CS 2110 : Hash Tables

A: O(1)
B: O(lg n)
C: O(n)
D: >O(n)

Data Structures for mutable collections

can implement 2 abstractions

1) set of elements e (e,e) or (e, null)
2) set of key/value pairs (k, v)
looked up by key k = map / associative array

	get/search/contains	add/insert	remove/delete
array	? O(n)	O(1)	O(n)
linked list	O(n)	O(1)	O(n)
sorted array	O(lg n)	O(n)	O(n)
balanced BST	O(lg n)	O(lg n)	O(lg n)
hash table	O(1)	O(1)	O(1)

① Direct Address Table

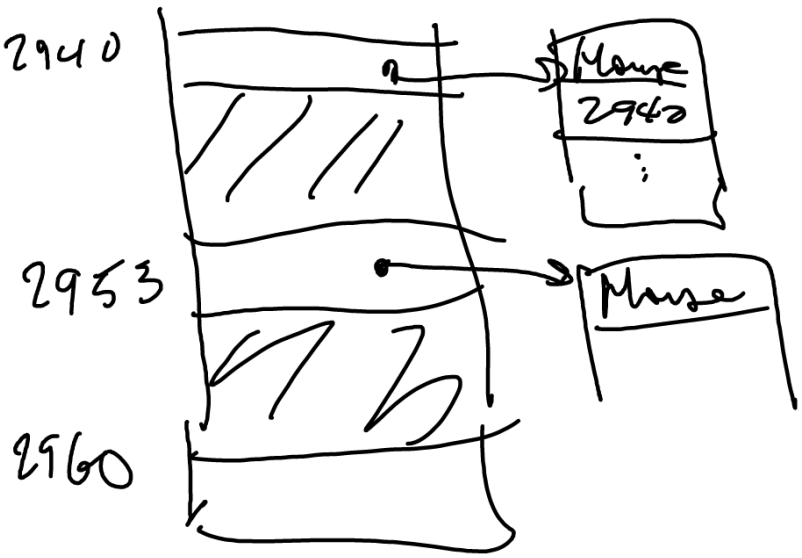
Array: O(1) add/remove if index known

Idea: assume injective function $f : k \to \text{index}$

No collisions where $k_1 \neq k_2$ but $f(k_1) = f(k_2)$

E.g. houses on street

2940, 2953, 2960, ...



But: social security # as index?

Sparse DAT wastes space

(>8G)

② Hash function

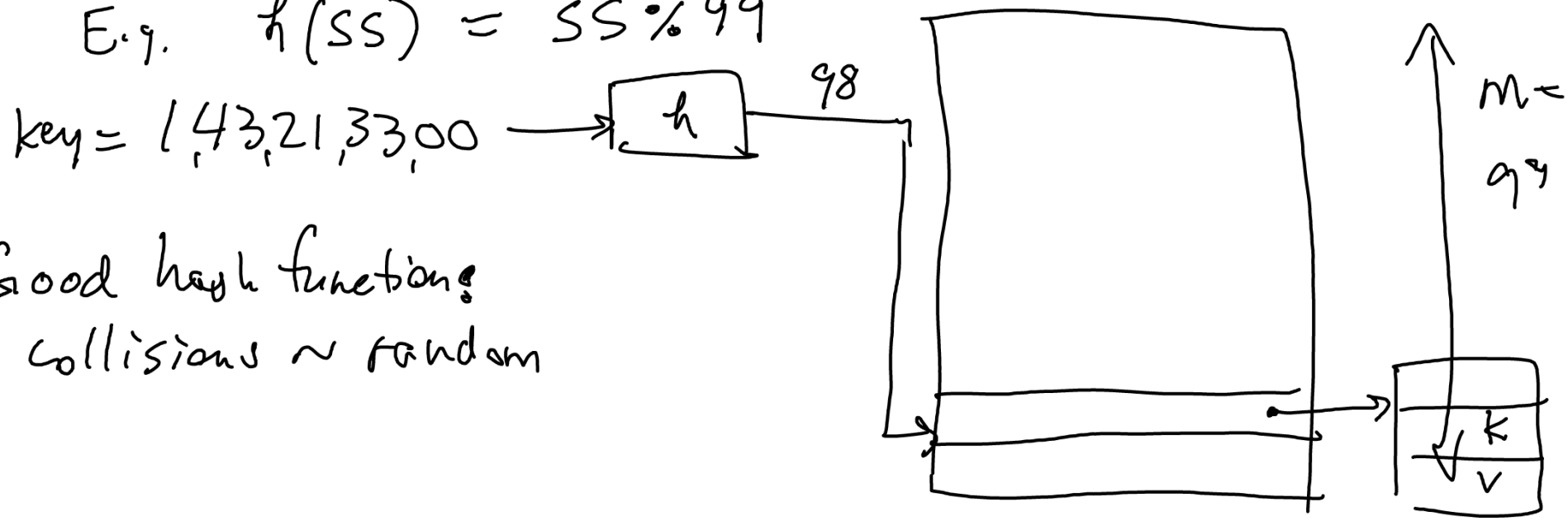
Idea: limit array to size m (# buckets)

