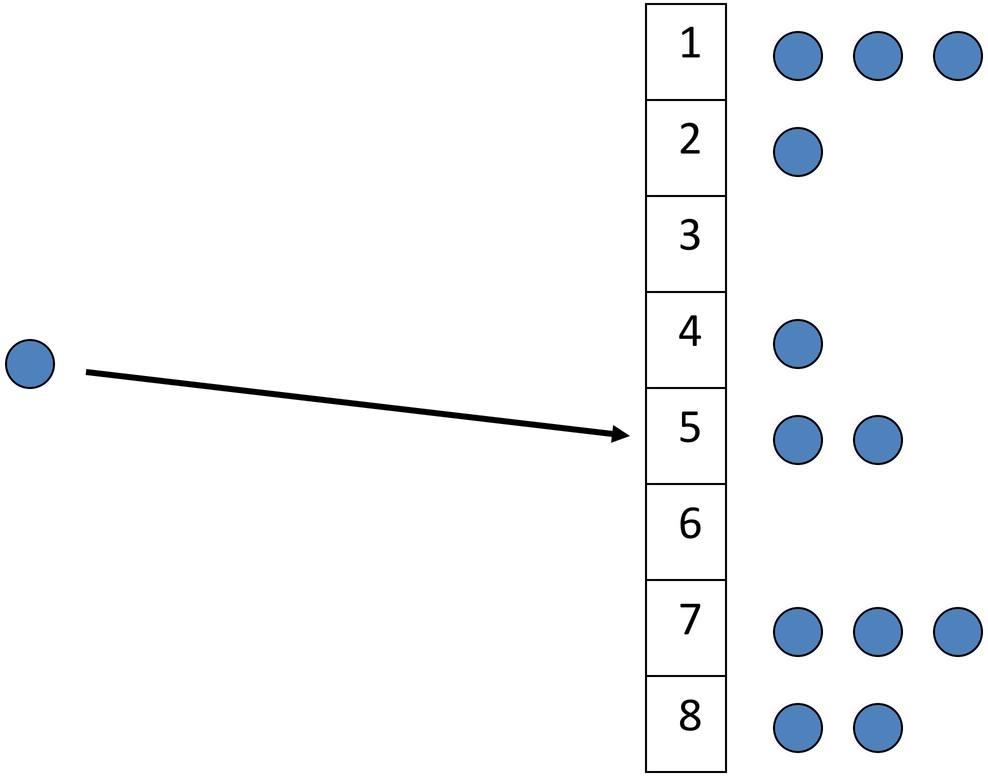
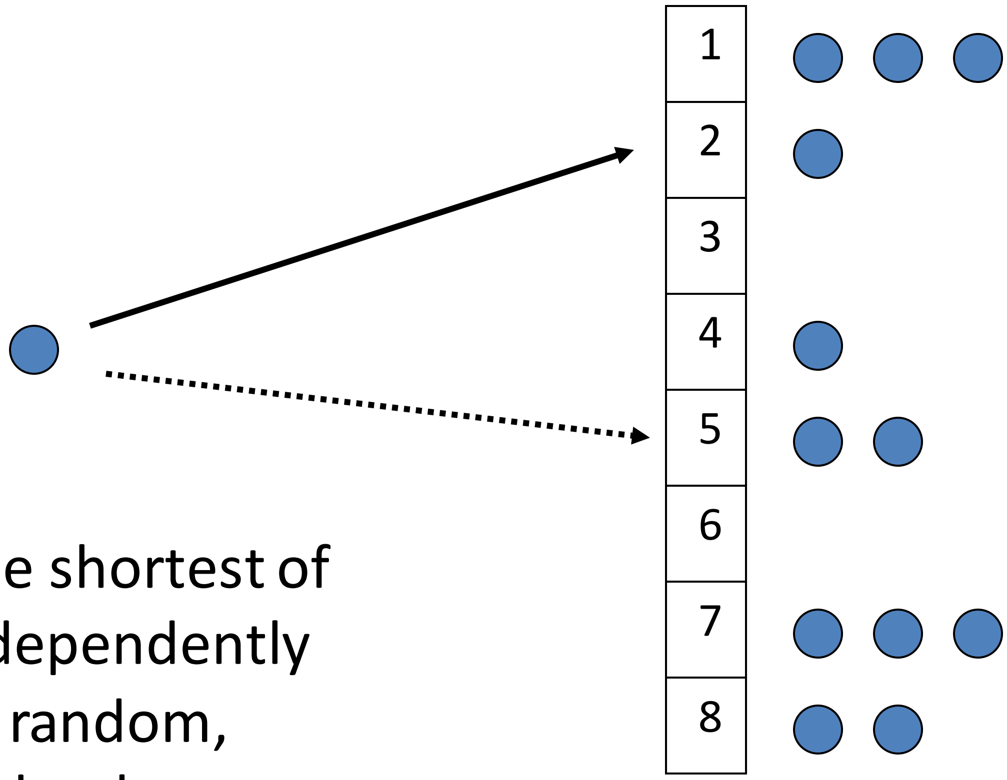
Balanced Allocations, Cuckoo Hash Tables, and Such

Michael Mitzenmacher
Harvard University

Hashing Model



Hashing Model: *d*-Random



Item placed in the shortest of *d* bins chosen independently and uniformly at random, breaking ties randomly.

Azar, Broder, Karlin, Upfal (STOC 94)

Throw n balls into n bins randomly.
Maximum load is $\log n / \log \log n$.

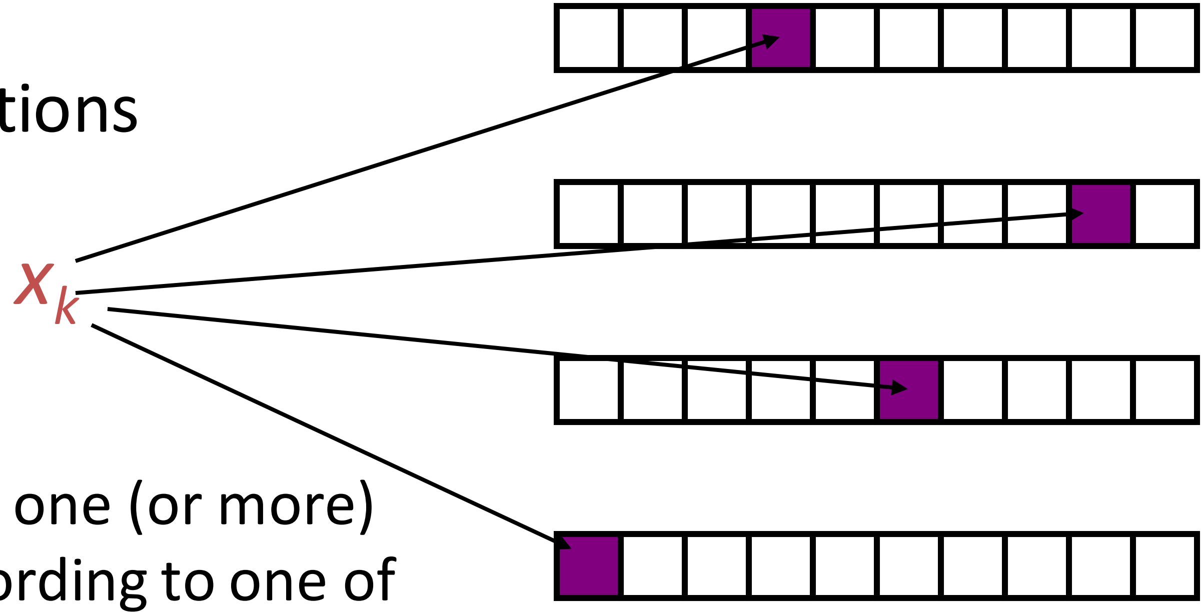
Place n balls into n bins sequentially, each ball going into the least loaded of d random locations.

Maximum load is $\log \log n / \log d$.
Two choices: $\log \log n / \log 2$.

