Algorithms and Data Structures

Part 4: Hashing

Department of Mathematics
Stockholm University

Hashing

Part 4: Hashing

Many applications require dictionary, i.e., a dynamic set that supports only the operations INSERT, SEARCH, DELETE.

A hash table is an effective data structure for implementing dictionaries.

Side note: the built-in dictionaries of Python are implemented with hash tables.

Although searching for an element in a hash table can take as long as searching for an element in a linked list - $\Theta(n)$ time in the worst case - in practice, hashing performs extremely well.

Under reasonable assumptions, the average time to search for an element in a hash table is $O(1)$.

Part 4: Hashing

Many applications require dictionary, i.e., a dynamic set that supports only the operations INSERT, SEARCH, DELETE.

A hash table is an effective data structure for implementing dictionaries.

Side note: the built-in dictionaries of Python are implemented with hash tables.

Although searching for an element in a hash table can take as long as searching for an element in a linked list - $\Theta(n)$ time in the worst case - in practice, hashing performs extremely well.

Under reasonable assumptions, the average time to search for an element in a hash table is $O(1)$.

Part 4: Hashing

Many applications require dictionary, i.e., a dynamic set that supports only the operations INSERT, SEARCH, DELETE.

A hash table is an effective data structure for implementing dictionaries.

Side note: the built-in dictionaries of Python are implemented with hash tables.

Although searching for an element in a hash table can take as long as searching for an element in a linked list - $\Theta(n)$ time in the worst case - in practice, hashing performs extremely well.

Under reasonable assumptions, the average time to search for an element in a hash table is $O(1)$.

Part 4: Hashing

Many applications require dictionary, i.e., a dynamic set that supports only the operations INSERT, SEARCH, DELETE.

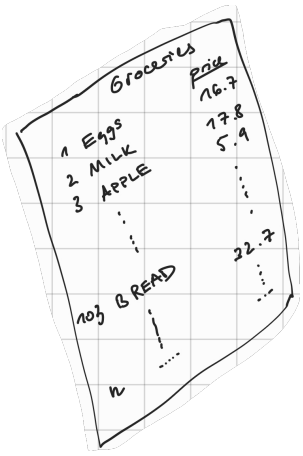
A hash table is an effective data structure for implementing dictionaries.

Side note: the built-in dictionaries of Python are implemented with hash tables.

Although searching for an element in a hash table can take as long as searching for an element in a linked list - $\Theta(n)$ time in the worst case - in practice, hashing performs extremely well.

Under reasonable assumptions, the average time to search for an element in a hash table is $O(1)$.

Part 4: A comic example



POV: You work at a grocery store and have this "list" of items in unsorted order together with prices.

When a customer buys products, you have to look up the prices in this huge list of n elements.

How much is a bread? $O(n)$ time :(!

What if we have a function

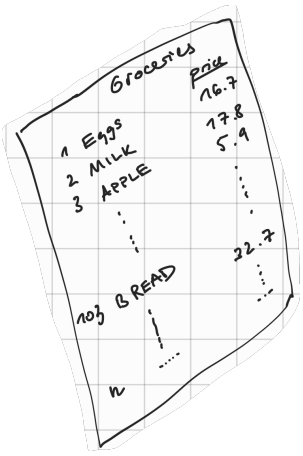
$$h\colon \text{products} \to \text{listNr}$$

$$\begin{aligned} \text{EGGS} &\mapsto 1 \\ \text{MILK} &\mapsto 2 \\ &\vdots \\ \text{BREAD} &\mapsto 103 \\ &\vdots \end{aligned}$$

Then, look up at takes $O(1)$ time

This is the essential idea of hashing, i.e, the function h maps keys (here: products) together with their satellite data (here: prices) to some entry in an array (hash table).

Part 4: A comic example



POV: You work at a grocery store and have this "list" of items in unsorted order together with prices.

When a customer buys products, you have to look up the prices in this huge list of n elements.

How much is a bread? $O(n)$ time :(!

What if we have a function

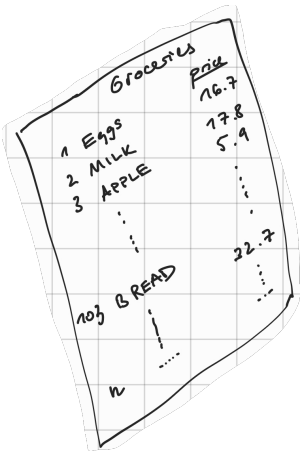
$$h\colon \text{products} \to \text{listNr}$$

$$\begin{aligned} \text{EGGS} &\mapsto 1 \\ \text{MILK} &\mapsto 2 \\ &\vdots \\ \text{BREAD} &\mapsto 103 \\ &\vdots \end{aligned}$$

Then, look up at takes $O(1)$ time

This is the essential idea of hashing, i.e, the function h maps keys (here: products) together with their satellite data (here: prices) to some entry in an array (hash table).

Part 4: A comic example



POV: You work at a grocery store and have this "list" of items in unsorted order together with prices.

When a customer buys products, you have to look up the prices in this huge list of n elements.

How much is a bread? $O(n)$ time :(!

What if we have a function

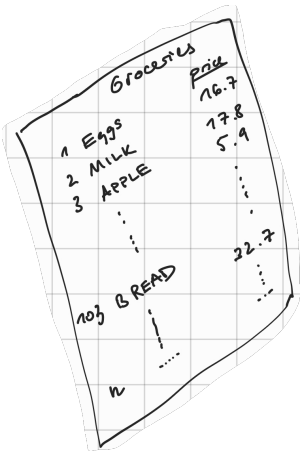
$$h\colon \text{products} \to \text{listNr}$$

$$\begin{aligned} \text{EGGS} &\mapsto 1 \\ \text{MILK} &\mapsto 2 \\ &\vdots \\ \text{BREAD} &\mapsto 103 \\ &\vdots \end{aligned}$$

Then, look up at takes $O(1)$ time

This is the essential idea of hashing, i.e, the function h maps keys (here: products) together with their satellite data (here: prices) to some entry in an array (hash table).

Part 4: A comic example



POV: You work at a grocery store and have this "list" of items in unsorted order together with prices.

When a customer buys products, you have to look up the prices in this huge list of n elements.

How much is a bread? $O(n)$ time :(!

What if we have a function

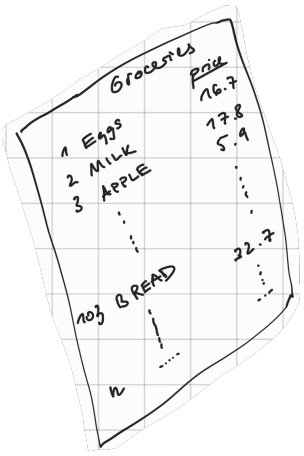
$$h\colon \text{products} \to \text{listNr}$$

$$\begin{aligned} &\text{EGGS} \mapsto 1\\ &\text{MILK} \mapsto 2\\ &\vdots\\ &\text{BREAD} \mapsto 103\\ &\vdots \end{aligned}$$

Then, look up at takes $O(1)$ time

This is the essential idea of hashing, i.e, the function h maps keys (here: products) together with their satellite data (here: prices) to some entry in an array (hash table).

Part 4: A comic example



POV: You work at a grocery store and have this "list" of items in unsorted order together with prices.

When a customer buys products, you have to look up the prices in this huge list of n elements.

How much is a bread? $O(n)$ time :(!

What if we have a function

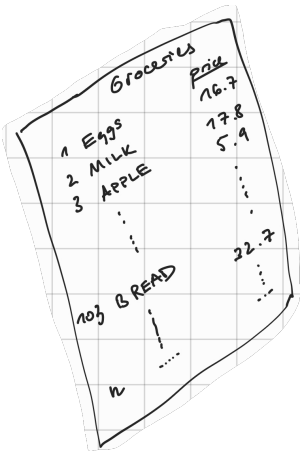
$$h\colon \text{products} \to \text{listNr}$$

$$\begin{aligned} &\text{EGGS} \mapsto 1\\ &\text{MILK} \mapsto 2\\ &\vdots\\ &\text{BREAD} \mapsto 103\\ &\vdots \end{aligned}$$

Then, look up at takes $O(1)$ time

This is the essential idea of hashing, i.e, the function h maps keys (here: products) together with their satellite data (here: prices) to some entry in an array (hash table).

Part 4: A comic example



POV: You work at a grocery store and have this "list" of items in unsorted order together with prices.

When a customer buys products, you have to look up the prices in this huge list of n elements.

How much is a bread? $O(n)$ time :(!

What if we have a function

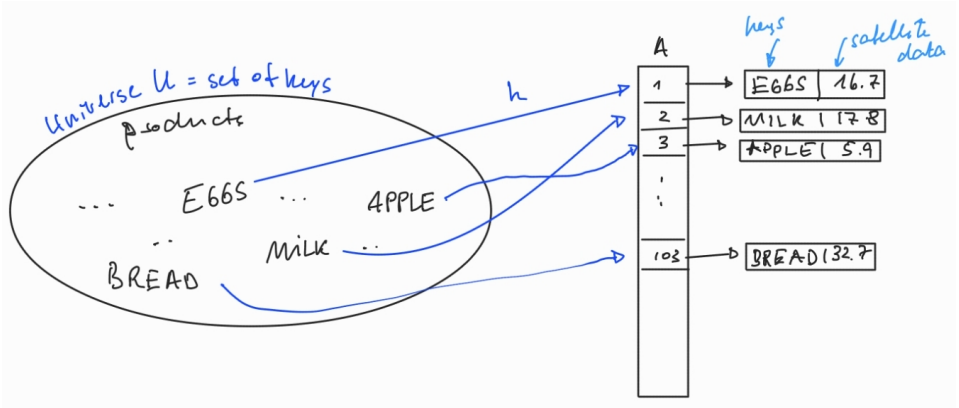
$$h\colon \text{products} \to \text{listNr}$$

$$\begin{aligned} &\text{EGGS} \mapsto 1\\ &\text{MILK} \mapsto 2\\ &\vdots\\ &\text{BREAD} \mapsto 103\\ &\vdots \end{aligned}$$

Then, look up at takes $O(1)$ time

This is the essential idea of hashing, i.e, the function h maps keys (here: products) together with their satellite data (here: prices) to some entry in an array (hash table).

Part 4: A comic example



Idea: Use hash function $h\colon U \to \{0, \ldots, m-1\}$ where $m \leq |U|$

Part 4: Interpreting keys as numbers

Interpreting keys as numbers

Most hash functions assume that the universe of keys is the set $\mathbb{N}_0 = \{0, 1, 2, \ldots\}$

If that is not the case, we need to interpret (injective mapping) them as numbers

Example: A string $s = s_0 s_1 \ldots s_r$ of characters may be interpreted as a number in the following way:

Look up ascii-code of each character (`www.ascii-code.com`)

Each character s_i is associated with a number a_i in $\{0, \ldots, 127\}$

Then s can be interpreted as a number k in a base 128 system:

$$k = \sum_{i=0}^{r} a_i \cdot 128^i$$

Example: for string $s =$`Mia` we have `M` $= 77$, `i` $= 105$, `a` $= 97$ (ASCII)

Thus, `Mia` corresponds to number $k = 77 + 105 \cdot 128 + 97 \cdot 128^2 = 1602765$

In what follows, we (mainly) assume that the keys are natural numbers.

Part 4: Interpreting keys as numbers

Interpreting keys as numbers

Most hash functions assume that the universe of keys is the set $\mathbb{N}_0 = \{0, 1, 2, \ldots\}$

If that is not the case, we need to interpret (injective mapping) them as numbers

Example: A string $s = s_0 s_1 \ldots s_r$ of characters may be interpreted as a number in the following way:

Look up ascii-code of each character (`www.ascii-code.com`)

Each character s_i is associated with a number a_i in $\{0, \ldots, 127\}$

Then s can be interpreted as a number k in a base 128 system:

$$k = \sum_{i=0}^{r} a_i \cdot 128^i$$

Example: for string $s =$`Mia` we have `M` $= 77$, `i` $= 105$, `a` $= 97$ (ASCII)

Thus, `Mia` corresponds to number $k = 77 + 105 \cdot 128 + 97 \cdot 128^2 = 1602765$

In what follows, we (mainly) assume that the keys are natural numbers.

Part 4: Interpreting keys as numbers

Interpreting keys as numbers

Most hash functions assume that the universe of keys is the set $\mathbb{N}_0 = \{0, 1, 2, \ldots\}$

If that is not the case, we need to interpret (injective mapping) them as numbers

Example: A string $s = s_0 s_1 \ldots s_r$ of characters may be interpreted as a number in the following way:

Look up ascii-code of each character (`www.ascii-code.com`)

Each character s_i is associated with a number a_i in $\{0, \ldots, 127\}$

Then s can be interpreted as a number k in a base 128 system:

$$k = \sum_{i=0}^{r} a_i \cdot 128^i$$

Example: for string $s =$`Mia` we have `M` $= 77$, `i` $= 105$, `a` $= 97$ (ASCII)

Thus, `Mia` corresponds to number $k = 77 + 105 \cdot 128 + 97 \cdot 128^2 = 1602765$

In what follows, we (mainly) assume that the keys are natural numbers.

Part 4: Interpreting keys as numbers

Interpreting keys as numbers

Most hash functions assume that the universe of keys is the set $\mathbb{N}_0 = \{0, 1, 2, \ldots\}$

If that is not the case, we need to interpret (injective mapping) them as numbers

Example: A string $s = s_0 s_1 \ldots s_r$ of characters may be interpreted as a number in the following way:

Look up ascii-code of each character (`www.ascii-code.com`)

Each character s_i is associated with a number a_i in $\{0, \ldots, 127\}$

Then s can be interpreted as a number k in a base 128 system:

$$k = \sum_{i=0}^{r} a_i \cdot 128^i$$

Example: for string $s =$`Mia` we have `M` $= 77$, `i` $= 105$, `a` $= 97$ (ASCII)

Thus, `Mia` corresponds to number $k = 77 + 105 \cdot 128 + 97 \cdot 128^2 = 1602765$

In what follows, we (mainly) assume that the keys are natural numbers.

Part 4: Interpreting keys as numbers

Interpreting keys as numbers

Most hash functions assume that the universe of keys is the set $\mathbb{N}_0 = \{0, 1, 2, \ldots\}$

If that is not the case, we need to interpret (injective mapping) them as numbers

Example: A string $s = s_0 s_1 \ldots s_r$ of characters may be interpreted as a number in the following way:

Look up ascii-code of each character (`www.ascii-code.com`)

Each character s_i is associated with a number a_i in $\{0, \ldots, 127\}$

Then s can be interpreted as a number k in a base 128 system:

$$k = \sum_{i=0}^{r} a_i \cdot 128^i$$

Example: for string $s =$`Mia` we have `M` $= 77$, `i` $= 105$, `a` $= 97$ (ASCII)

Thus, `Mia` corresponds to number $k = 77 + 105 \cdot 128 + 97 \cdot 128^2 = 1602765$

In what follows, we (mainly) assume that the keys are natural numbers.

Part 4: Interpreting keys as numbers

Interpreting keys as numbers

Most hash functions assume that the universe of keys is the set $\mathbb{N}_0 = \{0, 1, 2, \ldots\}$

If that is not the case, we need to interpret (injective mapping) them as numbers

Example: A string $s = s_0 s_1 \ldots s_r$ of characters may be interpreted as a number in the following way:

Look up ascii-code of each character (`www.ascii-code.com`)

Each character s_i is associated with a number a_i in $\{0, \ldots, 127\}$

Then s can be interpreted as a number k in a base 128 system:

$$k = \sum_{i=0}^{r} a_i \cdot 128^i$$

Example: for string $s =$`Mia` we have `M` $= 77$, `i` $= 105$, `a` $= 97$ (ASCII)

Thus, `Mia` corresponds to number $k = 77 + 105 \cdot 128 + 97 \cdot 128^2 = 1602765$

In what follows, we (mainly) assume that the keys are natural numbers.

Part 4: Interpreting keys as numbers

Interpreting keys as numbers

Most hash functions assume that the universe of keys is the set $\mathbb{N}_0 = \{0, 1, 2, \ldots\}$

If that is not the case, we need to interpret (injective mapping) them as numbers

Example: A string $s = s_0 s_1 \ldots s_r$ of characters may be interpreted as a number in the following way:

Look up ascii-code of each character (`www.ascii-code.com`)

Each character s_i is associated with a number a_i in $\{0, \ldots, 127\}$

Then s can be interpreted as a number k in a base 128 system:

$$k = \sum_{i=0}^{r} a_i \cdot 128^i$$

Example: for string $s =$`Mia` we have `M` $= 77$, `i` $= 105$, `a` $= 97$ (ASCII)

Thus, `Mia` corresponds to number $k = 77 + 105 \cdot 128 + 97 \cdot 128^2 = 1602765$

In what follows, we (mainly) assume that the keys are natural numbers.

Part 4: Notation

Let $T[0..m-1]$ be an array of size m (hash table).

Let U be the "universe" of all potential keys. In practice, we are often interested in subsets $K \subseteq U$ only.

The size m of the hash table is typically much less than $|U|$.

A hash function

$$h\colon U \to \{0, \ldots, m-1\}$$

is a map that assigns to each key $k \in U$ a number $h(k) \in \{0, \ldots, m-1\}$ as a potential slot in the hash table T.

We say that an element with key k hashes to slot $h(k)$ and that $h(k)$ is the hash value of key k.

Since usually $m < |U|$, it is easy to see that h is not necessarily injective (collisions)!

Part 4: Notation

Let $T[0..m-1]$ be an array of size m (hash table).

Let U be the "universe" of all potential keys. In practice, we are often interested in subsets $K \subseteq U$ only.

The size m of the hash table is typically much less than $|U|$.

A hash function

$$h\colon U \to \{0, \ldots, m-1\}$$

is a map that assigns to each key $k \in U$ a number $h(k) \in \{0, \ldots, m-1\}$ as a potential slot in the hash table T.

We say that an element with key k hashes to slot $h(k)$ and that $h(k)$ is the hash value of key k.

Since usually $m < |U|$, it is easy to see that h is not necessarily injective (collisions)!

Part 4: Notation

Let $T[0..m-1]$ be an array of size m (hash table).

Let U be the "universe" of all potential keys. In practice, we are often interested in subsets $K \subseteq U$ only.

The size m of the hash table is typically much less than $|U|$.

A hash function

$$h\colon U \to \{0, \ldots, m-1\}$$

is a map that assigns to each key $k \in U$ a number $h(k) \in \{0, \ldots, m-1\}$ as a potential slot in the hash table T.

We say that an element with key k hashes to slot $h(k)$ and that $h(k)$ is the hash value of key k.

Since usually $m < |U|$, it is easy to see that h is not necessarily injective (collisions)!

Part 4: Notation

Let $T[0..m-1]$ be an array of size m (hash table).

Let U be the "universe" of all potential keys. In practice, we are often interested in subsets $K \subseteq U$ only.

The size m of the hash table is typically much less than $|U|$.

A hash function

$$h\colon U \to \{0, \ldots, m-1\}$$

is a map that assigns to each key $k \in U$ a number $h(k) \in \{0, \ldots, m-1\}$ as a potential slot in the hash table T.

We say that an element with key k hashes to slot $h(k)$ and that $h(k)$ is the hash value of key k.

Since usually $m < |U|$, it is easy to see that h is not necessarily injective (collisions)!

Part 4: Notation

Let $T[0..m-1]$ be an array of size m (hash table).

Let U be the "universe" of all potential keys. In practice, we are often interested in subsets $K \subseteq U$ only.

The size m of the hash table is typically much less than $|U|$.

A hash function

$$h\colon U \to \{0, \ldots, m-1\}$$

is a map that assigns to each key $k \in U$ a number $h(k) \in \{0, \ldots, m-1\}$ as a potential slot in the hash table T.

We say that an element with key k hashes to slot $h(k)$ and that $h(k)$ is the hash value of key k.

