５．サーチ

５－１．線形探索

５－２．２分探索

５－３．ハッシュ

サーチ問題

- 入力：n個のデータ

$$a_0, a_1, \cdots, a_{n-1}$$

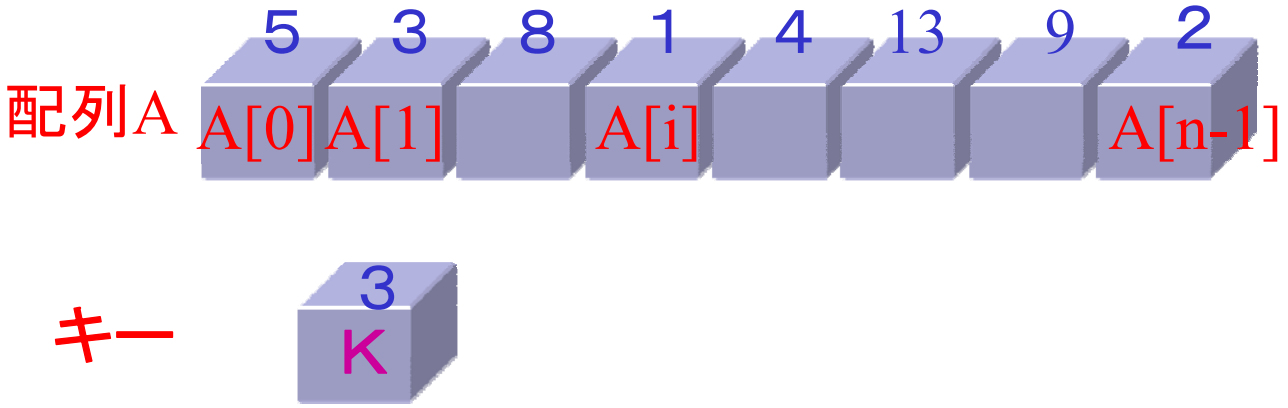
（ここで、入力サイズは、n とします。）

さらに、キー k

- 出力：
  - ✧ $k = a_i$ となる a_i があるときは、
    その位置 $i, (0 \leq i \leq n-1)$
  - ✧ キーが存在しないとき、－1

探索（サーチ）

入力：



出力：

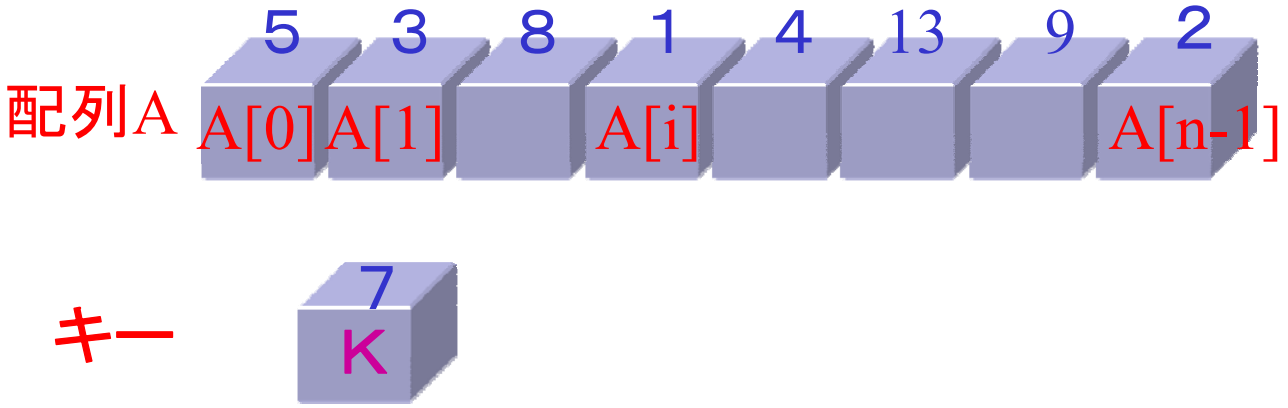
キーが存在：キーを保持している配列のインデックス（添え字）

1

A[1]にキーKの値が保存されている。

キーがない場合

入力:



出力:

キーが存在しない:データがないことを意味する値

－1

サーチ問題の重要性

- 実際に頻繁に利用される。（検索も、探索の応用である。）
- 多量のデータになるほど、計算機を用いた探索が重要。

計算機が扱うデータ量は、増加する一方である。

探索問題が益々重要。

サーチアルゴリズムの種類

- 線形探索

  素朴な探索技法
- ２分探索

  理論的に最適な探索技法
- ハッシュ

  応用上重要な探索技法

5-1：線形探索
（逐次探索）

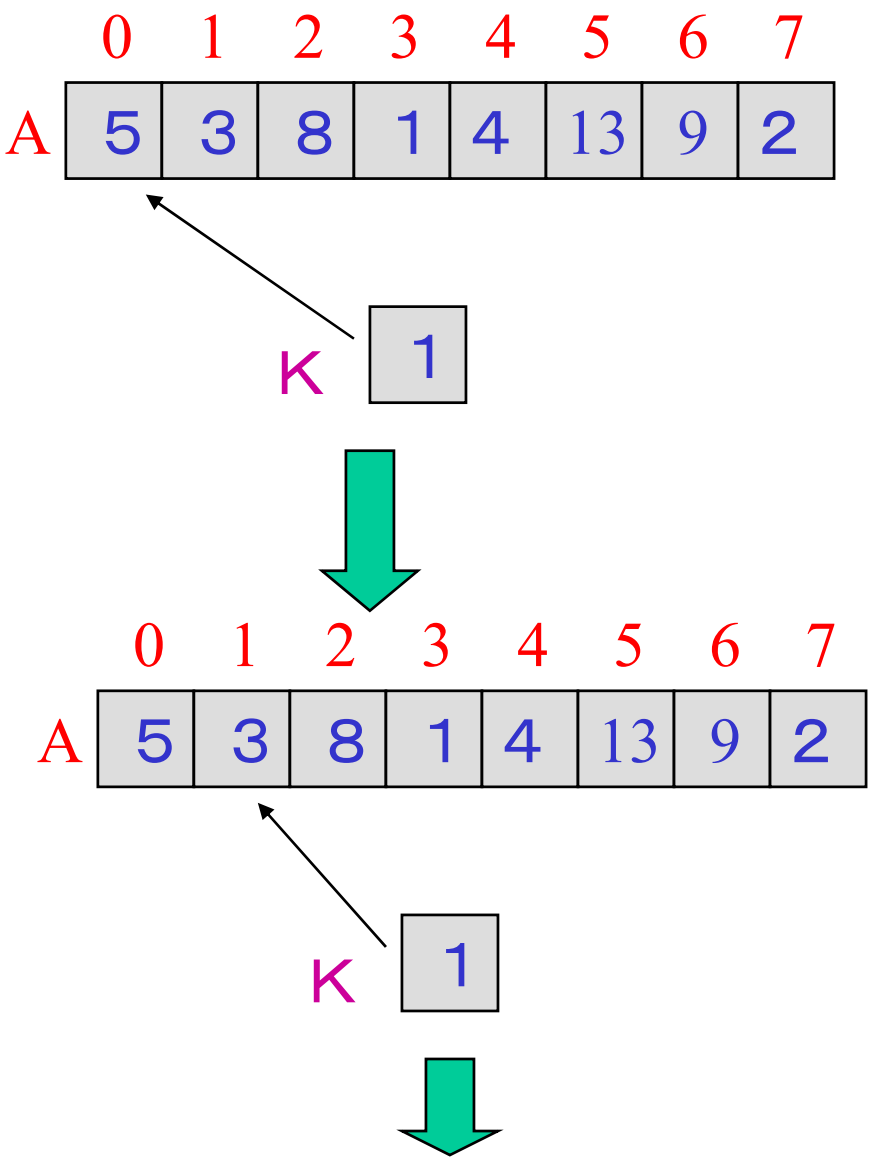
線形探索

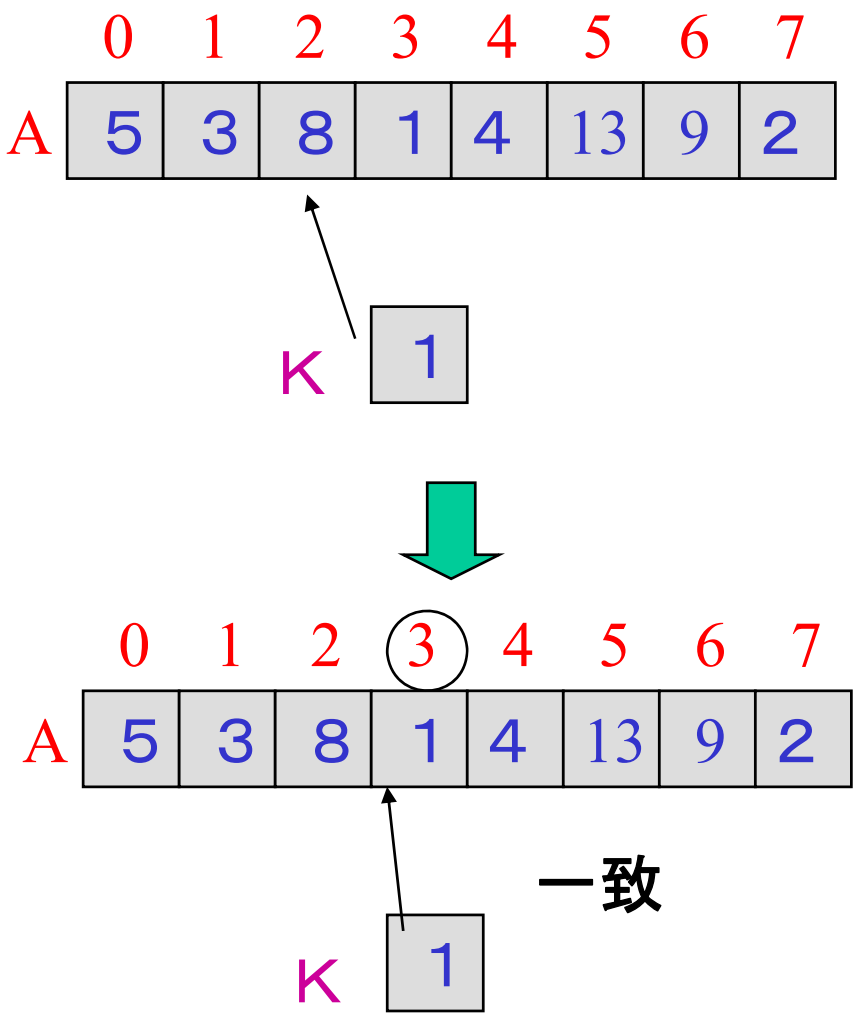
方針

- 前からキーと一致するかを順に調べる。
- 配列の最後かをチェックする。
- もし、配列の最後までキーが存在しなければ、
  キーは存在しない。

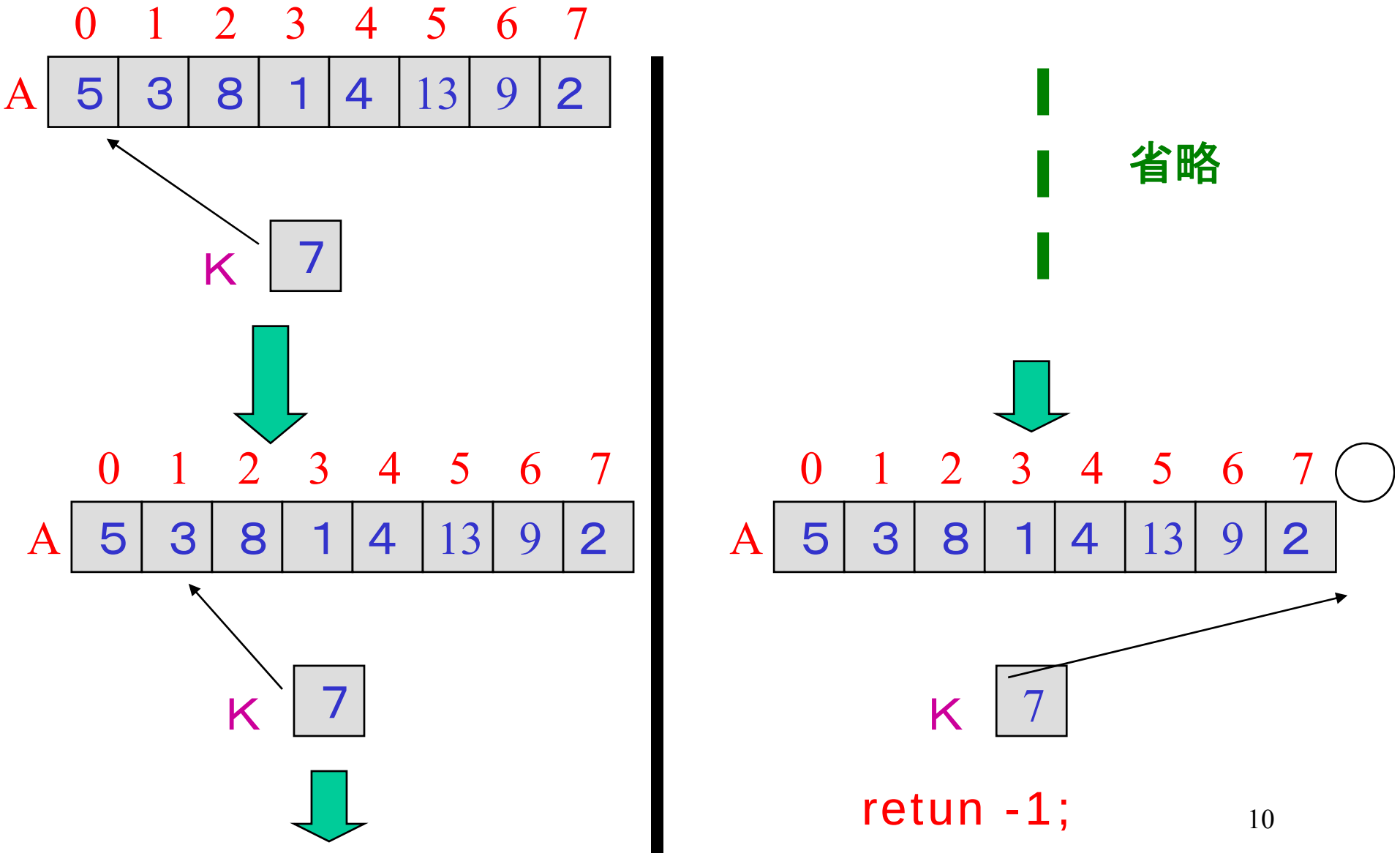
最も直感的で、素朴なアルゴリズム。
しかし、このアルゴリズムにも注意点がある。

線形探索の動き


0 1 2 3 4 5 6 7
A 5 3 8 1 4 13 9 2
K 1
0 1 2 3 4 5 6 7
A 5 3 8 1 4 13 9 2
K 1


0 1 2 3 4 5 6 7
A 5 3 8 1 4 13 9 2
K 1
0 1 2 3 4 5 6 7
A 5 3 8 1 4 13 9 2
一致
K 1
retun 3;

