Hash tables

Or: why maths *matters*

Yuan Yuan

20th July, 2017

[AUT CSC]

AUT Computer Science Club

Outline

The dictionary problem

Definition

An *entry* is a pair of *key* and *value*.

The dictionary problem

Definition

An *entry* is a pair of *key* and *value*.

Definition

A *dictionary* is a set of entries, such that no two entries have the same key.

The dictionary problem

Definition
An *entry* is a pair of *key* and *value*.

Definition
A *dictionary* is a set of entries, such that no two entries have the same key.

Definition
The *dictionary problem* requires us to maintain a dictionary D, with the following operations:

The dictionary problem

Definition

An *entry* is a pair of *key* and *value*.

Definition

A *dictionary* is a set of entries, such that no two entries have the same key.

Definition

The *dictionary problem* requires us to maintain a dictionary D, with the following operations:

len(D): Return the number of entries in D

The dictionary problem

Definition
An *entry* is a pair of *key* and *value*.

Definition
A *dictionary* is a set of entries, such that no two entries have the same key.

Definition
The *dictionary problem* requires us to maintain a dictionary D, with the following operations:

- **len**(D): Return the number of entries in D
- **put**(D, k, v): Add a new entry to D with key k and value v; if an entry with key k already exists, replace its value with v

The dictionary problem

Definition

An *entry* is a pair of *key* and *value*.

Definition

A *dictionary* is a set of entries, such that no two entries have the same key.

Definition

The *dictionary problem* requires us to maintain a dictionary D, with the following operations:

- len(D): Return the number of entries in D
- put(D, k, v): Add a new entry to D with key k and value v; if an entry with key k already exists, replace its value with v
- get(D, k): Return the value of the entry whose key is k, or null if no such entry exists

Why we care about the dictionary problem

Why we care about the dictionary problem

- Databases (keys are identifiers, values are data)

Why we care about the dictionary problem

- Databases (keys are identifiers, values are data)
- A solution to this problem can implement almost any other data structure:

Why we care about the dictionary problem

- Databases (keys are identifiers, values are data)
- A solution to this problem can implement almost any other data structure:
  - Arrays and lists (keys are consecutive integers)

Why we care about the dictionary problem

- Databases (keys are identifiers, values are data)
- A solution to this problem can implement almost any other data structure:
  - Arrays and lists (keys are consecutive integers)
  - Sets (keys and values are the same as each other)

Why we care about the dictionary problem

- Databases (keys are identifiers, values are data)
- A solution to this problem can implement almost any other data structure:
  - Arrays and lists (keys are consecutive integers)
  - Sets (keys and values are the same as each other)
  - Trees, graphs, etc

Why we care about the dictionary problem

- Databases (keys are identifiers, values are data)
- A solution to this problem can implement almost any other data structure:
  - Arrays and lists (keys are consecutive integers)
  - Sets (keys and values are the same as each other)
  - Trees, graphs, etc
  - At least one programming language (Lua) does exactly this!

Why we care about the dictionary problem

- Databases (keys are identifiers, values are data)
- A solution to this problem can implement almost any other data structure:
  - Arrays and lists (keys are consecutive integers)
  - Sets (keys and values are the same as each other)
  - Trees, graphs, etc
  - At least one programming language (Lua) does exactly this!
- Allows us to give *names* to data, and use those names instead of the data itself (c.f. any programming language ever)

Why we care about the dictionary problem

- Databases (keys are identifiers, values are data)
- A solution to this problem can implement almost any other data structure:
  - Arrays and lists (keys are consecutive integers)
  - Sets (keys and values are the same as each other)
  - Trees, graphs, etc
  - At least one programming language (Lua) does exactly this!
- Allows us to give *names* to data, and use those names instead of the data itself (c.f. any programming language ever)

In short, *we want good solutions to the dictionary problem!*

Why we care about the dictionary problem

- Databases (keys are identifiers, values are data)
- A solution to this problem can implement almost any other data structure:
  - Arrays and lists (keys are consecutive integers)
  - Sets (keys and values are the same as each other)
  - Trees, graphs, etc
  - At least one programming language (Lua) does exactly this!
- Allows us to give *names* to data, and use those names instead of the data itself (c.f. any programming language ever)

In short, *we want good solutions to the dictionary problem!* We also don't want to impose too many constraints on the data beyond equality being defined (so ordering shouldn't matter, for example).

