P2P Systems and Blockchains

Spring 2018,
instructor: Laura Ricci
laura.ricci@unipi.it

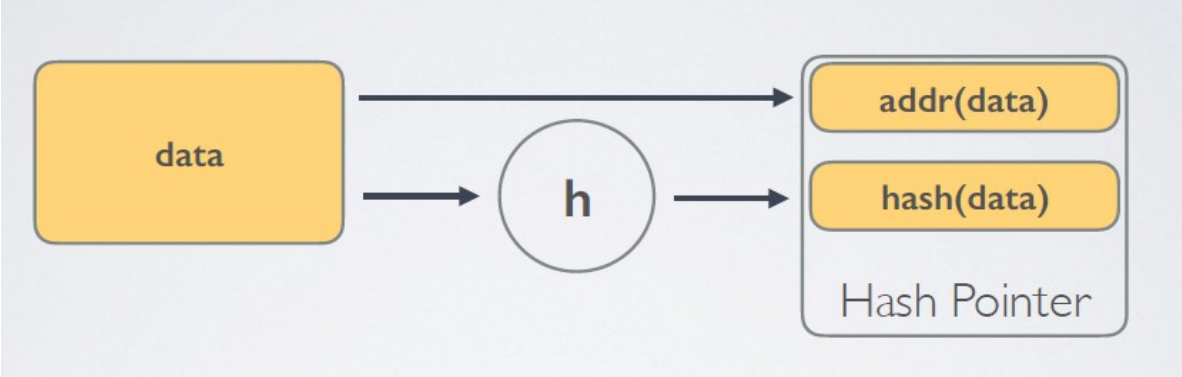
Lesson 12:

DATA STRUCTURE TOOLBOX FOR BLOCKCHAINS

11/4/2018

HASH POINTERS

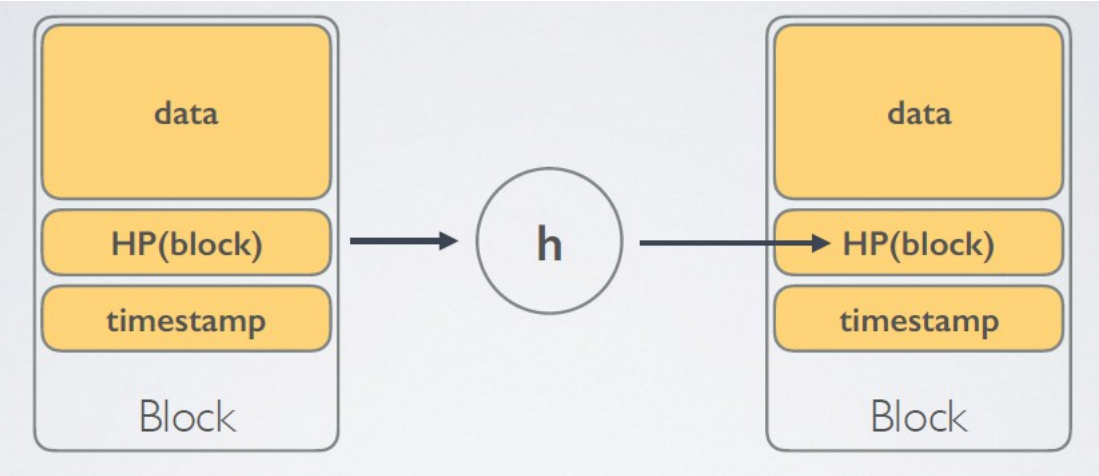
- an hash pointer is:
  - a pointer to where some info is stored
  - a cryptographic hash of the info
- if we have a hash pointer, we can
  - ask to get the info back
  - verify that it hasn't changed



Tamper-evident data pointer = Hash Pointer

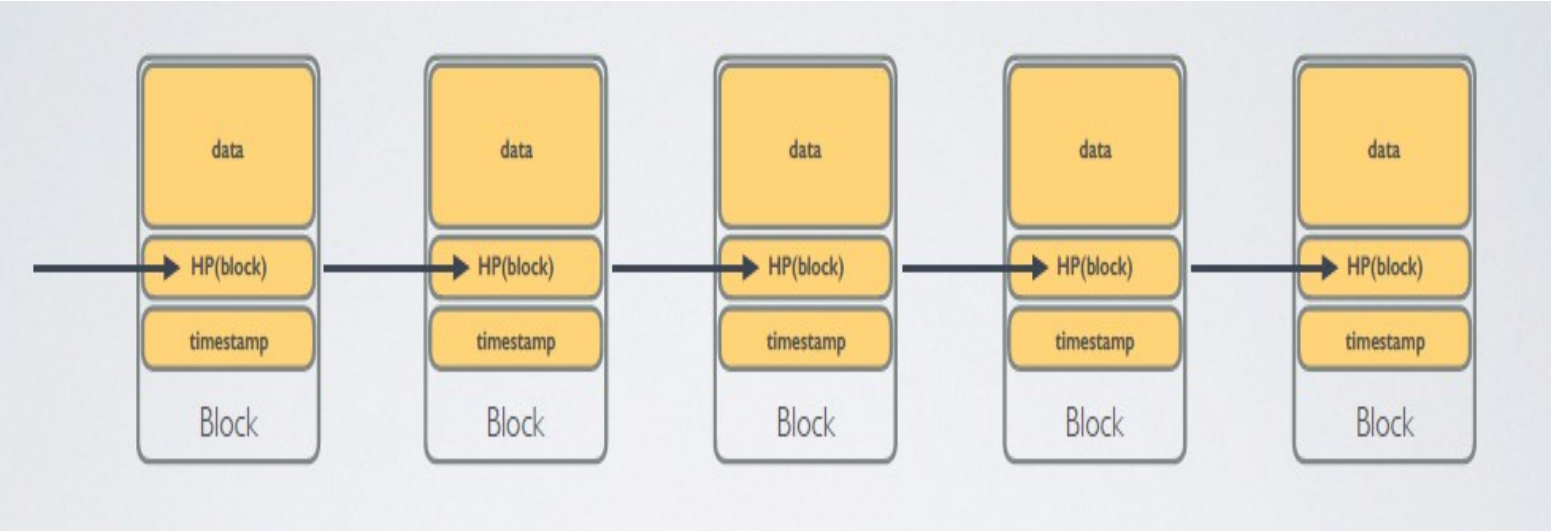
HASH DATA STRUCTURES

- key idea
  - build data structures with hash pointers
- block chain:
  - linked list with hash pointers
    - important: hash the entire block, also its hash pointer!
  - each block tells us where the value of the previous block is located and a digest to verify that the value has not been changed
- use case: tamper-evident log, a basic data structure in Bitcoin



