Algorithms and Data Structures

Lecture 5

Hash Tables 2:
Hash Functions, Universal Hashing, Rehash, Cuckoo Hashing

Fabian Kuhn

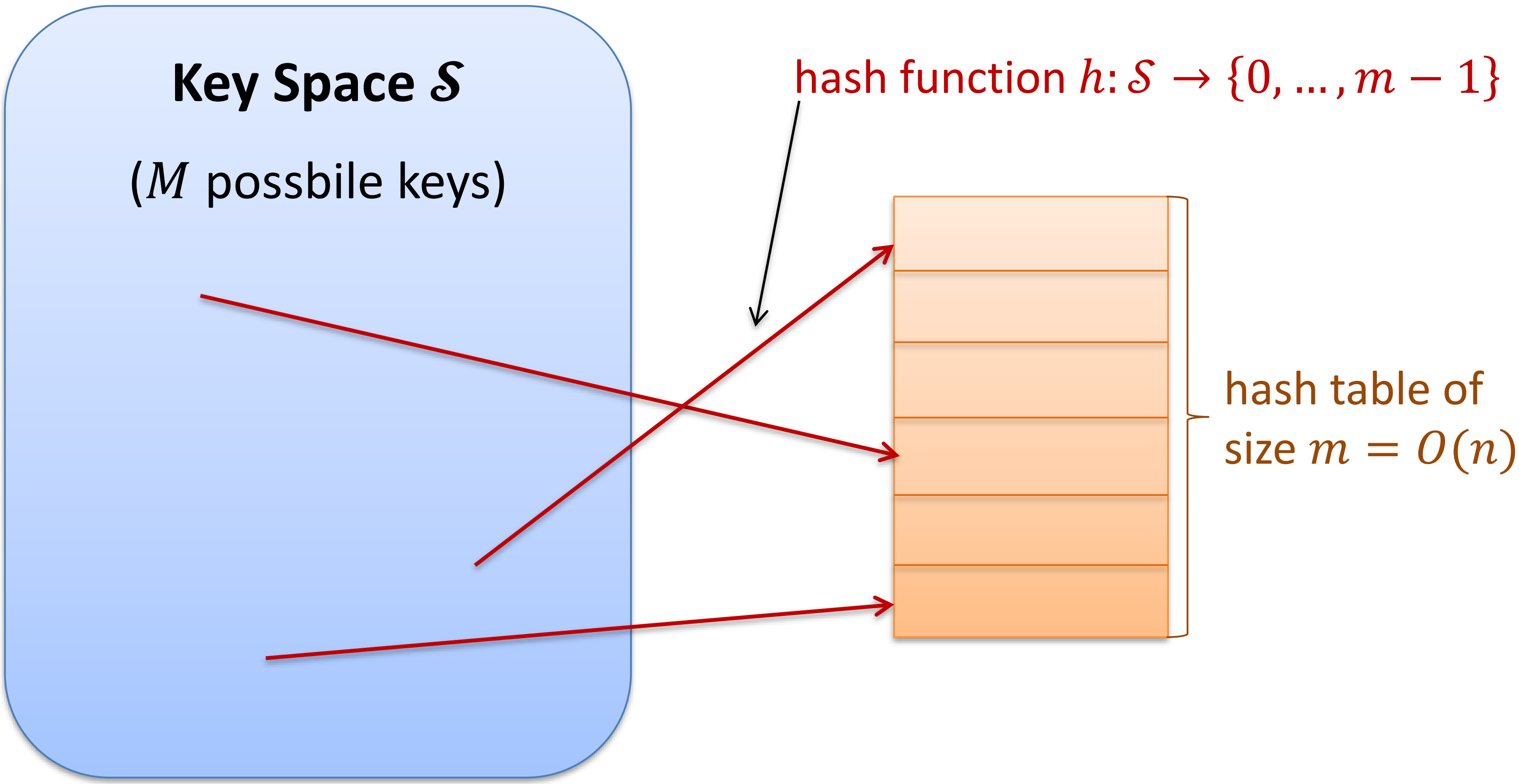
Algorithms and Complexity

UNI FREIBURG

Hash Tables

Implements a Dictionary

- Manage a set of (key, value) pairs
- Main operations: insert, find, delete



Hash Tables

We have seen so far:

efficient method to implement a dictionary

- All operations typically have running time $O(1)$
  - If the hash functions are sufficiently random and can be evaluated in time $O(1)$.
  - The worst-case running time is somewhat larger, in every application of hash tables, there will be some more expensive operations.

