CS 240E – Data Structures and Data Management (Enriched)

Module 7: Dictionaries via Hashing

Tom Iagovet

Based on lecture notes by many previous cs240 instructors

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

Spring 2026

version *2026-06-30 10:16*

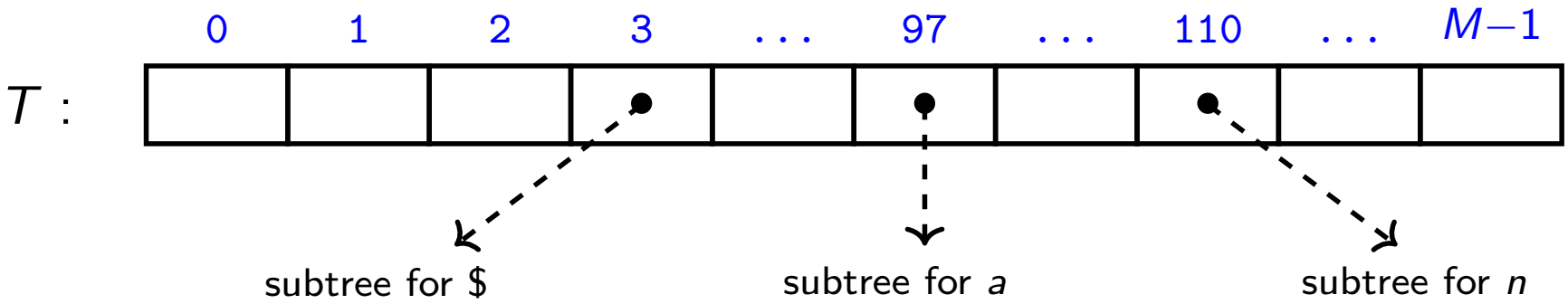
Outline

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 $= \{0, \ldots, 127\}$.

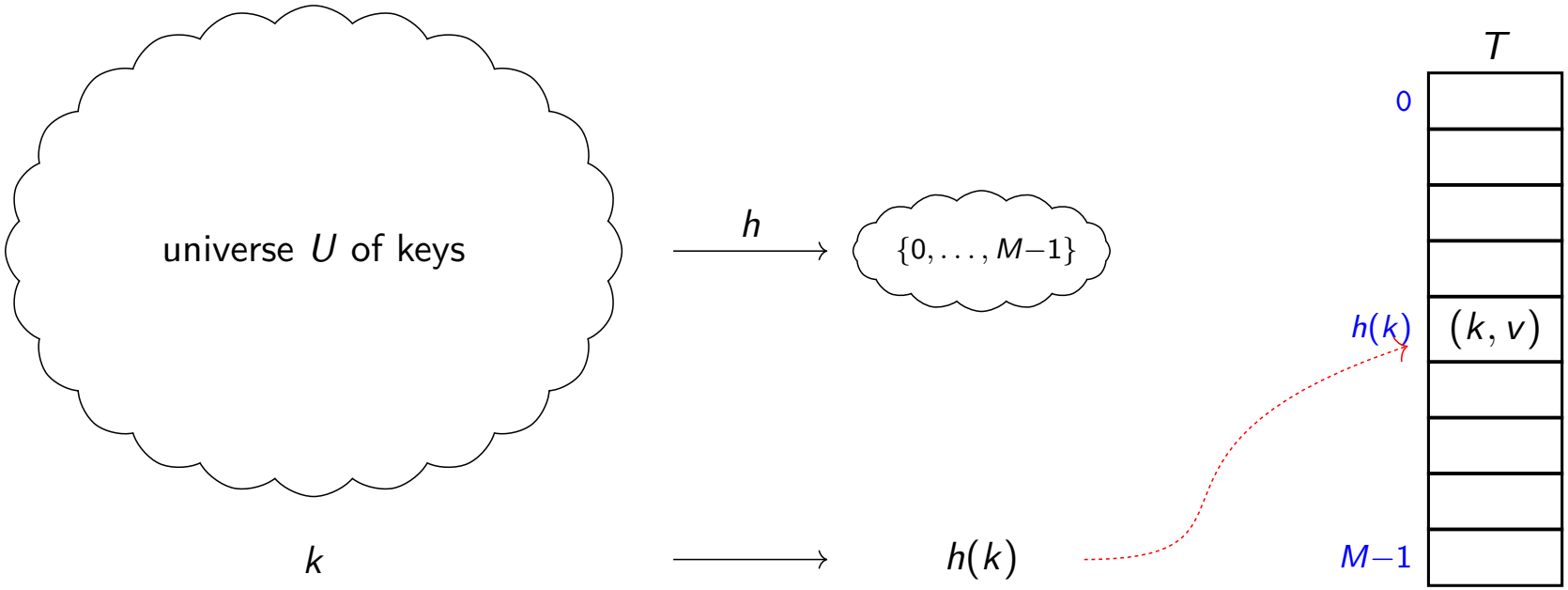
We can then implement a dictionary easily: Use an array T of size M that stores (k, v) via $T[k] \leftarrow v$.



