INF421, Lecture 5
Hashing

Leo Liberti

LIX, École Polytechnique, France

Course

- **Objective**: to teach you some data structures and associated algorithms
- **Evaluation**: TP noté en salle info le 16 septembre, Contrôle à la fin. Note: $\max(CC, \frac{3}{4}CC + \frac{1}{4}TP)$
- **Organization**: fri 26/8, 2/9, 9/9, 16/9, 23/9, 30/9, 7/10, 14/10, 21/10, amphi 1030-12 (Arago), TD 1330-1530, 1545-1745 (SI31,32,33,34)
- **Books**:
  1. Ph. Baptiste & L. Maranget, *Programmation et Algorithmique*, Ecole Polytechnique (Polycopié), 2006
  2. G. Dowek, *Les principes des langages de programmation*, Editions de l'X, 2008
  3. D. Knuth, *The Art of Computer Programming*, Addison-Wesley, 1997
  4. K. Mehlhorn & P. Sanders, *Algorithms and Data Structures*, Springer, 2008
- **Website**: `www.enseignement.polytechnique.fr/informatique/INF421`
- **Contact**: `liberti@lix.polytechnique.fr` (e-mail subject: INF421)

Lecture summary

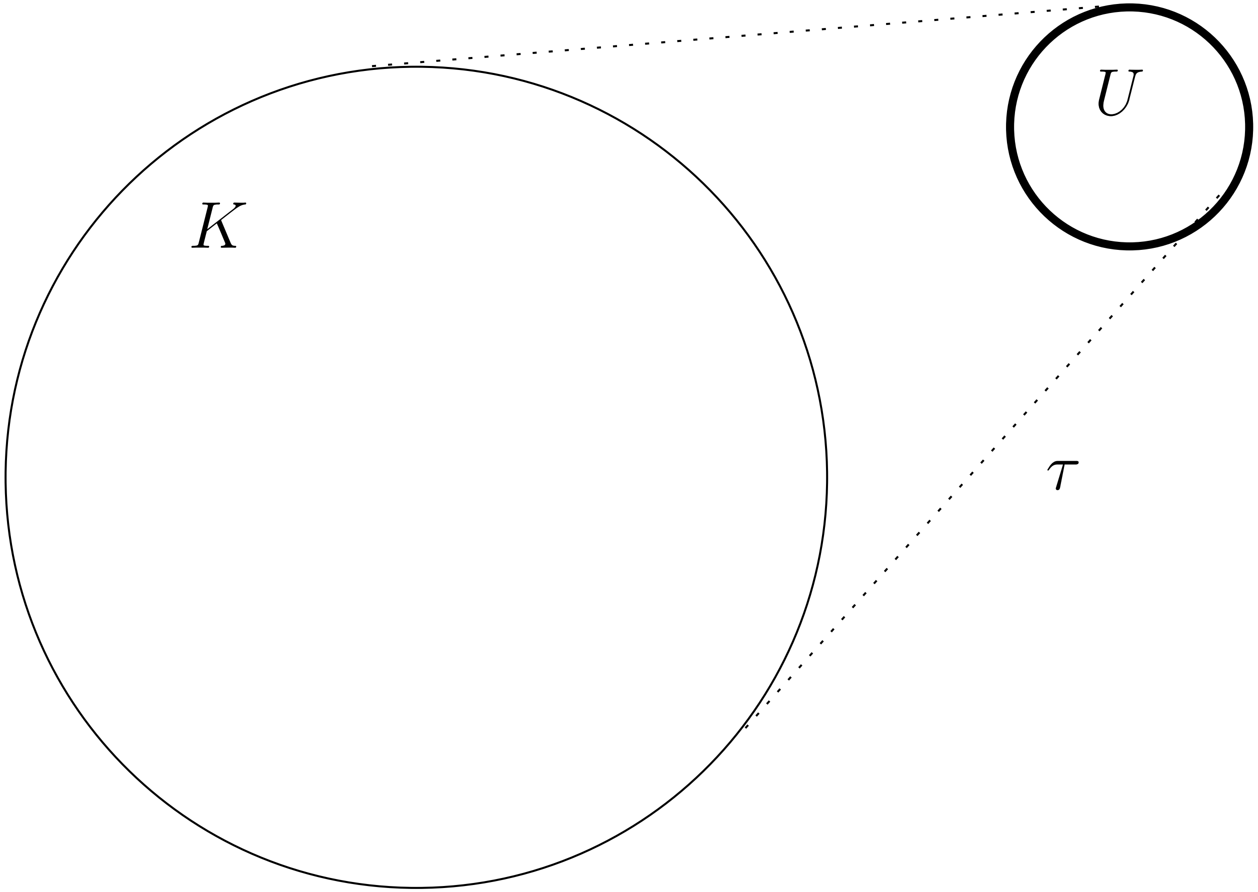
- Searching
- Tables
- Hashing
- Collisions
- Implementation

Why?

- Address book:
  1. each page corresponds to a character
  2. page with character k contains all names beginning with k
  3. easy to search: immediately find the correct page, then scan the list, which is at most as long as the page
- Can we use a list of pairs (name,telephone)?
  Slow to search
- Can we use a table name $\rightarrow$ telephone?
  Difficult to extend its size