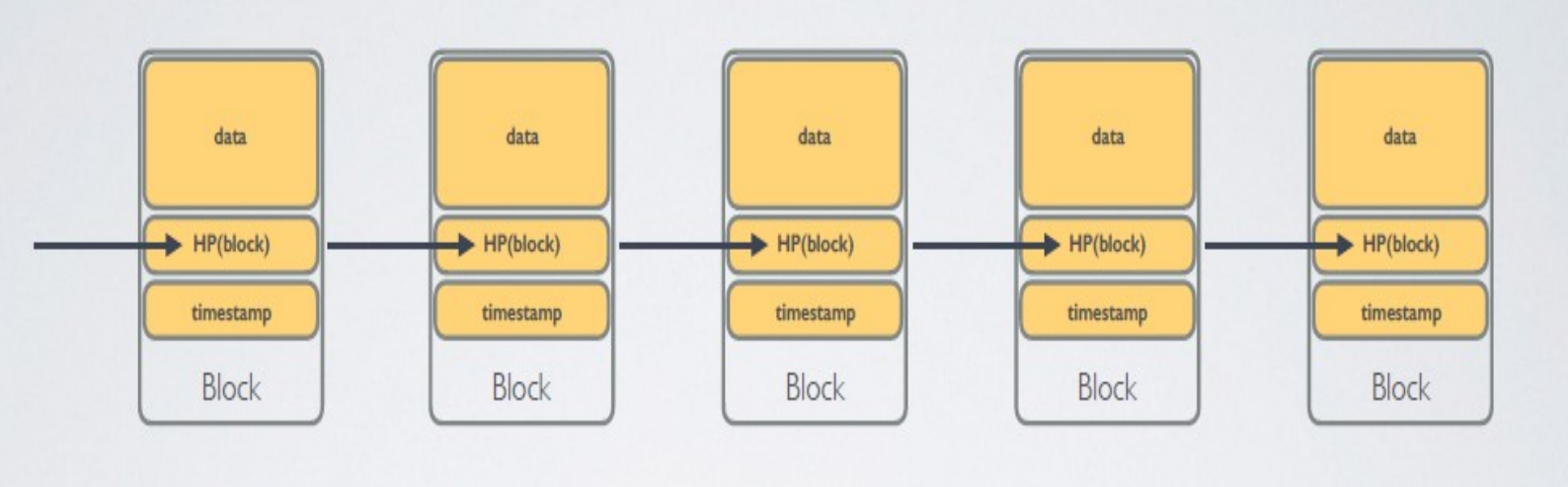
HASH DATA STRUCTURES

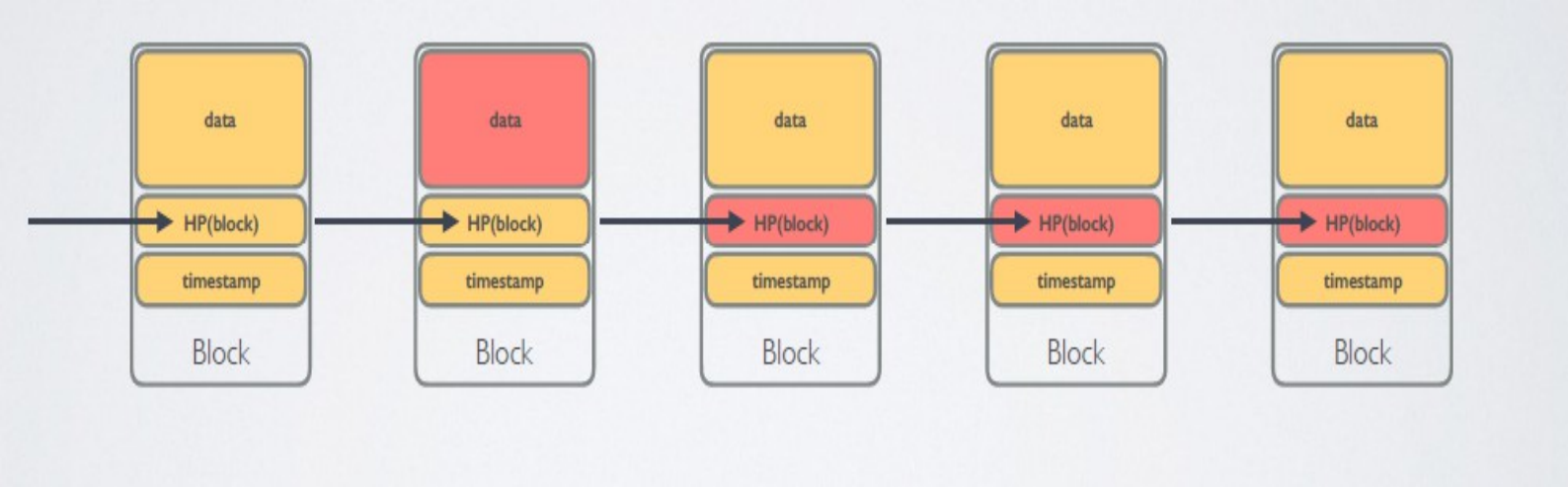
Tamper-evident linked list: blockchain



HASH DATA STRUCTURES

Tamper-evident linked list: blockchain

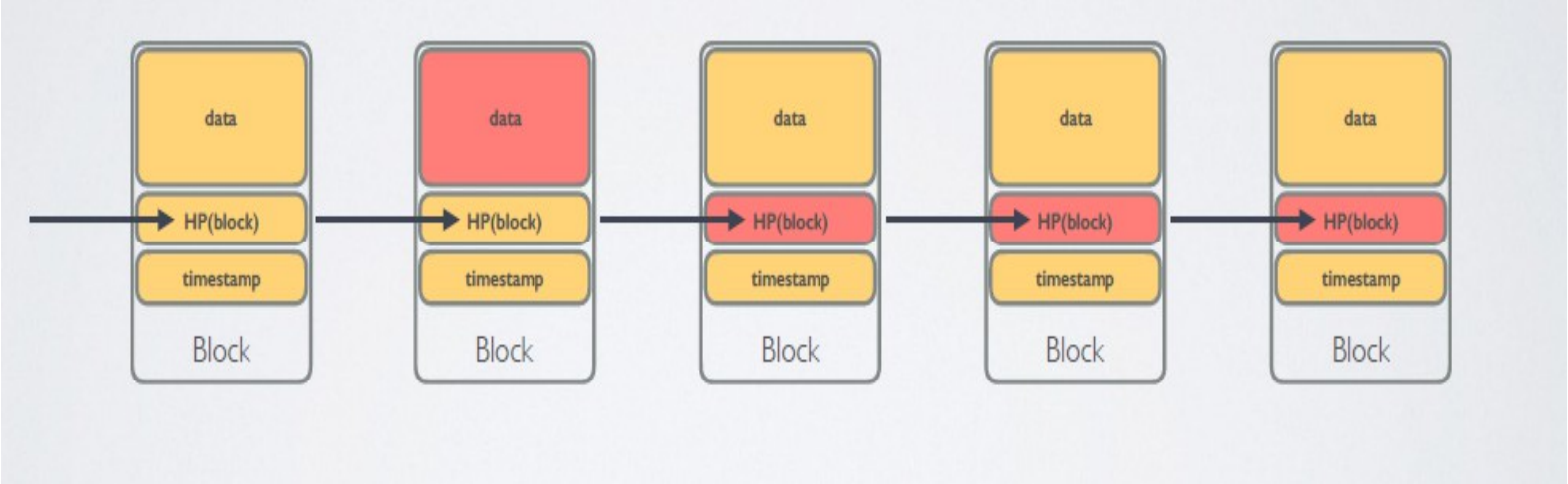




HASH DATA STRUCTURES

- if someone tampers the k-th block of the chain, the hash of block k+1 is not going to match up
- this is because the hash is collision resistant: an adversary cannot tamper the data so that its hash is the same of the data before the tampering

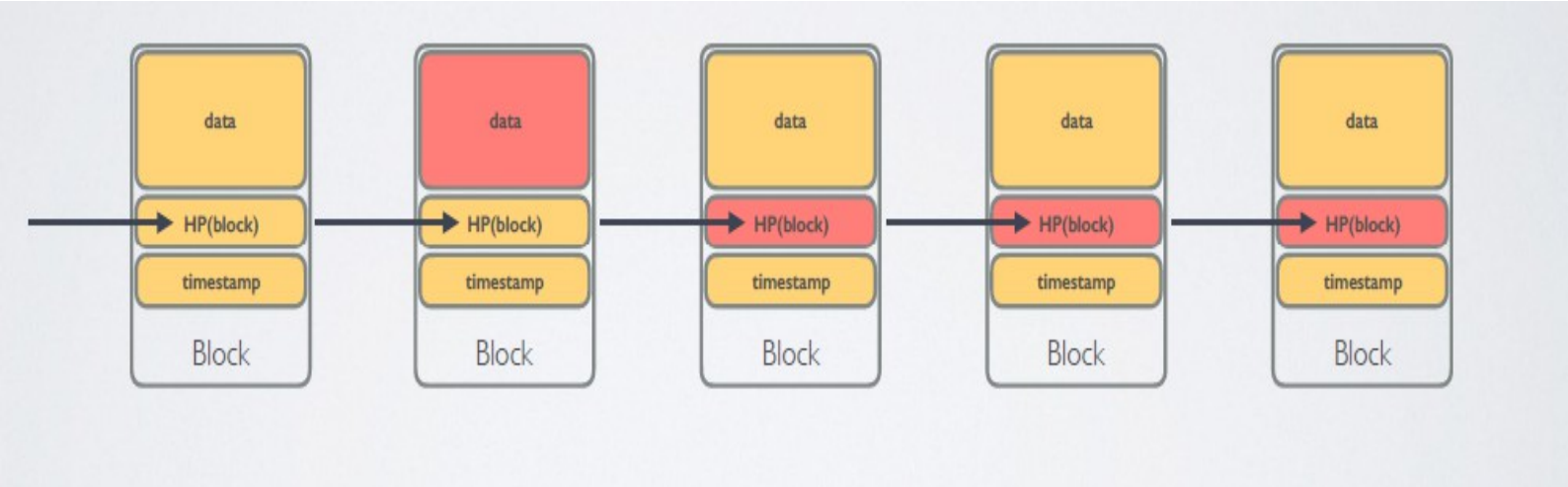
use case: tamper-evident log, a basic data structure in Bitcoin



HASH DATA STRUCTURES

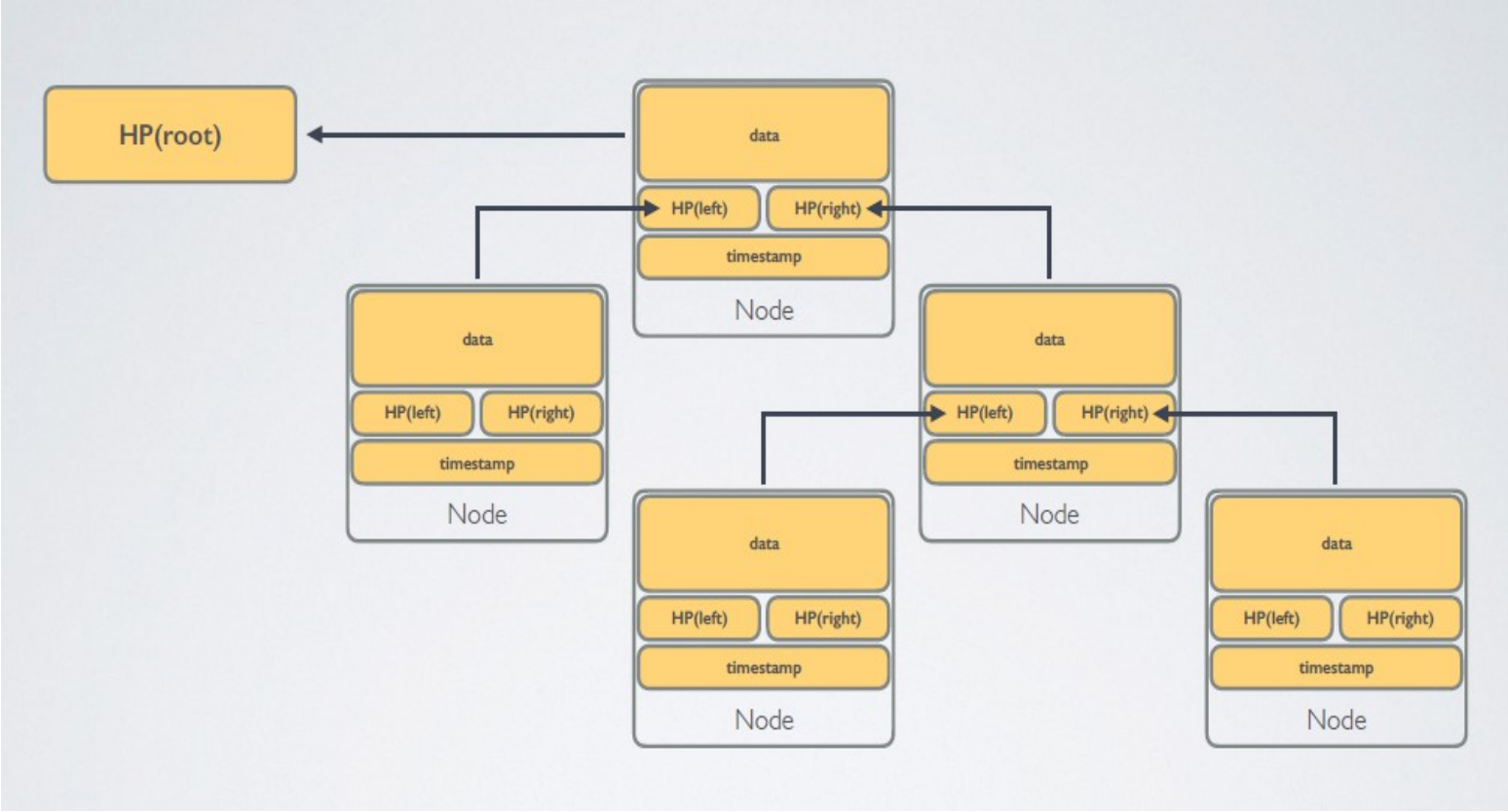
- the adversary can continue to change the next block as well, reaching the head of the list

- The structure is tamper free if
  - the hash pointer at the head of the list is stored in a place that the adversary cannot change, (a trusted location), the adversary is unable to change any block without being detected.
  - the modification of all the subsequent blocks is computationally unfeasible



