Cuckoo Hashing

Outline for Today

- ***Towards Perfect Hashing***
  - Reducing worst-case bounds
- ***Cuckoo Hashing***
  - Hashing with worst-case O(1) lookups.
- ***The Cuckoo Graph***
  - A framework for analyzing cuckoo hashing.
- ***Analysis of Cuckoo Hashing***
  - 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

- The expected cost of a lookup in a chained hash table is $O(1 + \alpha)$ for any load factor α.
- For any fixed load factor α, the expected cost of a lookup in linear probing is $O(1)$, where the constant depends on α.
- 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 linear probing hash table is $\Omega(\log n)$.
- ***Theorem:*** Assuming truly random hash functions, the expected worst-case cost of a lookup in a chained hash table is $\Theta(\log n / \log \log n)$.
- ***Proofs:*** Exercise 11-1 and 11-2 from CLRS. ☺

Perfect Hashing

- A perfect hash table is one where lookups take worst-case time O(1).
- There's a pretty sizable gap between the expected worst-case bounds from chaining and linear probing – and that's on *expected* worst-case, not worst-case.
- We're going to need to use some more creative techniques to close this gap.

Multiple-Choice Hashing

Second-Choice Hashing

- Suppose that we distribute n balls into m urns using the following strategy:
  - For each ball, choose two urns totally at random.
  - Place the ball into the urn with fewer balls in it.
- ***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.
  - The math involved is tricky but interesting – check it out if you're curious!

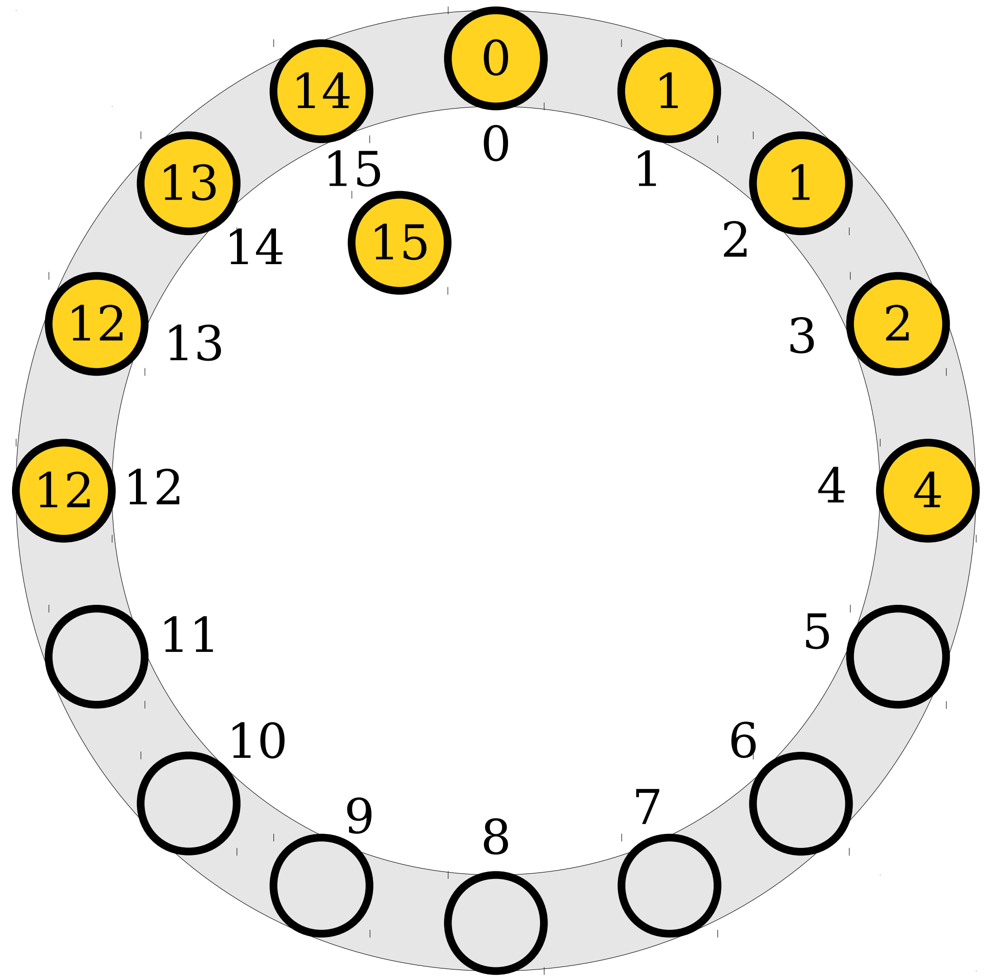
Second-Choice Hashing

- Imagine we 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.
- ***Theorem:*** The expected cost of a lookup in such a hash table is $O(1 + \alpha)$.
  - This is certainly no worse than chained hashing.
- ***Theorem:*** Assuming truly random hash functions, the expected worst-case cost of a lookup in such a hash table is $O(\log \log n)$.
- ***Open problem:*** What is the smallest k for which there are k-independent hash functions that match the bounds using truly random hash functions?

Hashing with Relocation

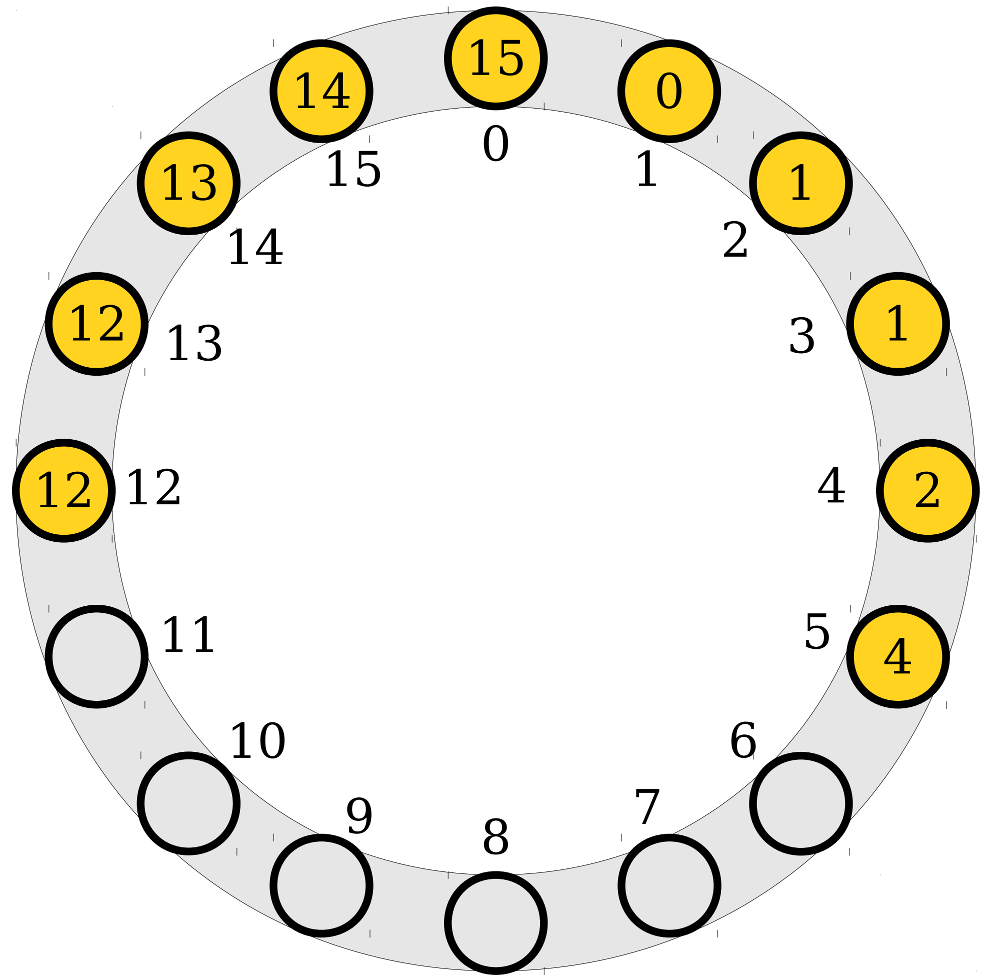
Robin Hood Hashing

- ***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.