Multiple-Choice Hashing

$S = \{x_1, x_2, \ldots, x_n\}$ of items

d hash functions



Item placed in one (or more) locations, according to one of the hash functions.

d subtables

Multiple-Choice Hashing

- Can significantly reduce load.
  - Choice allows much more even spread of items over buckets.
- But need to check *d* locations.
  - Parallelizable, but still a cost -- e.g. pin count in router design.

Multiple-Choice Hashing

- Can significantly reduce load.
  - Choice allows much more even spread of items over buckets.
- But need to check *d* locations.
  - Parallelizable, but still a cost -- e.g. pin count in router design.
  - Can sometimes use Bloom filters (or variants) to track which location(s) to check for an item.

Example: Cached Hash Tables

- May want hash table to live in cache: each bucket corresponds to a hash line.
  - Example: hash tables in routers.
  - Consider 4-6 items per hash line.
- With one choice, large and highly variable maximum load.
  - Lots of empty, lightly loaded bins.
- Using two choices, most bins mostly full: more efficient memory usage.
- Application: IP routing.

Analysis Methods

- Layered induction
- Witness trees
- Fluid limits

How Many Empty Bins?

- [Hajek 88]

Let $x(t)$ be fraction of non-empty bins;
throw all n balls in 1 second.

$$E[\Delta x(t)] = (1 - x(t)^d)/n\,;\, \Delta t = 1/n$$

$$dx/dt = 1 - x^d$$

Solve at $t = 1$, given $x(0) = 0$. Obtain $x(1) = 0.7616..$ for $d = 2$, matches simulations.

Generalize to Loads

Let $s_k(t)$ be fraction of bins with load at least k.

$$E[\Delta s_k(t)] = (s_{k-1}(t)^d - s_k(t)^d)/n\,;\ \Delta t = 1/n$$

All choices must have load at least k-1,
but not all can have load at least k.

$$ds_k/dt = s_{k-1}^d - s_k^d$$

Successively solve equations at $t = 1$.

Double Exponential Decrease

$$ds_k / dt = s_{k-1}^d - s_k^d$$

$$ds_k / dt \leq s_{k-1}^d(t) \leq s_{k-1}^d(1)$$

$$s_k(1) \leq s_{k-1}^d(1) \leq s_{k-2}^{d^2}(1) \ldots \leq s_1^{d^{k-1}}(1)$$

This is crux of the ABKU proof; implies maximum load of log log n / log d.

Kurtz's Theorem

- Over fixed time intervals and for fixed finite dimensional processes, the deviation of the random process from the solution to the differential equations obeys Chernoff-like bounds.

$$\Pr[\sup_{0\le t\le T,1\le i\le k} |s_i(t)-\hat{s}_i(t)| > \varepsilon] \le c_1 k \exp(-c_2 n \varepsilon^2)$$

Power of Two Generalizations

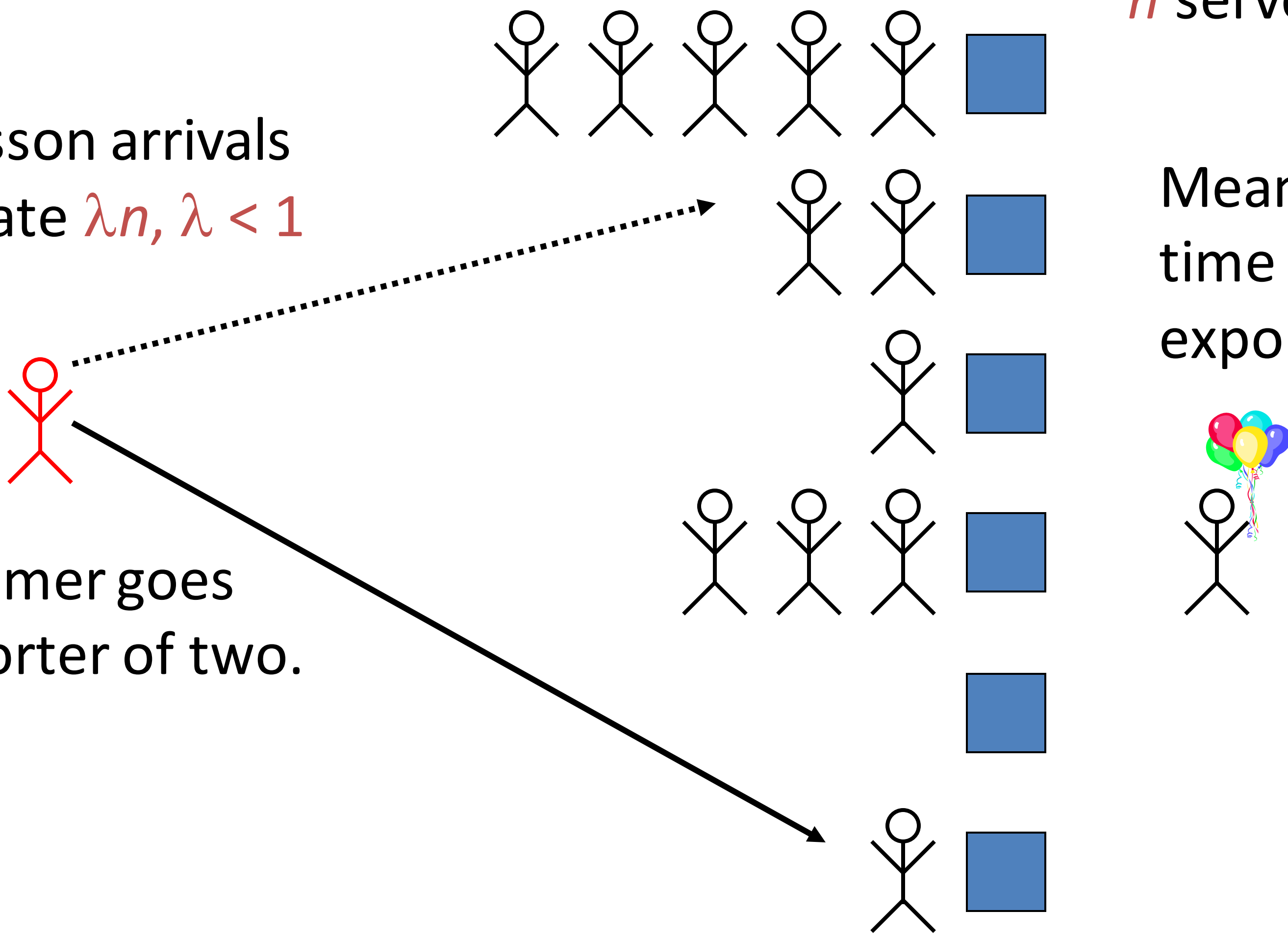
- Many more results…
  - Asymmetric load balancing
  - More balls than bins
  - Weighted balls
  - Unbalanced initial conditions
  - Insertions and deletion
  - Non-uniform chocies

Supermarket Model

n servers

Poisson arrivals at rate λn, $\lambda < 1$

Mean service time is 1, exponential



Customer goes to shorter of two.

