Cuckoo Hashing

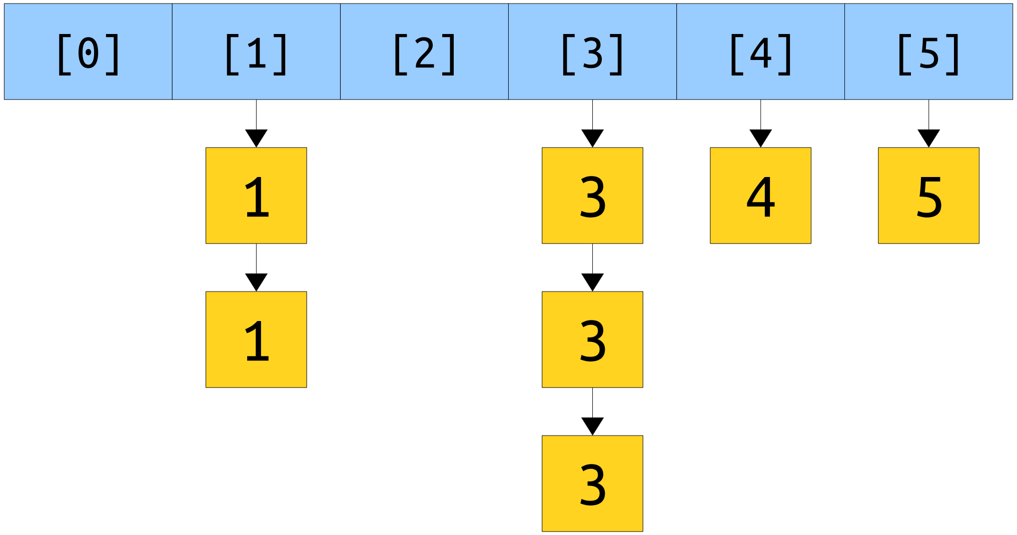
Outline for Today

- ***Cuckoo Hashing***
  - A simple, fast hashing system with worst-case efficient lookups.
- ***The Erdős–Rényi Model***
  - Randomly-generated graphs and their properties.
- ***Variants on Cuckoo Hashing***
  - Making a good idea even better.

Preliminaries: Hash Tables

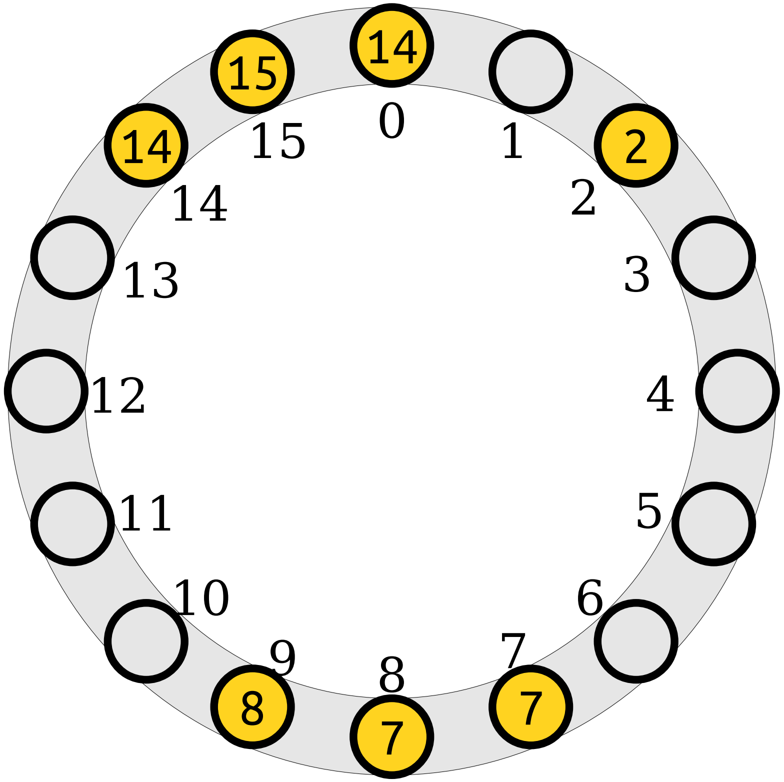
Collision Resolution

- All hash tables have to deal with hash collisions in some way.
- There are three general ways to do this:
  - ***Closed addressing:*** Store all colliding elements in an auxiliary data structure like a linked list or BST. (For example, standard chained hashing.)



Collision Resolution

- All hash tables have to deal with hash collisions in some way.
- There are three general ways to do this:
  - ***Closed addressing:*** Store all colliding elements in an auxiliary data structure like a linked list or BST. (For example, standard chained hashing.)
  - ***Open addressing:*** Allow elements to overflow out of their target bucket and into other spaces. (For example, linear probing hashing.)



Collision Resolution

- All hash tables have to deal with hash collisions in some way.
- There are three general ways to do this:
  - ***Closed addressing:*** Store all colliding elements in an auxiliary data structure like a linked list or BST. (For example, standard chained hashing.)
  - ***Open addressing:*** Allow elements to overflow out of their target bucket and into other spaces. (For example, linear probing hashing.)
  - ***Perfect hashing:*** Do something clever with multiple hash functions to eliminate collisions.
- What does that last option look like?

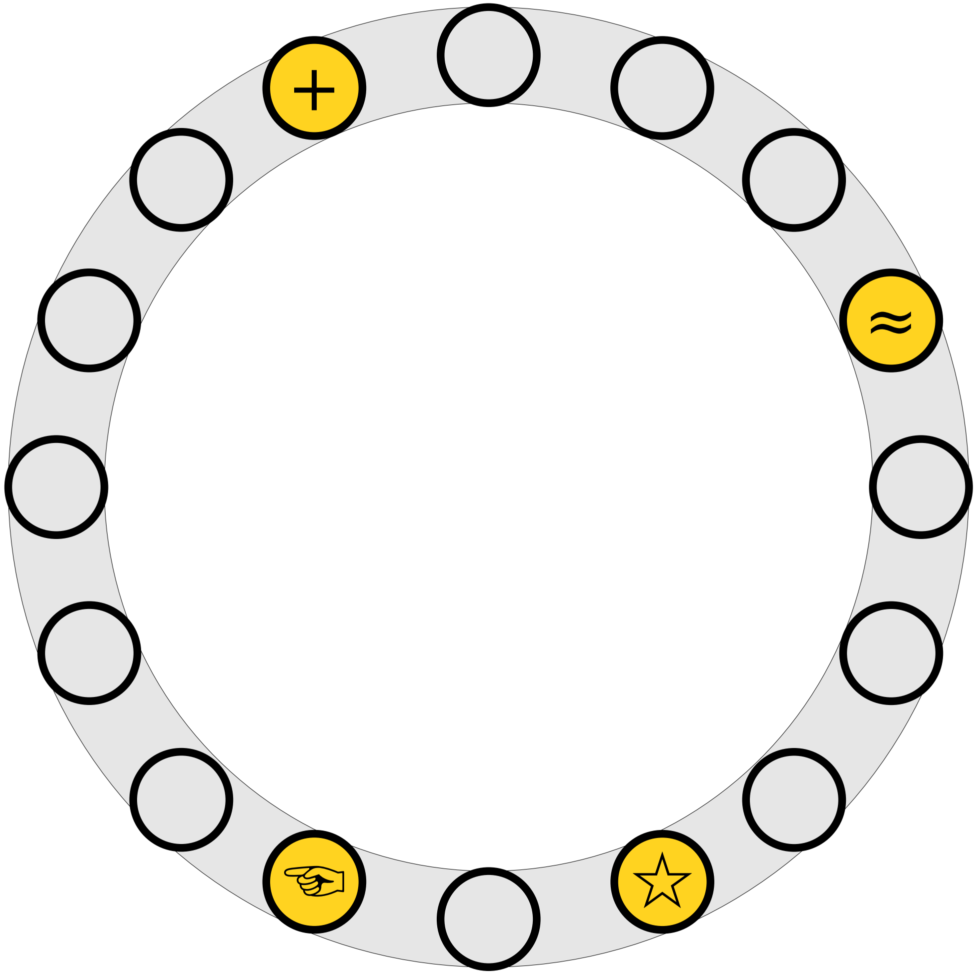
Cuckoo Hashing

Cuckoo Hashing

- Suppose we have a hash table with m slots.
- Unlike a normal hash table, we'll use *two* hash functions. We'll call them h_1 and h_2.
- Each hash function outputs a slot number in the set $\{ 0, 1, 2, \ldots, m - 1 \}$.
- We'll assume that these hash functions are truly random, with one constraint:

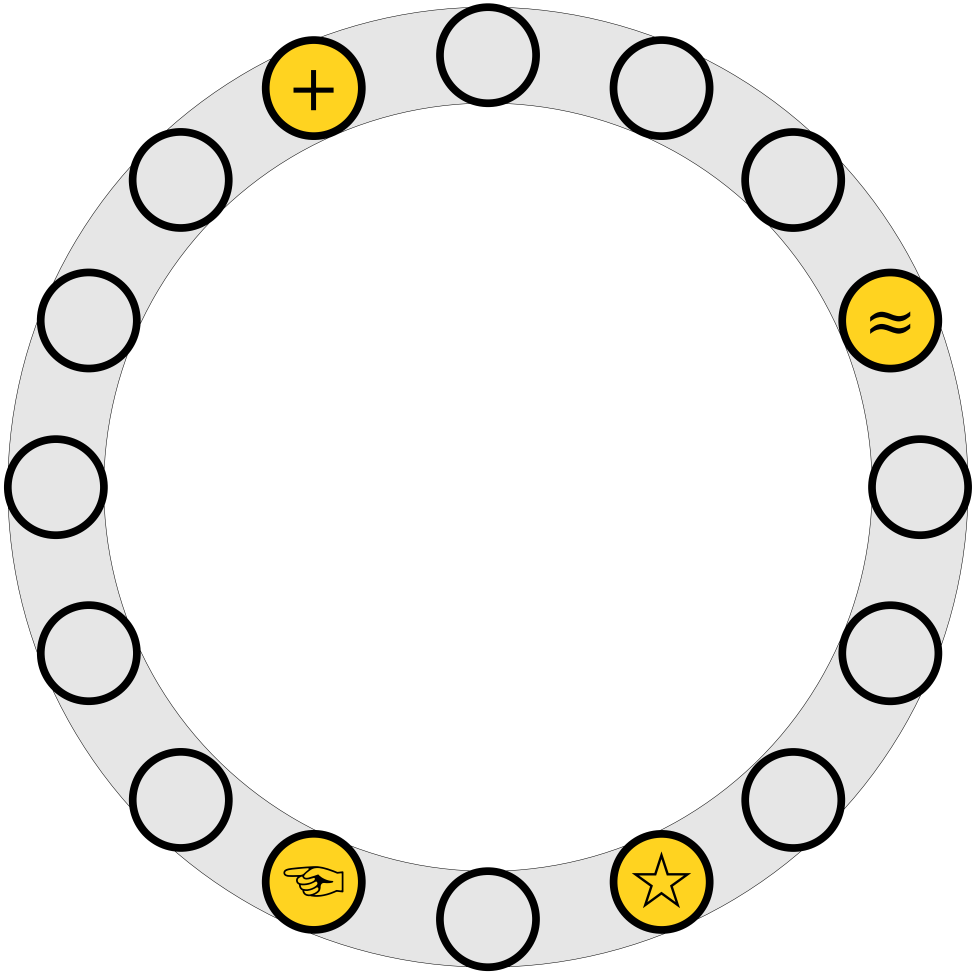
  $h_1(x) \neq h_2(x)$ for any key x.

- This is actually pretty easy to achieve both in theory and in practice – more on that later.



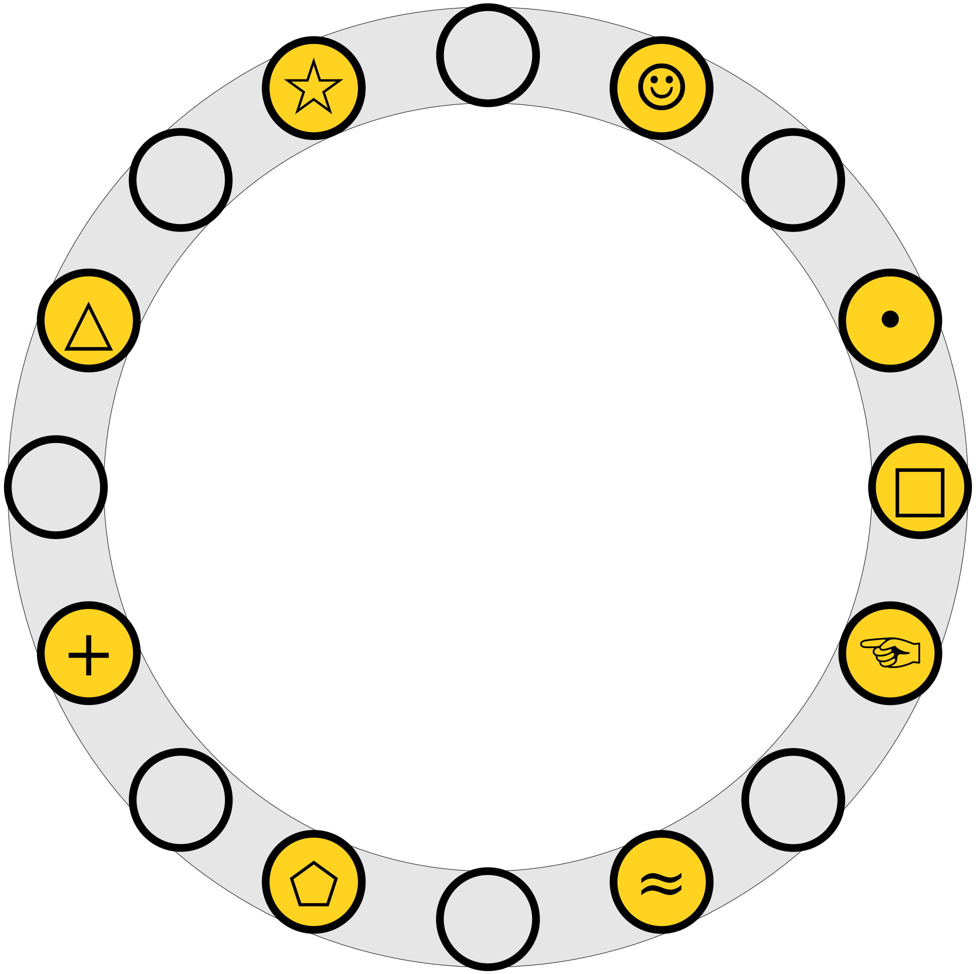
Cuckoo Hashing

- ***The Rule:*** Any item x must either be at position $h_1(x)$ or position $h_2(x)$ in the table.
- Lookups take *worst-case* O(1) time, since only two locations need to be checked.
- Deletions take *worst-case* O(1) time, since only two locations need to be checked.



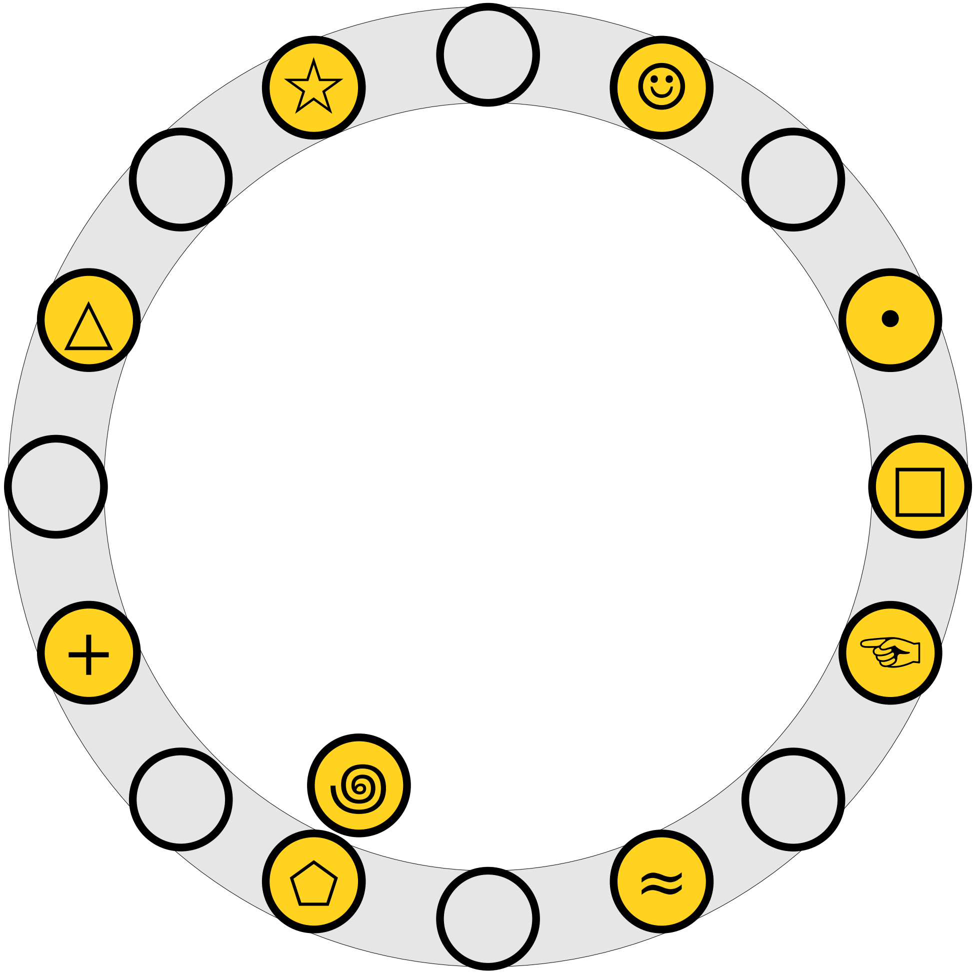
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.
- If that slot is full, kick out the element y that used to be in that slot and try placing it the other slot it can belong to (either $h_1(y)$ or $h_2(y)$).
- Repeat this process until all elements stabilize.



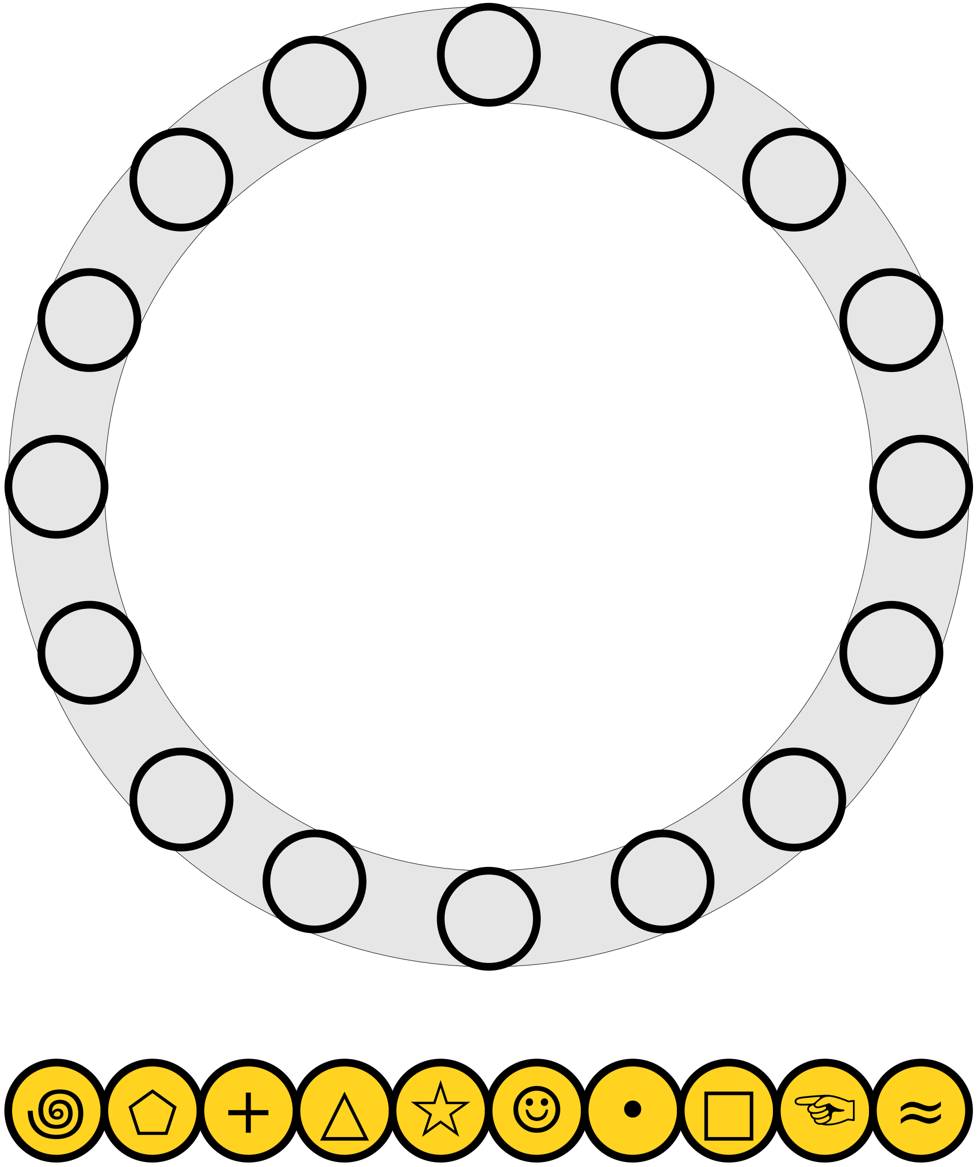
Cuckoo Hashing

- An insertion ***fails*** if the displacements form an infinite cycle.



