# Strongly connected components

# Hash tables

CS 146 - Spring 2017

# Today

- Tarjan’s algorithm wrap-up
- Hash tables
  - how to hash things well
  - how to handle collisions

```
Stack<Vertex> stack = new Stack();
Map<Vertex, Integer> index = new Map();
Map<Vertex, Integer> lowlink = new Map();
int nextIndex = 0;
for each vertex v
     if (!index.containsKey(v))
          scc(v)


scc(v) {
    // init v
    for every neighbor w of v {
        if (!index.containsKey(w)) { // tree edge
            scc(w)
            lowlink.put(v, min(lowlink.get(v), lowlink.get(w)))
        } else if (w on stack) { // non-tree edge
            lowlink.put(v, min(lowlink.get(v), index.get(w)))
        }
    }
    // pop current scc of v if done
}
```

Tarjan's algorithm

```
Set visited = new Set();
for each vertex v
     if (!visited.contains(v))
          dfs(v)

dfs(v) {
    visited.add(v)
    for every neighbor w of v
        if (!visited.contains(w
            dfs(w)
}
```

DFS

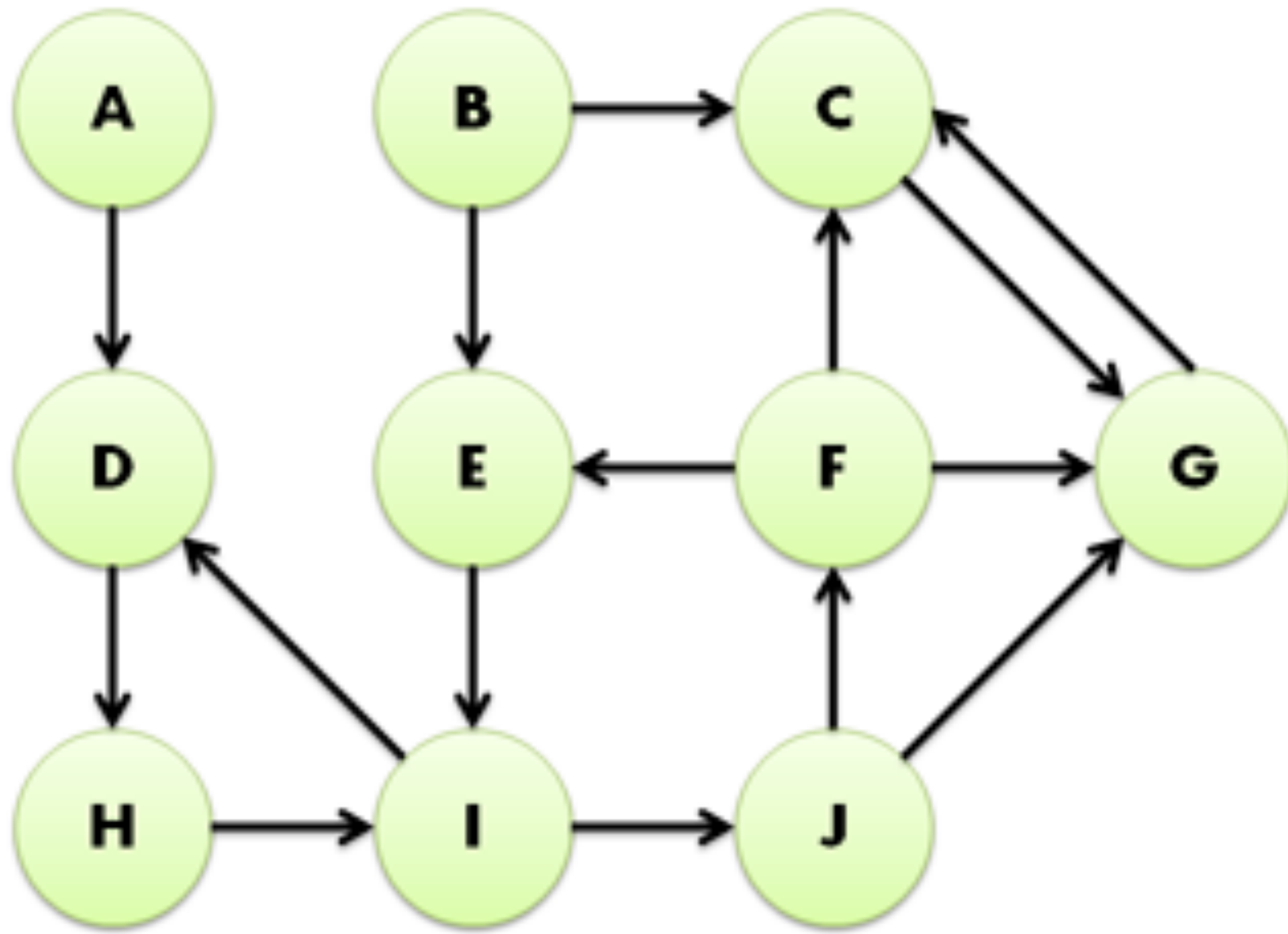
# Tarjan’s algorithm

```
Stack<Vertex> stack = new Stack();
Map<Vertex, Integer> index = new Map();
Map<Vertex, Integer> lowlink = new Map();
int nextIndex = 0;
for each vertex v
     if (!index.containsKey(v))
          scc(v)

scc(v) {
    // init v
    for every neighbor w of v {
        if (!index.containsKey(w)) { // tree edge
             scc(w)
             lowlink.put(v, min(lowlink.get(v), low
        } else if (w on stack) { // non-tree edge
             lowlink.put(v, min(lowlink.get(v), ind
        }
    }
    // pop current scc of v if done
}
```

```
stack.push(v);
index.put(v, nextIndex);
lowlink.put(v, nextIndex);
nextIndex++;
```

```
if (index.get(v) == lowlink.get(v)) {
   List<Vertex> component = new List();
   while (stack.peek() != v)
       component.add(stack.pop());
   component.add(stack.pop());
   print component;
}
```

# Tarjan's algorithm: example




http://www.cs.rpi.edu/~goldsd/fall2013-csci2300-labs.php

```
Stack<Vertex> stack = new Stack();
Map<Vertex, Integer> index = new Map();
Map<Vertex, Integer> lowlink = new Map();
int nextIndex = 0;
for each vertex v
    if (!index.containsKey(v))
        dfs(v)

scc(v) {
    // init v
    for every neighbor w of v {
        if (!index.containsKey(w)) { // tree edge
            scc(w)
            lowlink.put(v, min(lowlink.get(v), low
        } else if (w on stack) { // non-tree edge
            lowlink.put(v, min(lowlink.get(v), ind
        }
    }
    // pop current scc of v if done
}
```

**stack invariant:**
a vertex stays on stack after it
is visited if it has an edge to a vertex
earlier on stack (and visited earlier)

```
stack.push(v);
index.put(v, nextIndex);
lowlink.put(v, nextIndex);
nextIndex++;
```

```
if (index.get(v) == lowlink.get(v)) {
    List<Vertex> component = new List();
    while (stack.peek() != v)
        component.add(stack.pop());
    component.add(stack.pop());
    print component;
}
```

# Tarjan’s algorithm

```
Stack<Vertex> stack = new Stack();
Map<Vertex, Integer> index = new Map();
Map<Vertex, Integer> lowlink = new Map();
int nextIndex = 0;
for each vertex v
     if (!index.containsKey(v))
          dfs(v)


scc(v) {
    // init v
    for every neighbor w of v {
        if (!index.containsKey(w)) { // tree edge
             scc(w)
             lowlink.put(v, min(lowlink.get(v), low
        } else if (w on stack) { // non-tree edge
             lowlink.put(v, min(lowlink.get(v), ind
        }
    }
    // pop current scc of v if done
}
```

