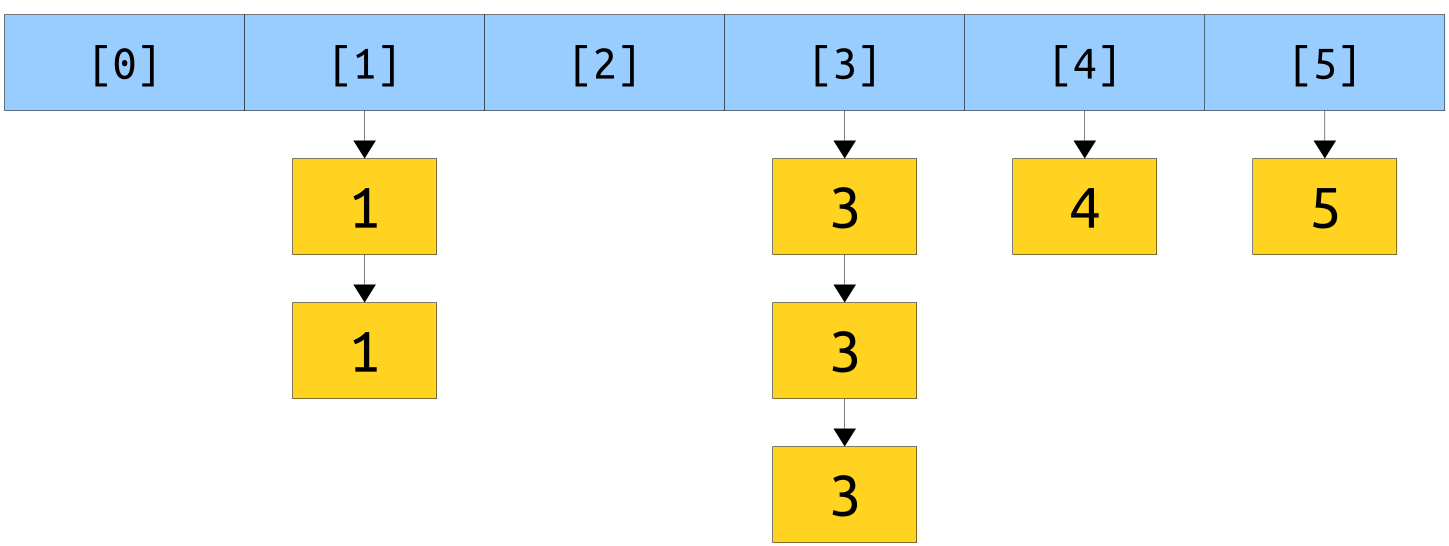
Cuckoo Hashing

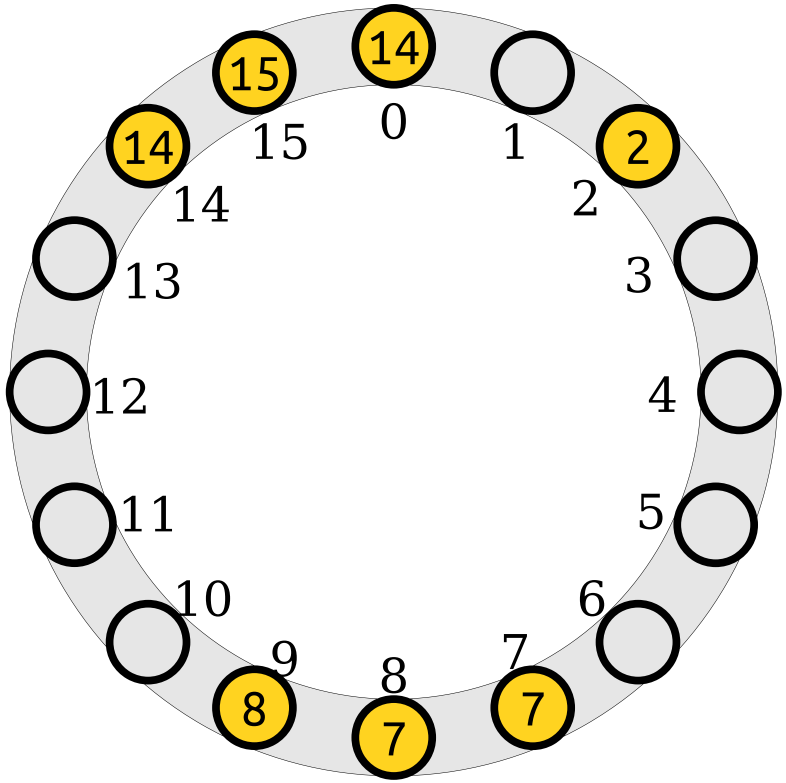
Collision Resolution

- 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.)



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.)



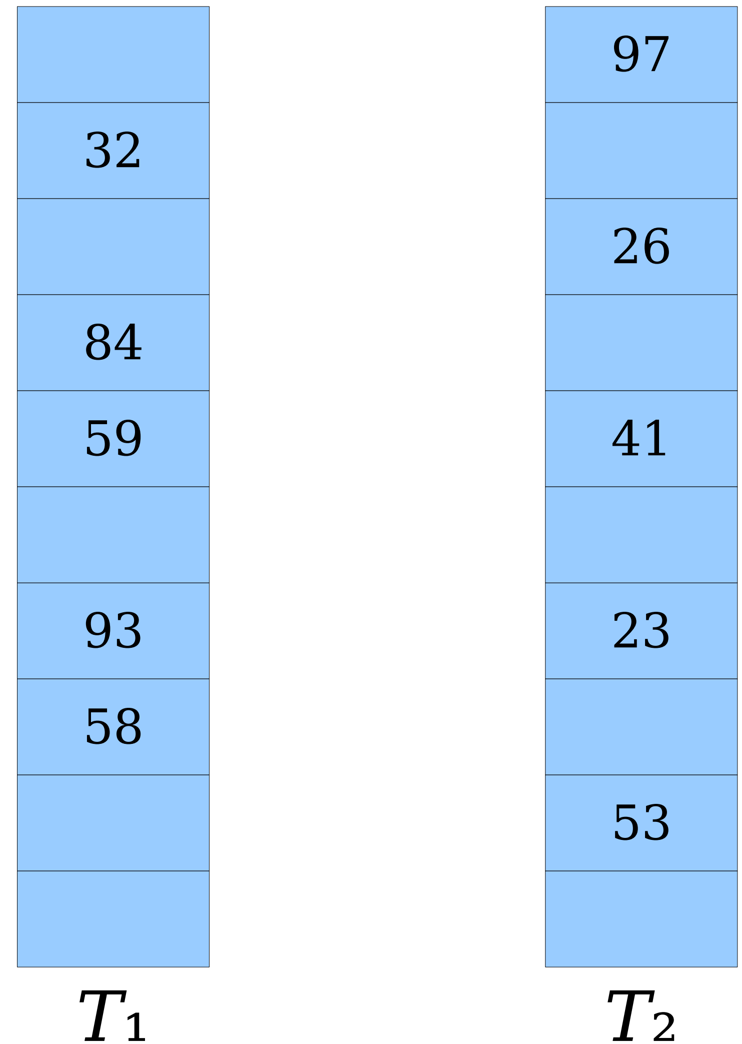
Collision Resolution

- 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

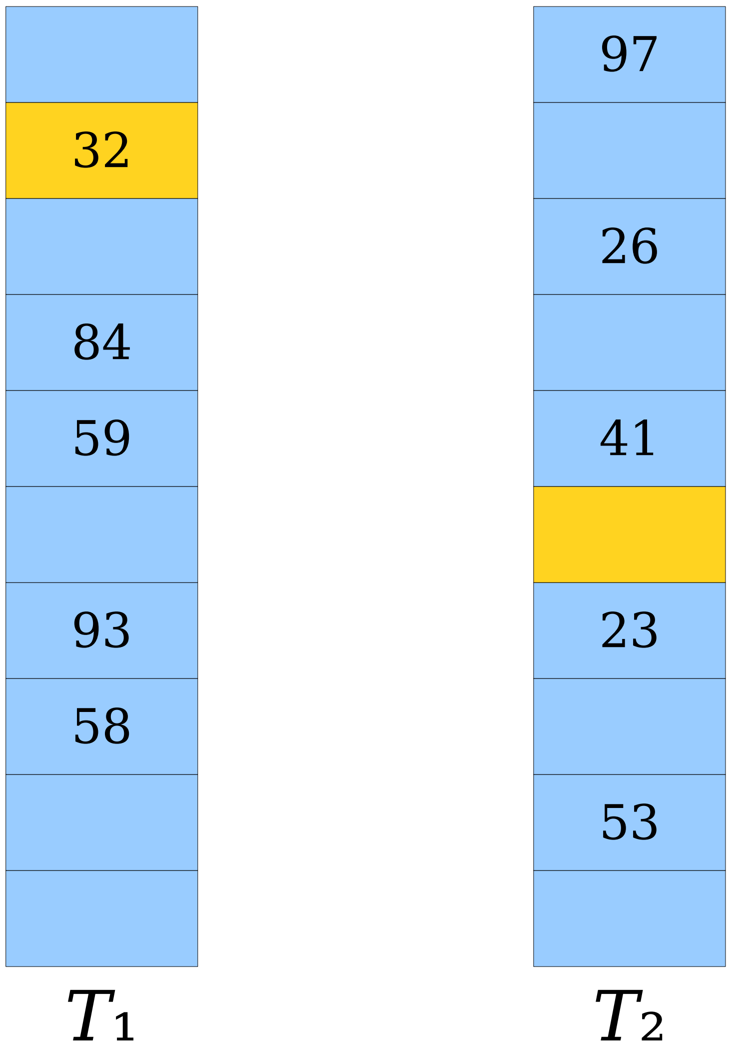
Cuckoo Hashing

- 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.
- We'll talk about hash strength later; for now, assume truly random hash functions.



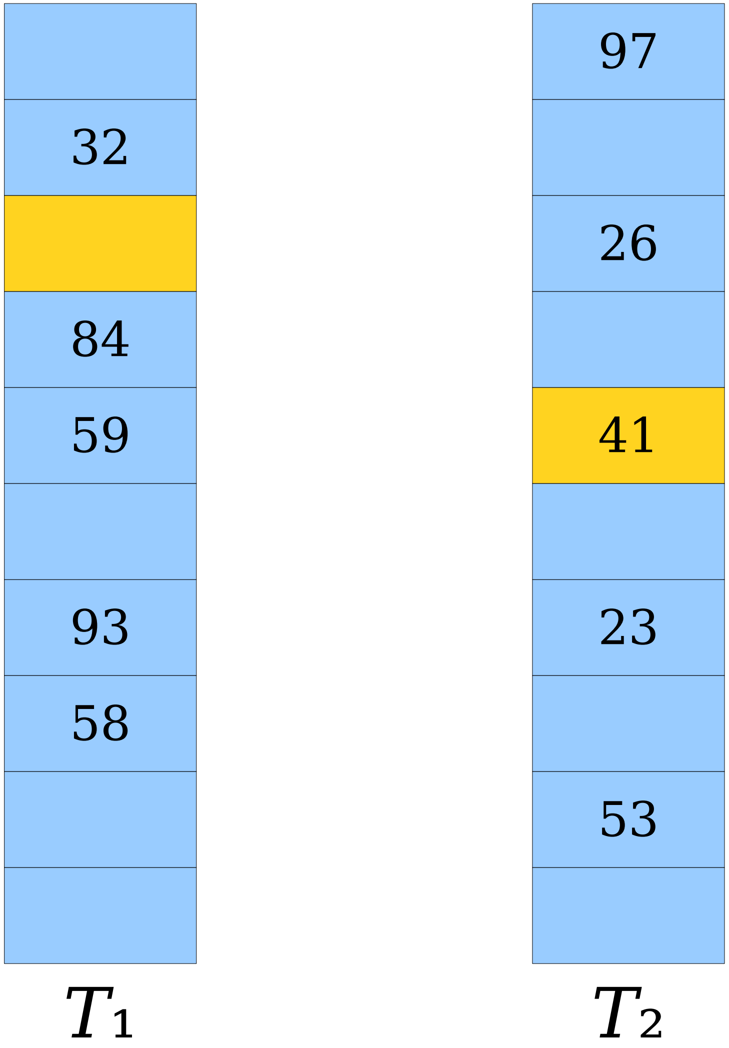
Cuckoo Hashing

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



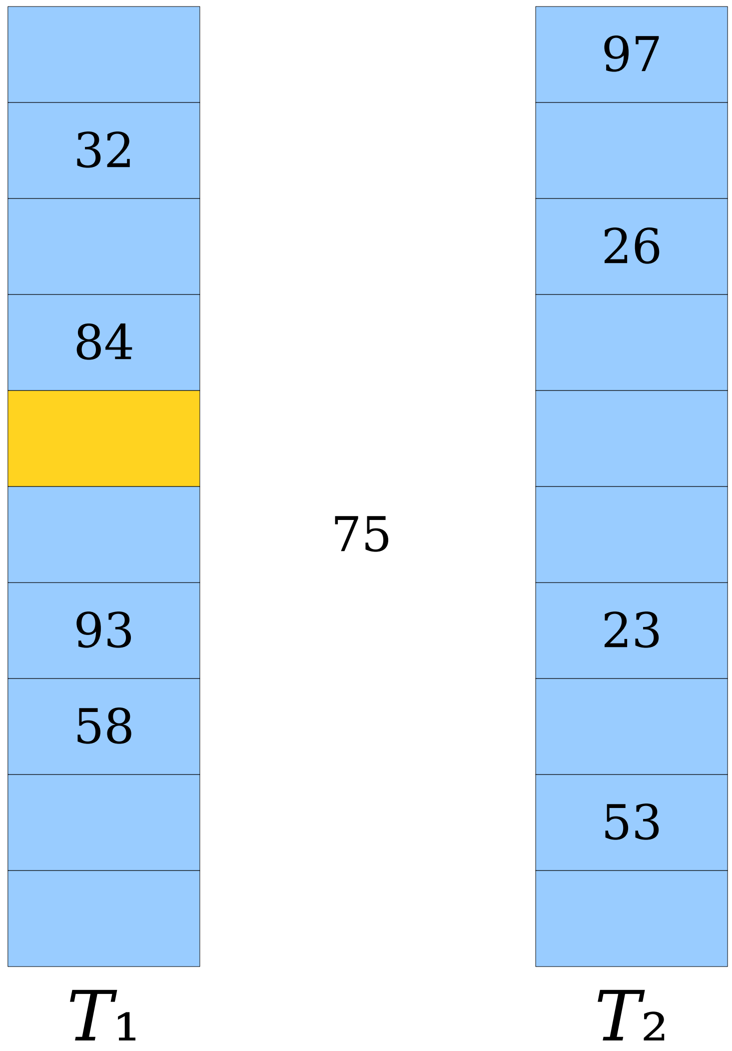
Cuckoo Hashing

- 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.



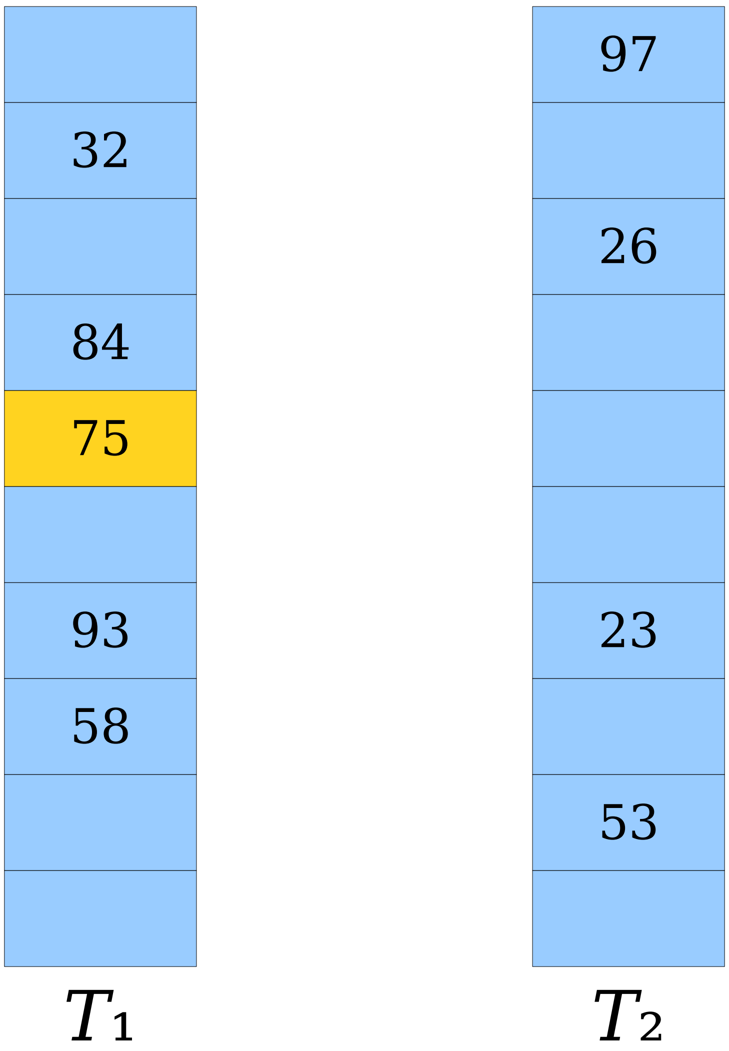
Cuckoo Hashing

- To insert an element x, start by inserting it into table 1.
- If $h_1(x)$ is empty, place x there.



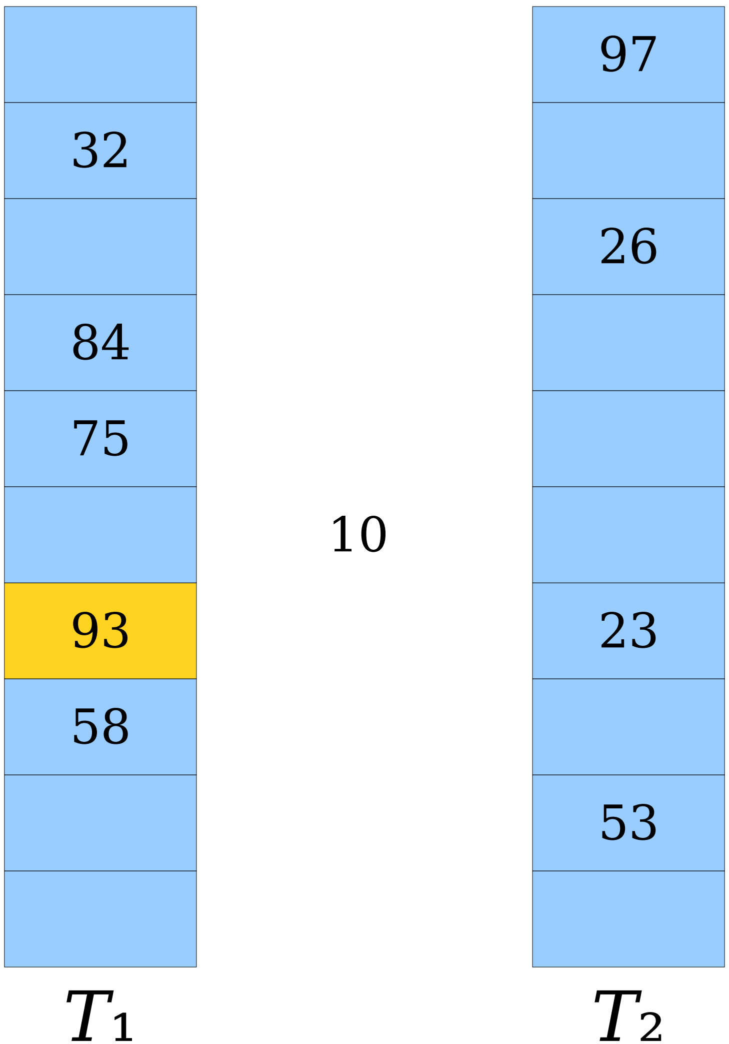
Cuckoo Hashing

- To insert an element x, start by inserting it into table 1.
- If $h_1(x)$ is empty, place x there.