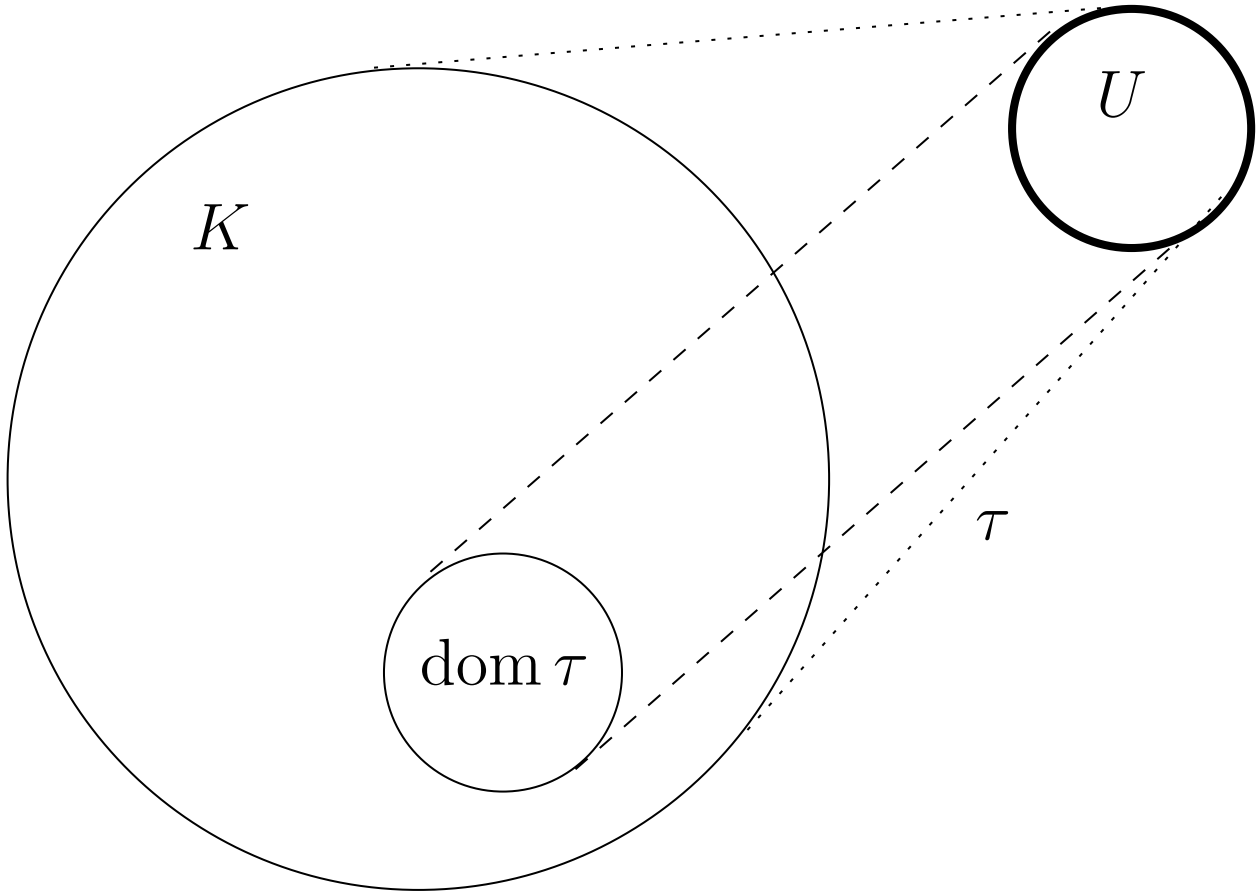
Hash tables are the appropriate data structures

The minimal knowledge



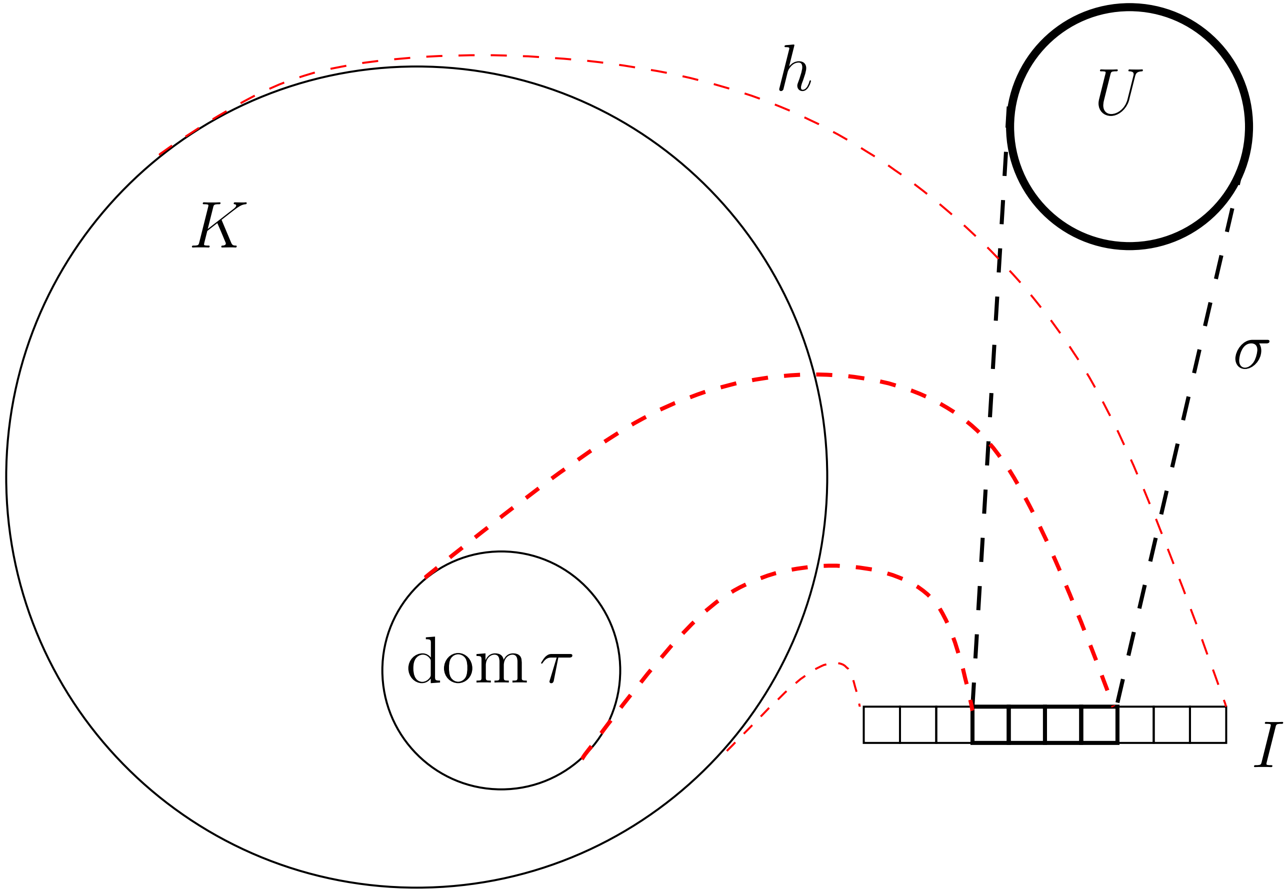
- K a very large set of keys; U: a set of objects; $\tau : K \to U$: a table
- Assume K too large to store, but $\mathrm{dom}\,\tau$ is small
- Find a function $h : K \to I$ with $I = \{0, 1, \ldots, p-1\}$ and $|I| \approx |U|$, then store $u = \tau(k)$ at $\sigma(i)$ where $i = h(k)$

The minimal knowledge



- K a very large set of keys; U: a set of objects; $\tau : K \to U$: a table
- Assume K too large to store, but $\mathrm{dom}\,\tau$ is small
- Find a function $h : K \to I$ with $I = \{0, 1, \ldots, p-1\}$ and $|I| \approx |U|$, then store $u = \tau(k)$ at $\sigma(i)$ where $i = h(k)$