Mathematical Description

- Let s_k be fraction of queues with at least k customers.
- System state: $(s_0(t), s_1(t), s_2(t), \ldots)$
- Fraction of queues with k customers is

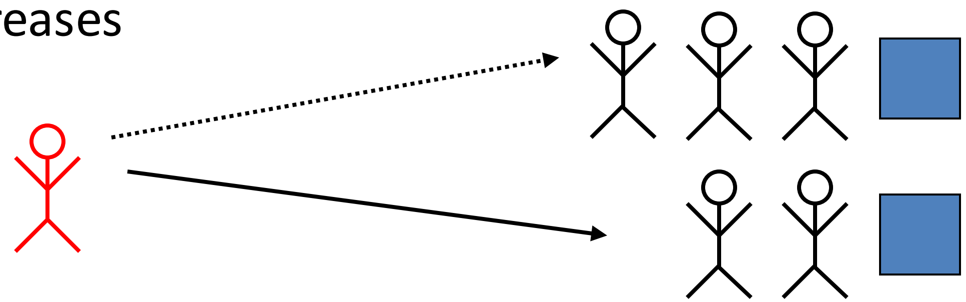
$$s_k - s_{k+1}$$

- Smallest of d random choices has k-1 customers with probability

$$s_{k-1}^d - s_k^d$$

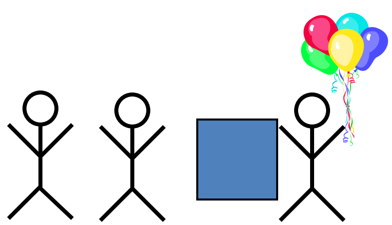
Setting Differential Equations

rate s_k increases



$$(\lambda n\, dt)\left(s_{k-1}^2 - s_k^2\right)/n$$

rate s_k decreases



$$n\left(s_k - s_{k+1}\right)(dt)/n$$

System behavior

- Expected behavior of process as differential equations.

$$ds_k / dt = \lambda\left(s_{k-1}^2 - s_k^2\right) - \left(s_k - s_{k+1}\right)$$

- Converges to fixed point

$$\pi = (\pi_0(t),\ \pi_1(t),\ \pi_2(t), \ldots) \quad \text{where}$$

$$ds_k / dt = 0$$

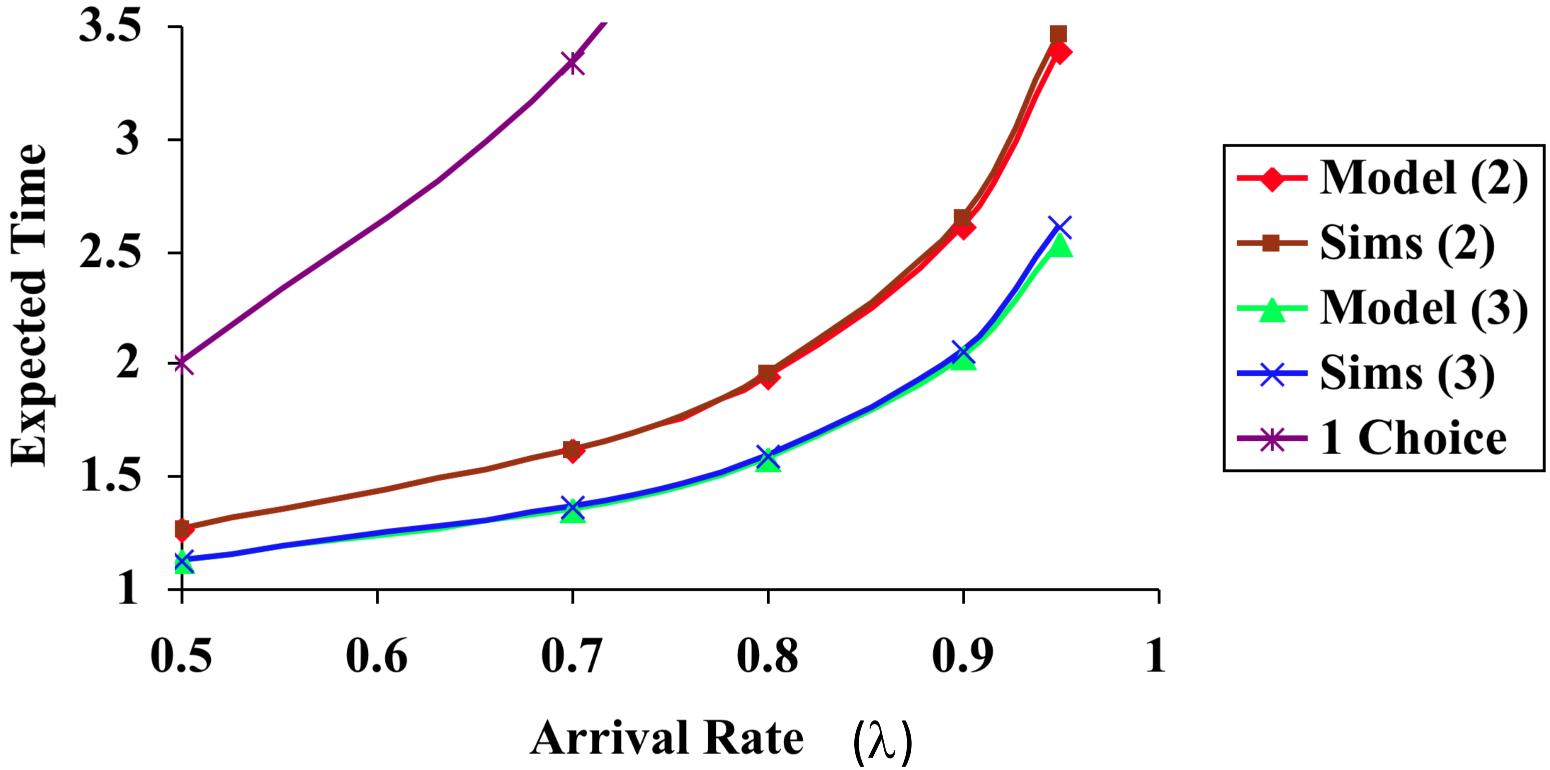
- At fixed point tails decrease doubly exponentially:

$$\pi_k = \lambda^{2^k - 1}$$

Relation to the Real World

- One choice yields exponential tails
  - $\pi_k = \lambda^k$ vs. $\pi_k = \lambda^{2^k - 1}$
- Two choices yields exponential gains in
  - Maximum queue size.
  - Average time in the system.
- Gains observable even for "small" systems.
  - 128 servers, average time in system within 2% of limiting model for $\lambda < 0.9$.

Simulations vs. Predictions



Balanced Allocations with Double Hashing

- Double hashing yields “same performance” for balanced allocations as fully random hashing.
  - Method 1: Double hashing starts with 2 random choices. Need to show adding choices can only help load distribution. So double hashing better than 2 random choices.
  - Method 2: Witness trees. Gets double hashing has same $\log \log n / \log d$ high order term.
  - Method 3: Differential equations. Same differential equations govern behavior of double hashing. Hence same empirical performance.

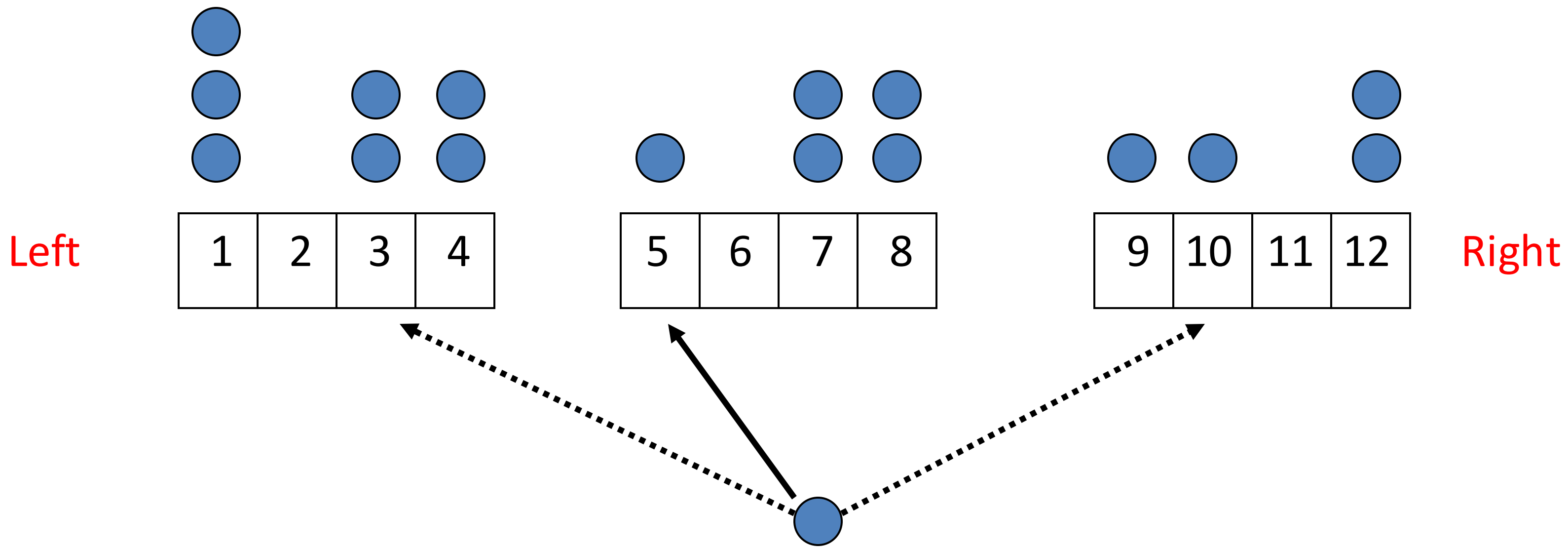
Variation: Double Hashing

- Let h_1 and h_2 be hash functions.
- For $i = 0, 1, 2, \ldots, k - 1$ and some f, let

