Cuckoo Hashing

Problem Set 6 and Final Project Proposals are due in the box up front.

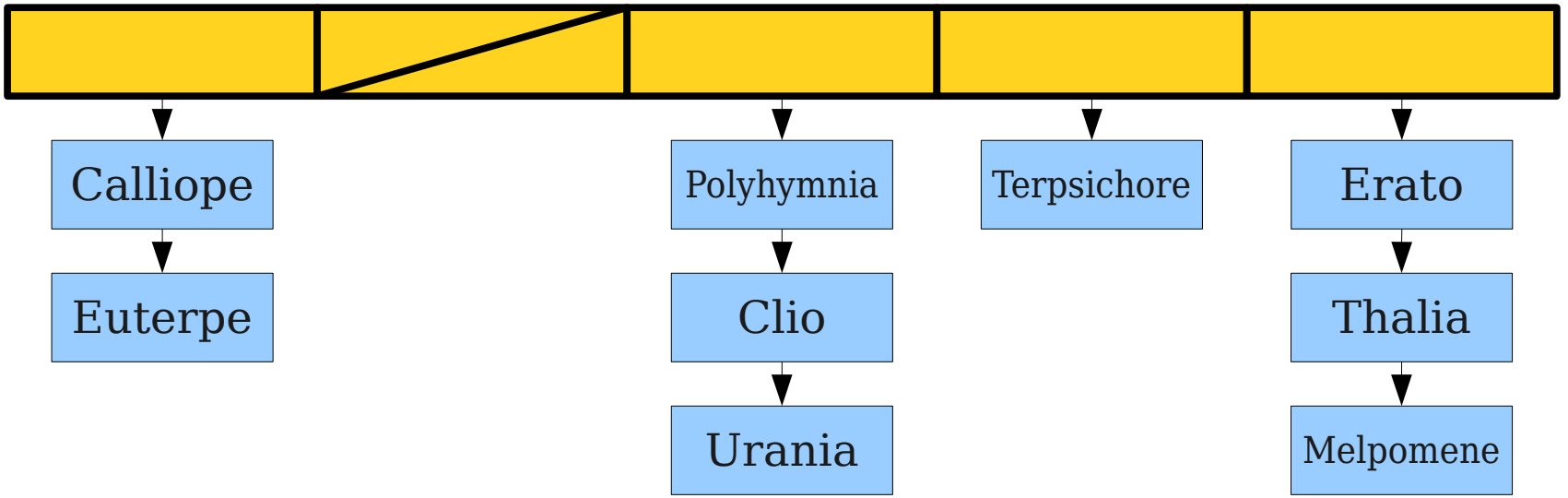
Outline for Today

- **Review of Chained Hash Tables**
  - A quick refresher on simple and efficient hash tables.
- **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?

Chained Hash Tables

Chained Hash Tables

- A **chained hash table** is a hash table in which collisions are resolved by placing all colliding elements into the same bucket.
- To determine whether an element is present, hash to its bucket and scan for it.
- Insertions and deletions are generalizations of lookups.



Chained Hash Tables

- The ***load factor*** of a chained hash table is the ratio of the number of elements (n) to the number of buckets (m).
  - Typically denoted $\alpha = n / m$.
- By doubling the table size whenever α exceeds two, α can be kept low with amortized O(1) work per insertion.
- With universal hash functions, the expected, amortized cost of a lookup is O(1).

Worst-Case Analyses

- In the worst case, a lookup in a chained hash table takes time $\Omega(n)$.
  - Happens when all elements are dropped into the same bucket.
  - Extremely unlikely in practice.
- Interestingly, the *expected worst-case* cost of a lookup is $O(\log n / \log \log n)$.
  - Check Exercise 11-2 in CLRS for details.

Worst-Case Efficient Hashing

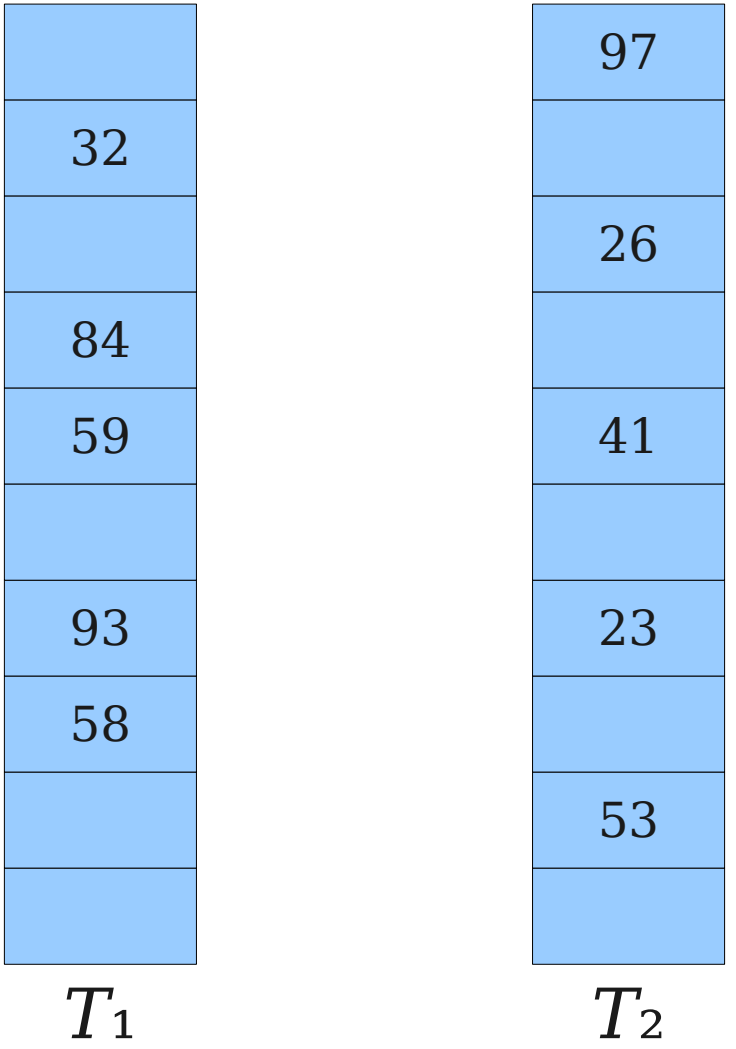
- **Question:** Is it possible to design a hash table where lookups are *worst-case* O(1)?
- This is challenging – we have to know exactly where to look to find the element we need, but there may be collisions!
- Many techniques for this have been developed over the years.
  - Check CLRS 11.5 for one such approach.

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.

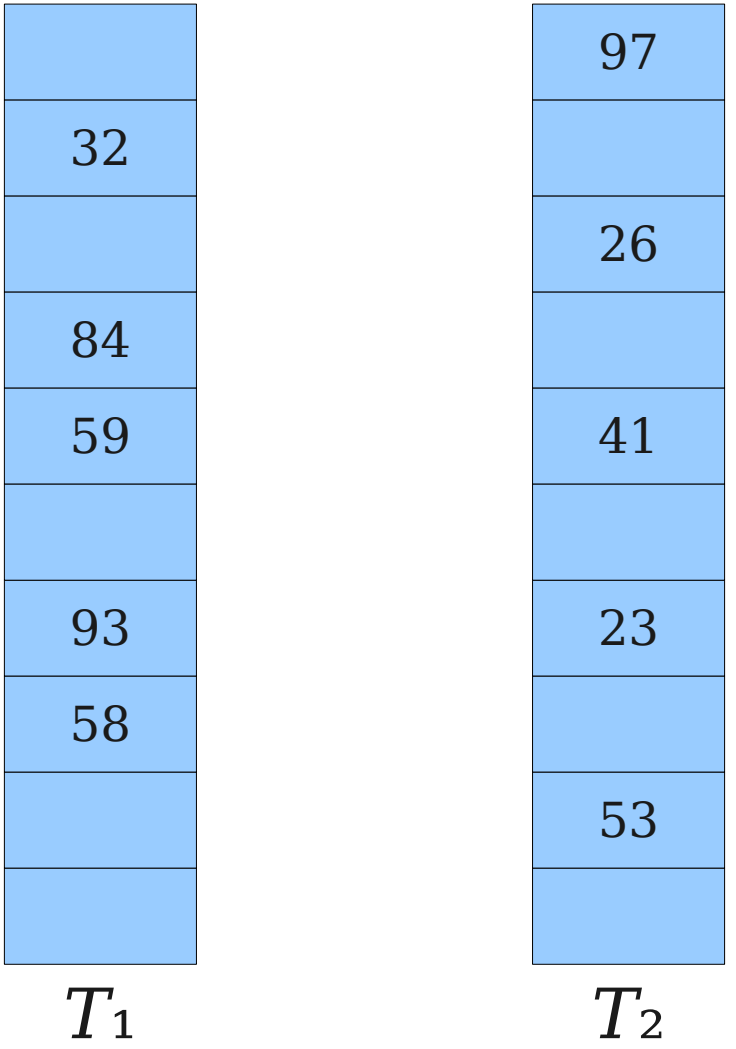
Cuckoo Hashing

- 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.