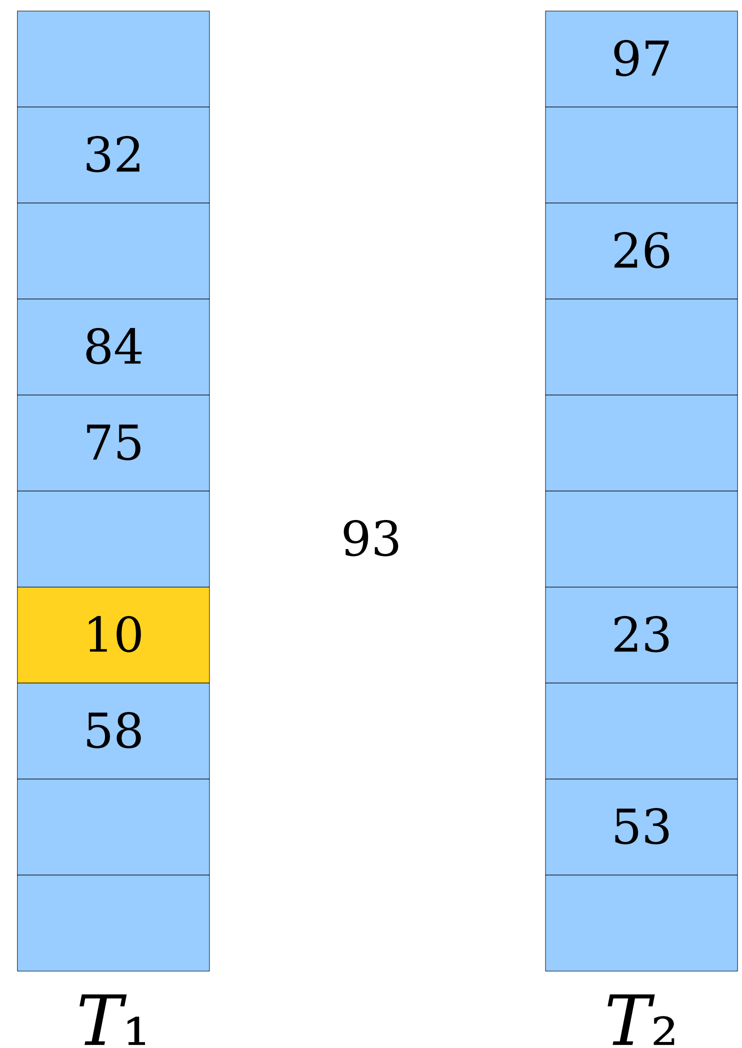
Cuckoo Hashing

- 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.



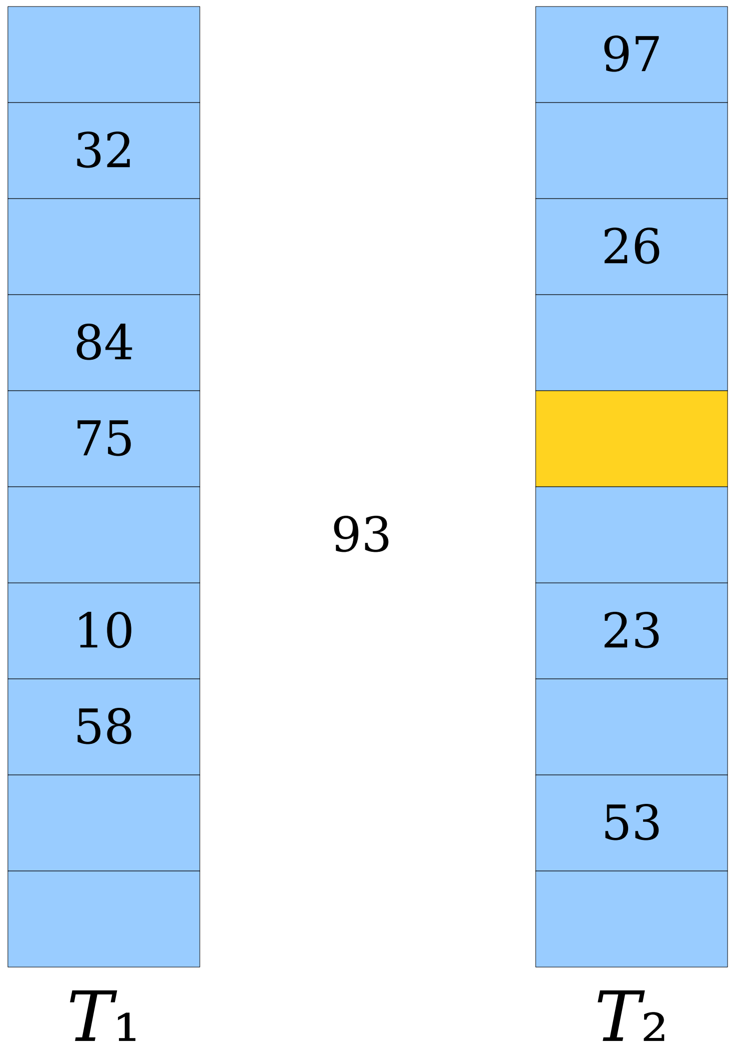
Cuckoo Hashing

- 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.



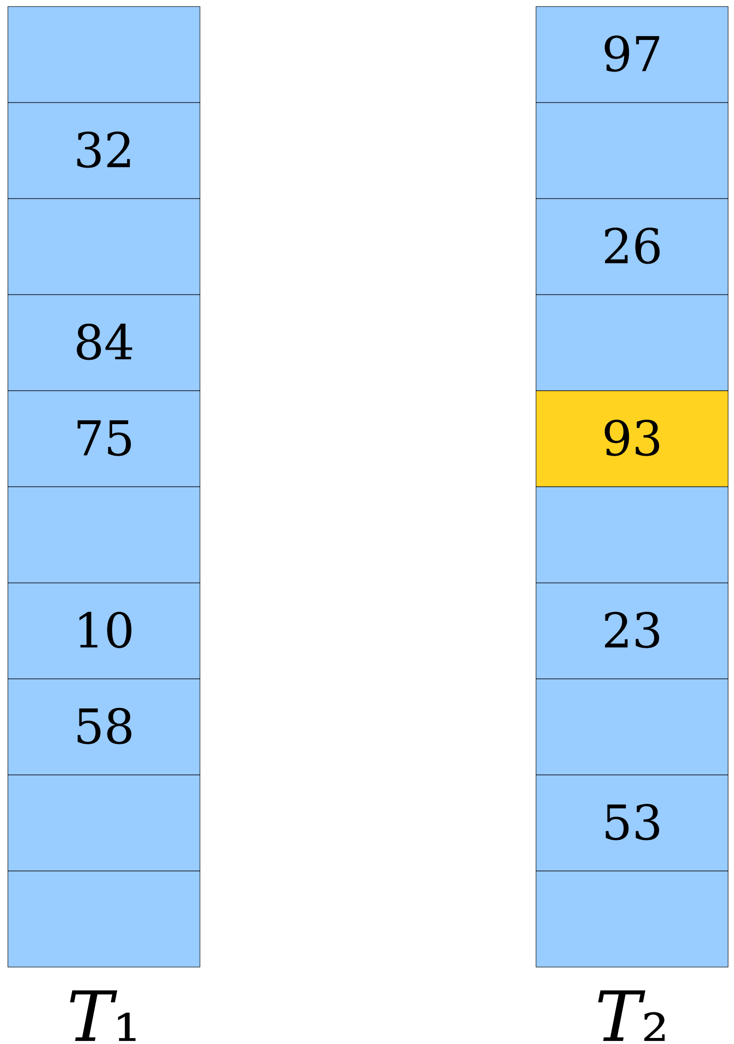
Cuckoo Hashing

- 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.



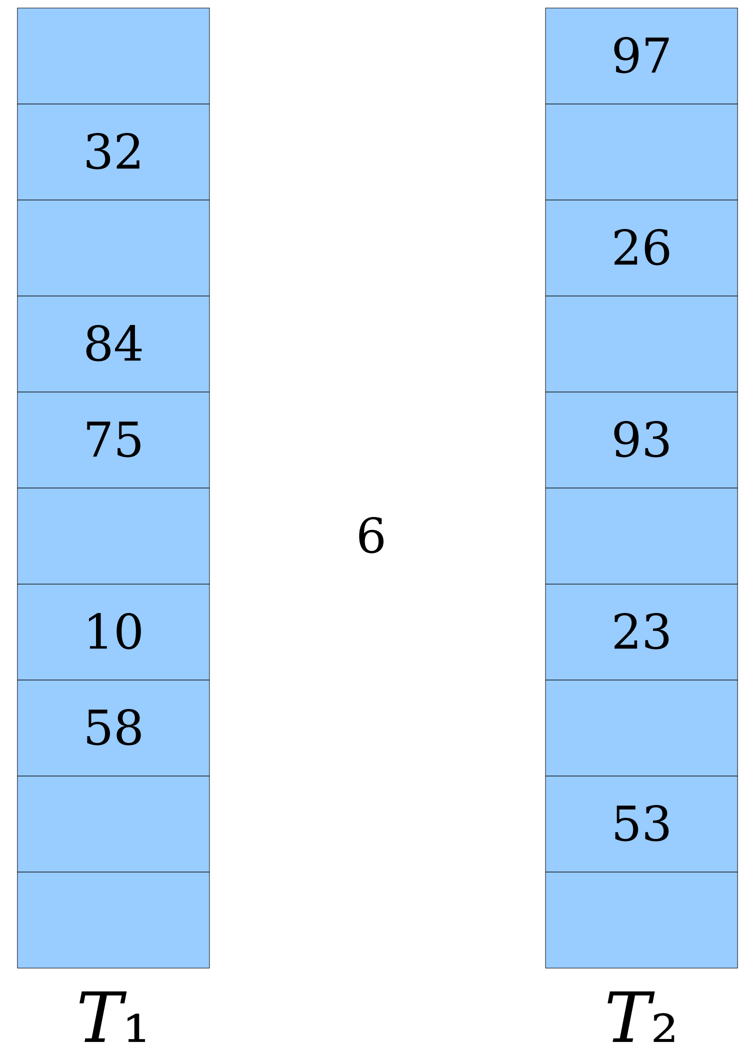
Cuckoo Hashing

- 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.



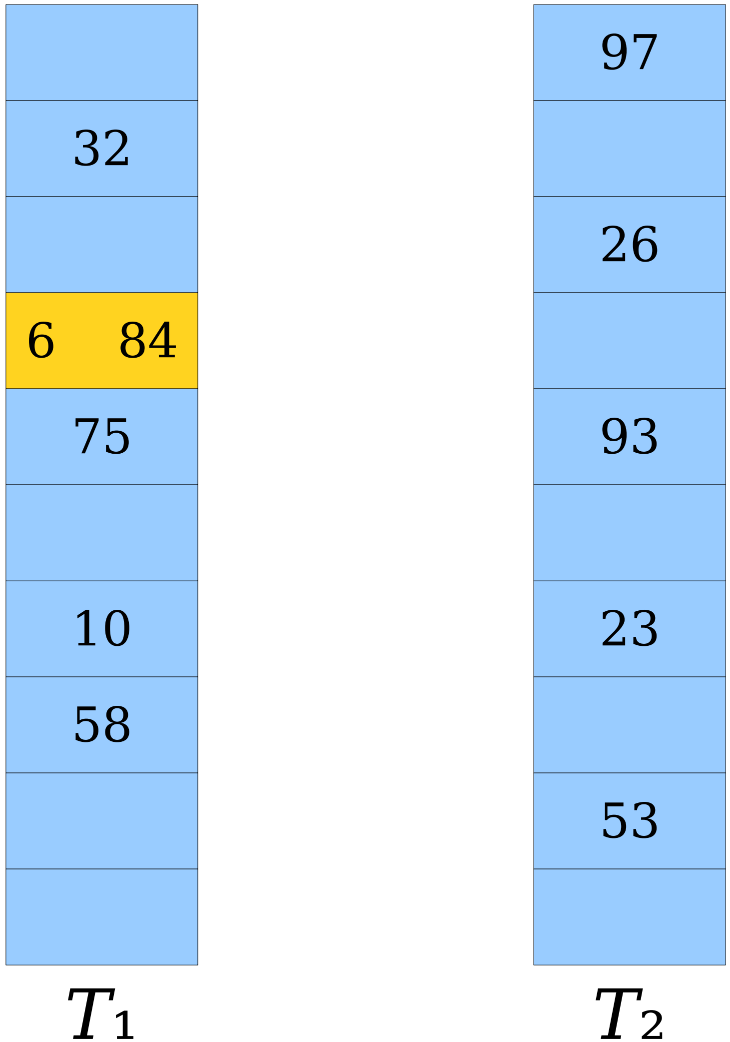
Cuckoo Hashing

- 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.



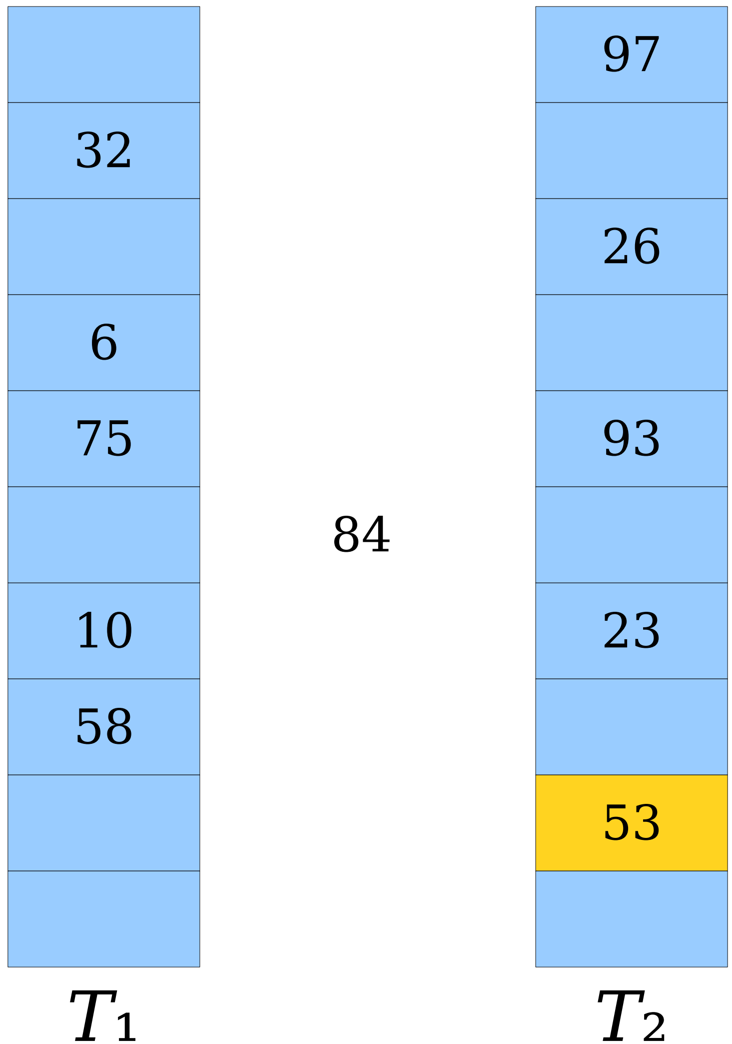
Cuckoo Hashing

- 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.



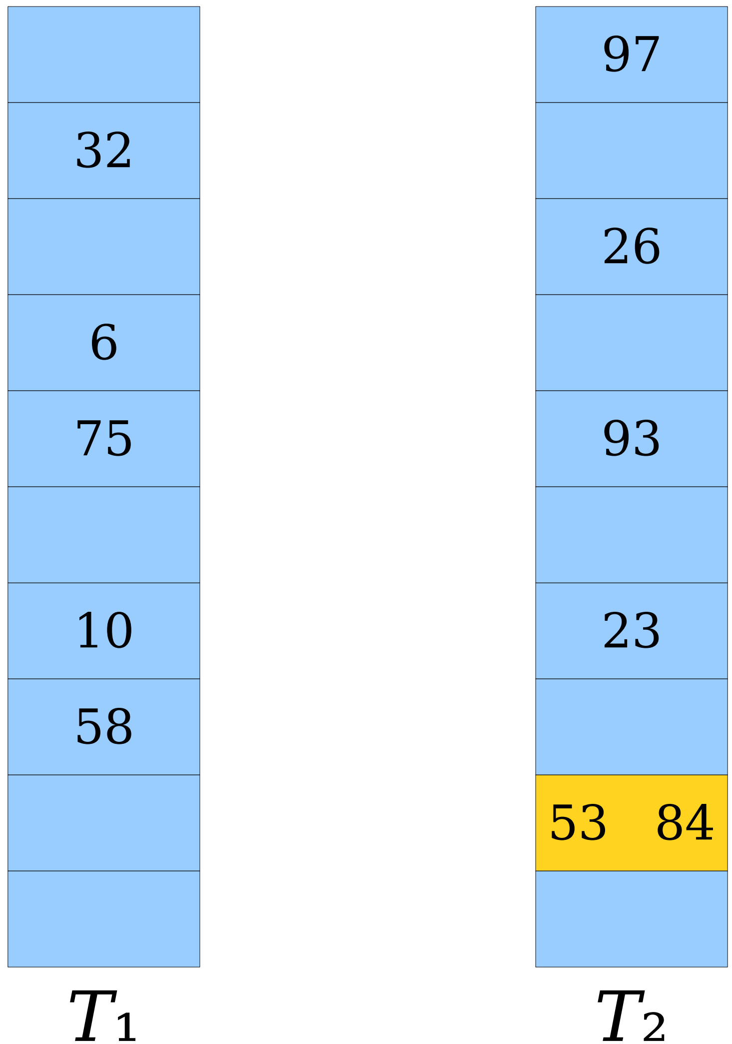
Cuckoo Hashing

- 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.



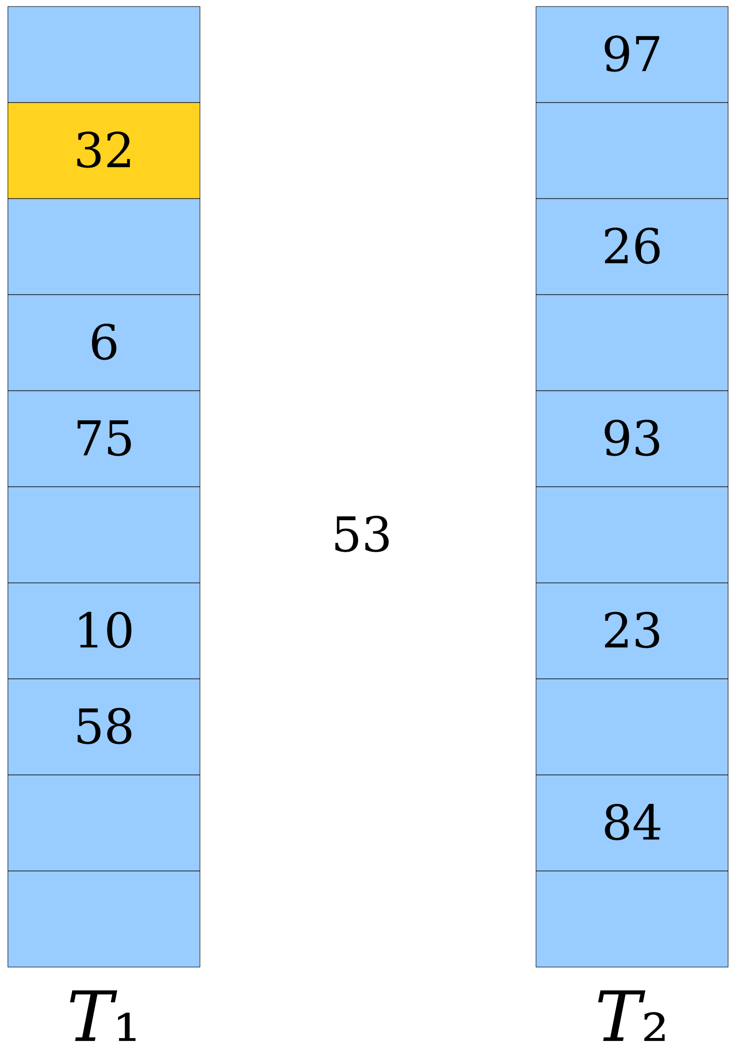
Cuckoo Hashing

- 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.



