

Dictionary:

- ▶ **$S.insert(x)$** : Insert an element x .
- ▶ **$S.delete(x)$** : Delete the element pointed to by x .
- ▶ **$S.search(k)$** : Return a pointer to an element e with $key[e] = k$ in S if it exists; otherwise return null.

So far we have implemented the search for a key by carefully choosing split-elements.

Then the memory location of an object x with key k is determined by successively comparing k to split-elements.

Hashing tries to **directly** compute the memory location from the given key. The goal is to have constant search time.

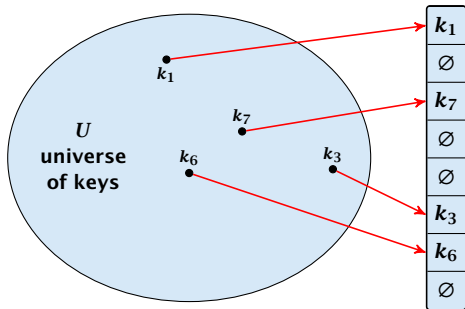
Definitions:

- ▶ Universe U of keys, e.g., $U \subseteq \mathbb{N}_0$. U very large.
- ▶ Set $S \subseteq U$ of keys, $|S| = m \leq |U|$.
- ▶ Array $T[0, \dots, n - 1]$ hash-table.
- ▶ Hash function $h : U \rightarrow [0, \dots, n - 1]$.

The hash-function h should fulfill:

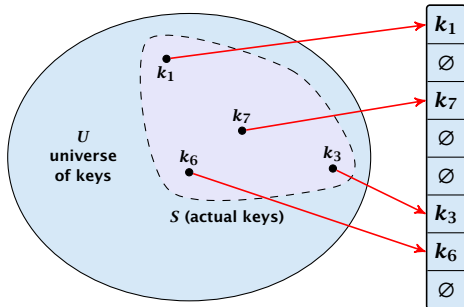
- ▶ Fast to evaluate.
- ▶ Small storage requirement.
- ▶ Good distribution of elements over the whole table.

Ideally the hash function maps **all** keys to different memory locations.



This special case is known as **Direct Addressing**. It is usually very unrealistic as the universe of keys typically is quite large, and in particular larger than the available memory.

Suppose that we **know** the set S of actual keys (no insert/no delete). Then we may want to design a **simple** hash-function that maps all these keys to different memory locations.



Such a hash function h is called a **perfect hash function** for set S .

If we do not know the keys in advance, the best we can hope for is that the hash function distributes keys evenly across the table.

Problem: Collisions

Usually the universe U is much larger than the table-size n .

Hence, there may be two elements k_1, k_2 from the set S that map to the same memory location (i.e., $h(k_1) = h(k_2)$). This is called a **collision**.

Typically, collisions do not appear once the size of the set S of actual keys gets close to n , but already when $|S| \geq \omega(\sqrt{n})$.

Lemma 1

*The probability of having a collision when hashing m elements into a table of size n under **uniform hashing** is at least*

$$1 - e^{-\frac{m(m-1)}{2n}} \approx 1 - e^{-\frac{m^2}{2n}} .$$

Uniform hashing:

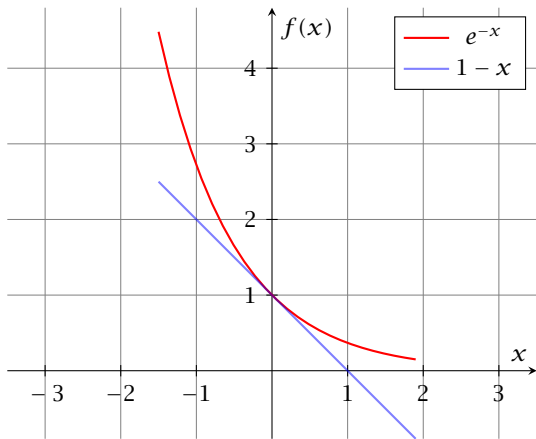
Choose a hash function uniformly at random from all functions $f : U \rightarrow [0, \dots, n - 1]$.

Proof.

Let $A_{m,n}$ denote the event that inserting m keys into a table of size n does **not** generate a collision. Then

$$\begin{aligned} \Pr[A_{m,n}] &= \prod_{\ell=1}^m \frac{n - \ell + 1}{n} = \prod_{j=0}^{m-1} \left(1 - \frac{j}{n}\right) \\ &\leq \prod_{j=0}^{m-1} e^{-j/n} = e^{-\sum_{j=0}^{m-1} \frac{j}{n}} = e^{-\frac{m(m-1)}{2n}}. \end{aligned}$$

Here the first equality follows since the ℓ -th element that is hashed has a probability of $\frac{n-\ell+1}{n}$ to not generate a collision under the condition that the previous elements did not induce collisions. □



The inequality $1 - x \leq e^{-x}$ is derived by stopping the Taylor-expansion of e^{-x} after the second term.

Resolving Collisions

The methods for dealing with collisions can be classified into the two main types

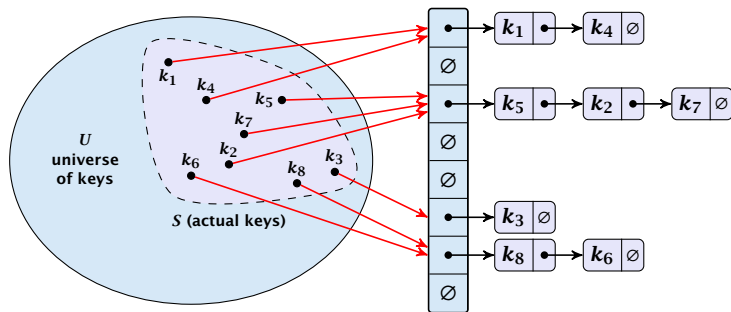
- ▶ **open addressing**, aka. closed hashing
- ▶ **hashing with chaining**, aka. closed addressing, open hashing.

There are applications e.g. computer chess where you do not resolve collisions at all.

