The Dictionary Problem: Randomized Approaches

Francesco De Felice Alessandro Lenzi

Cynny S.p.A

December 14, 2016

Outline

Outline

Dictionary Problem

Problem

Given a set of n objects drawn from a universe **U** that can be uniquely identified by a certain key k, we want to be able to:

- *Search*
- *Insert*
- *Delete*

elements from a data structure **D** in a efficient manner. If only search is supported, the data structure **D** is *static*, otherwise it is *dynamic*.

In the rest of the presentation we will refer only to the *keys* and consider them positive integers, without loss of generality.

Any Idea?

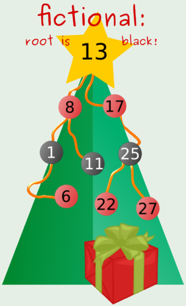
C++ Map

- Should *only* be used in case you need to store *sorted* elements.
- Usually implemented with a red-black tree [3]
- **Insert:** $O(log_2 n)$; **Search:** $O(log_2 n)$; **Delete:** $O(log_2 n)$;
- Efficient for ordered traversal
- Pointers are not particularly cache friendly!

Data structures for the dictionary problem

C++ Map

- Should *only* be used in case you need to store *sorted* elements.
- Usually implemented with a red-black tree [3]
- **Insert:** $O(log_2 n)$; **Search:** $O(log_2 n)$; **Delete:** $O(log_2 n)$;
- Efficient for ordered traversal
- Pointers are not particularly cache friendly!



Data structures for the dictionary problem

C++ Map

- Should *only* be used in case you need to store *sorted* elements.
- Usually implemented with a red-black tree [3]
- **Insert:** $O(log_2 n)$; **Search:** $O(log_2 n)$; **Delete:** $O(log_2 n)$;
- Efficient for ordered traversal
- Pointers are not particularly cache friendly!

C++ Unordered Map

- Good to store elements when no ordering and no traversal is needed
- Rumours say that it is implemented using *hash tables* with chaining for conflict resolution

Data structures for the dictionary problem

C++ Map

- Should *only* be used in case you need to store *sorted* elements.
- Usually implemented with a red-black tree [3]
- **Insert:** $O(log_2 n)$; **Search:** $O(log_2 n)$; **Delete:** $O(log_2 n)$;
- Efficient for ordered traversal
- Pointers are not particularly cache friendly!

C++ Unordered Map

- Good to store elements when no ordering and no traversal is needed
- Rumours say that it is implemented using *hash tables* with chaining for conflict resolution
- **Insert:** $O(1)$; **Search:** $O(n)$; **Delete:** $O(n)$

Data structures for the dictionary problem

C++ Map

- Should *only* be used in case you need to store *sorted* elements.
- Usually implemented with a red-black tree [3]
- **Insert:** $O(log_2 n)$; **Search:** $O(log_2 n)$; **Delete:** $O(log_2 n)$;
- Efficient for ordered traversal
- Pointers are not particularly cache friendly!

C++ Unordered Map

- Good to store elements when no ordering and no traversal is needed
- Rumours say that it is implemented using *hash tables* with chaining for conflict resolution
- **Insert:** $O(1)$; **Search:** $O(n)$; **Delete:** $O(n)$ **worst case**

Outline

A quick recap on hash tables

Naive solution to the Dictionary Problem: Direct-Address tables

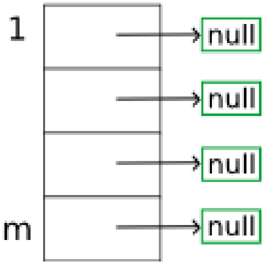
Given a set of K *possible* keys, allocate an array of size $|K|$. If an element with key x exists, it is in position x. Cost is always $O(1)$ for any given operation; space is proportional to $|K|$, so is huge.

Hashing

- We start from a function $h : U \rightarrow H$, such that $|H| << |U|$; let $m = |H|$
- We create a table of m buckets. Bucket i will *logically* contain all elements $\{k_j : h(k_j) = i\}$
- It is not possible to grant that there are not two keys k, k' s.t. $h(k) = h(k')$.
- Depending on the way in which we solve the *collision* we have different kinds of hash tables.

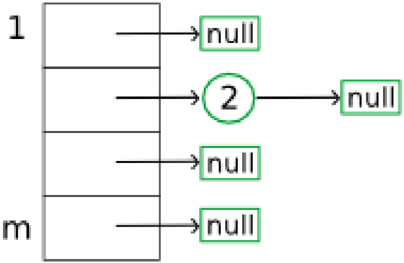
Collision resolution with chaining

- Simplest collision resolution strategy
- Every bucket is a *list*
- Elements are added in $O(1)$, but we use pointers to percolate the list in case of collisions



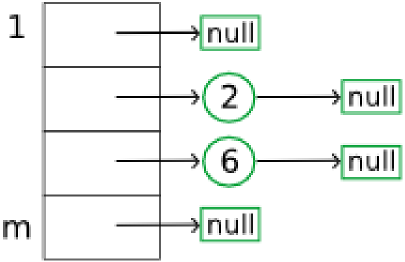
Collision resolution with chaining

- Simplest collision resolution strategy
- Every bucket is a *list*
- Elements are added in $O(1)$, but we use pointers to percolate the list in case of collisions