$$g_i(x) = h_1(x) + i h_2(x) \bmod m$$

  - So 2 hash functions can mimic k hash functions.

Hashing Model: *d*-Left



Item placed in the shortest of *d* bins,
one chosen independently and uniformly
from each of *d* disjoint groups (of equal size).
Ties broken by placing toward the left.
Lookups can be parallelized!

d-Left Scheme

- *d*-Left is better than *d*-Random [V 99]
  - *d*-Random: max. load with *n* balls and *n* bins is log log *n* / log *d* with *d* choices.
  - *d*-Left: log log *n* / *d* with *d* choices.
- Extensions [MV 99]:
  - Gives simple differential equations analysis for *d*-Left schemes.
  - Provides analysis, numerical results.
  - Suggests further improvements.

Comparison

- Considering n balls, n bins.
- For d-Random, fraction of bins with load at least k falls like 2^{-d^k}
  - Max. load $\log \log n / \log d$
- For d-Left, fraction of bins with load at least k falls like $2^{-\phi(d)^{dk}}$
  - $\phi(d)$ is rate of growth of generalized Fib. numbers
  - Max. load $\log \log n / d \log \phi(d)$

$$\phi_2 = 1.618\ldots, \phi_3 = 1.839\ldots;\ 2^{(d-1)/d} < \phi_d < 2.$$

Example of *d*-left hashing

- Consider 4-left performance with average load of 6, using differential equations.

Insertions only

Load $\geq$ 1	1.0000
Load $\geq$ 2	1.0000
Load $\geq$ 3	1.0000
Load $\geq$ 4	0.9999
Load $\geq$ 5	0.9971
Load $\geq$ 6	0.8747
Load $\geq$ 7	0.1283
Load $\geq$ 8	1.273e-10
Load $\geq$ 9	2.460e-138

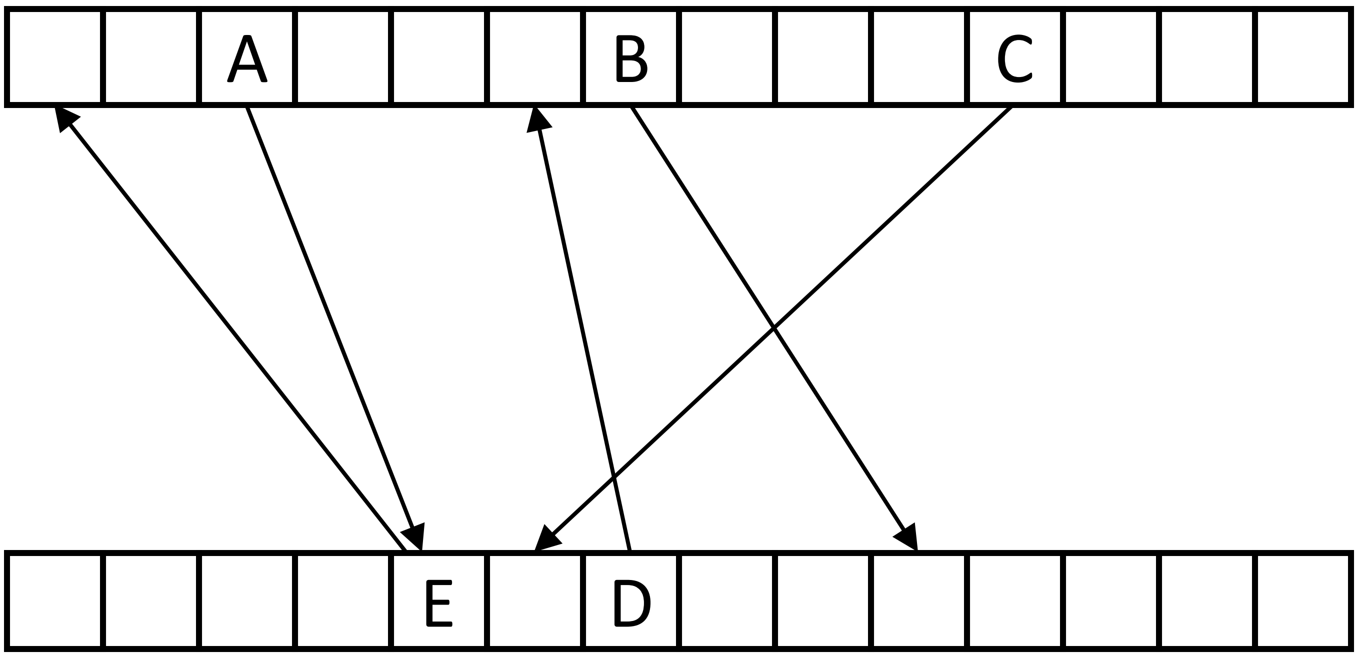
Alternating insertions/deletions
Steady state

Load $\geq$ 1	1.0000
Load $\geq$ 2	0.9999
Load $\geq$ 3	0.9990
Load $\geq$ 4	0.9920
Load $\geq$ 5	0.9505
Load $\geq$ 6	0.7669
Load $\geq$ 7	0.2894
Load $\geq$ 8	0.0023
Load $\geq$ 9	1.681e-27

