Cuckoo Hashing

Randomized Data Structures

- Randomization is a powerful tool for improving efficiency and solving problems under seemingly impossible constraints.
- Over the next three lectures, we'll explore a sampler of data structures that give a feel for the breadth of what's out there.
- You can easily spend an entire academic career just exploring this space; take CS265 for more on randomized algorithms!

Where We're Going

- ***Cuckoo Hashing (Today)***
  - Worst-case efficient hashing and deep properties of random graphs.
- ***Frequency Estimation (Next Week)***
  - Counting without counting, and how much randomness is needed to do it.

Outline for Today

- ***Cuckoo Hashing***
  - A simple, fast hashing system with worst-case efficient lookups.
- ***The Erdős–Rényi Model***
  - Randomly-generated graphs and their properties.
- ***Variants on Cuckoo Hashing***
  - Making a good idea even better.

Preliminaries: Hash Tables

Collision Resolution

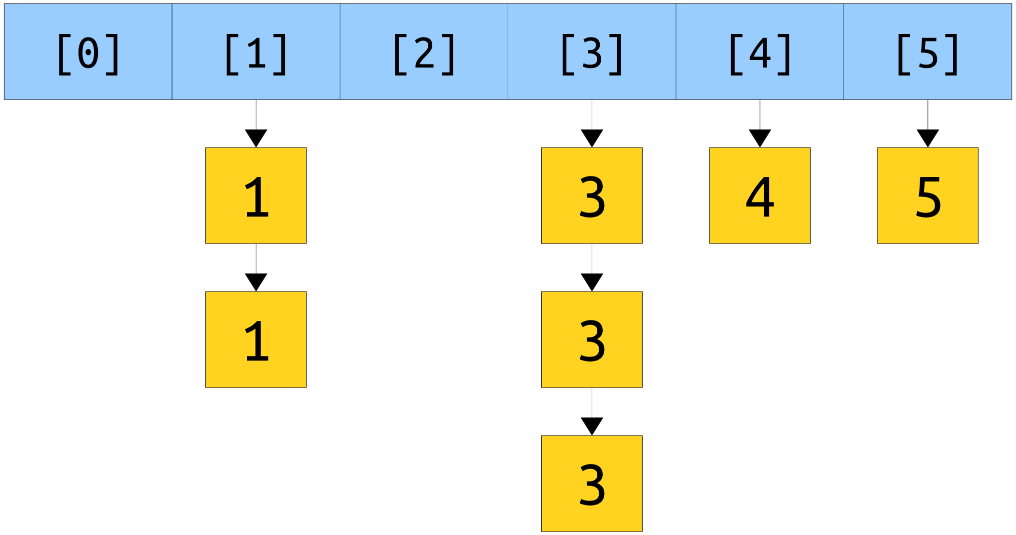
- All hash tables have to deal with hash collisions in some way.
- There are three general ways to do this:

Collision Resolution

- All hash tables have to deal with hash collisions in some way.
- There are three general ways to do this:
  - ***Closed addressing:*** Store all colliding elements in an auxiliary data structure like a linked list or BST. (For example, standard chained hashing.)

Collision Resolution

- All hash tables have to deal with hash collisions in some way.
- There are three general ways to do this:
    - ***Closed addressing:*** Store all colliding elements in an auxiliary data structure like a linked list or BST. (For example, standard chained hashing.)

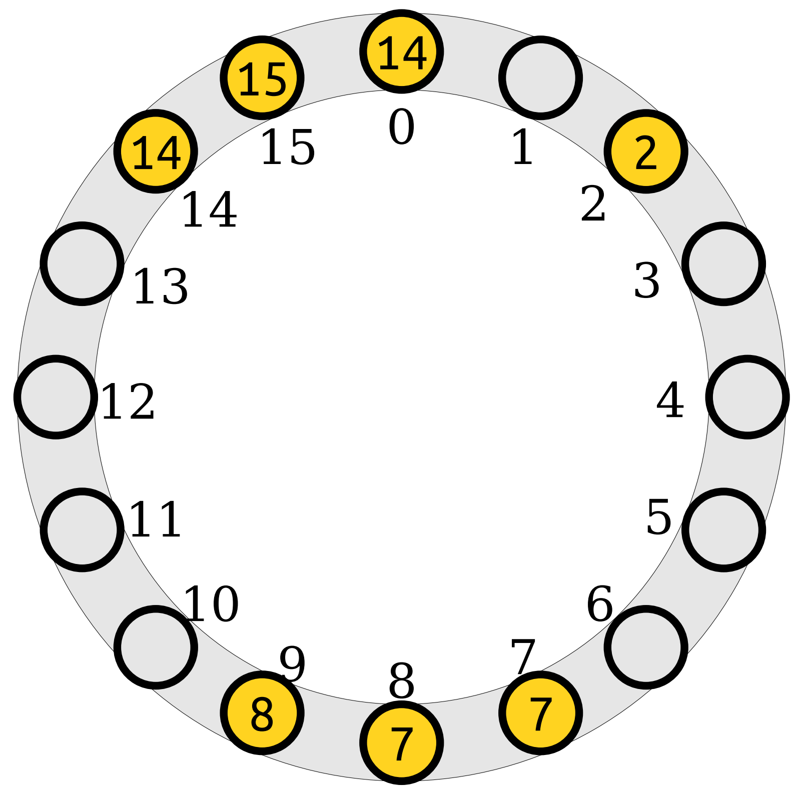


Collision Resolution

- All hash tables have to deal with hash collisions in some way.
- There are three general ways to do this:
  - ***Closed addressing:*** Store all colliding elements in an auxiliary data structure like a linked list or BST. (For example, standard chained hashing.)
  - ***Open addressing:*** Allow elements to overflow out of their target bucket and into other spaces. (For example, linear probing hashing.)

Collision Resolution

- All hash tables have to deal with hash collisions in some way.
- There are three general ways to do this:
  - ***Closed addressing:*** Store all colliding elements in an auxiliary data structure like a linked list or BST. (For example, standard chained hashing.)
  - ***Open addressing:*** Allow elements to overflow out of their target bucket and into other spaces. (For example, linear probing hashing.)



Collision Resolution

- All hash tables have to deal with hash collisions in some way.
- There are three general ways to do this:
  - ***Closed addressing:*** Store all colliding elements in an auxiliary data structure like a linked list or BST. (For example, standard chained hashing.)
  - ***Open addressing:*** Allow elements to overflow out of their target bucket and into other spaces. (For example, linear probing hashing.)
  - ***Perfect hashing:*** Do something clever with multiple hash functions to eliminate collisions.
- What does that last option look like?

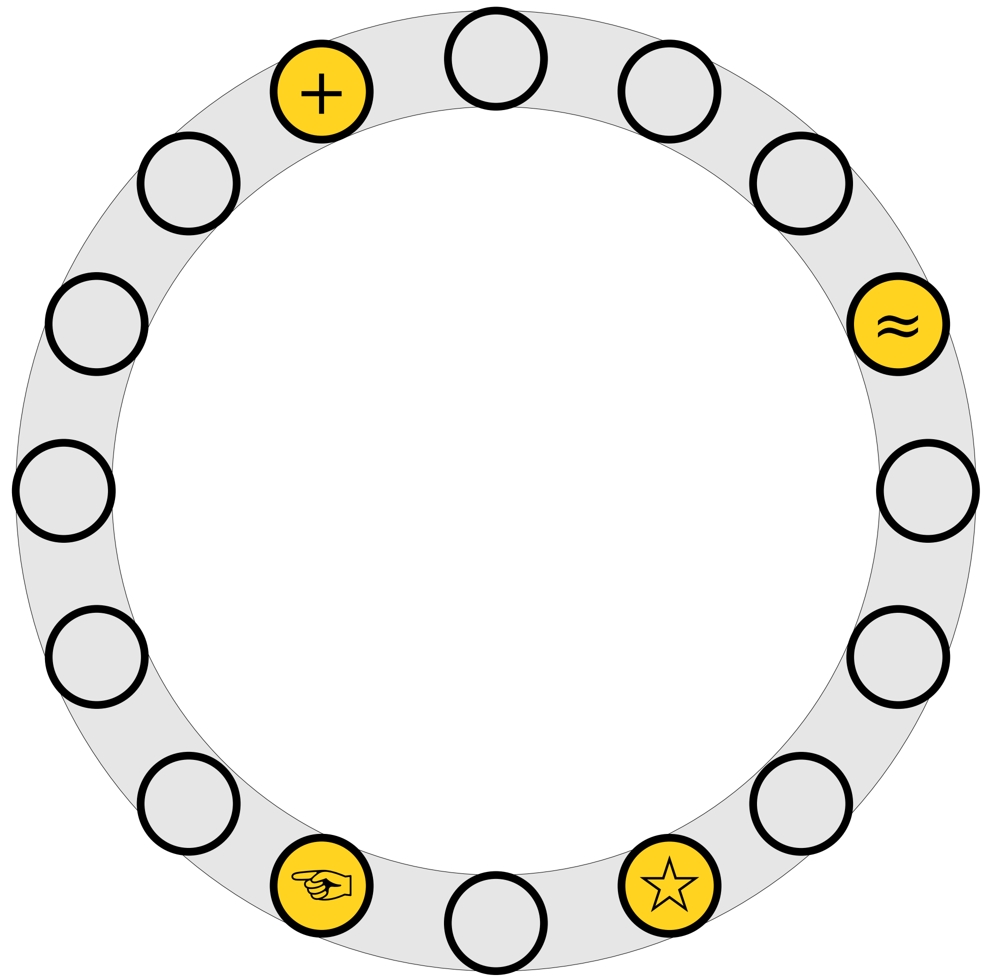
Cuckoo Hashing

Cuckoo Hashing

- Suppose we have a hash table with m slots.
- Unlike a normal hash table, we'll use *two* hash functions. We'll call them h_1 and h_2.
- Each hash function outputs a slot number in the set $\{ 0, 1, 2, \ldots, m - 1 \}$.
- We'll assume that these hash functions are truly random, with one constraint:

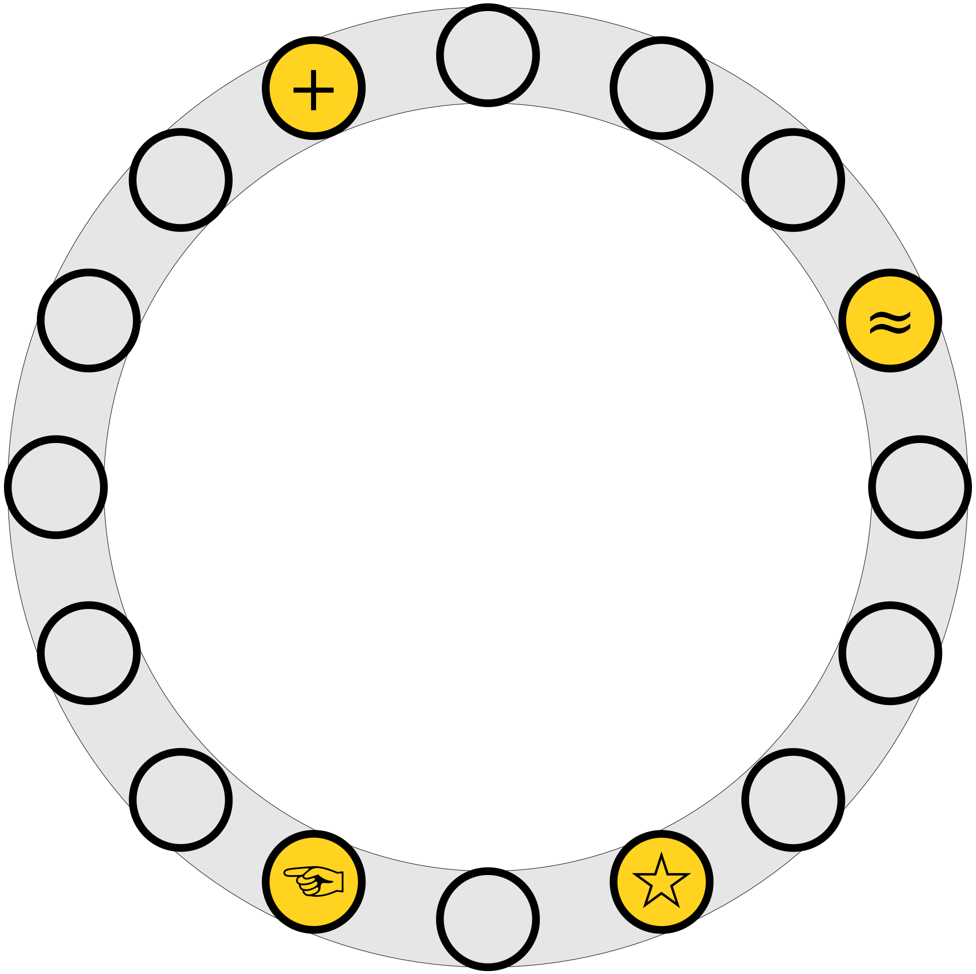
  $\boldsymbol{h_1(x) \neq h_2(x)}$ **for any key** $\boldsymbol{x}$**.**

- This is actually pretty easy to achieve both in theory and in practice – more on that later.



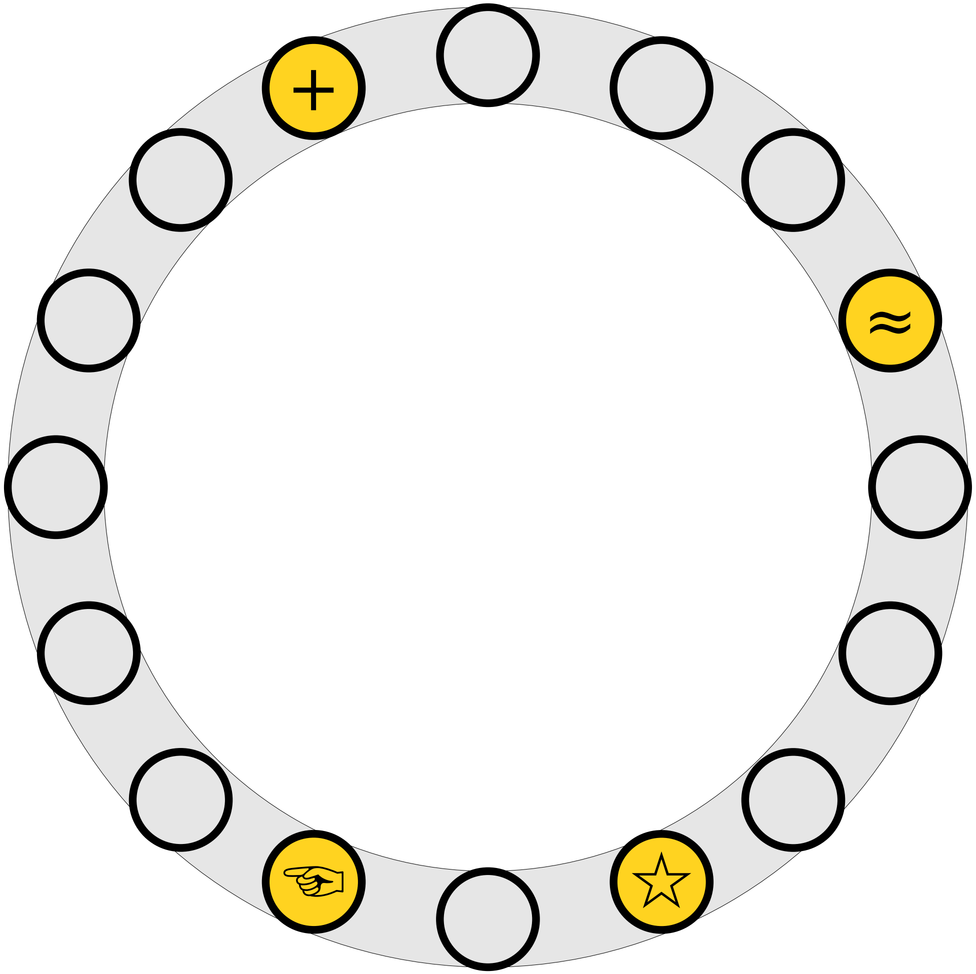
Cuckoo Hashing

- ***The Rule:*** Any item x must either be at position $h_1(x)$ or position $h_2(x)$ in the table.
- Lookups take *worst-case* O(1) time, since only two locations need to be checked.
- Deletions take *worst-case* O(1) time, since only two locations need to be checked.



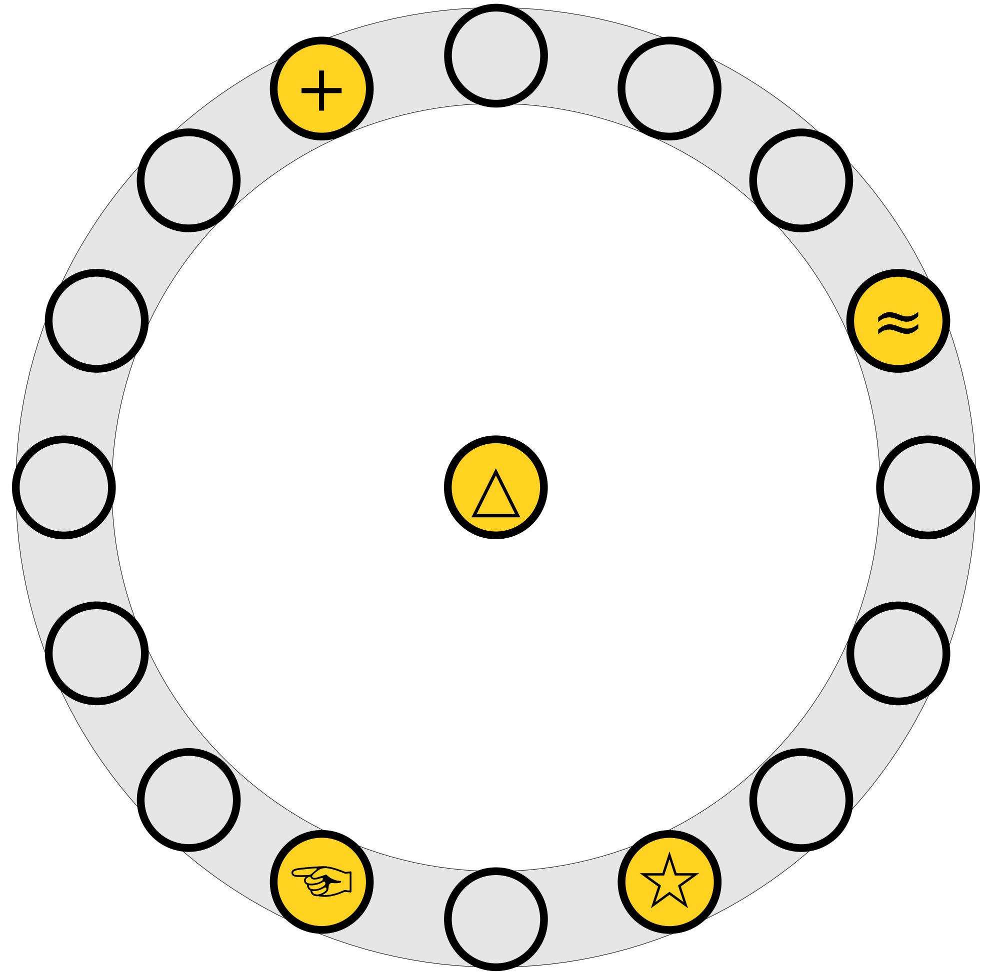
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.



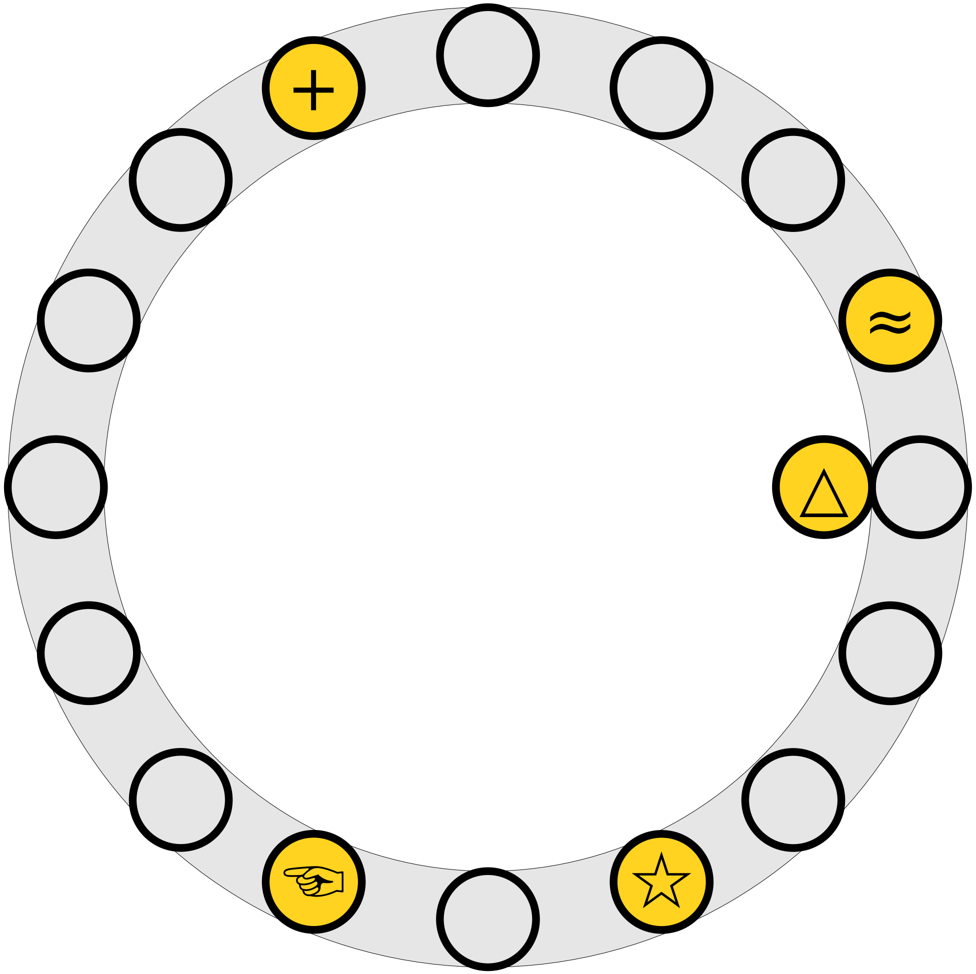
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.



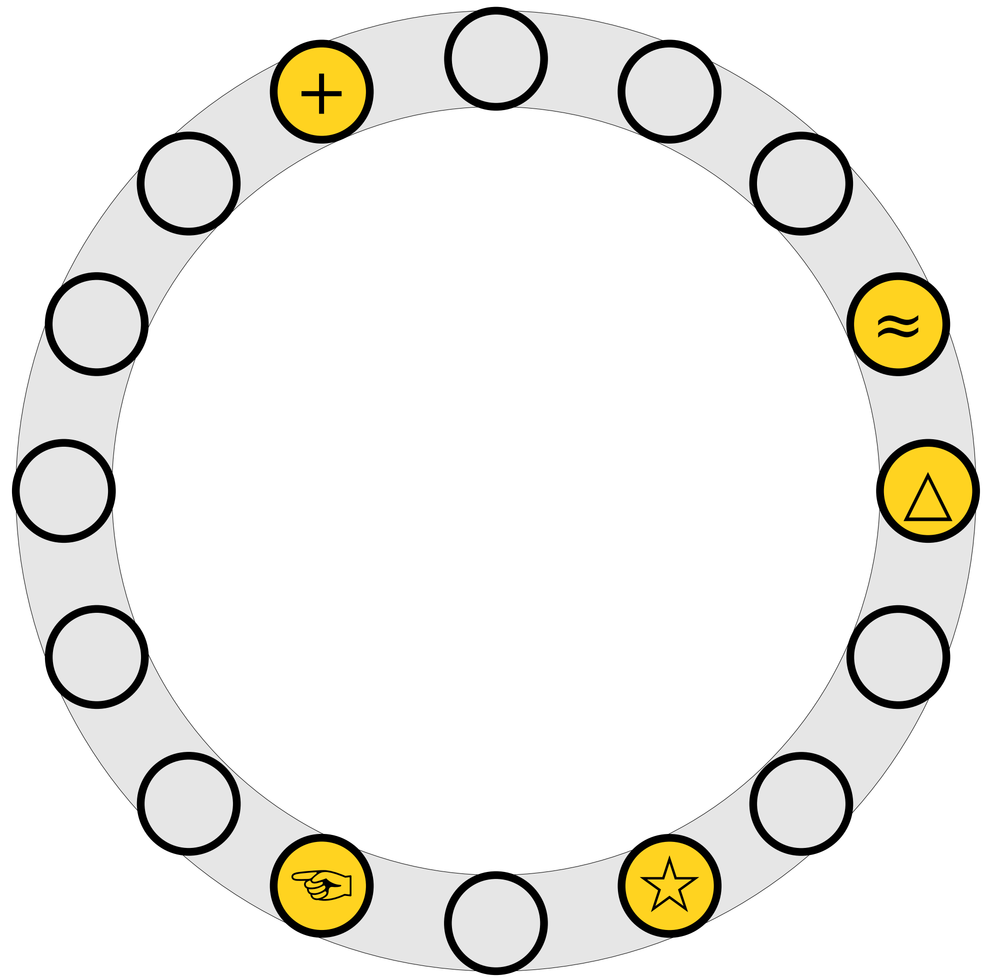
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.



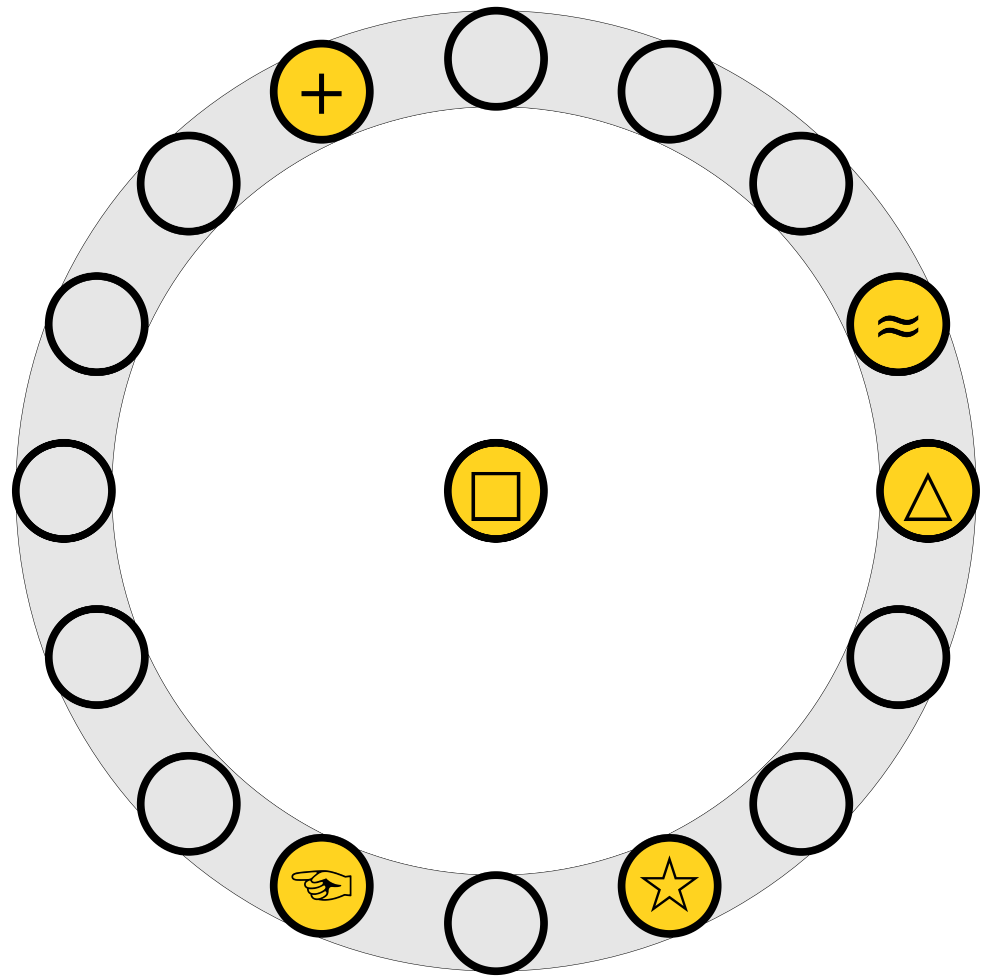
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.



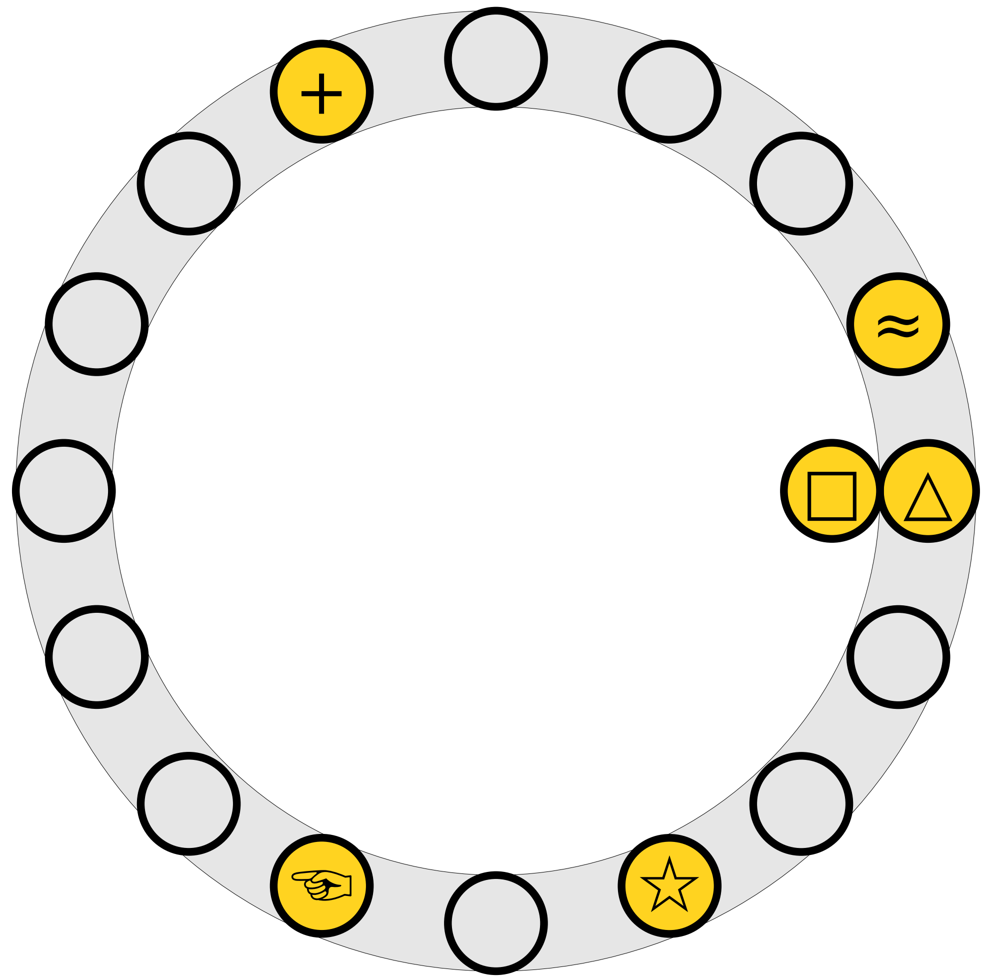
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.



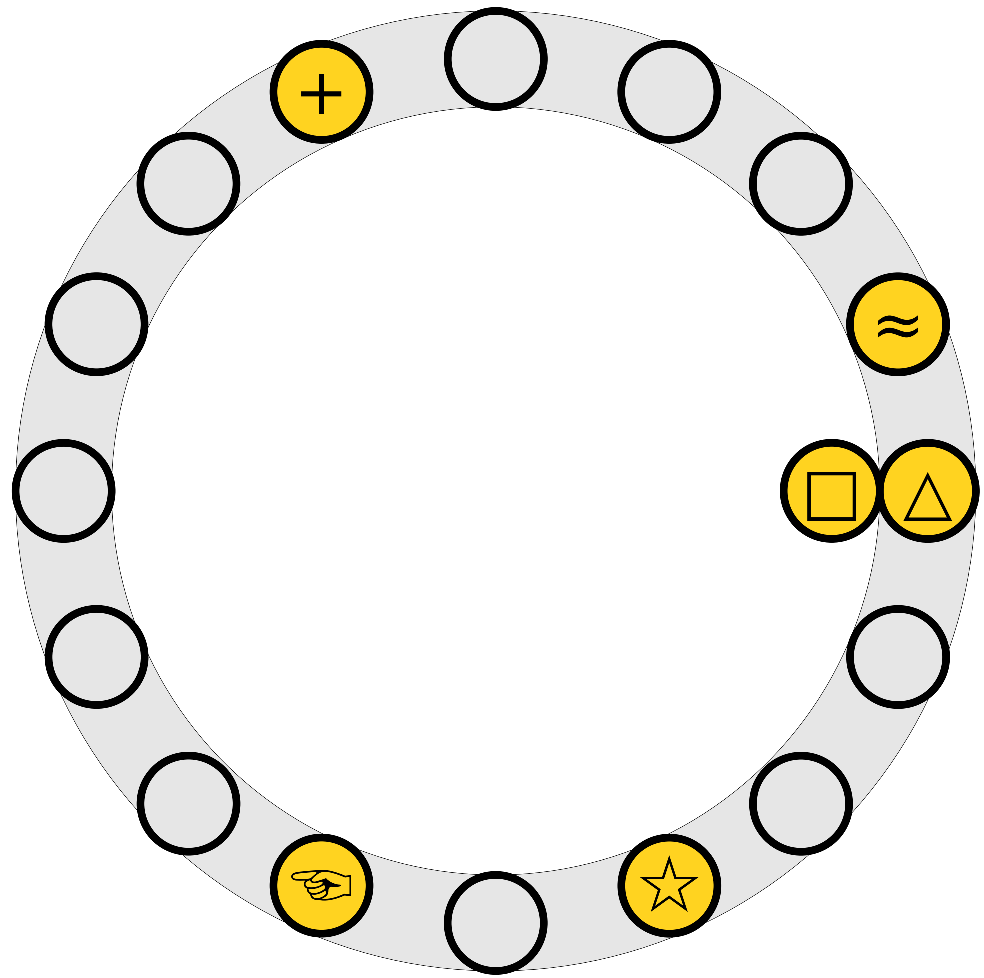
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.



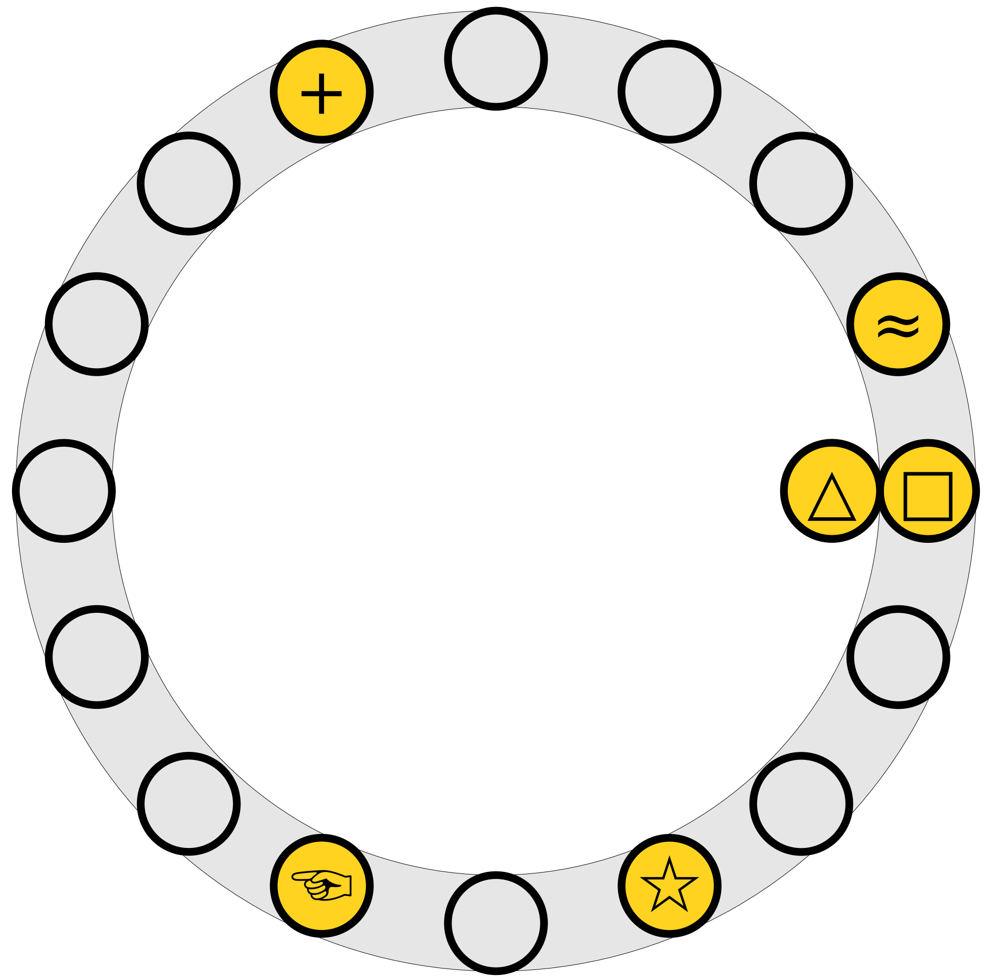
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.
- If that slot is full, kick out the element y that used to be in that slot and try placing it the other slot it can belong to (either $h_1(y)$ or $h_2(y)$).



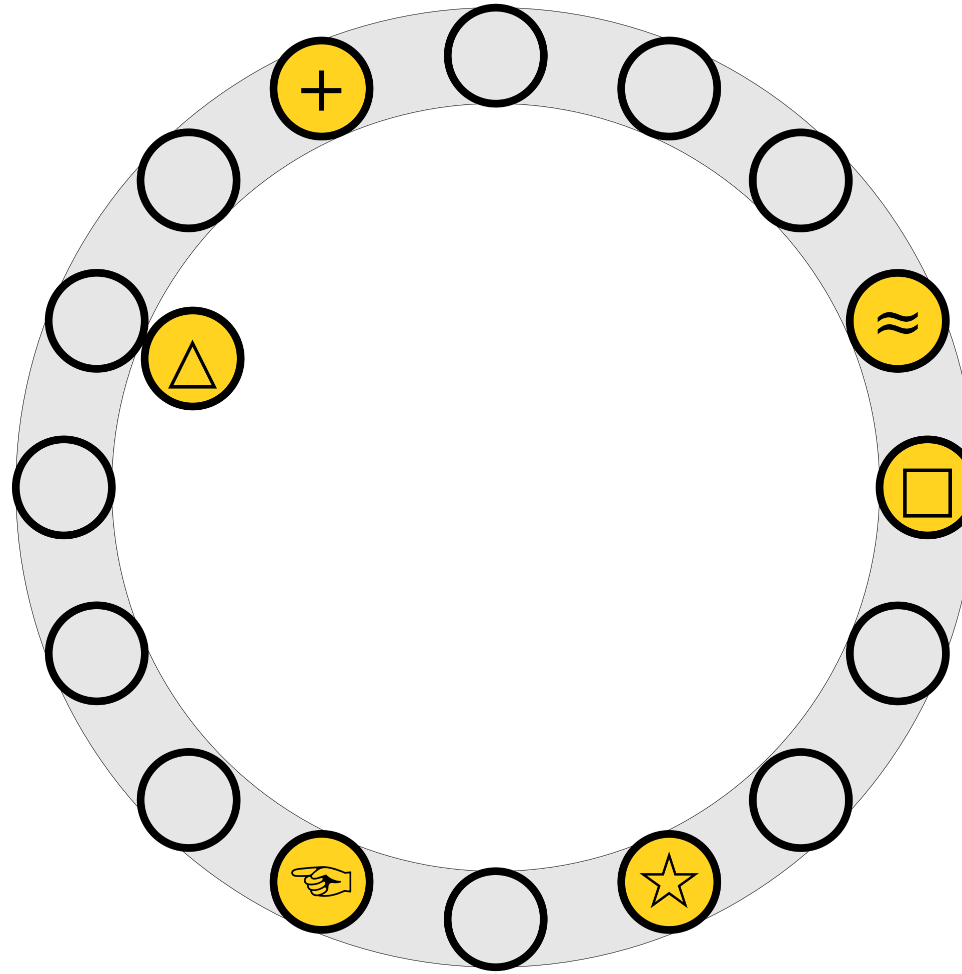
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.
- If that slot is full, kick out the element y that used to be in that slot and try placing it the other slot it can belong to (either $h_1(y)$ or $h_2(y)$).



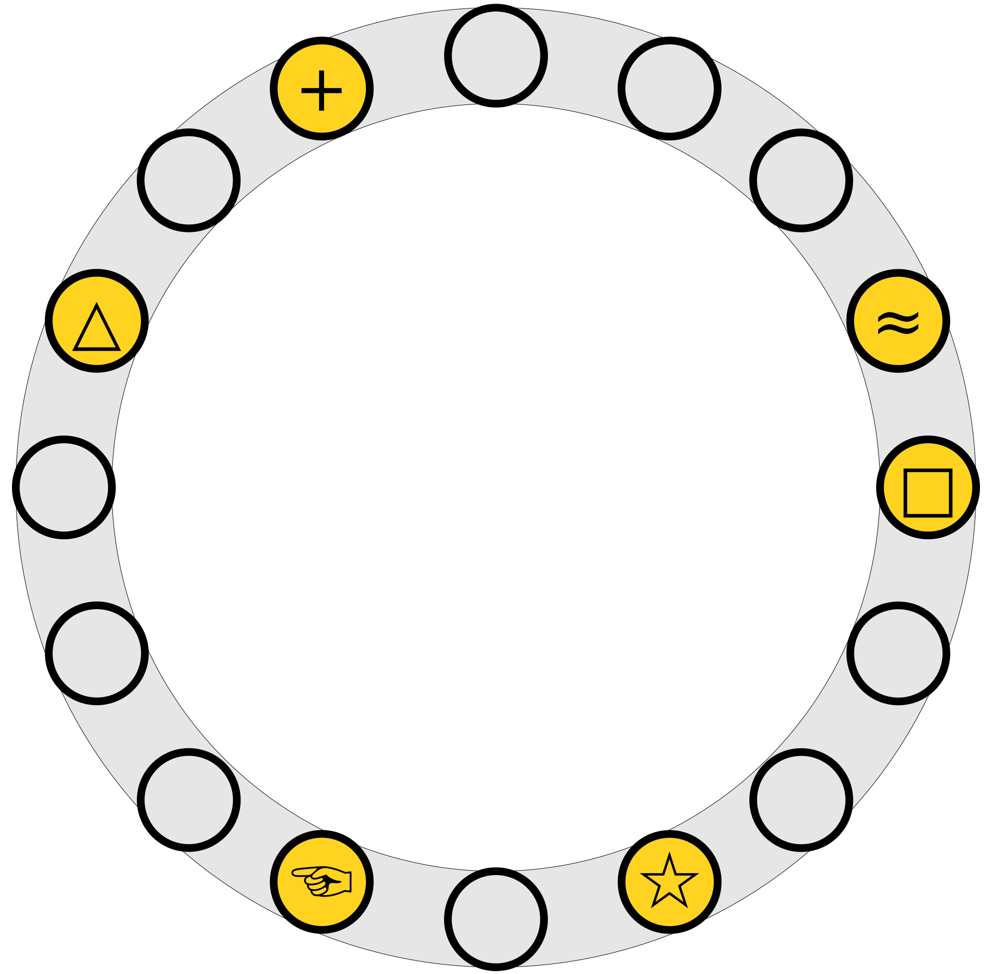
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.
- If that slot is full, kick out the element y that used to be in that slot and try placing it the other slot it can belong to (either $h_1(y)$ or $h_2(y)$).



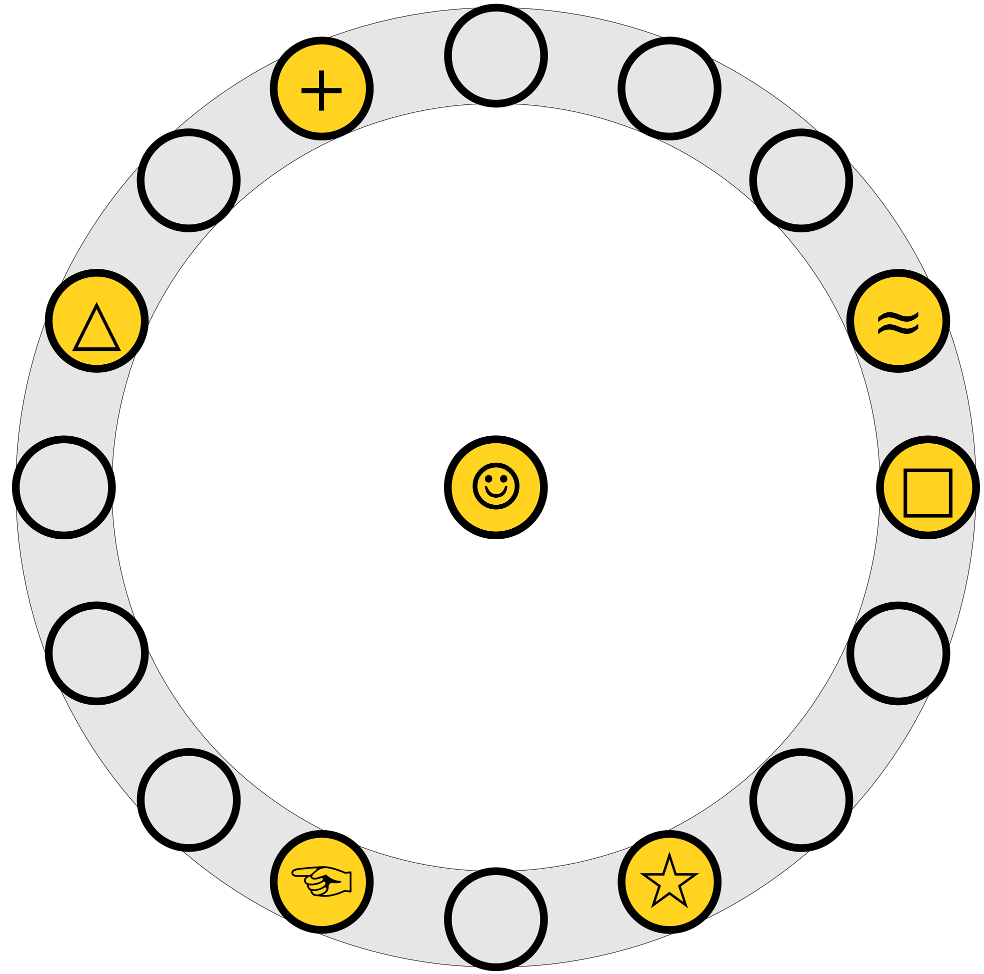
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.
- If that slot is full, kick out the element y that used to be in that slot and try placing it the other slot it can belong to (either $h_1(y)$ or $h_2(y)$).



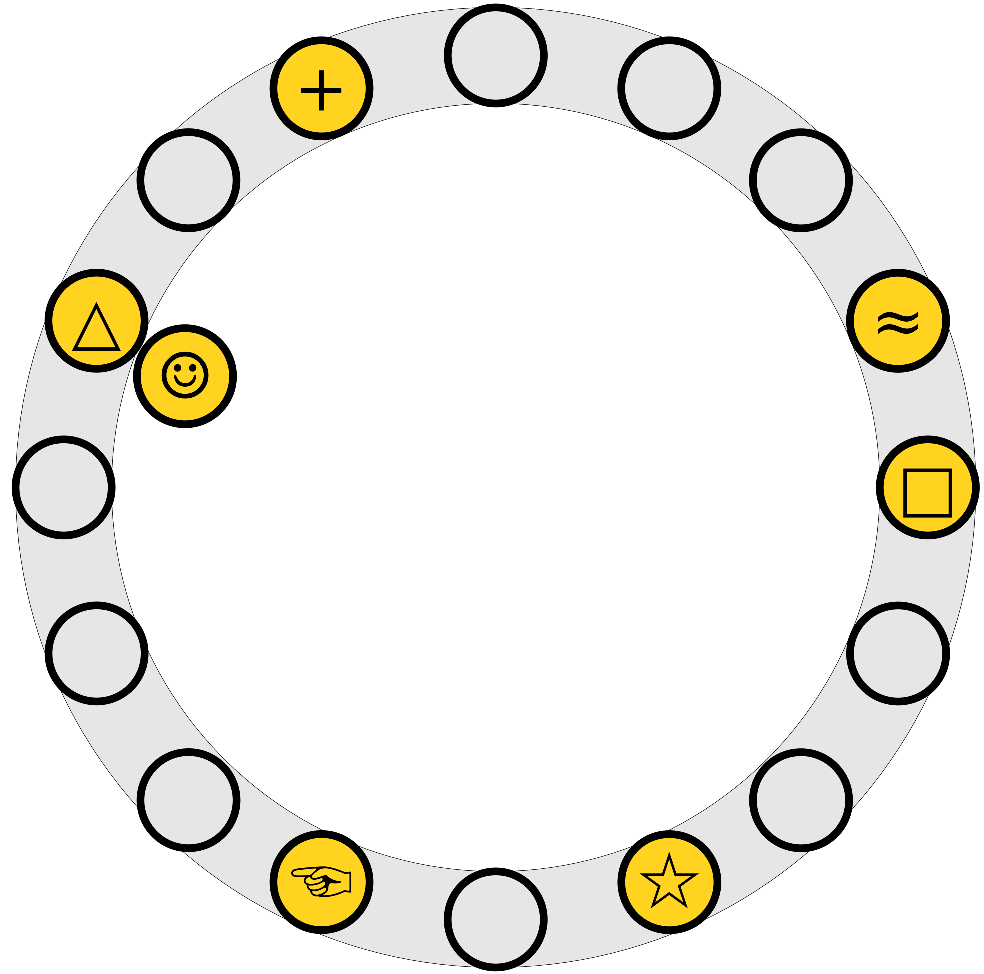
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.
- If that slot is full, kick out the element y that used to be in that slot and try placing it the other slot it can belong to (either $h_1(y)$ or $h_2(y)$).



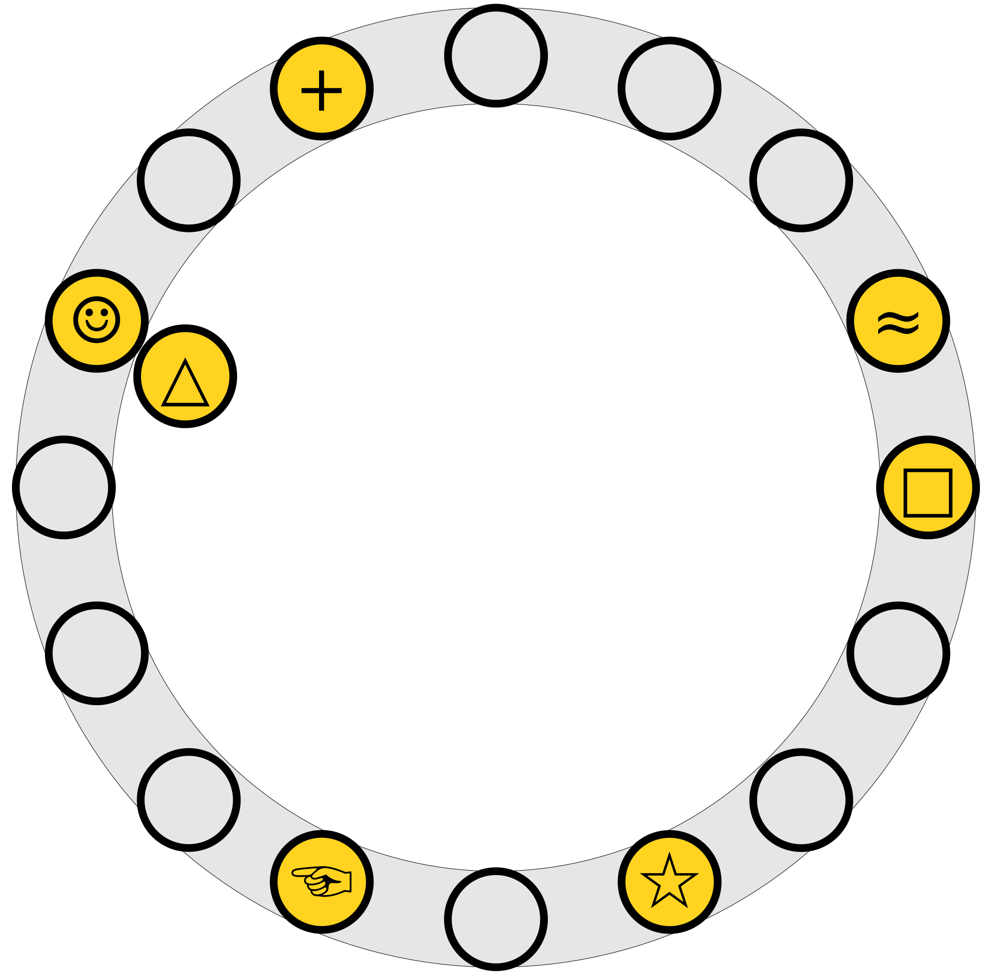
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.
- If that slot is full, kick out the element y that used to be in that slot and try placing it the other slot it can belong to (either $h_1(y)$ or $h_2(y)$).



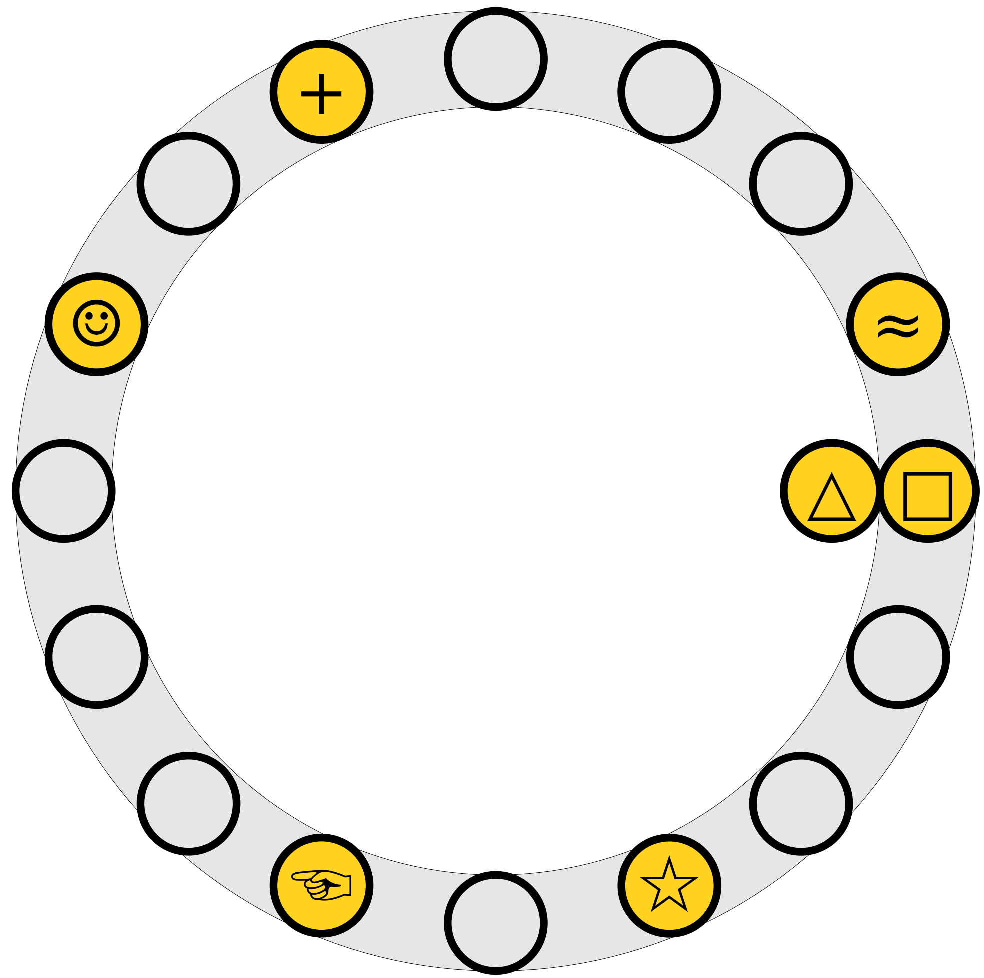
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.
- If that slot is full, kick out the element y that used to be in that slot and try placing it the other slot it can belong to (either $h_1(y)$ or $h_2(y)$).



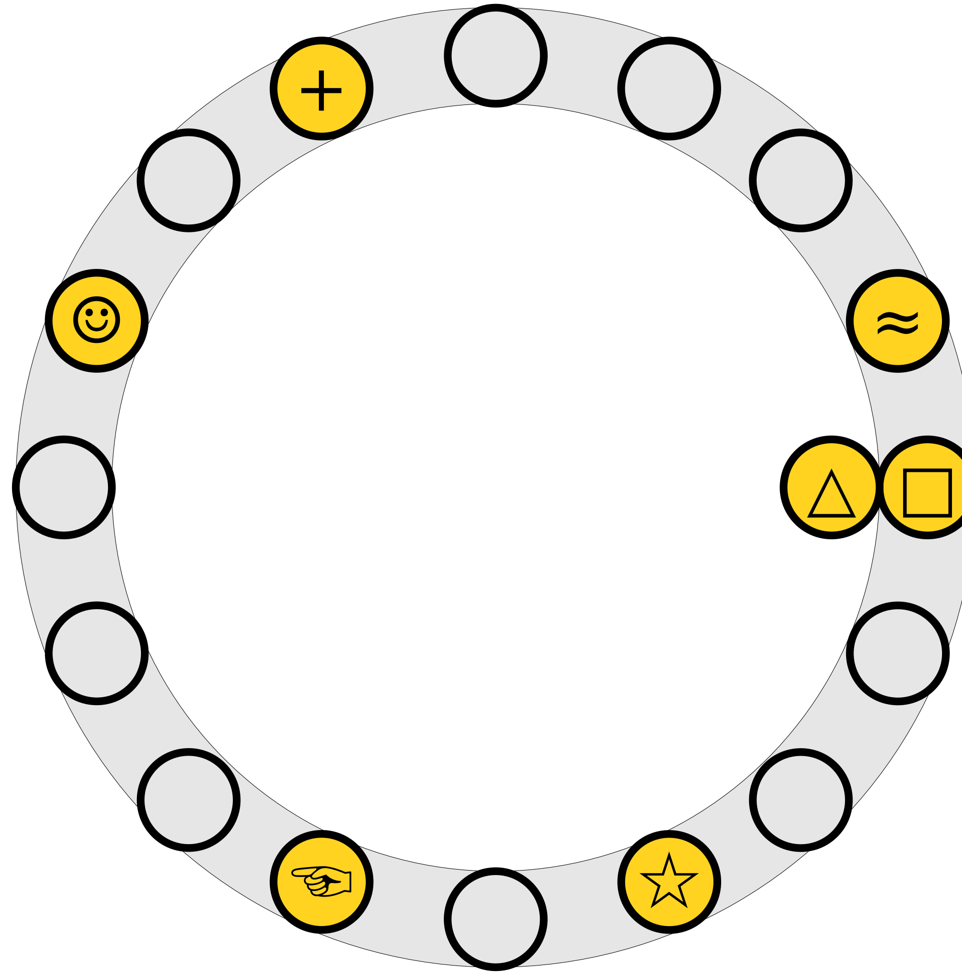
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.
- If that slot is full, kick out the element y that used to be in that slot and try placing it the other slot it can belong to (either $h_1(y)$ or $h_2(y)$).



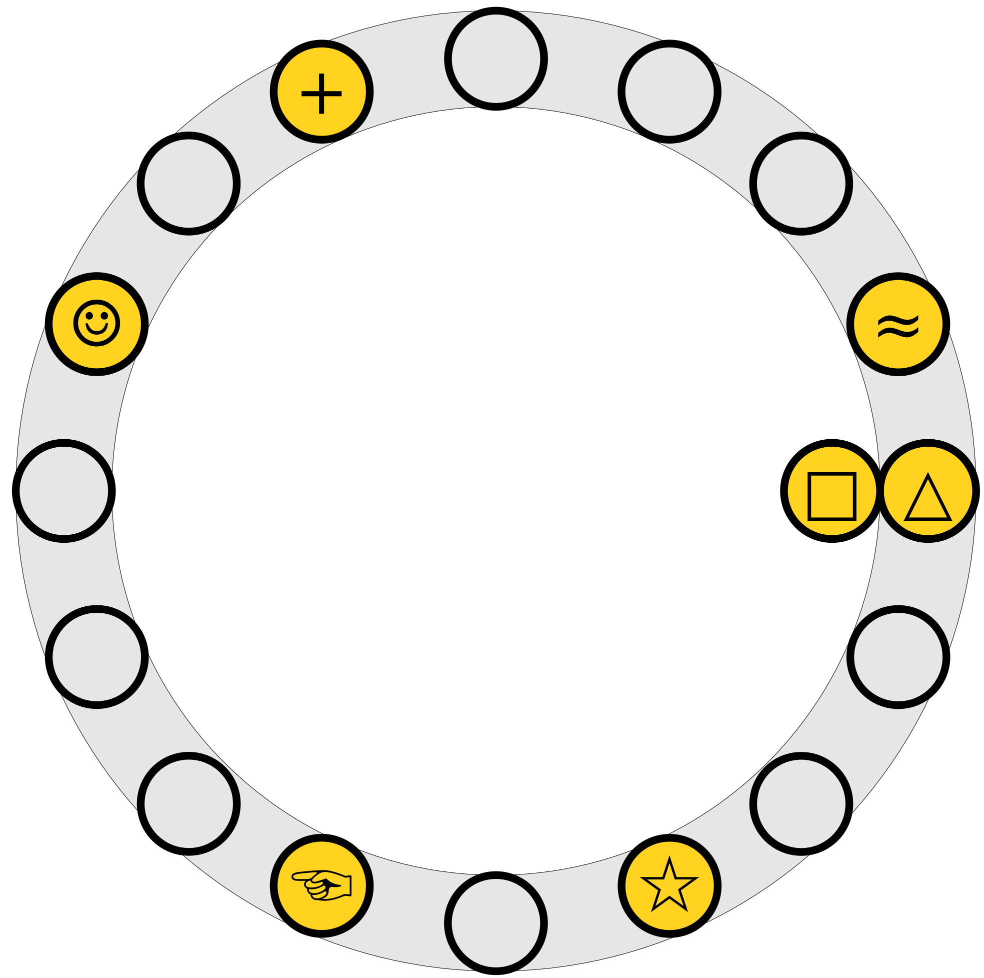
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.
- If that slot is full, kick out the element y that used to be in that slot and try placing it the other slot it can belong to (either $h_1(y)$ or $h_2(y)$).
- Repeat this process until all elements stabilize.



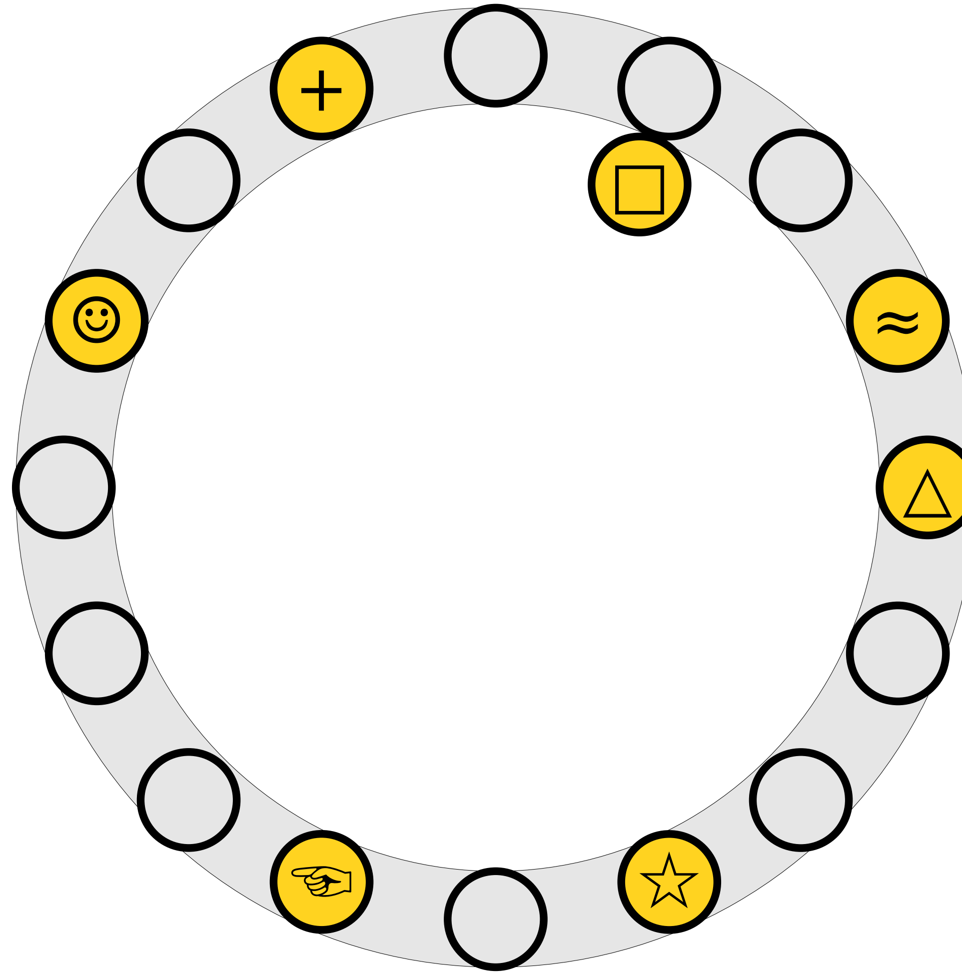
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.
- If that slot is full, kick out the element y that used to be in that slot and try placing it the other slot it can belong to (either $h_1(y)$ or $h_2(y)$).
- Repeat this process until all elements stabilize.



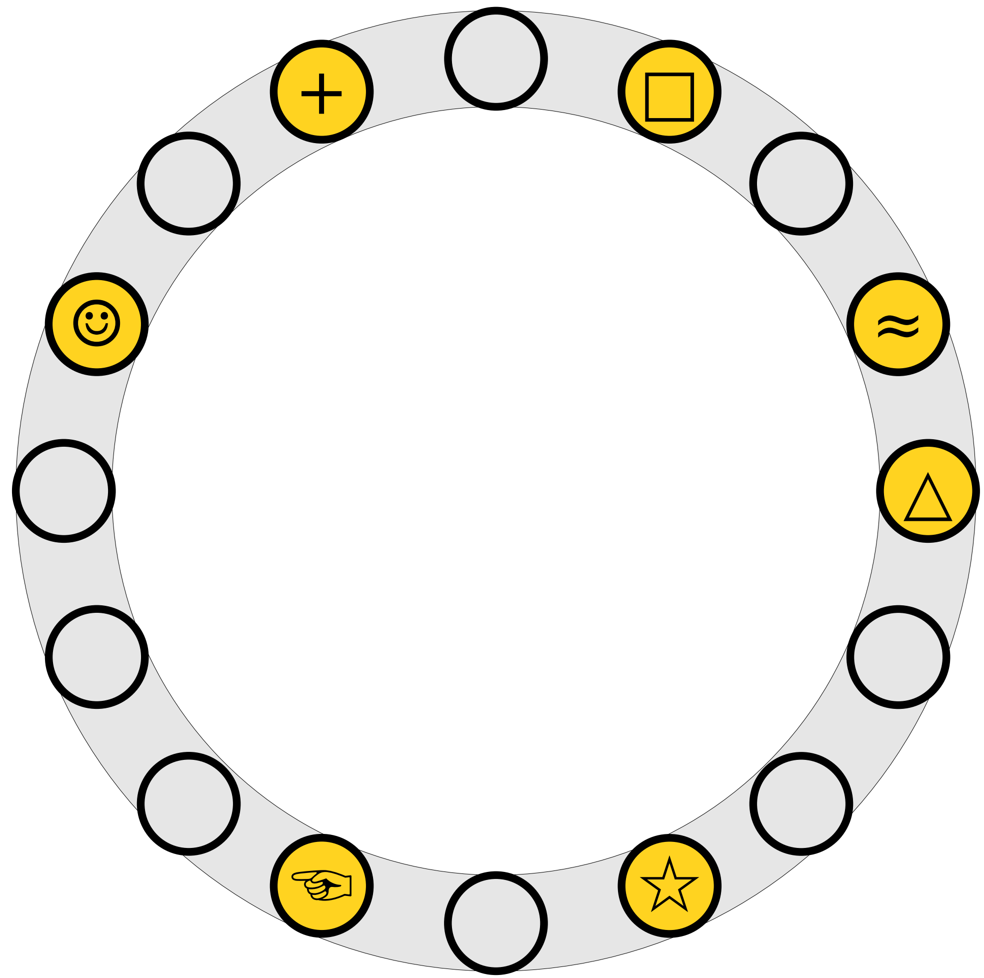
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.
- If that slot is full, kick out the element y that used to be in that slot and try placing it the other slot it can belong to (either $h_1(y)$ or $h_2(y)$).
- Repeat this process until all elements stabilize.



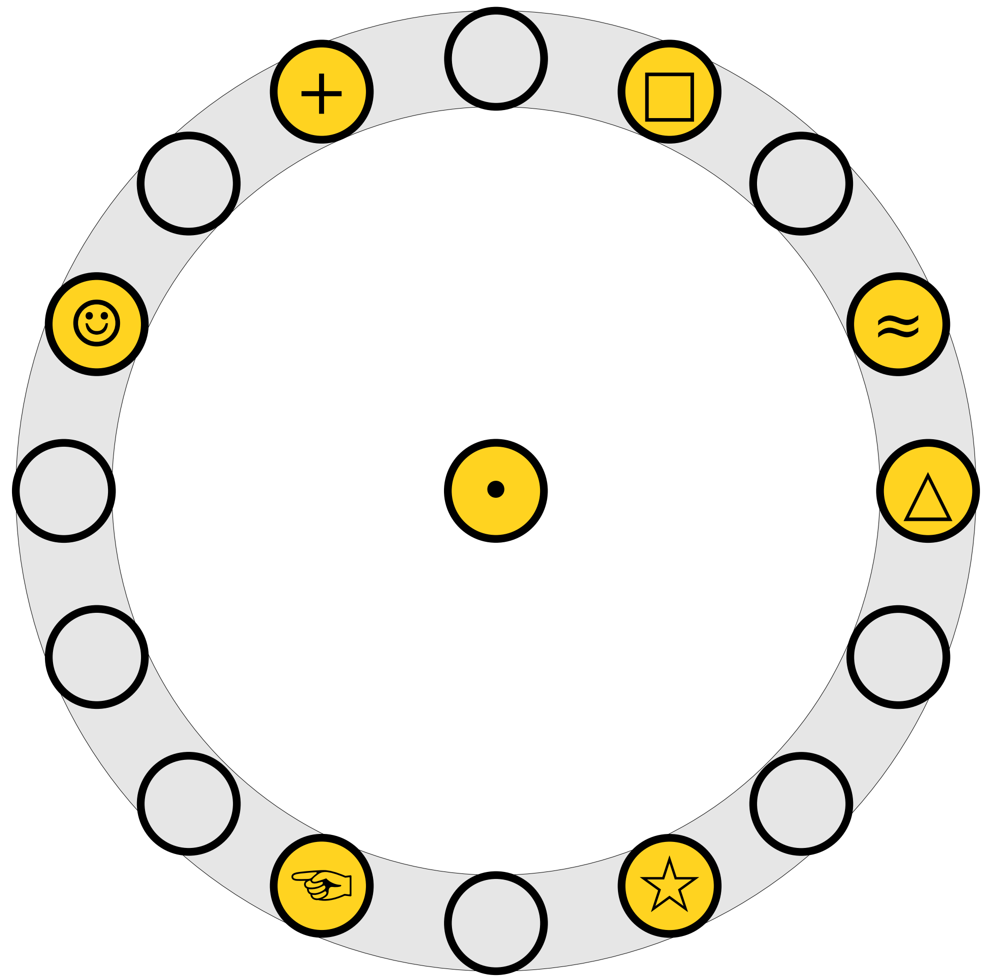
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.
- If that slot is full, kick out the element y that used to be in that slot and try placing it the other slot it can belong to (either $h_1(y)$ or $h_2(y)$).
- Repeat this process until all elements stabilize.



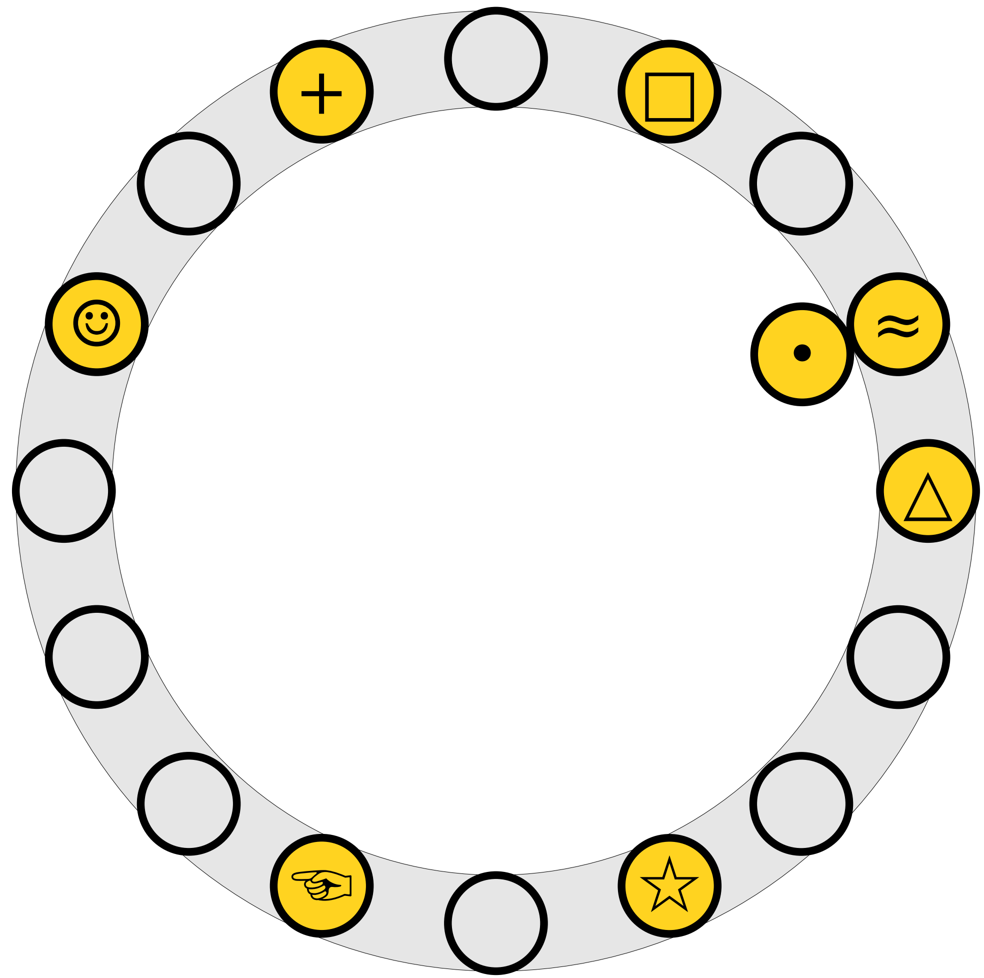
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.
- If that slot is full, kick out the element y that used to be in that slot and try placing it the other slot it can belong to (either $h_1(y)$ or $h_2(y)$).
- Repeat this process until all elements stabilize.



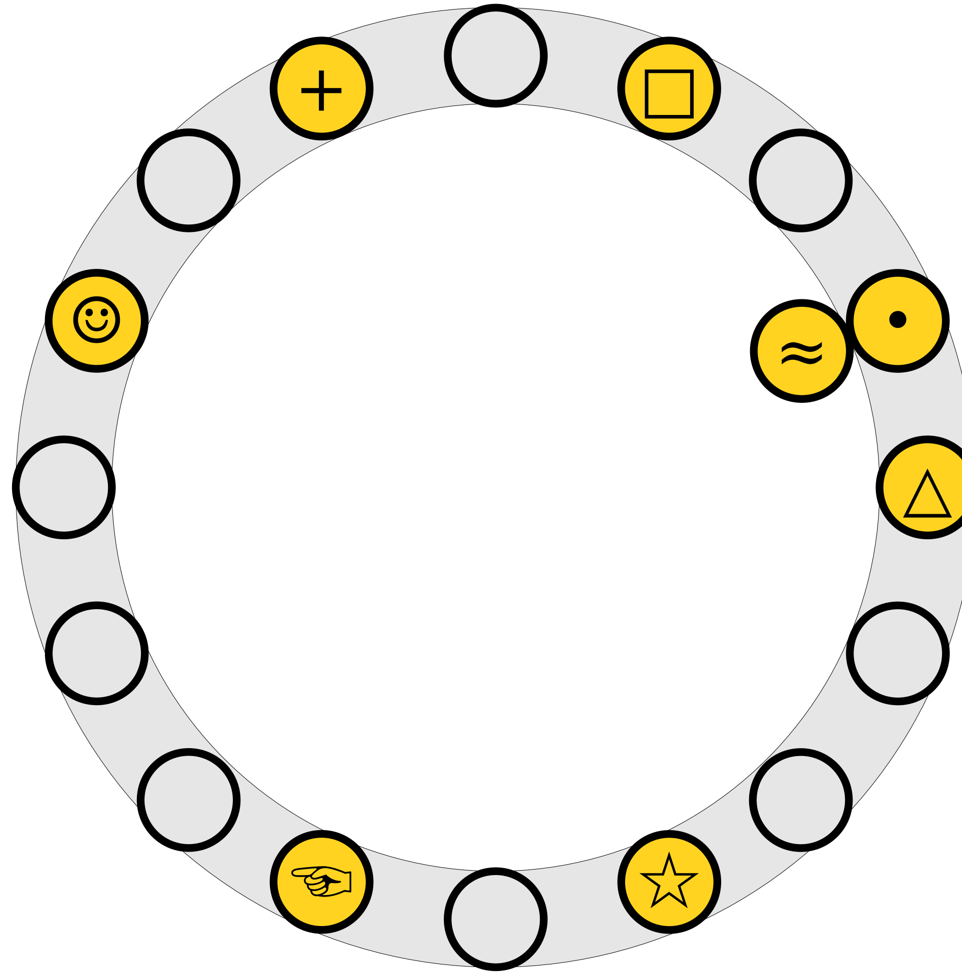
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.
- If that slot is full, kick out the element y that used to be in that slot and try placing it the other slot it can belong to (either $h_1(y)$ or $h_2(y)$).
- Repeat this process until all elements stabilize.



