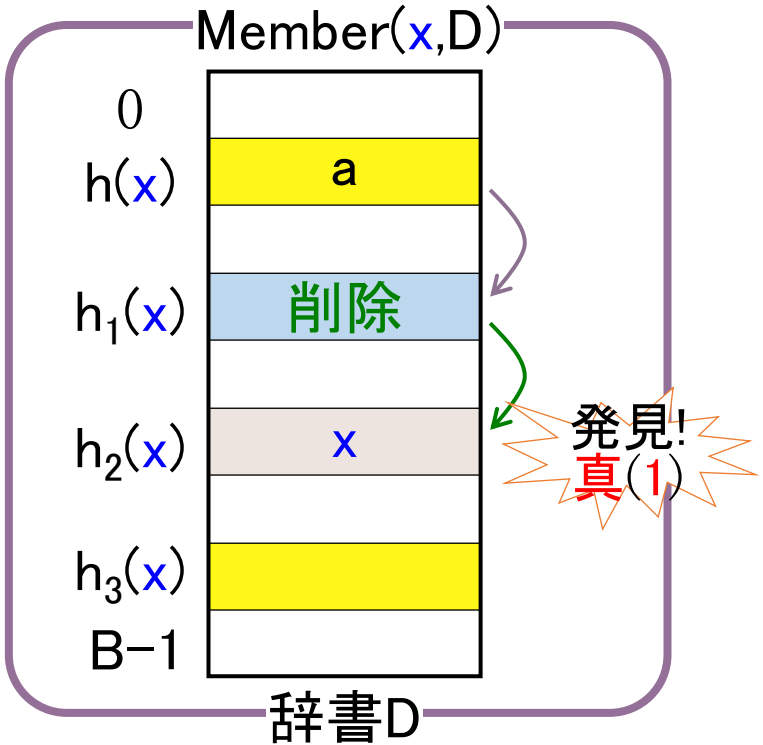
# 削除状態の必要性

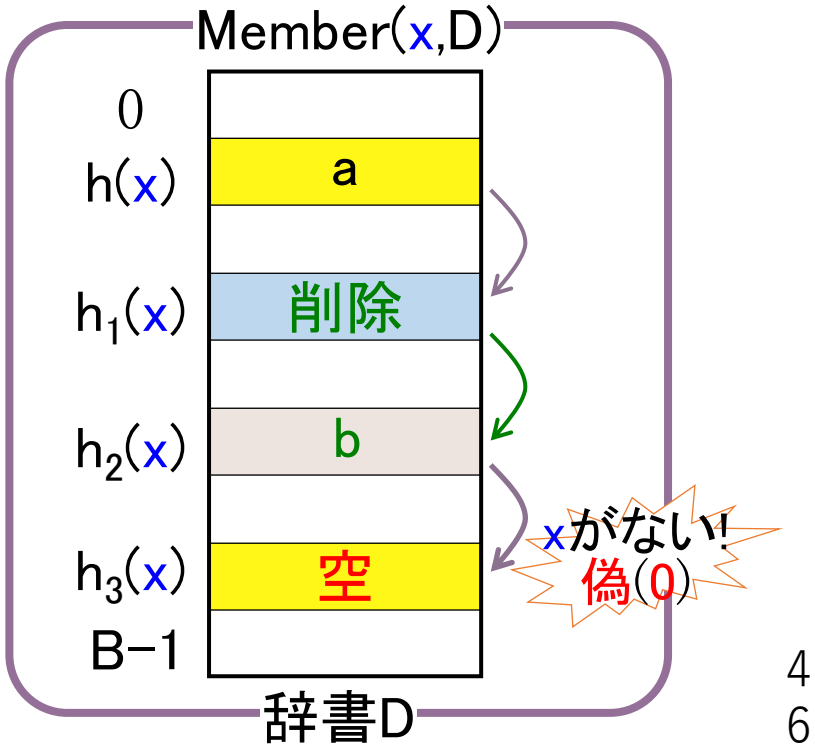
- キーcを削除して，ハッシュ表の要素T[i]を**空状態**にすると
  - 他の操作（例えば，xのMember操作の再ハッシュ）で，T[i]にきたとき，**空状態**のため**キーxがない**と誤った判断をしてしまう
  - 削除後，**削除状態**にし，この先の**再ハッシュの必要性**を示す



