# 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

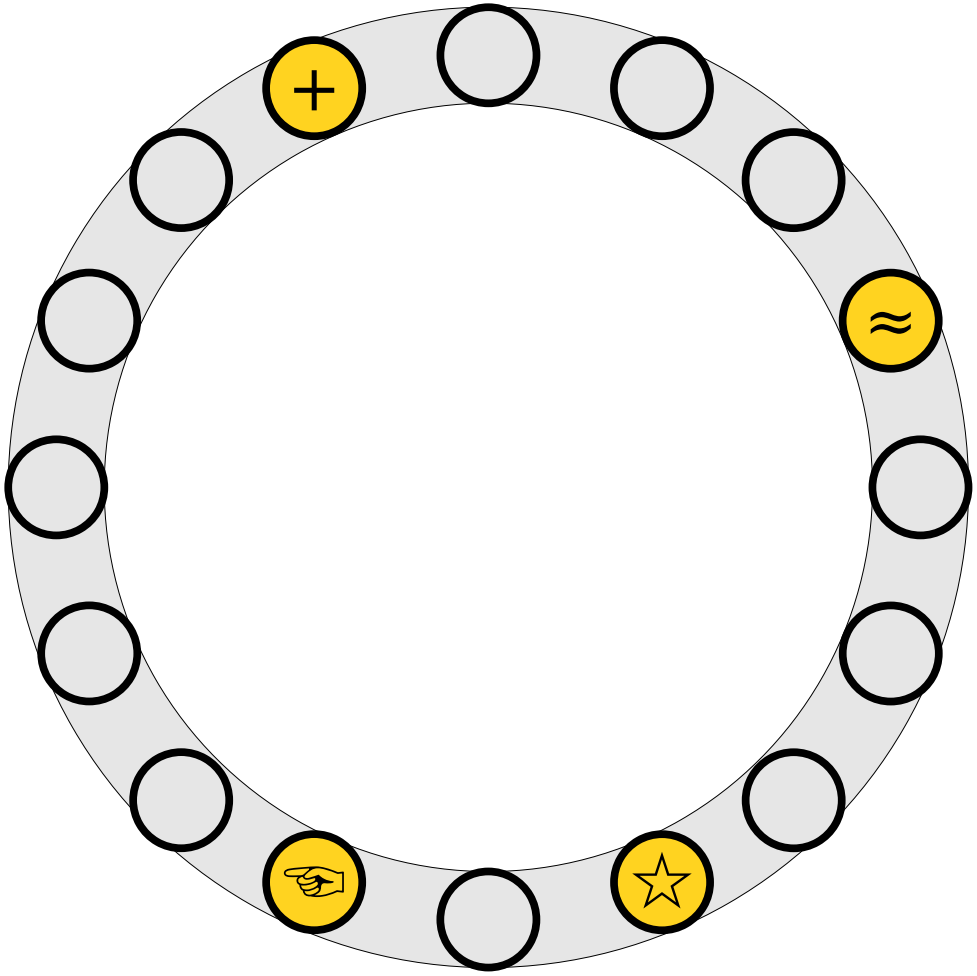
# Collision Resolution

- 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.)
  - ***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

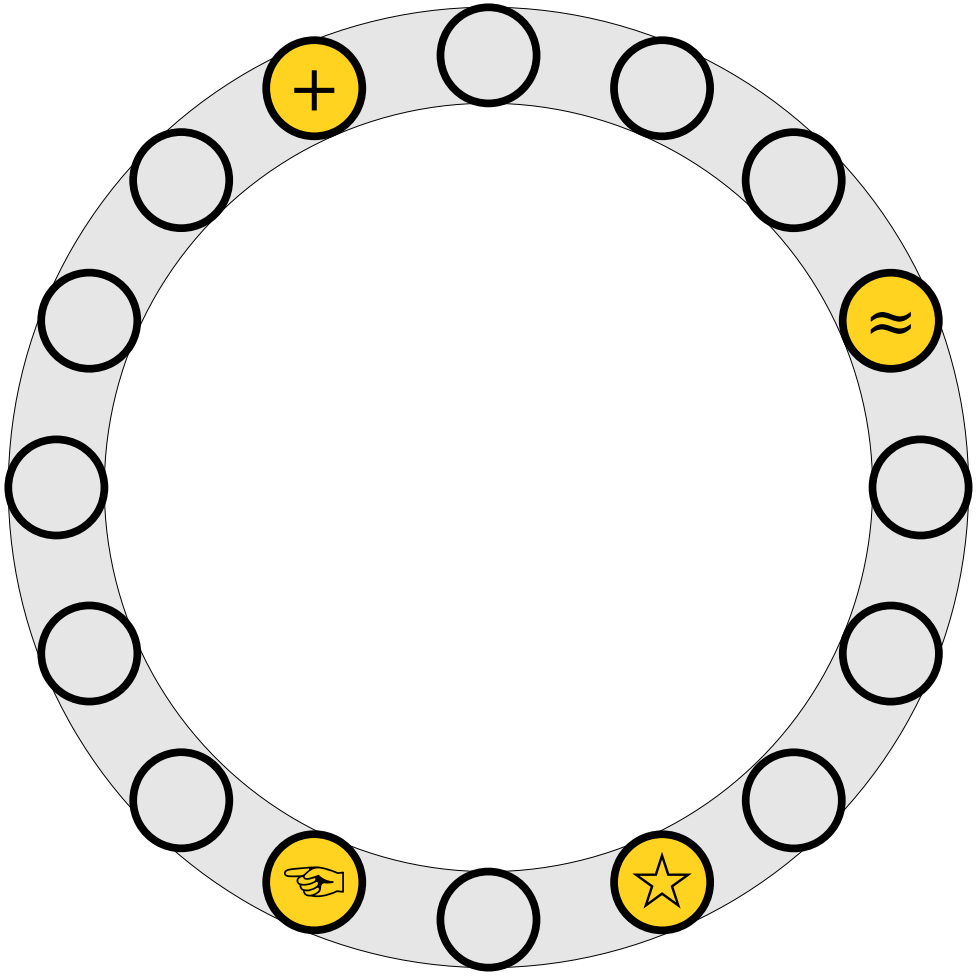
# Cuckoo Hashing

- 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 $\{ 0, 1, 2, \ldots, m - 1 \}$.



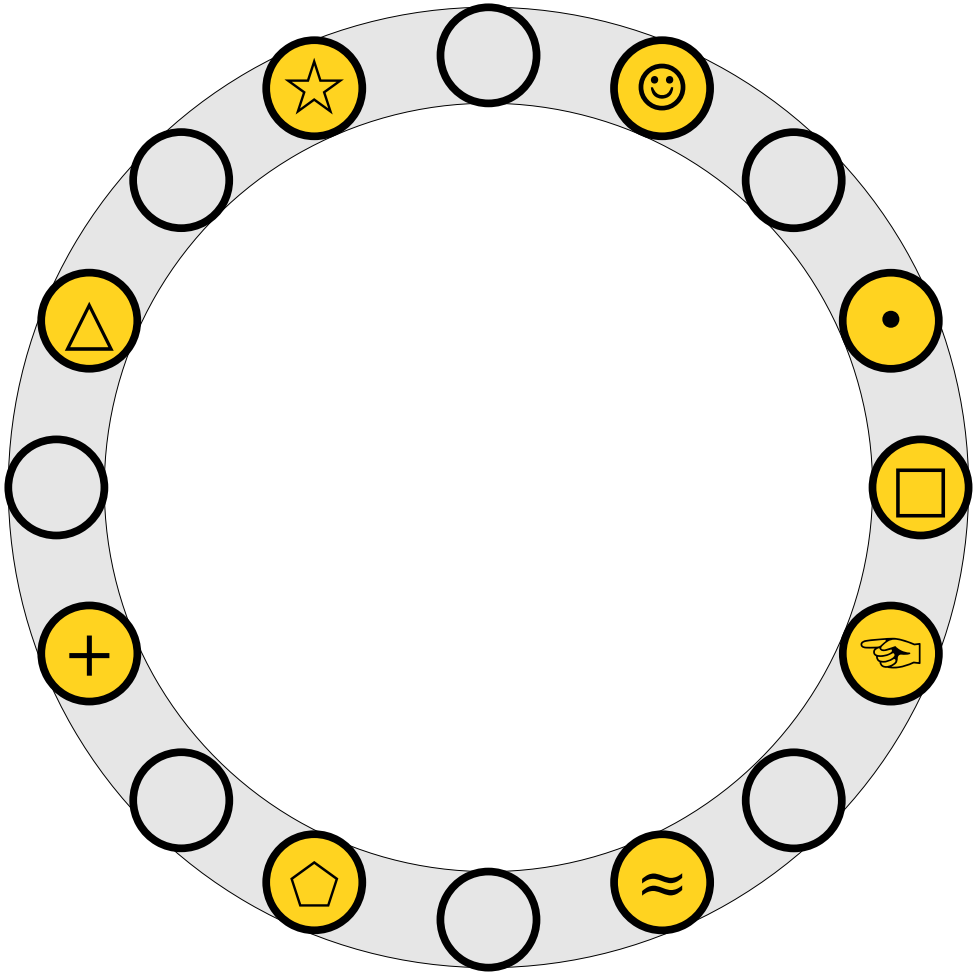
# Cuckoo Hashing

- ***The Rule:*** Any item $x$ must either be at position $h_1(x)$ or position $h_2(x)$ in the table.
  - (We assume $h_1(x) \neq h_2(x)$ for all $x$; this is easy to achieve in practice.)
- 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.



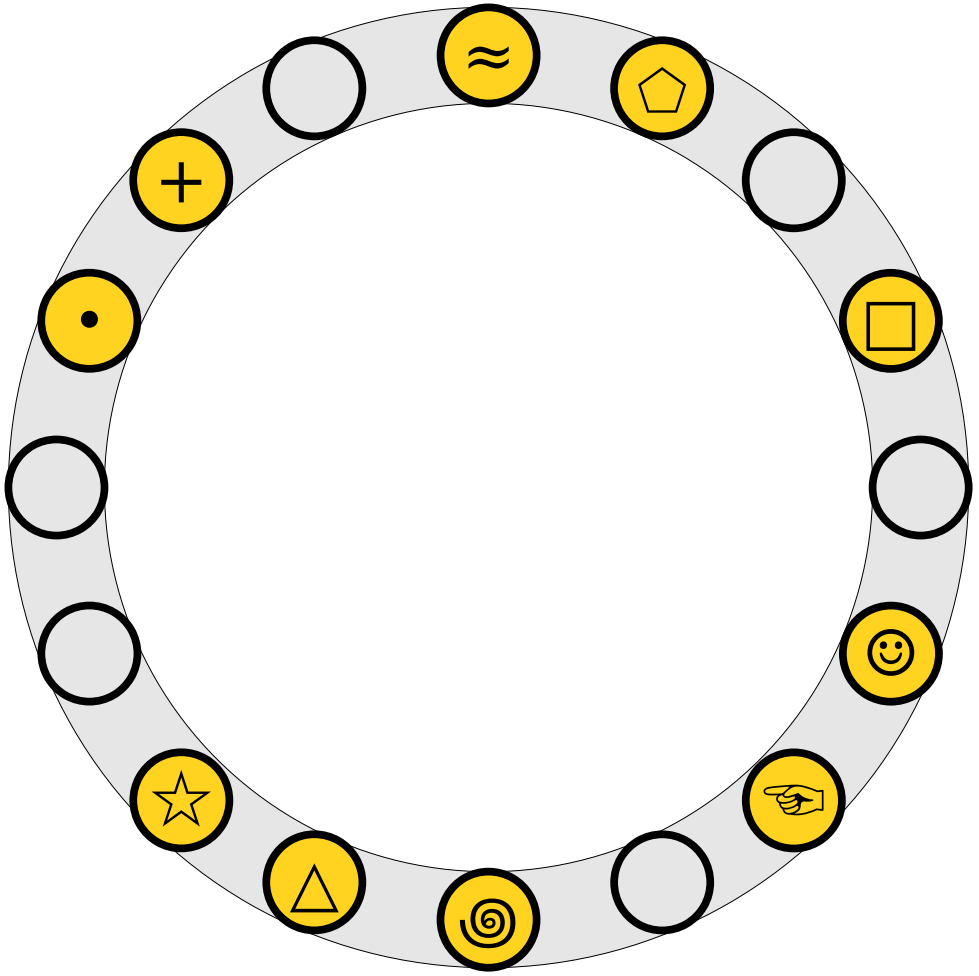
# Cuckoo Hashing

- 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.



