CSE202
Design and Analysis of Algorithms

Week 7 — Randomized Algorithms 2: Hashing & Applications

Recall: Hash Functions (CSE101)

Def. A hash function *h* maps objects from a given universe (e.g., integers, floats, strings, files,…) to integers in a prescribed range.

Desirable properties:

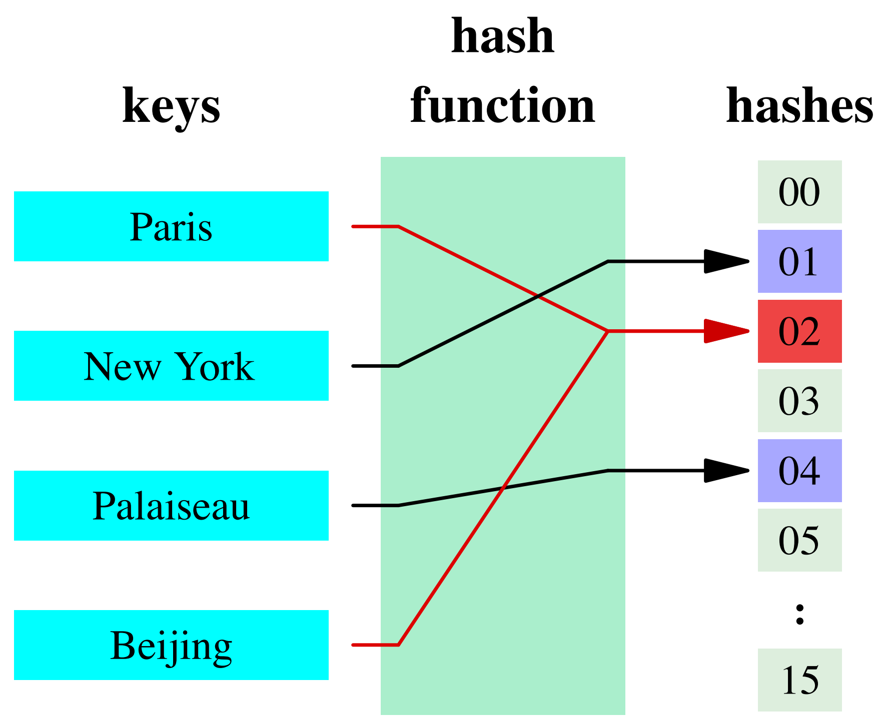
. the computation of *h* should be fast
. when $a \neq b$, $h(a)=h(b)$ should be unlikely;
. in cryptographic applications,
no information on a should be accessible from $h(a)$.

```
>>> import crypt
>>> crypt.crypt("My password")
'kDYaUpMaRUhrA'
>>> crypt.crypt("Not my password")
'Ny3sudvU07yTg'
```

Passwords don't need to be stored

Applications

Hash tables: this lecture.



Fingerprinting:

check that a file has not been corrupted/modified;
detect duplicate data;
avoid backup of unchanged portions of a file;
search pattern in a text (next tutorial).

I. Hash Functions

Python Hash Codes (Simplified)

Python's built-in `hash` returns a 64-bit integer

Integers: $a \bmod p := 2^{61} - 1$ (prime)

Prime numbers neutralize regularity in the keys

Ex: social security number, IP address,…

Rational & Floating-Point Numbers: same reduction

$x = y \Rightarrow$ hash(x) = hash(y) even if different types

Tuples can be hashed as $(a_0, a_1, a_2) \mapsto (a_2 x + a_1)x + a_0 \bmod p$

($x < p$ and large)

Strings can be viewed as tuples of characters

For a range $0,\ldots,m-1$, use hash(a) $\bmod m$.

Worst-Case and Randomization

Analogous to randomization in QuickSort

Worst-case: all keys hashed to the same value

(used in "hash flooding" denial-of-service attacks)

Randomization: make x session dependent

Protects against worst-cases/ malicious adversaries

Assumptions on Hash Functions

$$h : k \in U \mapsto h(k) \in \{0,\ldots,m-1\}$$

Complexity: $h(k)$ computed in $O(1)$ operations.

Uniformity Assumption:

$$k_1 \neq k_2 \Longrightarrow \mathbb{P}\big(h(k_1) = h(k_2)\big) = \frac{1}{m}.$$

Reasonable in practice.

Application. A Monte-Carlo equality test using $\log_2 m$ bits and failing with probability $\leq 1/m$.

