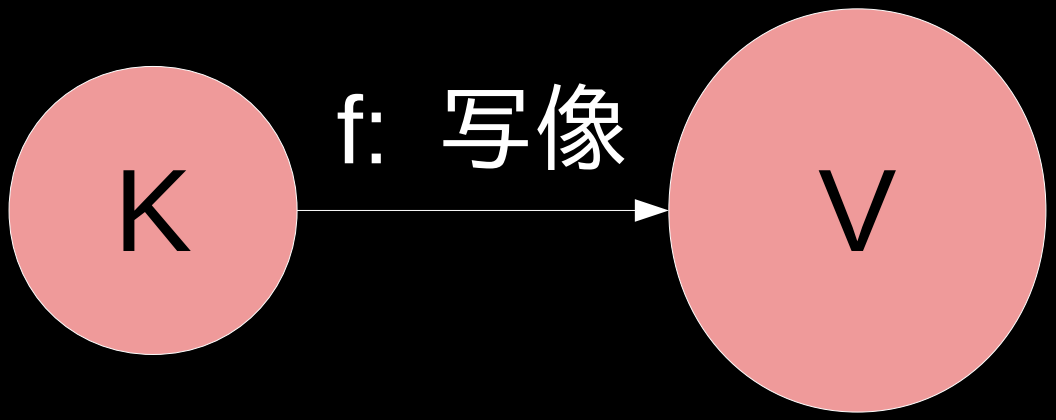
Compact Hashing
Meets
Separate Chaining

発表者 :
クップル ドミニク
九州大学

第 173 回アルゴリズム研究会
令和 1 年 5 月 10 日

連想配列 / 辞書

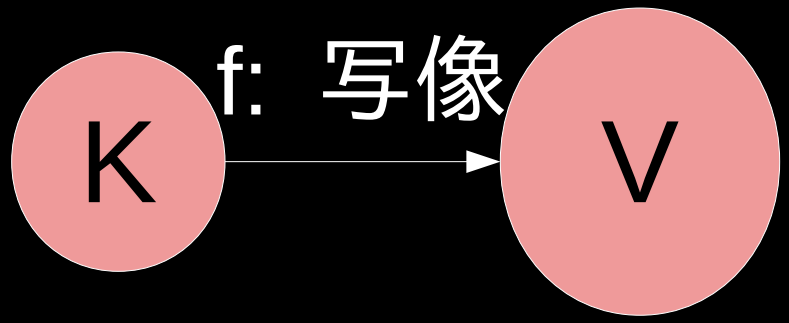


f の表現するデータ構造

- 探索木
- hash table

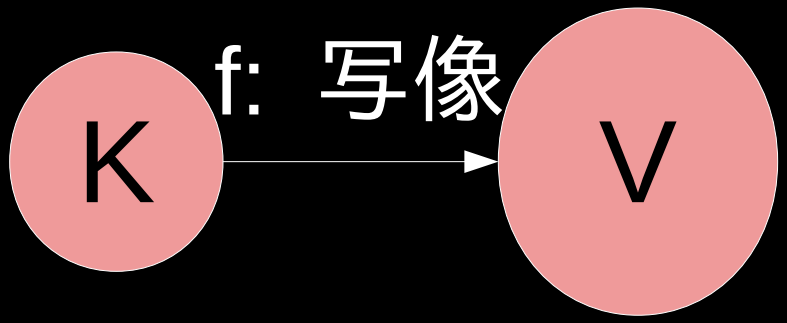
設定

- K = [1..|K|] 整数
- V = [1..|V|] 整数



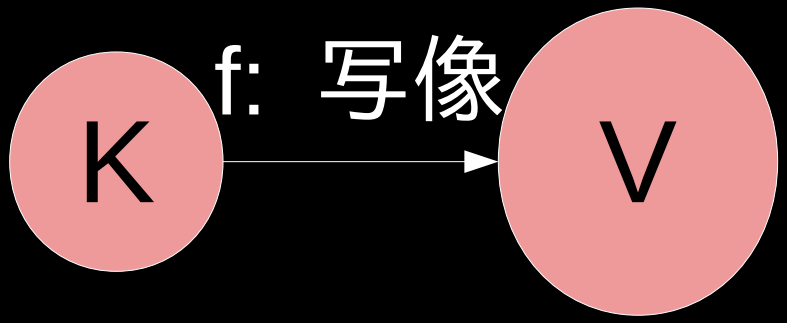
設定

- K = [1..|K|] 整数
- V = [1..|V|] 整数
- |K| < 2^{20} 場合は
  - 普通の配列で保存できる
  - 領域： lg |V| MiB



設定

- K = [1..|K|] 整数
- V = [1..|V|] 整数
- |K| < 2^{20} 場合は
  - 普通の配列で保存できる
  - 領域： lg |V| MiB

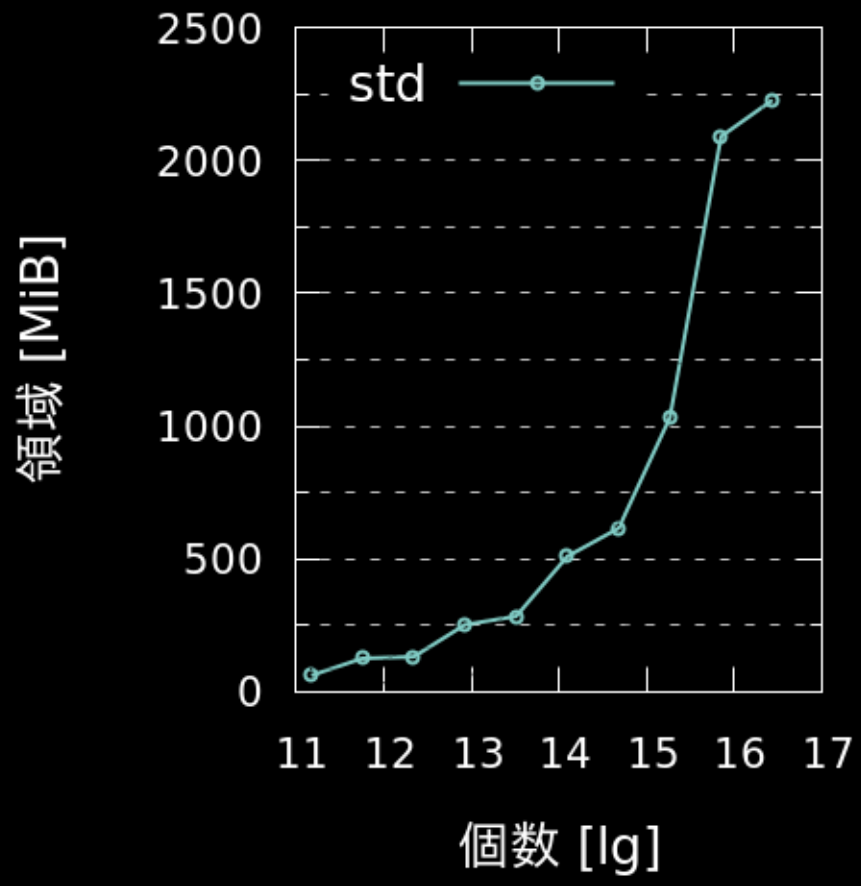


今回は

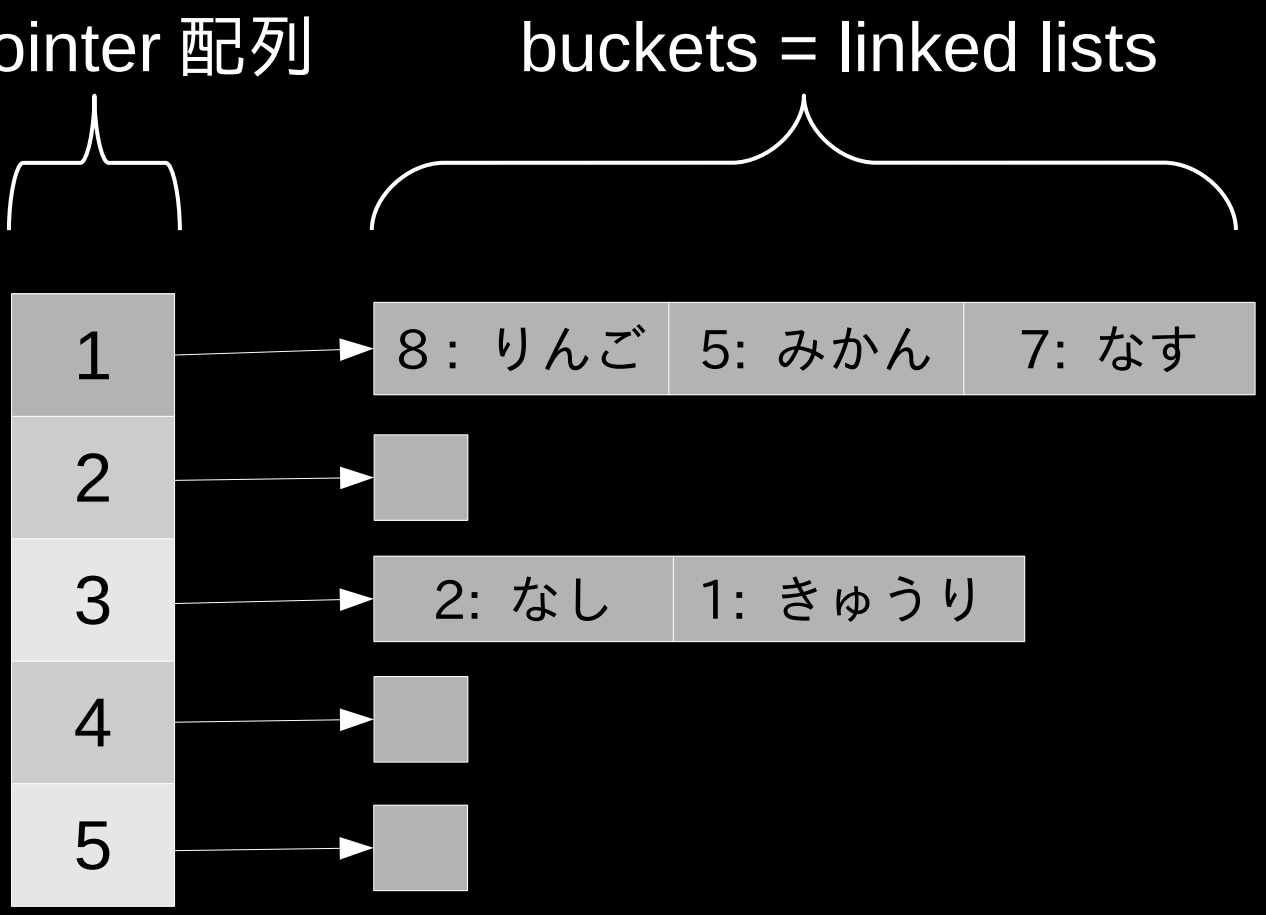
- |K| = 2^{32}
- |V| = 2^{32}

- 問題：
  - 32 bit key
  - 32 bit value
  - 無作為 (random)

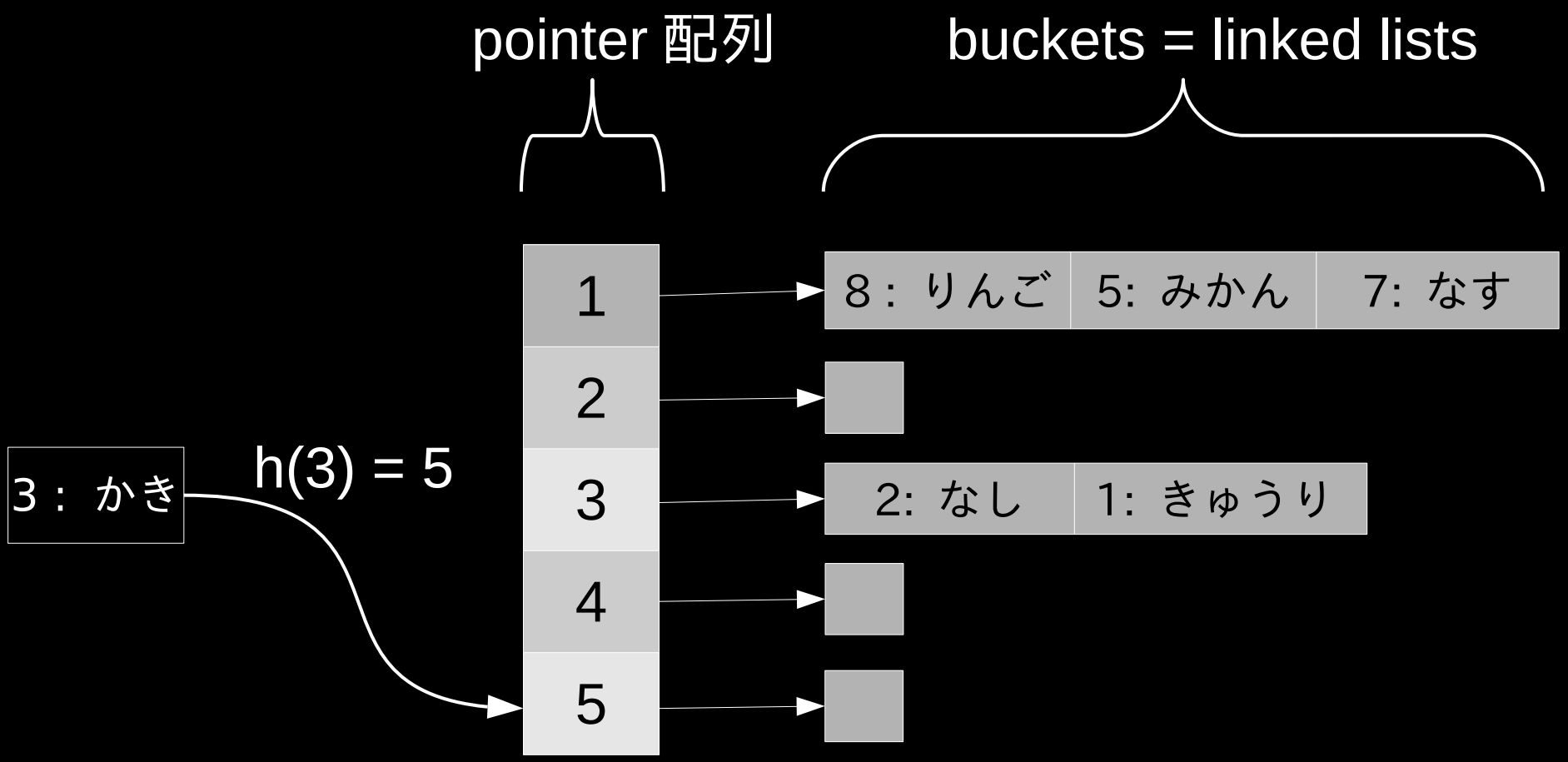
- 問題:
  - 32 bit key
  - 32 bit value
  - 無作為 (random)
- std: C++ の「unordered_map」
  - separate chaining
  - 領域を節約なし