**time complexity:**
DFS with additional bookkeeping
problem: “w on stack” not constant

```
stack.push(v);
index.put(v, nextIndex);
lowlink.put(v, nextIndex);
nextIndex++;
```

```
if (index.get(v) == lowlink.get(v)) {
   List<Vertex> component = new List();
   while (stack.peek() != v)
       component.add(stack.pop());
   component.add(stack.pop());
   print component;
}
```

# Tarjan’s algorithm

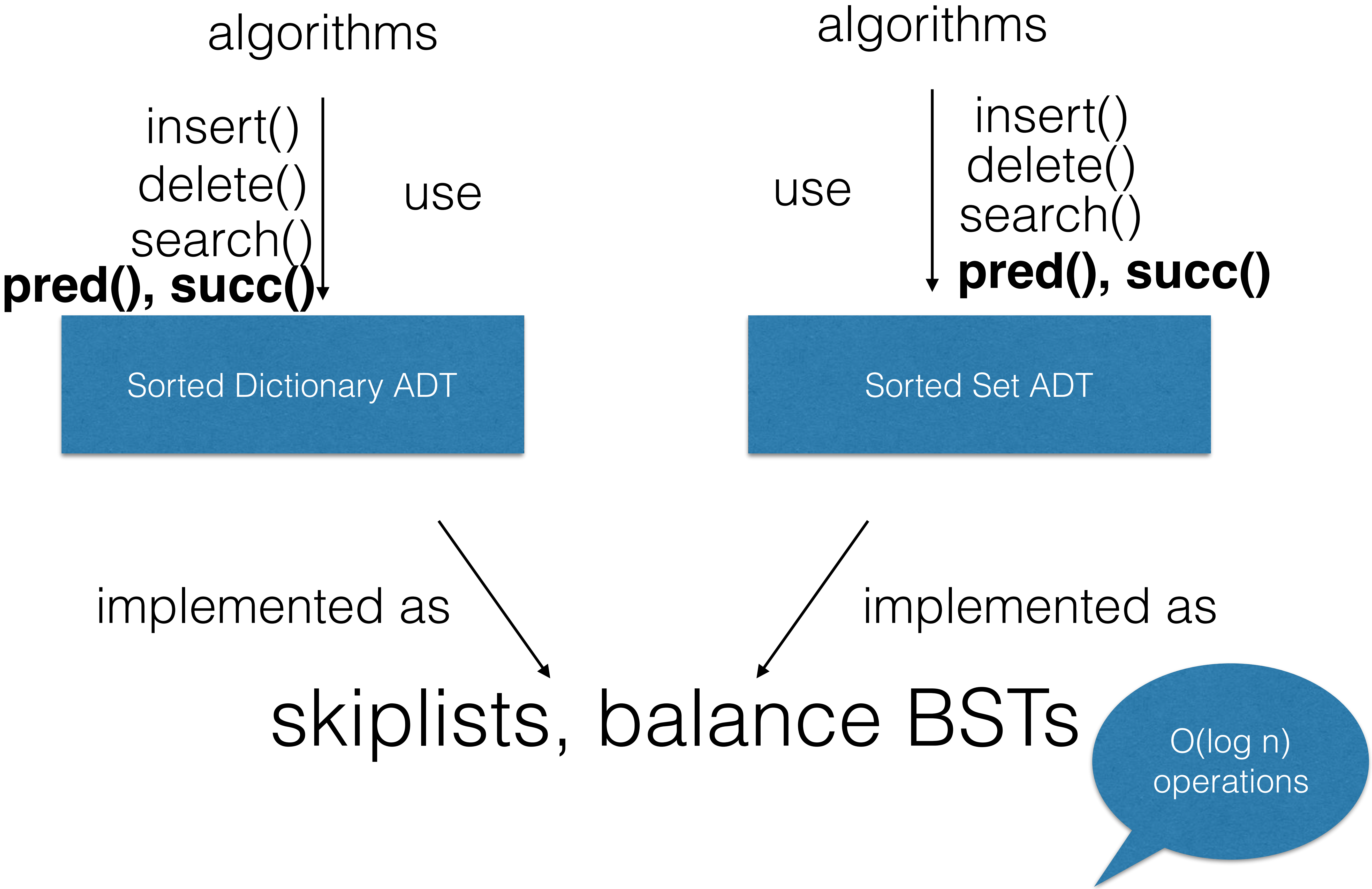
```
Stack<Vertex> stack = new Stack();
Set<Vertex> onStack = new Map();
Map<Vertex, Integer> index = new Map();
Map<Vertex, Integer> lowlink = new Map();
int nextIndex = 0;
for each vertex v
     if (!index.containsKey(v))
          dfs(v)

scc(v) {
    // init v
    for every neighbor w of v {
        if (!index.containsKey(w)) { // tree edge
             scc(w)
             lowlink.put(v, min(lowlink.get(v), low
        } else if (onStack.contains(w)) { // non-tr
             lowlink.put(v, min(lowlink.get(v), ind
        }
    }
    // pop current scc of v if done
}
```

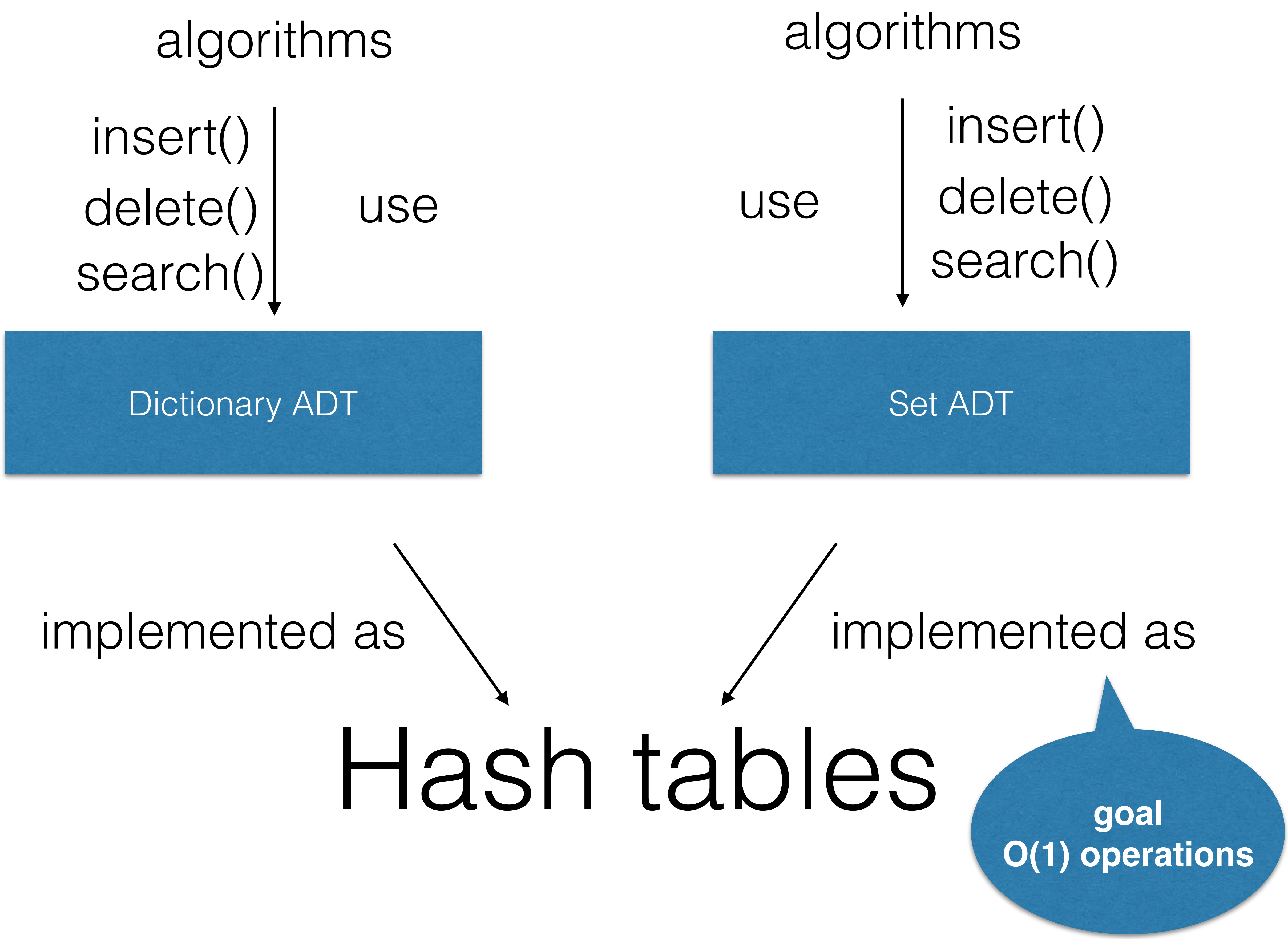
**time complexity:**
fix: use **set** to track what's on the stack
DFS with O(1) bookkeeping is **O(V+E)**

```
stack.push(v); onStack.add(v);
index.put(v, nextIndex);
lowlink.put(v, nextIndex);
nextIndex++;
```

```
if (index.get(v) == lowlink.get(v)) {
   List<Vertex> component = new List();
   while (stack.peek() != v) {
       onStack.remove(stack.peek());
       component.add(stack.pop());
   }
   onStack.remove(stack.peek());
   component.add(stack.pop());
   print component;
}
```