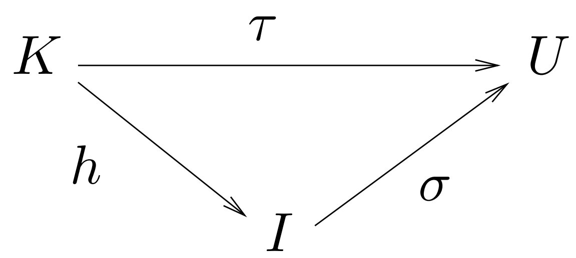
The minimal knowledge



- K a very large set of keys; U: a set of objects; $\tau : K \to U$: a table
- Assume K too large to store, but $\operatorname{dom}\tau$ is small
- Find a function $h : K \to I$ with $I = \{0, 1, \ldots, p-1\}$ and $|I| \approx |U|$, then store $u = \tau(k)$ in array element $\sigma(i)$ where $i = h(k)$

Minimal technical knowledge

- K =*keys*, U =*records*
- Associate some keys with records
- Get an injective *table function* $\tau : K \to U$, with $\mathrm{dom}\,\tau \subsetneq K$
- Given a key $k \in K$, determine whether $k \in \mathrm{dom}\,\tau$
- If τ was an array, $\tau(k) = u$ if $k \in \mathrm{dom}\,\tau$ or $\perp$ if $k \notin \mathrm{dom}\,\tau$: $O(1)$
- However, $|K|$ too large to be in an array
- Use *hash table* $\sigma : I \to U$ on an *index* set I with $|I| \approx |\mathrm{dom}\,\tau| \ll |K|$
- Need a *hash function* $h : K \to I$ to map keys to indices
- Store record u in σ at position $h(k)$: get $\sigma(h(k)) = u$
- Maps σ, h, τ must be such that $\tau = \sigma \circ h$:



- If this holds, then $k \in \mathrm{dom}\,\tau \Leftrightarrow h(k) \in I$
- Look $h(k)$ up in array σ in $O(1)$
- Scheme only works if h is injective, otherwise get *collisions*
- One way to address collisions is to let $\sigma(i) = \{u \in U \mid h(\tau^{-1}(u)) = i\}$

Searching

The set element problem

SET ELEMENT PROBLEM (SEP). Given a set U, a set $V \subseteq U$ and an element $u \in U$, determine whether $u \in V$

- Fundamental problem in computer science (and mathematics)
- Also known as the *searching problem*, the *find problem*, in some context the *feasibility problem*, and no doubt in several other ways too
- For computer implementations, one often also requires the index of u in V if the answer to the SEP is YES

Sequential search

- If the set V is stored as a sequence $(v_1, v_2, \ldots, v_n)$, can perform **sequential search**:

  for $i \leq n$ **do**
    if $v_i = u$ **then**
      return i; `// found`
    end if
  end for
  return $n+1$; `// not found`

- If seq. search returns $n+1$, $u \notin V$, otherwise $u \in V$ and the return value is the *index of u in V*
- Worst-case complexity: $O(n)$

Eliminate a test

1: Let $v_{n+1} = u$
2: **for** $i \in \mathbb{N}$ **do**
3: **if** $v_i = u$ **then**
4: **return** i;
5: **end if**
6: **end for**

Gets rid of test $i \leq n$ at each iteration

This “trick” already seen in Lecture 1

Self-organizing search

- Each time $u \in V$ at position i, swap $u = v_i$ and v_1:

```
Let v_{n+1} = u
for i ∈ ℕ do
  if v_i = u then
    if i ≤ n then
      swap(v, 1, i);
      return 1;
    else
      return n + 1;
    end if
  end if
end for
```

- Elements that are sought for most often take fewer iterations to be found
- Still $O(n)$ worst-case complexity

Binary search

- Assume $V = (v_1, \ldots, v_n)$ is ordered ($i < j \to v_i \leq v_j$)

$i = 1$;
$j = n$;
while $i \leq j$ **do**
  $\ell = \lfloor \frac{i+j}{2} \rfloor$;
  if $u < v_\ell$ **then**
    $j = \ell - 1$;
  else if $u > v_\ell$ **then**
    $i = \ell + 1$;
  else
    return ℓ; `// found`
  end if
end while
return $n + 1$; `// not found`

- Worst-case complexity: $O(\log n)$ (by INF311)

Tables

The data structure

- A **table** generalizes the concept of array: it maps a *key* $k \in K$ to a *record* $u \in U$

The data structure

- A **table** generalizes the concept of array: it maps a *key* $k \in K$ to a *record* $u \in U$
- We assume that each record $u \in U$ is given with its corresponding key

The data structure

- A **table** generalizes the concept of array: it maps a *key* $k \in K$ to a *record* $u \in U$
- We assume that each record $u \in U$ is given with its corresponding key
- Examples: telephone directory, nameservers, databases

The data structure

- A **table** generalizes the concept of array: it maps a *key* $k \in K$ to a *record* $u \in U$
- We assume that each record $u \in U$ is given with its corresponding key
- Examples: telephone directory, nameservers, databases
- Mathematically, tables are used to model injective maps $\tau : K \to U$

The data structure

- A **table** generalizes the concept of array: it maps a *key* $k \in K$ to a *record* $u \in U$
- We assume that each record $u \in U$ is given with its corresponding key
- Examples: telephone directory, nameservers, databases
- Mathematically, tables are used to model injective maps $\tau : K \to U$
- If $u \in U$ is associated to two different keys $k, k' \in K$, the data for u is duplicated in memory, so that τ remains injective

The data structure

- A **table** generalizes the concept of array: it maps a *key* $k \in K$ to a *record* $u \in U$
- We assume that each record $u \in U$ is given with its corresponding key
- Examples: telephone directory, nameservers, databases
- Mathematically, tables are used to model injective maps $\tau : K \to U$
- If $u \in U$ is associated to two different keys $k, k' \in K$, the data for u is duplicated in memory, so that τ remains injective
- Basic operations:

The data structure

- A **table** generalizes the concept of array: it maps a *key* $k \in K$ to a *record* $u \in U$
- We assume that each record $u \in U$ is given with its corresponding key
- Examples: telephone directory, nameservers, databases
- Mathematically, tables are used to model injective maps $\tau : K \to U$
- If $u \in U$ is associated to two different keys $k, k' \in K$, the data for u is duplicated in memory, so that τ remains injective
- Basic operations:
  - $\texttt{insert}(u)$: insert a new record u in the table

