7.7 Hashing

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 $\text{key}[e] = k$ 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.

7.7 Hashing

Dictionary:

- $\boldsymbol{S}$**.insert**$(\boldsymbol{x})$: Insert an element x.
- $\boldsymbol{S}$**.delete**$(\boldsymbol{x})$: Delete the element pointed to by x.
- $\boldsymbol{S}$**.search**$(\boldsymbol{k})$: Return a pointer to an element e with $\mathrm{key}[e] = k$ 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.

7.7 Hashing

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 $\text{key}[e] = k$ 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.

7.7 Hashing

Dictionary:

- $\boldsymbol{S}$**.insert**$(\boldsymbol{x})$: Insert an element x.
- $\boldsymbol{S}$**.delete**$(\boldsymbol{x})$: Delete the element pointed to by x.
- $\boldsymbol{S}$**.search**$(\boldsymbol{k})$: Return a pointer to an element e with $\text{key}[e] = k$ 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.

7.7 Hashing

Definitions:

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

The hash-function h should fulfill:

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

7.7 Hashing

Definitions:

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

The hash-function h should fulfill:

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

7.7 Hashing

Definitions:

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

The hash-function h should fulfill:

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

7.7 Hashing

Definitions:

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

The hash-function h should fulfill:

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

7.7 Hashing

Definitions:

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

The hash-function h should fulfill:

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

7.7 Hashing

Definitions:

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

The hash-function h should fulfill:

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

7.7 Hashing

Definitions:

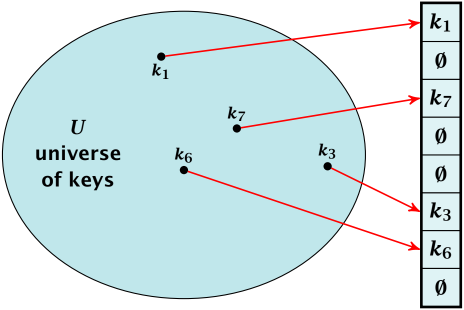
- Universe U of keys, e.g., $U \subseteq \mathbb{N}_0$. U very large.
- Set $S \subseteq U$ of keys, $|S| = m \leq n$.
- Array $T[0,\ldots,n-1]$ hash-table.
- Hash function $h : U \to [0,\ldots,n-1]$.

The hash-function h should fulfill:

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

7.7 Hashing

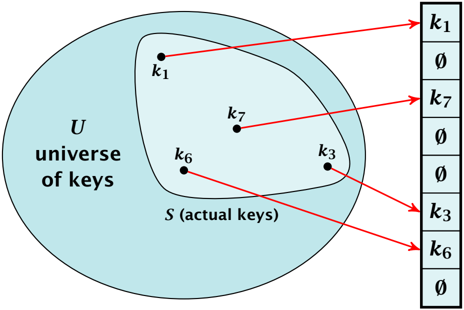
Ideally the hash function maps all keys to different memory locations.



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.

7.7 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.



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

7.7 Hashing

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.

7.7 Hashing

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.

7.7 Hashing

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.

7.7 Hashing

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

Lemma 21

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

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

Uniform hashing:

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

7.7 Hashing

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

Lemma 21

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

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

Uniform hashing:

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

7.7 Hashing

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

Lemma 21

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

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

Uniform hashing:

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

7.7 Hashing

Proof.

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

7.7 Hashing

Proof.

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

$$\Pr[A_{m,n}]$$

7.7 Hashing

Proof.

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

$$\Pr[A_{m,n}] = \prod_{\ell=1}^{m} \frac{n - \ell + 1}{n}$$

7.7 Hashing

Proof.

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

$$\Pr[A_{m,n}] = \prod_{\ell=1}^{m} \frac{n-\ell+1}{n} = \prod_{j=0}^{m-1}\left(1 - \frac{j}{n}\right)$$

7.7 Hashing

Proof.

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

$$\begin{aligned}\Pr[A_{m,n}] &= \prod_{\ell=1}^{m} \frac{n-\ell+1}{n} = \prod_{j=0}^{m-1} \Bigl(1-\frac{j}{n}\Bigr) \\ &\le \prod_{j=0}^{m-1} e^{-j/n}\end{aligned}$$

7.7 Hashing

Proof.

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

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

7.7 Hashing

Proof.

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

$$\Pr[A_{m,n}] = \prod_{\ell=1}^{m} \frac{n-\ell+1}{n} = \prod_{j=0}^{m-1}\Big(1-\frac{j}{n}\Big)$$

$$\le \prod_{j=0}^{m-1} e^{-j/n} = e^{-\sum_{j=0}^{m-1}\frac{j}{n}} = e^{-\frac{m(m-1)}{2n}} \quad .$$

7.7 Hashing

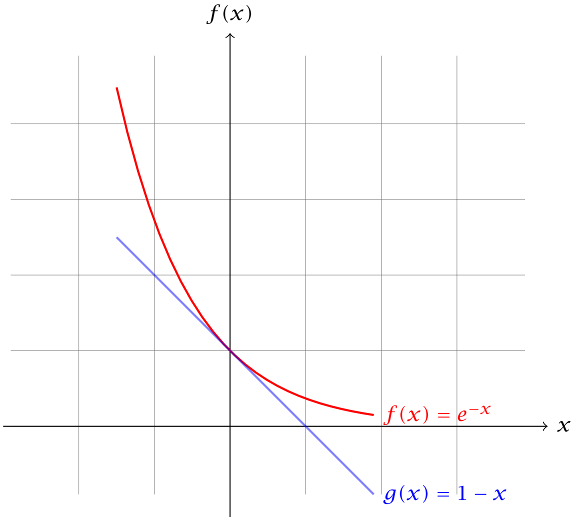
Proof.

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

$$\Pr[A_{m,n}] = \prod_{\ell=1}^{m} \frac{n-\ell+1}{n} = \prod_{j=0}^{m-1} \Big(1 - \frac{j}{n}\Big)$$

$$\leq \prod_{j=0}^{m-1} e^{-j/n} = e^{-\sum_{j=0}^{m-1} \frac{j}{n}} = e^{-\frac{m(m-1)}{2n}} \enspace .$$

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 $1 - x \le e^{-x}$ is derived by stopping the tayler-expansion of e^{-x} after the second term.

Resolving Collisions

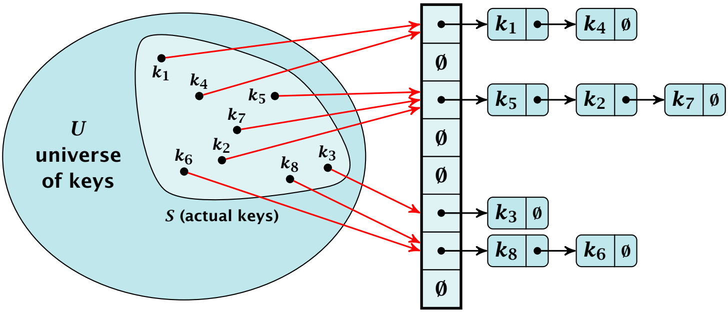
The methods for dealing with collisions can be classified into the two main types

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

Hashing with Chaining

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

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



7.7 Hashing

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.

7.7 Hashing

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.

7.7 Hashing

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.

7.7 Hashing

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.

7.7 Hashing

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.

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 $\alpha = \frac{m}{n}$.

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

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

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

Note that this result does not depend on the hash-function that is used.

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 event that i and j hash to the same position. Clearly, $\Pr[X_{ij} = 1] = 1/n$ for uniform hashing.

The expected successful search cost is

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

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 event that i and j hash to the same position. Clearly, $\Pr[X_{ij} = 1] = 1/n$ for uniform hashing.

The expected successful search cost is

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

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 event that i and j hash to the same position. Clearly, $\Pr[X_{ij} = 1] = 1/n$ for uniform hashing.

The expected successful search cost is

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

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 event that i and j hash to the same position. Clearly, $\Pr[X_{ij} = 1] = 1/n$ for uniform hashing.

The expected successful search cost is

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

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 event that i and j hash to the same position. Clearly, $\Pr[X_{ij} = 1] = 1/n$ for uniform hashing.

The expected successful search cost is

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

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 event that i and j hash to the same position. Clearly, $\Pr[X_{ij} = 1] = 1/n$ for uniform hashing.

The expected successful search cost is

keys before k_i

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

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 event that i and j hash to the same position. Clearly, $\Pr[X_{ij} = 1] = 1/n$ for uniform hashing.

The expected successful search cost is

$$\mathrm{E}\left[\frac{1}{m}\sum_{i=1}^{m}\underbrace{\left(1 + \sum_{j=i+1}^{m} X_{ij}\right)}_{\text{cost for key } k_i}\right]$$

Hashing with Chaining

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

Hashing with Chaining

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

Hashing with Chaining

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

$$=\frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}\frac{1}{n}\right)$$

Hashing with Chaining

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

Hashing with Chaining

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

Hashing with Chaining

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

Hashing with Chaining

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

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

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 $h(k,0)$,...,$h(k,n-1)$ form a permutation of $0,\ldots,n-1$.

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

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

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 $h(k, 0)$,...,$h(k, n-1)$ form a permutation of $0, \ldots, n-1$.

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

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

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 $h(k, 0)$,...,$h(k, n-1)$ form a permutation of $0, \ldots, n-1$.

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

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

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 $h(k,0)$,...,$h(k,n-1)$ form a permutation of 0, ..., $n-1$.

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

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

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 $h(k, 0)$,...,$h(k, n-1)$ form a permutation of $0, \ldots, n-1$.

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

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 $h(k,j)$:

- $h(k,i) = h(k) + i \mod n$. Linear probing.
- $h(k,i) = h(k) + c_1 i + c_2 i^2 \mod n$. Quadratic probing.
- $h(k,i) = h_1(k) + ih_2(k) \mod n$. Double hashing.

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; for quadratic probing c_1 and c_2 have to be chosen carefully).

Open Addressing

Choices for $h(k, j)$:

- $h(k, i) = h(k) + i \mod n$. Linear probing.
- $h(k, i) = h(k) + c_1 i + c_2 i^2 \mod n$. Quadratic probing.
- $h(k, i) = h_1(k) + i h_2(k) \mod n$. Double hashing.

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; for quadratic probing c_1 and c_2 have to be chosen carefully).

Open Addressing

Choices for $h(k, j)$:

- $h(k, i) = h(k) + i \mod n$. Linear probing.
- $h(k, i) = h(k) + c_1 i + c_2 i^2 \mod n$. Quadratic probing.
- $h(k, i) = h_1(k) + i h_2(k) \mod n$. Double hashing.

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; for quadratic probing c_1 and c_2 have to be chosen carefully).

Open Addressing

Choices for $h(k, j)$:

- $h(k, i) = h(k) + i \mod n$. Linear probing.
- $h(k, i) = h(k) + c_1 i + c_2 i^2 \mod n$. Quadratic probing.
- $h(k, i) = h_1(k) + i h_2(k) \mod n$. Double hashing.

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; 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 22

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

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

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

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 22

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 23

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

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

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

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 23

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 cash-miss.