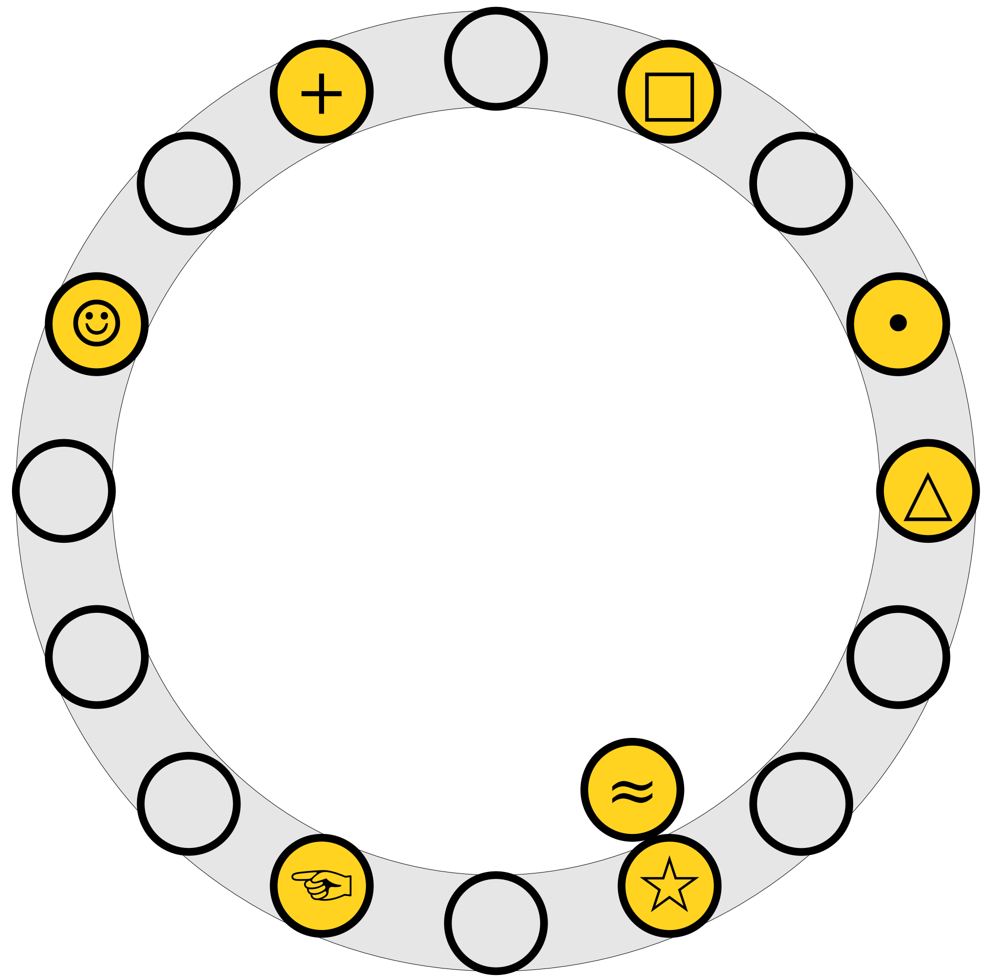
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.
- If that slot is full, kick out the element y that used to be in that slot and try placing it the other slot it can belong to (either $h_1(y)$ or $h_2(y)$).
- Repeat this process until all elements stabilize.



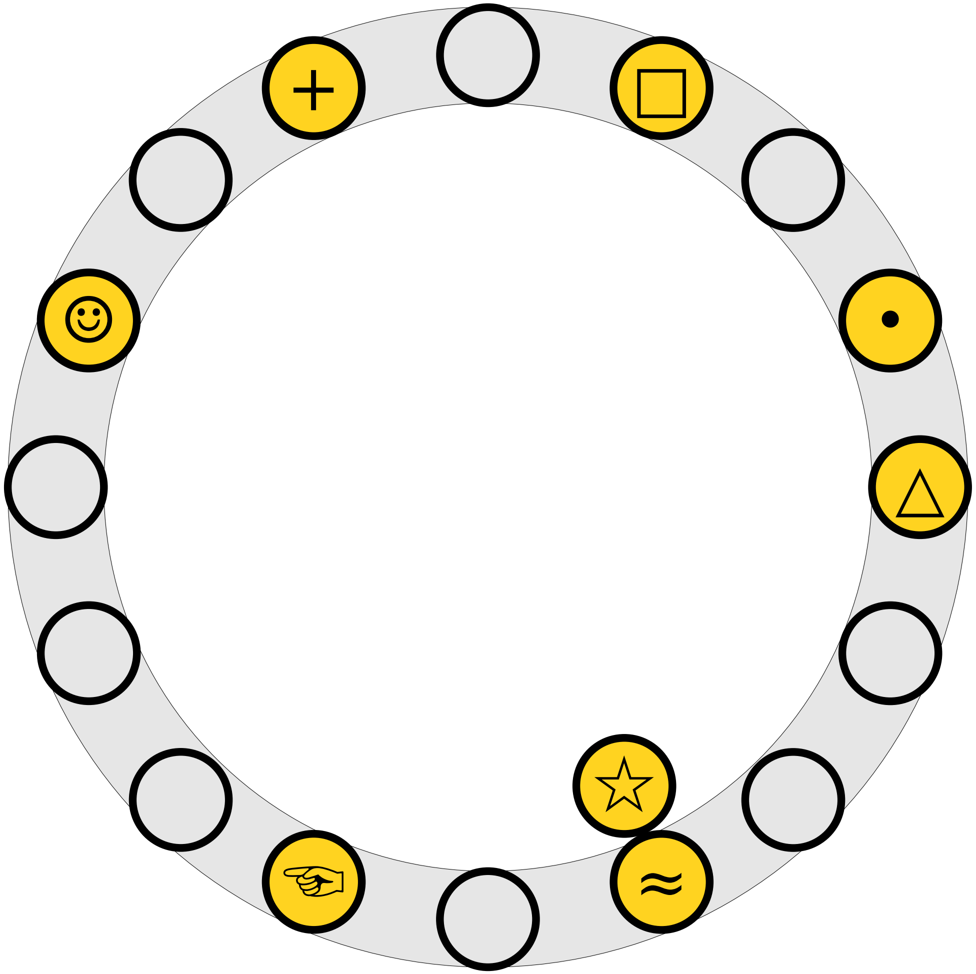
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.
- If that slot is full, kick out the element y that used to be in that slot and try placing it the other slot it can belong to (either $h_1(y)$ or $h_2(y)$).
- Repeat this process until all elements stabilize.



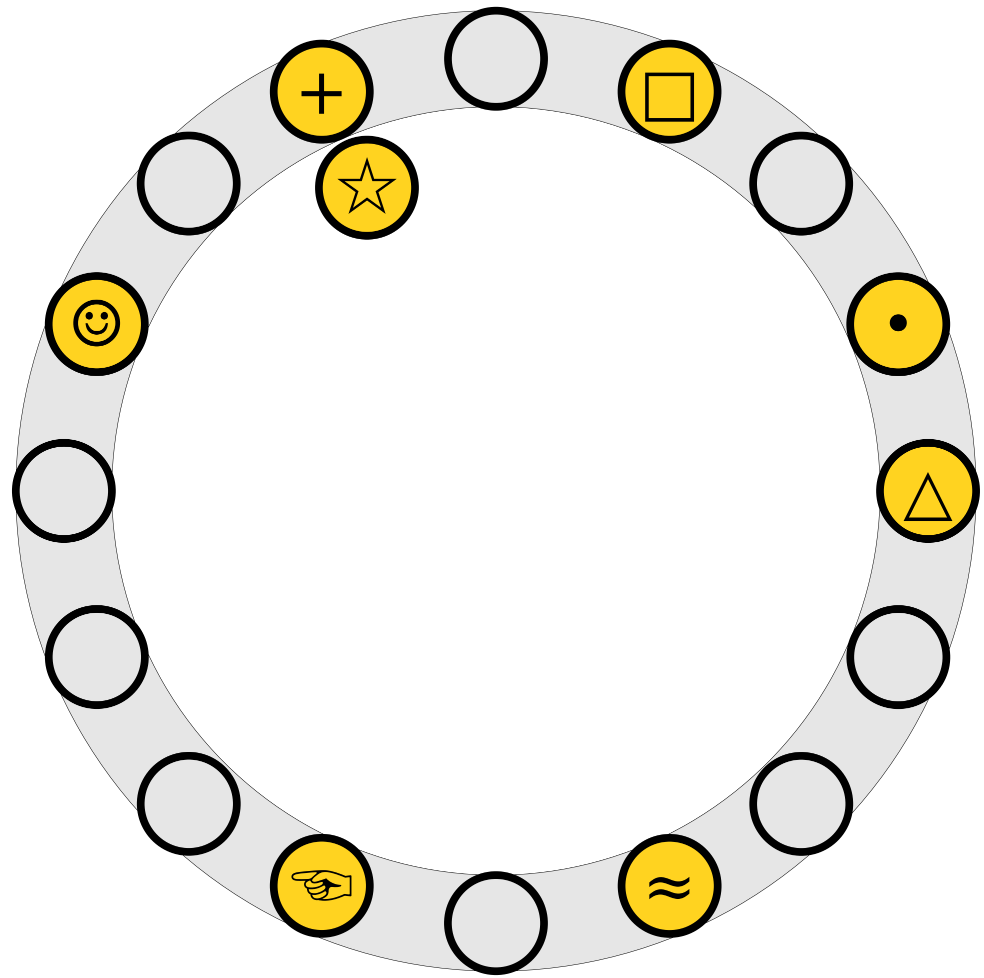
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.
- If that slot is full, kick out the element y that used to be in that slot and try placing it the other slot it can belong to (either $h_1(y)$ or $h_2(y)$).
- Repeat this process until all elements stabilize.



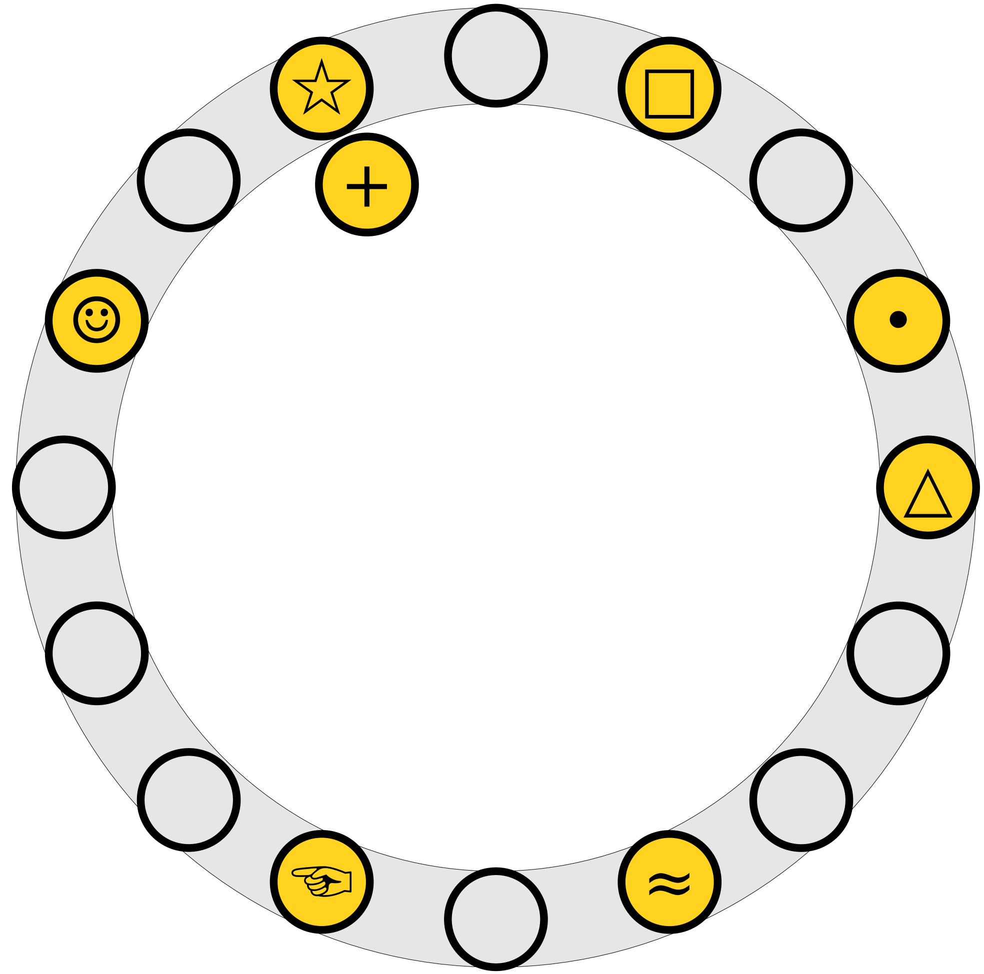
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.
- If that slot is full, kick out the element y that used to be in that slot and try placing it the other slot it can belong to (either $h_1(y)$ or $h_2(y)$).
- Repeat this process until all elements stabilize.



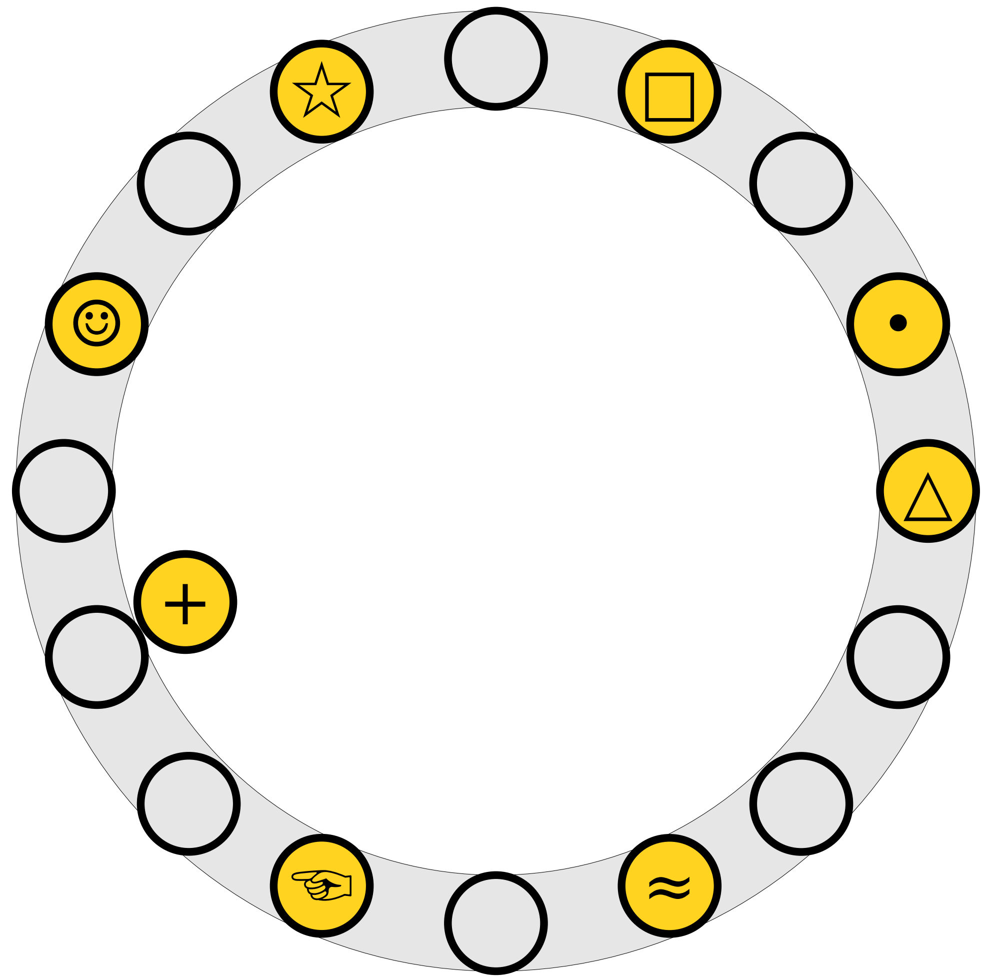
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.
- If that slot is full, kick out the element y that used to be in that slot and try placing it the other slot it can belong to (either $h_1(y)$ or $h_2(y)$).
- Repeat this process until all elements stabilize.



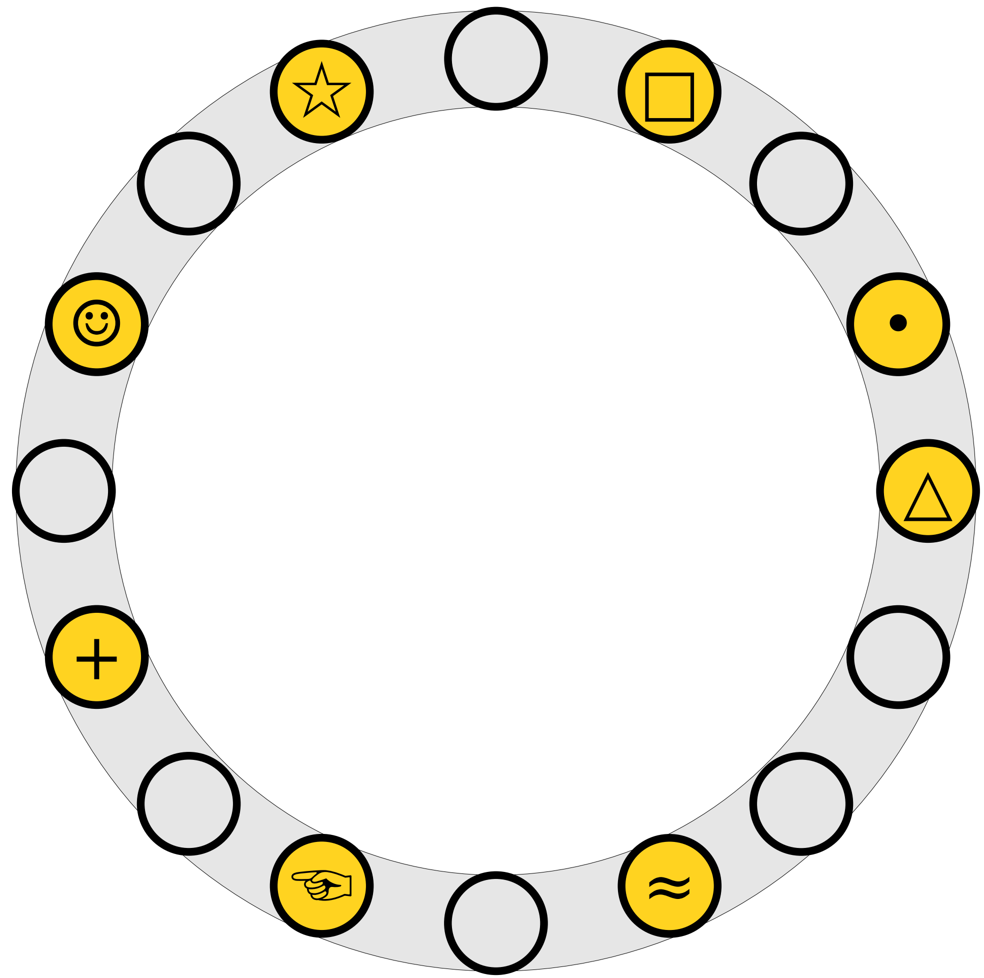
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.
- If that slot is full, kick out the element y that used to be in that slot and try placing it the other slot it can belong to (either $h_1(y)$ or $h_2(y)$).
- Repeat this process until all elements stabilize.



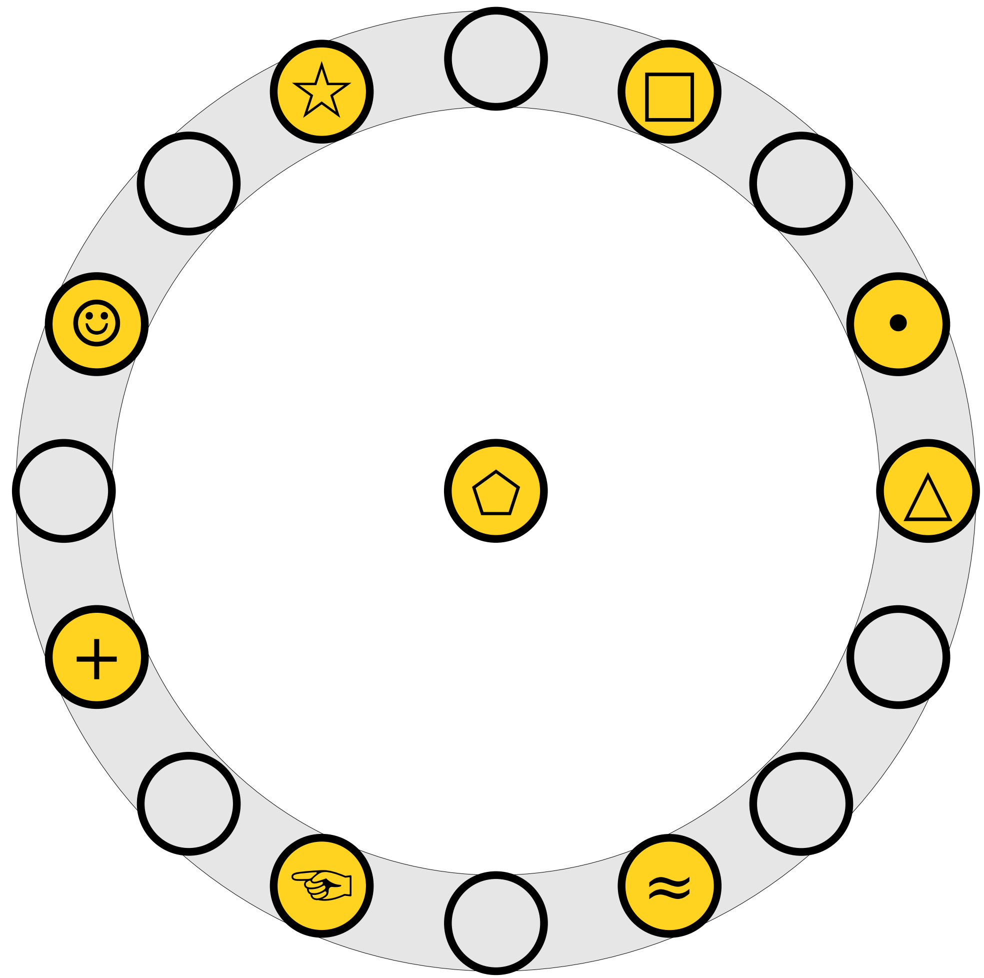
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.
- If that slot is full, kick out the element y that used to be in that slot and try placing it the other slot it can belong to (either $h_1(y)$ or $h_2(y)$).
- Repeat this process until all elements stabilize.



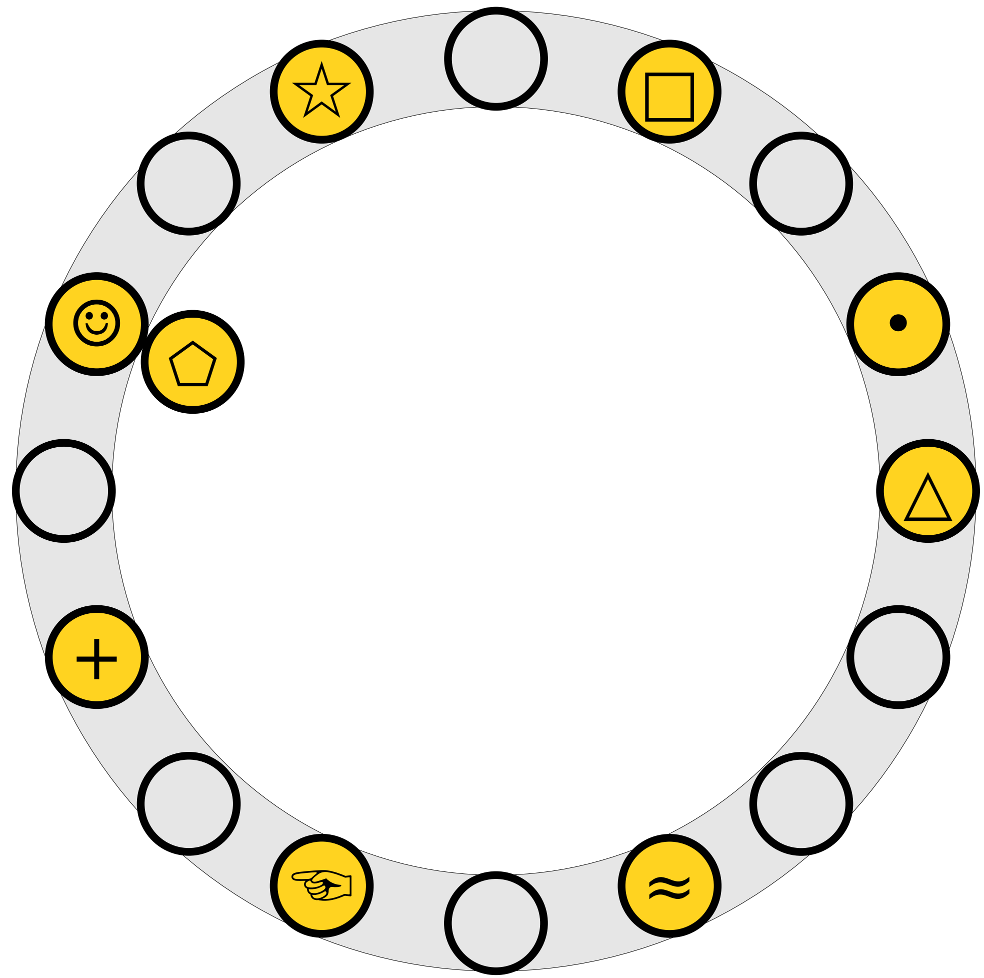
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.
- If that slot is full, kick out the element y that used to be in that slot and try placing it the other slot it can belong to (either $h_1(y)$ or $h_2(y)$).
- Repeat this process until all elements stabilize.



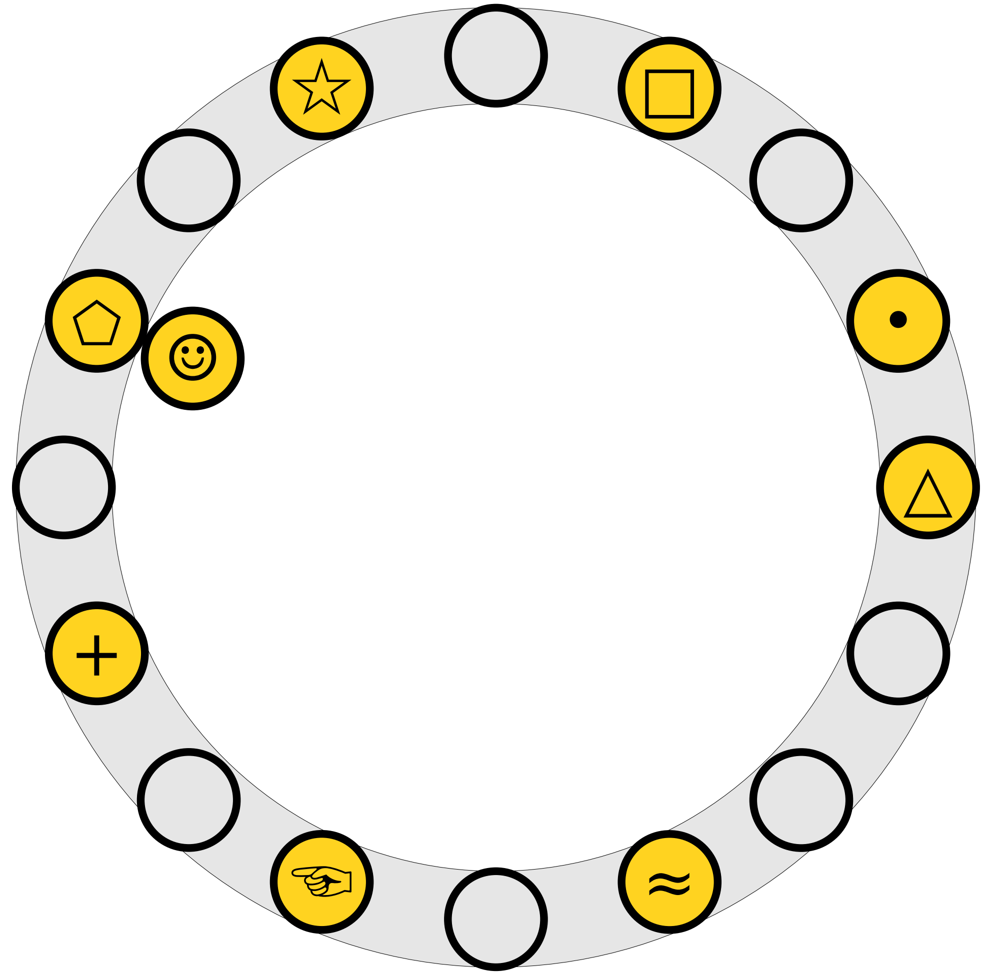
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.
- If that slot is full, kick out the element y that used to be in that slot and try placing it the other slot it can belong to (either $h_1(y)$ or $h_2(y)$).
- Repeat this process until all elements stabilize.



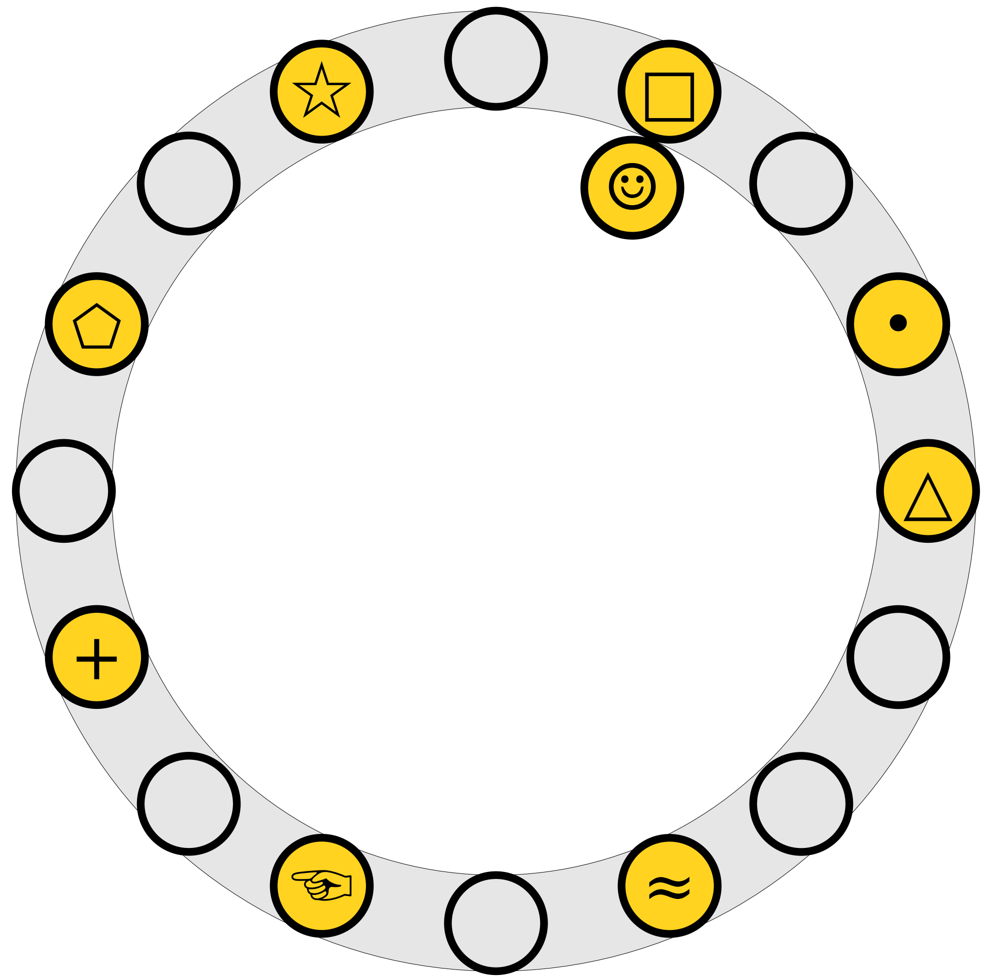
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.
- If that slot is full, kick out the element y that used to be in that slot and try placing it the other slot it can belong to (either $h_1(y)$ or $h_2(y)$).
- Repeat this process until all elements stabilize.



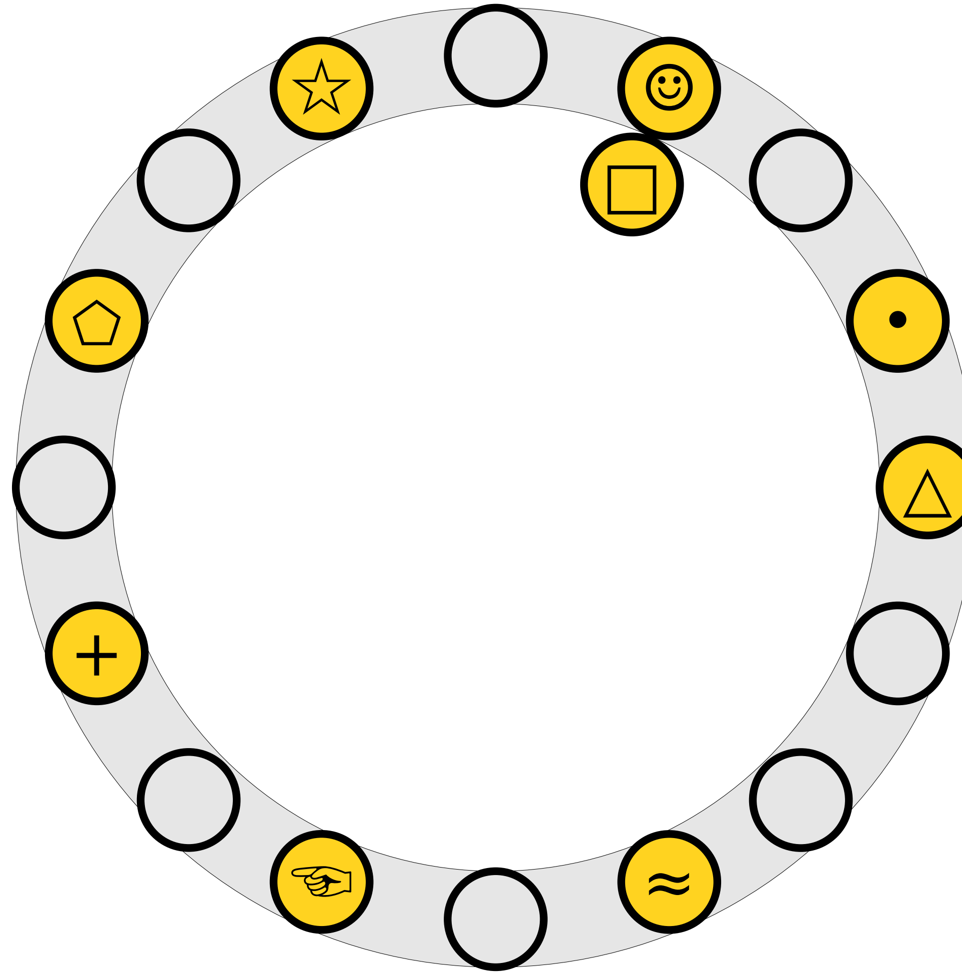
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.
- If that slot is full, kick out the element y that used to be in that slot and try placing it the other slot it can belong to (either $h_1(y)$ or $h_2(y)$).
- Repeat this process until all elements stabilize.



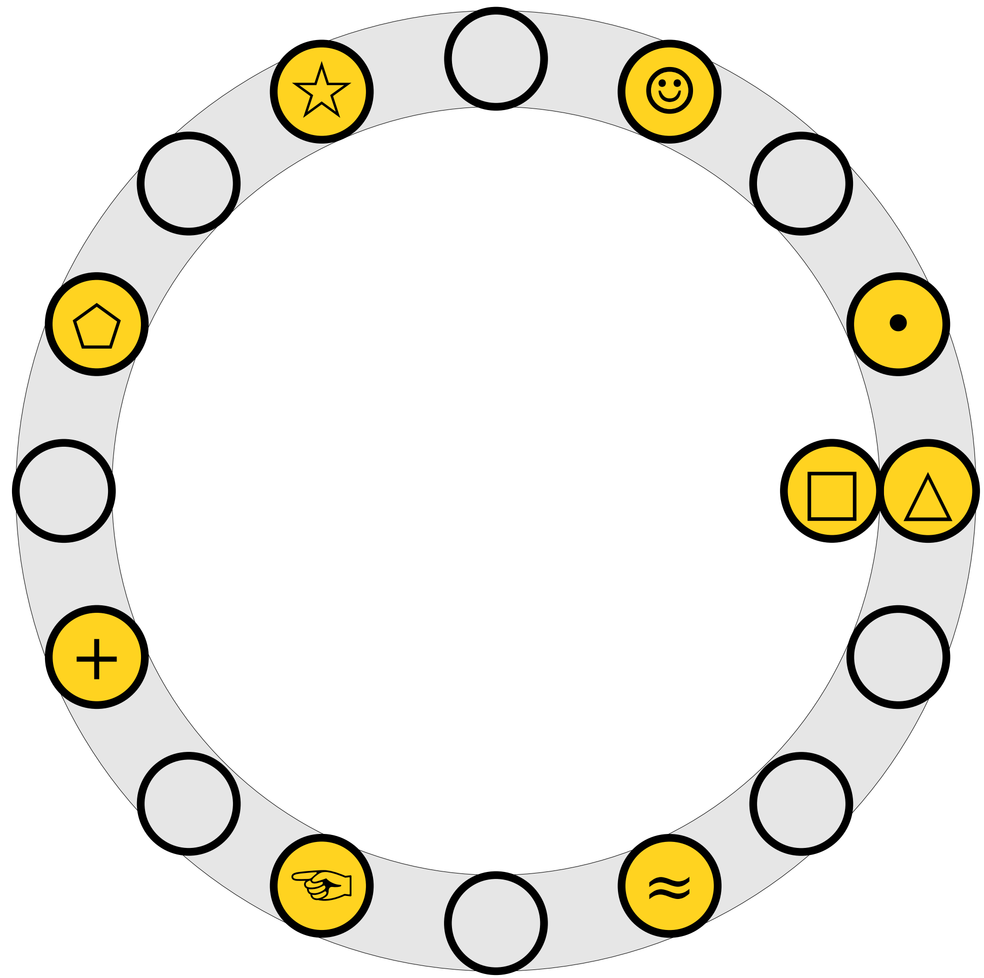
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.
- If that slot is full, kick out the element y that used to be in that slot and try placing it the other slot it can belong to (either $h_1(y)$ or $h_2(y)$).
- Repeat this process until all elements stabilize.



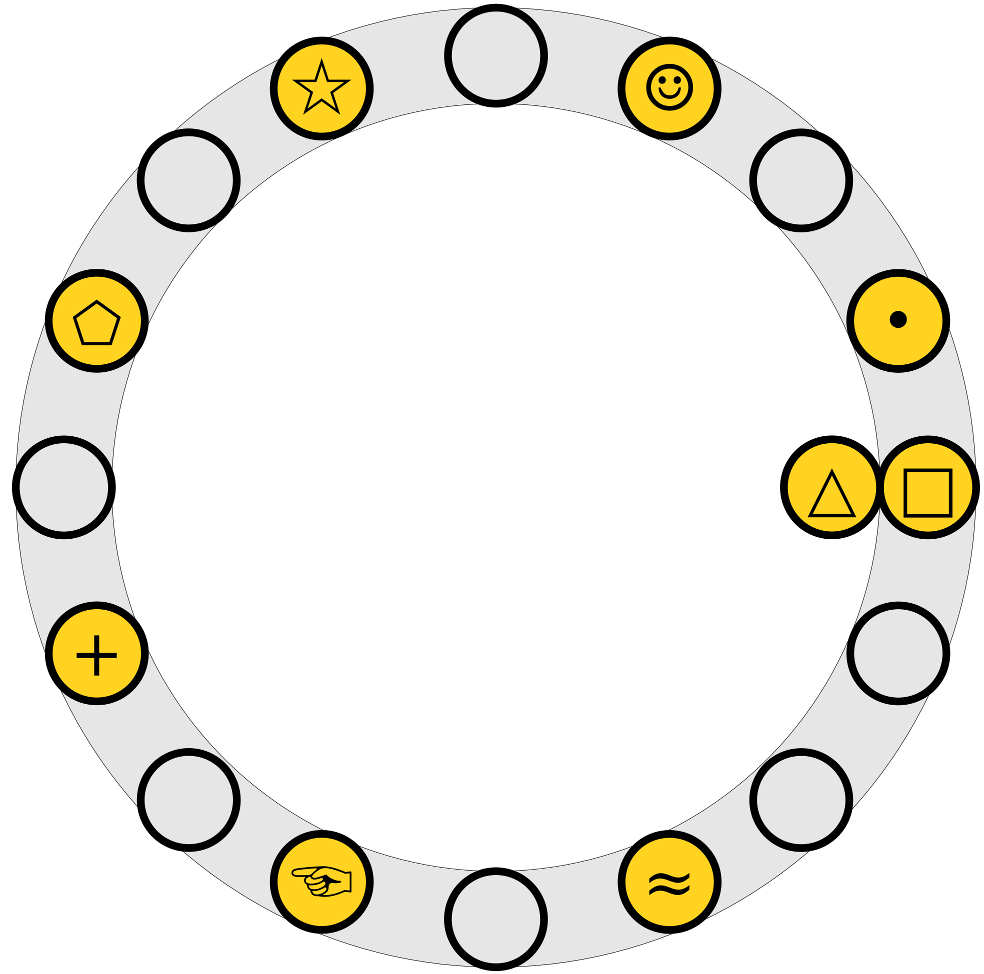
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.
- If that slot is full, kick out the element y that used to be in that slot and try placing it the other slot it can belong to (either $h_1(y)$ or $h_2(y)$).
- Repeat this process until all elements stabilize.



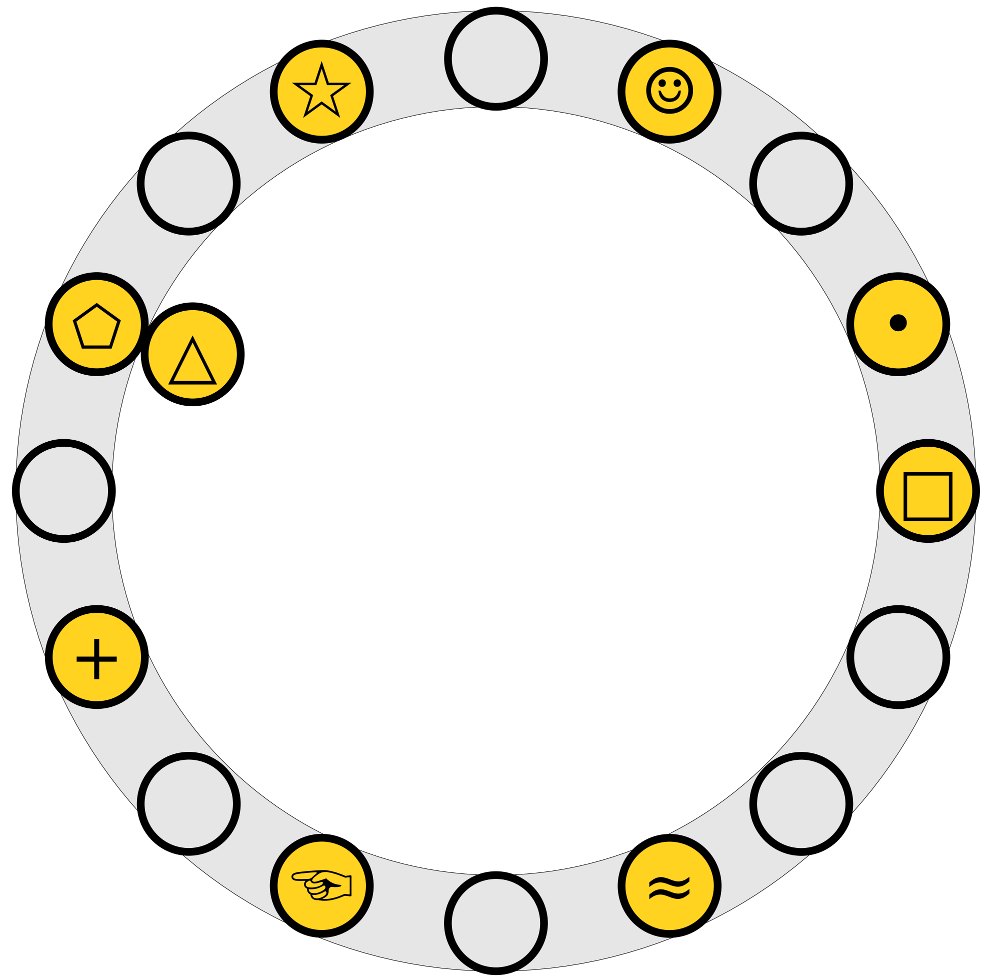
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.
- If that slot is full, kick out the element y that used to be in that slot and try placing it the other slot it can belong to (either $h_1(y)$ or $h_2(y)$).
- Repeat this process until all elements stabilize.



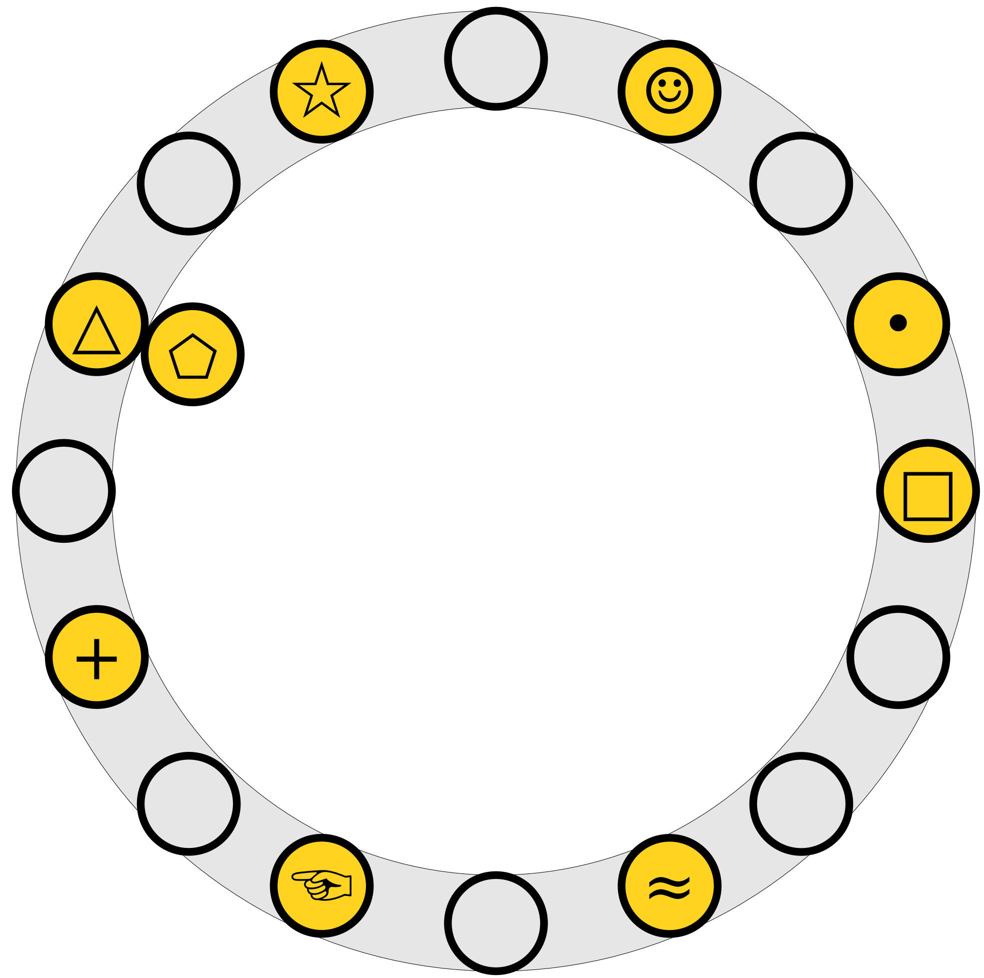
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.
- If that slot is full, kick out the element y that used to be in that slot and try placing it the other slot it can belong to (either $h_1(y)$ or $h_2(y)$).
- Repeat this process until all elements stabilize.



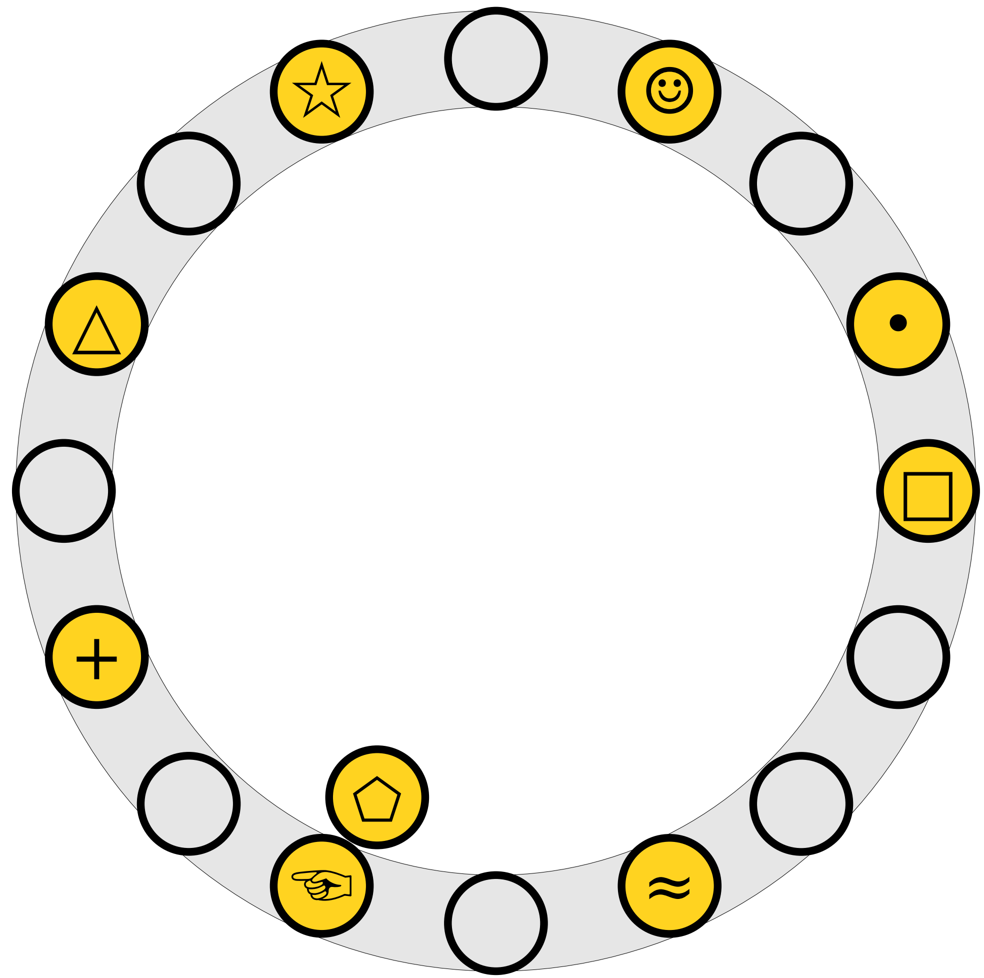
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.
- If that slot is full, kick out the element y that used to be in that slot and try placing it the other slot it can belong to (either $h_1(y)$ or $h_2(y)$).
- Repeat this process until all elements stabilize.



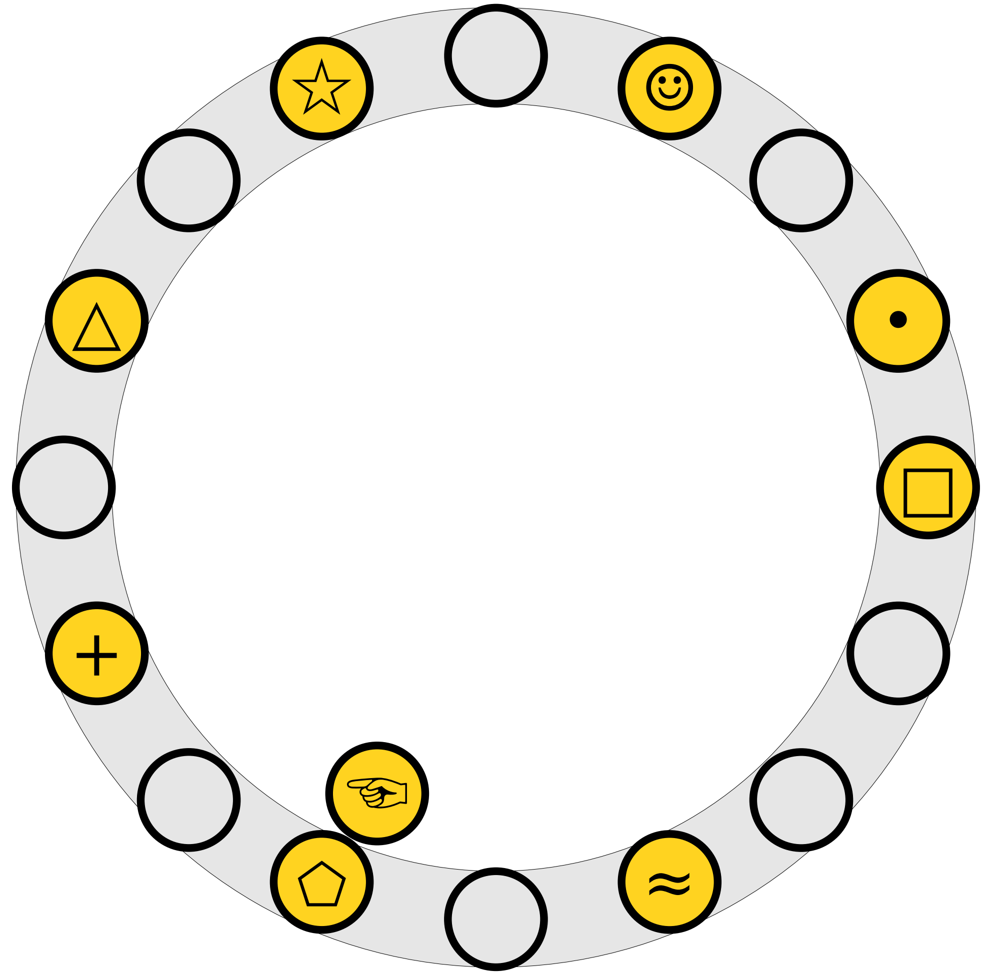
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.
- If that slot is full, kick out the element y that used to be in that slot and try placing it the other slot it can belong to (either $h_1(y)$ or $h_2(y)$).
- Repeat this process until all elements stabilize.



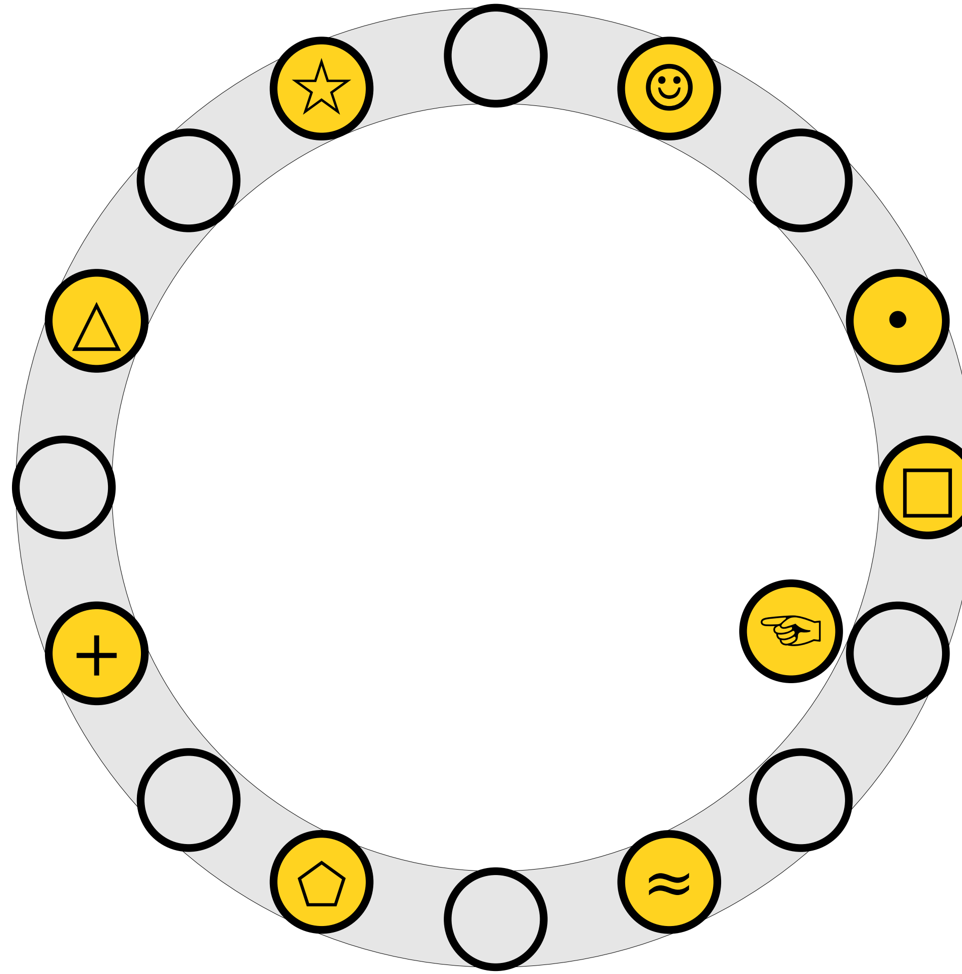
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.
- If that slot is full, kick out the element y that used to be in that slot and try placing it the other slot it can belong to (either $h_1(y)$ or $h_2(y)$).
- Repeat this process until all elements stabilize.



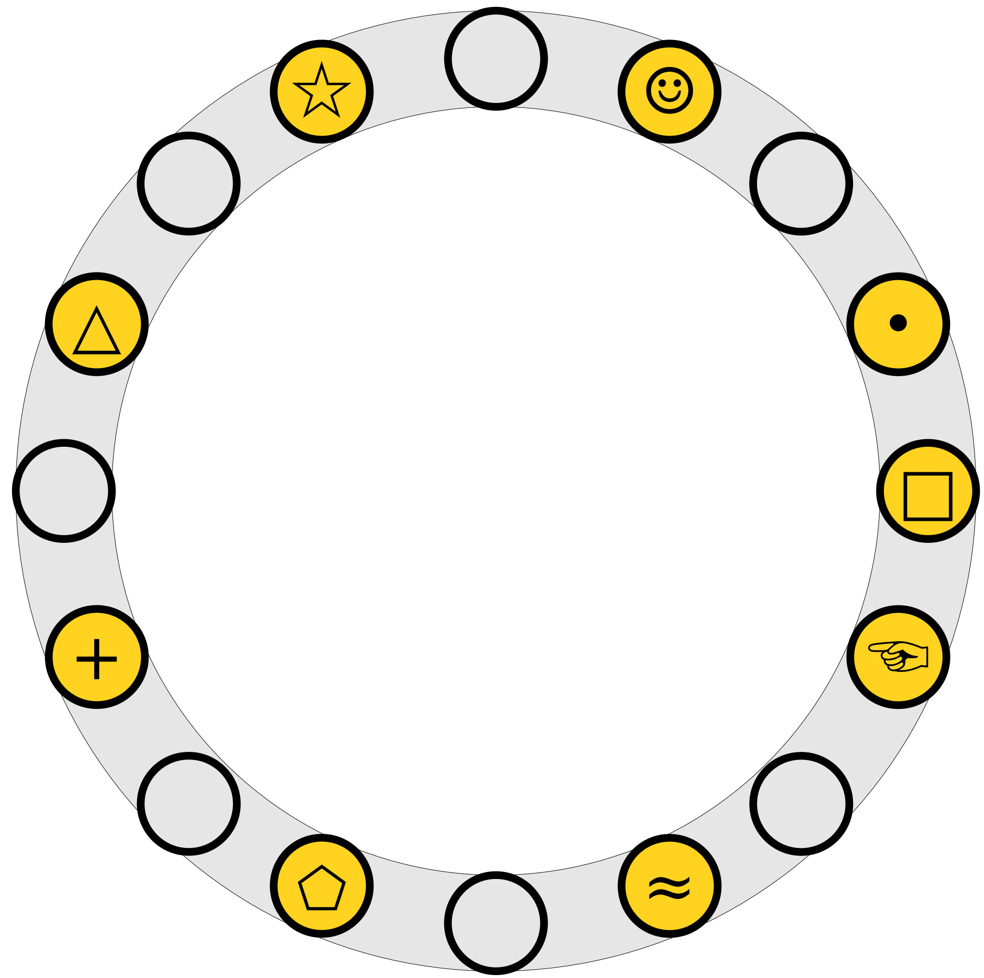
Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.
- If that slot is full, kick out the element y that used to be in that slot and try placing it the other slot it can belong to (either $h_1(y)$ or $h_2(y)$).
- Repeat this process until all elements stabilize.



Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.
- If that slot is full, kick out the element y that used to be in that slot and try placing it the other slot it can belong to (either $h_1(y)$ or $h_2(y)$).
- Repeat this process until all elements stabilize.

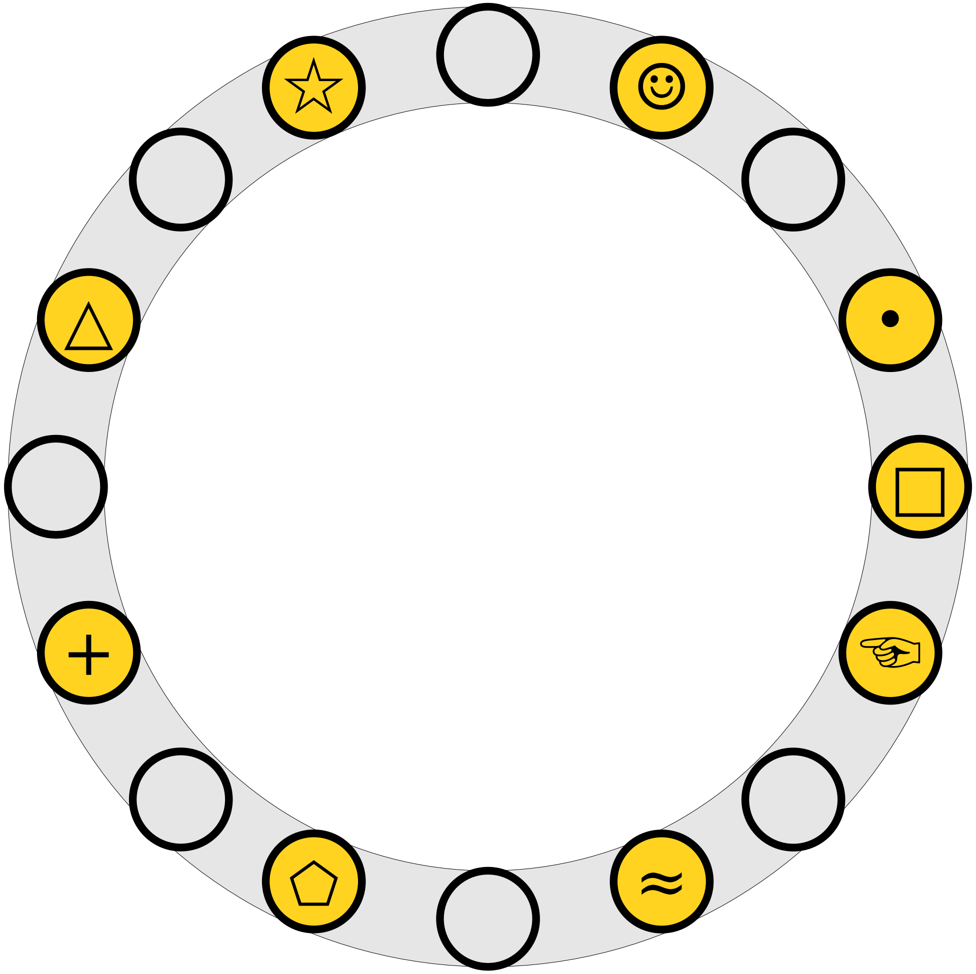


Cuckoo Hashing

- To insert x into the table, first try placing it at slot $h_1(x)$.
- If that slot is full, kick out the element y that used to be in that slot and try placing it the other slot it can belong to (either $h_1(y)$ or $h_2(y)$).
- Repeat this process until all elements stabilize.