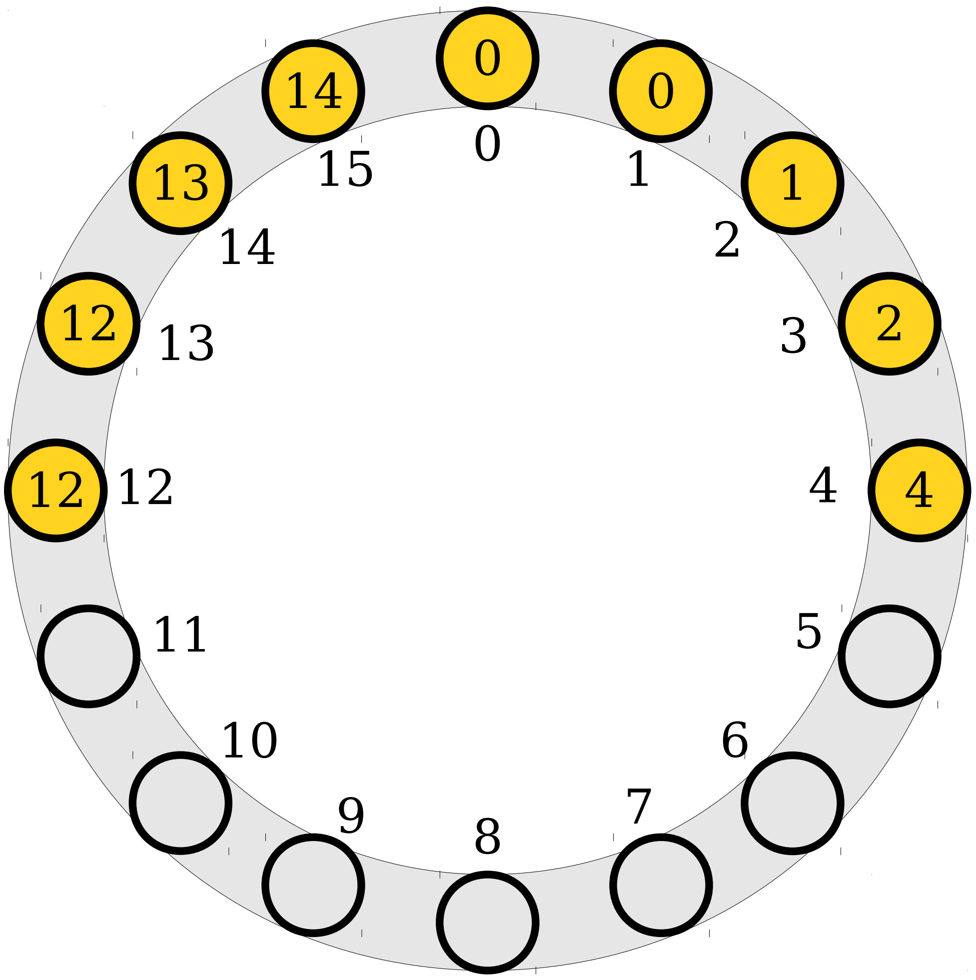
Robin Hood Hashing

- ***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.



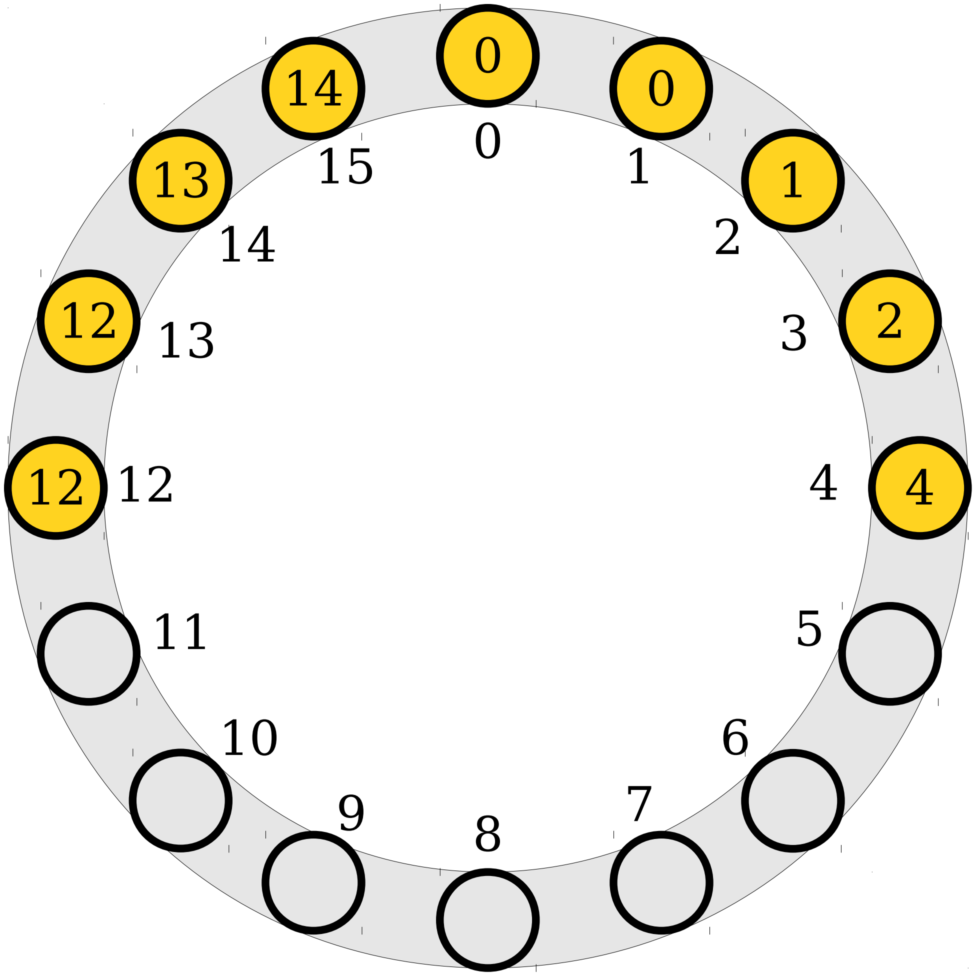
Robin Hood Hashing

- ***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. We bounded the cost of a lookup in linear probing by looking at the cost of the run the element was in, so the analysis still upper-bounds the cost.



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 that might be fun to combine:
  - ***Second-choice hashing:*** Give each element in a hash table two choices of where to go, and put it at the least-loaded location.
  - ***Relocation hashing:*** Allow objects in a hash table to move after being placed.
- 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?