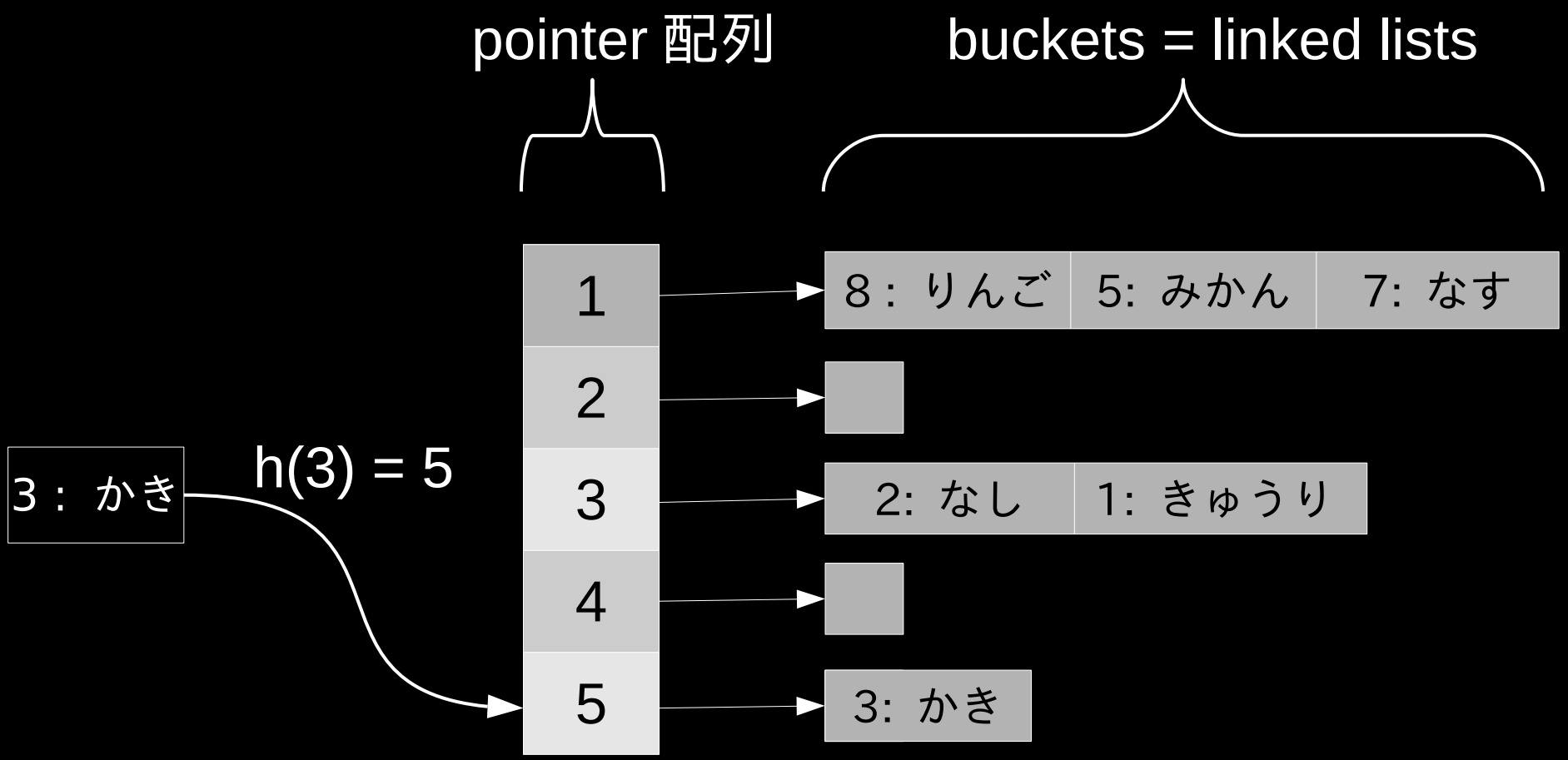
separate chaining



separate chaining



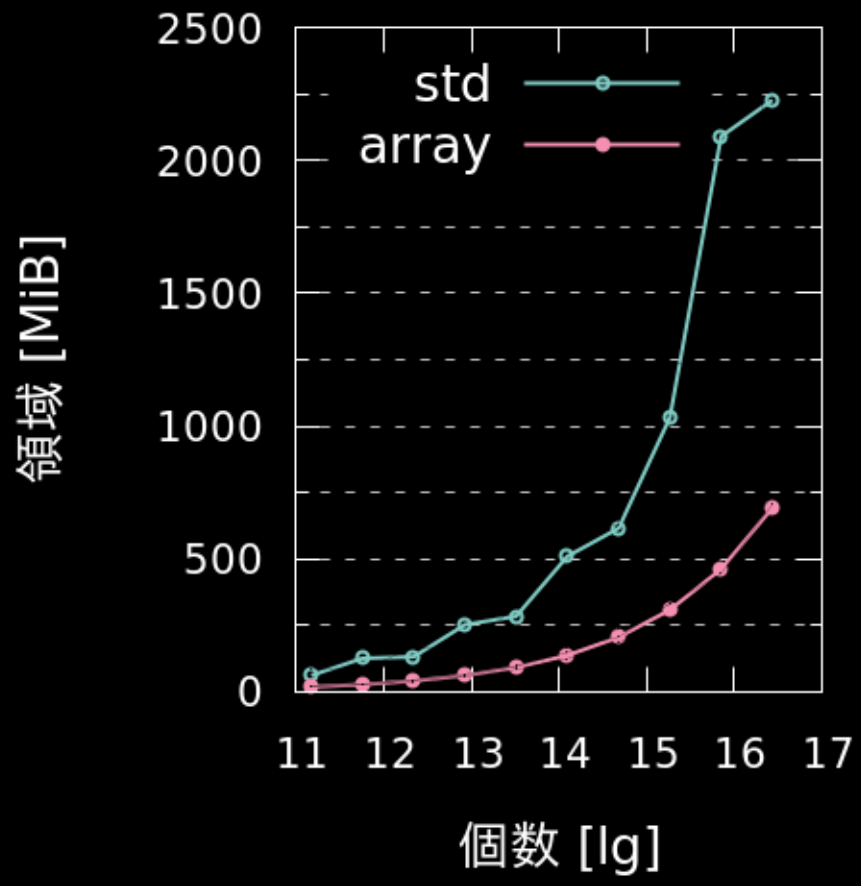
separate chaining



２つの配列

array:

- key 配列
- value 配列
- 挿入順序



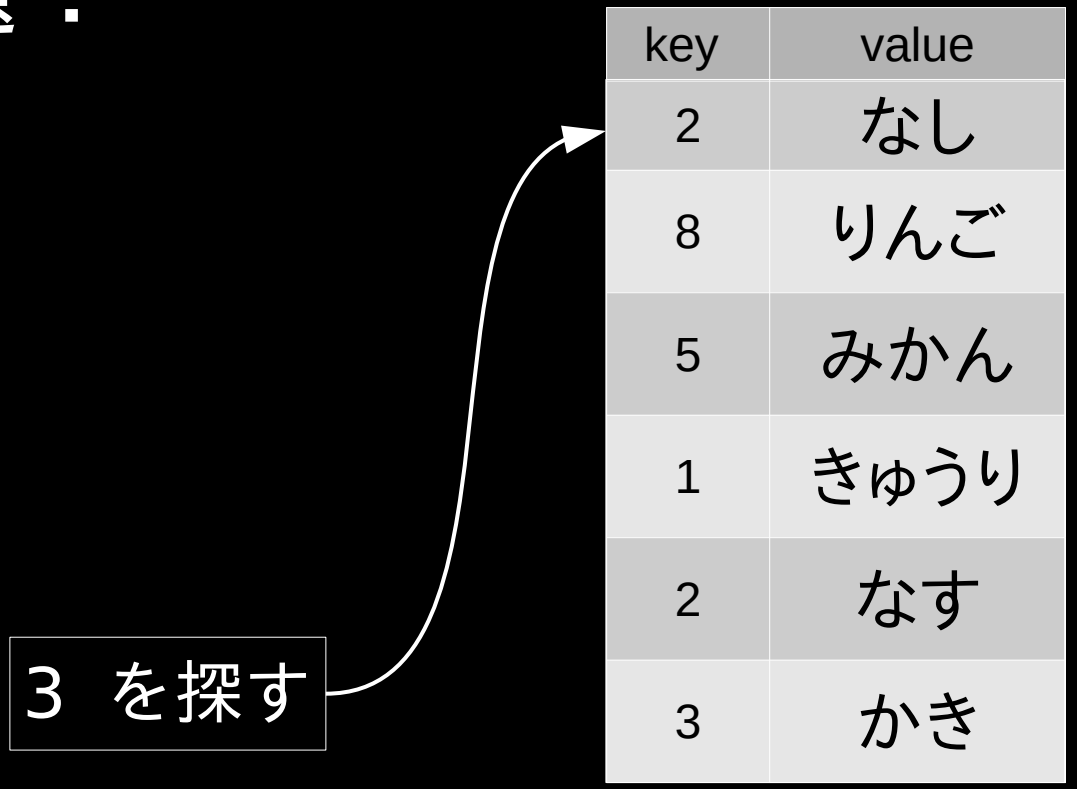
２つの配列

時間の問題：

key	value
2	なし
8	りんご
5	みかん
1	きゅうり
2	なす
3	かき

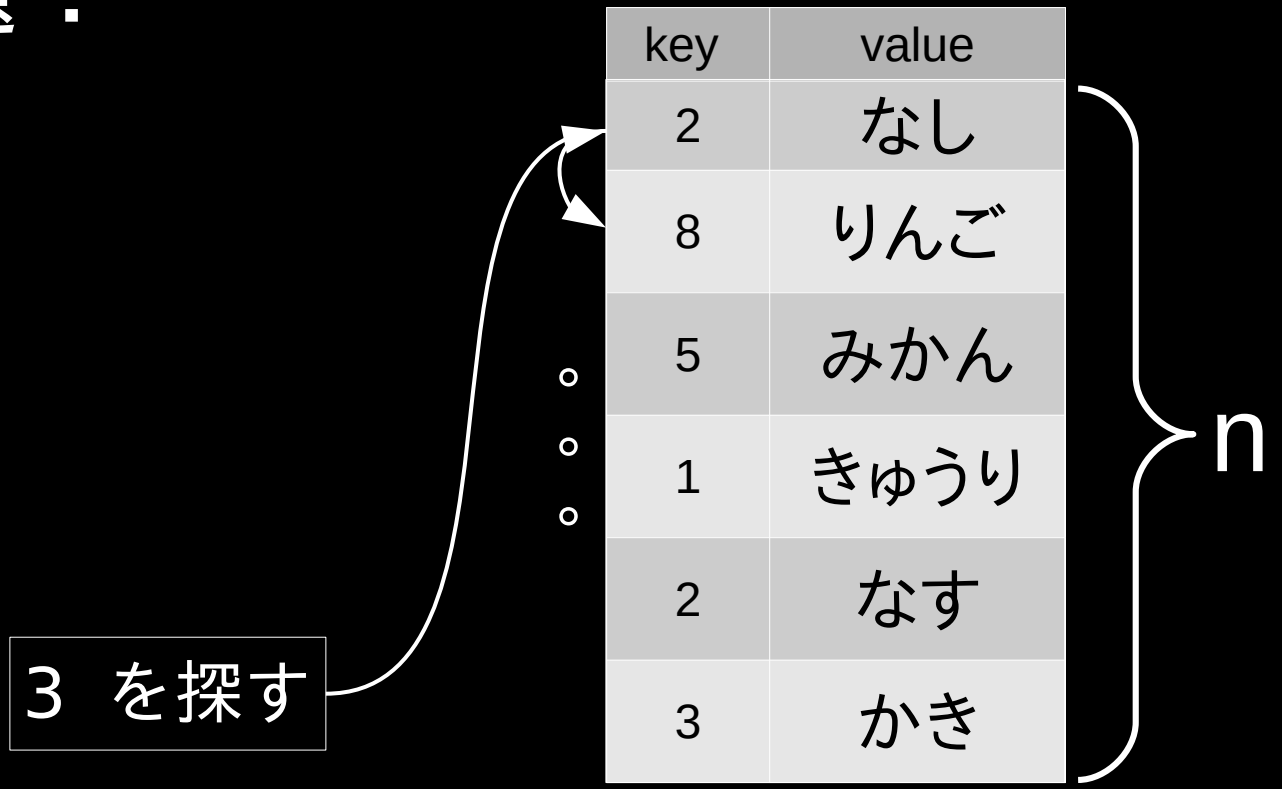
２つの配列

時間の問題：



２つの配列

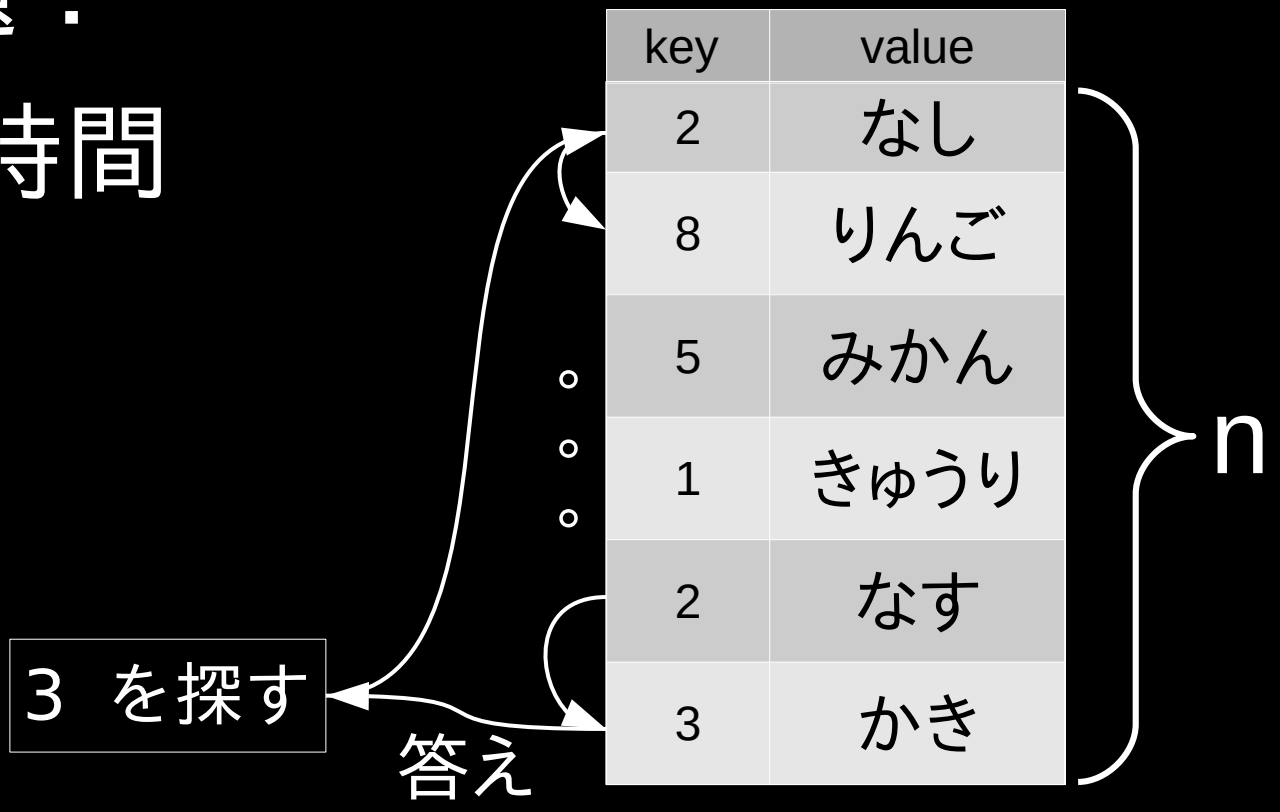
時間の問題：



２つの配列

時間の問題：

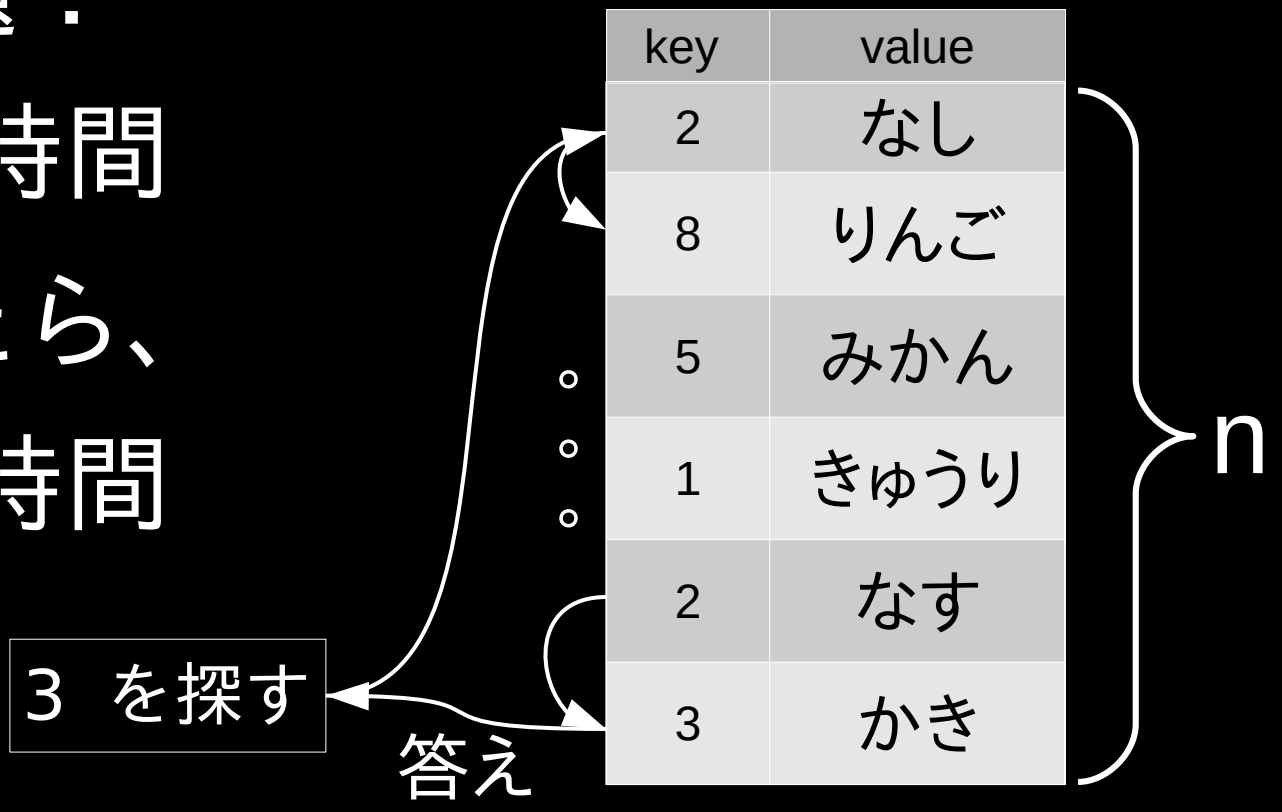
- O(n) 探索時間



２つの配列

時間の問題：

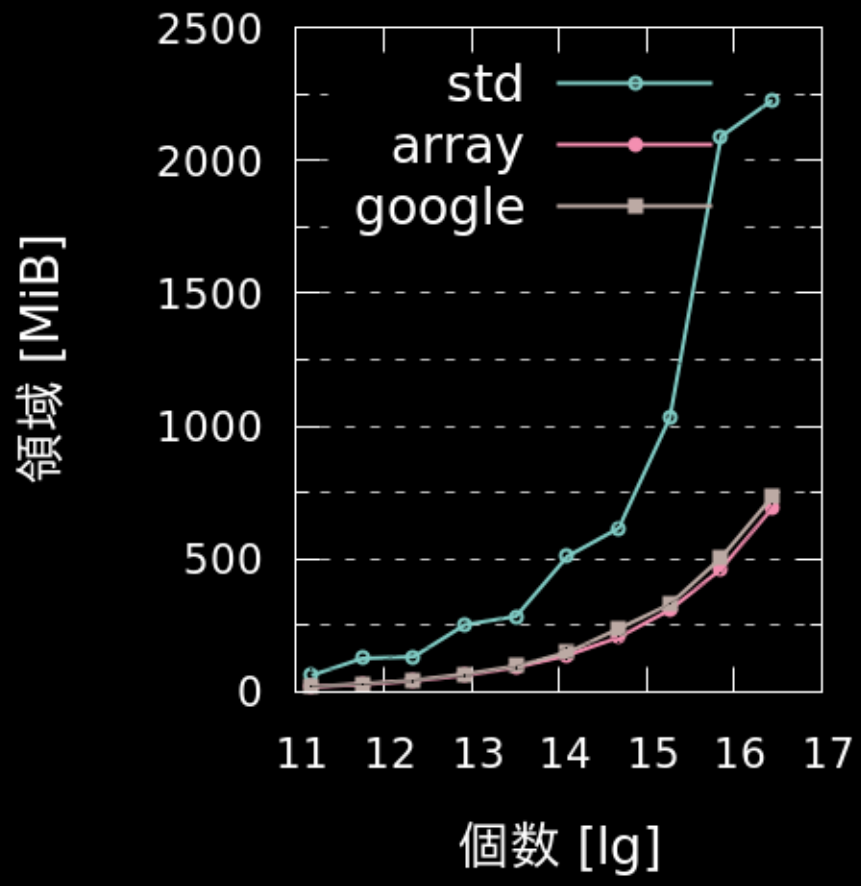
- O(n) 探索時間
- ソートしたら、
  O(n) 挿入時間



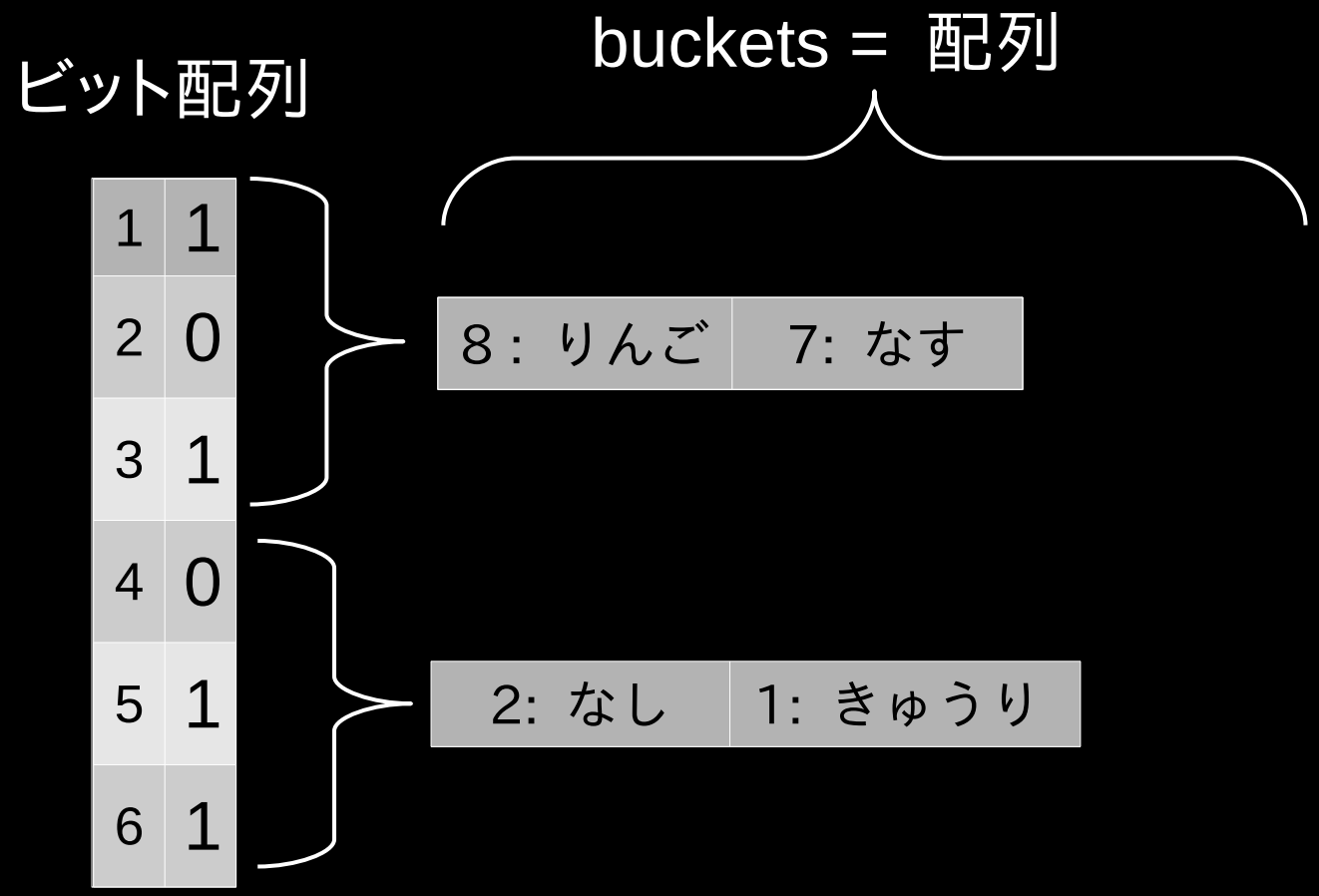
google sparse hash

google:

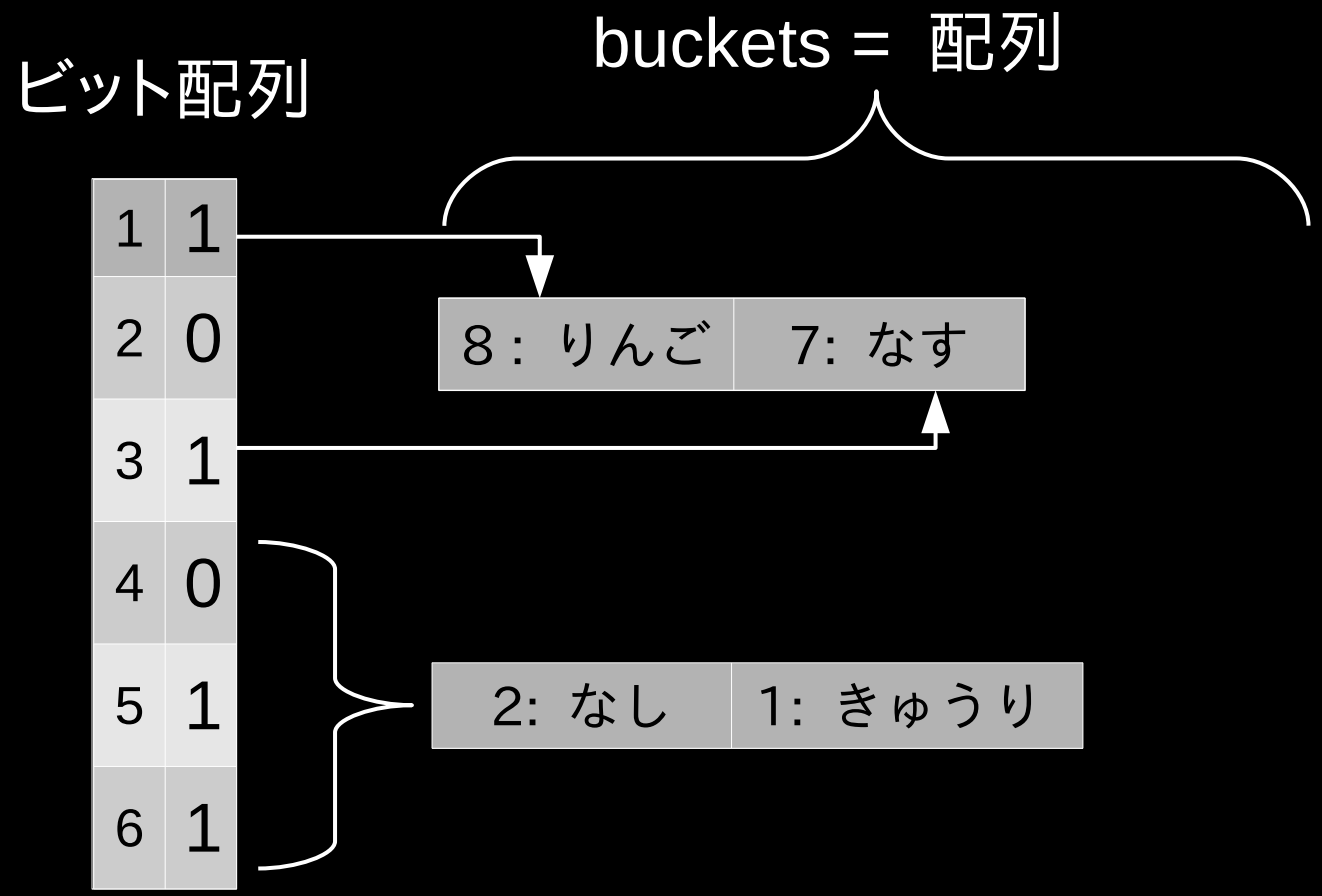
- closed hash table
- ビット配列
- buckets



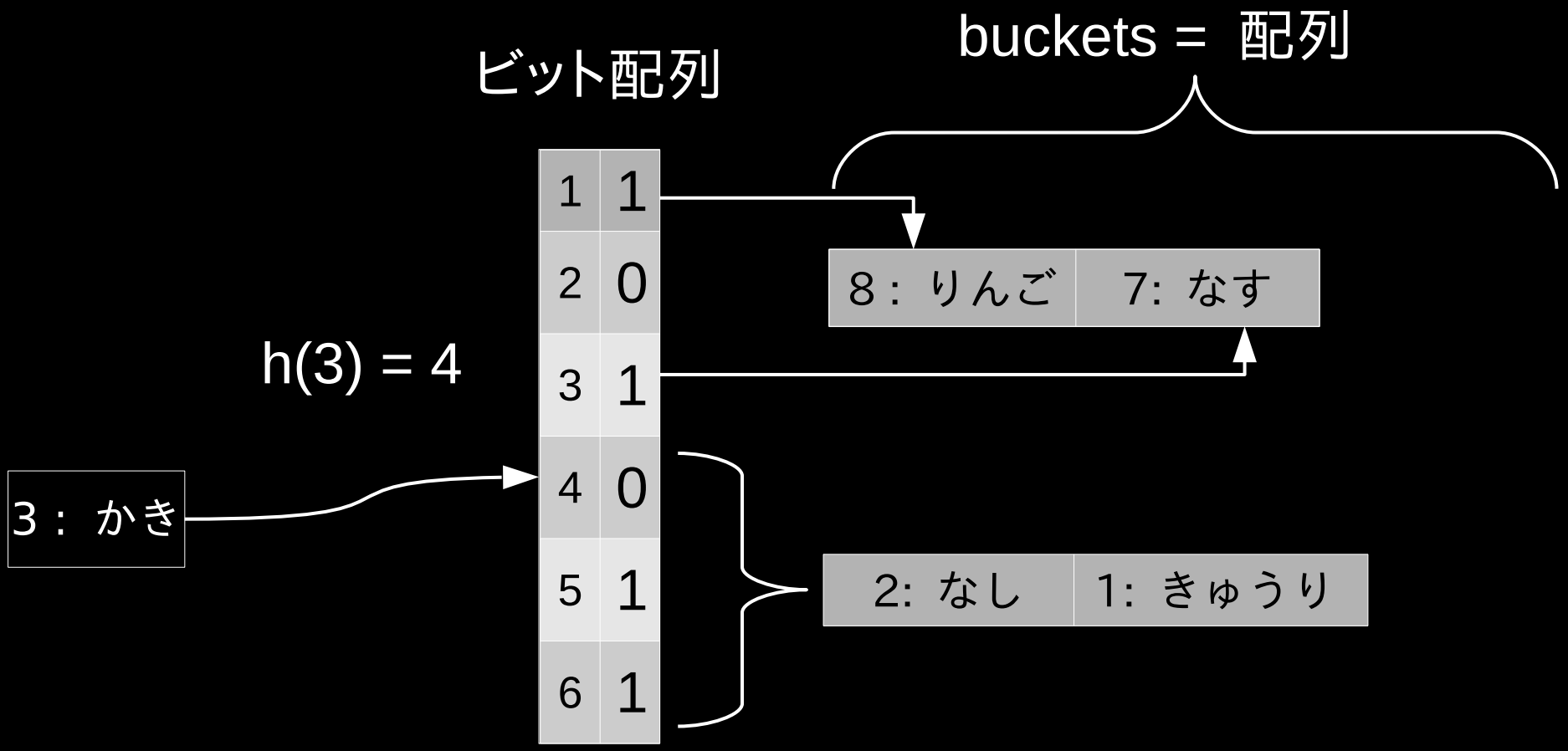
sparse hash table



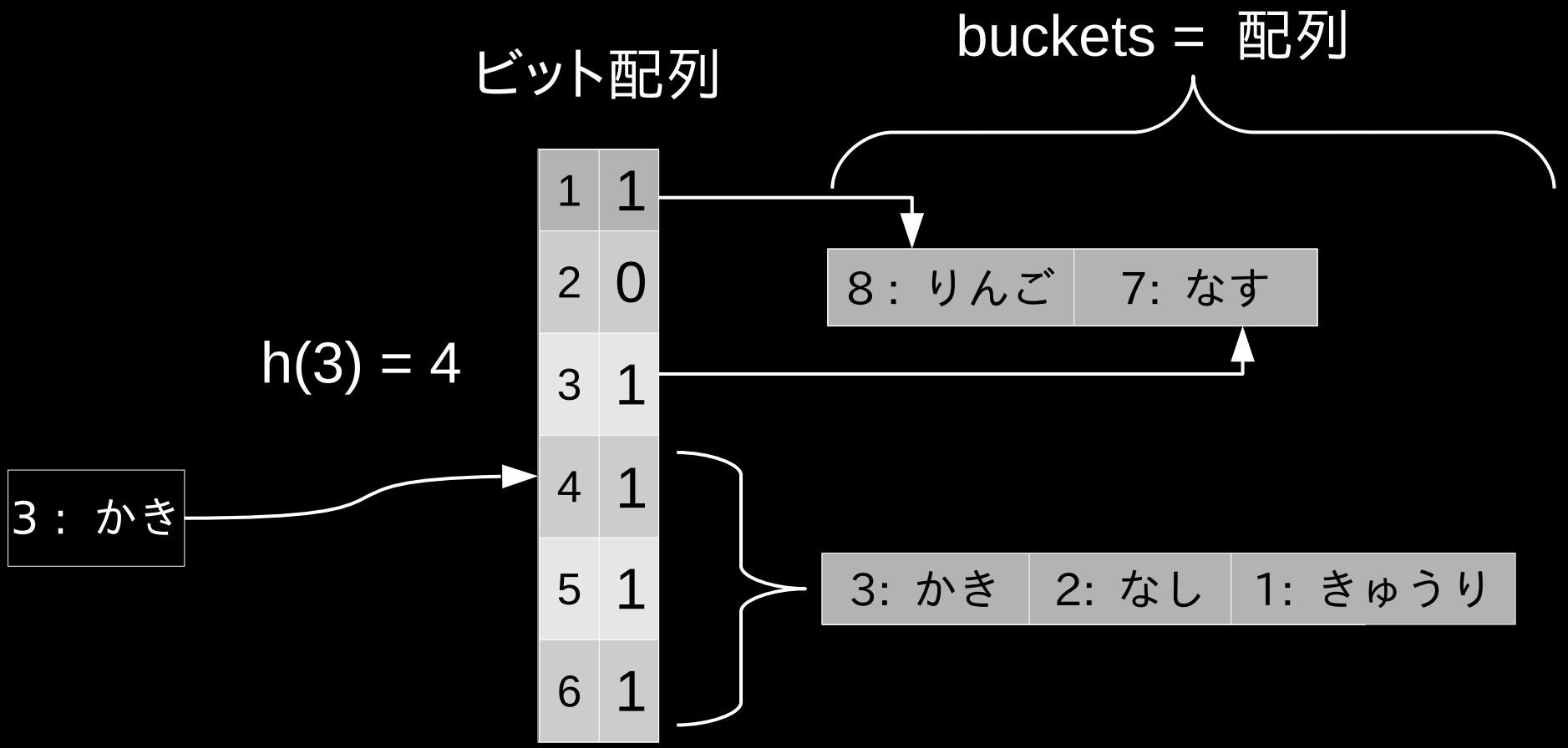
sparse hash table



sparse hash table



sparse hash table



compact hashing

Cleary'84:

- closed hash table
- φ : K → φ(K) 全単射
  - φ(k) = (h(k), r(k))
  - $\varphi^{-1}(h(k),r(k)) = k$
- k の代わりに r(k) を保存したら領域を節約できる

φ(k) = (h(k), r(k))

h(k)	(r(k), value)
1	2: なす
2	1 : りんご
3	
4	3: きゅうり
5	

φ(k) = (h(k), r(k))

φ(5) = (3,2)

5： みかん

h(k)	(r(k), value)
1	2: なす
2	1： りんご
3	
4	3: きゅうり
5	

φ(k) = (h(k), r(k))

φ(5) = (3,2)

5： みかん

h(k)	(r(k), value)
1	2: なす
2	1： りんご
3	2: みかん
4	3: きゅうり
5	

φ(k) = (h(k), r(k))

φ(5) = (3,2)

5： みかん

φ^{-1}(3,2)=5

h(k)	(r(k), value)
1	2: なす
2	1： りんご
3	2: みかん
4	3: きゅうり
5	

Cleary: linear probing

φ(k) = (h(k), r(k))

h(k)	(r(k), value)
1	2: なす
2	1 : りんご
3	2: みかん
4	3: きゅうり
5	

Cleary: linear probing

φ(k) = (h(k), r(k))

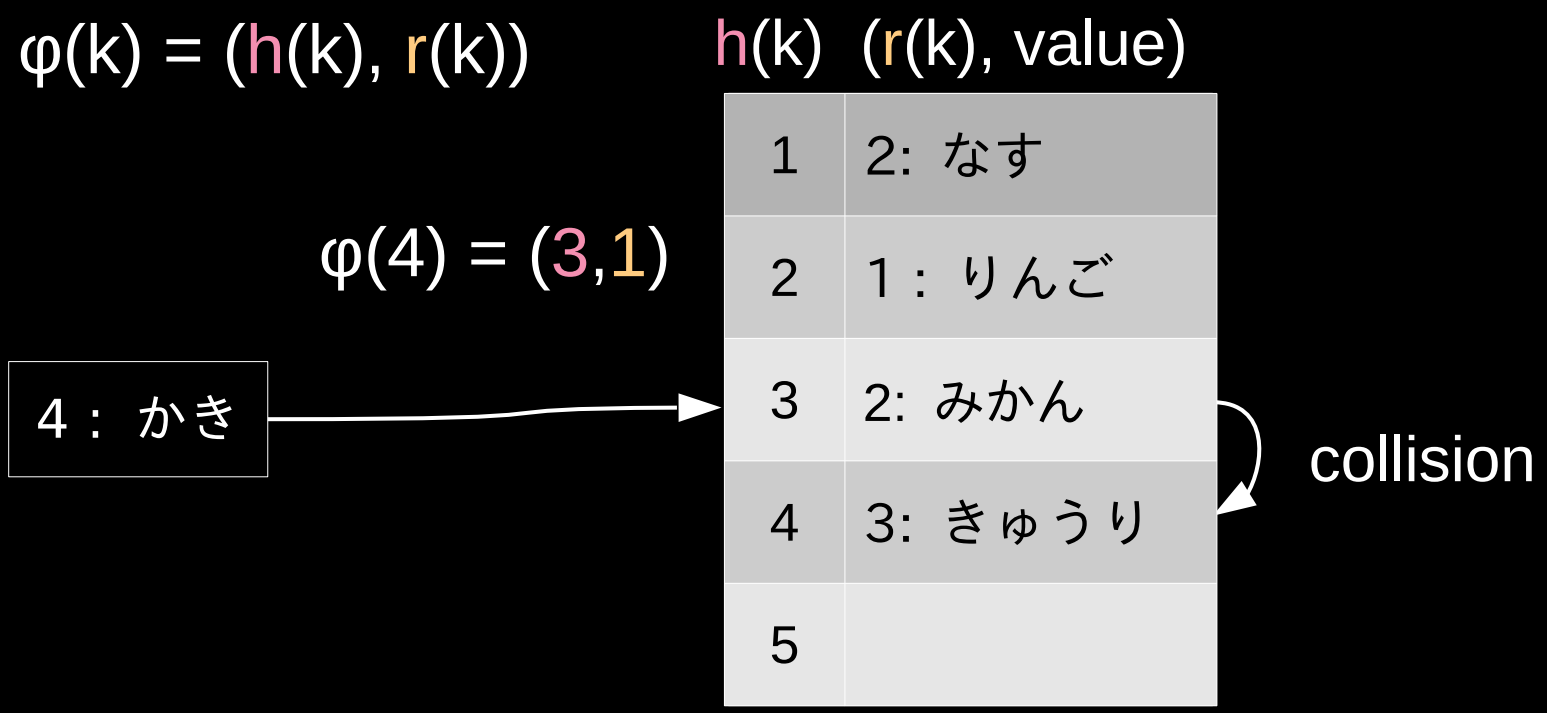
φ(4) = (3,1)

h(k)	(r(k), value)
1	2: なす
2	1：りんご
3	2: みかん
4	3: きゅうり
5	

Cleary: linear probing

φ(k) = (h(k), r(k))

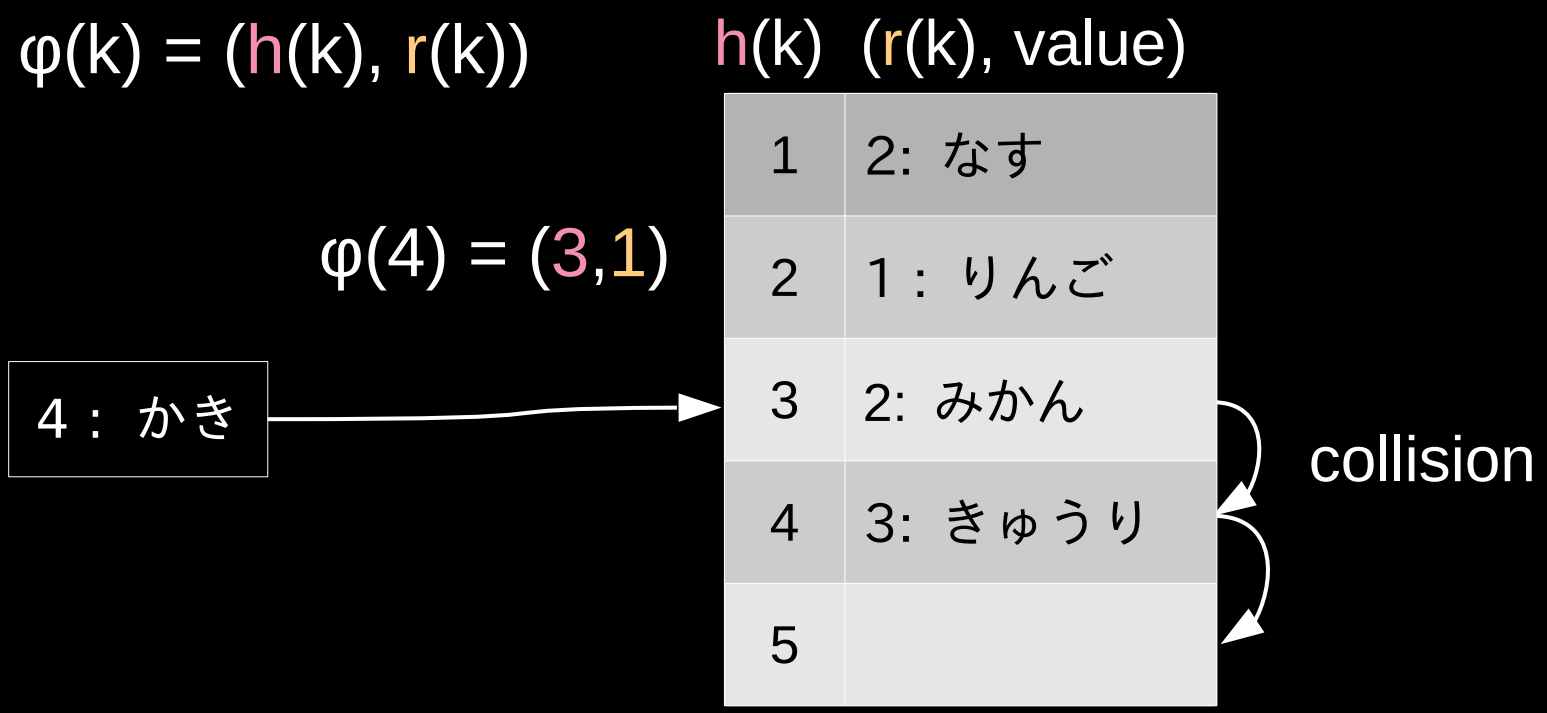
φ(4) = (3,1)



Cleary: linear probing

φ(k) = (h(k), r(k))

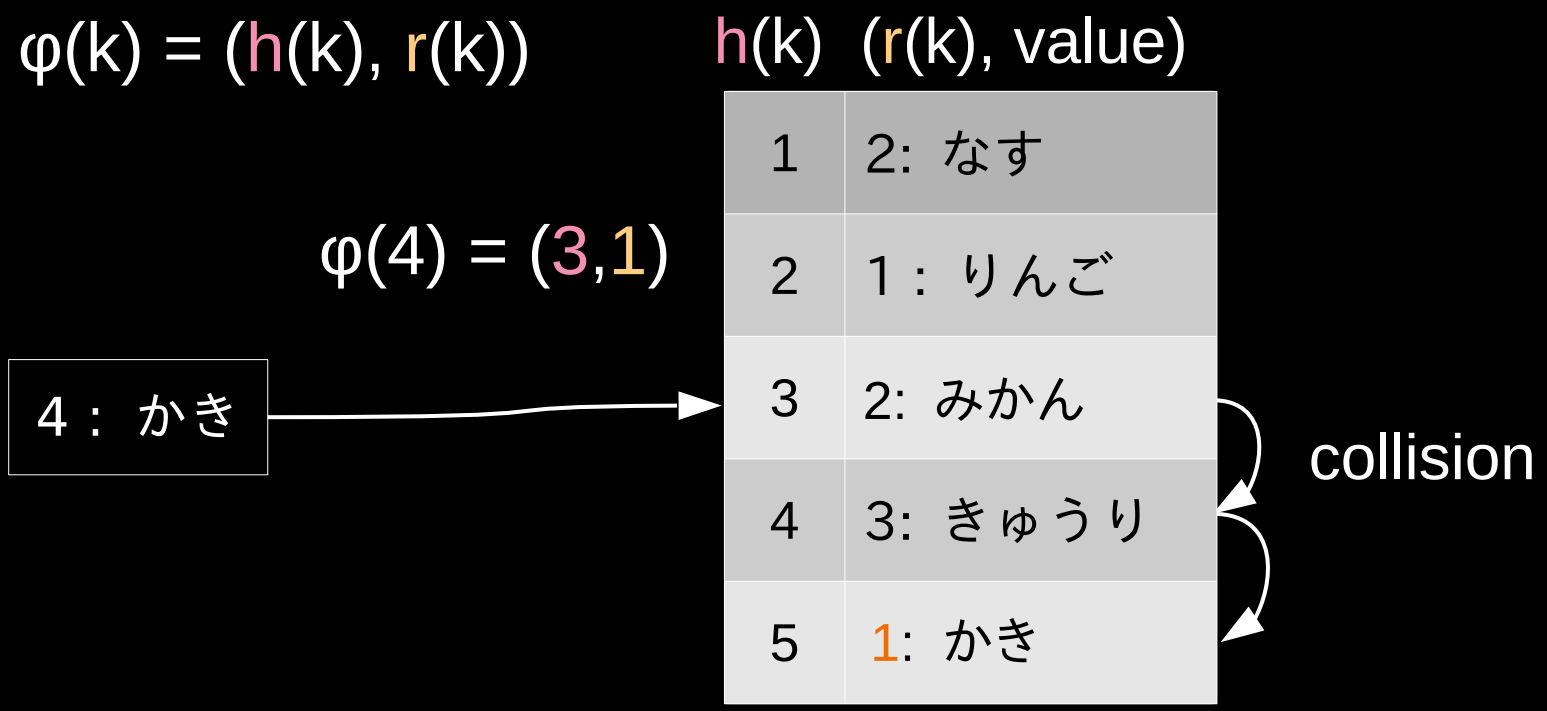
φ(4) = (3,1)