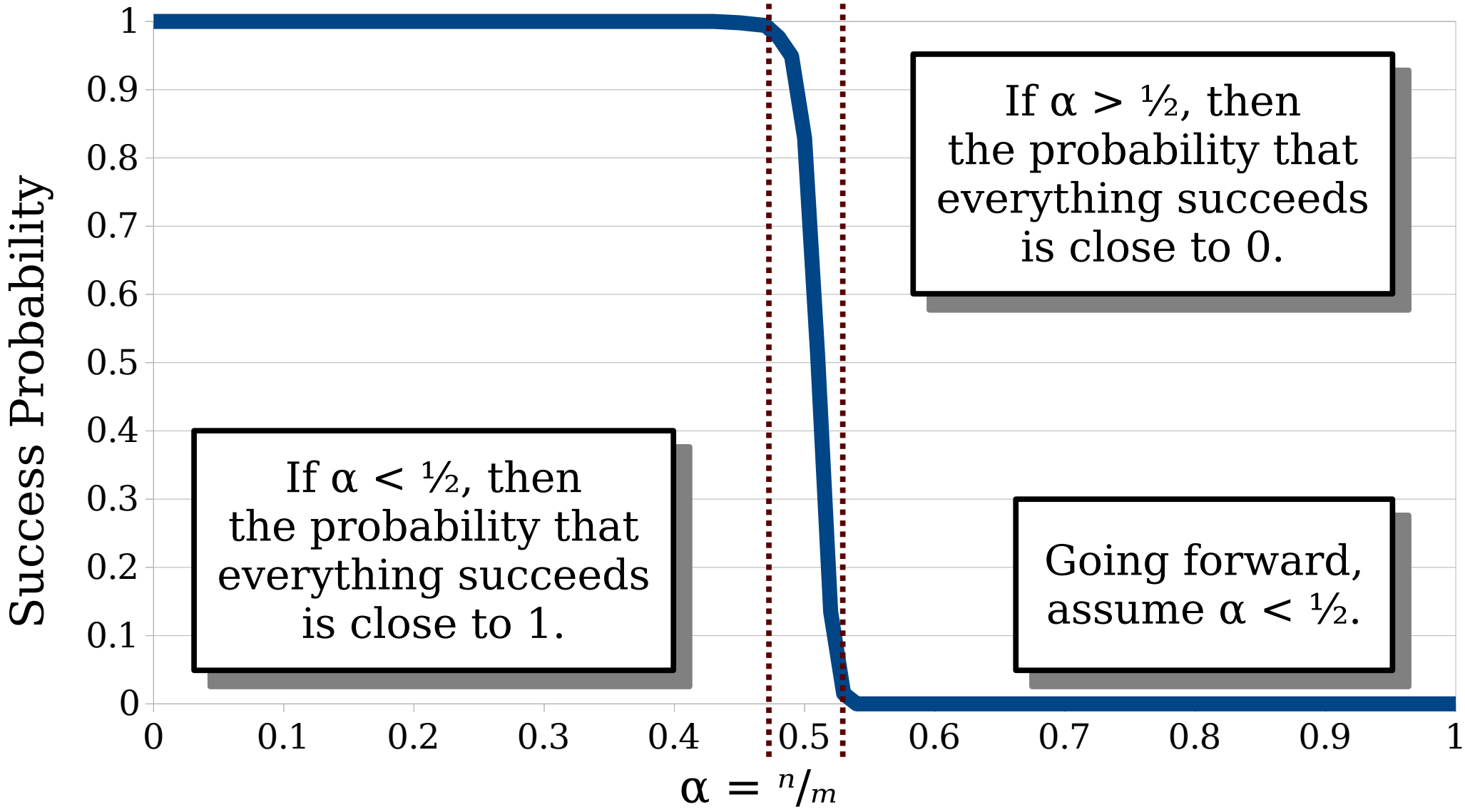
# Cuckoo Hashing

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

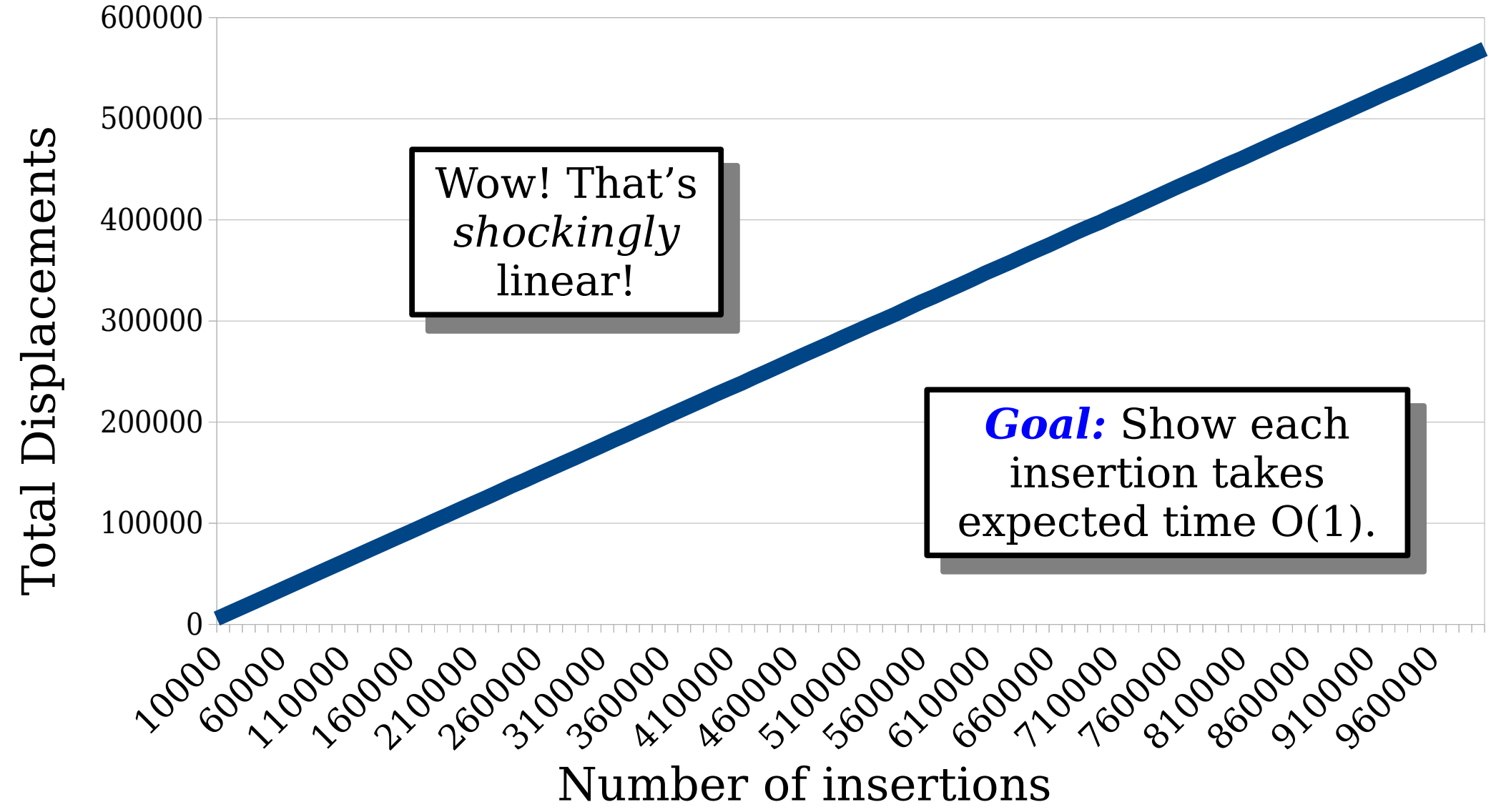


# How efficient is cuckoo hashing?

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




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*** $\alpha = n/m$?



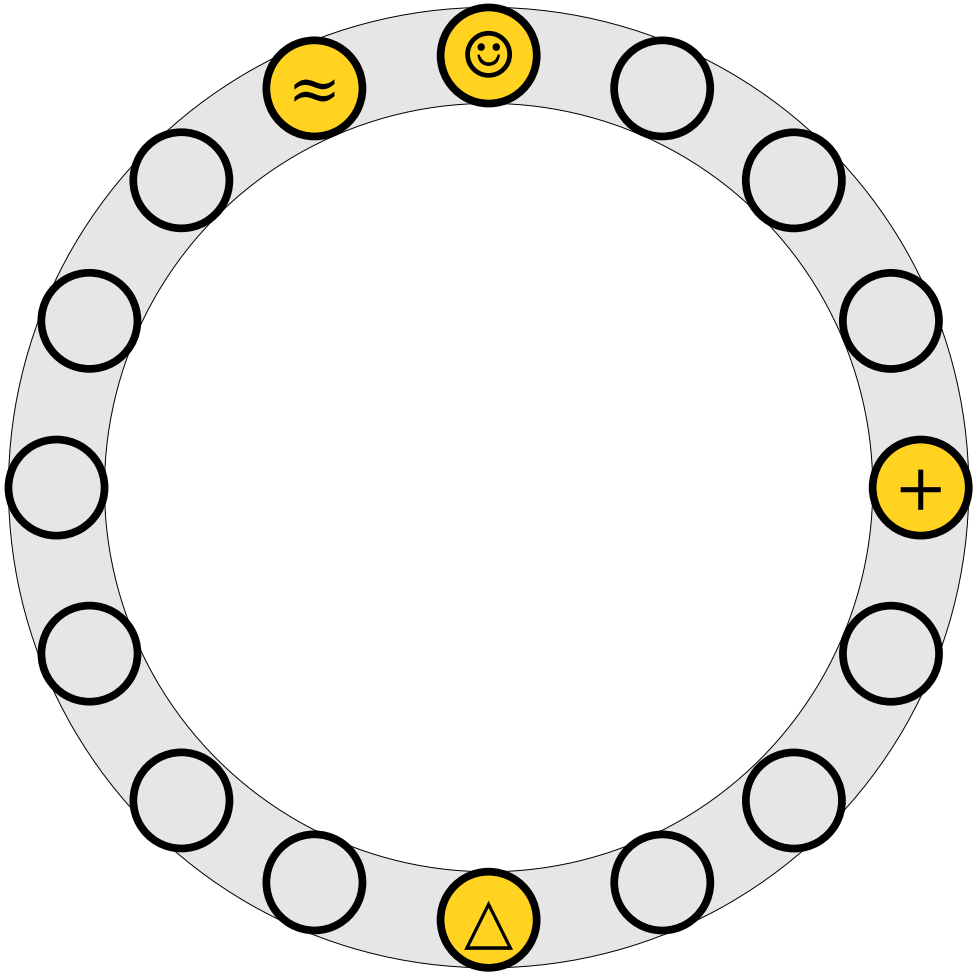

Suppose we store $n$ total elements in a table with $m$ slots, where $n < \frac{1}{2}m$.

How many total displacements occur across all insertions?

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

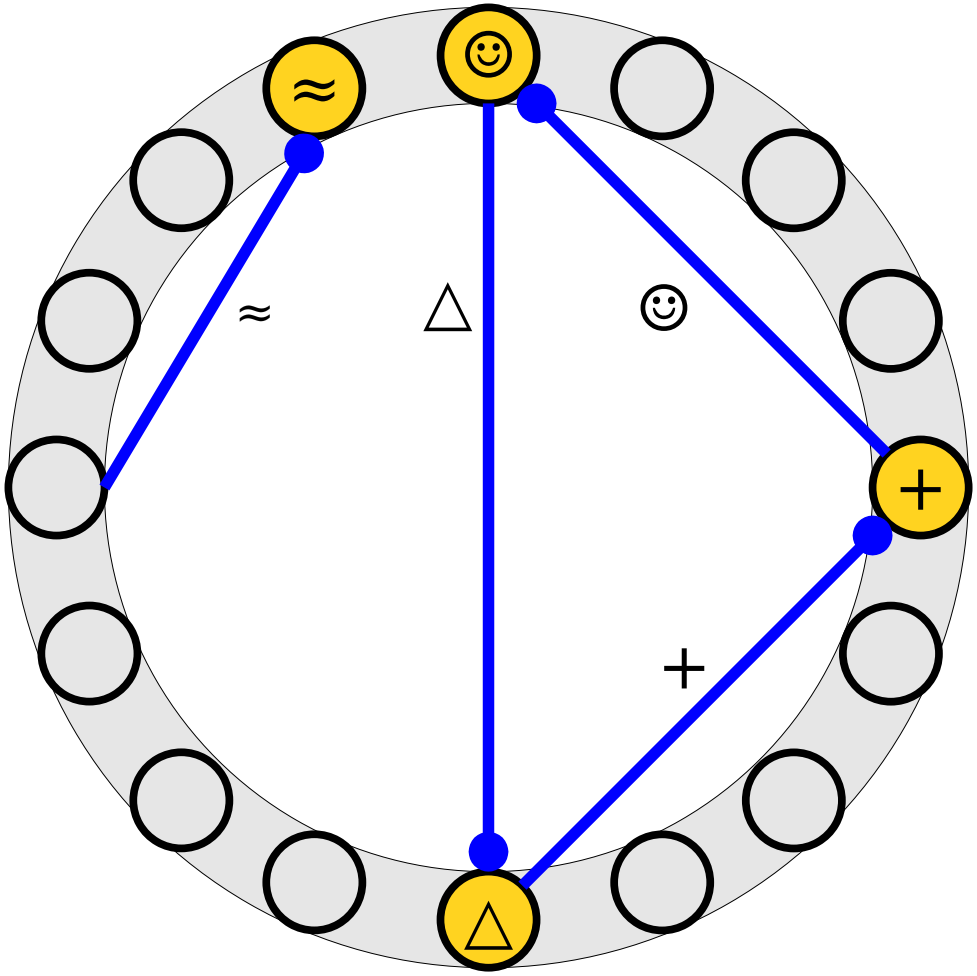
# Analyzing Cuckoo Hashing

- 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.