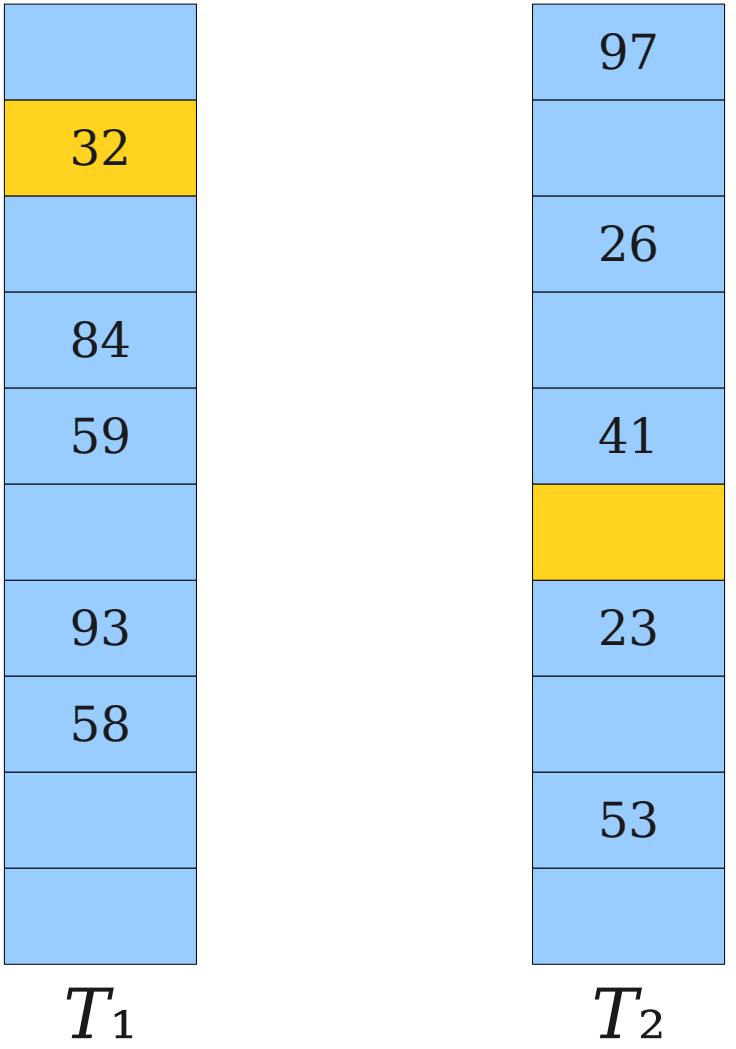
Cuckoo Hashing

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



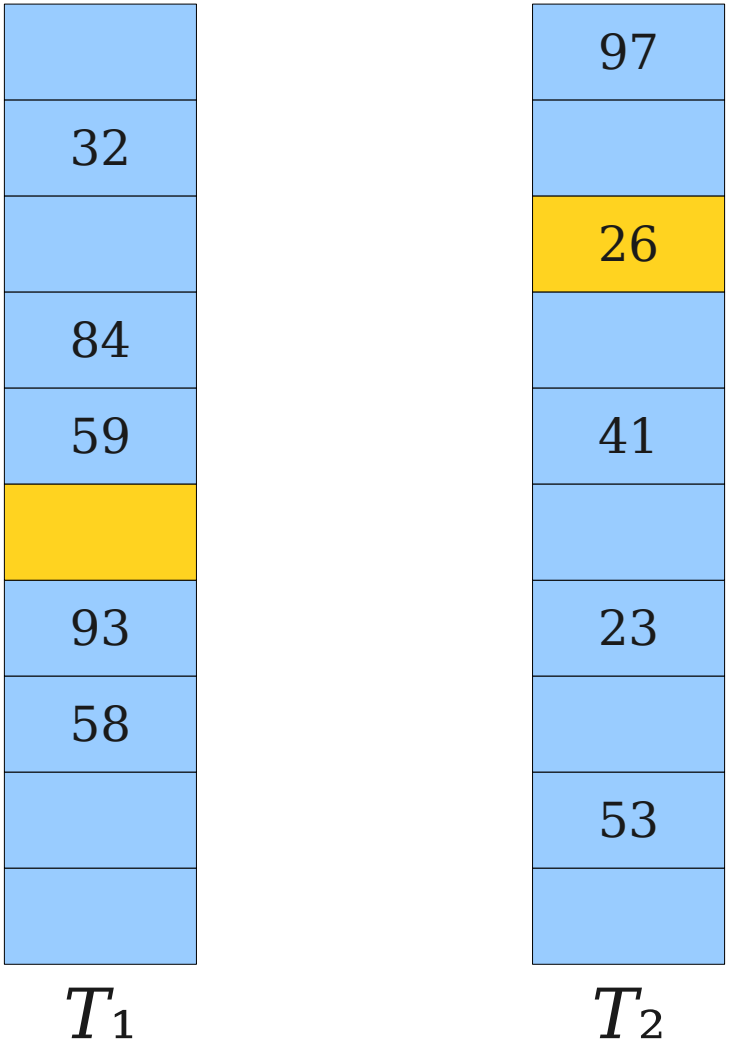
Cuckoo Hashing

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



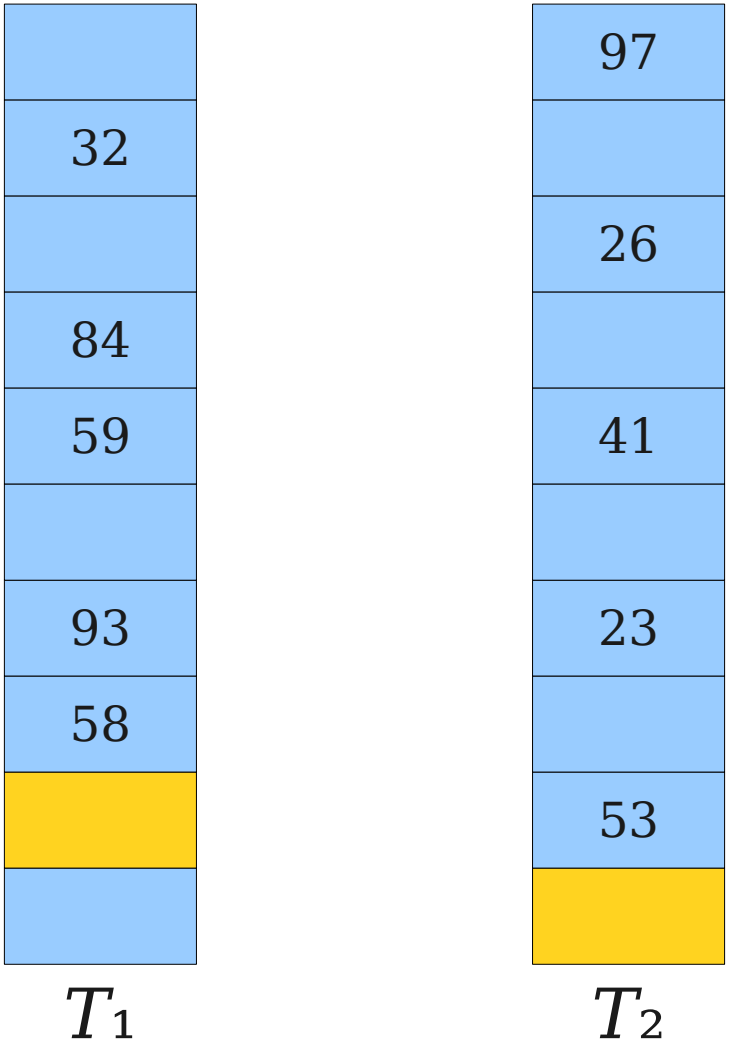
Cuckoo Hashing

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



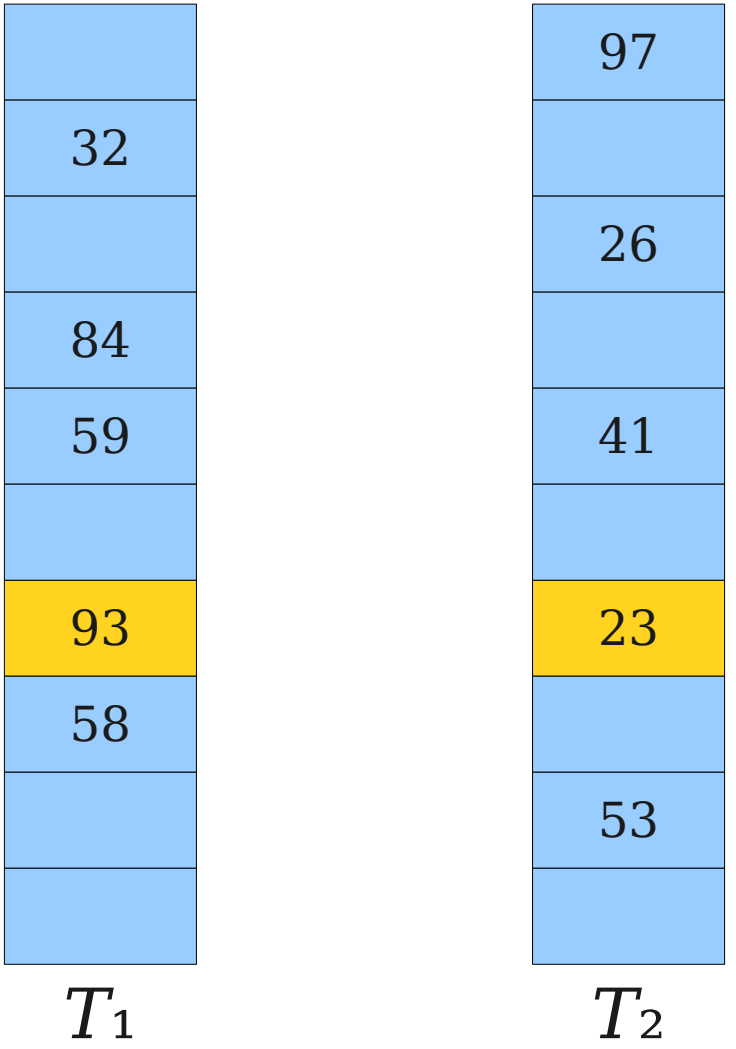
Cuckoo Hashing

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



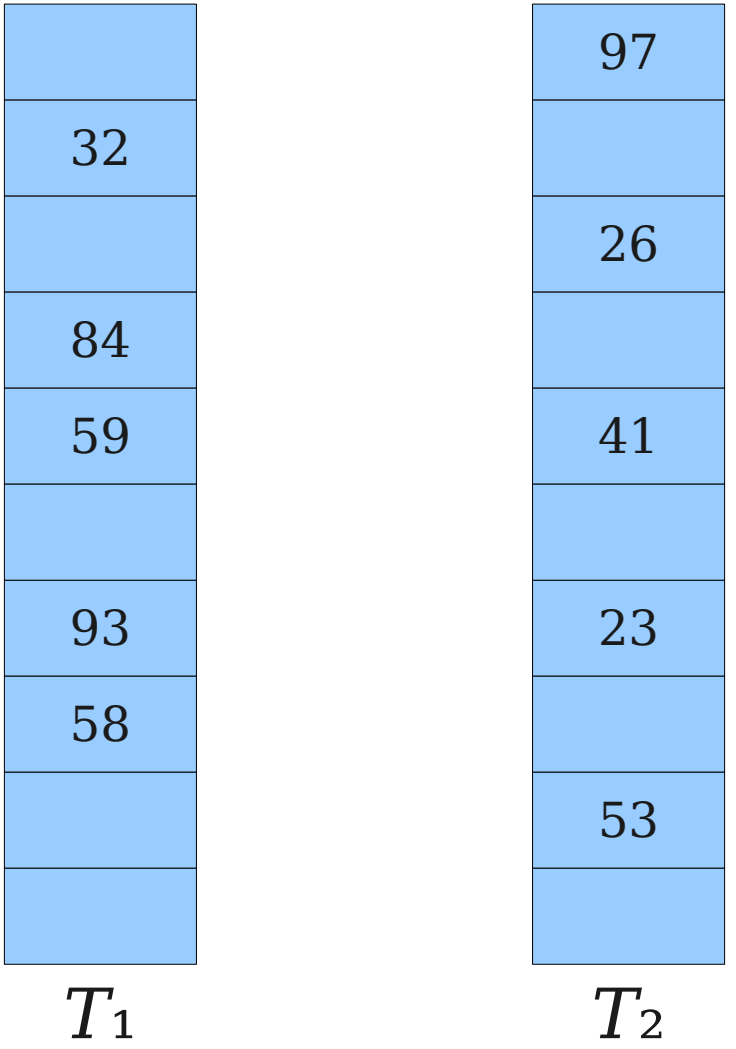
Cuckoo Hashing

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



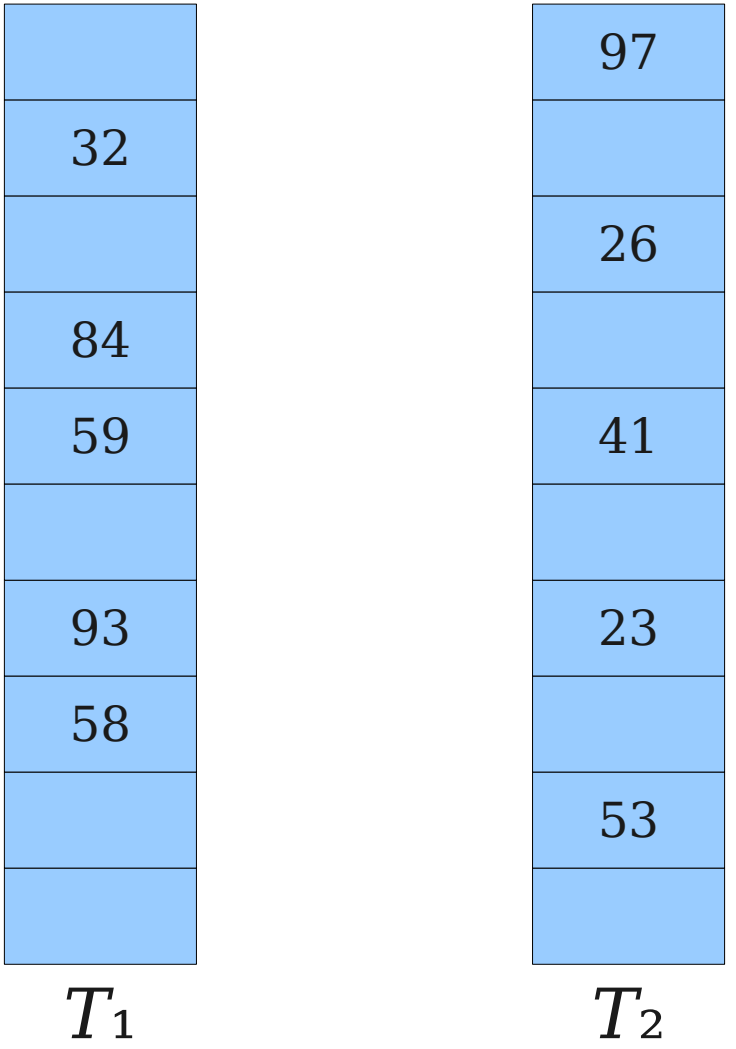
Cuckoo Hashing

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



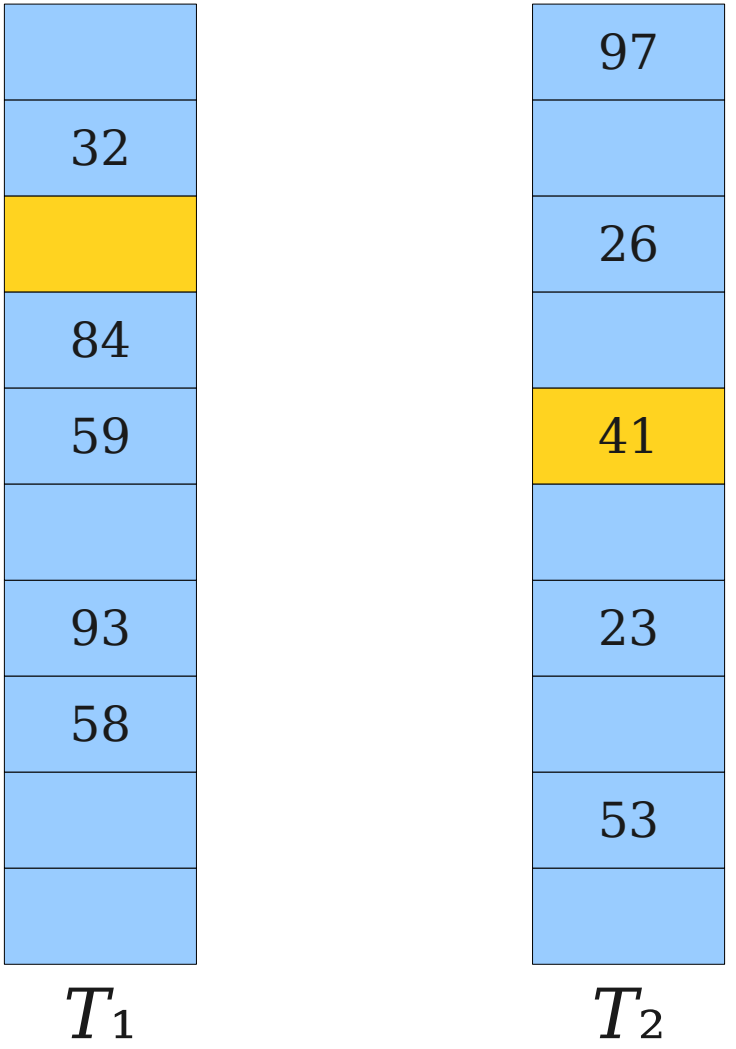
Cuckoo Hashing

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



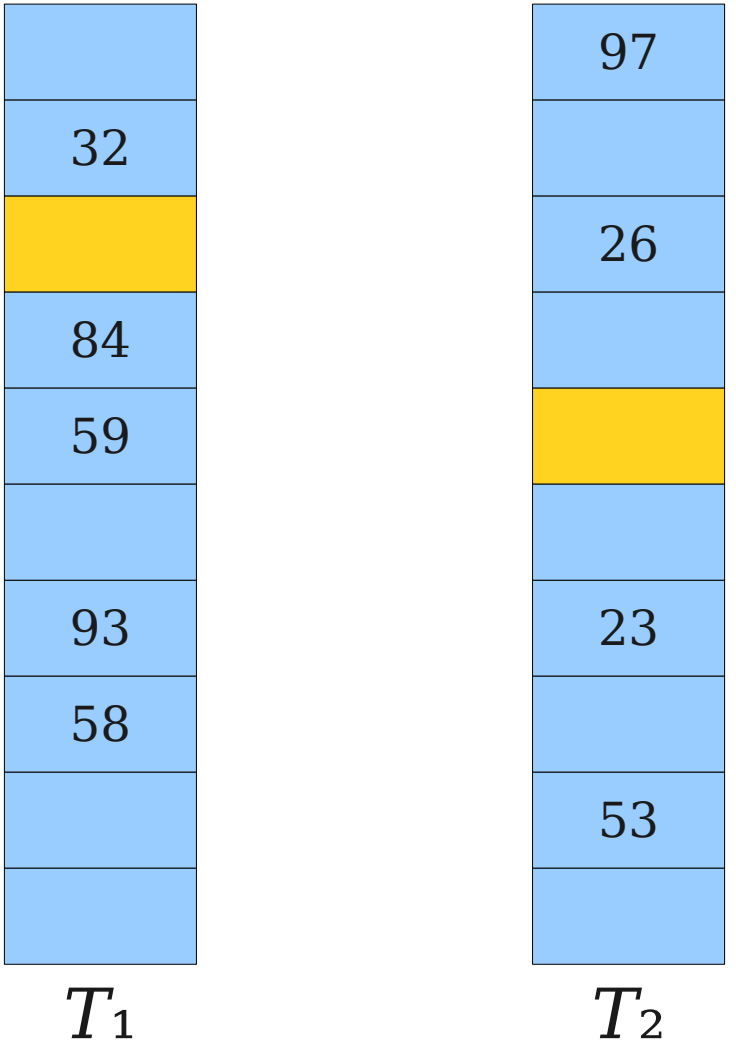
Cuckoo Hashing

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



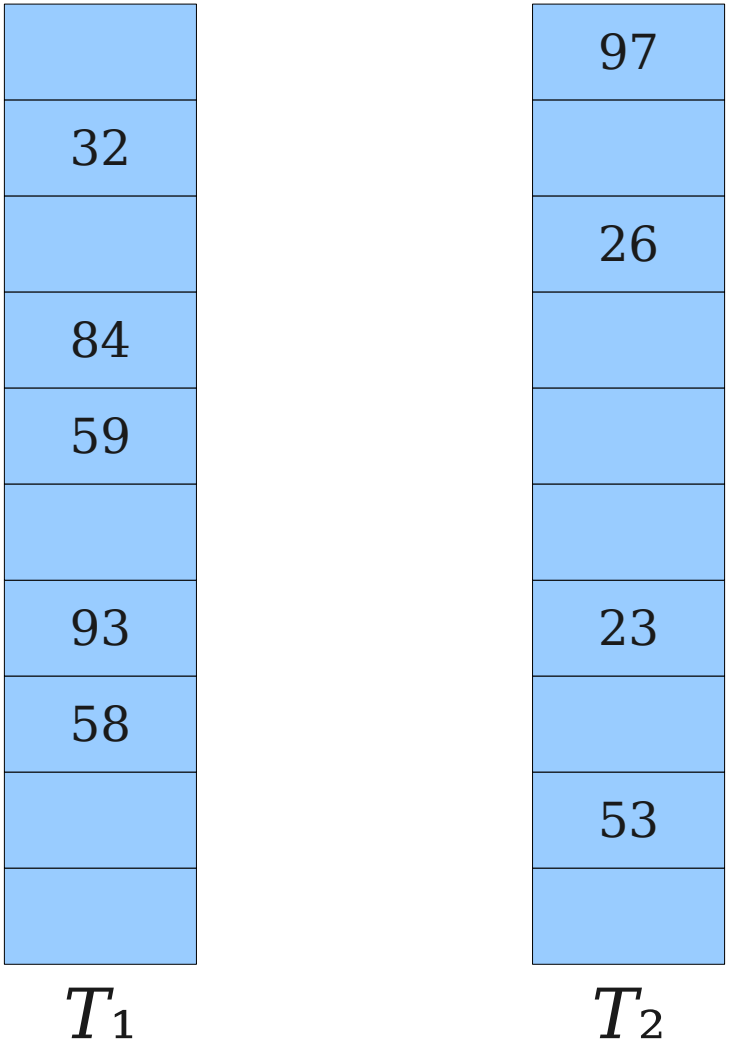
Cuckoo Hashing

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



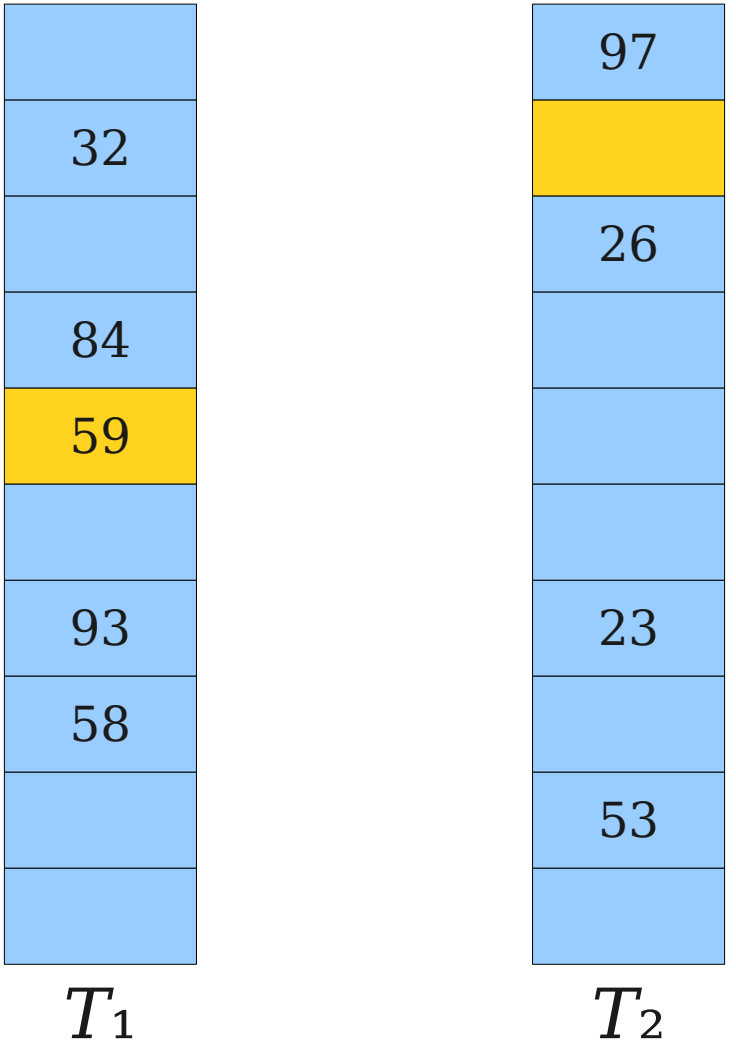
Cuckoo Hashing

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



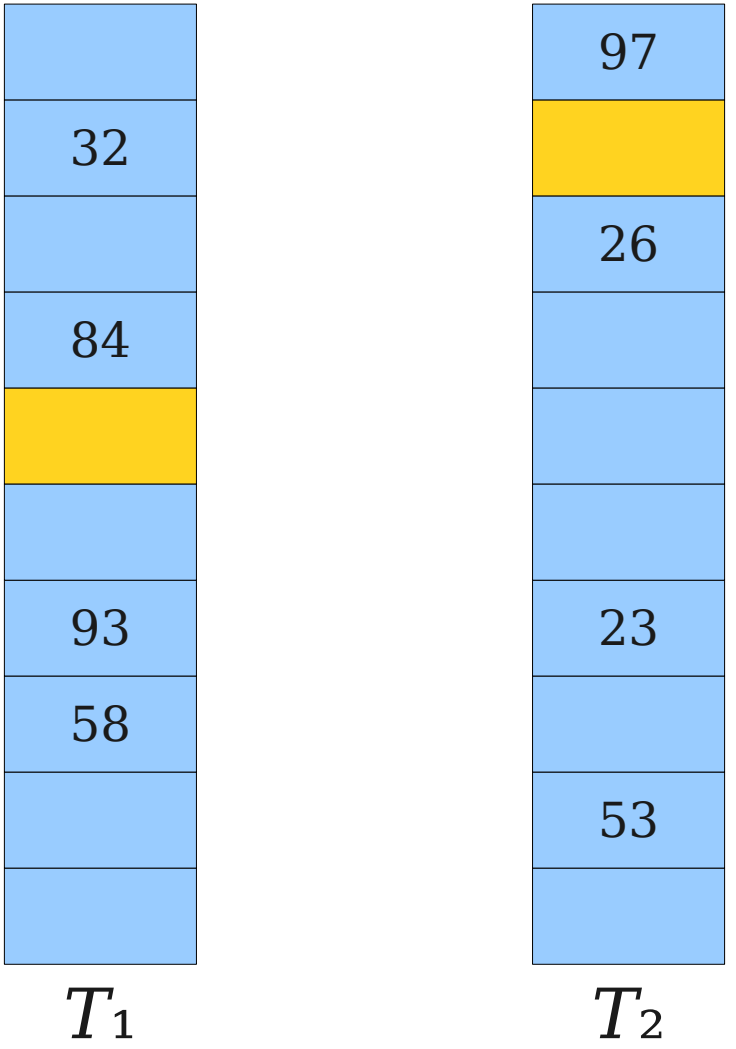
Cuckoo Hashing

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



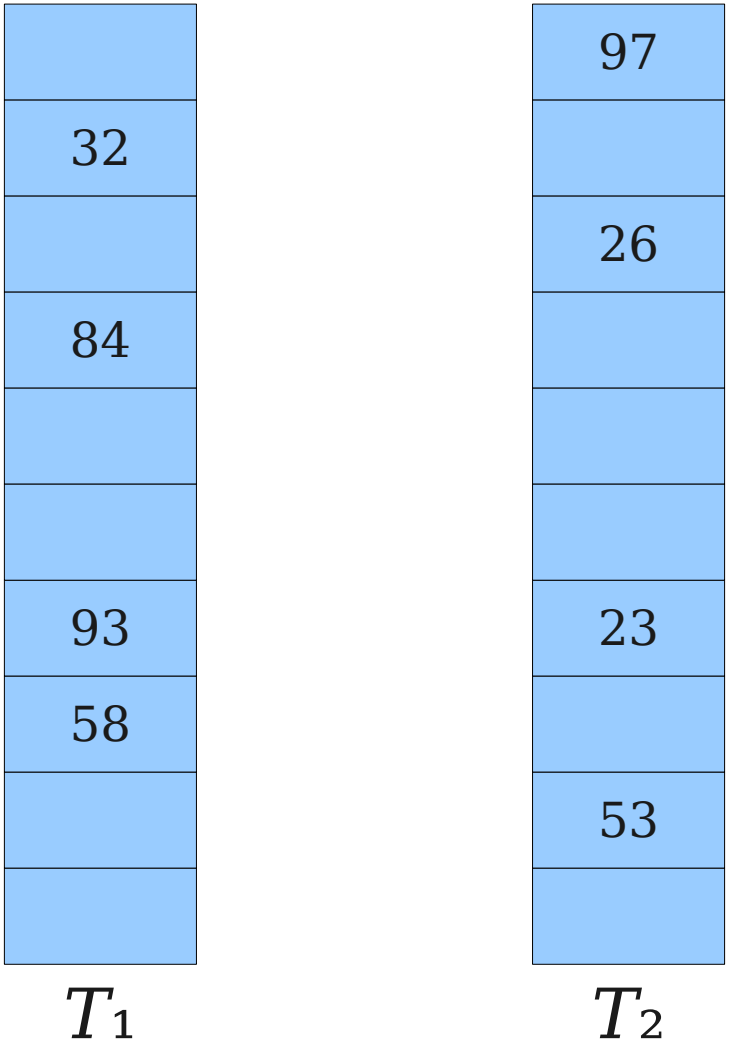
Cuckoo Hashing

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



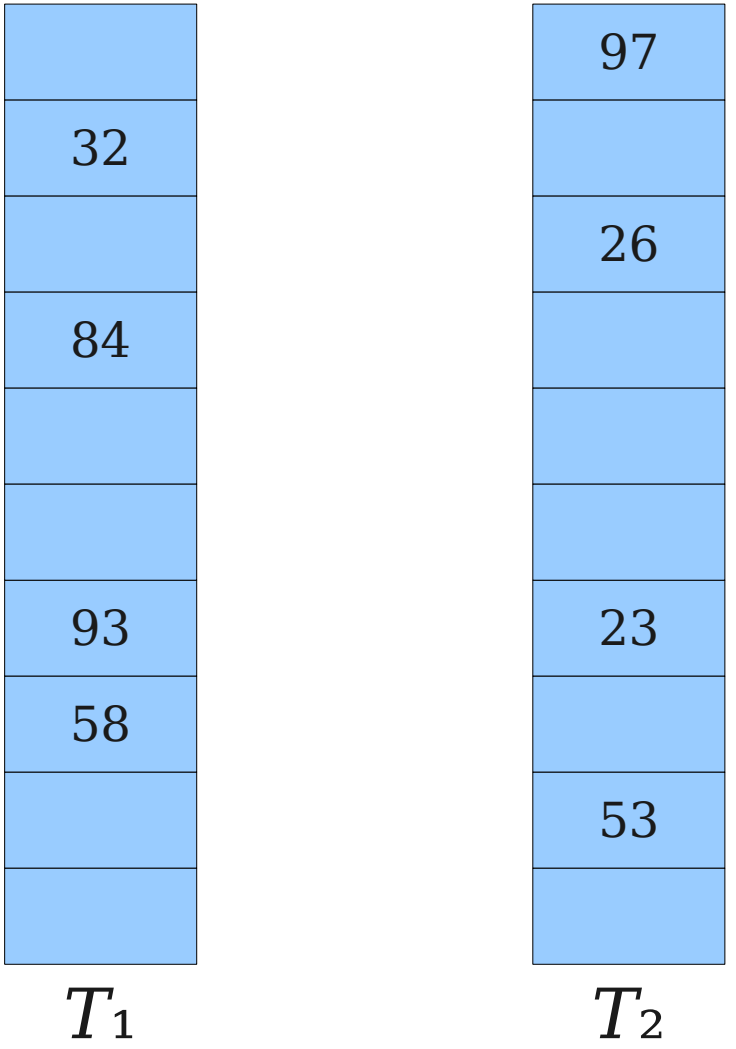
Cuckoo Hashing

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



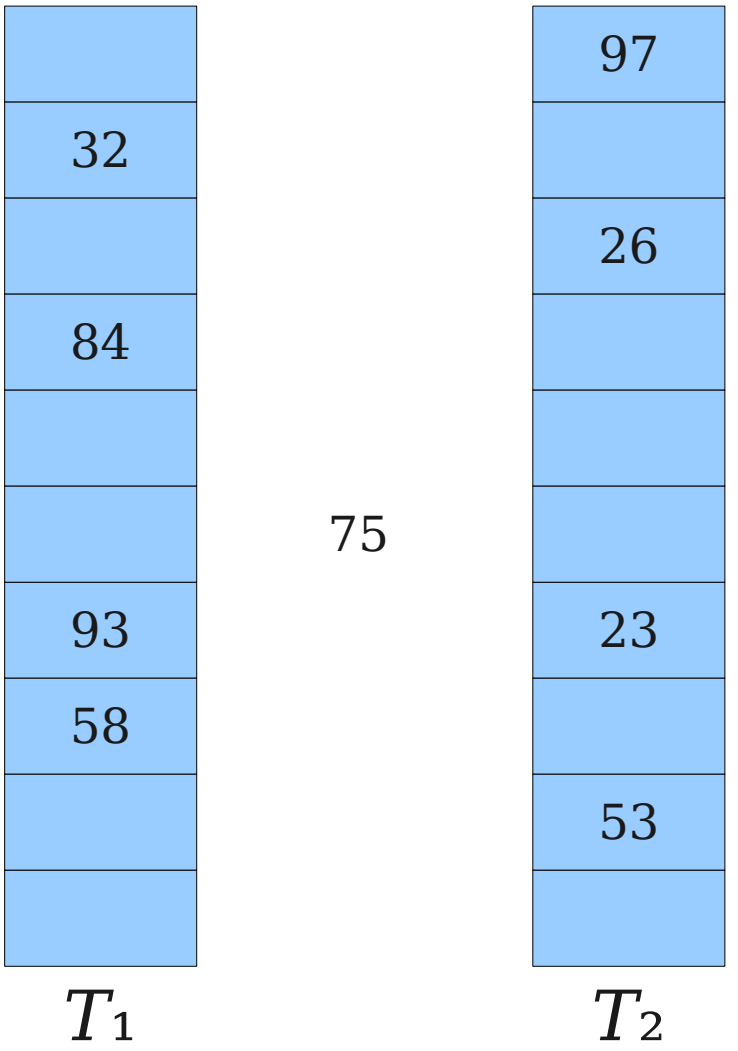
Cuckoo Hashing

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



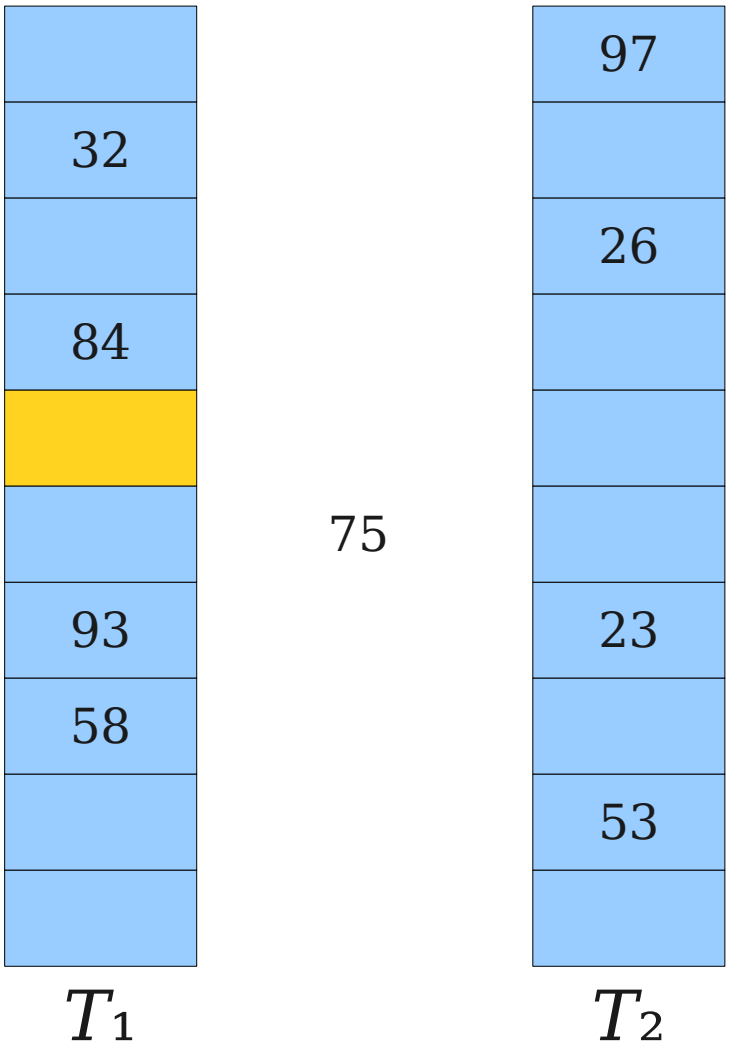
Cuckoo Hashing

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



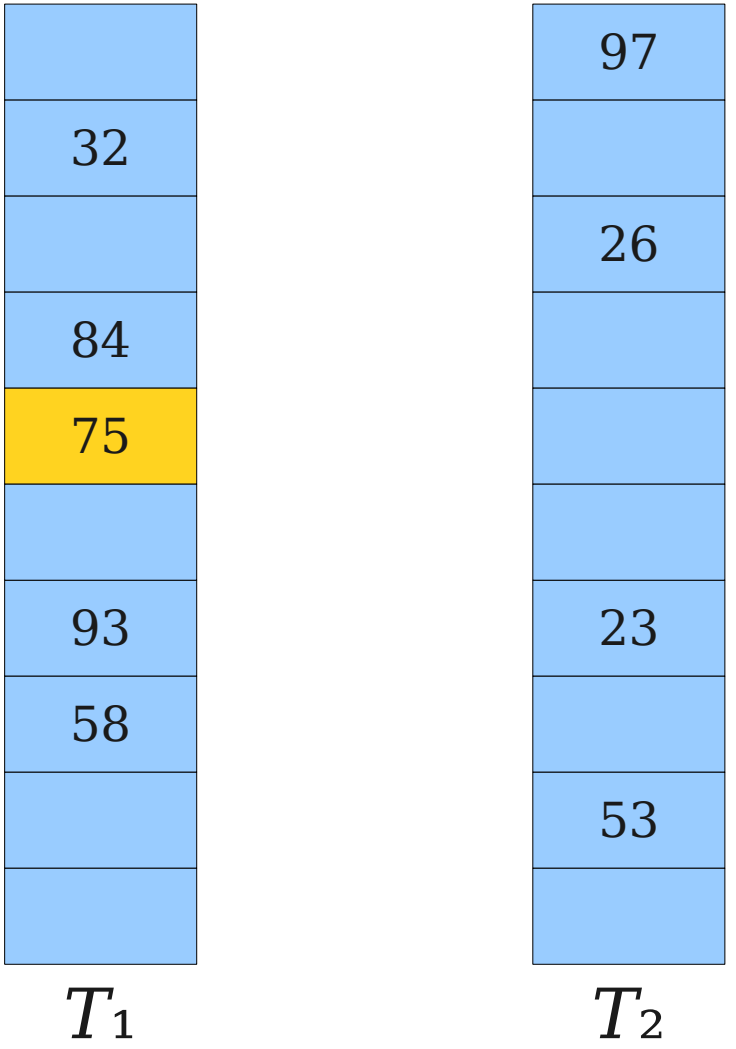
Cuckoo Hashing

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



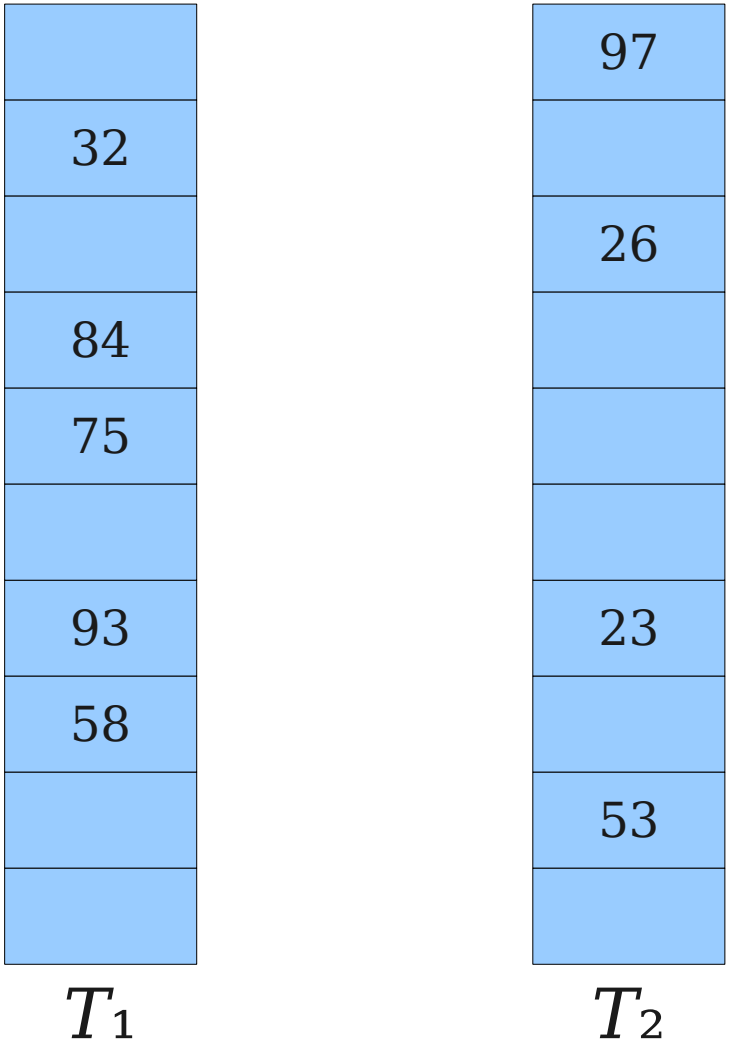
Cuckoo Hashing

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



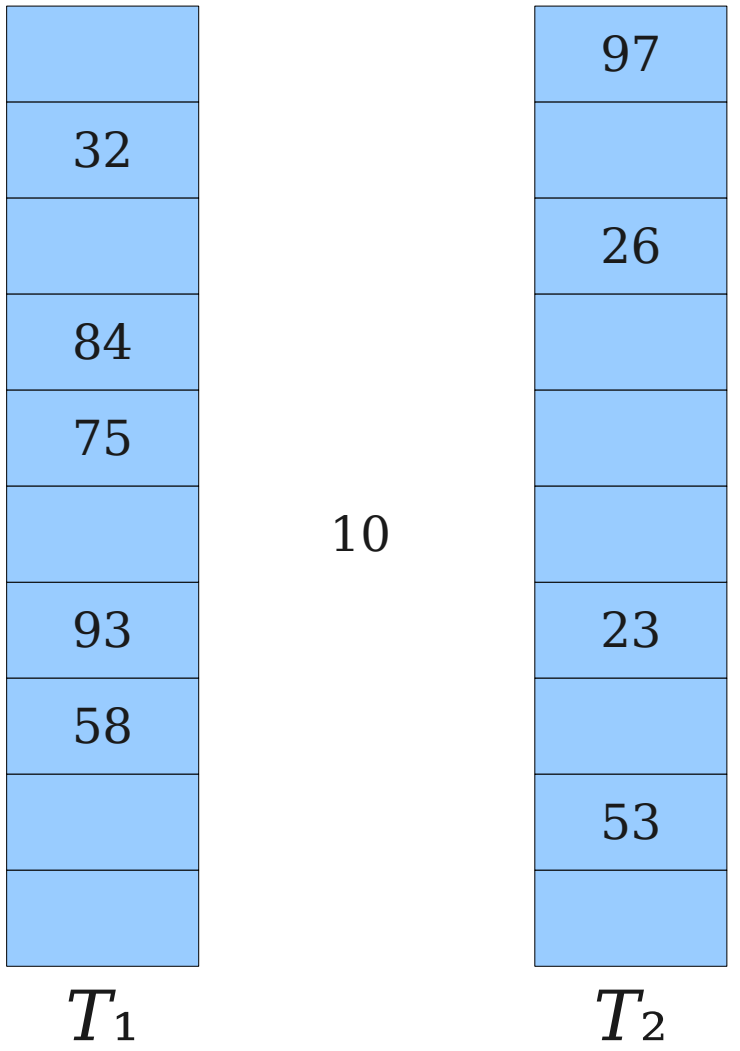
Cuckoo Hashing

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



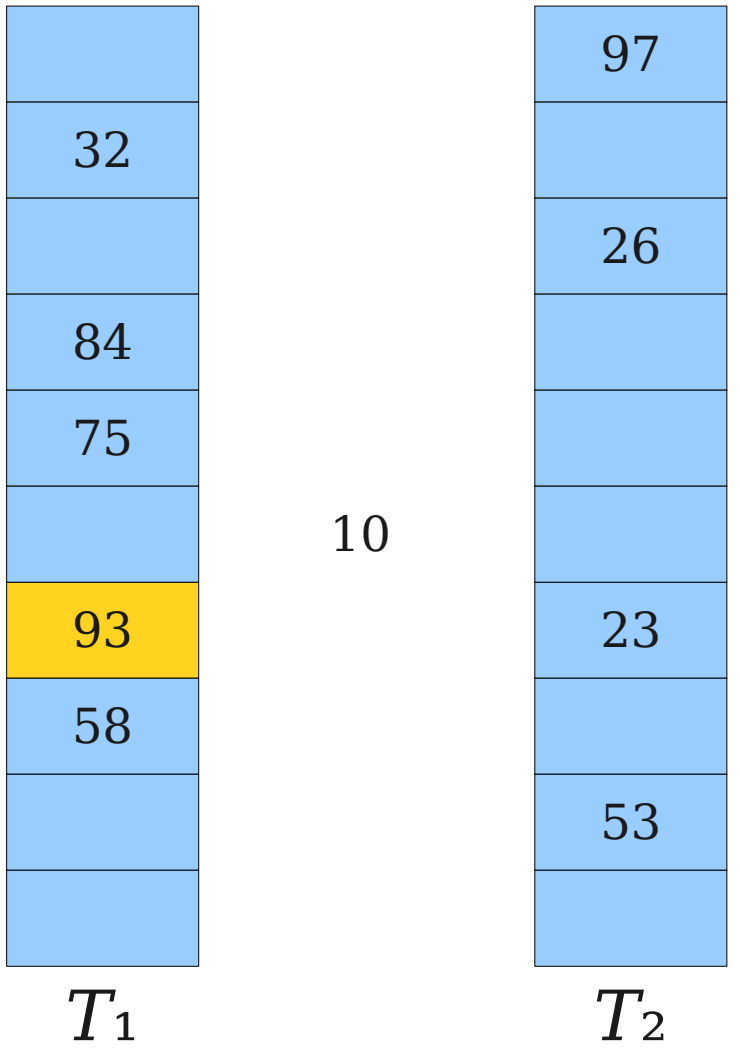
Cuckoo Hashing

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



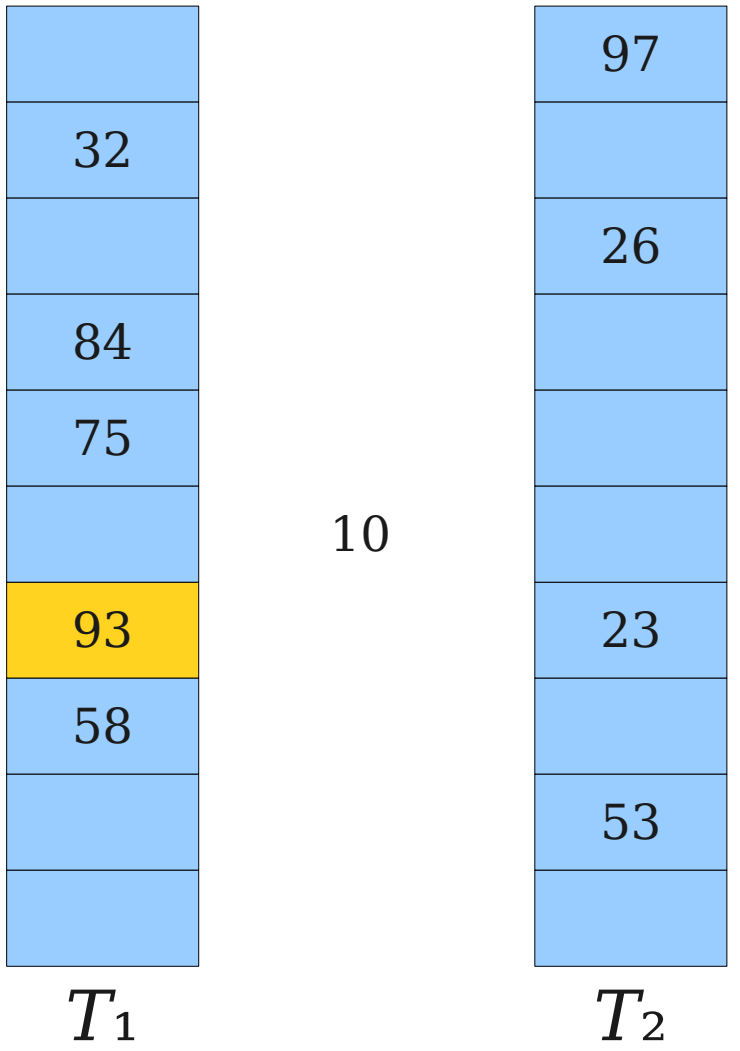
Cuckoo Hashing

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



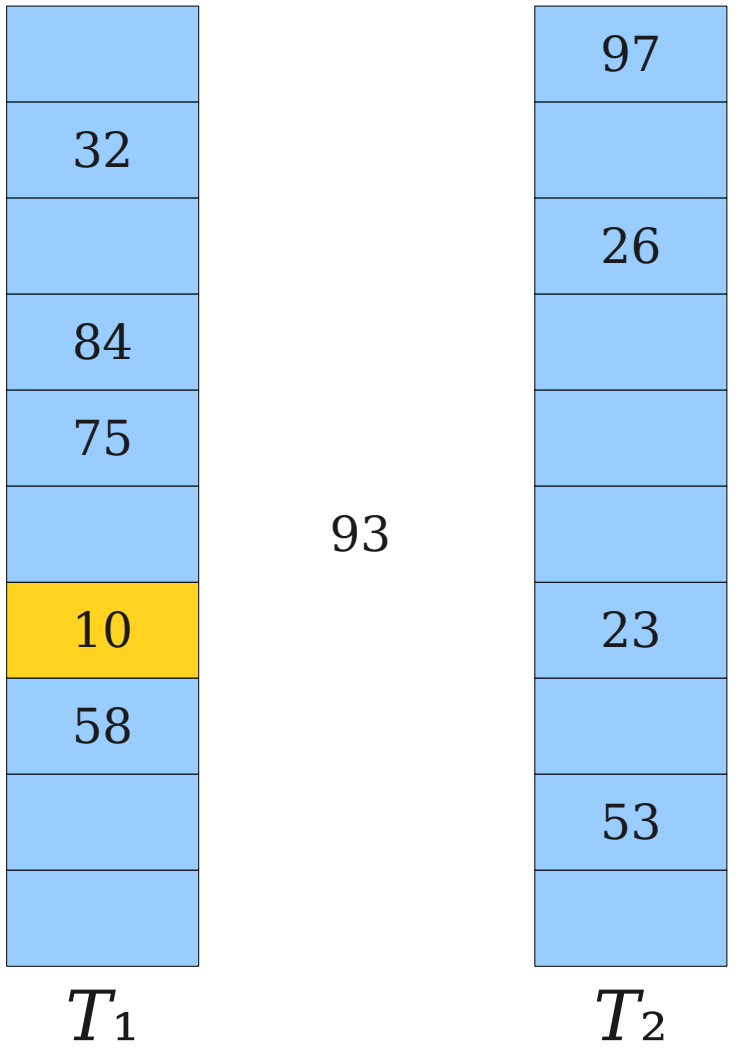
Cuckoo Hashing

- 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.



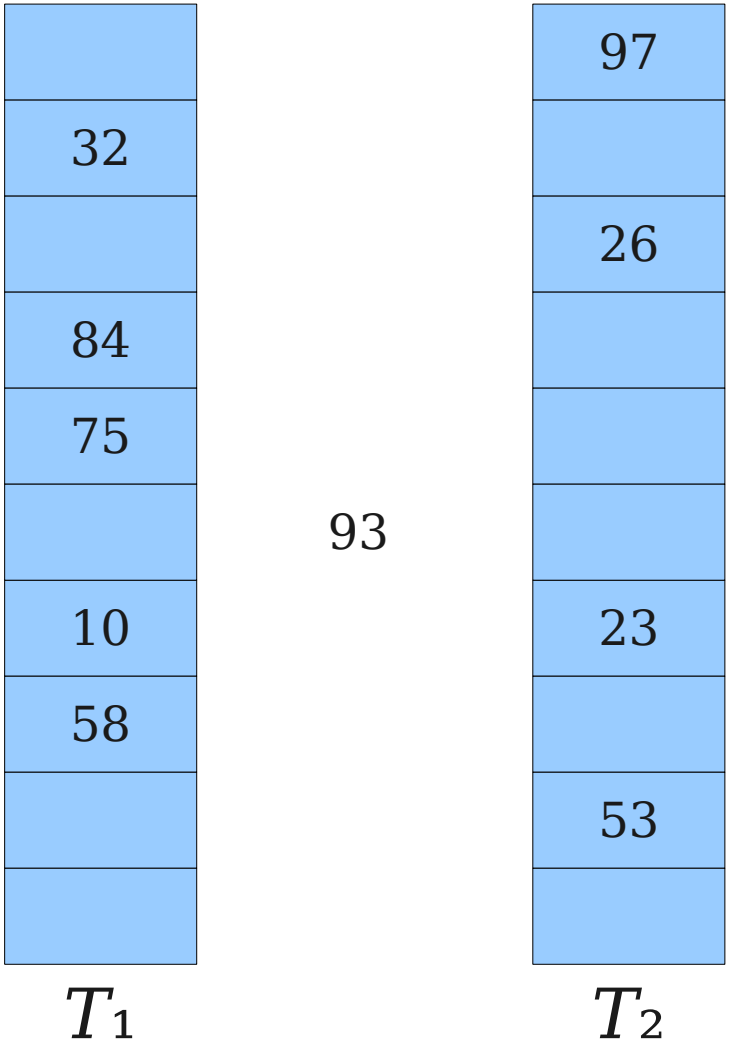
Cuckoo Hashing

- 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.



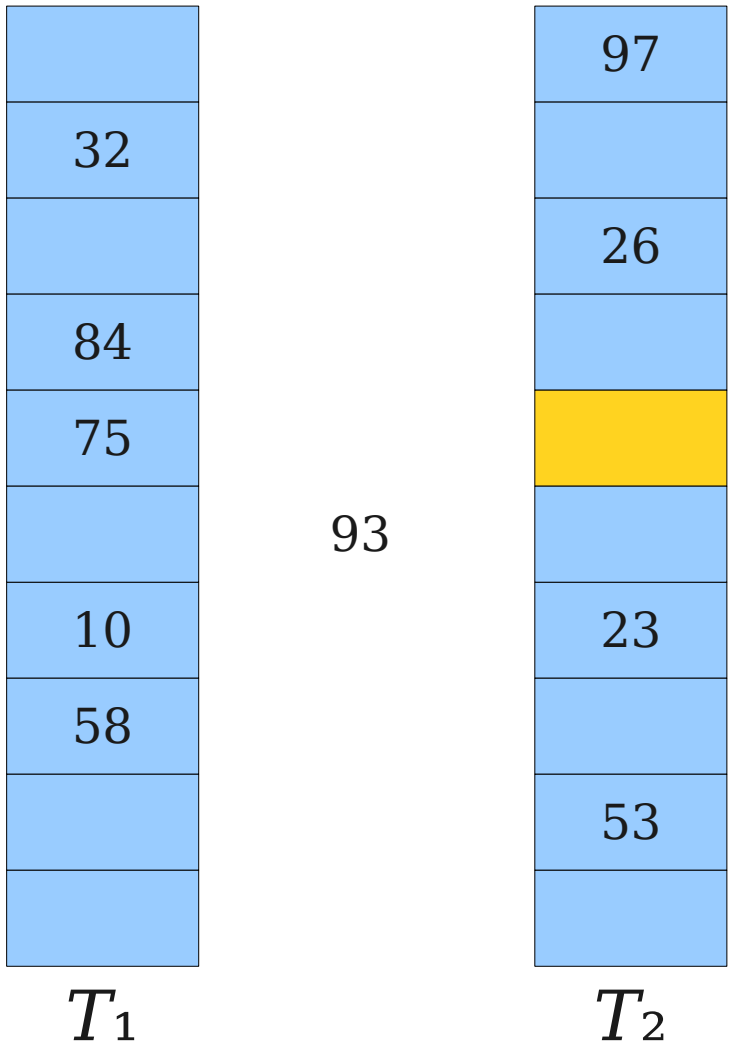
Cuckoo Hashing

- 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.



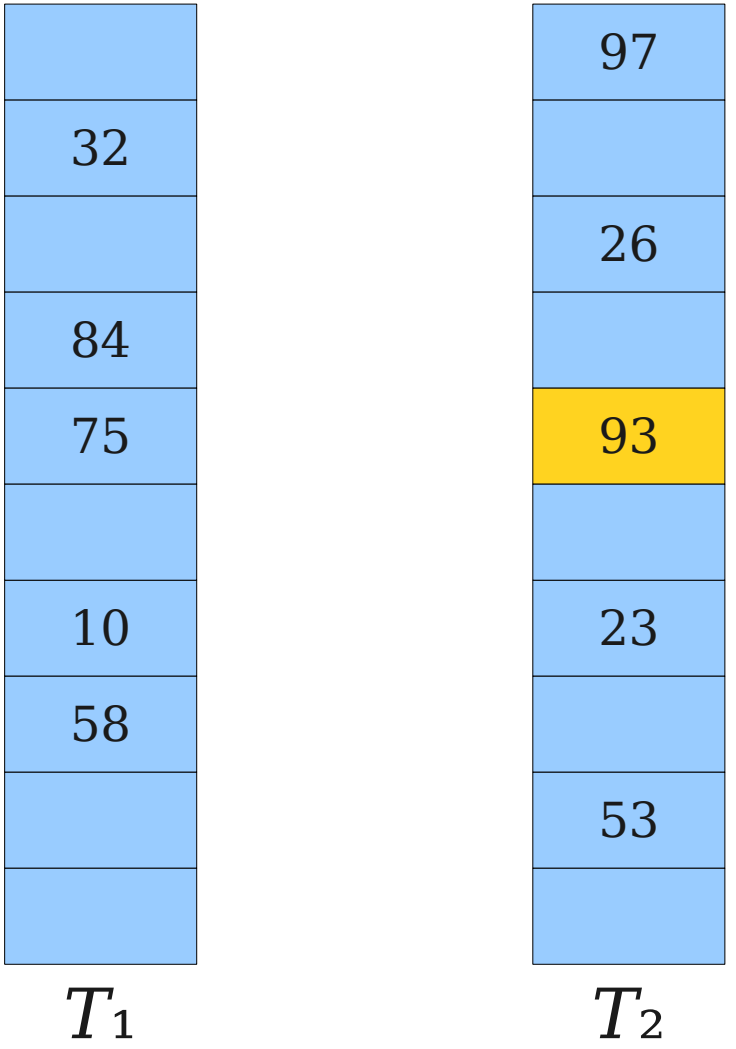
Cuckoo Hashing

- 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.



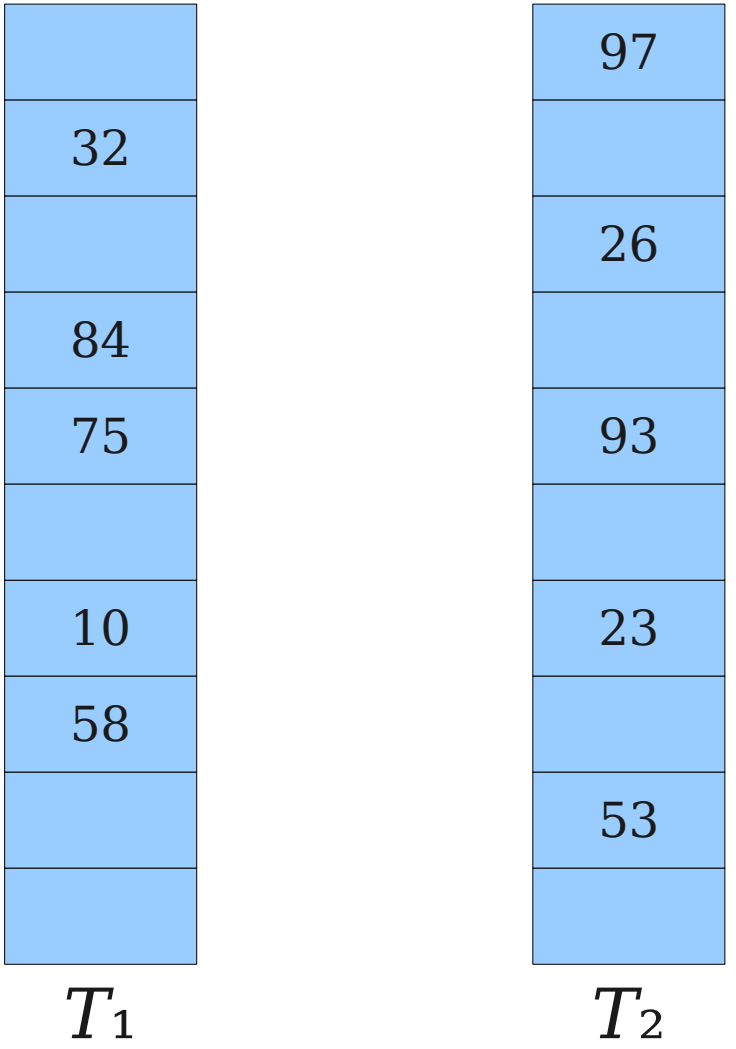
Cuckoo Hashing

- 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.



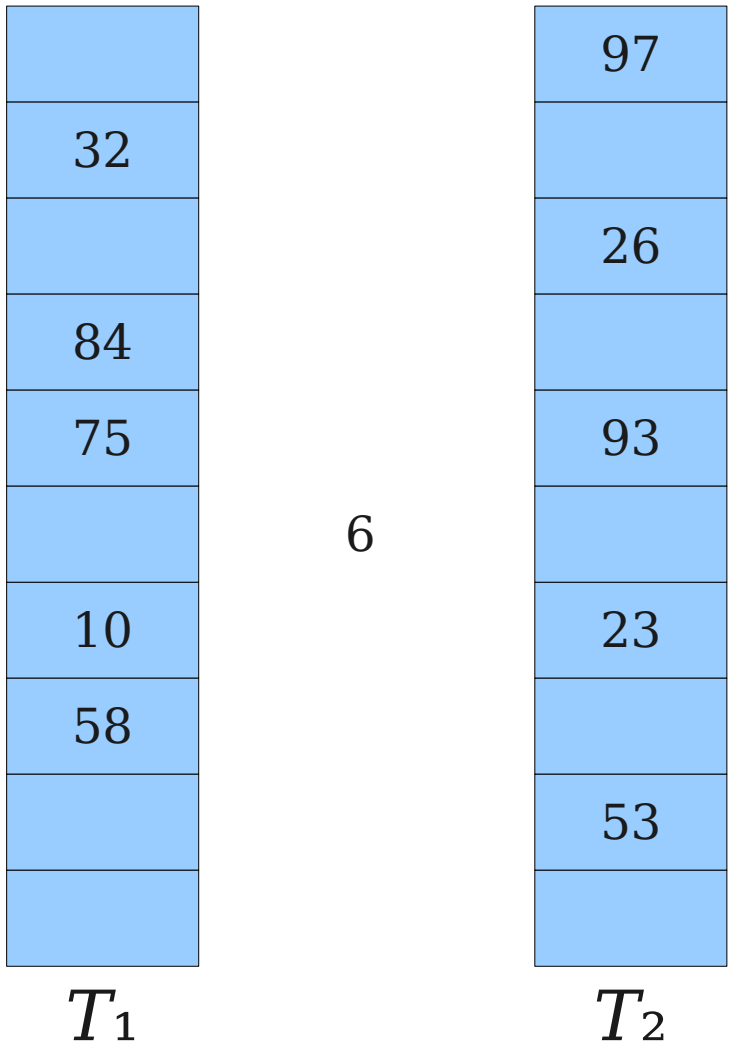
Cuckoo Hashing

- 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.



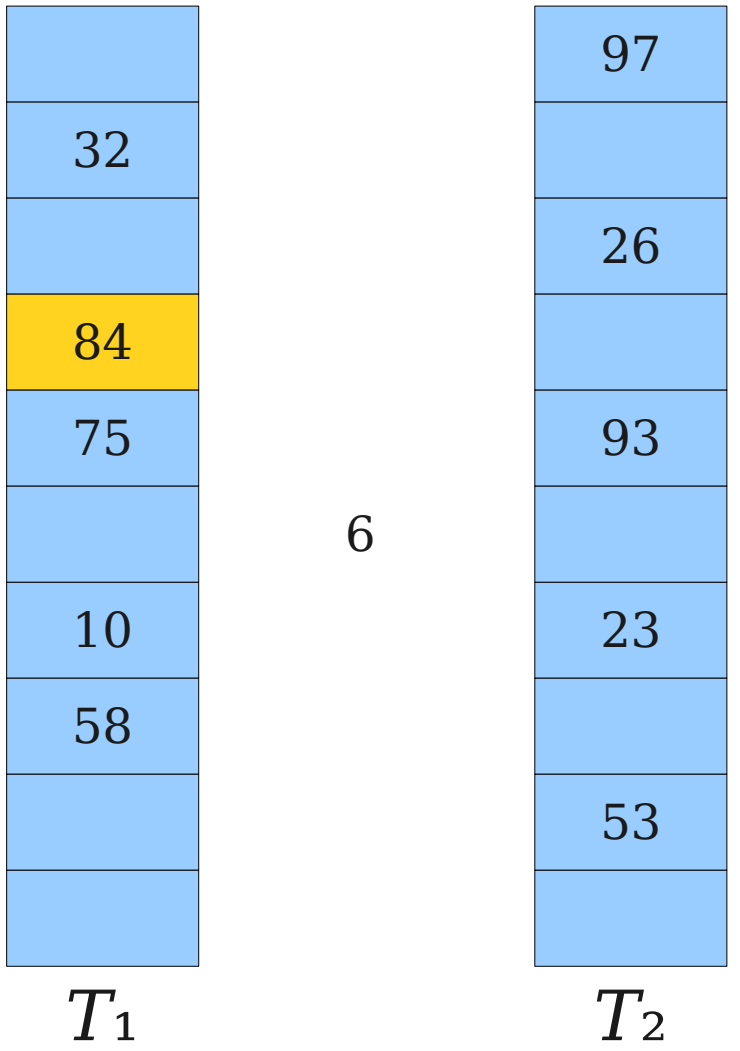
Cuckoo Hashing

- 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.



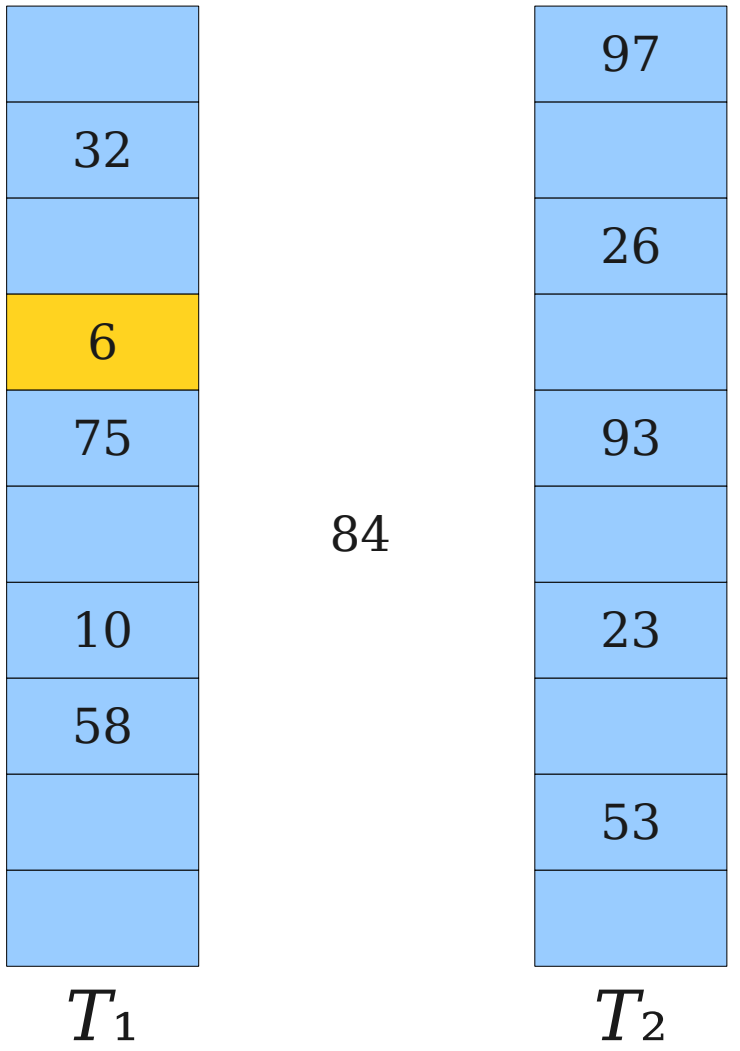
Cuckoo Hashing

- 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.



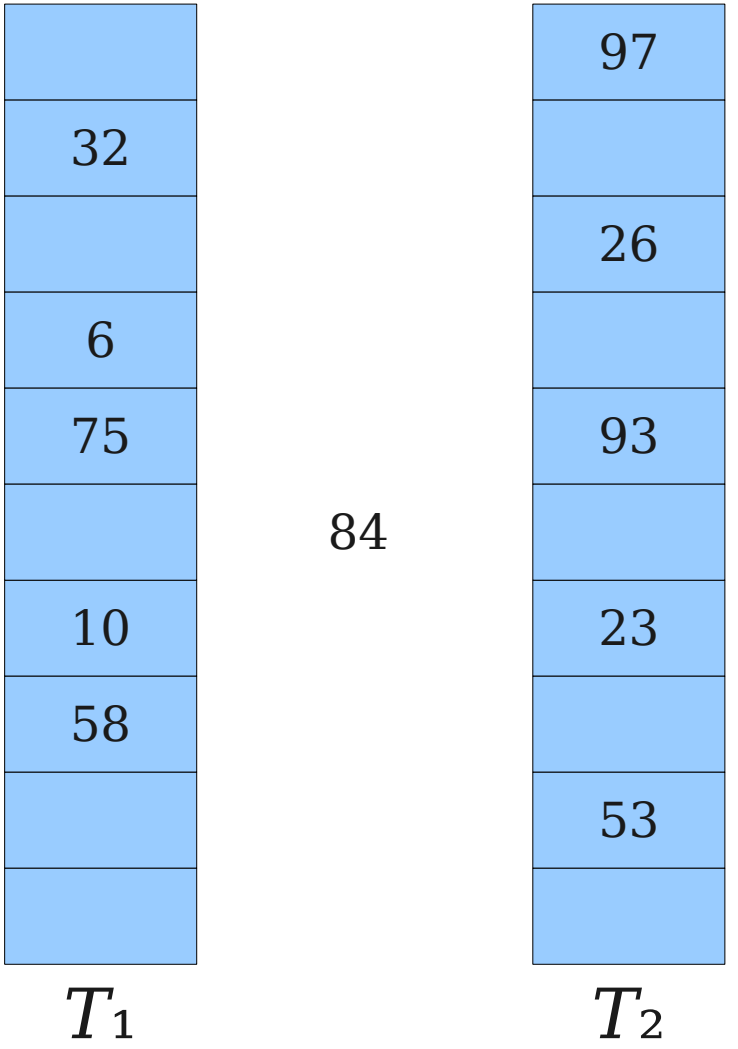
Cuckoo Hashing

- 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.



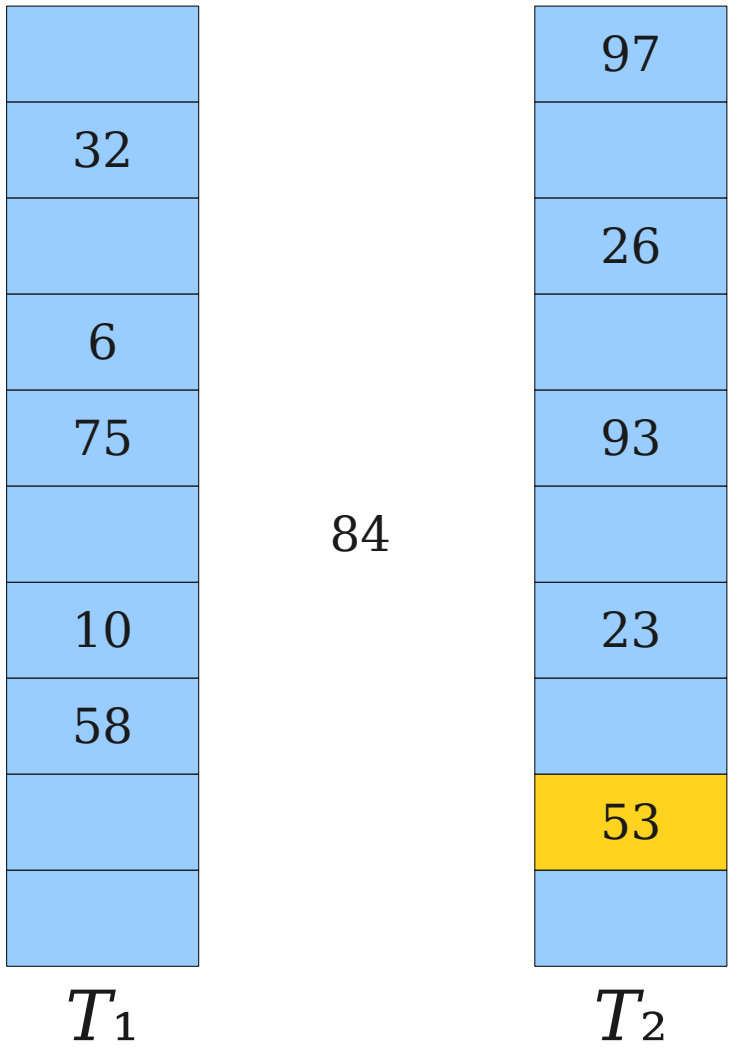
Cuckoo Hashing

- 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.



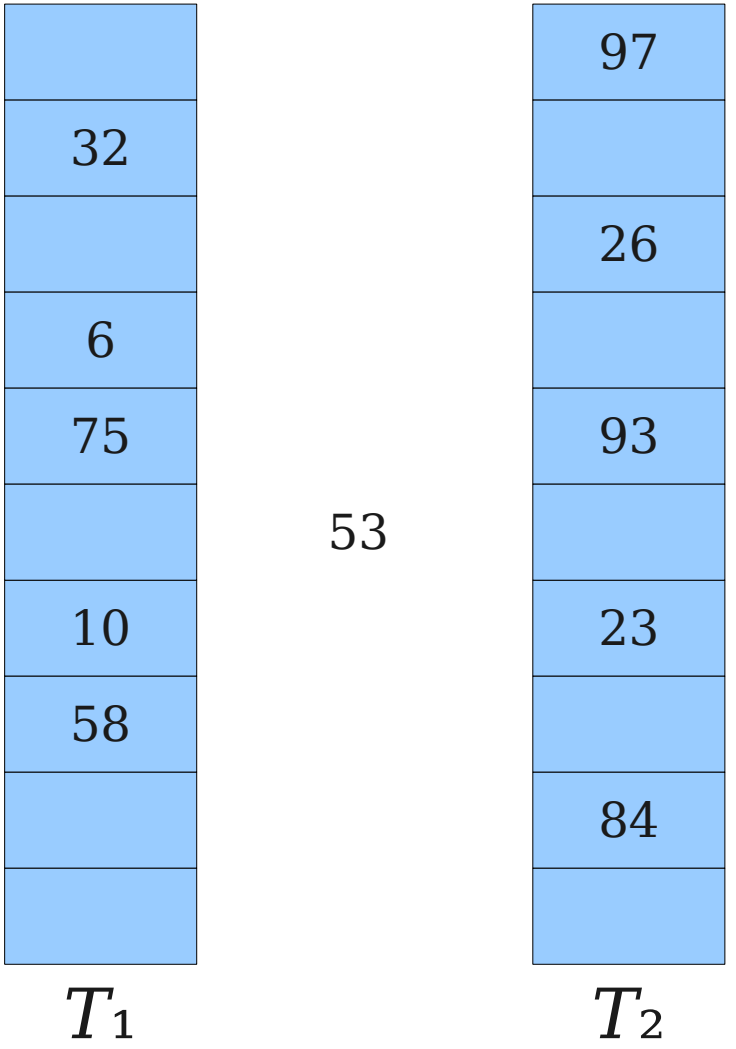
Cuckoo Hashing

- 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.



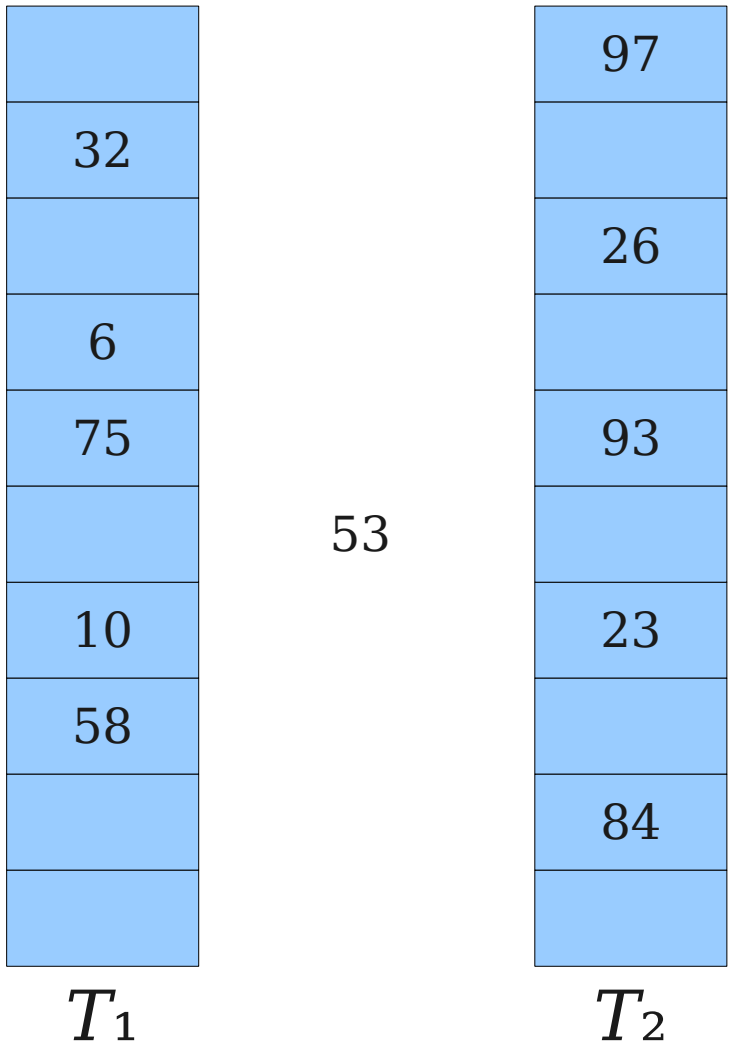
Cuckoo Hashing

- 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.



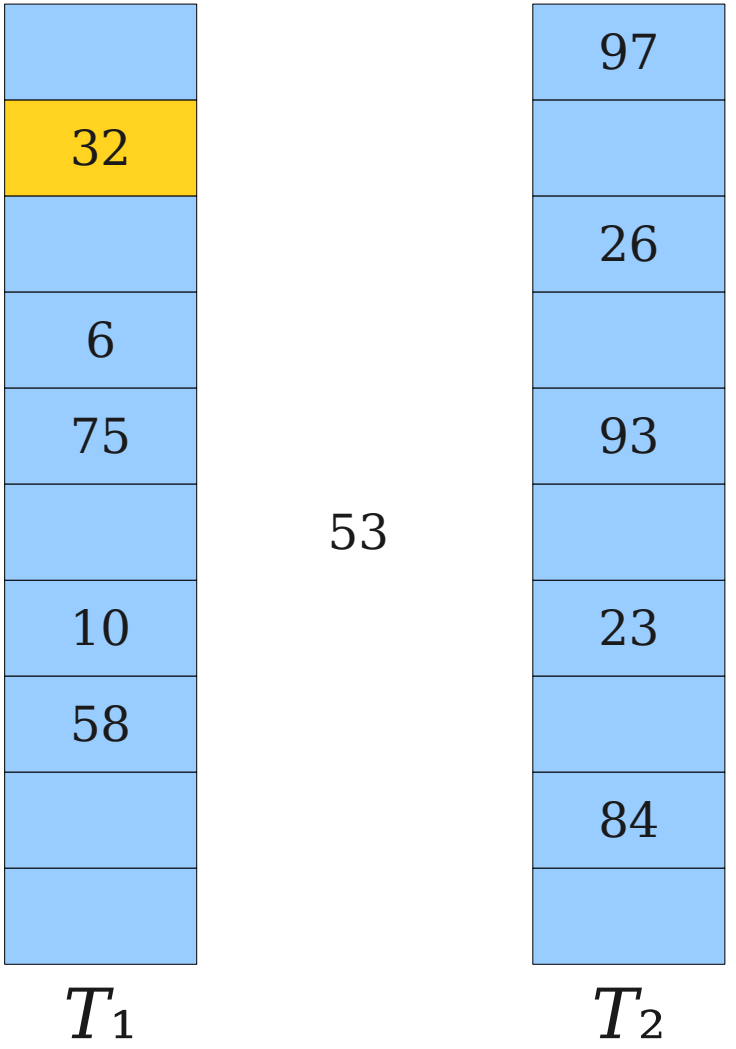
Cuckoo Hashing

- 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.



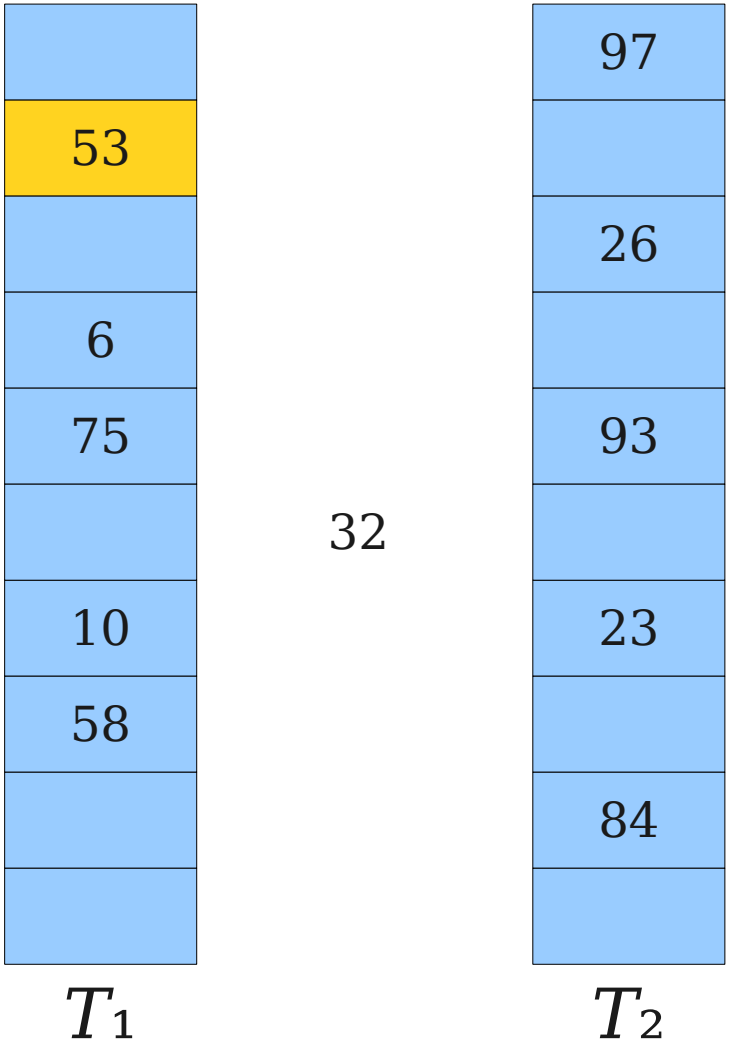
Cuckoo Hashing

- 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.



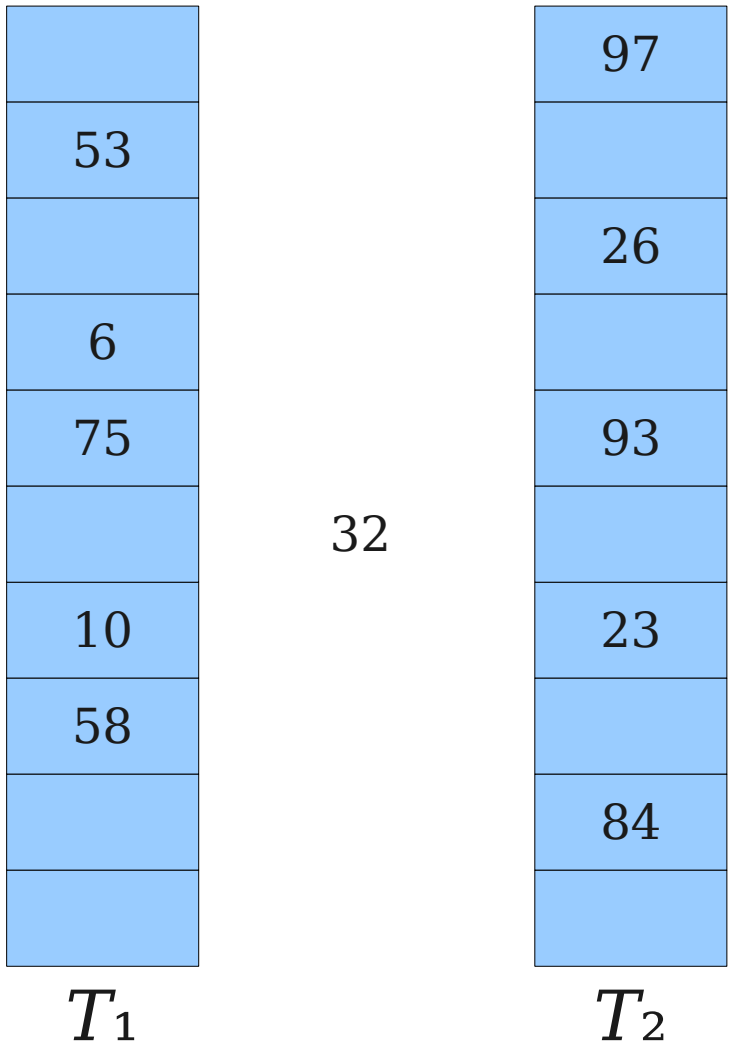
Cuckoo Hashing

- 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.



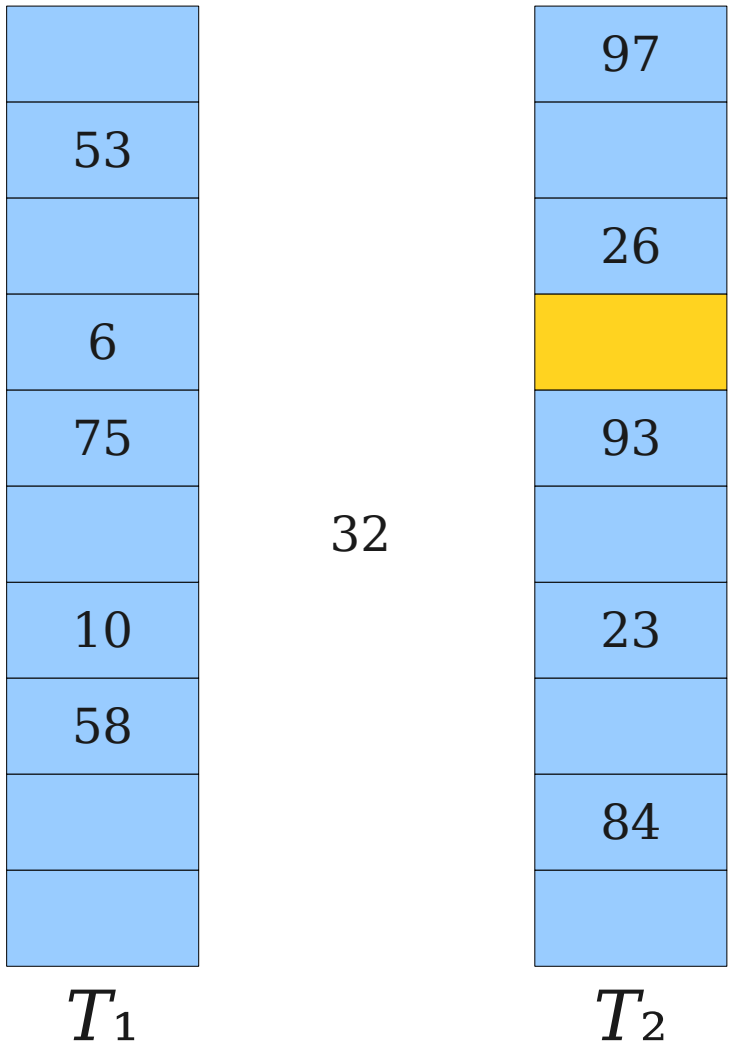
Cuckoo Hashing

- 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.