Lemma 24

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

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

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

Double Hashing

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

Lemma 24

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

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

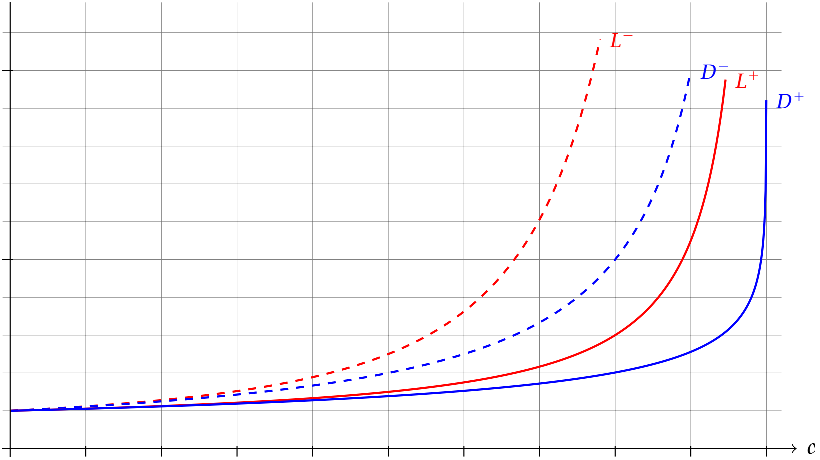
$$D^- \approx \frac{1}{1-\alpha}$$

7.7 Hashing

Some values:

α	Linear Probing		Quadratic Probing		Double Hashing	
	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

7.7 Hashing



Analysis of Idealized Open Address Hashing

Analysis of Idealized Open Address Hashing

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

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.

$$\Pr[A_1 \cap A_2 \cap \cdots \cap A_{i_1}]$$

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.

$$
\begin{aligned}
\Pr[A_1 \cap A_2 \cap \cdots \cap A_{i_1}] \\
&= \Pr[A_1] \cdot \Pr[A_2 \mid A_1] \cdot \Pr[A_3 \mid A_1 \cap A_2] \cdot \\
&\quad \ldots \cdot \Pr[A_{i_1} \mid A_1 \cap \cdots \cap A_{i-2}]
\end{aligned}
$$

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.

$$\Pr[A_1 \cap A_2 \cap \cdots \cap A_{i_1}]$$
$$= \Pr[A_1] \cdot \Pr[A_2 \mid A_1] \cdot \Pr[A_3 \mid A_1 \cap A_2] \cdot$$
$$\ldots \cdot \Pr[A_{i_1} \mid A_1 \cap \cdots \cap A_{i-2}]$$

$$\Pr[X \geq i]$$

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.

$$\begin{aligned}\Pr[A_1 \cap A_2 \cap \cdots \cap A_{i_1}] &\\ &= \Pr[A_1] \cdot \Pr[A_2 \mid A_1] \cdot \Pr[A_3 \mid A_1 \cap A_2] \cdot \\ &\quad \ldots \cdot \Pr[A_{i_1} \mid A_1 \cap \cdots \cap A_{i-2}]\end{aligned}$$

$$\Pr[X \geq i] = \frac{m}{n} \cdot \frac{m-1}{n-1} \cdot \frac{m-2}{n-2} \cdot \ldots \cdot \frac{m-i+2}{n-i+2}$$

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.

$$\begin{aligned}\Pr[A_1 \cap A_2 \cap \cdots \cap A_{i_1}] \\ &= \Pr[A_1] \cdot \Pr[A_2 \mid A_1] \cdot \Pr[A_3 \mid A_1 \cap A_2] \cdot \\ &\quad \ldots \cdot \Pr[A_{i_1} \mid A_1 \cap \cdots \cap A_{i-2}]\end{aligned}$$

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

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.

$$\Pr[A_1 \cap A_2 \cap \cdots \cap A_{i_1}]$$
$$= \Pr[A_1] \cdot \Pr[A_2 \mid A_1] \cdot \Pr[A_3 \mid A_1 \cap A_2] \cdot$$
$$\ldots \cdot \Pr[A_{i_1} \mid A_1 \cap \cdots \cap A_{i-2}]$$

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

Analysis of Idealized Open Address Hashing

$$\mathrm{E}[X]$$

Analysis of Idealized Open Address Hashing

$$E[X] = \sum_{i=1}^{\infty} \Pr[X \geq i]$$

Analysis of Idealized Open Address Hashing

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

Analysis of Idealized Open Address Hashing

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

Analysis of Idealized Open Address Hashing

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

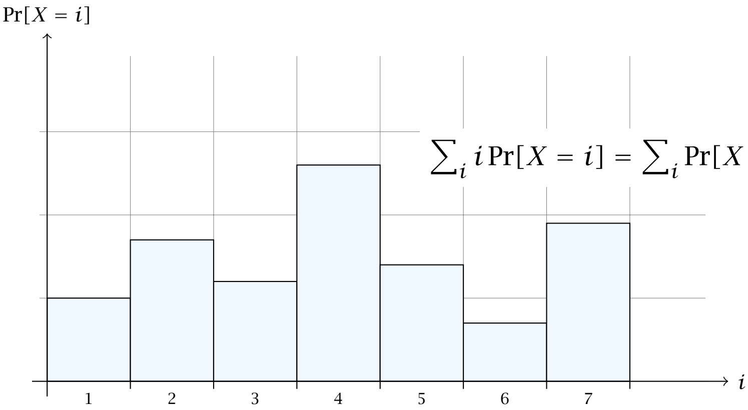
Analysis of Idealized Open Address Hashing

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

Analysis of Idealized Open Address Hashing

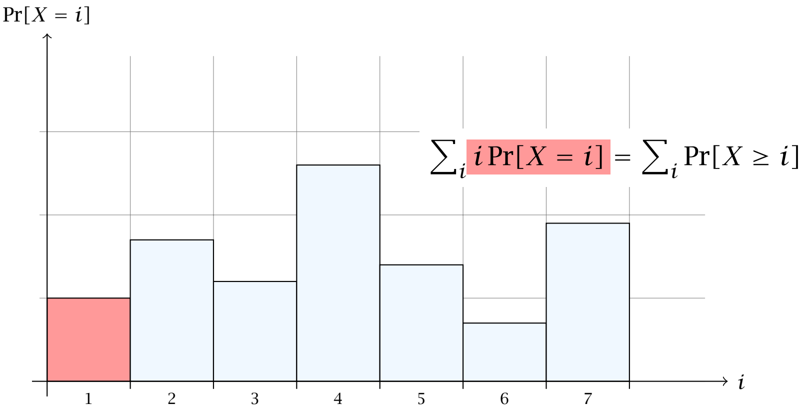
$$E[X] = \sum_{i=1}^{\infty} \Pr[X \geq i] \leq \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^{i} = \frac{1}{1-\alpha} \ .$$