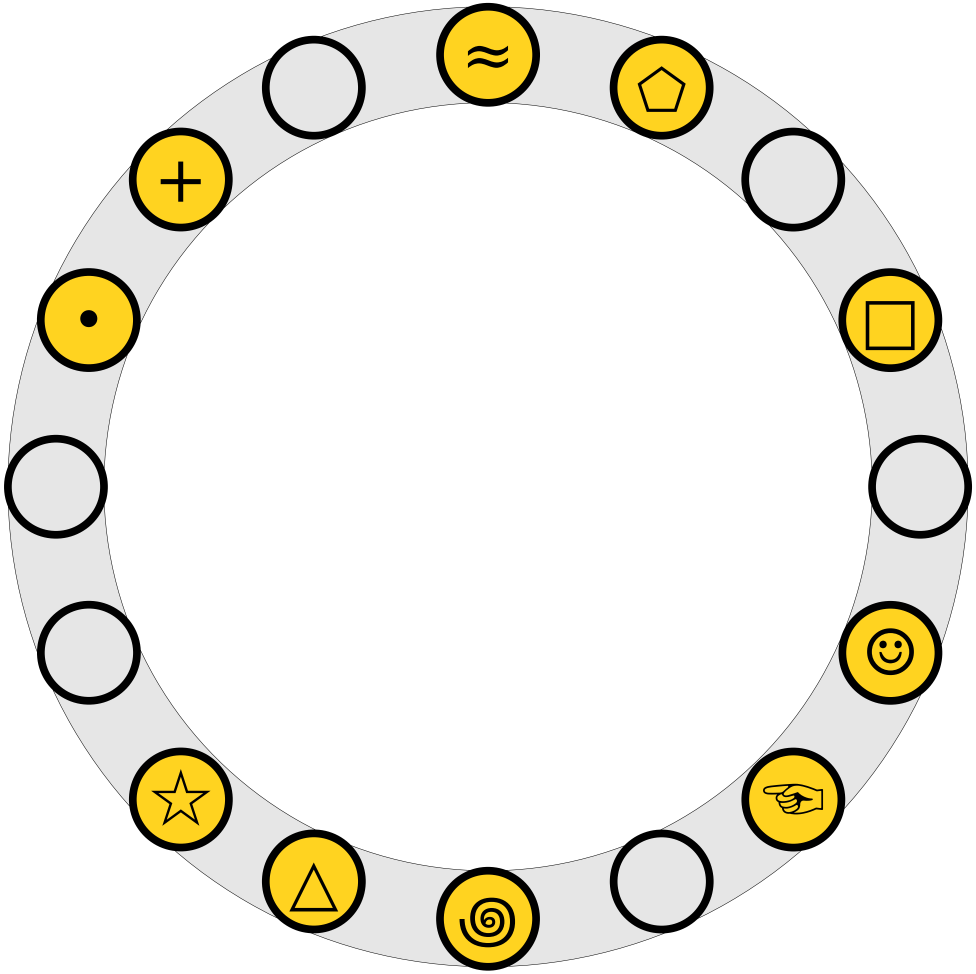
Cuckoo Hashing

- An insertion ***fails*** if the displacements form an infinite cycle.
- If that happens, perform a ***rehash*** by choosing a new h_1 and h_2 and inserting all elements back into the table.



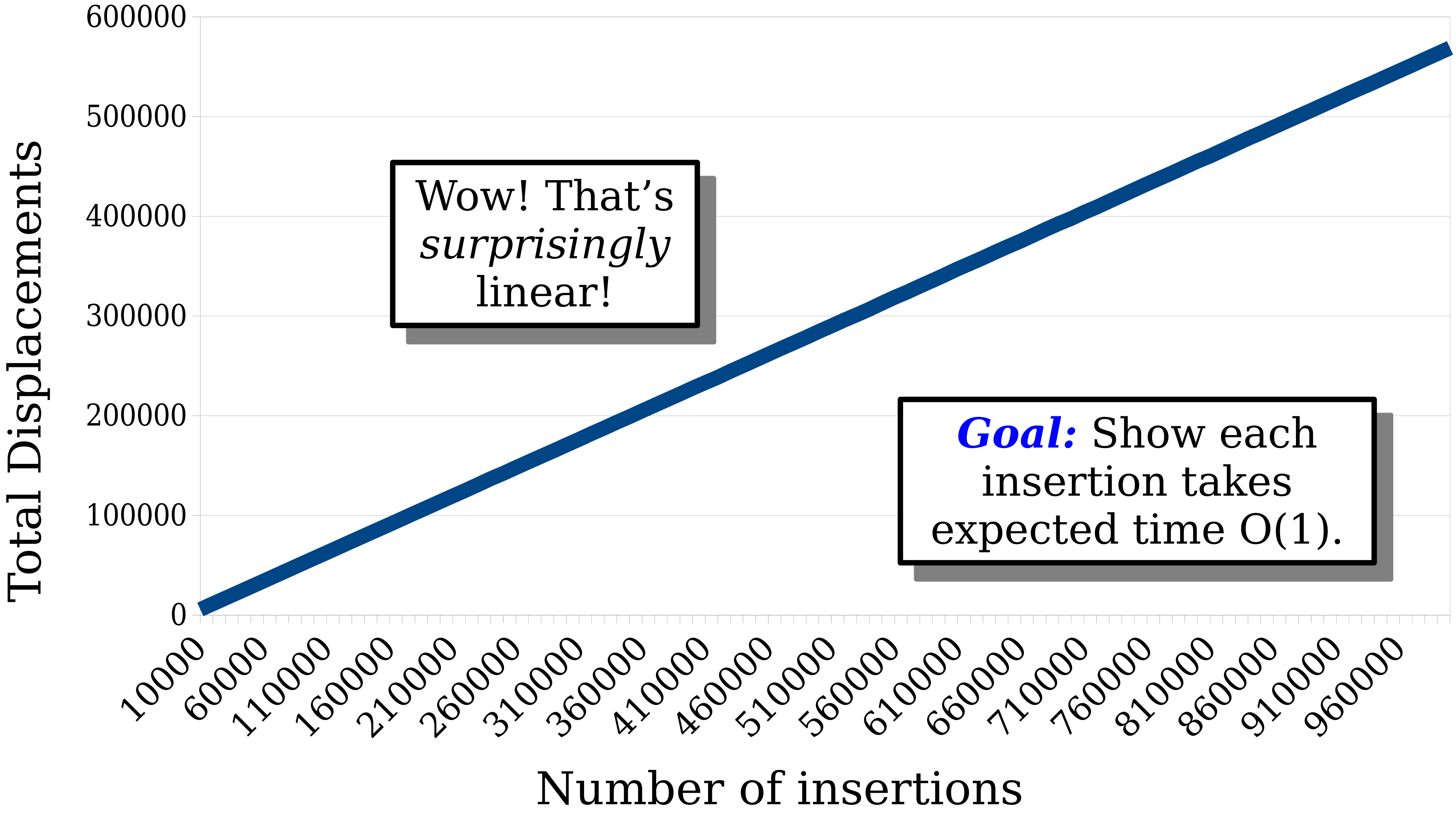
Cuckoo Hashing

- An insertion ***fails*** if the displacements form an infinite cycle.
- If that happens, perform a ***rehash*** by choosing a new h_1 and h_2 and inserting all elements back into the table.
- Multiple rehashes might be necessary before this succeeds – do you see why?



How efficient is cuckoo hashing?

Pro tip: When analyzing a data structure, it never hurts to get some empirical performance data first.

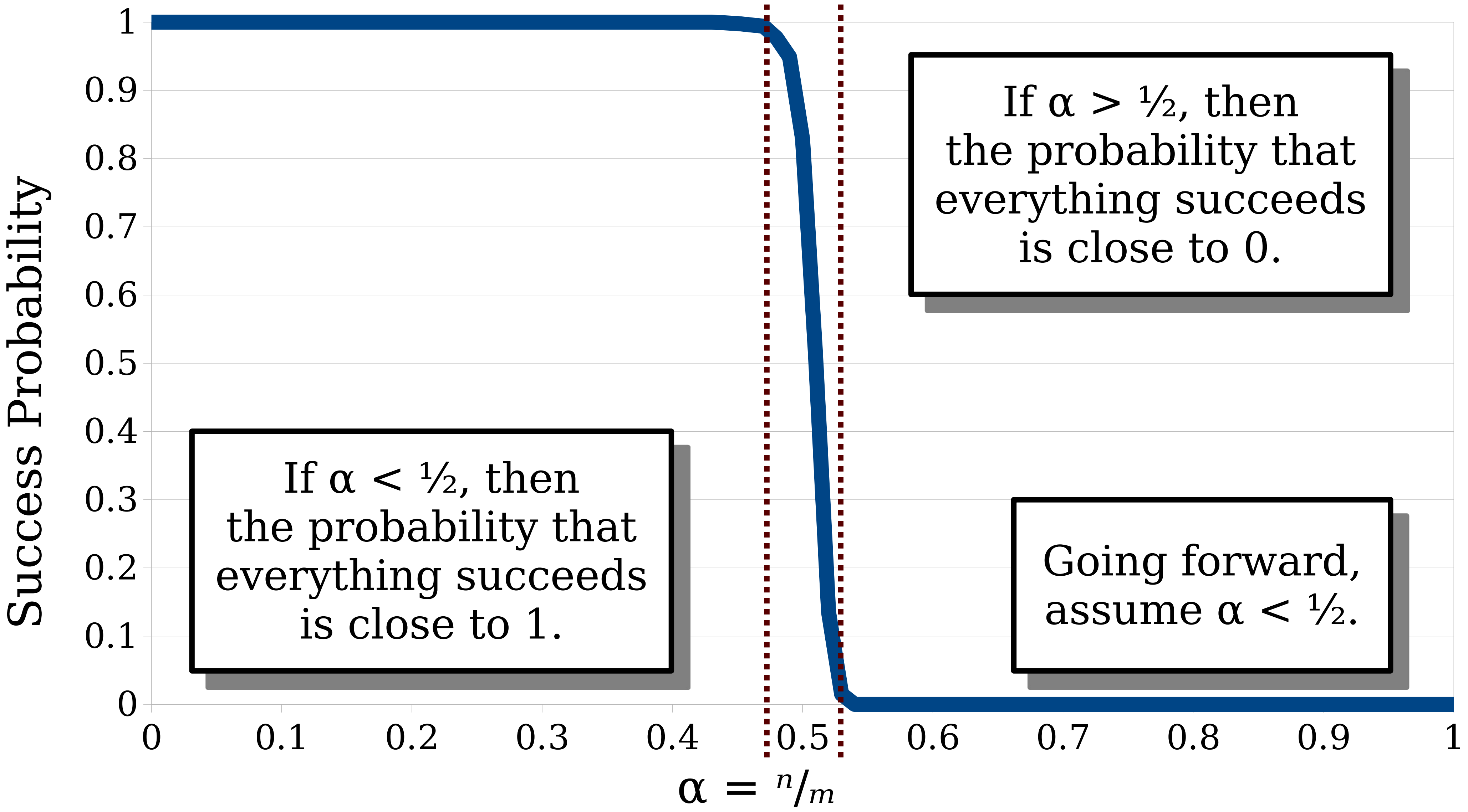


Wow! That's *surprisingly* linear!

Goal: Show each insertion takes expected time O(1).

Suppose we store *n* total elements in a table with *m* slots, where $n < \frac{1}{2}m$.

How many total displacements occur across all insertions?



If α > ½, then the probability that everything succeeds is close to 0.

If α < ½, then the probability that everything succeeds is close to 1.

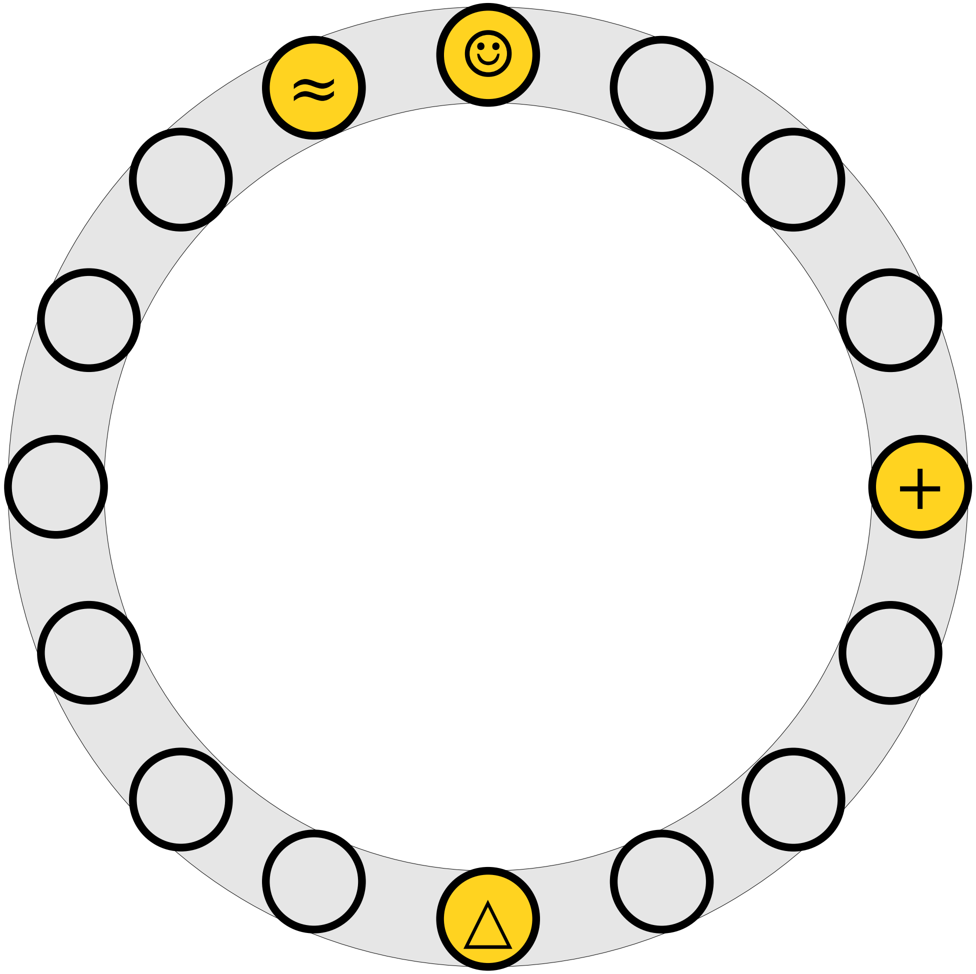
Going forward, assume α < ½.

Suppose we have *m* slots and store *n* total elements. What is the probability that all the insertions succeed, as a function of the ***load factor*** α = ⁿ/ₘ?

Goal: Show that insertions take expected time O(1), under the assumption that $n = \alpha m$ for some $\alpha < ½$.

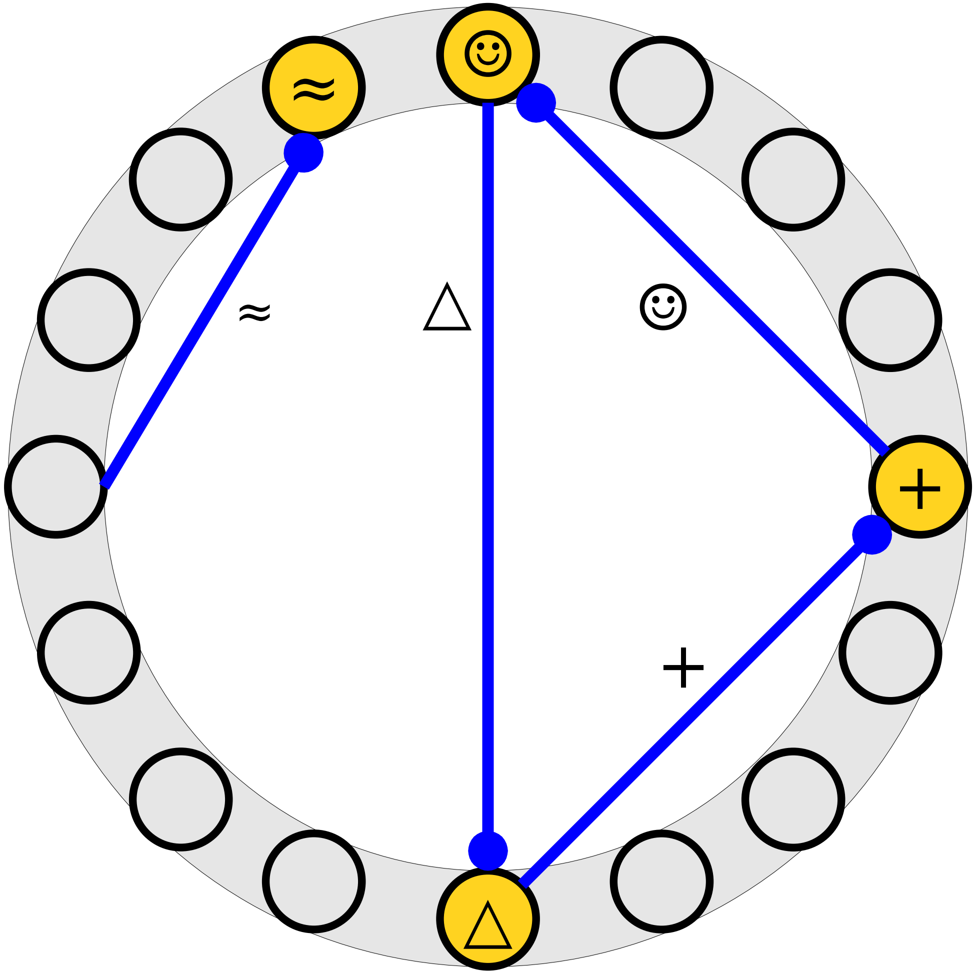
Analyzing Cuckoo Hashing

- The analysis of cuckoo hashing is more difficult than it might at first seem.
- ***Challenge 1:*** We may have to consider hash collisions across multiple hash functions.
- ***Challenge 2:*** We need to reason about chains of displacement, not just how many elements land somewhere.
- To resolve these challenges, we'll need to bring in some new techniques.