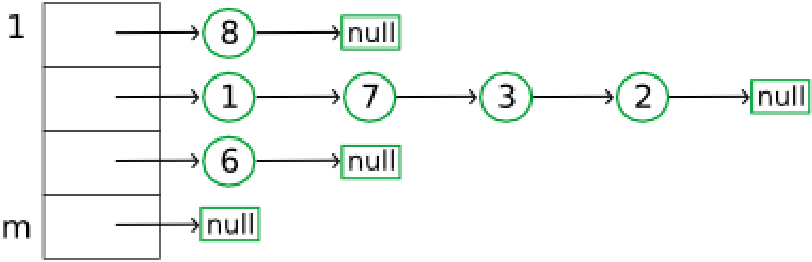
Collision resolution with chaining

- Simplest collision resolution strategy
- Every bucket is a *list*
- Elements are added in $O(1)$, but we use pointers to percolate the list in case of collisions



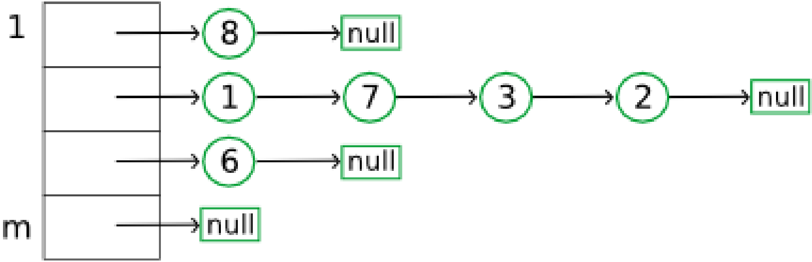
Collision resolution with chaining

- Simplest collision resolution strategy
- Every bucket is a *list*
- Elements are added in $O(1)$, but we use pointers to percolate the list in case of collisions



Collision resolution with chaining

- Simplest collision resolution strategy
- Every bucket is a *list*
- Elements are added in $O(1)$, but we use pointers to percolate the list in case of collisions



- In this case the worst-case cost for a search is $O(n)$. Expected cost is $1 + \frac{n}{m}$ accesses, still constant as long as $m = \Theta(n)$
- Lots of cache-misses.

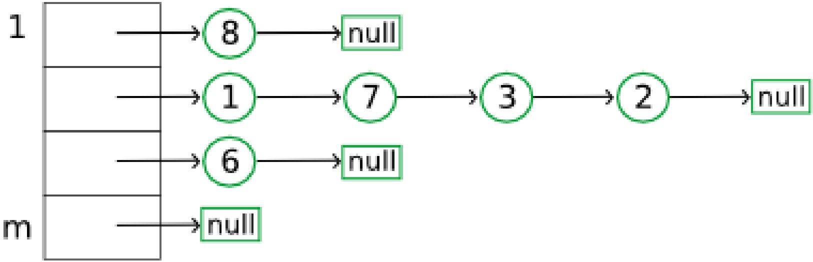
Re-hashing

Problem

For any given h, there exists a sequence $k_1, \ldots, k_n$ of keys s.t. $i, h(k_i) = z$ constant.

Solution

Re-hash! If the maximum number of collisions in D is higher than the threshold, throw D away and re-build from scratch.



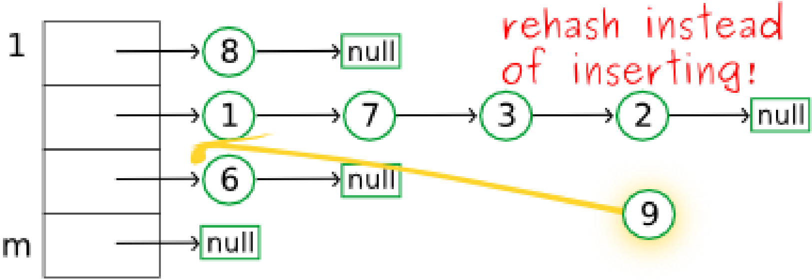
Re-hashing

Problem

For any given h, there exists a sequence $k_1, \dots, k_n$ of keys s.t. $i, h(k_i) = z$ constant.

Solution

Re-hash! If the maximum number of collisions in D is higher than the threshold, throw D away and re-build from scratch.



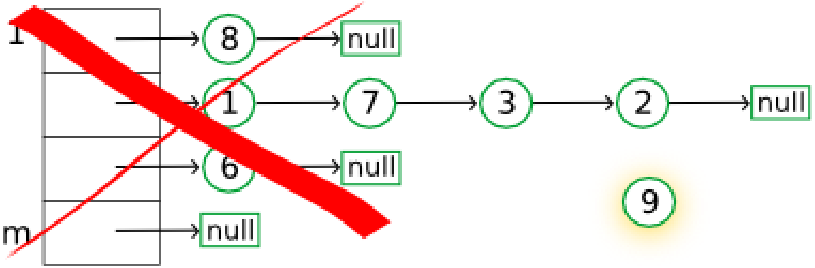
Re-hashing

Problem

For any given h, there exists a sequence $k_1, \ldots, k_n$ of keys s.t. $i, h(k_i) = z$ constant.