Hashing with Chaining

Arrange elements that map to the same position in a linear list.

- ▶ Access: compute $h(x)$ and search list for $\text{key}[x]$.
- ▶ Insert: insert at the front of the list.



Let A denote a strategy for resolving collisions. We use the following notation:

- ▶ A^+ denotes the average time for a **successful** search when using A ;
- ▶ A^- denotes the average time for an **unsuccessful** search when using A ;
- ▶ We parameterize the complexity results in terms of $\alpha := \frac{m}{n}$, the so-called **fill factor** of the hash-table.

We assume **uniform hashing** for the following analysis.

Hashing with Chaining

The time required for an unsuccessful search is 1 plus the length of the list that is examined. The average length of a list is $\alpha = \frac{m}{n}$. Hence, if A is the collision resolving strategy “Hashing with Chaining” we have

$$A^- = 1 + \alpha .$$

Hashing with Chaining

For a successful search observe that we do **not** choose a list at random, but we consider a random key k in the hash-table and ask for the search-time for k .

This is 1 plus the number of elements that lie before k in k 's list.

Let k_ℓ denote the ℓ -th key inserted into the table.

Let for two keys k_i and k_j , X_{ij} denote the indicator variable for the event that k_i and k_j hash to the same position. Clearly, $\Pr[X_{ij} = 1] = 1/n$ for uniform hashing.

The expected successful search cost is

$$\mathbb{E} \left[\frac{1}{m} \sum_{i=1}^m \left(1 + \sum_{j=i+1}^m X_{ij} \right) \right]$$

keys before k_i

cost for key k_i

Hashing with Chaining

$$\begin{aligned} E \left[\frac{1}{m} \sum_{i=1}^m \left(1 + \sum_{j=i+1}^m X_{ij} \right) \right] &= \frac{1}{m} \sum_{i=1}^m \left(1 + \sum_{j=i+1}^m E[X_{ij}] \right) \\ &= \frac{1}{m} \sum_{i=1}^m \left(1 + \sum_{j=i+1}^m \frac{1}{n} \right) \\ &= 1 + \frac{1}{mn} \sum_{i=1}^m (m - i) \\ &= 1 + \frac{1}{mn} \left(m^2 - \frac{m(m+1)}{2} \right) \\ &= 1 + \frac{m-1}{2n} = 1 + \frac{\alpha}{2} - \frac{\alpha}{2m} . \end{aligned}$$

Hence, the expected cost for a successful search is $A^+ \leq 1 + \frac{\alpha}{2}$.

Hashing with Chaining

Disadvantages:

- ▶ pointers increase memory requirements
- ▶ pointers may lead to bad cache efficiency

Advantages:

- ▶ no à priori limit on the number of elements
- ▶ deletion can be implemented efficiently
- ▶ by using balanced trees instead of linked list one can also obtain worst-case guarantees.

Open Addressing

All objects are stored in the table itself.

Define a function $h(k, j)$ that determines the table-position to be examined in the j -th step. The values $h(k, 0), \dots, h(k, n - 1)$ must form a permutation of $0, \dots, n - 1$.

Search(k): Try position $h(k, 0)$; if it is empty your search fails; otw. continue with $h(k, 1), h(k, 2), \dots$

Insert(x): Search until you find an empty slot; insert your element there. If your search reaches $h(k, n - 1)$, and this slot is non-empty then your table is full.

Open Addressing

Choices for $h(k, j)$:

- ▶ **Linear probing:**

$$h(k, i) = h(k) + i \pmod n$$

(sometimes: $h(k, i) = h(k) + ci \pmod n$).

- ▶ **Quadratic probing:**

$$h(k, i) = h(k) + c_1i + c_2i^2 \pmod n.$$

- ▶ **Double hashing:**

$$h(k, i) = h_1(k) + ih_2(k) \pmod n.$$

For quadratic probing and double hashing one has to ensure that the search covers all positions in the table (i.e., for double hashing $h_2(k)$ must be relatively prime to n (**teilerfremd**); for quadratic probing c_1 and c_2 have to be chosen carefully).

Linear Probing

- ▶ Advantage: **Cache-efficiency**. The new probe position is very likely to be in the cache.
- ▶ Disadvantage: **Primary clustering**. Long sequences of occupied table-positions get longer as they have a larger probability to be hit. Furthermore, they can merge forming larger sequences.

Lemma 2

Let L be the method of linear probing for resolving collisions:

$$L^+ \approx \frac{1}{2} \left(1 + \frac{1}{1 - \alpha} \right)$$

$$L^- \approx \frac{1}{2} \left(1 + \frac{1}{(1 - \alpha)^2} \right)$$

Quadratic Probing

- ▶ Not as cache-efficient as Linear Probing.
- ▶ **Secondary clustering**: caused by the fact that all keys mapped to the same position have the same probe sequence.

Lemma 3

Let Q be the method of quadratic probing for resolving collisions:

$$Q^+ \approx 1 + \ln\left(\frac{1}{1-\alpha}\right) - \frac{\alpha}{2}$$

$$Q^- \approx \frac{1}{1-\alpha} + \ln\left(\frac{1}{1-\alpha}\right) - \alpha$$

Double Hashing

- ▶ Any probe into the hash-table usually creates a cache-miss.