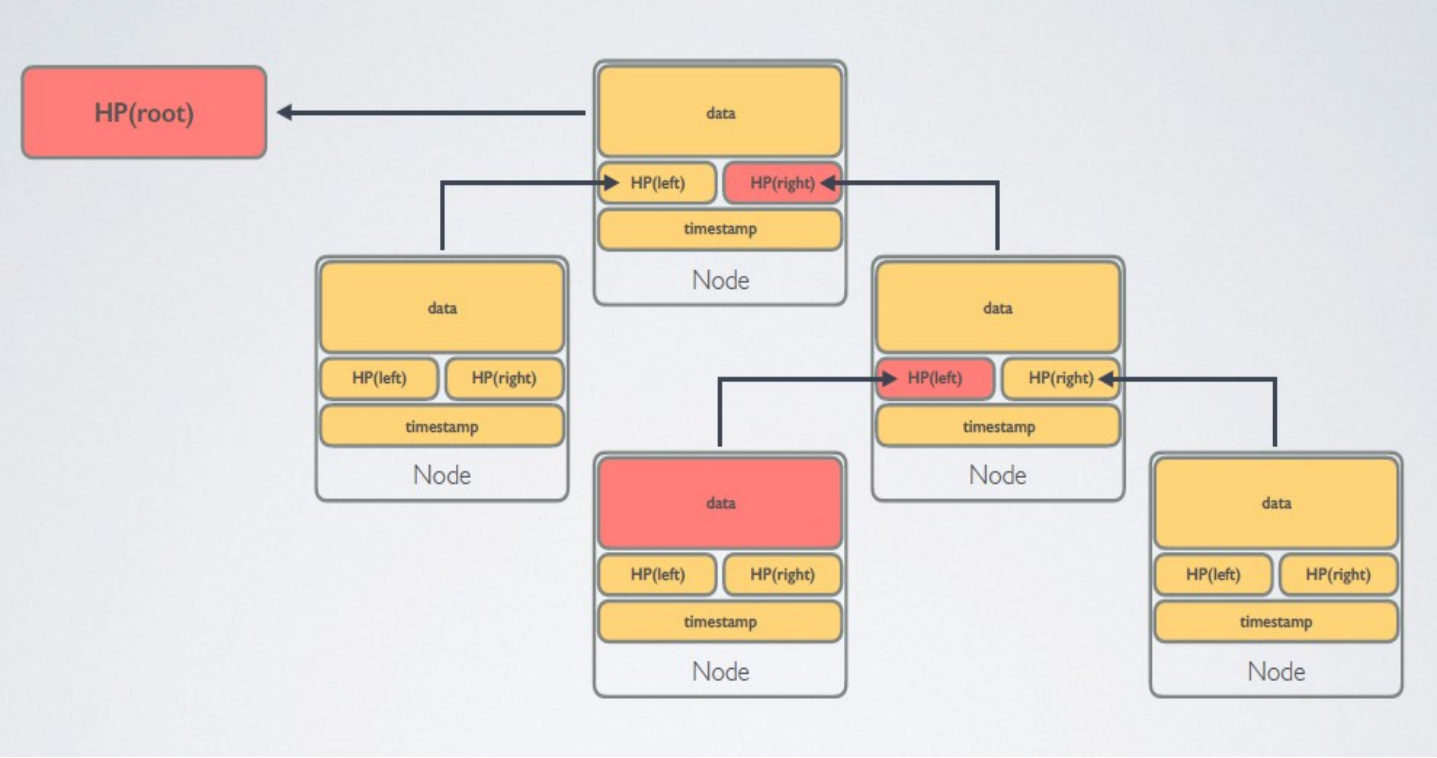
TAMPER EVIDENT BINARY TREES: MERKLE TREES

- binary tree with hash pointers
- remember just the root of the tree in a secure place
- if an adversary tampers some blocks at the bottom of the tree, the hash pointers one level up do nor match, and the so on...



TAMPER EVIDENT BINARY TREES: MERKLE TREES

- binary tree with hash pointers
- remember just the root of the tree in a secure place
- if an adversary tampers some blocks at the bottom of the tree, the hash pointers one level up do nor match, and the so on...

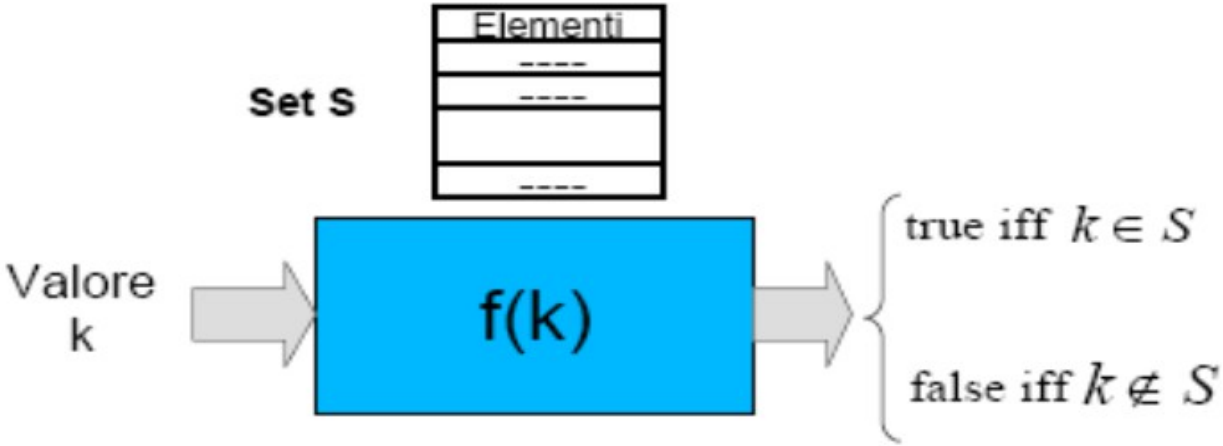


MORE GENERALLY...

- can use hash pointers in any pointer-based data structure that has no cycles, for instance, in a DAG.
- several applications applications:
  - Bitcoin: uses SHA-256 e RIPEMD-160 (double) cryptographic hash functions and the block chain technique
  - Advanced Intelligent Corruption Handling, introduced in eMule from version 0.44:
    - used to check that the block of a file which has been downloaded from the network has not been tampered
    - exploits Merkle trees
  - many other applications....

BLOOM FILTERS: SET MEMBERSHIP PROBLEM

Let us consider the set $S=\{s_1,s_2,...,s_n\}$ of n elements chosen from a very large universe U. Define a data structure supporting queries like "k is an element of S?"



The function f returns value true or false according to the presence of k in the given set

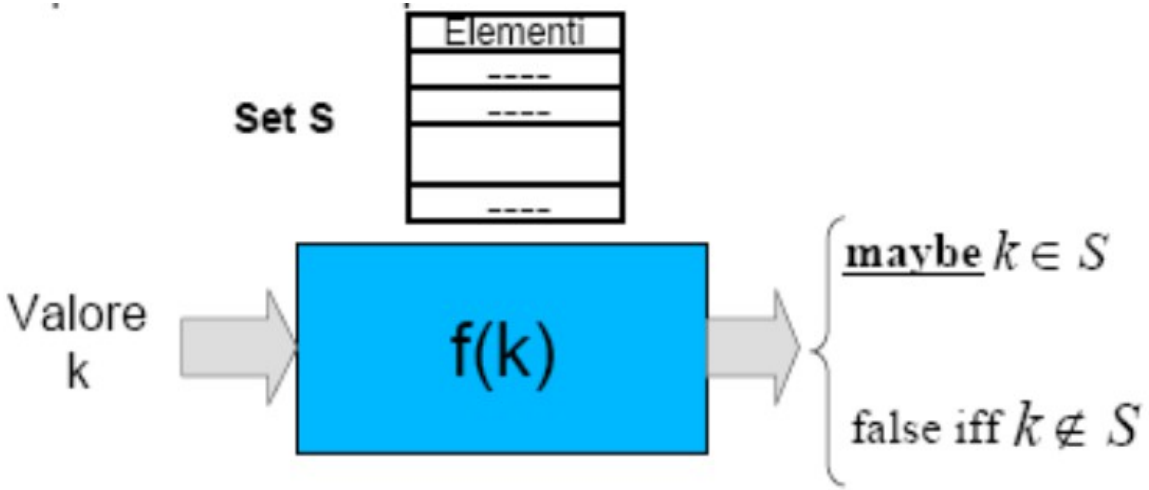
APPROXIMATE SET MEMBERSHIP PROBLEM

- S may be
  - a set of keywords describing the files shared by a peer, selected form the universe of all the keywords (Gnutella 0.6)
  - the set of pieces of file owned by a peer (BitTorrent)
  - light weight (mobile) mobile nodes build Bloom filters with the interesting addresses and send them to the full nodes: bandwith saving (Bitcoin)

- Problem: choose a representation of the elements in S such that:
  - the result of the query is computed efficiently
  - the space for the representation of the elements is reduced
    - to reduce the space required to represent the elements, the results may be approximated
    - possibility of returning false positives

APPROXIMATE SET MEMBERSHIP PROBLEM

An approximate solution to the set membership problem:



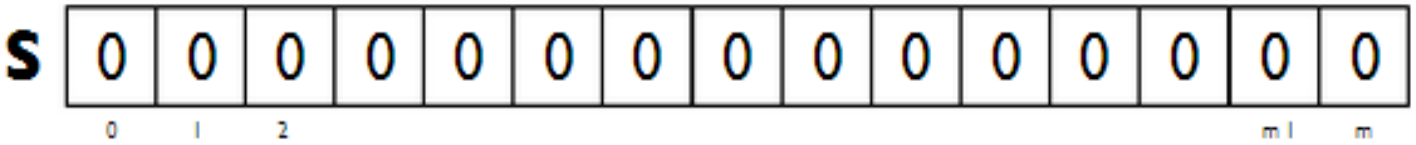
Trade off between:

- Space required
- Probability of false positives

BUILDING BLOOM FILTERS

m bits (initially set to 0)

k hash functions

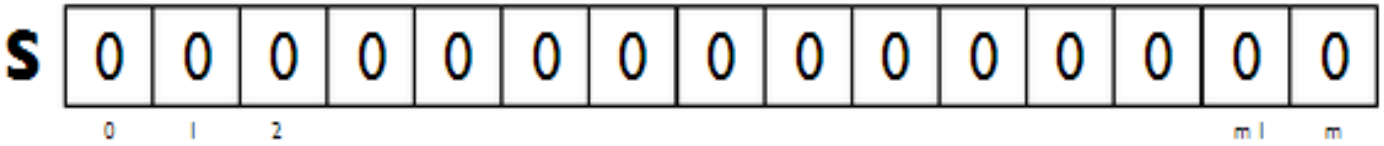


BUILDING BLOOM FILTERS

m bits (initially set to 0)

k hash functions

Add



BUILIDNG BLOOM FILTERS

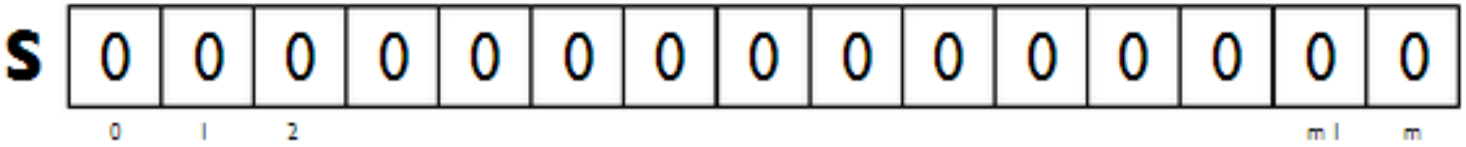
m bits (initially set to 0)

k hash functions

x

if f(x) = A,
set S[A] = 1

Add



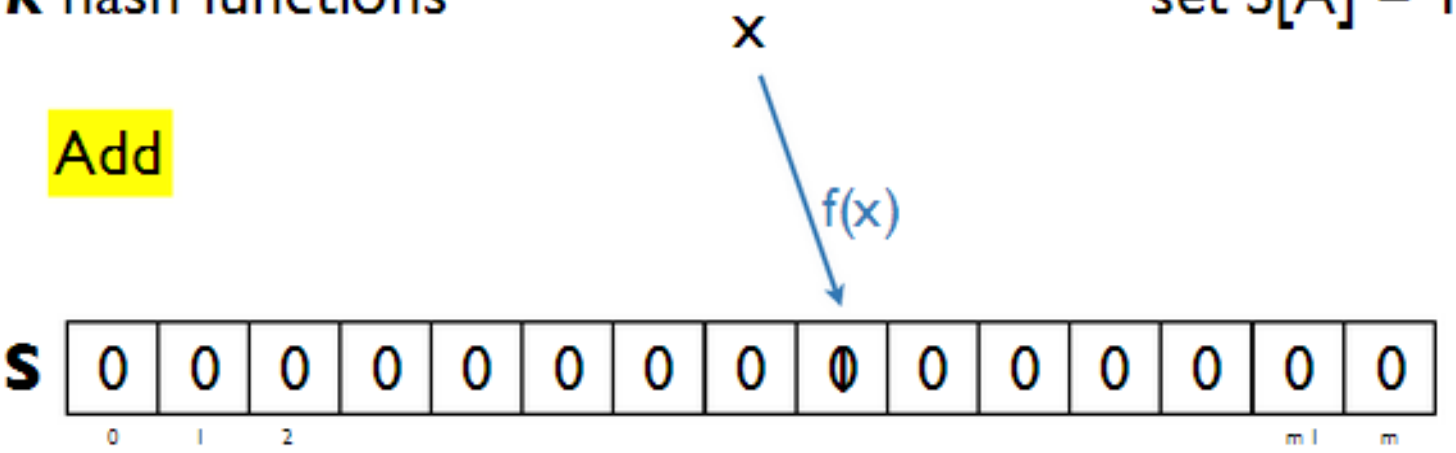
BUILDING BLOOM FILTERS

m bits (initially set to 0)