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



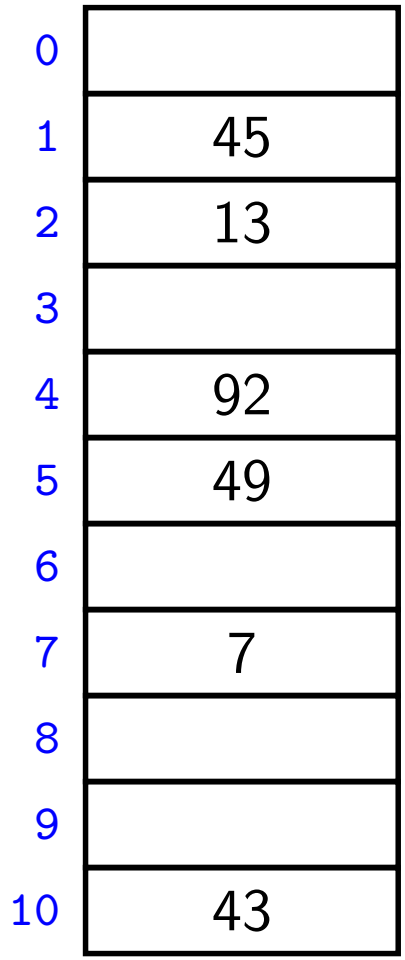
- **Assumption:** We know that all keys come from some **universe** U. (Typically U = non-negative integers, sometimes $|U|$ finite.)
- We pick a **table-size** M.
- We pick a **hash function** $h : U \to \{0, 1, \ldots, M-1\}$. (Commonly used: $h(k) = k \bmod M$. We will see other choices later.)

- 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

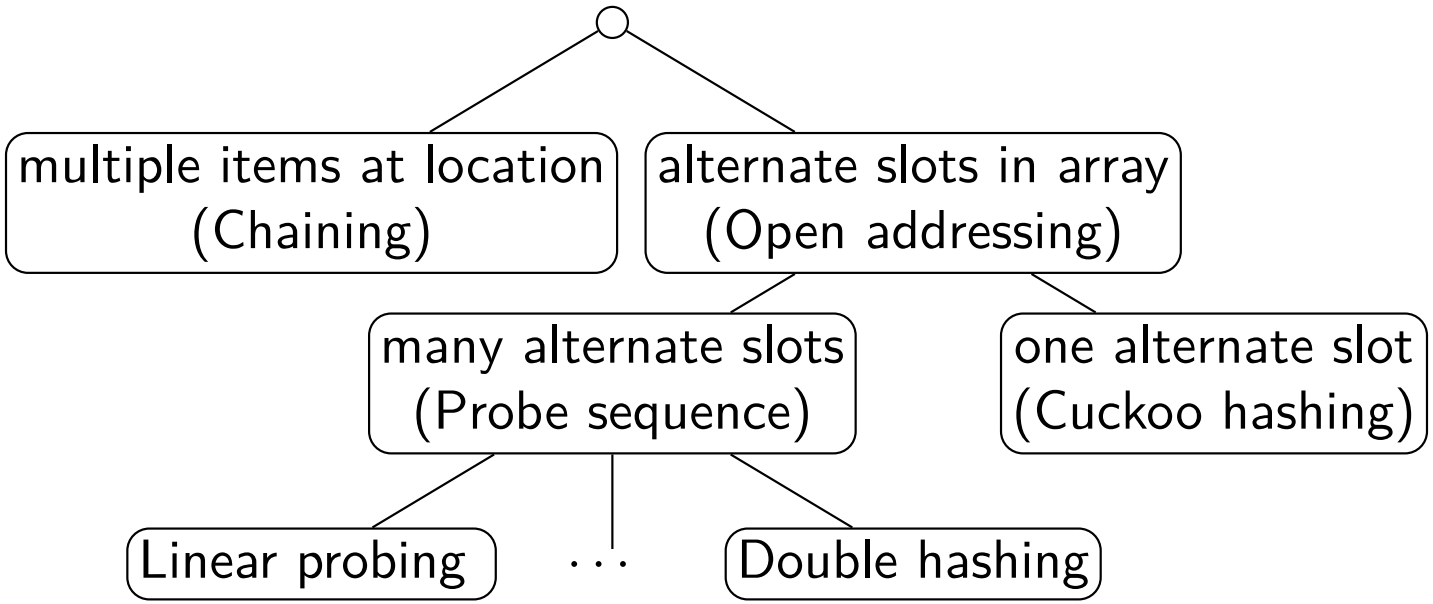
$U = \mathbb{N}$, $M = 11$, $h(k) = k \bmod 11$.