Lemma 4

Let A be the method of double hashing for resolving collisions:

$$D^+ \approx \frac{1}{\alpha} \ln \left(\frac{1}{1 - \alpha} \right)$$

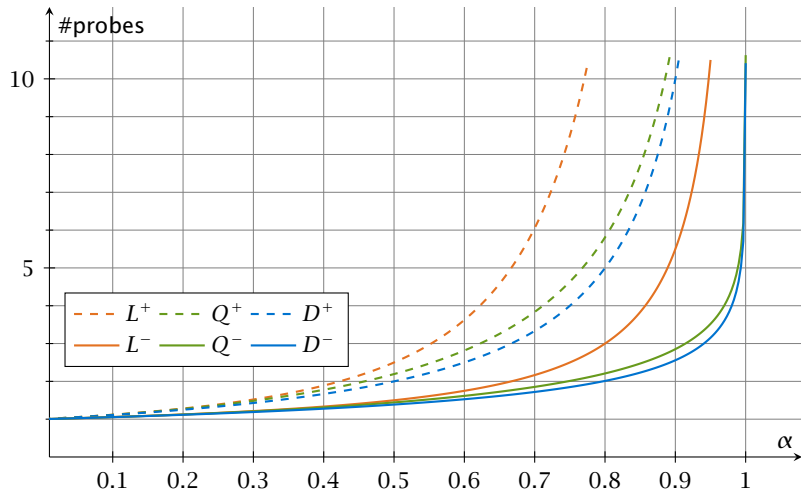
$$D^- \approx \frac{1}{1 - \alpha}$$

Open Addressing

Some values:

α	<i>Linear Probing</i>		<i>Quadratic Probing</i>		<i>Double Hashing</i>	
	L^+	L^-	Q^+	Q^-	D^+	D^-
0.5	1.5	2.5	1.44	2.19	1.39	2
0.9	5.5	50.5	2.85	11.40	2.55	10
0.95	10.5	200.5	3.52	22.05	3.15	20

Open Addressing



Analysis of Idealized Open Address Hashing

We analyze the time for a search in a very idealized Open Addressing scheme.

- ▶ The probe sequence $h(k, 0), h(k, 1), h(k, 2), \dots$ is equally likely to be any permutation of $\langle 0, 1, \dots, n - 1 \rangle$.

Analysis of Idealized Open Address Hashing

Let X denote a random variable describing the number of probes in an **unsuccessful** search.

Let A_i denote the event that the i -th probe **occurs** and is to a non-empty slot.

$$\begin{aligned}\Pr[A_1 \cap A_2 \cap \dots \cap A_{i-1}] \\ &= \Pr[A_1] \cdot \Pr[A_2 \mid A_1] \cdot \Pr[A_3 \mid A_1 \cap A_2] \cdot \\ &\quad \dots \cdot \Pr[A_{i-1} \mid A_1 \cap \dots \cap A_{i-2}]\end{aligned}$$

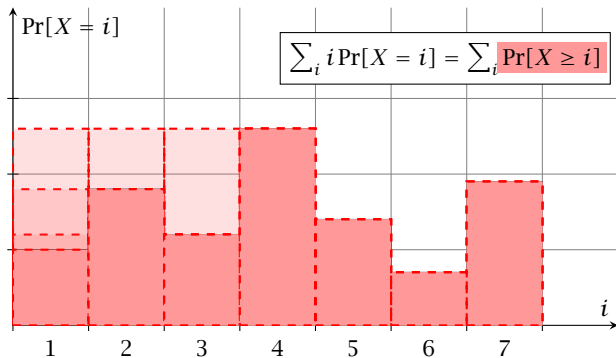
$$\begin{aligned}\Pr[X \geq i] &= \frac{m}{n} \cdot \frac{m-1}{n-1} \cdot \frac{m-2}{n-2} \cdot \dots \cdot \frac{m-i+2}{n-i+2} \\ &\leq \left(\frac{m}{n}\right)^{i-1} = \alpha^{i-1} .\end{aligned}$$

Analysis of Idealized Open Address Hashing

$$E[X] = \sum_{i=1}^{\infty} \Pr[X \geq i] \leq \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha} .$$

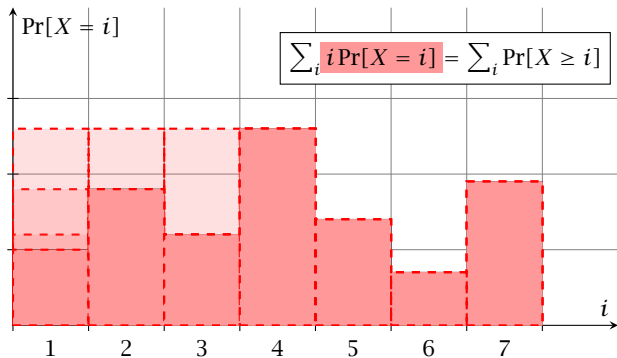
$$\frac{1}{1-\alpha} = 1 + \alpha + \alpha^2 + \alpha^3 + \dots$$

$i = 3$



The j -th rectangle appears in both sums j times. (j times in the first due to multiplication with j ; and j times in the second for summands $i = 1, 2, \dots, j$)

$i = 4$



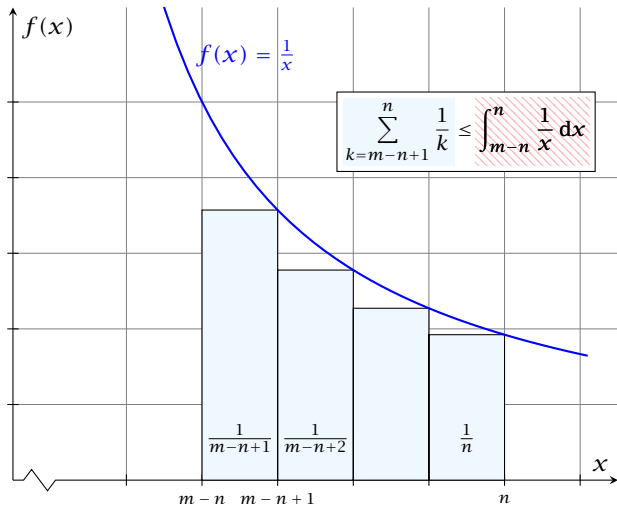
The j -th rectangle appears in both sums j times. (j times in the first due to multiplication with j ; and j times in the second for summands $i = 1, 2, \dots, j$)

Analysis of Idealized Open Address Hashing

The number of probes in a **successful** search for k is equal to the number of probes made in an unsuccessful search for k at the time that k is inserted.