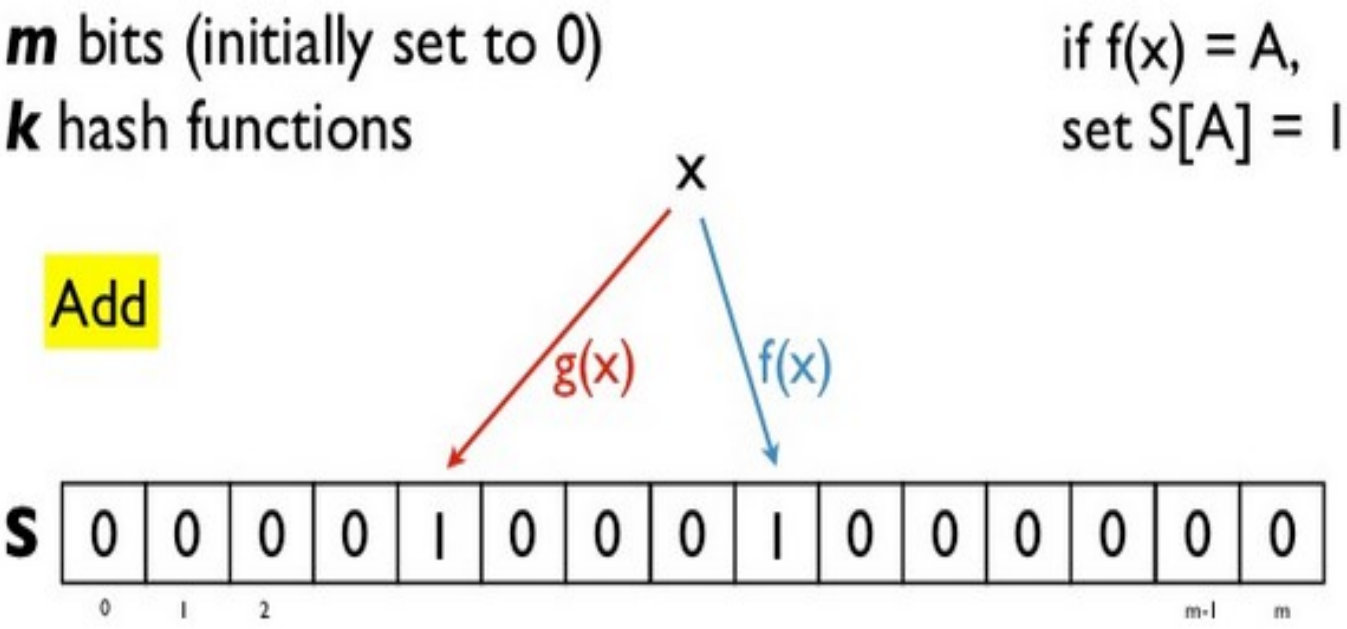
k hash functions

if f(x) = A,
set S[A] = 1

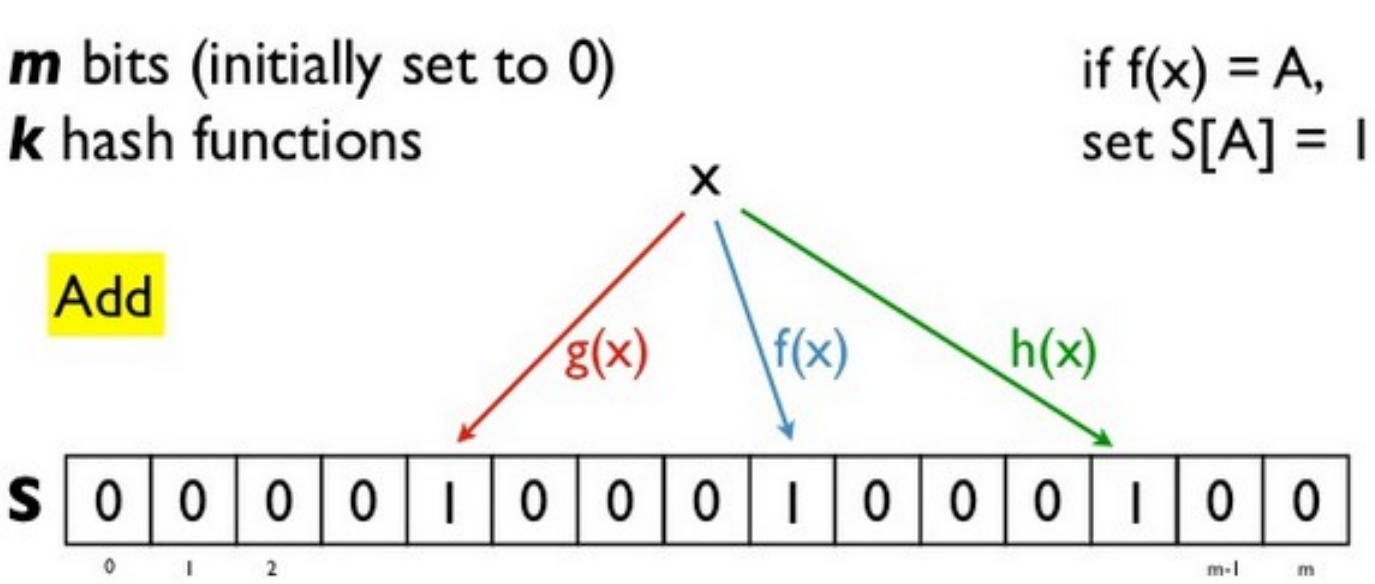
Add



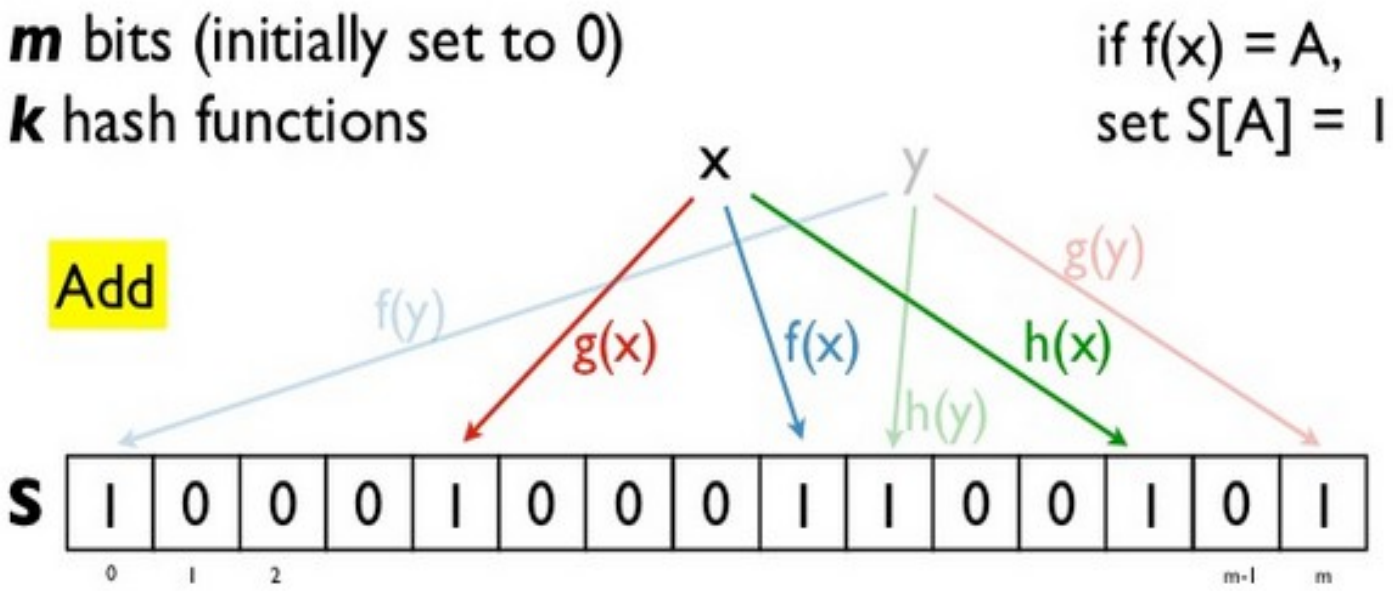
BUILDING BLOOM FILTERS



BUILDING BLOOM FILTERS



BUILDING BLOOM FILTERS

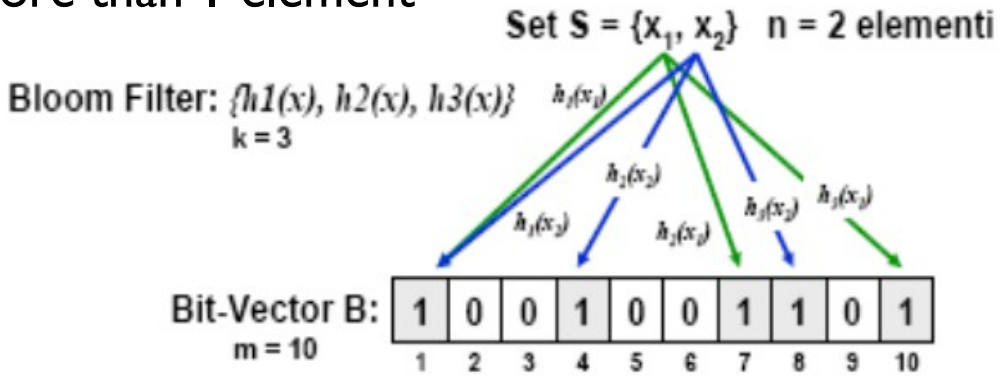


BUILDING BLOOM FILTERS

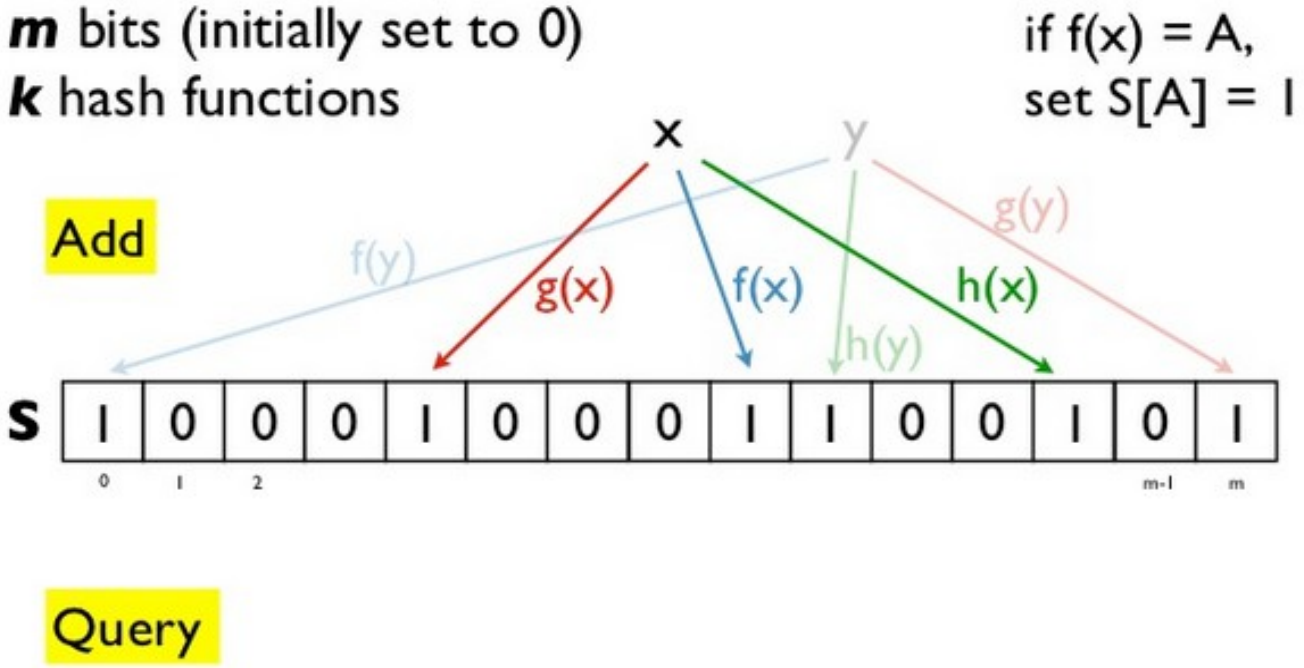
Given

- a set $S=\{s_1,s_2,...,s_n\}$ of n elements
- a vector B of m ($n<<m$, generally $nk<m$) bits, $b_i \in \{0,1\}$
- k hash independent functions $h_1, \ldots, h_k$, for each h_i: $S \subseteq U \rightarrow [1..m]$, which return a value uniformly distributed in the range [1..m].
- Construction procedure for a Bloom Filter B[1..m]:
- for each $x \in S$, $B[h_j(x)]=1$, $\forall$ j = 1,2,...k

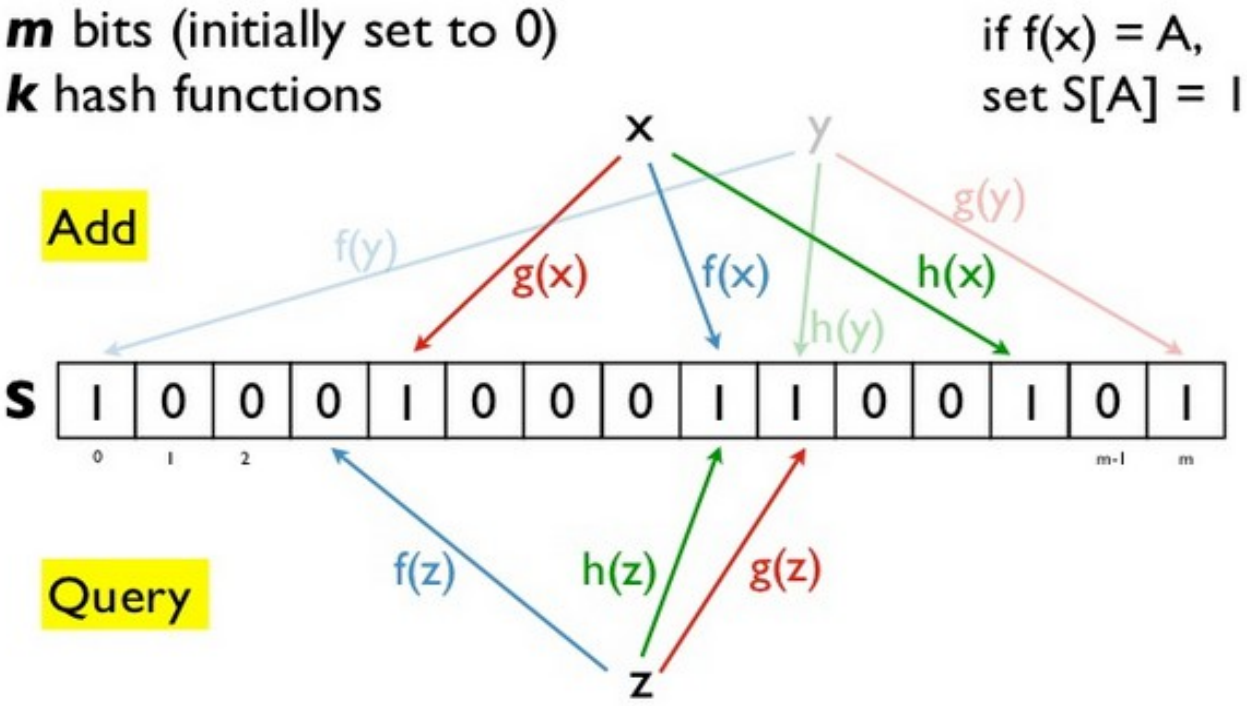
a bit in B may be target for more than 1 element



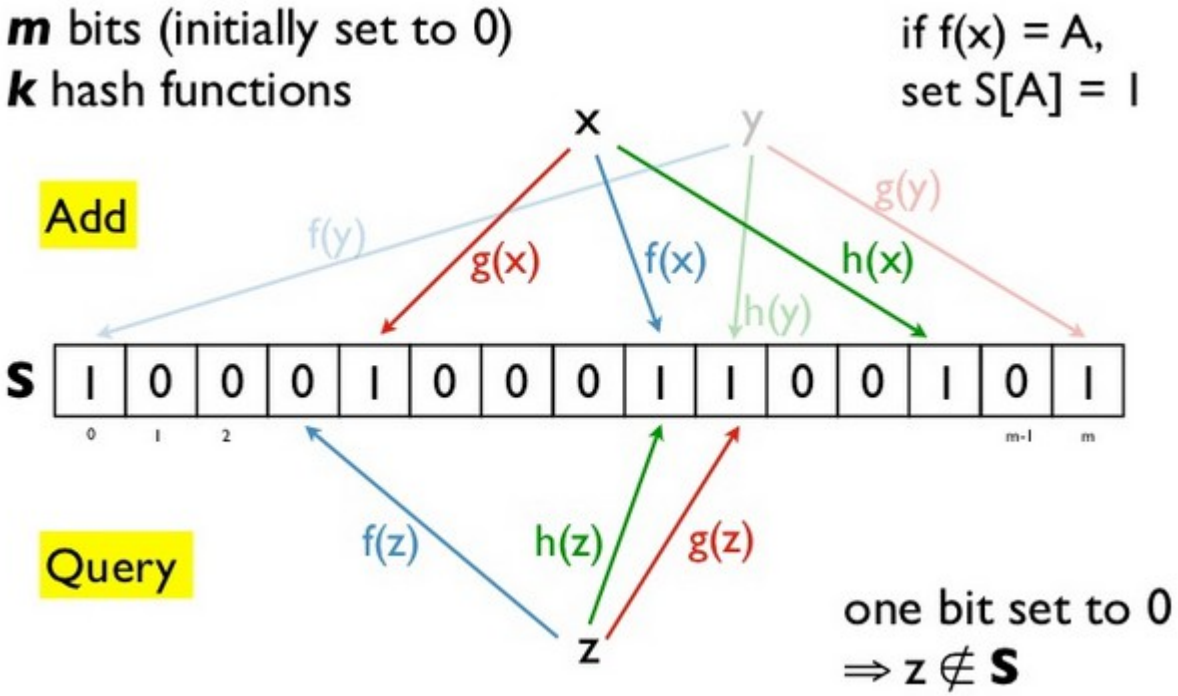