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:

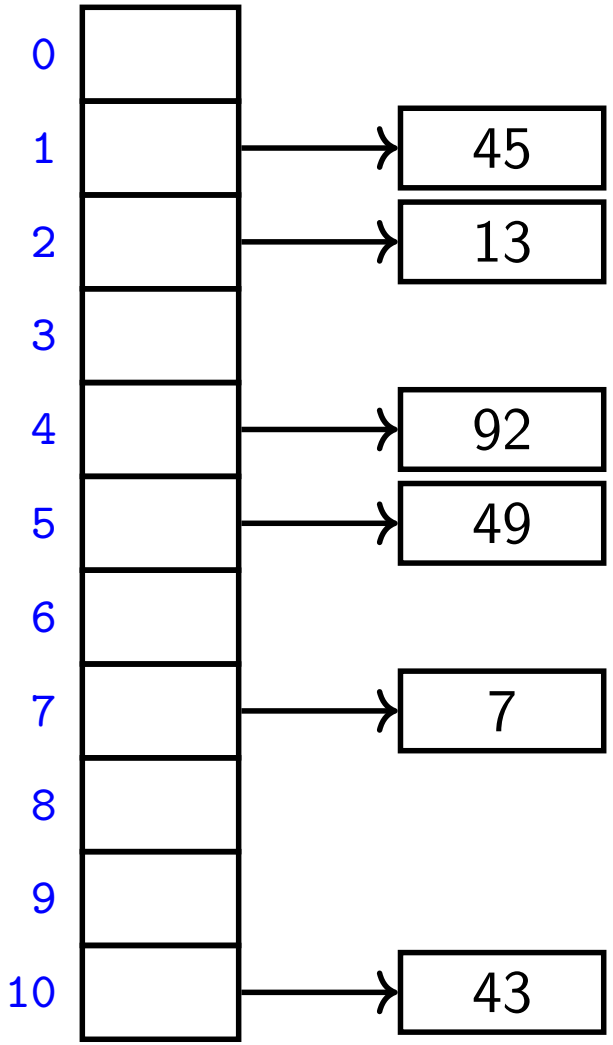


Hashing with chaining

Idea: **Bucket** $T[h(k)]$ holds multiple items (e.g. unsorted list with MTF).

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

insert(41)
insert(46)
insert(79)



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 $\frac{n}{M} =: \alpha$.

  (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 and 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.
- 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.)

Hashing with chaining: Analysis

UHA implies that the distribution of keys is unimportant.

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

- **Similar:** Two keys collide with probability $\frac{1}{M}$.

Back to analysis of chaining: How big is the bucket of key k?

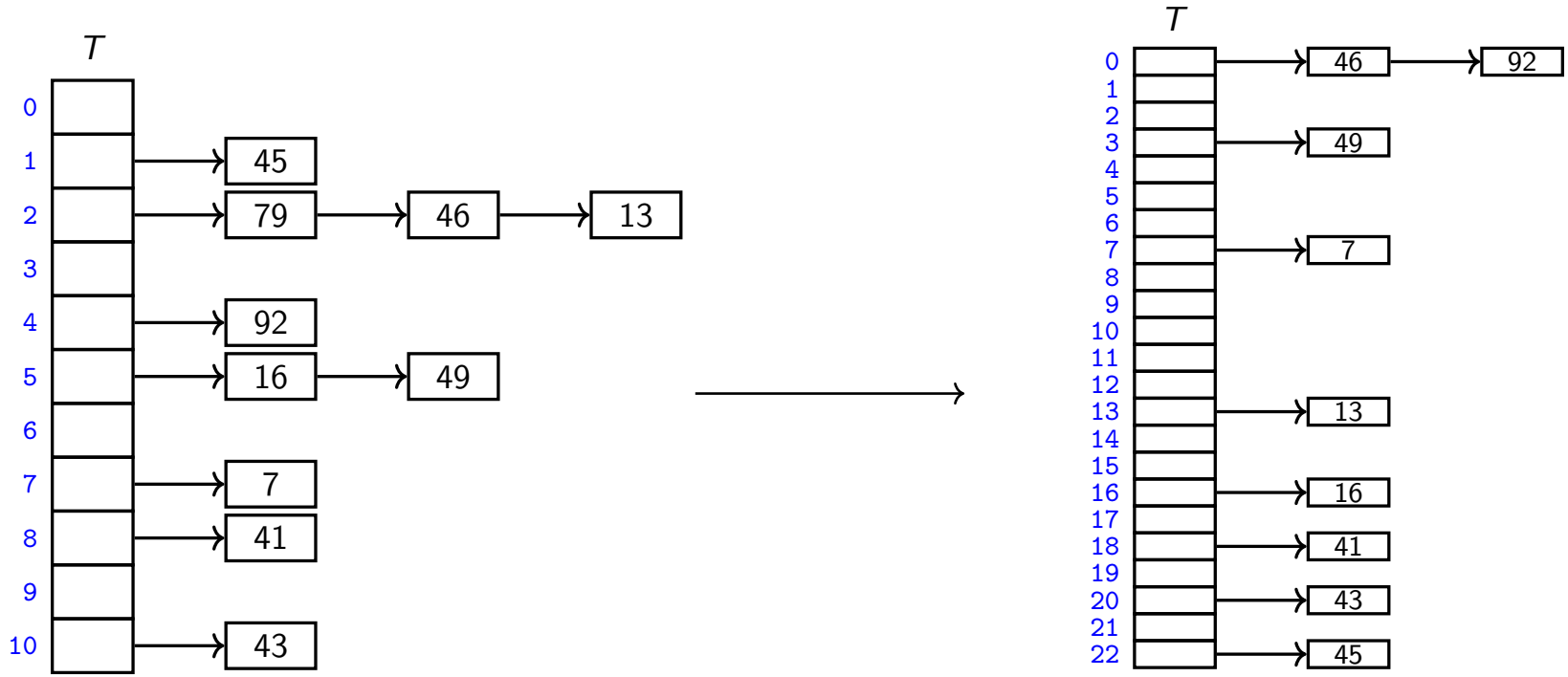
- If key k is *not* in dictionary:
  - n keys hash to this bucket with probability $\frac{1}{M}$ each
  - So bucket $T[h(k)]$ has expected length $\frac{n}{M} = \alpha$.
- If key k *is* in dictionary (e.g. during *delete*):
  - Key k is definitely in this bucket
  - Each of the other $n-1$ keys collides with probability $\frac{1}{M}$
  - So bucket $T[h(k)]$ has expected length $1 + \frac{n-1}{M} \leq 1 + \alpha$
- 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 α