Solution

Re-hash! If the maximum number of collisions in D is higher than the threshold, throw D away and re-build from scratch.



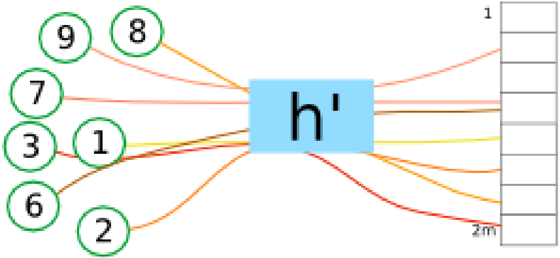
Re-hashing

Problem

For any given h, there exists a sequence $k_1, \ldots, k_n$ of keys s.t. $i, h(k_i) = z$ constant.

Solution

Re-hash! If the maximum number of collisions in D is higher than the threshold, throw D away and re-build from scratch.



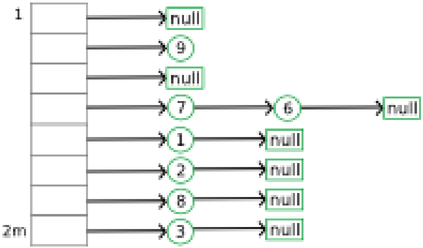
Re-hashing

Problem

For any given h, there exists a sequence $k_1, \ldots, k_n$ of keys s.t. $i, h(k_i) = z$ constant.

Solution

Re-hash! If the maximum number of collisions in D is higher than the threshold, throw D away and re-build from scratch.



Re-hashing

Problem

For any given h, there exists a sequence $k_1, \ldots, k_n$ of keys s.t. $i, h(k_i) = z$ constant.

Solution

Re-hash! If the maximum number of collisions in D is higher than the threshold, throw D away and re-build from scratch.

- Less costly search and delete operation (on average)
- More costly insert: in the worst case it becomes $\Theta(n)$.
- However, the probability of rehashing decreases exponentially
- With this solution, the amortized costs become:
  - Insert: $O(1)$; Search: $O(1)$; Delete: $O(1)$
- If we have a "good" hash function, by re-hashing we can find an hash table with $O(n)$ space cost and constant time for all operations.
  - C++'s `unordered_map` re-hashes once `max_load_factor` is > 1
  - Try setting it to 64 and see how things go... ;)

So, what is the cost of pushing back into a dynamic array (std::vector)?

... $O(n)$

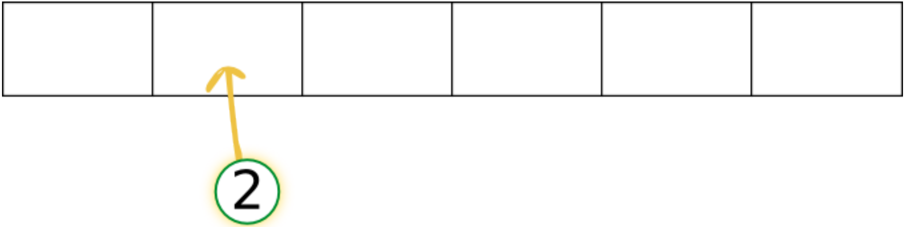
Amortized analysis

Amortized Cost

- The amortized cost is $O(1)$ because the periodic copy ($\Theta(n)$) is executed every n insertions.
- Think it like a factory buying a new, faster machine: in the long time, the higher cost is compensated by the benefits arising from its usage.
- Sometimes an algorithm incurs in large overheads in some operations in order to optimize the more frequent ones. [4]
- Re-hashing uses the same mechanisms: makes some insert operations more costly, with advantages for search and delete.
- Don't use this kind of reasoning if you need predictability in terms of single operation speed!

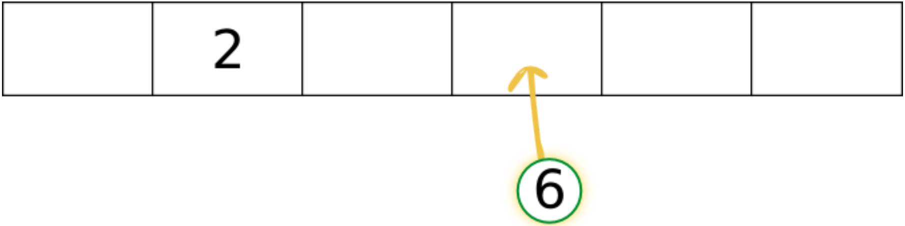
Collision resolution with open addressing

- Rather than storing pointers, we store directly the values in the bucket
- Conflicts are solved by *probing* the table for free positions
- Probing can be **linear**, **quadratic** or through **double-hashing**
- **Insert:** $O(n)$; **Search:** $O(n)$; **Delete:** $O(n)$
- Efficiency decreases *dramatically* with load factor!
- Also in this case, we use re-hashing to grant good amortized costs.