BLOOM FILTERS: LOOK UP



BLOOM FILTERS: LOOK UP



BLOOM FILTERS: LOOK UP



BLOOM FILTERS: LOOK UP

To verify if y belongs to the set S mapped on the Bloom Filter, apply the k hash functions to y

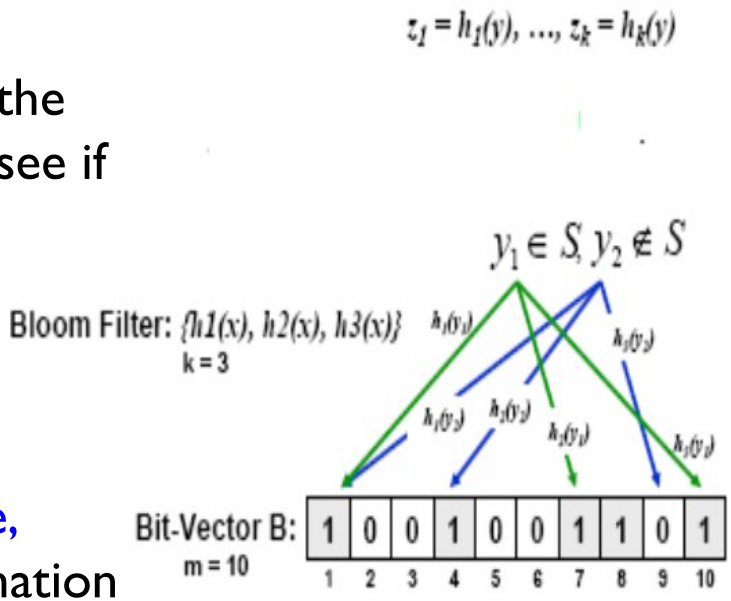
- $y \in S$ if $B[h_i(y)]=1$, $\forall$ $i=1,..k$
- if at least a bit = 0, the element does not belong to the set.

False positives:

To test for membership, you simply hash the string with the same hash functions, then see if those values are set in the bit vector.

If they aren't, you know that the element isn't in the set.

If they are, you only know that it might be, because another element or some combination of other elements could have set the same bits.



BLOOM FILTERS: TRADE OFF

The price paid for this efficiency is that a Bloom filter is a probabilistic data structure: it tells us that the element either definitely is not in the set or may be in the set.

