基本データ構造
と
ハッシュ表

2020/5/15

B4

佐藤光

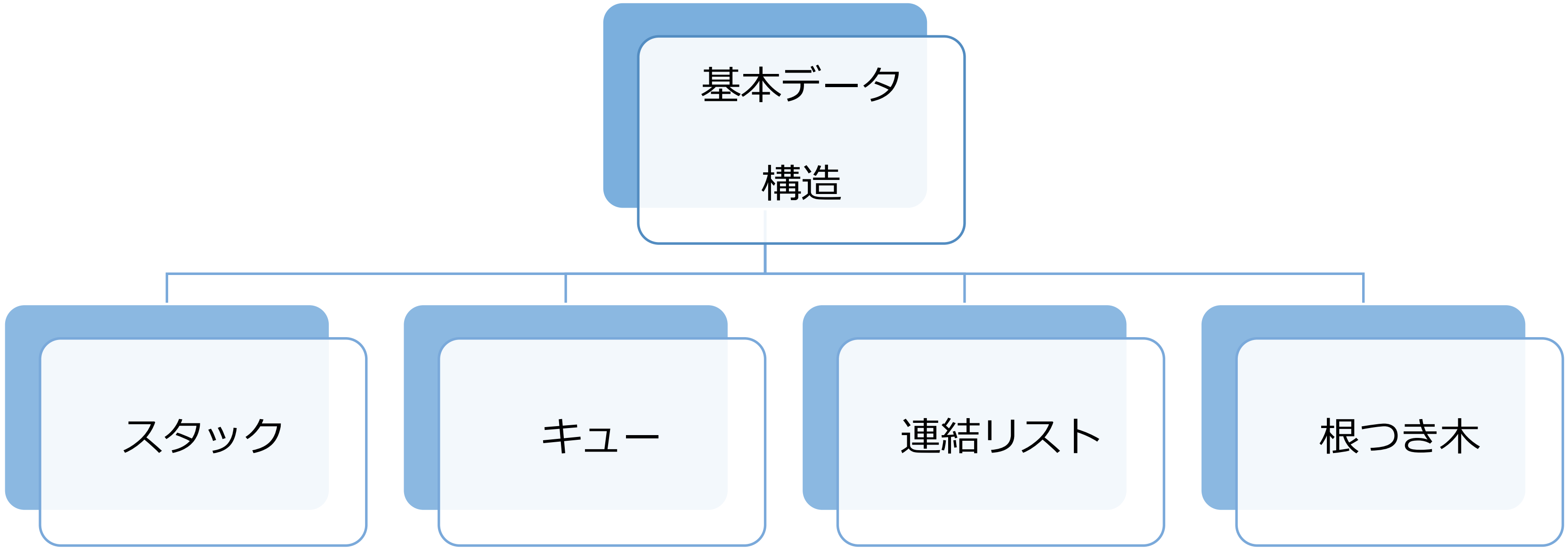
目次(Index)

基本

データ構造

ハッシュ表

基本データ構造



スタック(Stack)とキュー(Queue)

スタック(Stack)

LIFO

(last-in,first-out)

方策を実現する。

キュー(Queue)

FIFO

(first-in,first-out)

方策を実現する。

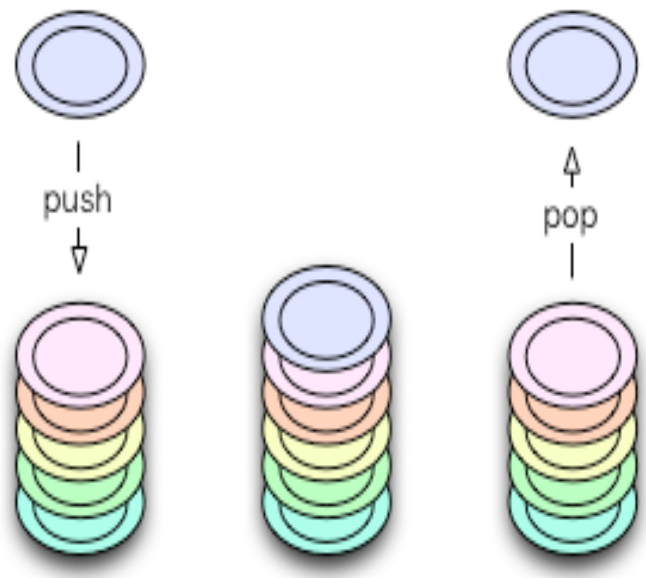
スタック(Stack)①

- スタック(Stack)…PUSHやPOPを用いて**末端**の要素のみを操作する。

配列S[1..n]を用いて実現する。

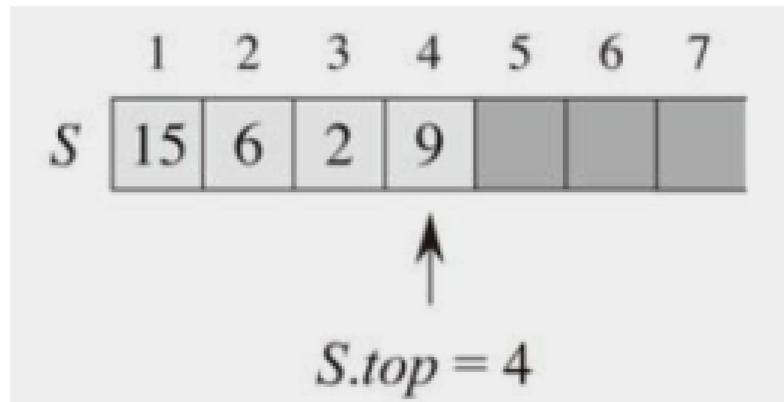
✓PUSH…スタックでの挿入(INSERT)操作

✓POP…スタックでの削除(DELETE)操作、要素を引数として取らない。

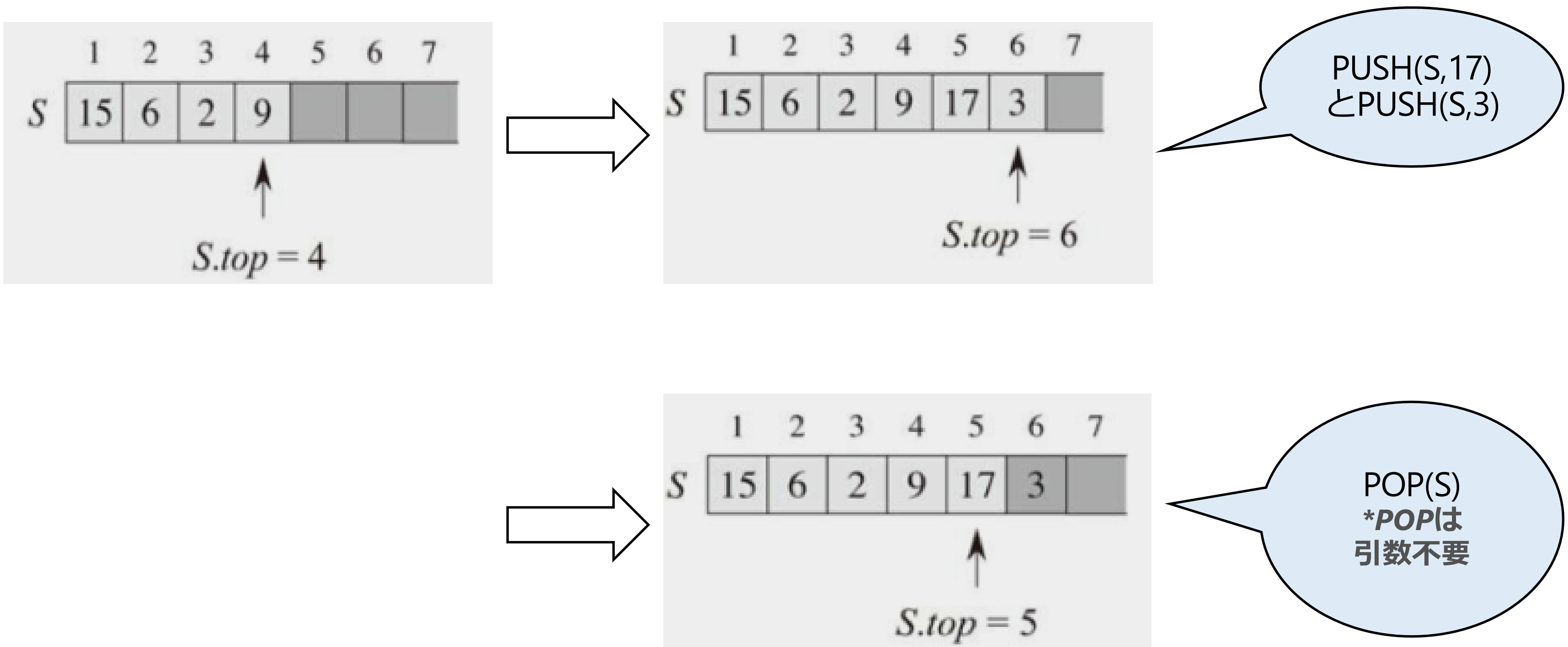


スタック(Stack)②

スタック…S[1..S.top]
から構成される。
S[1]…スタックの底
S[S.top]…スタックの
先頭要素
S.top=0→空(empty)