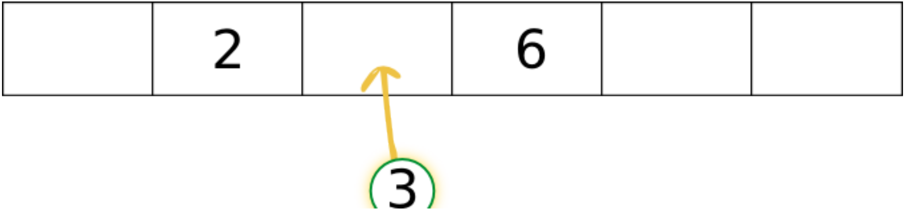
Collision resolution with open addressing

- Rather than storing pointers, we store directly the values in the bucket
- Conflicts are solved by *probing* the table for free positions
- Probing can be **linear**, **quadratic** or through **double-hashing**
- **Insert:** $O(n)$; **Search:** $O(n)$; **Delete:** $O(n)$
- Efficiency decreases *dramatically* with load factor!
- Also in this case, we use re-hashing to grant good amortized costs.



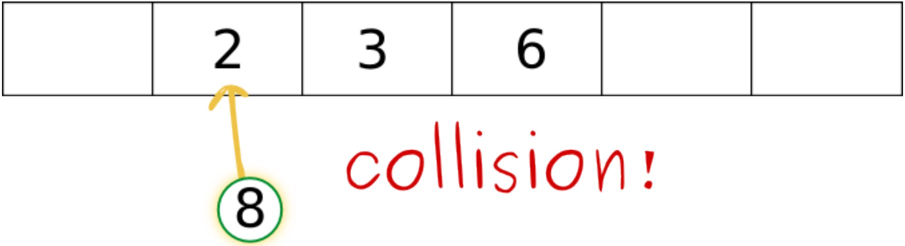
Collision resolution with open addressing

- Rather than storing pointers, we store directly the values in the bucket
- Conflicts are solved by *probing* the table for free positions
- Probing can be **linear**, **quadratic** or through **double-hashing**
- **Insert:** $O(n)$; **Search:** $O(n)$; **Delete:** $O(n)$
- Efficiency decreases *dramatically* with load factor!
- Also in this case, we use re-hashing to grant good amortized costs.



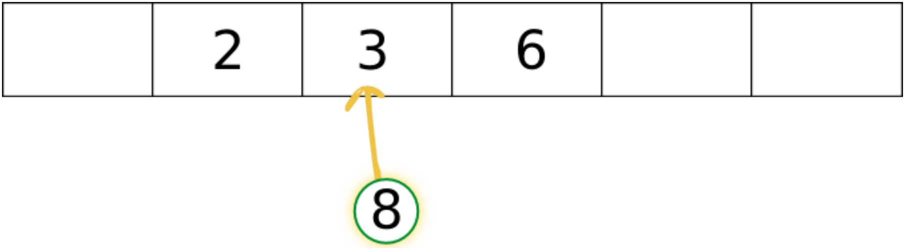
Collision resolution with open addressing

- Rather than storing pointers, we store directly the values in the bucket
- Conflicts are solved by *probing* the table for free positions
- Probing can be **linear**, **quadratic** or through **double-hashing**
- **Insert:** $O(n)$; **Search:** $O(n)$; **Delete:** $O(n)$
- Efficiency decreases *dramatically* with load factor!
- Also in this case, we use re-hashing to grant good amortized costs.



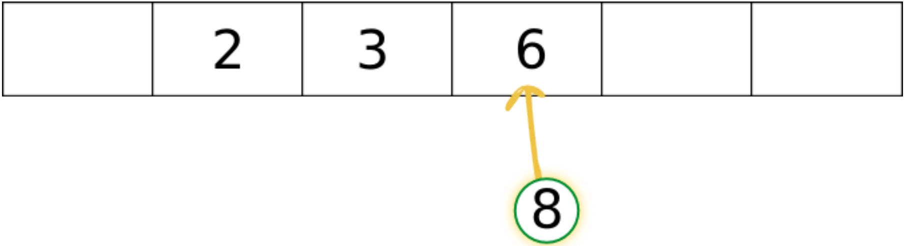
Collision resolution with open addressing

- Rather than storing pointers, we store directly the values in the bucket
- Conflicts are solved by *probing* the table for free positions
- Probing can be **linear**, **quadratic** or through **double-hashing**
- **Insert:** $O(n)$; **Search:** $O(n)$; **Delete:** $O(n)$
- Efficiency decreases *dramatically* with load factor!
- Also in this case, we use re-hashing to grant good amortized costs.



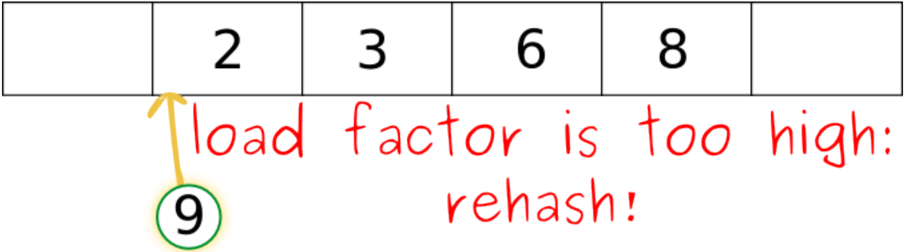
Collision resolution with open addressing

- Rather than storing pointers, we store directly the values in the bucket
- Conflicts are solved by *probing* the table for free positions
- Probing can be **linear**, **quadratic** or through **double-hashing**
- **Insert:** $O(n)$; **Search:** $O(n)$; **Delete:** $O(n)$
- Efficiency decreases *dramatically* with load factor!
- Also in this case, we use re-hashing to grant good amortized costs.



