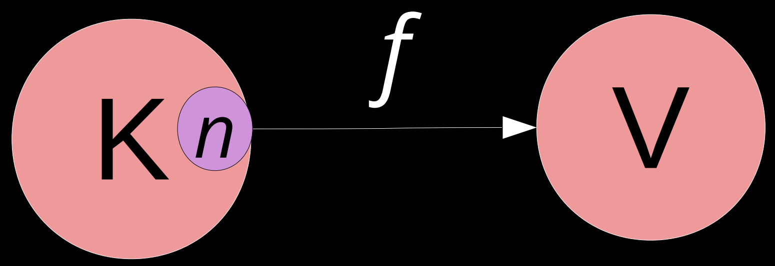
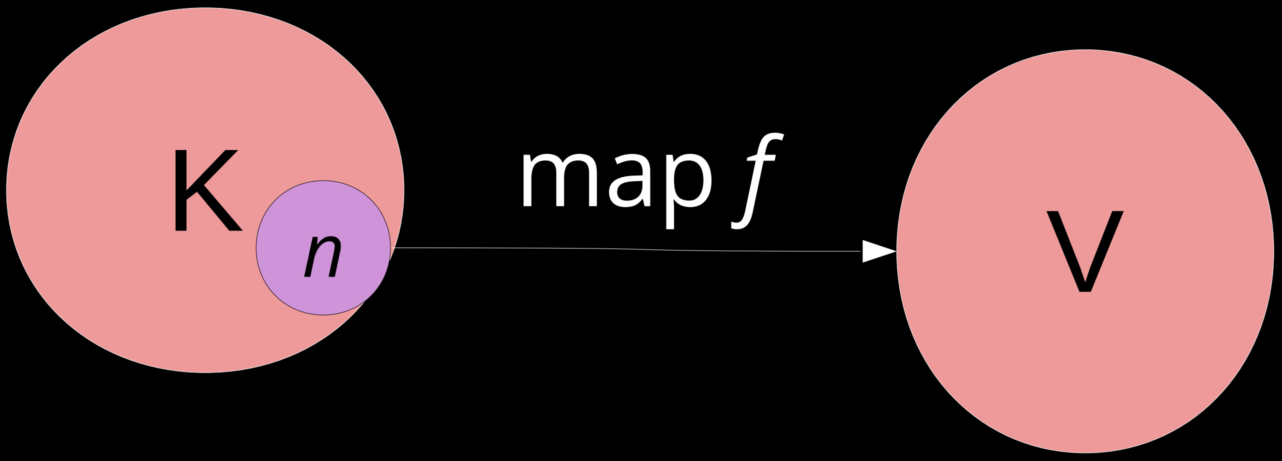
Fast and Simple Compact Hashing via Bucketing

Dominik Köppl
Simon J. Puglisi
Rajeev Raman



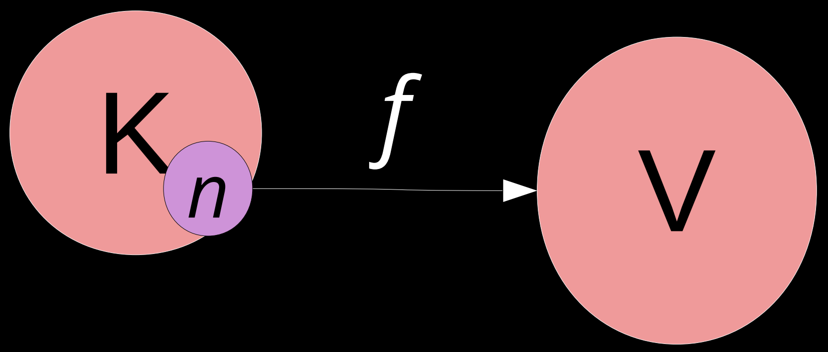
dynamic associative map



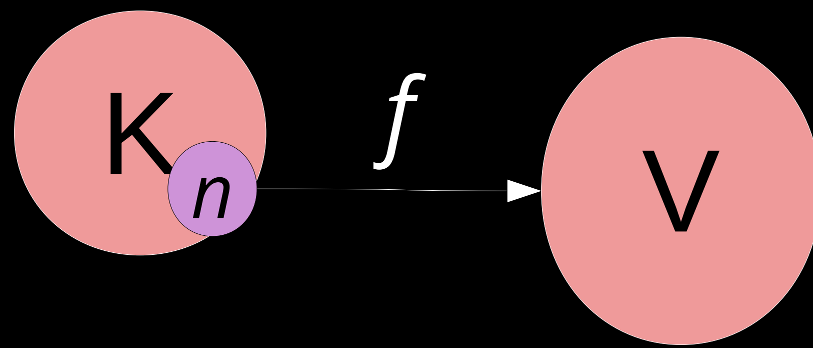
- K, V: sets
- *f* maps a *dynamic* subset of size *n* of K to V
- common representations of *f*
  - search tree
  - hash table

setting

- $K = [1..|2^\omega|]$
- $V = [1..|V|]$



setting

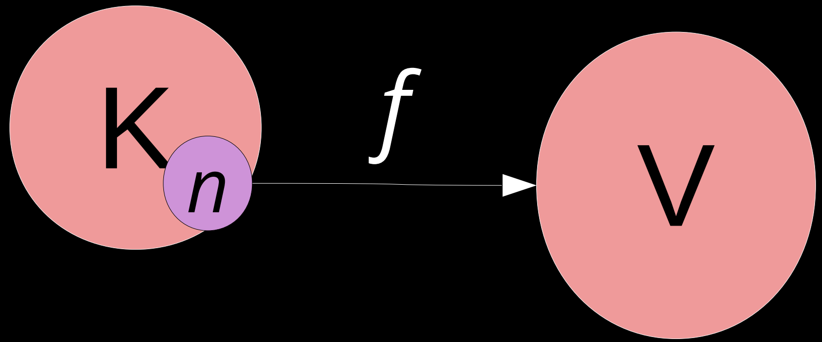


- $K = [1..|2^{\omega}|]$
- $V = [1..|V|]$
- in case that $\omega \leq 20$
  - use plain array to represent f
  - space: $\lg |V|/8$ MiB

MiB = 1024^2

- for larger ω not feasible

setting



- K = [1..|2^ω|]
- V = [1..|V|]
- in case that ω ≤ 20
  - use plain array to represent *f*
  - space: lg |V|/8 MiB
- for larger ω not feasible

MiB = 1024^2

example:

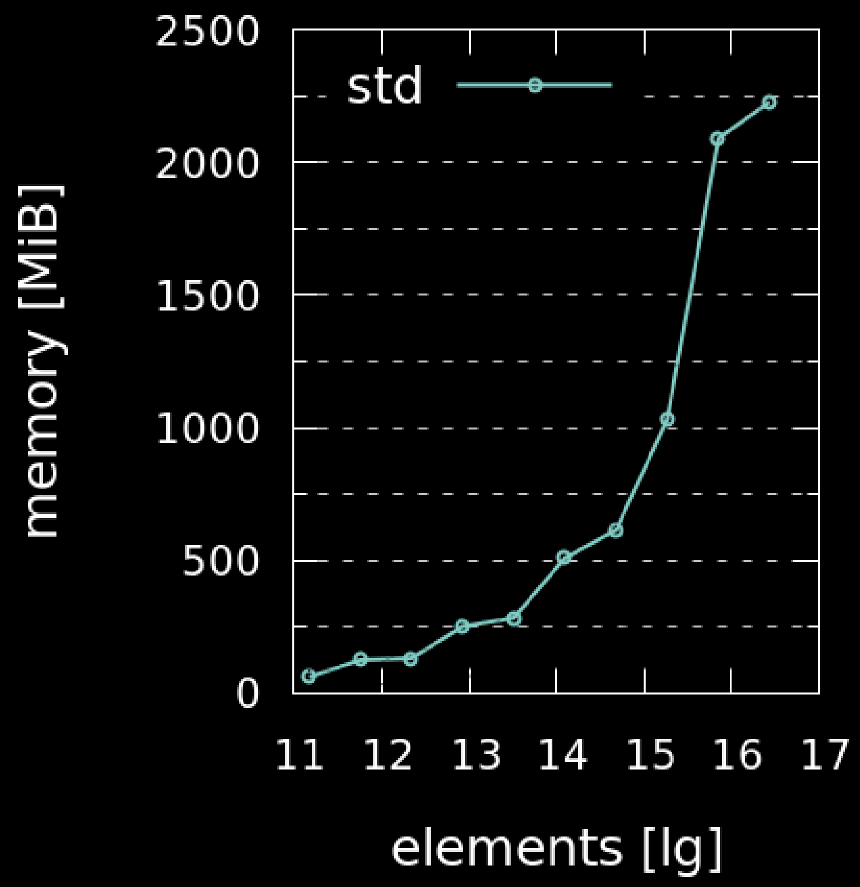
- $|K| = 2^{32}$
- $|V| = 2^{32}$

memory benchmark

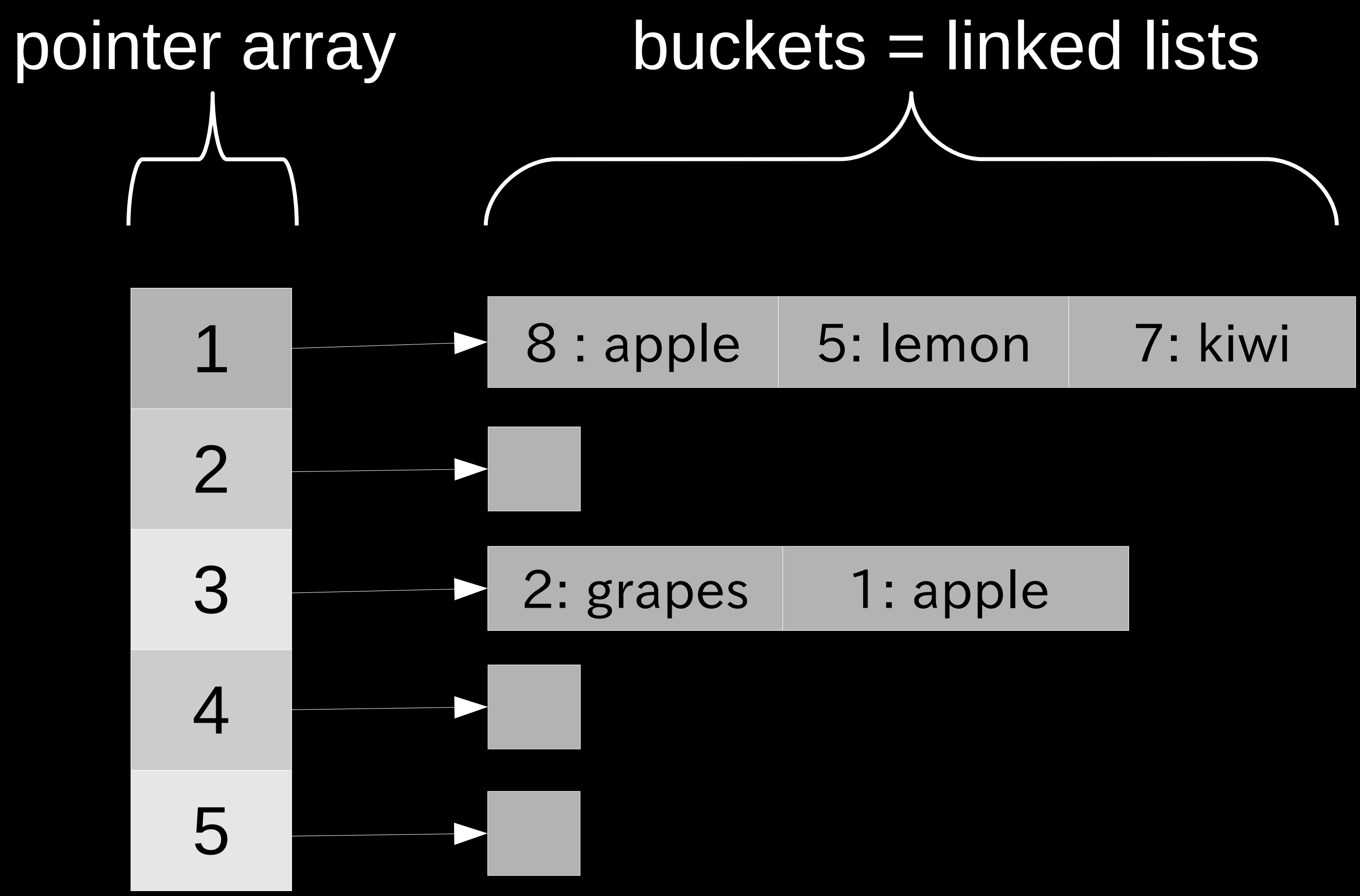
- setting :
  - 32 bit keys
  - 32 bit values
  - randomly generated

memory benchmark

- setting :
  - 32 bit keys
  - 32 bit values
  - randomly generated
- std: C++ STL hash table 「`unordered_map`」
  - closed addressing
  - $n = 2^{16} = 65536$: more than 2 GiB RAM needed!