スタック(Stack)③


1 2 3 4 5 6 7
S 15 6 2 9
S.top = 4
1 2 3 4 5 6 7
S 15 6 2 9 17 3
S.top = 6
PUSH(S,17)
とPUSH(S,3)
1 2 3 4 5 6 7
S 15 6 2 9 17 3
S.top = 5
POP(S)
*POPは
引数不要

スタック(Stack)④

STACK-EMPTYのアルゴリズム

```
STACK-EMPTY(S)
1. if S.top==0
2.     return True
3. else False
```

スタック(Stack)⑤

PUSH(S,x)のアルゴリズム

```
PUSH(S,x)
1. S.top = S.top + 1
2. S[S.top] = x
```

スタック(Stack)⑥

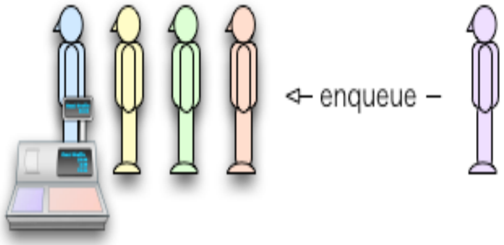
POP(S)のアルゴリズム

```
POP(S)
1. if STACK-EMPTY[S]
2.     error"underflow"
3. else S.top = S.top -1
4.     return S[S.top + 1]
```

3つのスタック(Stack)の操作の計算量はO(1)

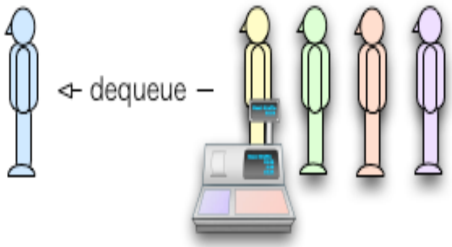
キュー(Queue)①

- キュー(Queue)…ENQUEUEやDEQUEUEを用いて**先頭**の要素のみを操作する。
  配列Q[1..n]を用いて実現する。



✓ENQUEUE…キューでの挿入(INSERT)操作

✓DEQUEUE…キューでの削除(DELETE)操作、要素を引数として取らない。



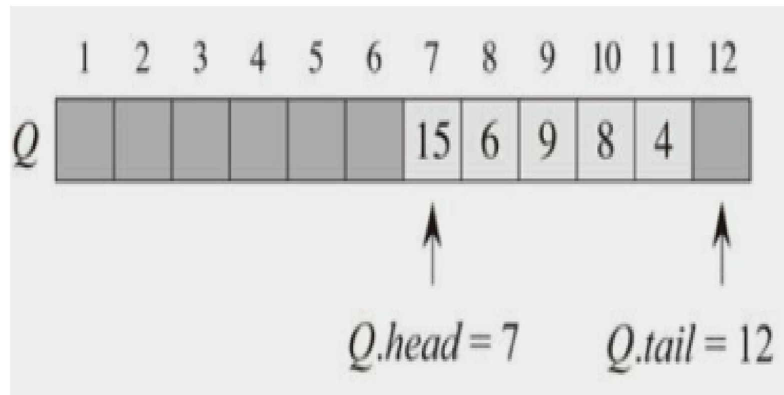
キュー(Queue)②

キュー…Q[Q.head..Q.tail-1]
から構成される。

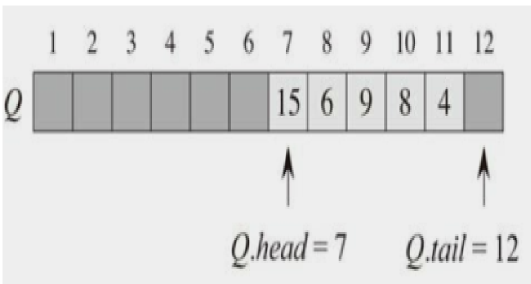
Q[Q.head]…キューの先頭要素

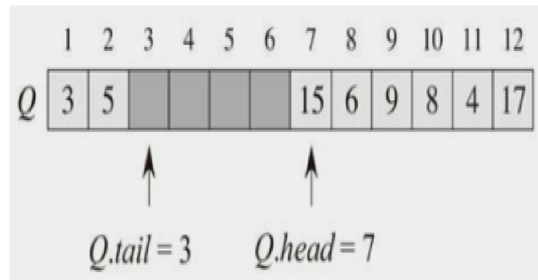
Q[Q.tail-1]…キューの末尾要素

Q.head=Q.tail→空(empty)

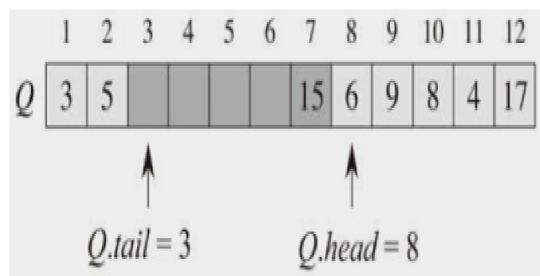


キュー(Queue)③





ENQUEUE
(S,17)
(S,3)
(S,5)



DEQUEUE
(S)
***DEQUEUE* は引数不要**

キュー(Queue)③

- ENQUEUE(Q,x)のアルゴリズム

```
ENQUEUE(Q,x)
1. Q[Q.tail] = x
2. if Q.tail == Q.length
3.     Q.tail = 1
4. else Q.tail = Q.tail + 1
```

キュー(Queue)

- DEQUEUE(Q)のアルゴリズム

```
DEQUEUE(Q)
1.  x = Q[Q.head]
2.  if Q.head == Q.length
3.      Q.head = 1
4.  else Q.head = Q.head + 1
5.  return x
```

2つのキュー
(Queue)
の操作の計算量
はO(1)

スタック(Stack)と キュー(Queue)まとめ

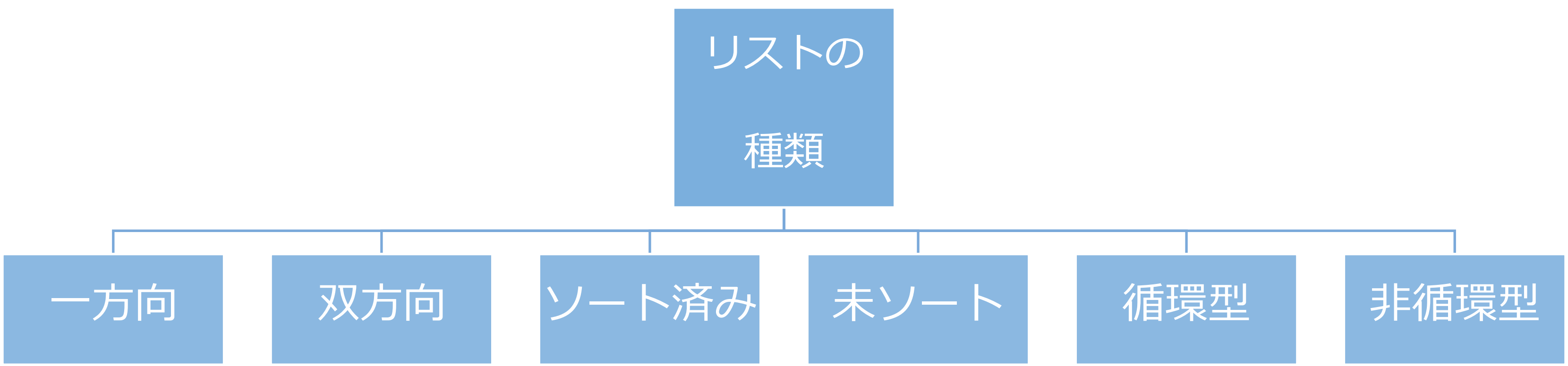
スタック(Stack)

削除は最後の要素を操作するデータ構造。

キュー(Queue)

