CS 240 – Data Structures and Data Management

Module 7: Dictionaries via Hashing

O. Veksler

Based on lecture notes by many previous cs240 instructors

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

Winter 2025

Outline

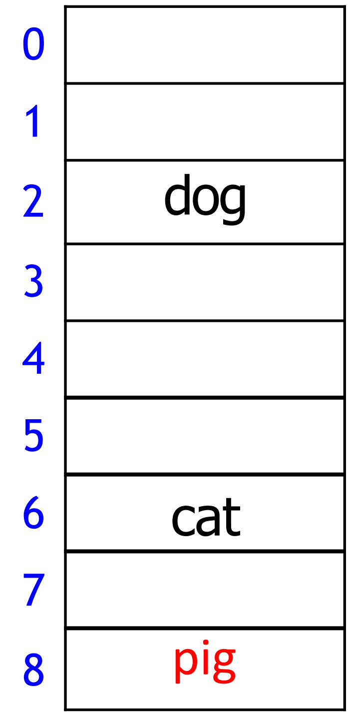
- Dictionaries via Hashing
    - Hashing Introduction
    - Hashing with Chaining
    - Open Addressing
        - probe sequences
        - cuckoo hashing
    - Hash Function Strategies

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
  - 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): $A[k] \leftarrow v$

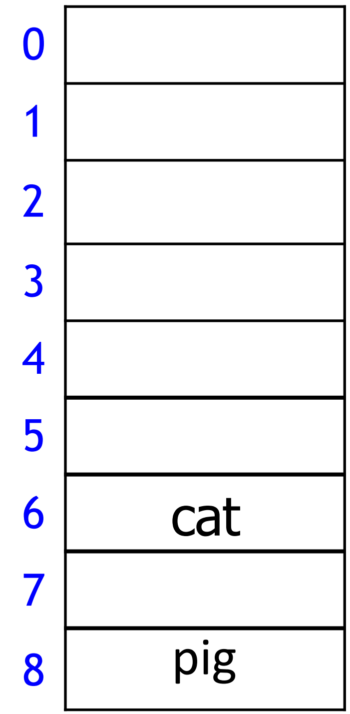


$D = \{(2, \text{dog}), (6, \text{cat})\}$

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

Direct Addressing

- Special situation: every key k is integer with $0 \leq k < M$
- Direct addressing implementation
  - 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*

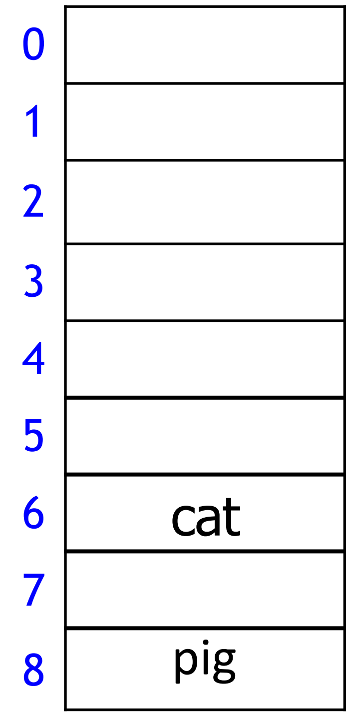


$D = \{(2, \text{dog}), (6, \text{cat}), (8, \text{pig})\}$

$delete(2)$

Direct Addressing

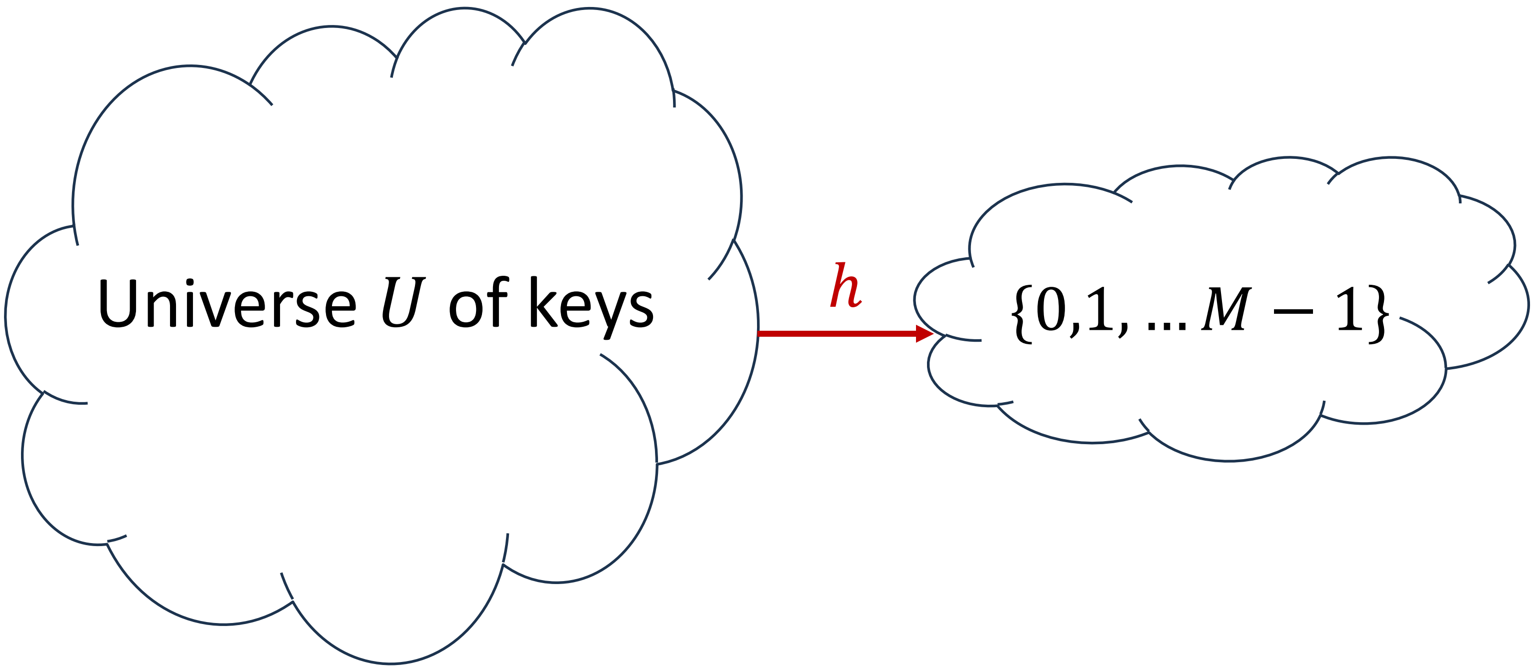
- Special situation: every key k is integer with $0 \leq k < M$
- Direct addressing implementation
  - store (k, v) in array A of size M *via* $A[k] \leftarrow v$
  - *search*(k): check if $A[k]$ is empty
  - *insert*(k, v): $A[k] \leftarrow v$
  - *delete*(k): $A[k] \leftarrow$ *empty*
  - all operations are $O(1)$
  - total storage is $\Theta(M)$
- Drawbacks
  1. space is wasteful if $n << M$
  2. keys must be integers



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

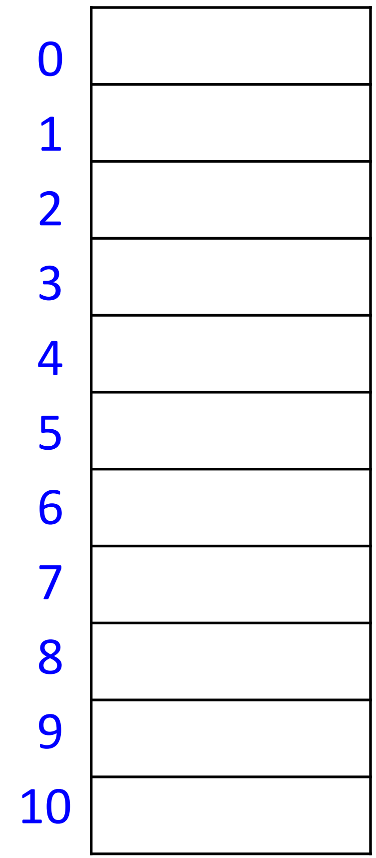
Hashing

- **Idea:** first map keys to a smaller integer range and then use direct addressing



Hashing

- **Idea:** first map keys to a smaller 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* $h: U \to \{0, 1, \ldots, M-1\}$
  - $h(k)$ 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 wants to be stored in *slot* $h(k)$ of array T
- Example
  - $U = N,\ M = 11,\ h(k) = k \bmod 11$
  - 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 wants to be stored in *slot* $h(k)$ of array T
- 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
- Typically choose $M \in \Theta(n)$
    - shrink or expand the hash table dynamically as items inserted/deleted
- There are good reasons for choosing M to be a prime number

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
     - linear probing
     - double hashing
   - cuckoo hashing: just one alternative location

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

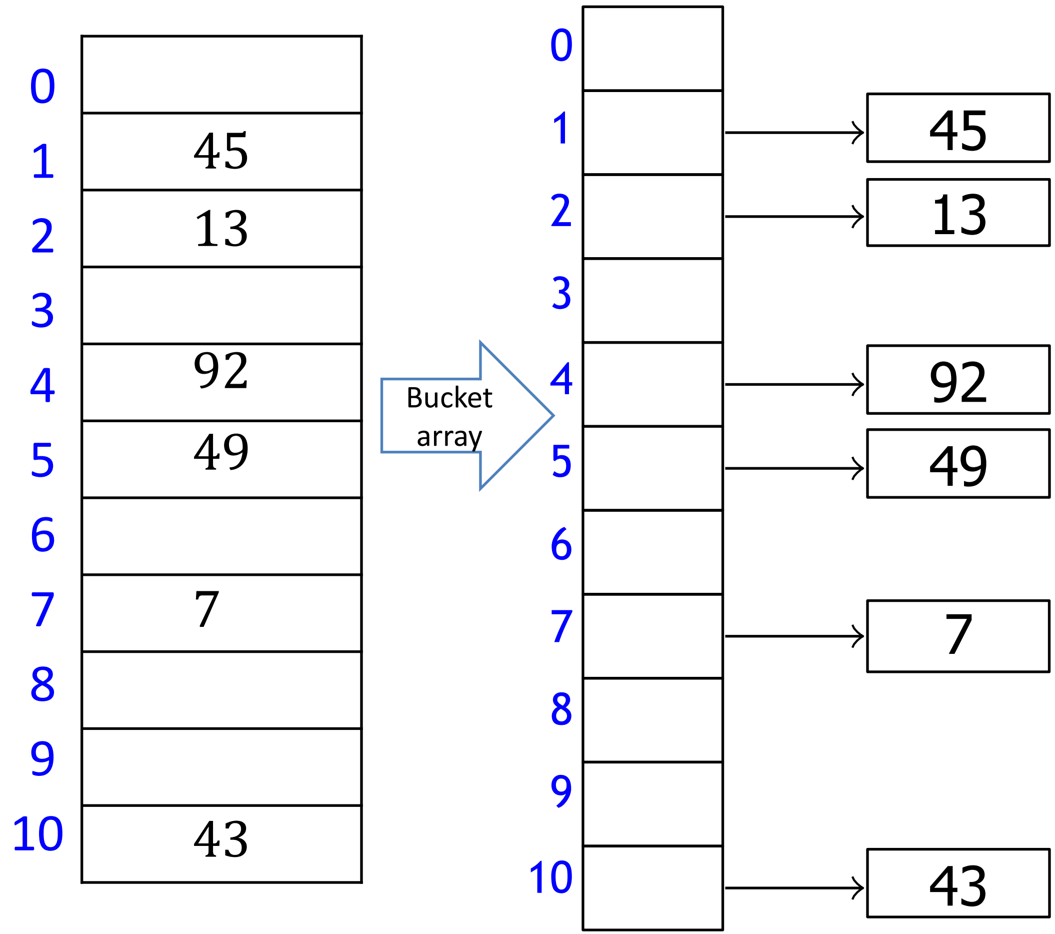
Outline

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

Hashing with Chaining

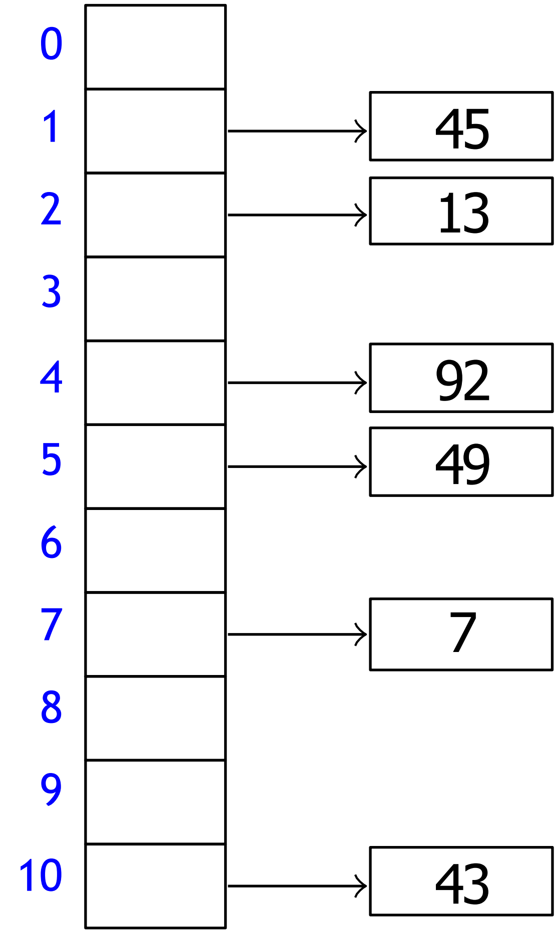
$$M = 11, h(k) = k \bmod 11$$

- 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 dictionary 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 *front* of list at $T\,[h(k)]$
  - *delete*(k): search and delete from the list at $T[h(k)]$

