CS 240 – Data Structures and Data Management

Module 7: Dictionaries via Hashing

Armin Jamshidpey, Éric Schost
Based on lecture notes by many previous cs240 instructors

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

Spring 2026

version *2026-07-07 08:03*

Outline

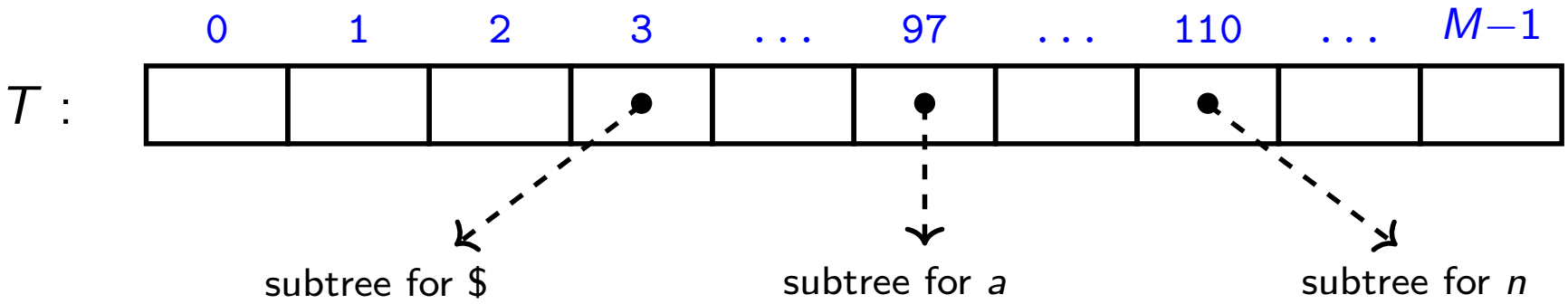
Outline

Direct addressing

Assume: For some $M \in \mathbb{N}$, every key k is an integer with $0 \leq k < M$.

- **Example:** To store child-links in multiway tries, the keys are characters in ASCII $= \{0, \ldots, 127\}$.

We can then implement a dictionary easily: Use an array T of size M that stores (k, v) via $T[k] \leftarrow v$.



Need $\Theta(M)$ space, but each operation takes $\Theta(1)$ time.

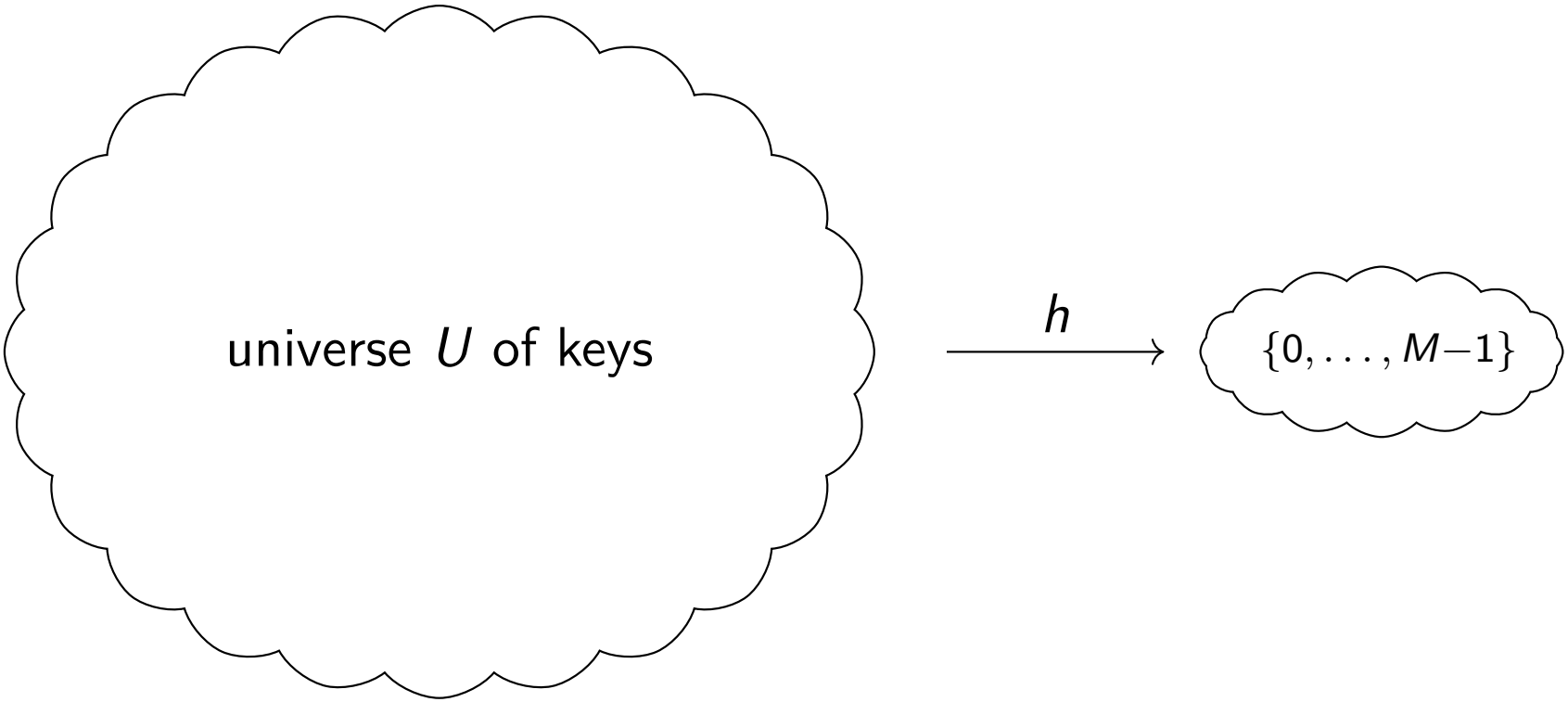
- *search*(k): Check whether $T[k]$ is `NULL`
- *insert*(k, v): $T[k] \leftarrow v$
- *delete*(k): $T[k] \leftarrow$ `NULL`

Hashing: Idea

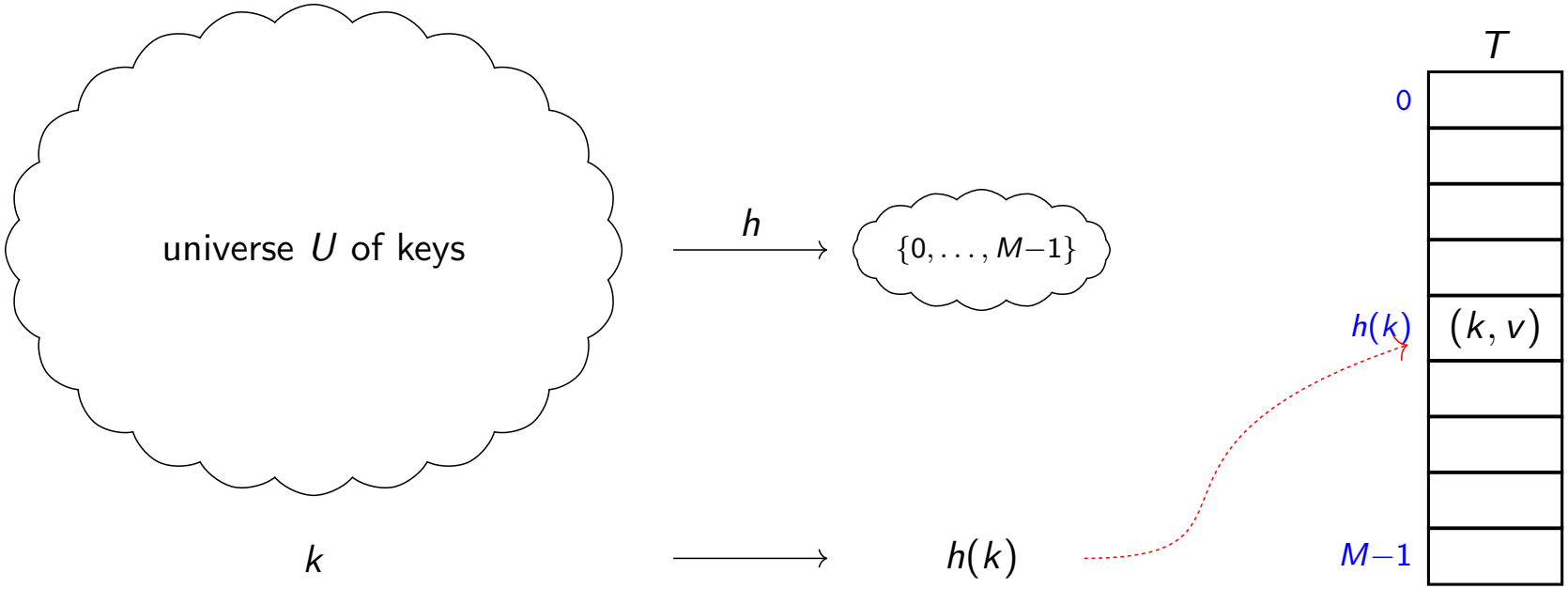
Two disadvantages of direct addressing:

- It cannot be used if the keys are not integers.
- It needs $\Theta(M)$ space: wasteful if M is unknown or $n \ll M$.

Hashing idea: Map (arbitrary) keys to integers in range $\{0, \ldots, M-1\}$ (for an integer M of our choice), then use direct addressing.



Hashing: Idea

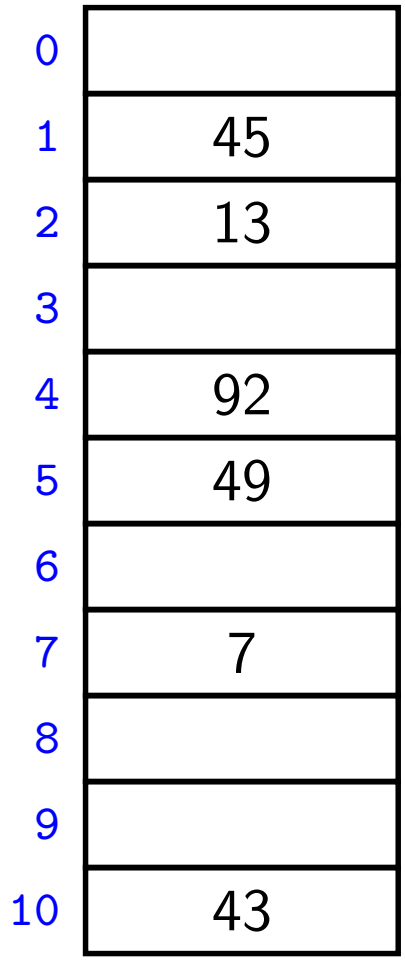


- **Assumption:** We know that all keys come from some **universe** U. (Typically U = non-negative integers, may assume U finite.)
- We pick a **table-size** M.
- We pick a **hash function** $h : U \to \{0, 1, \ldots, M-1\}$. (Commonly used: $h(k) = k \bmod M$. We will see other choices later.)
- Store dictionary in **hash table**, i.e., an array T of size M.
- An item with key k wants to be stored in **slot** $h(k)$, i.e., at $T[h(k)]$.

Hashing: Example

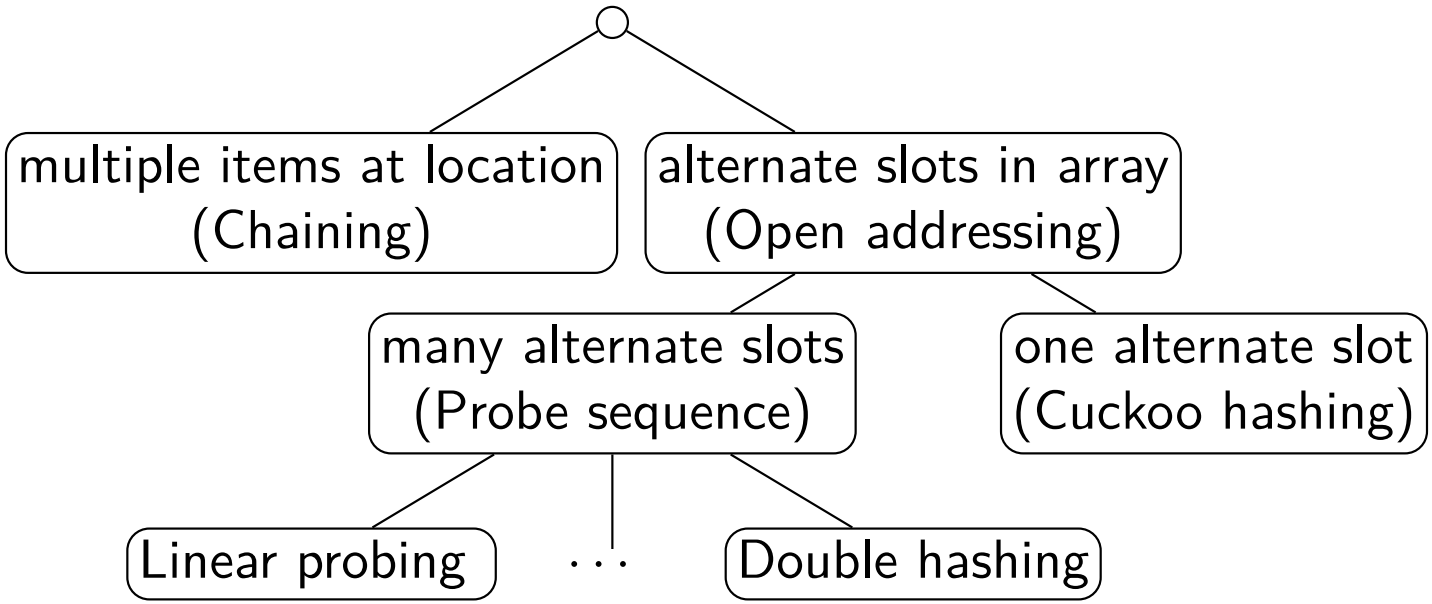
$U = \mathbb{N}$, $M = 11$, $h(k) = k \bmod 11$.

The hash table stores keys 7, 13, 43, 45, 49, 92. (Values are not shown).



Collisions

- Generally hash function h is not injective, so many keys can map to the same integer.
  - For example, $h(46) = 2 = h(13)$ if $h(k) = k \bmod 11$.
- We get **collisions**: we want to insert (k, v) into the table, but $T[h(k)]$ is already occupied.
- There are many strategies to resolve collisions:



Outline

Hashing with chaining

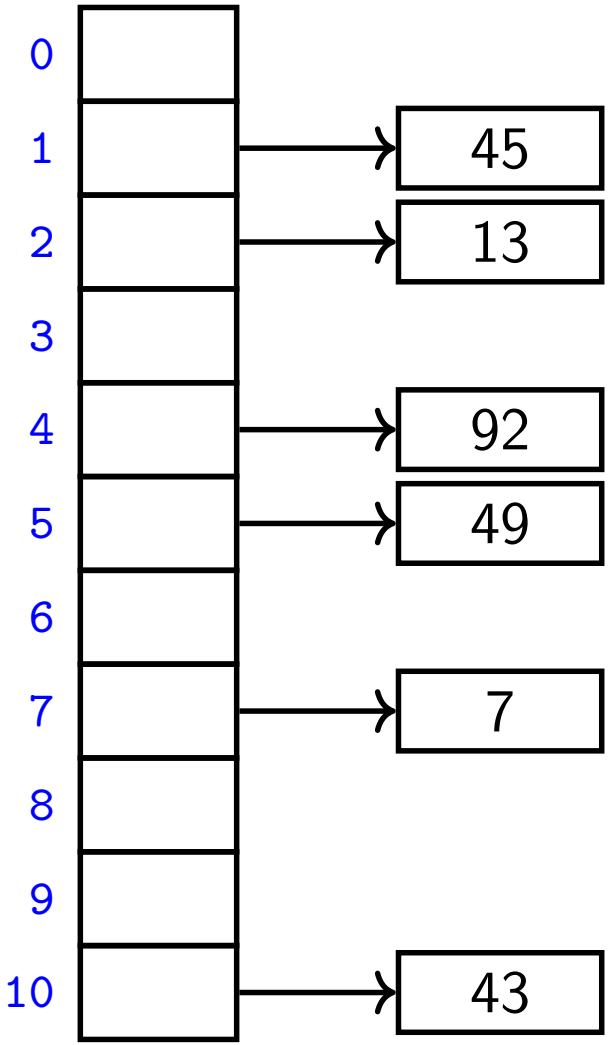
Simplest collision-resolution strategy: Each slot stores a **bucket** containing 0 or more KVPs.

- A bucket could be implemented by any dictionary realization (even another hash table!).
- The simplest approach is to use unsorted lists with MTF for buckets. This is called collision resolution by **chaining**.
- *insert*(k, v): Add (k, v) to the front of the list at $T[h(k)]$.
- *search*(k): Look for key k in the list at $T[h(k)]$.
  Apply MTF-heuristic
- *delete*(k): Perform a search, then delete from the linked list.

insert takes time $O(1)$.
search and *delete* have run-time $O(1 + \text{length of list at } T(h(k)))$.