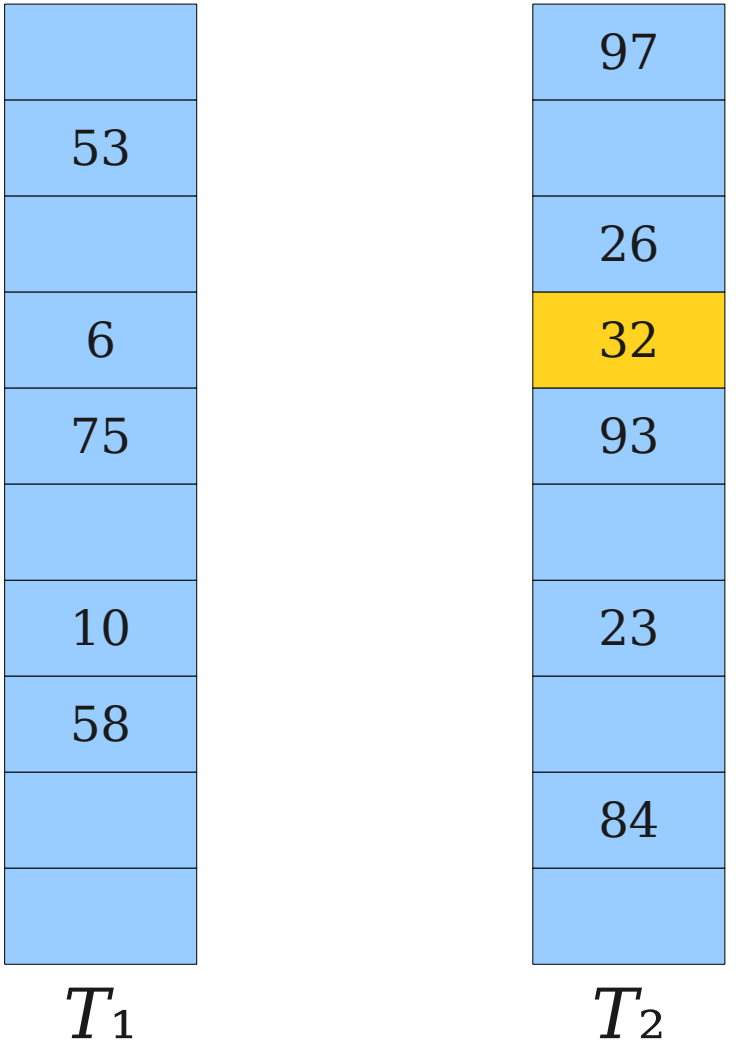
Cuckoo Hashing

- 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.



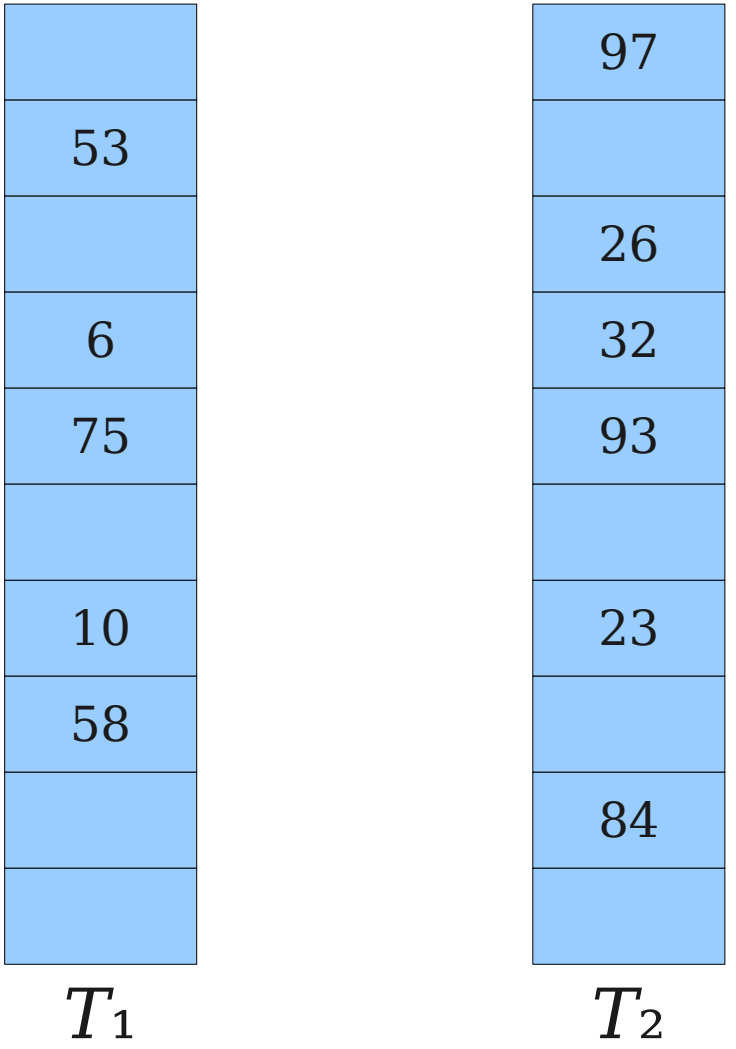
Cuckoo Hashing

- 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.

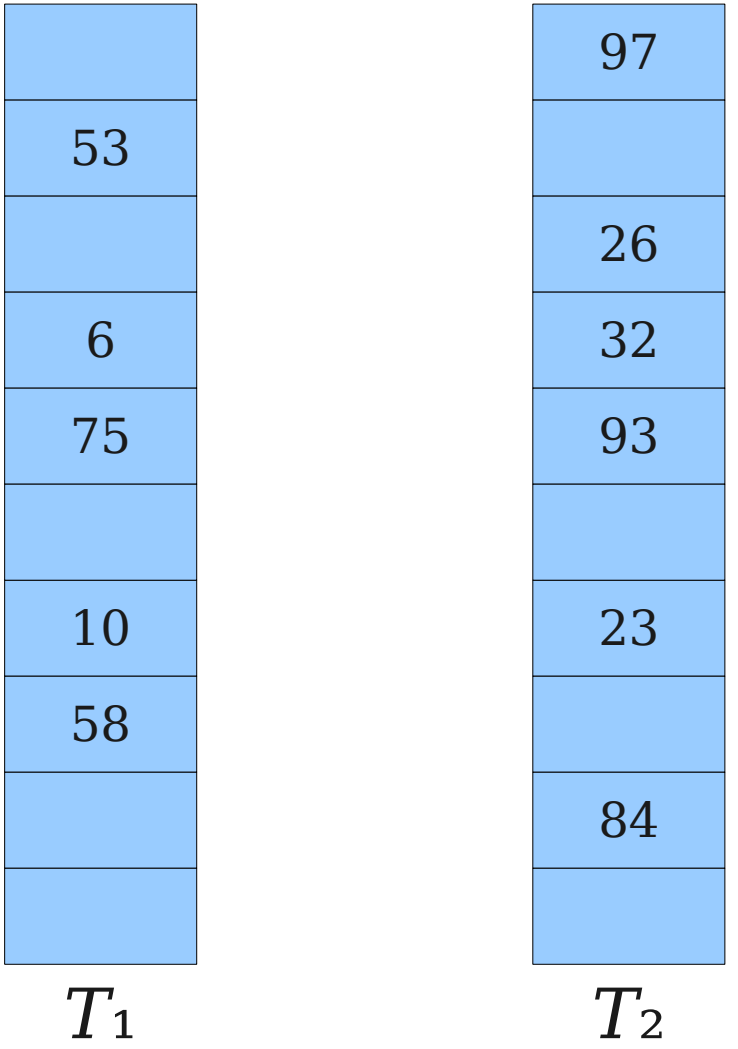


Cuckoo Hashing

- 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.

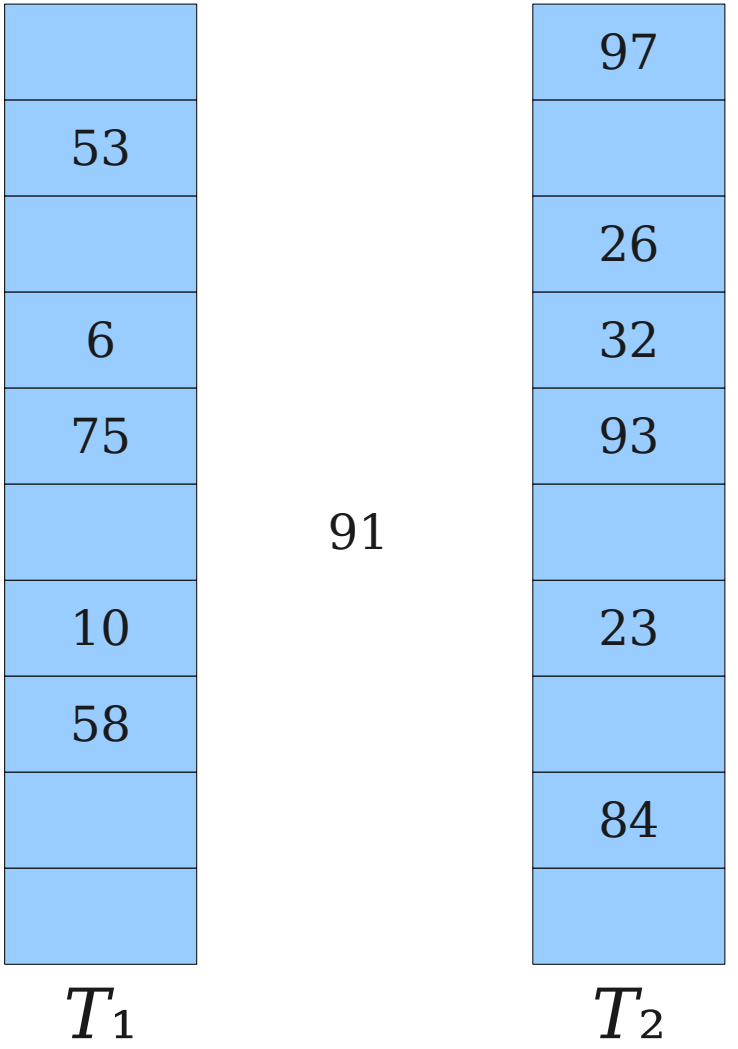
Cuckoo Hashing

- Insertions run into trouble if we run into a cycle.



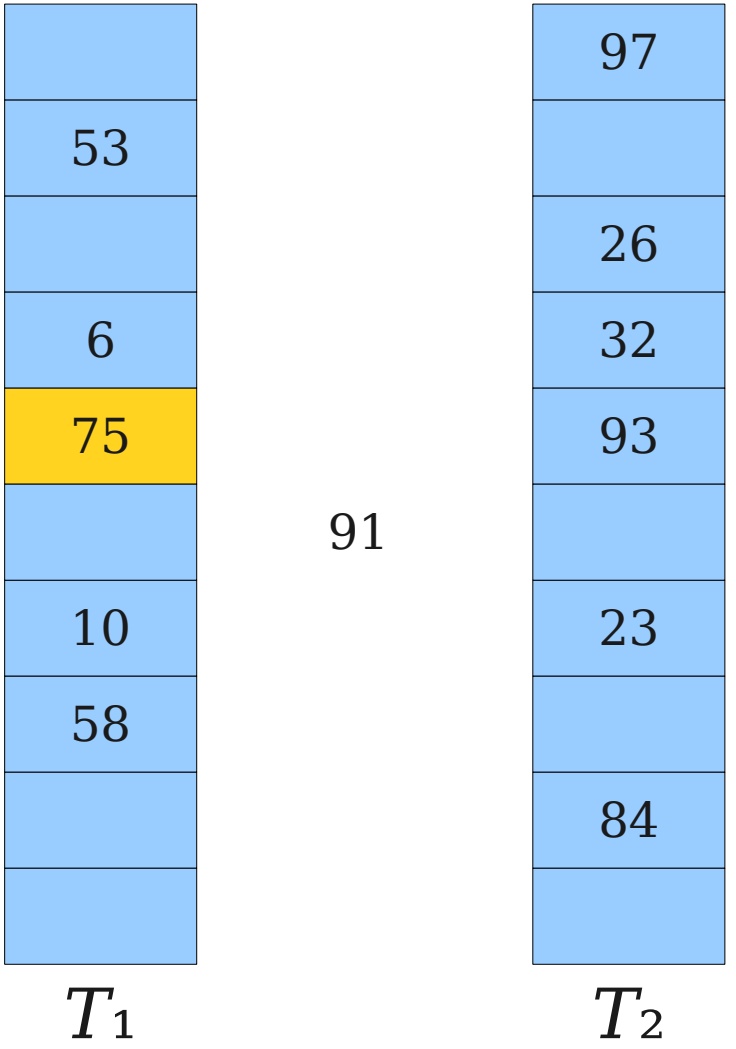
Cuckoo Hashing

- Insertions run into trouble if we run into a cycle.



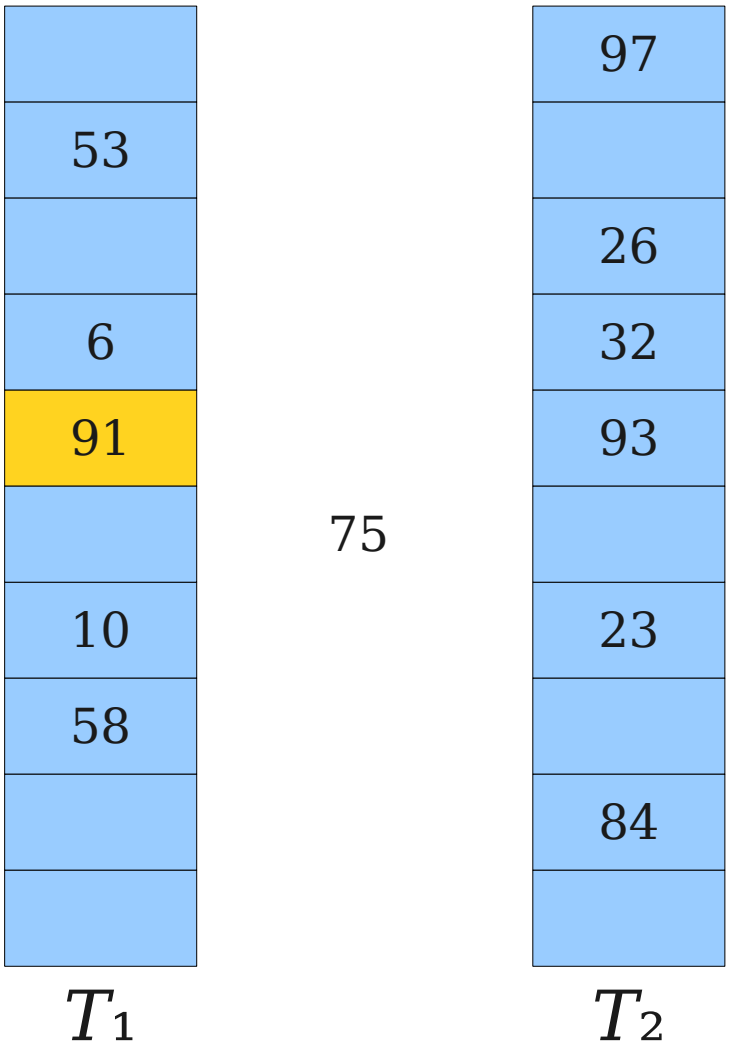
Cuckoo Hashing

- Insertions run into trouble if we run into a cycle.



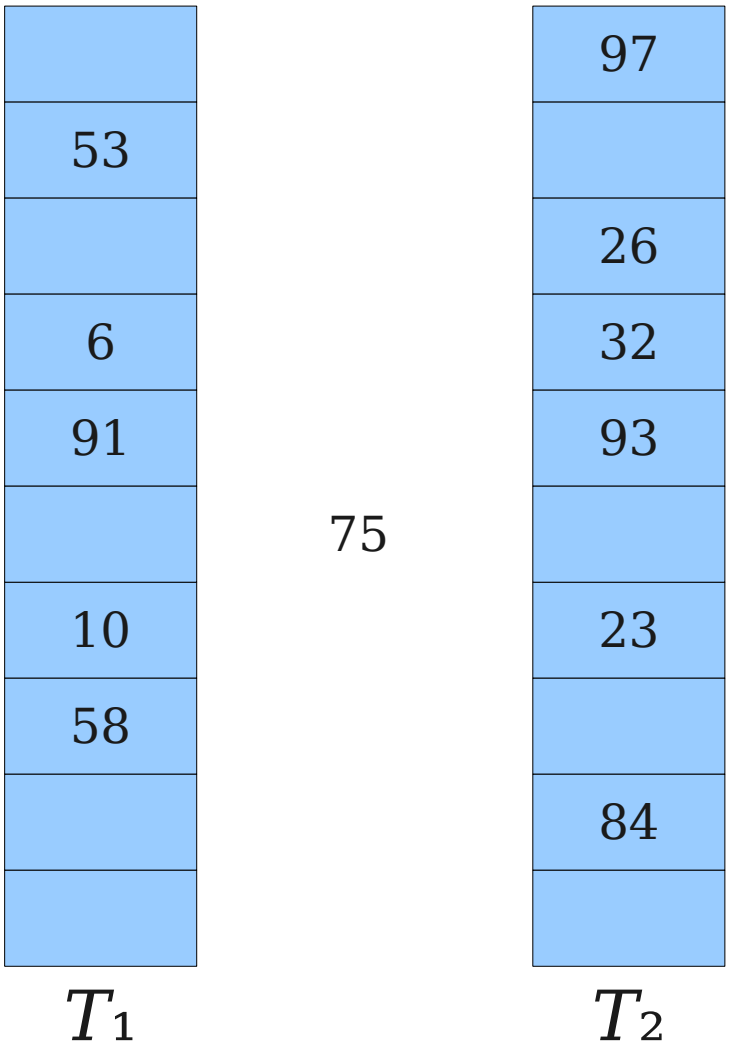
Cuckoo Hashing

- Insertions run into trouble if we run into a cycle.



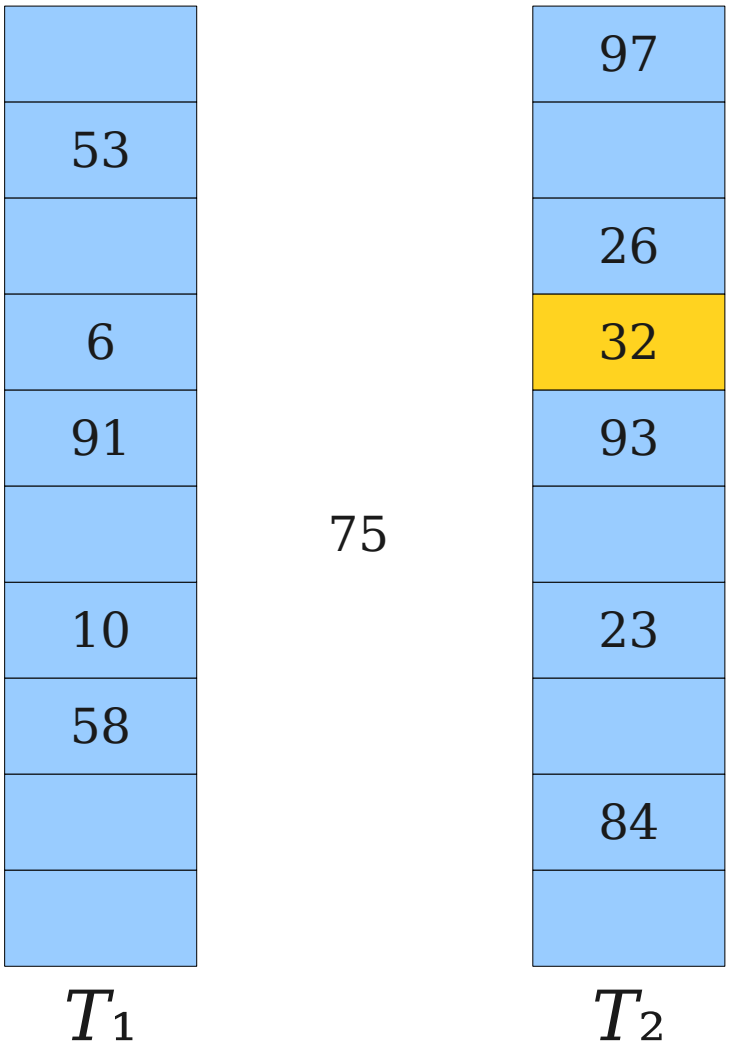
Cuckoo Hashing

- Insertions run into trouble if we run into a cycle.



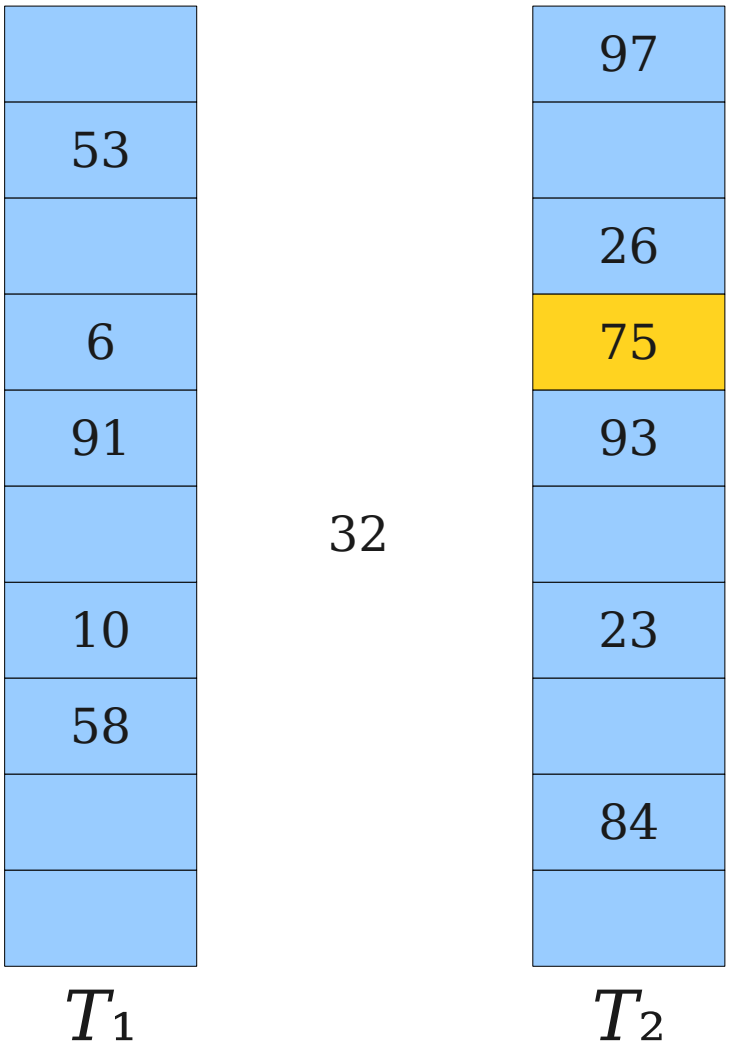
Cuckoo Hashing

- Insertions run into trouble if we run into a cycle.



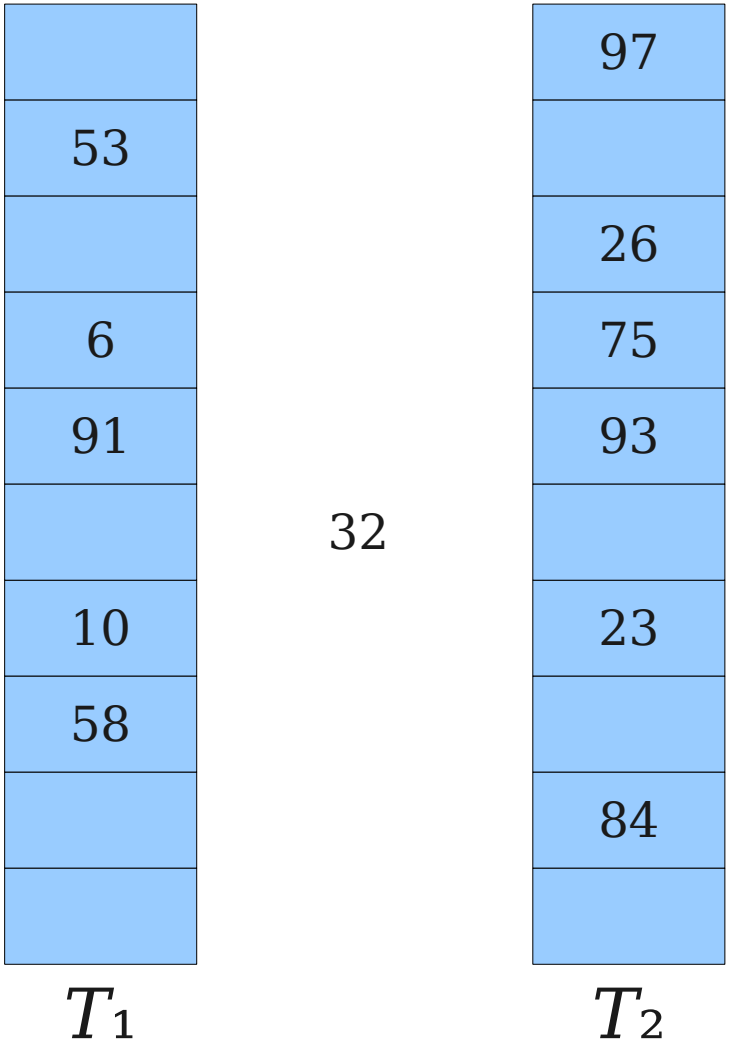
Cuckoo Hashing

- Insertions run into trouble if we run into a cycle.



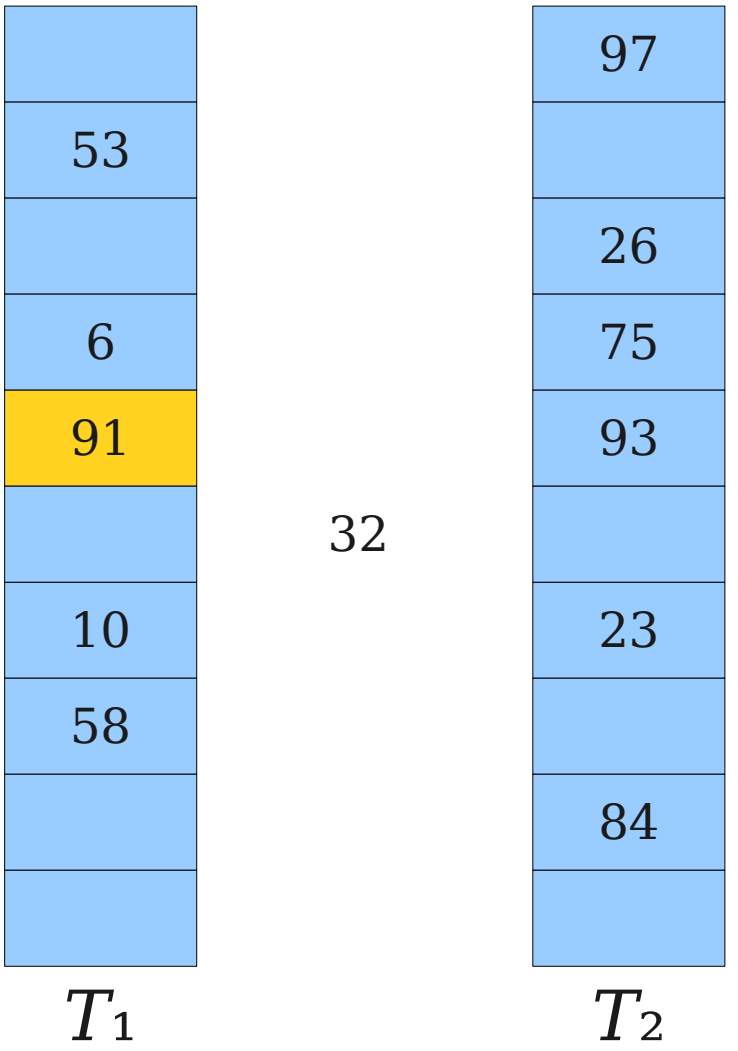
Cuckoo Hashing

- Insertions run into trouble if we run into a cycle.



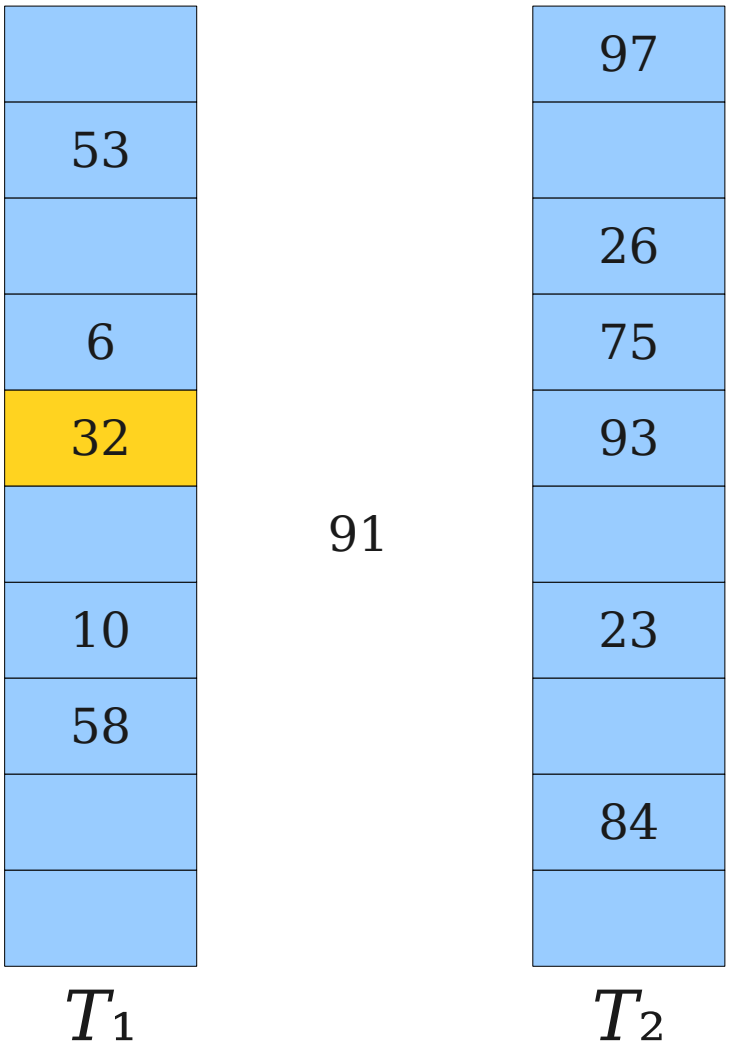
Cuckoo Hashing

- Insertions run into trouble if we run into a cycle.



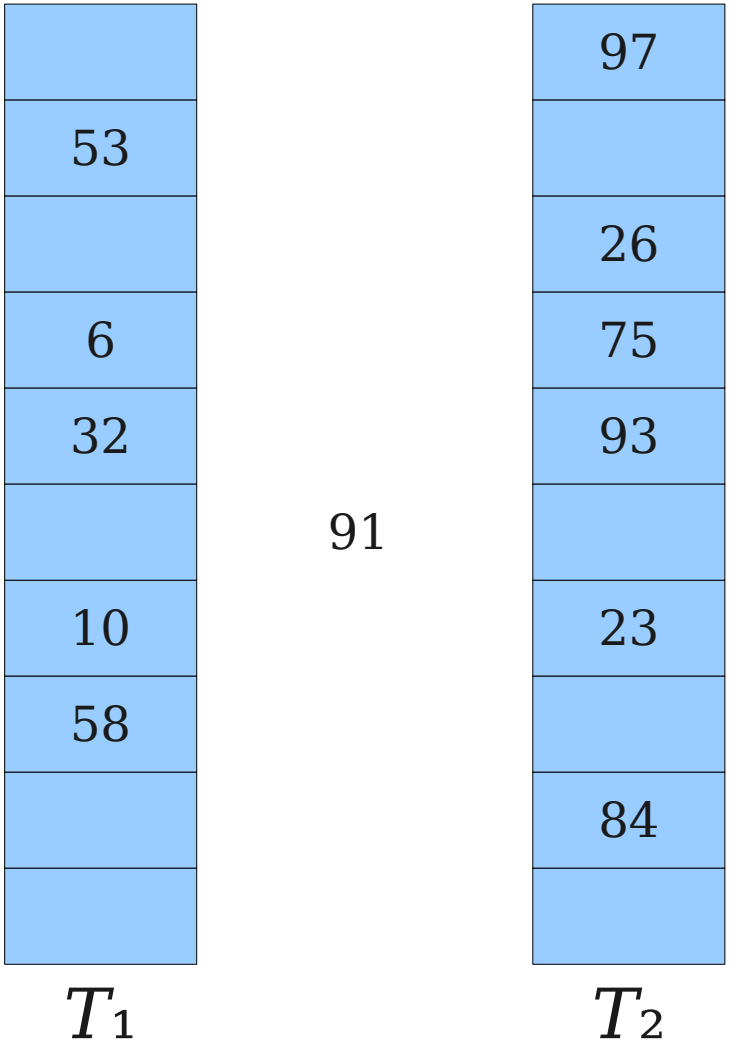
Cuckoo Hashing

- Insertions run into trouble if we run into a cycle.



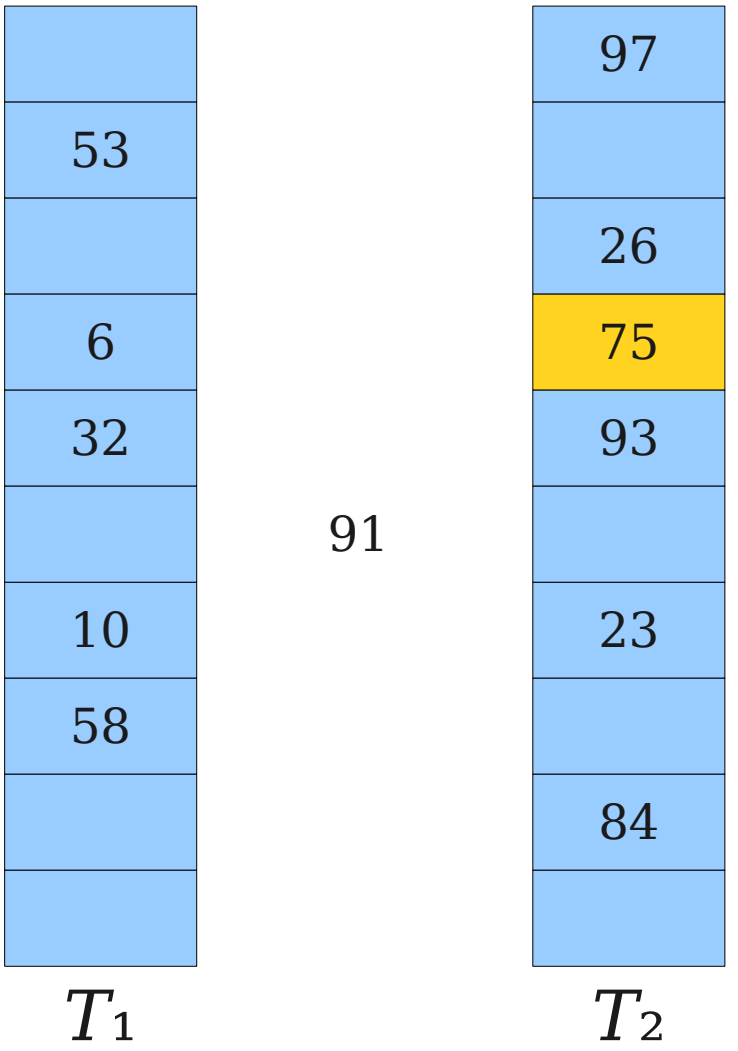
Cuckoo Hashing

- Insertions run into trouble if we run into a cycle.



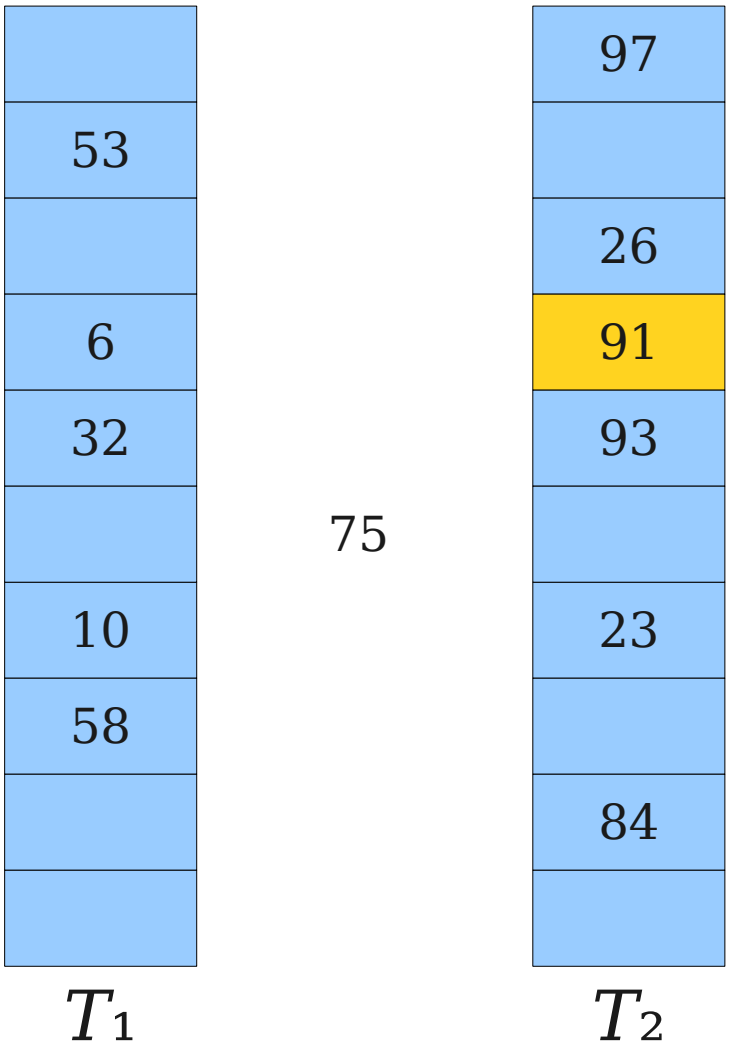
Cuckoo Hashing

- Insertions run into trouble if we run into a cycle.



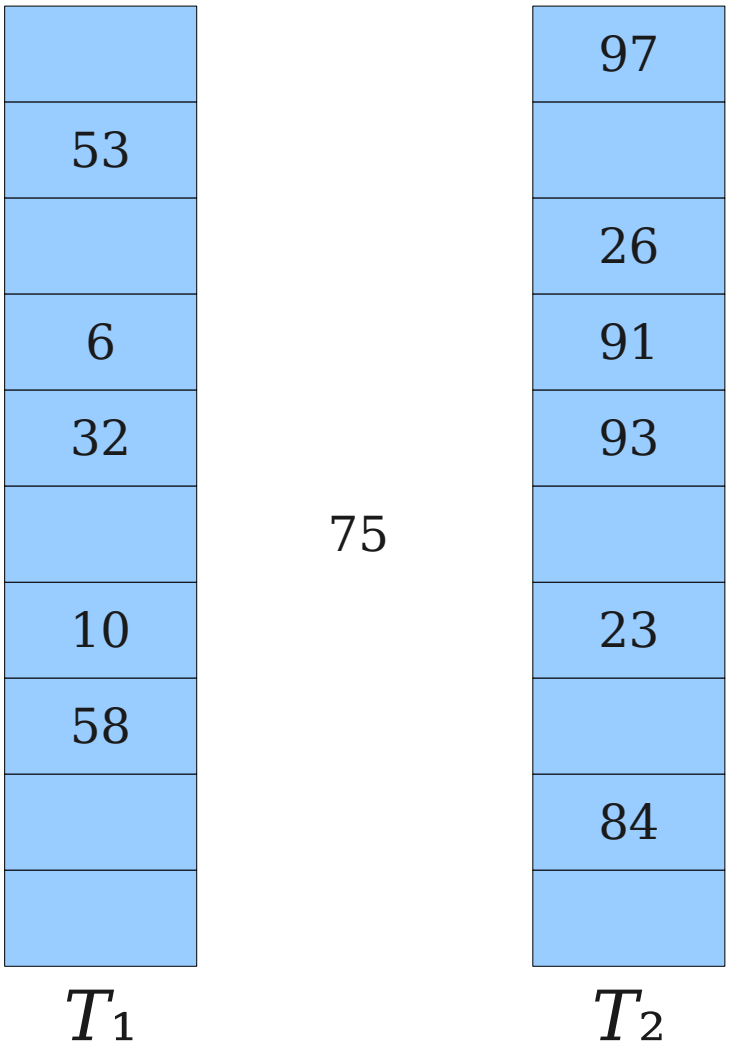
Cuckoo Hashing

- Insertions run into trouble if we run into a cycle.



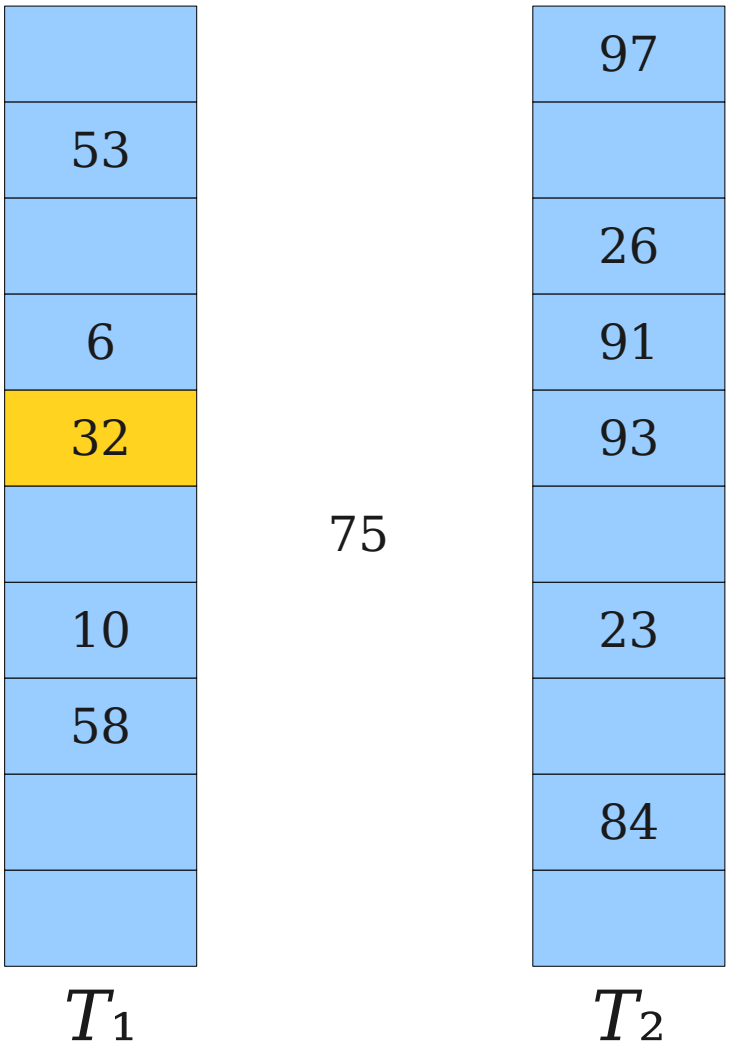
Cuckoo Hashing

- Insertions run into trouble if we run into a cycle.



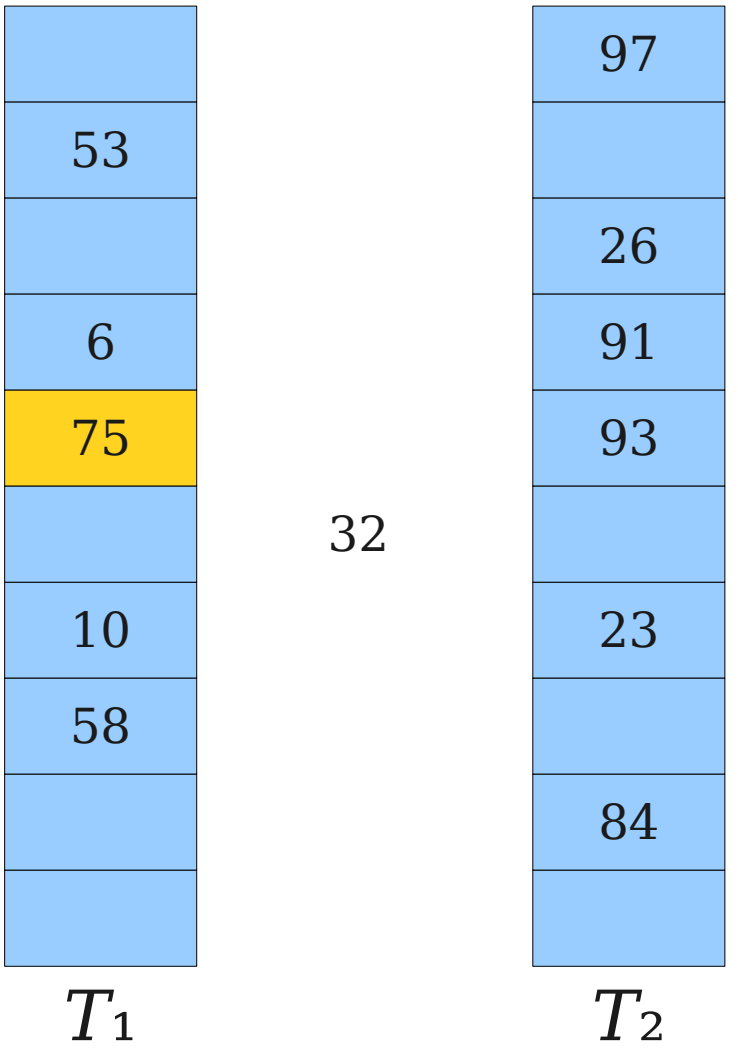
Cuckoo Hashing

- Insertions run into trouble if we run into a cycle.



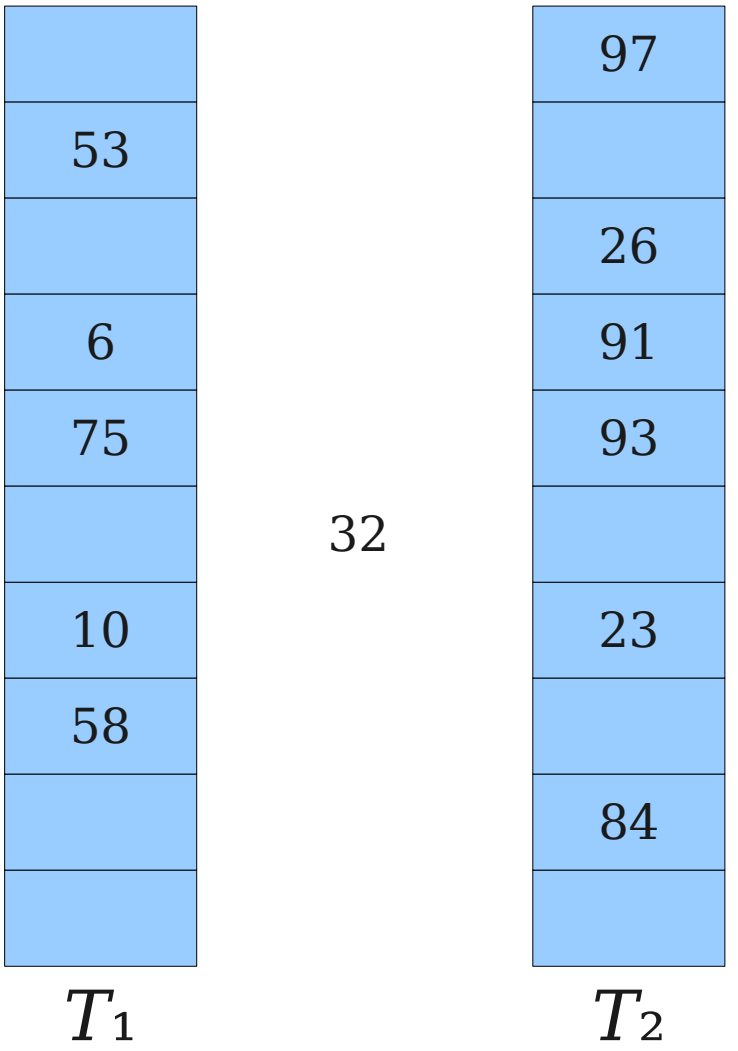
Cuckoo Hashing

- Insertions run into trouble if we run into a cycle.