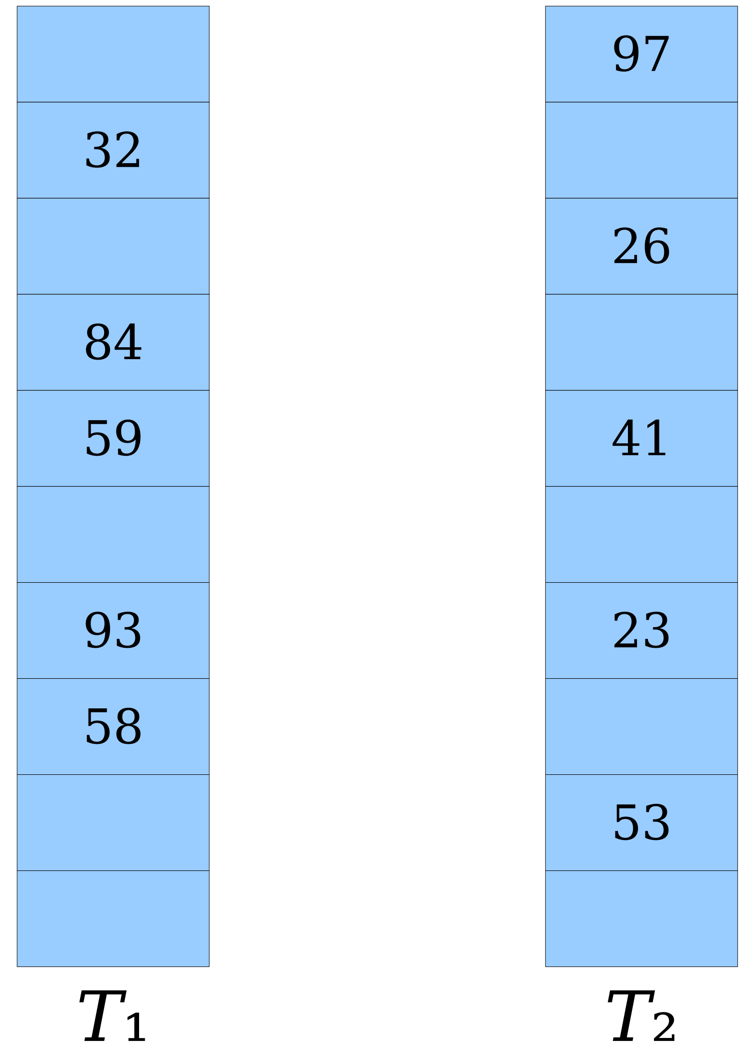
Cuckoo Hashing

Cuckoo Hashing

- ***Cuckoo hashing*** is a simple hash table where
  - Lookups are worst-case O(1).
  - Deletions are worst-case O(1).
  - Insertions are amortized, expected O(1).
  - Insertions are amortized O(1) with reasonably high probability.
- Today, we'll explore cuckoo hashing and work through the analysis.

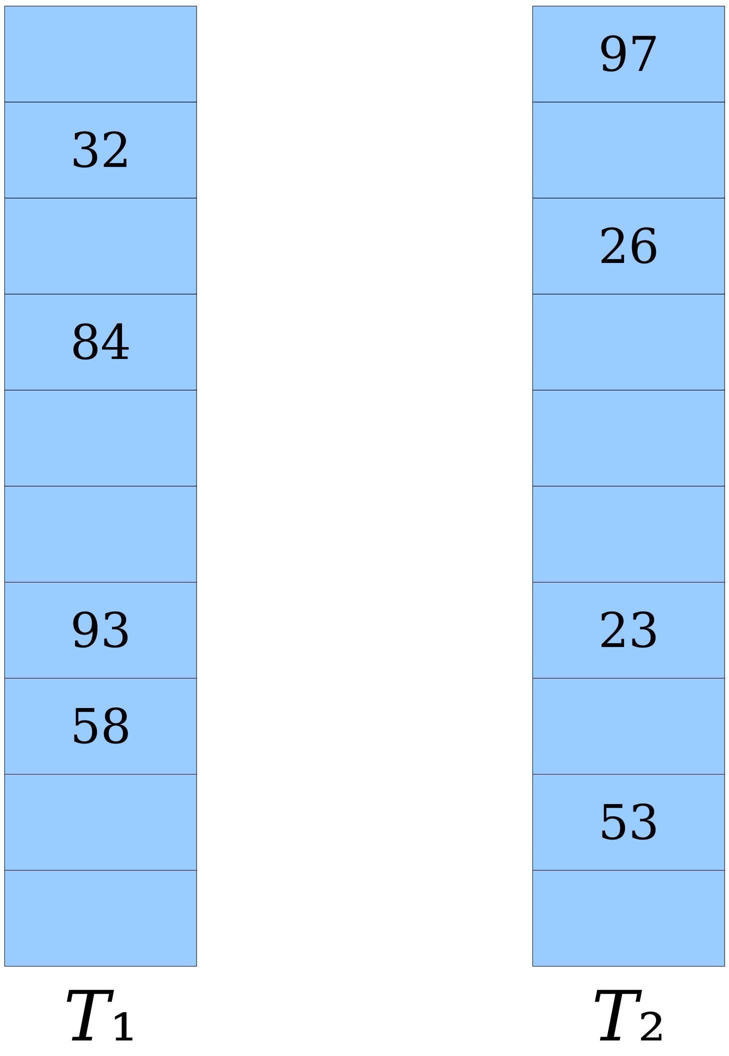
Cuckoo Hashing

- 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.