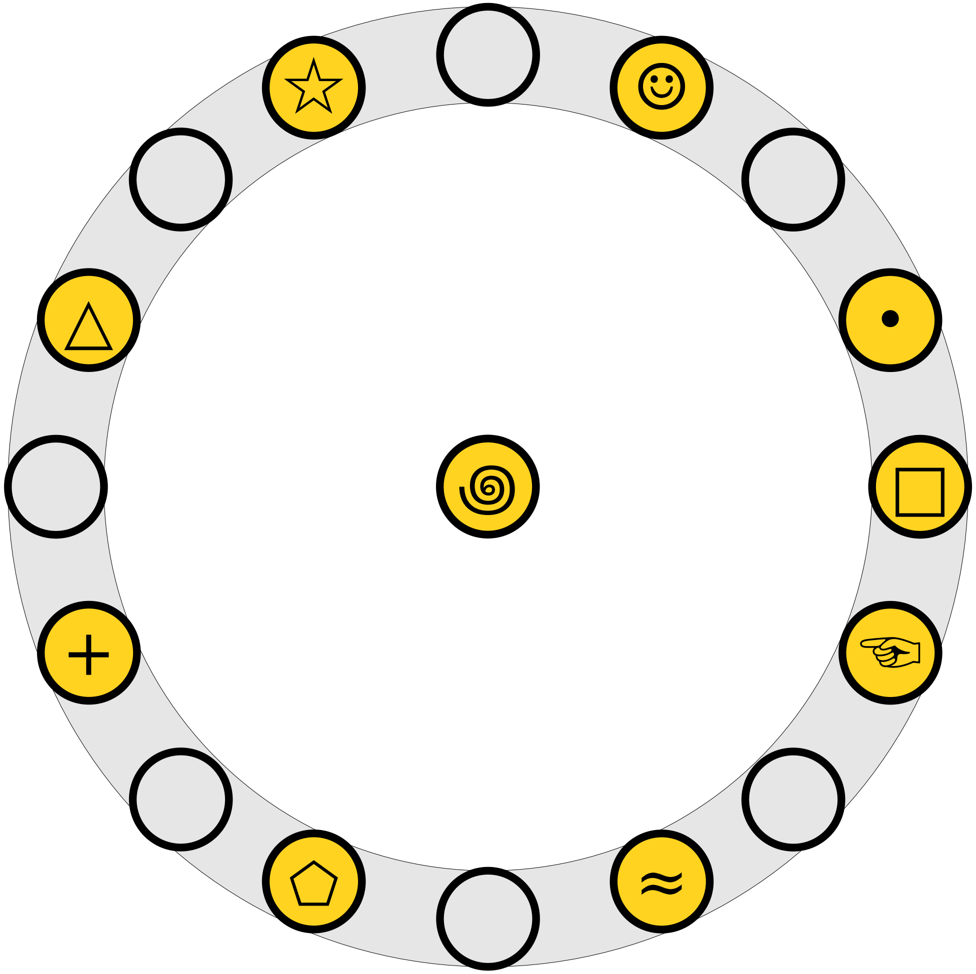
Cuckoo Hashing

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



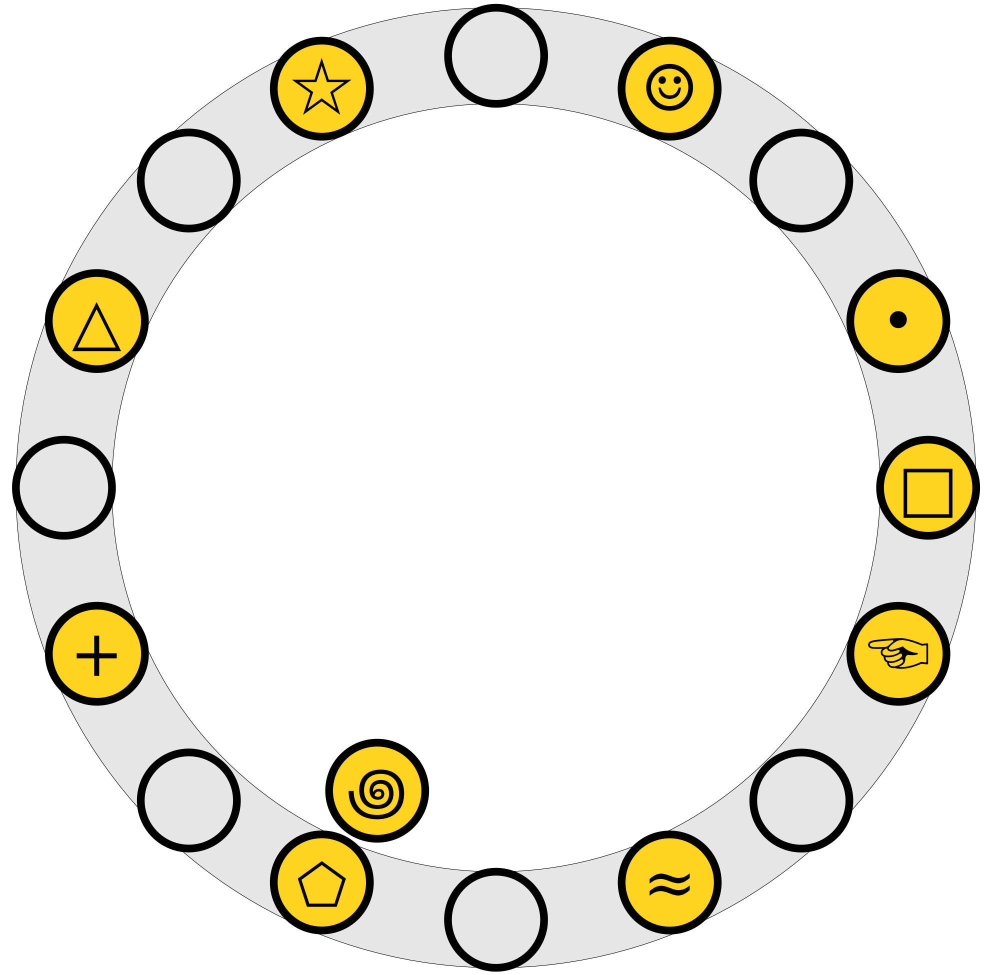
Cuckoo Hashing

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



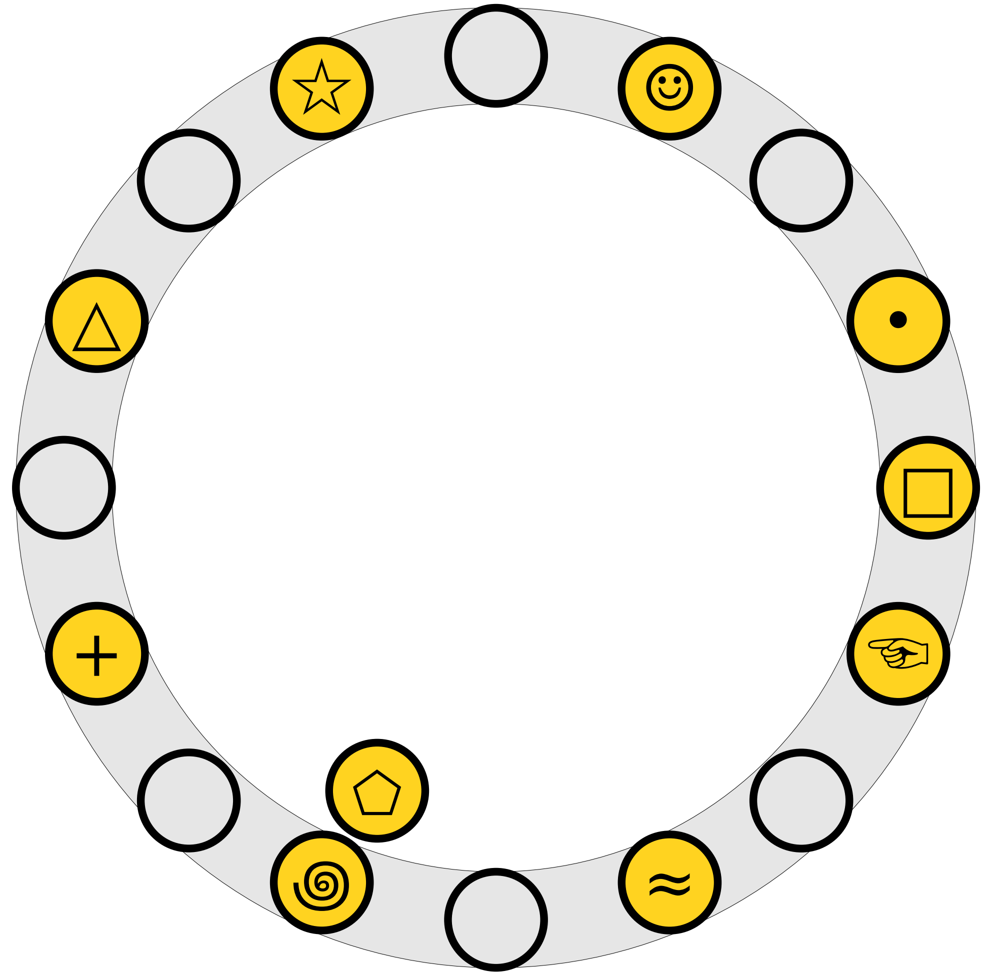
Cuckoo Hashing

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



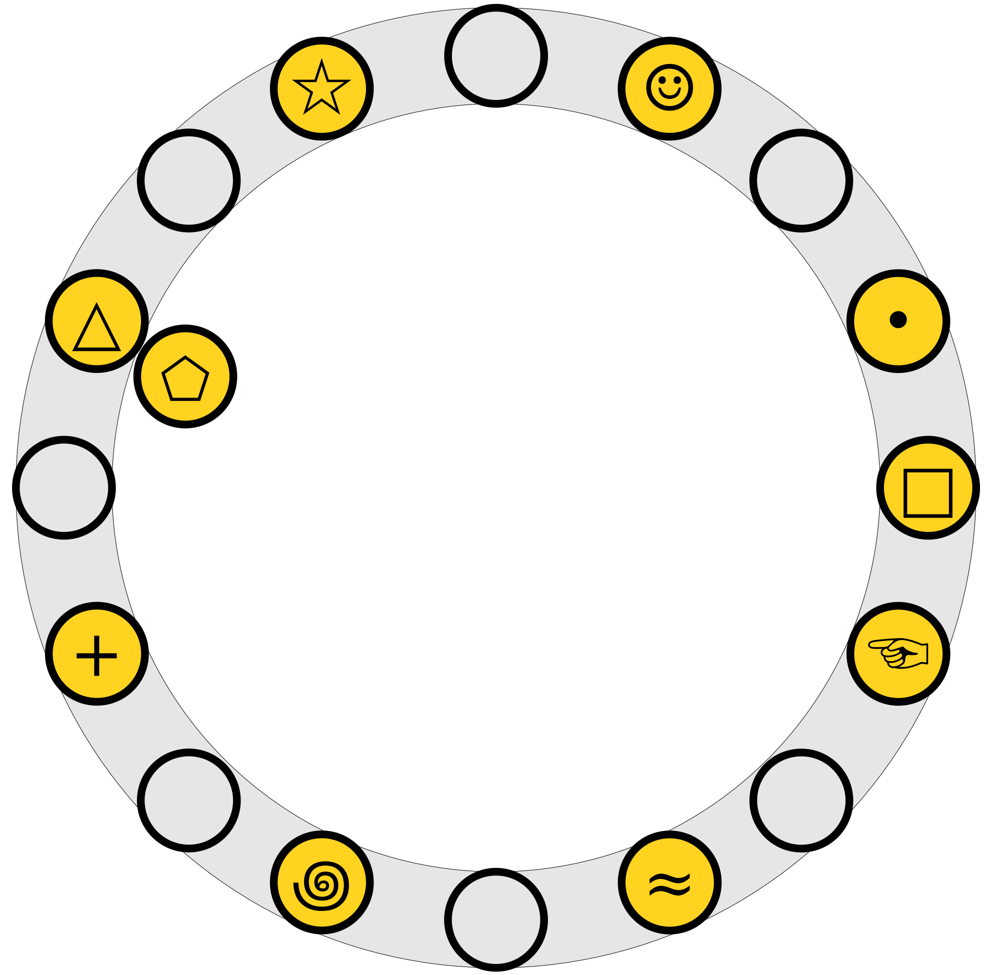
Cuckoo Hashing

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



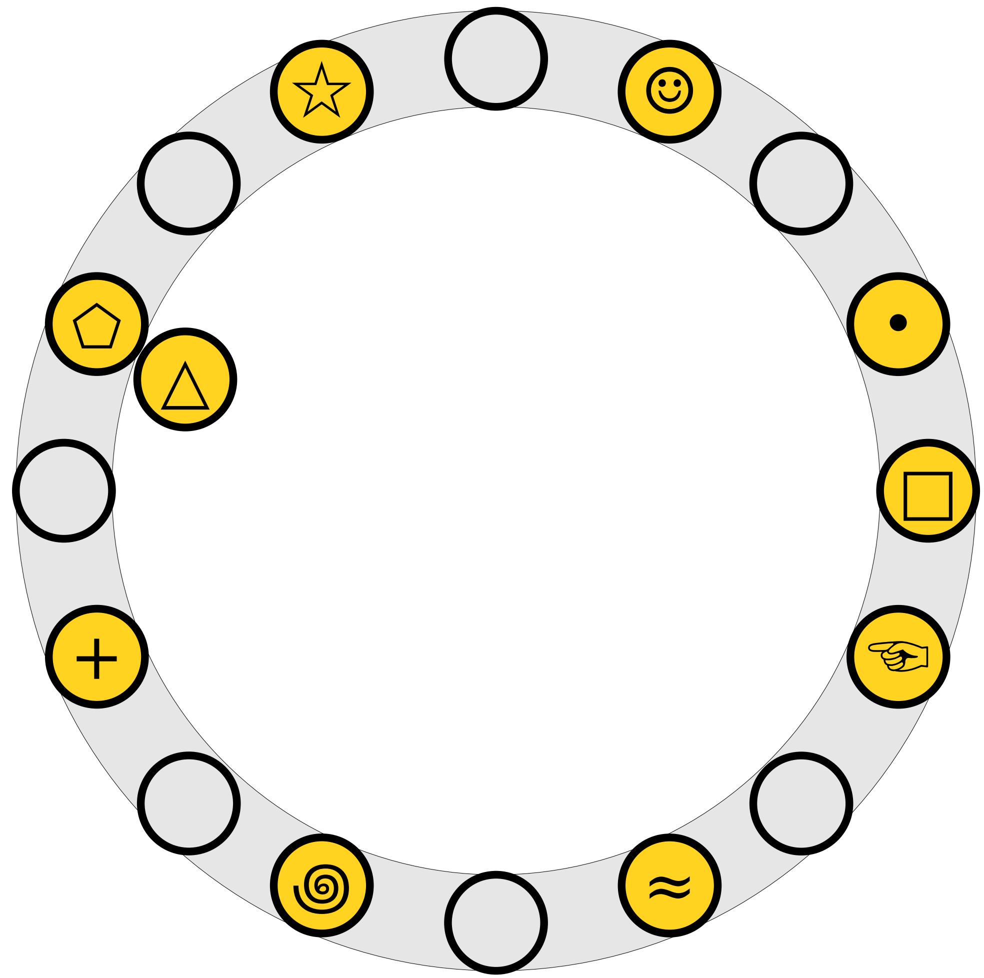
Cuckoo Hashing

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



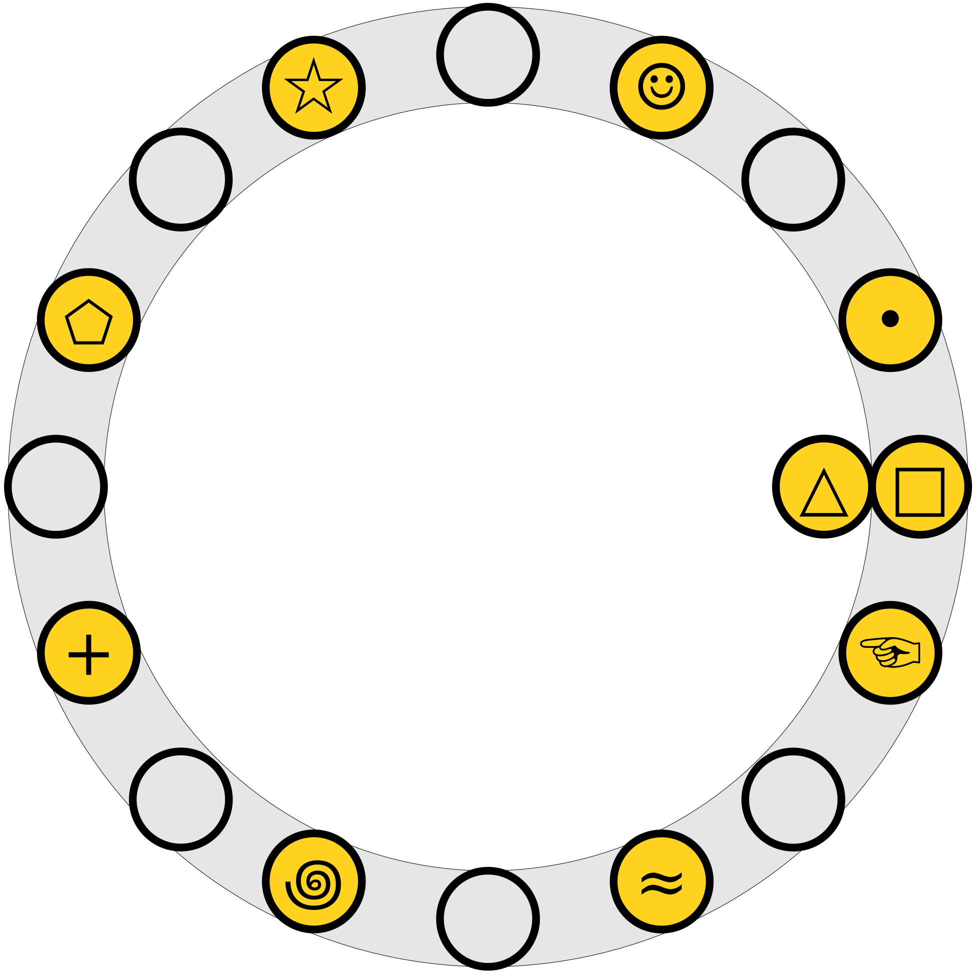
Cuckoo Hashing

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



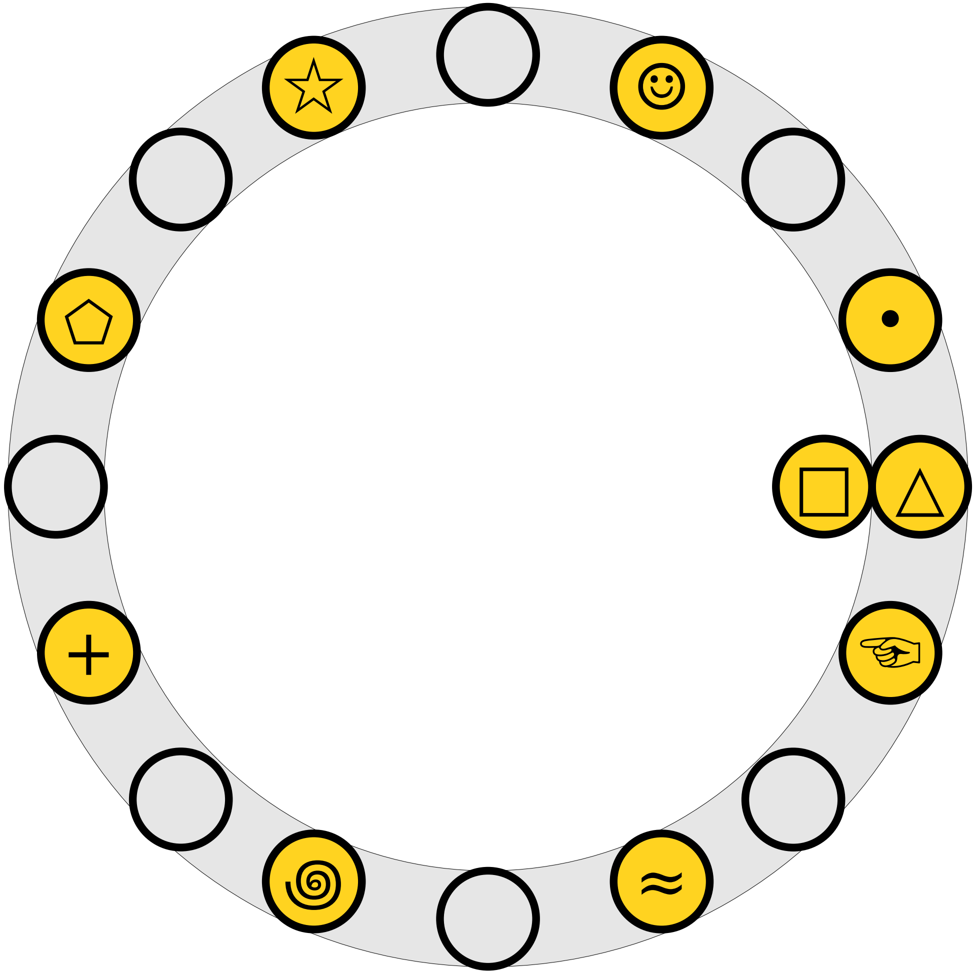
Cuckoo Hashing

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



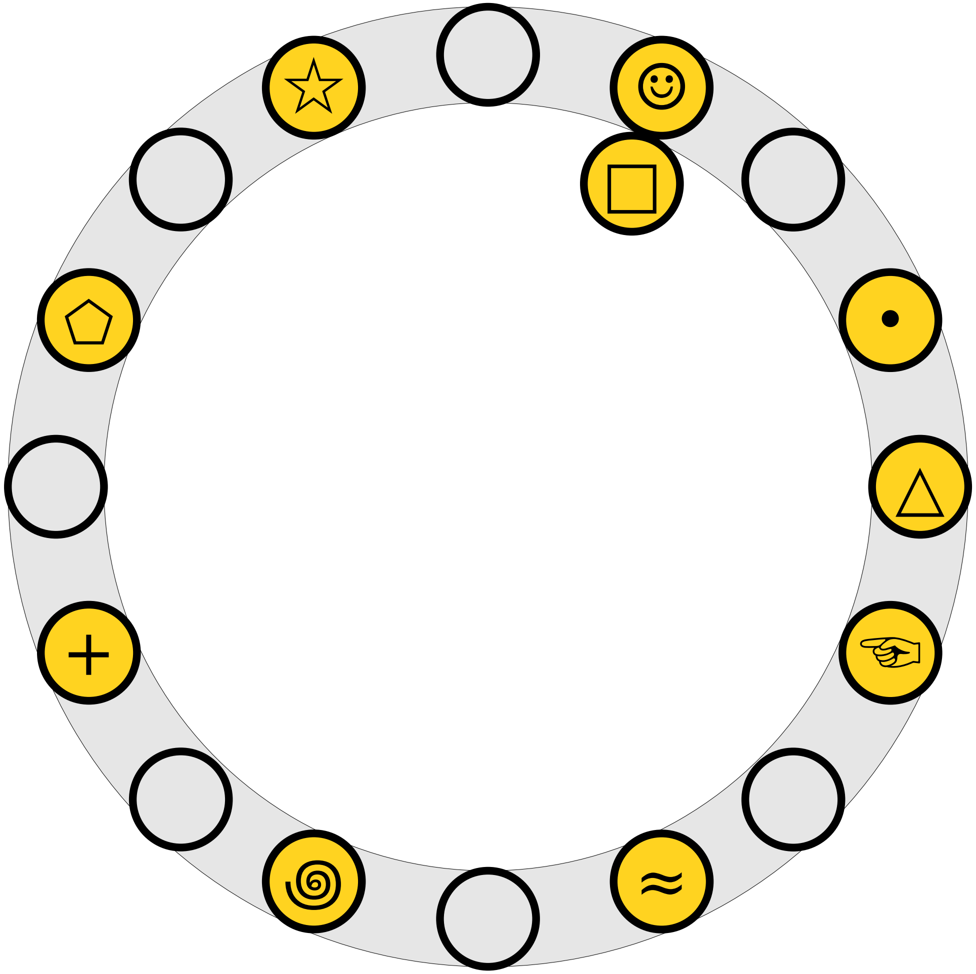
Cuckoo Hashing

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



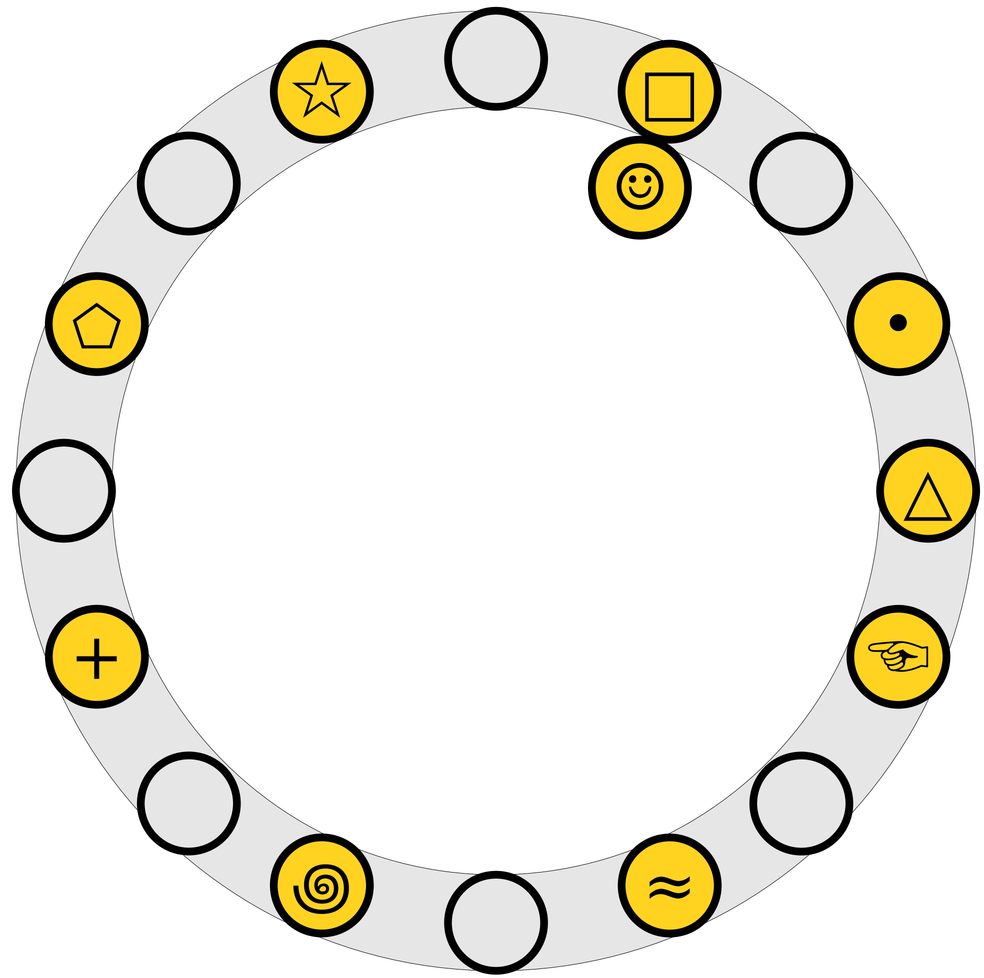
Cuckoo Hashing

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



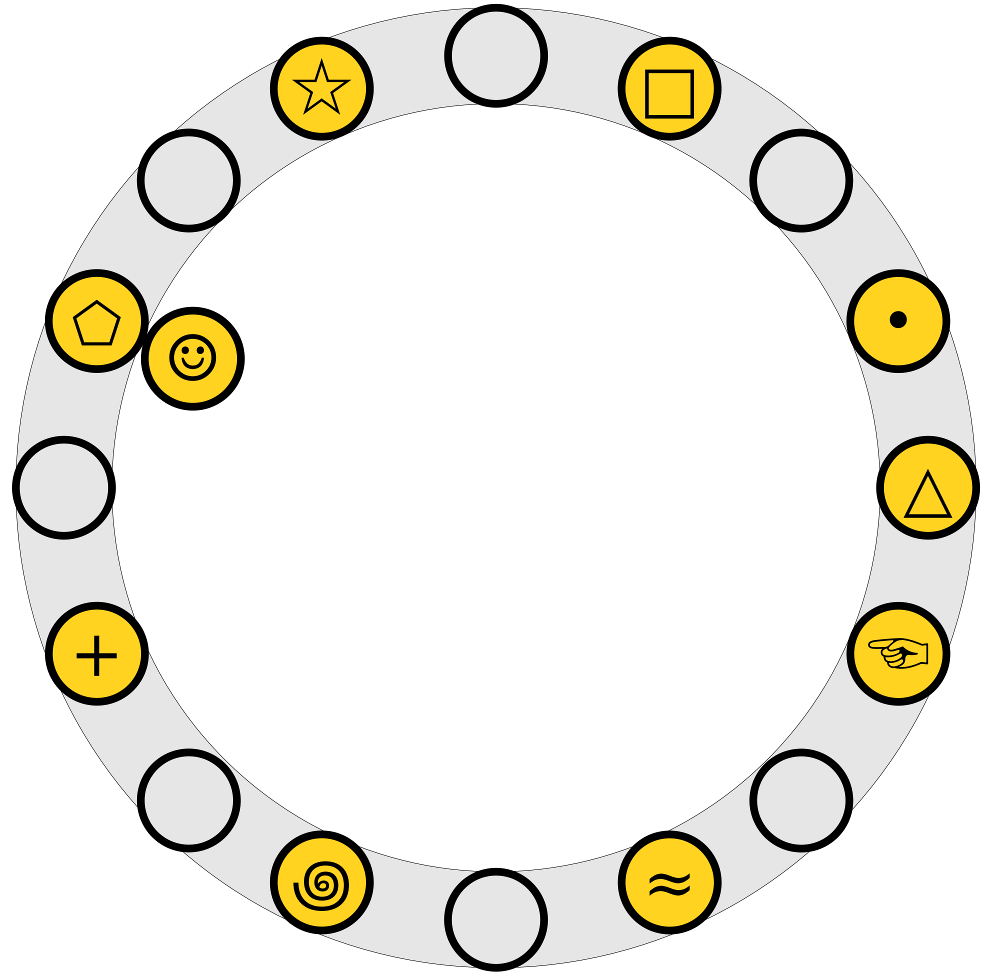
Cuckoo Hashing

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



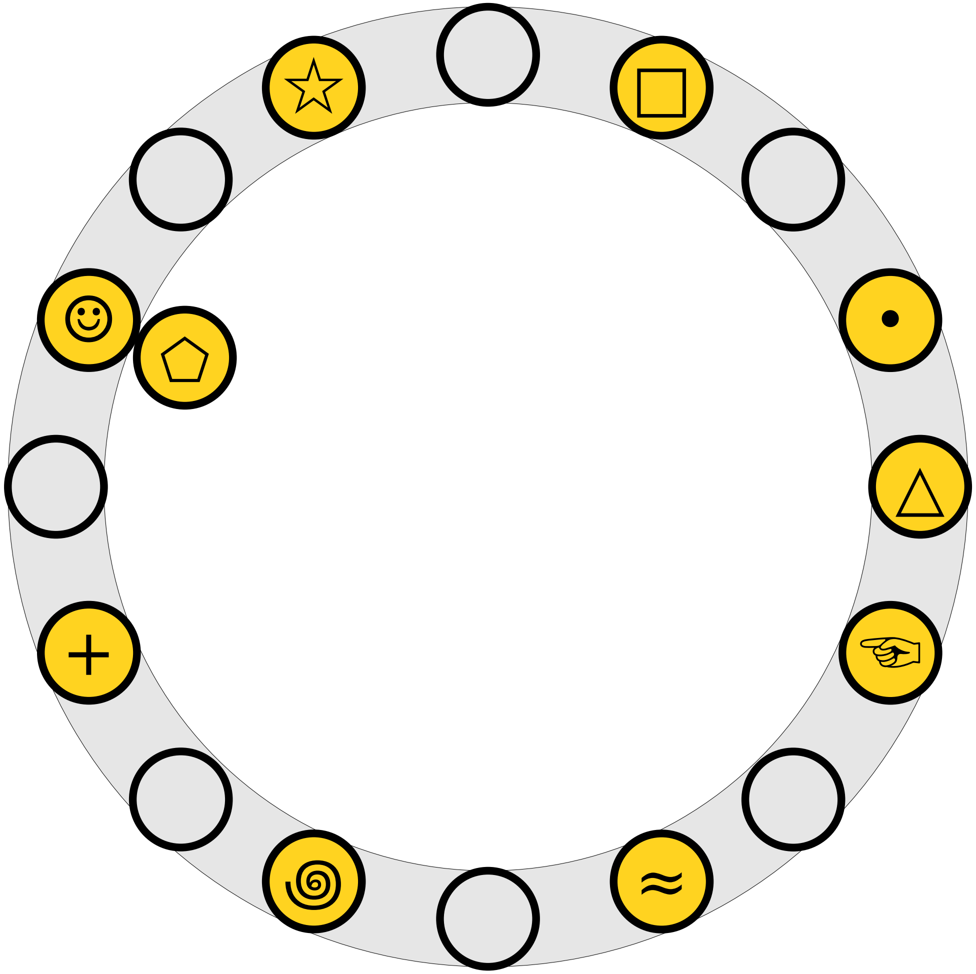
Cuckoo Hashing

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



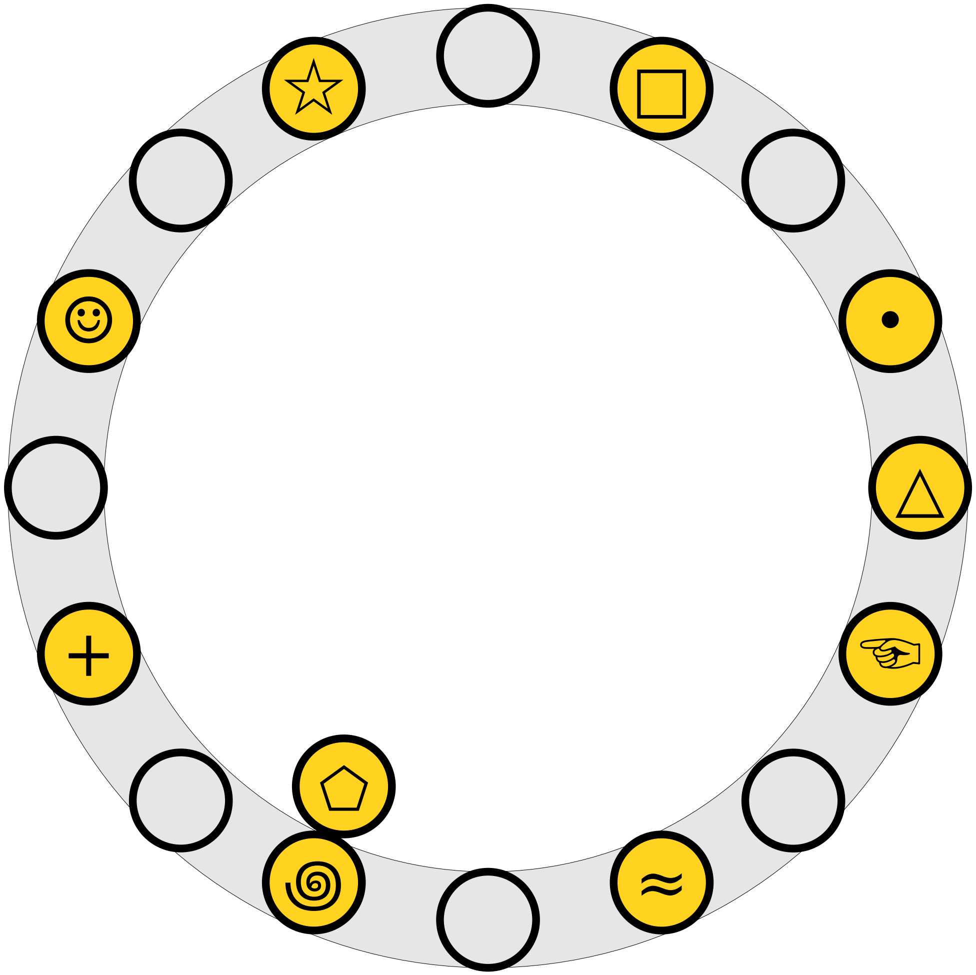
Cuckoo Hashing

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



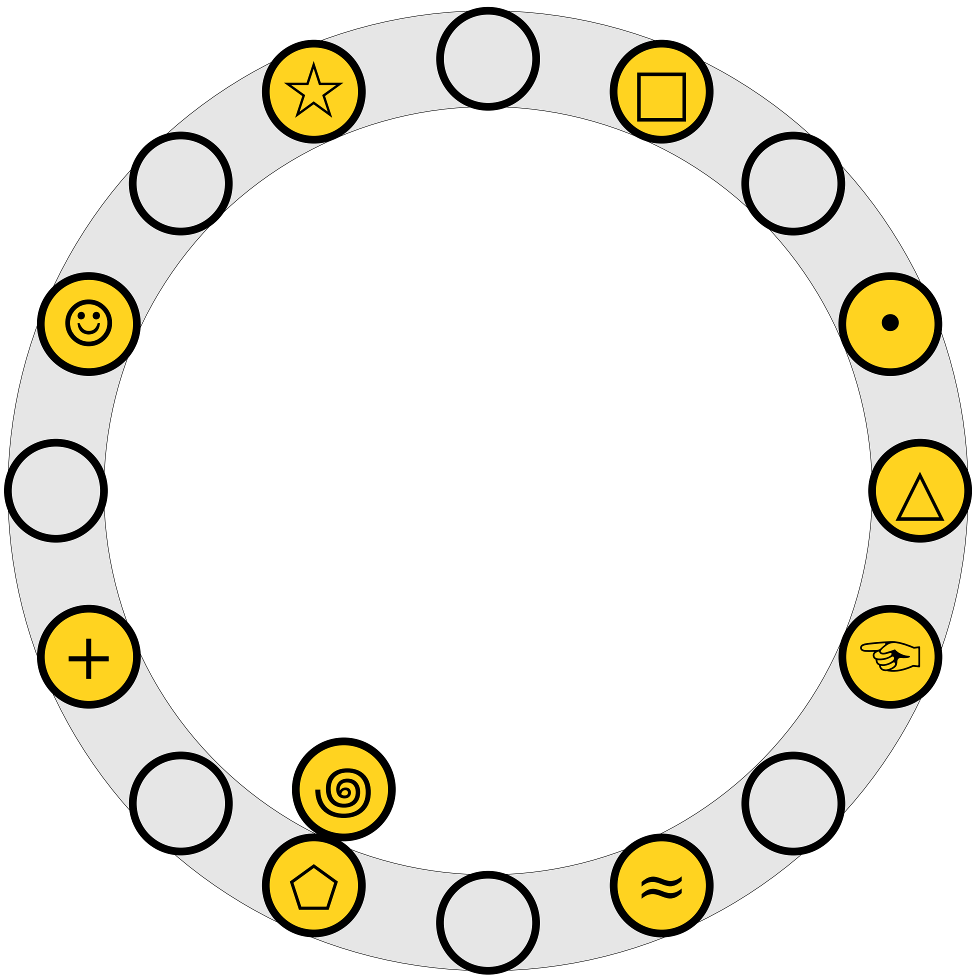
Cuckoo Hashing

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



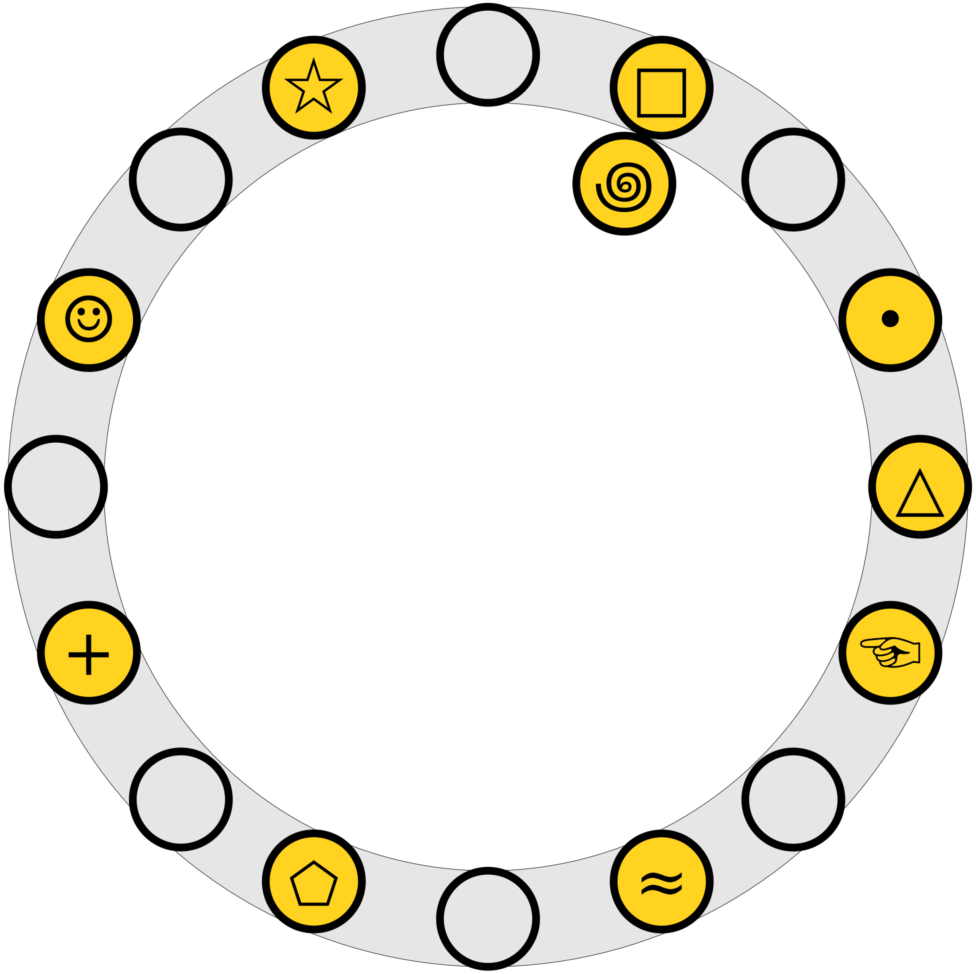
Cuckoo Hashing

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



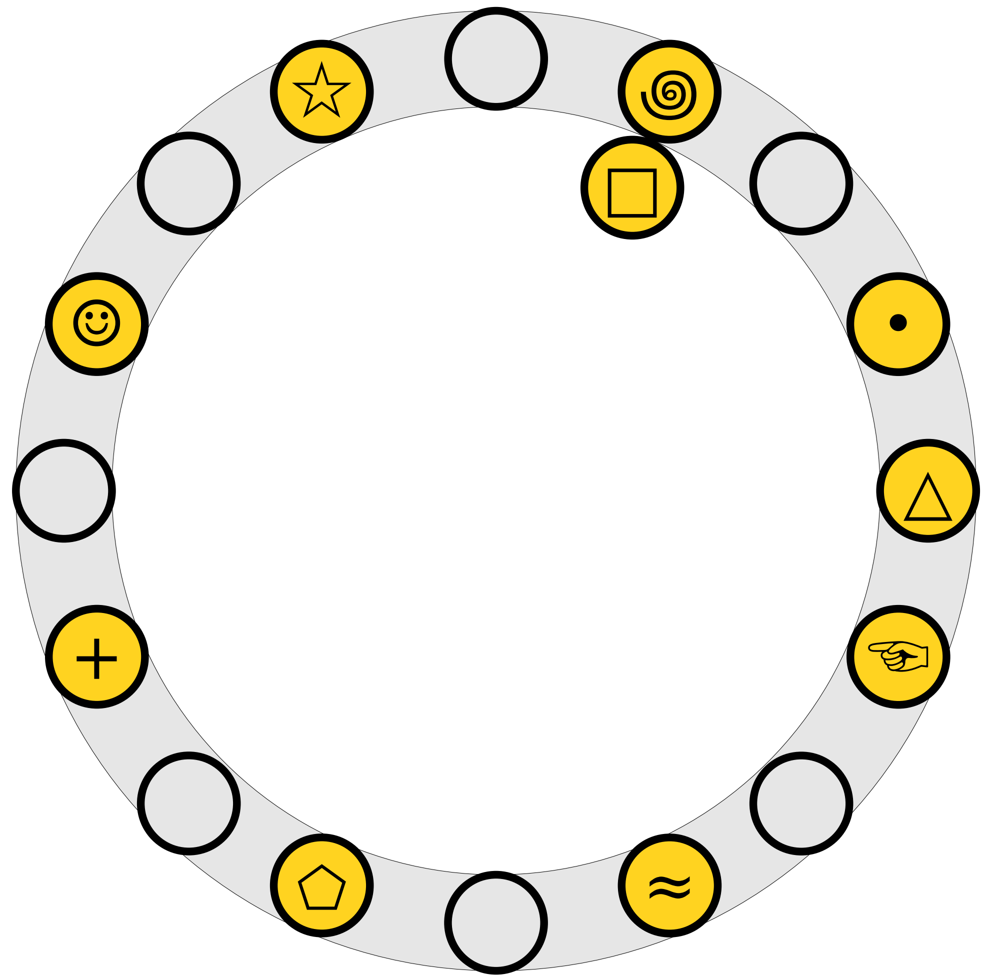
Cuckoo Hashing

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



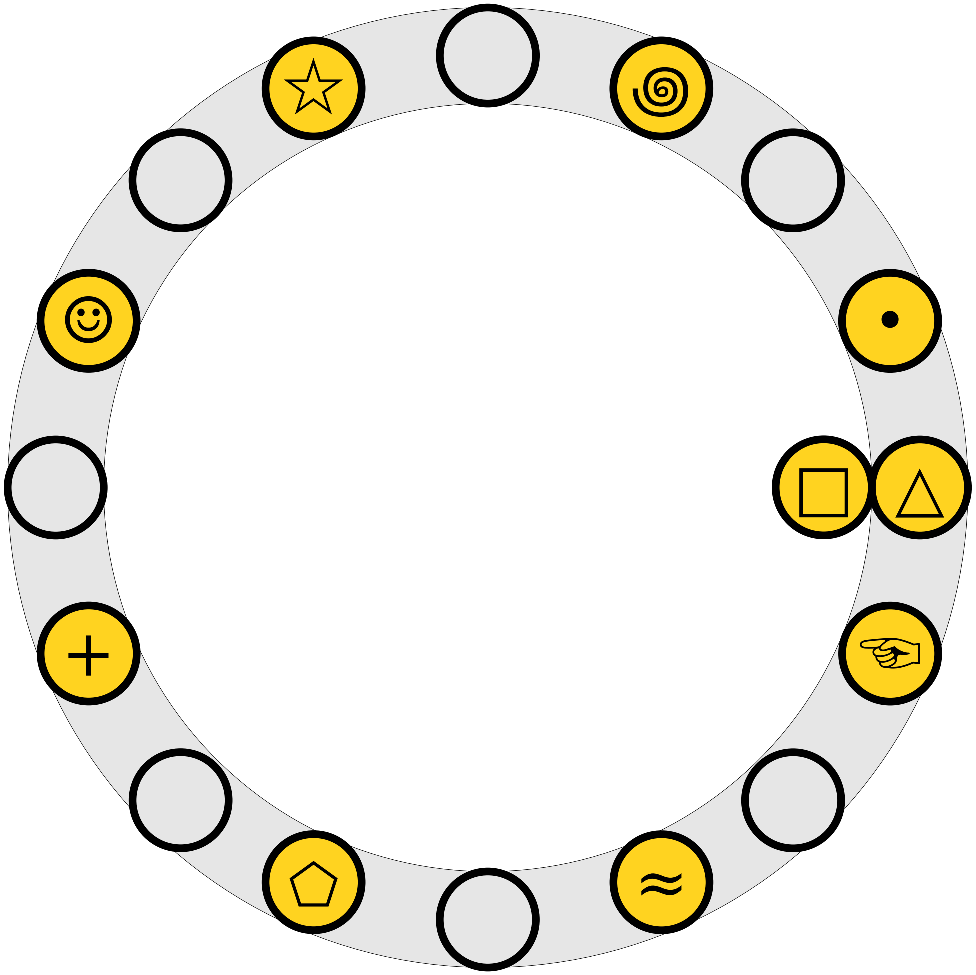
Cuckoo Hashing

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



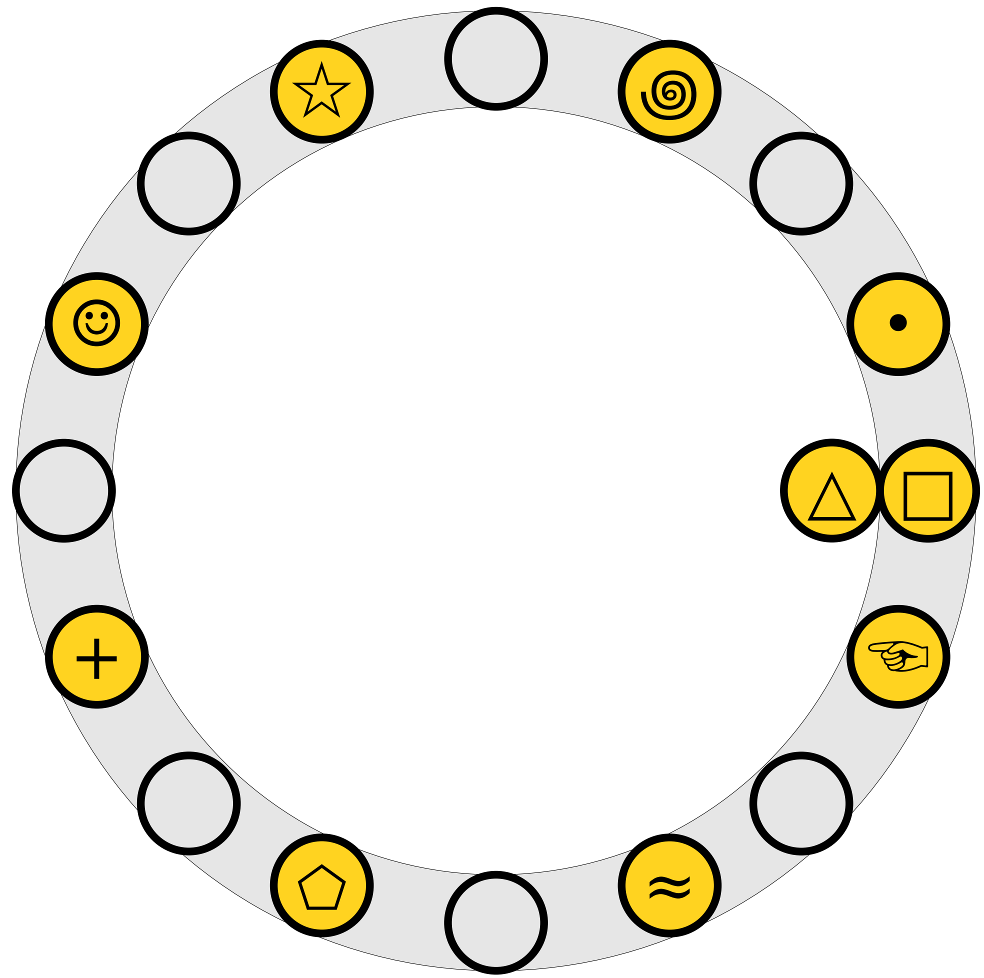
Cuckoo Hashing

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



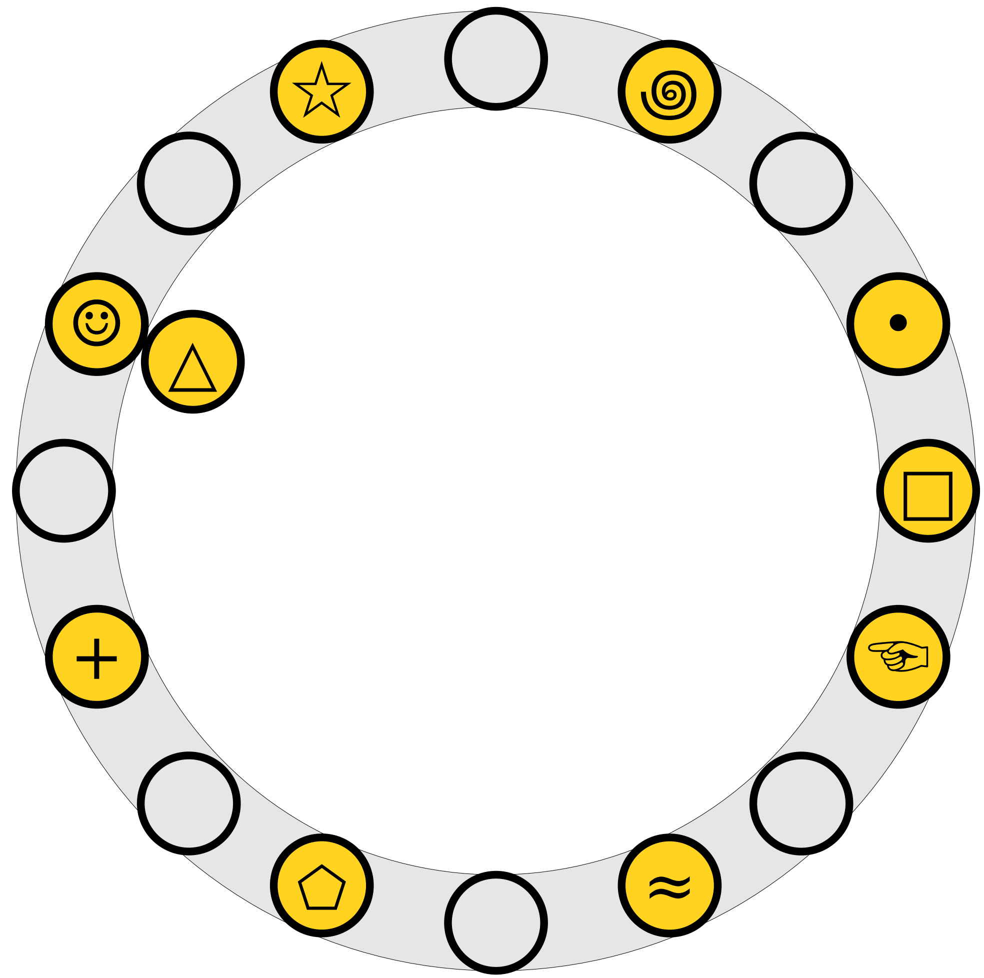
Cuckoo Hashing

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



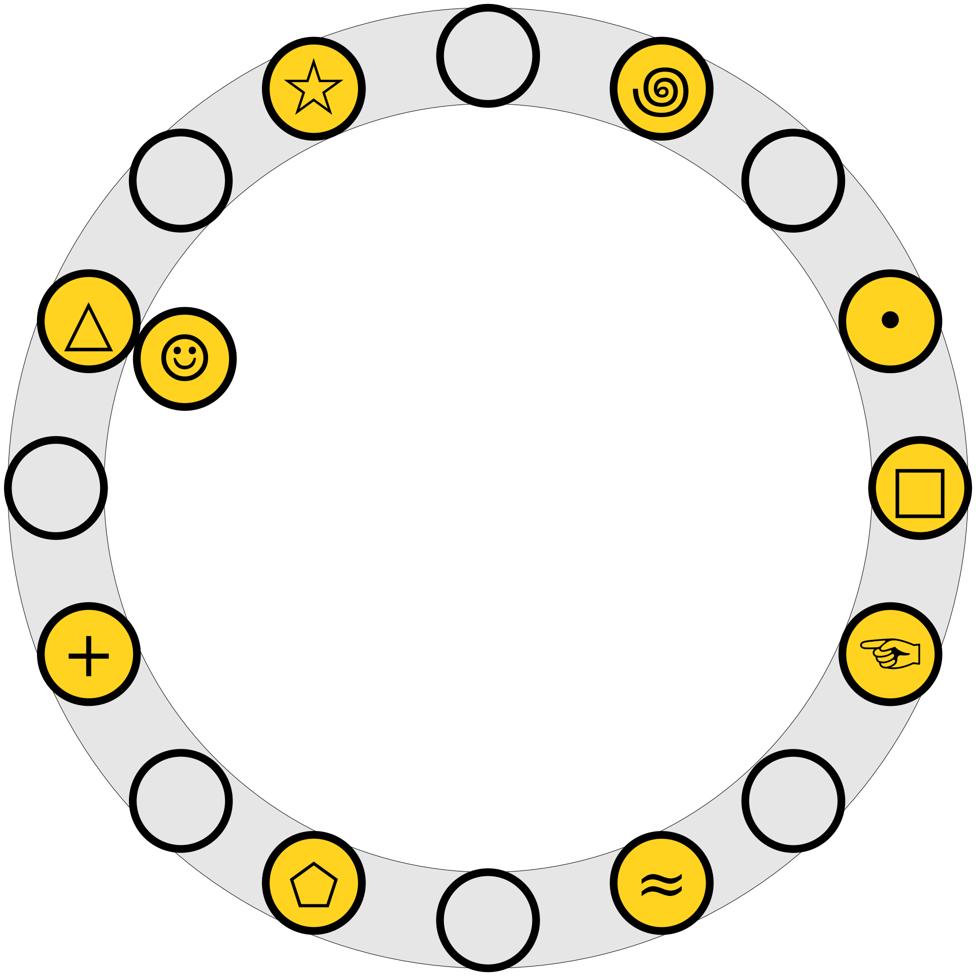
Cuckoo Hashing

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



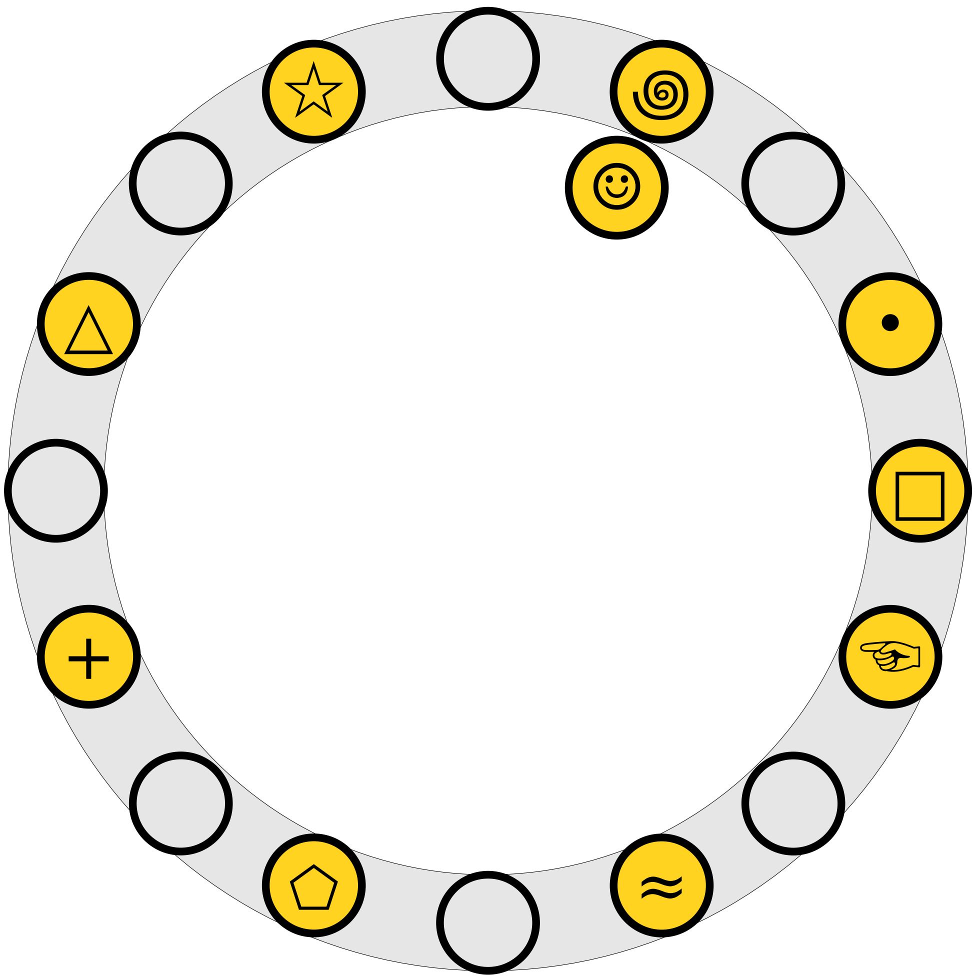
Cuckoo Hashing

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



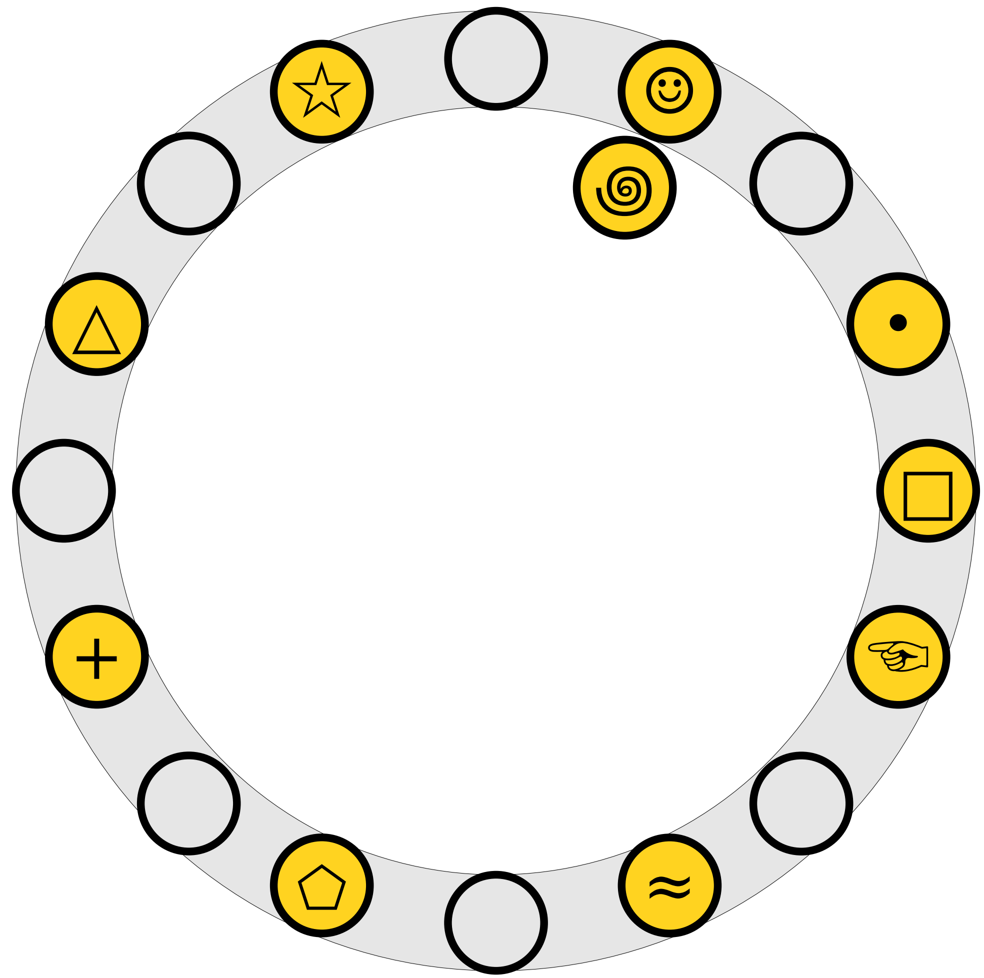
Cuckoo Hashing

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



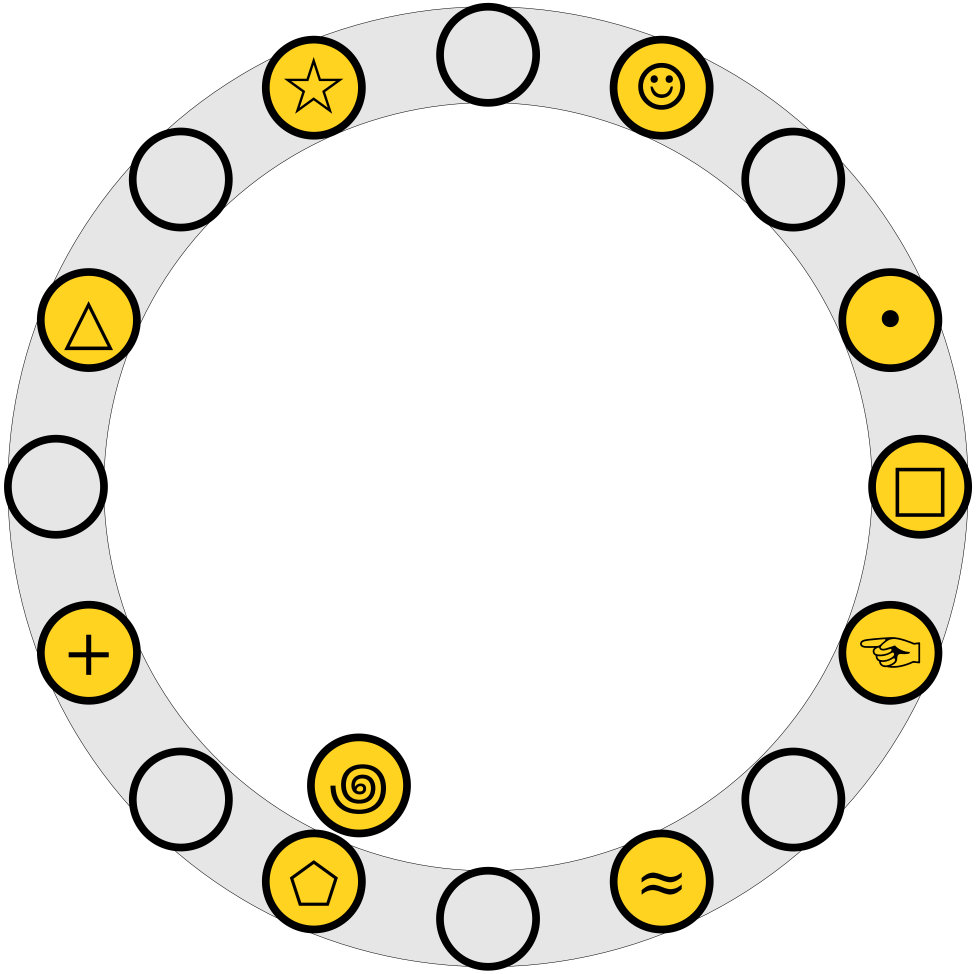
Cuckoo Hashing

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



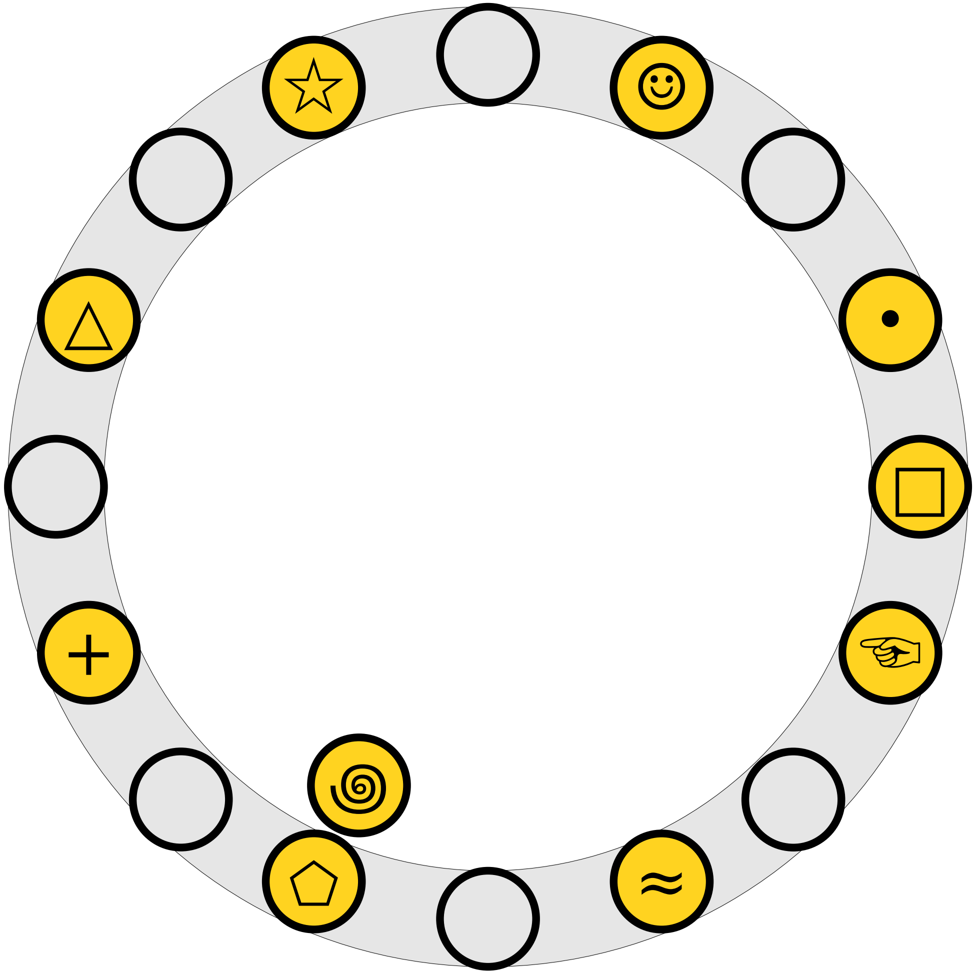
Cuckoo Hashing

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



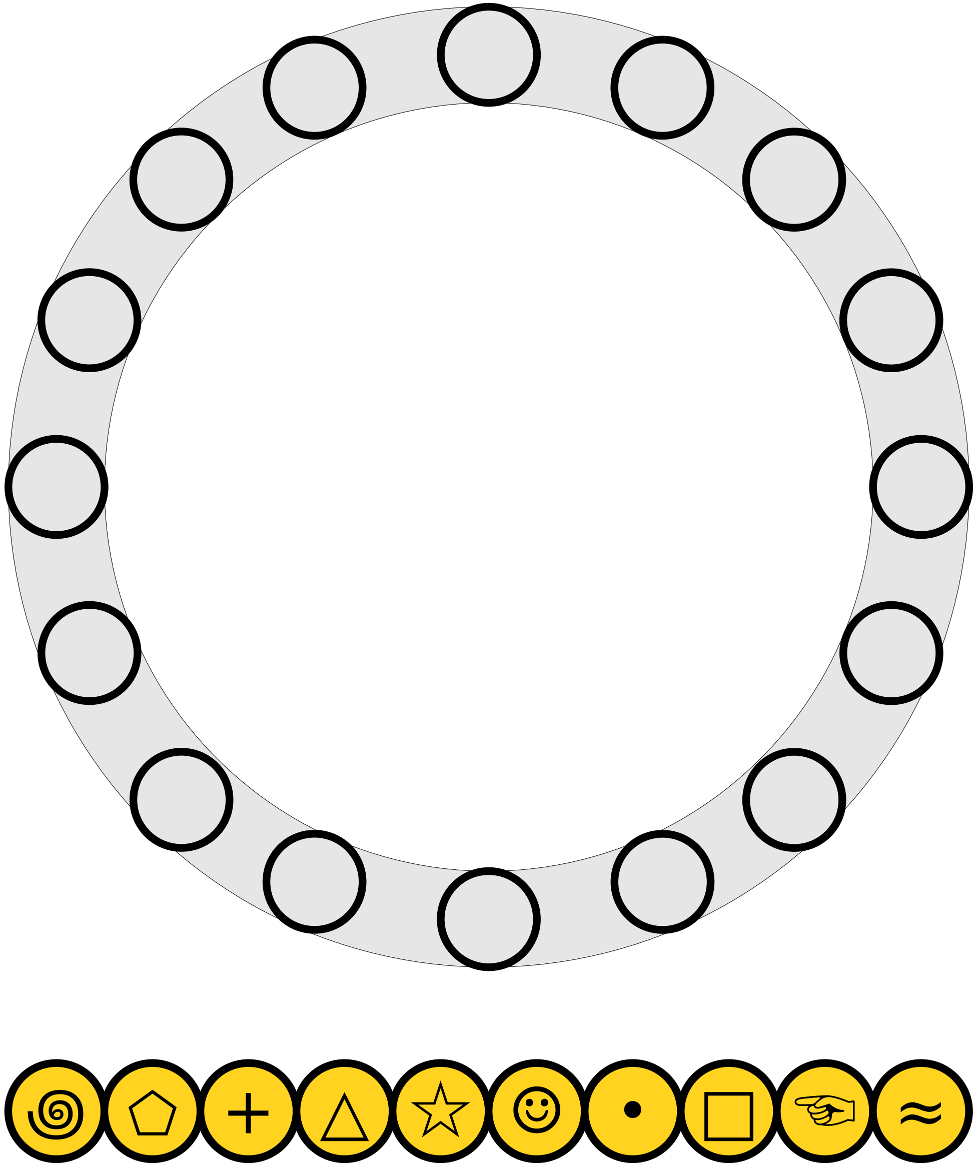
Cuckoo Hashing

- An insertion ***fails*** if the displacements form an infinite cycle.
- If that happens, perform a ***rehash*** by choosing a new h_1 and h_2 and inserting all elements back into the table.



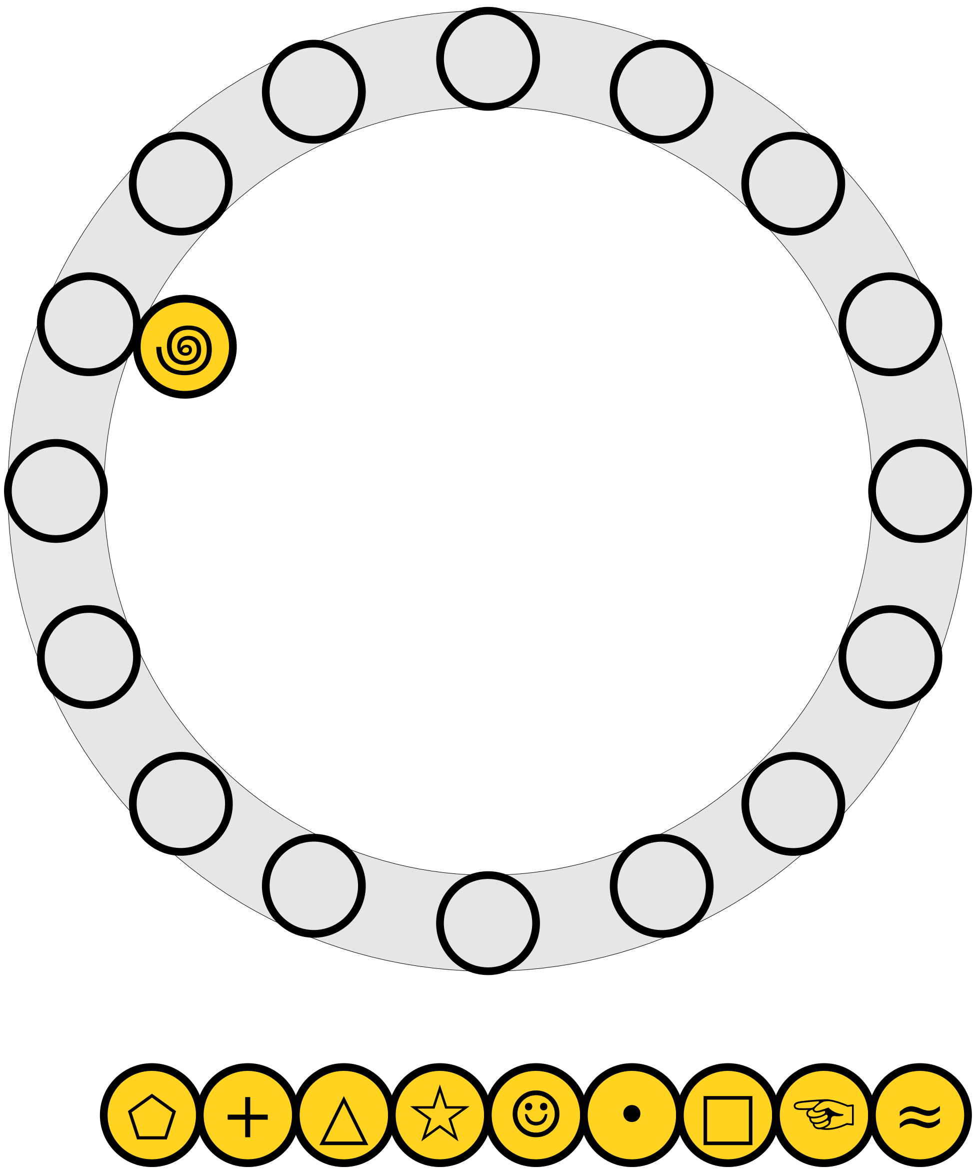
Cuckoo Hashing

- An insertion ***fails*** if the displacements form an infinite cycle.
- If that happens, perform a ***rehash*** by choosing a new h_1 and h_2 and inserting all elements back into the table.



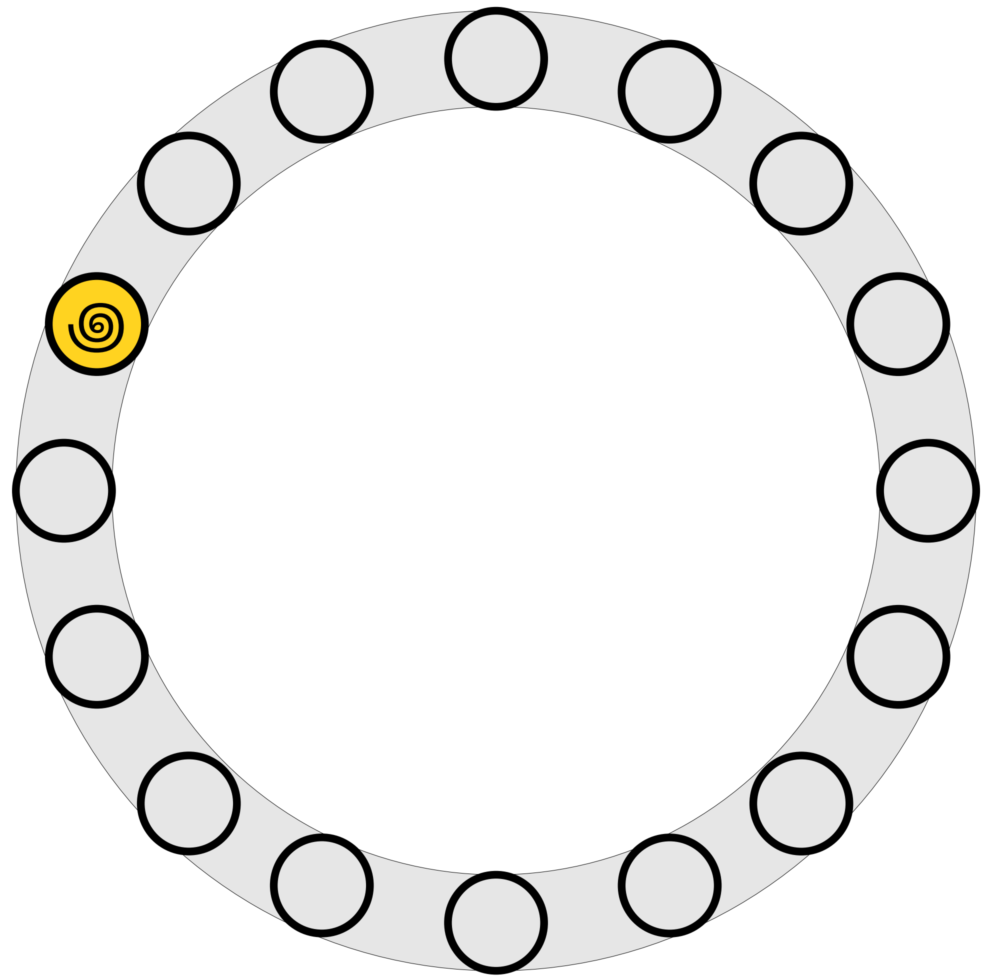
Cuckoo Hashing

- An insertion ***fails*** if the displacements form an infinite cycle.
- If that happens, perform a ***rehash*** by choosing a new h_1 and h_2 and inserting all elements back into the table.



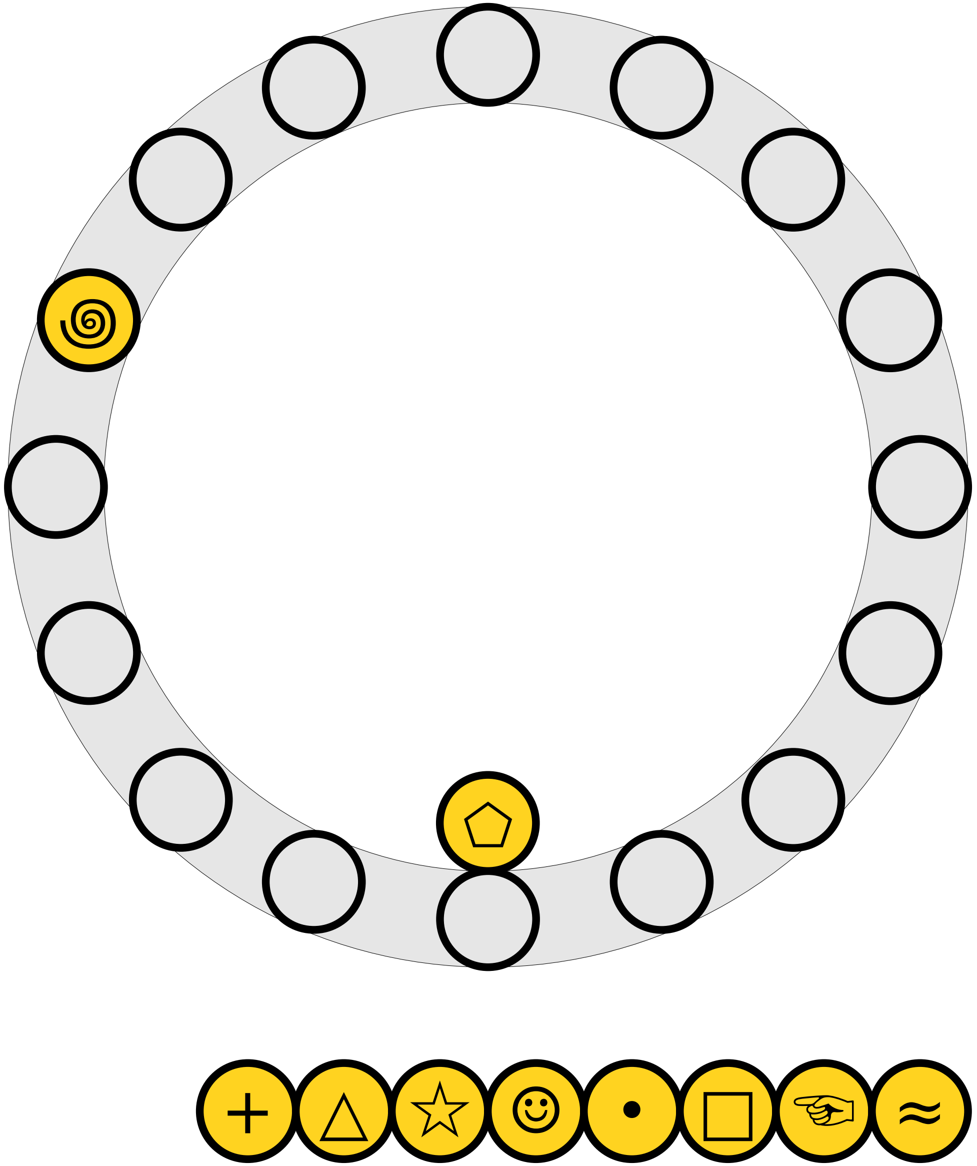
Cuckoo Hashing

- An insertion ***fails*** if the displacements form an infinite cycle.
- If that happens, perform a ***rehash*** by choosing a new h_1 and h_2 and inserting all elements back into the table.



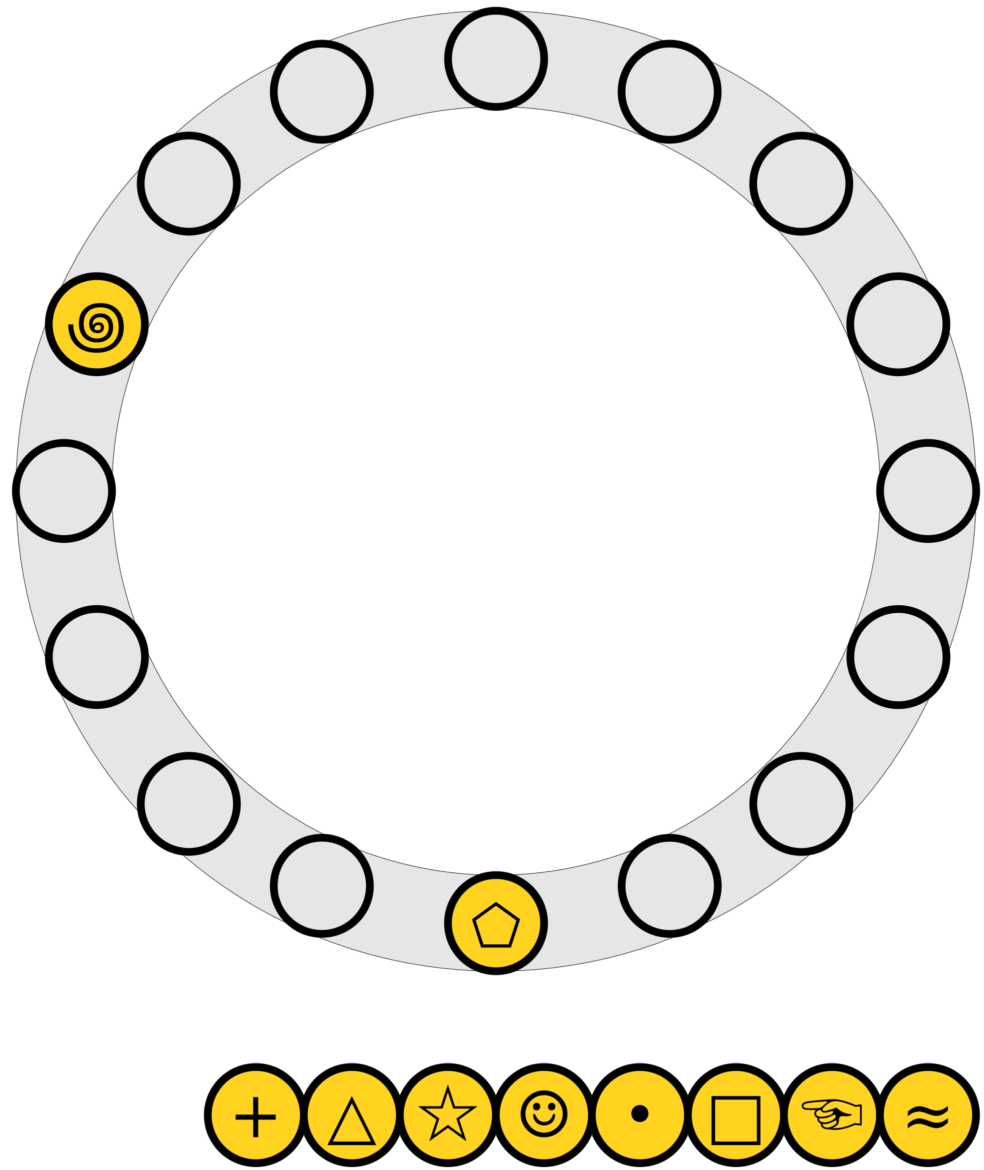
Cuckoo Hashing

- An insertion ***fails*** if the displacements form an infinite cycle.
- If that happens, perform a ***rehash*** by choosing a new h_1 and h_2 and inserting all elements back into the table.