Hashing with chaining: Example

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

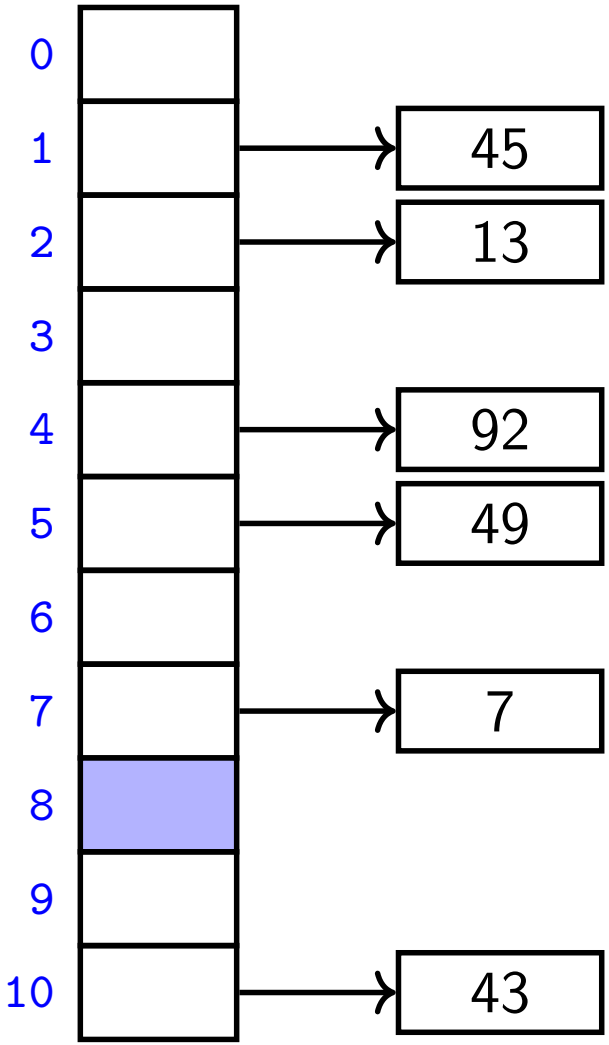


Hashing with chaining: Example

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

insert(41)

$h(41) = 8$

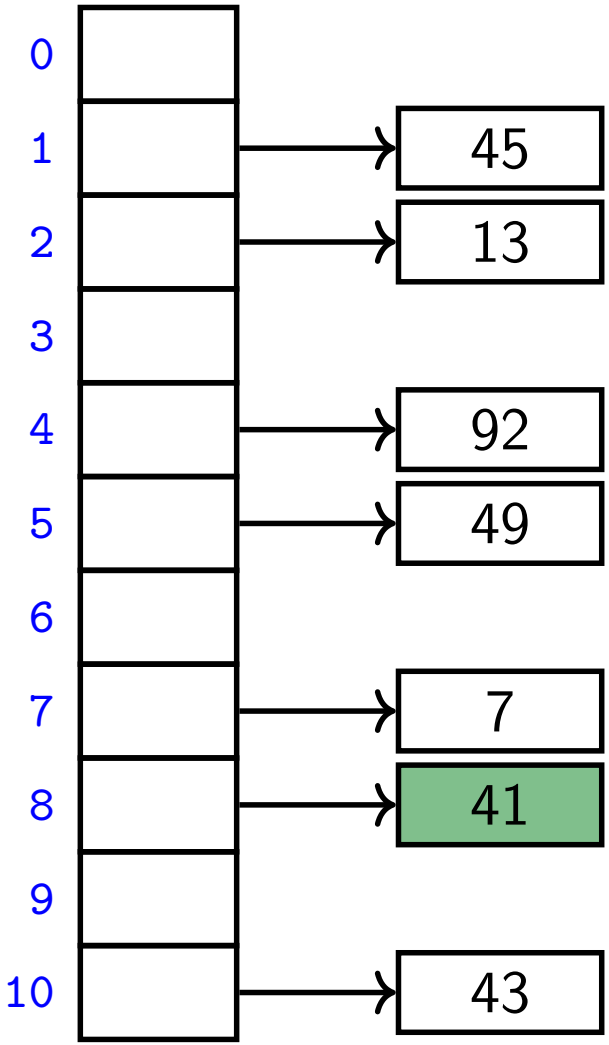


Hashing with chaining: Example

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

insert(41)

$h(41) = 8$

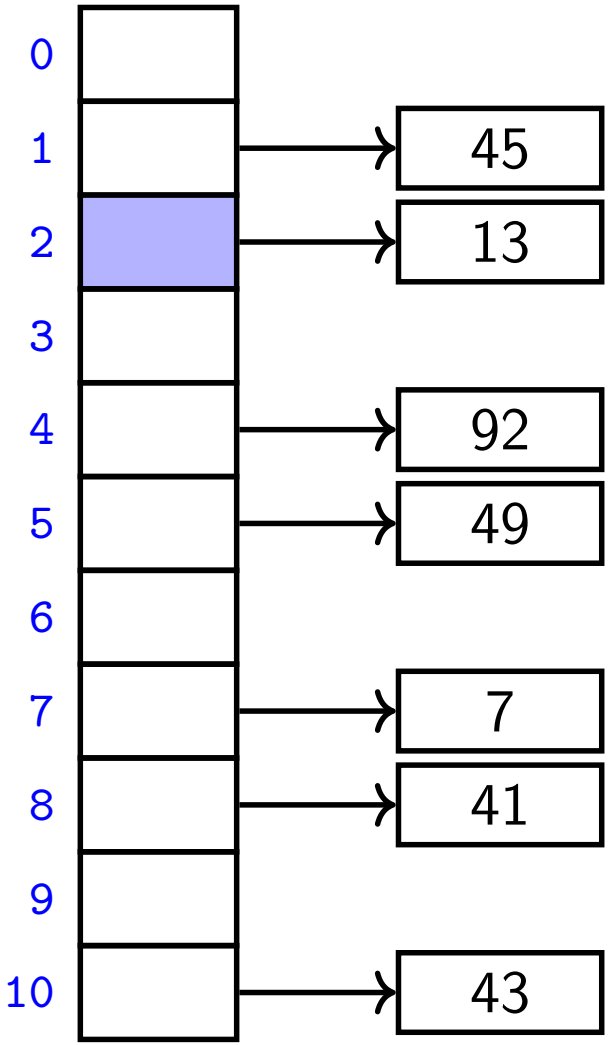


Hashing with chaining: Example

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

insert(46)

$h(46) = 2$

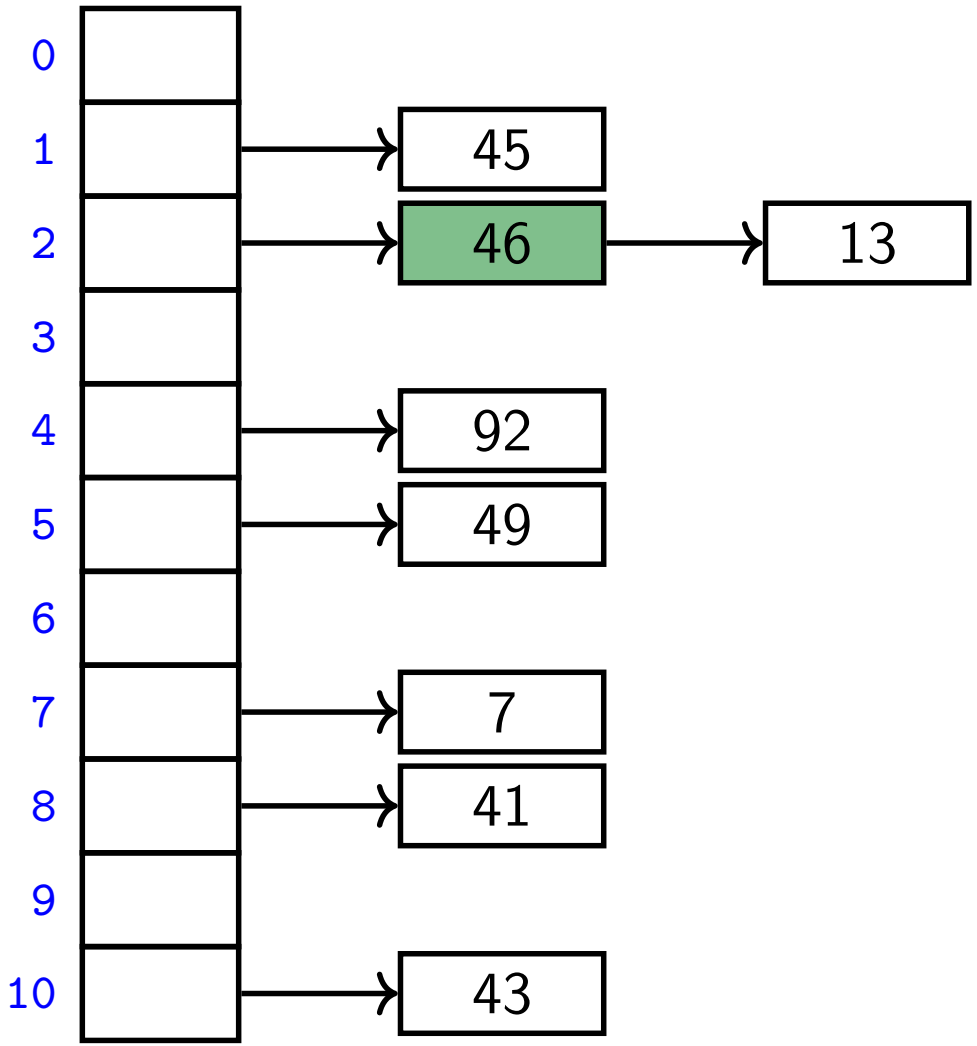


Hashing with chaining: Example

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

insert(46)

$h(46) = 2$

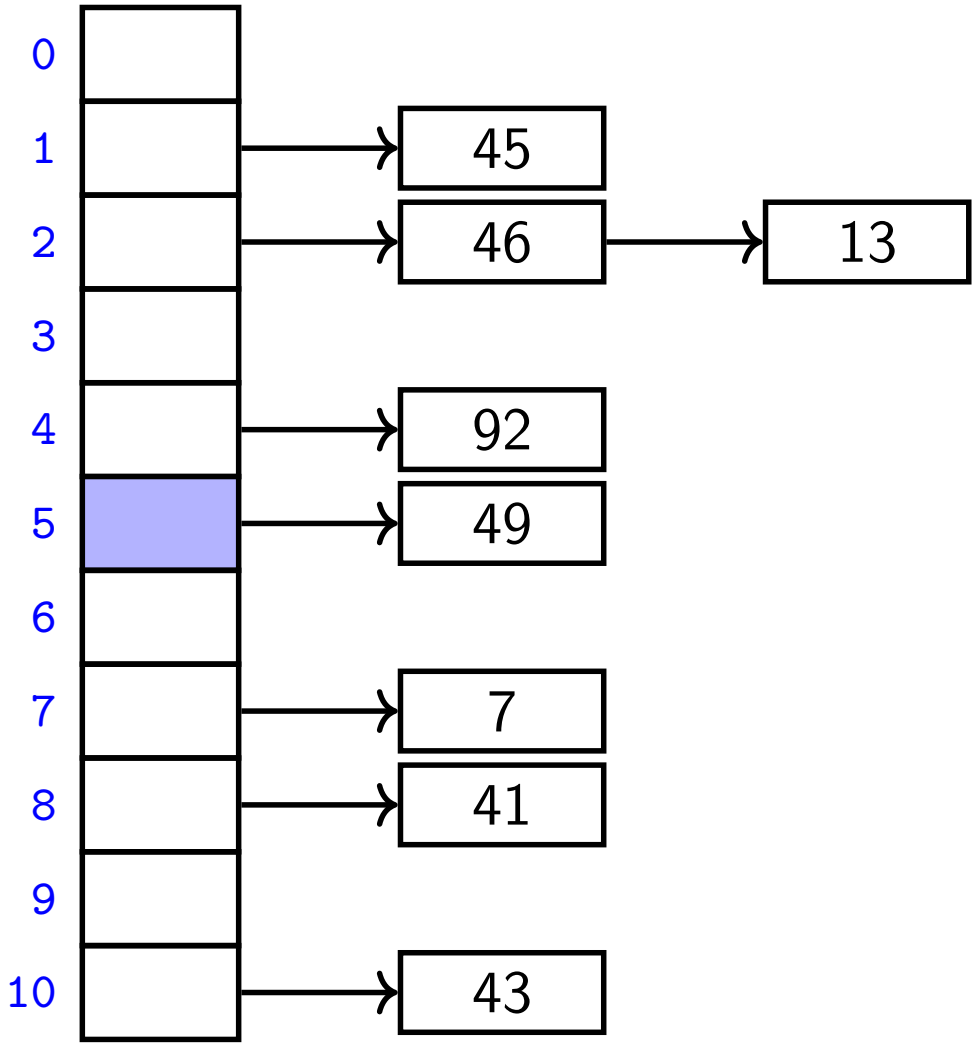


Hashing with chaining: Example

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

insert(16)

$h(16) = 5$

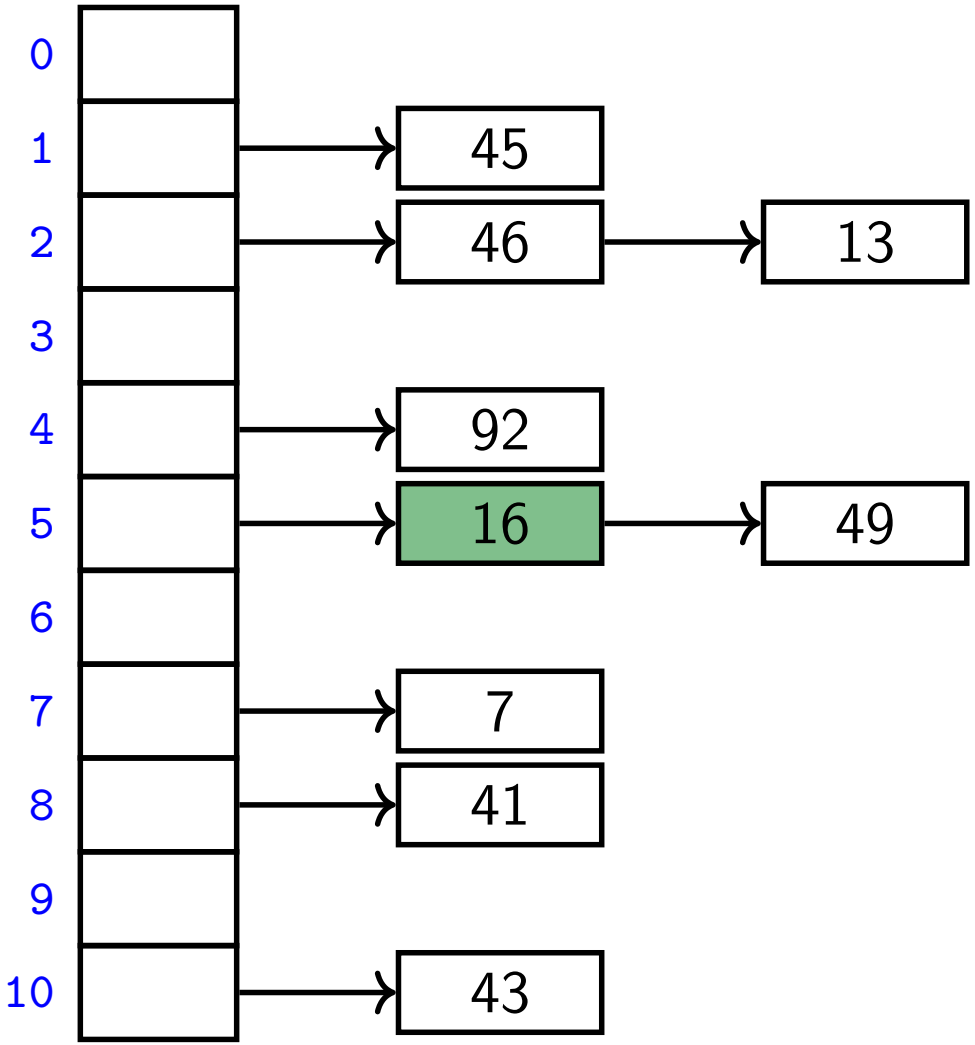


Hashing with chaining: Example

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

insert(16)

$h(16) = 5$

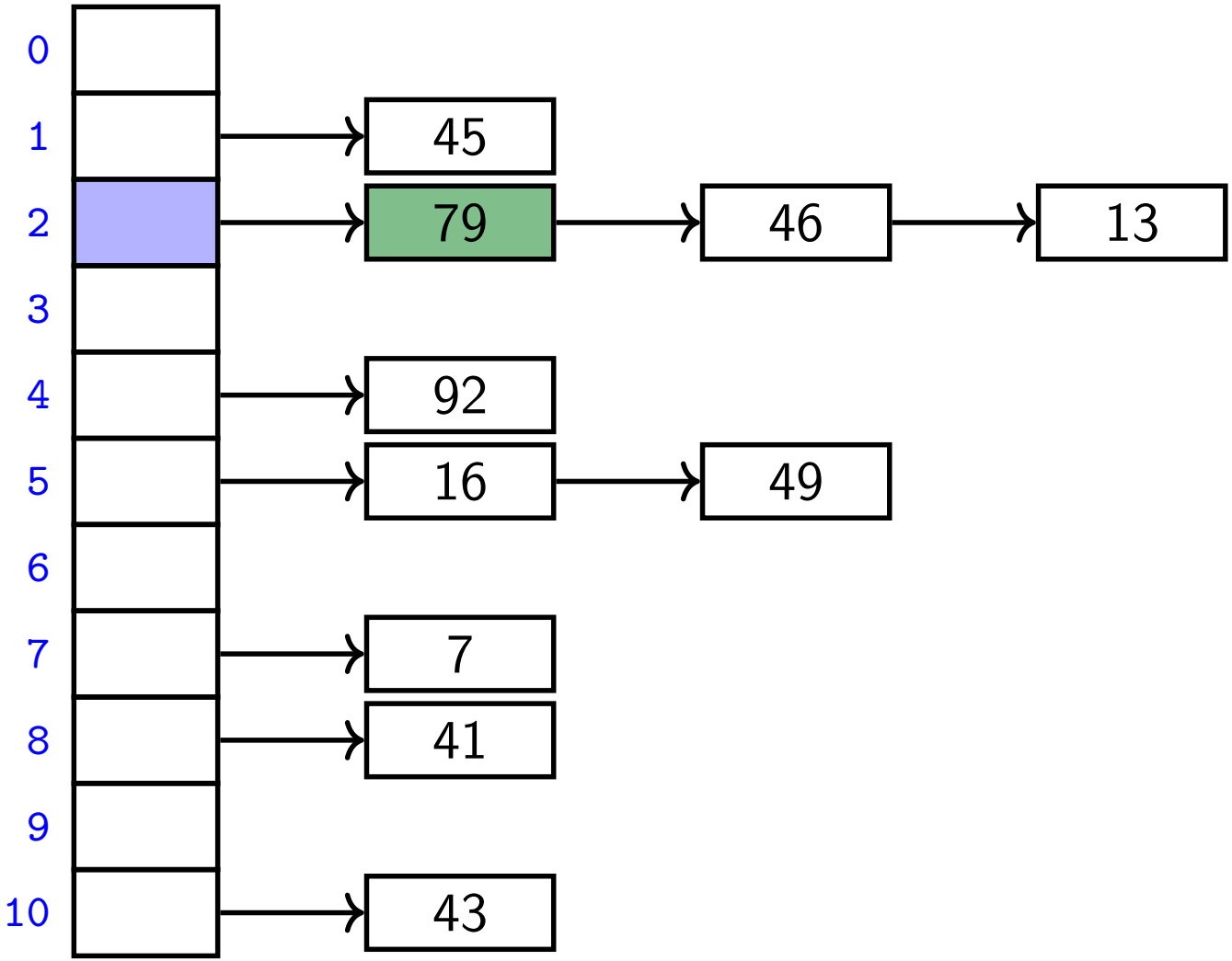


Hashing with chaining: Example

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

insert(79)

$h(79) = 2$



Hashing with chaining: Analysis

Run-times: *insert* takes time $\Theta(1)$.

search and *delete* have run-time $\Theta\Big(1 + \text{size of bucket } T[h(k)]\Big)$.

