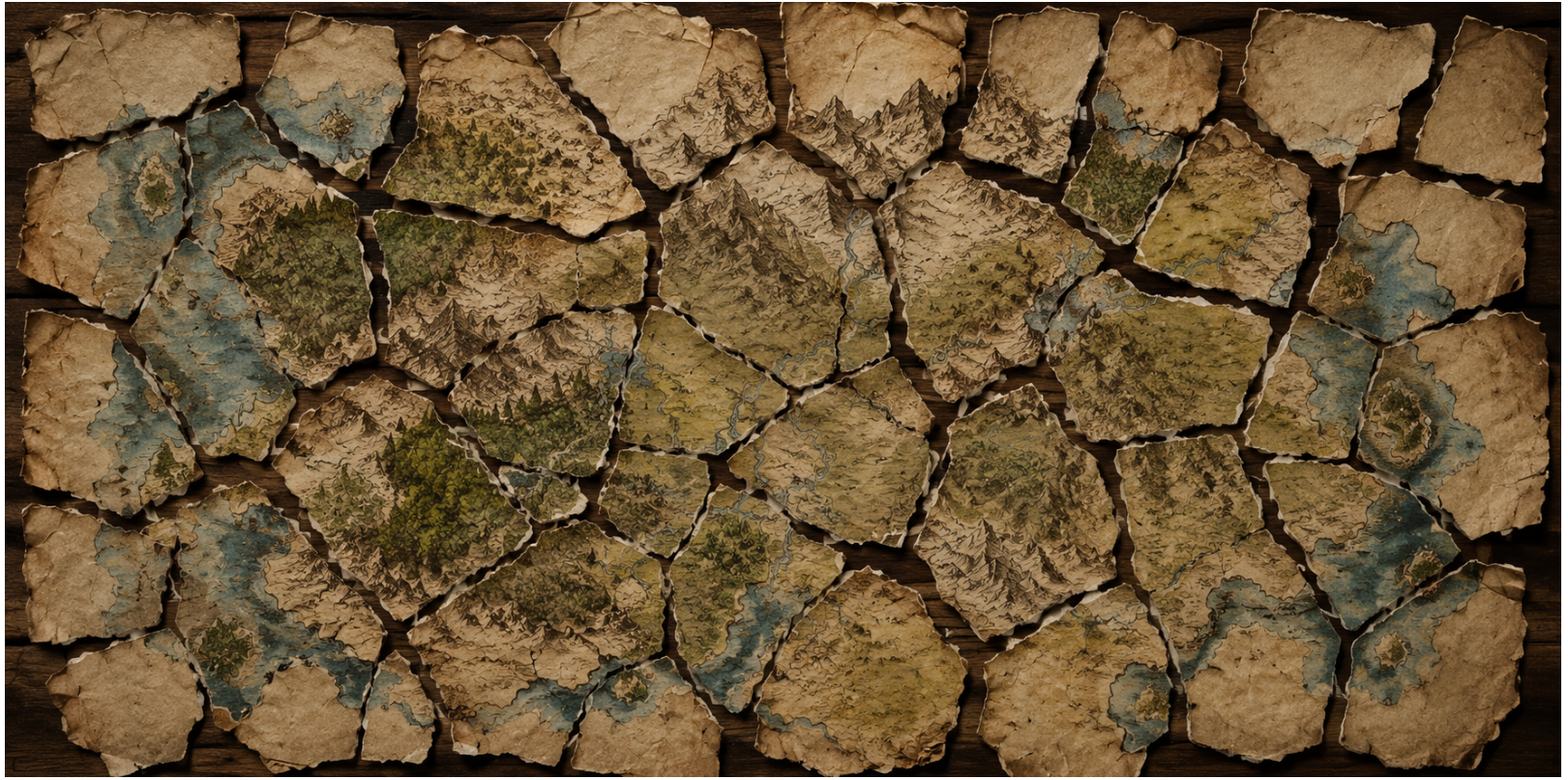
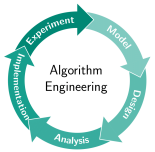


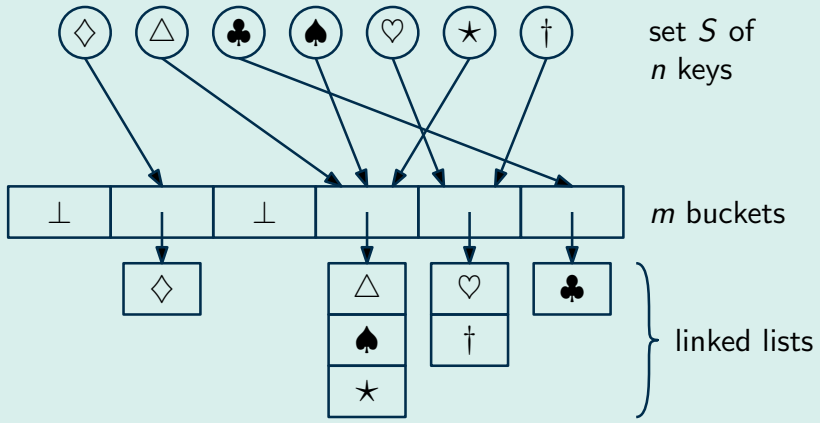
Hash Tables I

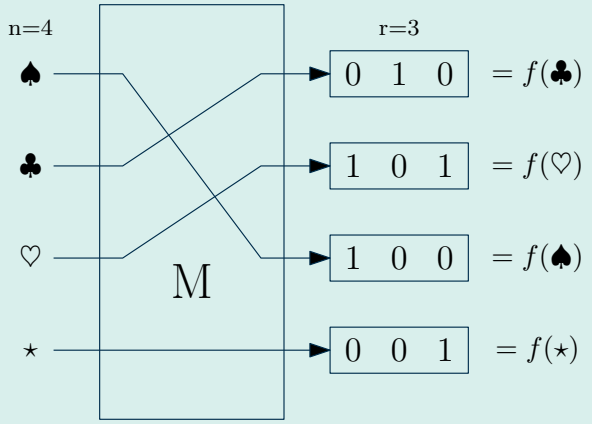
Stefan Walzer, Ragnar Groot Koerkamp, Stefan Hermann | compiled on 6. Juli 2026

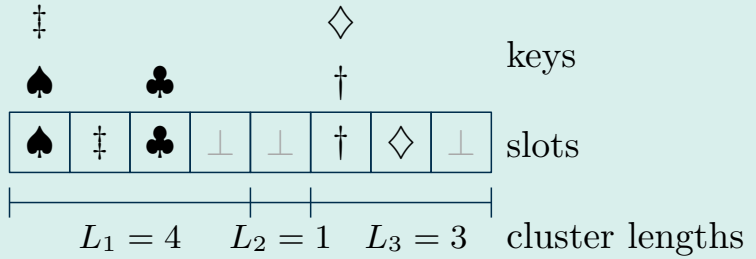


Plan For The Next Three Lectures

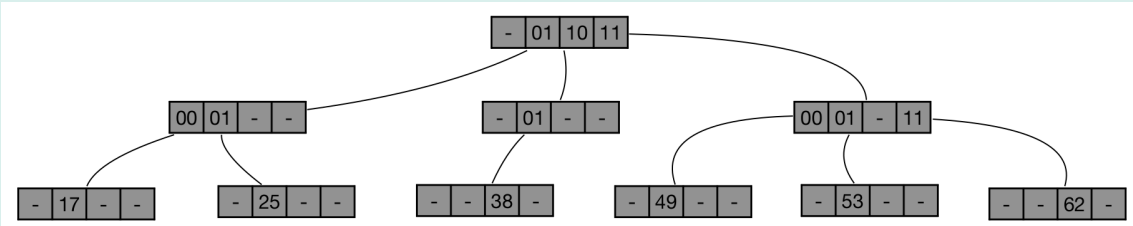
Today: high-level



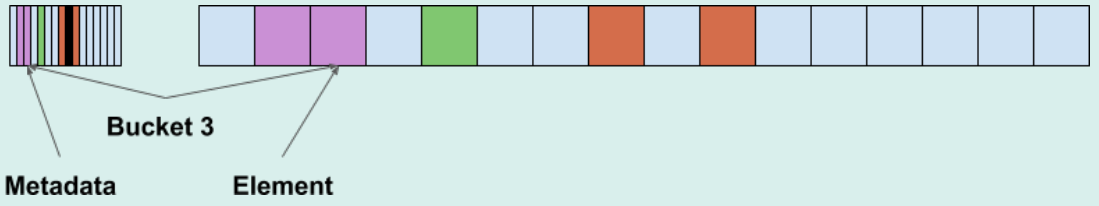




Next Lecture: Hash table theory



Finally: Highly engineered hash tables



Plan For The Next Three Lectures

Today: high-level

- Overview of techniques for hash tables
- Doing theoretical analysis of hash tables . . .
- . . . and see how close the theory is to actual experiments
- Compare Hash Tables to similar data structures (that are not well known)
- Some overlap with the probability and computing lecture

Next lecture: Deep dive into the theory world

- Hash Tables with extraordinary guarantees
- Fun with surprising data structure
- But quite unpractical

Last but not least: Deep dive into highly engineered hash tables

- Googles SwissTable
- Vectorization

Hash Functions

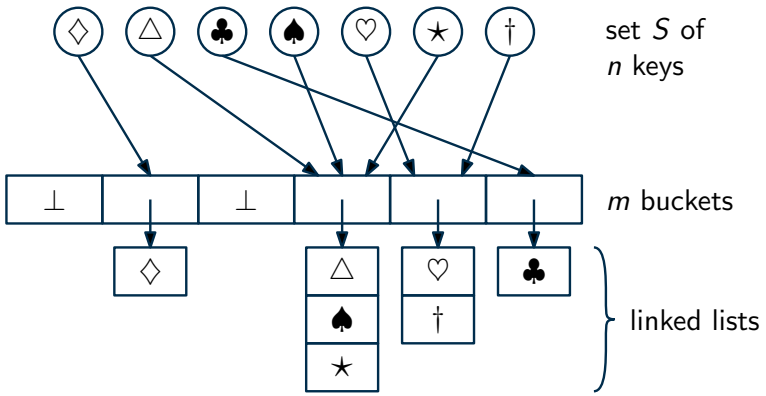
Terminology

D: *Universe* of *keys*

$S \subseteq D$: set of n keys (possibly with associated data)

$h : D \to [m]$: hash function

$\alpha = \frac{n}{m}$: *load factor*, $\alpha = \mathcal{O}(1)$



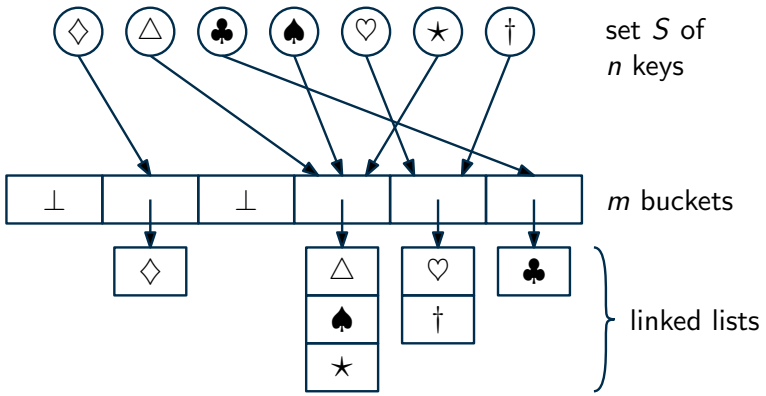
Hash Table with Chaining

Operations in time t with $\mathbb{E}[t] = \mathcal{O}(1)$.

Hash Functions

Terminology

D: *Universe* of *keys*
$S \subseteq D$: set of n keys (possibly with associated data)
$h : D \to [m]$: hash function
$\alpha = \frac{n}{m}$: *load factor*, $\alpha = \mathcal{O}(1)$



Hash Table with Chaining

Operations in time t with $\mathbb{E}[t] = \mathcal{O}(1)$.
Randomness comes from the hash function.

Ideal Hash Functions

Ideally, every function from D to $[n]$ is equally likely to be h:
$h \sim \mathcal{U}([m]^D)$.

Hash Functions

In Theory

In this lecture:
We use the *Simple Uniform Hashing Assumption*, i.e.
we assume $h \sim \mathcal{U}([m]^D)$

Hash Functions

In Theory

In this lecture:
We use the *Simple Uniform Hashing Assumption*, i.e. we assume $h \sim \mathcal{U}([m]^D)$

In Practice

1. sample a seed $\sim \mathcal{U}([2^{64}])$

Hash Functions

In Theory

In this lecture:
We use the *Simple Uniform Hashing Assumption*, i.e. we assume $h \sim \mathcal{U}([m]^D)$

In Practice

1. sample a seed $\sim \mathcal{U}([2^{64}])$
2. use e.g. GxHash (the following is very simplified)

```
__m256i gxHash(__m256i seed, __m256i* ptr, __m256i* end) {
        __m256i state = seed;
        while (ptr < end) {
                __m256i next = _mm256_loadu_si256(ptr++);
                state = _mm256_aesenc_epi128(state, next);
        }
        return state;
}
```

The result is a vector of
(hopefully uniformly distributed) integers.