Since usually $m < |U|$, it is easy to see that h is not necessarily injective (collisions)!

Types and choice of hash functions

There are different ways of defining hash functions. The typical ways are:

- Direct Addressing (perfect hashing)
- “Non-Direct” Addressing
  - Division Method
  - Multiplication method
  - Random Method
  - Selecting hash function from a set of well-designed hash functions

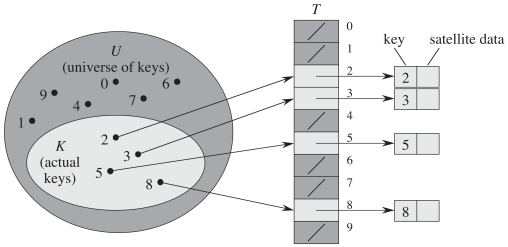
Part 4: Direct Addressing

Direct addressing is a simple technique that works well when the universe U of keys is reasonably small.

Let U be the universe and suppose $|U|$ that is not too large

Exmpl: U = set of all possible membership IDs in sports club in a town of a small number $|U|$ of people. (no two members have the same ID)

Init hash table $T[0..|U|-1]$.



Choose $h\colon U \to \{0, \ldots, |U|-1\}$ as a bijective map (e.g. via total/lexicographic order order on elements in U).

We are interested in $K \subseteq U$ (e.g. all those people in the town that actually are member of sports club.)

Hence, $h\colon K \to \{0, \ldots, |U|-1\}$ is injective, and

$T[h(k)]$ points to the element with key k or is NULL, when this key is not in the set

This allows an $O(1)$ time search (called perfect hashing)

Problems:

- If U gets large or infinite, storing an array T of size $|U|$ may be impractical, or even impossible, given the memory available on a typical computer. (e.g. $U = \{\text{all phone numbers}\}$)
- If actual sets $K \subset U$ of used keys are much smaller than U, then a direct-address table is a big waste of space

Solution: Use hash function $h\colon U \to \{0, \ldots, m-1\}$ where $m \ll |U|$

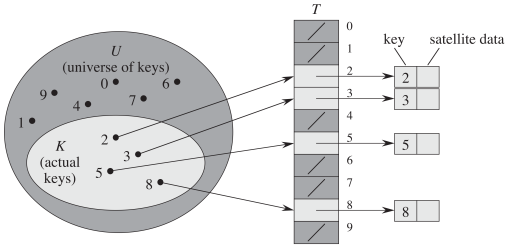
Part 4: Direct Addressing

Direct addressing is a simple technique that works well when the universe U of keys is reasonably small.

Let U be the universe and suppose $|U|$ that is not too large

Exmpl: U = set of all possible membership IDs in sports club in a town of a small number $|U|$ of people. (no two members have the same ID)

Init hash table $T[0..|U|-1]$.



Choose $h\colon U \to \{0,\ldots,|U|-1\}$ as a bijective map (e.g. via total/lexicographic order order on elements in U).

We are interested in $K \subseteq U$ (e.g. all those people in the town that actually are member of sports club.)

Hence, $h\colon K \to \{0,\ldots,|U|-1\}$ is injective, and

$T[h(k)]$ points to the element with key k or is NULL, when this key is not in the set

This allows an $O(1)$ time search (called perfect hashing)

Problems:

If U gets large or infinite, storing an array T of size $|U|$ may be impractical, or even impossible, given the memory available on a typical computer. (e.g. $U = \{\text{all phone numbers}\}$)

If actual sets $K \subset U$ of used keys are much smaller than U, then a direct-address table is a big waste of space

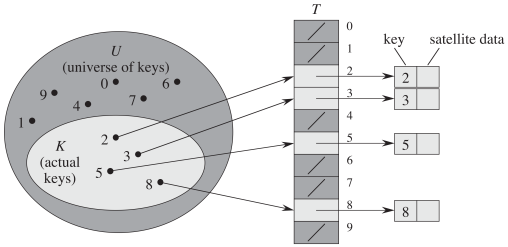
Solution: Use hash function $h\colon U \to \{0,\ldots,m-1\}$ where $m \ll |U|$

Part 4: Direct Addressing

Direct addressing is a simple technique that works well when the universe U of keys is reasonably small.

Let U be the universe and suppose $|U|$ that is not too large

Exmpl: U = set of all possible membership IDs in sports club in a town of a small number $|U|$ of people. (no two members have the same ID)



Init hash table $T[0..|U|-1]$.

Choose $h\colon U \to \{0,\ldots,|U|-1\}$ as a bijective map (e.g. via total/lexicographic order order on elements in U).

We are interested in $K \subseteq U$ (e.g. all those people in the town that actually are member of sports club.)

Hence, $h\colon K \to \{0,\ldots,|U|-1\}$ is injective, and

$T[h(k)]$ points to the element with key k or is NULL, when this key is not in the set

This allows an $O(1)$ time search (called perfect hashing)

Problems:

- If U gets large or infinite, storing an array T of size $|U|$ may be impractical, or even impossible, given the memory available on a typical computer. (e.g. $U = \{\text{all phone numbers}\}$)
- If actual sets $K \subset U$ of used keys are much smaller than U, then a direct-address table is a big waste of space

Solution: Use hash function $h\colon U \to \{0,\ldots,m-1\}$ where $m \ll |U|$

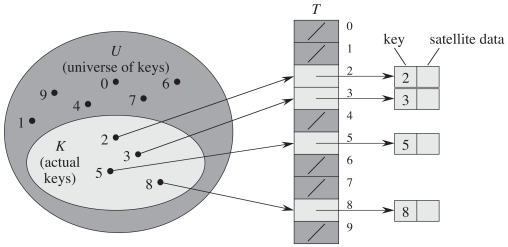
Part 4: Direct Addressing

Direct addressing is a simple technique that works well when the universe U of keys is reasonably small.

Let U be the universe and suppose $|U|$ that is not too large

Exmpl: U = set of all possible membership IDs in sports club in a town of a small number $|U|$ of people. (no two members have the same ID)

Init hash table $T[0..|U|-1]$.



Choose $h\colon U \to \{0, \ldots, |U|-1\}$ as a bijective map (e.g. via total/lexicographic order order on elements in U).

We are interested in $K \subseteq U$ (e.g. all those people in the town that actually are member of sports club.)

Hence, $h\colon K \to \{0, \ldots, |U|-1\}$ is injective, and

$T[h(k)]$ points to the element with key k or is NULL, when this key is not in the set

This allows an $O(1)$ time search (called perfect hashing)

Problems:

- If U gets large or infinite, storing an array T of size $|U|$ may be impractical, or even impossible, given the memory available on a typical computer. (e.g. $U = \{\text{all phone numbers}\}$)
- If actual sets $K \subset U$ of used keys are much smaller than U, then a direct-address table is a big waste of space

Solution: Use hash function $h\colon U \to \{0, \ldots, m-1\}$ where $m \ll |U|$

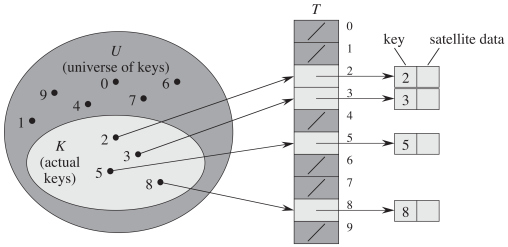
Part 4: Direct Addressing

Direct addressing is a simple technique that works well when the universe U of keys is reasonably small.

Let U be the universe and suppose $|U|$ that is not too large

Exmpl: U = set of all possible membership IDs in sports club in a town of a small number $|U|$ of people. (no two members have the same ID)

Init hash table $T[0..|U|-1]$.



Choose $h\colon U \to \{0,\ldots,|U|-1\}$ as a bijective map (e.g. via total/lexicographic order order on elements in U).

We are interested in $K \subseteq U$ (e.g. all those people in the town that actually are member of sports club.)

Hence, $h\colon K \to \{0,\ldots,|U|-1\}$ is injective, and

$T[h(k)]$ points to the element with key k or is NULL, when this key is not in the set

This allows an $O(1)$ time search (called perfect hashing)

Problems:

- If U gets large or infinite, storing an array T of size $|U|$ may be impractical, or even impossible, given the memory available on a typical computer. (e.g. $U = \{\text{all phone numbers}\}$)
- If actual sets $K \subset U$ of used keys are much smaller than U, then a direct-address table is a big waste of space

Solution: Use hash function $h\colon U \to \{0,\ldots,m-1\}$ where $m \ll |U|$

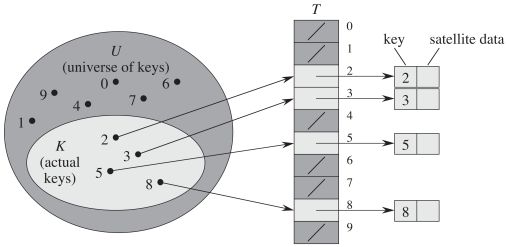
Part 4: Direct Addressing

Direct addressing is a simple technique that works well when the universe U of keys is reasonably small.

Let U be the universe and suppose $|U|$ that is not too large

Exmpl: U = set of all possible membership IDs in sports club in a town of a small number $|U|$ of people. (no two members have the same ID)

Init hash table $T[0..|U|-1]$.



Choose $h\colon U \to \{0, \ldots, |U|-1\}$ as a bijective map (e.g. via total/lexicographic order order on elements in U).

We are interested in $K \subseteq U$ (e.g. all those people in the town that actually are member of sports club.)

Hence, $h\colon K \to \{0, \ldots, |U|-1\}$ is injective, and

$T[h(k)]$ points to the element with key k or is NULL, when this key is not in the set

This allows an $O(1)$ time search (called perfect hashing)

Problems:

- If U gets large or infinite, storing an array T of size $|U|$ may be impractical, or even impossible, given the memory available on a typical computer. (e.g. $U = \{\text{all phone numbers}\}$)
- If actual sets $K \subset U$ of used keys are much smaller than U, then a direct-address table is a big waste of space

Solution: Use hash function $h\colon U \to \{0, \ldots, m-1\}$ where $m \ll |U|$

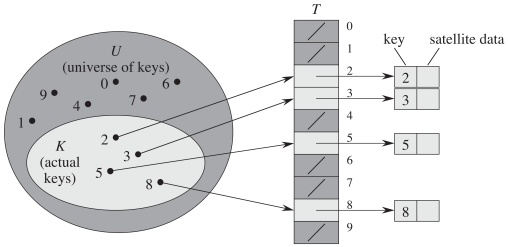
Part 4: Direct Addressing

Direct addressing is a simple technique that works well when the universe U of keys is reasonably small.

Let U be the universe and suppose $|U|$ that is not too large

Exmpl: U = set of all possible membership IDs in sports club in a town of a small number $|U|$ of people. (no two members have the same ID)

Init hash table $T[0..|U|-1]$.



Choose $h\colon U \to \{0, \ldots, |U|-1\}$ as a bijective map (e.g. via total/lexicographic order order on elements in U).

We are interested in $K \subseteq U$ (e.g. all those people in the town that actually are member of sports club.)

Hence, $h\colon K \to \{0, \ldots, |U|-1\}$ is injective, and

$T[h(k)]$ points to the element with key k or is NULL, when this key is not in the set

This allows an $O(1)$ time search (called perfect hashing)

Problems:

- If U gets large or infinite, storing an array T of size $|U|$ may be impractical, or even impossible, given the memory available on a typical computer. (e.g. $U = \{\text{all phone numbers}\}$)
- If actual sets $K \subset U$ of used keys are much smaller than U, then a direct-address table is a big waste of space

Solution: Use hash function $h\colon U \to \{0, \ldots, m-1\}$ where $m \ll |U|$

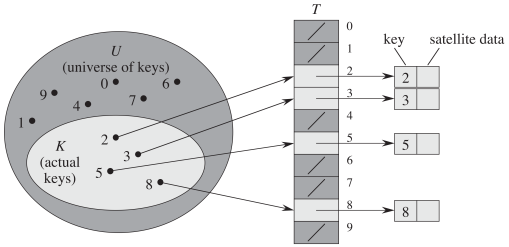
Part 4: Direct Addressing

Direct addressing is a simple technique that works well when the universe U of keys is reasonably small.

Let U be the universe and suppose $|U|$ that is not too large

Exmpl: U = set of all possible membership IDs in sports club in a town of a small number $|U|$ of people. (no two members have the same ID)

Init hash table $T[0..|U|-1]$.



Choose $h\colon U \to \{0, \ldots, |U|-1\}$ as a bijective map (e.g. via total/lexicographic order order on elements in U).

We are interested in $K \subseteq U$ (e.g. all those people in the town that actually are member of sports club.)

Hence, $h\colon K \to \{0, \ldots, |U|-1\}$ is injective, and

$T[h(k)]$ points to the element with key k or is NULL, when this key is not in the set

This allows an $O(1)$ time search (called perfect hashing)

Problems:

- If U gets large or infinite, storing an array T of size $|U|$ may be impractical, or even impossible, given the memory available on a typical computer. (e.g. $U = \{\text{all phone numbers}\}$)
- If actual sets $K \subset U$ of used keys are much smaller than U, then a direct-address table is a big waste of space

Solution: Use hash function $h\colon U \to \{0, \ldots, m-1\}$ where $m \ll |U|$

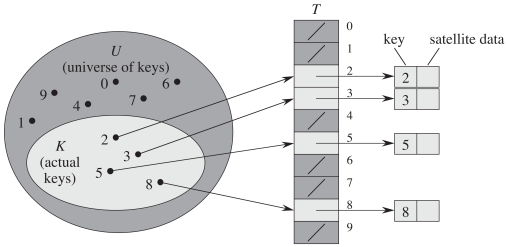
Part 4: Direct Addressing

Direct addressing is a simple technique that works well when the universe U of keys is reasonably small.

Let U be the universe and suppose $|U|$ that is not too large

Exmpl: U = set of all possible membership IDs in sports club in a town of a small number $|U|$ of people. (no two members have the same ID)

Init hash table $T[0..|U|-1]$.



Choose $h\colon U \to \{0, \ldots, |U|-1\}$ as a bijective map (e.g. via total/lexicographic order order on elements in U).

We are interested in $K \subseteq U$ (e.g. all those people in the town that actually are member of sports club.)

Hence, $h\colon K \to \{0, \ldots, |U|-1\}$ is injective, and

$T[h(k)]$ points to the element with key k or is NULL, when this key is not in the set

This allows an $O(1)$ time search (called perfect hashing)

Problems:

If U gets large or infinite, storing an array T of size $|U|$ may be impractical, or even impossible, given the memory available on a typical computer. (e.g. $U = \{\text{all phone numbers}\}$)

If actual sets $K \subset U$ of used keys are much smaller than U, then a direct-address table is a big waste of space

Solution: Use hash function $h\colon U \to \{0, \ldots, m-1\}$ where $m \ll |U|$

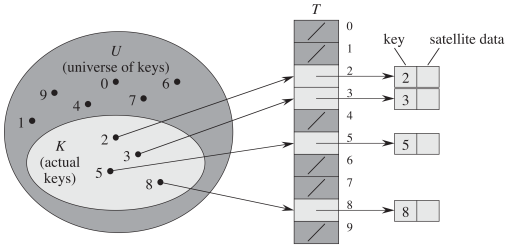
Part 4: Direct Addressing

Direct addressing is a simple technique that works well when the universe U of keys is reasonably small.

Let U be the universe and suppose $|U|$ that is not too large

Exmpl: U = set of all possible membership IDs in sports club in a town of a small number $|U|$ of people. (no two members have the same ID)

Init hash table $T[0..|U|-1]$.



Choose $h\colon U \to \{0, \ldots, |U|-1\}$ as a bijective map (e.g. via total/lexicographic order order on elements in U).

We are interested in $K \subseteq U$ (e.g. all those people in the town that actually are member of sports club.)

Hence, $h\colon K \to \{0, \ldots, |U|-1\}$ is injective, and

$T[h(k)]$ points to the element with key k or is NULL, when this key is not in the set

This allows an $O(1)$ time search (called perfect hashing)

Problems:

If U gets large or infinite, storing an array T of size $|U|$ may be impractical, or even impossible, given the memory available on a typical computer. (e.g. $U = \{\text{all phone numbers}\}$)

If actual sets $K \subset U$ of used keys are much smaller than U, then a direct-address table is a big waste of space

Solution: Use hash function $h\colon U \to \{0, \ldots, m-1\}$ where $m \ll |U|$

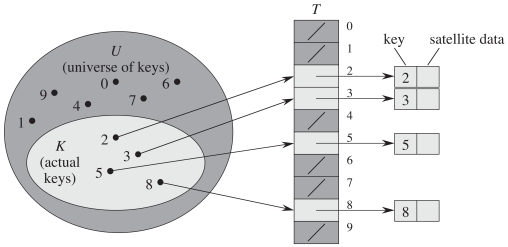
Part 4: Direct Addressing

Direct addressing is a simple technique that works well when the universe U of keys is reasonably small.

Let U be the universe and suppose $|U|$ that is not too large

Exmpl: U = set of all possible membership IDs in sports club in a town of a small number $|U|$ of people. (no two members have the same ID)

Init hash table $T[0..|U|-1]$.



Choose $h\colon U \to \{0, \ldots, |U|-1\}$ as a bijective map (e.g. via total/lexicographic order order on elements in U).

We are interested in $K \subseteq U$ (e.g. all those people in the town that actually are member of sports club.)

Hence, $h\colon K \to \{0, \ldots, |U|-1\}$ is injective, and

$T[h(k)]$ points to the element with key k or is NULL, when this key is not in the set

This allows an $O(1)$ time search (called perfect hashing)

Problems:

If U gets large or infinite, storing an array T of size $|U|$ may be impractical, or even impossible, given the memory available on a typical computer. (e.g. $U = \{\text{all phone numbers}\}$)

If actual sets $K \subset U$ of used keys are much smaller than U, then a direct-address table is a big waste of space

Solution: Use hash function $h\colon U \to \{0, \ldots, m-1\}$ where $m \ll |U|$

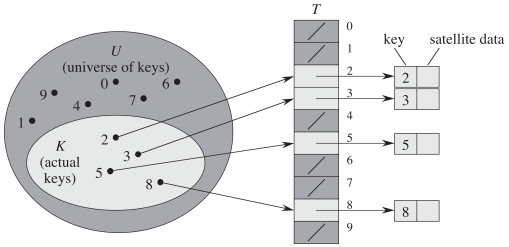
Part 4: Direct Addressing

Direct addressing is a simple technique that works well when the universe U of keys is reasonably small.

Let U be the universe and suppose $|U|$ that is not too large

Exmpl: U = set of all possible membership IDs in sports club in a town of a small number $|U|$ of people. (no two members have the same ID)

Init hash table $T[0..|U|-1]$.



Choose $h\colon U \to \{0,\ldots,|U|-1\}$ as a bijective map (e.g. via total/lexicographic order order on elements in U).

We are interested in $K \subseteq U$ (e.g. all those people in the town that actually are member of sports club.)

Hence, $h\colon K \to \{0,\ldots,|U|-1\}$ is injective, and

$T[h(k)]$ points to the element with key k or is NULL, when this key is not in the set

This allows an $O(1)$ time search (called perfect hashing)

Problems:

If U gets large or infinite, storing an array T of size $|U|$ may be impractical, or even impossible, given the memory available on a typical computer. (e.g. $U = \{\text{all phone numbers}\}$)

If actual sets $K \subset U$ of used keys are much smaller than U, then a direct-address table is a big waste of space

Solution: Use hash function $h\colon U \to \{0,\ldots,m-1\}$ where $m \ll |U|$

Part 4: Choice of hash function

If direct addressing is not feasible, use a hash function $h\colon U \to \{0, \ldots, m-1\}$ where $m \ll |U|$

A good hash function should be as easy and fast to compute as possible and evenly distribute the data records to be stored across the hash table to avoid collisions.