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

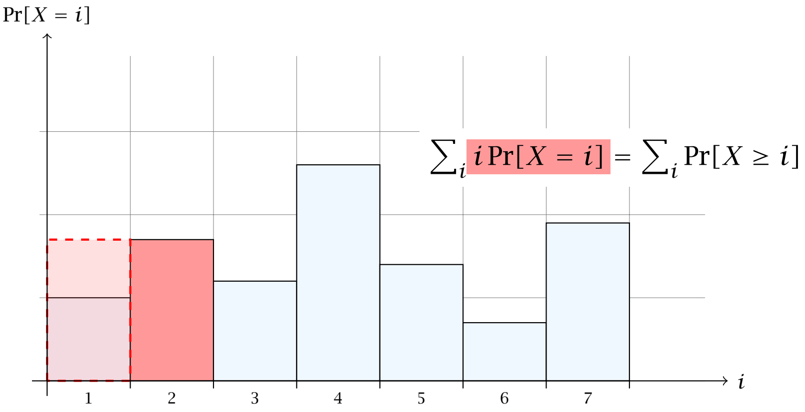


$$\sum_i i \Pr[X = i] = \sum_i \Pr[X \geq i]$$

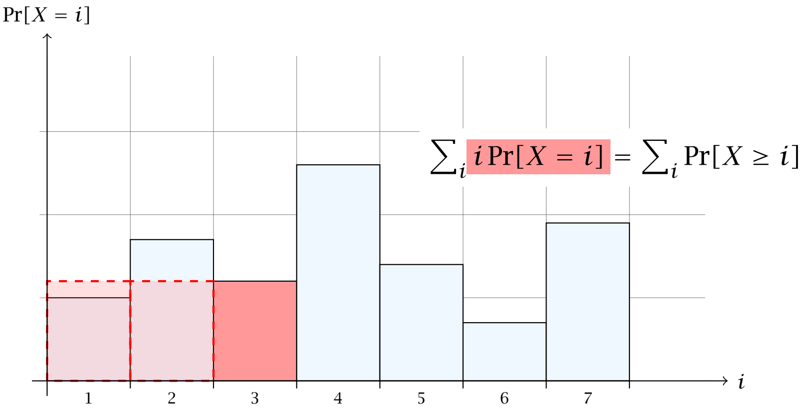
$i = 1$



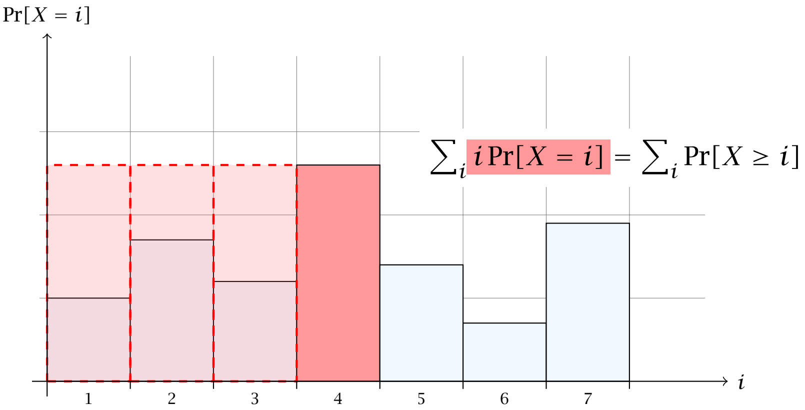
$i = 2$



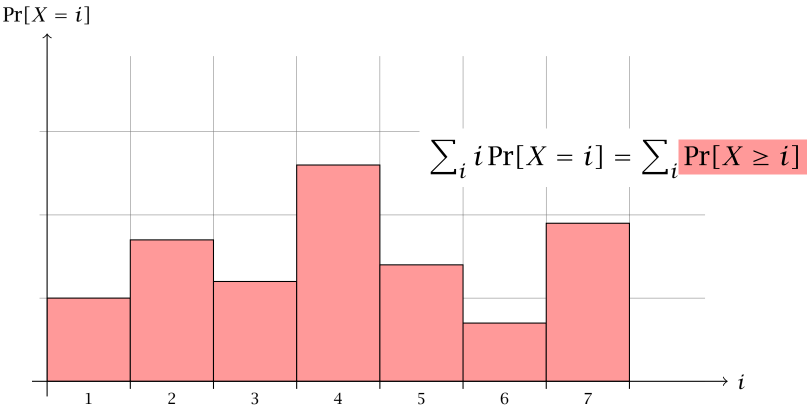
$i = 3$



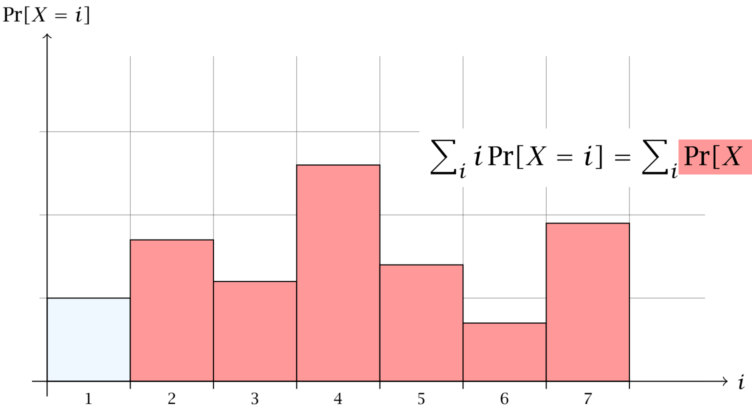
$i = 4$



$i = 1$

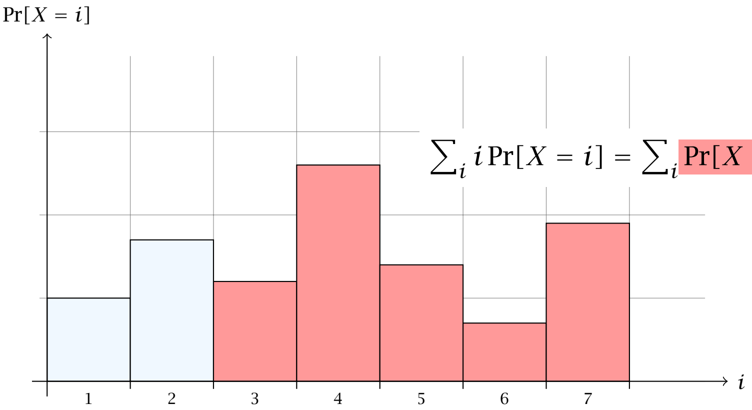


$i = 2$



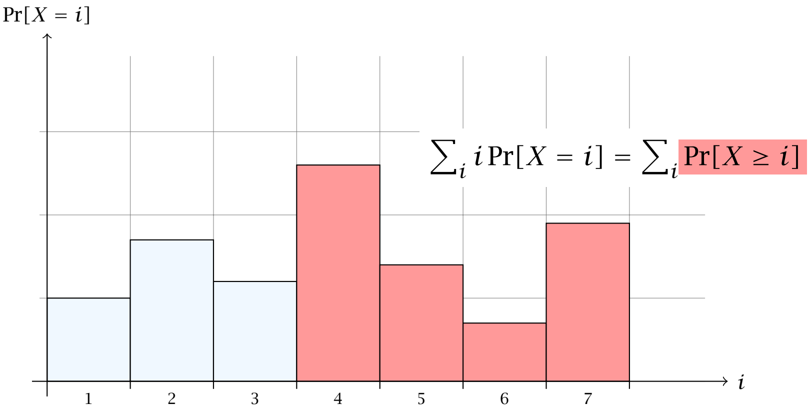
$$\sum_i i \Pr[X = i] = \sum_i \Pr[X \geq i]$$

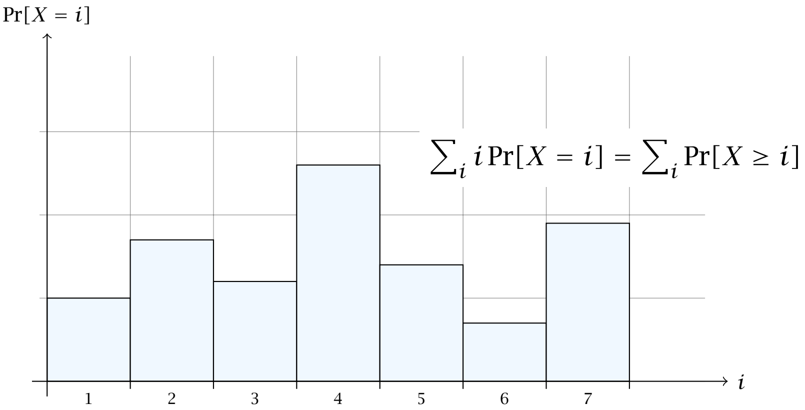
$i = 3$

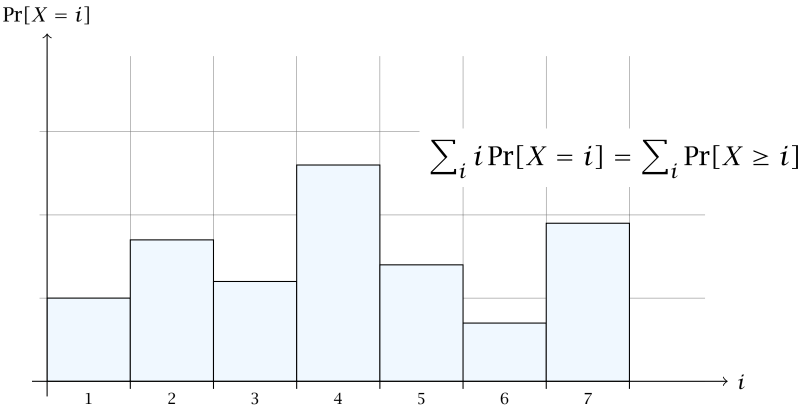


$\sum_i i \Pr[X = i] = \sum_i \Pr[X \geq i]$

$i = 4$




$\Pr[X = i]$
$\sum_i i \Pr[X = i] = \sum_i \Pr[X \geq i]$
1
2
3
4
5
6
7
i



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 $i = 1, 2, \ldots, j$)

Analysis of Idealized Open Address Hashing

Analysis of Idealized Open Address Hashing

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

Analysis of Idealized Open Address Hashing

The number of probes in a successful 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 $\frac{1}{1-i/n} = \frac{n}{n-i}$.

Analysis of Idealized Open Address Hashing

The number of probes in a successful 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 $\frac{1}{1-i/n} = \frac{n}{n-i}$.

$$\frac{1}{m}\sum_{i=0}^{m-1}\frac{n}{n-i}$$

Analysis of Idealized Open Address Hashing

The number of probes in a successful 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 $\frac{1}{1-i/n} = \frac{n}{n-i}$.

$$\frac{1}{m}\sum_{i=0}^{m-1}\frac{n}{n-i} = \frac{n}{m}\sum_{i=0}^{m-1}\frac{1}{n-i}$$

Analysis of Idealized Open Address Hashing

The number of probes in a successful 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 $\frac{1}{1-i/n} = \frac{n}{n-i}$.

$$\frac{1}{m}\sum_{i=0}^{m-1}\frac{n}{n-i} = \frac{n}{m}\sum_{i=0}^{m-1}\frac{1}{n-i} = \frac{1}{\alpha}\sum_{k=n-m+1}^{n}\frac{1}{k}$$

Analysis of Idealized Open Address Hashing

The number of probes in a successful 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 $\frac{1}{1-i/n} = \frac{n}{n-i}$.

$$\frac{1}{m}\sum_{i=0}^{m-1}\frac{n}{n-i} = \frac{n}{m}\sum_{i=0}^{m-1}\frac{1}{n-i} = \frac{1}{\alpha}\sum_{k=n-m+1}^{n}\frac{1}{k}$$

$$\leq \frac{1}{\alpha}\int_{n-m}^{n}\frac{1}{x}\mathrm{d}x$$

Analysis of Idealized Open Address Hashing

The number of probes in a successful 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 $\frac{1}{1-i/n} = \frac{n}{n-i}$.

$$\frac{1}{m}\sum_{i=0}^{m-1}\frac{n}{n-i} = \frac{n}{m}\sum_{i=0}^{m-1}\frac{1}{n-i} = \frac{1}{\alpha}\sum_{k=n-m+1}^{n}\frac{1}{k}$$

$$\leq \frac{1}{\alpha}\int_{n-m}^{n}\frac{1}{x}\mathrm{d}x = \frac{1}{\alpha}\ln\frac{n}{n-m}$$

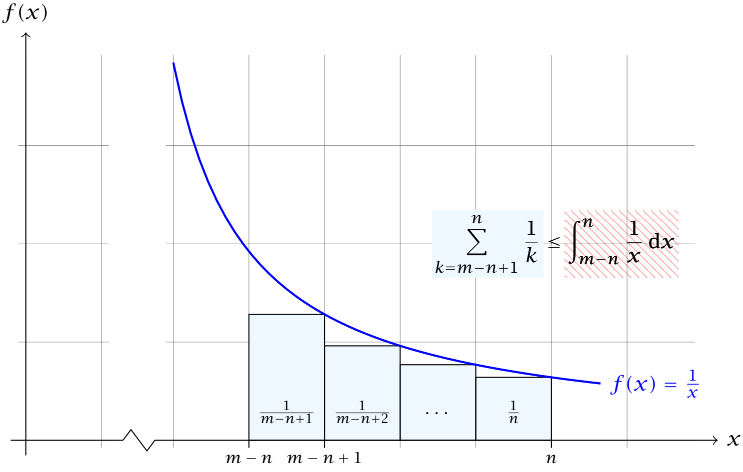
Analysis of Idealized Open Address Hashing

The number of probes in a successful 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 $\frac{1}{1-i/n} = \frac{n}{n-i}$.

$$\frac{1}{m}\sum_{i=0}^{m-1}\frac{n}{n-i} = \frac{n}{m}\sum_{i=0}^{m-1}\frac{1}{n-i} = \frac{1}{\alpha}\sum_{k=n-m+1}^{n}\frac{1}{k}$$

$$\leq \frac{1}{\alpha}\int_{n-m}^{n}\frac{1}{x}\mathrm{d}x = \frac{1}{\alpha}\ln\frac{n}{n-m} = \frac{1}{\alpha}\ln\frac{1}{1-\alpha} \;.$$



$f(x)$
x
$\sum_{k=m-n+1}^{n} \frac{1}{k} \le \int_{m-n}^{n} \frac{1}{x}\,\mathrm{d}x$
$f(x) = \frac{1}{x}$
$\frac{1}{m-n+1}$
$\frac{1}{m-n+2}$
$\dots$
$\frac{1}{n}$
$m-n$
$m-n+1$
n

7.7 Hashing

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.

7.7 Hashing

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.

7.7 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 $f : U \to [0, \ldots, n-1]$ 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 $\mathcal{H}$.

7.7 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 $f : U \to [0, \dots, n-1]$ 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 $\mathcal{H}$.

7.7 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 $f : U \to [0, \dots, n-1]$ 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 $\mathcal{H}$.

7.7 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 $f : U \to [0, \ldots, n-1]$ 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 $\mathcal{H}$.

7.7 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 $f : U \to [0, \ldots, n-1]$ 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 $\mathcal{H}$.

7.7 Hashing

Definition 25

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

7.7 Hashing

Definition 25

A class $\mathcal{H}$ of hash-functions from the universe U into the set $\{0, \ldots, n-1\}$ 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 $\Pr[h(u_1) = h(u_2)] = \frac{1}{n}$.

7.7 Hashing

Definition 26

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

- For any key $u \in U$, and $t \in \{0, \ldots, n-1\}$ $\Pr[h(u) = t] = \frac{1}{n}$, 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:

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

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

This requirement clearly implies a universal hash-function.

7.7 Hashing

Definition 26

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:

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

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

This requirement clearly implies a universal hash-function.

7.7 Hashing

Definition 27

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 $t_1, \dots, t_\ell$:

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

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

7.7 Hashing

Definition 28

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

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

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

7.7 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 29

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 $\{0, \dots, n-1\}$.

7.7 Hashing

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

Define

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

Lemma 29

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 $\{0, \ldots, n-1\}$.

