

# CS 240 – Data Structures and Data Management

## Module 7: Dictionaries via Hashing

Arne Storjohann

Based on lecture notes by many previous cs240 instructors

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

Fall 2024

*version 2024-10-24 07:47*

# Outline

## 7 Dictionaries via Hashing

- Hashing Introduction
- Hashing with Chaining
- Probe Sequences
- Cuckoo hashing
- Hash Function Strategies

# Outline

## 7 Dictionaries via Hashing

- Hashing Introduction
- Hashing with Chaining
- Probe Sequences
- Cuckoo hashing
- Hash Function Strategies

# Direct Addressing

**Special situation:** For a known  $M \in \mathbb{N}$ , every key  $k$  is an integer with  $0 \leq k < M$ .

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

0	
1	
2	dog
3	
4	
5	
6	cat
7	
8	pig

- *search*( $k$ ): Check whether  $A[k]$  is NULL
- *insert*( $k, v$ ):  $A[k] \leftarrow v$
- *delete*( $k$ ):  $A[k] \leftarrow \text{NULL}$

# Direct Addressing

**Special situation:** For a known  $M \in \mathbb{N}$ , every key  $k$  is an integer with  $0 \leq k < M$ .

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

0	
1	
2	dog
3	
4	
5	
6	cat
7	
8	pig

- *search*( $k$ ): Check whether  $A[k]$  is NULL
- *insert*( $k, v$ ):  $A[k] \leftarrow v$
- *delete*( $k$ ):  $A[k] \leftarrow \text{NULL}$

Each operation is  $\Theta(1)$ .

Total space is  $\Theta(M)$ .

What sorting algorithm does this remind you of?

# Direct Addressing

**Special situation:** For a known  $M \in \mathbb{N}$ , every key  $k$  is an integer with  $0 \leq k < M$ .

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

0	
1	
2	dog
3	
4	
5	
6	cat
7	
8	pig

- *search*( $k$ ): Check whether  $A[k]$  is NULL
- *insert*( $k, v$ ):  $A[k] \leftarrow v$
- *delete*( $k$ ):  $A[k] \leftarrow \text{NULL}$

Each operation is  $\Theta(1)$ .

Total space is  $\Theta(M)$ .

What sorting algorithm does this remind you of?

*Bucket Sort*

# Hashing

Two disadvantages of direct addressing:

- It cannot be used if the keys are not integers.
- It wastes space if  $M$  is unknown or  $n \ll M$ .

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

Details:

- **Assumption:** We know that all keys come from some **universe**  $U$ . (Typically  $U =$  non-negative integers, sometimes  $|U|$  finite.)
- We pick a **table-size**  $M$ .
- We pick a **hash function**  $h : U \rightarrow \{0, 1, \dots, M-1\}$ . (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).

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

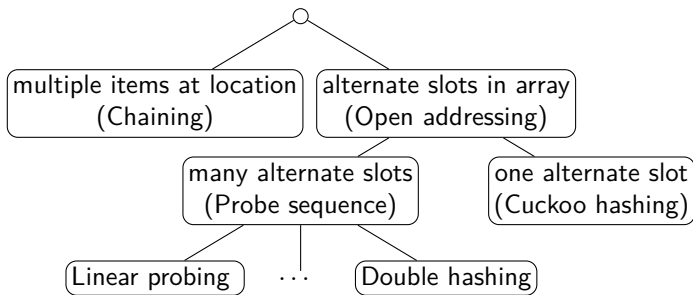


# Collisions

- 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.