In other words, a good hash function satisies (approximately) the assumption of simple uniform hashing:

Each key is equally likely to hash to any of the m slots, independently of where any other key has hashed to

Unfortunately, you typically have no way to check this condition, unless you happen to know the probability distribution from which the keys are drawn. Moreover, the keys might not be drawn independently (e.g. $U = \{\text{all phone numbers}\}$ and two phone numbers $k_1 \neq k_2$ often share the same prefix, the area code).

Choose m in the order of the number of elements expected to be stored ["good" m depend on h]

Part 4: Choice of hash function

If direct addressing is not feasible, use a hash function $h\colon U \to \{0, \ldots, m-1\}$ where $m \ll |U|$

A good hash function should be as easy and fast to compute as possible and evenly distribute the data records to be stored across the hash table to avoid collisions.

In other words, a good hash function satisies (approximately) the assumption of simple uniform hashing:

Each key is equally likely to hash to any of the m slots, independently of where any other key has hashed to

Unfortunately, you typically have no way to check this condition, unless you happen to know the probability distribution from which the keys are drawn. Moreover, the keys might not be drawn independently (e.g. $U = \{\text{all phone numbers}\}$ and two phone numbers $k_1 \neq k_2$ often share the same prefix, the area code).

Choose m in the order of the number of elements expected to be stored ["good" m depend on h]

Part 4: Choice of hash function

If direct addressing is not feasible, use a hash function $h\colon U \to \{0, \ldots, m-1\}$ where $m \ll |U|$

A good hash function should be as easy and fast to compute as possible and evenly distribute the data records to be stored across the hash table to avoid collisions.

In other words, a good hash function satisies (approximately) the assumption of simple uniform hashing:

Each key is equally likely to hash to any of the m slots, independently of where any other key has hashed to

Unfortunately, you typically have no way to check this condition, unless you happen to know the probability distribution from which the keys are drawn. Moreover, the keys might not be drawn independently (e.g. $U = \{\text{all phone numbers}\}$ and two phone numbers $k_1 \neq k_2$ often share the same prefix, the area code).

Choose m in the order of the number of elements expected to be stored ["good" m depend on h]

Part 4: Choice of hash function

If direct addressing is not feasible, use a hash function $h\colon U \to \{0, \ldots, m-1\}$ where $m \ll |U|$

A good hash function should be as easy and fast to compute as possible and evenly distribute the data records to be stored across the hash table to avoid collisions.

In other words, a good hash function satisies (approximately) the assumption of simple uniform hashing:

Each key is equally likely to hash to any of the m slots, independently of where any other key has hashed to

Unfortunately, you typically have no way to check this condition, unless you happen to know the probability distribution from which the keys are drawn. Moreover, the keys might not be drawn independently (e.g. $U = \{\text{all phone numbers}\}$ and two phone numbers $k_1 \neq k_2$ often share the same prefix, the area code).

Choose m in the order of the number of elements expected to be stored ["good" m depend on h]

Part 4: Choice of hash function

A natural way is the **division method** where a hash function is maps a key k into one of m slots by taking the remainder of k divided by m. That is, the hash function is

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

Example: If $m = 12$ and key is $k = 100$, then $h(k) = 4$.

Since it requires only a single division operation, hashing by division is quite fast.

Problems one needs to be aware of:

If $m = 2^p$, then $h(k)$ depends only on the last p bits of k.

Part 4: Choice of hash function

A natural way is the **division method** where a hash function is maps a key k into one of m slots by taking the remainder of k divided by m. That is, the hash function is

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

Example: If $m = 12$ and key is $k = 100$, then $h(k) = 4$.

Since it requires only a single division operation, hashing by division is quite fast.

Problems one needs to be aware of:

If $m = 2^p$, then $h(k)$ depends only on the last p bits of k.

Part 4: Choice of hash function

A natural way is the **division method** where a hash function is maps a key k into one of m slots by taking the remainder of k divided by m. That is, the hash function is

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

Example: If $m = 12$ and key is $k = 100$, then $h(k) = 4$.

Since it requires only a single division operation, hashing by division is quite fast.

Problems one needs to be aware of:

If $m = 2^p$, then $h(k)$ depends only on the last p bits of k.

Part 4: Choice of hash function

A natural way is the **division method** where a hash function is maps a key k into one of m slots by taking the remainder of k divided by m. That is, the hash function is

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

Example: If $m = 12$ and key is $k = 100$, then $h(k) = 4$.

Since it requires only a single division operation, hashing by division is quite fast.

Problems one needs to be aware of:

If $m = 2^p$, then $h(k)$ depends only on the last p bits of k.

Part 4: Choice of hash function

A natural way is the **division method** where a hash function is maps a key k into one of m slots by taking the remainder of k divided by m. That is, the hash function is

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

Example: If $m = 12$ and key is $k = 100$, then $h(k) = 4$.

Since it requires only a single division operation, hashing by division is quite fast.

Problems one needs to be aware of:

If $m = 2^p$, then $h(k)$ depends only on the last p bits of k.

Part 4: Choice of hash function

A natural way is the **division method** where a hash function is maps a key k into one of m slots by taking the remainder of k divided by m. That is, the hash function is

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

Example: If $m = 12$ and key is $k = 100$, then $h(k) = 4$.

Since it requires only a single division operation, hashing by division is quite fast.

Problems one needs to be aware of:

If $m = 2^p$, then $h(k)$ depends only on the last p bits of k.

Example: $m = 2^2 = 4$, then last 2 bits of a number in binary representation are always one of $00, 01, 10, 11$ and we have

$k \bmod 4 = 0$ iff last 2 bits are 00
$k \bmod 4 = 1$ iff last 2 bits are 01
$k \bmod 4 = 2$ iff last 2 bits are 10
$k \bmod 4 = 3$ iff last 2 bits are 11

Part 4: Choice of hash function

A natural way is the **division method** where a hash function is maps a key k into one of m slots by taking the remainder of k divided by m. That is, the hash function is

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

Example: If $m = 12$ and key is $k = 100$, then $h(k) = 4$.

Since it requires only a single division operation, hashing by division is quite fast.

Problems one needs to be aware of:

If $m = 2^p$, then $h(k)$ depends only on the last p bits of k.
Hence, all keys that agree in the last p bits hash to the same slot.

Example: $m = 2^2 = 4$, then last 2 bits of a number in binary representation are always one of $00, 01, 10, 11$ and we have

$k \bmod 4 = 0$ iff last 2 bits are 00
$k \bmod 4 = 1$ iff last 2 bits are 01
$k \bmod 4 = 2$ iff last 2 bits are 10
$k \bmod 4 = 3$ iff last 2 bits are 11

Part 4: Choice of hash function

A natural way is the **division method** where a hash function is maps a key k into one of m slots by taking the remainder of k divided by m. That is, the hash function is

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

Example: If $m = 12$ and key is $k = 100$, then $h(k) = 4$.

Since it requires only a single division operation, hashing by division is quite fast.

Problems one needs to be aware of:

If $m = 2^p$, then $h(k)$ depends only on the last p bits of k.

Hence, all keys that agree in the last p bits hash to the same slot.

Depending on distribution of keys, performance of such a hash table may be bad.

Example: $m = 2^2 = 4$, then last 2 bits of a number in binary representation are always one of $00, 01, 10, 11$ and we have

$k \bmod 4 = 0$ iff last 2 bits are 00
$k \bmod 4 = 1$ iff last 2 bits are 01
$k \bmod 4 = 2$ iff last 2 bits are 10
$k \bmod 4 = 3$ iff last 2 bits are 11

Part 4: Choice of hash function

A natural way is the **division method** where a hash function is maps a key k into one of m slots by taking the remainder of k divided by m. That is, the hash function is

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

Example: If $m = 12$ and key is $k = 100$, then $h(k) = 4$.

Since it requires only a single division operation, hashing by division is quite fast.

Problems one needs to be aware of:

If $m = 2^p$, then $h(k)$ depends only on the last p bits of k.

Hence, all keys that agree in the last p bits hash to the same slot.

Depending on distribution of keys, performance of such a hash table may be bad.

If $m = 2^p - 1$, and we are hashing strings that are interpreted as base 2^p numbers
Then a string $s = s_r \cdots s_1 s_0$ in which each character s_i is interpreted as a number in $\{0, \ldots, 2^p\}$ hashes to

$$\begin{aligned} h(s) &= \left(\sum_{i=0}^{r} s_i \cdot (2^p)^i \right) \bmod (2^p - 1) \qquad //(2^p)^i \bmod (2^p - 1) = 1 \text{ + mod-rules} \\ &= \left(\sum_{i=0}^{r} s_i \right) \bmod (2^p - 1) \end{aligned}$$

Part 4: Choice of hash function

A natural way is the **division method** where a hash function is maps a key k into one of m slots by taking the remainder of k divided by m. That is, the hash function is

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

Example: If $m = 12$ and key is $k = 100$, then $h(k) = 4$.

Since it requires only a single division operation, hashing by division is quite fast.

Problems one needs to be aware of:

If $m = 2^p$, then $h(k)$ depends only on the last p bits of k.

Hence, all keys that agree in the last p bits hash to the same slot.

Depending on distribution of keys, performance of such a hash table may be bad.

If $m = 2^p - 1$, and we are hashing strings that are interpreted as base 2^p numbers
Then a string $s = s_r \cdots s_1 s_0$ in which each character s_i is interpreted as a number in $\{0, \ldots, 2^p\}$ hashes to

$$\begin{aligned} h(s) &= \left(\sum_{i=0}^{r} s_i \cdot (2^p)^i \right) \bmod (2^p - 1) \qquad //(2^p)^i \bmod (2^p - 1) = 1 \text{ + mod-rules} \\ &= \left(\sum_{i=0}^{r} s_i \right) \bmod (2^p - 1) \end{aligned}$$

Thus, the hash value of a string is invariant against permutations of the string.

Example: $s =$`Mia` via ASCII: `M` $= 77$, `i` $= 105$, `a` $= 97$
$\implies s \hat{=} 77 + 105 \cdot 128 + 97 \cdot 128^2 = 1602765$ and $s' =$`iMa` $\hat{=} 105 + 77 \cdot 128 + 97 \cdot 128^2 = 1599209$
$\implies h(s) = 1602765 \bmod 127 = 25 = h(s') = 1599209 \bmod 127$

Part 4: Choice of hash function

A natural way is the **division method** where a hash function is maps a key k into one of m slots by taking the remainder of k divided by m. That is, the hash function is

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

Example: If $m = 12$ and key is $k = 100$, then $h(k) = 4$.

Since it requires only a single division operation, hashing by division is quite fast.

Problems one needs to be aware of:

If $m = 2^p$, then $h(k)$ depends only on the last p bits of k.

Hence, all keys that agree in the last p bits hash to the same slot.

Depending on distribution of keys, performance of such a hash table may be bad.

If $m = 2^p - 1$, and we are hashing strings that are interpreted as base 2^p numbers
Then a string $s = s_r \cdots s_1 s_0$ in which each character s_i is interpreted as a number in $\{0, \ldots, 2^p\}$ hashes to

$$\begin{aligned} h(s) &= \left(\sum_{i=0}^{r} s_i \cdot (2^p)^i \right) \bmod (2^p - 1) \qquad //(2^p)^i \bmod (2^p - 1) = 1 + \text{mod-rules} \\ &= \left(\sum_{i=0}^{r} s_i \right) \bmod (2^p - 1) \end{aligned}$$

Solutions to the latter problems are provided by carefully choosing m.
Usually m is a particular well-chosen prime number = number theory (not part of this course)

Part 4: Choice of hash function

Further ways:

multiplication method where a hash function is created that maps a key k into one of m slots by using

$$h(k) = \lfloor m(k\phi \bmod 1) \rfloor,$$

where $\phi \in (0, 1)$ is an (irrational) number and "$k\phi \bmod 1$" means the fractional part of $k\phi$, that is, $k\phi - \lfloor k\phi \rfloor$

random method, e.g., by using key k as the seed for a random number generator producing a number between $0, 1, \ldots, m - 1$

Selecting $h \in \mathcal{H}$: randomly pick h from a precomputed and carefully designed set $\mathcal{H}$ of hashfunction

The behavior/performance of a hash function depends on the chosen set of keys.

Therefore, they can only be insufficiently studied theoretically or with the help of analytical models.

Given a hash function, it is always possible to find a set of keys for which it generates many collisions.

No hash function is always better than all others.

However, there are several empirical studies on the quality of different hash functions.

The division method is generally the most efficient; however, for certain sets of keys, other techniques may perform better.

If the key distribution is unknown, then the division method is the preferred hashing technique.

Important: Use sufficiently large hash table and use a prime number as divisor.

Moreover, hashing is, in practice not based on a fixed hash function, but on carefully designed sets $\mathcal{H}$ of hash functions from which we randomly pick one.

Part 4: Choice of hash function

Further ways:

multiplication method where a hash function is created that maps a key k into one of m slots by using

$$h(k) = \lfloor m(k\phi \bmod 1) \rfloor,$$

where $\phi \in (0, 1)$ is an (irrational) number and "$k\phi \bmod 1$" means the fractional part of $k\phi$, that is, $k\phi - \lfloor k\phi \rfloor$

random method, e.g., by using key k as the seed for a random number generator producing a number between $0, 1, \ldots, m - 1$

Selecting $h \in \mathcal{H}$: randomly pick h from a precomputed and carefully designed set $\mathcal{H}$ of hashfunction

The behavior/performance of a hash function depends on the chosen set of keys.

Therefore, they can only be insufficiently studied theoretically or with the help of analytical models.

Given a hash function, it is always possible to find a set of keys for which it generates many collisions.

No hash function is always better than all others.

However, there are several empirical studies on the quality of different hash functions.

The division method is generally the most efficient; however, for certain sets of keys, other techniques may perform better.

If the key distribution is unknown, then the division method is the preferred hashing technique.

Important: Use sufficiently large hash table and use a prime number as divisor.

Moreover, hashing is, in practice not based on a fixed hash function, but on carefully designed sets $\mathcal{H}$ of hash functions from which we randomly pick one.

Part 4: Choice of hash function

Further ways:

multiplication method where a hash function is created that maps a key k into one of m slots by using

$$h(k) = \lfloor m(k\phi \bmod 1) \rfloor,$$

where $\phi \in (0, 1)$ is an (irrational) number and "$k\phi \bmod 1$" means the fractional part of $k\phi$, that is, $k\phi - \lfloor k\phi \rfloor$

random method, e.g., by using key k as the seed for a random number generator producing a number between $0, 1, \ldots, m - 1$

Selecting $h \in \mathcal{H}$: randomly pick h from a precomputed and carefully designed set $\mathcal{H}$ of hashfunction

The behavior/performance of a hash function depends on the chosen set of keys.

Therefore, they can only be insufficiently studied theoretically or with the help of analytical models.

Given a hash function, it is always possible to find a set of keys for which it generates many collisions.

No hash function is always better than all others.

However, there are several empirical studies on the quality of different hash functions.

The division method is generally the most efficient; however, for certain sets of keys, other techniques may perform better.

If the key distribution is unknown, then the division method is the preferred hashing technique.

Important: Use sufficiently large hash table and use a prime number as divisor.

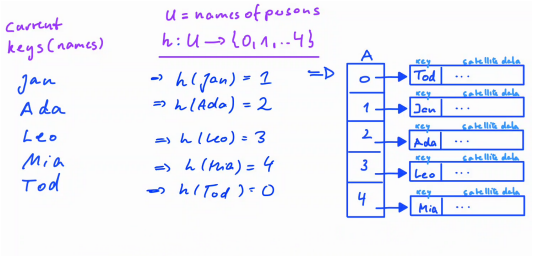
Moreover, hashing is, in practice not based on a fixed hash function, but on carefully designed sets $\mathcal{H}$ of hash functions from which we randomly pick one.

Part 4: Collisions

If direct addressing is not feasible, use a hash function $h\colon U \to \{0, \ldots, m-1\}$ where $m \ll |U|$

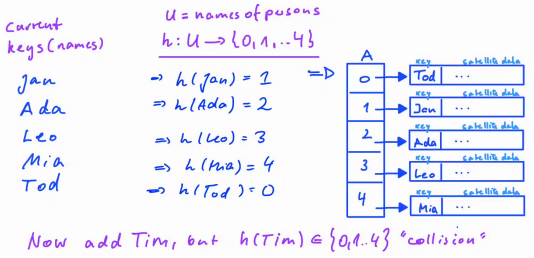
Part 4: Collisions

If direct addressing is not feasible, use a hash function $h\colon U \to \{0, \ldots, m-1\}$ where $m \ll |U|$



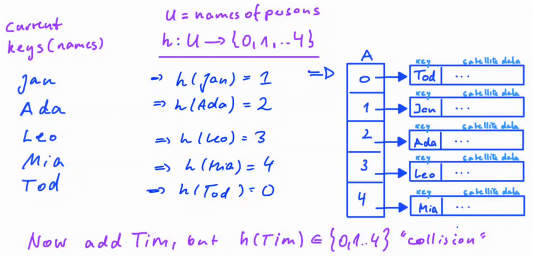
Part 4: Collisions

If direct addressing is not feasible, use a hash function $h\colon U \to \{0, \ldots, m-1\}$ where $m \ll |U|$

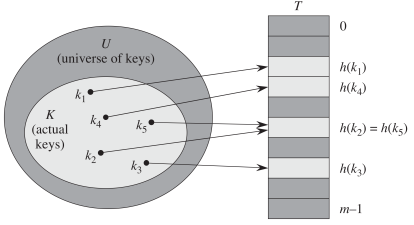


Part 4: Collisions

If direct addressing is not feasible, use a hash function $h\colon U \to \{0, \ldots, m-1\}$ where $m \ll |U|$



Problem: different keys may receive the same h value (collision).



Part 4: Collisions

If direct addressing is not feasible, use a hash function $h\colon U \to \{0, \ldots, m-1\}$ where $m \ll |U|$

Problem: different keys may receive the same h value (collision)

Part 4: Collisions

If direct addressing is not feasible, use a hash function $h\colon U \to \{0, \ldots, m-1\}$ where $m \ll |U|$

Problem: different keys may receive the same h value (collision)

Question: How likely is it that two keys receive the same h value assuming that $h(k)$ is randomly chosen for all keys k?

Part 4: Collisions

If direct addressing is not feasible, use a hash function $h\colon U \to \{0, \ldots, m-1\}$ where $m \ll |U|$

Problem: different keys may receive the same h value (collision)

Question: How likely is it that two keys receive the same h value assuming that $h(k)$ is randomly chosen for all keys k?

A problem that is similar in fashion:

Birthday problem: what is the probability that, in a set of ℓ randomly chosen people, at least two will share a birthday.

Part 4: Collisions

If direct addressing is not feasible, use a hash function $h\colon U \to \{0, \ldots, m-1\}$ where $m \ll |U|$

Problem: different keys may receive the same h value (collision)

Question: How likely is it that two keys receive the same h value assuming that $h(k)$ is randomly chosen for all keys k?

A problem that is similar in fashion:

Birthday problem: what is the probability that, in a set of ℓ randomly chosen people, at least two will share a birthday.

Here $m = 365$ and, say $\ell = 23$ persons (keys) – What is your gut feeling ?

Part 4: Collisions

If direct addressing is not feasible, use a hash function $h\colon U \to \{0, \ldots, m-1\}$ where $m \ll |U|$

Problem: different keys may receive the same h value (collision)

Question: How likely is it that two keys receive the same h value assuming that $h(k)$ is randomly chosen for all keys k?

A problem that is similar in fashion:

Birthday problem: what is the probability that, in a set of ℓ randomly chosen people, at least two will share a birthday.

Here $m = 365$ and, say $\ell = 23$ persons (keys) – What is your gut feeling ?

There are $\prod_{i=1}^{23} 365 = 365^{23}$ variations of all-birthday-person-cases (all cases equality likely).