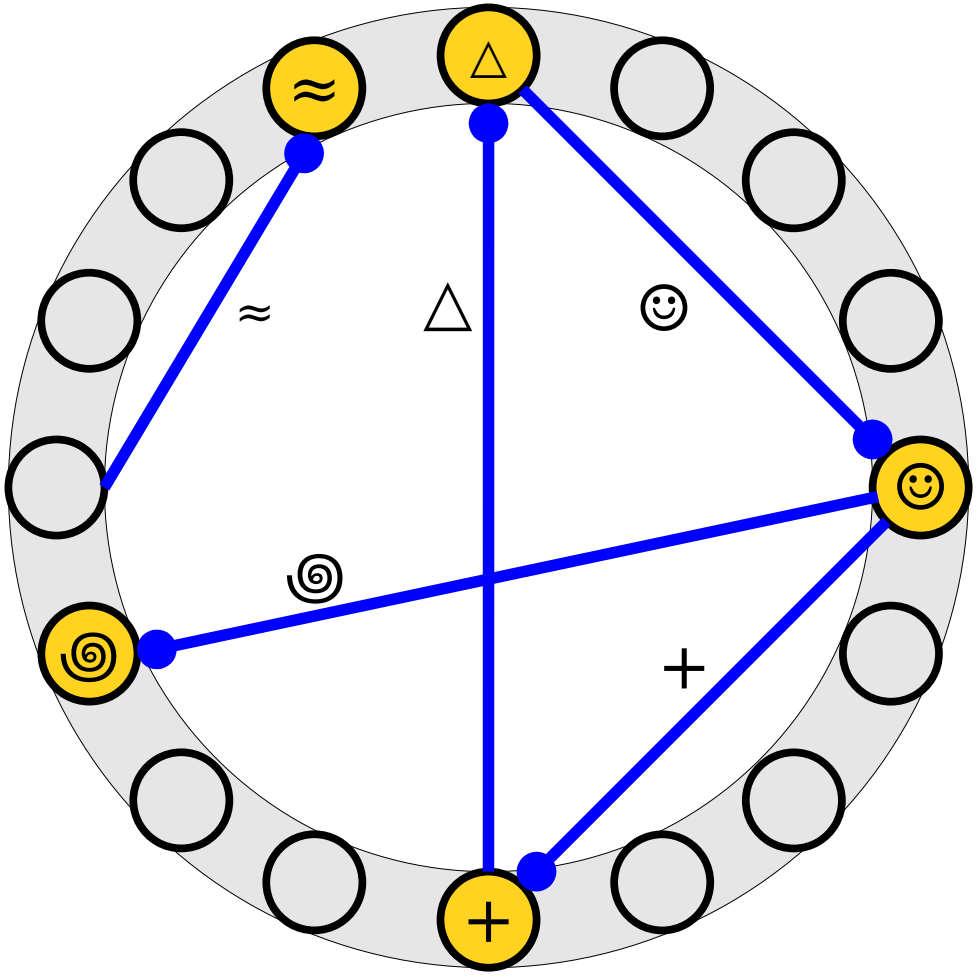
# The Cuckoo Graph

- 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

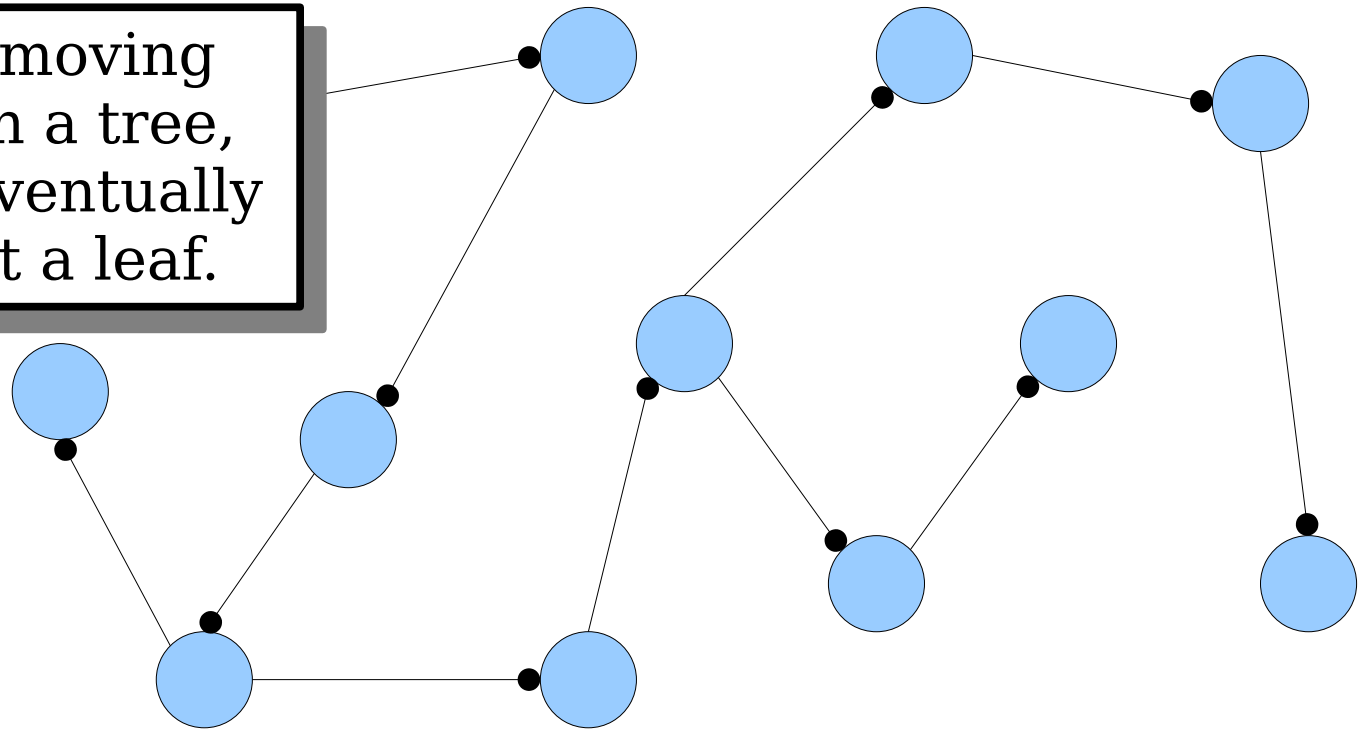
- Inserting an element into a cuckoo hash table adds a new edge to the graph linking two nodes (slots).
- The chain of displacements corresponds to flipping edges.



# The Cuckoo Graph

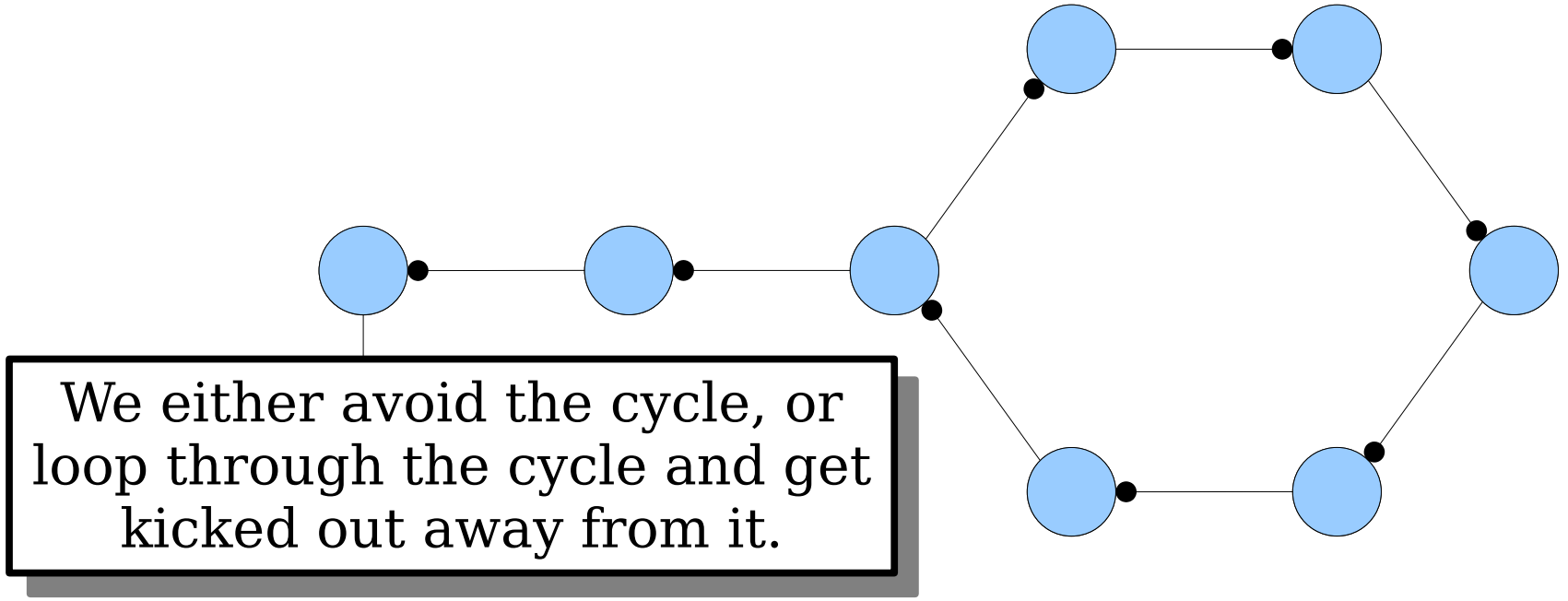
- ***Claim 1:*** If $x$ is inserted into a cuckoo hash table, the insertion succeeds if the connected component containing $x$ contains either no cycles or only one cycle.

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



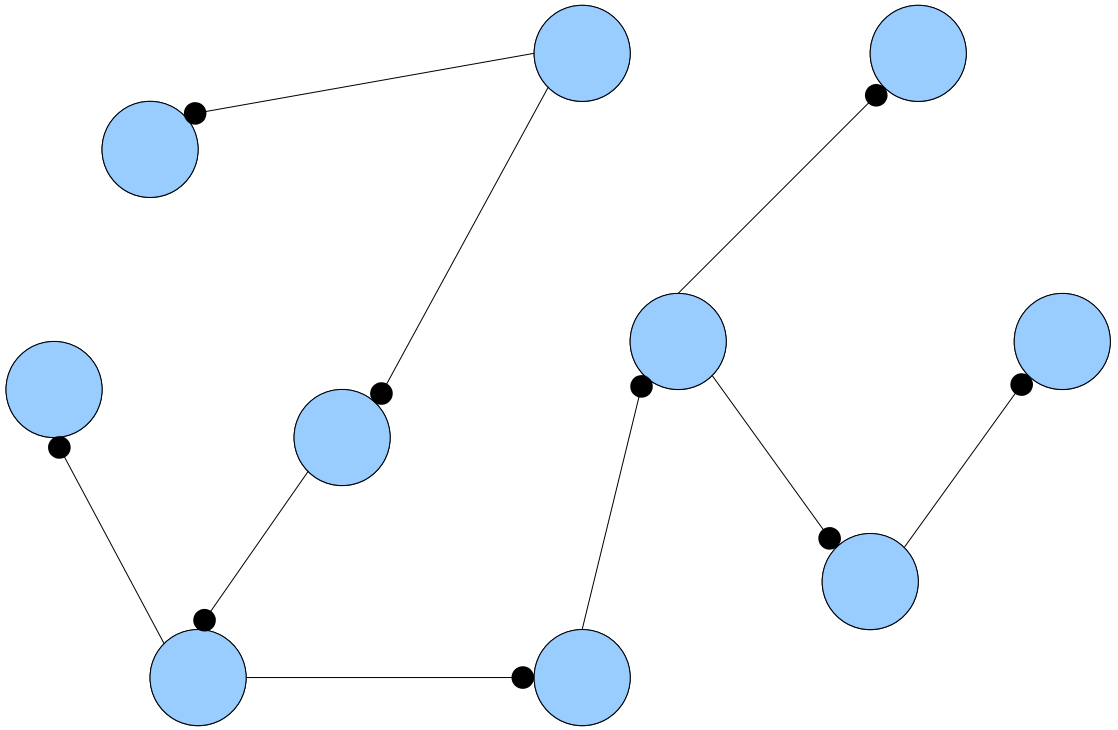
# The Cuckoo Graph

- ***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.



# The Cuckoo Graph

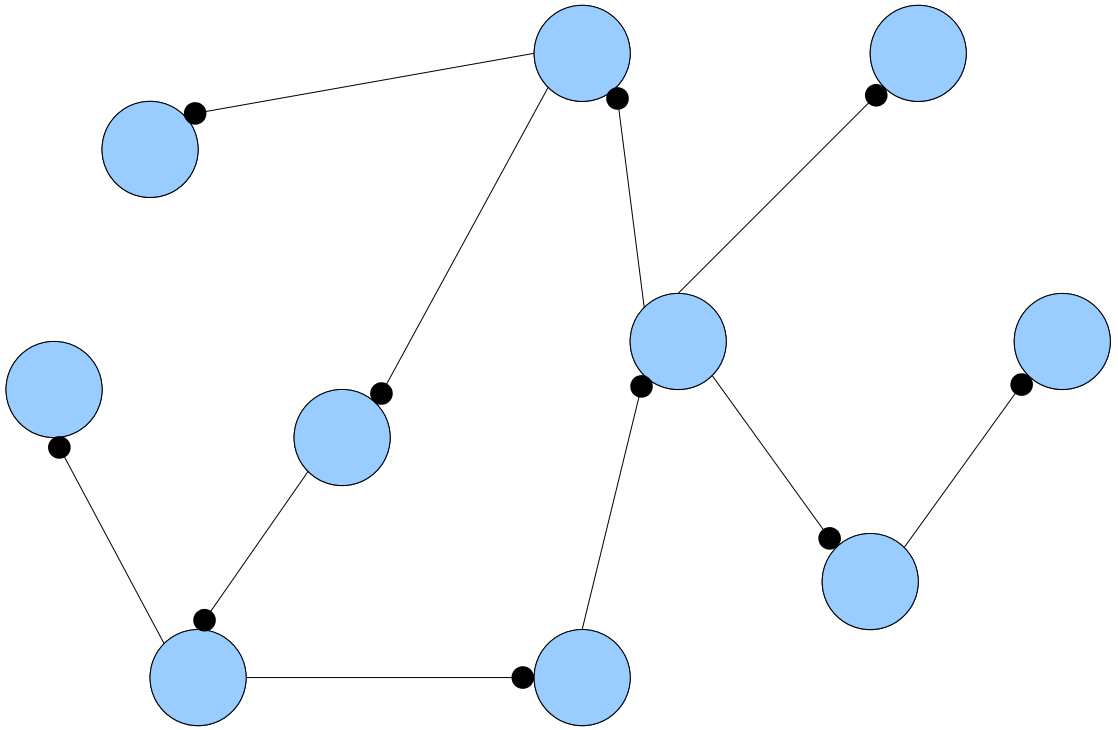
- ***Claim 2:*** If $x$ is inserted into a cuckoo hash table, the insertion fails if the connected component containing $x$ contains more than one cycle.



