# Cuckoo Hashing

# Outline for Today

- ***Towards Perfect Hashing***
  - Reducing worst-case bounds
- ***Cuckoo Hashing***
  - Hashing with worst-case O(1) lookups.
- ***Random Graph Theory***
  - Just how fast is cuckoo hashing?

# Perfect Hashing

# Collision Resolution

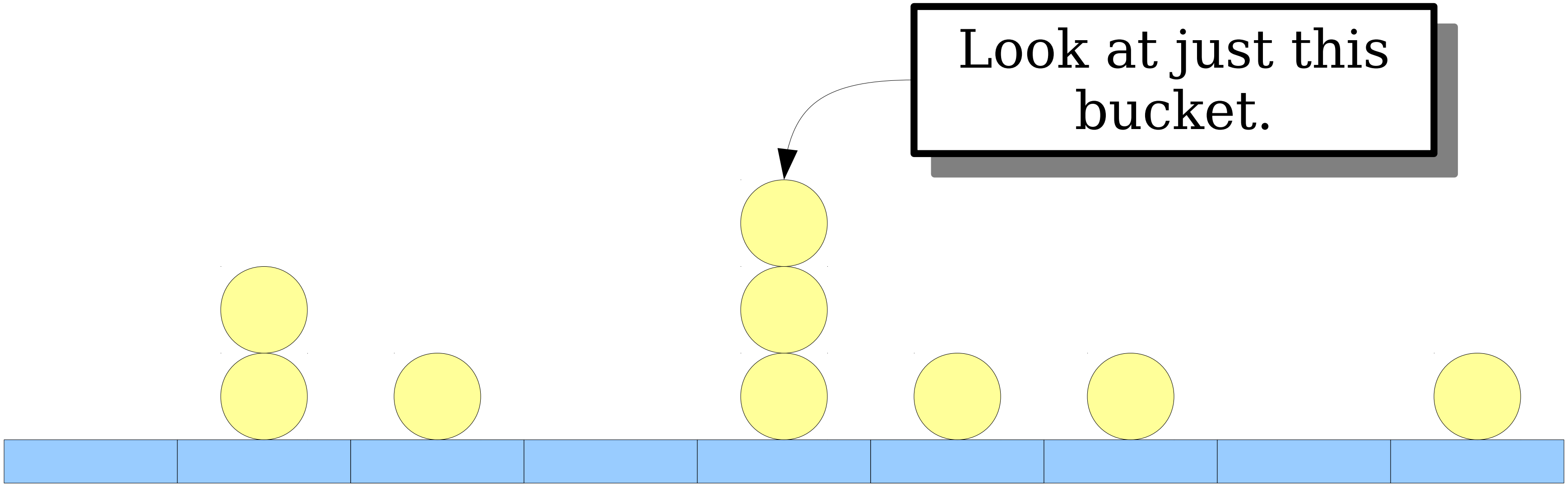
- Last time, we mentioned three general strategies for resolving hash collisions:
  - ***Closed addressing:*** Store all colliding elements in an auxiliary data structure like a linked list or BST.
  - ***Open addressing:*** Allow elements to overflow out of their target bucket and into other spaces.
  - ***Perfect hashing:*** Choose a hash function with no collisions.
- We have not spoken on this last topic yet.

# Why Perfect Hashing is Hard

- For fixed, constant load factors, the expected cost of a lookup in a chained hash table or linear probing table is O(1).
- However, the expected cost of a lookup in these tables is not the same as the expected *worst-case* cost of a lookup in these tables.

# Expected Worst-Case Bounds

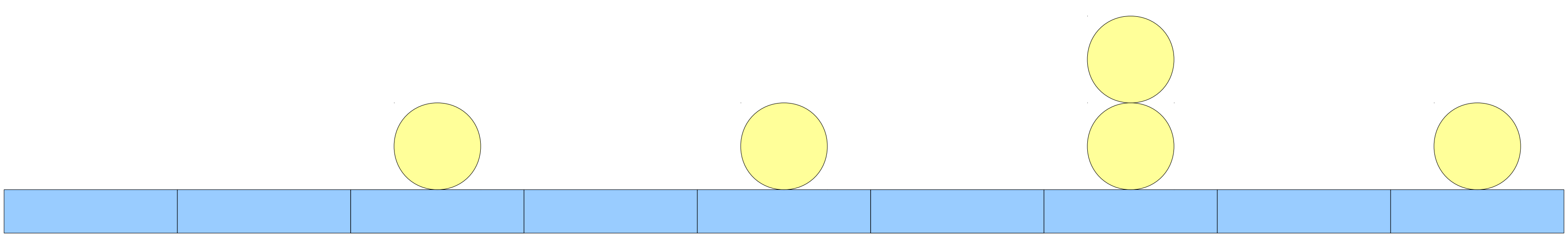
- ***Theorem:*** Assuming truly random hash functions, the expected worst-case cost of a lookup in a chained hash table is $\Theta(\log n / \log \log n)$, assuming the number of slots is $\Theta(n)$.
- ***Theorem:*** Assuming truly random hash functions, the expected worst-case cost of a lookup in a linear probing hash table is $\Omega(\log n)$.
- ***Proofs:*** Exercises 11-1 and 11-2 from CLRS. ☺



## ***Technique 1:*** Multiple-Choice Hashing

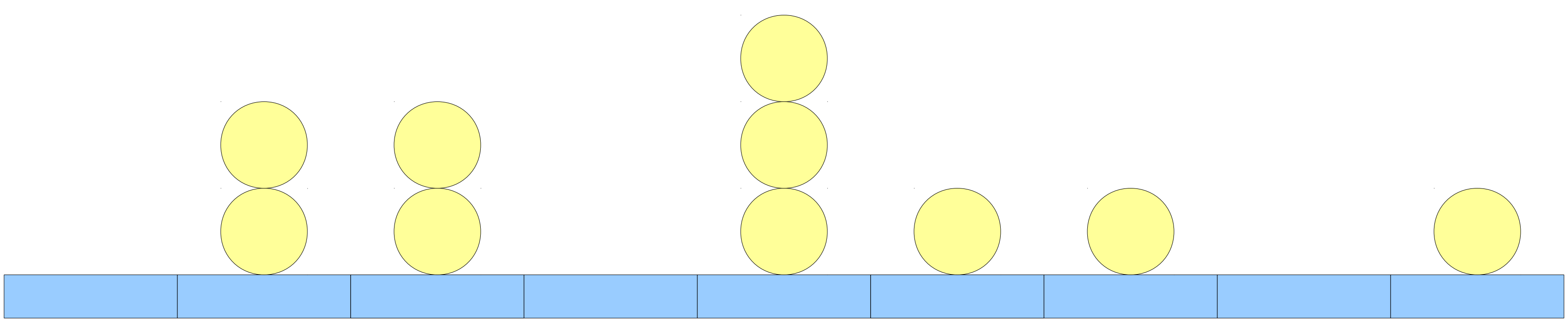
# Second-Choice Hashing

- Suppose that we distribute $n$ balls into $\Theta(n)$ bins using the following strategy:
  - For each ball, choose two bins totally at random.
  - Put the ball into the bin with fewer balls in it; tiebreak randomly.
- ***Theorem:*** The expected value of the maximum number of balls in any urn is $\Theta(\log \log n)$.
- ***Proof:*** Nontrivial; see "Balanced Allocations" by Azar et al.



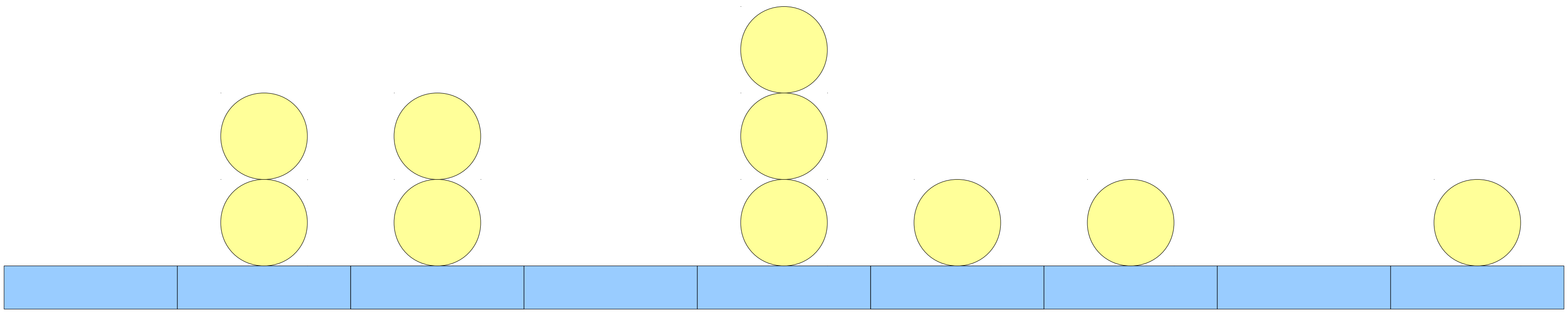
# Second-Choice Hashing

- ***Idea:*** Build a chained hash table with two hash functions $h_1$ and $h_2$.
- To insert an element $x$, compute $h_1(x)$ and $h_2(x)$ and place $x$ into whichever bucket is less full.
- To perform a lookup, compute $h_1(x)$ and $h_2(x)$ and search both buckets for $x$.



# Second-Choice Hashing

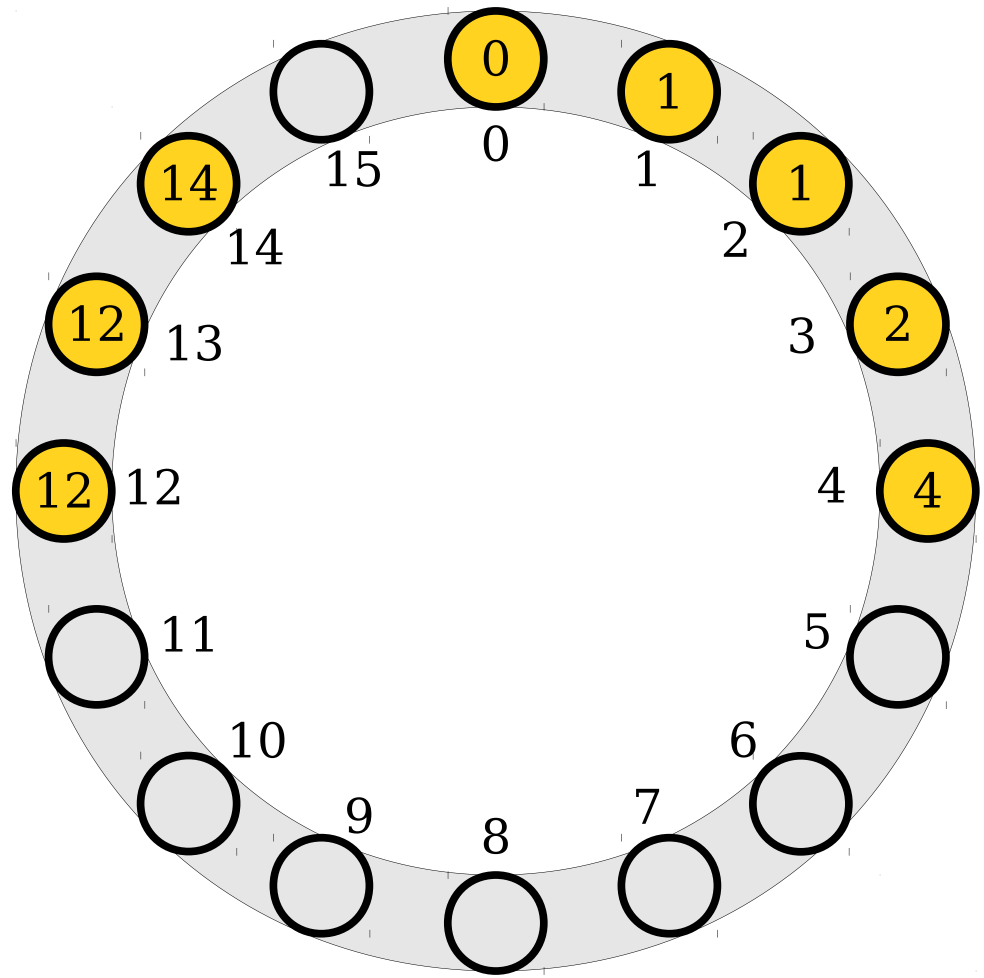
- ***Theorem:*** The expected cost of a lookup in such a hash table is $O(1 + \alpha)$. *(Why?)*
- ***Theorem:*** Assuming truly random hash functions, the expected worst-case cost of a lookup in such a hash table is $O(\log \log n)$. *(Why?)*
- ***Open problem:*** What is the smallest $k$ for which there are $k$-independent hash functions that match the bounds using truly random hash functions?



# ***Technique 2:*** Hashing with Relocation

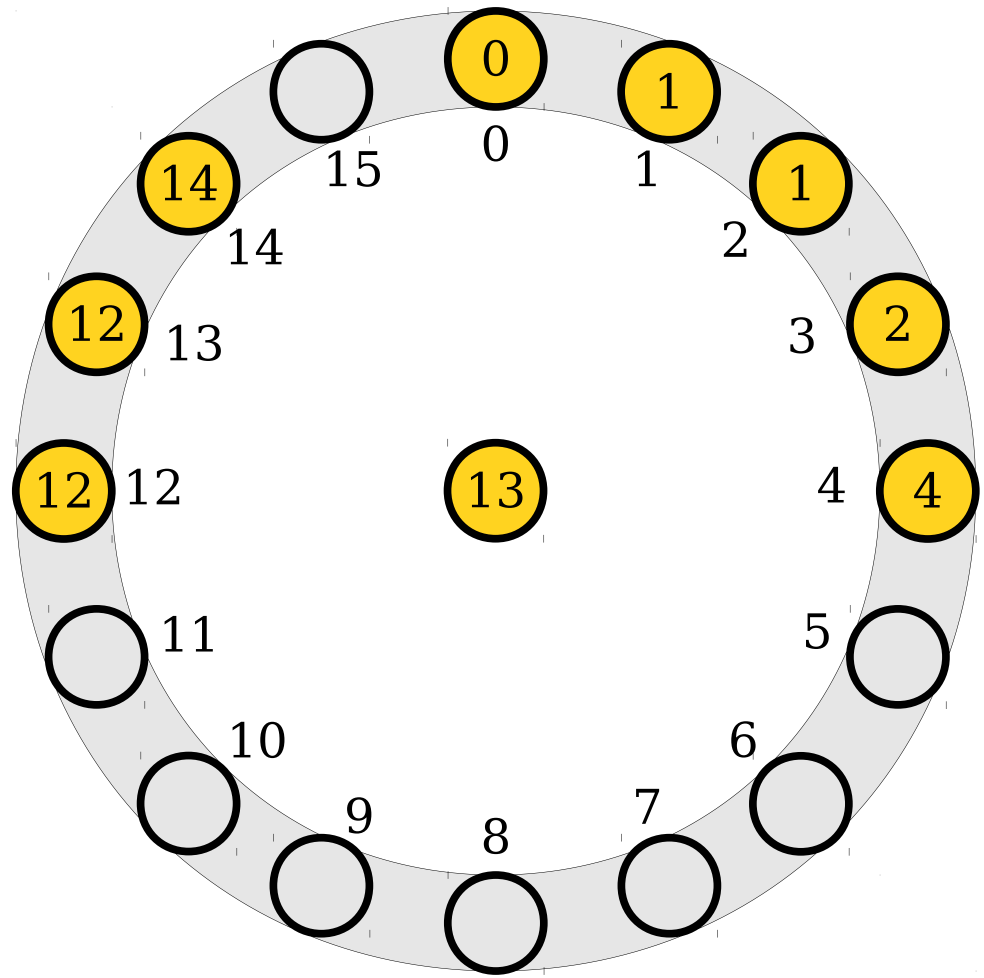
# Robin Hood Hashing

- ***Robin Hood hashing*** is a variation of open addressing where keys can be moved after they're placed.
- When an existing key is found during an insertion that's closer to its “home” location than the new key, it's displaced to make room for it.
- This dramatically decreases the variance in the expected number of lookups.
- It also makes it possible to terminate searches early.