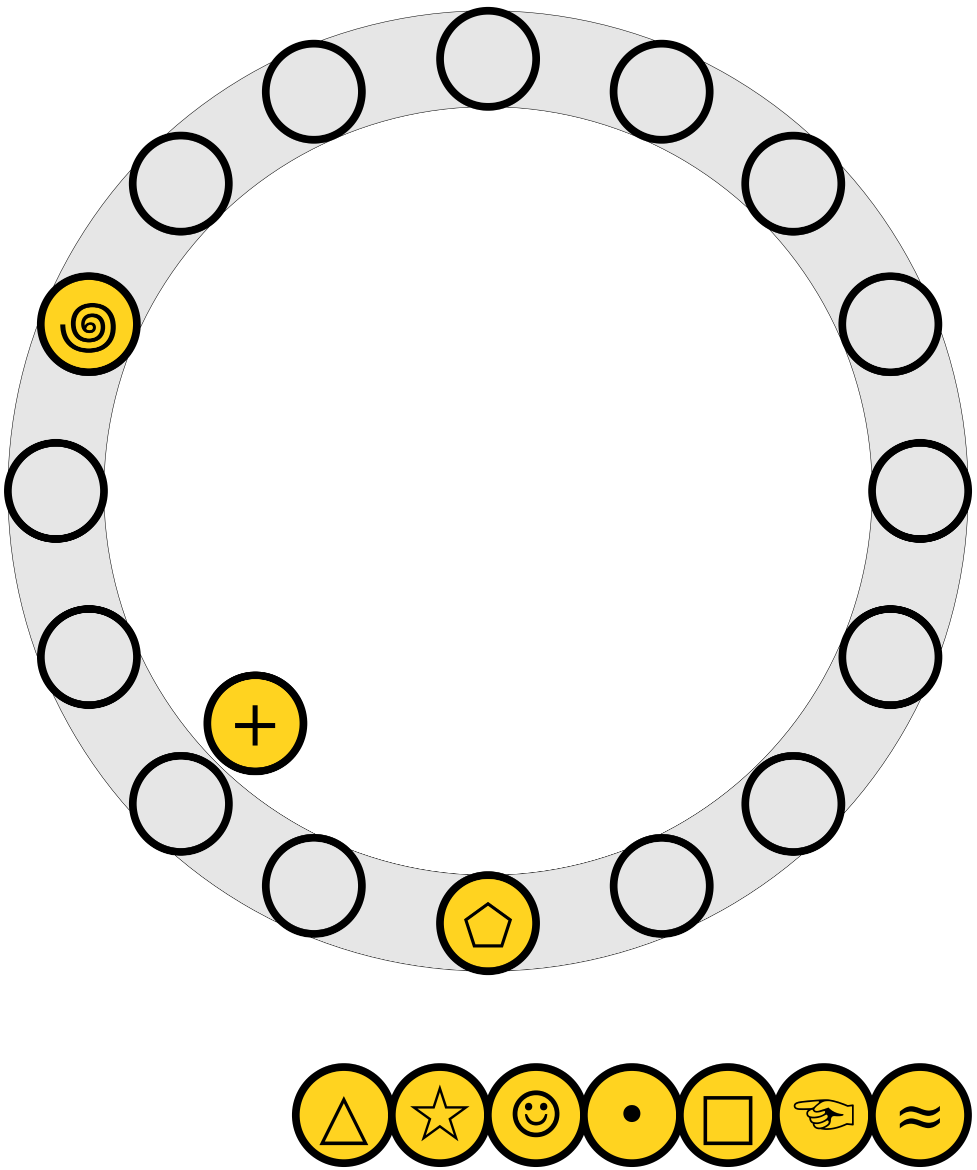
Cuckoo Hashing

- An insertion ***fails*** if the displacements form an infinite cycle.
- If that happens, perform a ***rehash*** by choosing a new h_1 and h_2 and inserting all elements back into the table.



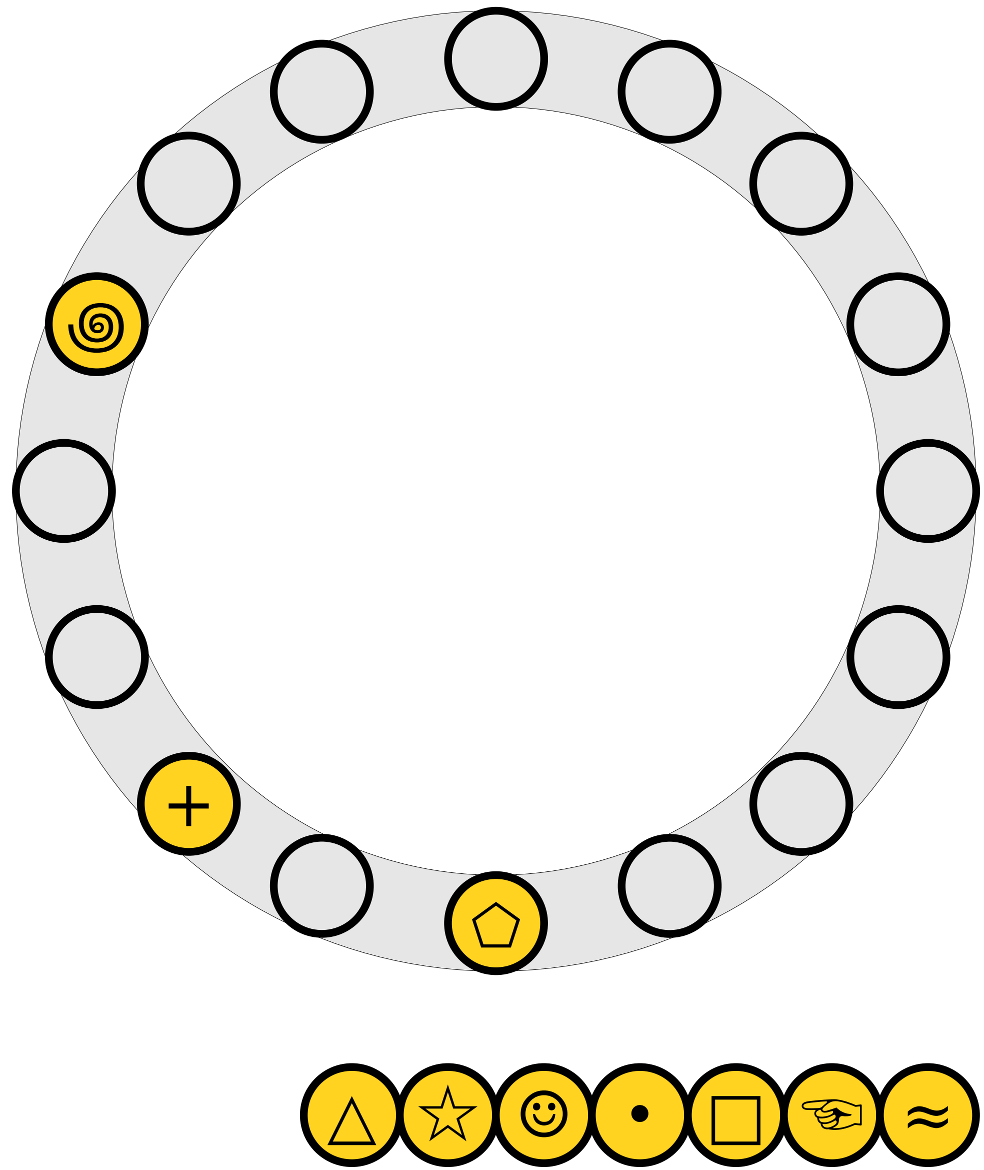
Cuckoo Hashing

- An insertion ***fails*** if the displacements form an infinite cycle.
- If that happens, perform a ***rehash*** by choosing a new h_1 and h_2 and inserting all elements back into the table.



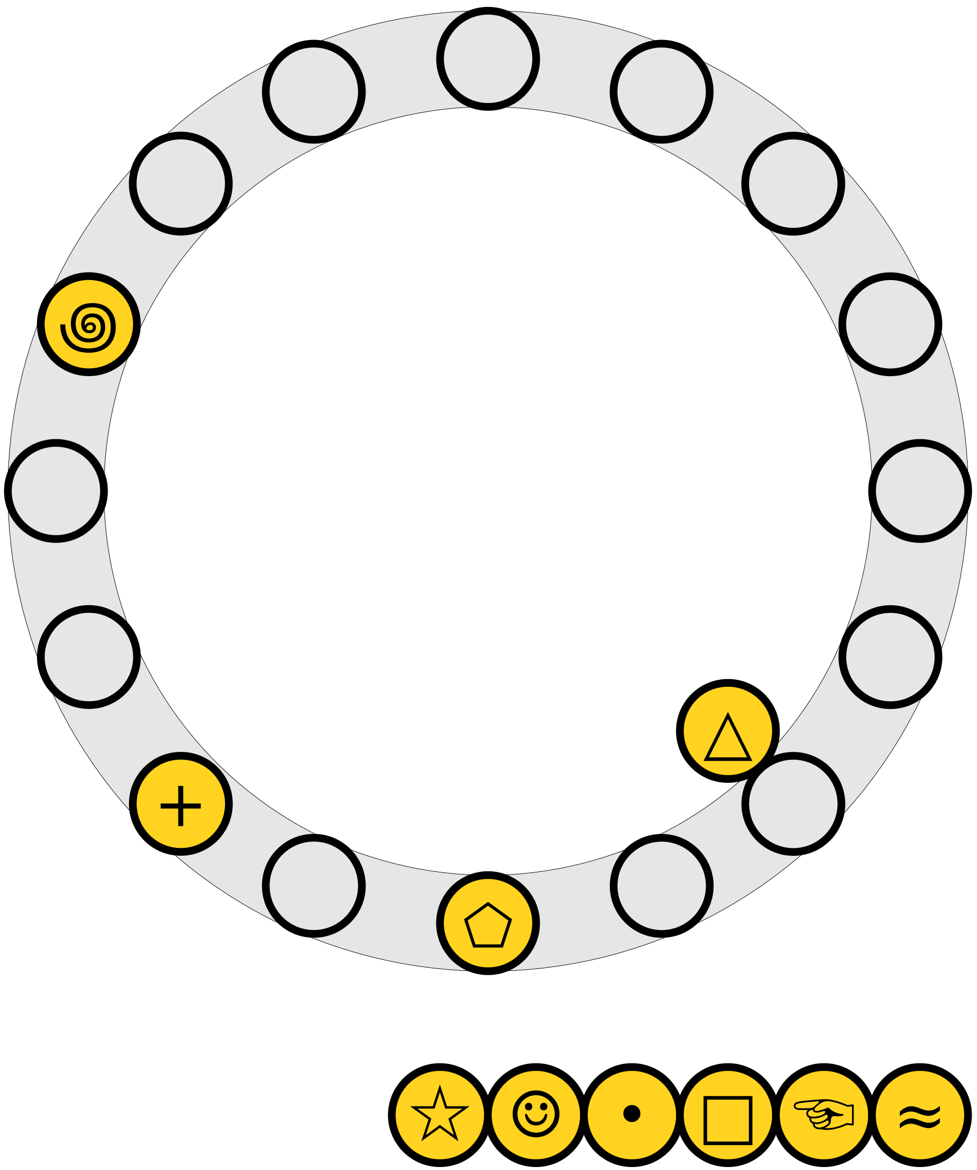
Cuckoo Hashing

- An insertion ***fails*** if the displacements form an infinite cycle.
- If that happens, perform a ***rehash*** by choosing a new h_1 and h_2 and inserting all elements back into the table.



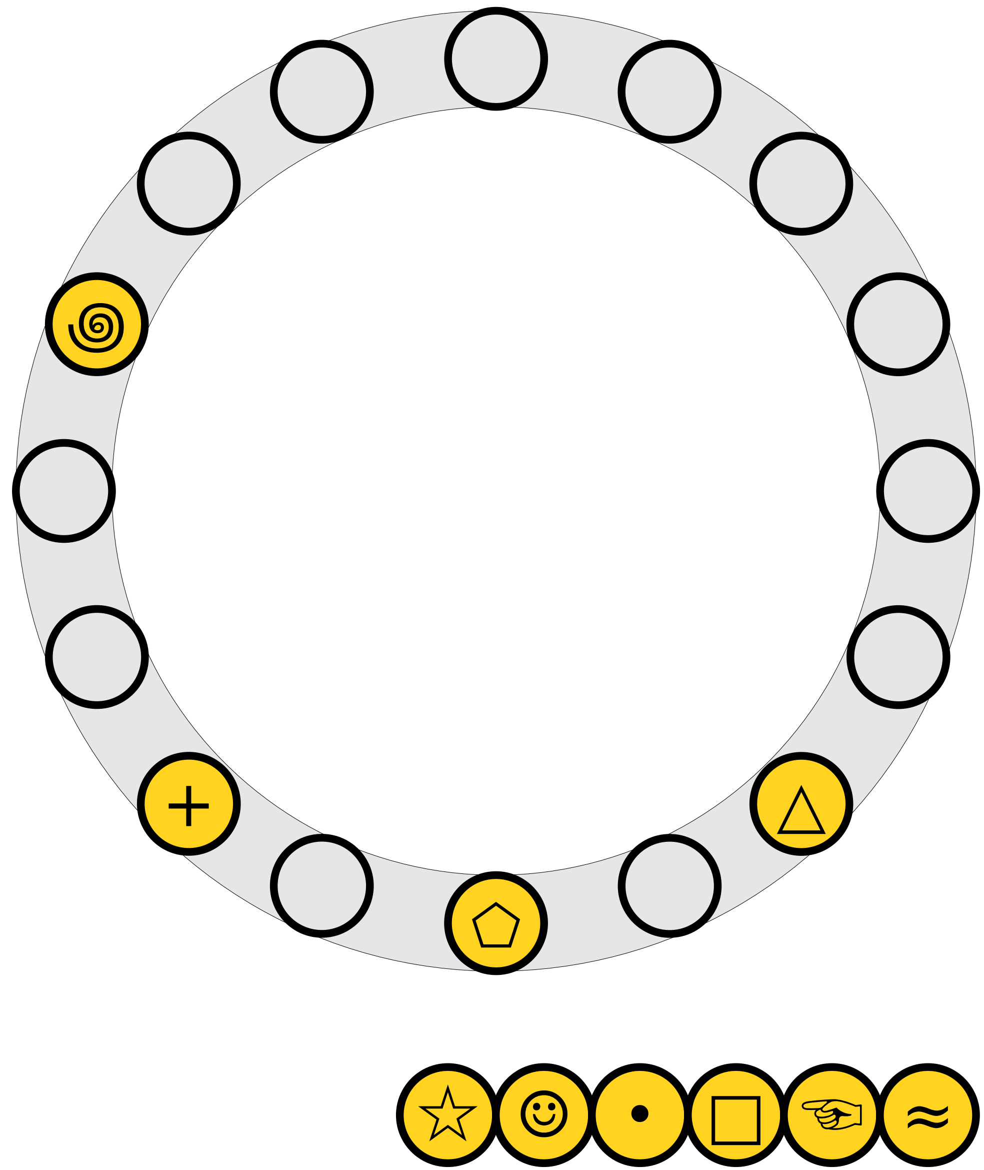
Cuckoo Hashing

- An insertion ***fails*** if the displacements form an infinite cycle.
- If that happens, perform a ***rehash*** by choosing a new h_1 and h_2 and inserting all elements back into the table.



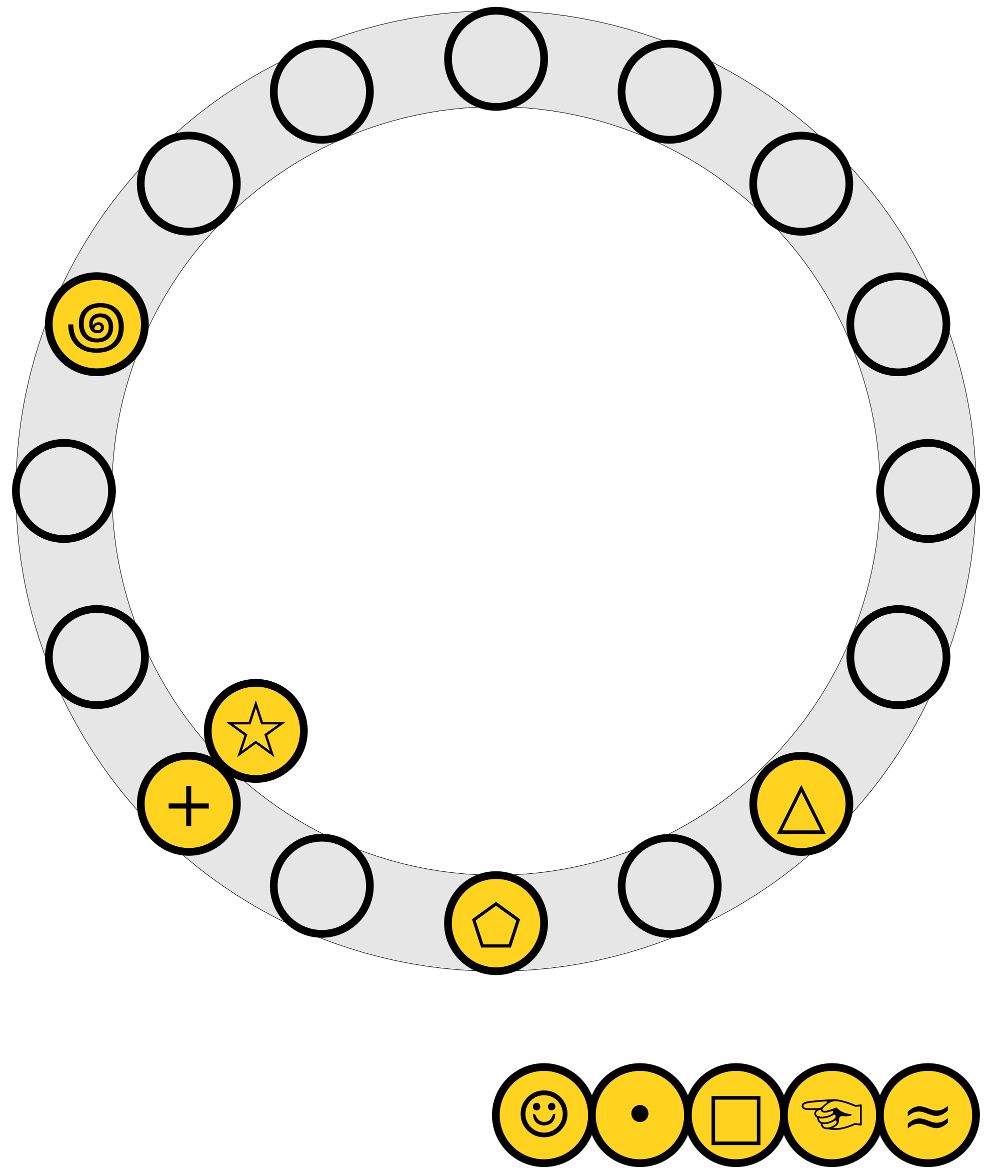
Cuckoo Hashing

- An insertion ***fails*** if the displacements form an infinite cycle.
- If that happens, perform a ***rehash*** by choosing a new h_1 and h_2 and inserting all elements back into the table.



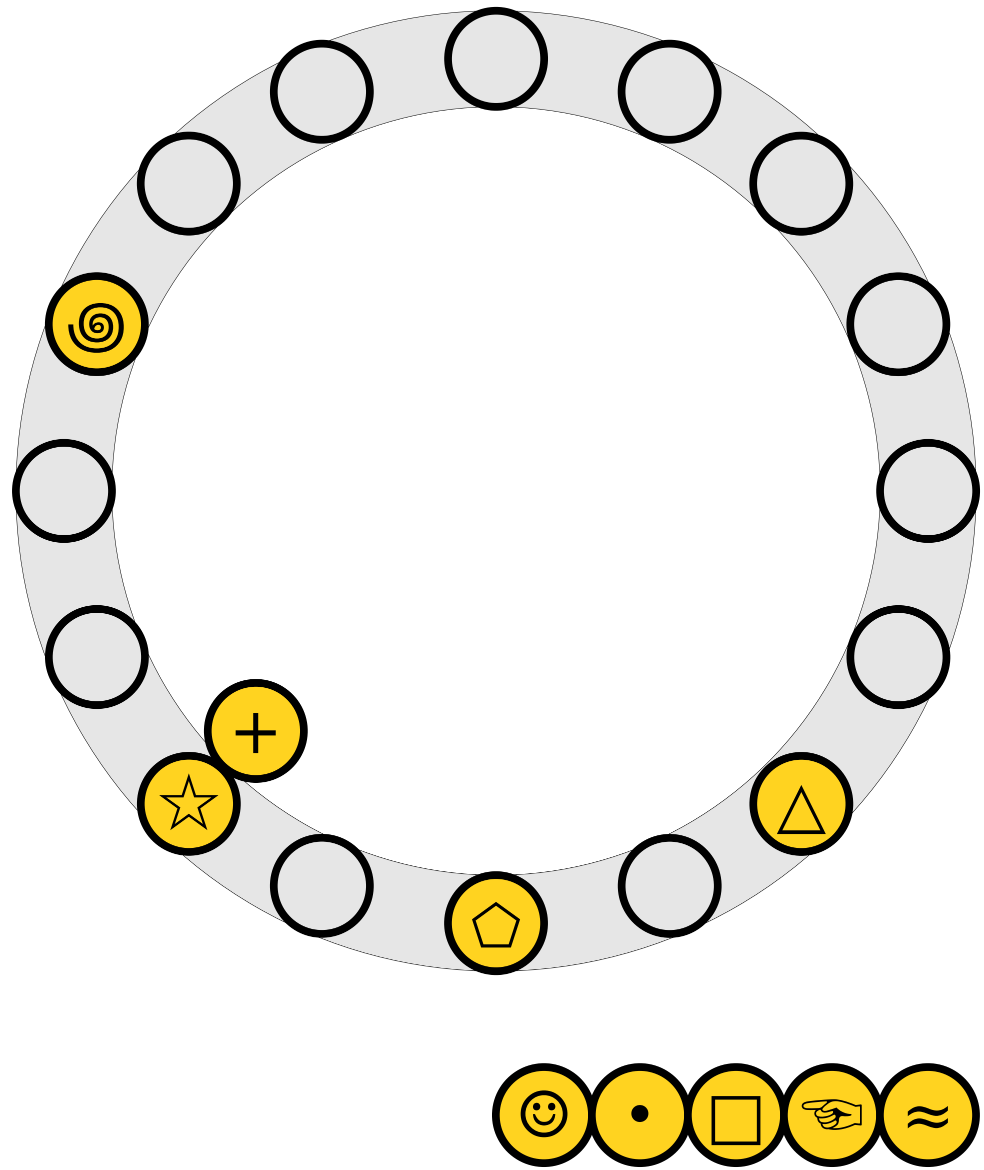
Cuckoo Hashing

- An insertion ***fails*** if the displacements form an infinite cycle.
- If that happens, perform a ***rehash*** by choosing a new h_1 and h_2 and inserting all elements back into the table.



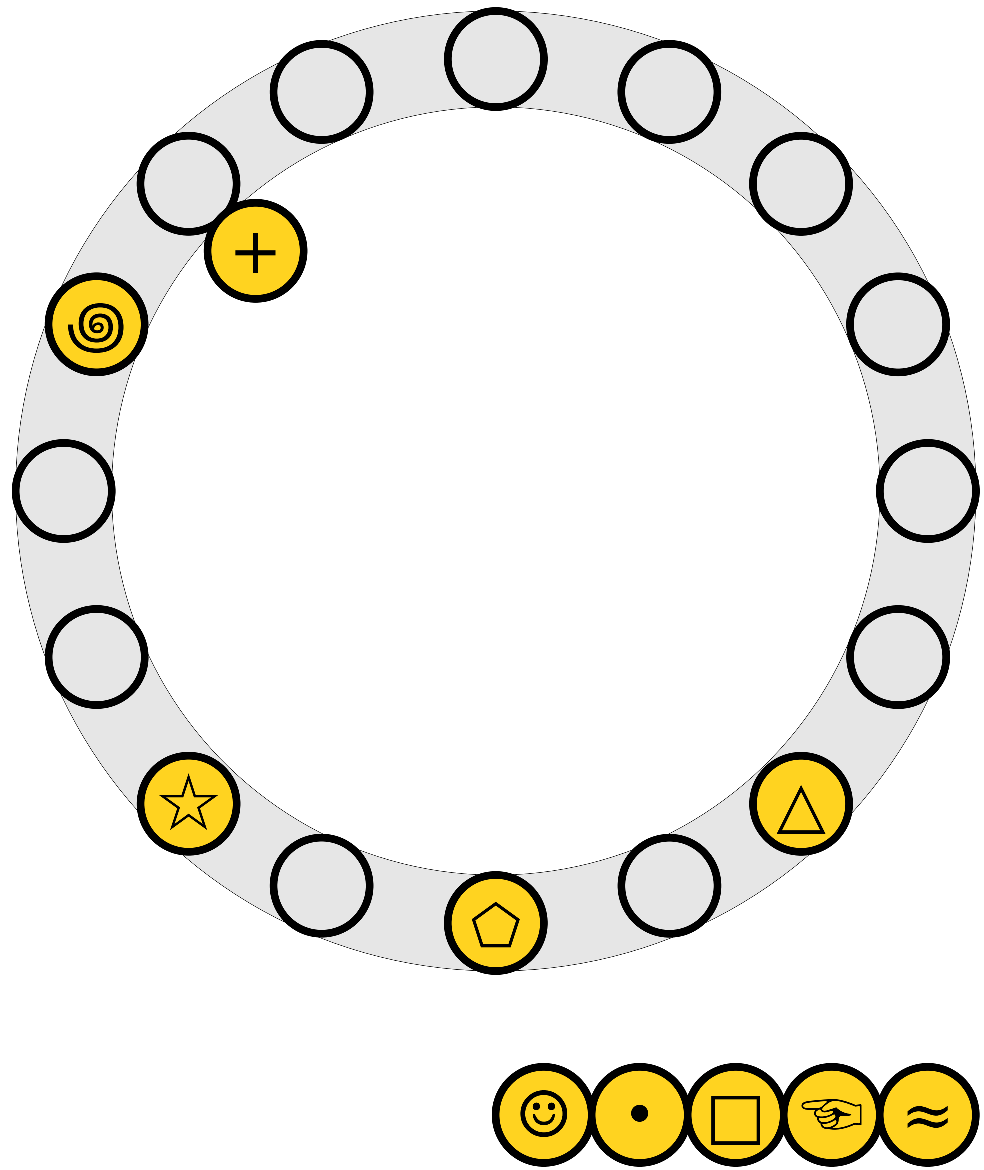
Cuckoo Hashing

- An insertion ***fails*** if the displacements form an infinite cycle.
- If that happens, perform a ***rehash*** by choosing a new h_1 and h_2 and inserting all elements back into the table.



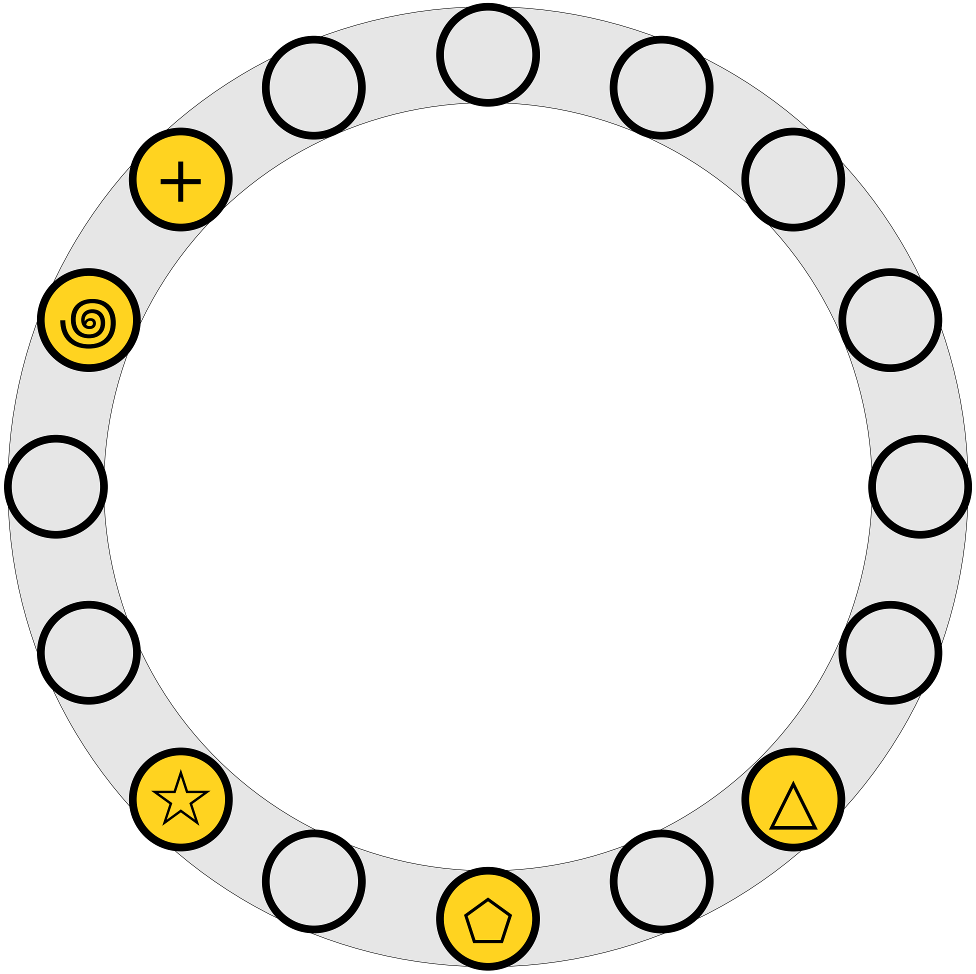
Cuckoo Hashing

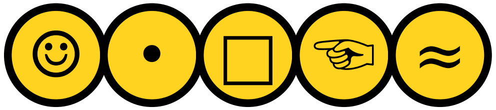
- An insertion ***fails*** if the displacements form an infinite cycle.
- If that happens, perform a ***rehash*** by choosing a new h_1 and h_2 and inserting all elements back into the table.



Cuckoo Hashing

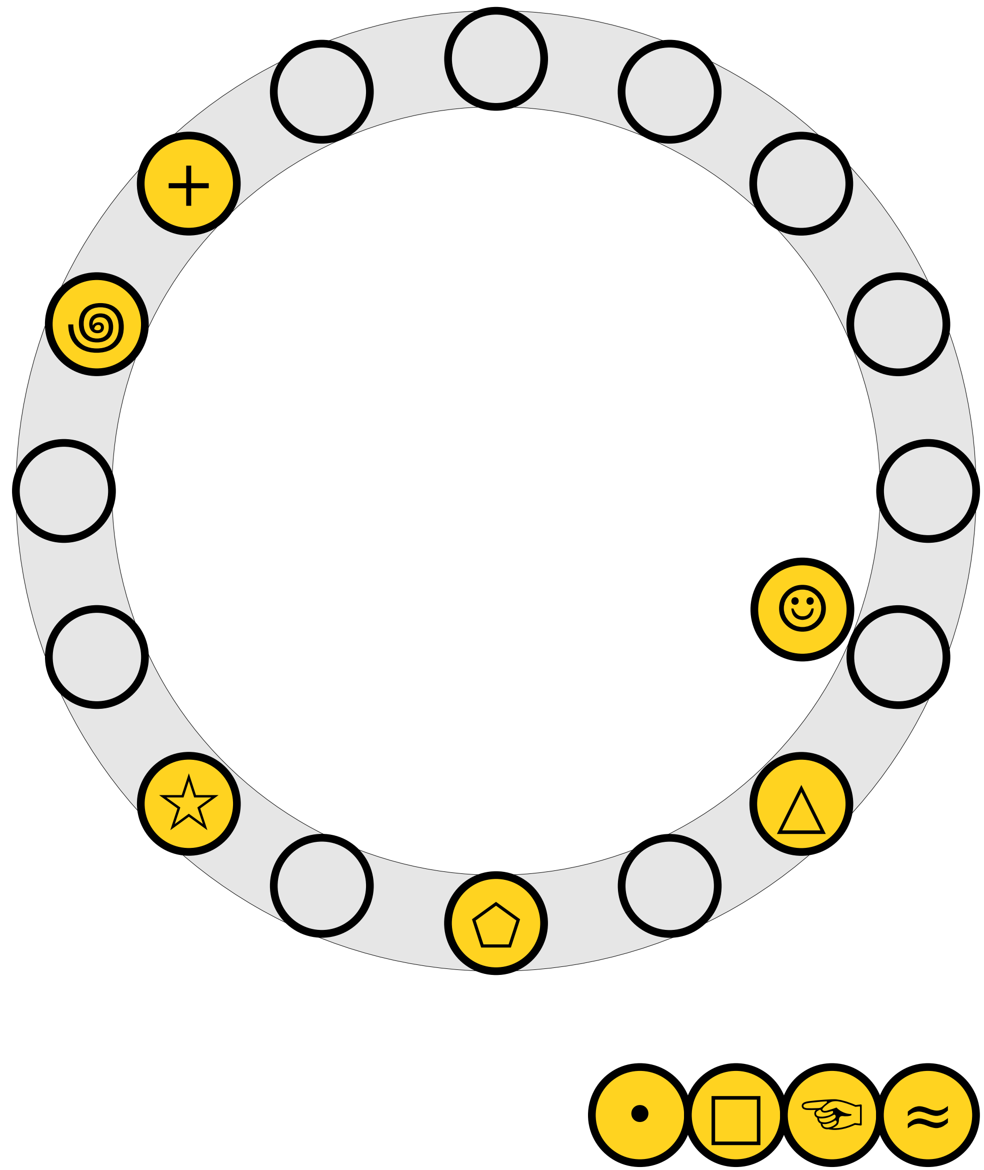
- An insertion ***fails*** if the displacements form an infinite cycle.
- If that happens, perform a ***rehash*** by choosing a new h_1 and h_2 and inserting all elements back into the table.





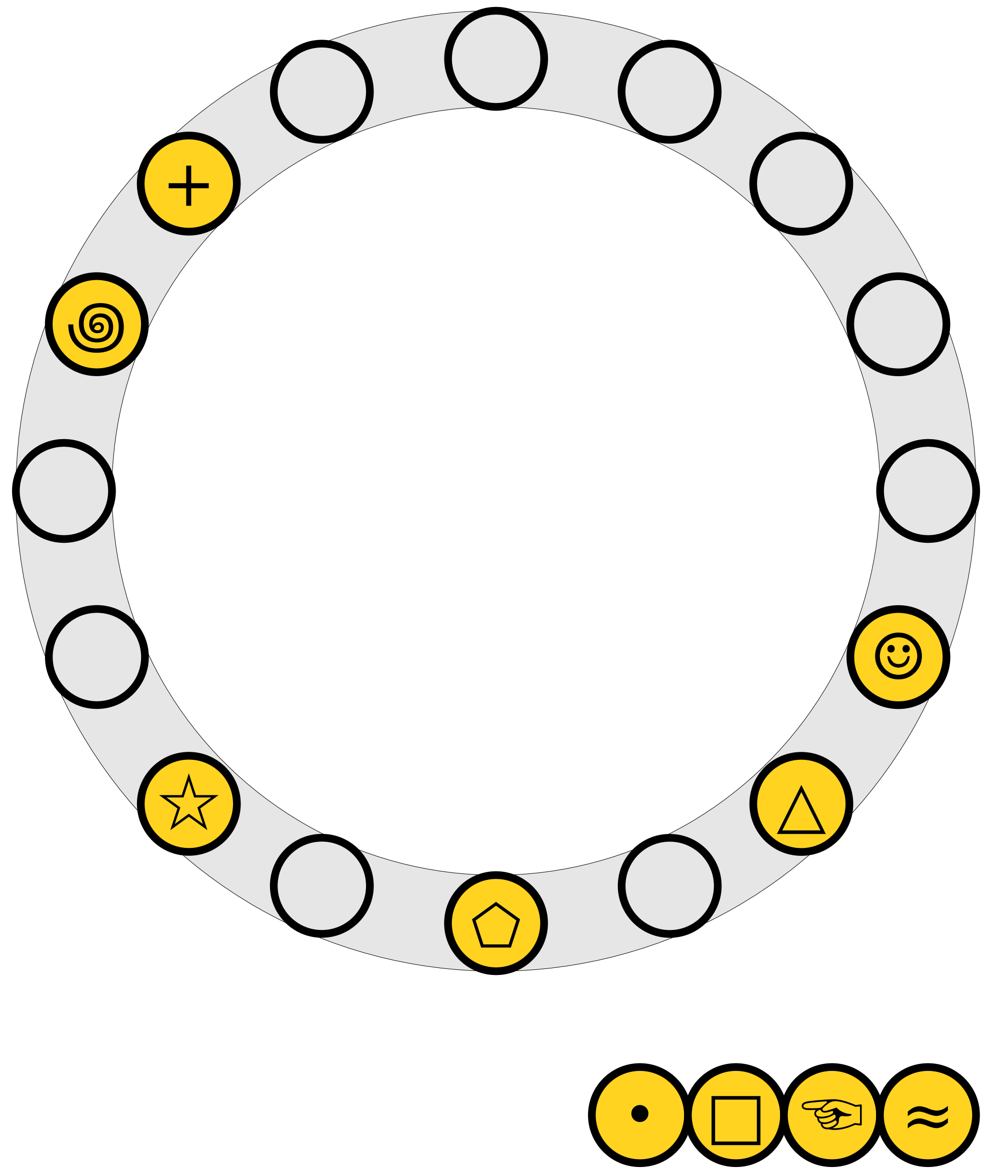
Cuckoo Hashing

- An insertion ***fails*** if the displacements form an infinite cycle.
- If that happens, perform a ***rehash*** by choosing a new h_1 and h_2 and inserting all elements back into the table.



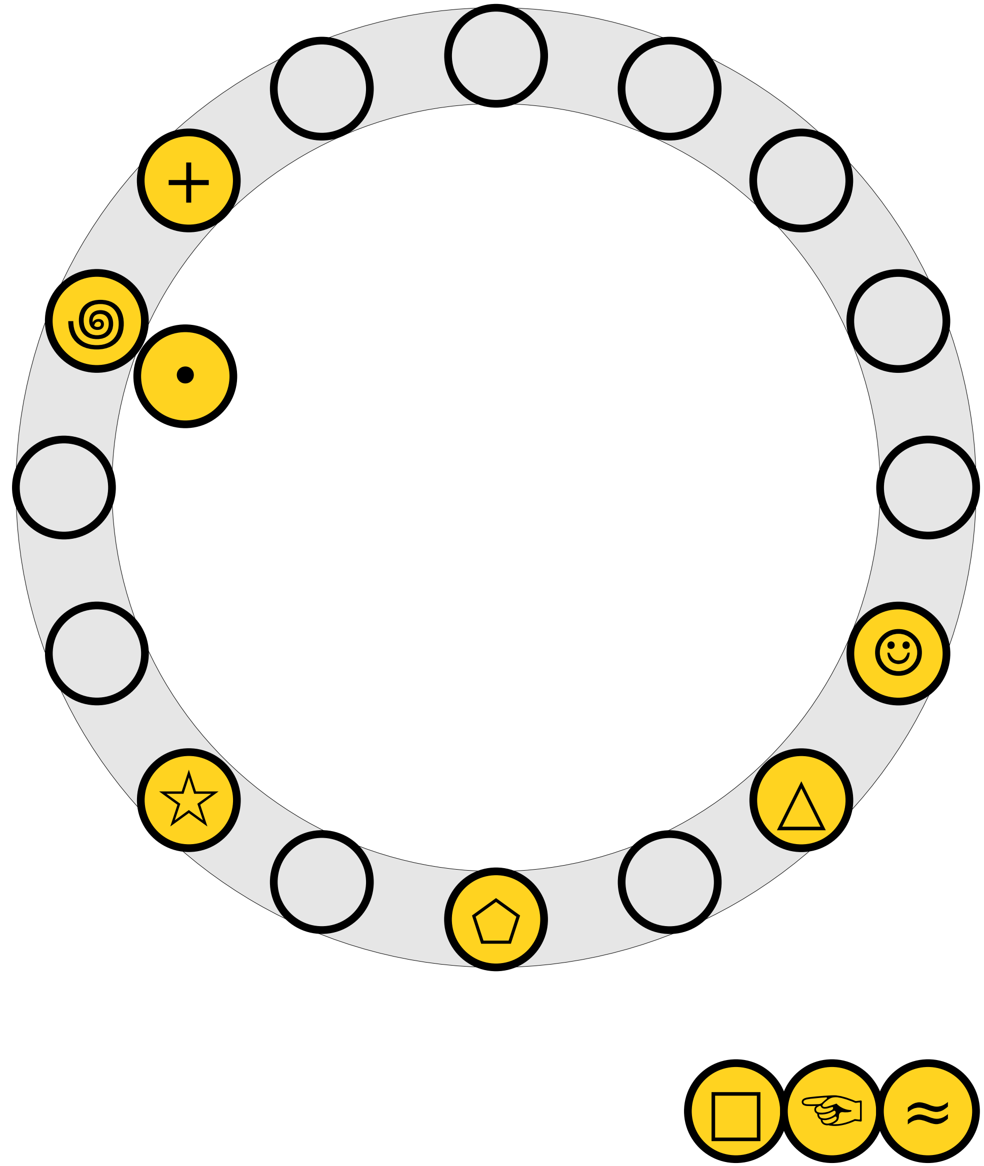
Cuckoo Hashing

- An insertion ***fails*** if the displacements form an infinite cycle.
- If that happens, perform a ***rehash*** by choosing a new h_1 and h_2 and inserting all elements back into the table.



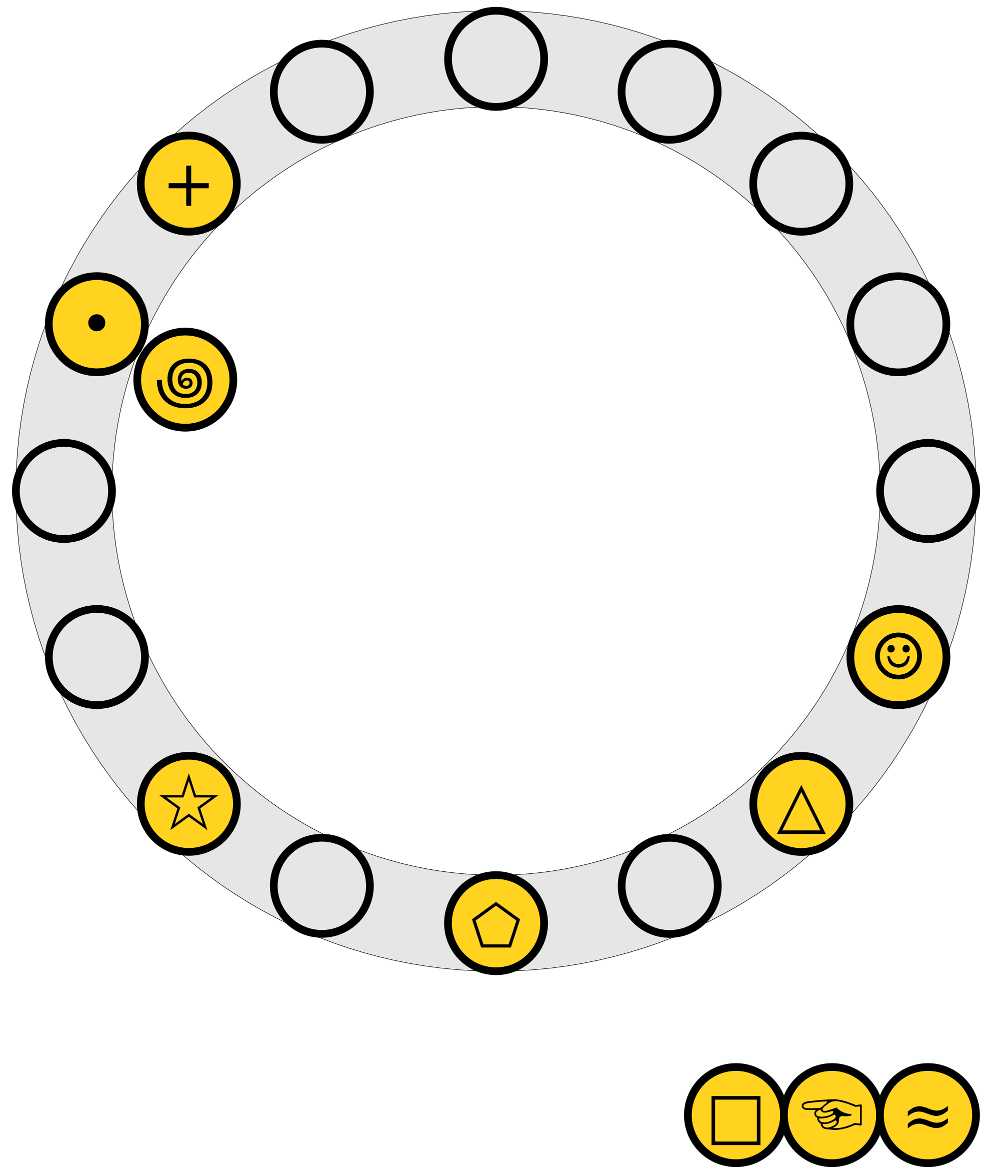
Cuckoo Hashing

- An insertion ***fails*** if the displacements form an infinite cycle.
- If that happens, perform a ***rehash*** by choosing a new h_1 and h_2 and inserting all elements back into the table.



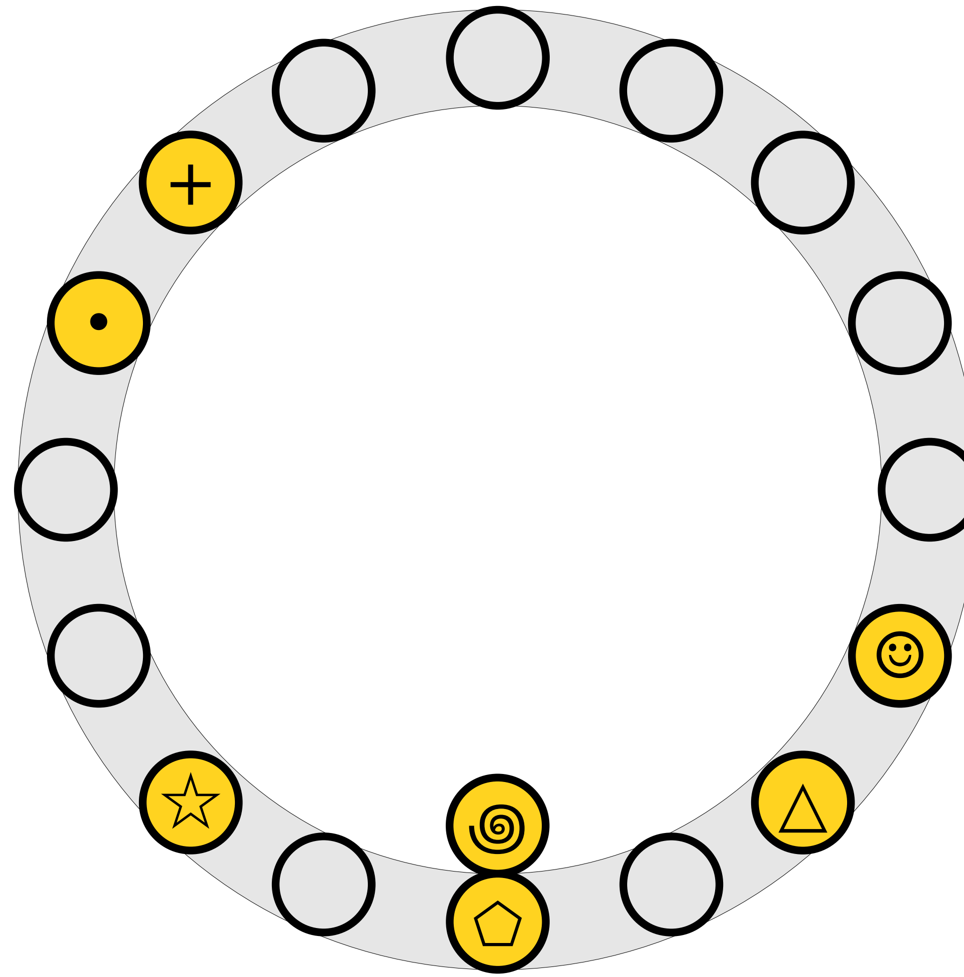
Cuckoo Hashing

- An insertion ***fails*** if the displacements form an infinite cycle.
- If that happens, perform a ***rehash*** by choosing a new h_1 and h_2 and inserting all elements back into the table.



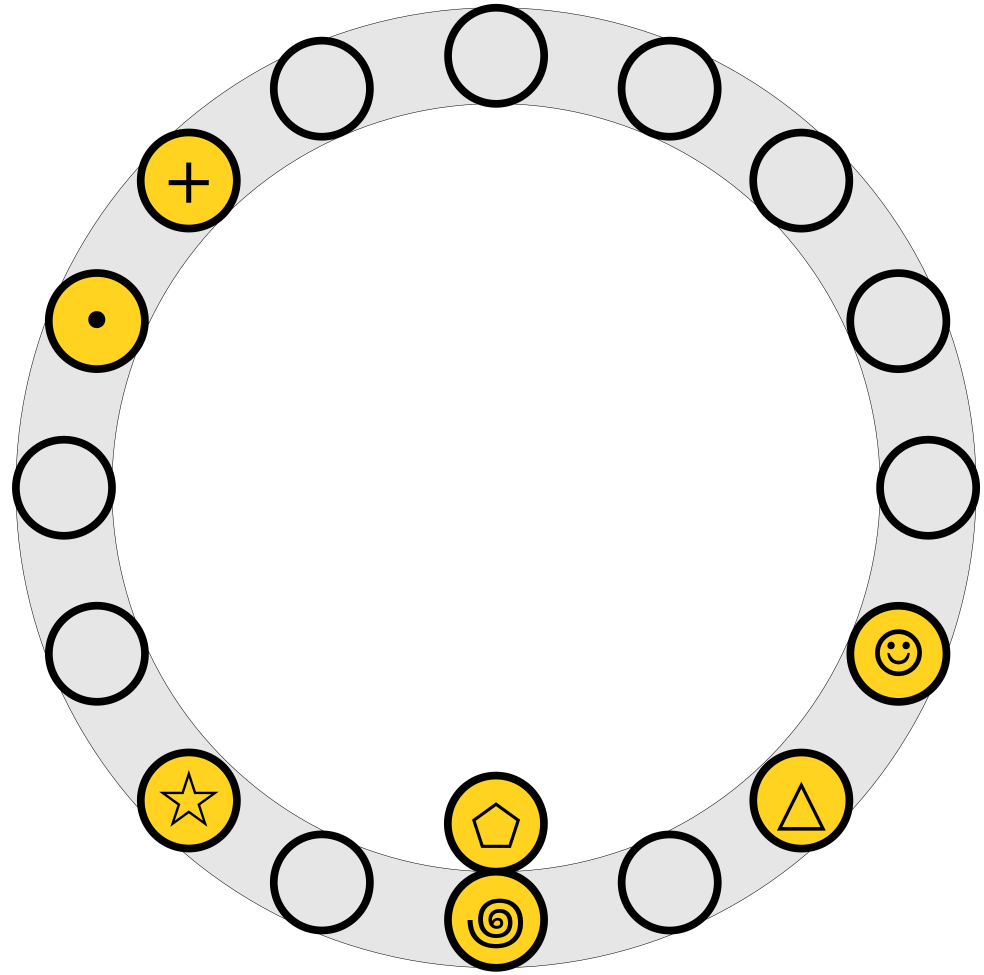
Cuckoo Hashing

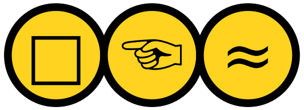
- An insertion ***fails*** if the displacements form an infinite cycle.
- If that happens, perform a ***rehash*** by choosing a new h_1 and h_2 and inserting all elements back into the table.



Cuckoo Hashing

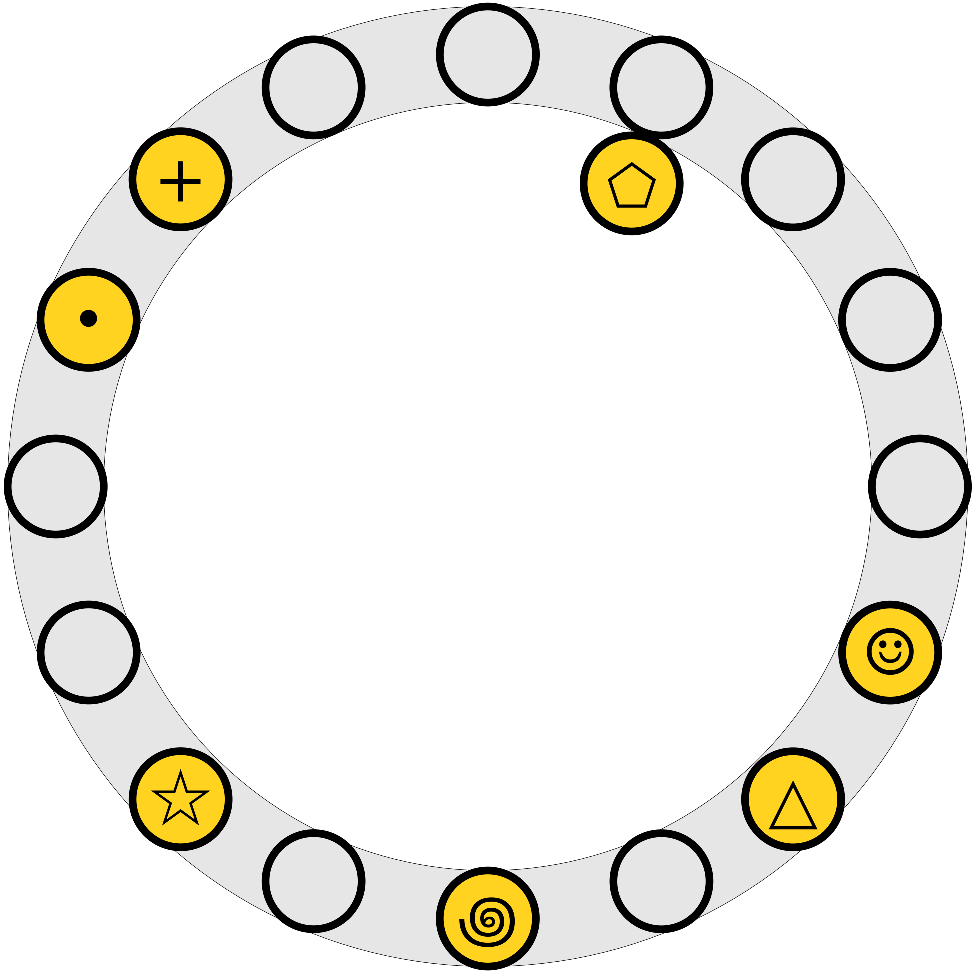
- An insertion ***fails*** if the displacements form an infinite cycle.
- If that happens, perform a ***rehash*** by choosing a new h_1 and h_2 and inserting all elements back into the table.

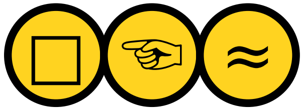




Cuckoo Hashing

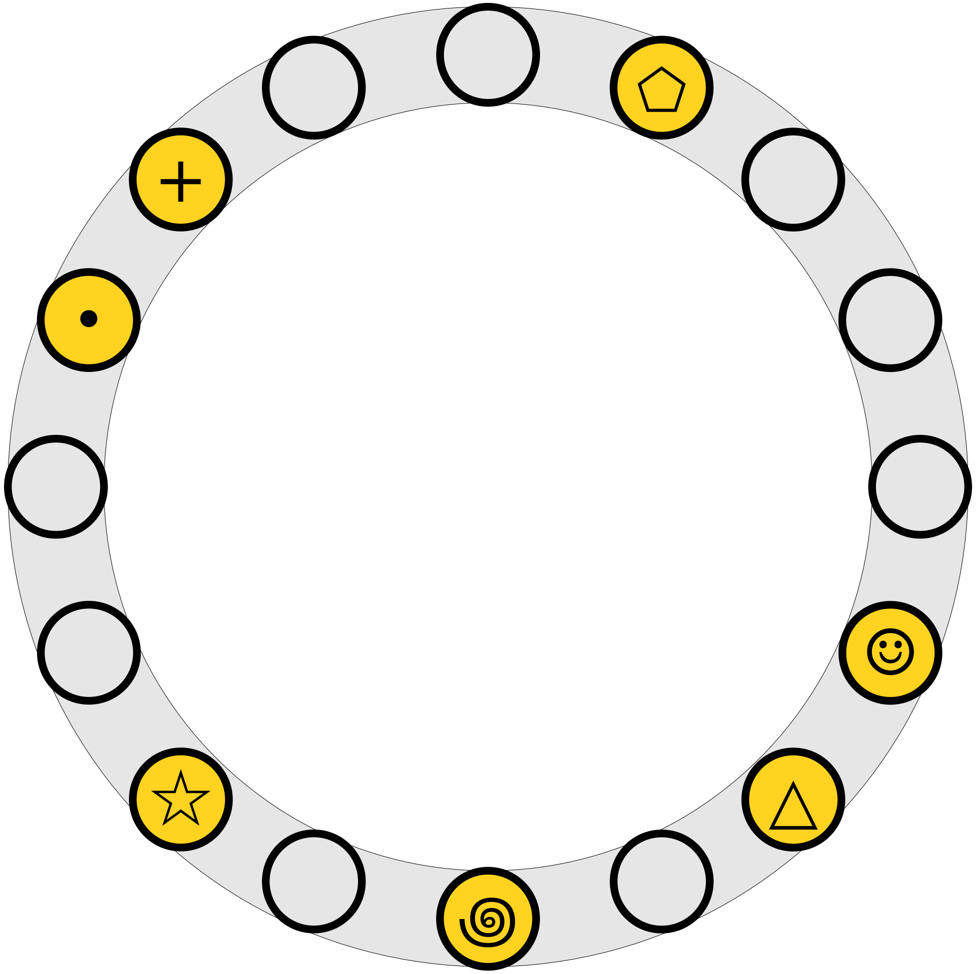
- An insertion ***fails*** if the displacements form an infinite cycle.
- If that happens, perform a ***rehash*** by choosing a new h_1 and h_2 and inserting all elements back into the table.





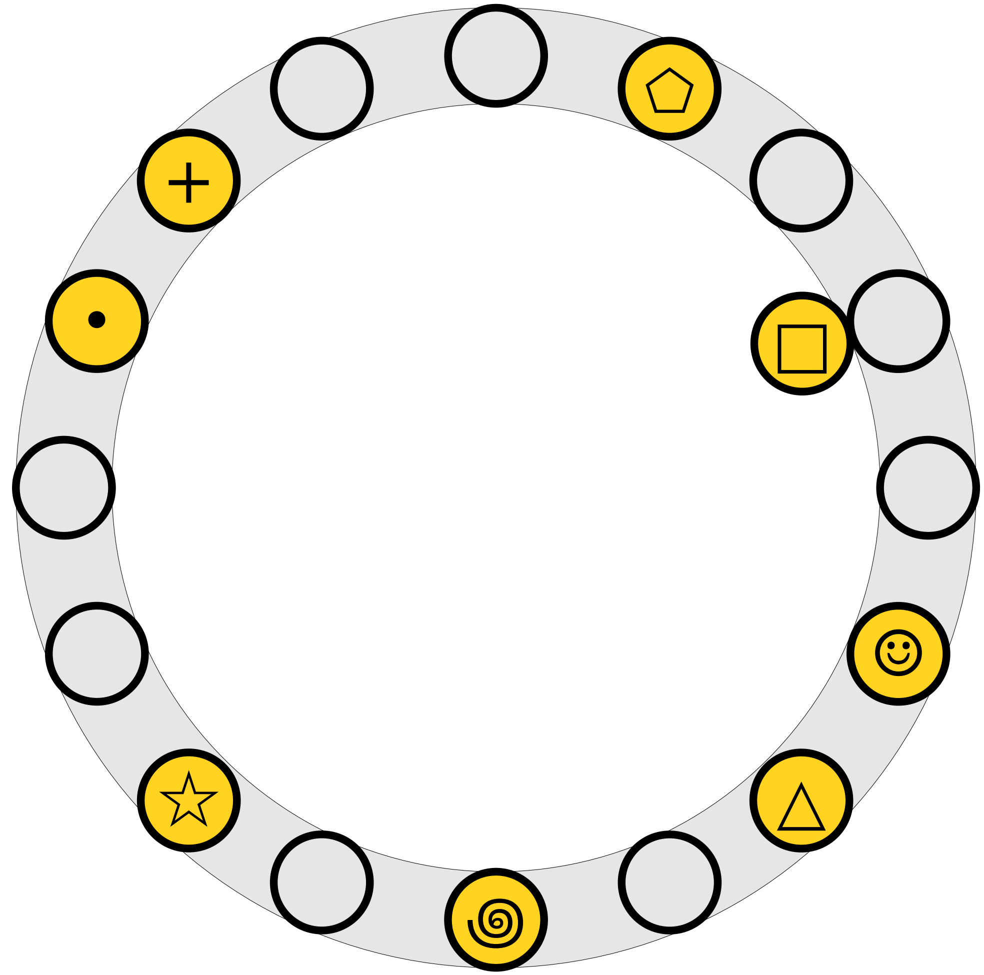
Cuckoo Hashing

- An insertion ***fails*** if the displacements form an infinite cycle.
- If that happens, perform a ***rehash*** by choosing a new h_1 and h_2 and inserting all elements back into the table.



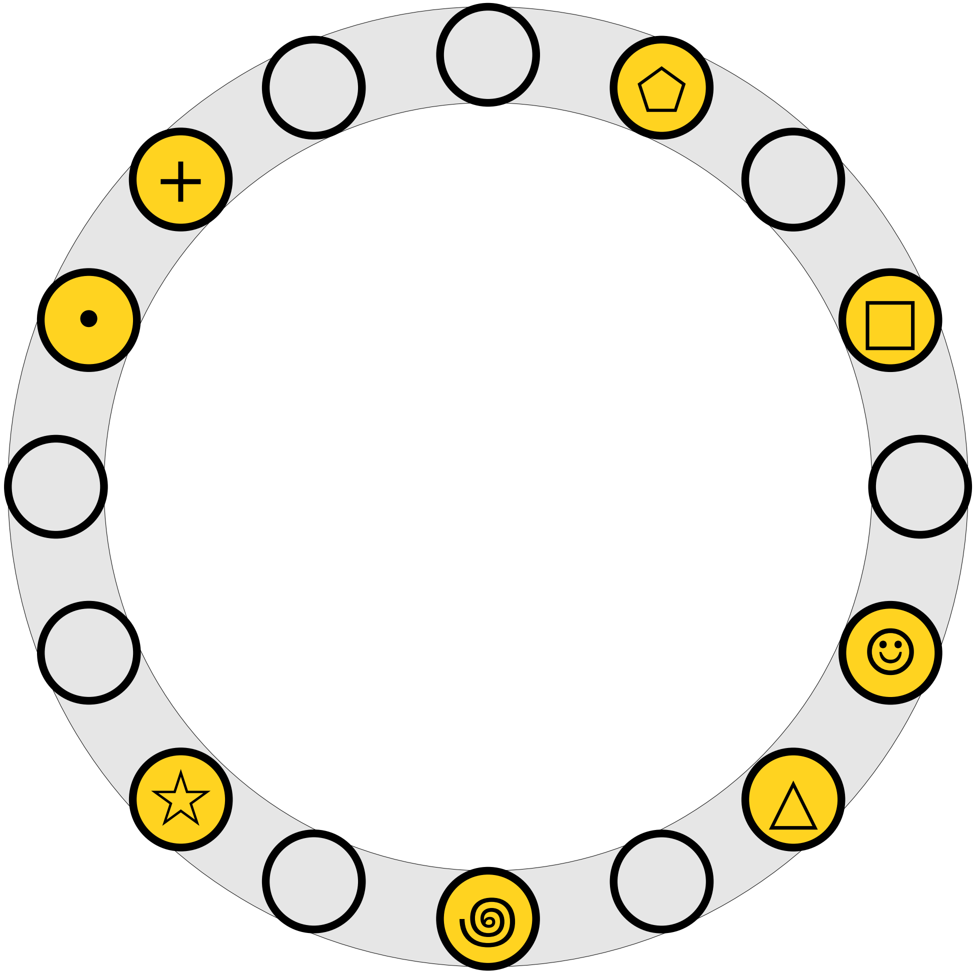
Cuckoo Hashing

- An insertion ***fails*** if the displacements form an infinite cycle.
- If that happens, perform a ***rehash*** by choosing a new h_1 and h_2 and inserting all elements back into the table.



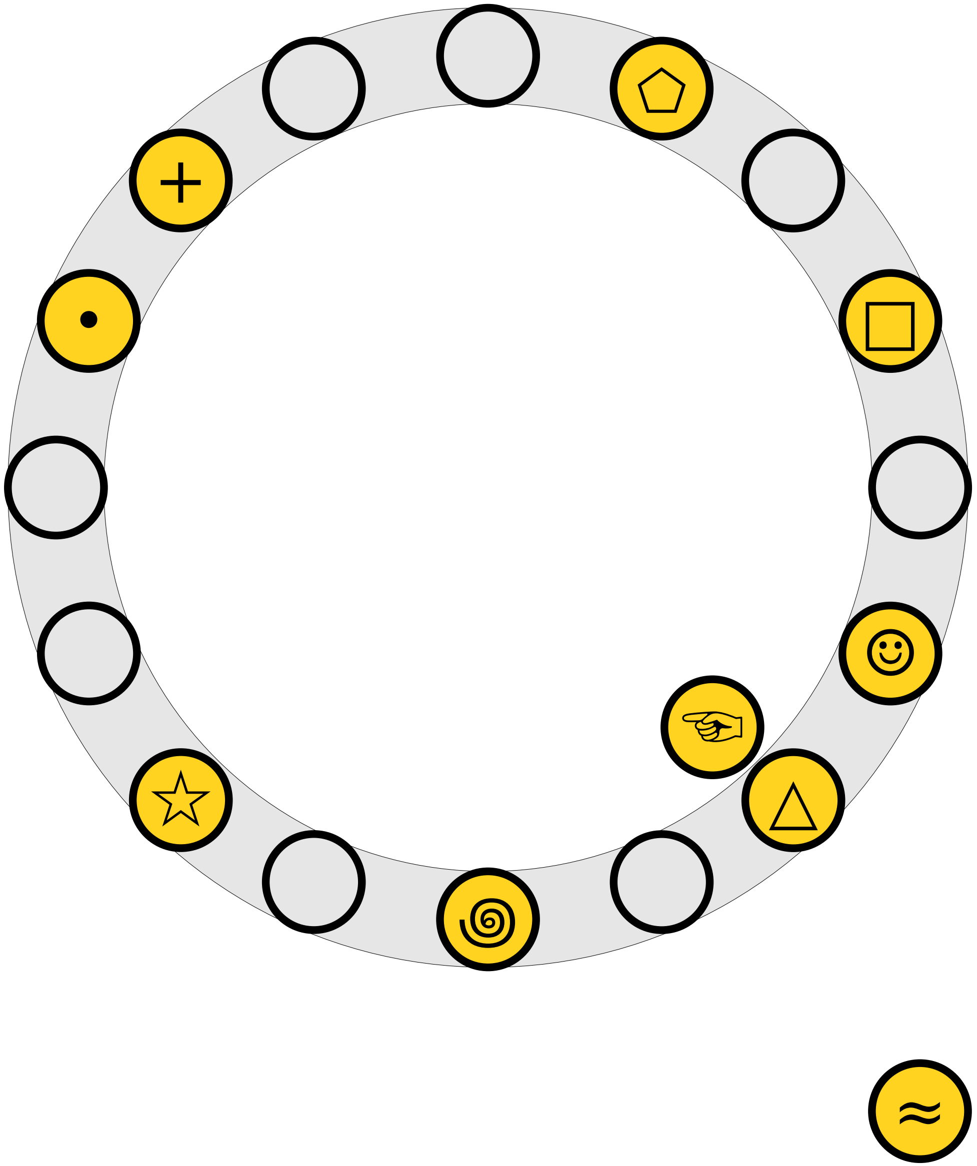
Cuckoo Hashing

- An insertion ***fails*** if the displacements form an infinite cycle.
- If that happens, perform a ***rehash*** by choosing a new h_1 and h_2 and inserting all elements back into the table.



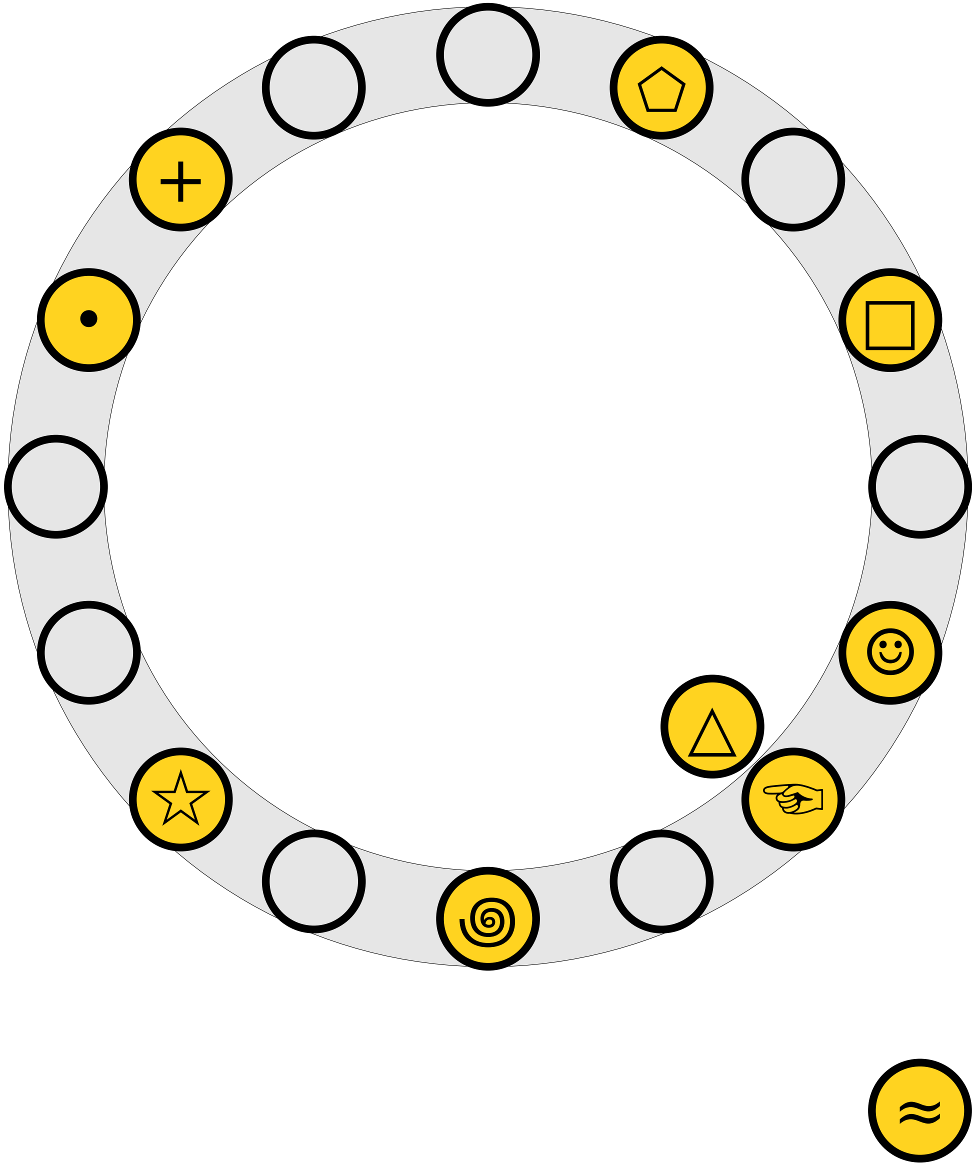
Cuckoo Hashing

- An insertion ***fails*** if the displacements form an infinite cycle.
- If that happens, perform a ***rehash*** by choosing a new h_1 and h_2 and inserting all elements back into the table.



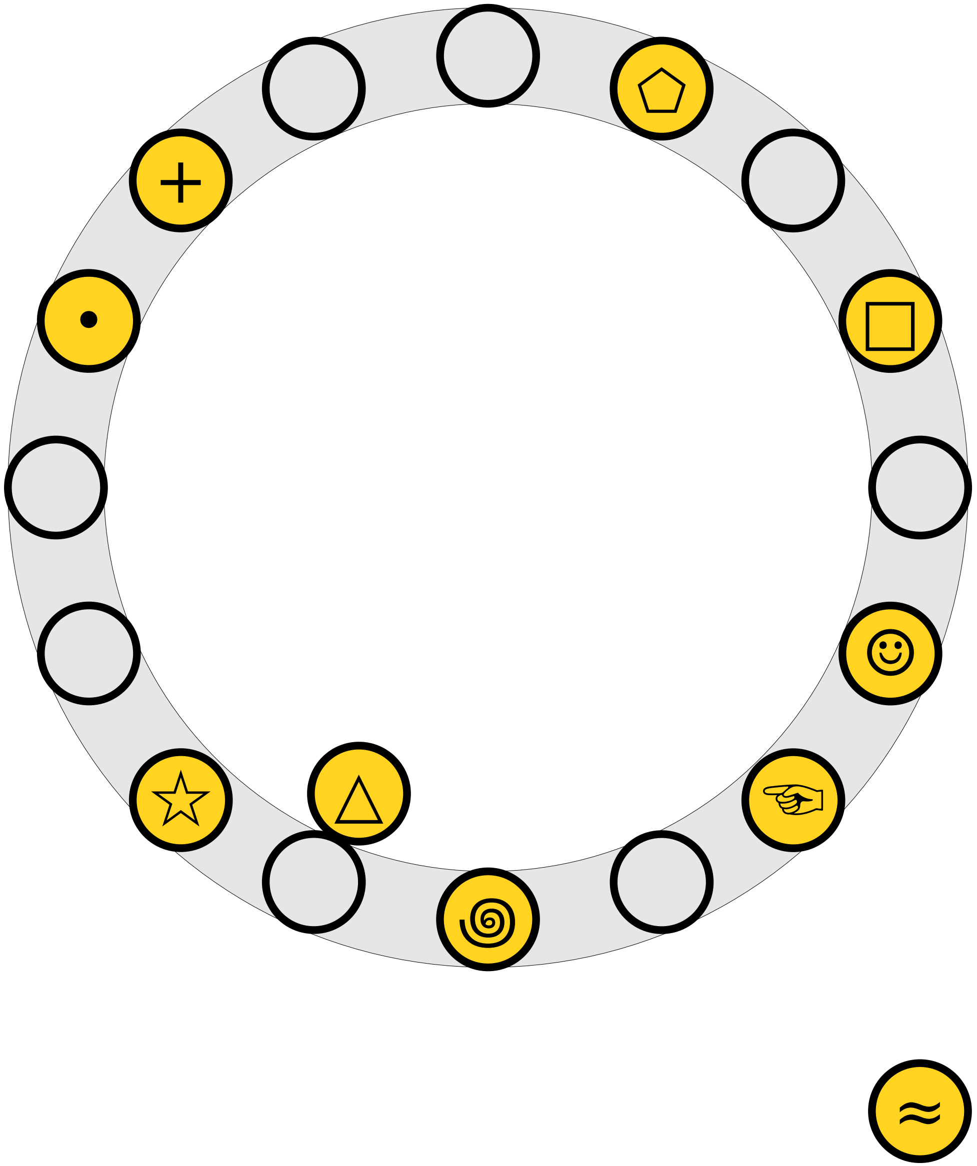
Cuckoo Hashing

- An insertion ***fails*** if the displacements form an infinite cycle.
- If that happens, perform a ***rehash*** by choosing a new h_1 and h_2 and inserting all elements back into the table.



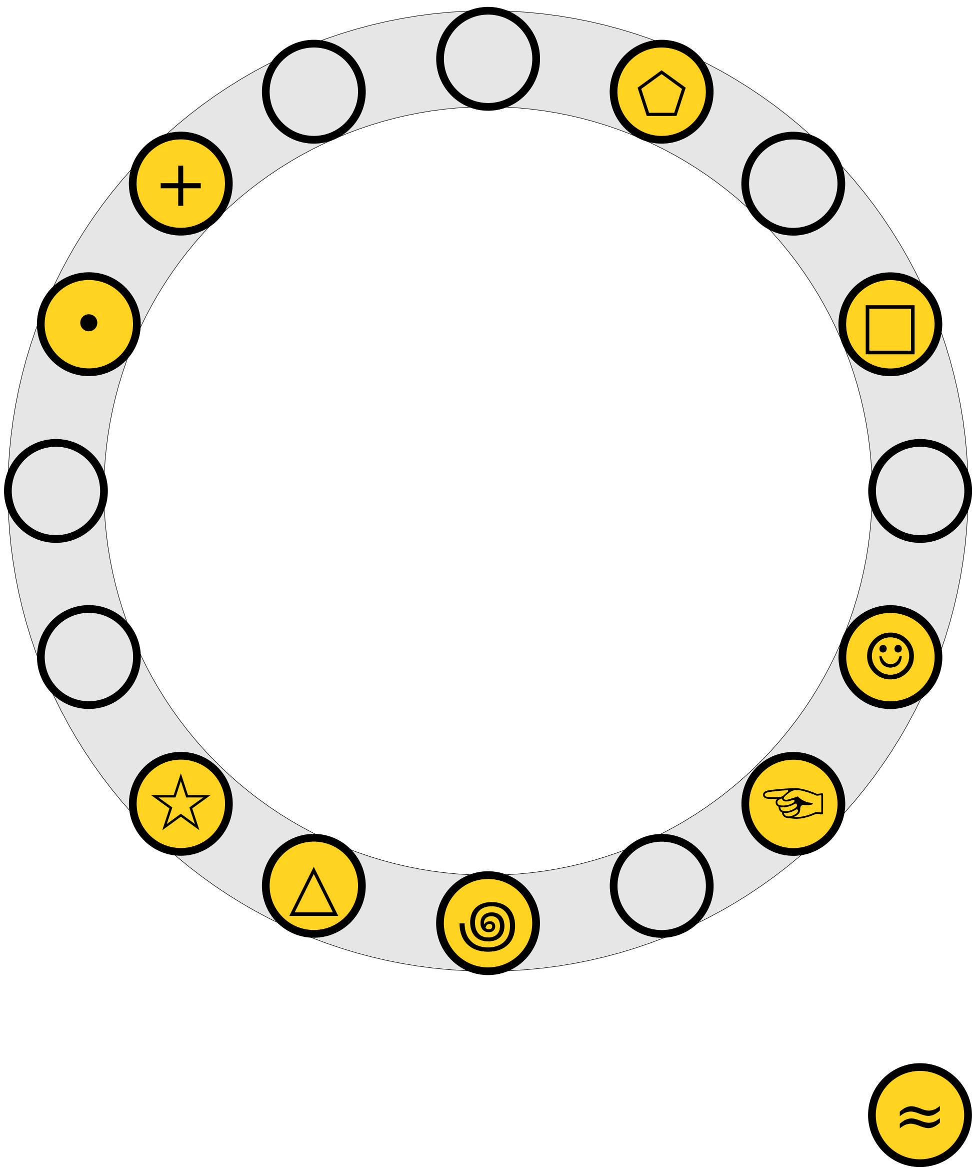
Cuckoo Hashing

- An insertion ***fails*** if the displacements form an infinite cycle.
- If that happens, perform a ***rehash*** by choosing a new h_1 and h_2 and inserting all elements back into the table.