削除は最初の要素を操作するデータ構造。

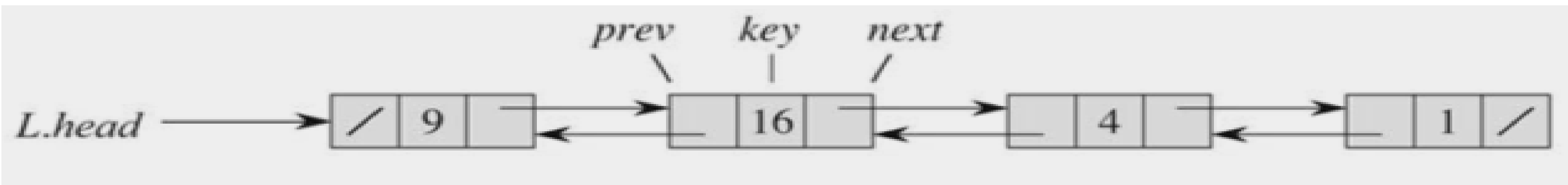
連結リスト①



連結リスト②

一方向リスト	各要素がキー(key)属性と 1つのポインタ属性nextを持つオブジェクト。
双方向リスト	各要素がキー(key)属性と 2つのポインタ属性nextとprevを持つオブジェクト。

双方向連結リスト①



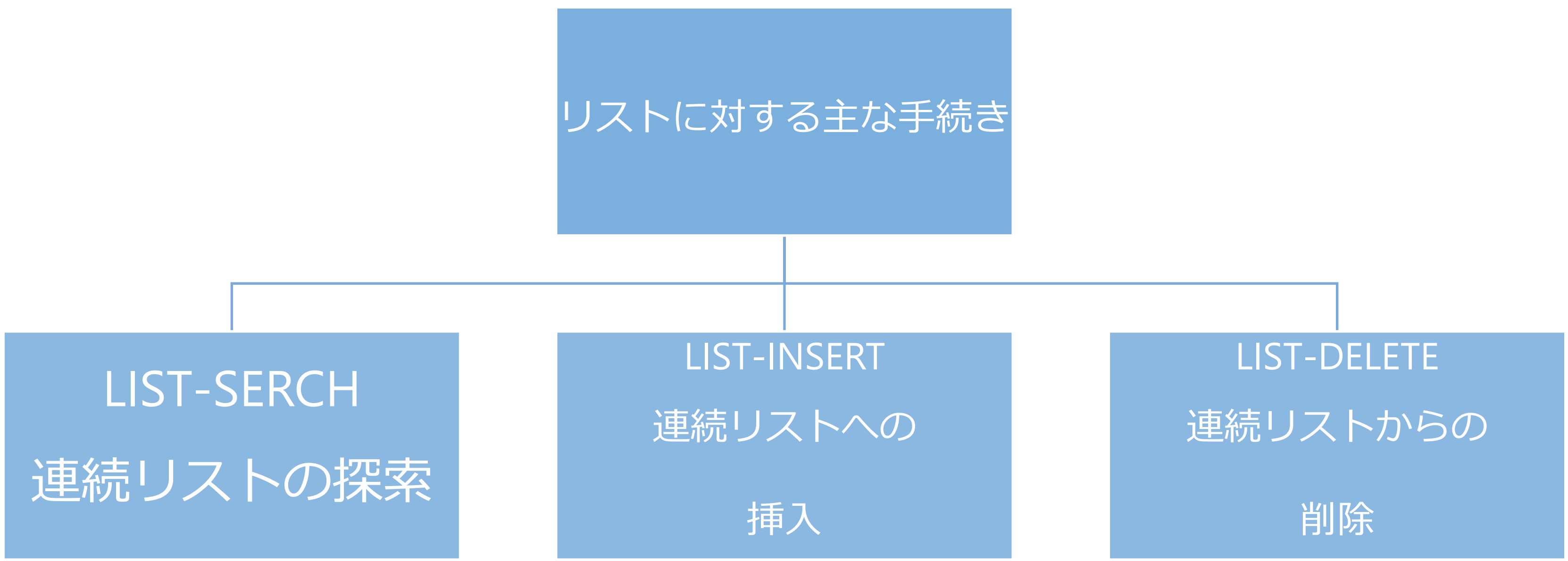
ある要素xに対して

x.next→xの次の要素を表す。x.next=NILならばxは末尾の要素。

x.prev→xの前の要素を表す。x.prev=NILならばxは先頭の要素。

L.head=NILならばリストは空(empty)

双方向連結リスト②
(以下、非循環未ソート)



双方向連結リスト③

- LIST-SEARCH(L,k)のアルゴリズム

```
LIST-SEARCH(L,k)
1.  x = L.head
2.  while x ≠ NIL and x.key ≠ k
3.      x = x.next
4.  return x
```

探索は
最悪時はリスト
全体を探索する
必要が
あるので
最悪実行時間はO(n)

双方向連結リスト④

- LIST-INSERT(L,x)のアルゴリズム

```
LIST-INSERT(L,x)
1. x.next = L.head
2. If L.head ≠ NIL
3.     L.head.prev = x
4. L.head = x
5. x.prev = NIL
```

実行時間は**O(1)**
である。

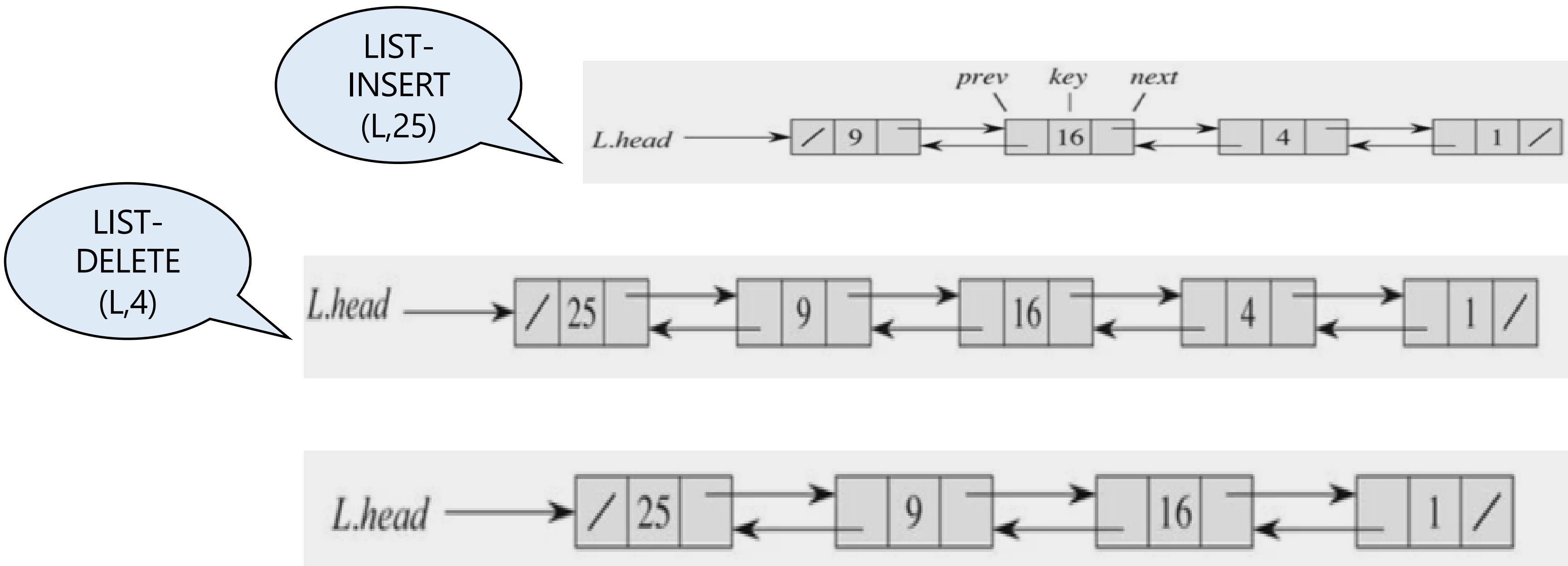
双方向連結リスト⑤

- LIST-DELETE(L,x)のアルゴリズム

```
LIST-DELETE(L,x)
1.  if x.prev ≠ NIL
2.      x.prev.next = x.next
3.  else L.head = x.next
4.  if x.next ≠ NIL
5.      x.next.prev = x.prev
```

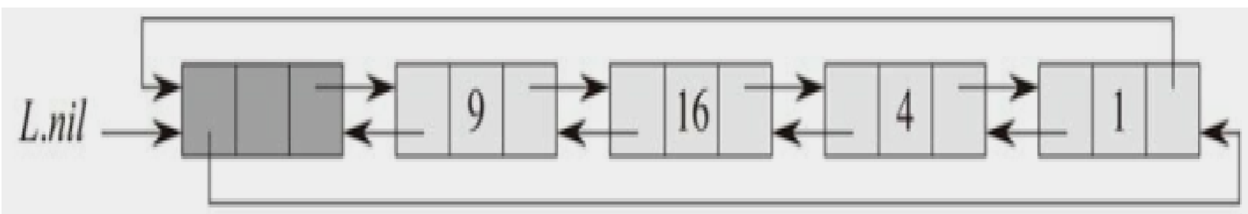
LIST-DELETE自体は
実行時間O(1)だが
その前にLIST-SEARCHを
呼び出す必要があるので
最悪時はO(n)である。