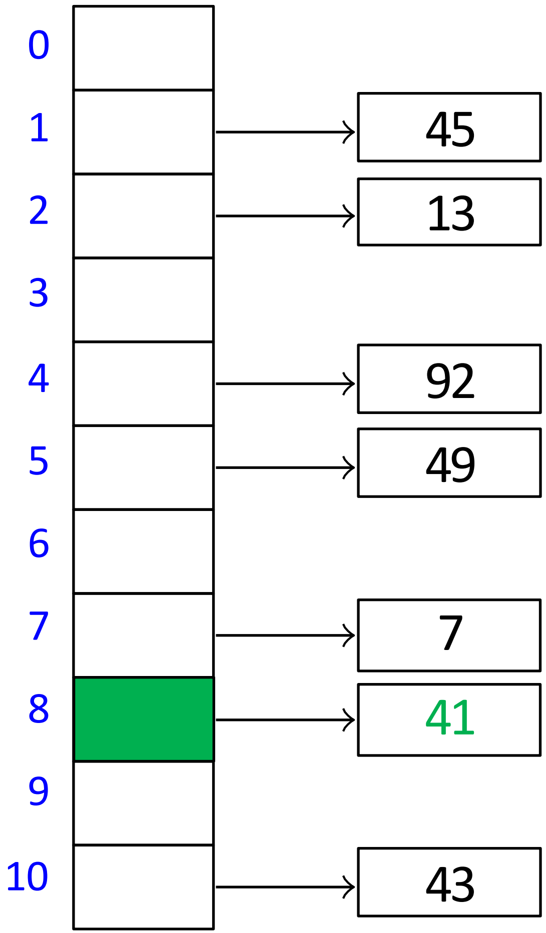


Hashing with Chaining Example

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

insert(41)

$$h(41) = 8$$

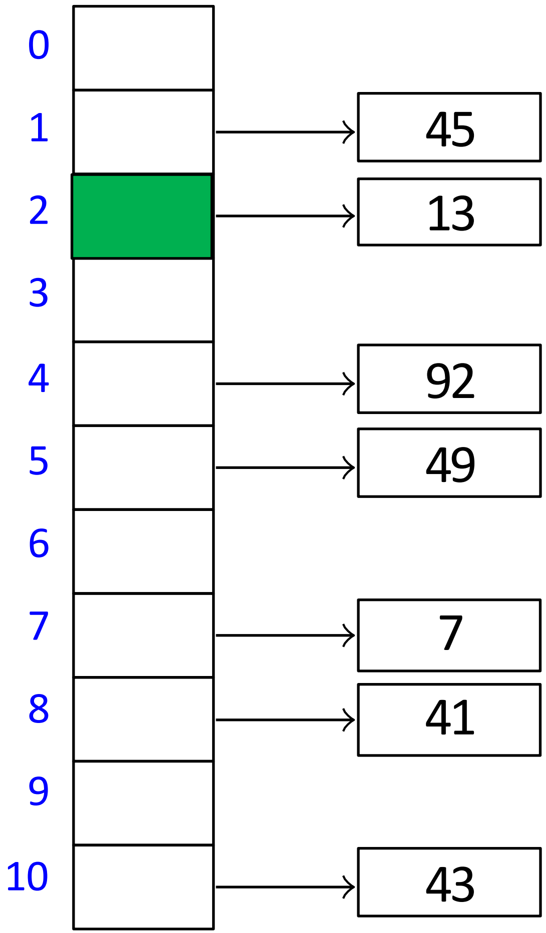


Hashing with Chaining Example

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

insert(46)

$h(46) = 2$

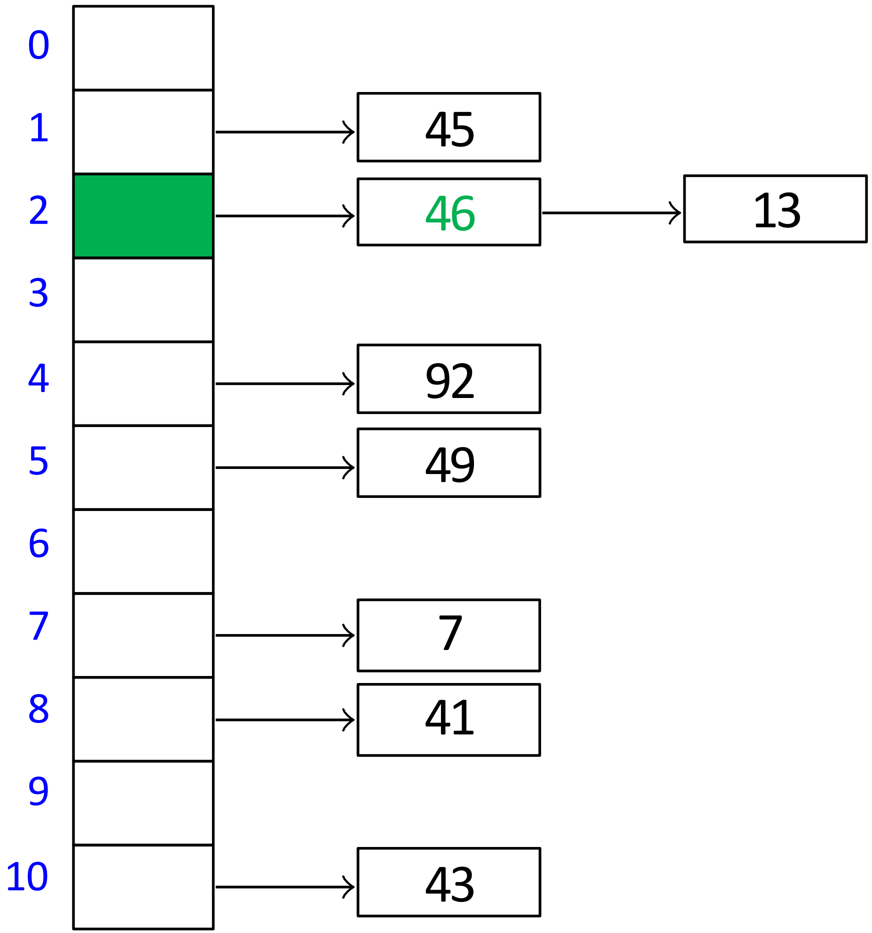


Hashing with Chaining Example

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

insert(46)

$h(46) = 2$

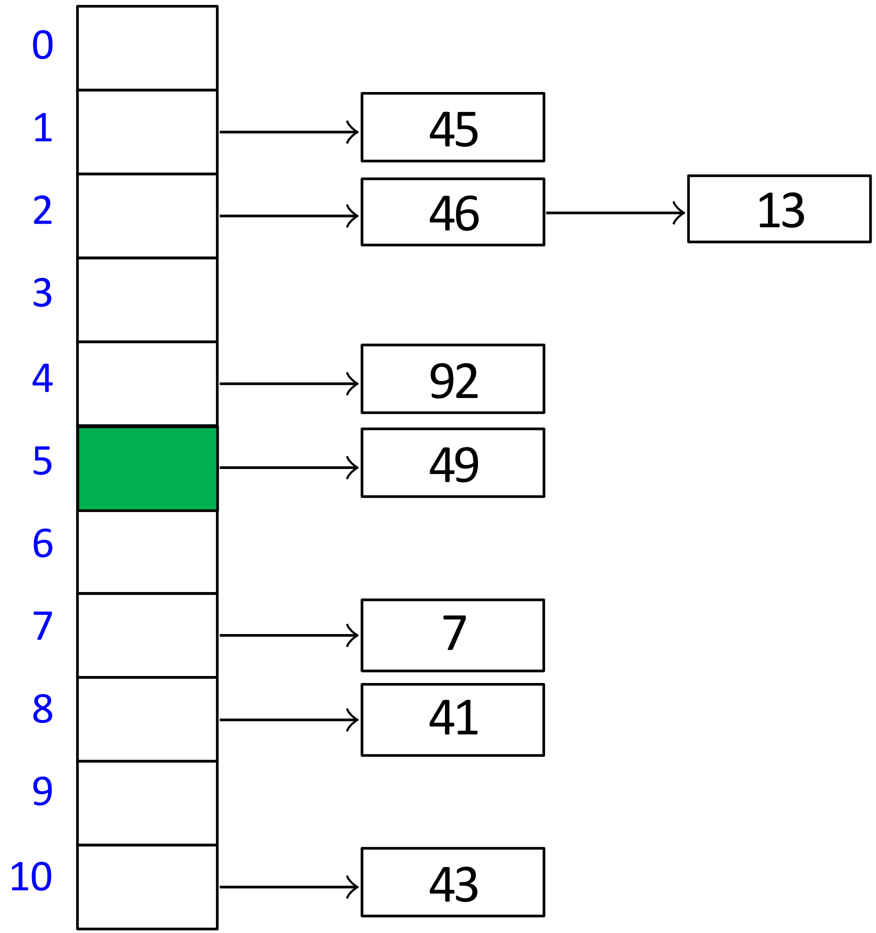


Hashing with Chaining Example

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

insert(16)

$h(16) = 5$

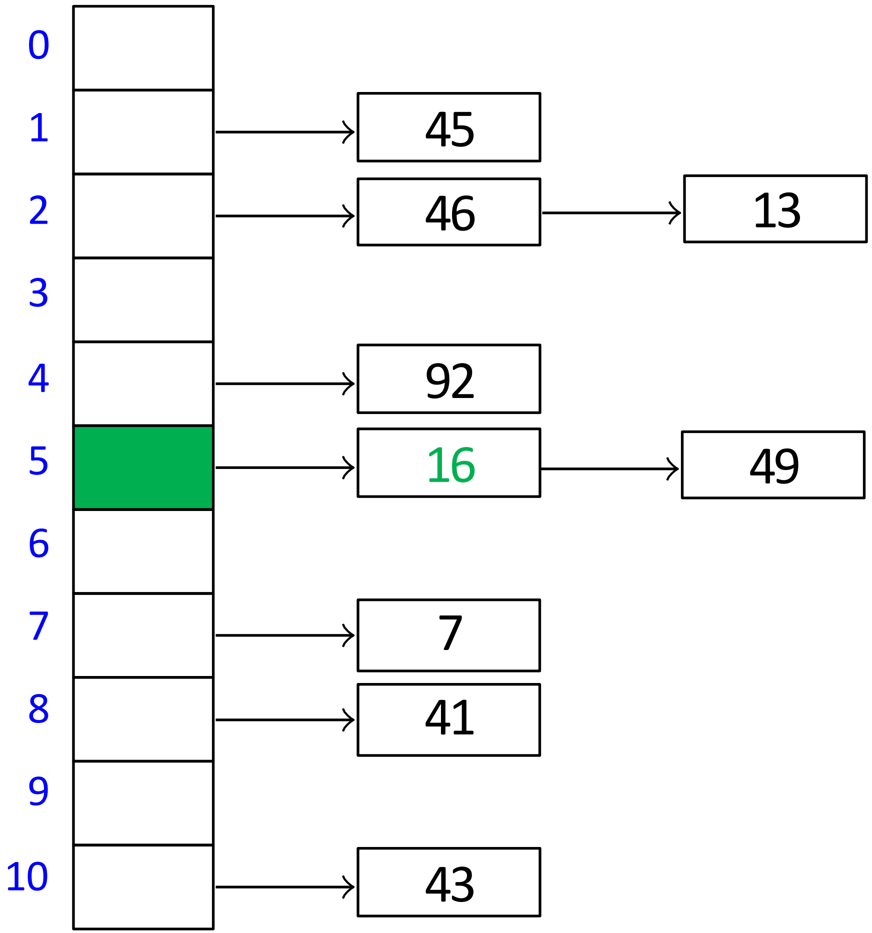


Hashing with Chaining Example

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

insert(16)

$h(16) = 5$

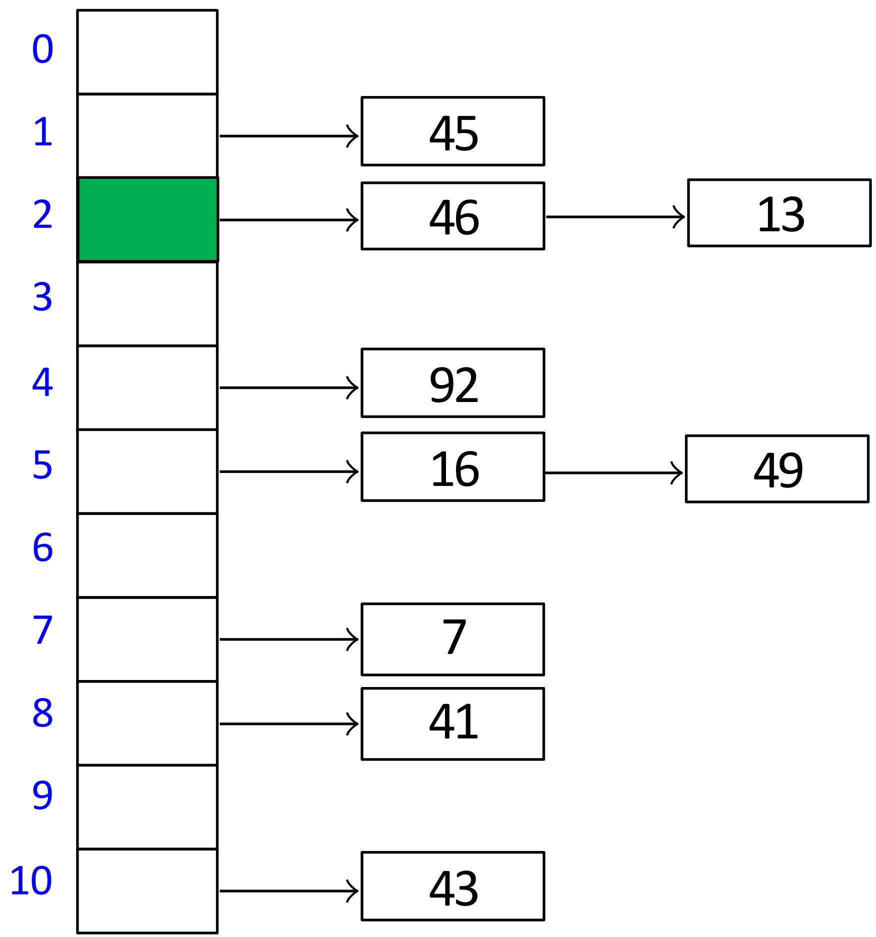


Hashing with Chaining Example

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

insert(79)

