CS 240 – Data Structures and Data Management

Module 7: Dictionaries via Hashing

A. Hunt and O. Veksler

Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Winter 2023

Outline

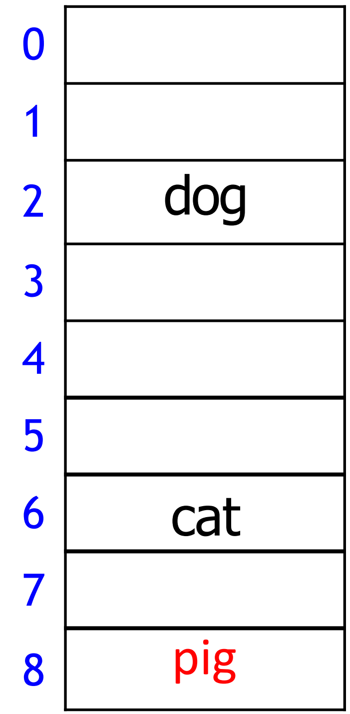
- Dictionaries via Hashing
  - Hashing Introduction
  - Hashing with Chaining
  - Open Addressing
    - probe sequences
    - cuckoo hashing
  - Hash Function Strategies

Outline

- Dictionaries via Hashing
    - Hashing Introduction
    - Hashing with Chaining
    - Open Addressing
        - probe sequences
        - cuckoo hashing
    - Hash Function Strategies

Direct Addressing

- Special situation: every key k is integer with $0 \leq k < M$
- Direct addressing implementation (similar to *Bucket Sort*)
  - store (k, v) in array A of size M *via* $A[k] \leftarrow v$
  - *search*(k): check if $A[k]$ is empty
  - *insert*(k, v): $A[k] \leftarrow v$



$D = \{(2, \text{dog}), (6, \text{cat})\}$

insert$(8, \text{pig})$

Direct Addressing

- Special situation: every key k is integer with $0 \leq k < M$
- Direct addressing implementation (similar to *Bucket Sort*)
  - store (k, v) in array A of size M *via* $A[k] \leftarrow v$
  - *search*(k): check if $A[k]$ is empty
  - *insert*(k, v): $A[k] \leftarrow v$
  - *delete*(k): $A[k] \leftarrow$ *empty*

0	
1	
2	
3	
4	
5	
6	cat
7	
8	pig

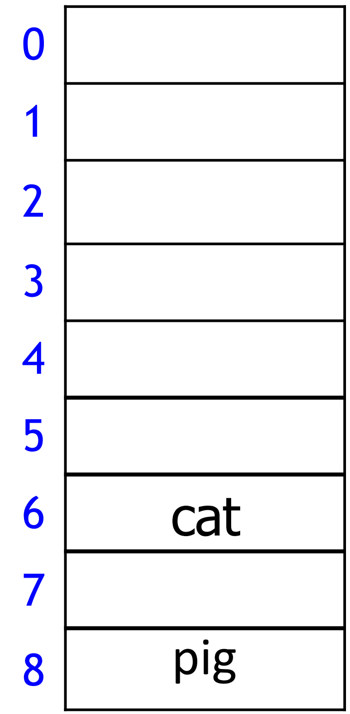
$D = \{(2, \text{dog}), (6, \text{cat}), (8, \text{pig})\}$

$delete(2)$

Direct Addressing

- Special situation: every key k is integer with $0 \leq k < M$
- Direct addressing implementation (similar to *Bucket Sort*)
  - store (k, v) in array A of size M *via* $A[k] \leftarrow v$
  - *search*(k): check if $A[k]$ is empty
  - *insert*(k, v): $A[k] \leftarrow v$
  - *delete*(k): $A[k] \leftarrow$ *empty*
  - all operations are $O(1)$
  - total storage is $\Theta(M)$

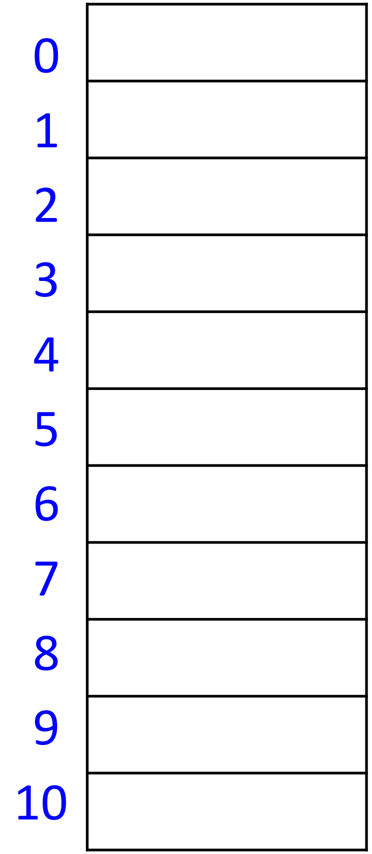
- Drawbacks
  1. space is wasteful if $n << M$
  2. keys must be integers



$D = \{(6, \text{cat}), (8, \text{pig})\}$

Hashing

- **Idea:** first map keys to small integer range and then use direct addressing
- **Assumption:** keys come from some *universe U*
  - typically $U = \{0,1, \ldots\}$, sometimes U is finite
- Design *hash function* $h : U \rightarrow \{0, 1, \ldots, M - 1\}$
  - $h(k)$ is called *hash value* of k
  - example: $h(k) = k \; mod \; M$
  - will see other choices later
- Store dictionary in array T of size M, called *hash table*
- Item with key k usually stored in $T\,[h(k)]$
  - $h(k)$ is called a *slot*
- Example
  - $U = N, \; M = 11, \; h(k) = k \; mod \; 11$
  - keys $7, 13, 43, 45, 49, 92$



Hashing

- **Idea:** first map keys to small integer range and then use direct addressing
- **Assumption:** keys come from some *universe U*
  - typically $U = \{0,1,\ldots\}$, sometimes U is finite
- Design *hash function* $h : U \rightarrow \{0,1,\ldots,M-1\}$
  - $h(k)$ is called *hash value* of k
  - example: $h(k) = k \bmod M$
  - will see other choices later
- Store dictionary in array T of size M, called *hash table*
- Item with key k usually stored in $T[h(k)]$
  - $h(k)$ is called a *slot*
- Example
  - $U = N,\ \ M = 11,\ h(k) = k \bmod 11$
  - keys $7, 13, 43, 45, 49, 92$
  - as usual, store KVP, but show only keys

0	
1	45
2	13
3	
4	92
5	49
6	
7	7
8	
9	
10	43

Hash Functions and Collisions

- Hash function
  - should be fast, $O(1)$, to compute
- Generally hash function h is not injective
  - many keys can map to the same integer, example
    - $h(k) = k \; mod \; 11,$
    - $h(46) = 2 = h(13)$
- *Collision*: want to insert (k, v), but $T[h(k)]$ is occupied
- Two main strategies to deal with collisions
  1. Chaining: allow multiple items at each table location
  2. Open addressing: alternative slots in array
     - probe sequence: many alternative locations
     - cuckoo hashing: just one alternative location

0	
1	45
2	13
3	
4	92
5	49
6	
7	7
8	
9	
10	43

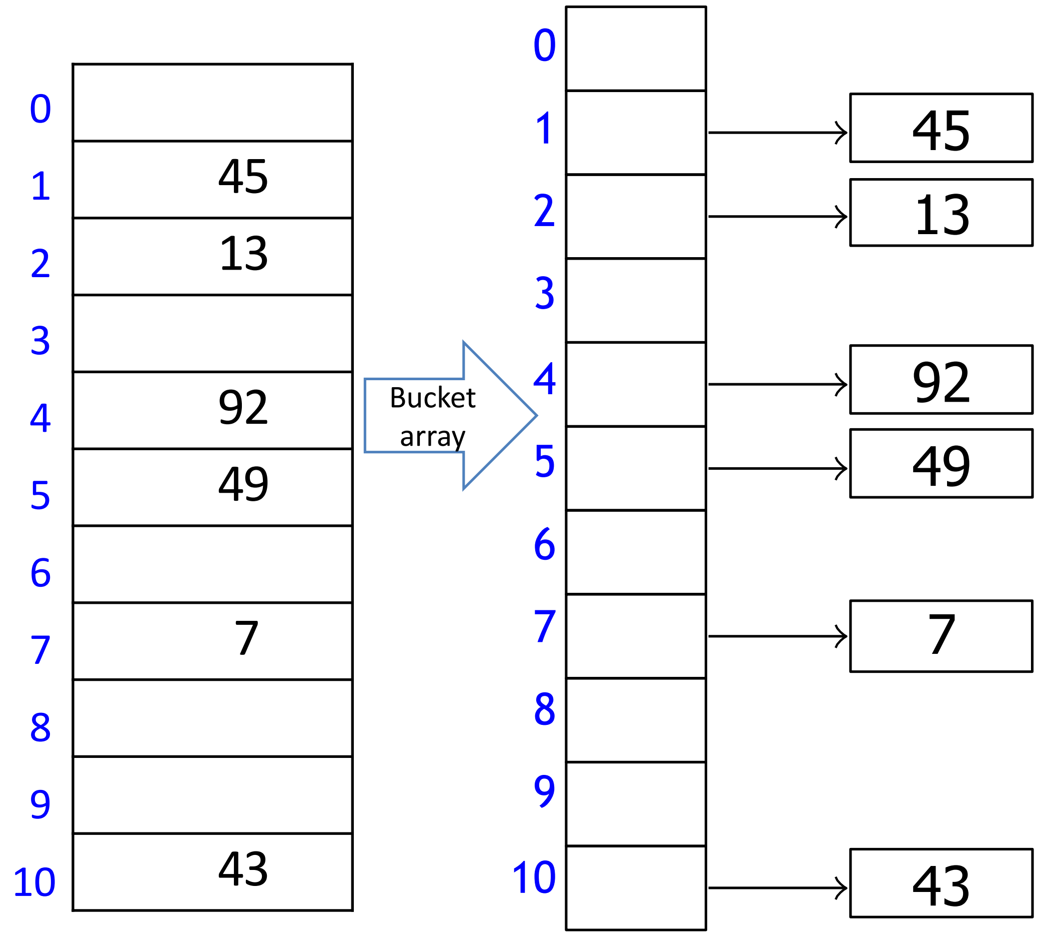
Outline

- Dictionaries via Hashing
  - Hashing Introduction
  - Hashing with Chaining
  - Open Addressing
    - probe Sequences
    - cuckoo hashing
  - Hash Function Strategies

Hashing with Chaining

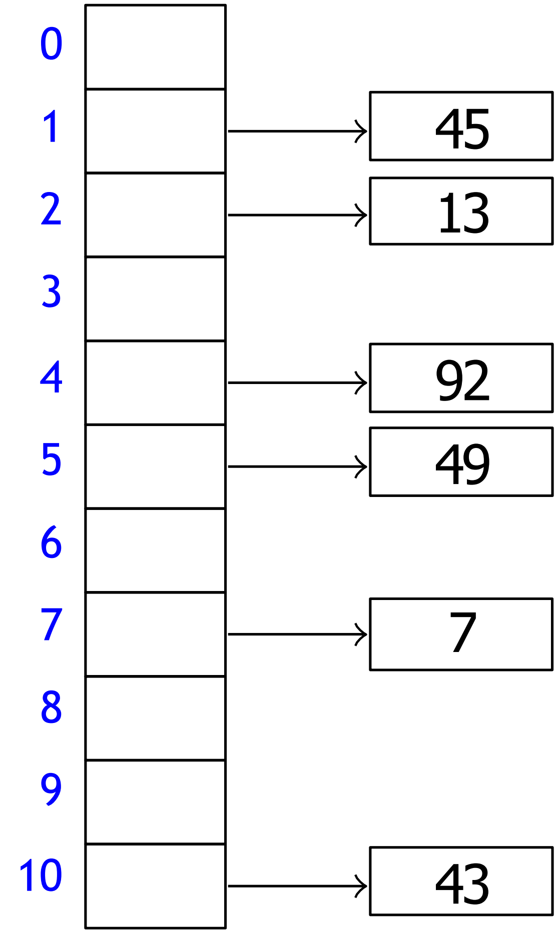
$$M = 11, h(k) = k \bmod 11$$

- Each slot is a *bucket* containing 0 or more KVPs
  - bucket can be implemented by any dictionary
  - even another hash table
  - simplest approach is unsorted linked list in each bucket
    - this is called chaining



Hashing with Chaining

- Operations
  - *search*(k): look for key k in the list at $T\,[h(k)]$
    - apply MTF heuristic
  - *insert*(k, v): add (k, v) to the list at $T\,[h(k)]$
    - add to the list front
  - *delete*(k): search and delete from the list at $T[h(k)]$

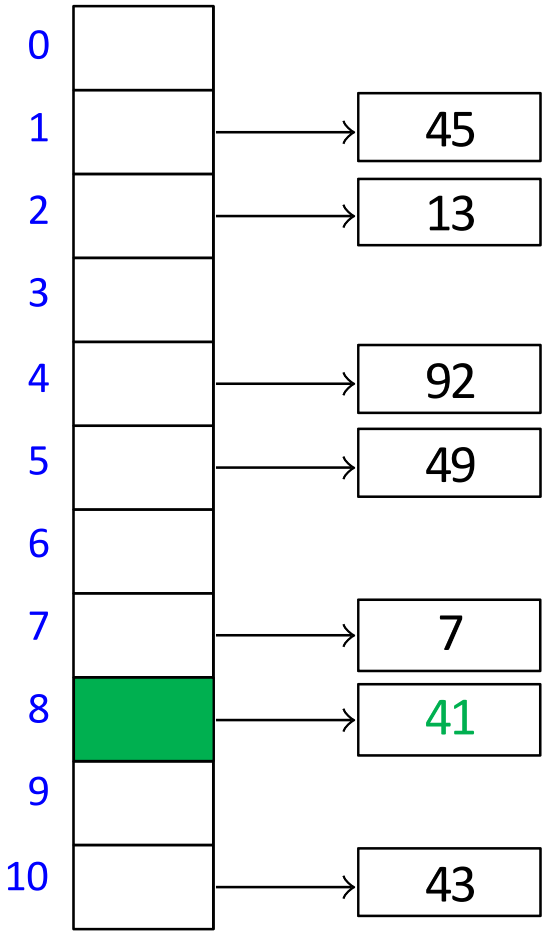


Hashing with Chaining Example

$$M = 11, h(k) = k \, mod \, 11$$

insert(41)

$h(41) = 8$

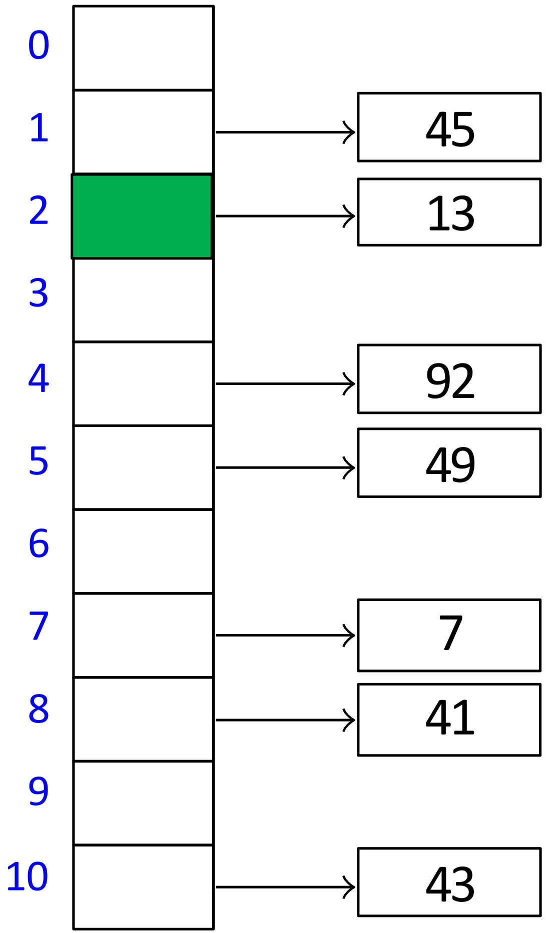


Hashing with Chaining Example

$$M = 11, h(k) = k \bmod 11$$

insert(46)