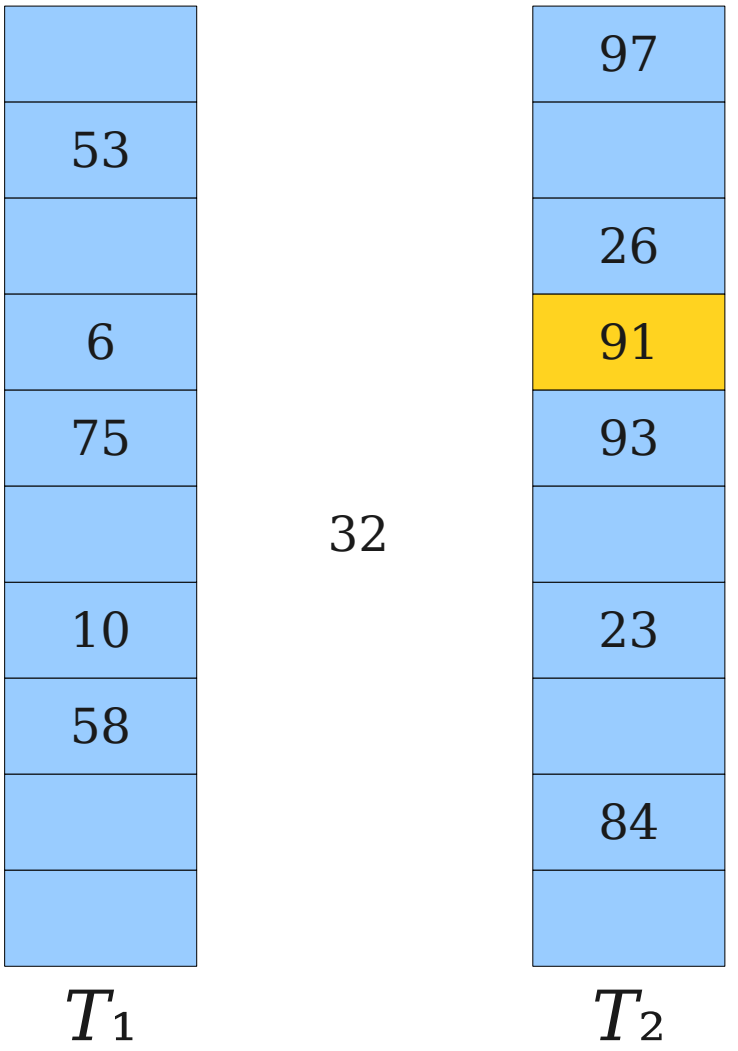
Cuckoo Hashing

- Insertions run into trouble if we run into a cycle.



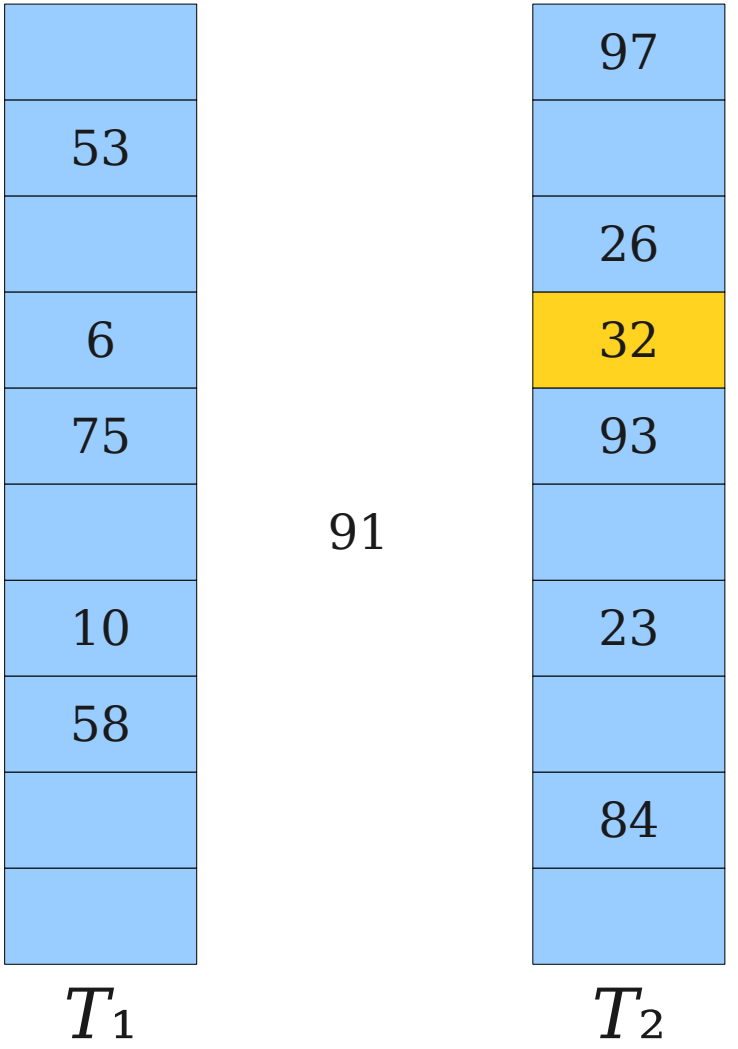
Cuckoo Hashing

- Insertions run into trouble if we run into a cycle.



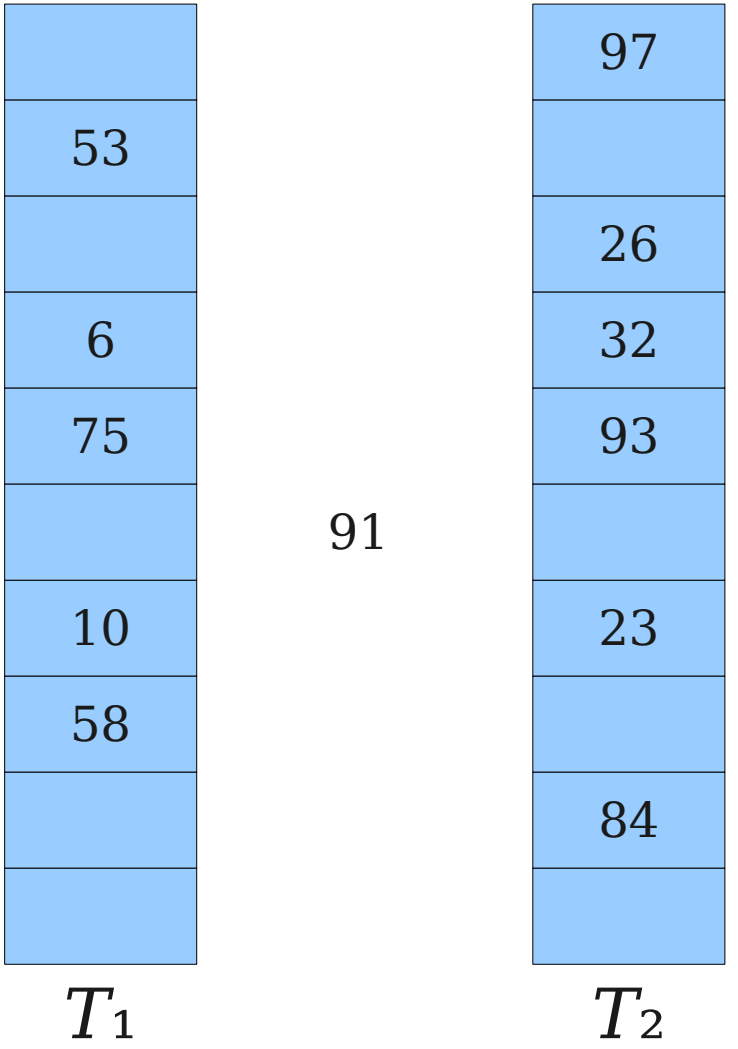
Cuckoo Hashing

- Insertions run into trouble if we run into a cycle.



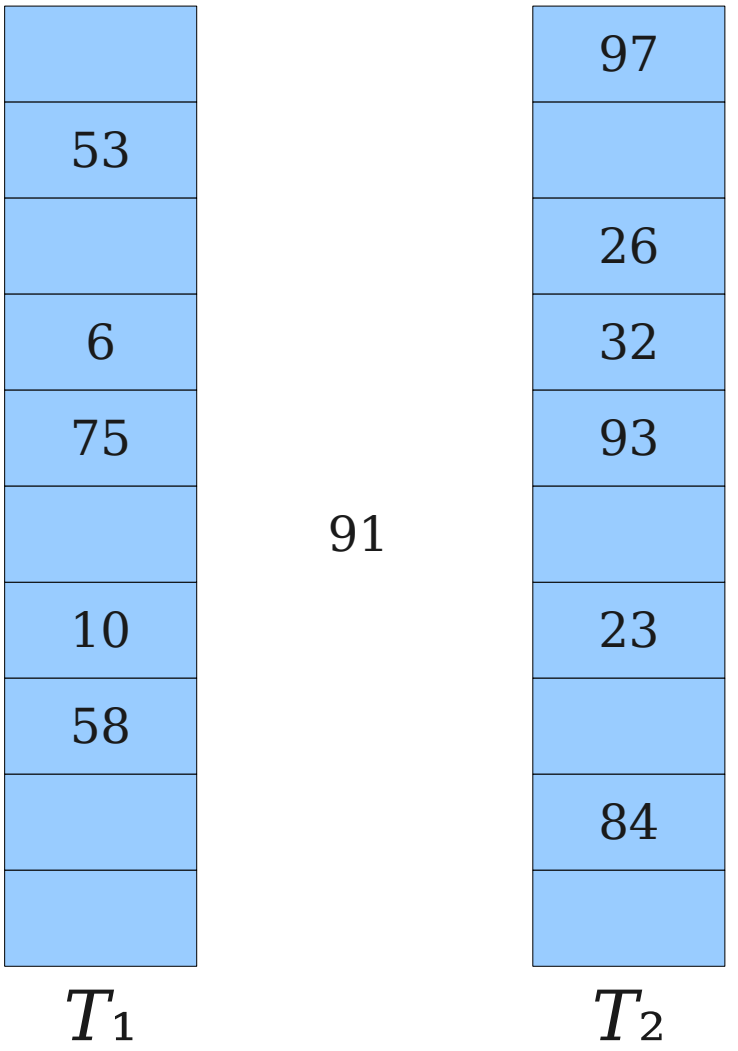
Cuckoo Hashing

- Insertions run into trouble if we run into a cycle.



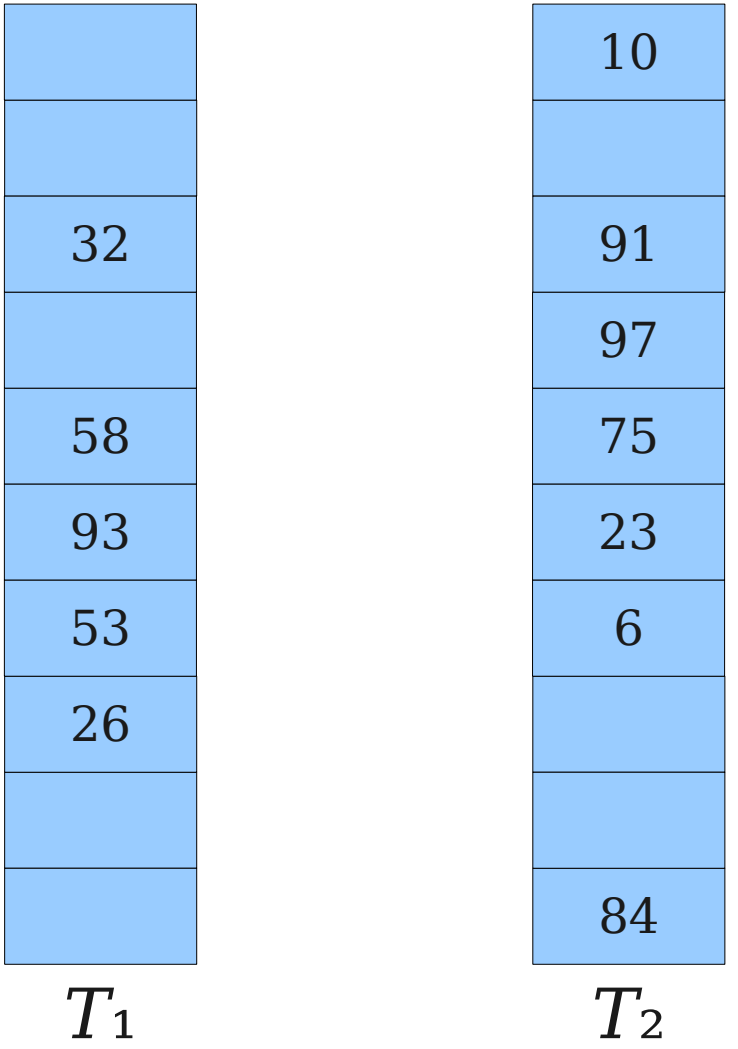
Cuckoo Hashing

- 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.



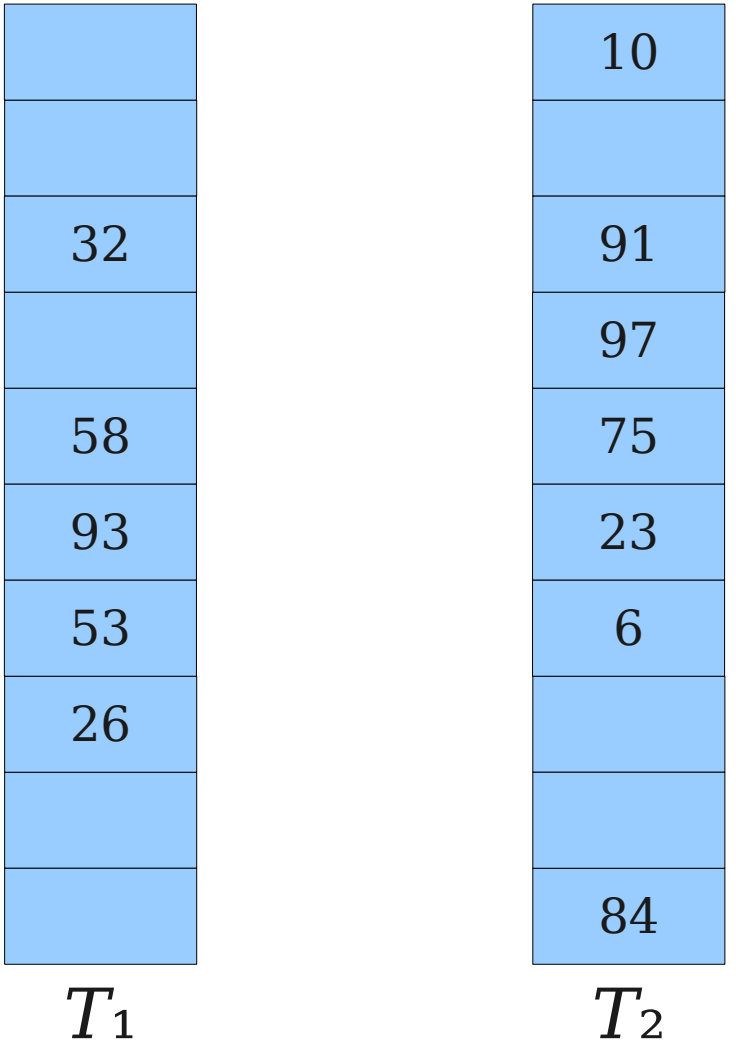
Cuckoo Hashing

- 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.



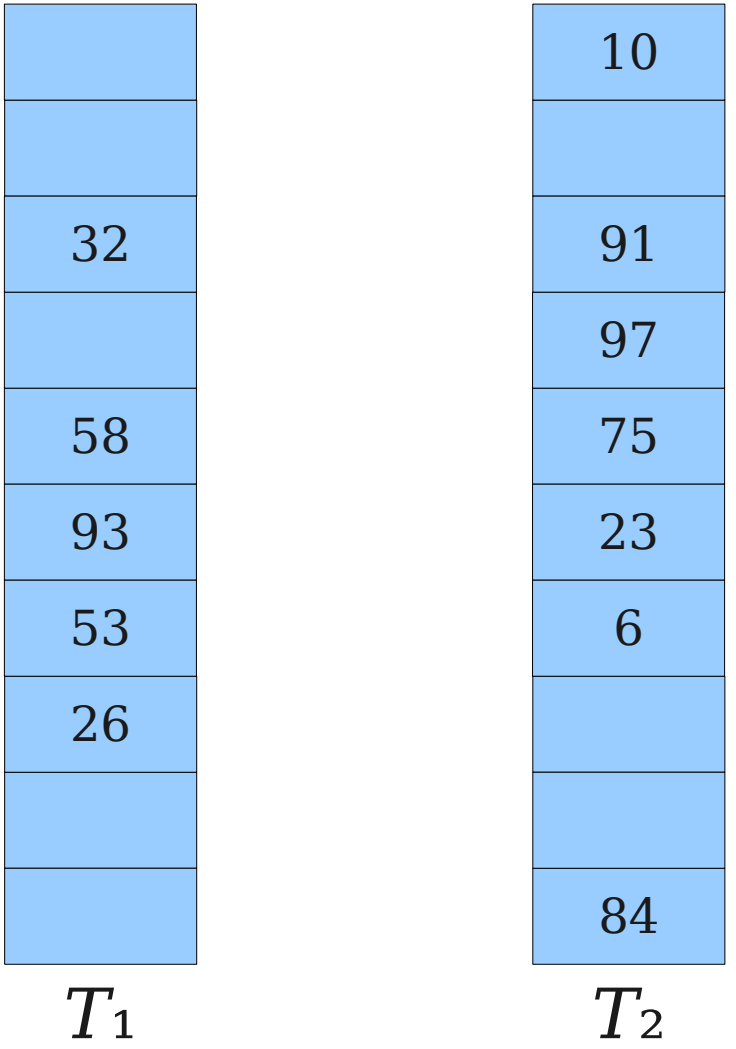
Cuckoo Hashing

- 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.



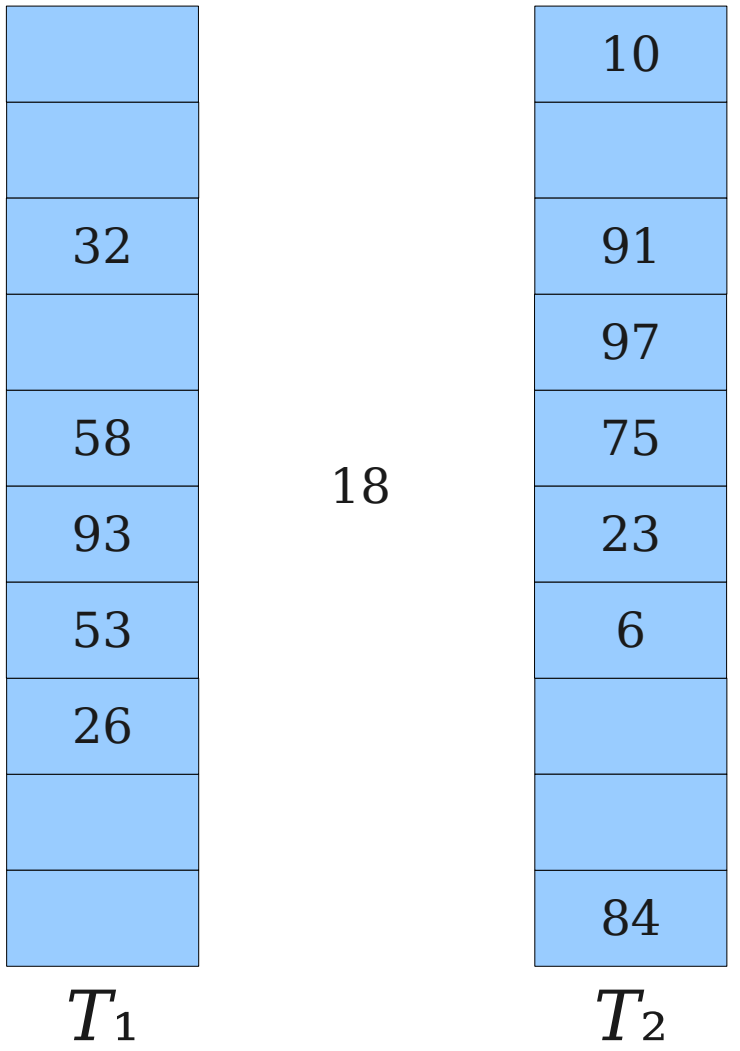
A Note on Cycles

- 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.



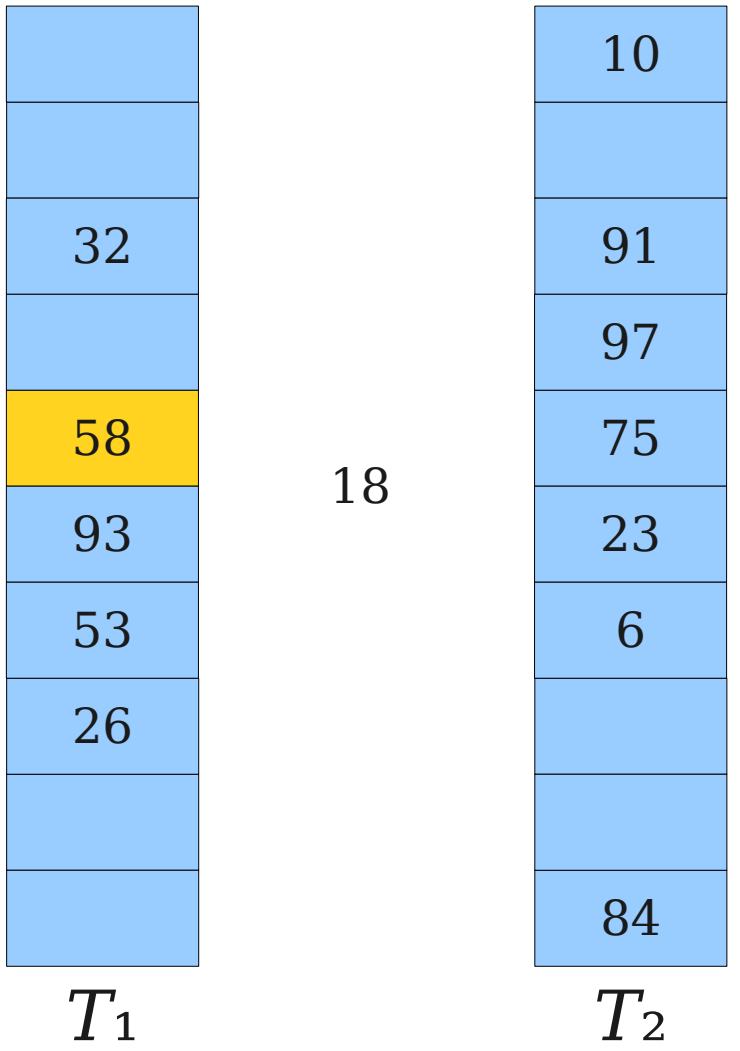
A Note on Cycles

- 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.



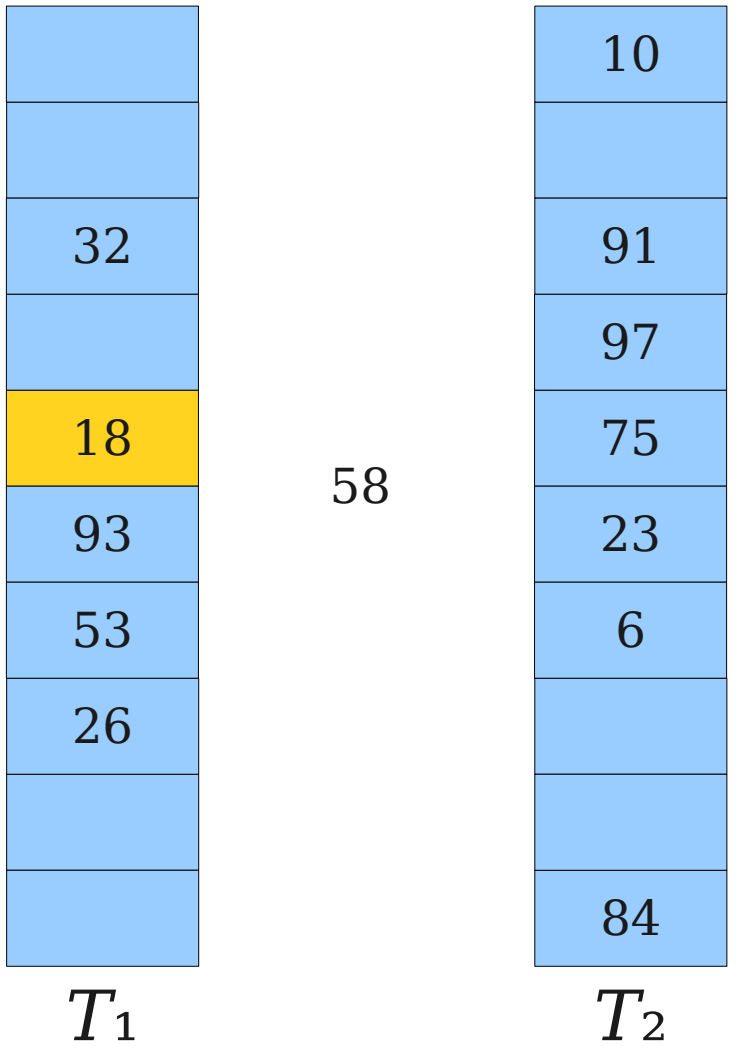
A Note on Cycles

- 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.



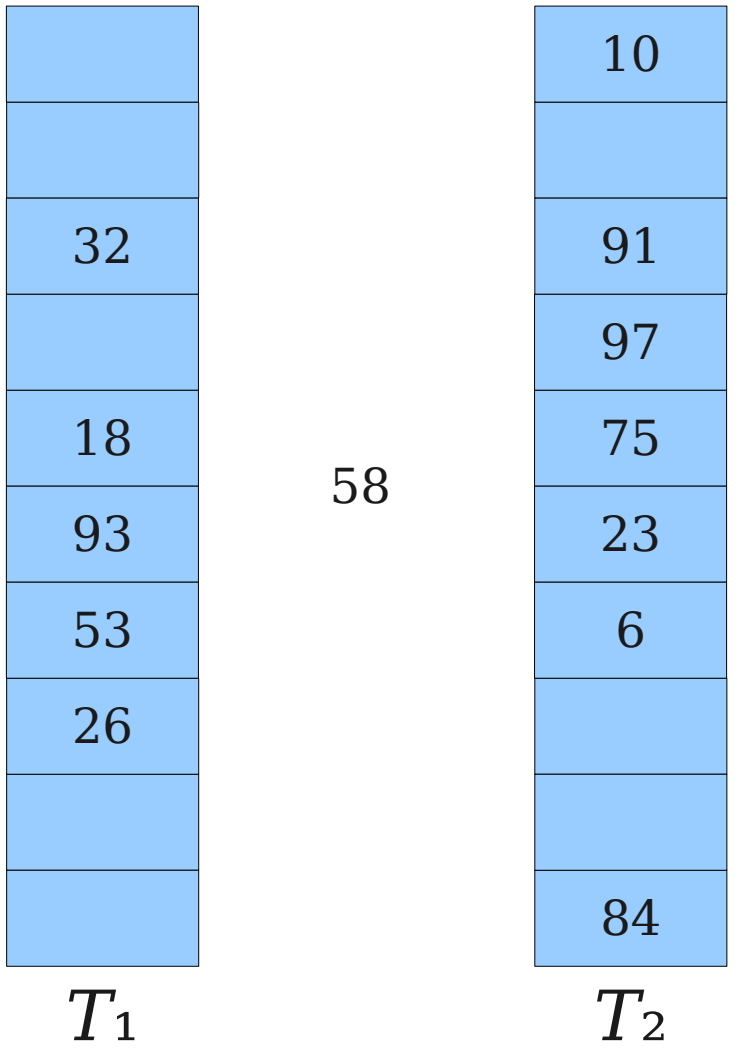
A Note on Cycles

- 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.



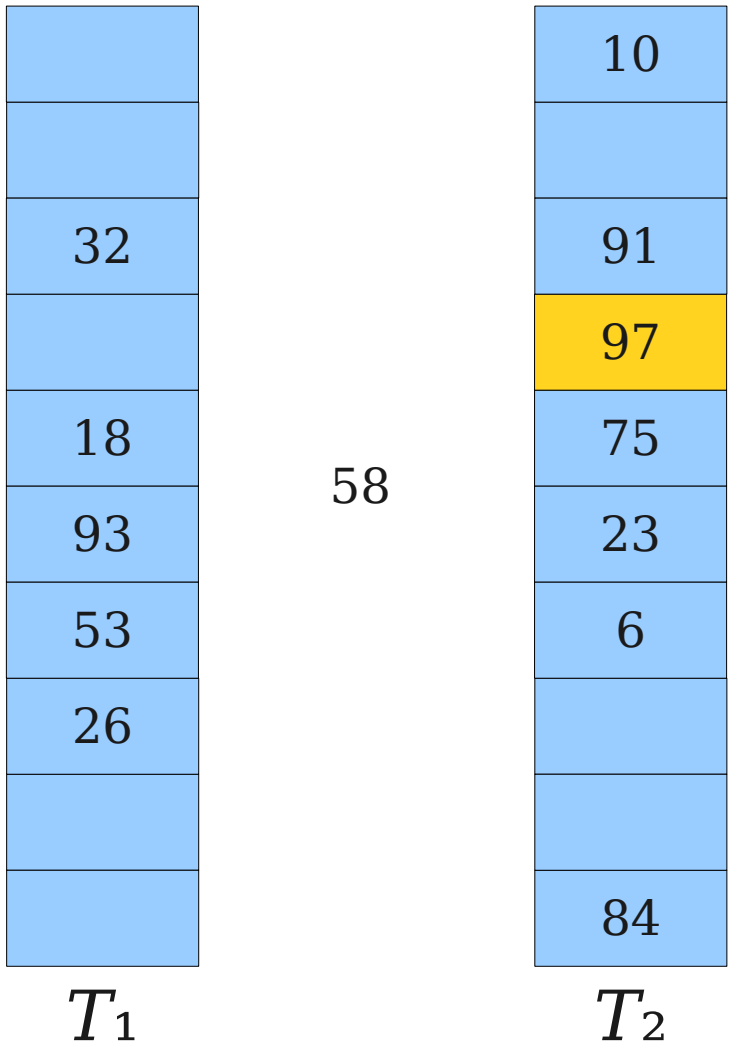
A Note on Cycles

- 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.



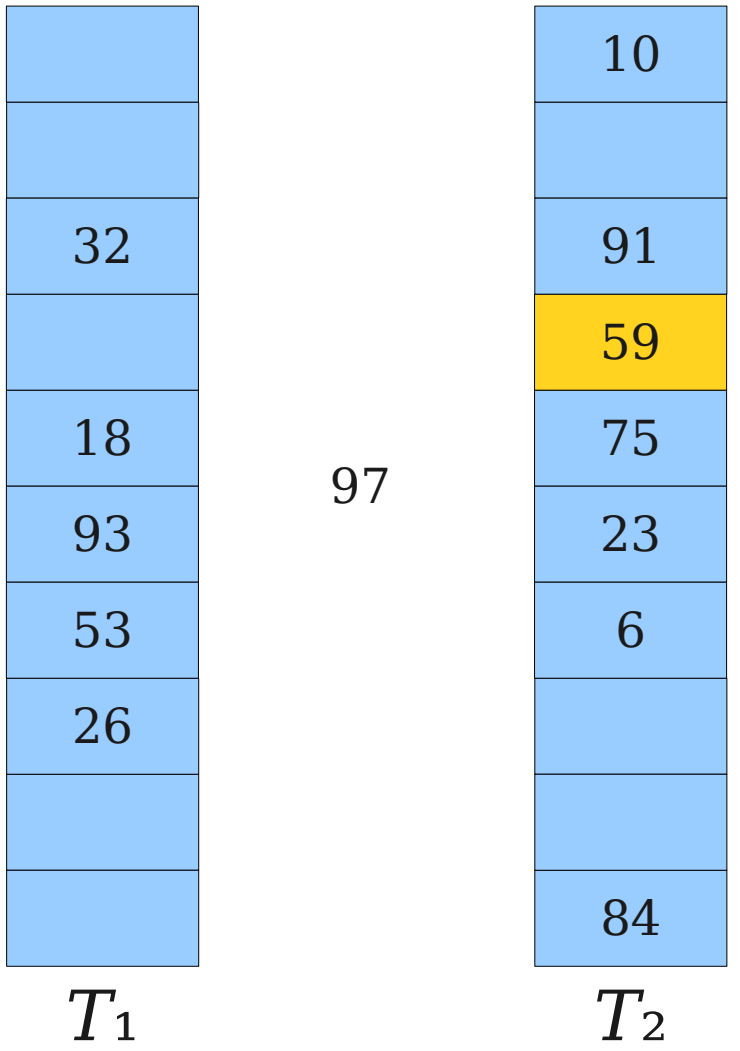
A Note on Cycles

- 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.



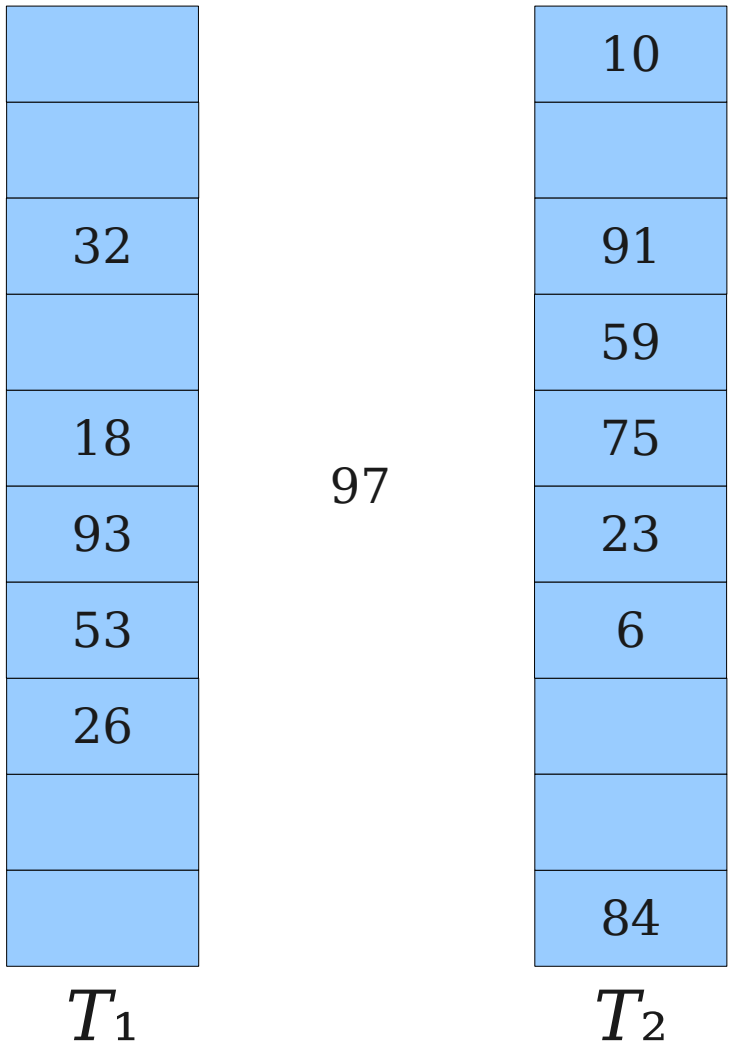
A Note on Cycles

- 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.



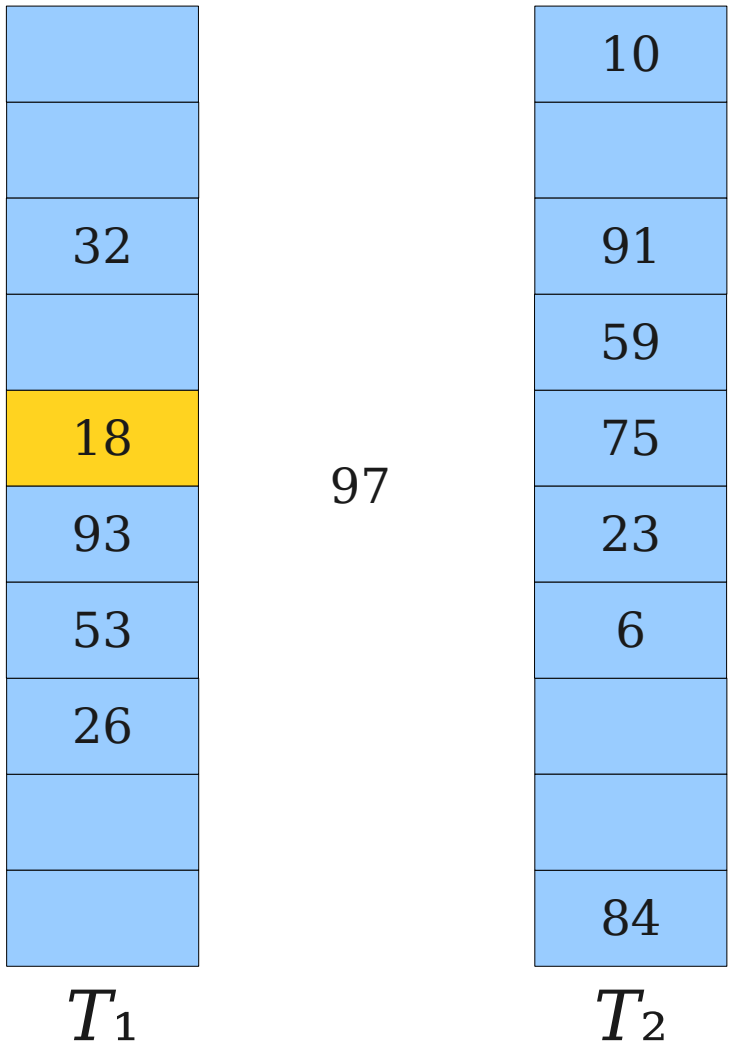
A Note on Cycles

- 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.



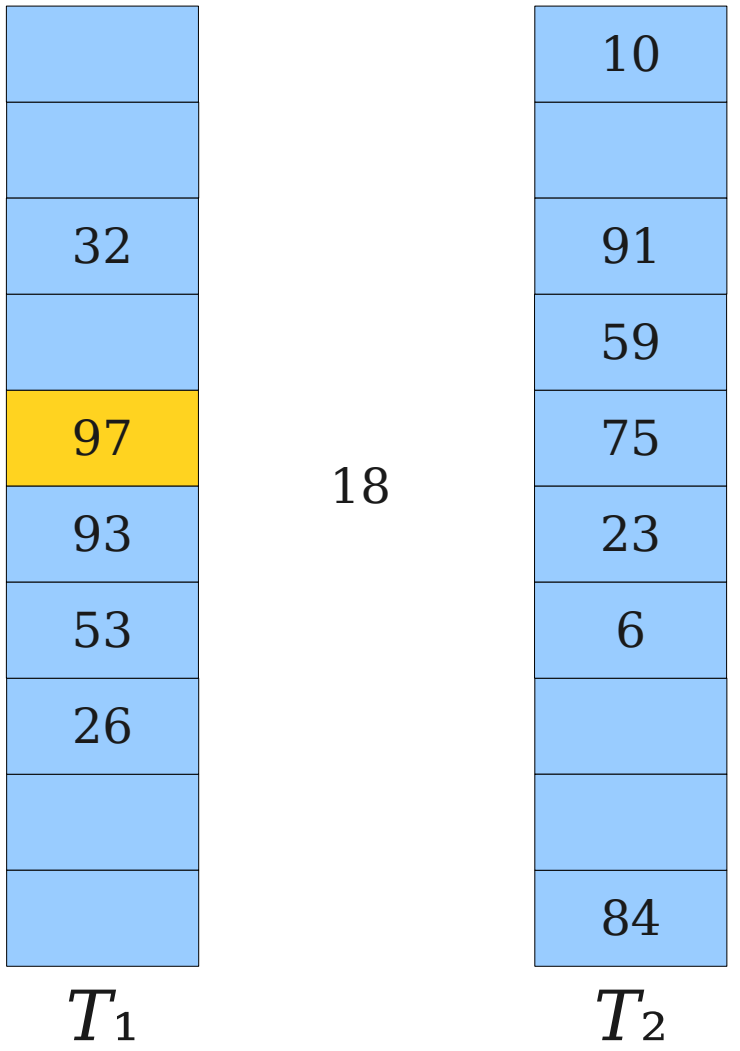
A Note on Cycles

- 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.



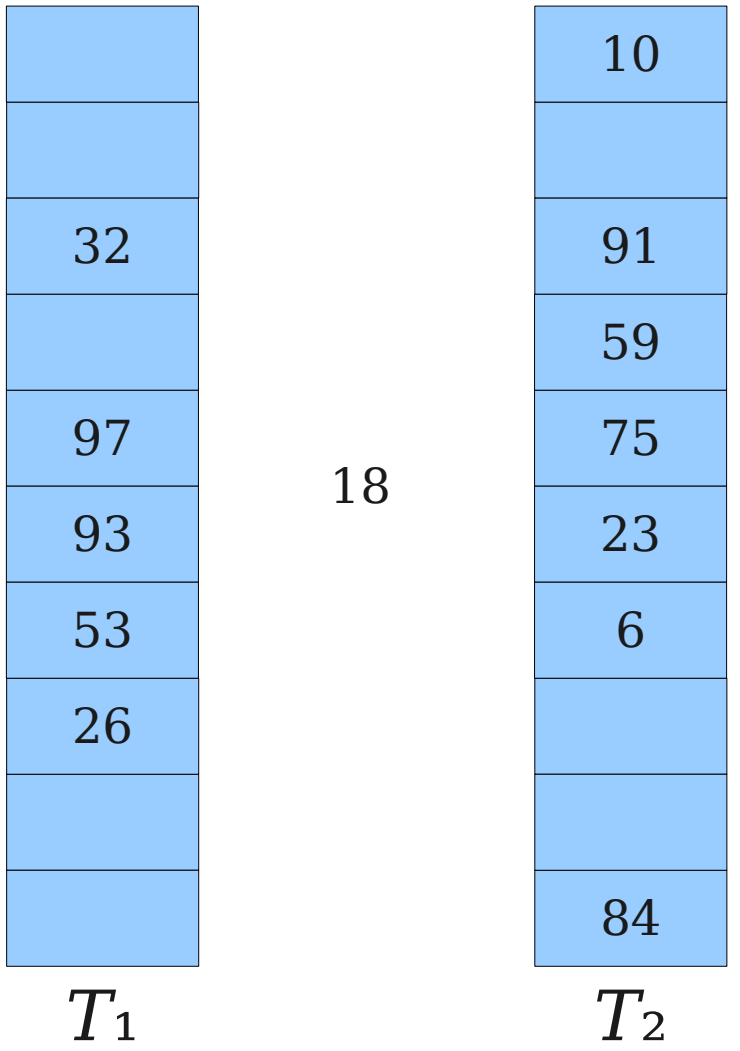
A Note on Cycles

- 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.



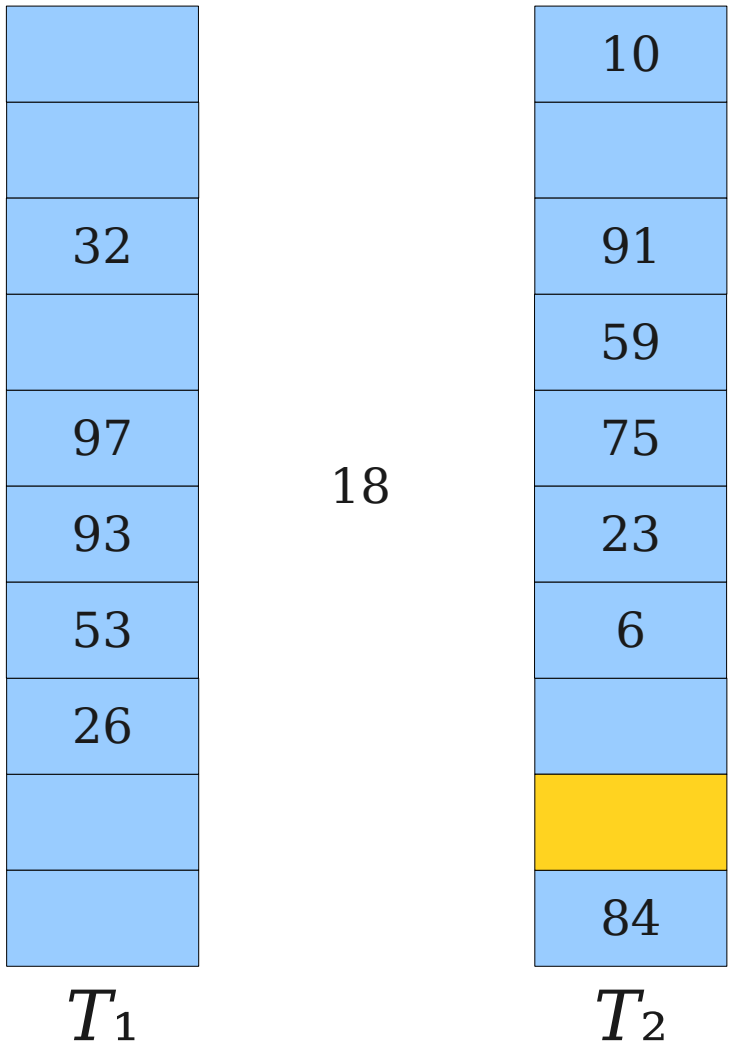
A Note on Cycles

- 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.



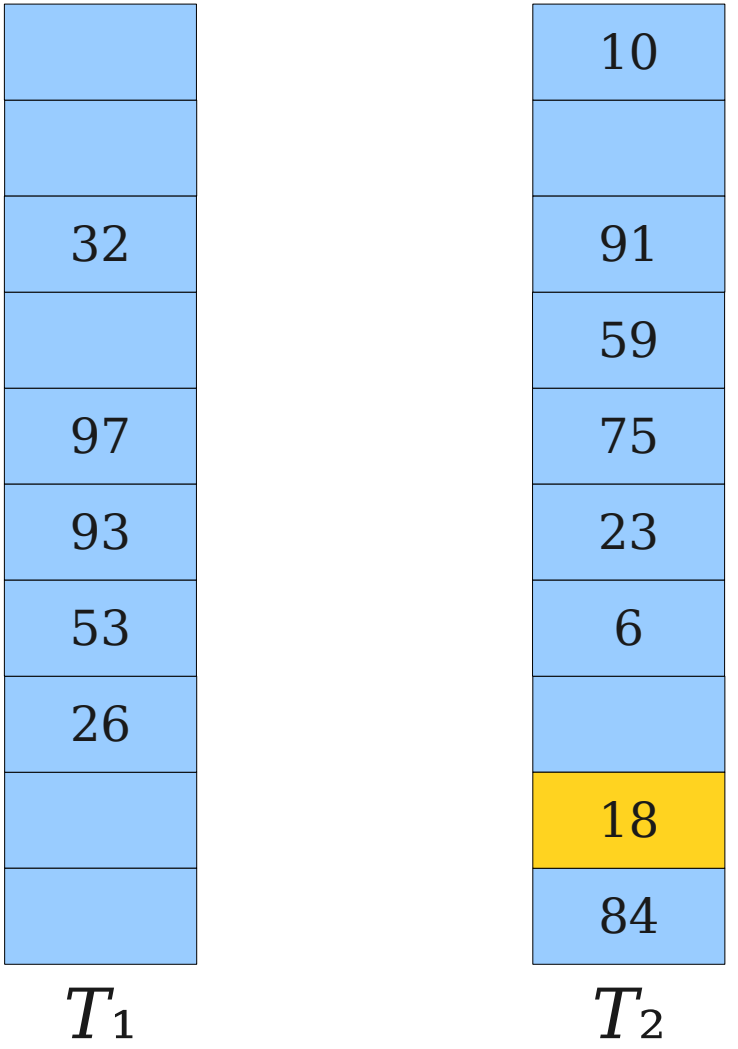
A Note on Cycles

- 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.



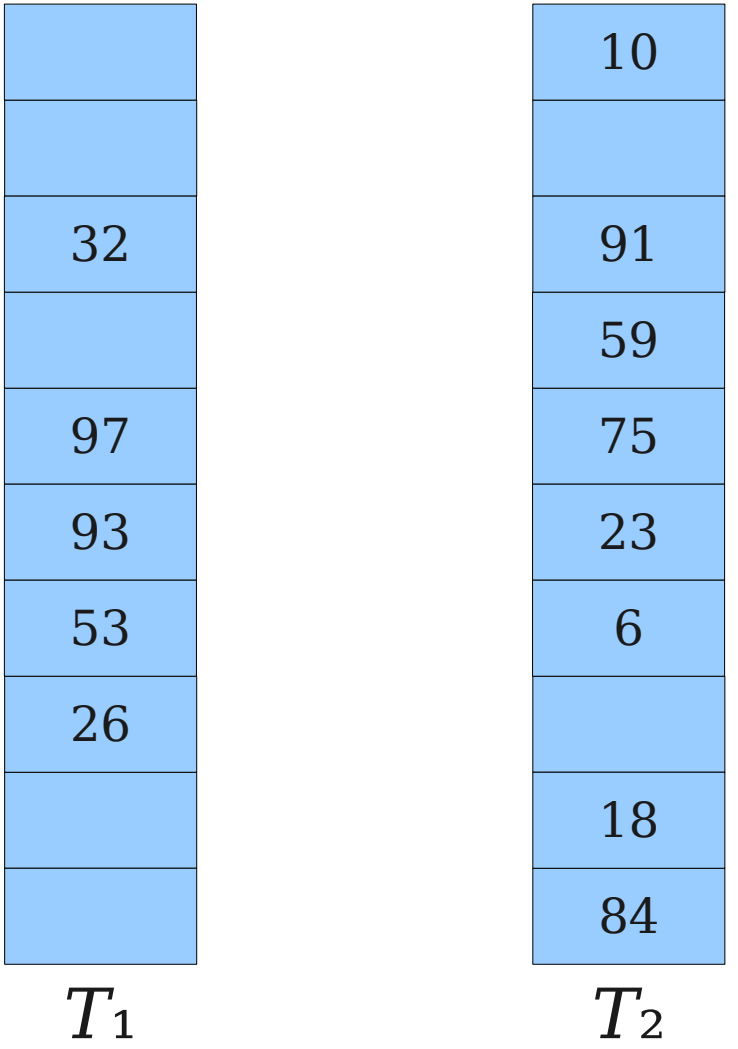
A Note on Cycles

- 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.