$h(79) = 2$

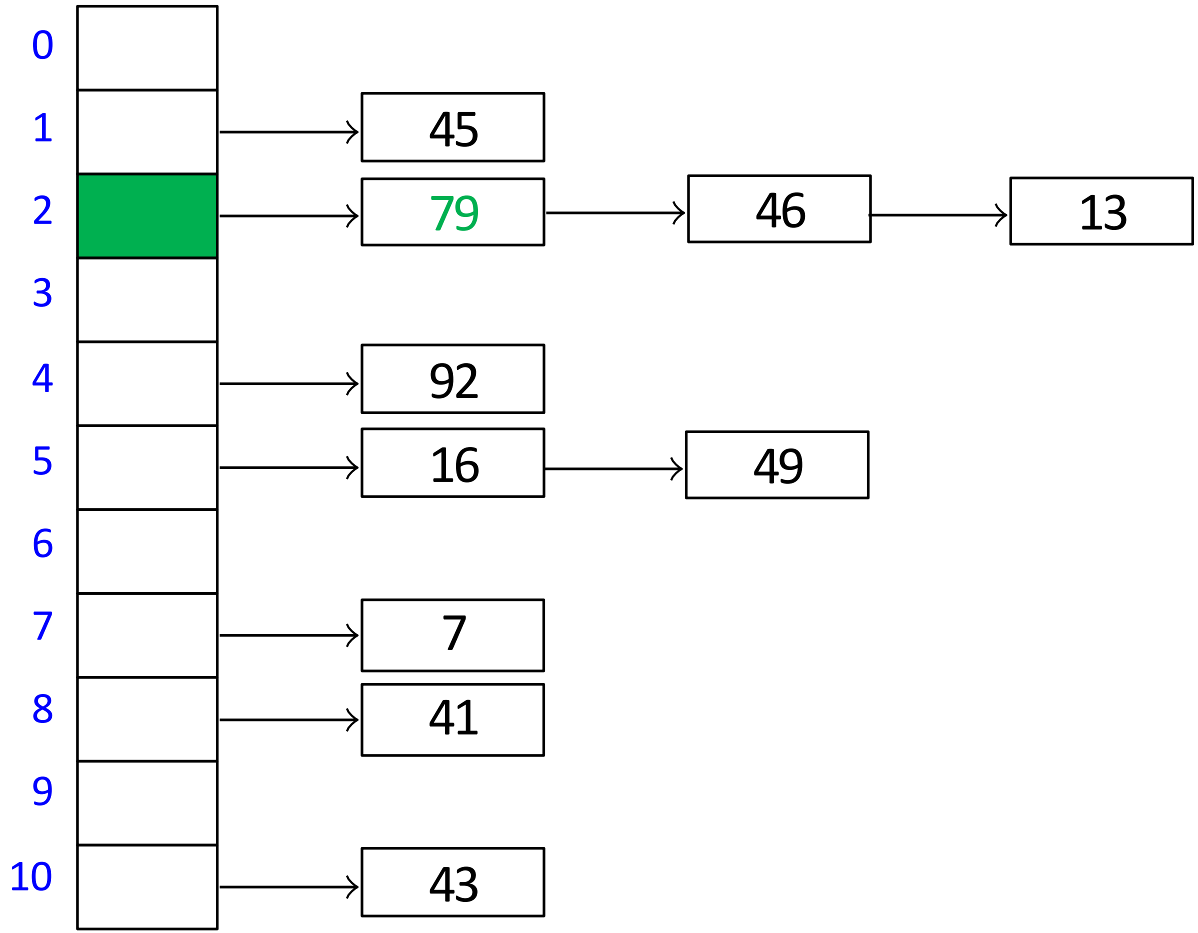


Hashing with Chaining Example

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

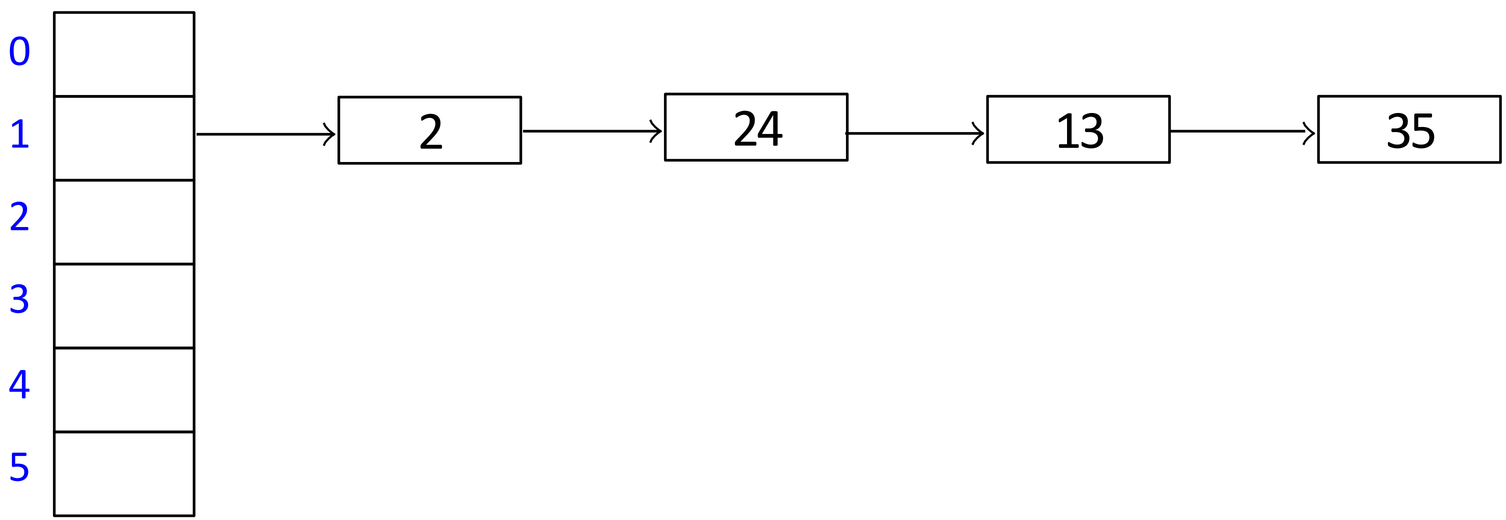
insert(79)

$h(79) = 2$



Hashing with Chaining: Running Time

- *insert* is $\Theta(1)$
  - unordered linked list insertion
- *search* and *delete* $\Theta(1 + \text{length of list at } T[h(k)])$
  - **not** $\Theta(\text{length of list at } T[h(k)])$, as list length can be 0
- In the *worst case* all n items hash to same array index
  - hash table is essentially a list, and *search* and *delete* $\Theta(n)$



Hashing with Chaining: Worst Case Running Time

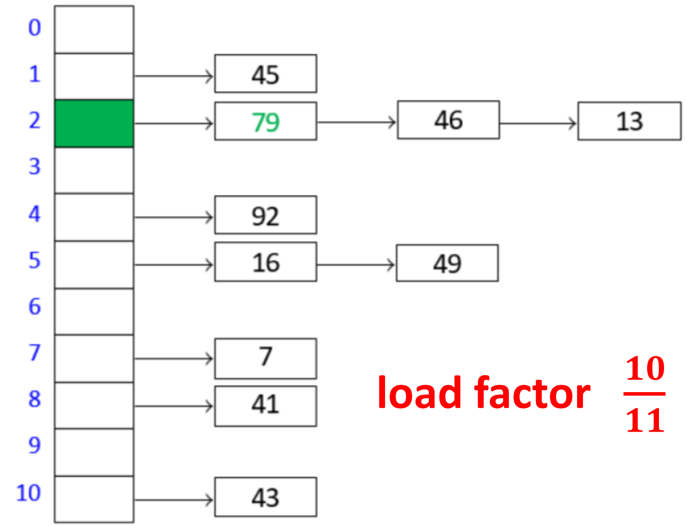
- When can all n items hash to the same array index?
  1. For bad hash function, i.e. $h(k) = 10$
  2. For *any* hash function, if universe is large enough, there are n keys that hash to the same slot

     Proof:
     - let $|U| \geq M(n-1)+1$
     - suppose at most $n-1$ keys hash to each table slot

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

     - then there at most $M(n-1)$ elements in U, contradiction
- The user may need to insert n keys that happen to hash to same slot

Hashing with Chaining: Average Case Runtime?



- Define *load factor* $\alpha = \frac{n}{M}$
    - n is the number of items
    - M is the size of hash table
- Average bucket size $= \frac{n}{M} = \alpha$
- This **does not** imply that average-case runtime of search and delete is $\Theta(1+\alpha)$
    - consider the case when user inserts keys which all hash to the same slot
    - average bucket-size is still α
    - but search and delete nevertheless take $\Theta(n)$ on average
    - message: when you hear 'average', ask 'average over what'
- To get meaningful average-case bounds, we need some assumptions on hash-function and keys the user will insert
    - hard to make realistic assumptions
- Easier to switch to *randomized* hashing

Hashing with Chaining: Randomization

- How can we randomize?
  - cannot insert at a random location, as key k must hash to the hash value $h(k)$
- Idea: assume the hash-function is chosen randomly from a set of all hash functions
- This is called *Uniform Hashing Assumption (UHA):* any possible hash-function is equally likely to be chosen
  - not realistic, but this assumption makes analysis easier

Uniform Hashing Assumption Properties

- Under UHA (any hash-function is chosen equally likely)

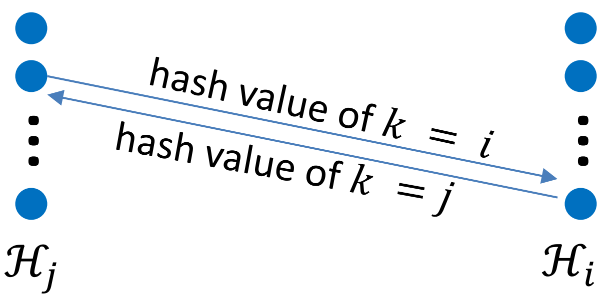
    1. $P(h(k) \;=\; i) \;= \frac{1}{M}$ for any key k and slot i

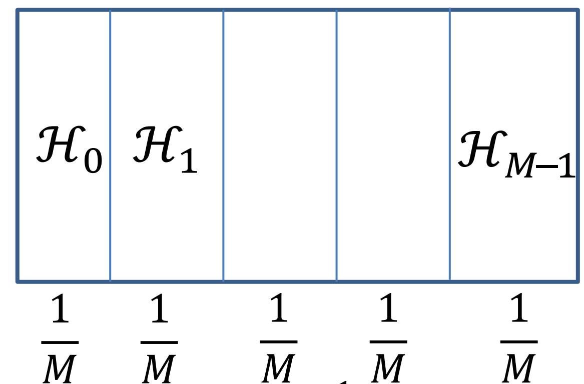
    Proof:

    Let k, i be some key and slot

    Let $\mathcal{H}_j$ (for $j = 0, .. M-1$) be set of hash-functions h s.t. $h(k) = j$

    For $j \neq i$, one-to-one map between $\mathcal{H}_j$ and $\mathcal{H}_i$





size of $\mathcal{H}_j$ equal to size of $\mathcal{H}_i$

when sampling $h(k)$ end up in $\mathcal{H}_j$ with probability $\frac{1}{M}$

$$P(h(k) = \; i) = P(h(k) \in \mathcal{H}_i) = \frac{1}{M}$$

2. hash-values of any two keys are independent of each other

$$P(h(k) = \; i \;\text{ and } h(k') = j) = P(h(k) = \; i)P(h(k') = j)$$

Proof: …

Hashing with Chaining with Randomly Chosen Hash Function

- $P(h(k) = i) = \frac{1}{M}$ for any key k and slot i
- *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 $\frac{1}{M}$
    - let $I_q^i = 1$ if key q hashes to i and $I_q^i = 0$ otherwise
    - $E[|T[i]|] = E\left[\sum_{\text{keys } q} I_q^i\right] = \sum_{\text{keys } q} E[I_q^i] = \sum_{\text{keys } q} Pr(I_q^i = 1) = \frac{n}{M} \leq 1 + \alpha$
- 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

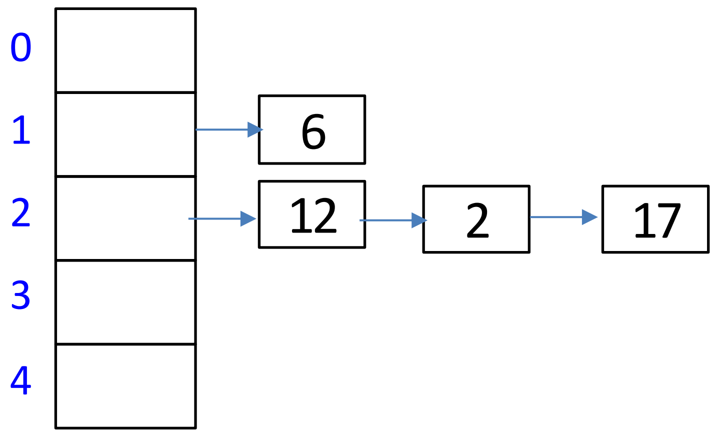
- Load factor $\alpha = \frac{n}{M}$
- Expected *space* is $\Theta(M + n) = \Theta(n/\alpha + n)$, expected *time* is $\Theta(1 + \alpha)$
    - if we maintain $\alpha \in \Theta(1)$, expected running time is $O(1)$ and space is $\Theta(n)$