双方向連結リスト⑥


LIST-INSERT (L,25)
LIST-DELETE (L,4)
prev
key
next
L.head
9
16
4
1
L.head
25
9
16
4
1
L.head
25
9
16
1

番兵(Sentinel)①

番兵を持つ双方循環リストによってアルゴリズムの簡略化



L.nil.next→リストの先頭、L.nil.prev→リストの末尾
→疑似コードの明確化が目的
(注)**実行速度の改善**が主目的ではない。

番兵(Sentinel)②

- LIST-SEARCH’(L,k)のアルゴリズム

```
LIST-SEARCH’(L,k)
1.  x = L.nil.next
2.  while x ≠ L.nil and x.key ≠ k
3.      x = x.next
4.  return x
```

*疑似コードを比較

```
LIST-SEARCH(L,k)
1.  x = L.head
2.  while x ≠ NIL and
    x.key ≠ k
3.      x = x.next
4.  return x
```

番兵(Sentinel)③

- LIST-INSERT'(L,x)のアルゴリズム

LIST-INSERT'(L,x)
1. x.next = L.nil.next
2. L.nil.next.prev = x
3. L.nil.next = x
4. x.prev = L.nil

*疑似コードを比較

LIST-INSERT(L,x)
1. x.next = L.head
2. If L.head ≠ NIL
3. 　L.head.prev = x
4. L.head = x
5. x.prev = NIL

番兵④

- LIST-DELETE’(L,x)

LIST-DELETE’(L,x)
1. x.prev.next = x.next
2. x.next.prev = x.prev

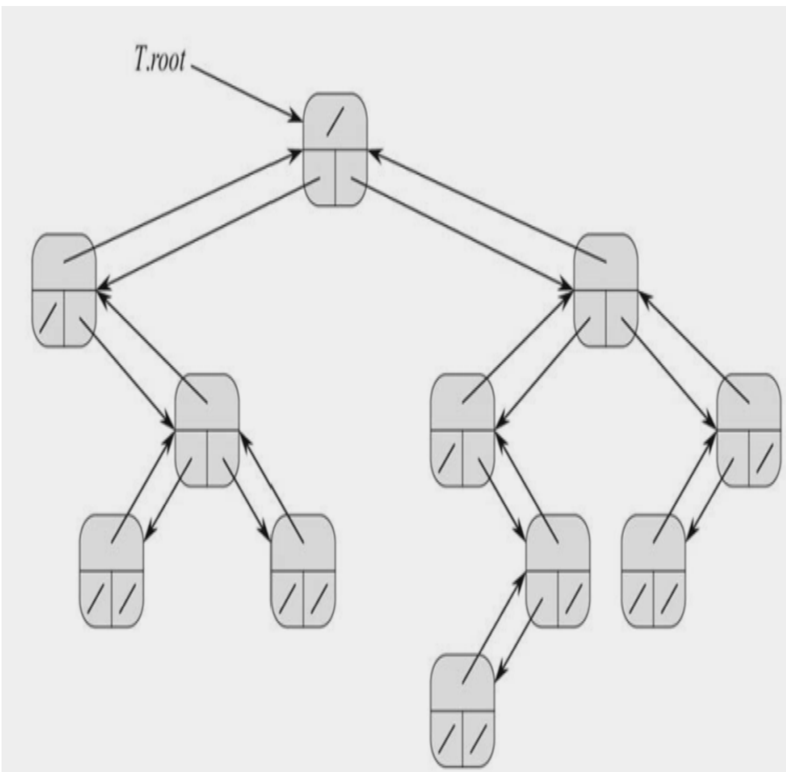
*疑似コードを比較
LIST-DELETE(L,x)
1. if x.prev ≠ NIL
2. x.prev.next = x.next
3. else L.head = x.next
4. if x.next ≠ NIL
5. x.next.prev = x.prev

リストまとめ

SEARCH,INSERT,DELETE

番兵と循環型リストを用いることで疑似コードの簡略化

二分木



いくつかの頂点(vertex)と

辺(edge)にて構成される。

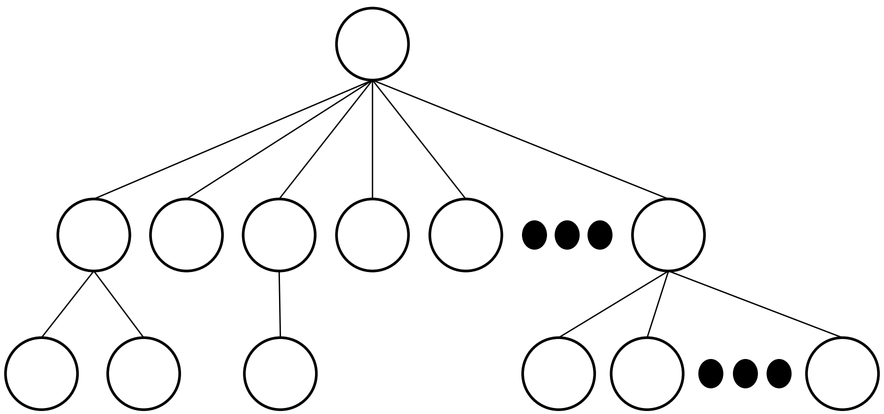
根(root)から木の枝のように辺が伸びる。

p,left,rightのポインタからなる。

✓x.p = NILならば根(root)である

✓T.root = NILならば空(empty)