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.

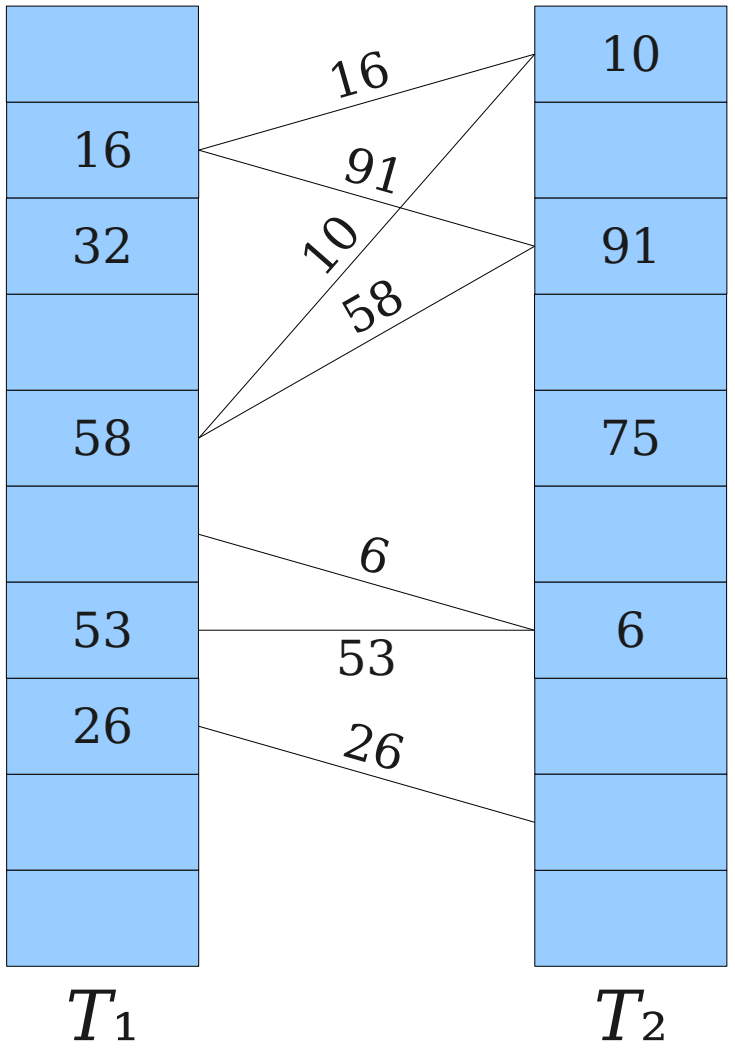


Analyzing Cuckoo Hashing

- 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.

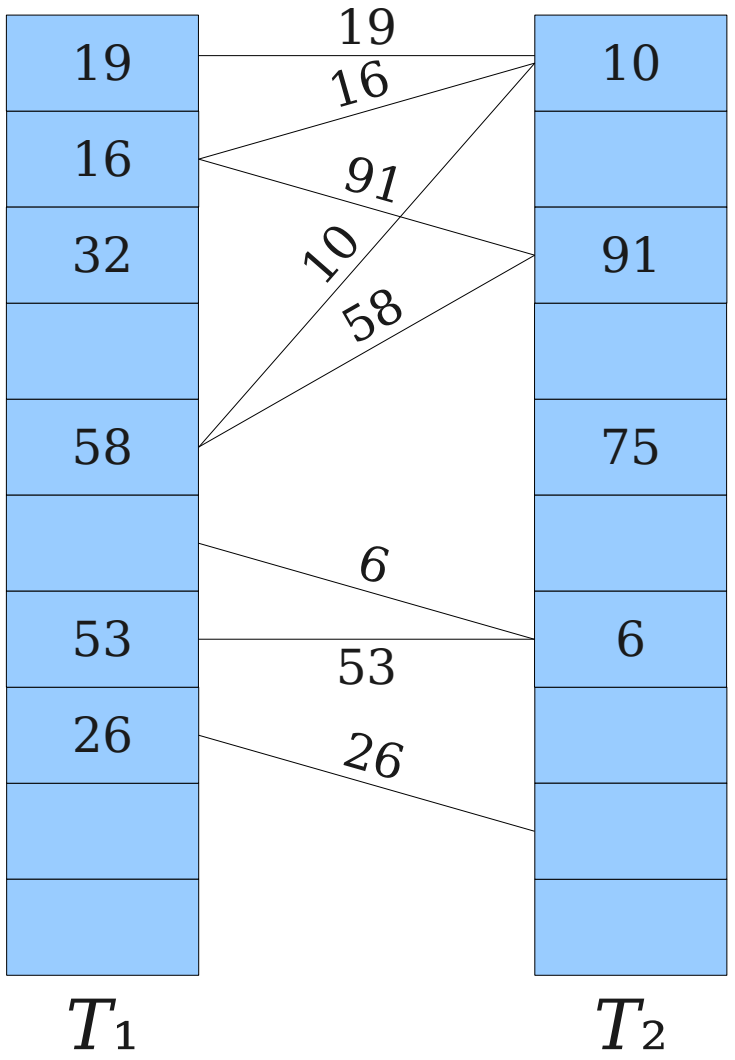
The Cuckoo Graph

- 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.



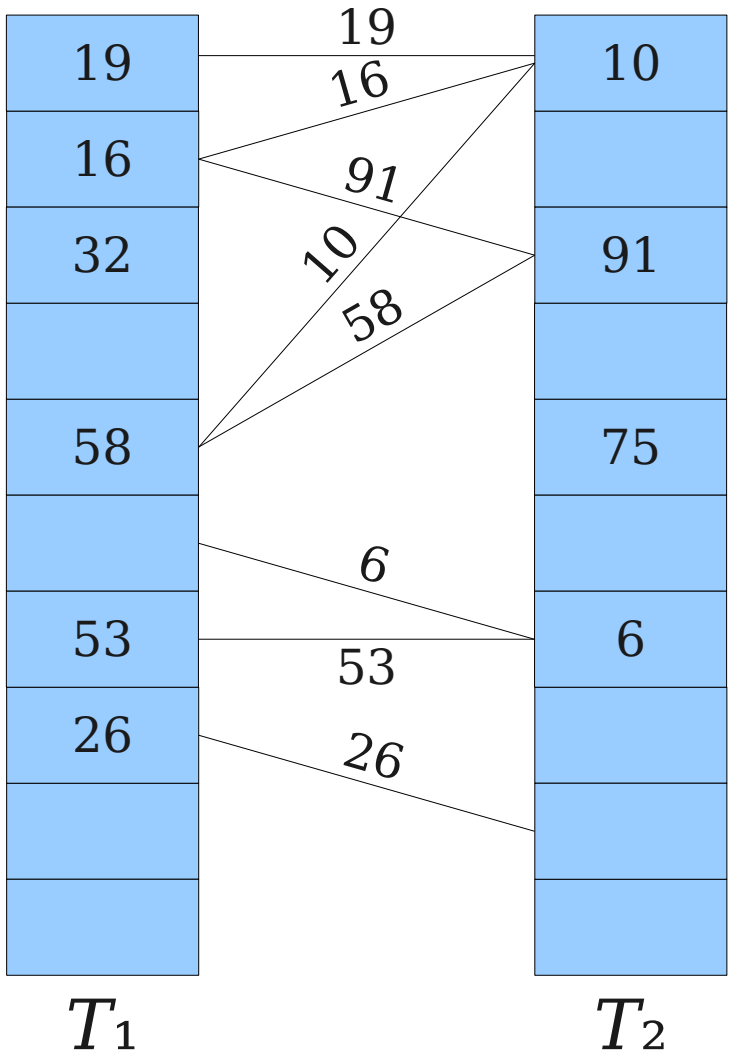
The Cuckoo Graph

- 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.



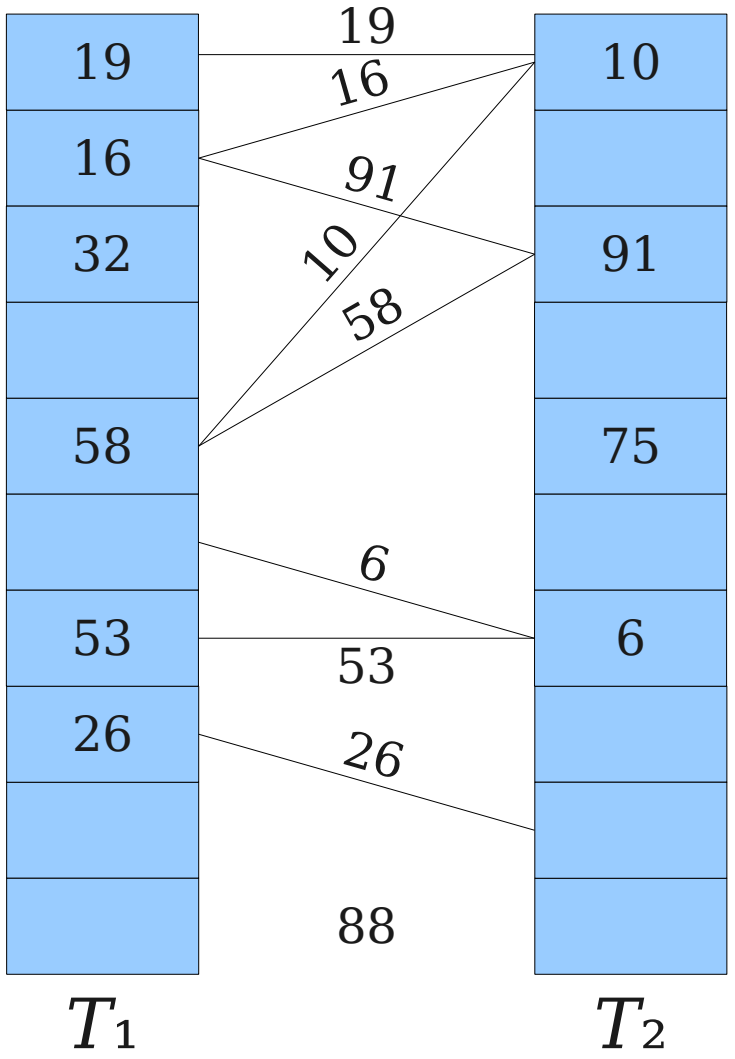
The Cuckoo Graph

- An insertion in a cuckoo hash table traces a path through the cuckoo graph.
- An insertion succeeds iff the connected component containing the inserted value contains at most one cycle.



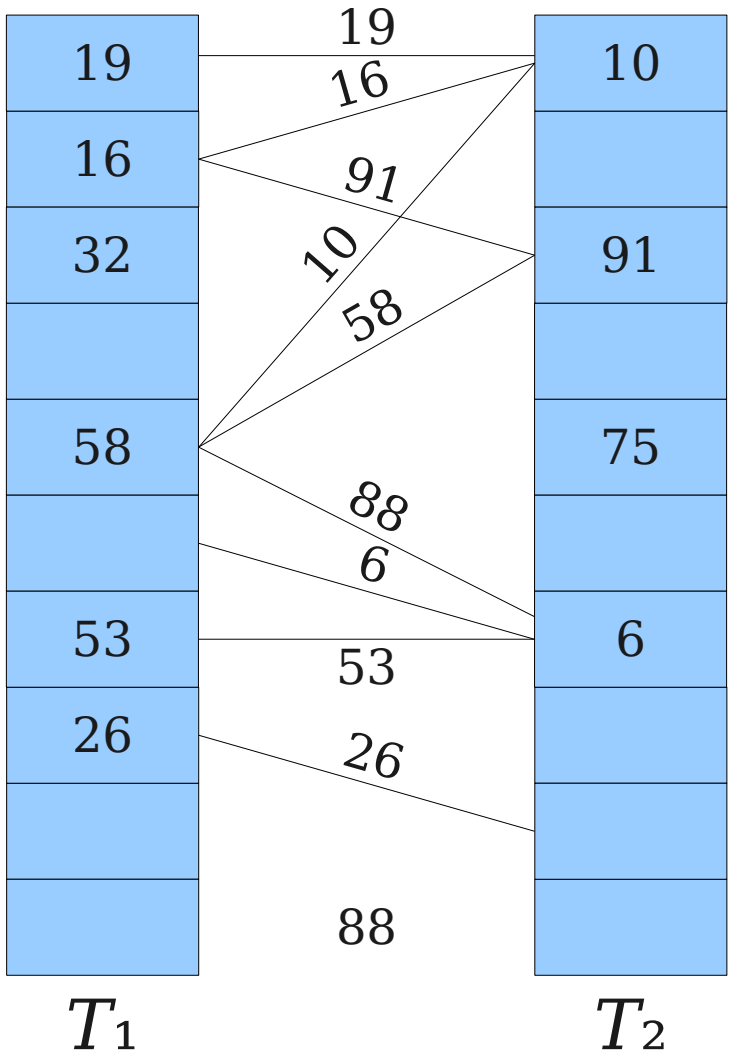
The Cuckoo Graph

- An insertion in a cuckoo hash table traces a path through the cuckoo graph.
- An insertion succeeds iff the connected component containing the inserted value contains at most one cycle.



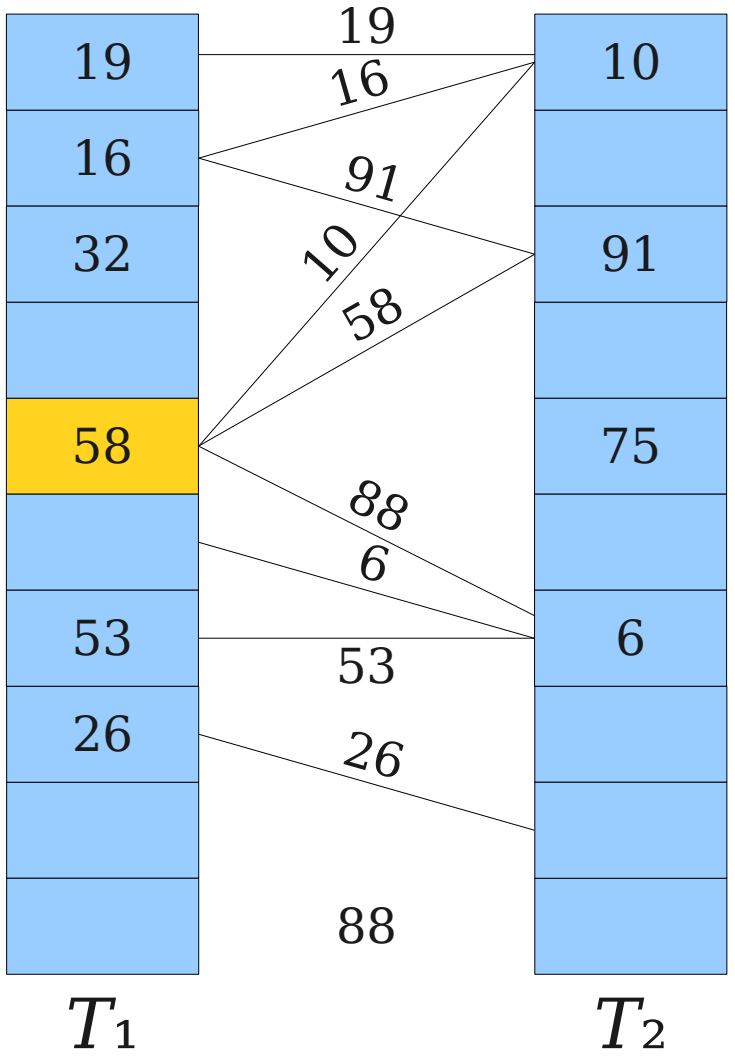
The Cuckoo Graph

- An insertion in a cuckoo hash table traces a path through the cuckoo graph.
- An insertion succeeds iff the connected component containing the inserted value contains at most one cycle.



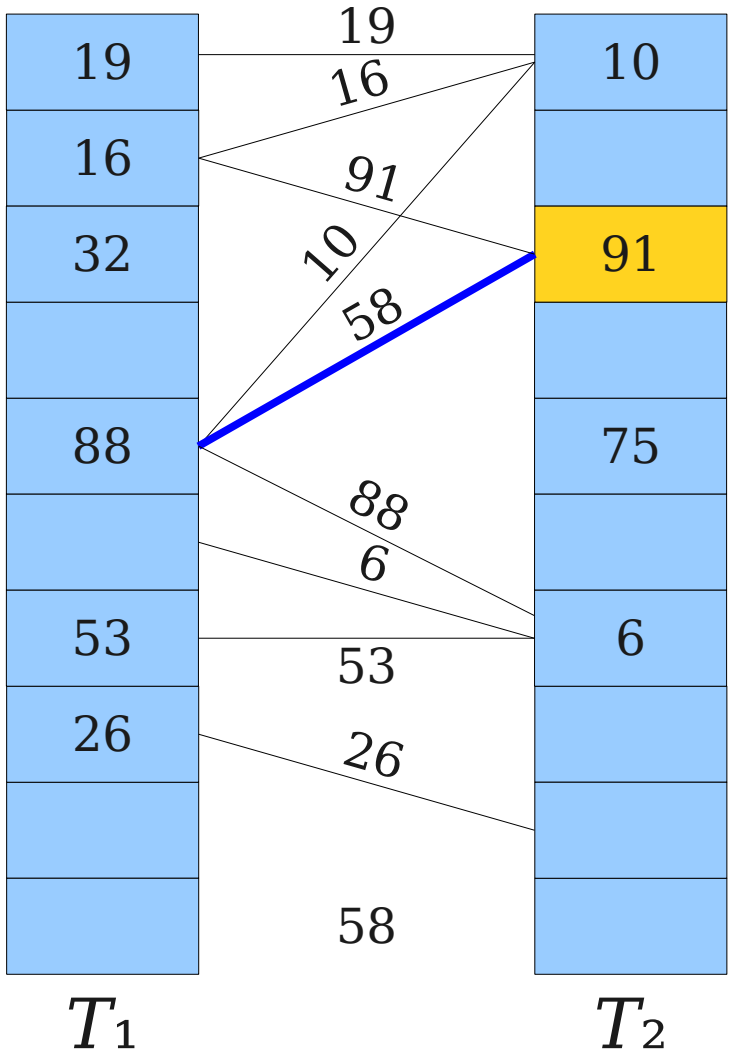
The Cuckoo Graph

- An insertion in a cuckoo hash table traces a path through the cuckoo graph.
- An insertion succeeds iff the connected component containing the inserted value contains at most one cycle.



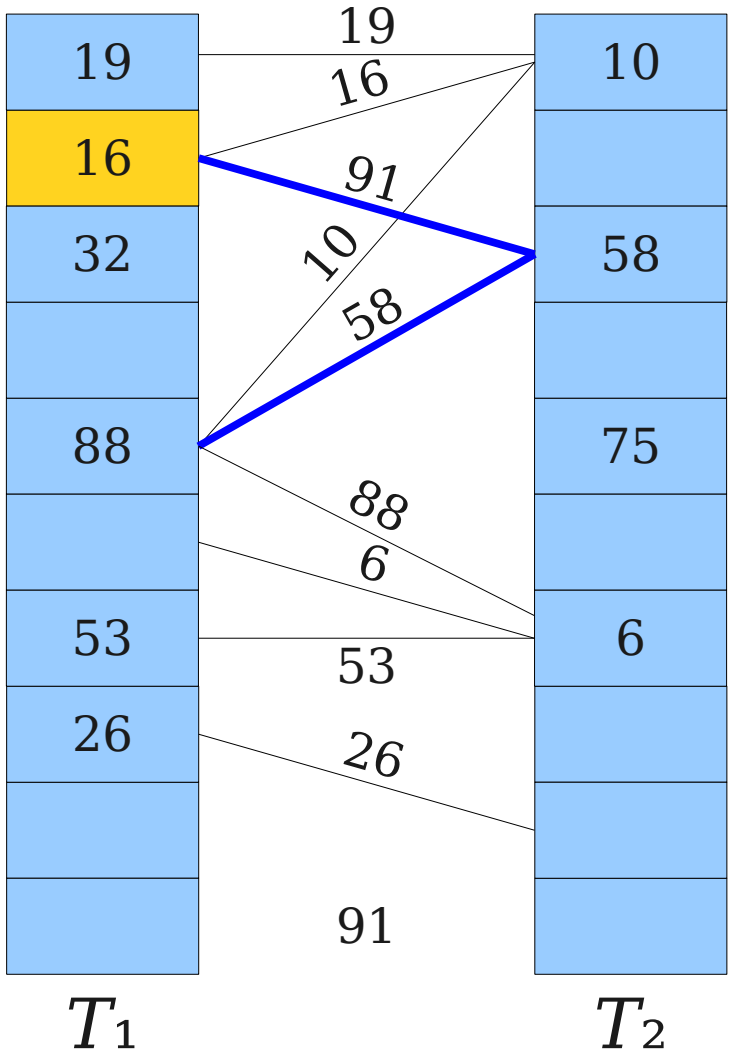
The Cuckoo Graph

- An insertion in a cuckoo hash table traces a path through the cuckoo graph.
- An insertion succeeds iff the connected component containing the inserted value contains at most one cycle.



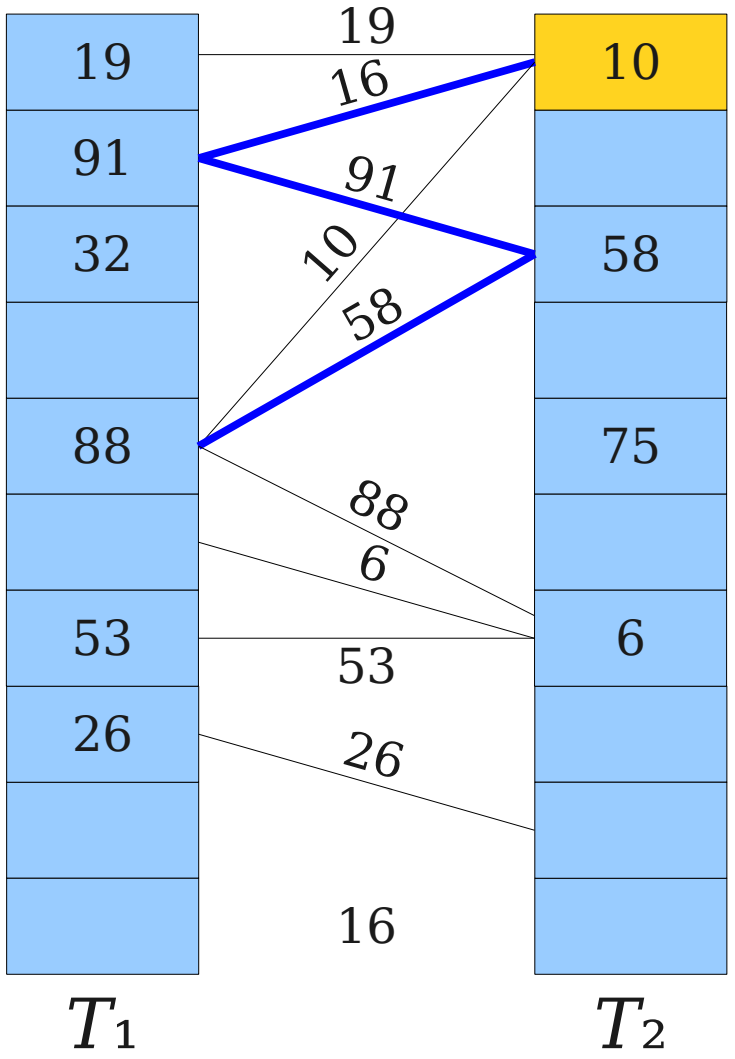
The Cuckoo Graph

- An insertion in a cuckoo hash table traces a path through the cuckoo graph.
- An insertion succeeds iff the connected component containing the inserted value contains at most one cycle.



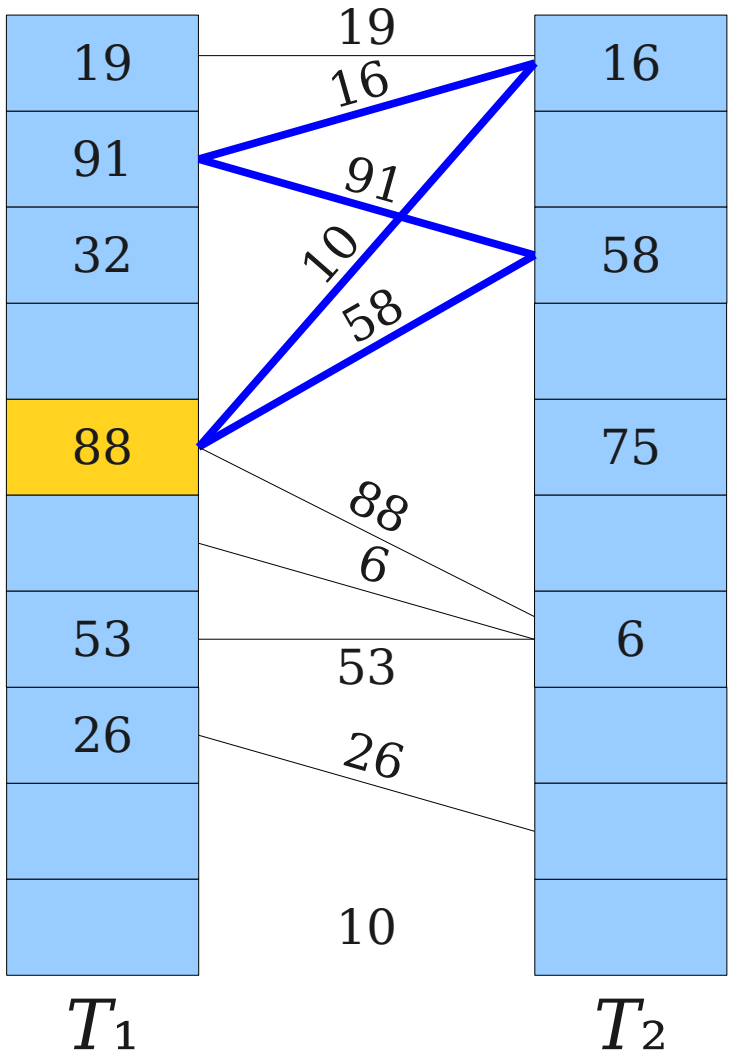
The Cuckoo Graph

- An insertion in a cuckoo hash table traces a path through the cuckoo graph.
- An insertion succeeds iff the connected component containing the inserted value contains at most one cycle.



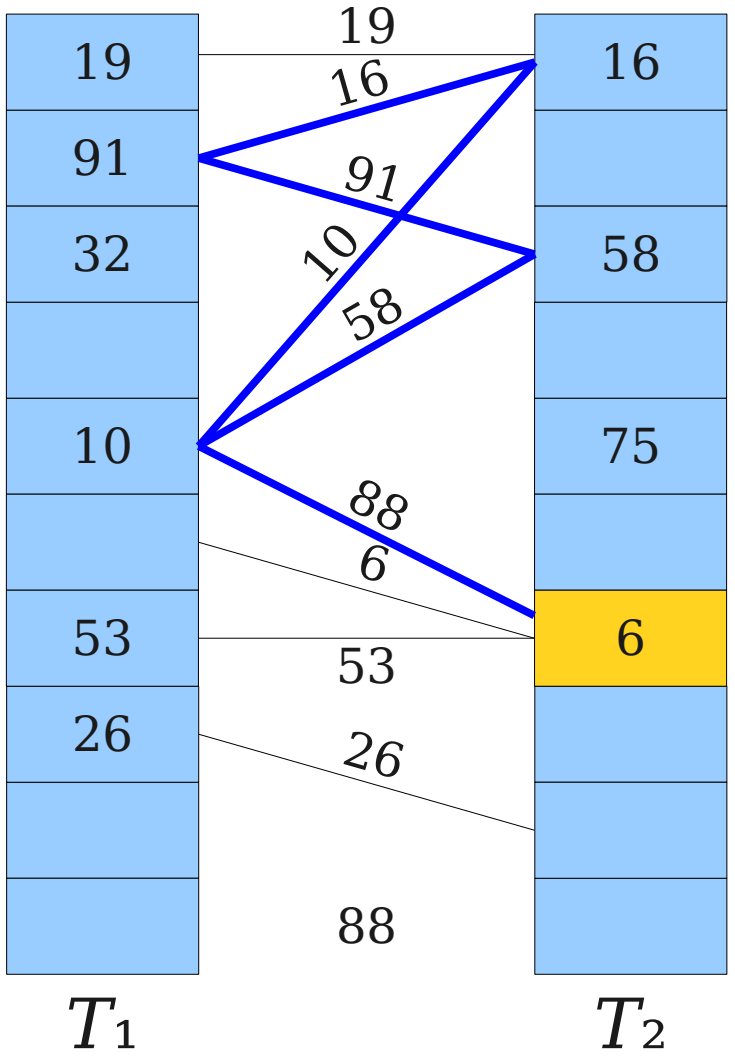
The Cuckoo Graph

- An insertion in a cuckoo hash table traces a path through the cuckoo graph.
- An insertion succeeds iff the connected component containing the inserted value contains at most one cycle.



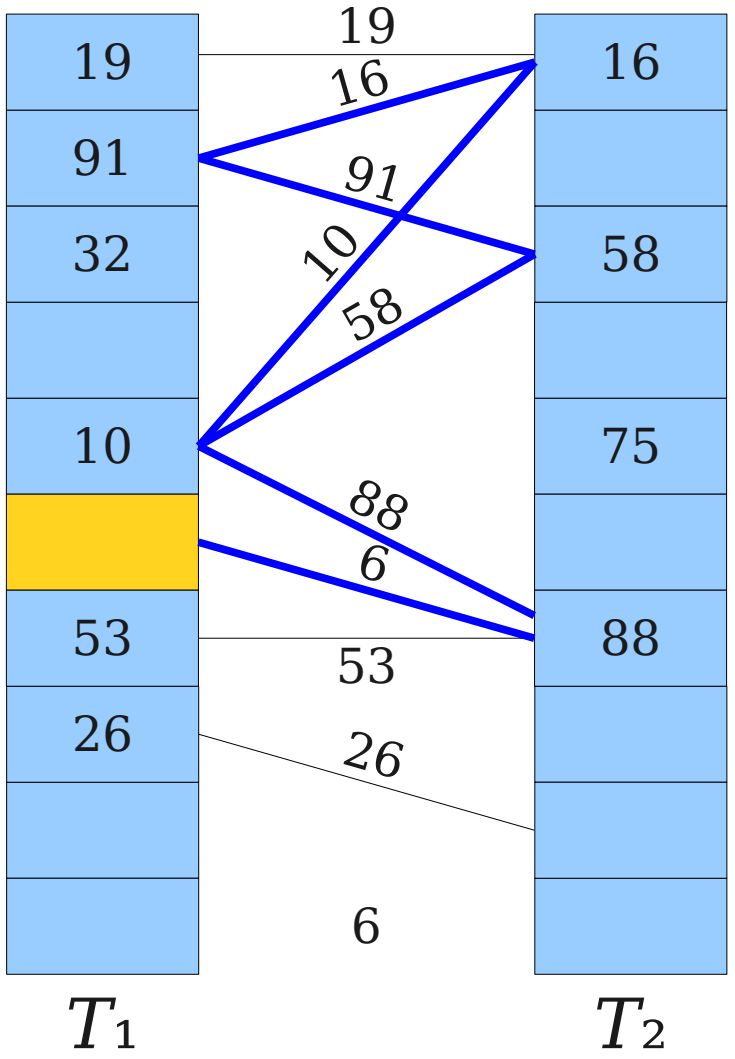
The Cuckoo Graph

- An insertion in a cuckoo hash table traces a path through the cuckoo graph.
- An insertion succeeds iff the connected component containing the inserted value contains at most one cycle.



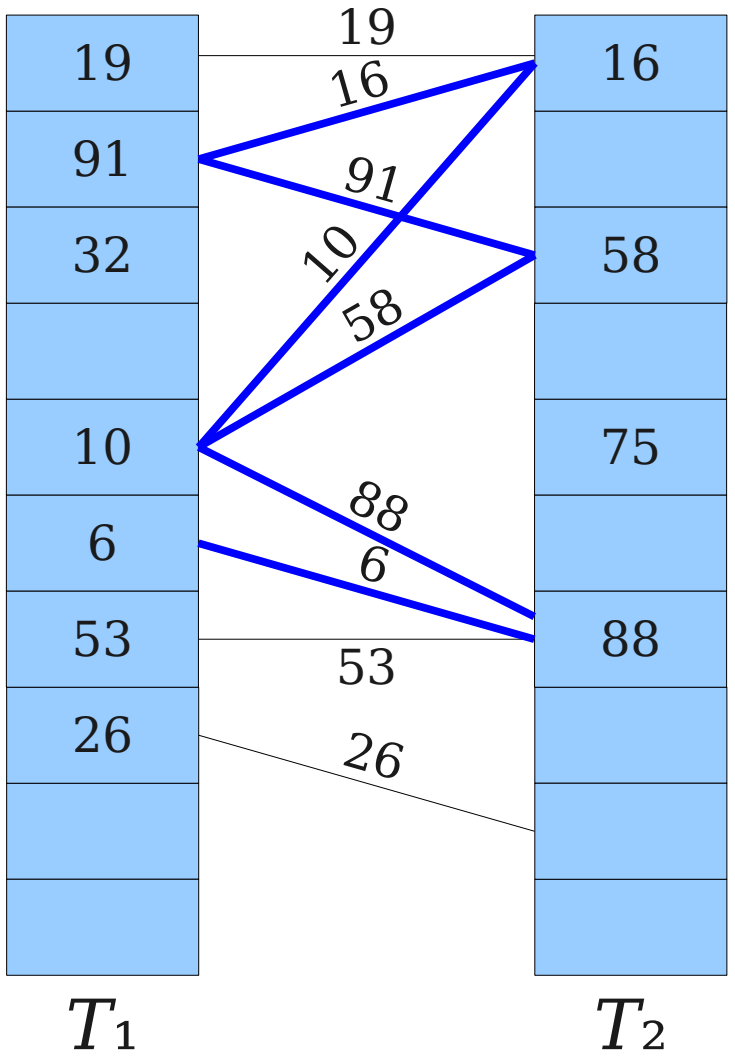
The Cuckoo Graph

- An insertion in a cuckoo hash table traces a path through the cuckoo graph.
- An insertion succeeds iff the connected component containing the inserted value contains at most one cycle.



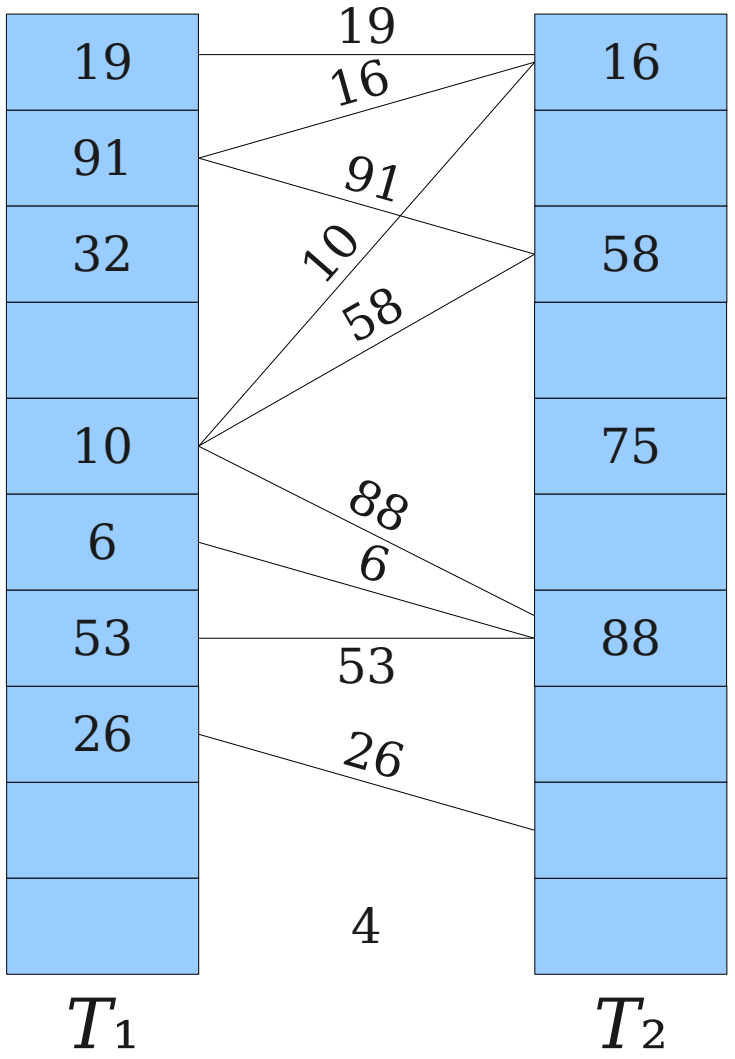
The Cuckoo Graph

- An insertion in a cuckoo hash table traces a path through the cuckoo graph.
- An insertion succeeds iff the connected component containing the inserted value contains at most one cycle.



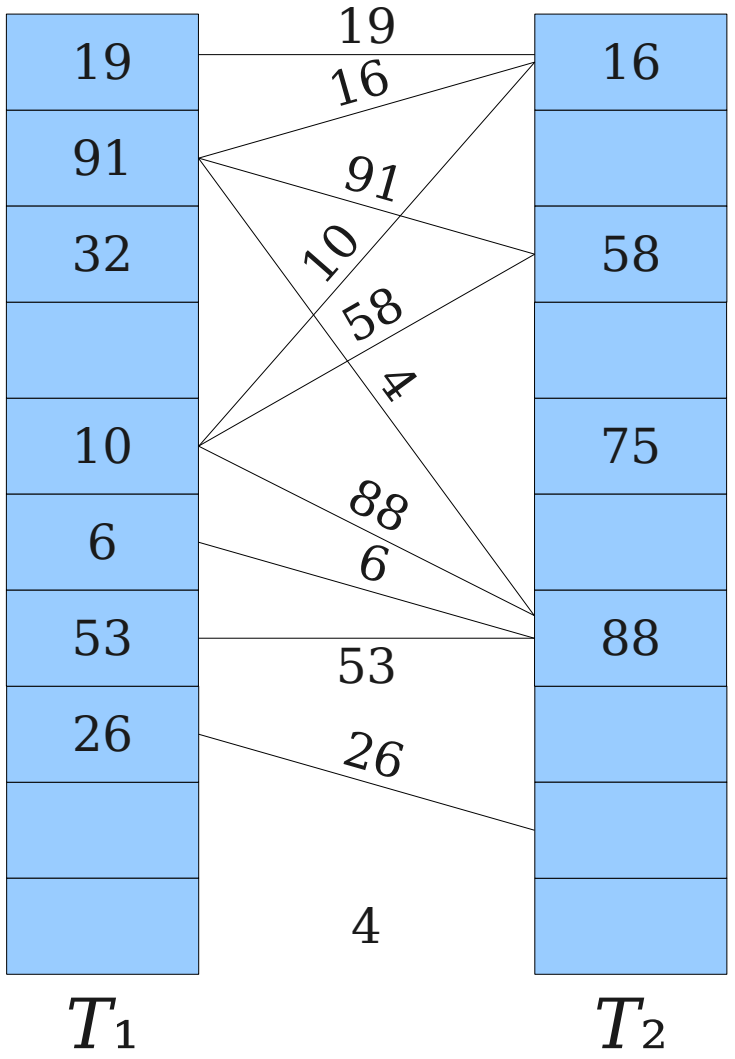
The Cuckoo Graph

- An insertion in a cuckoo hash table traces a path through the cuckoo graph.
- An insertion succeeds iff the connected component containing the inserted value contains at most one cycle.



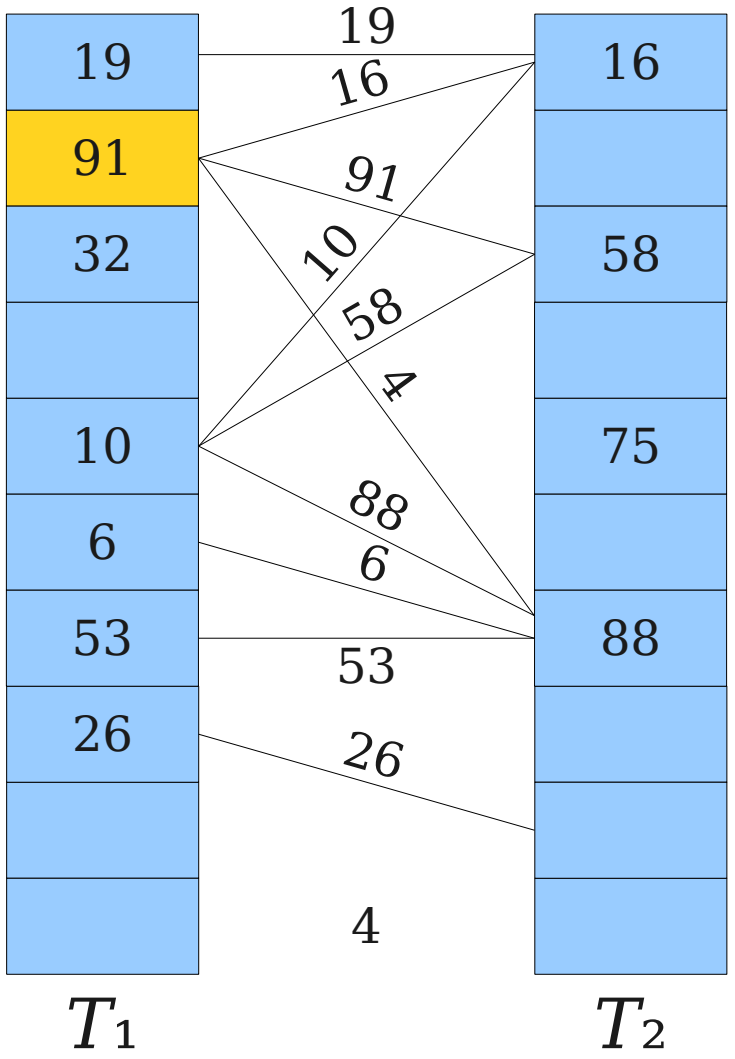
The Cuckoo Graph

- An insertion in a cuckoo hash table traces a path through the cuckoo graph.
- An insertion succeeds iff the connected component containing the inserted value contains at most one cycle.



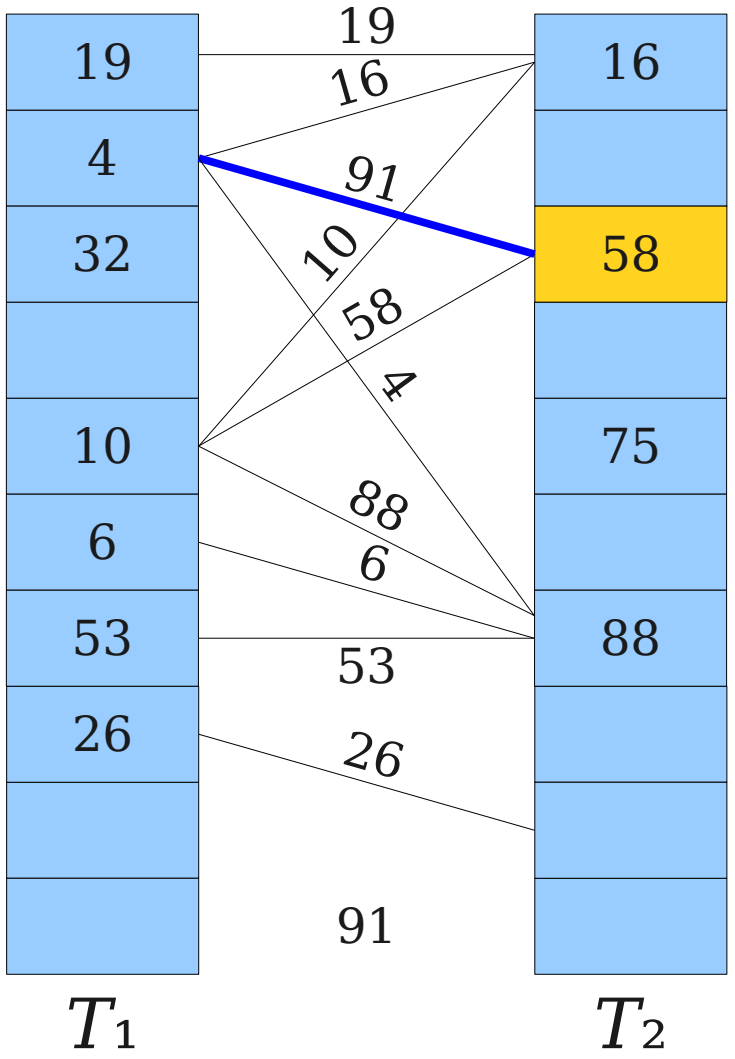
The Cuckoo Graph

- An insertion in a cuckoo hash table traces a path through the cuckoo graph.
- An insertion succeeds iff the connected component containing the inserted value contains at most one cycle.



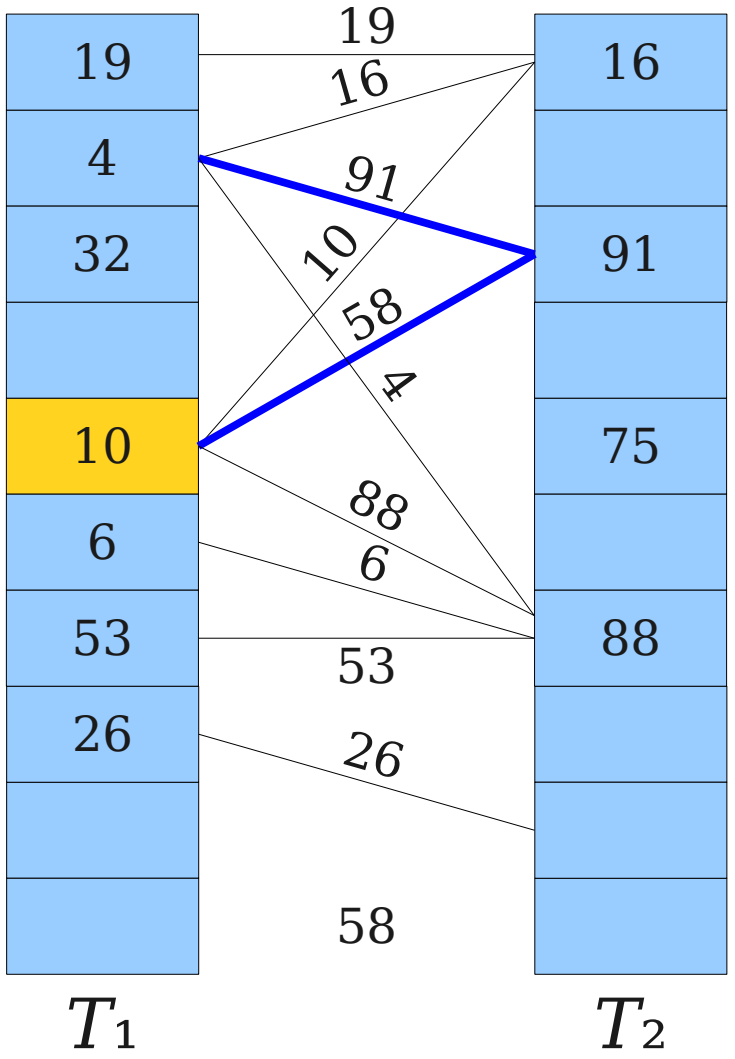
The Cuckoo Graph

- An insertion in a cuckoo hash table traces a path through the cuckoo graph.
- An insertion succeeds iff the connected component containing the inserted value contains at most one cycle.