First attempt: Array or list

We can store our entries in an array or list.

First attempt: Array or list

We can store our entries in an array or list. We can remember the number of elements we store, and modify it on all operations in constant time, which makes len a $O(1)$ operation.

First attempt: Array or list

We can store our entries in an array or list. We can remember the number of elements we store, and modify it on all operations in constant time, which makes len a $O(1)$ operation.

For put, we scan the array or list left-to-right, comparing each entry's key with k. If we get a match, replace that entry's value with v; otherwise, add a new entry with key k and value v at the end.

First attempt: Array or list

We can store our entries in an array or list. We can remember the number of elements we store, and modify it on all operations in constant time, which makes len a $O(1)$ operation.

For put, we scan the array or list left-to-right, comparing each entry's key with k. If we get a match, replace that entry's value with v; otherwise, add a new entry with key k and value v at the end. This is $O(n)$, because we might have to scan the entire array or list.

First attempt: Array or list

We can store our entries in an array or list. We can remember the number of elements we store, and modify it on all operations in constant time, which makes len a $O(1)$ operation.

For put, we scan the array or list left-to-right, comparing each entry's key with k. If we get a match, replace that entry's value with v; otherwise, add a new entry with key k and value v at the end. This is $O(n)$, because we might have to scan the entire array or list.

We can do get similarly, except that we return the value if we find a match, or null otherwise.

First attempt: Array or list

We can store our entries in an array or list. We can remember the number of elements we store, and modify it on all operations in constant time, which makes len a $O(1)$ operation.

For put, we scan the array or list left-to-right, comparing each entry's key with k. If we get a match, replace that entry's value with v; otherwise, add a new entry with key k and value v at the end. This is $O(n)$, because we might have to scan the entire array or list.

We can do get similarly, except that we return the value if we find a match, or null otherwise. This is also $O(n)$.

First attempt: Array or list

We can store our entries in an array or list. We can remember the number of elements we store, and modify it on all operations in constant time, which makes len a $O(1)$ operation.

For put, we scan the array or list left-to-right, comparing each entry's key with k. If we get a match, replace that entry's value with v; otherwise, add a new entry with key k and value v at the end. This is $O(n)$, because we might have to scan the entire array or list.

We can do get similarly, except that we return the value if we find a match, or null otherwise. This is also $O(n)$.

Verdict: Not very good at all.

Can we do better?

Ideally, we want the following:

Can we do better?

Ideally, we want the following:

- get and put to be *asymptotically* fast (as close to $O(1)$ as possible)

Can we do better?

Ideally, we want the following:

- get and put to be *asymptotically* fast (as close to $O(1)$ as possible)
- All operations to be *practically* fast (i.e. no asymptotic ‘abuse’)

Can we do better?

Ideally, we want the following:

- get and put to be *asymptotically* fast (as close to $O(1)$ as possible)
- All operations to be *practically* fast (i.e. no asymptotic 'abuse')
- A data structure which is as simple as possible (because programming is hard enough already)

Can we do better?

Ideally, we want the following:

- get and put to be *asymptotically* fast (as close to $O(1)$ as possible)
- All operations to be *practically* fast (i.e. no asymptotic 'abuse')
- A data structure which is as simple as possible (because programming is hard enough already)

There *is* a structure which achieves all of the above.

Can we do better?

Ideally, we want the following:

- get and put to be *asymptotically* fast (as close to $O(1)$ as possible)
- All operations to be *practically* fast (i.e. no asymptotic 'abuse')
- A data structure which is as simple as possible (because programming is hard enough already)

There *is* a structure which achieves all of the above. First, we need to do a bit of preparation...