$h(46) = 2$

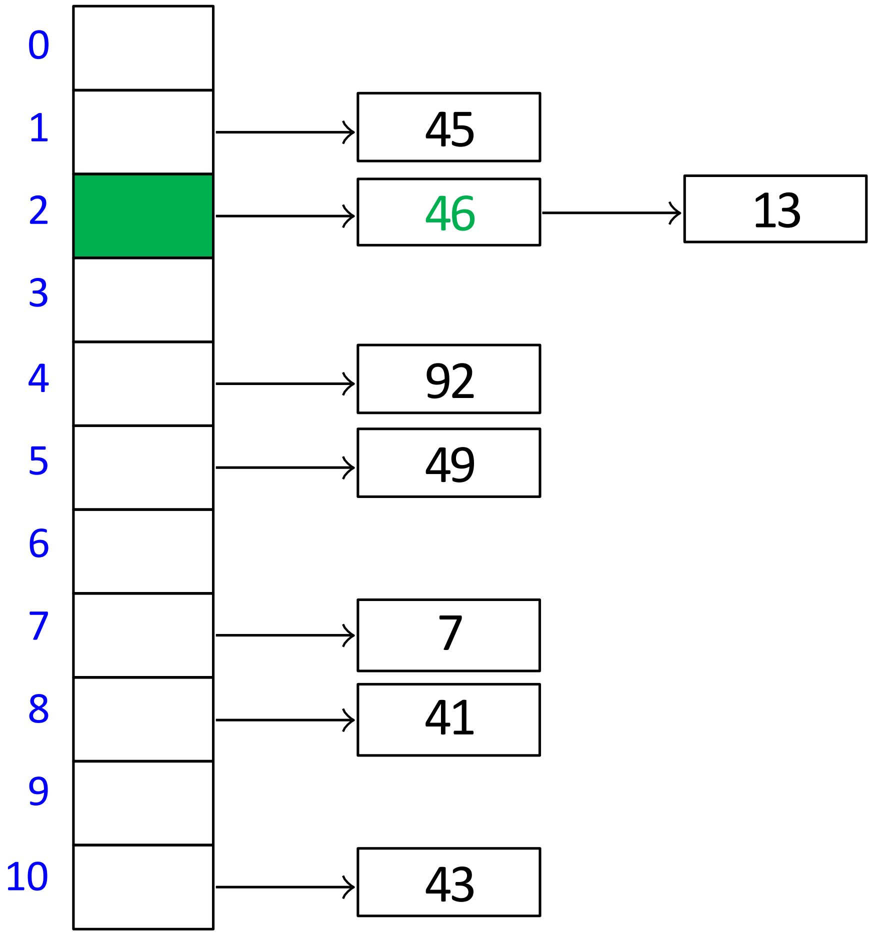


Hashing with Chaining Example

$$M = 11, h(k) = k \bmod 11$$

insert(46)

$h(46) = 2$

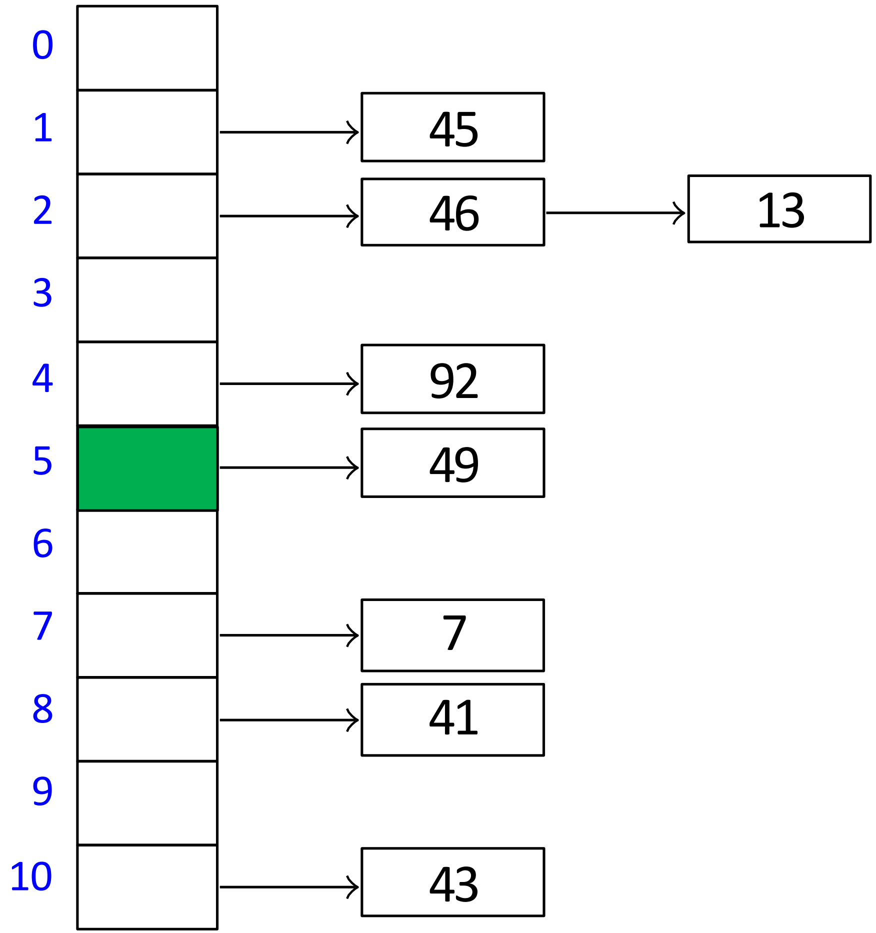


Hashing with Chaining Example

$$M = 11, h(k) = k \bmod 11$$

insert(16)

$h(16) = 5$

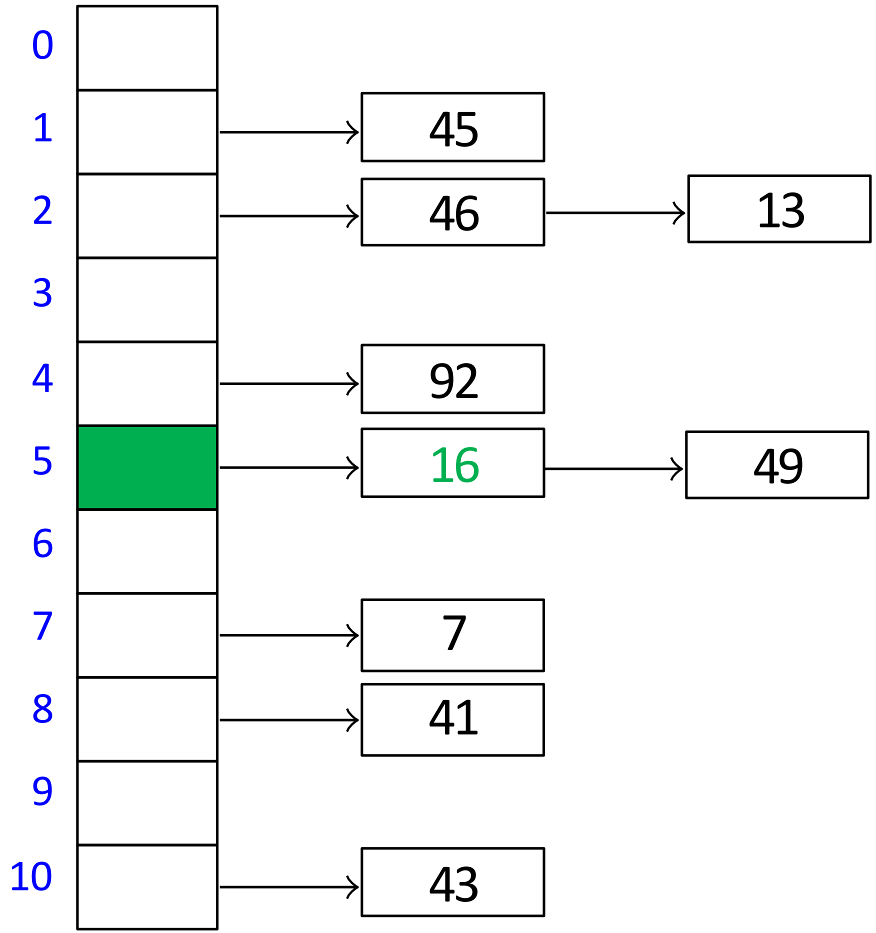


Hashing with Chaining Example

$$M = 11, h(k) = k \, mod \, 11$$

insert(16)

$h(16) = 5$

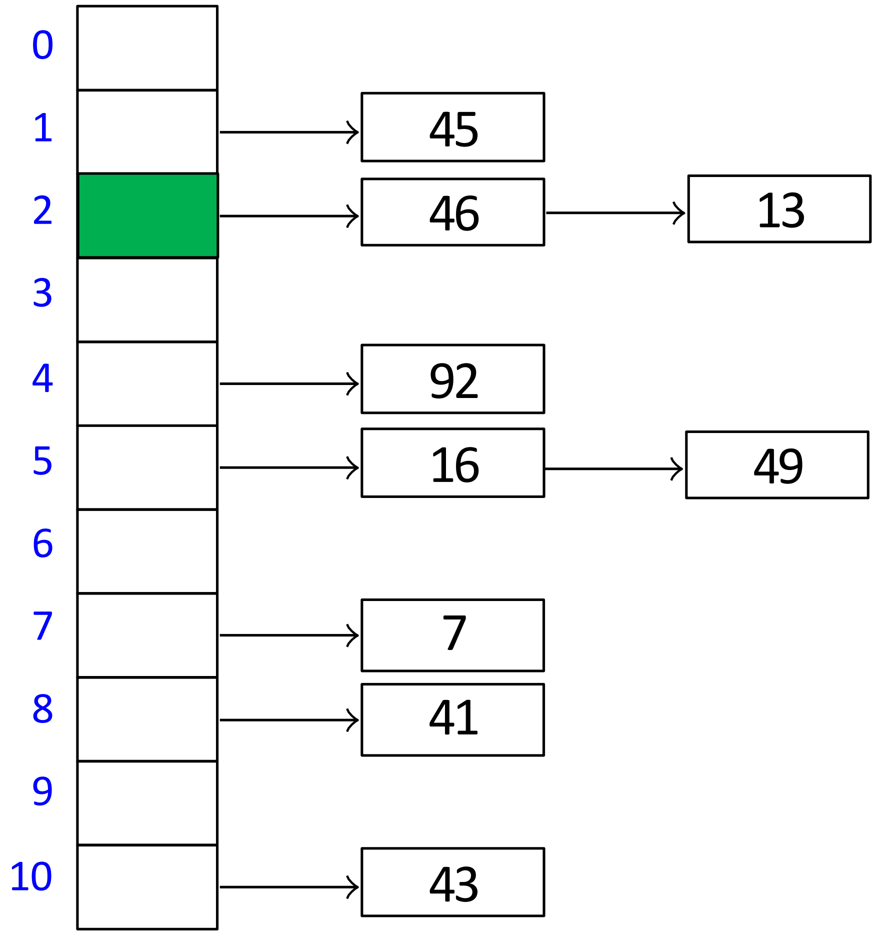


Hashing with Chaining Example

$$M = 11, h(k) = k \bmod 11$$

insert(79)

$h(79) = 2$

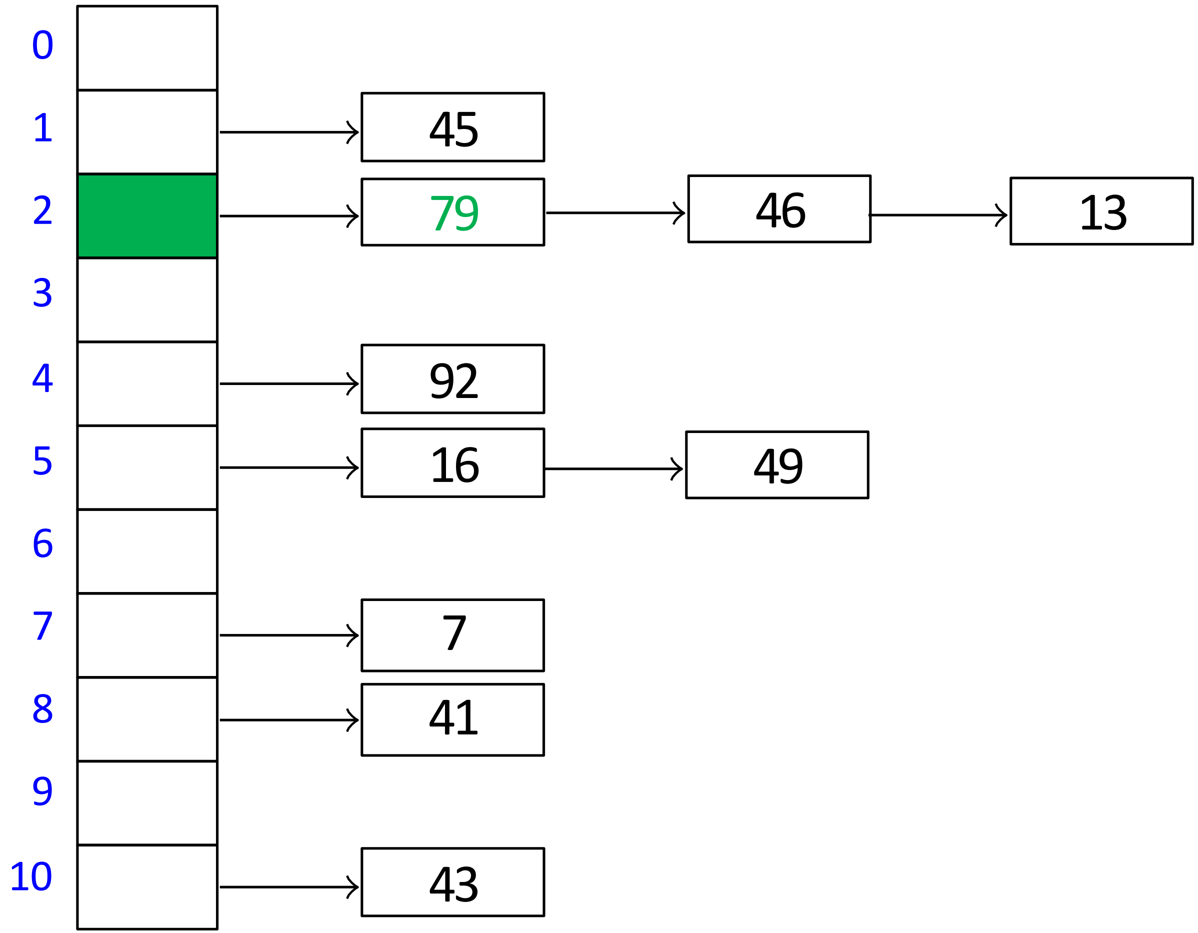


Hashing with Chaining Example

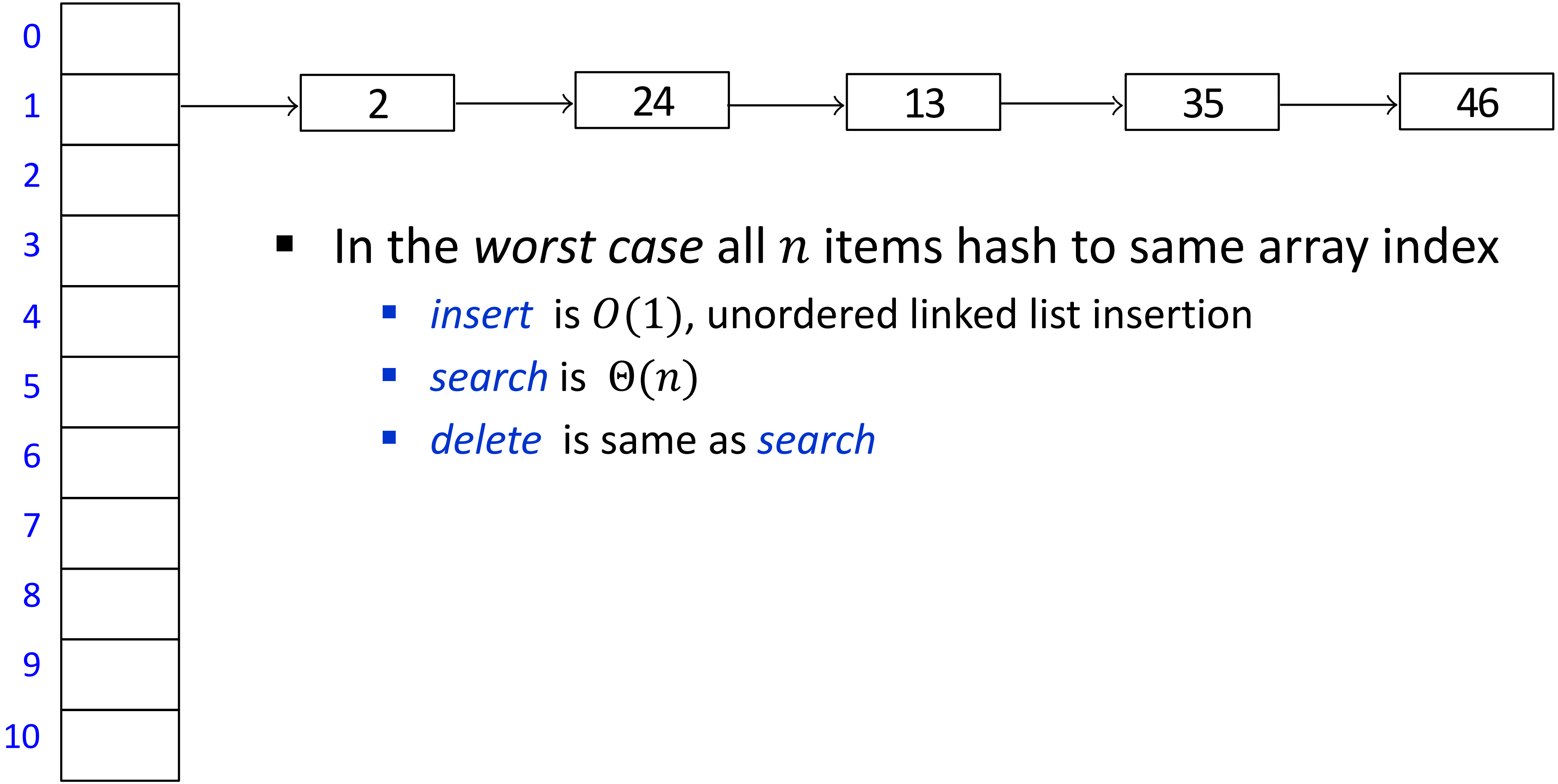
$$M = 11, h(k) = k \bmod 11$$

insert(79)

$h(79) = 2$



Hashing with Chaining: Worst Case Running Time



- In the *worst case* all n items hash to same array index
  - *insert* is $O(1)$, unordered linked list insertion
  - *search* is $\Theta(n)$
  - *delete* is same as *search*