# Tarjan's algorithm


algorithms
insert()
delete()
search()
pred(), succ()
use
Sorted Dictionary ADT
algorithms
use
insert()
delete()
search()
pred(), succ()
Sorted Set ADT
implemented as
implemented as
skiplists, balance BSTs
O(log n)
operations


algorithms
insert()
delete()
search()
use
Dictionary ADT
algorithms
use
insert()
delete()
search()
Set ADT
implemented as
implemented as
Hash tables
goal
O(1) operations

# Simplest approach: direct-address table

- store address of items directly in an array indexed by key

- how page tables are implemented (also need to deal with large space)

- problem: how to deal with keys that are not ints? → **prehashing**

- problem: **large key-range → large space**? → **hashing**

# Pre-hashing

- hash (string type, number type, tuple) → prehash to some int
- object.hashCode() in Java.
- Contract: Equal objects (o.equals(o1) == true) must have equal hashCodes
- Keys in Java maps do not have to be immutable, but certainly not recommended.

# Hashing

- to reduce the universe U of all keys down to a reasonable size m for the table
- idea: m (size of table) should be about n, # keys
- hash function: $h: U \rightarrow \{0, \ldots m-1\}$
- **two keys ki, kj collide if h(ki) = h(kj)**
- problem: how do you minimize collisions? → create good hash functions
- problem: what to do in case of a collision? → collision resolution via chaining, open addressing, cuckoo hashing

# Hashing and collisions

- problem: how do you minimize collisions? → create good hash functions
- problem: what to do in case of a collision? → collision resolution via chaining, open addressing, cuckoo hashing

# Simple uniform hashing

assumption that

each key is equally likely to be hashed to any slot of the table,

independently of where other keys are hashed

(depends on having on having a good hash function, and/or keys already being random)

# Good hash functions scramble things well

ie satisfy the condition of simple uniform hashing

- division method: does not satisfy uniform hashing
- multiplication method
- universal hashing

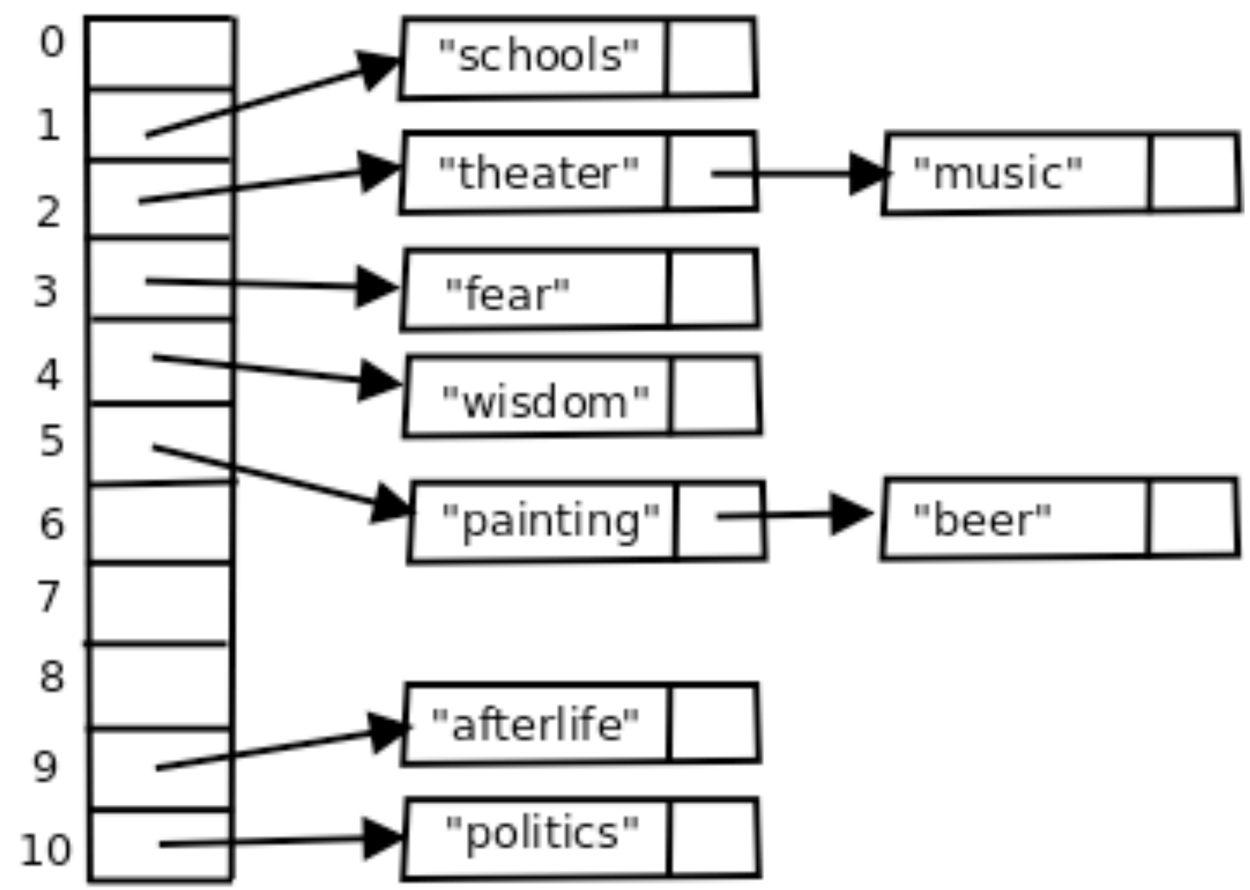
# Multiplication method

- $h(k) = [(ak) \bmod 2^w] >> (w - r)$, where a is chosen at random, and k is w bits
- practical when is a is odd and $2^{w-1} < a < 2^w$, and not too close to either
- fast

