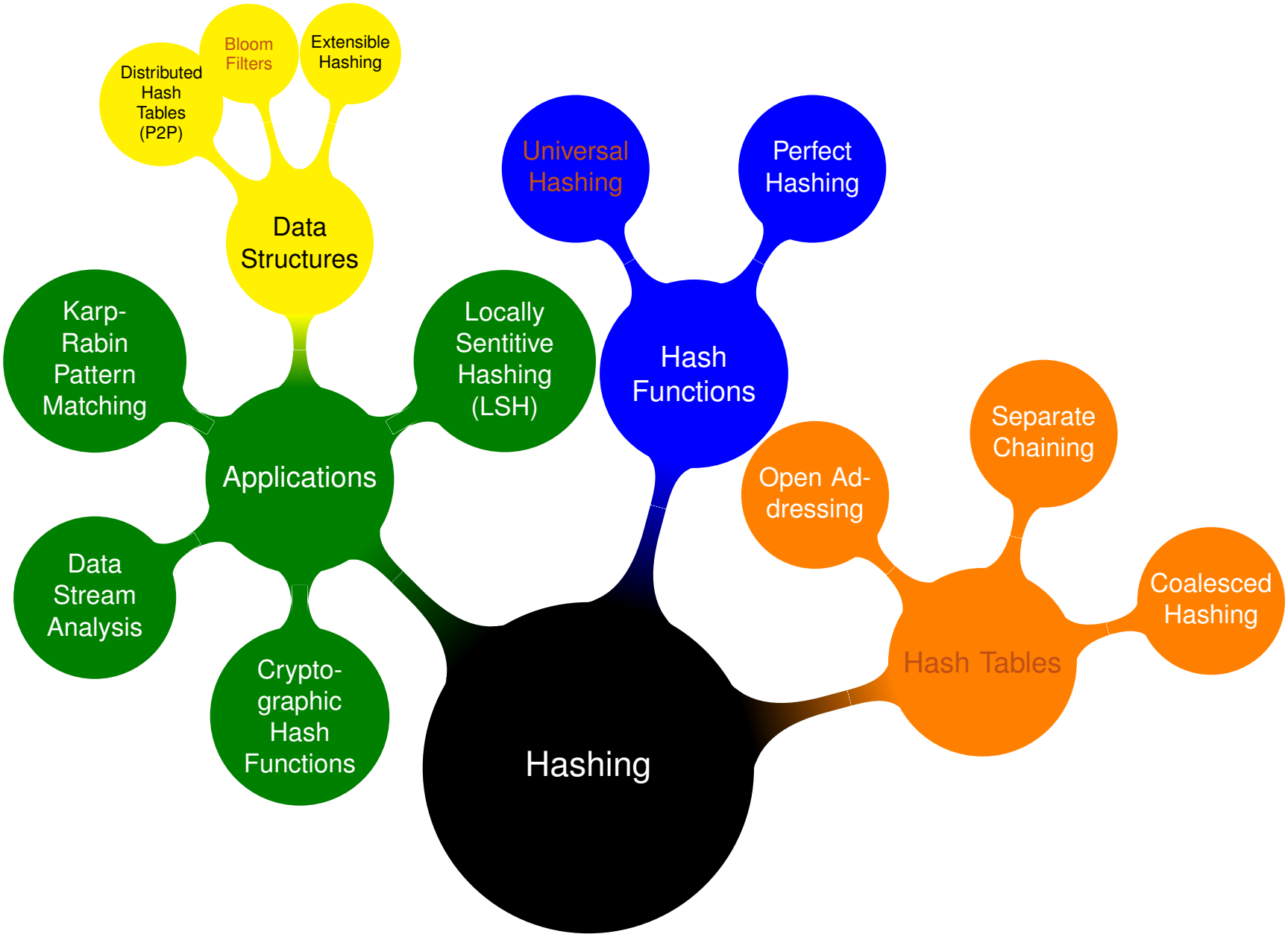
Hashing

Conrado Martínez
U. Politècnica de Catalunya

RA-MIRI 2023–2024

Hashing


Hashing
Data Structures
Distributed Hash Tables (P2P)
Bloom Filters
Extensible Hashing
Applications
Karp-Rabin Pattern Matching
Locally Sentitive Hashing (LSH)
Data Stream Analysis
Crypto-graphic Hash Functions
Hash Functions
Universal Hashing
Perfect Hashing
Hash Tables
Open Ad-dressing
Separate Chaining
Coalesced Hashing

Hashing

A hash function h maps the elements (keys) of a given domain (or *universe*) $\mathcal{U}$ in a finite range $0..M-1$.
Hash functions must:

1. Be easy and fast to compute
2. Be represented with little memory
3. Spread the universe as evenly as possible

$$\mathcal{U}_i = \{x \in \mathcal{U} \mid h(x) = i\}, \qquad 0 \leqslant i < M$$

$$|\mathcal{U}_i| \approx \frac{|\mathcal{U}|}{M}$$

4. Give very different hash values to “similar” keys

Part

Universal Hashing

M.N. Wegman

Definition

A class

$$\mathcal{H} = \{h \mid h : \mathcal{U} \to [0..M-1]\}$$

of hash functions is universal iff, for all $x, y \in \mathcal{U}$ with $x \neq y$ we have

$$\mathbb{P}[h(x) = h(y)] \leqslant \frac{1}{M},$$

where h is a hash function randomly drawn from $\mathcal{H}$

Universal Hashing

A stronger property is pairwise independence (a.k.a. strong universality). A class is strongly universal iff, for all $x, y \in \mathcal{U}$ with $x \neq y$ and any two values $i, j \in [0..M-1]$

$$\mathbb{P}[h(x) = i \wedge h(y) = j] = \frac{1}{M^2}$$

Strong universality implies universality; moreover

$$\mathbb{P}[h(x) = i] = \frac{1}{M}$$

for any x and i.

Universal Hashing

Let $\mathcal{H}$ be a universal class and $h \in \mathcal{H}$ drawn at random. For any fixed set of n keys $S \subseteq \mathcal{U}$ we have the following properties:

1. For any $x \in S$, the expected number of elements in S that hash to $h(x)$ is n/M.
2. The expected number of collisions is $O(n^2/M)$. If $M = \Theta(n)$ then the expected number of collisions is $O(n)$.

Universal Hashing

The big questions are:

- Are there universal classes? Strongly universal classes?
- If so, how complicated are its members? How much effort does it take to compute and represent the functions in the class?

Universal Hashing

In 1977 Carter and Wegman introduced the concept of universal class of hash functions and gave the first construction.
Put the universe $\mathcal{U}$ into one-to-one correspondence with $[0..U-1]$ ($U = |\mathcal{U}|$) and let p be a prime $\geqslant U$.
The class

$$\mathcal{H} = \{h_{a,b} \mid 0 < a < p, 0 \leqslant b < p\}$$

is (strongly) universal, with

$$h_{a,b}(x) = ((ax + b) \mod p) \mod M$$

Universal Hashing

The ingredients we need are thus a BIG prime p; picking a hash function at random from $\mathcal{H}$ amounts to choosing two integers a and b at random.
Let $\mathrm{r} = \lceil \log_2(\mathrm{U}+1) \rceil$. The prime number p and the numbers a and b will need roughly r bits each. For instance, if our universe are ASCII strings of length at most 30, $\mathrm{U} \approx 256^{30}$ and $\mathrm{r} \approx 240$ bits; these are huge numbers and a fast primality test is a must to have a practical scheme.

Universal Hashing

Suppose that $h_{a,b}$ has been picked at random and let x and y be two distinct keys that collide

$$h_{a,b}(x) = h_{a,b}(y)$$

Therefore

$$ax + b \equiv ay + b + \lambda \cdot M \pmod{p}$$

for some integer $\lambda \geqslant 0$, $\lambda \leqslant p/M$.

Universal Hashing

Since $x \neq y$, $x - y \neq 0$, hence $x - y$ has an inverse multiplicative in the ring $\mathbb{Z}_p$, denote it $(x-y)^{-1}$. Hence

$$\begin{aligned} ax &\equiv ay + \lambda \cdot M \pmod{p} \\ a(x-y) &\equiv \lambda \cdot M \pmod{p} \\ a &\equiv (x-y)^{-1} \cdot \lambda \cdot M \pmod{p} \end{aligned}$$

Universal Hashing

There are $p - 1$ possible choices for a and $\lfloor p/M \rfloor$ possible values for λ; hence the probability of collision is

$$\leqslant \frac{\lfloor p/M \rfloor}{p-1} \approx \frac{1}{M}$$

for sufficiently large p.