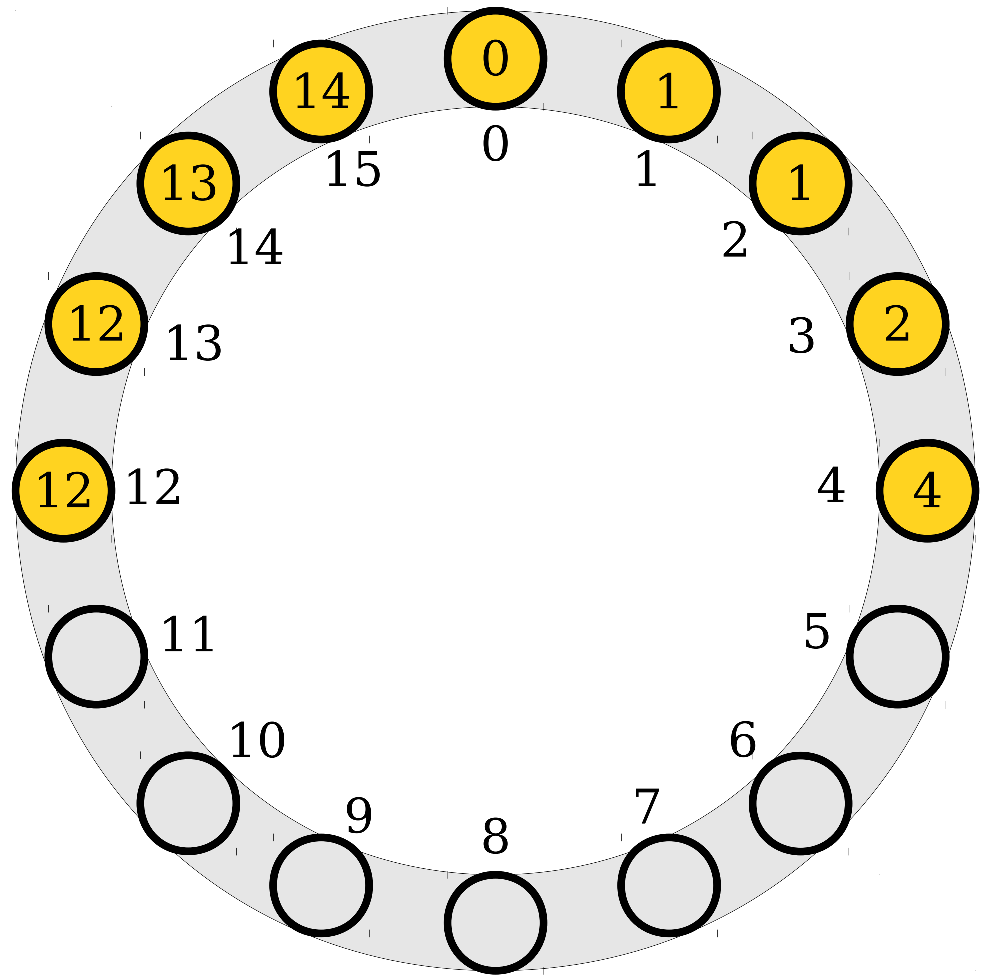
# Robin Hood Hashing

- ***Robin Hood hashing*** is a variation of open addressing where keys can be moved after they're placed.
- When an existing key is found during an insertion that's closer to its "home" location than the new key, it's displaced to make room for it.
- This dramatically decreases the variance in the expected number of lookups.
- It also makes it possible to terminate searches early.



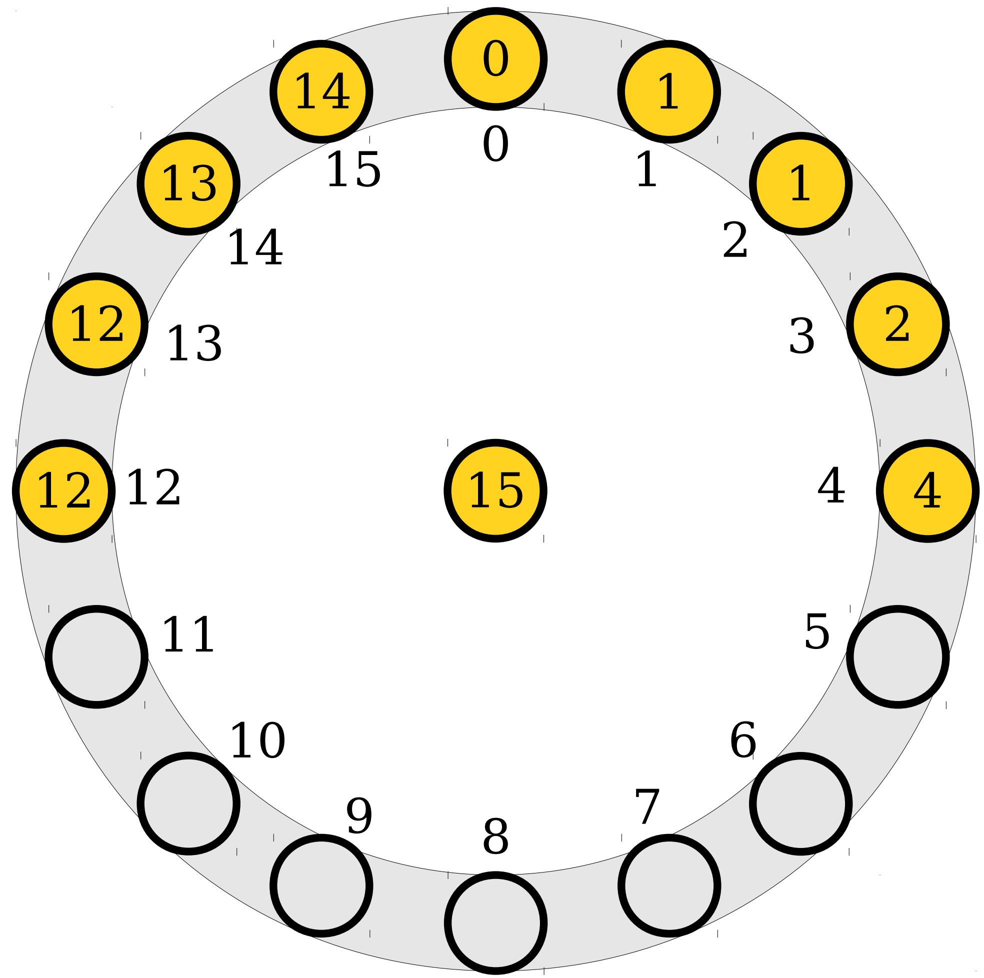
# Robin Hood Hashing

- ***Robin Hood hashing*** is a variation of open addressing where keys can be moved after they're placed.
- When an existing key is found during an insertion that's closer to its “home” location than the new key, it's displaced to make room for it.
- This dramatically decreases the variance in the expected number of lookups.
- It also makes it possible to terminate searches early.



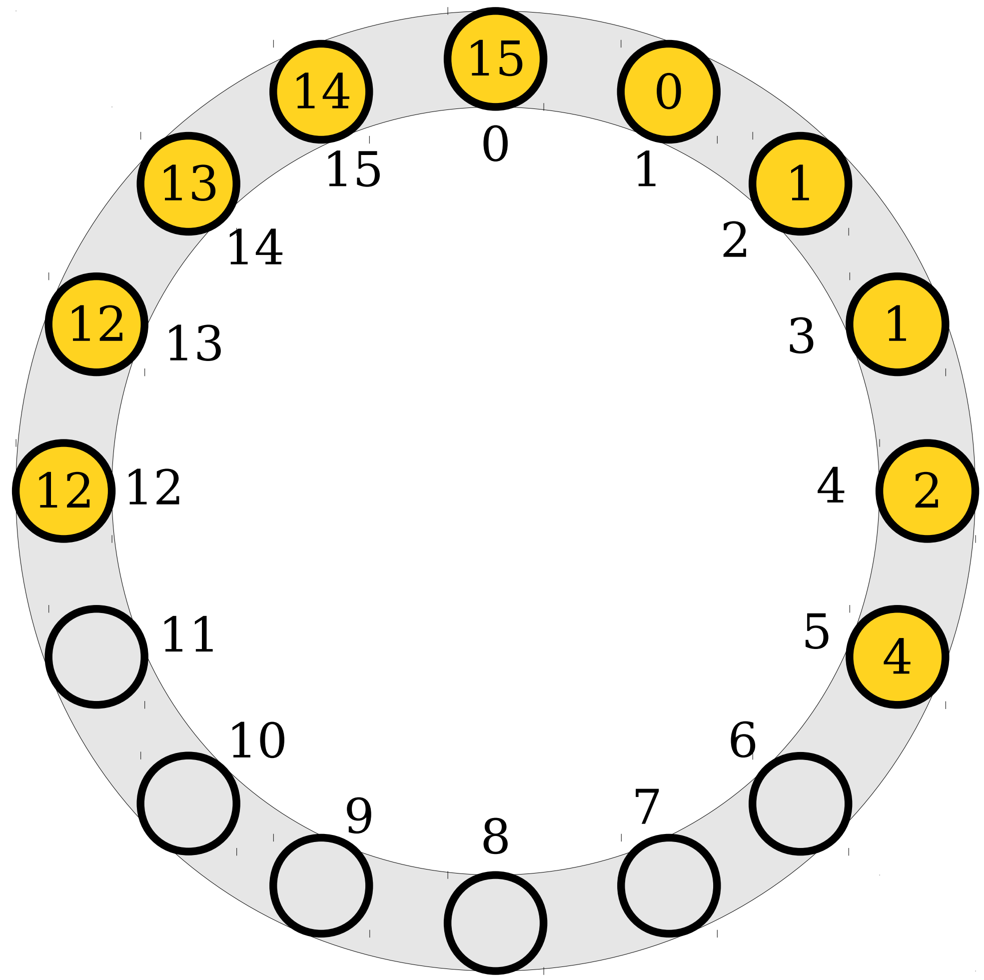
# Robin Hood Hashing

- ***Robin Hood hashing*** is a variation of open addressing where keys can be moved after they're placed.
- When an existing key is found during an insertion that's closer to its “home” location than the new key, it's displaced to make room for it.
- This dramatically decreases the variance in the expected number of lookups.
- It also makes it possible to terminate searches early.



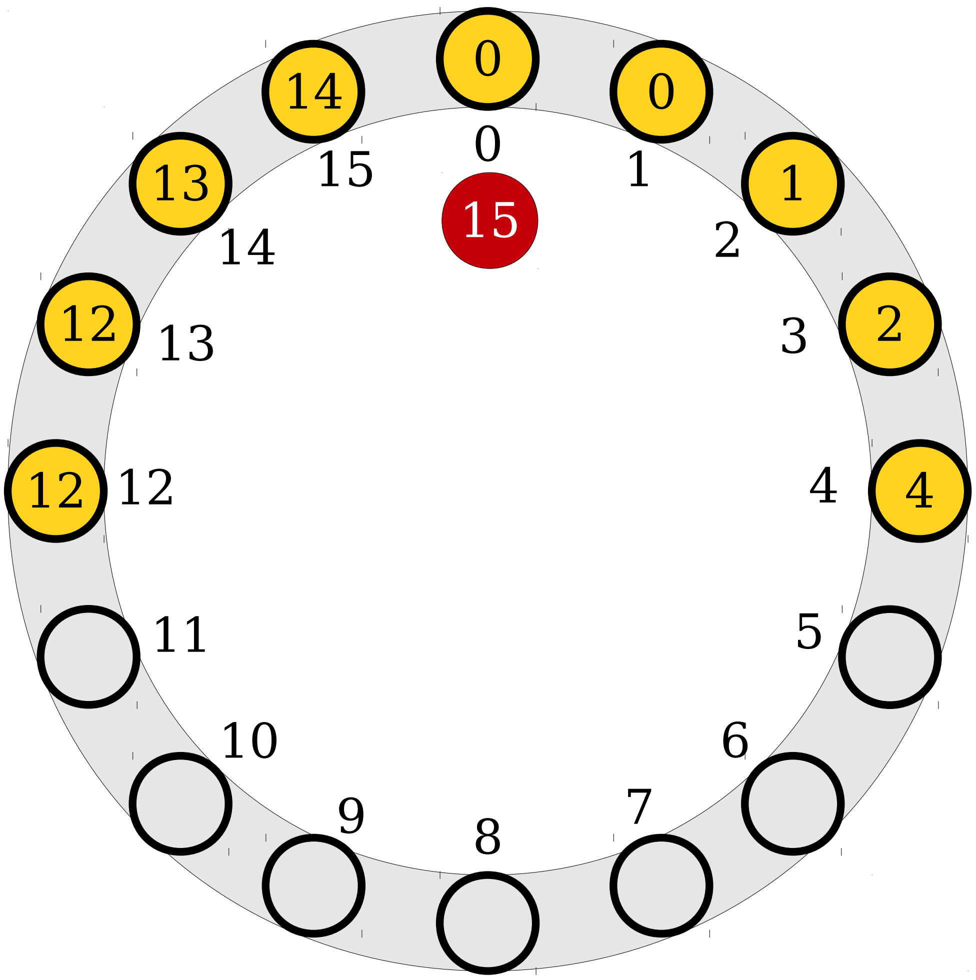
# Robin Hood Hashing

- ***Robin Hood hashing*** is a variation of open addressing where keys can be moved after they're placed.
- When an existing key is found during an insertion that's closer to its “home” location than the new key, it's displaced to make room for it.
- This dramatically decreases the variance in the expected number of lookups.
- It also makes it possible to terminate searches early.