線形探索の動き2
（データが無い場合）



線形探索の実現

```
/*　線形探索
引数:キー
戻り値:キーを保持している配列の添え字
*/
1. int linear_search(double k)
2. {
3.      int  i;   /* カウンタ*/
4.      for(i=0;i<n-1;i++)
5.        {
6.            if(A[i]==k)
7.                {
8.                    return i;  /*添え字を戻す*/
9.                 }
10.         }
11.    return -1;  /*未発見*/
12.}
```

命題LS1（linear_seachの正当性1）

forループがp回繰り返される必要十分条件は、
A[0]-A[p-1]にキーkと同じ値が存在しない。

命題LS2（linear_seachの正当性2）

キーと同じ値が存在すれば、
添え字が最小のものが求められる。

これらは、明らかに成り立つ。

線形探索の最悪計算量

配列中にキーが存在しないときが最悪である。

このときは、明らかに、すべての配列が走査される。

したがって、

$O(n)$ **時間のアルゴリズム**

線形探索の平均時間計算量

配列中にキーが存在する場合を考える。

キーが、各位置に対して等確率で保持されていると仮定する。

$$T(n) = \frac{1}{n}\{1 + 2 + 3 + \cdots + n\}$$

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

$$= \frac{n+1}{2}$$

$O(\frac{n}{2}) = O(n)$ **時間のアルゴリズム**

線形探索の注意事項

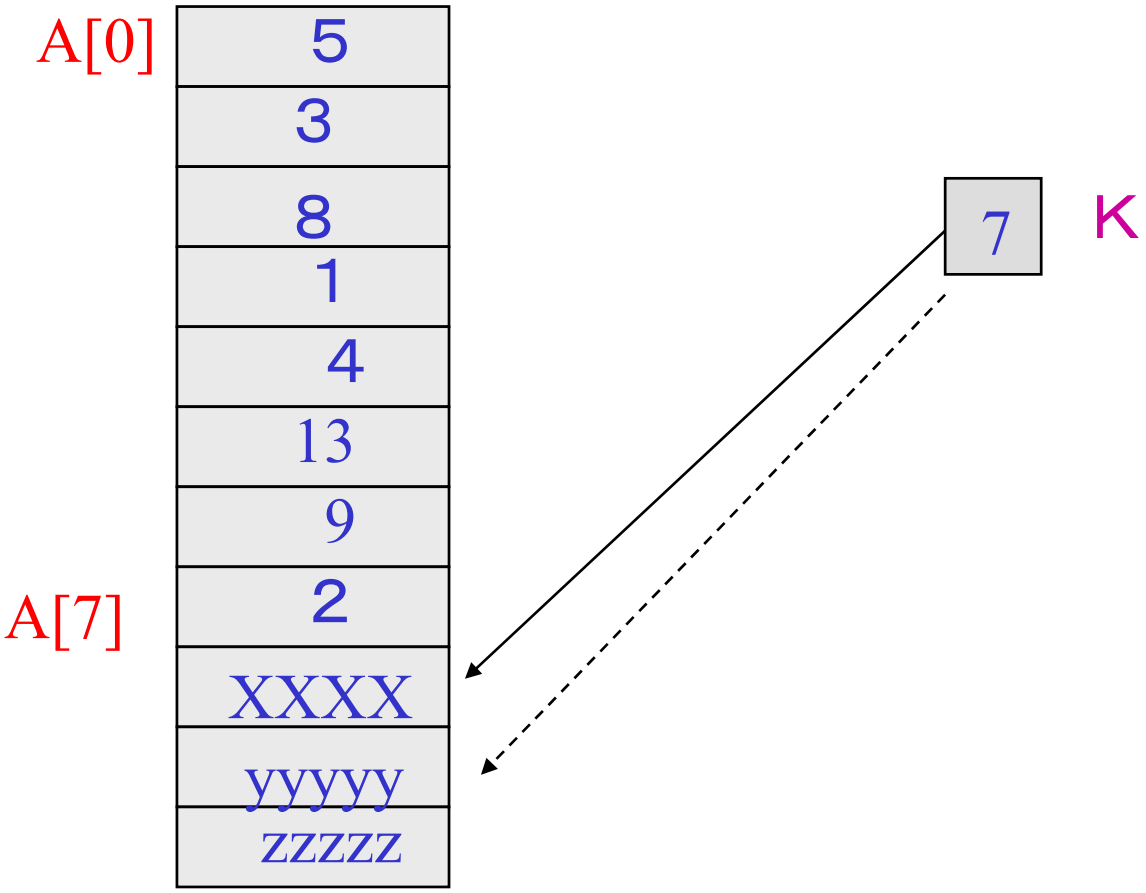
単純に前から走査するだけだと、
配列の範囲を超えて走査することがある。
（正当性では、キーの存在しない範囲を
増加させているだけに注意する。）

バッファーオーバーラン
というプログラムの不備である。

危険なプログラム

```
/*  危険な線形探索
    配列中にキーが存在しないときに、終了しない。*/
1.  int linear_search(double k)
2.  {
3.      int  i=0;   /* カウンタ*/
4.      while(A[i]!=k){
5.          i++;
6.       }
7.      return i;  /* 添え字を返す*/
8.  }
```