Part 4: Collisions

If direct addressing is not feasible, use a hash function $h\colon U \to \{0, \ldots, m-1\}$ where $m \ll |U|$

Problem: different keys may receive the same h value (collision)

Question: How likely is it that two keys receive the same h value assuming that $h(k)$ is randomly chosen for all keys k?

A problem that is similar in fashion:

Birthday problem: what is the probability that, in a set of ℓ randomly chosen people, at least two will share a birthday.

Here $m = 365$ and, say $\ell = 23$ persons (keys) – What is your gut feeling ?

There are $\prod_{i=1}^{23} 365 = 365^{23}$ variations of all-birthday-person-cases (all cases equality likely).

Let us consider the nr c of cases that all persons have a birthday on different days
(1st person 365 possibilities, 2nd person remaining 364 days, ...):

$$c = 365 \cdot 364 \cdot 363 \cdot \cdots \cdot 343$$

Part 4: Collisions

If direct addressing is not feasible, use a hash function $h\colon U \to \{0, \ldots, m-1\}$ where $m \ll |U|$

Problem: different keys may receive the same h value (collision)

Question: How likely is it that two keys receive the same h value assuming that $h(k)$ is randomly chosen for all keys k?

A problem that is similar in fashion:

Birthday problem: what is the probability that, in a set of ℓ randomly chosen people, at least two will share a birthday.

Here $m = 365$ and, say $\ell = 23$ persons (keys) – What is your gut feeling ?

There are $\prod_{i=1}^{23} 365 = 365^{23}$ variations of all-birthday-person-cases (all cases equality likely).

Let us consider the nr c of cases that all persons have a birthday on different days
(1st person 365 possibilities, 2nd person remaining 364 days, ...):

$$c = 365 \cdot 364 \cdot 363 \cdot \cdots \cdot 343$$

Probability p that all 23 persons have a birthday on different days

$$p = \frac{365 \cdot 364 \cdot 363 \cdot \cdots \cdot 343}{365^{23}}$$

Part 4: Collisions

If direct addressing is not feasible, use a hash function $h\colon U \to \{0, \ldots, m-1\}$ where $m \ll |U|$

Problem: different keys may receive the same h value (collision)

Question: How likely is it that two keys receive the same h value assuming that $h(k)$ is randomly chosen for all keys k?

A problem that is similar in fashion:

Birthday problem: what is the probability that, in a set of ℓ randomly chosen people, at least two will share a birthday.

Here $m = 365$ and, say $\ell = 23$ persons (keys) – What is your gut feeling ?

There are $\prod_{i=1}^{23} 365 = 365^{23}$ variations of all-birthday-person-cases (all cases equality likely).

Let us consider the nr c of cases that all persons have a birthday on different days
(1st person 365 possibilities, 2nd person remaining 364 days, ...):

$$c = 365 \cdot 364 \cdot 363 \cdot \cdots \cdot 343$$

Probability p that all 23 persons have a birthday on different days

$$p = \frac{365 \cdot 364 \cdot 363 \cdot \cdots \cdot 343}{365^{23}}$$

Probability p' that there are at least two persons having the same birthday

$$p' = 1 - \frac{365 \cdot 364 \cdot 363 \cdot \cdots \cdot 343}{365^{23}} \approx 0.51$$

Part 4: Collisions

If direct addressing is not feasible, use a hash function $h\colon U \to \{0, \ldots, m-1\}$ where $m \ll |U|$

Problem: different keys may receive the same h value (collision)

Question: How likely is it that two keys receive the same h value assuming that $h(k)$ is randomly chosen for all keys k?

A problem that is similar in fashion:

Birthday problem: what is the probability that, in a set of ℓ randomly chosen people, at least two will share a birthday.

Here $m = 365$ and, say $\ell = 23$ persons (keys) – What is your gut feeling ?

There are $\prod_{i=1}^{23} 365 = 365^{23}$ variations of all-birthday-person-cases (all cases equality likely).

Let us consider the nr c of cases that all persons have a birthday on different days
(1st person 365 possibilities, 2nd person remaining 364 days, ...):

$$c = 365 \cdot 364 \cdot 363 \cdot \cdots \cdot 343$$

Probability p that all 23 persons have a birthday on different days

$$p = \frac{365 \cdot 364 \cdot 363 \cdot \cdots \cdot 343}{365^{23}}$$

Probability p' that there are at least two persons having the same birthday

$$p' = 1 - \frac{365 \cdot 364 \cdot 363 \cdot \cdots \cdot 343}{365^{23}} \approx 0.51$$

With a 51% chance two elements collide when $\ell = 23 \ll m = 365$

Part 4: Collisions

If direct addressing is not feasible, use a hash function $h\colon U \to \{0, \ldots, m-1\}$ where $m \ll |U|$

Problem: different keys may receive the same h value (collision)

Question: How likely is it that two keys receive the same h value assuming that $h(k)$ is randomly chosen for all keys k?

A problem that is similar in fashion:

Birthday problem: what is the probability that, in a set of ℓ randomly chosen people, at least two will share a birthday.

Here $m = 365$ and, say $\ell = 23$ persons (keys) – What is your gut feeling ?

There are $\prod_{i=1}^{23} 365 = 365^{23}$ variations of all-birthday-person-cases (all cases equality likely).

Let us consider the nr c of cases that all persons have a birthday on different days
(1st person 365 possibilities, 2nd person remaining 364 days, ...):

$$c = 365 \cdot 364 \cdot 363 \cdot \cdots \cdot 343$$

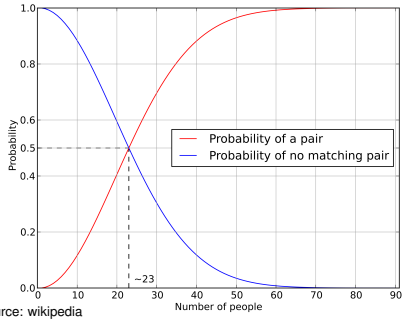
Probability p that all 23 persons have a birthday on different days

$$p = \frac{365 \cdot 364 \cdot 363 \cdot \cdots \cdot 343}{365^{23}}$$

Probability p' that there are at least two persons having the same birthday

$$p' = 1 - \frac{365 \cdot 364 \cdot 363 \cdot \cdots \cdot 343}{365^{23}} \approx 0.51$$

With a 51% chance two elements collide when $\ell = 23 \ll m = 365$



source: wikipedia
(red-line: at least 2 have same birthday, blue-line: all different birthdays)

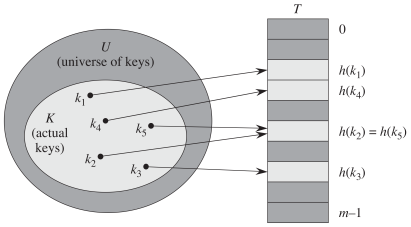
Even for $\ell = 23$ keys and $m = 365$ slots the chances are 51% to have collisions $\Longrightarrow$ collisions MUST BE addressed!

Part 4: Collisions

A hash function h maps the universe U of keys into the slots of a hash table $T[0..m-1]$

$$h\colon U \to \{0, \ldots, m-1\}$$

where the size m of the hash table is typically much less than $|U|$.



A collision occurs when two keys hash to the same slot.

Question:

How to deal with and to resolve collisions?

There are two main approaches:

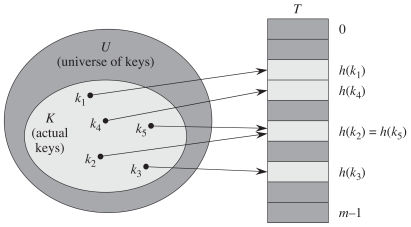
- Resolving collision via chaining
- Resolving collision via open addressing

Part 4: Collisions

A hash function h maps the universe U of keys into the slots of a hash table $T[0..m-1]$

$$h\colon U \to \{0, \ldots, m-1\}$$

where the size m of the hash table is typically much less than $|U|$.



A collision occurs when two keys hash to the same slot.

Question:

How to deal with and to resolve collisions?

There are two main approaches:

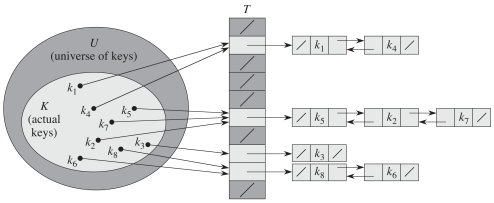
Resolving collision via chaining

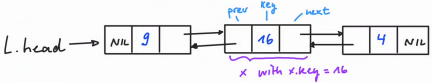
Resolving collision via open addressing

Part 4: Resolving collision via chaining

Chaining:

- $T[j] := NIL$ if no element in the set has a key k with Hashing $h(k) = j$
- Otherwise, store at $T[j]$ the pointer to the head of a doubly-linked list of all such elements



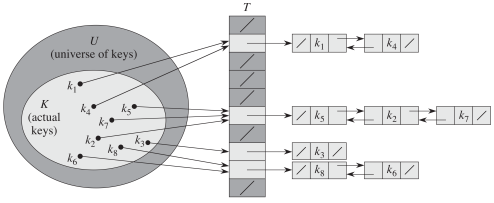


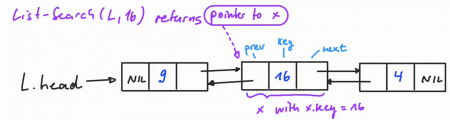
basic operations on doubly-linked lists L and hash-tables T:

Part 4: Resolving collision via chaining

Chaining:

- $T[j] \coloneqq NIL$ if no element in the set has a key k with Hashing $h(k) = j$
- Otherwise, store at $T[j]$ the pointer to the head of a doubly-linked list of all such elements





basic operations on doubly-linked lists L and hash-tables T:

`List-Search(`L`,`k`)`//finds the first element with key k in list L by a simple linear search, returning a pointer to this element

```
1  x := L.head //x is pointer to first element in L
2  WHILE x ≠ NIL and x.key ≠ k DO
3      x := x.next
4  return x
```

`Chained-Hash-Search(`T`,`k`)`

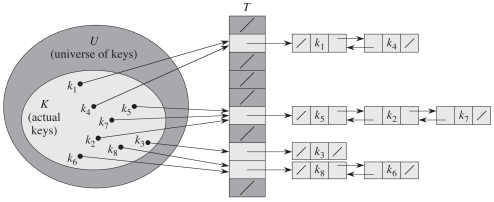
```
1  return List-Search(T([h(k)]),k)
```

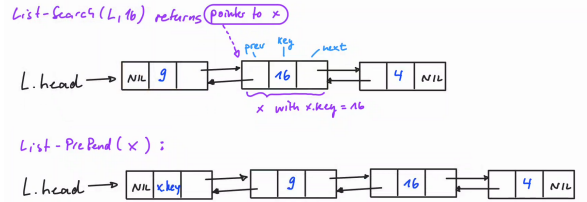
//Takes time proportional to number of elements in list $T[h(k)]$.

Part 4: Resolving collision via chaining

Chaining:

- $T[j] \coloneqq NIL$ if no element in the set has a key k with Hashing $h(k) = j$
- Otherwise, store at $T[j]$ the pointer to the head of a doubly-linked list of all such elements





basic operations on doubly-linked lists L and hash-tables T:

`List-PrePend(L,x)` //inserts object/element x to the front of L

1 $x.next \coloneqq L.head$ //x is pointer to first element in L
2 $x.prev \coloneqq NIL$
3 `IF` $L.head \neq NIL$ `THEN` $L.head.prev \coloneqq x$
4 $L.head \coloneqq x$

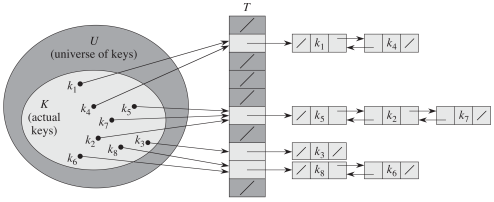
`Chained-Hash-Insert(T,x)`

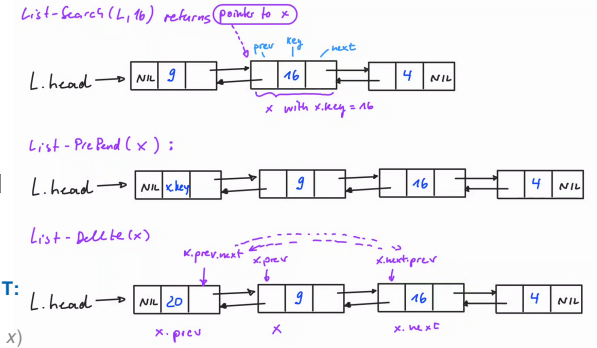
1 `List-PrePend(`$T[h(x.key)]$`,x)` //Takes constant time if we assume that x is not already in the hash table
// (if not known, do a search first).

Part 4: Resolving collision via chaining

Chaining:

- $T[j] \coloneqq NIL$ if no element in the set has a key k with Hashing $h(k) = j$
- Otherwise, store at $T[j]$ the pointer to the head of a doubly-linked list of all such elements





basic operations on doubly-linked lists L and hash-tables T:

`List-Delete(L,x)`

//deletes object/element x from L (It must be given a pointer to x)

1 `IF` $x.prev \neq NIL$ `THEN` $x.prev.next \coloneqq x.next$
2 `ELSE` $L.head \coloneqq x.next$
2 `IF` $x.next \neq NIL$ `THEN` $x.next.prev \coloneqq x.prev$

To delete an element with a given key k, fist call `List-Search(L,k)` to retrieve a pointer to the element

`Chained-Hash-Delete(T,x)`

1 `List-Delete(T[h(x.key)],x)`

//Takes constant time if we have pointer to x
// (if pointer not known, do a search first).

Part 4: Resolving collision via chaining

How well does hashing with chaining perform? In particular, how long does it take to search for an element with a given key?

Given: A hash table T with m slots that stores n elements

Worst case: h maps all n element to same slot (that is, the list to which $T[i]$ points to for some i)
$\Longrightarrow$ seaching takes $O(n)$ time + time to compute hash function
$\Longrightarrow$ no better than using one linked list for all the elements.

Average case is more interesting !

We define the load factor α for T as

$$\alpha = n/m,$$

that is, the average number of elements stored in a list $T[i]$, $i \in \{1, \ldots, m\}$.

The average-case performance of hashing depends on how well the hash function h distributes the set of keys to be stored among the m slots, on the average.

A hash function h is **simple uniform**:

Any given element is equally likely to hash into any of the m slots and

where a given element hashes to is independent of where any other elements hash to.

Theorem.
In hashing with chaining a search takes average-case time $\Theta(1 + \alpha)$ *under the simple uniform hashing assumption.*

[proof omitted - see Cormen course book]

$\Longrightarrow$ if $n \leq m$ then $\alpha \leq 1$ and thus, $\Theta(1 + \alpha) = \Theta(1)$
$\Longrightarrow$ Insert-, Delete-, Search-operations take constant expected time and
Insert- and Delete-operations take constant time in the worst-case

Part 4: Resolving collision via chaining

How well does hashing with chaining perform? In particular, how long does it take to search for an element with a given key?

Given: A hash table T with m slots that stores n elements

Worst case: h maps all n element to same slot (that is, the list to which $T[i]$ points to for some i)
$\Longrightarrow$ seaching takes $O(n)$ time + time to compute hash function
$\Longrightarrow$ no better than using one linked list for all the elements.

Average case is more interesting !

We define the load factor α for T as

$$\alpha = n/m,$$

that is, the average number of elements stored in a list $T[i]$, $i \in \{1, \ldots, m\}$.

The average-case performance of hashing depends on how well the hash function h distributes the set of keys to be stored among the m slots, on the average.

A hash function h is **simple uniform**:

Any given element is equally likely to hash into any of the m slots and

where a given element hashes to is independent of where any other elements hash to.

Theorem.
In hashing with chaining a search takes average-case time $\Theta(1 + \alpha)$ *under the simple uniform hashing assumption.*

[proof omitted - see Cormen course book]

$\Longrightarrow$ if $n \leq m$ then $\alpha \leq 1$ and thus, $\Theta(1 + \alpha) = \Theta(1)$
$\Longrightarrow$ Insert-, Delete-, Search-operations take constant expected time and
Insert- and Delete-operations take constant time in the worst-case

Part 4: Resolving collision via chaining

How well does hashing with chaining perform? In particular, how long does it take to search for an element with a given key?

Given: A hash table T with m slots that stores n elements

Worst case: h maps all n element to same slot (that is, the list to which $T[i]$ points to for some i)
$\Longrightarrow$ seaching takes $O(n)$ time + time to compute hash function
$\Longrightarrow$ no better than using one linked list for all the elements.

Average case is more interesting !

We define the load factor α for T as

$$\alpha = n/m,$$

that is, the average number of elements stored in a list $T[i]$, $i \in \{1, \ldots, m\}$.

The average-case performance of hashing depends on how well the hash function h distributes the set of keys to be stored among the m slots, on the average.

A hash function h is **simple uniform**:

Any given element is equally likely to hash into any of the m slots and

where a given element hashes to is independent of where any other elements hash to.

Theorem.
In hashing with chaining a search takes average-case time $\Theta(1 + \alpha)$ *under the simple uniform hashing assumption.*

[proof omitted - see Cormen course book]

$\Longrightarrow$ if $n \leq m$ then $\alpha \leq 1$ and thus, $\Theta(1 + \alpha) = \Theta(1)$
$\Longrightarrow$ Insert-, Delete-, Search-operations take constant expected time and
Insert- and Delete-operations take constant time in the worst-case

Part 4: Resolving collision via chaining

How well does hashing with chaining perform? In particular, how long does it take to search for an element with a given key?

Given: A hash table T with m slots that stores n elements

Worst case: h maps all n element to same slot (that is, the list to which $T[i]$ points to for some i)
$\Longrightarrow$ seaching takes $O(n)$ time + time to compute hash function
$\Longrightarrow$ no better than using one linked list for all the elements.

Average case is more interesting !

We define the load factor α for T as

$$\alpha = n/m,$$

that is, the average number of elements stored in a list $T[i]$, $i \in \{1, \ldots, m\}$.

The average-case performance of hashing depends on how well the hash function h distributes the set of keys to be stored among the m slots, on the average.

A hash function h is **simple uniform**:

Any given element is equally likely to hash into any of the m slots and

where a given element hashes to is independent of where any other elements hash to.

Theorem.
In hashing with chaining a search takes average-case time $\Theta(1 + \alpha)$ *under the simple uniform hashing assumption.*

[proof omitted - see Cormen course book]

$\Longrightarrow$ if $n \leq m$ then $\alpha \leq 1$ and thus, $\Theta(1 + \alpha) = \Theta(1)$

$\Longrightarrow$ Insert-, Delete-, Search-operations take constant expected time and

Insert- and Delete-operations take constant time in the worst-case

Part 4: Resolving collision via chaining

How well does hashing with chaining perform? In particular, how long does it take to search for an element with a given key?

Given: A hash table T with m slots that stores n elements

Worst case: h maps all n element to same slot (that is, the list to which $T[i]$ points to for some i)
$\Longrightarrow$ seaching takes $O(n)$ time + time to compute hash function
$\Longrightarrow$ no better than using one linked list for all the elements.

Average case is more interesting !

We define the load factor α for T as

$$\alpha = n/m,$$

that is, the average number of elements stored in a list $T[i]$, $i \in \{1, \ldots, m\}$.

The average-case performance of hashing depends on how well the hash function h distributes the set of keys to be stored among the m slots, on the average.

A hash function h is **simple uniform**:

Any given element is equally likely to hash into any of the m slots and

where a given element hashes to is independent of where any other elements hash to.

Theorem.
In hashing with chaining a search takes average-case time $\Theta(1 + \alpha)$ *under the simple uniform hashing assumption.*

[proof omitted - see Cormen course book]

$\Longrightarrow$ if $n \leq m$ then $\alpha \leq 1$ and thus, $\Theta(1 + \alpha) = \Theta(1)$

$\Longrightarrow$ Insert-, Delete-, Search-operations take constant expected time and

Insert- and Delete-operations take constant time in the worst-case

Part 4: Resolving collision via chaining

How well does hashing with chaining perform? In particular, how long does it take to search for an element with a given key?