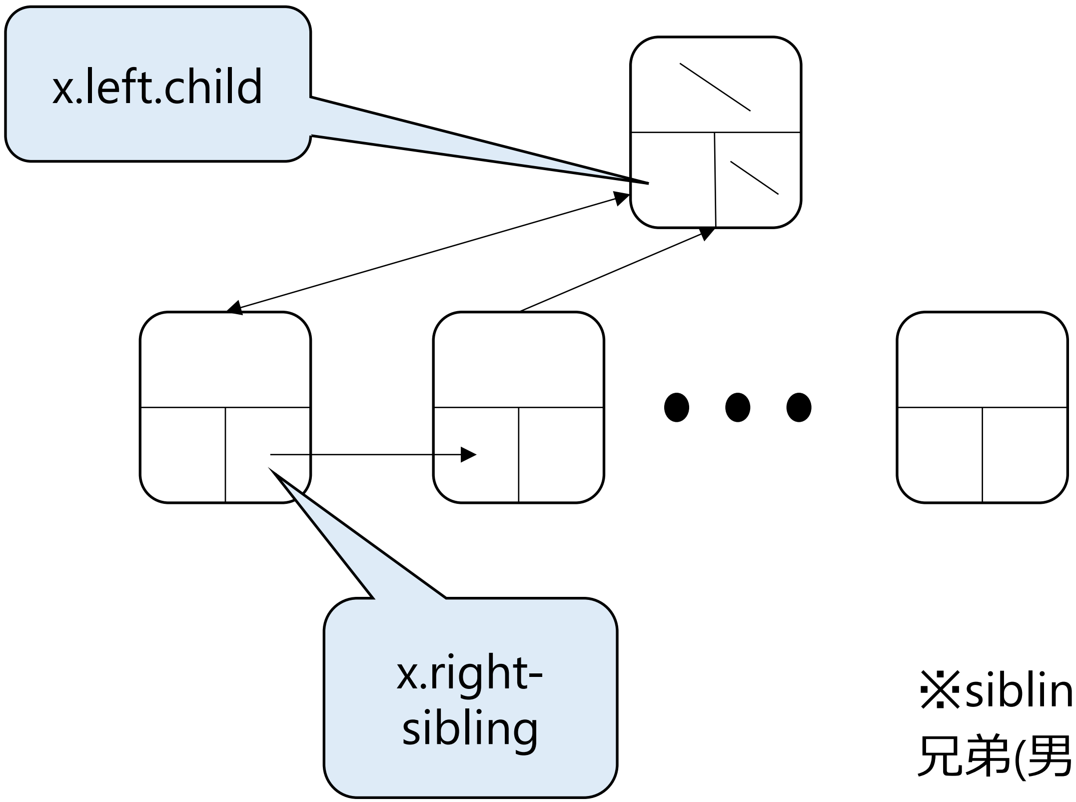
制約なし根つき木①



- 子の数が定数でない場合、属性数が確定できないのでこの方法では　　不可能
- 子の数kが定数であっても記憶領域を大きく消費する可能性がある。

→二分木で表現する。

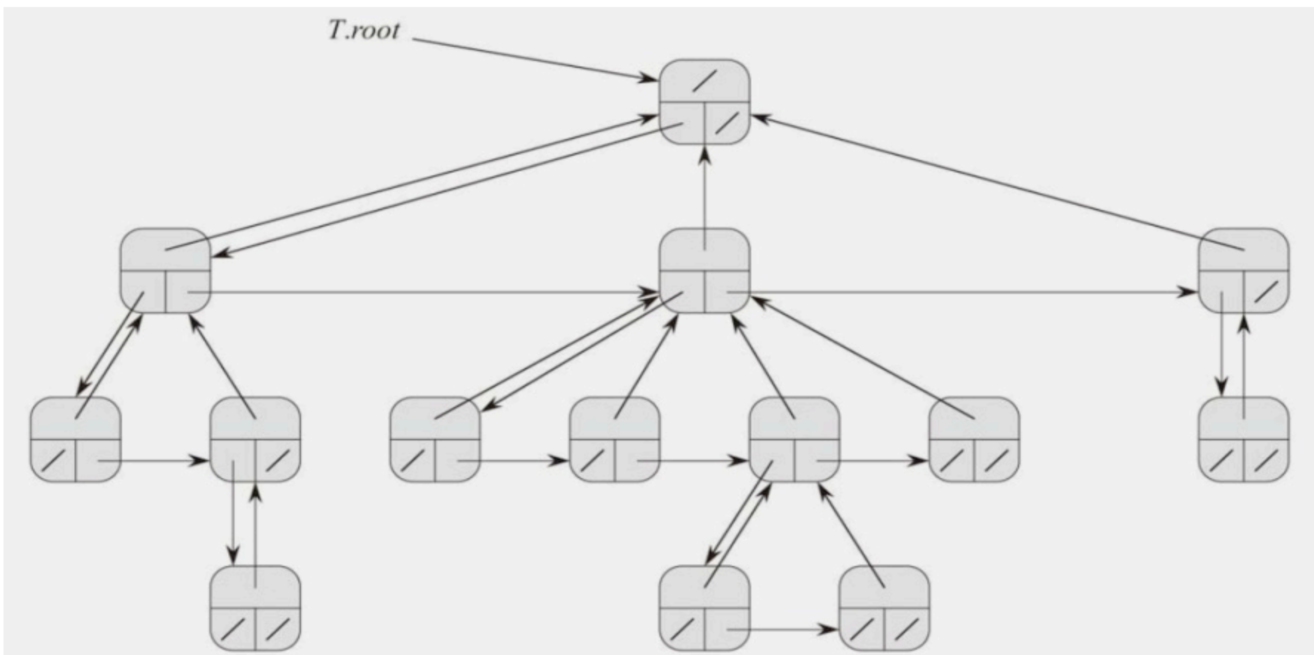
制約なし根つき木②



※sibling…
兄弟(男女関係なく)

制約なし根つき木③

※二分木での表現



ハッシュ表

直接

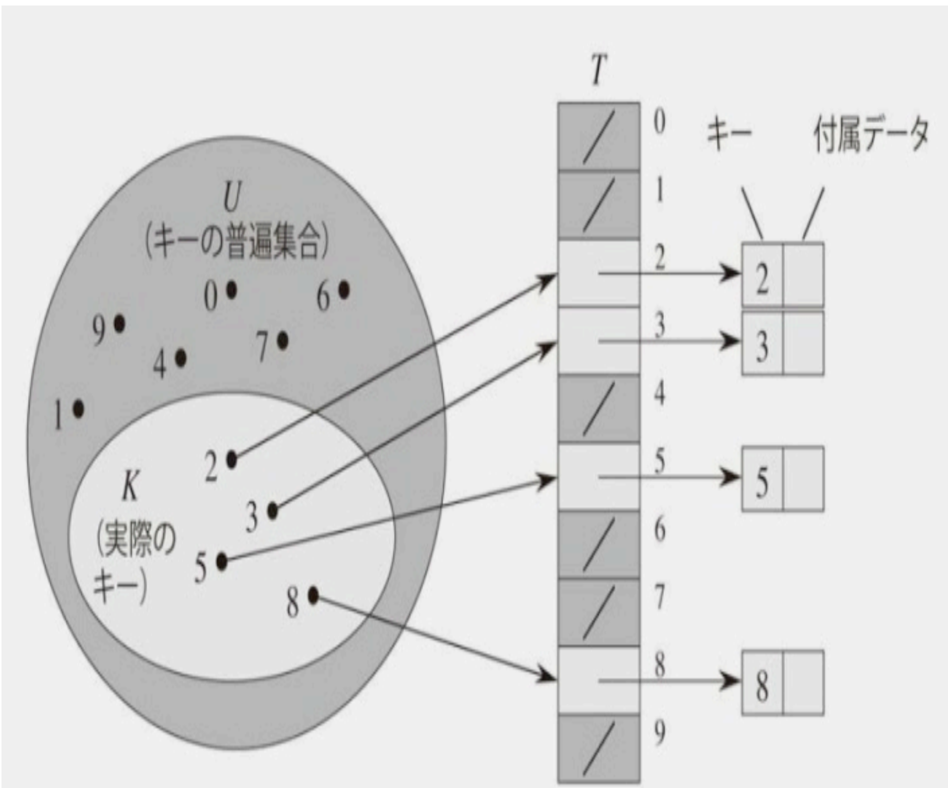
アドレス表

ハッシュ表

直接アドレス表

- 普遍集合U={0,1,2...m-1}より選択されるキーを持つ動的集合の表現方法、各要素は異なるキーを持つと仮定。
- 配列T[0,1,...m-1]を用いる。
- 集合がキーkを持つ要素を含まないとき
  T[k] = NIL

直接アドレス表



- SEARCH

return T[k]