PROBABILITY OF FALSE POSITIVES

- let us consider a set of n elements mapped on a vector of m bits through k hash functions.
- The hash functions used in a Bloom filter should be independent and uniformly distributed and as fast as possible .
  - for instance $h_i(x)$ = MD5(x + i) or MD5(x || i) would work
- basic assumption: hash functions random and independent
  - balls e bins paradigm: like throwing k×n balls in m buckets
- goal: evaluate the probability of false positives

PROBABILITY OF FALSE POSITIVES

First step: compute the probability that, after all the elements are mapped to the vector, a specific bit of the filter has still value 0

$$p' = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}.$$

The approximation is derived from the definition of e

$$\lim_{x \to \infty} \left(1 - \frac{1}{x}\right)^{-x} = e$$

PROBABILITY OF FALSE POSITIVES

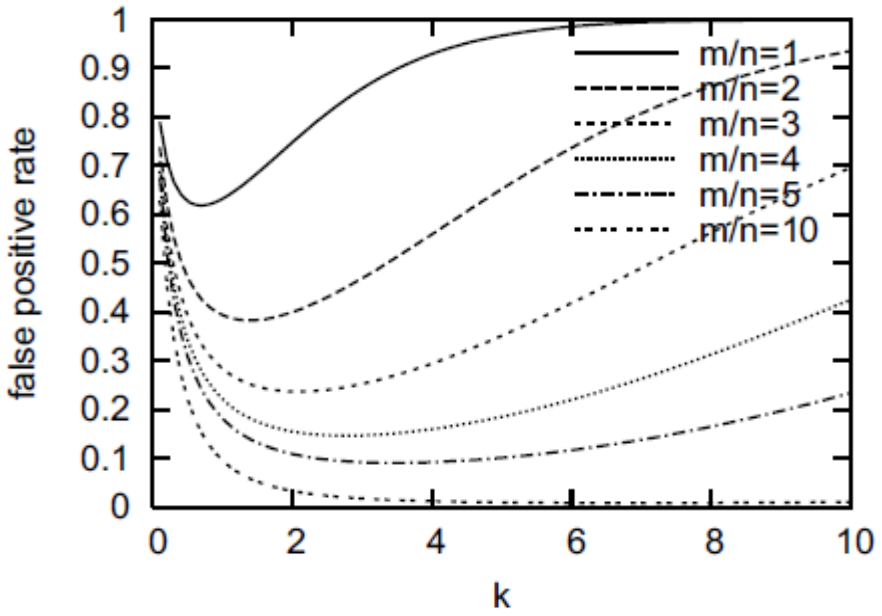
- A percentage of $e^{-kn/m}$ bits are 0, after its construction.
- Consider an element not belonging to the set: apply the k functions, a false positive is obtained if, all hash functions, return a value = 1
- Probability of false positives = $(1 - e^{-kn/m})^k$

  depends on
  - m/n: number of bits exploited for each element of the set
  - k: number of hash functions

- If m/n is fixed, it seems two conflicting factors for defining k do exist....
  - decreasing k increases the number of 0 and hence the probability to have a false positive should decrease, but....
  - increasing k increases the number of elements to be checked and the precision of the method. Hence the probability of false positive should decrease...

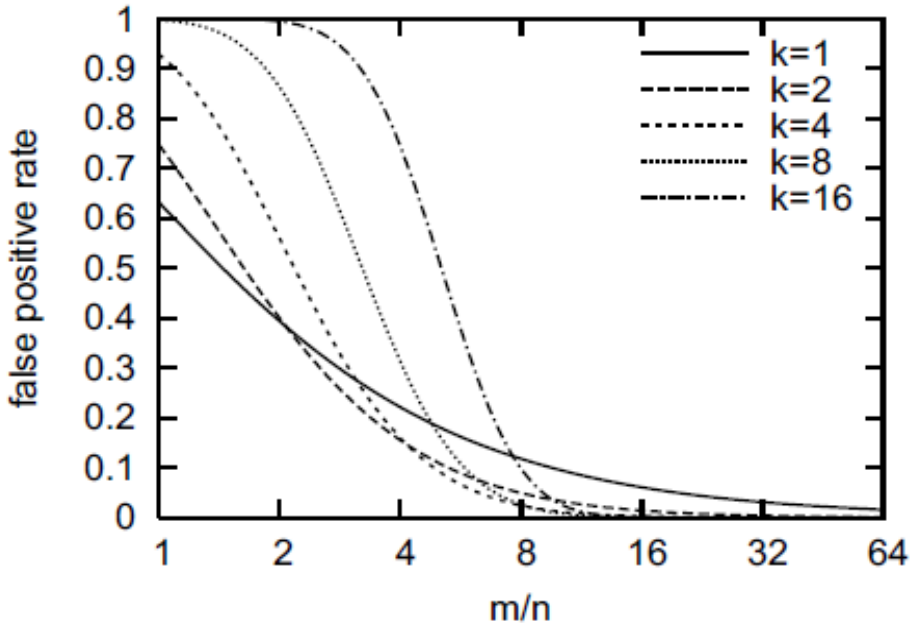
PROBABILITY OF FALSE POSITIVES

- Fixed the ratio m/n, the probability of false negatives first decreases, then increases, when considering increasing values of k
  - es: m/n=2, a few bits for each element, "too much hash functions" cannot be exploited because they fill the filter of 1.
  - es: m/n=10, a larger number of hash functions decreases the probability of false positives



PROBABILITY OF FALSE POSITIVES

- let us now suppose that k is fixed, the probability of false positives exponentially decreases when m increases (m number of bits in the filter).
- for low values of m/n (a few bits for each element), the probablity is higher for large values of k



PROBABILITY OF FALSE POSITIVES

bits/element	m/n	2	8	16	24
number of hash functions	k	1.39	5.55	11.1	16.6
false-positive probability	f	0.393	0.0216	$4.59 \cdot 10^{-4}$	$9.84 \cdot 10^{-6}$

A Bloom filter becomes effective when m= c × n, with c constant value (low value), for instance c = 8

In this case with 5-6 hash functions the probability of false positives is low

Good performances with a limited number of bits

PROBABILITY OF FALSE POSITIVES

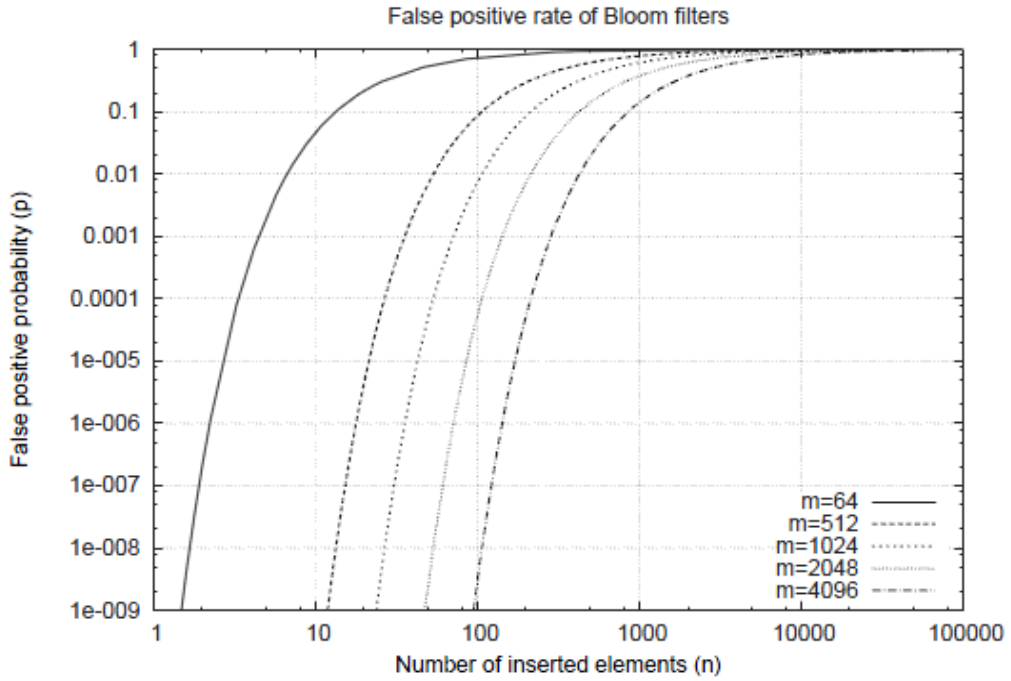
- trade-off between space/number of hash functions/probability of false positives
- if n and m are fixed (fix the number of bits for each element)
  - determine which is the value of k minimizing the probability of false positives
  - compute the derivative of the function of the previous slide so obtaining the minimum (ln 2 ≈ 0.7)

$$k = \frac{m}{n} \ln 2,$$

To this value, corresponds a value of the probability equal $0.62^{m/n}$

PROBABILITY OF FALSE POSITIVES

- probability as a function of the number of elements of the set, with k optimum and m variable as a function of n
- logarithmic scale
- if the number of bits for each element is not sufficient the probability exponentially grows



PROBABILITY OF FALSE POSITIVES

with the values computed in the previous slides, the probability that one bit is still equal to 0 after the application of the k functions is

$$p = e^{-kn/m} = \frac{1}{2}$$

the optimal values are obtained when the probability that a bit is equal to 0 after the application of the k functions to the n elements is equal to ½

an "optimal" Bloom Filter is a "random bitstring" where half of the bits chosen uniformly at random, is 0.

BLOOM FILTERS: OPERATIONS

- Union - Given two Bloom Filters, B1 e B2 representing, respectively, the set S1 and S2 through the same number of bits and the same number of hash functions, the Bloom filter representing S1 $\cup$ S2 is obtained by computing the bitwise OR bit of B1 and B2

- Delete: note that it is not possible to set to 0 all the elements indexed by the output of the hash functions, because of the conflicts

  - Counting Bloom Filters: each entry of the Bloom Filter is a counter, instead of a single bit

    - exploited to implement the removal of elements from the Bloom filter
    - at insertion time, increment the counter
    - at deletion time, decrement the counter

BLOOM FILTERS: OPERAZIONS

- Intersection: Given two Bloom Filters B1 and B2 representing respectively, the sets S1 and S2 through the same number of bits and the same number of hash functions.
  - Bloom Filter obtained by computing the bitwise and of B1 and B2 approximates $S1 \cap S2$
  - as a matter of fact, if a bit is set to 1 in both Bloom filters, this may happen because:
    - this bit corresponds to an element $\in S1 \cap S2$, therefore it is set to 1 in both filters: in this case no approximation
    - this bit corresponds to an element $\in S1 - (S1 \cap S2)$ and to an element $\in S2 - (S1 \cap S2)$ hence it does not correspond to any element in the intersection

BLOOM FILTERS APPLICATIONS

Some applications....

- Google BigTable and Apache Cassandra use Bloom filters to avoid costly disk lookups considerably and to increases the performance of a database query.
- The Google Chrome web browser uses a Bloom filter to identify malicious URLs. Any URL is first checked against a local Bloom filter and only upon a hit a full check of the URL is performed.
- Bitcoin uses Bloom filters to verify payments without running a full network node.
- Gnutella 0.6 exploits a simplified version of Bloom Filters

but many other ones currently exist....

BLOOM FILTERS: APPLICATIONS

Guard against expensive operations (like disk access)

First line of defence in high performance (distributed) caches

Peer to Peer communication

Routing - Resource Location

Squid Proxy Cache

Google BigTable

Cassandra

HBase

Various RDBMS'