# 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

## 7 Dictionaries via Hashing

- Hashing Introduction
- Hashing with Chaining
- Probe Sequences
- Cuckoo hashing
- Hash Function Strategies

# 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**.

# Hashing with Chaining

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

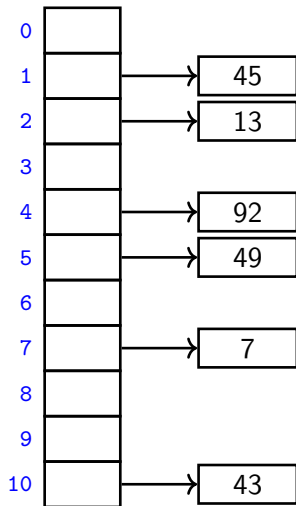
- A bucket could be implemented by any dictionary realization (even another hash table!).
- The simplest approach is to use unsorted lists with MTF for buckets. This is called collision resolution by **chaining**.
- *insert*( $k, v$ ): Add ( $k, v$ ) to the front of the list at  $T[h(k)]$ .
- *search*( $k$ ): 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  $O(1 + \text{length of list at } T(h(k)))$ .

## Chaining example

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

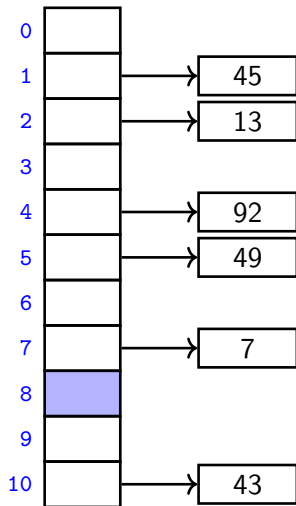


## Chaining example

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

*insert*(41)

$$h(41) = 8$$

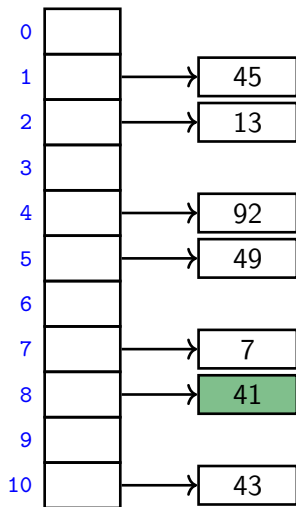


## Chaining example

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

*insert*(41)

$$h(41) = 8$$



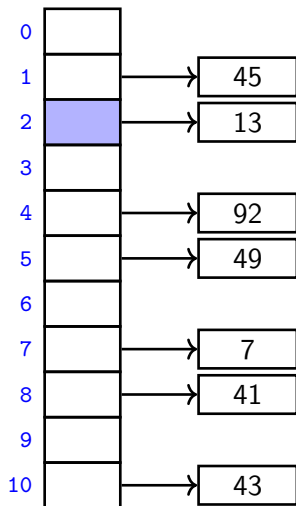


## Chaining example

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

*insert*(46)

$$h(46) = 2$$

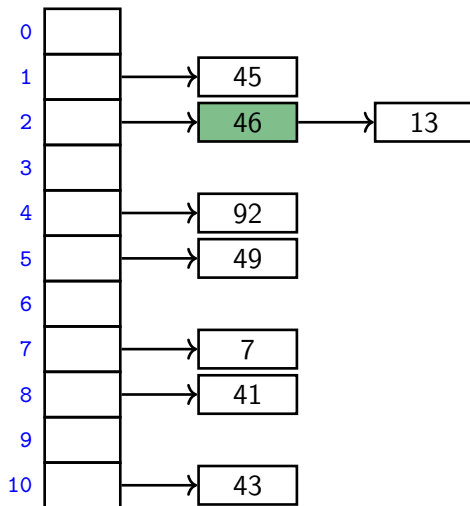


## Chaining example

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

*insert*(46)

$$h(46) = 2$$

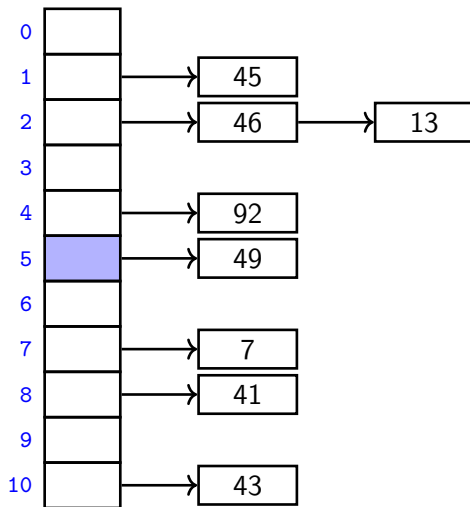


## Chaining example

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

*insert*(16)