Preliminaries

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ be the set of *natural numbers*. We use $\mathbb{N}_k$ to represent the set $\{x \in \mathbb{N} \mid x$ can be represented using k bits $\}$, for $k \in \mathbb{N}$.

Preliminaries

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ be the set of *natural numbers*. We use $\mathbb{N}_k$ to represent the set $\{x \in \mathbb{N} \mid x$ can be represented using k bits $\}$, for $k \in \mathbb{N}$. For example, $\mathbb{N}_0 = \{\}$ (no number can be represented in 0 bits) and $\mathbb{N}_1 = \{0, 1\}$.

Preliminaries

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ be the set of *natural numbers*. We use $\mathbb{N}_k$ to represent the set $\{x \in \mathbb{N} \mid x$ can be represented using k bits $\}$, for $k \in \mathbb{N}$. For example, $\mathbb{N}_0 = \{\}$ (no number can be represented in 0 bits) and $\mathbb{N}_1 = \{0, 1\}$.

Let A, B be sets. A *function* $f : A \to B$ is a set of pairs such that for any $(x, y) \in f$, $x \in A, y \in B$, and for every $x \in A$, there exists exactly one $y \in B$ such that $(x, y) \in f$.

Preliminaries

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ be the set of *natural numbers*. We use $\mathbb{N}_k$ to represent the set $\{x \in \mathbb{N} \mid x$ can be represented using k bits $\}$, for $k \in \mathbb{N}$. For example, $\mathbb{N}_0 = \{\}$ (no number can be represented in 0 bits) and $\mathbb{N}_1 = \{0, 1\}$.

Let A, B be sets. A *function* $f : A \to B$ is a set of pairs such that for any $(x, y) \in f$, $x \in A, y \in B$, and for every $x \in A$, there exists exactly one $y \in B$ such that $(x, y) \in f$. In this case, A is the *domain* of f and B is the *codomain* of f.

Preliminaries

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ be the set of *natural numbers*. We use $\mathbb{N}_k$ to represent the set $\{x \in \mathbb{N} \mid x$ can be represented using k bits $\}$, for $k \in \mathbb{N}$. For example, $\mathbb{N}_0 = \{\}$ (no number can be represented in 0 bits) and $\mathbb{N}_1 = \{0, 1\}$.

Let A, B be sets. A *function* $f : A \to B$ is a set of pairs such that for any $(x, y) \in f$, $x \in A, y \in B$, and for every $x \in A$, there exists exactly one $y \in B$ such that $(x, y) \in f$. In this case, A is the *domain* of f and B is the *codomain* of f. For any $x \in A$, we denote by $f(x)$ (the *value of* f *at* x) the y such that $(x, y) \in f$.

Preliminaries

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ be the set of *natural numbers*. We use $\mathbb{N}_k$ to represent the set $\{x \in \mathbb{N} \mid x$ can be represented using k bits $\}$, for $k \in \mathbb{N}$. For example, $\mathbb{N}_0 = \{\}$ (no number can be represented in 0 bits) and $\mathbb{N}_1 = \{0, 1\}$.

Let A, B be sets. A *function* $f : A \to B$ is a set of pairs such that for any $(x, y) \in f$, $x \in A, y \in B$, and for every $x \in A$, there exists exactly one $y \in B$ such that $(x, y) \in f$. In this case, A is the *domain* of f and B is the *codomain* of f. For any $x \in A$, we denote by $f(x)$ (the *value of* f *at* x) the y such that $(x, y) \in f$.

An example function $f : \{a, b, c\} \to \{1, 2, 3\}$ is $\{(a, 1), (b, 1), (c, 2)\}$.

Preliminaries

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ be the set of *natural numbers*. We use $\mathbb{N}_k$ to represent the set $\{x \in \mathbb{N} \mid x$ can be represented using k bits $\}$, for $k \in \mathbb{N}$. For example, $\mathbb{N}_0 = \{\}$ (no number can be represented in 0 bits) and $\mathbb{N}_1 = \{0, 1\}$.