配列を超えて走査するバグ



番兵付の線形探索

番兵付の線形探索

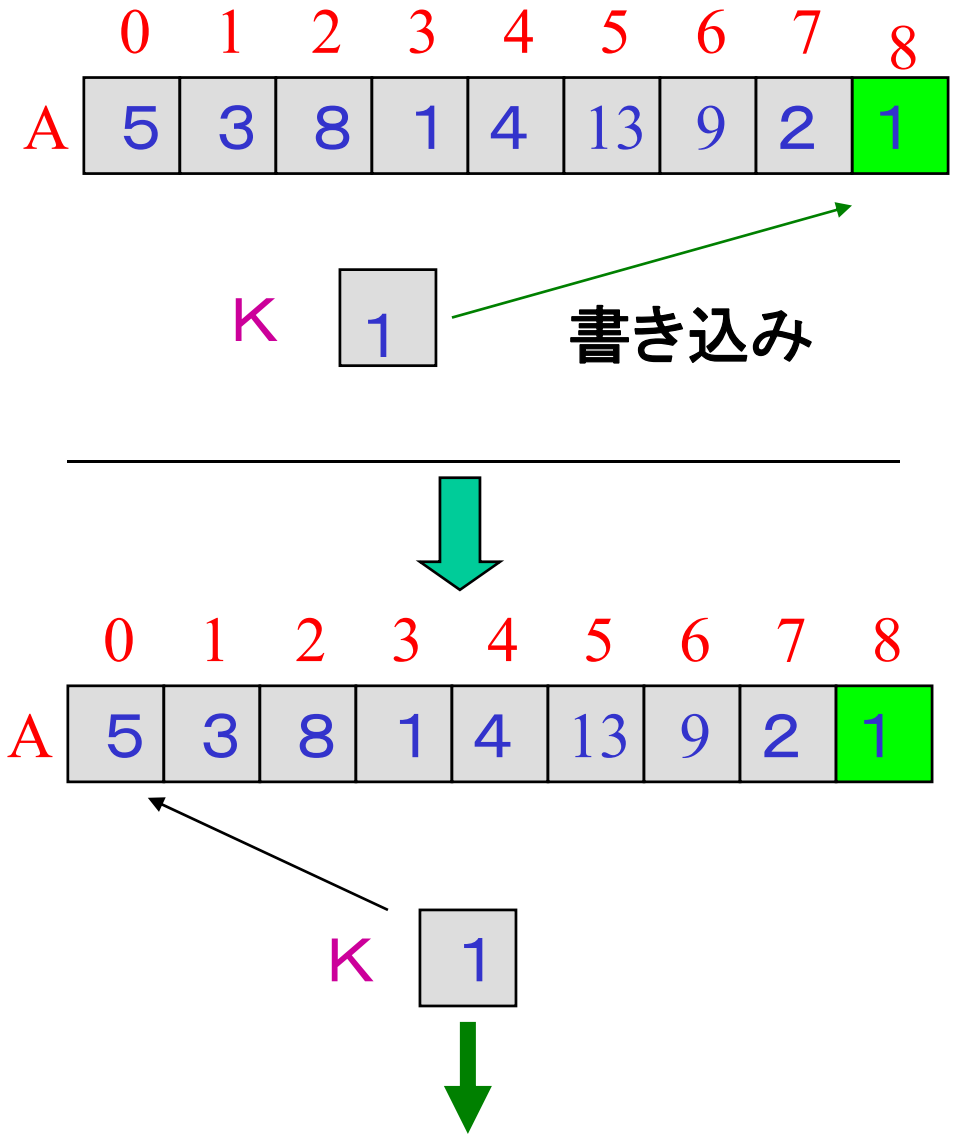
アィディア

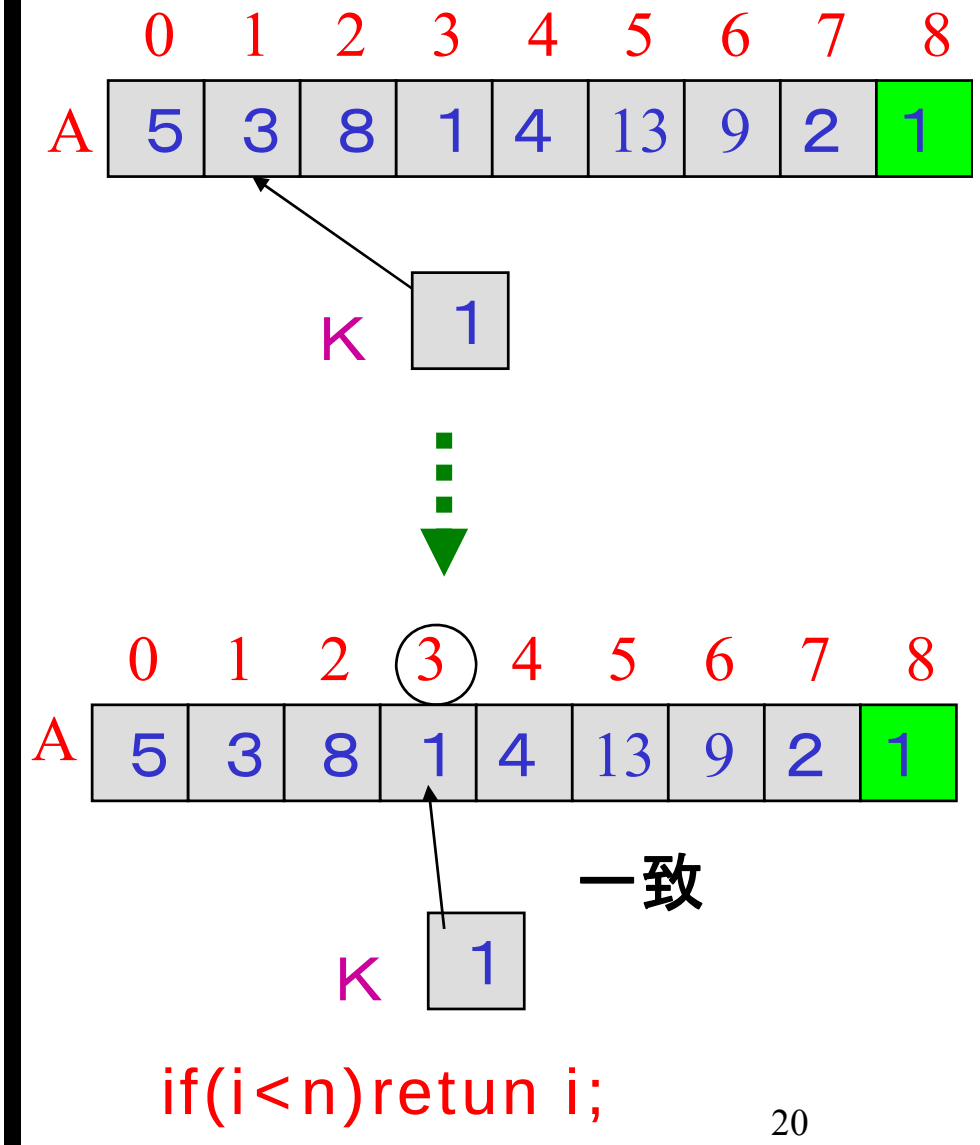
- 必ずキーが存在するように設定してから、線形探索をおこなう。

効果

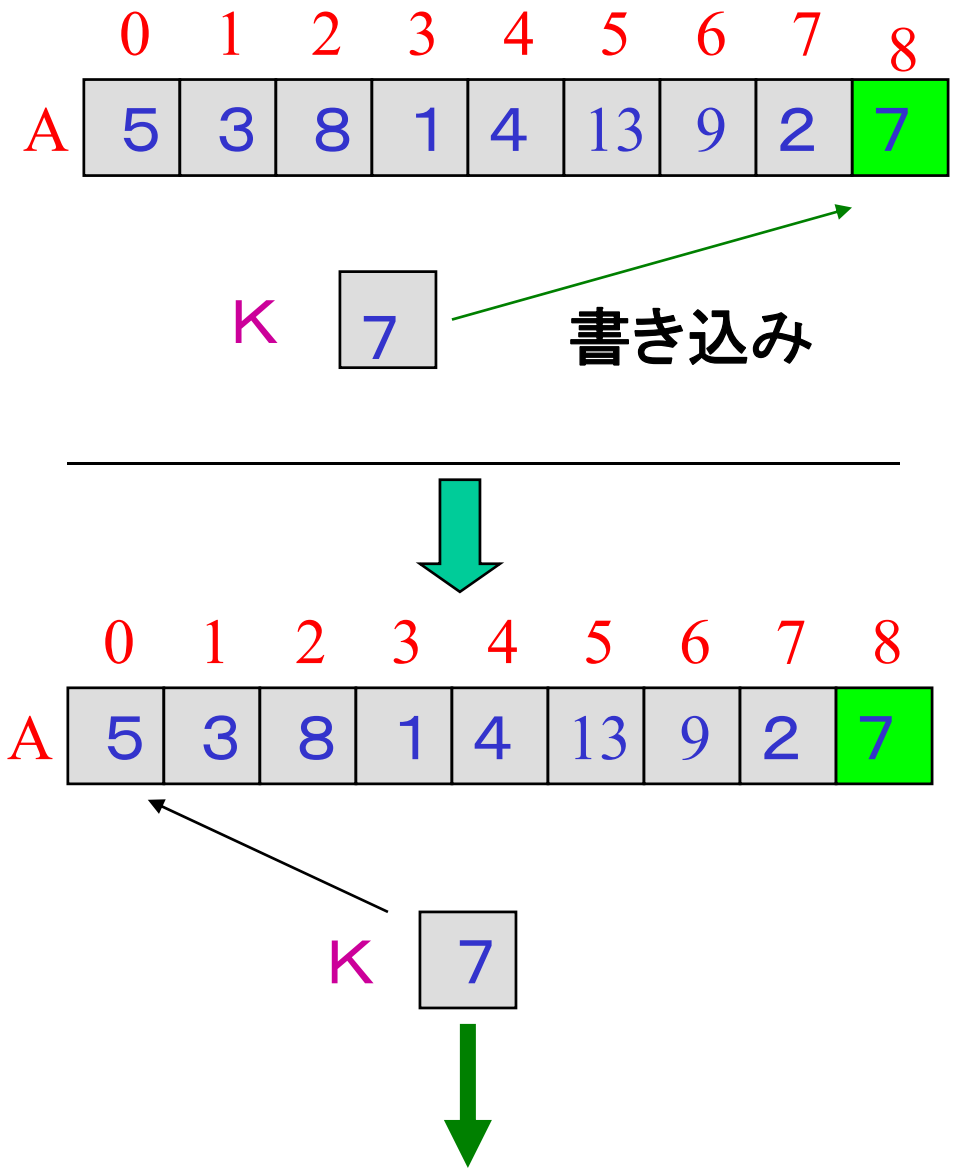
- バッファーオーバーランを無くせる。
- 比較回数を約半分に減らせる。

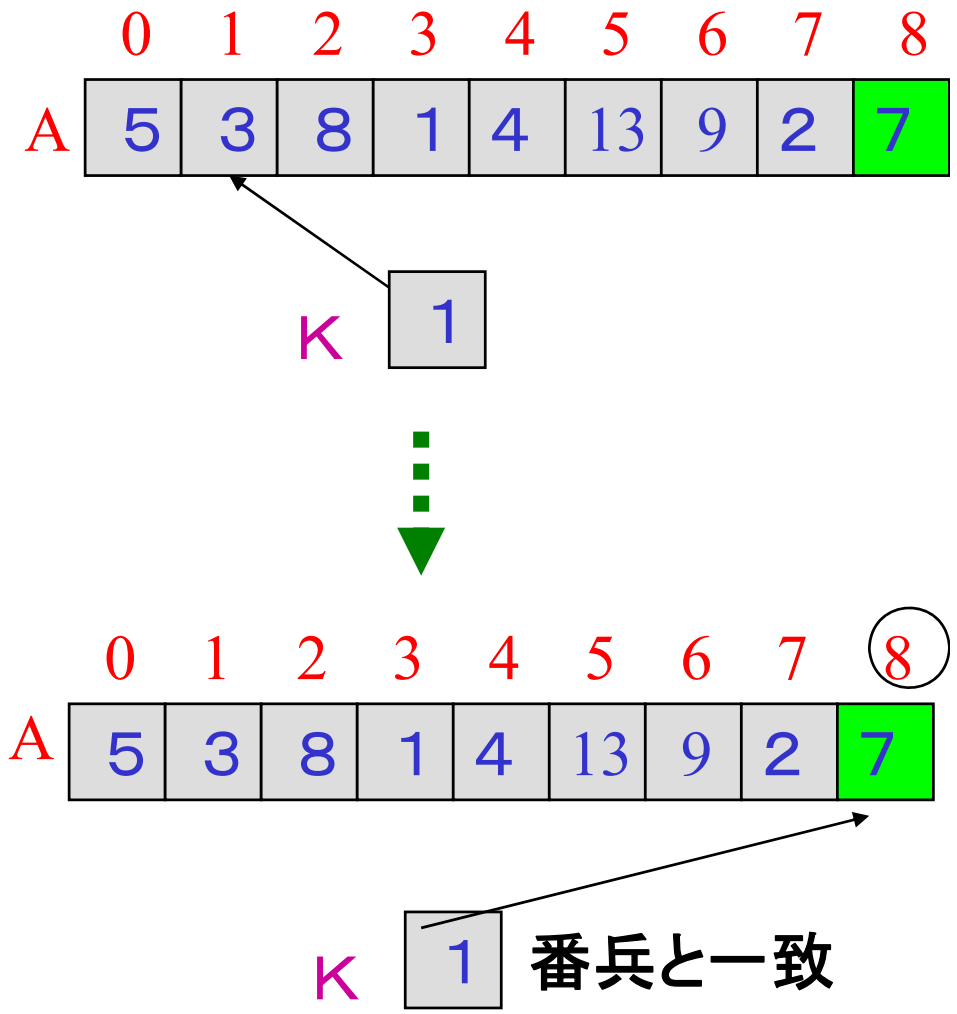
番兵付き線形探索（キーがある場合）





番兵付き線系探索(キーが無い場合)






```
if(i==n)retun -1;
```

番兵付線形探索の実現

```
/*  番兵付き線形探索*/
1. int banpei_search(double k)
2. {
3.      int  i=0;   /* カウンタ*/
4.      A[n]=k;    /*番兵の設定*/
5.      while(A[i]!=k) {
6.            i++;
7.         }
8.     }
9.     if(i<n){
10.        return i; /*見つかった場合*/
11.    }else {
12.       return -1;/*見つからない場合*/
13.    }
14.}
```

命題BAN1（banpei_seachの停止性）

banpei_searhは必ず停止する。

キーが必ずA[0]-A[n]中に存在するので
ステップ5の条件が必ず偽になり停止する。

番兵付線形探索の計算量

最悪時間計算量、平均時間計算量ともに、
線形探索と同じである。

$O(n)$ **時間のアルゴリズム**

実は、番兵を用いない線形探索では、各繰り返しにおいて、
配列の範囲のチェックとキーのチェックの２回の比較を行っている。
一方、番兵を用いると、配列の範囲チェックを毎回行う必要がない。したがって、比較回数は約半分にすることができる。

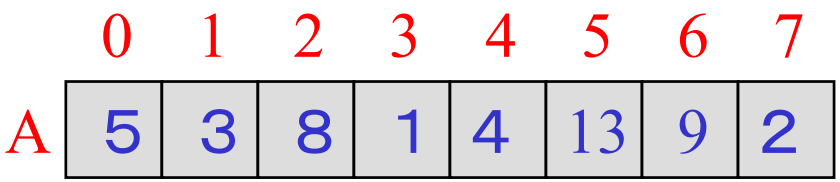
5-2：2分探索

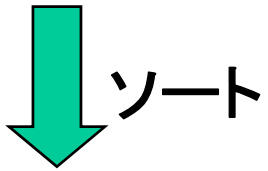
2分探索

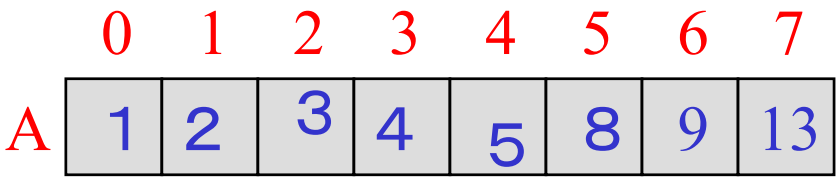
アィディア

- 配列をあらかじめソートしておけば、
  一回の比較でキーの存在していない範囲を
  大幅に特定できる。
- 探索範囲の半分の位置を調べれば、
  探索範囲を繰り返し事に半分にできる。
- 探索範囲が小さくなれば、サイズの小さい
  同じタイプの問題→再帰的に処理すればよい。
- 探索範囲の大きさが1であれば、
  キーそのものか、もとの配列にキーが存在しないか、
  のどちらかである。

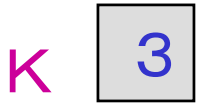
2分探索の動き
(キーが存在する場合)





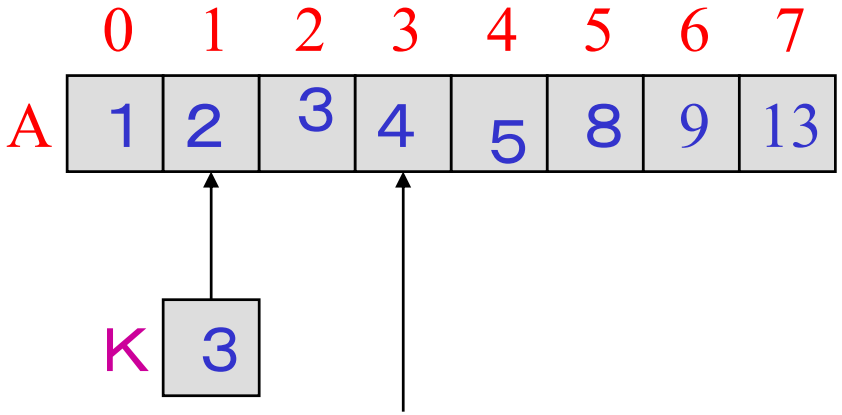


$mid = \left\lfloor \frac{0+(n-1)}{2} \right\rfloor = 3$



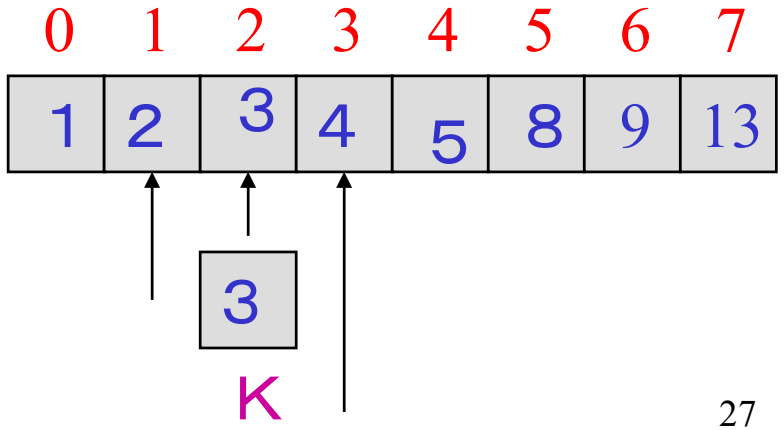
$k < A[mid]$

$mid = \left\lfloor \frac{0+2}{2} \right\rfloor = 1$



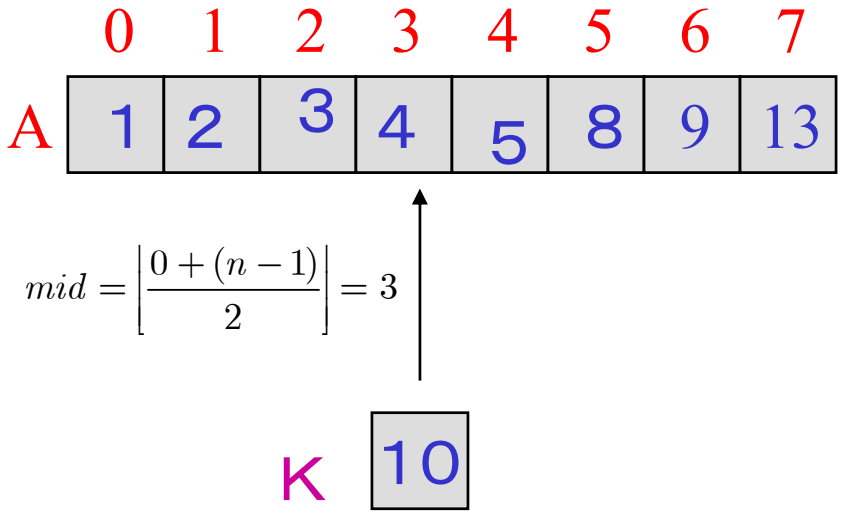
$mid = \left\lfloor \frac{2+2}{2} \right\rfloor = 2$