We will now see:

- How to choose a good hash function?
- What to do if the hash table becomes too small?
- How to implement hashing such that find always requires time $O(1)$.

Good Hash Functions

How to choose a good hash functions?

What properties should a good hash function satisfy?

- In principle, it should have the same properties as a random function:
  - Mapping is uniformly random (all hash values appear equally often)
  - Mapping of different keys is independent
    (not clear what exactly this means for a deterministic function)
- Usually, these conditions cannot be verified.
- If something about the distribution of key values is known, this knowledge can potentially be used.
- Luckily there are simple heuristics that work well in practice.

Division Method

Choose hash function as

$$h(x) = x \bmod m$$

- All values between 0 and $m - 1$ appear equally often
  - as far as this is possible

Advantages:

- Very simple function
- A single division → can be computed very fast
- Often works quite well, as long as m is chosen carefully...

Remarks:

- If the keys are not integers, one can interpret the bit sequences representing the keys as integers.
- Consecutive keys are mapped to consecutive hash values.

Division Method

UNI FREIBURG

Choose hash function as

$$\boldsymbol{h(x) = x \bmod m}$$

Choice of Divisor m

- $h(x)$ could be computed particularly fast if $m = 2^k$
- This is however no good choice because then the hash value is just the last k bits of the key!
  - The hash value should depend on all the bits.
- The best is to choose m as a prime number.
- A prime number m for which $m = 2^k - 1$ is also not ideal.
- Best: prime m that is not too close to a power of 2.

Multiplication Method

Choose hash function as

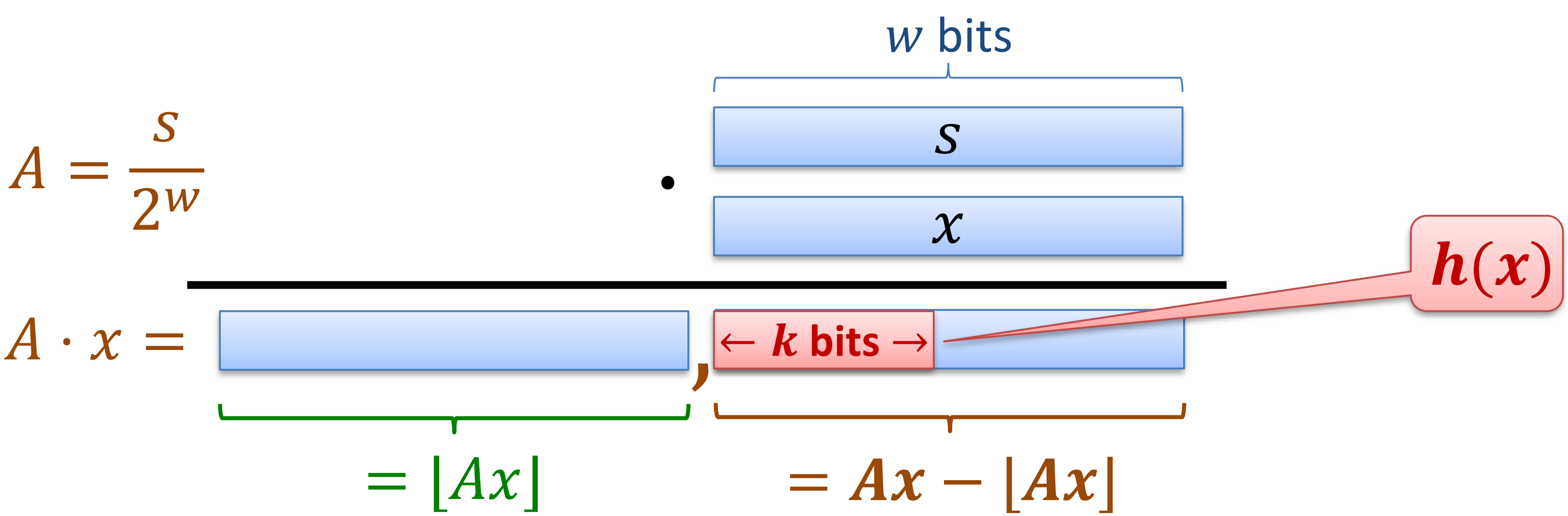
$$0 \le Ax - \lfloor Ax \rfloor < 1$$

$$\boldsymbol{h(x) = \lfloor m \cdot (Ax - \lfloor Ax \rfloor) \rfloor}$$

- A is a constant between 0 and 1

Remarks

- Here, one can choose $m = 2^k$ (for an integer k)
- If integers are values 0 to $2^w - 1$, one typically picks an integer $s \in \{1, \dots, 2^w - 1\}$ and defines $A = s \cdot 2^{-w}$