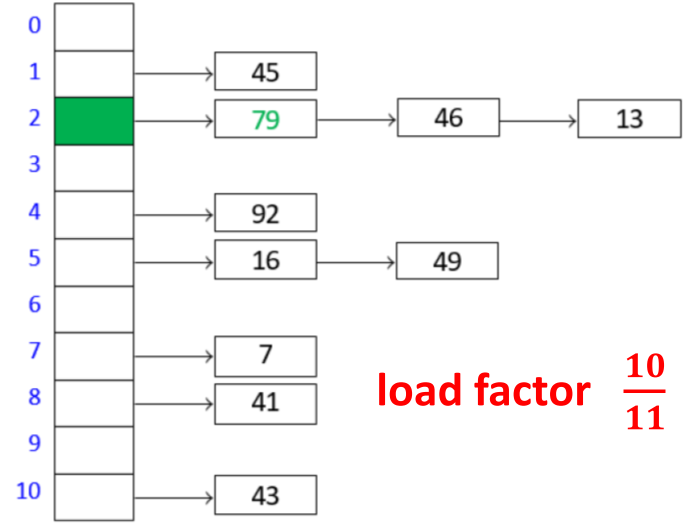
Hashing with Chaining: Worst Case Running Time

- When can all n items hash to the same array index?
  - bad hash function, i.e. $h(k) = 10$
  - for any hash function, if universe is large enough, there are n keys that will hash to the same slot
    - let $|U| \geq M(n-1)+1$
    - suppose less than n keys hash to each table slot

$$\begin{array}{ccccccc} \mathbf{0} & & & & & & \boldsymbol{M-1} \\ \underbrace{\begin{array}{|c|c|c|c|c|c|c|} \hline n-1 & n-1 & n-1 & n-1 & n-1 & n-1 & n-1 \\ \hline \end{array}}_{M(n-1)} \end{array}$$

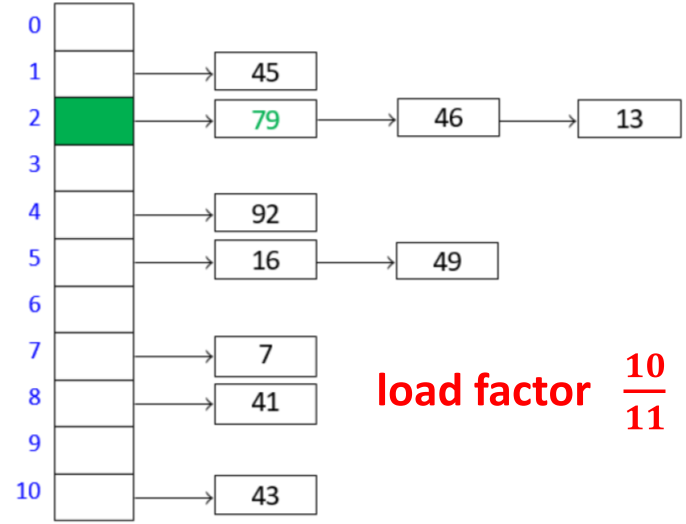
    - then there at most $M(n-1)$ elements in U, contradiction
    - user may or may not decide to insert the items that all hash into the same slot

Hashing with Chaining: Average time?



- Define *load factor* $\alpha = \frac{n}{M}$
  - n is the number of items
  - M is the size of hash table
- *insert* has runtime $\Theta(1)$
- *search*, *delete* have runtime $\Theta(1 + \text{size of bucket } T[h(k)])$
  - note we do not say $\Theta(\text{size of bucket } T[h(k)])$, as bucket can have size 0
    - runtime when bucket size is 0 is $\Theta(1)$, not $\Theta(0)$

Hashing with Chaining: Average time?



- Define *load factor* $\alpha = \frac{n}{M}$
  - n is the number of items
  - M is the size of hash table
- *insert* has runtime $\Theta(1)$
- *search*, *delete* have runtime $\Theta(1 + \text{size of bucket } T[h(k)])$
- The average bucket size is α
- This does not imply that the average-case cost of search and delete is $\Theta(1 + \alpha)$
  - then all keys hash to the same slot, then the average bucket size is still α, but *search*, *delete* still take $\Theta(n)$ on average
- Need to make some assumptions on how keys are distributed
  - too hard to make assumptions close to realistic
- Easier to make assumptions if we switch to randomization and expected time

Hashing with Chaining: Randomization

- Switch to randomized hashing
- How can we randomize?
  - sequence of insert/search/delete is given
  - key must hash to the particular value given by the hash function
- Idea: assume hash-function is chosen randomly
- *Uniform Hashing Assumption*
  - any possible hash-function is equally likely to be chosen
  - not realistic, but this assumption makes analysis possible
- Can show that under uniform hashing assumption
  - $P(h(k) = i) = \frac{1}{M}$ for any key k and slot i
  - hash-values of any two keys are independent of each other
- Practical way to chose a random hash function from a certain family of hash functions
  - $h(k) = \big((ak + b) \bmod p\big) \bmod M$
  - prime number $p > M$ and *random* $a, b \in \{0, \ldots p-1\}$, $a \neq 0$

Hashing with Chaining: Randomization

- $P(h(k) = i) = \frac{1}{M}$ for any key k and slot i
- hash-values of any two keys are independent of each other
- *load factor* $\alpha = \frac{n}{M}$

Claim: for any key k, the expected size of bucket $T[h(k)]$ is at most $1 + \alpha$

Proof:

- Let $h(k) = i$
- Case 1: k is not in the dictionary
  - then each of n dictionary items hashes to i with probability $\frac{1}{M}$
  - $E[T(i)] = \frac{n}{M} = \alpha \leq 1 + \alpha$
- Case 2: k is in the dictionary
  - $T(i)$ definitely has key k
  - the remaining n-1 dictionary items hash to i with probability $\frac{1}{M}$
  - $E[T(i)] = 1 + \frac{n-1}{M} \leq 1 + \alpha$

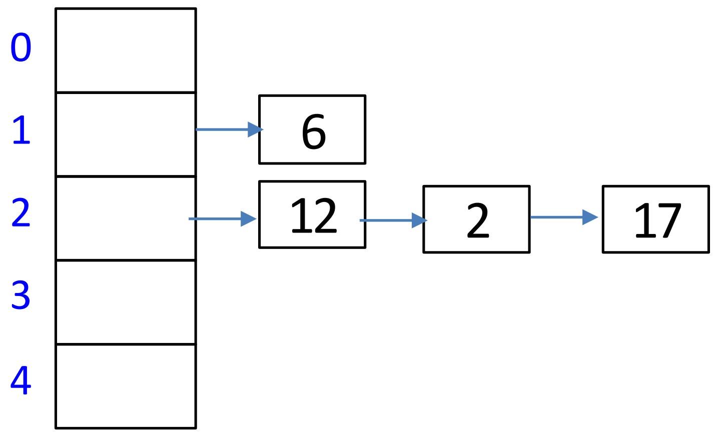
- *search*, *delete* have runtime $\Theta(1 + \text{size of bucket } T[h(k)])$
- Expected runtime of search and delete is $\Theta(1 + \alpha)$, insert is $\Theta(1)$

Load factor and re-hashing

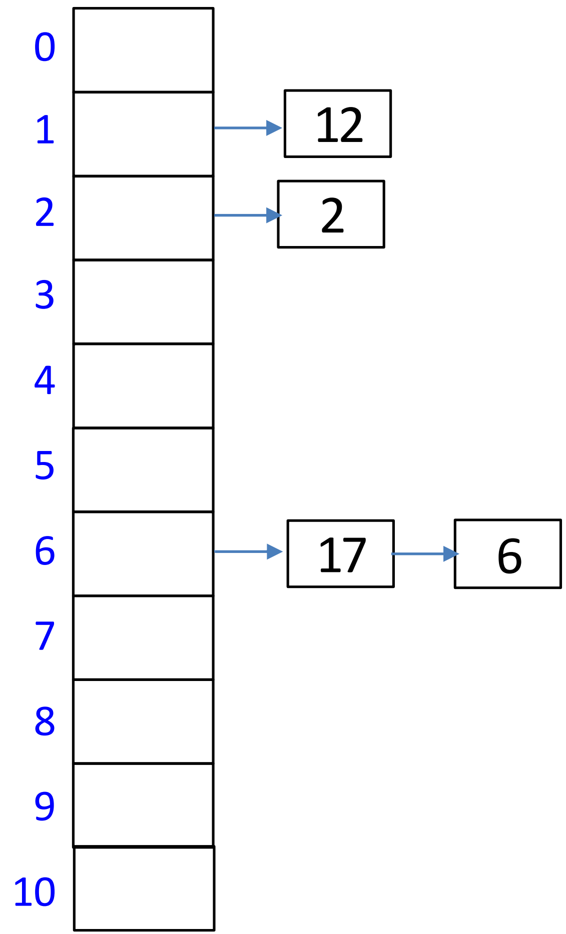
- Load factor $\alpha = \frac{n}{M}$
- *Space* is $\Theta(M + n) = \Theta(n/\alpha + n)$, time is $\Theta(1 + \alpha)$
  - if we maintain $\alpha \in \Theta(1)$, expected running time is $O(1)$ and space is $\Theta(n)$
- Accomplished by rehashing whenever $\frac{n}{M} < c_1$ or $\frac{n}{M} > c_2$
  - where c_1, c_2 are constants with $0 < c_1 < c_2$
  - c_1 is minimum allowed load factor, c_2 is maximum allowed load factor
- Maintaining hash array of appropriate size
  - start with small M
  - during insert/delete, update n
  - if load factor becomes too big, i.e. $\alpha = \frac{n}{M} > c_2$, rehash
    - chose new $M' \approx 2M$
    - find a new random hash function h' that maps U into $\{0,1, \ldots M' - 1\}$
    - create new hash table T' of size M'
    - reinsert each KVP from T into T'
    - update $T \leftarrow T', h \leftarrow h'$
  - If load factor becomes too small, i.e. $\alpha = \frac{n}{M} < c_1$, rehash with smaller M'
- Rehashing costs $\Theta(M + n)$ but happens rarely, cost amortized over all operations

Rehashing

$M = 5, h(k) = k \bmod 5$



$M' = 11, h'(k) = k \bmod 11$



Outline

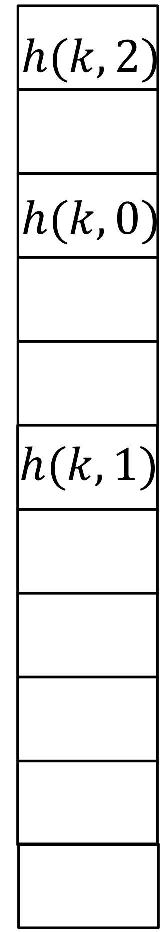
- Dictionaries via Hashing
  - Hashing Introduction
  - Hashing with Chaining
  - Open Addressing
    - probe sequences
    - cuckoo hashing
  - Hash Function Strategies

Open Addressing

- Chaining wastes space on links
- Can we resolve collisions in the array H?
- Idea: each hash table entry holds only one item, but key k can go in multiple locations
- *Probe sequence*
  - *search* and *insert* follow a probe sequence of possible locations for key k

$$h(k,0), h(k,1), h(k,2), \ldots$$

  - until an empty spot is found



Open Addressing: Linear Probing

- Linear probing is the simplest method for probe sequence
  - If $h(k)$ is occupied, place item in the next available location
    - probe sequence is
      - $h(k, 0) = h(k)$
      - $h(k, 1) = h(k) + 1$
      - $h(k, 2) = h(k) + 2$
      - etc…
- Assume circular array, i.e. modular arithmetic
  - $h(k, i) = (h(k) + i) \; mod \; M$

Linear Probing Example

$M = 11,\ h(k) = k \bmod 11$

insert(41)

$h(41) = 8$

0	
1	45
2	13
3	
4	92
5	49
6	
7	7
8	
9	
10	43

Linear Probing Example

$M = 11,\ h(k) = k \bmod 11$

insert(41)

$h(41) = 8$

0	
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	
10	43

Linear Probing Example

$M = 11, \; h(k) = k \bmod 11$

insert(84)

$h(84) = 7$

0	
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	
10	43

Linear Probing Example

$M = 11, \; h(k) = k \bmod 11$

insert(84)

$h(84) = 7$

0		
1	45	
2	13	
3		
4	92	
5	49	
6		
7	7	occupied
8	41	
9		
10	43	

Linear Probing Example

$M = 11,\ h(k) = k \bmod 11$

insert(84)

$h(84) = 7$

0		
1	45	
2	13	
3		
4	92	
5	49	
6		
7	7	occupied
8	41	occupied
9		
10	43	

Linear Probing Example

$M = 11, \; h(k) = k \bmod 11$

insert(84)

$h(84) = 7$

0		
1	45	
2	13	
3		
4	92	
5	49	
6		
7	7	occupied
8	41	occupied
9	84	
10	43	

Linear Probing Formula

- Linear probing explores positions

$$h(k,i) = (h(k) + i) \bmod M$$

  - for $i = 0, 1, \ldots$ until an empty location is found
  - where $h(k)$ is some hash function

Linear probing example Continued

$$M = 11, \; h(k) = k \; mod \; 11$$

$$h(k, i) = (h(k) + i) \text{ mod } M \text{ for sequence } i = 0, 1, \ldots$$

$insert(20)$

$h(20) = 9$

$h(20, 0) = (9 + 0) \text{mod } 11 = 9$

0	
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	84
10	43

Linear probing example Continued

$$M = 11, \; h(k) = k \, mod \, 11$$

$$h(k, i) = (h(k) + i) \bmod M \text{ for sequence } i = 0,1, \ldots$$

$insert(20)$

$h(20) = 9$

$h(20, 0) = (9 + 0) \bmod 11 = 9$

0		
1	45	
2	13	
3		
4	92	
5	49	
6		
7	7	
8	41	
9	84	occupied
10	43	

Linear probing example Continued

$$M = 11, \; h(k) = k \; mod \; 11$$

$$h(k, i) = (h(k) + i) \text{ mod } M \text{ for sequence } i = 0, 1, \ldots$$

insert(20)

$$h(20) = 9$$

$$h(20, 1) = (9 + 1) \text{mod } 11 = 10$$

0		
1	45	
2	13	
3		
4	92	
5	49	
6		
7	7	
8	41	
9	84	occupied
10	43	occupied

Linear probing example Continued

$$M = 11, \; h(k) = k \; mod \; 11$$

$$h(k, i) = (h(k) + i) \text{ mod } M \text{ for sequence } i = 0,1, \ldots$$

$insert(20)$

$h(20) = 9$

$h(20, 2) = (9 + 2) \text{mod } 11 = 0$

0	20	
1	45	
2	13	
3		
4	92	
5	49	
6		
7	7	
8	41	
9	84	occupied
10	43	occupied

Linear probing example: Search

$M = 11, \; h(k) = k \bmod 11$

$h(k, i) = (h(k) + i) \bmod M$ for sequence $i = 0, 1, \ldots$

search(23)

$h(23) = 1$

$h(23, 0) = (1 + 0) \bmod 11 = 1$

0	20	
1	45	occupied
2	13	
3		
4	92	
5	49	
6		
7	7	
8	41	
9	84	
10	43	

Linear probing example: Search

$$M = 11,\ h(k) = k \bmod 11$$

$$h(k,i) = (h(k) + i) \bmod M \text{ for sequence } i = 0,1,\ldots$$

search(23)

$h(23) = 1$

$h(23, 1) = (1 + 1) \bmod 11 = 2$

0	20	
1	45	occupied
2	13	occupied
3		
4	92	
5	49	
6		
7	7	
8	41	
9	84	
10	43	

Linear probing example: Search

$M = 11,\ h(k) = k \bmod 11$

$h(k, i) = (h(k) + i) \bmod M$ for sequence $i = 0,1, \ldots$

search(23)

$h(23) = 1$

$h(23, 2) = (1 + 2) \bmod 11 = 3$

0	20	
1	45	occupied
2	13	occupied
3		not found
4	92	
5	49	
6		
7	7	
8	41	
9	84	
10	43	

Linear probing: Delete

$M = 11, \; h(k) = k \bmod 11$

$h(k, i) = (h(k) + i) \bmod M$ for sequence $i = 0, 1, \ldots$

delete(84)

$h(84) = 7$

$h(84, 0) = (7 + 0) \bmod 11 = 7$

0	20
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	84
10	43

Linear probing: Delete

$$M = 11, \; h(k) = k \; mod \; 11$$

$$h(k, i) = (h(k) + i) \text{ mod } M \text{ for sequence } i = 0, 1, \ldots$$

$delete(84)$

$h(84) = 7$

$h(84, 0) = (7 + 0) \text{mod } 11 = 7$

0	20	
1	45	
2	13	
3		
4	92	
5	49	
6		
7	7	occupied
8	41	
9	84	
10	43	

Linear probing: Delete

$M = 11, \; h(k) = k \; mod \; 11$

$h(k, i) = (h(k) + i) \bmod M$ for sequence $i = 0, 1, \ldots$

$delete(84)$

$h(84) = 7$

$h(84, 1) = (7 + 1) \bmod 11 = 8$

0	20	
1	45	
2	13	
3		
4	92	
5	49	
6		
7	7	occupied
8	41	occupied
9	84	
10	43	

Linear probing: Delete

$$M = 11, \; h(k) = k \bmod 11$$

$$h(k, i) = (h(k) + i) \text{ mod } M \text{ for sequence } i = 0,1, \ldots$$

$delete(84)$

$h(84) = 7$

$h(84, 2) = (7 + 2) \text{mod } 11 = 9$

0	20	
1	45	
2	13	
3		
4	92	
5	49	
6		
7	7	occupied
8	41	occupied
9	84	found
10	43	