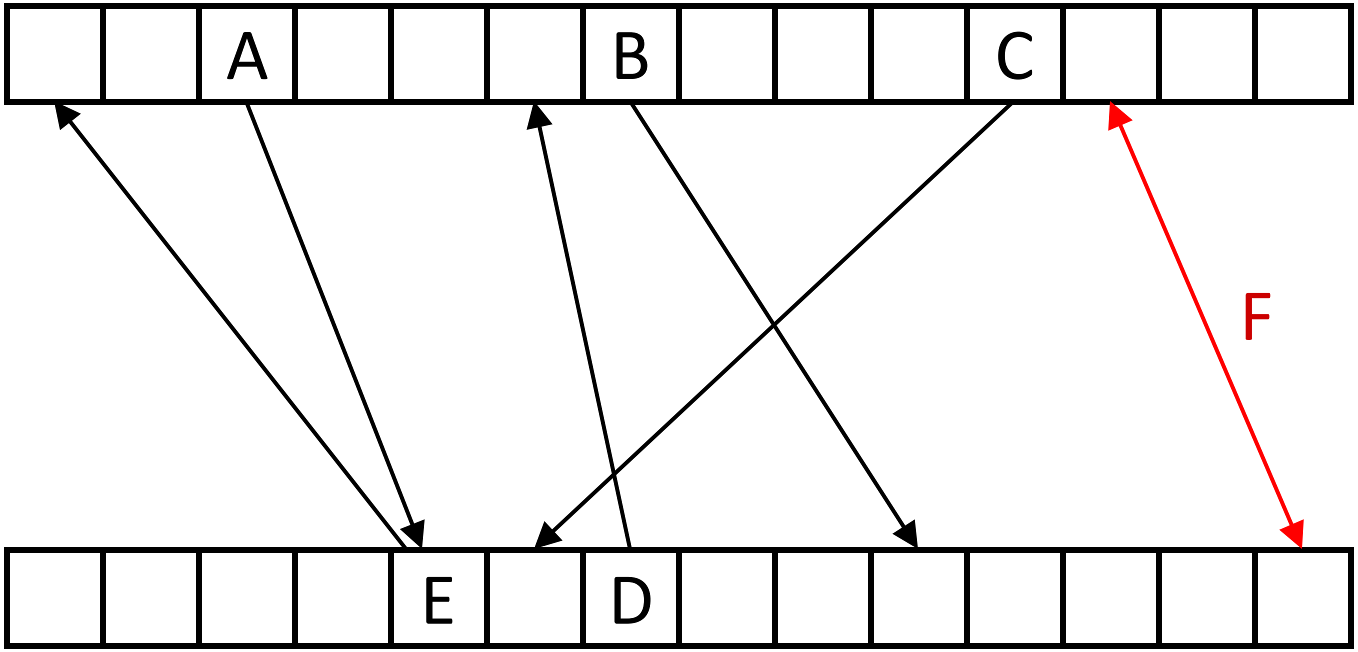
Cuckoo Hashing

- Basic scheme: each element gets two possible locations (uniformly at random).
- To insert *x*, check both locations for *x*. If one is empty, insert.
- If both are full, *x* kicks out an old element *y*. Then *y* moves to its other location.
- If that location is full, *y* kicks out *z*, and so on, until an empty slot is found.