Given: A hash table T with m slots that stores n elements

Worst case: h maps all n element to same slot (that is, the list to which $T[i]$ points to for some i)
$\Longrightarrow$ seaching takes $O(n)$ time + time to compute hash function
$\Longrightarrow$ no better than using one linked list for all the elements.

Average case is more interesting !

We define the load factor α for T as

$$\alpha = n/m,$$

that is, the average number of elements stored in a list $T[i]$, $i \in \{1, \ldots, m\}$.

The average-case performance of hashing depends on how well the hash function h distributes the set of keys to be stored among the m slots, on the average.

A hash function h is **simple uniform**:

Any given element is equally likely to hash into any of the m slots and

where a given element hashes to is independent of where any other elements hash to.

Theorem.
In hashing with chaining a search takes average-case time $\Theta(1 + \alpha)$ *under the simple uniform hashing assumption.*

[proof omitted - see Cormen course book]

$\Longrightarrow$ if $n \leq m$ then $\alpha \leq 1$ and thus, $\Theta(1 + \alpha) = \Theta(1)$

$\Longrightarrow$ Insert-, Delete-, Search-operations take constant expected time and

Insert- and Delete-operations take constant time in the worst-case

Part 4: Resolving collision via chaining

How well does hashing with chaining perform? In particular, how long does it take to search for an element with a given key?

Given: A hash table T with m slots that stores n elements

Worst case: h maps all n element to same slot (that is, the list to which $T[i]$ points to for some i)
$\Longrightarrow$ seaching takes $O(n)$ time + time to compute hash function
$\Longrightarrow$ no better than using one linked list for all the elements.

Average case is more interesting !

We define the load factor α for T as

$$\alpha = n/m,$$

that is, the average number of elements stored in a list $T[i]$, $i \in \{1, \ldots, m\}$.

The average-case performance of hashing depends on how well the hash function h distributes the set of keys to be stored among the m slots, on the average.

A hash function h is **simple uniform**:

Any given element is equally likely to hash into any of the m slots and

where a given element hashes to is independent of where any other elements hash to.

Theorem.
In hashing with chaining a search takes average-case time $\Theta(1 + \alpha)$ under the simple uniform hashing assumption.

[proof omitted - see Cormen course book]

$\Longrightarrow$ if $n \leq m$ then $\alpha \leq 1$ and thus, $\Theta(1 + \alpha) = \Theta(1)$
$\Longrightarrow$ Insert-, Delete-, Search-operations take constant expected time and
Insert- and Delete-operations take constant time in the worst-case

Part 4: Resolving collision via chaining

How well does hashing with chaining perform? In particular, how long does it take to search for an element with a given key?

Given: A hash table T with m slots that stores n elements

Worst case: h maps all n element to same slot (that is, the list to which $T[i]$ points to for some i)
$\Longrightarrow$ seaching takes $O(n)$ time + time to compute hash function
$\Longrightarrow$ no better than using one linked list for all the elements.

Average case is more interesting !

We define the load factor α for T as

$$\alpha = n/m,$$

that is, the average number of elements stored in a list $T[i]$, $i \in \{1, \ldots, m\}$.

The average-case performance of hashing depends on how well the hash function h distributes the set of keys to be stored among the m slots, on the average.

A hash function h is **simple uniform**:

Any given element is equally likely to hash into any of the m slots and

where a given element hashes to is independent of where any other elements hash to.

Theorem.
In hashing with chaining a search takes average-case time $\Theta(1 + \alpha)$ *under the simple uniform hashing assumption.*

[proof omitted - see Cormen course book]

$\Longrightarrow$ if $n \leq m$ then $\alpha \leq 1$ and thus, $\Theta(1 + \alpha) = \Theta(1)$
$\Longrightarrow$ Insert-, Delete-, Search-operations take constant expected time and
Insert- and Delete-operations take constant time in the worst-case

Part 4: Resolving collision via chaining

How well does hashing with chaining perform? In particular, how long does it take to search for an element with a given key?

Given: A hash table T with m slots that stores n elements

Worst case: h maps all n element to same slot (that is, the list to which $T[i]$ points to for some i)
$\Longrightarrow$ seaching takes $O(n)$ time + time to compute hash function
$\Longrightarrow$ no better than using one linked list for all the elements.

Average case is more interesting !

We define the load factor α for T as

$$\alpha = n/m,$$

that is, the average number of elements stored in a list $T[i]$, $i \in \{1, \ldots, m\}$.

The average-case performance of hashing depends on how well the hash function h distributes the set of keys to be stored among the m slots, on the average.

A hash function h is **simple uniform**:

Any given element is equally likely to hash into any of the m slots and

where a given element hashes to is independent of where any other elements hash to.

Theorem.
In hashing with chaining a search takes average-case time $\Theta(1 + \alpha)$ *under the simple uniform hashing assumption.*

[proof omitted - see Cormen course book]

$\Longrightarrow$ if $n \leq m$ then $\alpha \leq 1$ and thus, $\Theta(1 + \alpha) = \Theta(1)$
$\Longrightarrow$ Insert-, Delete-, Search-operations take constant expected time and
Insert- and Delete-operations take constant time in the worst-case

Part 4: Resolving collision via chaining

How well does hashing with chaining perform? In particular, how long does it take to search for an element with a given key?

Given: A hash table T with m slots that stores n elements

Worst case: h maps all n element to same slot (that is, the list to which $T[i]$ points to for some i)
$\Longrightarrow$ seaching takes $O(n)$ time + time to compute hash function
$\Longrightarrow$ no better than using one linked list for all the elements.

Average case is more interesting !

We define the load factor α for T as

$$\alpha = n/m,$$

that is, the average number of elements stored in a list $T[i]$, $i \in \{1, \ldots, m\}$.

The average-case performance of hashing depends on how well the hash function h distributes the set of keys to be stored among the m slots, on the average.

A hash function h is **simple uniform**:

Any given element is equally likely to hash into any of the m slots and

where a given element hashes to is independent of where any other elements hash to.

Theorem.
In hashing with chaining a search takes average-case time $\Theta(1 + \alpha)$ *under the simple uniform hashing assumption.*

[proof omitted - see Cormen course book]

$\Longrightarrow$ if $n \leq m$ then $\alpha \leq 1$ and thus, $\Theta(1 + \alpha) = \Theta(1)$

$\Longrightarrow$ Insert-, Delete-, Search-operations take constant expected time and
Insert- and Delete-operations take constant time in the worst-case

Part 4: Resolving collision via open addressing

In contrast to chaining, in open addressing, all elements occupy the hash table itself. That is, each hash table entry contains either an element of the dynamic set or NIL.

The idea: when trying to enter the key k into the hash table at position $h(k)$ and it is discovered that $T[h(k)]$ is already occupied, then – according to a fixed rule – unoccupied space (an open one) is used to accommodate k.

Since you cannot know in advance which locations will be occupied and which will not, you define an order for each key in which all storage locations are viewed, one after the other. As soon as a space in question is free, the key is stored there.

In open addressing, the hash function h is a function

$$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$$

such that $(h(k, 0), h(k, 1), \ldots, h(k, m-1))$ is a permutation of $(0, 1, \ldots, m-1)$ for every $k \in U$, called the probe sequence.

Examples ($k \in U$, $i \in \{0, \ldots, m-1\}$ and $h' : U \to \{0, \ldots, m-1\}$ is "auxiliary" hash functions):

- linear probing uses hash function $h(k, i) = (h'(k) + a \cdot i) \bmod m$, $a \neq 0$ constant

Part 4: Resolving collision via open addressing

In contrast to chaining, in open addressing, all elements occupy the hash table itself. That is, each hash table entry contains either an element of the dynamic set or NIL.

The idea: when trying to enter the key k into the hash table at position $h(k)$ and it is discovered that $T[h(k)]$ is already occupied, then – according to a fixed rule – unoccupied space (an open one) is used to accommodate k.

Since you cannot know in advance which locations will be occupied and which will not, you define an order for each key in which all storage locations are viewed, one after the other. As soon as a space in question is free, the key is stored there.

In open addressing, the hash function h is a function

$$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$$

such that $(h(k, 0), h(k, 1), \ldots, h(k, m-1))$ is a permutation of $(0, 1, \ldots, m-1)$ for every $k \in U$, called the probe sequence.

Examples ($k \in U$, $i \in \{0, \ldots, m-1\}$ and $h' : U \to \{0, \ldots, m-1\}$ is "auxiliary" hash functions):

- linear probing uses hash function $h(k, i) = (h'(k) + a \cdot i) \bmod m$, $a \neq 0$ constant

Part 4: Resolving collision via open addressing

In contrast to chaining, in open addressing, all elements occupy the hash table itself. That is, each hash table entry contains either an element of the dynamic set or NIL.

The idea: when trying to enter the key k into the hash table at position $h(k)$ and it is discovered that $T[h(k)]$ is already occupied, then – according to a fixed rule – unoccupied space (an open one) is used to accommodate k.

Since you cannot know in advance which locations will be occupied and which will not, you define an order for each key in which all storage locations are viewed, one after the other. As soon as a space in question is free, the key is stored there.

In open addressing, the hash function h is a function

$$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$$

such that $(h(k, 0), h(k, 1), \ldots, h(k, m-1))$ is a permutation of $(0, 1, \ldots, m-1)$ for every $k \in U$, called the probe sequence.

Examples ($k \in U$, $i \in \{0, \ldots, m-1\}$ and $h' : U \to \{0, \ldots, m-1\}$ is "auxiliary" hash functions):

linear probing uses hash function $h(k, i) = (h'(k) + a \cdot i) \bmod m$, $a \neq 0$ constant

Part 4: Resolving collision via open addressing

In contrast to chaining, in open addressing, all elements occupy the hash table itself. That is, each hash table entry contains either an element of the dynamic set or NIL.

The idea: when trying to enter the key k into the hash table at position $h(k)$ and it is discovered that $T[h(k)]$ is already occupied, then – according to a fixed rule – unoccupied space (an open one) is used to accommodate k.

Since you cannot know in advance which locations will be occupied and which will not, you define an order for each key in which all storage locations are viewed, one after the other. As soon as a space in question is free, the key is stored there.

In open addressing, the hash function h is a function

$$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$$

such that $(h(k, 0), h(k, 1), \ldots, h(k, m-1))$ is a permutation of $(0, 1, \ldots, m-1)$ for every $k \in U$, called the probe sequence.

Examples ($k \in U$, $i \in \{0, \ldots, m-1\}$ and $h' : U \to \{0, \ldots, m-1\}$ is "auxiliary" hash functions):

linear probing uses hash function $h(k, i) = (h'(k) + a \cdot i) \bmod m$, $a \neq 0$ constant

Resolving collision via open addressing – Linear Probing

Example (Linear Probing)
Insert in order: 79, 28, 49, 88, 59 into hash table of size $m = 10$.

Assume: $h'(k) = k$ and $a = 1$ (just for simplicity)
Recall: $h(k, i) = (h'(k) + i) \bmod 10, \ i = 0, 1, 2, \ldots$

0	1	2	3	4	5	6	7	8	9

Resolving collision via open addressing – Linear Probing

Example (Linear Probing)
Insert in order: 79, 28, 49, 88, 59 into hash table of size $m = 10$.

Assume: $h'(k) = k$ and $a = 1$ (just for simplicity)
Recall: $h(k, i) = (h'(k) + i) \bmod 10, \ i = 0, 1, 2, \ldots$

0	1	2	3	4	5	6	7	8	9
									79

$h(79, 0) = (79 + 0) \bmod 10 = 9 \Rightarrow$ slot 9 unoccupied $\Rightarrow T[9] = 79$

Resolving collision via open addressing – Linear Probing

Example (Linear Probing)
Insert in order: 79, 28, 49, 88, 59 into hash table of size $m = 10$.

Assume: $h'(k) = k$ and $a = 1$ (just for simplicity)
Recall: $h(k, i) = (h'(k) + i) \bmod 10, \ i = 0, 1, 2, \ldots$

0	1	2	3	4	5	6	7	8	9
								28	**79**

$h(79, 0) = (79 + 0) \bmod 10 = 9 \Rightarrow$ slot 9 unoccupied $\Rightarrow T[9] = 79$
Then, $h(28, 0) = (28 + 0) \bmod 10 = 8 \Rightarrow$ slot 8 unoccupied $\Rightarrow T[8] = 28$

Resolving collision via open addressing – Linear Probing

Example (Linear Probing)
Insert in order: 79, 28, 49, 88, 59 into hash table of size $m = 10$.

Assume: $h'(k) = k$ and $a = 1$ (just for simplicity)
Recall: $h(k, i) = (h'(k) + i) \bmod 10, \ i = 0, 1, 2, \ldots$

0	1	2	3	4	5	6	7	8	9
49								28	79

Since $h(49, 0) = (49 + 0) \bmod 10 = 9$ and slot 9 is already occupied, we have a collision.
Since $h(49, 1) = (49 + 1) \bmod 10 = 0$ and slot 0 is unoccupied $\Rightarrow T[0] = 49$

Resolving collision via open addressing – Linear Probing

Example (Linear Probing)
Insert in order: 79, 28, 49, 88, 59 into hash table of size $m = 10$.

Assume: $h'(k) = k$ and $a = 1$ (just for simplicity)
Recall: $h(k, i) = (h'(k) + i) \bmod 10, \ i = 0, 1, 2, \ldots$

0	1	2	3	4	5	6	7	8	9
49	**88**							28	79

$h(88, 0) = (88 + 0) \bmod 10 = 8$ occupied!
$h(88, 1) = (88 + 1) \bmod 10 = 9$ occupied!
$h(88, 2) = (88 + 2) \bmod 10 = 0$ occupied!
$h(88, 3) = (88 + 3) \bmod 10 = 1$ unoccupied!

Resolving collision via open addressing – Linear Probing

Example (Linear Probing)
Insert in order: 79, 28, 49, 88, 59 into hash table of size $m = 10$.

Assume: $h'(k) = k$ and $a = 1$ (just for simplicity)
Recall: $h(k, i) = (h'(k) + i) \bmod 10, \ i = 0, 1, 2, \ldots$

0	1	2	3	4	5	6	7	8	9
49	88	**59**						28	79

$h(59, 0) = (59 + 0) \bmod 10 = 9$ occupied!
$h(59, 1) = (59 + 1) \bmod 10 = 0$ occupied!
$h(59, 2) = (59 + 2) \bmod 10 = 1$ occupied!
$h(59, 3) = (59 + 3) \bmod 10 = 2$ unoccupied!

Part 4: Resolving collision via open addressing

In open addressing, the hash function h is a function

$$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$$

such that $(h(k, 0), h(k, 1), \ldots, h(k, m-1))$ is a permutation of $(0, 1, \ldots, m-1)$ for every $k \in U$, called the probe sequence.

Examples ($k \in U$, $i \in \{0, \ldots, m-1\}$ and $h', h'' : U \to \{0, \ldots, m-1\}$ are "auxiliary" hash function):

linear probing uses hash function $h(k, i) = (h'(k) + a \cdot i) \bmod m$, $a \neq 0$ constant

quadratic probing uses hash function $h(k, i) = (h'(k) + a \cdot i + b \cdot i^2) \bmod m$ where a and $b \neq 0$ are constants.

a, b must be chosen such that the probe sequence is indeed a permutation of $(0, \ldots, m-1)$.

double hashing uses the hash function $h(k, i) = (h'(k) + i \cdot h''(k)) \bmod m$

(linear probing is just a special case where $h''(k) = a$ for all k)

$h''(k)$ must be relatively prime to m so that all slots are probed.

We omit further theoretical aspects of runtime at this point.

Part 4: Resolving collision via open addressing

In open addressing, the hash function h is a function

$$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$$

such that $(h(k, 0), h(k, 1), \ldots, h(k, m-1))$ is a permutation of $(0, 1, \ldots, m-1)$ for every $k \in U$, called the probe sequence.

Examples ($k \in U$, $i \in \{0, \ldots, m-1\}$ and $h', h'' : U \to \{0, \ldots, m-1\}$ are "auxiliary" hash function):

- linear probing uses hash function $h(k, i) = (h'(k) + a \cdot i) \bmod m$, $a \neq 0$ constant
- quadratic probing uses hash function $h(k, i) = (h'(k) + a \cdot i + b \cdot i^2) \bmod m$ where a and $b \neq 0$ are constants.
  a, b must be chosen such that the probe sequence is indeed a permutation of $(0, \ldots, m-1)$.
- double hashing uses the hash function $h(k, i) = (h'(k) + i \cdot h''(k)) \bmod m$
  (linear probing is just a special case where $h''(k) = a$ for all k)
  $h''(k)$ must be relatively prime to m so that all slots are probed.

We omit further theoretical aspects of runtime at this point.

Part 4: Resolving collision via open addressing

In open addressing, the hash function h is a function

$$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$$

such that $(h(k, 0), h(k, 1), \ldots, h(k, m-1))$ is a permutation of $(0, 1, \ldots, m-1)$ for every $k \in U$, called the probe sequence.

Examples ($k \in U$, $i \in \{0, \ldots, m-1\}$ and h', $h'' : U \to \{0, \ldots, m-1\}$ are "auxiliary" hash function):

linear probing uses hash function $h(k, i) = (h'(k) + a \cdot i) \bmod m$, $a \neq 0$ constant

quadratic probing uses hash function $h(k, i) = (h'(k) + a \cdot i + b \cdot i^2) \bmod m$ where a and $b \neq 0$ are constants.

a, b must be chosen such that the probe sequence is indeed a permutation of $(0, \ldots, m-1)$.

double hashing uses the hash function $h(k, i) = (h'(k) + i \cdot h''(k)) \bmod m$

(linear probing is just a special case where $h''(k) = a$ for all k)
$h''(k)$ must be relatively prime to m so that all slots are probed.

We omit further theoretical aspects of runtime at this point.

Part 4: Resolving collision via open addressing

In open addressing, the hash function h is a function

$$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$$

such that $(h(k, 0), h(k, 1), \ldots, h(k, m-1))$ is a permutation of $(0, 1, \ldots, m-1)$ for every $k \in U$, called the probe sequence.

Examples ($k \in U$, $i \in \{0, \ldots, m-1\}$ and $h', h'' : U \to \{0, \ldots, m-1\}$ are "auxiliary" hash function):

linear probing uses hash function $h(k, i) = (h'(k) + a \cdot i) \bmod m$, $a \neq 0$ constant

quadratic probing uses hash function $h(k, i) = (h'(k) + a \cdot i + b \cdot i^2) \bmod m$ where a and $b \neq 0$ are constants.
a, b must be chosen such that the probe sequence is indeed a permutation of $(0, \ldots, m-1)$.

double hashing uses the hash function $h(k, i) = (h'(k) + i \cdot h''(k)) \bmod m$
(linear probing is just a special case where $h''(k) = a$ for all k)
$h''(k)$ must be relatively prime to m so that all slots are probed.

We omit further theoretical aspects of runtime at this point.

Part 4: Resolving collision – summary

Two different approaches for resolving collisions: chaining and open addressing (e.g. linear probing).

Which one is better? This question is beyond theoretical analysis, as the answer depends on the intended use and many technical parameters.

A disadvantage of open addressing is that search times become high when the number of elements approaches the table size. For chaining, the expected access time remains small. On the other hand, chaining wastes space on pointers that open addressing could use for a larger table.

However, experimental results show that both techniques performed almost equally well when they were given the same amount of memory.

Algorithms and Data Structures - The Basic Toolbox, Mehlhorn and Sanders, Springer, 2008

Part 4: Resolving collision – summary

Two different approaches for resolving collisions: chaining and open addressing (e.g. linear probing).

Which one is better? This question is beyond theoretical analysis, as the answer depends on the intended use and many technical parameters.

A disadvantage of open addressing is that search times become high when the number of elements approaches the table size. For chaining, the expected access time remains small. On the other hand, chaining wastes space on pointers that open addressing could use for a larger table.