hash function: cheap $h: K \longrightarrow [0, m)$

— which avoids collisions

$k_1 \neq k_2 \Rightarrow \text{Prob}(h(k_1) = h(k_2)) \approx 1/m$

E.g. $h(SS) = SS \% 99$



Good hash functions:

collisions ~ random

③ Collision resolution

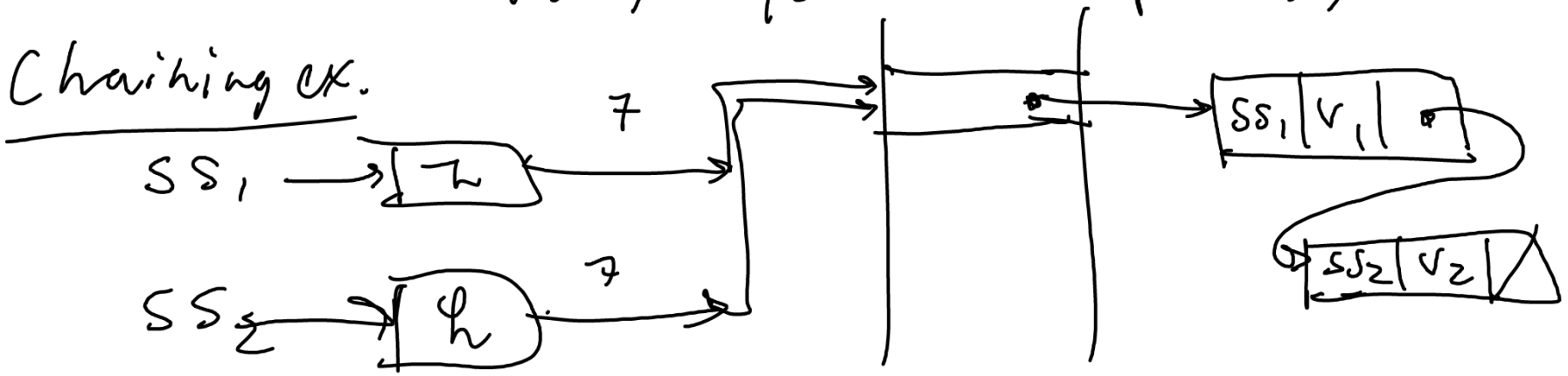
Idea #1: chaining (= open hashing / closed addressing)

- Each bucket stores small linked list of colliding keys

Idea #2: probing (= closed hashing / open addressing)

- On collision, search for empty bucket
  - many ways to search (probe), but usu. slower

Chaining ex.



search: 1. hash key k to bucket i
2. search linked list at buckets[i] for k

add/remove: 1. hash key
2. add/remove from list

Performance

m = # buckets
n = # elements) keys
load factor $\alpha = n/m$
Average list length = α

search for absent key: initial hash $O(1)$
+ traversal $O(\alpha)$
$O(1+\alpha)$

... for present key: hash: $O(1)$
traversal: $O(\frac{1+\alpha}{2})$
$O(1+\alpha)$

For best performance, rule of thumb: $\frac{1}{2} \leq \alpha \leq 1$

too big? long lists

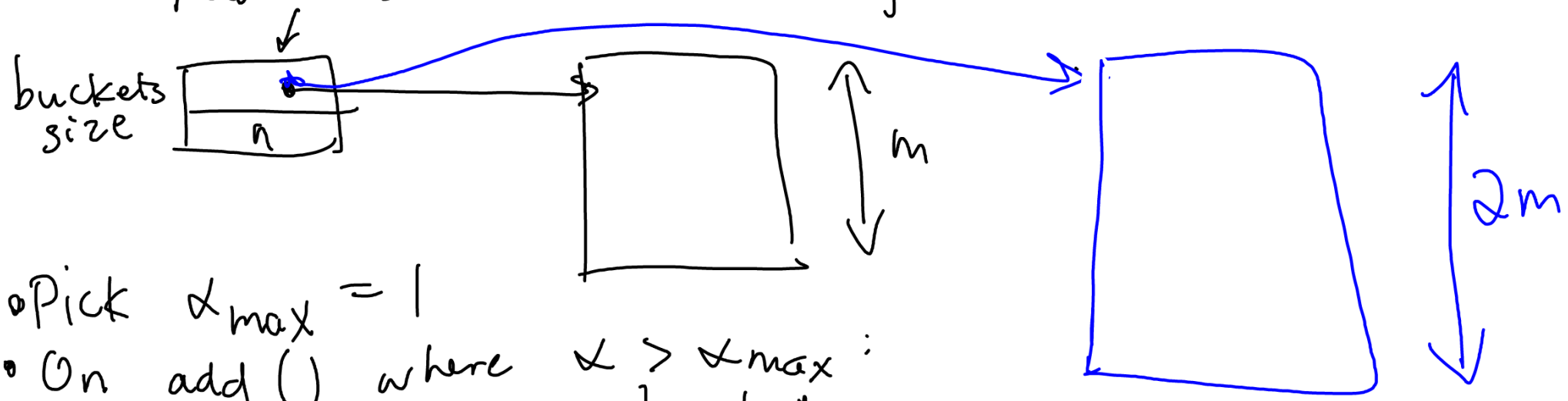
too small? sparse array

How to keep α bounded?