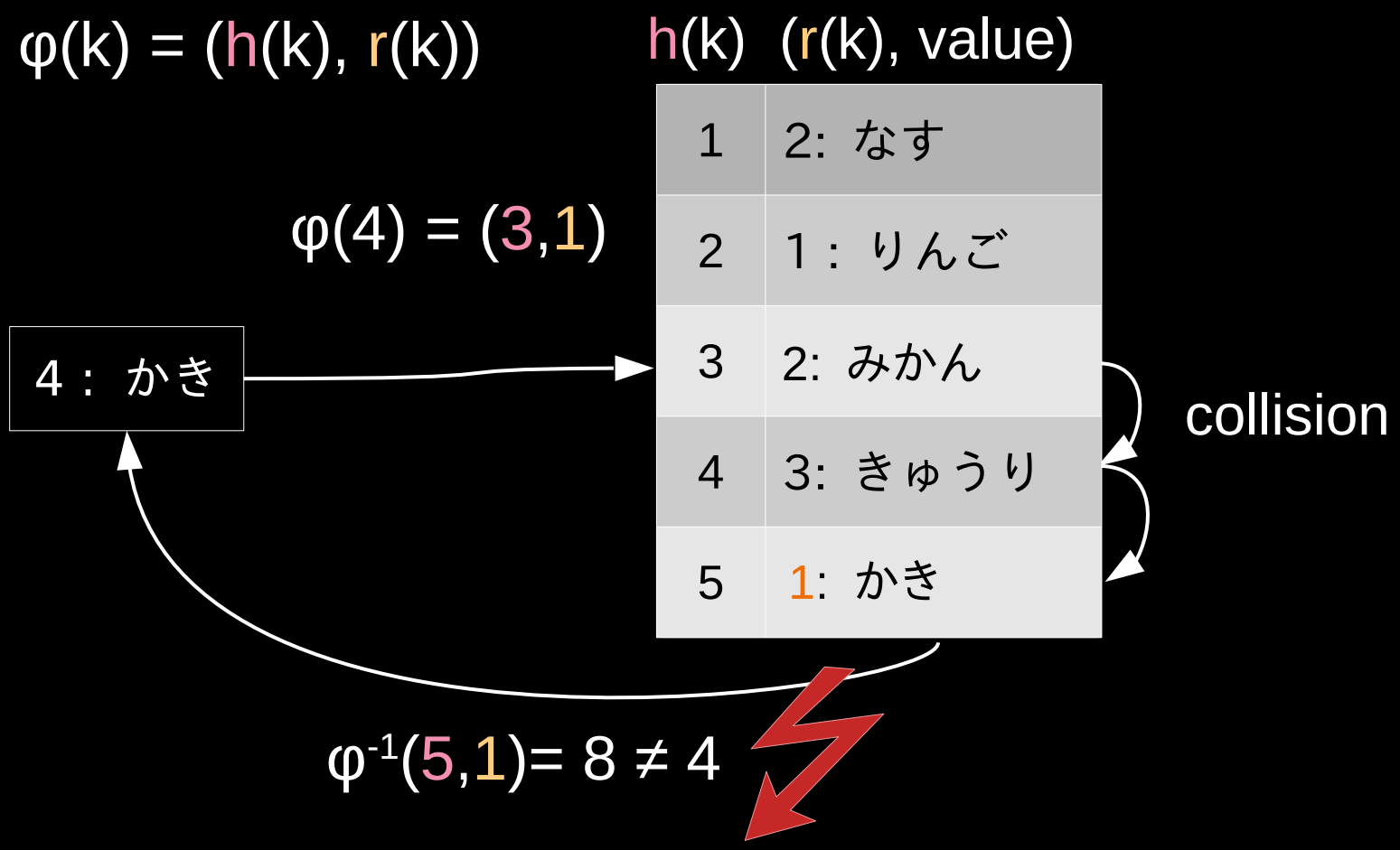
Cleary: linear probing

φ(k) = (h(k), r(k))

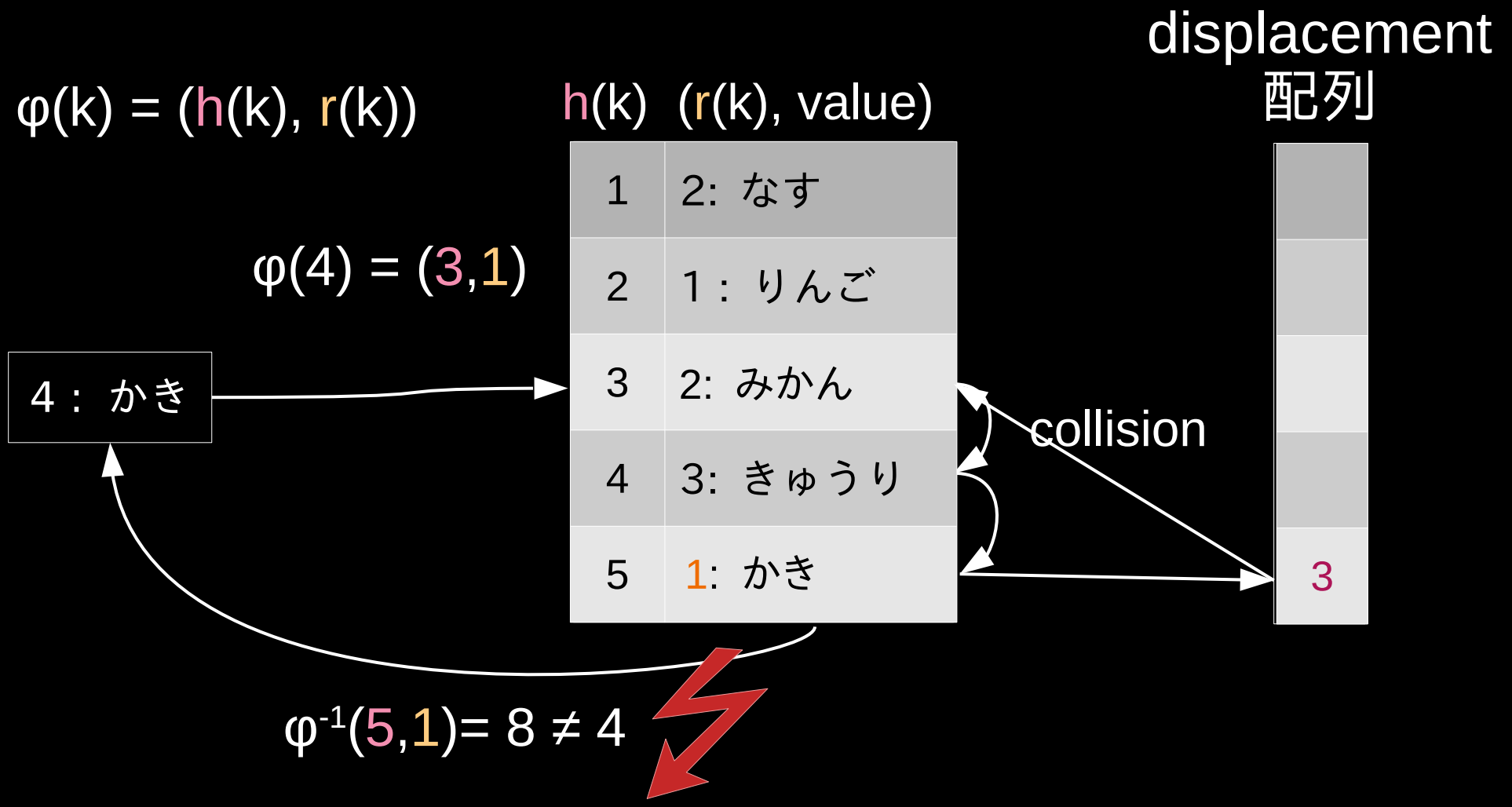
φ(4) = (3,1)



Cleary: linear probing



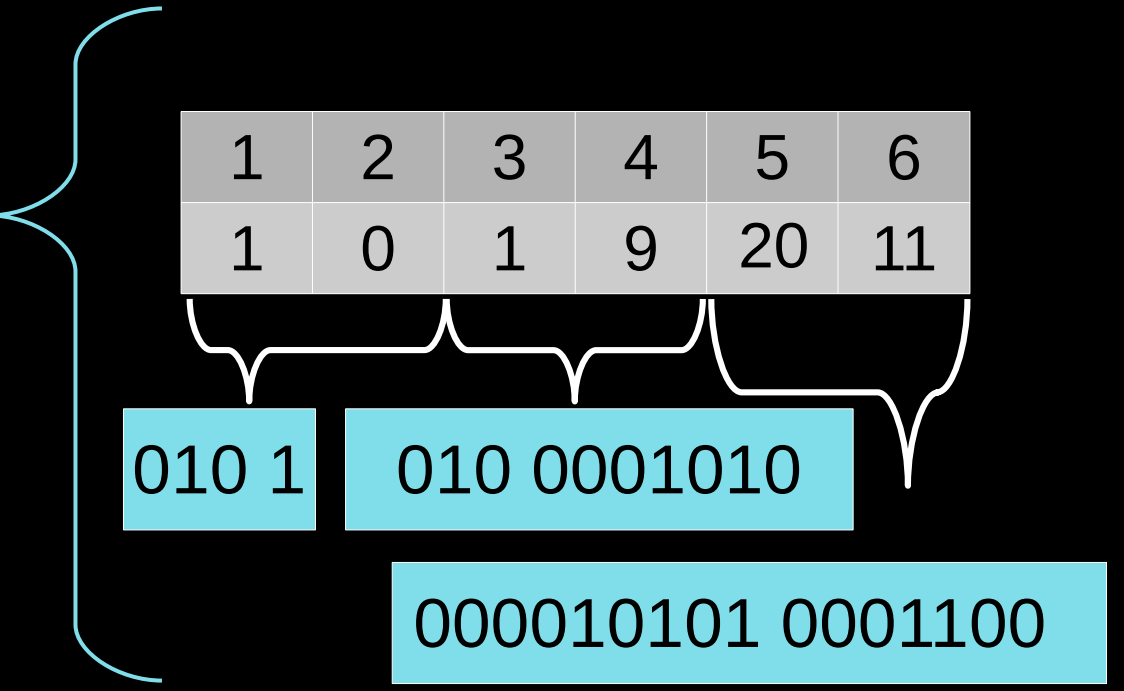
Cleary: linear probing



displacement 配列

表現：

- Cleary '84: 2|H| bits
- Poyias ら '15:
  - Elias γ 符号化
  - layered array



displacement 配列

表現：

- Cleary '84: 2|H| bits
- Poyias ら '15:
  - Elias γ 符号化
  - layered array

4 bit 整数　配列

1	2	3	4	5	6
1	0	1	9		11

displacement 配列

表現：

- Cleary '84: 2|H| bits
- Poyias ら '15:
  - Elias γ 符号化
  - layered array

displacement: 20

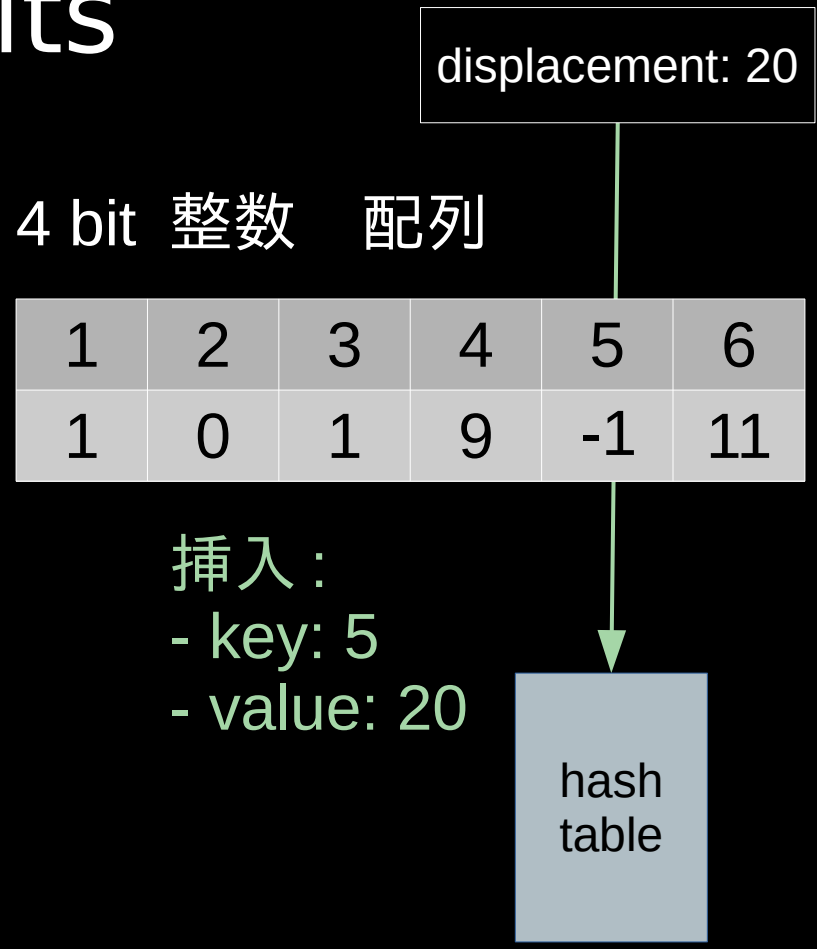
4 bit 整数　配列

1	2	3	4	5	6
1	0	1	9		11

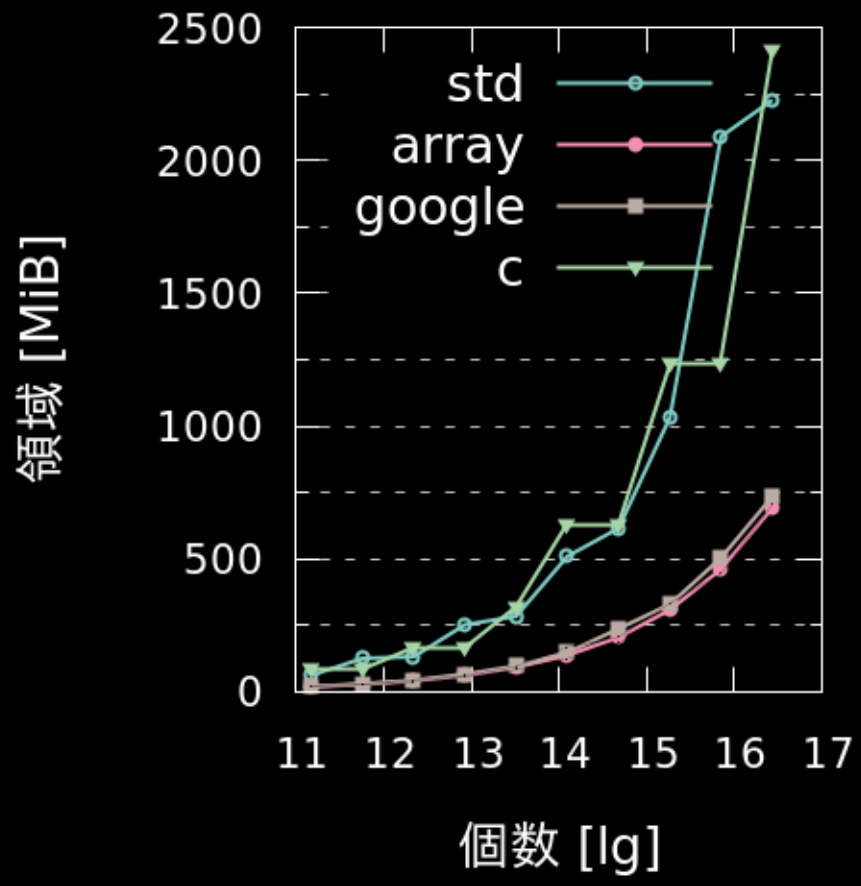
displacement 配列

表現：

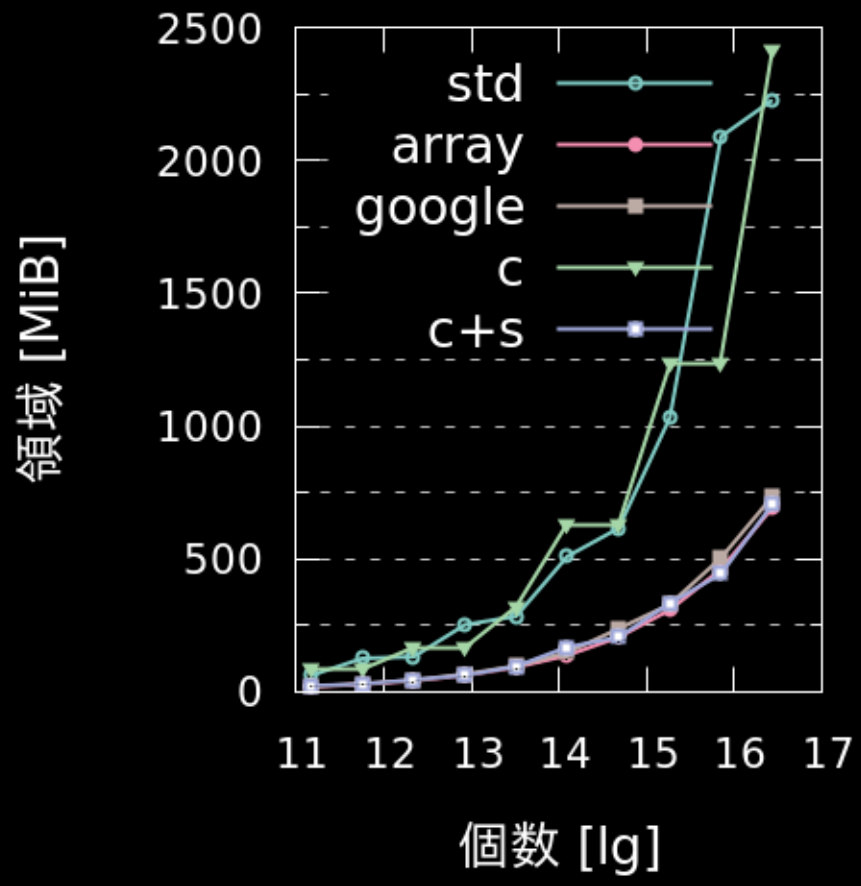
- Cleary '84: 2|H| bits
- Poyias ら '15:
  - Elias γ 符号化
  - layered array