Universal Hashing

Notice that $\mathfrak{b}$ plays no rôle in the universality of the family. We might have choosen $\mathfrak{b} = 0$ or any other convenient fixed value. However, picking $\mathfrak{b}$ at random makes the class strongly universal.

Part

Hash Tables

A hash table (cat: *taula de dispersió*, esp: *tabla de dispersión*) allows us to store a set of elements (or pairs $\langle \mathrm{key}, \mathrm{value} \rangle$) using a hash function $\mathrm{h} : \mathrm{K} \Longrightarrow \mathrm{I}$, where I is the set of indices or addresses into the table, e.g., $\mathrm{I} = [0..\mathrm{M} - 1]$.

Ideally, the hash function h would map every element (their keys) to a distinct address of the table, but this is hardly possible in a general situation, and we should expect to find collisions (different keys mapping to the same address) as soon as the number of elements stored in the table is $\mathrm{n} = \Omega(\sqrt{\mathrm{M}})$.

Hash Tables

If the hash function evenly “spreads” the keys, the hash table will be useful as there will be a small number of keys mapping to any given address of the table.

Given two distinct keys x and y, we say that they are synonyms, also that they collide if $h(x) = h(y)$.

A fundamental problem in the implementation of a dictionary using a hash table is to design a collision resolution strategy.

Hash Tables

```
template <typename T> class Hash {
public:
  int operator()(const T& x) const ;
};

template <typename Key, typename Value,
          template <typename> class HashFunct = Hash>
class Dictionary {
public:
   ...
private:
   struct node {
       Key _k;
       Value _v;
       ...
   };
   nat _M; // capacity of the table
   nat _n; // number of elements (size) of the table
   double _alpha_max; // max. load factor
   HashFunct<Key> h;

   // open addressing
   vector<node> _Thash; // an array with pairs <key,value>

   // separate chaining
   // vector<list<node>> _Thash; // an array of lists of synonyms

   int hash(const Key& k)  {
        return h(k) % _M;
   }
};
```

Hash Functions

A good hash function h must enjoy the following properties

1. It is easy to compute
2. It must evenly spread the set of keys K: for all i, $0 \leqslant i < M$

$$\frac{\#\{k \in K \mid h(k) = i\}}{\#\{k \in K\}} \approx \frac{1}{M}$$

Hash Functions

In our implementation, the class `Hash<T>` overloads operator `()` so that for an object `h` of the class `Hash<T>`, `h(x)` is the result of "applying" `h` to the object `x` of class `T`. The operation returns a positive integer.
The private method `hash` in class Dictionary computes

```
h(x) % _M
```

to obtain a valid position into the table, an index between 0 and `_M - 1`.

Hash Functions

```
// specialization of the template for T = string
template <> class Hash<string> {
public:
  int operator()(const string& x) const  {

      int s = 0;
      for (int i = 0; i < x.length(); ++i)
          s = s * 37 + x[i];
      return s;
};

// specialization of the template for T = int
template <> class Hash<int> {
static long const MULT = 31415926;
public:
  int operator()(const int& x) const  {

      long y = ((x * x * MULT) << 20) >> 4;
      return y;
  }
};
```

Other sophisticated hash functions use weighted sums or non-linear transformations (e.g., they square the number represented by the k central bits of the key).

Collision Resolution

Collision resolution strategies can be grouped into two main families. By historical reasons (not very logically) they are called

- Open hashing: separate chaining, 2-way chaining, coalesced hashing, ...
- Open addressing: linear probing, double hashing, quadratic hashing, cuckoo hashing, ...

Part

Separate Chaining

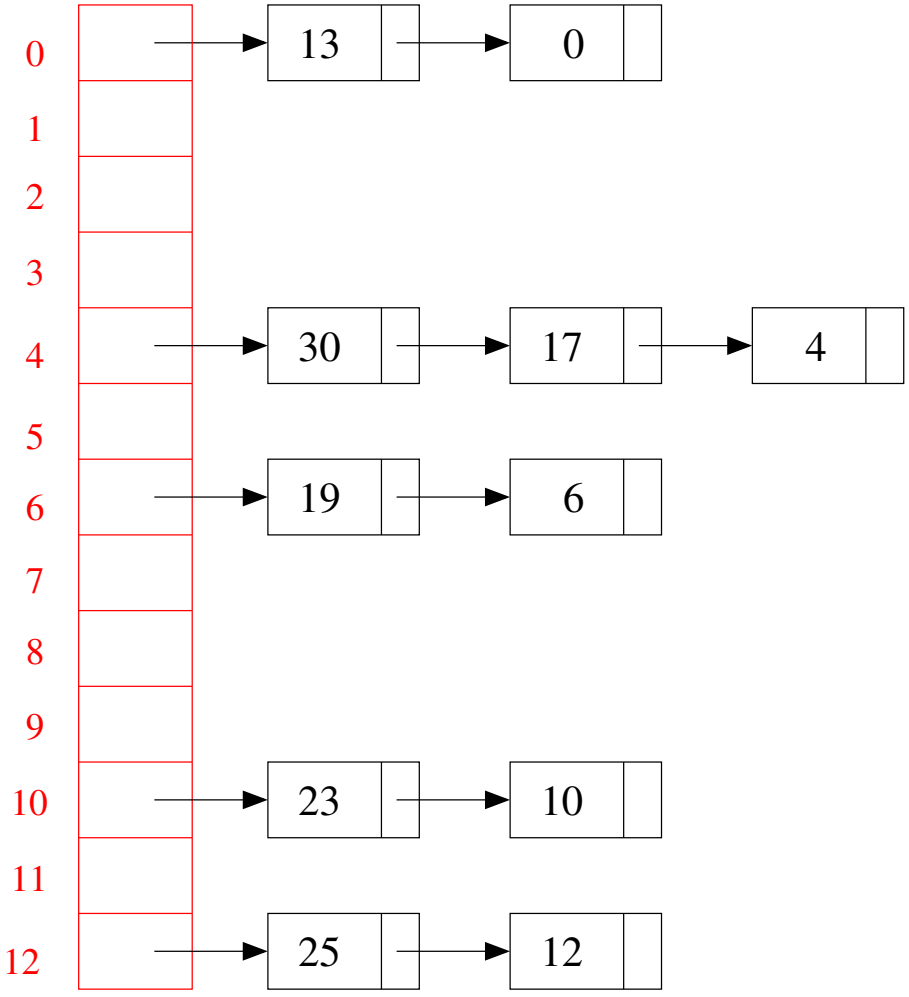
In separate chaining, each slot in the hash table has a pointer to a linked list of synonyms.

```
template <typename Key, typename Value,
          template <typename> class HashFunct = Hash>
class Dictionary {
   ...
private:

   struct node {
      Key _k;
      Value _v;
      ..-
   };
   vector<list<node>> _Thash; // array of linked lists of synonyms
   int _M;         // capacity of the table
   int _n;         // number of elements
   double _alpha_max;  // max. load factor

   list<node>::const_iterator lookup_sep_chain(const Key& k, int i) const ;
   void insert_sep_chain(const Key& k,
           const Value& v);
   void remove_sep_chain(const Key& k) ;
};
```

Separate Chaining

M = 13 X = { 0, 4, 6, 10, 12, 13, 17, 19, 23, 25, 30}

h (x) = x mod M



Separate Chaining

For insertions, we access the apropriate linked list using the hash function, and scan the list to find out whether the key was already present or not. If present, we modify the associated value; if not, a new node with the pair $\langle key, value \rangle$ is added to the list.
Since the lists contain very few elements each, the simplest and more efficient solution is to add elements to the front. There is no need for double links, sentinels, etc. Sorting the lists or using some other sophisticated data structure instead of linked lists does not report real practical benefits.

Separate Chaining

Searching is also simple: access the apropriate linked list using the hash function and sequentially scan it to locate the key or to report unsuccessful search.