$$h(16) = 5$$

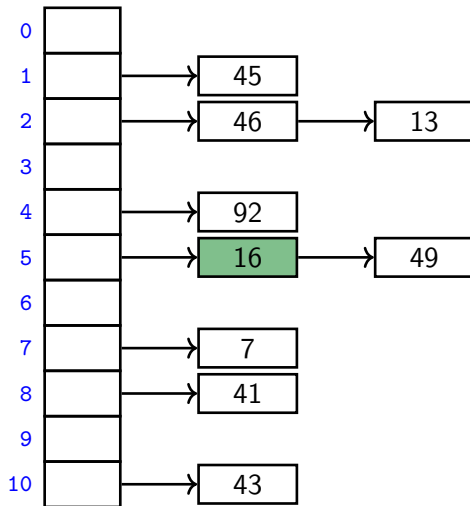


## Chaining example

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

*insert*(16)

$$h(16) = 5$$

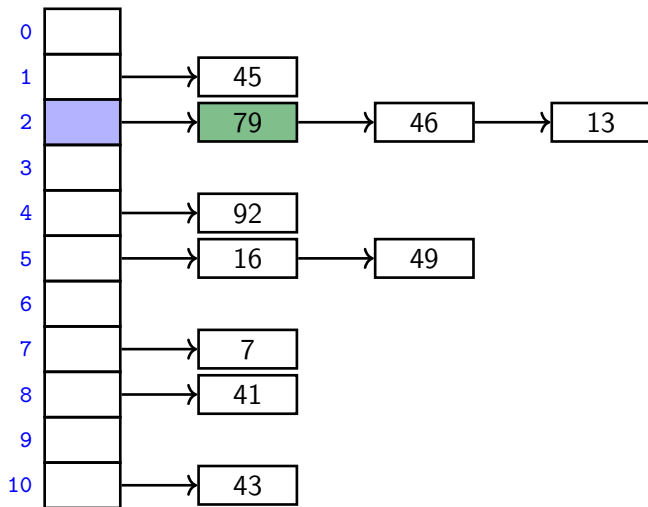


## Chaining example

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

*insert*(79)

$$h(79) = 2$$



# Complexity of chaining

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

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

- The *average* bucket-size is  $\frac{n}{M} =: \alpha$ .  
( $\alpha$  is also called the **load factor**.)

# Complexity of chaining

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

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

- The *average* bucket-size is  $\frac{n}{M} =: \alpha$ .  
( $\alpha$  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  $\alpha$
  - ▶ But the operations take  $\Theta(n)$  time on average
- To get meaningful average-case bounds, we need some assumptions on the hash-functions and the keys!

# Complexity of chaining

- To analyze what happens ‘on average’, switch to *randomized* hashing.
- How can we randomize?



# Complexity of chaining

- 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.
- To be able to analyze, we assume the following:

***Uniform Hashing Assumption:*** Any possible hash-function is equally likely to be chosen as hash-function.

(This is not at all realistic, but the assumption makes analysis possible.)

# Complexity of chaining

UHA implies that the distribution of keys is unimportant.

- **Claim:** Hash-values are uniform.

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

- ▶ Let  $\mathcal{H}_j$  (for  $j = 0, \dots, M-1$ ) be hash-functions with  $h(k) = j$ .
- ▶ For any  $i \neq j$ , can map  $\mathcal{H}_i$  to  $\mathcal{H}_j$  and vice versa.
- ▶ So  $P(h(k) = i) = P(h \in \mathcal{H}_i) = \frac{1}{M}$ .

- **Claim:** Hash-values of any two keys are independent of each other.

Proof: similar

# Complexity of chaining

UHA implies that the distribution of keys is unimportant.

- **Claim:** Hash-values are uniform.