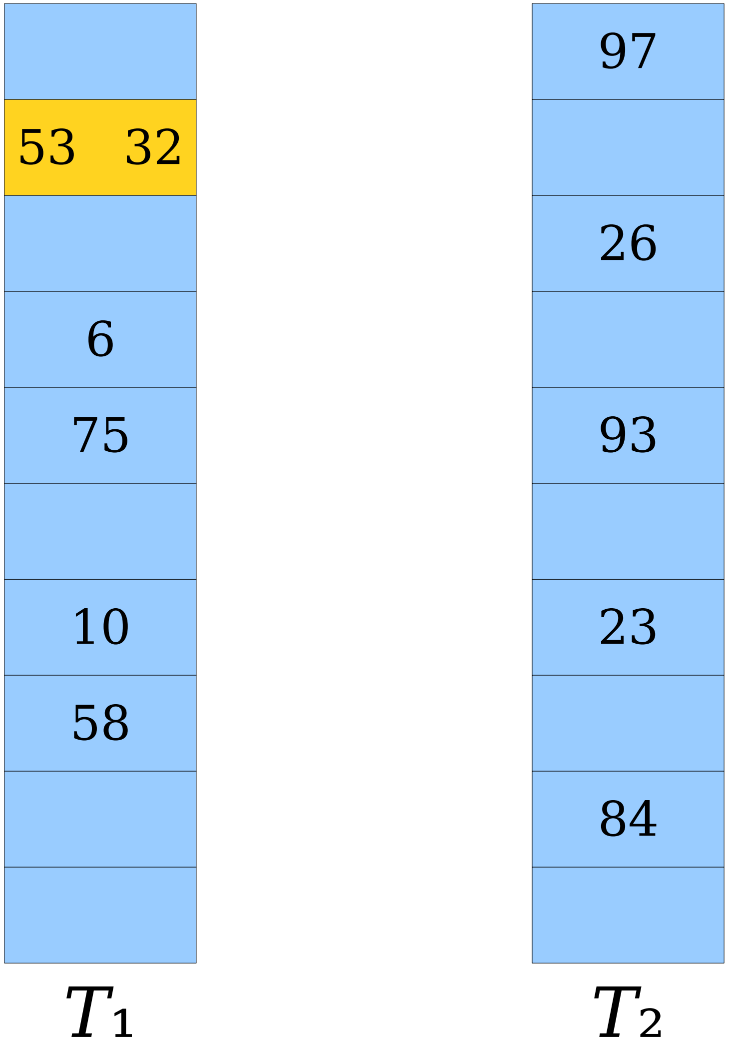
Cuckoo Hashing

- 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.



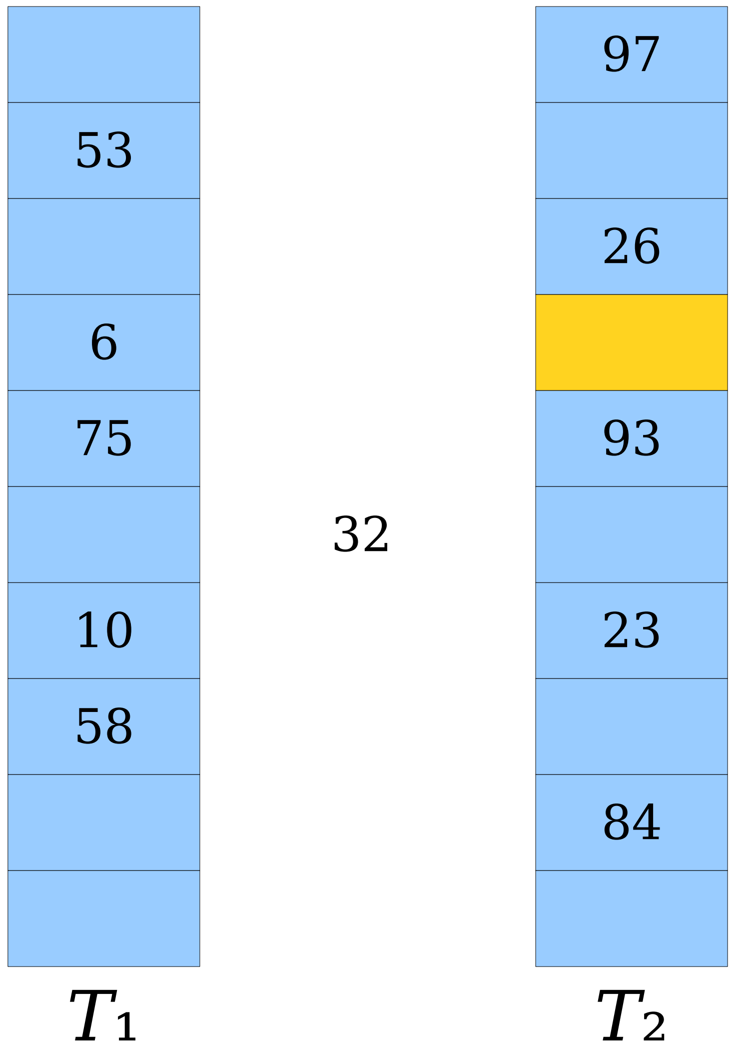
Cuckoo Hashing

- 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.



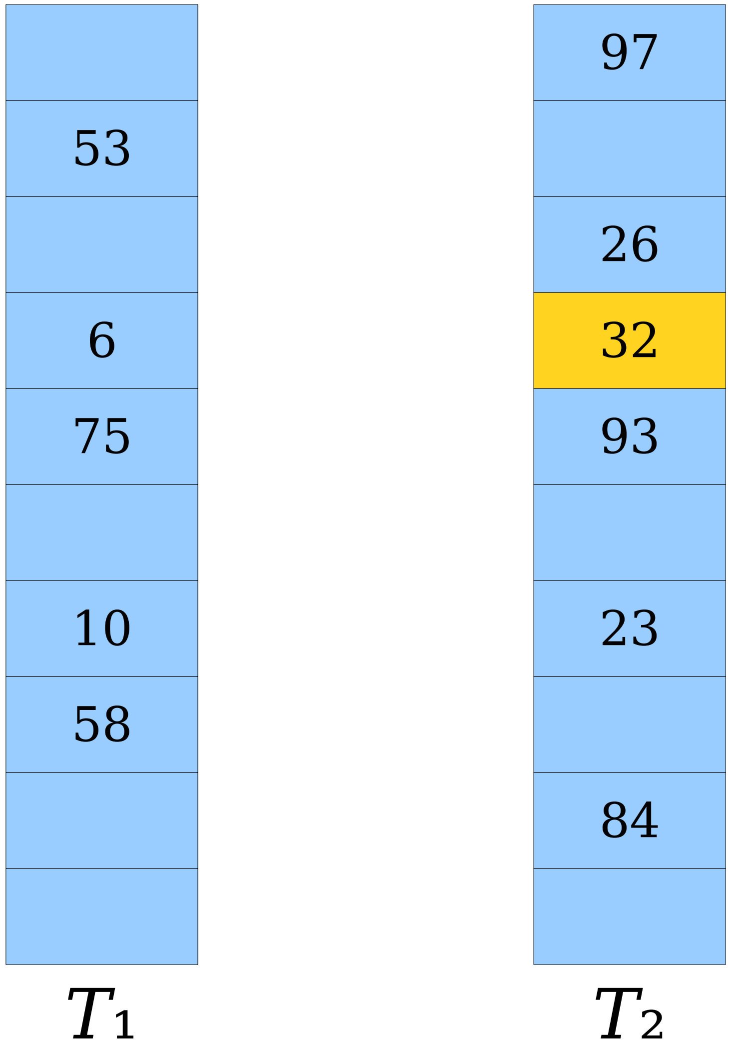
Cuckoo Hashing

- 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.



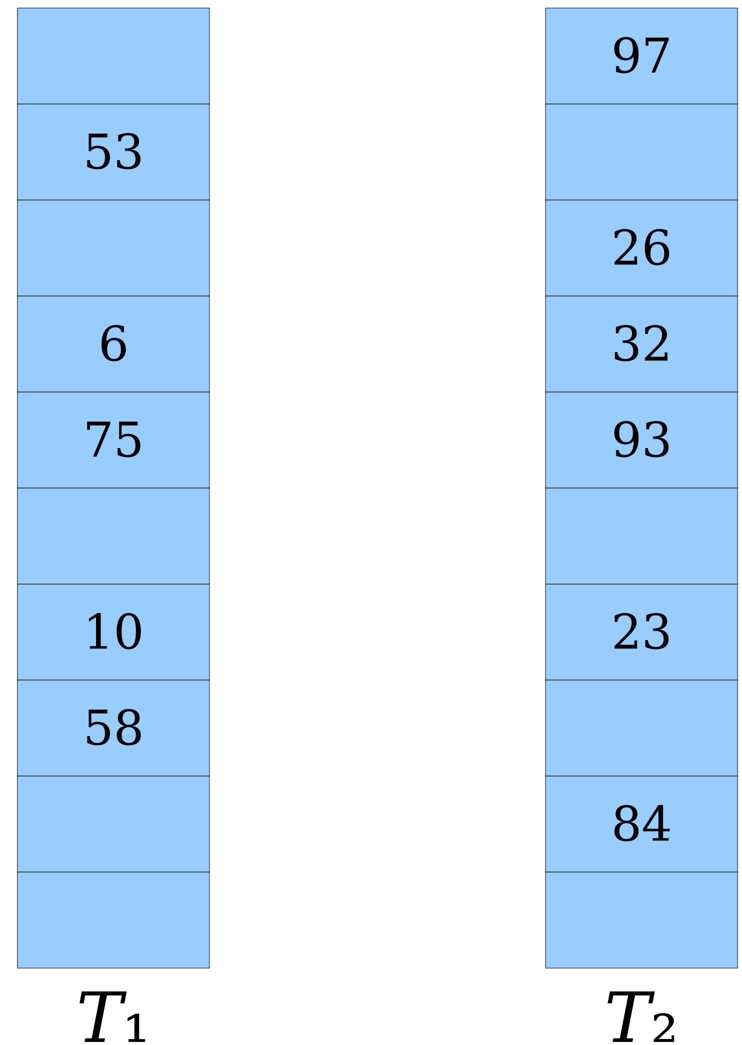
Cuckoo Hashing

- 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.



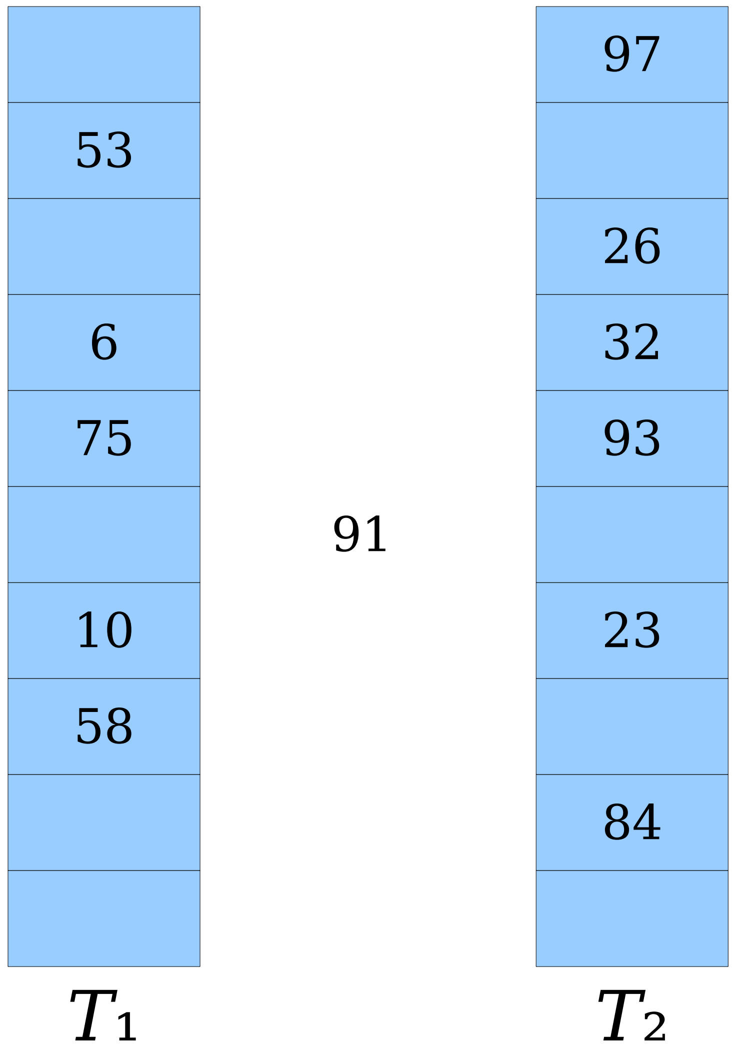
Cuckoo Hashing

- 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.



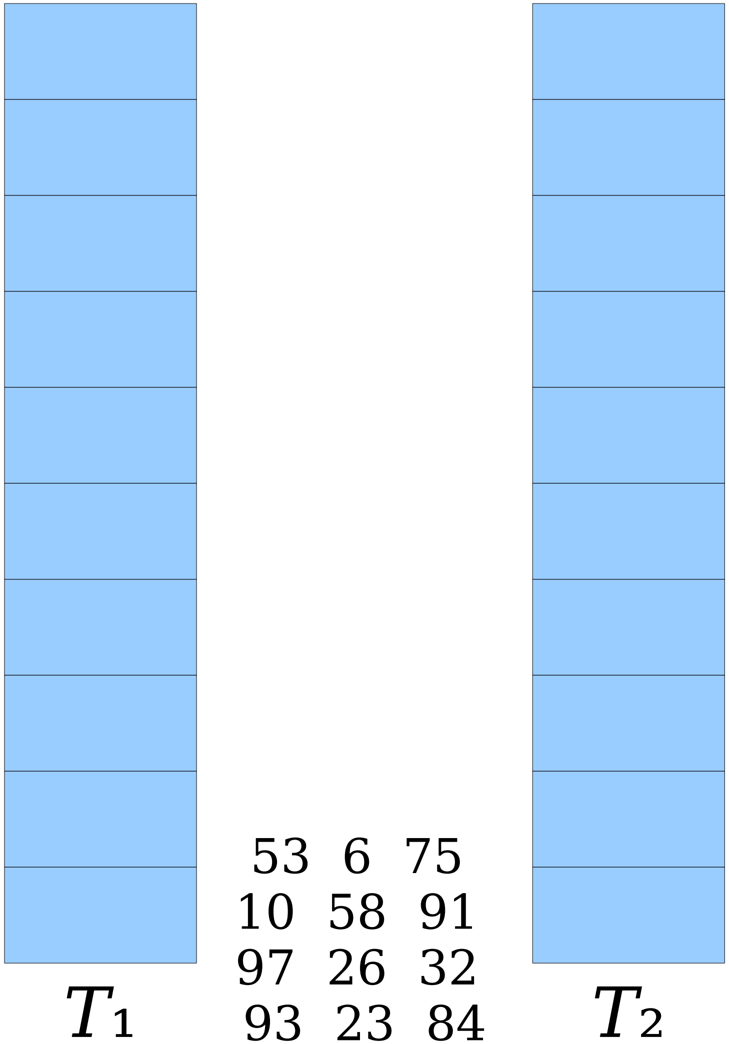
Cuckoo Hashing

- 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.



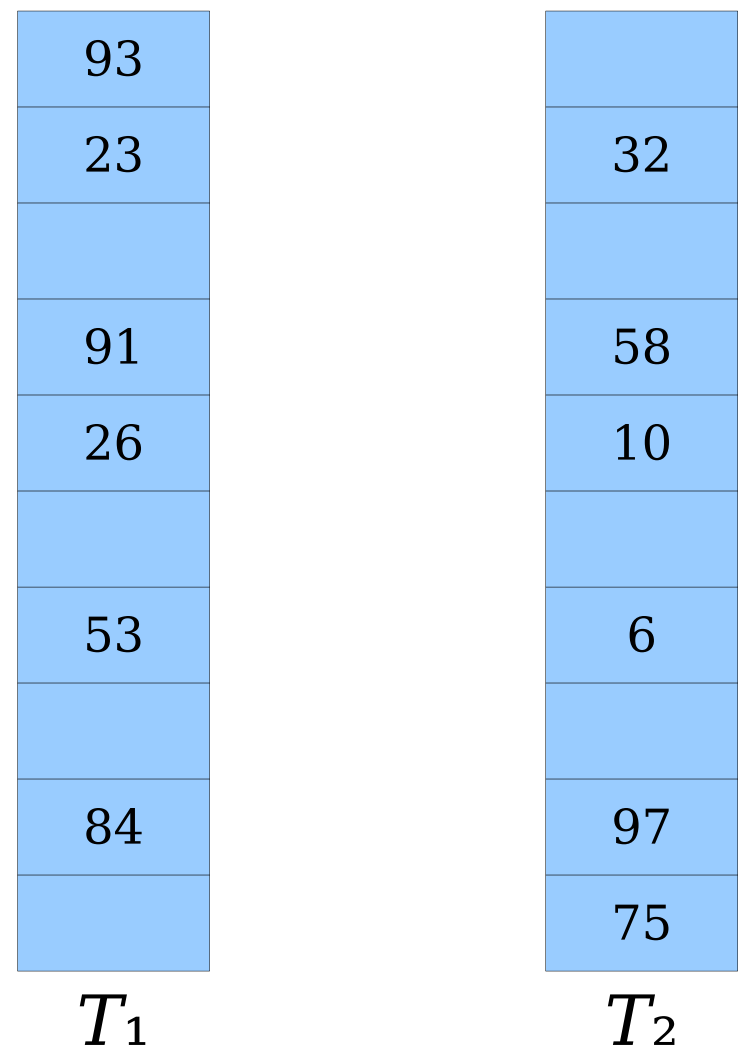
Cuckoo Hashing

- 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.



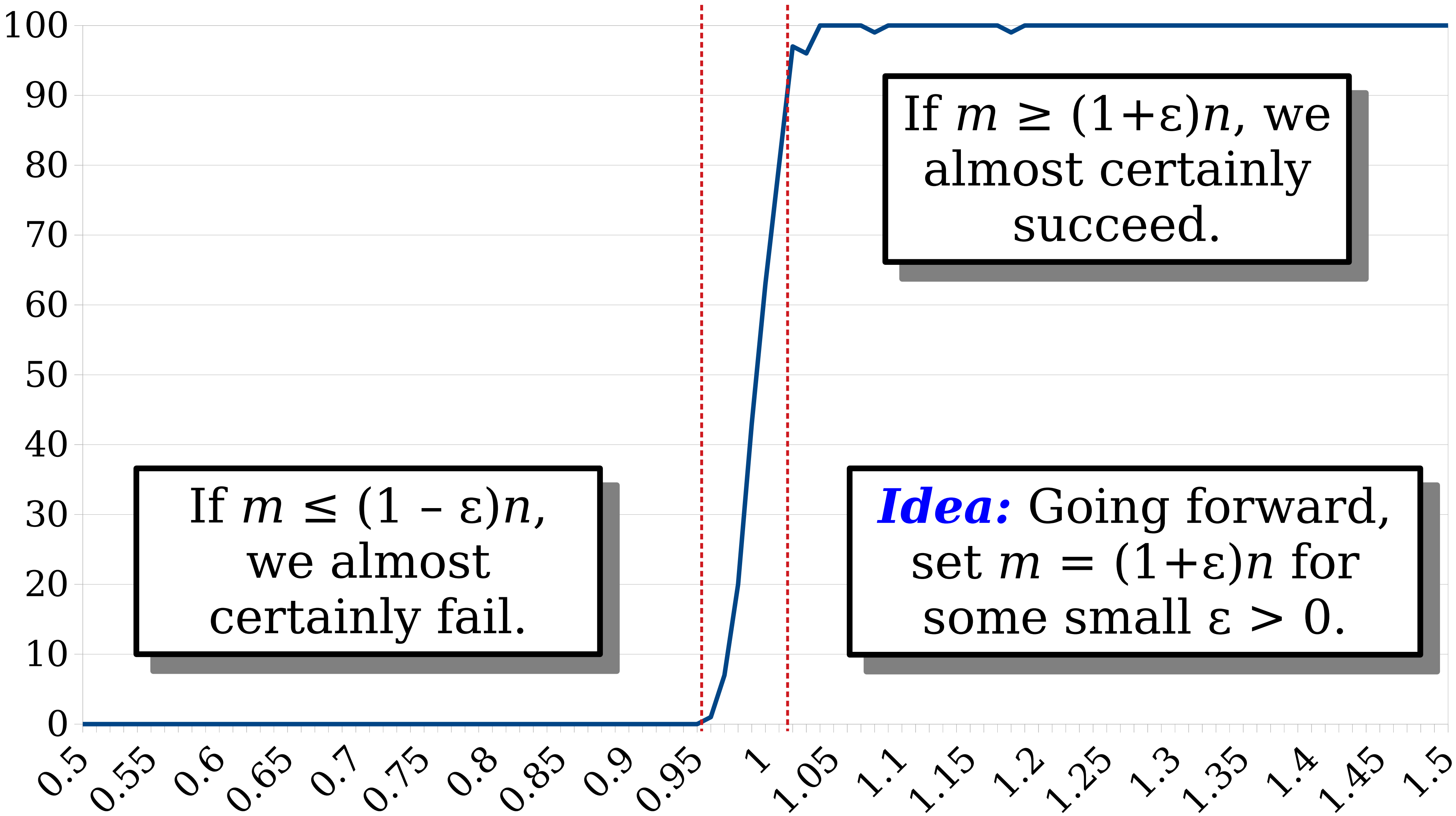
Cuckoo Hashing

- 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 – do you see why?