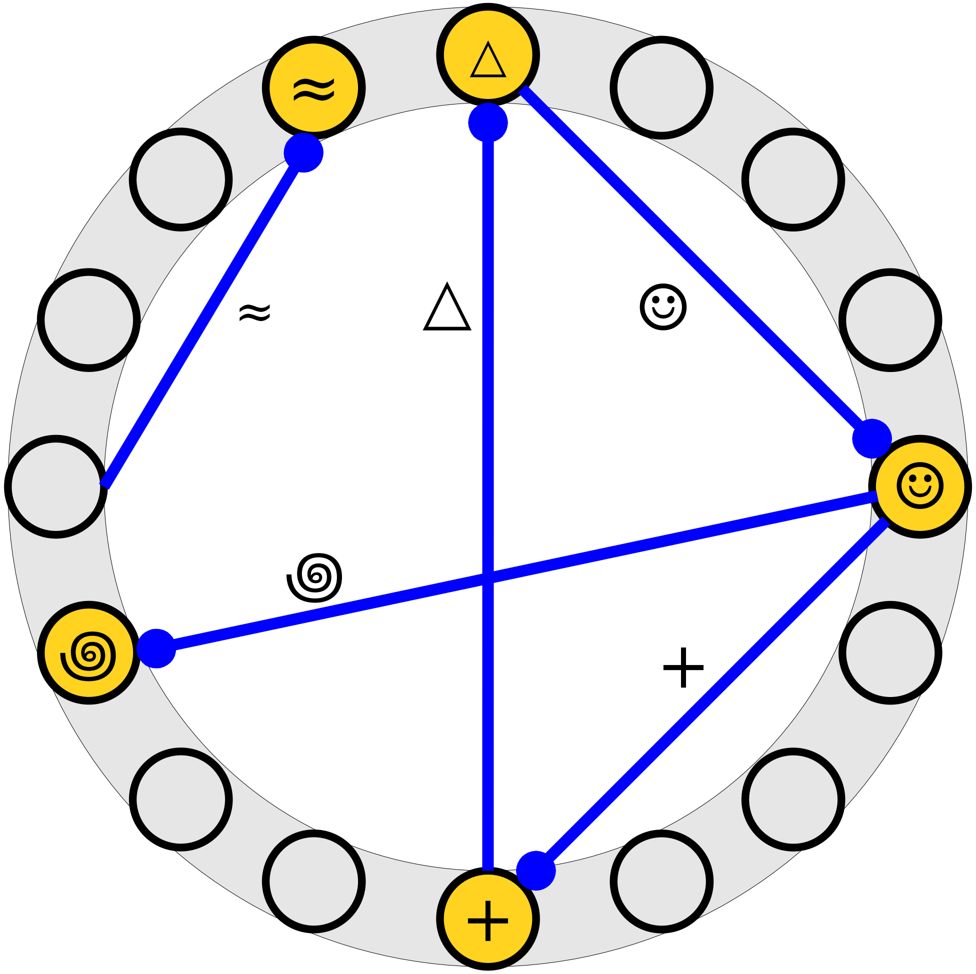
The Cuckoo Graph

- The ***cuckoo graph*** is a (multi)graph derived from a cuckoo hash table.
- Each table slot is a node.
- Each element is an edge linking the slots where it can be placed.
- An item's position in the table is denoted with a dot at the end of the line.
- Each node has at most one dot touching it.



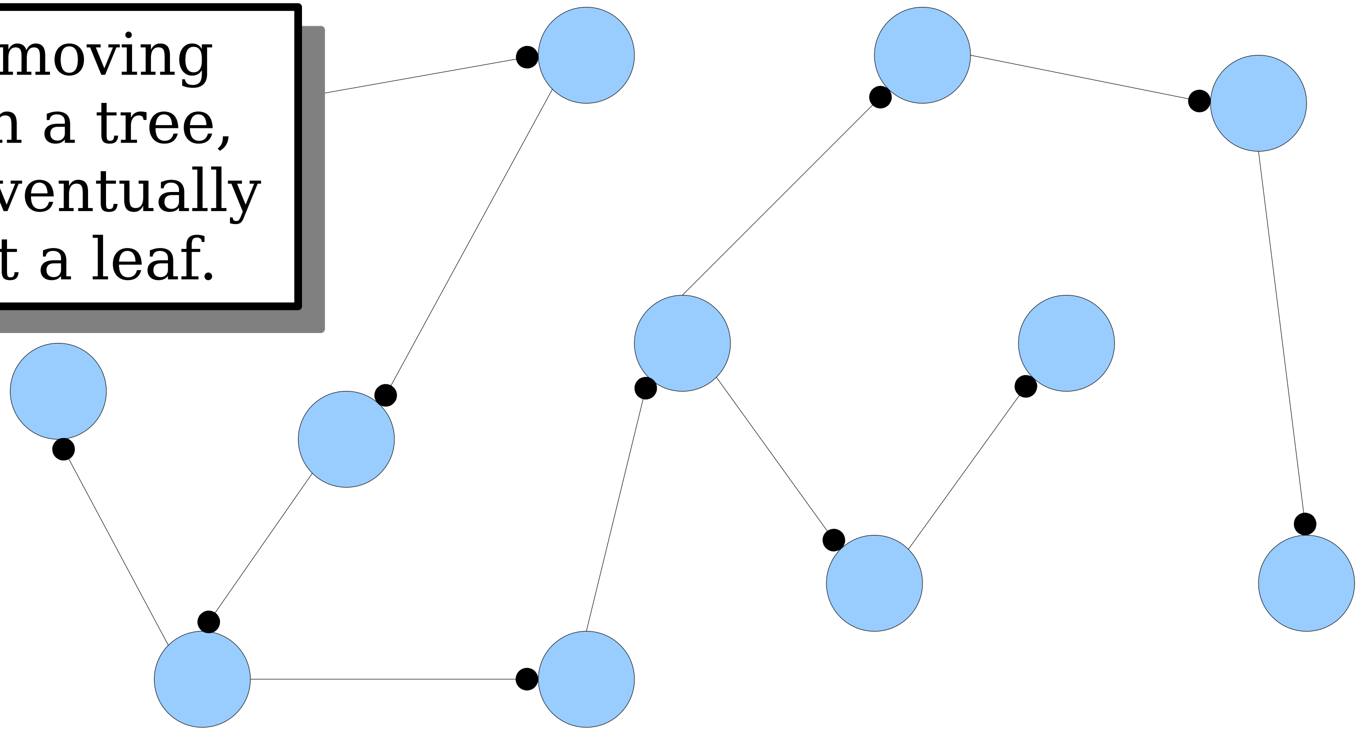
The Cuckoo Graph

- Inserting an element into a cuckoo hash table adds a new edge to the graph linking two nodes (slots).
- The chain of displacements corresponds to flipping edges.



The Cuckoo Graph

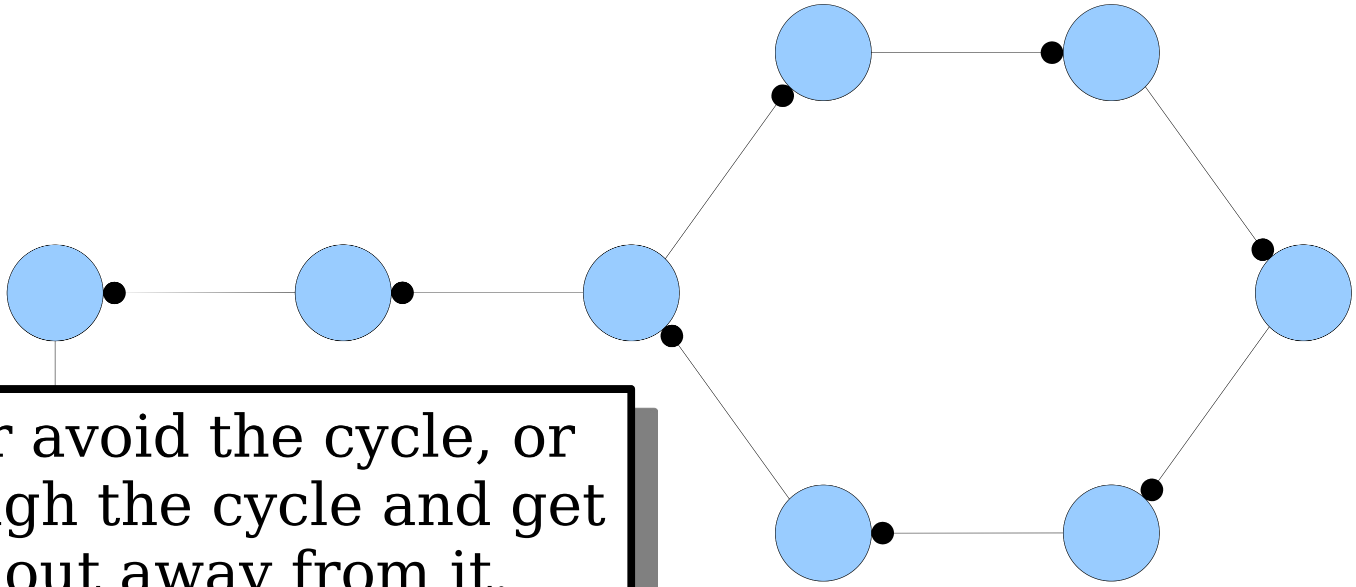
- ***Claim 1:*** If *x* is inserted into a cuckoo hash table, the insertion succeeds if the connected component containing *x* contains either no cycles or only one cycle.



We’re moving through a tree, which eventually ends at a leaf.

The Cuckoo Graph

- ***Claim 1:*** If *x* is inserted into a cuckoo hash table, the insertion succeeds if the connected component containing *x* contains either no cycles or only one cycle.



We either avoid the cycle, or loop through the cycle and get kicked out away from it.

The Cuckoo Graph

- ***Claim 2:*** If *x* is inserted into a cuckoo hash table, the insertion fails if the connected component containing *x* contains more than one cycle.

Why?

Formulate a hypothesis!

The Cuckoo Graph

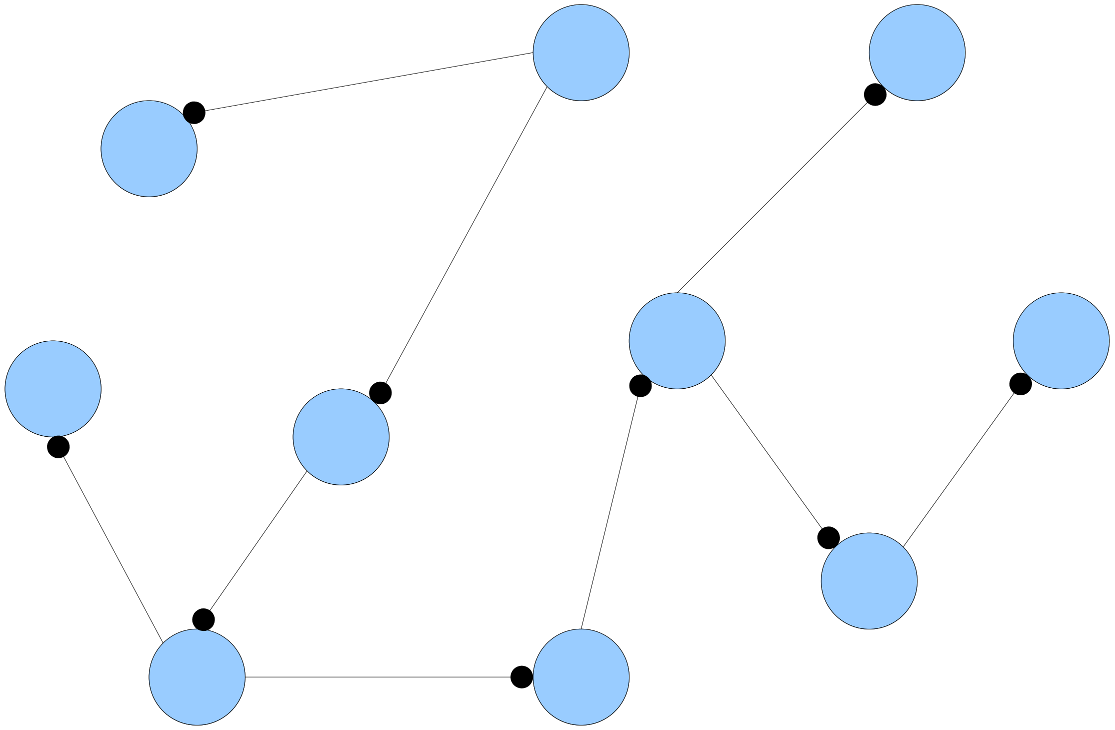
- ***Claim 2:*** If *x* is inserted into a cuckoo hash table, the insertion fails if the connected component containing *x* contains more than one cycle.

Why?

Discuss with your neighbors!

The Cuckoo Graph

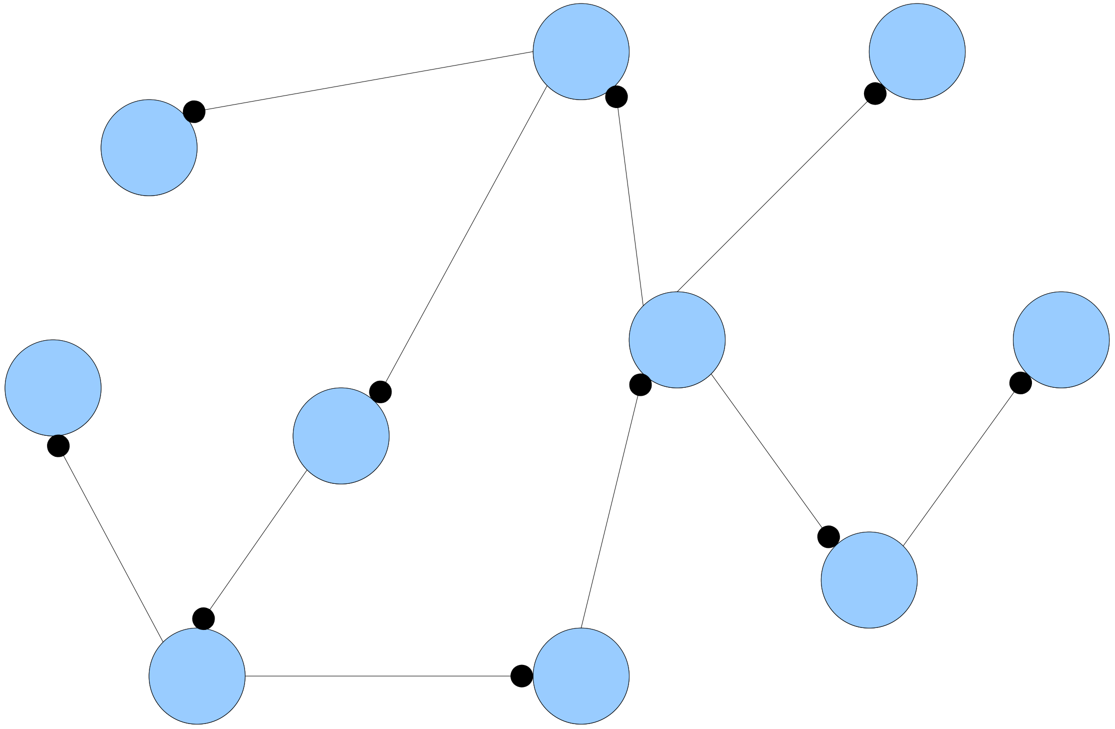
- ***Claim 2:*** If *x* is inserted into a cuckoo hash table, the insertion fails if the connected component containing *x* contains more than one cycle.



No cycles: The CC containing our edge is a tree. A tree with *k* nodes has *k* – 1 edges.

The Cuckoo Graph

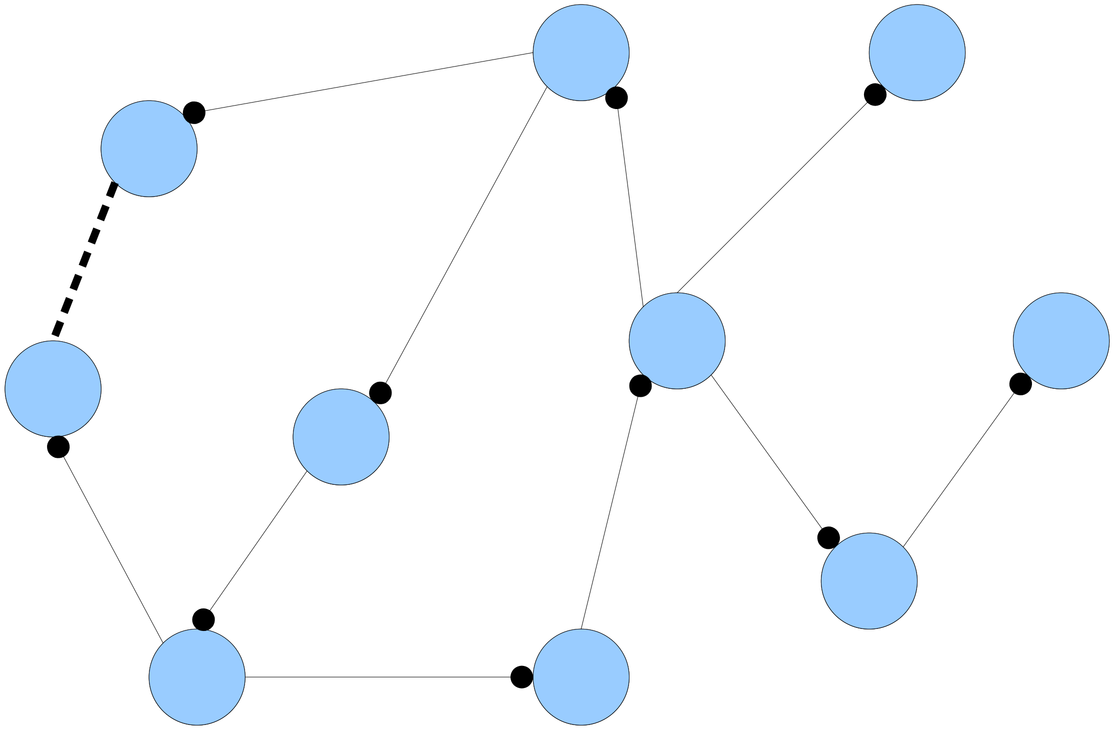
- ***Claim 2:*** If *x* is inserted into a cuckoo hash table, the insertion fails if the connected component containing *x* contains more than one cycle.



One cycle: We've added an edge, giving *k* nodes and *k* edges.

The Cuckoo Graph

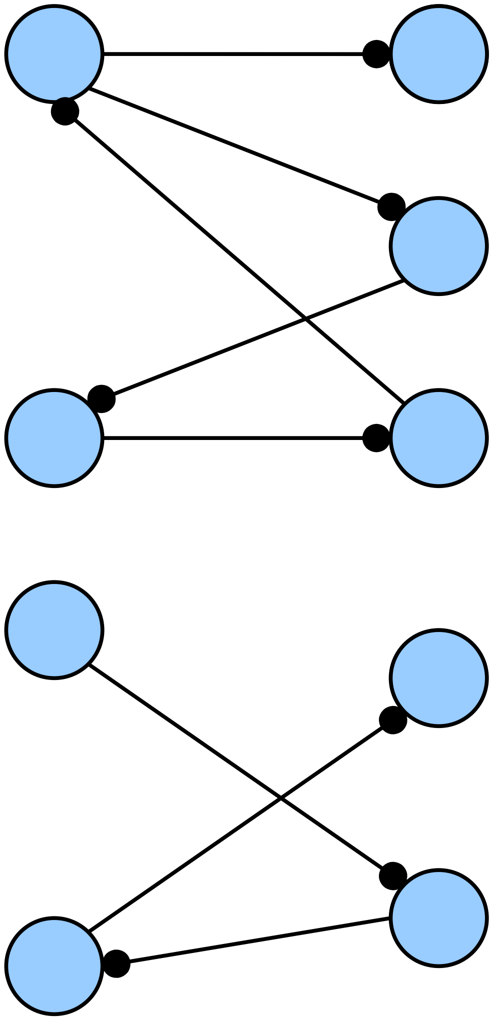
- ***Claim 2:*** If *x* is inserted into a cuckoo hash table, the insertion fails if the connected component containing *x* contains more than one cycle.



Two cycles: There are *k* nodes and *k*+1 edges. There are too many edges to place at most one item per node.

