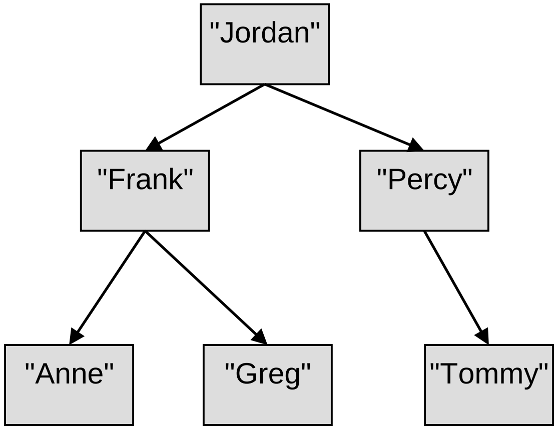
CSCI 104
Bloom Filters & Tries

Mark Redekopp
David Kempe
Sandra Batista

Set Review

- Recall the operations a set performs…
  - Insert(key)
  - Remove(key)
  - Contains(key) : bool   (a.k.a. find())
- We can implement a set using
  - List
    - O(n) for some of the three operations
  - (Balanced) Binary Search Tree
    - O(log n) insert/remove/contains
  - Hash table
    - O(1) insert/remove/contains



Bloom Filter

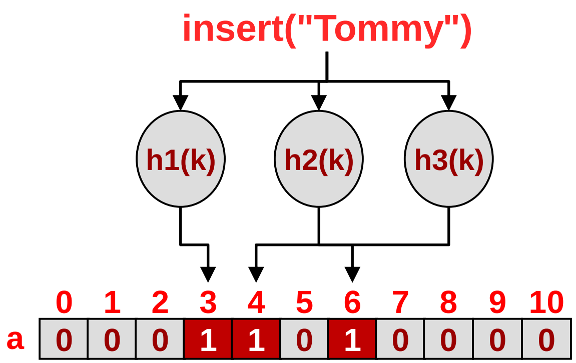
- A Bloom filter is a set such that "contains()" will *quickly* answer…
  - "No" correctly (i.e. if the key is not present)
  - "Yes" with a chance of being incorrect (i.e. the key may not be present but it might still say "yes")

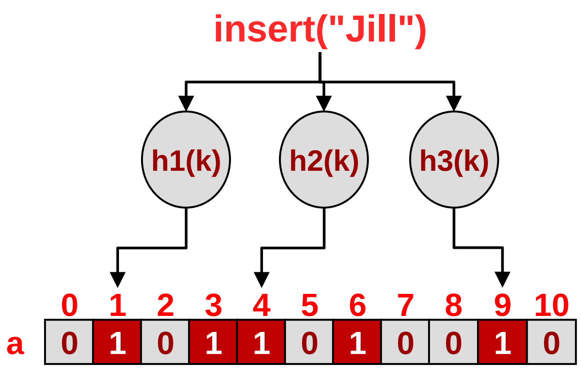
Bloom Filter Motivation

- Why would we want this?
  - A Bloom filter usually sits in front of an actual set/map
  - Suppose that set/map is EXPENSIVE to access
    - if set/map doesn't sits on a disk drive or another server
      - Disk/Network access = ~milliseconds
      - Memory access = ~nanoseconds
  - The Bloom filter is small enough to reside in memory for quick access and can answer quickly if the set/map on disk contains a key:
    - If it answers "No" do not search the EXPENSIVE set
    - If it answers "Yes" search the EXPENSIVE set

Bloom Filters

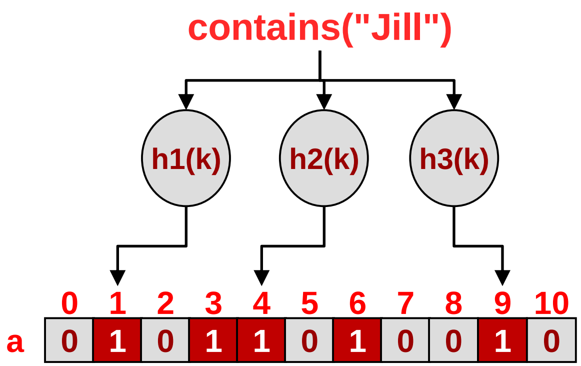
- A Bloom filter is…
  - A hash table of individual bits (Booleans: T/F)
  - A set of hash functions, $\{h_1(k), h_2(k), \ldots h_s(k)\}$
- Insert()
  - Apply each $h_i(k)$ to the key
  - Set $a[h_i(k)]$ = True