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

- ▶ Let  $\mathcal{H}_j$  (for  $j = 0, \dots, M-1$ ) be hash-functions with  $h(k) = j$ .
- ▶ For any  $i \neq j$ , can map  $\mathcal{H}_i$  to  $\mathcal{H}_j$  and vice versa.
- ▶ So  $P(h(k) = i) = P(h \in \mathcal{H}_i) = \frac{1}{M}$ .

- **Claim:** Hash-values of any two keys are independent of each other.

Proof: similar

Back to complexity of chaining:

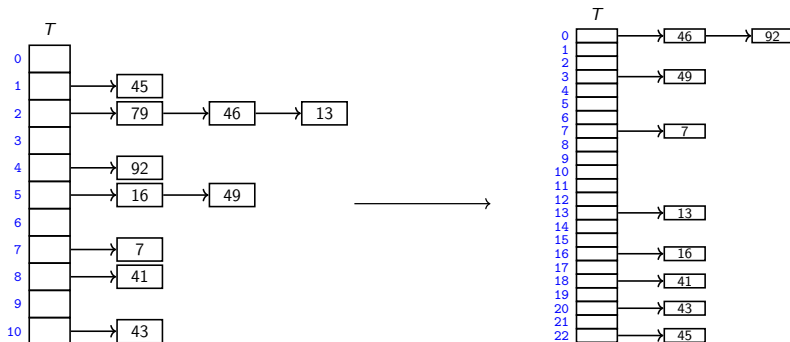
- Each bucket has expected length  $\frac{n}{M} \leq \alpha$ 
  - ▶  $n$  other keys are in this slot with probability  $\frac{1}{M}$
- Each key in dictionary is expected to collide with  $\frac{n-1}{M}$  other keys
  - ▶  $n-1$  other keys are in same slot with probability  $\frac{1}{M}$
- Expected cost of *search* and *delete* is hence  $\Theta(1 + \alpha)$

# Load factor and re-hashing

- For hashing with chaining (and also other collision resolution strategies), the run-time bound depends on  $\alpha$

(Recall: *load factor*  $\alpha = n/M$ .)

- We keep the load factor small by **rehashing** when needed:



- ▶ Keep track of  $n$  and  $M$  throughout operations
- ▶ If  $\alpha$  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  $\alpha \in \Theta(1)$  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.
- We should also re-hash when  $\alpha$  gets too small, so that  $M \in \Theta(n)$  throughout, and the space is always  $\Theta(n)$ .

**Summary:** The amortized expected cost for hashing with chaining is  $O(1)$  and the space is  $\Theta(n)$   
(assuming uniform hashing and  $\alpha \in \Theta(1)$  throughout)

Theoretically perfect, but too slow in practice.

# Outline

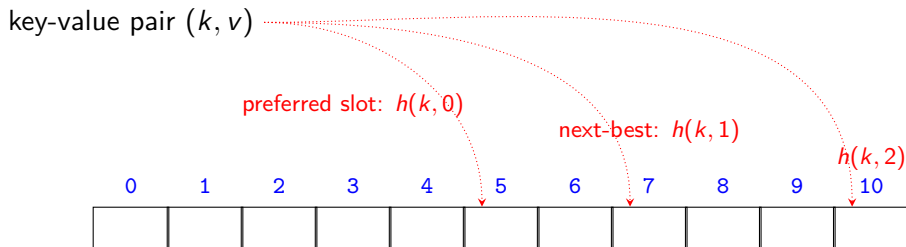
## 7 Dictionaries via Hashing

- Hashing Introduction
- Hashing with Chaining
- **Probe Sequences**
- Cuckoo hashing
- Hash Function Strategies

# 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 multiple slots.

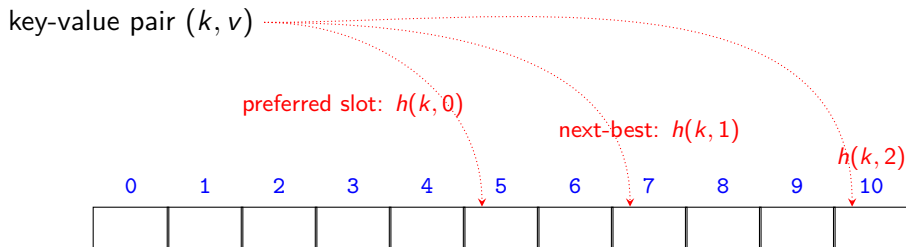
*search* and *insert* follow a **probe sequence** of possible locations for key  $k$ :  $\langle h(k, 0), h(k, 1), h(k, 2), \dots, h(k, M-1) \rangle$  until an empty spot is found.