Hash Functions

In Theory

In this lecture:
We use the *Simple Uniform Hashing Assumption*, i.e. we assume $h \sim \mathcal{U}([m]^D)$

In Practice

1. sample a seed $\sim \mathcal{U}([2^{64}])$
2. use e.g. GxHash (the following is very simplified)

```
__m256i gxHash(__m256i seed, __m256i* ptr, __m256i* end) {
        __m256i state = seed;
        while (ptr < end) {
                __m256i next = _mm256_loadu_si256(ptr++);
                state = _mm256_aesenc_epi128(state, next);
        }
        return state;
}
```

   The result is a vector of (hopefully uniformly distributed) integers.
3. map a uniformly distributed `uint64_t hash` to a uniformly distributed value in $[m]$

Hash Functions

In Theory

In this lecture:
We use the *Simple Uniform Hashing Assumption*, i.e. we assume $h \sim \mathcal{U}([m]^D)$

Middle Ground (not here)

E.g. Universal or independent hash functions

- amenable to analysis (but more difficult)
- still uncommon in practice

In Practice

1. sample a seed $\sim \mathcal{U}([2^{64}])$
2. use e.g. GxHash (the following is very simplified)

```
__m256i gxHash(__m256i seed, __m256i* ptr, __m256i* end) {
        __m256i state = seed;
        while (ptr < end) {
                __m256i next = _mm256_loadu_si256(ptr++);
                state = _mm256_aesenc_epi128(state, next);
        }
        return state;
}
```

   The result is a vector of (hopefully uniformly distributed) integers.
3. map a uniformly distributed `uint64_t hash` to a uniformly distributed value in $[m]$

```
// slow mod
return hash % m;

// "fast range"
return (uint64_t)(((__uint128_t)hash * (__uint128_t)m) >> 64);
```

Hash table overview

Metrics

- Exact query time, i.e. counting the number of probes
  - Positive: Query an existing key (we only consider the average over all existing keys)
  - Negative: Query a non-existing key (today's focus)
- Insertion time
- Space: Memory needed by the hash table (b bit elements and p bit pointers)

Technique	Avg. Positive Query	Negative Query	Insertion	Bits per Key
Chaining				

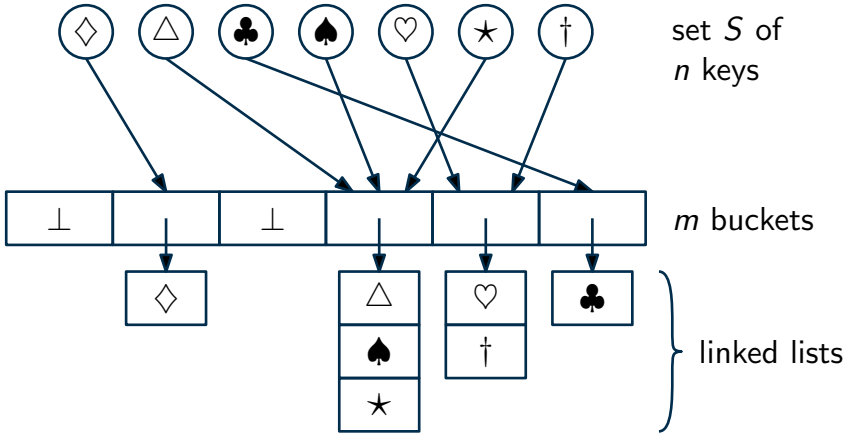
Negative Queries for Hash Tables using Chaining

Goal

- For chaining, we use the number of pointers that we have to look at as a model for the query time.
- What is the query time of a key $x \notin S$?

Negative query time

- A negative query will go trough the entire chain: It first looks at the pointer of the bucket $h(x)$, and then at one additional pointer for each key y mapped to the same bucket $h(y) = h(x)$.
- Using the simple uniform hashing assumption, the n keys are uniformly distributed among the m buckets.
- The expected number of keys in a bucket is therefore $\frac{n}{m} = \alpha$.
- This gives us an expected query time of $1 + \alpha$.



Hash table overview

Metrics

- Exact query time, i.e. counting the number of probes
  - Positive: Query an existing key (we only consider the average over all existing keys)
  - Negative: Query a non-existing key (today's focus)
- Insertion time
- Space: Memory needed by the hash table (b bit elements and p bit pointers)

Technique	Avg. Positive Query	Negative Query	Insertion	Bits per Key
Chaining	$\mathcal{O}(1)$ expected	$1 + \alpha$ expected	$=$ neg. query [1]	$p/\alpha + b + p$

[1] even in the worst-case if we can assume the key does not exist yet then we can insert at the beginning of the list

Hash table overview

Metrics

- Exact query time, i.e. counting the number of probes
  - Positive: Query an existing key (we only consider the average over all existing keys)
  - Negative: Query a non-existing key (today's focus)
- Insertion time
- Space: Memory needed by the hash table (b bit elements and p bit pointers)

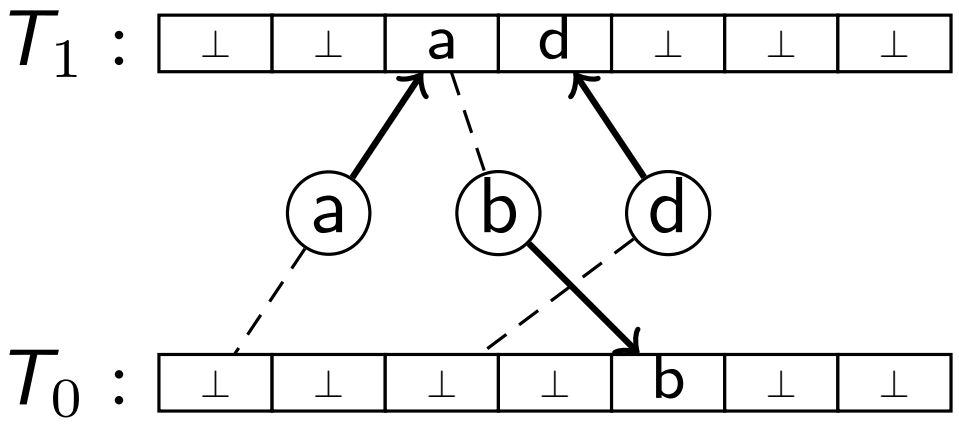
Technique	Avg. Positive Query	Negative Query	Insertion	Bits per Key
Chaining	$\mathcal{O}(1)$ expected	$1+\alpha$ expected	$=$ neg. query [1]	$p/\alpha + b + p$
Cuckoo Hashing	≤ 2	2	$\mathcal{O}(1)$ expected	b/α

[1] even in the worst-case if we can assume the key does not exist yet then we can insert at the beginning of the list

Classic Cuckoo Hashing

Setup

$S \subseteq D$	key set of size n
T_0, T_1	two tables of size $m/2$ each
h_0, h_1	two hash functions
$\alpha = \frac{n}{m}$:	load factor



Algorithm lookup(x)**:**

```
return x ∈ {T_0[h_0(x)], T_1[h_1(x)]}
```

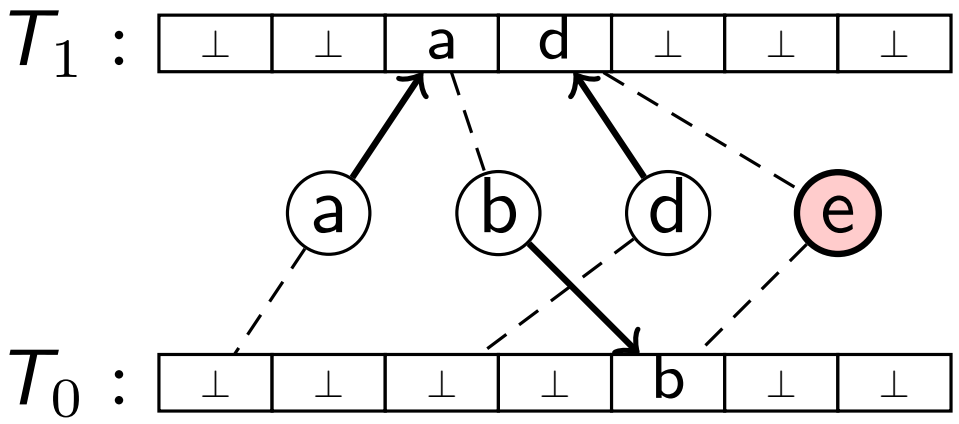
Algorithm insert(x)**:**

```
for i = 0 to LIMIT do
    b ← i mod 2
    swap(x, T_b[h_b(x)])
    if x = ⊥ then
        return SUCCESS
return FAILURE
```

Classic Cuckoo Hashing

Setup

$S \subseteq D$	key set of size n
T_0, T_1	two tables of size $m/2$ each
h_0, h_1	two hash functions
$\alpha = \frac{n}{m}$:	load factor



Algorithm lookup(x)**:**

```
return x ∈ {T_0[h_0(x)], T_1[h_1(x)]}
```

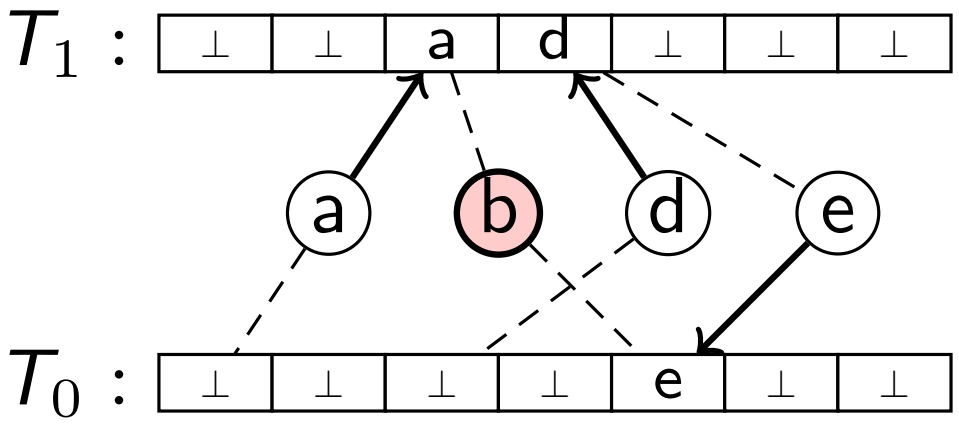
Algorithm insert(x)**:**

```
for i = 0 to LIMIT do
    b ← i mod 2
    swap(x, T_b[h_b(x)])
    if x = ⊥ then
        return SUCCESS
return FAILURE
```

Classic Cuckoo Hashing

Setup

$S \subseteq D$	key set of size n
T_0, T_1	two tables of size $m/2$ each
h_0, h_1	two hash functions
$\alpha = \frac{n}{m}$:	load factor



Algorithm lookup(x)**:**

```
return x ∈ {T_0[h_0(x)], T_1[h_1(x)]}
```

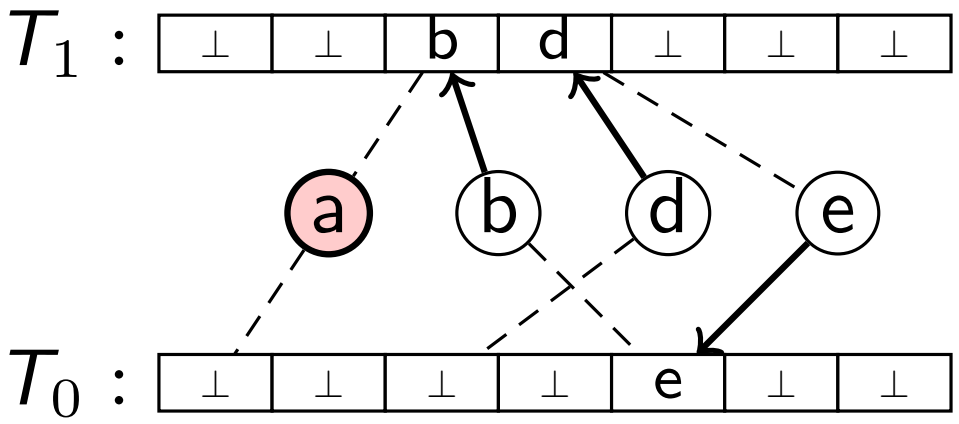
Algorithm insert(x)**:**

```
for i = 0 to LIMIT do
    b ← i mod 2
    swap(x, T_b[h_b(x)])
    if x = ⊥ then
        return SUCCESS
return FAILURE
```

Classic Cuckoo Hashing

Setup

$S \subseteq D$	key set of size n
T_0, T_1	two tables of size $m/2$ each
h_0, h_1	two hash functions
$\alpha = \frac{n}{m}$:	load factor