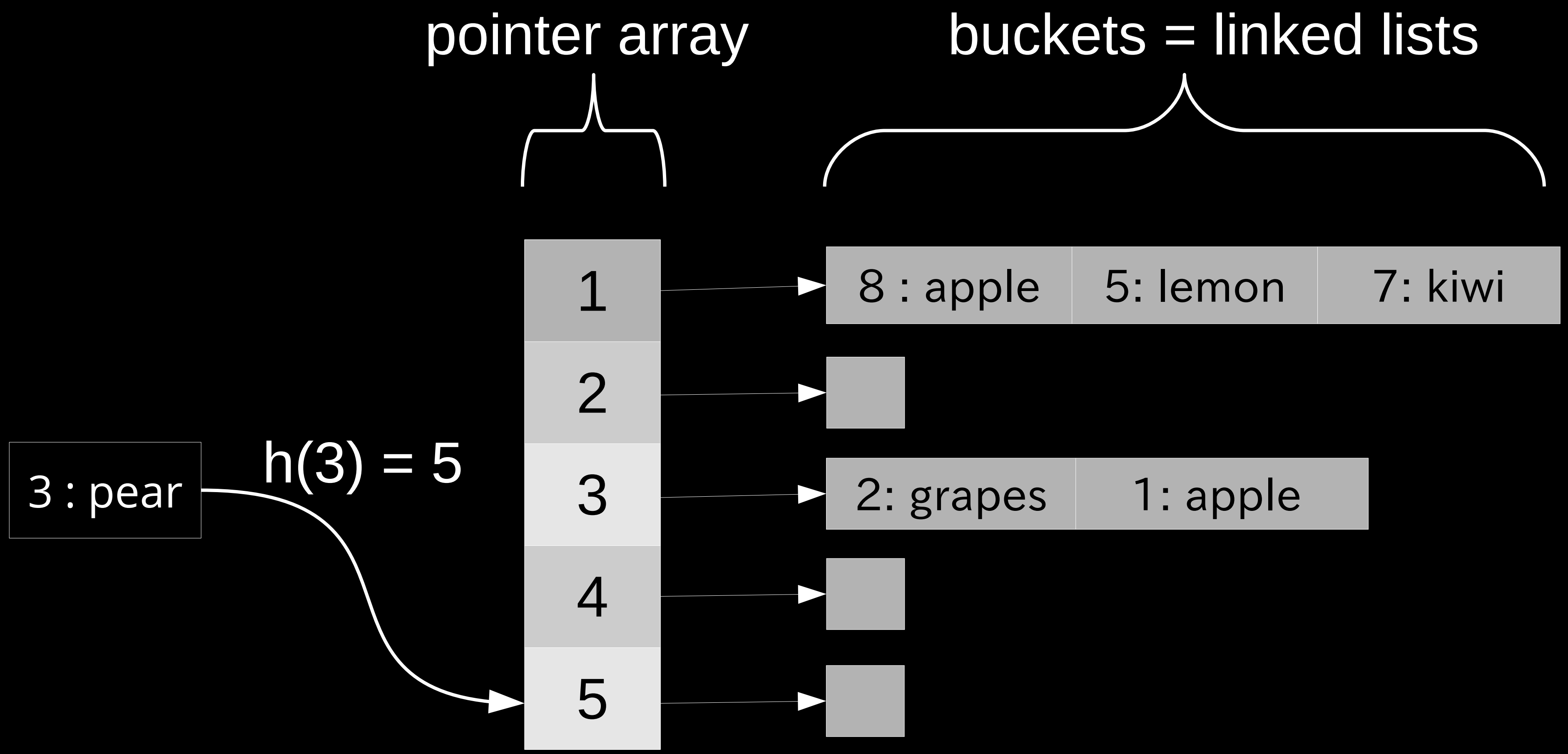


closed addressing



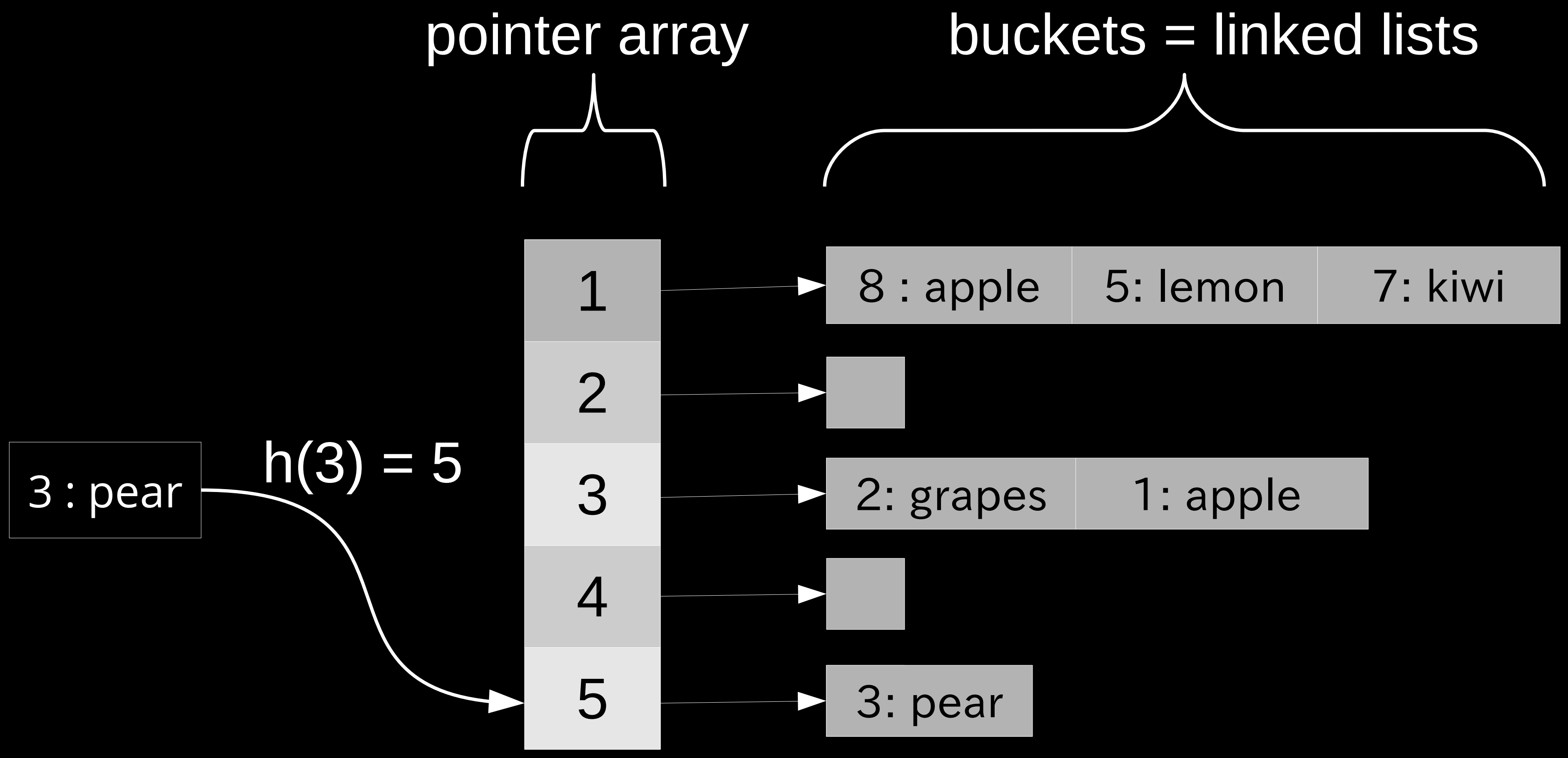
h: hash function

closed addressing



h: hash function

closed addressing

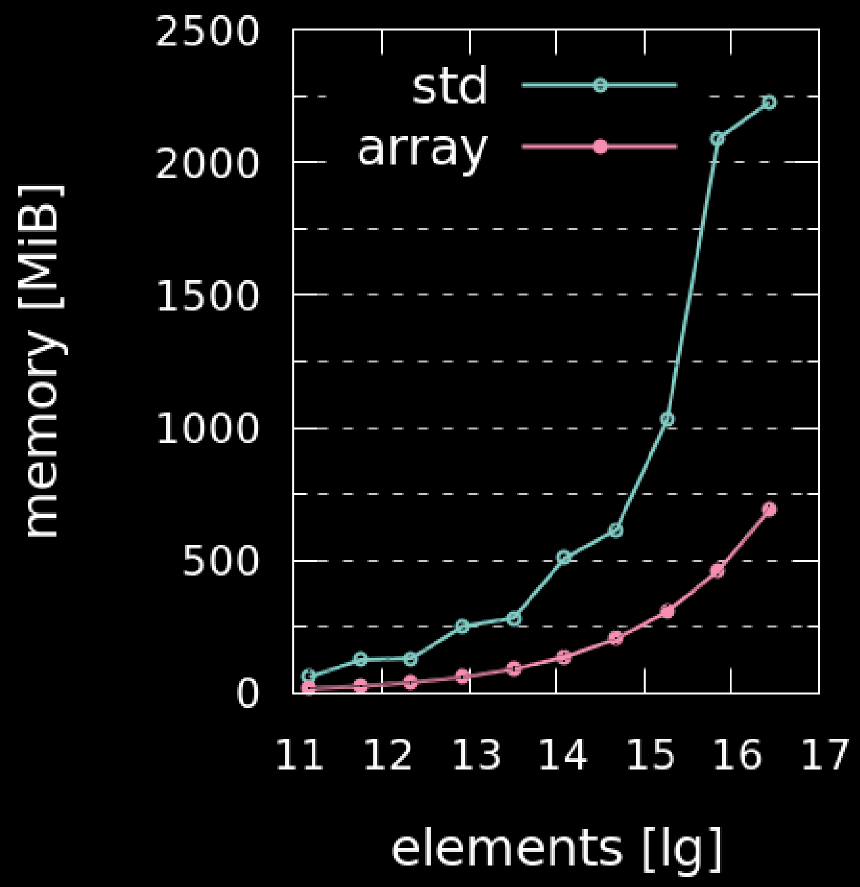


h: hash function

array list

array:

- key and values stored in a list
- ordered by insertion time



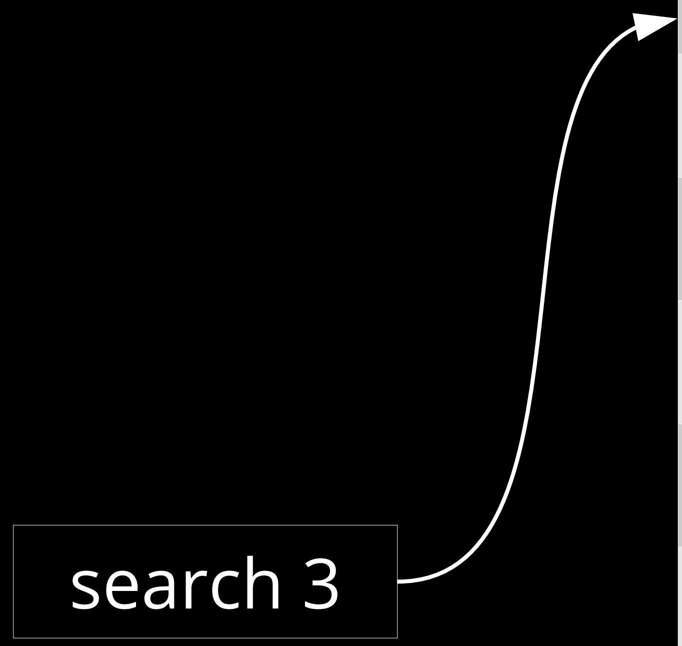
array list

searching a key:

key	value
2	grapes
8	apple
5	lemon
1	apple
7	kiwi
3	pear

array list

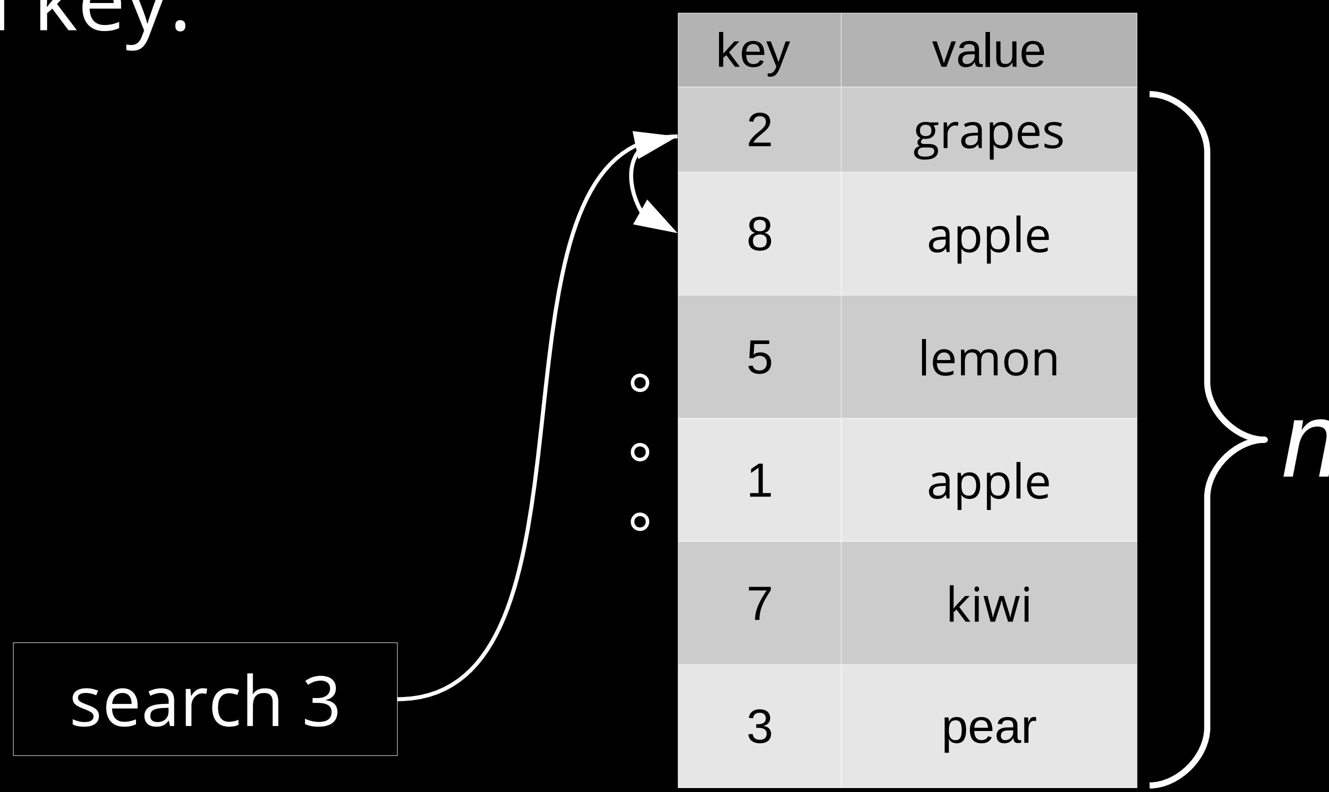
searching a key:



key	value
2	grapes
8	apple
5	lemon
1	apple
7	kiwi
3	pear

array list

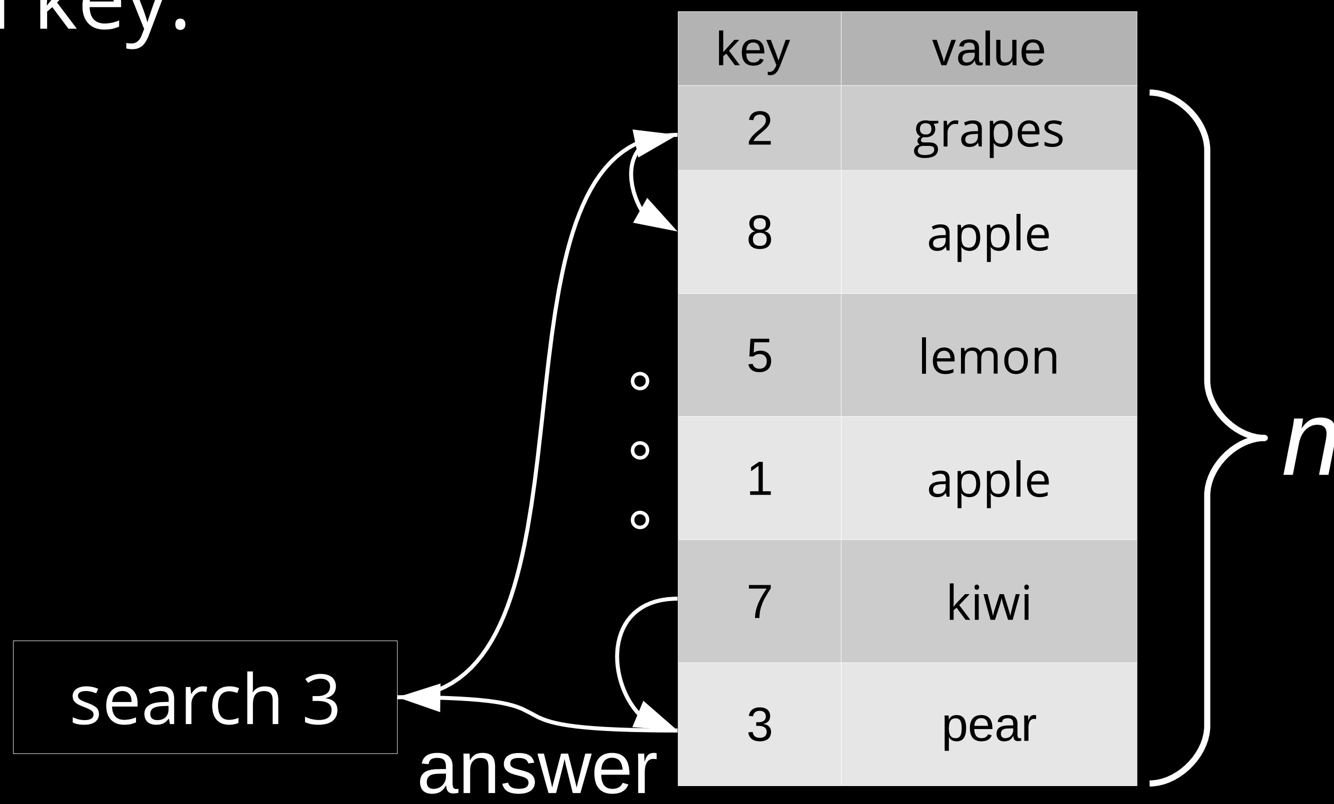
searching a key:



array list

searching a key:

- O(n) time

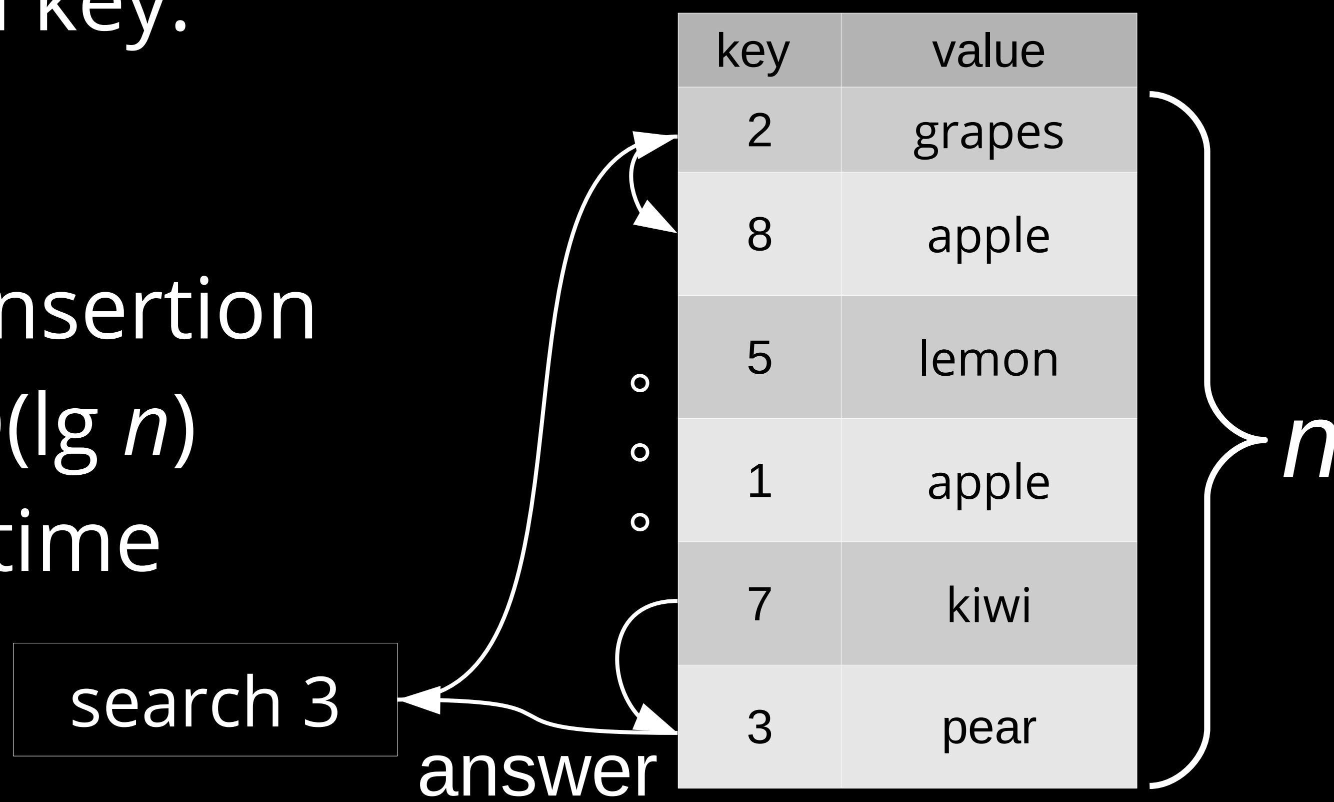


array list

searching a key:

- O(*n*) time
- if we sort, insertion becomes O(lg *n*) amortized time

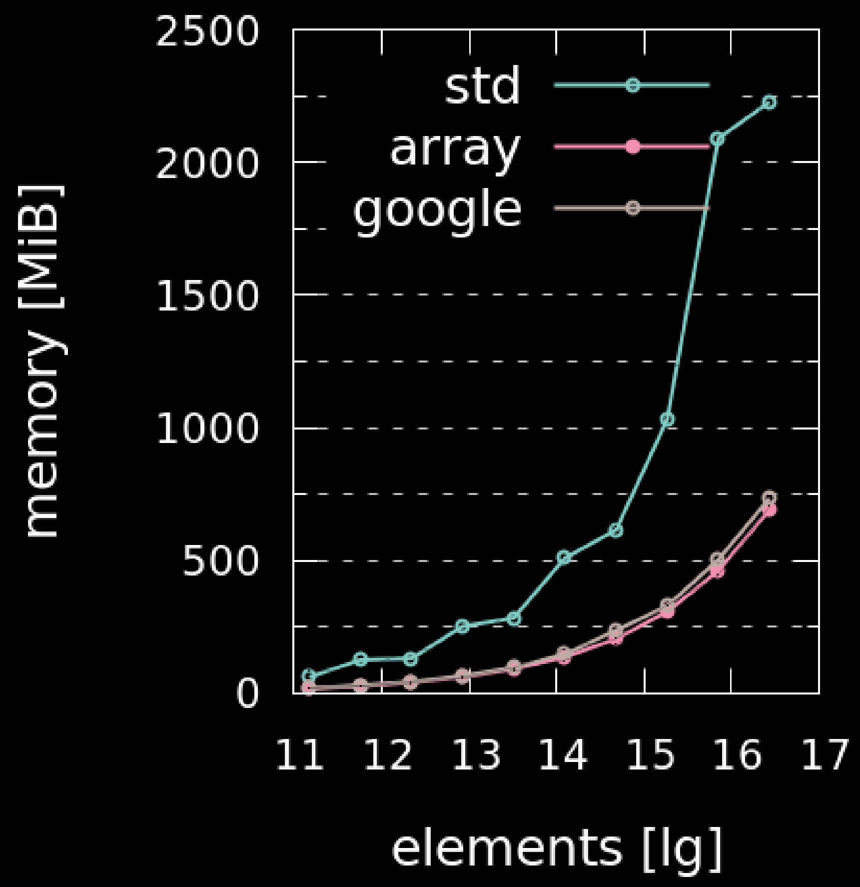
(not fast)



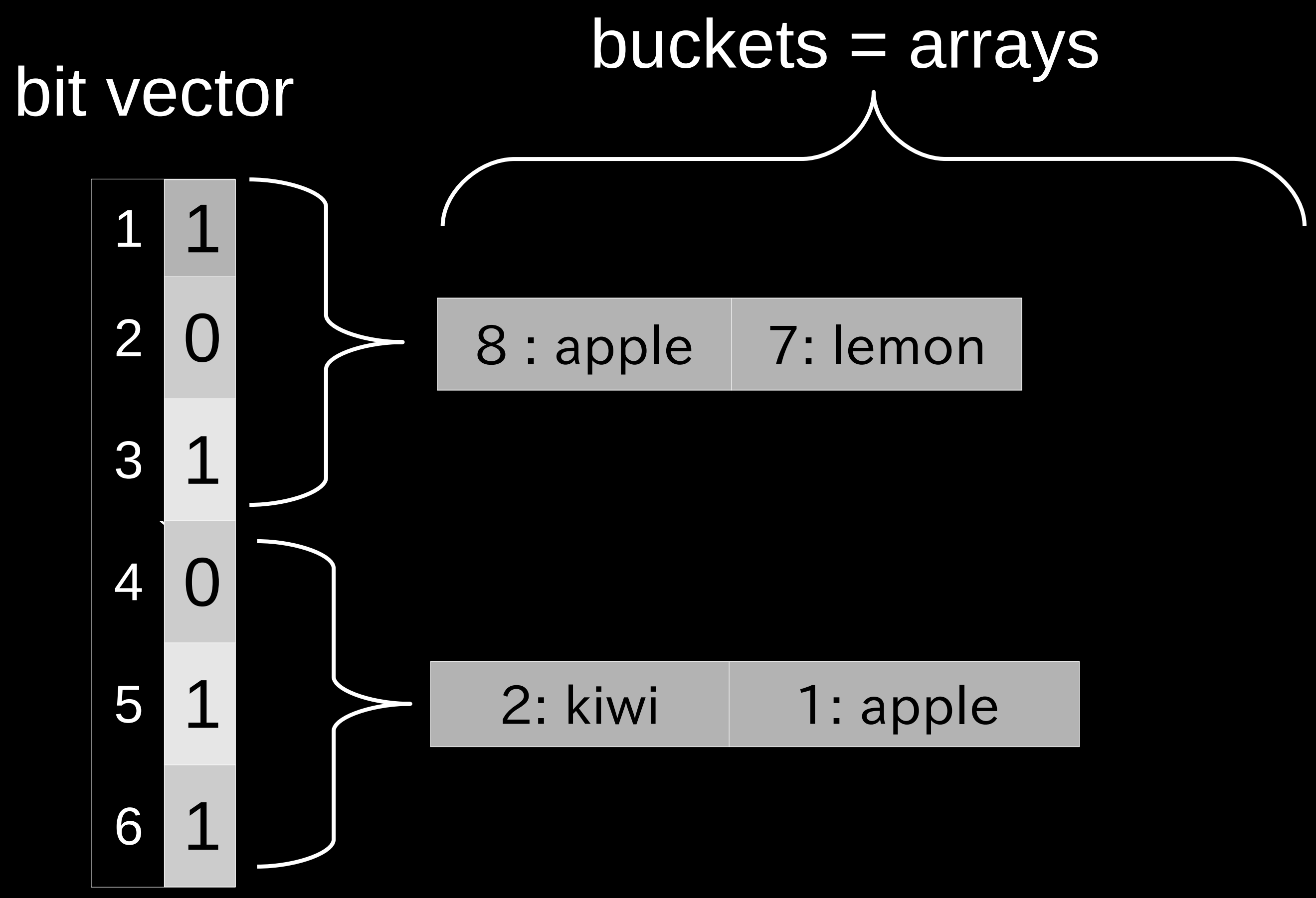
google sparse hash

google:

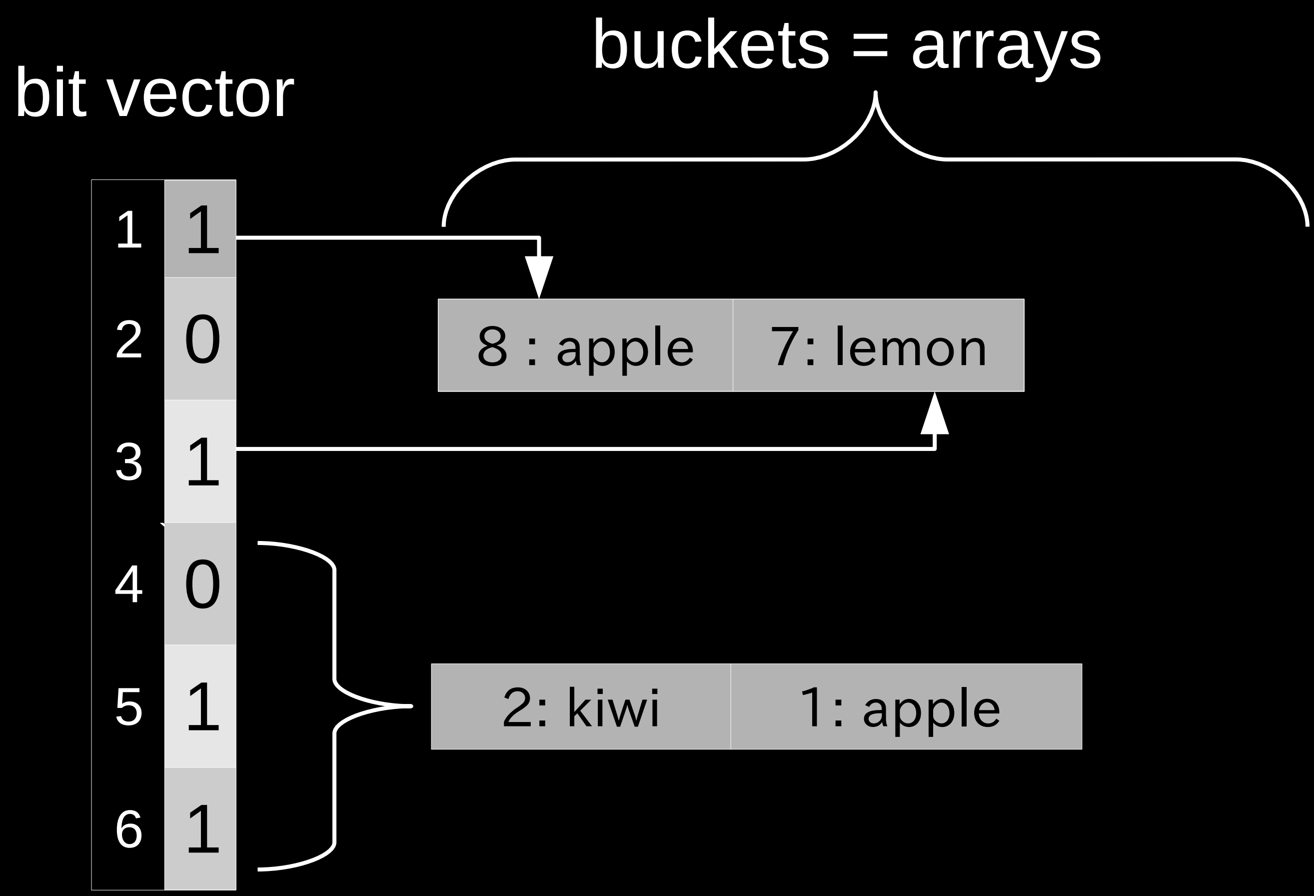
- open addressing
- grouped into *dynamic* buckets
- a bit vector addresses buckets