Linear probing: Delete

$M = 11, \; h(k) = k \; mod \; 11$

$h(k, i) = (h(k) + i)$ mod M for sequence $i = 0, 1, \ldots$

$delete(84)$

$h(84) = 7$

$h(84, 2) = (7 + 2) \text{mod}\, 11 = 9$

0	20
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	
10	43

Linear probing: Delete

$M = 11, \; h(k) = k \; mod \; 11$

$h(k, i) = (h(k) + i) \bmod M$ for sequence $i = 0,1, \ldots$

search(20)

$h(20) = 9$

$h(20, 0) = (9 + 0) \bmod 11 = 9$

0	20	
1	45	
2	13	
3		
4	92	
5	49	
6		
7	7	
8	41	
9		not found
10	43	

Open Addressing

- *delete* becomes problematic
  - cannot leave an *empty* spot behind
    - next search might otherwise not go far enough
  - Idea: lazy deletion
    - mark spot as *deleted* (rather than *empty*)
    - continue searching past *deleted* spots
    - insert in empty or *deleted* spot

Linear probing: Delete

$$M = 11, \; h(k) = k \; mod \; 11$$

$$h(k, i) = (h(k) + i) \bmod M \text{ for sequence } i = 0,1, \ldots$$

$delete(84)$

$h(84) = 7$

$h(84, 0) = (7 + 0) \text{mod } 11 = 7$

$h(84, 1) = (7 + 1) \text{mod } 11 = 8$

$h(84, 2) = (7 + 2) \text{mod } 11 = 9$

0	20	
1	45	
2	13	
3		
4	92	
5	49	
6		
7	7	occupied
8	41	occupied
9	84	found
10	43	

Linear probing: Delete

$M = 11, \; h(k) = k \; mod \; 11$

$h(k, i) = (h(k) + i) \bmod M$ for sequence $i = 0,1, \ldots$

$delete(84)$

$h(84) = 7$

$h(84, 0) = (7 + 0) \mathrm{mod}\, 11 = 7$

$h(84, 1) = (7 + 1) \mathrm{mod}\, 11 = 8$

$h(84, 2) = (7 + 2) \mathrm{mod}\, 11 = 9$

0	20	
1	45	
2	13	
3		
4	92	
5	49	
6		
7	7	occupied
8	41	occupied
9	deleted	
10	43	

Linear probing example

$M = 11, \; h(k) = k \bmod 11$

$h(k, i) = (h(k) + i) \bmod M$ for sequence $i = 0, 1, \ldots$

search(20)

$h(20) = 9$

$h(20, 0) = (9 + 0) \bmod 11 = 9$

0	20
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	deleted
10	43

occupied

Linear probing example

$$M = 11, \; h(k) = k \; mod \; 11$$

$h(k, i) = (h(k) + i) \bmod M$ for sequence $i = 0, 1, \ldots$

search(20)

$h(20) = 9$

$h(20, 1) = (9 + 1) \bmod 11 = 10$

0	20	
1	45	
2	13	
3		
4	92	
5	49	
6		
7	7	
8	41	
9	84	occupied
10	43	occupied

Linear probing example

$$M = 11, \; h(k) = k \; mod \; 11$$

$h(k, i) = (h(k) + i) \bmod M$ for sequence $i = 0, 1, \ldots$

search(20)

$h(20) = 9$

$h(20, 2) = (9 + 2) \bmod 11 = 0$

0	20	found
1	45	
2	13	
3		
4	92	
5	49	
6		
7	7	
8	41	
9	84	occupied
10	43	occupied

Linear probing example

$$M = 11, \; h(k) = k \bmod 11$$

$$h(k, i) = (h(k) + i) \bmod M \text{ for sequence } i = 0, 1, \ldots$$

insert(10)

$$h(10) = 10$$

$$h(10, 0) = (10 + 0) \bmod 11 = 10$$

0	20
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	84
10	deleted

Linear probing example

$M = 11, \; h(k) = k \; mod \; 11$

$h(k, i) = (h(k) + i) \text{ mod } M$ for sequence $i = 0,1, \ldots$

insert(10)

$h(10) = 10$

$h(10, 0) = (10 + 0) \text{mod } 11 = 10$

0	20
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	84
10	10

Probe Sequence Operations

probe-sequence::insert$(T, (k, v))$

for $(i = 0; i < M; i{+}{+})$

if $T[h(k,i)]$ is *empty* or *deleted*

$T[h(k,i)] = (k, v)$

return *success*

return *failure to insert*

- Stop inserting after M tries
  - provided $\alpha < 1$, linear probing does not need this
  - some probing methods need this
- If insert fails, call rehash

probe-sequence::search$(T, (k, v))$

for $(i = 0; i < M; i{+}{+})$

if $T[h(k,i)]$ is *empty*

return *item-not-found*

if $T[h(k,i)]$ is has key k

return $T[h(k,i)]$

// ignore $T[h(k,i)] =$ deleted and keep searching

return *item not found*

Linear probing drawbacks

- Entries tend to cluster into contiguous regions
  - “snowball” effect
- Many probes for each search, insert, and delete
- How to avoid clustering?

0	
1	45
2	
3	
4	92
5	
6	**28**
7	**7**
8	**41**
9	**84**
10	

Double Hashing Motivation

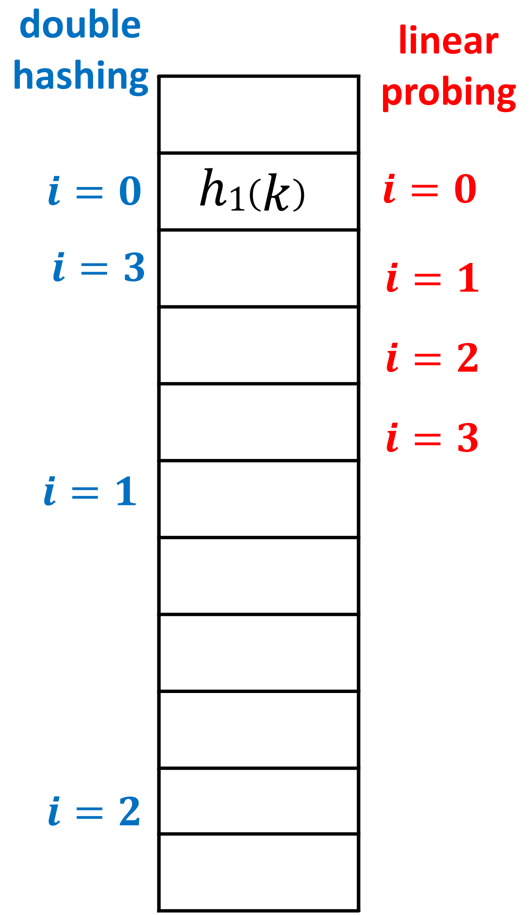
- Linear probing attempts inserting into sequence of probes which is far from random

$$h_1(k) \quad h_1(k)\ +1 \quad h_1(k)+2$$

- Want a more 'random' sequence of probes

$$h_1(k) \quad h_1(k)\ +8 \quad h_1(k)+6$$

- This will help to avoid the clustering side effect
- Note for each key k, the probe sequence must always be the same
  - for $k = 14$, probe sequence is always
    - 4, 3, 0, 2, 1, 5
  - for $k = 24$, probe sequence is always
    - 5, 0, 2, 4, 1, 3

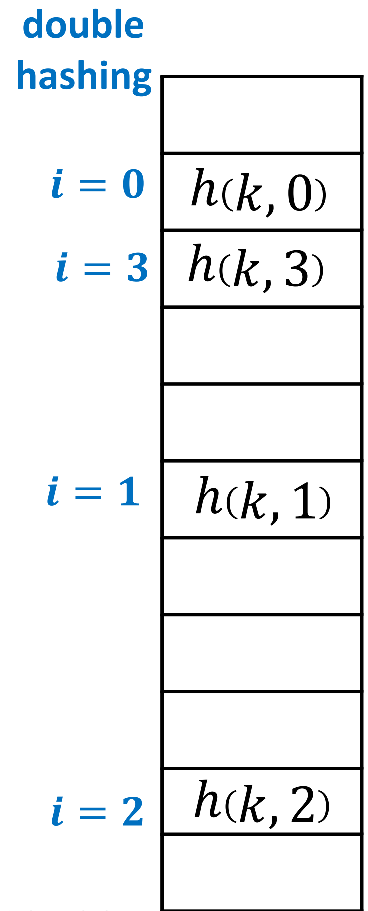


Double Hashing

- *Double hashing* : open addressing with probe sequence

$$h(k,i) = (h_0(k) + i \cdot h_1(k)) \bmod M \text{ for } i = 0,1,\ldots$$

- Where
    - h_1 is another (secondary) hash function
    - $h_1(k) \neq 0$
    - $h_1(k)$ is relative prime with M for all keys k
        - otherwise probe-sequence does not explore the entire hash table
        - easiest to choose M prime



- Double hashing with a good secondary hash function does not cause the bad clustering produced by linear probing
- *search*, *insert*, *delete* work as in linear probing, but with this different probe sequence
    - linear probing is a special case of double hashing with $h_1(k) = 1$

Independent Hash functions

- When two hash functions h_1, h_2 are required, they should be independent

$$P(h_1(k) = i\ ) \text{ and } P(h_2(k) = j) \text{ are independent}$$

- Using two modular hash-functions may lead to dependencies
- Better idea: Use *multiplicative method* for second hash function
  - let $0 < A < 1$
  - $h(k) = \lfloor M(\underbrace{kA - \lfloor kA \rfloor}) \rfloor$

    $0 \leq$ fractional part of $kA < 1$

    $0 \leq M \cdot$ (fractional part of kA) $< M$
- Example
  - $M = 11, A = 0.2$
  - $h(34) = \lfloor 11 \cdot (34 \cdot 0.2 - \lfloor 34 \cdot 0.2 \rfloor) \rfloor = \lfloor 11 \cdot (6.8 - \lfloor 6.8 \rfloor) \rfloor = \lfloor 11 \cdot 0.8 \rfloor = 8$
- $A = \varphi = \frac{\sqrt{5}-1}{2} \approx 0.618033988749$ works well to scramble the keys
  - should use at least $\log |U| + \log |M|$ bits of A
- For secondary hash function, to avoid $h(k) = 0$, use

$$h_1(k) = \lfloor (M-1)(kA - \lfloor kA \rfloor) \rfloor + 1$$

Double Hashing Example

$$\frac{\sqrt{5}-1}{2}$$

$$M = 11,\ h_0(k) = k \bmod 11,\ h_1(k) = \lfloor 10(\varphi k - \lfloor \varphi k \rfloor) \rfloor + 1$$

$h(k,i) = (h_0(k) + i \cdot h_1(k))mod\ M$ for sequence $i = 0,1,\ldots$

0	
1	45
2	13
3	
4	92
5	49
6	
7	7
8	
9	
10	43

Double Hashing Example

$$M = 11, \; h_0(k) = k \bmod 11, \; h_1(k) = \lfloor 10(\varphi k - \lfloor \varphi k \rfloor) \rfloor + 1$$

$h(k,i) = (h_0(k) + i \cdot h_1(k)) mod\ M$ for sequence $i = 0,1,\ldots$

insert(41)

$$h_0(41) = 8$$

$$h_1(41) = 4$$

$$h(41, 0) = (8 + 0 \cdot 4)\ mod\ 11 = 8$$

0	
1	45
2	13
3	
4	92
5	49
6	
7	7
8	
9	
10	43

Double Hashing Example

$$M = 11,\ h_0(k) = k \bmod 11,\ h_1(k) = \lfloor 10(\varphi k - \lfloor \varphi k \rfloor) \rfloor + 1$$

$h(k, i) = (h_0(k) + i \cdot h_1(k)) mod\ M$ for sequence $i = 0,1, \ldots$

insert(41)

$$h_0(41) = 8$$

$$h_1(41) = 4$$

$$h(41, 0) = (8 + 0 \cdot 4)\ mod\ 11 = 8$$

0	
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	
10	43

Double Hashing Example

$$M = 11, \; h_0(k) = k \bmod 11, \; h_1(k) = \lfloor 10(\varphi k - \lfloor \varphi k \rfloor) \rfloor + 1$$

$h(k,i) = (h_0(k) + i \cdot h_1(k)) mod\ M$ for sequence $i = 0,1,\ldots$

insert(194)

$$h_0(194) = 7$$
$$h_1(194) = 9$$
$$h(194, 0) = (7 + 0 \cdot 9)\ mod\ 11 = 7$$

0	
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	
10	43

Double Hashing Example

$$M = 11, \; h_0(k) = k \bmod 11, \; h_1(k) = \lfloor 10(\varphi k - \lfloor \varphi k \rfloor) \rfloor + 1$$

$$h(k, i) = (h_0(k) + i \cdot h_1(k)) mod\ M$$ for sequence $i = 0,1,\ldots$

insert(194)

$$h_0(194) = 7$$

$$h_1(194) = 9$$

$$h(194, 0) = (7 + 0 \cdot 9)\ mod\ 11 = 7$$

0	
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	
10	43

Double Hashing Example

$$M = 11, \; h_0(k) = k \bmod 11, \; h_1(k) = \lfloor 10(\varphi k - \lfloor \varphi k \rfloor) \rfloor + 1$$

$h(k,i) = (h_0(k) + i \cdot h_1(k)) mod\ M$ for sequence $i = 0,1,\ldots$

insert(194)

$$h_0(194) = 7$$
$$h_1(194) = 9$$
$$h(194, 1) = (7 + 1 \cdot 9)\ mod\ 11 = 5$$

0	
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	
10	43

Double Hashing Example

$$M = 11,\ h_0(k) = k \bmod 11,\ h_1(k) = \lfloor 10(\varphi k - \lfloor \varphi k \rfloor) \rfloor + 1$$

$h(k,i) = (h_0(k) + i \cdot h_1(k)) mod\ M$ for sequence $i = 0,1,\ldots$

insert(194)

$$h_0(194) = 7$$

$$h_1(194) = 9$$

$$h(194, 1) = (7 + 1 \cdot 9)\ mod\ 11 = 5$$

0	
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	
10	43

Double Hashing Example

$$M = 11, \ h_0(k) = k \bmod 11, \ h_1(k) = \lfloor 10(\varphi k - \lfloor \varphi k \rfloor) \rfloor + 1$$

$h(k, i) = (h_0(k) + i \cdot h_1(k)) mod\ M$ for sequence $i = 0, 1, \ldots$

insert(194)

$$h_0(194) = 7$$

$$h_1(194) = 9$$

$$h(194, 2) = (7 + 2 \cdot 9)\ mod\ 11 = 3$$

0	
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	
10	43

Double Hashing Example

$$M = 11,\ h_0(k) = k \bmod 11,\ h_1(k) = \lfloor 10(\varphi k - \lfloor \varphi k \rfloor) \rfloor + 1$$

$h(k,i) = (h_0(k) + i \cdot h_1(k)) mod\ M$ for sequence $i = 0,1,\ldots$

insert(194)

$$h_0(194) = 7$$

$$h_1(194) = 9$$

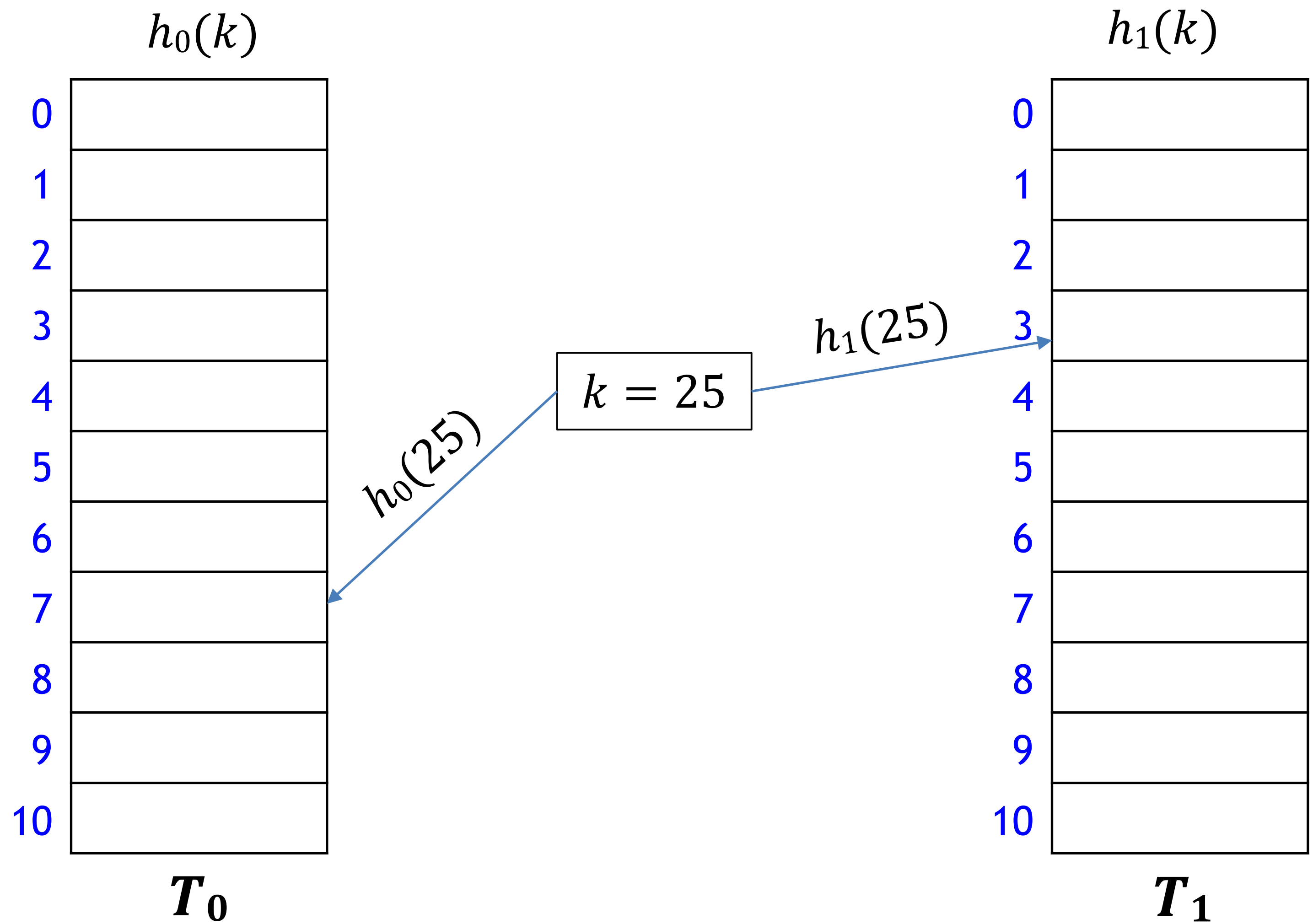
$$h(194, 2) = (7 + 2 \cdot 9)\ mod\ 11 = 3$$