Separate Chaining

```
template <typename Key, typename Value,
          template <typename> class HashFunct>
void Dictionary<Key,Value,HashFunct>::insert(const Key& k,
     const Value& v) {
   insert_sep_chain(k, v);
   if (_n / _M > _alpha_max)
      // the current load factor is too large, raise here an exception or
      // resize the table and rehash
}

template <typename Key, typename Value,
          template <typename> class HashFunct>
void Dictionary<Key,Value,HashFunct>::insert_sep_chain(
           const Key& k, const Value& v) {
   int i = hash(k);
   list<node>::const_iterator p = lookup_sep_chain(k,i);
   // insert as first item in the list
   // if not present
   if (p == _thash[i].end()) {

      _thash[i].push_back(node(k, v));
      ++_n;
   }
   else
      p -> _v = v;
}
```

Separate Chaining

```
template <typename Key, typename Value,
          template <typename> class HashFunct>
void Dictionary<Key,Value,HashFunct>::lookup(const Key& k,
   bool& exists, Value& v) const {
   int i = hash(k);
   list<node>::const_iterator p = lookup_sep_chain(k, i);
   if (p == _thash[i].end())
     exists = false;
   else {
     exists = true;
     v = p -> _v;
  }
}

template <typename Key, typename Value,
          template <typename> class HashFunct>
list<Dictionary<Key,Value,HashFunct>::node >::const_iterator
   Dictionary<Key,Value,HashFunct>::lookup_sep_chain(const Key& k,
                                                     int i) const  {

   list<node>::const_iterator p = _Thash[i].begin();
   // sequential search in the i-th list of synonyms
   while (p != _thash[i].end() and p -> _k != k)
      ++p;

   return p;
}
```

The Cost of Separate Chaining

Let n be the number of elements stored in the hash table. On average, each linked list contains $\alpha = n/M$ elements and the cost of lookups (either successful or unsuccessful), of insertions and of deletions will be proportional to α. If α is a small constant value then the cost of all basic operations is, on average, $\Theta(1)$. However, it can be shown that the expected length of the largest synonym list is $\Theta(\log n / \log \log n)$.
The value α is called load factor, and the performance of the hash table will be dependent on it.

The Cost of Separate Chaining

- $L_n^{(i)}$: the number of elements hashing to the i-th list, $0 \leqslant i < M$, after the insertion of n items.
- Standard assumption: the probability that the j-th inserted item hashes to position i, $0 \leqslant i < M$, is $1/M$
- The random variables $L_n^{(i)}$, $0 \leqslant i < M$, are **not** independent, but they are identically distributed
- Set $L_n := L_n^{(0)}$. Let $Y_j = 1$ iff the j-th inserted item goes to list 0, and $Y_j = 0$ otherwise.

$$\begin{aligned} L_n &= Y_1 + \ldots + Y_n \\ \mathbb{E}[L_n] &= \mathbb{E}[Y_1 + \ldots + Y_n] = \mathbb{E}[Y_1] + \ldots + \mathbb{E}[Y_n] \\ &= 1/M + \ldots + 1/M = n/M = \alpha \end{aligned}$$

The Cost of Separate Chaining

- Cost of unsuccessful search $U_n \approx$ cost of insertion of the $(n+1)$-th item

$$\begin{aligned}
\mathbb{E}[U_n] &= \sum_{0 \leqslant i < M} \mathbb{E}[U_n | \text{search in list } i] \cdot \mathbb{P}[\text{search in list } i] \\
&= \frac{1}{M} \sum_{0 \leqslant i < M} \mathbb{E}[U_n | \text{search in list } i] = \frac{1}{M} \sum_{0 \leqslant i < M} (1 + \mathbb{E}\left[L_n^{(i)}\right] \\
&= 1 + \alpha
\end{aligned}$$

The Cost of Separate Chaining

- Cost of succesful search of a random item $S_n \approx$ cost of deletion of a random item

$$\mathbb{E}[S_n] = \sum_{0 \leqslant i < M : L_n^{(i)} > 0} \mathbb{E}[S_n | \text{search in list } i] \cdot \mathbb{P}[\text{search in list } i]$$

$$= \sum_{0 \leqslant i < M : L_n^{(i)} > 0} \left(\sum_{\ell > 0} \frac{\ell + 1}{2} \mathbb{P}\left[L_n^{(i)} = \ell \right] \right) \cdot \frac{L_n^{(i)}}{n}$$

$$= \sum_{0 \leqslant i < M : L_n^{(i)} > 0} \frac{1 + \alpha}{2} \frac{L_n^{(i)}}{n}$$

$$= \frac{1 + \alpha}{2} \sum_{0 \leqslant i < M : L_n^{(i)} > 0} \frac{L_n^{(i)}}{n} = \frac{1 + \alpha}{2}$$

The Cost of Separate Chaining

- The Poisson model: in order to avoid the dependence between $\mathrm{L}_n^{(i)}$ we can consider a Poisson random model in which "balls" (items) are thrown into "bins" (slots in the hash table) at a rate α, then the length of each list $\mathcal{L}_i \sim \text{Poisson}(\alpha)$ is independent of all other
- We have, for instance, $\mathbb{E}[\mathcal{L}_i] = \alpha = \mathbb{E}\left[\mathrm{L}_n^{(i)}\right]$
- In general we can make our computations in the easier Poisson model then (rigorously) transfer these results to the "exact model"

The Cost of Separate Chaining

- Let $L_n^* = \max\{L_n^{(0)}, \ldots, L_n^{(M-1)}\}$. This random variable gives the worst-case cost of search, insertions and deletions
- An important identity for positive discrete r.v.

$$\mathbb{E}[X] = \sum_{k \geqslant 0} k\,\mathbb{P}[X = k] = \sum_{k \geqslant 0} \mathbb{P}[X > k]$$

The Cost of Separate Chaining

- In th Poisson model, we have M i.i.d. Poisson r.v. $\mathcal{L}_i$, all with parameter $\alpha = n/M$, giving the length of the i-th list, $0 \leqslant i < M$
- Then for $\mathcal{L}^* = \max_{0 \leqslant i < M}\{\mathcal{L}_i\}$ we have

$$\mathbb{P}[\mathcal{L}^* \leqslant k] = \prod_i \mathbb{P}[\mathcal{L}_i \leqslant k]$$

$$= \left(\sum_{0 \leqslant j \leqslant k} \frac{\alpha^j e^{-\alpha}}{j!} \right)^M$$

and

$$\mathbb{E}[\mathcal{L}^*] = \sum_{k>0} \left(1 - \left(\sum_{0 \leqslant j \leqslant k} \frac{\alpha^j e^{-\alpha}}{j!} \right)^M \right)$$

- But this path leads us nowhere.

The Cost of Separate Chaining

We will try a different way:

- Compute (or give useful bounds) for the median of $\mathcal{L}^*$, i.e., the value of j such that $\mathbb{P}[\mathcal{L}^* \leqslant j] = 1/2$
- Show that the expectation (mean) of $\mathcal{L}^*$ is close to its median, namely we show that

$$\frac{\mathbb{E}[\mathcal{L}^*]}{j} \to 1$$

if n is large enough (and $\alpha = n/M$ is kept constant)

The Cost of Separate Chaining

For which value j do we have

$$\mathbb{P}[\mathcal{L}^* \leqslant j] = \prod_i \mathbb{P}[\mathcal{L}_i \leqslant j] = \left(\sum_{0 \leqslant k \leqslant j} \frac{\alpha^k e^{-\alpha}}{k!} \right)^M = 1/2?$$

The summation

$$\sum_{0 \leqslant k \leqslant j} \frac{\alpha^k}{k!} \approx e^\alpha - \frac{\alpha^{j+1}}{(j+1)!},$$

hence

$$\mathbb{P}[\mathcal{L}^* \leqslant j] \approx \left(1 - \frac{\alpha^{j+1}}{(j+1)!} e^{-\alpha} \right)^M$$

The Cost of Separate Chaining

We want j such that

$$\left(1 - \frac{\alpha^{j+1}}{(j+1)!}e^{-\alpha}\right)^{M} = \frac{1}{2}$$

Taking natural logs on both sides

$$M \ln\left(1 - \frac{\alpha^{j+1}e^{-\alpha}}{(j+1)!}\right) = -\ln 2$$

Since $\ln(1-x) \sim x + x^2/2 + O(x^3)$

$$\left(\frac{\alpha^{j+1}e^{-\alpha}}{(j+1)!} + \ldots\right) \approx \frac{-\ln 2}{M}$$

The Cost of Separate Chaining

Hence

$$\frac{\alpha^{j+1}e^{-\alpha}}{(j+1)!} = \Theta(1/M)$$

- $\alpha \to 1$ implies $(j+1)! = \frac{M}{e \ln 2}$, that is, $j = \Gamma^{(-1)}(M/(e \ln 2))$.
- For $\alpha < 1$ we also have $j = \Theta(\Gamma^{(-1)}(M))$.
- Since $\Gamma^{(-1)}(n) \sim \ln n / \ln \ln n$, and $n = \alpha M$ we have that the median j of $\mathcal{L}^*$ is $j = \Theta(\log n / \log \log n)$.