Let A, B be sets. A *function* $f : A \to B$ is a set of pairs such that for any $(x, y) \in f$, $x \in A, y \in B$, and for every $x \in A$, there exists exactly one $y \in B$ such that $(x, y) \in f$. In this case, A is the *domain* of f and B is the *codomain* of f. For any $x \in A$, we denote by $f(x)$ (the *value of* f *at* x) the y such that $(x, y) \in f$.

An example function $f : \{a, b, c\} \to \{1, 2, 3\}$ is $\{(a, 1), (b, 1), (c, 2)\}$. In this case, the domain of f is $\{a, b, c\}$, the codomain of f is $\{1, 2, 3\}$ and $f(a) = 1$.

More about functions

Let $f : A \to B$ be a function. We say f is *one-to-one* if, for any $x, y \in A$, if $x \neq y$, then $f(x) \neq f(y)$.

More about functions

Let $f : A \to B$ be a function. We say f is *one-to-one* if, for any $x, y \in A$, if $x \neq y$, then $f(x) \neq f(y)$. An example one-to-one function $f : \{a, b, c\} \to \{1, 2, 3\}$ would be $\{(a, 1), (b, 2), (c, 3)\}$.

More about functions

Let $f : A \to B$ be a function. We say f is *one-to-one* if, for any $x, y \in A$, if $x \neq y$, then $f(x) \neq f(y)$. An example one-to-one function $f : \{a, b, c\} \to \{1, 2, 3\}$ would be $\{(a, 1), (b, 2), (c, 3)\}$.

Lemma

Let A be an infinite set and B be a finite set. There is no function $f : A \to B$ such that f is one-to-one.

More about functions

Let $f : A \to B$ be a function. We say f is *one-to-one* if, for any $x, y \in A$, if $x \neq y$, then $f(x) \neq f(y)$. An example one-to-one function $f : \{a, b, c\} \to \{1, 2, 3\}$ would be $\{(a, 1), (b, 2), (c, 3)\}$.

Lemma

Let A be an infinite set and B be a finite set. There is no function $f : A \to B$ such that f is one-to-one.

Definition

Let A be a set, and let $k \in \mathbb{N}$. A *hash function for A* is some $f : A \to \mathbb{N}_k$. We call the value of $f(x)$ the *hash* of x.

More about functions

Let $f : A \to B$ be a function. We say f is *one-to-one* if, for any $x, y \in A$, if $x \neq y$, then $f(x) \neq f(y)$. An example one-to-one function $f : \{a, b, c\} \to \{1, 2, 3\}$ would be $\{(a, 1), (b, 2), (c, 3)\}$.

Lemma

Let A be an infinite set and B be a finite set. There is no function $f : A \to B$ such that f is one-to-one.

Definition

Let A be a set, and let $k \in \mathbb{N}$. A *hash function for A* is some $f : A \to \mathbb{N}_k$. We call the value of $f(x)$ the *hash* of x.

You can think of a hash function as producing a *fixed-length summary* of its input.

The hash table

Let K be a set of keys, V be a set of values, and $k \in \mathbb{N}$.

The hash table

Let K be a set of keys, V be a set of values, and $k \in \mathbb{N}$. A *hash table* H for K, V consists of:

The hash table

Let K be a set of keys, V be a set of values, and $k \in \mathbb{N}$. A *hash table* H for K, V consists of:

- A hash function $H.\mathsf{hash} : K \to \mathbb{N}_k$

The hash table

Let K be a set of keys, V be a set of values, and $k \in \mathbb{N}$. A *hash table* H for K, V consists of:

- A hash function $H.\mathsf{hash} : K \to \mathbb{N}_k$
- An array $H.\mathsf{buckets}$ of *buckets*, capable of storing elements of V. Additionally, $\mathsf{len}(H.\mathsf{buckets}) \leq 2^k$, initially full of nulls.

The hash table

Let K be a set of keys, V be a set of values, and $k \in \mathbb{N}$. A *hash table* H for K, V consists of:

- A hash function $H.\mathsf{hash} : K \to \mathbb{N}_k$
- An array $H.\mathsf{buckets}$ of *buckets*, capable of storing elements of V. Additionally, $\mathsf{len}(H.\mathsf{buckets}) \leq 2^k$, initially full of nulls.