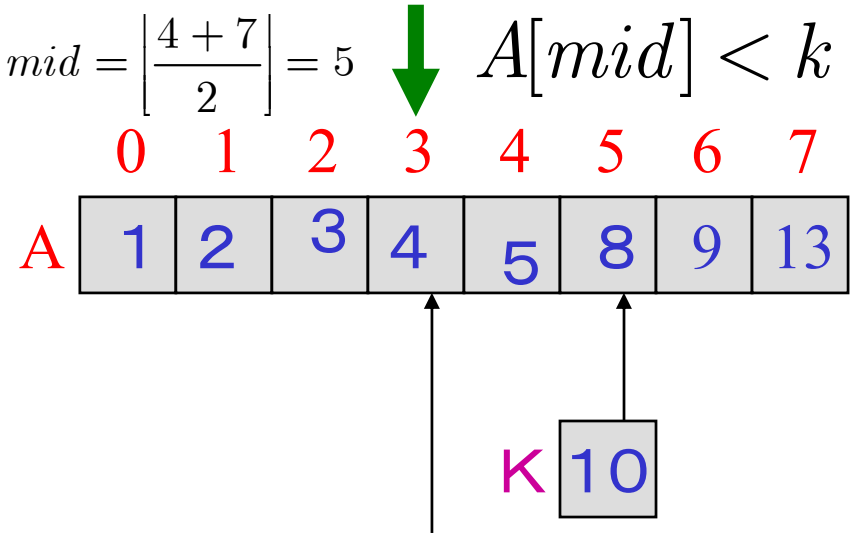
$A[mid] < k$

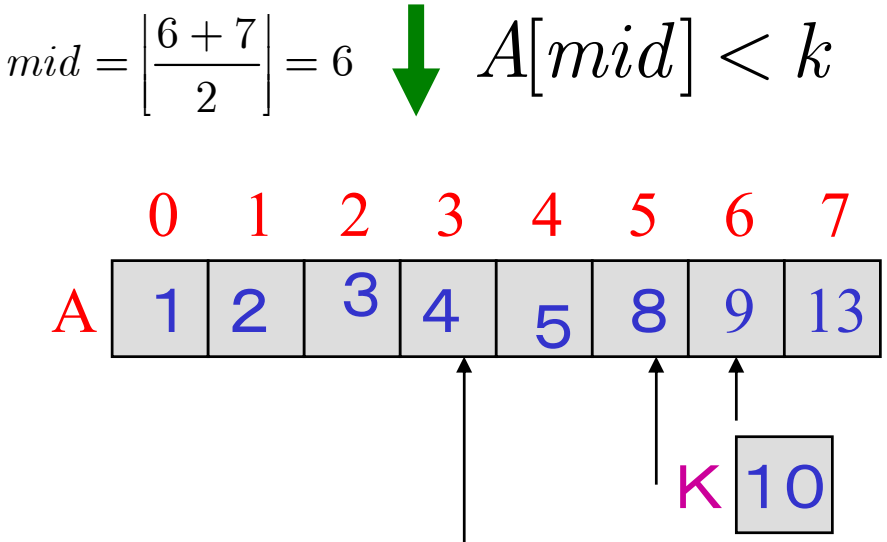


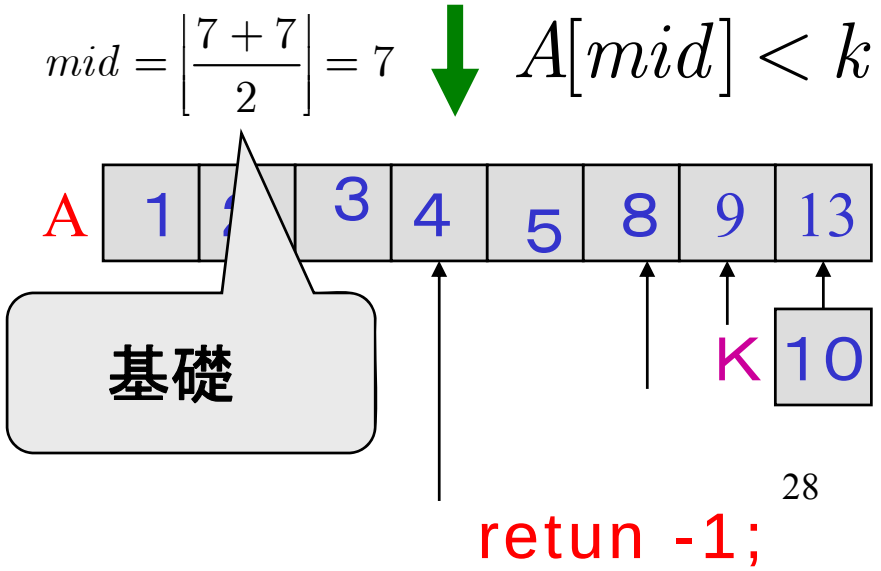
retun 3;

2分探索の動き
(キーが存在しない場合)

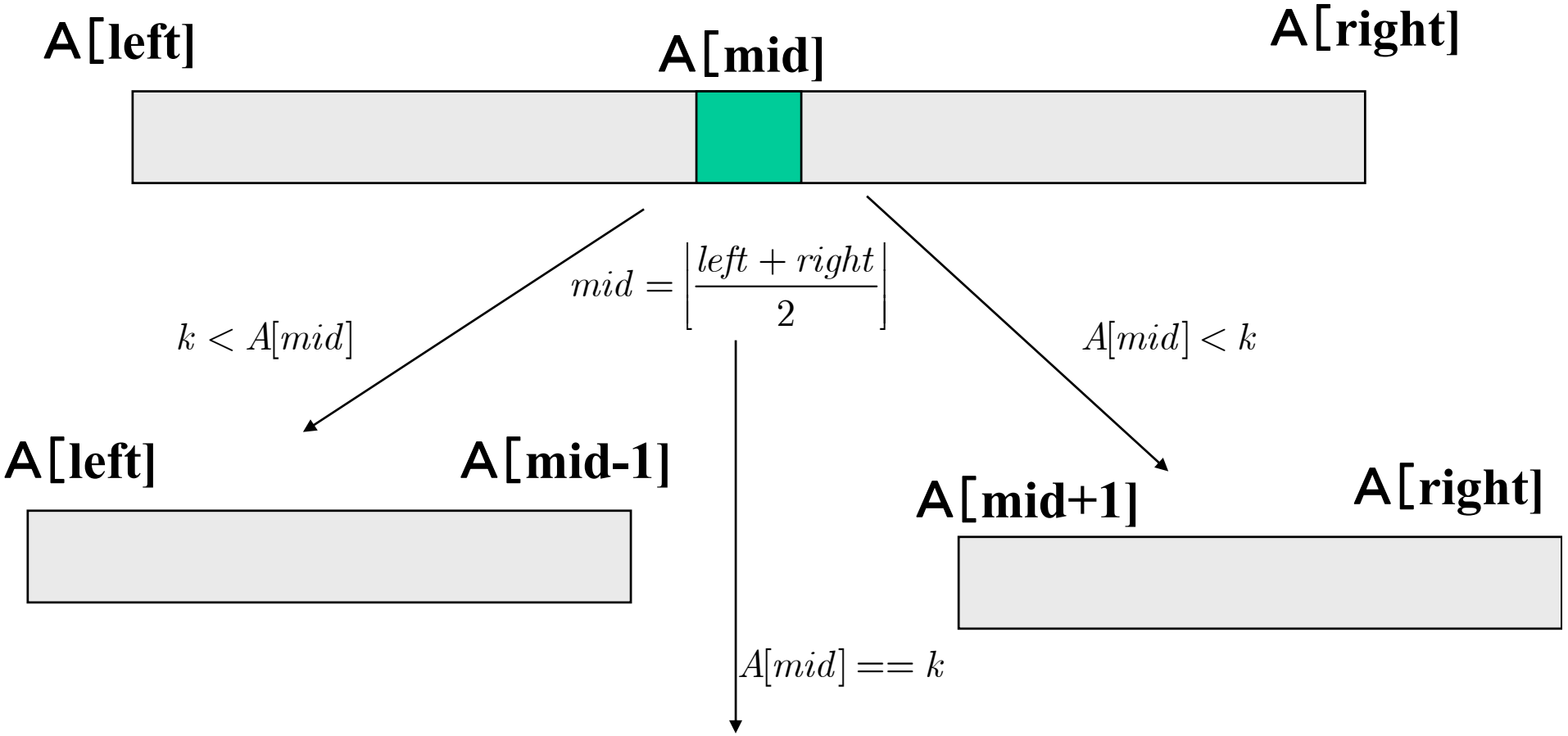






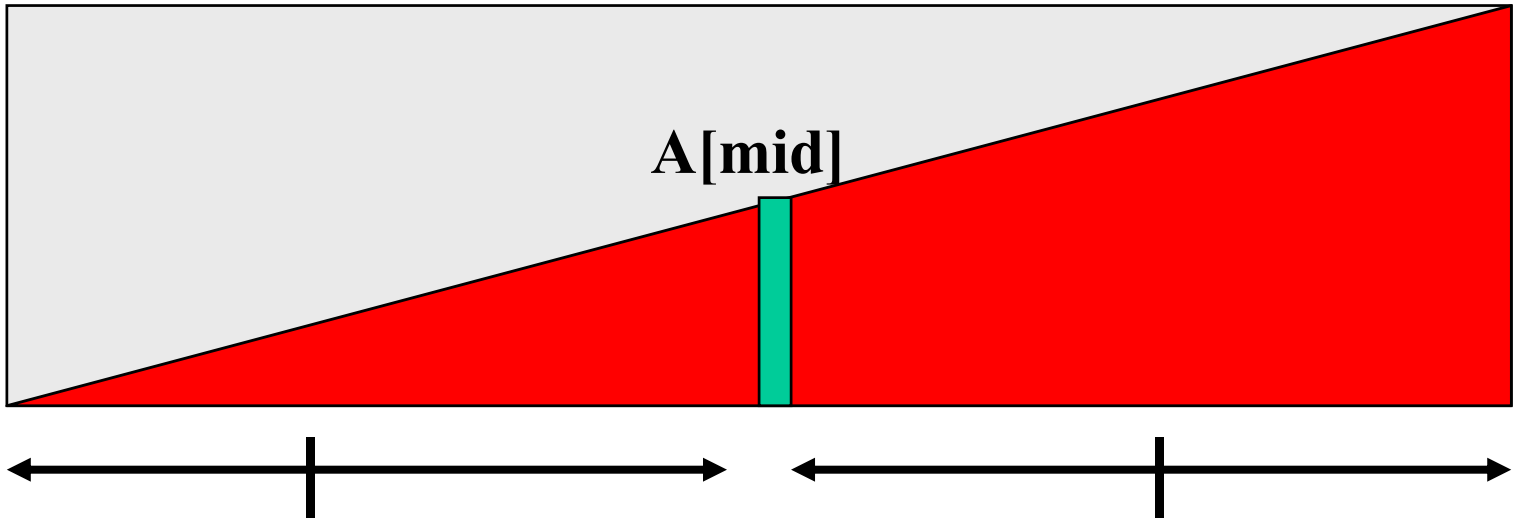


2分探索の原理



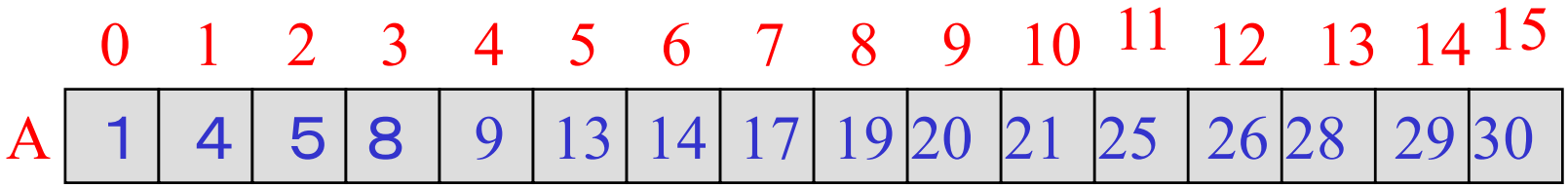
retun mid;

2分探索のイメージ



練習

次の配列に対して、各キーに対して、2分探索で調べられる要素の系列を示せ。



(1) key 5

(2) key 10

(3) key 20

(4) key 23

2分探索の注意

注意:

- アィディアは、結構シンプルであるが、
  実現には細心の注意が必要。

- 特に、再帰を用いて実現する場合には、
  その境界部分やサイズ減少について
  吟味する必要がある。

- 一見、正しく動作しているようにみえても、
  データによっては無限に再帰呼び出し
  を行うことがある。

2分探索の実現(繰り返し版)

```
/*　繰り返し2分探索
引数:キー
戻り値:キーを保持している配列の添え字
*/
1. int loop_b_search(double k){
2.       int  left=0,right=N-1,mid; /* カウンタ*/
3.       while(left<=right){
4.           mid=(left+right)/2;
5.           if(A[mid]==k){return mid;}/*発見*/
6.           if(k<A[mid]){right=mid-1;}/*小さい方*/
7.           if(A[mid]<k){left=mid+1;}/*大きい方*/
8.        }
9.        return -1;/*キーは存在しない。*/
10.}
```

命題LBS1 （loop_b_searchの正当性1）

A[left]～A[right]はソートしてあるとする。
このとき、次が成り立つ。

（1） A[mid]<kであるならば、
A[left]～A[mid]にはkは存在しない。

（2） K<A[mid]であるならば、
A[mid]～A[right]にはkは存在しない。

証明

（1）だけを証明する。（2）も同様に証明できる。

$$A[mid] < k \Rightarrow \forall i, left \leq i \leq mid, (A[i] \neq k)$$

を証明するために、より強い命題として次を証明する。

$$A[mid] < k \Rightarrow \forall i, left \leq i \leq mid, (A[i] < k)$$

まず、配列がソートされているので、
A[left]ーA[mid]の部分配列もソートされている。
すなわち、次式が成り立つ。

$$A[left] \leq A[left+1] \leq \cdots \leq A[mid-1] \leq A[mid]$$

この式と、 $A[mid] < k$ より、明らかに命題は成り立つ。

$$QED$$

命題LBS2 （loop_b_searchの正当性2）

A[left]～A[right]はソートしてあるとする。
このとき、次が成り立つ。

（1） A[mid]＜kであるとき、
もしkが存在するならばA[mid＋1]～A[right]
中に存在する。

（2）K＜A[mid]であるとき、
もしkが存在するならばA[left]～A[mid-1]
中に存在する。

証明

命題LBS1より明らかに成り立つ。

QED

命題LBS3 （loop_b_searchの停止性）

loop_b_searchは停止する。

証明

whileループの１回の繰り返しにより、
次の2つのいずれかが成り立つ。

（1）キーが発見されて、ステップ5により終了する。

（2）探索範囲が減少する。すなわち、
right-leftが1は減少する。

ここが重要。

特に、 $mid = \left\lfloor \frac{left + right}{2} \right\rfloor$ **であるが、**

$left \leq mid$ **としかいえないことに注意する必要がある。**

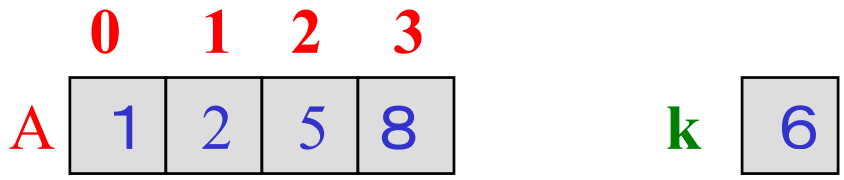
QED

間違った実装

```
/*  繰り返し2分探索*/
1. int loop_b_search(double k){
2.      int  left=0,right=N-1,mid; /* カウンタ*/
3.      while(left<=right){
4.          mid=(left+right)/2;
5.          if(A[mid]==k)return mid;/*発見*/
6.          if(k<A[mid]){right=mid;}/*小さい方*/
7.          if(A[mid]<k){left=mid;}/*大きい方*/
8.       }
9.       return -1;/*キーは存在しない。*/
10.}
```

この実装では、繰り返しによってもサイズが減少するとは限らない。

間違った実装が停止しない例



1回目 $left = 0 \quad right = 3 \quad mid = \left\lfloor \frac{0+3}{2} \right\rfloor = \lfloor 1.5 \rfloor = 1$

2回目 $left = 1 \quad right = 3 \quad mid = \left\lfloor \frac{1+3}{2} \right\rfloor = \lfloor 2 \rfloor = 2$

3回目 $left = 2 \quad right = 3 \quad mid = \left\lfloor \frac{2+3}{2} \right\rfloor = \lfloor 2.5 \rfloor = 2$

4回目 $left = 2 \quad right = 3 \quad mid = \left\lfloor \frac{2+3}{2} \right\rfloor = \lfloor 2.5 \rfloor = 2$

5回目 $left = 2 \quad right = 3 \quad mid = \left\lfloor \frac{2+3}{2} \right\rfloor = \lfloor 2.5 \rfloor = 2$

.....