1. preallocate to large m (hard to predict)
2. Resizeble bucket array

④ Resizable Arrays

Hash table has header object



- Pick $\alpha_{max} = 1$
- On add() where $\alpha > \alpha_{max}$:
  - Allocate new bucket array
  - Rehash all elems into new array
  - Update header to point to new array.

Time = $O(n)$ = worst case for 1 operation

But: worst-case time for n operations is $O(n)$

Amortized run time is $O(1)$ per operation

$$\frac{O(n) \text{ time}}{n \text{ operations.}}$$

worst case: $n = 2^j$

$\alpha_{max} = 1$, $m = 1$

rehash/resize at $1, 2, 4, 8, 16, \ldots, 2^j$

total hashes: $2^j + \underbrace{1 + 2 + 4 + 8 + \cdots + 2^j}_{\approx 2^{j+1}}$

initial hashes

$$2^j + 2^{j+1} = 2^j + 2 \cdot 2^j = 3 \cdot 2^j = 3n = O(n)$$

Hash functions

Key: good hash functions

Ideal: acts like it generates indices randomly (but repeatably)

Goal: diffusion: "close" $k_1 \neq k_2$ generate unrelated hashes.

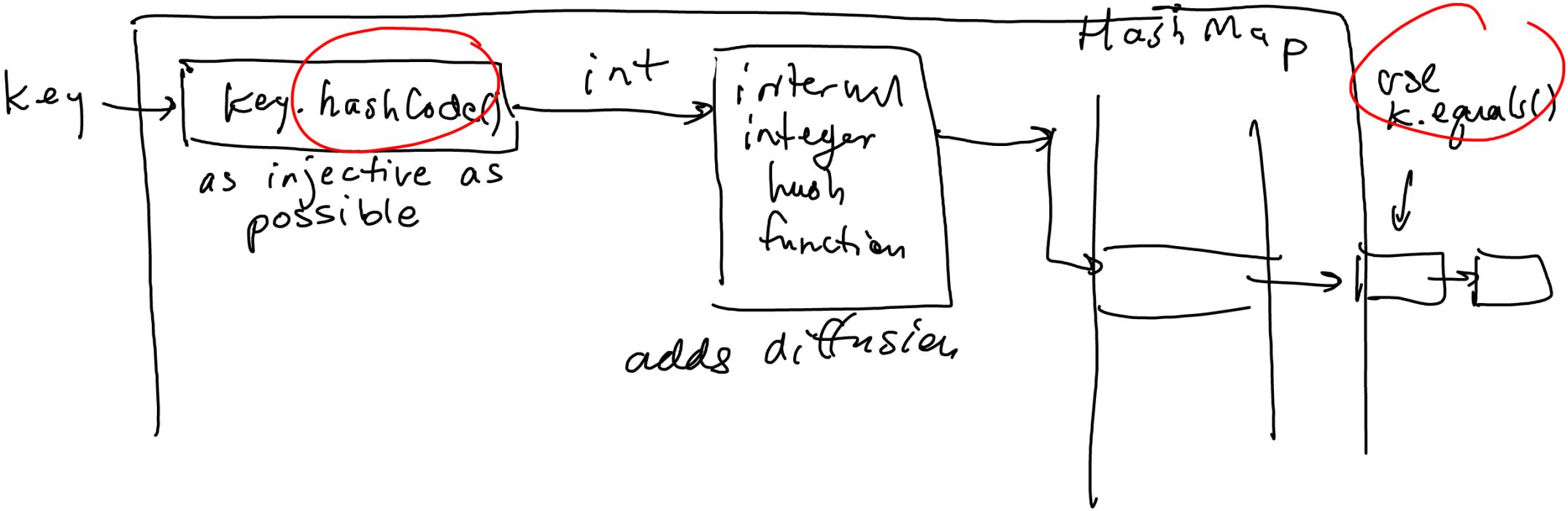
Integer hash function maps int $\to [0, m)$ with diffusion

E.g. modular hashing: $k \mapsto k \bmod m$
but: bad diffusion if $m = 2^i$

Also: CRC, multiplicatives ...

Hash codes

HashMap<K,V> mutable map
HashSet<E> u set.



hashCode() must agree with equals():

- k_1.equals(k_2)
  $\Rightarrow$ k_1.hashCode() == k_2.hashCode()

Object.equals() : pointer equality
Object.hashCode() : use pointer value

} agree

Immutable abstractions?
State equality + hashCode() based on state.
override both