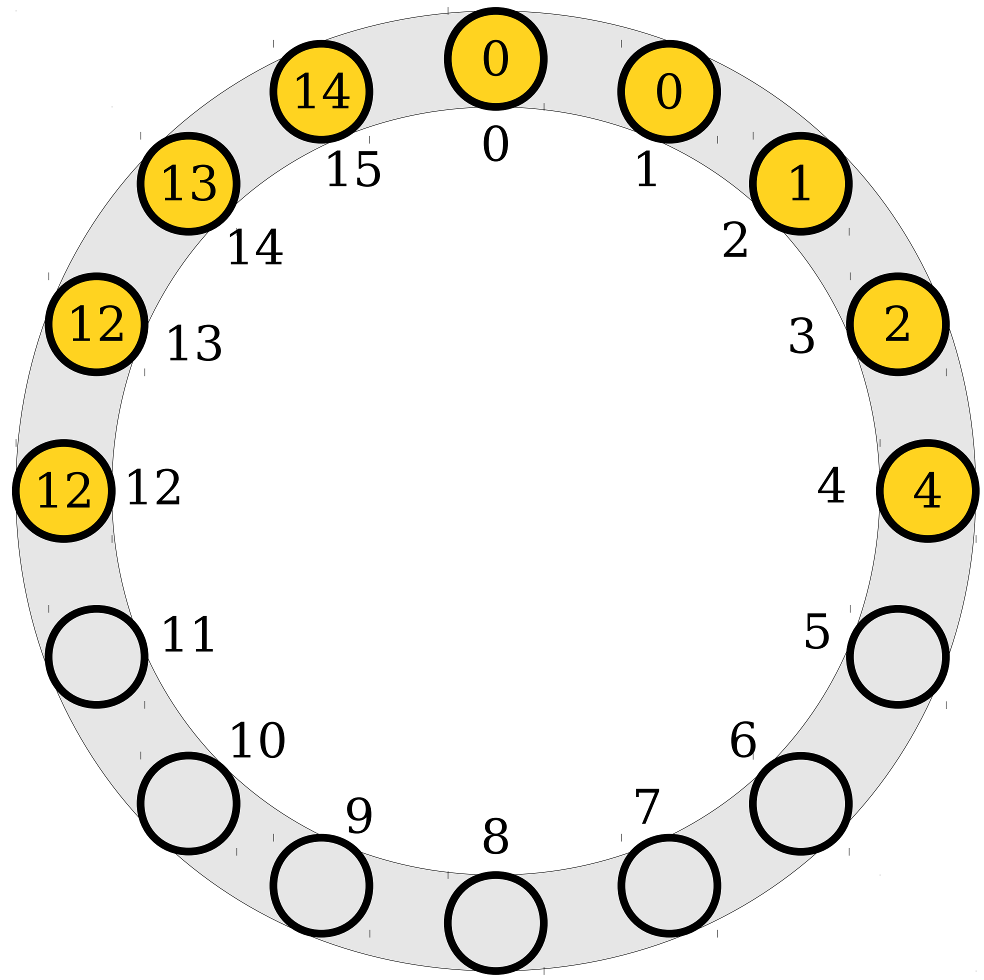
# Robin Hood Hashing

- ***Robin Hood hashing*** is a variation of open addressing where keys can be moved after they're placed.
- When an existing key is found during an insertion that's closer to its “home” location than the new key, it's displaced to make room for it.
- This dramatically decreases the variance in the expected number of lookups.
- It also makes it possible to terminate searches early.



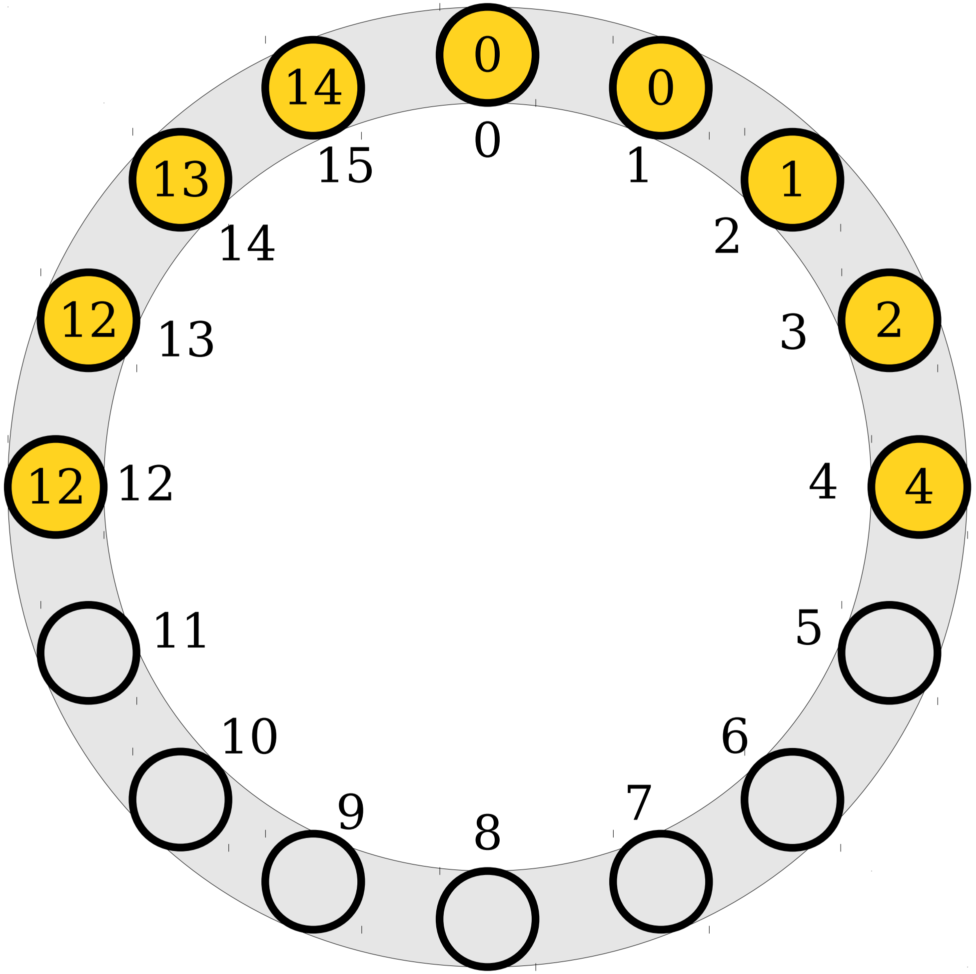
# Robin Hood Hashing

- ***Theorem:*** The expected cost of a lookup in Robin Hood hashing, using 5-independent hashing, is O(1), assuming a constant load factor.
- ***Proof idea:*** Each element is hashed into the same run as it would have been hashed to in linear probing.



# Robin Hood Hashing

- ***Theorem:*** Assuming truly random hash functions, the variance of the expected number of probes required in Robin Hood hashing is O(log log $n$).
- ***Proof:*** Tricky; see Celis' Ph.D thesis.



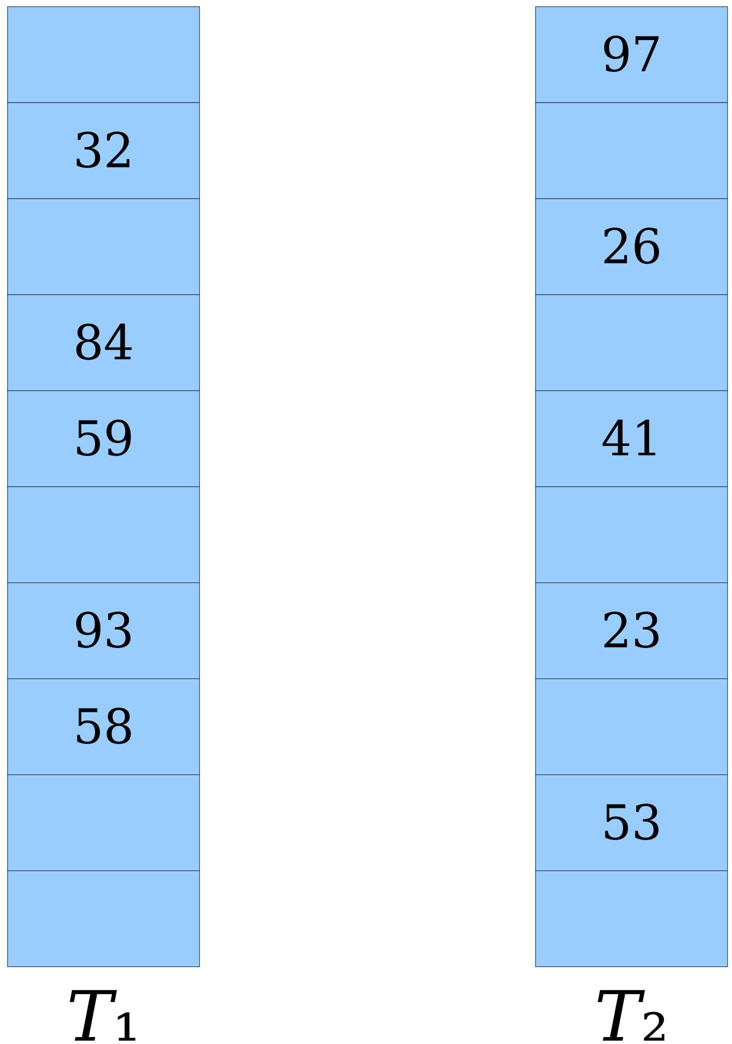
# Where We Stand

- We now have two interesting ideas for improving performance:
  - ***Second-choice hashing:*** Give each element multiple homes to pick from.
  - ***Relocation hashing:*** Move elements after placing them.
- Each idea, individually, exponentially decreases the worst-case cost of a lookup by decreasing the variance in the element distribution.
- What happens if we combine these ideas together?

# Cuckoo Hashing

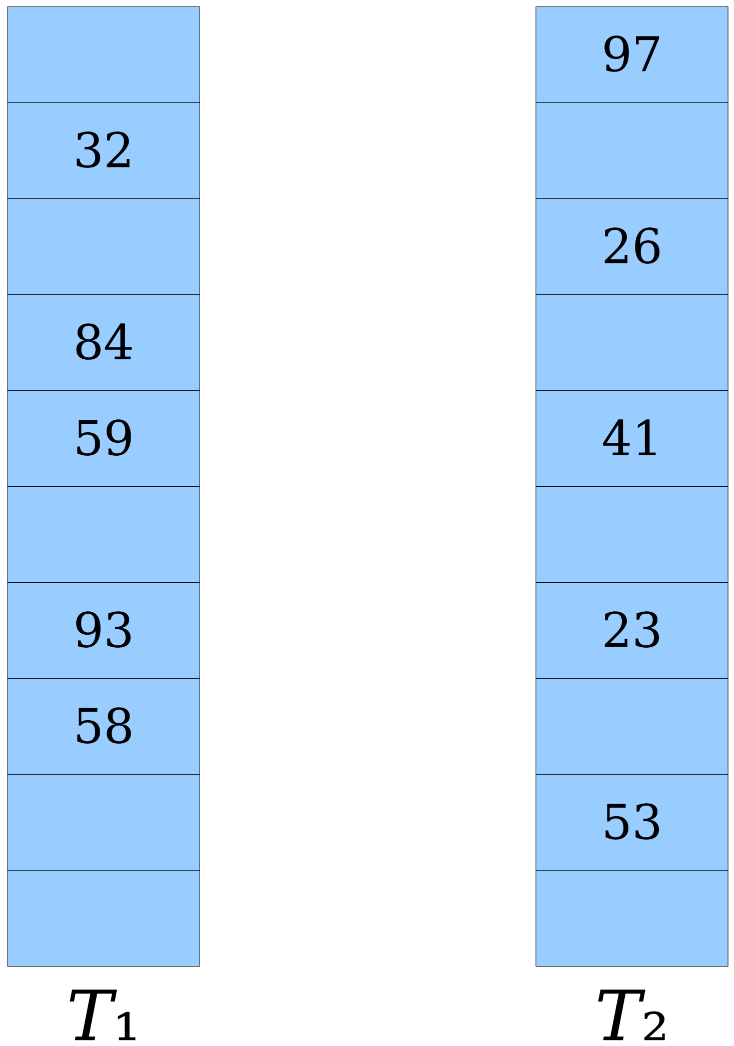
# Cuckoo Hashing

- Maintain two tables, each of which has $m$ elements.
- We choose two hash functions $h_1$ and $h_2$ from $\mathscr{U}$ to $[m]$.
- Every element $x \in \mathscr{U}$ will either be at position $h_1(x)$ in the first table or $h_2(x)$ in the second.



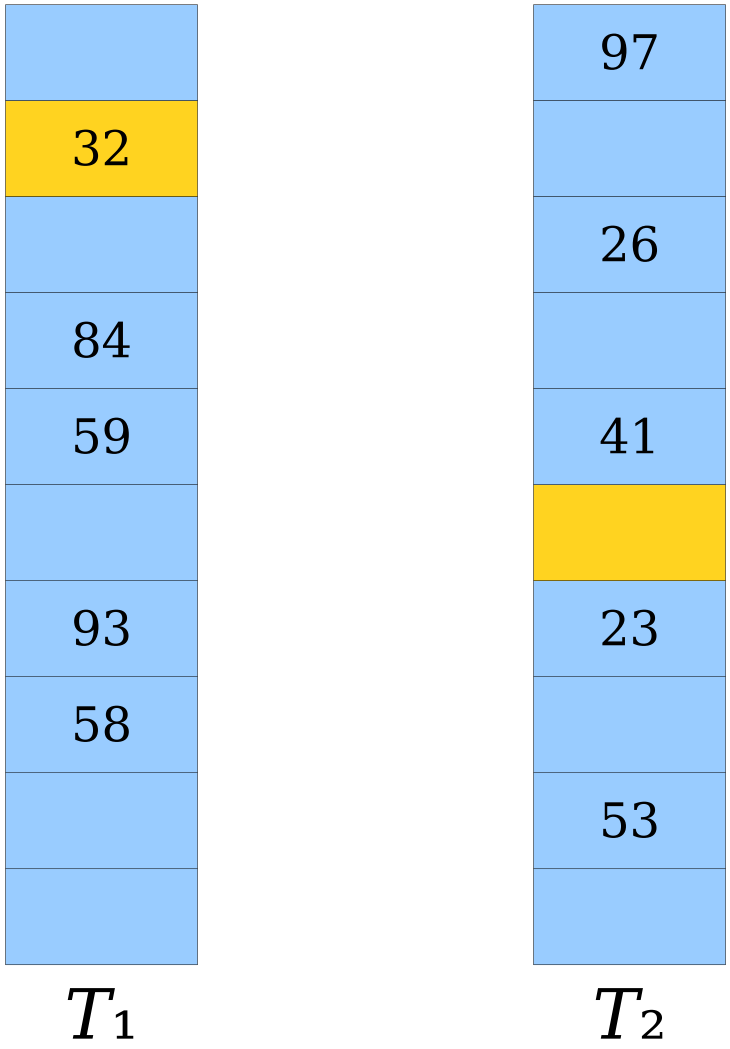
# Cuckoo Hashing

- Lookups take *worst-case* time O(1) because only two locations must be checked.