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

# The Cuckoo Graph

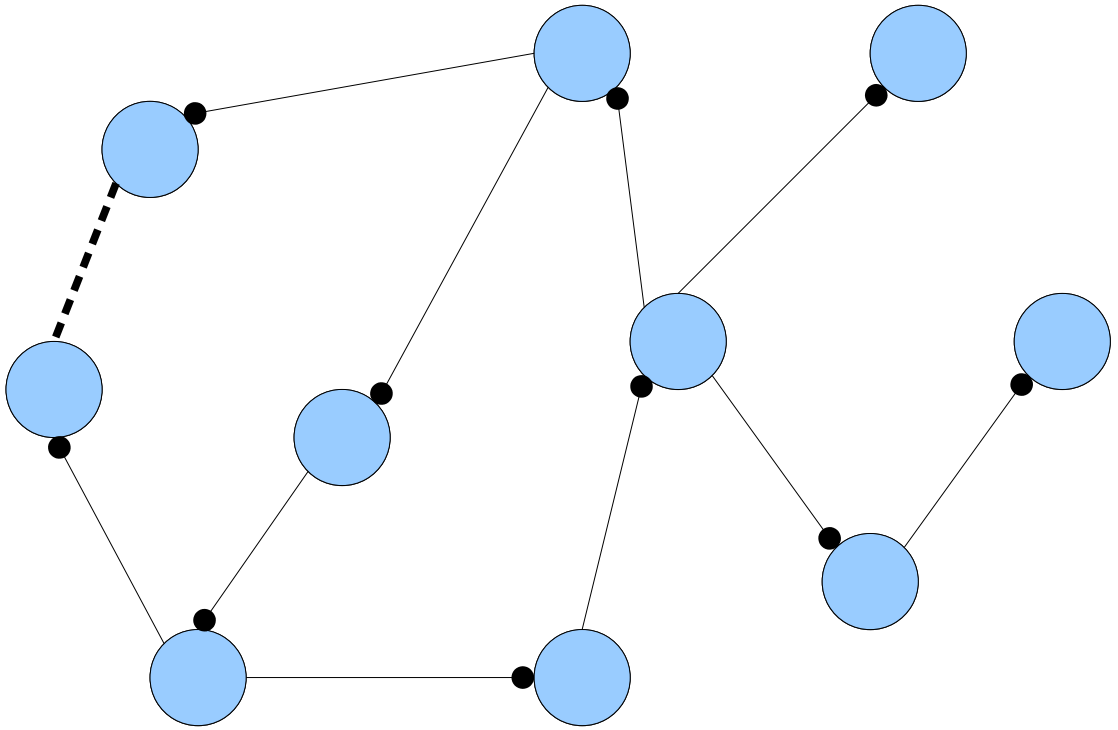
- ***Claim 2:*** If $x$ is inserted into a cuckoo hash table, the insertion fails if the connected component containing $x$ contains more than one cycle.



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

# The Cuckoo Graph

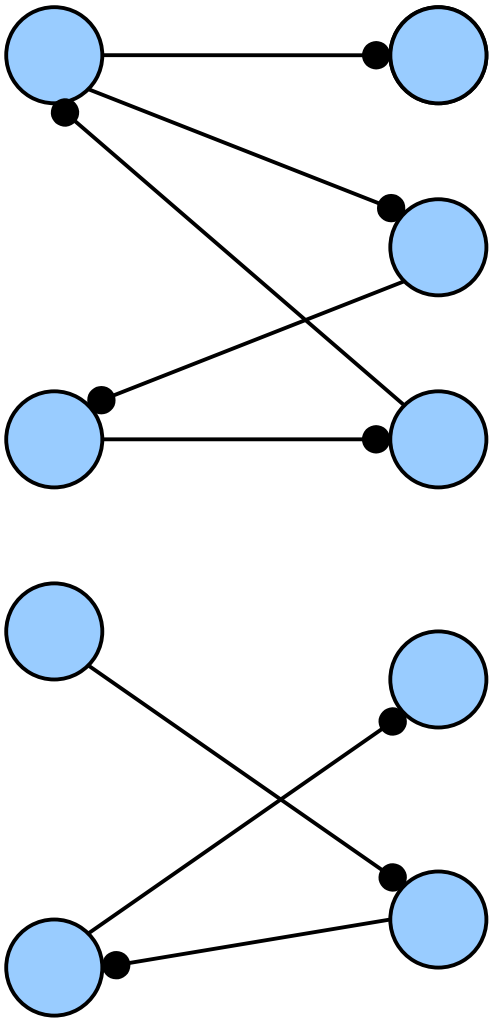
- ***Claim 2:*** If $x$ is inserted into a cuckoo hash table, the insertion fails if the connected component containing $x$ contains more than one cycle.



***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.)***

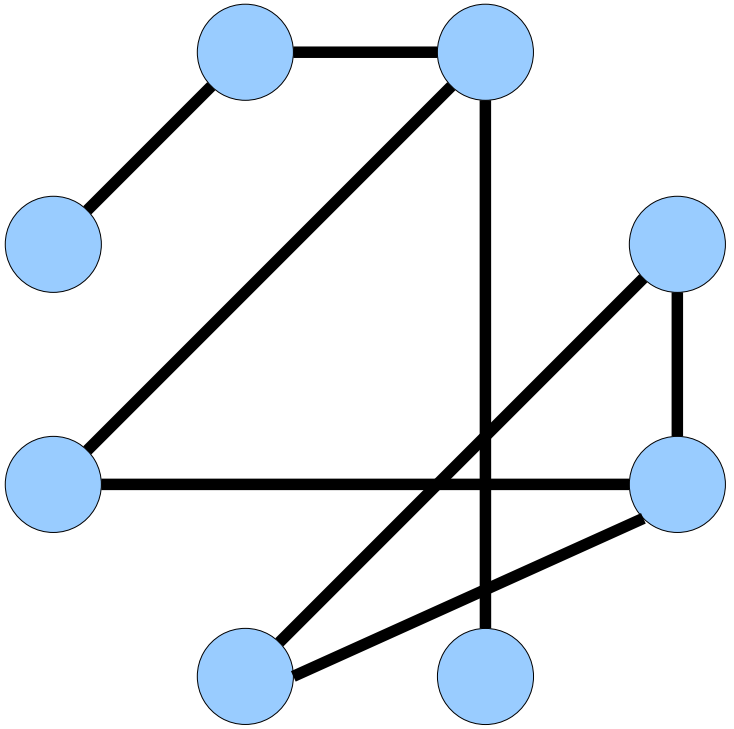
***What is the probability that a connected component in the cuckoo graph is complex?***

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

# The Erdős–Rényi model

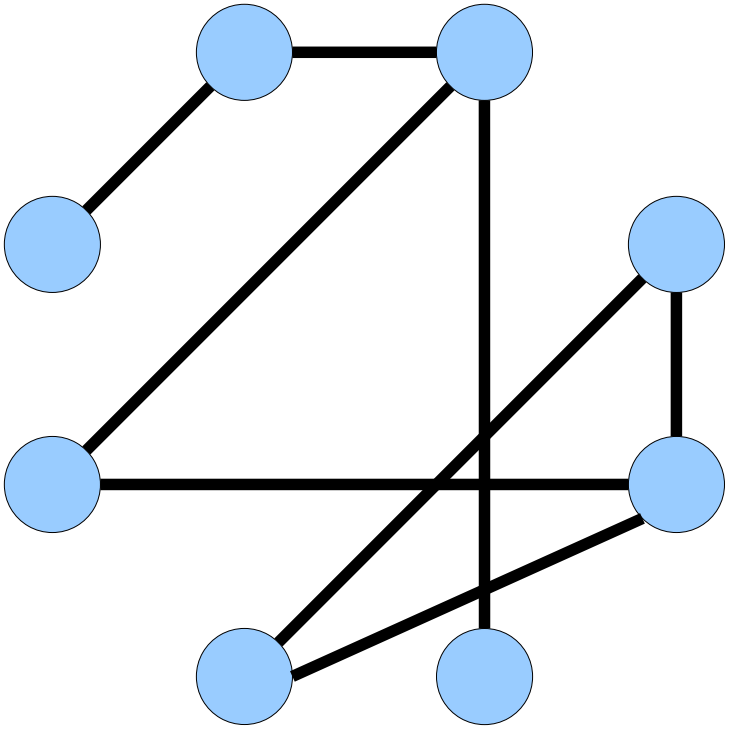
# Random Graph Evolution

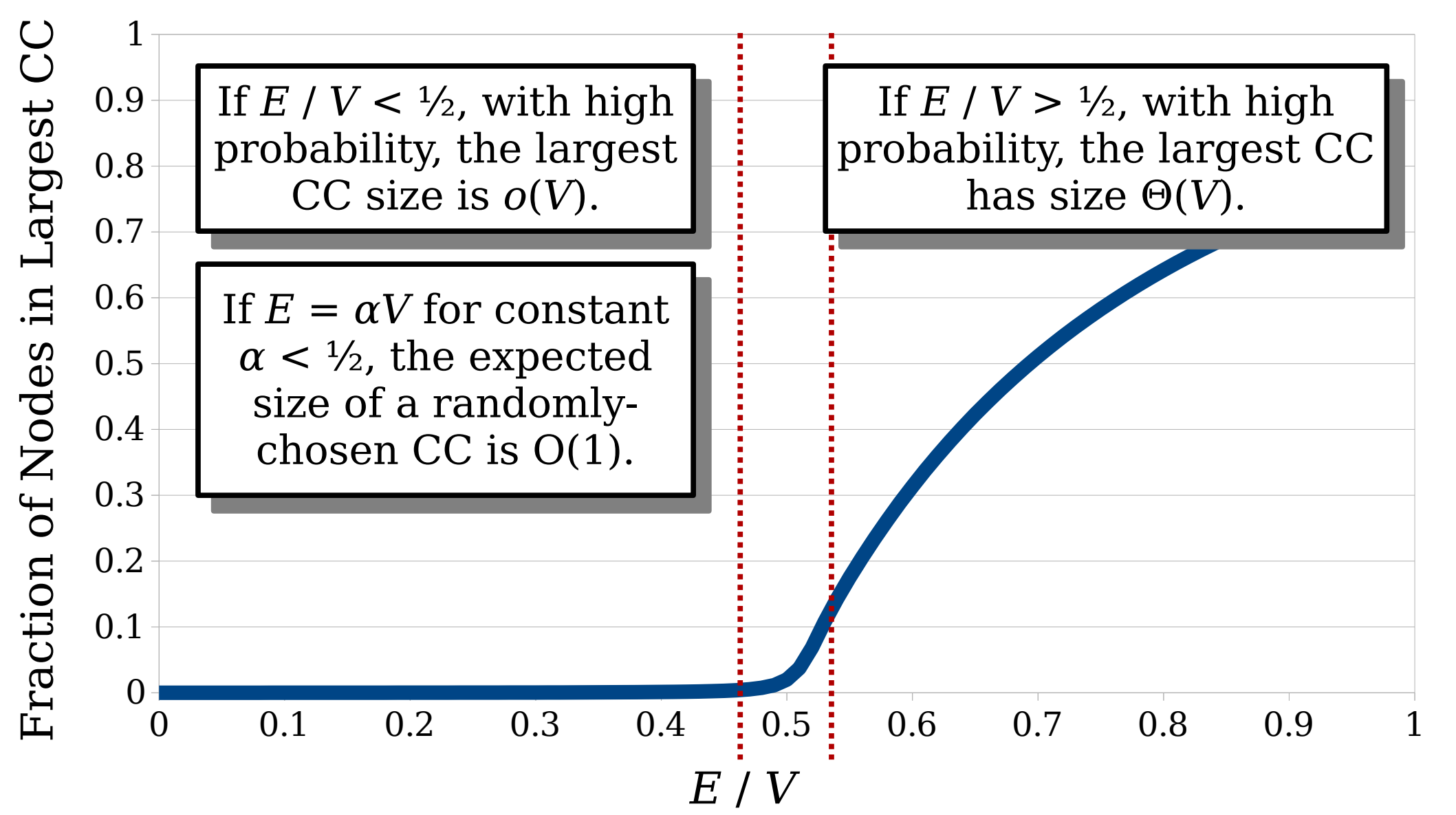
- Consider a graph with $V$ nodes and no edges.
- Incrementally add $E$ edges to the graph, each chosen uniformly at random, possibly with repetition.
- ***Question:*** What properties will this graph (probably) have?