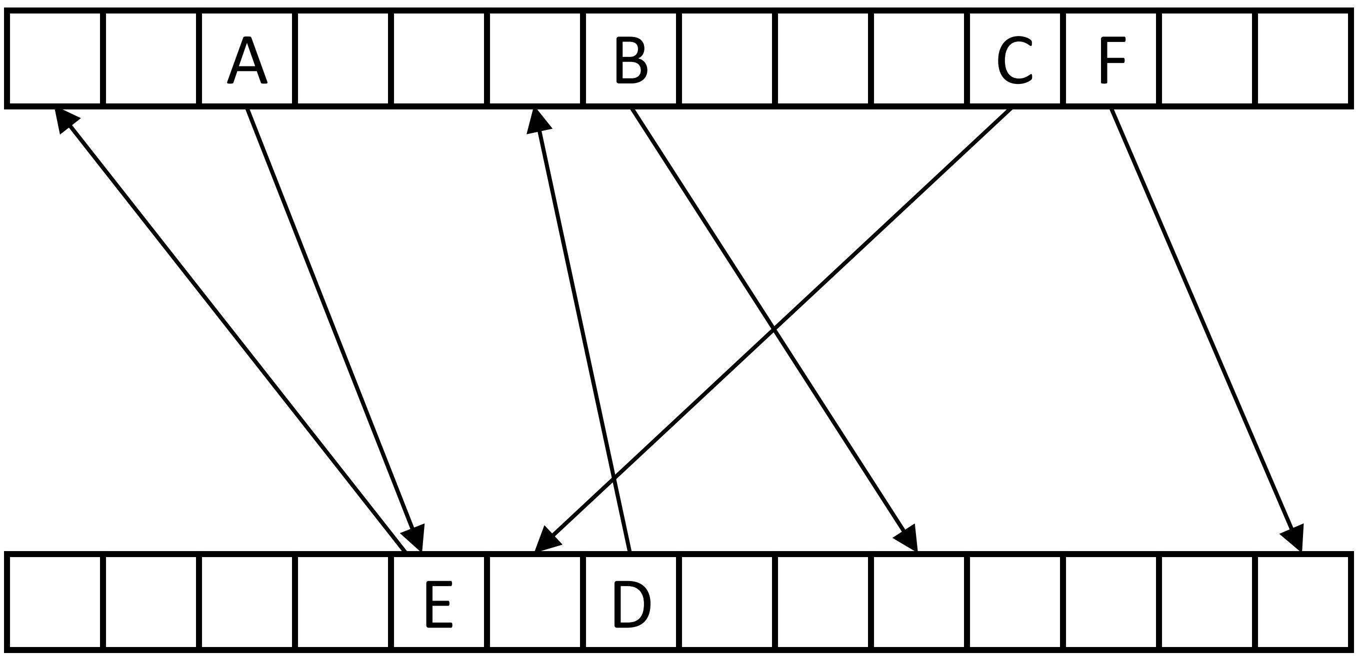
Cuckoo Hashing Examples



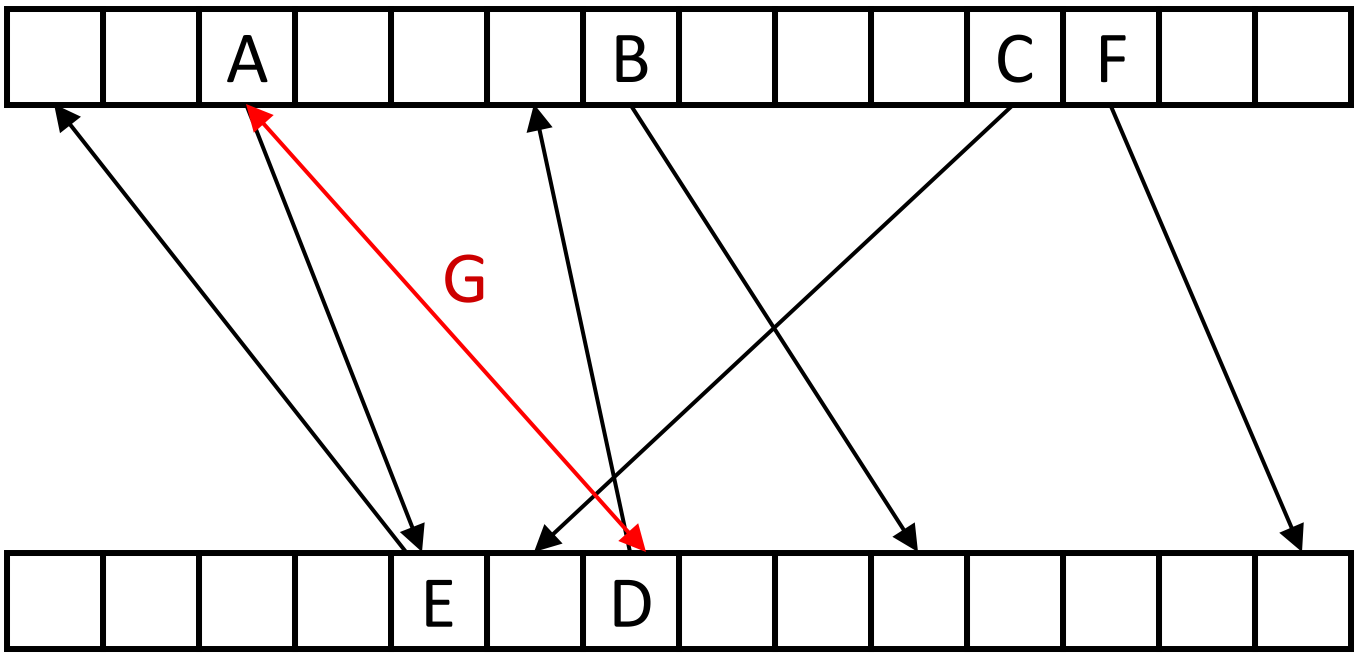
Cuckoo Hashing Examples



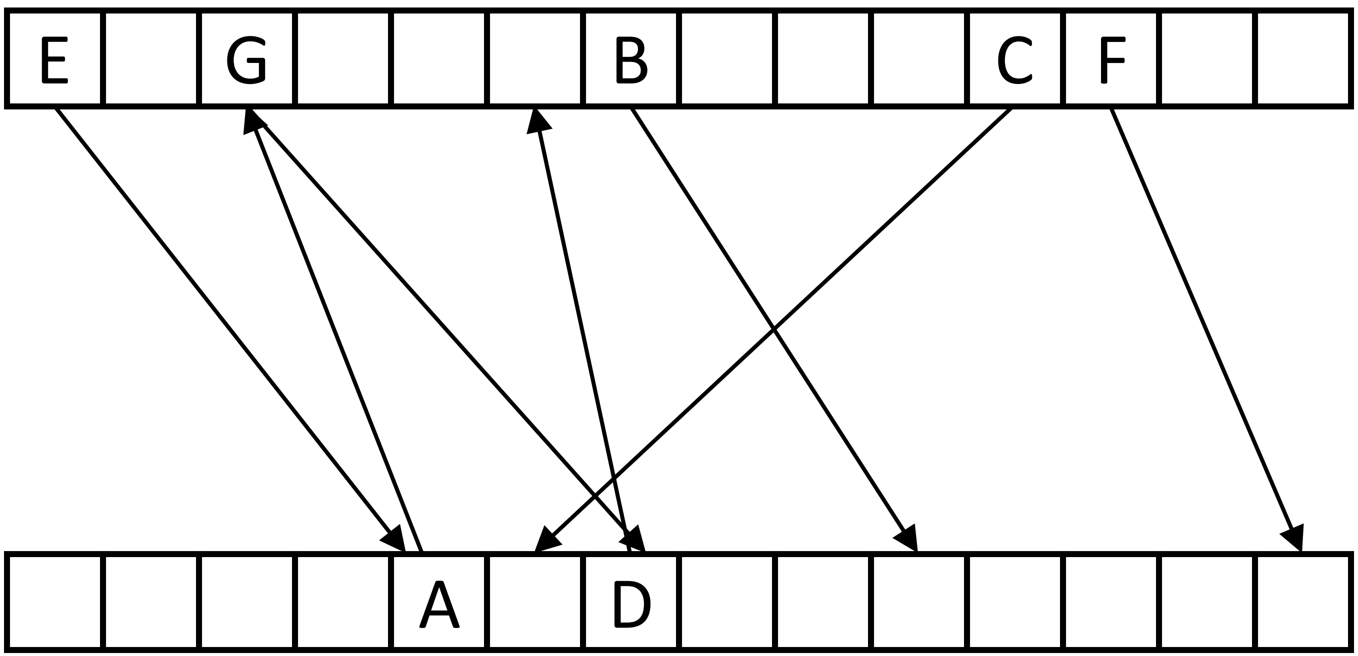
Cuckoo Hashing Examples



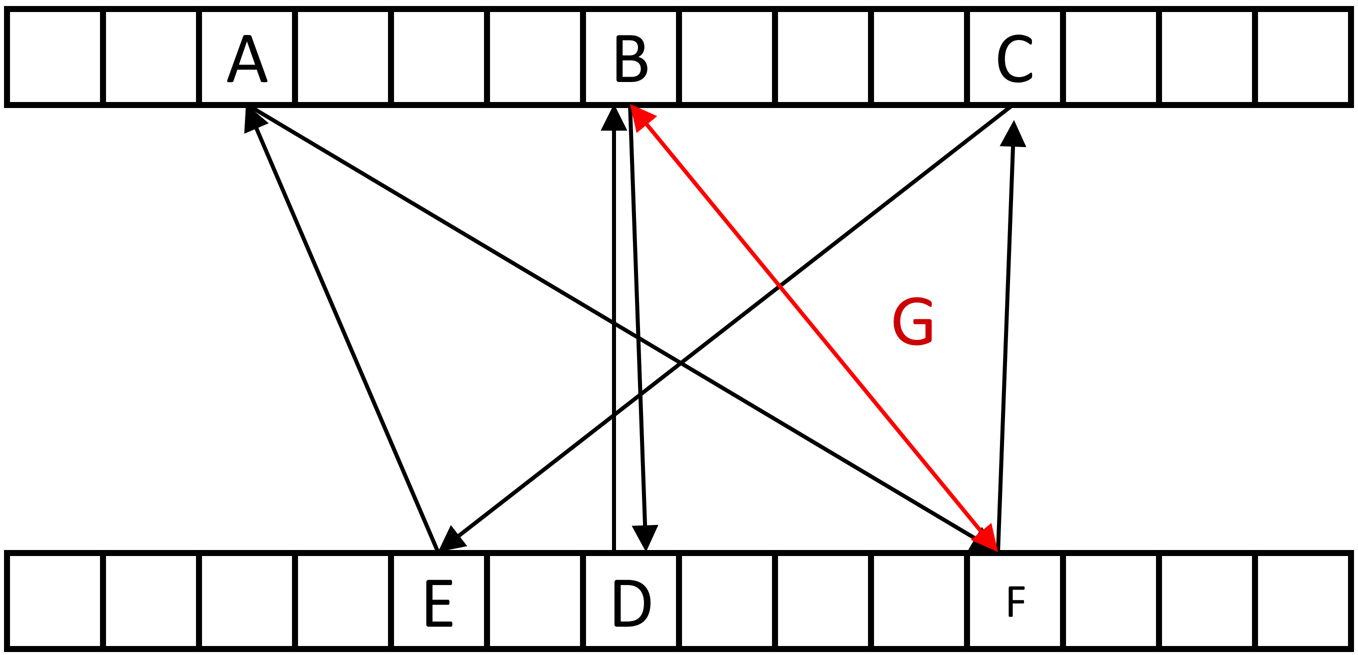
Cuckoo Hashing Examples



Cuckoo Hashing Examples



Cuckoo Hashing Examples



Multiple Choice vs. Cuckoo Hashing?

- Multiple-choice hashing yields tables with
  - High memory utilization.
  - Constant time look-ups.
  - Simplicity – easily coded, parallelized.
- Cuckoo hashing expands on this, combining multiple choices with ability to move elements.
  - Is moving elements worth the cost?
- Practical potential, and theoretically interesting!

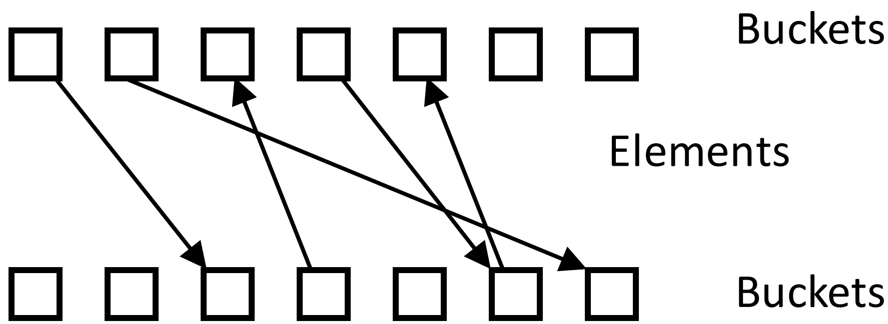
Good Properties of Cuckoo Hashing

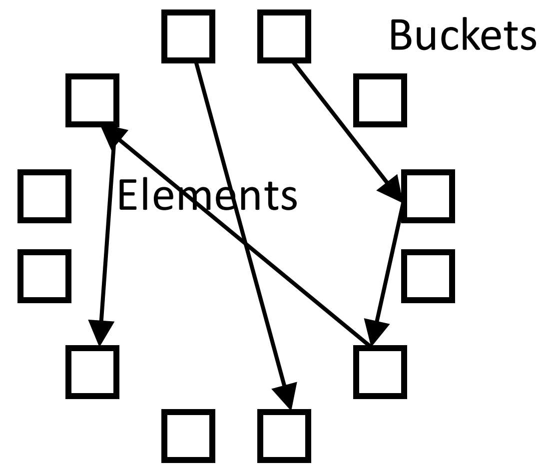
- *Worst case constant lookup time.*
- High memory utilizations possible.
- Simple to build, design.