0	
1	45
2	13
3	194
4	92
5	49
6	
7	7
8	41
9	
10	43

Outline

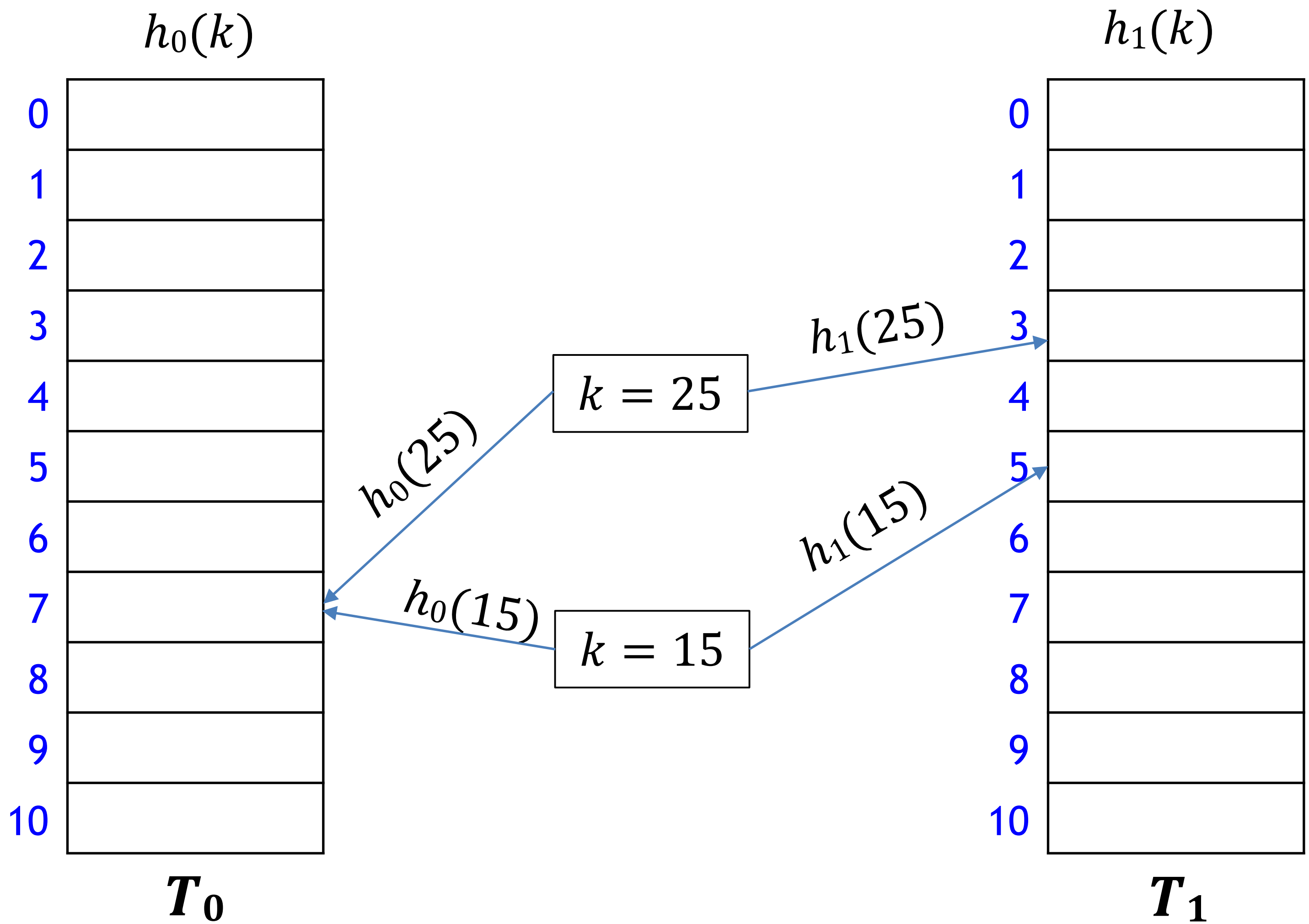
- Dictionaries via Hashing
    - Hashing Introduction
    - Hashing with Chaining
    - Open Addressing
        - probe Sequences
        - cuckoo hashing
    - Hash Function Strategies

Cuckoo Hashing



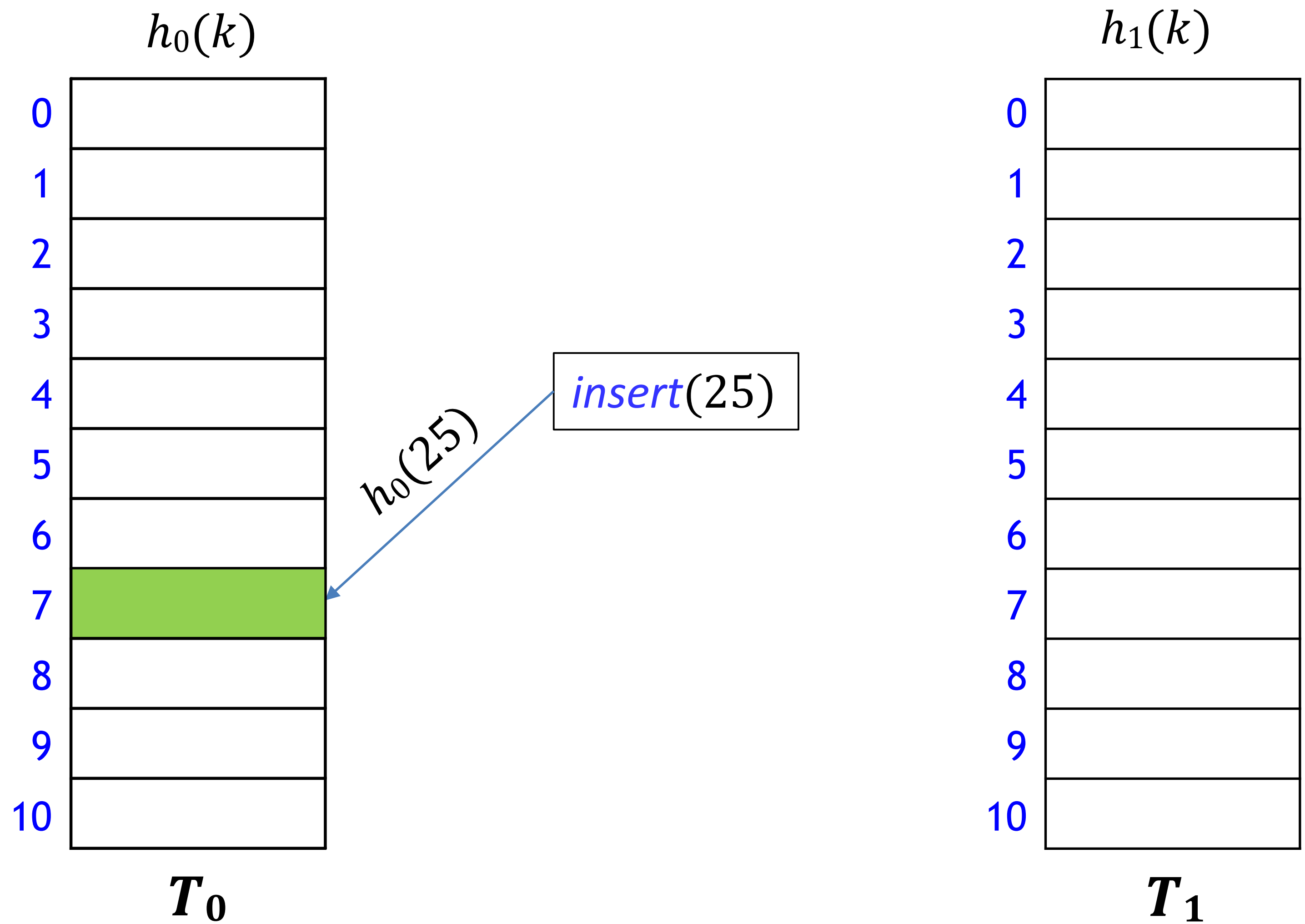
- **Main idea:** An item with key k can be only at $T_0[h_0(k)]$ or $T_1[h_1(k)]$

Cuckoo Hashing



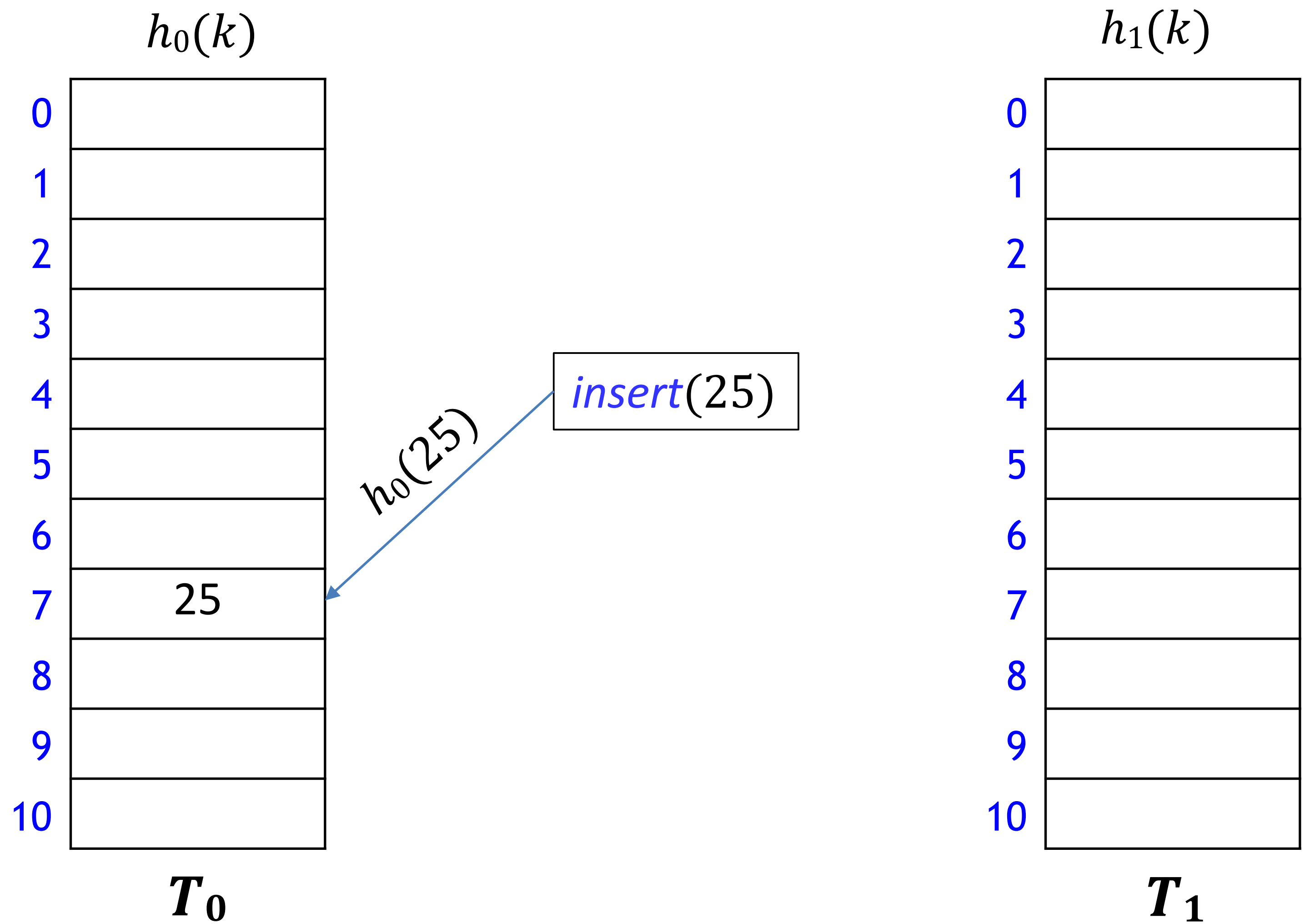
- **Main idea:** An item with key *k* can be only at $T_0[h_0(k)]$ or $T_1[h_1(k)]$
    - *search* and *delete* take $O(1)$ time

Cuckoo Hashing



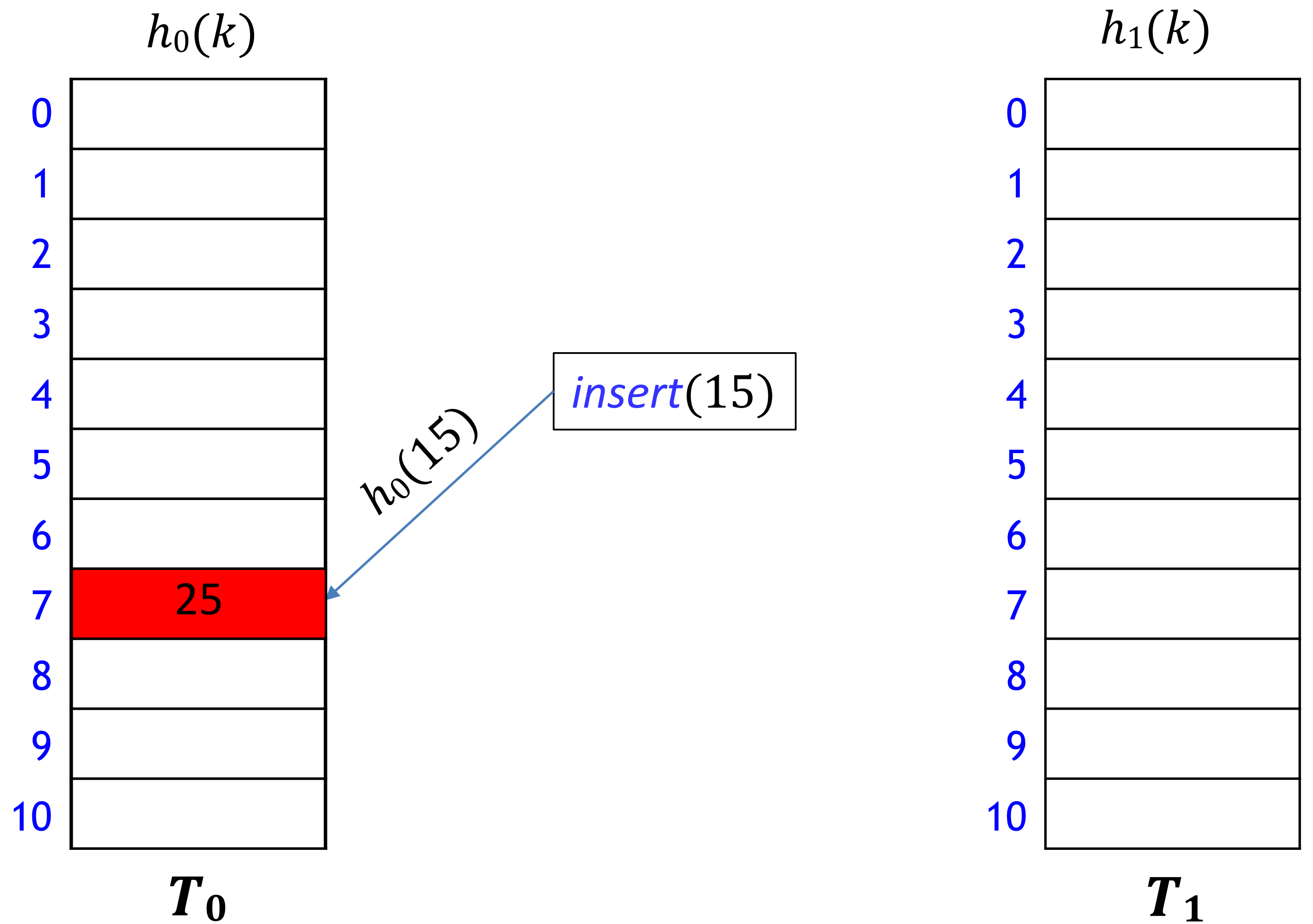
- How to insert?