(see next slide for definitions and remarks)

The Cost of Separate Chaining

Note:

- $\Gamma^{(-1)} =$ inverse of the Gamma function $\Gamma(z)$
- Γ generalizes factorials to complex numbers ($\Gamma(z+1) = z\Gamma(z)$).
- Since $\ln n! \sim n \ln n - n + O(1)$ (Stirling's approximation) we can easily prove $\Gamma^{(-1)}(n) \sim \ln n / \ln \ln n$ if $n \to \infty$.

For the rest of the proof (showing that the expected value of $\mathcal{L}^*$ has the same order of growth as its median) you can check Section 2.2 in [Gon81]

d-way Chaining

Azar, Broder, Karlin and Upfal [ABKU99] have shown the following important result

Theorem

Suppose n *balls are sequentailly placed in* $m \geqslant n$ *bins, so that for each ball* $d \geqslant 2$ *random bins are chosen and the ball is placed in the least full bin —with ties broken arbitrarily. Then with high probability, as* $n \to \infty$*, the fullest bin contains*

$$(1 + o(1)) \ln \ln n / \ln d + \Theta(m/n)$$

balls.

d-way Chaining

This result has many applications, not only for data structures design. In the context of hashing, the hashing scheme suggested by this result is very straightforward:

- To insert an item x, compute $i = h_1(x)$ and $j = h_2(x)$ with two (or in general d) independent hash functions and insert x in the synonym list which is shorter, i.e., list i if $L_i \leqslant L_j$ and vice-versa
- To search (or delete) an item x compute $i = h_1(x)$ and $j = h_2(x)$ and search for x in both lists (why? why not only the shortest?) Clearly if x is present it must be in one of these two lists

d-way Chaining

In d-way chaining we basically multiply by d the expected costs of all operations, as compared to separate chaining, as we need to evaluate d hash functions and search in that many lists. We also need to keep the size of each list.
It is very easy to show that with d-way chaining the expected length of each list is $\alpha = n/m$ like in ordinary separate chaining. However:

- the variance of each $L_n^{(i)}$ is smaller than in separate chaining
- the expected longest list has length $\Theta(\log \log n)$, a huge improvement w.r.t. the $\Theta(\log n / \log \log n)$ in separate chaining

For those interested in the details (in particular the proof of the result) check [ABKU99].

Part

Open Addressing

In open addressing, synonyms are stored in the hash table. Searches and insertions probe a sequence of positions until the given key or an empty slot is found. The sequence of probes starts in position $i_0 = h(k)$ and continues with $i_1, i_2, \ldots$ The different open addressing strategies use different rules to define the sequence of probes. The simplest one is linear probing:

$$i_1 = i_0 + 1, i_2 = i_1 + 1, \ldots,$$

taking modulo M in all cases.

Linear Probing

```
template <typename Key, typename Value,
          template <typename> class HashFunct = Hash>

class Dictionary {
   ...
private:
struct node {
   Key _k;
   Value _v;
   bool _free;
   // constructor for class node
   node(const Key& k, const Value& v, bool free = true);
};
vector<node>  _Thash;  // array of nodes
int _M;         // capacity of the table
int _n;         // number of elements
double _alpha_max;  // max. load factor (must be < 1)


int lookup_linear_probing(const Key& k) const ;
void insert_linear_probing(const Key& k,
        const Value& v);
void remove_linear_probing(const Key& k) ;
};
```

Linear Probing

M = 13 X = { 0, 4, 6, 10, 12, 13, 17, 19, 23, 25, 30}

h (x) = x mod M (incremento 1)

0	0
1	
2	
3	
4	4
5	
6	6
7	
8	
9	
10	10
11	
12	12

0	0	occupied
1	13	occupied
2		free
3		free
4	4	occupied
5	17	occupied
6	6	occupied
7	19	occupied
8		free
9		free
10	10	occupied
11	23	occupied
12	12	occupied

0	0	occupied
1	13	occupied
2	25	occupied
3		free
4	4	occupied
5	17	occupied
6	6	occupied
7	19	occupied
8	30	occupied
9		free
10	10	occupied
11	23	occupied
12	12	occupied

+ {0, 4, 6, 10, 12} + {13, 17, 19, 23} + {25, 30}

Linear Probing

```
template <typename Key, typename Value,
          template <typename> class HashFunct>
int Dictionary<Key,Value,HashFunct>::lookup(
        const Key& k,
        bool& exists, Value& v) const {
   int i = lookup_linear_probing(k);
   if (not _Thash[i]._free and _Thash[i]._k == k) {
      exists = true; v = _Thash[i]. _v;
   }
   else
      exists = false;
}

template <typename Key, typename Value,
          template <typename> class HashFunct>
int Dictionary<Key,Value,HashFunct>::lookup_linear_probing(
        const Key& k) const  {
   int i = hash(k);
   int visited = 0;  // this is only necessary if
                     // _n == _M, otherwise there is at least
                     // a free position
   while (not _Thash[i]._free and _Thash[i]._k != k
          and visited < _M) {
      ++visited;
      i = (i + 1) % _M;
   }
   return i;
}
```

Deletions in Open Addressing

There is no general solution for true deletions in open addressing tables. It is not enough to mark the position of the element to be removed as “free”, since later searches might report as not present some element which is stored in the table.

The general technique that can be used is lazy deletions. Each slot can be free, occupied or **deleted**. Deleted slots can be used to store there a new element, but they are not free and searches must pass them over and continue.

Deletions in Linear Probing

For linear probing, we can do true deletions. The deletion algorithm must continue probing the positions after the removed element, and moving to the emptied slot any element whose hash address is equal (or smaller in the cyclic order) to the address of the emptied slot. Moving an element creates a new emptied slot, and the procedure is repeated until an empty slot is found. In our implementation we will use the function $\texttt{displ}(j, i)$ which gives us the distance between j e i in the cyclic order: if $j > i$ we must turn around position _$M - 1$ and go back to position 0.

```
int displ(j, i, M) {
  if (i >= j)
     return i - j;
  else
     return M + (i - j);
}
```

Deletions in Linear Probing

```
// we assume _n < _M

template <typename Key, typename Value,
          template <typename> class HashFunct>
int Dictionary<Key,Value,HashFunct>::remove_linear_probing(
       const Key& k) const  {
    int i = lookup_linear_probing(k);
    if (not _Thash[i]._free) {
       // _Thash[i] is the element to remove
       int free = i; i = (i + 1) % _M; int d = 1;
       while (not _Thash[i]._free) {
             int i_home = hash(_Thash[i]._k);
             if (displ(i_home, i, _M) >= d) {
                _Thash[free] = _Thash[i]; free = i; d = 0;
             }
             i = (i + 1) % _M;  ++d;
       }
       _Thash[free]._free = true; --_n;
    }
}
```

Other Open Addressing Schemes

As we have already mention different probe sequences give us different open addressing strategies. In general, the sequence of probes is given by

$$i_0 = h(x),$$
$$i_j = i_{j-1} \oplus \Delta(j, x),$$

where $x \oplus y$ denotes $x + y \pmod{M}$.

Other Open Addressing Schemes

1. Linear Probing: $\Delta(j, x) = 1$ (or a constant); $i_j = h(x) \oplus j$
2. Quadratic Hashing: $\Delta(j, x) = a \cdot j + b$; $i_j = h(x) \oplus (Aj^2 + Bj + C)$; constants a and b must be carefully choosen to guarantee that the probe sequence will ultimately explore all the table if necessary
3. Double Hashing: $\Delta(j, x) = h_2(x)$ for a second independent hash function h_2 such that $h_2(x) \neq 0$; $i_j = h(x) \oplus j \cdot h_2(x)$
4. Uniform Hashing: $i_0, i_1, \ldots$ is a random permutation of $\{0, \ldots, M-1\}$
5. Random Probing: $i_0, i_1, \ldots$ is a random sequence such that $0 \leqslant i_k < M$, for all k, and it contains every value in $\{0, \ldots, M-1\}$ at least once

Other Open Addressing Schemes

Uniform Hashing and Random Probing are completely impractical algorithms; they are interesting as idealizations —they do not suffer from clustering

- Linear Probing suffers primary clustering. There are only M distinct probe sequences, the M circular permutations of $0, 1 \ldots, M - 1$
- Quadratic Hashing and other methods with $H(j, x) = f(j)$ (a non-constant function only of j) behave almost as the schemes with *secondary clustering*: two keys such that $h(x) = h(y)$ will probe exactly the same sequence of slots, but if a key x probes i_j in the j-th step and y probes i'_k in the k-th step then i_{j+1} and i'_{k+1} will be probably different
- Double Hashing is even better and generalizations, they exhibit secondary (more generally k-ary clustering) as they depend on $(k - 1)$ evaluations of independent hash functions