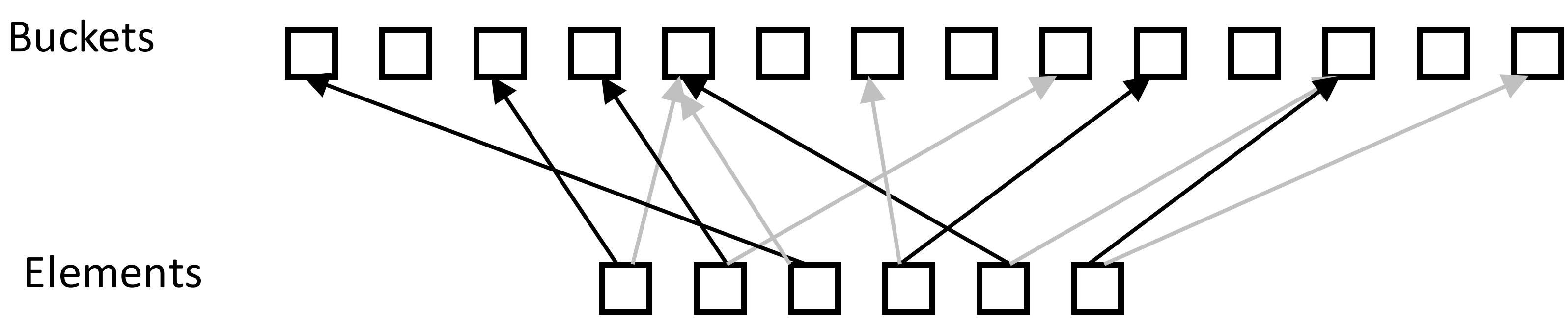
Cuckoo Hashing Failures

- Bad case 1: inserted element runs into cycles.
- Bad case 2: inserted element has very long path before insertion completes.
  - Could be on a long cycle.
- Bad cases occur with very small probability when load is sufficiently low.
- Theoretical solution: re-hash everything if a failure occurs.

Various Representations







Basic Performance

- For 2 choices, load less than 50%, n elements gives failure rate of $\Theta(1/n)$; maximum insert time $O(\log n)$.
- Related to random graph representation.
  - Each element is an edge, buckets are vertices.
  - Edge corresponds to two random choices of an element.
  - Small load implies small acyclic or unicyclic components, of size at most $O(\log n)$.

Natural Extensions

- More than 2 choices per element.
  - Very different : hypergraphs instead of graphs.
  - D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis.
  - Space efficient hash tables with worst case constant access time.
- More than 1 element per bucket.
  - M. Dietzfelbinger and C. Weidling.
  - Balanced allocation and dictionaries with tightly packed constant size bins.

Thresholds

Bucket Size 1

Choices	1	2	3	4	5	6
Load	0.5	0.918	0.976	0.992	0.997	0.999

2 Choices

Bucket size	1	2	3	4	5	8	10
Load	0.5	0.897	0.959	0.980	0.989	0.997	0.999

Proofs for Thresholds

- Most insight comes from viewing the process a branching tree from a node.
  - Cuckoo process as a hypergraph.
    - Each “edge” corresponds to a key, vertices are buckets.
  - Random neighborhood.
    - Distribution known/understood.
  - Locally a tree (with high probability).
- Then one fixes up the tree argument.
  - Challenging details.

Related to *l*-core

- The *l*-core is the maximal subgraph where each vertex has degree at least *l*.
- Can be found by peeling.
  - Take any vertex of degree at most *l*-1, remove it and corresponding edges.
- For cuckoo hashing with buckets of size *l*-1
  - If a bucket has at most *l*-1 keys that could be assigned to it, assign the keys to that bucket.
  - Remove bucket and keys from consideration.
- Cuckoo hashing succeeds when *l*-core is empty.
- Harder: cuckoo hashing succeeds when *l*-core has x remaining vertices, but less than (*l*-1)*x* edges.
  - The remaining core can be "matched".

Recursion Argument

- Let b be a node.
- Let q_h = probability node b is peeled after h rounds.
- Let p_j = probability node at distance h-j from b is peeled after j rounds. (Note $p_0 = 0$.) $p = \lim p_j$.

$$p_1 = \Pr\left[\mathrm{Bin}\left(\binom{m-1}{k-1}, \ k! \cdot \frac{c}{m^{k-1}}\right) \leq \ell - 2\right]$$

$$= \Pr[\mathrm{Po}(kc) \leq \ell - 2] \pm o(1),$$

$$p_{j+1} = \Pr\left[\mathrm{Bin}\left(\binom{m-1}{k-1}, \ k! \cdot \frac{c}{m^{k-1}} \cdot (1-p_j)^{k-1}\right) \leq \ell - 2\right]$$

$$= \Pr[\mathrm{Po}(kc(1-p_j)^{k-1}) \leq \ell - 2] \pm o(1), \text{ for } j = 1, \ldots, h-2.$$

$$p = \Pr[\mathrm{Po}(kc(1-p)^{k-1}) \leq \ell - 2].$$

$$q_h = \Pr[\mathrm{Po}(kc(1-p_{h-1})^{k-1}) \leq \ell - 1] \pm o(1).$$

Stashes

- A failure in cuckoo hashing occurs whenever one element can't be placed.
- Is that really necessary?
- What if we could keep one element unplaced? Or eight? Or $O(\log n)$? Or εn?
- Goal : Reduce the failure probability.

Motivation : CAMs

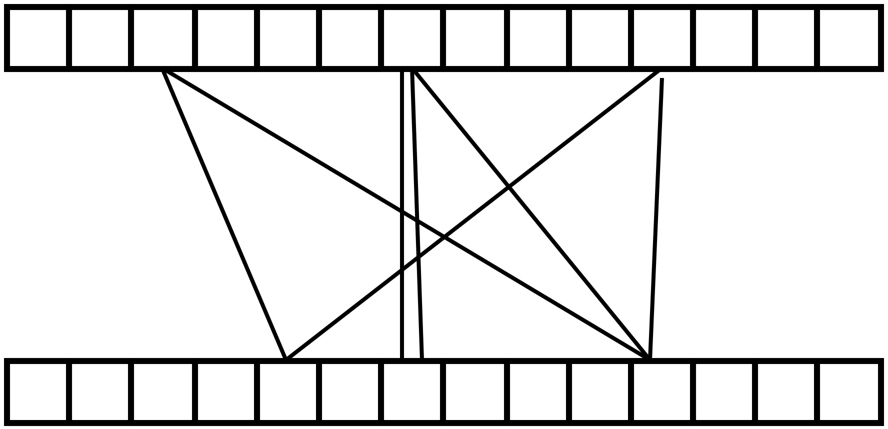
- CAM = content addressable memory
  - Fully associative lookup.
  - Usually expensive, so must be kept small.
  - Hardware solution, or a dedicated cache line in software.
- Not usually considered in theoretical work, but very useful in practice.
- Can we bridge this gap?
  - What can CAMs do for us?

A CAM-Stash

- Use a CAM to stash away elements that would cause failure.
- Intuition: if failures were independent, probability that *s* elements cause failures goes to $\Theta(1/n^s)$.
  - Failures not independent, but nearly so.
  - A stash holding a *constant* number of elements greatly reduces failure probability.
  - Implemented as hardware CAM or cache line.
- Lookup requires also looking at stash.
  - But generally empty.

Analysis Method

- Treat cells as vertices, elements as edges in bipartite graph.
- Count components that have excess edges to be placed in stash.
- Random graph analysis to bound excess edges.



6 vertices, 7 edges:
1 edge must go into stash.

A Simple Experiment

- 10,000 elements, table of size 24,000, 2 choices per element, 10^7 trials.

Stash Size	Needed Trials
0	9989861
1	10040
2	97
3	2
4	0

Random Walk Cuckoo Hashing

- When it is time to kick something out, choose one randomly.
- Small state, effective.
- Intuition : if fraction *p* of the buckets are empty, random walk "should" have fraction *p* of finding empty bucket at each step.
  - Clearly wrong, but nice intuition.
  - Suggests logarithmic time to find an empty slot.