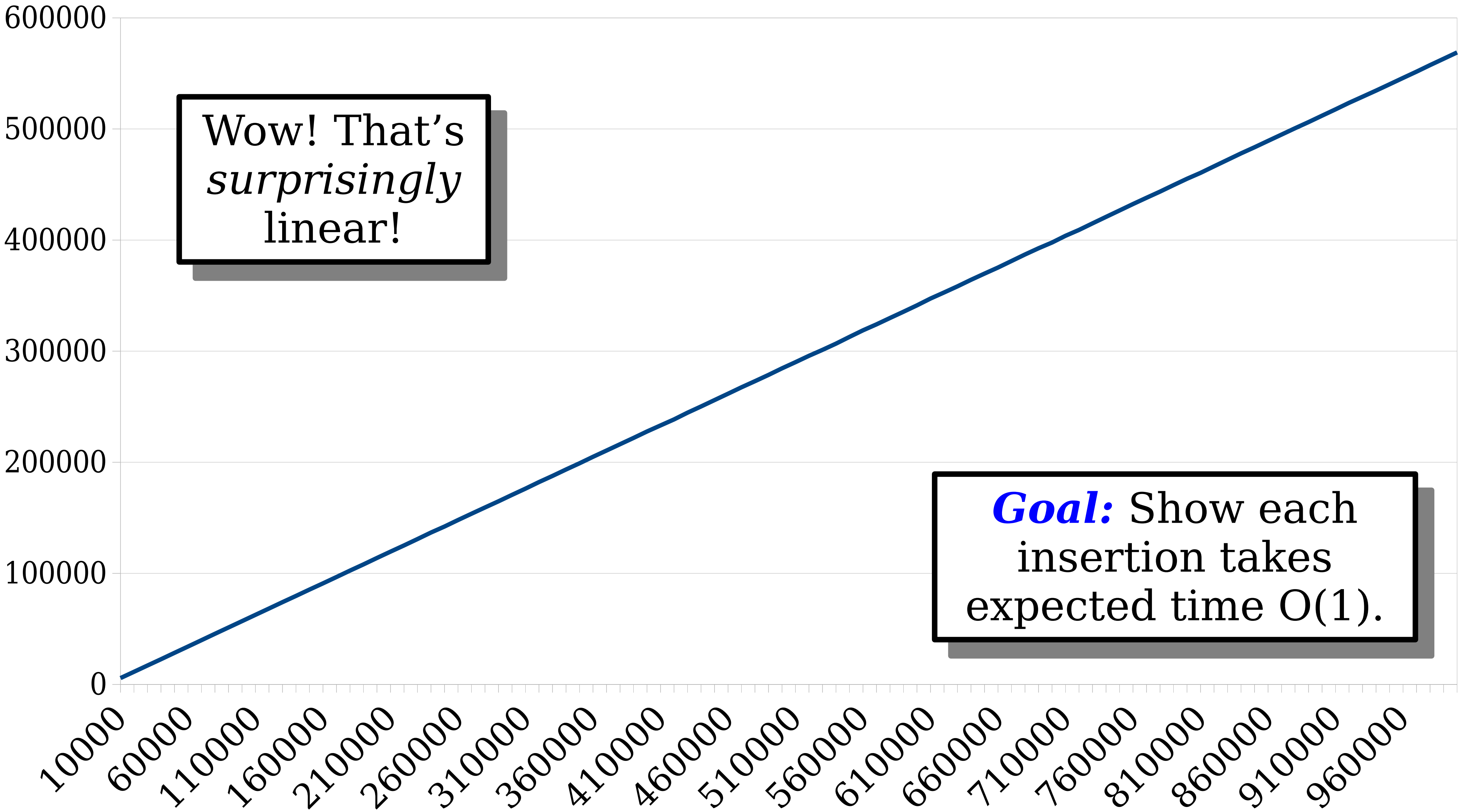
How efficient is cuckoo hashing?

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



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

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



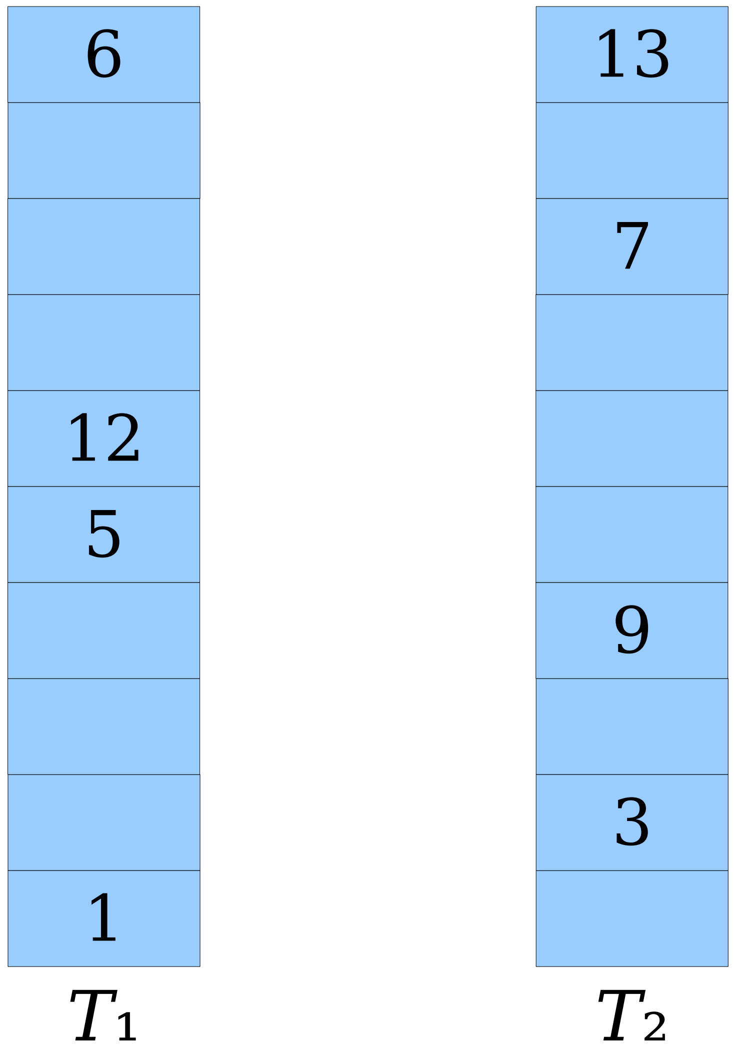
Suppose we store n total elements with $m = (1+\varepsilon)n$.

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 $\varepsilon > 0$.

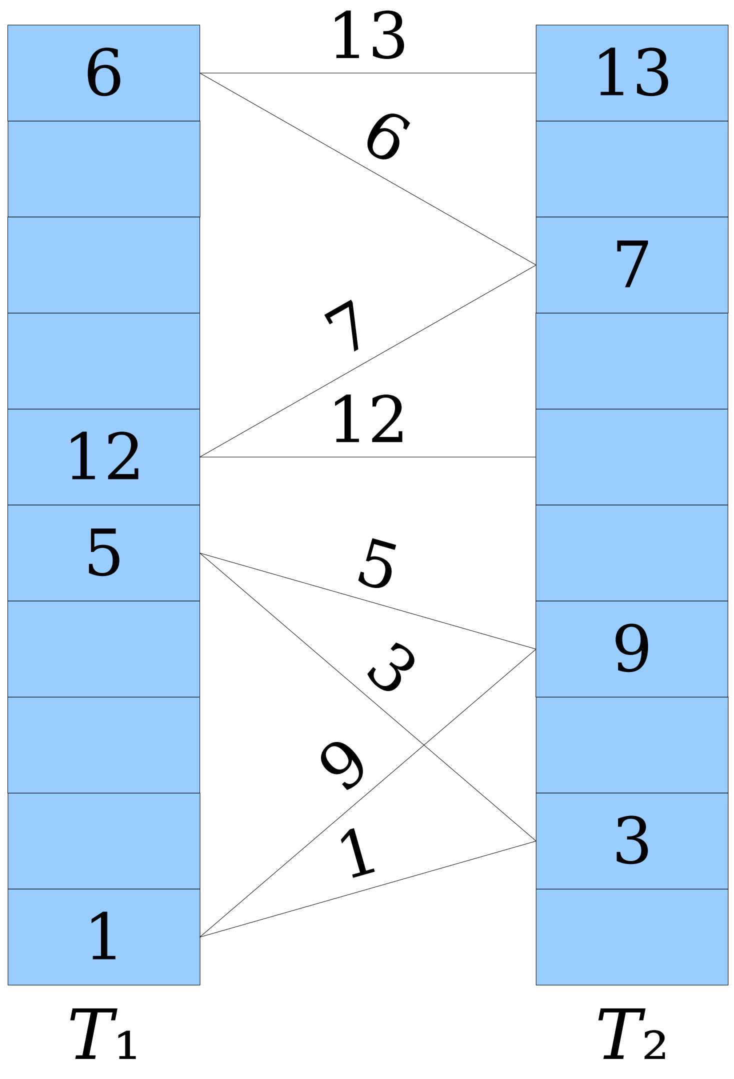
Analyzing Cuckoo Hashing

- The analysis of cuckoo hashing 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.

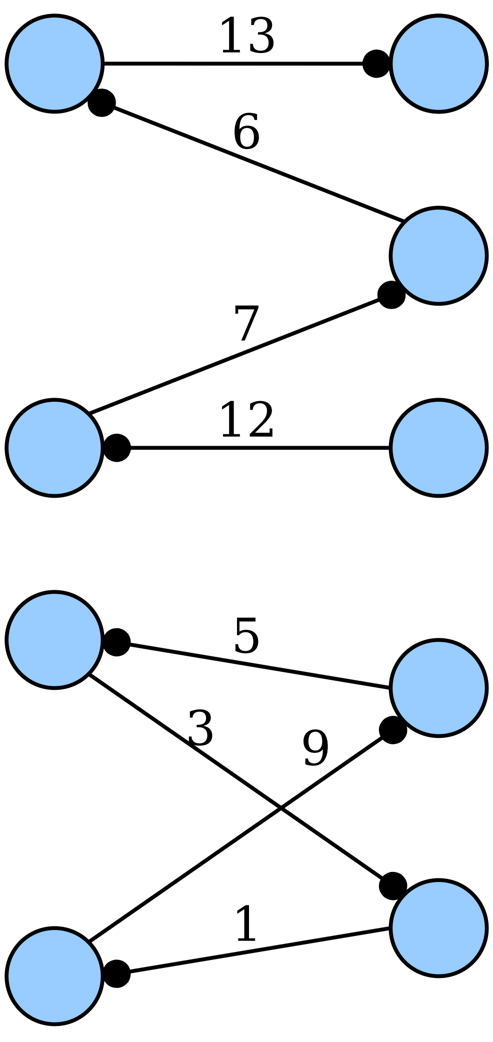


The Cuckoo Graph

- The ***cuckoo graph*** is a 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.

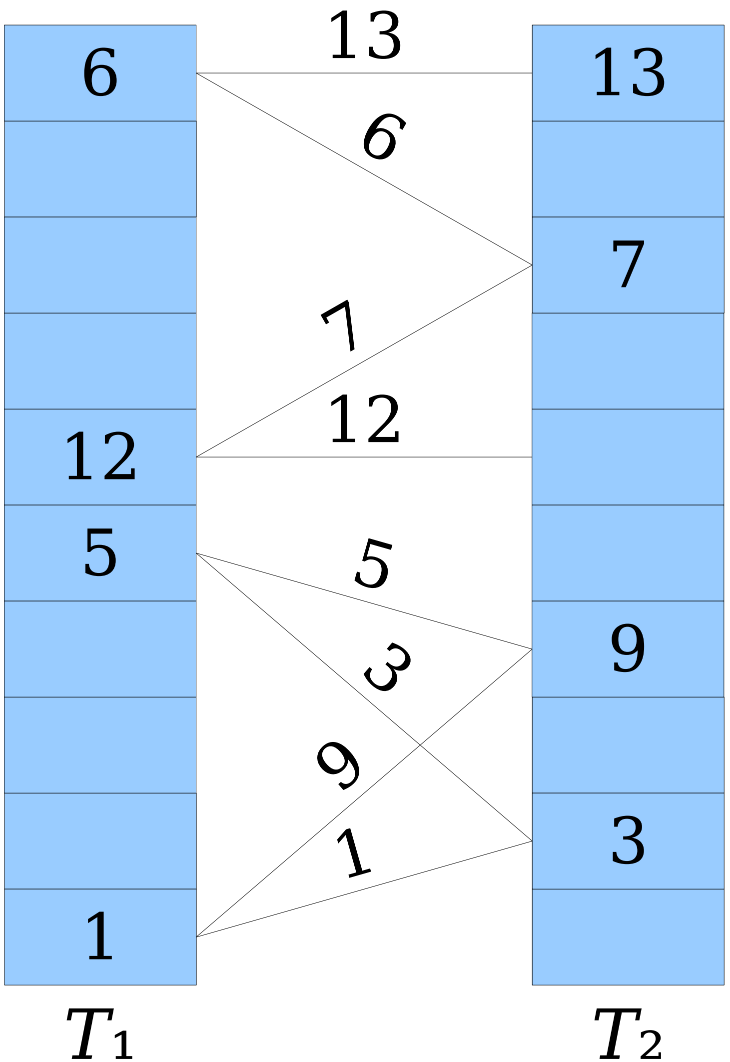


The Cuckoo Graph

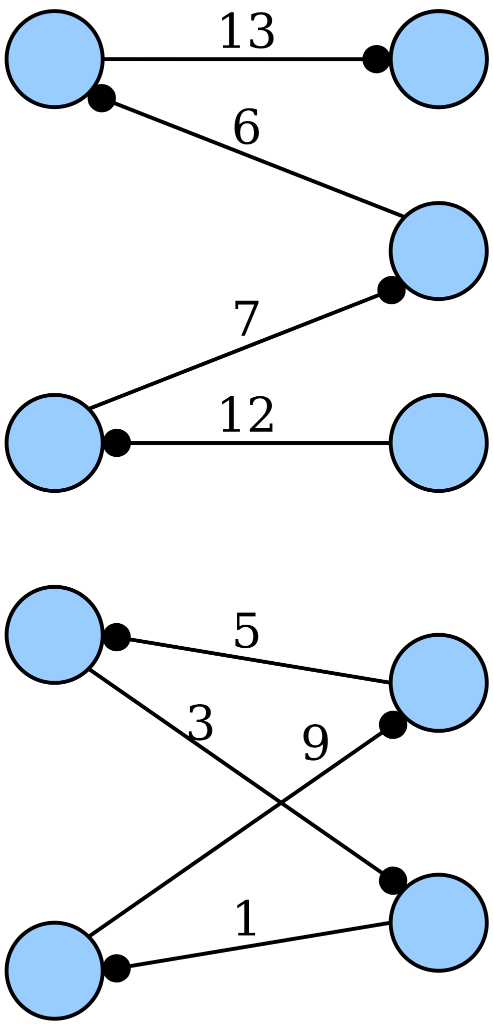


Circles indicate which slots elements are stored in.

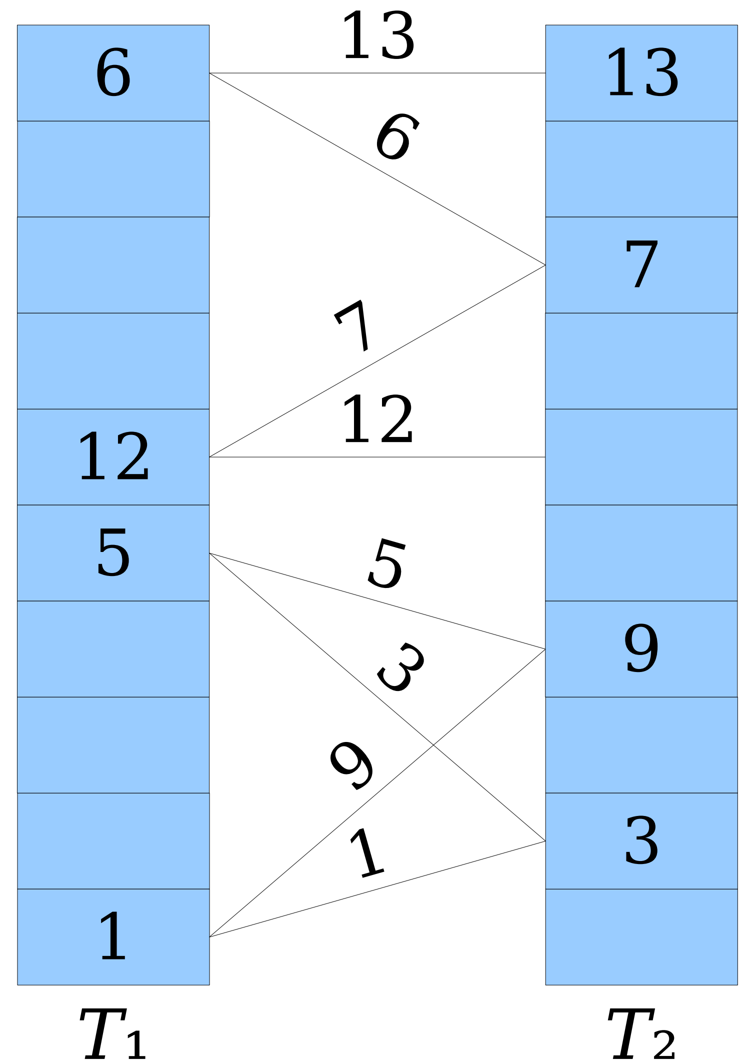
Each node has at most one circle touching it.