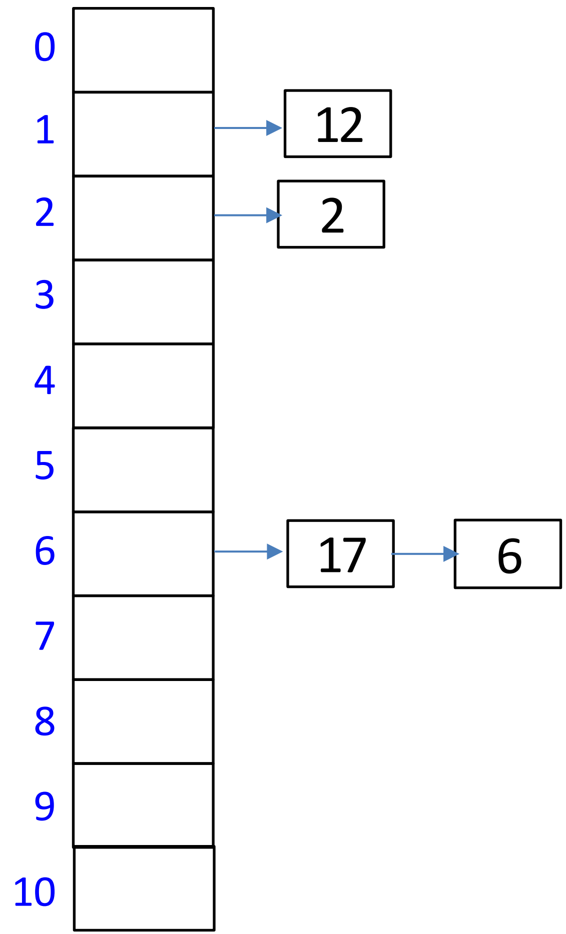
- Maintaining hash array of appropriate size
    - start with small M
    - during insert/delete, update n
    - if load factor becomes too big, i.e. $\alpha = \frac{n}{M} > maxLoadF$, rehash
        - chose new $M' \approx 2M$
        - find a new random hash function h' that maps U into $\{0,1, \ldots M' - 1\}$
        - 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. $\alpha = \frac{n}{M} < minLoadF$, rehash with smaller M'
- Rehashing costs $\Theta(M + n)$ but happens rarely, cost amortized over all operations

Rehashing when Load Factor Too Large

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

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





Randomization in Practice

- Uniform Hashing Assumption is not possible to satisfy in practice
- In practice can chose a random hash function from a certain *family* of hash function
- The following family of functions is often used
    - choose prime number $p > M$ and *random* $a, b \in \{0, \ldots p-1\},\ a \neq 0$
    - $h(k) = \big((ak + b) \bmod p\big) \bmod M$
    - can show that the expected runtime of search/delete hold in this case

Hashing with Chaining Summary

- Rehash so that $\alpha \in \Theta(1)$
- Rehashing costs $\Theta(M + n)$ time (plus the time to find a new hash function)
- Rehashing happens rarely enough that we can ignore this term when amortizing over all operations
- We should also re-hash when α gets too small, so that $M \in \Theta(n)$ and the space is always $\Theta(n)$
- The amortized expected cost for hashing with changing is and the space is $O(1)$
  - assuming uniform hashing and $\alpha \in \Theta(1)$throughout
- Theoretically perfect, but slow in practice

Outline

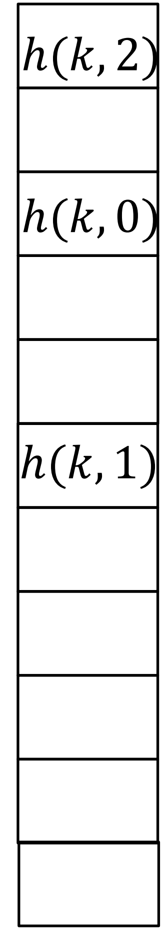
- Dictionaries via Hashing
  - Hashing Introduction
  - Hashing with Chaining
  - Open Addressing
    - probe sequences
    - cuckoo hashing
  - Hash Function Strategies

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

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

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

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

insert(84)

$h(84) = 7$

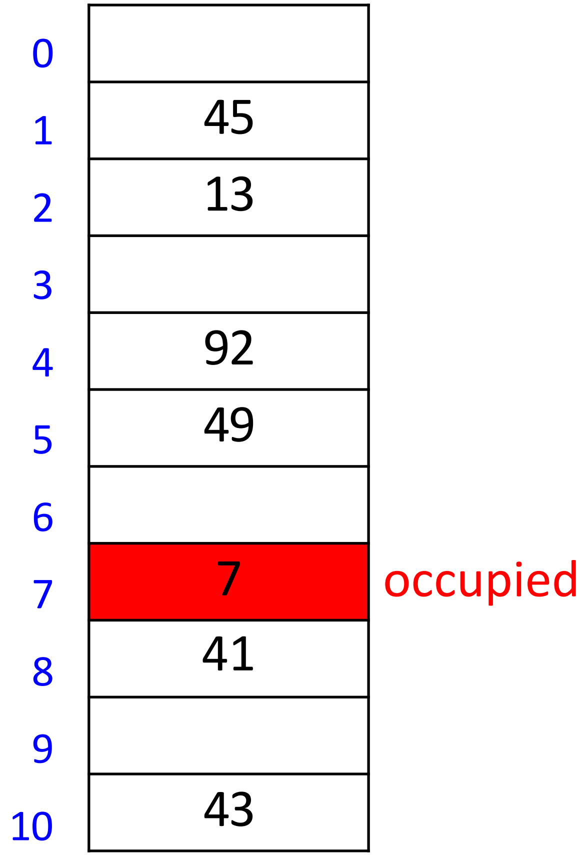
0	
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	
10	43

Linear Probing Example

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

insert(84)

$h(84) = 7$



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

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

$h(k, i) = (h(k) + i) \bmod M$ 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
10	43

Linear probing example Continued

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

$h(k, i) = (h(k) + i)$ mod M 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	occupied
10	43	

Linear probing example Continued

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

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

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

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

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

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

search(23)

$h(23) = 1$

$h(23, 1) = (1 + 1) \mathrm{mod}\, 11 = 2$

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

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

$h(k, i) = (h(k) + i)$ mod M 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

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

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

$\textit{delete}(84)$

$h(84) = 7$

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

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

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

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

$delete(84)$

$h(84) = 7$

$h(84, 2) = (7 + 2) \bmod 11 = 9$

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
    - keep track of how many items are *deleted* and re-hash if there are too many
      - to keep space $\Theta(n)$

Linear probing: Delete

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

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

$delete(84)$

$h(84) = 7$

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

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

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

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

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

$h(k, i) = (h(k) + i)$ mod M 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	deleted	
10	43	

Linear probing example

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

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

search(20)

$h(20) = 9$

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

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

Linear probing example

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

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

search(20)

$h(20) = 9$

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

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

Linear probing example

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

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

$search(20)$

$h(20) = 9$

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

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

Linear probing example

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

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

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

Open Addressing

- Can use lazy deletion for other data structures
  - mark as deleted items in AVL tree instead of actual deletion
  - if a lot of items are deleted, rebuild AVL tree
- While in other data structures lazy deletion can be used to improve performance, in probing lazy deletion is required for correct performance

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)
    for $(i = 0; i < M; i{+}{+})$
      if $T[h(k,i)]$ is *empty*
        return *item-not-found*
      if $T[h(k,i)]$ has key k **return** $T[h(k,i)]$
      // $T[h(k,i)] =$ deleted or not in the data structure
      // therefore keep searching
    return *item not found*

Linear probing drawbacks

- Entries tend to cluster into contiguous regions
- 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

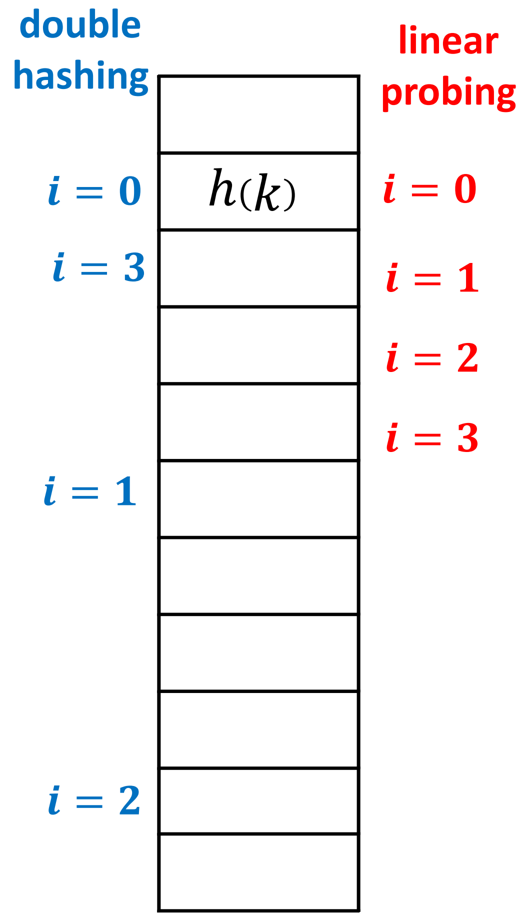
- Linear probing attempts inserting into consecutive locations, i.e. step size 1

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

- To avoid consecutive locations, let each key have its own step size

$$h(k) \quad h(k)+1\cdot step(k) \quad h(k)+2\cdot step(k)$$

- This helps to avoid the clustering side effect
- For each key k, probe sequence is always the same
- Example
  - for $k=14$, probe sequence is always
    - 4, 7, 10, 13
  - for $k=24$, probe sequence is always
    - 5, 10, 15, 20

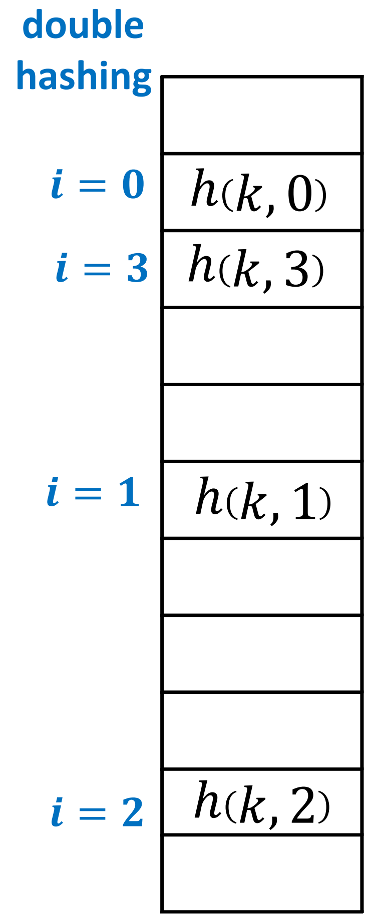


Double Hashing

- *Double hashing* : open addressing with probe sequence

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

- Where
  - h_1 is a secondary hash function (step size) s.t. $h_1(k) \neq 0$
  - $h_1(k)$ is relative prime with M for all keys k
    - otherwise probe-sequence does not explore the entire hash table
    - easiest to choose M prime, and ensure $h_1(k) < M$
- 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_0, h_1 are required, they should be independent

$$P(h_0(k) = i, h_1(k) = j) = P(h_0(k) = i\ )\ P(h_1(k) = j)$$

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

$$0 \leq \text{fractional part of } kA < 1$$

$$0 \leq M \cdot (\text{fractional part of } kA) < M$$

- Example: $M = 11, A = 0.2$
  - $h(34) = \lfloor 11 \cdot (34 \cdot 0.2 - \lfloor 34 \cdot 0.2 \rfloor) \rfloor = \lfloor 11 \cdot (6.8 - \lfloor 6.8 \rfloor) \rfloor = \lfloor 11 \cdot 0.8 \rfloor = 8$
- Multiplying with A scrambles the keys
  - should use at least $\log |U| + \log |M|$ bits of A
- $A = \varphi = \frac{\sqrt{5}-1}{2} \approx 0.618033988749$ works well
- For double hashing, to ensure $0 < h(k) < M$, use

$$h_1(k) = \lfloor \underbrace{(M-1)(kA - \lfloor kA \rfloor)}_{\text{for table size } M-1\text{: } 0 \leq \text{values} < M-1} \rfloor + 1$$

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)

$$h_0(194) = 7$$

$$h_1(194) = 9$$

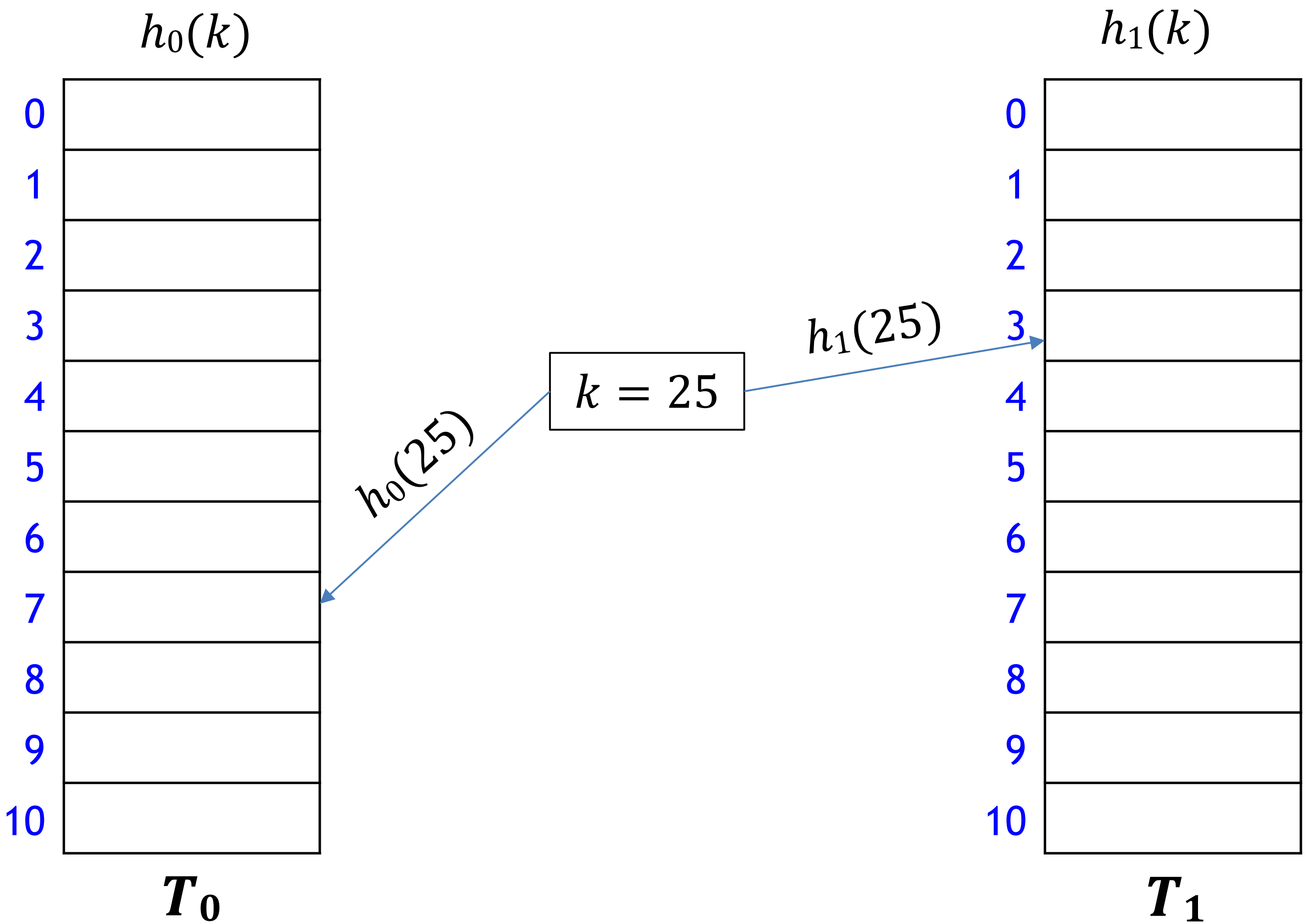
$$h(194, 2) = (7 + 2 \cdot 9)\ mod\ 11 = 3$$