# Universal hashing

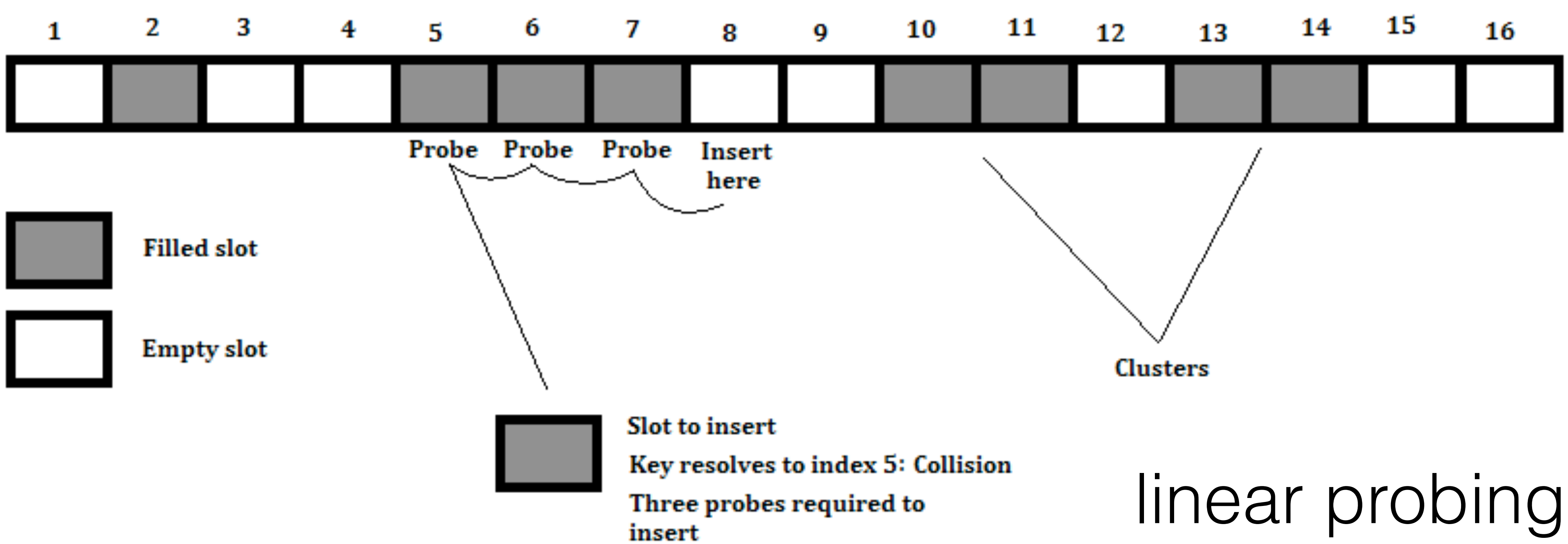
- h(k) = [(ak + b) mod p]  mod m, where a and b are chosen at random, p is a large prime > |U|
- lemma: can prove that for worst-case keys k1 != k2, Pr_{a,b}(h(k1) = h(k2)) = 1/m (proof relies on number theory)
- consequence: E_{a,b}[# collisions with k1] = E[sum_k2 X_k1k2] = sum_k2 E[Xk1k2] = sum_k2 Pr[X_k1k2 = 1] = n/m = alpha

# How do you resolve collisions?




**hash chaining**

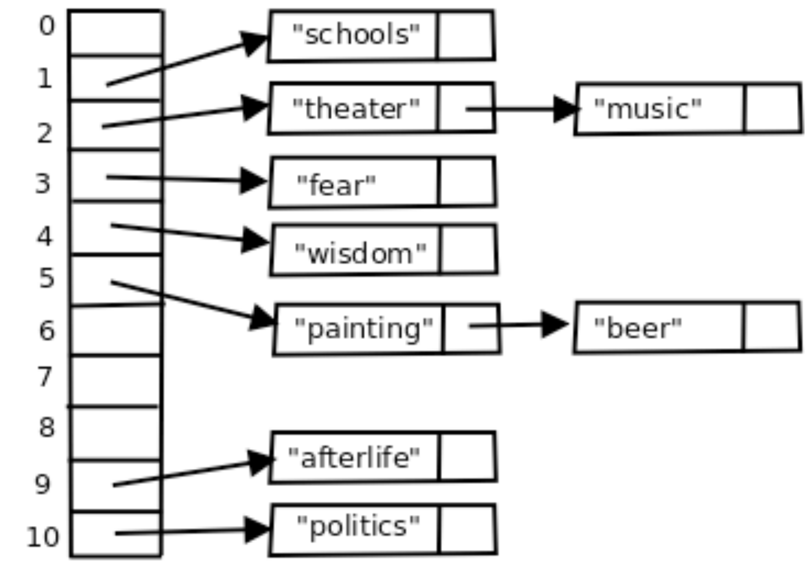
# How do you resolve collisions?




linear probing

**Open addressing**: store values directly in the array

# Hash chaining




- let n = # keys stored in table
- let m = # slots in table
- load factor alpha = n/m = expected # keys per slot = expected length of a chain
- expected running time for search is $\Theta(1+\alpha)$
- for applying hash function and random access to slot + search the list, which is O(1) if $\alpha$ is O(1), for example if m is $\Omega(n)$

# how large should the table be?

- want m = Theta(n) at all times
- don’t know how large n will be at creation itme
- m too small → slow, m too large → wasted space
- idea: start small (constant), grow and shrink as needed

# What does it take to resize the table?

- changing the size of the table m
- changes the hash function (eg h(k) = ak mod m)
- must rebuild the hash table from scratch
- insert each item into new table at a new location
- takes Theta(n + m) time or
  **Theta(n) time** if m = Theta(n)

# How often to grow table?

- if rebuild every time n = m, let m = m + 1. on every insert.
- cost is Theta(1 + 2 + … + n) = Theta(n^2)
- if rebuild every time n = m an let m = 2*m
- cost is Theta(1 + 2 + 4 + 8 + … + n) = Theta(n)
- a few inserts cost linear time, but Theta(1) “on average”

# Amortized analysis

- operation has amortized cost T(n) if k operations cost <= k T(n)
- “T(n) amortized” roughly means T(n) “on average” but average is over all ops
- like paying rent: $1500/month in rent is $50/day