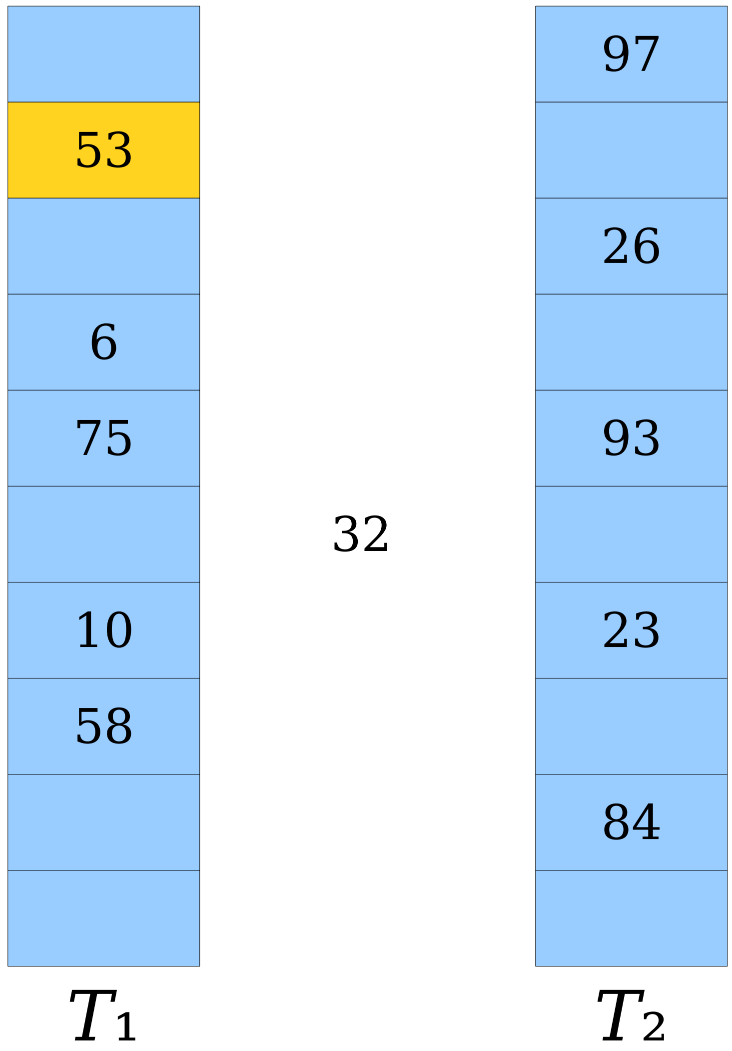
Cuckoo Hashing

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



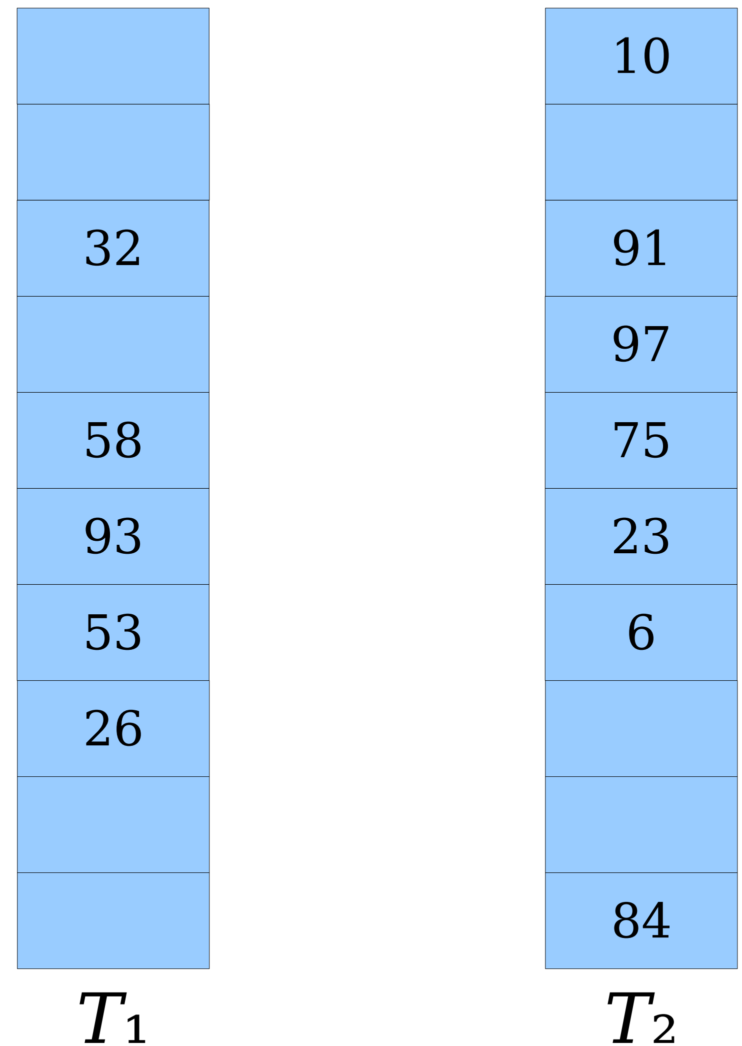
Cuckoo Hashing

- 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.



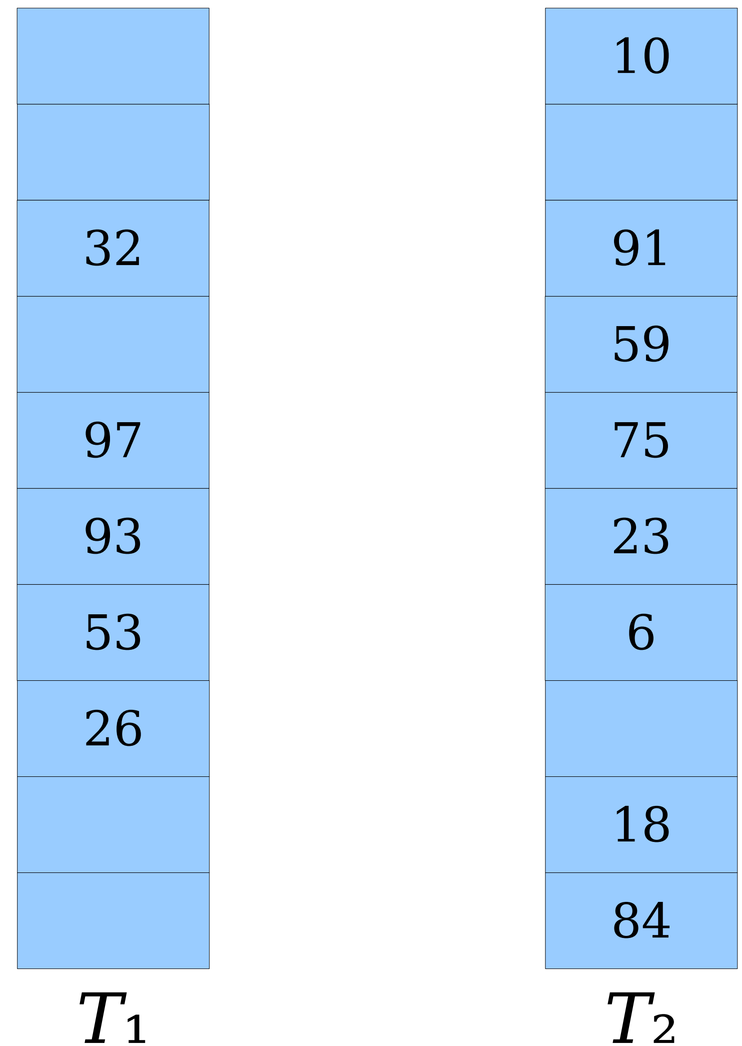
Cuckoo Hashing

- Insertions run into trouble if we run into a 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.



A Note on Cycles

- It's possible for a successful insertion to revisit the same slot twice.
- Cycles only arise if we revisit the same slot with the same element to insert.