# 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 multiple slots.

*search* and *insert* follow a **probe sequence** of possible locations for key  $k$ :  $\langle h(k, 0), h(k, 1), h(k, 2), \dots, h(k, M-1) \rangle$  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  $h$ .



## Linear probing example

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

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

## Linear probing example

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

*insert*(41)

$$h(41, 0) = 8$$

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

## Linear probing example

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

*insert*(84)

$$h(84, 0) = 7$$

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

## Linear probing example

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

*insert*(84)

$$h(84, 1) = 8$$

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

## Linear probing example

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

*insert*(84)

$$h(84, 2) = 9$$

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

## Linear probing example

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

*insert*(20)

$$h(20, 0) = 9$$

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

## Linear probing example

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

*insert*(20)

$$h(20, 1) = 10$$

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

## Linear probing example

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

*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



# Probe sequence operations

*delete* becomes problematic:

- Cannot leave an empty spot behind; the next search might otherwise not go far enough.

# Probe sequence 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.)

# Probe sequence 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.

# Probe sequence 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.

## Linear probing example

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

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

## Linear probing example

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

*delete*(43)

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

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

## Linear probing example

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

*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	<i>deleted</i>

## Linear probing example

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

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

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



## Linear probing example

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

*search*(63)

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

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

## Linear probing example

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

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

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

## Linear probing example

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

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

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

## Linear probing example

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

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

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

## Linear probing example

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

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

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

## Probe sequence 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  $P(h_0(k) = i)$  and  $P(h_1(k) = j)$  are independent.
- Using two modular hash-functions often leads to dependencies.

# 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  $P(h_0(k) = i)$  and  $P(h_1(k) = j)$  are independent.
- Using two modular hash-functions often leads to dependencies.
- Better idea: Use *multiplication method* for second hash function:

$$h(k) = \lfloor M(kA - \lfloor kA \rfloor) \rfloor$$

- ▶  $A$  is some floating-point number with  $0 < A < 1$
- ▶  $kA - \lfloor kA \rfloor$  computes fractional part of  $kA$ , which is in  $[0, 1)$
- ▶ Multiply with  $M$  to get floating-point number in  $[0, M)$
- ▶ Round down to get integer in  $\{0, \dots, M-1\}$

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



# Double Hashing

- Assume we have two hash independent functions  $h_0, h_1$ .
- Assume further that  $h_1(k) \neq 0$  and that  $h_1(k)$  is relative prime with the table-size  $M$  for all keys  $k$ .
  - ▶ 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**: open addressing with probe sequence

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

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

## Double hashing example

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

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

## Double hashing example

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

*insert*(41)

$$h_0(41) = 8$$

$$h(41, 0) = 8$$

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

## Double hashing example

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

*insert*(194)

$$h_0(194) = 7$$

$$h(194, 0) = 7$$

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

## Double hashing example

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

*insert*(194)

$$h_0(194) = 7$$

$$h(194, 0) = 7$$

$$h_1(194) = 9$$

$$h(194, 1) = 5$$

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

## Double hashing example

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

*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

## 7 Dictionaries via Hashing

- Hashing Introduction
- Hashing with Chaining
- Probe Sequences
- Cuckoo hashing
- Hash Function Strategies

# 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.

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



# 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.      $(k_{evict}, v_{evict}) \leftarrow T_i[h_i(k_{insert})]$      // save old KVP
5.      $T_i[h_i(k_{insert})] \leftarrow (k_{insert}, v_{insert})$      // put in new KVP
6.     **if**  $(k_{evict}, v_{evict})$  is NULL **return** “success”
7.     **else**     // repeat in other table
8.          $(k_{insert}, v_{insert}) \leftarrow (k_{evict}, v_{evict}); i \leftarrow 1 - i$
9. **return** “failure to insert”     // need to re-hash