# How often to grow table?

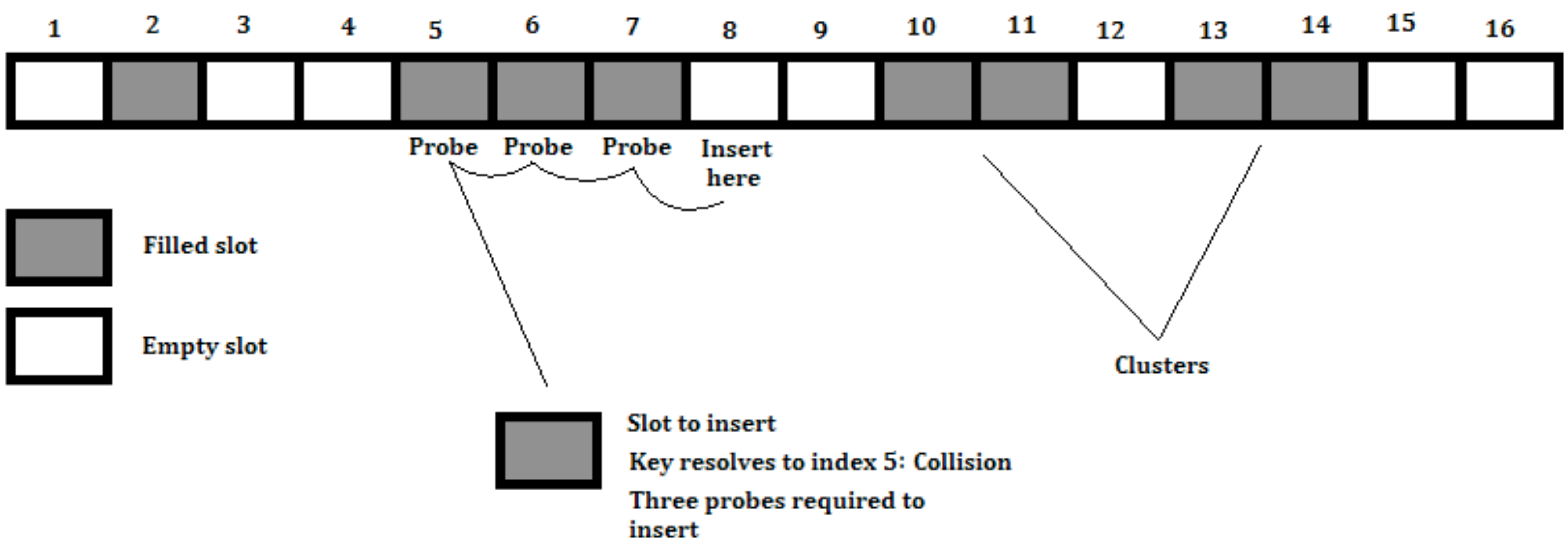
- if rebuild every time n = m an let m = 2*m
- maintain m = Theta(n)
- so alpha = Theta(1),
- **supports search in O(1) expected time, assuming simple uniform hashing or universal**

# How often to shrink table?

- $O(1)$ expected as is
- space can get too big with respect to n eg n inserts, n deletes
- solution: when n decreases to m/4, shrink to half m → 1/2m
- $O(1)$ amortized cost for both inserts and deletes
- analysis is harder (see CLRS 17.4)