Other Open Addressing Schemes

- In linear probing two keys will have the same probe sequence with probability $1/M$; in an scheme with secondaty clustering that probability drops to $1/M(M-1)$
- The average performance of schemes with k-ary clustering, $k \geqslant 2$, is close to that of uniform hashing (no clustering)
- Random probing also approximates well the performance of uniform hashing

The Cost of Open Addressing

We will focus in the following parameters (we assume M is fixed):

1. $\mathcal{U}_n$: number of probes in an unsuccessful search that starts at a random slot in a table with n items
2. $\mathcal{S}_{n,i}$: number of probes in the successful search of the i-th inserted item when the table contains n items, $1 \leqslant i \leqslant n$

We will actually be more interested in $\mathcal{S}_n := \mathcal{S}_{n,\mathcal{U}_n}$ where $\mathcal{U}_n$ is a random uniform value in $\{1, \ldots, n\}$, that is, $\mathcal{S}_n$ is the cost of a successful search of a random item in a table with n items

The Cost of Open Addressing

- The cost of the $(n+1)$-th insertion is given by $\mathcal{U}_n$
- With the FCFS insertion policy (see next slides), an item will be inserted where the unsuccessful search terminated and never be moved from there, hence

$$\mathcal{S}_{n,i} \stackrel{\mathcal{D}}{=} \mathcal{U}_{i-1}$$

where $\stackrel{\mathcal{D}}{=}$ denotes equal distribution

The Cost of Open Addressing

Consider random probing. What is $U_n = \mathbb{E}[\mathcal{U}_n]$?
With one probe we land in an empty slot and we are done. Probability is $(1-\alpha)$. If the first place is occupated, probability α, we probe a second slot, which is empty with probability $1-\alpha$. And so on. Thus

$$U_n = 1 \times (1-\alpha) + 2 \times \alpha \cdot (1-\alpha) + 3 \times \alpha^2 \cdot (1-\alpha)$$

$$= \sum_{k>0} k\alpha^{k-1} \cdot (1-\alpha) = (1-\alpha) \sum_{k>0} \frac{d(\alpha^k)}{d\alpha}$$

$$= (1-\alpha)\frac{d}{d\alpha} \sum_{k>0} \alpha^k = \frac{1}{1-\alpha}.$$

The Cost of Open Addressing

And for the expected successful search we have

$$S_n = \mathbb{E}[\mathcal{S}_n] = \frac{1}{n} \sum_{1 \leqslant i \leqslant n} \mathbb{E}[\mathcal{S}_{n,i}] = \frac{1}{n} \sum_{1 \leqslant i \leqslant n} \mathbb{E}[\mathcal{U}_{i-1}] = \frac{1}{n} \sum_{1 \leqslant i \leqslant n} U_{i-1}$$

Using Euler-McLaurin

$$S_n = \frac{1}{\alpha M} \sum_{1 \leqslant i \leqslant n} U_{i-1} = \frac{1}{\alpha} \int_0^{\alpha} \frac{1}{1-\beta} \, d\beta = \frac{1}{\alpha} \ln \left(\frac{1}{1-\alpha} \right)$$

The Cost of Open Addressing

The actual expected costs of hashing with uniform hashing (and thus of quadratic hashing, double hashing) are slightly different from those of random probing, a few small corrections must be introduced:

- $U_n = 1/(1-\alpha) - \alpha - \ln(1-\alpha)$
- $S_n = 1/\alpha \int_0^\alpha U(\beta)\, d\beta = 1 - \alpha/2 - \ln(1-\alpha) \qquad (*)$

The Cost of Open Addressing

The analysis of linear probing turns out to be more challenging than one could think at first.

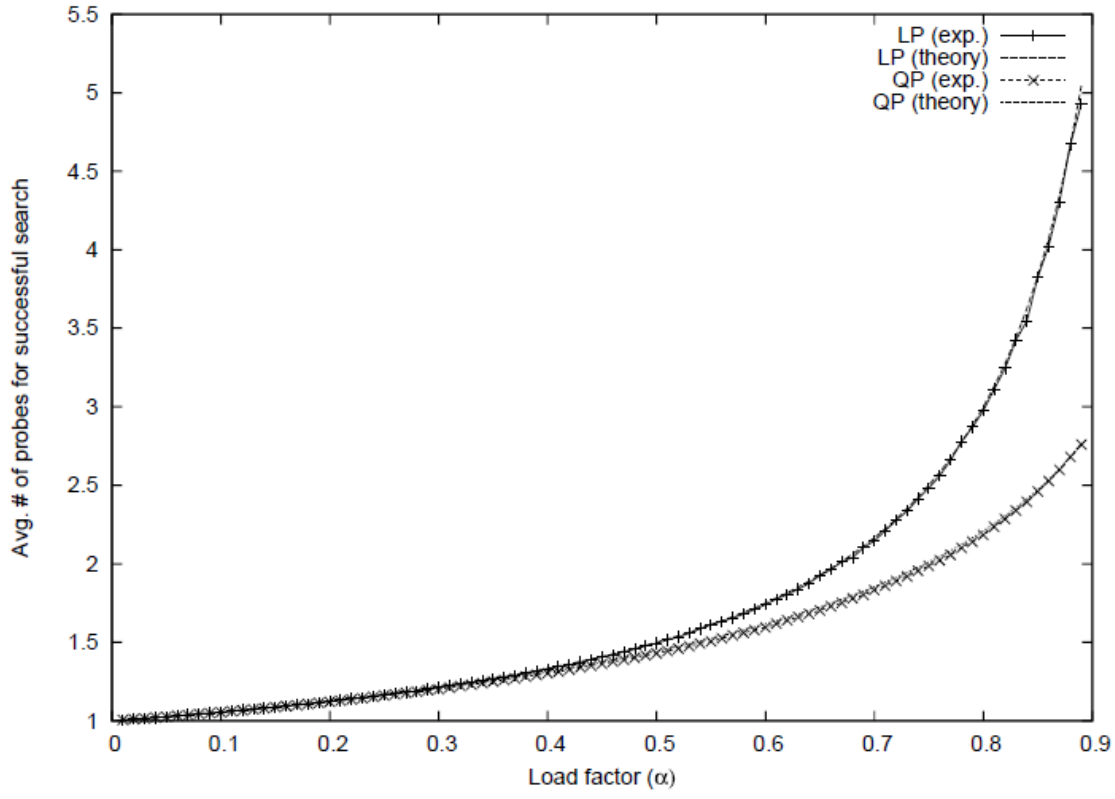
- The average cost of unsuccessful search is

$$U_n = \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

- The average cost of successful search is

$$S_n = \frac{1}{\alpha}\int_0^{\alpha} U(\beta)\,d\beta = \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) \qquad (**)$$

The Cost of Open Addressing



Comparison of experimental vs. theoretical expected cost of successful search in linear probing and quadratic hashing

Insertion Policies in Open Addresing

- The standard insertion policy in case of a collision is FCFS (first-come-first-served): the item x that occupies an slot remains there, and the colliding item y continues with its probe sequence
- But other policies are also possible and have been proposed in the literature:
  - LCFS (last-come-first-served): y kicks out x, x continues with its probe sequence
  - Ordered hashing: If $x \leqslant y$, x remains and y continues, and the other way around otherwise
  - Robin Hood: the item farthest away from its home location stays, the other continues, ties are resolved arbitrarily

Insertion Policies in Open Addresing

All these strategies lead to the same average successful search cost as FCFS, but:

- the variance is significantly reduced
- most importantly, the expected worst case is reduced from $\Theta(\log n)$ to $\Theta(\log \log n)$