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

Outline

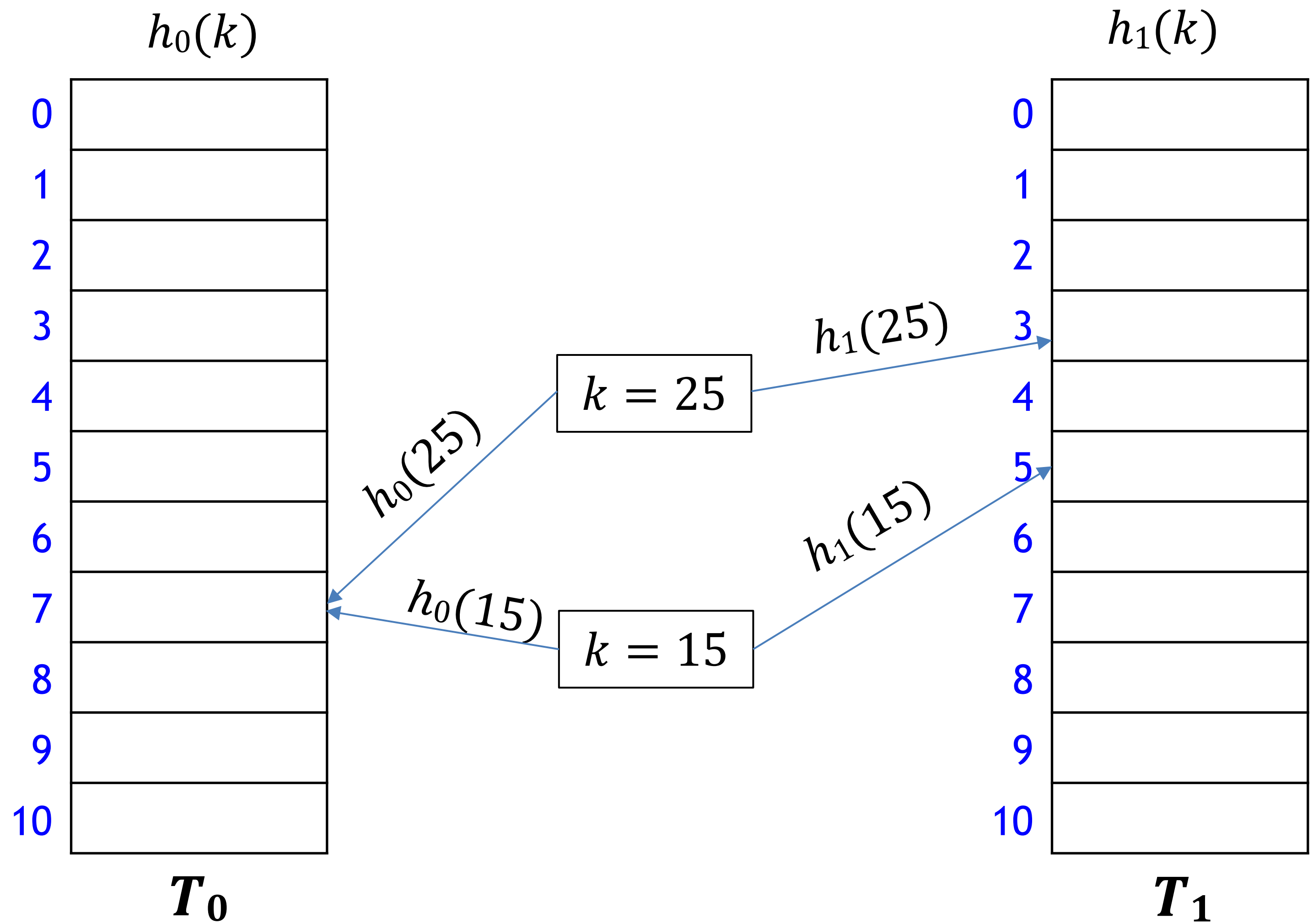
- Dictionaries via Hashing
  - Hashing Introduction
  - Hashing with Chaining
  - Open Addressing
    - probe Sequences
    - cuckoo hashing
  - Hash Function Strategies

Cuckoo Hashing



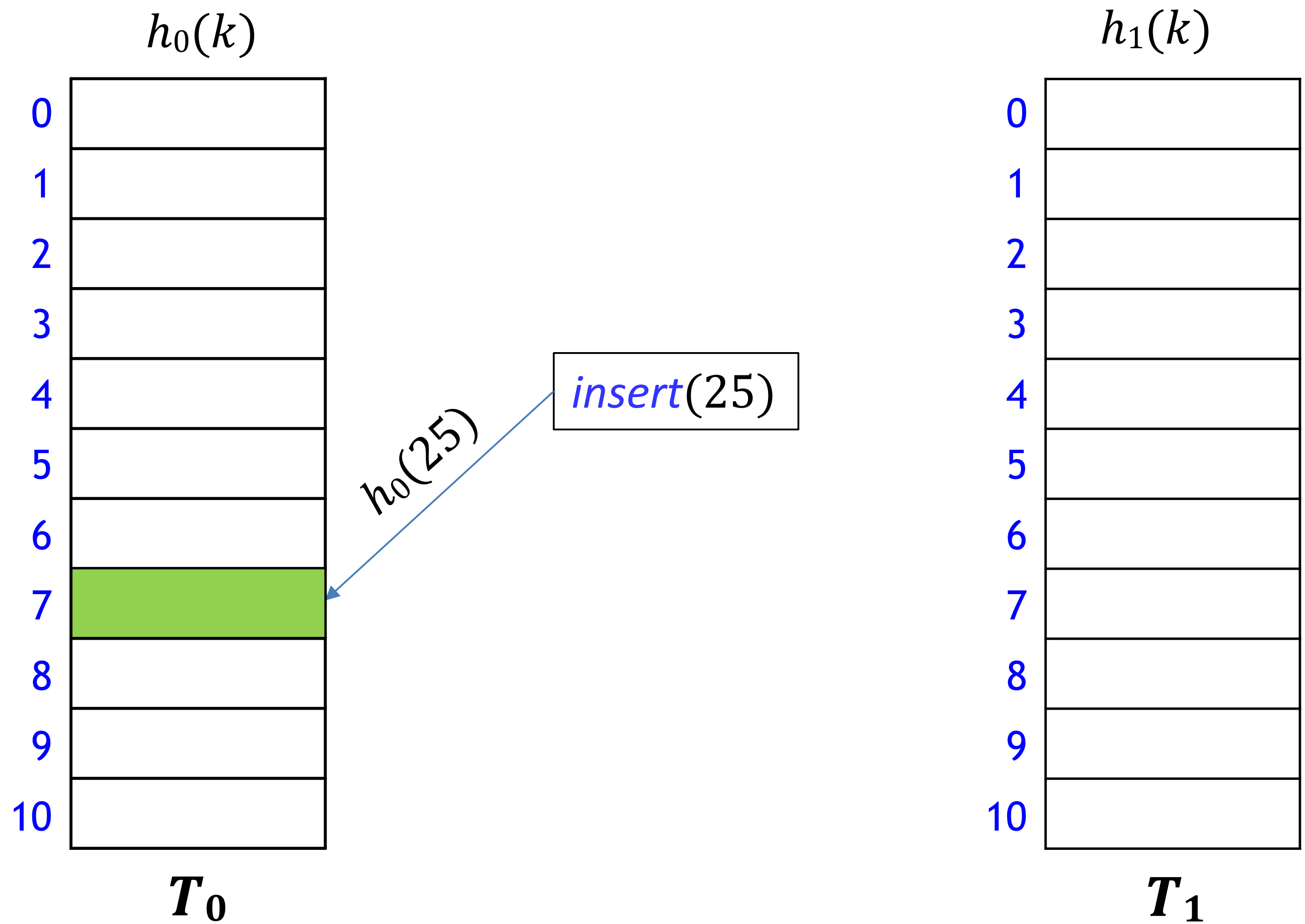
- **Main idea:** An item with key k can be only at $T_0[h_0(k)]$ or $T_1[h_1(k)]$

Cuckoo Hashing



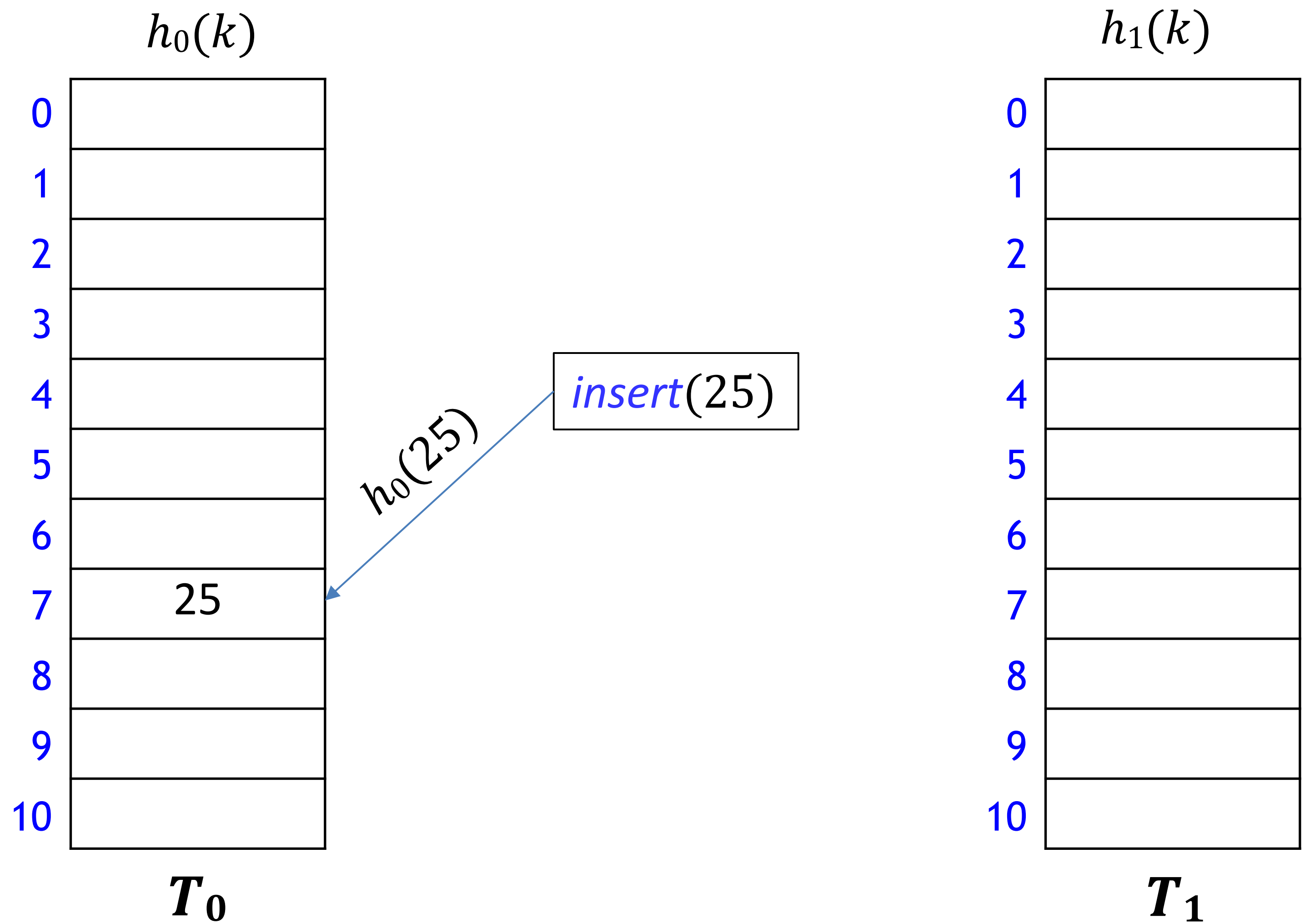
- **Main idea:** An item with key *k* can be only at $T_0[h_0(k)]$ or $T_1[h_1(k)]$
  - *search* and *delete* take $O(1)$ time