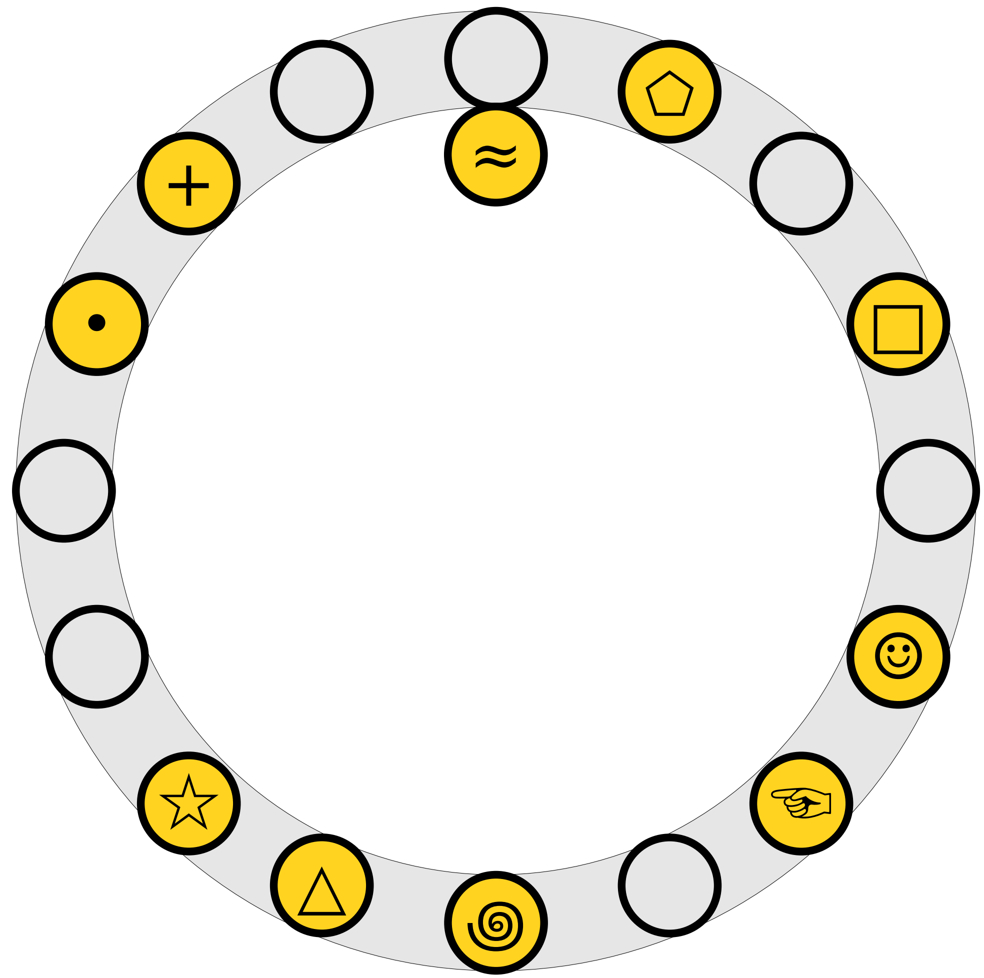
Cuckoo Hashing

- An insertion ***fails*** if the displacements form an infinite cycle.
- If that happens, perform a ***rehash*** by choosing a new h_1 and h_2 and inserting all elements back into the table.



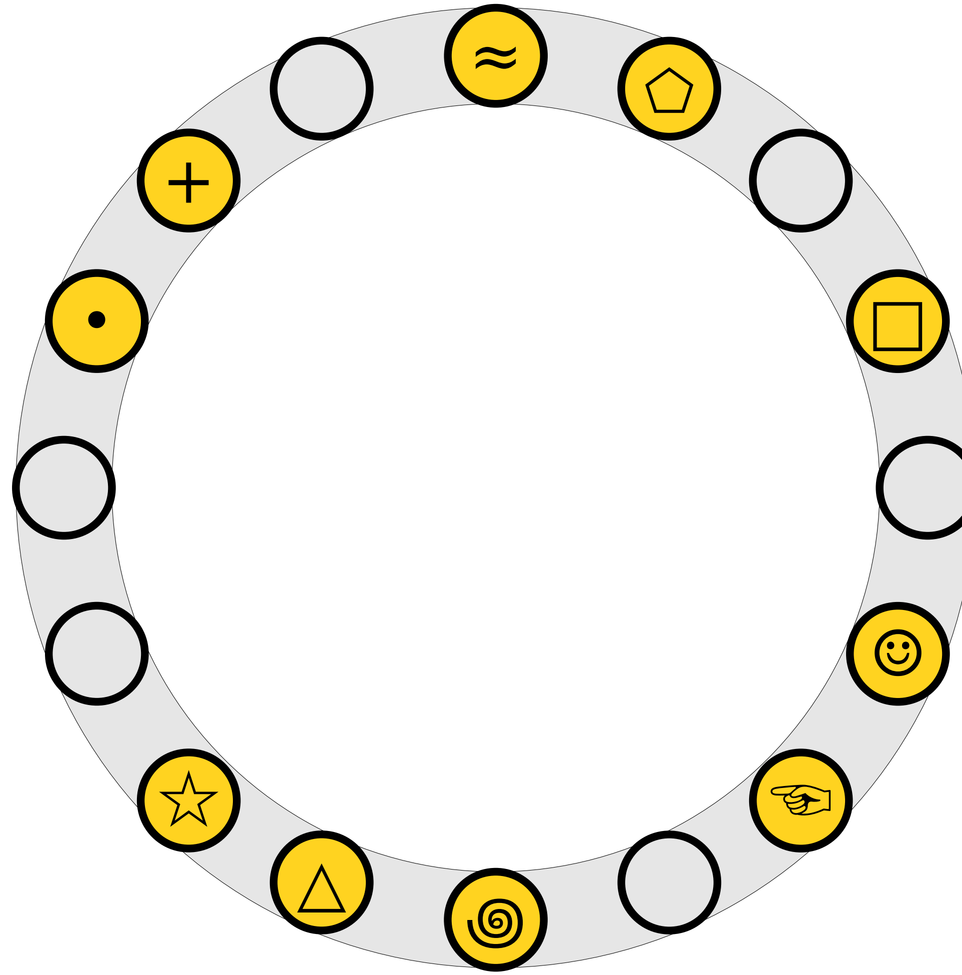
Cuckoo Hashing

- An insertion ***fails*** if the displacements form an infinite cycle.
- If that happens, perform a ***rehash*** by choosing a new h_1 and h_2 and inserting all elements back into the table.



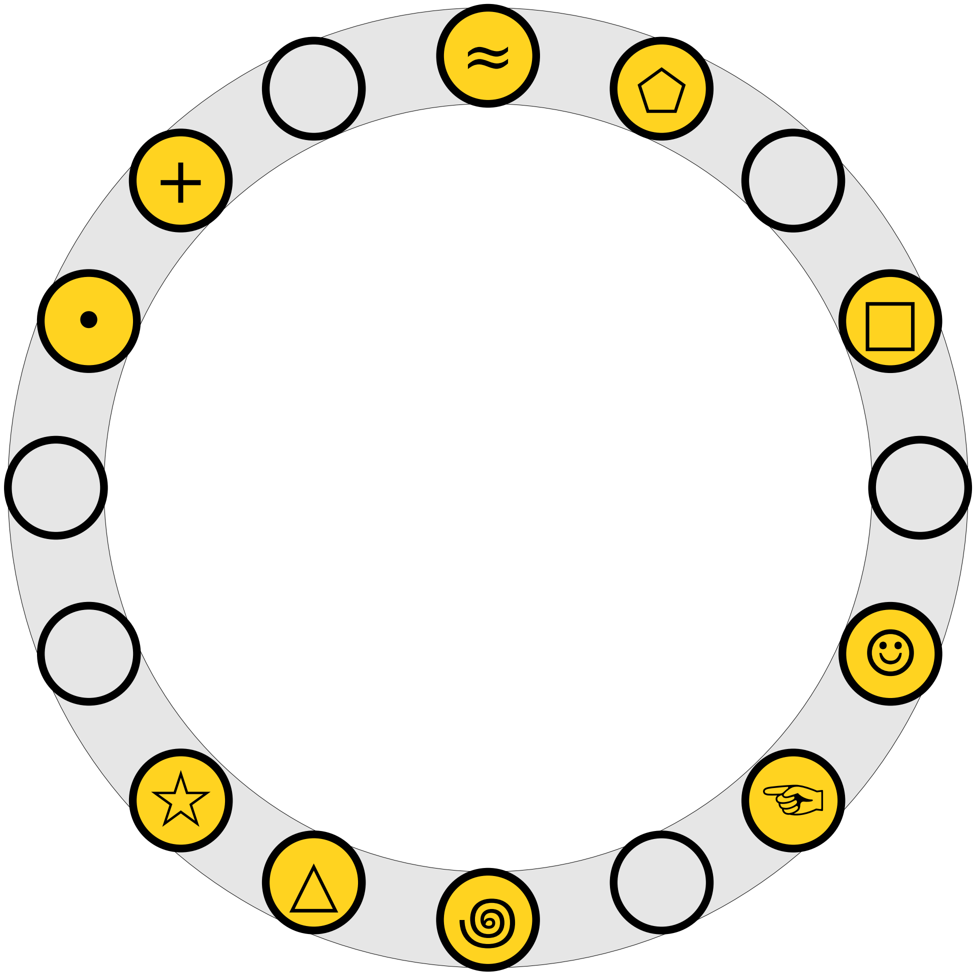
Cuckoo Hashing

- An insertion ***fails*** if the displacements form an infinite cycle.
- If that happens, perform a ***rehash*** by choosing a new h_1 and h_2 and inserting all elements back into the table.



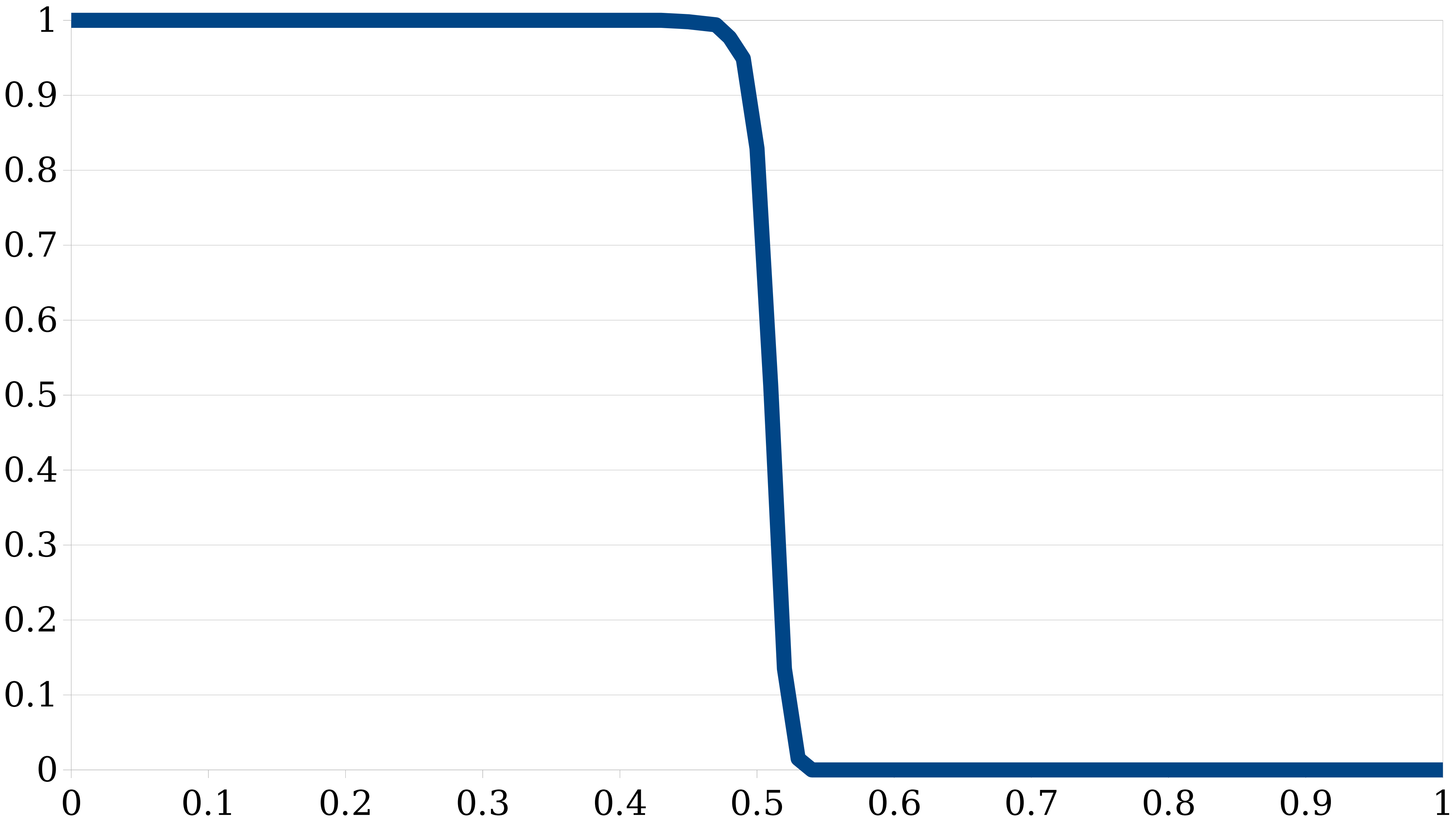
Cuckoo Hashing

- An insertion ***fails*** if the displacements form an infinite cycle.
- If that happens, perform a ***rehash*** by choosing a new h_1 and h_2 and inserting all elements back into the table.
- Multiple rehashes might be necessary before this succeeds – do you see why?

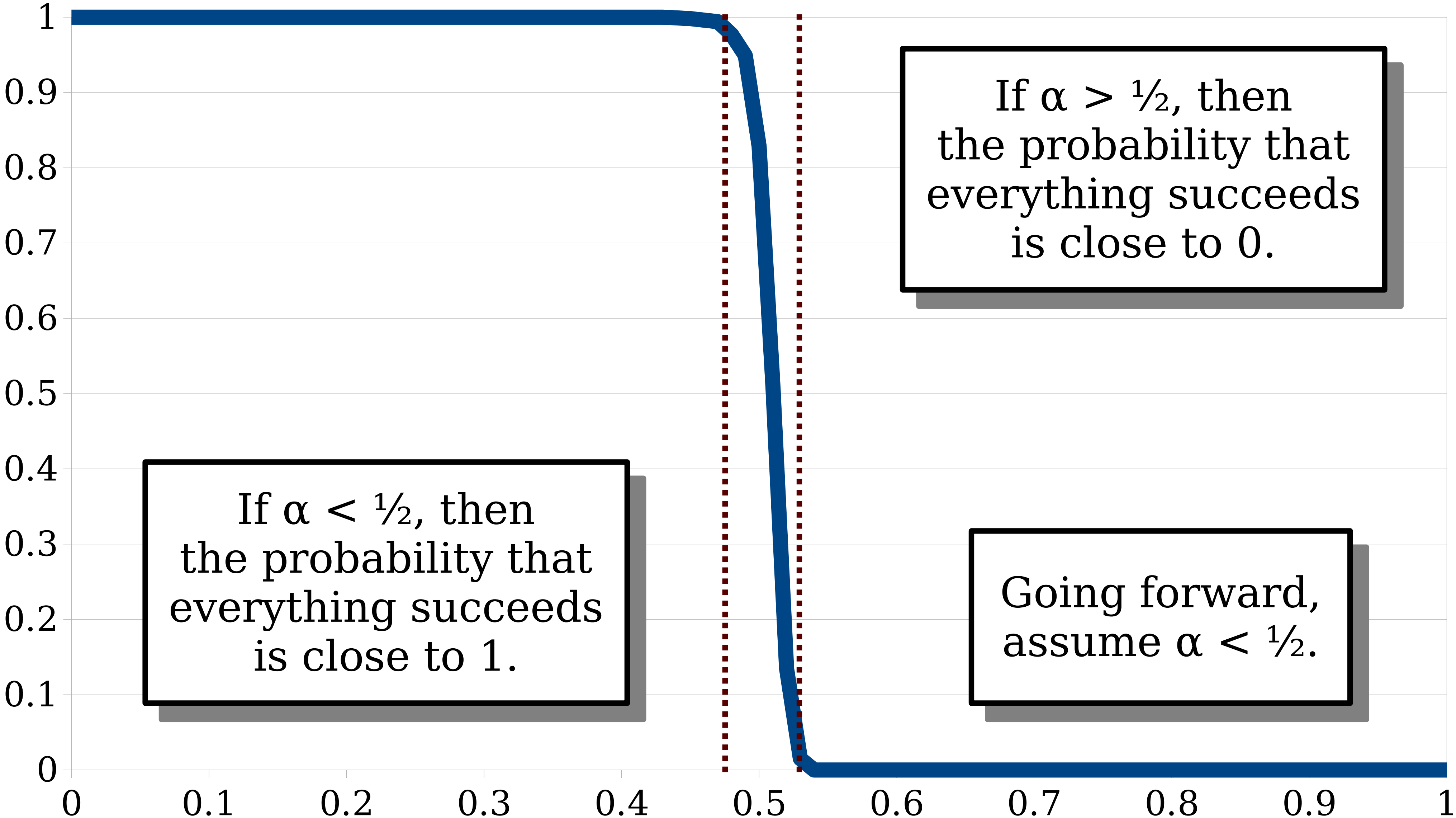


How efficient is cuckoo hashing?

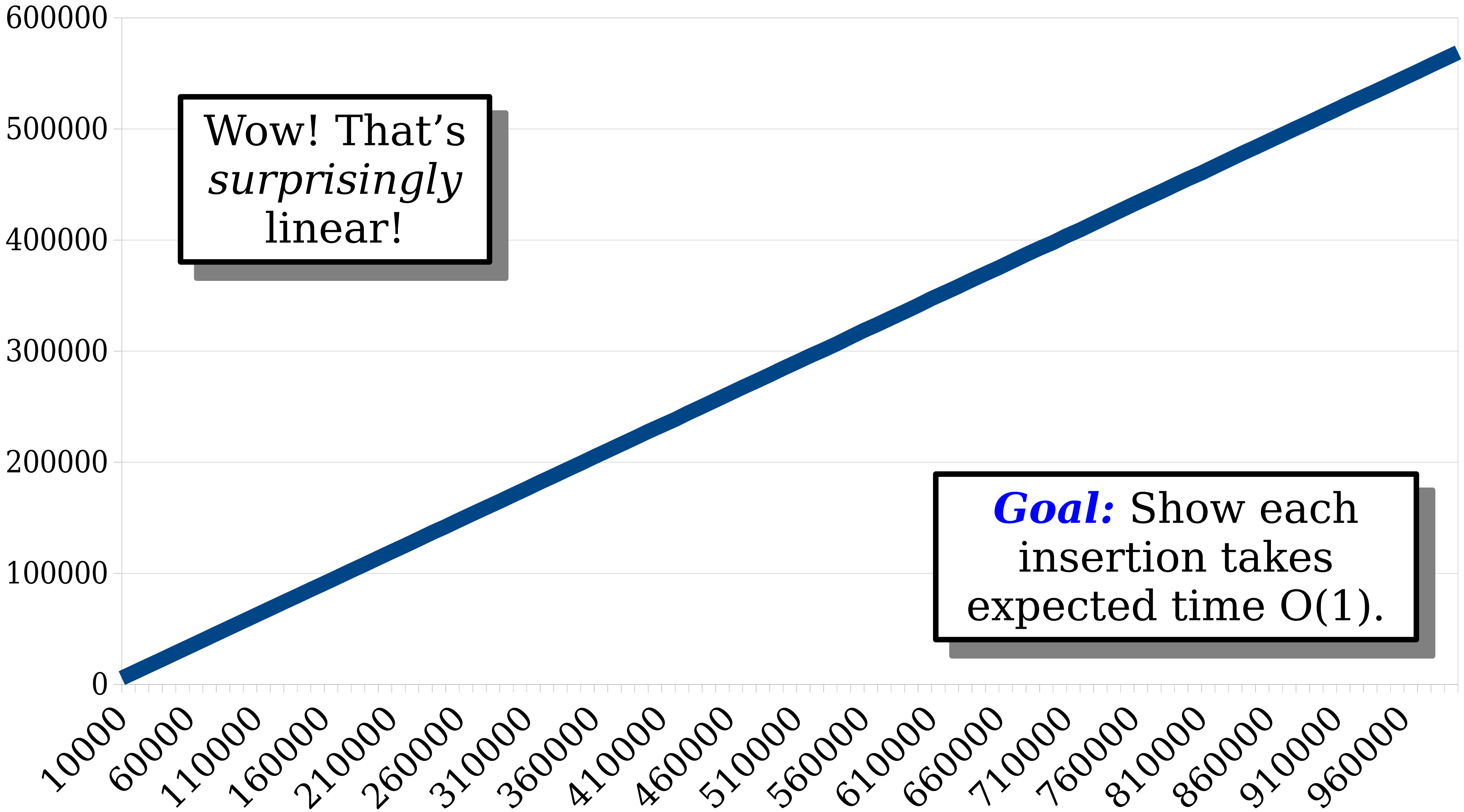
Pro tip: When analyzing a data structure, it never hurts to get some empirical performance data first.



Suppose we have m slots and store n total elements.
What is the probability that all the insertions succeed,
as a function of the ***load factor*** $\alpha = {}^{n}/_{m}$?



Suppose we have m slots and store n total elements.
What is the probability that all the insertions succeed,
as a function of the ***load factor*** $\alpha = {}^{n}/_{m}$?



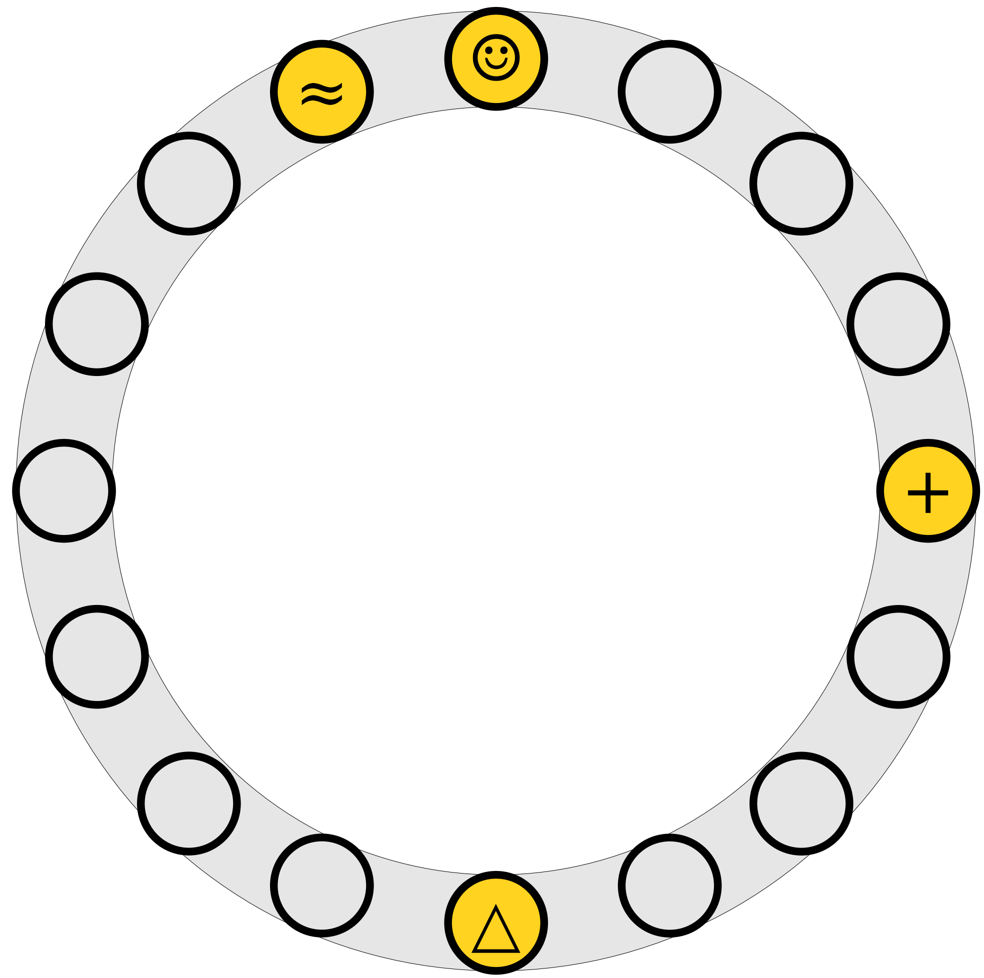
Suppose we store n total elements in a table with m slots, where $n < \frac{1}{2}m$.

How many total displacements occur across all insertions?

Goal: Show that insertions take expected time O(1), under the assumption that $n = \alpha m$ for some $\alpha < \frac{1}{2}$.

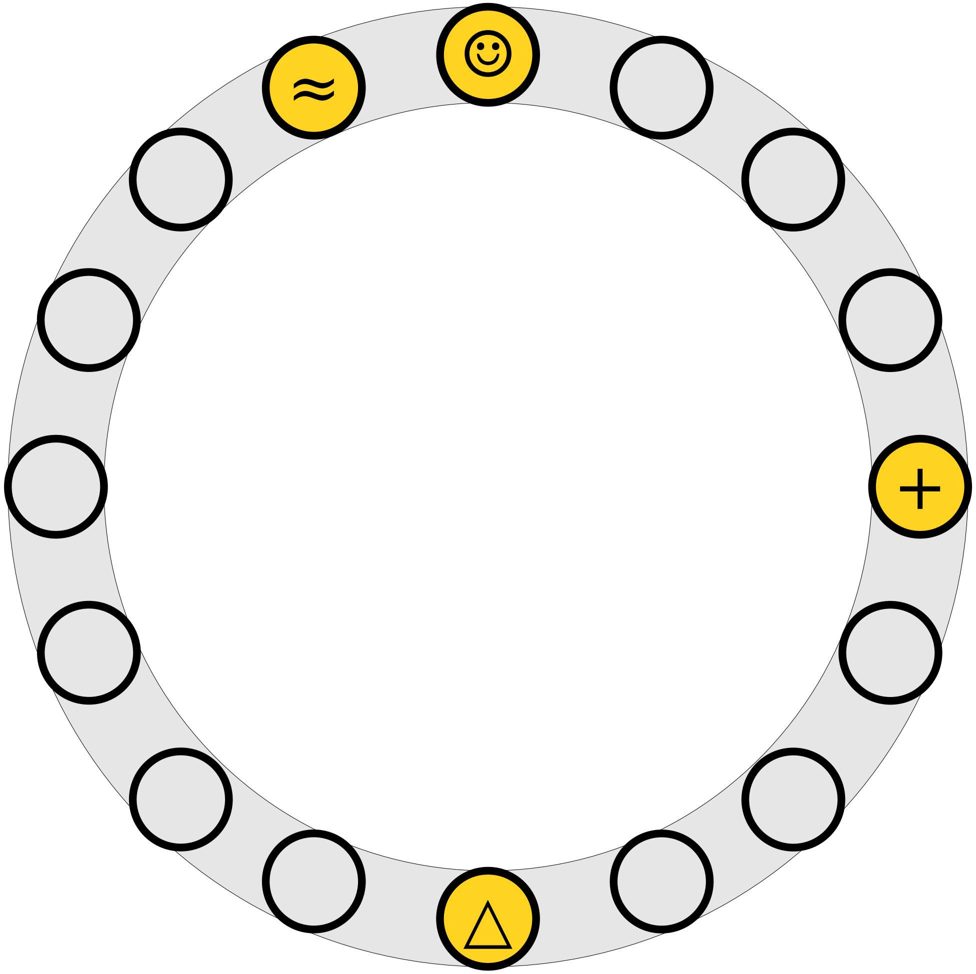
Analyzing Cuckoo Hashing

- The analysis of cuckoo hashing is more difficult than it might at first seem.
- ***Challenge 1:*** We may have to consider hash collisions across multiple hash functions.
- ***Challenge 2:*** We need to reason about chains of displacement, not just how many elements land somewhere.
- To resolve these challenges, we'll need to bring in some new techniques.



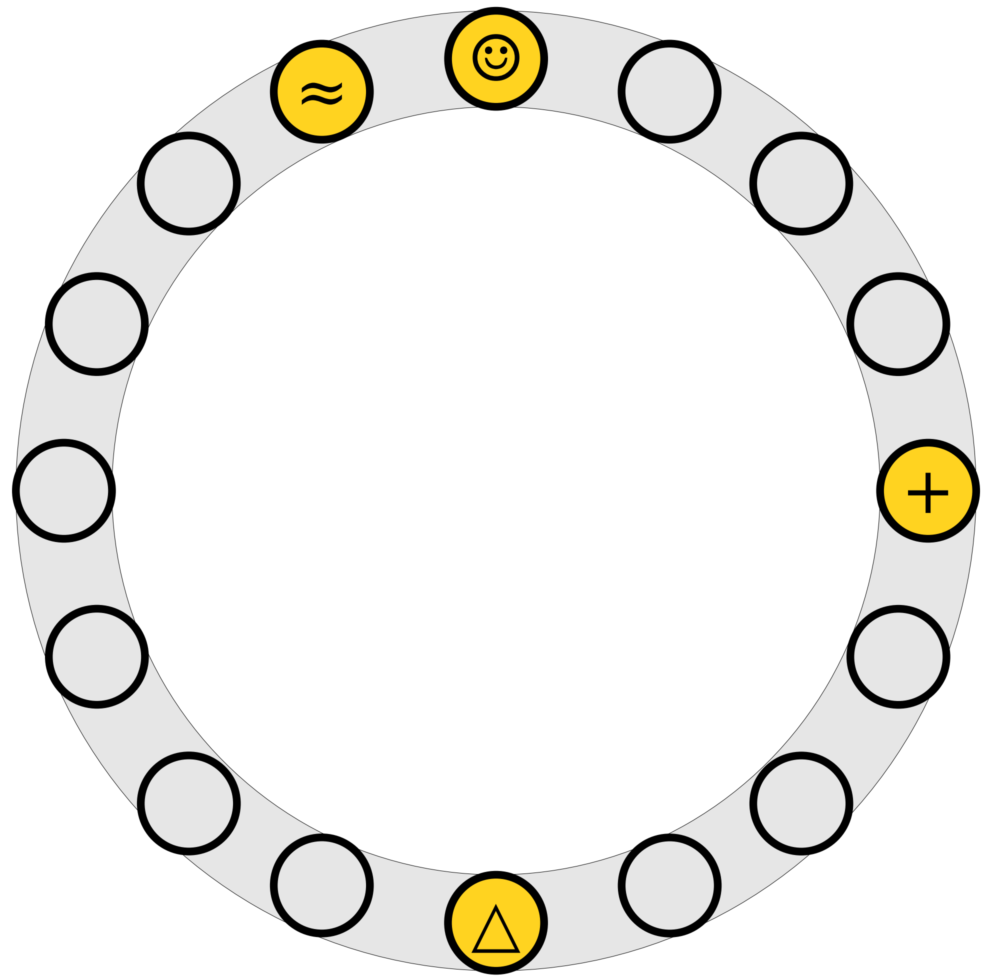
The Cuckoo Graph

- The ***cuckoo graph*** is a (multi)graph derived from a cuckoo hash table.



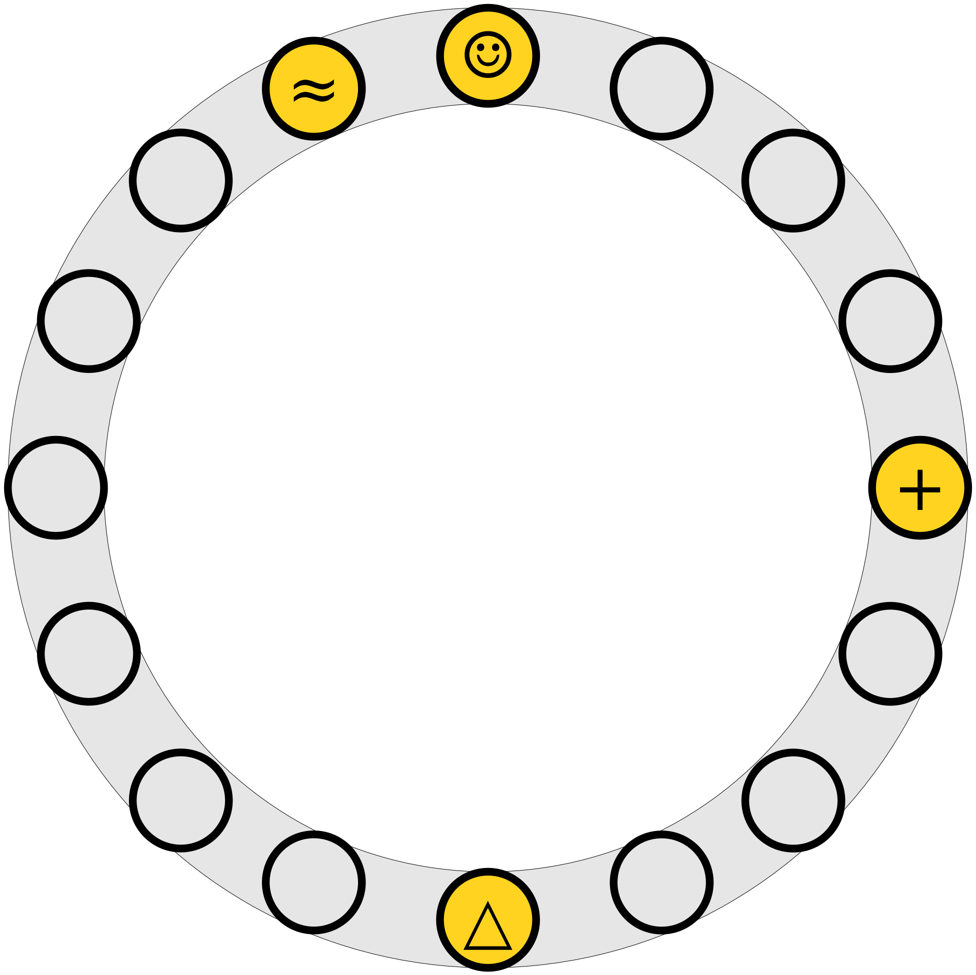
The Cuckoo Graph

- The ***cuckoo graph*** is a (multi)graph derived from a cuckoo hash table.
- Each table slot is a node.



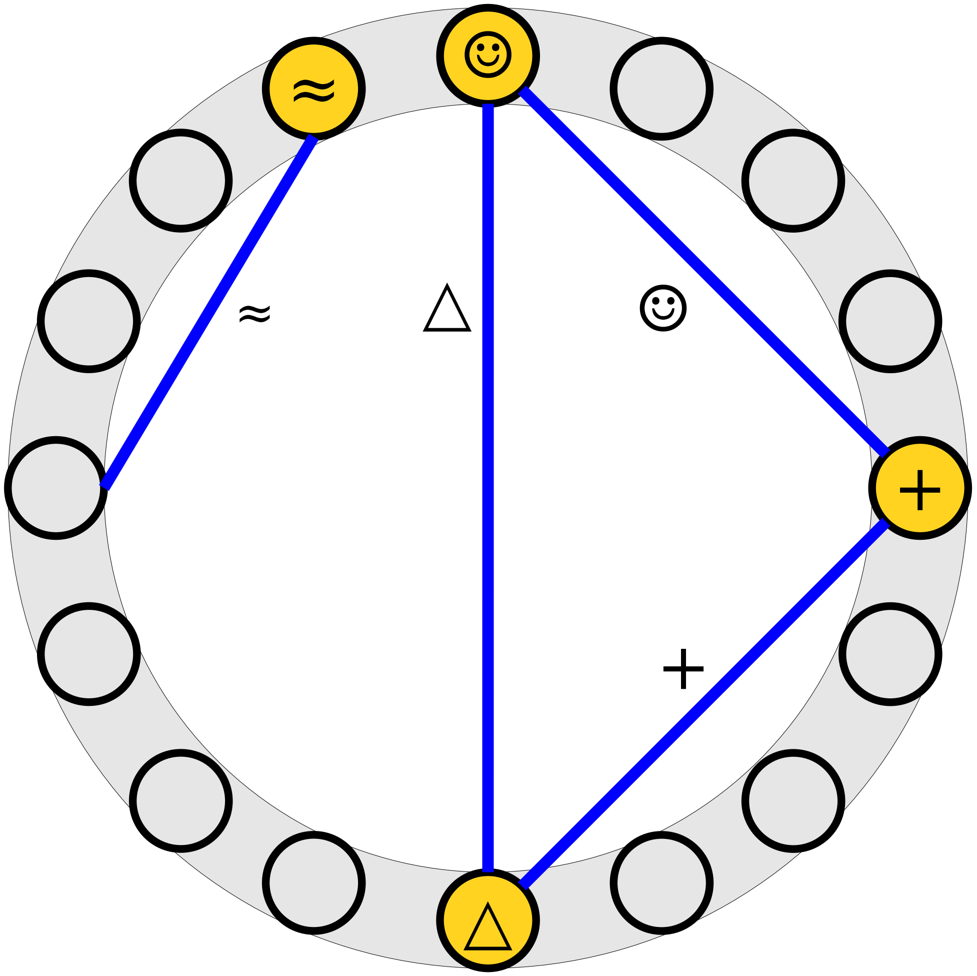
The Cuckoo Graph

- The ***cuckoo graph*** is a (multi)graph derived from a cuckoo hash table.
- Each table slot is a node.
- Each element is an edge linking the slots where it can be placed.



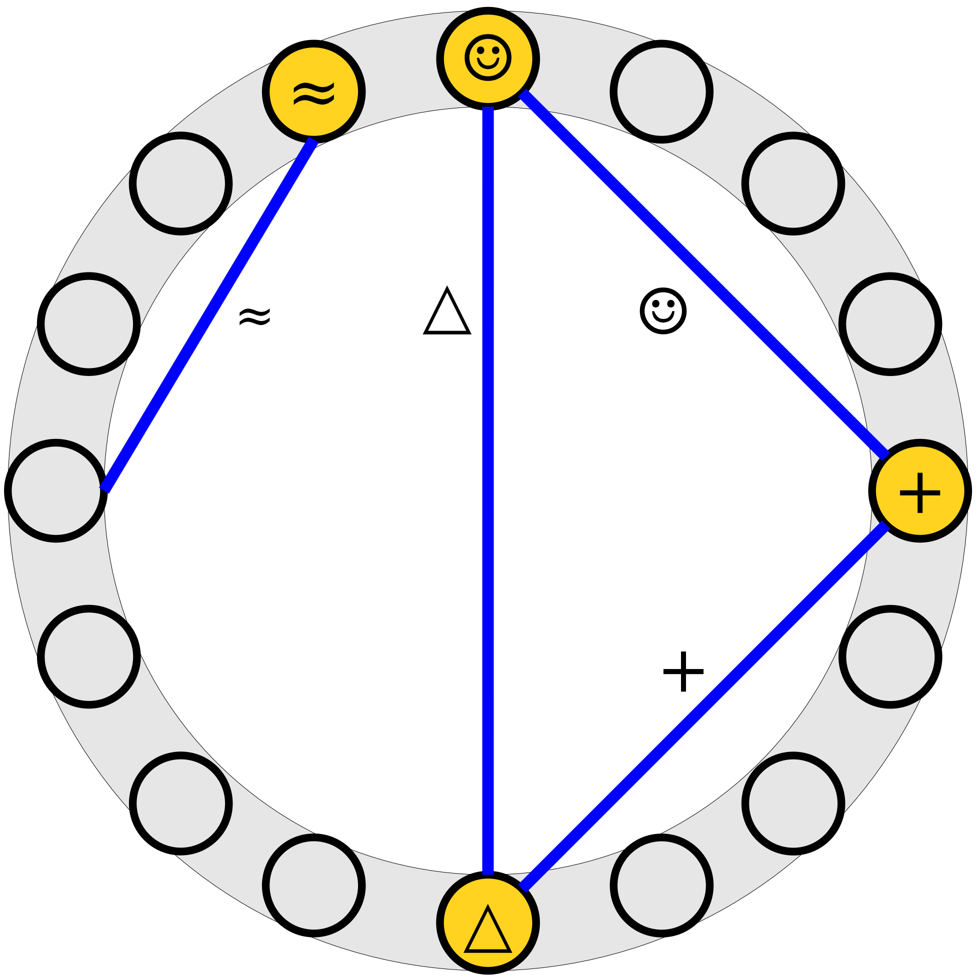
The Cuckoo Graph

- The ***cuckoo graph*** is a (multi)graph derived from a cuckoo hash table.
- Each table slot is a node.
- Each element is an edge linking the slots where it can be placed.



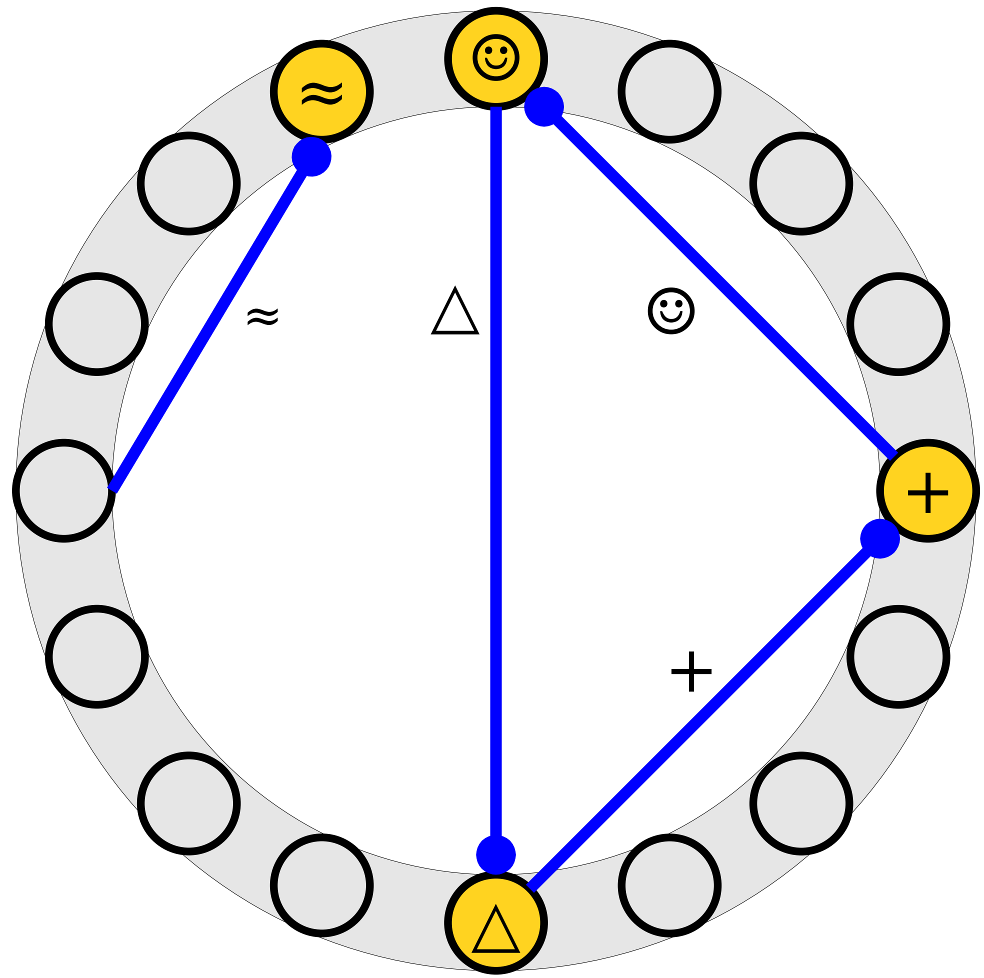
The Cuckoo Graph

- The ***cuckoo graph*** is a (multi)graph derived from a cuckoo hash table.
- Each table slot is a node.
- Each element is an edge linking the slots where it can be placed.
- An item's position in the table is denoted with a dot at the end of the line.



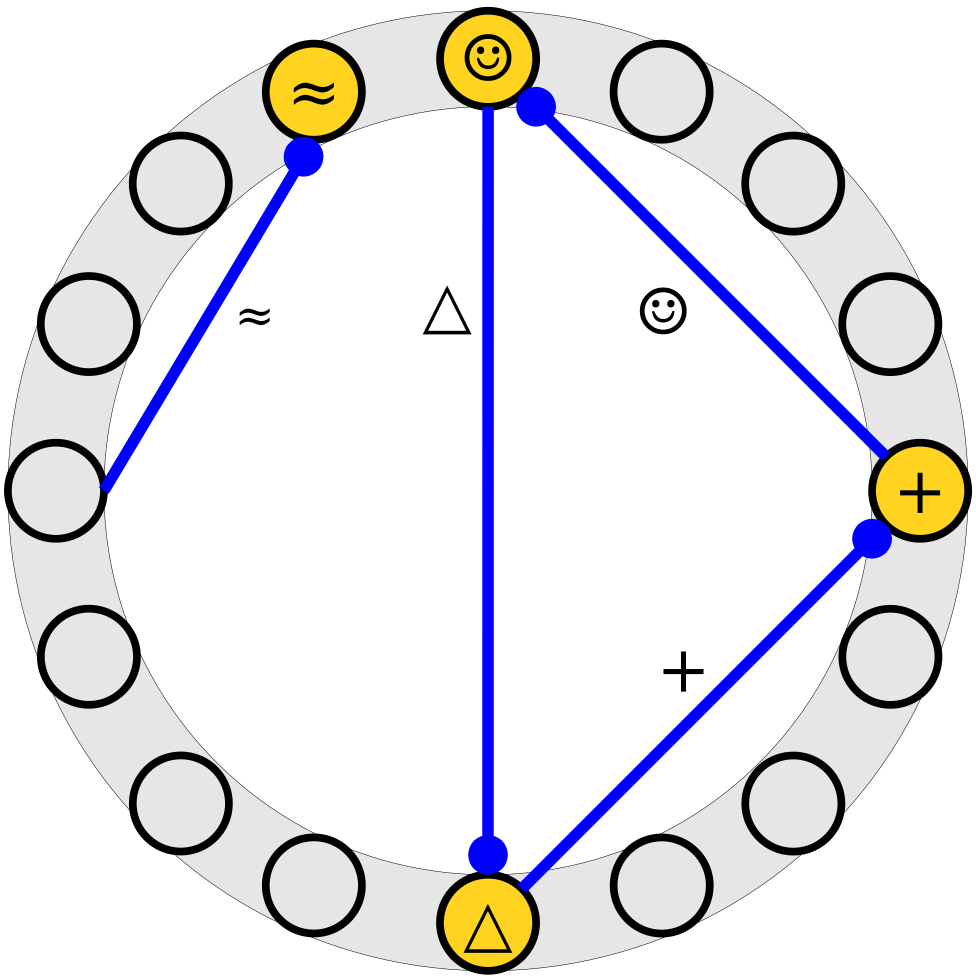
The Cuckoo Graph

- The ***cuckoo graph*** is a (multi)graph derived from a cuckoo hash table.
- Each table slot is a node.
- Each element is an edge linking the slots where it can be placed.
- An item's position in the table is denoted with a dot at the end of the line.



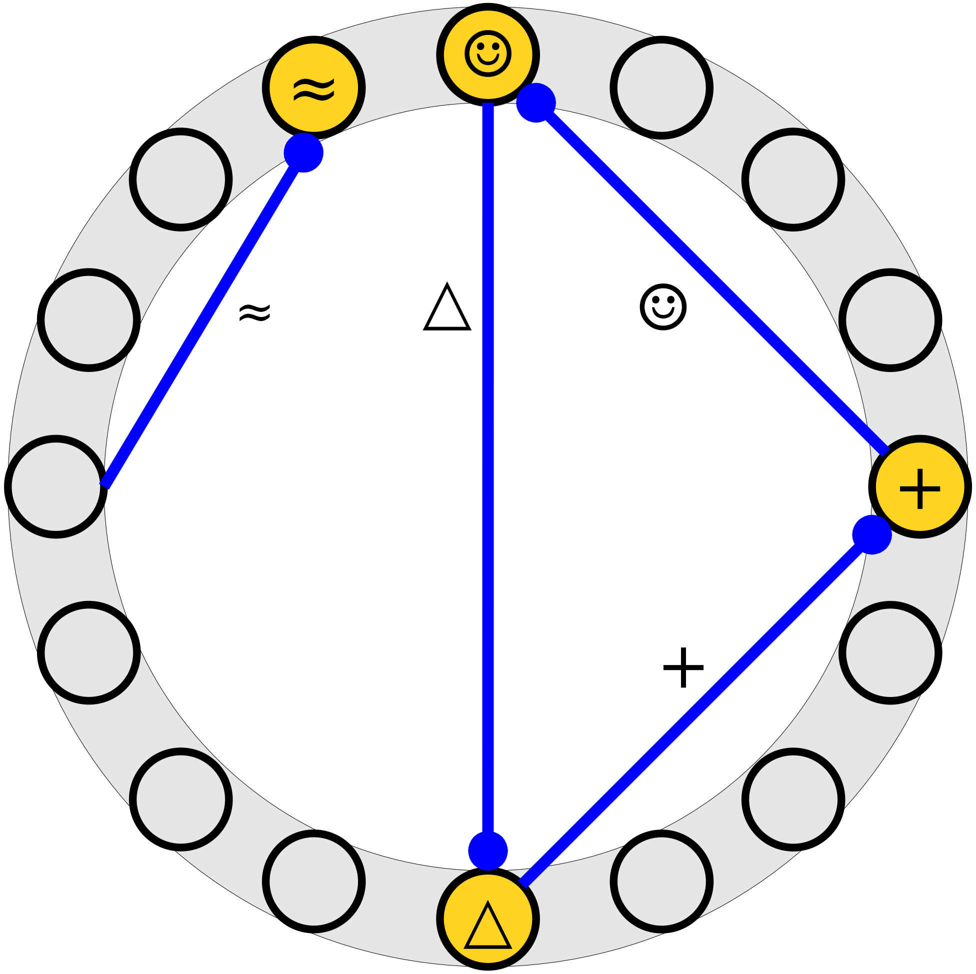
The Cuckoo Graph

- The ***cuckoo graph*** is a (multi)graph derived from a cuckoo hash table.
- Each table slot is a node.
- Each element is an edge linking the slots where it can be placed.
- An item's position in the table is denoted with a dot at the end of the line.
- Each node has at most one dot touching it.



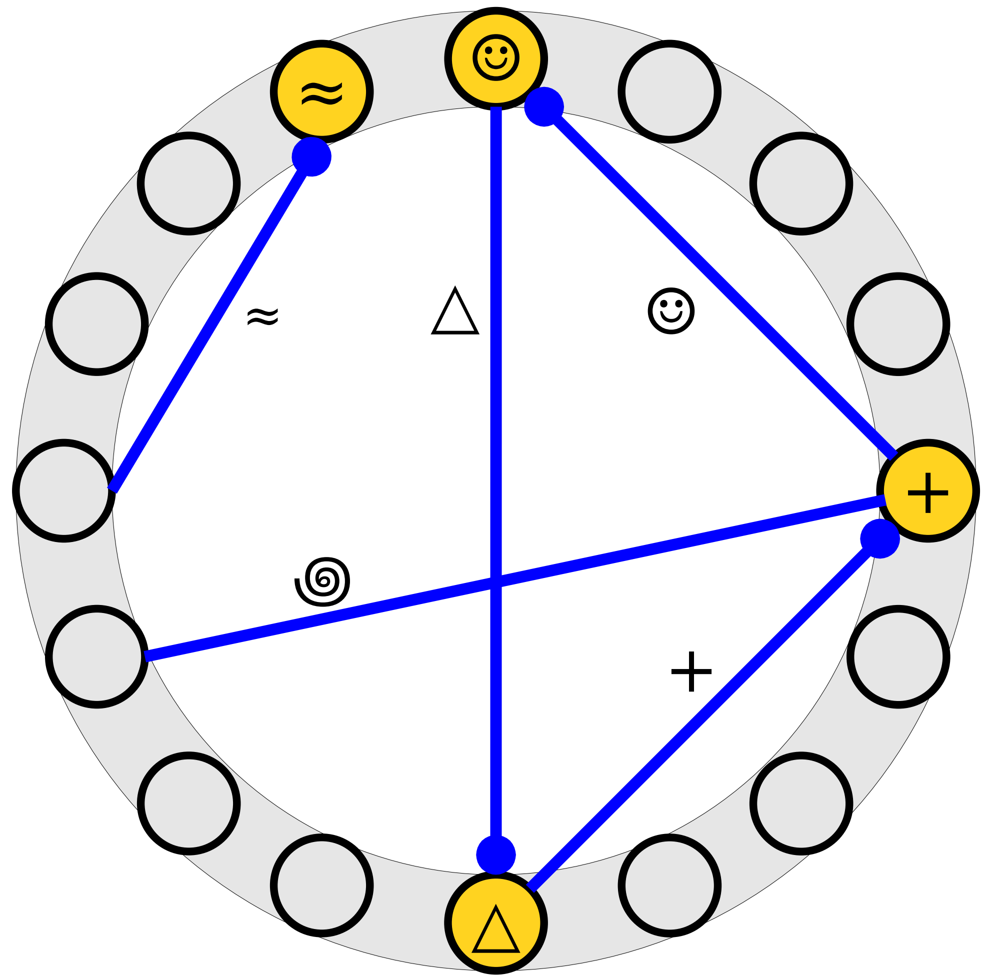
The Cuckoo Graph

- Inserting an element into a cuckoo hash table adds a new edge to the graph linking two nodes (slots).



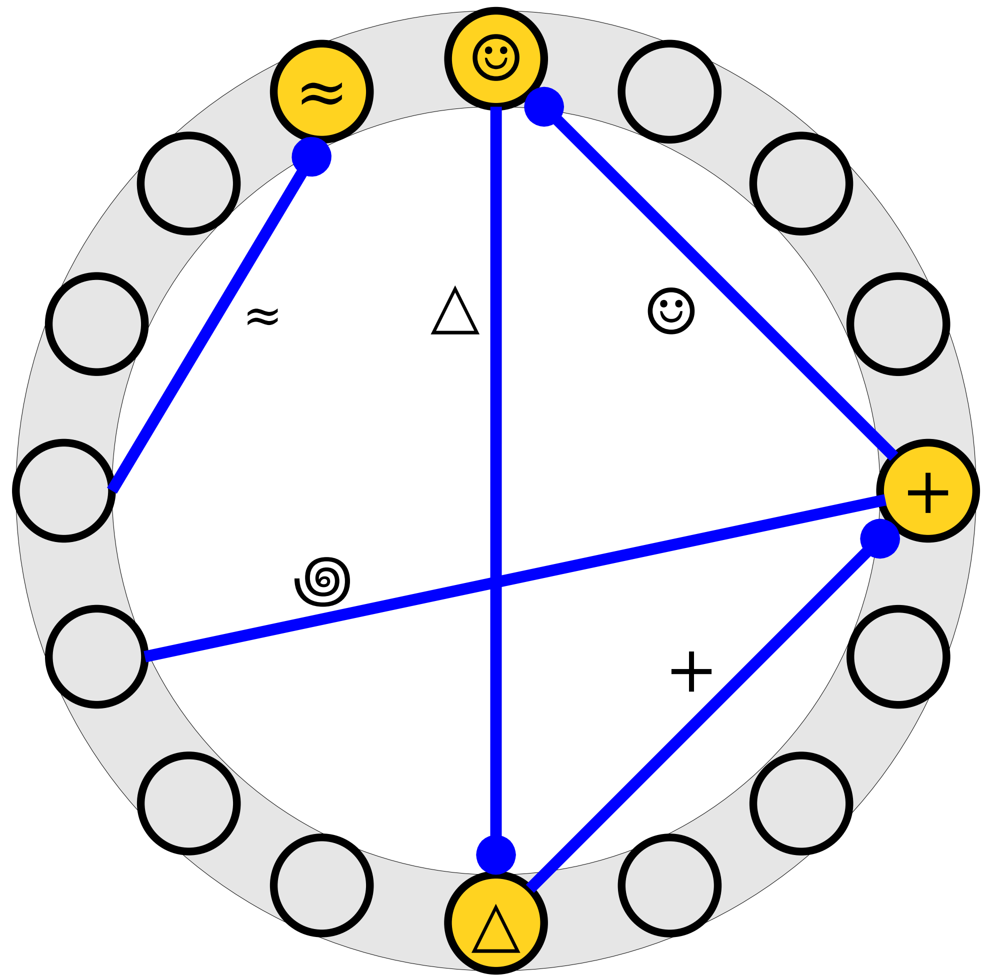
The Cuckoo Graph

- Inserting an element into a cuckoo hash table adds a new edge to the graph linking two nodes (slots).