Collision resolution with open addressing

- Rather than storing pointers, we store directly the values in the bucket
- Conflicts are solved by *probing* the table for free positions
- Probing can be **linear**, **quadratic** or through **double-hashing**
- **Insert:** $O(n)$; **Search:** $O(n)$; **Delete:** $O(n)$
- Efficiency decreases *dramatically* with load factor!
- Also in this case, we use re-hashing to grant good amortized costs.



A simple benchmark

- Insert 2^{20} random generated key-value pairs in a dictionary
- Search 2^{21} strings (about 50% belonging to the dictionary)
- Which one, between `std::map`, `std::unordered_map` and a custom `open addressing` will be faster?

A simple benchmark

- Insert 2^{20} random generated key-value pairs in a dictionary
- Search 2^{21} strings (about 50% belonging to the dictionary)
- Which one, between `std::map`, `std::unordered_map` and a custom `open addressing` will be faster?

- **map:** 55,771s; 614.245.753 cache misses

A simple benchmark

- Insert 2^{20} random generated key-value pairs in a dictionary
- Search 2^{21} strings (about 50% belonging to the dictionary)
- Which one, between `std::map`, `std::unordered_map` and a custom `open addressing` will be faster?

- **map:** 55,771s; 614.245.753 cache misses
- **unordered map:** 49,1442s; 291.645.582 cache misses

A simple benchmark

- Insert 2^{20} random generated key-value pairs in a dictionary
- Search 2^{21} strings (about 50% belonging to the dictionary)
- Which one, between `std::map`, `std::unordered_map` and a custom `open addressing` will be faster?

- **map:** 55,771s; 614.245.753 cache misses
- **unordered map:** 49,1442s; 291.645.582 cache misses
- **openaddressing map:** 28,7577s; 272.400.800 cache misses

A simple benchmark

- Insert 2^{20} random generated key-value pairs in a dictionary
- Search 2^{21} strings (about 50% belonging to the dictionary)
- Which one, between `std::map`, `std::unordered_map` and a custom `open addressing` will be faster?

- **map:** 55,771s; 614.245.753 cache misses
- **unordered map:** 49,1442s; 291.645.582 cache misses
- **openaddressing map:** 28,7577s; 272.400.800 cache misses

Open-addressing makes less cache faults, hence improving performances notably.

d-left Hashing

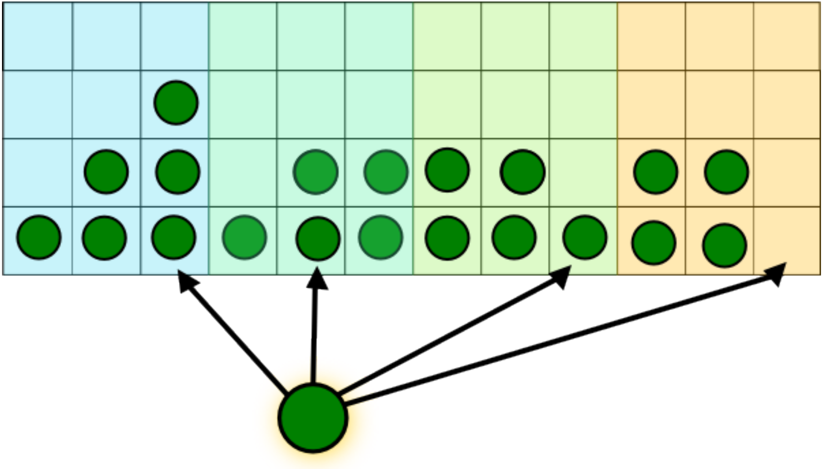
The power of d-choices [5]

- By having multiple choice we can improve *balancing*
- This will reduce the cost of searching, inserting and deleting even in case of collisions
- Multiple alternatives mean that we can select the best one

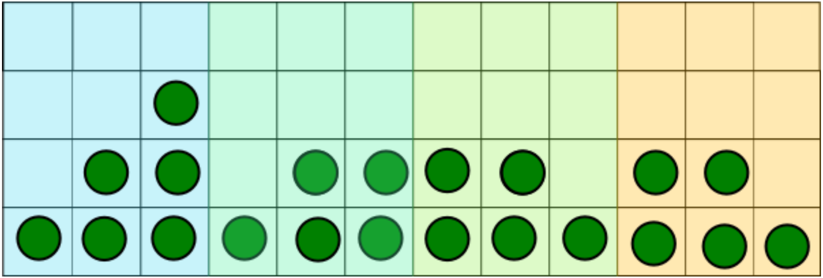
d-left hashing

- Use d hash functions, and d sub-tables of size $\frac{m}{d}$
- We define a different hash function h_i for each of the sub-tables
- To insert, each sub-table's load factor is tested and the elements are inserted in the sparsest one.
- In case of a tie between all elements, the one with lowest i is selected
- Every bucket can contain multiple element

d-left hashing example



d-left hashing example



Cuckoo hashing

Problem

Dynamic Dictionary Problem.