The data structure

- A **table** generalizes the concept of array: it maps a *key* $k \in K$ to a *record* $u \in U$
- We assume that each record $u \in U$ is given with its corresponding key
- Examples: telephone directory, nameservers, databases
- Mathematically, tables are used to model injective maps $\tau : K \to U$
- If $u \in U$ is associated to two different keys $k, k' \in K$, the data for u is duplicated in memory, so that τ remains injective
- Basic operations:
  - `insert`(u): insert a new record u in the table
  - `find`(k): determine if a given key k appears in the table

The data structure

- A **table** generalizes the concept of array: it maps a *key* $k \in K$ to a *record* $u \in U$
- We assume that each record $u \in U$ is given with its corresponding key
- Examples: telephone directory, nameservers, databases
- Mathematically, tables are used to model injective maps $\tau : K \to U$
- If $u \in U$ is associated to two different keys $k, k' \in K$, the data for u is duplicated in memory, so that τ remains injective
- Basic operations:
  - `insert`(u): insert a new record u in the table
  - `find`(k): determine if a given key k appears in the table
  - `remove`(k): delete a record with key k from the table

Searching tables

- Searching a table for a given key is an extremely important problem (also known as *table look-up problem*)
- Needs to be solved as efficiently as possible
- E.g. in Lecture 2, I stated that we could find whether an arc was in a certain table (in BFS) in $O(1)$
- However:
  - Sequential search: $O(n)$
  - Binary search: $O(\log n)$
- How do we look a key up in $O(1)$?

Motivating examples

Telephone directory

- τ maps the set K of all personal names to a set U of telephone numbers
- Clearly, not all names are mapped, but only those of existing people having telephones: $|\operatorname{dom}\tau| \ll |K|$
- Two trivial solutions:
  - a table $\tau : K \to U$ *(which lists all possible names, and* $\tau(k) = \perp$ *if* k *is not the name of an existing person with a telephone)*
  - a table $\tau' : \operatorname{dom}\tau \to U$ which only lists existing people with telephones
- τ: $O(1)$ `find` but $O(|K|)$ space (impractical)
- τ': $O(|\operatorname{dom}\tau|)$ `find` if K is unsorted, $O(\log|\operatorname{dom}\tau|)$ if sorted (we want $O(1)$)

Comparing Java objects

- An object could occupy a fairly large chunk of memory (e.g. a whole database table)
- Sometimes we wish to test whether two objects `a`, `b` in memory are equal
- Requires a byte comparison: $O(\max(|\mathrm{a}|, |\mathrm{b}|))$: inefficient
- How do we do it in $O(1)$?

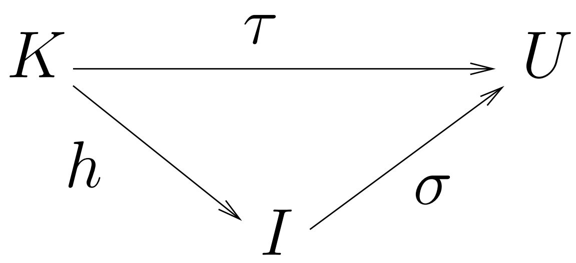
Back to tables

Tables in arrays

- Usually, $|K|$ is monstrously large
  - **nameserver**: K =set of fully qualified domain names
  - **database**: K =set of all possible entries from an index field
- Trivial implementation — array of size $|K|$: **impossible**
- Notice that $|\mathrm{dom}\,\tau|$ is usually *much smaller* than $|K|$
- Consider a map $h : K \to I$ where I is a set of *indices* (which could be integers, or memory addresses), and a **hash table** $\sigma : I \to U$
- Then, if $u = \tau(k)$, u is stored in σ at index $h(k)$
- Look-up in σ rather than τ

Clarification I

- We're concerned with *three sets*:
  - U is the set of records
  - K is the set of keys
  - I is the set of indices
- ... and *three maps*:
  - $\tau : K \to U$: given a $k \in K$, is it in $\mathrm{dom}\,\tau$?
  - $h : K \to I$: maps keys to a smaller set of indices
  - $\sigma : I \to U$: table actually used for storing records



Clarification II

- If K were small, we could store $\tau : K \to U$ in an array with as many components as $|K|$

Clarification II

- If K were small, we could store $\tau : K \to U$ in an array with as many components as $|K|$
- This array would be initialized to $\perp$ (=not found) if $k \notin \operatorname{dom} \tau$, and to the record $u = \tau(k)$ otherwise (=found)

Clarification II

- If K were small, we could store $\tau : K \to U$ in an array with as many components as $|K|$
- This array would be initialized to $\perp$ (=not found) if $k \notin \mathrm{dom}\,\tau$, and to the record $u = \tau(k)$ otherwise (=found)
- Then the question $k \in \mathrm{dom}\,\tau?$ could be answered in $O(1)$ by simply looking up the value at position k in this array

Clarification II

- If K were small, we could store $\tau : K \to U$ in an array with as many components as $|K|$
- This array would be initialized to $\perp$ (=not found) if $k \notin \operatorname{dom}\tau$, and to the record $u = \tau(k)$ otherwise (=found)
- Then the question $k \in \operatorname{dom}\tau?$ could be answered in $O(1)$ by simply looking up the value at position k in this array
- But $|K|$ is too large, so we map $\operatorname{dom}\tau$ to a set I of indices with $|I| \approx |\operatorname{dom}\tau|$, using a map $h : K \to I$, and store records in hash table $\sigma : I \to U$

Clarification II

- If K were small, we could store $\tau : K \to U$ in an array with as many components as $|K|$
- This array would be initialized to $\perp$ (=not found) if $k \notin \mathrm{dom}\,\tau$, and to the record $u = \tau(k)$ otherwise (=found)
- Then the question $k \in \mathrm{dom}\,\tau$? could be answered in $O(1)$ by simply looking up the value at position k in this array
- But $|K|$ is too large, so we map $\mathrm{dom}\,\tau$ to a set I of indices with $|I| \approx |\mathrm{dom}\,\tau|$, using a map $h : K \to I$, and store records in hash table $\sigma : I \to U$
- We use the $O(1)$ table look-up method on the array σ

Clarification II

- If K were small, we could store $\tau : K \to U$ in an array with as many components as $|K|$
- This array would be initialized to $\perp$ (=not found) if $k \notin \operatorname{dom} \tau$, and to the record $u = \tau(k)$ otherwise (=found)
- Then the question $k \in \operatorname{dom} \tau$? could be answered in $O(1)$ by simply looking up the value at position k in this array
- But $|K|$ is too large, so we map $\operatorname{dom} \tau$ to a set I of indices with $|I| \approx |\operatorname{dom} \tau|$, using a map $h : K \to I$, and store records in hash table $\sigma : I \to U$
- We use the $O(1)$ table look-up method on the array σ
- **The map h apparently reduces $O(|K|)$ to $O(1)$**