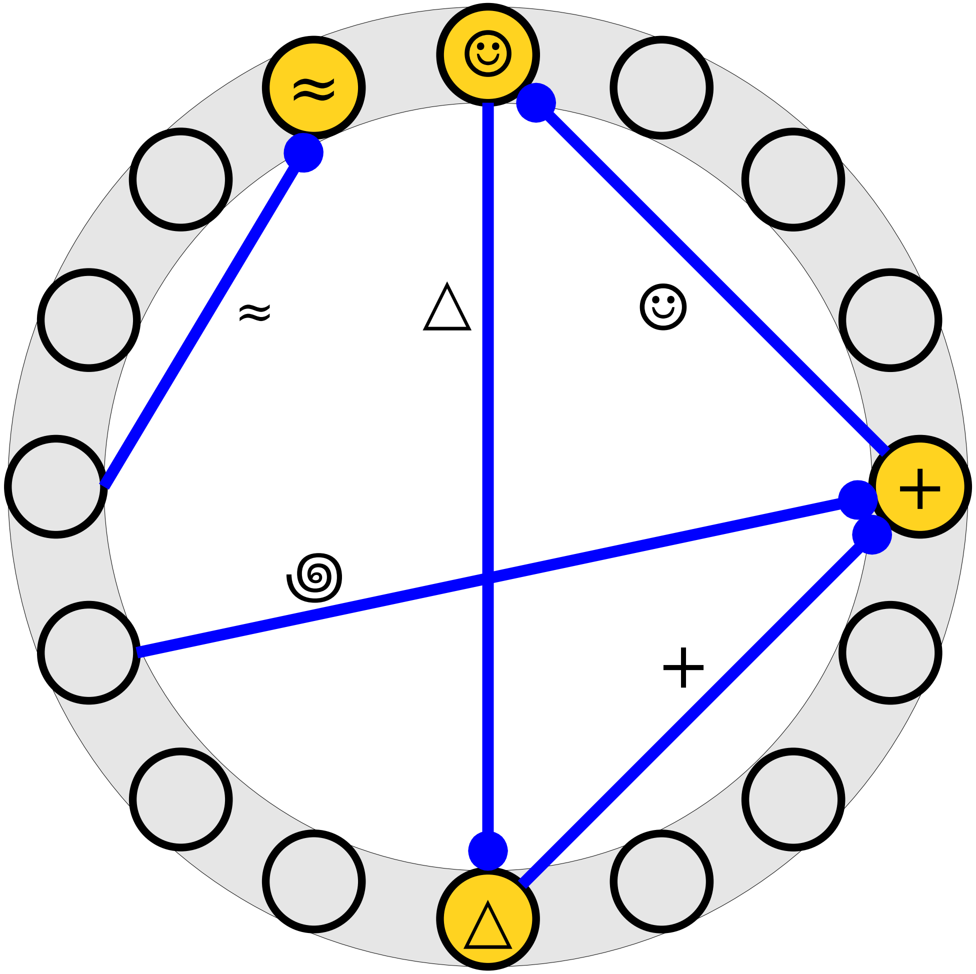
The Cuckoo Graph

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



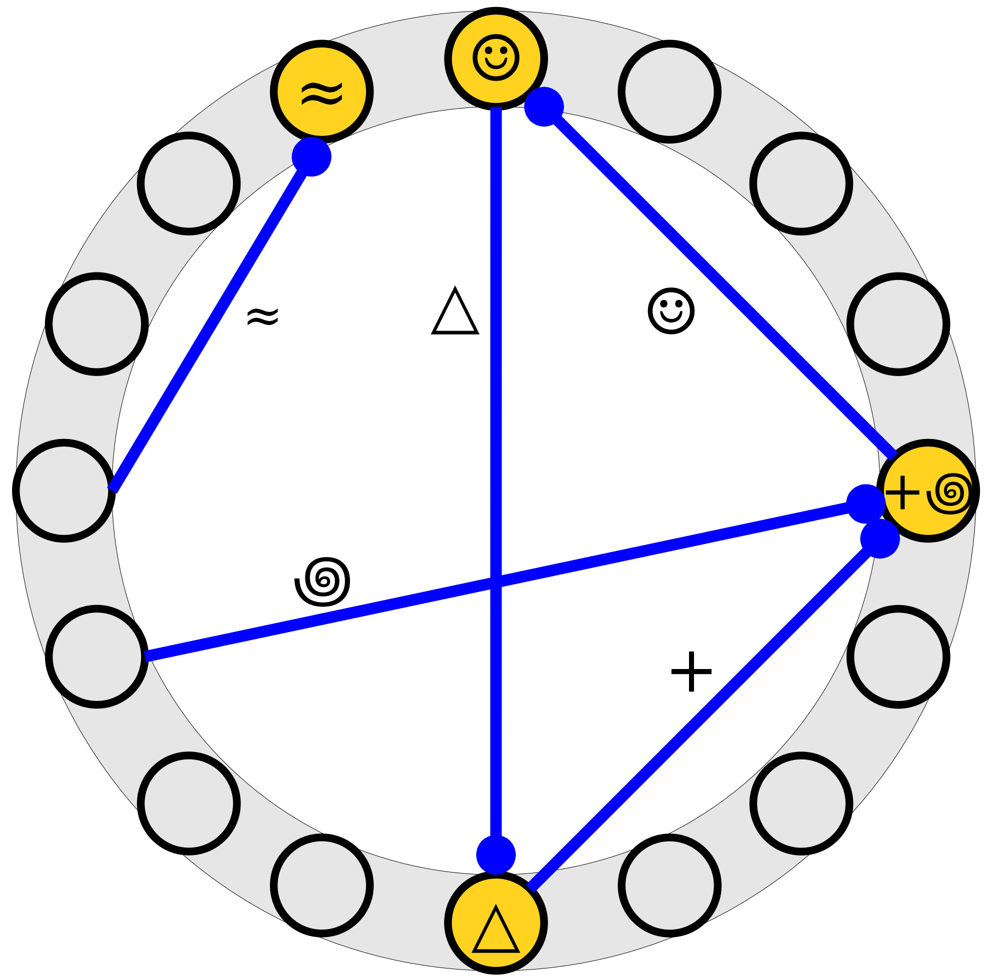
The Cuckoo Graph

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



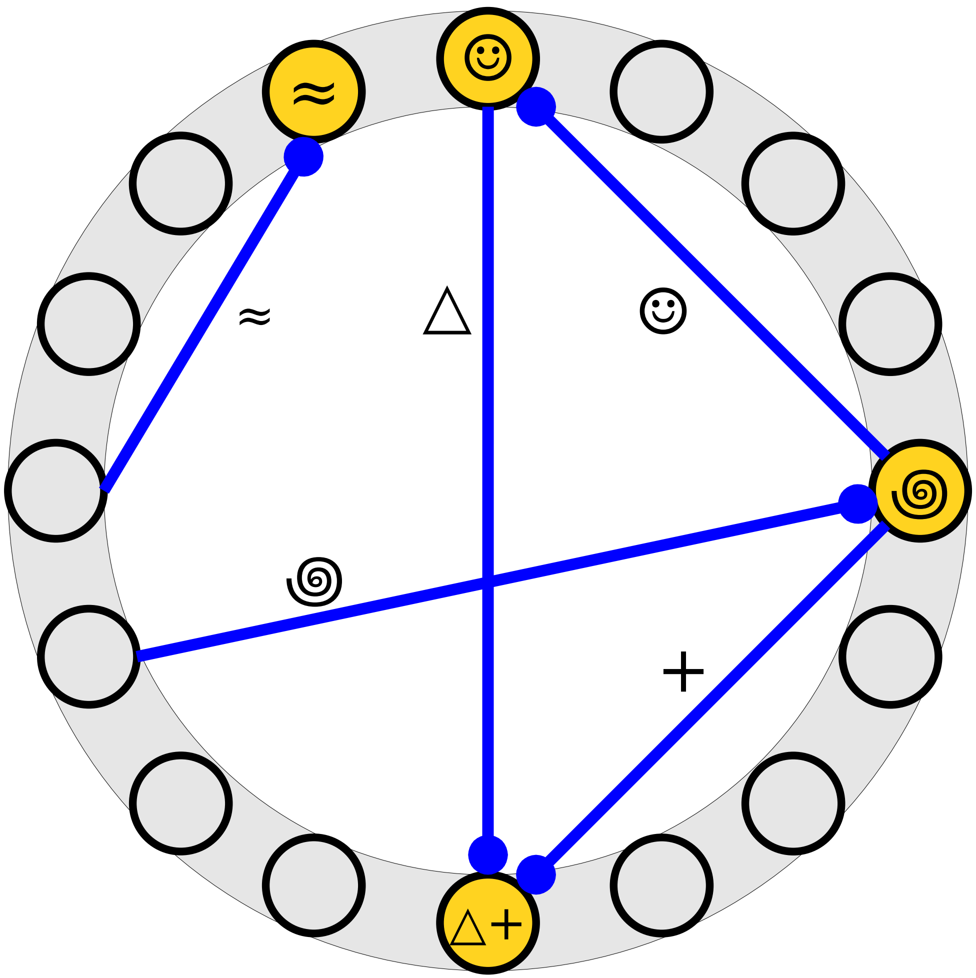
The Cuckoo Graph

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



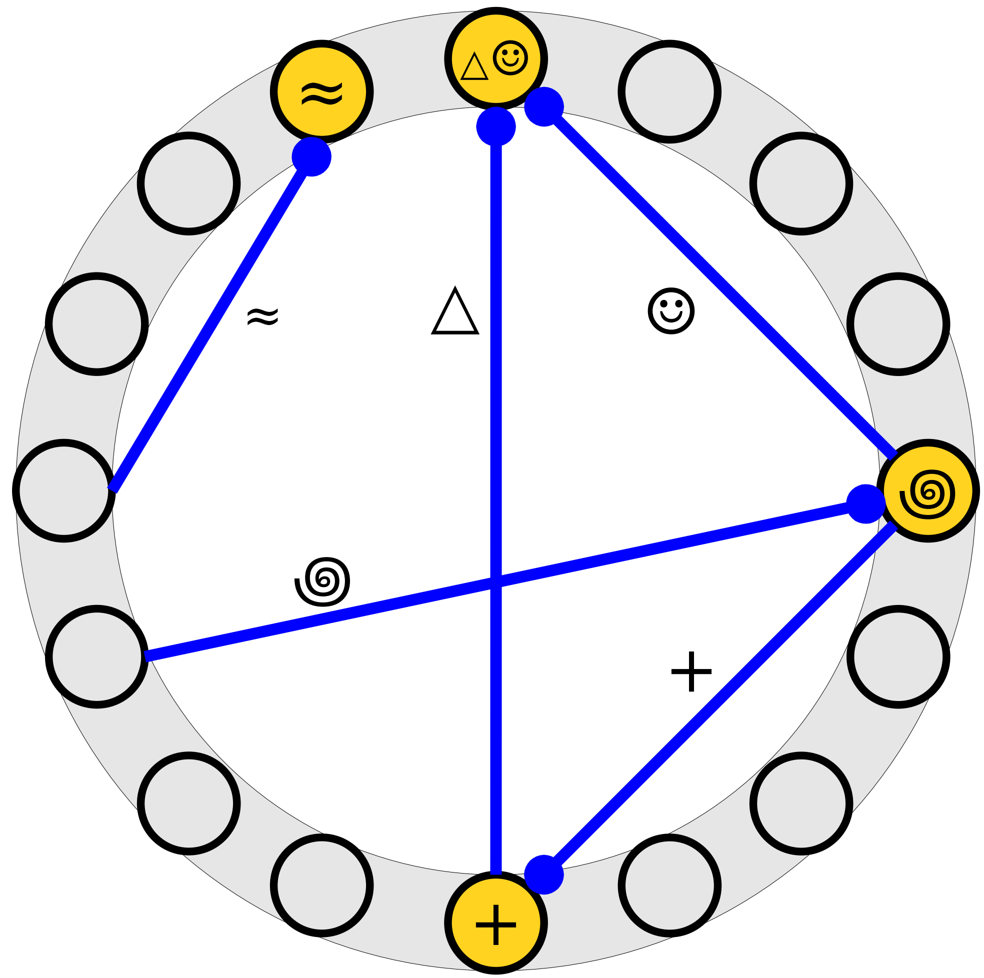
The Cuckoo Graph

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



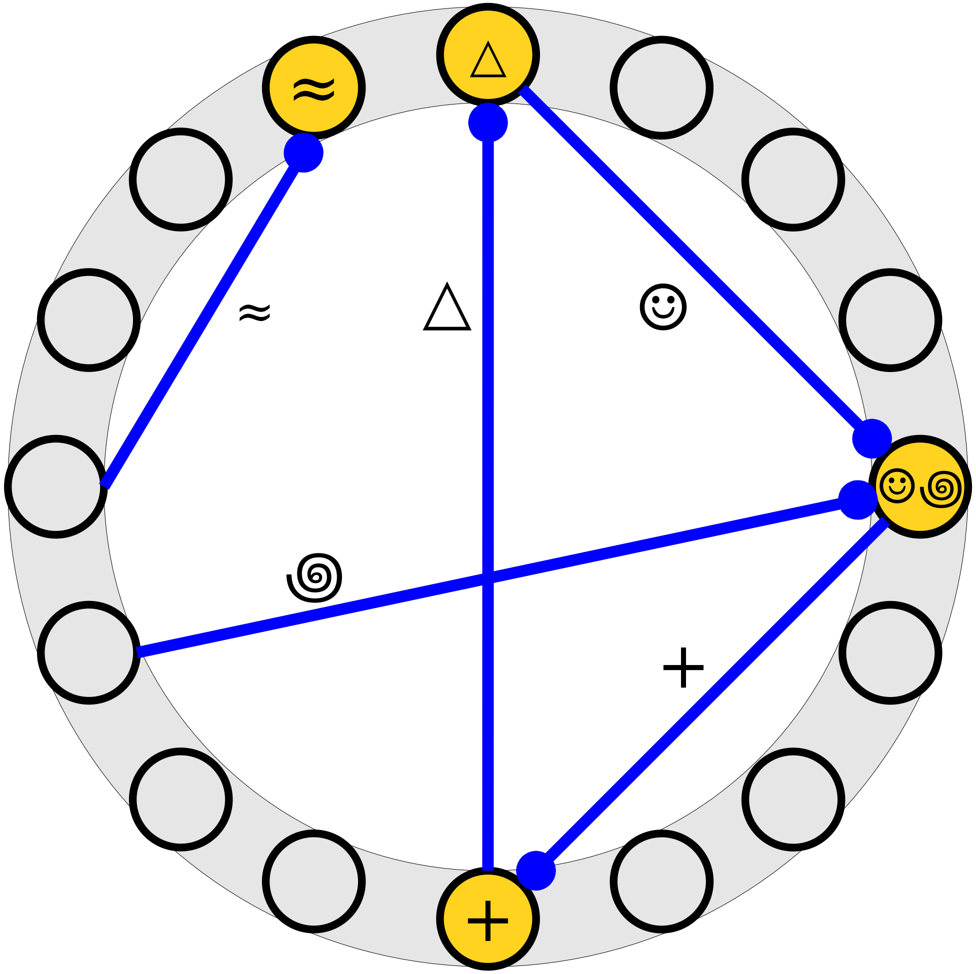
The Cuckoo Graph

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



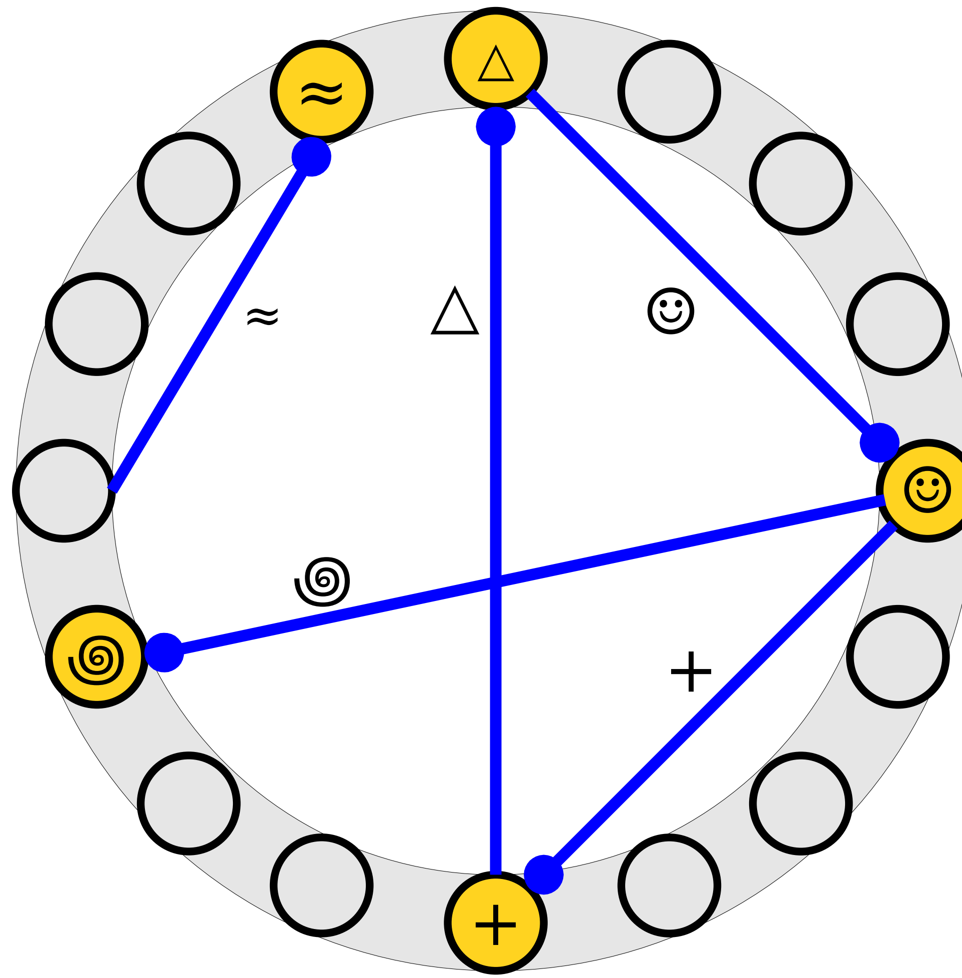
The Cuckoo Graph

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



The Cuckoo Graph

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

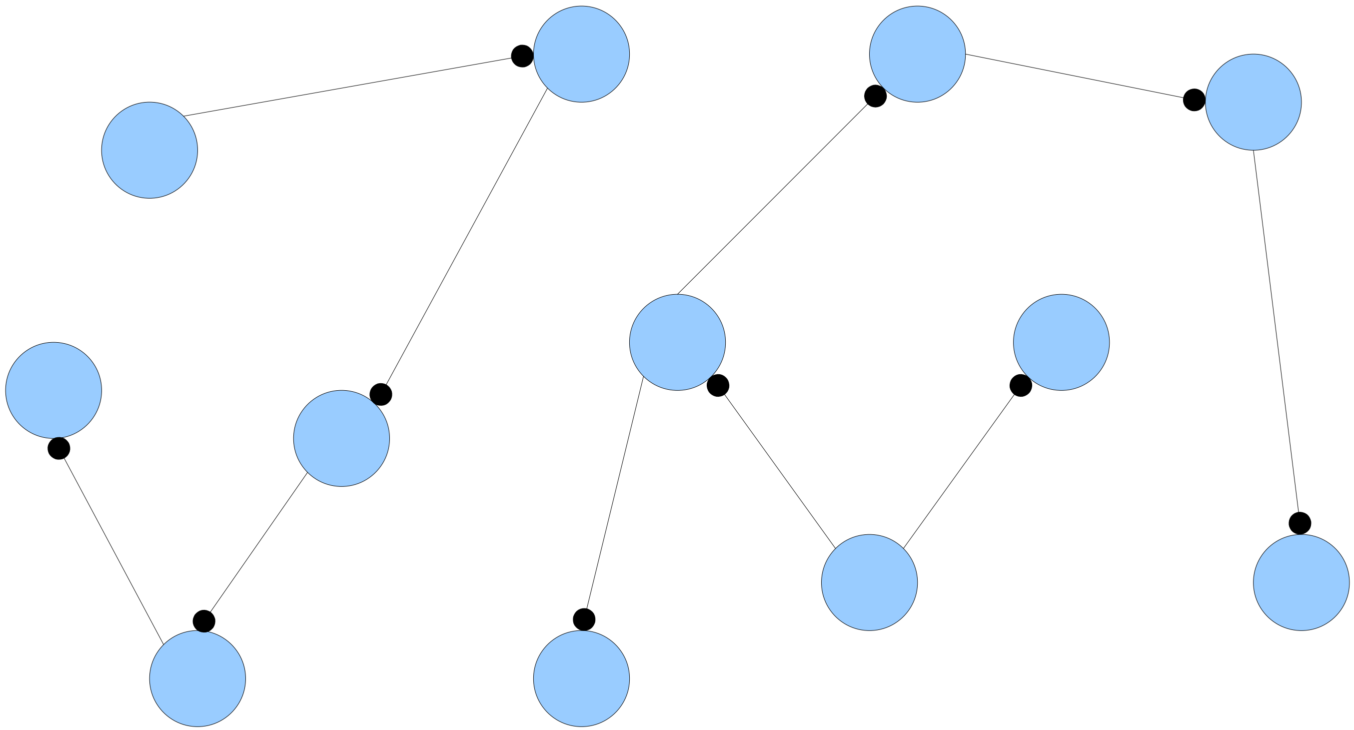


The Cuckoo Graph

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

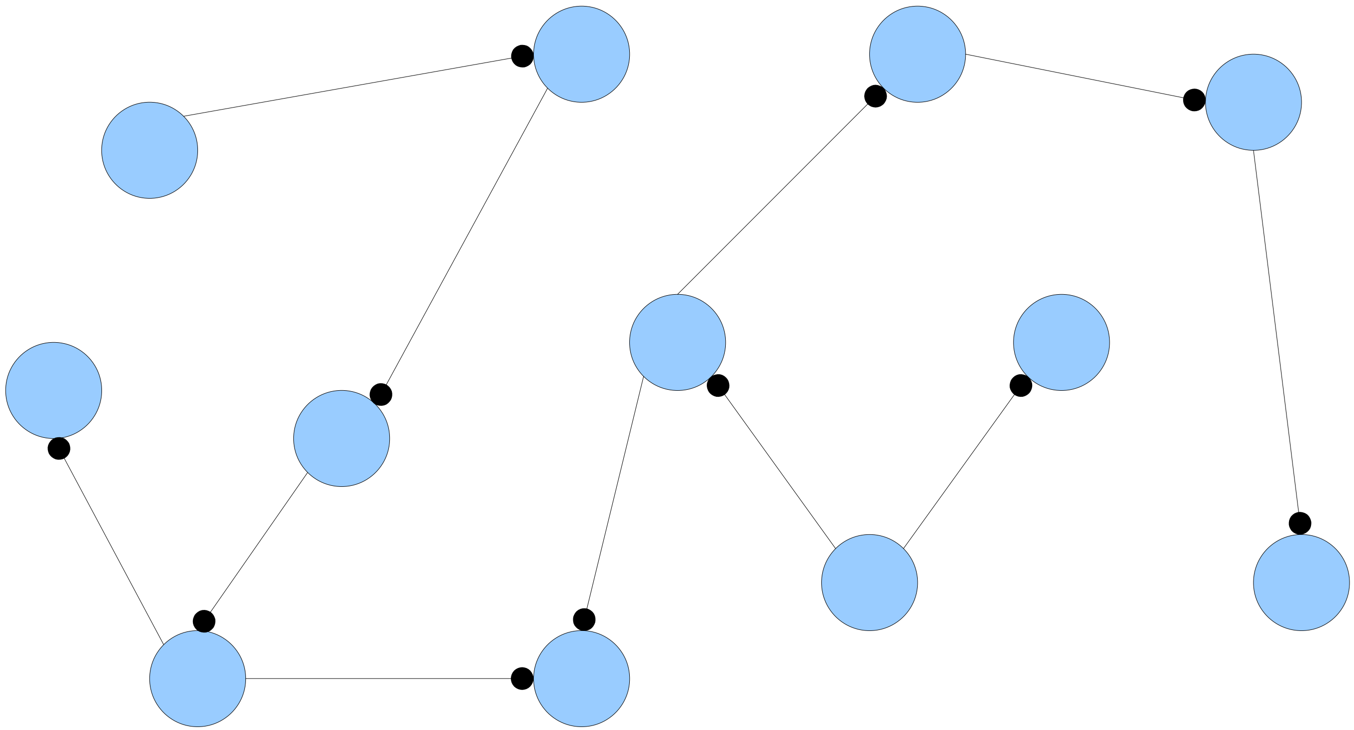
The Cuckoo Graph

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



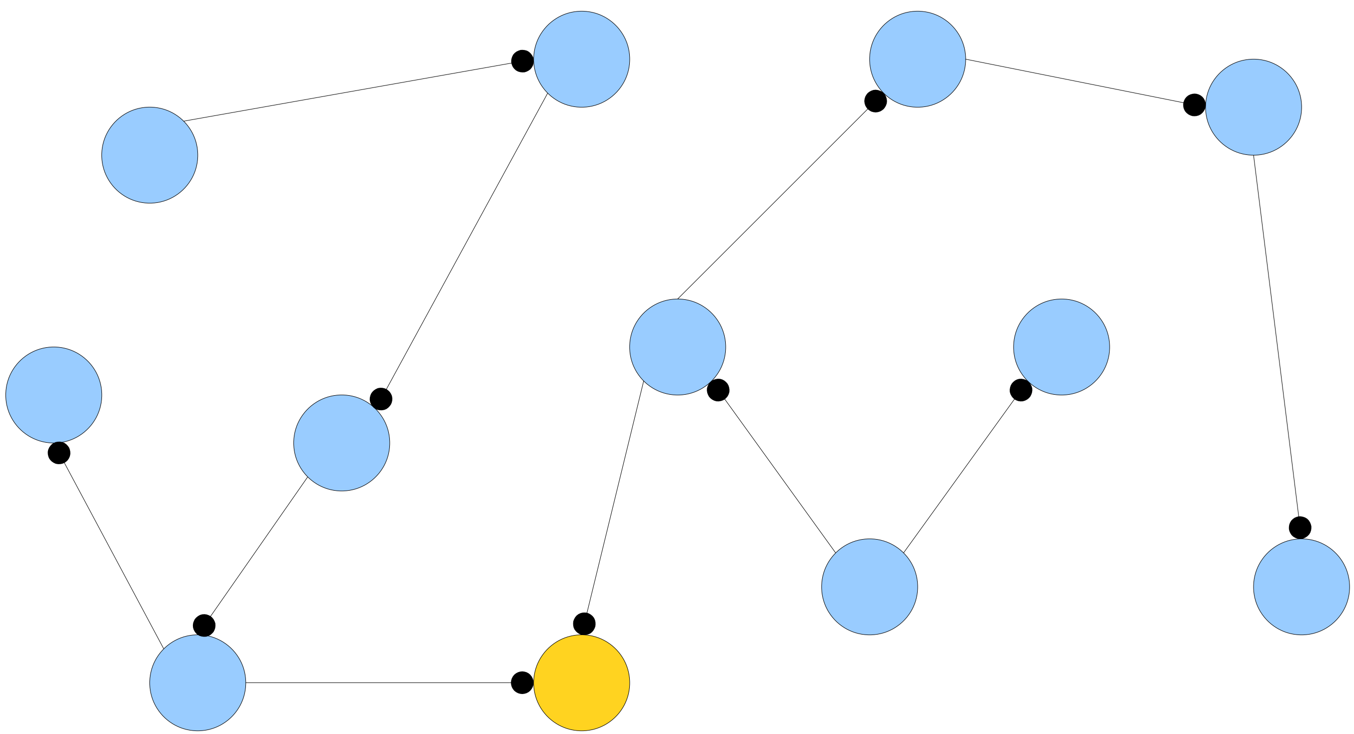
The Cuckoo Graph

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



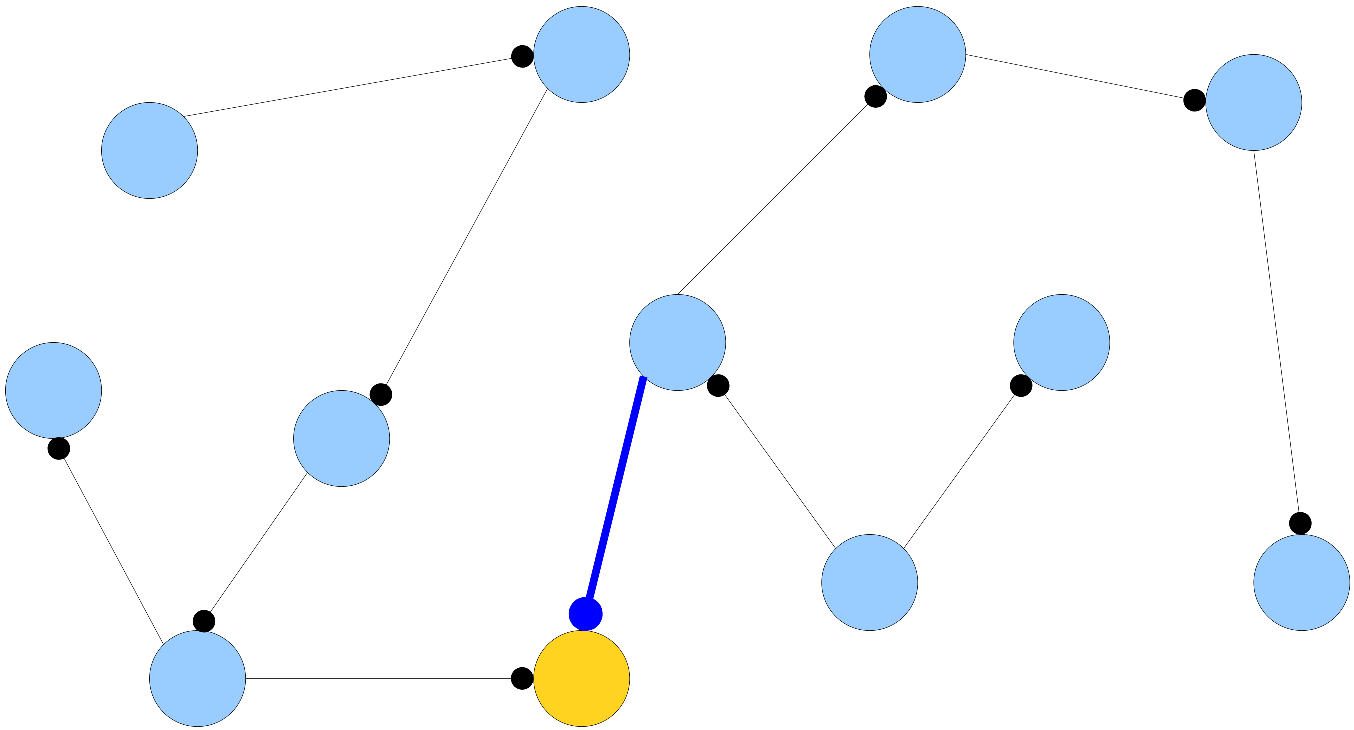
The Cuckoo Graph

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



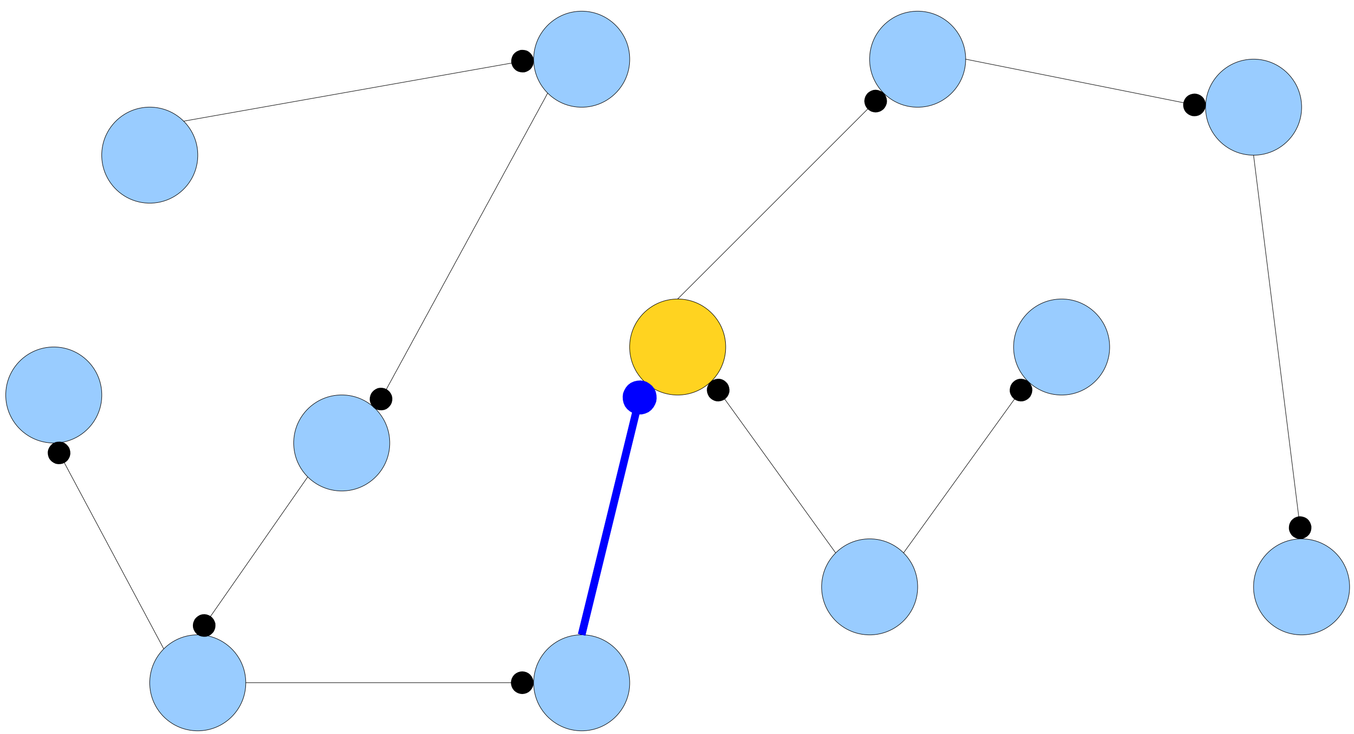
The Cuckoo Graph

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



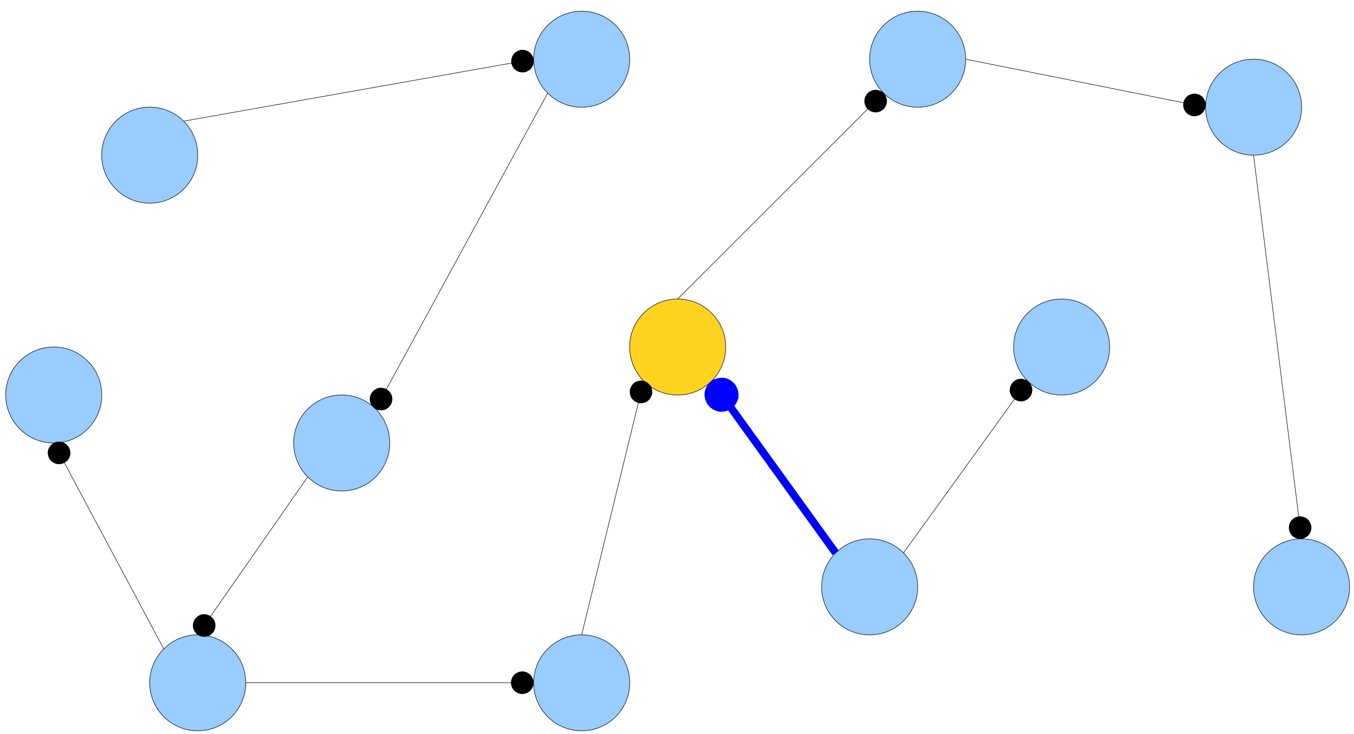
The Cuckoo Graph

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



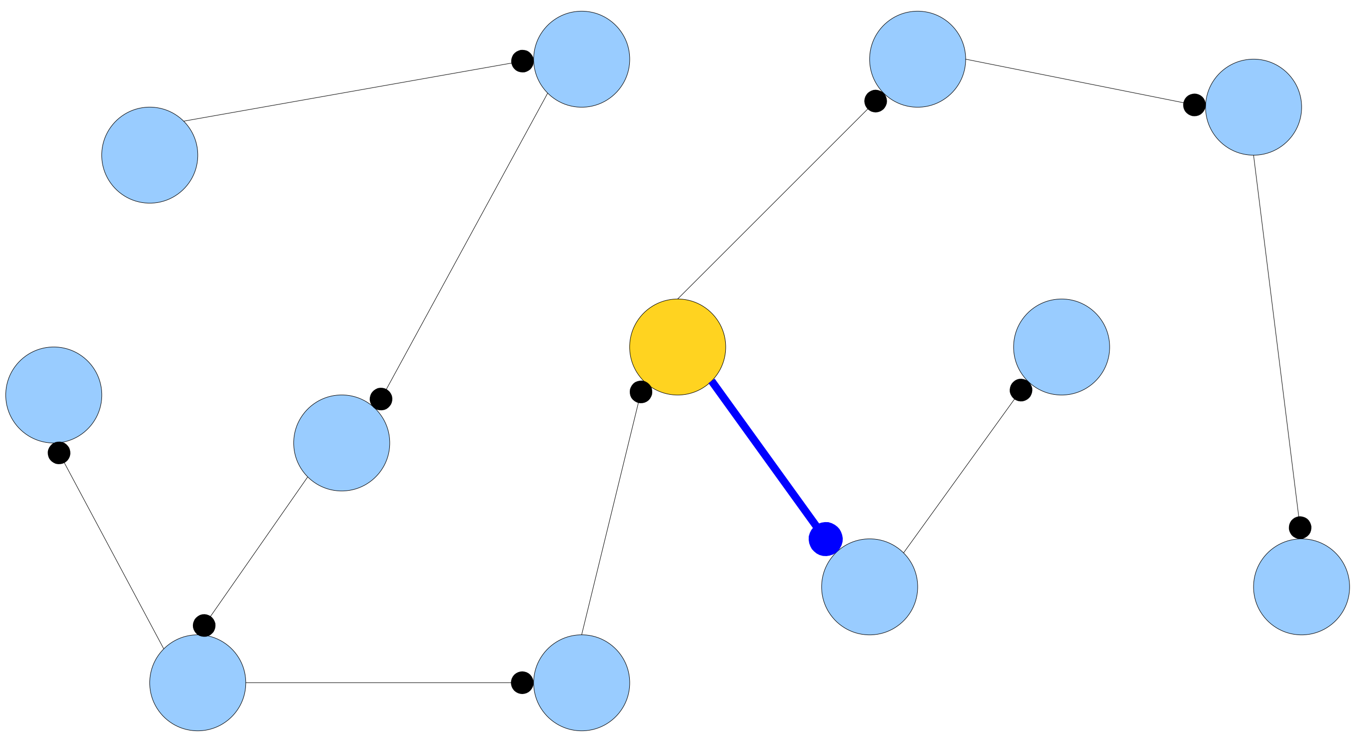
The Cuckoo Graph

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



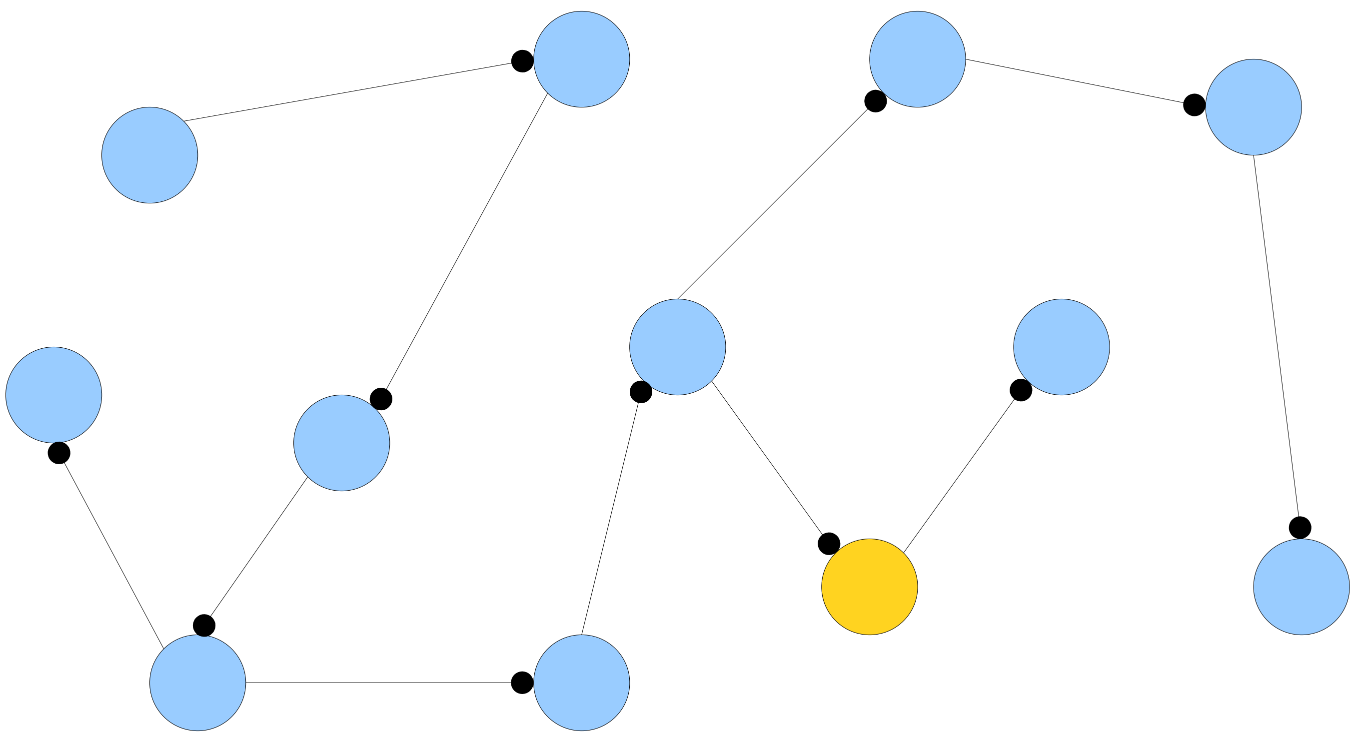
The Cuckoo Graph

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



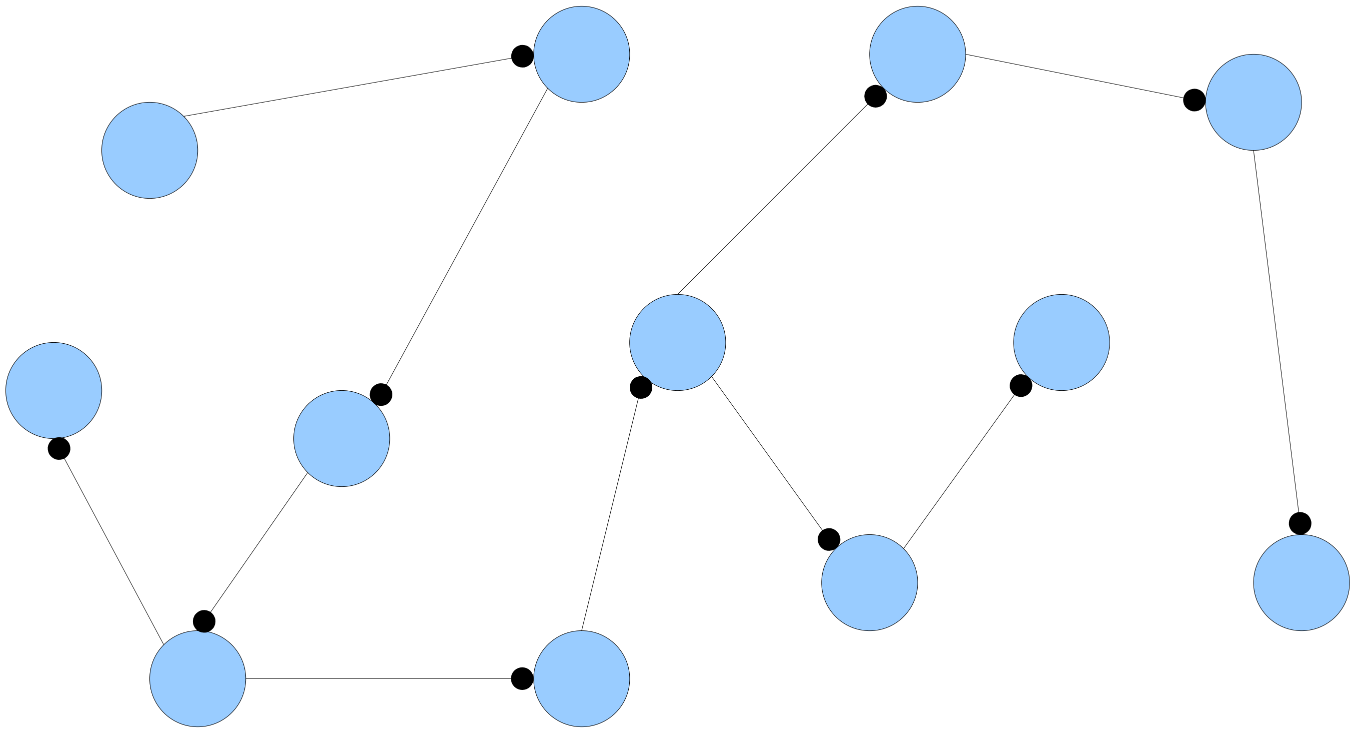
The Cuckoo Graph

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



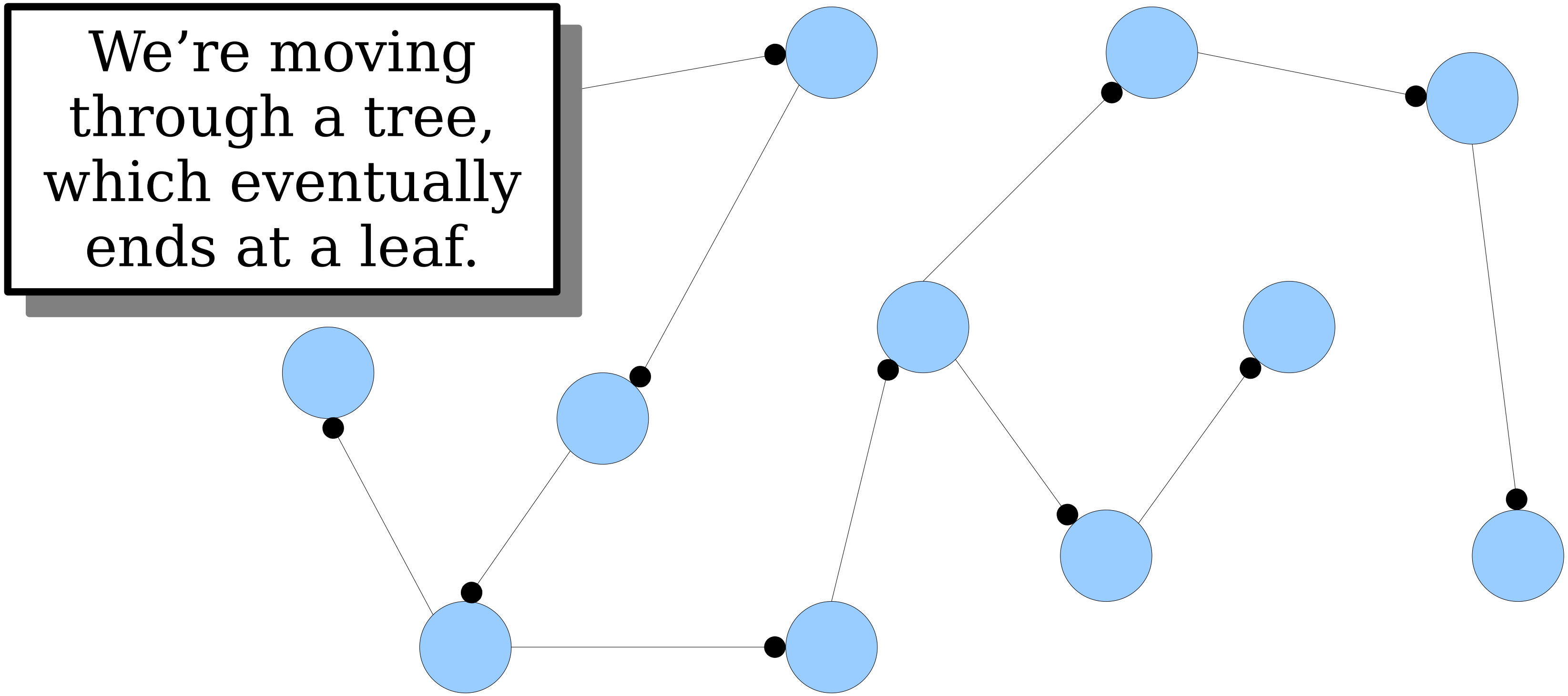
The Cuckoo Graph

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



The Cuckoo Graph

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

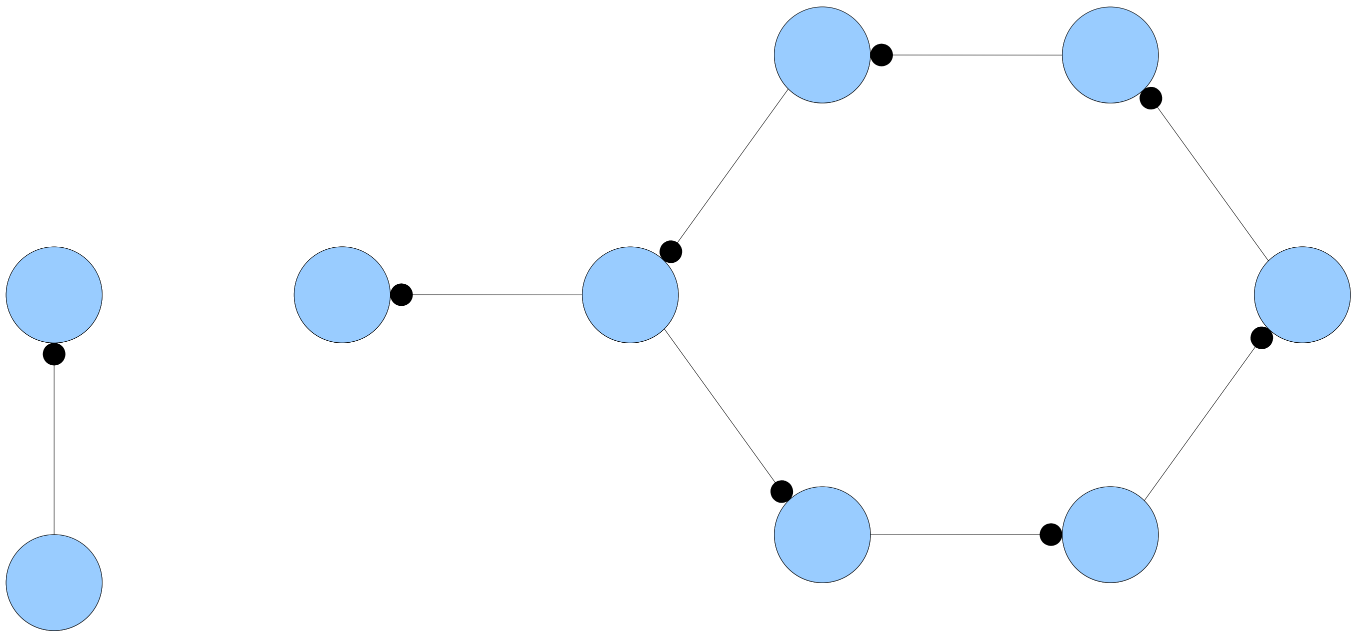


The Cuckoo Graph

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

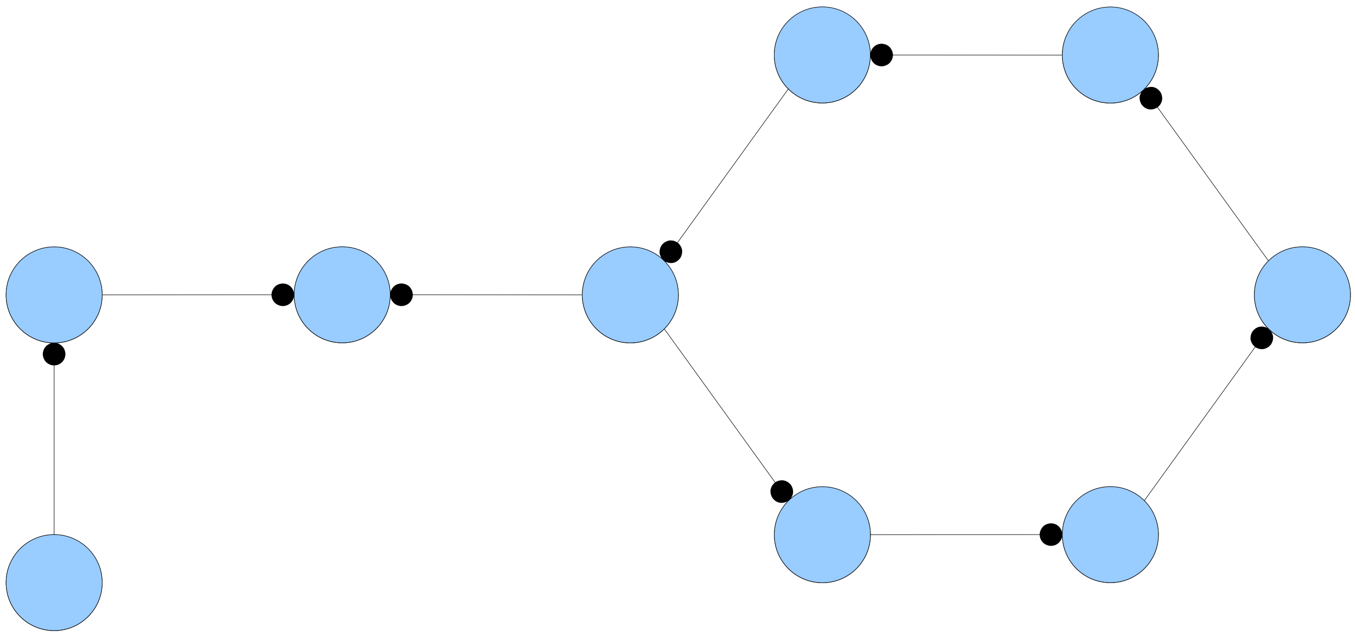
The Cuckoo Graph

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



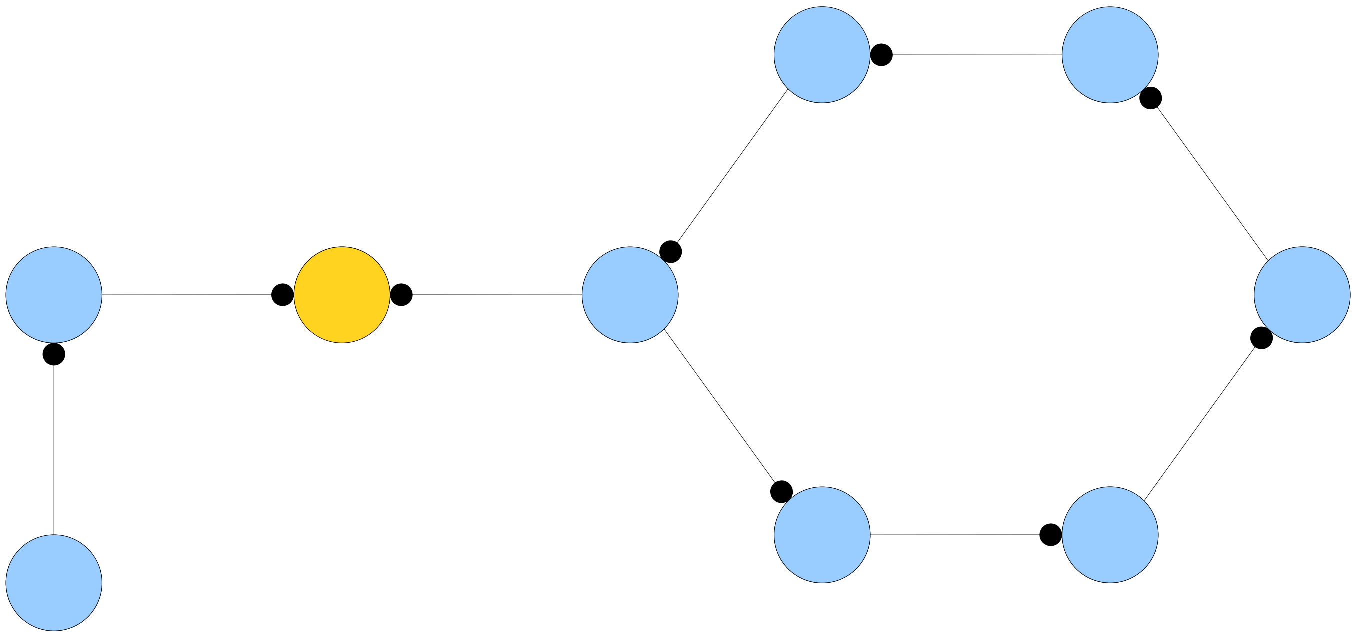
The Cuckoo Graph

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



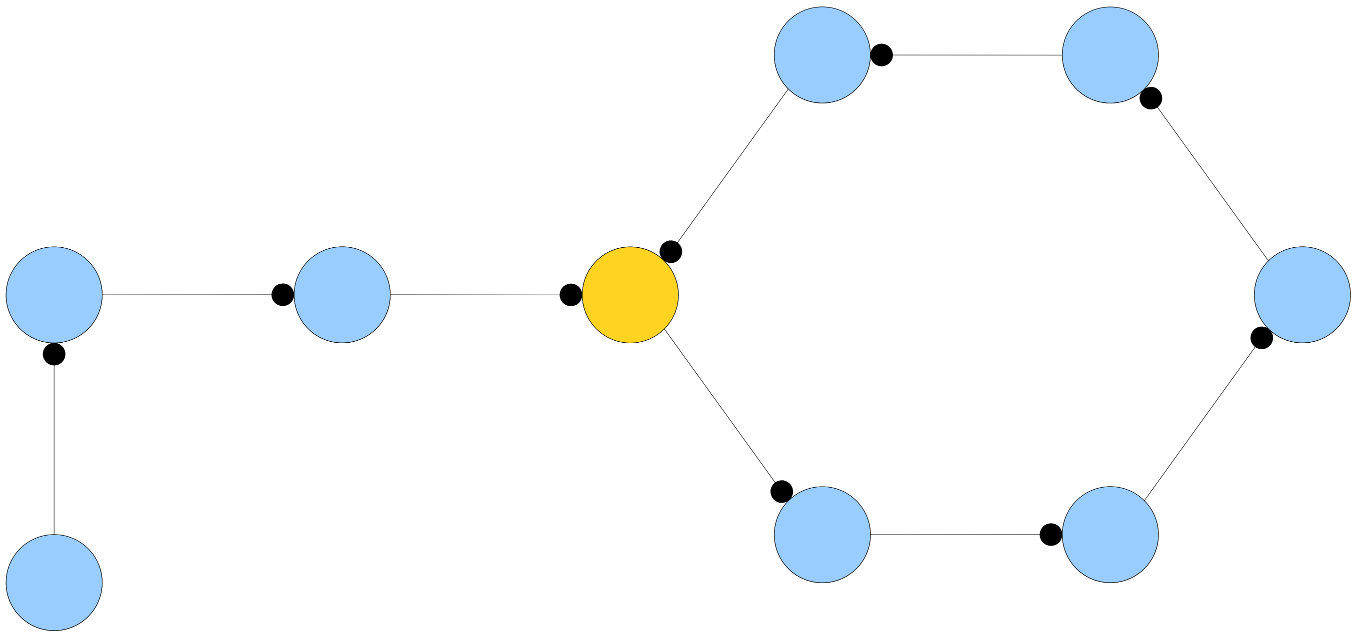
The Cuckoo Graph

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



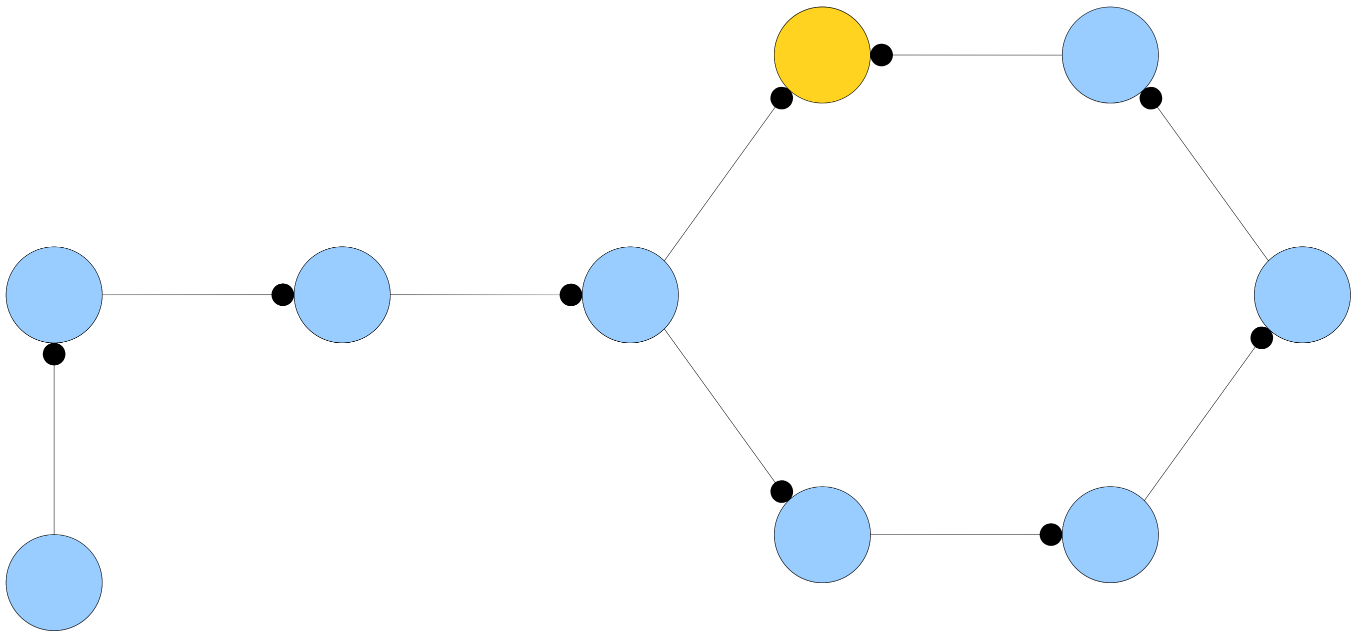
The Cuckoo Graph

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



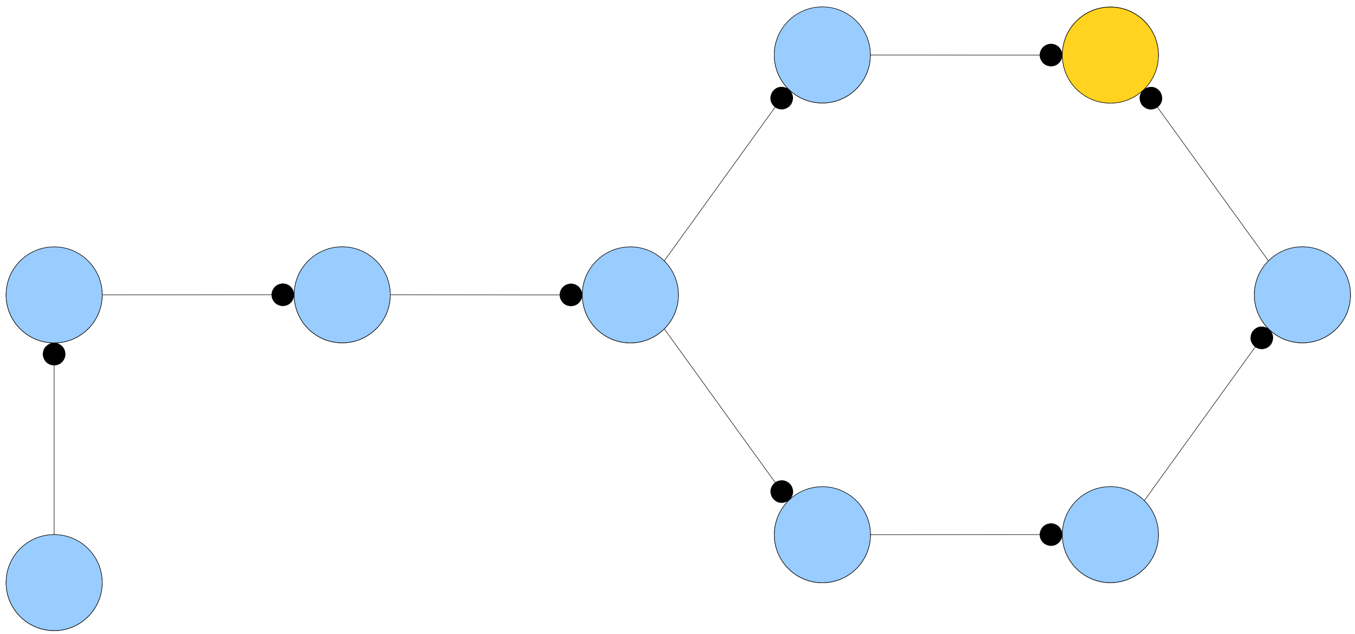
The Cuckoo Graph

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



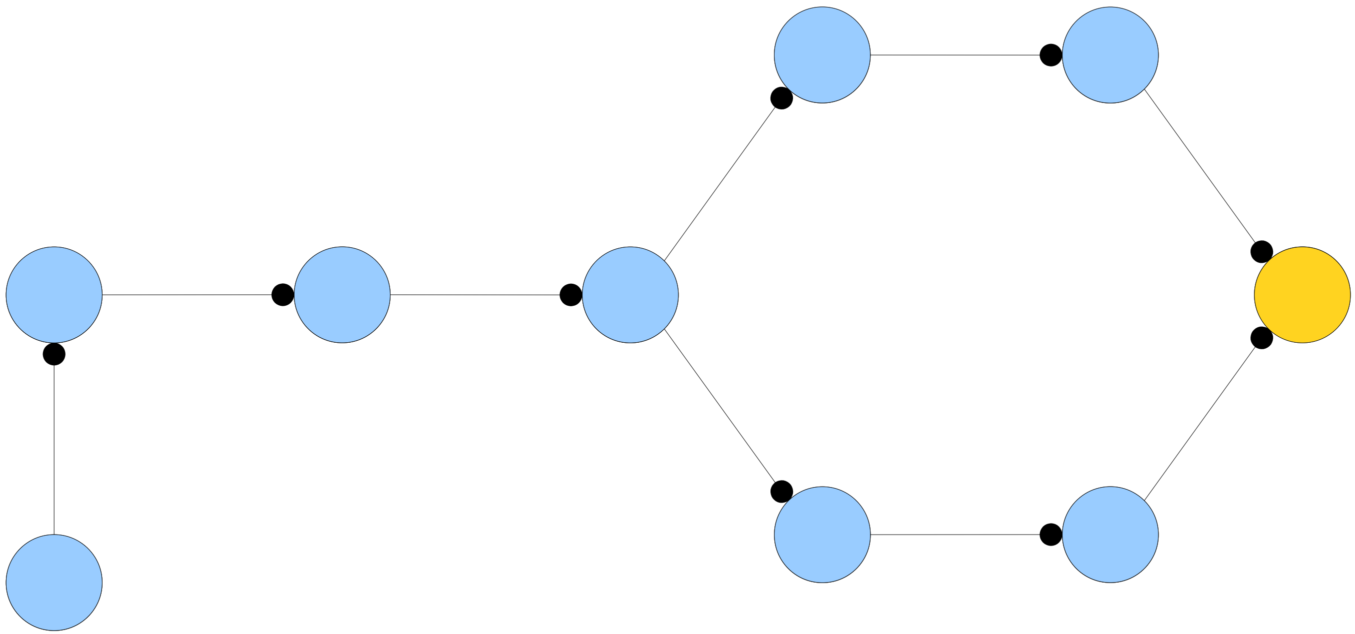
The Cuckoo Graph

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



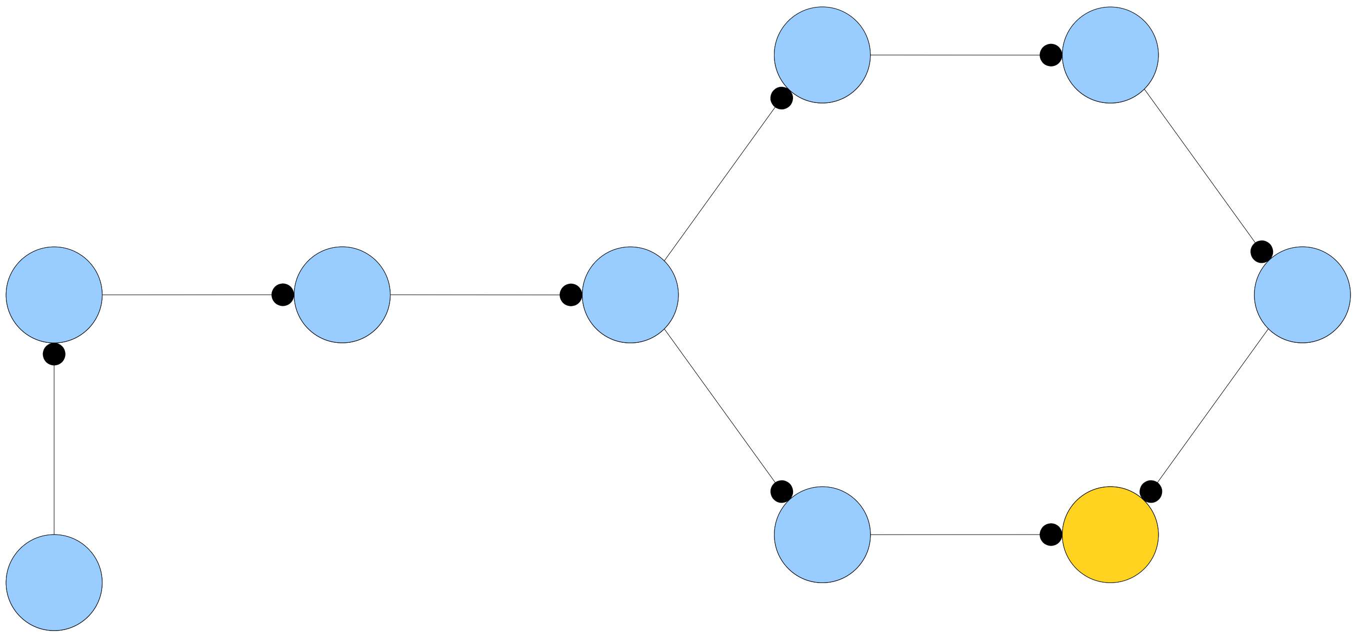
The Cuckoo Graph

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



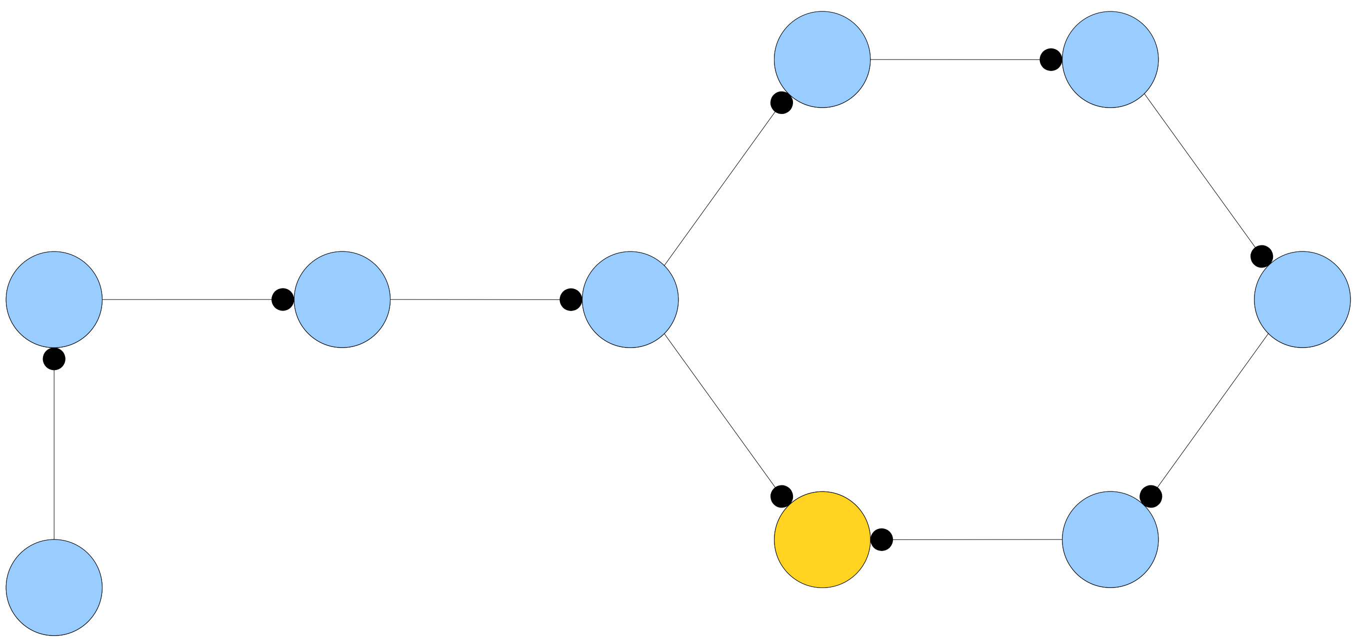
The Cuckoo Graph

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



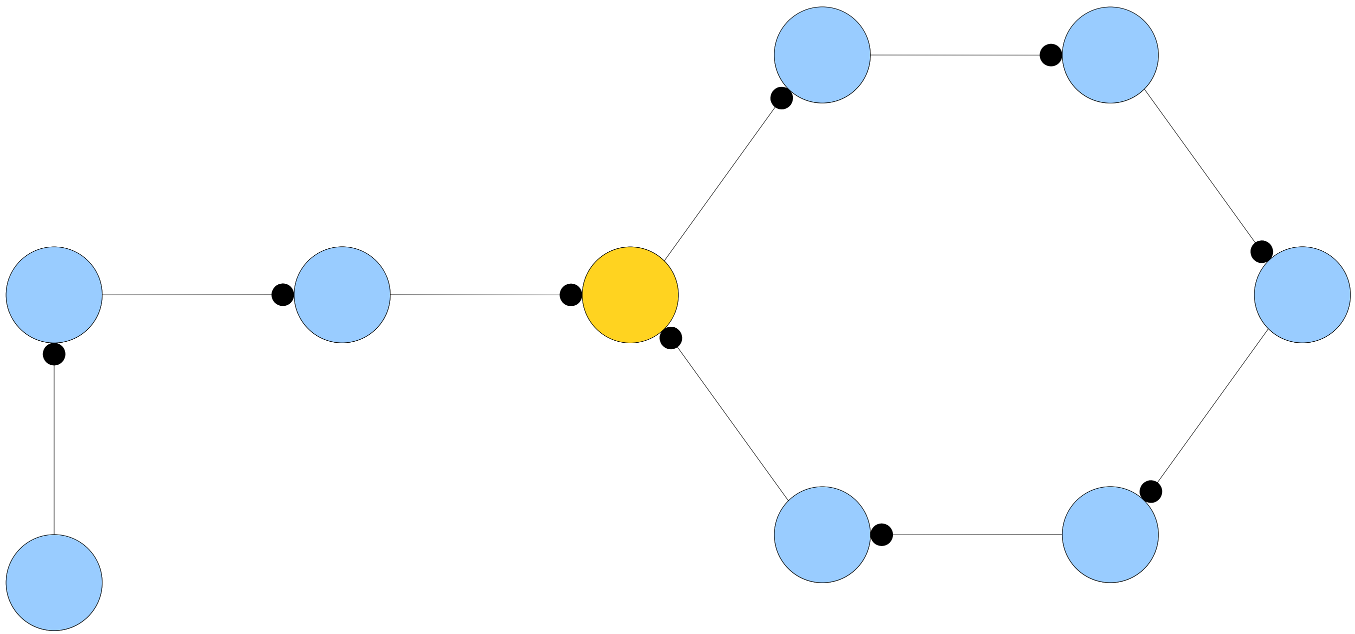
The Cuckoo Graph

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



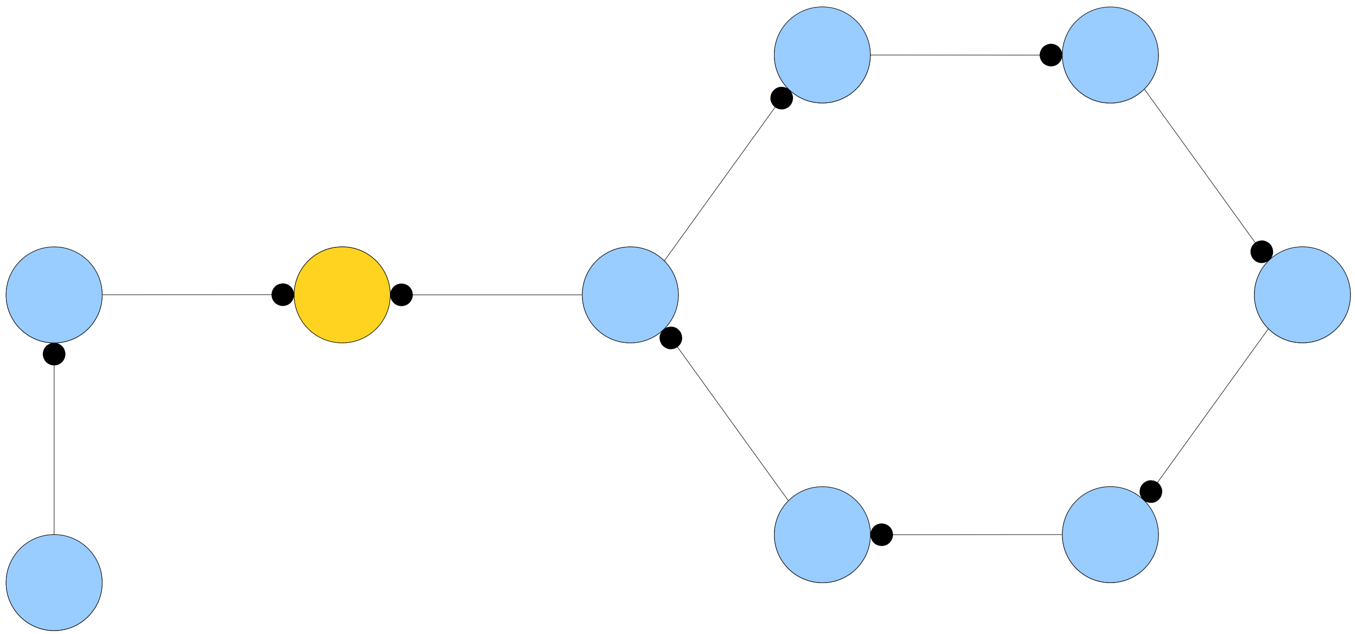
The Cuckoo Graph

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



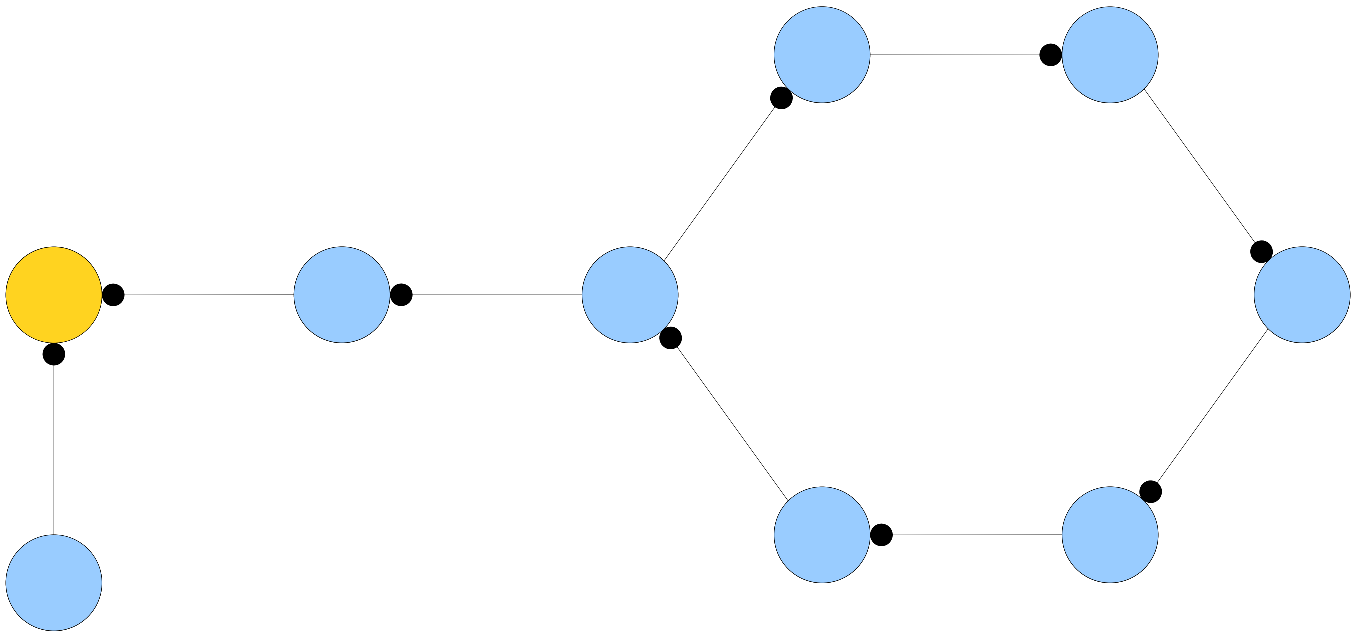
The Cuckoo Graph

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



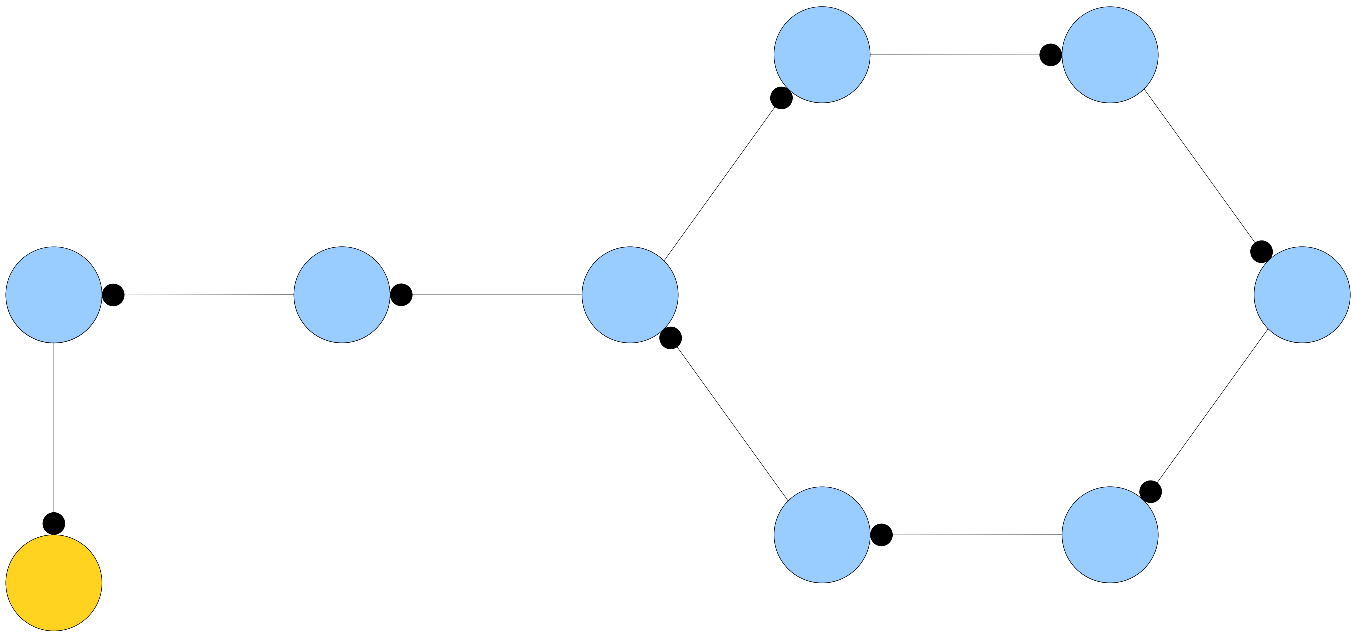
The Cuckoo Graph

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



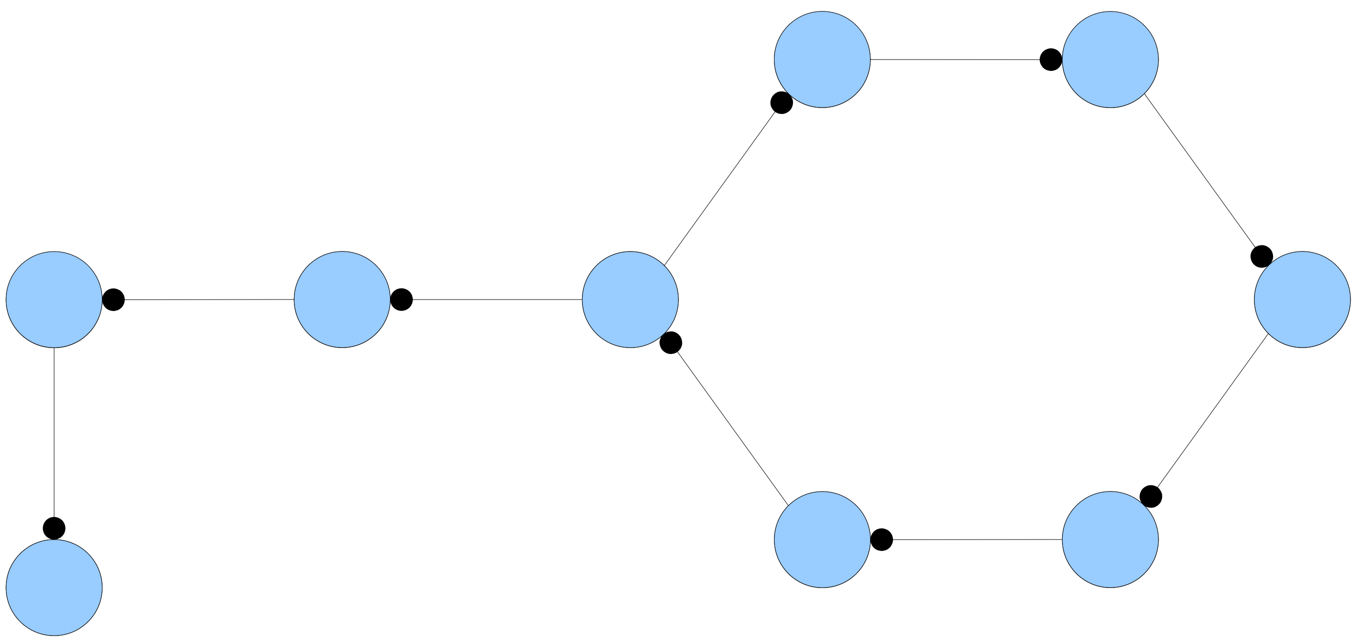
The Cuckoo Graph

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



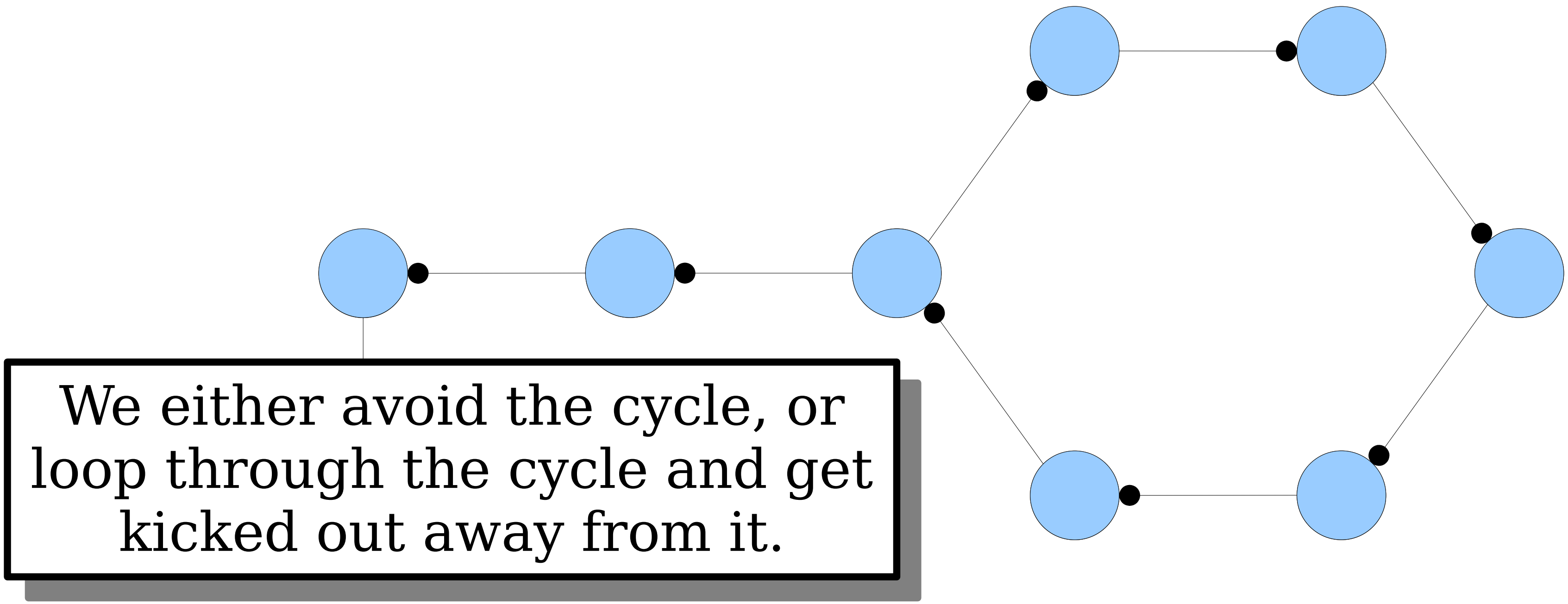
The Cuckoo Graph

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



The Cuckoo Graph

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



The Cuckoo Graph

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

Why?

Formulate a hypothesis, but ***don't post anything in chat just yet***.

The Cuckoo Graph

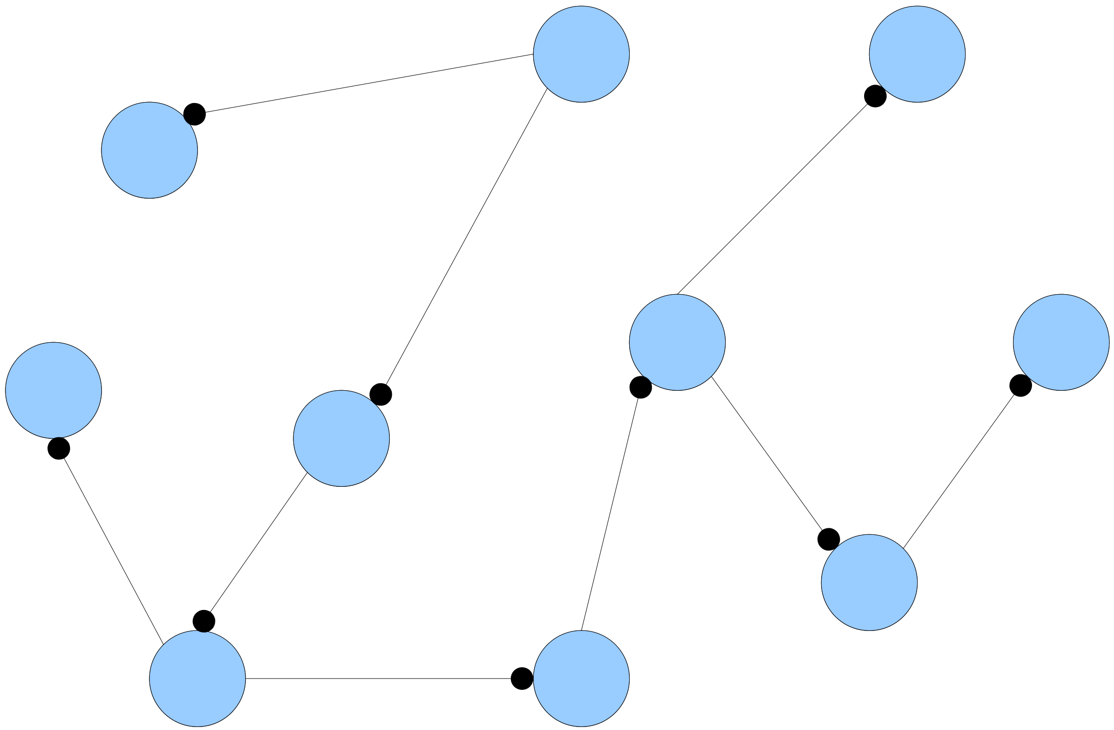
- ***Claim 2:*** If x is inserted into a cuckoo hash table, the insertion fails if the connected component containing x contains more than one cycle.

Why?

Now, ***private chat me your best guess***. Not sure? Just answer “??”.

The Cuckoo Graph

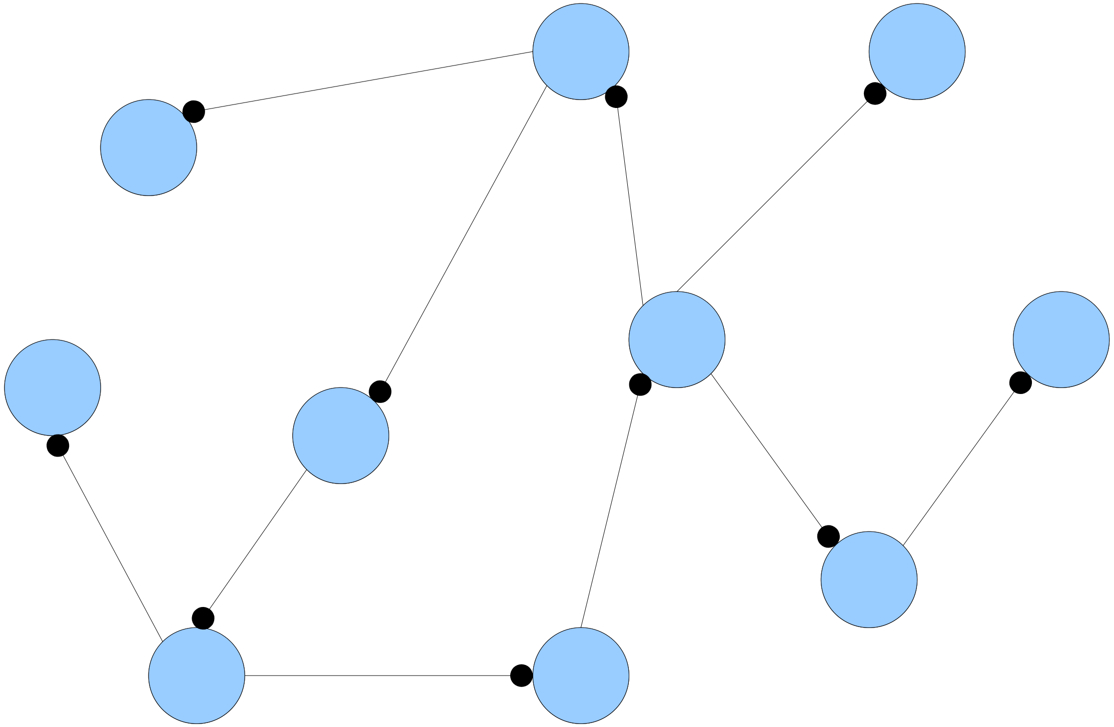
- ***Claim 2:*** If x is inserted into a cuckoo hash table, the insertion fails if the connected component containing x contains more than one cycle.