# Cuckoo hashing example

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

$T_0$

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

$T_1$

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

# Cuckoo hashing example

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

*insert*(51)

$$i = 0$$

$$k = 51$$

$$h_0(k) = 7$$

$$h_1(k) = 5$$

$T_0$	
0	44
1	
2	
3	
4	59
5	
6	
7	
8	
9	
10	

$T_1$	
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	92
10	

# Cuckoo hashing example

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

*insert*(51)

$$i = 0$$

$$k = 51$$

$$h_0(k) = 7$$

$$h_1(k) = 5$$

$T_0$	
0	44
1	
2	
3	
4	59
5	
6	
7	51
8	
9	
10	

$T_1$	
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	92
10	

# Cuckoo hashing example

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

*insert*(95)

$$i = 0$$

$$k = 95$$

$$h_0(k) = 7$$

$$h_1(k) = 7$$

$T_0$	
0	44
1	
2	
3	
4	59
5	
6	
7	51
8	
9	
10	

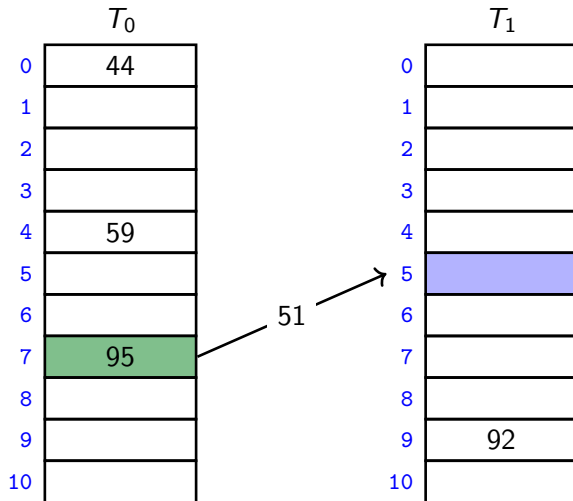
$T_1$	
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	92
10	

# Cuckoo hashing example

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

*insert*(95)

$$\begin{aligned} i &= 1 \\ k &= 51 \\ h_0(k) &= 7 \\ h_1(k) &= 5 \end{aligned}$$



# Cuckoo hashing example

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

*insert*(95)

$$\begin{aligned} i &= 1 \\ k &= 51 \\ h_0(k) &= 7 \\ h_1(k) &= 5 \end{aligned}$$

$T_0$	
0	44
1	
2	
3	
4	59
5	
6	
7	95
8	
9	
10	

$T_1$	
0	
1	
2	
3	
4	
5	51
6	
7	
8	
9	92
10	

# Cuckoo hashing example

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

*insert*(26)

$$i = 0$$

$$k = 26$$

$$h_0(k) = 4$$

$$h_1(k) = 0$$

$T_0$	
0	44
1	
2	
3	
4	59
5	
6	
7	95
8	
9	
10	

$T_1$	
0	
1	
2	
3	
4	
5	51
6	
7	
8	
9	92
10	



# Cuckoo hashing example

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

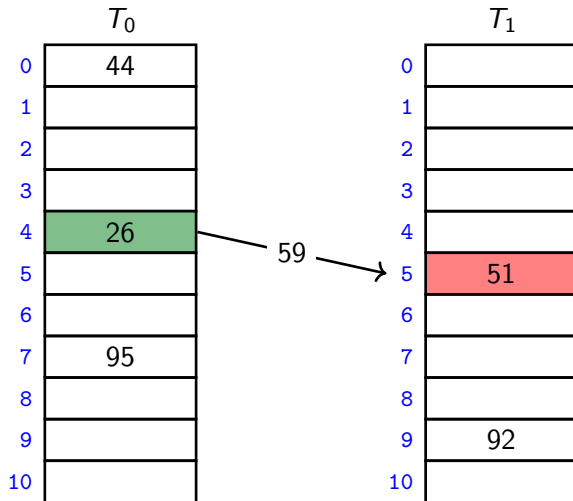
*insert*(26)