(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 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 $\alpha \in \Theta(1)$ 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.
- If space is critical: also re-hash when α 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 $\Theta(n)$

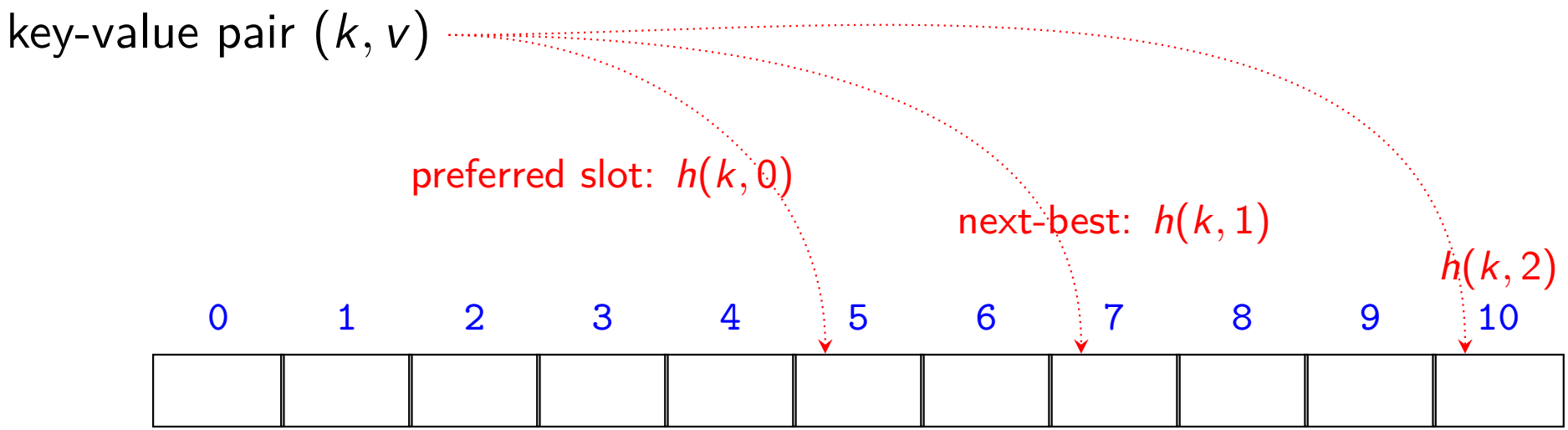
(assuming uniform hashing and $\alpha \in \Theta(1)$ throughout)

Theoretically perfect, but too slow in practice.

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: $\langle h(k,0), h(k,1), h(k,2), \ldots h(k,M-1)\rangle$ until an empty spot is found.

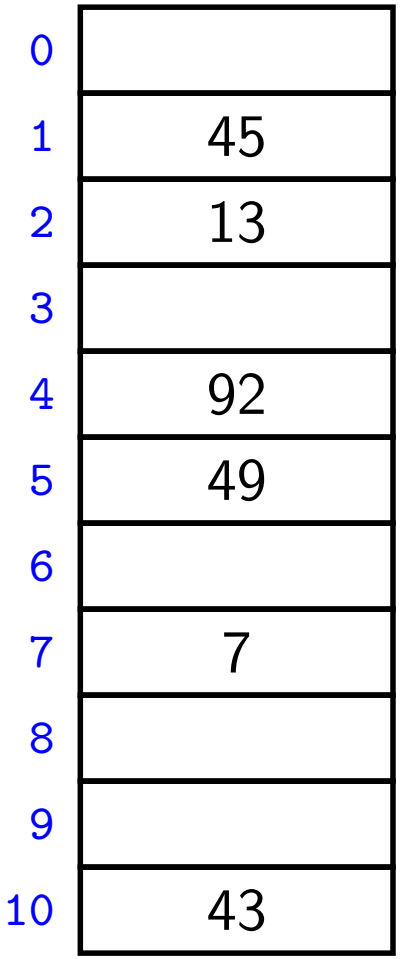


Simplest method for open addressing: *linear probing*
$h(k,j) = (h(k) + j) \bmod M$, for some hash function h.