Cuckoo Hashing



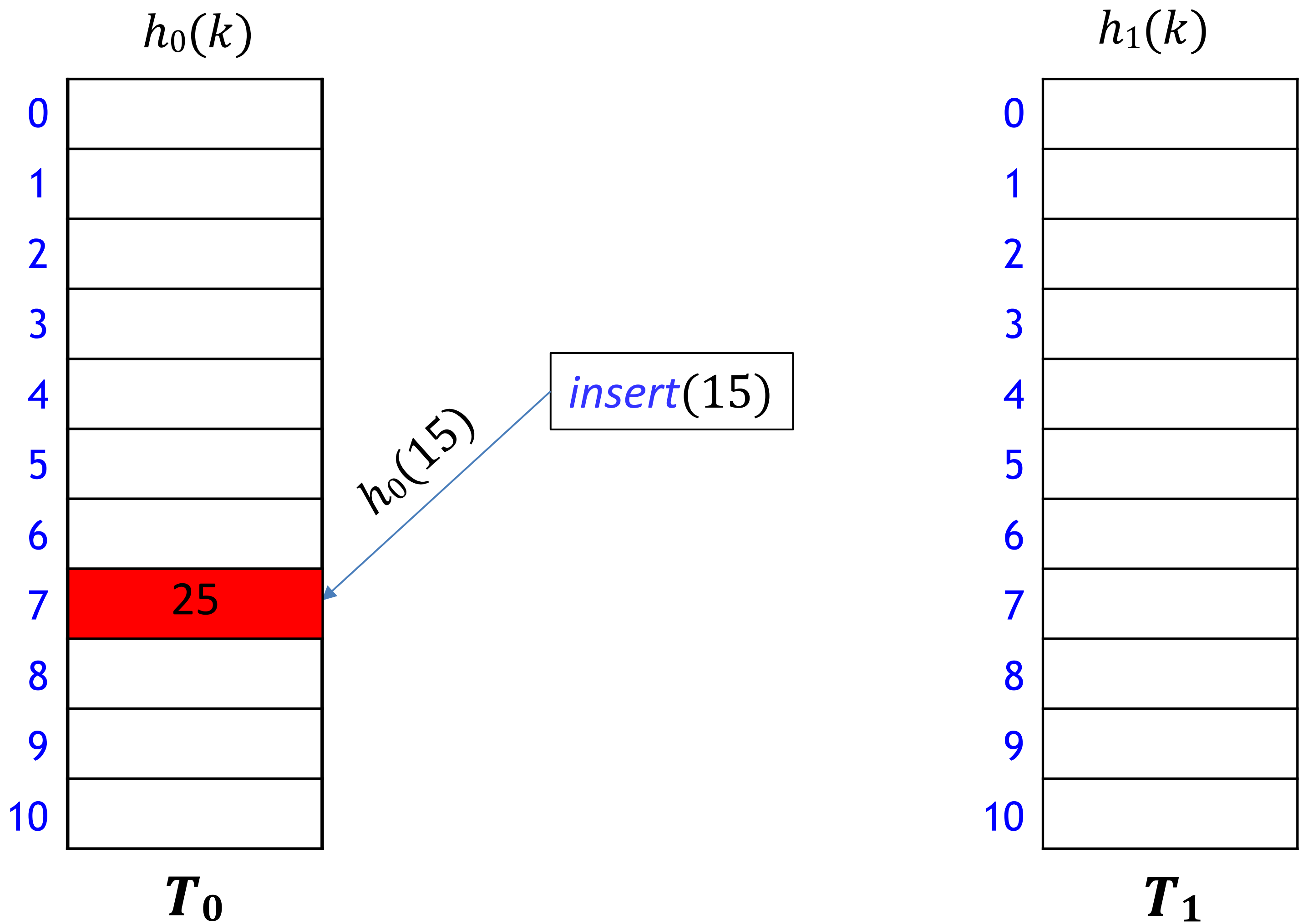
- How to insert?

Cuckoo Hashing



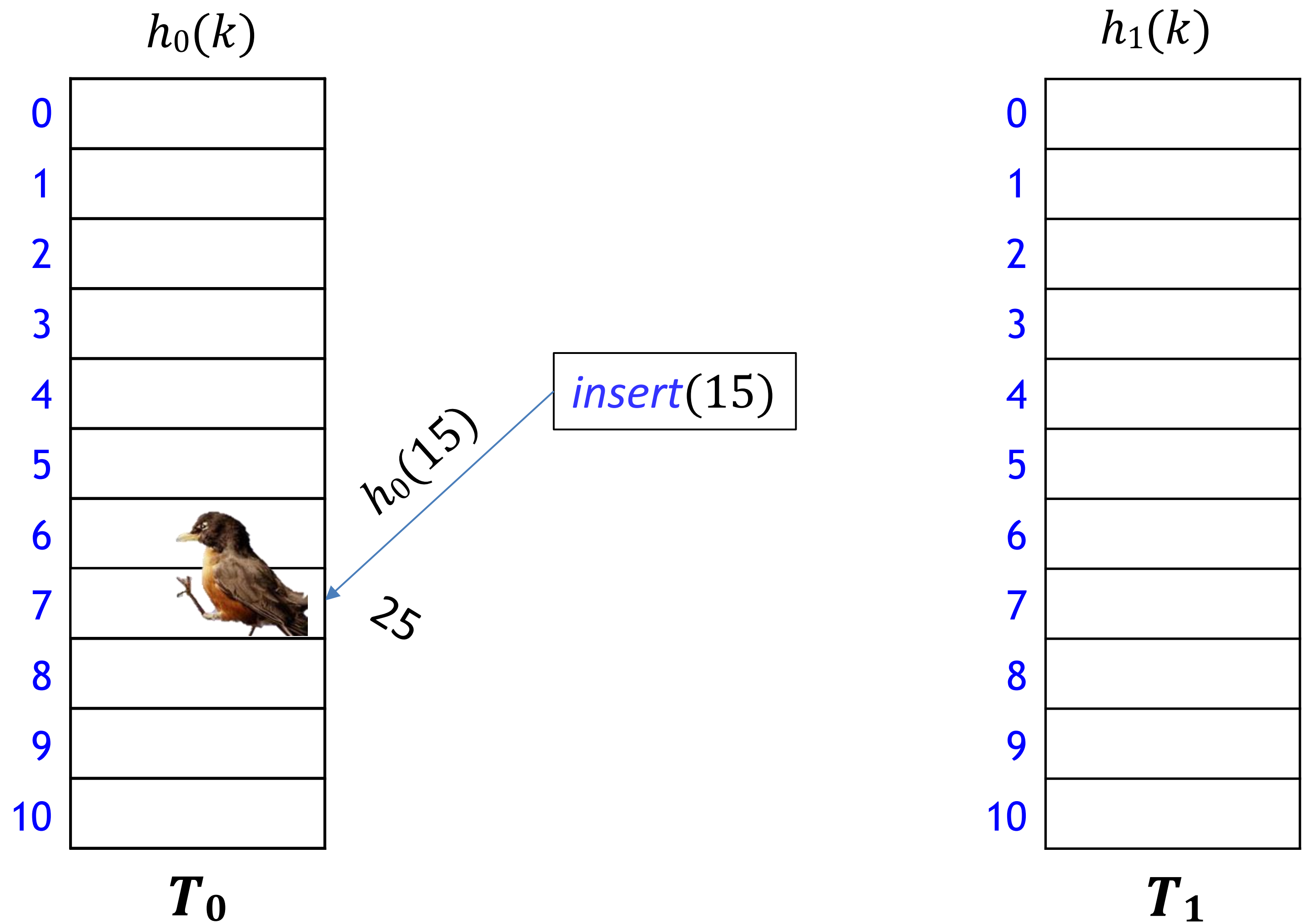
- How to insert?

Cuckoo Hashing



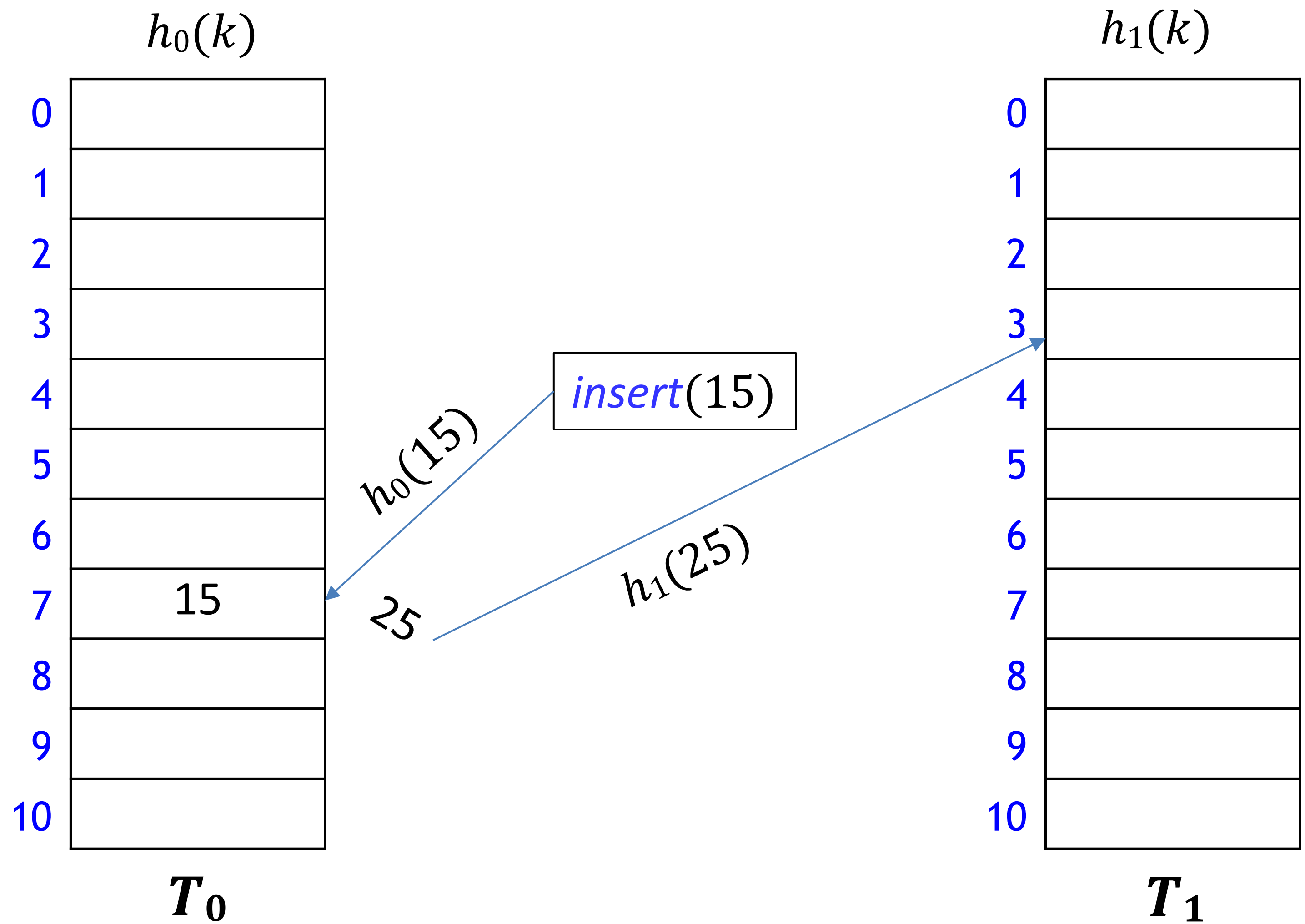
- How to insert k when $h_0(k)$ is already occupied?

Cuckoo Hashing



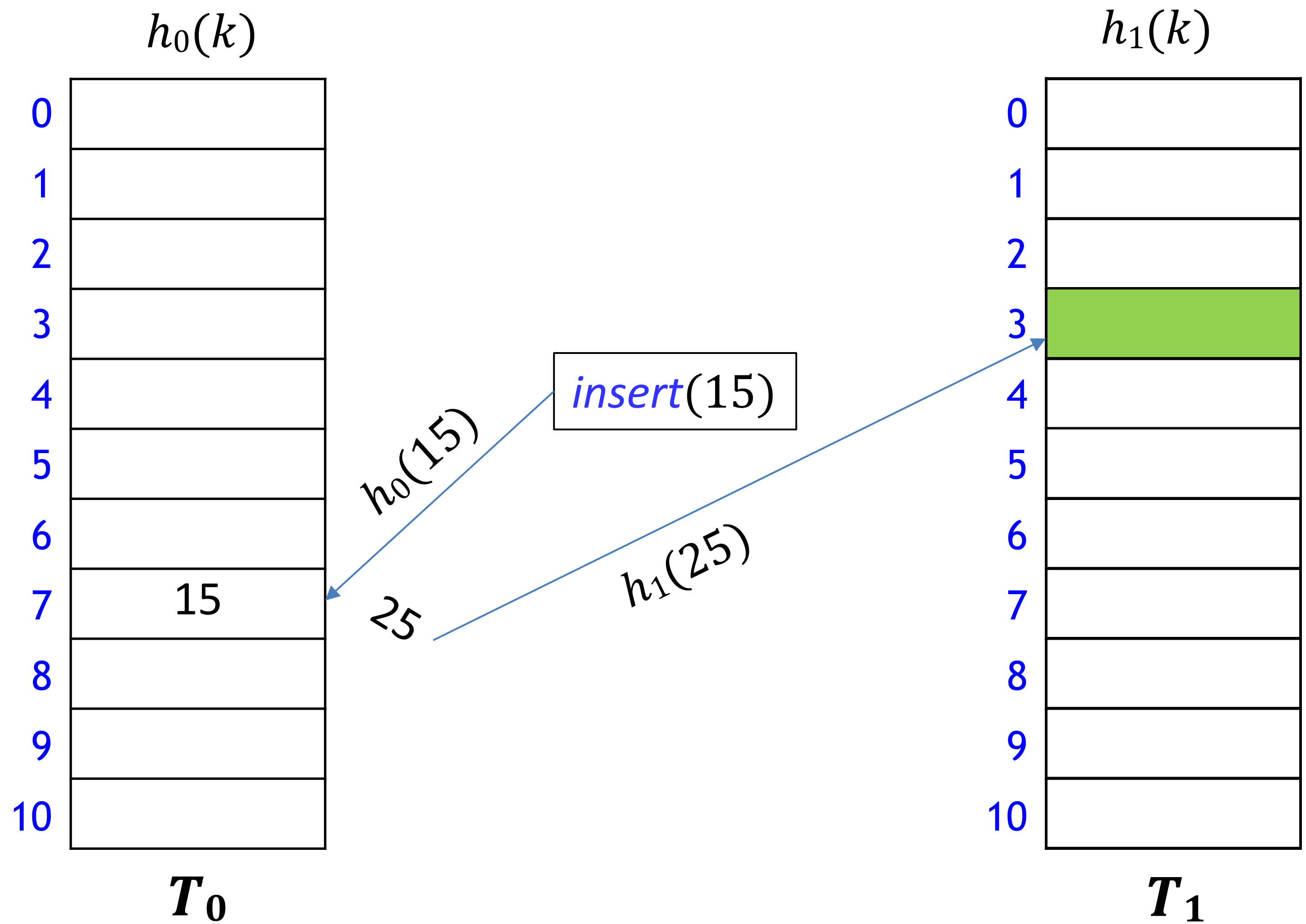
- How to insert k when $h_0(k)$ is already occupied?