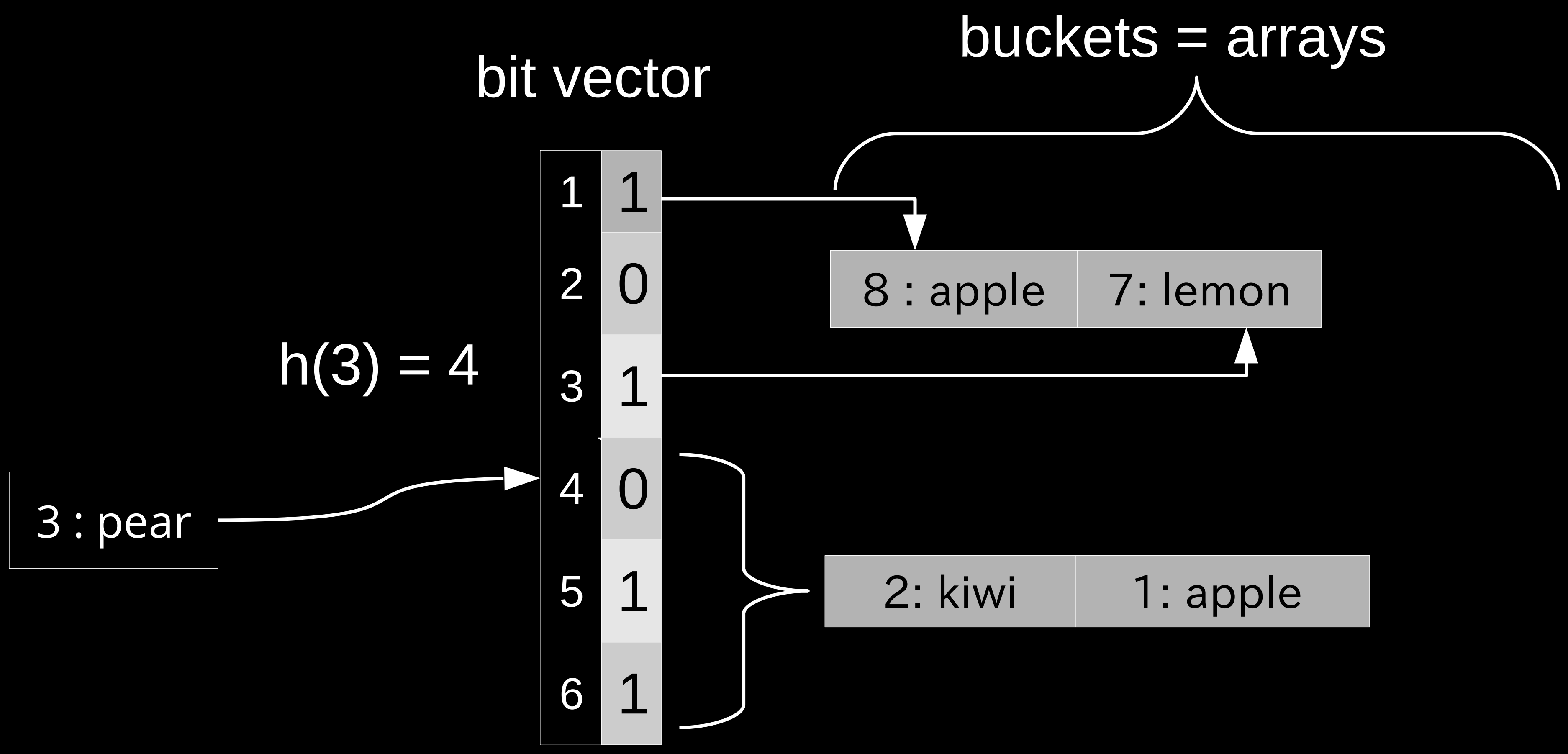
sparse hash table



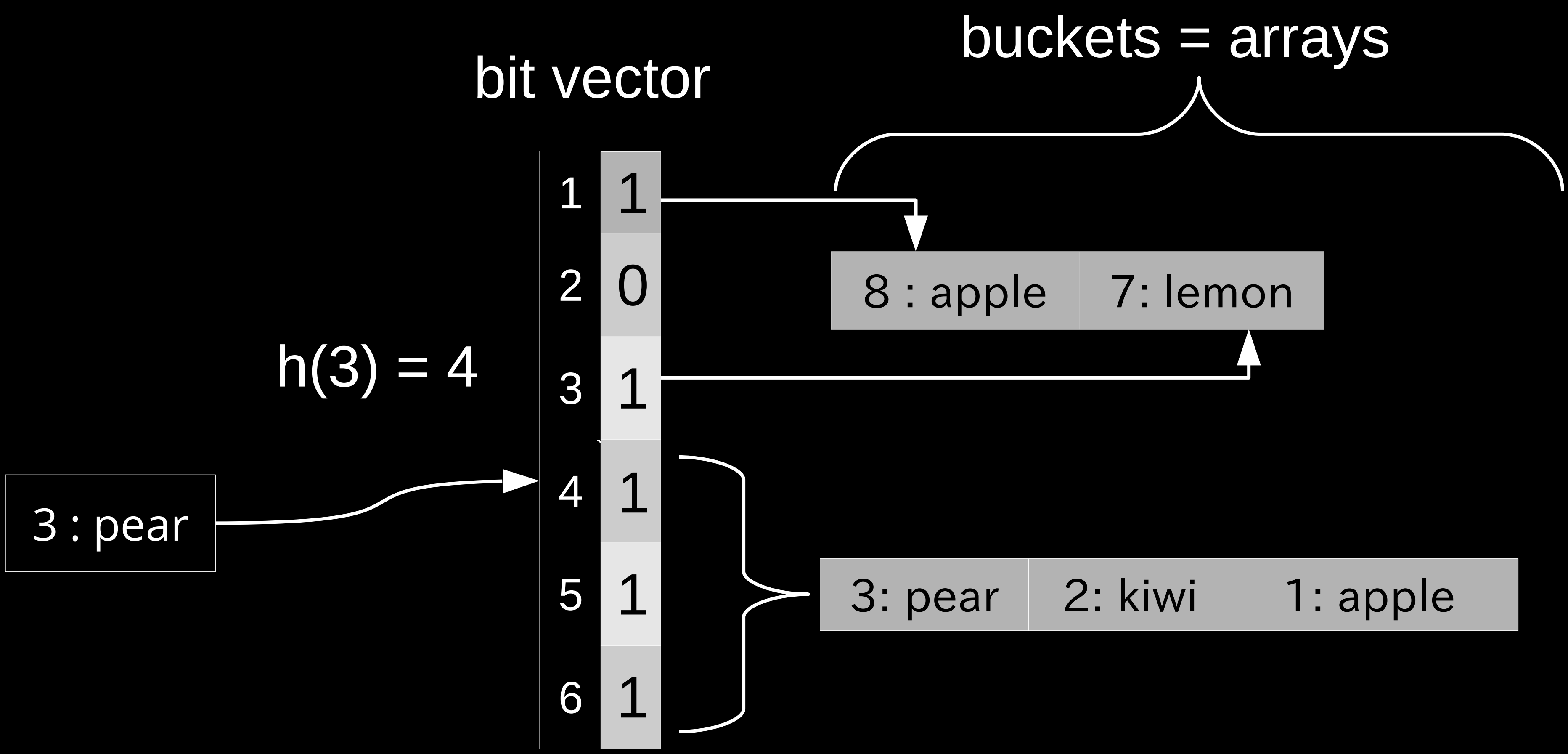
sparse hash table



sparse hash table



sparse hash table



compact hashing

Cleary '84:

- open addressing
- $\varphi : K \rightarrow \varphi(K)$ bijection
  - $\varphi(k) = (h(k), r(k))$
  - $\varphi^{-1}(h(k),r(k)) = k$
- instead of k store $r(k)$

  (may need less space than k)

compact hashing

$\varphi(k) = (h(k), r(k))$

$h(k)$	$(r(k)$, value)
1	2: kiwi
2	1: apple
3	
4	3: apple
5	

compact hashing

$\varphi(k) = (h(k), r(k))$

$\varphi(5) = (3,2)$

5 : lemon

$h(k)$	$(r(k)$, value)
1	2: kiwi
2	1: apple
3	
4	3: apple
5	

compact hashing

$\varphi(k) = (h(k), r(k))$

$\varphi(5) = (3,2)$

5 : lemon

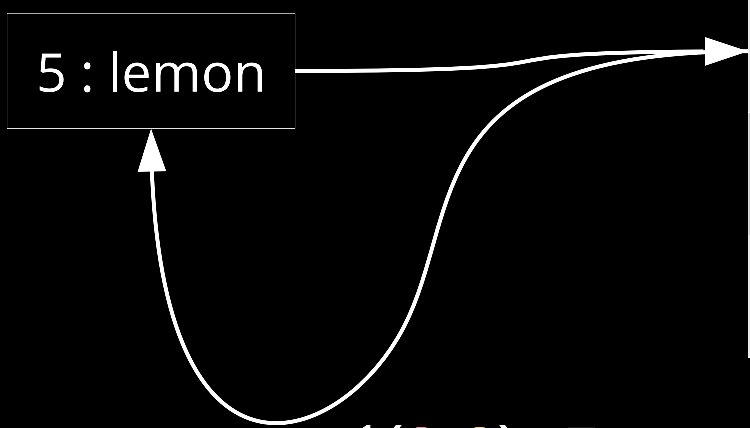
h(k)	(r(k), value)
1	2: kiwi
2	1: apple
3	2: lemon
4	3: apple
5	