# 削除状態を設けた場合の所属Member(x,D)

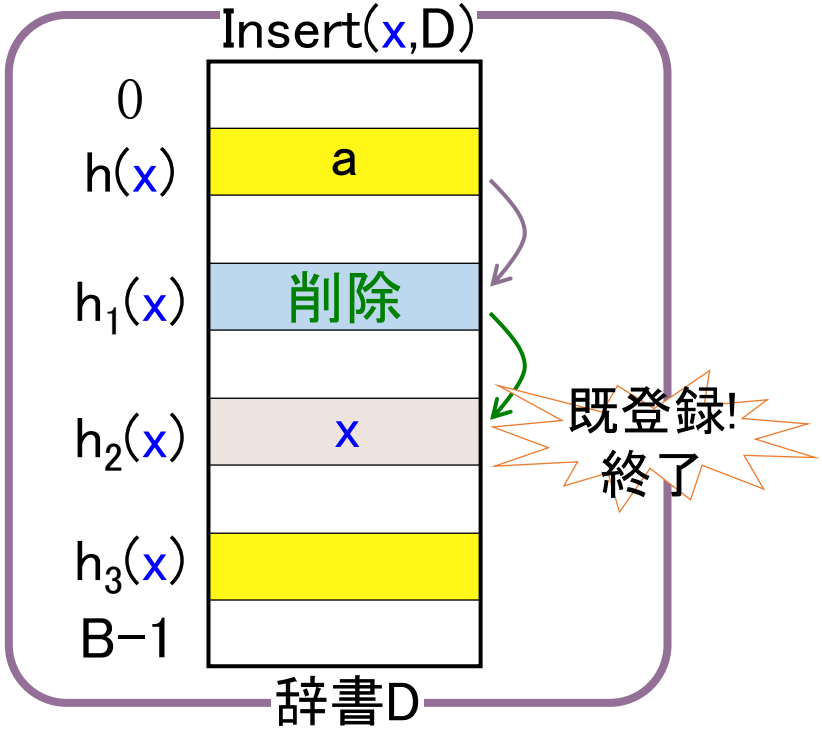
1. **キーx**をハッシュとするハッシュ値 i =h(x) を求める
2. ハッシュ表の要素**T**[i]が
   - xならば，既登録によりMember値を**真（1）**にして，**終了**
   - **空**の状態ならば，**未登録**なので，Member値を**偽（0）**にし**終了**
   - **x以外**のキーまたは**削除状態**ならば，**再ハッシュ**し再ハッシュ値 i を求め2.を繰り返す