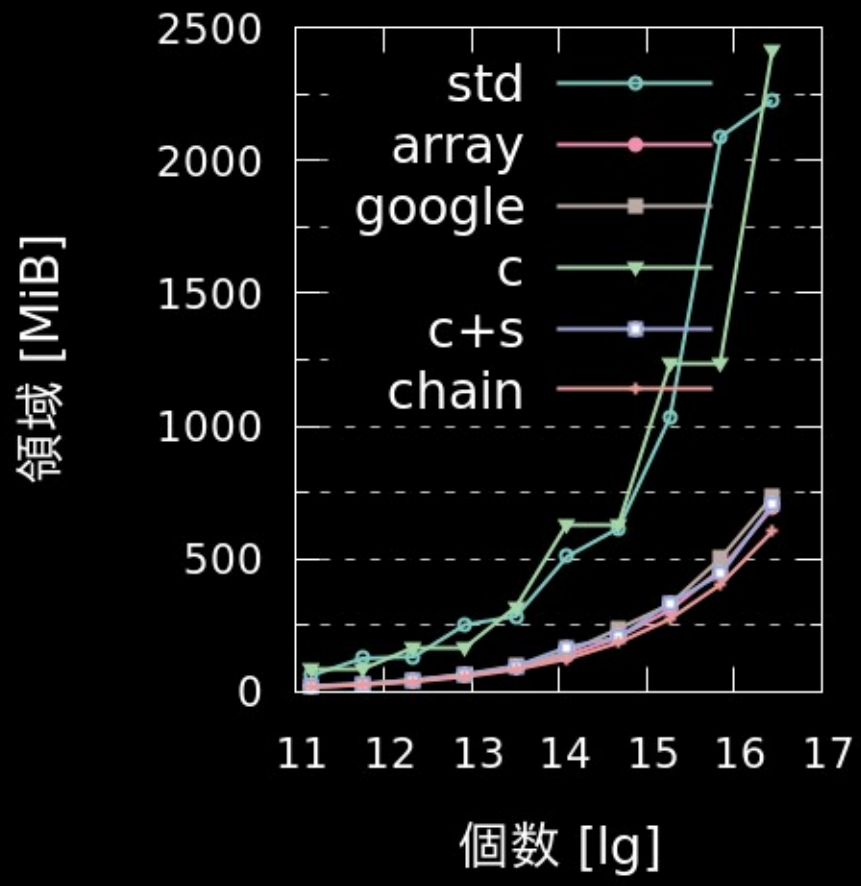
- c: compact
- 領域はがっかり！



- c+s: 合成手法
  - compact と
  - sparse
- array と同程度に小さい

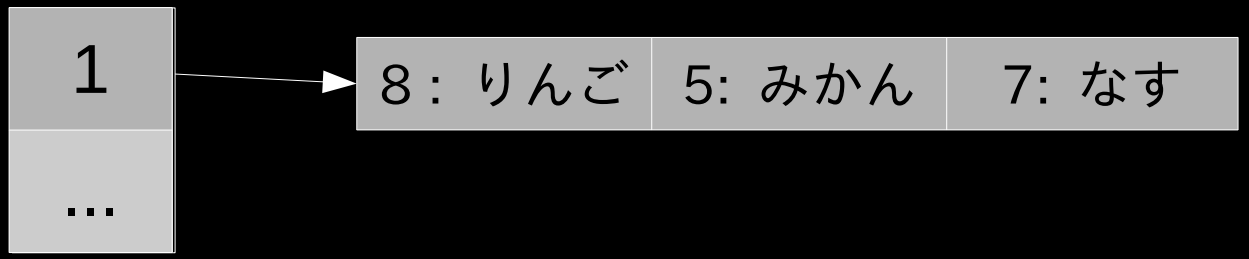


- chain:
  - separate chaining と
  - array と
  - compact
- 一番小さい
- 今回の考案



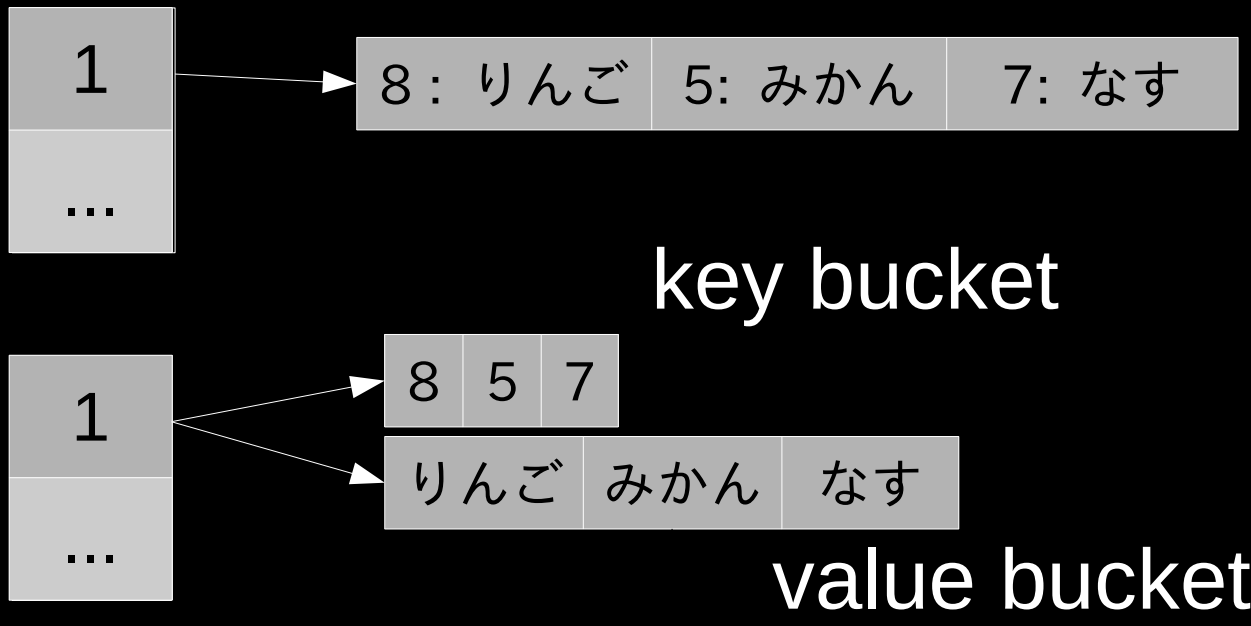
chain

- separate chaining



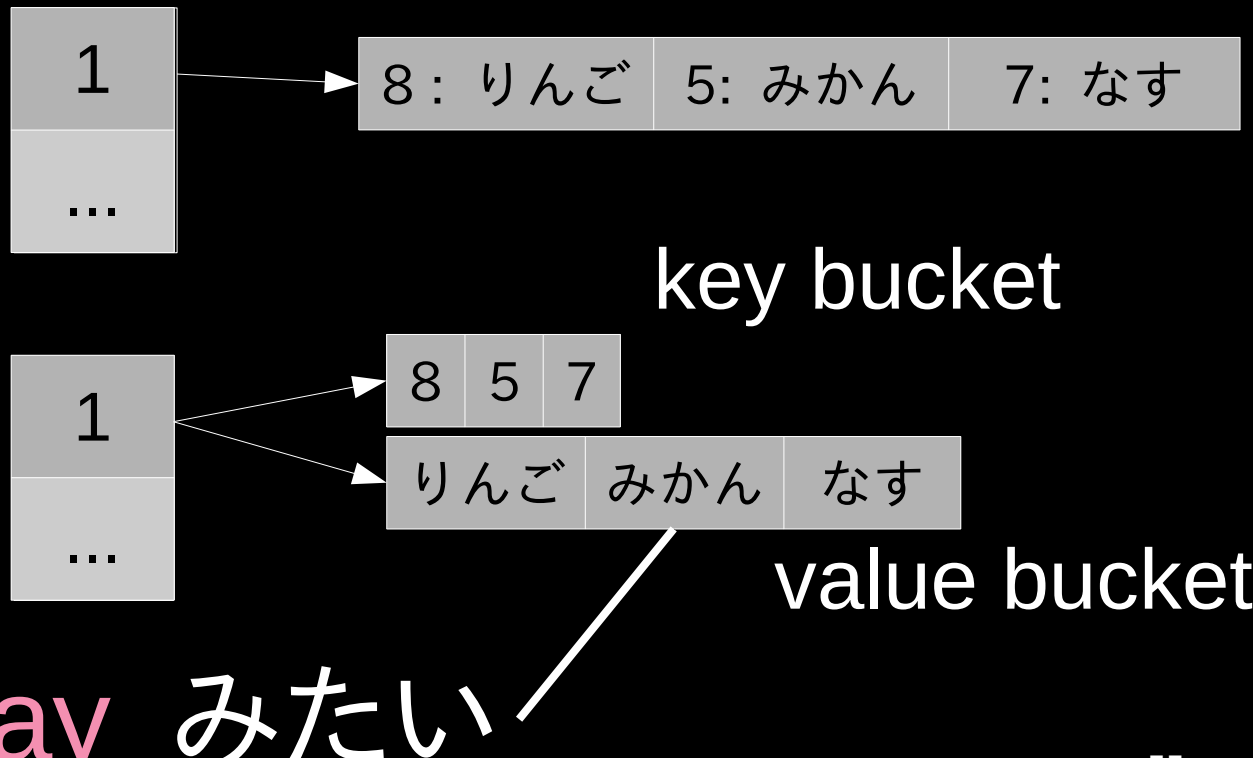
chain

- separate chaining
- list の代わりに２つの配列を使う



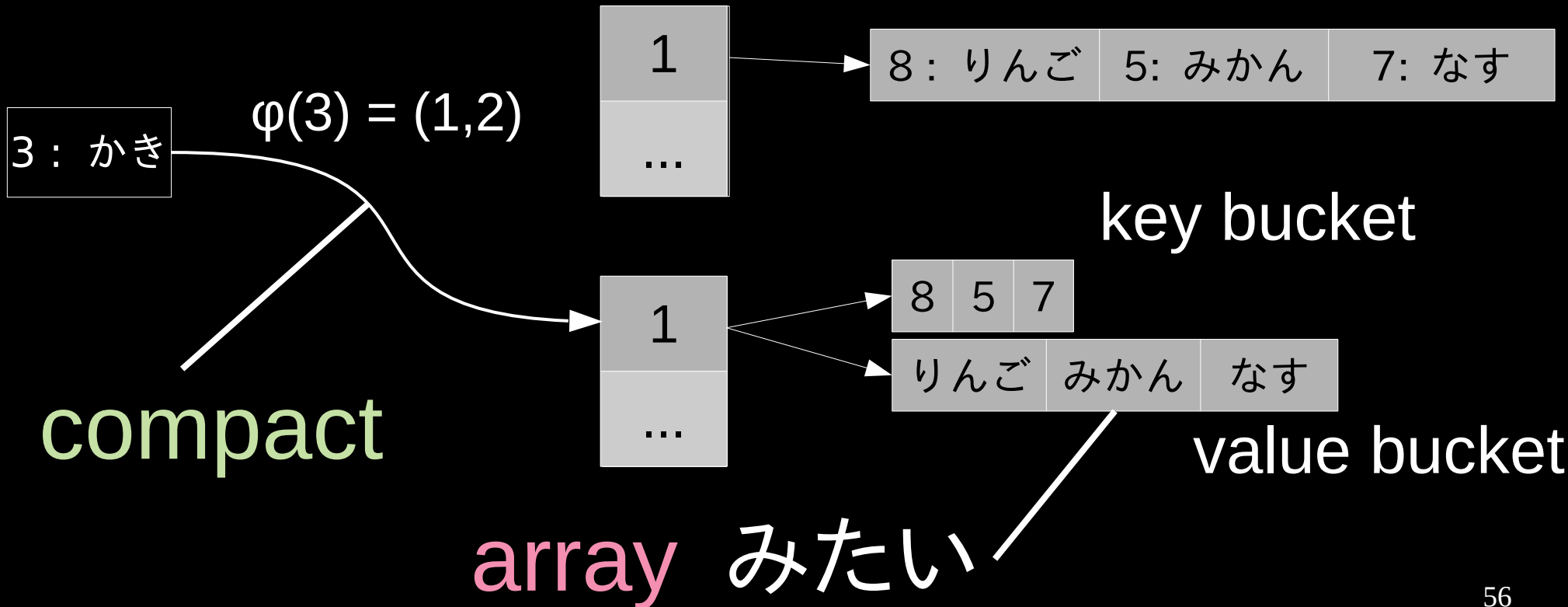
chain

- separate chaining
- list の代わりに２つの配列を使う



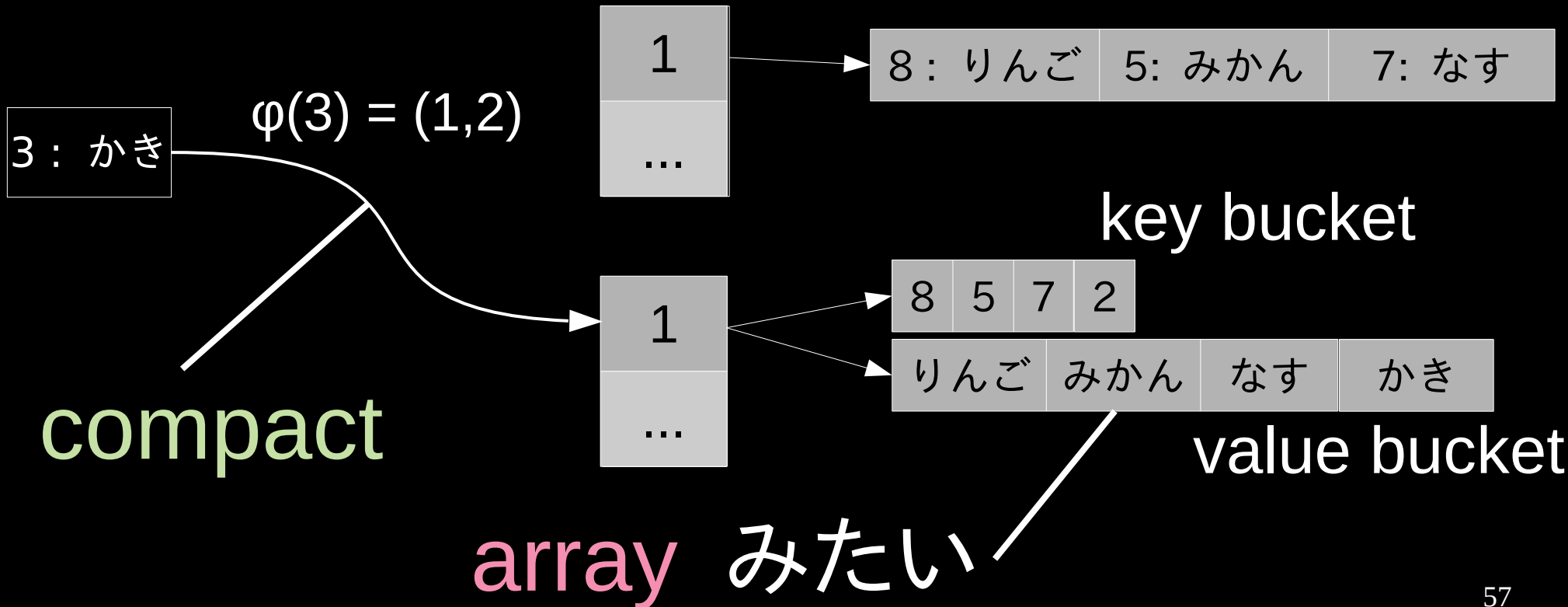
chain

- separate chaining
- list の代わりに２つの配列を使う

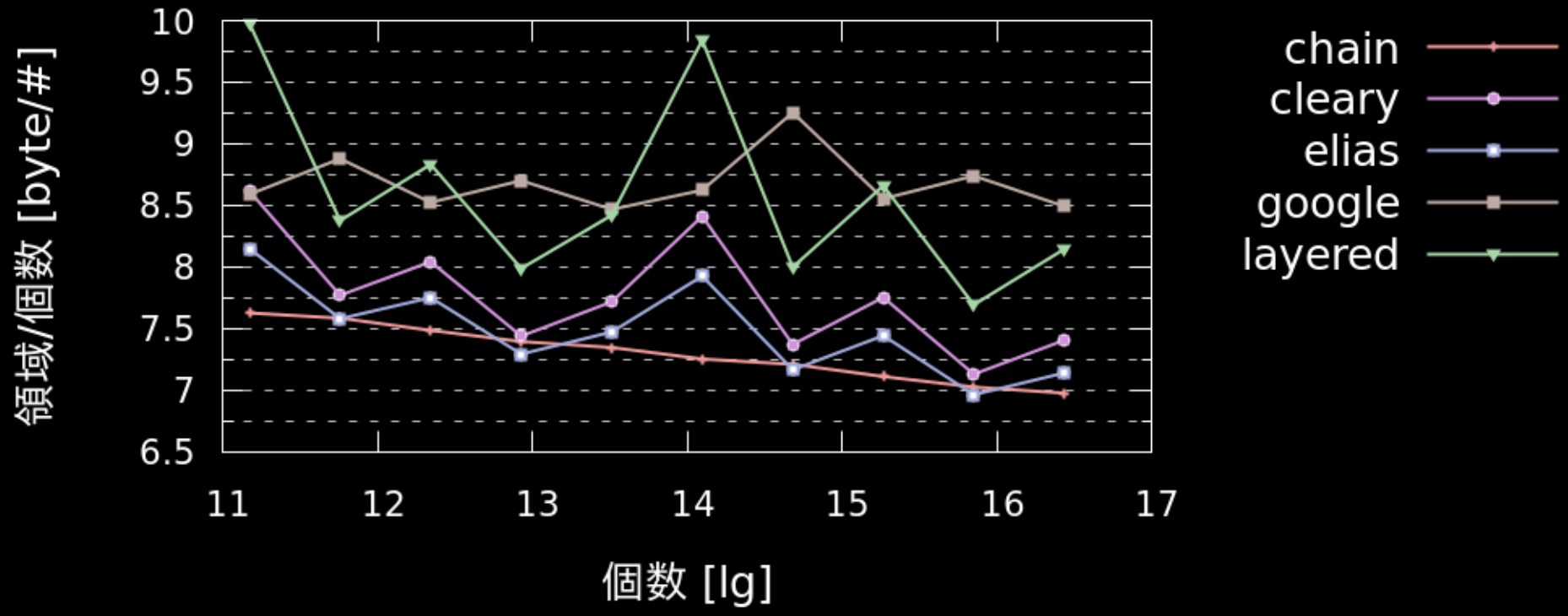


chain

- separate chaining
- list の代わりに２つの配列を使う

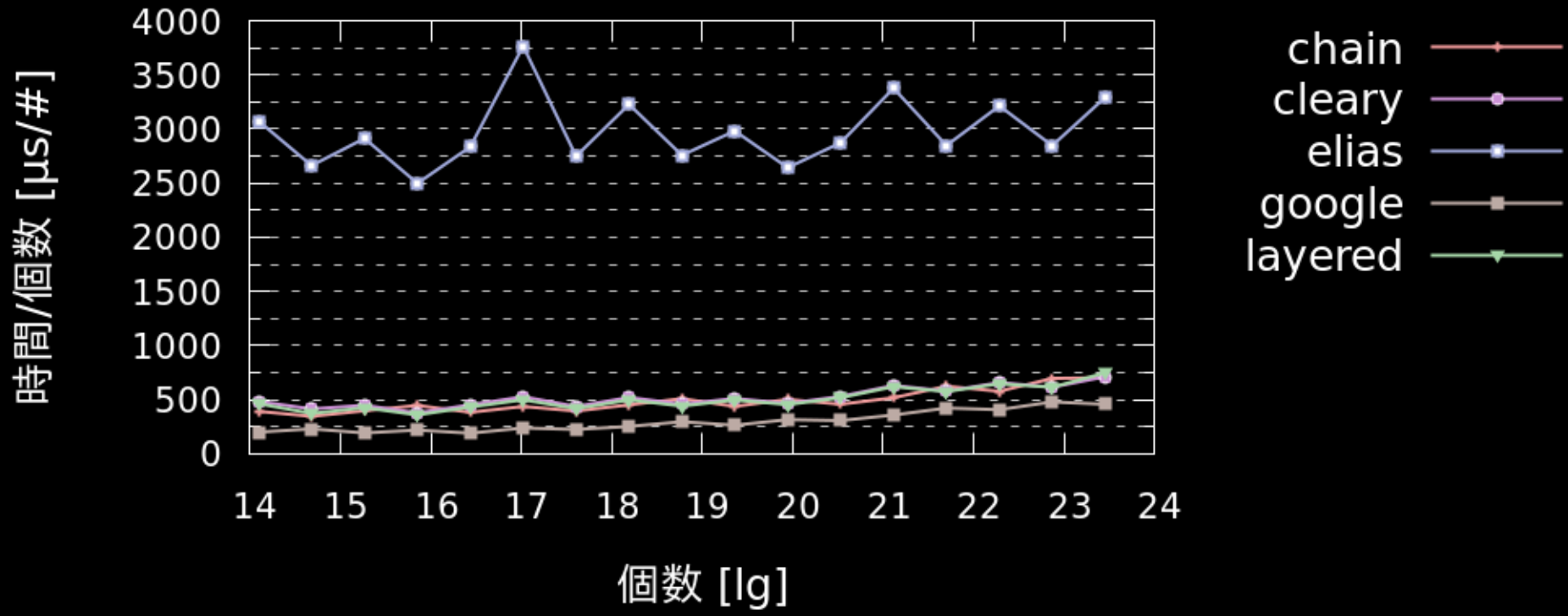