Hashing with linear probing: Example

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

insert(41)
insert(84)
insert(20)



Hashing with probe sequences: Operations

Use *lazy deletion* (cannot handle updates after *delete* efficiently).

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 $\mathbb{P}(h_0(k) = i)$ and $\mathbb{P}(h_1(k) = j)$ are independent.
- Using two modular hash-functions often leads to dependencies.
- Better idea: Use *multiplication method* for second hash function:
  - Fix some floating-point number A with $0 < A < 1$

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

$$\text{integer in } [0, M)$$

  - 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

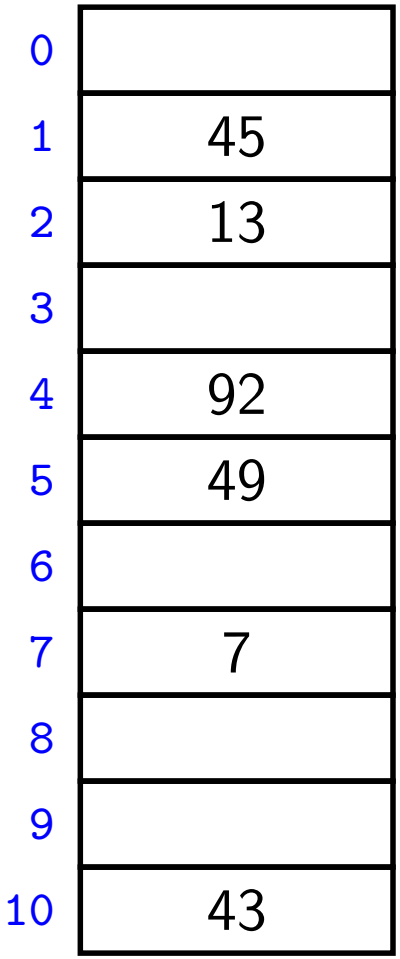
- So choose M prime.
- Modify standard hash-functions to ensure $h_1(k) \neq 0$
  E.g. modified multiplication method: $h(k) = 1 + \lfloor (M-1)(kA - \lfloor kA \rfloor) \rfloor$

Double hashing: Example

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

insert(41)
insert(194)

$h_0(41) = 8$
$h_0(194) = 7$
$h_1(194) = 9$



Analysis of uniform probing

- Analyzing linear probing and double hashing is difficult (no details).
- Instead, analyze an idealized setup: **uniform probing**

$$\mathbb{P}(\text{slot } i \text{ is occupied}) = \frac{n}{M}$$

- As before, $\alpha = \frac{n}{M}$ is the load factor.

Claim: The expected run-time of *search* is $O(\frac{1}{1-\alpha})$.

Similar: The exp.-luck avg.-instance run-time of *delete* is $O(\frac{1}{\alpha}) \ln\left(\frac{1}{1-\alpha}\right)$.

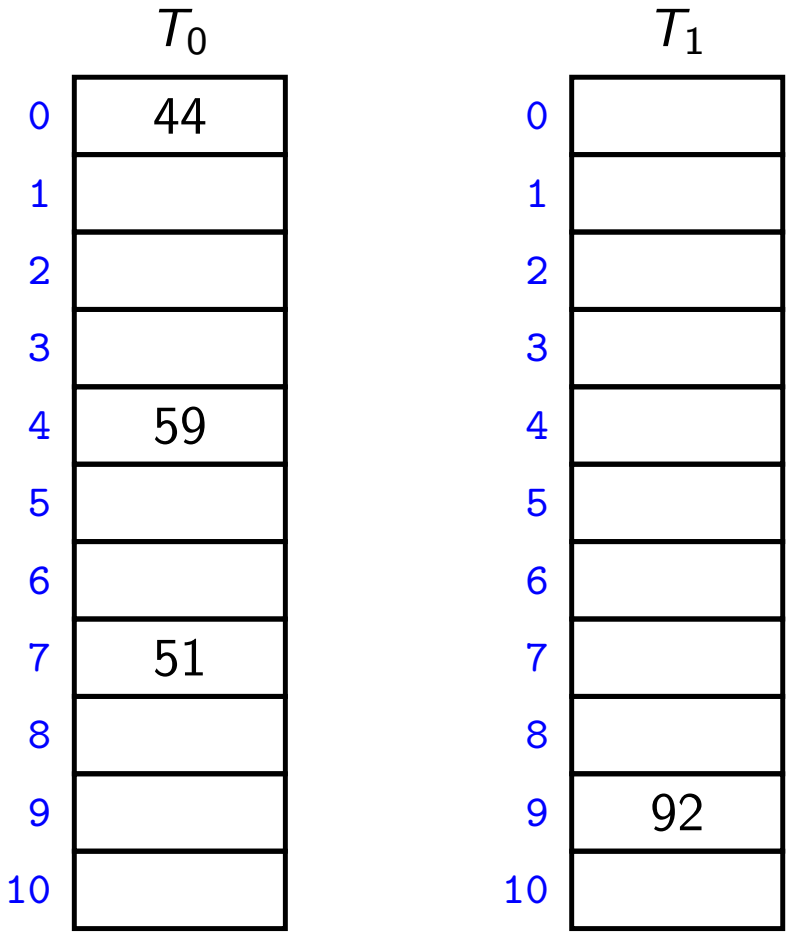
Summary: No lists, and all operations have $O(1)$ expected run-time if α is kept small enough.