# Open addressing to resolve collisions

key value pairs are **stored inside the table**




**a lot of different types**

linear probing

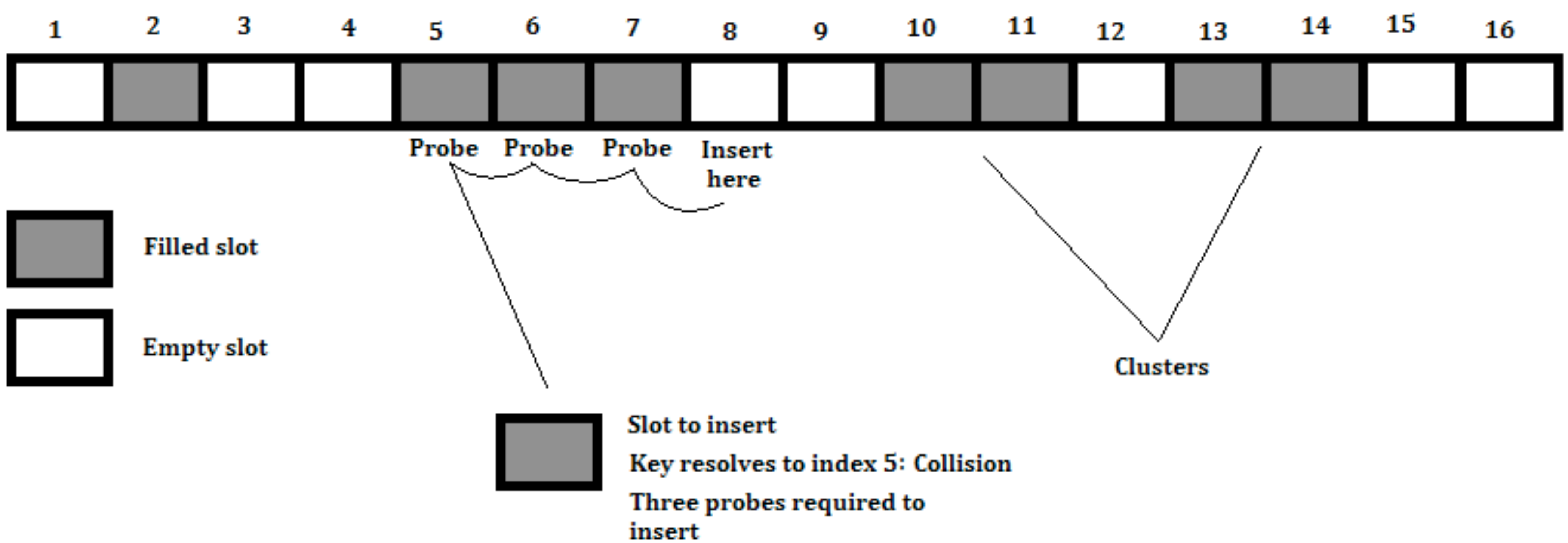
double hashing

quadratic probing

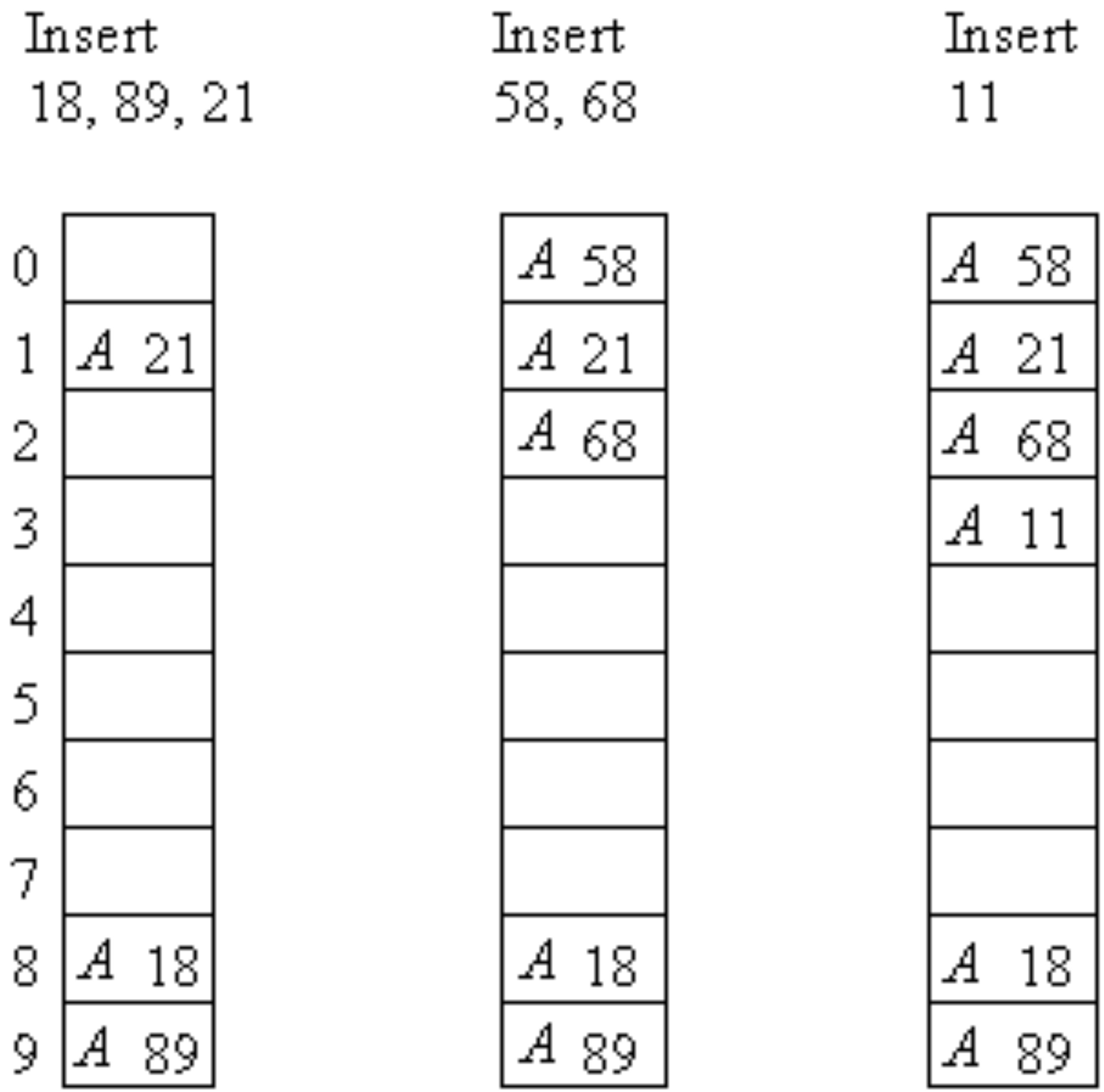
cuckoo hashing

# Linear probing

- **to insert k,v**
- go to slot at h(k), if filled, go to slot at h(k) + 1, etc.
- store at first encountered slot that is empty

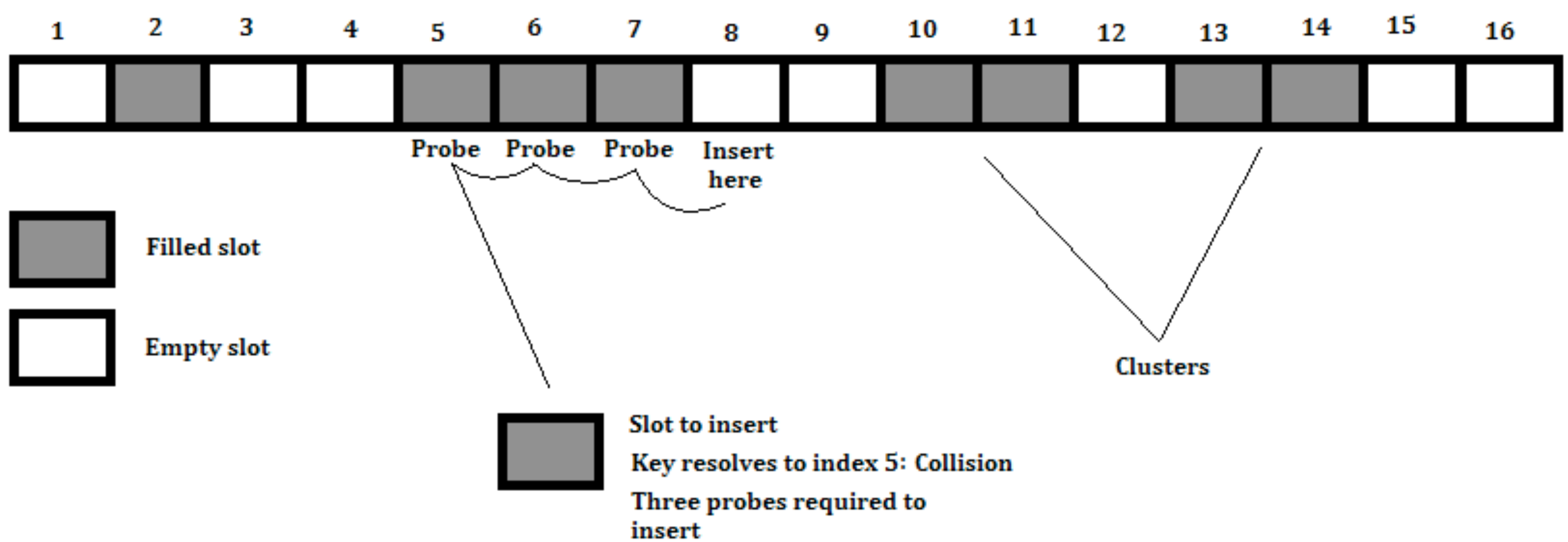


assume h(x) = x % 10




http://condor.depaul.edu/ntomuro/courses/417/notes/lecture4.html

# Linear probing: search for value with key k




- scan each slot starting at h(k) (wrap around)
  - if empty, done, item not there
  - if key at that slot matches k, done, item there

# Open addressing schemes in general

hash function specifies order of slots to probe for a key (for insert/search/delete)

$h(k, 0)$, $h(k, 1)$, $h(k, 2)$, …

- linear probing, uses an auxiliary hash function h'

$$h(k, i) = h'(k) + i \mod m$$

- quadratic probing, uses an auxiliary hash function h'

$$h(k, i) = h'(k) + c1\ i + c2\ i^2 \mod m$$

- double hashing, 2 auxiliary hash functions h1, h2

$$h(k, i) = h1(k) + i\ h2(k) \mod m$$

# How to delete key 73?

| k | h(k) |
|---|---|
| 3 | 6 |
| 8 | 2 |
| 16 | 4 |
| 20 | 3 |
| 34 | 2 |
| 52 | 6 |
| 73 | 2 |