- INSERT

T[x.key] = x

- DELETE

T[x.key] = NIL

計算時間は**O(1)**となる。

直接アドレス表

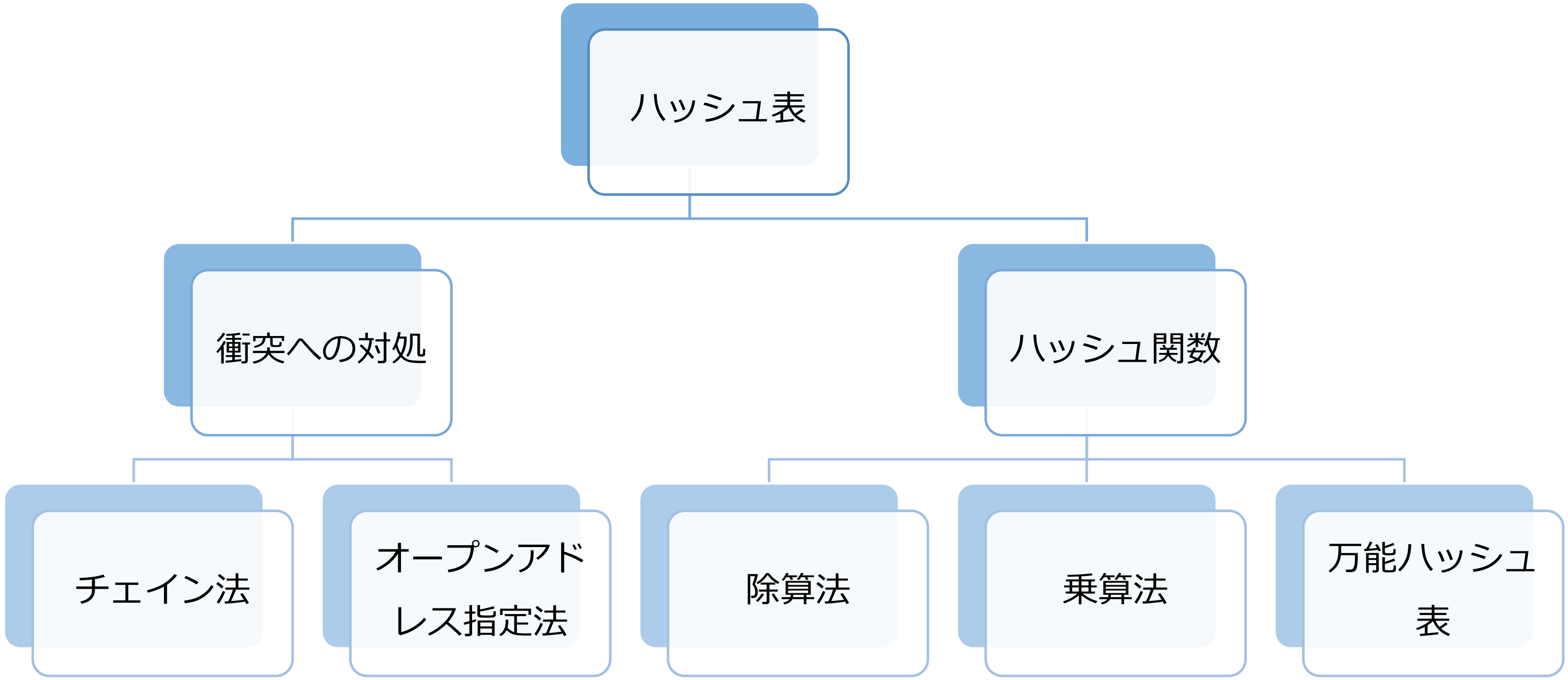
2つの弱点

①キーの普遍集合Uが非常に大きい時に、コンピューターの記憶領域を超えてしまう。

②実際に格納されるキーの集合がUと比較して非常に小さい時に、Tに割り当てられた配列が無駄になる。

→**ハッシュ法**を用いて表現する。

ハッシュ表



ハッシュ表

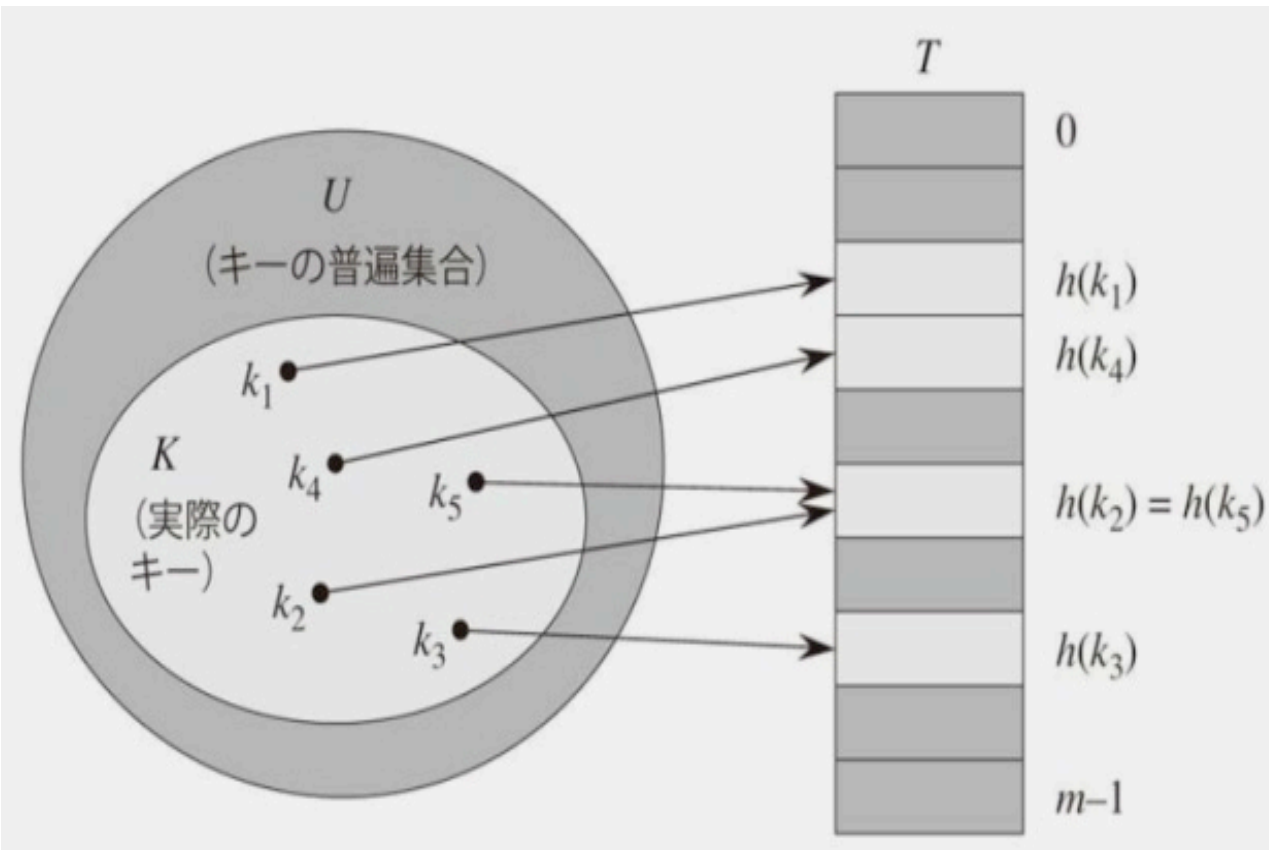
ハッシュ法…キーkをh(k)に格納する。

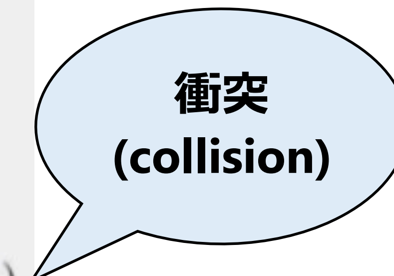
　h…ハッシュ関数、キーの普遍集合Uから
ハッシュ表T[0..m-1]の枠への集合の写像

⇩

$$\boldsymbol{h:U \rightarrow \{0,1,\ldots m-1\}}$$

ハッシュ表


T
0
U
(キーの普遍集合)
$h(k_1)$
$h(k_4)$
k_1
K
k_4
k_5
(実際のキー)
$h(k_2) = h(k_5)$
k_2
k_3
$h(k_3)$
$m-1$


衝突
(collision)