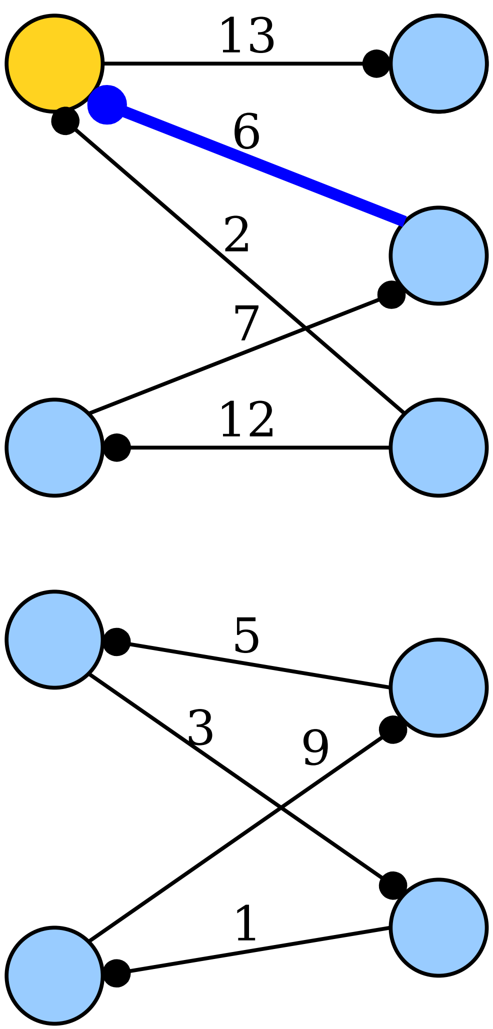
The Cuckoo Graph



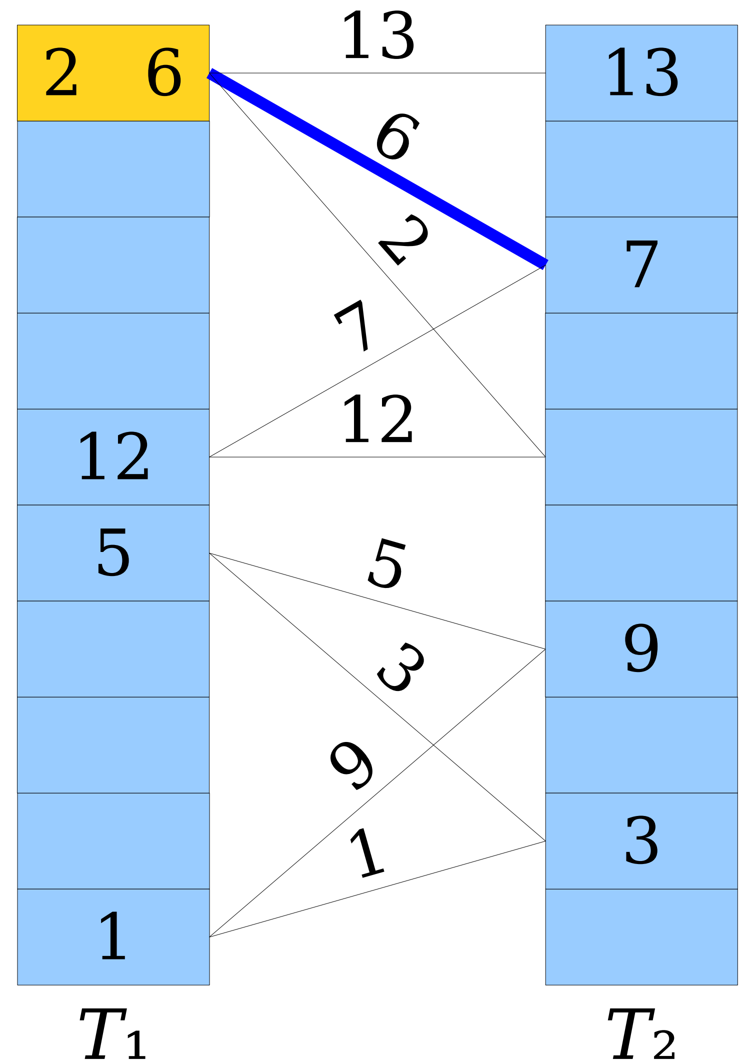
Insertions correspond to sequences of flipping edges.



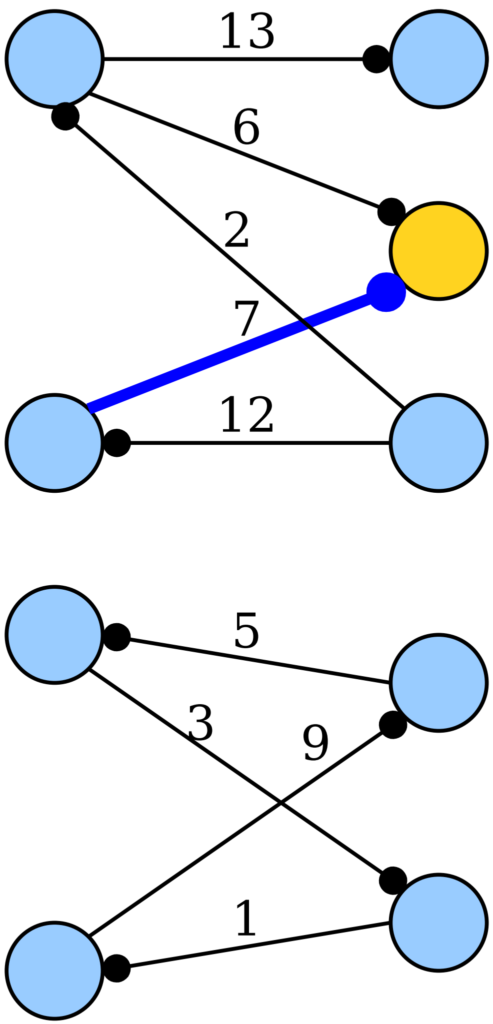
The Cuckoo Graph



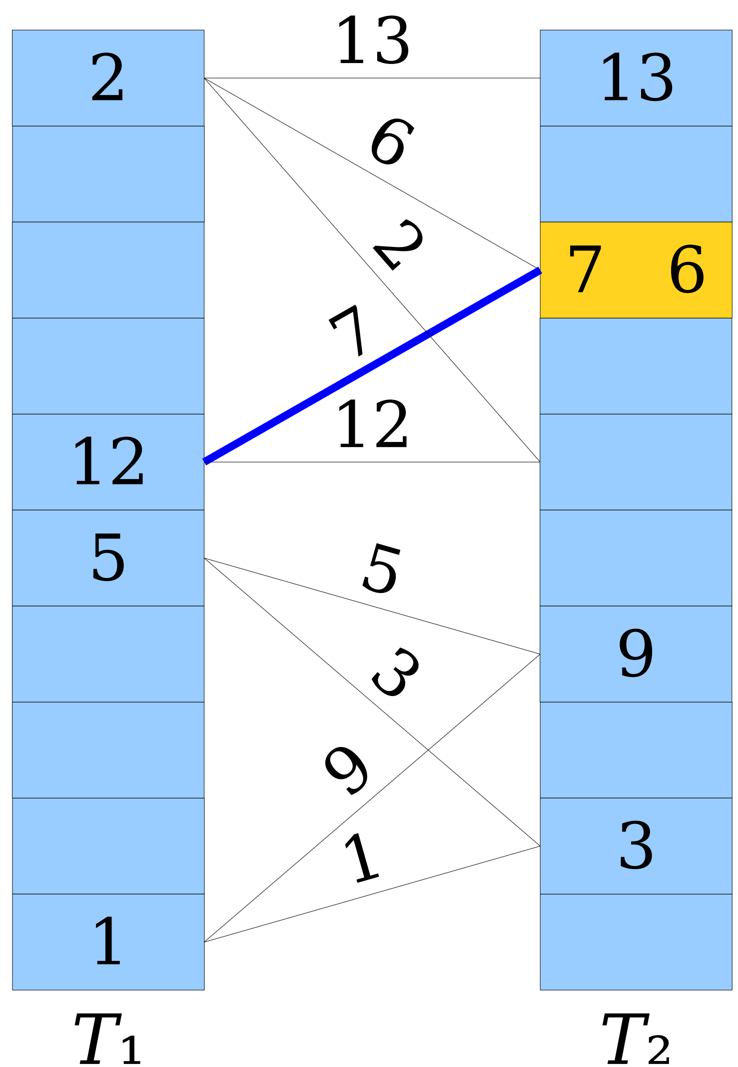
Insertions correspond to sequences of flipping edges.



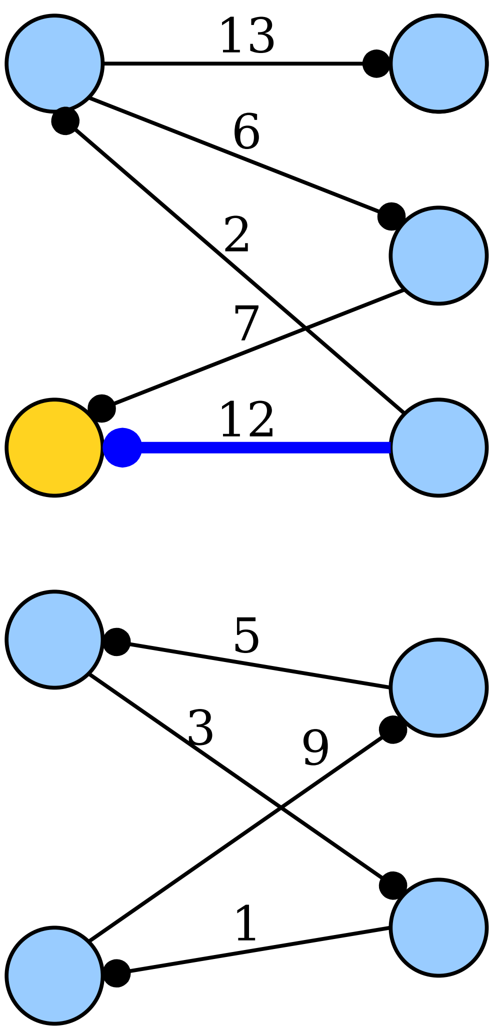
The Cuckoo Graph



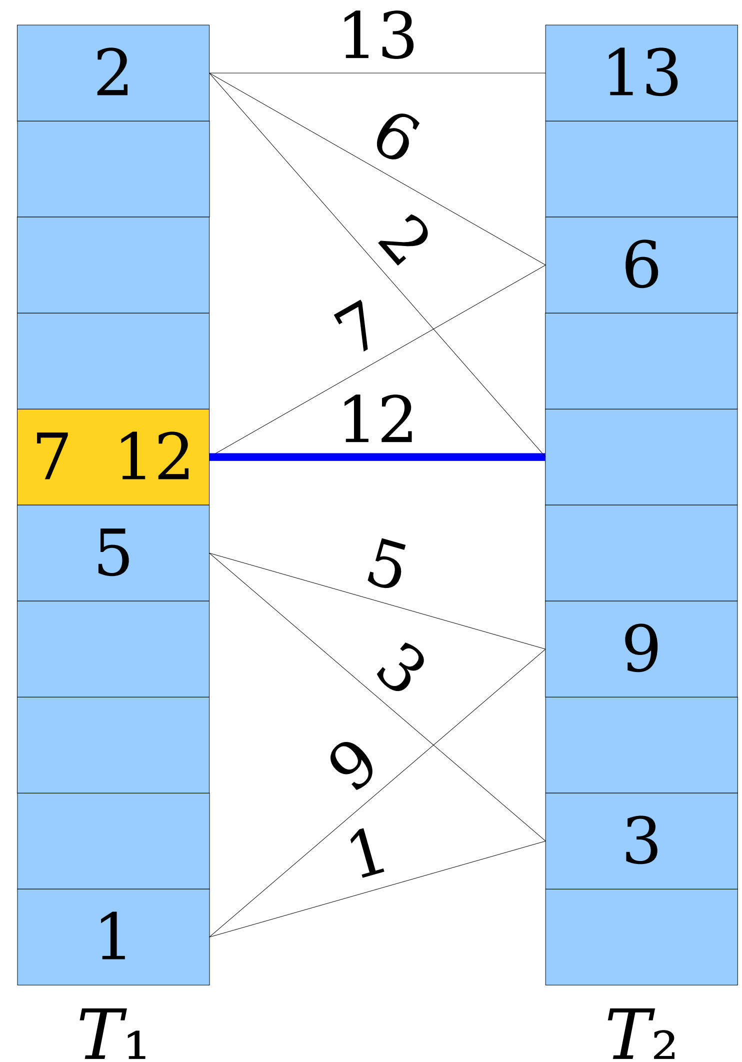
Insertions correspond to sequences of flipping edges.



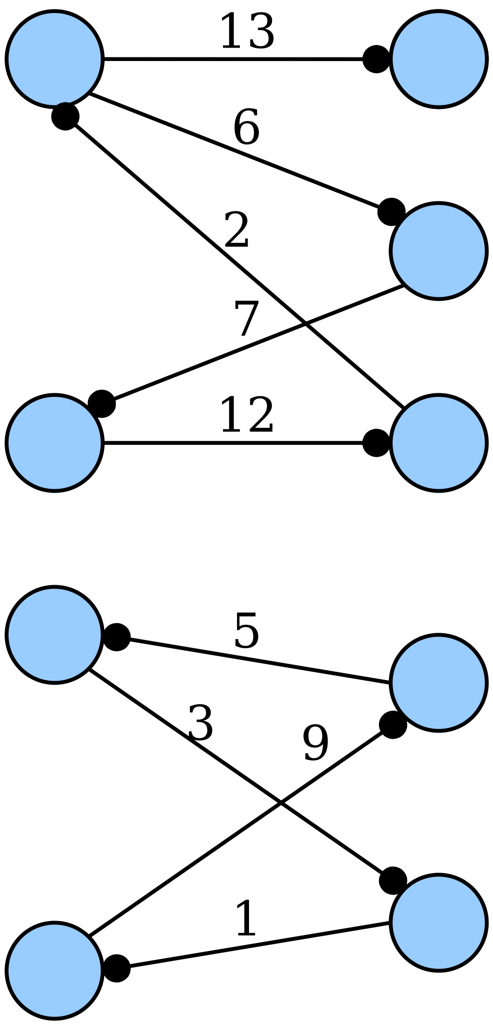
The Cuckoo Graph



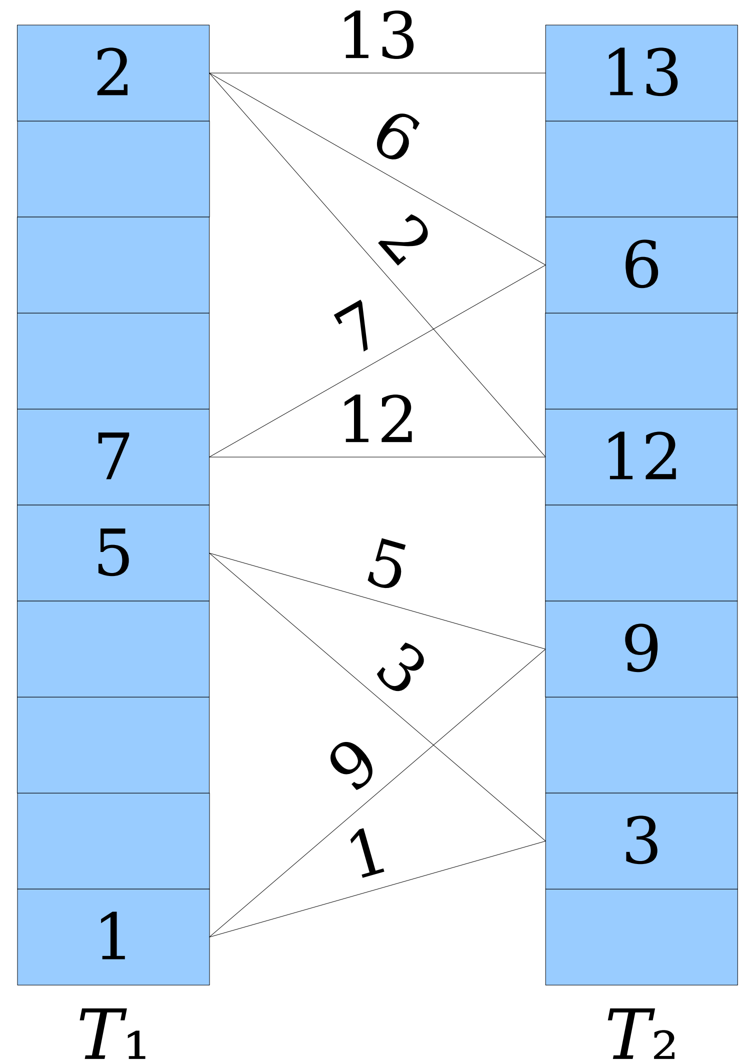
Insertions correspond to sequences of flipping edges.



The Cuckoo Graph



Insertions correspond to sequences of flipping edges.

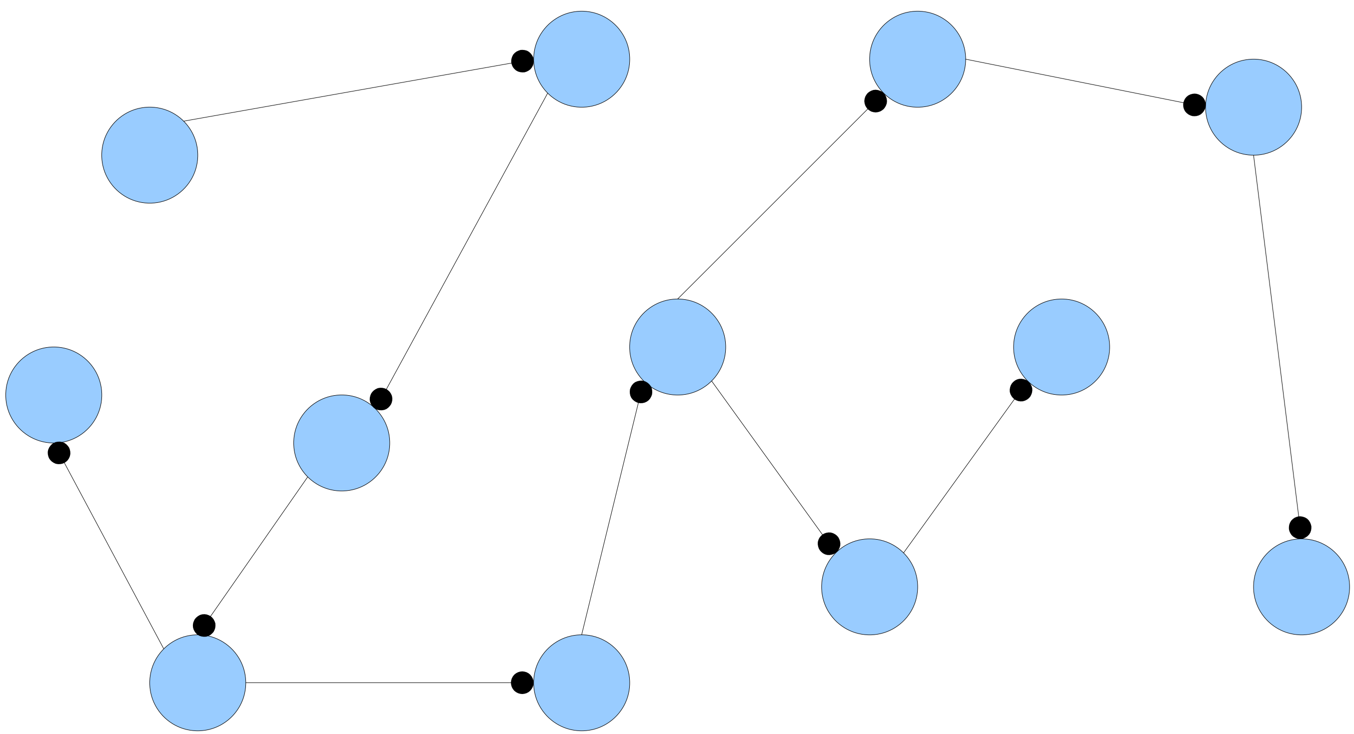


The Cuckoo Graph

- ***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.