Clarification II

- If K were small, we could store $\tau : K \to U$ in an array with as many components as $|K|$
- This array would be initialized to $\perp$ (=not found) if $k \notin \mathrm{dom}\,\tau$, and to the record $u = \tau(k)$ otherwise (=found)
- Then the question $k \in \mathrm{dom}\,\tau$? could be answered in $O(1)$ by simply looking up the value at position k in this array
- But $|K|$ is too large, so we map $\mathrm{dom}\,\tau$ to a set I of indices with $|I| \approx |\mathrm{dom}\,\tau|$, using a map $h : K \to I$, and store records in hash table $\sigma : I \to U$
- We use the $O(1)$ table look-up method on the array σ
- **The map h apparently reduces $O(|K|)$ to $O(1)$**

Where am I cheating?

Clarification III

- Since the size of K is the problem, why didn't I simply index σ by $\mathrm{dom}\,\tau$? Why introducing the function h at all?

Clarification III

- Since the size of K is the problem, why didn't I simply index σ by $\mathrm{dom}\,\tau$? Why introducing the function h at all?
- Consider that $\mathrm{dom}\,\tau \subsetneq K$, but $\mathrm{dom}\,\tau$ might well contain small as well as large keys in K

Clarification III

- Since the size of K is the problem, why didn't I simply index σ by $\mathrm{dom}\,\tau$? Why introducing the function h at all?
- Consider that $\mathrm{dom}\,\tau \subsetneq K$, but $\mathrm{dom}\,\tau$ might well contain small as well as large keys in K
- In order to find an array element in $O(1)$, the array components must be stored **contiguously**

Clarification III

- Since the size of K is the problem, why didn't I simply index σ by $\operatorname{dom}\tau$? Why introducing the function h at all?
- Consider that $\operatorname{dom}\tau \subsetneq K$, but $\operatorname{dom}\tau$ might well contain small as well as large keys in K
- In order to find an array element in $O(1)$, the array components must be stored **contiguously**
- If $K = \{0, 1, \ldots, 10^{50} - 1\}$ and $\operatorname{dom}\tau = \{0, 10^{50} - 1\}$, the fact that $|\operatorname{dom}\tau| = 2$ is useless: we must index the array over the whole of K

Clarification III

- Since the size of K is the problem, why didn't I simply index σ by $\mathrm{dom}\,\tau$? Why introducing the function h at all?
- Consider that $\mathrm{dom}\,\tau \subsetneq K$, but $\mathrm{dom}\,\tau$ might well contain small as well as large keys in K
- In order to find an array element in $O(1)$, the array components must be stored **contiguously**
- If $K = \{0, 1, \ldots, 10^{50} - 1\}$ and $\mathrm{dom}\,\tau = \{0, 10^{50} - 1\}$, the fact that $|\mathrm{dom}\,\tau| = 2$ is useless: we must index the array over the whole of K
- However, by defining $I = \{0, 1\}$ and $h(k) = k \bmod 2$, we can really use an array of length 2

A very special case

- $K = I = \{0x0, 0x1, 0x2, 0x3, 0x4\}$ (set of addresses)
- $\mathrm{dom}\,\tau = \{0x0, 0x3, 0x4\}$

$I = K$	U
0x0	1
0x1	0
0x2	0
0x3	1
0x4	1

- Let $h : K \to I$ be the identity function
- To find whether $k \in K$ is in $\mathrm{dom}\,\tau$, look at $\sigma(h(k))$: $k \in \mathrm{dom}\,\tau$ iff it is 1 (answer in time $O(1)$)

A very special case

- $K = I = \{\text{0x0}, \text{0x1}, \text{0x2}, \text{0x3}, \text{0x4}\}$ (set of addresses)
- $\text{dom}\,\tau = \{\text{0x0}, \text{0x3}, \text{0x4}\}$

$I = K$	U
0x0	**1**
0x1	0
0x2	0
0x3	1
0x4	1

- Let $h : K \to I$ be the identity function
- To find whether $k \in K$ is in $\text{dom}\,\tau$, look at $\sigma(h(k))$: $k \in \text{dom}\,\tau$ iff it is 1 (answer in time $O(1)$)

$u = \text{0x0} \in \text{dom}\,\tau$

A very special case

- $K = I = \{$0x0, 0x1, 0x2, 0x3, 0x4$\}$ (set of addresses)
- $\mathrm{dom}\,\tau = \{$0x0, 0x3, 0x4$\}$

$I = K$	U
0x0	1
0x1	**0**
0x2	0
0x3	1
0x4	1

- Let $h : K \to I$ be the identity function
- To find whether $k \in K$ is in $\mathrm{dom}\,\tau$, look at $\sigma(h(k))$: $k \in \mathrm{dom}\,\tau$ iff it is 1 (answer in time $O(1)$)

$u =$ 0x1 $\notin \mathrm{dom}\,\tau$

A very special case

- $K = I = \{\text{0x0}, \text{0x1}, \text{0x2}, \text{0x3}, \text{0x4}\}$ (set of addresses)
- $\text{dom}\,\tau = \{\text{0x0}, \text{0x3}, \text{0x4}\}$

$I = K$	U
0x0	1
0x1	**0**
0x2	0
0x3	1
0x4	1

- Let $h : K \to I$ be the identity function
- To find whether $k \in K$ is in $\text{dom}\,\tau$, look at $\sigma(h(k))$: $k \in \text{dom}\,\tau$ iff it is 1 (answer in time $O(1)$)

How far can we generalize this concept?