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)

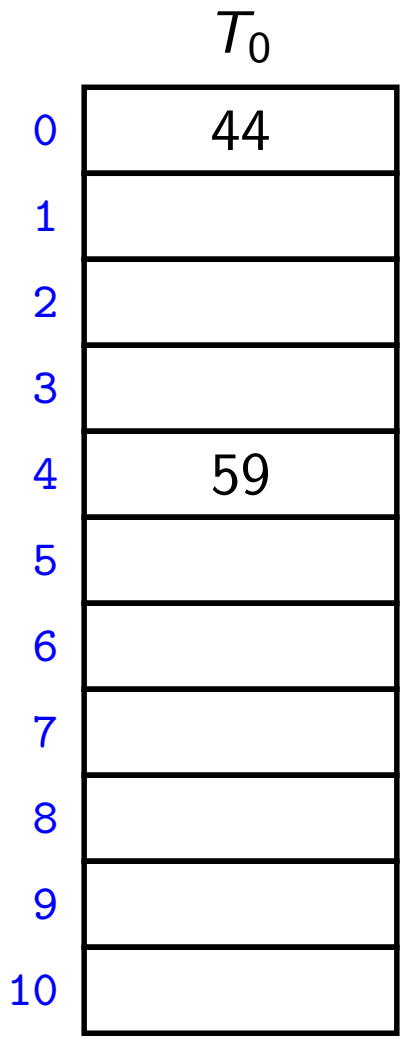
1. $(k_{insert}, v_{insert}) \leftarrow$ new key-value pair with (k, v)
2. $i \leftarrow 0$
3. **do** at most $2n$ times:
4. $\quad (k_{evict}, v_{evict}) \leftarrow T_i[h_i(k_{insert})]$ `// save old KVP`
5. $\quad T_i[h_i(k_{insert})] \leftarrow (k_{insert}, v_{insert})$ `// put in new KVP`
6. $\quad$ **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`

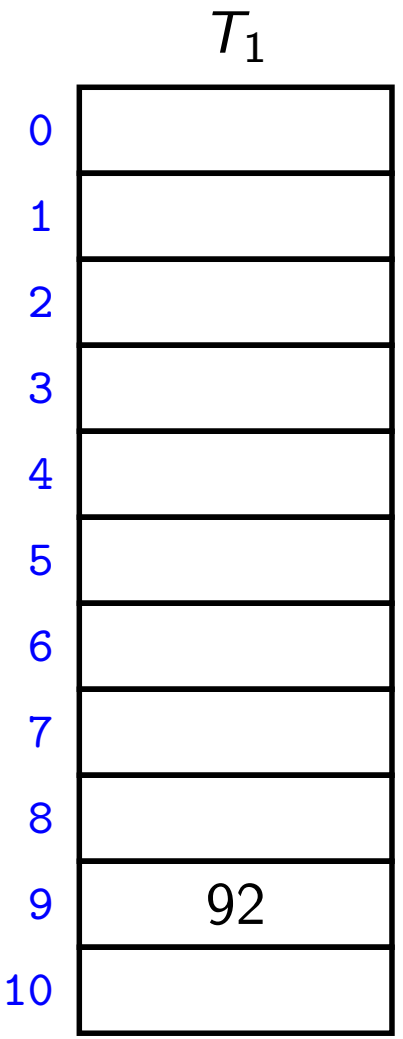
Cuckoo hashing insertion: Example

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

insert(51)
insert(95)
insert(26)

k	h_0	h_1
51	7	5
95	7	7
26	4	0
59	4	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 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).