Let k be the $i + 1$ -st element. The expected time for a search for k is at most $\frac{1}{1-i/n} = \frac{n}{n-i}$.

$$\begin{aligned} \frac{1}{m} \sum_{i=0}^{m-1} \frac{n}{n-i} &= \frac{n}{m} \sum_{i=0}^{m-1} \frac{1}{n-i} = \frac{1}{\alpha} \sum_{k=n-m+1}^n \frac{1}{k} \\ &\leq \frac{1}{\alpha} \int_{n-m}^n \frac{1}{x} dx = \frac{1}{\alpha} \ln \frac{n}{n-m} = \frac{1}{\alpha} \ln \frac{1}{1-\alpha} . \end{aligned}$$



How do we delete in a hash-table?

- ▶ For hashing with chaining this is not a problem. Simply search for the key, and delete the item in the corresponding list.
- ▶ For open addressing this is difficult.

Deletions

- ▶ Simply removing a key might interrupt the probe sequence of other keys which then cannot be found anymore.
- ▶ One can delete an element by replacing it with a deleted-marker.
 - ▶ During an insertion if a deleted-marker is encountered an element can be inserted there.
 - ▶ During a search a deleted-marker must not be used to terminate the probe sequence.
- ▶ The table could fill up with deleted-markers leading to bad performance.
- ▶ If a table contains many deleted-markers (linear fraction of the keys) one can rehash the whole table and amortize the cost for this rehash against the cost for the deletions.

Deletions for Linear Probing

- ▶ For Linear Probing one can delete elements without using deletion-markers.
- ▶ Upon a deletion elements that are further down in the probe-sequence may be moved to guarantee that they are still found during a search.

Deletions for Linear Probing

Algorithm 16 delete(p)

```
1:  $T[p] \leftarrow \text{null}$ 
2:  $p \leftarrow \text{succ}(p)$ 
3: while  $T[p] \neq \text{null}$  do
4:    $y \leftarrow T[p]$ 
5:    $T[p] \leftarrow \text{null}$ 
6:    $p \leftarrow \text{succ}(p)$ 
7:   insert( $y$ )
```

p is the index into the table-cell that contains the object to be deleted.

Pointers into the hash-table become invalid.

Universal Hashing

Regardless, of the choice of hash-function there is always an input (a set of keys) that has a very poor worst-case behaviour.

Therefore, so far we assumed that the hash-function is random so that regardless of the input the average case behaviour is good.

However, the assumption of uniform hashing that h is chosen randomly from all functions $f : U \rightarrow [0, \dots, n - 1]$ is clearly unrealistic as there are $n^{|U|}$ such functions. Even writing down such a function would take $|U| \log n$ bits.

Universal hashing tries to define a set \mathcal{H} of functions that is much smaller but still leads to good average case behaviour when selecting a hash-function uniformly at random from \mathcal{H} .

Universal Hashing

Definition 5

A class \mathcal{H} of hash-functions from the universe U into the set $\{0, \dots, n-1\}$ is called **universal** if for all $u_1, u_2 \in U$ with $u_1 \neq u_2$

$$\Pr[h(u_1) = h(u_2)] \leq \frac{1}{n} ,$$

where the probability is w. r. t. the choice of a random hash-function from set \mathcal{H} .

Note that this means that the probability of a collision is at most $\frac{1}{n}$.

Universal Hashing

Definition 6

A class \mathcal{H} of hash-functions from the universe U into the set $\{0, \dots, n-1\}$ is called **2-independent** (pairwise independent) if the following two conditions hold

- ▶ For any key $u \in U$, and $t \in \{0, \dots, n-1\}$ $\Pr[h(u) = t] = \frac{1}{n}$, i.e., a key is distributed uniformly within the hash-table.
- ▶ For all $u_1, u_2 \in U$ with $u_1 \neq u_2$, and for any two hash-positions t_1, t_2 :

$$\Pr[h(u_1) = t_1 \wedge h(u_2) = t_2] \leq \frac{1}{n^2} .$$

This requirement clearly implies a universal hash-function.

Definition 7

A class \mathcal{H} of hash-functions from the universe U into the set $\{0, \dots, n-1\}$ is called **k-independent** if for any choice of $\ell \leq k$ distinct keys $u_1, \dots, u_\ell \in U$, and for any set of ℓ not necessarily distinct hash-positions t_1, \dots, t_ℓ :

$$\Pr[h(u_1) = t_1 \wedge \dots \wedge h(u_\ell) = t_\ell] \leq \frac{1}{n^\ell} ,$$

where the probability is w. r. t. the choice of a random hash-function from set \mathcal{H} .

Definition 8

A class \mathcal{H} of hash-functions from the universe U into the set $\{0, \dots, n-1\}$ is called (μ, k) -independent if for any choice of $\ell \leq k$ distinct keys $u_1, \dots, u_\ell \in U$, and for any set of ℓ not necessarily distinct hash-positions t_1, \dots, t_ℓ :

$$\Pr[h(u_1) = t_1 \wedge \dots \wedge h(u_\ell) = t_\ell] \leq \frac{\mu}{n^\ell} ,$$

where the probability is w. r. t. the choice of a random hash-function from set \mathcal{H} .

Universal Hashing

Let $U := \{0, \dots, p - 1\}$ for a prime p . Let $\mathbb{Z}_p := \{0, \dots, p - 1\}$, and let $\mathbb{Z}_p^* := \{1, \dots, p - 1\}$ denote the set of invertible elements in \mathbb{Z}_p .

Define

$$h_{a,b}(x) := (ax + b \bmod p) \bmod n$$

Lemma 9

The class

$$\mathcal{H} = \{h_{a,b} \mid a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\}$$

is a universal class of hash-functions from U to $\{0, \dots, n - 1\}$.

Universal Hashing

Proof.

Let $x, y \in U$ be two distinct keys. We have to show that the probability of a collision is only $1/n$.

$$\blacktriangleright ax + b \not\equiv ay + b \pmod{p}$$

If $x \neq y$ then $(x - y) \not\equiv 0 \pmod{p}$.

Multiplying with $a \not\equiv 0 \pmod{p}$ gives

$$a(x - y) \not\equiv 0 \pmod{p}$$

where we use that \mathbb{Z}_p is a field (**Körper**) and, hence, has no zero divisors (**nullteilerfrei**).