Address book again

- In an address book, K is the set of all names
- I is the set of all (capital) letters
- h maps a surname to its initial letter
- Assuming all our names start with a different letter, we're in business
- Otherwise, we have collisions (see later)

Hashing

Main idea

- The main insight of these examples is that

  the index $h(k)$ is obtained from the key k

Main idea

- The main insight of these examples is that

  the index $h(k)$ *is obtained from the key* k

- **Idea**

  Construct each index from the corresponding key

Main idea

- The main insight of these examples is that

  the index $h(k)$ is obtained from the key k

- **Idea**

  Construct each index from the corresponding key

- For example, if the key is the string `Leo`, we could take the ASCII codes of all characters and sum them together

Main idea

- The main insight of these examples is that

  the index $h(k)$ is obtained from the key k

- **Idea**

  Construct each index from the corresponding key

- For example, if the key is the string `Leo`, we could take the ASCII codes of all characters and sum them together

- This gives $h(\texttt{Leo}) = 76 + 101 + 111 = 288$: we store `Leo` in the table σ at position 288

Main idea

- The main insight of these examples is that

  the index $h(k)$ is obtained from the key k

- **Idea**

  Construct each index from the corresponding key

- For example, if the key is the string `Leo`, we could take the ASCII codes of all characters and sum them together
- This gives $h(\texttt{Leo}) = 76 + 101 + 111 = 288$: we store `Leo` in the table σ at position 288
- If we use the same rule for every key, we have an implementation of h

Hash functions

- I wrote *“we could sum the ASCII codes of the characters”*

Hash functions

- I wrote *"we could sum the ASCII codes of the characters"*
- Sounds a little vague... why sum? why not multiply? why not raise them to a prime power, sum them, then reduce the sum modulo a prime?

Hash functions

- I wrote *"we could sum the ASCII codes of the characters"*
- Sounds a little vague...why sum? why not multiply? why not raise them to a prime power, sum them, then reduce the sum modulo a prime?
- Let $\mathcal{H}$ be the set of all programs h which:
  - take keys in K as input
  - output indices in I as output
  - run fast

Hash functions

- I wrote *"we could sum the ASCII codes of the characters"*
- Sounds a little vague…why sum? why not multiply? why not raise them to a prime power, sum them, then reduce the sum modulo a prime?
- Let $\mathcal{H}$ be the set of all programs h which:
  - take keys in K as input
  - output indices in I as output
  - run fast
- Each $h \in \mathcal{H}$ defines a **hash function** $h : K \to I$

Hash functions

- I wrote *"we could sum the ASCII codes of the characters"*
- Sounds a little vague…why sum? why not multiply? why not raise them to a prime power, sum them, then reduce the sum modulo a prime?
- Let $\mathcal{H}$ be the set of all programs h which:
  - take keys in K as input
  - output indices in I as output
  - run fast
- Each $h \in \mathcal{H}$ defines a **hash function** $h : K \to I$

We initialize σ to the "not found" value $\perp$

We store $u = \tau(k)$ in σ at position $h(k)$

Hash speed

- How fast should h be in order to define a useful hash function?
- We assume the maximum size ℓ of the memory taken to store an element of K to be **constant with respect to** $|\operatorname{dom} \tau|$
- In other words: **keys have the same size ℓ independently of how many we store in τ**
- We require h to run in time proportional to some function of ℓ
- This means h runs in $O(1)$ with respect to $|\operatorname{dom} \tau|$

Example with names

- Consider the set of names $\{\text{Tim}, \text{John}, \text{Leo}\}$
- We store names as `char` arrays using ASCII codes:

Tim	54	69	6D
Jon	4A	6F	6E
Leo	4C	65	6F

- We now form the map h as follows:

$$h(\text{Tim}) = \text{0x0054696D}$$
$$h(\text{John}) = \text{0x004A6F6E}$$
$$h(\text{Leo}) = \text{0x004C656F}$$

- For $k \in K$ we can store $\tau(k)$ in σ at the address $h(k)$

Requires large hash table, but computing h is $O(1)$

A general hash function

- All computer-representable data can be written as byte sequences of various lengths

A general hash function

- All computer-representable data can be written as byte sequences of various lengths
- Each byte holds an integer in the range $0, \ldots, 255$

A general hash function

- All computer-representable data can be written as byte sequences of various lengths
- Each byte holds an integer in the range $0, \ldots, 255$
- Hence, we can assume K to be a set of m finite integer sequences (with m large)

A general hash function

- All computer-representable data can be written as byte sequences of various lengths
- Each byte holds an integer in the range $0, \ldots, 255$
- Hence, we can assume K to be a set of m finite integer sequences (with m large)
- We also assume that all sequences in $k = (k_1, \ldots, k_\ell) \in K$ have the same length ℓ (if not, pad shorter sequences with initial zeroes)

A general hash function

- All computer-representable data can be written as byte sequences of various lengths
- Each byte holds an integer in the range $0, \ldots, 255$
- Hence, we can assume K to be a set of m finite integer sequences (with m large)
- We also assume that all sequences in $k = (k_1, \ldots, k_\ell) \in K$ have the same length ℓ (if not, pad shorter sequences with initial zeroes)
- p: smallest prime $\geq |U|$, let $I = \{0, \ldots, p-1\}$

A general hash function

- All computer-representable data can be written as byte sequences of various lengths
- Each byte holds an integer in the range $0, \ldots, 255$
- Hence, we can assume K to be a set of m finite integer sequences (with m large)
- We also assume that all sequences in $k = (k_1, \ldots, k_\ell) \in K$ have the same length ℓ (if not, pad shorter sequences with initial zeroes)
- p: smallest prime $\geq |U|$, let $I = \{0, \ldots, p-1\}$
- For each $a \in I^\ell$, the following is a hash function:

$$h_a(k) = ak \bmod p \tag{6}$$

  - ak is the scalar product $\sum_{j \leq \ell} a_j k_j$ of a and k
  - h_a maps K to I,
  - computing h_a is $O(\ell)$ as required, and very fast in practice

Some hash functions

- Up to now, we've seen four types of hash functions
  - The identity $h(k) = k$ (first example with $K = I$)
  - The projection $h(k) = k_j$ for some $j \leq |k|$ (address book)
  - The base change $h((u_1, \ldots, u_n)) = \sum_{j \leq n} u_j b^{j-1}$, where b is "large enough" (table of first names)
  - The scalar product by $a \in \mathbb{N}^n$ modulo p:

$$h_a((k_1, \ldots, k_n)) = \left(\sum_{j \leq n} a_j k_j \right) \bmod p$$

- Identity and base change are **not** often used:
- Projection and scalar product modulo p are used in practice

Collisions

What can go wrong

- Consider the scalar product modulo p with $a = (2, 3, 5)$ and $p = 7$
- Let $k = (1, 1, 1)$ and $k' = (3, 2, 1)$
- We have:

$$h_a(k) = 2 + 3 + 5 \bmod 7 = 3 = 6 + 6 + 5 \bmod 7 = h_a(k')$$

- **How can we store both k and k' at index 3 in σ?**
- This is called a *collision*
- It happens when hash functions are not injective

Table injectivity

- Recall we store $u = \tau(k)$ at $\sigma(h(k))$

Table injectivity

- Recall we store $u = \tau(k)$ at $\sigma(h(k))$
- $\Rightarrow \quad \forall k \in \mathrm{dom}\,\tau \ (\ \tau(k) = \sigma(h(k))\ )$