Multiplication Method

Choose hash function as

$$h(x) = \lfloor m \cdot (Ax - \lfloor Ax \rfloor) \rfloor$$

- A is a constant between 0 and 1

Remarks

- Here, one can choose $m = 2^k$ (for an integer k)
- If integers are values 0 to $2^w - 1$, one typically picks an integer $s \in \{1, \dots, 2^w - 1\}$ and defines $A = s \cdot 2^{-w}$

  – In principle every A works, in [Knuth; The Art of Comp. Progr. Vol. 3] it is suggested to use

$$A \approx \frac{\sqrt{5} - 1}{2} = 0.6180339887 \dots$$

Random Hash Functions

If h is chosen randomly among all possible hash functions:

$$\forall x_1, x_2 : \Pr\big(h(x_1) = h(x_2)\big) = \frac{1}{m}$$

and many other good properties …

Problem:

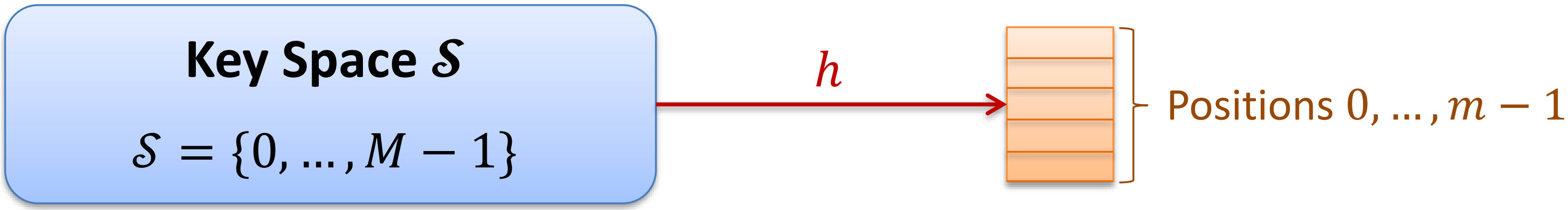
- Such a function cannot be represented and implemented efficiently.
  - One essentially needs a table with an entry for each possible key

Idea:

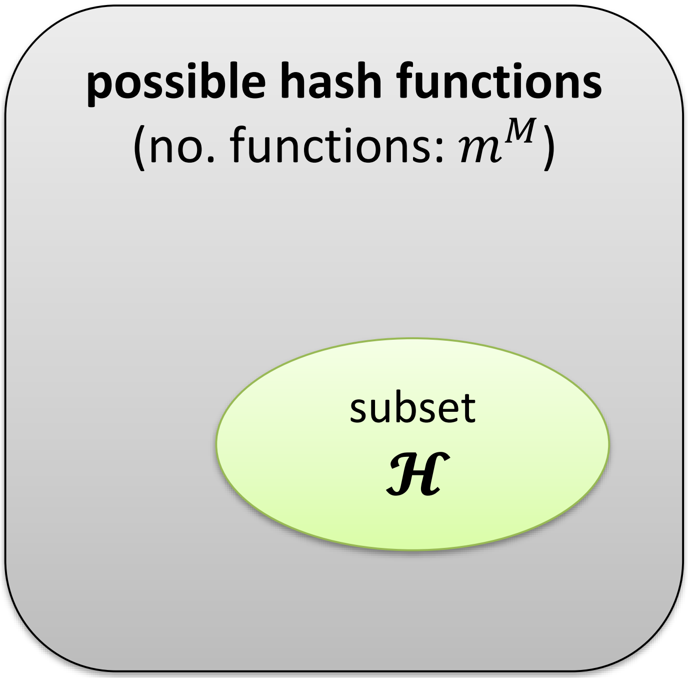
- Choose a function at random from a smaller space
  - E.g., use the multiplication method $h(x) = \lfloor m \cdot (Ax - \lfloor Ax \rfloor) \rfloor$ with a random parameter A
- Not quite as good as a uniformly random hash function, but if it is done correctly, the ideas works → **universal hashing**