# 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.

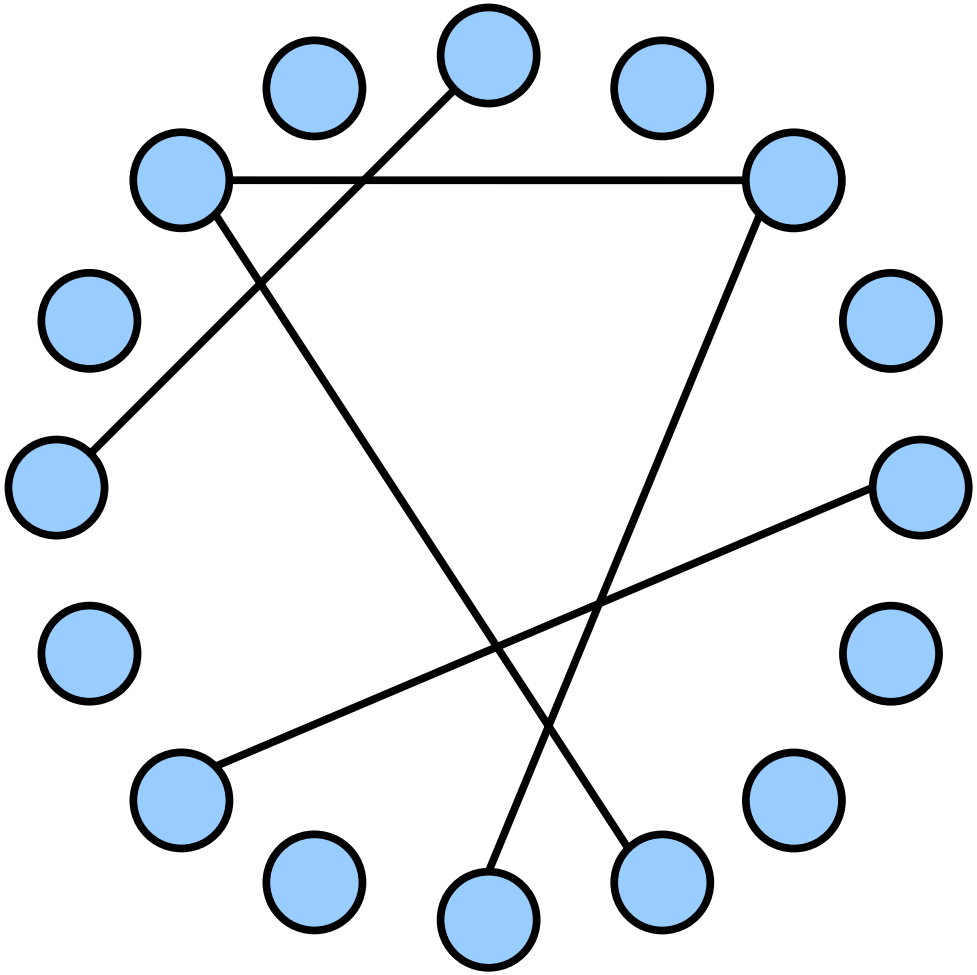





Consider a random (multi)graph $G$ with $V$ nodes and $E$ edges. What fraction of the nodes are in the largest connected component of $G$, as a function of $E / V$?

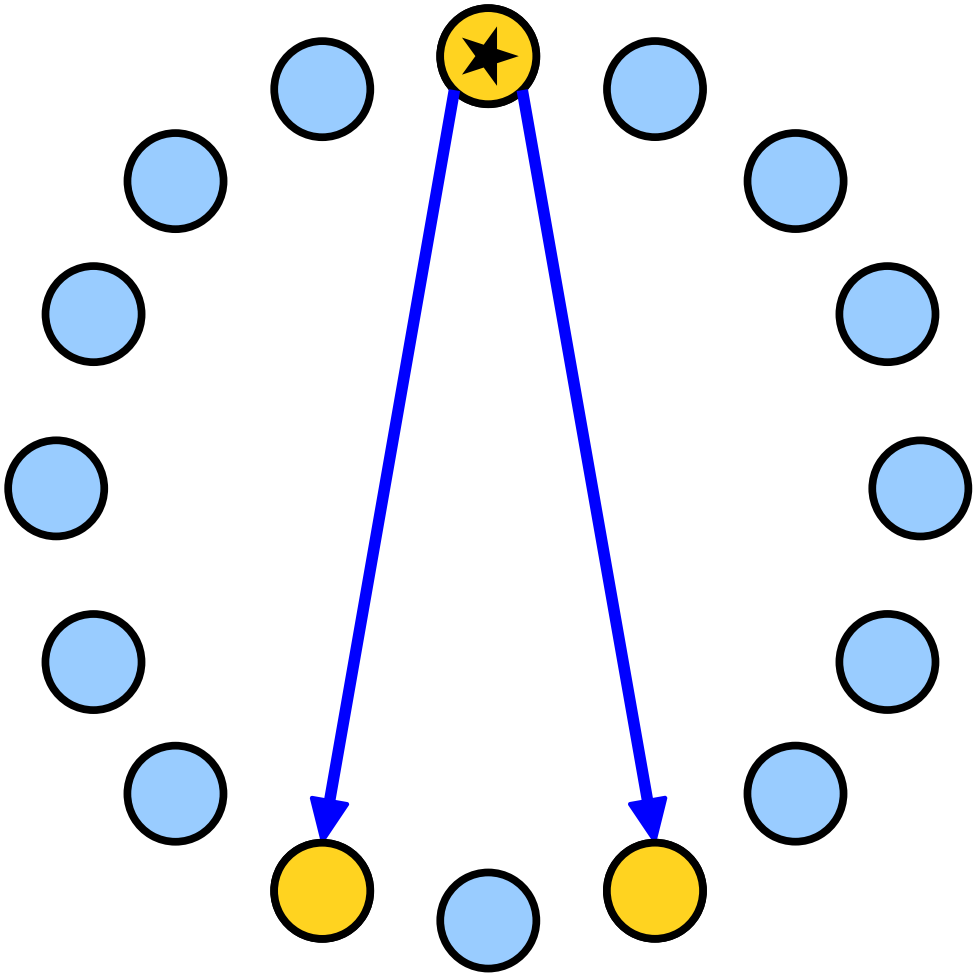
# Sizing a Connected Component

- ***Goal:*** Show that if $E = \alpha V$ for some constant $\alpha < ½$, then the expected size of a CC in a randomly-built graph is O(1).
- This seems hard, so let's step away from random graphs for a moment.
- Suppose you have a graph $G$ and a node $v$ in the graph.
- What algorithms might you use to determine the size of the connected component containing $v$?



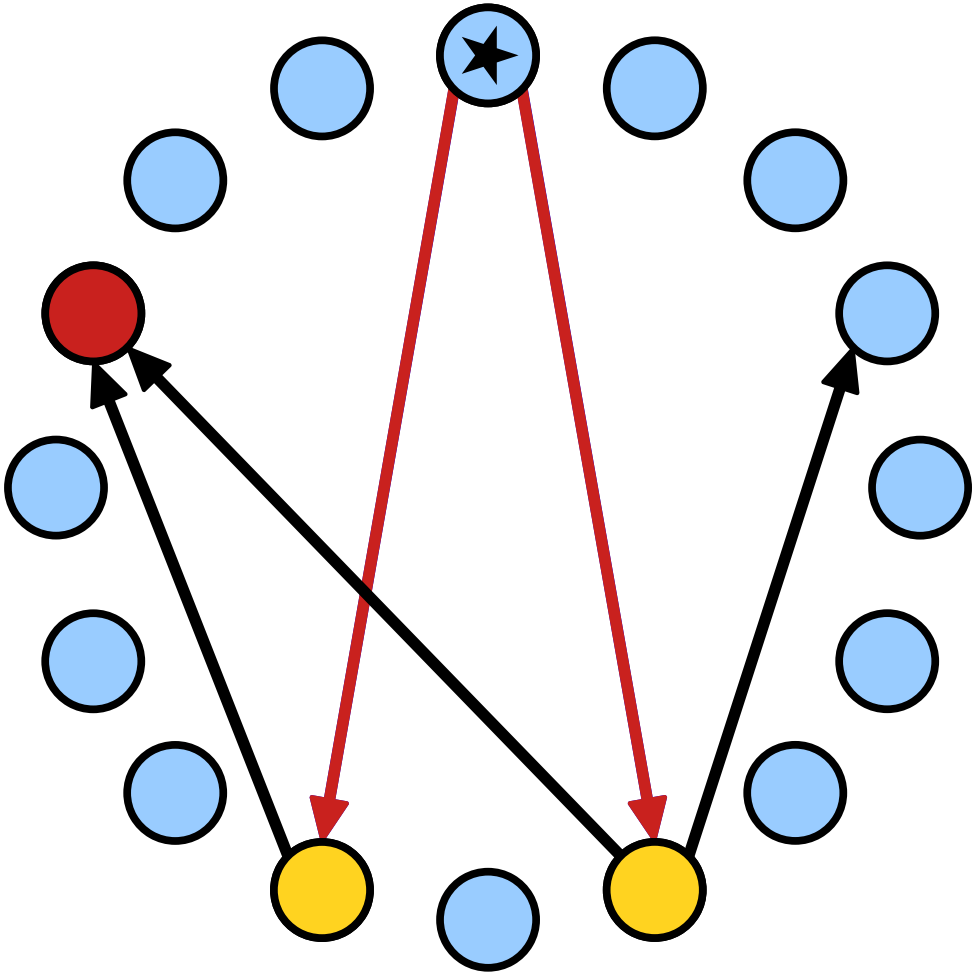
# Sizing a Connected Component

- Pick a starting node for our BFS.
- We want to model its child count in the BFS tree.
  - There are $E$ total edges.
  - Each edge has a $^2/_V$ chance of touching our node.
- So this node's number of children is a Binom($E$, $^2/_V$) random variable.
- We can approximate this by a Poisson($^{2E}/_V$) = **Poisson(2α)** random variable.
- ***Key Insight:*** 2α < 1, so on expectation the start node has fewer than one child.



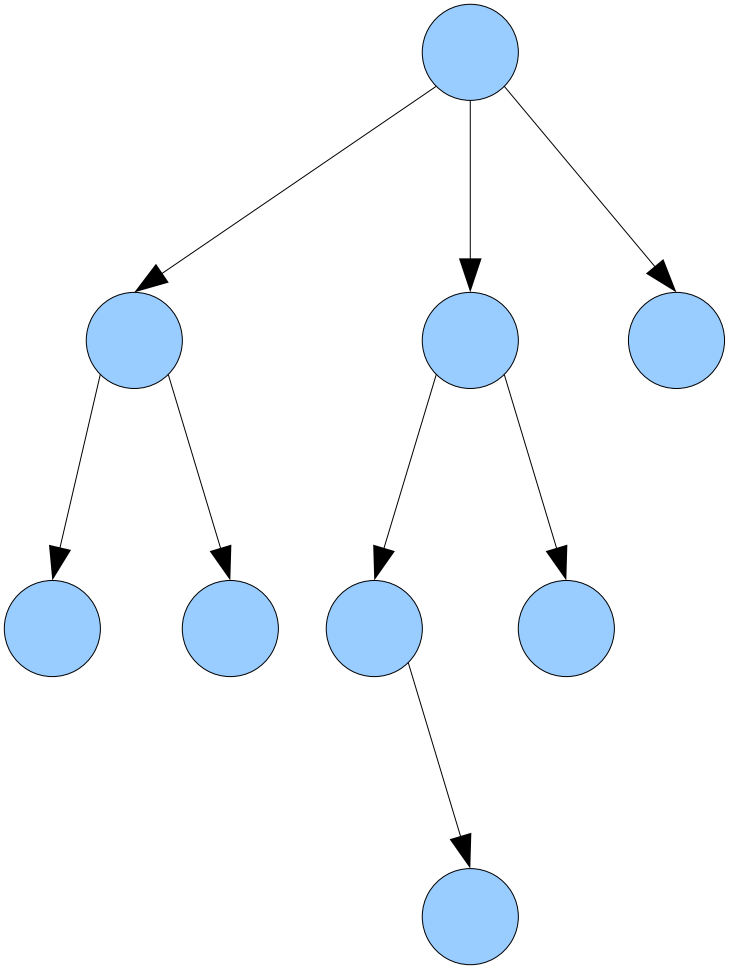
# Sizing a Connected Component

- Each new node kinda sorta ish also touches a number of new nodes that can be modeled as a Poisson(2α) 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 in the next BFS layer.
- ***Intuition:*** Each of these children has, on expectation, $2\alpha < 1$ children.