We may use **cuckoo hashing** with the following features:

- *insert(x)*: $O(1)$ amortized time;
- *search(x)* and *delete(x)*: $O(1)$ time, in the worst case;
- power of 2-choices combined with keys movement

.

Cuckoo hashing

Problem

Dynamic Dictionary Problem.

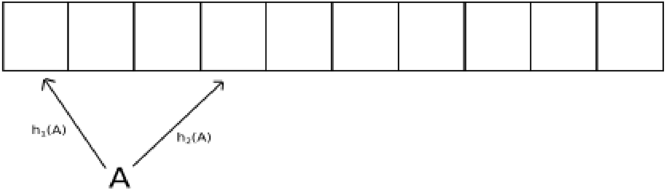
We may use **cuckoo hashing** with the following features:

- *insert(x)*: $O(1)$ amortized time;
- *search(x)* and *delete(x)*: $O(1)$ time, in the worst case;
- power of 2-choices combined with keys movement

.

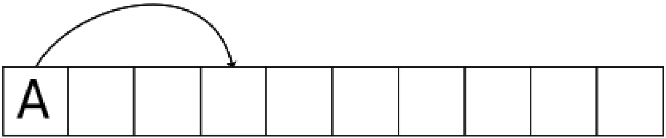
A running example using a Cuckoo Graph

Given two hash functions, $h_1(x)$ and $h_2(x)$, and a hash table T we may simulate *insertion*(x) using a **Cuckoo Graph**.



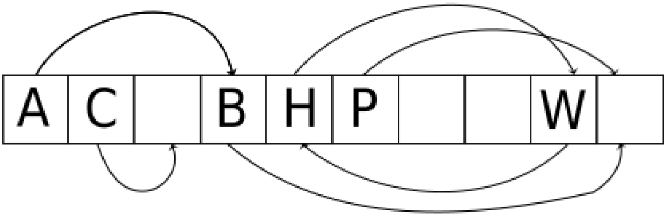
A running example using a Cuckoo Graph

Given two hash functions, $h_1(x)$ and $h_2(x)$, and a hash table T we may simulate *insertion*(x) using a **Cuckoo Graph**.



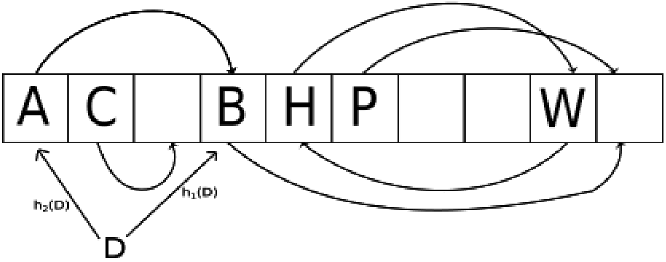
A running example using a Cuckoo Graph

Given two hash functions, $h_1(x)$ and $h_2(x)$, and a hash table T we may simulate *insertion*(x) using a **Cuckoo Graph**.



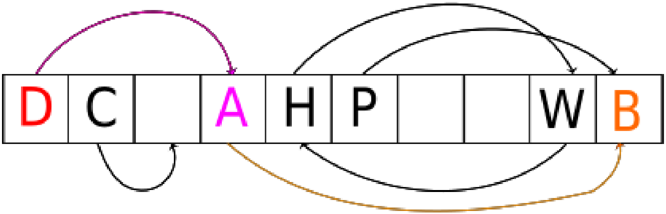
A running example using a Cuckoo Graph

Given two hash functions, $h_1(x)$ and $h_2(x)$, and a hash table T we may simulate *insertion*(x) using a **Cuckoo Graph**.



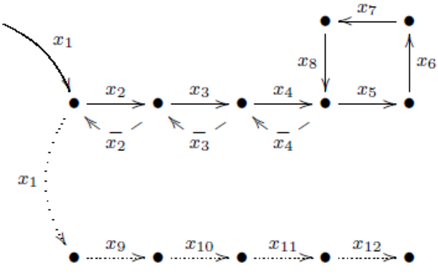
A running example using a Cuckoo Graph

Given two hash functions, $h_1(x)$ and $h_2(x)$, and a hash table T we may simulate *insertion*(x) using a **Cuckoo Graph**.



Cycles in the Cuckoo Graph

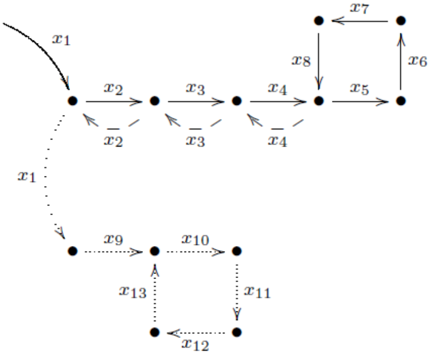
Cuckoo Graph may contain *cycles*.