2分探索の実現(再帰版)

```
/*  繰り返し2分探索*/
1. int rec_b_search(double k,int left,int right){
2.     int mid;
3.     if(left>right){/*基礎*/
4.         return -1;/* 未発見*/
5.     }else{        /*帰納*/
6.         mid=(left+right)/2;
7.         if(A[mid]==k){/*発見*/
8.             return mid;
9.         }else if(k<A[mid]){/*小さい方*/
10.            return rec_b_search(k,left,mid-1);
11.        }else if(A[mid]<k){/*大きい方*/
12.            return rec_b_search(k,mid+1,right);
13.        }
14.    }
15.}
```

rec_b_searchの正当性および停止性は、
loop_b_searchと同様に示すことができる。

2分探索の最悪計算量

探索される部分配列の大きさ $K \equiv right - left$ に注目する。
i回の繰り返しの後の探索される部分配列の大きさを K_i と表す。
このとき次の漸化式が成り立つ。

$$\begin{cases} K_0 = n & i = 0 \\ K_i \leq \dfrac{K_{i-1}}{2} & i > 0 \end{cases}$$

これより、次式が成り立つ。

$$K_i \leq \frac{n}{2^i}$$

最悪の場合でも、部分配列の大きさが1以上でしか繰り返すことができない。

$$1 \leq K_i = right - left < \frac{n}{2^i}$$

これより、次のように計算できる。

$$2^i \leq n$$

$$\therefore i \leq \log_2 n$$

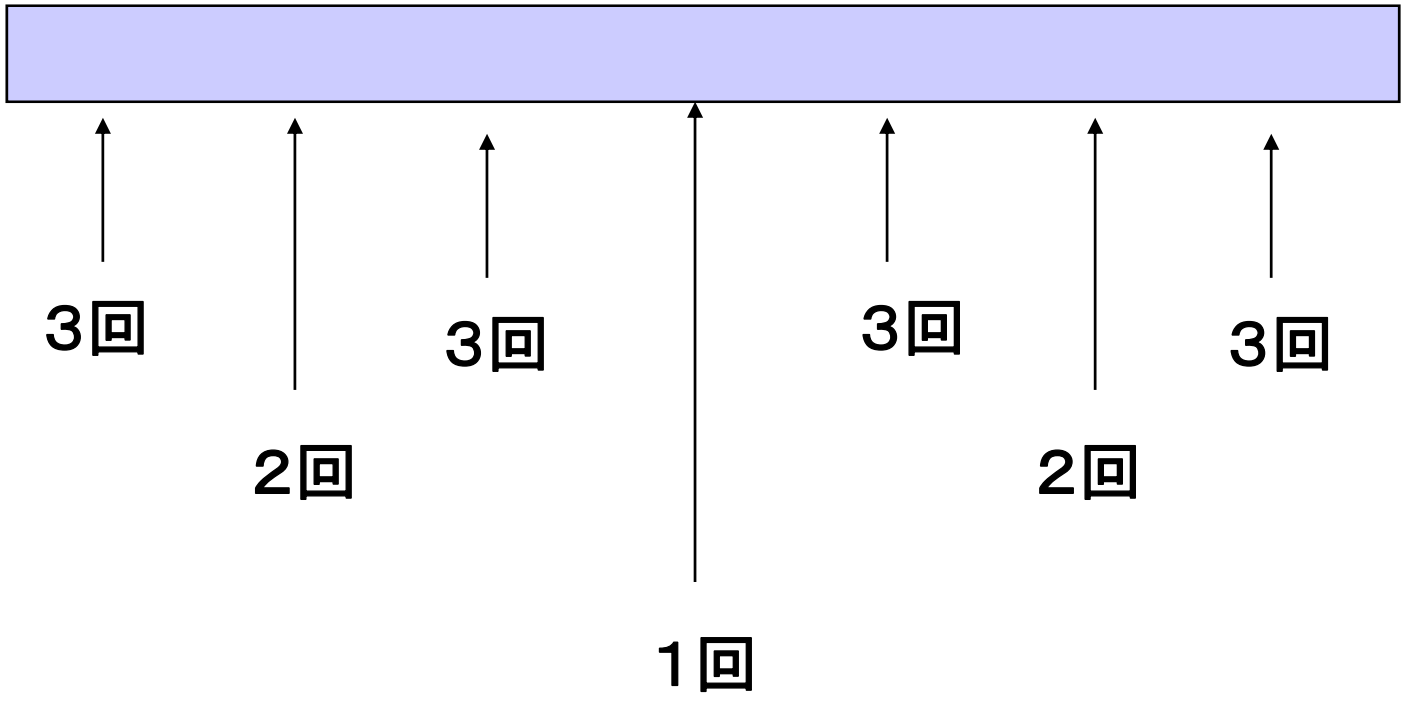
よって、最悪時間計算量は、

$$O(\log_2 n)$$

のアルゴリズムである。

2分探索の平均計算量

平均時間計算量を求める。
要素数を $n < 2^k$ とし、すべて等確率でキーが保持されていると仮定する。 このとき、最大反復回数は、$k = \log_2 n$ である。要素位置により下図のように計算時間を割り振ることができる。



よって、平均反復回数$E(n)$ は次式を満たす。

$$E(n) = \frac{1}{n}\left\{1 + 2\bullet 2 + 3\bullet 4 + 4\bullet 8 \cdots + k\bullet 2^{k-1}\right\}$$

$$= \frac{1}{n}\sum_{j=1}^{k} j\bullet 2^{j-1}$$

$$2E(n) = \frac{1}{n}\left\{\qquad 1\bullet 2 + 2\bullet 4 + 3\bullet 8 + 4\bullet 8 + \cdots + k\bullet 2^{k}\right\}$$

$$-) \qquad E(n) = \frac{1}{n}\left\{1\bullet 1 + 2\bullet 2 + 3\bullet 4 + 4\bullet 8 + \cdots + k\bullet 2^{k-1}\right\}$$

$$E(n) = \frac{1}{n}\left\{k\bullet 2^{k} - (1 + 2 + 4 + \cdots + 2^{k-1})\right\}$$

$$E(n) = \frac{1}{n}\{n \log n - (n-1)\}$$

$$\therefore E(n) = \log n - 1 + \frac{1}{n}$$

よって、平均時間計算量も、

$$O(\log_2 n)$$

のアルゴリズムである。

5-3. ハッシュ

線形探索と2分探索の問題点

- 線形探索
  - 多大な計算時間が必要。
  - （データの順序に制限はない。）
- ２分探索
  - （検索時間は高速。）
  - 事前にソートが必要。

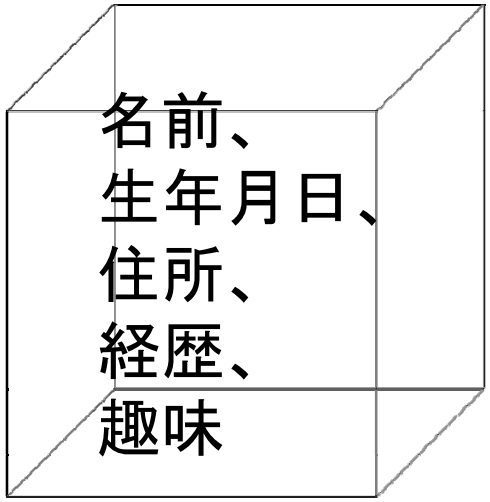
データの保存時、とデータ検索時の両方に
効率的な手法が望まれる。→ハッシュ法

ハッシュとは

- 整数への写像を利用して、高速な検索を可能とする技法。
- 探索データに割り当てられる整数値を配列の添え字に利用する。
- ハッシュを用いることにより、ほとんど定数時間($O(1)$ 時間)の高速な探索が可能となる。

ハッシュのイメージ

大きいデータ



x

写像

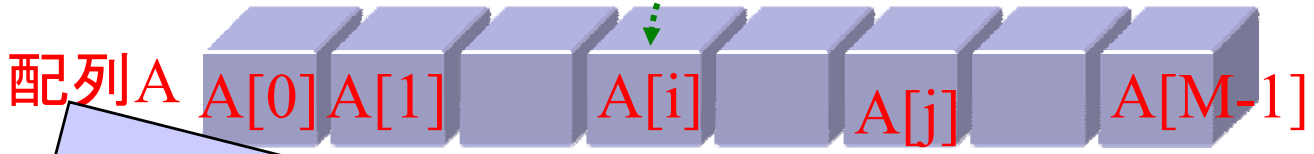
ハッシュ関数

h

（範囲の制限された）
整数

$i = h(x)$

配列の添え字として利用。



ハッシュ表（ハッシュテーブル）といいます。

具体的なハッシュ関数

ここでは、名前データから具体的なハッシュ関数を構成する。
簡単のため、名前はアルファベットの小文字の8文字からなるものだけを考える。

入力: $\boldsymbol{x} = x_1 x_2 \cdots x_8$

ただし、 $1 \leq i \leq 8$ に対して、 $x_i \in \{a, b, \cdots, z\}$

（入力例：suzuki,sato,kusakari,・・・）

配列の大きさ

ハッシュ値:

$$h(\boldsymbol{x}) \in \{0, 1, 2, \cdots, M-1\}$$

（ハッシュ値の例：3，7，11、・・・）

アスキーコード

アスキーコードは、以下に示すように、アルファベットへの整数値の割り当てである。

$$a \leftrightarrow (61)_{16} = (01100001)_2 = (97)_{10}$$

$$b \leftrightarrow (62)_{16} = (01100010)_2 = (98)_{10}$$

$$\vdots$$

$$z \leftrightarrow (7A)_{16} = (01111010)_2 = (122)_{10}$$

これを利用する。

このコードを、次のように記述する。

$$code(x_i)$$

（例： $code(a) = 97$ ）

ハッシュ関数の構成例1

$$h(\boldsymbol{x}) = \sum_{i=1}^{8} code(x_i) \pmod{M}$$

この余りを求める演算により、
ハッシュ値がつねに、
$h(\boldsymbol{x}) \in \{0,1,2,\cdots,M-1\}$
となることが保証される。
→配列の添え字として利用可能。

名前とハッシュ関数の構成例

ここでは、名前データから具体的なハッシュ関数を構成してみる。簡単のため、名前はアルファベットの小文字の８文字からなるものだけを考える。

入力: $\boldsymbol{x} = x_1 x_2 \cdots x_8$

ただし、 $1 \leq i \leq 8$ に対して、 $x_i \in \{a, b, \cdots, z\}$

（入力例：suzuki,sato,kusakari,・・・）

ハッシュ値:

$$h(\boldsymbol{x}) \in \{0, 1, 2, \cdots, M-1\}$$

（ハッシュ値の例：３，７，１１、・・・）

ハッシュ値の計算例

ここでは、M=8として具体的にハッシュ値を計算してみる。

$$h(abe) = \big(code(a) + code(b) + code(e)\big) \pmod{10}$$

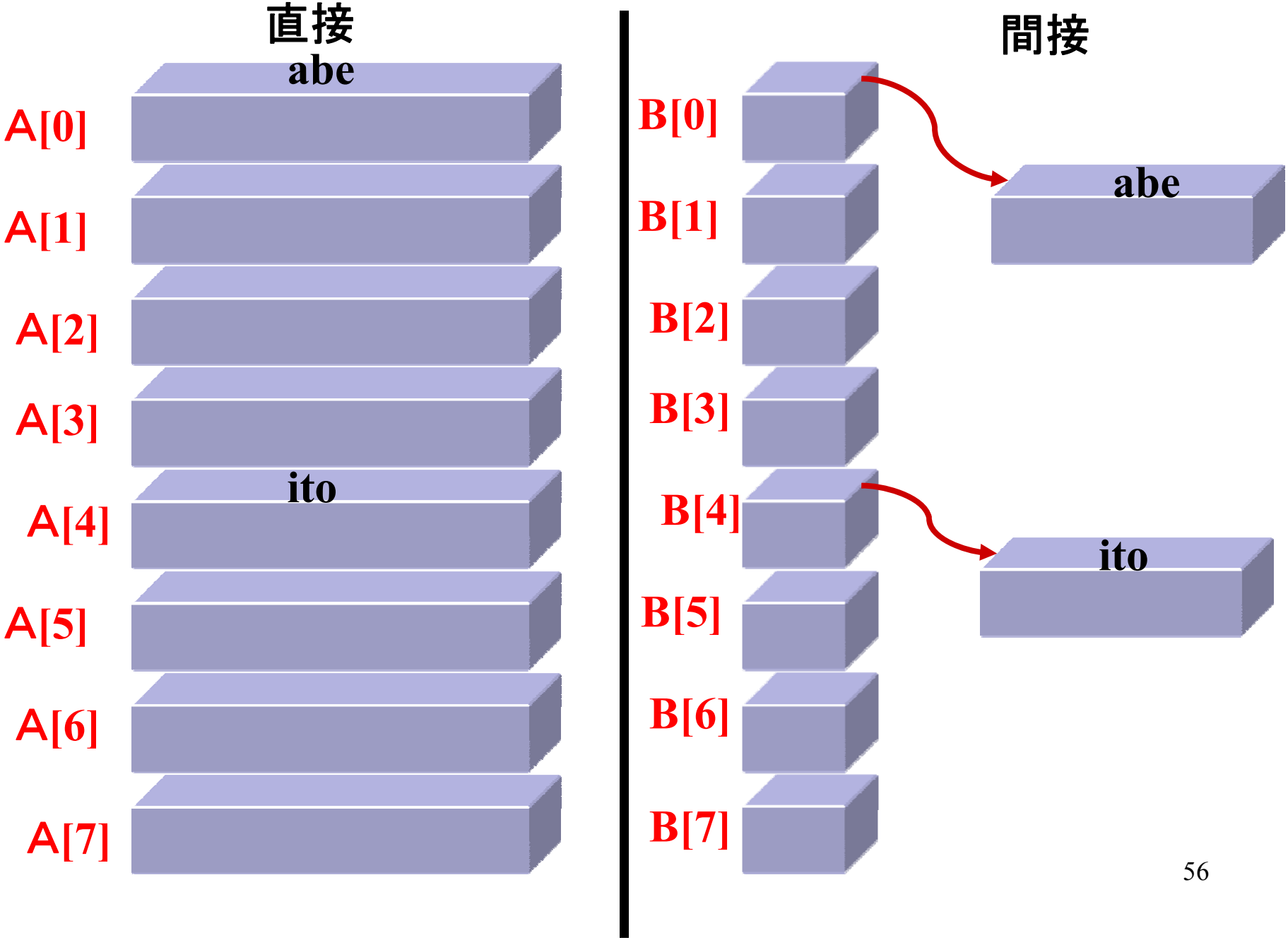
$$= 97 + 98 + 101 \pmod{8}$$

$$= 296 \pmod{8}$$

$$= 0$$

$$h(ito) = 105 + 116 + 111 \pmod{8} = 4$$

このハッシュ値をもとに配列に保存する。



練習

先ほどのハッシュ関数を用いて自分の苗字に対するハッシュ値と、
名前に対するハッシュ値を求めよ。

ハッシュ関数の定義域と値域

ここでは、ハッシュ関数の定義域と値域を考察する。

先ほどの、ハッシュ関数では、
ハッシュ関数の定義域の大きさは、 26^8 である。
この定義域を名前空間と呼ぶこともある。

$$S = \{a, b, \cdots, z\}$$

とすると、名前空間は、の8個の直積で表される。
すなわち、

$$S \times S \times \cdots \times S = S^8$$

が定義域になる。

次に値域は、

$$\{0, 1, 2, \cdots, M-1\}$$

であるが、これを $\mathbb{Z}_M$ と書く。

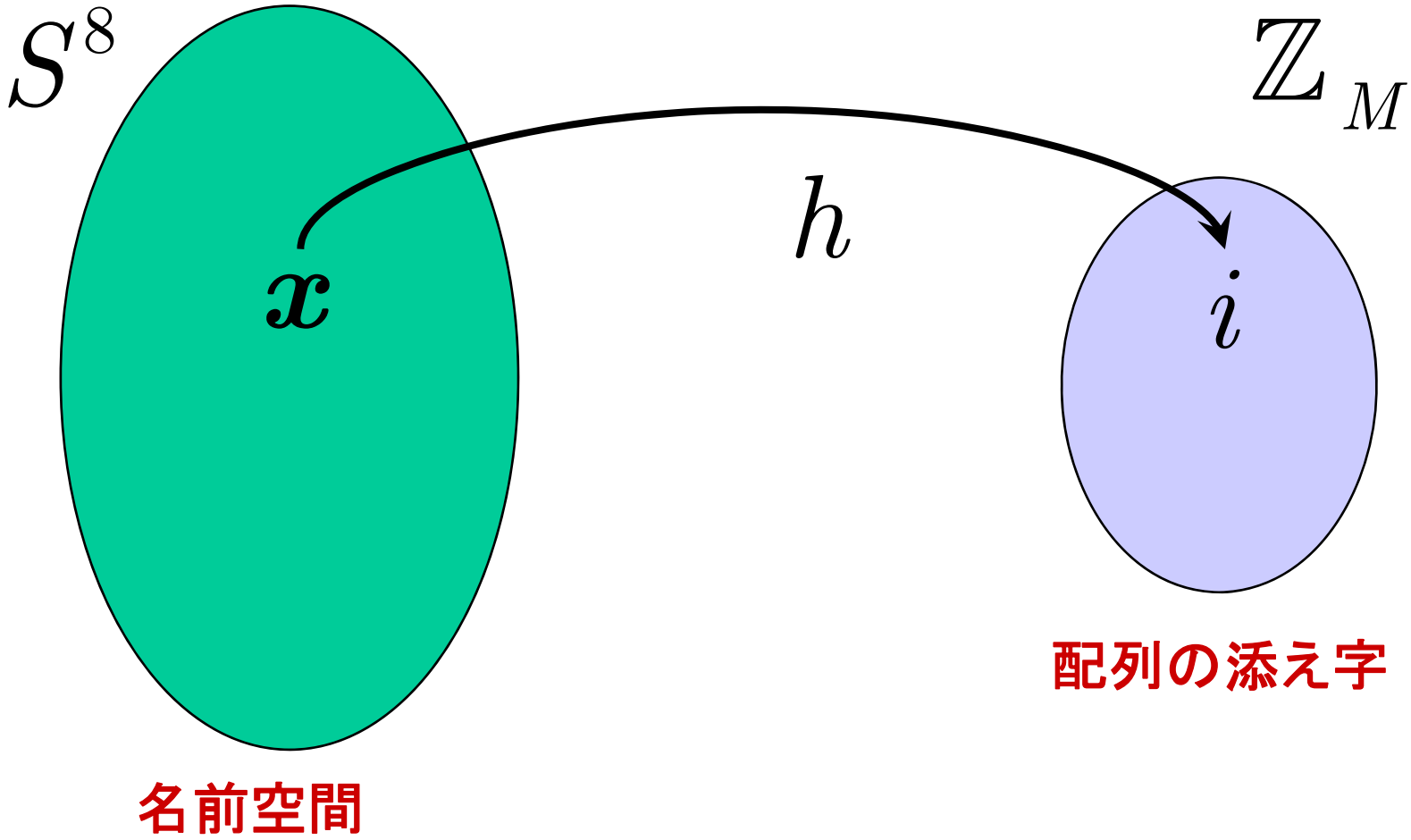
これらの記号を用いると、ハッシュ関数は次のように記述される。

$$h : S^8 \longrightarrow \mathbb{Z}_M$$

$$\boldsymbol{x} \longmapsto h(\boldsymbol{x})$$

$$\boldsymbol{x} \in S^8, h(\boldsymbol{x}) \in \mathbb{Z}_M$$

関数のイメージ



ハッシュ関数への要求

- 探索には、ハッシュ値にしたがって、検索される。
- ハッシュ値からもとのデータ（名前）を得るには、逆写像が必要。
- 全単射が望ましいが、名前空間が膨大なため実現困難。（すくなくとも、単射にしたい。）

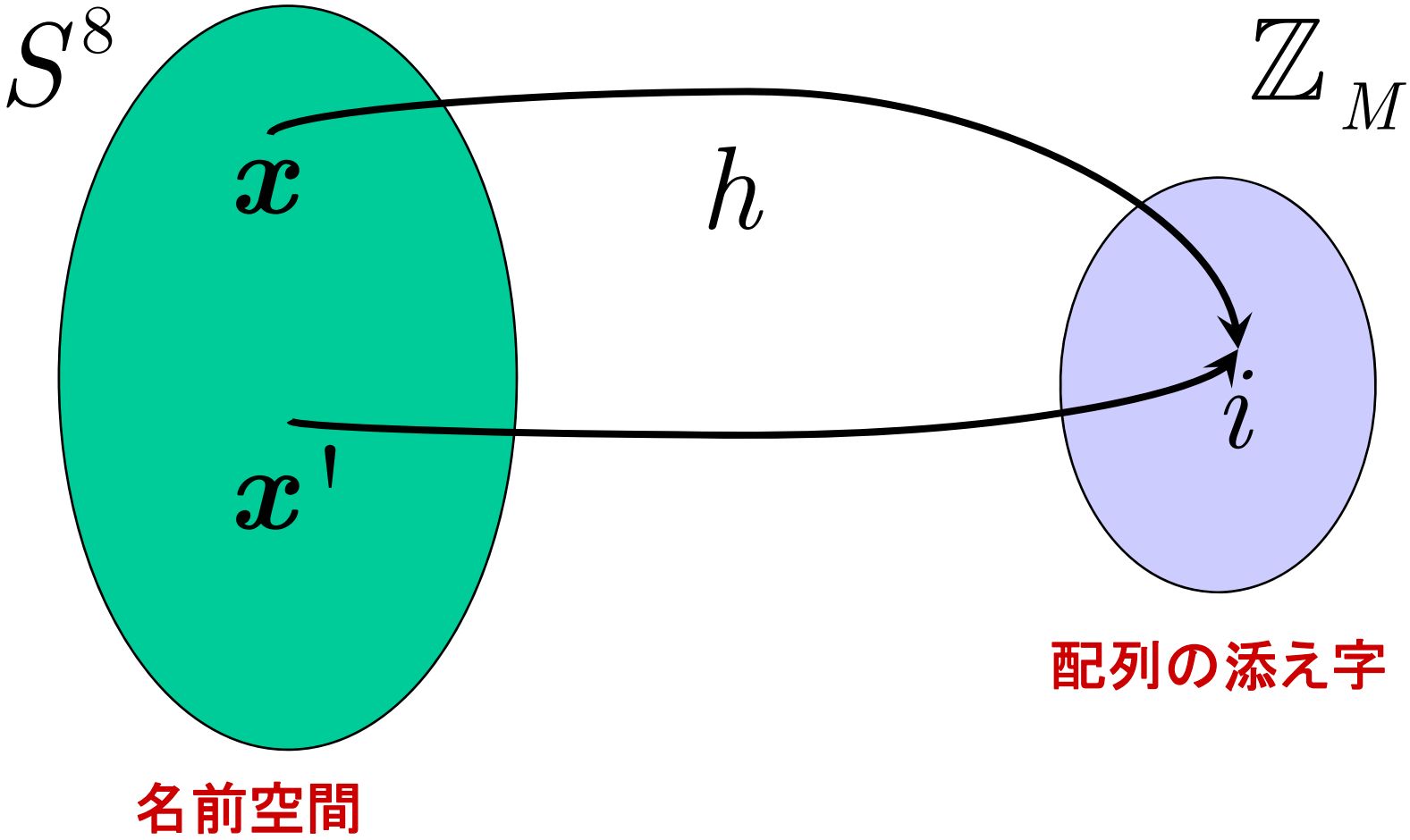
$$\left|S^8\right| = 26^8 \gg \mathbb{Z}_{10} = 10$$

衝突

｜定義域｜＞｜値域｜のときには、理論的には単射は存在しない。しかし、ハッシュが適用される場面の多くでは、
｜定義域｜＞＞｜値域｜
である。つまり、ハッシュ関数の多くは単射にならない。

値域の１つの要素に対して、複数の定義域の要素が対応する。このことを、衝突という。衝突しているデータを同義語（シノニム）ということもある。

衝突のイメージ1



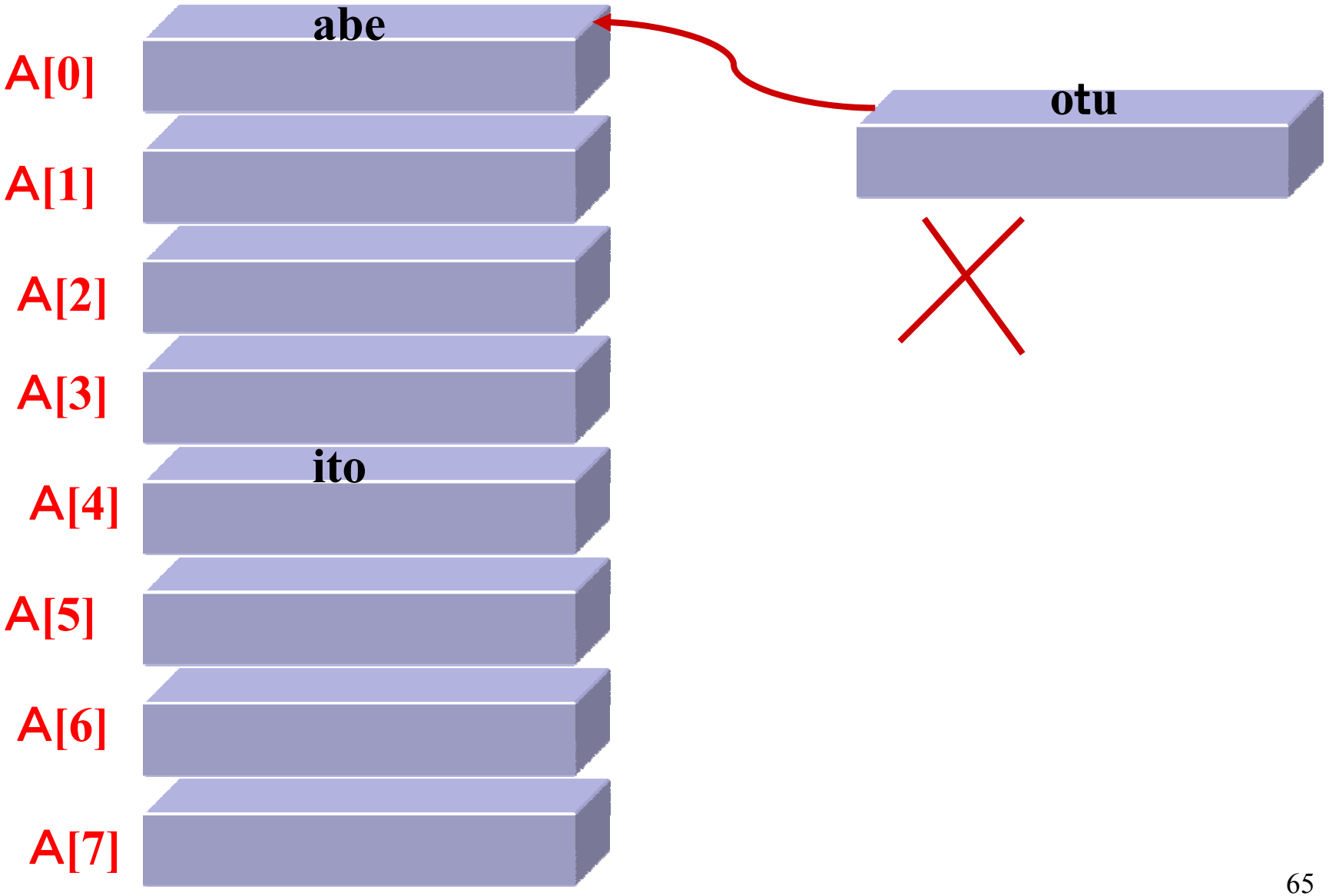
衝突例

ここでは、M=8として具体的にハッシュ値を計算してみる。

$$h(abe) = 97 + 98 + 101 \pmod 8 = 0$$

$$h(otu) = 111 + 116 + 117 \pmod 8 = 0$$

衝突のイメージ2



衝突への対処

衝突の関数に関係した、
ハッシュ関数の系列で対処する。

**衝突の回数が　k　回のとき、ハッシュ関数に、
次を用いる。**

$$h_k(\boldsymbol{x}) = h(\boldsymbol{x}) + k \pmod{M}$$

$$= \sum_{i=1}^{8} code(x_i) + k \pmod{M}$$

$$h_0(\boldsymbol{x}) = \sum_{i=1}^{8} code(x_i) + 0 \pmod{M} = h(\boldsymbol{x})$$