Algorithm lookup(x)**:**

```
return x ∈ {T_0[h_0(x)], T_1[h_1(x)]}
```

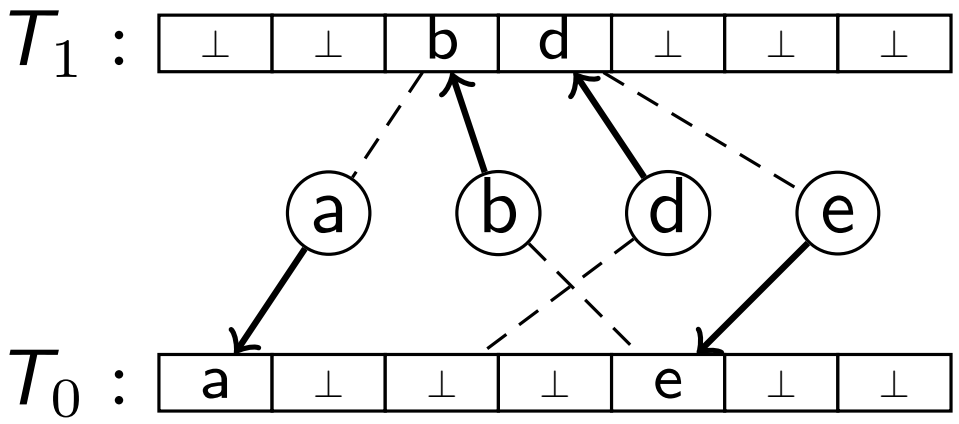
Algorithm insert(x)**:**

```
for i = 0 to LIMIT do
    b ← i mod 2
    swap(x, T_b[h_b(x)])
    if x = ⊥ then
        return SUCCESS
return FAILURE
```

Classic Cuckoo Hashing

Setup

$S \subseteq D$	key set of size n
T_0, T_1	two tables of size $m/2$ each
h_0, h_1	two hash functions
$\alpha = \frac{n}{m}$:	load factor



Algorithm lookup(x)**:**

```
return x ∈ {T_0[h_0(x)], T_1[h_1(x)]}
```

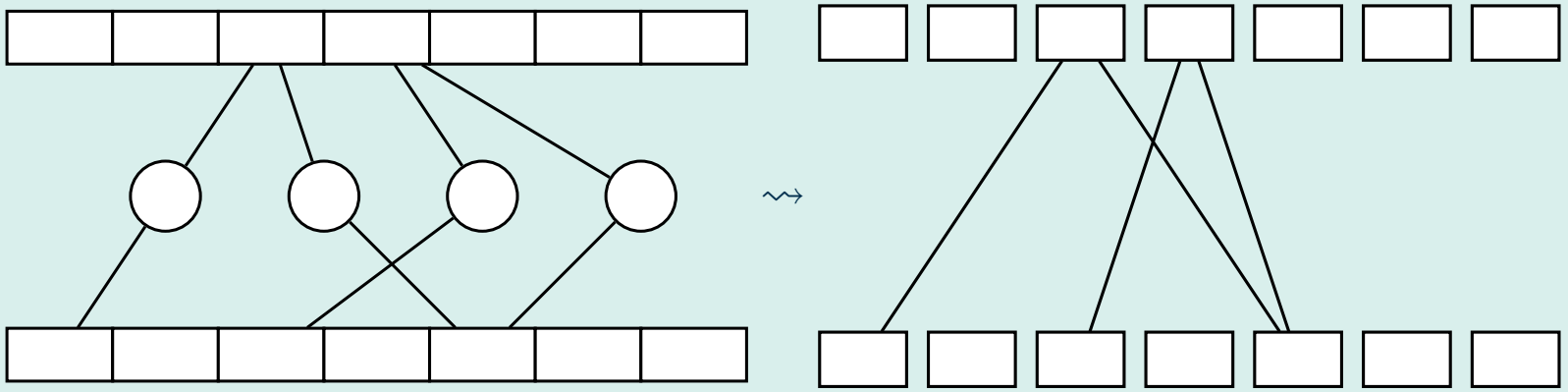
Algorithm insert(x)**:**

```
for i = 0 to LIMIT do
    b ← i mod 2
    swap(x, T_b[h_b(x)])
    if x = ⊥ then
        return SUCCESS
return FAILURE
```

Classic Cuckoo Hashing

The Cuckoo Graph

Consider the *cuckoo graph*: a slot corresponds to a node and a key x corresponds to an edge $(h_0(x), h_1(x))$.

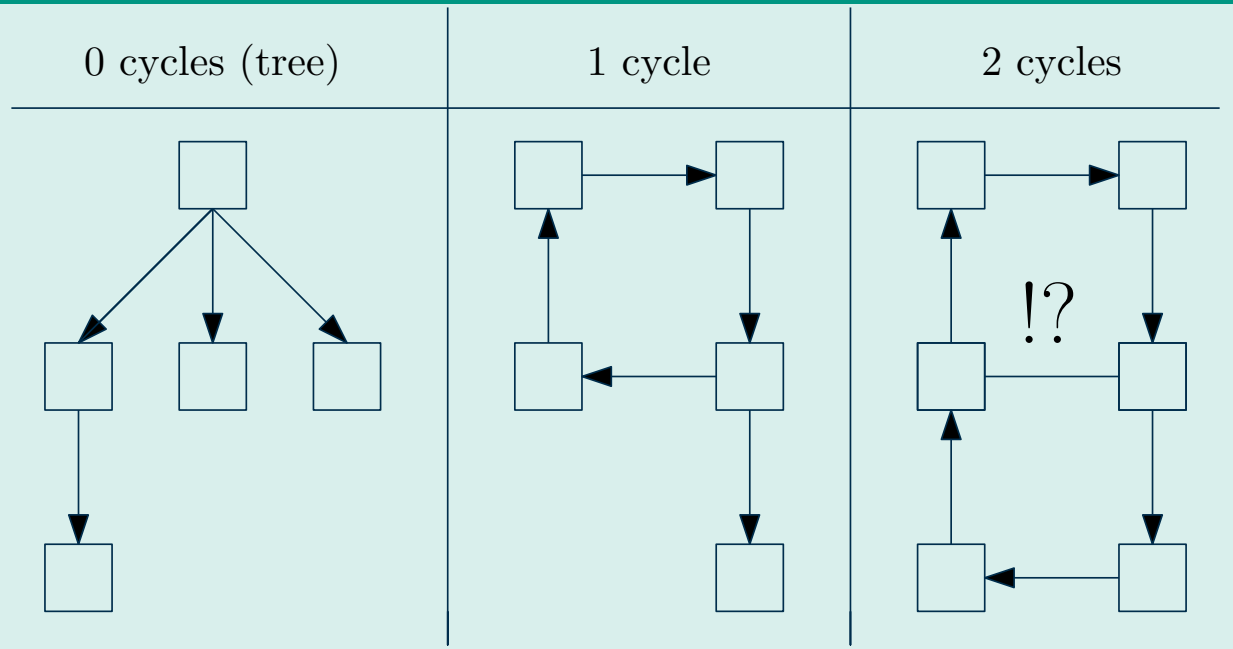


Classic Cuckoo Hashing

Observation

Placing a key in one of its two slots corresponds to giving its edge an orientation. Each key can be placed into one of its slots if and only if the graph is *1-orientable*, i.e. each edge can be oriented s.t. the indegree of each node is 1.

Which components are 1-orientable?



Only components with at most one cycle are 1-orientable.

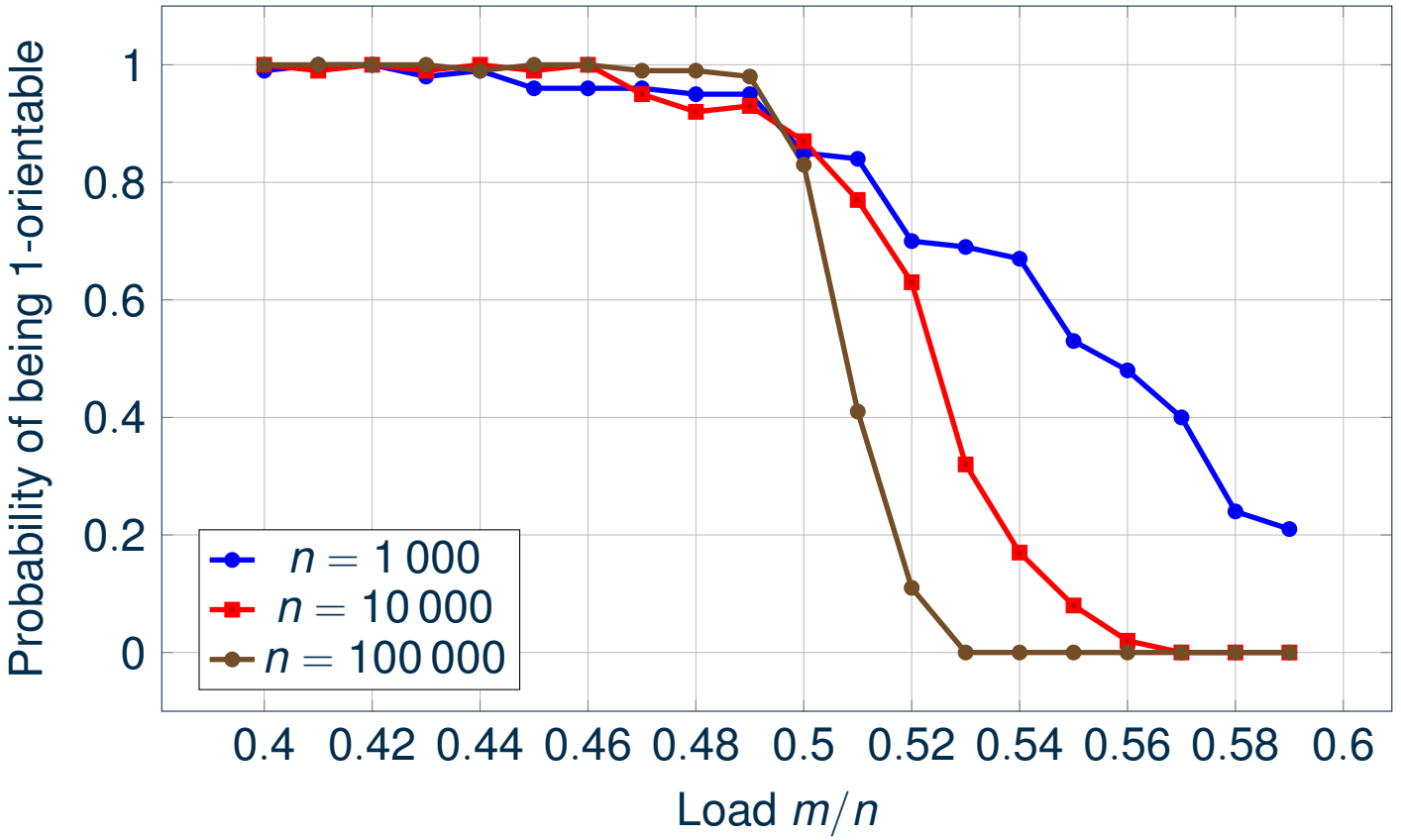
Classic Cuckoo Hashing

Experiment

Its always a good idea to make an experiment to get an idea of what is going on. Here we observe the probability that the cuckoo graph is 1-orientable for different loads and different n.

Observation

For large n, the graph appears to converge to being 1-orientable exactly for loads <0.5.



Classic Cuckoo Hashing // Intuition: Galton–Watson tree

Idealized model

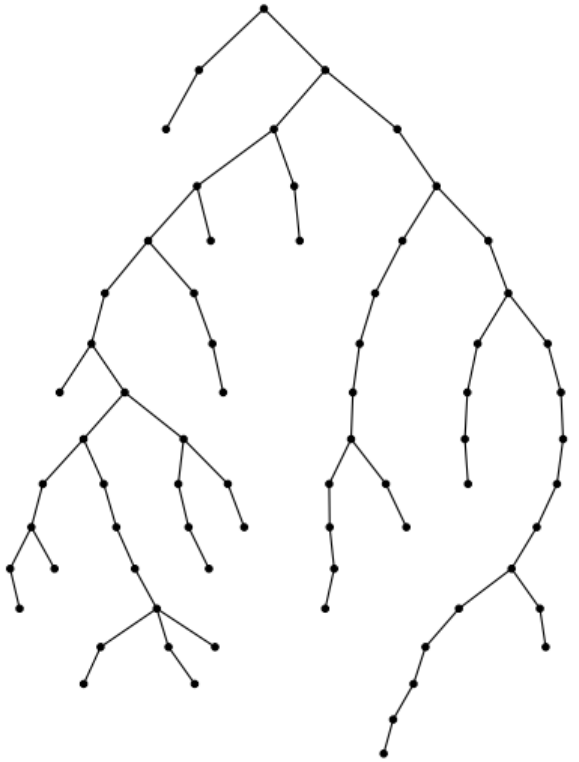
Look at some node. The node is connected to a random number X of other nodes. The random variable X follows a Binomial distribution $\text{Bin}(n, \frac{2}{m})$. The connected nodes are in turn connected with a random number of other node. Those random variables follow the the same distribution but they are (slightly) dependent on each other. However, for simplicity we assume independence here.

Definition

The process describes above results in a Galton-Watson tree: A Galton–Watson tree is a random rooted tree in which each node has a random number of children X drawn independently from the same *offspring distribution* ($\text{Bin}(n, \frac{2}{m})$ in our case).

Intuition about the $\alpha = 0.5$ threshold

The expected number of children of each node is 2α. For $\alpha < 0.5$, the “population goes extinct” (birthrate below 1 per person). For $\alpha > 0.5$, the “population likely explodes”, a component’s size becomes $\mathcal{O}(n)$, almost surely containing >1 cycles.