Cuckoo Hashing



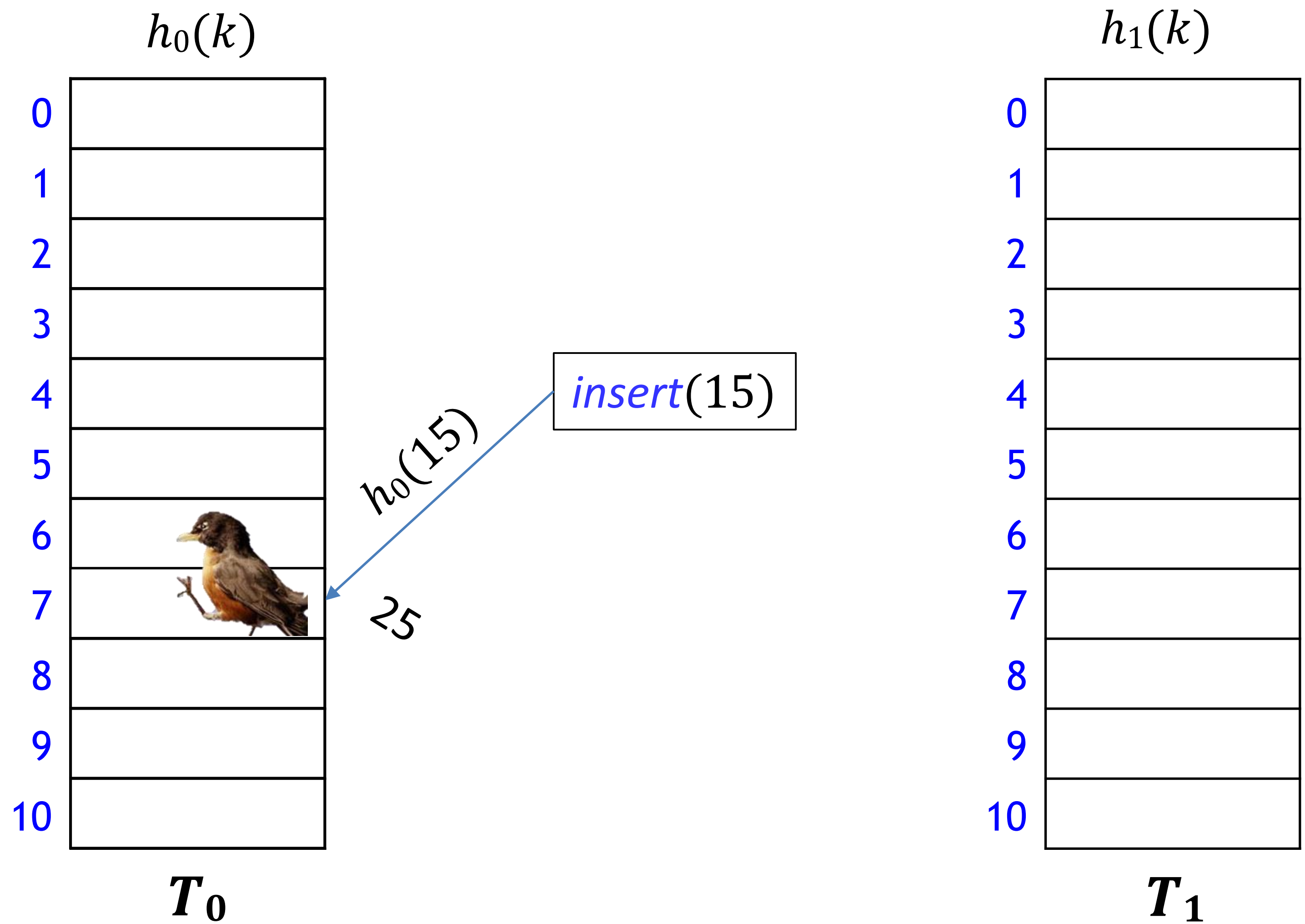
- How to insert?

Cuckoo Hashing



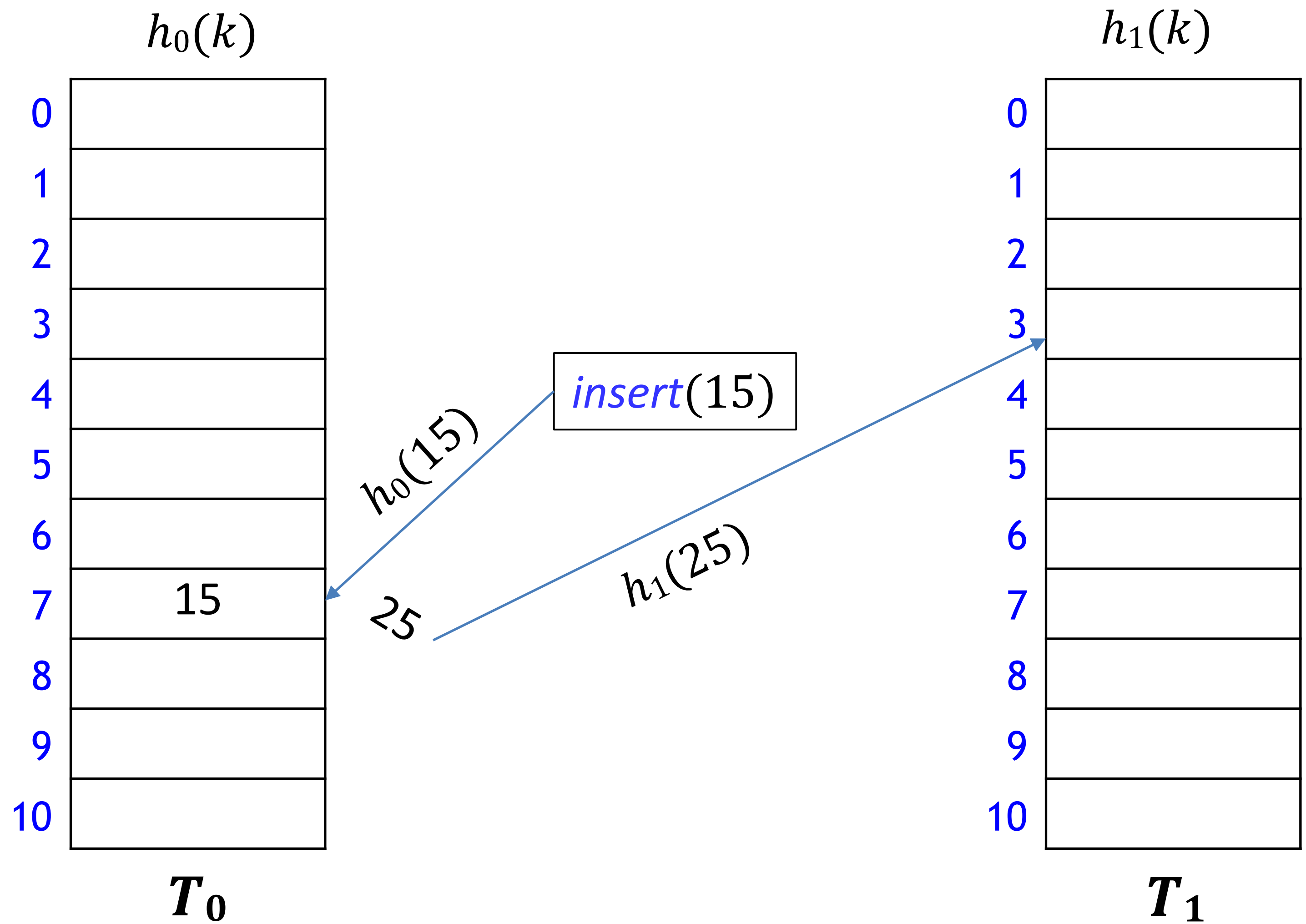
- How to insert k when $h_0(k)$ is already occupied?

Cuckoo Hashing



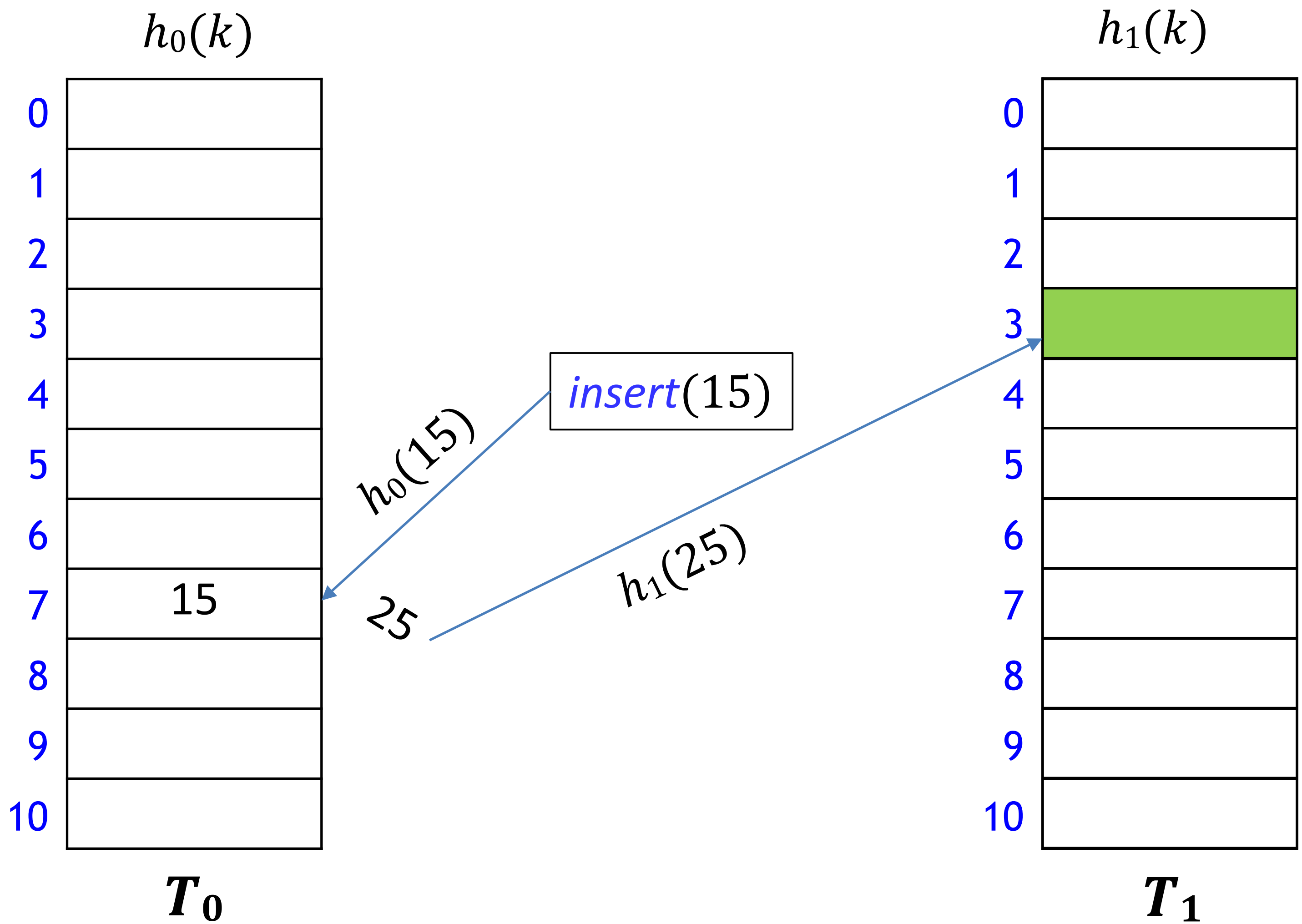
- How to insert k when $h_0(k)$ is already occupied?

Cuckoo Hashing



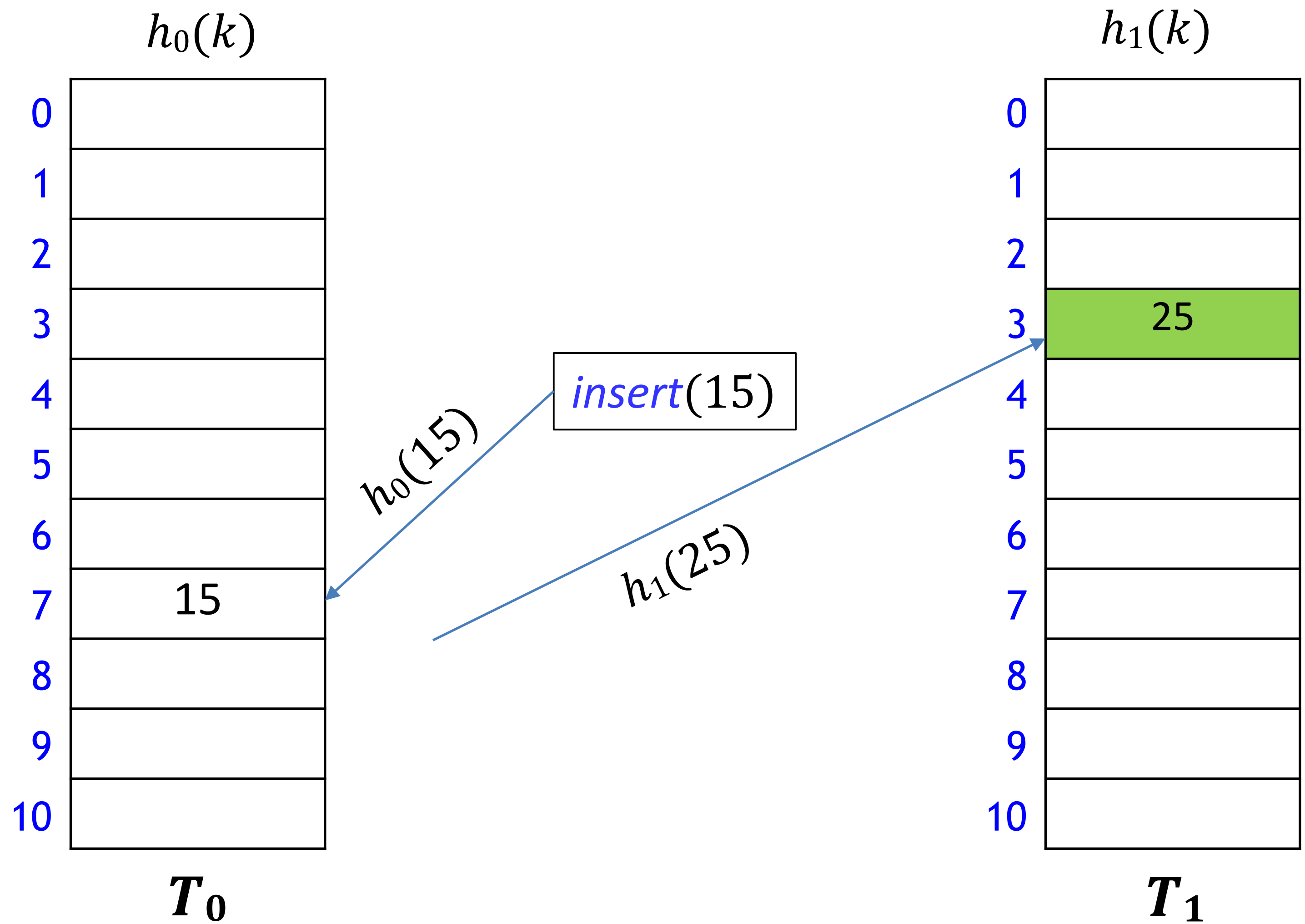
- How to insert k when $h_0(k)$ is already occupied?

Cuckoo Hashing



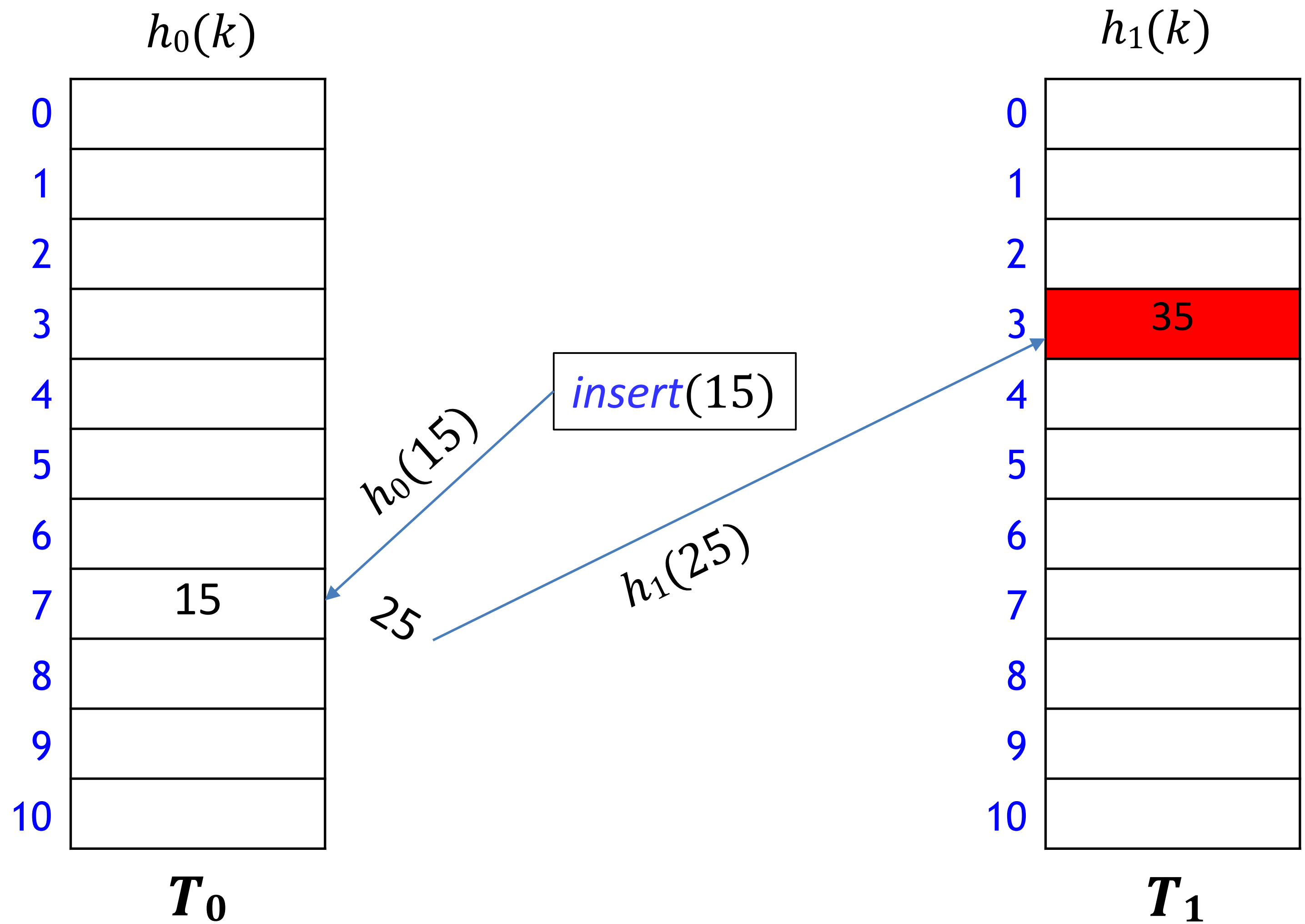
- How to insert k when $h_0(k)$ is already occupied?