As $H.\mathsf{buckets}$ is an array, we store and update its length similarly to our original solution, giving us len in $O(1)$ time.

The hash table

Let K be a set of keys, V be a set of values, and $k \in \mathbb{N}$. A *hash table* H for K, V consists of:

- A hash function $H.\mathsf{hash} : K \to \mathbb{N}_k$
- An array $H.\mathsf{buckets}$ of *buckets*, capable of storing elements of V. Additionally, $\mathsf{len}(H.\mathsf{buckets}) \leq 2^k$, initially full of nulls.

As $H.\mathsf{buckets}$ is an array, we store and update its length similarly to our original solution, giving us len in $O(1)$ time.

put for a hash table

As hash produces a number, we can convert the hash of any key x into a valid index for buckets by taking hash(x) modulo len(buckets).

put for a hash table

As hash produces a number, we can convert the hash of any key x into a valid index for buckets by taking hash(x) modulo len(buckets). If there is nothing at that index, we simply store our value there; otherwise, we replace the value there with the one we are given.

put for a hash table

As hash produces a number, we can convert the hash of any key x into a valid index for buckets by taking hash(x) modulo len(buckets). If there is nothing at that index, we simply store our value there; otherwise, we replace the value there with the one we are given.

function put(H, k, v)
  $i \leftarrow$ H.hash(k) % len(H.buckets)
  if H.buckets$[i] =$ null **then**
    len$(H) \leftarrow$ len(H) $+1$
  H.buckets$[i] \leftarrow v$

put for a hash table

As hash produces a number, we can convert the hash of any key x into a valid index for buckets by taking hash(x) modulo len(buckets). If there is nothing at that index, we simply store our value there; otherwise, we replace the value there with the one we are given.

function put(H, k, v)
  $i \leftarrow$ H.hash(k) % len(H.buckets)
  if H.buckets$[i] =$ null **then**
    len$(H) \leftarrow$ len(H) $+1$
  H.buckets$[i] \leftarrow v$

This only requires a constant amount of time, plus however long it takes to call hash.

get for a hash table

This is very similar to put.

get for a hash table

This is very similar to put.

function get(H, k)
  $i \leftarrow$ H.hash(k) % len$(H$.buckets$)$
  return H.buckets$[i]$

get for a hash table

This is very similar to put.

function get(H, k)
  $i \leftarrow$ H.hash(k) % len$(H$.buckets$)$
  return H.buckets$[i]$

This also requires a constant amount of time, plus the call time of hash.

get for a hash table

This is very similar to put.

function get(H, k)
  $i \leftarrow$ H.hash(k) % len$(H$.buckets$)$
  return H.buckets$[i]$

This also requires a constant amount of time, plus the call time of hash.

This looks great!

get for a hash table

This is very similar to put.

function get(H, k)
  $i \leftarrow$ H.hash(k) % len(H.buckets)
  return H.buckets[i]

This also requires a constant amount of time, plus the call time of hash.

This looks great! However, this is too simplistic, and won't work in practice.

Problems with our hash table

Our design assumes two things:

Problems with our hash table

Our design assumes two things:

- We will never try to put more entries into H than len(H.buckets)

Problems with our hash table

Our design assumes two things:

- We will never try to put more entries into H than len(H.buckets)
- Given two different keys, H.hash will produce two different hashes

Problems with our hash table

Our design assumes two things:

- We will never try to put more entries into H than $\mathsf{len}(H.\mathsf{buckets})$
- Given two different keys, $H.\mathsf{hash}$ will produce two different hashes

Both of these are false in general:

Problems with our hash table

Our design assumes two things:

- We will never try to put more entries into H than len(H.buckets)
- Given two different keys, H.hash will produce two different hashes

Both of these are false in general: the former obviously so, the latter because of our prior lemma, and the fact that most interesting sets of keys (e.g. all strings) are infinite.