Hash table overview

Metrics

- Exact query time, i.e. counting the number of probes
  - Positive: Query an existing key (we only consider the average over all existing keys)
  - Negative: Query a non-existing key (today's focus)
- Insertion time
- Space: Memory needed by the hash table (b bit elements and p bit pointers)

Technique	Avg. Positive Query	Negative Query	Insertion	Bits per Key
Chaining	$\mathcal{O}(1)$ expected	$1+\alpha$ expected	$=$ neg. query [1]	$p/\alpha + b + p$
Cuckoo Hashing	≤ 2	2	$\mathcal{O}(1)$ expected	b/α

[1] even in the worst-case if we can assume the key does not exist yet then we can insert at the beginning of the list

Hash table overview

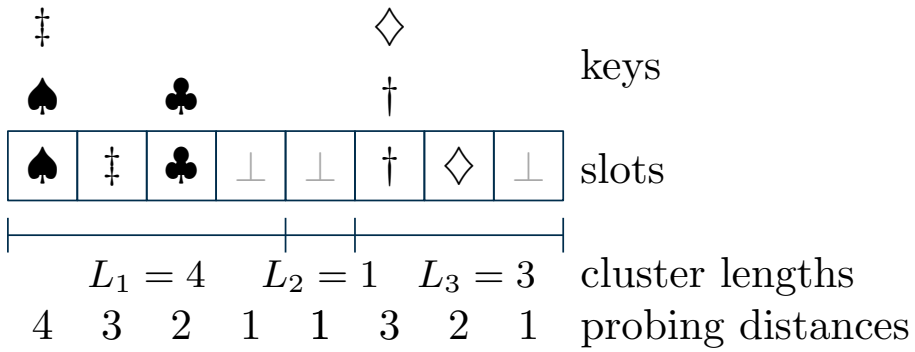
Metrics

- Exact query time, i.e. counting the number of probes
  - Positive: Query an existing key (we only consider the average over all existing keys)
  - Negative: Query a non-existing key (today's focus)
- Insertion time
- Space: Memory needed by the hash table (b bit elements and p bit pointers)

Technique	Avg. Positive Query	Negative Query	Insertion	Bits per Key
Chaining	$\mathcal{O}(1)$ expected	$1+\alpha$ expected	$=$ neg. query [1]	$p/\alpha + b + p$
Cuckoo Hashing	≤ 2	2	$\mathcal{O}(1)$ expected	b/α
Linear Probing	$\frac{1}{2}(1+\frac{1}{1-\alpha})$ expected	$\frac{1}{2}(1+\frac{1}{(1-\alpha)^2})$ expected	$=$ neg. query	b/α

[1] even in the worst-case if we can assume the key does not exist yet then we can insert at the beginning of the list

Negative Queries in Linear Probing Hash Table



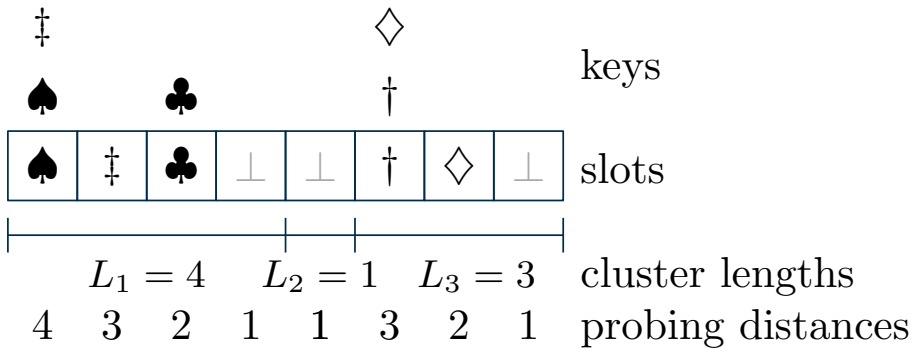
Definition

The *probing distance* of a query is the number of slots a query has to look at.

Definition

A *cluster* is an interval beginning with the first occupied slot after a free slot and ending with the next free slot.

Negative Queries in Linear Probing Hash Table



Definition

The *probing distance* of a query is the number of slots a query has to look at.

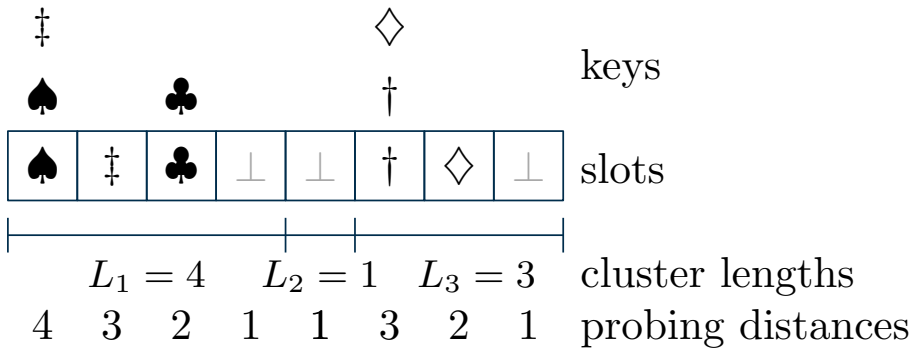
Definition

A *cluster* is an interval beginning with the first occupied slot after a free slot and ending with the next free slot.

Observation

There are $m - n$ free slots and therefore $m - n$ clusters.

Negative Queries in Linear Probing Hash Table



Definition

The *probing distance* of a query is the number of slots a query has to look at.

Definition

A *cluster* is an interval beginning with the first occupied slot after a free slot and ending with the next free slot.

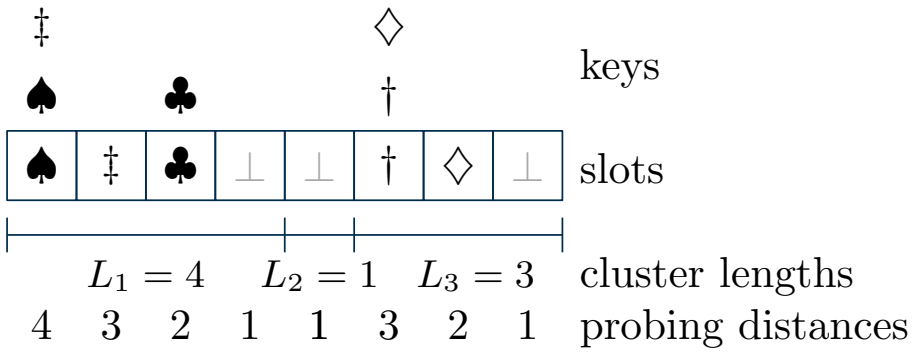
Observation

There are $m - n$ free slots and therefore $m - n$ clusters.

Goal

Let $L_1, \ldots L_{m-n}$ be the random variables describing the length of the clusters (e.g. L_1 for the cluster that ends on the first free slot of the table). Each slot is queried with equal probability. Hence, the expected probing distance for a negative query T^- is the average probing distance of a slot.

Negative Queries in Linear Probing Hash Table



Identities

1. $\mathbb{E}(A + B) = \mathbb{E}(A) + \mathbb{E}(B)$
2. $\sum_{i=1}^{n} i = \frac{n+n^2}{2}$

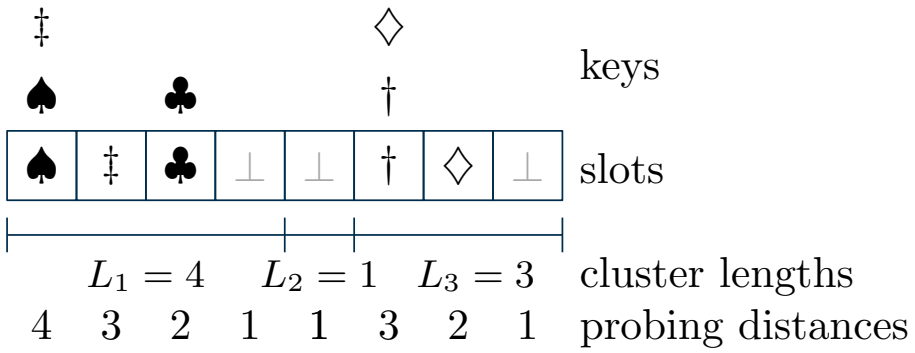
Definition

For brevity we define $L := L_1$

Average slot probing distance

$$\mathbb{E}[T^-] = \mathbb{E}\left[\frac{1}{m}\sum_{i=1}^{m-n}\sum_{d=1}^{L_i} d\right] \quad \text{// use 1. identity}$$

Negative Queries in Linear Probing Hash Table



Identities

1. $\mathbb{E}(A + B) = \mathbb{E}(A) + \mathbb{E}(B)$
2. $\sum_{i=1}^{n} i = \frac{n+n^2}{2}$

Definition

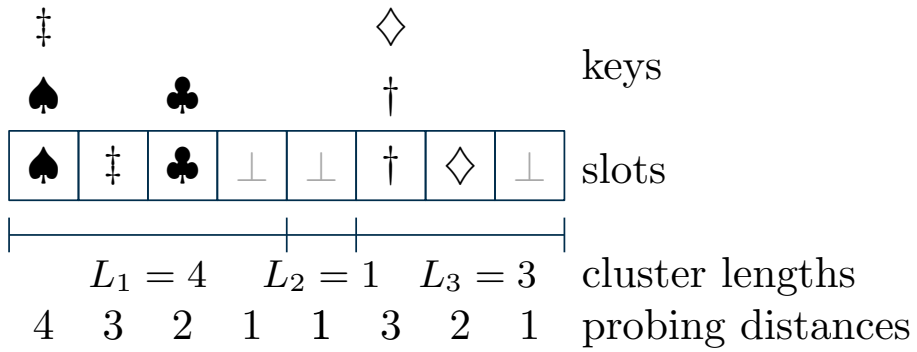
For brevity we define $L := L_1$

Average slot probing distance

$$\mathbb{E}[T^-] = \mathbb{E}\left[\frac{1}{m}\sum_{i=1}^{m-n}\sum_{d=1}^{L_i} d\right] \quad \text{// use 1. identity}$$

$$= \frac{1}{m}\sum_{i=1}^{m-n}\mathbb{E}\left[\sum_{d=1}^{L_i} d\right] \quad \text{// all } L_i \text{ are equally distributed}$$

Negative Queries in Linear Probing Hash Table



Identities

1. $\mathbb{E}(A + B) = \mathbb{E}(A) + \mathbb{E}(B)$
2. $\sum_{i=1}^{n} i = \frac{n+n^2}{2}$

Definition

For brevity we define $L := L_1$

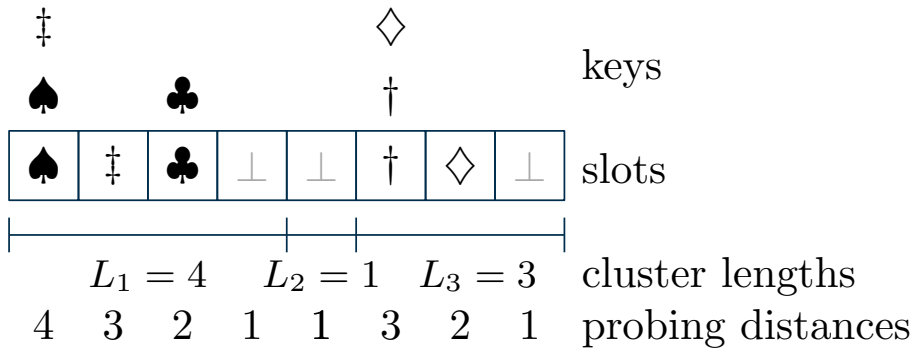
Average slot probing distance

$$\mathbb{E}[T^-] = \mathbb{E}\left[\frac{1}{m}\sum_{i=1}^{m-n}\sum_{d=1}^{L_i} d\right] \quad \text{// use 1. identity}$$

$$= \frac{1}{m}\sum_{i=1}^{m-n}\mathbb{E}\left[\sum_{d=1}^{L_i} d\right] \quad \text{// all } L_i \text{ are equally distributed}$$

$$= \frac{1}{m}(m-n)\cdot\mathbb{E}\left[\sum_{d=1}^{L} d\right] \quad \text{// use 2. then 1. identity}$$

Negative Queries in Linear Probing Hash Table



Identities

1. $\mathbb{E}(A + B) = \mathbb{E}(A) + \mathbb{E}(B)$
2. $\sum_{i=1}^{n} i = \frac{n+n^2}{2}$

Definition

For brevity we define $L := L_1$

Average slot probing distance

$$\mathbb{E}[T^-] = \mathbb{E}\left[\frac{1}{m}\sum_{i=1}^{m-n}\sum_{d=1}^{L_i} d\right] \quad \text{// use 1. identity}$$

$$= \frac{1}{m}\sum_{i=1}^{m-n}\mathbb{E}\left[\sum_{d=1}^{L_i} d\right] \quad \text{// all } L_i \text{ are equally distributed}$$

$$= \frac{1}{m}(m-n)\cdot\mathbb{E}\left[\sum_{d=1}^{L} d\right] \quad \text{// use 2. then 1. identity}$$

$$= (1-\alpha)\cdot\mathbb{E}\left[\frac{L+L^2}{2}\right] = \frac{1-\alpha}{2}(\mathbb{E}[L] + \mathbb{E}[L^2])$$