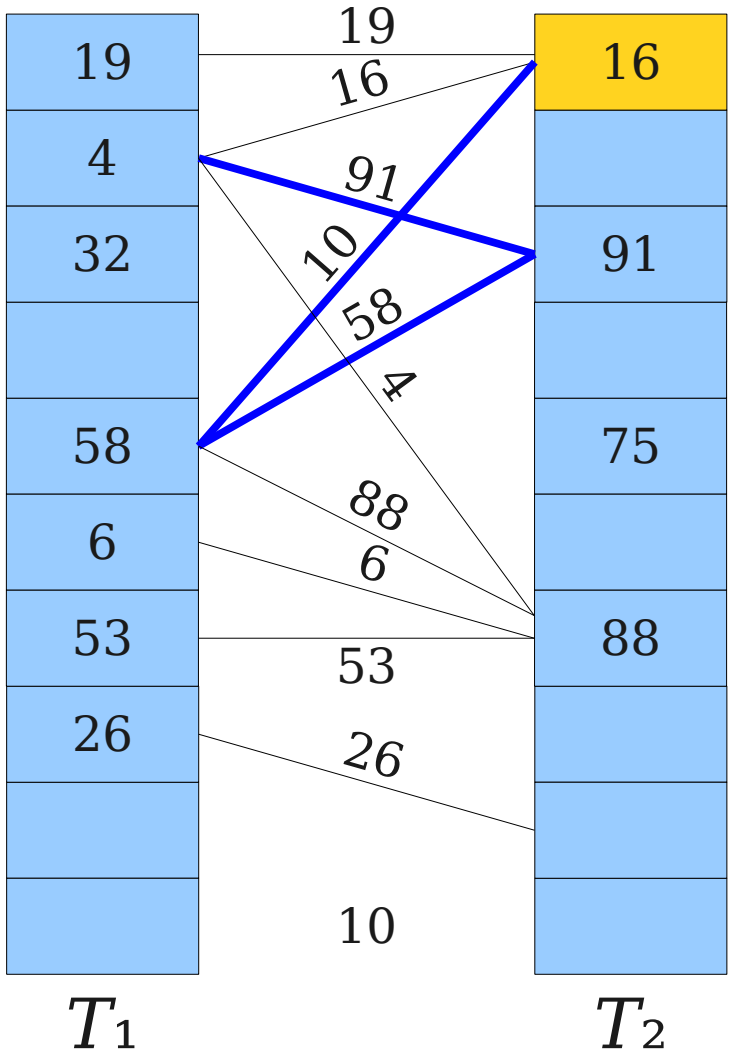
The Cuckoo Graph

- An insertion in a cuckoo hash table traces a path through the cuckoo graph.
- An insertion succeeds iff the connected component containing the inserted value contains at most one cycle.



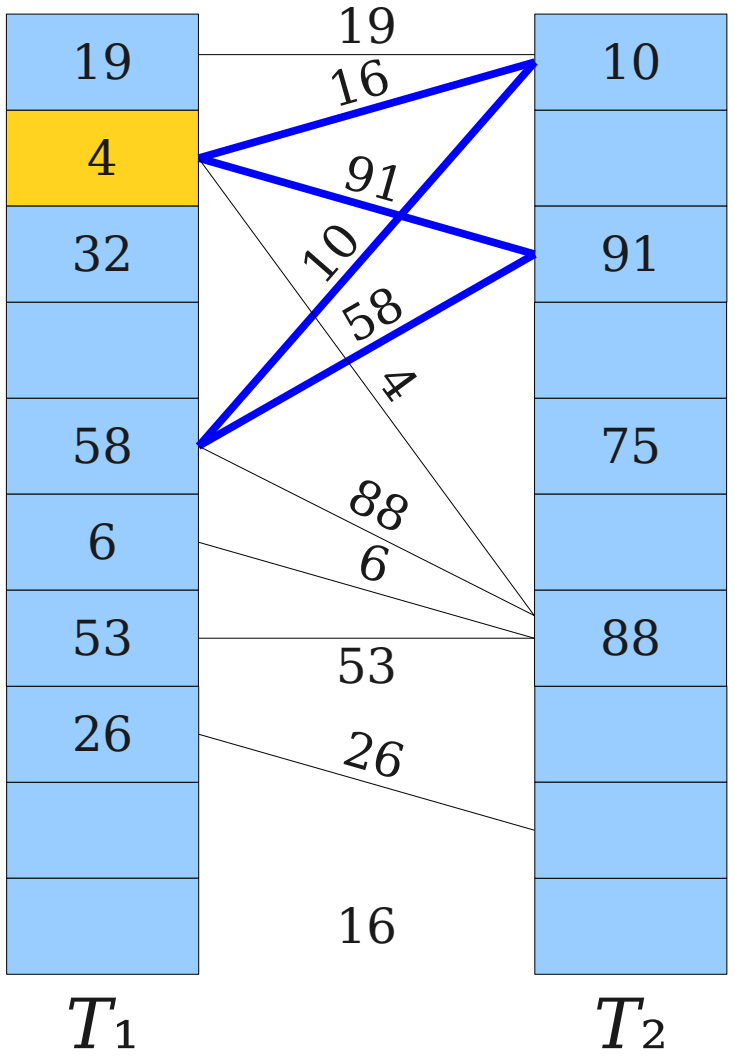
The Cuckoo Graph

- An insertion in a cuckoo hash table traces a path through the cuckoo graph.
- An insertion succeeds iff the connected component containing the inserted value contains at most one cycle.



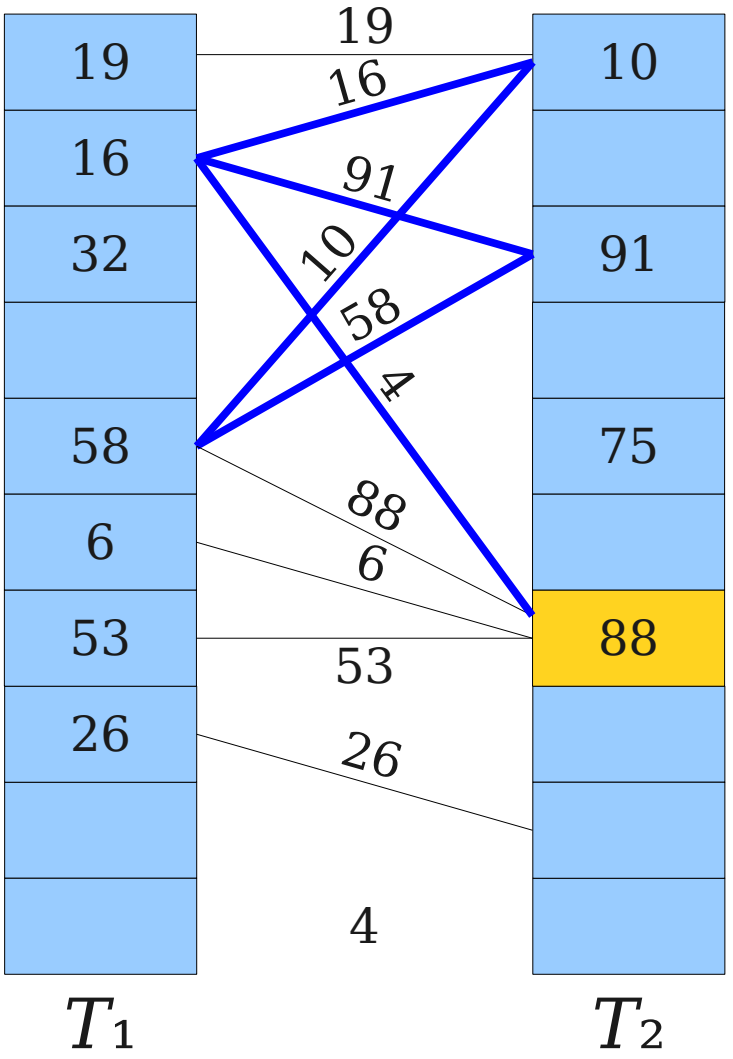
The Cuckoo Graph

- An insertion in a cuckoo hash table traces a path through the cuckoo graph.
- An insertion succeeds iff the connected component containing the inserted value contains at most one cycle.



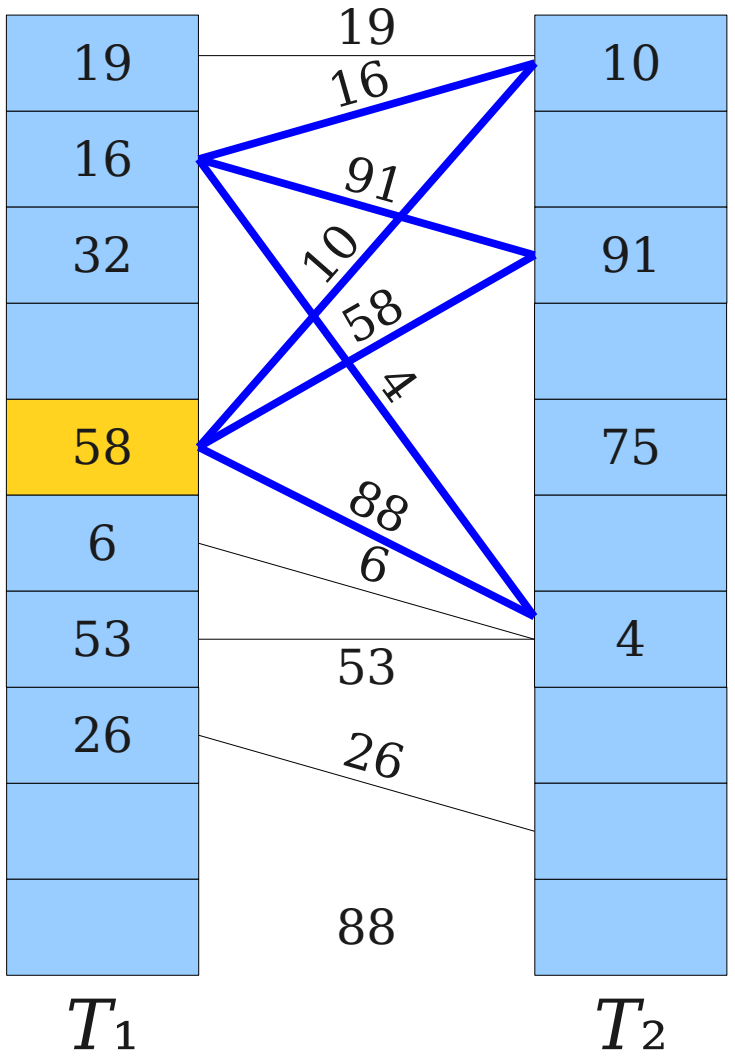
The Cuckoo Graph

- An insertion in a cuckoo hash table traces a path through the cuckoo graph.
- An insertion succeeds iff the connected component containing the inserted value contains at most one cycle.



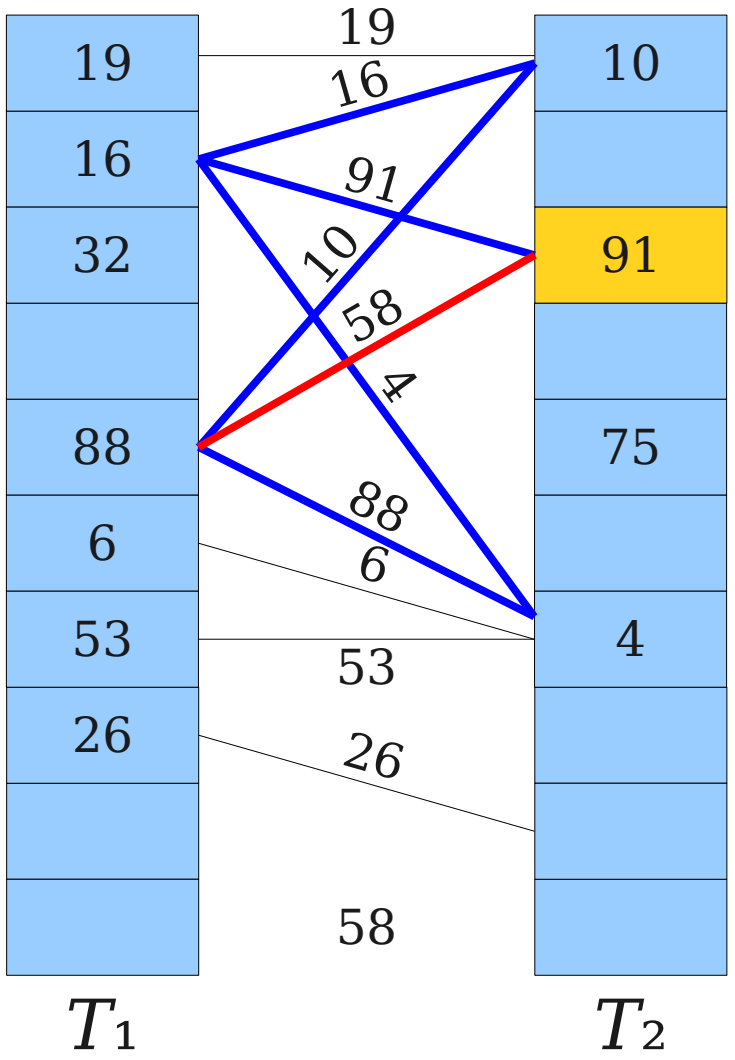
The Cuckoo Graph

- An insertion in a cuckoo hash table traces a path through the cuckoo graph.
- An insertion succeeds iff the connected component containing the inserted value contains at most one cycle.



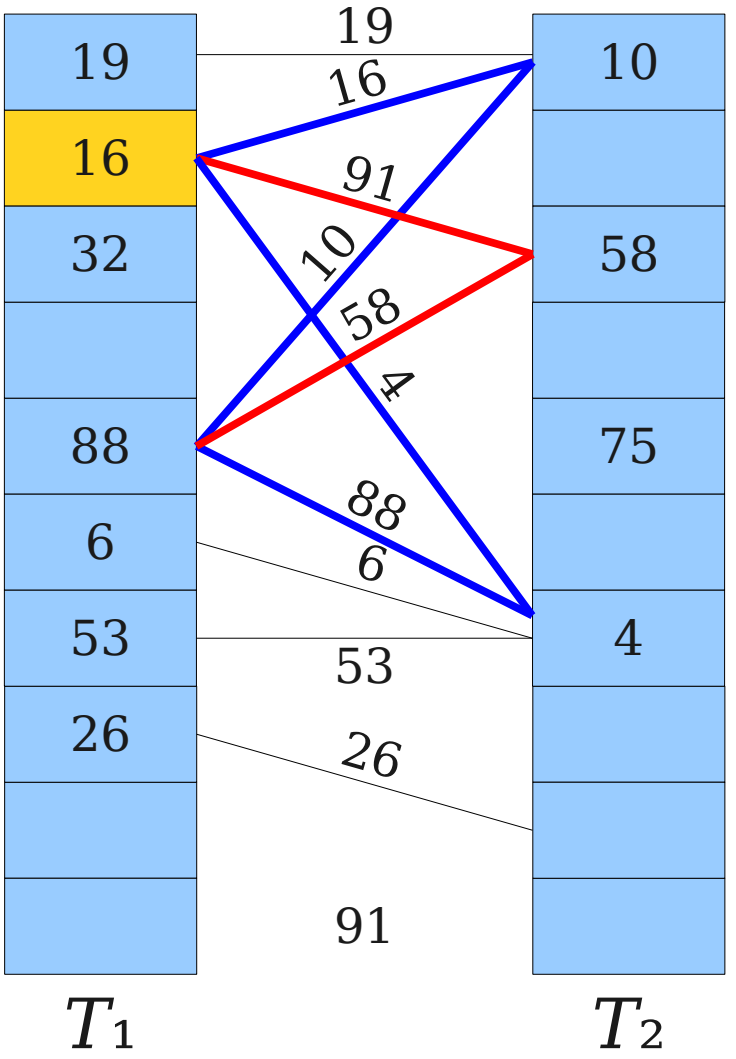
The Cuckoo Graph

- An insertion in a cuckoo hash table traces a path through the cuckoo graph.
- An insertion succeeds iff the connected component containing the inserted value contains at most one cycle.



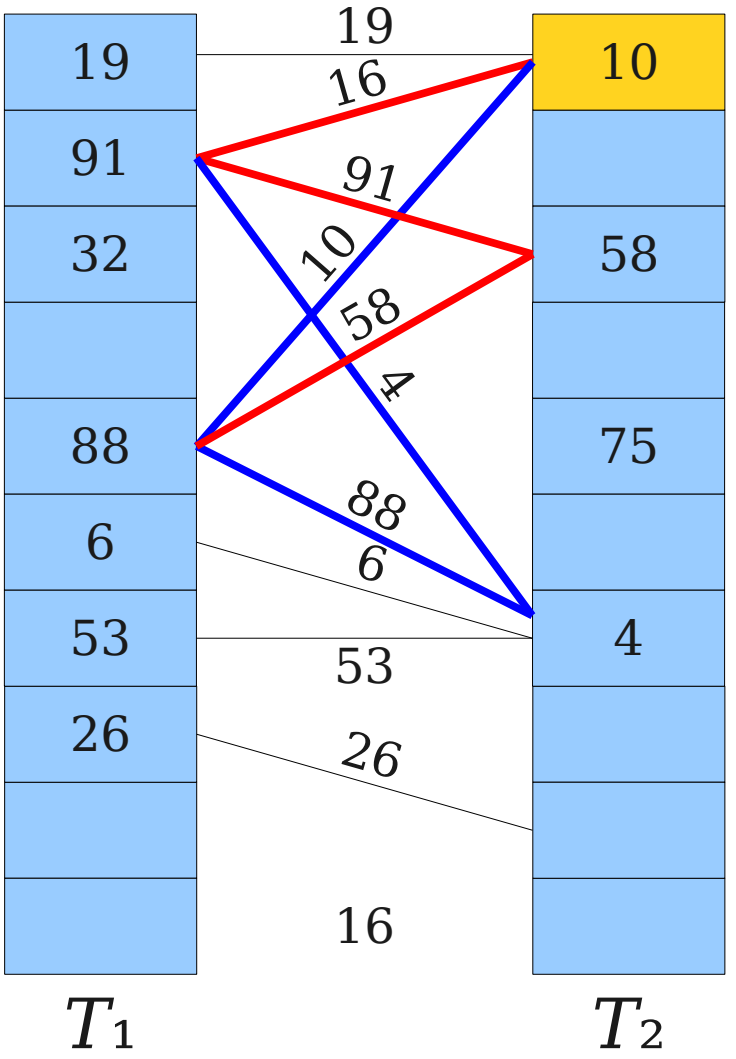
The Cuckoo Graph

- An insertion in a cuckoo hash table traces a path through the cuckoo graph.
- An insertion succeeds iff the connected component containing the inserted value contains at most one cycle.



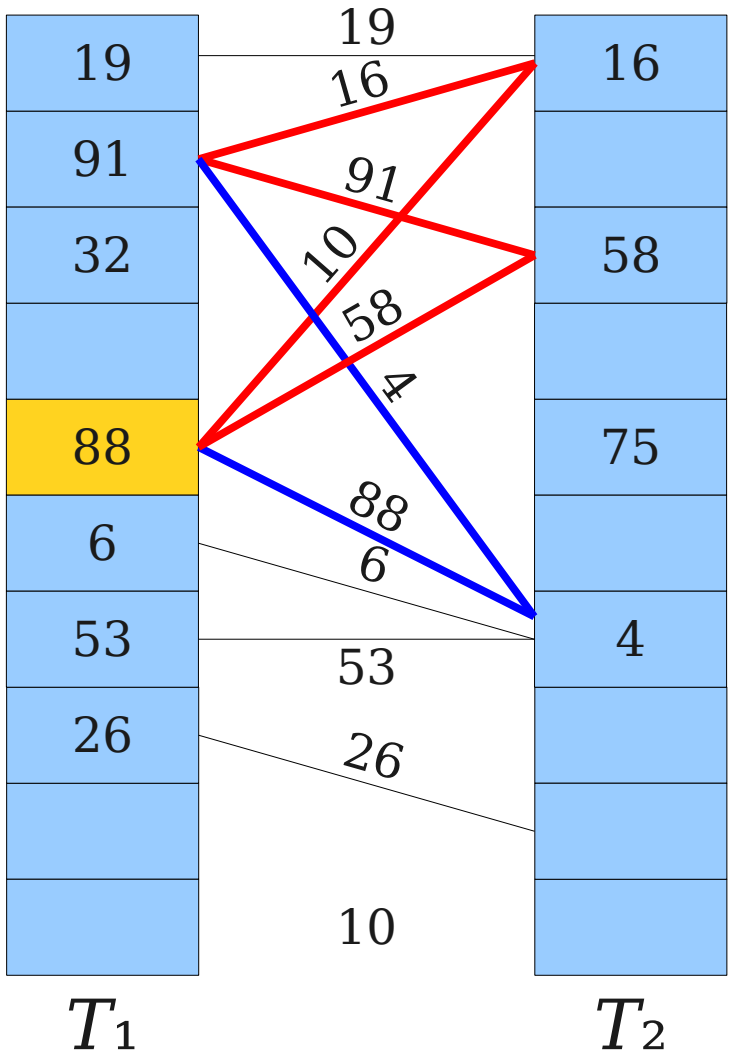
The Cuckoo Graph

- An insertion in a cuckoo hash table traces a path through the cuckoo graph.
- An insertion succeeds iff the connected component containing the inserted value contains at most one cycle.



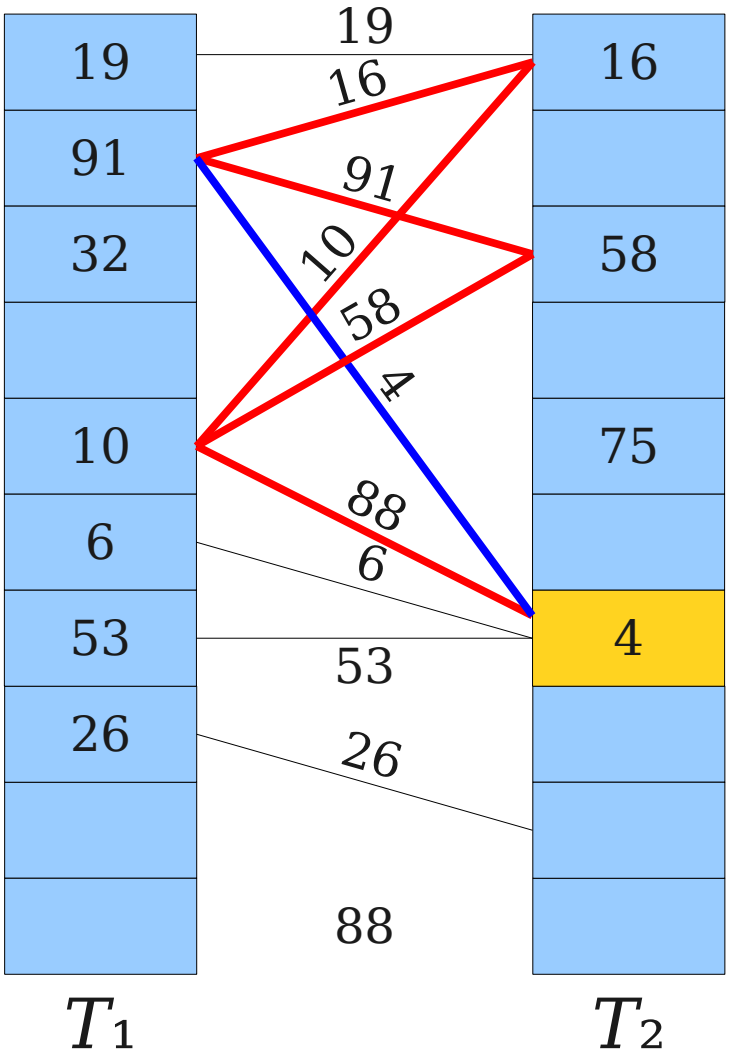
The Cuckoo Graph

- An insertion in a cuckoo hash table traces a path through the cuckoo graph.
- An insertion succeeds iff the connected component containing the inserted value contains at most one cycle.



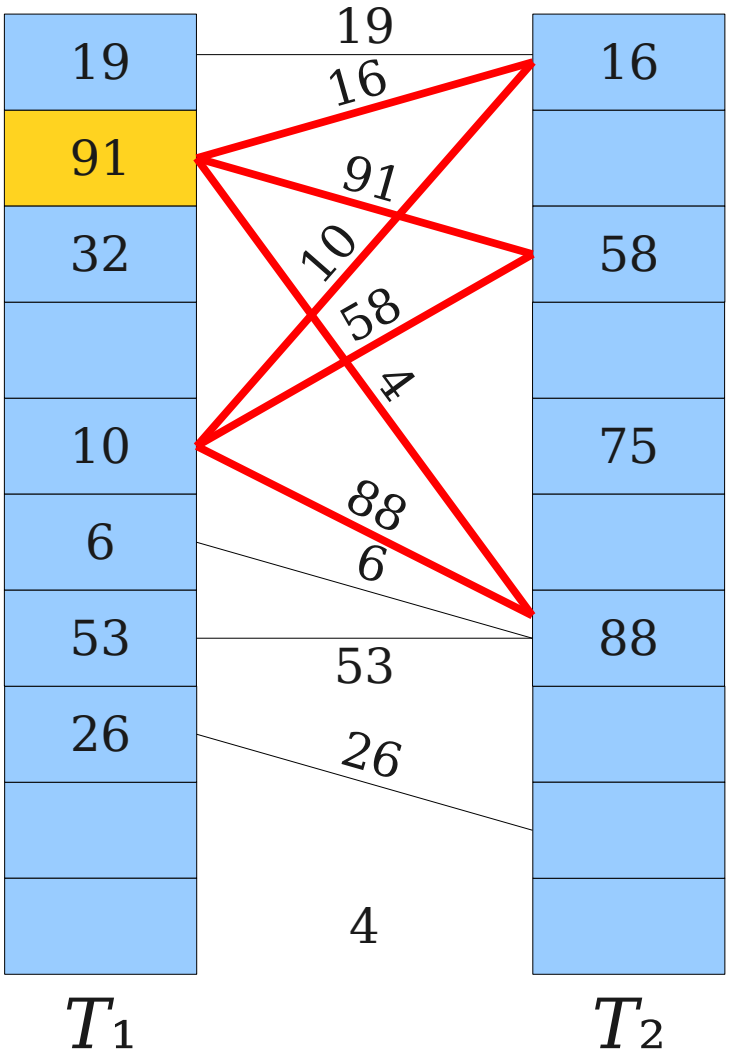
The Cuckoo Graph

- An insertion in a cuckoo hash table traces a path through the cuckoo graph.
- An insertion succeeds iff the connected component containing the inserted value contains at most one cycle.



The Cuckoo Graph

- An insertion in a cuckoo hash table traces a path through the cuckoo graph.
- An insertion succeeds iff the connected component containing the inserted value contains at most one cycle.



The Cuckoo Graph

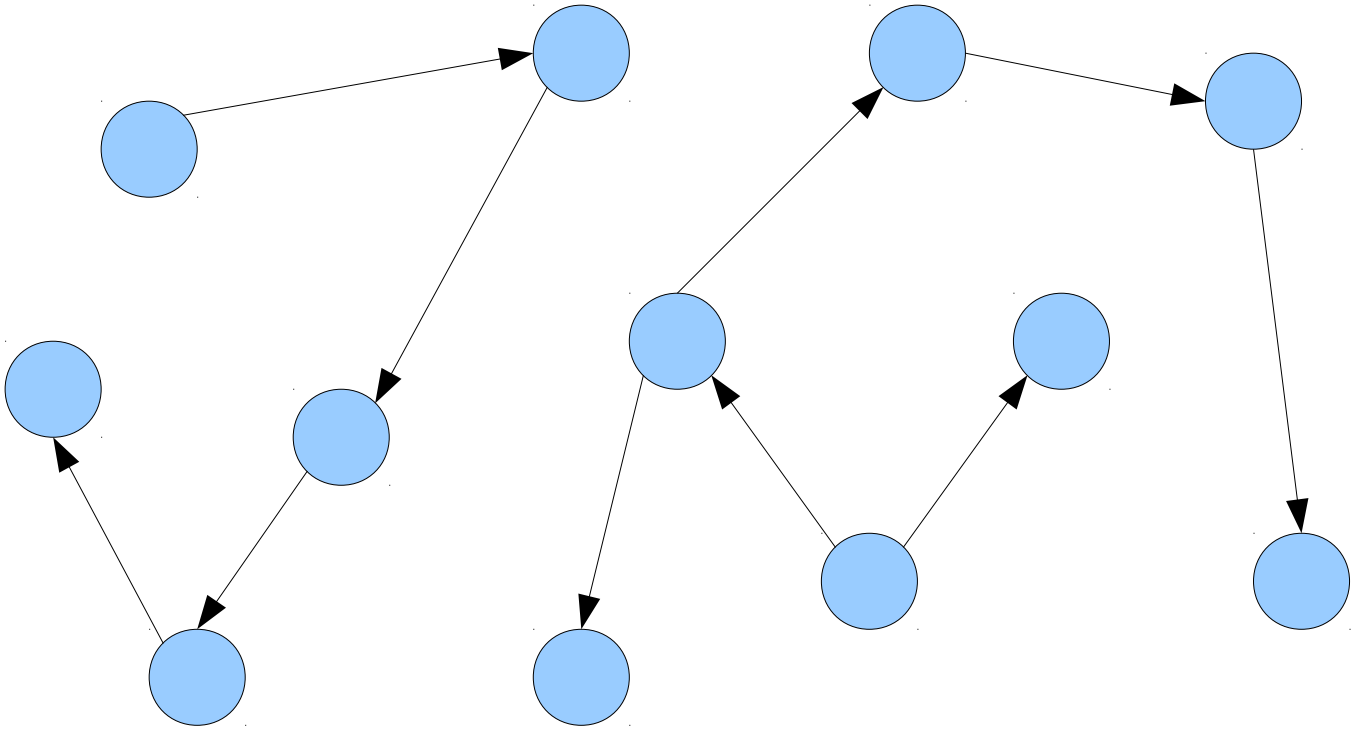
- **Claim 1:** If x is inserted into a cuckoo hash table, the insertion fails if the connected component containing x has two or more cycles.
- **Proof:** Each edge represents an element and needs to be placed in a bucket.
- If the number of nodes (buckets) in the CC is k, then there must be at least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

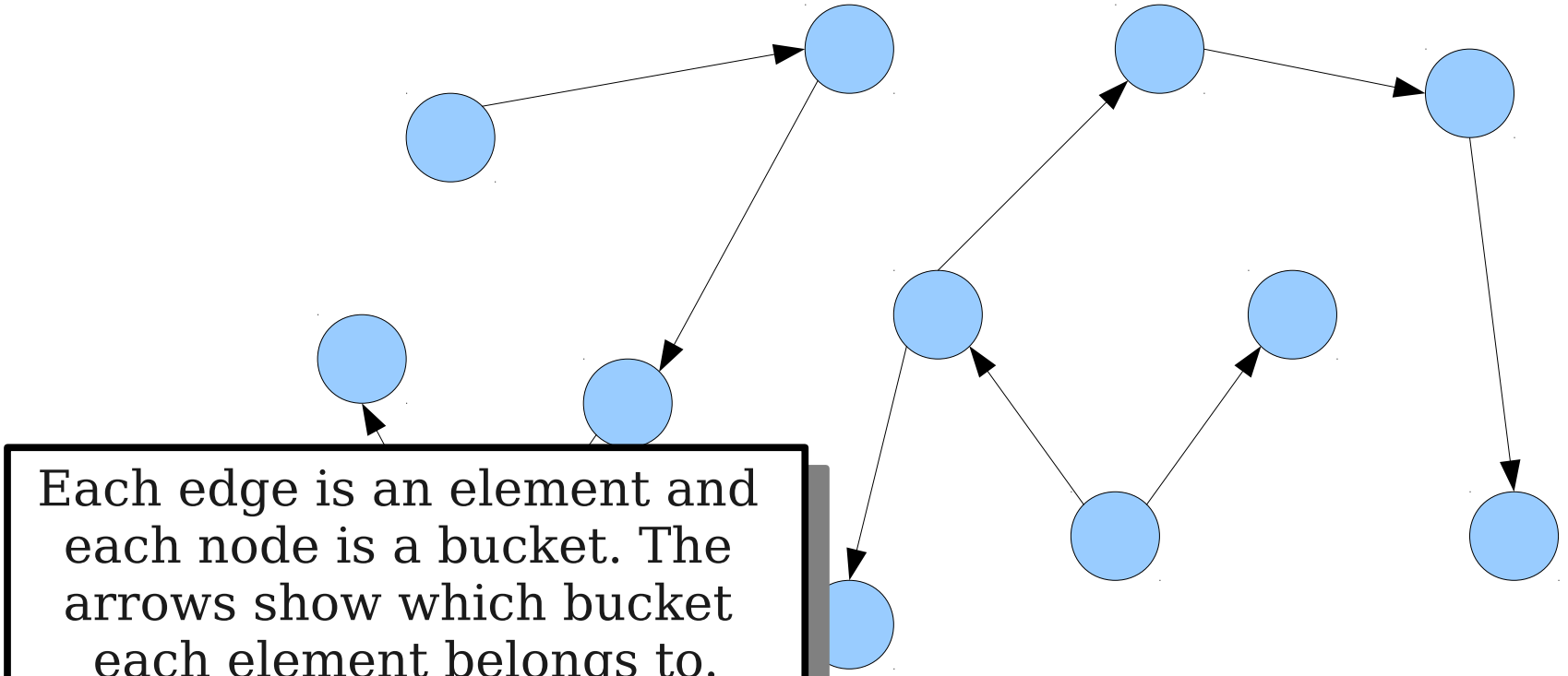
The Cuckoo Graph

- **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

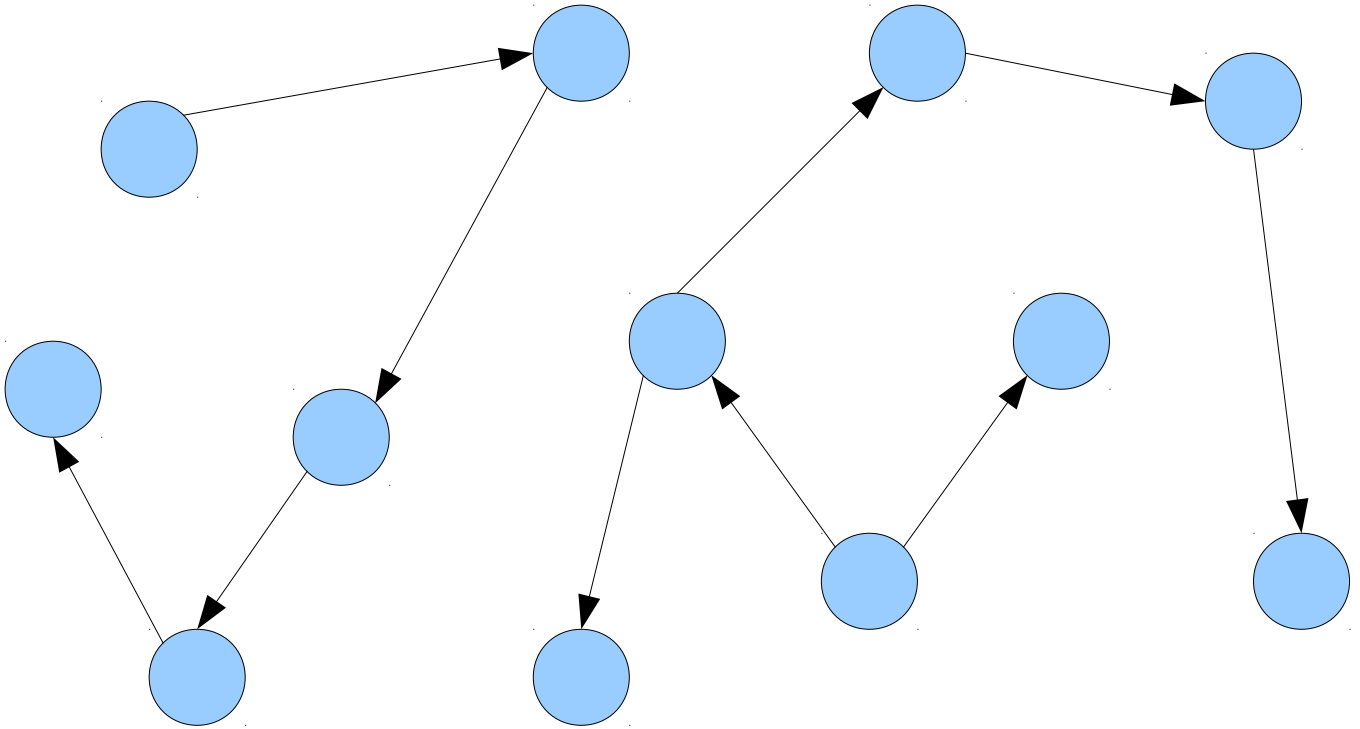
- **Claim 2:** If x is inserted into a cuckoo hash table, the insertion succeeds if the connected component containing x contains either no cycles or only one cycle.



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

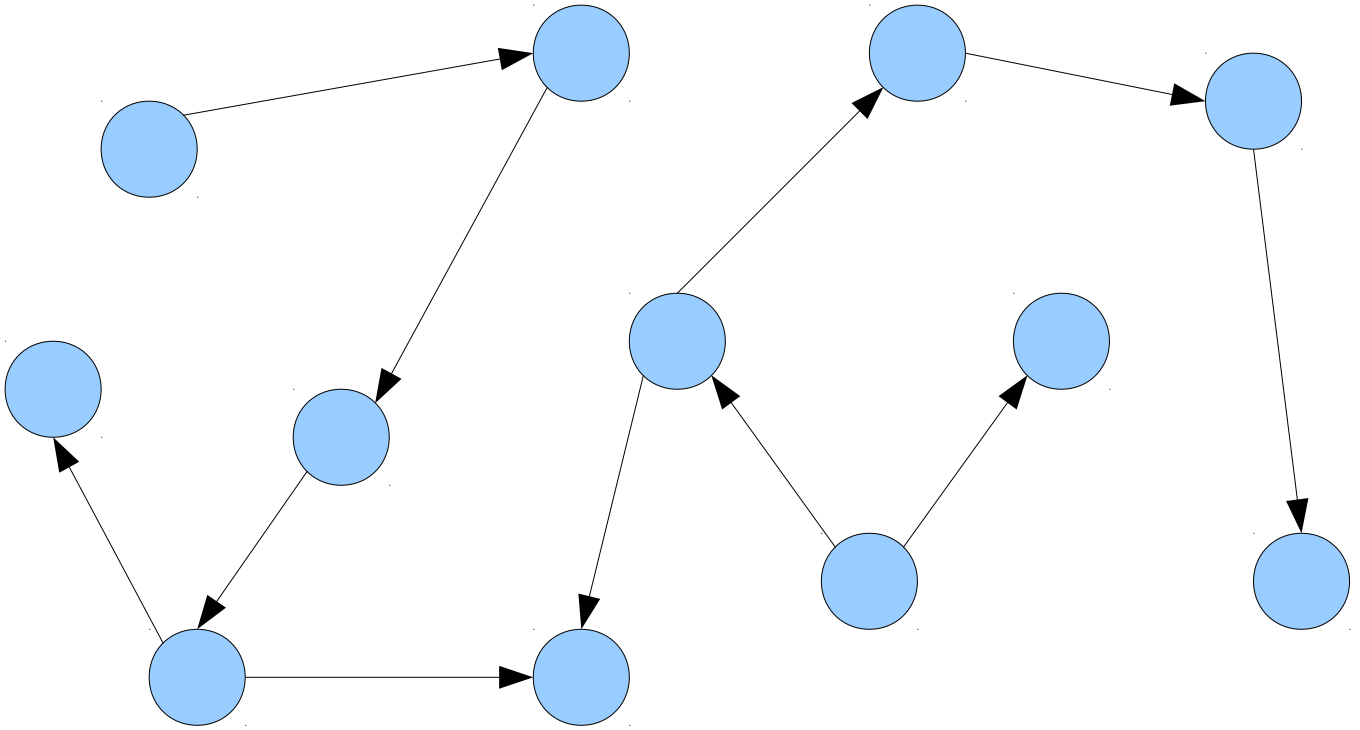
The Cuckoo Graph

- **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.



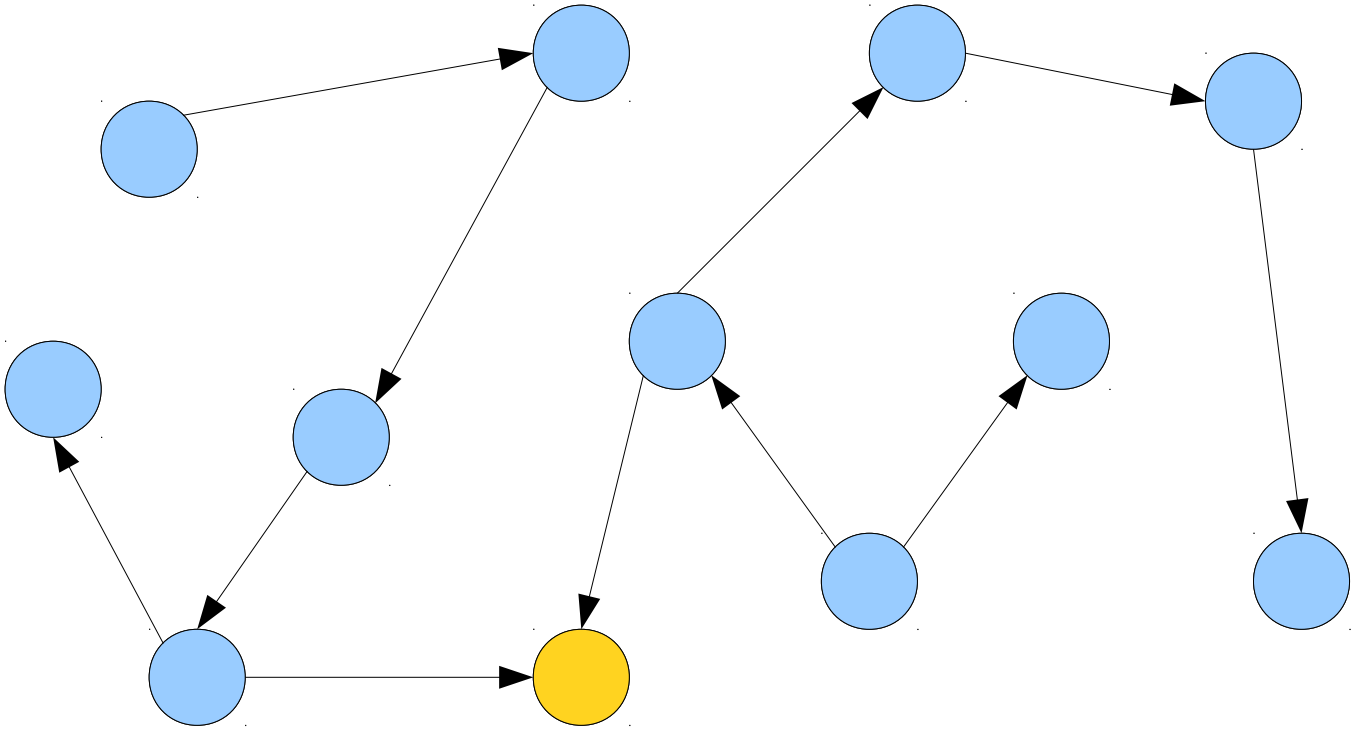
The Cuckoo Graph

- **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.



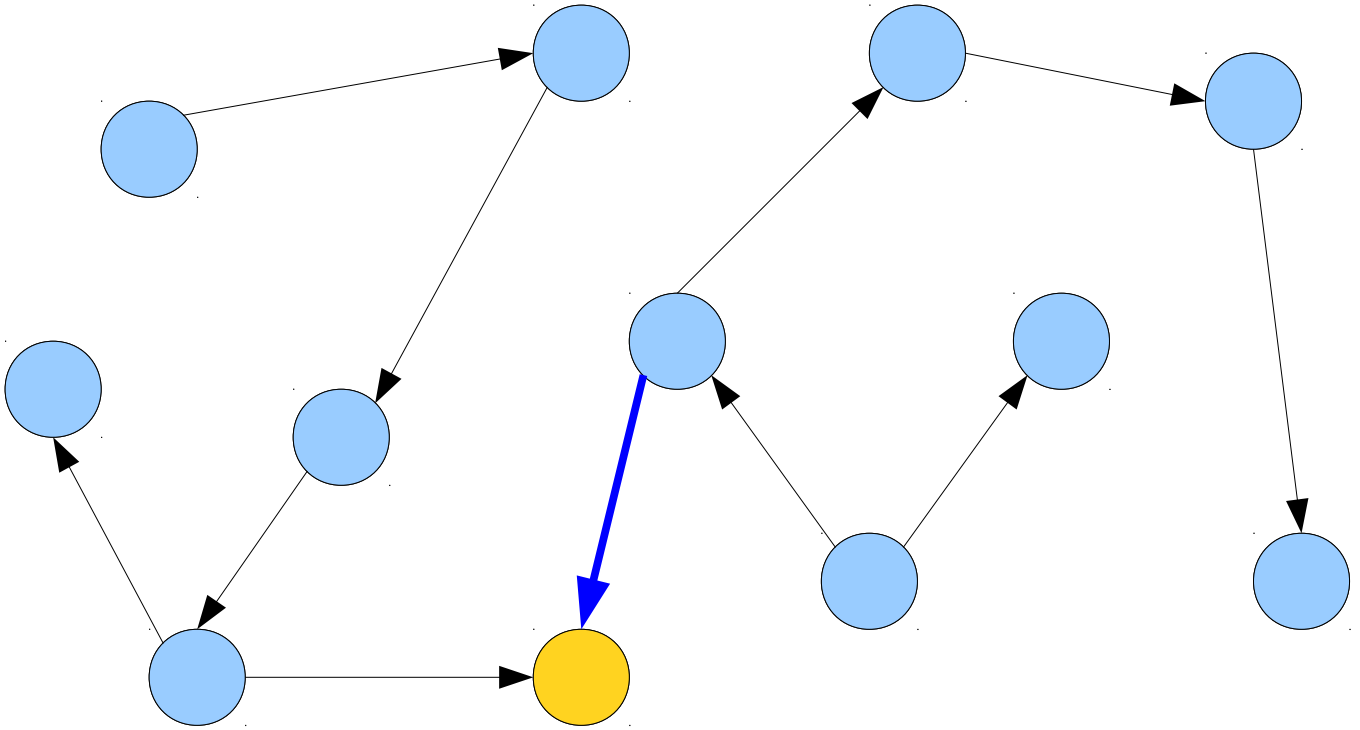
The Cuckoo Graph

- **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.



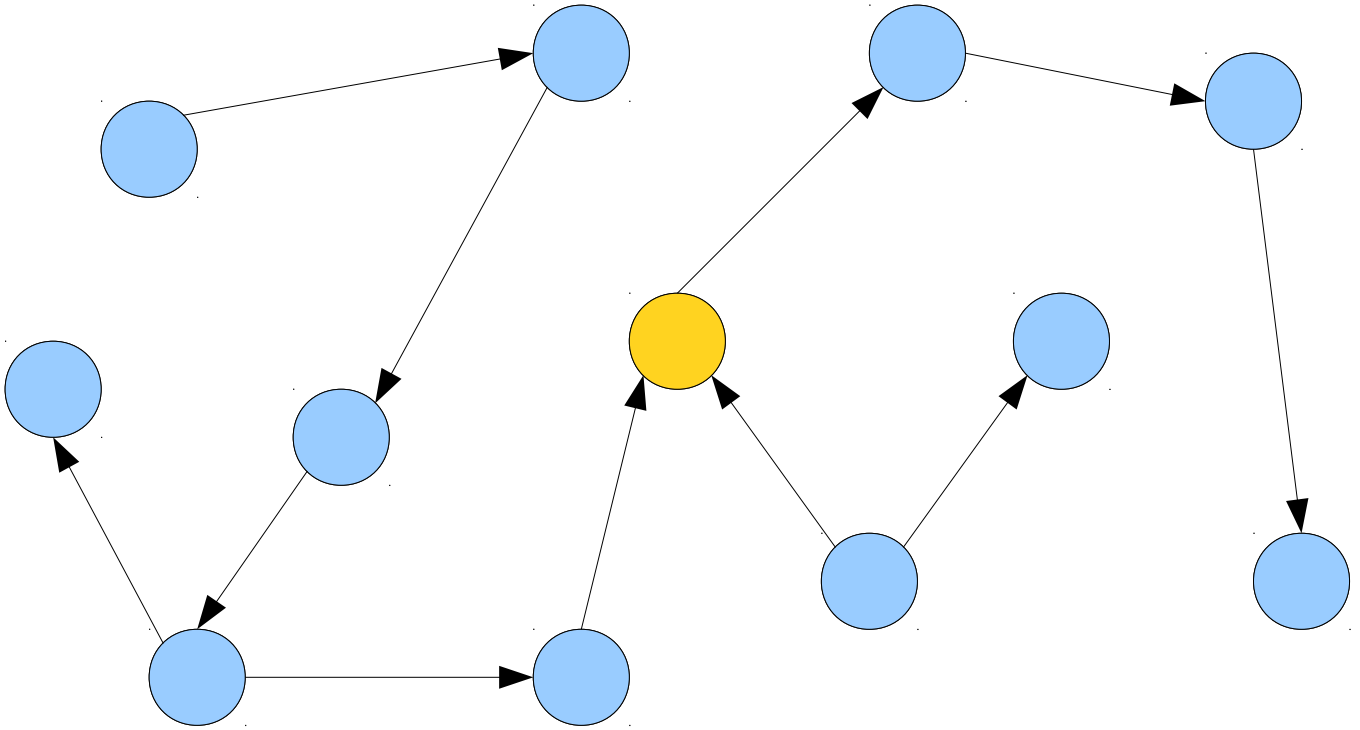
The Cuckoo Graph

- **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.