The Cuckoo Graph

- A connected component of a graph is called ***complex*** if it contains two or more cycles.
- ***Theorem:*** Insertion into a cuckoo hash table succeeds if and only if the resulting cuckoo graph has no complex connected components.



How big are the connected components in the cuckoo graph?

(This tells us how much work we do on a successful insertion.)

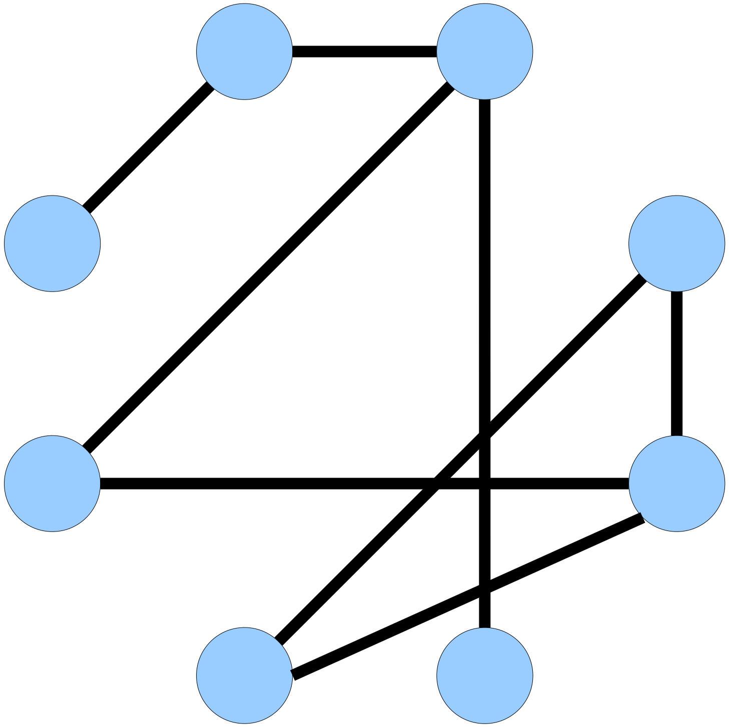
What is the probability that a connected component in the cuckoo graph is complex?

(This lets us see how much time we should expect to spend rehashing.)

The Erdős–Rényi model

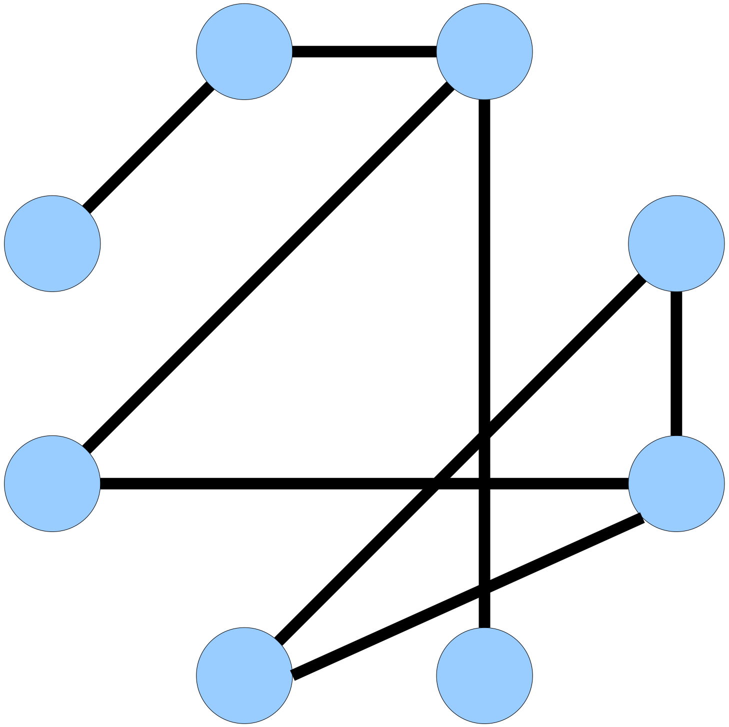
Random Graph Evolution

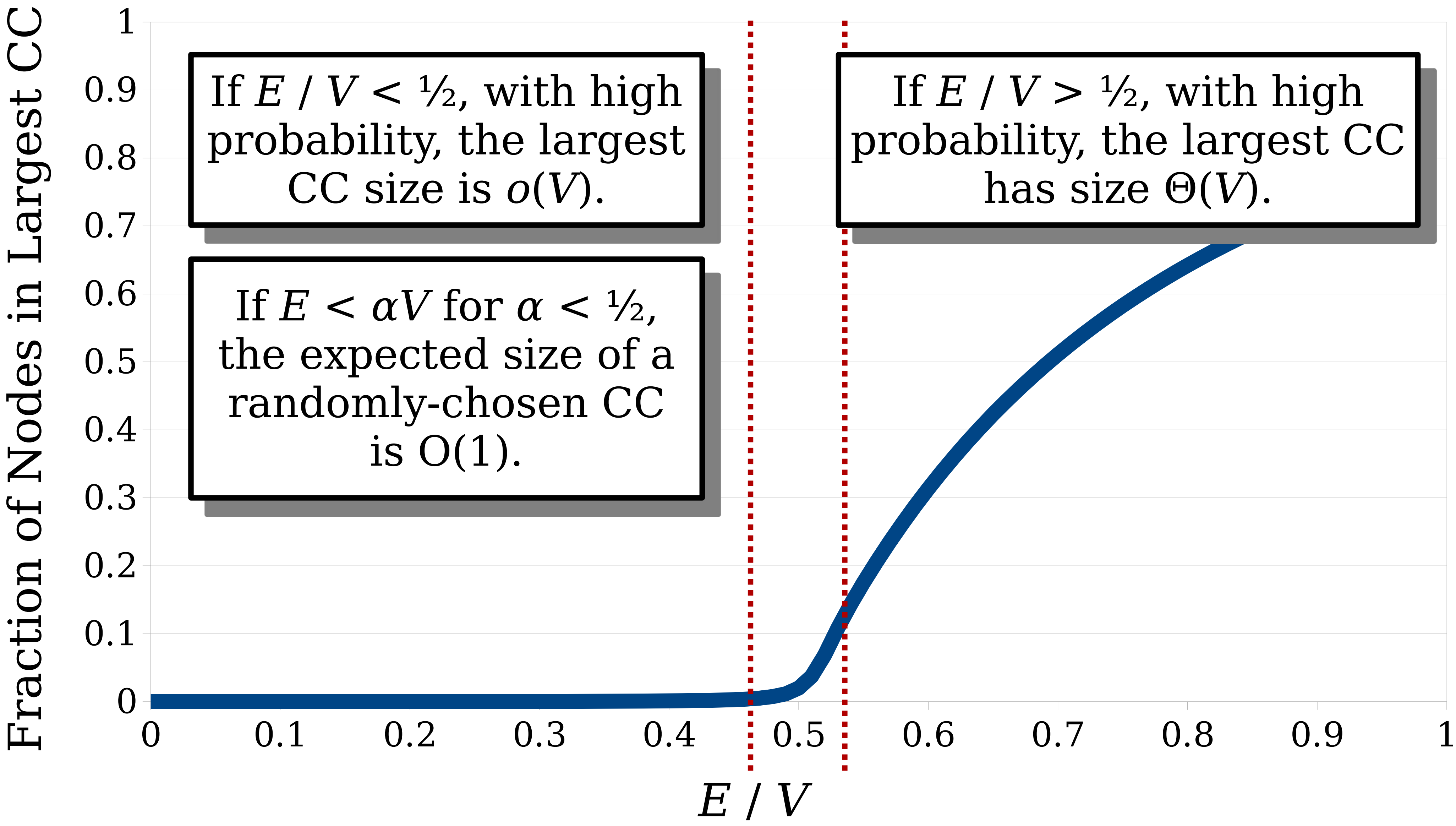
- Consider a graph with V nodes and no edges.
- Incrementally add E edges to the graph, each chosen uniformly at random, possibly with repetition.
- ***Question:*** What properties will this graph (probably) have?



Random Graph Evolution

- ***Claim:*** The phenomena we’re observing with cuckoo hashing are, in large part, due to properties of random graphs.
- ***Good News:*** This is a well-studied field! All the results we need were first proved by Erdős and Rényi in 1960.
- This model of incrementally constructing a graph is therefore called the ***Erdős–Rényi*** model.





If E / V < ½, with high probability, the largest CC size is $o(V)$.

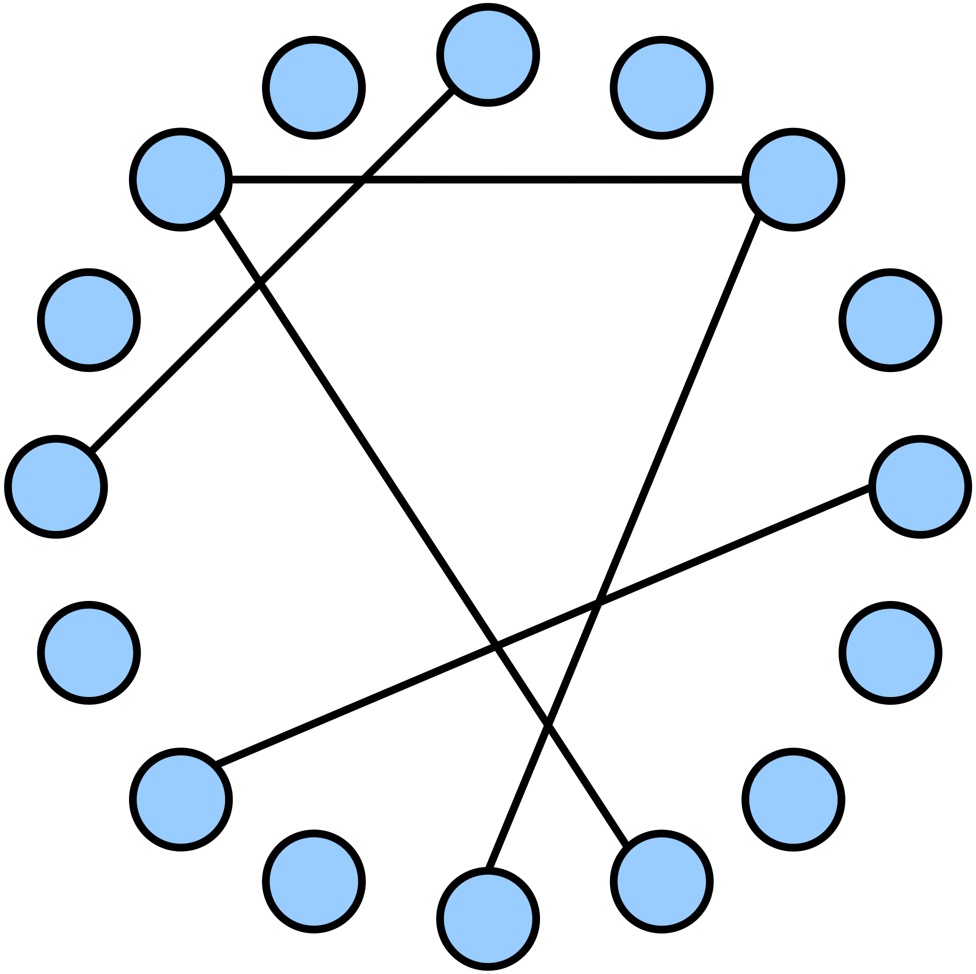
If $E < \alpha V$ for α < ½, the expected size of a randomly-chosen CC is O(1).

If E / V > ½, with high probability, the largest CC has size $\Theta(V)$.

Consider a random (multi)graph G with V nodes and E edges. What fraction of the nodes are in the largest connected component of G, as a function of E / V?

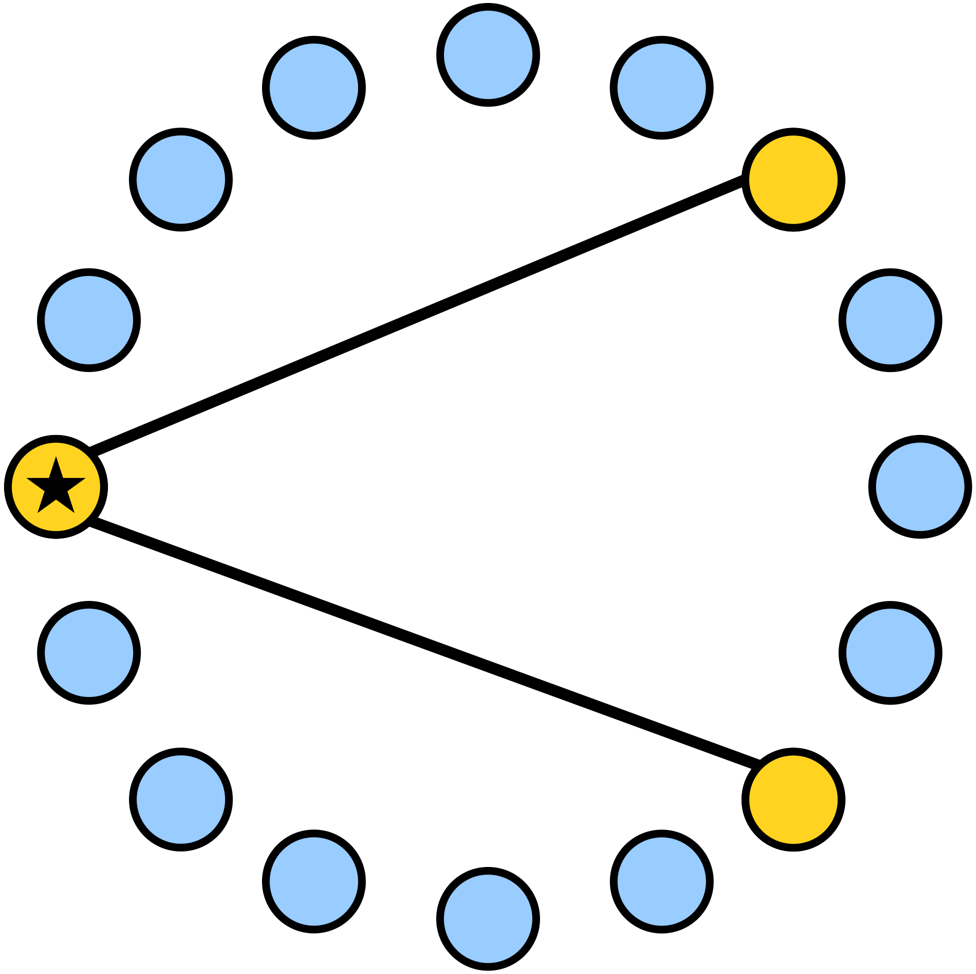
Sizing a Connected Component

- ***Goal:*** Show that if $E < \alpha V$ for some α < ½, then the expected size of a CC in a randomly-built graph is O(1).
- If we wanted to compute the size of a CC in an actual graph, we might run a BFS over its nodes and count how many we get back.
- ***Idea:*** Analyze, theoretically, what a BFS over a randomly-built graph of this form would look like.



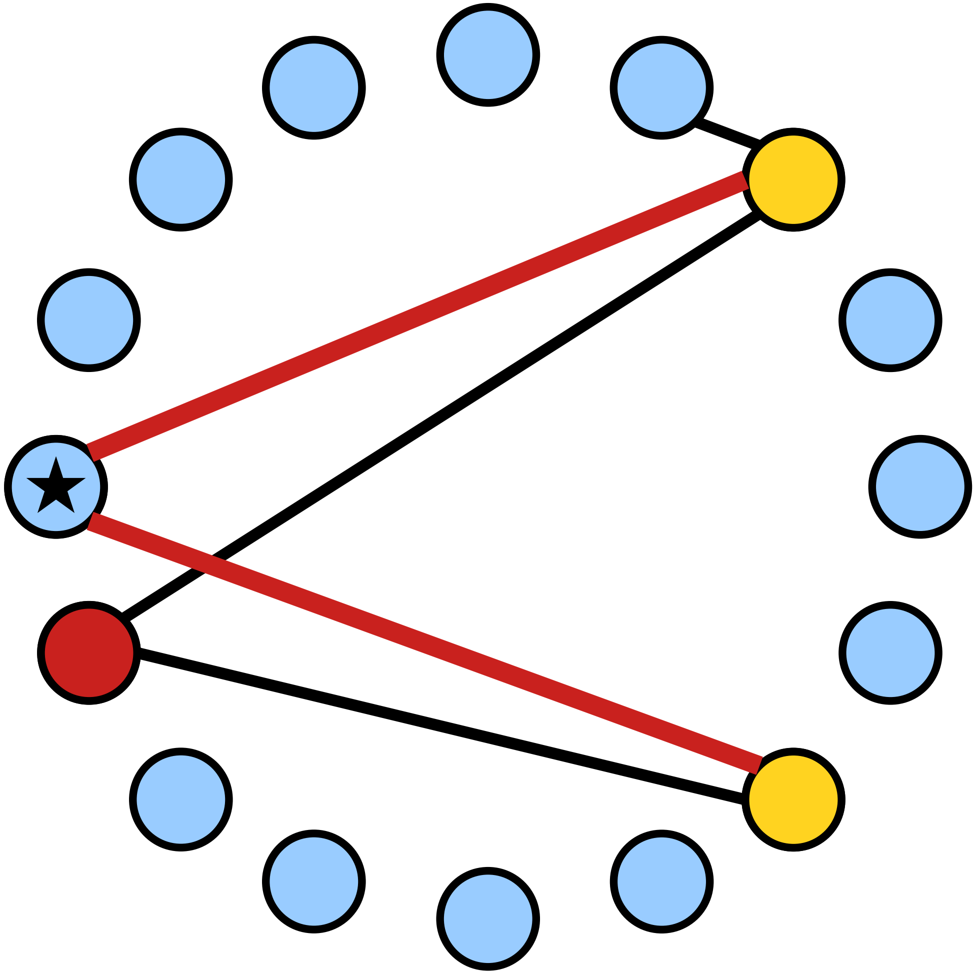
Sizing a Connected Component

- Pick a starting node for our BFS.
- Each of the E edges has a $^2/_V$ chance of touching this node.
- The number of nodes adjacent to our starting node, which will be visited in the next step of BFS, is a distributed as a Binom(E, $^2/_V$) variable.



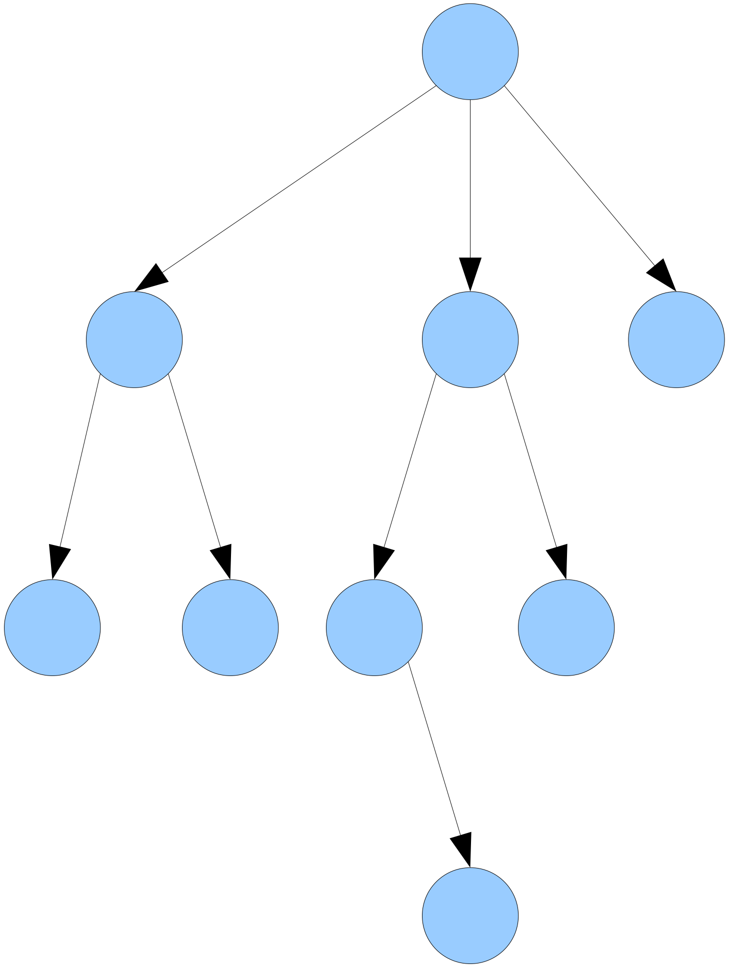
Sizing a Connected Component

- Each new node kinda sorta ish also touches a number of new nodes that can be modeled as a Binom(E, $^2/_V$) variable.
  - This ignores double-counting nodes.
  - This ignores existing edges.
  - This ignores correlations between edge counts.
- However, this conservatively bounds the number of new nodes visited in the next BFS step.