Negative Queries in Linear Probing Hash Table

$\mathbb{E}[L]$

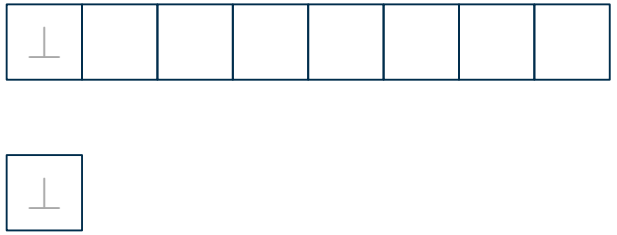
The expected cluster length $\mathbb{E}[L]$ is just the number of slots over the number of clusters:

$$\begin{aligned}\mathbb{E}[L] &= \frac{m}{m-n} && \text{// divide everything by } m \\ &= \frac{1}{1-\alpha}\end{aligned}$$

Negative Queries in Linear Probing Hash Table

Approach to understand L and to obtain $\mathbb{E}[L^2]$

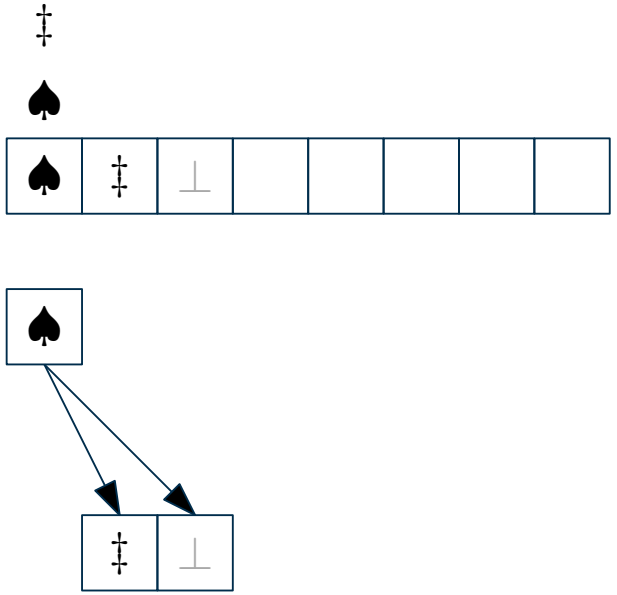
Let X be the random variable describing the number of keys that hash to the first slot of the first cluster. X follows a the distribution $Bin(n, \frac{1}{m})$. Each of the X keys has to be accommodated in an additional slot. Hence, the cluster grows by X slots. Now there may be other keys hashing to those new slots, which in turn have to be accommodated in this cluster, resulting in a recursive structure.



Negative Queries in Linear Probing Hash Table

Approach to understand L and to obtain $\mathbb{E}[L^2]$

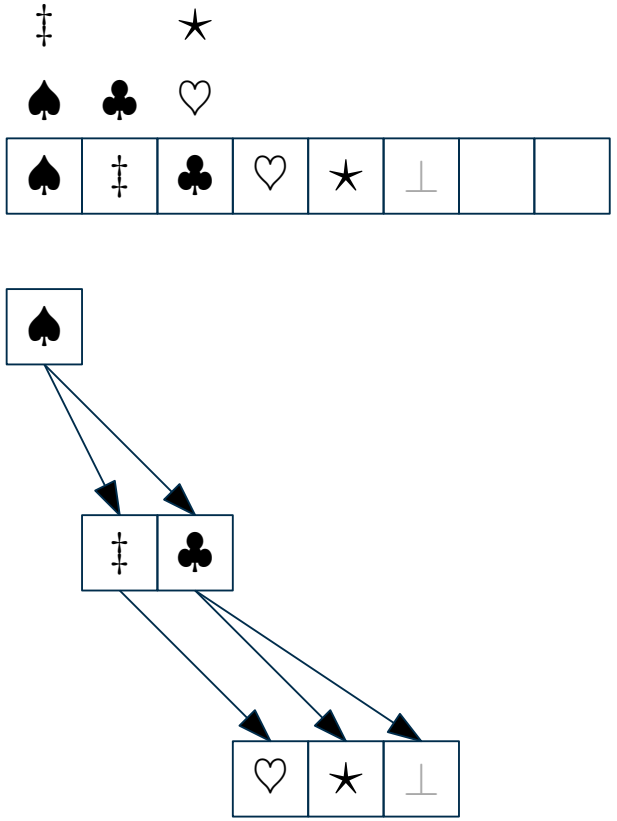
Let X be the random variable describing the number of keys that hash to the first slot of the first cluster. X follows a the distribution $Bin(n, \frac{1}{m})$. Each of the X keys has to be accommodated in an additional slot. Hence, the cluster grows by X slots. Now there may be other keys hashing to those new slots, which in turn have to be accommodated in this cluster, resulting in a recursive structure.



Negative Queries in Linear Probing Hash Table

Approach to understand L and to obtain $\mathbb{E}[L^2]$

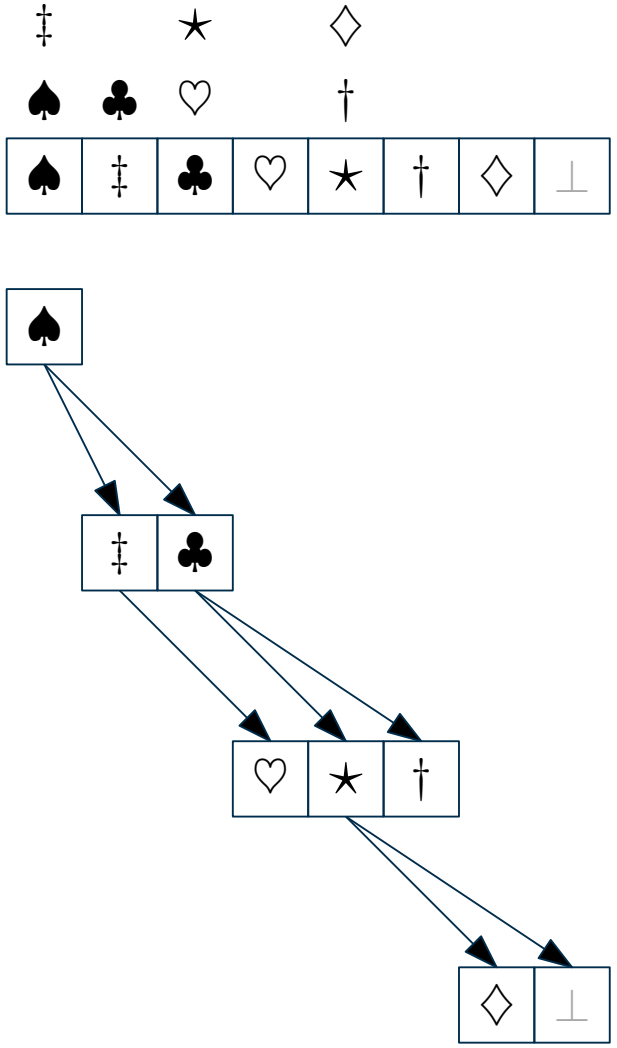
Let X be the random variable describing the number of keys that hash to the first slot of the first cluster. X follows a the distribution $Bin(n, \frac{1}{m})$. Each of the X keys has to be accommodated in an additional slot. Hence, the cluster grows by X slots. Now there may be other keys hashing to those new slots, which in turn have to be accommodated in this cluster, resulting in a recursive structure.



Negative Queries in Linear Probing Hash Table

Approach to understand L and to obtain $\mathbb{E}[L^2]$

Let X be the random variable describing the number of keys that hash to the first slot of the first cluster. X follows a the distribution $Bin(n, \frac{1}{m})$. Each of the X keys has to be accommodated in an additional slot. Hence, the cluster grows by X slots. Now there may be other keys hashing to those new slots, which in turn have to be accommodated in this cluster, resulting in a recursive structure.



Negative Queries in Linear Probing Hash Table

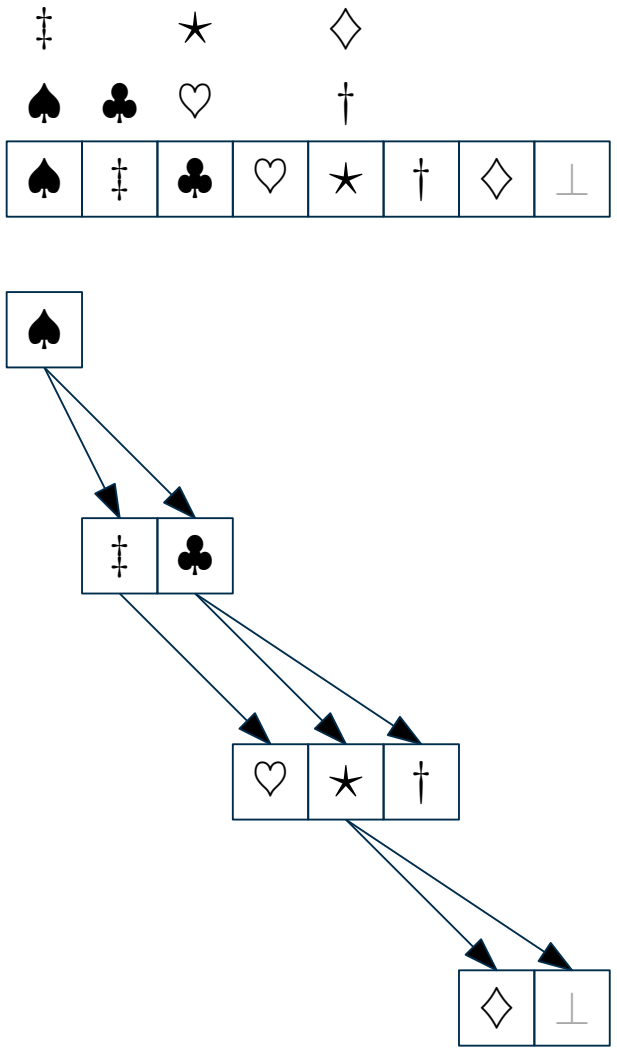
Approach to understand L and to obtain $\mathbb{E}[L^2]$

Let X be the random variable describing the number of keys that hash to the first slot of the first cluster. X follows a the distribution $Bin(n, \frac{1}{m})$. Each of the X keys has to be accommodated in an additional slot. Hence, the cluster grows by X slots. Now there may be other keys hashing to those new slots, which in turn have to be accommodated in this cluster, resulting in a recursive structure.

The surprise appearance

This is again a Galton-Watson tree with offspring distribution X. Let L'_i describe the number of nodes in the sub-tree below the i-th child of the root. The cluster length L corresponds to the number of nodes of the tree, which is 1 + the nodes in the sub-trees:

$$L = 1 + \sum_{i=1}^{X} L'_i \quad \text{// where the } L'_i \text{ follow the same distribution as } L$$



Negative Queries in Linear Probing Hash Table

Previously

$$L = 1 + \sum_{i=1}^{X} L'_i$$

where the L'_i follow the same distribution as L

Useful identities for $\mathbb{E}[L^2]$

$$\mathbb{E}[L^2] = \mathrm{Var}(L) + \mathbb{E}[L]^2$$
$$\mathrm{Var}(L) = \underline{\mathbb{E}[\mathrm{Var}(L \mid X)]} + \mathrm{Var}(\mathbb{E}[L \mid X])$$

$$\mathrm{Var}(A + B) = \mathrm{Var}(A) + \mathrm{Var}(A) \quad \text{// independent } A, B$$

Negative Queries in Linear Probing Hash Table

Previously

$$L = 1 + \sum_{i=1}^{X} L'_i$$

where the L'_i follow the same distribution as L

Useful identities for $\mathbb{E}[L^2]$

$$\mathbb{E}[L^2] = \mathrm{Var}(L) + \mathbb{E}[L]^2$$
$$\mathrm{Var}(L) = \underline{\mathbb{E}[\mathrm{Var}(L \mid X)]} + \mathrm{Var}(\mathbb{E}[L \mid X])$$

$$\mathrm{Var}(A + B) = \mathrm{Var}(A) + \mathrm{Var}(A) \quad \text{// independent } A, B$$

$\mathbb{E}[\mathrm{Var}(L \mid X)]$

$$\mathrm{Var}(L \mid X) = \mathrm{Var}\left(\sum_{i=0}^{X} L'_i \,\middle|\, X\right) \quad \text{// independence of } L'_i$$

Negative Queries in Linear Probing Hash Table

Previously

$$L = 1 + \sum_{i=1}^{X} L'_i$$

where the L'_i follow the same distribution as L

Useful identities for $\mathbb{E}[L^2]$

$$\mathbb{E}[L^2] = \mathrm{Var}(L) + \mathbb{E}[L]^2$$
$$\mathrm{Var}(L) = \underline{\mathbb{E}[\mathrm{Var}(L \mid X)]} + \mathrm{Var}(\mathbb{E}[L \mid X])$$

$$\mathrm{Var}(A + B) = \mathrm{Var}(A) + \mathrm{Var}(A) \quad \text{// independent } A, B$$

$\mathbb{E}[\mathrm{Var}(L \mid X)]$

$$\mathrm{Var}(L \mid X) = \mathrm{Var}\left(\sum_{i=0}^{X} L'_i \,\middle|\, X\right) \quad \text{// independence of } L'_i$$
$$= \sum_{i=1}^{X} \mathrm{Var}\left(L'_i \mid X\right) \quad \text{// independence of } L'_i \text{ and } X$$