- The *average bucket-size* is $\alpha = \frac{n}{M}$.
  (α is also called the **load factor**.)
- However, this does not imply that the *average-case* cost of *search* and *delete* is $\Theta(1 + \alpha)$.
  - Consider the case where all keys hash to the same slot
  - The average bucket-size is still α
  - But the operations take $\Theta(n)$ time on average
- To get meaningful average-case bounds, we need some assumptions on the hash-functions or the keys

The uniform hashing assumption (UHA)

- To analyze what happens 'on average', switch to *randomized* hashing.
- How can we randomize?
  Assume that the *hash-function* is chosen randomly.
  - We will later see examples how to do this.
- **Uniform Hashing Assumption**: the hash-function is chosen uniformly at random among all possible functions $U \to \{0, \ldots, M-1\}$
- This is not at all realistic, but the assumption makes analysis possible

Hashing with chaining: Analysis

UHA implies that the distribution of keys is unimportant.

- **Claim:** Hash-values are uniform.
  Formally: $Pr(h(k) = i) = \frac{1}{M}$ for any key k and slot i.

- **Similar:** For $k \neq k'$, $Pr(h(k) = h(k')) = 1/M$

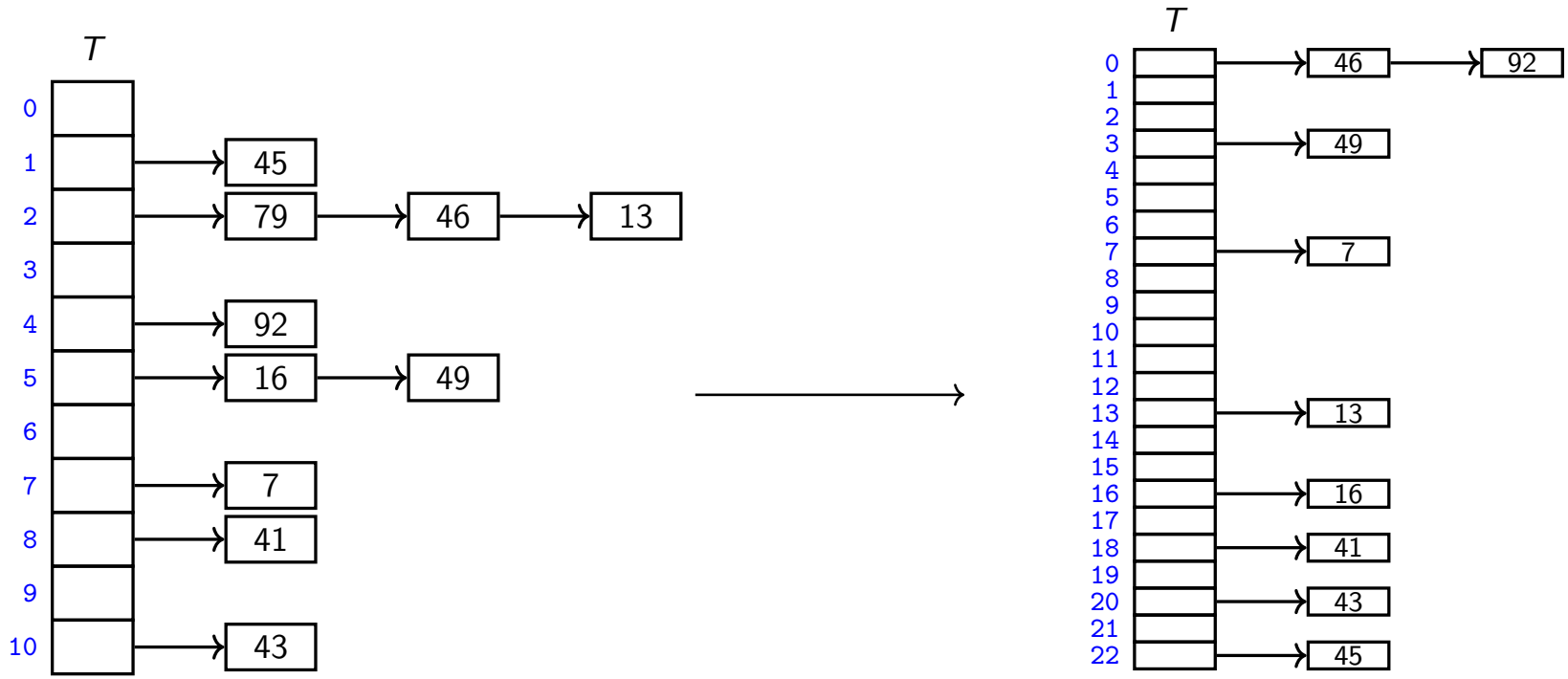
Back to analysis of chaining: search time for k at most equal to

$$1 \; + \; |\{k' \in D | h(k) = h(k')\}| = 1 + \sum_{k' \in D} 1_{h(k)=h(k')}$$

- If key k is *not* in dictionary, $E(\sum_{k'} \dots) = n/M = \alpha$
- If key k *is* in dictionary, $E(\sum_{k'} \dots) = 1 + (n-1)/M \leq 1 + \alpha$
- Expected cost of *search* and *delete* is hence $O(1 + \alpha)$

Load factor and re-hashing

- For hashing with chaining (and also other collision resolution strategies), the run-time bound depends on α
- We keep the load factor small by **rehashing** when needed:



- Keep track of n and M throughout operations
- If α gets too large, create new (roughly twice as big) hash-table, new hash-function(s) and re-insert all items in the new table.

Hashing with chaining: Summary

- For hashing with chaining: Rehash so that $\alpha \in O(1)$ throughout
- 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 (as in module 2)
- If space is critical: also re-hash when α gets too small, so that $\alpha \in \Theta(1)$ throughout, and the space is always $\Theta(n)$.

Summary: The (amortized) expected cost for hashing with chaining is $\Theta(1)$ and the space is $\Theta(n)$

(assuming uniform hashing and $\alpha \in \Theta(1)$ throughout)

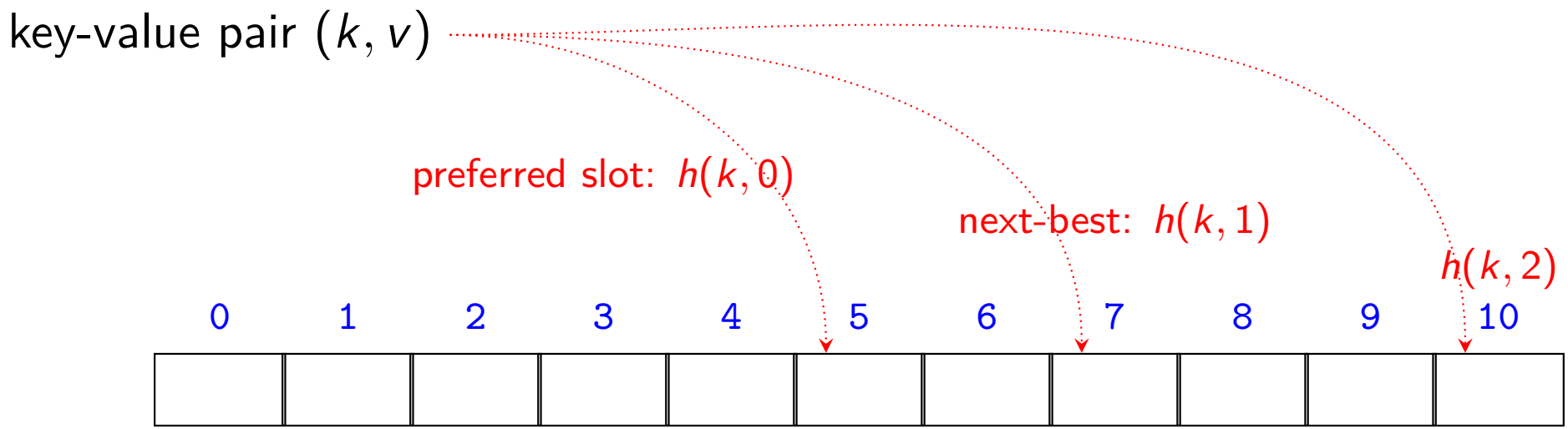
Theoretically perfect, but somewhat slow in practice (because of linked lists)

Outline

Open addressing

Main idea: Avoid the links needed for chaining by permitting only one item per slot, but allowing a key k to be in one of multiple slots.

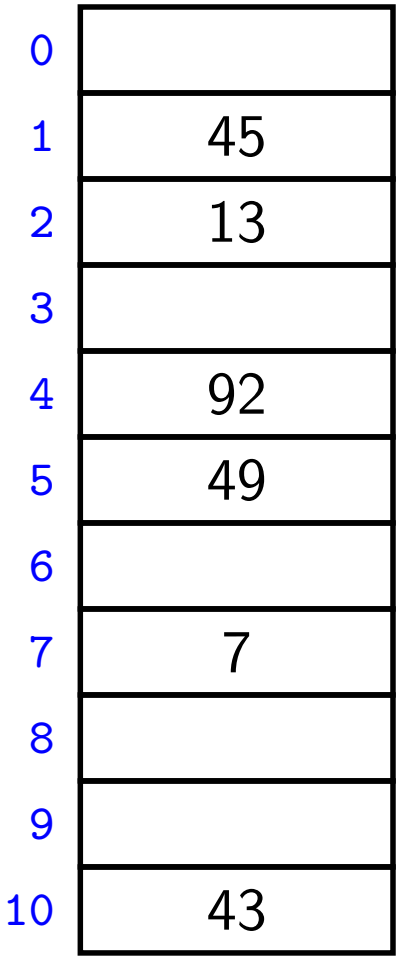
search and *insert* follow a **probe sequence** of possible locations for key k: $\langle h(k,0), h(k,1), h(k,2), \ldots h(k,M-1)\rangle$ until an empty spot is found.



Simplest method for open addressing: *linear probing*
$h(k,j) = (h(k) + j) \bmod M$, for some hash function h.

Hashing with linear probing: Example

$M = 11, \qquad h(k) = k \bmod 11, \qquad h(k, j) = (h(k) + j) \bmod 11.$



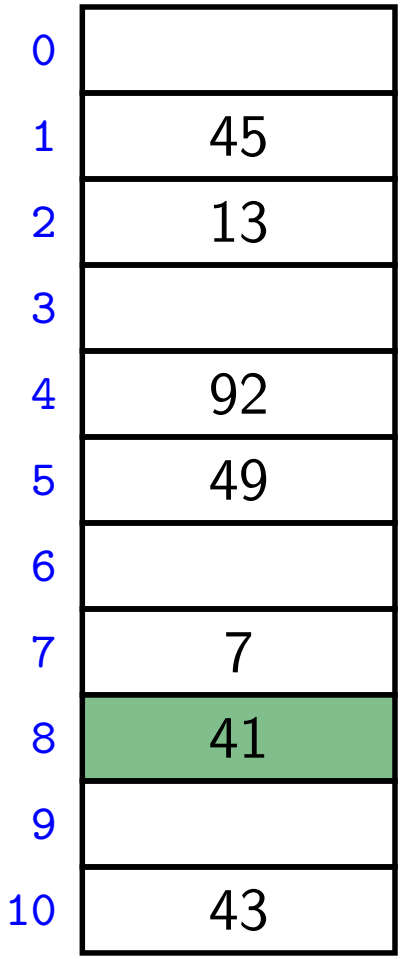
Note: we are creating clusters of filled slots.

Hashing with linear probing: Example

$M = 11, \qquad h(k) = k \bmod 11, \qquad h(k, j) = (h(k) + j) \bmod 11.$

insert(41)

$h(41, 0) = 8$



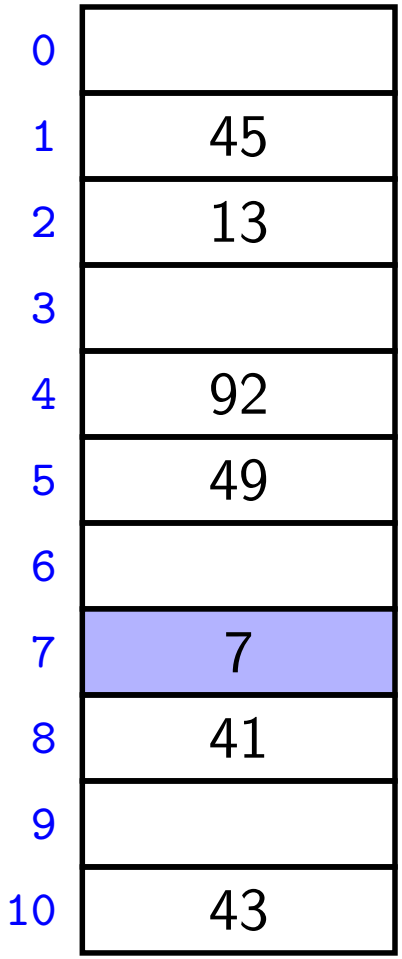
Note: we are creating clusters of filled slots.

Hashing with linear probing: Example

$M = 11, \qquad h(k) = k \bmod 11, \qquad h(k, j) = (h(k) + j) \bmod 11.$

insert(84)

$h(84, 0) = 7$



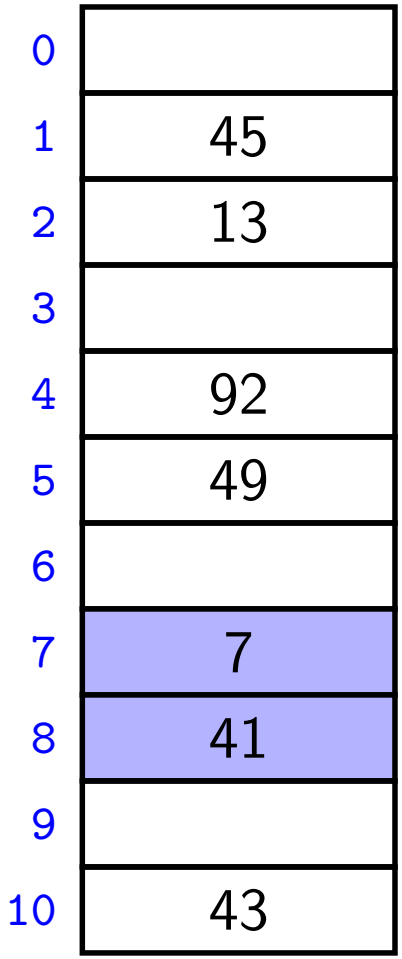
Note: we are creating clusters of filled slots.