7.7 Hashing

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

Define

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

Lemma 29

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 $\{0, \ldots, n-1\}$.

7.7 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 29

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 $\{0, \dots, n-1\}$.

7.7 Hashing

Proof.

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

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

7.7 Hashing

Proof.

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

7.7 Hashing

Proof.

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

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

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 (KÃČÂűrper) and, hence, has no zero divisors (nullteilerfrei).

7.7 Hashing

Proof.

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

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

  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 (KÃČÂűrper) and, hence, has no zero divisors (nullteilerfrei).

7.7 Hashing

Proof.

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

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

  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 (KÃČÂűrper) and, hence, has no zero divisors (nullteilerfrei).

7.7 Hashing

Proof.

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

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

  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 (KÃČÂűrper) and, hence, has no zero divisors (nullteilerfrei).

- The hash-function does not generate collisions before the $\pmod{n}$-operation. Furthermore, every choice (a, b) is mapped to different hash-values $t_x := h_{a,b}(x)$ and $t_y := h_{a,b}(y)$.

- The hash-function does not generate collisions before the $\pmod{n}$-operation. Furthermore, every choice (a, b) is mapped to different hash-values $t_x := h_{a,b}(x)$ and $t_y := h_{a,b}(y)$.

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

- The hash-function does not generate collisions before the $(\bmod n)$-operation. Furthermore, every choice (a, b) is mapped to different hash-values $t_x := h_{a,b}(x)$ and $t_y := h_{a,b}(y)$.

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

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

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

- The hash-function does not generate collisions before the $\pmod{n}$-operation. Furthermore, every choice (a, b) is mapped to different hash-values $t_x := h_{a,b}(x)$ and $t_y := h_{a,b}(y)$.

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

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

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

- The hash-function does not generate collisions before the $\pmod{n}$-operation. Furthermore, every choice (a, b) is mapped to different hash-values $t_x := h_{a,b}(x)$ and $t_y := h_{a,b}(y)$.

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

$$\begin{aligned} t_x &\equiv ax + b && \pmod{p} \\ t_y &\equiv ay + b && \pmod{p} \end{aligned}$$

$$\begin{aligned} t_x - t_y &\equiv a(x - y) && \pmod{p} \\ t_y &\equiv ay + b && \pmod{p} \end{aligned}$$

$$\begin{aligned} a &\equiv (t_x - t_y)(x - y)^{-1} && \pmod{p} \\ b &\equiv ay - t_y && \pmod{p} \end{aligned}$$

7.7 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 $(\bmod n)$-operation) as choosing a pair (t_x, t_y), $t_x \neq t_y$ uniformly at random.

What happens when we do the $(\bmod n)$ 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.

7.7 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 $(\bmod n)$-operation) as choosing a pair (t_x, t_y), $t_x \neq t_y$ uniformly at random.

What happens when we do the $(\bmod n)$ 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.

7.7 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 $(\bmod n)$-operation) as choosing a pair (t_x, t_y), $t_x \neq t_y$ uniformly at random.

What happens when we do the $(\bmod n)$ operation?

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

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

7.7 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 $(\bmod n)$-operation) as choosing a pair (t_x, t_y), $t_x \neq t_y$ uniformly at random.

What happens when we do the $(\bmod n)$ operation?

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

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

7.7 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 $(\bmod n)$-operation) as choosing a pair (t_x, t_y), $t_x \neq t_y$ uniformly at random.

What happens when we do the $(\bmod n)$ 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.

7.7 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 $(\bmod n)$-operation) as choosing a pair (t_x, t_y), $t_x \neq t_y$ uniformly at random.

What happens when we do the $(\bmod n)$ 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.

7.7 Hashing

As $t_y \neq t_x$ there are

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

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

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

7.7 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 $\frac{1}{n}$.

7.7 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 $\frac{1}{n}$.

7.7 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 $\frac{1}{n}$.

7.7 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 $\frac{1}{n}$.

7.7 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 $\frac{1}{n}$.

7.7 Hashing

7.7 Hashing

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

$$\Pr_{t_x \neq t_y \in \mathbb{Z}_p^2}\left[\begin{array}{c} t_x \bmod n = h_1 \\ \wedge \\ t_y \bmod n = h_2 \end{array}\right]$$

7.7 Hashing

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

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

7.7 Hashing

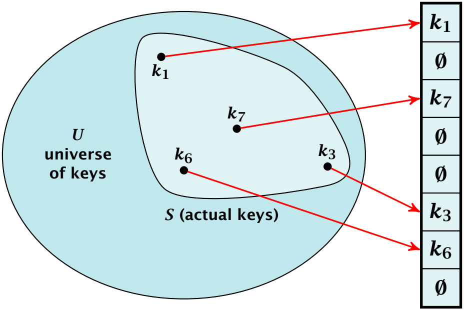
It is also possible to show that $\mathcal{H}$ is an (almost) pairwise independent class of hash-functions.

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

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 $\lceil \frac{p}{n} \rceil$.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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 $\sum_j m_j^2$.

$$\mathrm{E}\Big[\sum_j m_j^2\Big]$$

Perfect Hashing

The total memory that is required by all hash-tables is $\sum_j m_j^2$.

$$\mathrm{E}\Big[\sum_j m_j^2\Big] = \mathrm{E}\Big[2\sum_j \binom{m_j}{2} + \sum_j m_j\Big]$$

Perfect Hashing

The total memory that is required by all hash-tables is $\sum_j m_j^2$.

$$\begin{aligned}
\mathrm{E}\Big[\sum_j m_j^2\Big] &= \mathrm{E}\Big[2\sum_j \binom{m_j}{2} + \sum_j m_j\Big] \\
&= 2\,\mathrm{E}\Big[\sum_j \binom{m_j}{2}\Big] + \mathrm{E}\Big[\sum_j m_j\Big]
\end{aligned}$$

Perfect Hashing

The total memory that is required by all hash-tables is $\sum_j m_j^2$.

$$
\begin{aligned}
\mathrm{E}\Big[\sum_j m_j^2\Big] &= \mathrm{E}\Big[2\sum_j \binom{m_j}{2} + \sum_j m_j\Big] \\
&= 2\,\mathrm{E}\Big[\sum_j \binom{m_j}{2}\Big] + \mathrm{E}\Big[\sum_j m_j\Big]
\end{aligned}
$$

The first expectation is simply the expected number of collisions, for the first level.

Perfect Hashing

The total memory that is required by all hash-tables is $\sum_j m_j^2$.

$$
\begin{aligned}
\mathrm{E}\Big[\sum_j m_j^2\Big] &= \mathrm{E}\Big[2\sum_j \binom{m_j}{2} + \sum_j m_j\Big] \\
&= 2\,\mathrm{E}\Big[\sum_j \binom{m_j}{2}\Big] + \mathrm{E}\Big[\sum_j m_j\Big]
\end{aligned}
$$

The first expectation is simply the expected number of collisions, for the first level.

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

Perfect Hashing

We need only $\mathcal{O}(m)$ time to construct a hash-function h with $\sum_j m_j^2 = \mathcal{O}(4m)$.

Then we construct a hash-table h_j for every bucket. This takes expected time $\mathcal{O}(m_j)$ for every bucket.

We only need that the hash-function is universal!!!

Cuckoo Hashing

Goal:
Try to generate a perfect hash-table (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

Goal:

Try to generate a perfect hash-table (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

Goal:

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

- Two hash-tables $T_1[0,\ldots,n-1]$ and $T_2[0,\ldots,n-1]$, 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.

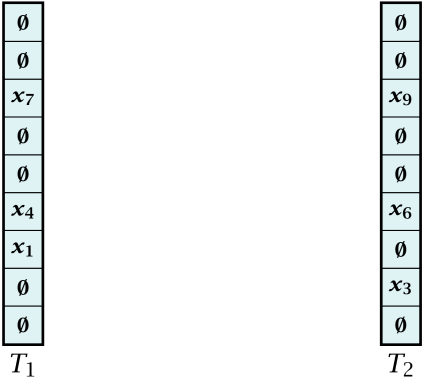
Cuckoo Hashing

Goal:
Try to generate a perfect hash-table (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.

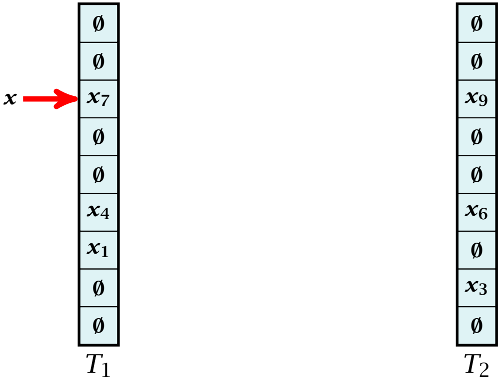
Cuckoo Hashing

Insert:



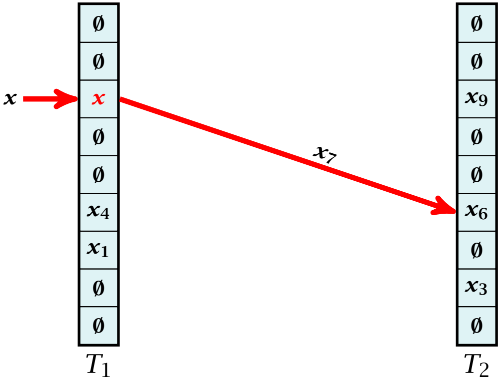
Cuckoo Hashing

Insert:



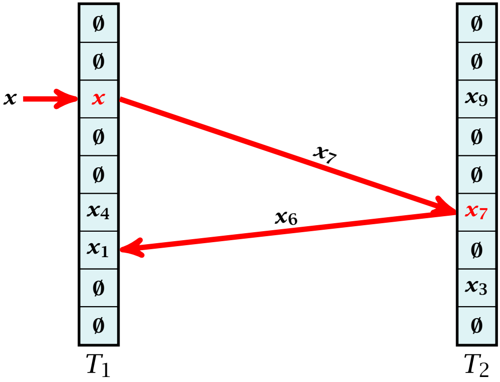
Cuckoo Hashing

Insert:



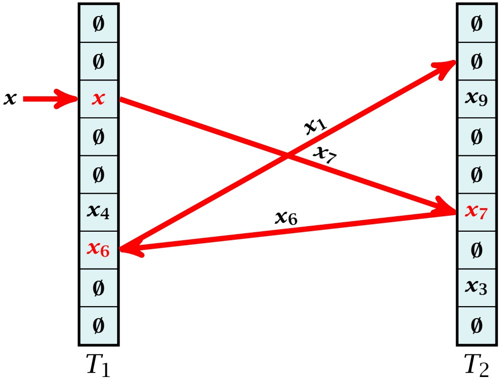
Cuckoo Hashing

Insert:



Cuckoo Hashing

Insert:



Cuckoo Hashing

Algorithm 16 Cuckoo-Insert(x)

```
if T_1[h_1(x)] = x ∨ T_2[h_2(x)] = x then return
steps ← 1
while steps ≤ maxsteps do
        exchange x and T_1[h_1(x)]
        if x = null then return
        exchange x and T_2[h_2(x)]
        if x = null then return
rehash() // change table-size and rehash everything
Cuckoo-Insert(x)
```

Cuckoo Hashing

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 $\mathrm{maxsteps}$ steps).

Formally what is the probability to enter an infinite loop that touches ℓ different keys (apart from x)?