However, experimental results show that both techniques performed almost equally well when they were given the same amount of memory.

Algorithms and Data Structures - The Basic Toolbox, Mehlhorn and Sanders, Springer, 2008

Part 4: Resolving collision – summary

Two different approaches for resolving collisions: chaining and open addressing (e.g. linear probing).

Which one is better? This question is beyond theoretical analysis, as the answer depends on the intended use and many technical parameters.

A disadvantage of open addressing is that search times become high when the number of elements approaches the table size. For chaining, the expected access time remains small. On the other hand, chaining wastes space on pointers that open addressing could use for a larger table.

However, experimental results show that both techniques performed almost equally well when they were given the same amount of memory.

Algorithms and Data Structures - The Basic Toolbox, Mehlhorn and Sanders, Springer, 2008

Part 4: Resolving collision – summary

Two different approaches for resolving collisions: chaining and open addressing (e.g. linear probing).

Which one is better? This question is beyond theoretical analysis, as the answer depends on the intended use and many technical parameters.

A disadvantage of open addressing is that search times become high when the number of elements approaches the table size. For chaining, the expected access time remains small. On the other hand, chaining wastes space on pointers that open addressing could use for a larger table.

However, experimental results show that both techniques performed almost equally well when they were given the same amount of memory.

Algorithms and Data Structures - The Basic Toolbox, Mehlhorn and Sanders, Springer, 2008

Factors Affecting Hash Function Performance

The efficiency of a hash function depends on many factors and parameters!

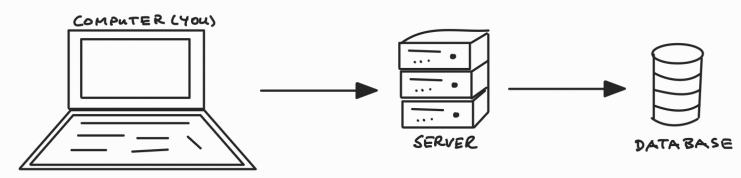
- Type of hash function
- Data type of the key space: Integer, String, ...
- Distribution of currently used keys
- Load factor α of the hash table
- Number of records that can be stored at an address without causing collisions (List capacity)
- Collision resolution technique
- Possibly, the order of storing the records (open addressing)

Part 4: Application of Hashing - Bloom Filter

Bloom filters are probabilistic data structures based on hashing to check memberships in sets.

Aim: avoid time-consuming queries.

Example: Suppose you want to register to a web-service and are ask to provide a username.



Question: Does your username already exist?

Answer: check in database – can be very time-consuming.

Solution: Bloom-filter "If we can say for sure that username does not exist we can skip time-consuming query!"

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

Comic Example: $S = \{Jan, Ada, Leo, Tod\}$, $B[0..7] = [0, 0, 0, 0, 0, 0, 0, 0]$, and $\mathcal{H} = \{h_1\}$

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

Comic Example: $S = \{Jan, Ada, Leo, Tod\}$, $B[0..7] = [0, 0, 0, 0, 0, 0, 0, 0]$, and $\mathcal{H} = \{h_1\}$

Add Jan: $h_1(Jan) = 1$

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

Comic Example: $S = \{Jan, Ada, Leo, Tod\}$, $B[0..7] = [0, 0, 0, 0, 0, 0, 0, 0]$, and $\mathcal{H} = \{h_1\}$

$$\text{Add Jan: } h_1(Jan) = 1 \qquad \Longrightarrow \quad B = [0, 1, 0, 0, 0, 0, 0, 0].$$

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

Comic Example: $S = \{Jan, Ada, Leo, Tod\}$, $B[0..7] = [0, 0, 0, 0, 0, 0, 0, 0]$, and $\mathcal{H} = \{h_1\}$

Add Jan: $h_1(Jan) = 1$
Add Ada: $h_1(Ada) = 2$

$\implies B = [0, 1, 0, 0, 0, 0, 0, 0]$.

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

Comic Example: $S = \{Jan, Ada, Leo, Tod\}$, $B[0..7] = [0, 0, 0, 0, 0, 0, 0, 0]$, and $\mathcal{H} = \{h_1\}$

Add Jan: $h_1(Jan) = 1 \implies B = [0, 1, 0, 0, 0, 0, 0, 0]$.
Add Ada: $h_1(Ada) = 2 \implies B = [0, 1, 1, 0, 0, 0, 0, 0]$.

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

Comic Example: $S = \{Jan, Ada, Leo, Tod\}$, $B[0..7] = [0, 0, 0, 0, 0, 0, 0, 0]$, and $\mathcal{H} = \{h_1\}$

Add Jan: $h_1(Jan) = 1 \implies B = [0, 1, 0, 0, 0, 0, 0, 0]$.
Add Ada: $h_1(Ada) = 2 \implies B = [0, 1, 1, 0, 0, 0, 0, 0]$.
Add Leo: $h_1(Leo) = 3$

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

Comic Example: $S = \{Jan, Ada, Leo, Tod\}$, $B[0..7] = [0, 0, 0, 0, 0, 0, 0, 0]$, and $\mathcal{H} = \{h_1\}$

Add Jan: $h_1(Jan) = 1 \implies B = [0, 1, 0, 0, 0, 0, 0, 0]$.
Add Ada: $h_1(Ada) = 2 \implies B = [0, 1, 1, 0, 0, 0, 0, 0]$.
Add Leo: $h_1(Leo) = 3 \implies B = [0, 1, 1, 1, 0, 0, 0, 0]$.

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

Comic Example: $S = \{Jan, Ada, Leo, Tod\}$, $B[0..7] = [0, 0, 0, 0, 0, 0, 0, 0]$, and $\mathcal{H} = \{h_1\}$

Add Jan: $h_1(Jan) = 1 \implies B = [0, 1, 0, 0, 0, 0, 0, 0]$.
Add Ada: $h_1(Ada) = 2 \implies B = [0, 1, 1, 0, 0, 0, 0, 0]$.
Add Leo: $h_1(Leo) = 3 \implies B = [0, 1, 1, 1, 0, 0, 0, 0]$.
Add Tod: $h_1(Tod) = 0$

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

Comic Example: $S = \{Jan, Ada, Leo, Tod\}$, $B[0..7] = [0, 0, 0, 0, 0, 0, 0, 0]$, and $\mathcal{H} = \{h_1\}$

Add Jan: $h_1(Jan) = 1 \implies B = [0, 1, 0, 0, 0, 0, 0, 0]$.
Add Ada: $h_1(Ada) = 2 \implies B = [0, 1, 1, 0, 0, 0, 0, 0]$.
Add Leo: $h_1(Leo) = 3 \implies B = [0, 1, 1, 1, 0, 0, 0, 0]$.
Add Tod: $h_1(Tod) = 0 \implies B = [1, 1, 1, 1, 0, 0, 0, 0]$.

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

Comic Example: $S = \{Jan, Ada, Leo, Tod\}$, $B[0..7] = [0, 0, 0, 0, 0, 0, 0, 0]$, and $\mathcal{H} = \{h_1\}$

Add Jan: $h_1(Jan) = 1 \implies B = [0, 1, 0, 0, 0, 0, 0, 0]$.
Add Ada: $h_1(Ada) = 2 \implies B = [0, 1, 1, 0, 0, 0, 0, 0]$.
Add Leo: $h_1(Leo) = 3 \implies B = [0, 1, 1, 1, 0, 0, 0, 0]$.
Add Tod: $h_1(Tod) = 0 \implies B = [1, 1, 1, 1, 0, 0, 0, 0]$.

$(B, \mathcal{H})$ represents S.

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

Comic Example: $S = \{Jan, Ada, Leo, Tod\}$, $B[0..7] = [0, 0, 0, 0, 0, 0, 0, 0]$, and $\mathcal{H} = \{h_1\}$

Add Jan: $h_1(Jan) = 1 \implies B = [0, 1, 0, 0, 0, 0, 0, 0]$.
Add Ada: $h_1(Ada) = 2 \implies B = [0, 1, 1, 0, 0, 0, 0, 0]$.
Add Leo: $h_1(Leo) = 3 \implies B = [0, 1, 1, 1, 0, 0, 0, 0]$.
Add Tod: $h_1(Tod) = 0 \implies B = [1, 1, 1, 1, 0, 0, 0, 0]$.

$(B, \mathcal{H})$ represents S.

Now we want to check if username "Mia" exists and say we have $h_1(Mia) = 4$.

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

Comic Example: $S = \{Jan, Ada, Leo, Tod\}$, $B[0..7] = [0, 0, 0, 0, 0, 0, 0, 0]$, and $\mathcal{H} = \{h_1\}$

Add Jan: $h_1(Jan) = 1 \implies B = [0, 1, 0, 0, 0, 0, 0, 0]$.
Add Ada: $h_1(Ada) = 2 \implies B = [0, 1, 1, 0, 0, 0, 0, 0]$.
Add Leo: $h_1(Leo) = 3 \implies B = [0, 1, 1, 1, 0, 0, 0, 0]$.
Add Tod: $h_1(Tod) = 0 \implies B = [1, 1, 1, 1, 0, 0, 0, 0]$.

$(B, \mathcal{H})$ represents S.

Now we want to check if username "Mia" exists and say we have $h_1(Mia) = 4$.

Since $B[4] = 0$, we can say for sure that "Mia" does not exist and we don't need to look up the database S!

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

Comic Example: $S = \{Jan, Ada, Leo, Tod\}$, $B[0..7] = [0, 0, 0, 0, 0, 0, 0, 0]$, and $\mathcal{H} = \{h_1\}$

Add Jan: $h_1(Jan) = 1 \implies B = [0, 1, 0, 0, 0, 0, 0, 0]$.
Add Ada: $h_1(Ada) = 2 \implies B = [0, 1, 1, 0, 0, 0, 0, 0]$.
Add Leo: $h_1(Leo) = 3 \implies B = [0, 1, 1, 1, 0, 0, 0, 0]$.
Add Tod: $h_1(Tod) = 0 \implies B = [1, 1, 1, 1, 0, 0, 0, 0]$.

$(B, \mathcal{H})$ represents S.

Now we want to check if username "Mia" exists and say we have $h_1(Mia) = 4$.
Since $B[4] = 0$, we can say for sure that "Mia" does not exist and we don't need to look up the database S!

Now we want to check if username "Tim" exists and say we have $h_1(Tim) = 1$.

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

Comic Example: $S = \{Jan, Ada, Leo, Tod\}$, $B[0..7] = [0, 0, 0, 0, 0, 0, 0, 0]$, and $\mathcal{H} = \{h_1\}$

Add Jan: $h_1(Jan) = 1 \implies B = [0, 1, 0, 0, 0, 0, 0, 0]$.
Add Ada: $h_1(Ada) = 2 \implies B = [0, 1, 1, 0, 0, 0, 0, 0]$.
Add Leo: $h_1(Leo) = 3 \implies B = [0, 1, 1, 1, 0, 0, 0, 0]$.
Add Tod: $h_1(Tod) = 0 \implies B = [1, 1, 1, 1, 0, 0, 0, 0]$.

$(B, \mathcal{H})$ represents S.

Now we want to check if username "Mia" exists and say we have $h_1(Mia) = 4$.
Since $B[4] = 0$, we can say for sure that "Mia" does not exist and we don't need to look up the database S!

Now we want to check if username "Tim" exists and say we have $h_1(Tim) = 1$.
Although $B[1] = 1$, we cannot say with certainty that "Tim" exists or not due to possible collisions and we must look up S!

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

Comic Example: $S = \{Jan, Ada, Leo, Tod\}$, $B[0..7] = [0, 0, 0, 0, 0, 0, 0, 0]$, and $\mathcal{H} = \{h_1\}$

$$
\begin{array}{ll}
\text{Add Jan: } h_1(Jan) = 1 & \Longrightarrow B = [0, 1, 0, 0, 0, 0, 0, 0]. \\
\text{Add Ada: } h_1(Ada) = 2 & \Longrightarrow B = [0, 1, 1, 0, 0, 0, 0, 0]. \\
\text{Add Leo: } h_1(Leo) = 3 & \Longrightarrow B = [0, 1, 1, 1, 0, 0, 0, 0]. \\
\text{Add Tod: } h_1(Tod) = 0 & \Longrightarrow B = [1, 1, 1, 1, 0, 0, 0, 0].
\end{array}
$$

$(B, \mathcal{H})$ represents S.

Now we want to check if username "Mia" exists and say we have $h_1(Mia) = 4$.

Since $B[4] = 0$, we can say for sure that "Mia" does not exist and we don't need to look up the database S!

Now we want to check if username "Tim" exists and say we have $h_1(Tim) = 1$.

Although $B[1] = 1$, we cannot say with certainty that "Tim" exists or not due to possible collisions and we must look up S!

Important Observation:

We can never have false-negatives (0 bit in B leads never to wrong conclusion that key is in S)!

However, false-positives are possible (key is "possibly" in the S (bit in B = 1), but it's not.)

To decrease false-positive rate use more hash-functions!

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

Comic Example: $S = \{Jan, Ada, Leo, Tod\}$, $B[0..7] = [0, 0, 0, 0, 0, 0, 0, 0]$, and $\mathcal{H} = \{h_1, h_2\}$

Important Observation:

We can never have false-negatives (0 bit in B leads never to wrong conclusion that key is in S)!

However, false-positives are possible (key is "possibly" in the S (bit in B = 1), but it's not.)

To decrease false-positive rate use more hash-functions!

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

Comic Example: $S = \{Jan, Ada, Leo, Tod\}$, $B[0..7] = [0, 0, 0, 0, 0, 0, 0, 0]$, and $\mathcal{H} = \{h_1, h_2\}$

$$\left.\begin{array}{l} h_1(Jan) = 1, h_2(Jan) = 5 \\ h_1(Ada) = 2, h_2(Ada) = 7 \\ h_1(Leo) = 3, h_2(Leo) = 5 \\ h_1(Tod) = 0, h_2(Tod) = 3 \end{array}\right\}$$

Important Observation:

We can never have false-negatives (0 bit in B leads never to wrong conclusion that key is in S)!

However, false-positives are possible (key is "possibly" in the S (bit in B = 1), but it's not.)

To decrease false-positive rate use more hash-functions!

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

Comic Example: $S = \{Jan, Ada, Leo, Tod\}$, $B[0..7] = [0, 0, 0, 0, 0, 0, 0, 0]$, and $\mathcal{H} = \{h_1, h_2\}$

$$\left.\begin{array}{l} h_1(Jan) = 1, h_2(Jan) = 5 \\ h_1(Ada) = 2, h_2(Ada) = 7 \\ h_1(Leo) = 3, h_2(Leo) = 5 \\ h_1(Tod) = 0, h_2(Tod) = 3 \end{array}\right\} \implies B = [1, 1, 1, 1, 0, 1, 0, 1]$$

$(B, \mathcal{H})$ represents S.

Important Observation:

We can never have false-negatives (0 bit in B leads never to wrong conclusion that key is in S)!

However, false-positives are possible (key is "possibly" in the S (bit in B = 1), but it's not.)

To decrease false-positive rate use more hash-functions!

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

Comic Example: $S = \{Jan, Ada, Leo, Tod\}$, $B[0..7] = [0, 0, 0, 0, 0, 0, 0, 0]$, and $\mathcal{H} = \{h_1, h_2\}$

$$\left.\begin{array}{l} h_1(Jan) = 1, h_2(Jan) = 5 \\ h_1(Ada) = 2, h_2(Ada) = 7 \\ h_1(Leo) = 3, h_2(Leo) = 5 \\ h_1(Tod) = 0, h_2(Tod) = 3 \end{array}\right\} \implies B = [1, 1, 1, 1, 0, 1, 0, 1]$$

$(B, \mathcal{H})$ represents S.

Now we want to check if username "Mia" exists and say we have $h_1(Mia) = 4$ and $h_2(Mia) = 6$.

Important Observation:

We can never have false-negatives (0 bit in B leads never to wrong conclusion that key is in S)!

However, false-positives are possible (key is "possibly" in the S (bit in B = 1), but it's not.)

To decrease false-positive rate use more hash-functions!

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

Comic Example: $S = \{Jan, Ada, Leo, Tod\}$, $B[0..7] = [0, 0, 0, 0, 0, 0, 0, 0]$, and $\mathcal{H} = \{h_1, h_2\}$

$$\left.\begin{array}{l} h_1(Jan) = 1, h_2(Jan) = 5 \\ h_1(Ada) = 2, h_2(Ada) = 7 \\ h_1(Leo) = 3, h_2(Leo) = 5 \\ h_1(Tod) = 0, h_2(Tod) = 3 \end{array}\right\} \implies B = [1, 1, 1, 1, 0, 1, 0, 1]$$

$(B, \mathcal{H})$ represents S.

Now we want to check if username "Mia" exists and say we have $h_1(Mia) = 4$ and $h_2(Mia) = 6$.

Again, since $B[4] = 0$, we can say for sure that "Mia" does not exist and we don't need to look up the database S!

Important Observation:

We can never have false-negatives (0 bit in B leads never to wrong conclusion that key is in S)!

However, false-positives are possible (key is "possibly" in the S (bit in B = 1), but it's not.)

To decrease false-positive rate use more hash-functions!

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

Comic Example: $S = \{Jan, Ada, Leo, Tod\}$, $B[0..7] = [0, 0, 0, 0, 0, 0, 0, 0]$, and $\mathcal{H} = \{h_1, h_2\}$

$$\left.\begin{array}{l} h_1(Jan) = 1, h_2(Jan) = 5 \\ h_1(Ada) = 2, h_2(Ada) = 7 \\ h_1(Leo) = 3, h_2(Leo) = 5 \\ h_1(Tod) = 0, h_2(Tod) = 3 \end{array}\right\} \implies B = [1, 1, 1, 1, 0, 1, 0, 1]$$

$(B, \mathcal{H})$ represents S.

Now we want to check if username "Mia" exists and say we have $h_1(Mia) = 4$ and $h_2(Mia) = 6$.
Again, since $B[4] = 0$, we can say for sure that "Mia" does not exist and we don't need to look up the database S!

Now we want to check if username "Tim" exists and say we have $h_1(Tim) = 1$ and $h_2(Tim) = 6$.

Important Observation:

We can never have false-negatives (0 bit in B leads never to wrong conclusion that key is in S)!
However, false-positives are possible (key is "possibly" in the S (bit in B = 1), but it's not.)
To decrease false-positive rate use more hash-functions!

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

Comic Example: $S = \{Jan, Ada, Leo, Tod\}$, $B[0..7] = [0, 0, 0, 0, 0, 0, 0, 0]$, and $\mathcal{H} = \{h_1, h_2\}$

$$\left.\begin{array}{l} h_1(Jan) = 1, h_2(Jan) = 5 \\ h_1(Ada) = 2, h_2(Ada) = 7 \\ h_1(Leo) = 3, h_2(Leo) = 5 \\ h_1(Tod) = 0, h_2(Tod) = 3 \end{array}\right\} \implies B = [1, 1, 1, 1, 0, 1, 0, 1]$$

$(B, \mathcal{H})$ represents S.

Now we want to check if username "Mia" exists and say we have $h_1(Mia) = 4$ and $h_2(Mia) = 6$.
Again, since $B[4] = 0$, we can say for sure that "Mia" does not exist and we don't need to look up the database S!

Now we want to check if username "Tim" exists and say we have $h_1(Tim) = 1$ and $h_2(Tim) = 6$.
Although $B[1] = 1$ we have $B[6] = 0$ and we can say for sure that "Tim" does not exist and we don't need to look up S!

Important Observation:

We can never have false-negatives (0 bit in B leads never to wrong conclusion that key is in S)!
However, false-positives are possible (key is "possibly" in the S (bit in B = 1), but it's not.)
To decrease false-positive rate use more hash-functions!

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

In summary, if we want to check membership:

If $B[h_i(x)] = 0$ for at least one i, then it is ensured that $x \notin S$.