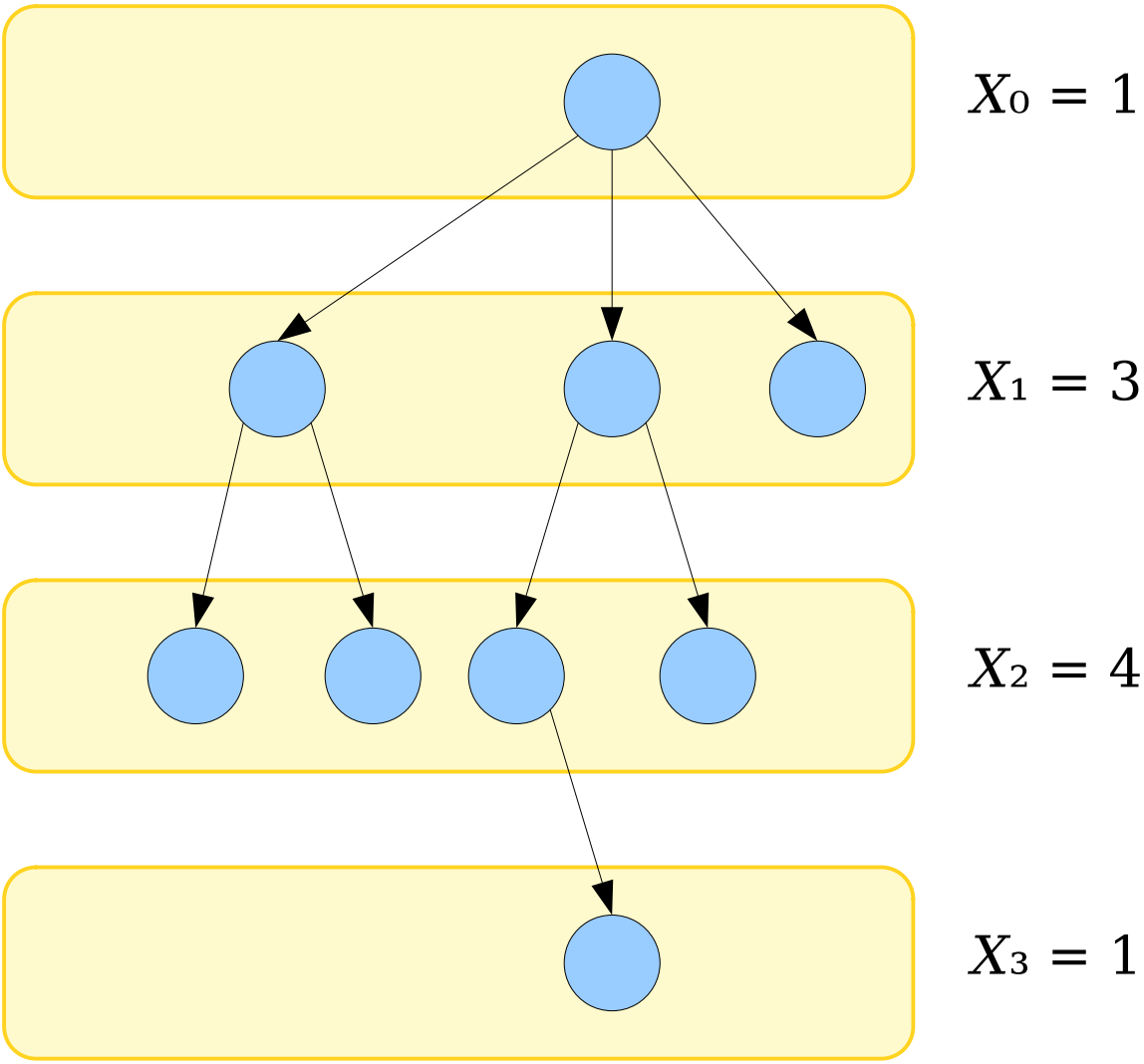
# Sizing a Connected Component

- Each node in the BFS tree has a number of children that is well-approximated as a Poisson($2\alpha$) random variable.
- The “expected branching factor” of the tree is $2\alpha < 1$.
- ***Intuition:*** Since the branching factor is less than 1, the number of nodes per layer decays geometrically, and the total number of nodes found by the BFS is expected to be O(1).
- ***Goal:*** Prove this.



# Modeling the BFS

Let $X_k$ denote the number of nodes in level $k$ of the tree.



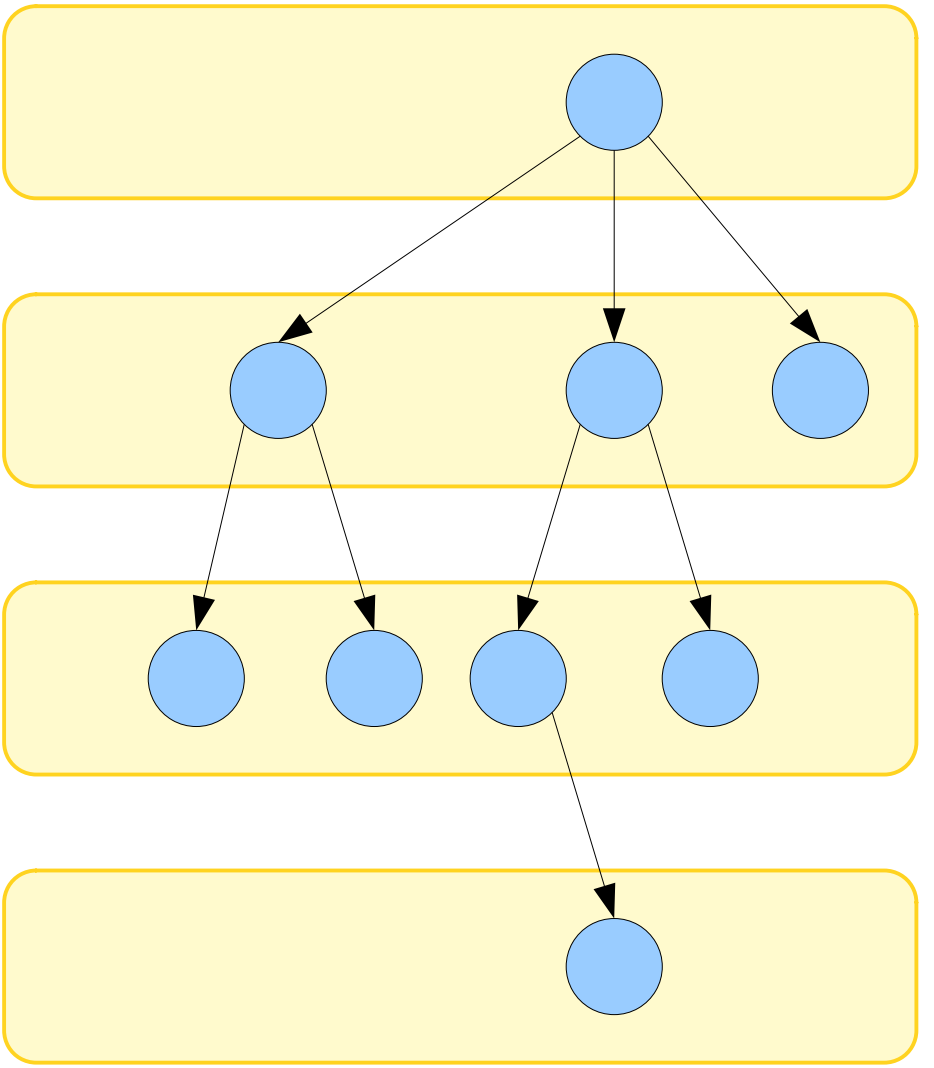
# Modeling the BFS

There is always one node here.

On expectation, we'd find $2\alpha$ nodes here.

On expectation, we'd find $(2\alpha)^2$ nodes here.

On expectation, we'd find $(2\alpha)^3$ nodes here.



$X_0 = 1$

$X_1 = 3$

$X_2 = 4$

$X_3 = 1$

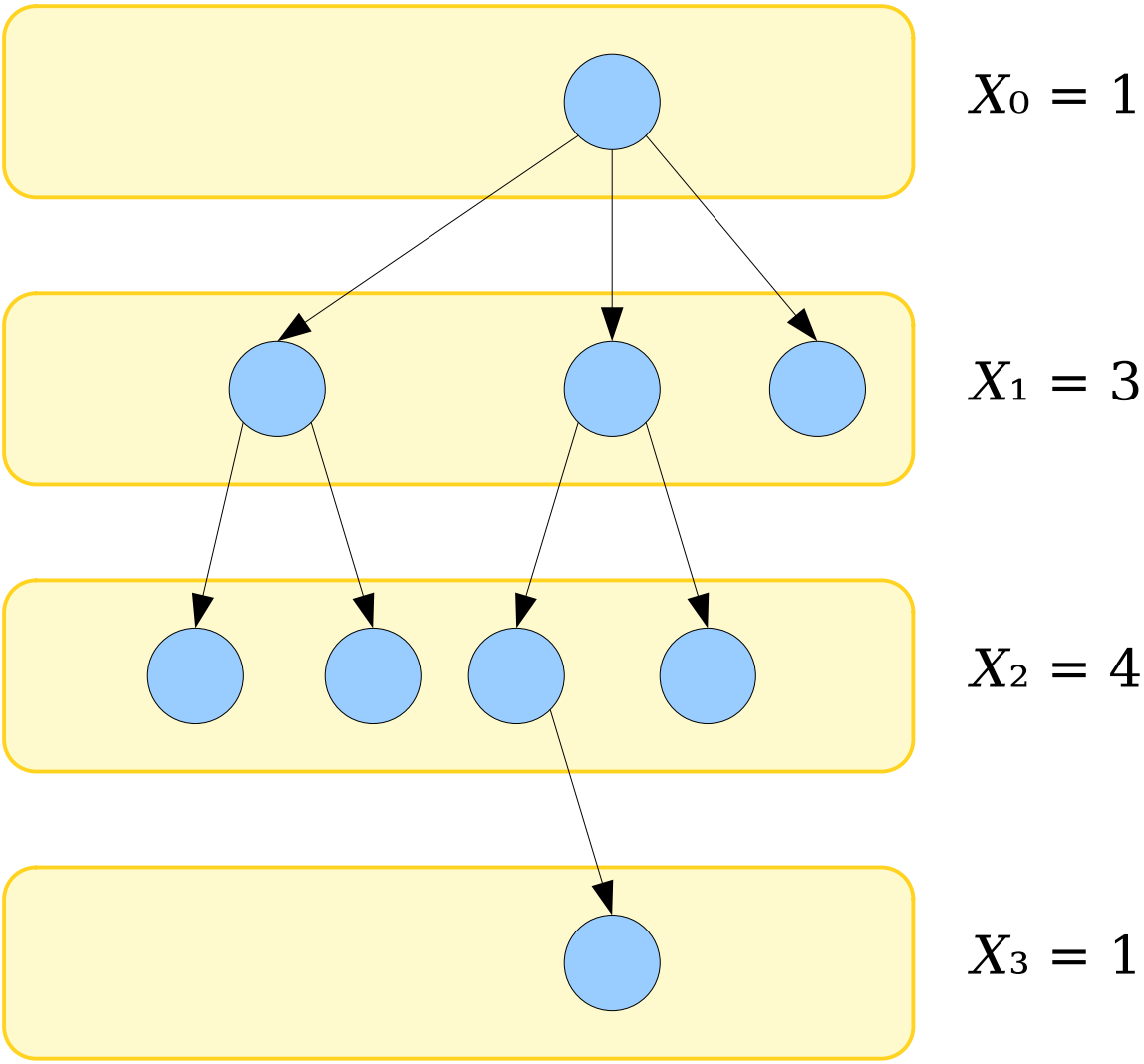
# Modeling the BFS

***Claim:*** $\mathrm{E}[X_k] = (2\alpha)^k$.

***Proof Idea:*** Show that

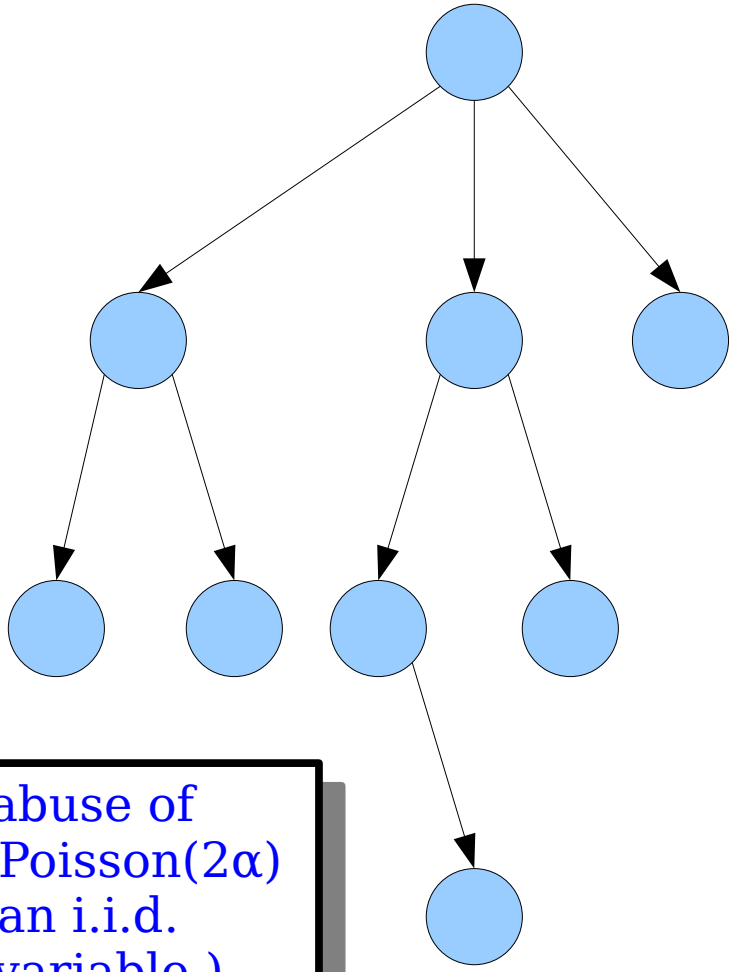
$$\mathrm{E}[X_{k+1}] = 2\alpha \cdot \mathrm{E}[X_k]$$

and apply induction. How do we do that?