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

The Cuckoo Graph

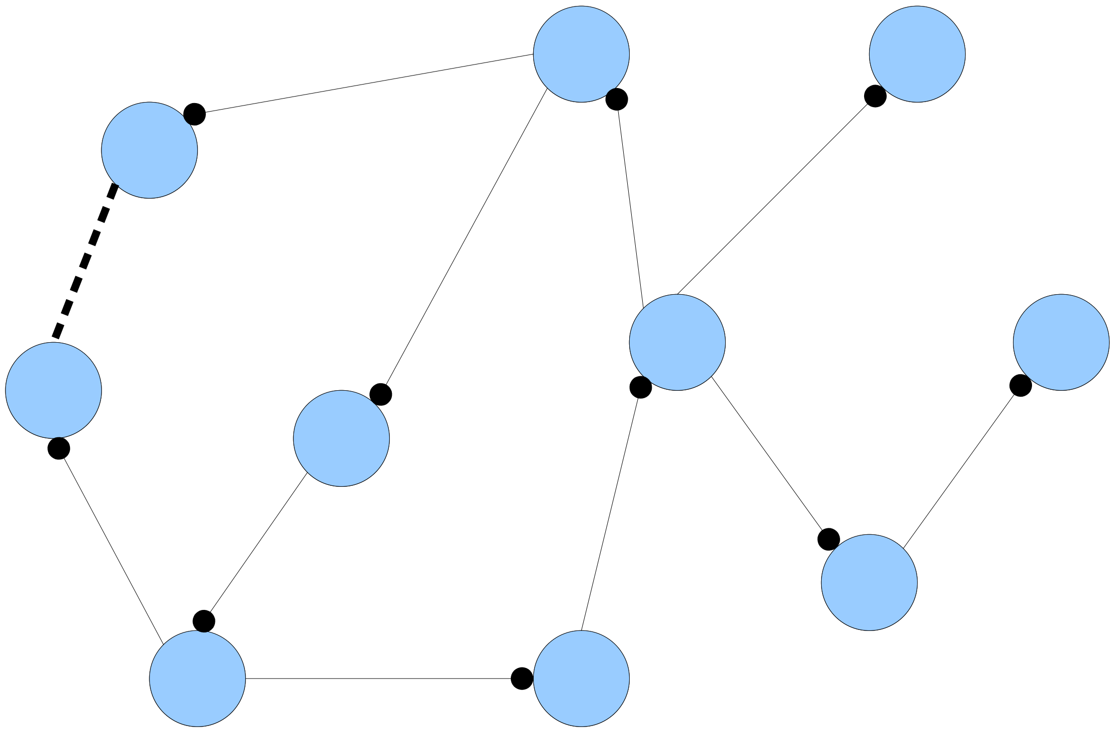
- ***Claim 2:*** If x is inserted into a cuckoo hash table, the insertion fails if the connected component containing x contains more than one cycle.



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

The Cuckoo Graph

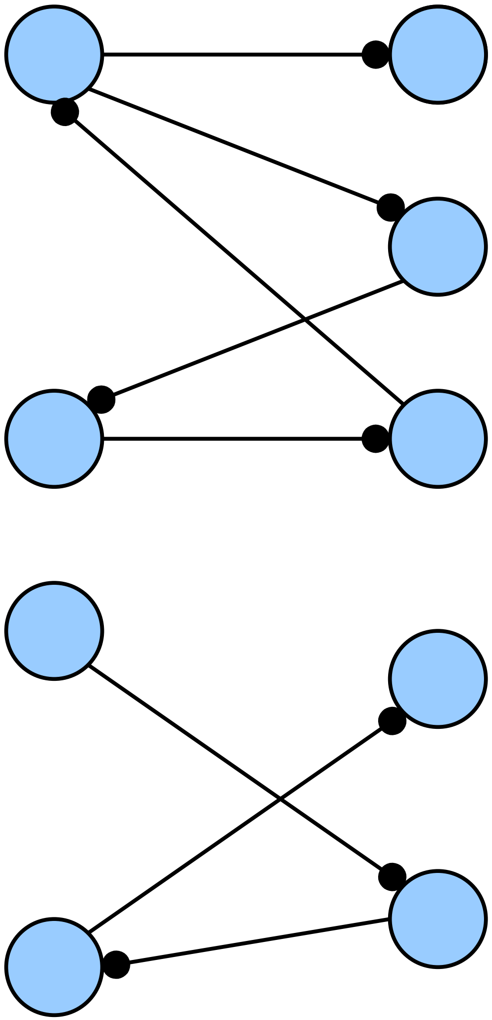
- ***Claim 2:*** If x is inserted into a cuckoo hash table, the insertion fails if the connected component containing x contains more than one cycle.



Two cycles: There are k nodes and $k+1$ edges. There are too many edges to place at most one item per node.

The Cuckoo Graph

- A connected component of a graph is called ***complex*** if it contains two or more cycles.
- ***Theorem:*** Insertion into a cuckoo hash table succeeds if and only if the resulting cuckoo graph has no complex connected components.



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

How big are the connected components in the cuckoo graph?

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

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

How big are the connected components in the cuckoo graph?

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

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

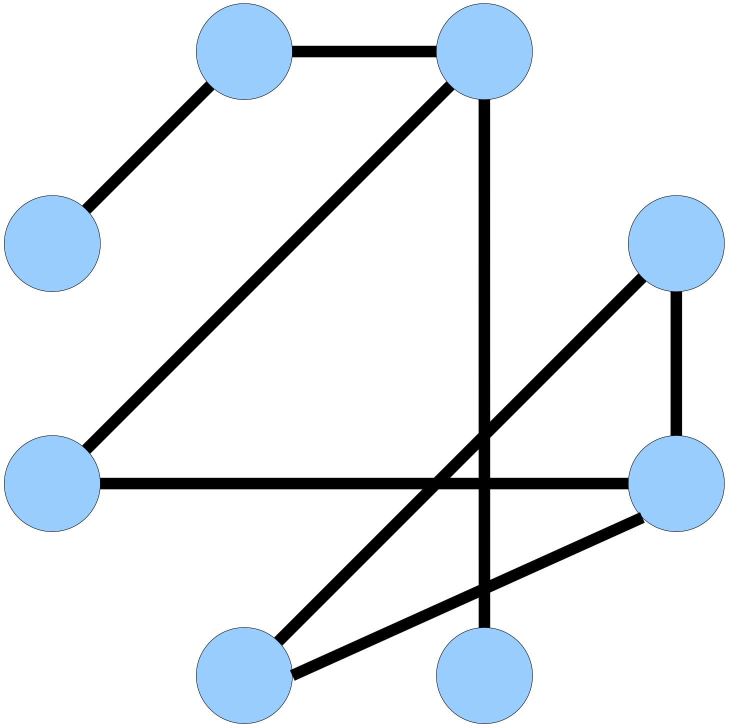
How big are the connected components in the cuckoo graph?

(This tells us how much work we do on a successful insertion.)

The Erdős–Rényi model

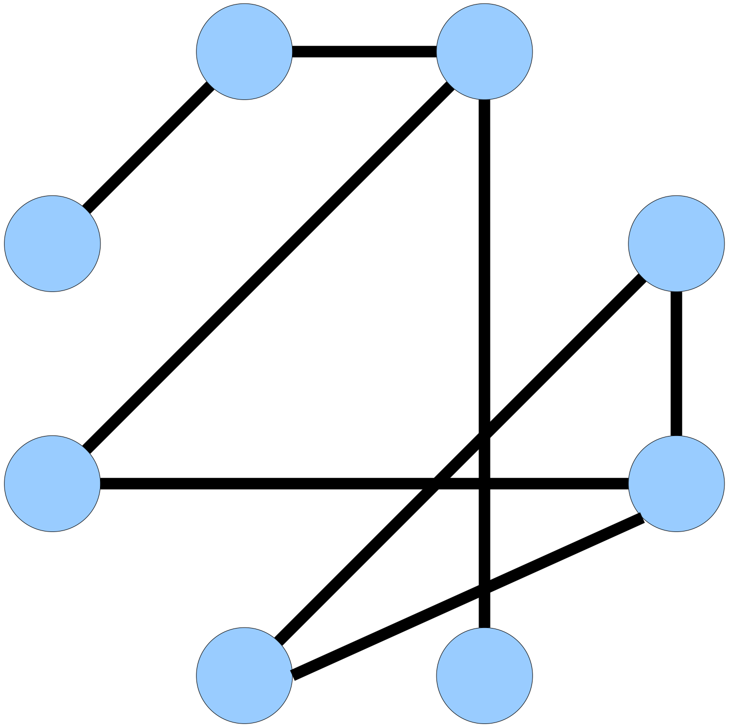
Random Graph Evolution

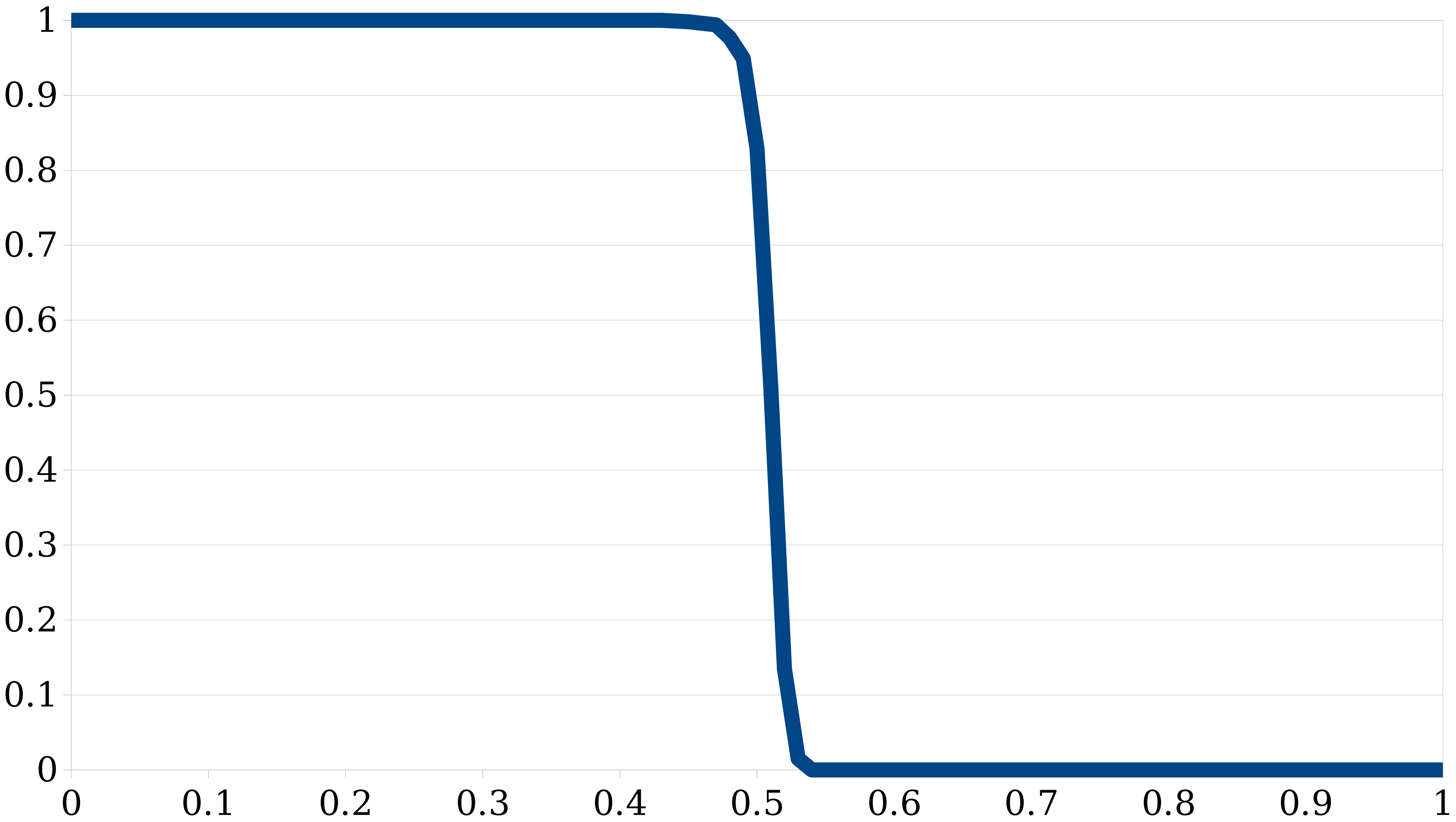
- Consider a graph with V nodes and no edges.
- Incrementally add E edges to the graph, each chosen uniformly at random, possibly with repetition.
- ***Question:*** What properties will this graph (probably) have?



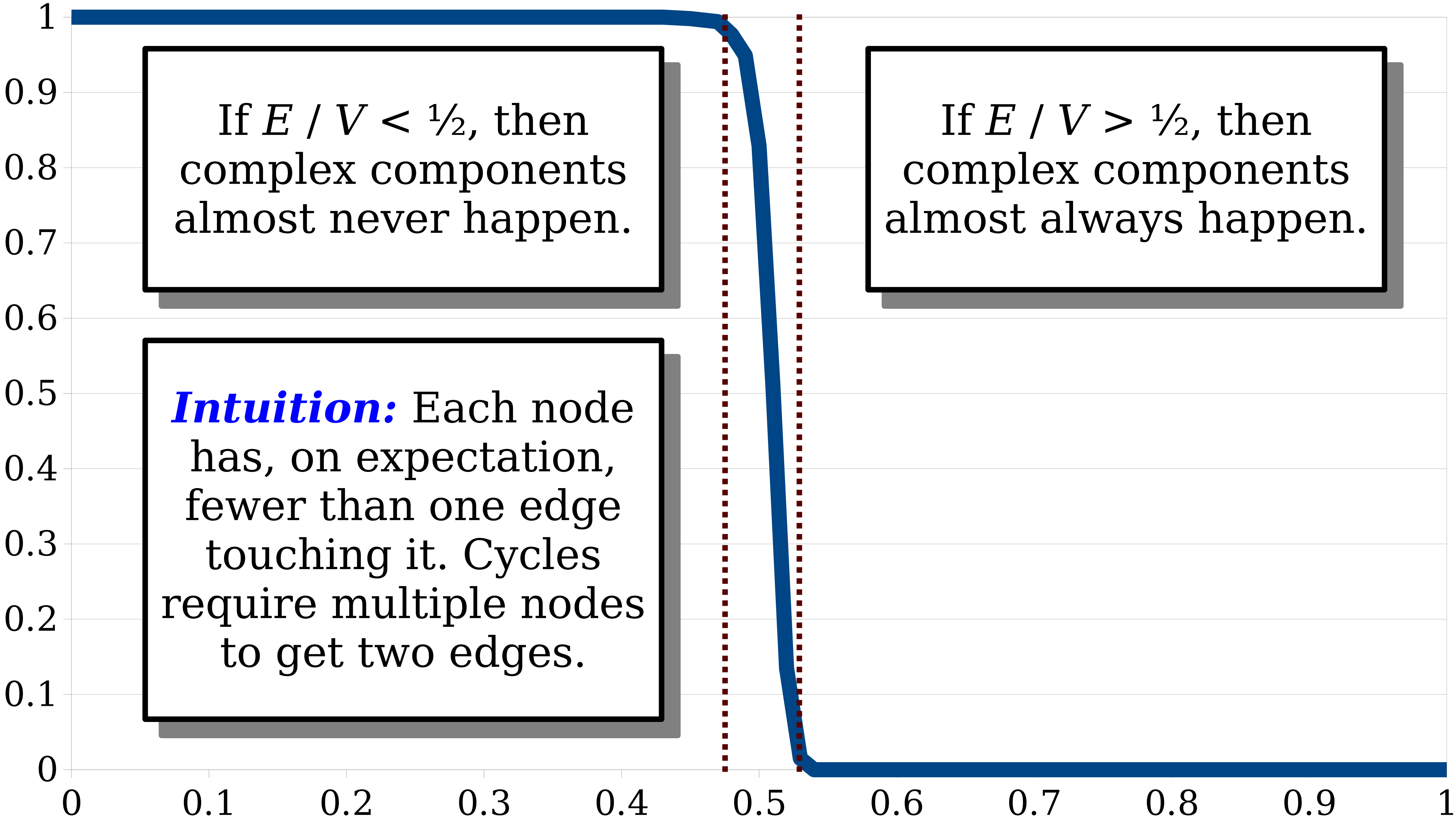
Random Graph Evolution

- ***Claim:*** The phenomena we're observing with cuckoo hashing are, in large part, due to properties of random graphs.
- ***Good News:*** This is a well-studied field! All the results we need were first proved by Erdős and Rényi in 1960.
- This model of incrementally constructing a graph is therefore called the ***Erdős–Rényi*** model.

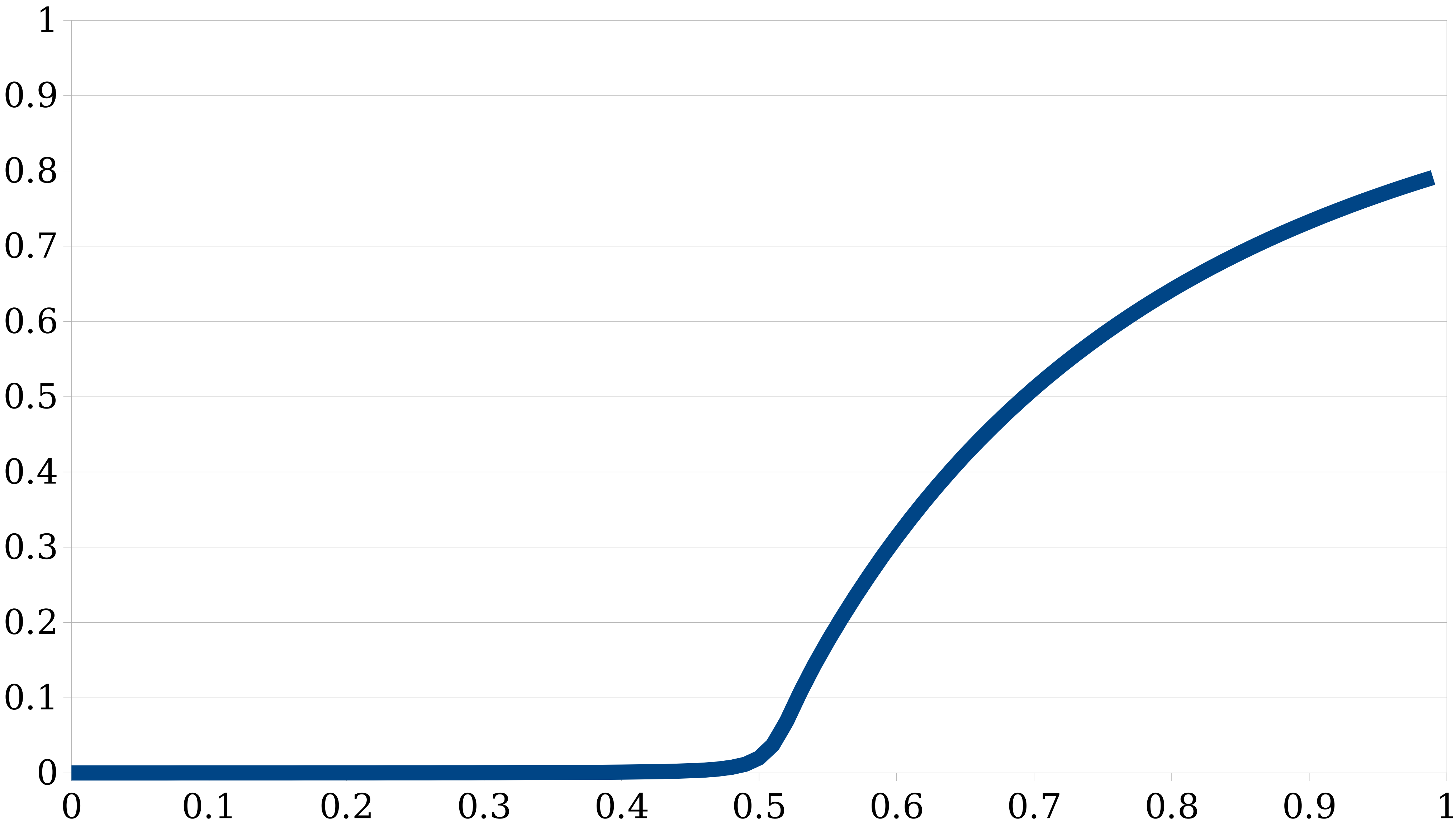




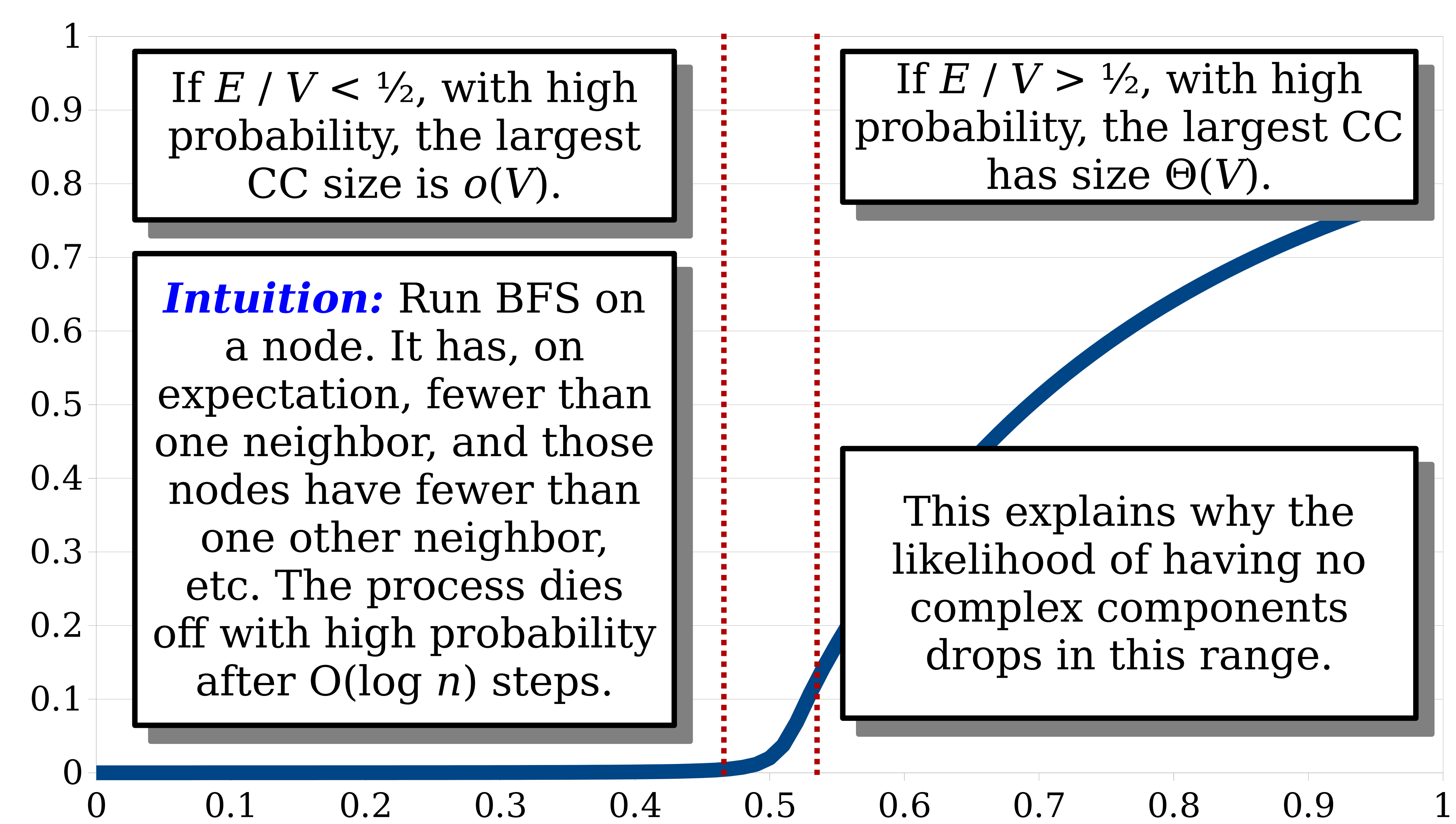
Consider a random (multi)graph G with V nodes and E edges. What is the the probability that every connected component in G is simple, as a function of the ratio E / V?



Consider a random (multi)graph G with V nodes and E edges. What is the the probability that every connected component in G is simple, as a function of the ratio E / V?



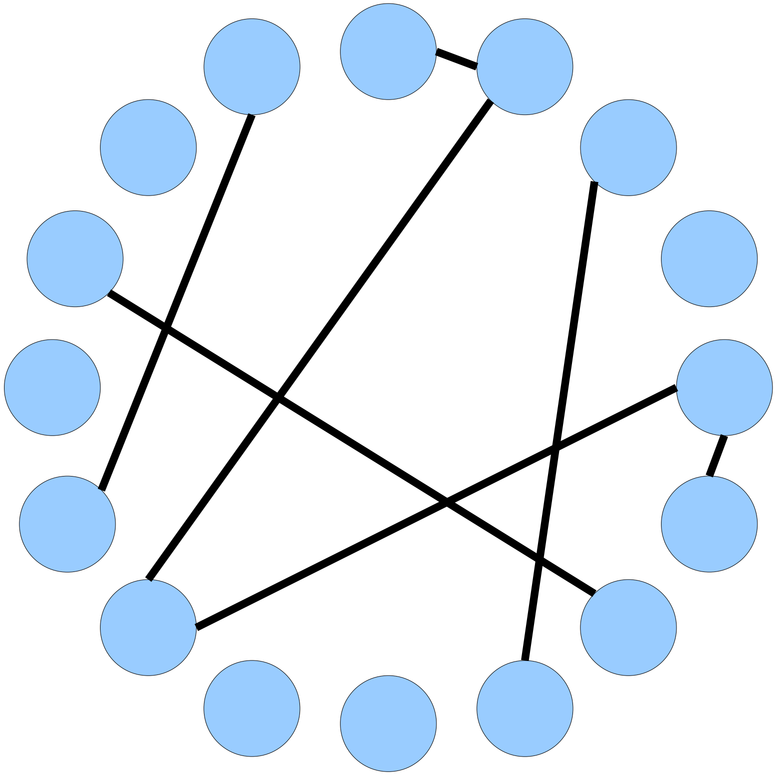
Consider a random (multi)graph G with V nodes and E edges. What fraction of the nodes are in the largest connected component of G, as a function of E / V?



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

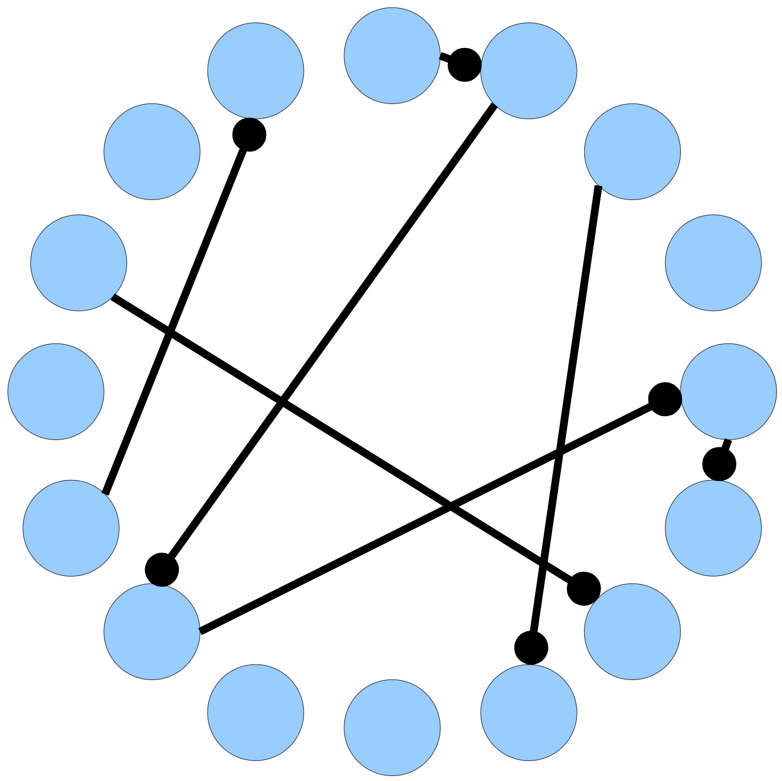
Random Graph Theory

- ***Theorem:*** Let $n = \alpha m$ with $\alpha < ½$. Then the expected size of a connected component in a randomly-chosen graph with m nodes and n edges is O(1), and with high probability the largest connected component has size O(log n).
- ***Corollary:*** Using cuckoo hashing with m slots and $n = \alpha m$ items, for $\alpha < ½$, assuming all inserts succeed, the expected cost of each insert is O(1), and the worst-case cost of each insert is O(log n) with high probability.



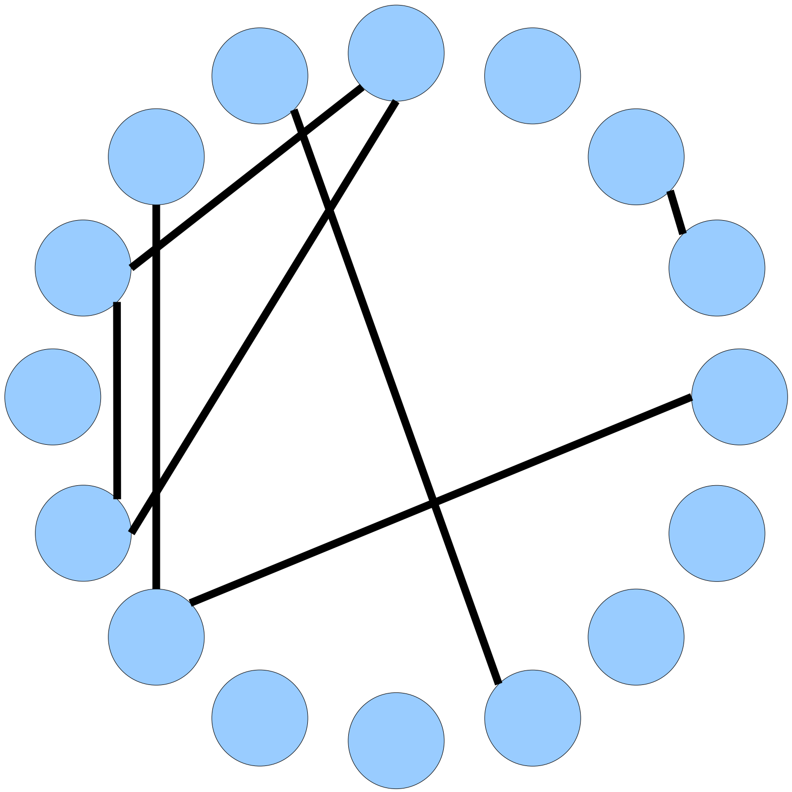
Random Graph Theory

- ***Theorem:*** Let $n = \alpha m$ with $\alpha < ½$. Then the expected size of a connected component in a randomly-chosen graph with m nodes and n edges is O(1), and with high probability the largest connected component has size O(log n).
- ***Corollary:*** Using cuckoo hashing with m slots and $n = \alpha m$ items, for $\alpha < ½$, assuming all inserts succeed, the expected cost of each insert is O(1), and the worst-case cost of each insert is O(log n) with high probability.



Random Graph Theory

- ***Theorem:*** Let $n = \alpha m$ for some $\alpha < \frac{1}{2}$. Then the probability that any connected component is complex is $O(1/n)$.
- ***Corollary:*** Using cuckoo hashing with m slots and $n = \alpha m$ items, the probability that a series of n insertions fails is $O(1/n)$, and the expected number of times a rehash is required before it succeeds is $O(1)$.



The Overall Analysis

- Cuckoo hashing gives worst-case lookups and deletions.
- Insertions are expected, amortized O(1).
  - The amortization kicks in because we need to periodically double the sizes of the table as the number of elements increases.
- The hidden constants are small, and this is a practical technique for building hash tables.

Cuckoo Hashing:

- ***lookup***: O(1)
- ***insert***: O(1)*
- ***delete***: O(1)

* *expected, amortized*

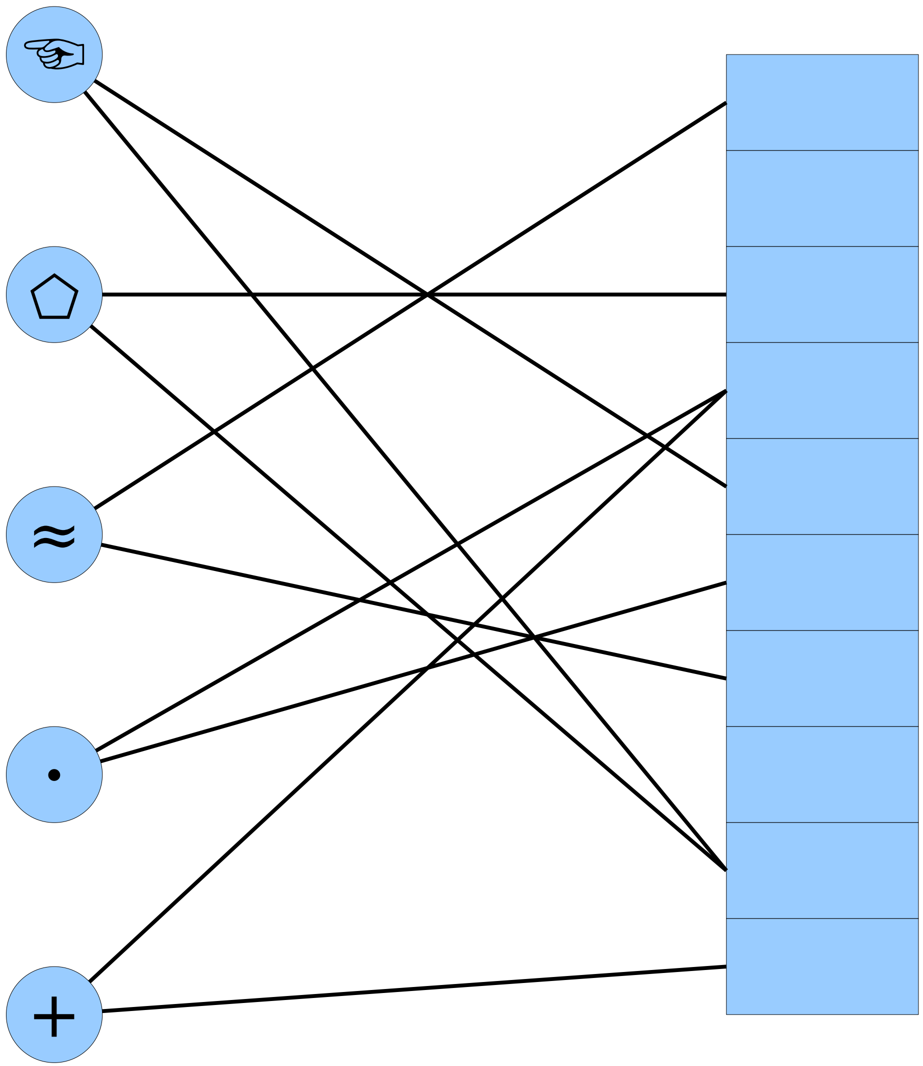
Improving Our Space Usage

Improving Space Usage

- A cuckoo hash table with n elements requires a table of size n / α, with $\alpha < ½$.
- This means at least 50% of the table slots will be empty.
- The root cause is a fundamental property of random graphs; exceeding this threshold makes failure almost certain.
- ***Question:*** How can we push past this to improve cuckoo hashing space usage?

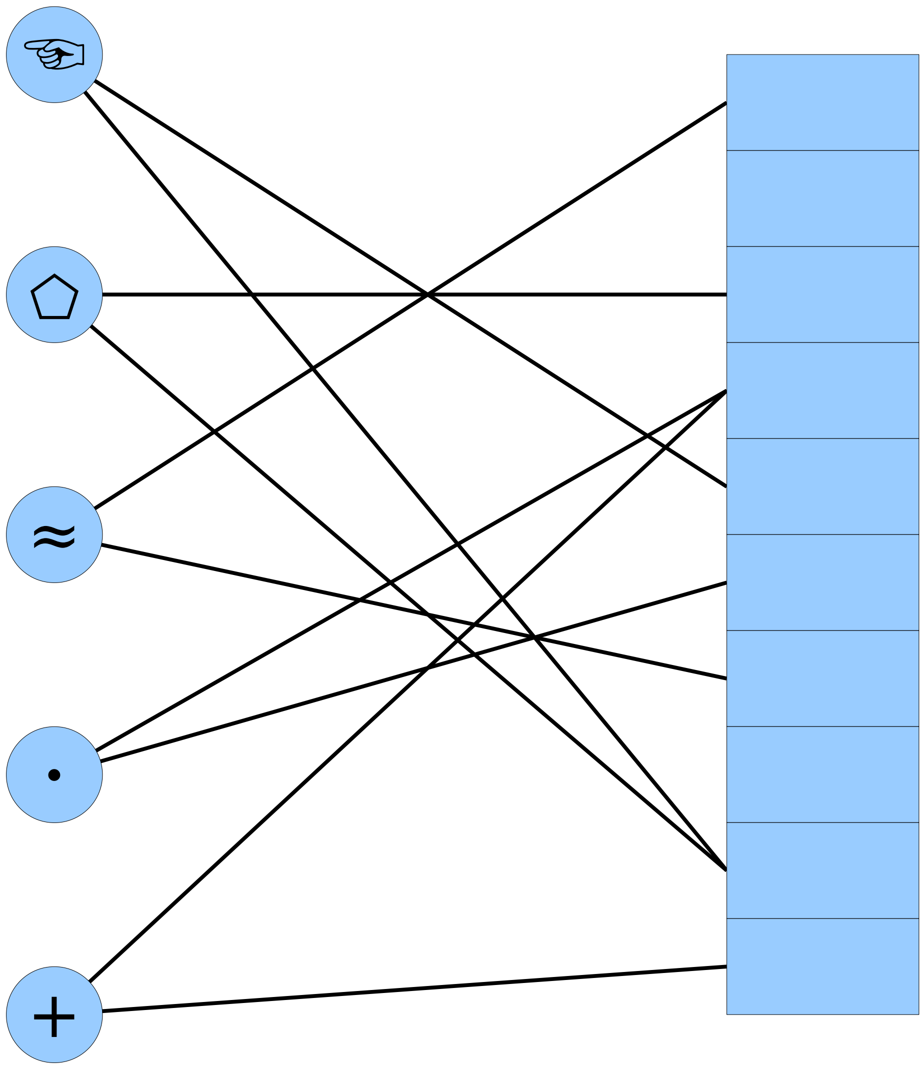
Rethinking Cuckoo Hashing

- Earlier, we used the ***cuckoo graph*** to model cuckoo hashing.
- There's another graph we can use to model cuckoo hashing.
  - Create two groups of nodes: one for items, and one for table slots.
  - Add edges from each item to the table slots that can store it.
- ***Key idea:*** When hashing items to multiple positions, draw this bipartite graph and figure out which of its properties you want to study.



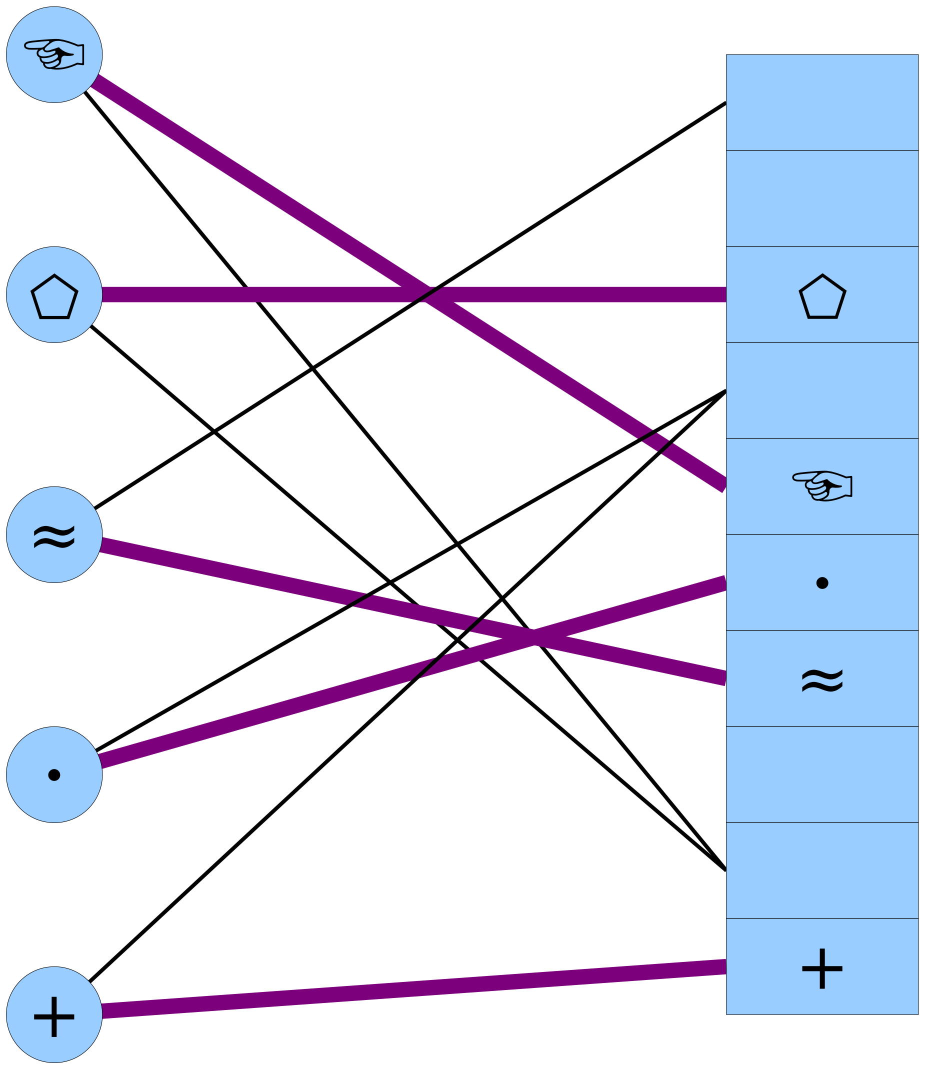
Rethinking Cuckoo Hashing

- Each item needs to go into a table slot.
- ***Goal:*** Choose a group of edges where
  - each node on the left is adjacent to *exactly one* edge, and
  - each node on the right is adjacent to *at most one* chosen edge.
- This models our assignment of items to slots.



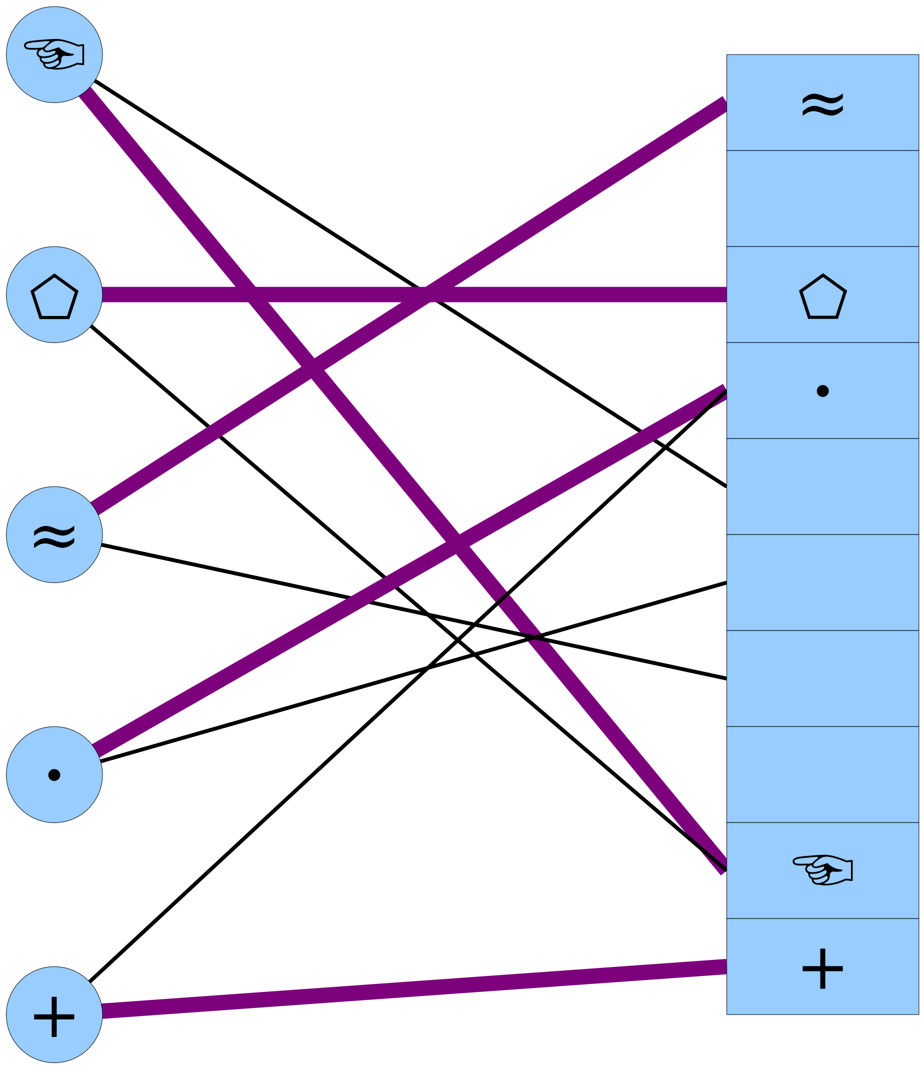
Rethinking Cuckoo Hashing

- Each item needs to go into a table slot.
- ***Goal:*** Choose a group of edges where
  - each node on the left is adjacent to *exactly one* edge, and
  - each node on the right is adjacent to *at most one* chosen edge.
- This models our assignment of items to slots.



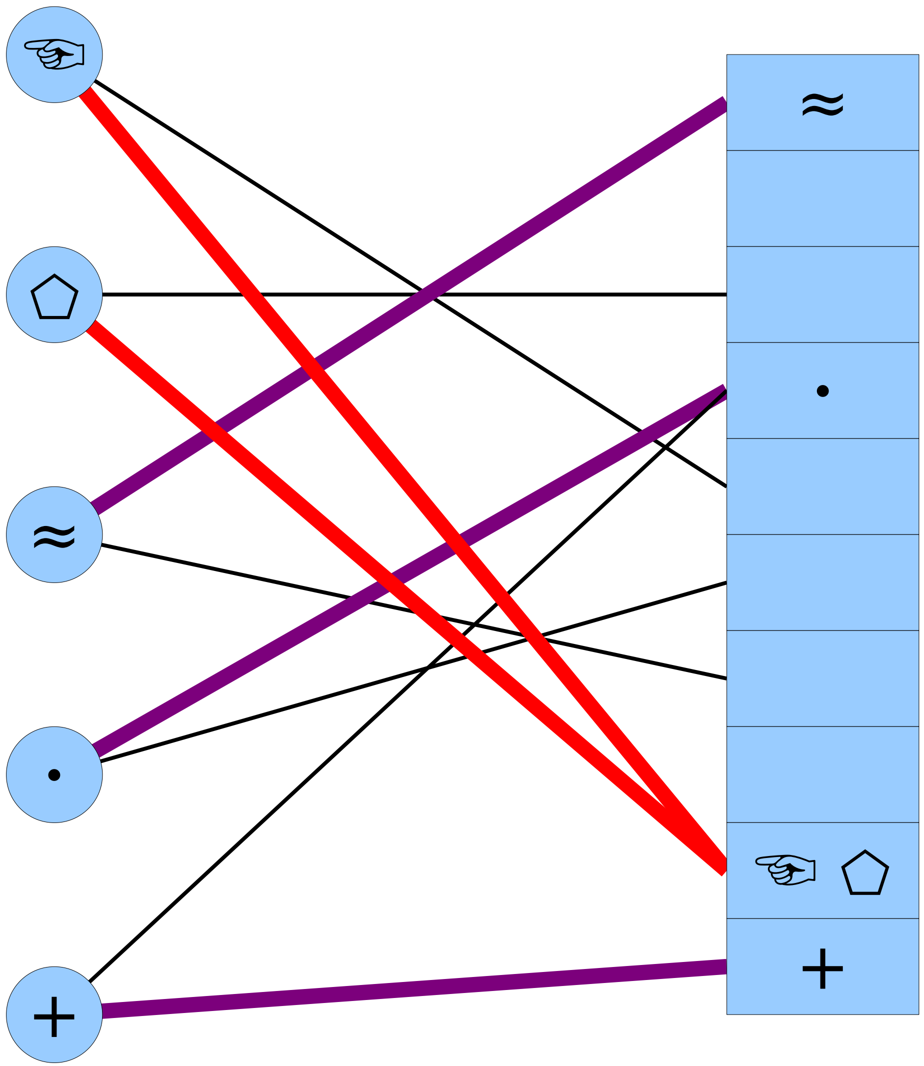
Rethinking Cuckoo Hashing

- Each item needs to go into a table slot.
- ***Goal:*** Choose a group of edges where
  - each node on the left is adjacent to *exactly one* edge, and
  - each node on the right is adjacent to *at most one* chosen edge.
- This models our assignment of items to slots.



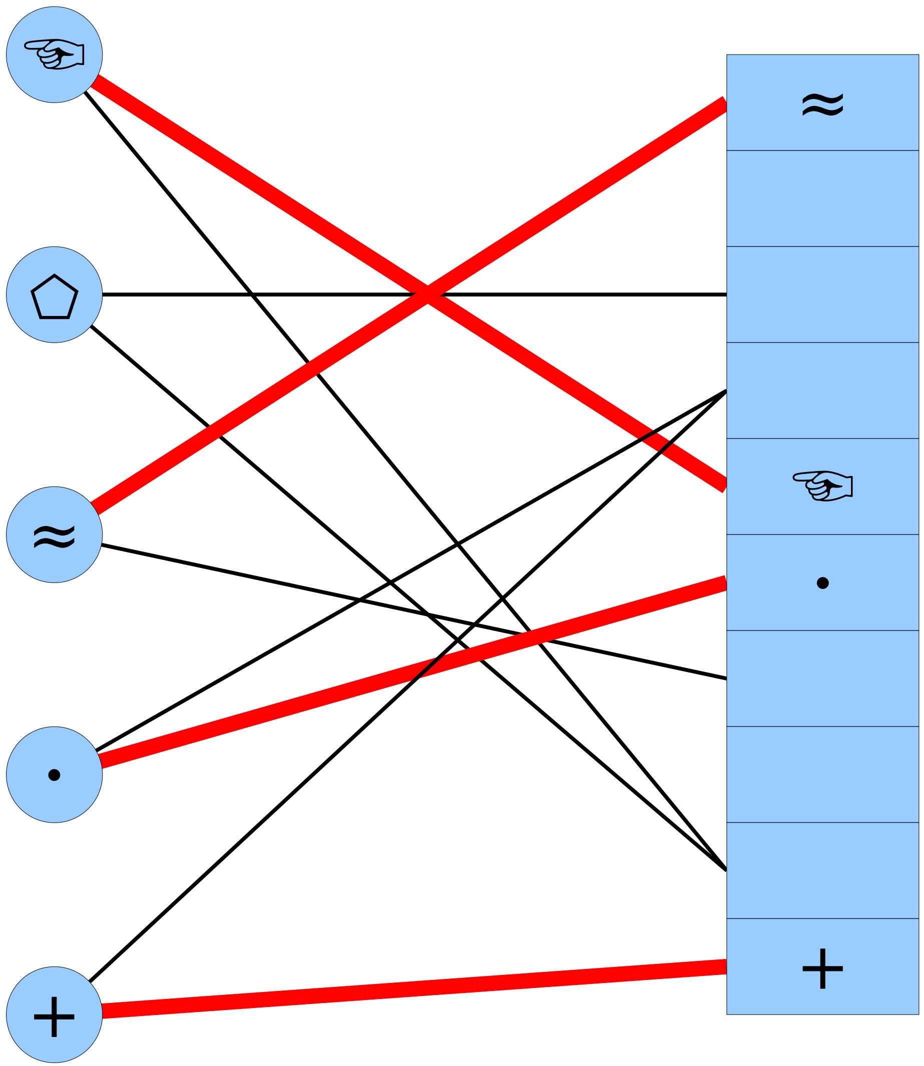
Rethinking Cuckoo Hashing

- Each item needs to go into a table slot.
- ***Goal:*** Choose a group of edges where
  - each node on the left is adjacent to *exactly one* edge, and
  - each node on the right is adjacent to *at most one* chosen edge.
- This models our assignment of items to slots.



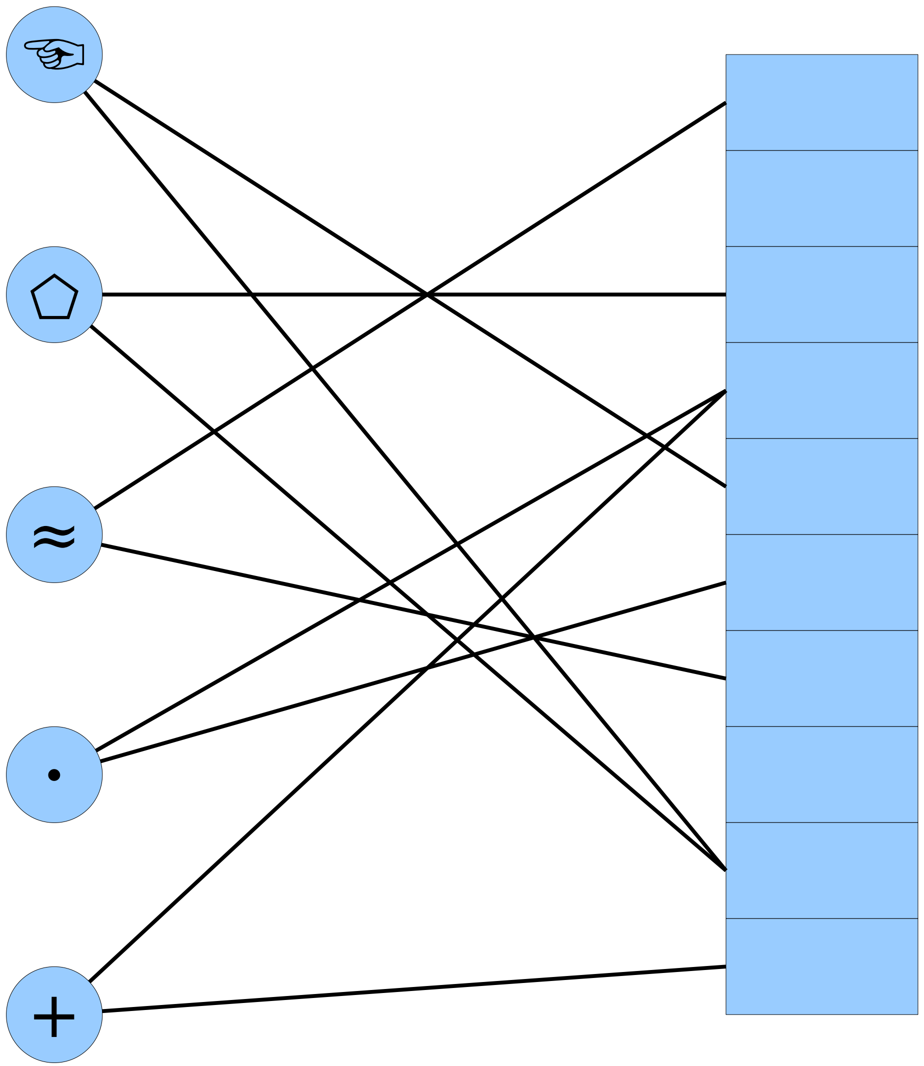
Rethinking Cuckoo Hashing

- Each item needs to go into a table slot.
- ***Goal:*** Choose a group of edges where
  - each node on the left is adjacent to *exactly one* edge, and
  - each node on the right is adjacent to *at most one* chosen edge.
- This models our assignment of items to slots.



Rethinking Cuckoo Hashing

- We're now left with these constraints:
  - Each item has two outgoing edges.
  - Each item must have one of its outgoing edges selected.
  - Each slot must have at most one of its incoming edges selected.
- We have to relax at least one of these constraints if we want to push past the 50% space utilization limit.

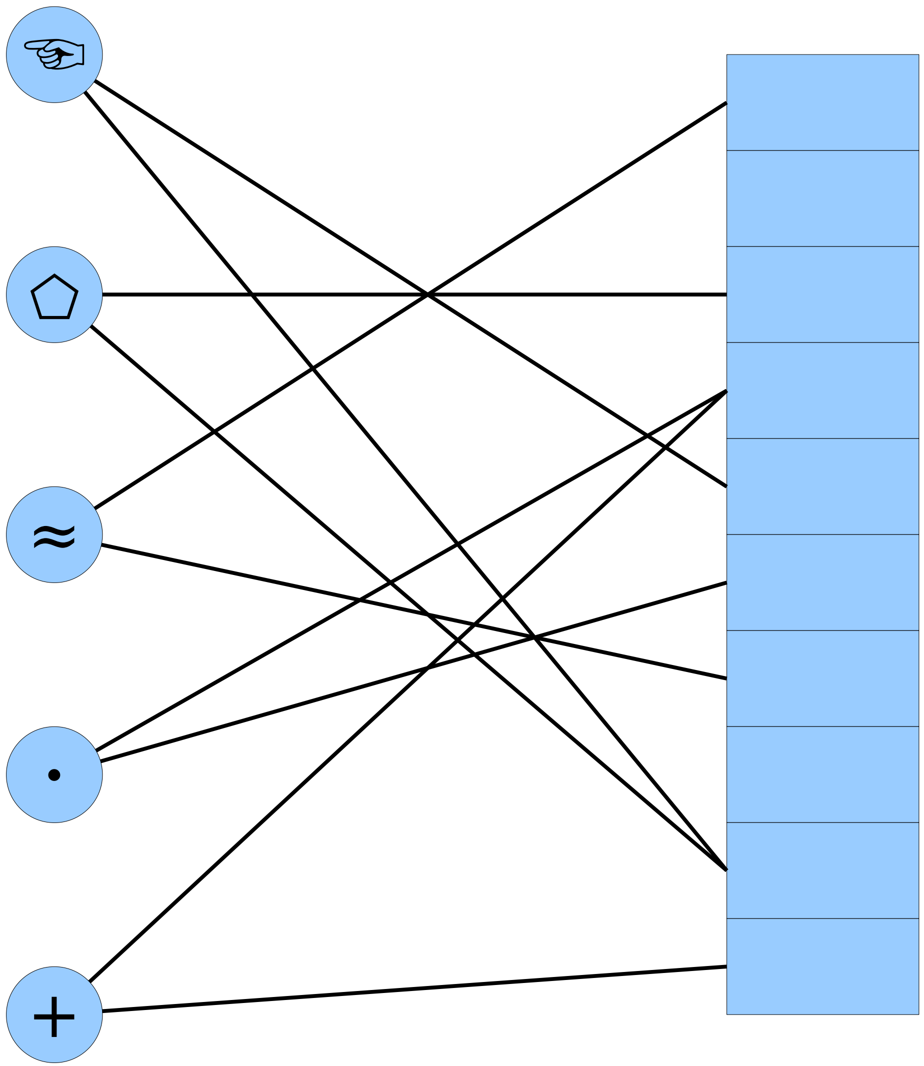


Which of these constraints could we relax, and what might it look like if we did?

Formulate a hypothesis, but ***don't post anything in chat just yet***.

Rethinking Cuckoo Hashing

- We're now left with these constraints:
  - Each item has two outgoing edges.
  - Each item must have one of its outgoing edges selected.
  - Each slot must have at most one of its incoming edges selected.
- We have to relax at least one of these constraints if we want to push past the 50% space utilization limit.

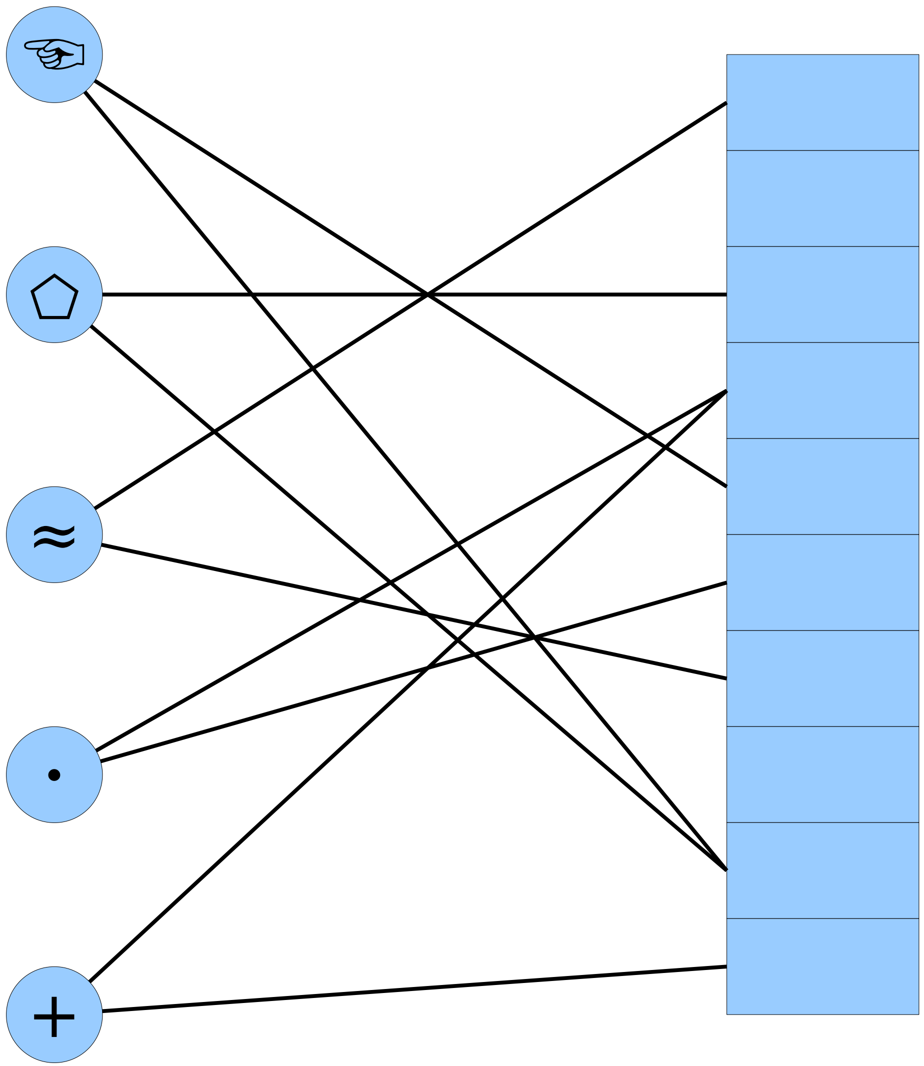


Which of these constraints could we relax, and what might it look like if we did?

Now, ***private chat me your ideas***. Not sure? Just answer "??".

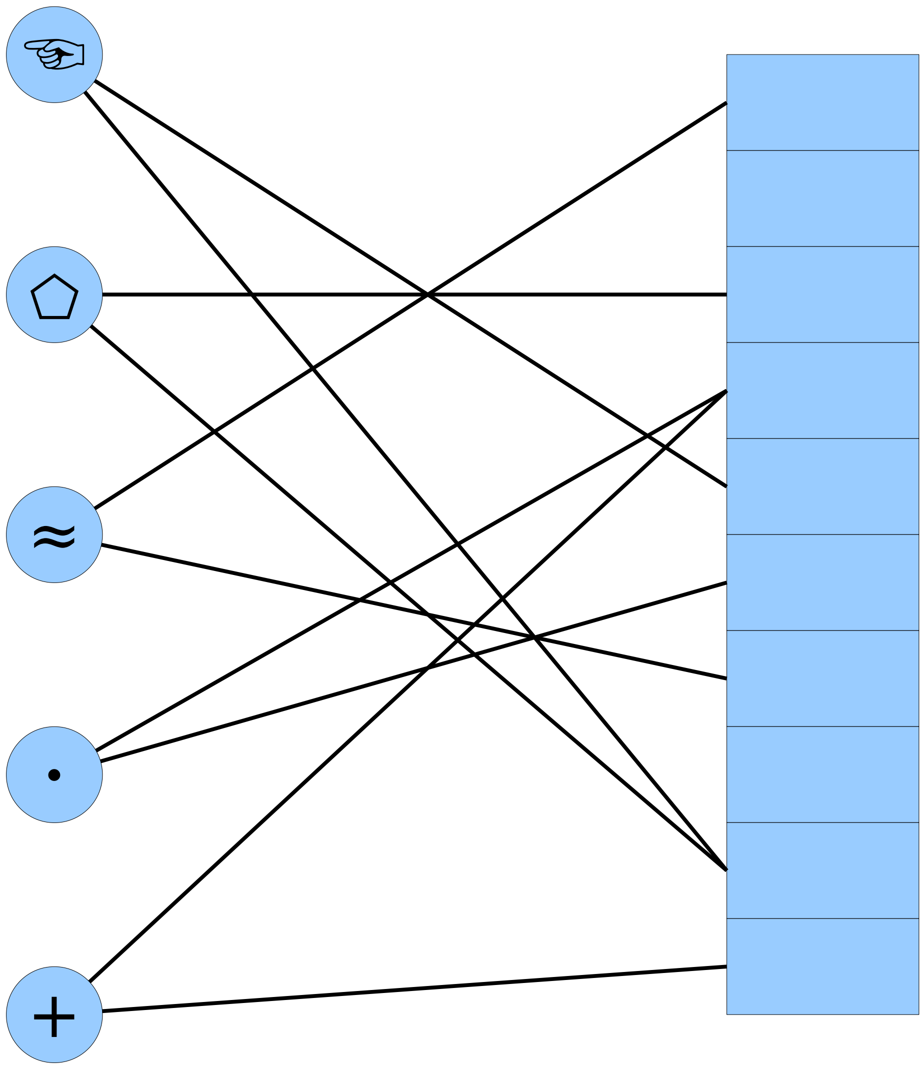
Rethinking Cuckoo Hashing

- We're now left with these constraints:
  - Each item has two outgoing edges.
  - Each item must have one of its outgoing edges selected.
  - Each slot must have at most one of its incoming edges selected.
- We have to relax at least one of these constraints if we want to push past the 50% space utilization limit.



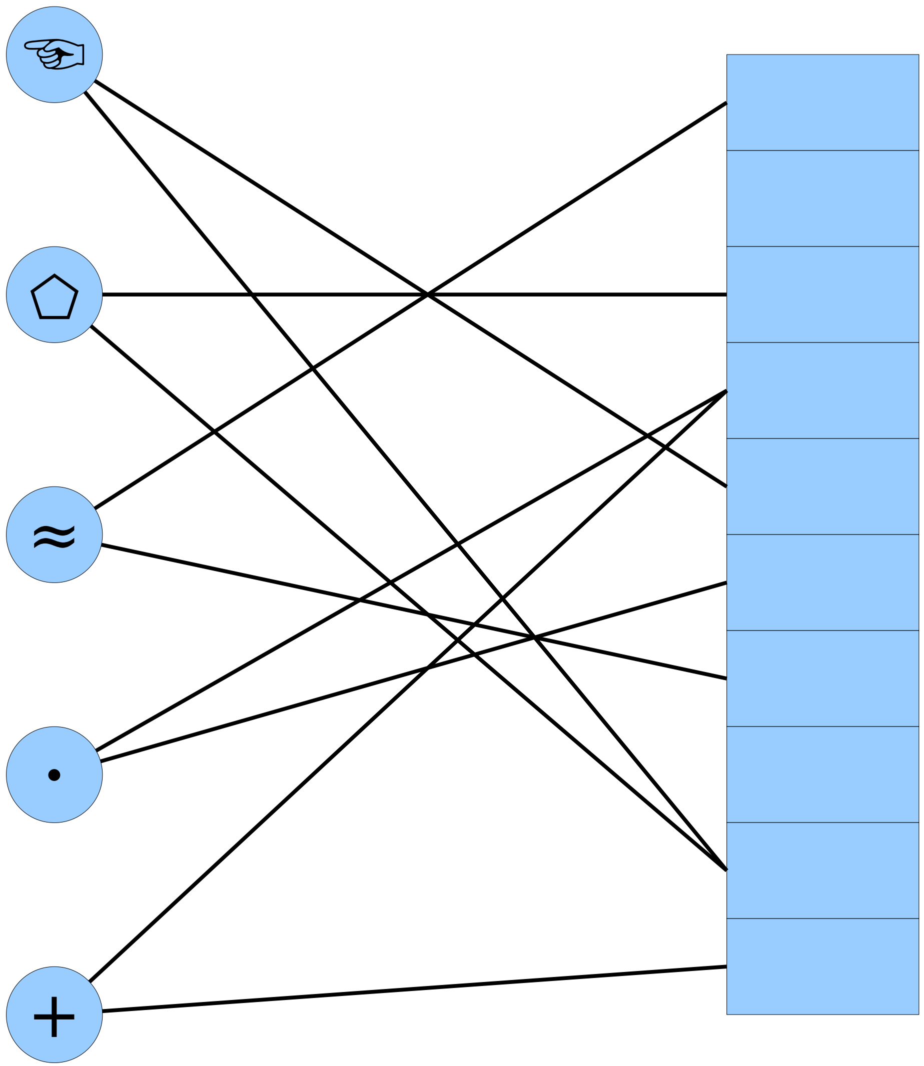
Rethinking Cuckoo Hashing

- Each item has two outgoing edges.
- Each item must have one of its outgoing edges selected.
- Each slot must have at most one of its incoming edges selected.



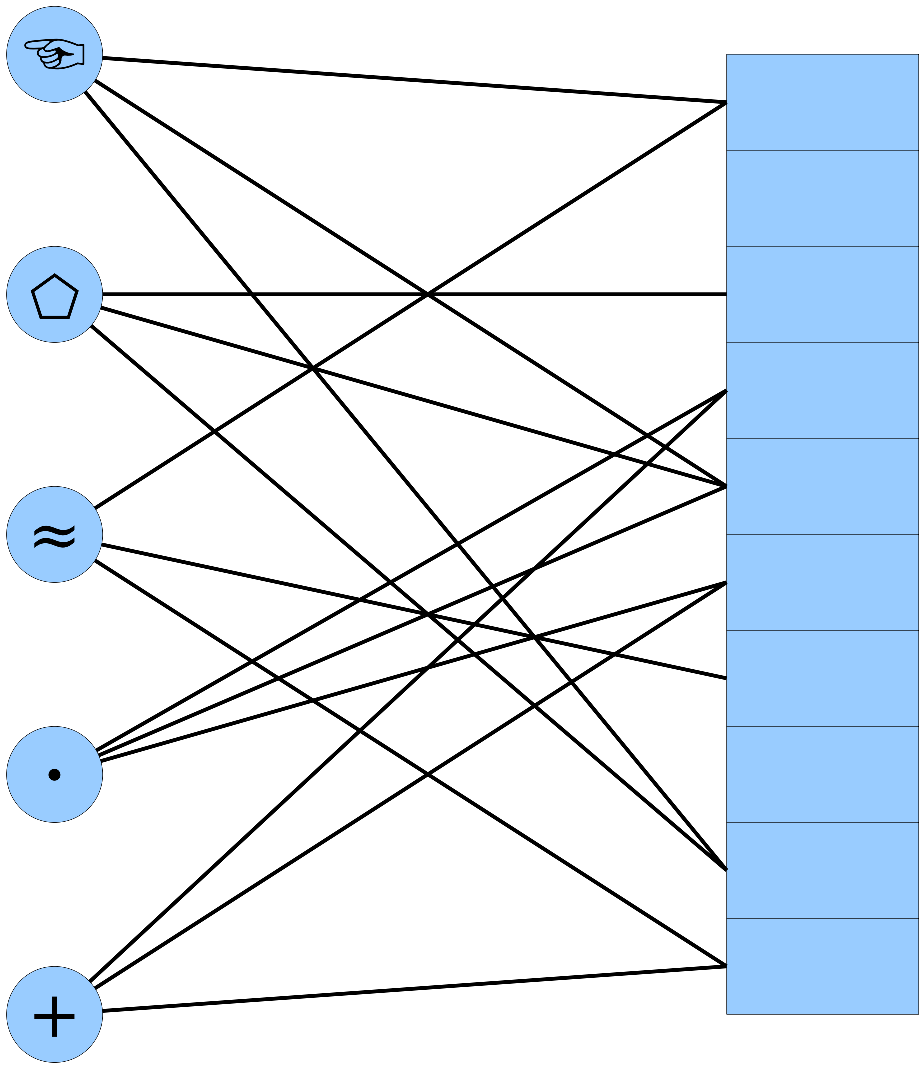
Rethinking Cuckoo Hashing

- Each item has two outgoing edges.
- Each item must have one of its outgoing edges selected.
- Each slot must have at most one of its incoming edges selected.



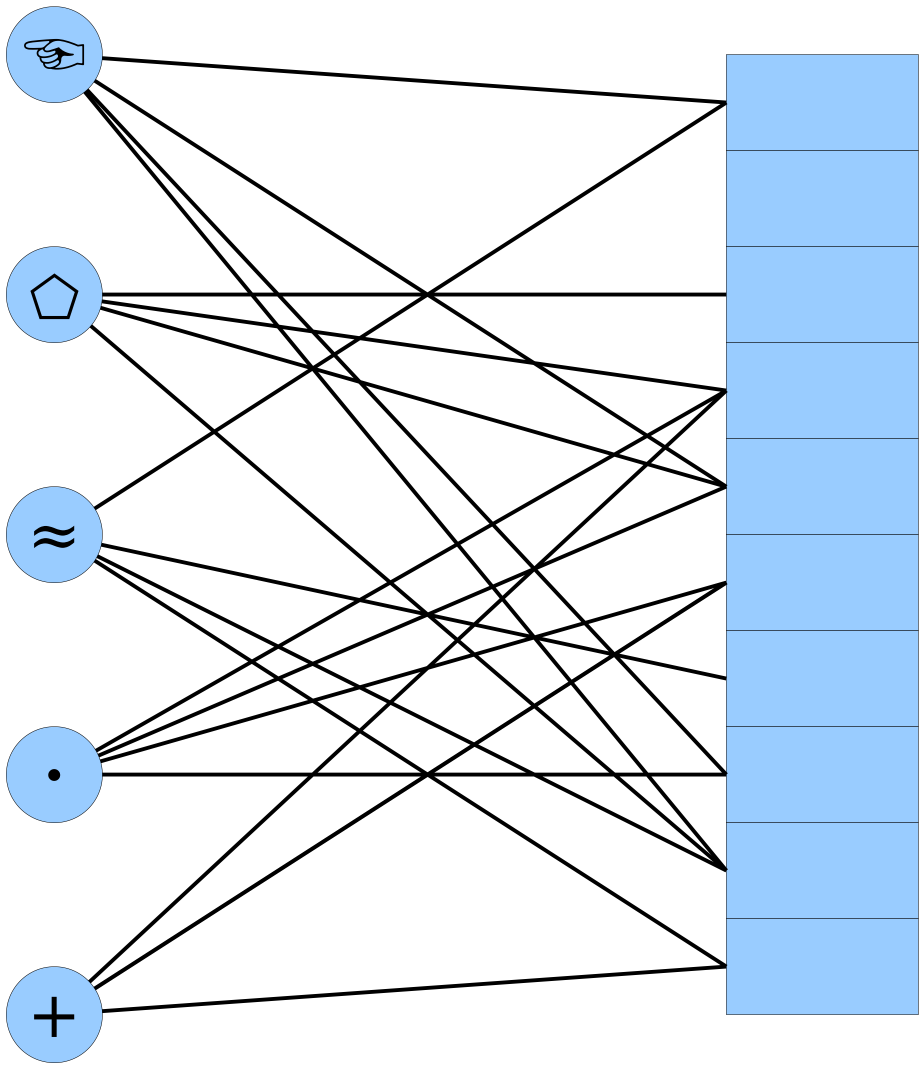
Rethinking Cuckoo Hashing

- Each item has two outgoing edges.
- Each item must have one of its outgoing edges selected.
- Each slot must have at most one of its incoming edges selected.