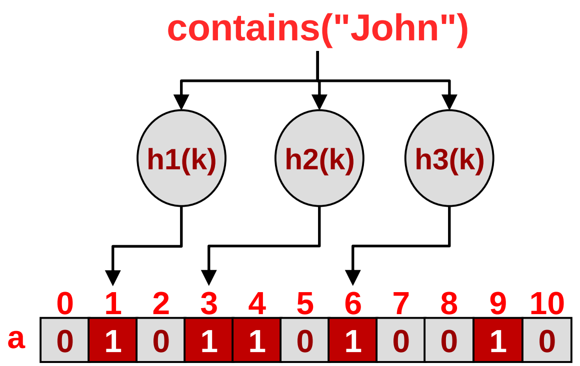




Bloom Filters

- A Bloom filter is…
  - A hash table of individual bits (Booleans: T/F)
  - A set of hash functions, $\{h_1(k), h_2(k), \ldots h_s(k)\}$
- Contains()
  - Apply each $h_i(k)$ to the key
  - Return True if ***all*** $a[h_i(k)]$ = True
  - Return False otherwise
  - In other words, answer is "Maybe" or "No"
    - May produce "false positives"
    - May NOT produce "false negatives"
- We will ignore removal for now





Practice

- Trace a Bloom Filter on the following operations:
  - insert(0), insert(1), insert(2), insert(8), contains(2), contains(3), contains(4), contains(9)
  - The hash functions are
    - h1(k)=(7k+4)%10
    - h2(k) = (2k+1)%10
    - h3(k) = (5k+3)%10
    - The table size is 10 (m=10).

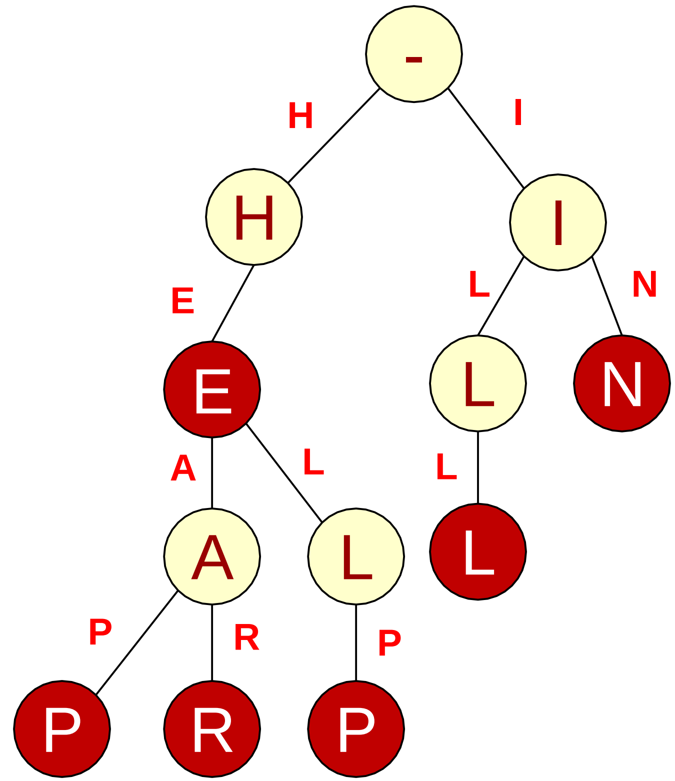
	0	1	2	3	4	5	6	7	8	9
a	0	0	0	0	0	0	0	0	0	0

	H1(k)	H2(k)	H3(k)	Hit?
Insert(0)	4	1	3	N/A
Insert(1)	1	3	8	N/A
Insert(2)	8	5	3	N/A
Insert(8)	0	7	3	N/A
Contains(2)	8	5	3	Yes
Contains(3)	5	7	8	Yes
Contains(4)	2	9	3	No
Contains(9)	7	9	8	No

USC Viterbi School of Engineering

Tries

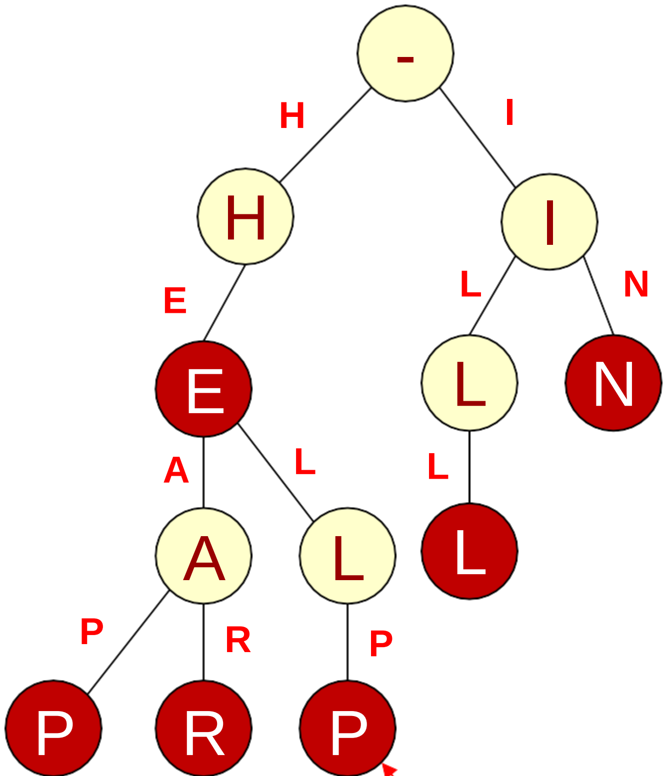
- Goal: Achieve linear search in length of keys independent of number of keys
- Rather than each node storing a full key value, each node represents a prefix of the key
- Highlighted nodes indicate terminal locations
  - For a map we could store the associated value of the key at that terminal location
- Notice we "share" paths for keys that have a common prefix