Modeling the BFS

- ***Idea:*** Count nodes in a connected component by simulating a BFS tree, where the number of children of each node is a Binom(E, $^2/_V$) variable.
  - Begin with a root node.
  - Each node has children distributed as a Binom(E, $^2/_V$) variable.
- ***Question:*** How many total nodes will this simulated BFS discover before terminating?

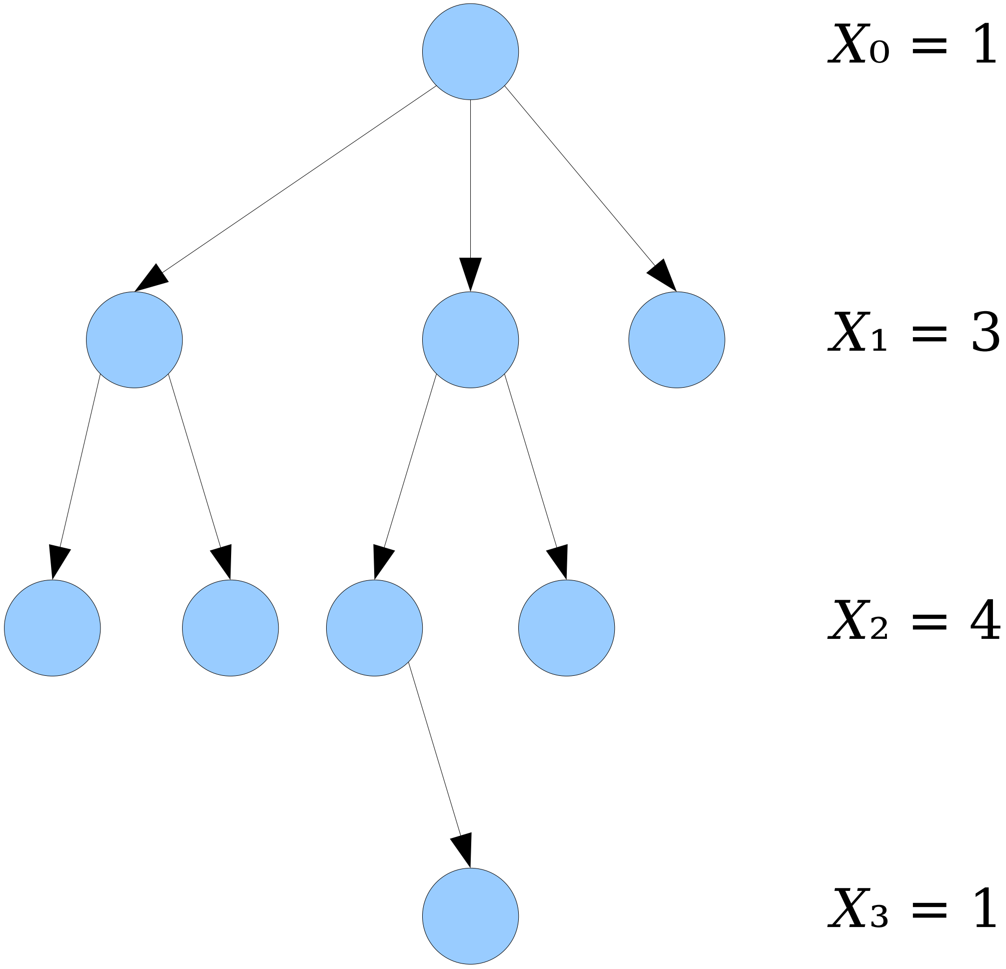


Modeling the BFS

- Denote by X_k the number of nodes at level k. This gives a series of random variables X_0, X_1, X_2, … .
- These variables are defined by the following randomized recurrence relation:

$$X_0 = 1 \qquad X_{k+1} = \sum_{i=1}^{X_k} \xi_{i,k}$$

- Here, each $\xi_{i,k}$ is an i.i.d. $\mathrm{Binom}(E, {}^2/_V)$ variable.



$X_0 = 1$

$X_1 = 3$

$X_2 = 4$

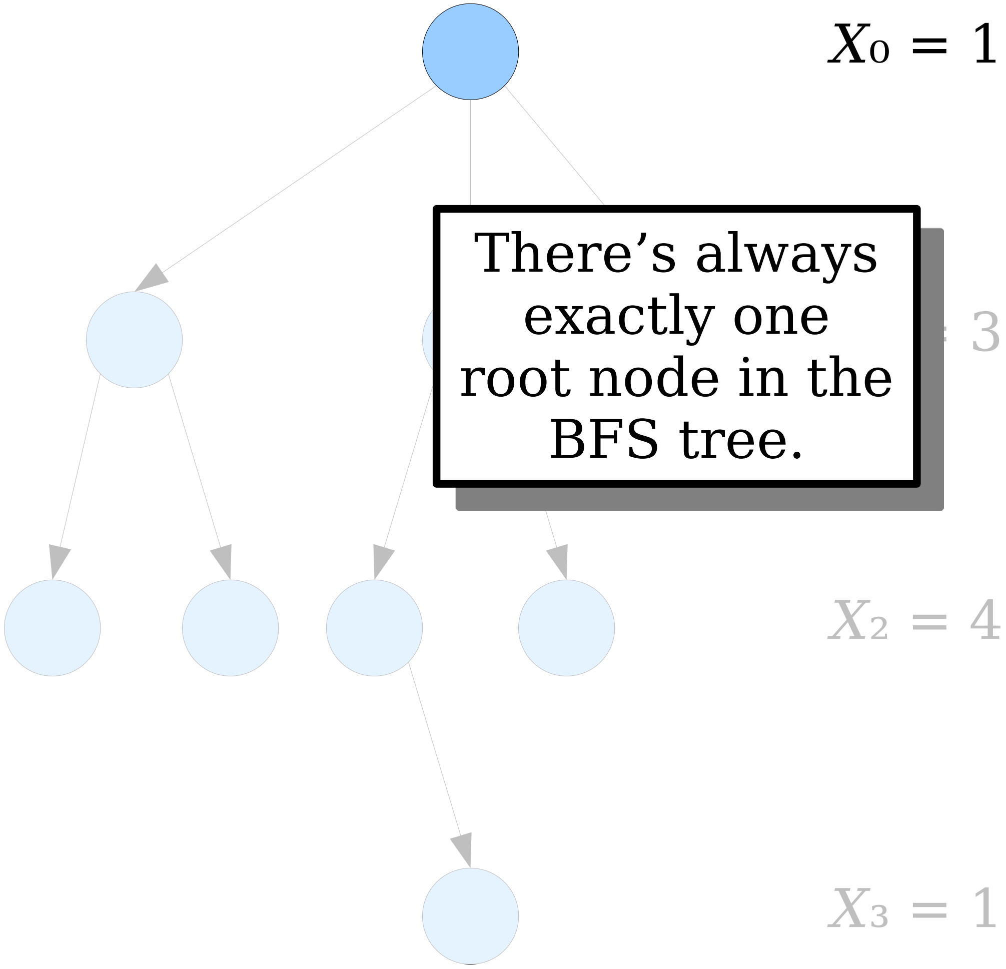
$X_3 = 1$

Modeling the BFS

- Denote by X_k the number of nodes at level k. This gives a series of random variables $X_0, X_1, X_2, \ldots$.
- These variables are defined by the following randomized recurrence relation:

$$X_0 = 1 \qquad X_{k+1} = \sum_{i=1}^{X_k} \xi_{i,k}$$

- Here, each $\xi_{i,k}$ is an i.i.d. Binom(E, $^2/_V$) variable.



There's always exactly one root node in the BFS tree.

Modeling the BFS

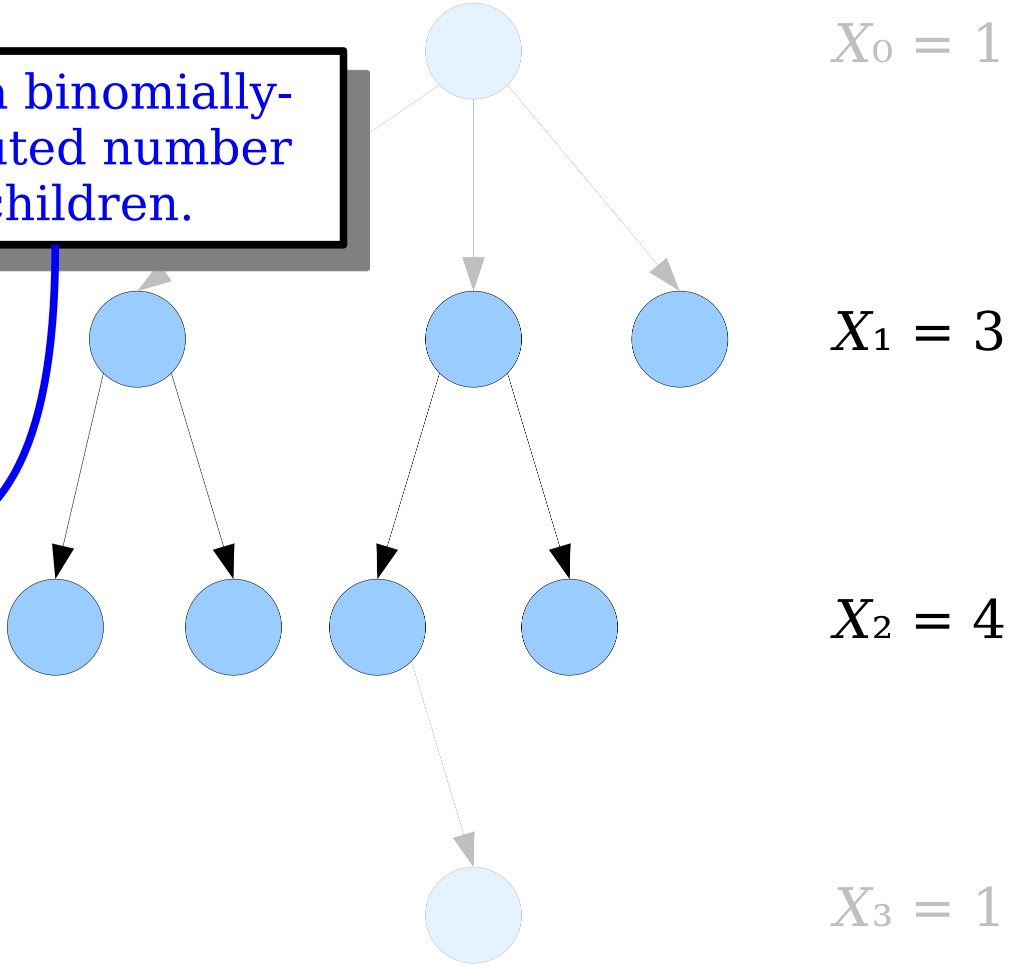
- Denote by X_k the number of nodes at level k. This gives a series of random va , X_2, .
- Th s are defined by the following randomized recurrence relation:

Each of the X_k nodes in layer k…

… has a binomially-distributed number of children.

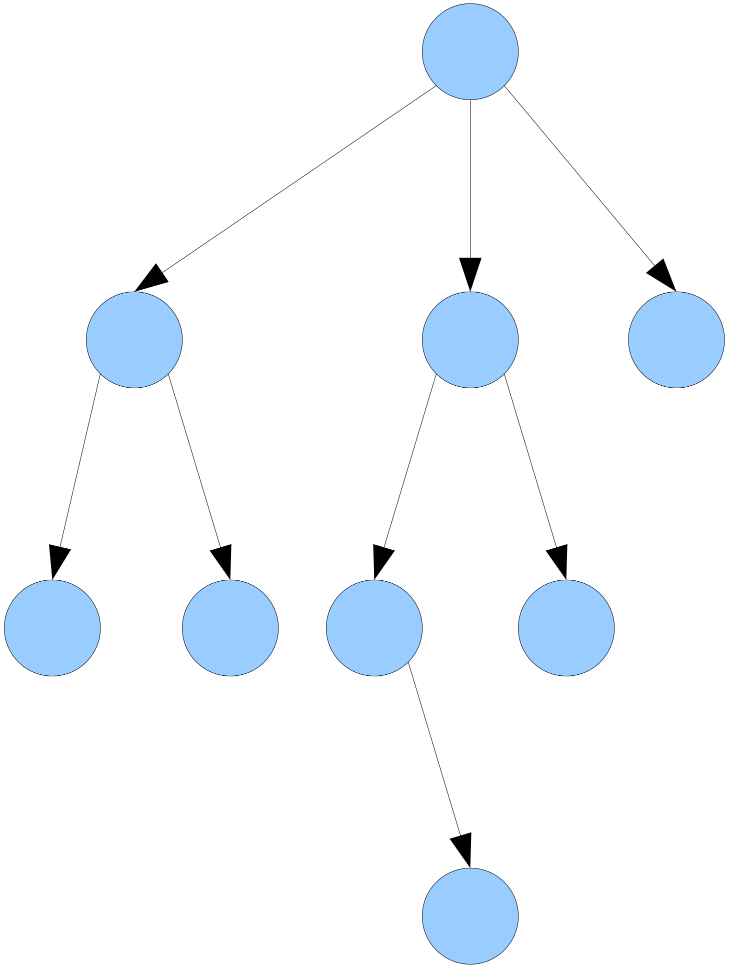
$$X_0 = 1 \qquad X_{k+1} = \sum_{i=1}^{X_k} \xi_{i,k}$$

- Here, each $\xi_{i,k}$ is an i.i.d. Binom(E, ${}^2/_V$) variable.



Modeling the BFS

- ***Observation:*** On expectation, each node has $^{2E}/_{V}$ children.
- The "expected branching factor" of the tree is $^{2E}/_{V}$, which is less than 1.
- How many nodes are there in the tree, assuming each layer has the expected number of nodes?



$X_0 = 1$

$X_1 = 3$

$X_2 = 4$

$X_3 = 1$

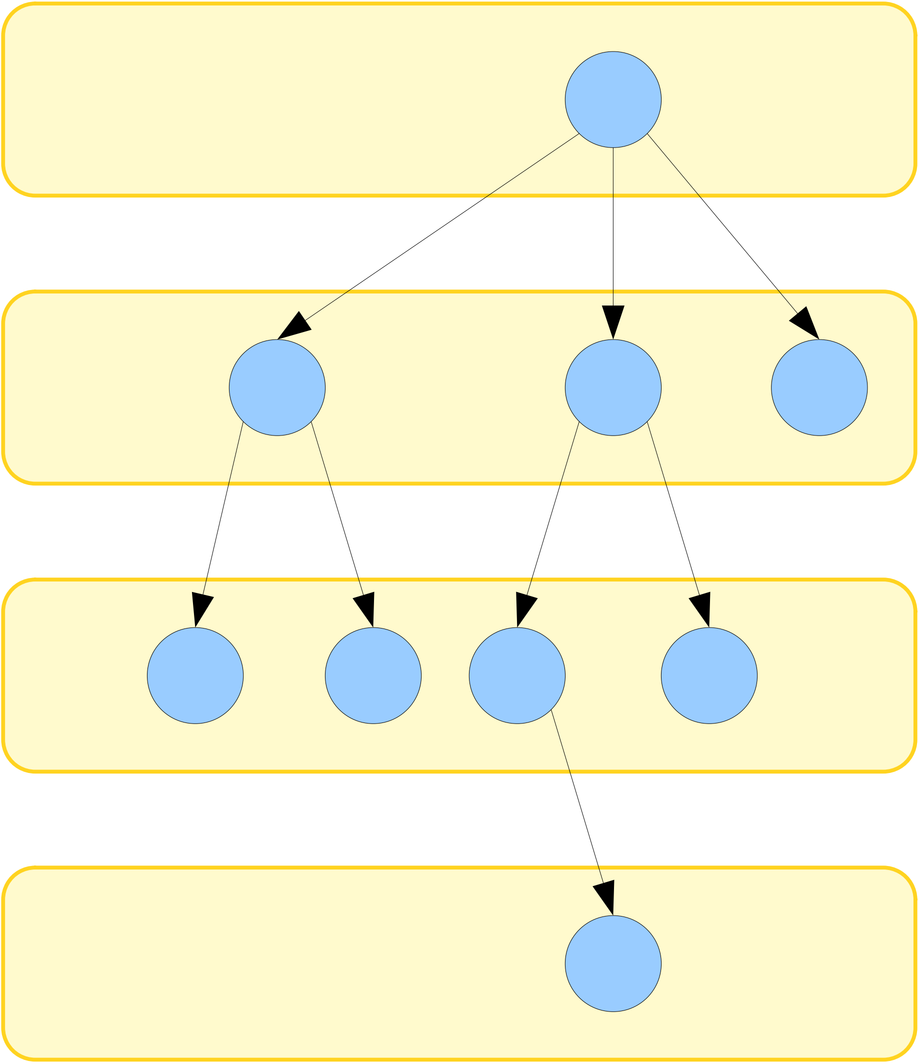
Modeling the BFS

There is always one node here.

On expectation, we’d find $^{2E}/_{V}$ nodes here.

On expectation, we’d find $(^{2E}/_{V})^2$ nodes here.

On expectation, we’d find $(^{2E}/_{V})^3$ nodes here.



$X_0 = 1$

$X_1 = 3$

$X_2 = 4$

$X_3 = 1$

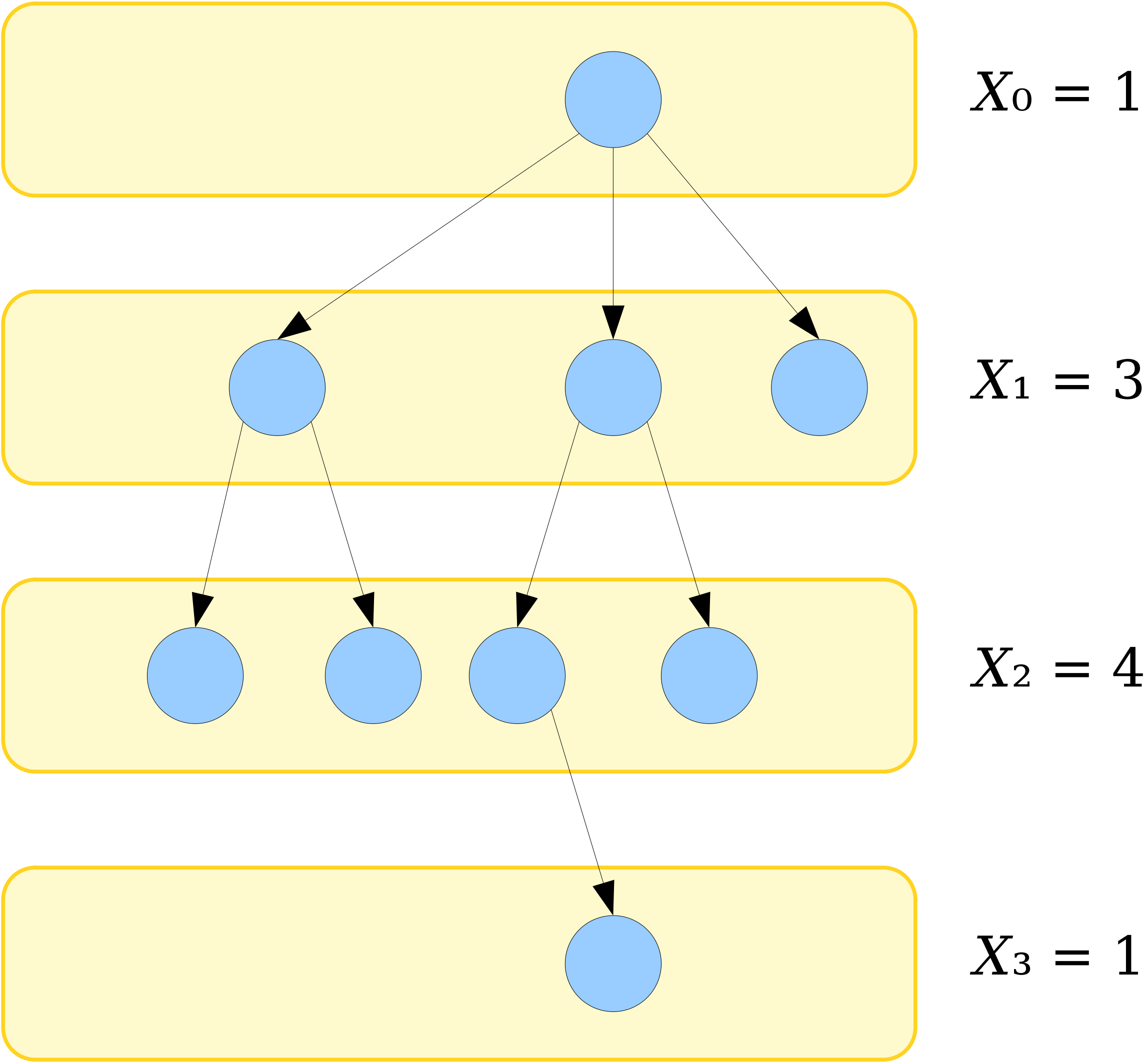
Modeling the BFS

Lemma: $\mathrm{E}[X_k] = ({}^{2E}/_{V})^k$.