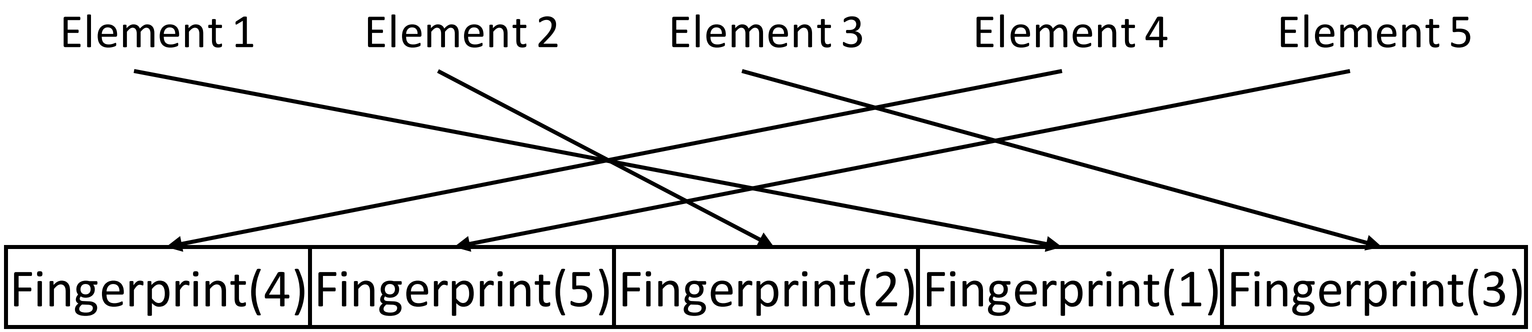
Some Progress

- Polylogarithmic bounds on insertion time.
- Open question: better bounds on performance of random walk cuckoo hashing?

Bloom Filters via Hash Tables

- Recall one could obtain an optimal static Bloom filter using perfect hashing
- Can we use multiple-choice hashing/cuckoo hashing to get a "near-perfect" hash table for a Bloom filter type object?

Perfect Hashing Approach



Near-Perfect Hash Functions via *d*-left Hashing

- Maximum load equals 1
  - Requires significant space to avoid all collisions, or some small fraction of spillovers.
- Maximum load greater than 1
  - Multiple buckets must be checked, and multiple cells in a bucket must be checked.
  - Not perfect in space usage.
    - In practice, 75% space usage is very easy.
    - In theory, can do even better.
- False positives increase with bucket size.

Modern Update : Cuckoo Filters

- Use a cuckoo hash table to obtain a near-perfect hash table
- Store a fingerprint in the hash table
- Can support insertion and deletion of keys
- Very space efficient
  - From cuckoo hash table construction, with buckets that hold multiple keys.

Cuckoo Filters : Issues

- Consider cuckoo hash table, 2 choice per key, 4 fingerprints of keys per bucket.
- Buckets fill, an item has to be moved.
- How do we know where to move it?
  - We don’t have the key any more.
  - Just the fingerprint.

Partial-key Cuckoo Hashing

- Can't use the key when moving a key.
- So we have to use the fingerprint instead.

$$h_1(x) = \text{hash}(x)$$

$$h_2(x) = h_1(x) \oplus \text{hash}(x\text{'s fingerprint})$$

- Note fingerprint is the same in both locations.
- Can compute h_1 from h_2 and vice versa with the stored fingerprint.

Partial-key Cuckoo Hashing

- Can't use the key when moving a key.
- So we have to use the fingerprint instead.

$$h_1(x) = \text{hash}(x)$$

$$h_2(x) = h_1(x) \oplus \text{hash}(x\text{'s fingerprint})$$

- Note fingerprint is the same in both locations.
- Can compute h_1 from h_2 and vice versa with the stored fingerprint.
- But now the two choices are limited, not completely random. Will this still work?

Partial-key Cuckoo Hashing

- Does it work?
- In practice, yes.
  - Essentially no discernible change in the threshold under reasonable settings.
- In theory, no.
  - You “need” logarithmic sized fingerprints…
  - But with a small constant factor.
  - So in practice it ends up OK.
- Open problem – better provable bounds on performance of partial-key cuckoo hashing.

Bit-Saving Tricks

- Every bit counts for space purposes.
- Bucket size of 4.
- Sort the fingerprints.
- Take the first 4 most significant bits.
- After sorting there are 3876 possible outcomes.
  - Less that 2^{12}.
  - So use only 12 bits to represent these 16.
  - Saves 1 bit per item.

Cuckoo Filter Performance

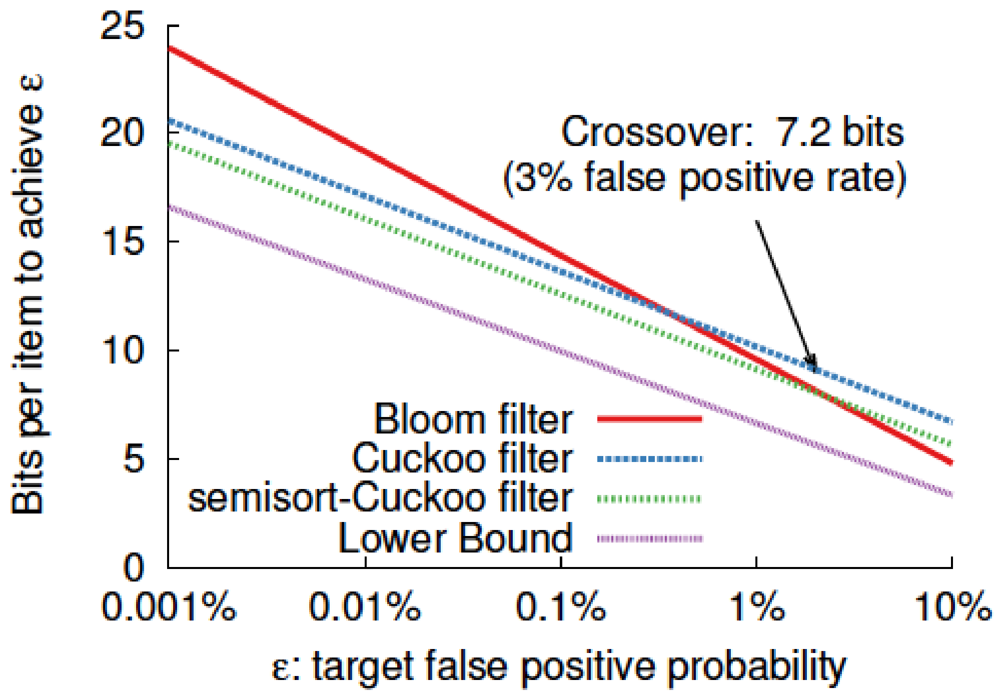


Figure 4: False positive rate vs. space cost per element. For low false positive rates ($< 3\%$), cuckoo filters require fewer bits per element than the space-optimized Bloom filters. The load factors to calculate space cost of cuckoo filters are obtained empirically.

Conclusion

- Power of two (or more) choices
  - A little choice usually goes a long way
- More and more uses for multiple-choice and cuckoo hashing
- Still lots of theoretical questions on cuckoo hashing to solve.

Exercise (Hard)

- Derive a family of differential equations that describe the d-left scheme of Vöcking.
- Assuming the differential equations are accurate, show that they yield a "Fibonacci exponential" decrease in the fraction of bins with load j as j increases for the case of n balls being placed into n bins.

Exercise

- Partial-key cuckoo hashing, with 2 choices per key, bucket size b, fingerprint size f bits, n items, table size $m = cn$ buckets for constant c.
- A failure happens if $2b+1$ keys map to the same pair of buckets.
- What is the expected number of sets of $2b+1$ keys that map to the same pair of buckets?
- What value of f is needed so this is o(1)?

Open Problems

- Better analysis of random walk cuckoo hashing.
- Better analysis of partial-key cuckoo hashing.
- Analyzing double hashing+cuckoo hashing. Can one prove the same thresholds apply?
- Analyzing double hashing and peeling on random hypergraphs. Can one prove the same thresholds apply?