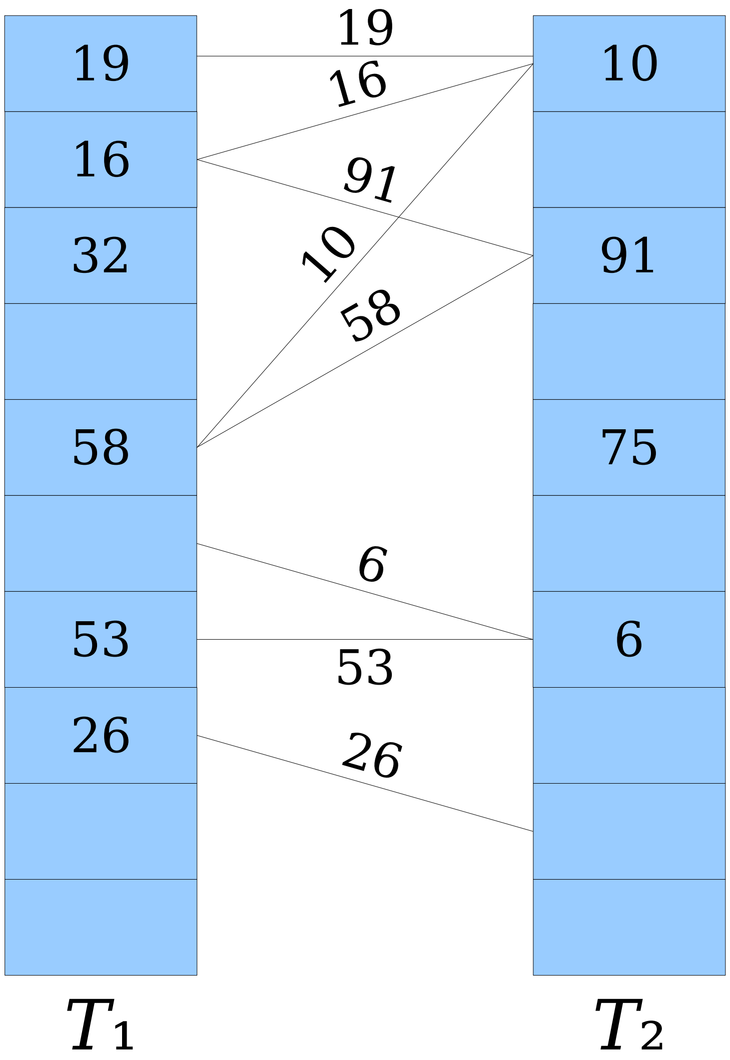


Analyzing Cuckoo Hashing

- Cuckoo hashing can be tricky to analyze for a few reasons:
  - Elements move around and can be in one of two different places.
  - The sequence of displacements can jump chaotically over the table.
- It turns out there's a beautiful framework for analyzing cuckoo hashing.

The Cuckoo Graph

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

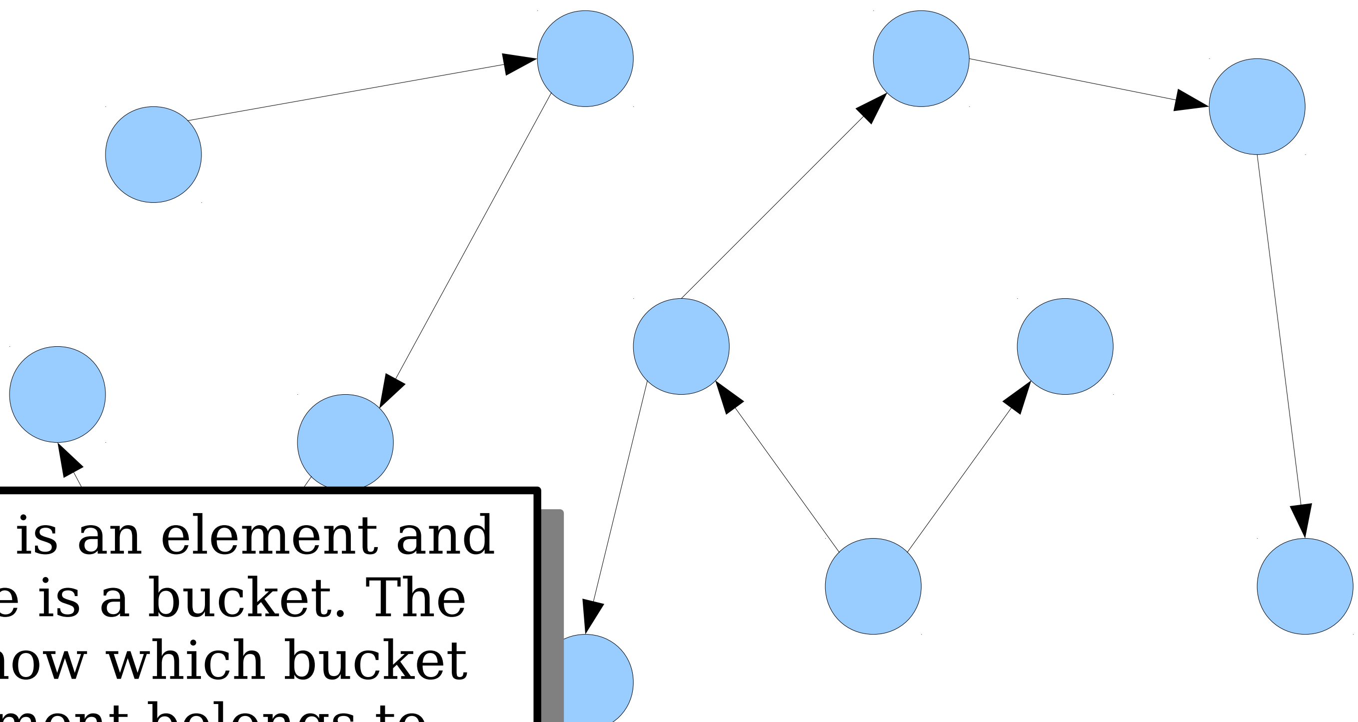


The Cuckoo Graph

- ***Claim 1:*** If x is inserted into a cuckoo hash table, the insertion fails if the connected component containing x has two or more cycles.
- ***Proof:*** Each edge represents an element and needs to be placed in a bucket.
- If the number of nodes (buckets) in the CC is k, then there must be at least $k + 1$ elements (edges) in that CC to have two cycles.
- Therefore, there are too many nodes to place into the buckets.

The Cuckoo Graph

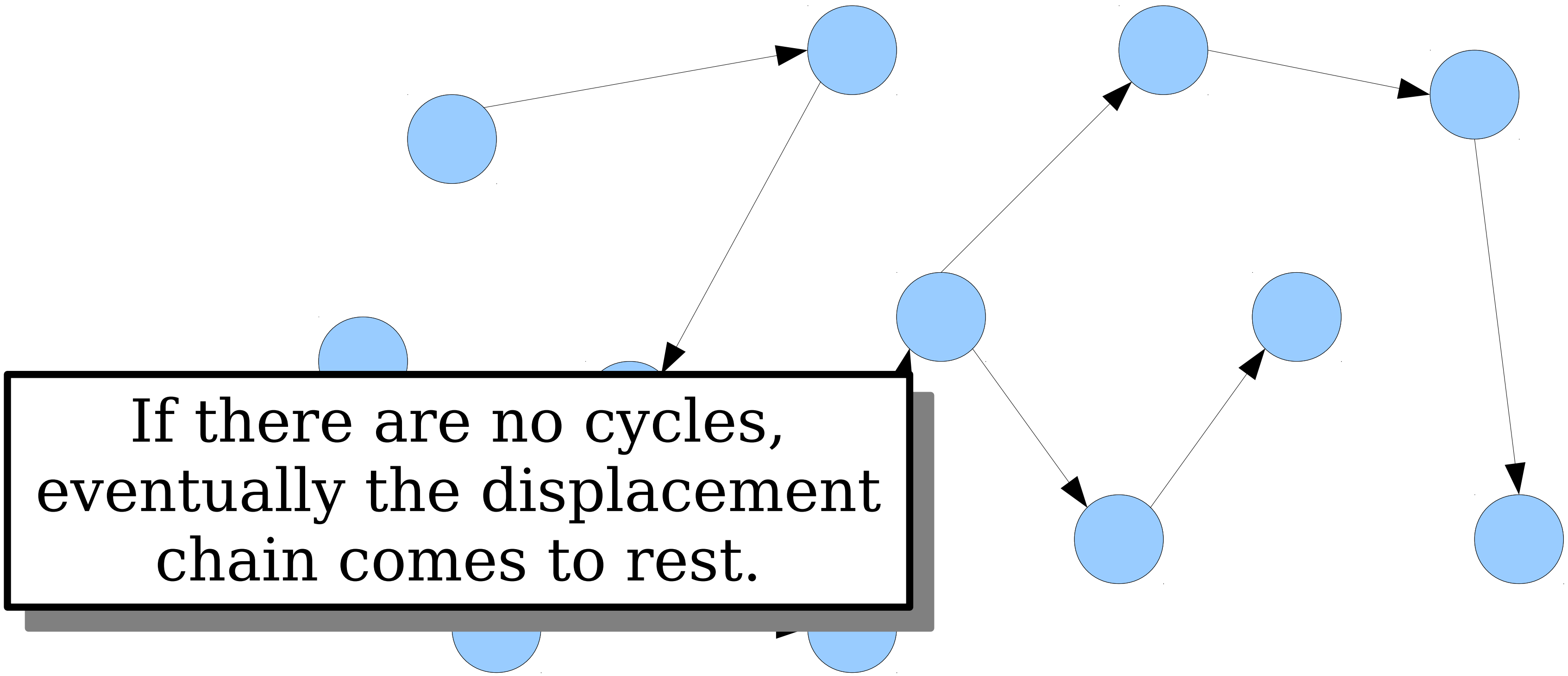
- ***Claim 2:*** If x is inserted into a cuckoo hash table, the insertion succeeds if the connected component containing x contains either no cycles or only one cycle.