Cuckoo Hashing

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 ℓ different keys (apart from x)?

Cuckoo Hashing

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 $\mathrm{maxsteps}$ steps).

Formally what is the probability to enter an infinite loop that touches ℓ different keys (apart from x)?

Cuckoo Hashing

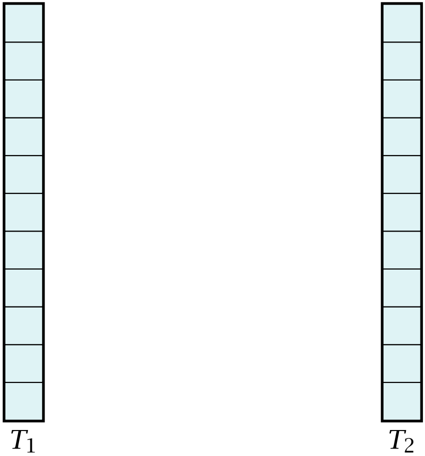
What is the expected time for an insert-operation?

We first analyze the probability that we end-up in an infinite loop (that is then terminated after $\mathrm{maxsteps}$ steps).

Formally what is the probability to enter an infinite loop that touches ℓ different keys (apart from x)?

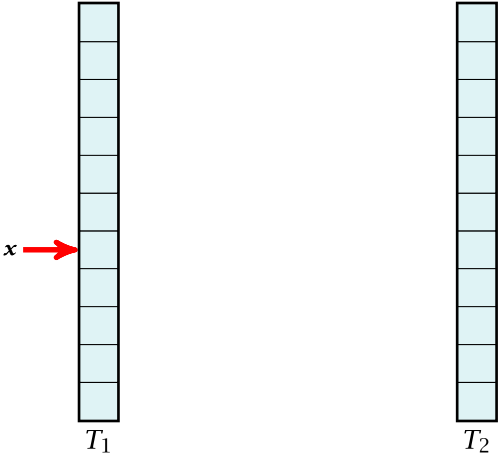
Cuckoo Hashing

Insert:



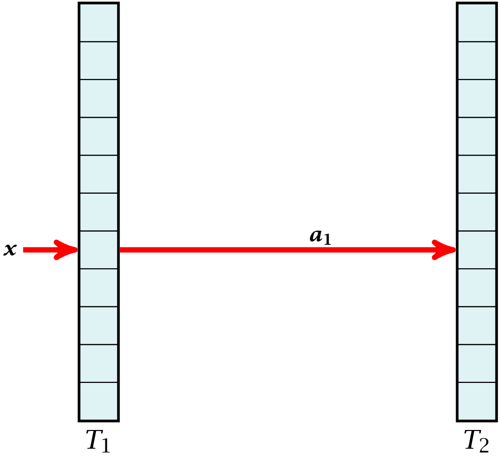
Cuckoo Hashing

Insert:



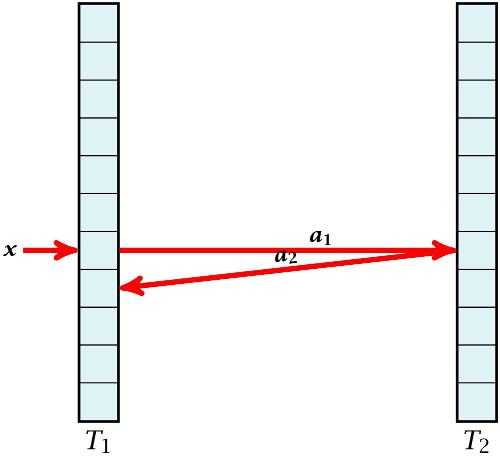
Cuckoo Hashing

Insert:



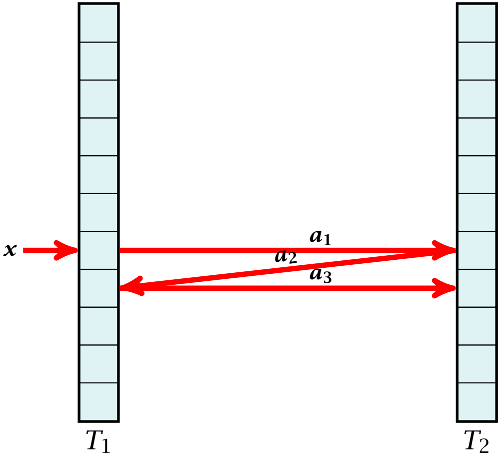
Cuckoo Hashing

Insert:



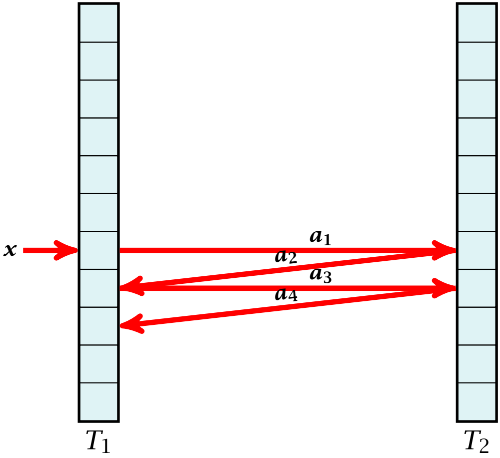
Cuckoo Hashing

Insert:



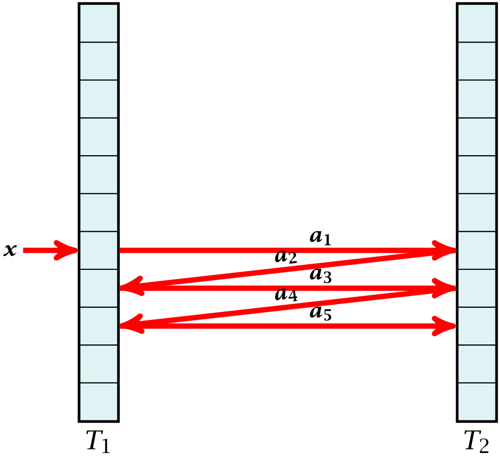
Cuckoo Hashing

Insert:



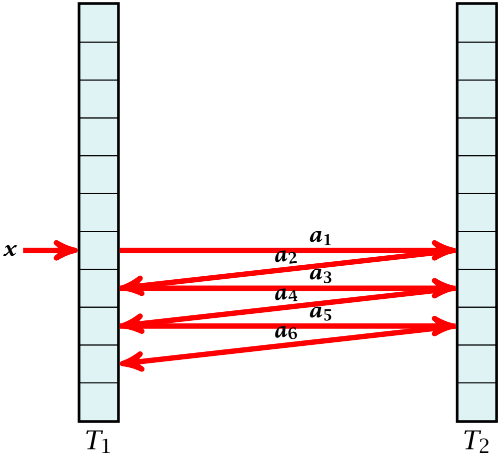
Cuckoo Hashing

Insert:



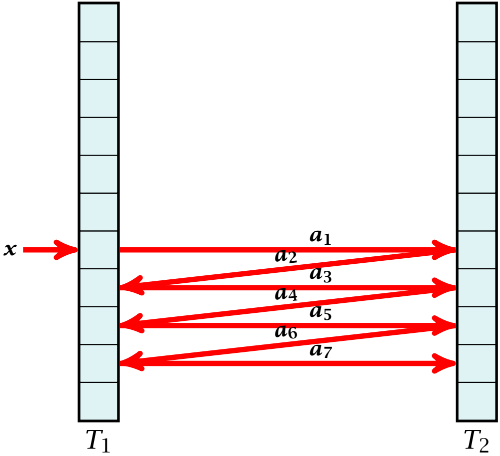
Cuckoo Hashing

Insert:



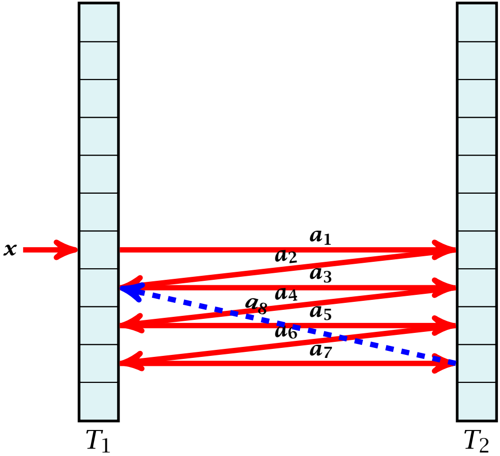
Cuckoo Hashing

Insert:



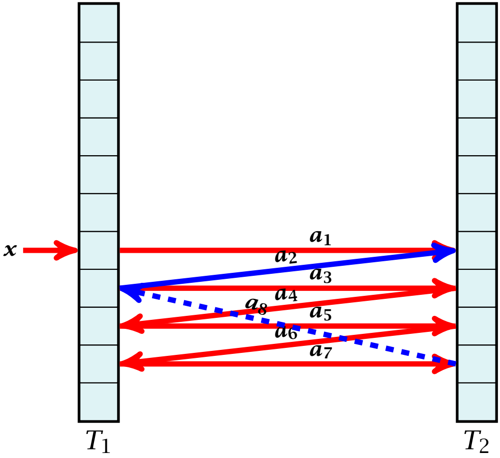
Cuckoo Hashing

Insert:



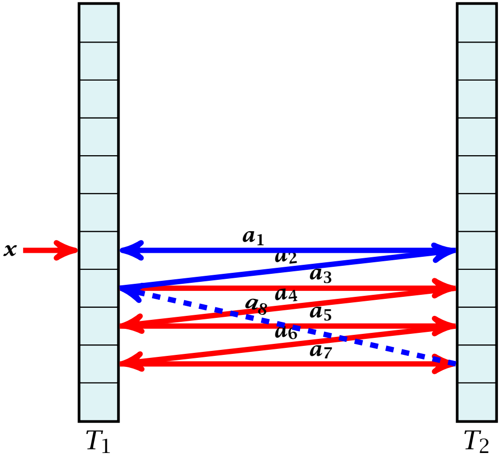
Cuckoo Hashing

Insert:



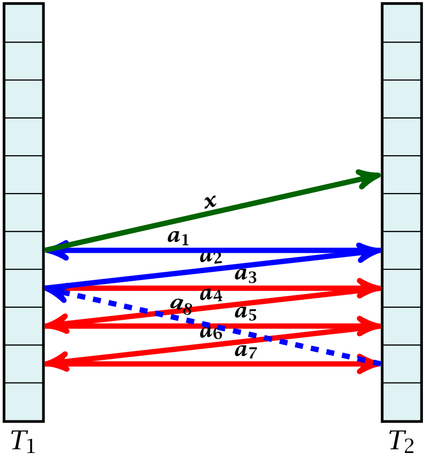
Cuckoo Hashing

Insert:



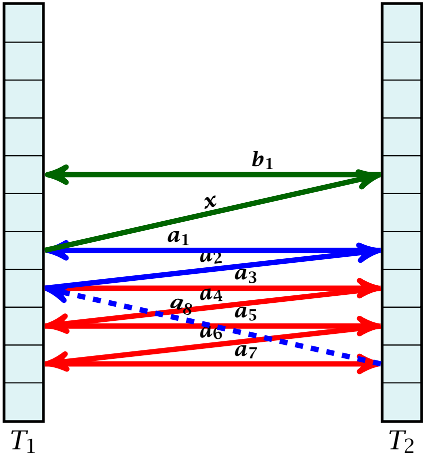
Cuckoo Hashing

Insert:



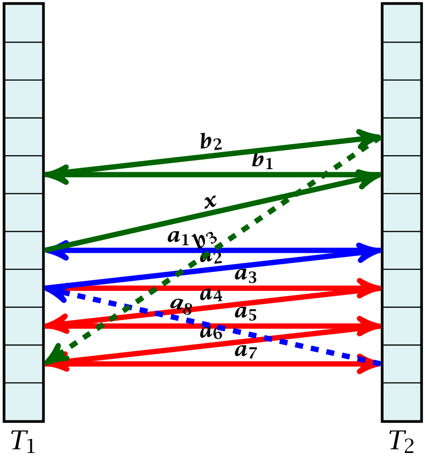
Cuckoo Hashing

Insert:



Cuckoo Hashing

Insert:



Cuckoo Hashing

A cycle-structure is defined by

- ℓ_a keys $a_1, a_2, \dots, a_{\ell_a}$, $\ell_a \geq 2$.
- An index $j_a \in \{1, \dots, \ell_a - 1\}$ that defines how much the last item a_{ℓ_a} "jumps back" in the sequence.
- ℓ_b keys $b_1, b_2, \dots, b_{\ell_b}$. $b \geq 0$.
- An index $j_b \in \{1, \dots, \ell_a + \ell_b\}$ that defines how much the last item b_{ℓ_b} "jumps back" in the sequence.
- An assignment of positions for the keys in both tables. Formally we have positions $p_1, \dots, p_{\ell_a}$ and $p'_1, \dots, p'_{\ell_b}$.
- The size of a cycle-structure is defined as $\ell_a + \ell_b$.