- ▶ The hash-function does not generate collisions before the $(\text{mod } n)$ -operation. Furthermore, every choice (a, b) is mapped to a different pair (t_x, t_y) with $t_x := ax + b$ and $t_y := ay + b$.

This holds because we can compute a and b when given t_x and t_y :

$$t_x \equiv ax + b \pmod{p}$$

$$t_y \equiv ay + b \pmod{p}$$

$$t_x - t_y \equiv a(x - y) \pmod{p}$$

$$t_y \equiv ay + b \pmod{p}$$

$$a \equiv (t_x - t_y)(x - y)^{-1} \pmod{p}$$

$$b \equiv t_y - ay \pmod{p}$$

Universal Hashing

There is a one-to-one correspondence between hash-functions (pairs (a, b) , $a \neq 0$) and pairs (t_x, t_y) , $t_x \neq t_y$.

Therefore, we can view the first step (before the mod n -operation) as choosing a pair (t_x, t_y) , $t_x \neq t_y$ uniformly at random.

What happens when we do the mod n operation?

Fix a value t_x . There are $p - 1$ possible values for choosing t_y .

From the range $0, \dots, p - 1$ the values $t_x, t_x + n, t_x + 2n, \dots$ map to t_x after the modulo-operation. These are at most $\lceil p/n \rceil$ values.

Universal Hashing

As $t_y \neq t_x$ there are

$$\left\lceil \frac{p}{n} \right\rceil - 1 \leq \frac{p}{n} + \frac{n-1}{n} - 1 \leq \frac{p-1}{n}$$

possibilities for choosing t_y such that the final hash-value creates a collision.

This happens with probability at most $\frac{1}{n}$.

Universal Hashing

It is also possible to show that \mathcal{H} is an (almost) pairwise independent class of hash-functions.

$$\frac{\lfloor \frac{p}{n} \rfloor^2}{p(p-1)} \leq \Pr_{t_x \neq t_y \in \mathbb{Z}_p^2} \left[\begin{array}{l} t_x \bmod n = h_1 \\ t_y \bmod n = h_2 \end{array} \right] \leq \frac{\lceil \frac{p}{n} \rceil^2}{p(p-1)}$$

Note that the middle is the probability that $h(x) = h_1$ and $h(y) = h_2$. The total number of choices for (t_x, t_y) is $p(p-1)$. The number of choices for t_x (t_y) such that $t_x \bmod n = h_1$ ($t_y \bmod n = h_2$) lies between $\lfloor \frac{p}{n} \rfloor$ and $\lceil \frac{p}{n} \rceil$.

Definition 10

Let $d \in \mathbb{N}$; $q \geq (d + 1)n$ be a prime; and let $\vec{a} \in \{0, \dots, q - 1\}^{d+1}$. Define for $x \in \{0, \dots, q\}$

$$h_{\vec{a}}(x) := \left(\sum_{i=0}^d a_i x^i \bmod q \right) \bmod n .$$

Let $\mathcal{H}_n^d := \{h_{\vec{a}} \mid \vec{a} \in \{0, \dots, q\}^{d+1}\}$. The class \mathcal{H}_n^d is $(e, d + 1)$ -independent.

Note that in the previous case we had $d = 1$ and chose $a_d \neq 0$.

For the coefficients $\bar{a} \in \{0, \dots, q - 1\}^{d+1}$ let $f_{\bar{a}}$ denote the polynomial

$$f_{\bar{a}}(x) = \left(\sum_{i=0}^d a_i x^i \right) \bmod q$$

The polynomial is defined by $d + 1$ distinct points.

Fix $\ell \leq d + 1$; let $x_1, \dots, x_\ell \in \{0, \dots, q - 1\}$ be keys, and let t_1, \dots, t_ℓ denote the corresponding hash-function values.

Let $A^\ell = \{h_{\bar{a}} \in \mathcal{H} \mid h_{\bar{a}}(x_i) = t_i \text{ for all } i \in \{1, \dots, \ell\}\}$

Then

$$h_{\bar{a}} \in A^\ell \Leftrightarrow h_{\bar{a}} = f_{\bar{a}} \bmod n \text{ and}$$

$$f_{\bar{a}}(x_i) \in \underbrace{\{t_i + \alpha \cdot n \mid \alpha \in \{0, \dots, \lceil \frac{q}{n} \rceil - 1\}\}}_{=: B_i}$$

In order to obtain the cardinality of A^ℓ we choose our polynomial by fixing $d + 1$ points.

We first fix the values for inputs x_1, \dots, x_ℓ .

We have

$$|B_1| \cdot \dots \cdot |B_\ell|$$

possibilities to do this (so that $h_{\bar{a}}(x_i) = t_i$).

- A^ℓ denotes the set of hash-functions such that every x_i hits its pre-defined position t_i .
- B_i is the set of positions that $f_{\bar{a}}$ can hit so that $h_{\bar{a}}$ still hits t_i .

Now, we choose $d - \ell + 1$ other inputs and choose their value arbitrarily. We have $q^{d-\ell+1}$ possibilities to do this.

Therefore we have

$$|B_1| \cdot \dots \cdot |B_\ell| \cdot q^{d-\ell+1} \leq \left\lceil \frac{q}{n} \right\rceil^\ell \cdot q^{d-\ell+1}$$

possibilities to choose \bar{a} such that $h_{\bar{a}} \in A_\ell$.