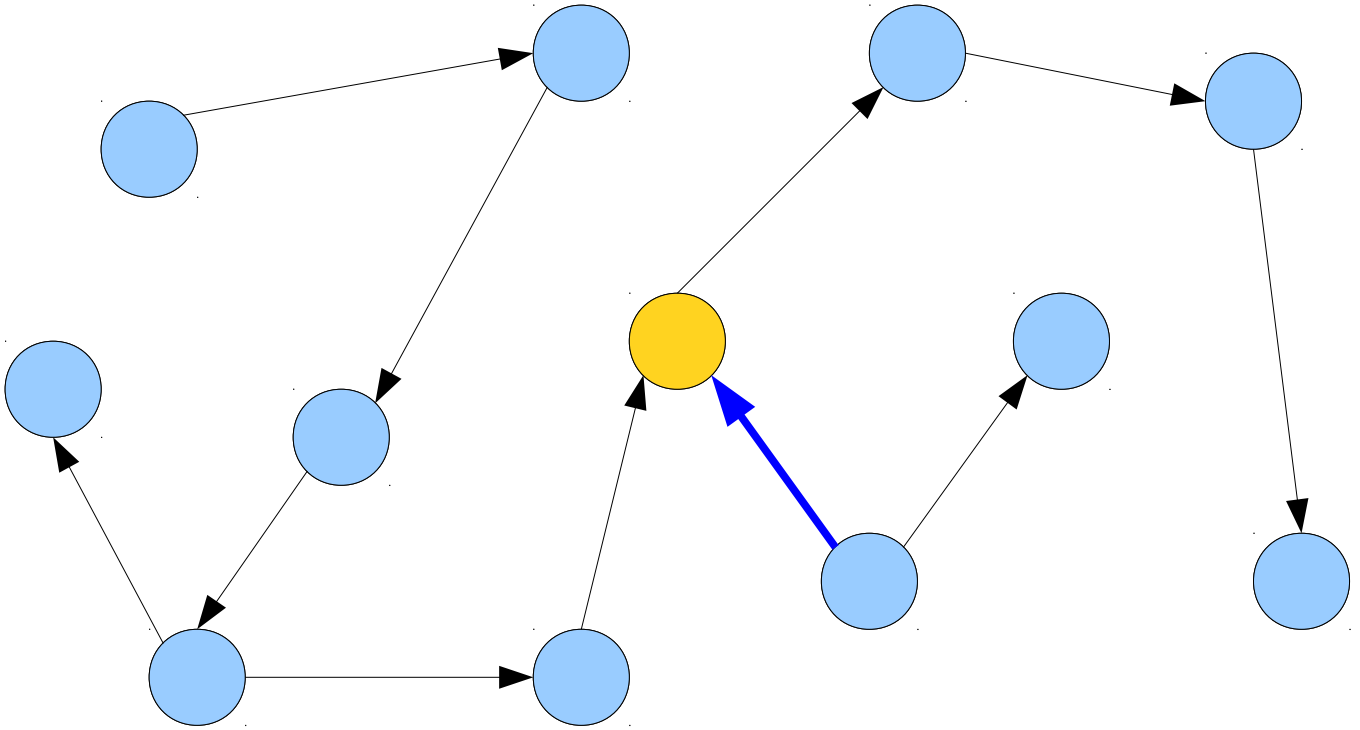
The Cuckoo Graph

- **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.



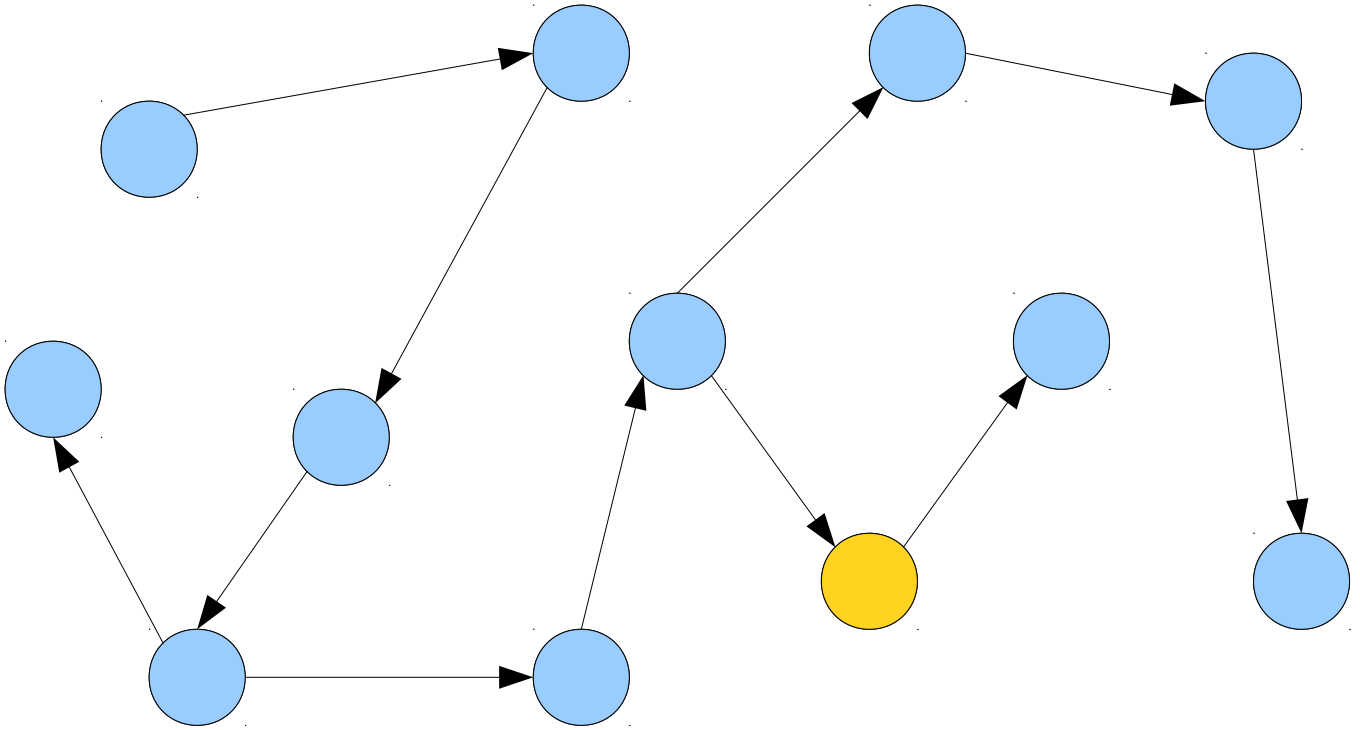
The Cuckoo Graph

- **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.



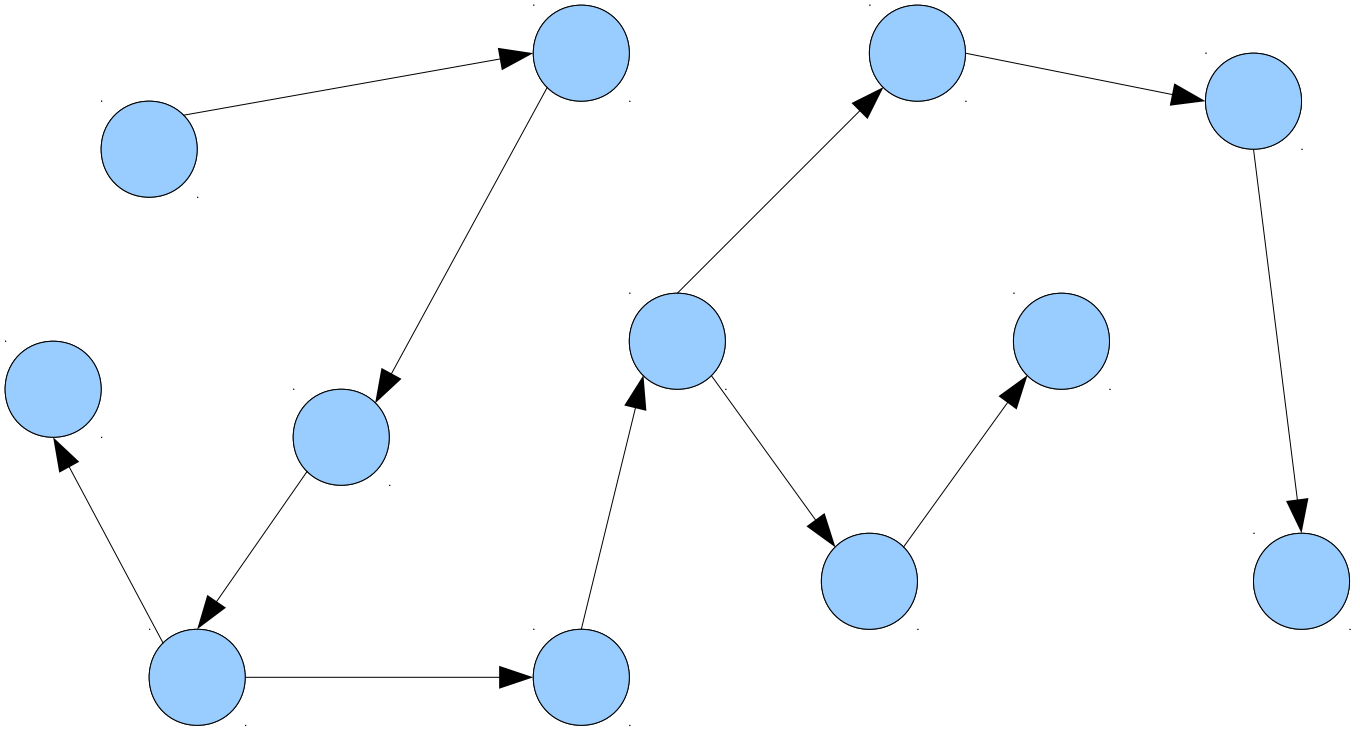
The Cuckoo Graph

- **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.



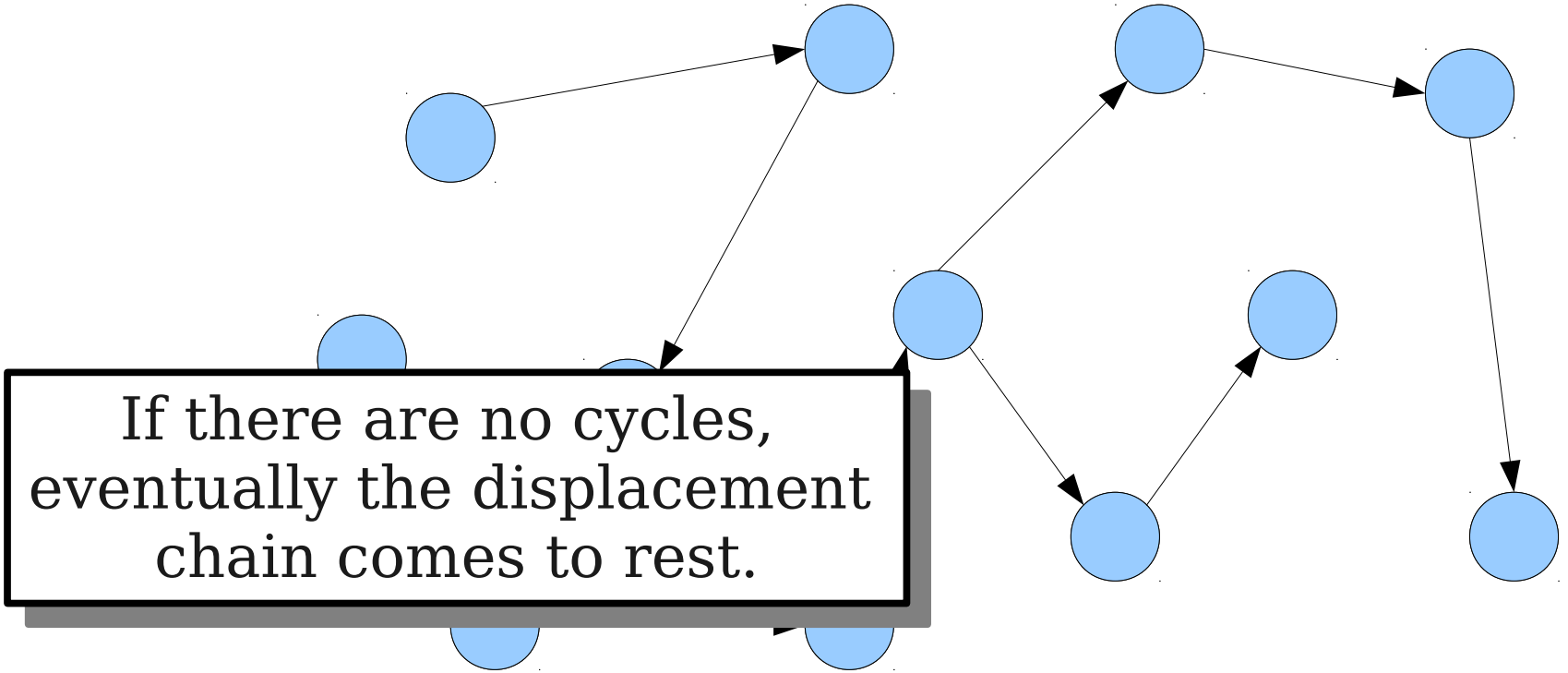
The Cuckoo Graph

- **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.

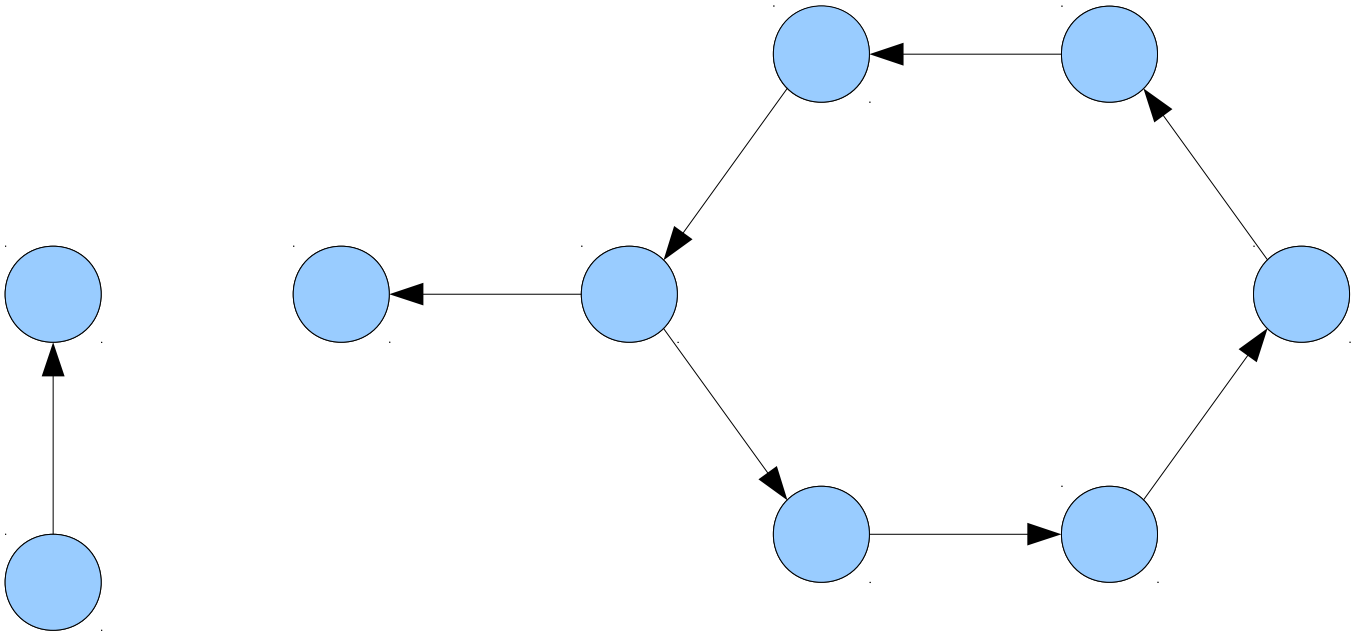


The Cuckoo Graph

- **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.

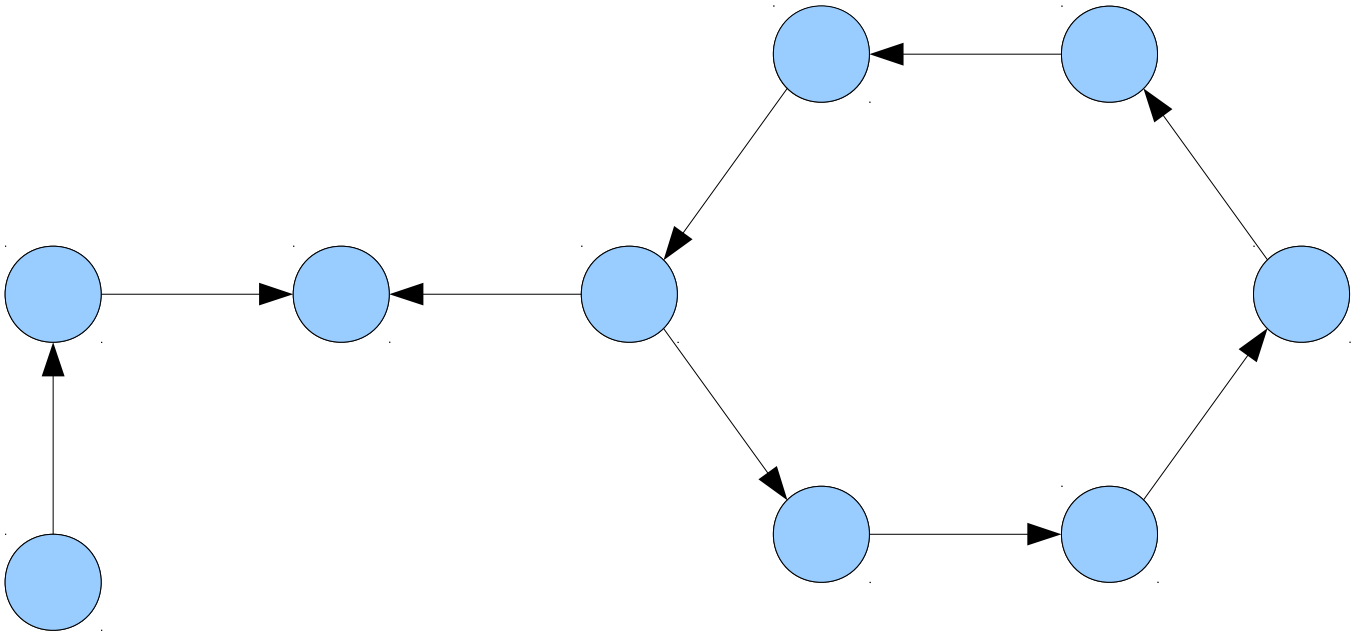
The Cuckoo Graph

- **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.



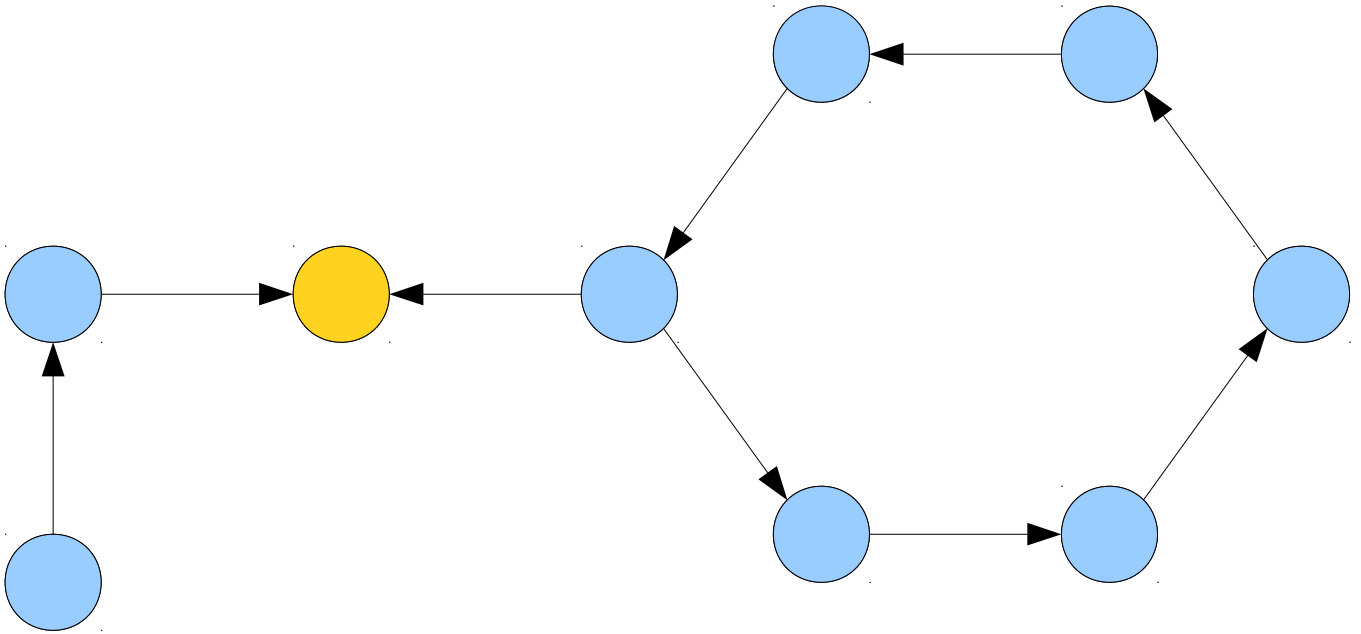
The Cuckoo Graph

- **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.



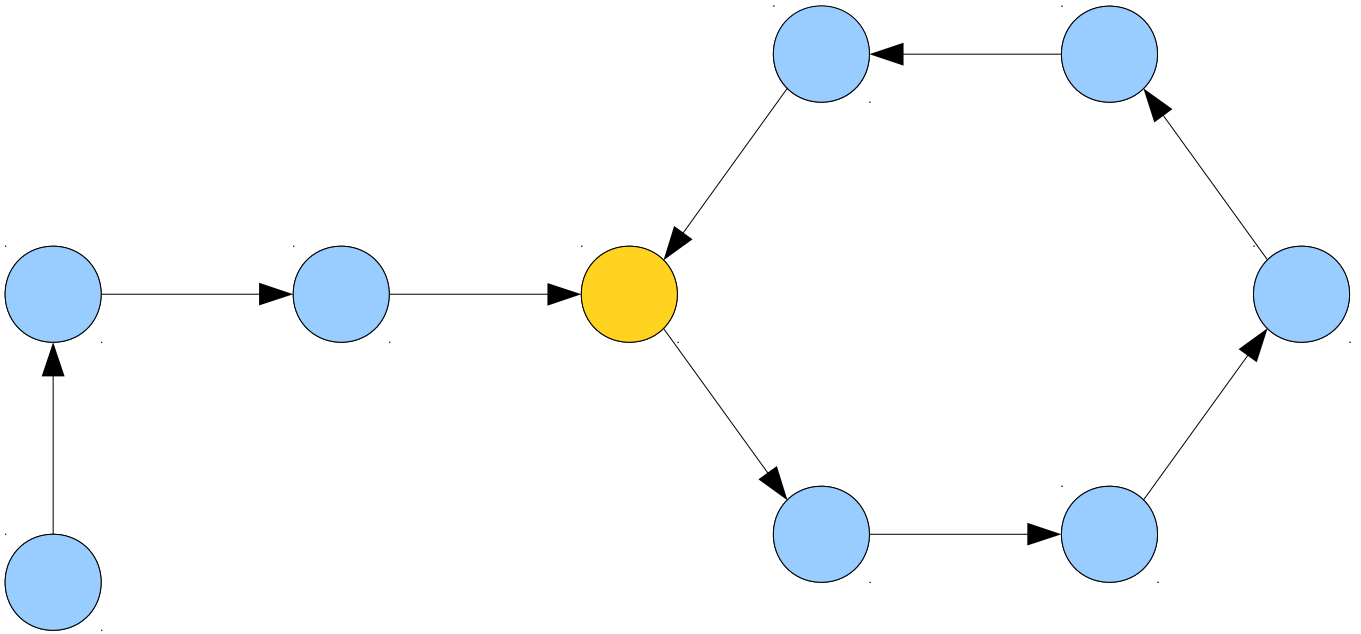
The Cuckoo Graph

- **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.



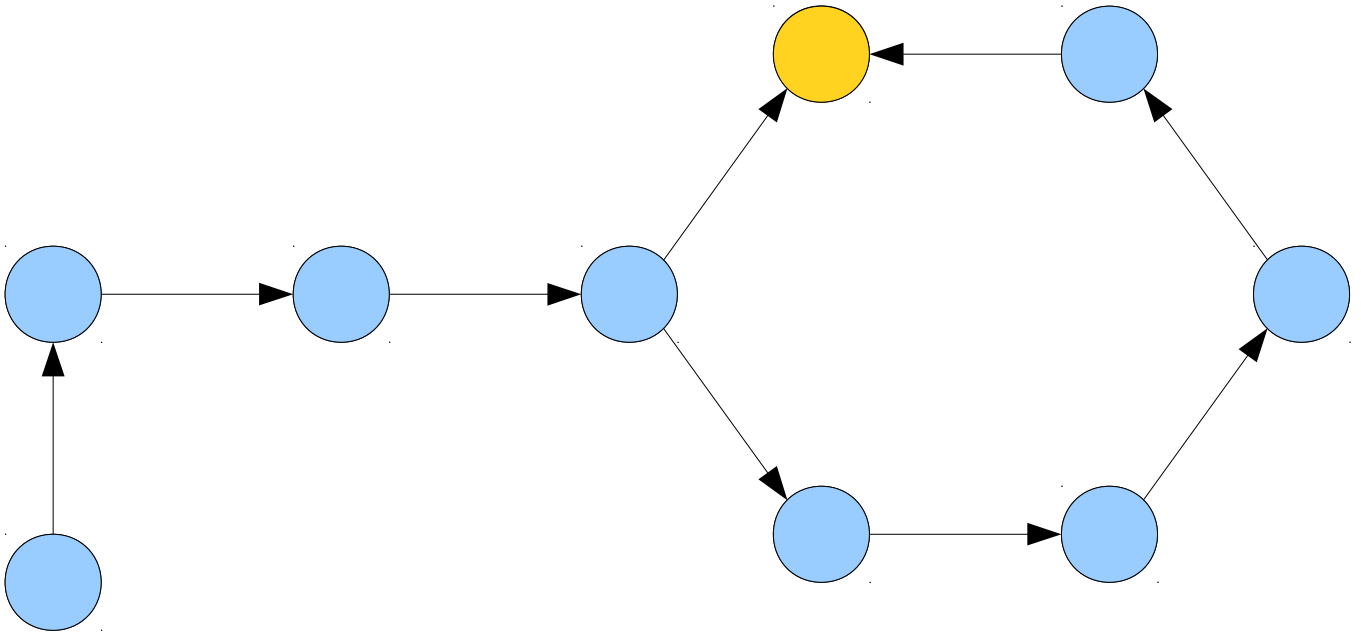
The Cuckoo Graph

- **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.



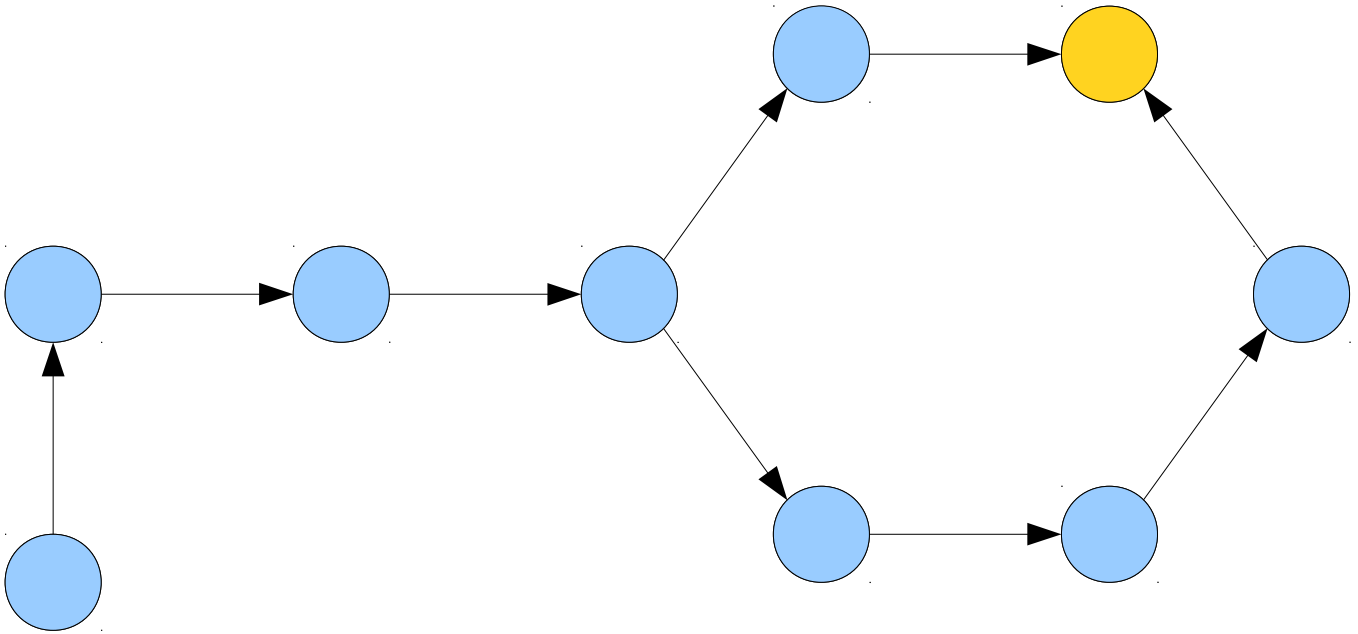
The Cuckoo Graph

- **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.



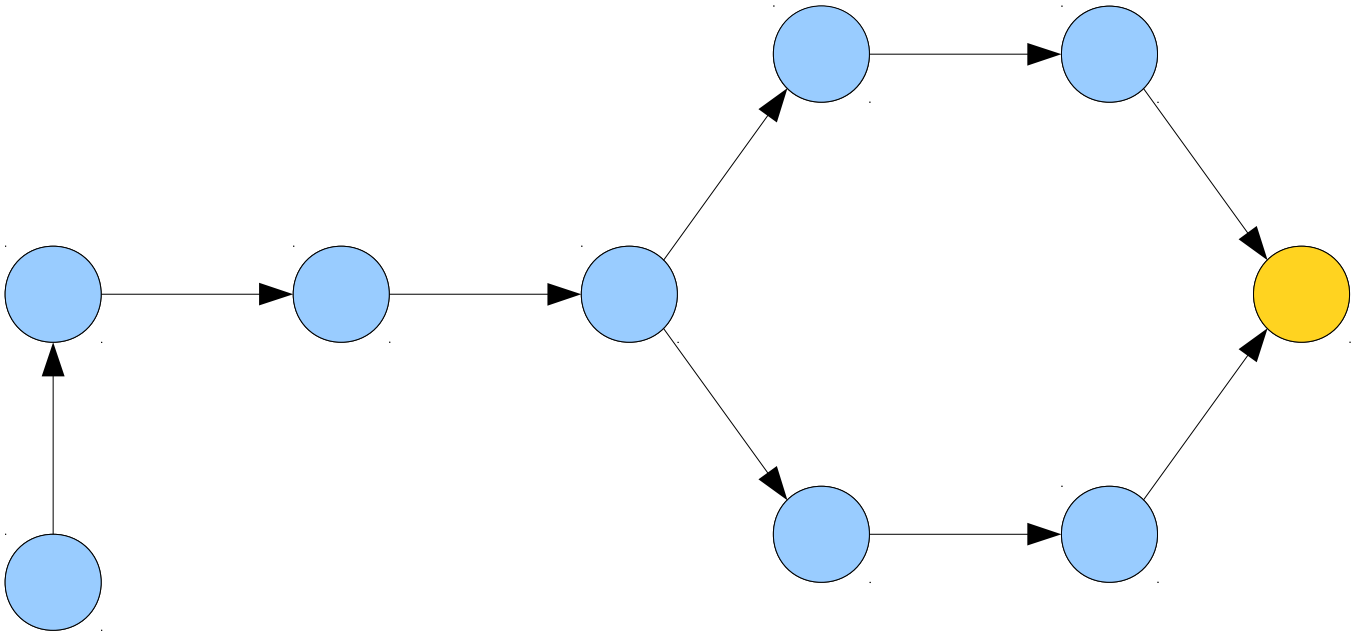
The Cuckoo Graph

- **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.



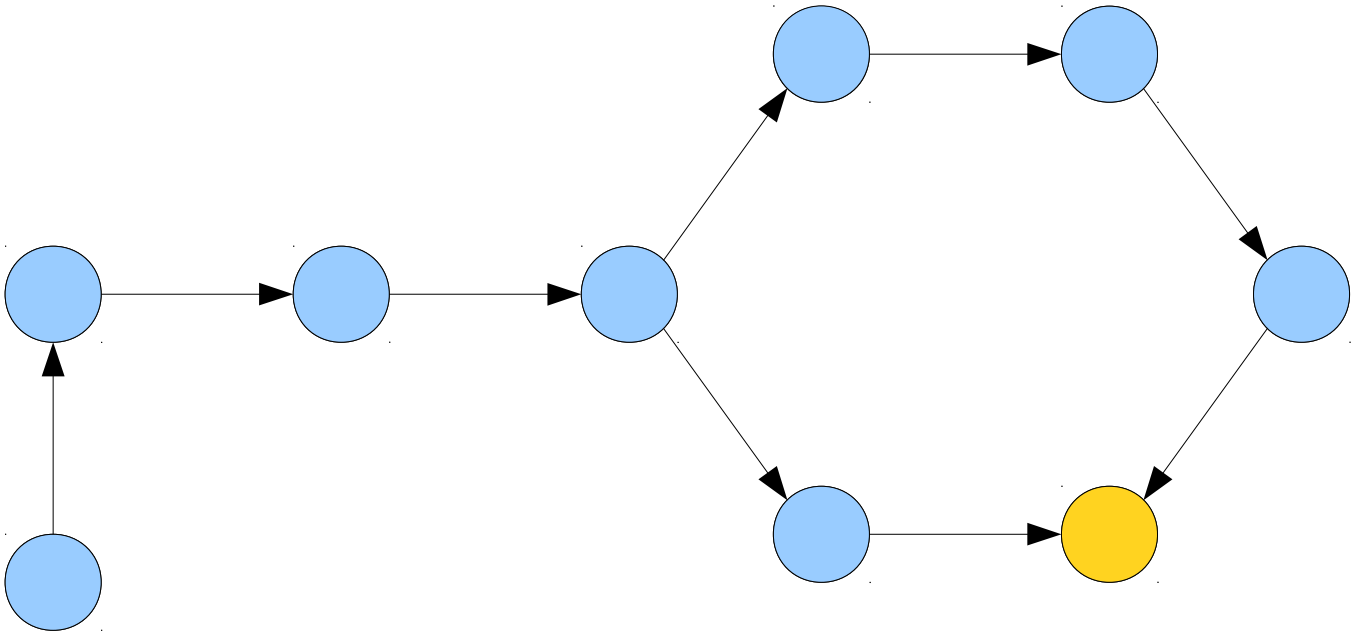
The Cuckoo Graph

- **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.



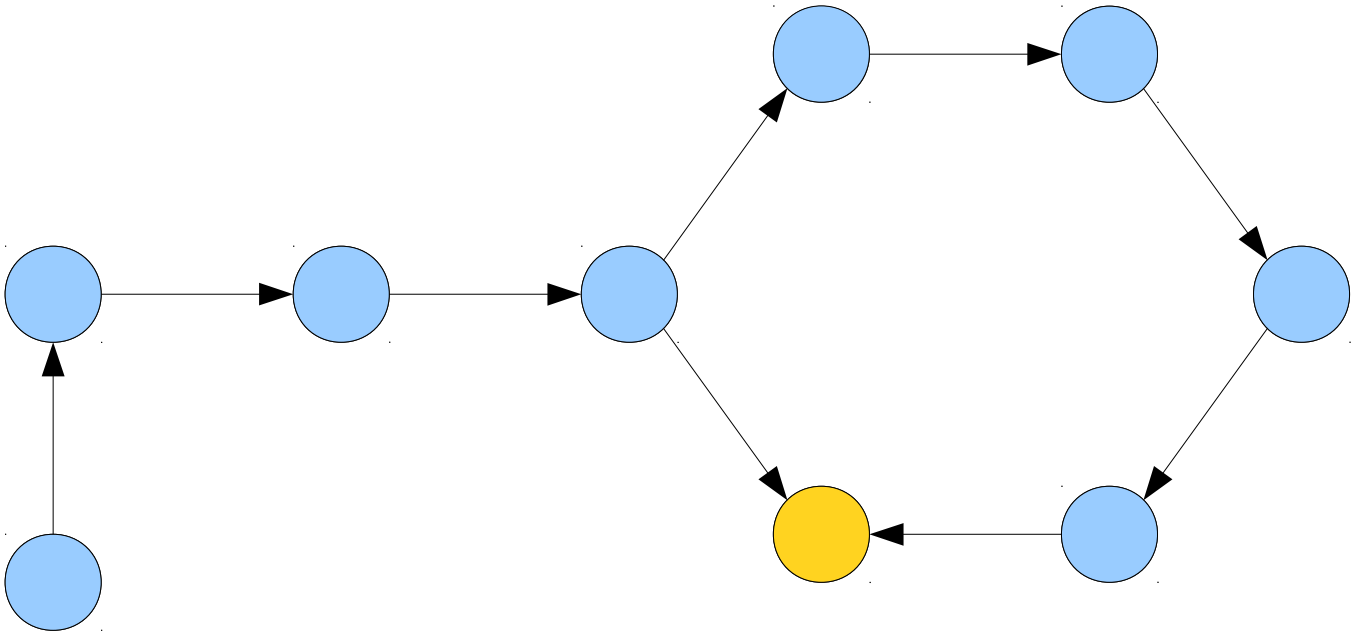
The Cuckoo Graph

- **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.



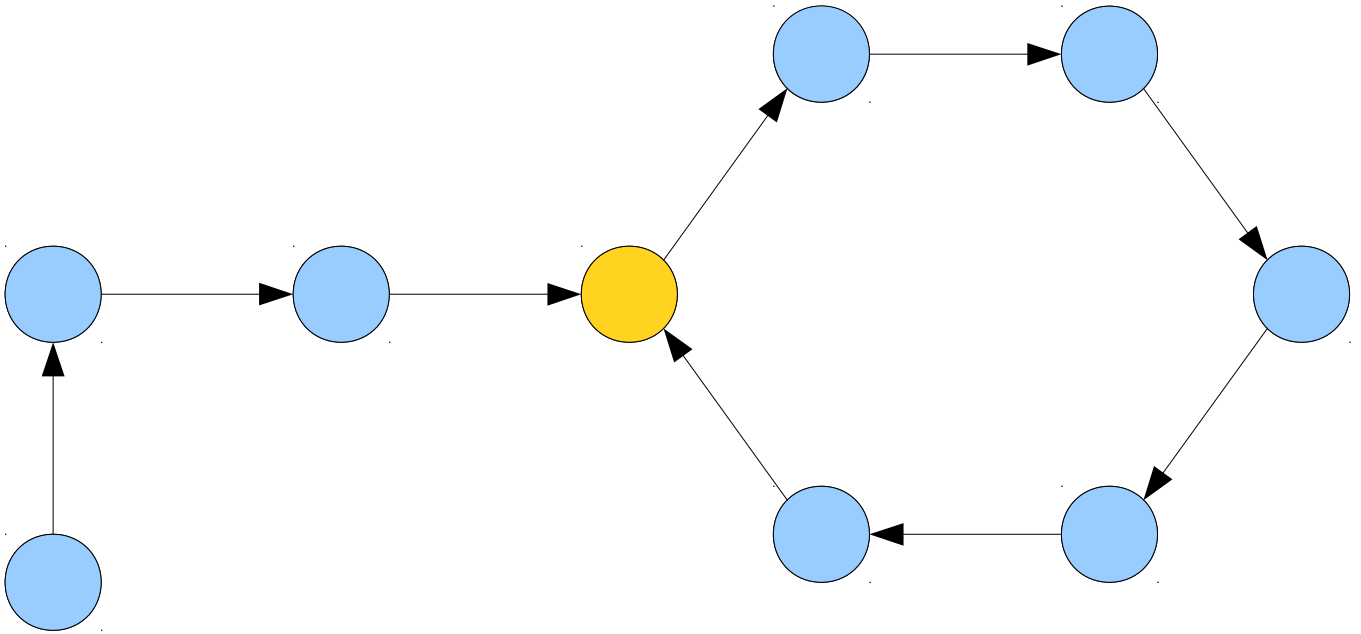
The Cuckoo Graph

- **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.



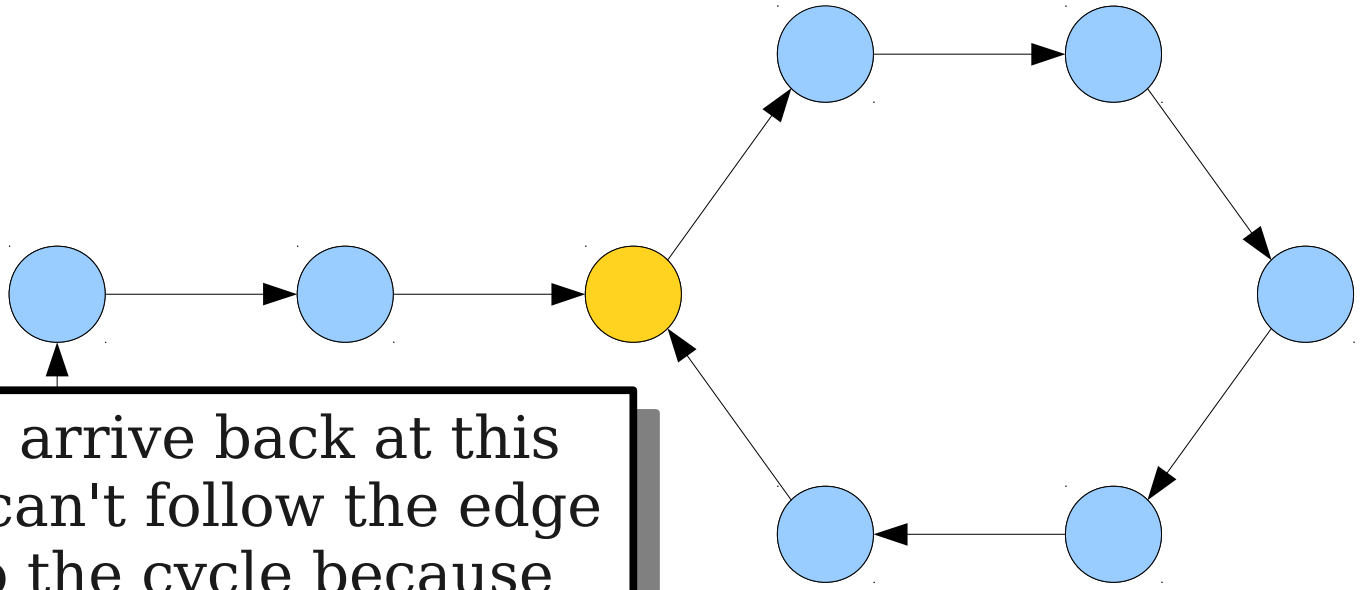
The Cuckoo Graph

- **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

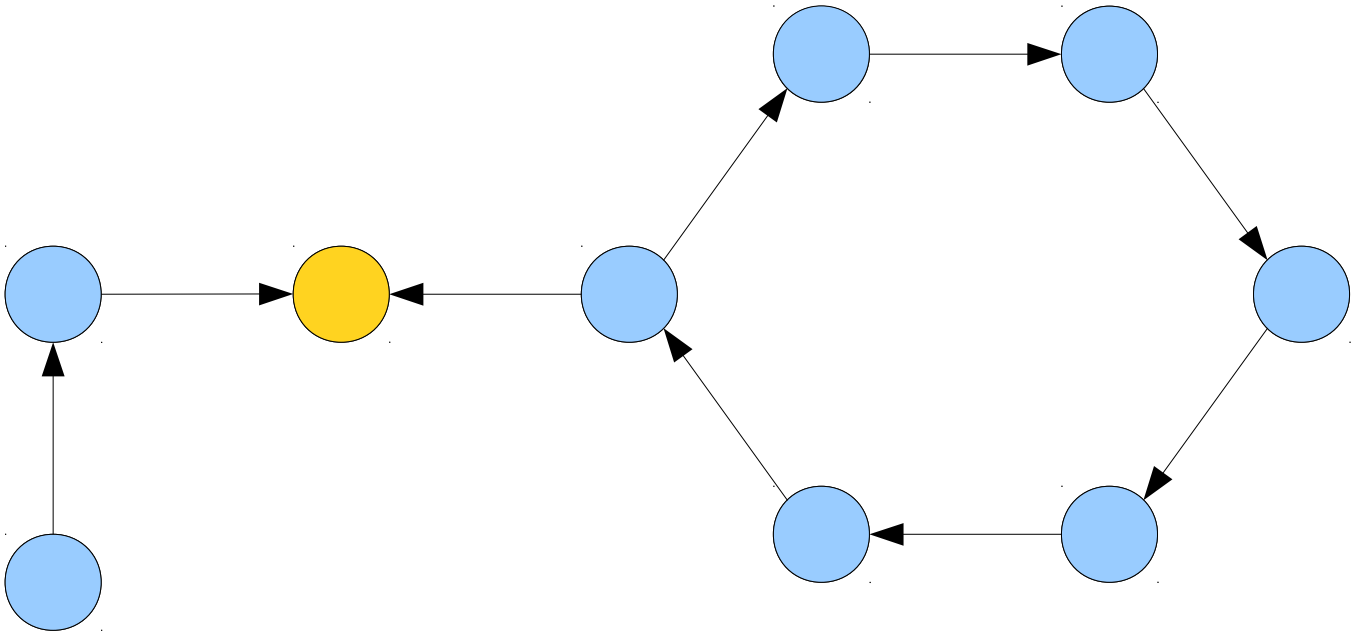
- **Claim 2:** If x is inserted into a cuckoo hash table, the insertion succeeds if the connected component containing x contains either no cycles or only one cycle.



When we arrive back at this node, we can't follow the edge back into the cycle because it's flipped the wrong way.

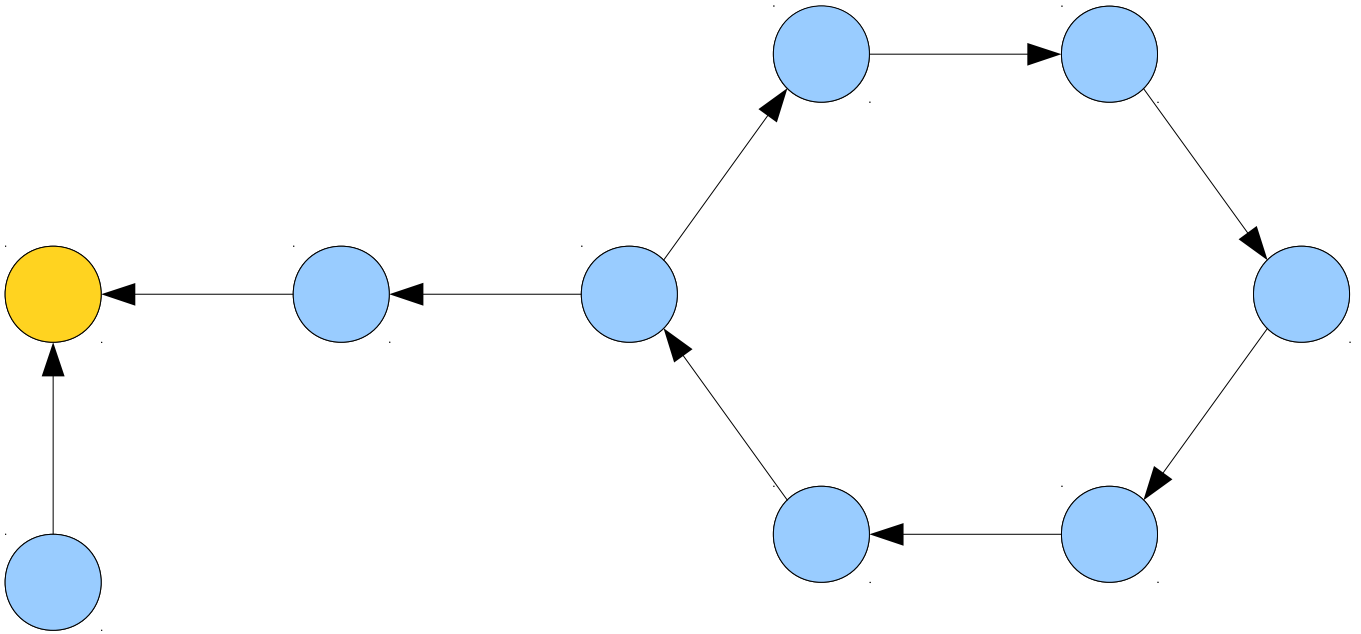
The Cuckoo Graph

- **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.