Trie for the keys: "HE", "HEAP", "HEAR", "HELP", "ILL", "IN"

Tries

- To search for a key, start at the root consuming one unit (bit, char, etc.) of the key at a time
  - If highlighted node, SUCCESS
  - else FAILURE
- Examples:
  - Search for "He"
  - Search for "Help"
  - Search for "Head"
- Search takes O(m) where m = length of key



USC Viterbi School of Engineering

Application: IP Lookups

- Network routers form the backbone of the Internet
- Incoming packets contain a destination IP address (128.125.73.60)
- Routers contain a "routing table" mapping some prefix of destination IP address to output port
  - 128.125.x.x => Output port C
  - 128.209.32.x => Output port B
  - 128.209.44.x => Output port D
  - 132.x.x.x => Output port A
- Keys = Match the longest prefix
  - Keys are unique
- Value = Output port

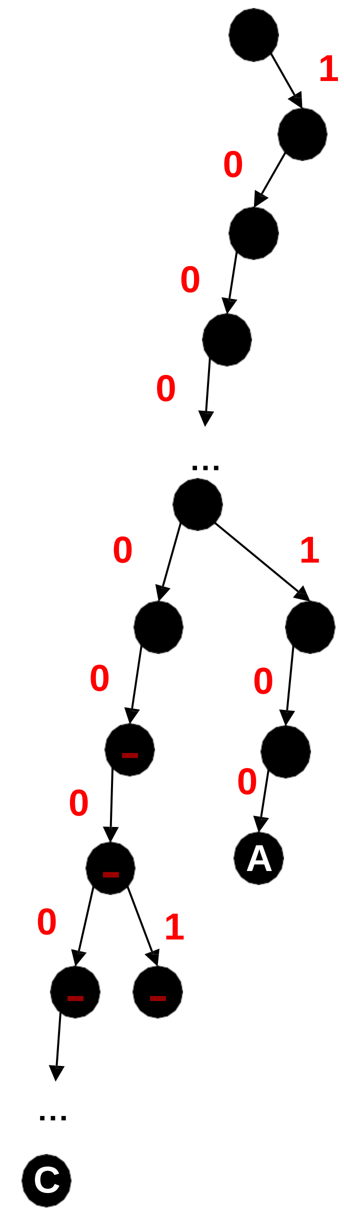
Octet 1	Octet 2	Octet 3	Port
10000000	01111101		C
10000000	11010001	00100000	B
10000000	11010001	00101100	D
10000100			A

USC Viterbi School of Engineering

IP Lookup Trie

- A binary trie implies that the
  - Left child is for bit '0'
  - Right child is for bit '1'
- Routing Table:
  - 128.125.x.x => Output port C
  - 128.209.32.x => Output port B
  - 128.209.44.x => Output port D
  - 132.x.x.x => Output port A

Octet 1	Octet 2	Octet 3	Port
10000000	01111101		C
10000000	11010001	00100000	B
10000000	11010001	00101100	D
10000100			A



Trie Creation Algorithm

- Given a set of strings, S

Let Alpha be set of union of all letters in strings in S.

– The root corresponds to empty string at level 0.

For (each node at level i) /*prefix length at node is i*/

    For (each letter, w, in Alpha)

        For (each string, s, in S)

            If (x*w is a prefix of s)

                Add new node and edge labeled w.

                New node prefix is x*w of length i+1.

              if (x*w equals s) then highlight new node and add value.

      If all s in S are highlighted, stop.

Trie Creation

- Let's construct a trie to store the set of words:
  - Heap
  - Helm
  - Hear
  - Help
  - He
  - In
  - Ink
  - Irk
  - She

Trie Complexity

- What is the runtime of insert?

- What is runtime of search?

- What is size of trie?

Compressed Trie

- In our construction many nodes were redundant: only a single child along path to node
- Compressed trie is when paths of single node are combined into a node
- In compressed trie all nodes have at least one of the following properties:

- Root node
- Word node
- Has at least 2 children

Compressed Trie Practice

- Let's construct a compressed trie to store the following set of words:
  - Ten
  - Tent
  - Then
  - Tense
  - Tens
  - Tenth