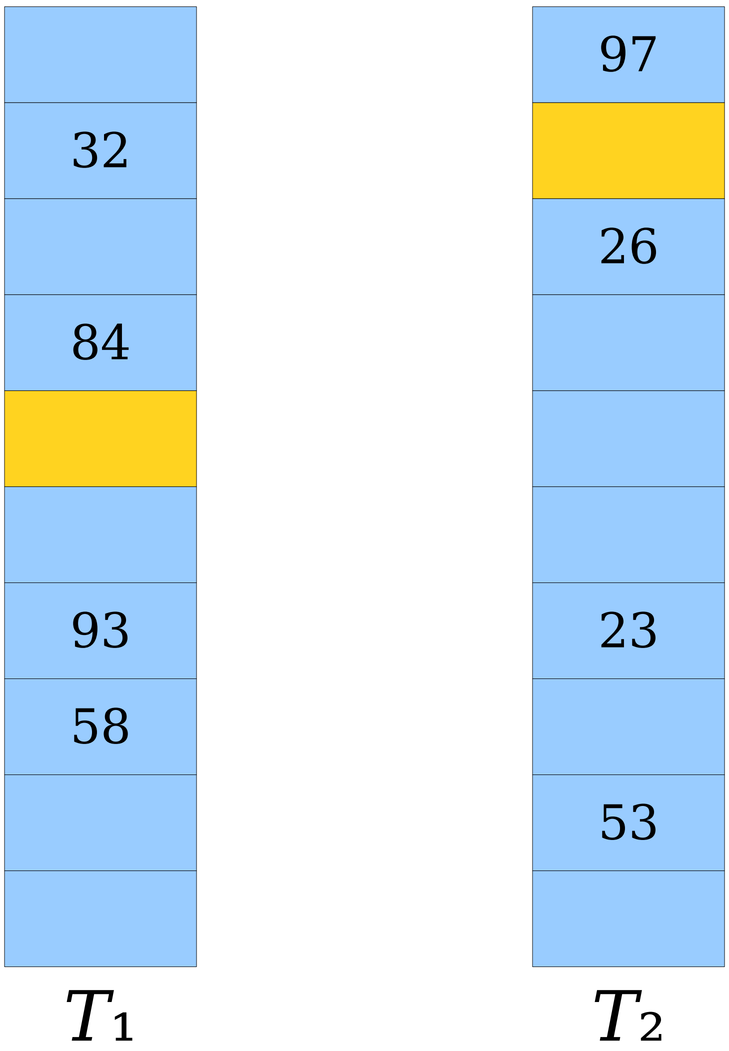
# Cuckoo Hashing

- Lookups take *worst-case* time O(1) because only two locations must be checked.



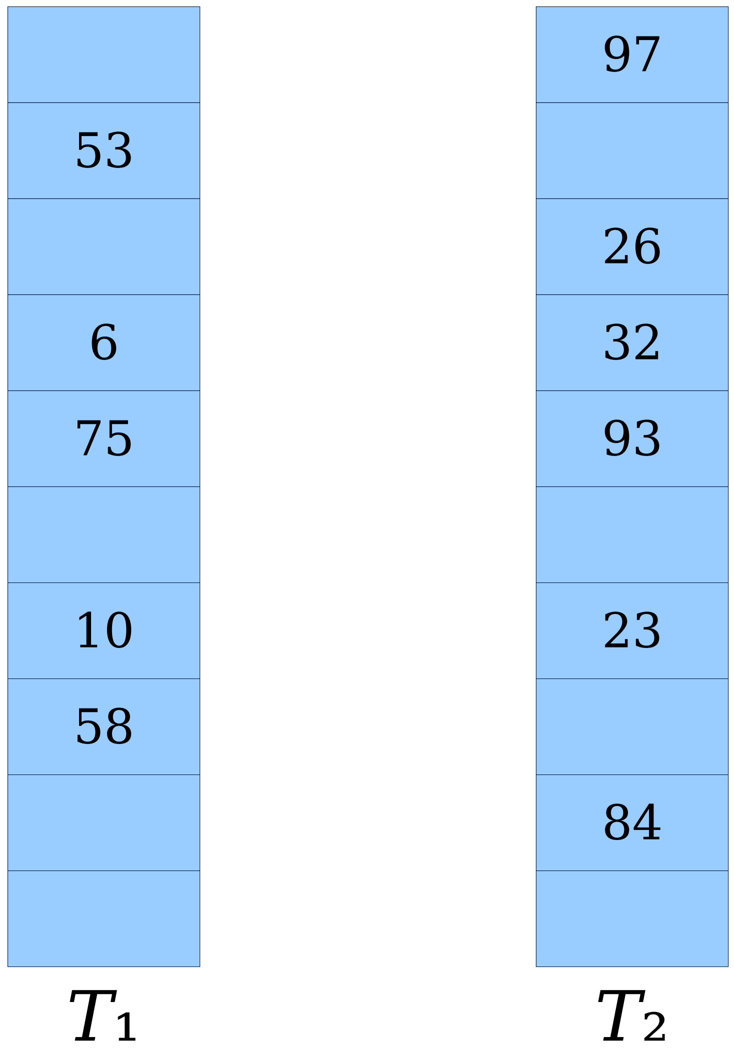
# Cuckoo Hashing

- Lookups take *worst-case* time O(1) because only two locations must be checked.
- Deletions take *worst-case* time O(1) because only two locations must be checked.



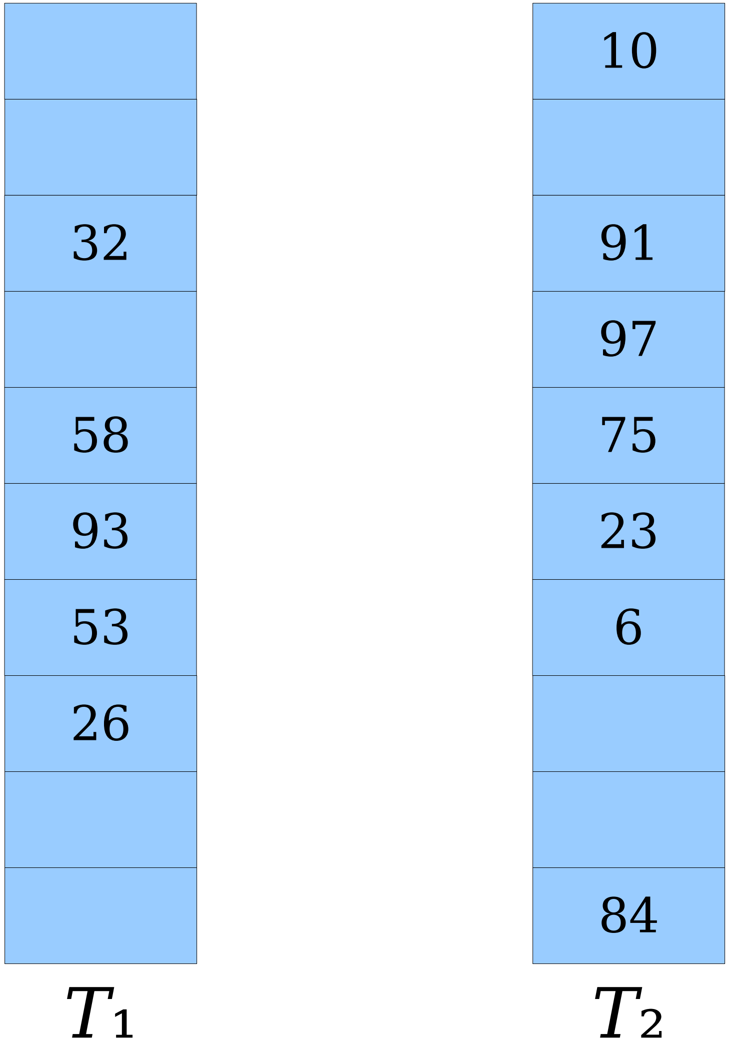
# Cuckoo Hashing

- To insert an element $x$, start by inserting it into table 1.
- If $h_1(x)$ is empty, place $x$ there.
- Otherwise, place $x$ there, evict the old element $y$, and try placing $y$ into table 2.
- Repeat this process, bouncing between tables, until all elements stabilize.



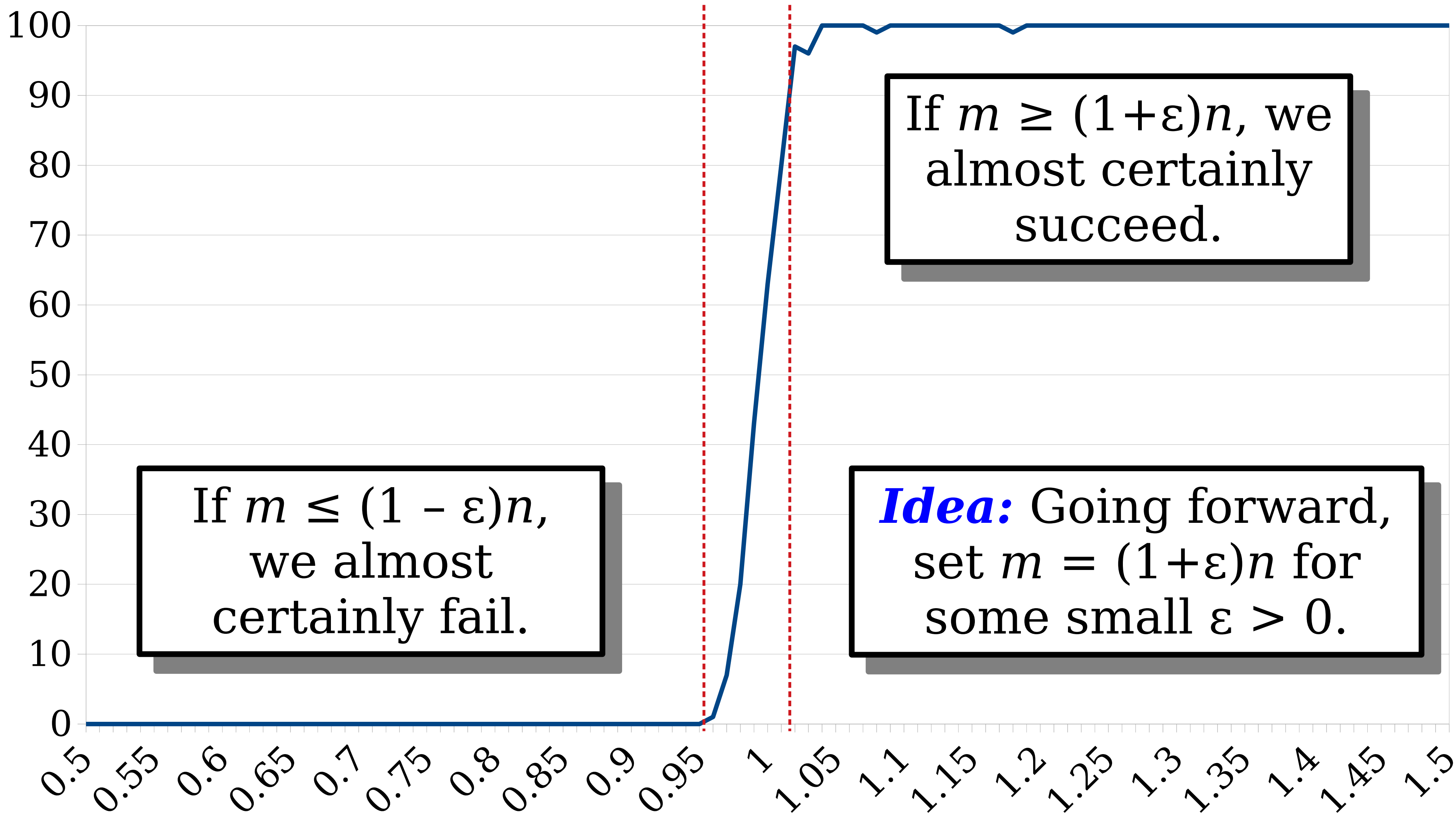
# Cuckoo Hashing

- 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 tables.
- Multiple rehashes might be necessary before this succeeds.



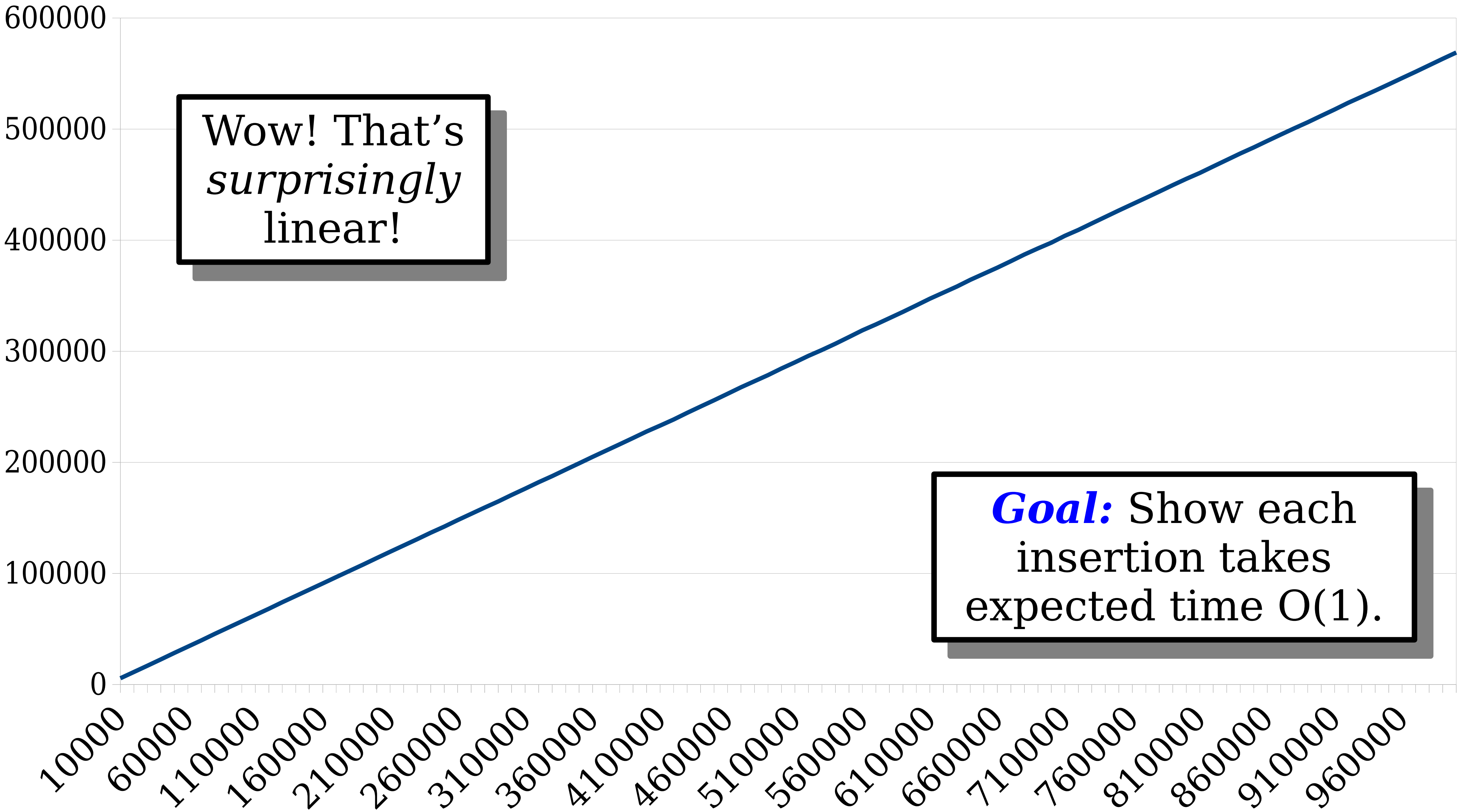
# How efficient is cuckoo hashing?

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




Suppose we store $n$ total elements in two tables of $m$ slots each.

What's probability all insertions succeed, assuming $m = \alpha n$?