$$i = 1$$

$$k = 59$$

$$h_0(k) = 4$$

$$h_1(k) = 5$$



# Cuckoo hashing example

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

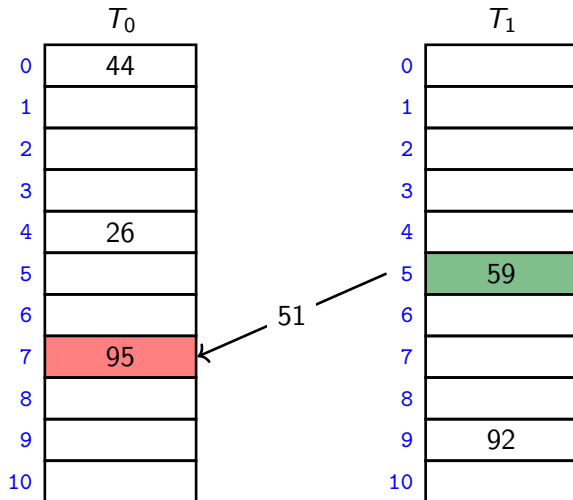
*insert*(26)

$$i = 0$$

$$k = 51$$

$$h_0(k) = 7$$

$$h_1(k) = 5$$



# Cuckoo hashing example

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

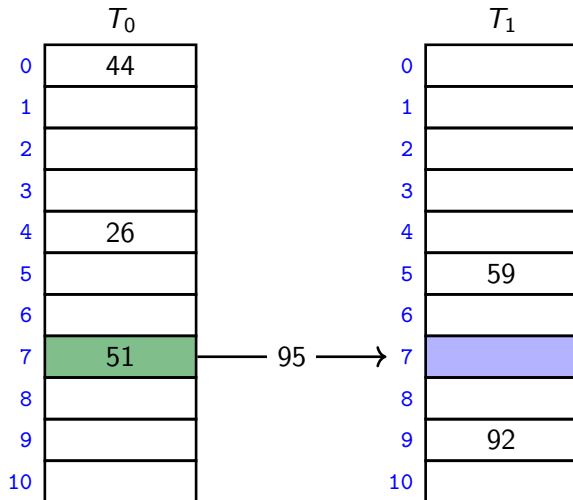
*insert*(26)

$$i = 1$$

$$k = 95$$

$$h_0(k) = 4$$

$$h_1(k) = 7$$



# Cuckoo hashing example

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

*insert*(26)

$$\begin{aligned} i &= 1 \\ k &= 95 \\ h_0(k) &= 4 \\ h_1(k) &= 7 \end{aligned}$$

$T_0$	
0	44
1	
2	
3	
4	26
5	
6	
7	51
8	
9	
10	

$T_1$	
0	
1	
2	
3	
4	
5	59
6	
7	95
8	
9	92
10	

# Cuckoo hashing example

$$M = 11,$$

$$h_0(k) = k \bmod 11,$$

$$h_1(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$$

*search*(59)

$$h_0(59) = 4$$

$$h_1(59) = 5$$

$T_0$	
0	44
1	
2	
3	
7	26
5	
6	
7	51
8	
9	
10	

$T_1$	
0	
1	
2	
3	
4	
5	59
6	
7	95
8	
9	92
10	

# Cuckoo hashing example

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

*delete*(59)

$$h_0(59) = 4$$

$$h_1(59) = 5$$

$T_0$	
0	44
1	
2	
3	
4	
5	26
6	
7	51
8	
9	
10	

$T_1$	
0	
1	
2	
3	
4	
5	
6	
7	95
8	
9	92
10	

# Cuckoo hashing discussions

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

# Cuckoo hashing discussions

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

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).



# Complexity of open addressing strategies

For any open addressing scheme, we *must* have  $\alpha \leq 1$  (why?).

For the analysis, we require  $0 < \alpha < 1$  (not arbitrarily close).