Cuckoo Hashing

A cycle-structure is defined by

- ℓ_a keys $a_1, a_2, \dots a_{\ell_a}$, $\ell_a \geq 2$,
- An index $j_a \in \{1 \dots, \ell_a - 1\}$ that defines how much the last item a_{ℓ_a} "jumps back" in the sequence.
- ℓ_b keys $b_1, b_2, \dots b_{\ell_b}$. $b \geq 0$.
- An index $j_b \in \{1 \dots, \ell_a + \ell_b\}$ that defines how much the last item b_{ℓ_b} "jumps back" in the sequence.
- An assignment of positions for the keys in both tables. Formally we have positions $p_1, \dots, p_{\ell_a}$, and $p'_1, \dots, p'_{\ell_b}$.
- The size of a cycle-structure is defined as $\ell_a + \ell_b$.

Cuckoo Hashing

A cycle-structure is defined by

- ℓ_a keys $a_1, a_2, \dots a_{\ell_a}$, $\ell_a \geq 2$,
- An index $j_a \in \{1 \dots, \ell_a - 1\}$ that defines how much the last item a_{ℓ_a} "jumps back" in the sequence.
- ℓ_b keys $b_1, b_2, \dots b_{\ell_b}$. $b \geq 0$.
- An index $j_b \in \{1 \dots, \ell_a + \ell_b\}$ that defines how much the last item b_{ℓ_b} "jumps back" in the sequence.
- An assignment of positions for the keys in both tables. Formally we have positions $p_1, \dots, p_{\ell_a}$, and $p'_1, \dots, p'_{\ell_b}$.
- The size of a cycle-structure is defined as $\ell_a + \ell_b$.

Cuckoo Hashing

A cycle-structure is defined by

- ℓ_a keys $a_1, a_2, \dots a_{\ell_a}$, $\ell_a \geq 2$,
- An index $j_a \in \{1 \dots, \ell_a - 1\}$ that defines how much the last item a_{ℓ_a} "jumps back" in the sequence.
- ℓ_b keys $b_1, b_2, \dots b_{\ell_b}$. $b \geq 0$.
- An index $j_b \in \{1 \dots, \ell_a + \ell_b\}$ that defines how much the last item b_{ℓ_b} "jumps back" in the sequence.
- An assignment of positions for the keys in both tables. Formally we have positions $p_1, \dots, p_{\ell_a}$, and $p'_1, \dots, p'_{\ell_b}$.
- The size of a cycle-structure is defined as $\ell_a + \ell_b$.

Cuckoo Hashing

A cycle-structure is defined by

- ℓ_a keys $a_1, a_2, \ldots a_{\ell_a}$, $\ell_a \geq 2$,
- An index $j_a \in \{1 \ldots, \ell_a - 1\}$ that defines how much the last item a_{ℓ_a} "jumps back" in the sequence.
- ℓ_b keys $b_1, b_2, \ldots b_{\ell_b}$. $b \geq 0$.
- An index $j_b \in \{1 \ldots, \ell_a + \ell_b\}$ that defines how much the last item b_{ℓ_b} "jumps back" in the sequence.
- An assignment of positions for the keys in both tables. Formally we have positions $p_1, \ldots, p_{\ell_a}$, and $p'_1, \ldots, p'_{\ell_b}$.
- The size of a cycle-structure is defined as $\ell_a + \ell_b$.

Cuckoo Hashing

A cycle-structure is defined by

- ℓ_a keys $a_1, a_2, \ldots a_{\ell_a}$, $\ell_a \geq 2$,
- An index $j_a \in \{1 \ldots, \ell_a - 1\}$ that defines how much the last item a_{ℓ_a} "jumps back" in the sequence.
- ℓ_b keys $b_1, b_2, \ldots b_{\ell_b}$. $b \geq 0$.
- An index $j_b \in \{1 \ldots, \ell_a + \ell_b\}$ that defines how much the last item b_{ℓ_b} "jumps back" in the sequence.
- An assignment of positions for the keys in both tables. Formally we have positions $p_1, \ldots, p_{\ell_a}$, and $p'_1, \ldots, p'_{\ell_b}$.
- The size of a cycle-structure is defined as $\ell_a + \ell_b$.

Cuckoo Hashing

A cycle-structure is defined by

- ℓ_a keys $a_1, a_2, \dots a_{\ell_a}$, $\ell_a \geq 2$,
- An index $j_a \in \{1 \dots, \ell_a - 1\}$ that defines how much the last item a_{ℓ_a} "jumps back" in the sequence.
- ℓ_b keys $b_1, b_2, \dots b_{\ell_b}$. $b \geq 0$.
- An index $j_b \in \{1 \dots, \ell_a + \ell_b\}$ that defines how much the last item b_{ℓ_b} "jumps back" in the sequence.
- An assignment of positions for the keys in both tables. Formally we have positions $p_1, \dots, p_{\ell_a}$, and $p'_1, \dots, p'_{\ell_b}$.
- The size of a cycle-structure is defined as $\ell_a + \ell_b$.

Cuckoo Hashing

We say a cycle-structure is active for key x if the hash-functions are chosen in such a way that the hash-function results match the pre-defined key-positions.

- $h_1(x) = h_1(a_1) = p_1$
- $h_2(a_1) = h_2(a_2) = p_2$
- $h_1(a_2) = h_1(a_3) = p_3$
- ...
- if ℓ_a is even then $h_1(a_\ell) = p_{s_a}$, otw. $h_2(a_\ell) = p_{s_a}$
- $h_2(x) = h_2(b_1) = p'_1$
- $h_1(b_1) = h_1(b_2) = p'_2$
- ...

Cuckoo Hashing

We say a cycle-structure is active for key x if the hash-functions are chosen in such a way that the hash-function results match the pre-defined key-positions.

- $h_1(x) = h_1(a_1) = p_1$
- $h_2(a_1) = h_2(a_2) = p_2$
- $h_1(a_2) = h_1(a_3) = p_3$
- $\dots$
- if ℓ_a is even then $h_1(a_\ell) = p_{s_a}$, otw. $h_2(a_\ell) = p_{s_a}$
- $h_2(x) = h_2(b_1) = p'_1$
- $h_1(b_1) = h_1(b_2) = p'_2$
- $\dots$

Cuckoo Hashing

We say a cycle-structure is active for key x if the hash-functions are chosen in such a way that the hash-function results match the pre-defined key-positions.

- $h_1(x) = h_1(a_1) = p_1$
- $h_2(a_1) = h_2(a_2) = p_2$
- $h_1(a_2) = h_1(a_3) = p_3$
- $\dots$
- if ℓ_a is even then $h_1(a_\ell) = p_{s_a}$, otw. $h_2(a_\ell) = p_{s_a}$
- $h_2(x) = h_2(b_1) = p'_1$
- $h_1(b_1) = h_1(b_2) = p'_2$
- $\dots$

Cuckoo Hashing

We say a cycle-structure is active for key x if the hash-functions are chosen in such a way that the hash-function results match the pre-defined key-positions.

- $h_1(x) = h_1(a_1) = p_1$
- $h_2(a_1) = h_2(a_2) = p_2$
- $h_1(a_2) = h_1(a_3) = p_3$
- $\dots$
- if ℓ_a is even then $h_1(a_\ell) = p_{s_a}$, otw. $h_2(a_\ell) = p_{s_a}$
- $h_2(x) = h_2(b_1) = p_1'$
- $h_1(b_1) = h_1(b_2) = p_2'$
- $\dots$

Cuckoo Hashing

We say a cycle-structure is active for key x if the hash-functions are chosen in such a way that the hash-function results match the pre-defined key-positions.

- $h_1(x) = h_1(a_1) = p_1$
- $h_2(a_1) = h_2(a_2) = p_2$
- $h_1(a_2) = h_1(a_3) = p_3$
- $\dots$
- if ℓ_a is even then $h_1(a_\ell) = p_{s_a}$, otw. $h_2(a_\ell) = p_{s_a}$
- $h_2(x) = h_2(b_1) = p'_1$
- $h_1(b_1) = h_1(b_2) = p'_2$
- $\dots$

Cuckoo Hashing

We say a cycle-structure is active for key x if the hash-functions are chosen in such a way that the hash-function results match the pre-defined key-positions.

- $h_1(x) = h_1(a_1) = p_1$
- $h_2(a_1) = h_2(a_2) = p_2$
- $h_1(a_2) = h_1(a_3) = p_3$
- $\dots$
- if ℓ_a is even then $h_1(a_\ell) = p_{s_a}$, otw. $h_2(a_\ell) = p_{s_a}$
- $h_2(x) = h_2(b_1) = p'_1$
- $h_1(b_1) = h_1(b_2) = p'_2$
- $\dots$

Cuckoo Hashing

We say a cycle-structure is active for key x if the hash-functions are chosen in such a way that the hash-function results match the pre-defined key-positions.

- $h_1(x) = h_1(a_1) = p_1$
- $h_2(a_1) = h_2(a_2) = p_2$
- $h_1(a_2) = h_1(a_3) = p_3$
- $\dots$
- if ℓ_a is even then $h_1(a_\ell) = p_{s_a}$, otw. $h_2(a_\ell) = p_{s_a}$
- $h_2(x) = h_2(b_1) = p'_1$
- $h_1(b_1) = h_1(b_2) = p'_2$
- $\dots$

Cuckoo Hashing

We say a cycle-structure is active for key x if the hash-functions are chosen in such a way that the hash-function results match the pre-defined key-positions.

- $h_1(x) = h_1(a_1) = p_1$
- $h_2(a_1) = h_2(a_2) = p_2$
- $h_1(a_2) = h_1(a_3) = p_3$
- $\dots$
- if ℓ_a is even then $h_1(a_\ell) = p_{s_a}$, otw. $h_2(a_\ell) = p_{s_a}$
- $h_2(x) = h_2(b_1) = p'_1$
- $h_1(b_1) = h_1(b_2) = p'_2$
- $\dots$

Cuckoo Hashing

We say a cycle-structure is active for key x if the hash-functions are chosen in such a way that the hash-function results match the pre-defined key-positions.