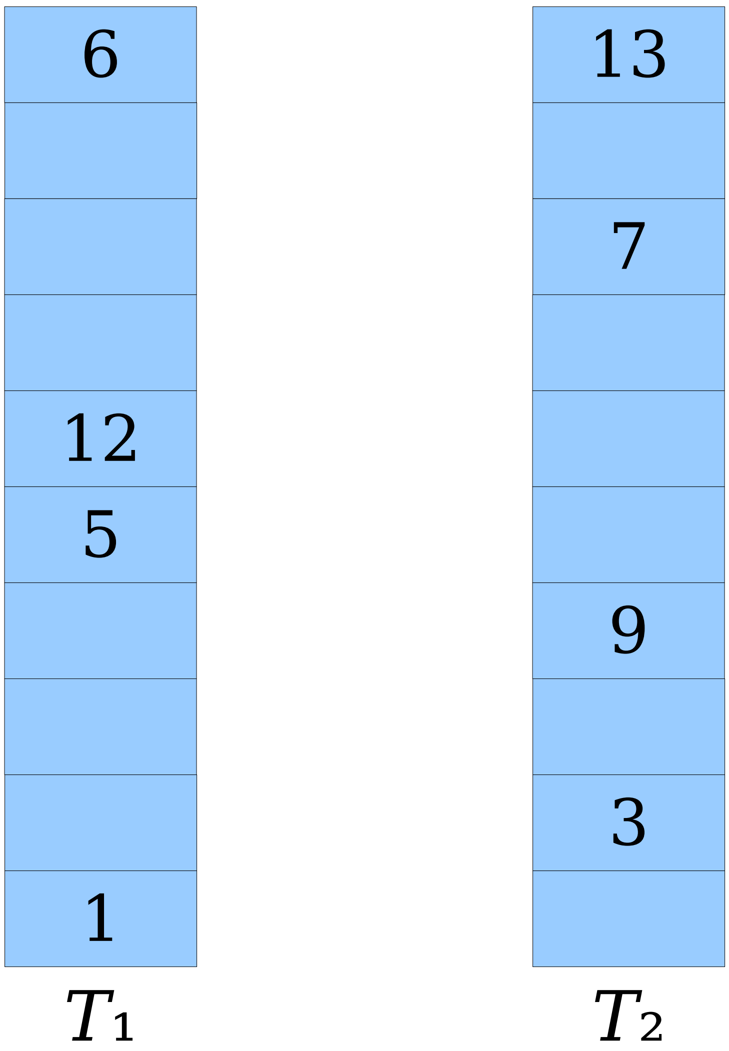

Suppose we store $n$ total elements with $m = (1+\varepsilon)n$.

How many total displacements occur across all insertions?

***Goal:*** Show that insertions take expected time O(1), under the assumption that $m = (1+\varepsilon)n$ for some $\varepsilon > 0$.

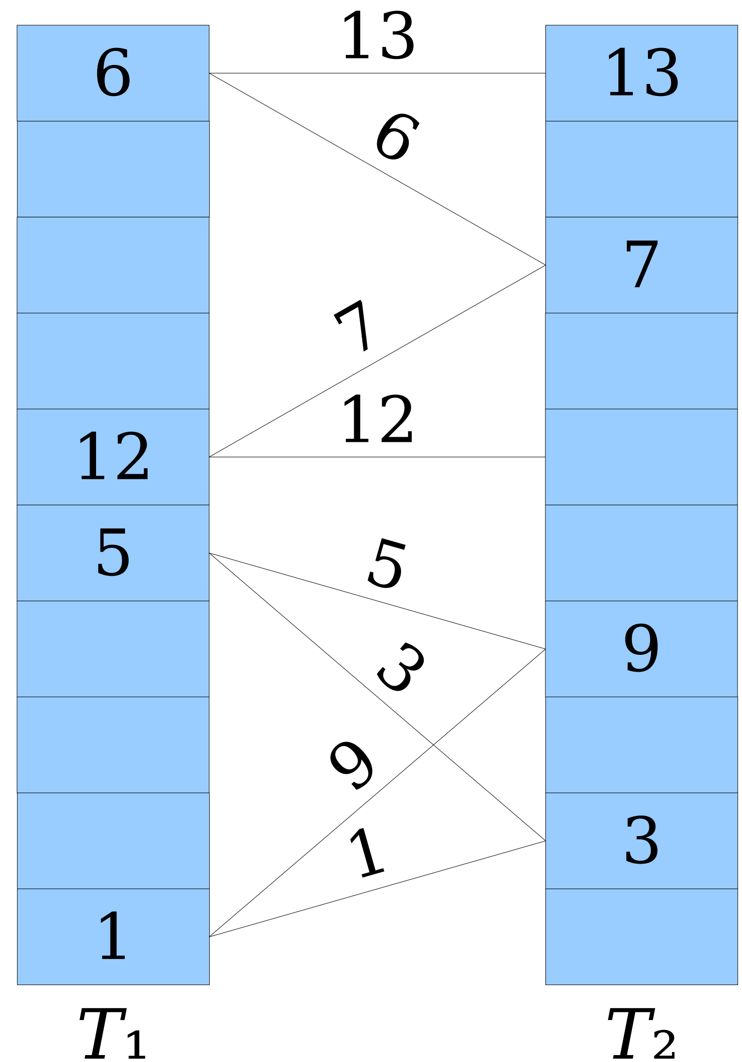
# Analyzing Cuckoo Hashing

- The analysis of cuckoo hashing presents some challenges.
- ***Challenge 1:*** We may have to consider hash collisions across multiple hash functions.
- ***Challenge 2:*** We need to reason about chains of displacement, which can be fairly complicated.
- To resolve these challenges, we'll need to bring in some new techniques.

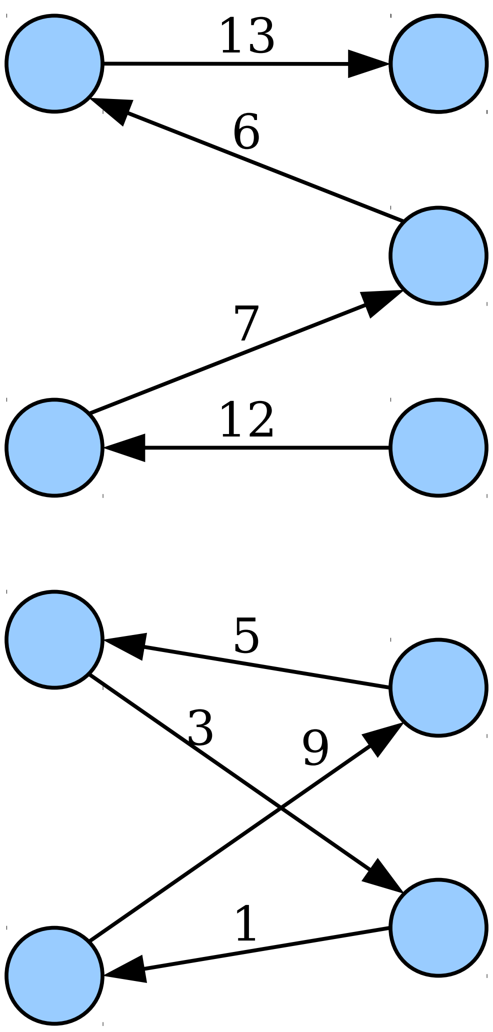


# The Cuckoo Graph

- The ***cuckoo graph*** is a bipartite multigraph derived from a cuckoo hash table.
- Each table slot is a node.
- Each element is an edge.
- Edges link slots where each element can be.
- Each insertion introduces a new edge into the graph.

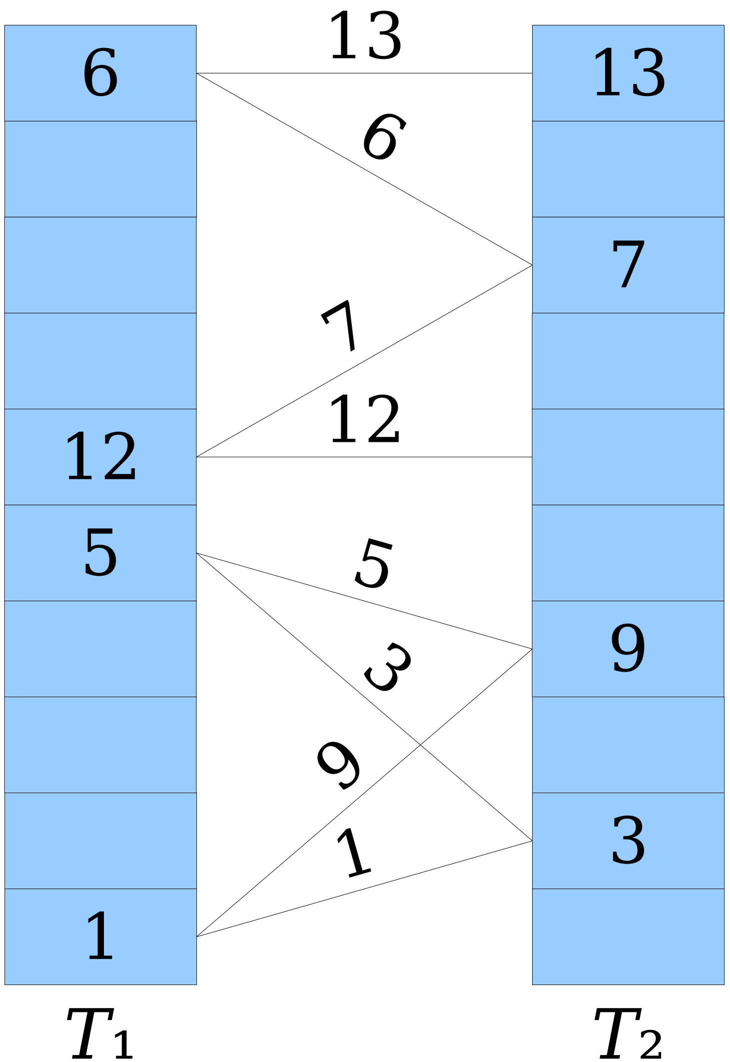


# The Cuckoo Graph

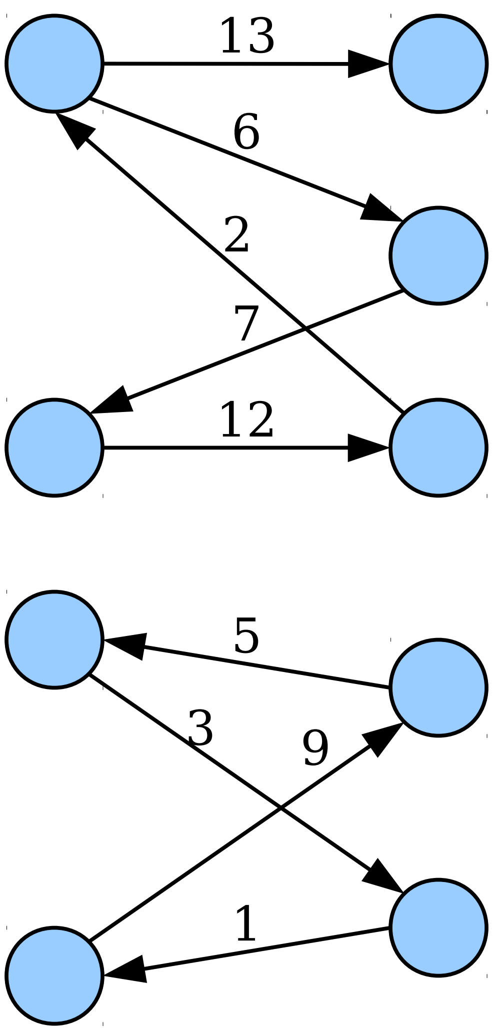



Arrowheads indicate which slots elements is stored in.

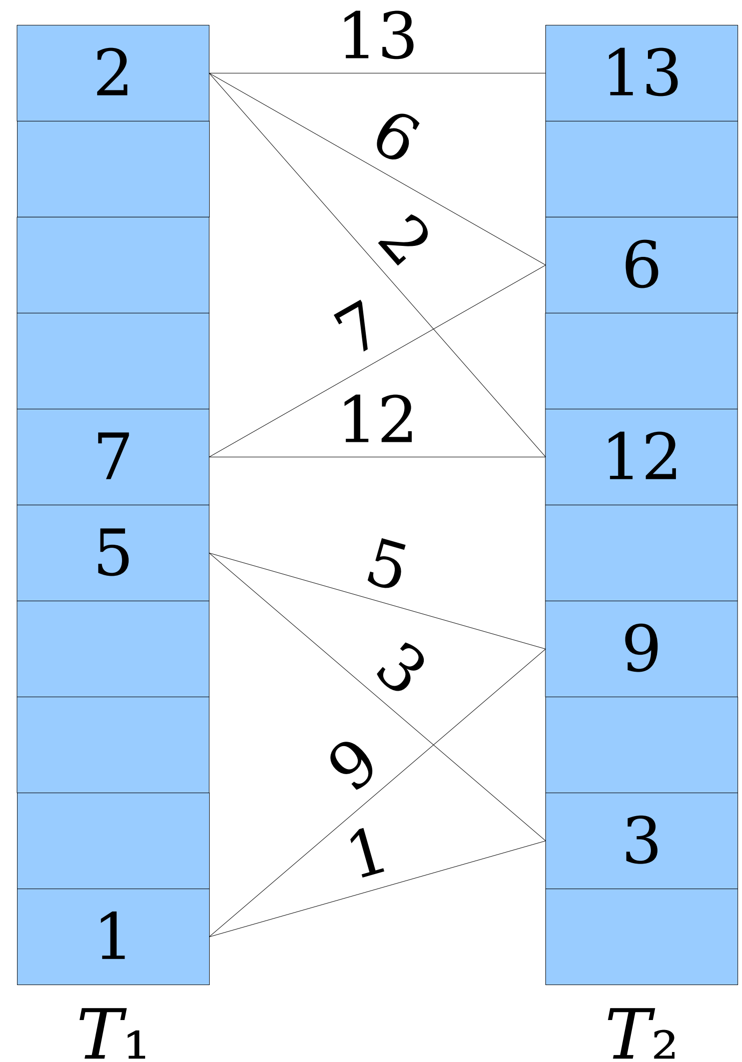
Each node has at most one incoming edge.