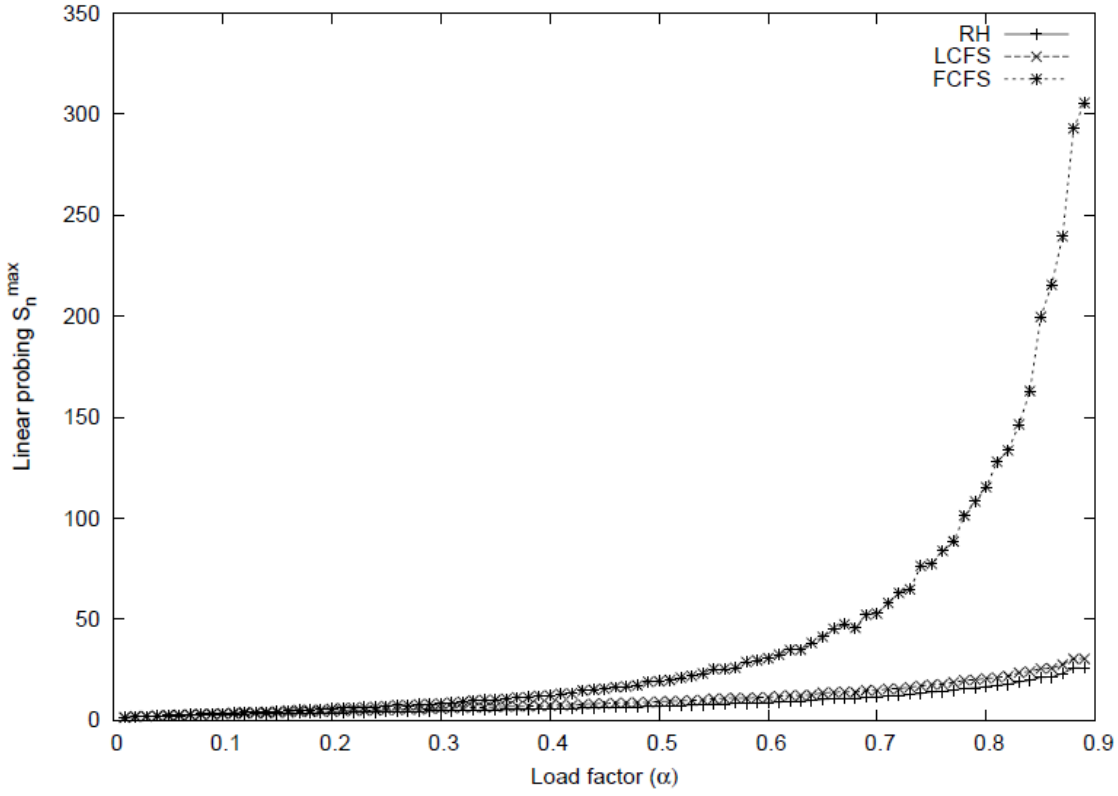
Think for instance in linear probing. The length of clusters will be the same for FCFS, LCFS, OH and RH, and the **sum of the distances** of all items to their respective home locations also changes but the distribution of distances to home location will vastly differ—like in the two sums below

$$1 + 3 + 7 = 3 + 4 + 4$$

Insertion Policies in Open Addresing

- Both ordered hashing and Robin Hood have the very nice feature that, given a set X of items to be inserted the final table is always the same, irrespective of the order in which item are inserted.
- This invariance with respect the prder of insertions notably simplifies some analysis.
- The unsuccessful search cost in OH and RH can be greatly improved; no need to continue until an empty slot is found (why?)
- But the insertion cost is the same, we either stop at an empty location or we kick out some item to insert the new item, but then we must continue

Insertion Policies in Open Addresing



Comparison of the maximum expected cost of a successful search in linear probing with FCFS (standard), LCFS and RH

Part

Cuckoo Hashing

Rasmus Pagh

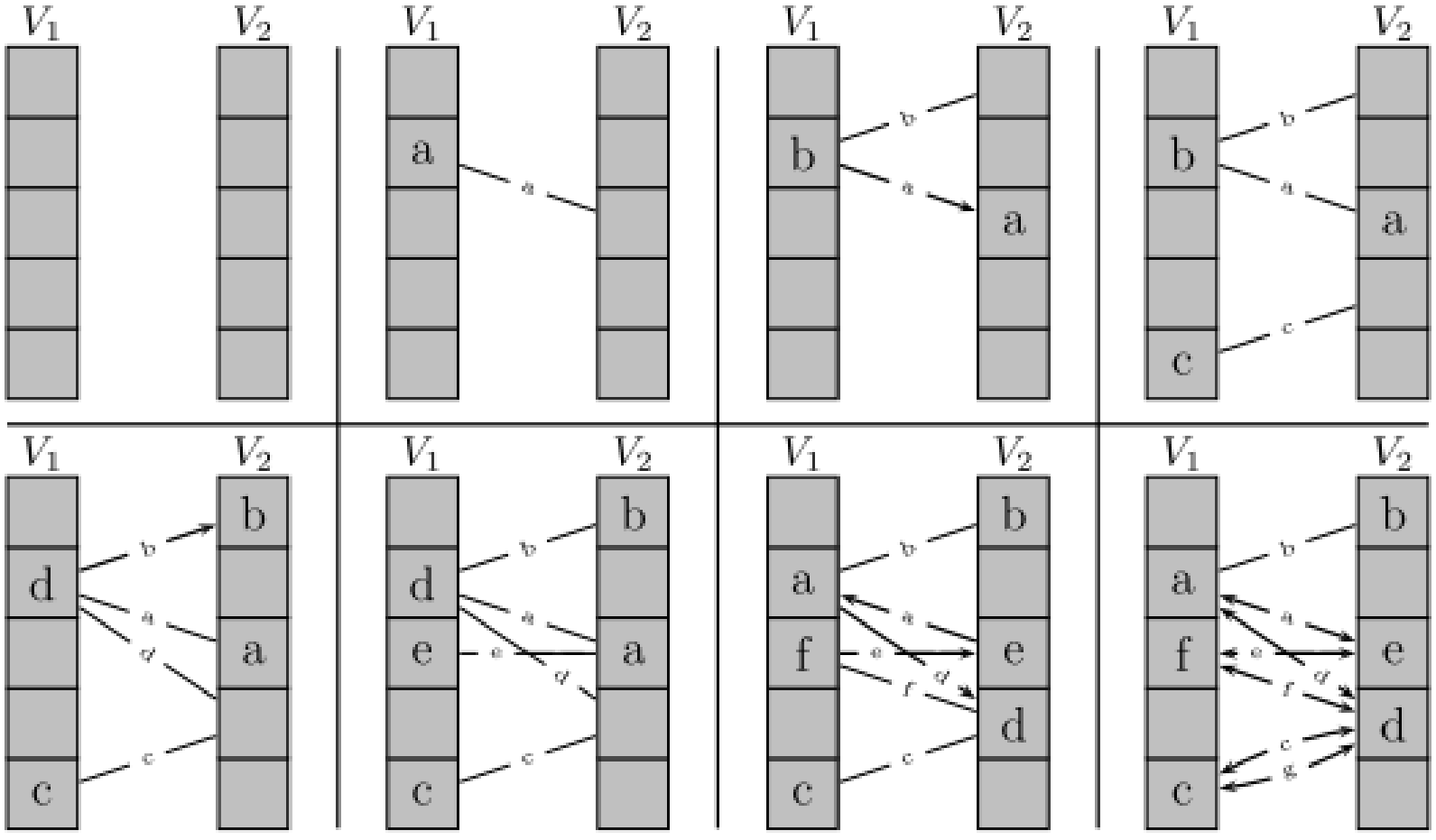
Flemming F. Rodler

In cuckoo hashing we have **two** tables T_1 and T_2 of size M each, and two hash functions $h_1, h_2 : 0 \to M - 1$.
The worst-case complexity of searches and deletions in a cuckoo hash table is $\Theta(1)$. We can insert in such table $n < M$ items: the load factor $\alpha = n/2M$ must be strictly less than 1/2 in order to guarantee constant expected time for insetions.

Cuckoo Hashing

To insert a new item x, we probe slot $T_1[h_1(x)]$, if it is empty, we put x there and stop. Otherwise if y sits already in that slot, then x kicks out y— x is put in $T_1[h_1(x)]$ and y moves to $T_2[h_2(y)]$.
If that slot in T_2 is empty, we're done, but if some z occupies $T_2[h_2(y)]$, then y is put in its second "nest" and z is kicked out to $T_1[h_1(z)]$, and so on.
These "kicks out" give the name to this strategy. If this procedure succeeds to insert n keys then each key x **can only appear in one of its two nests**: $T_1[h_1(x)]$ or $T_2[h_2(x)]$, nowhere else!