# Modeling the BFS

- We can write a ***randomized recurrence relation*** for the variables $X_i$:
  - $X_0 = 1$
  - $X_{k+1} = \sum_{i=1}^{X_k} \text{Poisson}(2\alpha)$
- We can use this to rigorously establish that $E[X_{k+1}] = 2\alpha \cdot E[X_k]$.

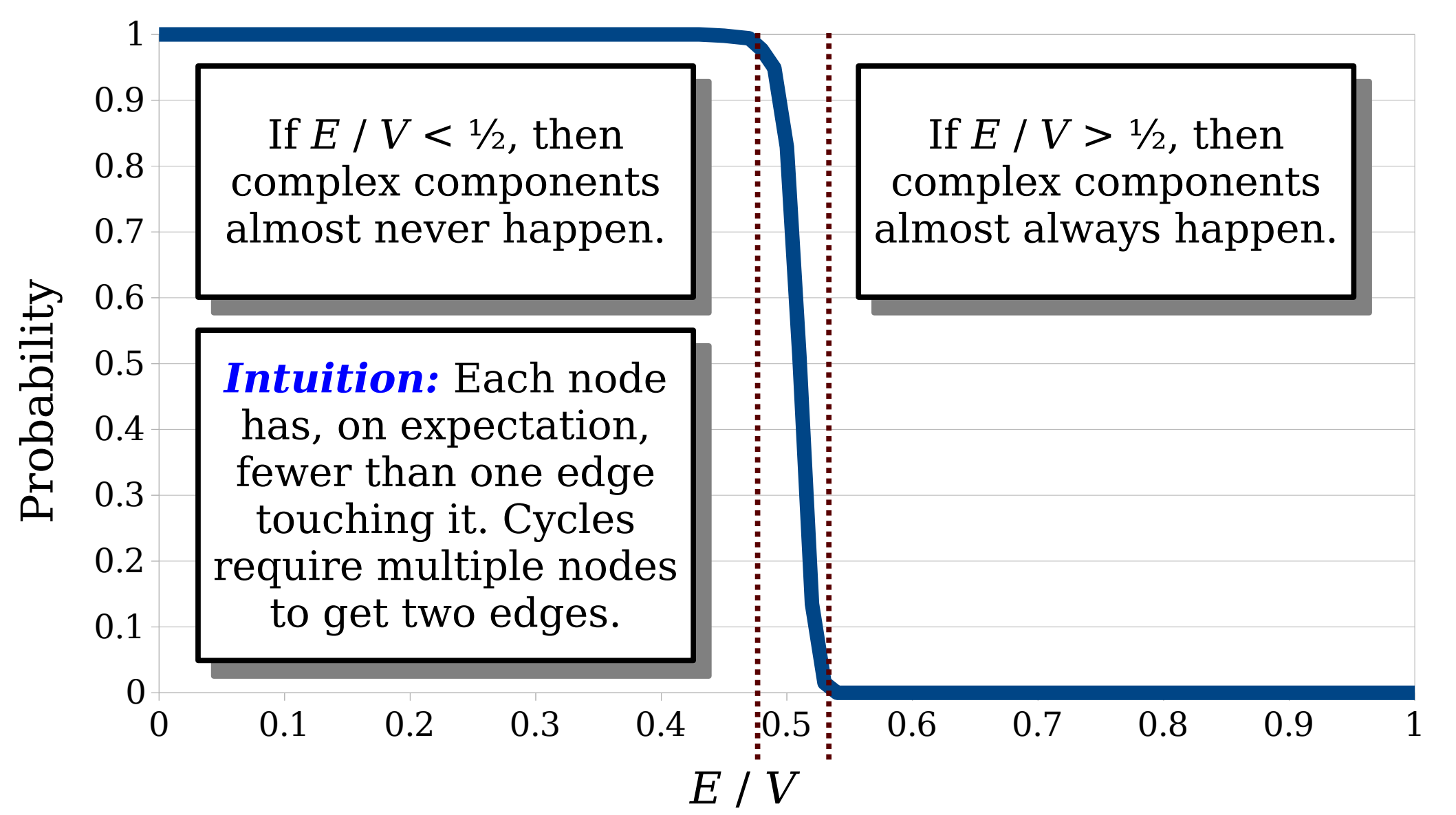



(This is an abuse of notation; each Poisson(2α) stands for an i.i.d. Poisson(2α) variable.)

$$
\begin{aligned}
\mathrm{E}[X_{k+1}] &= \mathrm{E}[\sum_{i=1}^{X_k} \mathrm{Poisson}(2\alpha)] \\
&= \sum_{j=0}^{\infty} \mathrm{E}[\sum_{i=1}^{X_k} \mathrm{Poisson}(2\alpha) \mid X_k = j] \cdot \Pr[X_k = j] \\
&= \sum_{j=0}^{\infty} \mathrm{E}[\sum_{i=1}^{j} \mathrm{Poisson}(2\alpha) \mid X_k = j] \cdot \Pr[X_k = j] \\
&= \sum_{j=0}^{\infty} \mathrm{E}[\sum_{i=1}^{j} \mathrm{Poisson}(2\alpha)] \cdot \Pr[X_k = j] \\
&= \sum_{j=0}^{\infty} \mathrm{E}[\mathrm{Poisson}(2\alpha\ j)] \cdot \Pr[X_k = j] \\
&= \sum_{j=0}^{\infty} 2\alpha\ j \cdot \Pr[X_k = j] \\
&= 2\alpha \cdot \sum_{j=0}^{\infty} j \cdot \Pr[X_k = j] \\
&= 2\alpha \cdot \mathrm{E}[X_k]
\end{aligned}
$$

# The Finishing Touches

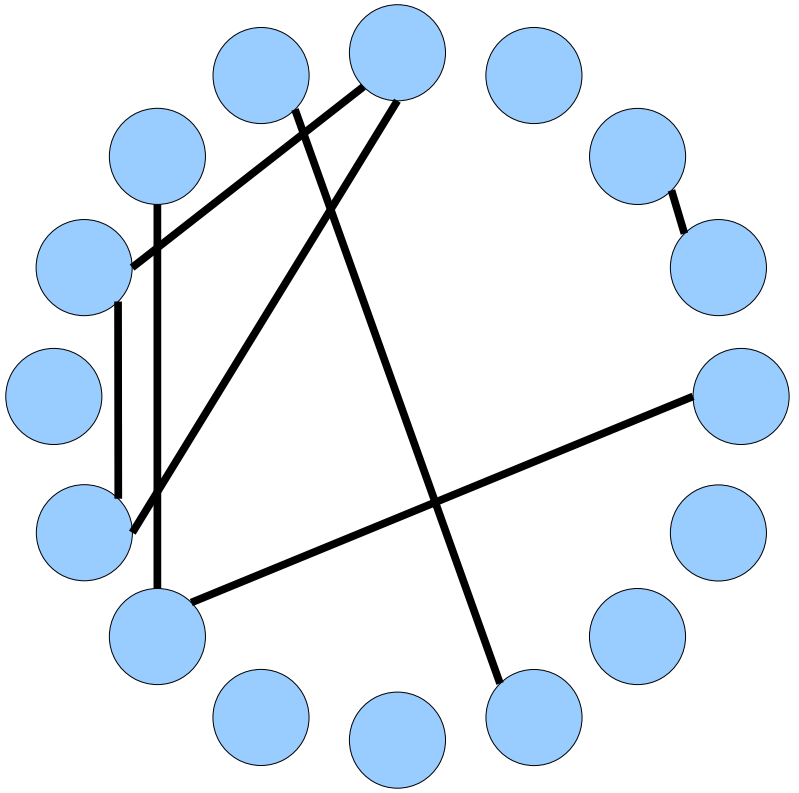
- On expectation, there are $(2\alpha)^k$ nodes in layer $k$ of the BFS tree.
- Summing across all layers, on expectation there are $(1 - 2\alpha)^{-1}$ total nodes in the BFS tree.
- Thus the expected number of nodes in a CC in the cuckoo graph is O(1).
- Therefore, in cuckoo hashing, assuming we set $n = \alpha m$ for some constant $\alpha < ½$, 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!




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 $E / V$?

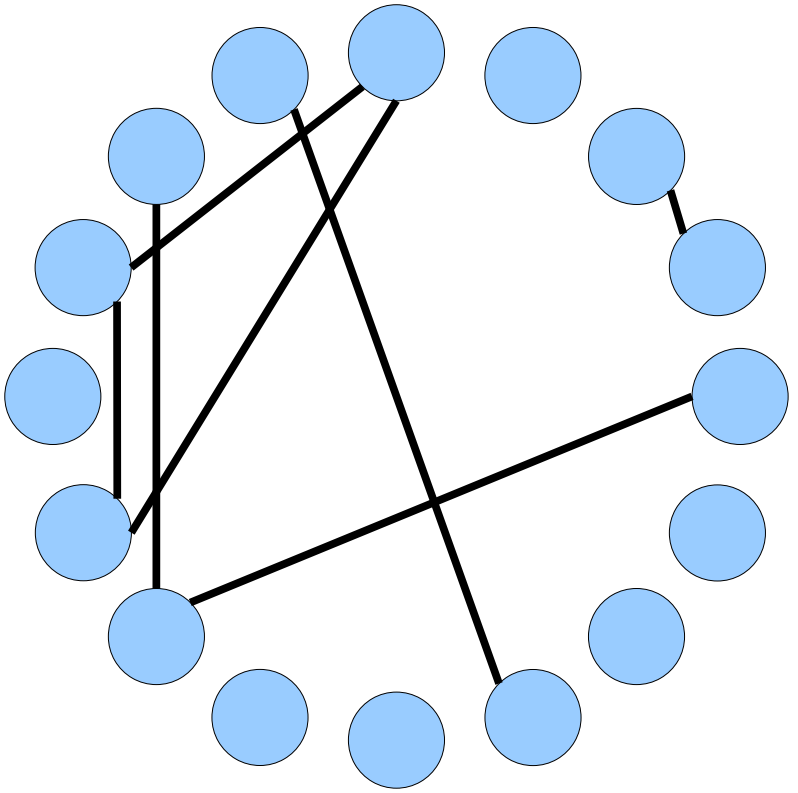
# Random Graph Theory

- ***Theorem:*** Let $E = \alpha V$ for some constant $\alpha < ½$. 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, amortized*

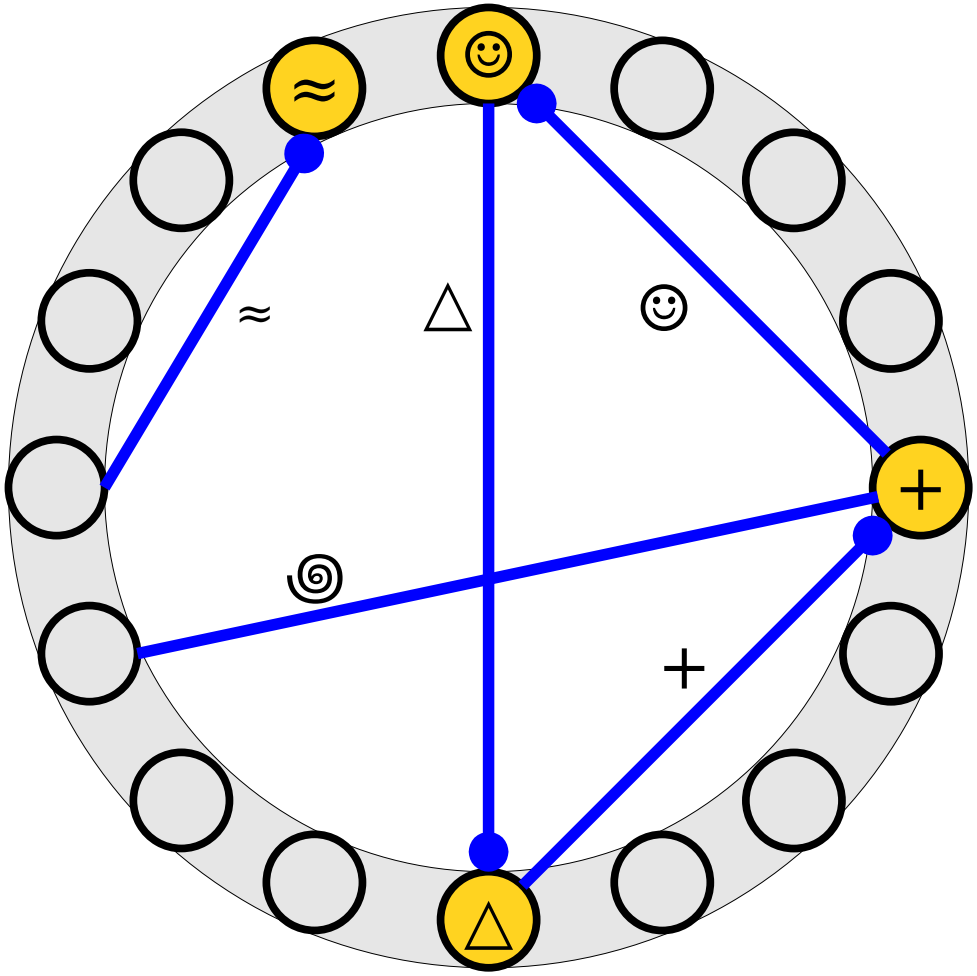
# Improving Our Space Usage

# Improving Space Usage

- A cuckoo hash table with $n$ elements requires a table of size $n / \alpha$, with $\alpha < ½$.
- 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