$$h_1(\boldsymbol{x}) = h(\boldsymbol{x}) + 1 \pmod{M}$$

$$h_2(\boldsymbol{x}) = h(\boldsymbol{x}) + 2 \pmod{M}$$

$$\vdots$$

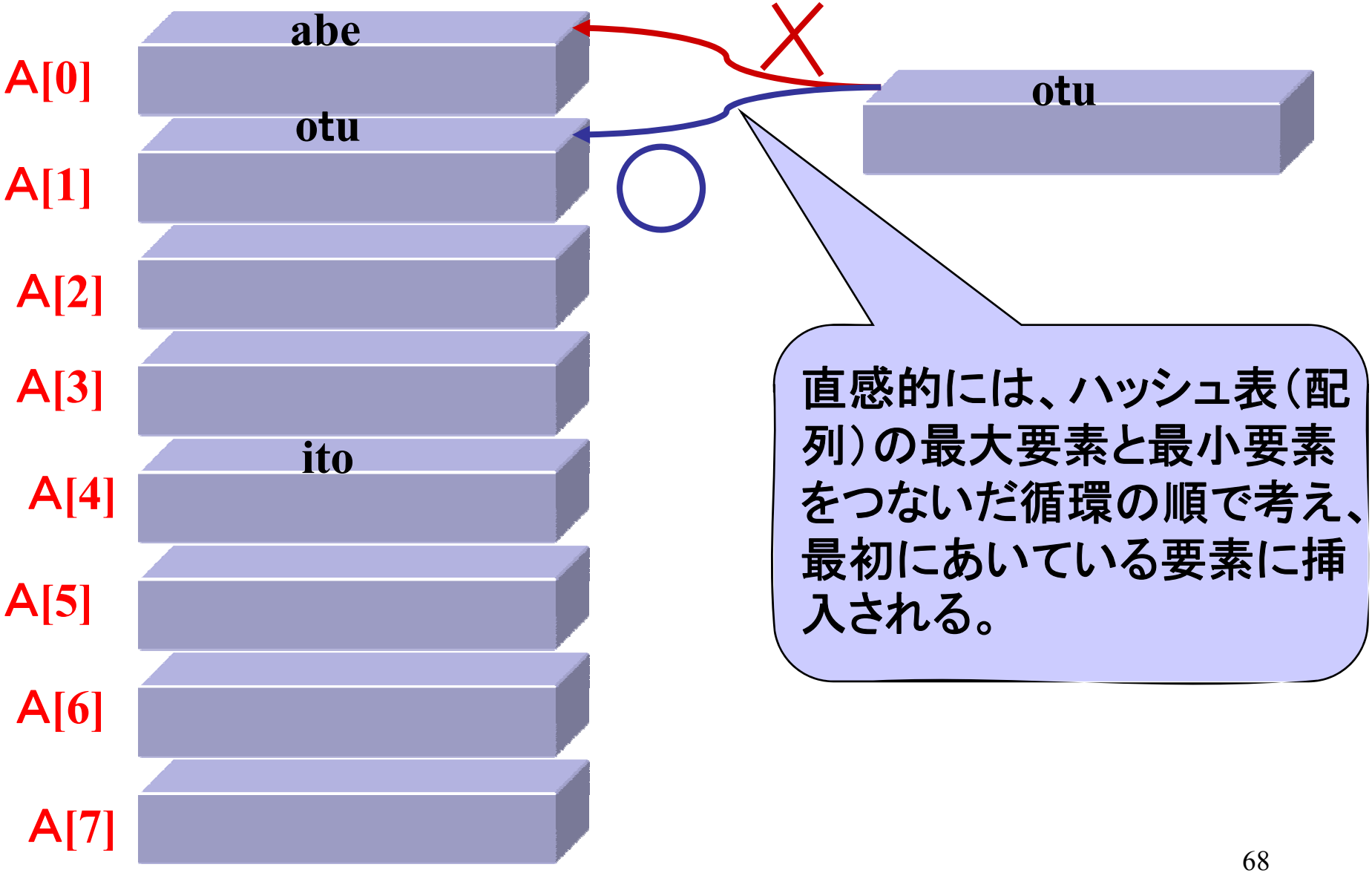
このハッシュ関数を用いると、abe-> okuの順にデータが挿入された場合、次のように割り当てられる。

$$h_0(abe) = 0$$

$$h_0(otu) = 0 \Rightarrow$$

$$h_1(otu) = 0 + 1 \pmod{8} = 1$$

衝突の対処


abe
A[0]
otu
A[1]
A[2]
A[3]
ito
A[4]
A[5]
A[6]
A[7]
otu
直感的には、ハッシュ表（配列）の最大要素と最小要素をつないだ循環の順で考え、最初にあいている要素に挿入される。

ハッシュ表への検索

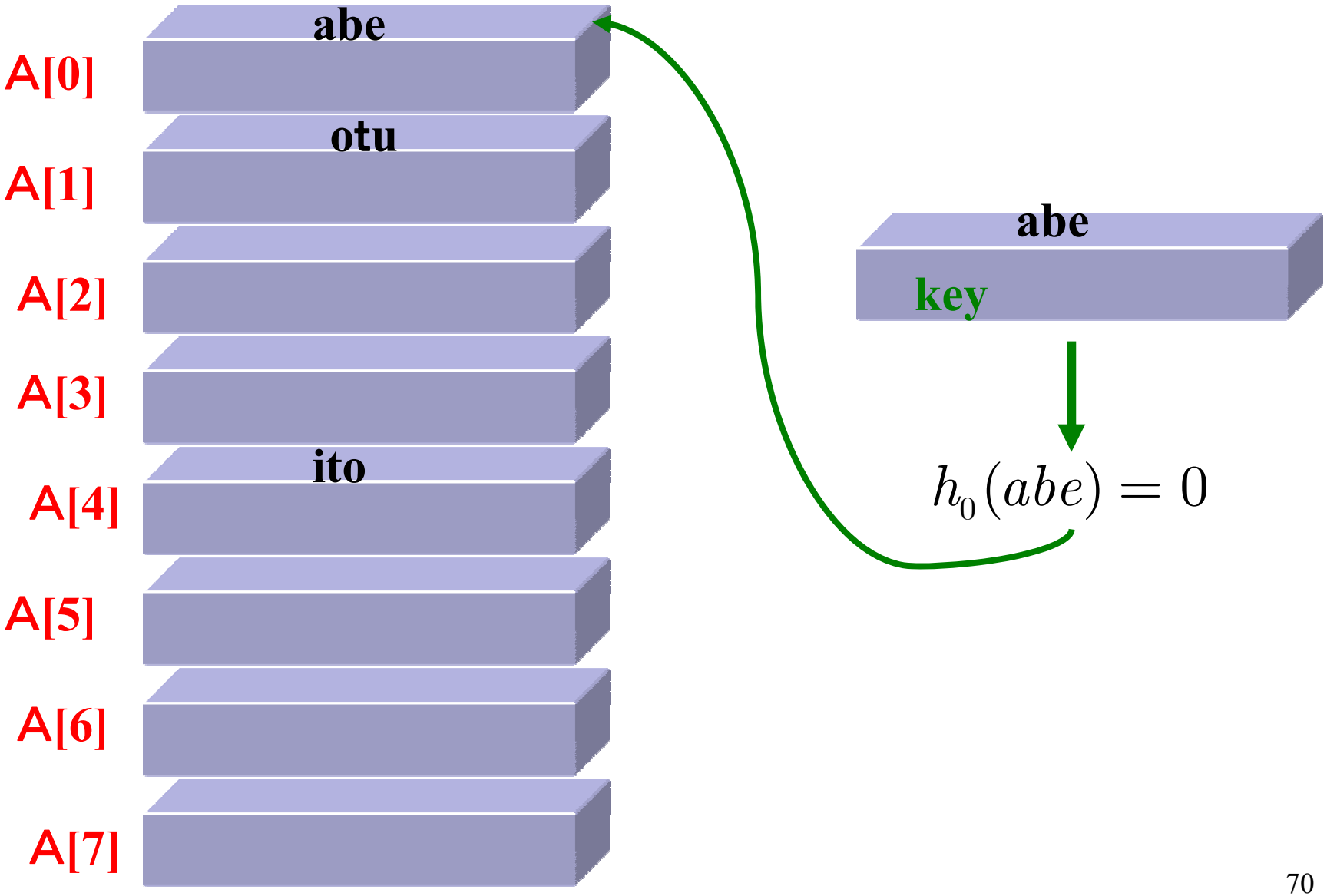
ハッシュ表への検索は、キーに対して、ハッシュ表作成時と同じハッシュ関数を用いることで実現される。

したがって、キーを、 $\boldsymbol{y}$ とすると、次の関数によって、ハッシュ値を計算して、ハッシュ表を調べる。

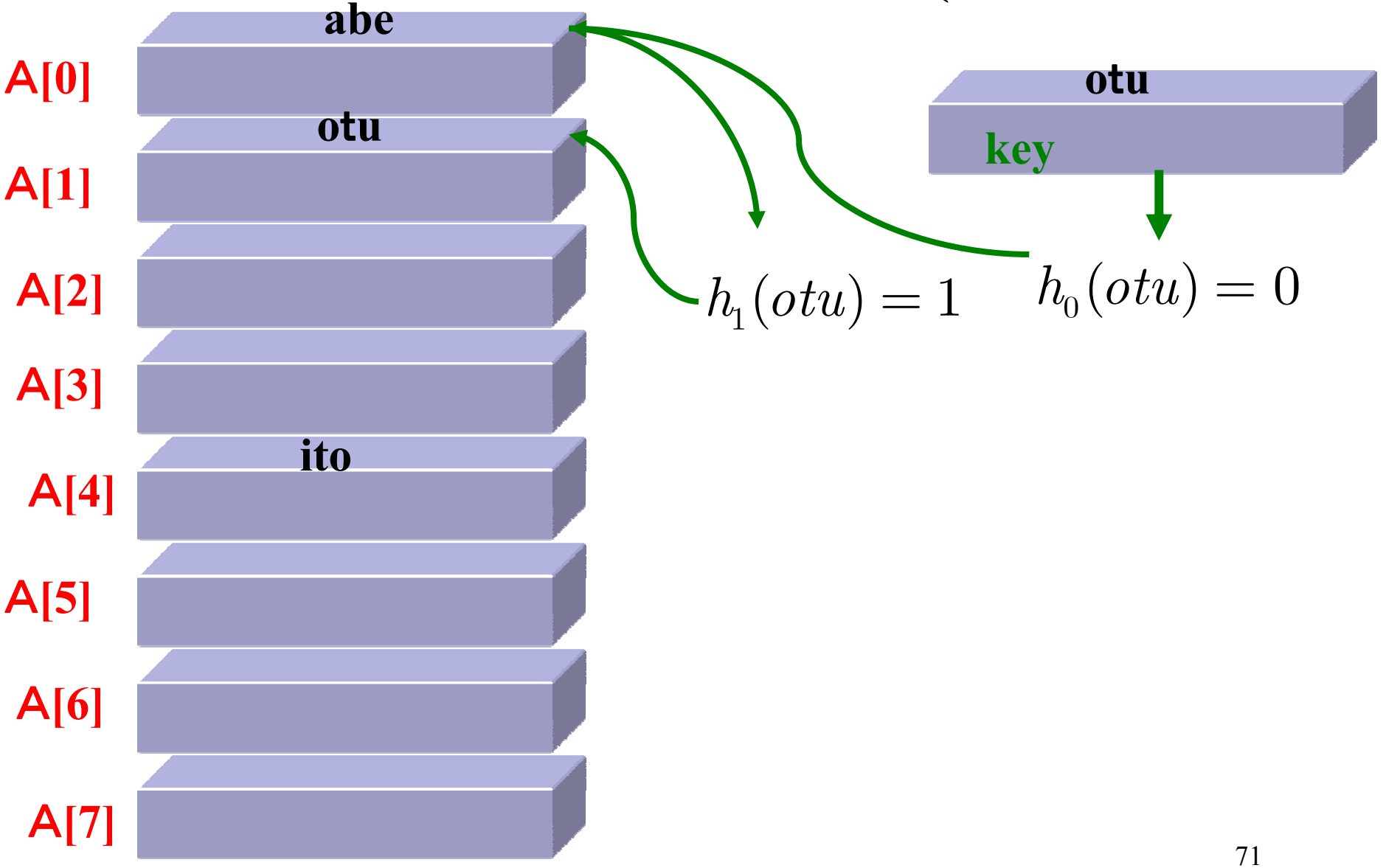
$$h_k(\boldsymbol{y}) = h(\boldsymbol{y}) + k \pmod{M}$$

$$= \sum_{i=1}^{8} code(y_i) + k \pmod{M}$$

ハッシュ表からの検索



ハッシュ表からの検索(衝突時)



ハッシュテーブルへのデータ挿入 (衝突が無い場合)

```
/* ハッシュへの挿入 */
1. void input()
2. {
3.      int h; /*ハッシュ値*/
4.     for(i=0;i<n;i++)
5.         {
6.            h=hash(x[i]);
7.             A[h]=x[i];
8.           }
9.     }
10.    return;
11.}
```

ハッシュ表からの検索（衝突が無い場合）

```
/* ハッシュからの検索 */
1.  int search(double key)
2.  {
3.      int h; /*ハッシュ値*/
4.      h=hash(key);
5.       if(A[h]が空) {
6.           return -1;/*データ無し*/
7.       } else if(A[h]==key) {/*発見*/
8.           return h;
9.       }
10. }
```

ハッシュテーブルへのデータ挿入 (衝突がある場合)

```
/* ハッシュへの挿入 */
1.  void input()
2.  {
3.       int h=0; /*ハッシュ値*/
4.      for(i=0;i<n;i++){
5.          h=hash(x[i]);
6.          while(A[h]が空でない){/*衝突の処理*/
7.              h=(h+1)%M;
8.          }
9.          A[h]=X[j];
10.     }
11.     return;
12.}
```