compact hashing

$\varphi(k) = (h(k), r(k))$

$\varphi(5) = (3,2)$

$h(k)$	$(r(k)$, value$)$
1	2: kiwi
2	1: apple
3	2: lemon
4	3: apple
5	



$\varphi^{-1}(3,2)=5$

Cleary: linear probing

$\varphi(k) = (h(k), r(k))$

$h(k)$	($r(k)$, value)
1	2: kiwi
2	1: apple
3	2: lemon
4	3: apple
5	

Cleary: linear probing

$\varphi(k) = (h(k), r(k))$

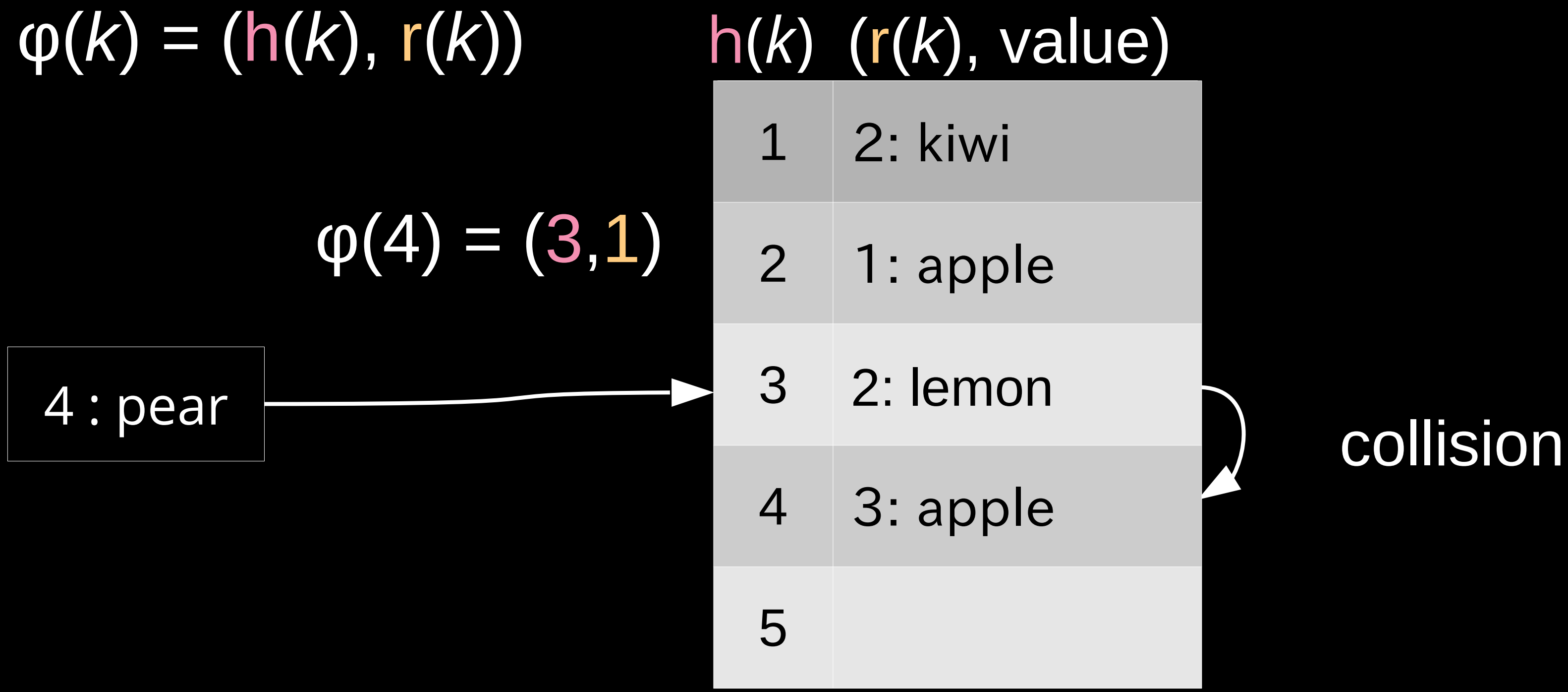
$\varphi(4) = (3,1)$

4 : pear

$h(k)$	$(r(k)$, value)
1	2: kiwi
2	1: apple
3	2: lemon
4	3: apple
5	

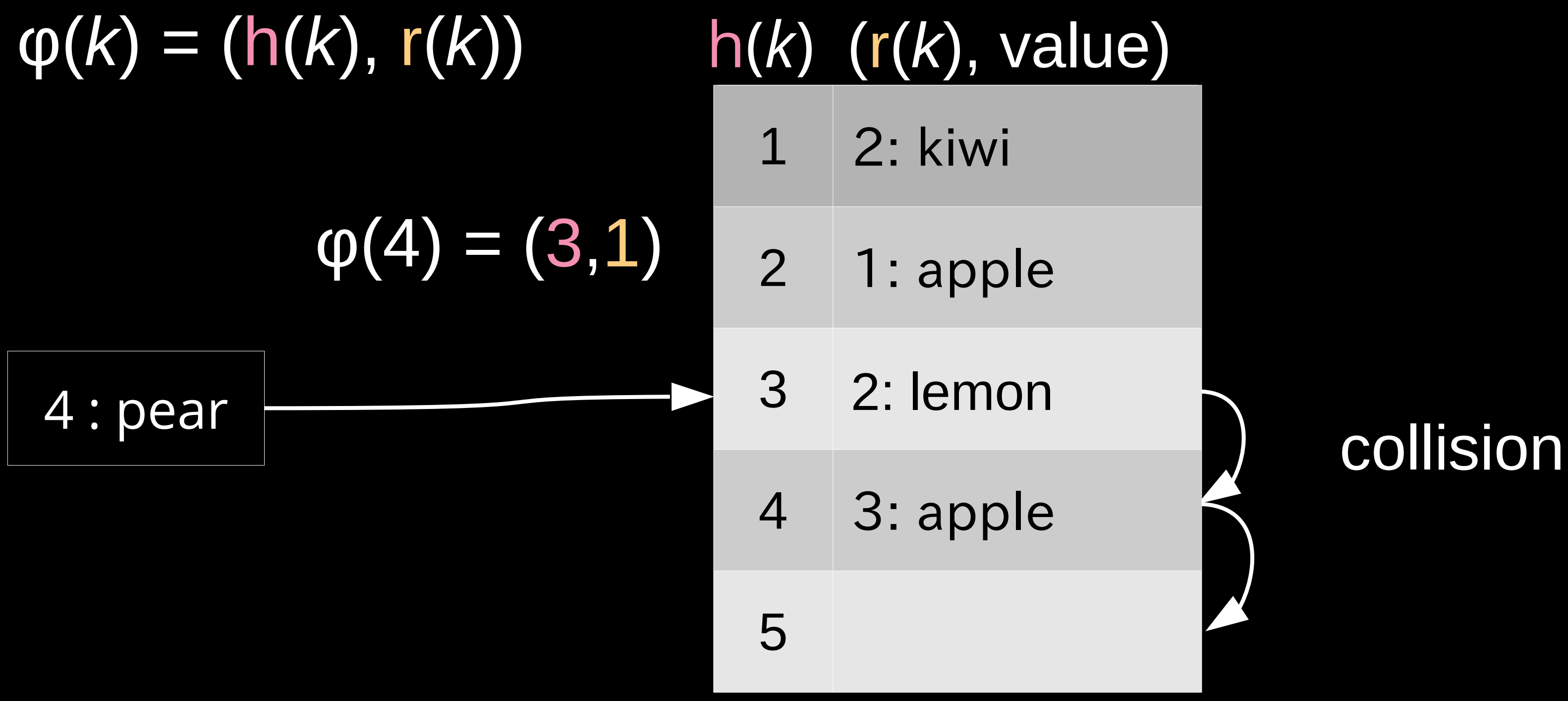
Cleary: linear probing

$\varphi(k) = (h(k), r(k))$



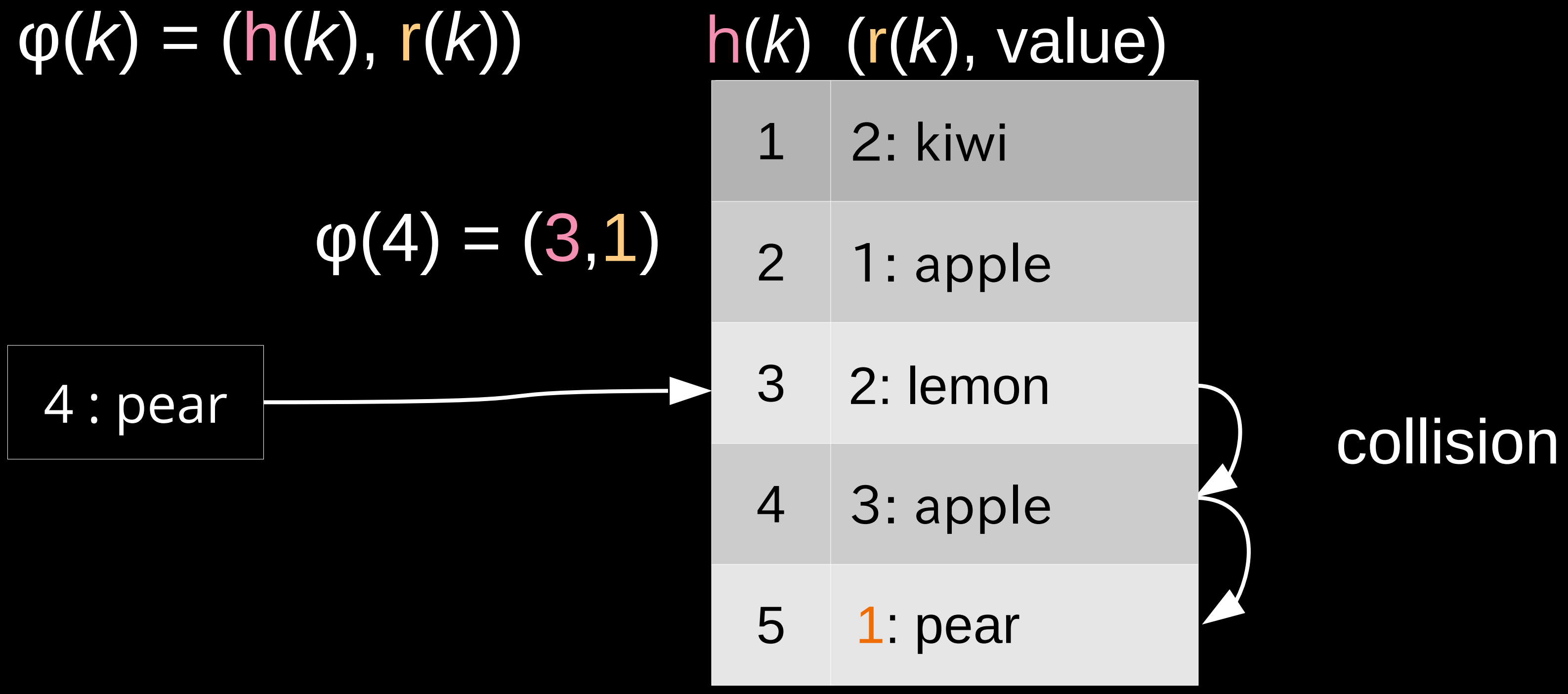
Cleary: linear probing

$\varphi(k) = (h(k), r(k))$

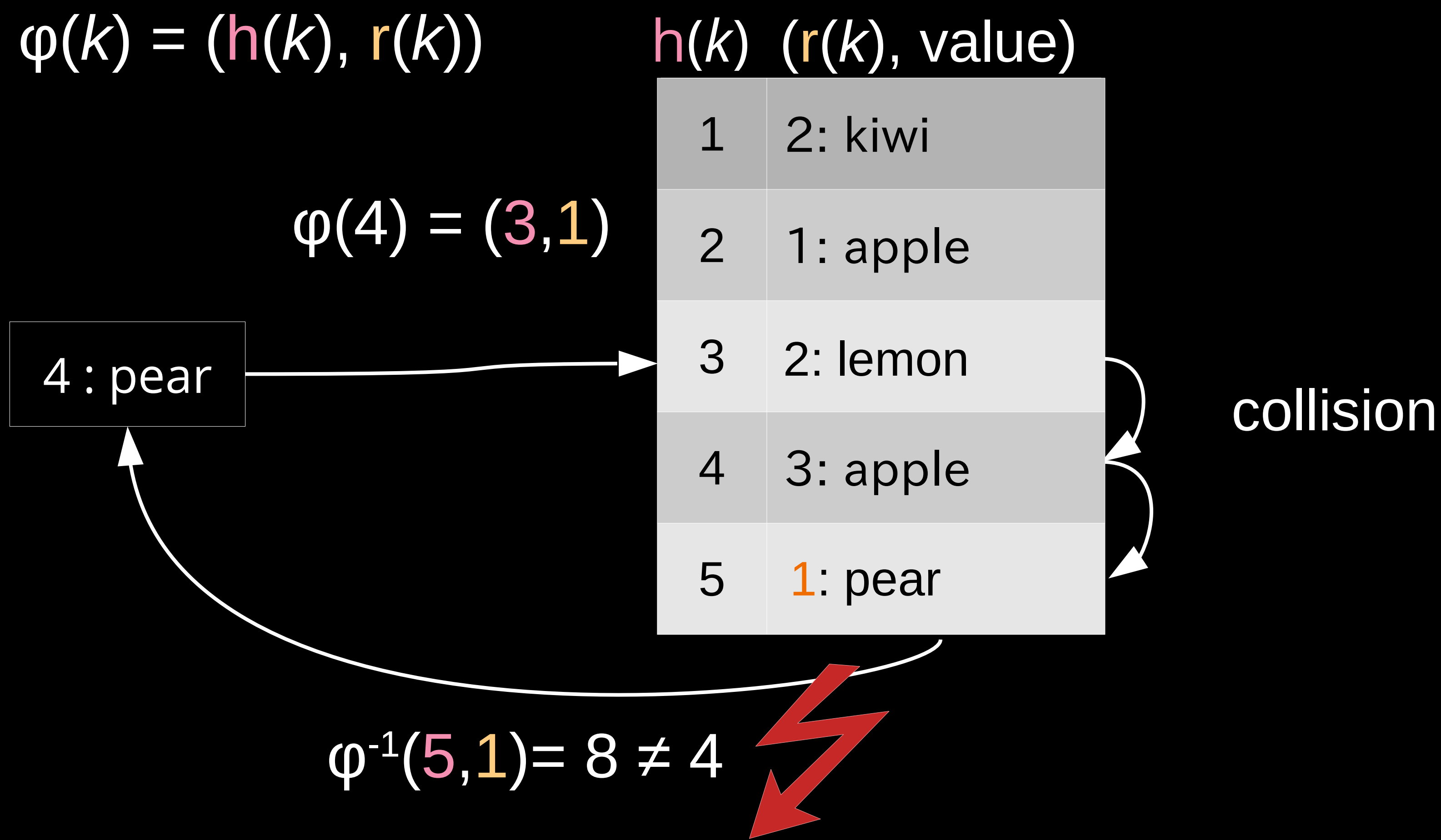


Cleary: linear probing

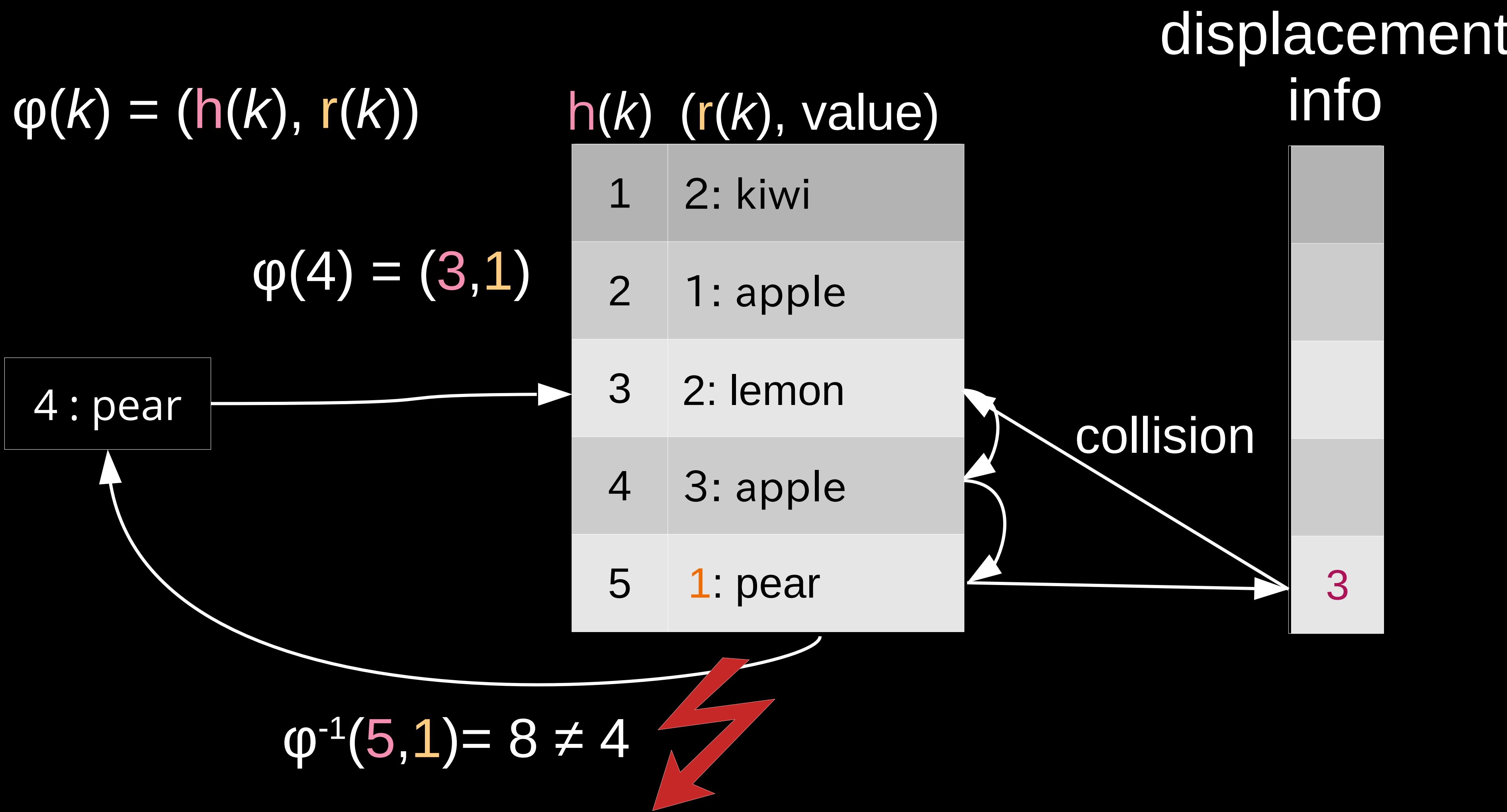
$\varphi(k) = (h(k), r(k))$



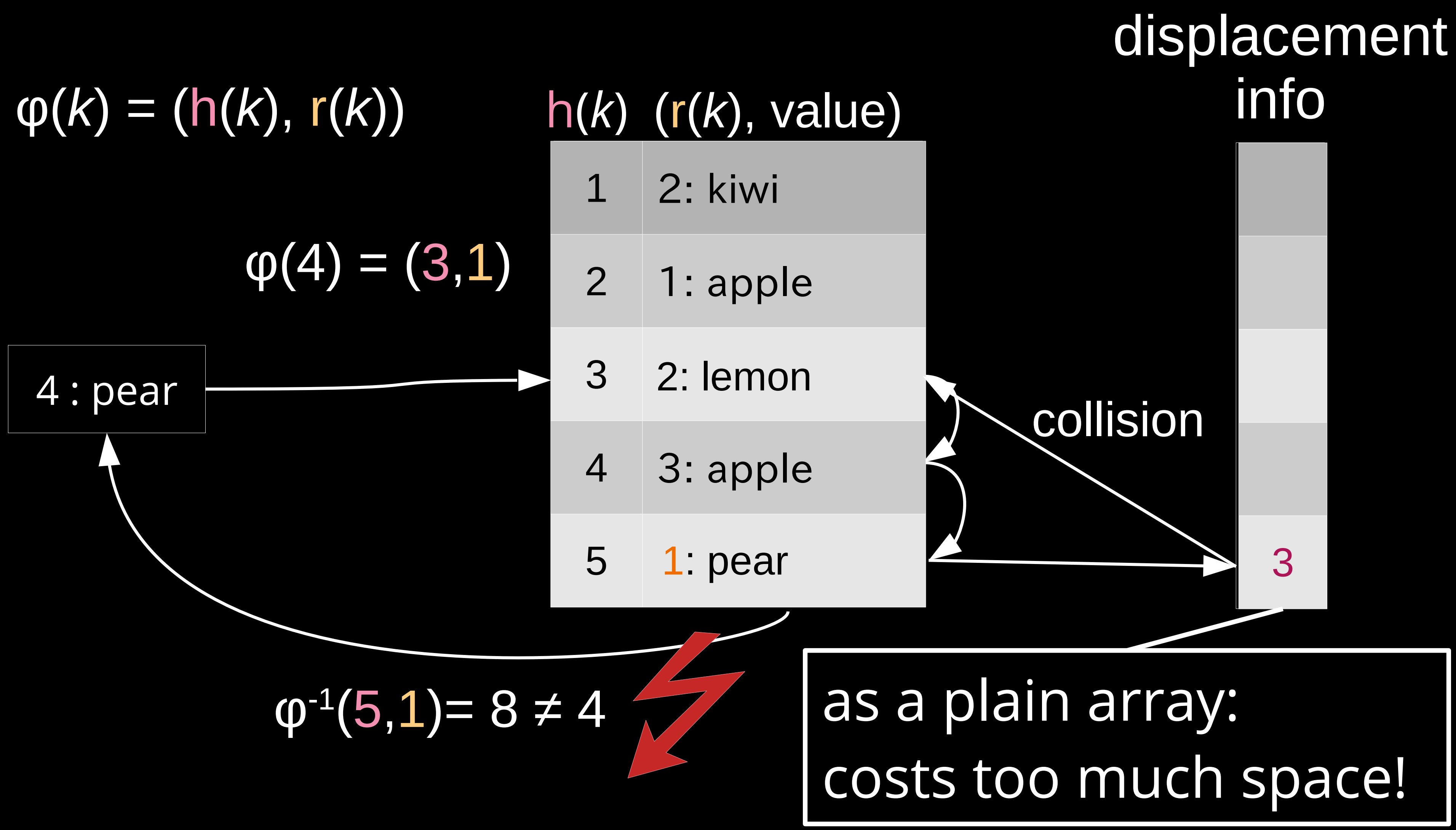
Cleary: linear probing



Cleary: linear probing



Cleary: linear probing

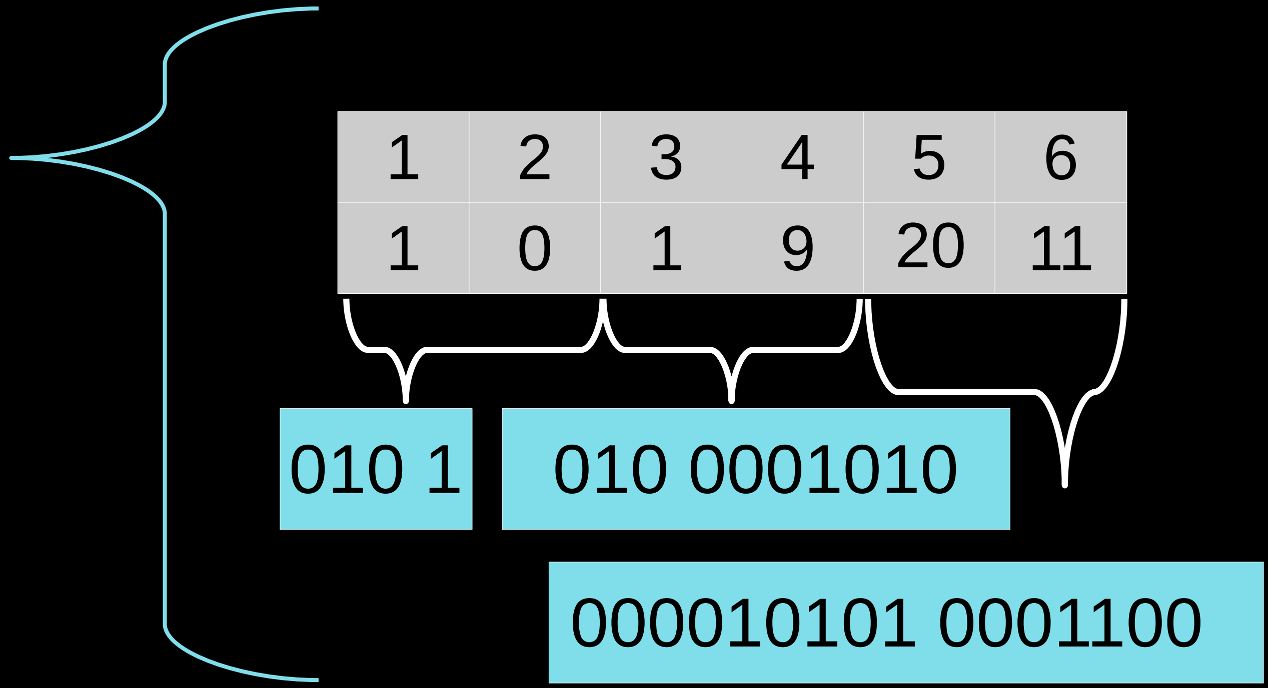


displacement info

m : image size of h
= # cells in H

representations :