Hashing with linear probing: Example

$M = 11, \qquad h(k) = k \bmod 11, \qquad h(k, j) = (h(k) + j) \bmod 11.$

insert(84)

$h(84, 1) = 8$



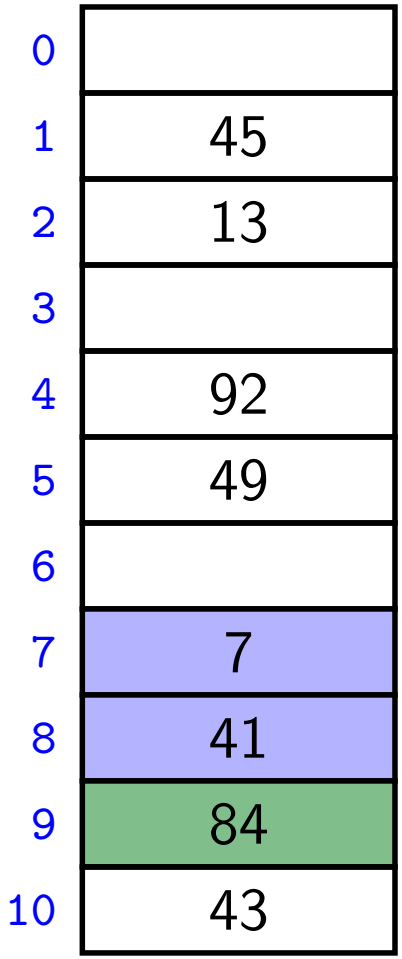
Note: we are creating clusters of filled slots.

Hashing with linear probing: Example

$M = 11, \qquad h(k) = k \bmod 11, \qquad h(k, j) = (h(k) + j) \bmod 11.$

insert(84)

$h(84, 2) = 9$



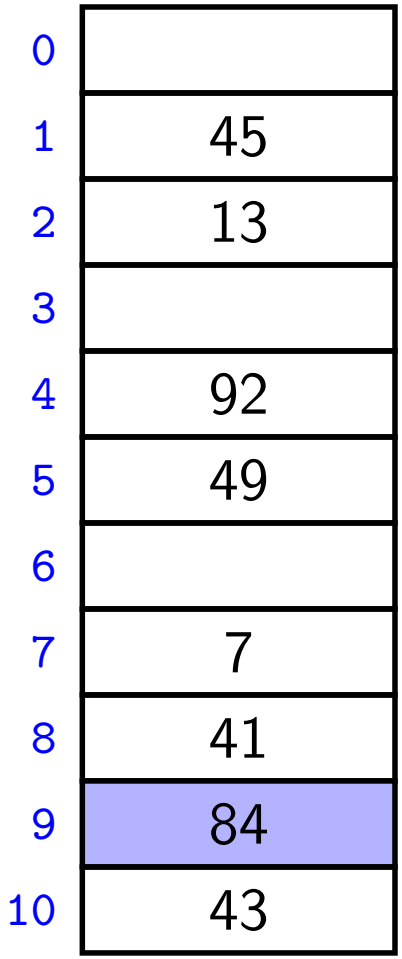
Note: we are creating clusters of filled slots.

Hashing with linear probing: Example

$M = 11, \qquad h(k) = k \bmod 11, \qquad h(k, j) = (h(k) + j) \bmod 11.$

insert(20)

$h(20, 0) = 9$



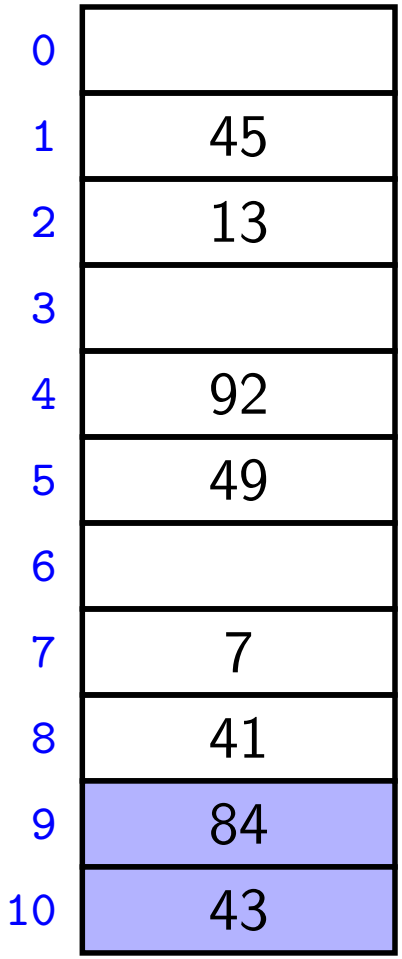
Note: we are creating clusters of filled slots.

Hashing with linear probing: Example

$M = 11, \qquad h(k) = k \bmod 11, \qquad h(k, j) = (h(k) + j) \bmod 11.$

insert(20)

$h(20, 1) = 10$



Note: we are creating clusters of filled slots.

Hashing with linear probing: Example

$M = 11, \qquad h(k) = k \bmod 11, \qquad h(k, j) = (h(k) + j) \bmod 11.$

insert(20)

$h(20, 2) = 0$

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

Note: we are creating clusters of filled slots.

Hashing with probe sequences: Operations

delete becomes problematic:

- Cannot leave an empty spot behind; the next search might otherwise not go far enough.
- We could try to move later items in probe sequence forward. (But it is non-trivial to find one that can be moved.)
- Better idea: **lazy deletion**:
  - Mark spot as *deleted* (rather than `NULL`)
  - Search continues past deleted spots.
  - Insertion reuses deleted spots.

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

Hashing with linear probing: Example

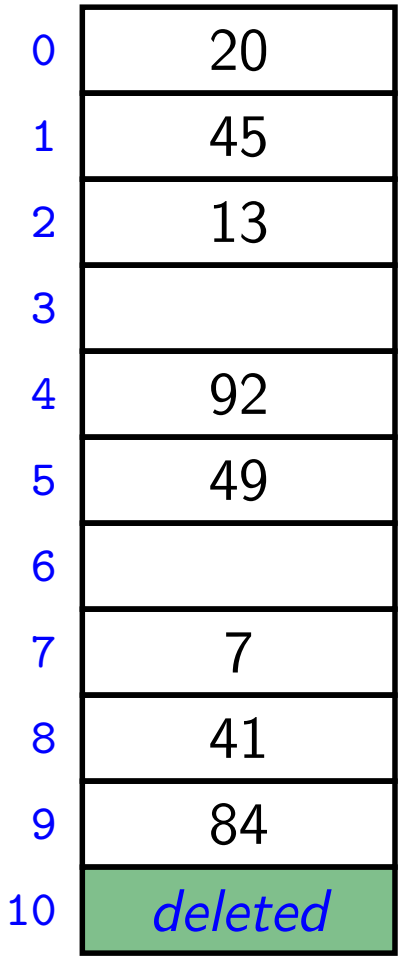
$M = 11, \qquad h(k) = k \bmod 11, \qquad h(k, j) = (h(k) + j) \bmod 11.$

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

Hashing with linear probing: Example

$M = 11, \qquad h(k) = k \bmod 11, \qquad h(k, j) = (h(k) + j) \bmod 11.$

delete(43)
$h(43, 0) = 10$



Hashing with linear probing: Example

$M = 11, \qquad h(k) = k \bmod 11, \qquad h(k, j) = (h(k) + j) \bmod 11.$

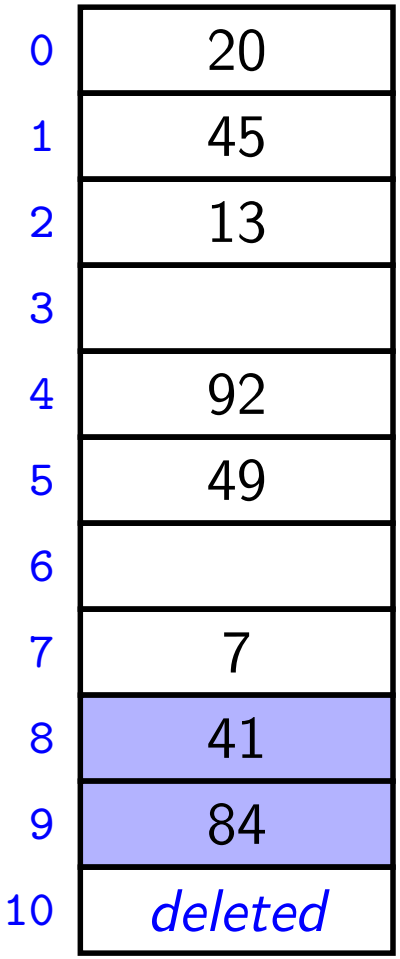
search(63)
$h(63, 0) = 8$

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

Hashing with linear probing: Example

$M = 11, \qquad h(k) = k \bmod 11, \qquad h(k, j) = (h(k) + j) \bmod 11.$

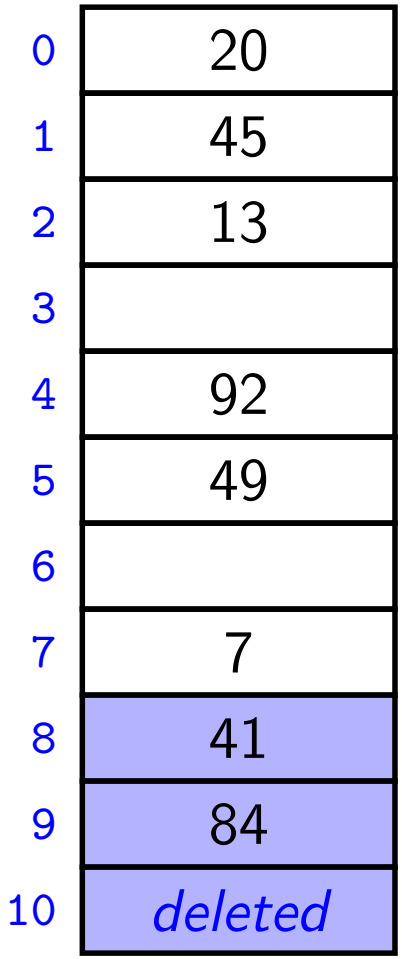
search(63)
$h(63, 1) = 9$



Hashing with linear probing: Example

$M = 11, \qquad h(k) = k \bmod 11, \qquad h(k, j) = (h(k) + j) \bmod 11.$

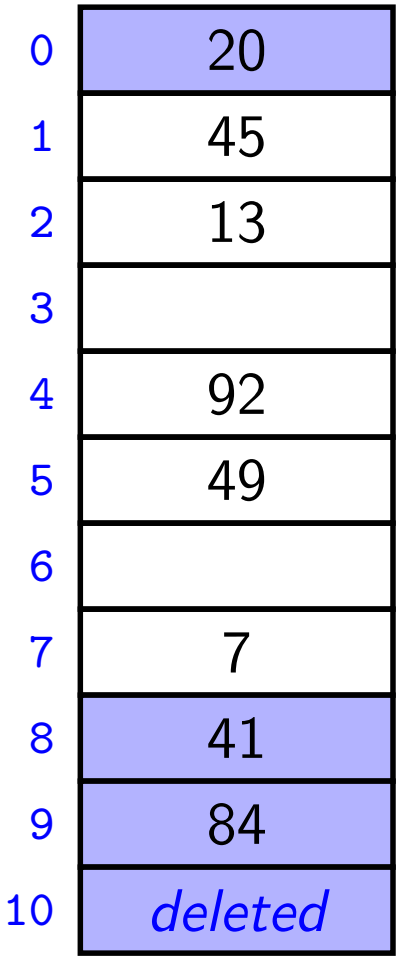
search(63)
$h(63, 2) = 10$



Hashing with linear probing: Example

$M = 11, \qquad h(k) = k \bmod 11, \qquad h(k, j) = (h(k) + j) \bmod 11.$

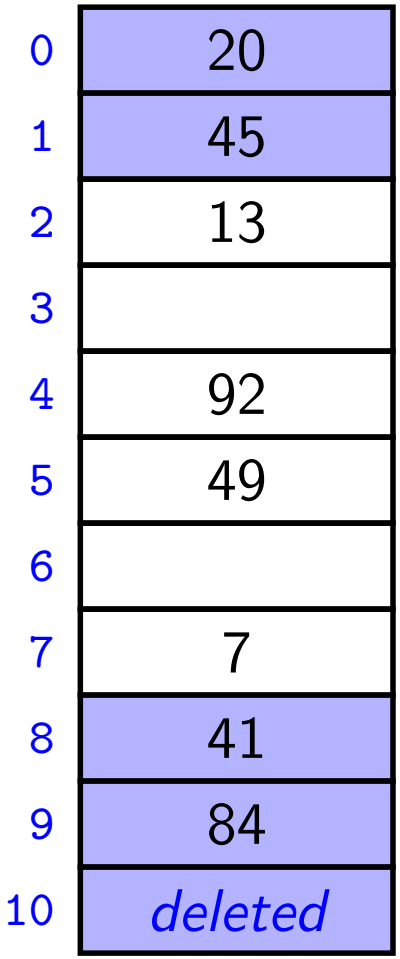
search(63)
$h(63, 3) = 0$



Hashing with linear probing: Example

$M = 11, \qquad h(k) = k \bmod 11, \qquad h(k, j) = (h(k) + j) \bmod 11.$

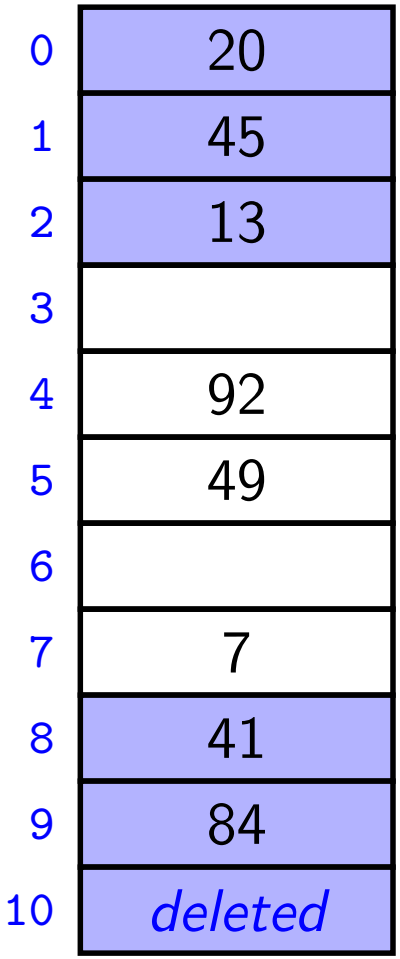
search(63)
$h(63, 4) = 1$



Hashing with linear probing: Example

$M = 11, \qquad h(k) = k \bmod 11, \qquad h(k, j) = (h(k) + j) \bmod 11.$

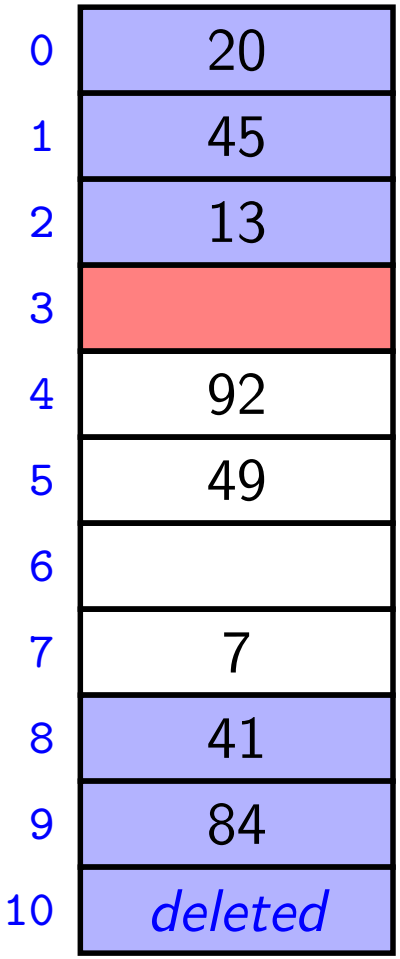
search(63)
$h(63, 5) = 2$



Hashing with linear probing: Example

$M = 11, \qquad h(k) = k \bmod 11, \qquad h(k, j) = (h(k) + j) \bmod 11.$

search(63)
$h(63, 6) = 3$
not found



Hashing with probe sequences: Operations

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

1. **for** $(j = 0; j < M; j$++$)$
2. **if** $T[h(k, j)]$ is `NULL` or "deleted"
3. $T[h(k, j)] = (k, v)$
4. **return** "success"
5. **return** "failure to insert" `// need to re-hash`

probe-sequence-search(T, k)

1. **for** $(j = 0; j < M; j$++$)$
2. **if** $T[h(k, j)]$ is `NULL` **return** "item not found"
3. **if** $T[h(k, j)]$ has key k **return** $T[h(k, j)]$
4. `// key is incorrect or "deleted"`
5. `// try next probe, i.e., continue for-loop`
6. **return** "item not found"

Independent hash functions

- Some hashing methods require *two* hash functions h_0, h_1.
- These hash functions should be *independent* in the sense that the random variables $Pr(h_0(k) = i)$ and $Pr(h_1(k) = j)$ are independent.
- In practice, using two modular hash functions often leads to dependencies.
- Better idea: Use *multiplication method* for second hash function:
  - Fix some floating-point number A with $0 < A < 1$

$$h(k) = \Big\lfloor \underbrace{\underbrace{M \cdot \big(\underbrace{A \cdot k}_{\text{multiply}} - \underbrace{\lfloor A \cdot k \rfloor}_{\text{integral part}} \big)}_{\text{fractional part, in } [0,1)} \Big\rfloor}_{\text{integer in } [0, M)}.$$

  - Our examples use $\varphi = \frac{\sqrt{5}-1}{2} \approx 0.618033988749...$ as A.

Double hashing

Double hashing idea: open addressing with probe sequence

$$h(k, j) = \Big(h_0(k) + j \cdot h_1(k)\Big) \bmod M$$

where we have two hash independent functions h_0, h_1.