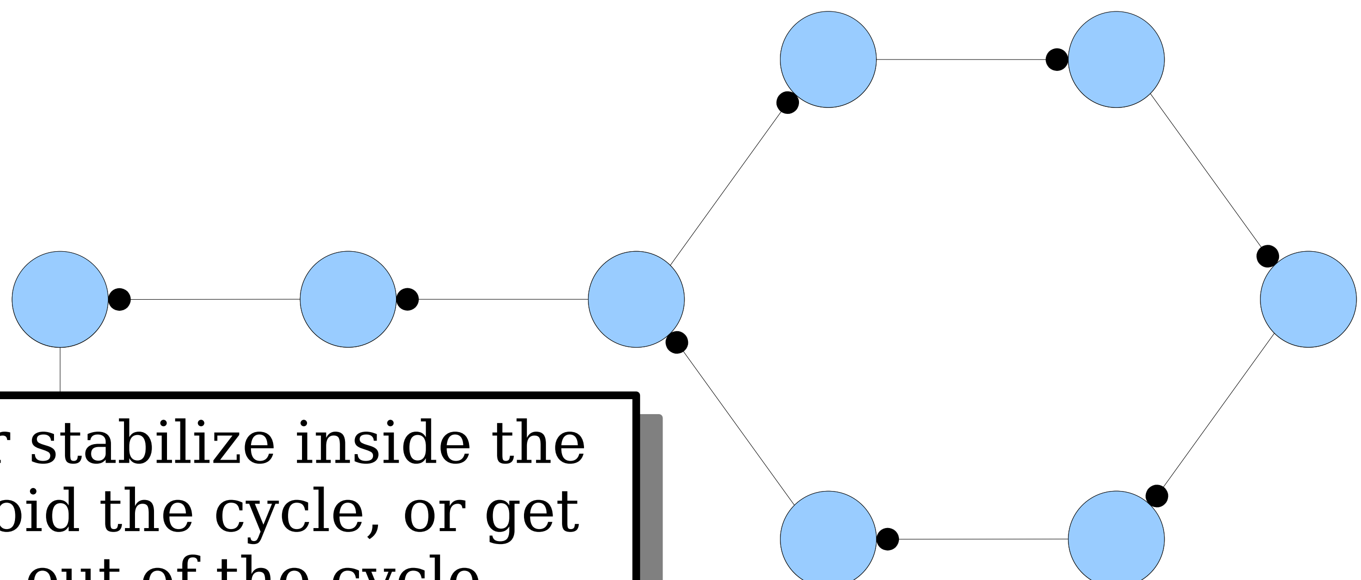
The Cuckoo Graph

- ***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

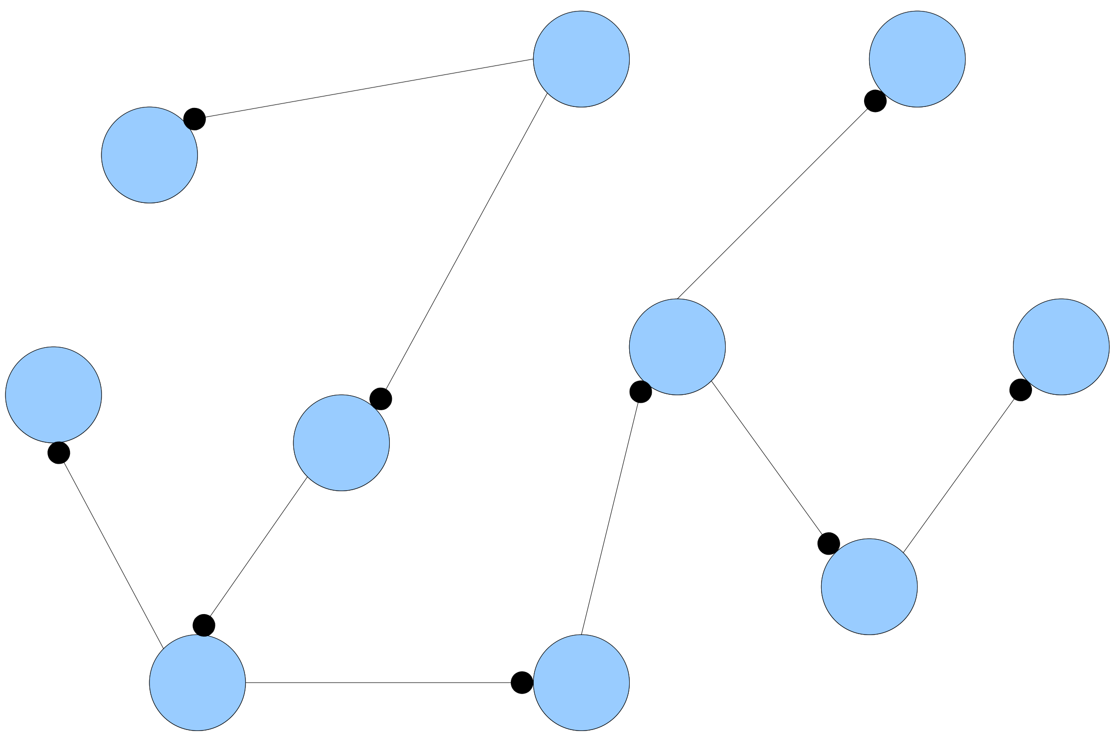
- ***Claim 1:*** If *x* is inserted into a cuckoo hash table, the insertion succeeds if the connected component containing *x* contains either no cycles or only one cycle.



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

The Cuckoo Graph

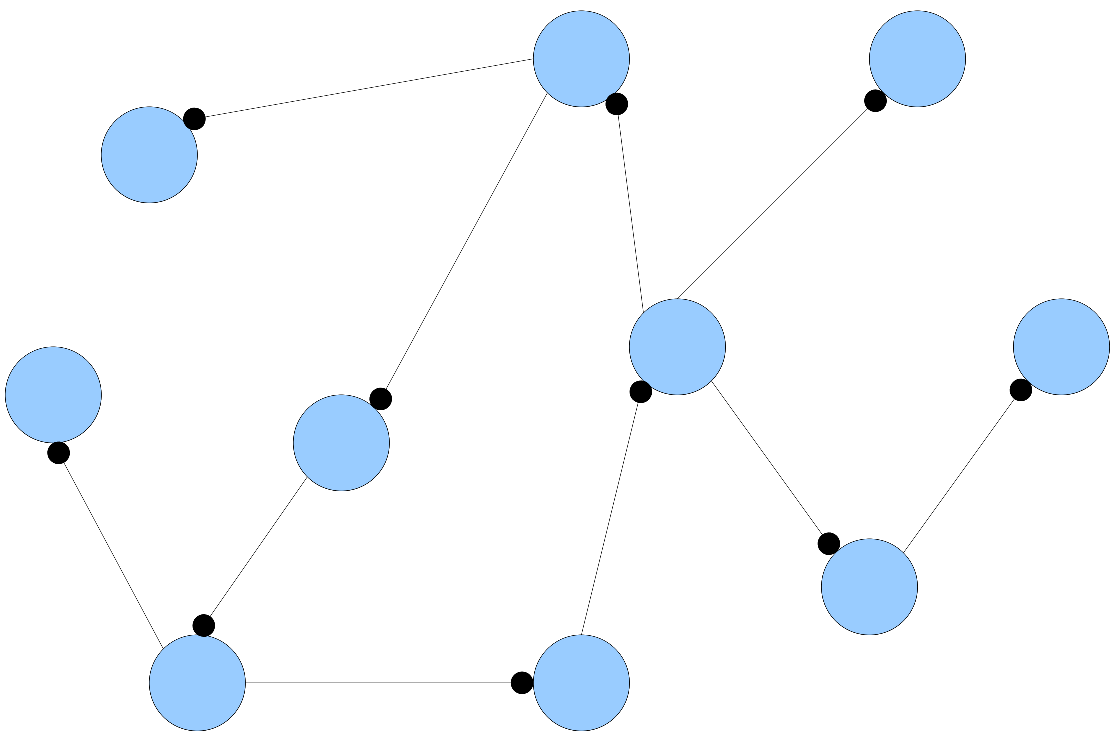
- ***Claim 2:*** If x is inserted into a cuckoo hash table, the insertion fails if the connected component containing x contains more than one cycle.



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

The Cuckoo Graph

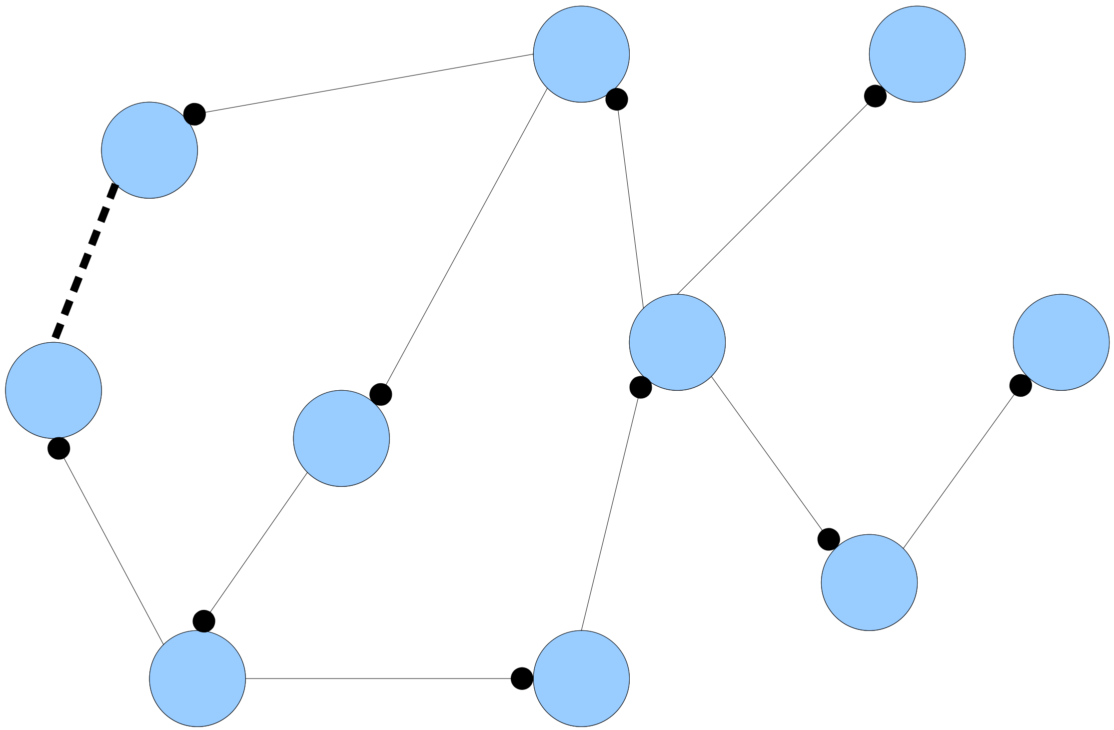
- ***Claim 2:*** If x is inserted into a cuckoo hash table, the insertion fails if the connected component containing x contains more than one cycle.



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 circles to place at most one circle 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 an insert fails?

(This lets us determine how much average work we do on an insertion.)

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.

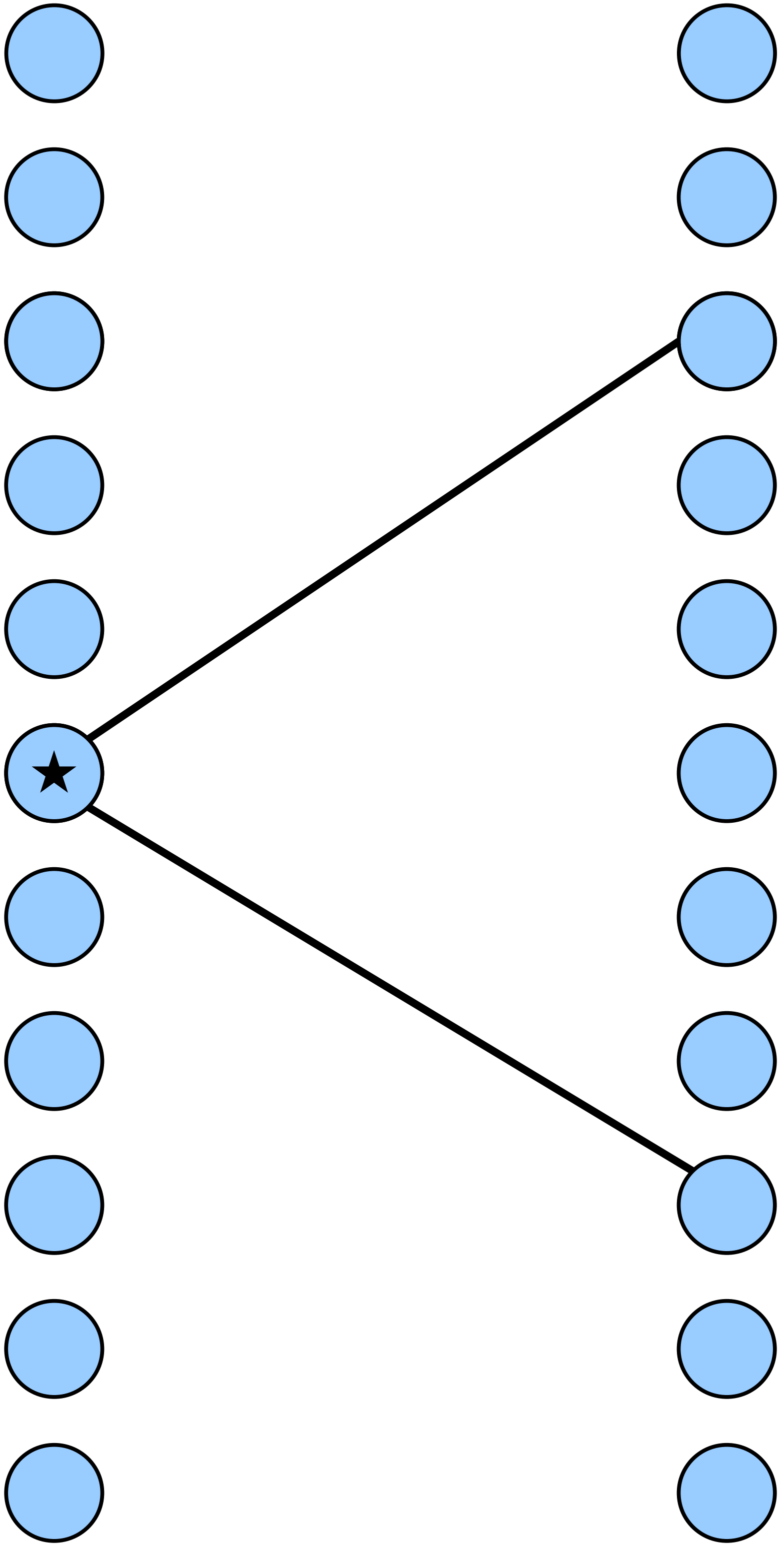
Pick some starting table slot.

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.

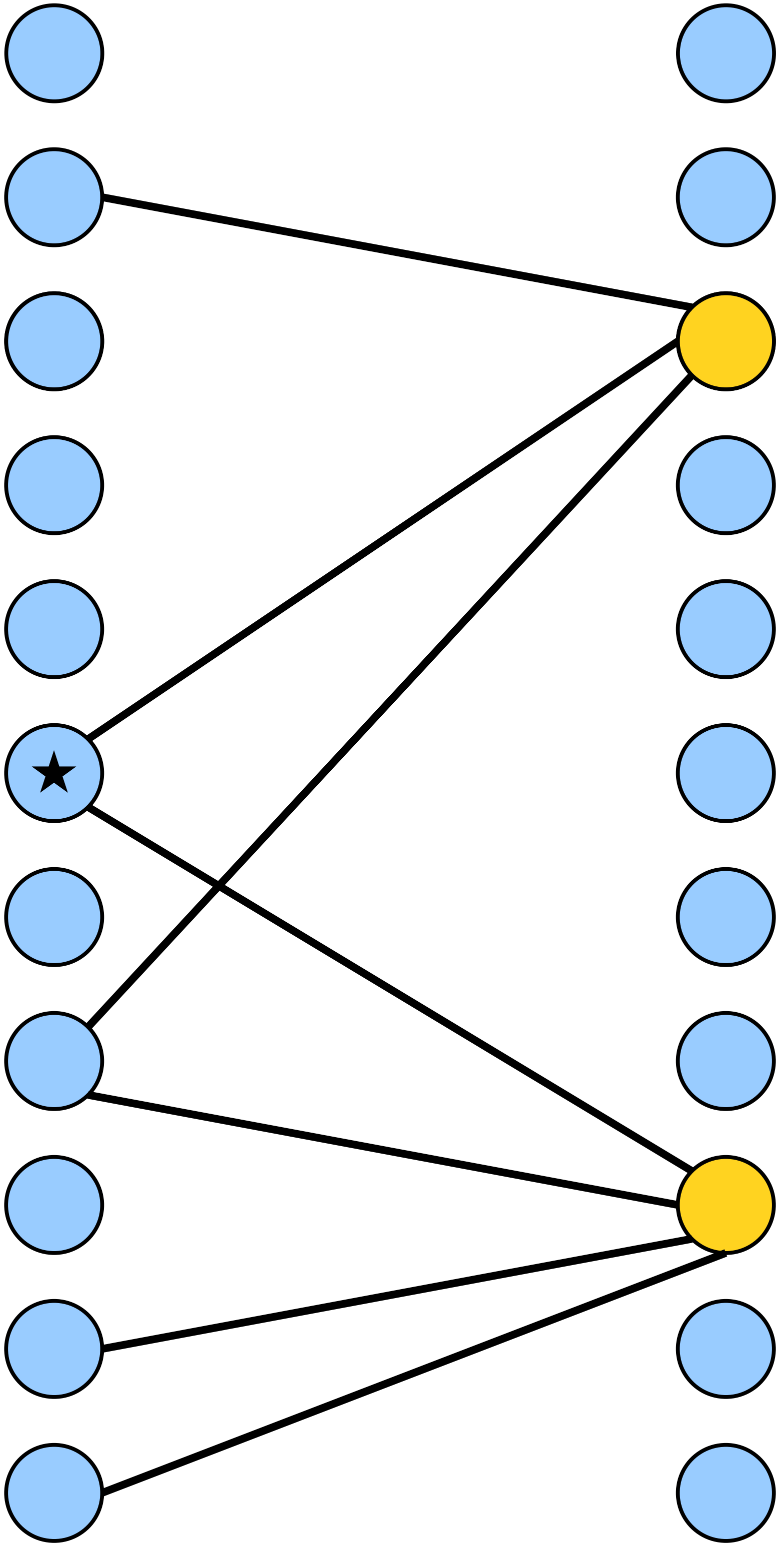
Each new node kinda sorta ish also touches a number of new nodes on the other side that can be modeled as a Binom(n, $^1/_m$) variable.

This ignores double-counting nodes.

This ignores existing edges.

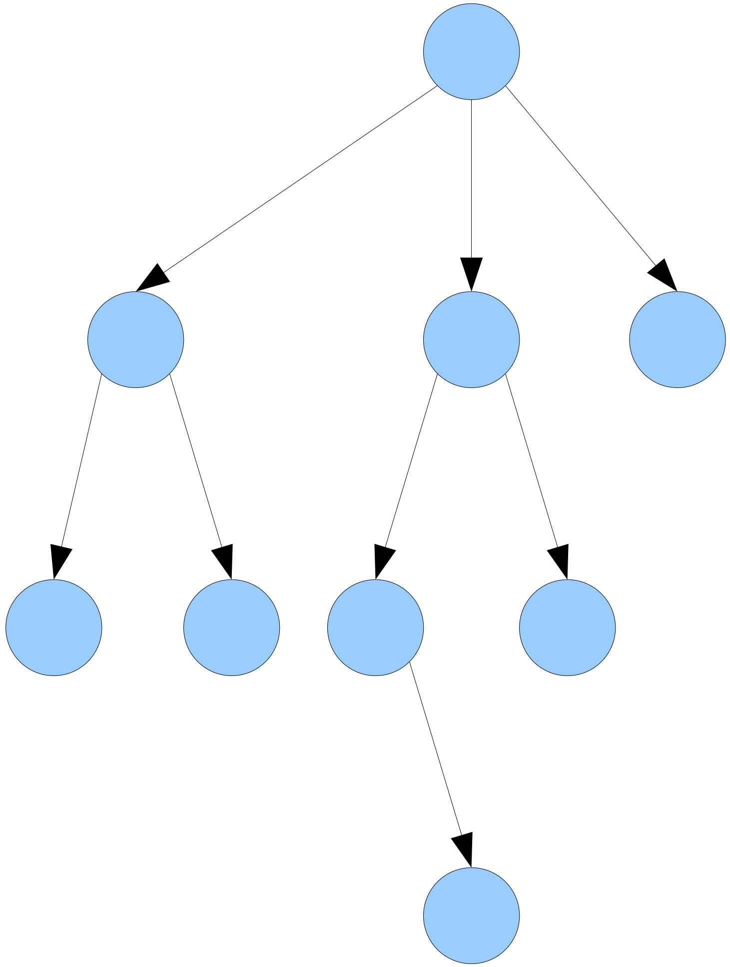
This ignores correlations between edge counts.

However, it conservatively bounds the next BFS step.



Modeling the BFS

- ***Idea:*** Count nodes in a connected component by simulating a BFS tree, where the number of children of each node is a Binom(n, $^1/_m$) variable.
  - 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?