Cuckoo Hashing



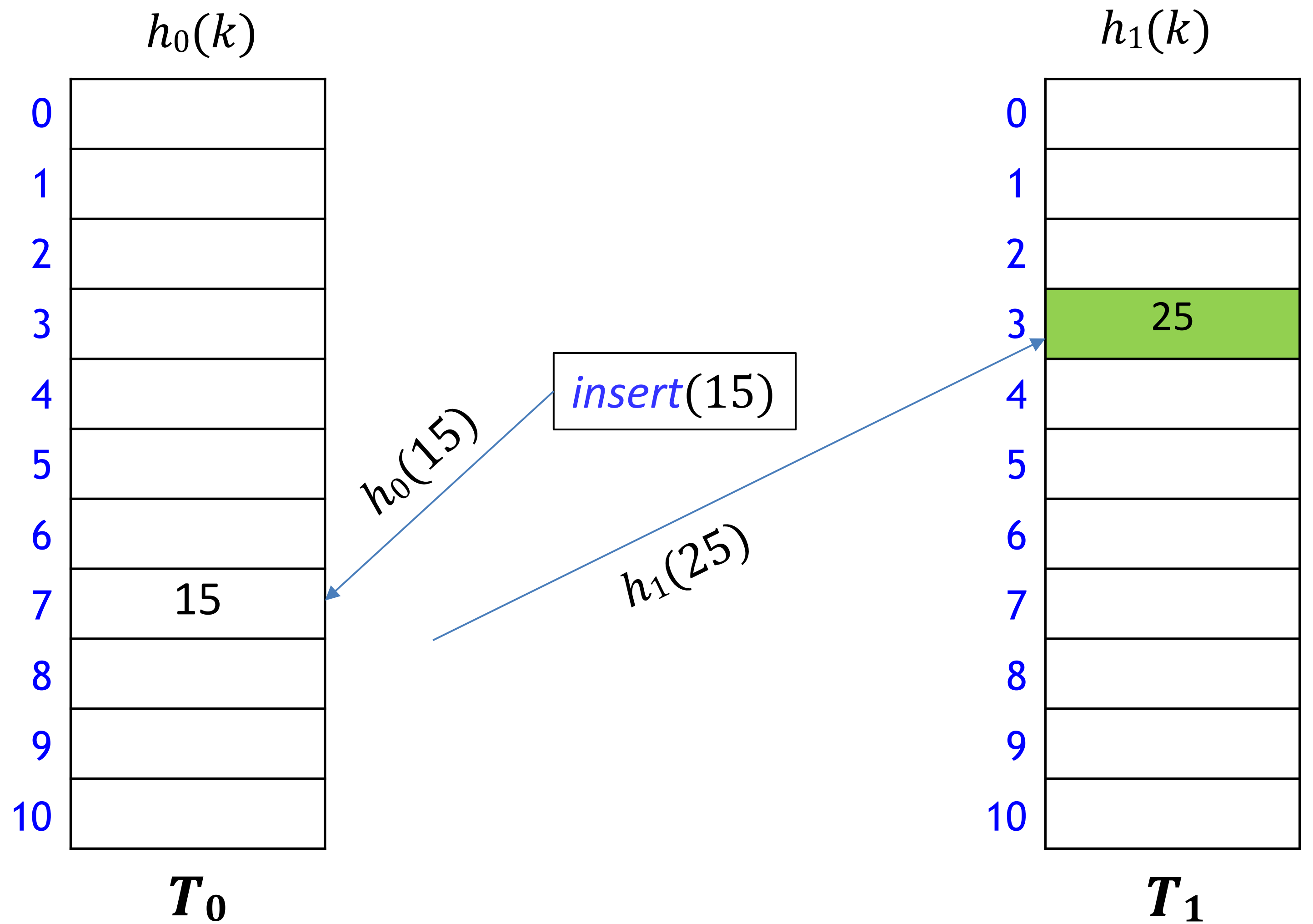
- How to insert k when $h_0(k)$ is already occupied?

Cuckoo Hashing



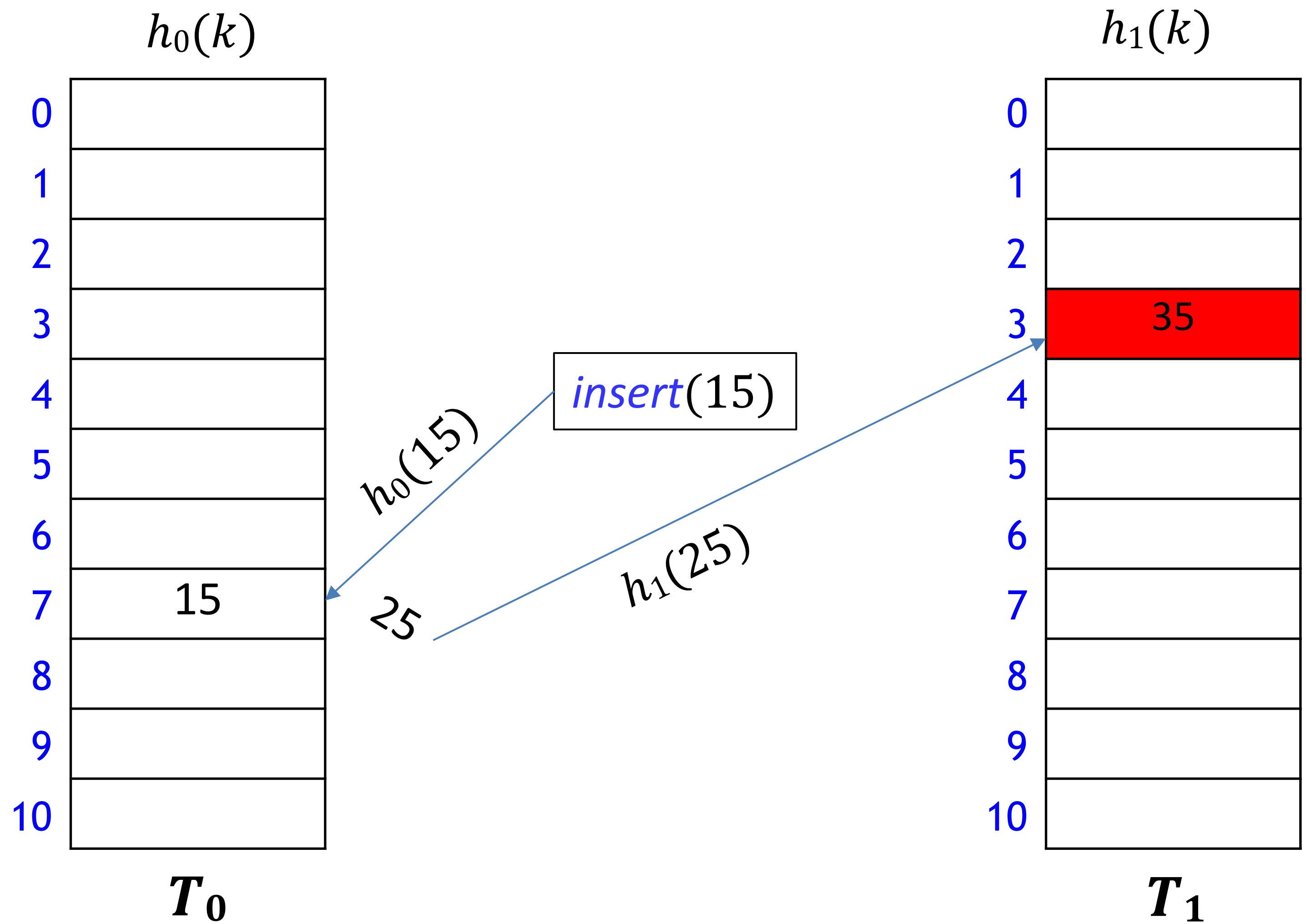
- How to insert k when $h_0(k)$ is already occupied?

Cuckoo Hashing



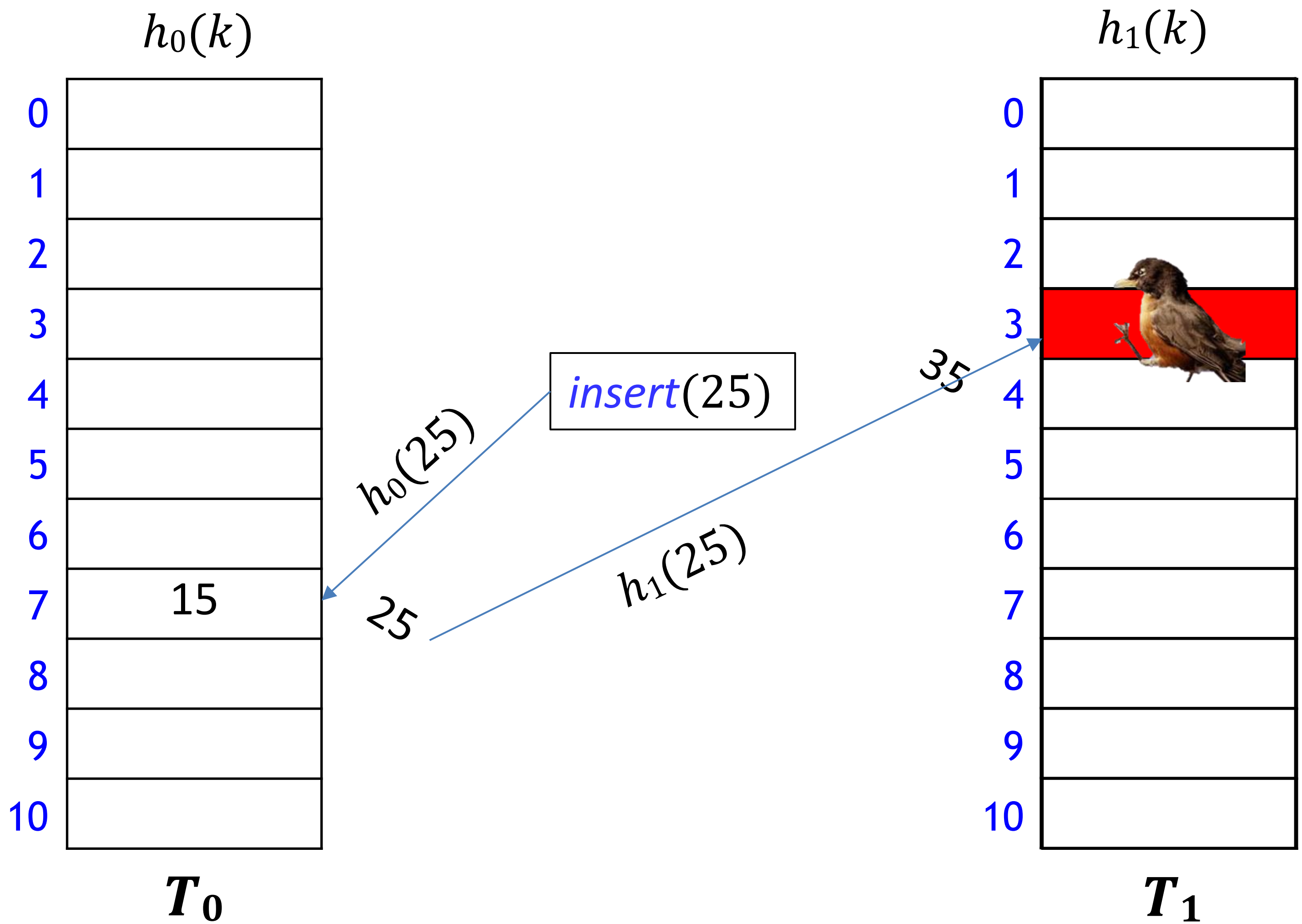
- How to insert k when $h_0(k)$ is already occupied?

Cuckoo Hashing



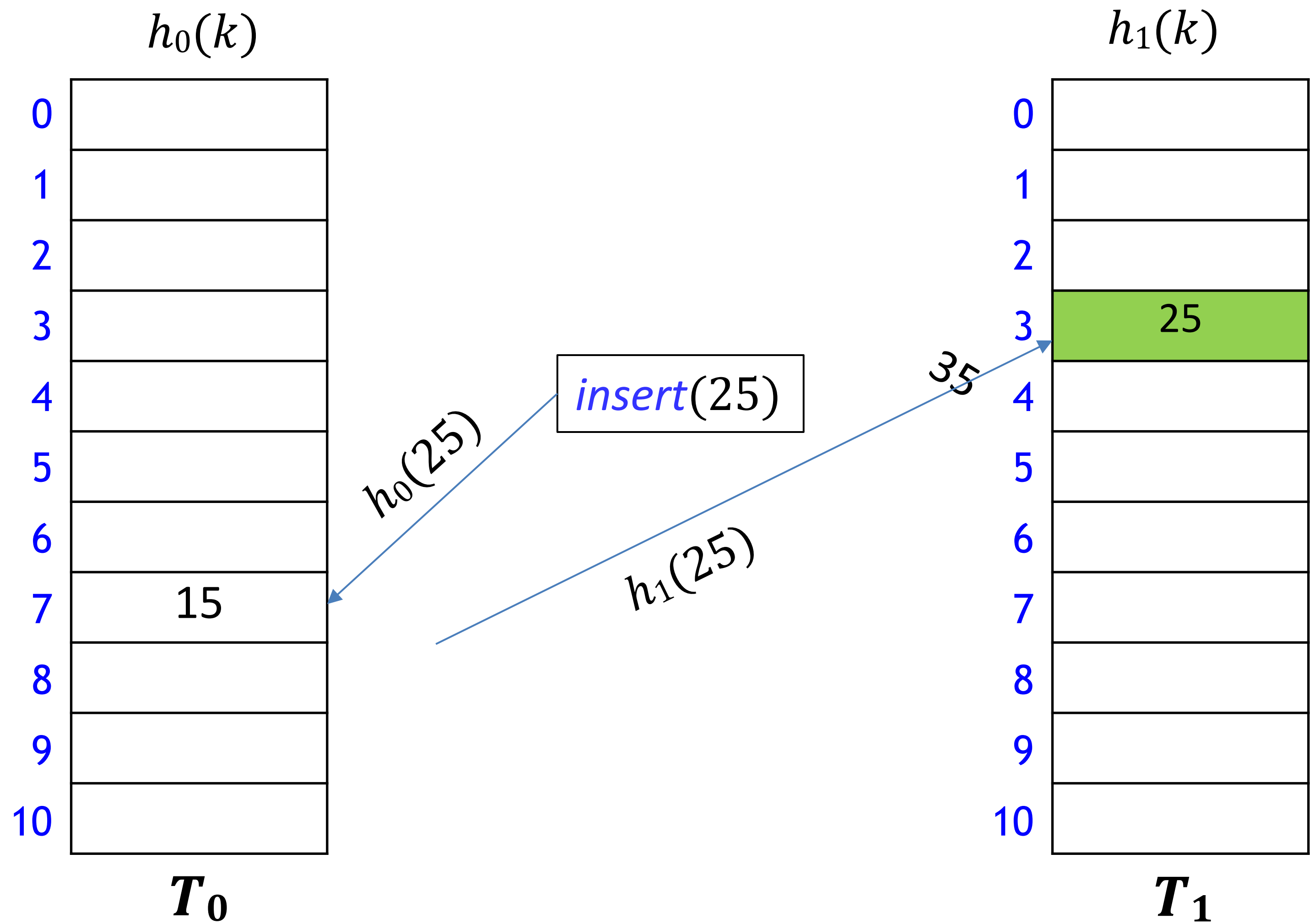
- How to insert k when $h_0(k)$ is already occupied?