- Cleary '84: $2m$ bits
- Poyias+ '15:
  - Elias γ code
  - layered array



displacement info

representations :

- Cleary '84: 2*m* bits
- Poyias+ '15:
  - Elias γ code
  - layered array

4 bit integer array

1	2	3	4	5	6
1	0	1	9		11

displacement info

representations :

- Cleary '84: 2*m* bits
- Poyias+ '15:
  - Elias γ code
  - layered array

displacement: 20

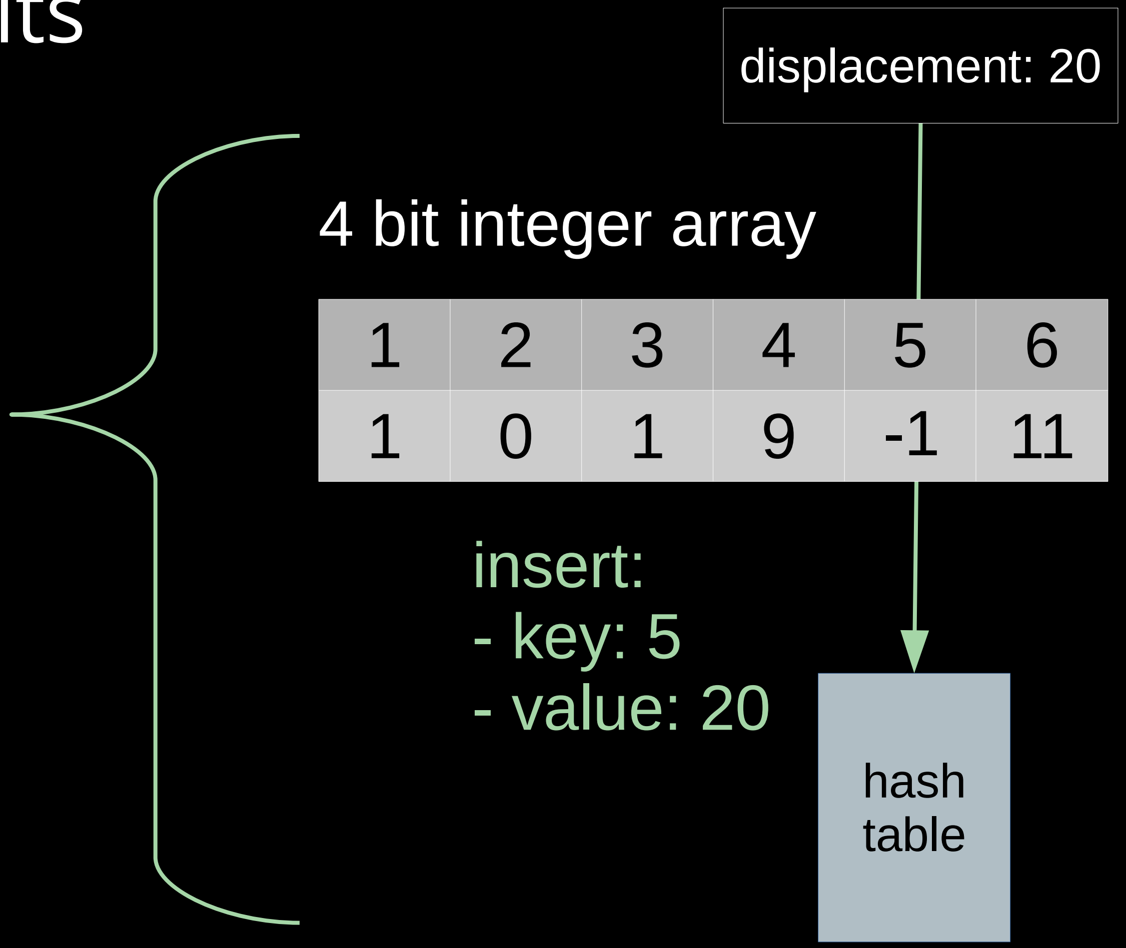
4 bit integer array

1	2	3	4	5	6
1	0	1	9		11

displacement info

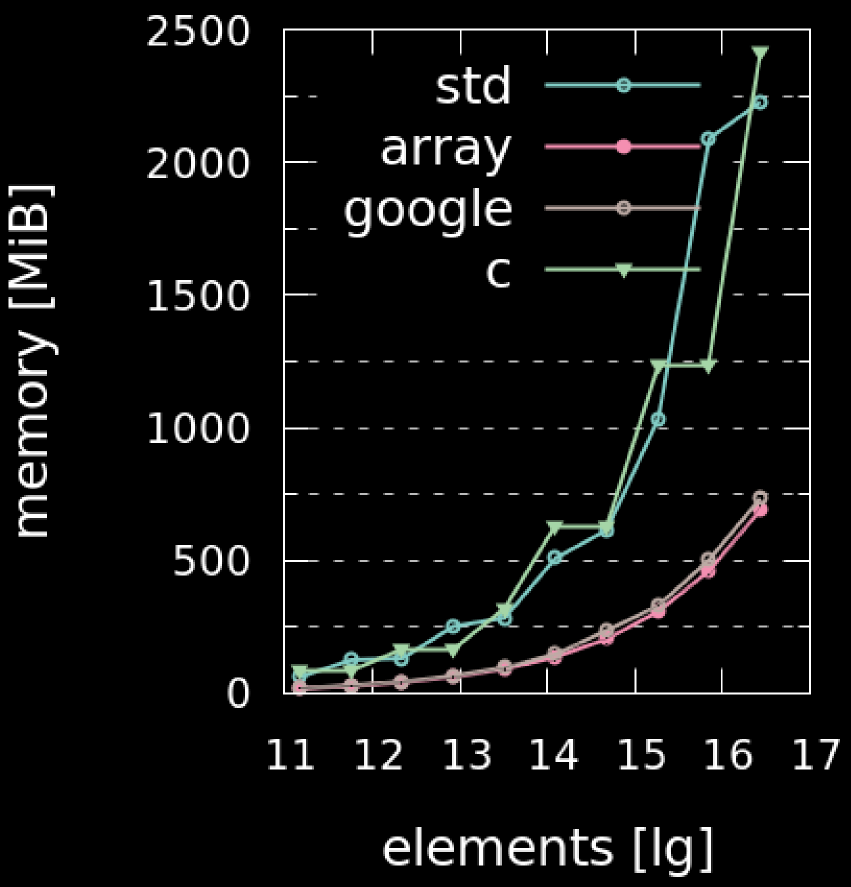
representations :

- Cleary '84: 2m bits
- Poyias+ '15:
  - Elias γ code
  - layered array



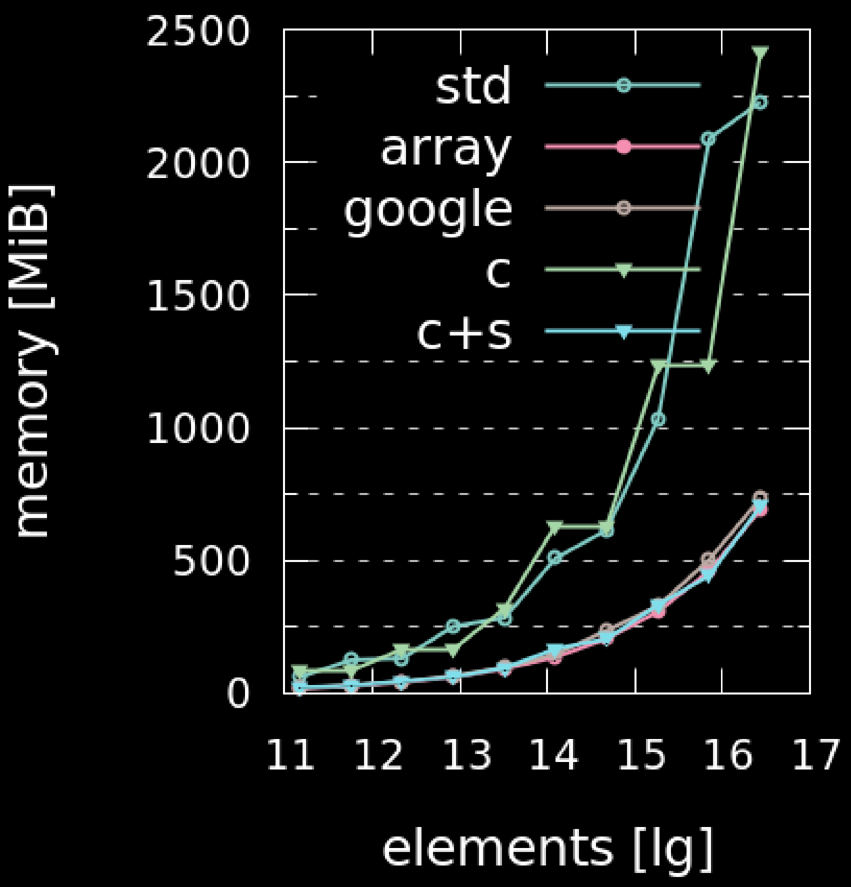
memory benchmark

- c: compact
  - layered
  - max. load factor 0.5
- not space efficient!



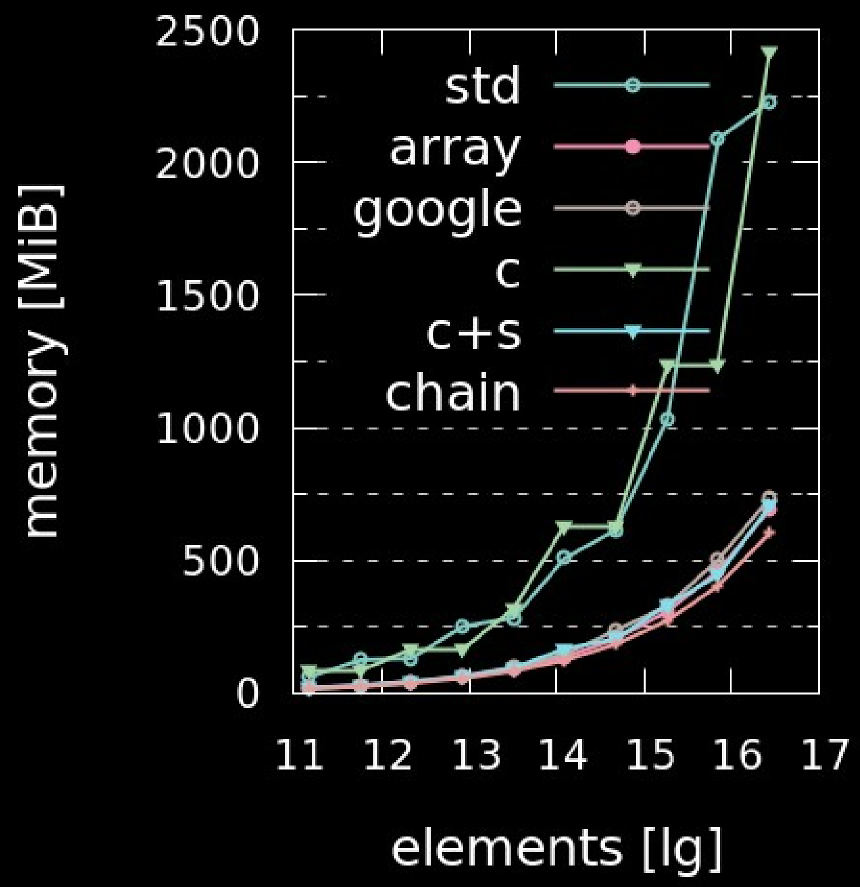
memory benchmark

- c+s: composition of
  - compact with
  - sparse
- competitive with array



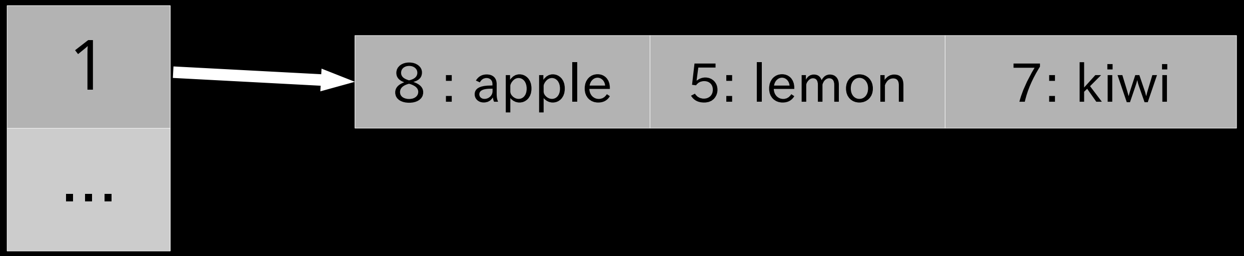
chain

- composition of
  - closed addressing
  - array
  - compact
- most space efficient (our contribution)