Negative Queries in Linear Probing Hash Table

Previously

$$L = 1 + \sum_{i=1}^{X} L'_i$$

where the L'_i follow the same distribution as L

Useful identities for $\mathbb{E}[L^2]$

$$\mathbb{E}[L^2] = \mathrm{Var}(L) + \mathbb{E}[L]^2$$
$$\mathrm{Var}(L) = \underline{\mathbb{E}[\mathrm{Var}(L \mid X)]} + \mathrm{Var}(\mathbb{E}[L \mid X])$$

$$\mathrm{Var}(A + B) = \mathrm{Var}(A) + \mathrm{Var}(A) \quad \text{// independent } A, B$$

$\mathbb{E}[\mathrm{Var}(L \mid X)]$

$$\mathrm{Var}(L \mid X) = \mathrm{Var}\left(\sum_{i=0}^{X} L'_i \,\middle|\, X\right) \quad \text{// independence of } L'_i$$
$$= \sum_{i=1}^{X} \mathrm{Var}\left(L'_i \mid X\right) \quad \text{// independence of } L'_i \text{ and } X$$
$$= \sum_{i=1}^{X} \mathrm{Var}\left(L'_i\right) \quad \text{// All } L'_i, L \text{ equally distributed}$$

Negative Queries in Linear Probing Hash Table

Previously

$$L = 1 + \sum_{i=1}^{X} L'_i$$

where the L'_i follow the same distribution as L

Useful identities for $\mathbb{E}[L^2]$

$$\mathbb{E}[L^2] = \mathrm{Var}(L) + \mathbb{E}[L]^2$$
$$\mathrm{Var}(L) = \underline{\mathbb{E}[\mathrm{Var}(L \mid X)]} + \mathrm{Var}(\mathbb{E}[L \mid X])$$

$$\mathrm{Var}(A + B) = \mathrm{Var}(A) + \mathrm{Var}(A) \quad \text{// independent } A, B$$

$\mathbb{E}[\mathrm{Var}(L \mid X)]$

$$\begin{aligned}
\mathrm{Var}(L \mid X) &= \mathrm{Var}\left(\sum_{i=0}^{X} L'_i \,\middle|\, X\right) && \text{// independence of } L'_i \\
&= \sum_{i=1}^{X} \mathrm{Var}\left(L'_i \mid X\right) && \text{// independence of } L'_i \text{ and } X \\
&= \sum_{i=1}^{X} \mathrm{Var}\left(L'_i\right) && \text{// All } L'_i, L \text{ equally distributed} \\
&= X \cdot \mathrm{Var}\left(L\right)
\end{aligned}$$

Negative Queries in Linear Probing Hash Table

Previously

$$L = 1 + \sum_{i=1}^{X} L'_i$$

where the L'_i follow the same distribution as L

Useful identities for $\mathbb{E}[L^2]$

$$\mathbb{E}[L^2] = \mathrm{Var}(L) + \mathbb{E}[L]^2$$
$$\mathrm{Var}(L) = \underline{\mathbb{E}[\mathrm{Var}(L \mid X)]} + \mathrm{Var}(\mathbb{E}[L \mid X])$$

$$\mathrm{Var}(A + B) = \mathrm{Var}(A) + \mathrm{Var}(A) \quad \text{// independent } A, B$$

$\mathbb{E}[\mathrm{Var}(L \mid X)]$

$$\begin{aligned}
\mathrm{Var}(L \mid X) &= \mathrm{Var}\left(\sum_{i=0}^{X} L'_i \,\middle|\, X\right) && \text{// independence of } L'_i \\
&= \sum_{i=1}^{X} \mathrm{Var}\left(L'_i \mid X\right) && \text{// independence of } L'_i \text{ and } X \\
&= \sum_{i=1}^{X} \mathrm{Var}\left(L'_i\right) && \text{// All } L'_i, L \text{ equally distributed} \\
&= X \cdot \mathrm{Var}\left(L\right)
\end{aligned}$$

$$\mathbb{E}[\mathrm{Var}(L \mid X)] = \mathbb{E}[X \cdot \mathrm{Var}\left(L\right)] = \alpha \mathrm{Var}(L)$$

Negative Queries in Linear Probing Hash Table

Previously

$$L = 1 + \sum_{i=1}^{X} L'_i$$

where the L'_i follow the same distribution as L

Useful identities for $\mathbb{E}[L^2]$

$$\mathbb{E}[L^2] = \mathrm{Var}(L) + \mathbb{E}[L]^2$$
$$\mathrm{Var}(L) = \mathbb{E}[\mathrm{Var}(L \mid X)] + \underline{\mathrm{Var}(\mathbb{E}[L \mid X])}$$

$$\mathrm{Var}(cA) = c^2\mathrm{Var}(A) \quad \text{// constant } c$$

$\mathrm{Var}(\mathbb{E}[L \mid X])$

$$\mathbb{E}[L \mid X] = \mathbb{E}\left[1 + \sum_{i=1}^{X} L'_i \,\middle|\, X\right] \quad \text{// independence of } L'_i \text{ and } X$$
$$= 1 + X \cdot \mathbb{E}[L]$$

Negative Queries in Linear Probing Hash Table

Previously

$$L = 1 + \sum_{i=1}^{X} L'_i$$

where the L'_i follow the same distribution as L

Useful identities for $\mathbb{E}[L^2]$

$$\mathbb{E}[L^2] = \text{Var}(L) + \mathbb{E}[L]^2$$
$$\text{Var}(L) = \mathbb{E}[\text{Var}(L \mid X)] + \underline{\text{Var}(\mathbb{E}[L \mid X])}$$

$$\text{Var}(cA) = c^2\text{Var}(A) \quad \text{// constant } c$$

$\text{Var}(\mathbb{E}[L \mid X])$

$$\mathbb{E}[L \mid X] = \mathbb{E}\left[1 + \sum_{i=1}^{X} L'_i \,\middle|\, X\right] \quad \text{// independence of } L'_i \text{ and } X$$
$$= 1 + X \cdot \mathbb{E}[L]$$

$$\begin{aligned}\text{Var}(\mathbb{E}[L \mid X]) &= \text{Var}(X \cdot \mathbb{E}[L]) \\ &= \mathbb{E}[L]^2 \cdot \text{Var}(X) \\ &= \mathbb{E}[L]^2 \cdot \alpha\end{aligned}$$

Negative Queries in Linear Probing Hash Table

$\mathbb{E}[L^2]$

$$\begin{aligned}\mathrm{Var}(L) &= \mathbb{E}[\mathrm{Var}(L \mid X)] + \mathrm{Var}(\mathbb{E}[L \mid X]) \\ &= \alpha\mathrm{Var}(L) + \mathbb{E}[L]^2 \cdot \alpha\end{aligned}$$

Negative Queries in Linear Probing Hash Table

$\mathbb{E}[L^2]$

$$\begin{aligned}\text{Var}(L) &= \mathbb{E}[\text{Var}(L \mid X)] + \text{Var}(\mathbb{E}[L \mid X]) \\ &= \alpha\text{Var}(L) + \mathbb{E}[L]^2 \cdot \alpha\end{aligned}$$

Solving for Var(L) gives:

$$\text{Var}(L) = \frac{\alpha \cdot \mathbb{E}[L]^2}{1 - \alpha}$$

Negative Queries in Linear Probing Hash Table

$\mathbb{E}[L^2]$

$$\begin{aligned}\mathrm{Var}(L) &= \mathbb{E}[\mathrm{Var}(L \mid X)] + \mathrm{Var}(\mathbb{E}[L \mid X]) \\ &= \alpha \mathrm{Var}(L) + \mathbb{E}[L]^2 \cdot \alpha\end{aligned}$$

Solving for $\mathrm{Var}(L)$ gives:

$$\mathrm{Var}(L) = \frac{\alpha \cdot \mathbb{E}[L]^2}{1-\alpha}$$

Finally:

$$\begin{aligned}\mathbb{E}[L^2] = \mathrm{Var}(L) + \mathbb{E}[L]^2 &= \frac{\alpha \cdot \mathbb{E}[L]^2}{1-\alpha} + \mathbb{E}[L]^2 \\ &= \frac{\alpha}{(1-\alpha)^3} + \frac{1}{(1-\alpha)^2} = \frac{\alpha}{(1-\alpha)^3} + \frac{1-\alpha}{(1-\alpha)^3} \\ &= \frac{1}{(1-\alpha)^3}\end{aligned}$$

Negative Queries in Linear Probing Hash Table

$\mathbb{E}[L^2]$

$$\begin{aligned}\mathrm{Var}(L) &= \mathbb{E}[\mathrm{Var}(L \mid X)] + \mathrm{Var}(\mathbb{E}[L \mid X]) \\ &= \alpha \mathrm{Var}(L) + \mathbb{E}[L]^2 \cdot \alpha\end{aligned}$$

Solving for $\mathrm{Var}(L)$ gives:

$$\mathrm{Var}(L) = \frac{\alpha \cdot \mathbb{E}[L]^2}{1-\alpha}$$

Finally:

$$\begin{aligned}\mathbb{E}[L^2] = \mathrm{Var}(L) + \mathbb{E}[L]^2 &= \frac{\alpha \cdot \mathbb{E}[L]^2}{1-\alpha} + \mathbb{E}[L]^2 \\ &= \frac{\alpha}{(1-\alpha)^3} + \frac{1}{(1-\alpha)^2} = \frac{\alpha}{(1-\alpha)^3} + \frac{1-\alpha}{(1-\alpha)^3} \\ &= \frac{1}{(1-\alpha)^3}\end{aligned}$$

$\mathbb{E}[T^-]$

$$\begin{aligned}\mathbb{E}[T^-] &= \frac{1-\alpha}{2}(\mathbb{E}[L] + \mathbb{E}[L^2]) \\ &= \frac{1-\alpha}{2}\left(\frac{1}{1-\alpha} + \frac{1}{(1-\alpha)^3}\right) \\ &= \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)\end{aligned}$$

Negative Queries in Linear Probing Hash Table

$\mathbb{E}[L^2]$

$$\begin{aligned}\mathrm{Var}(L) &= \mathbb{E}[\mathrm{Var}(L \mid X)] + \mathrm{Var}(\mathbb{E}[L \mid X]) \\ &= \alpha \mathrm{Var}(L) + \mathbb{E}[L]^2 \cdot \alpha\end{aligned}$$

Solving for $\mathrm{Var}(L)$ gives:

$$\mathrm{Var}(L) = \frac{\alpha \cdot \mathbb{E}[L]^2}{1-\alpha}$$

Finally:

$$\begin{aligned}\mathbb{E}[L^2] = \mathrm{Var}(L) + \mathbb{E}[L]^2 &= \frac{\alpha \cdot \mathbb{E}[L]^2}{1-\alpha} + \mathbb{E}[L]^2 \\ &= \frac{\alpha}{(1-\alpha)^3} + \frac{1}{(1-\alpha)^2} = \frac{\alpha}{(1-\alpha)^3} + \frac{1-\alpha}{(1-\alpha)^3} \\ &= \frac{1}{(1-\alpha)^3}\end{aligned}$$

$\mathbb{E}[T^-]$

$$\begin{aligned}\mathbb{E}[T^-] &= \frac{1-\alpha}{2}(\mathbb{E}[L] + \mathbb{E}[L^2]) \\ &= \frac{1-\alpha}{2}\left(\frac{1}{1-\alpha} + \frac{1}{(1-\alpha)^3}\right) \\ &= \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)\end{aligned}$$

Remark

We assumed that the number of keys in each slot are independent random variables. This is not entirely true, but for large n we can come arbitrarily close to this ideal.

Hash table overview

Metrics

- Exact query time, i.e. counting the number of probes
  - Positive: Query an existing key (we only consider the average over all existing keys)
  - Negative: Query a non-existing key (today's focus)
- Insertion time
- Space: Memory needed by the hash table (b bit elements and p bit pointers)

Technique	Avg. Positive Query	Negative Query	Insertion	Bits per Key
Chaining	$\mathcal{O}(1)$ expected	$1+\alpha$ expected	$=$ neg. query [1]	$p/\alpha + b + p$
Cuckoo Hashing	≤ 2	2	$\mathcal{O}(1)$ expected	b/α
Linear Probing	$\frac{1}{2}(1+\frac{1}{1-\alpha})$ expected	$\frac{1}{2}(1+\frac{1}{(1-\alpha)^2})$ expected	$=$ neg. query	b/α

[1] even in the worst-case if we can assume the key does not exist yet then we can insert at the beginning of the list

Positive Queries in Linear Probing Hash Table

Observation

Start with an empty hash table of m slots and insert n keys one after another. Then, the number of probes when querying a key corresponds exactly to the number of probes that were needed to insert it. Hence, the expected number of probes for a positive query at load α is the average number of probes over all insertions.

Positive Queries in Linear Probing Hash Table

Observation

Start with an empty hash table of m slots and insert n keys one after another. Then, the number of probes when querying a key corresponds exactly to the number of probes that were needed to insert it. Hence, the expected number of probes for a positive query at load α is the average number of probes over all insertions.

Average expected positive query probes T^+

$$\begin{aligned}\mathbb{E}[T^+] &= \frac{1}{n}\sum_{i=0}^{n-1}\frac{1}{2}\left(1-\frac{1}{(1-i/m)^2}\right) && \text{use } x := i/m, n \to \infty \\ &= \frac{1}{\alpha}\int_0^{\alpha}\frac{1}{2}\left(1-\frac{1}{(1-x)^2}\right)dx \\ &= \frac{1}{2}\left(1-\frac{1}{1-\alpha}\right)\end{aligned}$$

Positive Queries in Linear Probing Hash Table

Observation

Start with an empty hash table of m slots and insert n keys one after another. Then, the number of probes when querying a key corresponds exactly to the number of probes that were needed to insert it. Hence, the expected number of probes for a positive query at load α is the average number of probes over all insertions.