(b) one cycle

Cycles in the Cuckoo Graph

Cuckoo hashing may fail in case of **infinite loops** in the Cuckoo Graph.



In this case the insertion procedure fails and *rehashing* has to be performed with another pair of (hopefully better) hash functions.

Escaping from Cycles

The *escaping condition* to detect a cycle consists in *approximately* bounding the number of evictions...i.e. path-length in the graph.

Theoretical Solution

Loop $O(\log n)$ times (i.e. path-length) and *rehash* everything in case of failures.

Insertion cost

Fact

With a load less than 50% (i.e. $\frac{m}{2} > n$) the average failure rate is $\Theta(\frac{1}{n})$.

Given that rehashing time is $O(n)$ (two hash values computation for every item) then the insertion amortized cost is $O(1)$.

Outline

Motivation

Problem

We want to know if an element belongs or not to a certain dictionary

- Often a dictionary could not fit in memory
- Fetching pages from disk or from a remote location could be very costly
- We want a *succint* representation of the elements really contained in the set
- This allows to access a costly memory hierarchy only if needed.

As Bloom said

Wherever a list or set is used, and space is a consideration, a Bloom Filter should be considered. When using a Bloom Filter, consider the effects of false positives.

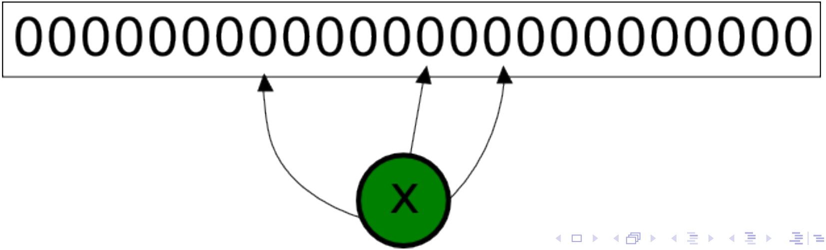
Bloom Filters: the idea

- We use a *fingerprint* to represent a key
- Rather than calculating it and storing it explicitly, we use k hash functions
- We use a vector of m bits (zeroed in the beginning) to store the fingerprints
- When an element with key x is inserted, we just calculate $h_1(x), \ldots, h_k(x)$ and set the bits in the corresponding positions to 1.

Bloom Filters

Bloom Filters: the idea

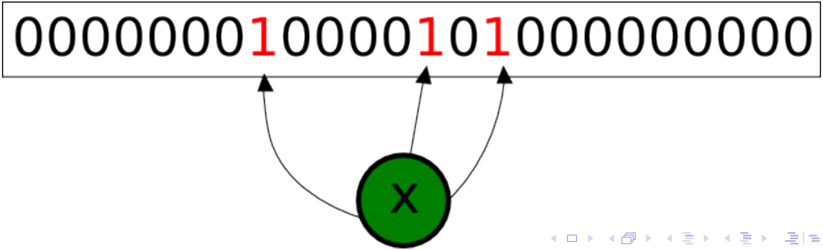
- We use a *fingerprint* to represent a key
- Rather than calculating it and storing it explicitly, we use k hash functions
- We use a vector of m bits (zeroed in the beginning) to store the fingerprints
- When an element with key x is inserted, we just calculate $h_1(x), \dots, h_k(x)$ and set the bits in the corresponding positions to 1.



Bloom Filters

Bloom Filters: the idea

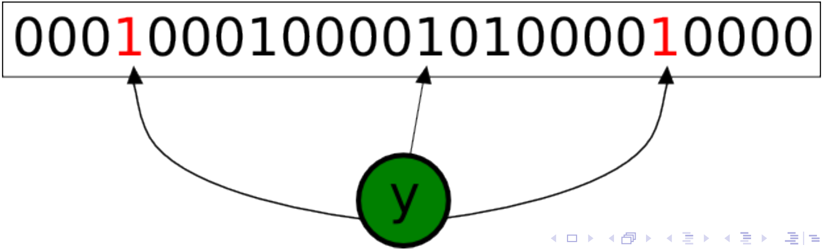
- We use a *fingerprint* to represent a key
- Rather than calculating it and storing it explicitly, we use k hash functions
- We use a vector of m bits (zeroed in the beginning) to store the fingerprints
- When an element with key x is inserted, we just calculate $h_1(x), \ldots, h_k(x)$ and set the bits in the corresponding positions to 1.



Bloom Filters

Bloom Filters: the idea

- We use a *fingerprint* to represent a key
- Rather than calculating it and storing it explicitly, we use k hash functions
- We use a vector of m bits (zeroed in the beginning) to store the fingerprints
- When an element with key x is inserted, we just calculate $h_1(x), \ldots, h_k(x)$ and set the bits in the corresponding positions to 1.



Bloom Filter operations

- **Insertion:** the cost is $O(k)$; k is practically a constant, hence $O(1)$
- **Membership:**
  - y *can be* in the set if $i = 1, \dots, k, BF[h_i(y)] = 1$
  - The cost of this operation is once again $O(k)$
- **Deletion:** *is not possible*, because of collisions.