各要素の平均的な領域



- chain の領域は分散が小さい
- 時々、 elias が一番小さい

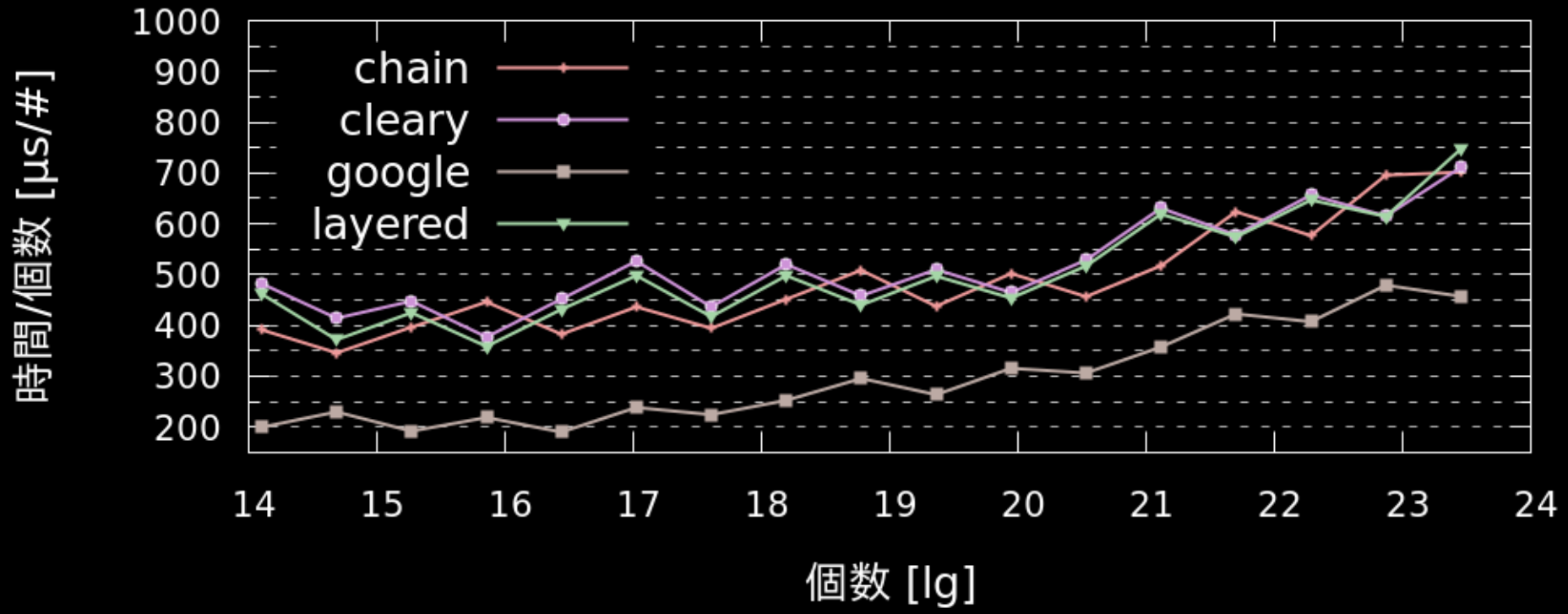
構築時間



- しかし、 elias も一番遅い

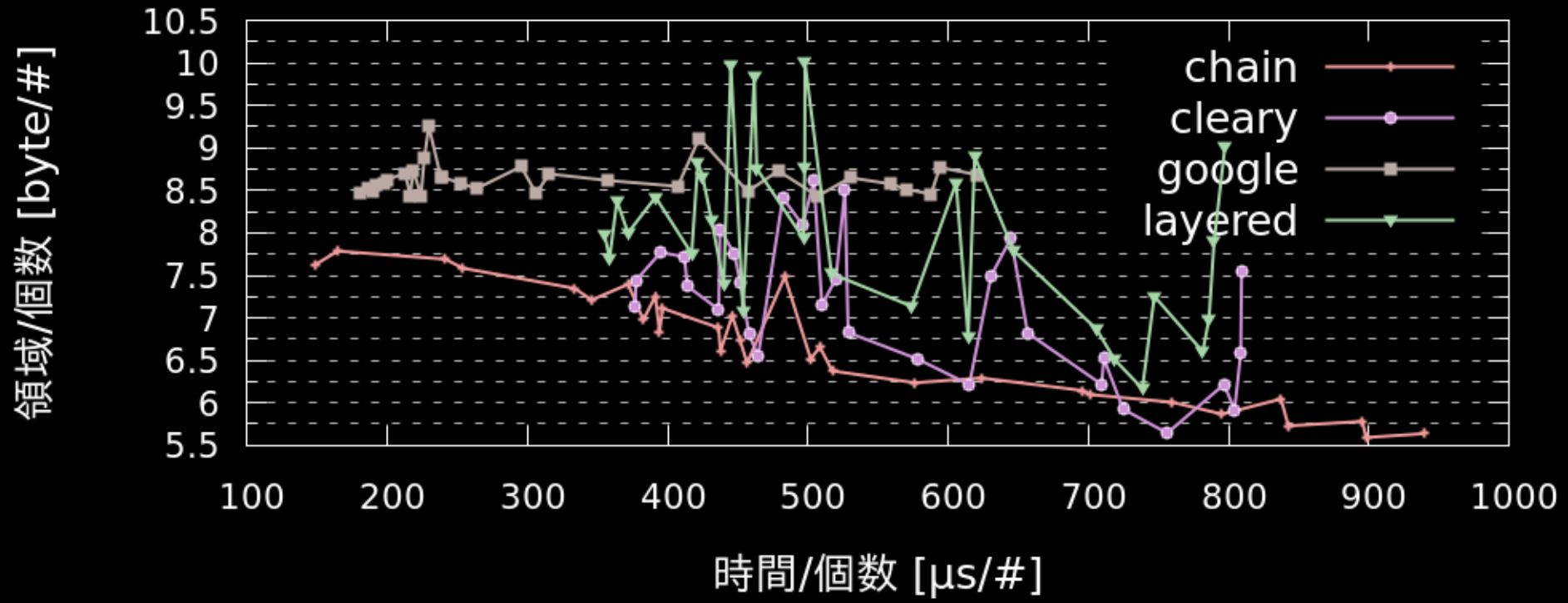
  ⇒ 不便

構築時間



- chain は cleary と layered のように速い

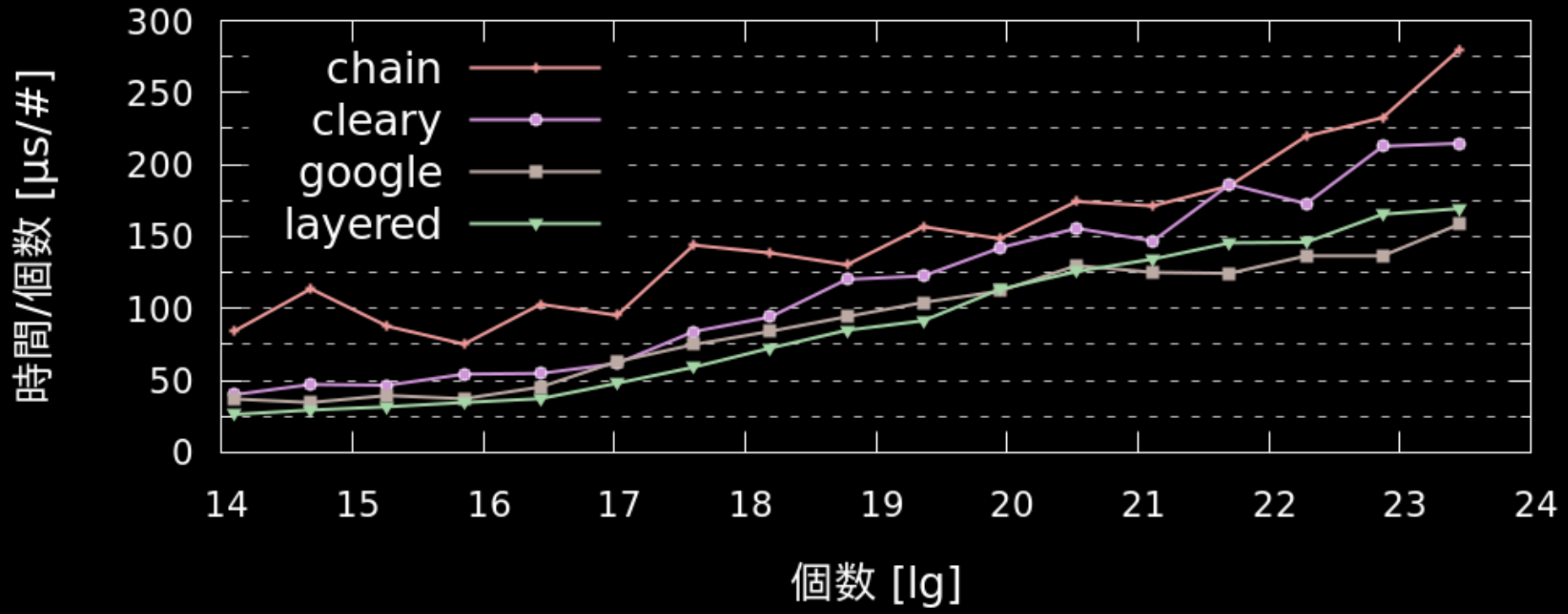
探索：時間 対 領域



- 大体 chain の勝ち

（線は見やすさのため）

探索時間

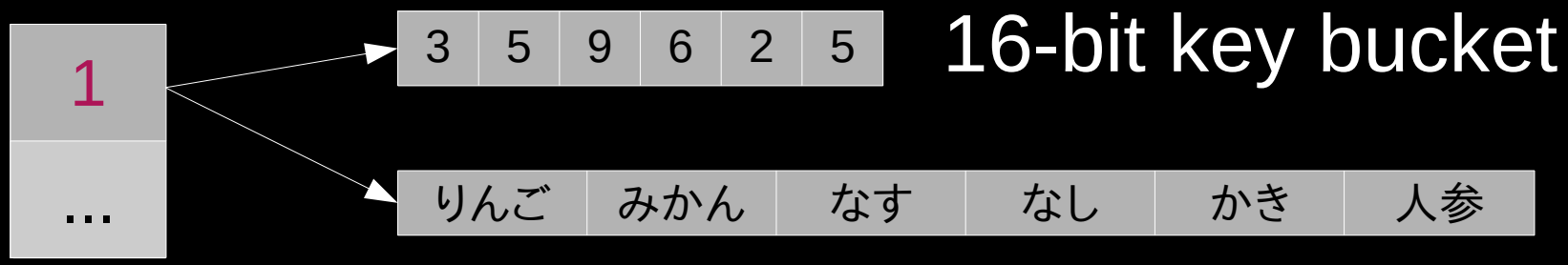


- 255 個の無作為に選んだ key を探す
- 探索は chain の短所

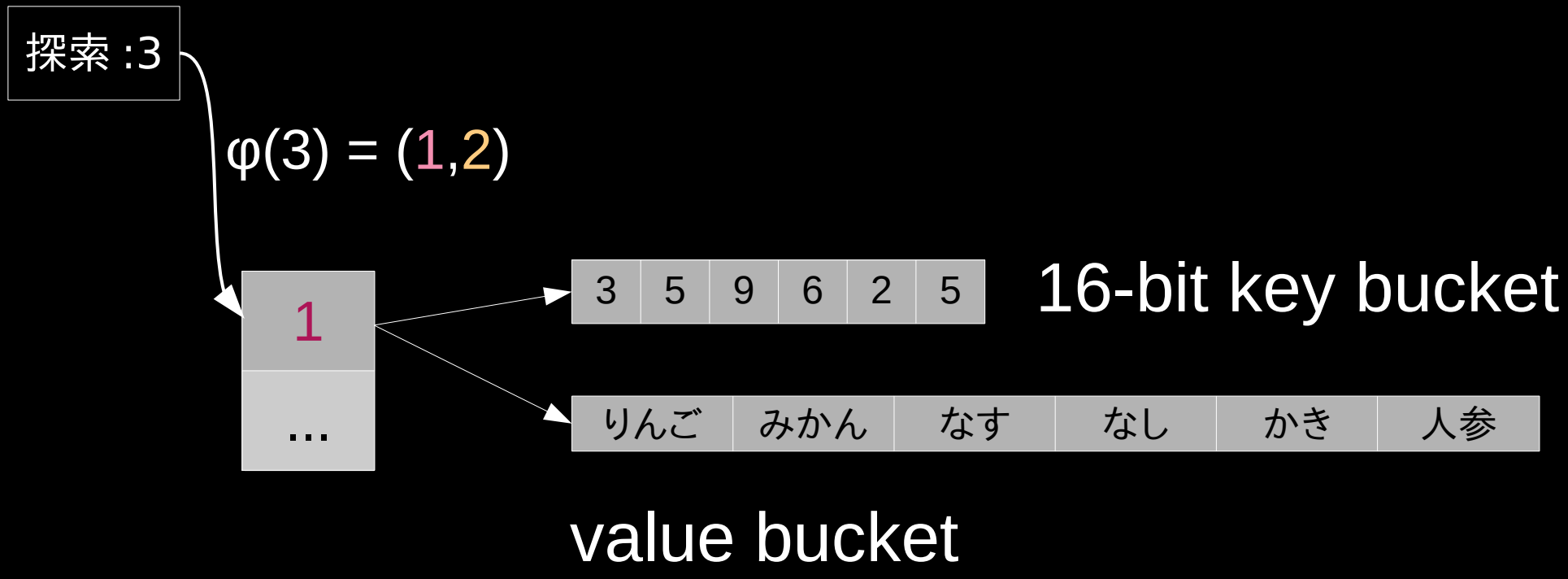
設定の変更

- 入力： 32 bit key + 32 bit value
- 今まで：
  - hash table は入力の個数を知らない
- 今から：
  - 入力が x 個なら、 $x\text{-}2^{16}$ の buckets の領域を確保する
  - 32 bit key の k の代わりに 16 bit の $r(k)$ だけ保存すればよい

avx

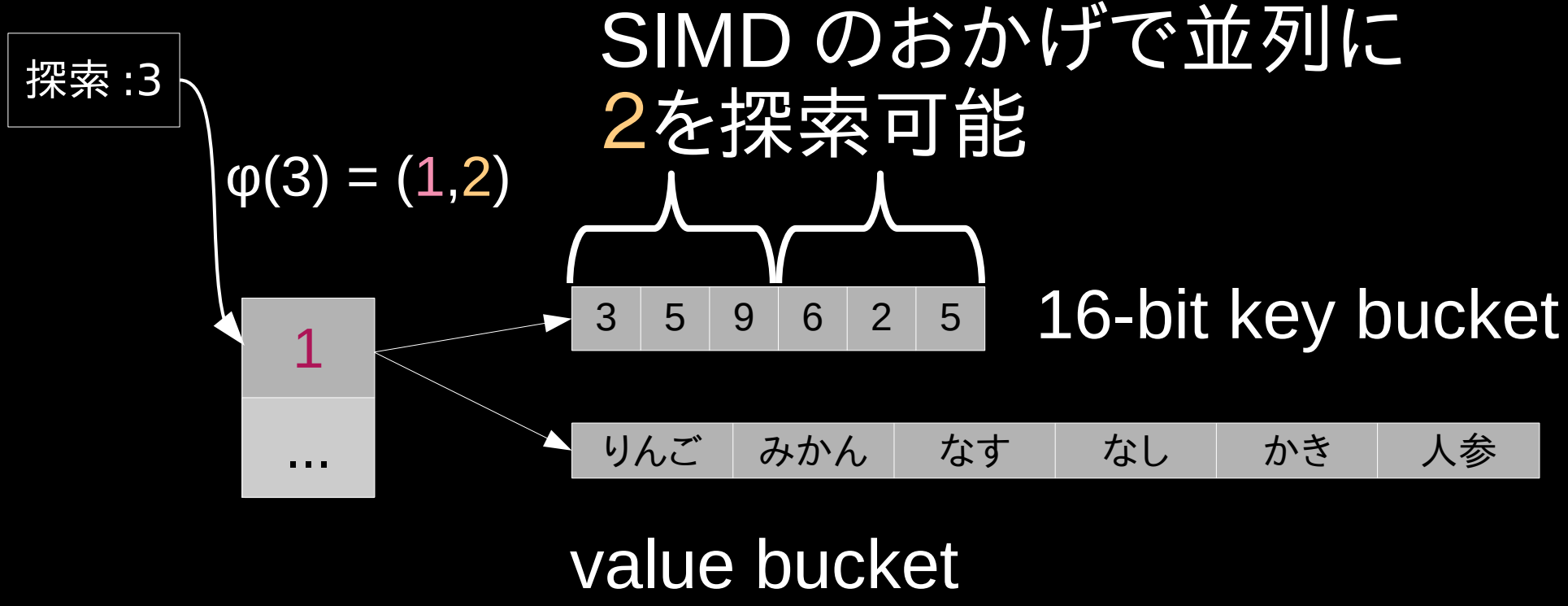


avx