- $h_1(x) = h_1(a_1) = p_1$
- $h_2(a_1) = h_2(a_2) = p_2$
- $h_1(a_2) = h_1(a_3) = p_3$
- ...
- if ℓ_a is even then $h_1(a_\ell) = p_{s_a}$, otw. $h_2(a_\ell) = p_{s_a}$
- $h_2(x) = h_2(b_1) = p_1'$
- $h_1(b_1) = h_1(b_2) = p_2'$
- ...

Cuckoo Hashing

Observation If we end up in an infinite loop there must exist a cycle-structure that is active for x.

Cuckoo Hashing

A cycle-structure is defined without knowing the hash-functions.

Whether a cycle-structure is active for key x depends on the hash-functions.

Lemma 30

A given cycle-structure of size s is active for key x with probability at most

$$\left(\frac{\mu}{n}\right)^{2(s+1)} ,$$

if we use $(\mu, s+1)$-independent hash-functions.

Cuckoo Hashing

A cycle-structure is defined without knowing the hash-functions.

Whether a cycle-structure is active for key x depends on the hash-functions.

Lemma 30

A given cycle-structure of size s is active for key x with probability at most

$$\left(\frac{\mu}{n}\right)^{2(s+1)},$$

if we use $(\mu, s+1)$-independent hash-functions.

Cuckoo Hashing

A cycle-structure is defined without knowing the hash-functions.

Whether a cycle-structure is active for key x depends on the hash-functions.

Lemma 30

A given cycle-structure of size s is active for key x with probability at most

$$\left(\frac{\mu}{n}\right)^{2(s+1)} \quad ,$$

if we use $(\mu, s+1)$-independent hash-functions.

Cuckoo Hashing

Proof.

All positions are fixed by the cycle-structure. Therefore we ask for the probability of mapping $s+1$ keys (the a-keys, the b-keys and x) to pre-specified positions in T_1, **and** to pre-specified positions in T_2.

The probability is

$$\left(\frac{\mu}{n}\right)^{s+1} \cdot \left(\frac{\mu}{n}\right)^{s+1},$$

since h_1 and h_2 are chosen independently. □

Cuckoo Hashing

Proof.

All positions are fixed by the cycle-structure. Therefore we ask for the probability of mapping $s+1$ keys (the a-keys, the b-keys and x) to pre-specified positions in T_1, **and** to pre-specified positions in T_2.

The probability is

$$\left(\frac{\mu}{n}\right)^{s+1} \cdot \left(\frac{\mu}{n}\right)^{s+1} ,$$

since h_1 and h_2 are chosen independently. □

Cuckoo Hashing

The number of cycle-structures of size s is small:

- There are at most s ways to choose ℓ_1. This fixes ℓ_2.
- There are at most s^2 ways to choose j_1 and j_2.
- There are at most m^s possibilities to choose the keys $a_1, \dots, a_{\ell_1}$ and $b_1, \dots, b_{\ell_2}$.
- There are at most n^s choices for choosing the positions $p_1, \dots, p_{\ell_1}$ and $p'_1, \dots, p'_{\ell_2}$.

Cuckoo Hashing

The number of cycle-structures of size s is small:

- There are at most s ways to choose ℓ_a. This fixes ℓ_b.
- There are at most s^2 ways to choose j_a, and j_b.
- There are at most m^s possibilities to choose the keys $a_1, \dots, a_{\ell_a}$ and $b_1, \dots, b_{\ell_b}$.
- There are at most n^s choices for choosing the positions $p_1, \dots, p_{\ell_a}$ and $p'_1, \dots, p'_{\ell_a}$.

Cuckoo Hashing

The number of cycle-structures of size s is small:

- There are at most s ways to choose ℓ_a. This fixes ℓ_b.
- There are at most s^2 ways to choose j_a, and j_b.
- There are at most m^s possibilities to choose the keys $a_1, \dots, a_{\ell_a}$ and $b_1, \dots, b_{\ell_b}$.
- There are at most n^s choices for choosing the positions $p_1, \dots, p_{\ell_a}$ and $p'_1, \dots, p'_{\ell_a}$.

Cuckoo Hashing

The number of cycle-structures of size s is small:

- There are at most s ways to choose ℓ_a. This fixes ℓ_b.
- There are at most s^2 ways to choose j_a, and j_b.
- There are at most m^s possibilities to choose the keys $a_1, \dots, a_{\ell_a}$ and $b_1, \dots, b_{\ell_b}$.
- There are at most n^s choices for choosing the positions $p_1, \dots, p_{\ell_a}$ and $p'_1, \dots, p'_{\ell_a}$.

Cuckoo Hashing

The number of cycle-structures of size s is small:

- There are at most s ways to choose ℓ_a. This fixes ℓ_b.
- There are at most s^2 ways to choose j_a, and j_b.
- There are at most m^s possibilities to choose the keys $a_1, \dots, a_{\ell_a}$ and $b_1, \dots, b_{\ell_b}$.
- There are at most n^s choices for choosing the positions $p_1, \dots, p_{\ell_a}$ and $p'_1, \dots, p'_{\ell_a}$.

Cuckoo Hashing

Cuckoo Hashing

Hence, there are at most $s^3(mn)^2$ cycle-structures of size s.

Cuckoo Hashing

Hence, there are at most $s^3(mn)^2$ cycle-structures of size s.

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

$$s^3(mn)^s \cdot \left(\frac{\mu}{n}\right)^{2(s+1)}$$

Cuckoo Hashing

Hence, there are at most $s^3(mn)^2$ cycle-structures of size s.

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

$$s^3(mn)^s \cdot \left(\frac{\mu}{n}\right)^{2(s+1)} = \frac{s^3}{mn}\Big(mn\Big)^{s+1}\left(\frac{\mu^2}{n^2}\right)^{s+1}$$

Cuckoo Hashing

Hence, there are at most $s^3(mn)^2$ cycle-structures of size s.

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

$$s^3(mn)^s \cdot \Big(\frac{\mu}{n}\Big)^{2(s+1)} = \frac{s^3}{mn}\Big(mn\Big)^{s+1}\Big(\frac{\mu^2}{n^2}\Big)^{s+1}$$

$$= \frac{s^3}{mn}\Big(\frac{\mu^2 m}{n}\Big)^{s+1}$$

Cuckoo Hashing

Cuckoo Hashing

If we make sure that $n \geq (1+\delta)\mu^2 m$ for a constant δ (i.e., the hash-table is not too full) we obtain

$$\Pr[\text{there exists an active cycle-structure}]$$

Cuckoo Hashing

If we make sure that $n \geq (1+\delta)\mu^2 m$ for a constant δ (i.e., the hash-table is not too full) we obtain

$$\Pr[\text{there exists an active cycle-structure}]$$
$$\leq \sum_{s=2}^{\infty} \Pr[\text{there exists an act. cycle-structure of size } s]$$

Cuckoo Hashing

If we make sure that $n \geq (1+\delta)\mu^2 m$ for a constant δ (i.e., the hash-table is not too full) we obtain

$$\begin{aligned}
&\Pr[\text{there exists an active cycle-structure}]\\
&\qquad\leq \sum_{s=2}^{\infty} \Pr[\text{there exists an act. cycle-structure of size } s]\\
&\qquad\leq \sum_{s=2}^{\infty} \frac{s^3}{mn}\Big(\frac{\mu^2 m}{n}\Big)^{s+1}
\end{aligned}$$

Cuckoo Hashing

If we make sure that $n \geq (1+\delta)\mu^2 m$ for a constant δ (i.e., the hash-table is not too full) we obtain

$$\begin{aligned}
&\Pr[\text{there exists an active cycle-structure}] \\
&\qquad \leq \sum_{s=2}^{\infty} \Pr[\text{there exists an act. cycle-structure of size } s] \\
&\qquad \leq \sum_{s=2}^{\infty} \frac{s^3}{mn}\Big(\frac{\mu^2 m}{n}\Big)^{s+1} \\
&\qquad \leq \frac{1}{mn}\sum_{s=0}^{\infty} s^3\Big(\frac{1}{1+\delta}\Big)^{s}
\end{aligned}$$

Cuckoo Hashing

If we make sure that $n \geq (1+\delta)\mu^2 m$ for a constant δ (i.e., the hash-table is not too full) we obtain

$$
\begin{aligned}
&\Pr[\text{there exists an active cycle-structure}]\\
&\quad\leq \sum_{s=2}^{\infty} \Pr[\text{there exists an act. cycle-structure of size } s]\\
&\quad\leq \sum_{s=2}^{\infty} \frac{s^3}{mn}\Big(\frac{\mu^2 m}{n}\Big)^{s+1}\\
&\quad\leq \frac{1}{mn}\sum_{s=0}^{\infty} s^3\Big(\frac{1}{1+\delta}\Big)^{s}\\
&\quad\leq \frac{1}{m^2}\cdot \mathcal{O}(1)\ .
\end{aligned}
$$

Now assume that the insert operation takes t steps and does not create an infinite loop.

Consider the sequences $x, a_1, a_2, \dots, a_{\ell_a}$ and $x, b_1, b_2, \dots, b_{\ell_b}$ where the a_i's and b_i's are defined as before (but for the construction we only use keys examined during the while loop)

If the insert operation takes t steps then

$$t \leq 2\ell_a + 2\ell_b + 2$$

as no key is examined more than twice.

Hence, one of the sequences $x, a_1, a_2, \dots, a_{\ell_a}$ and $x, b_1, b_2, \dots, b_{\ell_b}$ must contain at least $t/4$ keys (either $\ell_a + 1$ or $\ell_b + 1$ must be larger than $t/4$).

Now assume that the insert operation takes t steps and does not create an infinite loop.

Consider the sequences $x, a_1, a_2, \dots, a_{\ell_a}$ and $x, b_1, b_2, \dots, b_{\ell_b}$ where the a_i's and b_i's are defined as before (but for the construction we only use keys examined during the while loop)

If the insert operation takes t steps then

$$t \le 2\ell_a + 2\ell_b + 2$$

as no key is examined more than twice.

Hence, one of the sequences $x, a_1, a_2, \dots, a_{\ell_a}$ and $x, b_1, b_2, \dots, b_{\ell_b}$ must contain at least $t/4$ keys (either $\ell_a + 1$ or $\ell_b + 1$ must be larger than $t/4$).

Now assume that the insert operation takes t steps and does not create an infinite loop.

Consider the sequences $x, a_1, a_2, \dots, a_{\ell_a}$ and $x, b_1, b_2, \dots, b_{\ell_b}$ where the a_i's and b_i's are defined as before (but for the construction we only use keys examined during the while loop)

If the insert operation takes t steps then

$$t \le 2\ell_a + 2\ell_b + 2$$

as no key is examined more than twice.

Hence, one of the sequences $x, a_1, a_2, \dots, a_{\ell_a}$ and $x, b_1, b_2, \dots, b_{\ell_b}$ must contain at least $t/4$ keys (either $\ell_a + 1$ or $\ell_b + 1$ must be larger than $t/4$).

Now assume that the insert operation takes t steps and does not create an infinite loop.