False Positives

A false positive arises if, for a certain element y, $BF[h_1(y)] = 1 \dots BF[h_2(y)] = 1$ even though $y \notin S$

- After $|S|$ is n, the probability that a given bit $BF[i]$ is 0 can be approximated to $e^{-kn/m} = p$
- The overall probability of false positive is hence $(1 - p)^k$
- Changing k and m we can make this probability albitrarily small.

Counting Bloom Filters

Overcoming the limitations of Bloom Filters

- The main limitation of Bloom Filters is that no elements can be deleted from the set it represents
- In case of deletion, we must accept to look up (even in case of failure) or rebuild the Bloom Filter

Counting Bloom Filters

- A slight variation of Bloom Filters, where we use c bits rather than 1
- When an element is inserted, we *increment* the c bits corresponding to the positions of the k hashes
- When we want to remove an element, we *decrement* the bits
- At the cost of a factor c in space, we can operate on *dynamic sets*

A real application

Doormat's Cache

- Doormat's cache must store order of millions of elements and reply in a very fast manner
- Elements' keys are URIs, strings whose size is in the order of 1000 chars
- Values are stored on disk, keys are kept in main memory

Performance gain from using Counting Bloom Filters

Benchmark executed using real URLs as collected from our balancers.
p is the probability that an element is found in cache. On *has* operation:

- With $p = 1$, CBF increases time 97%
- With $p = 0.5$, CBF gives about 15% gain
- With $p = 0.2$, CBF gives about 22% gain
- With $p = 0$, CBF gives about 42% gain

Count-Min Sketch: the Count Tracking problem

Problem

Count Tracking problem: given a set with a large number of items and a frequency estimate associated to each one of them, when a query for item x arrives, the answer to the query is the current frequency of x.

Examples:

- a search engine could be interested in queries statistics (e.g. *top-k* list of more frequent queries)
- in network security we could be interested in finding the IP addresses, whose contribution to the network traffic exceeds a given threshold (so called **Heavy-Hitters**).

A bit more formally...

Given:

- a vector $A_t[1, n]$ whose state changes with time t (with $A_0 = \mathbf{0}$)

Then, the updates of an individual entry of A at time t consists of a pair (i_t, c_t) of numbers, so that:

- $A_{t+1}[i_t] = A_t[i] + c_t$ (in the basic case $c_t = 1$)
- $A_{t+1}[i'] = A_t[i']$, if $i' \neq i_t$.

At any time t, a given *query*(i) may arrive asking for $A_t[i]$.

Goal

achieve sub-linear space in n, fast update and query but with frequency estimates that are inevitably (ϵ, δ)-*approximated*, i.e. error is within a factor ϵ with probability δ.

The Count-Min Sketch Data Structure

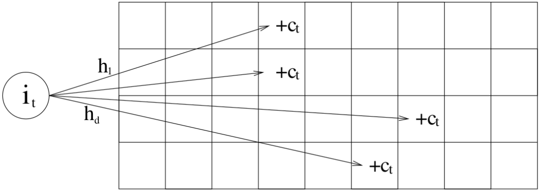
The (ϵ,δ)-Count-Min sketch data structure consists of:

- a matrix $CM[d][w]$ of counters with $d = \lceil \log \frac{1}{\delta} \rceil$ and $w = \lceil \frac{e}{\epsilon} \rceil$
- a set of hash functions $h_1, ..., h_d$ hashing over [1,w], chosen from a *pairwise independent* set, i.e.: $P(h_i(x) = h_j(x)) = \frac{1}{w}$.

Space Occupancy

$O(dw) = O(\frac{e}{\epsilon} \log \frac{1}{\delta})$ which is sub-linear in n.

CM Sketch: Updates and Point Queries



- **Update**(i_t, c_t): for $j = 1, ..., d$ do $CM[j][h_j(i_t)] += c_t$
- **Query**(x): return $\hat{A}[x] = min_j CM[j][h_j(x)]$

CM-Sketch vs Bloom Filters: Error value is bounded not only its probability of occurrence!

Point Queries on positive updates

Theorem

The estimate for item i, where $i = 1, 2, ..., n$ is such that:

- $\hat{A}[i] \geq A[i]$;
- with probability at least $1 - \delta$, it is $\hat{A}[i] \leq A[i] + \epsilon ||A||_1$

with: $||A||_1 = \sum_i^n A[i]$

Time Complexity

Point queries and Updates take $O(d) = O(\log \frac{1}{\delta})$ time.

Thank you! Questions?

For Further Reading I

P. Ferragina.
Chapter 13: The Dictionary Problem
Teaching material of Algorithm Engineering course.

G. Gormode, S. Muthukrishnan
An improved data stream summary: the count-min sketch and its applications
in Journal of Algorithms, Volume 55 Issue 1, Pages 58-75, April 2005.

cppreference - map
cppreference.com page dedicated to std::map

R. E. Tarjan
Amortized computational complexity
SIAM Journal on Algebraic Discrete Methods 6.2 (1985): 306-318

For Further Reading II

MitzenMacher, Richa, Sitaraman
The power of two random choices: a survey of techniques and results
Combinatorial Optimization 9 (2001): 255-304

Broder, Andrei, Mitzenmacher
Network applications of bloom filters: A survey.
Internet mathematics 1.4 (2004): 485-509