Proof Idea: Show that

$$\mathrm{E}[X_{k+1}] = ({}^{2E}/_{V})\ \mathrm{E}[X_k]$$

and apply induction.



$$\mathrm{E}[X_{k+1}] = \mathrm{E}\left[\sum_{i=1}^{X_k} \xi_{i,k}\right]$$

$$= \sum_{j=0}^{\infty} \mathrm{E}\left[\sum_{i=1}^{X_k} \xi_{i,k} \;\middle|\; X_k = j\right] \cdot \Pr[X_k = j]$$

This is a sum of a random number of terms, so we can't use linearity of expectation.

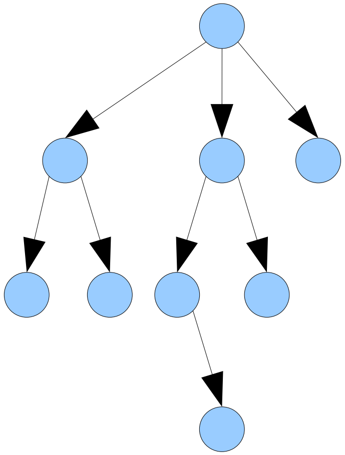
However, we can use the ***law of total expectation:***

$$\mathrm{E}[X] = \sum_j \mathrm{E}[X \mid Y = j] \cdot \Pr[Y = j]$$

$$X_0 = 1$$

$$X_{k+1} = \sum_{i=1}^{X_k} \xi_{i,k}$$

$$\xi_{i,k} \sim \mathrm{Binom}\left(E, \frac{2}{V}\right)$$



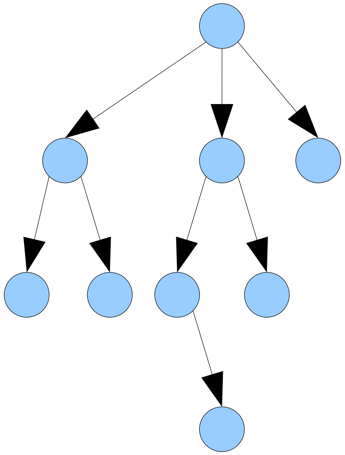
$$
\begin{aligned}
\mathrm{E}[X_{k+1}] &= \mathrm{E}\Big[\sum_{i=1}^{X_k} \xi_{i,k}\Big] \\
&= \sum_{j=0}^{\infty} \mathrm{E}\Big[\sum_{i=1}^{X_k} \xi_{i,k} \;\Big|\; X_k = j\Big] \cdot \Pr[X_k = j] \\
&= \sum_{j=0}^{\infty} \mathrm{E}\Big[\sum_{i=1}^{j} \xi_{i,k} \;\Big|\; X_k = j\Big] \cdot \Pr[X_k = j]
\end{aligned}
$$

Well, that makes things easier!

$$X_0 = 1$$

$$X_{k+1} = \sum_{i=1}^{X_k} \xi_{i,k}$$

$$\xi_{i,k} \sim \mathrm{Binom}\left(E, \frac{2}{V}\right)$$



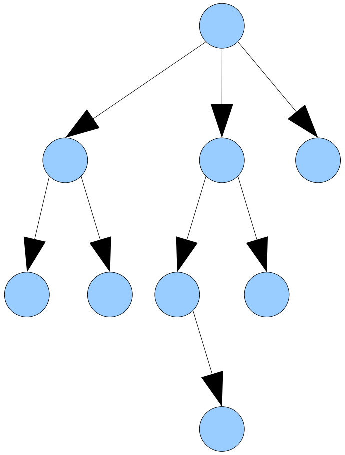
$$
\begin{aligned}
\mathrm{E}[X_{k+1}] &= \mathrm{E}[\sum_{i=1}^{X_k} \xi_{i,k}] \\
&= \sum_{j=0}^{\infty} \mathrm{E}[\sum_{i=1}^{X_k} \xi_{i,k} \mid X_k = j] \cdot \Pr[X_k = j] \\
&= \sum_{j=0}^{\infty} \mathrm{E}[\sum_{i=1}^{j} \xi_{i,k} \mid X_k = j] \cdot \Pr[X_k = j] \\
&= \sum_{j=0}^{\infty} \left(\sum_{i=1}^{j} \mathrm{E}[\xi_{i,k} \mid X_k = j] \right) \cdot \Pr[X_k = j]
\end{aligned}
$$

This sum ranges over a fixed number of terms, so we can apply linearity of (conditional) expectation.

$$X_0 = 1$$

$$X_{k+1} = \sum_{i=1}^{X_k} \xi_{i,k}$$

$$\xi_{i,k} \sim \mathrm{Binom}(E, \frac{2}{V})$$



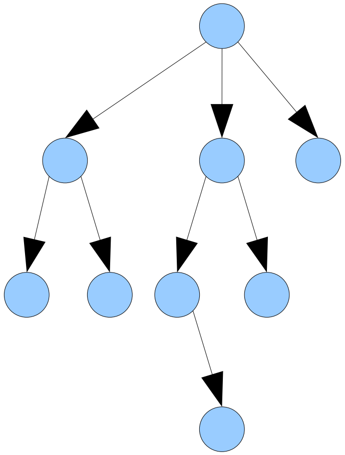
$$\begin{aligned} \mathrm{E}[X_{k+1}] &= \mathrm{E}[\sum_{i=1}^{X_k} \xi_{i,k}] \\ &= \sum_{j=0}^{\infty} \mathrm{E}[\sum_{i=1}^{X_k} \xi_{i,k} \mid X_k = j] \cdot \Pr[X_k = j] \\ &= \sum_{j=0}^{\infty} \mathrm{E}[\sum_{i=1}^{j} \xi_{i,k} \mid X_k = j] \cdot \Pr[X_k = j] \\ &= \sum_{j=0}^{\infty} \left(\sum_{i=1}^{j} \mathrm{E}[\xi_{i,k} \mid X_k = j] \right) \cdot \Pr[X_k = j] \\ &= \sum_{j=0}^{\infty} \left(\sum_{i=1}^{j} \mathrm{E}[\xi_{i,k}] \right) \cdot \Pr[X_k = j] \end{aligned}$$

These random variables are independent – one represents the number of nodes in a particular layer. One represents the number of children that a specific node might have.

$$X_0 = 1$$

$$X_{k+1} = \sum_{i=1}^{X_k} \xi_{i,k}$$

$$\xi_{i,k} \sim \mathrm{Binom}\left(E, \frac{2}{V}\right)$$

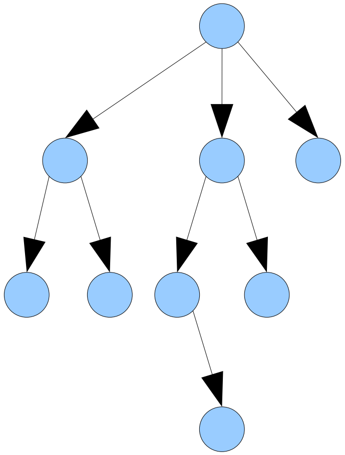


$$
\begin{aligned}
\mathrm{E}[X_{k+1}] &= \mathrm{E}\Big[\sum_{i=1}^{X_k} \xi_{i,k}\Big] \\
&= \sum_{j=0}^{\infty} \mathrm{E}\Big[\sum_{i=1}^{X_k} \xi_{i,k} \;\Big|\; X_k = j\Big] \cdot \Pr[X_k = j] \\
&= \sum_{j=0}^{\infty} \mathrm{E}\Big[\sum_{i=1}^{j} \xi_{i,k} \;\Big|\; X_k = j\Big] \cdot \Pr[X_k = j] \\
&= \sum_{j=0}^{\infty} \Big(\sum_{i=1}^{j} \mathrm{E}[\xi_{i,k} \mid X_k = j]\Big) \cdot \Pr[X_k = j] \\
&= \sum_{j=0}^{\infty} \Big(\sum_{i=1}^{j} \mathrm{E}[\xi_{i,k}]\Big) \cdot \Pr[X_k = j] \\
&= \sum_{j=0}^{\infty} \Big(\sum_{i=1}^{j} \frac{2E}{V}\Big) \cdot \Pr[X_k = j]
\end{aligned}
$$

$$X_0 = 1$$

$$X_{k+1} = \sum_{i=1}^{X_k} \xi_{i,k}$$

$$\xi_{i,k} \sim \mathrm{Binom}\Big(E, \frac{2}{V}\Big)$$

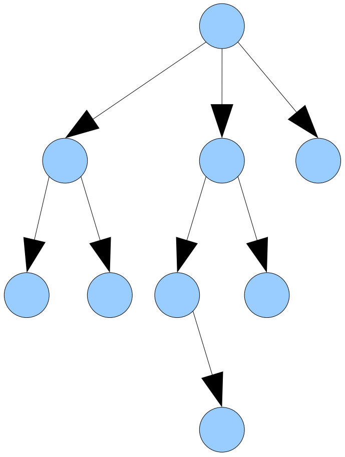


$$
\begin{aligned}
\mathrm{E}[X_{k+1}] &= \mathrm{E}\Big[\sum_{i=1}^{X_k} \xi_{i,k}\Big] \\
&= \sum_{j=0}^{\infty} \mathrm{E}\Big[\sum_{i=1}^{X_k} \xi_{i,k} \;\Big|\; X_k = j\Big] \cdot \Pr[X_k = j] \\
&= \sum_{j=0}^{\infty} \mathrm{E}\Big[\sum_{i=1}^{j} \xi_{i,k} \;\Big|\; X_k = j\Big] \cdot \Pr[X_k = j] \\
&= \sum_{j=0}^{\infty} \Big(\sum_{i=1}^{j} \mathrm{E}[\xi_{i,k} \mid X_k = j]\Big) \cdot \Pr[X_k = j] \\
&= \sum_{j=0}^{\infty} \Big(\sum_{i=1}^{j} \mathrm{E}[\xi_{i,k}]\Big) \cdot \Pr[X_k = j] \\
&= \sum_{j=0}^{\infty} \Big(\sum_{i=1}^{j} \frac{2E}{V}\Big) \cdot \Pr[X_k = j] \\
&= \frac{2E}{V} \cdot \sum_{j=0}^{\infty} \big(j \cdot \Pr[X_k = j]\big)
\end{aligned}
$$

$$X_0 = 1$$

$$X_{k+1} = \sum_{i=1}^{X_k} \xi_{i,k}$$

$$\xi_{i,k} \sim \mathrm{Binom}\Big(E, \frac{2}{V}\Big)$$

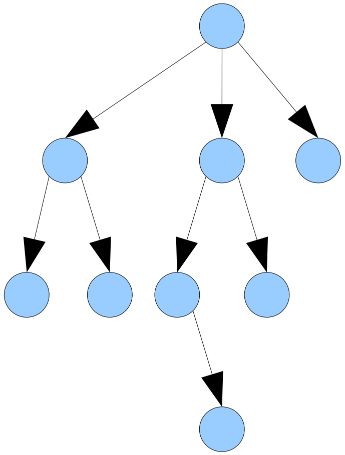


$$
\begin{aligned}
\mathrm{E}[X_{k+1}] &= \mathrm{E}\left[\sum_{i=1}^{X_k} \xi_{i,k}\right] \\
&= \sum_{j=0}^{\infty} \mathrm{E}\left[\sum_{i=1}^{X_k} \xi_{i,k} \;\middle|\; X_k = j\right] \cdot \mathrm{Pr}[X_k = j] \\
&= \sum_{j=0}^{\infty} \mathrm{E}\left[\sum_{i=1}^{j} \xi_{i,k} \;\middle|\; X_k = j\right] \cdot \mathrm{Pr}[X_k = j] \\
&= \sum_{j=0}^{\infty} \left(\sum_{i=1}^{j} \mathrm{E}[\xi_{i,k} \mid X_k = j] \right) \cdot \mathrm{Pr}[X_k = j] \\
&= \sum_{j=0}^{\infty} \left(\sum_{i=1}^{j} \mathrm{E}[\xi_{i,k}] \right) \cdot \mathrm{Pr}[X_k = j] \\
&= \sum_{j=0}^{\infty} \left(\sum_{i=1}^{j} \frac{2E}{V} \right) \cdot \mathrm{Pr}[X_k = j] \\
&= \frac{2E}{V} \cdot \sum_{j=0}^{\infty} \left(j \cdot \mathrm{Pr}[X_k = j] \right) \\
&= \frac{2E}{V} \cdot \mathrm{E}[X_k]
\end{aligned}
$$

$$X_0 = 1$$

$$X_{k+1} = \sum_{i=1}^{X_k} \xi_{i,k}$$

$$\xi_{i,k} \sim \mathrm{Binom}\left(E, \frac{2}{V}\right)$$

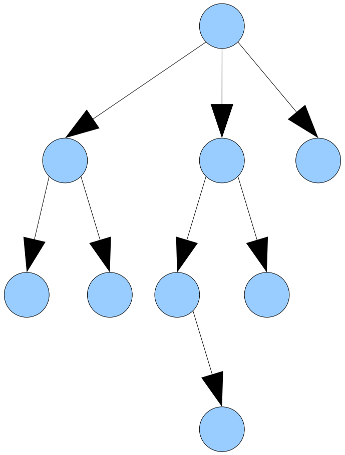


$$\begin{aligned}
\mathrm{E}[X_{k+1}] &= \mathrm{E}\Big[\sum_{i=1}^{X_k} \xi_{i,k}\Big] \\
&= \sum_{j=0}^{\infty} \mathrm{E}\Big[\sum_{i=1}^{X_k} \xi_{i,k} \;\Big|\; X_k = j\Big] \cdot \Pr[X_k = j] \\
&= \sum_{j=0}^{\infty} \mathrm{E}\Big[\sum_{i=1}^{j} \xi_{i,k} \;\Big|\; X_k = j\Big] \cdot \Pr[X_k = j] \\
&= \sum_{j=0}^{\infty} \left(\sum_{i=1}^{j} \mathrm{E}[\xi_{i,k} \;|\; X_k = j]\right) \cdot \Pr[X_k = j] \\
&= \sum_{j=0}^{\infty} \left(\sum_{i=1}^{j} \mathrm{E}[\xi_{i,k}]\right) \cdot \Pr[X_k = j] \\
&= \sum_{j=0}^{\infty} \left(\sum_{i=1}^{j} \frac{2E}{V}\right) \cdot \Pr[X_k = j] \\
&= \frac{2E}{V} \cdot \sum_{j=0}^{\infty} \big(j \cdot \Pr[X_k = j]\big) \\
&= \frac{2E}{V} \cdot \mathrm{E}[X_k]
\end{aligned}$$

$$X_0 = 1$$

$$X_{k+1} = \sum_{i=1}^{X_k} \xi_{i,k}$$

$$\xi_{i,k} \sim \mathrm{Binom}\left(E, \frac{2}{V}\right)$$



The Finishing Touches

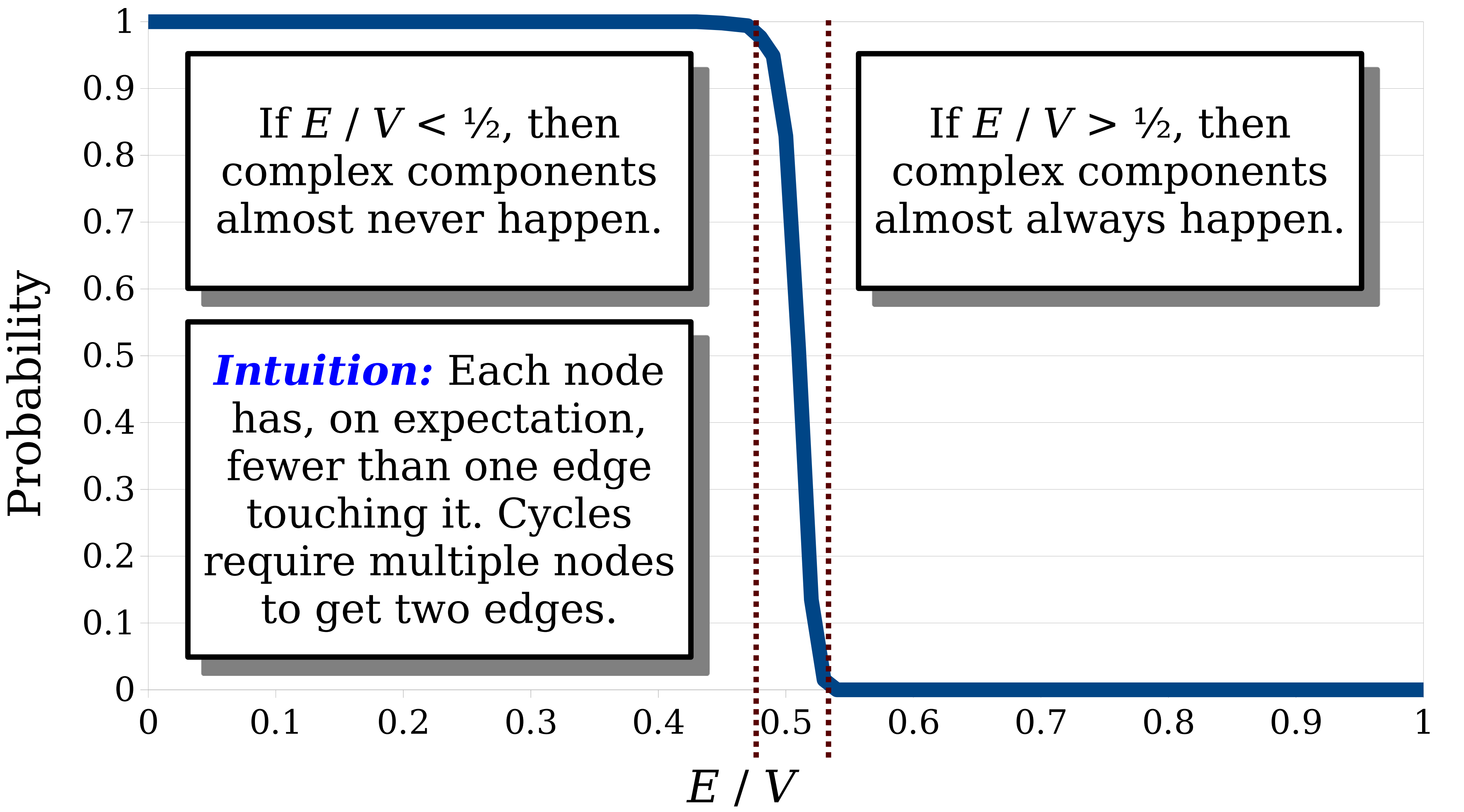
- On expectation, there are $(^{2E}/_{V})^k$ nodes in layer k of the BFS tree.
- Summing across all layers, on expectation there are $(1 – {}^{2E}/_{V})^{-1}$ total nodes in the BFS tree.
- Assuming $E = \alpha V$, for a fixed constant $\alpha < ½$, this is O(1) nodes per CC.
- Therefore, in cuckoo hashing, assuming we set $n = \alpha m$ for some $\alpha < ½$, each insertion touches a CC with expected size O(1), so each insertion does only expected O(1) displacements.
- This explains why the total number of displacements was such a strongly linear plot!