Table injectivity

- Recall we store $u = \tau(k)$ at $\sigma(h(k))$
- $\Rightarrow \quad \forall k \in \mathrm{dom}\,\tau \; (\; \tau(k) = \sigma(h(k)) \;)$
- Since τ is injective, $k \neq k' \Rightarrow \tau(k) \neq \tau(k')$

Table injectivity

- Recall we store $u = \tau(k)$ at $\sigma(h(k))$
- $\Rightarrow \quad \forall k \in \mathrm{dom}\,\tau \ (\ \tau(k) = \sigma(h(k)) \ )$
- Since τ is injective, $k \neq k' \Rightarrow \tau(k) \neq \tau(k')$
- Let $u = \tau(k)$ and $u' = \tau(k')$

Table injectivity

- Recall we store $u = \tau(k)$ at $\sigma(h(k))$
- $\Rightarrow \quad \forall k \in \mathrm{dom}\,\tau \ (\ \tau(k) = \sigma(h(k))\ )$
- Since τ is injective, $k \neq k' \Rightarrow \tau(k) \neq \tau(k')$
- Let $u = \tau(k)$ and $u' = \tau(k')$
- If h fails to be injective on $\{k, k'\}$, there is an $i \in I$ such that $h(k) = i = h(k')$

Table injectivity

- Recall we store $u = \tau(k)$ at $\sigma(h(k))$
- $\Rightarrow \quad \forall k \in \mathrm{dom}\,\tau \ (\ \tau(k) = \sigma(h(k)) \ )$
- Since τ is injective, $k \neq k' \Rightarrow \tau(k) \neq \tau(k')$
- Let $u = \tau(k)$ and $u' = \tau(k')$
- If h fails to be injective on $\{k, k'\}$, there is an $i \in I$ such that $h(k) = i = h(k')$
- This means that both u, u' should both be stored at $\sigma(i)$

Table injectivity

- Recall we store $u = \tau(k)$ at $\sigma(h(k))$
- $\Rightarrow \quad \forall k \in \mathrm{dom}\,\tau \; (\; \tau(k) = \sigma(h(k)) \;)$
- Since τ is injective, $k \neq k' \Rightarrow \tau(k) \neq \tau(k')$
- Let $u = \tau(k)$ and $u' = \tau(k')$
- If h fails to be injective on $\{k, k'\}$, there is an $i \in I$ such that $h(k) = i = h(k')$
- This means that both u, u' should both be stored at $\sigma(i)$
- Impossible as long as the hash table σ is implemented as an array

Hashes do not inject

- A sad fact of life: most hash functions are *not* injective

Hashes do not inject

- A sad fact of life: most hash functions are *not* injective
- There are $|I|^{|K|}$ functions from $K \to I$, all could potentially be hash functions

Hashes do not inject

- A sad fact of life: most hash functions are *not* injective
- There are $|I|^{|K|}$ functions from $K \to I$, all could potentially be hash functions
- If $|I| < |K|$, none is injective

Hashes do not inject

- A sad fact of life: most hash functions are *not* injective
- There are $|I|^{|K|}$ functions from $K \to I$, all could potentially be hash functions
- If $|I| < |K|$, none is injective
- If $|I| \geq |K|$:
  - there are $|I|$ ways to choose the image of the first element of K,
  - $|I| - 1$ ways to choose the second, and so on
  - get $\binom{|I|}{|K|}$ injective functions $K \to I$

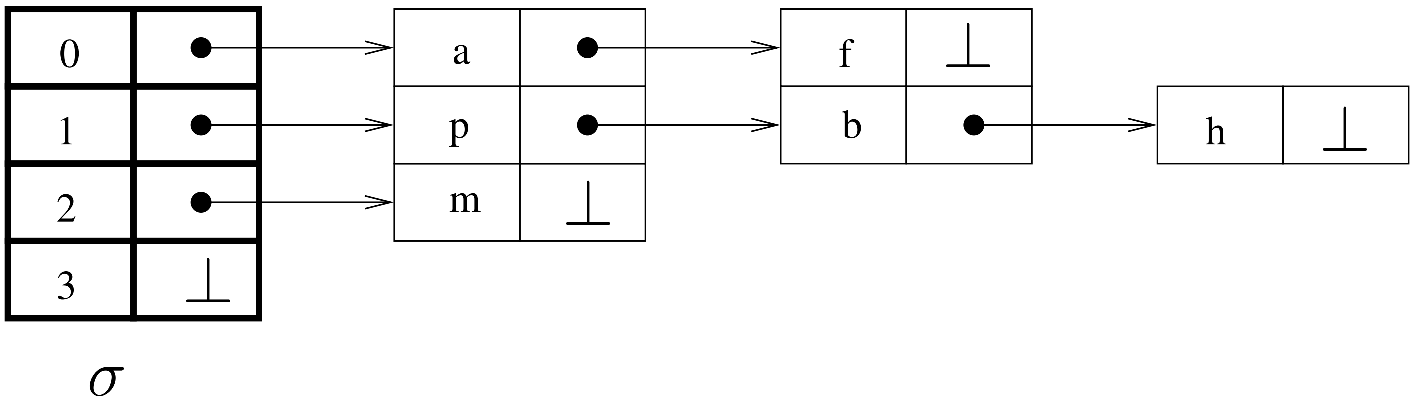
Hashes do not inject

- A sad fact of life: most hash functions are *not* injective
- There are $|I|^{|K|}$ functions from $K \to I$, all could potentially be hash functions
- If $|I| < |K|$, none is injective
- If $|I| \geq |K|$:
  - there are $|I|$ ways to choose the image of the first element of K,
  - $|I| - 1$ ways to choose the second, and so on
  - get $\begin{pmatrix} |I| \\ |K| \end{pmatrix}$ injective functions $K \to I$
- If $|K| = 31$ and $|I| = 41$, there are around 10^{50} functions, only 10^{43} of which are injective (*one in ten million: rare*)

Thanks to D. Knuth for this calculation

Resolving collisions: chaining

- The array σ maps I to the *power set* of U
- I.e. $\sigma(i)$ stores the set of all $u \in U$ having keys which all hash to i
- In this context, such sets are also called *buckets*
- We can implement these sets as lists



$h(\mathtt{a}) = h(\mathtt{f}) = 0$

$h(\mathtt{p}) = h(\mathtt{b}) = h(\mathtt{h}) = 1$

$h(\mathtt{m}) = 2$

$\perp$ stands for the `null` reference

Implementation

Implementation: `find`

- `find`(k) $\{$

  $i = h(k)$

  if $\sigma(i) = \perp$ **then**

  return $\perp$; `// not found`

  else

  return $\sigma(i)$.`find`(u);

  end if

  $\}$

Note: the list's `find` returns a reference to list element containing u or $\perp$ if u is not in the list