# The Cuckoo Graph

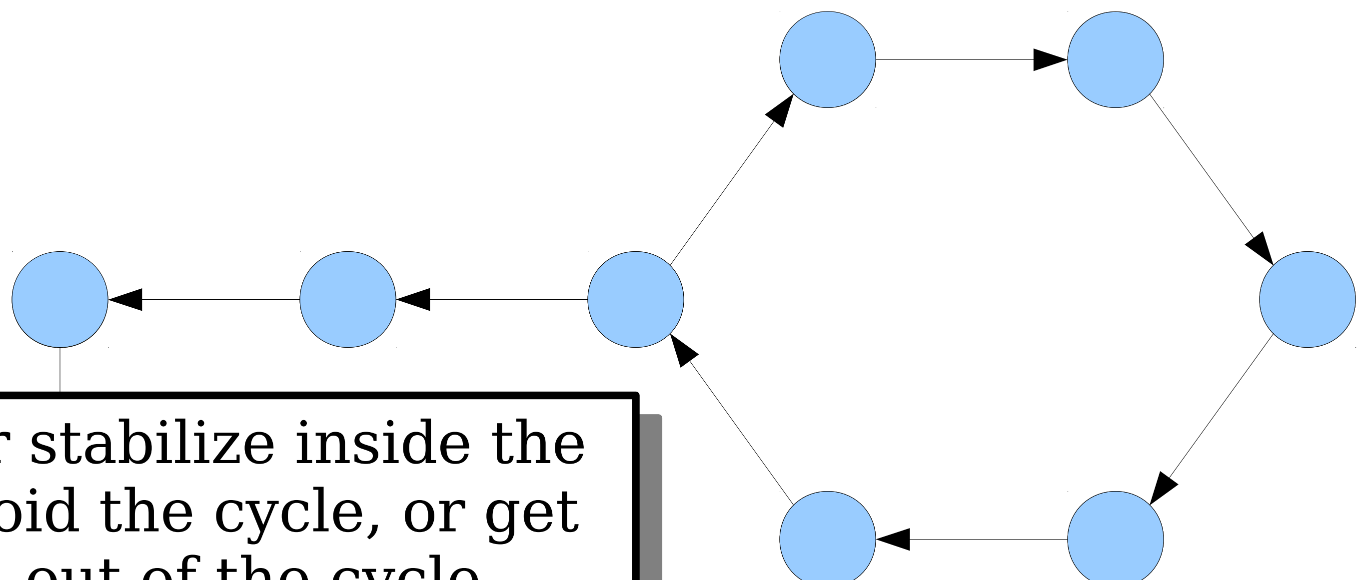



Insertions correspond to sequences of flipping arrowheads.



# The Cuckoo Graph

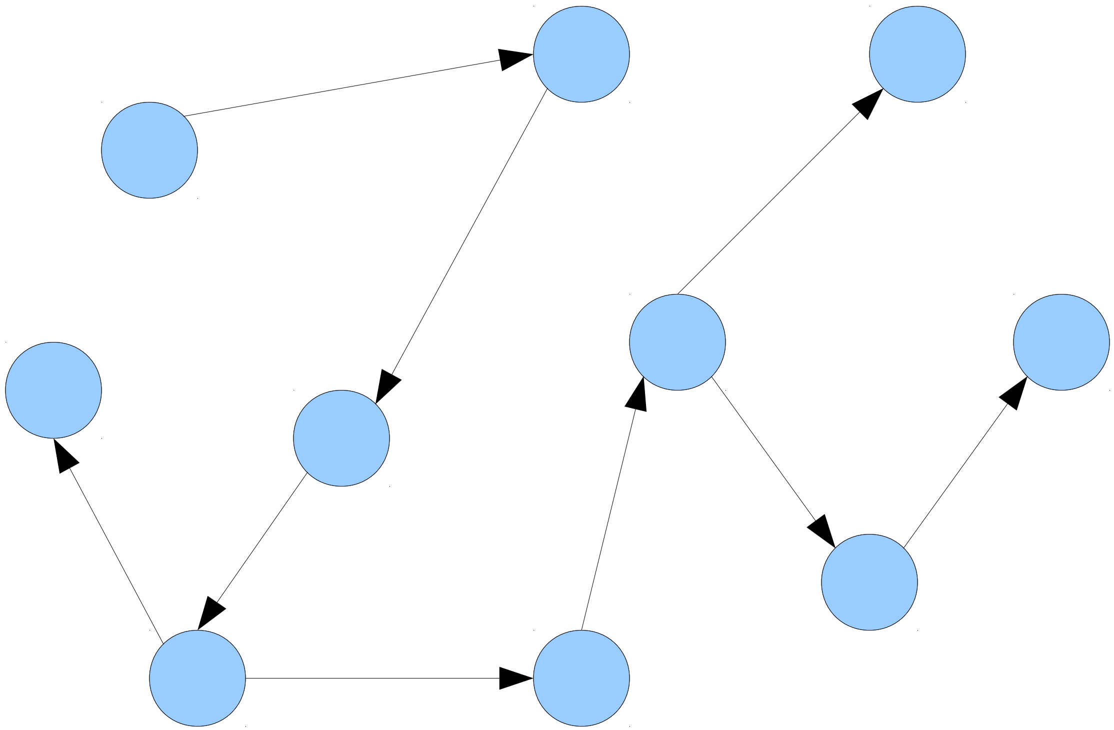
- ***Claim 1:*** If *x* is inserted into a cuckoo hash table, the insertion succeeds if the connected component containing *x* contains either no cycles or only one cycle.



We either stabilize inside the cycle, avoid the cycle, or get kicked out of the cycle.

# The Cuckoo Graph

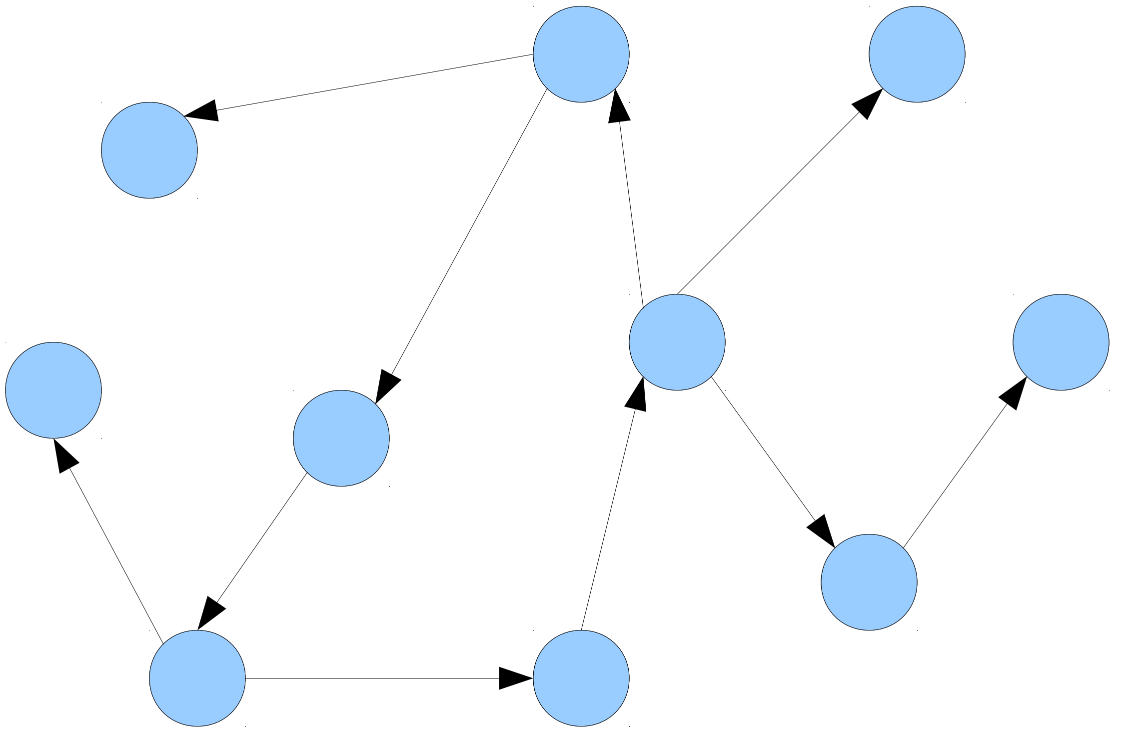
- ***Claim 2:*** If $x$ is inserted into a cuckoo hash table, the insertion fails if the connected component containing $x$ contains more than one cycle.



***No cycles:*** The graph is a directed tree. A tree with $k$ nodes has $k$ – 1 edges.

# The Cuckoo Graph

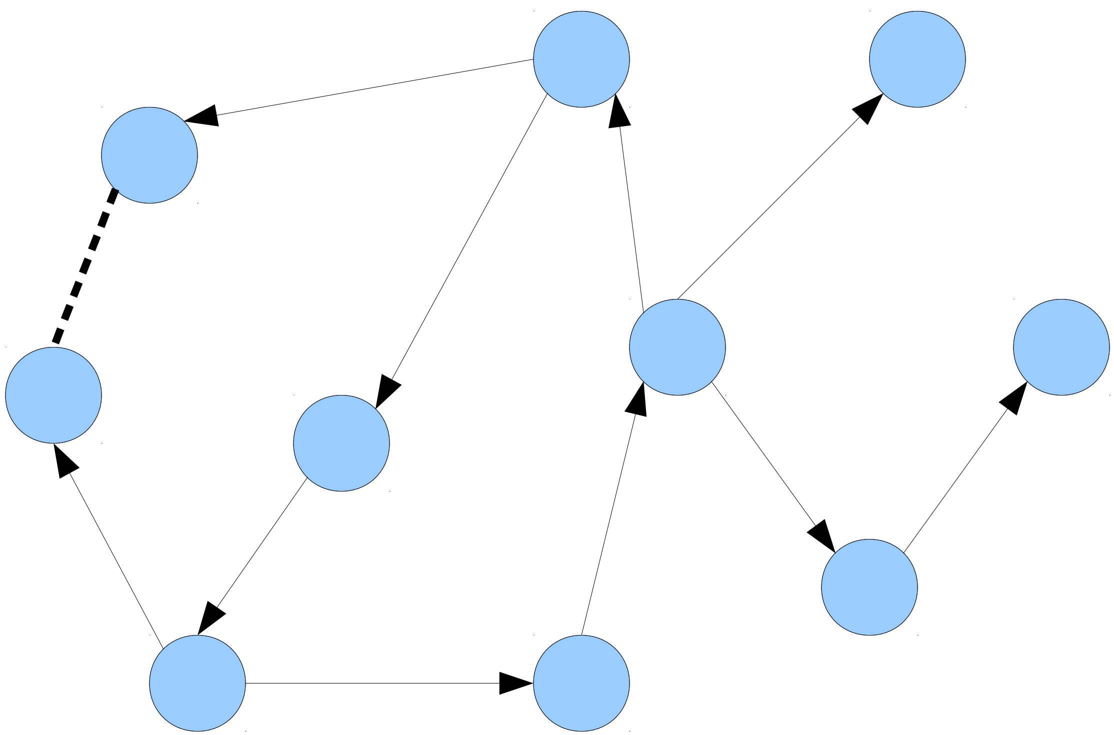
- ***Claim 2:*** If $x$ is inserted into a cuckoo hash table, the insertion fails if the connected component containing $x$ contains more than one cycle.



***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 arrowheads to place at most one arrowhead 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.
- Questions we still need to answer:
  - The number of nodes in a connected component can be used to bound the cost of an insertion. On average, how big are those connected components?
  - What is the probability that an insertion fails because we create a complex connected component?

Time-Out for Announcements!

# Project Checkpoints

- Project checkpoints were due today at 2:30PM.
- We'll be reviewing them over the weekend with the aim of getting back to you by early next week.
- In general, please feel free to reach out to us if you have any questions about the project or your topic! We're happy to help out.

# Problem Set Four

- Problem Set Four is due next Tuesday at 2:30PM.
- You know the drill! Ask questions if you have them, get in touch with us if there’s anything we can assist with, and have a lot of fun working through these exercises!

Back to CS166!

## *Two Major Questions*

How big are the connected components in the cuckoo graph?

How likely is it for an insertion to fail?

***Step One:*** Sizing Connected Components

# Analyzing Connected Components

- The cost of inserting $x$ into a cuckoo hash table is proportional to the size of the CC containing $x$.
- ***Question:*** What is the expected size of a CC in the cuckoo graph?

***Idea:*** Count the number of nodes in a connected component by simulating a BFS.

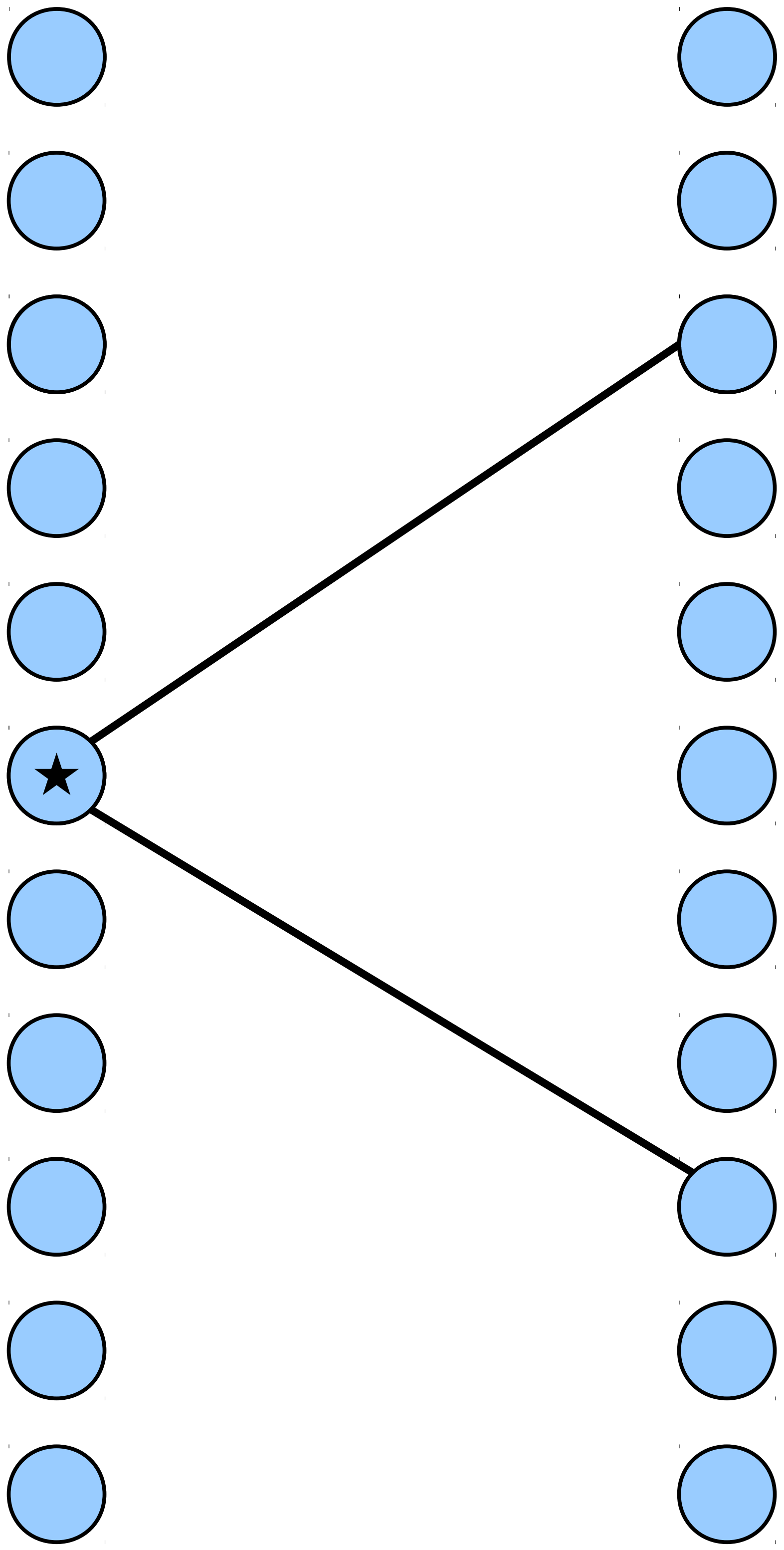
Pick some starting table slot.