Hashing with open addressing: Summary

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

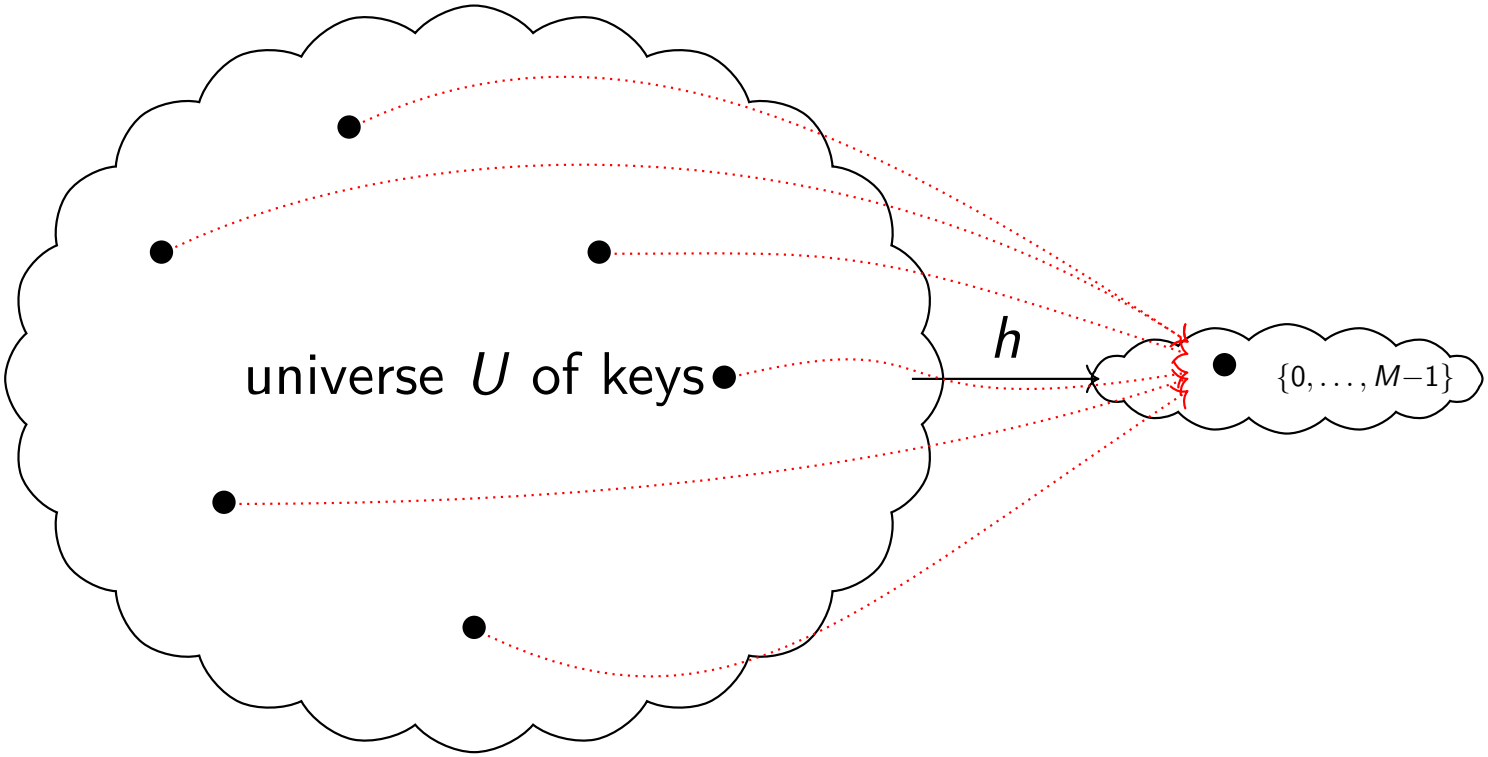
But for any hash-function the worst-case run-time is $\Theta(n)$ for *insert*.

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

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, and
   - depends on all parts of the key.
2. Randomized: Choose randomly among a limited set of functions.
   - But aim for $\mathbb{P}(\text{two keys collide}) = \frac{1}{M}$ w.r.t. key-distribution.
   - This is enough to prove the expected run-time bounds for chaining

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 $\varphi = \frac{\sqrt{5}-1}{2} \approx 0.618033988749.....$
  - How many bits should we use?
  - Won't the computation be terribly slow?

Multiplication method

- $h(k) = \lfloor M(kA - \lfloor kA \rfloor) \rfloor$. We can use $M = 2^\ell$ for some $\ell > 0$.
- So $h(k) =$ first ℓ bits of fractional part of $A \cdot k$.

(So computing $h(k)$ is a multiplication plus a bit-shift. This may actually be faster than taking "modulo a prime number".)

Claim: Only $\approx \log|U| + \ell$ bits of A influence $h(k)$.

- Write $A = 0.a_1 a_2 a_3 \ldots$ and $k = b_1 b_2 \ldots b_{\log|U|}$ (in base 2)
- Which bits of A matter? (Here $\ell = 3$, $\log|U| = 6$):

		(leading bits)						(fractional part)							
$A \cdot k$	$=$	0	0	0	..	0	0								
	$+a_1 \cdot$	0	b_1	b_2	b_3	..	b_5	b_6							
	$+a_2 \cdot$	0	0	b_1	b_2	b_3	..	b_5	b_6						
	$+a_3 \cdot$	0	0	0	b_1	b_2	b_3	..	b_5	b_6					$\log\|U\|$
	$\vdots$					$\ddots$	$\ddots$	$\ddots$	$\ddots$	$\ddots$	$\ddots$				
	$+a_5 \cdot$	0	0	0	0	0	b_1	b_2	b_3	..	b_5	b_6			
	$+a_6 \cdot$	0	0	0	0	0	0	b_1	b_2	b_3	..	b_5	b_6		ℓ
	$+a_7 \cdot$	0	0	0	0	0	0	0	b_1	b_2	b_3	..	b_5	b_6	
	$+a_8 \cdot$	0	0	0	0	0	0	0	0	b_1	b_2	b_3	..	b_5	b_6
	$\vdots$							$\leftarrow h(k) \rightarrow$							

Randomly chosen hash-functions

- Ideally we would choose randomly uniformly among all hash functions. But this is impossible.