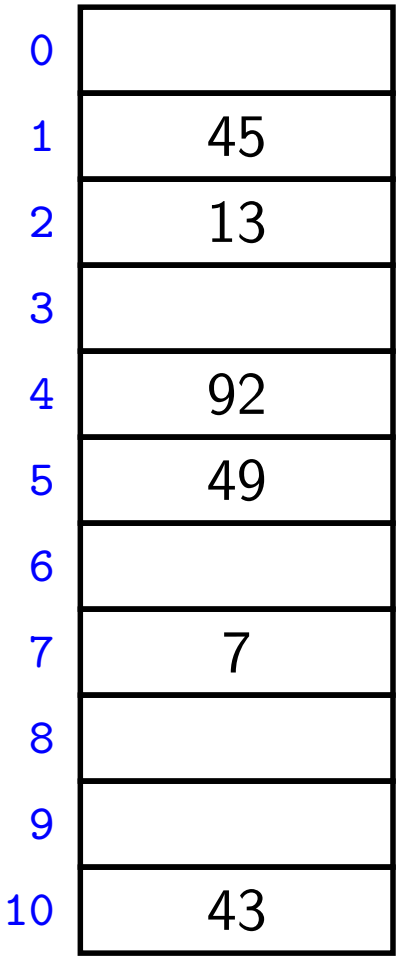
- *search*, *insert*, *delete* work just like for linear probing, but with this different probe sequence.

We require (for all keys k):

- $h_1(k) \neq 0$
- $h_1(k)$ is relative prime with the table-size M, so choose M prime.
- Modify standard hash-functions to ensure $h_1(k) \neq 0$
  E.g. modified multiplication method: $h(k) = 1 + \lfloor (M-1)(kA - \lfloor kA \rfloor) \rfloor$

Double hashing: Example

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



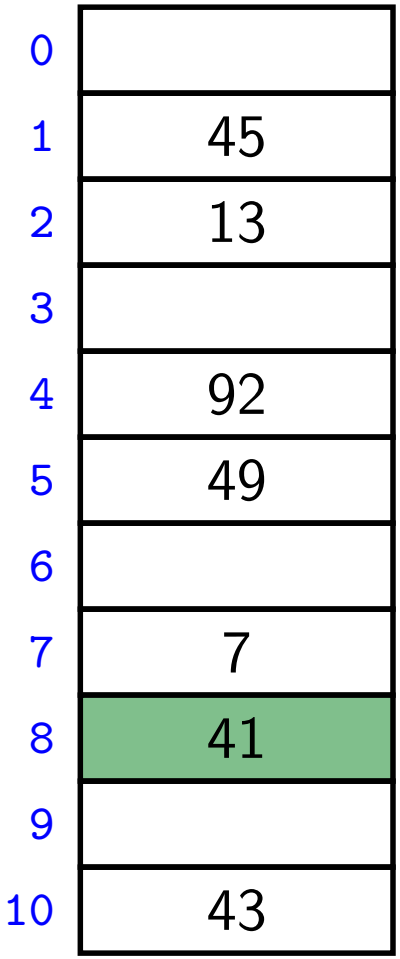
Double hashing: Example

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

insert(41)

$h_0(41) = 8$

$h(41, 0) = 8$



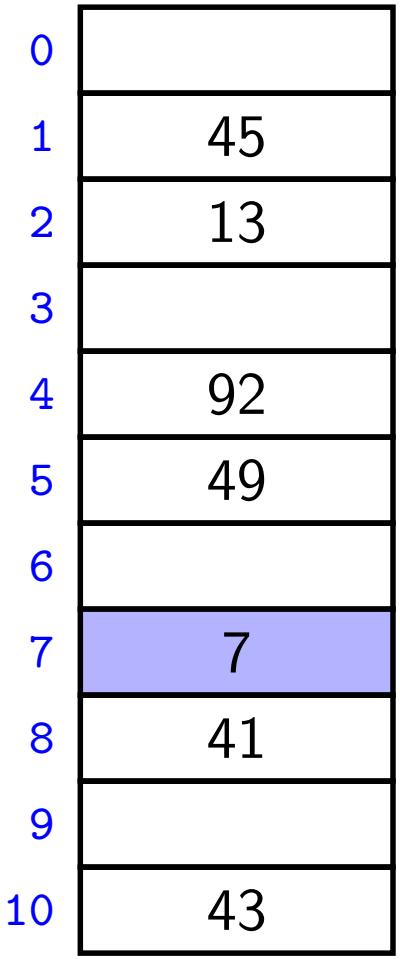
Double hashing: Example

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

insert(194)

$h_0(194) = 7$

$h(194, 0) = 7$



Double hashing: Example

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

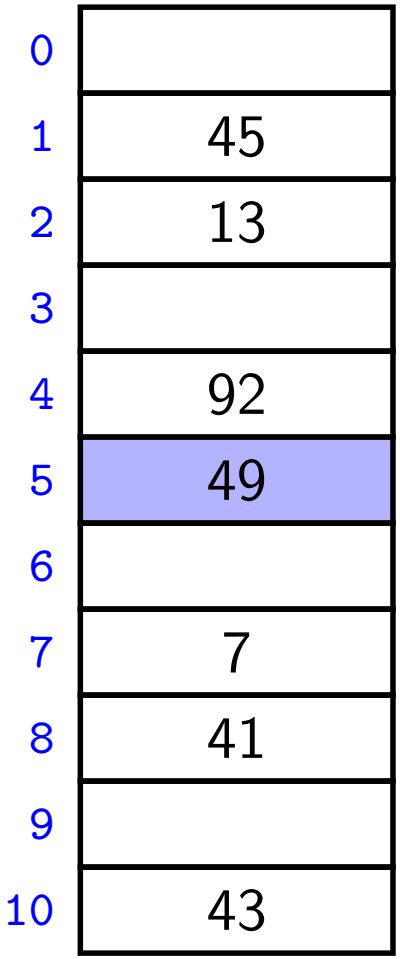
insert(194)

$h_0(194) = 7$

$h(194, 0) = 7$

$h_1(194) = 9$

$h(194, 1) = 5$



Double hashing: Example

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

insert(194)

$h_0(194) = 7$

$h(194, 0) = 7$

$h_1(194) = 9$

$h(194, 1) = 5$

$h(194, 2) = 3$

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

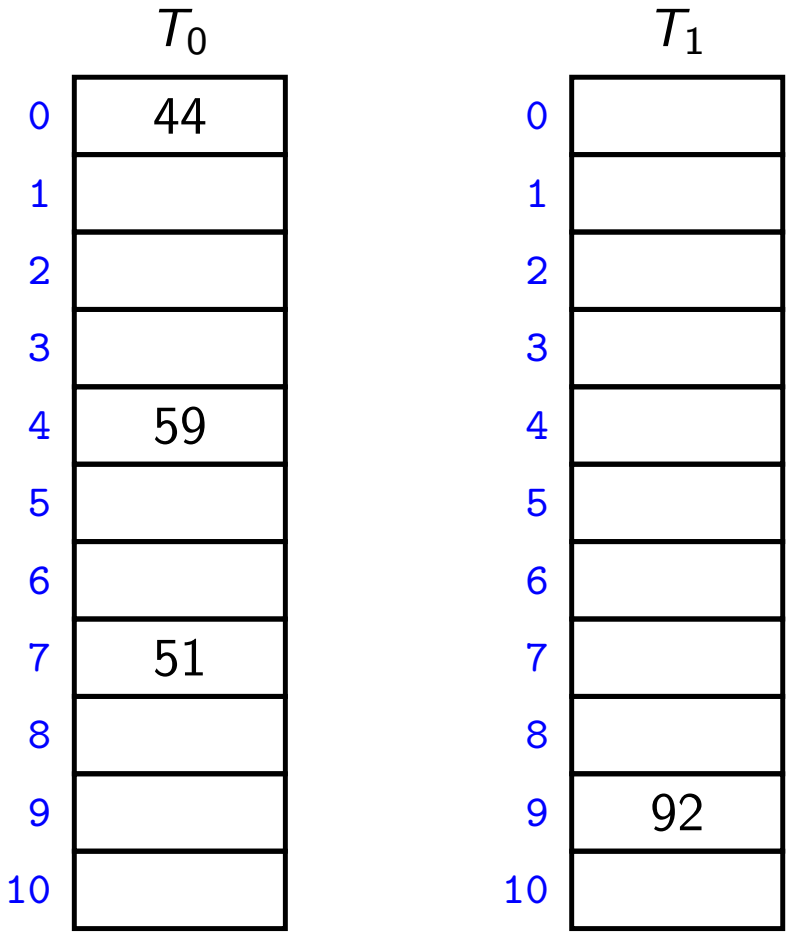
Outline

Cuckoo hashing

We use two independent hash functions h_0, h_1 and two tables T_0, T_1.

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

search and *delete* then *always* take constant time.



Cuckoo hashing: Insertion

insert always initially puts the new item into $T_0[h_0(k)]$

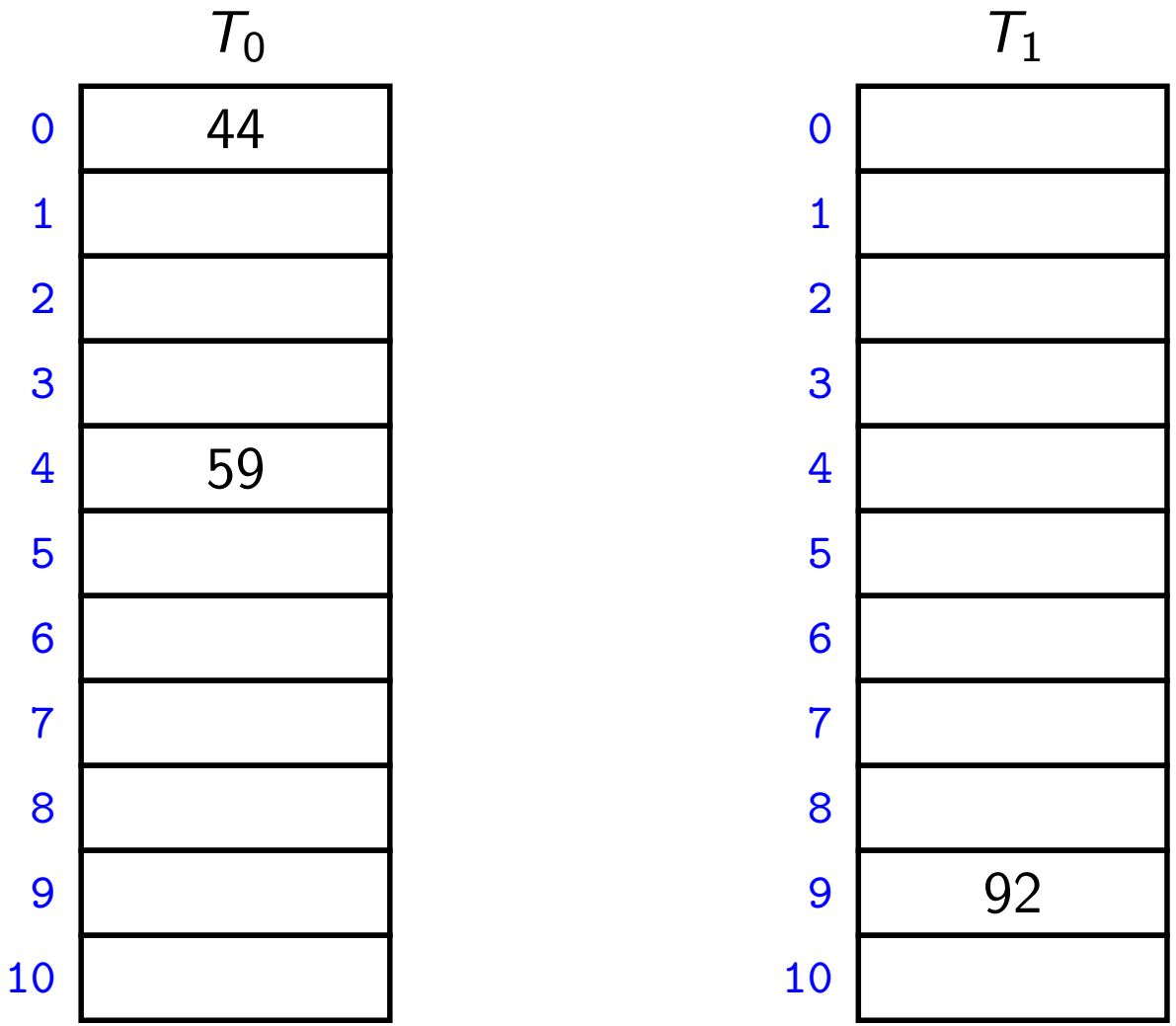
- Evict item that may have been there already.
- If so, evicted item inserted at alternate position
- This may lead to a loop of evictions.
- **Can show:** If insertion is possible, then there are at most $2n$ evictions, so abort after too many attempts.

cuckoo::insert(k, v)

1. $(k_{insert}, v_{insert}) \leftarrow$ new key-value pair with (k, v)
2. $i \leftarrow 0$
3. **do** at most $2n$ times:
4. $\quad (k_{evict}, v_{evict}) \leftarrow T_i[h_i(k_{insert})]$ `// save old KVP`
5. $\quad T_i[h_i(k_{insert})] \leftarrow (k_{insert}, v_{insert})$ `// put in new KVP`
6. $\quad$ **if** (k_{evict}, v_{evict}) is `NULL` **return** "success"
7. $\quad$ **else** `// repeat in other table`
8. $\quad\quad (k_{insert}, v_{insert}) \leftarrow (k_{evict}, v_{evict})$; $i \leftarrow 1 - i$
9. **return** "failure to insert" `// need to re-hash`

Cuckoo hashing insertion: Example

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



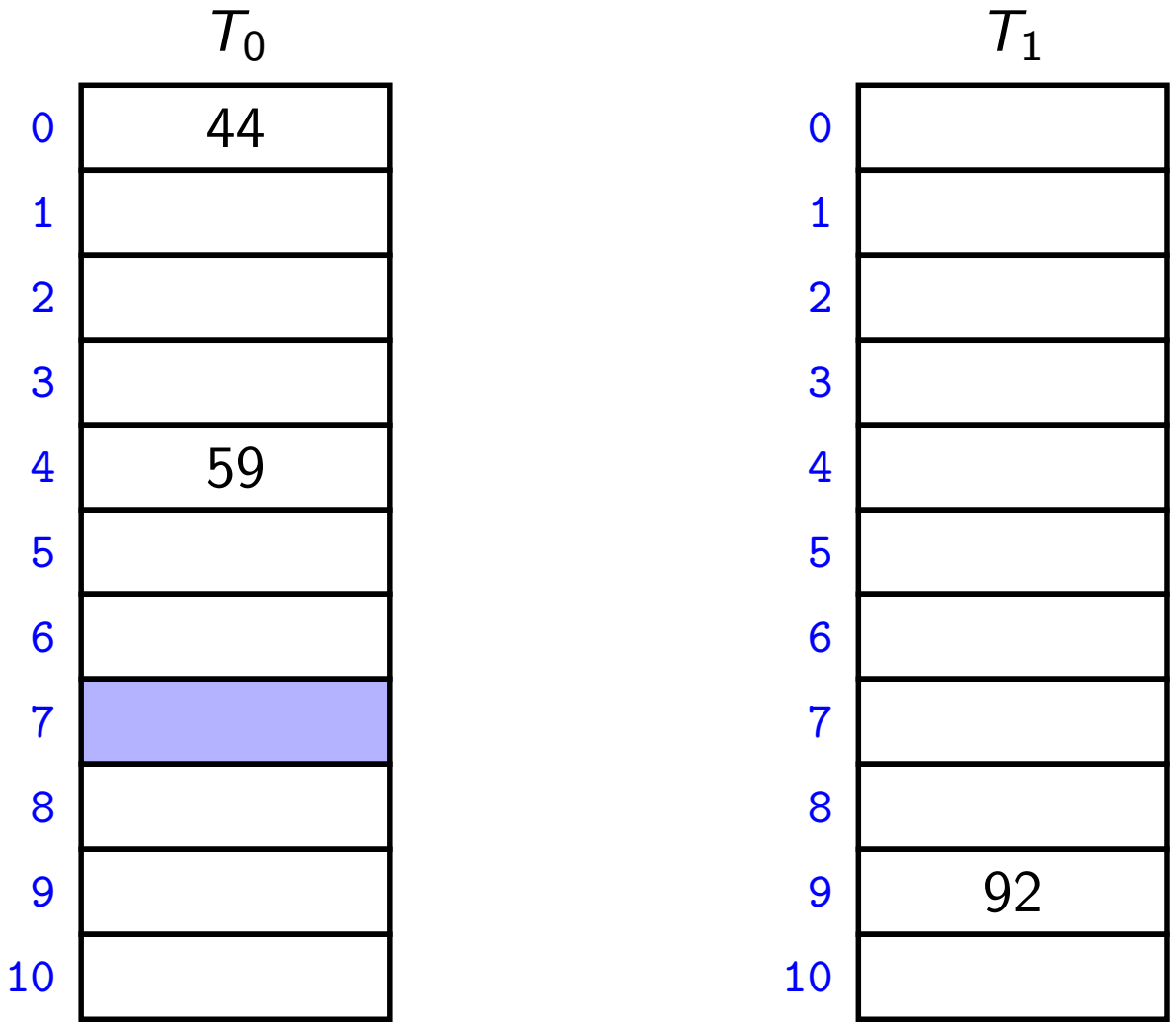
Cuckoo hashing insertion: Example

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

insert(51)