There are *n* elements in the table, so this graph has *n* edges.

Assume, for now, that our hash functions are truly random.

Each edge has a $1/m$ chance of touching this table slot.

The number of adjacent nodes, which will be visited in the next step of BFS, is a Binom($n$, $1/m$) variable.



***Idea:*** Count the number of nodes in a connected component by simulating a BFS.

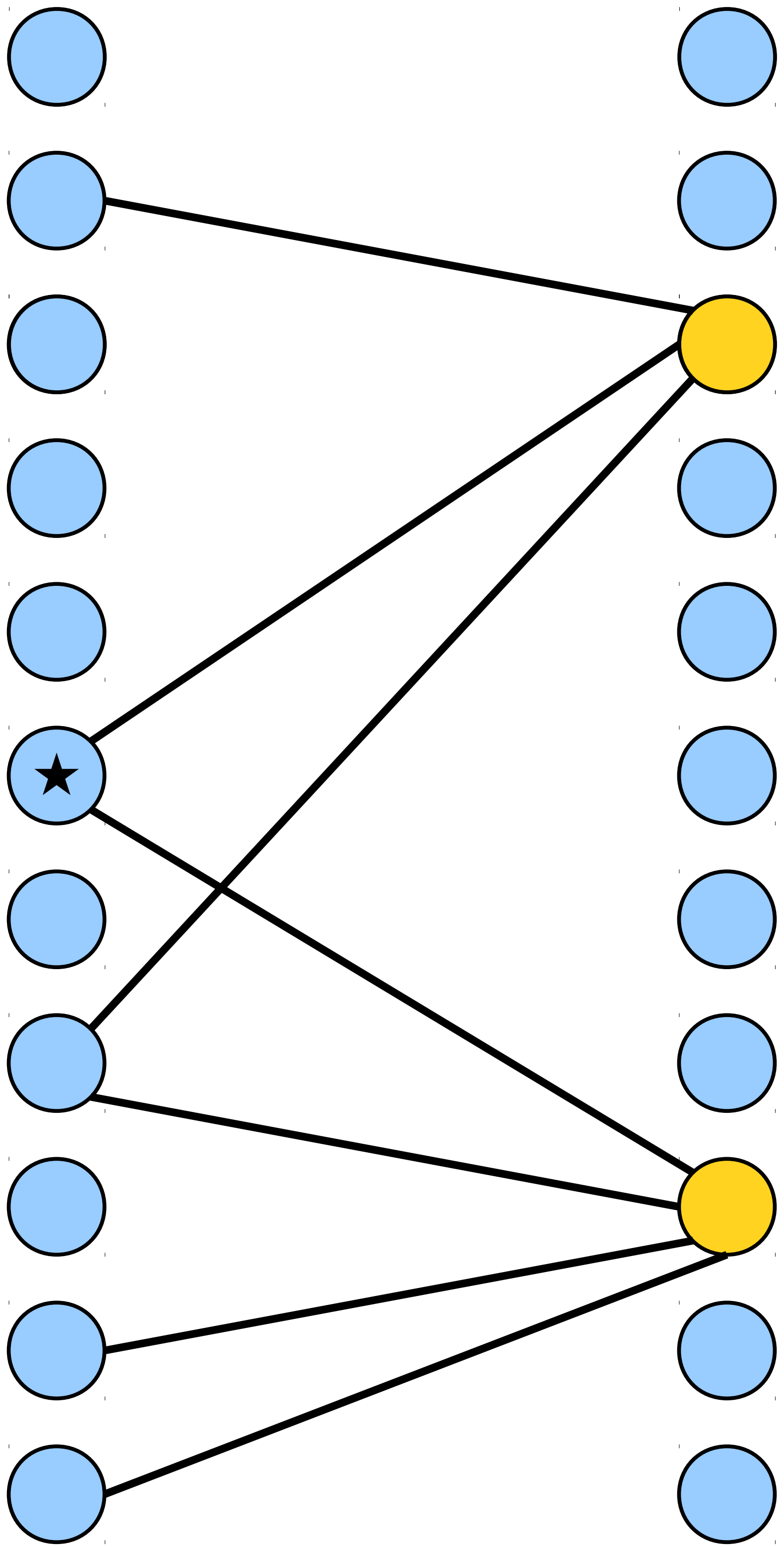
Each new node kinda sorta ish also touches a number of new nodes on the other side that can be modeled as a Binom($n$, $^1/_m$) variable.

This ignores double-counting nodes.

This ignores existing edges.

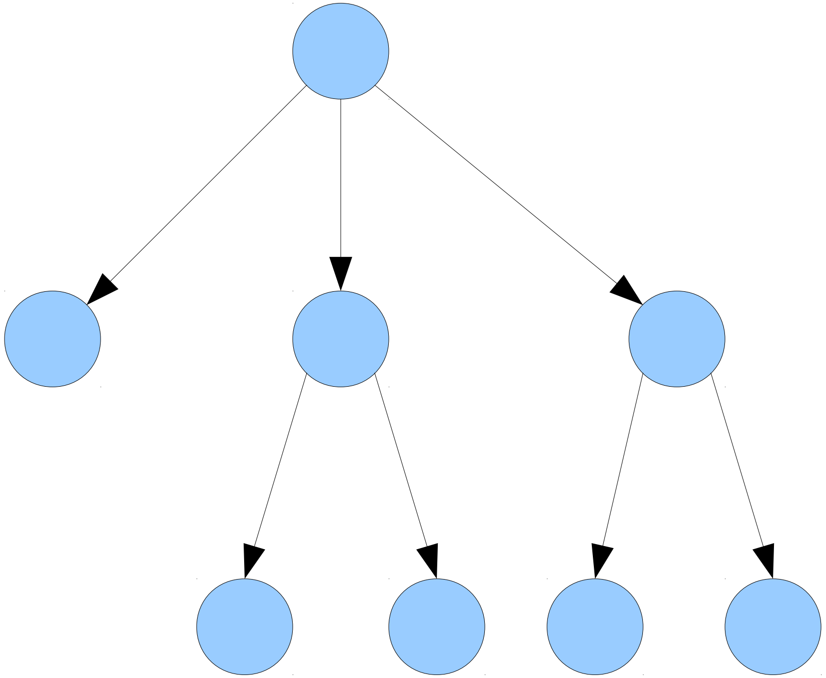
This ignores correlations between edge counts.

However, it conservatively bounds the next BFS step.



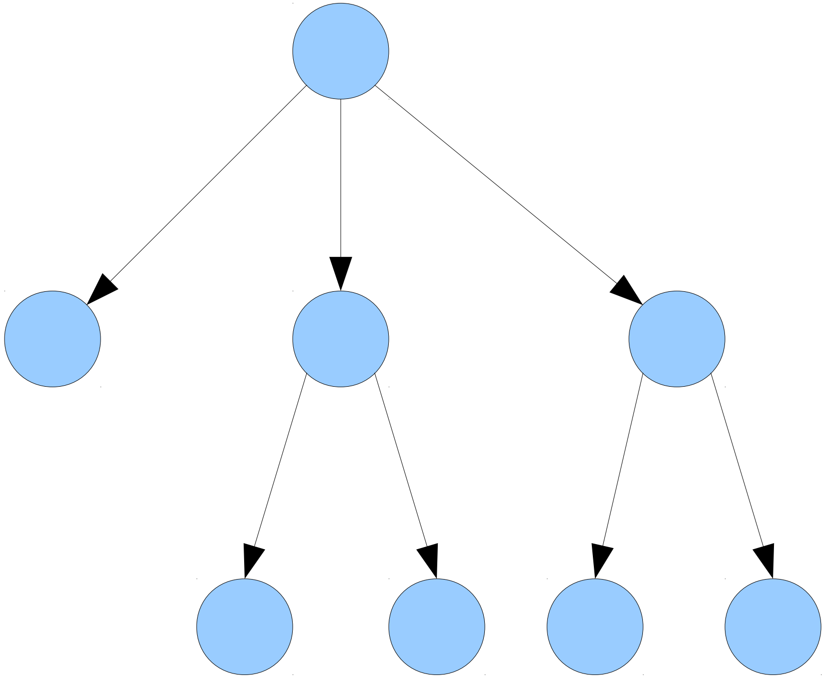
# Modeling the BFS

- Simulate a BFS tree using Binom($n$, $^1/_m$) variables!
  - Begin with a root node.
  - Each node has children distributed as a Binom($n$, $^1/_m$) variable.
- ***Question:*** How many total nodes will this simulated BFS discover before terminating?



# Modeling the BFS

- A tree-shaped process where each node has an i.i.d. number of children is called a ***Galton-Watson process***.
- A ***subcritical*** process is one where the expected number of children of each node is less than one.
- ***Question:*** Assuming each node's child count is an i.i.d. copy of some random variable ξ, how many nodes should we expect to see in the tree?

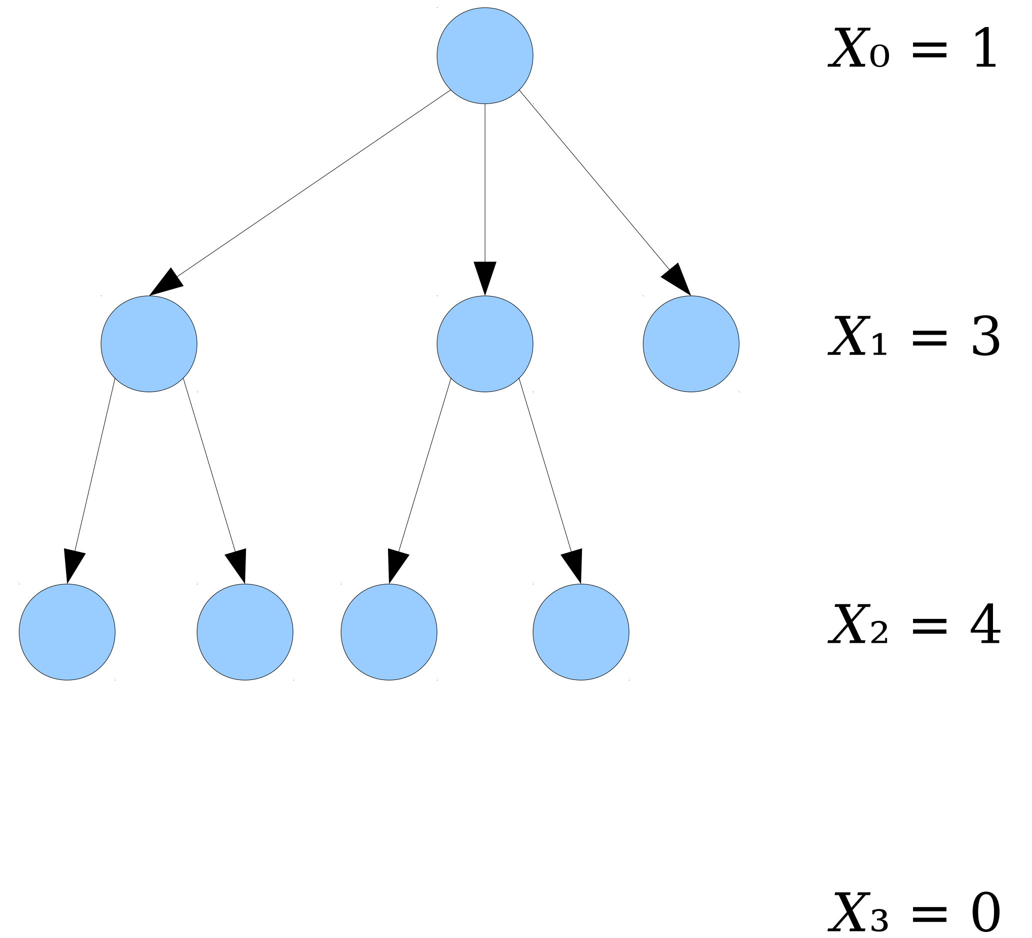


# Subcritical Galton-Watson Processes

- Denote by $X_n$ the number of nodes alive at depth $n$. This gives a series of random variables $X_0$, $X_1$, $X_2$, … .
- These variables are defined by the following randomized recurrence:

$$X_0 = 1 \qquad X_{n+1} = \sum_{i=1}^{X_n} \xi_{i,n}$$

- Here, each $\xi_{i,n}$ is an i.i.d. copy of ξ.



**Lemma 1:** $E[X_i] = E[\xi]^i$.

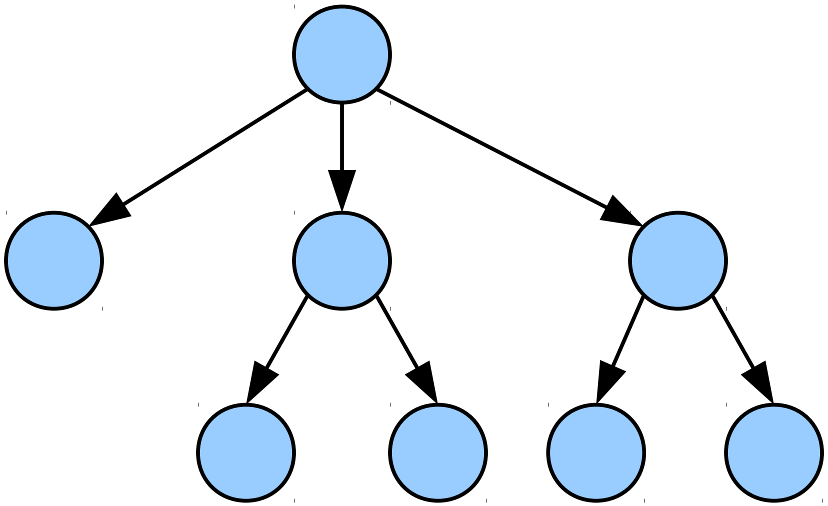
***(Induction and conditional expectation.)***

**Lemma 2:** $E\left[\sum_{i=0}^{\infty} X_i\right] = \frac{1}{1-E[\xi]}$

***(Linearity of expectation; sum of a geometric series.)***

**Theorem:** The expected number of nodes in a connected component of the cuckoo graph is O(1), assuming that $m = (1+\varepsilon)n$.

**Proof:** $\xi$ in this case is a Binom($n$, $1/m$) variable. So $E[\xi] = n/m = O(1)$.



---

$$X_0 = 1 \qquad X_{n+1} = \sum_{i=1}^{X_n} \xi_{i,n} \qquad E[\xi] < 1$$

# The Story So Far

- The expected size of a connected component in the cuckoo graph is O(1).
- Therefore, each *successful* insertion takes expected time O(1).
- ***Question:*** What happens in an unsuccessful insertion? And what does that do for our expected cost of *any* insertion?