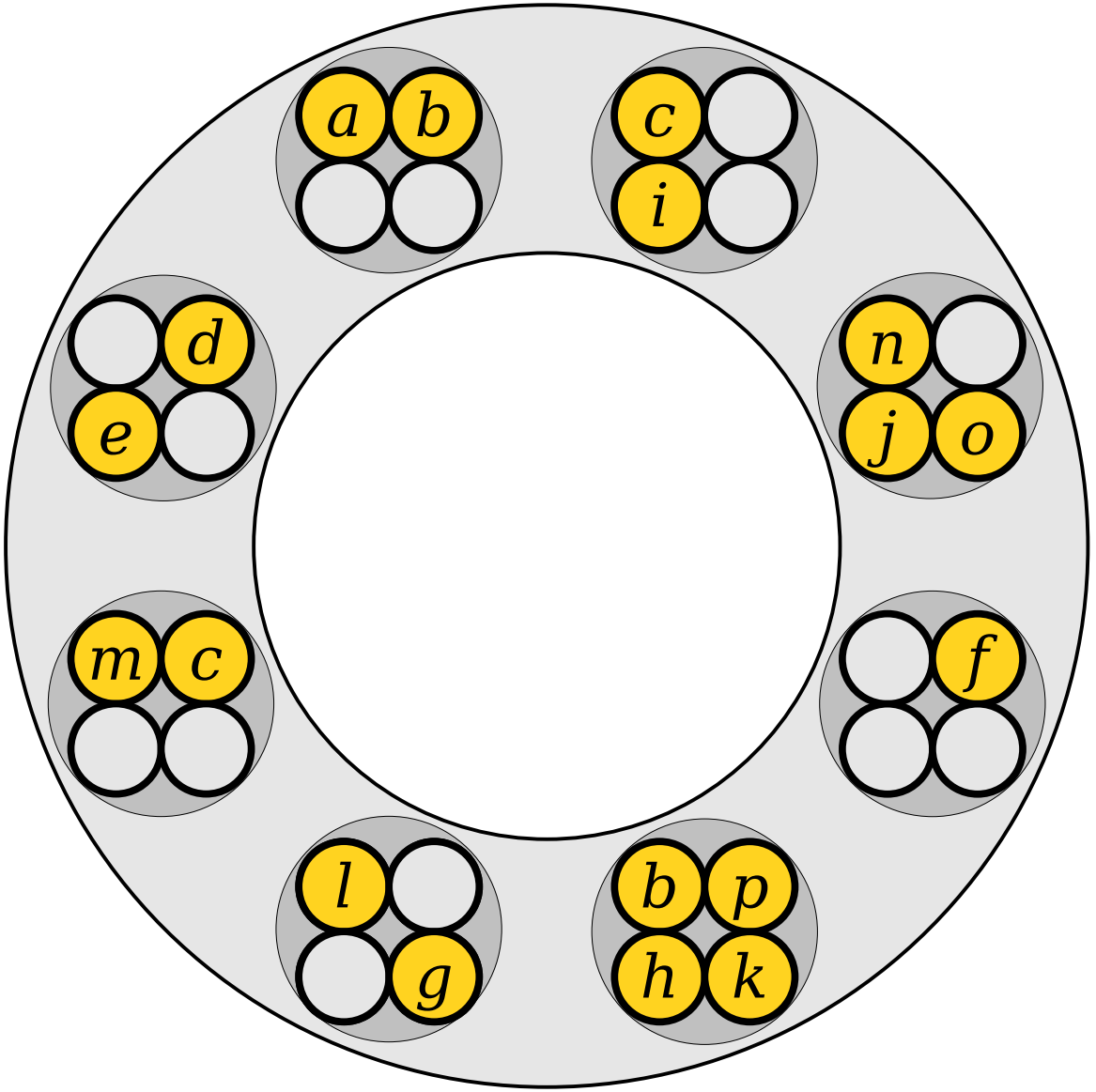
- Our cuckoo graph – and the associated limitations on cuckoo hashing – result from these two assumptions:
  - Each table slot can hold at most one item.
  - Each item can be placed into one of two positions.
- We need to relax these constraints.

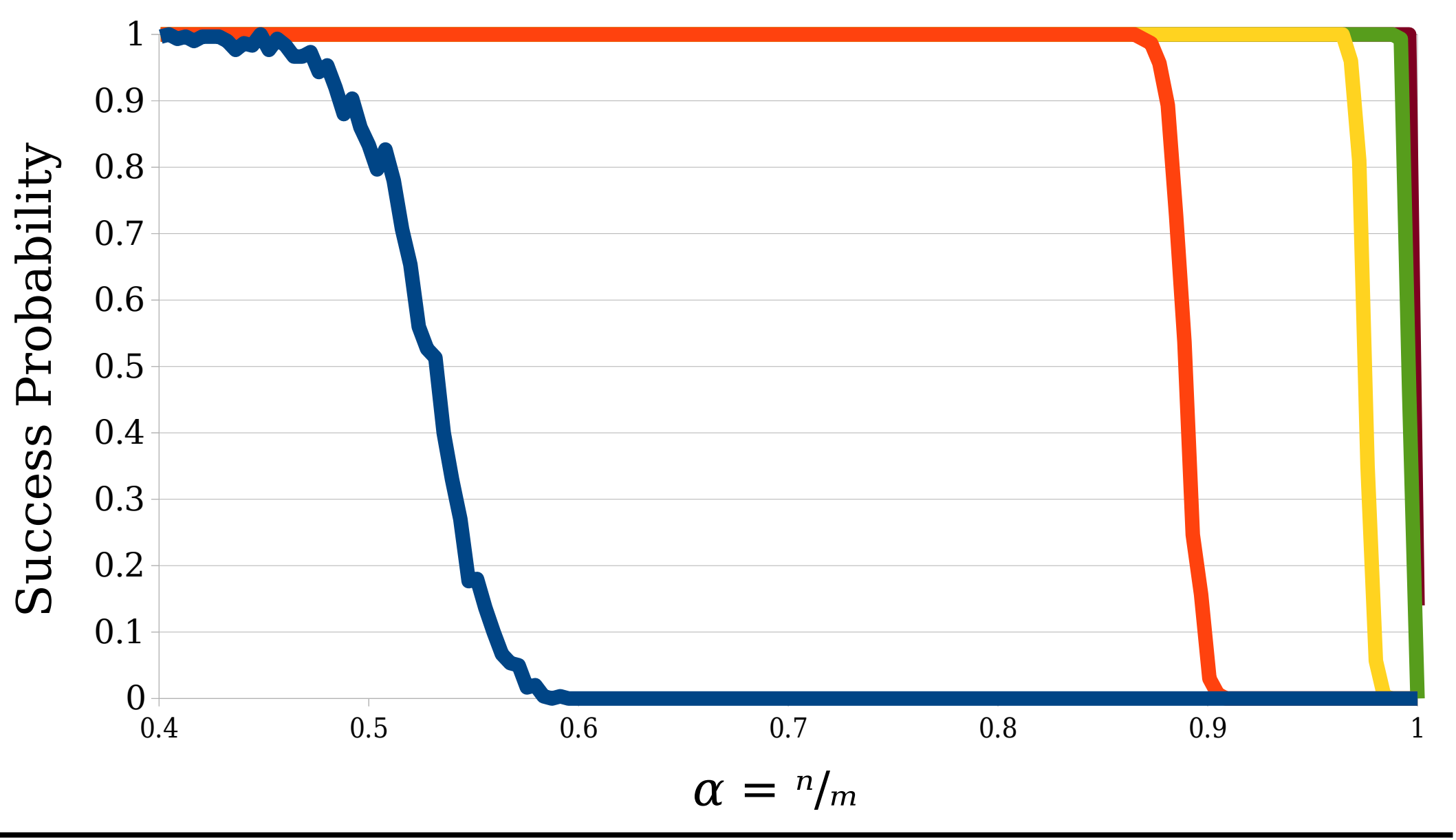


***Idea 1:*** Allow each table slot to store multiple items.

# 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

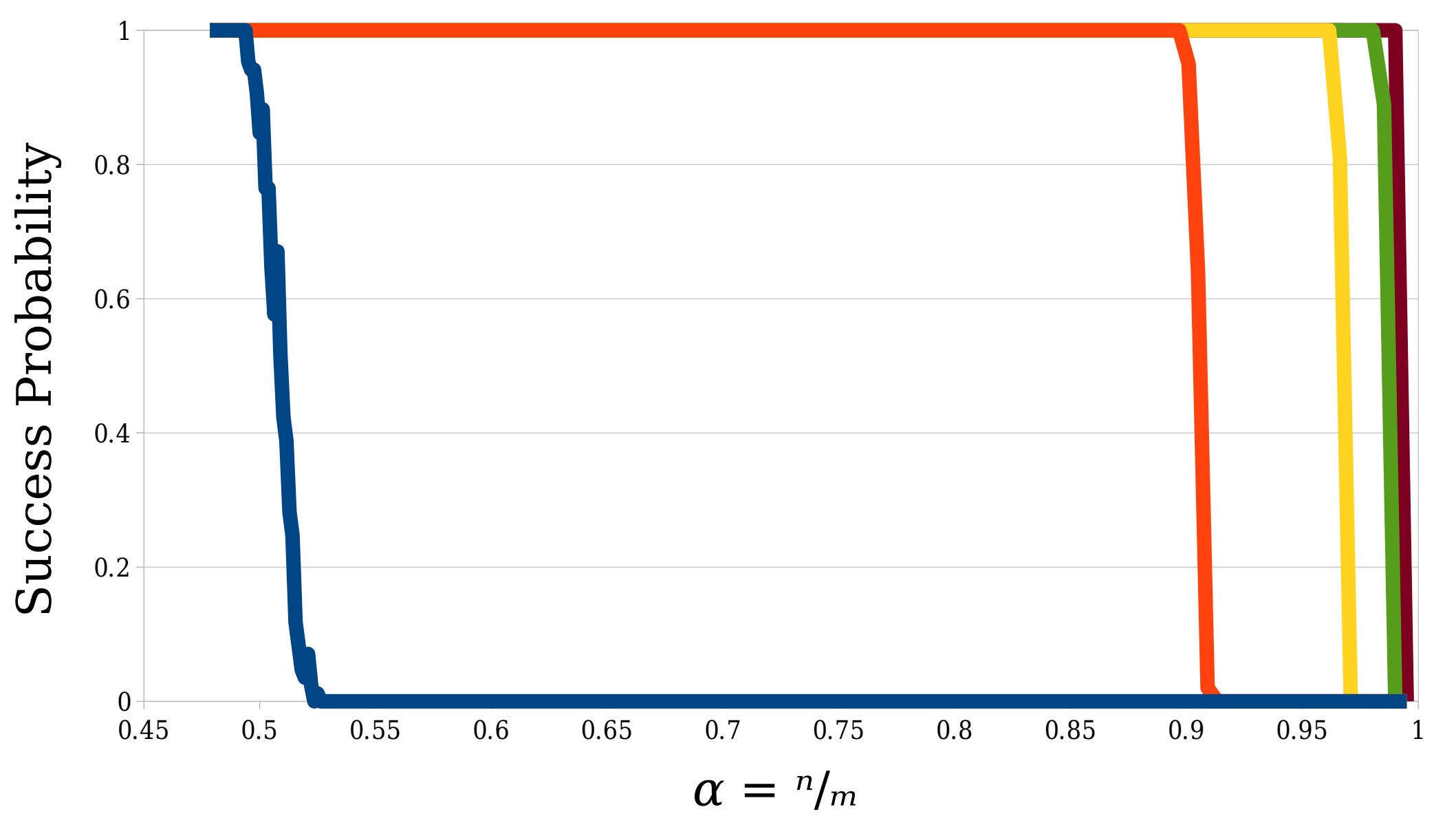
- Suppose we have a table with $m/b$ slots, each of which hold $b$ items. Assume $n = m\alpha$ and we use two hash functions.
- The thresholds given below show the maximum value of α a blocked cuckoo hash table can use.
- As you can see, modest increases to $b$ dramatically increase the space utilization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

***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

- Here are the theoretical limiting load factors of *d*-ary cuckoo hashing.
  - These were worked out over a series of papers in the early 2000s.
- 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

- We now have two options for improving space utilization (blocking, *d*-ary hashing).
- In practice, blocked cuckoo hashing is *much* faster than *d*-ary hashing.
  - Hash functions are fast to evaluate, but not *that* fast.
  - Blocked cuckoo hashing has better locality of reference, especially if the blocks fit into cache lines.
  - You can use vectorized (SIMD) operations to do parallel checks in blocked cuckoo hashing, but not in *d*-ary hashing.
- Good question to ponder: what if you mix and match the approaches?

# 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.

# Next Time

- ***Approximate Membership Queries***
  - Storing a set, approximately.
- ***Data Structure Lower Bounds***
  - Knowing how low we can go.
- ***Bloom Filters***
  - The OG approximate set, which is now obsolete.
- ***Cuckoo Filters***
  - A modern replacement for the Bloom filter.