$i = 0$
$k = 51$

$h_0(k) = 7$
$h_1(k) = 5$



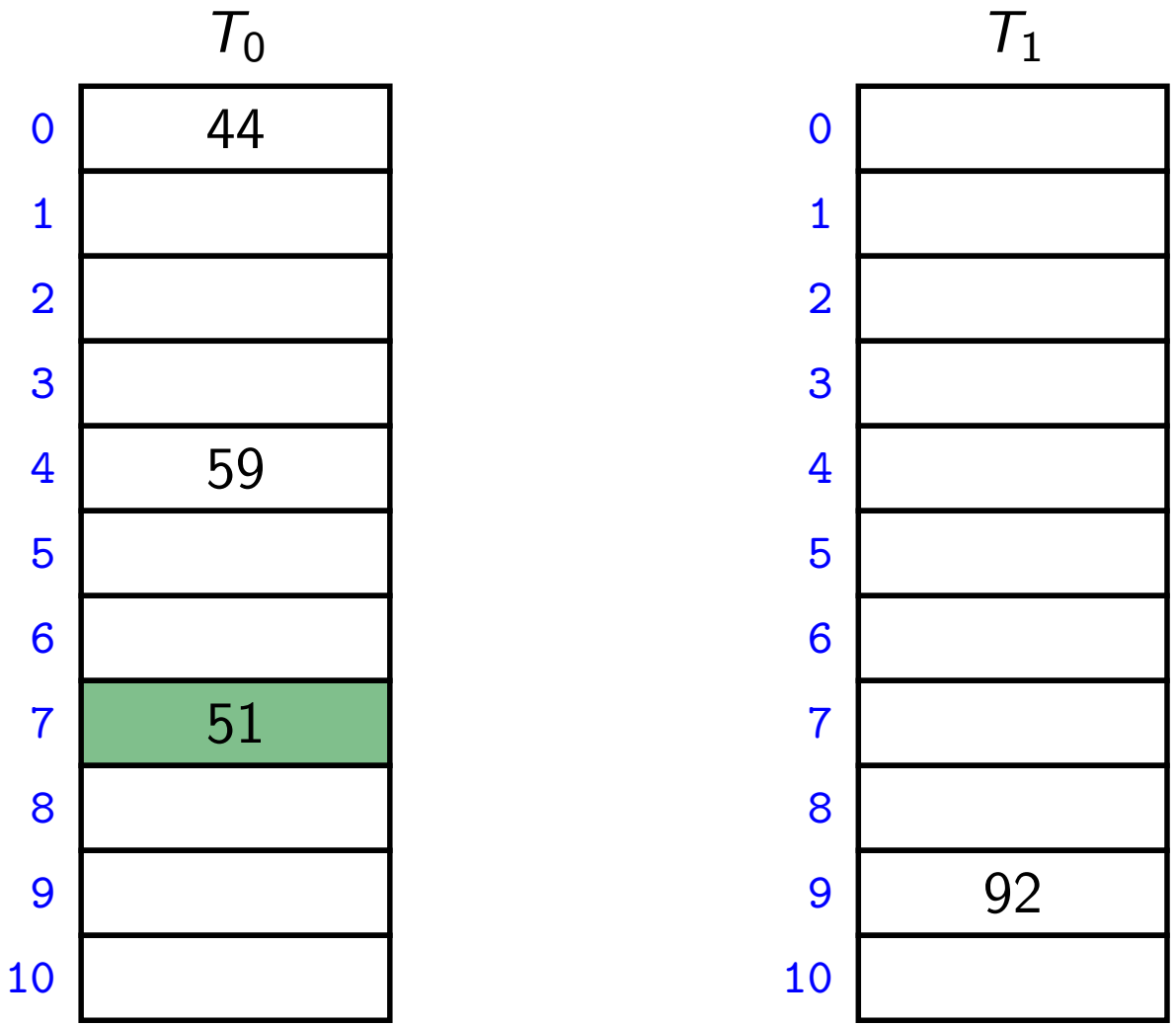
Cuckoo hashing insertion: Example

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

insert(51)

$i = 0$
$k = 51$

$h_0(k) = 7$
$h_1(k) = 5$



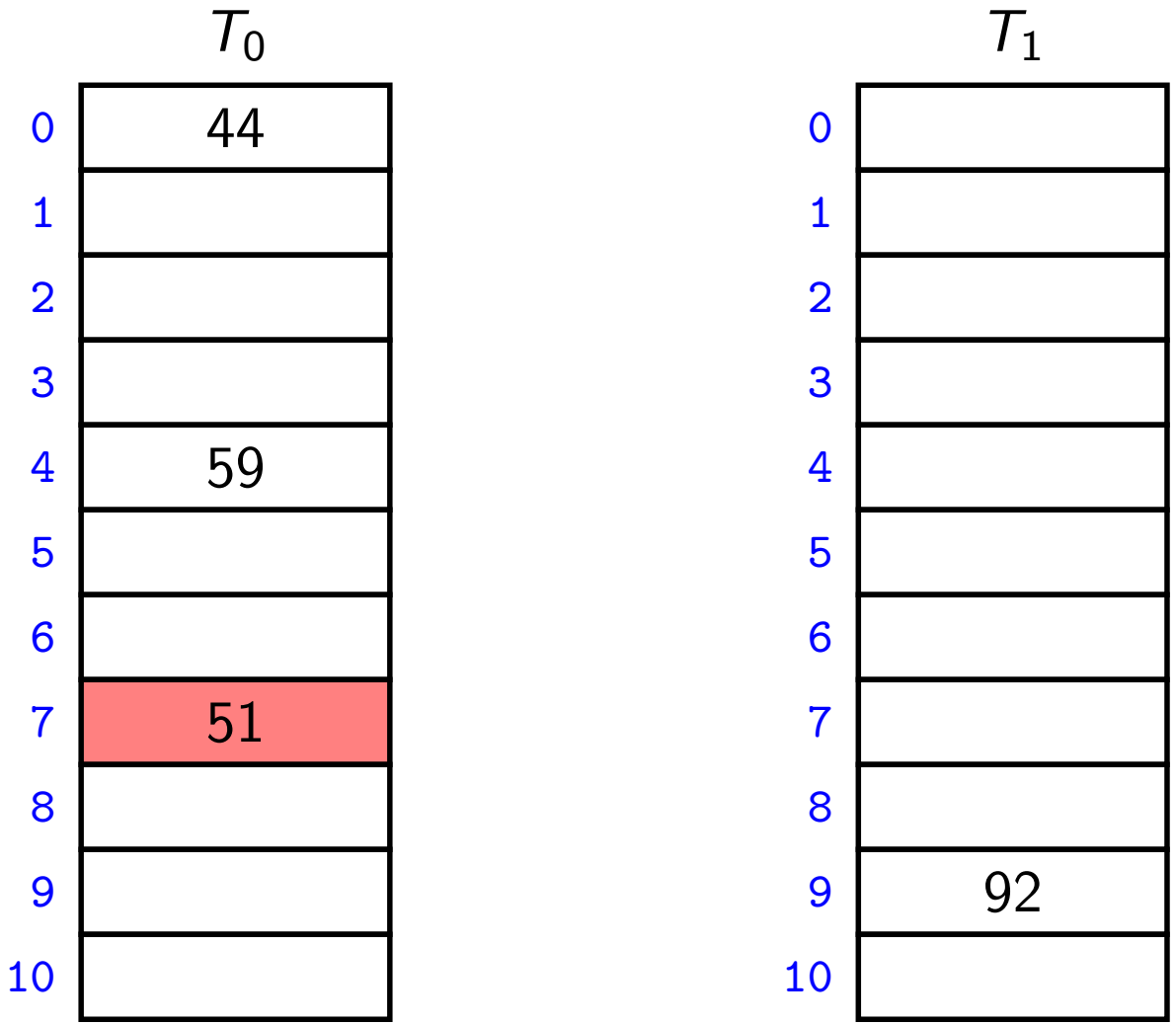
Cuckoo hashing insertion: Example

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

insert(95)

$i = 0$
$k = 95$

$h_0(k) = 7$
$h_1(k) = 7$



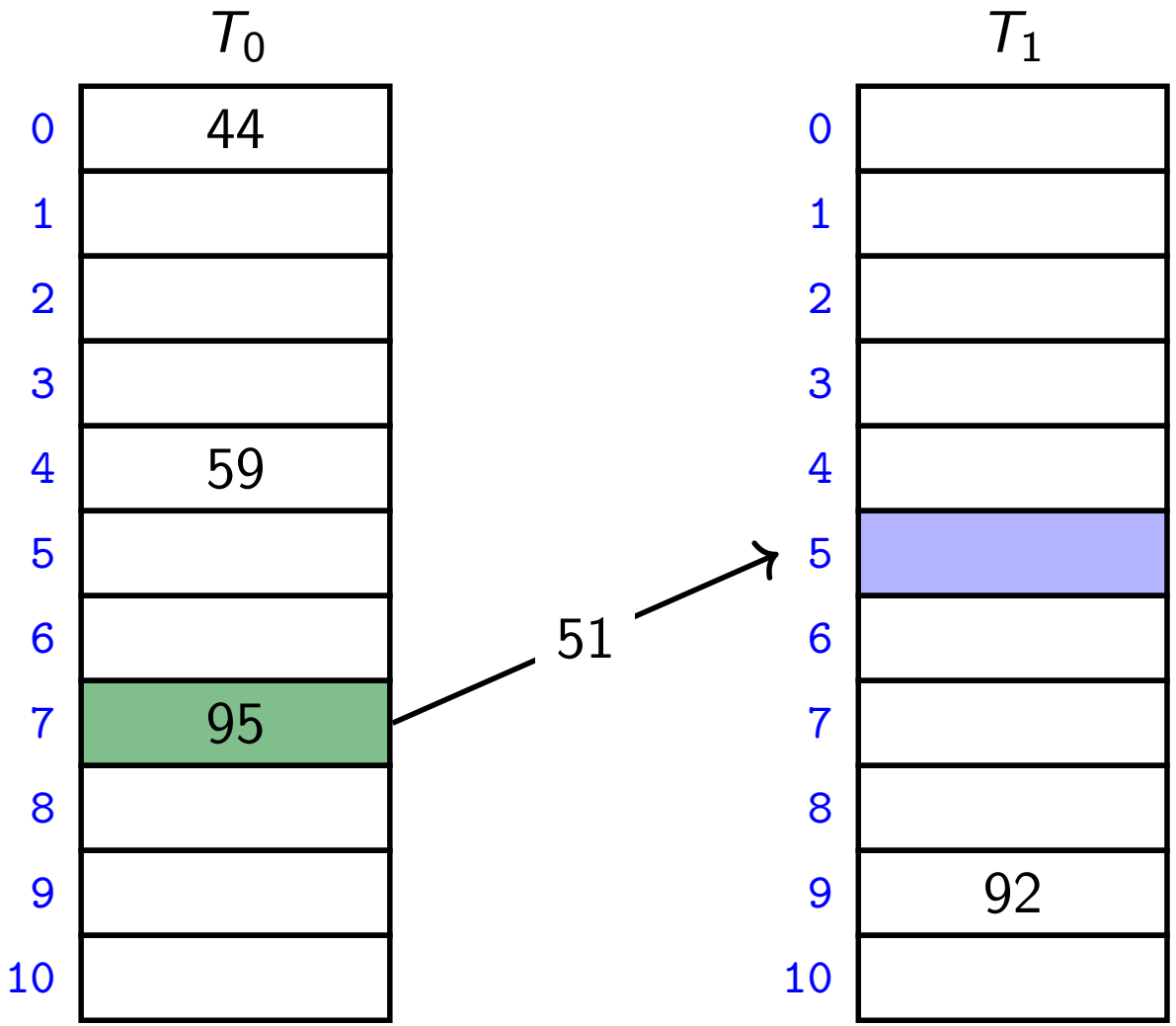
Cuckoo hashing insertion: Example

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

insert(95)

$i = 1$
$k = 51$

$h_0(k) = 7$
$h_1(k) = 5$



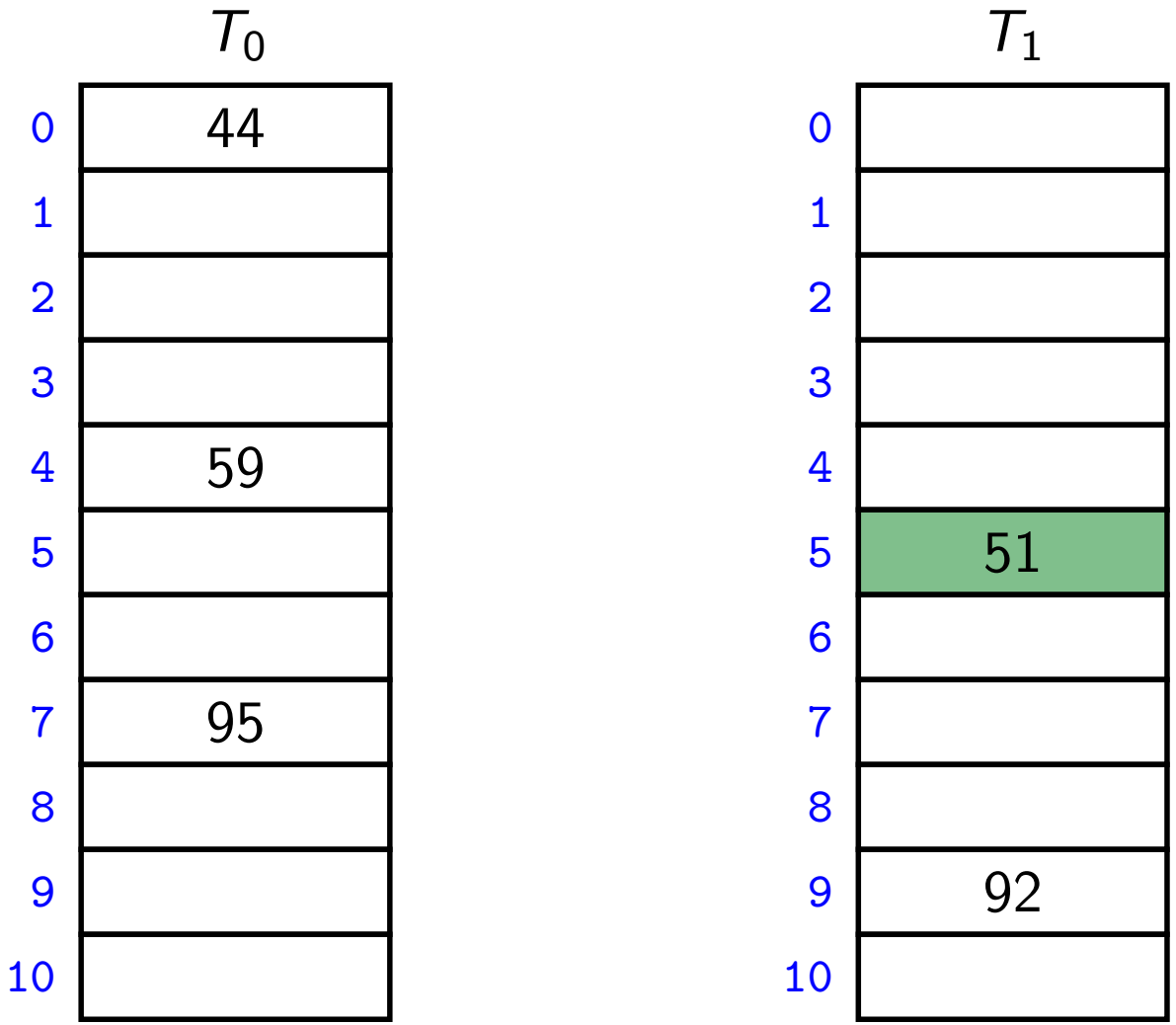
Cuckoo hashing insertion: Example

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

insert(95)