How big are the connected components in the cuckoo graph?

(This tells us how much work we do on a successful insertion.)

What is the probability that a connected component in the cuckoo graph is complex?

(This lets us see how much time we should expect to spend rehashing.)



If *E* / *V* < ½, then complex components almost never happen.

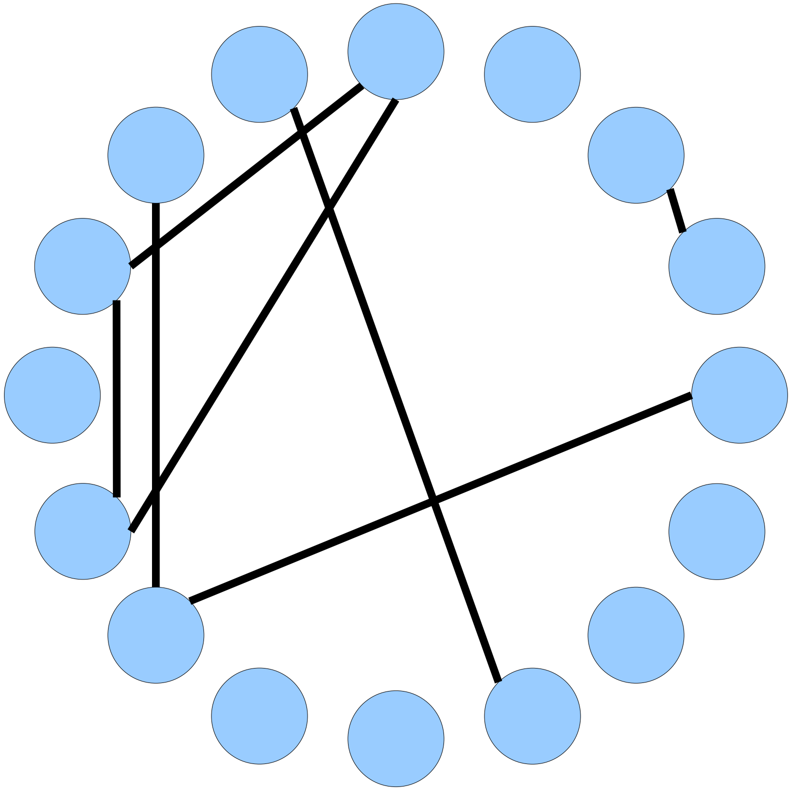
If *E* / *V* > ½, then complex components almost always happen.

Intuition: Each node has, on expectation, fewer than one edge touching it. Cycles require multiple nodes to get two edges.

Consider a random (multi)graph *G* with *V* nodes and *E* edges. What is the the probability that every connected component in *G* is simple, as a function of the ratio *E* / *V*?

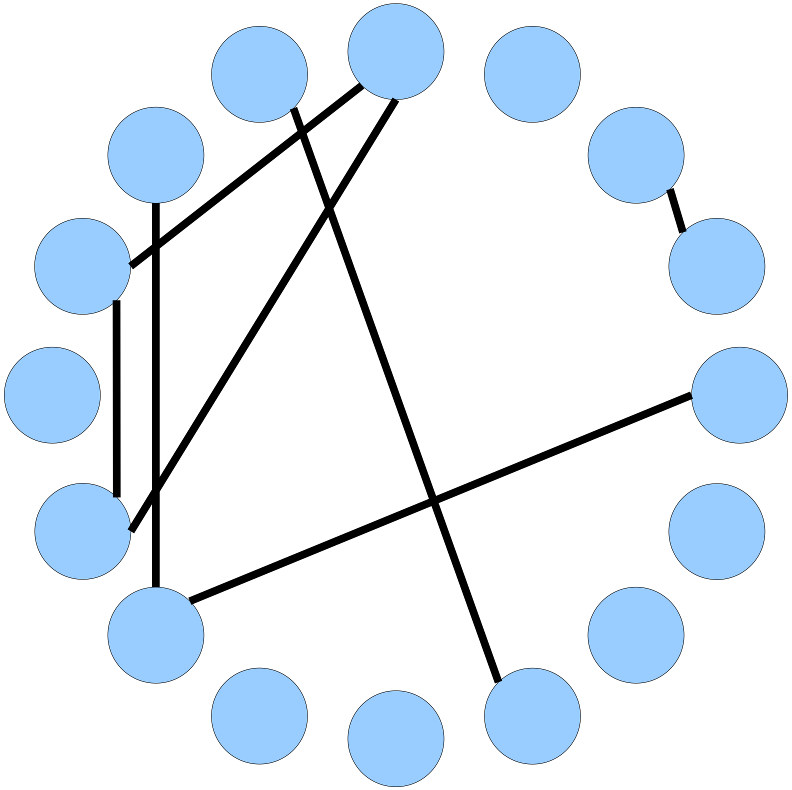
Random Graph Theory

- ***Theorem:*** Let $E = \alpha V$ for some $\alpha < ½$. Then the probability that any connected component is complex is $O(^1/_V)$.
- ***Corollary:*** Using cuckoo hashing with m slots and $n = \alpha m$ items, the probability that a series of n insertions fails is $O(^1/_n)$, and the expected number of times a rehash is required before it succeeds is $O(1)$.



Random Graph Theory

- Every proof I’ve seen of this result boils down to a (messy) counting argument of enumerating possible complex CC shapes and evaluating their probabilities.
- ***(Possibly?) Open Problem:*** Find a short, simple proof of the result about complex CCs.



The Overall Analysis

- Cuckoo hashing gives worst-case lookups and deletions.
- Insertions are expected O(1).
  - This assumes you periodically double the size of the table and rehash when things get too full.
- The hidden constants are small, and this is a practical technique for building hash tables.

Cuckoo Hashing:

- ***lookup***: O(1)
- ***insert***: O(1)*
- ***delete***: O(1)

* *expected*

Improving Our Space Usage

Improving Space Usage

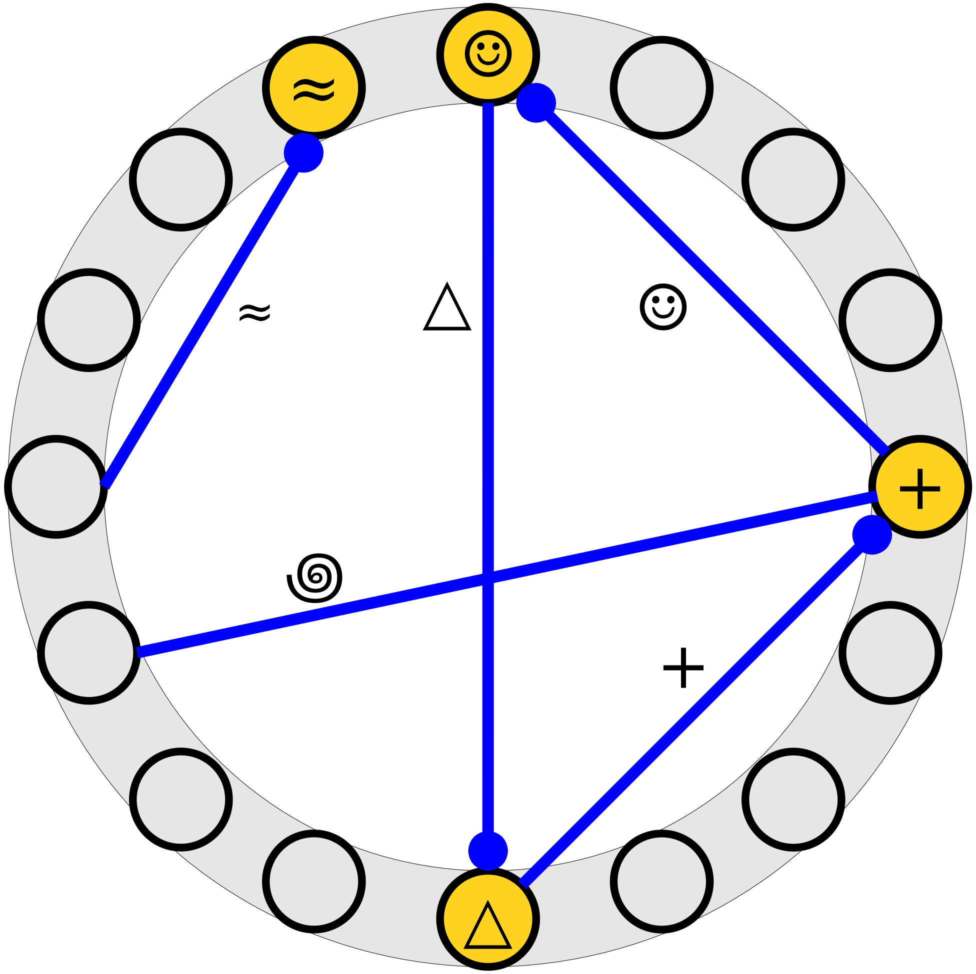
- A cuckoo hash table with n elements requires a table of size n / α, with α < ½.
- This means at least 50% of the table slots will be empty.
- The root cause is a fundamental property of random graphs; exceeding this threshold makes failure almost certain.
- ***Question:*** How can we push past this to improve cuckoo hashing space usage?

Improving Space Usage

- Our cuckoo graph – and the associated limitations on cuckoo hashing – result from these two assumptions:
  - Each table slot can hold at most one item.
  - Each item can be placed into one of two positions.
- We need to relax these constraints.

Which of these constraints could we relax, and what might it look like if we did?

Formulate a hypothesis!

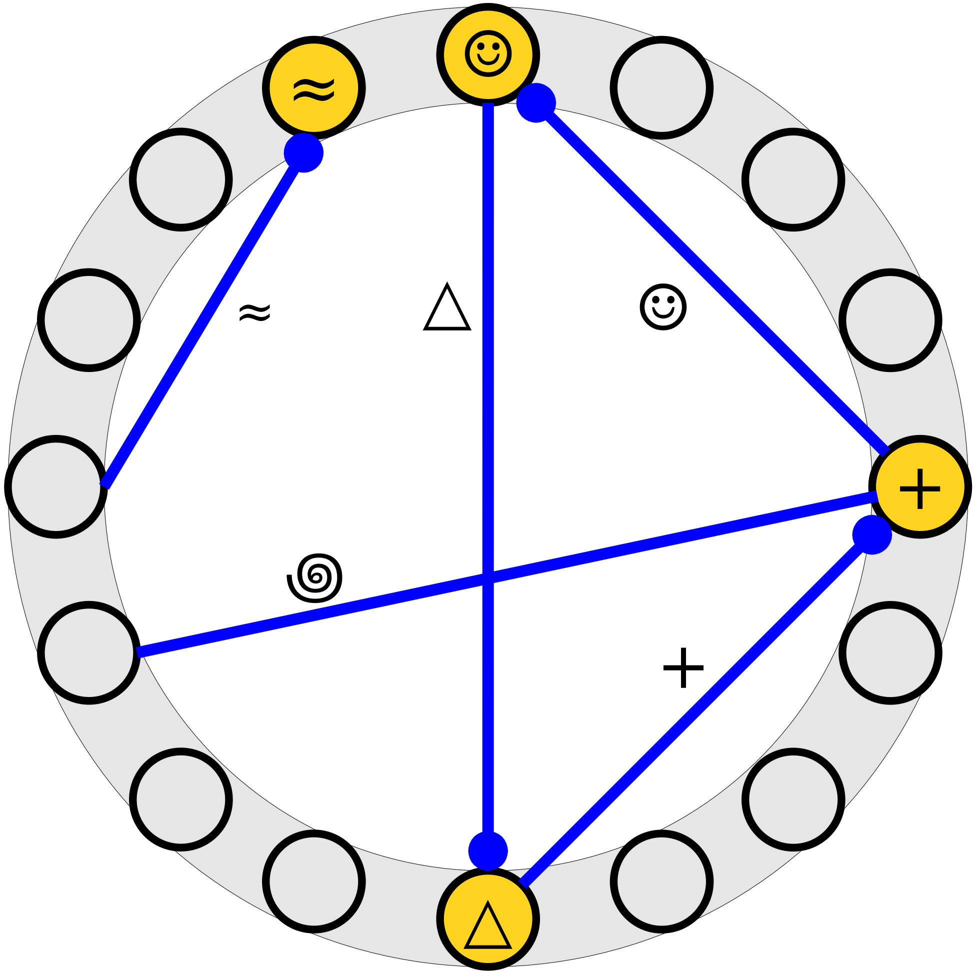


Improving Space Usage

- Our cuckoo graph – and the associated limitations on cuckoo hashing – result from these two assumptions:
  - Each table slot can hold at most one item.
  - Each item can be placed into one of two positions.
- We need to relax these constraints.

Which of these constraints could we relax, and what might it look like if we did?

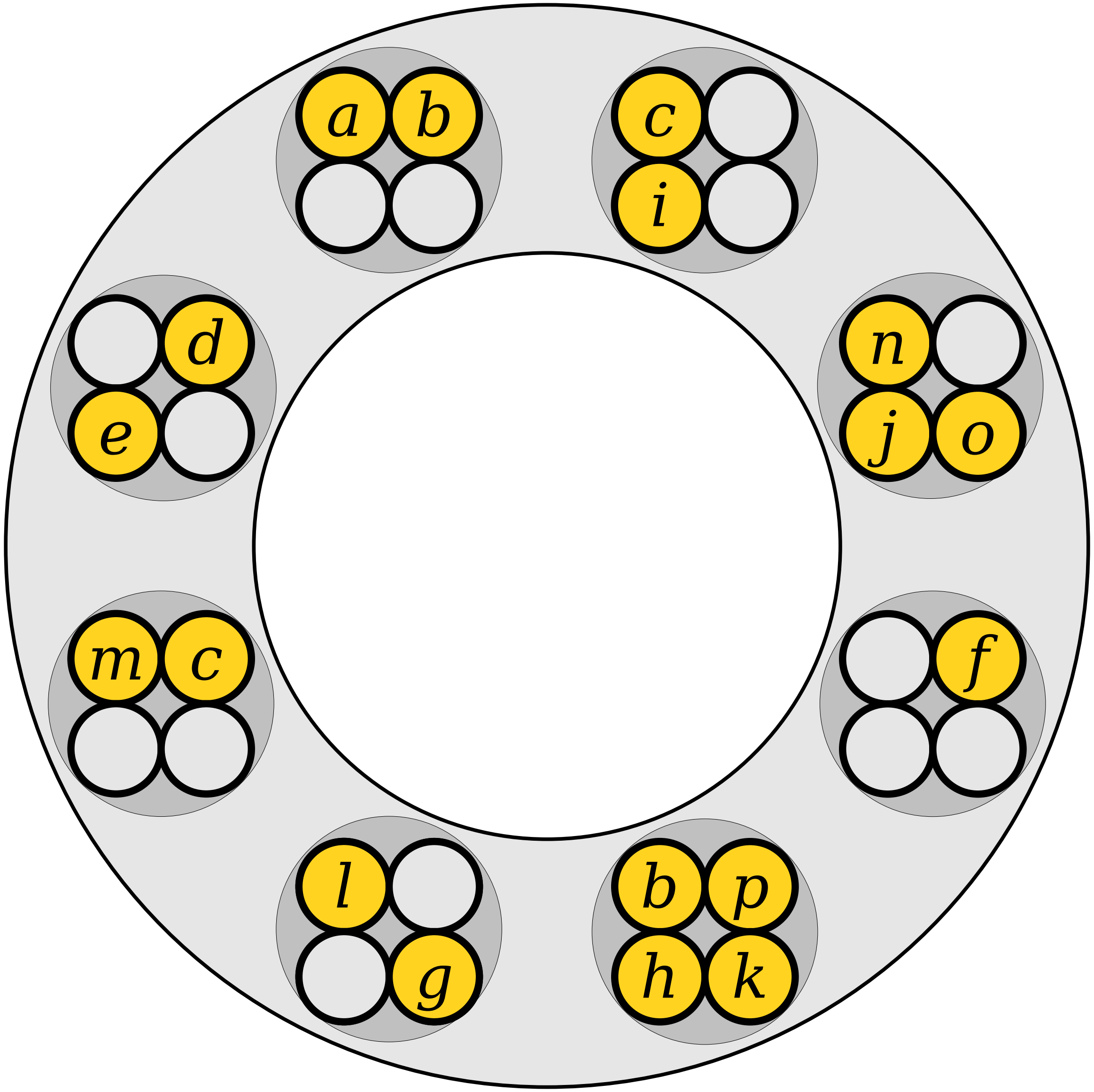
Discuss with your neighbors!

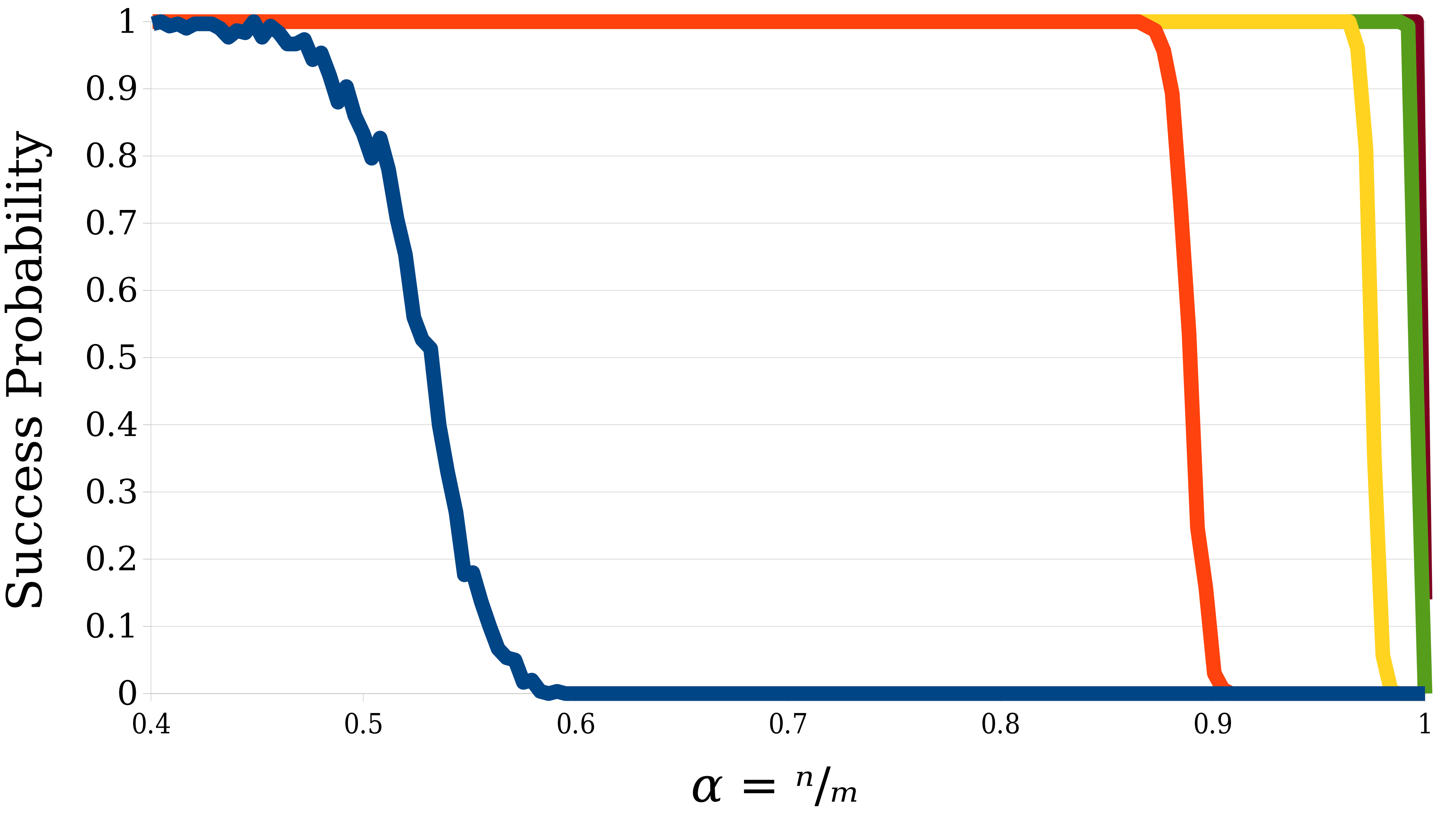


Idea 1: Allow each table slot
to store multiple items.