7 bits $\rightarrow \mathbb{P}(\text{err}) < 1\,\%$

32 bits $\rightarrow \mathbb{P}(\text{err}) < 10^{-9}$

II. Hash Tables

Recall: Dictionary (CSE101)

An abstract data type with the following operations:

Create
Insert(key,value)
Contains(key)
Get(key)
Delete(key)

Examples:

dictionary: (word,definition)
phone book: (name,phone number)
internet:(domain name, IP address)
compiler:(variable,memory address)
…

also, in many implementations:

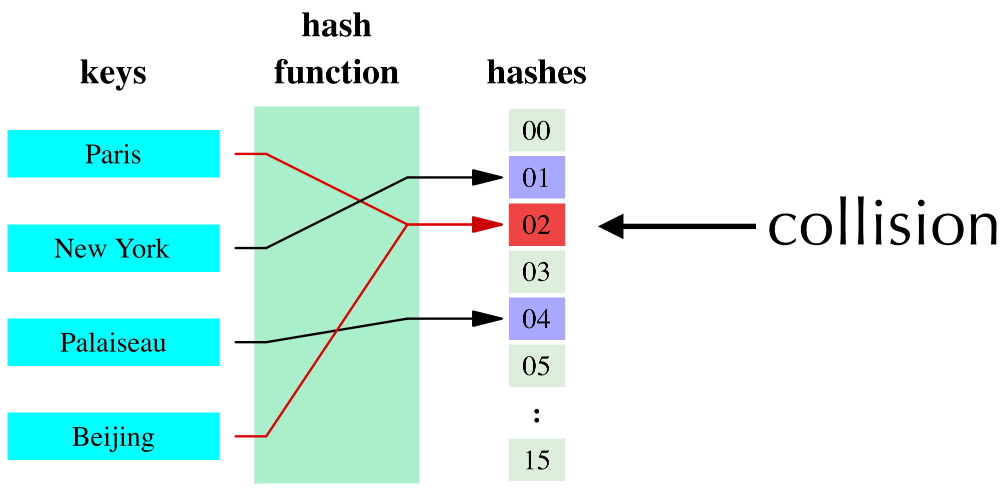
Size
Iter_keys

Hash tables
provide dictionaries & sets
with good complexity

Simpler variant: Sets

Collisions & Birthday Paradox

m : table size
n : number of keys



Under the uniformity assumption,

$$\mathbb{P}(\text{no collision}) = \left(1 - \frac{1}{m}\right)\left(1 - \frac{2}{m}\right)\cdots\left(1 - \frac{n-1}{m}\right)$$

Birthday paradox: assuming birthdays uniformly distributed

with $m = 365$, $n = 23$, $\mathbb{P}(\text{distinct birthdays}) < 1/2$.

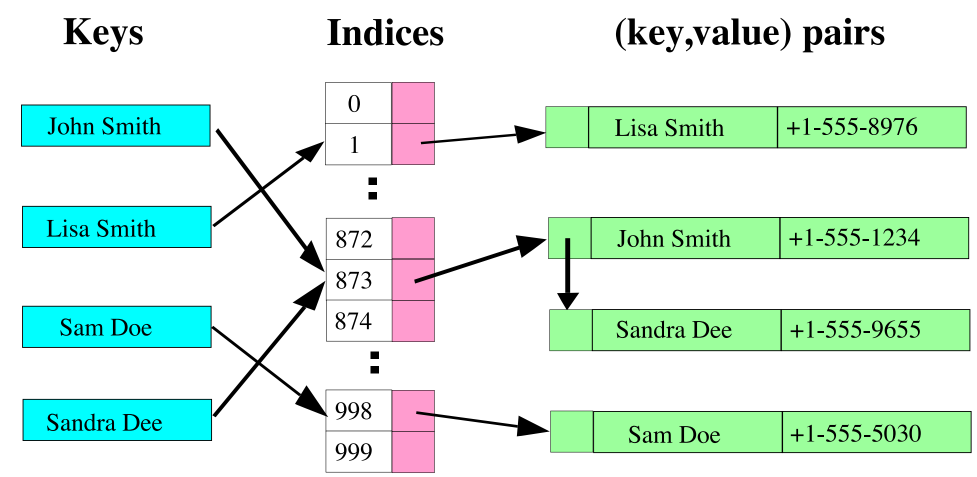
$n = 57$, $\mathbb{P}(\text{distinct birthdays}) < 1\,\%$.