Cuckoo Hashing



	a	b	c	d	e	f	g
h_1	2	2	5	2	3	3	5
h_2	3	1	4	4	3	4	4

Cuckoo Hashing

```
// Representation of the dictionary with cuckoo hashing
    struct node {
      Key _k;
      Value _v;
      bool _free;
    };
    vector<node> _T1, _T2;
    int _M, _n;
    Hash<Key> _h1, _h2;
    ...
```

Cuckoo Hashing

```
void lookup(const Key& k, bool& exists, Value& v) const {
  exists = false;
  node& n1 = _T1[_h1(x)];
  if (not n1._free and n1._key == k) {
    exists = true; v = n1._v;
  } else {
    node& n2 = _T2[_h2(x)];
    if (not n1._free and n1._key == k) {
      exists = true; v = n2._v;
    }
  }
}
```

Only two probes are necessary in the worst-case! To delete we localate with $\leqslant 2$ probes the key to remove and mark the slot as free.

Cuckoo Hashing

- The insertion of an item x can fail because we enter in an infinite loop of items each kicking out the next in the cycle . . .
- The solution to the problem: nuke the table! Draw two new hash functions, and rehash everything again with the two new functions.
- This rehashing is clearly quite costly; moreover, we don't have a guarantee that the new functions will succeed where the old failed!

We will see, however, that insertion has expected amortized constant cost, or equivalently, that the expected cost of n insertions is $\Theta(n)$

Cuckoo Hashing

```
void insert(const Key& k, const Value& v) {
  if (_n == _M - 1) { // resize and rehash, cannot insert >= _M items
  }
  // _n < _M - 1
  if (``k in the table'') { // update v and return
  }
  node x = { k, v, false };
  for (int i = 1; i <= MaxIter(_n, _M);++i) {
    // we can take MaxIter = 2n, we should never see
    // more than 2n vertices unles we're in an infinite loop
    swap(x, _T1[_h1(x._k)]);
    if (x._free) return;
    swap(x, _T2[_h2(x._k)]);
    if (x._free) return;
  }
  // failure!! rehash and try again:
  rehash(); insert(k,v);
}
```

Cuckoo Hashing

We say that an insertion is *good* if it does not run into a infinite loop (our implementation protects from ∞-loops by bounding the number of iterations).
A "high-level analysis" of the cost of insertions follows from:

1. The expected number of steps/iterations in a good insertion is $\Theta(1)$
2. The probability that the insertion of an item is not good is $O(1/n^2)$

Cuckoo Hashing

3. By the union bound, the probability that we fail to make n consecutive good insertions is $O(1/n)$
4. The expected total cost of making n good insertions—conditioned on the event that we can make them—is $n \times \Theta(1) = \Theta(n)$

Cuckoo Hashing

1. The expected number of times we need to rehash a set of n items until we can insert all with good insertions is given by a geometric r.v. with probability of success $1 - O(1/n)$:

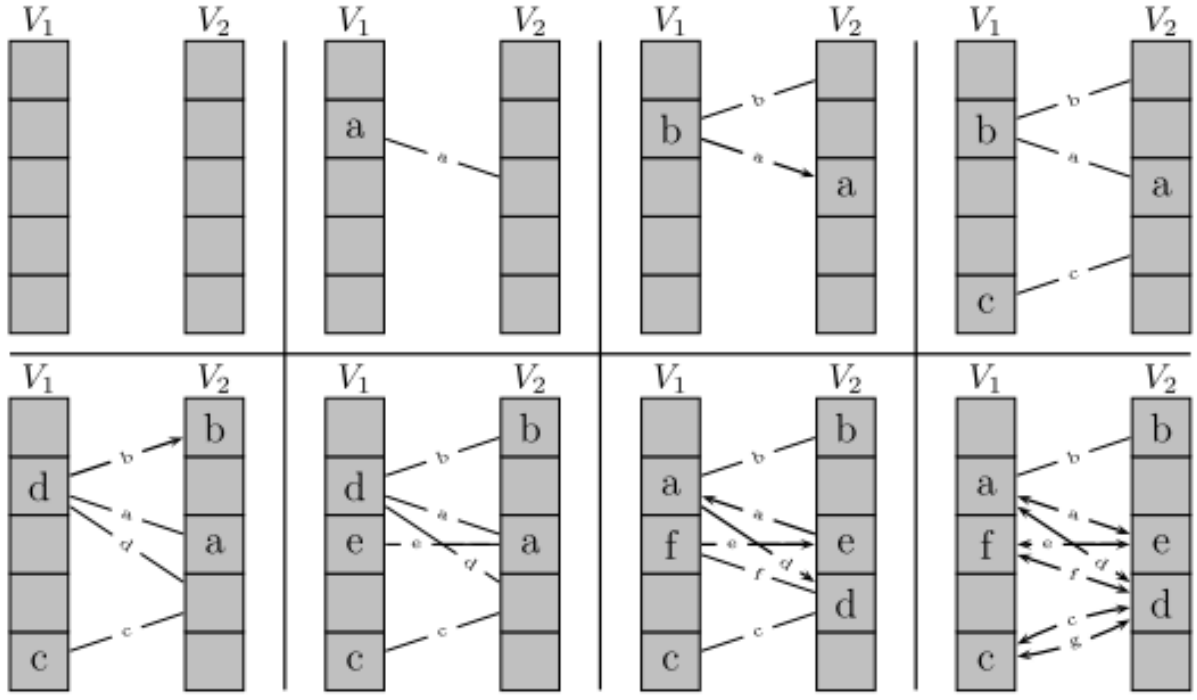
$$\mathbb{E}[\text{\# rehashes}] = \frac{1}{1 - O(1/n)} = 1 + O(1/n)$$

2. Each rehash plus the attempt to insert with good insertions the n items has expected cost $\Theta(n)$
3. By Wald's lemma, the expected cost of the insertion will be

$$\mathbb{E}[\#\text{rehashes}] \times \mathbb{E}[\text{cost of rehash}] = (1 + O(1/n)) \times O(n) = O(n)$$

Cuckoo Hashing

To prove facts #1 (good insertion needs expected O(1) time) and #2 (probability of a good insertion is $1 - O(1/n^2)$) we formulate the problem in graph theoretic terms.



	a	b	c	d	e	f	g
h_1	2	2	5	2	3	3	5
h_2	3	1	4	4	3	4	4

Cuckoo Hashing

Cuckoo graph:

- Vertices: $V = \{v_{1,i}, v_{2,i} \mid 0 \leqslant i < M\} =$ the set of $2M$ slots in the tables
- Edges: If $T_1[j]$ is occupied by x then there's an edge $(v_{1,j}, v_{2,h_2(x)})$, where $v_{\ell,j}$ is the vertex associated to $T_\ell[j]$; x is the label of the edge. If $T_2[k]$ is occupied by y then there is an edge $(v_{2,k}, v_{1,h_1(y)})$ with label y.

This is a labeled directed "bipartite" multigraph—all edges go from $v_{1,j}$ to $v_{2,k}$ or from $v_{2,k}$ to $v_{1,j}$.

Cuckoo Hashing

Consider the connected components of the cuckoo graph. A component can be either a tree (no cycles), unicyclic (exactly one cycle–with trees "hanging") or complex (two or more cycles). Trees with k nodes have exactly $k-1$ edges, unicycles have exactly k edges and complex components have $> k$ edges.

- Fact 1: An insertion that creates a complex component is not good $\implies$ if the cuckoo graph contains no complex components then all insertions were good
- Fact 2: the expected time of a good insertion is bounded by the expected diameter of the component in which we make the insertion (also by the size)

Cuckoo Hashing

Then we convert the analysis to that of the cuckoo graph as a random bipartite graph with $2M$ vertices and $n = (1 - \epsilon)M$ edges—each item gives us an edge.
This is a very "sparse" graph, but if the density n/M grew to 1/2 there will be an complex component with very high probability (a similar thing happens in random Erdös-Renyi graphs).

Cuckoo Hashing

The most detailed analysis of the cuckoo graph has been made by Drmota and Kutzelnigg (2012). They prove, among many other things:

1. The probability that the cuckoo graph contains no complex component is

$$1 - h(\epsilon)\frac{1}{M} + O(1/M^2)$$

   We do not reproduce their explicit formula for $h(\epsilon)$ here ($h(\epsilon) \to \infty$ as $\epsilon \to 0$)

2. The expected number of steps in n good insertions is

$$\leqslant n \cdot \min\left(4, \frac{\ln(1/\epsilon)}{1-\epsilon}\right) + O(1)$$

These two results prove the two Facts that we needed for our analysis

Cuckoo Hashing

- Several variants of Cuckoo hashing have appeared in the literature, for instance, using $\mathrm{d} > 2$ tables and d hash functions. With such d-Cuckoo Hashing higher load factor, approaching 1 can be achieved
- An interesting variant puts all items in one single table, all the $\mathrm{d} \geqslant 2$ hash functions map keys into the range $0..M-1$; the load factor n/M must be below some threshold α_{d}. We need to now which function was used to put the item at an occupied location—easily using $\log_2 \mathrm{d}$ bits.