Google Chrome

Cisco Routers

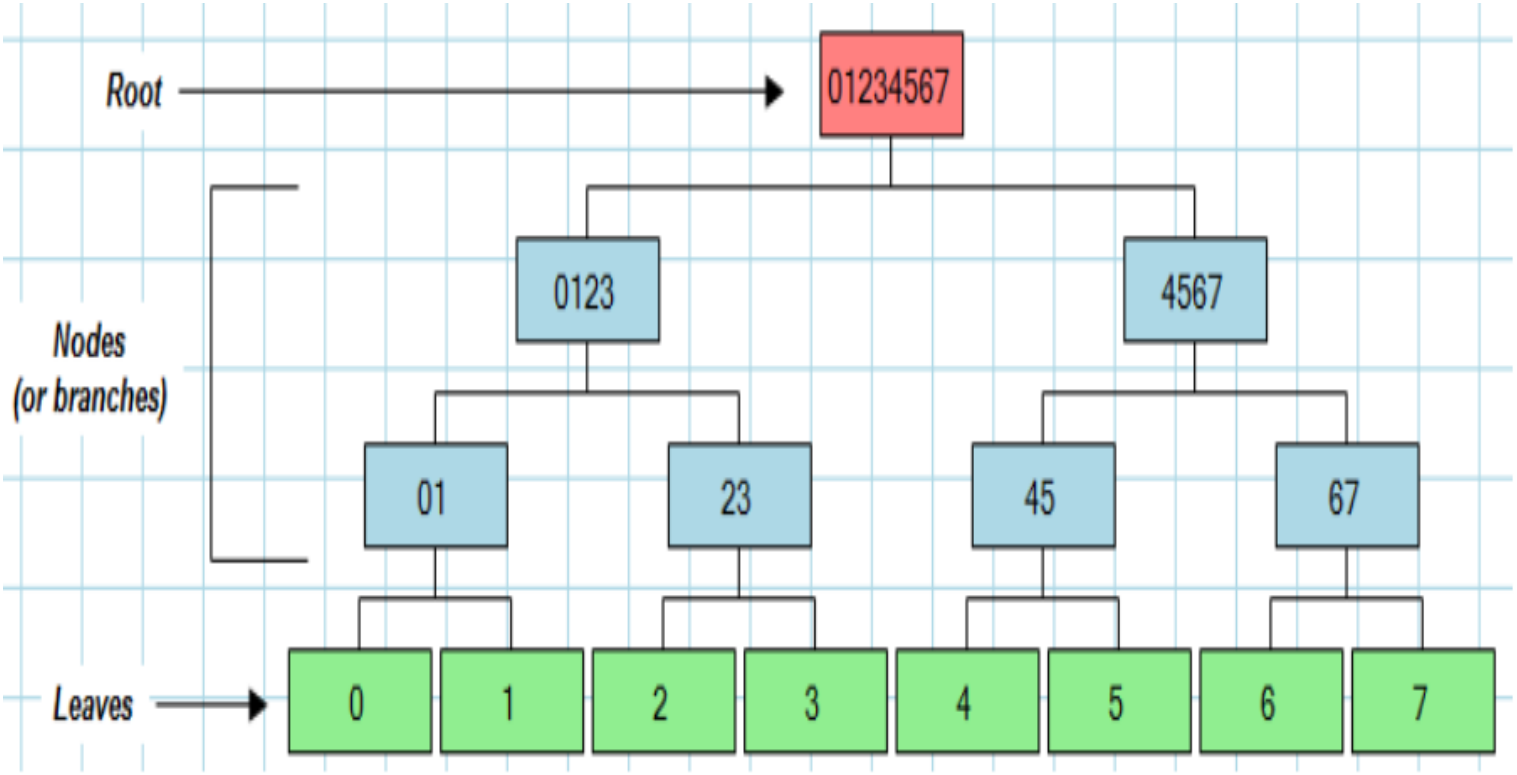
...

http://www.docjar.com/html/api/org/apache/cassandra/utils/BloomFilter.java.html

MERKLE TREE

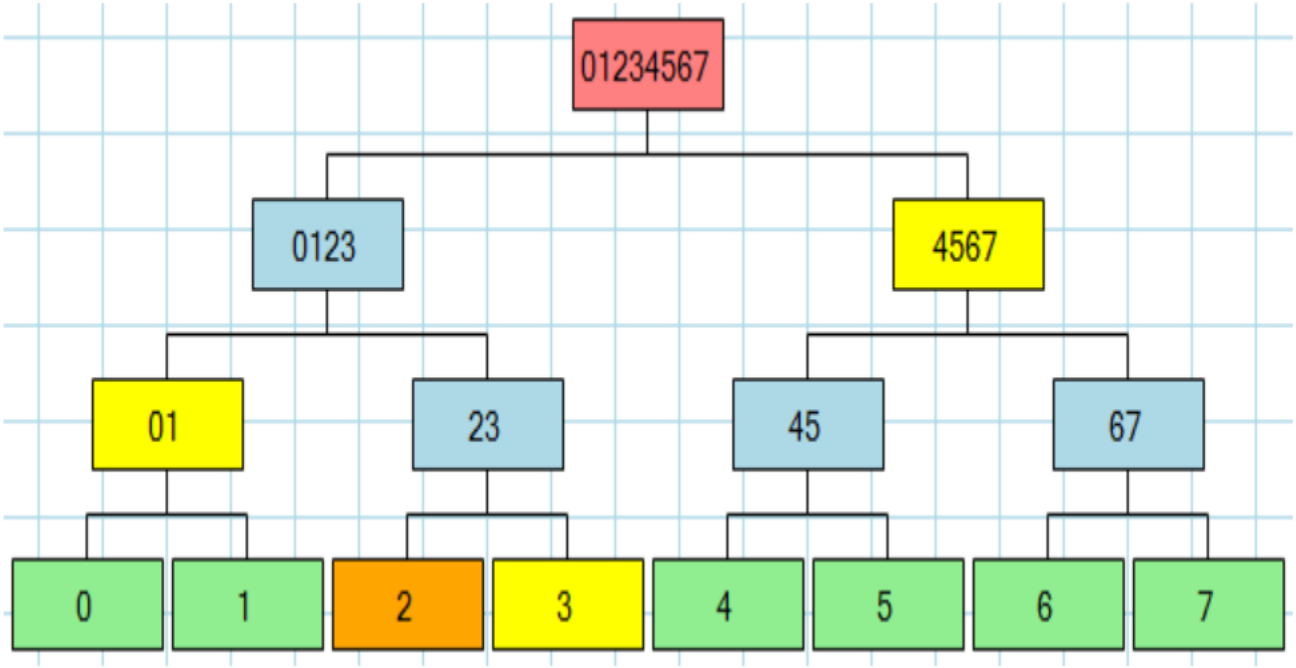
- Hash trees or Merkle trees
  - a data structure summarizing information about a big quantity of data with the goal to verifying the correctness of the content
- introduced by Ralph Merkle in 1979
- characteristics
  - simple
  - efficient
  - versatile
- a complete binary tree of hashes built starting form a initial set of blocks
  - exploits a hash function H
  - leaves: H is applied to the initial blocks
  - internal nodes: H is applied to the concatenation of the hashes of the sons of a node

MERKLE TREE



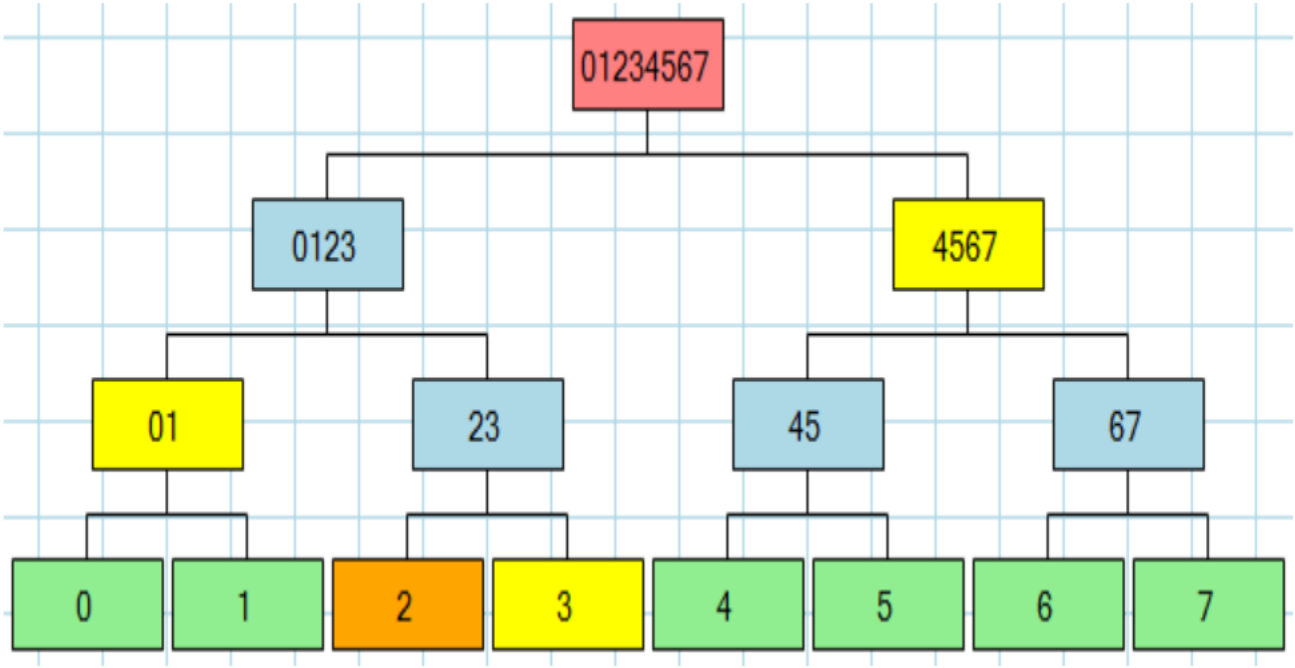
- to simplify the presentation, internal nodes are simply shown as concatenation of the hashes of the sons
- actually, hash is recursively applied to the concatenation

MERKLE PROOF



- Suppose that we want to check that block 2 is not tampered

MERKLE PROOF



- a Merkle proof consists of
  - a chunk: 2
  - the root hash of the tree: the red one
  - the "branch" consisting of all of the hashes going up along the path from the chunk to the root.
- this is sufficient to verify that 2 is really in the tree and in that position

MERKLE TREE FORMAL DEFINITION

- Let us consider an initial set of symbols S, $S=\{s_1, ..., s_n\}$, $n=2^h$, h tree height
- Merkle Tree Procedure MTP = <CMT,DMT>
- CMT (Coding Merkle Tree): starting from the initial set S of symbols, build the complete binary tree of height h
  - Leaves = $H(s_i)$
  - Internal Nodes = hash of the concatenation of the hash values of the sons H(L||R)
- Output
  - the root of the binary tree
  - a "witness" w_i for each symbol s_i
    - w_i = siblings of the nodes on the path from the leaf s_i to the root

MERKLE TREE

- DMT(s_i,w_i,root): Decoding Merkle Tree
- The soundness of s_i is deduced from the comparison between the root generated during the decoding phase and the root generated in the coding phase
- Each symbol s_i is authenticated by considering the witness w_i and the root

MERKLE TREE

- Build Merkle tree
- 3rd party trusted stores root
- log(n) hashes are sufficient for checking each fragment
- "Data integrity over untrusted storage with small communication cost"
- Pros:
  - scales logarithmically in the number of objects
  - fine-grained data integrity
  - each write does work proportional to the size of the fragment
- Cons:
  - static: smallest segment size = smallest unit of verification

MERKLE TREES: BITTORRENT

- a trusted site (for instance that indexing the .torrent) stores:
  - the root hash
  - the total size of the file
  - the piece size
- a peer:
  - receives a piece from another peer togeter with the hashes of the piece's sibling and of its uncles, that is the sibling Y of its parent X, and the uncle of that Y until the root is reached.
  - calculates the hash of that piece.
  - request the root hash from a trusted site and
  - using this information the client recalculates the root hash of the tree, and compares it to the root hash it received from the trusted source.
- subject to possible pre-image attacks.