Cuckoo Hashing



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

Cuckoo Hashing

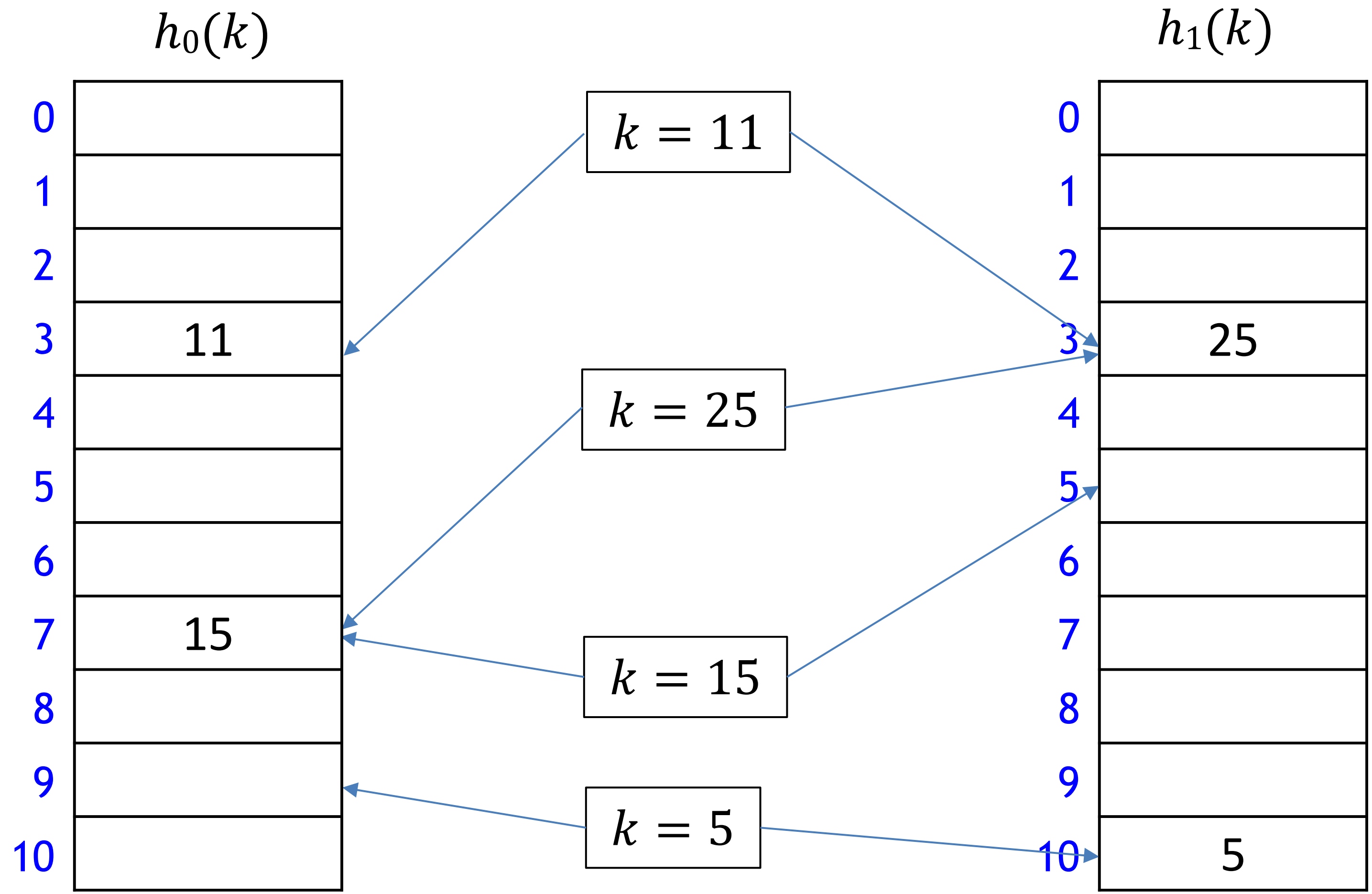


- Continue until all items placed, or *failure*
    - rehash if failure

Cuckoo Hashing [Pagh & Rodler, 2001]

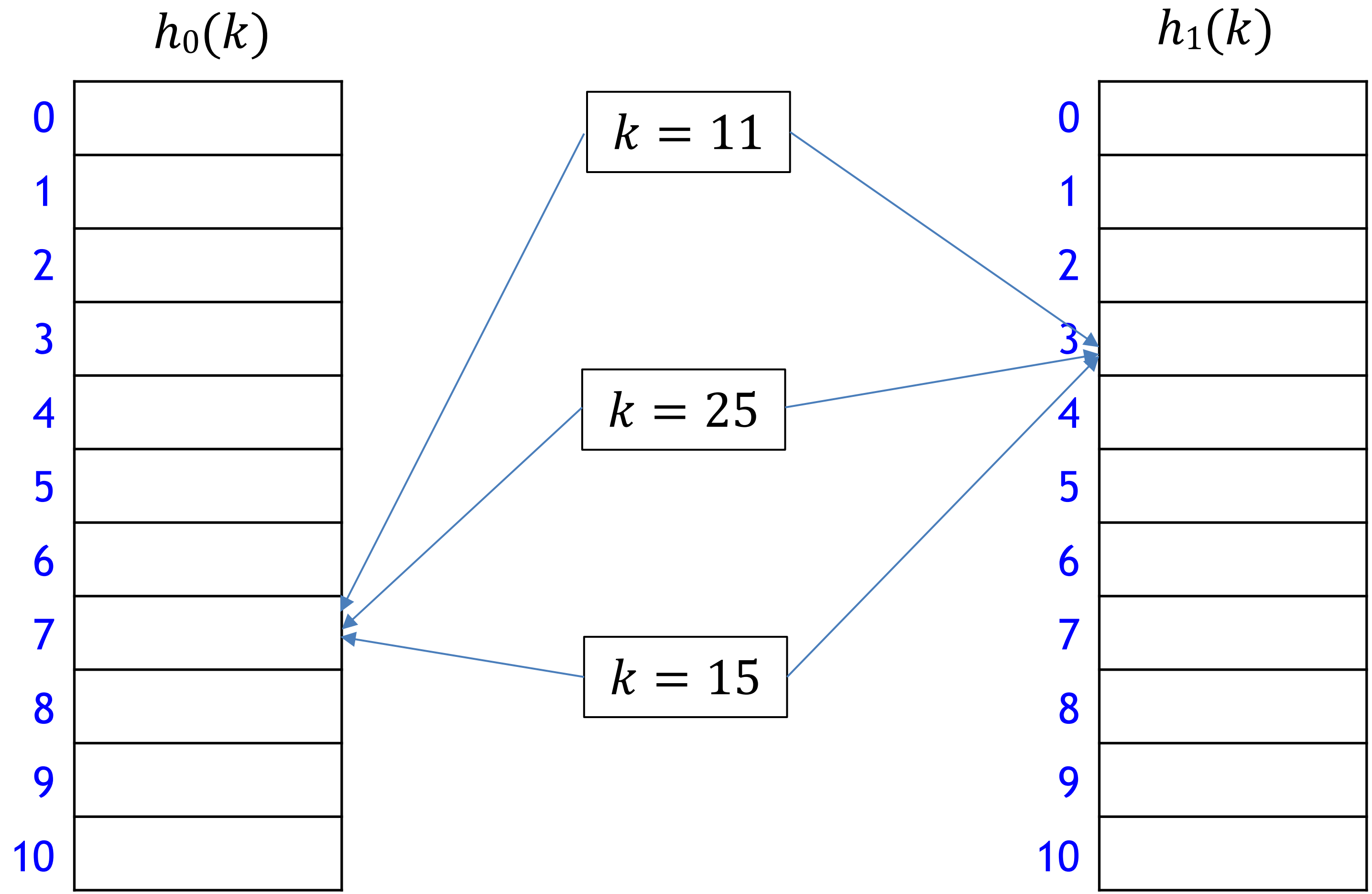
- Use independent hash functions h_0, h_1 and two tables T_0, T_1
- Key k can be only at $T_0[h_0(k)]$ or $T_1[h_1(k)]$
  - *search* and *delete* take constant time
  - *insert* always initially puts key k into $T_0[h_0(k)]$
    - evict item that my have been there already
    - if so, evicted item k' is inserted at $T_1[h_1(k')]$
    - may lead to a loop of evictions
    - can show that if insertion is possible, then there are at most $2n$ evictions
    - so abort after too many attempts

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

Cuckoo hashing: Insert

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

$insert(51)$

$i = 0$

$k = 51$

$h_0(k) = 7$

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	
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	95	51	7	
8			8	
9	92		9	
10			10	

Cuckoo hashing: Insert

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

$insert(95)$

$i = 1$

$k = 51$

$h_1(k) = 5$

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

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

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

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

Cuckoo hashing: Insert

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

$insert(26)$

$i = 0$

$k = 26$

$h_0(k) = 4$

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

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)

$i = 0$

$k = 26$

$h_0(k) = 4$

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

Cuckoo hashing: Insert

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

$insert(26)$

$i = 1$

$k = 59$

$h_1(k) = 5$

59

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

Cuckoo hashing: Insert

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

$insert(26)$
$i = 1$
$k = 59$
$h_1(k) = 5$

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

Cuckoo hashing: Insert

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

$\textit{insert}(26)$

$i = 0$

$k = 51$

$h_0(k) = 7$

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

Cuckoo hashing: Insert

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

$insert(26)$

$i = 0$

$k = 51$

$h_0(k) = 7$

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

Cuckoo hashing: Insert

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

$insert(26)$

$i = 1$

$k = 95$

$h_1(k) = 7$

	T_0			T_1
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

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

$insert(26)$

$i = 1$

$k = 95$

$h_1(k) = 7$

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

- Practical tip
  - do not wait for $2n$ unsuccessful tries to declare failure
  - In practice, declare failure much earlier than $2n$

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

0		
1		
2		
3		
4		
5	59	found
6		
7	95	
8		
9		
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

- Load factor $\alpha = n/(\text{size of } T_0 + \text{size of } T_1)$
- Can show that if the load factor is small enough, then insertion has $O(1)$ expected time
  - this requires $\alpha < 1/2$
  - so wasted space
- Can show expected space is $O(n)$
- There are many variations of cuckoo hashing
  - two hash tables do not have to be of the same size
  - 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

Running Time of Open Addressing Strategies

- For any open addressing scheme, we *must* have $\alpha \leq 1$ (why?)
- For analysis, require $0 < \alpha < 1$, for Cuckoo hashing require $\alpha < 1/2$
  - not arbitrarily close
- Under these restrictions and the Universal Hashing Assumption
  - All strategies have $O(1)$ expected time for search, insert, delete
  - Cuckoo hashing has $O(1)$ worst case for search, delete
  - Probe sequence use $O(n)$ worst case space
  - Cuckoo hashing uses $O(n)$ expected space
- For any hashing, the worst case runtime is $\Theta(n)$ for insert
- In practice, double hashing is the most popular
  - Or cuckoo hashing if there are many more searches than insertions

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 for most of them
- Two ways to compromise
  1. Deterministic: hope for a good performance by choosing a hash function that is
     - unrelated to any possible patterns in the data
     - depends on all parts of the key
  2. Randomized: choose randomly among a limited set of functions
     - but aim for $P(\text{two keys collide}) = \frac{1}{M}$
       - this is enough to prove expected runtime bounds for chaining

Deterministic Hash Functions

- We saw two basic methods (for integer keys)
- Modular method: $h(k) = k \; mod \; M$
    - chose M to be a prime
    - Means finding a suitable prime quickly when re-hashing
        - can be done in $O(\mathrm{M}\log\log n)$ time
            - no details
- Multiplicative method: $h(k) = \lfloor M(kA - \lfloor kA \rfloor)\rfloor$
    - multiplying with $0 < A < 1$ is used to scramble keys
    - so A should be irrational to avoid patterns in keys
    - experiments show that good scrambling is achieved when A is the golden ratio
    - should use at least $\log|U| + \log M$ bits of $\log|\mathrm{U}|$

Randomized Hash Functiosn: Carter-Wegman's Universal Hashing

- Randomization that uses easy-to-compute hash functions
  - Requires: all keys are in $\{0, \ldots p-1\}$ for some (big) prime p
  - At initialization and whenever rehash
    - choose number $M < p$
    - M equal to some power of 2 is ok
    - choose (and store) two random numbers $a, b \in \{0, \ldots p-1\}$
      - $b = random(p)$
      - $a = 1 + random(p-1)$
        - so that $a \neq 0$
    - Use as hash function

      $$h(k) = ((ak+b) \bmod p) \bmod M$$

      - can be computed quickly
  - can prove that two keys collide with probability at most $\frac{1}{M}$
    - enough to prove the expected runtime bounds for chaining, although uniform hashing assumption is not satisfied

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
   - should depend on all parts of the key
   - 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

$$f(APPLE) = \Big(\big((65R + 80)R + 80\big)R + 76\Big)R + 69$$

     - 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 trade-off via load factor
  - cuckoo hashing achieves $O(1)$ worst-case for search & delete