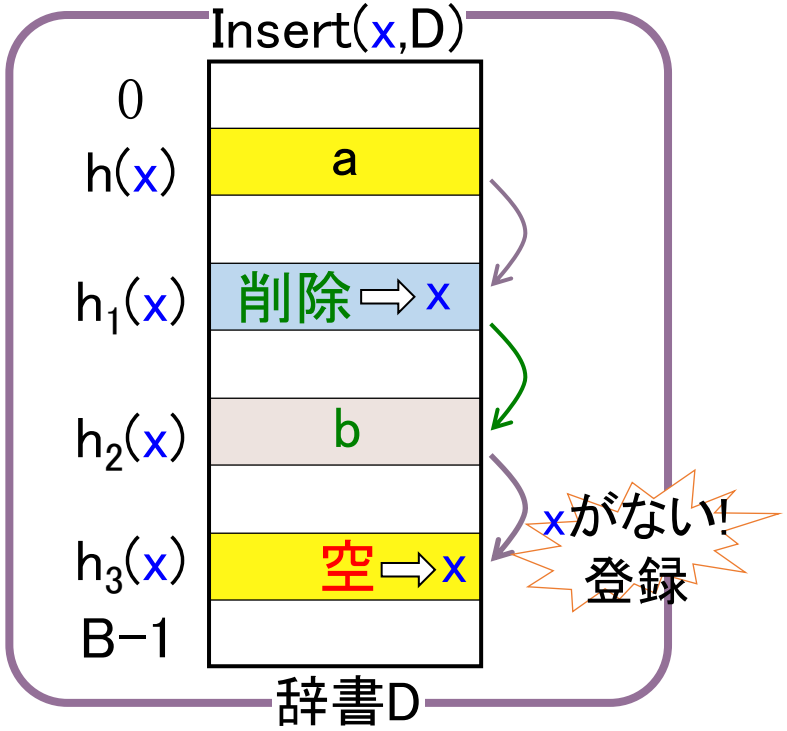


# 削除状態を設けた場合の挿入Insert(x, D)

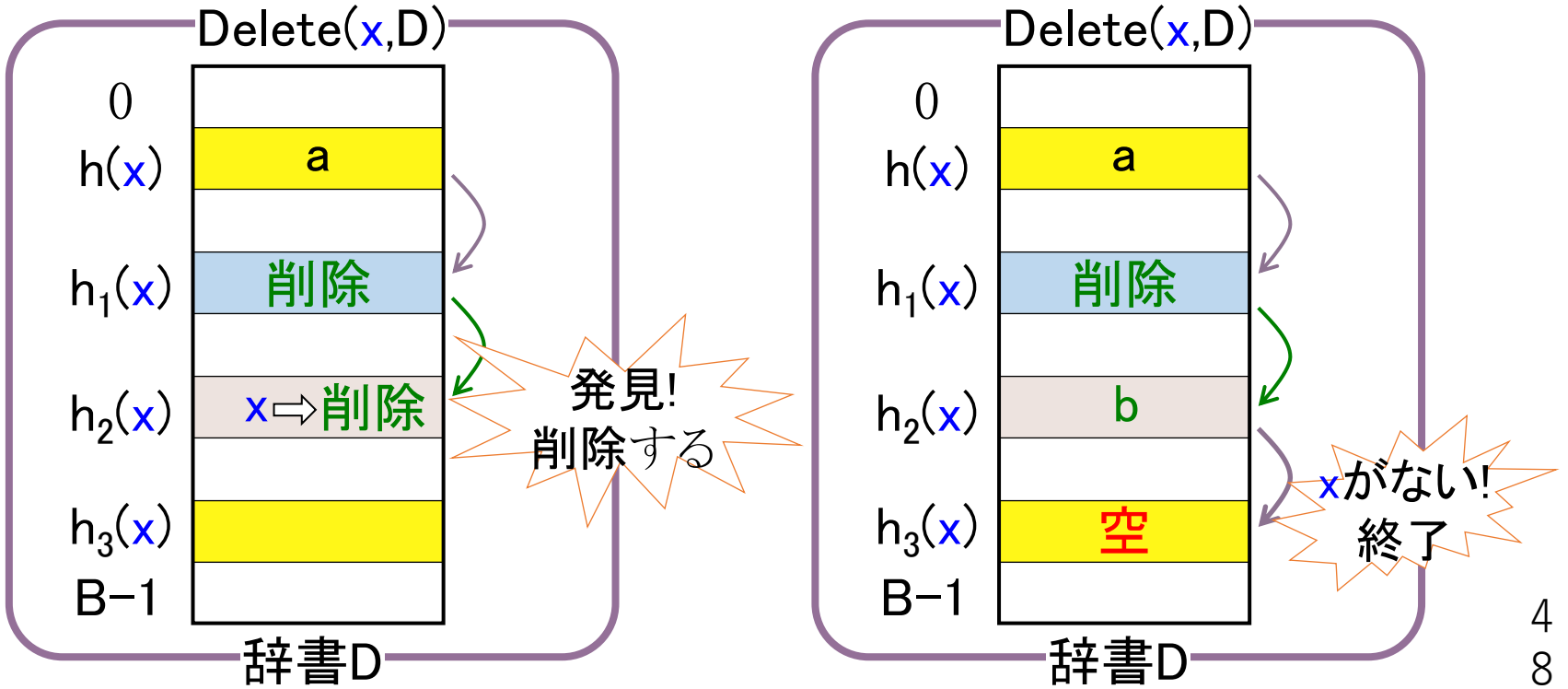
1. **キーx**をハッシュとするハッシュ値 i =h(x) を求める
2. ハッシュ表の要素**T**[i]が
   - xならば，**既登録**より**終了**
   - **空**の状態ならば，**未登録**なので，3. にいき**登録**
   - x**以外**のキーまたは**削除状態**ならば，**再ハッシュ**し再ハッシュ値 i を求め2.を繰り返す
3. **再ハッシュ中**に**削除状態**の要素があれば，そこにxを**登録**．なければ，最後の**空状態**に**登録**，終了