hash function

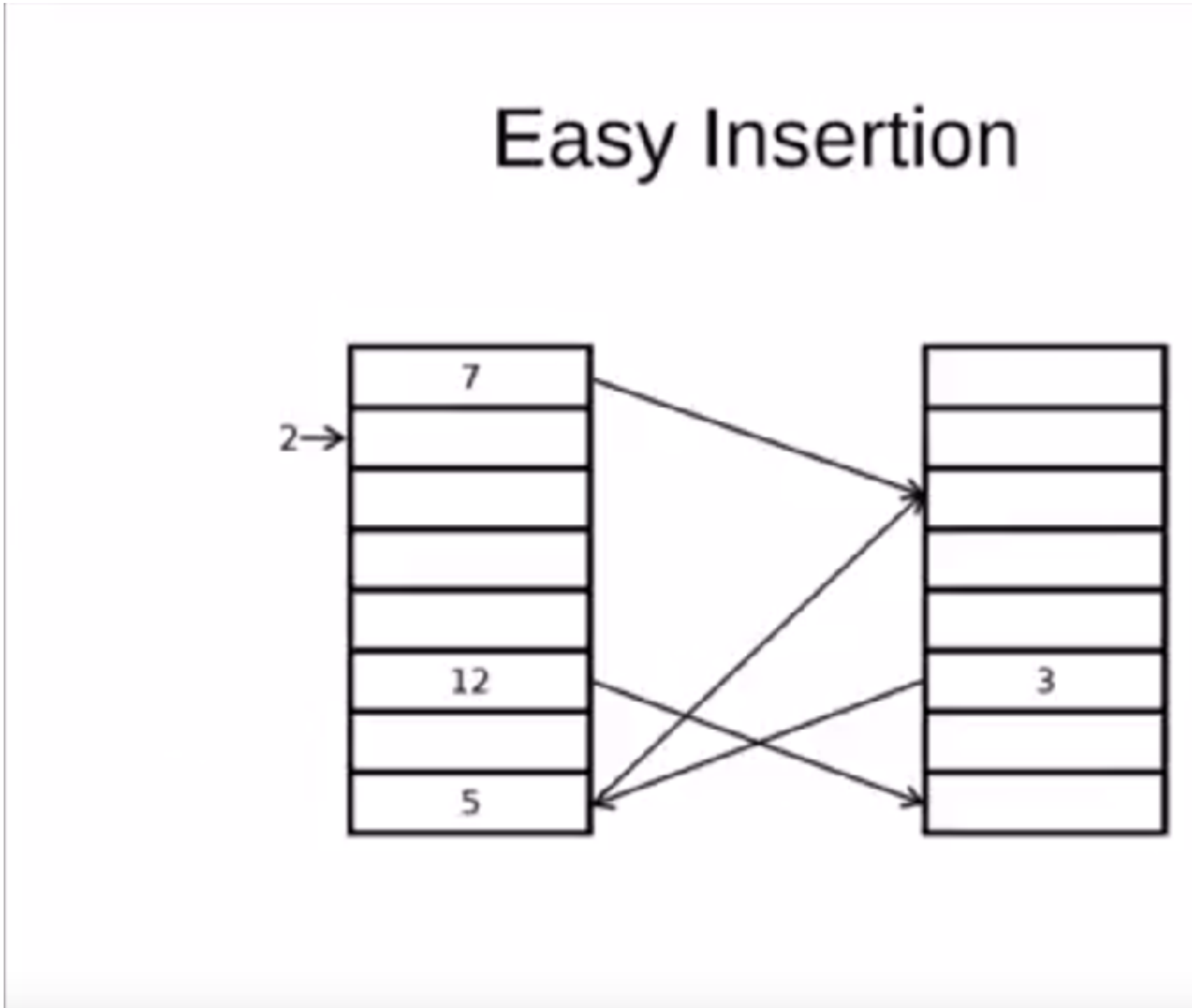
assume linear probing

| index | table |
|---|---|
| 0 | 3, bob |
| 1 | 20, alice |
| 2 | 8, cat |
| 3 | 73, doc |
| 4 | 34, denis |
| 5 | 16, dude |
| 6 | 52, elf |

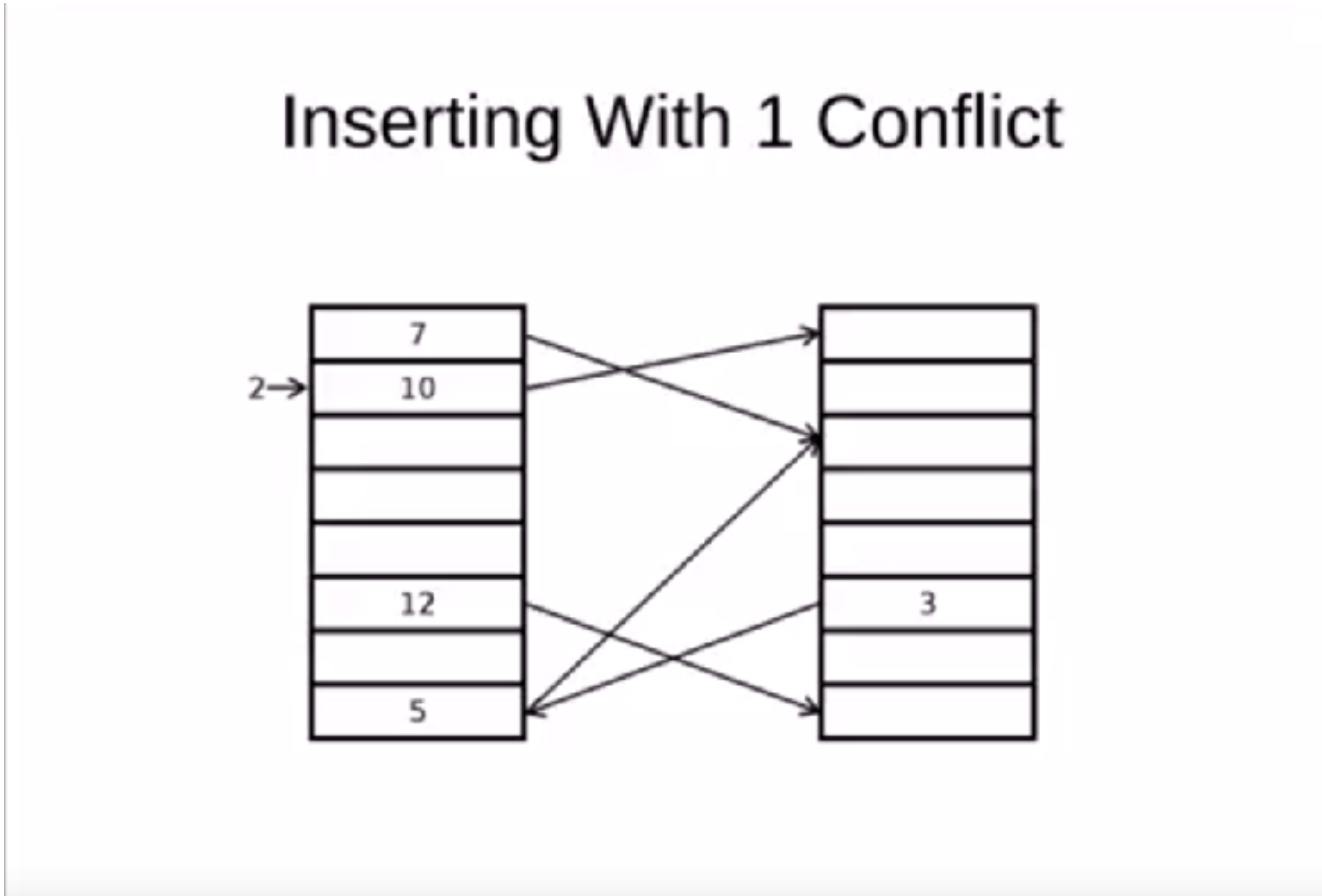
# Deletion

- don’t empty the slot
- mark it with special flag “delete me”
- when searching, treat “delete me” as a full slot
- when inserting, treat “delete me” as an empty slot