Cuckoo Hashing



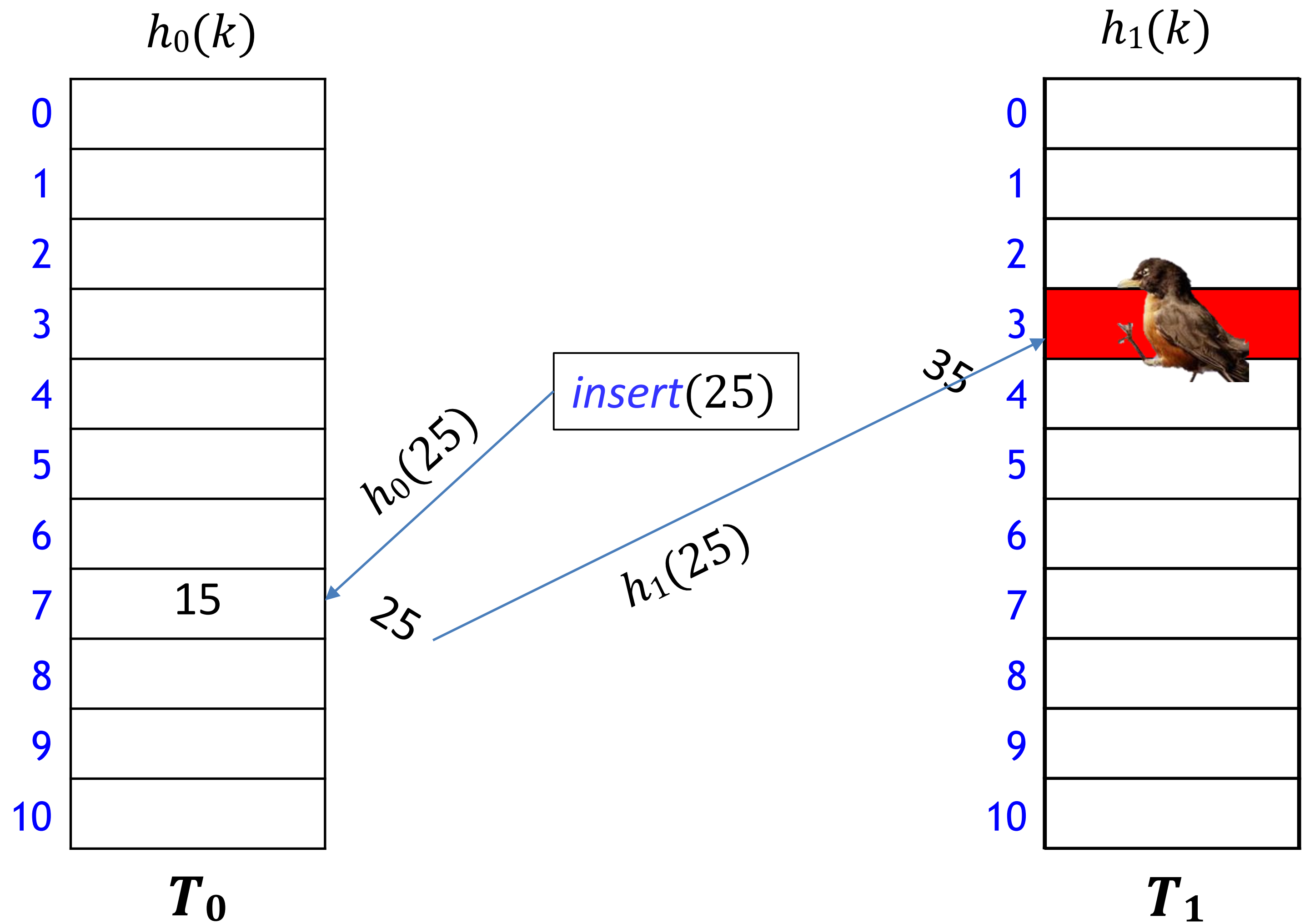
- How to insert k when $h_0(k)$ is already occupied?

Cuckoo Hashing



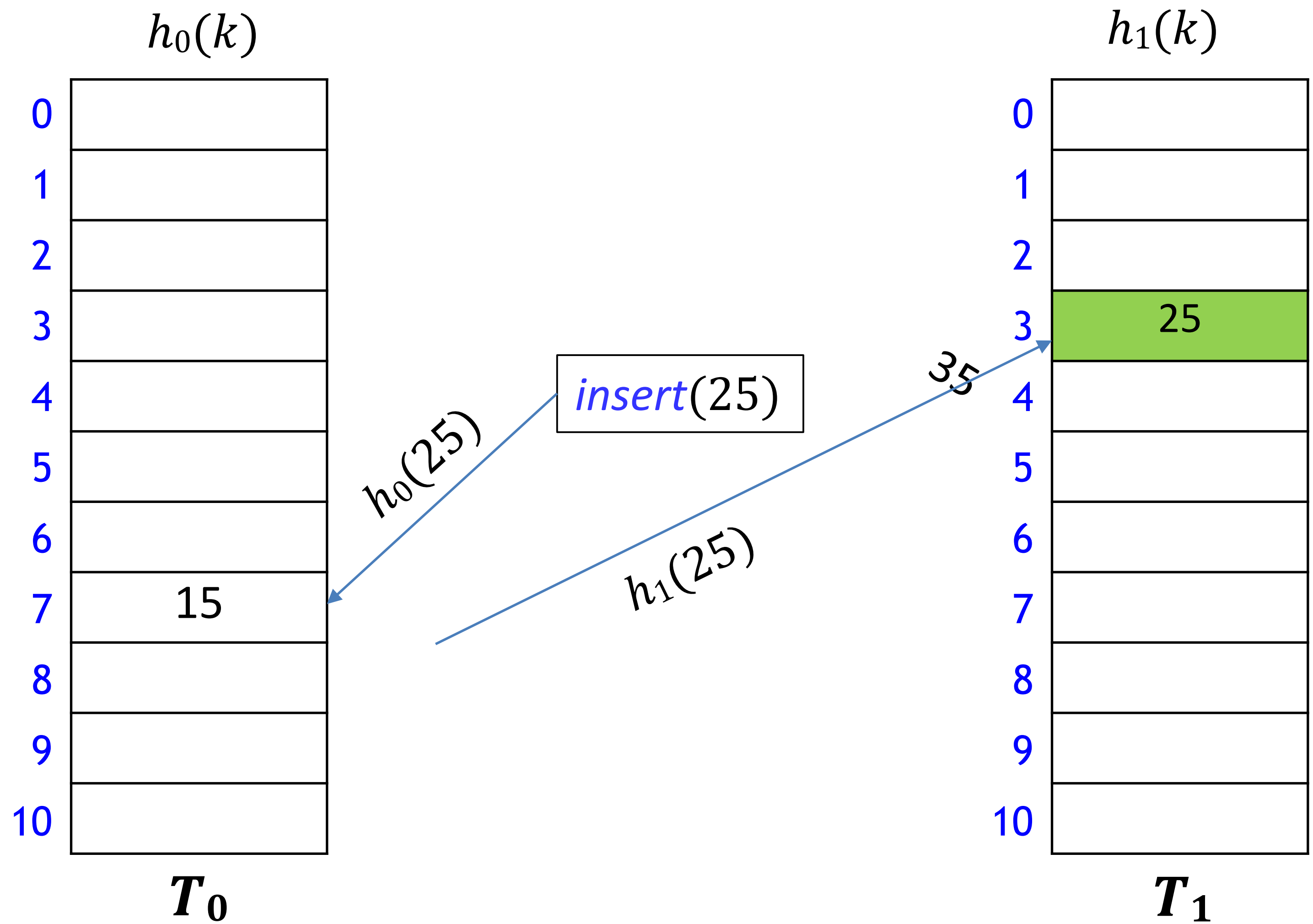
- How to insert k when $h_0(k)$ is already occupied?

Cuckoo Hashing



- How to insert k when $h_0(k)$ is already occupied?

Cuckoo Hashing

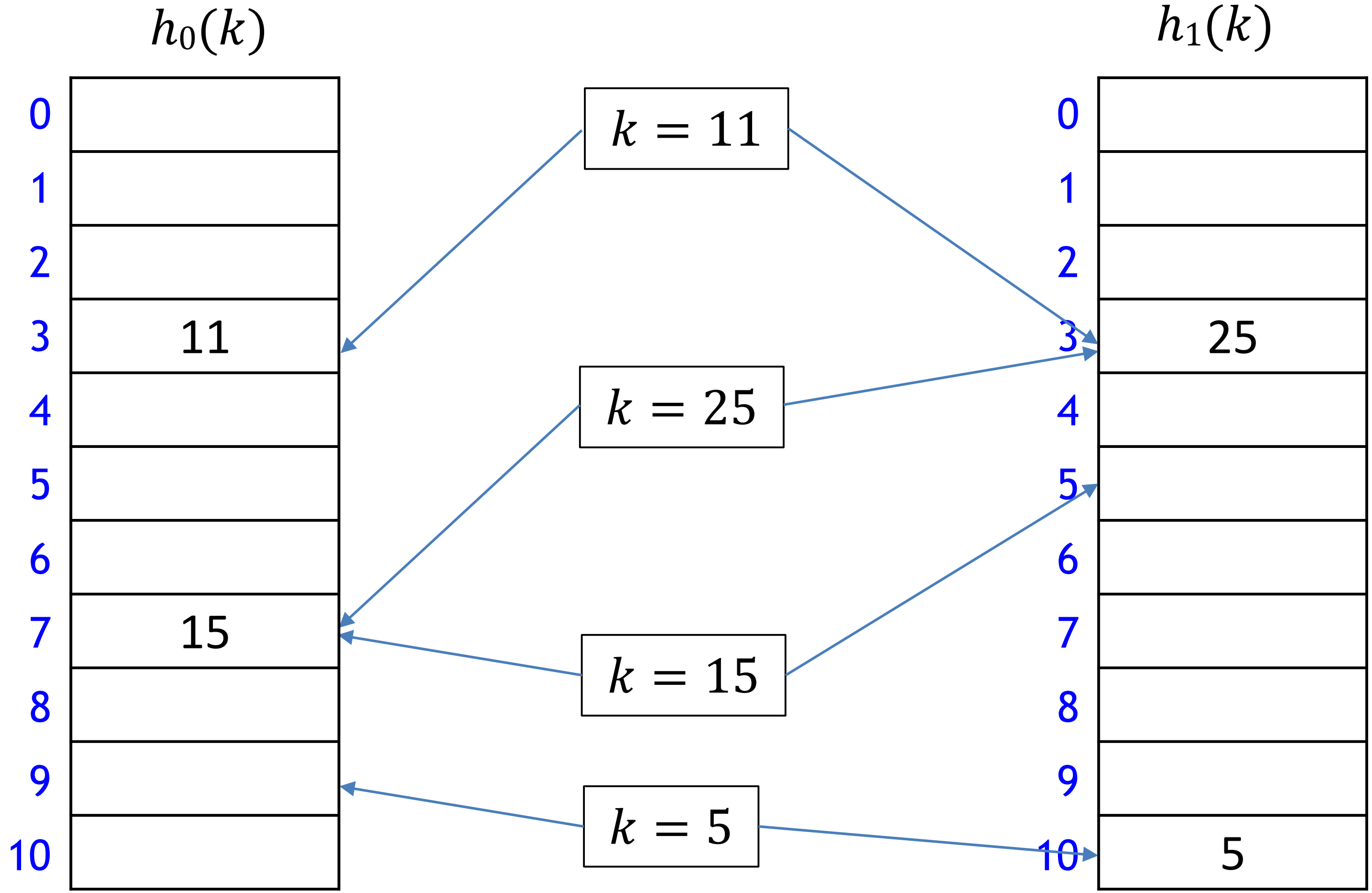


- Continue until all items placed, or *failure*
  - rehash if failure

Cuckoo Hashing [Pagh & Rodler, 2001]

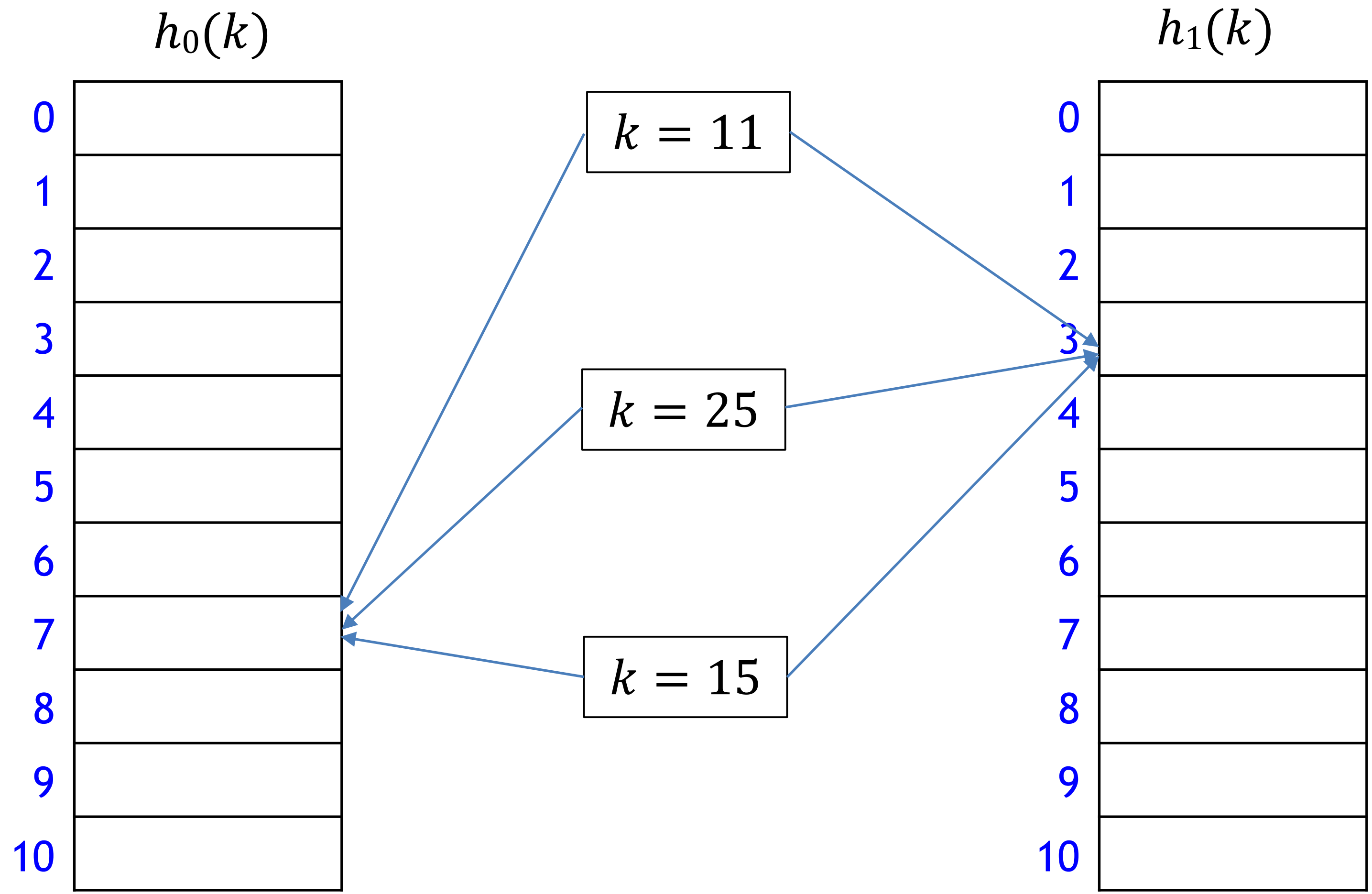
- Use independent hash functions h_0, h_1 and two tables T_0, T_1
- Key k can be only at $T_0[h_0(k)]$ or $T_1[h_1(k)]$
  - *search* and *delete* take constant time
  - *insert* starts with T_0 and alternates between T_0 and T_1 kicking out current occupant, if necessary, until no item is kicked out
    - may lead to a loop of "kicking out"
      - detect loops by aborting after too many attempts
      - signal failure
      - if failure, rehash with larger M and new hash functions
- *insert* may be slow, but expected constant time if the load factor is small
- Works well in practice