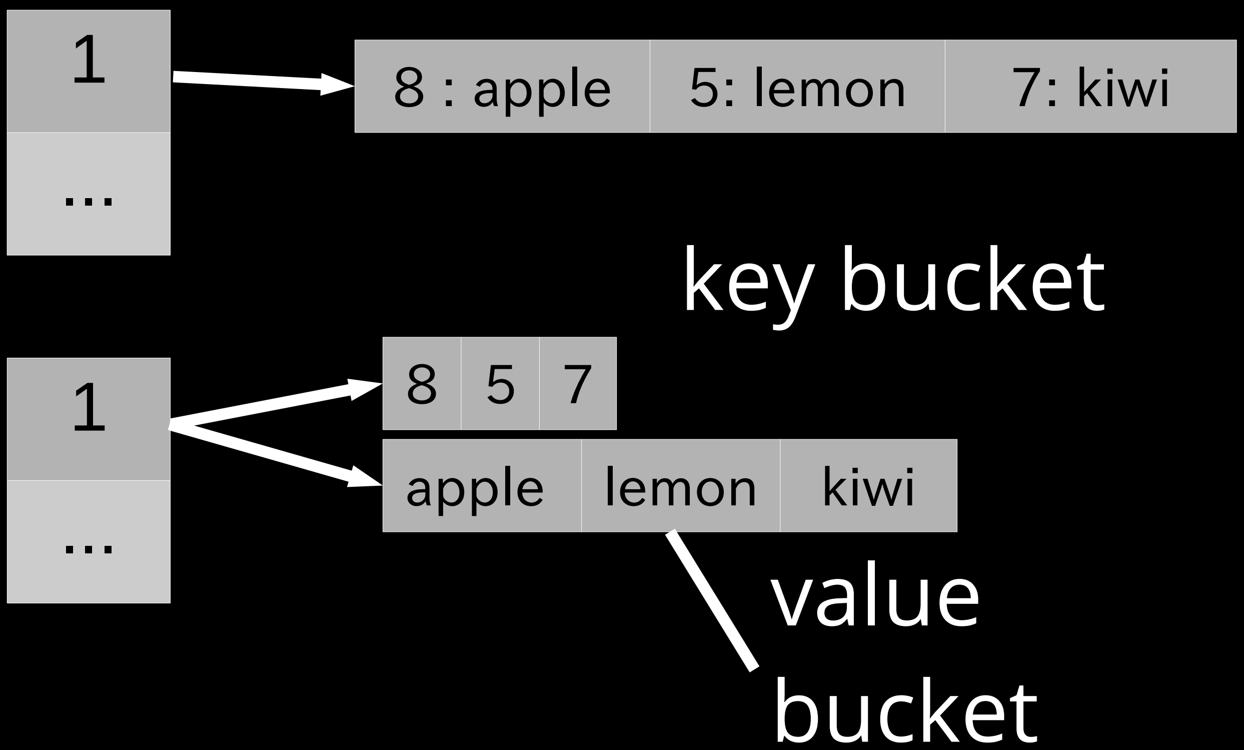
chain

- closed addressing



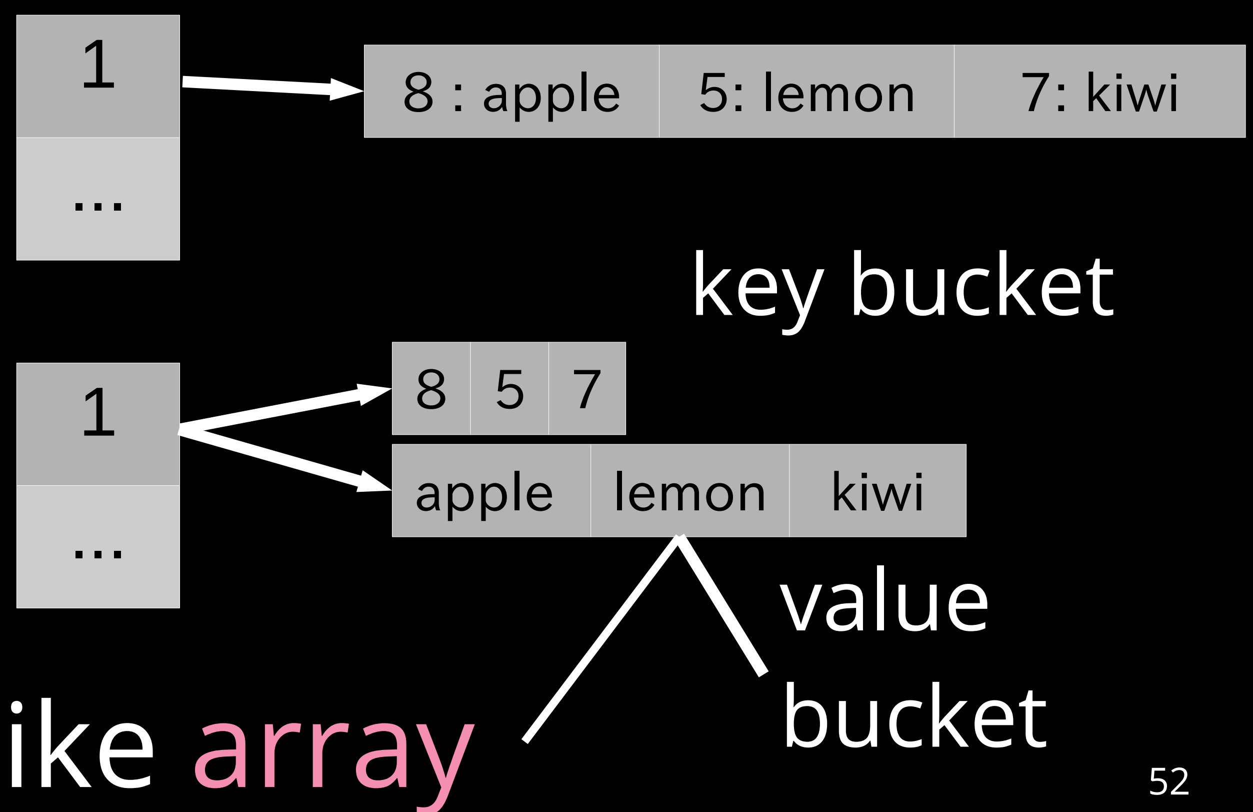
chain

- closed addressing
- buckets: instead of lists use two arrays



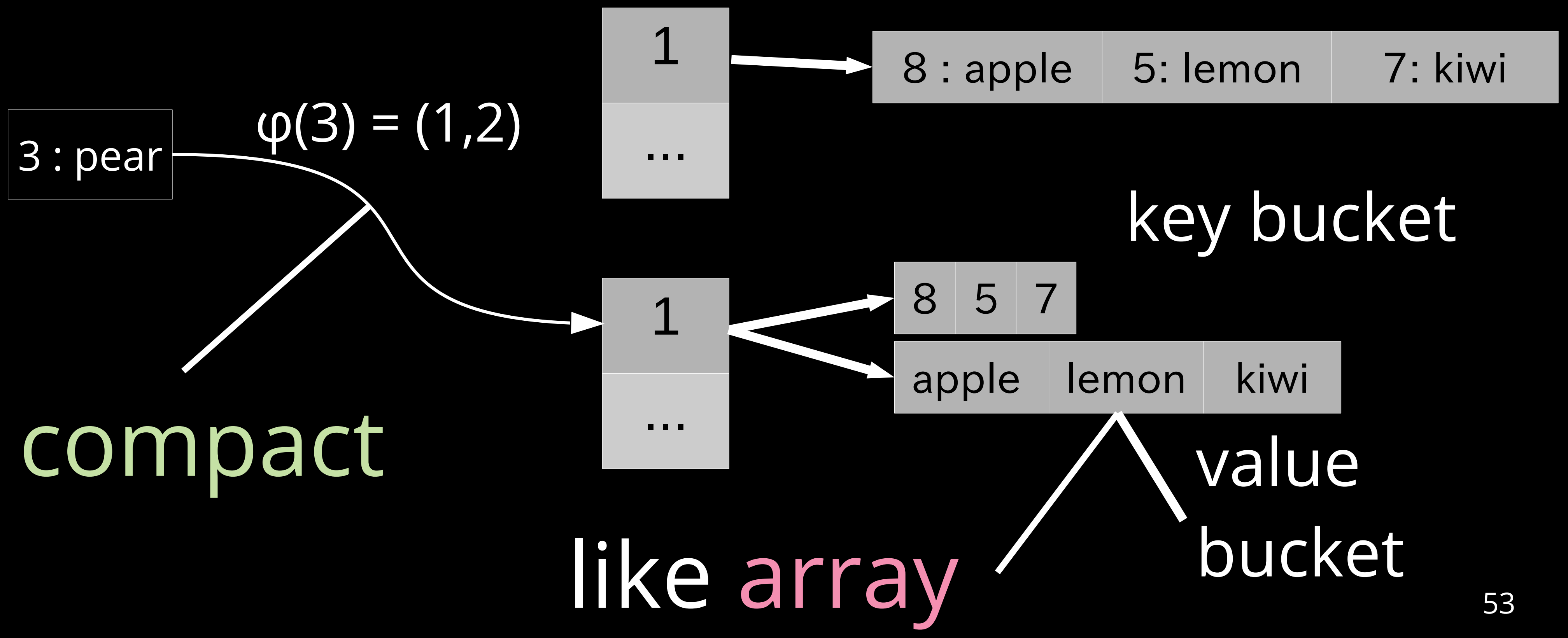
chain

- closed addressing
- buckets: instead of lists use two arrays



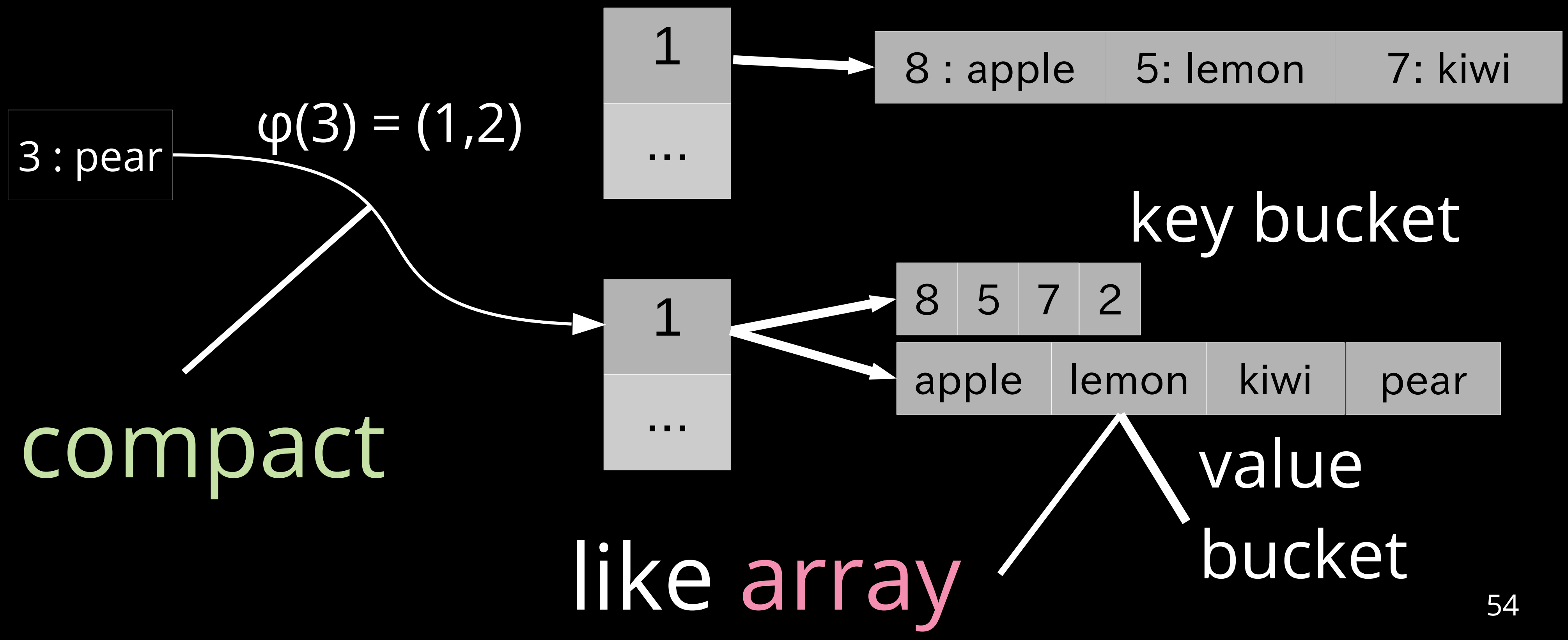
chain

- closed addressing
- buckets: instead of lists use two arrays



chain

- closed addressing
- buckets: instead of lists use two arrays



chain: space analysis

- a bucket costs O(ω) bits (pointer + length)
- want O(n lg n) bits

  ⇒ # buckets: O(n / ω)
- then m = n / ω (image size of h)
- r(k) uses ~ ω - lg(n /ω) = ω - lg n + lg ω bits

- K = [1..2^{ω}]
- n: #elements

chain: space analysis

- a bucket costs O(ω) bits (pointer + length)
- want O(n lg n) bits
  ⇒ # buckets: O(n / ω)
- then m = n / ω (image size of h)
- r(k) uses ~ ω - lg(n /ω) = ω - lg n + lg ω bits

space for improvement!

r(k) of compact

- K = [1..$2^{ω}$]
- n: #elements

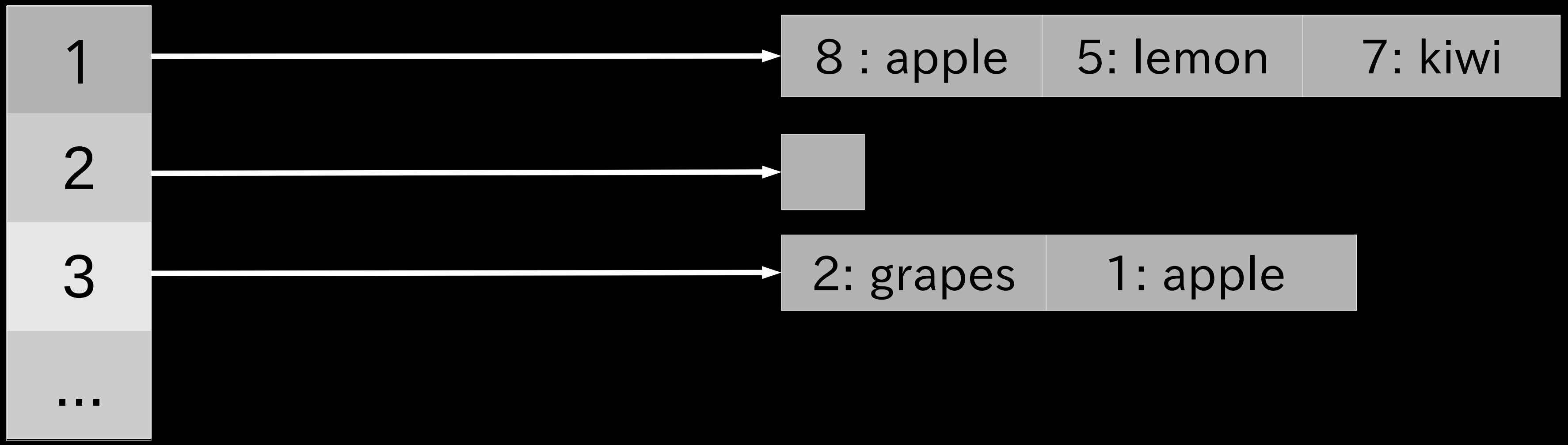
improve space

- want n buckets such that $m = n$
- but each bucket costs $O(\omega)$ bits!
- idea: maintain buckets in a group

  (similar to sparse)

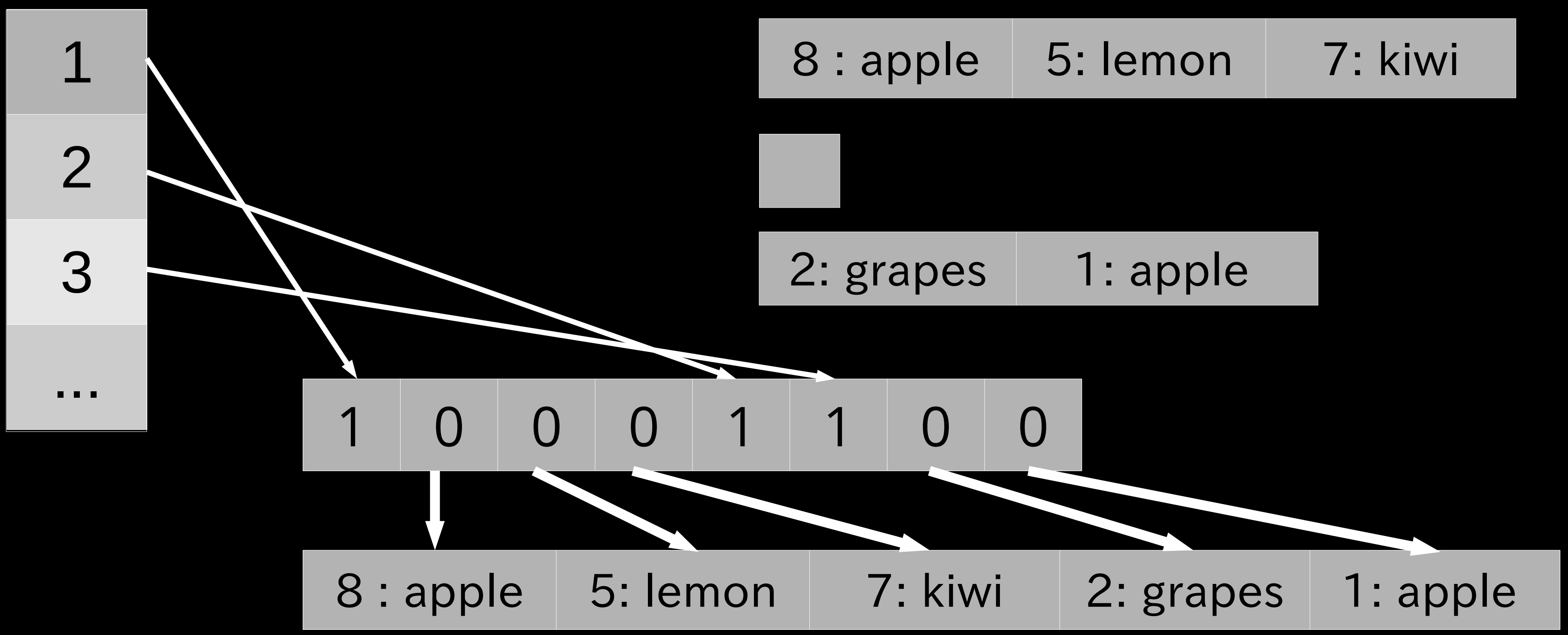
chain → grp

- chain represents each bucket separately



chain → grp

- chain represents each bucket separately
- grp uses bit vector to mark bucket boundaries