Problems with our hash table

Our design assumes two things:

- We will never try to put more entries into H than len(H.buckets)
- Given two different keys, H.hash will produce two different hashes

Both of these are false in general: the former obviously so, the latter because of our prior lemma, and the fact that most interesting sets of keys (e.g. all strings) are infinite. Thus, what will inevitably happen at some point is that our put procedure will assign the same index to two different keys.

Problems with our hash table

Our design assumes two things:

- We will never try to put more entries into H than len(H.buckets)
- Given two different keys, H.hash will produce two different hashes

Both of these are false in general: the former obviously so, the latter because of our prior lemma, and the fact that most interesting sets of keys (e.g. all strings) are infinite. Thus, what will inevitably happen at some point is that our put procedure will assign the same index to two different keys. This is called a *collision*, and it really ruins our day (and design).

Problems with our hash table

Our design assumes two things:

- We will never try to put more entries into H than len(H.buckets)
- Given two different keys, H.hash will produce two different hashes

Both of these are false in general: the former obviously so, the latter because of our prior lemma, and the fact that most interesting sets of keys (e.g. all strings) are infinite. Thus, what will inevitably happen at some point is that our put procedure will assign the same index to two different keys. This is called a *collision*, and it really ruins our day (and design).

Collisions are inevitable — based on this, we have to design with them in mind.

Problems with our hash table

Our design assumes two things:

- We will never try to put more entries into H than len(H.buckets)
- Given two different keys, H.hash will produce two different hashes

Both of these are false in general: the former obviously so, the latter because of our prior lemma, and the fact that most interesting sets of keys (e.g. all strings) are infinite. Thus, what will inevitably happen at some point is that our put procedure will assign the same index to two different keys. This is called a *collision*, and it really ruins our day (and design).

Collisions are inevitable — based on this, we have to design with them in mind. Luckily, our design is easy to fix to take collisions into account.

Hash chaining

Instead of buckets storing values, each index stores a linked list of entries, which start out empty (a *bucket list*).

Hash chaining

Instead of buckets storing values, each index stores a linked list of entries, which start out empty (a *bucket list*). After we calculate an index, we work with the list at that position (just like with our first attempt), for both of our operations.

Hash chaining

Instead of buckets storing values, each index stores a linked list of entries, which start out empty (a *bucket list*). After we calculate an index, we work with the list at that position (just like with our first attempt), for both of our operations.

As long as our bucket lists fill up roughly evenly, the time required for get and put will be roughly $O(\frac{n}{m})$, where m is the size of our bucket array.

Hash chaining

Instead of buckets storing values, each index stores a linked list of entries, which start out empty (a *bucket list*). After we calculate an index, we work with the list at that position (just like with our first attempt), for both of our operations.

As long as our bucket lists fill up roughly evenly, the time required for get and put will be roughly $O(\frac{n}{m})$, where m is the size of our bucket array. This is pretty good in practice, as long as we don't try to crowd too many entries into our hash table.

Hash chaining

Instead of buckets storing values, each index stores a linked list of entries, which start out empty (a *bucket list*). After we calculate an index, we work with the list at that position (just like with our first attempt), for both of our operations.

As long as our bucket lists fill up roughly evenly, the time required for get and put will be roughly $O(\frac{n}{m})$, where m is the size of our bucket array. This is pretty good in practice, as long as we don't try to crowd too many entries into our hash table.

How can we be sure that our bucket lists will fill evenly?

Hash chaining

Instead of **buckets** storing values, each index stores a linked list of entries, which start out empty (a *bucket list*). After we calculate an index, we work with the list at that position (just like with our first attempt), for both of our operations.

As long as our bucket lists fill up roughly evenly, the time required for **get** and **put** will be roughly $O(\frac{n}{m})$, where m is the size of our bucket array. This is pretty good in practice, as long as we don't try to crowd too many entries into our hash table.

How can we be sure that our bucket lists will fill evenly? A function which hashes *everything* to 1 is a hash function, and most certainly *won't* cause our bucket lists to fill evenly!

Good hash functions

Good hash functions

In order to make sure our buckets fill evenly, we need more from our hash functions.