Universal Hashing : Idea

Hash functions: $h : \mathcal{S} \rightarrow \{0, \dots, m-1\}$



Space of all possible hash functions



Choose $\mathcal{H}$ such that:

- $|\mathcal{H}|$ is not too large and the functions in $\mathcal{H}$ are easy to implement
- A random function h from $\mathcal{H}$ behaves similarly to a uniformly random function
- In particular regarding the collision prob.:

$$\forall x_1, x_2 : \Pr\big(h(x_1) = h(x_2)\big) \approx \frac{1}{m}$$

Universal Hashing : Definition

Definition:

- Let $\mathcal{S}$ be the set of possible keys and m be the size of the hash table
- Let $\mathcal{H}$ be a set of hash functions $\mathcal{S} \to \{0, \dots, m-1\}$

The set $\mathcal{H}$ is called c-universal if

$$\forall \boldsymbol{x}, \boldsymbol{y} \in \boldsymbol{\mathcal{S}} : \boldsymbol{x} \neq \boldsymbol{y} \Longrightarrow |\{\boldsymbol{h} \in \boldsymbol{\mathcal{H}} : \boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{h}(\boldsymbol{y})\}| \leq \boldsymbol{c} \cdot \frac{|\boldsymbol{\mathcal{H}}|}{\boldsymbol{m}}.$$

- With other words, if h is chosen at random from $\mathcal{H}$, we have

$$\forall x, y \in S : x \neq y \Longrightarrow \Pr\big(h(x) = h(y)\big) \leq \frac{c}{m}$$

- **Remark:**
  The set $\mathcal{H}$ of all m^M possible hash functions is 1-universal.

Universal Hashing : List Lengths

Theorem:

- Let $\mathcal{H}$ be a c-universal set of hash functions $\mathcal{S} \to \{0, \ldots, m-1\}$
- Let $X \subset \mathcal{S}$ be an arbitrary set of keys
- Let $h \in \mathcal{H}$ be a random hash function from the set $\mathcal{H}$
- For a given $x \in X$, let

$$B_x \coloneqq \{y \in X : h(y) = h(x)\}$$

- In expectation, B_x has size $< 1 + c \cdot \frac{|X|}{m}$

Therefore:

- In expectation, all lists are short!

Universal Hashing : Example I

The set $\mathcal{H}$ is called c-universal if

$$\forall x, y \in \mathcal{S} : x \neq y \Longrightarrow |\{h \in \mathcal{H} : h(x) = h(y)\}| \leq c \cdot \frac{|\mathcal{H}|}{m}.$$

Negative Example:

- Parametrized variant of the division method

$$\mathcal{H} = \{h : x \to a \cdot x \mod m \quad \text{for } a \in \{1, \dots, M-1\}\}$$

- Counterexample: choose an arbitrary x and choose $y = x + m$
  - $h(x) = a \cdot x \mod m$
  - $h(y) = a \cdot (x + m) \mod m = (a \cdot x + a \cdot m) \mod m = a \cdot x \mod m$

Universal Hashing : Example II

The set $\mathcal{H}$ is called c-universal if

$$\forall x, y \in \mathcal{S} : x \neq y \Longrightarrow |\{h \in \mathcal{H} : h(x) = h(y)\}| \leq c \cdot \frac{|\mathcal{H}|}{m}.$$

Positive Example 1:

- m arbitrary, p: prime such that $p > M$

$$\mathcal{H} = \{h : x \to ((a \cdot x + b) \bmod p) \bmod m \quad \text{for } a, b \in \mathcal{S}, a \neq 0\}$$

- The set is c-universal für $c \approx 1$ if $p \approx M$
- For x, y, we have $h(x) = h(y)$, if for some $i \in \mathbb{Z}$:

$$(ax + b) \bmod p = (ay + b) \bmod p + i \cdot m$$

$$a \equiv i \cdot m \cdot (x - y)^{-1} \pmod{p}$$

holds for at most $2 \cdot \left\lfloor \frac{p-1}{m} \right\rfloor + 1$ diff. values of i

- For every x and y and for every b, for each possible value of i, there is only one value of a, for which x and y collide.