Each edge is an element and each node is a bucket. The arrows show which bucket each element belongs to.

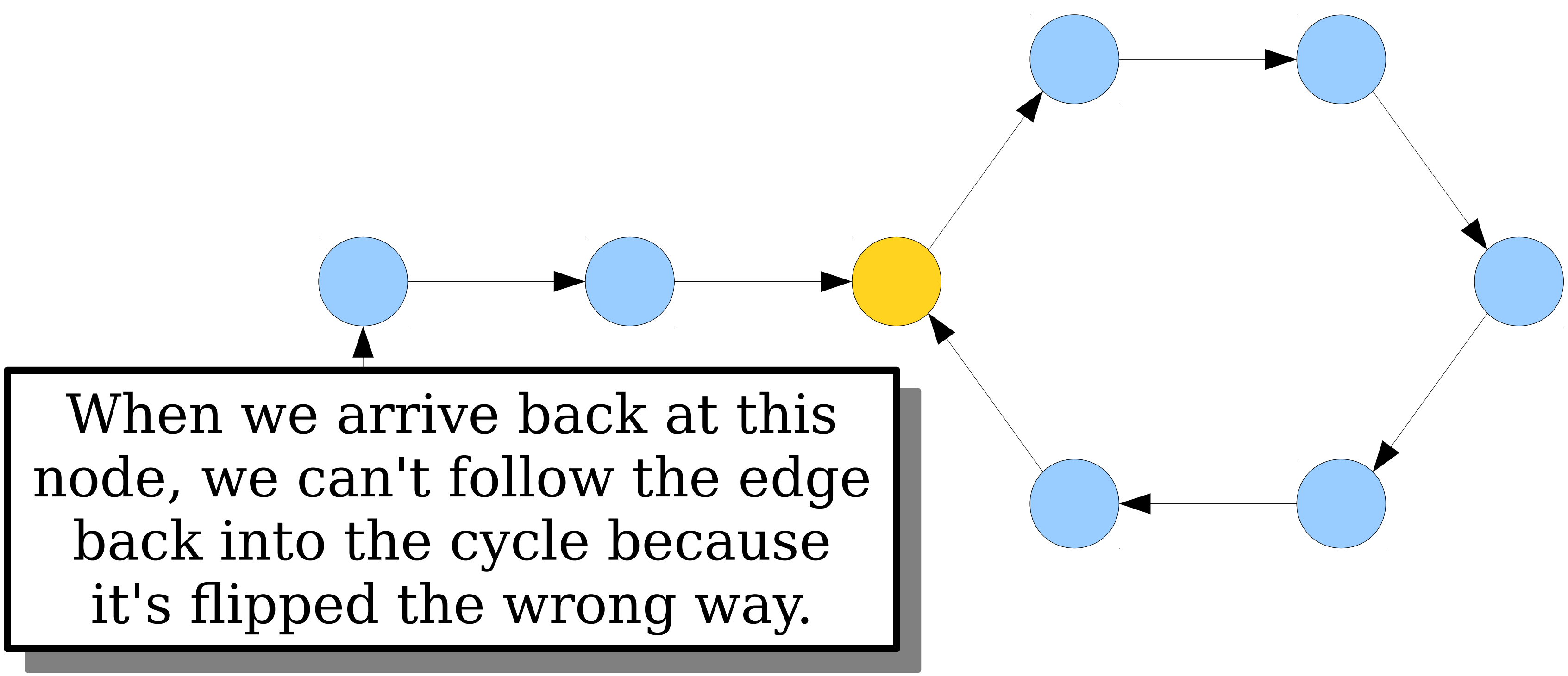
The Cuckoo Graph

- ***Claim 2:*** 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.



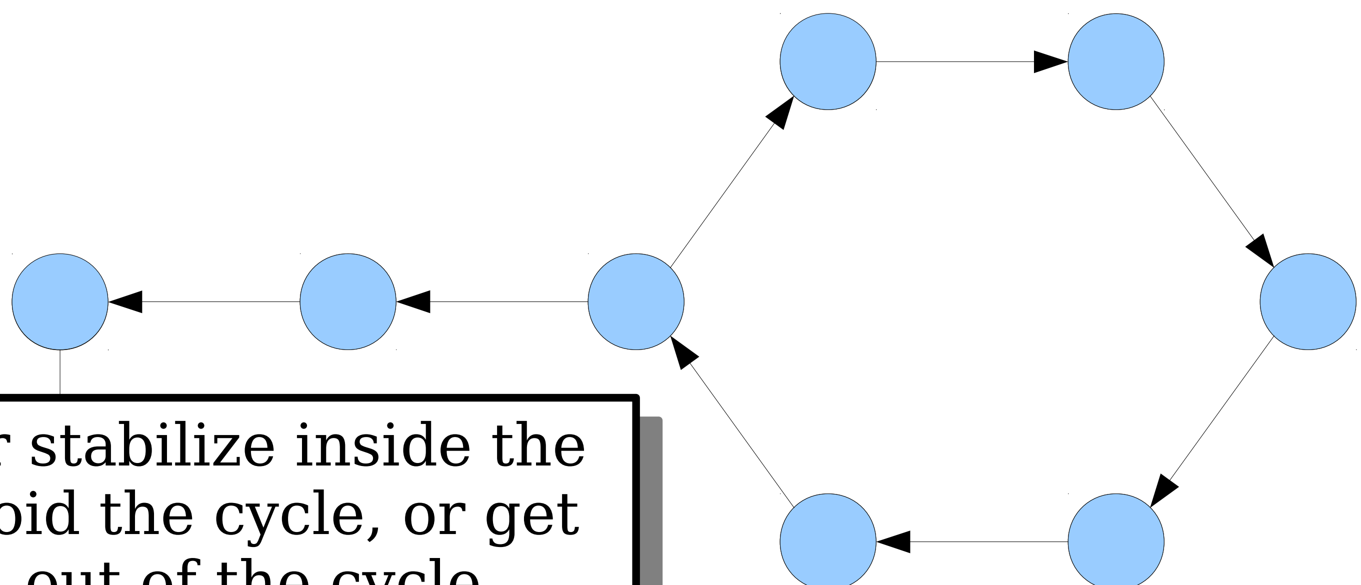
The Cuckoo Graph

- ***Claim 2:*** If *x* is inserted into a cuckoo hash table, the insertion 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 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 succeeds if the connected component containing x contains either no cycles or only one cycle.
- ***Claim 3:*** If x is inserted in a connected component with k nodes, the insertion process does at most $2k$ displacements.