rehashing

chain

- if a bucket reaches $O(\omega)$ elements

grp

- if a group reaches $O(\omega)$ elements
- group bit vector has $O(\omega)$ bits,
- scan bit vector naively

we set this maximum bucket / group size to 255 in practice (⇒length costs a byte)

insertion time

chain

- bucket has O(ω) elements

grp

- group has O(ω) elements

⇒ O(ω) worst-case time
(assuming that we do not need to rehash)

query time

chain

- bucket has
  $O(\omega)$ elements
  $\Rightarrow$ $O(\omega)$ worst-case time

assume that $\Omega(\omega)$ bits fit into a machine word

grp

- bit vector has $O(\omega)$ bits

$\Rightarrow$ find respective bucket in $O(1)$ expected time

- bucket size is $O(1)$ expected

$\Rightarrow$ $O(1)$ expected time

theoretic space bounds

to store n keys from K = [1..2^ω]

we need at least

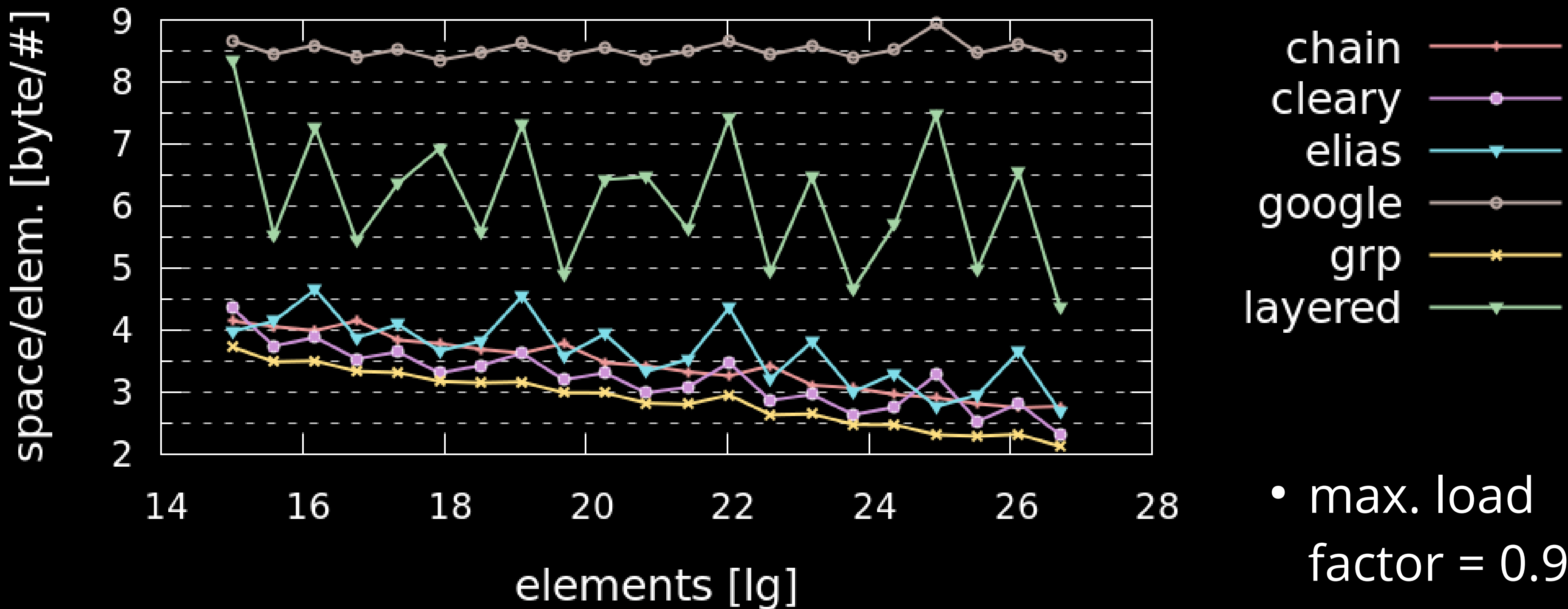
$$B := \lg \binom{2^\omega}{n} = n\,\omega - n \lg n + O(n) \text{ bits}$$

theoretic space bounds

$\varepsilon \in (0,1]$ constant

	construction		query
hash table	**space in bits**	**time**	**expected time**
cleary	$(1+\varepsilon)\, B + O(n)$	$O(1/\varepsilon^3)$ exp.	$O(1/\varepsilon^2)$
elias	$(1+\varepsilon)\, B + O(n)$	$O(1/\varepsilon)$ exp.	$O(1/\varepsilon)$
layered	$(1+\varepsilon)\, B + O(n \lg\lg\lg\lg n)$	$O(1/\varepsilon)$ exp.	$O(1/\varepsilon)$
chain	$B + O(n \lg \omega)$	$O(\omega)$ worst	$O(\omega)$ worst
grp	$B + O(n)$	$O(\omega)$ worst	$O(1)$

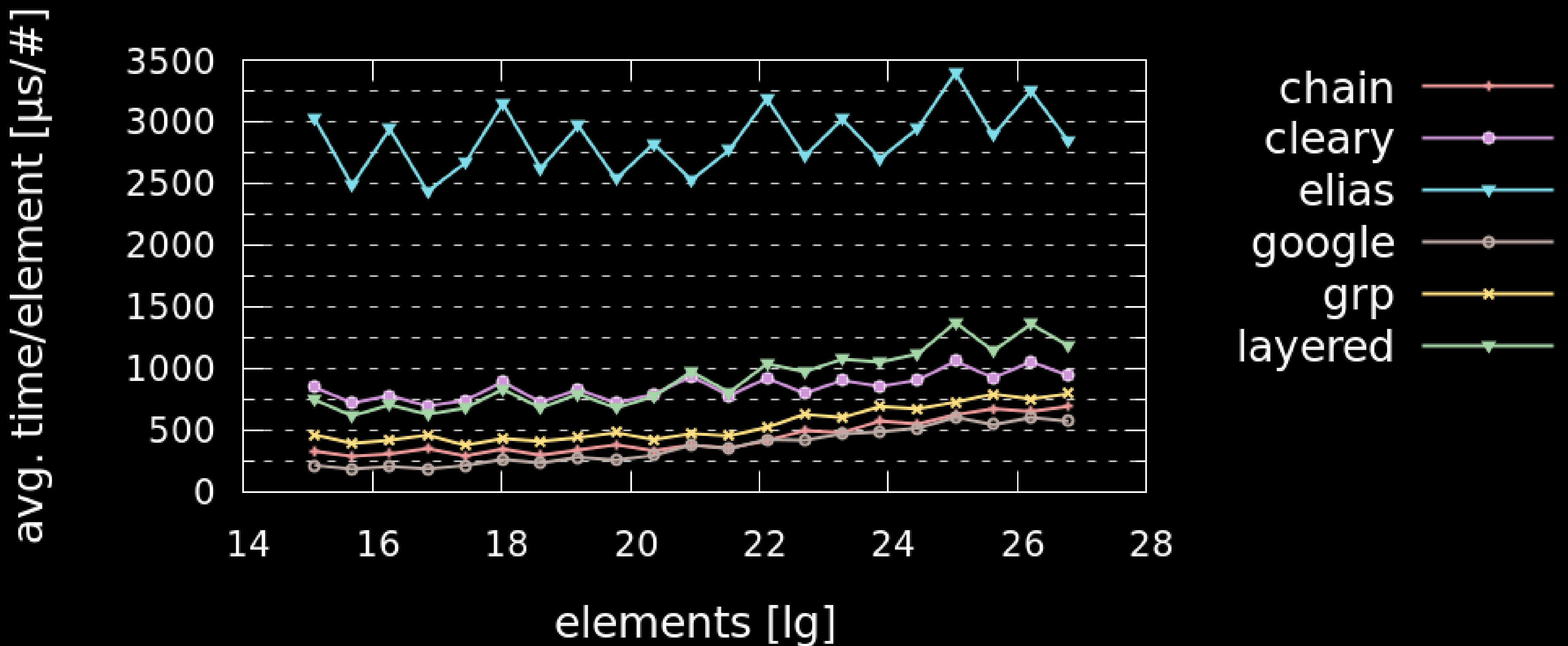
average space per element



- max. load factor = 0.95
- use sparse layout
- 32 bit keys
- 8 bit values

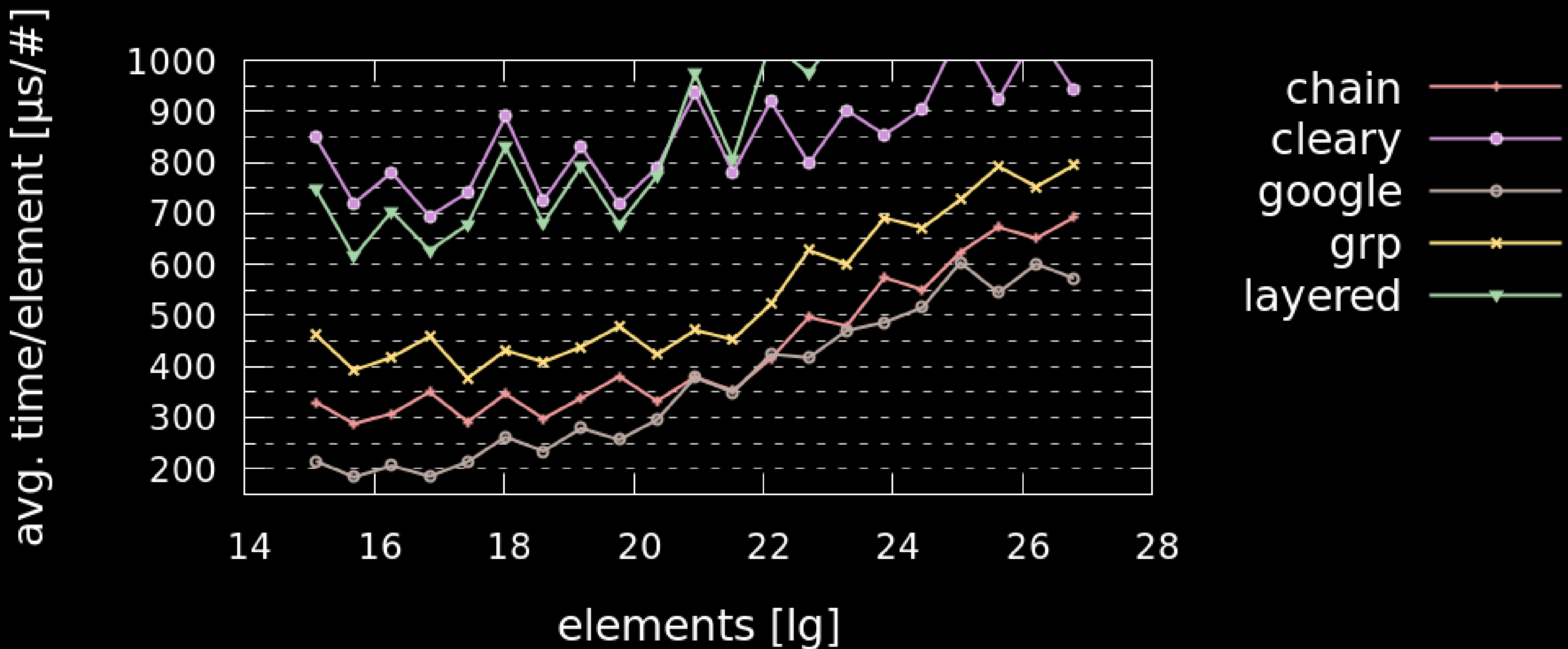
- grp has the smallest space requirements
- cleary, chain, and elias are roughly equal
- google and layered are not as space economic

construction time



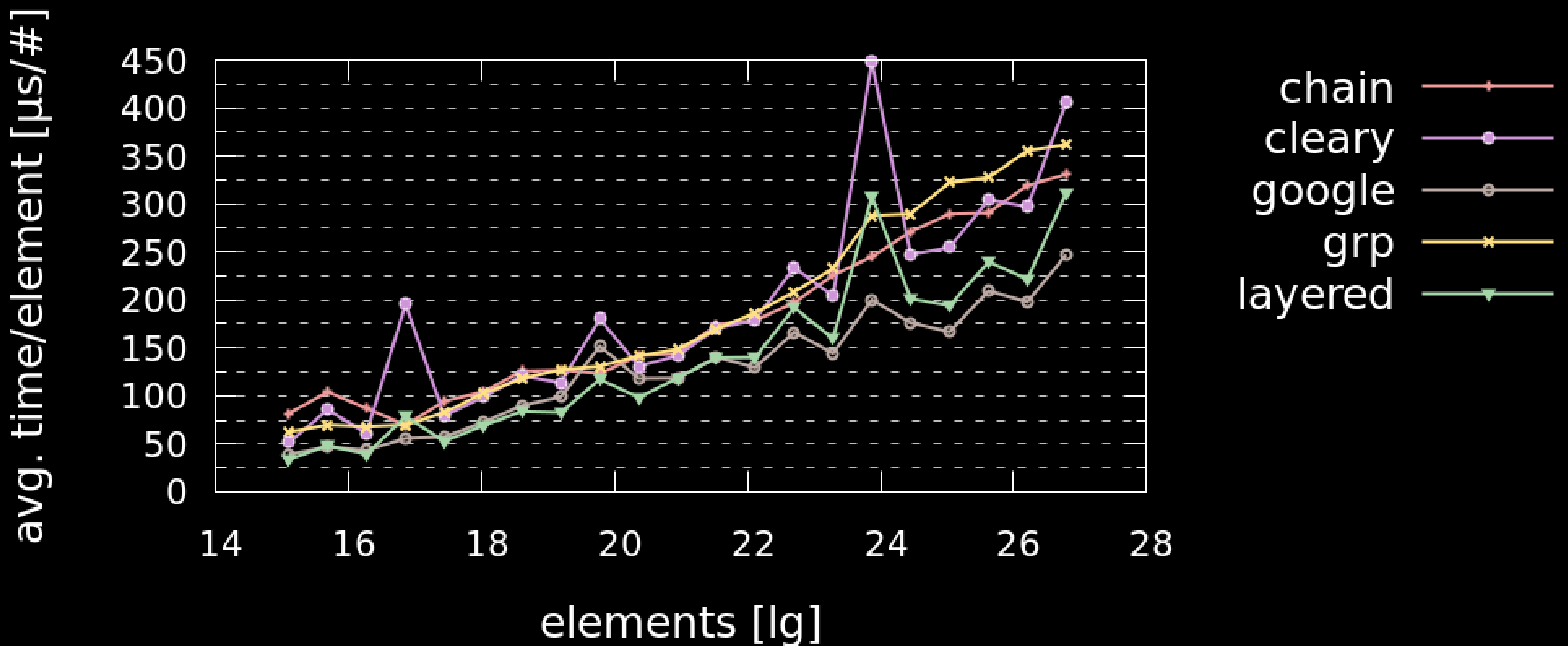
elias is very slow → omit it

construction time



- google is fastest
- grp is always slower than chain
- cleary and layered are slow

query time



- grp is mostly slower than chain
- google is fastest. cleary and layered have spikes (happening at high load factors)

experimental summary

	construction		query
hash table	**space**	**time**	**time**
google	bad	fast	fast
cleary	good	slow	slow
elias	good	very slow	very slow
layered	average	slow	fast
chain	good	fast	slow
grp	best	fast	slow

but sometimes slower than grp at high loads

proposed two hash tables

- techniques are combination of
  - closed addressing
  - bucketing [Askitis'09]
  - compact hashing [Cleary'84]
  - bit vector like in google's sparse table
- characteristics:
  - no displacement info
  - memory-efficient
  - fast construction but
  - slow query times
- current research:
  - speed up queries with SIMD
  - overflow table for averaging the loads of the buckets

proposed two hash tables

- techniques are combination of
  - closed addressing
  - bucketing [Askitis'09]
  - compact hashing [Cleary'84]
  - bit vector like in google's sparse table
- characteristics:
  - no displacement info
  - memory-efficient
  - fast construction but
  - slow query times
- current research:
  - speed up queries with SIMD
  - overflow table for averaging the loads of the buckets

thank you for watching!