Universal Hashing : Example III

The set $\mathcal{H}$ is called c-universal if

$$\forall x, y \in \mathcal{S} : x \neq y \Longrightarrow |\{h \in \mathcal{H} : h(x) = h(y)\}| \leq c \cdot \frac{|\mathcal{H}|}{m}.$$

Positive Example 2:

- m prime, $k = \lfloor \log_m M \rfloor$, parameter $a \in \mathcal{S} = \{0, \dots, M-1\}$
- Consider parameter a and key x in basis-m representation:

$$a = a_0 + a_1 \cdot m + a_2 \cdot m^2 + \dots + a_k \cdot m^k$$
$$x = x_0 + x_1 \cdot m + x_2 \cdot m^2 + \dots + x_k \cdot m^k$$

$a_i, x_i \in \{0, \dots, m-1\}$

$$\mathcal{H} = \left\{ h : x \to \left(\sum_{i=0}^{k} a_i \cdot x_i \right) \bmod m \ \text{ for } a_i \in \{0, \dots, m-1\} \right\}$$

- The set $\mathcal{H}$ is 1-universal

Universal Hashing : Summary

- If the hash function is chosen at random from a universal set of hash functions, the collision probability for two keys x and y is equal as for a random hash function.
- There are simple and efficient constructions of universal sets of hash functions.

One can take this further:

- Pairwise independent set of hash functions

$$\forall x, y \in \mathcal{S}, \forall a, b \in \mathbb{Z}_m: \Pr(h(x) = a \ \land \ h(y) = b) = \frac{1}{m^2}$$

  – A random function from such a set behaves exactly the same as a random function for every pair of keys x, y (not just regarding collisions)

- k-independent set of hash functions
  – A random function from such a set behaves exactly the same as a random hash function for every set of k different keys.

Rehash

Remember:

- Load of a hash table: $\alpha = n/m$

What if a hash table becomes too full?

- Open Addressing:
  - $\alpha > 1$ impossible, for $\alpha \to 1$ very inefficient
  - If one inserts and deletes a lot, the table also becomes inefficient (because of the deleted marks)
- Chaining: Complexity grows linearly with α

What it the chosen hash function behaves badly?

Rehash:

- Create a new, larger hash table, choose a new hash function h'.
- Insert all existing (key, value) pairs.

Cost of Rehash

A rehash is expensive!

Cost (time):

- $\Theta(m + n)$: grows linearly in the number of inserted values and in the length of the old hash table
  - typically, this is just $\Theta(n)$

- **If done correctly, a rehash is rarely necessary:**
  - good hash function (e.g., from a universal set)
  - good choice of table sizes:
    with each **rehash**, the **table size** should be roughly **doubled**
    old size $m \implies$ new size $\approx 2m$
  - With doubling, one gets constant time per hash table operation on average
    → amortisierte Analyse

Cost of Rehash

UNI FREIBURG

Analysis Doubling Strategy

- We make a few simplifying assumptions:
  - Up to load α_0 (e.g., $\alpha_0 = {}^1/_2$) all hash table operations cost $\leq c$.
  - At load α_0, we double the table size:
    old size m, new size $2m$, cost $\leq c \cdot m$.
  - At the beginning, the table has size $m_0 \in O(1)$.
  - The table size is never decreased…

- How large is the cost for rehashing, compared to the total cost of all other operations?

Cost of Rehash

Overall Cost

- We assume that the table size is $m = m_0 \cdot 2^k$ for $k \geq 1$
  – i.e., up to now, we have done $k \geq 1$ rehash steps
  – remark: for $k = 0$ the rehash cost is still 0.
- The overall rehash cost is

$$\leq \sum_{i=0}^{k-1} c \cdot m_0 \cdot 2^i = c \cdot m_0 \cdot \left(2^k - 1\right) \leq c \cdot m$$

- Overall cost for the remaining operations
  – For the rehash from size $m/2$ to size m we had $\geq \alpha_0 \cdot m/2$ entries in the table.
  – Number of hash table operations (without rehash)

$$\geq \frac{\alpha_0}{2} \cdot m$$

Cost of Rehash

- The overall rehash cost is

$$\leq \sum_{i=0}^{k-1} c \cdot m_0 \cdot 2^i = c \cdot m_0 \cdot \left(2^k - 1\right) \leq c \cdot m$$