MERKLE TREES: APPLICATIONS

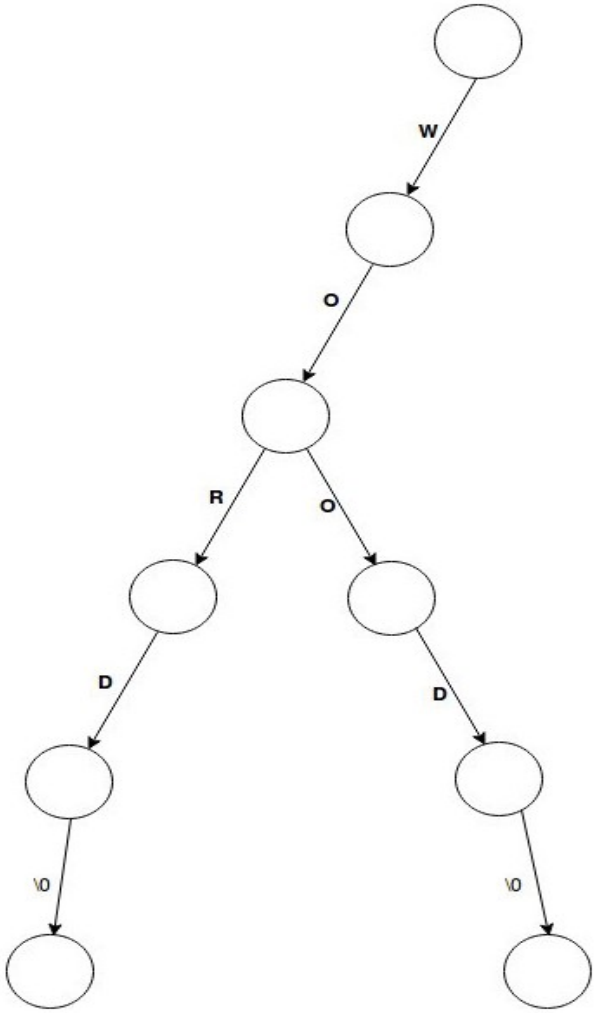
- Merkle trees and peer to peer applications:
  - can help ensuring in a P2P network that data blocks received from other peers are received undamaged and unaltered
  - to check that the other peers do not lie and send fake blocks.
- BitTorrent: uses Merkle trees to ensure that the files you download from peers haven't been tampered.
- Similar for eMule
- Distributed Version Control (Git/Mercurial)
- Copy-On-Write Filesystems (btrfs/ZFS)
- Distributed NoSQL Databases (Cassandra, Riak, Dynamo)
- The distributed web! (IPFS)
- Cryptos (Bitcoin, Ethereum)

MERKLE TREES: APPLICATION

TRIE

- a data structure used for storing sequences of characters.

- adding a string S to a trie: follow the search path guided by S.
  - when adding a (shorter) string contained in another (longer) string, exhaust all of the characters and then add a null pointer (terminator)
  - otherwise:
    - if a null pointer is encountered or a non matching character is found, create new nodes until the string is finished, then create a null pointer (terminator).

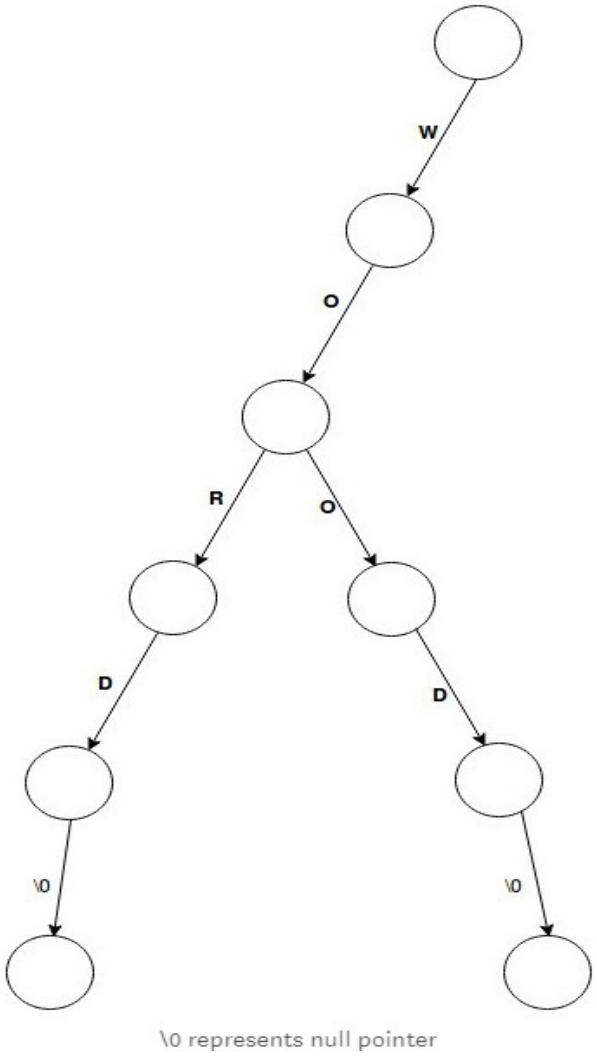


\0 represents null pointer

TRIE

- a data structure used for storing sequences of characters.

- deleting a string from the trie
  - search for a leaf (the very end of a branch) on the trie that represents the string to delete
  - start deleting all nodes from the leaf back to the root of the trie, unless we hit a node with more than one child; in this case stop.

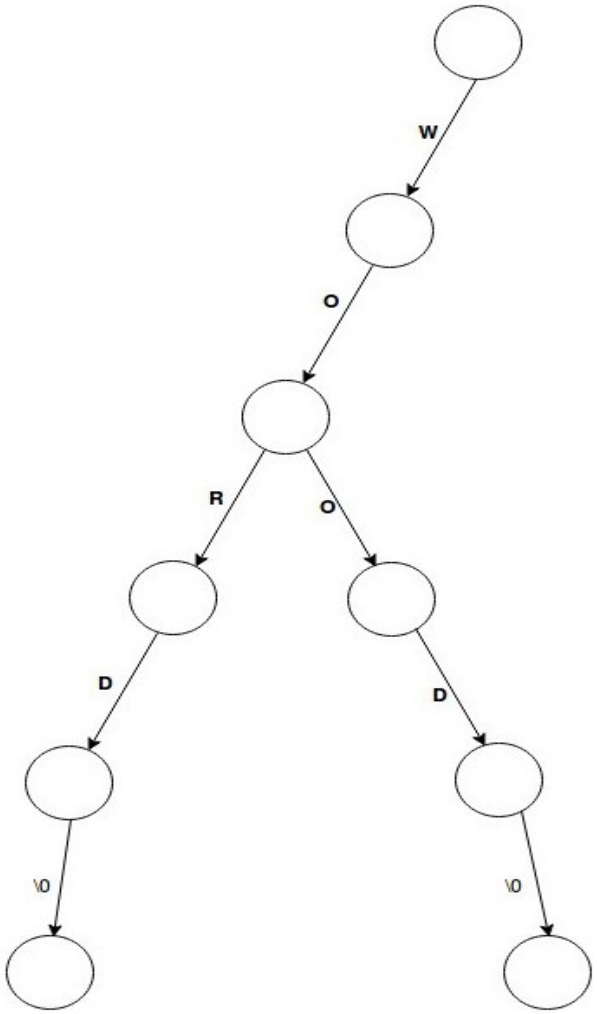


\0 represents null pointer

TRIE

- a data structure used for storing sequences of characters.

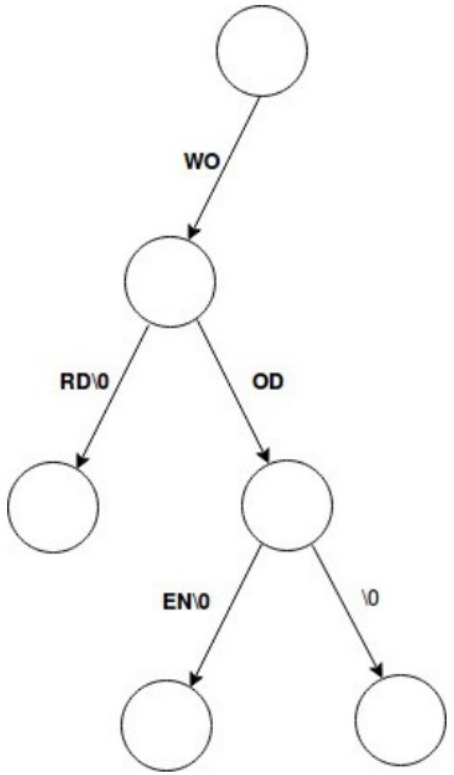
- search a string in the trie
  - examine each of the characters in the string and follow the trie for as long as it provides the path
  - If a null pointer is encountered before exhausting all of the characters in the string the string is not stored in the trie.
  - if a leaf (the end of a branch)is reached the path (from the leaf back to the root of the trie) represents the string.



\0 represents null pointer

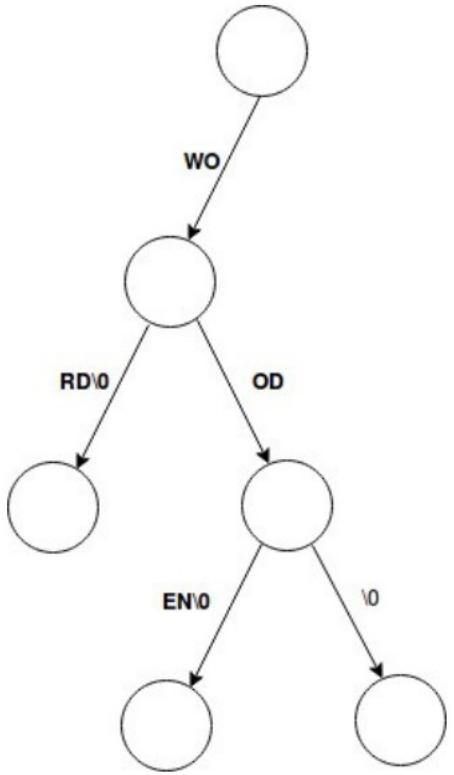
PATRICIA TRIE

- Practical algorithm to retrieve information coded in alphanumeric (Patricia)
- a data structure used for storing sequences of characters.
- group all common characters into single edge (branch)
- when encountering any non common characters create a new branch in the path.



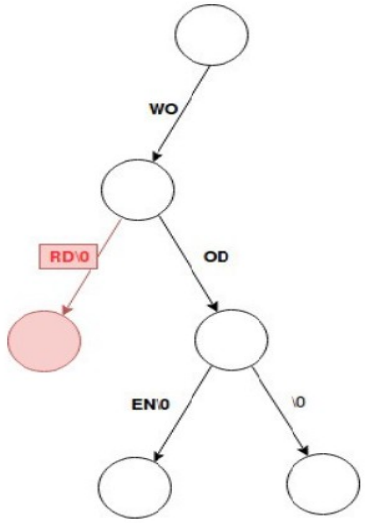
PATRICIA TRIE

- Practical algorithm to retrieve information coded in alphanumeric (Patricia)
- a data structure used for storing sequences of characters.
- a single child node may just have characters and a null pointer: at the very end of every string
- a single child node may just have a null pointer if a shorter word is contained in a longer word. ("wood" and "wooden" co-exist in the same trie.)
- String insertion: exhaust all of the characters by foollowing existing path, and then add the null pointer (terminator).

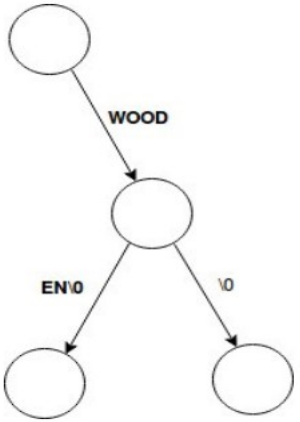


PATRICIA TRIE

- Practical algorithm to retrieve information coded in alphanumeric (Patricia)
- a data structure used for storing sequences of characters.
- deleting from a trie
  - a path can not be left with a parent nodes which just connects to single child node. If this occurs to concatenate the appropriate characters.
  - In the figure: delete the word "word" from the trie.



Before deleting the word "word" from the trie



After—reorganization of trie as part of delete