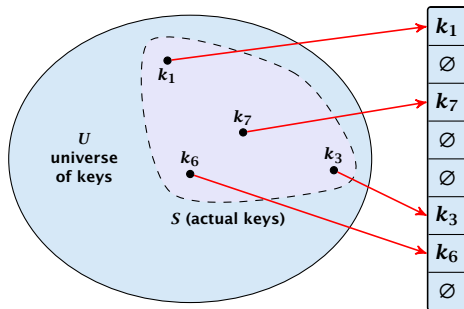
Therefore the probability of choosing $h_{\bar{a}}$ from A_ℓ is only

$$\begin{aligned} \frac{\left[\frac{q}{n}\right]^\ell \cdot q^{d-\ell+1}}{q^{d+1}} &\leq \frac{\left(\frac{q+n}{n}\right)^\ell}{q^\ell} \leq \left(\frac{q+n}{q}\right)^\ell \cdot \frac{1}{n^\ell} \\ &\leq \left(1 + \frac{1}{\ell}\right)^\ell \cdot \frac{1}{n^\ell} \leq \frac{e}{n^\ell} . \end{aligned}$$

This shows that the \mathcal{H} is $(e, d + 1)$ -universal.

Perfect Hashing

Suppose that we **know** the set S of actual keys (no insert/no delete). Then we may want to design a **simple** hash-function that maps all these keys to different memory locations.



Perfect Hashing

Let $m = |S|$. We could simply choose the hash-table size very large so that we don't get any collisions.

Using a universal hash-function the expected number of collisions is

$$E[\#\text{Collisions}] = \binom{m}{2} \cdot \frac{1}{n}.$$

If we choose $n = m^2$ the **expected number** of collisions is strictly less than $\frac{1}{2}$.

Can we get an upper bound on the **probability of having collisions**?

The probability of having 1 or more collisions can be at most $\frac{1}{2}$ as otherwise the expectation would be larger than $\frac{1}{2}$.

Perfect Hashing

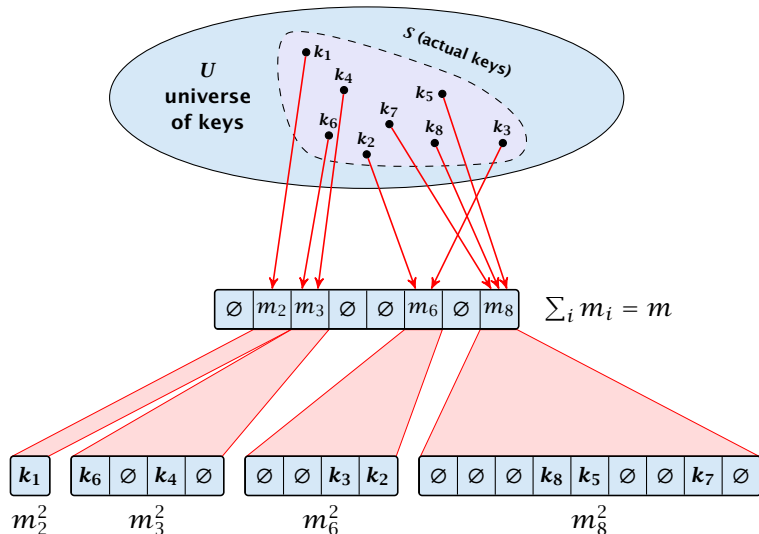
We can find such a hash-function by a few trials.

However, a hash-table size of $n = m^2$ is very very high.

We construct a two-level scheme. We first use a hash-function that maps elements from S to m buckets.

Let m_j denote the number of items that are hashed to the j -th bucket. For each bucket we choose a second hash-function that maps the elements of the bucket into a table of size m_j^2 . The second function can be chosen such that all elements are mapped to different locations.

Perfect Hashing



Perfect Hashing

The total memory that is required by all hash-tables is $\mathcal{O}(\sum_j m_j^2)$. Note that m_j is a random variable.

$$\begin{aligned} \mathbb{E} \left[\sum_j m_j^2 \right] &= \mathbb{E} \left[2 \sum_j \binom{m_j}{2} + \sum_j m_j \right] \\ &= 2 \mathbb{E} \left[\sum_j \binom{m_j}{2} \right] + \mathbb{E} \left[\sum_j m_j \right] \end{aligned}$$

The first expectation is simply the expected number of collisions, for the first level. Since we use universal hashing we have

$$= 2 \binom{m}{2} \frac{1}{m} + m = 2m - 1 .$$

Perfect Hashing

We need only $\mathcal{O}(m)$ time to construct a hash-function h with $\sum_j m_j^2 = \mathcal{O}(4m)$, because with probability at least $1/2$ a random function from a universal family will have this property.

Then we construct a hash-table h_j for every bucket. This takes expected time $\mathcal{O}(m_j)$ for every bucket. A random function h_j is collision-free with probability at least $1/2$. We need $\mathcal{O}(m_j)$ to test this.

We only need that the hash-functions are chosen from a universal family!!!

Cuckoo Hashing

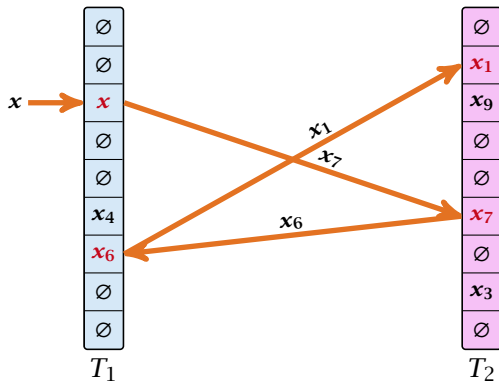
Goal:

Try to generate a hash-table with constant worst-case search time in a dynamic scenario.

- ▶ Two hash-tables $T_1[0, \dots, n - 1]$ and $T_2[0, \dots, n - 1]$, with hash-functions h_1 , and h_2 .
- ▶ An object x is either stored at location $T_1[h_1(x)]$ or $T_2[h_2(x)]$.
- ▶ A search clearly takes constant time if the above constraint is met.