Average expected positive query probes T^+

$$\begin{aligned}\mathbb{E}[T^+] &= \frac{1}{n}\sum_{i=0}^{n-1}\frac{1}{2}\left(1-\frac{1}{(1-i/m)^2}\right) && \text{use } x := i/m, n \to \infty \\ &= \frac{1}{\alpha}\int_0^{\alpha}\frac{1}{2}\left(1-\frac{1}{(1-x)^2}\right)dx \\ &= \frac{1}{2}\left(1-\frac{1}{1-\alpha}\right)\end{aligned}$$

Does this hold even after deletions?

Yes.
The order of keys within clusters does not effect the average probing distance. The observation applies at any moment.

Positive Queries in Linear Probing Hash Table

Observation

Start with an empty hash table of m slots and insert n keys one after another. Then, the number of probes when querying a key corresponds exactly to the number of probes that were needed to insert it. Hence, the expected number of probes for a positive query at load α is the average number of probes over all insertions.

Average expected positive query probes T^+

$$\begin{aligned}\mathbb{E}[T^+] &= \frac{1}{n}\sum_{i=0}^{n-1}\frac{1}{2}\left(1-\frac{1}{(1-i/m)^2}\right) && \text{use } x := i/m, n \to \infty \\ &= \frac{1}{\alpha}\int_0^{\alpha}\frac{1}{2}\left(1-\frac{1}{(1-x)^2}\right)dx \\ &= \frac{1}{2}\left(1-\frac{1}{1-\alpha}\right)\end{aligned}$$

Does this hold even after deletions?

Yes.
The order of keys within clusters does not effect the average probing distance. The observation applies at any moment.

Beware

The result is only a average over all keys. E.g., querying the key that was just inserted will take the same number of probes as the insertions:
$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$.

Hash table overview

Metrics

- Exact query time, i.e. counting the number of probes
  - Positive: Query an existing key (we only consider the average over all existing keys)
  - Negative: Query a non-existing key (today's focus)
- Insertion time
- Space: Memory needed by the hash table (b bit elements and p bit pointers)

Technique	Avg. Positive Query	Negative Query	Insertion	Bits per Key
Chaining	$\mathcal{O}(1)$ expected	$1+\alpha$ expected	$=$ neg. query [1]	$p/\alpha + b + p$
Cuckoo Hashing	≤ 2	2	$\mathcal{O}(1)$ expected	b/α
Linear Probing	$\frac{1}{2}(1+\frac{1}{1-\alpha})$ expected	$\frac{1}{2}(1+\frac{1}{(1-\alpha)^2})$ expected	$=$ neg. query	b/α

[1] even in the worst-case if we can assume the key does not exist yet then we can insert at the beginning of the list

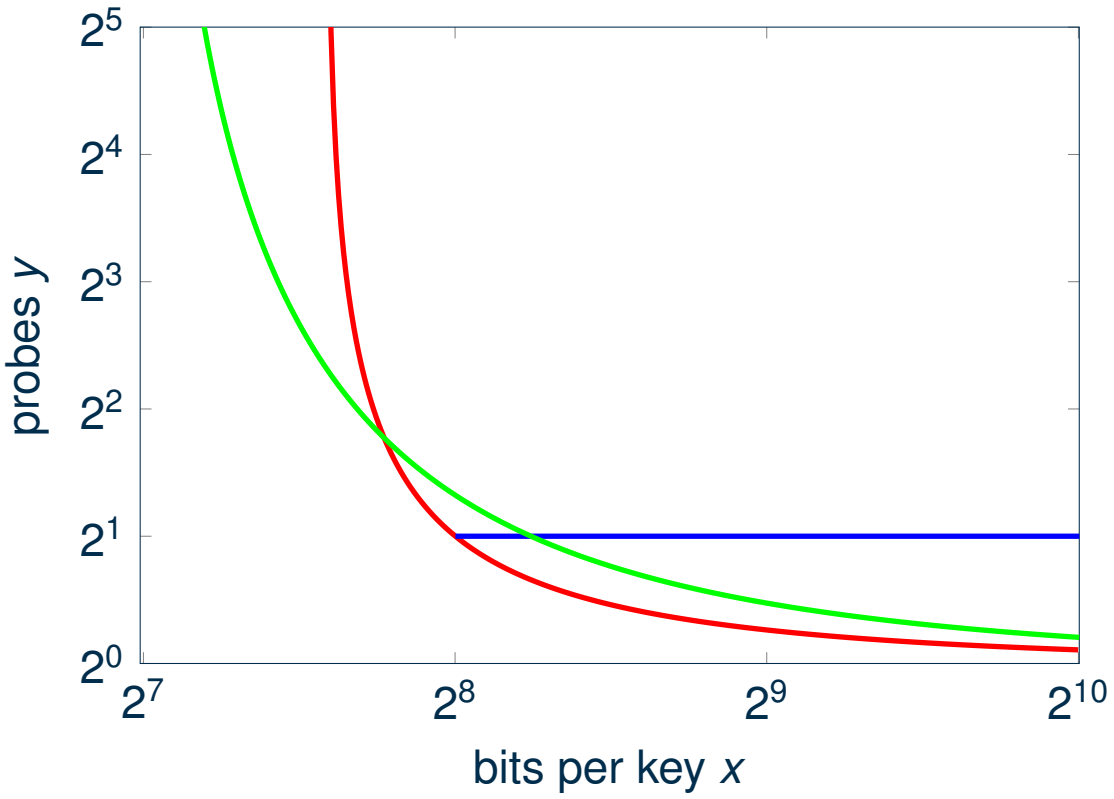
trade-off: space and negative queries

trade-off

We know for each technique how the number of probes depends on the load α.
But that does not give a meaningful way to compare the techniques.
So let's compare the techniques for equal space consumption instead.
Let $x(\alpha)$ be the space in bits per key and $y(\alpha)$ the expected number of probes of a negative query.
We invert $x(\alpha)$ to obtain $y(x)$.

Technique	**Negative Query** $y(\alpha)$	**Bits per Key** $x(\alpha)$	$\alpha(x)$	$y(x)$
Chaining $(0 < \alpha)$	$1 + \alpha$ expected	$p/\alpha + b + p$	$\frac{p}{x-b+p}$	$1 + \frac{p}{x-b+p}$ expected
Cuckoo Hashing $(0 < \alpha < 1/2)$	2	b/α	b/x	2
Linear Probing $(0 < \alpha < 1)$	$\frac{1}{2}(1 + \frac{1}{(1-\alpha)^2})$ expected	b/α	b/x	$\frac{1}{2}(1 + \frac{1}{(1-b/x)^2})$ expected

trade-off: space and negative queries

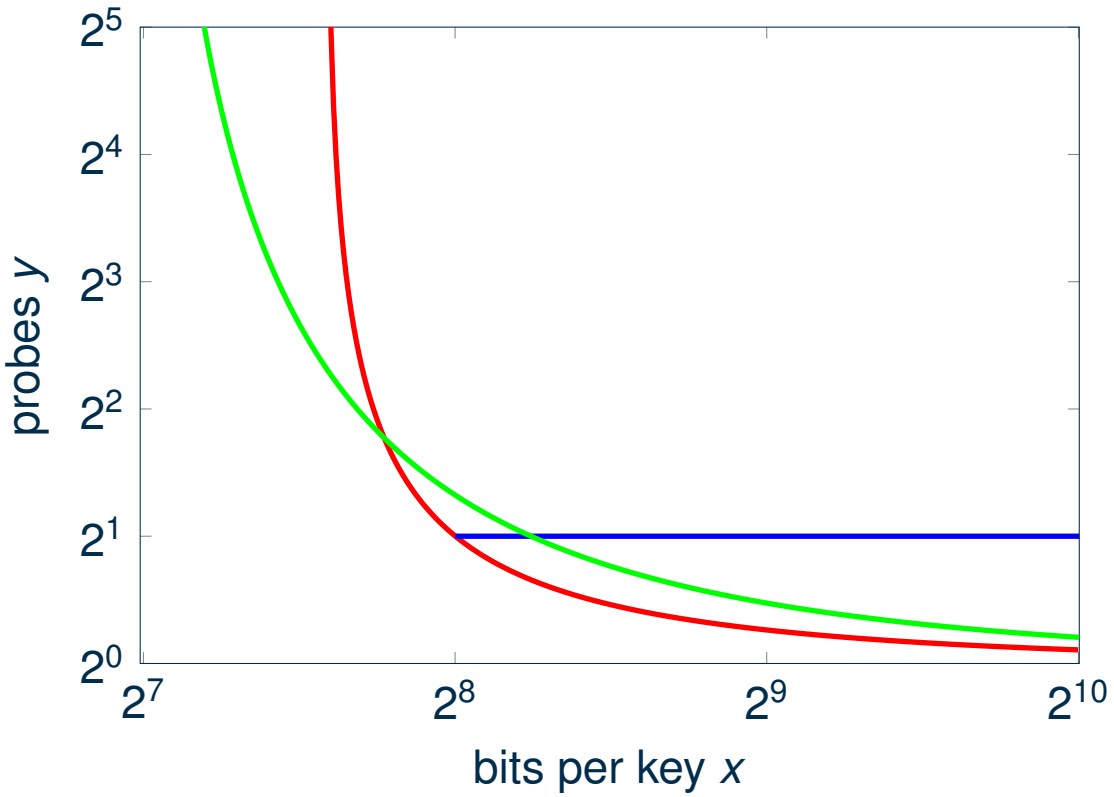


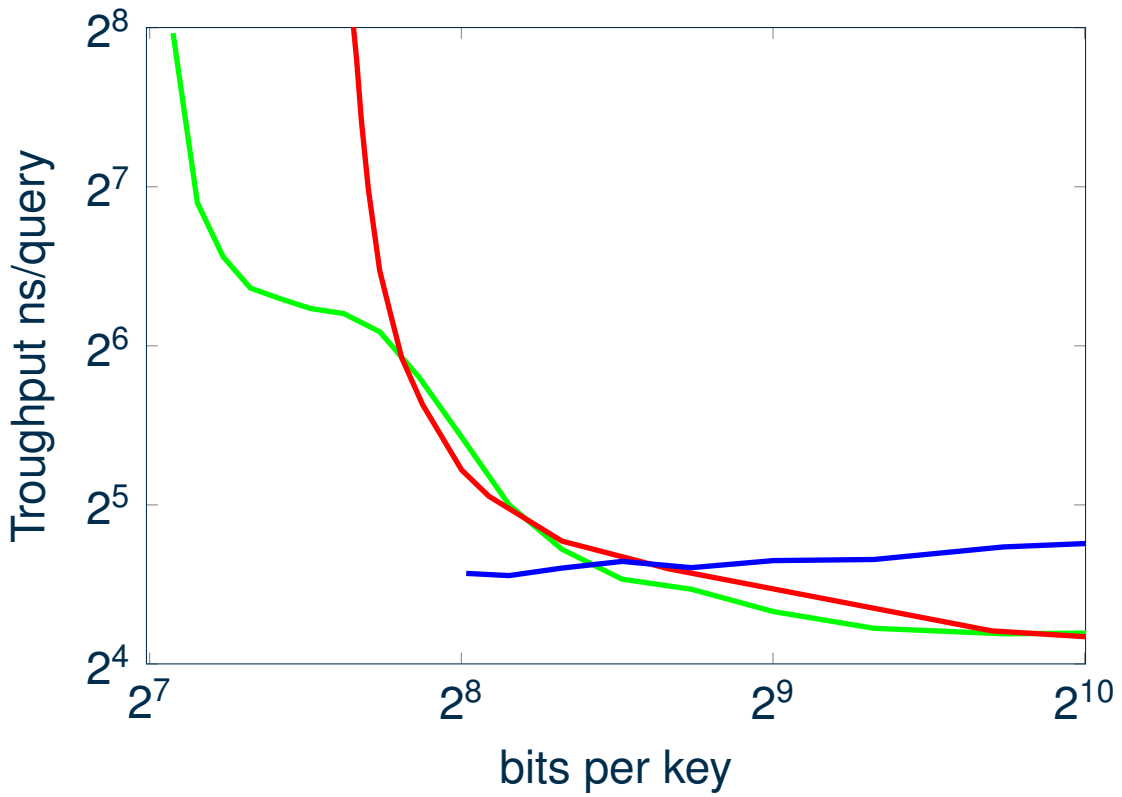
- Chaining $y = 1 + \frac{64}{x-(128+64)}$
- Cuckoo Hashing $y = 2$
- Linear Probing $y = \frac{1}{2}\left(1 + \frac{1}{(1-128/x)^2}\right)$

Benchmark

$n = 1\,000\,000$, $p = 64$ bit pointer and $b = 128$ bit (e.g. a hash set on 128 bit keys or a hash map on 64 bit keys and 64 bit values).