ハッシュ表

- ハッシュ表の長所

直接アドレス表と比較して、十分に小さな領域しか必要としない。

- ハッシュ表の短所

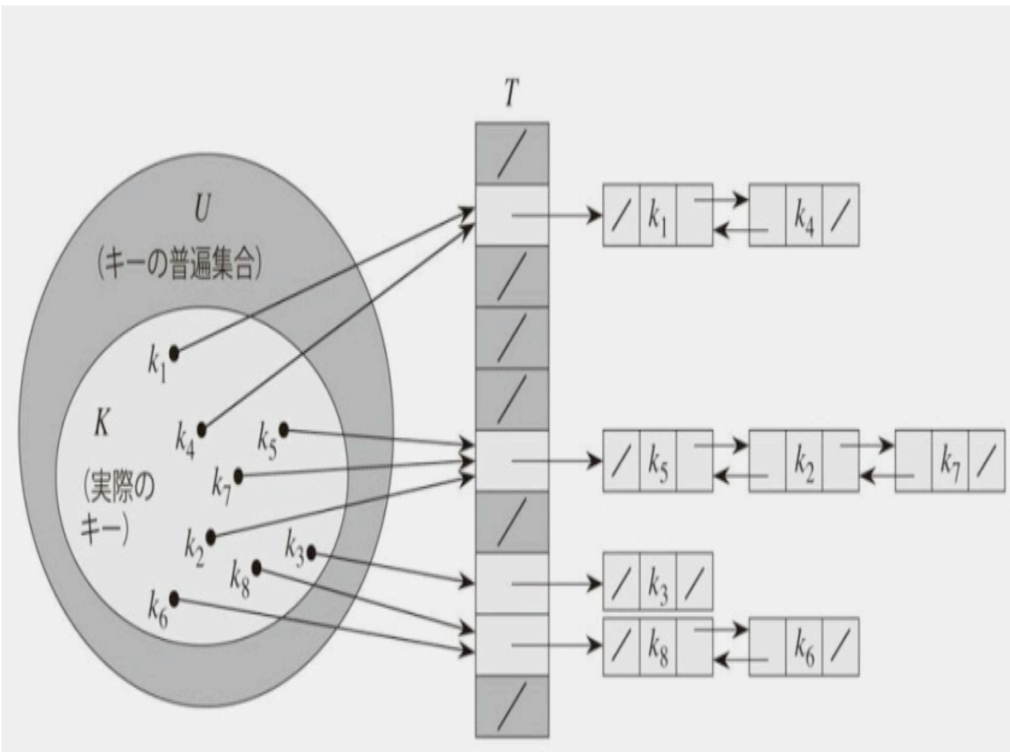
衝突の発生

⇒衝突の発生が生じないようなハッシュ関数の設定。

⇒完全に発生しないようにするのは不可能

☛先に解決策を用意しておく。

チェイン法



チェイン法…

衝突している枠で
全ての要素を
1つの連結リスト
を用いて表現する。

チェイン法の時間評価

- INSERT(T,x):リストT[h(x.key)]にxを挿入

→実行時間**O(1)**

- DELETE(T,x):リストT[h(x.key)]からxを削除

→実行時間**O(1)**

- SEARCH(T,k):リストT[h.(x.key)]からキーkを持つ要素を探索

探索の時間評価①

n個の要素を格納する枠mの負荷率α=n/m

最悪探索時間$\Theta(n)$→1つの枠にn個のキーが全てハッシュされたとき。
単純一様ハッシュ仮定の下では

成功時:$\Theta(1+\alpha)$

失敗時:$\Theta(1+\alpha)$

探索の時間評価②

単純一様ハッシュ仮定

任意に与えられた要素は、他の要素が既にどの枠にハッシュされているか関係なく、m個の任意の枠に等確率にハッシュされるという仮定

結論:ハッシュ表の枠数が表が含む要素数に少なくとも比例すると

$n=O(m)$、$\alpha=n/m=O(m)/m=O(1)$

ハッシュ表全ての辞書操作は平均時間*O(1)*

ハッシュ関数

優れたハッシュ関数

→<u>単純一様ハッシュ仮定</u>をおおよそ満たす。

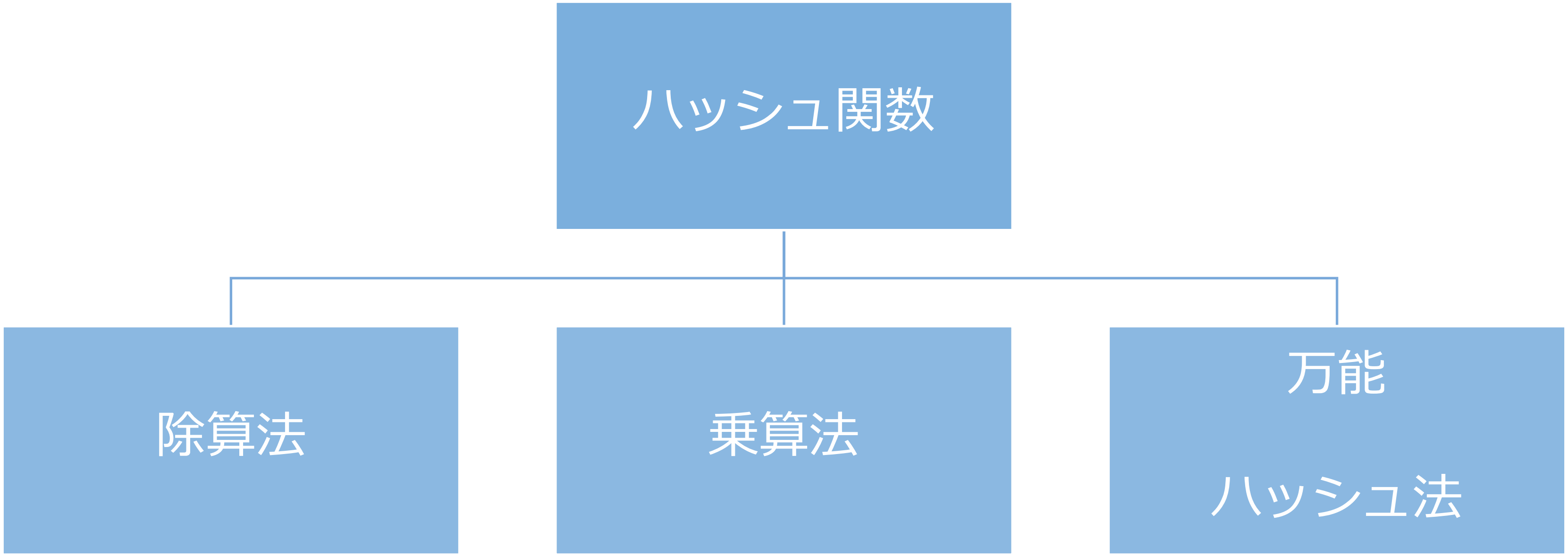
任意に与えられた要素は、他の要素が既にどの枠にハッシュされているか関係なく、m個の任意の枠に等確率にハッシュされるという仮定

例えば、$0 \leq k \leq 1$を満たすkが独立且つ一様

→h(k)=$\lfloor km \rfloor$

は優れたハッシュ関数である。

ハッシュ関数



除算法

h(k) = k mod m

※m = 2^pは避ける

※2のべき乗に近くない素数

乗算法

① キーkに定数A(0<A<1)をかける
② kAの小数部分を取り出して、任意の定数mをかける。
③ その結果の小数部分を捨てる。

h(k) = $\lfloor m(kA \bmod 1) \rfloor$
kA mod 1 = kA - $\lfloor kA \rfloor$

乗算法

※mを2のべき乗に設定する。

※Knuthは

$$A \approx {\left(\sqrt{5}-1\right)}\Big/{2} = 0.6180339887 \ldots$$

が優れていると検証した。

万能ハッシュ法

固定されたハッシュ関数に対して悪意のあるキーの選択⇒**最悪探索時間$\Theta(n)$となる。**

(対策)
万能ハッシュ法の設計

ランダムにハッシュ関数を選択する。
・異なるキーk,lに対してh(k)=k(l)(衝突する)となる確率が1/m以下の場合にハッシュ関数の有限集合$\mathcal{H}$は万能と考えられる。

万能ハッシュ表設計

1. 全てのキーkに対して$0 \leq k \leq p-1$を満たす十分大きな素数pを選ぶ。
2. $\mathbb{Z}_p = \{0,1..p-1\}, \mathbb{Z}_p^* = \{1,..p-1\}$とする。
3. $a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p$に対してのハッシュ関数を

$$\boldsymbol{h_{ab} = \big((ak+b) mod\, p\big) mod\, m}$$
$$\boldsymbol{\mathcal{H}_{pm} = \{h_{ab} : a \in \mathbb{Z}_p^* \text{かつ} b \in \mathbb{Z}_p\}}$$