Otherwise (i.e., $B[h_i(x)] = 1$ for all i with $1 \leq i \leq k$), we must compare all elements in S with x to verify if $x \in S$ or $x \notin S$.

Question: What is the false-positive rate and how can we minimize it?

Important Observation:

We can never have false-negatives (0 bit in B leads never to wrong conclusion that key is in S)!

However, false-positives are possible (key is "possibly" in the S (bit in B = 1), but it's not.)

To decrease false-positive rate use more hash-functions!

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

We assume that all hash functions $h \in \mathcal{H}$ are simple uniform and independent from each other.

$\frac{1}{m}$ probability that $h(x) = i$ and thus, $B[h(x)] = 1$ for $h \in \mathcal{H}$

$1 - \frac{1}{m}$ probability that $B[i]$ is not set to 1 by a given $h \in \mathcal{H}$

$(1 - \frac{1}{m})^k$ probability that $B[i]$ is not set to 1 by any of the k hash-functions $h \in \mathcal{H}$

From analysis: $\frac{1}{e} = \lim_{m \to \infty} \left(1 - \frac{1}{m}\right)^m \implies \left(1 - \frac{1}{m}\right)^k = \left(1 - \frac{1}{m}\right)^{\frac{km}{m}} = \left(\left(1 - \frac{1}{m}\right)^m\right)^{\frac{k}{m}}$ for large m $\approx \left(\frac{1}{e}\right)^{\frac{k}{m}} = e^{-\frac{k}{m}}$

$(1 - \frac{1}{m})^{kn} \approx e^{-\frac{kn}{m}}$ probability that $B[i]$ is not set to 1 by any $h \in \mathcal{H}$ after "adding" the n elements from S

$1 - (1 - \frac{1}{m})^{kn} \approx 1 - e^{-\frac{kn}{m}}$ probability that $B[i] = 1$ after "adding" the n elements from S

$\epsilon = \left(1 - e^{-\frac{kn}{m}}\right)^k$ probabiliy that all k entries $B[h_i(x)] = 1$ for x (this could cause the algorithm to erroneously claim that the element is in the set)

Observation: ϵ decreases with increasing m (more slots) and increases with with increasing n (more elements)

Assume we have $|S| = n$ and some m. What is an optimal number k of hash functions?: $\frac{d\epsilon}{dk} = 0 \implies k = \frac{m}{n} \ln(2)$

Assume we have $|S| = n$, a desired false-postive rate ϵ and an optimal k.

What is the required number m of bits?: replace k by $\frac{m}{n} \ln(2)$ in $\epsilon \implies m = \frac{-n \ln(\epsilon)}{\ln(2)^2}$ ($\epsilon < 1$, so $-n \ln(\epsilon)$ is positive)

Example: $|S| = 1000$ and want $\epsilon = 0.1$ then we should choose $m \approx 4800$ and $k \approx 3$

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

We assume that all hash functions $h \in \mathcal{H}$ are simple uniform and independent from each other.

$\frac{1}{m}$ probability that $h(x) = i$ and thus, $B[h(x)] = 1$ for $h \in \mathcal{H}$

$1 - \frac{1}{m}$ probability that $B[i]$ is not set to 1 by a given $h \in \mathcal{H}$

$(1 - \frac{1}{m})^k$ probability that $B[i]$ is not set to 1 by any of the k hash-functions $h \in \mathcal{H}$

From analysis: $\frac{1}{e} = \lim_{m \to \infty} \left(1 - \frac{1}{m}\right)^m \implies \left(1 - \frac{1}{m}\right)^k = \left(1 - \frac{1}{m}\right)^{\frac{km}{m}} = \left(\left(1 - \frac{1}{m}\right)^m\right)^{\frac{k}{m}}$ for large m $\approx \left(\frac{1}{e}\right)^{\frac{k}{m}} = e^{-\frac{k}{m}}$

$(1 - \frac{1}{m})^{kn} \approx e^{-\frac{kn}{m}}$ probability that $B[i]$ is not set to 1 by any $h \in \mathcal{H}$ after "adding" the n elements from S

$1 - (1 - \frac{1}{m})^{kn} \approx 1 - e^{-\frac{kn}{m}}$ probability that $B[i] = 1$ after "adding" the n elements from S

$\epsilon = \left(1 - e^{-\frac{kn}{m}}\right)^k$ probabiliy that all k entries $B[h_i(x)] = 1$ for x (this could cause the algorithm to erroneously claim that the element is in the set)

Observation: ϵ decreases with increasing m (more slots) and increases with with increasing n (more elements)

Assume we have $|S| = n$ and some m. What is an optimal number k of hash functions?: $\frac{d\epsilon}{dk} = 0 \implies k = \frac{m}{n} \ln(2)$

Assume we have $|S| = n$, a desired false-postive rate ϵ and an optimal k.

What is the required number m of bits?: replace k by $\frac{m}{n} \ln(2)$ in $\epsilon \implies m = \frac{-n \ln(\epsilon)}{\ln(2)^2}$ ($\epsilon < 1$, so $-n \ln(\epsilon)$ is positive)

Example: $|S| = 1000$ and want $\epsilon = 0.1$ then we should choose $m \approx 4800$ and $k \approx 3$

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

We assume that all hash functions $h \in \mathcal{H}$ are simple uniform and independent from each other.

$\frac{1}{m}$ probability that $h(x) = i$ and thus, $B[h(x)] = 1$ for $h \in \mathcal{H}$

$1 - \frac{1}{m}$ probability that $B[i]$ is not set to 1 by a given $h \in \mathcal{H}$

$(1 - \frac{1}{m})^k$ probability that $B[i]$ is not set to 1 by any of the k hash-functions $h \in \mathcal{H}$

From analysis: $\frac{1}{e} = \lim_{m \to \infty} \left(1 - \frac{1}{m}\right)^m \implies \left(1 - \frac{1}{m}\right)^k = \left(1 - \frac{1}{m}\right)^{\frac{km}{m}} = \left(\left(1 - \frac{1}{m}\right)^m\right)^{\frac{k}{m}}$ for large m $\approx \left(\frac{1}{e}\right)^{\frac{k}{m}} = e^{-\frac{k}{m}}$

$(1 - \frac{1}{m})^{kn} \approx e^{-\frac{kn}{m}}$ probability that $B[i]$ is not set to 1 by any $h \in \mathcal{H}$ after "adding" the n elements from S

$1 - (1 - \frac{1}{m})^{kn} \approx 1 - e^{-\frac{kn}{m}}$ probability that $B[i] = 1$ after "adding" the n elements from S

$\epsilon = \left(1 - e^{-\frac{kn}{m}}\right)^k$ probabiliy that all k entries $B[h_i(x)] = 1$ for x (this could cause the algorithm to erroneously claim that the element is in the set)

Observation: ϵ decreases with increasing m (more slots) and increases with with increasing n (more elements)

Assume we have $|S| = n$ and some m. What is an optimal number k of hash functions?: $\frac{d\epsilon}{dk} = 0 \implies k = \frac{m}{n} \ln(2)$

Assume we have $|S| = n$, a desired false-postive rate ϵ and an optimal k.

What is the required number m of bits?: replace k by $\frac{m}{n} \ln(2)$ in $\epsilon \implies m = \frac{-n \ln(\epsilon)}{\ln(2)^2}$ ($\epsilon < 1$, so $-n \ln(\epsilon)$ is positive)

Example: $|S| = 1000$ and want $\epsilon = 0.1$ then we should choose $m \approx 4800$ and $k \approx 3$

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

We assume that all hash functions $h \in \mathcal{H}$ are simple uniform and independent from each other.

$\frac{1}{m}$ probability that $h(x) = i$ and thus, $B[h(x)] = 1$ for $h \in \mathcal{H}$

$1 - \frac{1}{m}$ probability that $B[i]$ is not set to 1 by a given $h \in \mathcal{H}$

$(1 - \frac{1}{m})^k$ probability that $B[i]$ is not set to 1 by any of the k hash-functions $h \in \mathcal{H}$

From analysis: $\frac{1}{e} = \lim_{m \to \infty} \left(1 - \frac{1}{m}\right)^m \implies \left(1 - \frac{1}{m}\right)^k = \left(1 - \frac{1}{m}\right)^{\frac{km}{m}} = \left(\left(1 - \frac{1}{m}\right)^m\right)^{\frac{k}{m}}$ for large m $\approx \left(\frac{1}{e}\right)^{\frac{k}{m}} = e^{-\frac{k}{m}}$

$(1 - \frac{1}{m})^{kn} \approx e^{-\frac{kn}{m}}$ probability that $B[i]$ is not set to 1 by any $h \in \mathcal{H}$ after "adding" the n elements from S

$1 - (1 - \frac{1}{m})^{kn} \approx 1 - e^{-\frac{kn}{m}}$ probability that $B[i] = 1$ after "adding" the n elements from S

$\epsilon = \left(1 - e^{-\frac{kn}{m}}\right)^k$ probabiliy that all k entries $B[h_i(x)] = 1$ for x (this could cause the algorithm to erroneously claim that the element is in the set)

Observation: ϵ decreases with increasing m (more slots) and increases with with increasing n (more elements)

Assume we have $|S| = n$ and some m. What is an optimal number k of hash functions?: $\frac{d\epsilon}{dk} = 0 \implies k = \frac{m}{n} \ln(2)$

Assume we have $|S| = n$, a desired false-postive rate ϵ and an optimal k.

What is the required number m of bits?: replace k by $\frac{m}{n} \ln(2)$ in $\epsilon \implies m = \frac{-n \ln(\epsilon)}{\ln(2)^2}$ ($\epsilon < 1$, so $-n \ln(\epsilon)$ is positive)

Example: $|S| = 1000$ and want $\epsilon = 0.1$ then we should choose $m \approx 4800$ and $k \approx 3$

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

We assume that all hash functions $h \in \mathcal{H}$ are simple uniform and independent from each other.

$\frac{1}{m}$ probability that $h(x) = i$ and thus, $B[h(x)] = 1$ for $h \in \mathcal{H}$

$1 - \frac{1}{m}$ probability that $B[i]$ is not set to 1 by a given $h \in \mathcal{H}$

$(1 - \frac{1}{m})^k$ probability that $B[i]$ is not set to 1 by any of the k hash-functions $h \in \mathcal{H}$

From analysis: $\frac{1}{e} = \lim_{m \to \infty} \left(1 - \frac{1}{m}\right)^m \implies \left(1 - \frac{1}{m}\right)^k = \left(1 - \frac{1}{m}\right)^{\frac{km}{m}} = \left(\left(1 - \frac{1}{m}\right)^m\right)^{\frac{k}{m}}$ for large m $\approx \left(\frac{1}{e}\right)^{\frac{k}{m}} = e^{-\frac{k}{m}}$

$(1 - \frac{1}{m})^{kn} \approx e^{-\frac{kn}{m}}$ probability that $B[i]$ is not set to 1 by any $h \in \mathcal{H}$ after "adding" the n elements from S

$1 - (1 - \frac{1}{m})^{kn} \approx 1 - e^{-\frac{kn}{m}}$ probability that $B[i] = 1$ after "adding" the n elements from S

$\epsilon = \left(1 - e^{-\frac{kn}{m}}\right)^k$ probabiliy that all k entries $B[h_i(x)] = 1$ for x (this could cause the algorithm to erroneously claim that the element is in the set)

Observation: ϵ decreases with increasing m (more slots) and increases with with increasing n (more elements)

Assume we have $|S| = n$ and some m. What is an optimal number k of hash functions?: $\frac{d\epsilon}{dk} = 0 \implies k = \frac{m}{n} \ln(2)$

Assume we have $|S| = n$, a desired false-postive rate ϵ and an optimal k.

What is the required number m of bits?: replace k by $\frac{m}{n} \ln(2)$ in $\epsilon \implies m = \frac{-n \ln(\epsilon)}{\ln(2)^2}$ ($\epsilon < 1$, so $-n \ln(\epsilon)$ is positive)

Example: $|S| = 1000$ and want $\epsilon = 0.1$ then we should choose $m \approx 4800$ and $k \approx 3$

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

We assume that all hash functions $h \in \mathcal{H}$ are simple uniform and independent from each other.

$\frac{1}{m}$ probability that $h(x) = i$ and thus, $B[h(x)] = 1$ for $h \in \mathcal{H}$

$1 - \frac{1}{m}$ probability that $B[i]$ is not set to 1 by a given $h \in \mathcal{H}$

$(1 - \frac{1}{m})^k$ probability that $B[i]$ is not set to 1 by any of the k hash-functions $h \in \mathcal{H}$

From analysis: $\frac{1}{e} = \lim_{m \to \infty} \left(1 - \frac{1}{m}\right)^m \implies \left(1 - \frac{1}{m}\right)^k = \left(1 - \frac{1}{m}\right)^{\frac{km}{m}} = \left(\left(1 - \frac{1}{m}\right)^m\right)^{\frac{k}{m}}$ for large m $\approx \left(\frac{1}{e}\right)^{\frac{k}{m}} = e^{-\frac{k}{m}}$

$(1 - \frac{1}{m})^{kn} \approx e^{-\frac{kn}{m}}$ probability that $B[i]$ is not set to 1 by any $h \in \mathcal{H}$ after "adding" the n elements from S

$1 - (1 - \frac{1}{m})^{kn} \approx 1 - e^{-\frac{kn}{m}}$ probability that $B[i] = 1$ after "adding" the n elements from S

$\epsilon = \left(1 - e^{-\frac{kn}{m}}\right)^k$ probabiliy that all k entries $B[h_i(x)] = 1$ for x (this could cause the algorithm to erroneously claim that the element is in the set)

Observation: ϵ decreases with increasing m (more slots) and increases with with increasing n (more elements)

Assume we have $|S| = n$ and some m. What is an optimal number k of hash functions?: $\frac{d\epsilon}{dk} = 0 \implies k = \frac{m}{n} \ln(2)$

Assume we have $|S| = n$, a desired false-postive rate ϵ and an optimal k.

What is the required number m of bits?: replace k by $\frac{m}{n} \ln(2)$ in $\epsilon \implies m = \frac{-n \ln(\epsilon)}{\ln(2)^2}$ ($\epsilon < 1$, so $-n \ln(\epsilon)$ is positive)

Example: $|S| = 1000$ and want $\epsilon = 0.1$ then we should choose $m \approx 4800$ and $k \approx 3$

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

We assume that all hash functions $h \in \mathcal{H}$ are simple uniform and independent from each other.

$\frac{1}{m}$ probability that $h(x) = i$ and thus, $B[h(x)] = 1$ for $h \in \mathcal{H}$

$1 - \frac{1}{m}$ probability that $B[i]$ is not set to 1 by a given $h \in \mathcal{H}$

$(1 - \frac{1}{m})^k$ probability that $B[i]$ is not set to 1 by any of the k hash-functions $h \in \mathcal{H}$

From analysis: $\frac{1}{e} = \lim_{m \to \infty} \left(1 - \frac{1}{m}\right)^m \implies \left(1 - \frac{1}{m}\right)^k = \left(1 - \frac{1}{m}\right)^{\frac{km}{m}} = \left(\left(1 - \frac{1}{m}\right)^m\right)^{\frac{k}{m}}$ for large m $\approx \left(\frac{1}{e}\right)^{\frac{k}{m}} = e^{-\frac{k}{m}}$

$(1 - \frac{1}{m})^{kn} \approx e^{-\frac{kn}{m}}$ probability that $B[i]$ is not set to 1 by any $h \in \mathcal{H}$ after "adding" the n elements from S

$1 - (1 - \frac{1}{m})^{kn} \approx 1 - e^{-\frac{kn}{m}}$ probability that $B[i] = 1$ after "adding" the n elements from S

$\epsilon = \left(1 - e^{-\frac{kn}{m}}\right)^k$ probabiliy that all k entries $B[h_i(x)] = 1$ for x (this could cause the algorithm to erroneously claim that the element is in the set)

Observation: ϵ decreases with increasing m (more slots) and increases with with increasing n (more elements)

Assume we have $|S| = n$ and some m. What is an optimal number k of hash functions?: $\frac{d\epsilon}{dk} = 0 \implies k = \frac{m}{n} \ln(2)$

Assume we have $|S| = n$, a desired false-postive rate ϵ and an optimal k.

What is the required number m of bits?: replace k by $\frac{m}{n} \ln(2)$ in $\epsilon \implies m = \frac{-n \ln(\epsilon)}{\ln(2)^2}$ ($\epsilon < 1$, so $-n \ln(\epsilon)$ is positive)

Example: $|S| = 1000$ and want $\epsilon = 0.1$ then we should choose $m \approx 4800$ and $k \approx 3$

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

We assume that all hash functions $h \in \mathcal{H}$ are simple uniform and independent from each other.

$\frac{1}{m}$ probability that $h(x) = i$ and thus, $B[h(x)] = 1$ for $h \in \mathcal{H}$

$1 - \frac{1}{m}$ probability that $B[i]$ is not set to 1 by a given $h \in \mathcal{H}$

$(1 - \frac{1}{m})^k$ probability that $B[i]$ is not set to 1 by any of the k hash-functions $h \in \mathcal{H}$

From analysis: $\frac{1}{e} = \lim_{m \to \infty} \left(1 - \frac{1}{m}\right)^m \implies \left(1 - \frac{1}{m}\right)^k = \left(1 - \frac{1}{m}\right)^{\frac{km}{m}} = \left(\left(1 - \frac{1}{m}\right)^m\right)^{\frac{k}{m}}$ for large m $\approx \left(\frac{1}{e}\right)^{\frac{k}{m}} = e^{-\frac{k}{m}}$

$(1 - \frac{1}{m})^{kn} \approx e^{-\frac{kn}{m}}$ probability that $B[i]$ is not set to 1 by any $h \in \mathcal{H}$ after "adding" the n elements from S

$1 - (1 - \frac{1}{m})^{kn} \approx 1 - e^{-\frac{kn}{m}}$ probability that $B[i] = 1$ after "adding" the n elements from S

$\epsilon = \left(1 - e^{-\frac{kn}{m}}\right)^k$ probabiliy that all k entries $B[h_i(x)] = 1$ for x (this could cause the algorithm to erroneously claim that the element is in the set)

Observation: ϵ decreases with increasing m (more slots) and increases with with increasing n (more elements)

Assume we have $|S| = n$ and some m. What is an optimal number k of hash functions?: $\frac{d\epsilon}{dk} = 0 \implies k = \frac{m}{n} \ln(2)$

Assume we have $|S| = n$, a desired false-postive rate ϵ and an optimal k.

What is the required number m of bits?: replace k by $\frac{m}{n} \ln(2)$ in $\epsilon \implies m = \frac{-n \ln(\epsilon)}{\ln(2)^2}$ ($\epsilon < 1$, so $-n \ln(\epsilon)$ is positive)

Example: $|S| = 1000$ and want $\epsilon = 0.1$ then we should choose $m \approx 4800$ and $k \approx 3$

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

We assume that all hash functions $h \in \mathcal{H}$ are simple uniform and independent from each other.

$\frac{1}{m}$ probability that $h(x) = i$ and thus, $B[h(x)] = 1$ for $h \in \mathcal{H}$

$1 - \frac{1}{m}$ probability that $B[i]$ is not set to 1 by a given $h \in \mathcal{H}$

$(1 - \frac{1}{m})^k$ probability that $B[i]$ is not set to 1 by any of the k hash-functions $h \in \mathcal{H}$

From analysis: $\frac{1}{e} = \lim_{m \to \infty} \left(1 - \frac{1}{m}\right)^m \implies \left(1 - \frac{1}{m}\right)^k = \left(1 - \frac{1}{m}\right)^{\frac{km}{m}} = \left(\left(1 - \frac{1}{m}\right)^m\right)^{\frac{k}{m}} \underset{\text{for large } m}{\approx} \left(\frac{1}{e}\right)^{\frac{k}{m}} = e^{-\frac{k}{m}}$

$(1 - \frac{1}{m})^{kn} \approx e^{-\frac{kn}{m}}$ probability that $B[i]$ is not set to 1 by any $h \in \mathcal{H}$ after "adding" the n elements from S

$1 - (1 - \frac{1}{m})^{kn} \approx 1 - e^{-\frac{kn}{m}}$ probability that $B[i] = 1$ after "adding" the n elements from S

$\epsilon = \left(1 - e^{-\frac{kn}{m}}\right)^k$ probabiliy that all k entries $B[h_i(x)] = 1$ for x (this could cause the algorithm to erroneously claim that the element is in the set)

Observation: ϵ decreases with increasing m (more slots) and increases with with increasing n (more elements)

Assume we have $|S| = n$ and some m. What is an optimal number k of hash functions?: $\frac{d\epsilon}{dk} = 0 \implies k = \frac{m}{n} \ln(2)$

Assume we have $|S| = n$, a desired false-postive rate ϵ and an optimal k.