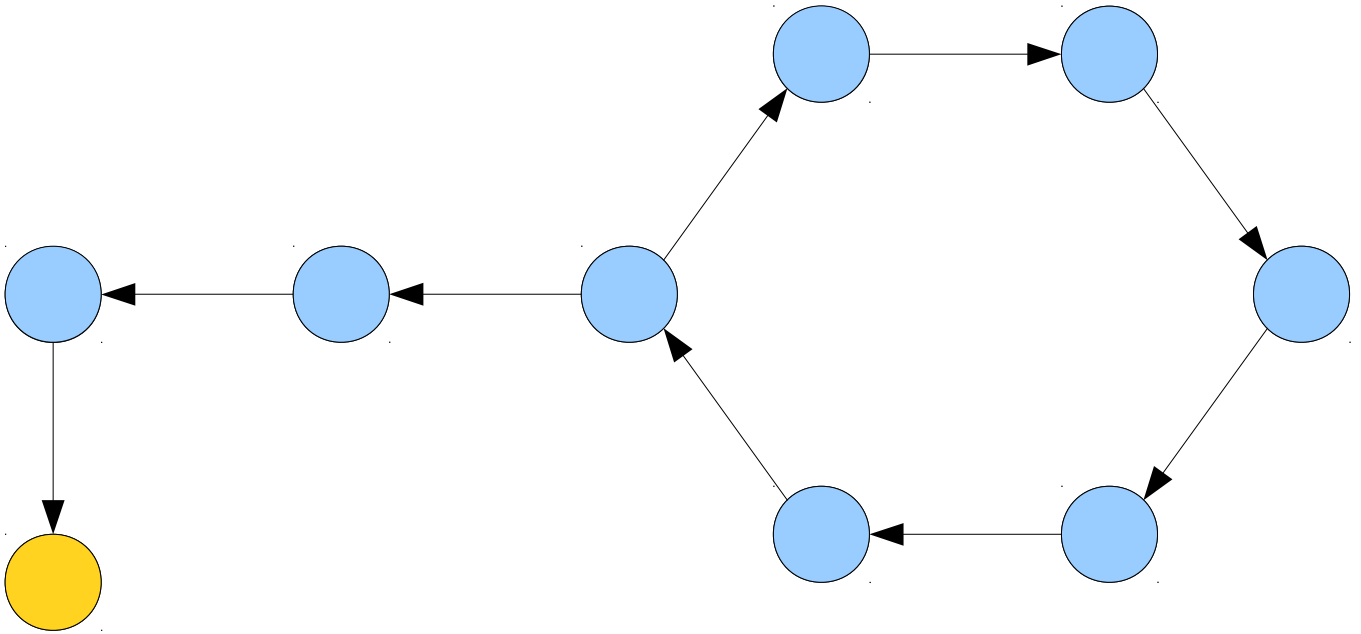
The Cuckoo Graph

- **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.



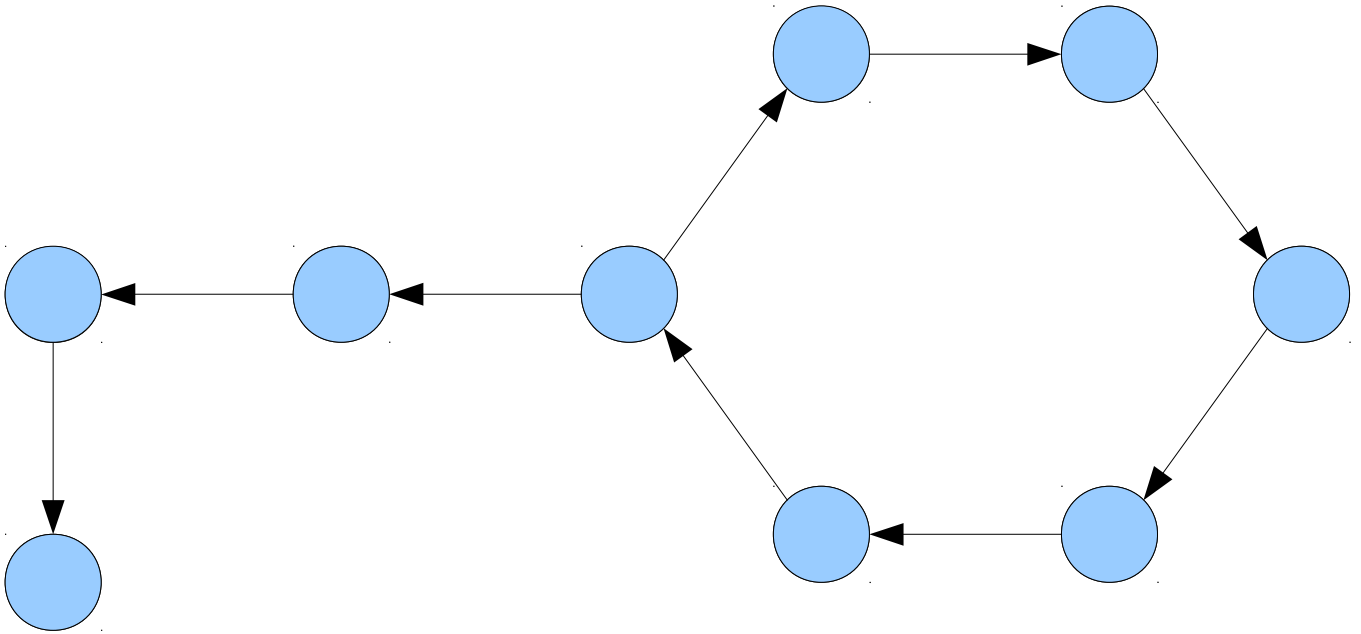
The Cuckoo Graph

- **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.



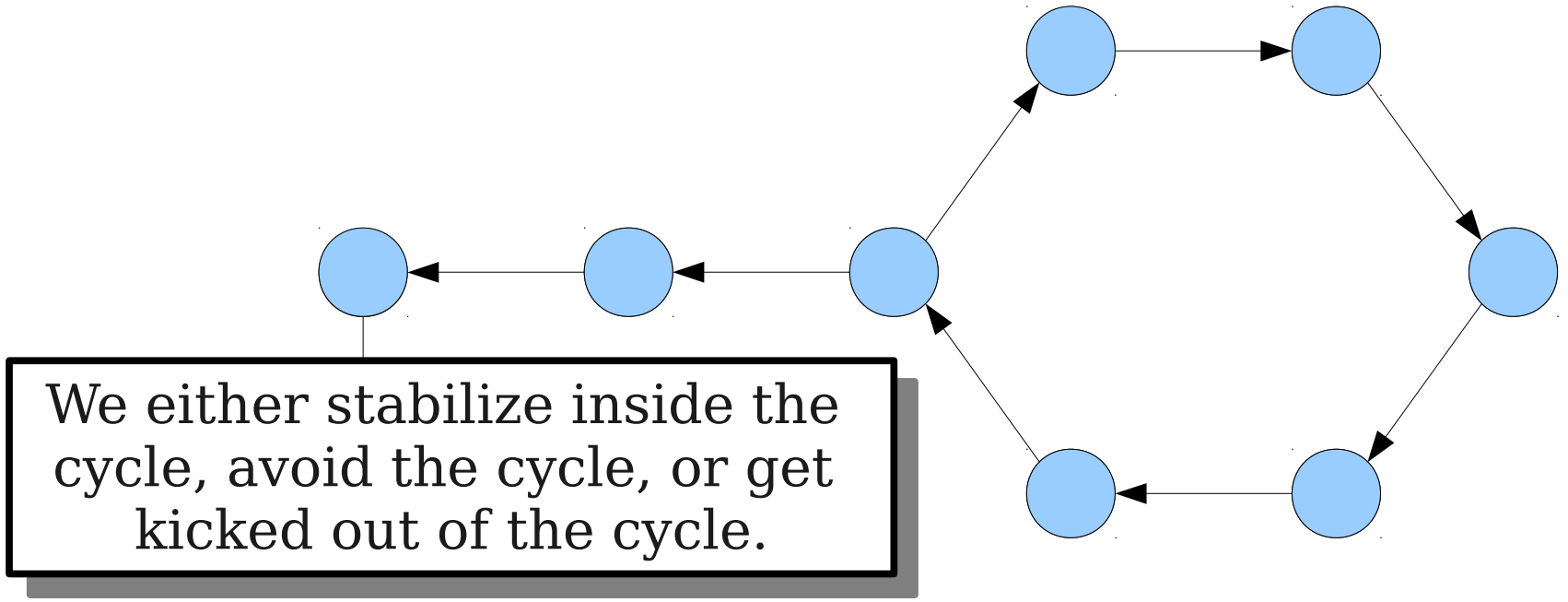
The Cuckoo Graph

- **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.
- **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 7

- Problem Set 7 goes out now and is due next Wednesday at the start of class.
- Play around with randomized data structures and related mathematical techniques!

Final Project Presentations

- We'll review the project proposals we receive today and set up a schedule for presentations.
- Presentations will be in Week 10 and are open to the public.
- We'll send out a sign-up form once we have everything ready.

Midterm

- The midterm is next **Wednesday**, **May 21** from **7PM – 10PM** at **Cemex Auditorium**.
  - Can't make that time? Let us know ASAP so that we can set up an alternate.
- Covers material up through and including next Monday's lecture, but with a strong focus on the topics up through and including this week.
- We'll hold a review session (time/place TBA) and give out a practice midterm later this week.

Your Questions

“Would you consider giving us some ungraded practice exercises for the last two weeks of material? This stuff won't be on the psets or the midterm, so it would be nice to have a few exercises to better solidify our understanding.”

Sure, I can try to do that.

“Why have you been so down on math?”

I'm not sure I understand this question… sorry!

“Can you give an application of data structures in solving a machine learning problem?”

There are lots of applications in clustering. For low-dimensional clustering, data structures like the ***k-d* tree** are useful for doing nearest-neighbor searches. The ***sparse partition*** for multidimensional, dynamic closest pair of points is useful in multidimensional clustering. The ***count-min sketch*** from last time is often used to do exploratory data mining.

“Suppose we need to estimate the k most frequent search queries. With a populated count-min sketch, how can we quickly find k frequent queries? Are there other structures that do this in O(k) time (reservoir sampling?) and what are their error bounds?”

The original paper on count sketches and count-min sketches have great expositions on this. You basically maintain a max-heap coupled with a count(-min) sketch and update frequencies in response to updates.

Random sampling can also be used here, but the sample rate needs to be close to the frequency of the most-frequent element to have good bounds. Check the paper on count sketches for details.

“Can a deterministic algorithm utilize randomized data structure?”

Yep! There's a technique called ***derandomization*** that can be used to turn randomized algorithms and data structures into fully-deterministic ones. It's certainly worth looking into!

Back to CS166!

Step One:

Exploring the Graph Structure

Exploring the Graph Structure

- 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, what the probability is that this occurs.

Awful 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 $m \geq (1 + \varepsilon)n$, then the hash table will have a load factor of $1 / (2 + 2\varepsilon)$.
- This means that roughly half of the table cells will be empty.
- There are techniques for improving upon this; more details later on.

Step Two:

Analyzing Connected Components

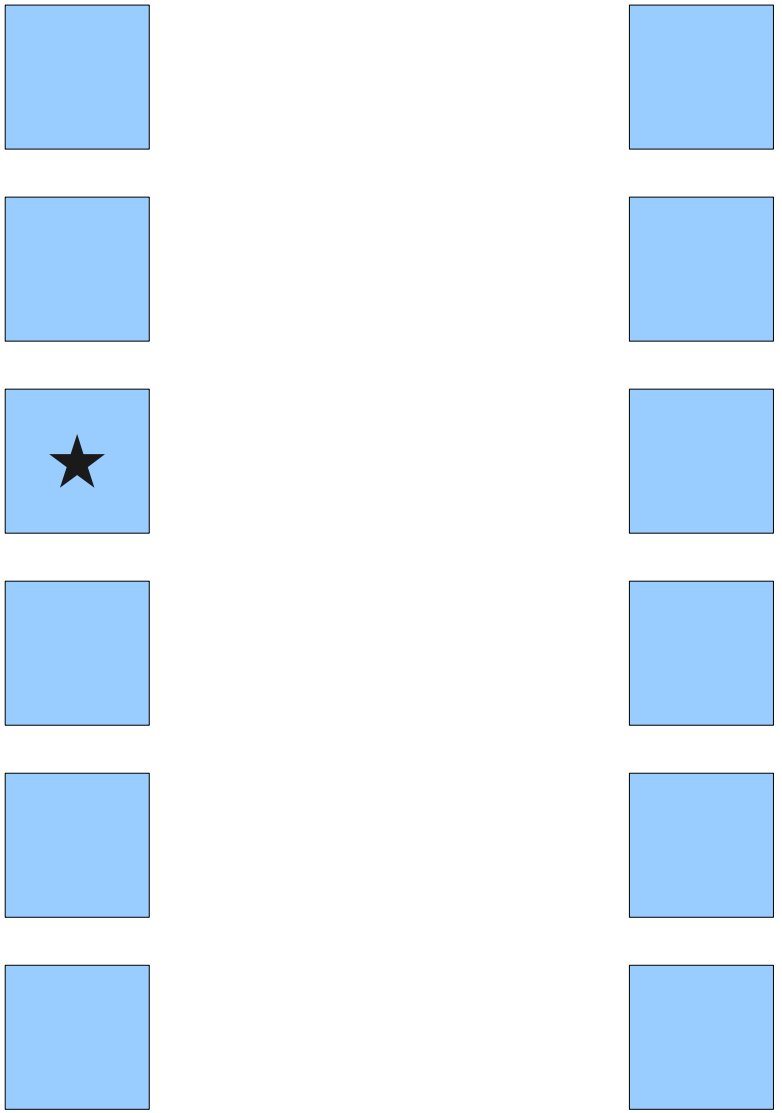
Analyzing Connected Components

- 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?

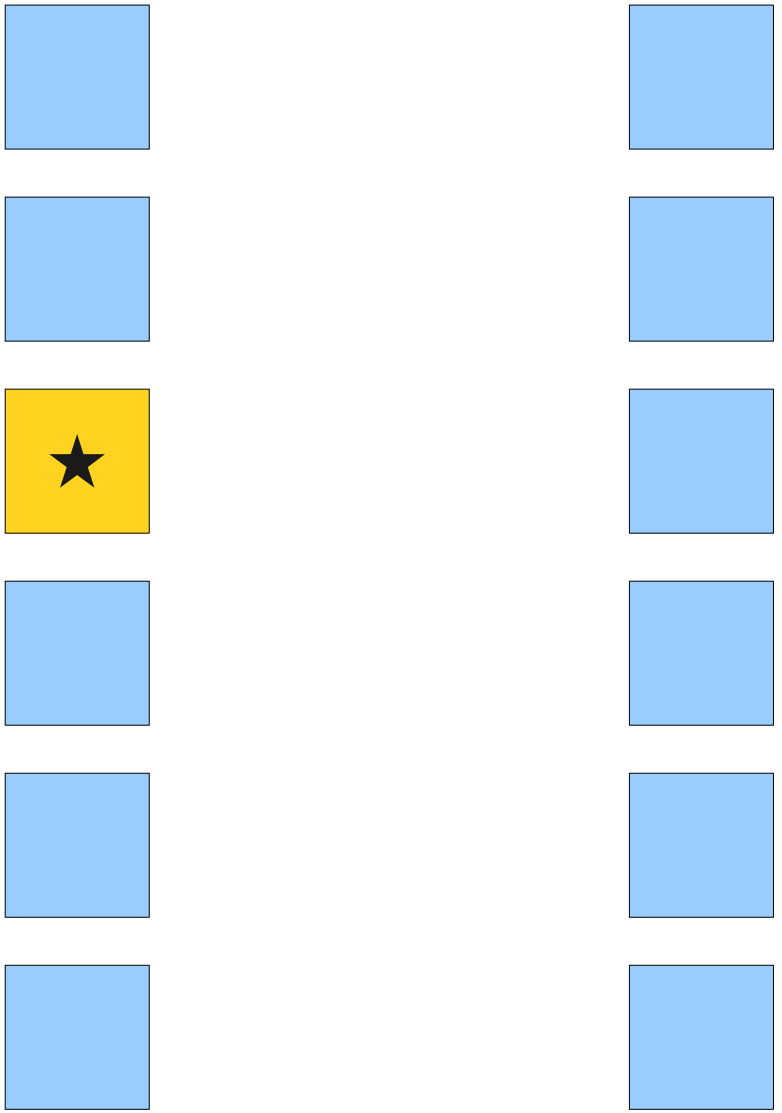
The Result

- **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!

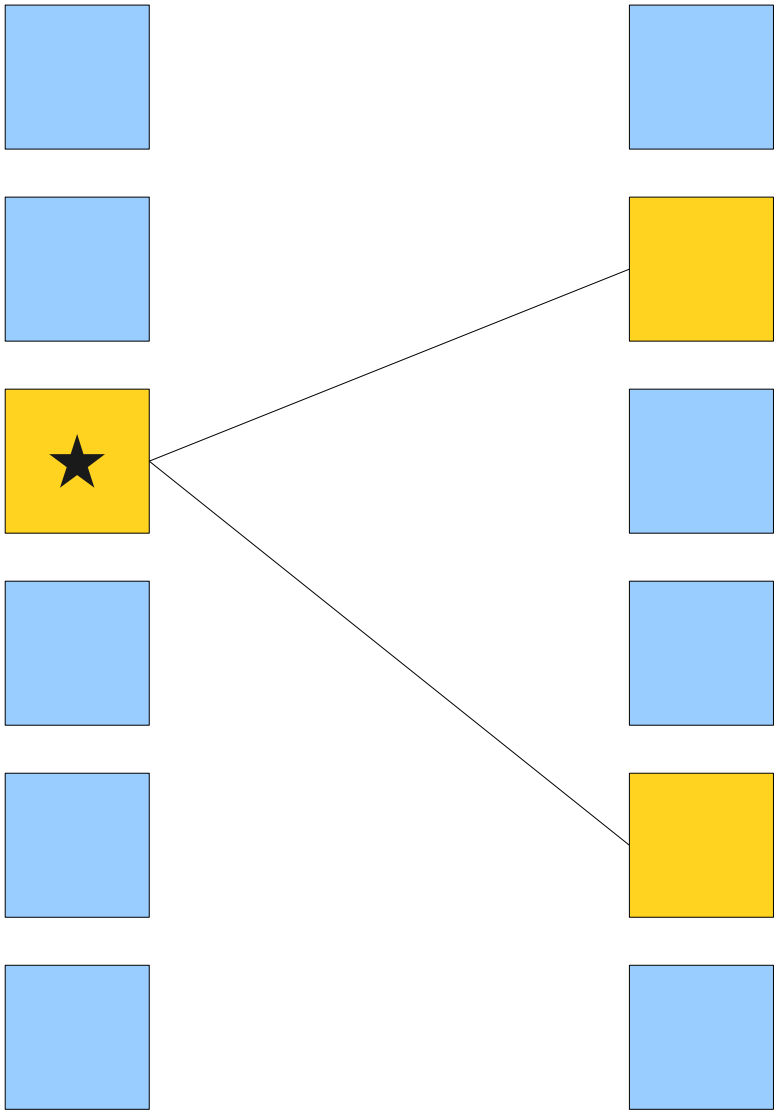
Sizing a Connected Component



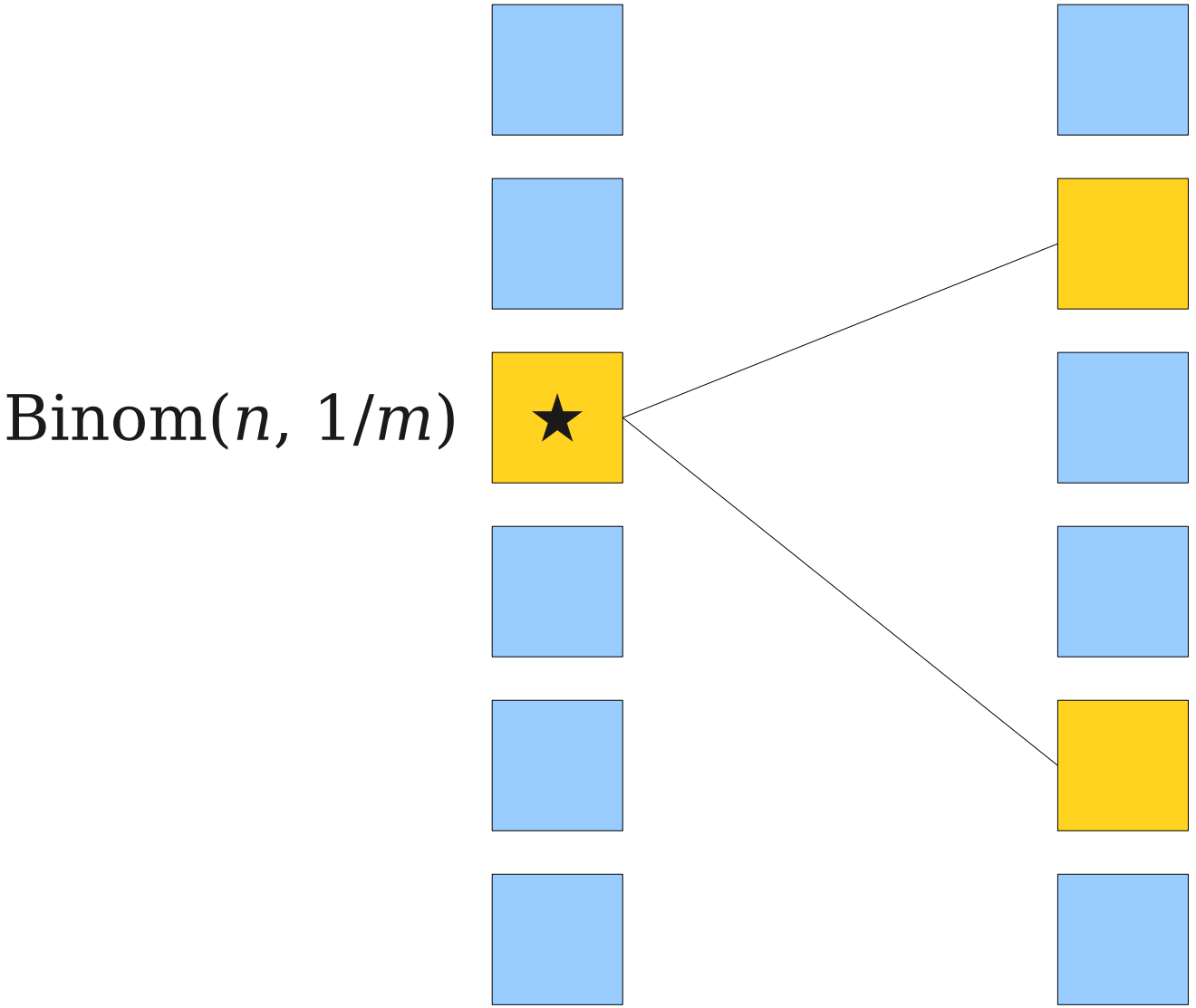
Sizing a Connected Component



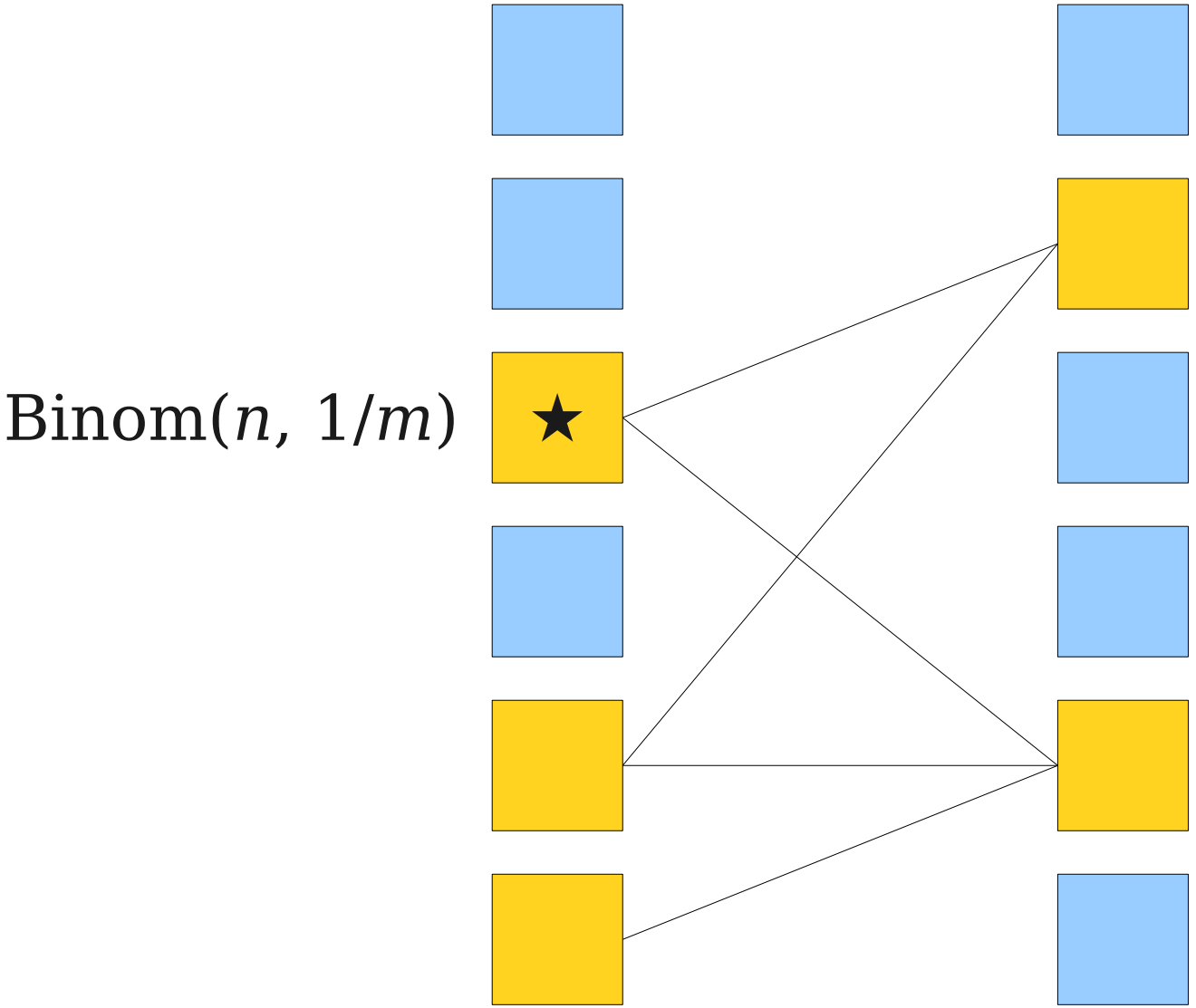
Sizing a Connected Component



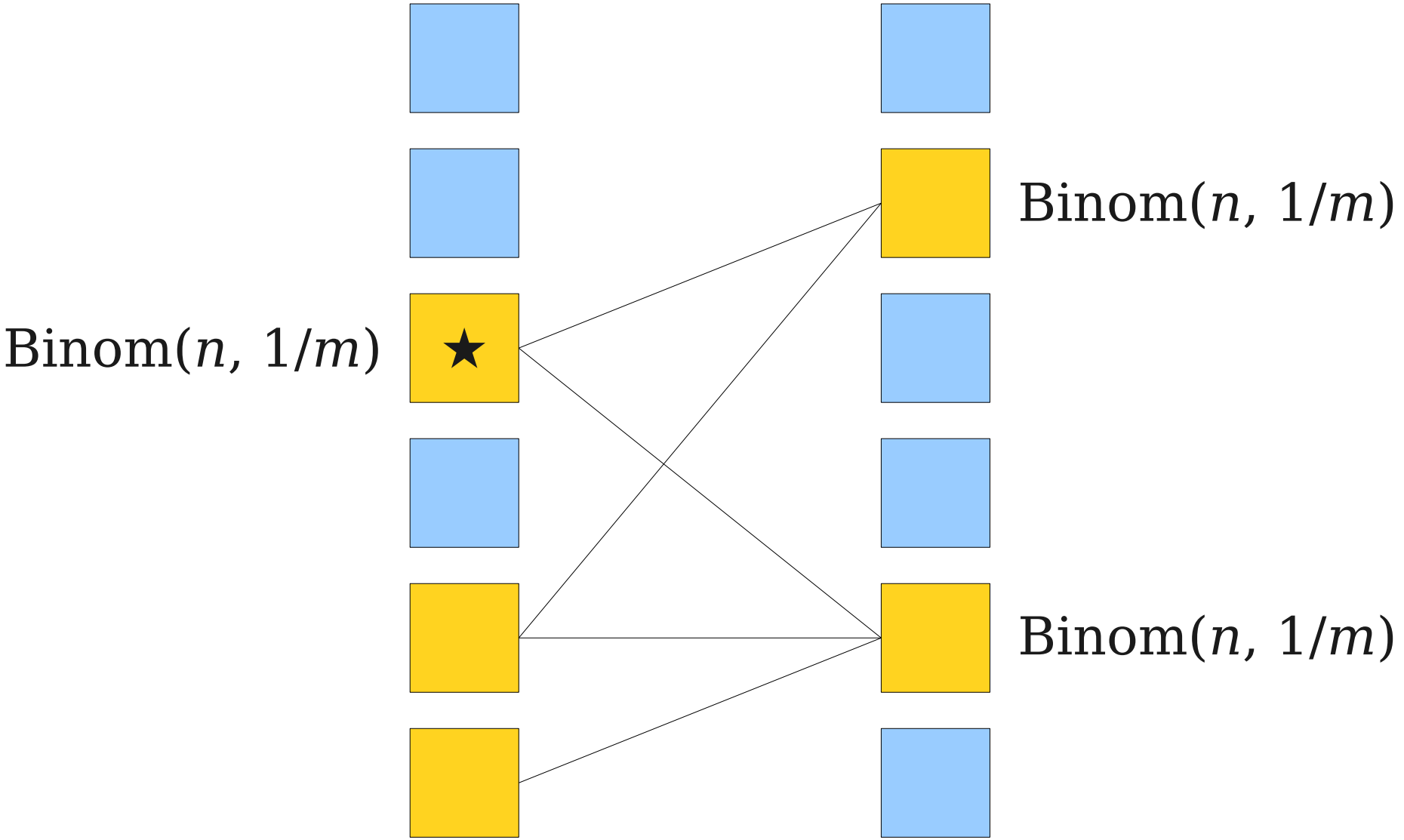
Sizing a Connected Component



Sizing a Connected Component

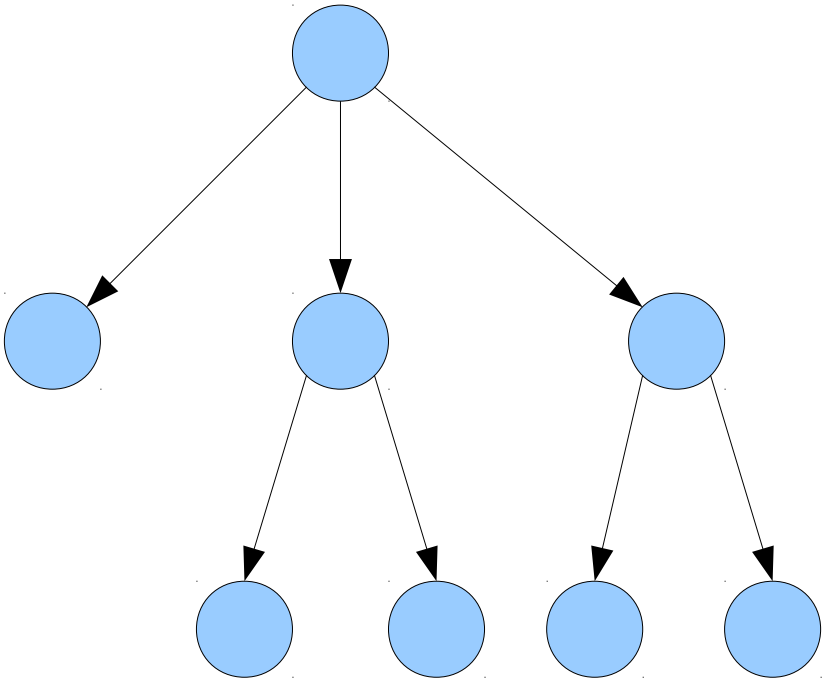


Sizing a Connected Component



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:** $E[\xi]$ must be less than 1.

Paris, Lauvour & Morin, dess. Imp. Lemercier & Cie, Paris February 1st 1868.

GAZETTE OF FASHION

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 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: Problem Set 7. ■

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

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 $\xi \sim \text{Binom}(n, 1 / m)$.

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

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

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

$$\text{E}[X] = \text{E}[\sum_{i=0}^{\infty} X_i]$$

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

$$\mathrm{E}[X] = \mathrm{E}[\sum_{i=0}^{\infty} X_i] = \sum_{i=0}^{\infty} \mathrm{E}[X_i]$$

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

$$\mathrm{E}[X] = \mathrm{E}[\sum_{i=0}^{\infty} X_i] = \sum_{i=0}^{\infty} \mathrm{E}[X_i] = \sum_{i=0}^{\infty} \mathrm{E}[\xi]^i$$

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

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

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

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

Note that $\mathrm{E}[\xi] = n / m \leq 1 / 1 + \varepsilon$, so

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

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

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

Note that $\mathrm{E}[\xi] = n / m \leq 1 / 1 + \varepsilon$, so

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

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

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

Note that $\mathrm{E}[\xi] = n / m \leq 1 / 1 + \varepsilon$, so

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

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

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

Note that $\mathrm{E}[\xi] = n / m \leq 1 / 1 + \varepsilon$, so

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

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

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

Note that $\mathrm{E}[\xi] = n / m \leq 1 / 1 + \varepsilon$, so

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

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

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

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

Note that $\mathrm{E}[\xi] = n / m \leq 1 / 1 + \varepsilon$, so

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

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, where we 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 $O(1 + \varepsilon + \varepsilon^{-1})$.

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.

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 $O(1 + \varepsilon + \varepsilon^{-1})$.

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 $O(m + n\varepsilon^{-1})$.

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

$$O(1 + \varepsilon + \varepsilon^{-1}) + O(1 + n\varepsilon^{-1} / m)$$

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

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

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. ■

Finishing it Off: **High Confidence**

High Confidence

- We've shown that the amortized, expected cost of an insertion into a cuckoo hash table is $O(1 + \varepsilon + \varepsilon^{-1})$.
- How likely are we to actually get this runtime?
- **Claim:** With probability $1 - O(1 / m)$, any sequence of n insertions will run in amortized time $O(1 + \varepsilon + \varepsilon^{-1})$.

Two Components

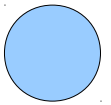
- There are two ways that the insertion time will not be amortized $O(1 + \varepsilon + \varepsilon^{-1})$:
  - We introduce a complex CC into the cuckoo graph. This has probability $O(1 / m)$.
  - We have a large CC whose size is $\omega(1 + \varepsilon^{-1})$.
- We'll show that this second case happens with probability $o(1 / m)$, so the probability of a fast insertion is $O(1 / m)$.

A Tricky Question

- **Question:** Given a fixed CC in the cuckoo hash table, what is the probability that this CC contains at least k nodes?
- Another way of interpreting this question is as the *survival probability* of our subcritical Galton-Watson process: what is the probability that the process will continue for at least k steps?

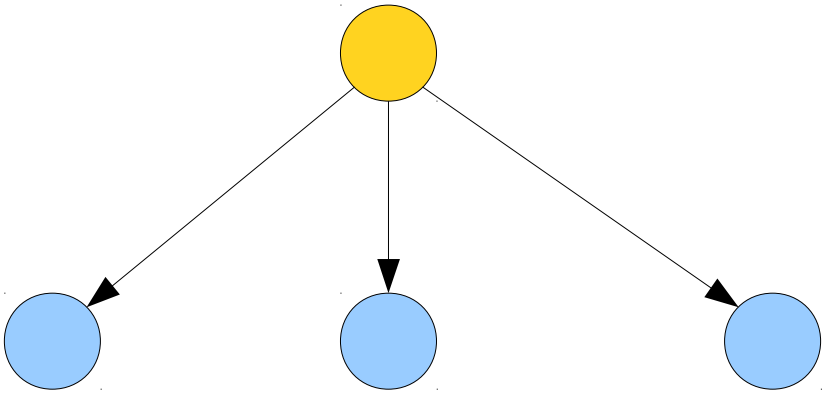
The Survival Probability

- Imagine the expansion of the tree structure.
- Call a node *spent* if we've already processed it to add children and *unspent* otherwise.
- **Claim:** This process continues for more than k steps if after k nodes are spent, there is at least one unspent node.



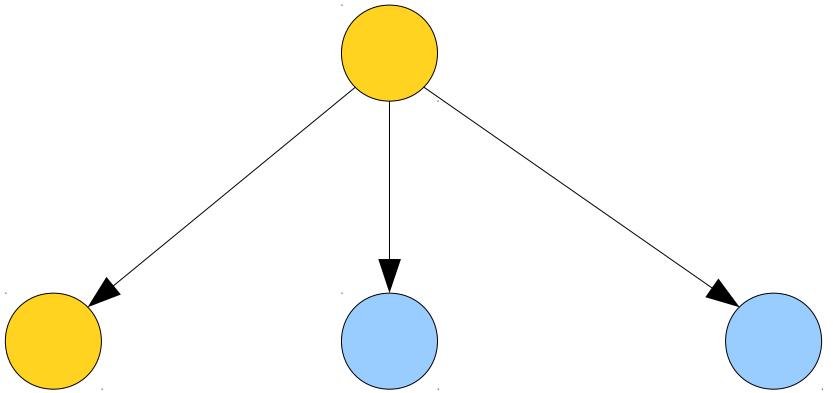
The Survival Probability

- Imagine the expansion of the tree structure.
- Call a node *spent* if we've already processed it to add children and *unspent* otherwise.
- **Claim:** This process continues for more than k steps if after k nodes are spent, there is at least one unspent node.



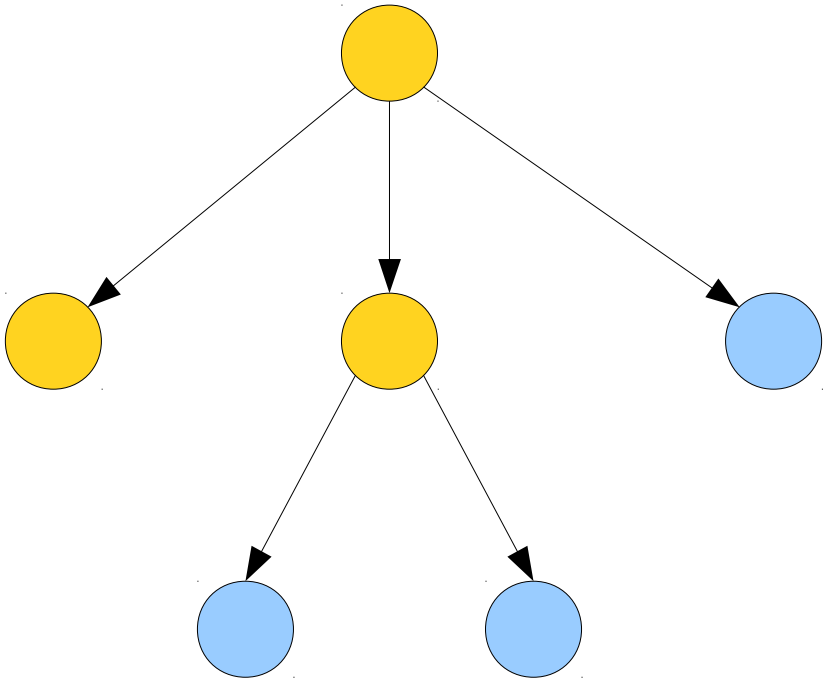
The Survival Probability

- Imagine the expansion of the tree structure.
- Call a node *spent* if we've already processed it to add children and *unspent* otherwise.
- **Claim:** This process continues for more than k steps if after k nodes are spent, there is at least one unspent node.



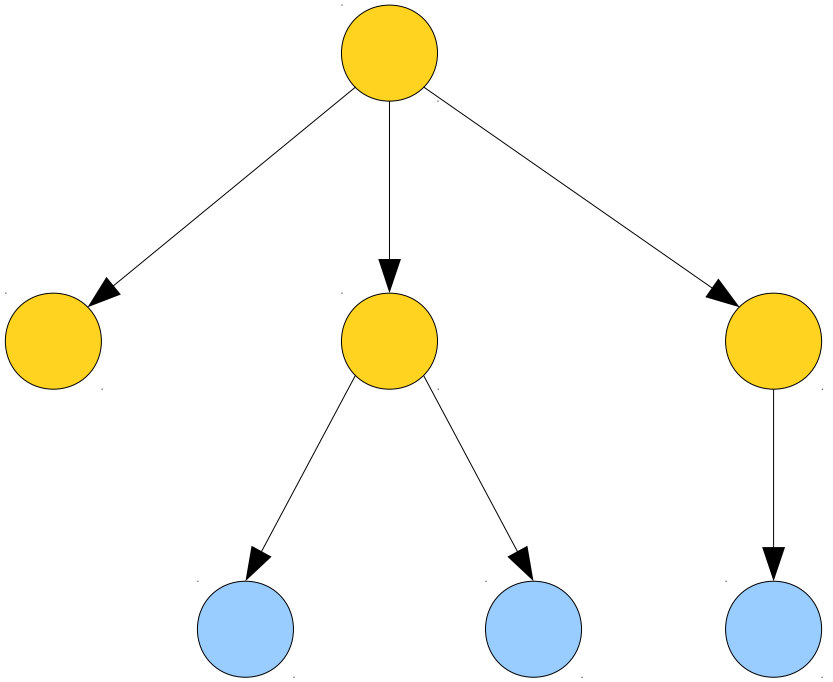
The Survival Probability

- Imagine the expansion of the tree structure.
- Call a node *spent* if we've already processed it to add children and *unspent* otherwise.
- **Claim:** This process continues for more than k steps if after k nodes are spent, there is at least one unspent node.



The Survival Probability

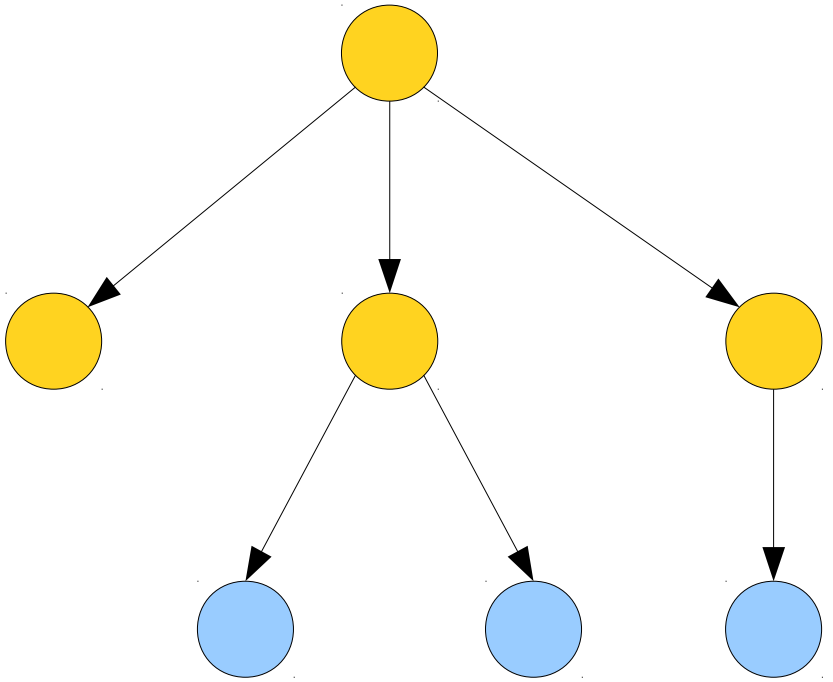
- Imagine the expansion of the tree structure.
- Call a node *spent* if we've already processed it to add children and *unspent* otherwise.
- **Claim:** This process continues for more than k steps if after k nodes are spent, there is at least one unspent node.



The Survival Probability

- Start by making the root node unspent.
- Every spent node increases unspent nodes by Y_i – 1, where $Y_i \sim \text{Binom}(n, 1 / m)$.
- The process stops after k total nodes are spent if

$$1 + \sum_{i=1}^{k} (Y_i - 1) < 1$$



The Survival Probability

- **Claim:** The probability that the process continues after k nodes have been spent is upper-bounded by

$$\Pr\left[1+\sum_{i=1}^{k}\left(Y_i-1\right)\geq 1\right]$$

- This is not a tight bound. It includes cases where the number of nodes becomes negative and then increases.

The Survival Probability

- Simplifying:

$$\Pr\left[1+\sum_{i=1}^{k}\left(Y_i-1\right)\geq 1\right] = \Pr\left[\sum_{i=1}^{k}\left(Y_i-1\right)\geq 0\right]$$

The Survival Probability

- Simplifying:

$$\Pr\left[1 + \sum_{i=1}^{k} (Y_i - 1) \geq 1\right] = \Pr\left[\sum_{i=1}^{k} (Y_i - 1) \geq 0\right]$$
$$= \Pr\left[\sum_{i=1}^{k} Y_i \geq k\right]$$

- Notice that this sum is the sum of k i.i.d. variables drawn from a Binom(n, 1 / m) distribution.
- Therefore, it itself is distributed according to a Binom(kn, 1 / m) distribution.

The Survival Probability

- Let $X \sim \text{Binom}(nk, 1 / m)$.
- The probability that a CC contains more than k nodes is then given by

$$\Pr\left[X \geq k\right]$$

- Our goal now is to obtain a bound on this probability.
- To do so, we'll need to introduce another tail inequality.

Hoeffding's Inequality

- **Hoeffding's inequality** states, among other things, that if $X \sim \text{Binom}(n, p)$, then

$$\Pr\left[X \geq np+\delta\right] \leq e^{-2\delta^2 n}$$

Hoeffding's Inequality

- **Hoeffding's inequality** states, among other things, that if $X \sim \text{Binom}(n, p)$, then

$$\Pr\left[X \geq np + \delta\right] \leq e^{-2\delta^2 n}$$

- In our case, $Y \sim \text{Binom}(nk, 1 / m)$ and we want

$$\Pr\left[Y \geq k\right]$$

Hoeffding's Inequality

- **Hoeffding's inequality** states, among other things, that if $X \sim \text{Binom}(n, p)$, then

$$\Pr\left[X \geq np + \delta\right] \leq e^{-2\delta^2 n}$$

- In our case, $Y \sim \text{Binom}(nk, 1 / m)$ and we want

$$\Pr\left[Y \geq k\right]$$

- Rewriting:

$$\Pr\left[Y \geq k\right] = \Pr\left[Y \geq \frac{nk}{m} + k - \frac{nk}{m}\right]$$

Hoeffding's Inequality

- **Hoeffding's inequality** states, among other things, that if $X \sim \text{Binom}(n, p)$, then

$$\Pr\left[X \geq np + \delta\right] \leq e^{-2\delta^2 n}$$

- In our case, $Y \sim \text{Binom}(nk, 1 / m)$ and we want

$$\Pr\left[Y \geq k\right]$$

- Rewriting:

$$\begin{aligned}\Pr\left[Y \geq k\right] &= \Pr\left[Y \geq \frac{nk}{m} + k - \frac{nk}{m}\right] \\ &= \Pr\left[Y \geq \frac{nk}{m} + k\left(1 - \frac{n}{m}\right)\right]\end{aligned}$$

Hoeffding's Inequality

- **Hoeffding's inequality** states, among other things, that if $X \sim \text{Binom}(n, p)$, then

$$\Pr\left[X \geq np + \delta\right] \leq e^{-2\delta^2 n}$$

- In our case, $Y \sim \text{Binom}(nk, 1 / m)$ and we want

$$\Pr\left[Y \geq k\right]$$

- Rewriting:

$$\begin{aligned}
\Pr\left[Y \geq k\right] &= \Pr\left[Y \geq \frac{nk}{m} + k - \frac{nk}{m}\right] \\
&= \Pr\left[Y \geq \frac{nk}{m} + k\left(1 - \frac{n}{m}\right)\right] \\
&\leq e^{-2k^2\left(1-\frac{n}{m}\right)^2 n}
\end{aligned}$$

Hoeffding's Inequality

- We now have

$$\Pr\left[Y \geq k\right] \leq e^{-2k^2\left(1-\frac{n}{m}\right)^2 n}$$

- Notice that

$$1-\frac{n}{m} \geq 1-\frac{n}{(1+\varepsilon)n} = \frac{\varepsilon}{1+\varepsilon} = \mathrm{O}(1)$$

- Therefore,

$$\Pr\left[Y \geq k\right] \leq e^{-\mathrm{O}(1)k^2 n}$$

- It is *exponentially unlikely* that we get large connected components!

The Final Analysis

- The previous theorem says

  For any node v in the cuckoo graph, the probability that the CC containing that node has more than k nodes is at most $e^{-n \cdot O(1)}$.

- Using the union bound, we get that the probability that *any* connected component contains k or more nodes is at most $2m\ e^{-n \cdot O(1)}$

- This is $e^{\ln 2m}\ e^{-n \cdot O(1)} = e^{-n \cdot O(1) + \ln 2m} = e^{-O(n)}$.

- Therefore, it is *exponentially unlikely* that any connected component will contain more than O(1) nodes.

The Final Analysis

- ***Theorem:*** For any $\varepsilon > 0$, cuckoo hashing supports the following:
  - Worst-case O(1) lookups and deletions.
  - Expected, amortized $O(1 + \varepsilon + \varepsilon^{-1})$ insertions.
  - Amortized $O(1 + \varepsilon + \varepsilon^{-1})$ time for m insertions with probability $1 - O(1 / m)$.
- And it works really well in practice, too!

Final Thoughts on Cuckoo Hashing

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.
- 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.
- **Stopping time:** Typically, cuckoo hashing triggers a rehash as soon as $C \log n$ elements have been displaced, for some constant C.
- Need to repeat the analysis to show that this addition doesn't cause rehashing with high frequency.

Further Reading

- Many variations on cuckoo hashing have been proposed:
  - If each table bucket can hold k elements each for $k > 1$, the table can become significantly more space efficient without sacrificing much performance.
  - Can set aside a “stash” where elements that would cause cycles can live. This can lead to marked performance increases.
  - Can try to use cuckoo hashing with at most one move by using multiway associative memory devices.

Next Time

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