- Number of hash table operations:

$$\#\mathrm{OP} \geq \frac{\alpha_0}{2} \cdot m$$

- Average cost per operation

$$\frac{\#\mathbf{OP} \cdot \boldsymbol{c} + \mathbf{Rehash\_Kosten}}{\#\mathbf{OP}} \leq \boldsymbol{c} + \frac{\mathbf{2}\boldsymbol{c}}{\boldsymbol{\alpha}_\mathbf{0}} \in \boldsymbol{O}(\mathbf{1})$$

- On average, the cost per operation is constant
  - also for worst-case inputs (as long as the simplifying assumptions hold)
  - **average cost per operation = amortized cost per operation**

Amortized Analysis

UNI FREIBURG

Algorithm analysis so far:

- worst case, best case, average case

Now additionaly **amortized worst case**:

- n operations $o_1, \dots, o_n$ on some data structure, t_i: cost of o_i
- Costs can be very different from each other (z.B. $t_i \in [1, c \cdot i]$)
- Amortized cost per operation

$$\frac{T}{n}, \qquad \text{where } T = \sum_{i=1}^{n} t_i$$

- **Amortized cost:** Average cost per operation in a worst-case execution
  - amortized worst case $\neq$ average case!

- More on this in the algorithm theory lecture

Amortized Analysis Rehash

- If one only increases the table size and assumes that for small load, hash table operations require time $O(1)$, the amortized cost (time) per operation is $O(1)$.
- Analysis also works for a random hash function from a universal set of hash functions (with high probability)
  - Then, for small load, hash table operations with high probability have amortized cost $O(1)$.
- Analysis can be adapted for rehashs for decreasing the table size
  - And also for cases where a rehash is necessary because of a lot of delete operations (and the resulting deleted marks)
- In a similar way, one can build dynamic-size arrays from fixed-size arrays
  - All array operations have $O(1)$ amortized running time.
  - ADT only allows increasing/decreasing size in 1-element steps at the end.

Cuckoo Hashing Idea

Hashing Summary:

- Efficient dictionary data structure
- Operations in expectation (usually) require $O(1)$ time.
- Hashing with separate chaining can be implemented such that insert always has running time $O(1)$.
- Can we also guarantee **running time $O(1)$ for find**?
  - if at the same time insert is only $O(1)$ time in expectation…

Cuckoo Hashing Idea:

- Open addressing
  - At each table position, there is only space for one entry.
- Two hash functions h_1 and h_2
- A key x is always stored at position $h_1(x)$ or $h_2(x)$
  - If both positions are occupied when inserting x, one has to reorganize…

Cuckoo Hashing

Inserting a key x:

- x is always inserted at position $h_1(x)$
- If there already is another key y at position $h_1(x)$:
  - Remove y from this position (thus the name cuckoo hashing)
  - y has to be inserted at its alternative position (if it was at pos. $h_1(y)$, it has to go to pos. $h_2(y)$, otherwise to pos. $h_1(y)$)
  - If there is already a key z at this position, remove z from there and place it at its alternative position
  - And so on …

Find / Delete:

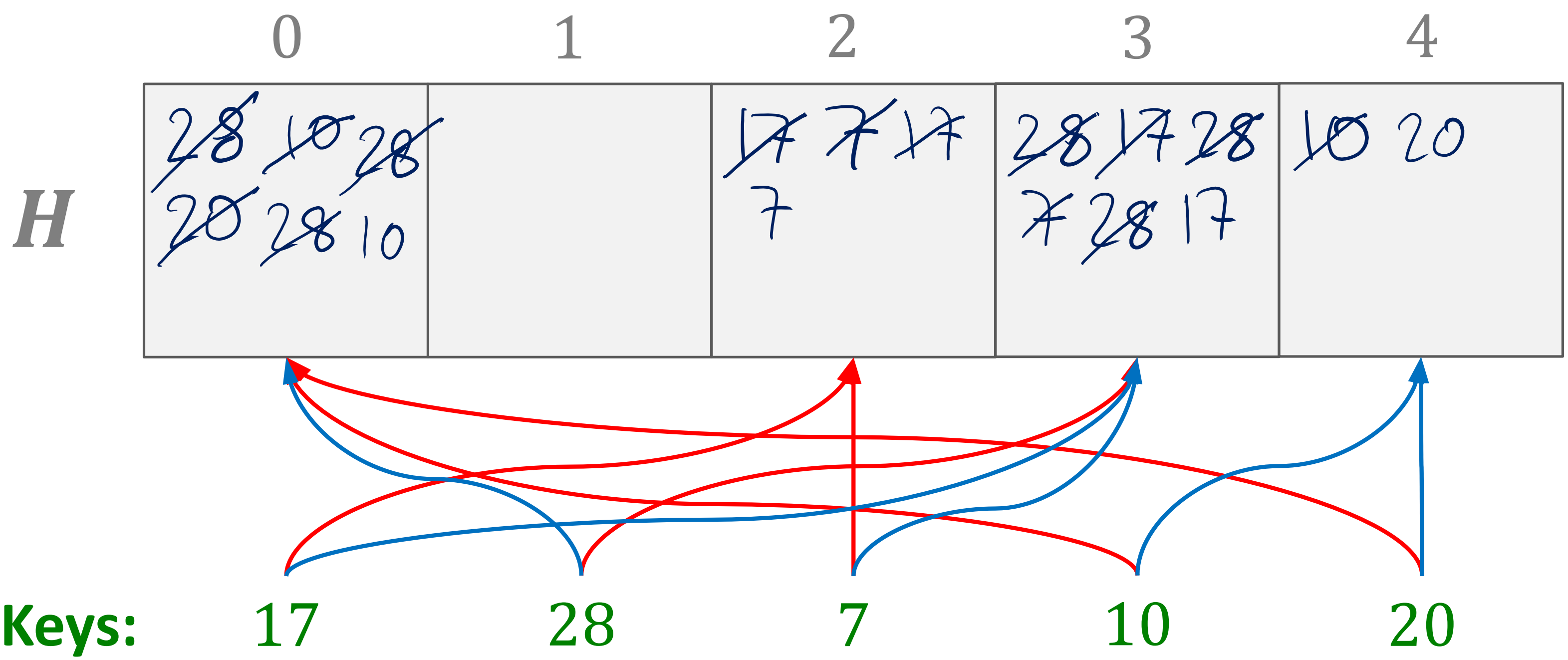
- If x is in the table, it is at position $h_1(x)$ or $h_2(x)$
- For delete: Mark table entry as empty!
- Both operations always require time $O(1)$!

Cuckoo Hashing Example

Table size: $m = 5$

Hash functions: $h_1(x) = x \bmod 5$, $h_2(x) = 2x - 1 \bmod 5$

Insert keys 17, 28, 7, 10, 20:



Cuckoo Hashing : Cycles

- When inserting, we can get a cycle
  - x replaces y_1
  - y_1 replaces y_2
  - y_2 replaces y_3
  - …
  - $y_{\ell-1}$ replaces y_ℓ
  - y_ℓ replaces x or y_i for some $i < \ell$
- Or it can happen that for some key $h_1(y_i) = h_2(y_i)$
- If this happens, we can also try the alternative position for x, but there the same can happen again…
- In this case, one chooses new hash functions and performs a rehash (usually with a larger table).

Cuckoo Hashing : Hash Functions

How to choose the two hash functions?

- They should be as “independent” as possible…
- Few keys x for which $h_1(x) = h_2(x)$
- A good choice:

  two independent, random functions from a universal set

- Then, one can show that cycles only occur rarely as long as $n \leq m/2$.

- As soon as the table is half full ($n \geq m/2$), one should do a rehash and switch to a table of twice the size.

Cuckoo Hashing : Running Time

Find / Delete:

- Always running time $O(1)$
- One only has to inspect the two positions $h_1(x)$ and $h_2(x)$.
- This is the big advantage of cuckoo hashing.

Insert:

- One can show that on average, it also requires time $O(1)$
- If the table is not filled to more than half its size
- Doubling the table size when rehashing leads to constant average running time per operation!

Hashing Summary

Efficient method to implement a dictionary

Handling of Collisions

- Hashing with separate chaining
  - simple, very flexible, with 2 hash functions, the list lengths can be restricted to $O(\log \log n)$ with high probability
- Open Addressing
  - different possibilities, more efficient in practice
  - possible to implement such that find has worst-case time $O(1)$.
  - load $\alpha > 1$ impossible, if α becomes large, one has to do a rehash

Hash Functions

- There are simple strategies to obtain good hash functions.
  - In practice, often, a single fixed hash function is used.

Rehash

- If a hash table becomes too full, one has to reset the whole table
  - This can be done such that the average running time per operation is still constant.