Cuckoo Hashing

Insert:



Algorithm 17 Cuckoo-Insert(x)

```
1: if  $T_1[h_1(x)] = x \vee T_2[h_2(x)] = x$  then return  
2: steps  $\leftarrow 1$   
3: while steps  $\leq$  maxsteps do  
4:     exchange  $x$  and  $T_1[h_1(x)]$   
5:     if  $x = \text{null}$  then return  
6:     exchange  $x$  and  $T_2[h_2(x)]$   
7:     if  $x = \text{null}$  then return  
8:     steps  $\leftarrow$  steps + 1  
9: rehash() // change hash-functions; rehash everything  
10: Cuckoo-Insert( $x$ )
```

Cuckoo Hashing

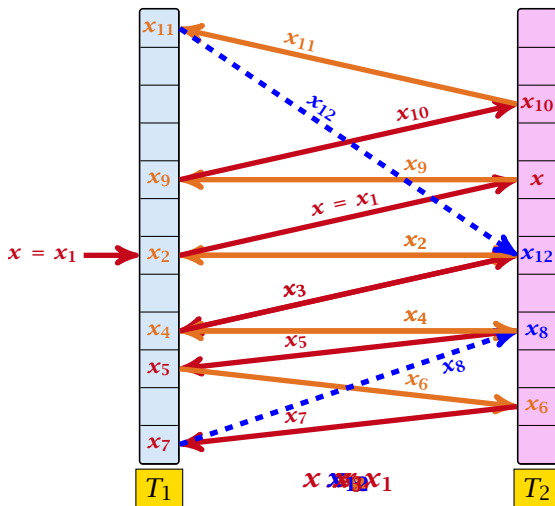
- ▶ We call one iteration through the while-loop a **step** of the algorithm.
- ▶ We call a sequence of iterations through the while-loop without the termination condition becoming true a **phase** of the algorithm.
- ▶ We say a phase is **successful** if it is not terminated by the maxstep-condition, but the while loop is left because $x = \text{null}$.

What is the expected time for an insert-operation?

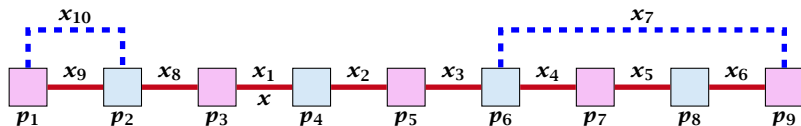
We first analyze the probability that we end-up in an infinite loop (that is then terminated after `maxsteps` steps).

Formally what is the probability to enter an infinite loop that touches s different keys?

Cuckoo Hashing: Insert



Cuckoo Hashing



Cuckoo Hashing

A cycle-structure is **active** if for every key x_ℓ (linking a cell p_i from T_1 and a cell p_j from T_2) we have

$$h_1(x_\ell) = p_i \quad \text{and} \quad h_2(x_\ell) = p_j$$

Observation:

If during a phase the insert-procedure runs into a cycle there must exist an active cycle structure of size $s \geq 3$.

Cuckoo Hashing

What is the probability that all keys in a cycle-structure of size s correctly map into their T_1 -cell?

This probability is at most $\frac{\mu}{n^s}$ since h_1 is a (μ, s) -independent hash-function.

What is the probability that all keys in the cycle-structure of size s correctly map into their T_2 -cell?

This probability is at most $\frac{\mu}{n^s}$ since h_2 is a (μ, s) -independent hash-function.

These events are independent.

Cuckoo Hashing

The probability that a given cycle-structure of size s is active is at most $\frac{\mu^2}{n^{2s}}$.

What is the probability that **there exists** an active cycle structure of size s ?

Cuckoo Hashing

The number of cycle-structures of size s is at most

$$s^3 \cdot n^{s-1} \cdot m^{s-1} .$$

- ▶ There are at most s^2 possibilities where to attach the forward and backward links.
- ▶ There are at most s possibilities to choose where to place key x .
- ▶ There are m^{s-1} possibilities to choose the keys apart from x .
- ▶ There are n^{s-1} possibilities to choose the cells.

Cuckoo Hashing

The probability that there exists an active cycle-structure is therefore at most

$$\begin{aligned} \sum_{s=3}^{\infty} s^3 \cdot n^{s-1} \cdot m^{s-1} \cdot \frac{\mu^2}{n^{2s}} &= \frac{\mu^2}{nm} \sum_{s=3}^{\infty} s^3 \left(\frac{m}{n}\right)^s \\ &\leq \frac{\mu^2}{m^2} \sum_{s=3}^{\infty} s^3 \left(\frac{1}{1+\epsilon}\right)^s \leq \mathcal{O}\left(\frac{1}{m^2}\right). \end{aligned}$$

Here we used the fact that $(1 + \epsilon)m \leq n$.

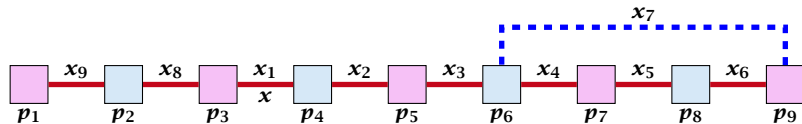
Hence,

$$\Pr[\text{cycle}] = \mathcal{O}\left(\frac{1}{m^2}\right).$$

Cuckoo Hashing

Now, we analyze the probability that a phase is not successful without running into a closed cycle.

Cuckoo Hashing



Sequence of visited keys:

$x = x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_3, x_2, x_1 = x, x_8, x_9, \dots$

Cuckoo Hashing

Consider the sequence of not necessarily distinct keys starting with x in the order that they are visited during the phase.

Lemma 11

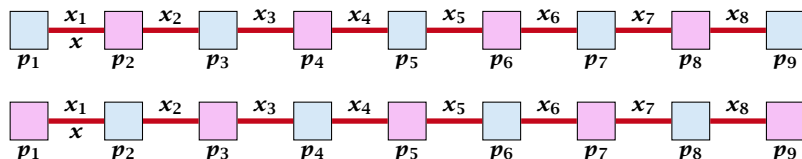
*If the sequence is of length p then there exists a sub-sequence of at least $p/3$ keys starting with x of *distinct* keys.*

Proof.

x is contained at most twice in the sequence.

Either the sub-sequence starting from x until right before the first repeated key, or the sub-sequence starting from the repetition of x until the end must contain at least $p/3$ distinct keys. □

Cuckoo Hashing



A path-structure of size s is defined by

- ▶ $s + 1$ different cells (alternating btw. cells from T_1 and T_2).
- ▶ s distinct keys $x = x_1, x_2, \dots, x_s$, linking the cells.
- ▶ The leftmost cell is either from T_1 or T_2 .