Part

Bloom Filters

A Bloom Filter is a probabilistic data structure representing a set of items; it supports:

- Addition of items: $F := F \cup \{x\}$
- Fast lookup: $x \in F$?

Bloom filters do require very little memory and are specially well suited for unsuccessful search (when $x \notin F$)

Bloom Filters

- The price to pay for the reduced memory consumption and very fast lookup is the non-null probability of false positives.
- If $x \in \mathbb{F}$ then a lookup in the filter will always return true; but if $x \notin \mathbb{F}$ then there is some probability that we get a positive answer from the filter.
- In other words, if the filter says $x \notin \mathbb{F}$ we are sure that's the case, but if the filter says $x \in \mathbb{F}$ there is some probability that this is an error.

Bloom Filters

```
template <class T>
class BloomFilter {
public:
   // creates a Bloom filter to store at most nmax items
   // with an upper bound 'fp' for false positives
   BloomFilter(int nmax, double fp = 0.05);
void insert(const T& x);
bool contains(const T& x) const;
private:
   ...
}
```

Bloom Filters

```
template <class T>
class HashFunction {
public:
   HashFunction(int M);
int operator()(const T& x) const;
   ...
};

template <class T>
class BloomFilter {
   ...
private:
   bitvector F;
   vector<HashFunction<T> > h;
   int M, k;
   ...
};

template <class T>
BloomFilter::BloomFilter(int nmax, double fp = 0.05) {
    // compute here M and k to achieve the guarantee on false
    // positives
    F = bitvector(M, 0);
    for (int i = 0; i < k; ++i)
       h.push_back(HashFunction<T>(M));
}
```

Bloom Filters

```
template <class T>
void BloomFilter::insert(const T& x) {
  for (int i = 0; i < k; ++i)
     F[h[i](x)] = 1;
}

template <class T>
void BloomFilter::contains(const T& x) {
  for (int i = 0; i < k; ++i)
     if (F[h[i](x)] == 0)
        return false;
  return true; // might be a false positive!
}
```

Bloom Filters

- Probability that the j-th bit is not updated in an insertion

$$\prod_{i=0}^{k-1} \mathbb{P}[h_i(x) \neq j] = \left(1 - \frac{1}{M}\right)^k$$

- Probability that the j-th bit is not updated after n insertions

$$\prod_{\ell=1}^{n} \mathbb{P}[F[j] \text{ is not updated in } \ell\text{-th insertion}] =$$

$$\left(\left(1 - \frac{1}{M}\right)^k\right)^n = \left(1 - \frac{1}{M}\right)^{k \cdot n}$$

Bloom Filters

- Probability that $F[j] = 1$ after n insertions

$$1 - \left(1 - \frac{1}{M}\right)^{k \cdot n}$$

- Probability that the k checked bits are set to 1 $\approx$ probability of a false positive

$$\left(1 - \left(1 - \frac{1}{M}\right)^{k \cdot n}\right)^{k} \approx \left(1 - e^{-kn/M}\right)^{k}$$

if $n = \alpha M$, for some $\alpha > 0$

$$\left(1 - \frac{a}{x}\right)^{bx} \to e^{-ba}, \quad x \to \infty$$

Bloom Filters

- The derivation above is the so-called classic model for Bloom filters—but it i snot the formula that Bloom himslef derived in his paper!
- The approximation fails for small filters; correct formulas have been derived by Bose et al. (2008) and Christensen et al. (2010)
- For the rest of the presentation we will take

$$\begin{aligned}\mathbb{P}[x \text{ is a false positive}] &= \mathbb{P}[x \notin F \wedge F.\texttt{contains}(x) = \textbf{true}] \\ &\approx \left(1 - e^{-kn/M}\right)^k,\end{aligned}$$

where x is drawn at random. Be careful! The formula does not give the probability that the filter reports x as a positive, conditioned to x being negative!

Bloom Filters

- Fix n and M. The optimal value k^* minimizes the probability of false positive, thus

$$\frac{d}{dk}\left[\left(1-e^{-kn/M}\right)^k\right]_{k=k^*}=0$$

  which gives

$$k^* \approx \frac{M}{n}\ln 2 \approx 0.69\frac{M}{n}$$

- Call p the probability of a false positive. This probability is a function of k, $p = p(k)$; for the optimal choice k^* we have

$$p(k^*) \approx \left(1-e^{-\ln 2}\right)^{\frac{M}{n}\ln 2} = \left(\frac{1}{2}\right)^{\ln 2\frac{M}{n}} \approx 0.6185^{\frac{M}{n}}$$

Bloom Filters

- Suppose that you want the probability of false positive $p^* = p(k^*)$ to remain below some bound P

$$p^* \leqslant P \implies \ln p^* = -\frac{M}{n}(\ln 2)^2 \leqslant \ln P$$

$$\frac{M}{n}(\ln 2)^2 \geqslant -\ln P = \ln(1/P)$$

$$\frac{M}{n} \geqslant \frac{1}{\ln 2}\log_2(1/P) \approx 1.44\log_2(1/P)$$

$$M \geqslant 1.44 \cdot n \cdot \log_2(1/P)$$

Bloom Filters

- If we want a Bloom filter for a database that will store about $n \approx 10^8$ elements and a false positive rate $\leqslant 5\%$, we need a bitvector of size $M \geqslant 624 \cdot 10^6$ bits (that's around 74GB of memory).
- Despite this amount of memory is big, it is only a small fraction of the size of the database itself: even if we store only keys of 32 bytes each, the database occupies more than 3TB.
- The optimal number k^* of hash functions for the example above is 4.32 ($\Longrightarrow$ use 4 or 5 hash functions for optimal performance)

Bloom Filters

```
template <class T>
BloomFilter::BloomFilter(int nmax, double fp = 0.05) {
    // compute here M and k to achieve the guarantee on false
    // positives
    M = int(log(1/P)*nmax/log(2)*log(2));
    k = int(log(2)* M/nmax);
    ...
}
```

References

Hash Tables & Cuckoo Hashing

[1] Y. Azar, A. Broder, A. Karlin and E. Upfal.
Ballanced Allocations
SIAM J. Computing, 29(1):180-200, 1999.

[2] Gaston H. Gonnet
Expected Length of the Longest Probe Sequence in Hash Code Searching
Journal of the ACM, 28 (2): 289–304, 1981.

[3] Leo J. Guibas
The Analysis of Hashing Techniques That Exhibit k-ary Clustering
Journal of the ACM, 25 (4): 544–555, 1978.

Hash Tables & Cuckoo Hashing

[1] Donald E. Knuth
The Art of Computer Programming. Volume 3: Sorting and Searching (2nd ed)
Addison-Wesley, Reading, MA, 1998.

[2] Rasmus Pagh and Flemming F. Rodler
Cuckoo Hashing
Journal of Algorithms 51:122-144, 2004

Hash Tables & Cuckoo Hashing

[1] L. Devroye and P. Morin
Cucko hashing: further Analysis
Information Proc. Letters 86:215–219, 2003

[2] M. Drmota and R. Kutzelnigg
A precise analysis of Cuckoo Hashing
ACM Transactions on Algorithms 8(2):1–36, 2012

Universal Hashing

[1] L. Carter and M.N. Wegman.
Universal Classes of Hash Functions.
Journal of Computer and System Sciences, 18 (2): 143–154, 1979.

[2] R. Motwani and P. Raghavan.
Randomized Algorithms.
Cambridge University Press, 1995.

[3] O. Kaser and D. Lemire.
Strongly universal string hashing is fast.
Computer Journal (published on-line in 2013)

Bloom Filters

[1] M. Mitzenmacher and E. Upfal.
Probability and computing: Randomized algorithms and probabilistic analysis.
Cambridge University Press, 2005.

[2] B.H. Bloom.
Space/Time Trade-offs in Hash Coding with Allowable Errors.
Communications of the ACM 13 (7): 422–426, 1970.

[3] A. Broder and M. Mitzenmacher.
Network Applications of Bloom Filters: A Survey
Internet Mathematics 1 (4):485–509, 2003.

Bloom Filters

[1] P. Bose, H. Guo, E.Kranakis et al.
On the False-Positive Rate of Bloom Filters
Information Processing Letters 108 (4):210–213, 2004.

[2] K. Christensen, A. Roginsky and M. Jimeneo.
A New Analysis of the False-Positive Rate of a Bloom Filter
Information Processing Letters 110 (21):944–949, 2010.