Cuckoo hashing requires  $0 < \alpha < 1/2$  (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 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*.

# Complexity of open addressing strategies

For any open addressing scheme, we *must* have  $\alpha \leq 1$  (why?).

For the analysis, we require  $0 < \alpha < 1$  (not arbitrarily close).

Cuckoo hashing requires  $0 < \alpha < 1/2$  (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 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

## 7 Dictionaries via Hashing

- Hashing Introduction
- Hashing with Chaining
- Probe Sequences
- Cuckoo hashing
- Hash Function Strategies

# Choosing a good hash function

- Recall **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 look up  $h(k)$ .
- We need to compromise:
  - ▶ Choose a hash-function that is easy to compute.
  - ▶ But aim for  $P(\text{two keys collide}) = \frac{1}{M}$  w.r.t. key-distribution.
  - ▶ This is enough to prove the expected run-time bounds for chaining

# Choosing a good hash function

- Recall **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 look up  $h(k)$ .
- We need to compromise:
  - ▶ Choose a hash-function that is easy to compute.
  - ▶ But aim for  $P(\text{two keys collide}) = \frac{1}{M}$  w.r.t. key-distribution.
  - ▶ This is enough to prove the expected run-time bounds for chaining
- In practice: hope for good performance by choosing a hash-function that is
  - ▶ unrelated to any possible patterns in the data, and
  - ▶ depends on all parts of the key.

# Choosing a good hash function

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  $O(M \log \log n)$  time (no details).

# Choosing a good hash function

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  $O(M \log \log n)$  time (no details).
- **Multiplication method:**  $h(k) = \lfloor M(kA - \lfloor kA \rfloor) \rfloor$ ,  
for some 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  $\varphi = \frac{\sqrt{5}-1}{2} \approx 0.618033988749\dots$
  - ▶ We should use at least  $\log |U| + \log |M|$  bits of  $A$ .

# Choosing a good hash function

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  $O(M \log \log n)$  time (no details).
- **Multiplication method:**  $h(k) = \lfloor M(kA - \lfloor kA \rfloor) \rfloor$ ,  
for some 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  $\varphi = \frac{\sqrt{5}-1}{2} \approx 0.618033988749\dots$
  - ▶ We should use at least  $\log |U| + \log |M|$  bits of  $A$ .

But every hash function *must* do badly for some sequences of inputs:

- If the universe contains at least  $M \cdot n$  keys, then there are  $n$  keys that all hash to the same value.



# Carter-Wegman's universal hashing

**Better idea:** Choose hash-function randomly!

- Requires: all keys are in  $\{0, \dots, p-1\}$  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  $a, b$ 
    - ★  $b = \text{random}(p)$
    - ★  $a = 1 + \text{random}(p-1)$  (so  $a \neq 0$ )
  - ▶ Use as hash-function  $h(k) = ((ak + b) \bmod p) \bmod M$
- $h(k)$  can be computed quickly.

# Carter-Wegman's universal hashing

**Better idea:** Choose hash-function randomly!

- Requires: all keys are in  $\{0, \dots, p-1\}$  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  $a, b$ 
    - ★  $b = \text{random}(p)$
    - ★  $a = 1 + \text{random}(p-1)$  (so  $a \neq 0$ )
  - ▶ Use as hash-function  $h(k) = ((ak + b) \bmod p) \bmod M$
- $h(k)$  can be computed quickly.

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 are multi-dimensional, such as strings?

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

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

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

To compute this in  $O(|w|)$  time without overflow, use Horner's rule and apply mod early. For example,  $h(\text{APPLE})$  is

$$\left( \left( \left( \left( \left( (65R + 80) \bmod M \right) R + 80 \right) \bmod M \right) R + 76 \right) \bmod M \right) R + 69 \right) \bmod M$$

# 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 (successor, select, rank etc.)

## Advantages of Hash Tables

- $O(1)$  operation cost (if hash-function random and load factor small)
- We can choose space-time tradeoff via load factor
- Cuckoo hashing achieves  $O(1)$  worst-case for search & delete