Step Two:

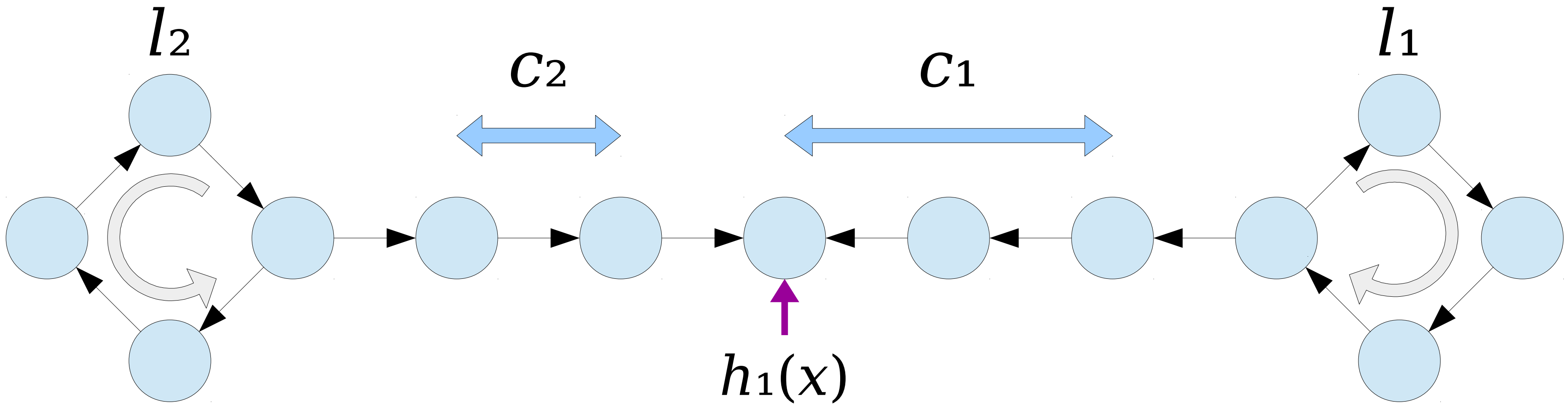
***Exploring the Graph Structure***

# Exploring the Graph Structure

- Cuckoo hashing will always succeed in the case where the cuckoo graph has no complex connected components.
- If there are no complex CC's, then we will not get into a loop and insertion time will depend only on the sizes of the CC's.
- It's reasonable to ask, therefore, how likely we are to not have complex components.

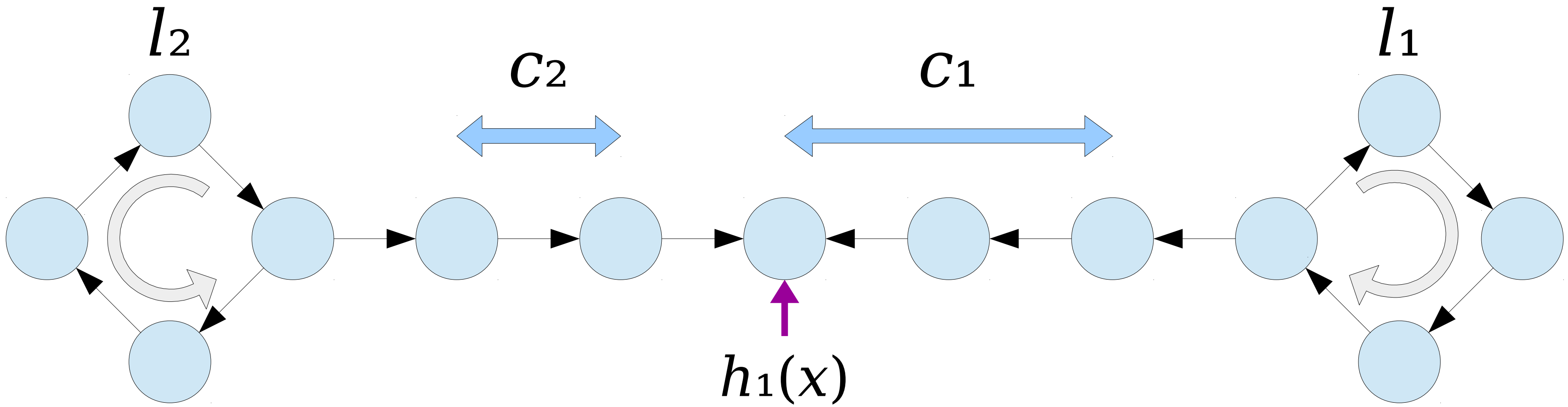
# Exploring the Graph Structure

- ***Question:*** What is the probability that a randomly-chosen bipartite multigraph with $2m$ nodes and $n$ edges will contain a complex connected component?
- Directly answering this question is challenging and requires some fairly detailed combinatorics.
- However, there's a very clever technique we can use to bound this probability indirectly.




***Question:*** What's the probability that we end up with a configuration like this one?

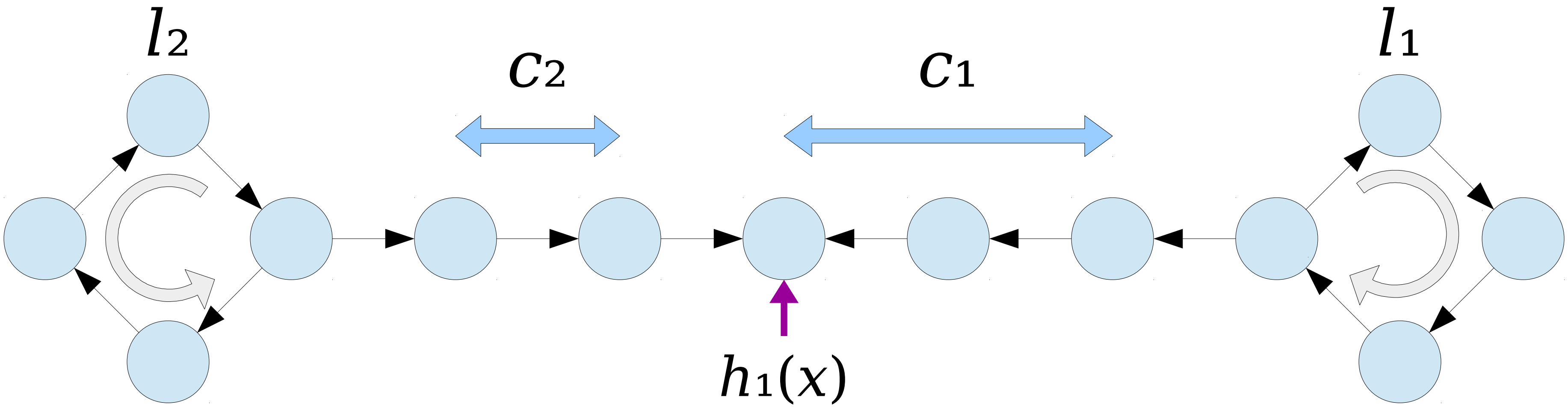
Insertion fails if we have a complex connected component. What specifically happens in that case?




This next proof comes from a CS166 final project by Noah Arthurs, Joseph Chang, and Nolan Handali. It's inspired by another argument due to Charles Chen (another Stanford student), which is a modification of one by Sanders and Vöcking, which was an improvement of one by Pagh and Rodler.

***Key idea:*** Use a traditional, CS109-style counting argument. Admittedly, it's a *nontrivial* counting argument, but it's a counting argument nonetheless!

Insertion fails if we have a complex connected component. What specifically happens in that case?




Ways to pick $k$ nodes (table slots) given the first is $h_1(x)$. *(upper bound)*

Ways to split $k$ nodes into $c_1$, $l_1$, $c_2$, and $l_2$. *(upper bound)*

Ways to assign $k$ keys to those slots. *(upper bound)*

Sum over all possible numbers of other keys being displaced.

$$\sum_{k=1}^{n} \left( \frac{(k+1)^4 \, m^{k-1} \, n^k}{m^{2k} \, m} \right)$$

Ways $h_1$ and $h_2$ can be chosen for those keys.

Ways $h_2(x)$ can be chosen.

---

Insertion fails if we have a complex connected component.
What specifically happens in that case?

$$\sum_{k=1}^{n}\left(\frac{(k+1)^4 m^{k-1} n^k}{m^{2k} m}\right) = \sum_{k=1}^{n}\left((k+1)^4 n^k m^{k-1-2k-1}\right)$$

$$= \sum_{k=1}^{n}\left((k+1)^4 n^k m^{k-2}\right)$$

$$= \frac{1}{m^2}\sum_{k=1}^{n}\left((k+1)^4 n^k m^{-k}\right)$$

$$= \frac{1}{m^2}\sum_{k=1}^{n}(k+1)^4\left(\frac{n}{m}\right)^k$$

$$m = (1 + \varepsilon)n$$

$$= \frac{1}{m^2}\sum_{k=1}^{n}\frac{(k+1)^4}{(1+\varepsilon)^k}$$

$$\sum_{k=1}^{n}\left(\frac{(k+1)^4\, m^{k-1}\, n^k}{m^{2k}\, m}\right) = \sum_{k=1}^{n}\left((k+1)^4\, n^k\, m^{k-1-2k-1}\right)$$

$$= \sum_{k=1}^{n}\left((k+1)^4\, n^k\, m^{k-2}\right)$$

$$= \frac{1}{m^2}\sum_{k=1}^{n}\left((k+1)^4\, n^k\, m^{-k}\right)$$

Numerator grows *polynomially* as a function of $k$.

$$= \frac{1}{m^2}\sum_{k=1}^{n}(k+1)^4\left(\frac{n}{m}\right)^k$$

Denominator grows *exponentially* as a function of $k$.

$$= \frac{1}{m^2}\sum_{k=1}^{n}\frac{(k+1)^4}{(1+\varepsilon)^k}$$

$$= \frac{1}{m^2}\cdot O(1)$$

$$
\begin{aligned}
\sum_{k=1}^{n}\left(\frac{(k+1)^4\, m^{k-1}\, n^k}{m^{2k}\, m}\right) &= \sum_{k=1}^{n}\left((k+1)^4\, n^k\, m^{k-1-2k-1}\right) \\
&= \sum_{k=1}^{n}\left((k+1)^4\, n^k\, m^{k-2}\right) \\
&= \frac{1}{m^2}\sum_{k=1}^{n}\left((k+1)^4\, n^k\, m^{-k}\right) \\
&= \frac{1}{m^2}\sum_{k=1}^{n}(k+1)^4\left(\frac{n}{m}\right)^k \\
&= \frac{1}{m^2}\sum_{k=1}^{n}\frac{(k+1)^4}{(1+\varepsilon)^k} \\
&= \frac{1}{m^2}\cdot \mathrm{O}(1) \\
&= \mathbf{O}\left(\frac{\mathbf{1}}{\boldsymbol{m}^{\mathbf{2}}}\right)
\end{aligned}
$$

**Question 1:** What is the probability at least one insert fails if we do $n$ total insertions?

$$
\begin{aligned}
& \Pr[\text{some insert fails}] \\
\leq\ & \sum_{k=1}^{n} \Pr[\text{the } k\text{th insert fails}] \\
=\ & \sum_{k=1}^{n} O\left(\frac{1}{m^2}\right) \\
=\ & O\left(\frac{n}{m^2}\right) \\
=\ & \mathbf{O\left(\frac{1}{m}\right)}
\end{aligned}
$$

The probability that a single insertion fails is $O(1 / m^2)$ if $m = (1+\varepsilon)n$.

If an insertion fails, we ***rehash*** by building a brand-new table, with new hash functions, and inserting all old elements.

It’s possible that, when we do a rehash, one of the insertions fails. Therefore, we keep rehashing until we find a working table.

***Question 2:*** On expectation, how many rehashes are needed per insertion?

---

The probability that a series of $n$ insertions fail is $O(1 / m)$.

***Question 2:*** On expectation, how many rehashes are needed per insertion?

Let $X$ be a random variable counting the number of rehashes assuming at least one rehash occurs.

$X$ is geometrically distributed with success probability $1 - O(1 / m)$.

$$E[X] = \frac{1}{1 - O(1/m)} = \mathbf{O(1)}$$

$$\begin{aligned} & E[\#\text{rehashes}] \\ = \; & E[X] \cdot \Pr[\#\text{rehashes} > 0] \\ = \; & O(1) \cdot O(1/m^2) \\ = \; & \mathbf{O(1/m^2)} \end{aligned}$$

---

The probability that a series of $n$ insertions fail is $O(1 / m)$.

***Question 3:*** What is the expected cost of an insertion into a cuckoo hash table?

$$\mathbf{O(1)} + \mathbf{O(1 / m^2)} \cdot \mathbf{O(m)}$$

- $O(1)$: Expected cost of successful insertion.
- $O(1 / m^2)$: Expected number of rehashes.
- $O(m)$: Cost of doing one rehash.

---

The expected number of rehashes on any insertion is $O(1 / m^2)$.

***Question 3:*** What is the expected cost of an insertion into a cuckoo hash table?

**O(1)**

The expected number of rehashes on any insertion is $O(1 / m^2)$.

# The Overall Analysis

- Cuckoo hashing gives worst-case lookups and deletions.
- Insertions are expected, amortized O(1).
- 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*

# More to Explore

# Hash Function Strength

- We analyzed cuckoo hashing assuming our hash functions were truly random. That’s often too strong of an assumption.
- What we know:
  - 6-independent hashing isn't sufficient for expected O(1) insertion time, but that O(log $n$)-independence is.
  - Some simple classes of hash functions (e.g. 2-independent polynomial) perform poorly for cuckoo hashing.
  - Some simple classes of hash functions (e.g. 3-independent simple tabulation) perform very well.
- ***Open problem:*** Determine the strength of hash function needed for cuckoo hashing to work efficiently.

# Multiple Tables

- Cuckoo hashing works well with two tables. So why not 3, 4, 5, …, or $k$ tables?
- In practice, cuckoo hashing with $k \geq 3$ tables tends to perform much better than cuckoo hashing with $k = 2$ tables:
  - The load factor can increase substantially; with $k=3$, it's only around $\alpha = 0.91$ that you run into trouble with the cuckoo graph.
  - Displacements are less likely to chain together; they only occur when all hash locations are filled in.
- ***Open problem:*** Determine where these phase transition thresholds are for arbitrary $k$.

# Increasing Bucket Sizes

- What if each slot in a cuckoo hash table can store multiple elements?
- When displacing an element, choose a random one to move and move it.
- This turns out to work remarkably well in practice, since it makes it really unlikely that you'll have long chains of displacements.
- ***Open problem:*** Quantify the effect of larger bucket sizes on the overall runtime of cuckoo hashing.

# Restricting Moves

- Insertions in cuckoo hashing only run into trouble when you encounter long chains of displacements during insertions.
- ***Idea:*** Cap the number of displacements at some fixed factor, then store overflowing elements in a secondary hash table.
- In practice, this works remarkably well, since the auxiliary table doesn't tend to get very large.
- ***Open problem:*** Quantify the effects of “hashing with a stash” for arbitrary stash sizes and displacement limits.

# Other Dynamic Schemes

- There is another famous dynamic perfect hashing scheme called ***dynamic FKS hashing***.
- It works by using closed addressing and resolving collisions at the top level with a secondary (static) perfect hash table.
- In practice, it's not as fast as these other approaches. However, it only requires 2-independent hash functions.
- Check CLRS for details!

# Lower Bounds?

- ***Open Problem:*** Is there a hash table that supports amortized O(1) insertions, deletions, and lookups?
- You'd think that we'd know the answer to this question, but, sadly, we don't.

# Next Time

- ***Approximate Membership Queries***
  - Educated guesses about whether things have been seen before.
- ***Bloom Filters***
  - The original – and one of the most popular – solutions to this problem.
- ***Quotient Filters***
  - Adapting linear probing for AMQ.
- ***Cuckoo Filters***
  - Adapting cuckoo hashing for AMQ.