Implementation: `insert`

- $\texttt{insert}(u)$ {

  $\sigma(h(\tau^{-1}(u))).\texttt{add}(u)$; `// uses the list's add`

  }

- $\texttt{remove}(k)$ {

  $t = \texttt{find}(k)$;
  if $t \neq \bot$ **then**
  $\sigma(h(k)).\texttt{remove}(t)$; `//` t `points to the list node with` u
  end if

  }

Complexity

- All the table methods employ the underlying list methods
- In particular, `find` is $O(\texttt{list.size}())$ and is used by all three methods
- However, if there are no collisions, the lists all have size 1, so methods are $O(1)$ as required
- Choose h so that the probability of collisions is low
- Collisions are “evenly spread” over the keys
- Aim to have short lists of similar size

Can show that avg. case complexity is $O(1 + \alpha)$

where $\alpha = |\operatorname{ran} \tau|/|I|$

Hash function implementation

- Above code assumes h to be available
- Designing good hash functions is very difficult
- So difficult, in fact, as to require several clock cycles
- This computer work, as any useful work, is worth some money

  `http://bitcoin.org/`

- Moreover, this work prevents spam

  `http://hashcash.org/`

- Java provides a ready-made method `hashCode()` which applies to all classes
- However, an ad-hoc implementation is often needed

Testing Java object equality

Perfect hash

- Let $\mathtt{a}$, $\mathtt{b}$ are Java (or C++) objects of a class $\mathtt{C}$
- Suppose they have a large size when stored in memory
- Suppose also you want to test whether $\mathtt{a}=\mathtt{b}$
- Byte-comparison takes $O(\max(|\mathtt{a}|, |\mathtt{b}|))$ (too long)
- Consider a hash function $h : K \to I$ where $K = \mathtt{C}$ and I are integers modulo a given prime p
- Since we can never allow $h(\mathtt{a}) = h(\mathtt{b})$ whenever $\mathtt{a} \neq \mathtt{b}$, h must be injective
- An injective hash function is also known as a *perfect hash function*
- A perfect hash function is *minimal* (MPHF) if $|\operatorname{dom} \tau| = |I|$
- MPHFs can be found in time $O(|\operatorname{dom} \tau|)$ [Czech, Majewski, 1992]
- This requires $\operatorname{dom} \tau$ to be known in advance: impractical for transient memory objects

Or else...

- Use normal hash functions
- Design them so that the chances of a collision are as low as possible
- Only test for *difference* rather than equality
- If $h(\mathtt{a}) \neq h(\mathtt{b})$, then certainly $\mathtt{a} \neq \mathtt{b}$
- If $h(\mathtt{a}) = h(\mathtt{b})$, it may be because $\mathtt{a} = \mathtt{b}$ or because of a collision
- Only perform lengthy byte comparisons whenever $h(\mathtt{a}) = h(\mathtt{b})$
- Remark that there are $|I|$ pairs $i, j \in I$ such that $i = j$ but $\frac{|I|(|I|-1)}{2}$ unordered pairs with $i \neq j$
- Probability that $h(\mathtt{a}) = h(\mathtt{b})$: $\frac{2}{|I|-1}$
- Most comparisons are expected to take $O(1)$, $O(\frac{1}{|I|})$ are expected to take $O(\max(|\mathtt{a}|, |\mathtt{b}|))$

Appendix

The obvious won't work

Why $h(k)$ should be computed in function of k

- Let K =all words and $\mathrm{dom}\,\tau = \{\text{Leo}, \text{Jon}, \text{Tim}, \text{Joe}, \ldots\}$
- Why not let $h(\text{Leo}) = 1$, $h(\text{Jon}) = 2$ and so on?
- Store "Joe" in $\sigma(h(\text{Joe})) = \sigma(4)$
- Find if "Joe" is in $\mathrm{dom}\,\tau$: see if $\sigma(4) = \perp$ or not
- **Trouble**: for a key $k \in \mathrm{dom}\,\tau$, how do you find the value of $h(k)$?
- Have to search the sequence of pairs $((\text{Leo}, 1), (\text{Jon}, 2), \ldots)$
- $O(n)$ if sequence unsorted, $O(\log n)$ if sorted

Process fails to be $O(1)$

Open addressing

- Often, $\mathrm{dom}\,\sigma \subsetneq I$
- $\Rightarrow$ some hash values in I are never used
- $\Rightarrow$ hash table has unused entries
- Can use them to store colliding keys
- If $h(k) = h(k') = i$ with $k \neq k'$, store $\tau(k) = u$ at $\sigma(i)$ and $\tau(k') = u'$ at first unused hash table entry after the i-th one

Open addressing: collision

$\vdots$	
$i-1$	
i	
$i+1$	w
$i+2$	
$\vdots$	

Open addressing: collision

$\vdots$	
$i-1$	
i	u
$i+1$	w
$i+2$	
$\vdots$	

Open addressing: collision

$\vdots$	
$i-1$	
i	u
$i+1$	w
$i+2$	u'
$\vdots$	

Open addressing: `insert`

- `insert`(u)

  $i = h(\tau^{-1}(u))$;
  $c = 0$;
  while $c < |\sigma| \wedge \sigma(i) \neq \perp$ **do**
    $i \leftarrow (i+1) \bmod |\sigma|$;
    $c \leftarrow c + 1$;
  end while
  if $c \geq |\sigma|$ **then**
    error: *hash table full*;
  else
    $\sigma(i) = u$;
  end if

Open addressing: `find`

- `find`(k)

  $i = h(k)$;
  $c = 0$;
  while $c < |\sigma| \wedge \tau^{-1}(\sigma(i)) \neq k$ **do**
    $i \leftarrow (i+1) \bmod |\sigma|$;
    $c \leftarrow c + 1$;
  end while
  if $c \geq |\sigma|$ **then**
    return $\perp$;
  else
    return $\sigma(i)$;
  end if

`remove` is not easy to implement

An implementation secret

- In the pseudocodes, I've been referring to $\tau(k)$ and $\tau^{-1}(u)$ as if they'd be easy to compute
- That is *mathematical notation*: I simply meant "the record associated with the key k" and "the key associated with the record u"
- In an implementation, record pairs $\langle k, u \rangle$ in the hash table
- Then $\sigma : I \to K \times U$
- Pseudocode adapts perfectly: τ, τ^{-1} simply mean "the other element of the pair"

End of Lecture 5