Terminology

- A ***tree*** is an undirected, connected component with no cycles.
- A ***unicyclic component*** is a connected component with exactly one cycle.
- A connected component is called ***complex*** if it's neither a tree nor unicyclic.
- Cuckoo hashing fails iff any of the connected components in the cuckoo graph are complex.

The Gameplan

- To analyze cuckoo hashing, we'll do the following.
  - First, we'll analyze the probability that a connected component is complex.
  - Next, under the assumption that no connected component is complex, we'll analyze the expected cost of an insertion.
  - Finally, we'll put the two together to complete the analysis.

Time-Out for Announcements!

Problem Set Five

- Problem Set Five goes out today. It's due one week from today at the start of lecture.
  - Play around with randomized data structures, both in theory and in practice!
- Problem Set Four was due at 3:00PM today.

Final Project Topics

- Final project topics have been assigned. The list of teams and topics is available on the course website under the “Announcements” section.
- Feel free to get started!

Back to CS166!

Step One:

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.

Tricky Combinatorics

- ***Question:*** What is the probability that a randomly-chosen bipartite graph with $2m$ nodes and n edges will contain a complex connected component:
- ***Answer:*** If $n = (1 - \delta)m$, the answer is

$$1-\frac{(2\delta^2-5\delta+5)(1-\delta)^3}{12(2-\delta)^2\delta^3}\frac{1}{m}+O\left(\frac{1}{m^2}\right)$$

- Source: "Bipartite Random Graphs and Cuckoo Hashing" by Reinhard Kutzelnigg.

The Main Result

- ***Theorem:*** If $m = (1 + \varepsilon)n$ for some $\varepsilon > 0$, the probability that the cuckoo graph contains a complex connected component is $O(1 / m)$.
- I have scoured the literature and cannot seem to find a simple proof of this result.
- ***Challenge Problem:*** Provide a simple proof of this result.

The Implications

- If $m = (1 + \varepsilon)n$, then the hash table will have a load factor of

$$n \,/\, 2m = 1 \,/\, (2 + 2\varepsilon).$$

- This means that roughly half of the table cells will be empty.
- ***Fact:*** There is an abrupt phase transition in the success probability when the tables get close to half full. The likelihood of getting a complex connected component is extremely high at that point.
- There are techniques for improving the space usage of cuckoo hashing; more on that later on.

Step Two:

Analyzing Connected Components

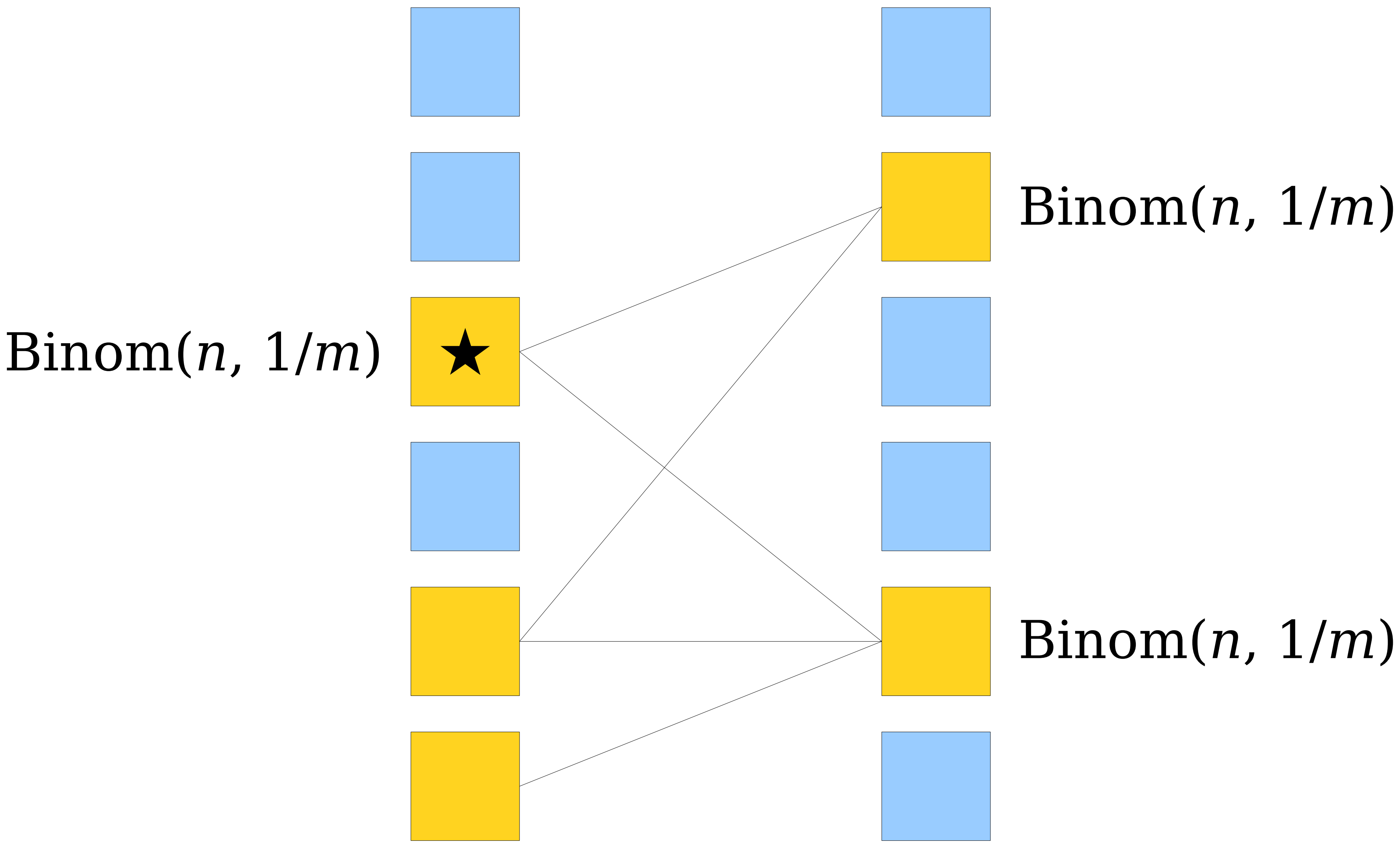
Analyzing Connected Components

- 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?