⇒$\mathcal{H}_{pm}$はp(p-1)個のハッシュ関数を含む。

また、$\mathcal{H}_{pm}$は万能、つまり衝突確率は1/m以下

オープンアドレス指定法

衝突が発生した時のアプローチ

→

・衝突を起こした枠でリストを用いる。

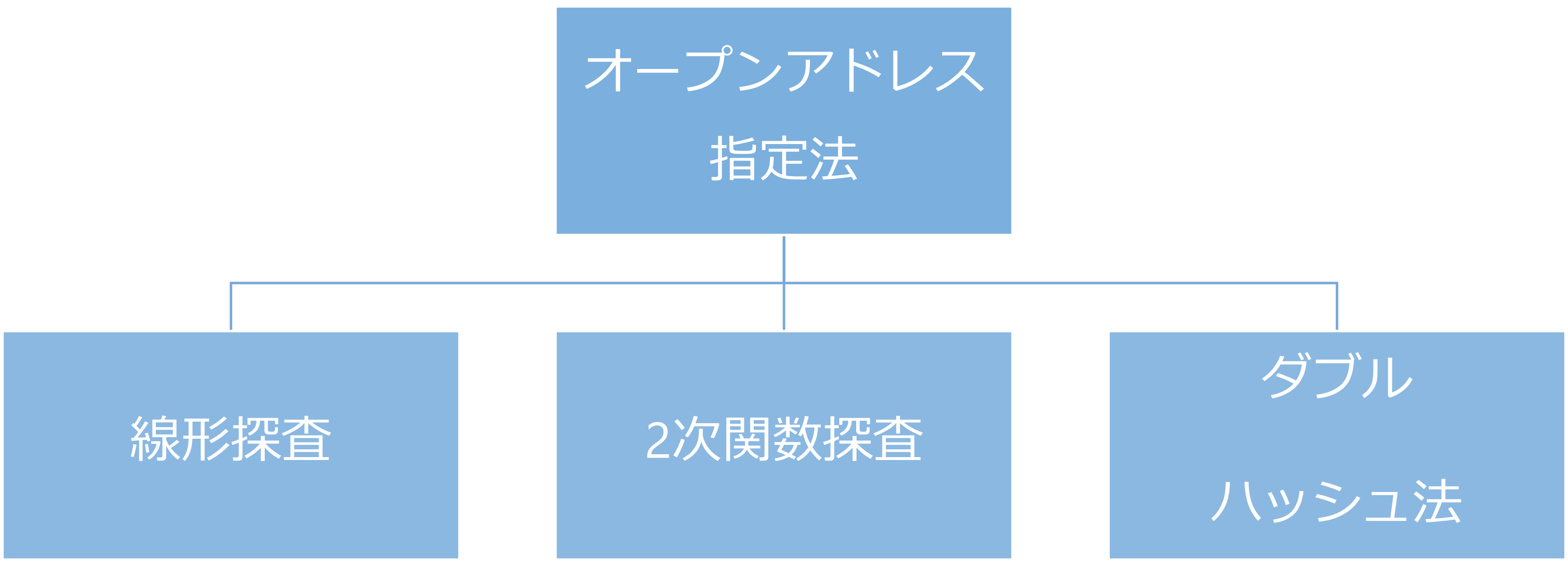
チェイン法

・空いている枠を探す。

オープンアドレス指定法

ポインタを用いないので
容量減、衝突減
探索の高速化の
可能性

オープンアドレス指定法



線形探査

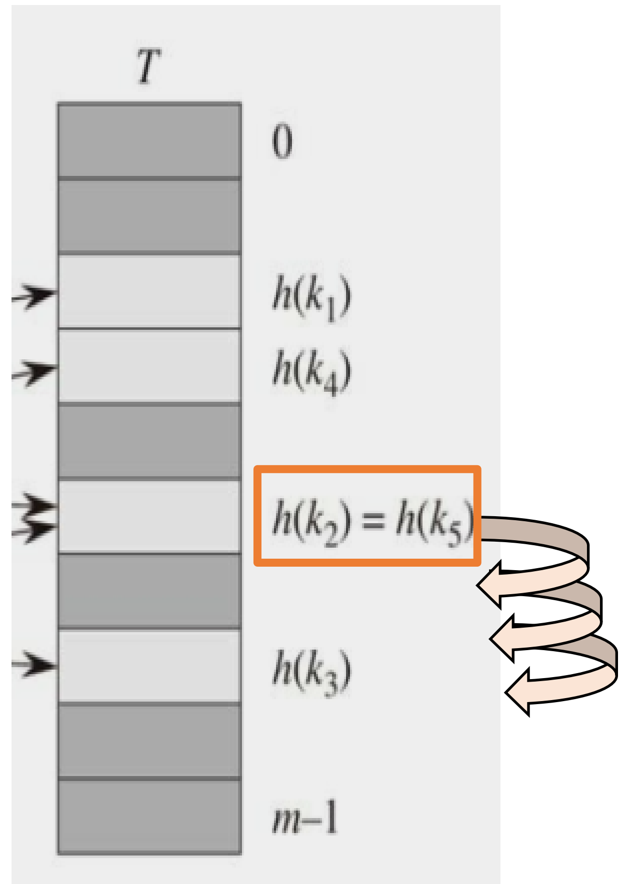
h′:U→{0,1..m-1},i=0,1..m-1として

h(k,i) = (h′(k) + i) mod m

h′(k)を補助ハッシュ関数という。

問題点:**主クラスタ化**

…長い区間の枠が埋まっているとき、平均探査時間の悪化



2次関数探査

h(k,i) = (h′(k) + $c_1 i$ + $c_2 i^2$) mod m

c_1, c_2は正の補助定数。

問題点:**副クラスタ化**
…同じ初期探査位置を持つ2つのキーは同じ探査列を持つ。
h(k_1, 0)=h(k_2, 0)ならばh(k_1, i)=h(h_2, i)

ダブルハッシュ法

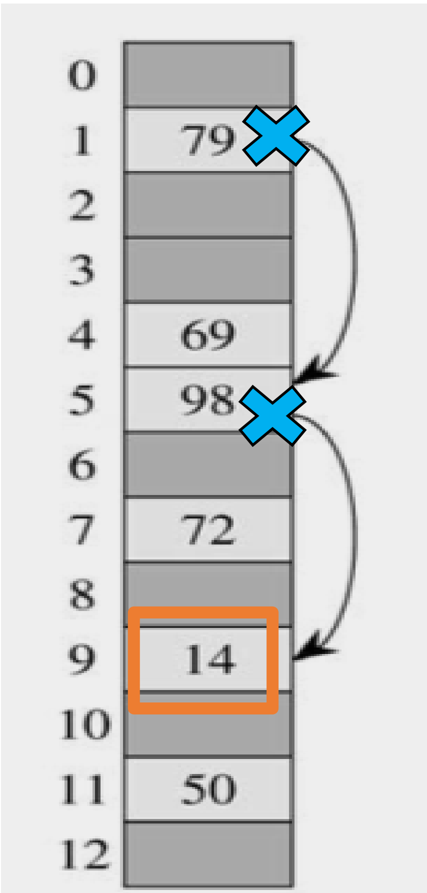
h(k,i) = $(h_1(k) + ih_2(k))$ mod m

初期位置と次に探査する位置までの距離が
一方又は両方変わる可能性がある。

⇒オープンアドレス指定法に利用できる

最良の方法の1つである。

ダブルハッシュ法



$h_1(k) \ = k \ mod \ 13$

$h_2(k) = 1 + k \ mod \ 11$

と設定してk=14の挿入

$h_1(14) \ = 14 \ mod \ 13 \ \equiv 1$

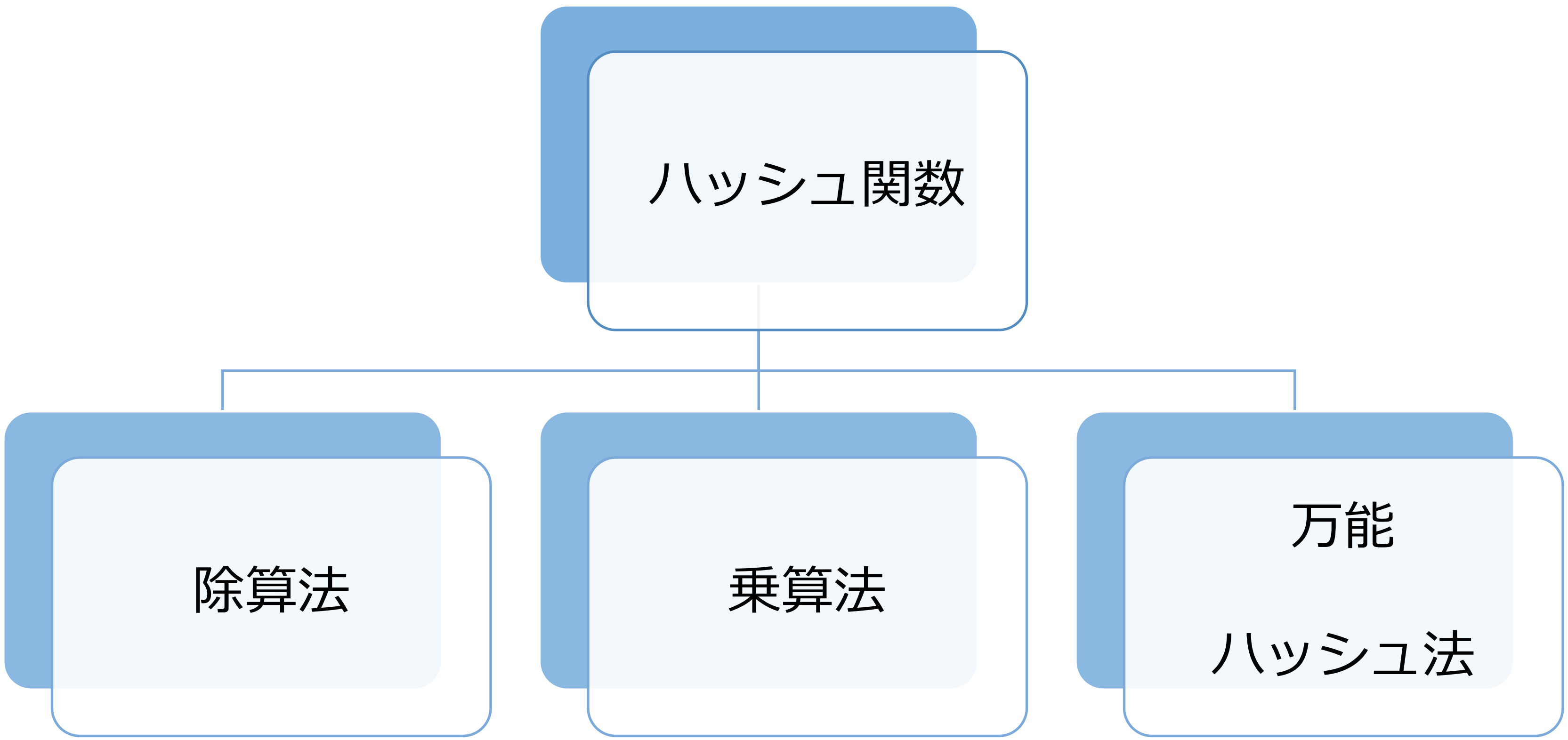
$h_2(14) = 1 + 14 \ mod \ 11 \ \equiv 4$

1回目(1+0・4)mod 13≡ 1

2回目(1+1・4)mod 13≡ 5

3回目(1+2・4)mod 13≡ 9

ハッシュ関数まとめ



ハッシュ表まとめ