# 削除Delete(x, D)

1. **キーx**をハッシュとするハッシュ値 i =h(x) を求める
2. ハッシュ表の要素**T**[i]が
   - xならば，**削除状態**にして**終了**
   - **空**の状態ならば，**未登録**なので**終了**
   - **x以外**のキーまたは**削除状態**ならば，**再ハッシュ**し再ハッシュ値 i を求め2.を繰り返す



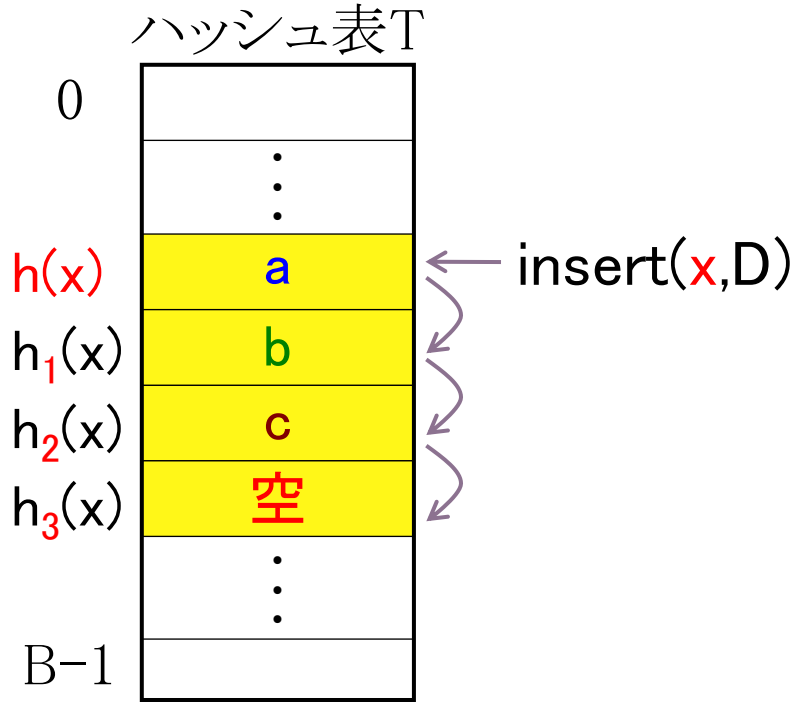
# 再ハッシュ関数の選定

- 線形検査法
- c個離れた要素を調べる
- 2次関数検査法
- ランダムな順列を用いる

# 再ハッシュ関数：線形検査法

- $h_i(x) = (h(x) + i) \% B$　$(i = 1, \cdots, B-1)$, Bはハッシュ表の大きさ
  - ハッシュ表を**環状**に考えて，**空**状態の要素が見つかるまで，
    $h(x) + 1$, $h(x) + 2$,…と次の位置を調べていく