Consider the sequences $x, a_1, a_2, \dots, a_{\ell_a}$ and $x, b_1, b_2, \dots, b_{\ell_b}$ where the a_i's and b_i's are defined as before (but for the construction we only use keys examined during the while loop)

If the insert operation takes t steps then

$$t \le 2\ell_a + 2\ell_b + 2$$

as no key is examined more than twice.

Hence, one of the sequences $x, a_1, a_2, \dots, a_{\ell_a}$ and $x, b_1, b_2, \dots, b_{\ell_b}$ must contain at least $t/4$ keys (either $\ell_a + 1$ or $\ell_b + 1$ must be larger than $t/4$).

Now assume that the insert operation takes t steps and does not create an infinite loop.

Consider the sequences $x, a_1, a_2, \dots, a_{\ell_a}$ and $x, b_1, b_2, \dots, b_{\ell_b}$ where the a_i's and b_i's are defined as before (but for the construction we only use keys examined during the while loop)

If the insert operation takes t steps then

$$t \le 2\ell_a + 2\ell_b + 2$$

as no key is examined more than twice.

Hence, one of the sequences $x, a_1, a_2, \dots, a_{\ell_a}$ and $x, b_1, b_2, \dots, b_{\ell_b}$ must contain at least $t/4$ keys (either $\ell_a + 1$ or $\ell_b + 1$ must be larger than $t/4$).

Define a sub-sequence of length ℓ starting with x, as a sequence $x_1, \dots, x_\ell$ of keys with $x_1 = x$, together with $\ell + 1$ positions $p_0, p_1, \dots, p_\ell$ from $\{0, \dots, n-1\}$.

We say a sub-sequence is right-active for h_1 and h_2 if $h_1(x) = h_1(x_1) = p_0$, $h_2(x_1) = h_2(x_2) = p_1$, $h_1(x_2) = h_1(x_3) = p_2$, $h_2(x_3) = h_2(x_4) = p_3, \dots$.

We say a sub-sequence is left-active for h_1 and h_2 if $h_2(x_1) = p_0$, $h_1(x_1) = h_1(x_2) = p_1$, $h_2(x_2) = h_2(x_3) = p_2$, $h_1(x_3) = h_1(x_4) = p_3, \dots$.

For an active sequence starting with x the key x is supposed to have a collision with the second element in the sequence. This collision could either be in the table T_1 (left) or in the table T_2 (right). Therefore the above definitions differentiate between left-active and right-active.

Define a sub-sequence of length ℓ starting with x, as a sequence $x_1, \dots, x_\ell$ of keys with $x_1 = x$, together with $\ell + 1$ positions $p_0, p_1, \dots, p_\ell$ from $\{0, \dots, n-1\}$.

We say a sub-sequence is right-active for h_1 and h_2 if $h_1(x) = h_1(x_1) = p_0$, $h_2(x_1) = h_2(x_2) = p_1$, $h_1(x_2) = h_1(x_3) = p_2$, $h_2(x_3) = h_2(x_4) = p_3, \dots$.

We say a sub-sequence is left-active for h_1 and h_2 if $h_2(x_1) = p_0$, $h_1(x_1) = h_1(x_2) = p_1$, $h_2(x_2) = h_2(x_3) = p_2$, $h_1(x_3) = h_1(x_4) = p_3, \dots$.

For an active sequence starting with x the key x is supposed to have a collision with the second element in the sequence. This collision could either be in the table T_1 (left) or in the table T_2 (right). Therefore the above definitions differentiate between left-active and right-active.

Define a sub-sequence of length ℓ starting with x, as a sequence $x_1, \dots, x_\ell$ of keys with $x_1 = x$, together with $\ell + 1$ positions $p_0, p_1, \dots, p_\ell$ from $\{0, \dots, n-1\}$.

We say a sub-sequence is right-active for h_1 and h_2 if $h_1(x) = h_1(x_1) = p_0$, $h_2(x_1) = h_2(x_2) = p_1$, $h_1(x_2) = h_1(x_3) = p_2$, $h_2(x_3) = h_2(x_4) = p_3, \dots$.

We say a sub-sequence is left-active for h_1 and h_2 if $h_2(x_1) = p_0$, $h_1(x_1) = h_1(x_2) = p_1$, $h_2(x_2) = h_2(x_3) = p_2$, $h_1(x_3) = h_1(x_4) = p_3, \dots$.

For an active sequence starting with x the key x is supposed to have a collision with the second element in the sequence. This collision could either be in the table T_1 (left) or in the table T_2 (right). Therefore the above definitions differentiate between left-active and right-active.

Define a sub-sequence of length ℓ starting with x, as a sequence $x_1, \dots, x_\ell$ of keys with $x_1 = x$, together with $\ell + 1$ positions $p_0, p_1, \dots, p_\ell$ from $\{0, \dots, n-1\}$.

We say a sub-sequence is right-active for h_1 and h_2 if
$h_1(x) = h_1(x_1) = p_0$, $h_2(x_1) = h_2(x_2) = p_1$,
$h_1(x_2) = h_1(x_3) = p_2$, $h_2(x_3) = h_2(x_4) = p_3, \dots$.

We say a sub-sequence is left-active for h_1 and h_2 if $h_2(x_1) = p_0$,
$h_1(x_1) = h_1(x_2) = p_1$, $h_2(x_2) = h_2(x_3) = p_2$,
$h_1(x_3) = h_1(x_4) = p_3, \dots$.

For an active sequence starting with x the key x is supposed to have a collision with the second element in the sequence. This collision could either be in the table T_1 (left) or in the table T_2 (right). Therefore the above definitions differentiate between left-active and right-active.

Define a sub-sequence of length ℓ starting with x, as a sequence $x_1, \dots, x_\ell$ of keys with $x_1 = x$, together with $\ell + 1$ positions $p_0, p_1, \dots, p_\ell$ from $\{0, \dots, n-1\}$.

We say a sub-sequence is right-active for h_1 and h_2 if
$h_1(x) = h_1(x_1) = p_0$, $h_2(x_1) = h_2(x_2) = p_1$,
$h_1(x_2) = h_1(x_3) = p_2$, $h_2(x_3) = h_2(x_4) = p_3, \dots$.

We say a sub-sequence is left-active for h_1 and h_2 if $h_2(x_1) = p_0$,
$h_1(x_1) = h_1(x_2) = p_1$, $h_2(x_2) = h_2(x_3) = p_2$,
$h_1(x_3) = h_1(x_4) = p_3, \dots$.

For an active sequence starting with x the key x is supposed to have a collision with the second element in the sequence. This collision could either be in the table T_1 (left) or in the table T_2 (right). Therefore the above definitions differentiate between left-active and right-active.

Cuckoo Hashing

Observation:
If the insert takes $t \geq 4\ell$ steps there must either be a left-active or a right-active sub-sequence of length ℓ starting with x.

Cuckoo Hashing

The probability that a given sub-sequence is left-active (right-active) is at most

$$\left(\frac{\mu}{n}\right)^{2\ell} ,$$

if we use (μ, ℓ)-independent hash-functions. This holds since there are ℓ keys whose hash-values (two values per key) have to map to pre-specified positions.

Cuckoo Hashing

The number of sequences is at most $m^{\ell-1}p^{\ell+1}$ as we can choose $\ell - 1$ keys (apart from x) and we can choose $\ell + 1$ positions $p_0, \dots, p_\ell$.

The probability that there exists a left-active **or** right-active sequence of length ℓ is at most

$$\begin{aligned} &\Pr[\text{there exists active sequ. of length } \ell] \\ &\qquad \le 2 \cdot m^{\ell-1} \cdot n^{\ell+1} \cdot \Big(\frac{\mu}{n}\Big)^{2\ell} \\ &\qquad \le 2\Big(\frac{1}{1+\delta}\Big)^{\ell} \end{aligned}$$

Cuckoo Hashing

If the search does not run into an infinite loop the probability that it takes more than 4ℓ steps is at most

$$2\Big(\frac{1}{1+\delta}\Big)^{\ell}$$

We choose $\mathrm{maxsteps} = 4(1 + 2\log m)/\log(1+\delta)$. Then the probability of terminating the while-loop because of reaching $\mathrm{maxsteps}$ is only $\mathcal{O}(\frac{1}{m^2})$ ($\mathcal{O}(1/m^2)$ because of reaching an infinite loop and $1/m^2$ because the search takes $\mathrm{maxsteps}$ steps without running into a loop).

Cuckoo Hashing

The expected time for an insert under the condition that $\mathrm{maxsteps}$ is not reached is

$$\sum_{\ell\geq 0} \Pr[\text{search takes at least } \ell \text{ steps} \mid \text{iteration successful}]$$

$$\leq \sum_{\ell\geq 0} 8\Big(\frac{1}{1+\delta}\Big)^{\ell} = \mathcal{O}(1) \enspace .$$

More generally, the above expression gives a bound on the cost in the successful iteration of an insert-operation (there is exactly one successful iteration).

An iteration that is not successful induces cost $\mathcal{O}(m)$ for doing a complete rehash.

Cuckoo Hashing

The expected number of unsuccessful operations is $\mathcal{O}(\frac{1}{m^2})$.
Hence, the expected cost in unsuccessful iterations is only $\mathcal{O}(\frac{1}{m})$.

Hence, the total expected cost for an insert-operation is constant.

Cuckoo Hashing

What kind of hash-functions do we need?

Since $\mathrm{maxsteps}$ is $\Theta(\log m)$ it is sufficient to have $(\mu, \Theta(\log m))$-independent hash-functions.

Cuckoo Hashing

How do we make sure that $n \geq \mu^2(1+\delta)m$?

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

Definition 31

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

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

Let $\mathcal{H}_n^d := \{h_{\vec{a}} \mid \vec{a} \in \{0, \dots, q\}^{d+1}\}$. The class $\mathcal{H}_n^d$ is $(2, d+1)$-independent.

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

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

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

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

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

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

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

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

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

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

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

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

$$f_{\bar{a}}(x_i) \in \{t_i + \alpha \cdot n \mid \alpha \in \{0, \ldots, \lceil \tfrac{q}{n} \rceil - 1\}\}$$

Therefore I have

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

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

Fix $\ell \leq d+1$; let $x_1, \dots, x_\ell \in \{0, \dots, q-1\}$ be keys, and let $t_1, \dots, t_\ell$ 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

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

$$f_{\bar{a}}(x_i) \in \{t_i + \alpha \cdot n \mid \alpha \in \{0, \dots, \lceil \tfrac{q}{n} \rceil - 1\}\}$$

Therefore I have

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

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

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

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

Then

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

$$f_{\bar{a}}(x_i) \in \{t_i + \alpha \cdot n \mid \alpha \in \{0, \ldots, \lceil \tfrac{q}{n} \rceil - 1\}\}$$

Therefore I have

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

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

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

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

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

$$f_{\bar{a}}(x_i) \in \{t_i + \alpha \cdot n \mid \alpha \in \{0, \dots, \lceil \tfrac{q}{n} \rceil - 1\}\}$$

Therefore I have

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

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

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

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

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

$$f_{\bar{a}}(x_i) \in \{t_i + \alpha \cdot n \mid \alpha \in \{0, \ldots, \lceil \tfrac{q}{n} \rceil - 1\}\}$$

Therefore I have

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

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

$$\frac{\lceil \frac{q}{n} \rceil^\ell \cdot q^{d-\ell+1}}{q^{d+1}} \le \left(\frac{2}{n}\right)^\ell$$