What is the required number m of bits?: replace k by $\frac{m}{n} \ln(2)$ in $\epsilon \implies m = \frac{-n \ln(\epsilon)}{\ln(2)^2}$ ($\epsilon < 1$, so $-n \ln(\epsilon)$ is positive)

Example: $|S| = 1000$ and want $\epsilon = 0.1$ then we should choose $m \approx 4800$ and $k \approx 3$

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

We assume that all hash functions $h \in \mathcal{H}$ are simple uniform and independent from each other.

$\frac{1}{m}$ probability that $h(x) = i$ and thus, $B[h(x)] = 1$ for $h \in \mathcal{H}$

$1 - \frac{1}{m}$ probability that $B[i]$ is not set to 1 by a given $h \in \mathcal{H}$

$(1 - \frac{1}{m})^k$ probability that $B[i]$ is not set to 1 by any of the k hash-functions $h \in \mathcal{H}$

From analysis: $\frac{1}{e} = \lim_{m \to \infty} \left(1 - \frac{1}{m}\right)^m \implies \left(1 - \frac{1}{m}\right)^k = \left(1 - \frac{1}{m}\right)^{\frac{km}{m}} = \left(\left(1 - \frac{1}{m}\right)^m\right)^{\frac{k}{m}}$ for large m $\underset{\approx}{} \left(\frac{1}{e}\right)^{\frac{k}{m}} = e^{-\frac{k}{m}}$

$(1 - \frac{1}{m})^{kn} \approx e^{-\frac{kn}{m}}$ probability that $B[i]$ is not set to 1 by any $h \in \mathcal{H}$ after "adding" the n elements from S

$1 - (1 - \frac{1}{m})^{kn} \approx 1 - e^{-\frac{kn}{m}}$ probability that $B[i] = 1$ after "adding" the n elements from S

$\epsilon = \left(1 - e^{-\frac{kn}{m}}\right)^k$ probabiliy that all k entries $B[h_i(x)] = 1$ for x (this could cause the algorithm to erroneously claim that the element is in the set)

Observation: ϵ decreases with increasing m (more slots) and increases with with increasing n (more elements)

Assume we have $|S| = n$ and some m. What is an optimal number k of hash functions?: $\frac{d\epsilon}{dk} = 0 \implies k = \frac{m}{n} \ln(2)$

Assume we have $|S| = n$, a desired false-postive rate ϵ and an optimal k.

What is the required number m of bits?: replace k by $\frac{m}{n} \ln(2)$ in $\epsilon \implies m = \frac{-n \ln(\epsilon)}{\ln(2)^2}$ ($\epsilon < 1$, so $-n \ln(\epsilon)$ is positive)

Example: $|S| = 1000$ and want $\epsilon = 0.1$ then we should choose $m \approx 4800$ and $k \approx 3$

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

We assume that all hash functions $h \in \mathcal{H}$ are simple uniform and independent from each other.

$\frac{1}{m}$ probability that $h(x) = i$ and thus, $B[h(x)] = 1$ for $h \in \mathcal{H}$

$1 - \frac{1}{m}$ probability that $B[i]$ is not set to 1 by a given $h \in \mathcal{H}$

$(1 - \frac{1}{m})^k$ probability that $B[i]$ is not set to 1 by any of the k hash-functions $h \in \mathcal{H}$

From analysis: $\frac{1}{e} = \lim_{m \to \infty} \left(1 - \frac{1}{m}\right)^m \implies \left(1 - \frac{1}{m}\right)^k = \left(1 - \frac{1}{m}\right)^{\frac{km}{m}} = \left(\left(1 - \frac{1}{m}\right)^m\right)^{\frac{k}{m}}$ for large m $\underset{\approx}{} \left(\frac{1}{e}\right)^{\frac{k}{m}} = e^{-\frac{k}{m}}$

$(1 - \frac{1}{m})^{kn} \approx e^{-\frac{kn}{m}}$ probability that $B[i]$ is not set to 1 by any $h \in \mathcal{H}$ after "adding" the n elements from S

$1 - (1 - \frac{1}{m})^{kn} \approx 1 - e^{-\frac{kn}{m}}$ probability that $B[i] = 1$ after "adding" the n elements from S

$\epsilon = \left(1 - e^{-\frac{kn}{m}}\right)^k$ probabiliy that all k entries $B[h_i(x)] = 1$ for x (this could cause the algorithm to erroneously claim that the element is in the set)

Observation: ϵ decreases with increasing m (more slots) and increases with with increasing n (more elements)

Assume we have $|S| = n$ and some m. What is an optimal number k of hash functions?: $\frac{d\epsilon}{dk} = 0 \implies k = \frac{m}{n} \ln(2)$

Assume we have $|S| = n$, a desired false-postive rate ϵ and an optimal k.

What is the required number m of bits?: replace k by $\frac{m}{n} \ln(2)$ in $\epsilon \implies m = \frac{-n \ln(\epsilon)}{\ln(2)^2}$ ($\epsilon < 1$, so $-n \ln(\epsilon)$ is positive)

Example: $|S| = 1000$ and want $\epsilon = 0.1$ then we should choose $m \approx 4800$ and $k \approx 3$

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

We assume that all hash functions $h \in \mathcal{H}$ are simple uniform and independent from each other.

$\frac{1}{m}$ probability that $h(x) = i$ and thus, $B[h(x)] = 1$ for $h \in \mathcal{H}$

$1 - \frac{1}{m}$ probability that $B[i]$ is not set to 1 by a given $h \in \mathcal{H}$

$(1 - \frac{1}{m})^k$ probability that $B[i]$ is not set to 1 by any of the k hash-functions $h \in \mathcal{H}$

From analysis: $\frac{1}{e} = \lim_{m \to \infty} \left(1 - \frac{1}{m}\right)^m \implies \left(1 - \frac{1}{m}\right)^k = \left(1 - \frac{1}{m}\right)^{\frac{km}{m}} = \left(\left(1 - \frac{1}{m}\right)^m\right)^{\frac{k}{m}}$ for large m $\approx \left(\frac{1}{e}\right)^{\frac{k}{m}} = e^{-\frac{k}{m}}$

$(1 - \frac{1}{m})^{kn} \approx e^{-\frac{kn}{m}}$ probability that $B[i]$ is not set to 1 by any $h \in \mathcal{H}$ after "adding" the n elements from S

$1 - (1 - \frac{1}{m})^{kn} \approx 1 - e^{-\frac{kn}{m}}$ probability that $B[i] = 1$ after "adding" the n elements from S

$\epsilon = \left(1 - e^{-\frac{kn}{m}}\right)^k$ probabiliy that all k entries $B[h_i(x)] = 1$ for x (this could cause the algorithm to erroneously claim that the element is in the set)

Observation: ϵ decreases with increasing m (more slots) and increases with with increasing n (more elements)

Assume we have $|S| = n$ and some m. What is an optimal number k of hash functions?: $\frac{d\epsilon}{dk} = 0 \implies k = \frac{m}{n} \ln(2)$

Assume we have $|S| = n$, a desired false-postive rate ϵ and an optimal k.

What is the required number m of bits?: replace k by $\frac{m}{n} \ln(2)$ in $\epsilon \implies m = \frac{-n \ln(\epsilon)}{\ln(2)^2}$ ($\epsilon < 1$, so $-n \ln(\epsilon)$ is positive)

Example: $|S| = 1000$ and want $\epsilon = 0.1$ then we should choose $m \approx 4800$ and $k \approx 3$

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

We assume that all hash functions $h \in \mathcal{H}$ are simple uniform and independent from each other.

$\frac{1}{m}$ probability that $h(x) = i$ and thus, $B[h(x)] = 1$ for $h \in \mathcal{H}$

$1 - \frac{1}{m}$ probability that $B[i]$ is not set to 1 by a given $h \in \mathcal{H}$

$(1 - \frac{1}{m})^k$ probability that $B[i]$ is not set to 1 by any of the k hash-functions $h \in \mathcal{H}$

From analysis: $\frac{1}{e} = \lim_{m \to \infty} \left(1 - \frac{1}{m}\right)^m \implies \left(1 - \frac{1}{m}\right)^k = \left(1 - \frac{1}{m}\right)^{\frac{km}{m}} = \left(\left(1 - \frac{1}{m}\right)^m\right)^{\frac{k}{m}}$ for large m $\approx \left(\frac{1}{e}\right)^{\frac{k}{m}} = e^{-\frac{k}{m}}$

$(1 - \frac{1}{m})^{kn} \approx e^{-\frac{kn}{m}}$ probability that $B[i]$ is not set to 1 by any $h \in \mathcal{H}$ after "adding" the n elements from S

$1 - (1 - \frac{1}{m})^{kn} \approx 1 - e^{-\frac{kn}{m}}$ probability that $B[i] = 1$ after "adding" the n elements from S

$\epsilon = \left(1 - e^{-\frac{kn}{m}}\right)^k$ probabiliy that all k entries $B[h_i(x)] = 1$ for x (this could cause the algorithm to erroneously claim that the element is in the set)

Observation: ϵ decreases with increasing m (more slots) and increases with with increasing n (more elements)

Assume we have $|S| = n$ and some m. What is an optimal number k of hash functions?: $\frac{d\epsilon}{dk} = 0 \implies k = \frac{m}{n} \ln(2)$

Assume we have $|S| = n$, a desired false-postive rate ϵ and an optimal k.

What is the required number m of bits?: replace k by $\frac{m}{n} \ln(2)$ in $\epsilon \implies m = \frac{-n \ln(\epsilon)}{\ln(2)^2}$ ($\epsilon < 1$, so $-n \ln(\epsilon)$ is positive)

Example: $|S| = 1000$ and want $\epsilon = 0.1$ then we should choose $m \approx 4800$ and $k \approx 3$

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

We assume that all hash functions $h \in \mathcal{H}$ are simple uniform and independent from each other.

$\frac{1}{m}$ probability that $h(x) = i$ and thus, $B[h(x)] = 1$ for $h \in \mathcal{H}$

$1 - \frac{1}{m}$ probability that $B[i]$ is not set to 1 by a given $h \in \mathcal{H}$

$(1 - \frac{1}{m})^k$ probability that $B[i]$ is not set to 1 by any of the k hash-functions $h \in \mathcal{H}$

From analysis: $\frac{1}{e} = \lim_{m \to \infty} \left(1 - \frac{1}{m}\right)^m \implies \left(1 - \frac{1}{m}\right)^k = \left(1 - \frac{1}{m}\right)^{\frac{km}{m}} = \left(\left(1 - \frac{1}{m}\right)^m\right)^{\frac{k}{m}}$ for large m $\approx \left(\frac{1}{e}\right)^{\frac{k}{m}} = e^{-\frac{k}{m}}$

$(1 - \frac{1}{m})^{kn} \approx e^{-\frac{kn}{m}}$ probability that $B[i]$ is not set to 1 by any $h \in \mathcal{H}$ after "adding" the n elements from S

$1 - (1 - \frac{1}{m})^{kn} \approx 1 - e^{-\frac{kn}{m}}$ probability that $B[i] = 1$ after "adding" the n elements from S

$\epsilon = \left(1 - e^{-\frac{kn}{m}}\right)^k$ probabiliy that all k entries $B[h_i(x)] = 1$ for x (this could cause the algorithm to erroneously claim that the element is in the set)

Observation: ϵ decreases with increasing m (more slots) and increases with with increasing n (more elements)

Assume we have $|S| = n$ and some m. What is an optimal number k of hash functions?: $\frac{d\epsilon}{dk} = 0 \implies k = \frac{m}{n} \ln(2)$

Assume we have $|S| = n$, a desired false-postive rate ϵ and an optimal k.

What is the required number m of bits?: replace k by $\frac{m}{n} \ln(2)$ in $\epsilon \implies m = \frac{-n \ln(\epsilon)}{\ln(2)^2}$ ($\epsilon < 1$, so $-n \ln(\epsilon)$ is positive)

Example: $|S| = 1000$ and want $\epsilon = 0.1$ then we should choose $m \approx 4800$ and $k \approx 3$

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

We assume that all hash functions $h \in \mathcal{H}$ are simple uniform and independent from each other.

$\frac{1}{m}$ probability that $h(x) = i$ and thus, $B[h(x)] = 1$ for $h \in \mathcal{H}$

$1 - \frac{1}{m}$ probability that $B[i]$ is not set to 1 by a given $h \in \mathcal{H}$

$(1 - \frac{1}{m})^k$ probability that $B[i]$ is not set to 1 by any of the k hash-functions $h \in \mathcal{H}$

From analysis: $\frac{1}{e} = \lim_{m \to \infty} \left(1 - \frac{1}{m}\right)^m \implies \left(1 - \frac{1}{m}\right)^k = \left(1 - \frac{1}{m}\right)^{\frac{km}{m}} = \left(\left(1 - \frac{1}{m}\right)^m\right)^{\frac{k}{m}}$ for large m $\approx \left(\frac{1}{e}\right)^{\frac{k}{m}} = e^{-\frac{k}{m}}$

$(1 - \frac{1}{m})^{kn} \approx e^{-\frac{kn}{m}}$ probability that $B[i]$ is not set to 1 by any $h \in \mathcal{H}$ after "adding" the n elements from S

$1 - (1 - \frac{1}{m})^{kn} \approx 1 - e^{-\frac{kn}{m}}$ probability that $B[i] = 1$ after "adding" the n elements from S

$\epsilon = \left(1 - e^{-\frac{kn}{m}}\right)^k$ probabiliy that all k entries $B[h_i(x)] = 1$ for x (this could cause the algorithm to erroneously claim that the element is in the set)

Observation: ϵ decreases with increasing m (more slots) and increases with with increasing n (more elements)

Assume we have $|S| = n$ and some m. What is an optimal number k of hash functions?: $\frac{d\epsilon}{dk} = 0 \implies k = \frac{m}{n} \ln(2)$

Assume we have $|S| = n$, a desired false-postive rate ϵ and an optimal k.

What is the required number m of bits?: replace k by $\frac{m}{n} \ln(2)$ in $\epsilon \implies m = \frac{-n \ln(\epsilon)}{\ln(2)^2}$ ($\epsilon < 1$, so $-n \ln(\epsilon)$ is positive)

Example: $|S| = 1000$ and want $\epsilon = 0.1$ then we should choose $m \approx 4800$ and $k \approx 3$

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

We assume that all hash functions $h \in \mathcal{H}$ are simple uniform and independent from each other.

$\frac{1}{m}$ probability that $h(x) = i$ and thus, $B[h(x)] = 1$ for $h \in \mathcal{H}$

$1 - \frac{1}{m}$ probability that $B[i]$ is not set to 1 by a given $h \in \mathcal{H}$

$(1 - \frac{1}{m})^k$ probability that $B[i]$ is not set to 1 by any of the k hash-functions $h \in \mathcal{H}$

From analysis: $\frac{1}{e} = \lim_{m \to \infty} \left(1 - \frac{1}{m}\right)^m \implies \left(1 - \frac{1}{m}\right)^k = \left(1 - \frac{1}{m}\right)^{\frac{km}{m}} = \left(\left(1 - \frac{1}{m}\right)^m\right)^{\frac{k}{m}}$ for large m $\approx \left(\frac{1}{e}\right)^{\frac{k}{m}} = e^{-\frac{k}{m}}$

$(1 - \frac{1}{m})^{kn} \approx e^{-\frac{kn}{m}}$ probability that $B[i]$ is not set to 1 by any $h \in \mathcal{H}$ after "adding" the n elements from S

$1 - (1 - \frac{1}{m})^{kn} \approx 1 - e^{-\frac{kn}{m}}$ probability that $B[i] = 1$ after "adding" the n elements from S

$\epsilon = \left(1 - e^{-\frac{kn}{m}}\right)^k$ probabiliy that all k entries $B[h_i(x)] = 1$ for x (this could cause the algorithm to erroneously claim that the element is in the set)

Observation: ϵ decreases with increasing m (more slots) and increases with with increasing n (more elements)

Assume we have $|S| = n$ and some m. What is an optimal number k of hash functions?: $\frac{d\epsilon}{dk} = 0 \implies k = \frac{m}{n} \ln(2)$

Assume we have $|S| = n$, a desired false-postive rate ϵ and an optimal k.

What is the required number m of bits?: replace k by $\frac{m}{n} \ln(2)$ in $\epsilon \implies m = \frac{-n \ln(\epsilon)}{\ln(2)^2}$ ($\epsilon < 1$, so $-n \ln(\epsilon)$ is positive)

Example: $|S| = 1000$ and want $\epsilon = 0.1$ then we should choose $m \approx 4800$ and $k \approx 3$

Part 4: Application of Hashing - Bloom Filter

A Bloom filter $(B, \mathcal{H})$ consists of

an array B of m bits, initially all set to 0 and

a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, \ldots, x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

We assume that all hash functions $h \in \mathcal{H}$ are simple uniform and independent from each other.

$\frac{1}{m}$ probability that $h(x) = i$ and thus, $B[h(x)] = 1$ for $h \in \mathcal{H}$

$1 - \frac{1}{m}$ probability that $B[i]$ is not set to 1 by a given $h \in \mathcal{H}$

$(1 - \frac{1}{m})^k$ probability that $B[i]$ is not set to 1 by any of the k hash-functions $h \in \mathcal{H}$

From analysis: $\frac{1}{e} = \lim_{m \to \infty} \left(1 - \frac{1}{m}\right)^m \implies \left(1 - \frac{1}{m}\right)^k = \left(1 - \frac{1}{m}\right)^{\frac{km}{m}} = \left(\left(1 - \frac{1}{m}\right)^m\right)^{\frac{k}{m}}$ for large m $\underset{\approx}{} \left(\frac{1}{e}\right)^{\frac{k}{m}} = e^{-\frac{k}{m}}$

$(1 - \frac{1}{m})^{kn} \approx e^{-\frac{kn}{m}}$ probability that $B[i]$ is not set to 1 by any $h \in \mathcal{H}$ after "adding" the n elements from S

$1 - (1 - \frac{1}{m})^{kn} \approx 1 - e^{-\frac{kn}{m}}$ probability that $B[i] = 1$ after "adding" the n elements from S

$\epsilon = \left(1 - e^{-\frac{kn}{m}}\right)^k$ probabiliy that all k entries $B[h_i(x)] = 1$ for x (this could cause the algorithm to erroneously claim that the element is in the set)

Observation: ϵ decreases with increasing m (more slots) and increases with with increasing n (more elements)

Assume we have $|S| = n$ and some m. What is an optimal number k of hash functions?: $\frac{d\epsilon}{dk} = 0 \implies k = \frac{m}{n} \ln(2)$

Assume we have $|S| = n$, a desired false-postive rate ϵ and an optimal k.

What is the required number m of bits?: replace k by $\frac{m}{n} \ln(2)$ in $\epsilon \implies m = \frac{-n \ln(\epsilon)}{\ln(2)^2}$ ($\epsilon < 1$, so $-n \ln(\epsilon)$ is positive)

Example: $|S| = 1000$ and want $\epsilon = 0.1$ then we should choose $m \approx 4800$ and $k \approx 3$