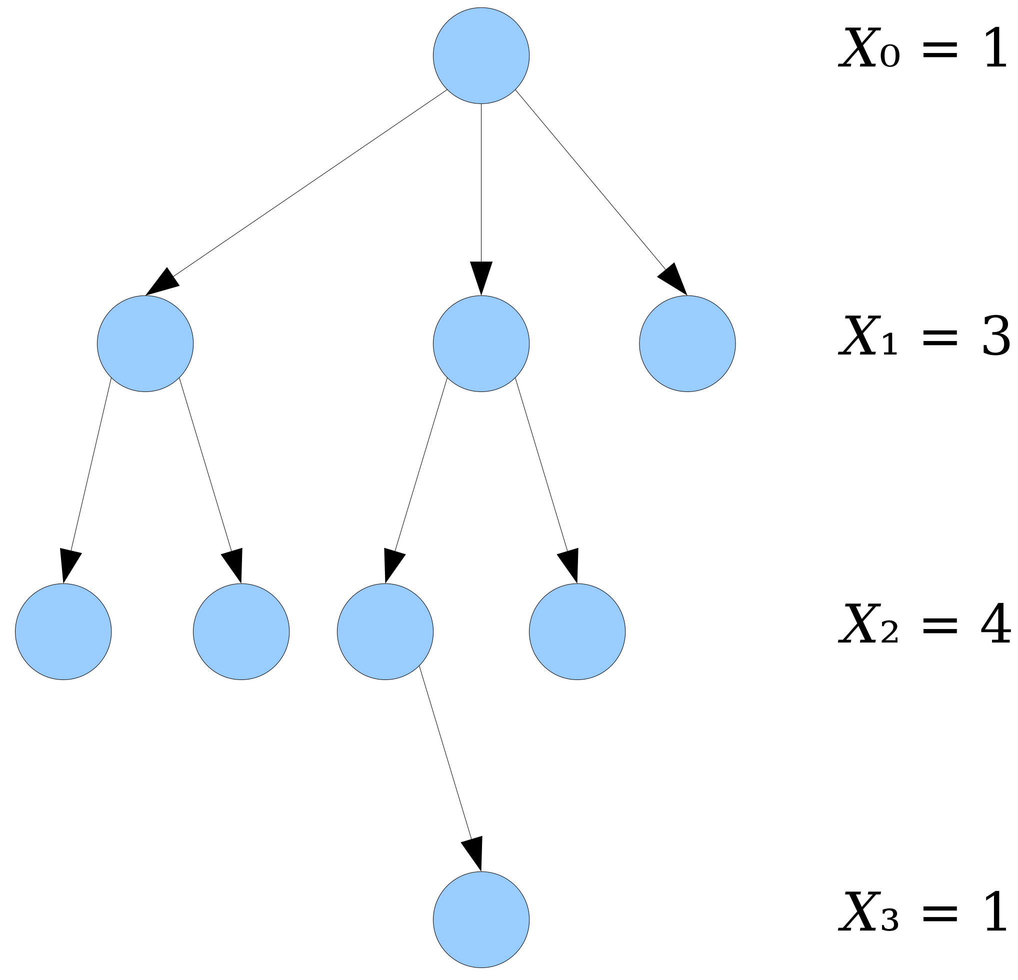


Modeling the BFS

- Denote by X_k the number of nodes at level n. 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. Binom(n, $^1/_m$) variable.

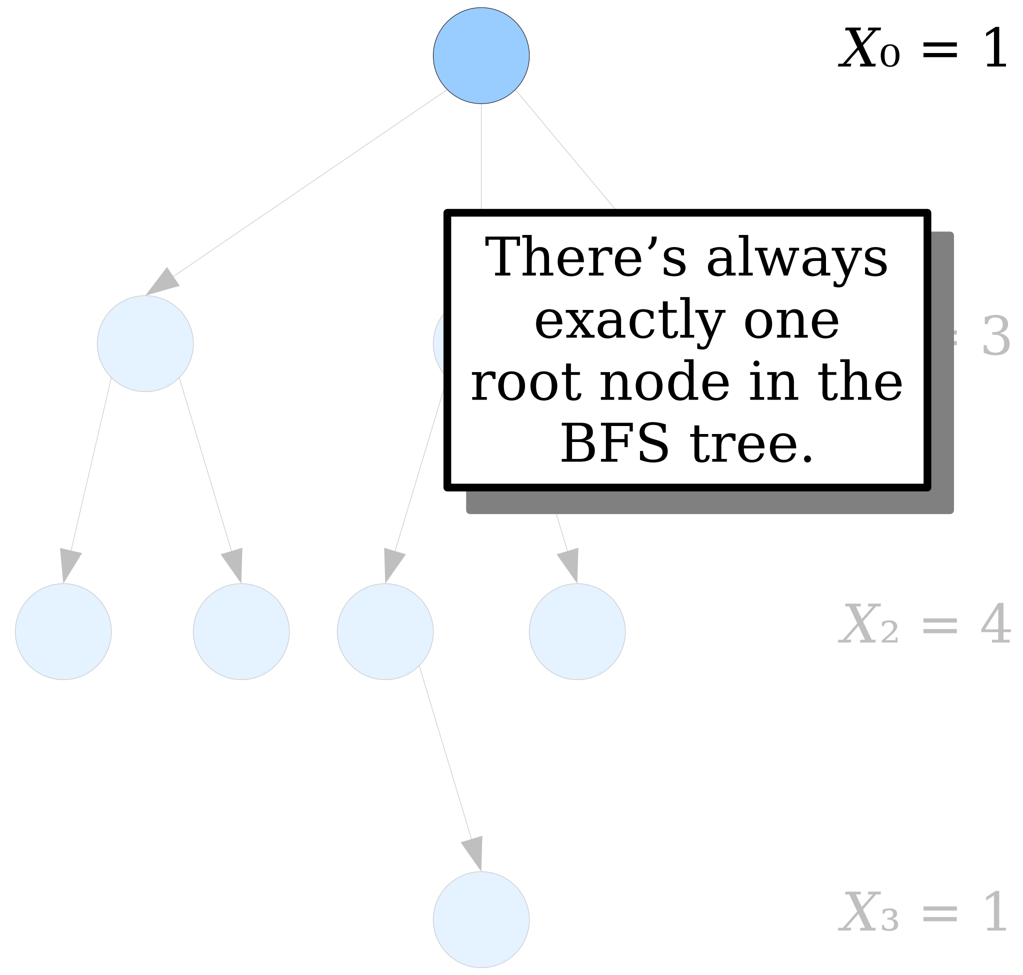


Modeling the BFS

- Denote by X_k the number of nodes at level n. 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(n, $^1/_m$) variable.



Modeling the BFS

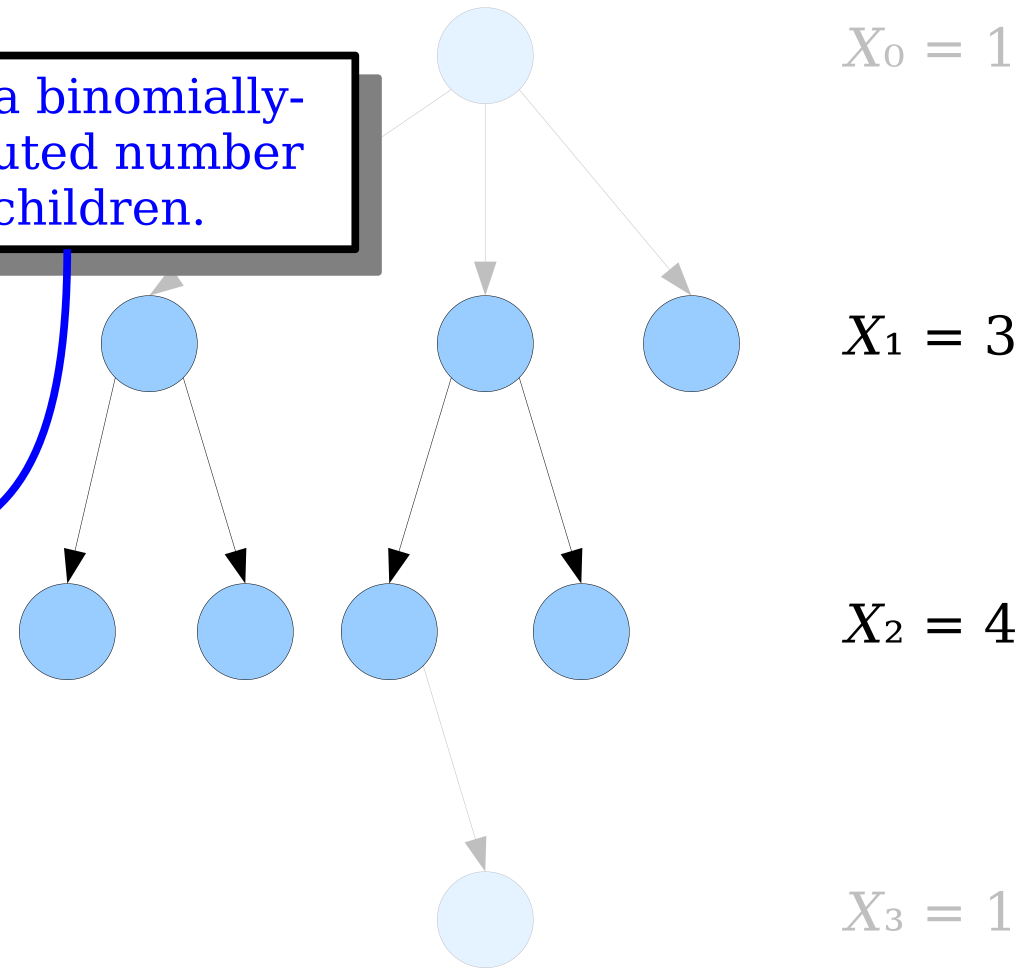
- Denote by X_k the number of nodes at level n. This gives a series of random va… , X_2, …
- Th… are de… by the following randomized recurrence relation:

Each of the X_k nodes in layer k…

… has a binomially-distributed number of children.

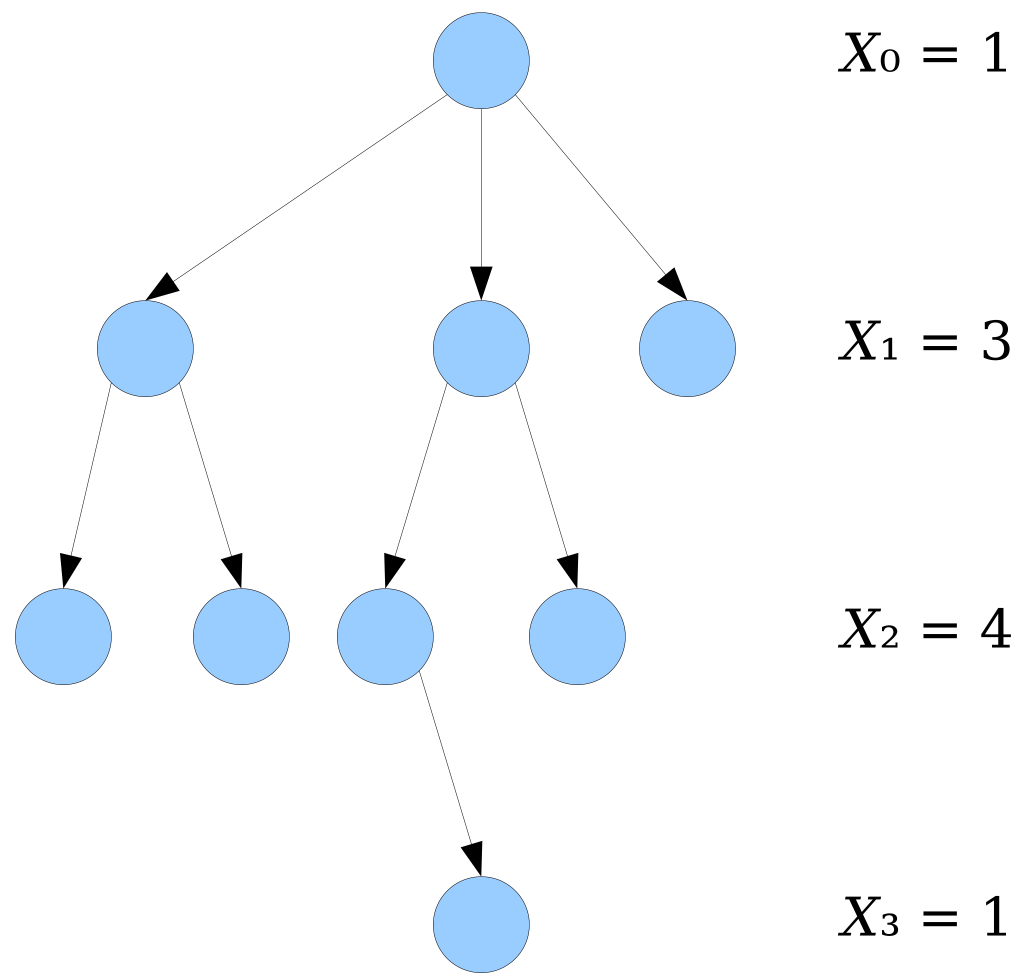
$$X_0 = 1 \qquad X_{k+1} = \sum_{i=1}^{X_k} \xi_{i,k}$$

- Here, each $\xi_{i,k}$ is an i.i.d. Binom(n, $1/m$) variable.



Modeling the BFS

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



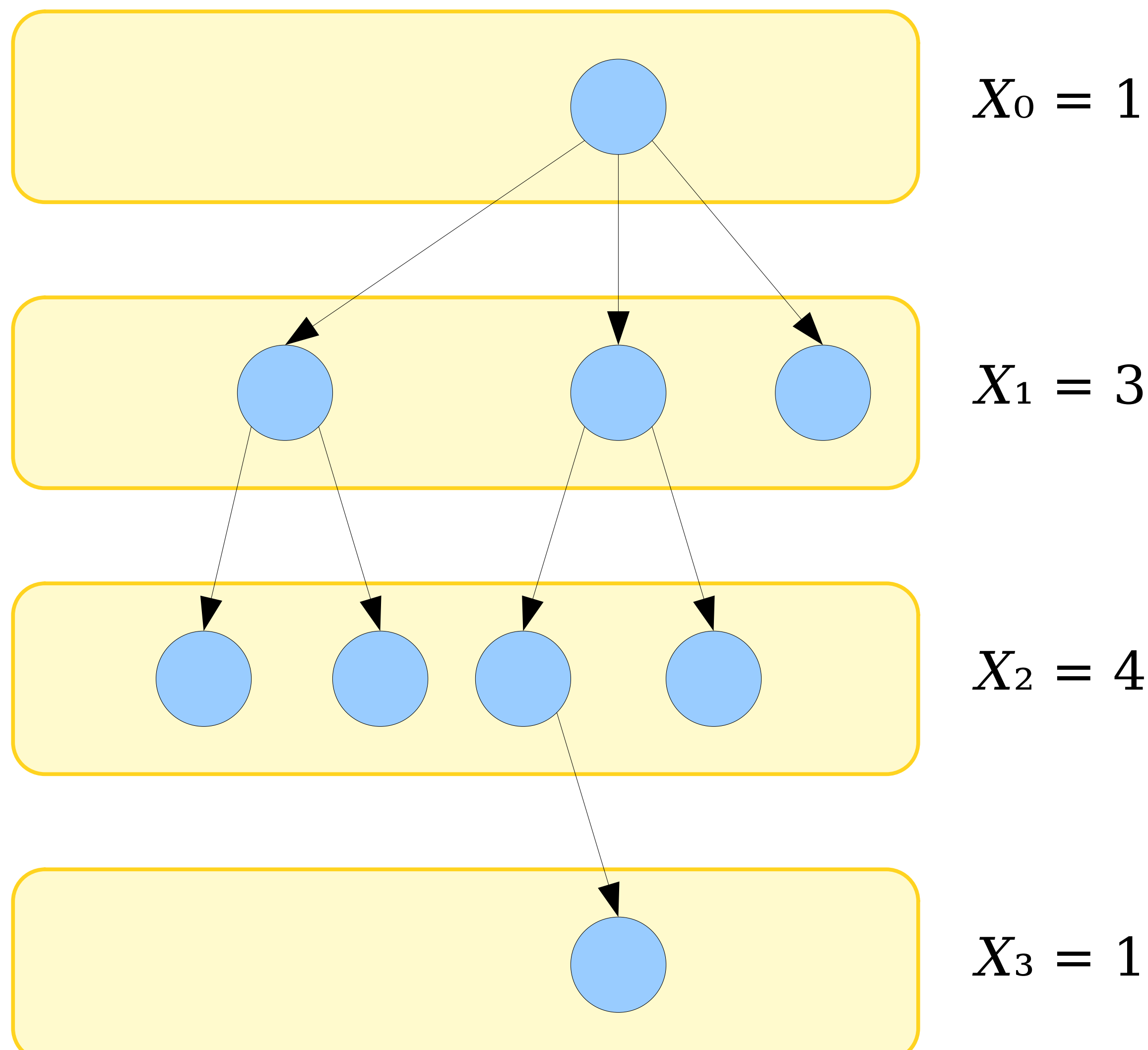
Modeling the BFS

There is always one node here.

On expectation, we'd find n/m nodes here.

On expectation, we'd find $(n/m)^2$ nodes here.

On expectation, we'd find $(n/m)^3$ nodes here.



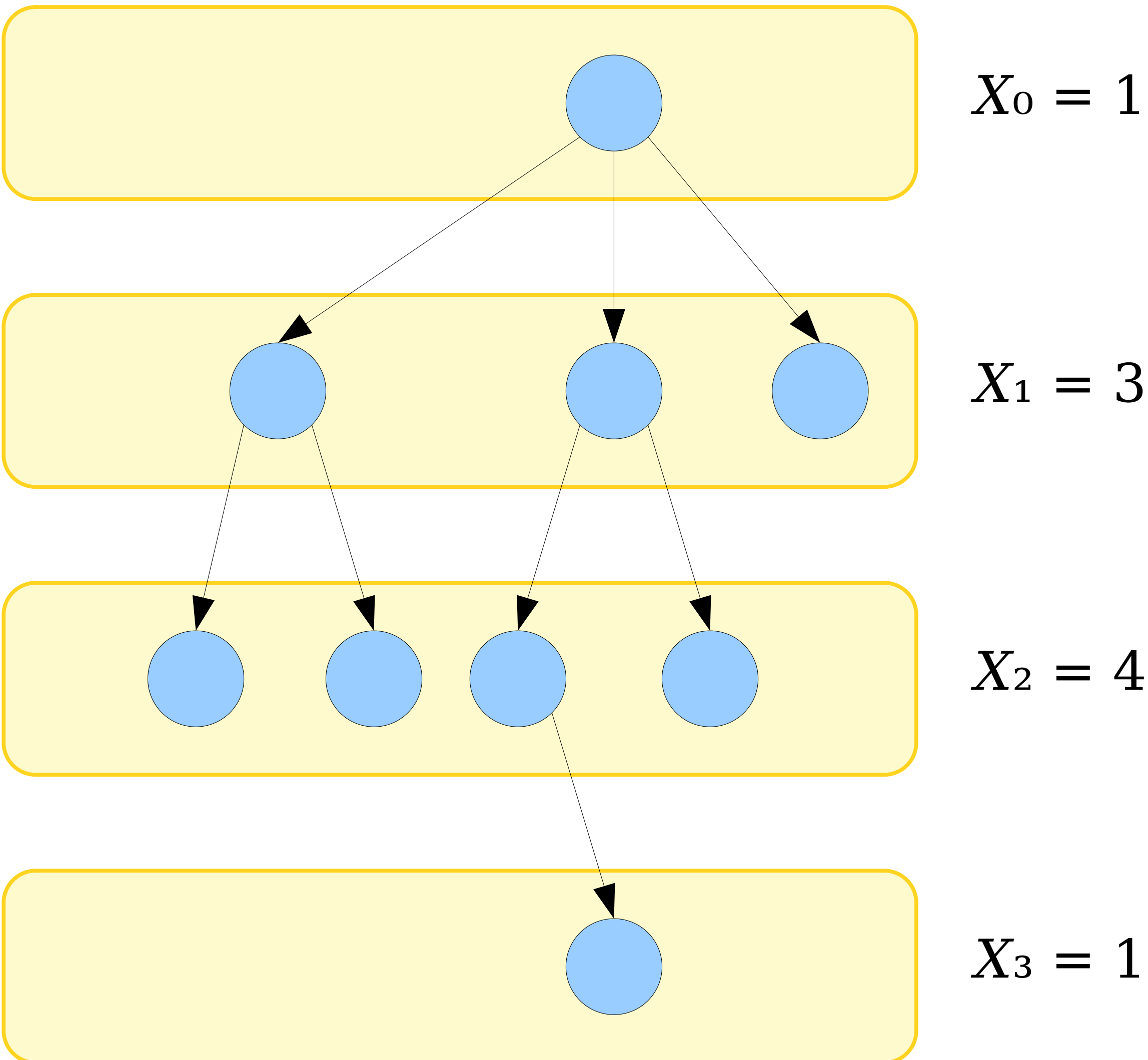
Modeling the BFS

Lemma: $E[X_k] = ({}^{n}/_{m})^k$.