The Result

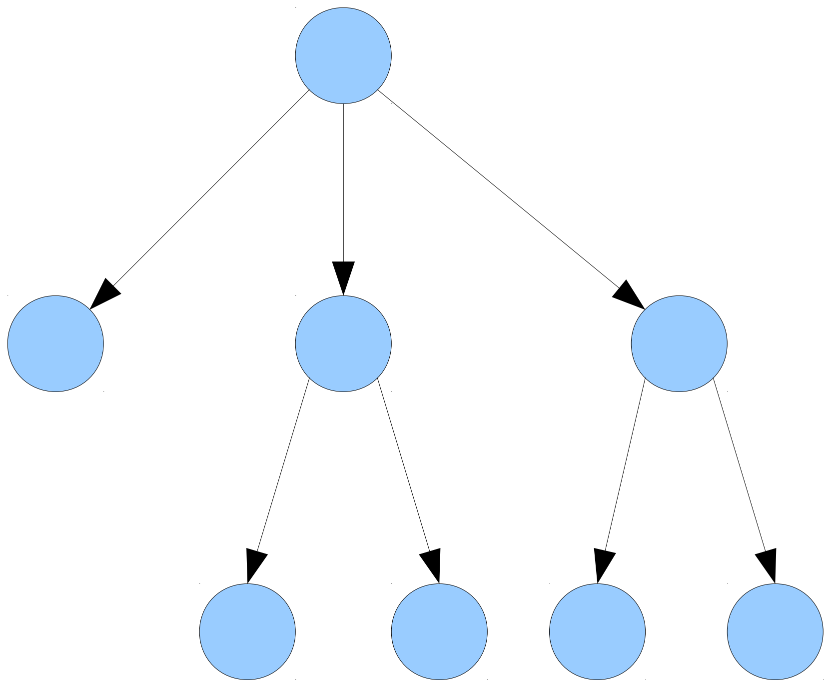
- ***Claim:*** If $m \geq (1 + \varepsilon)n$ for any $\varepsilon > 0$, then on expectation, the cost of an insertion in a cuckoo hash table that does not trigger a rehash is $O(1 + \varepsilon^{-1})$.
- ***Proof idea:*** Show that the expected number of nodes in a connected component is at most $1 + \varepsilon^{-1}$.
- Let's see how to do this!

Sizing a Connected Component



Modeling the DFS

- Fix a start node *v*.
- The number of nodes incident to *v* is modeled by a Binom(*n*, 1 / *m*) variable.
- For each node *u* connected to *v*, we can upper-bound the number of nodes connected to *u* by a Binom(*n*, 1 / *m*) variable.



Subcritical Galton-Watson Processes

- The process modeled by this tree is called a ***subcritical Galton-Watson process***.
- Models a tree where each node has a number of children given by i.i.d. copies of some variable ξ.
- ***Constraint:*** E[ξ] must be less than 1.

Subcritical Galton-Watson Processes

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

Subcritical Galton-Watson Processes

Lemma: $\mathrm{E}[X_n] = \mathrm{E}[\xi]^n$.

Proof: Conditional expectation. $\blacksquare$

Theorem: The expected size of a connected component in the cuckoo graph is at most $1 + \varepsilon^{-1}$.

Proof: We can upper-bound the number of nodes in a CC with the number of nodes in a subcritical Galton-Watson process where $\xi \sim \text{Binom}(n, 1 / m)$. If we denote by X the total number of nodes in the CC, we see that

$$X = \sum_{i=0}^{\infty} X_i$$

Therefore, the expected value of X is given by

$$\mathrm{E}[X] = \mathrm{E}\left[\sum_{i=0}^{\infty} X_i\right] = \sum_{i=0}^{\infty} \mathrm{E}[X_i] = \sum_{i=0}^{\infty} \mathrm{E}[\xi]^i = \frac{1}{1 - \mathrm{E}[\xi]}$$

Note that $\mathrm{E}[\xi] = n / m \le (1 + \varepsilon)^{-1}$, so

$$\mathrm{E}[X] = \left(1 - \frac{n}{m}\right)^{-1} \le \left(1 - \frac{1}{1+\varepsilon}\right)^{-1} = \frac{1+\varepsilon}{\varepsilon} = 1 + \varepsilon^{-1}$$

Therefore, the expected size of a CC in the cuckoo graph is at most $1 + \varepsilon^{-1}$. ■

Finishing Touches

Lemma: The expected cost of a single rehash, assuming that it succeeds, is $O(m + n\varepsilon^{-1})$.

Proof: If the rehash succeeds, each insertion takes expected time $O(1 + \varepsilon^{-1})$. There are n insertions, so the time will be $O(n + \varepsilon^{-1}n)$. We also do $O(m)$ work reinitializing the buckets, so the total time is $O(m + n\varepsilon^{-1})$. ■

Lemma: The expected cost of a rehash is $O(m + n\varepsilon^{-1})$.

Proof: Each rehash succeeds with probability $1 - O(1 / m)$. Therefore, on expectation, only $1 / (1 - O(1/m)) = O(1)$ rehashes are necessary. Since each one takes expected time $O(m + n\varepsilon^{-1})$, the expected total time is $O(m + n\varepsilon^{-1})$. ■

Planned Rehashing

- We need to rehash in two situations:
  - A ***planned*** rehash, a rehash to ensure that $m \geq (1 + \varepsilon)n$. This needs to happen periodically to ensure the table grows.
  - An ***unplanned*** rehash, where we rehash because there is a complex CC in the table.
- If we repeatedly double the size of the table, the expected total work done in planned rehashing is $O(m + n\varepsilon^{-1})$ across the lifetime of the table.
  - Analysis similar to chained hashing.
- This amortizes out to expected $O(1 + \varepsilon + \varepsilon^{-1})$ additional work per insertion.

Theorem: The expected, amortized cost of an insertion into a cuckoo hash table is $O(1 + \varepsilon + \varepsilon^{-1})$.

Proof: We have already shown that the amortized overhead of an insertion due to planned rehashing is $O(1 + \varepsilon + \varepsilon^{-1})$, so all we need to do is analyze the expected cost ignoring planned rehashes.

With probability $1 - O(1 / m)$, the expected cost is $O(1 + \varepsilon + \varepsilon^{-1})$. With probability $O(1 / m)$, we have to rehash, which takes expected time $O(m + n\varepsilon^{-1})$. Therefore, the expected cost of an insert is

$$\begin{aligned} & O(1 + \varepsilon + \varepsilon^{-1}) + O(1 + n\varepsilon^{-1} / m) \\ = \ & O(1 + \varepsilon + \varepsilon^{-1}) + O(1 + \varepsilon^{-1}) \\ = \ & O(1 + \varepsilon + \varepsilon^{-1}) \end{aligned}$$

As required. ■

Some Technical Details

A Few Technical Details

- There are a few technical details we glossed over in this analysis.
- ***Stopping time:*** Typically, cuckoo hashing triggers a rehash as soon as C log *n* elements have been displaced, for some constant C.
  - If a hash has been going on for that long, then it's almost certainly going to fail.
- Need to repeat the analysis to show that this addition doesn't cause rehashing with high frequency.

A Few Technical Details

- There are a few technical details we glossed over in this analysis.
- ***Hash function choice:*** The hash functions chosen need to have a high degree of independence for these results to hold.
  - It's known that 6-independent hashing isn't sufficient, and that O(log n)-independence is.
- In practice, most simple hash functions will work, though some particular classes do not. See “On the Risks of Using Cuckoo Hashing with Simple Universal Hash Classes” by Dietzfelbinger et al. for more details.

A Few Technical Details

- There are a few technical details we glossed over in this analysis.
- ***Load Factor:*** Cuckoo hashing's performance is highly influenced by the load factor.
- With two tables each of size $(1 + \varepsilon)n$, cuckoo hashing performs quite well and rarely needs to rehash. This leads to an effective load factor of less than ½.
- At higher load factors, cuckoo hashing tends to perform dramatically worse and frequently needs to rehash. This is due to a phase transition in the structure of the cuckoo graph.

Variations on Cuckoo Hashing

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 three hash locations are filled in.

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.

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.

More to Explore

- 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!

An Interesting Fact

- ***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

- ***Integer Data Structures***
  - Data structures for storing and manipulating integers.
- ***van Emde Boas Trees***
  - Searching in $o(\log n)$ time for integers.