Good hash functions

In order to make sure our buckets fill evenly, we need more from our hash functions. Specifically, for a hash function $f : A \to \mathbb{N}_k$, we would like f to have one (or both!) of the following properties:

Good hash functions

In order to make sure our buckets fill evenly, we need more from our hash functions. Specifically, for a hash function $f : A \to \mathbb{N}_k$, we would like f to have one (or both!) of the following properties:

Definition

f is *uniform* if, given a random $x \in A$ and any specific $y \in \mathbb{N}_k$, there is a $\frac{1}{2^k}$ probability that $f(x) = y$.

Good hash functions

In order to make sure our buckets fill evenly, we need more from our hash functions. Specifically, for a hash function $f : A \to \mathbb{N}_k$, we would like f to have one (or both!) of the following properties:

Definition

f is *uniform* if, given a random $x \in A$ and any specific $y \in \mathbb{N}_k$, there is a $\frac{1}{2^k}$ probability that $f(x) = y$.

Definition

f *exhibits the avalanche effect* if, for any $x, y \in A$ which differ by 1 bit, $f(x), f(y)$ will differ by $\frac{k}{2}$ bits.

Good hash functions

In order to make sure our buckets fill evenly, we need more from our hash functions. Specifically, for a hash function $f : A \to \mathbb{N}_k$, we would like f to have one (or both!) of the following properties:

Definition

f is *uniform* if, given a random $x \in A$ and any specific $y \in \mathbb{N}_k$, there is a $\frac{1}{2^k}$ probability that $f(x) = y$.

Definition

f *exhibits the avalanche effect* if, for any $x, y \in A$ which differ by 1 bit, $f(x), f(y)$ will differ by $\frac{k}{2}$ bits.

These guarantee that, when the number of entries gets large, there is a high probability that they will be distributed evenly over all bucket lists.

Limitations of hash tables

Limitations of hash tables

- For small hash tables, the cost of hashing overwhelms everything else (making them slower than array or list-based approaches)

Limitations of hash tables

- For small hash tables, the cost of hashing overwhelms everything else (making them slower than array or list-based approaches)
- Finding a good hash function can be difficult, especially for variable-length keys (but easier now with cryptographic hashing and universal hash functions)

Limitations of hash tables

- For small hash tables, the cost of hashing overwhelms everything else (making them slower than array or list-based approaches)
- Finding a good hash function can be difficult, especially for variable-length keys (but easier now with cryptographic hashing and universal hash functions)
- No order for entries (or rather, no *specific* order)

Limitations of hash tables

- For small hash tables, the cost of hashing overwhelms everything else (making them slower than array or list-based approaches)
- Finding a good hash function can be difficult, especially for variable-length keys (but easier now with cryptographic hashing and universal hash functions)
- No order for entries (or rather, no *specific* order)
- Cache-unfriendly due to bucket lists (but alternative approaches exist which don't require them)

Limitations of hash tables

- For small hash tables, the cost of hashing overwhelms everything else (making them slower than array or list-based approaches)
- Finding a good hash function can be difficult, especially for variable-length keys (but easier now with cryptographic hashing and universal hash functions)
- No order for entries (or rather, no *specific* order)
- Cache-unfriendly due to bucket lists (but alternative approaches exist which don't require them)

Overall, hash tables work best for dictionaries with large numbers of entries and simple fixed-length keys. This is not necessarily a problem, as a lot of real data fits these requirements.

Limitations of hash tables

- For small hash tables, the cost of hashing overwhelms everything else (making them slower than array or list-based approaches)
- Finding a good hash function can be difficult, especially for variable-length keys (but easier now with cryptographic hashing and universal hash functions)
- No order for entries (or rather, no *specific* order)
- Cache-unfriendly due to bucket lists (but alternative approaches exist which don't require them)

Overall, hash tables work best for dictionaries with large numbers of entries and simple fixed-length keys. This is not necessarily a problem, as a lot of real data fits these requirements. However, as always, *know your data and your tradeoffs*!

Questions?