$i = 1$
$k = 51$

$h_0(k) = 7$
$h_1(k) = 5$



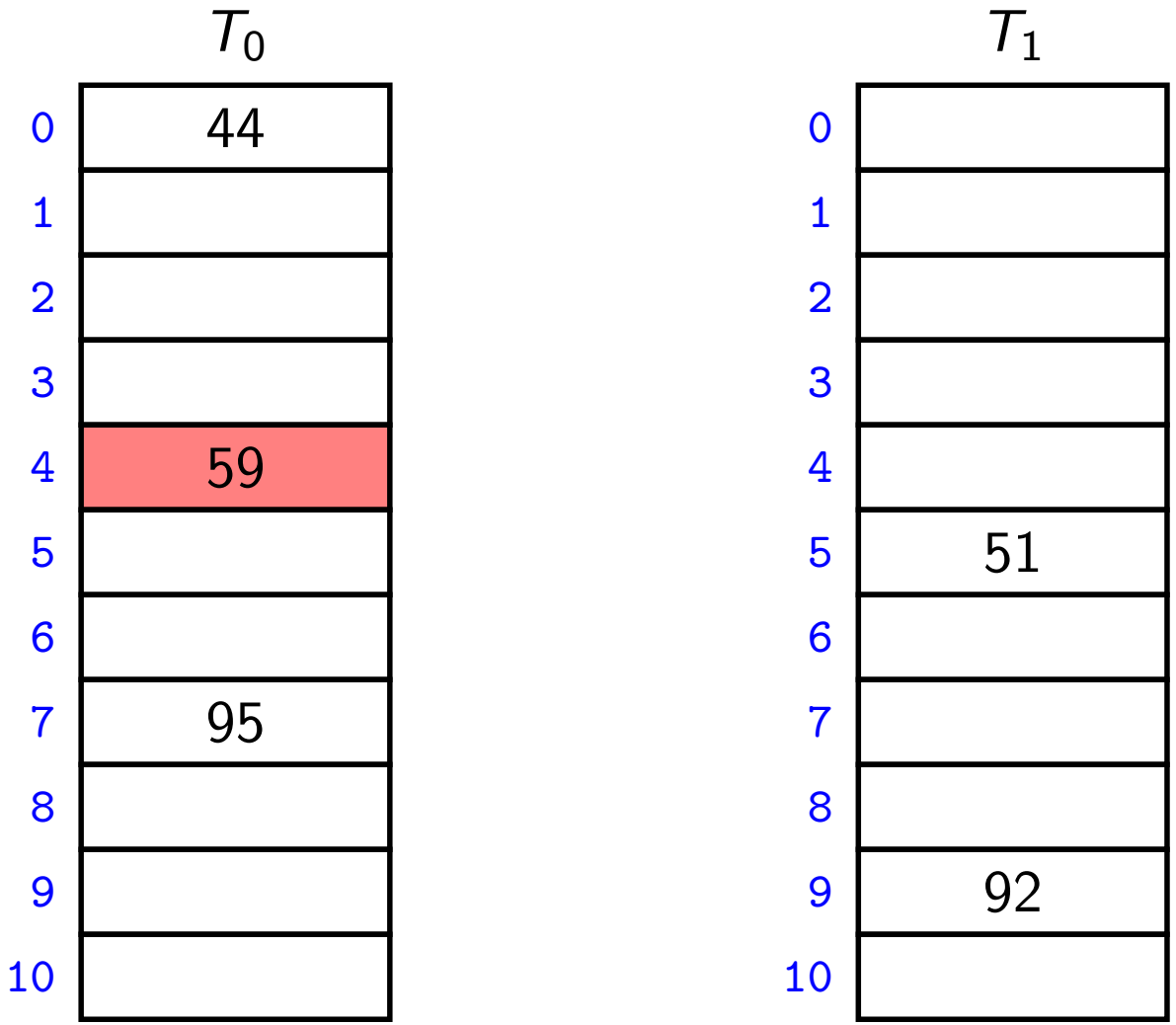
Cuckoo hashing insertion: Example

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

insert(26)

$i = 0$
$k = 26$

$h_0(k) = 4$
$h_1(k) = 0$



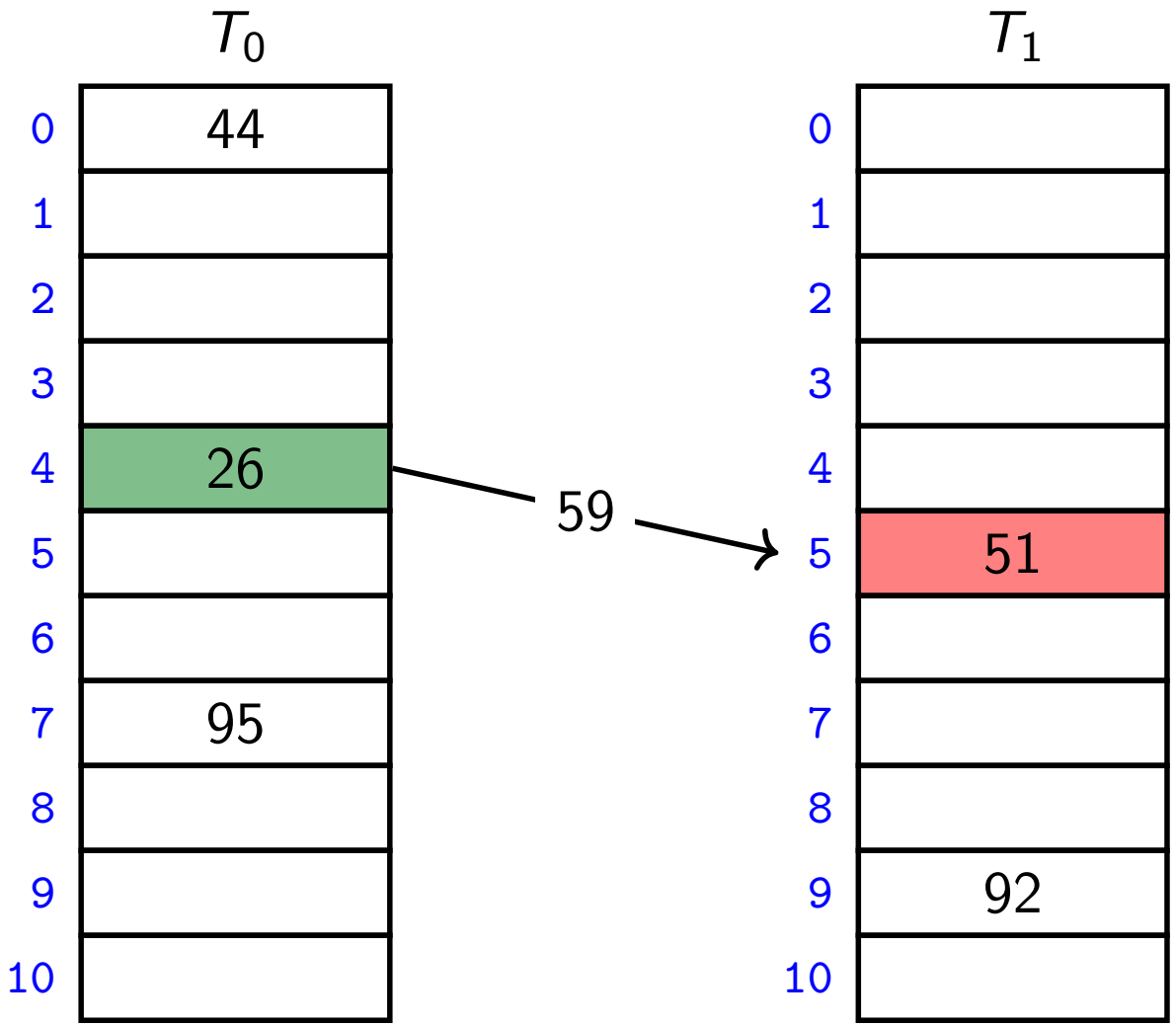
Cuckoo hashing insertion: Example

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

insert(26)

$i = 1$
$k = 59$

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



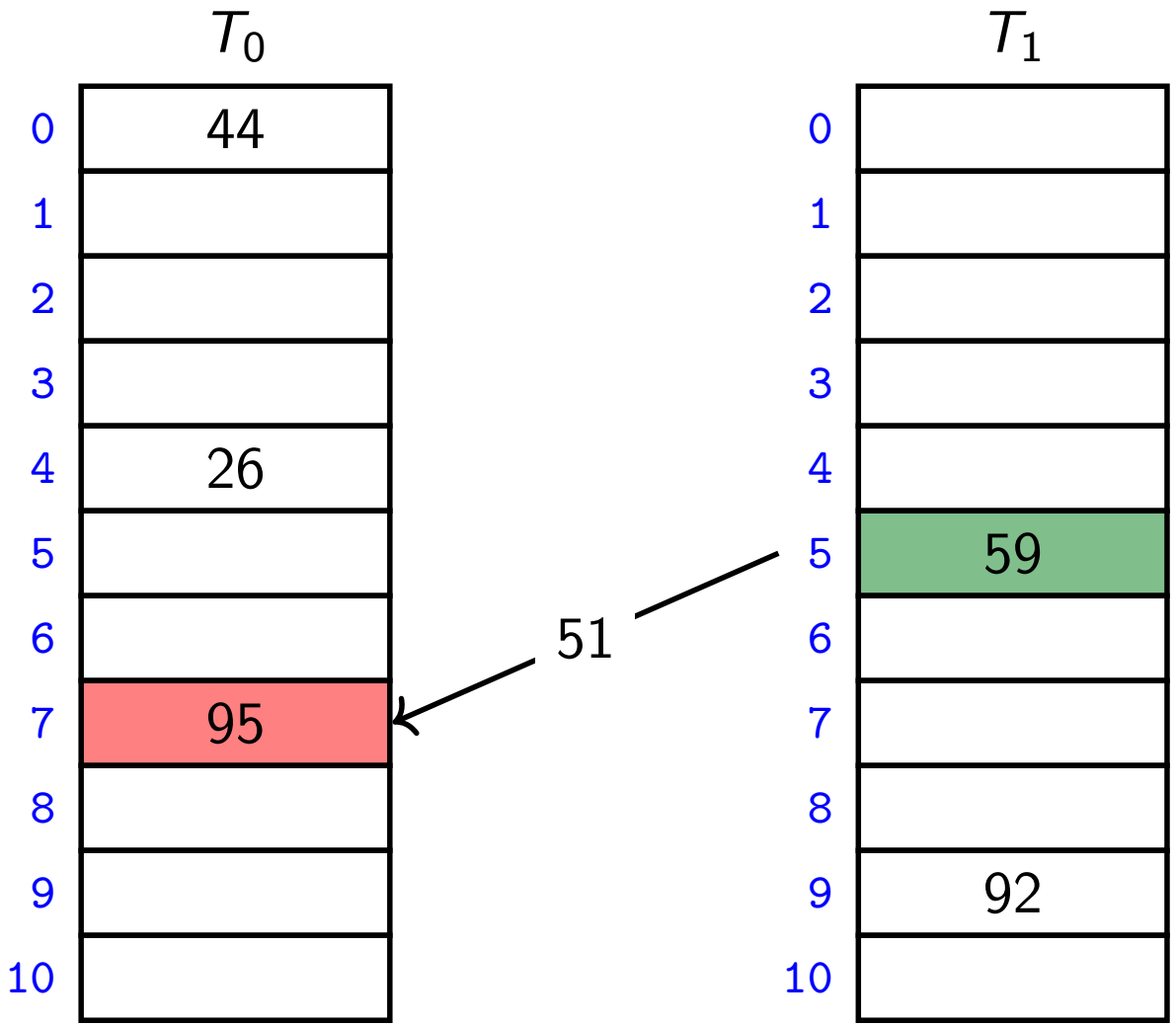
Cuckoo hashing insertion: Example

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

insert(26)

$i = 0$
$k = 51$

$h_0(k) = 7$
$h_1(k) = 5$



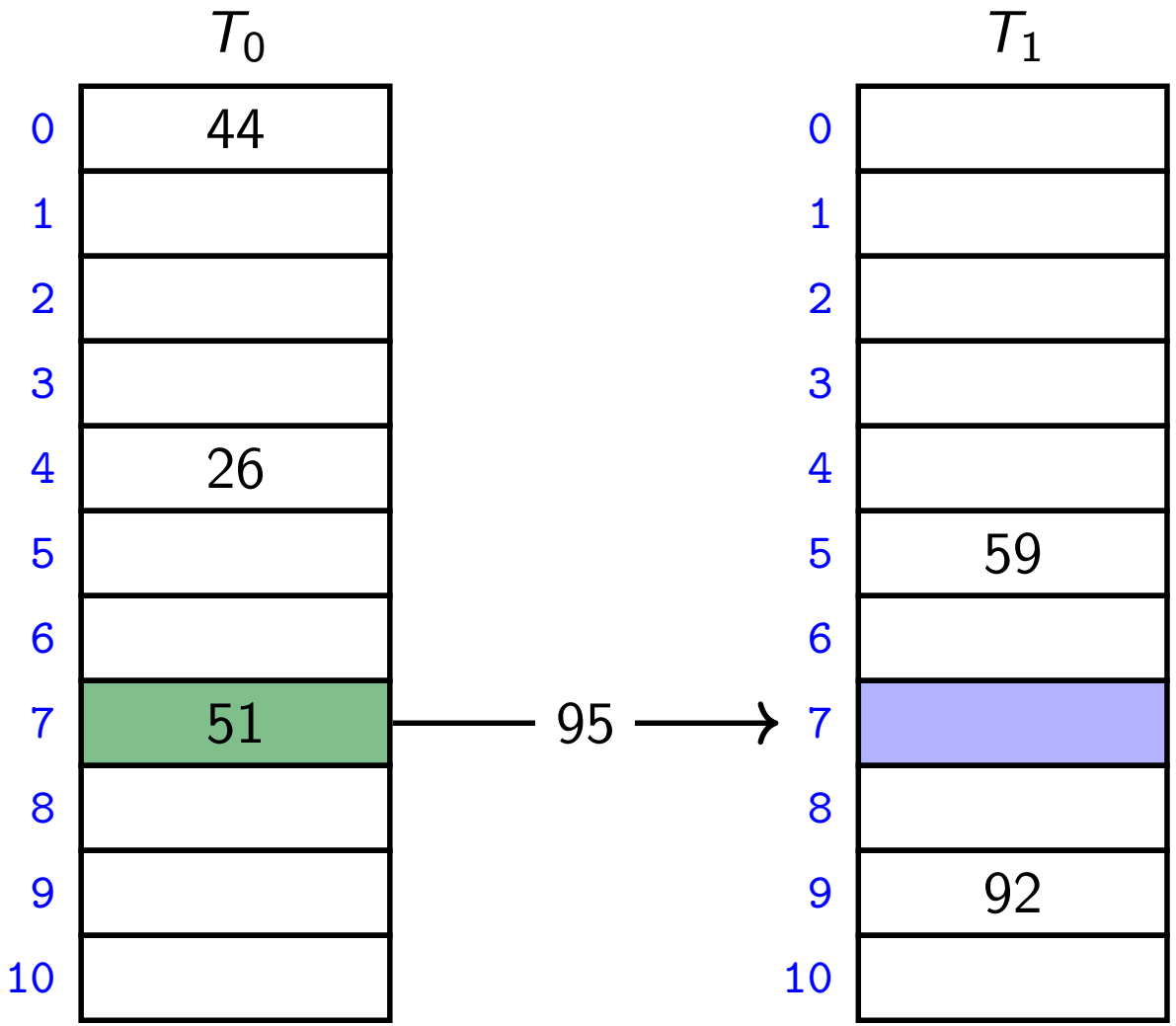
Cuckoo hashing insertion: Example

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

insert(26)

$i = 1$
$k = 95$

$h_0(k) = 4$
$h_1(k) = 7$



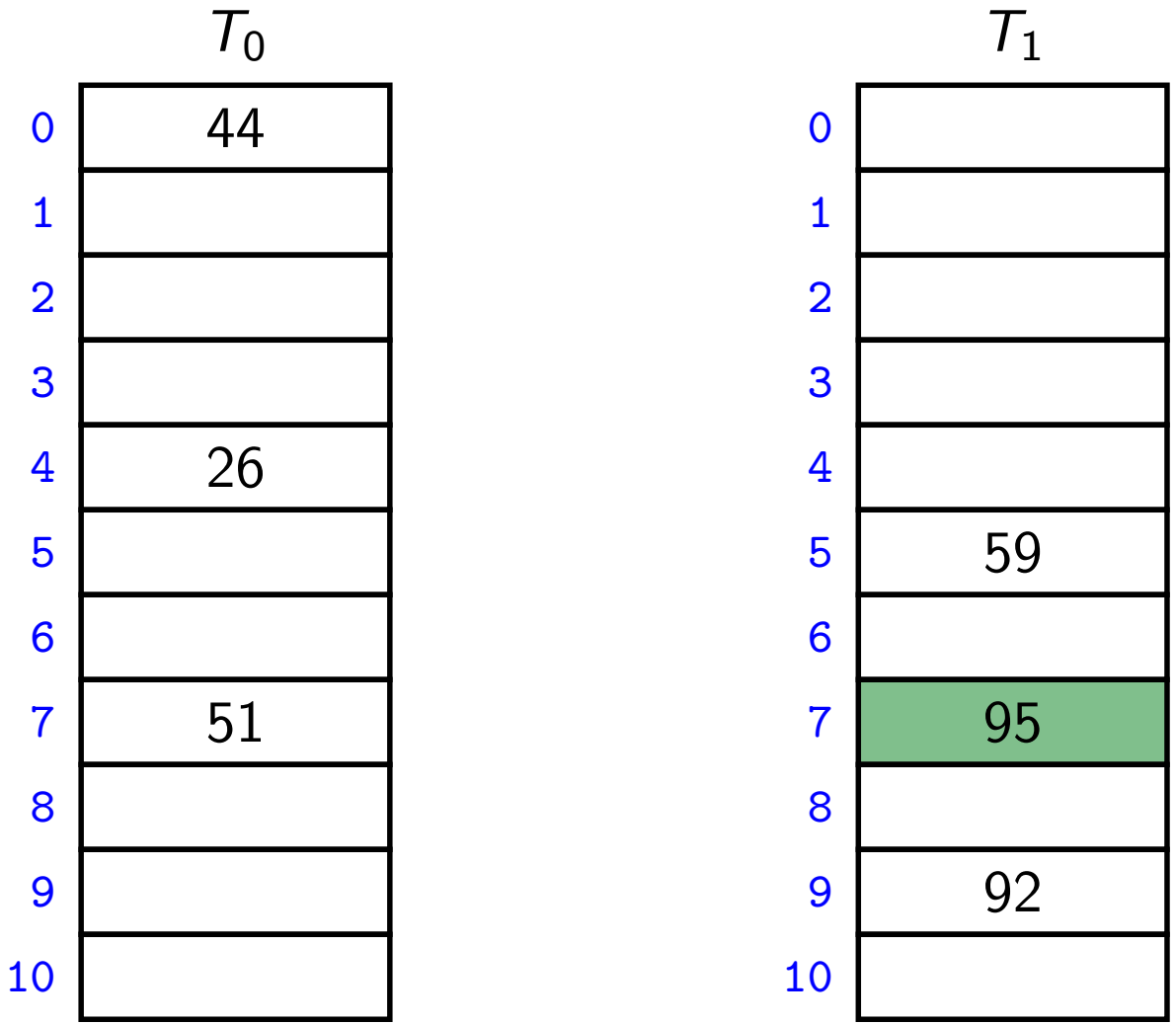
Cuckoo hashing insertion: Example

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

insert(26)