trade-off: space and negative queries



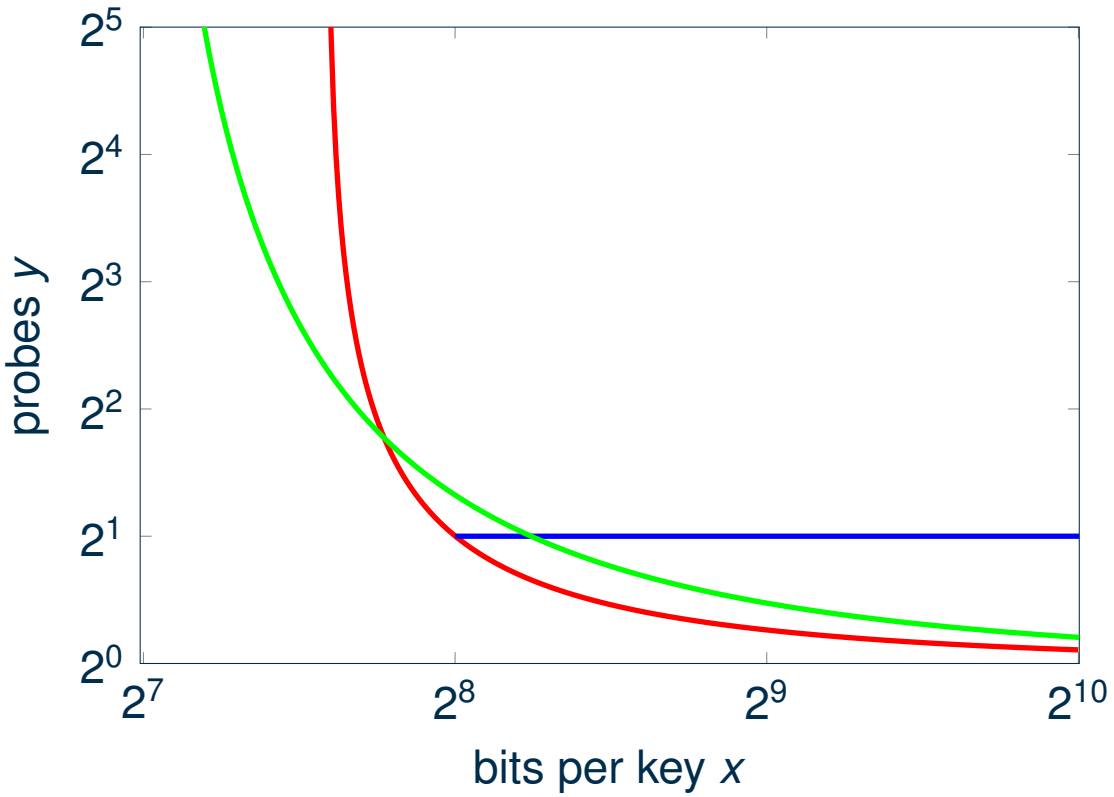


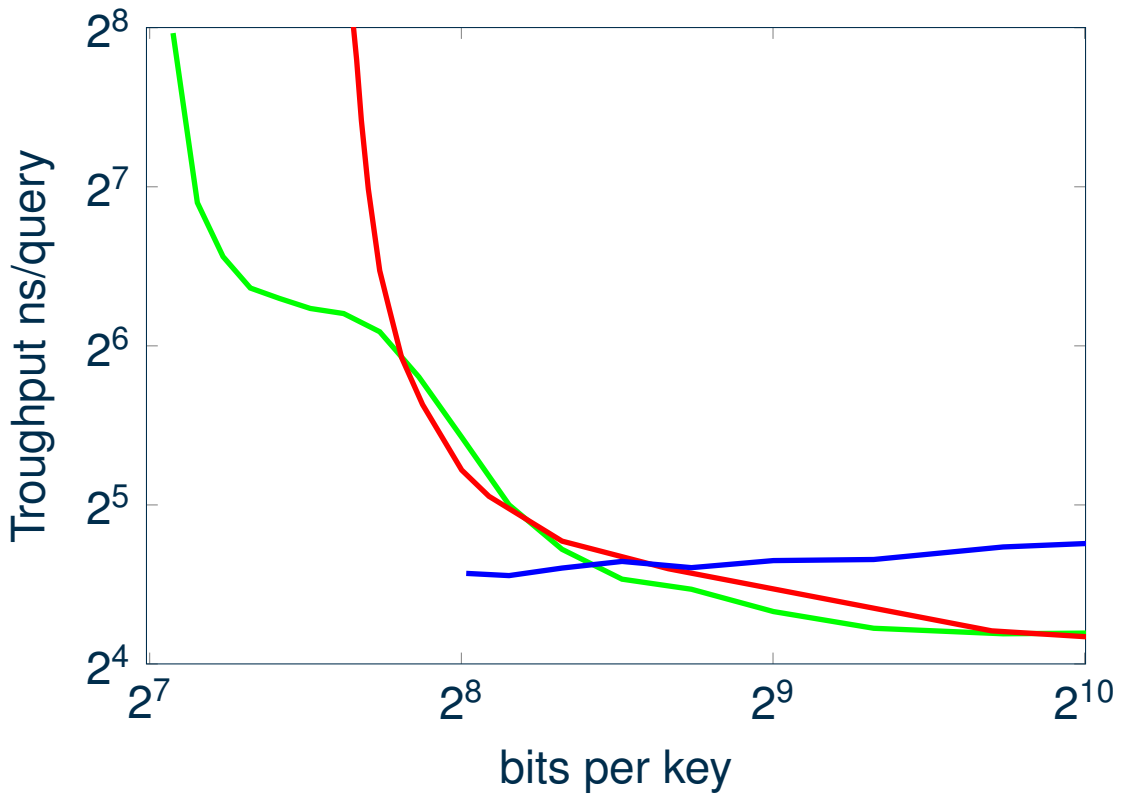
- Chaining $y = 1 + \frac{64}{x-(128+64)}$
- Cuckoo Hashing $y = 2$
- Linear Probing $y = \frac{1}{2}\left(1 + \frac{1}{(1-128/x)^2}\right)$

Benchmark

$n = 1\,000\,000$, $p = 64$ bit pointer and $b = 128$ bit (e.g. a hash set on 128 bit keys or a hash map on 64 bit keys and 64 bit values).

trade-off: space and negative queries





- Chaining $y = 1 + \frac{64}{x-(128+64)}$
- Cuckoo Hashing $y = 2$
- Linear Probing $y = \frac{1}{2}\left(1 + \frac{1}{(1-128/x)^2}\right)$

Observations

- Chaining performs worse then predicted.
- What is that bump in the probing curve?

Hash table overview

Metrics

- Exact query time, i.e. counting the number of probes
  - Positive: Query an existing key (we only consider the average over all existing keys)
  - Negative: Query a non-existing key (today's focus)
- Insertion time
- Space: Memory needed by the hash table (b bit elements and p bit pointers)

Technique	Avg. Positive Query	Negative Query	Insertion	Bits per Key	Comment
Chaining	$\mathcal{O}(1)$ expected	$1+\alpha$ expected	$=$ neg. query [1]	$p/\alpha + b + p$	Stable
Cuckoo Hashing	≤ 2	2	$\mathcal{O}(1)$ expected	b/α	worst-case $\mathcal{O}(1)$ queries!
Linear Probing	$\frac{1}{2}(1+\frac{1}{1-\alpha})$ expected	$\frac{1}{2}(1+\frac{1}{(1-\alpha)^2})$ expected	$=$ neg. query	b/α	cache friendliness

[1] even in the worst-case if we can assume the key does not exist yet then we can insert at the beginning of the list

Hash table overview

Metrics

- Exact query time, i.e. counting the number of probes
  - Positive: Query an existing key (we only consider the average over all existing keys)
  - Negative: Query a non-existing key (today's focus)
- Insertion time
- Space: Memory needed by the hash table (b bit elements and p bit pointers)

Technique	Avg. Positive Query	Negative Query	Insertion	Bits per Key	Comment
Chaining	$\mathcal{O}(1)$ expected	$1+\alpha$ expected	$=$ neg. query [1]	$p/\alpha + b + p$	Stable
Cuckoo Hashing	≤ 2	2	$\mathcal{O}(1)$ expected	b/α	worst-case $\mathcal{O}(1)$ queries!
Linear Probing	$\frac{1}{2}(1+\frac{1}{1-\alpha})$ expected	$\frac{1}{2}(1+\frac{1}{(1-\alpha)^2})$ expected	$=$ neg. query	b/α	cache friendliness
Quadratic Probing	$\mathcal{O}(\log \frac{1}{1-\alpha})$ [2]	$\mathcal{O}(\frac{1}{1-\alpha})$ [2]	$\mathcal{O}(\frac{1}{1-\alpha})$ [2]	b/α	current research (for decades)

[1] even in the worst-case if we can assume the key does not exist yet then we can insert at the beginning of the list
[2] Conjectured

Hash table overview

Metrics

- Exact query time, i.e. counting the number of probes
  - Positive: Query an existing key (we only consider the average over all existing keys)
  - Negative: Query a non-existing key (today's focus)
- Insertion time
- Space: Memory needed by the hash table (b bit elements and p bit pointers)

Technique	Avg. Positive Query	Negative Query	Insertion	Bits per Key	Comment
Chaining	$\mathcal{O}(1)$ expected	$1+\alpha$ expected	$=$ neg. query [1]	$p/\alpha + b + p$	Stable
Cuckoo Hashing	≤ 2	2	$\mathcal{O}(1)$ expected	b/α	worst-case $\mathcal{O}(1)$ queries!
Linear Probing	$\frac{1}{2}(1+\frac{1}{1-\alpha})$ expected	$\frac{1}{2}(1+\frac{1}{(1-\alpha)^2})$ expected	$=$ neg. query	b/α	cache friendliness
Quadratic Probing	$\mathcal{O}(\log \frac{1}{1-\alpha})$ [2]	$\mathcal{O}(\frac{1}{1-\alpha})$ [2]	$\mathcal{O}(\frac{1}{1-\alpha})$ [2]	b/α	current research (for decades)
Next Lecture	$\mathcal{O}(1)$ worst-case	$\mathcal{O}(1)$ worst-case	$\mathcal{O}(1)$ whp [3]	$(1+o(1))b$	strong simultaneous guarantees

[1] even in the worst-case if we can assume the key does not exist yet then we can insert at the beginning of the list

[2] Conjectured

[3] by with high probability (whp) we mean with probability $1 - 1/poly(n)$

Robin Hood Hashing

Robin Hood Hashing

On the next slide, Stefan Walzer explained Robin Hood hashing on the board, resulting in $\mathcal{O}\left(\frac{1}{1-\alpha}\right)$ query time (positive and negative).

Ordered Linear Probing & History Independence

Assumptions

- table stores S of size $n = |S| = \underbrace{(1-\varepsilon)}_{\alpha} \cdot m$
- no deletions

We have seen:

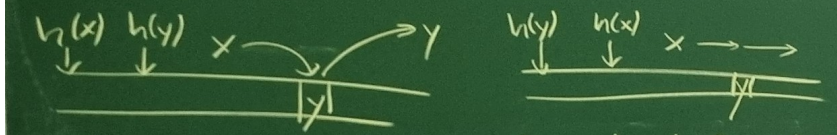
avg positive query time is $O(1/\varepsilon)$ in exp.

neg " " " $O(1/\varepsilon^2)$ in exp.

Goal: all query times $O(1/\varepsilon)$ in exp.

Robin Hood Ordering

If x probes slot occupied by y whoever comes from further away stays the other key keeps probing.



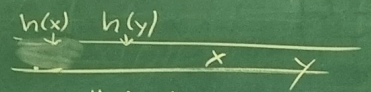
Tie-Breaking via some other hash function.

Def A dynamic data structure is history independent if its binary representation only depends on the data it currently stores.

Note

- prevents leaking information about past states.

(- perfect space efficiency requires HI)
- normal Lin probe is not HI

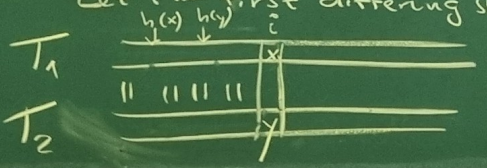


"leaks" that x was inserted before y.

Claim Robin Hood Linear Probing is HI.

Proof Simplified with assumption: "no wrap around"

Assume for contradiction. Two insertion orders result in distinct states. Let i be first differing slot



wlog x has priority over y

x must have probed i in T_2.

the reason it's not placed there is some other key z with priority over x.

priorities of keys placed in $T_2[i]$ can only increase over time

$\Rightarrow$ y was placed there eventually.

Consequence of HI:

$\Rightarrow \forall x \in S$: exp query time of x is $O(1/\varepsilon)$

We can modify the query algo to stop when finding a key that the queried key has priority over.

Consider $y \notin S$.

- its query time is equal to the query it would have if we inserted it now.

$\hookrightarrow$ that's a positive query time

$\Rightarrow O(1/\varepsilon)$.

Possible Exam Questions

- What is a hash function? What is random in a hash table?
- What is the negative query time of chaining? Can you derive it?
- When is the intuition behind the threshold behavior at $\alpha = 0.5$ in cuckoo hashing?
- How did we derive the negative query time of linear probing? (The exact calculations are not required)
- Based on the negative query time, how did we obtain the average positive query time?
- What are advantages of chaining, linear probing and cuckoo hashing over the respective other two techniques?
- What is history independence?
- Why does Robin Hood hashing improve the expected query time?