Blocked Cuckoo Hashing

- In ***blocked cuckoo hashing***, each slot can hold $b \geq 1$ items.
- When inserting an item, place it in one of the two slots it hashes to if there's free space in either.
- If there's no room left, displace a randomly-chosen other element in the slot.
- Increasing b decreases the likelihood that insertions fail, but increases the cost of lookups and deletions.
- b is often chosen so each slot fits cleanly in a cache line, improving performance.





🟦 $b = 1$
🟧 $b = 2$
🟨 $b = 4$
🟩 $b = 8$
🟥 $b = 16$

Suppose we insert $n = \alpha m$ elements into a cuckoo hash table with ${}^{m}/_{b}$ slots, each of which can hold b elements. What is the probability that all insertions succeed?

Blocked Cuckoo Hashing

- Suppose we have a table with m/b slots, each of which hold b items. Assume $n = m\alpha$ and we use two hash functions.
- The thresholds given below show the maximum value of α a blocked cuckoo hash table can use.
- As you can see, modest increases to b dramatically increase the space utilization of the table!

	$b = 1$	$b = 2$	$b = 3$	$b = 4$	$b = 5$
Theoretical max α	**0.500**	**0.897**	**0.959**	**0.980**	**0.989**

Blocked Cuckoo Hashing

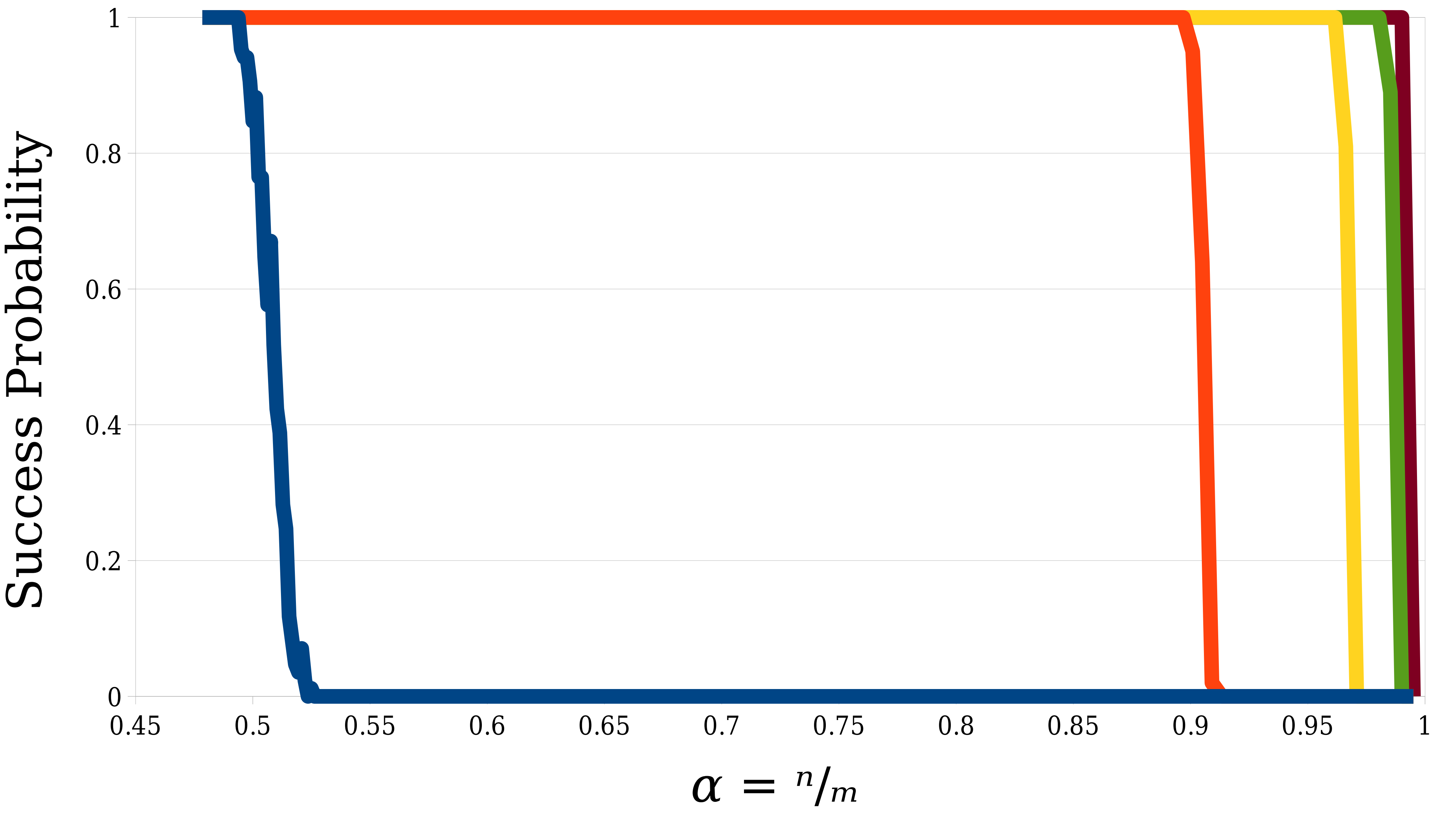
- These bounds are derived from the ***k-orientability threshold*** for random graphs.
- A ***k-orientation*** of a graph is a way of placing dots on one endpoint of each edge such that each node has at most k dots touching it.
  - In cuckoo hashing, this means that each node stores at most k items.
- Deriving these bounds requires a deep dive into random graph theory that's above the CS166 pay grade, but which could make for a fascinating final project topic!

	$b = 1$	$b = 2$	$b = 3$	$b = 4$	$b = 5$
Theoretical max α	**0.500**	**0.897**	**0.959**	**0.980**	**0.989**

Idea 2: Use multiple hash functions.

d-ary Cuckoo Hashing

- In *d*-ary cuckoo hashing, we pick an integer $d \geq 2$ and choose *d* different hash functions.
- Each item can be stored in one up to *d* slots, with choices given by the hash functions.
  - You could do extra work to ensure there are *d* separate locations, or be okay with duplicates if the hashes collide.
- To check if an item is in the table, hash it *d* times and see if it's in any of those slots.
- To insert an item, hash it *d* times and place the item in a free slot. If none exists, evict a randomly-chosen item from a slot, place the new item there, and repeat.

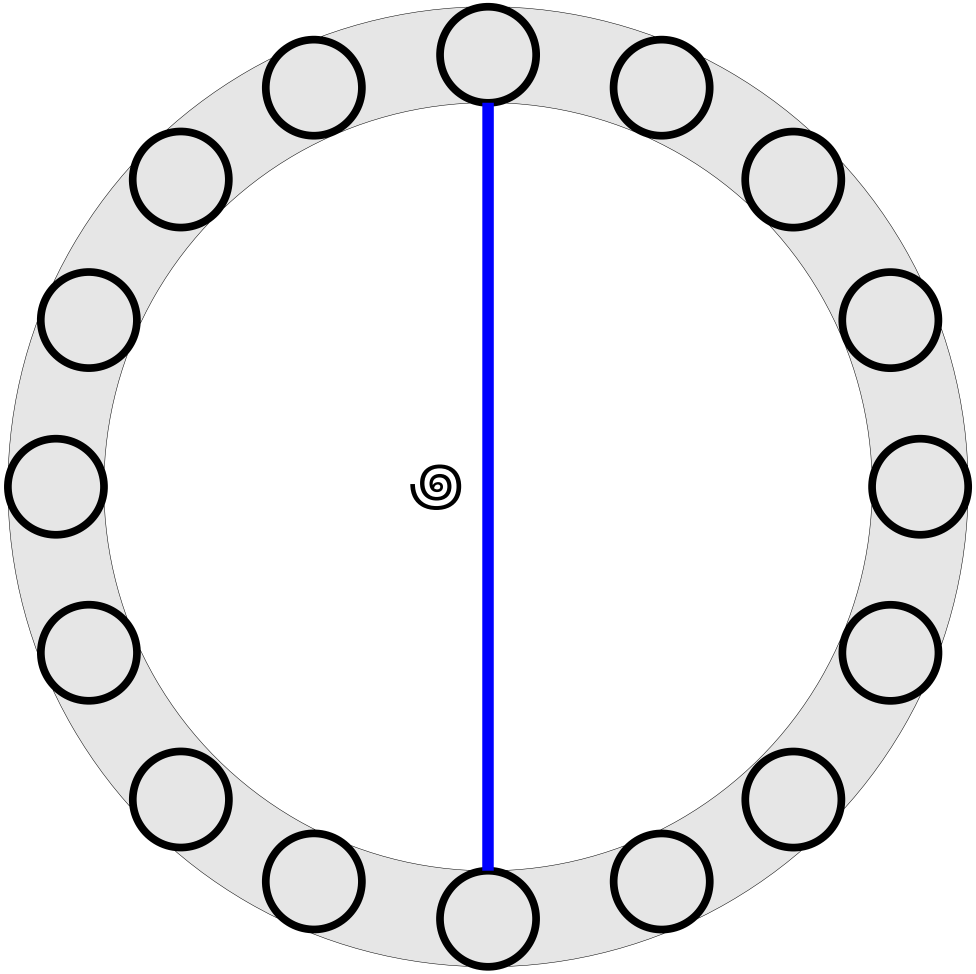


🟦 $d = 2$
🟧 $d = 3$
🟨 $d = 4$
🟩 $d = 5$
🟥 $d = 6$

Suppose we insert $n = \alpha m$ elements into a hash table with m slots. What is the probability that all insertions succeed?

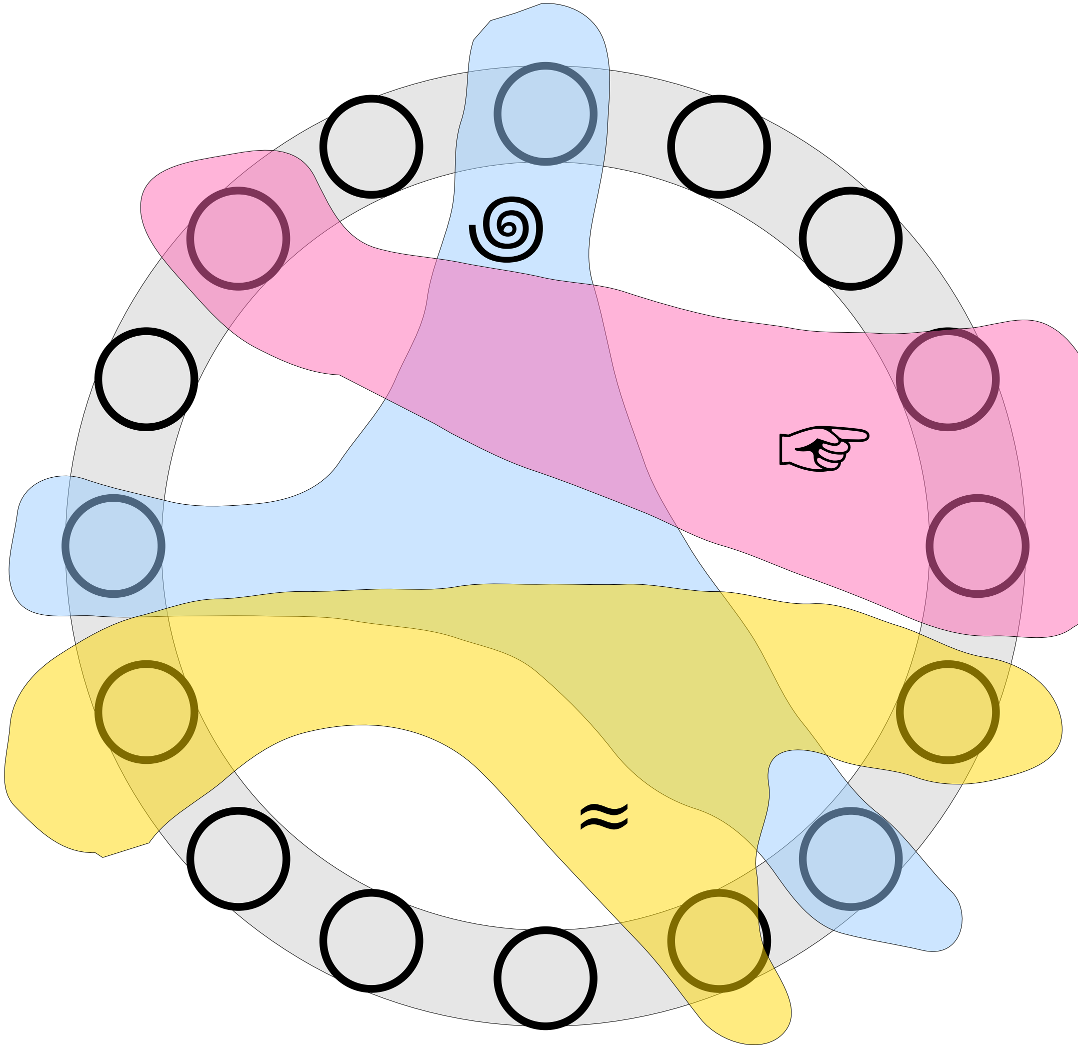
d-ary Cuckoo Hashing

- With $d = 2$, each item in the cuckoo hash table can be in one of two locations.
- We can model this as a graph: nodes are slots and edges are items.
- With $d > 2$, each item can be in more than two locations.
- How do we model this?



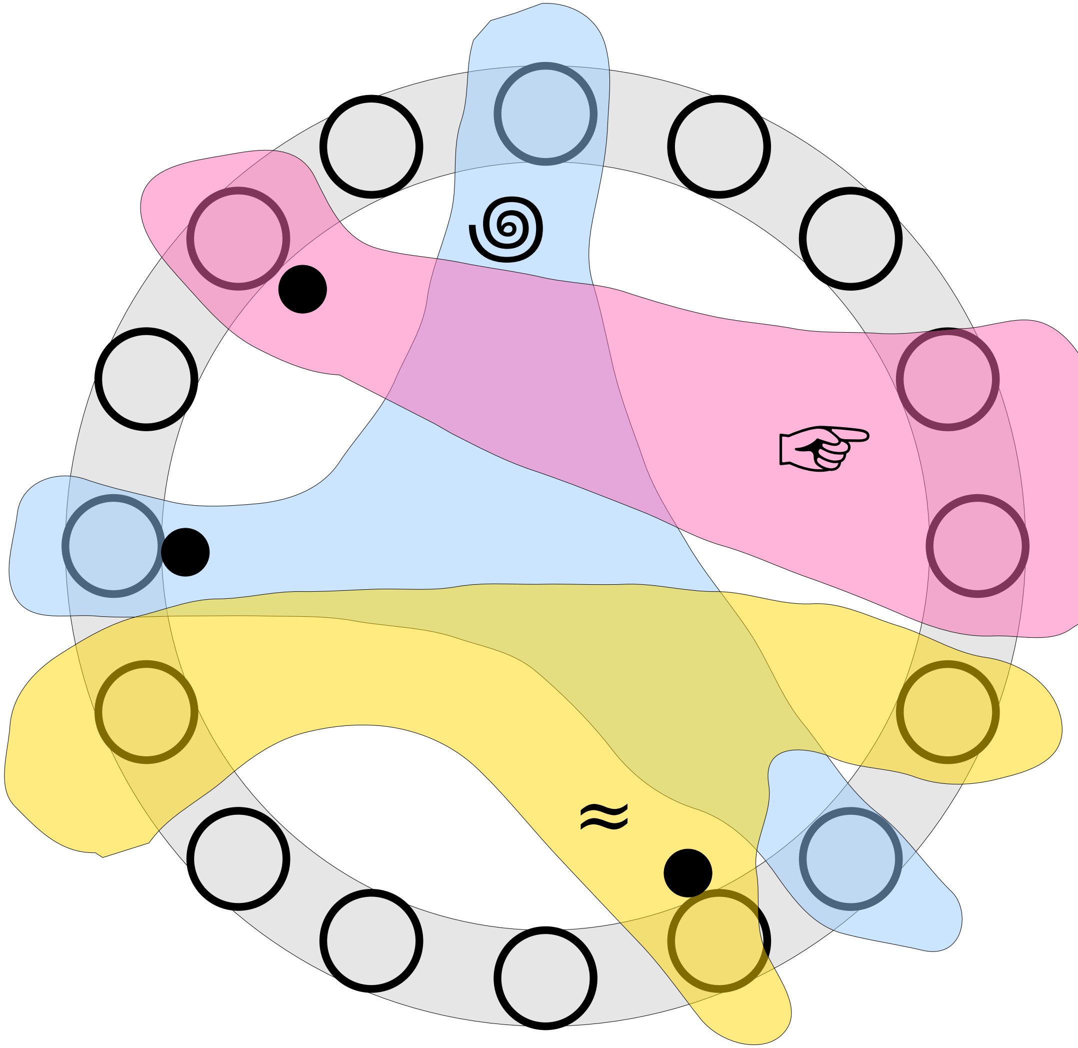
d-ary Cuckoo Hashing

- A ***hypergraph*** is a generalization of a graph.
- It consists of nodes and ***hyperedges***, where a hyperedge can link any number of nodes.
- A ***d-regular*** hypergraph is one where each hyperedge links exactly *d* nodes.
- With *d*-ary cuckoo hashing, we get a *d*-regular cuckoo hypergraph.



d-ary Cuckoo Hashing

- A ***1-orientation*** of a hypergraph is a way of placing a dot on one endpoint of each hyperedge so that each node has at most one dot on it.
- This corresponds of an assignments of items to slots in a *d*-ary cuckoo hash table.



d-ary Cuckoo Hashing

- Below are the 1-orientability thresholds for d-regular hypergraphs.
  - These were worked out over a series of papers in the early 2000s.
- In d-ary cuckoo hashing, these give the theoretical maximum α where we can store n items in a table with m slots, where $n = \alpha m$.
- Notice that adding in even a single extra hash function dramatically increases the space efficiency of the table.

	$d = 2$	$d = 3$	$d = 4$	$d = 5$	$d = 6$
Theoretical max α	**0.500**	**0.917**	**0.976**	**0.992**	**0.997**

In Practice

- Both blocked cuckoo hashing with $b > 1$ slots per node and d-ary cuckoo hashing with $d > 2$ significantly improve the space usage of cuckoo hashing.
- In practice, blocked cuckoo hashing tends to be faster than d-ary cuckoo hashing.
    - While we tend to forget this, evaluating hash functions is costly.
    - Blocked cuckoo hashing has better locality of reference, since there are (typically) only two cache misses.
- What happens if you combine these approaches together? That’s something you’ll see on the next problem set.

To Summarize

Summary of Cuckoo Hashing

- Cuckoo hashing is a fast and powerful way to build perfect hash tables.
- We can increase the number of hash functions to increase the load factor, though at a cost to lookup and insert times.
- We can increase the number of items per slot to increase the load factor, though at a cost to lookup and insert times.

Major Ideas for Today

- Randomized data structures using multiple hash functions can often be analyzed from a graph-theoretic perspective.
- Many properties of random graphs and hypergraphs exhibit sharp phase transitions.
- Running experiments is a great way to learn more about randomized data structures.

Next Time

- ***Amortized Analysis***
  - Trading aggregate runtime for per-operation runtime.
- ***Potential Functions***
  - Lying with mathematics, but doing so honestly.