$i = 1$
$k = 95$

$h_0(k) = 4$
$h_1(k) = 7$

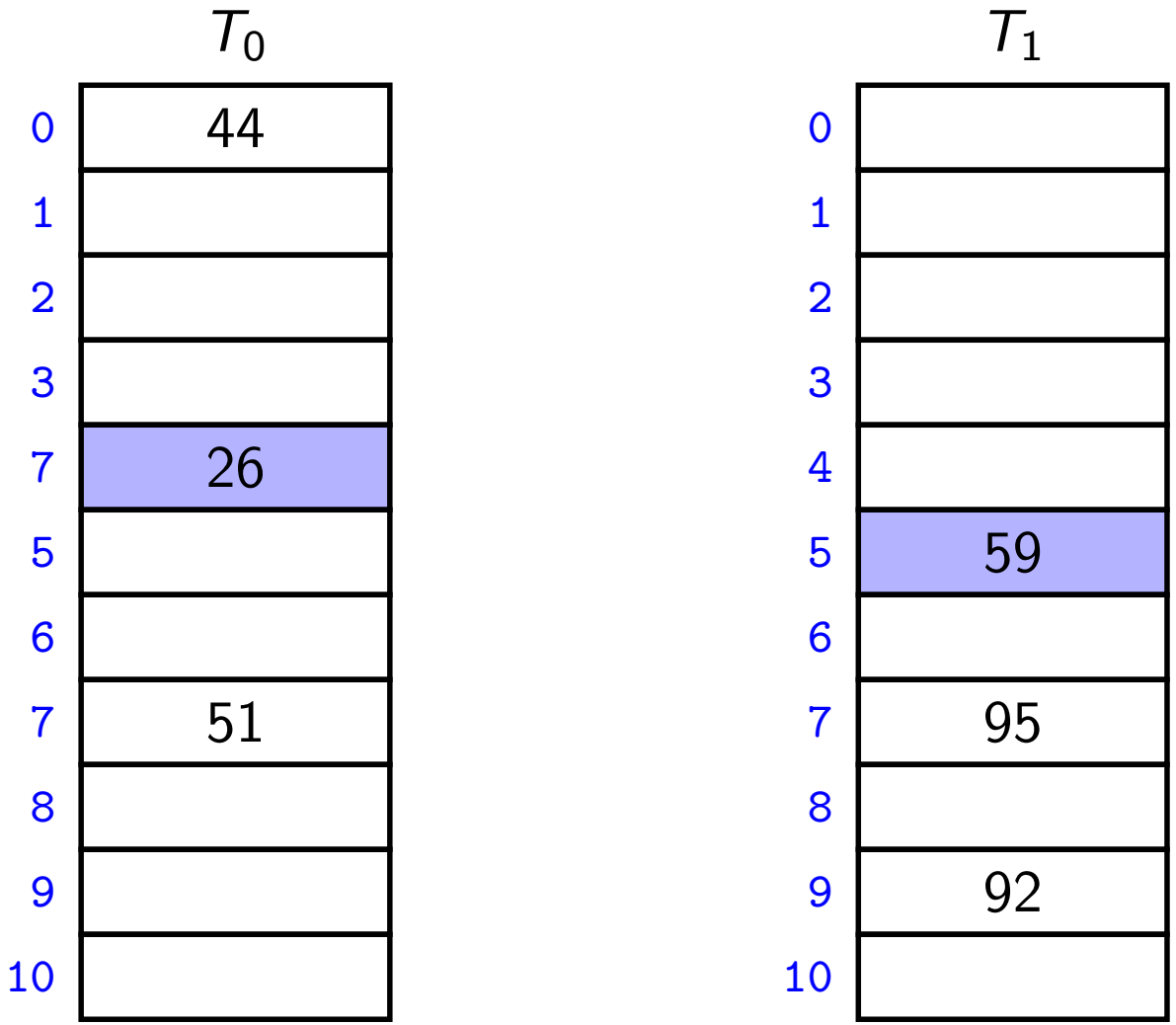


Cuckoo hashing insertion: Example

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

search(59)

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

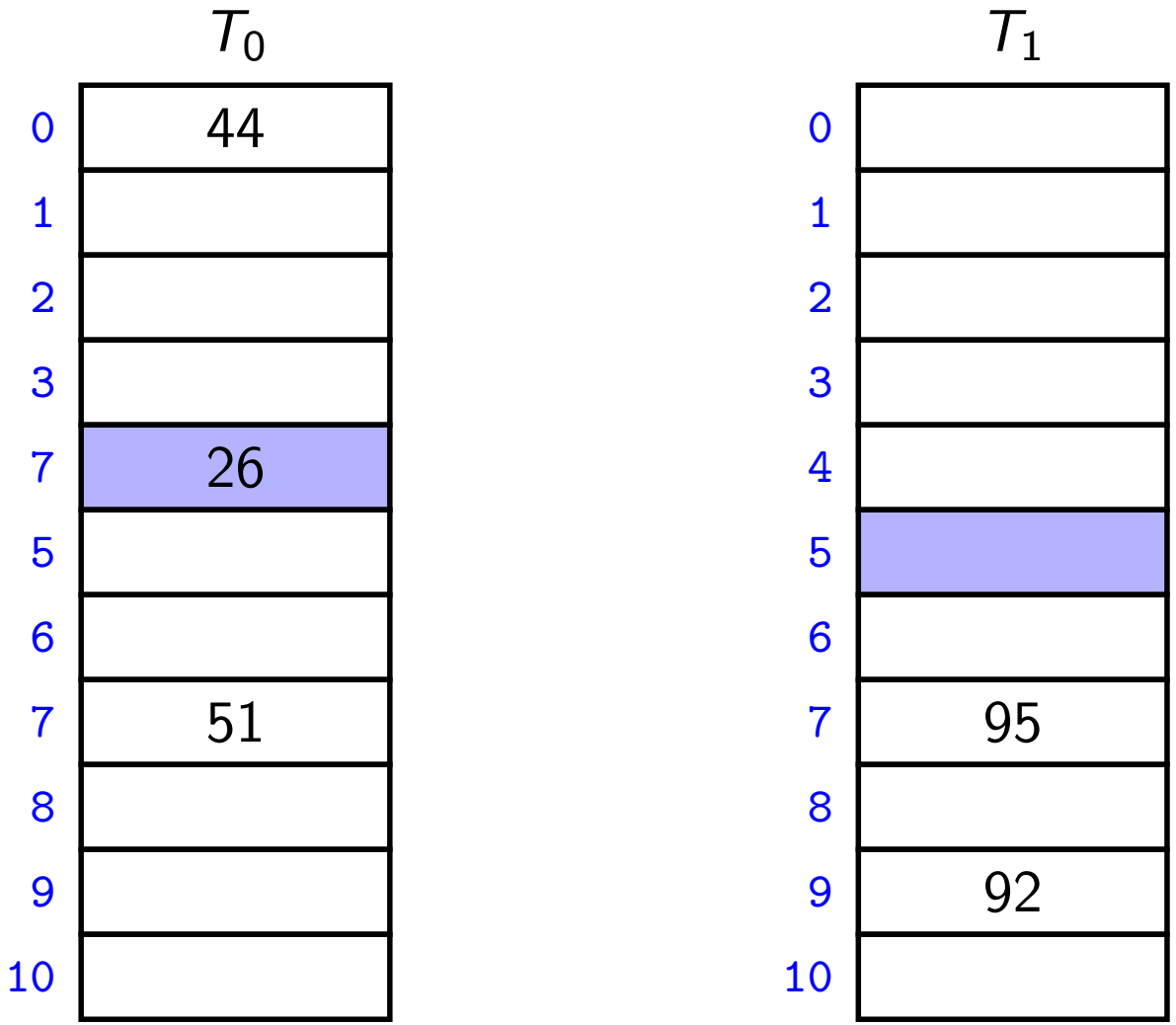


Cuckoo hashing insertion: Example

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

delete(59)

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



Cuckoo hashing: Discussion

- **Can show**: expected number of evictions during *insert* is $O(1)$.
  - So in practice, stop evictions much earlier than $2n$ rounds.
- This requires load factor $\alpha < \frac{1}{2}$.
  - Here $\alpha = n/(\text{size of } T_0 + \text{size of } T_1)$
- So cuckoo hashing is somewhat wasteful on space.

There are many possible variations:

- The two hash-tables could be combined into one.
- Be more flexible when inserting: Always consider both possible positions.
- Use $k > 2$ allowed locations (i.e., k hash-functions).

Hashing with open addressing: Summary

For any open addressing scheme, we *must* have $\alpha \leq 1$.
For the analysis, we require $0 < \alpha < 1$.
Cuckoo hashing requires $0 < \alpha < 1/2$.

Under these restrictions (and the universal hashing assumption):

- All strategies have $O(1)$ expected timefor *search*, *insert*, *delete*.
- Cuckoo Hashing has $O(1)$ worst-case time for *search*, *delete*.
- Probe sequences use $O(n)$ worst-case space, Cuckoo Hashing uses $O(n)$ expected space.

But for any hash-function the worst-case run-time is $\Theta(n)$ for *insert*.

Hashing with open addressing: Summary

For any open addressing scheme, we *must* have $\alpha \leq 1$.
For the analysis, we require $0 < \alpha < 1$.
Cuckoo hashing requires $0 < \alpha < 1/2$.

Under these restrictions (and the universal hashing assumption):

- All strategies have $O(1)$ expected timefor *search*, *insert*, *delete*.
- Cuckoo Hashing has $O(1)$ worst-case time for *search*, *delete*.
- Probe sequences use $O(n)$ worst-case space, Cuckoo Hashing uses $O(n)$ expected space.

But for any hash-function the worst-case run-time is $\Theta(n)$ for *insert*.

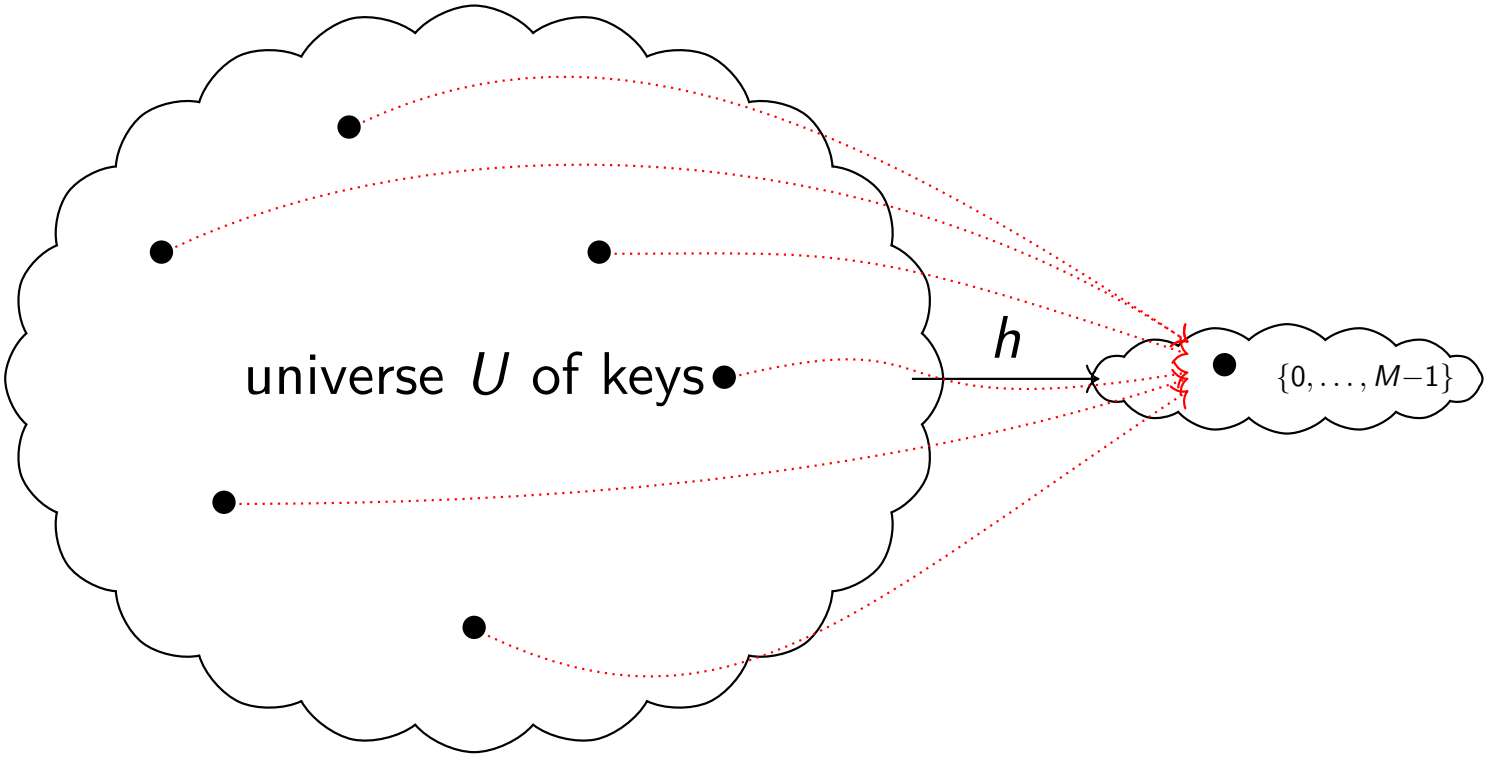
In practice, double hashing seems the most popular, or cuckoo hashing if there are many more searches than insertions.

Outline

Hash functions

Every hash function *must* do badly for some inputs:

- If the universe is big enough ($|U| \geq M(n-1)+1$), then there are n keys that all hash to the same value.



- If we insert this set of keys, then we have $\Theta(n)$ run-time.

Choosing a good hash function

- Analysis works only under **uniform hashing assumption**: Hash function is randomly chosen among all possible hash-functions.
- Satisfying this is impossible: There are too many hash functions; we would not know how to compute $h(k)$ efficiently.

Two ways to compromise:

1. Deterministic: hope for good performance by choosing a hash-function that is unrelated to any possible patterns in the data
2. Randomized: Choose randomly among a limited set of functions.
   - But aim for $Pr(\text{two keys collide}) = \frac{1}{M}$
   - This is enough to prove the expected run-time bounds for chaining

Deterministic hash functions

We saw two basic methods for integer keys:

- **Modular method**: $h(k) = k \bmod M$.
  - We should choose M to be a prime.
  - This means finding a suitable prime quickly when re-hashing.
  - This can be done in $O(M \log\log M)$ time (no details).
- **Multiplication method**: $h(k) = \lfloor M(Ak - \lfloor Ak \rfloor) \rfloor$, for some floating-point number A with $0 < A < 1$.
  - Multiplying with A is used to scramble the keys. So A should be irrational to avoid patterns in the keys.
  - Experiments show that good scrambling is achieved when A is the golden ratio $\varphi = \frac{\sqrt{5}-1}{2} \approx 0.618033988749.....$

Carter-Wegman's universal hashing

Better idea: Choose hash-function randomly!

- Requires: all keys are in $\{0, \ldots, p-1\}$ for some (big) prime p.
- At initialization, and whenever we re-hash:
  - Choose $M < p$ arbitrarily, power of 2 is ok.
  - Choose (and store) two *random* numbers a, b
    - $b = \mathit{random}(p)$
    - $a = 1 + \mathit{random}(p-1)$ (so $a \neq 0$)
  - Use as hash-function $\boxed{h_{a,b}(k) = \big((ak+b) \bmod p\big) \bmod M}$
- hash-value can be computed in constant time.

Analysis of these **Carter-Wegman hash functions** (no details):

- Choosing h in this way does not satisfy uniform hashing assumption
- But can show: two keys collide with probability at most $\frac{1}{M}$.
- This suffices to prove the run-time bounds for hashing with chaining.

Multi-dimensional data

What if the keys occupy more than one word in memory, e.g. strings?

Standard approach is to *flatten* string w to integer $f(w) \in \mathbb{N}$, e.g.

$$\begin{aligned} A \cdot P \cdot P \cdot L \cdot E &\rightarrow (65, 80, 80, 76, 69) \quad \text{(ASCII)} \\ &\rightarrow 65R^4 + 80R^3 + 80R^2 + 76R^1 + 69R^0 \\ &\qquad \text{(for some radix } R\text{, e.g. } R = 255) \end{aligned}$$

We combine this with a modular hash function: $\boxed{h(w) = f(w) \bmod M}$

To compute this, can use Horner's rule and apply mod early. For example, $h(APPLE)$ is

$$\Big(\Big(\Big(\Big(\Big(\big((65R{+}80) \bmod M\big)R{+}80\Big) \bmod M\Big)R{+}76\Big) \bmod M\Big)R{+}69\Big) \bmod M$$