Collisions do occur!

Hash tables need to detect & handle them.

Hashing with Separate Chaining

The table stores (key,value) pairs in linked lists.



Filling ratio: $\alpha = n/m$.

m : table size
n : number of keys

Time for insertion (or unsuccessful search):

worst-case: $O(n)$

uniformity assumption

expectation: $\mathbb{E}(\#\text{comparisons}) = \sum_{k=0}^{m-1} \frac{1}{m} \text{length}(T_k) = \alpha$.

All operations in $O(1)$ for bounded α

Time for successful search (or deletion):

$$\mathbb{E}(\#\text{comparisons}) = 1 + \sum_{i=1}^{n} \frac{1}{n} \frac{i-1}{m} = 1 + \frac{\alpha}{2} - \frac{1}{2m}.$$

Simple Dictionaries via Hash Tables with Separate Chaining

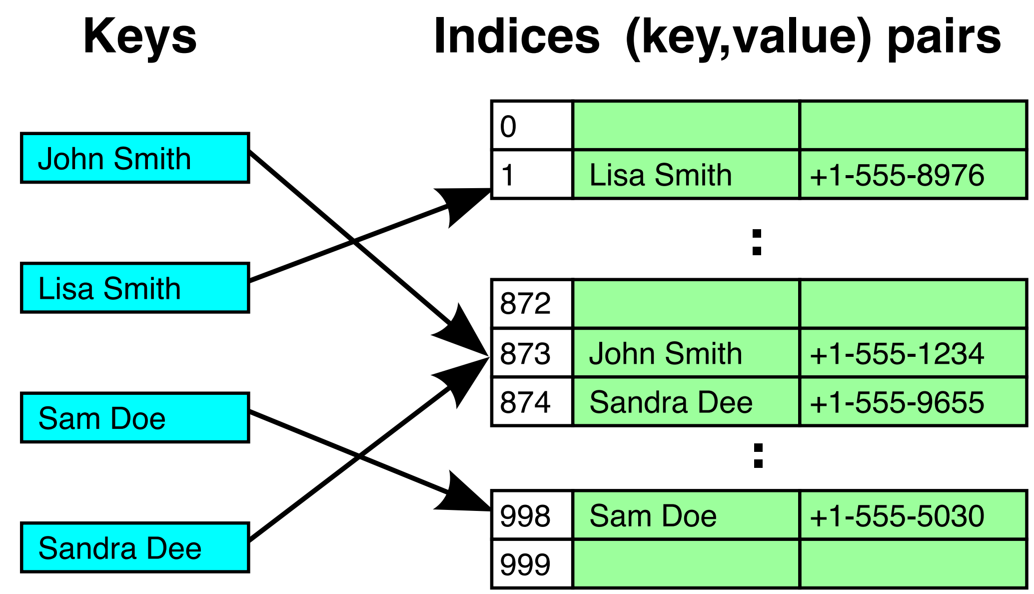
```
def Create(m):
    return [[]]*m

def FindInList(key,L):
    for i,(k,v) in enumerate(L):
        if key==k: return i
    return -1

def FindInTable(key,T):
    L = T[hash(key)]
    return L,FindInList(key,L)
```

```
def Insert(key,value,T):
    L,i = FindInTable(key,T)
    if (i==-1): L.append((key,value))
    else: L[i] = (key,value)

def Get(key,T):
    L,i = FindInTable(key,T)
    if i==-1: raise "Not Found"
    return L[i][1]

def Delete(key,T):
    L,i = FindInTable(key,T)
    if i!=-1: L.pop(i)

def Contains(T,key):
    return FindInTable(key,T)[1]!=-1
```

Exercise: modify to add Iter_keys

Hashing with Linear Probing

The table stores (key,value) pairs in successive slots.



Problem: long clusters tend to occur.

m : table size
n : number of keys

Time for insertion (or unsuccessful search):

$$\text{When } \alpha = n/m < 1, \quad \mathbb{E}(\#\text{probes}) = O(1)$$

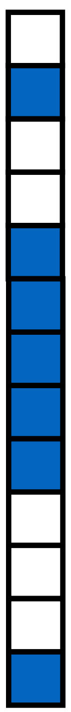
Proof next slide

and this is an upper bound on successful search.

All operations in $O(1)$ for bounded α

In practice, α is kept in (1/8,1/2) by resizing the table if necessary.