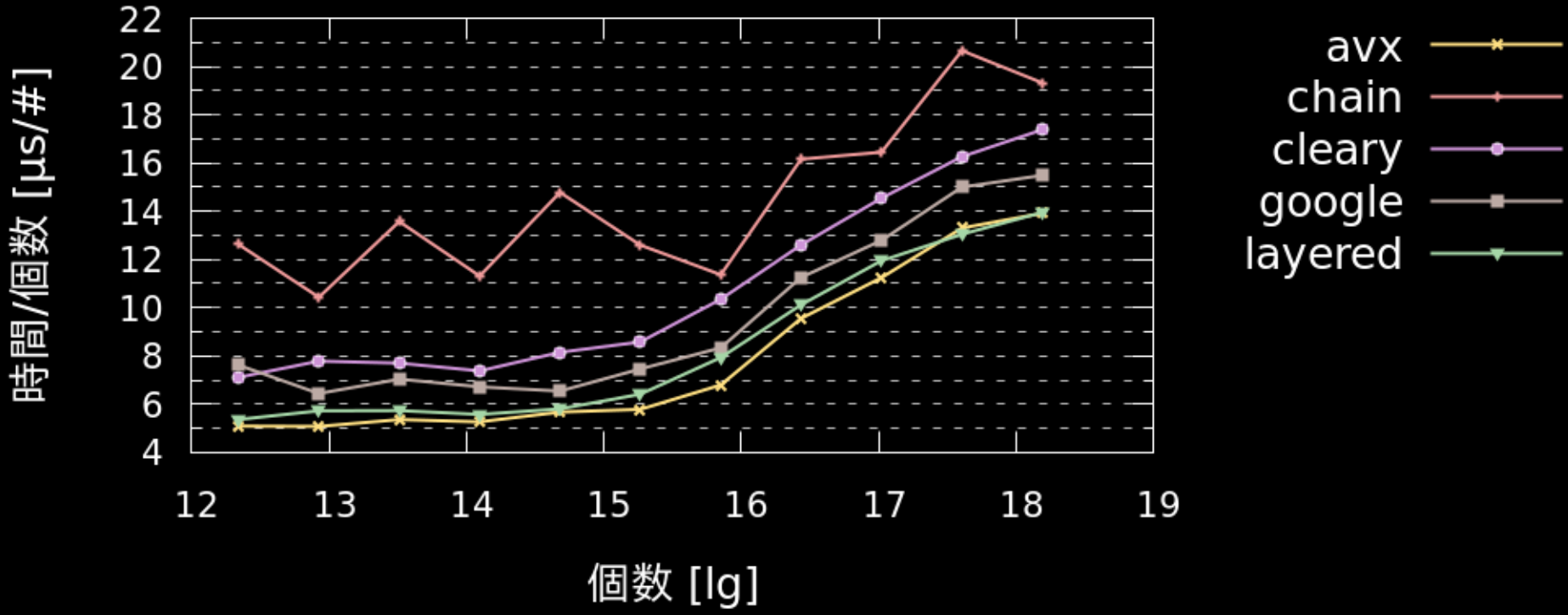

探索 :3
φ(3) = (1,2)
1
...
3 5 9 6 2 5
16-bit key bucket
りんご みかん なす なし かき 人参
value bucket

avx

avx 2 : SIMD 命令セット

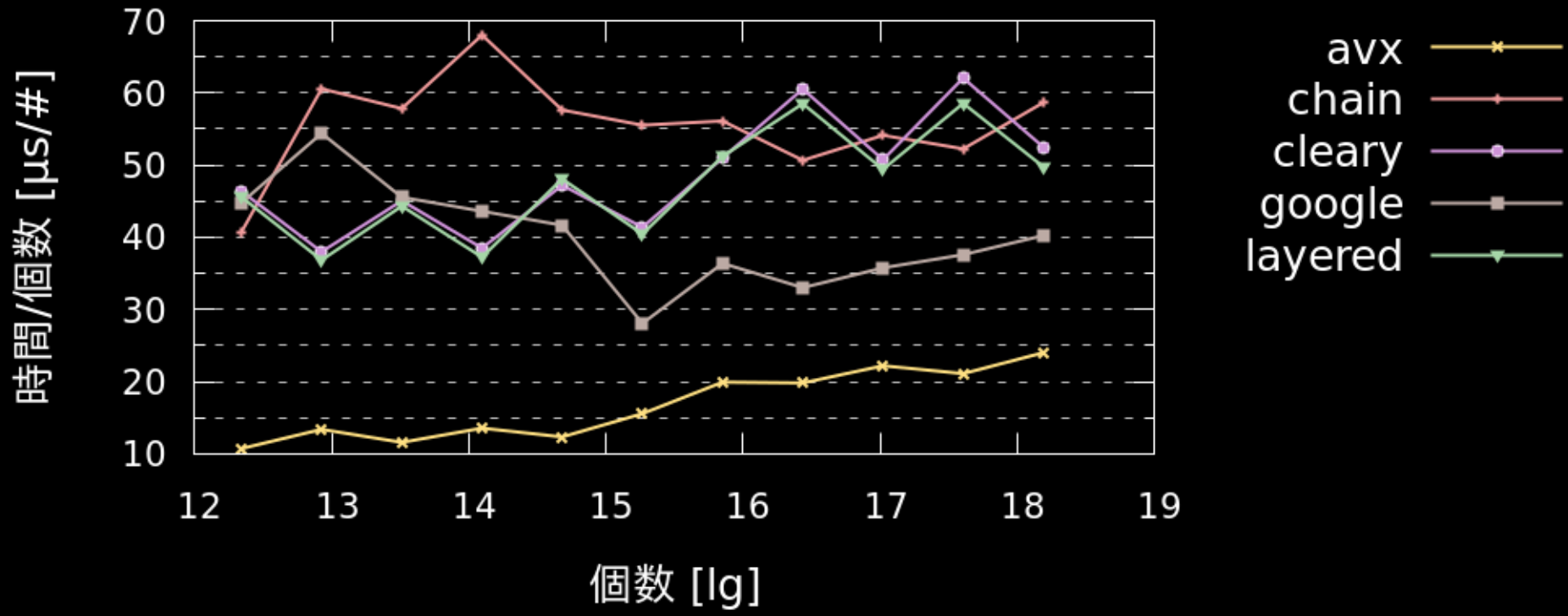


探索時間



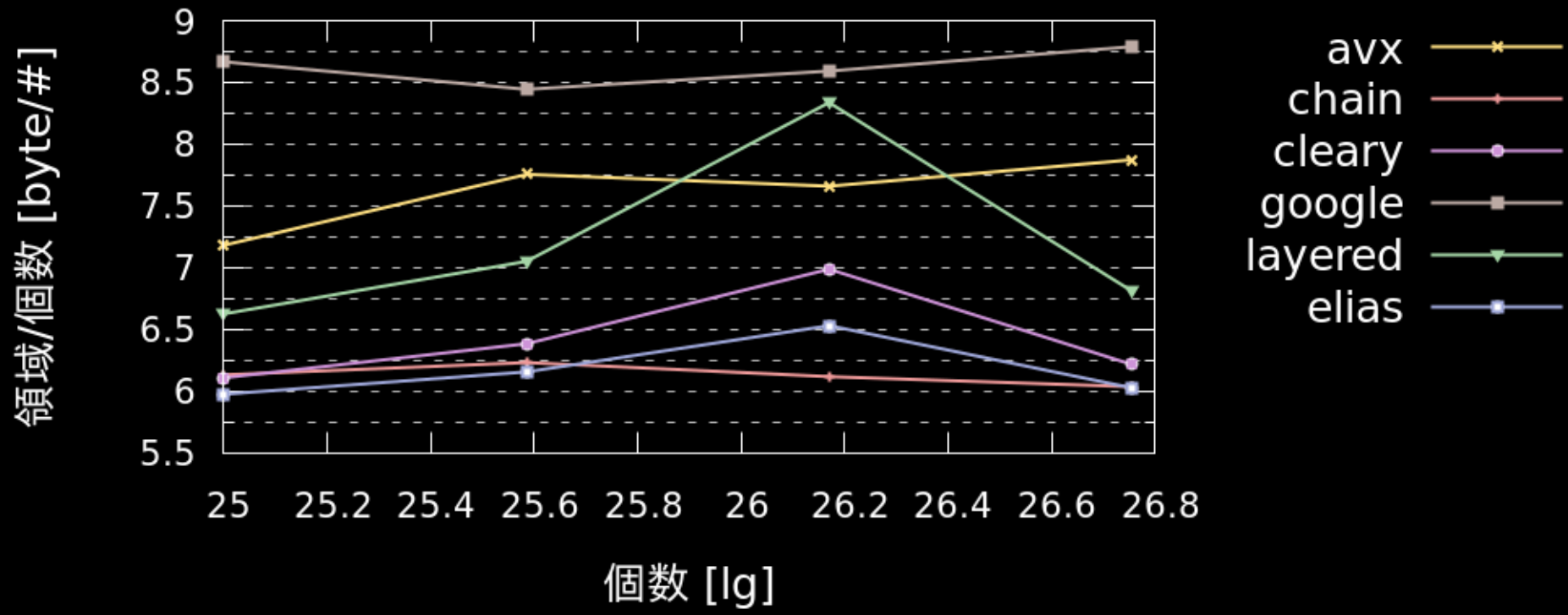
- avx は一番速い

構築時間



- avx は一番速い

領域



- avx は最良ではない

まとめ

	構築		検索
hash table	**領域を節約**	**時間**	**時間**
google	悪い	速い	速い
cleary	普通	普通	遅い
elias	一番良い	超遅い	超遅い
layered	普通	普通	速い
chain	良い	普通	遅い
avx	普通	一番速い	一番速い

入力の概数を知っている場合

まとめ

新しい hash table を提案

- 既存手法:
  - separate chaining
  - bucket は array の表現がある　[Askitis'09]
  - displacement なし compact hashing [Cleary'84]
- 特徴:
  - 領域は小さい
  - 構築は遅くない
  - 検索は遅い
- 入力の数を知っている場合:
  - 探索を SIMD で早くする [Ross'07]
  - 領域を節約する hash table の中で一番速い方法