Cuckoo Hashing



- Intuitively
  - each key has 2 locations (locations can coincide)
  - try to "match" keys to locations so that everyone is placed

Cuckoo Hashing



- Sometimes no solution for the "matching" problem
    - would loop infinitely if not stopped by force

Cuckoo hashing: Insert

$$M = 11,\ h_0(k) = k \bmod 11,\ \ h_1(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$$

insert(51)

$i = 0$

$k = 51$

$h_0(k) = 7$

0	44
1	
2	
3	
4	59
5	
6	
7	
8	
9	92
10	

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

Cuckoo hashing: Insert

$M = 11,\ h_0(k) = k \bmod 11,\ \ h_1(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$

$insert(51)$

$i = 0$

$k = 51$

$h_0(k) = 7$

0	44	
1		
2		
3		
4	59	
5		
6		
7	51	
8		
9	92	
10		

Cuckoo hashing: Insert

$$M = 11,\ h_0(k) = k \bmod 11,\ \ h_1(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$$

$insert(95)$

$i = 0$

$k = 95$

$h_0(k) = 7$

0	44	0	
1		1	
2		2	
3		3	
4	59	4	
5		5	
6		6	
7	51	7	
8		8	
9	92	9	
10		10	

Cuckoo hashing: Insert

$$M = 11,\ h_0(k) = k \bmod 11,\ \ h_1(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$$

$insert(95)$

$i = 0$

$k = 95$

$h_0(k) = 7$

0	44	0	
1		1	
2		2	
3		3	
4	59	4	
5		5	
6		6	
7	51	7	
8		8	
9	92	9	
10		10	

Cuckoo hashing: Insert

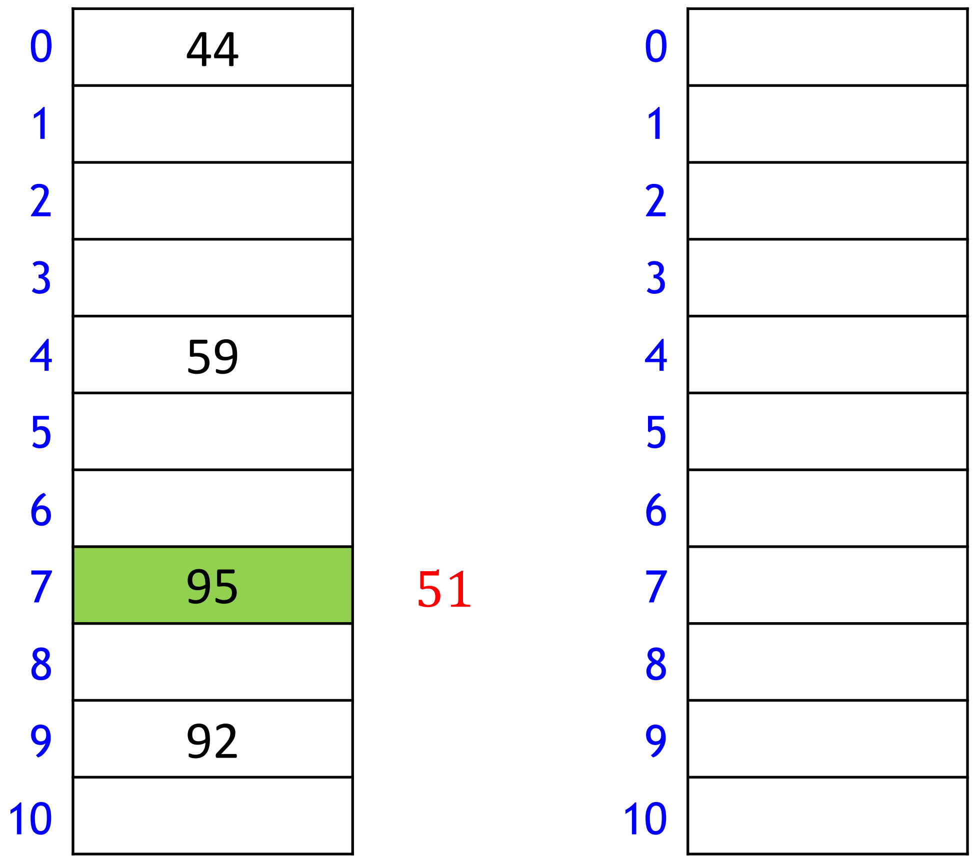
$$M = 11,\ h_0(k) = k \bmod 11,\ \ h_1(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$$

insert(95)

$i = 0$

$k = 95$

$h_0(k) = 7$



Cuckoo hashing: Insert

$$M = 11,\ h_0(k) = k \bmod 11,\ \ h_1(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$$

$insert(95)$
$i = 1$
$k = 51$
$h_1(k) = 5$

0	44		0	
1			1	
2			2	
3			3	
4	59		4	
5			5	
6			6	
7	95	51	7	
8			8	
9	92		9	
10			10	

Cuckoo hashing: Insert

$$M = 11,\ h_0(k) = k \bmod 11,\ \ h_1(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$$

$insert(95)$
$i = 1$
$k = 51$
$h_1(k) = 5$

	T_0		T_1
0	44		
1			
2			
3			
4	59		
5			
6			
7	95	51	
8			
9	92		
10			

Cuckoo hashing: Insert

$$M = 11,\ h_0(k) = k \bmod 11,\ h_1(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$$

$insert(95)$

$i = 1$

$k = 51$

$h_1(k) = 5$

	T_0	T_1
0	44	
1		
2		
3		
4	59	
5		51
6		
7	95	
8		
9	92	
10		

Cuckoo hashing: Insert

$$M = 11,\ h_0(k) = k \bmod 11,\ \ h_1(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$$

$insert(26)$

$i = 0$

$k = 26$

$h_0(k) = 4$

0	44	
1		
2		
3		
4	59	
5		51
6		
7	95	
8		
9	92	
10		

Cuckoo hashing: Insert

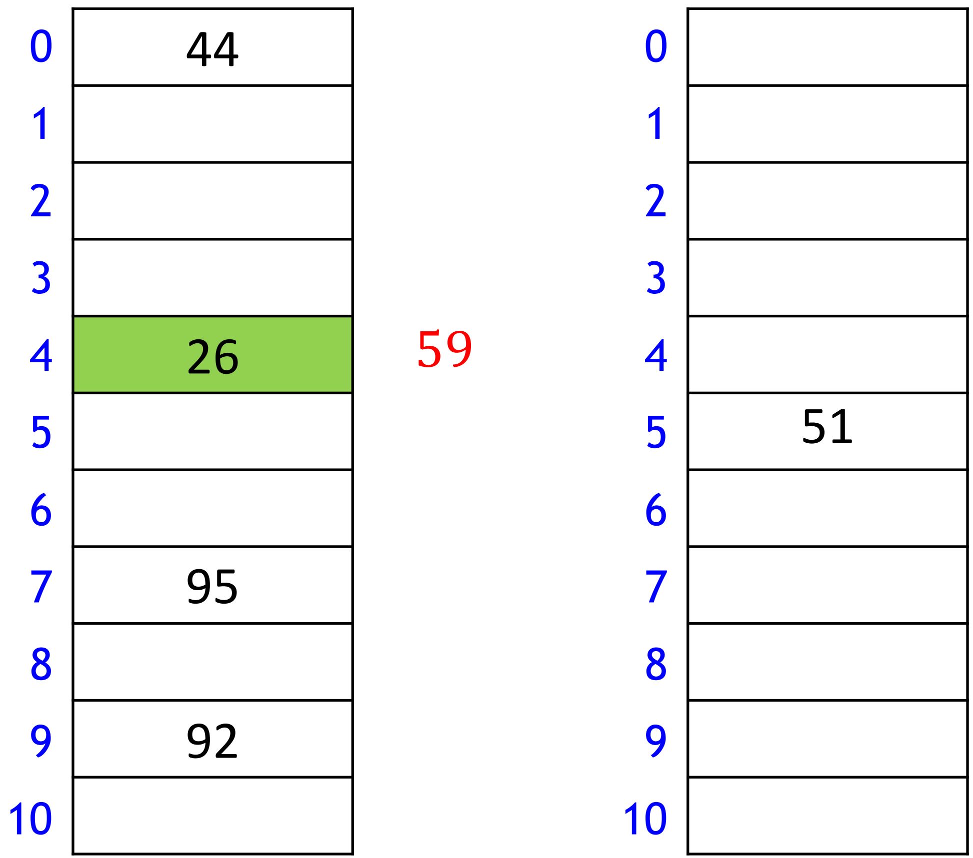
$M = 11,\ h_0(k) = k \bmod 11,\ \ h_1(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$

$insert(26)$

$i = 0$

$k = 26$

$h_0(k) = 4$



Cuckoo hashing: Insert

$M = 11,\ h_0(k) = k \bmod 11,\ \ h_1(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$

$insert(26)$

$i = 1$

$k = 59$

$h_1(k) = 5$

0	44
1	
2	
3	
4	26
5	
6	
7	95
8	
9	92
10	

59

0	
1	
2	
3	
4	
5	51
6	
7	
8	
9	
10	

Cuckoo hashing: Insert

$M = 11,\ h_0(k) = k \bmod 11,\ \ h_1(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$

$insert(26)$

$i = 1$

$k = 59$

$h_1(k) = 5$

	T_0	T_1	
0	44		
1			
2			
3			
4	26		
5		59	51
6			
7	95		
8			
9	92		
10			

Cuckoo hashing: Insert

$$M = 11,\ h_0(k) = k \bmod 11,\ \ h_1(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$$

$insert(26)$

$i = 0$

$k = 51$

$h_0(k) = 7$

0	44
1	
2	
3	
4	26
5	
6	
7	95
8	
9	92
10	

0		
1		
2		
3		
4		
5	59	51
6		
7		
8		
9		
10		

Cuckoo hashing: Insert

$M = 11,\ h_0(k) = k \bmod 11,\ \ h_1(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$

insert(26)

$i = 0$

$k = 51$

$h_0(k) = 7$

0	44	
1		
2		
3		
4	26	
5		
6		
7	51	95
8		
9	92	
10		

0	
1	
2	
3	
4	
5	59
6	
7	
8	
9	
10	

Cuckoo hashing: Insert

$M = 11,\ h_0(k) = k \bmod 11,\ \ h_1(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$

$insert(26)$

$i = 1$

$k = 95$

$h_1(k) = 7$

0	44		0	
1			1	
2			2	
3			3	
4	26		4	
5			5	59
6			6	
7	51	95	7	
8			8	
9	92		9	
10			10	

Cuckoo hashing: Insert

$M = 11,\ h_0(k) = k \bmod 11,\ \ h_1(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$

$insert(26)$

$i = 1$

$k = 95$

$h_1(k) = 7$

	T_0	T_1
0	44	
1		
2		
3		
4	26	
5		59
6		
7	51	95
8		
9	92	
10		

Cuckoo Hashing: Insert Pseudocode

cuckoo::insert(k, v)

$i \leftarrow 0$

do at most $2n$ times

if $T_i[h_i(k)]$ is empty

$T_i[h_i(k)] \leftarrow (k, v)$

return "success"

//insert $T_i\,[h_i(k)]$ into the other table

swap$\big((k, v), T_i[h_i(k)]\big)$ // kick out current occupant

$i \leftarrow 1 - i$ // alternate between 0 and 1

return *failure* // re-hash

- After $2n$ iterations, there is definitely an infinite loop of 'kicking out'
- Practical tip
  - do not wait for $2n$ unsuccessful tries to declare failure
  - declare failure after, say, 10 unsuccessful iterations

Cuckoo hashing: Search

$M = 11,\ h_0(k) = k \bmod 11,\ h_1(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$

search(59)

$h_0(59) = 4$
$h_1(59) = 5$

0	44	0		
1		1		
2		2		
3		3		
4	26	4		
5		5	59	found
6		6		
7	51	7	95	
8		8		
9	92	9		
10		10		

Cuckoo hashing: Delete

$$M = 11,\ h_0(k) = k \bmod 11,\ \ h_1(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$$

delete(59)

$h_0(59) = 4$
$h_1(59) = 5$

0	44	0		
1		1		
2		2		
3		3		
4	26	4		
5		5	59	found
6		6		
7	51	7	95	
8		8		
9	92	9		
10		10		

Cuckoo hashing: Delete

$$M = 11,\ h_0(k) = k \bmod 11,\ \ h_1(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$$

delete(59)

$$h_0(59) = 4$$
$$h_1(59) = 5$$

0	44
1	
2	
3	
4	26
5	
6	
7	51
8	
9	92
10	

0	
1	
2	
3	
4	
5	
6	
7	95
8	
9	
10	

no need to mark deleted spot

Cuckoo hashing discussion

- The two hash tables do not have to be of the same size
- Load factor $\alpha = n/(\text{size of } T_0 + \text{size of } T_1)$
- One can argue that if the load factor is small enough, then insertion has $O(1)$ expected time
    - this requires $\alpha < 1/2$
- There are many variations of cuckoo hashing
    - two hash tables can be combined into one
    - more flexible when inserting: always consider both possible positions
    - Use $k > 2$ allowed locations
        - k tables or k hash functions

Complexity of Open Addressing Strategies

- For any open addressing scheme, we *must* have $\alpha \leq 1$ (why?)
- For analysis, require $\alpha < 1$, for Cuckoo hashing require $\alpha < 1/2$

Expected # probes $\leq$	*search(unsuccessful)*	*insert*	*search (successful)*
Linear Probing	$\frac{1}{(1-\alpha)^2}$	$\frac{1}{(1-\alpha)^2}$	$\frac{1}{1-\alpha}$ (on avg. over keys)
Double Hashing	$\frac{1}{1-\alpha}+o(1)$	$\frac{1}{1-\alpha}+o(1)$	$\frac{1}{1-\alpha}+o(1)$
Cuckoo Hashing	1 (worst case)	$\frac{\alpha}{(1-2\alpha)^2}$	1 (worst case)

- All operations have $O(1)$ expected run-time if hash-function chosen uniformly and α is kept sufficiently small
- But the worst case runtime is (usually) $\Theta(n)$

Outline

- Dictionaries via Hashing
    - Hashing Introduction
    - Hashing with Chaining
    - Open Addressing
        - probe Sequences
        - cuckoo hashing
    - Hash Function Strategies

Choosing Good Hash Function

- Satisfying the uniform hashing assumption is impossible
  - too many hash functions and for most, computing $h(k)$ is not cheap
- We need to compromise
  - choose hash function that is easy to compute
  - but aim for $P(h(k) = i) = \frac{1}{M}$
- If all keys are used equally often, this is easy
- In practice, keys are not used equally often
- Can get good performance by choosing hash-function that is
  - unrelated to any possible patterns in the data, and
  - depends on all parts of the key
- We saw two basic methods for integer keys
  - **Modular method**: $h(k) = k \ mod \ M$
    - M should be prime
  - **Multiplicative method**: $h(k) = \lfloor M(kA - \lfloor kA \rfloor) \rfloor$
    - $0 < A < 1$

Carter-Wegman's Universal Hashing

- Even better: randomization that uses easy-to-compute hash functions
  - Requires: all keys are in $\{0, \ldots p-1\}$ for some (big) prime p
  - choose number $M < p$
    - M equal to some power of 2 is ok
  - Choose two random numbers $a, b \in \{0, \ldots p-1\}, \; a \neq 0$
  - Use as hash function
$$h(k) = ((ak + b) \bmod p) \bmod M$$
    - can be computed in $O(1)$ time
  - Uniform hashing assumption is not satisfied, but
    - can prove that two keys collide with probability at most $\frac{1}{M}$
    - this is enough to prove the expected runtime bounds we had for chainging

Multi-dimensional Data

- May need multi-dimensional non integer keys
  - example: strings in Σ^*

1. Construct $f(w) \in N$ for converting string w to integer
   - ASCII representation of APPLE is $(65, 80, 80, 76, 69)$
   - simple addition: $f(APPLE) = 65 + 80 + 80 + 76 + 69$
   - many collisions, 'stop'='tops'='pots'
   - *polynomial accumulation* works better
     - choose radix R, e.g. $R = 255$
     - $f(APPLE) = 65R^4 + 80R^3 + 80R^2 + 76R^1 + 69R^0$
     - compute in $O(|w|)$ time with Horner's rule
     - either ignoring overflow

$$f(APPLE) = \left(\left((65R + 80)R + 80\right)R + 76\right)R + 69$$

     - or apply $mod\ M$ after each addition

2. Now apply any hash function, such as $h(w) = f(w)\ mod\ M$

Hashing vs. Balanced Search Trees

- **Advantages of Balanced Search Trees**
  - $O(\log n)$ worst-case operation cost
  - does not require any assumptions, special functions, or known properties of input distribution
  - predictable space usage (exactly n nodes)
  - never need to rebuild the entire structure
  - supports ordered dictionary operations (rank, select etc.)
- **Advantages of Hash Tables**
  - $O(1)$ expected time operations (if hashes well-spread and load factor small)
  - can choose space-time tradeoff via load factor
  - cuckoo hashing achieves $O(1)$ worst-case for search & delete