Proof of the Complexity (1/2)



$$
\begin{aligned}
\mathbb{E}(\#\text{probes}) &= 1 + \sum_{k\geq 1} k\,\mathbb{P}(\#\text{probes on occupied slots} = k)\\
&\leq 1 + \sum_{i=0}^{m-1} \frac{1}{m} \sum_{k\geq 1} k\,\mathbb{P}(i \text{ part of a cluster of length } k)\\
&\leq 1 + \sum_{i=0}^{m-1} \frac{1}{m} \sum_{k\geq 1} k^2\,\mathbb{P}(i \text{ starts a cluster of length } k)\\
&\leq 1 + \sum_{k\geq 1} k^2 c^k, \quad \text{with } c < 1\\
&= O(1)\,.
\end{aligned}
$$

Proof
next slide

Proof of the Complexity (2/2)

$q_k := \mathbb{P}(i$ starts a cluster of length $k)$

$$q_k \le \binom{n}{k} \left(\frac{k}{m}\right)^k \left(1 - \frac{k}{m}\right)^{n-k}$$

(k previous keys, in any order,
landed in those k slots)

$$\boxed{\textbf{Lemma.}\ \binom{n}{k} \le \frac{n^n}{k^k (n-k)^{n-k}}.}$$

Expand $(k + (n-k))^n$

$$q_k \le \frac{n^n}{k^k (n-k)^{n-k}} \left(\frac{\alpha k}{n}\right)^k \left(1 - \frac{\alpha k}{n}\right)^{n-k}$$

$$= \alpha^k \left(1 + \frac{k(1-\alpha)}{n-k}\right)^{n-k} \le \left(\alpha e^{1-\alpha}\right)^k.$$

$c < 1$ for $\alpha < 1$.

Lemma.

$(1 + x/m)^m \le e^x$.

Reduce to $\ln(1+x) \le x$

Simple Dictionaries via Hash Tables with Linear Probing

```
def Create(m):
    return [None]*m

def FindInTable(key,T):
    v = hash(key)
    while T[v]!=None and T[v][0]!=key:
        v = (v+1)%len(T)
    return v
```

Delete requires attention.

Still $O(1)$ on average (not proved here).

```
def Insert(key,value,T):
    v = FindInTable(key,T)
    T[v] = (key,value)

def Get(key,T):
    v = FindInTable(key,T)
    if T[v]==None: raise "Not Found"
    return T[v][1]

def Delete(key,T):
    v = FindInTable(key,T)
    T[v] = None
    while True:
        v = (v+1)%len(T)
        if T[v] == None: return
        k,val = T[v]
        T[v] = None
        Insert(k,val,T)

def Contains(T,key):
    return T[FindInTable(key,T)]!=None
```

III. Application to Sparse Matrices

Sparse Matrices & Google PageRank

Def. An $n \times m$ matrix is called *sparse* when its number of nonzero entries is $t \ll n \times m$.

Ex. Adjacency matrix of the graph of the web.

Data-structure: array of dictionaries, where only the nonzero entries are stored.

Matrix-vector product in $O(n + t)$ operations.

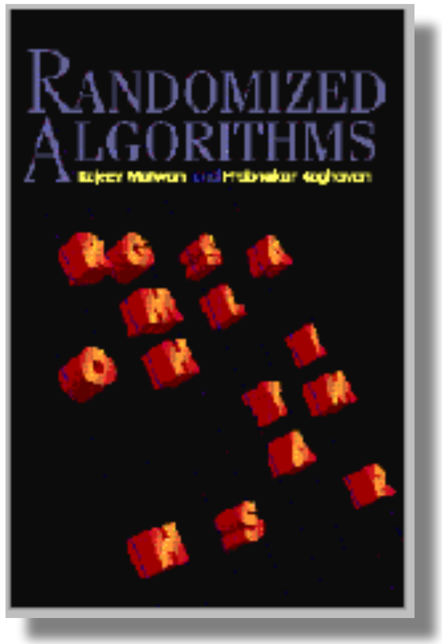
Exercise: implement it

Google PageRank iterates this until the vector converges.

References for this lecture

The slides are designed to be self-contained.

They were prepared using the following books that I recommend if you want to learn more:



Next

Assignment this week: more on hashing with separate chaining

Next tutorial: fingerprinting for text search

Next week: Randomization 3 — hard search problems

Midterm programming exam: next week

Feedback

Moodle for the slides, TDs and exercises.

Questions or comments: Bruno.Salvy@inria.fr