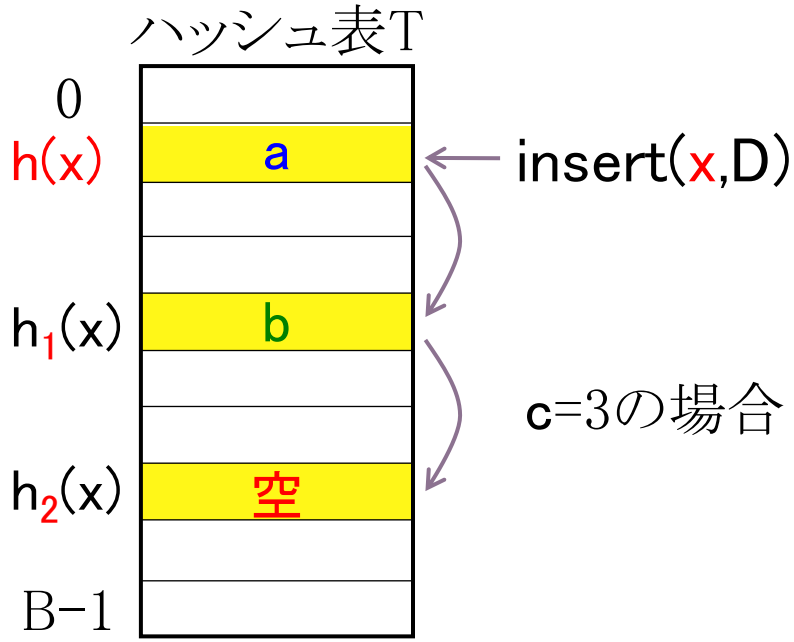
辞書要素が **h(x) の周り**に集中しやすい欠点を持つ



# 再ハッシュ関数：c個離れた要素

- $h_i(x) = (h(x) + ci) \% B \quad (i = 1, \cdots, B-1)$, cとBは**互いに素**
  - ハッシュ表を**環状**に考えて，**空**状態の要素が見つかるまで，
    h(x) + c, h(x) + 2c,…と次の位置を調べていく

線形検索法と同じく，c**個目**ごとに**団子状態**になる欠点を持つ



# 再ハッシュ関数：2次関数検査法

- $h_i(x) = (h(x) + i^2) \% B \quad (i = 1, \cdots, B-1)$
  - 団子状態は生じないが，ハッシュ表のすべてが検索されるとは限らない
  - Bが素数のときは，少なくともハッシュ表の半分は調べられる
  - 次の再帰的公式を用いれば，上述の再ハッシュ関数で，2乗演算を使わずにすむ

    $h_{i+1}(x) = h_i(x) + d_i(x)$
    $d_{i+1}(x) = d_i(x) + 2$
    $h_0(x) = h(x)$
    $d_0(x) = 1$

# 再ハッシュ関数：ランダムな順序

- $h_i(x) = (h(x) + d_i) \% B \quad (i = 1, \cdots, B-1)$
  - $d_1, d_2, \cdots, d_{B-1}$は，**ランダムな順列**で，シフトレジスタを用いて生成する方法がある

擬似乱数の生成方法

# クローズドハッシュの平均時間計算量

- ハッシュ表の大きさB
  - ハッシュ表は**N個**の辞書要素でふさがっており，どのふさがり型も**等確率**とする
- ハッシュ表に**N個登録**されている場合，**N+1個目**の新しい辞書要素を登録するときの平均時間計算量を考える
- **k回目**に空状態の要素を見つける**確率$P_k$**を考える

$$P_1 = \frac{B-N}{B}$$

$$P_2 = \frac{N}{B}\frac{B-N}{B-1}$$

$$\cdots$$

$$P_k = \frac{N}{B}\frac{N-1}{B-1}\cdots\frac{N-(k-2)}{B-(k-2)}\frac{B-N}{B-(k-1)}$$

# N+1個目の辞書要素の登録

- **N+1個目**の新しい辞書要素をハッシュ表に**登録**するときの時間計算量を考える
  - **k回目**に**空**状態の要素を見つける確率$P_k$

$$P_k = \frac{N}{B}\frac{N-1}{B-1}\cdots\frac{N-(k-2)}{B-(k-2)}\frac{B-N}{B-(k-1)}$$

〈N＋1個目登録の平均時間計算量〉

$$= \sum_{k=1}^{N+1} kP_k = \frac{B+1}{B+1-N} \approx \frac{B}{B-N} = \frac{1}{1-\alpha}$$

ただし、$\alpha$＝N/Bで、
ハッシュ表中の辞書要素の割合

# 1個登録に要する平均時間計算量

- ハッシュ表にN個まで登録したとき，1個登録に要した平均時間計算量は，

〈**N個まで登録**したときの**1個登録**平均時間計算量〉

$$= \frac{1}{N}\sum_{k=1}^{N} \langle \text{k 個目登録の平均時間計算量} \rangle$$

$$= \frac{1}{N}\sum_{k=1}^{N} \frac{B+1}{B+2-k} = \frac{B+1}{N}\ln\frac{B+1}{B+2-N} \fallingdotseq \frac{B}{N}\ln\frac{B}{B-N}$$