- **Idea:** Fix a family $\mathcal{H}$ of hash-functions that are easy to compute. Then choose uniformly among them.
- Example:
  - $U = \{0, 1, 2, 3, 4\}$, $M = 3$
  - $h_b(k) = ((k + b) \bmod 5) \bmod 3$
  - $\mathcal{H} = \{h_b : b \in \{0, 1, 2, 3, 4\}\}$
  - Choose b randomly to get hash-function

$\mathcal{H}$	keys				
	0	1	2	3	4
h_0	0	1	2	0	1
h_1	1	2	0	1	2
h_2	2	0	1	2	0
h_3	0	1	2	0	1
h_4	1	2	0	1	2

- But how do we measure whether these are "good"?

Universal hash-functions

- For analysis, we needed *uniform hash-values*: $\mathbb{P}(h(k) = i) = \frac{1}{M}$
- But this is *not* good enough.

$\mathcal{H}$	keys				
	0	1	2	3	...
h_0	0	0	0	0	...
h_1	1	1	1	1	...
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
h_{M-1}	$M-1$	$M-1$	$M-1$	$M-1$	...

- These hash-functions are terrible! (All keys want the same slot.)
- But $\mathbb{P}(h(k) = i) = \frac{1}{M}$ for all i, k
- Problem: hash-values not independent

- We also need a small probability of collisions:

$$\mathbb{P}\Big(h(k) = h(k')\Big) \leq \frac{1}{M} \quad \text{for any two keys } k \neq k'$$

- This is enough for analyzing hashing with chaining as before.

Carter-Wegman's universal hashing

- 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 $\boxed{h_{a,b}(k) = \big((ak+b) \bmod p\big) \bmod M}$

  (So we choose randomly from $\mathcal{H}_{CW} := \{h_{a,b} : 1 \leq a < p, 0 \leq b < p\}$.)
- Observe: hash-value can be computed in constant time.

Goal: $\mathbb{P}(h(k) = h(k')) \leq \frac{1}{M}$, i.e., small collision-probability.

So hashing with chaining and randomly chosen hash function in $\mathcal{H}_{CW}$ has expected amortized run-time $O(1)$.

Carter-Wegman hash functions: Details

- Recall: $U \subseteq \{0, \ldots, p-1\} =: \mathbb{Z}_p$ (integers modulo prime p)
- $h_{a,b}(k) = \Big(\underbrace{(a \cdot k + b) \bmod p}_{f_{a,b}(k)} \Big) \bmod M \quad : a, b \in \mathbb{Z}_p, a \neq 0$

(where $M < p$)

Example: $(p = 5, M = 3)$:

	keys				
	0	1	2	3	4
$f_{1,0}$	0	1	2	3	4
$f_{2,0}$	0	2	4	1	3
$f_{1,2}$	2	3	4	0	1
$f_{2,1}$	1	3	0	2	4
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

	keys				
	0	1	2	3	4
$h_{1,0}$	0	1	2	0	1
$h_{2,0}$	0	2	1	1	0
$h_{1,2}$	2	0	1	0	1
$h_{2,1}$	1	0	0	2	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Observe: $f_{a,b}$ is a permutation of $\mathbb{Z}_p$ since p is a prime.

Carter-Wegman hash functions: Universality

Goal: Bound $\mathbb{P}(h_{a,b}(k) = h_{a,b}(k'))$ for any pair $k \neq k'$ of keys in $\mathbb{Z}_p$.

(Here the probability is over the random choice of $a, b \in \mathbb{Z}_p$, $a \neq 0$)

Equivalent: Bound $\mathbb{P}\Big(f_{a,b}(k) \bmod M = f_{a,b}(k') \bmod M\Big)$

Equivalent: Bound $\mathbb{P}\Big(\underbrace{f_{a,b}(k)}_{x}$ and $\underbrace{f_{a,b}(k')}_{x'}$ form a **bad pair**$\Big)$

Bad pair: $x, x' \in \mathbb{Z}_p$, $x \neq x'$, but $x \bmod M = x' \bmod M$

Equivalent: Bound $\displaystyle\sum_{x,x' \text{ bad pair}} \mathbb{P}\Big(\underbrace{f_{a,b}(k) = x \text{ and } f_{a,b}(k') = x'}_{(*)}\Big)$

- **Claim 1:** (*) uniquely determines a and b (so probability is $\frac{1}{p-1} \cdot \frac{1}{p}$)
- **Claim 2:** There are at most $\frac{p(p-1)}{M}$ bad pairs.

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.

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

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

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

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

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