Cuckoo Hashing

A path-structure is **active** if for every key x_ℓ (linking a cell p_i from T_1 and a cell p_j from T_2) we have

$$h_1(x_\ell) = p_i \quad \text{and} \quad h_2(x_\ell) = p_j$$

Observation:

If a phase takes at least t steps without running into a cycle there must exist an active path-structure of size $(2t - 1)/3$.

Cuckoo Hashing

The probability that a given path-structure of size s is active is at most $\frac{\mu^2}{n^{2s}}$.

The probability that there exists an active path-structure of size s is at most

$$\begin{aligned} & 2 \cdot n^{s+1} \cdot m^{s-1} \cdot \frac{\mu^2}{n^{2s}} \\ & \leq 2\mu^2 \left(\frac{m}{n}\right)^{s-1} \leq 2\mu^2 \left(\frac{1}{1+\epsilon}\right)^{s-1} \\ & \leq 2\mu^2 \left(\frac{1}{1+\epsilon}\right)^{(2t-1)/3-1} = 2\mu^2 \left(\frac{1}{1+\epsilon}\right)^{2(t-2)/3} . \end{aligned}$$

Cuckoo Hashing

We choose $\text{maxsteps} \geq \frac{3\ell}{2} + 1$. Then the probability that a phase is terminated unsuccessfully without running into a cycle is at most

$$\begin{aligned} & \Pr[\text{unsuccessful} \mid \text{no cycle}] \\ & \leq \Pr[\exists \text{ active path-structure of size at least } \frac{2(\text{maxsteps} + 1) - 1}{3}] \\ & \leq \Pr[\exists \text{ active path-structure of size at least } \ell + 1] \\ & \leq 2\mu^2 \left(\frac{1}{1 + \epsilon} \right)^\ell \leq \frac{1}{m^2} \end{aligned}$$

by choosing $\ell \geq \log \left(\frac{1}{2\mu^2 m^2} \right) / \log \left(\frac{1}{1 + \epsilon} \right) = \log(2\mu^2 m^2) / \log(1 + \epsilon)$

Note that this gives $\text{maxsteps} = \Theta(\log m)$.

Cuckoo Hashing

The expected number of steps in the **successful phase** of an insert operation is:

$$\begin{aligned} & E[\text{number of steps} \mid \text{phase successful}] \\ &= \sum_{t \geq 1} \Pr[\text{search takes at least } t \text{ steps} \mid \text{phase successful}] \end{aligned}$$

We have

$$\begin{aligned} & \Pr[\text{search at least } t \text{ steps} \mid \text{successful}] \\ &= \Pr[\text{search at least } t \text{ steps} \wedge \text{successful}] / \Pr[\text{successful}] \\ &\leq \frac{1}{c} \Pr[\text{search at least } t \text{ steps} \mid \text{no cycle}] , \end{aligned}$$

where we use the fact that for a suitable **constant** $c \geq 0$

$$\begin{aligned} \Pr[\text{successful}] &= \Pr[\text{no cycle}] - \Pr[\text{unsuccessful} \mid \text{no cycle}] \\ &\geq c \cdot \Pr[\text{no cycle}] \end{aligned}$$

Hence,

$$\begin{aligned} & \mathbb{E}[\text{number of steps} \mid \text{phase successful}] \\ &= \frac{1}{c} \sum_{t \geq 1} \Pr[\text{search at least } t \text{ steps} \mid \text{no cycle}] \\ &\leq \frac{1}{c} \left[1 + \sum_{t \geq 2} 2\mu^2 \left(\frac{1}{1+\epsilon} \right)^{2(t-2)/3} \right] \\ &= \frac{1}{c} + \frac{2\mu^2}{c} \sum_{t \geq 0} \left(\frac{1}{(1+\epsilon)^{2/3}} \right)^t = \mathcal{O}(1) . \end{aligned}$$

Cuckoo Hashing

A phase that is not successful induces cost $\mathcal{O}(m)$ for doing a complete rehash (this dominates the cost for the steps in the phase).

The probability that a phase is not successful is $p = \mathcal{O}(1/m^2)$ (probability $\mathcal{O}(1/m^2)$ of running into a cycle and probability $\mathcal{O}(1/m^2)$ of reaching maxsteps without running into a cycle).

The expected number of unsuccessful phases is

$$\sum_{i \geq 1} p^i = \frac{1}{1-p} - 1 = \frac{p}{1-p} = \mathcal{O}(p).$$

Therefore the expected cost for re-hashes is

$$\mathcal{O}(m) \cdot \mathcal{O}(p) = \mathcal{O}(1/m).$$

What kind of hash-functions do we need?

Since maxsteps is $\Theta(\log m)$ the largest size of a path-structure or cycle-structure contains just $\Theta(\log m)$ different keys.

Therefore, it is sufficient to have $(\mu, \Theta(\log m))$ -independent hash-functions.

Cuckoo Hashing

How do we make sure that $n \geq (1 + \epsilon)m$?

- ▶ Let $\alpha := 1/(1 + \epsilon)$.
- ▶ Keep track of the number of elements in the table. When $m \geq \alpha n$ we double n and do a complete re-hash (**table-expand**).
- ▶ Whenever m drops below $\alpha n/4$ we divide n by 2 and do a rehash (**table-shrink**).
- ▶ Note that right after a change in table-size we have $m = \alpha n/2$. In order for a table-expand to occur at least $\alpha n/2$ insertions are required. Similar, for a table-shrink at least $\alpha n/4$ deletions must occur.
- ▶ Therefore we can amortize the rehash cost after a change in table-size against the cost for insertions and deletions.

Cuckoo Hashing

Lemma 12

Cuckoo Hashing has an expected constant insert-time and a worst-case constant search-time.

Note that the above lemma only holds if the fill-factor (number of keys/total number of hash-table slots) is at most $\frac{1}{2(1+\epsilon)}$.