ハッシュ表からの検索（衝突がある場合）

```
/* ハッシュからの検索 */
1.  int search(double key)
2.  {
3.       int h; /*ハッシュ値*/
4.       h=hash(key);
5.      while(1){/*ハッシュ値による繰り返し検索*/
6.          if(A[h]が空) {
7.              return -1;/*データ無し*/
8.           }else if(A[h]!=key) {/*衝突*/
9.              h=(h+1)%M; /*ハッシュ値の更新*/
10.          }else if(A[h]==key) {/*発見*/
11.             return h;
12.          }
13.    }
14.}
```

ハッシュ法を用いた計算量（衝突が定数回の場合）

- ハッシュ法の計算時間はハッシュ関数の計算に必要な計算量に依存するが、通常、ハッシュ関数の計算は、入力サイズのnに依存していないことが多い。
- したがって、次のように解析される。
  - ハッシュ表の作成は、線形時間（$O(n)$ 時間）
  - ハッシュ表からのキーの検索は、定数時間（ $O(1)$時間）

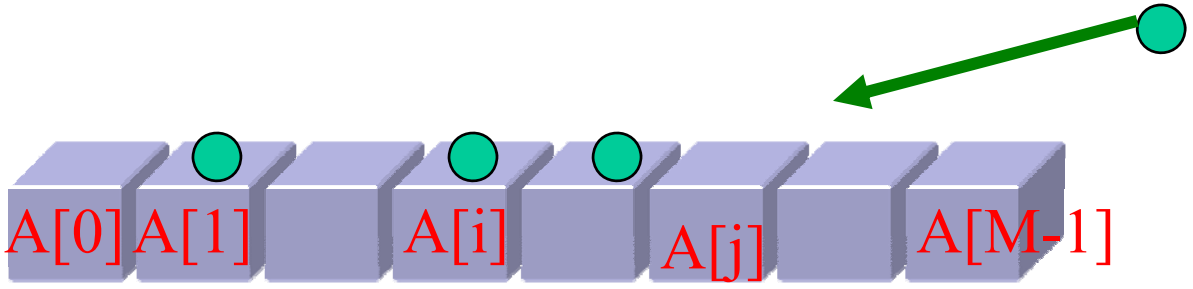
衝突がある場合の
平均計算量解析

衝突がある場合は少し複雑な解析が必要である。

挿入の評価：

ここでは、まず、サイズMのハッシュ表に、N個のデータを挿入する計算量を評価する。

今、k番目のデータが既に挿入されているときに、k＋1番目のデータを挿入することを考える。



k個のデータが存在

このとき、$h(x) = h_0(x)$　により求められる最初のセルがふさがっている確率は、

$$\frac{k}{M}$$

である。このときは、ハッシュ関数 $h_1(x)$ により2つ目のハッシュ値が求められる。このハッシュ値を添え字とするセルがふさがっている確率は、Ｍ－１個中の、ｋ－１個がふさがっている確率であるので、

$$\frac{k-1}{M-1}$$

である。よって、$h_0(x), h_1(x), \cdots, h_{i-1}(x)$　までが全てふさがっている確率は、次式で表される。

$$\frac{k(k-1)\cdots(k-i+1)}{M(M-1)\cdots(M-i+1)} \simeq \left(\frac{k}{M}\right)^i$$

これは、空きセルを見つけるための失敗の回数を表している。これに、空きセルの発見（成功）の分の１回を考慮することで、挿入位置を求める際に調べるセルの期待値が次式で表される。

$$1+\sum_{i=1}^{M-1}\frac{k(k-1)\cdots(k-i+1)}{M(M-1)\cdots(M-i+1)}\frac{k(k-1)\cdots(k-i+1)}{M(M-1)\cdots(M-i+1)}$$

$$\simeq 1+\sum_{i=1}^{\infty}\left(\frac{k}{M}\right)^{i}$$

$$=1+\frac{\frac{k}{M}}{1-\frac{k}{M}} \qquad (\because k<M)$$

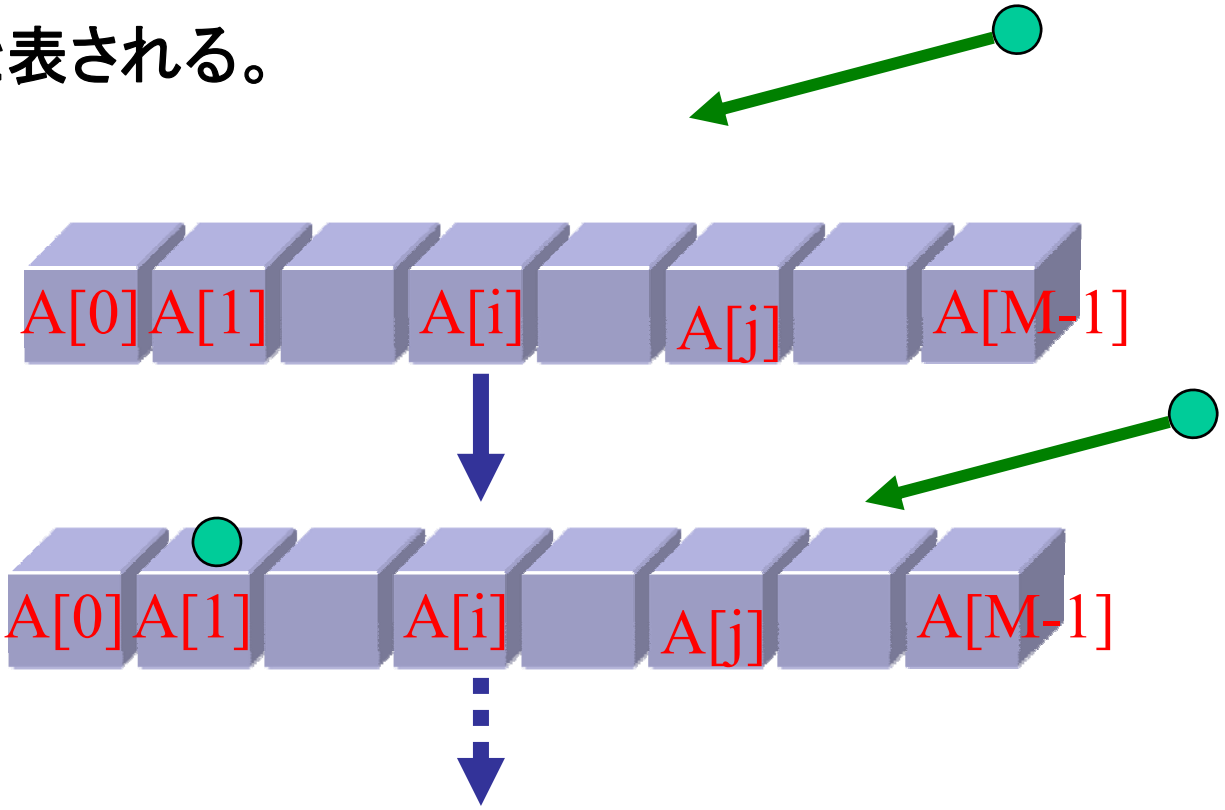
$$=\frac{(M-k)+k}{M-k}$$

$$=\frac{M}{M-k}$$

これは、1回の挿入の際に調べるセルの期待値である。したがって、ハッシュ表にN個のデータを挿入する際の総計算量は、

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

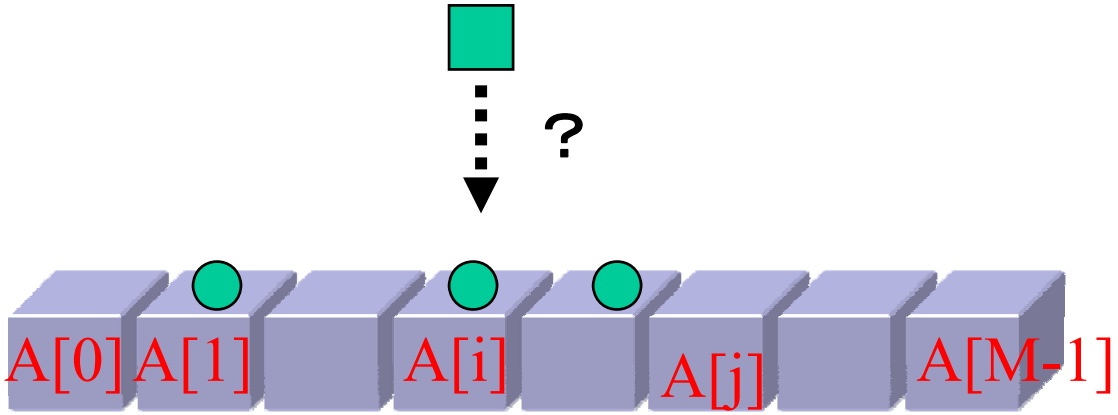
と表される。



したがって、一回あたりの平均計算量は、次式で表される。

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

ここで、$\alpha \equiv \frac{N}{M}$ はハッシュ表におけるデータの使用率である。



検索の評価：

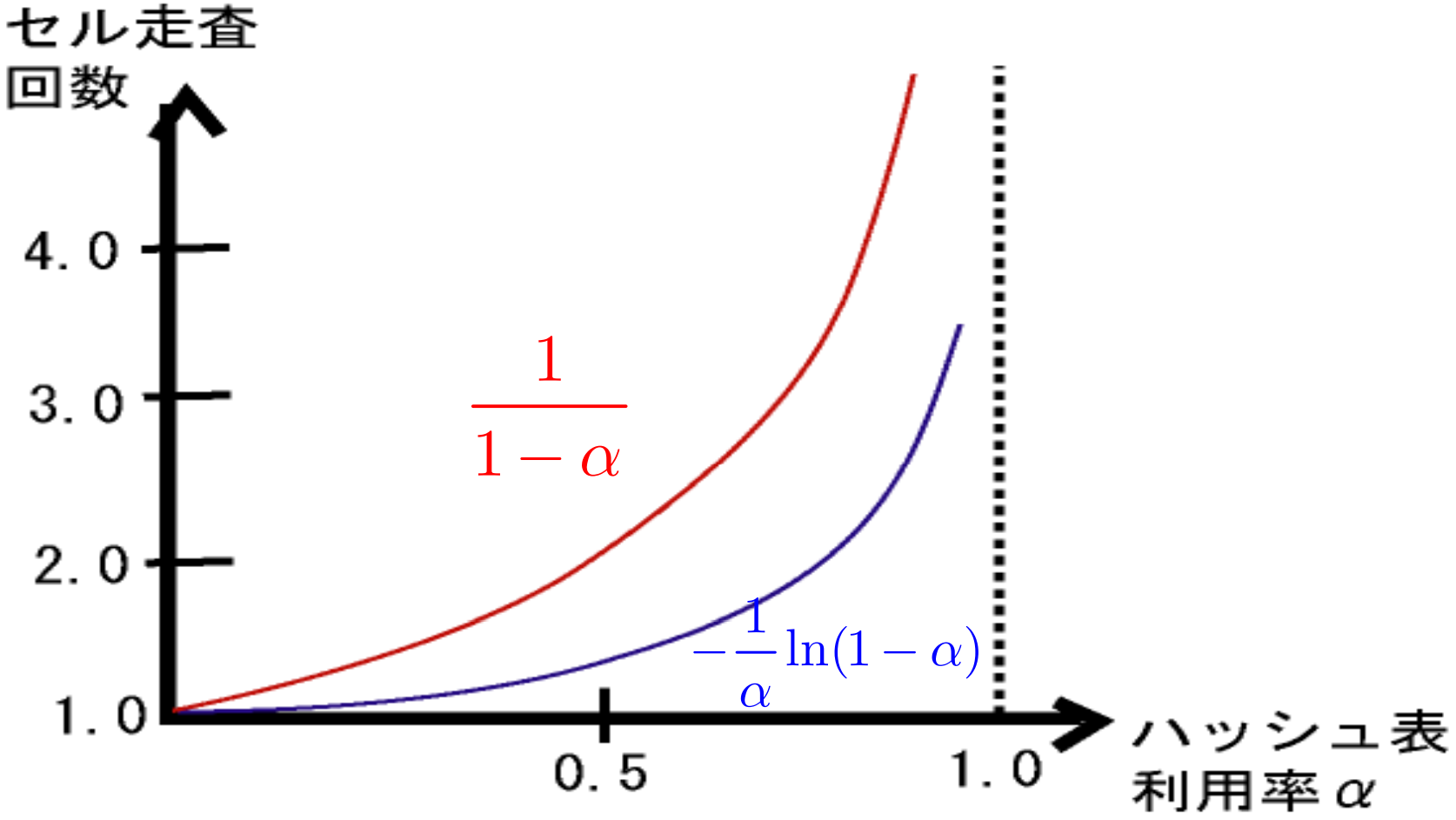
データがハッシュ表に存在する場合は、挿入時の1回当たりの平均計算量と同じである。

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

データがハッシュ表に存在しない場合は、N個のデータが存在しているときの、挿入位置をもとめる平均計算量と同じであり、次式で表される。

$$1+\sum_{i=1}^{\infty}\left(\frac{N}{M}\right)^{i} \simeq \frac{M}{M-N} = \frac{1}{1-\alpha}$$

内部ハッシュ関数の計算量の概形



ハッシュ法のまとめ

- 衝突が少ない場合には、極めて高速に、データの保存、検索を行える。
  - ハッシュ表の作成は、線形時間（$O(n)$時間）
  - ハッシュ表からのキーの検索は、定数時間（ $O(1)$ 時間）

- 衝突への対処を考慮する必要がある。
  - 今回の説明で用いたように、すべてのデータを配列内部に保持する方法を内部ハッシュ（クローズドハッシュ）という。
  - 間接参照を利用して、衝突を処理する方法も考えられる。（この方法を外部ハッシュ法（オープンハッシュ）という

- 衝突が生じる場合：
  - ハッシュ表の大きさMとしては、データ数の２倍以上にしておくと検索時間は定数時間とみなせることが多い。
  - データ数がハッシュ表の大きさに近いと、性能は急激に劣化する。特に、Ｍ＜Ｎとすると、アルゴリズムの停止性に問題が生じる。

他のハッシュ関数

- キーのデータの２乗和をバケットで割った余り。

$$h(\boldsymbol{x}) = \sum_{i=0}^{l} {x_i}^2 \pmod{M}$$

ここで、lは名前の長さ。

- キーの２乗の中央 $\log M$ ビット部分。

$$h(\boldsymbol{x}) = \left\lfloor \frac{\boldsymbol{x}^2}{C} \right\rfloor \pmod{M}$$

$$(MC^2 = K^2)$$

ここで、Kは名前空間の上限値。