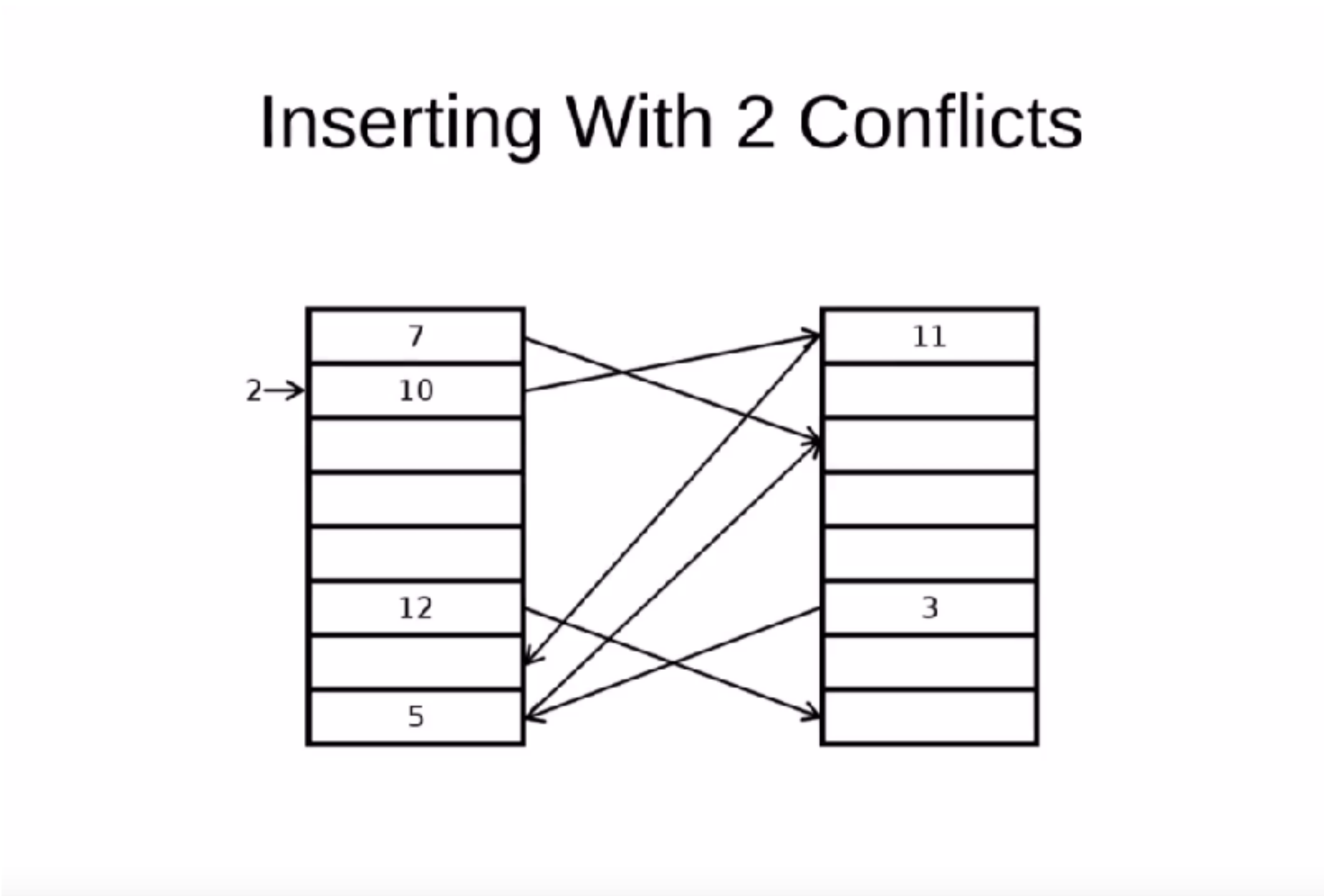
# Cuckoo hashing (FYI)



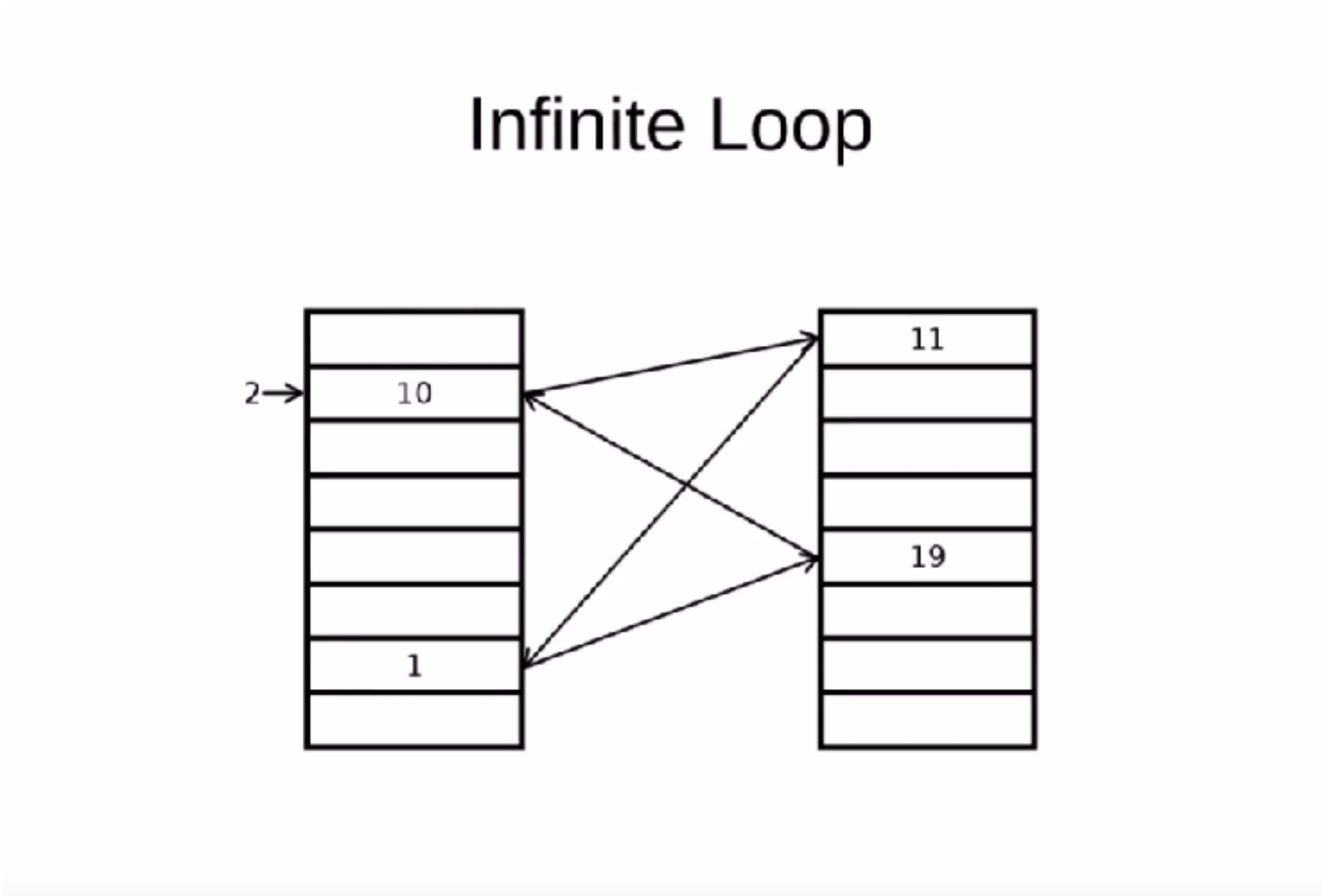

# Cuckoo hashing



# Cuckoo hashing



# Cuckoo hashing



# Cuckoo hashing

- Pagh and Rodler, 2001
- expected amortized O(1)
- O(1) lookups in worst case, rather than expected case

# Open addressing analysis

- clustering when load factor is high

▪ under uniform hashing assumption, next op has
**expected cost of <= 1/ (1 - alpha)**

where alpha = n/m

▪ eg alpha = 90%, 10 expected probes

# Open addressing vs chaining

- open addressing:
  - better cache performance (stored contiguously)
  - sensitive to hash functions: extra care to avoid clustering
  - sensitive to load factor: degrades above 70%
- chaining: less sensitive to hash functions
  - less efficient storage, pointers needed
  - less sensitive to load factors, still O(1)

# Hash tables - the summary

- how to achieve O(1) search, insert, delete ?
- good hash functions
  - simple uniform hashing assumption
  - multiplication method
  - universal hashing
- handling collisions
  - hash chaining
  - open addressing: linear/quadratic probing, cuckoo hashing

# a hash function is **not** a random function

- it’s a function randomly chosen in a family of functions,
- eg choosing the parameter at random, but once chosen, it’s deterministic, running the same function again and again will yield the same value