Proof Idea: Show that

$$E[X_{k+1}] = ({}^{n}/_{m})\ E[X_k]$$

and apply induction.



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

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

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

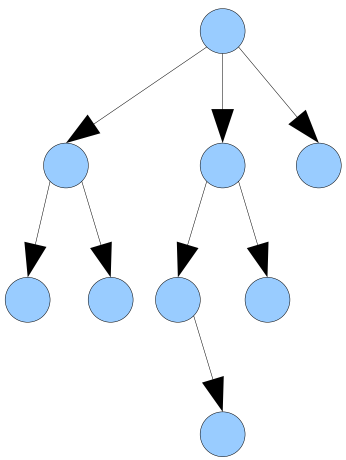
However, we can use the ***law of total expectation:***

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

$$X_0 = 1$$

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

$$\xi_{i,k} \sim \text{Binom}\left(n, \frac{1}{m}\right)$$



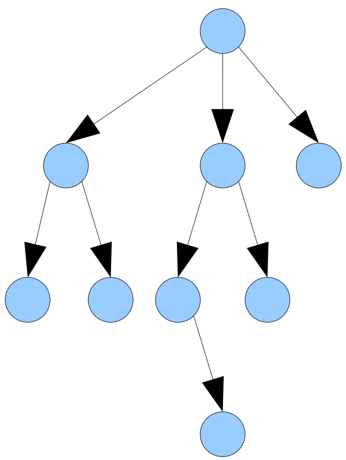
$$
\begin{aligned}
\mathrm{E}[X_{k+1}] &= \mathrm{E}\left[\sum_{i=1}^{X_k} \xi_{i,k}\right] \\
&= \sum_{j=0}^{\infty} \mathrm{E}\left[\sum_{i=1}^{X_k} \xi_{i,k} \;\middle|\; X_k = j\right] \cdot \Pr[X_k = j] \\
&= \sum_{j=0}^{\infty} \mathrm{E}\left[\sum_{i=1}^{j} \xi_{i,k} \;\middle|\; X_k = j\right]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

Well, that makes things easier!

$$X_0 = 1$$

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

$$\xi_{i,k} \sim \mathrm{Binom}\left(n, \frac{1}{m}\right)$$



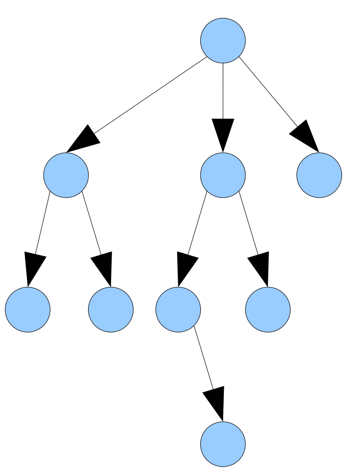
$$
\begin{aligned}
\mathrm{E}[X_{k+1}] &= \mathrm{E}\left[\sum_{i=1}^{X_k} \xi_{i,k}\right] \\
&= \sum_{j=0}^{\infty} \mathrm{E}\left[\sum_{i=1}^{X_k} \xi_{i,k} \;\middle|\; X_k = j\right] \cdot \Pr[X_k = j] \\
&= \sum_{j=0}^{\infty} \mathrm{E}\left[\sum_{i=1}^{j} \xi_{i,k} \;\middle|\; X_k = j\right]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

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

$$X_0 = 1$$

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

$$\xi_{i,k} \sim \mathrm{Binom}\left(n, \frac{1}{m}\right)$$



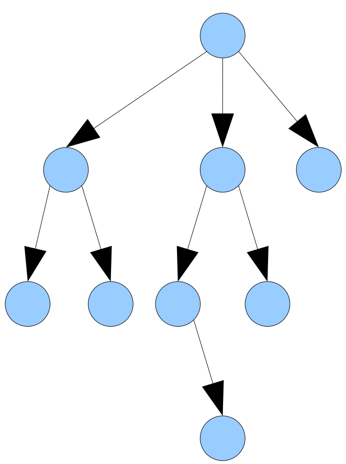
$$
\begin{aligned}
\mathrm{E}[X_{k+1}] &= \mathrm{E}\left[\sum_{i=1}^{X_k} \xi_{i,k}\right] \\
&= \sum_{j=0}^{\infty} \mathrm{E}\left[\sum_{i=1}^{X_k} \xi_{i,k} \;\middle|\; X_k = j\right] \cdot \Pr[X_k = j] \\
&= \sum_{j=0}^{\infty} \mathrm{E}\left[\sum_{i=1}^{j} \xi_{i,k} \;\middle|\; X_k = j\right]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

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.

$$X_0 = 1$$

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

$$\xi_{i,k} \sim \mathrm{Binom}\left(n, \frac{1}{m}\right)$$

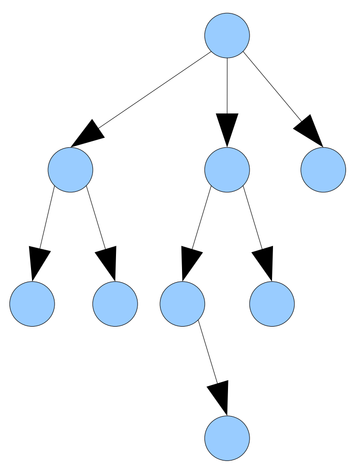


$$
\begin{aligned}
\mathrm{E}[X_{k+1}] &= \mathrm{E}\Big[\sum_{i=1}^{X_k} \xi_{i,k}\Big] \\
&= \sum_{j=0}^{\infty} \mathrm{E}\Big[\sum_{i=1}^{X_k} \xi_{i,k} \;\Big|\; X_k = j\Big] \cdot \Pr[X_k = j] \\
&= \sum_{j=0}^{\infty} \mathrm{E}\Big[\sum_{i=1}^{j} \xi_{i,k} \;\Big|\; X_k = j\Big] \cdot \Pr[X_k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

$$X_0 = 1$$

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

$$\xi_{i,k} \sim \mathrm{Binom}\left(n, \frac{1}{m}\right)$$

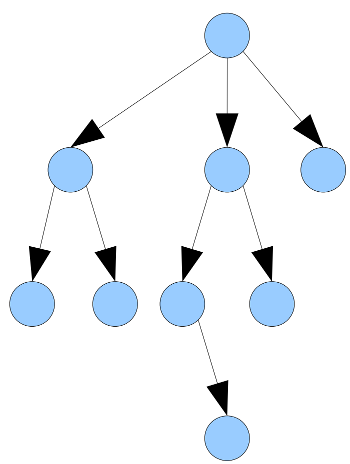


$$
\begin{aligned}
\mathrm{E}[X_{k+1}] &= \mathrm{E}\Big[\sum_{i=1}^{X_k} \xi_{i,k}\Big] \\
&= \sum_{j=0}^{\infty} \mathrm{E}\Big[\sum_{i=1}^{X_k} \xi_{i,k} \;\Big|\; X_k = j\Big] \cdot \Pr[X_k = j] \\
&= \sum_{j=0}^{\infty} \mathrm{E}\Big[\sum_{i=1}^{j} \xi_{i,k} \;\Big|\; X_k = j\Big] \cdot \Pr[X_k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

$$X_0 = 1$$

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

$$\xi_{i,k} \sim \mathrm{Binom}\left(n, \frac{1}{m}\right)$$

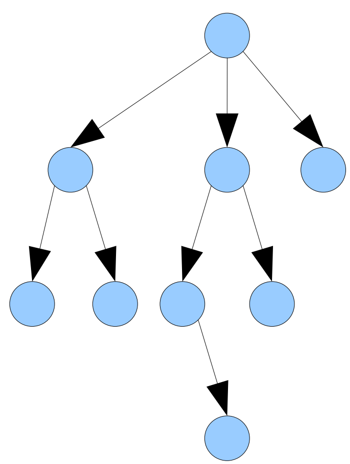


$$
\begin{aligned}
\mathrm{E}[X_{k+1}] &= \mathrm{E}\Big[\sum_{i=1}^{X_k} \xi_{i,k}\Big] \\
&= \sum_{j=0}^{\infty} \mathrm{E}\Big[\sum_{i=1}^{X_k} \xi_{i,k} \;\Big|\; X_k = j\Big] \cdot \Pr[X_k = j] \\
&= \sum_{j=0}^{\infty} \mathrm{E}\Big[\sum_{i=1}^{j} \xi_{i,k} \;\Big|\; X_k = j\Big] \cdot \Pr[X_k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

$$X_0 = 1$$

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

$$\xi_{i,k} \sim \mathrm{Binom}\left(n, \frac{1}{m}\right)$$

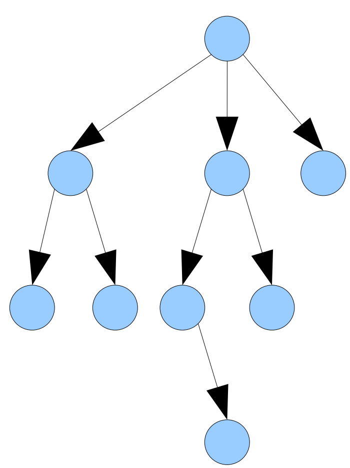


$$
\begin{aligned}
\mathrm{E}[X_{k+1}] &= \mathrm{E}\Big[\sum_{i=1}^{X_k} \xi_{i,k}\Big] \\
&= \sum_{j=0}^{\infty} \mathrm{E}\Big[\sum_{i=1}^{X_k} \xi_{i,k} \;\Big|\; X_k = j\Big] \cdot \Pr[X_k = j] \\
&= \sum_{j=0}^{\infty} \mathrm{E}\Big[\sum_{i=1}^{j} \xi_{i,k} \;\Big|\; X_k = j\Big] \cdot \Pr[X_k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

$$X_0 = 1$$

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

$$\xi_{i,k} \sim \mathrm{Binom}\left(n, \frac{1}{m}\right)$$



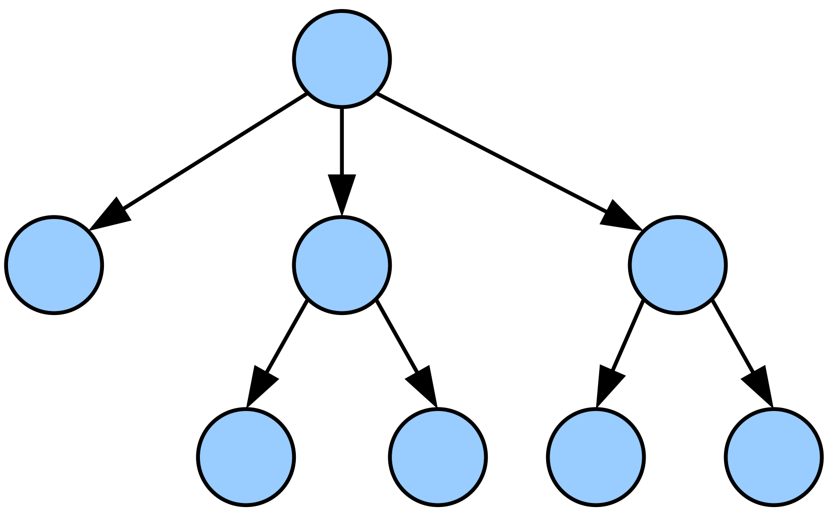
Lemma 1: $E[X_k] = (n/m)^k$.

(Induction and conditional expectation.)

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

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



$$X_0 = 1 \qquad X_{k+1} = \sum_{i=1}^{X_k} \xi_{i,k} \qquad \xi \sim \text{Binom}\left(n, \frac{1}{m}\right)$$

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:

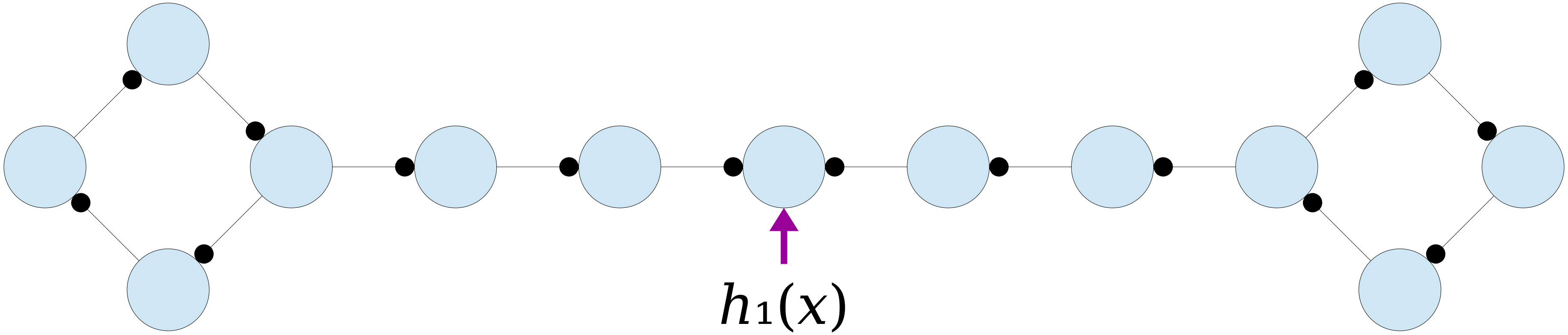
Exploring the Graph Structure

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.

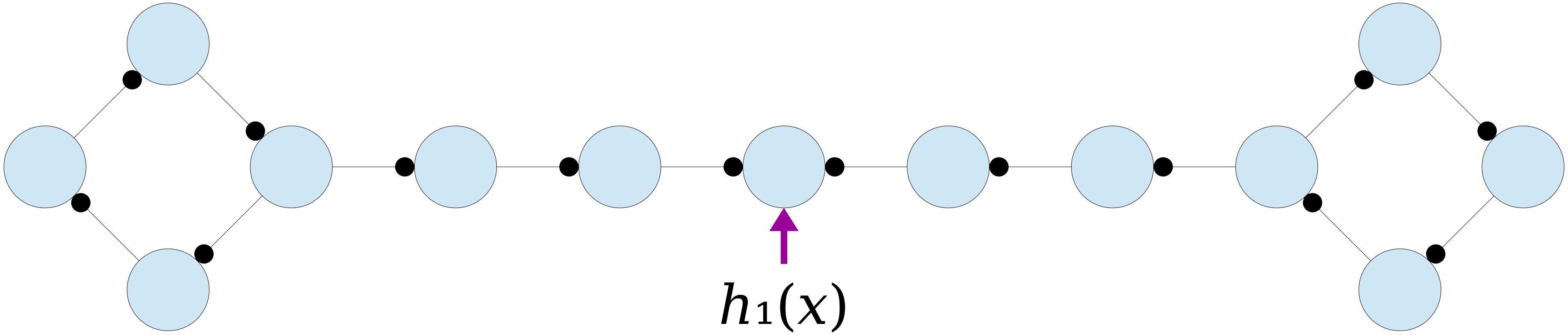
Exploring the Graph Structure

- ***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 clever technique we can use to bound this probability indirectly.



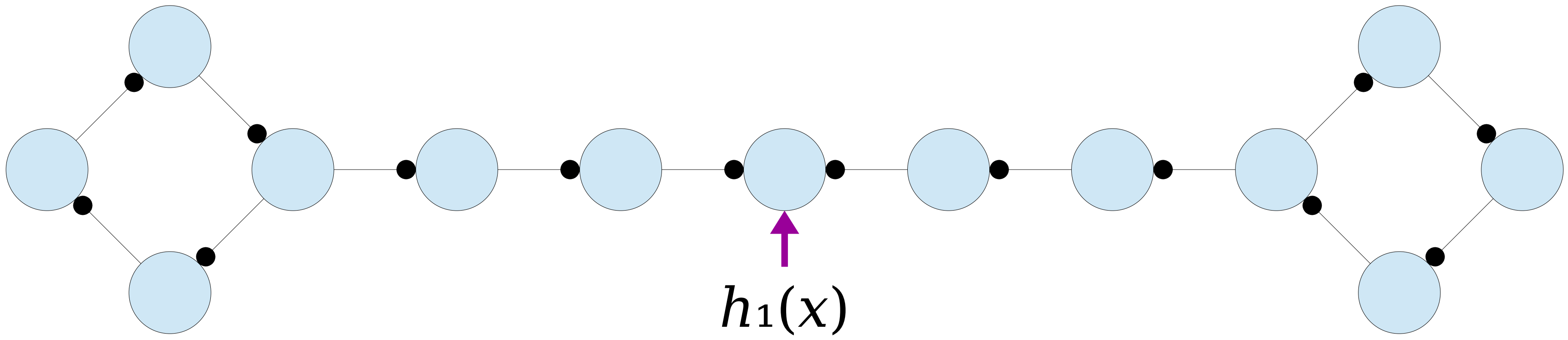
We're right back where we started. This pattern will continue indefinitely.

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



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

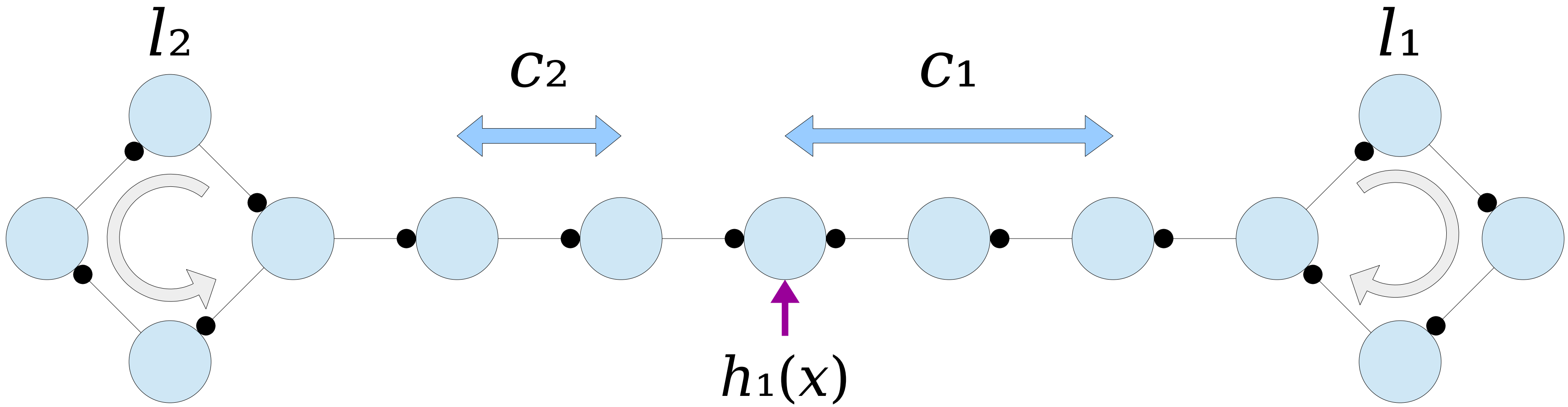
Insertion fails if we have a complex connected component. What specifically happens in that case?



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 $h_1(x)$.

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?

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

$$\boxed{m = (1 + \varepsilon)n}$$

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

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

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

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?

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

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 fails 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 fails is $O(1 / m)$.

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

$$\underbrace{\mathbf{O(1)}}_{\text{Expected cost of successful insertion.}} + \underbrace{\mathbf{O(1 / \mathit{m}^2)}}_{\text{Expected number of rehashes.}} \cdot \underbrace{\mathbf{O(\mathit{m})}}_{\text{Cost of doing one rehash.}}$$

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 amortization kicks in because we need to periodically double the sizes of the tables as the number of elements increases.
- 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 too strong of an assumption.
- What we know:
  - O(log n)-independence is sufficient for expected O(1) insertion time, but 6-independence isn't.
  - The simplest 2-independent family of hash functions (polynomial hashing) are *terrible* for cuckoo hashing.
  - Some simple classes of 3-independent hash functions (tabulation hashing) perform well both theoretically and practically.
- ***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 $k > 2$ tables leads to better memory efficiency than $k = 2$ 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

- ***Beyond Worst-Case Analysis***
  - Is O(log *n*) the be-all, end-all of BST analysis? (Hint: Betteridge's Law of Headlines)
- ***Weight-Balanced Trees***
  - A different way of balancing a tree.
- ***Finger Search Trees***
  - Picking up where we left off.
- ***Iacono's Working Set Structure***
  - Storing elements in doubly-exponentially-increasing forests.