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

ただし、$\alpha$＝N/Bで、
ハッシュ表中の辞書要素の割合

  - ハッシュ表の90%($\alpha$=0.9)を辞書要素でうめたときの1個登録に要した平均時間計算量は、2.56。

# 探索に要する平均時間計算量

- ハッシュ表に**N個まで**登録したとき

〈**表にない要素を探す**平均時間計算量〉

＝〈N＋１個目登録の平均時間計算量〉

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

ただし、$\alpha$＝N/Bで、
ハッシュ表中の辞書要素の割合

〈**表にある要素を探す**平均時間計算量〉

＝〈**N個まで登録**したときの**1個登録**平均時間計算量〉

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

# 削除に要する平均時間計算量

- ハッシュ表にN個まで登録したとき

〈**要素**を**削除する**平均時間計算量〉

=〈その要素を探索する平均時間計算量〉

# クローズドハッシュの平均時間計算量まとめ

- 登録要素数Nがハッシュ表の90%以上（$\alpha > 0.9$）のとき
  - **表中にある要素**を探す平均時間計算量は**2.56以上**
  - **表中にない要素**を探す平均時間計算量は**10以上**

- **表の再構成**
  - 登録要素数がハッシュ表の**90%以上**を占めたとき
  - 大きさが2倍の**新しいハッシュ表**を作る

  ⇒それぞれの平均時間計算量が**2.56以下**および**10以下**に保つことはできる

  ⇒**登録要素数N**は**表の大きさB**の**90%**程度

- 最小の要素を取り出す操作**Min**の効率のよいアルゴリズムはない

# まとめ

- 集合
  - 集合の仕様
  - 実現：ビットベクトル
  - 実現：連結リスト

- 辞書
  - 辞書の仕様
  - 実現：配列へのベタ詰め法
  - 実現：ハッシュ法
    - オープンハッシュ法
    - クローズドハッシュ法

# 演習1 オープンハッシュ法

- 抽象データ型辞書を**オープンハッシュ法**で実現する.
- 辞書操作, **挿入, 探索(所属), 削除**について以下の設問に答えよ.
- ただし,
  - ハッシュ表の大きさBは5, インデックスは0からB−1までとする.
  - **辞書要素(キー)の集合** = {0,1,・・・,14} とする.
  - **ハッシュ関数** h(x) = x % Bとする.
  - 最初、**ハッシュ表**(辞書)には何も登録されていないとする.

1. 5つの**辞書要素(キー)** 2, 8, 14, 3, 9 をこの順でハッシュ表に登録する. このときのハッシュ表への**登録手順を概説**し, できあがった**ハッシュ表の図を描け**.
2. つづいて、9, 13がハッシュ表に登録されているか否かを調べる. このときのハッシュ表の**探索手順を概説**せよ.
3. 最後に、9, 13をハッシュ表から**削除**する. このときのハッシュ表の**削除手順を概説**し, できあがった**ハッシュ表の図を描け**.

# 演習2 クローズドハッシュ法

- 抽象データ型辞書Dを**クローズドハッシュ法**で実現する.
- ただし,
  - 辞書Dの**ハッシュ表T**の大きさBは11, そのインデックスは0からB−1とする.
  - 辞書要素(キー)の集合 = {0,1,･･･,109} とする.
  - ‘空の状態’ を −1, ‘削除状態’ を −2 で表す.
  - ハッシュ関数 h(x) = x % B とする.
  - 再ハッシュ関数 $h_i(x)$＝( h(x) + 3 * i ) % B とする.
  - いま、辞書D(ハッシュ表T)には右図のように、キーが登録されており、以下の設問を順に操作していくとする.

ハッシュ表T

| | |
|---|---|
| 0 | −1 |
| 1 | −1 |
| 2 | 32 |
| 3 | −1 |
| 4 | −1 |
| 5 | −1 |
| 6 | −1 |
| 7 | 18 |
| 8 | −1 |
| 9 | −1 |
| 10 | 21 |

1. 辞書要素(キー) 7を登録する。そのときの**ハッシュ，再ハッシュの手順**を説明し，結果の**ハッシュ表Tを図示**せよ.
2. 辞書要素(キー) 32を削除する. そのときの**ハッシュ，再ハッシュの手順**を説明し，結果の**ハッシュ表Tを図示**せよ.
3. 再び、辞書要素(キー) 7を登録するときの**ハッシュ、再ハッシュの手順**を説明し，結果の**ハッシュ表Tを図示**せよ.

# 提出方法

- ドローイングソフトを使ってもかまいませんが，手書きを写真でとったものでOKです.
- pdfや画像フォーマットで提出してください

- 提出方法：LETUS
- 締め切り：2023/7/10 10:30まで