Rethinking Cuckoo Hashing

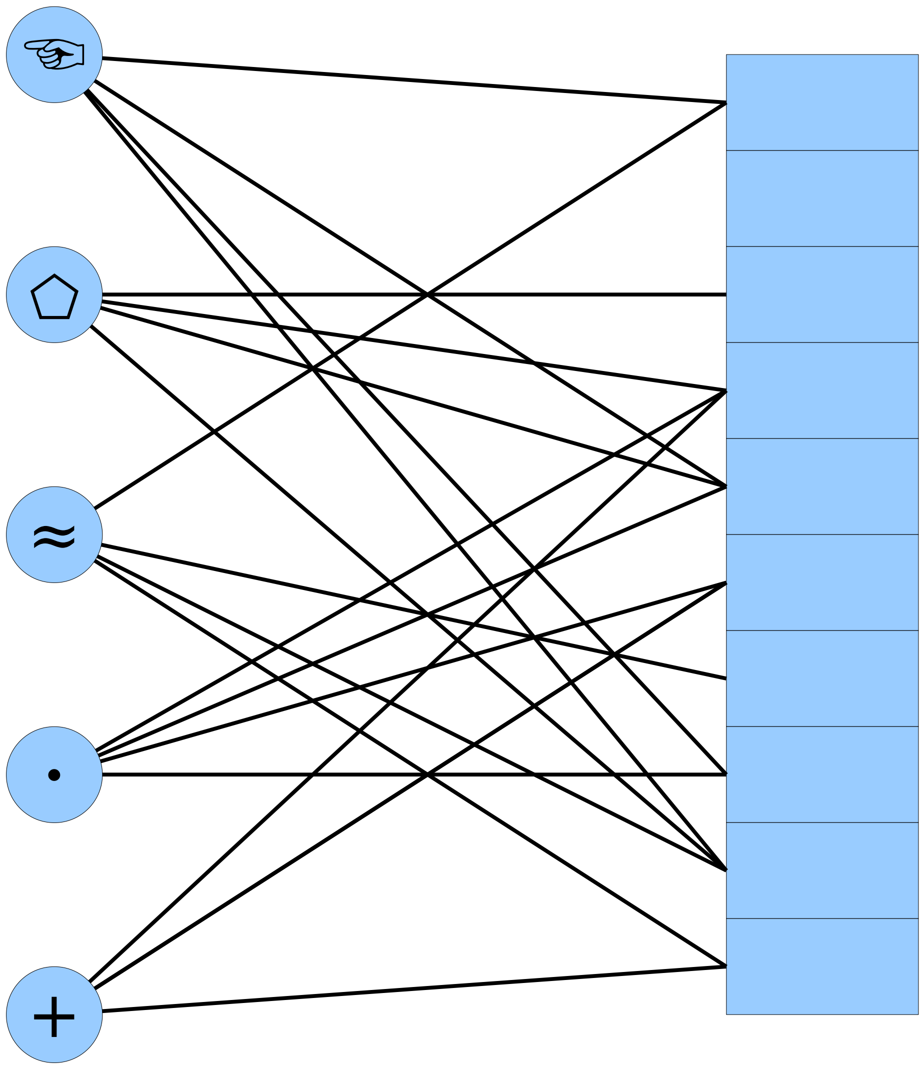
- Each item has two outgoing edges.
- Each item must have one of its outgoing edges selected.
- Each slot must have at most one of its incoming edges selected.



Rethinking Cuckoo Hashing

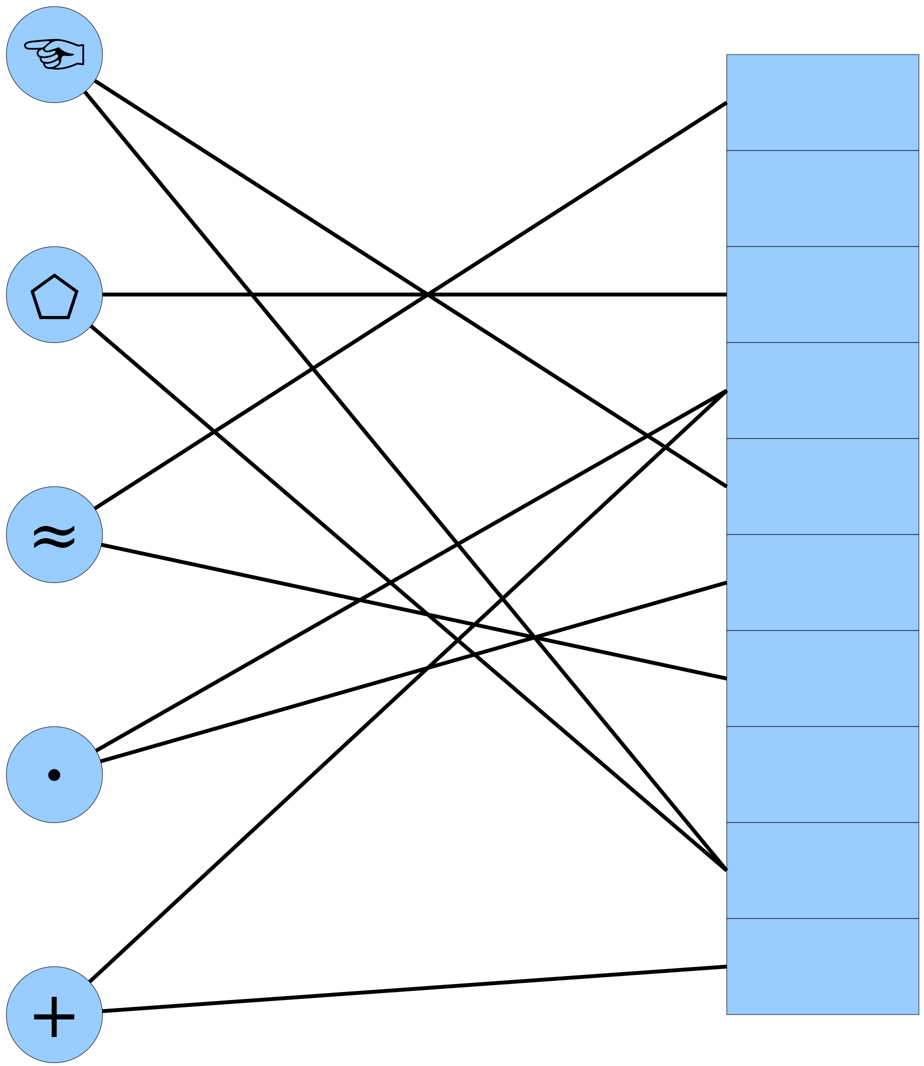
- Each item has two outgoing edges.
- Each item must have one of its outgoing edges selected.
- Each slot must have at most one of its incoming edges selected.

Idea 1: Use more hash functions to give each item more places to land.



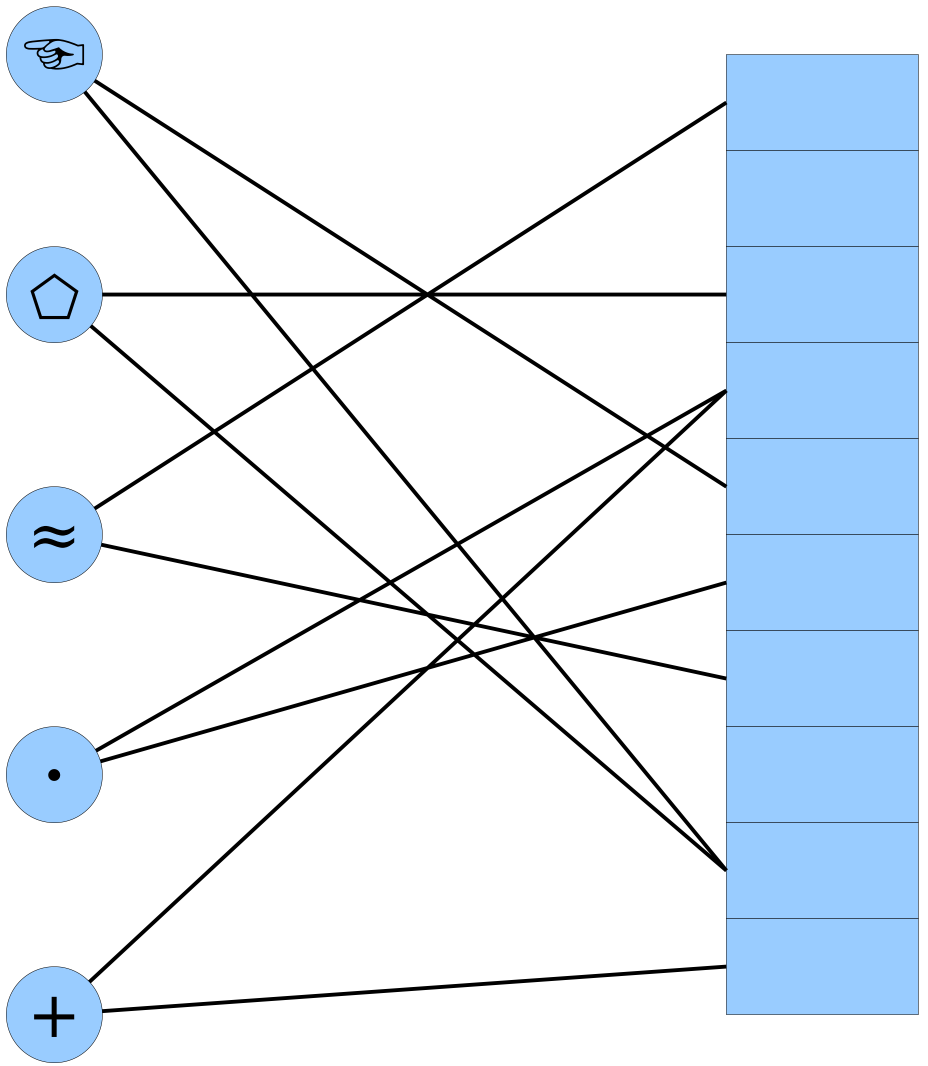
Rethinking Cuckoo Hashing

- Each item has two outgoing edges.
- Each item must have one of its outgoing edges selected.
- Each slot must have at most one of its incoming edges selected.



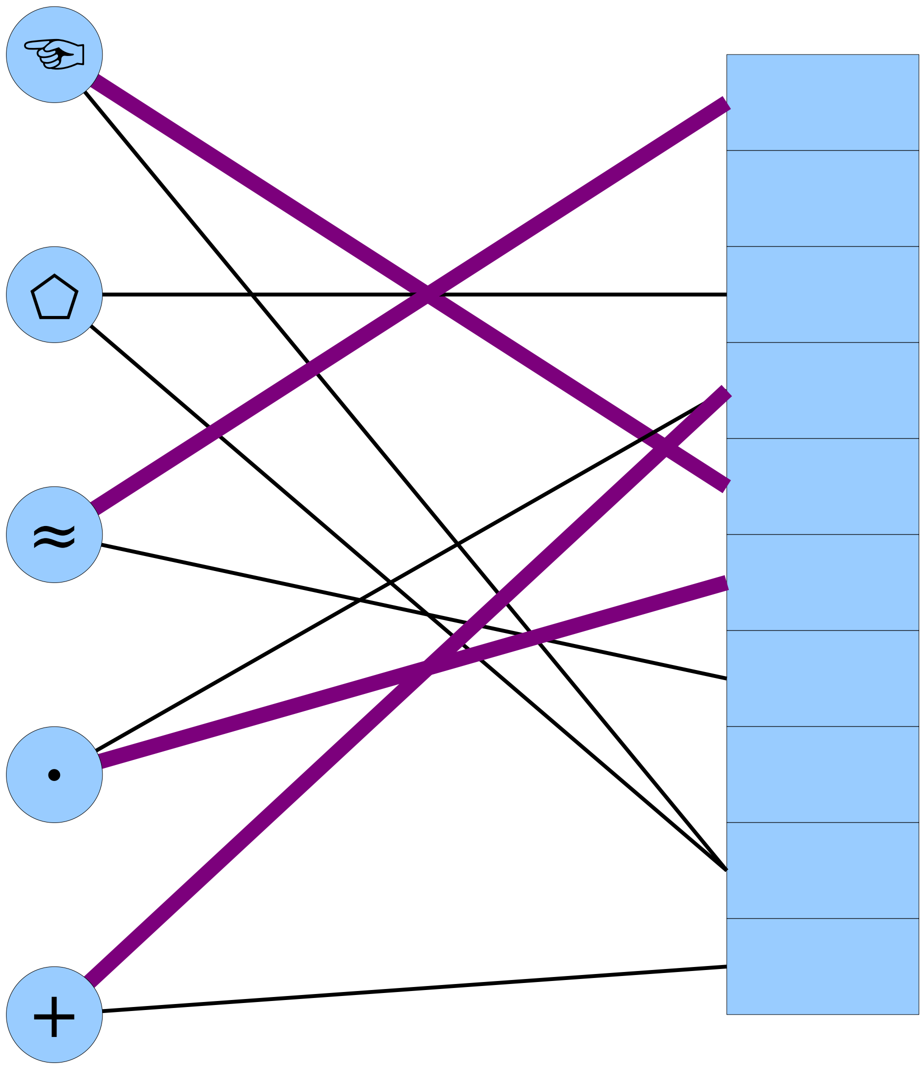
Rethinking Cuckoo Hashing

- Each item has two outgoing edges.
- Each item must have one of its outgoing edges selected.
- Each slot must have at most one of its incoming edges selected.



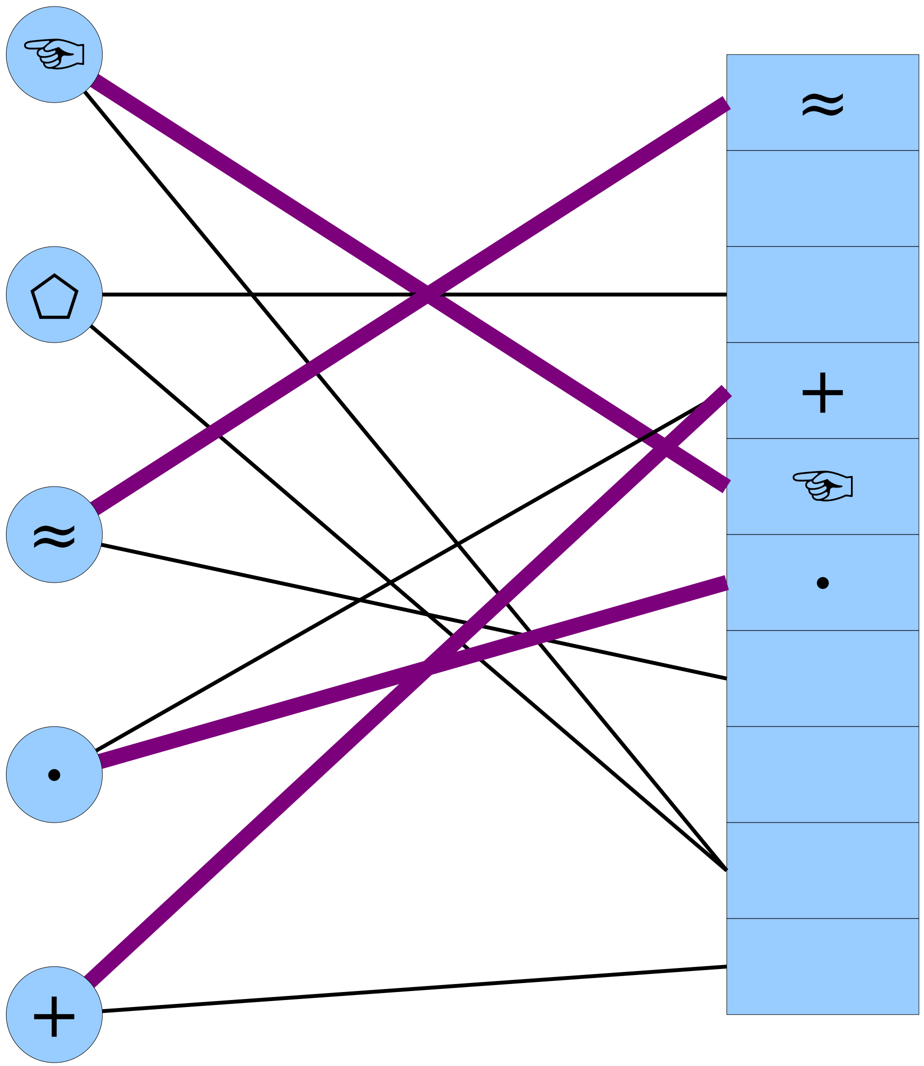
Rethinking Cuckoo Hashing

- Each item has two outgoing edges.
- Each item must have one of its outgoing edges selected.
- Each slot must have at most one of its incoming edges selected.



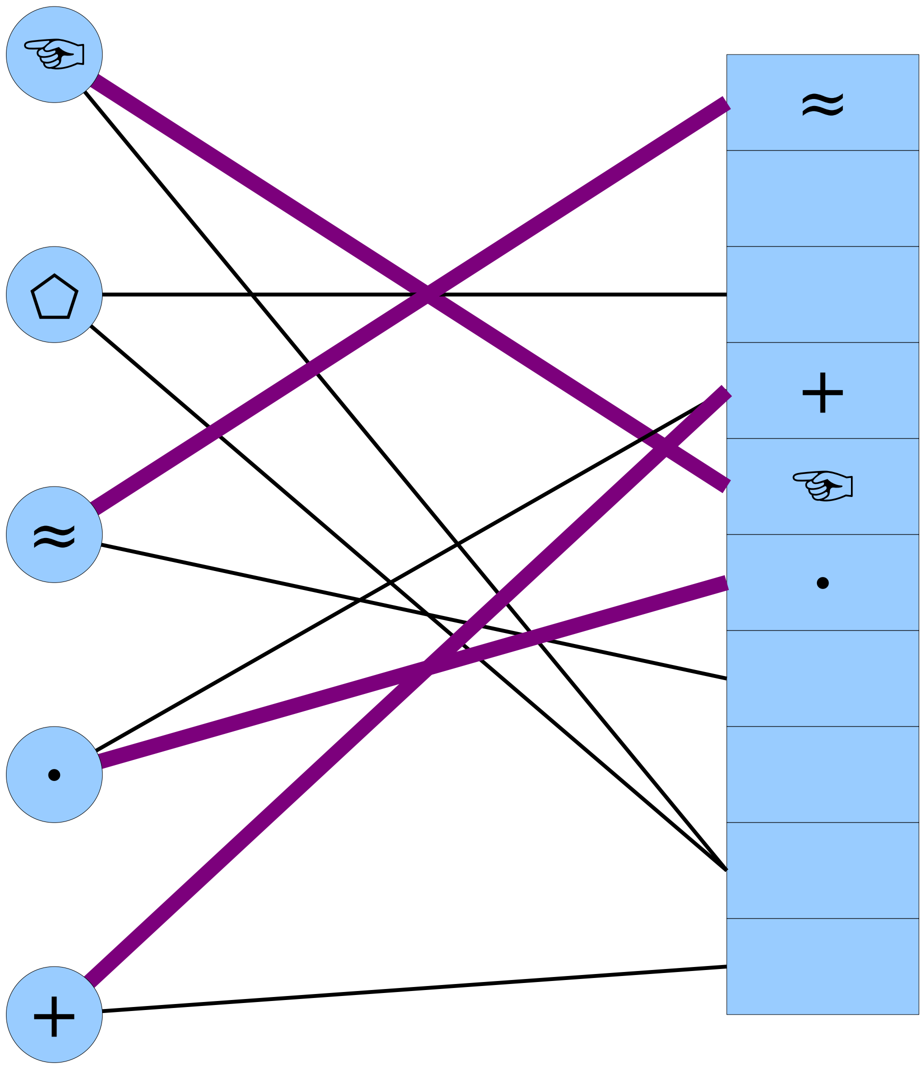
Rethinking Cuckoo Hashing

- Each item has two outgoing edges.
- Each item must have one of its outgoing edges selected.
- Each slot must have at most one of its incoming edges selected.



Rethinking Cuckoo Hashing

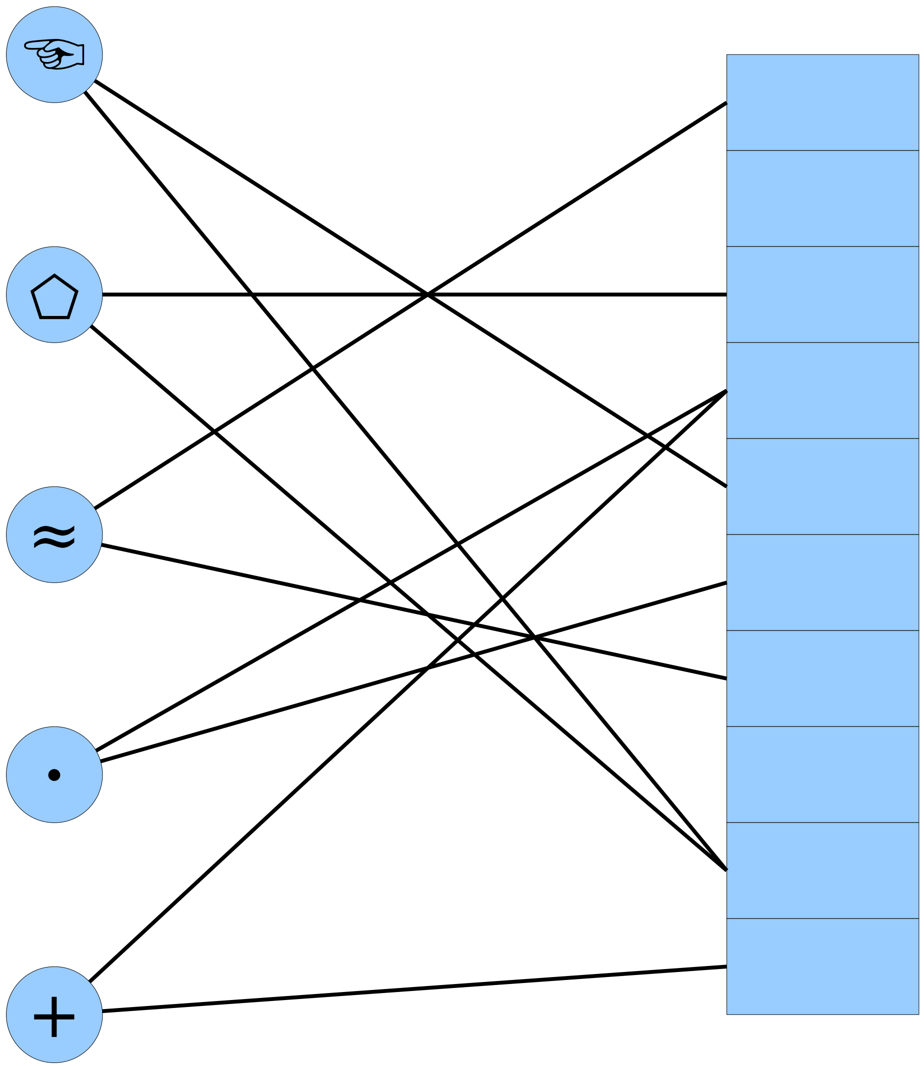
- Each item has two outgoing edges.
- Each item must have one of its outgoing edges selected.
- Each slot must have at most one of its incoming edges selected.



Idea 2: Don't put all items in the table. Let some elements live in another data structure.

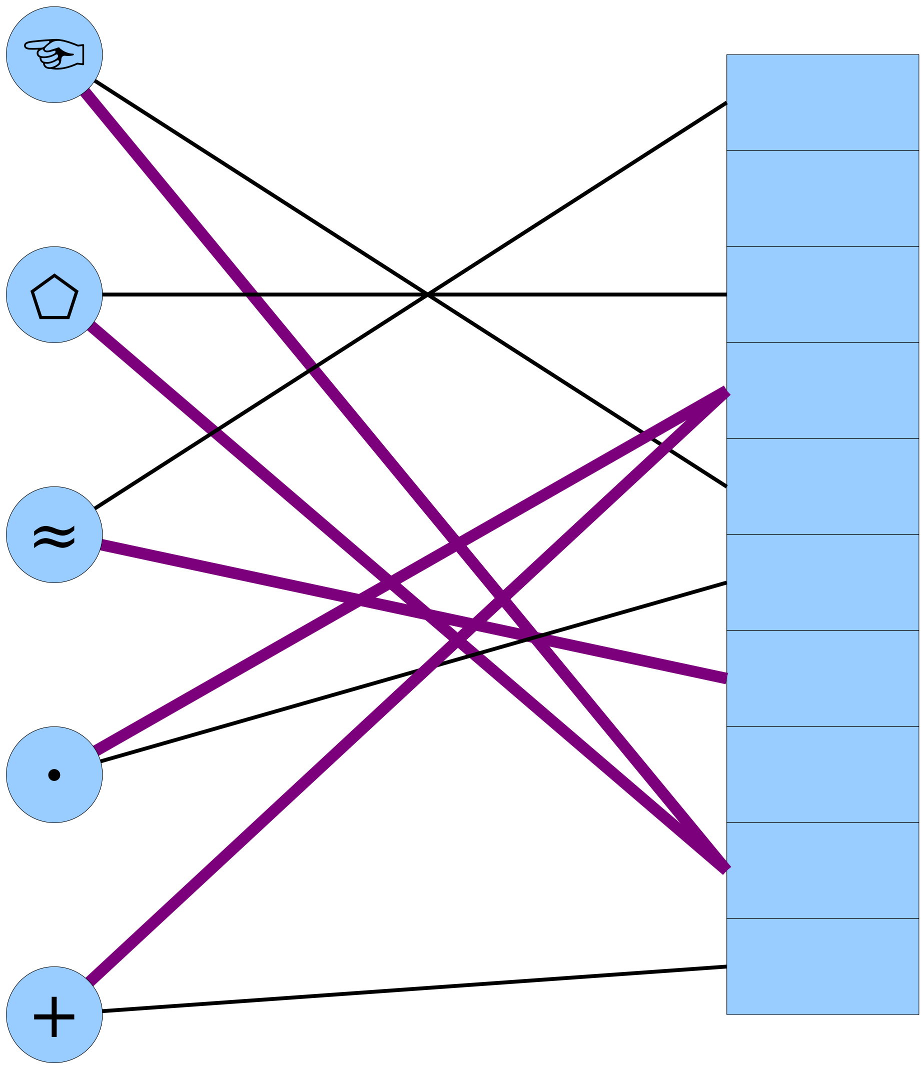
Rethinking Cuckoo Hashing

- Each item has two outgoing edges.
- Each item must have one of its outgoing edges selected.
- Each slot must have at most one of its incoming edges selected.



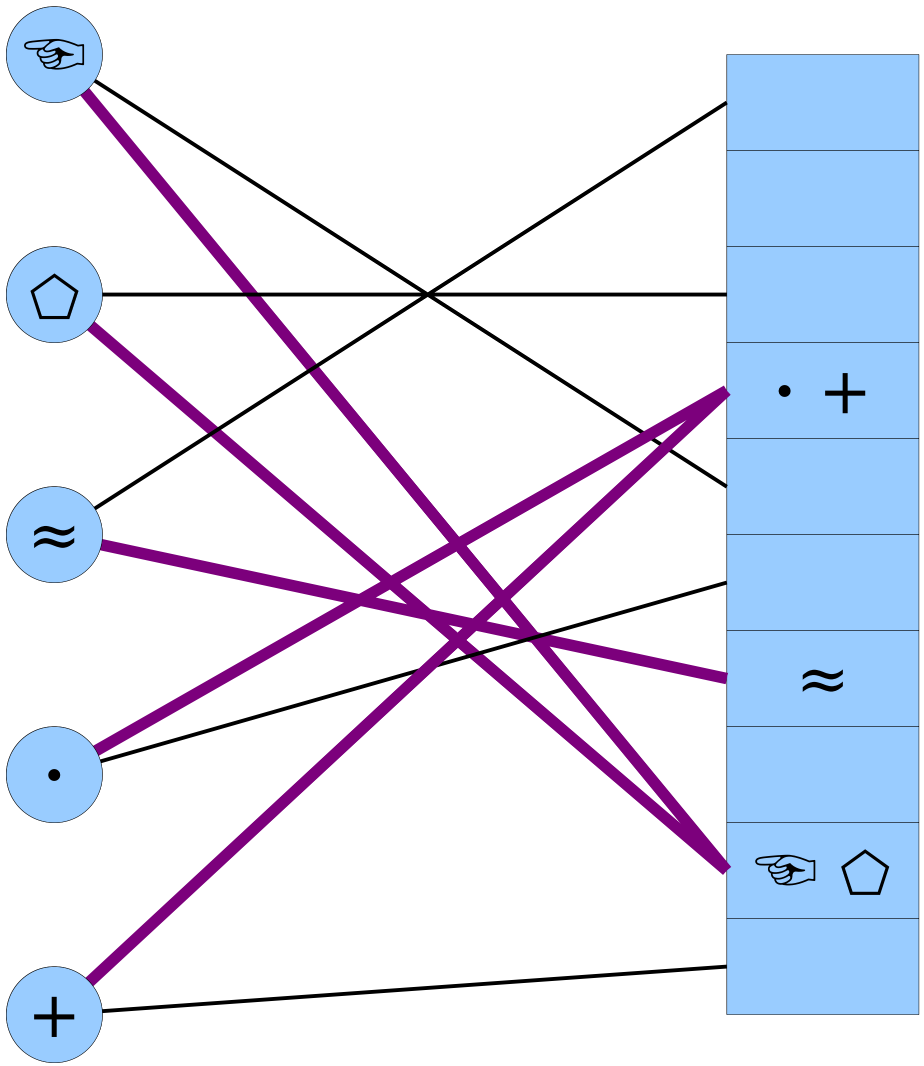
Rethinking Cuckoo Hashing

- Each item has two outgoing edges.
- Each item must have one of its outgoing edges selected.
- Each slot must have at most one of its incoming edges selected.



Rethinking Cuckoo Hashing

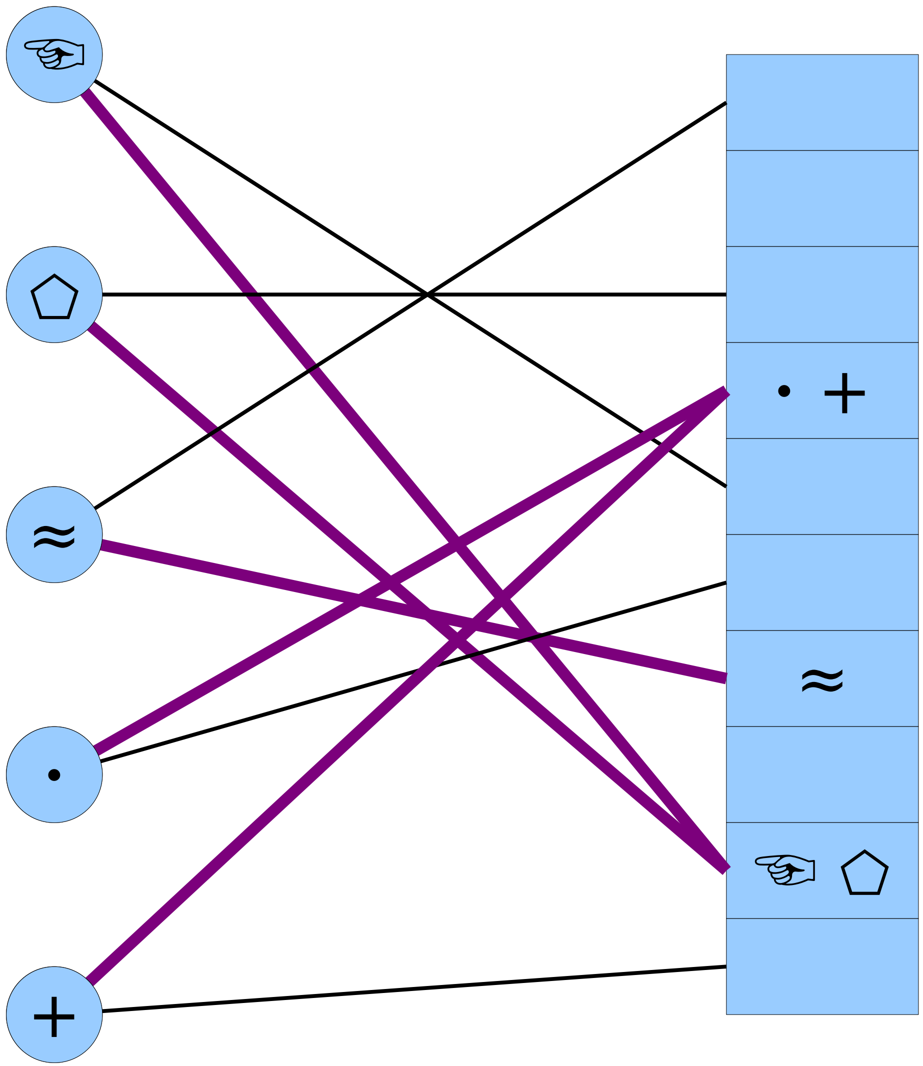
- Each item has two outgoing edges.
- Each item must have one of its outgoing edges selected.
- Each slot must have at most one of its incoming edges selected.



Rethinking Cuckoo Hashing

- Each item has two outgoing edges.
- Each item must have one of its outgoing edges selected.
- Each slot must have at most one of its incoming edges selected.

Idea 3: Allow multiple items to be stored in each slot.



Cuckoo Hashing Revisited

- We now have three ideas for improving our basic cuckoo hash table.
  - ***Idea 1:*** Use multiple hash functions to give items more wiggle room about where to go.
  - ***Idea 2:*** Don't place all items in the table; let some of them go somewhere else.
  - ***Idea 3:*** Allow for multiple items to be placed in each slot.
- Each of these ideas has been explored. We'll do a quick survey of what these options look like.

Cuckoo Hashing Revisited

- We now have three ideas for improving our basic cuckoo hash table.
  - ***Idea 1:*** Use multiple hash functions to give items more wiggle room about where to go.
  - ***Idea 2:*** Don’t place all items in the table; let some of them go somewhere else.
  - ***Idea 3:*** Allow for multiple items to be placed in each slot.
- Each of these ideas has been explored. We’ll do a quick survey of what these options look like.

d-ary Cuckoo Hashing

- In *d*-ary cuckoo hashing, we pick an integer $d \geq 2$ and choose *d* different hash functions.
- Each item can be stored in one up to *d* slots, with choices given by the hash functions.
  - You could do extra work to ensure there are *d* separate locations, or be okay with duplicates if the hashes collide.
- To check if an item is in the table, hash it *d* times and see if it's in any of those slots.

30		51	62		87	73	15		11

d-ary Cuckoo Hashing

- In d-ary cuckoo hashing, we pick an integer $d \geq 2$ and choose d different hash functions.
- Each item can be stored in one up to d slots, with choices given by the hash functions.
  - You could do extra work to ensure there are d separate locations, or be okay with duplicates if the hashes collide.
- To check if an item is in the table, hash it d times and see if it's in any of those slots.

62

30		51	62		87	73	15		11

d-ary Cuckoo Hashing

- In d-ary cuckoo hashing, we pick an integer $d \geq 2$ and choose d different hash functions.
- Each item can be stored in one up to d slots, with choices given by the hash functions.
  - You could do extra work to ensure there are d separate locations, or be okay with duplicates if the hashes collide.
- To check if an item is in the table, hash it d times and see if it's in any of those slots.

62

30		51	62		87	73	15		11

d-ary Cuckoo Hashing

- In *d*-ary cuckoo hashing, we pick an integer $d \geq 2$ and choose *d* different hash functions.
- Each item can be stored in one up to *d* slots, with choices given by the hash functions.
  - You could do extra work to ensure there are *d* separate locations, or be okay with duplicates if the hashes collide.
- To check if an item is in the table, hash it *d* times and see if it's in any of those slots.

73

30		51	62		87	73	15		11

d-ary Cuckoo Hashing

- In d-ary cuckoo hashing, we pick an integer $d \geq 2$ and choose d different hash functions.
- Each item can be stored in one up to d slots, with choices given by the hash functions.
  - You could do extra work to ensure there are d separate locations, or be okay with duplicates if the hashes collide.
- To check if an item is in the table, hash it d times and see if it's in any of those slots.

73

30		51	62		87	73	15		11

d-ary Cuckoo Hashing

- In d-ary cuckoo hashing, we pick an integer $d \geq 2$ and choose d different hash functions.
- Each item can be stored in one up to d slots, with choices given by the hash functions.
  - You could do extra work to ensure there are d separate locations, or be okay with duplicates if the hashes collide.
- To check if an item is in the table, hash it d times and see if it's in any of those slots.

46

30		51	62		87	73	15		11

d-ary Cuckoo Hashing

- In d-ary cuckoo hashing, we pick an integer $d \geq 2$ and choose d different hash functions.
- Each item can be stored in one up to d slots, with choices given by the hash functions.
  - You could do extra work to ensure there are d separate locations, or be okay with duplicates if the hashes collide.
- To check if an item is in the table, hash it d times and see if it's in any of those slots.

46

30		51	62		87	73	15		11

d-ary Cuckoo Hashing

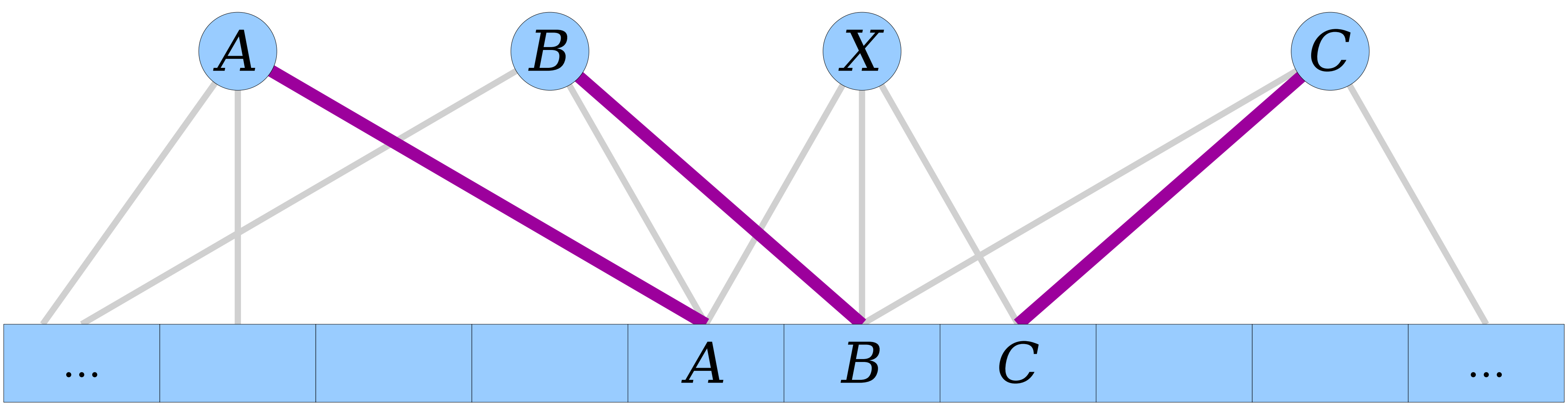
- As with regular cuckoo hashing, when inserting an item, if any of its slots are free, just place the item there.
- If not, we have to displace an existing item.
- ***Question:*** How do you decide which item to pick?

46

30		51	62		87	73	15		11

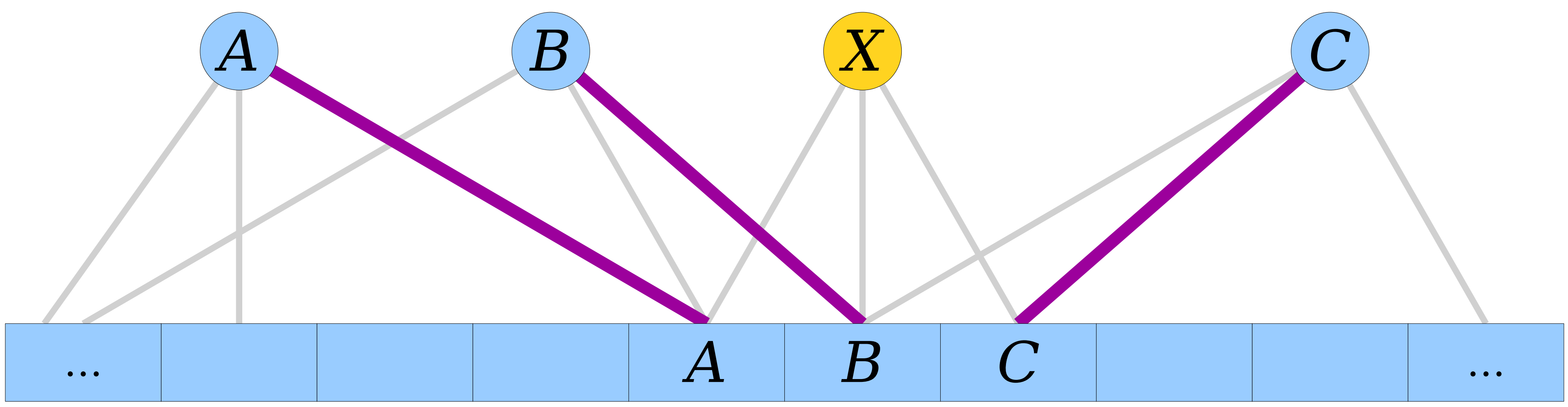
d-ary Cuckoo Hashing

- Here's our bipartite graph view of our table. Edges in ***purple*** represent where items were placed. Edges in ***gray*** represent unused alternates.
- Let's insert X into this table.



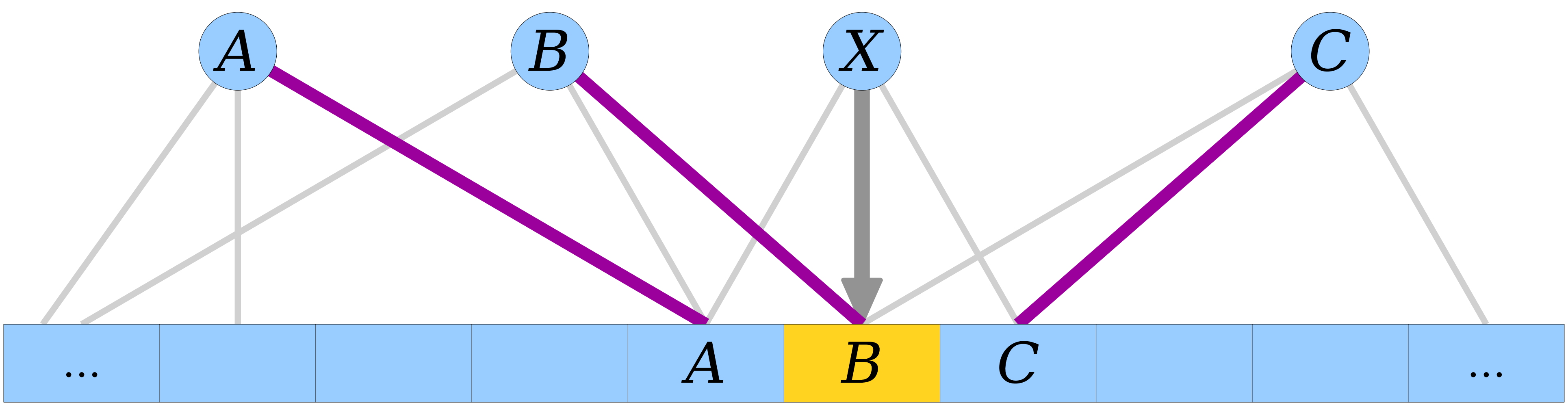
d-ary Cuckoo Hashing

- Here's our bipartite graph view of our table. Edges in ***purple*** represent where items were placed. Edges in ***gray*** represent unused alternates.
- Let's insert *X* into this table.



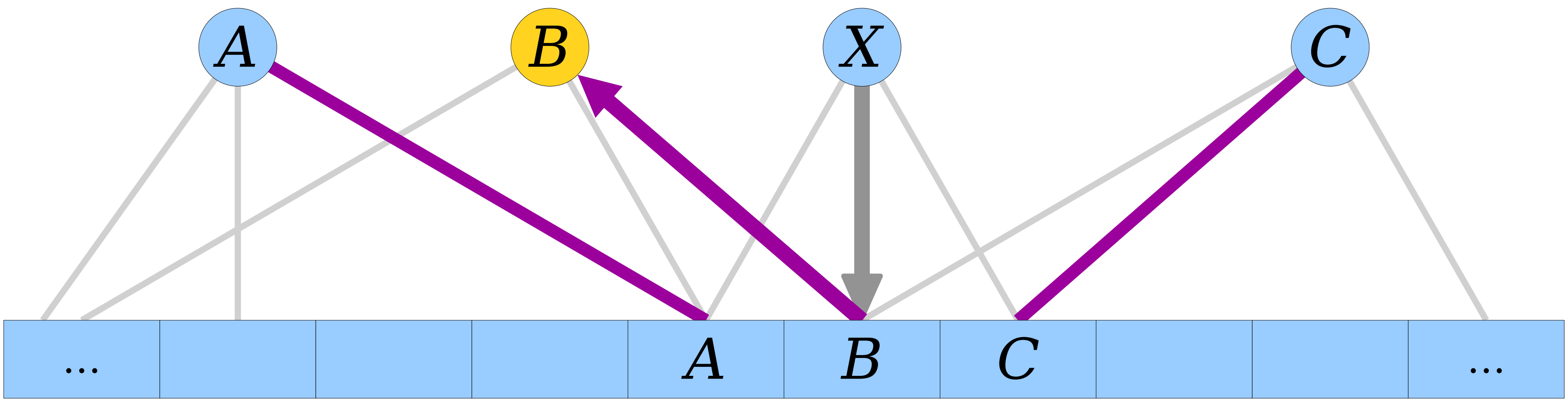
d-ary Cuckoo Hashing

- Here's our bipartite graph view of our table. Edges in ***purple*** represent where items were placed. Edges in ***gray*** represent unused alternates.
- Let's insert X into this table.



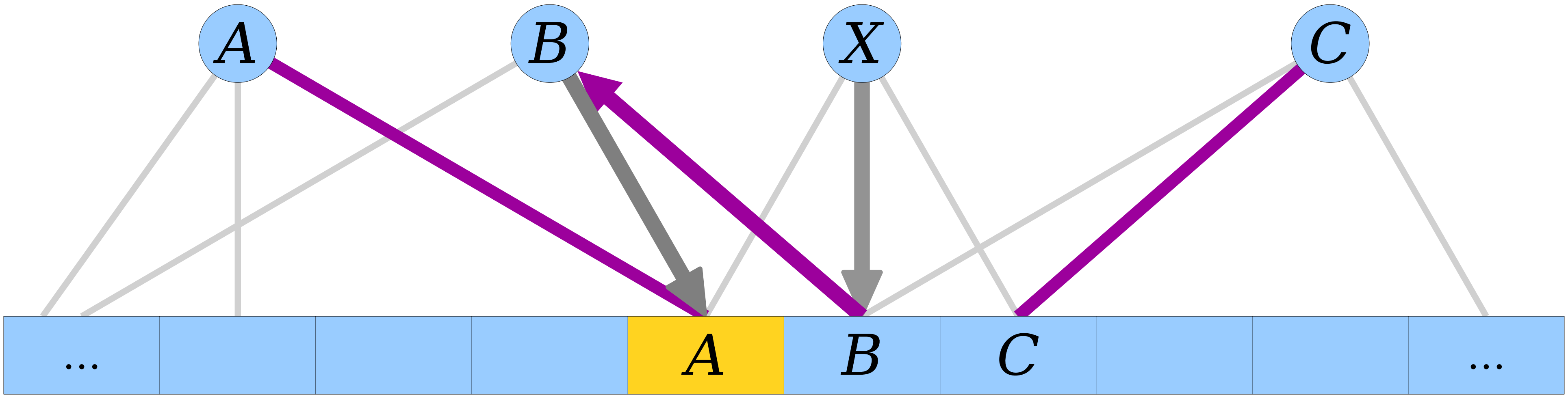
d-ary Cuckoo Hashing

- Here's our bipartite graph view of our table. Edges in ***purple*** represent where items were placed. Edges in ***gray*** represent unused alternates.
- Let's insert X into this table.



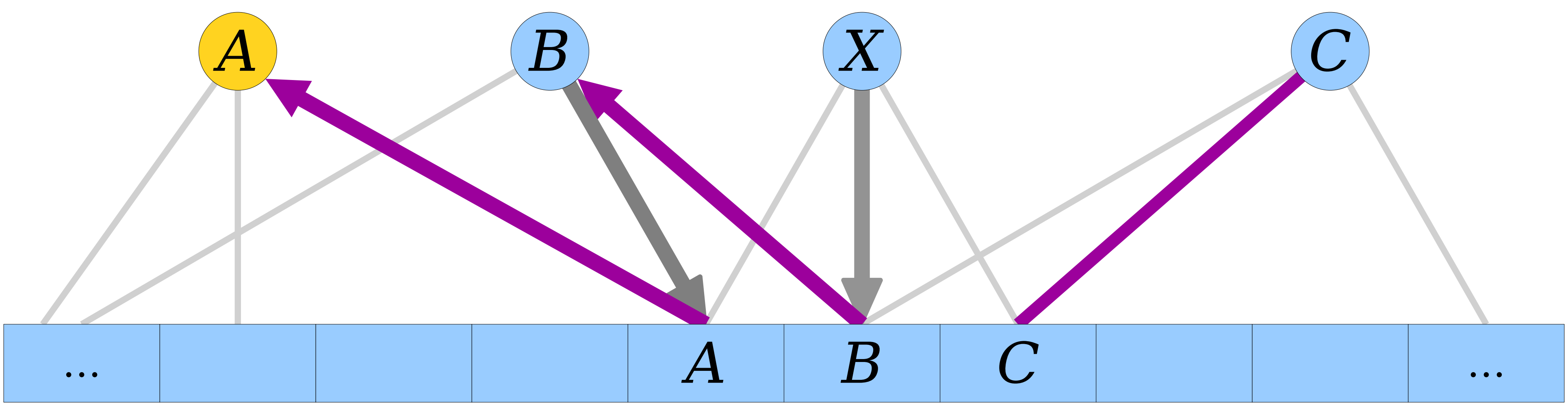
d-ary Cuckoo Hashing

- Here's our bipartite graph view of our table. Edges in ***purple*** represent where items were placed. Edges in ***gray*** represent unused alternates.
- Let's insert X into this table.



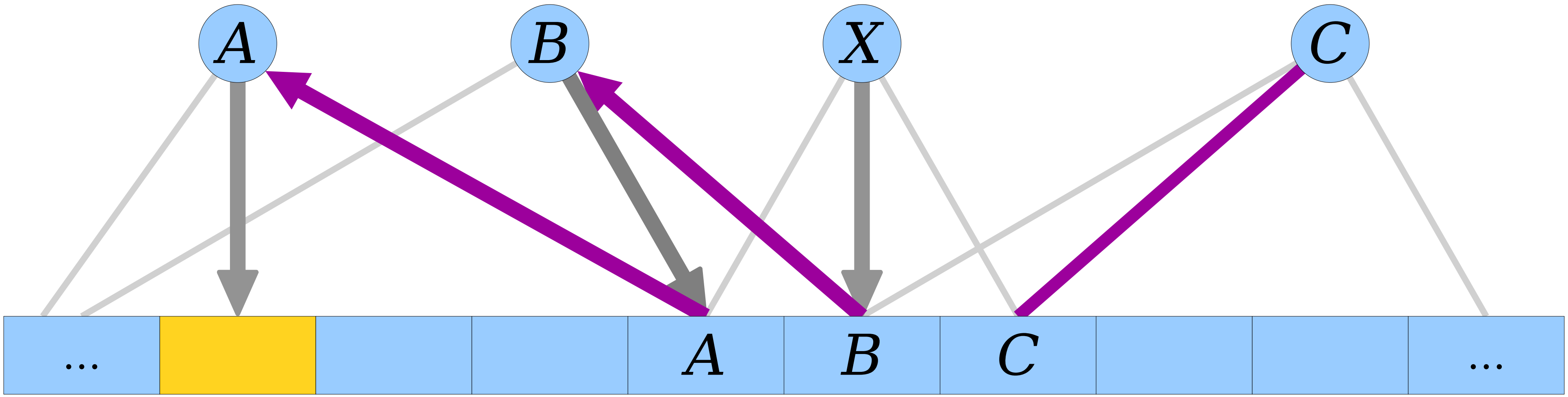
d-ary Cuckoo Hashing

- Here's our bipartite graph view of our table. Edges in ***purple*** represent where items were placed. Edges in ***gray*** represent unused alternates.
- Let's insert X into this table.



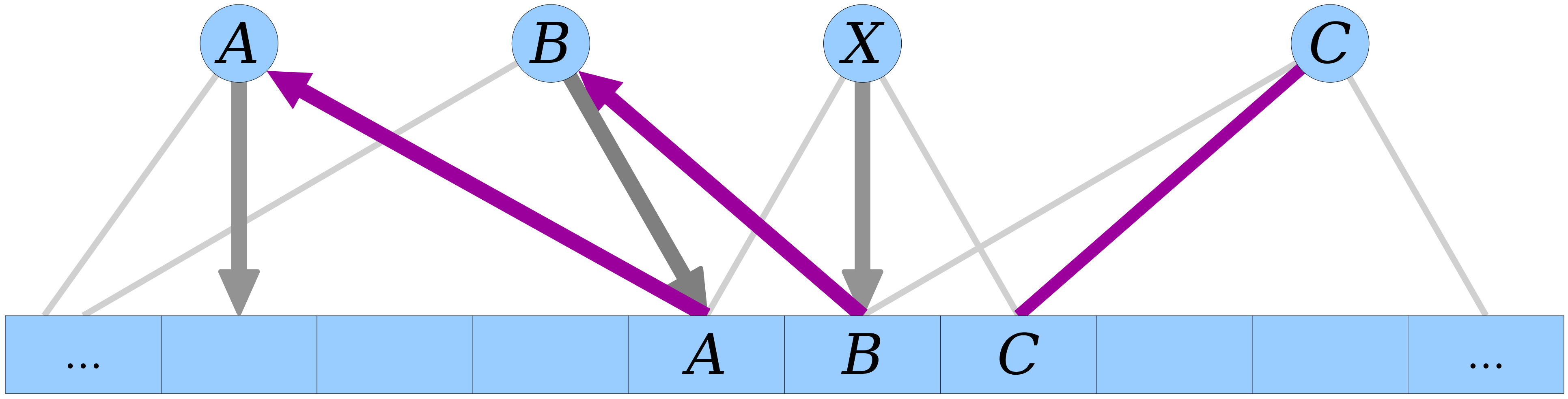
d-ary Cuckoo Hashing

- Here's our bipartite graph view of our table. Edges in ***purple*** represent where items were placed. Edges in ***gray*** represent unused alternates.
- Let's insert X into this table.



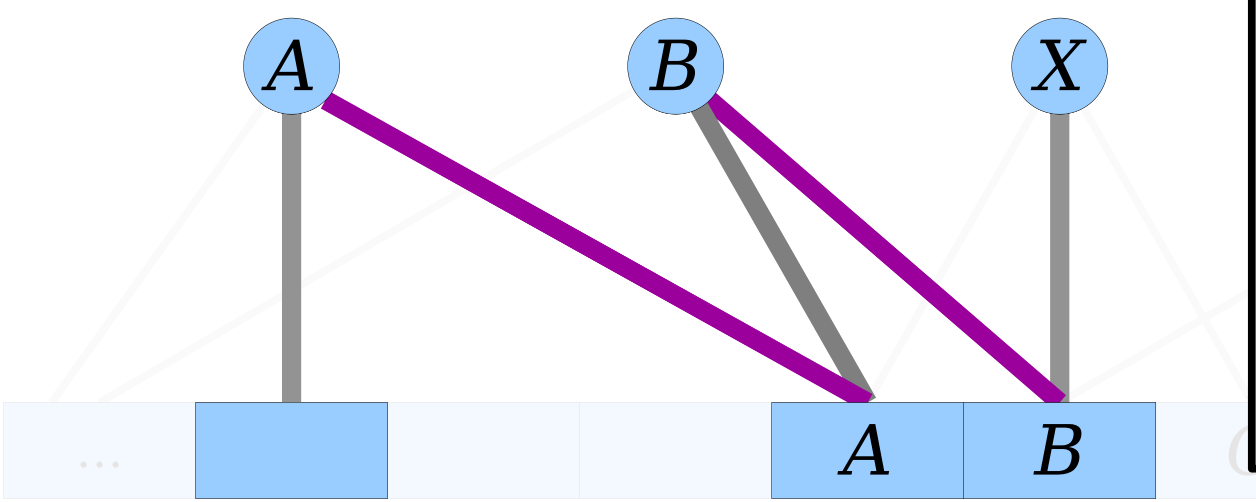
d-ary Cuckoo Hashing

- Here's our bipartite graph view of our table. Edges in ***purple*** represent where items were placed. Edges in ***gray*** represent unused alternates.
- Let's insert X into this table.



d-ary Cuckoo Hashing

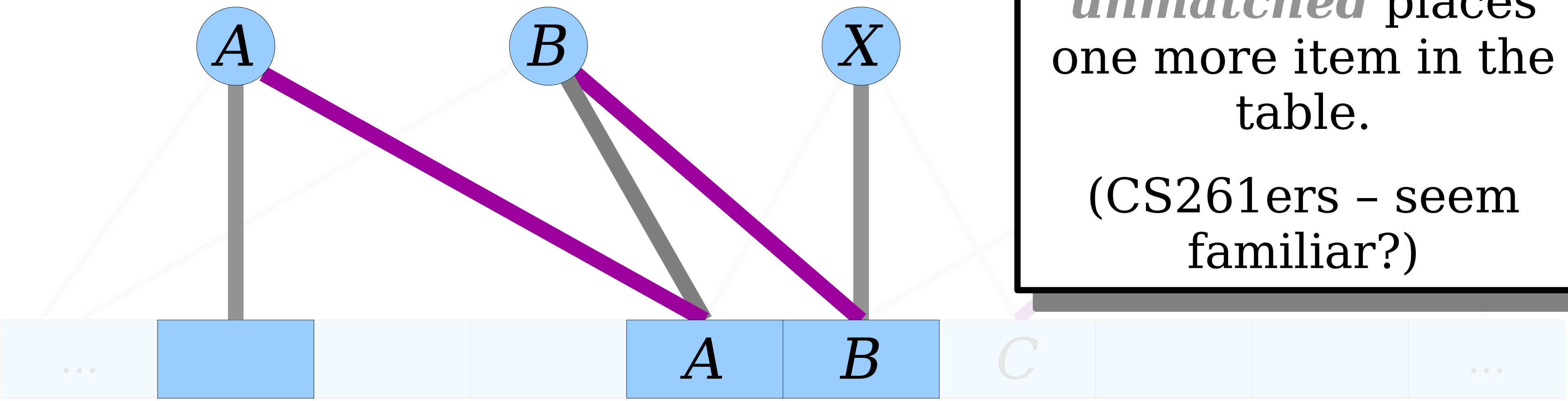
- Here's our bipartite graph view of our table. Edges in ***purple*** represent where items were placed. Edges in ***gray*** represent unused alternates.
- Let's insert *X* into this table.



This path starts with an unmatched item, ends with an unmatched slot, and alternates between ***matched*** and ***unmatched*** edges. It's called an ***augmenting path***.

d-ary Cuckoo Hashing

- Here's our bipartite graph view of our table. Edges in ***purple*** represent where items were placed. Edges in ***gray*** represent unused alternates.
- Let's insert X into this table.

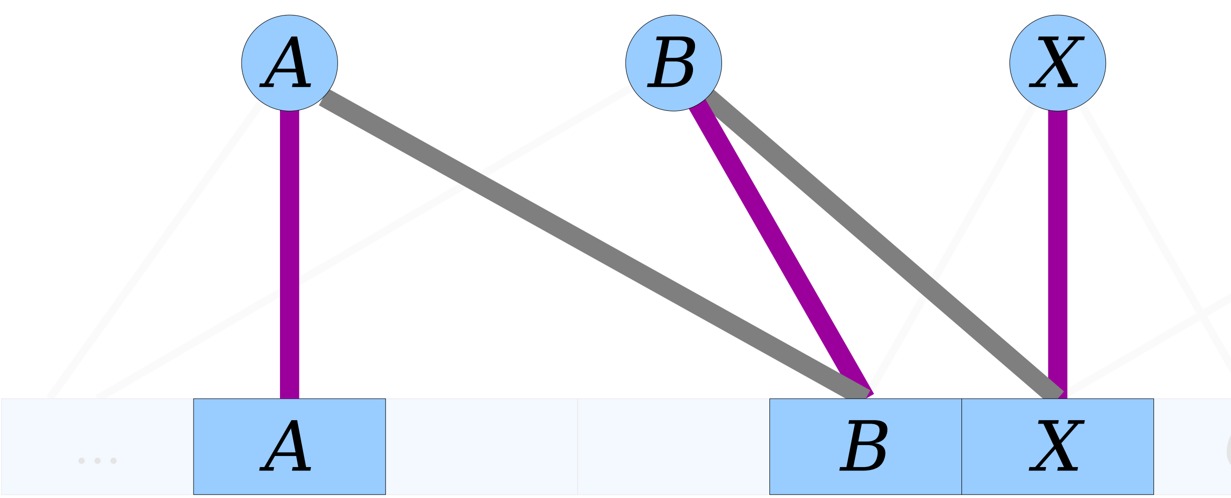


Toggling which edges are ***matched*** and ***unmatched*** places one more item in the table.

(CS261ers – seem familiar?)

d-ary Cuckoo Hashing

- Here's our bipartite graph view of our table. Edges in ***purple*** represent where items were placed. Edges in ***gray*** represent unused alternates.
- Let's insert X into this table.

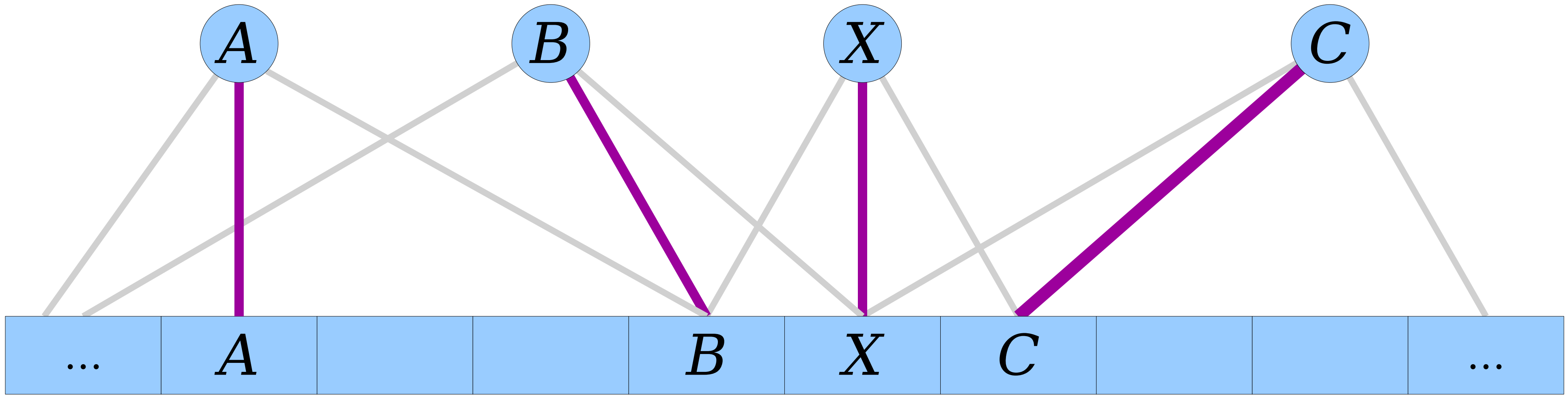


Toggling which edges are ***matched*** and ***unmatched*** places one more item in the table.

(CS261ers – seem familiar?)

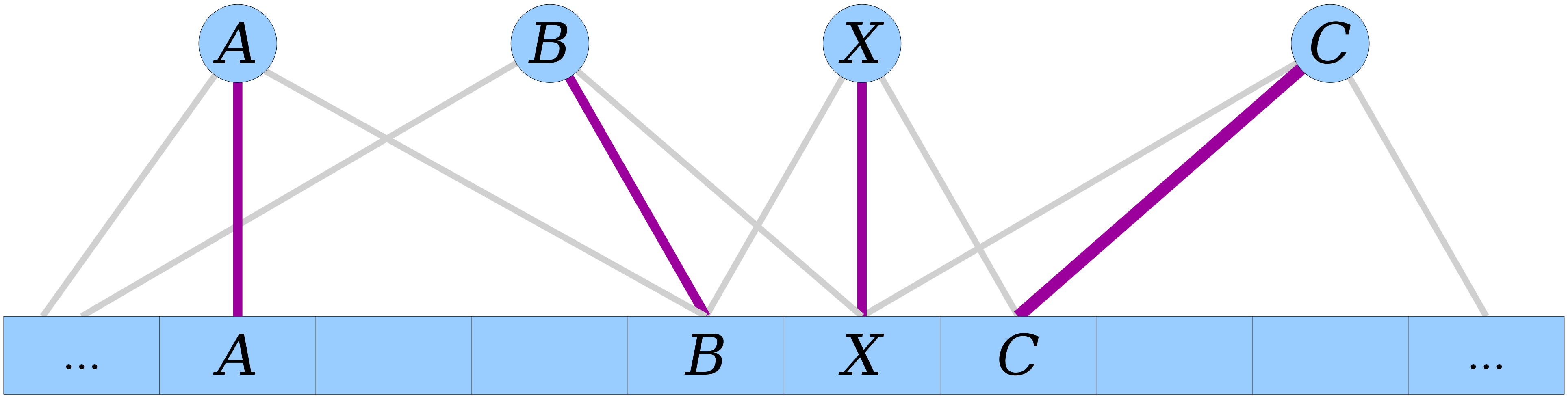
d-ary Cuckoo Hashing

- Here's our bipartite graph view of our table. Edges in ***purple*** represent where items were placed. Edges in ***gray*** represent unused alternates.
- Let's insert X into this table.



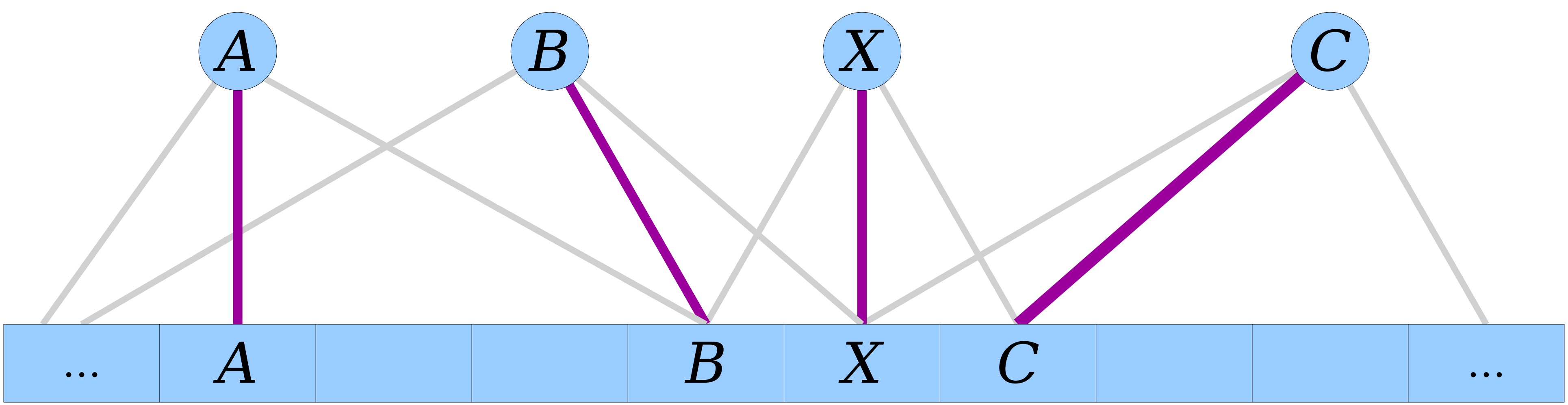
d-ary Cuckoo Hashing

- An ***augmenting path*** is a path starting at an unmatched item, ending at an unmatched slot, alternating between ***matched*** and ***unmatched*** edges.
- Flipping the edges in an alternating path produces a new matching with one more matched node.
- ***Claim:*** A chain of displacements is successful if and only if it finds an augmenting path.

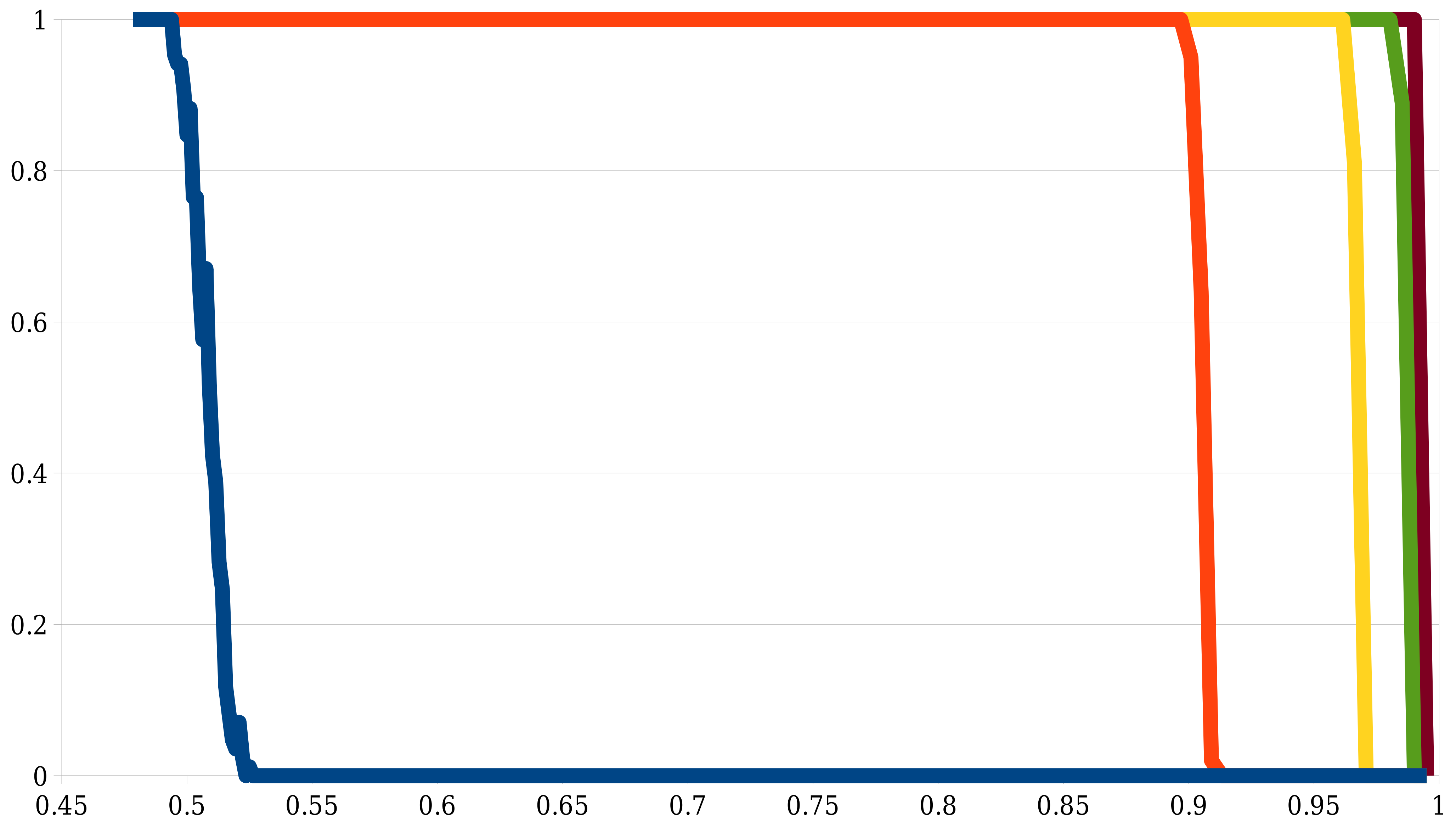


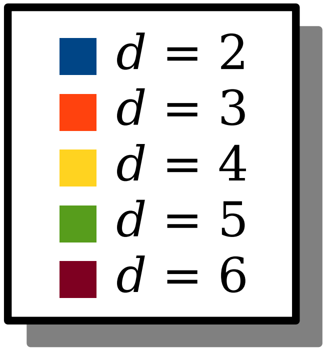
d-ary Cuckoo Hashing

- We can find an augmenting path using a ***random walk***:
  - Kick the first item out by picking any of the d options uniformly at random.
  - From that point forward, kick out a uniformly-random option of any of the d – 1 slots that wasn't the one you came from.
- Alternatively, we can use ***BFS***:
  - Start a BFS at the item to insert. Walk forwards using unmatched edges and backwards using matched edges. Stop when you find an augmenting path.
- Random walks are faster on average, but can be slower in the worst case and have trouble finding cycles. BFS is usually slower in practice but has better worst-case bounds and can find cycles.



How space-efficient is this approach?





Suppose we insert $n = \alpha m$ elements into a hash table with m slots. What is the probability that all insertions succeed?

d-ary Cuckoo Hashing

- ***Theorem:*** For each $d \geq 2$, there is a load factor α_d such that
  - for load factors $\alpha < \alpha_d$, there is a high chance that the table can be built, and
  - for load factors $\alpha > \alpha_d$, the odds that the table can be built are close to zero.
- The proof involves exploring the phase transitions in when matchings in randomly-chosen bipartite graphs start becoming increasingly rare.

d-ary Cuckoo Hashing

- The exact phase transitions for $d \geq 3$ was worked out in the late 2000s, and we have a solution courtesy of Fountoulakis and Panagiotou: the phase transition value α_d is

$$\alpha_d = \frac{x}{d\left(1 - e^{-x}\right)^{d-1}}, \text{ where } x \text{ solves } x = \frac{d\left(e^x - 1 - x\right)}{e^x - 1}$$

- These closely track the numbers I got from my simulations.

	$d = 2$	$d = 3$	$d = 4$	$d = 5$	$d = 6$
Theoretical α_d	**0.500**	**0.917**	**0.976**	**0.992**	**0.997**
Empirical α_d	**0.500**	**0.901**	**0.966**	**0.985**	**0.990**

Cuckoo Hashing Revisited

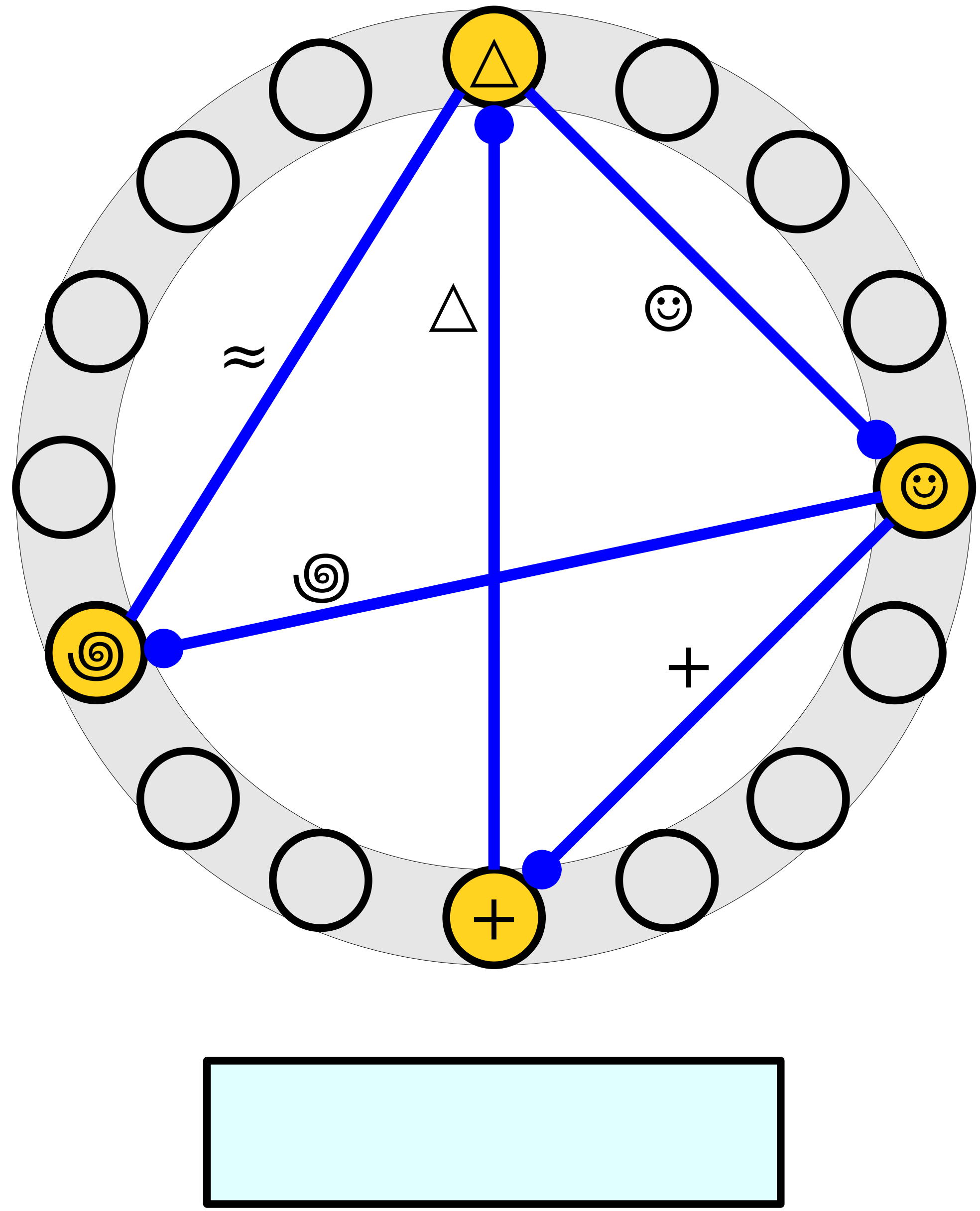
- We now have three ideas for improving our basic cuckoo hash table.
  - ***Idea 1:*** Use multiple hash functions to give items more wiggle room about where to go.
  - ***Idea 2:*** Don't place all items in the table; let some of them go somewhere else.
  - ***Idea 3:*** Allow for multiple items to be placed in each slot.
- Each of these ideas has been explored. We'll do a quick survey of what these options look like.

Cuckoo Hashing Revisited

- We now have three ideas for improving our basic cuckoo hash table.
  - ***Idea 1:*** Use multiple hash functions to give items more wiggle room about where to go.
  - ***Idea 2:*** Don’t place all items in the table; let some of them go somewhere else.
  - ***Idea 3:*** Allow for multiple items to be placed in each slot.
- Each of these ideas has been explored. We’ll do a quick survey of what these options look like.

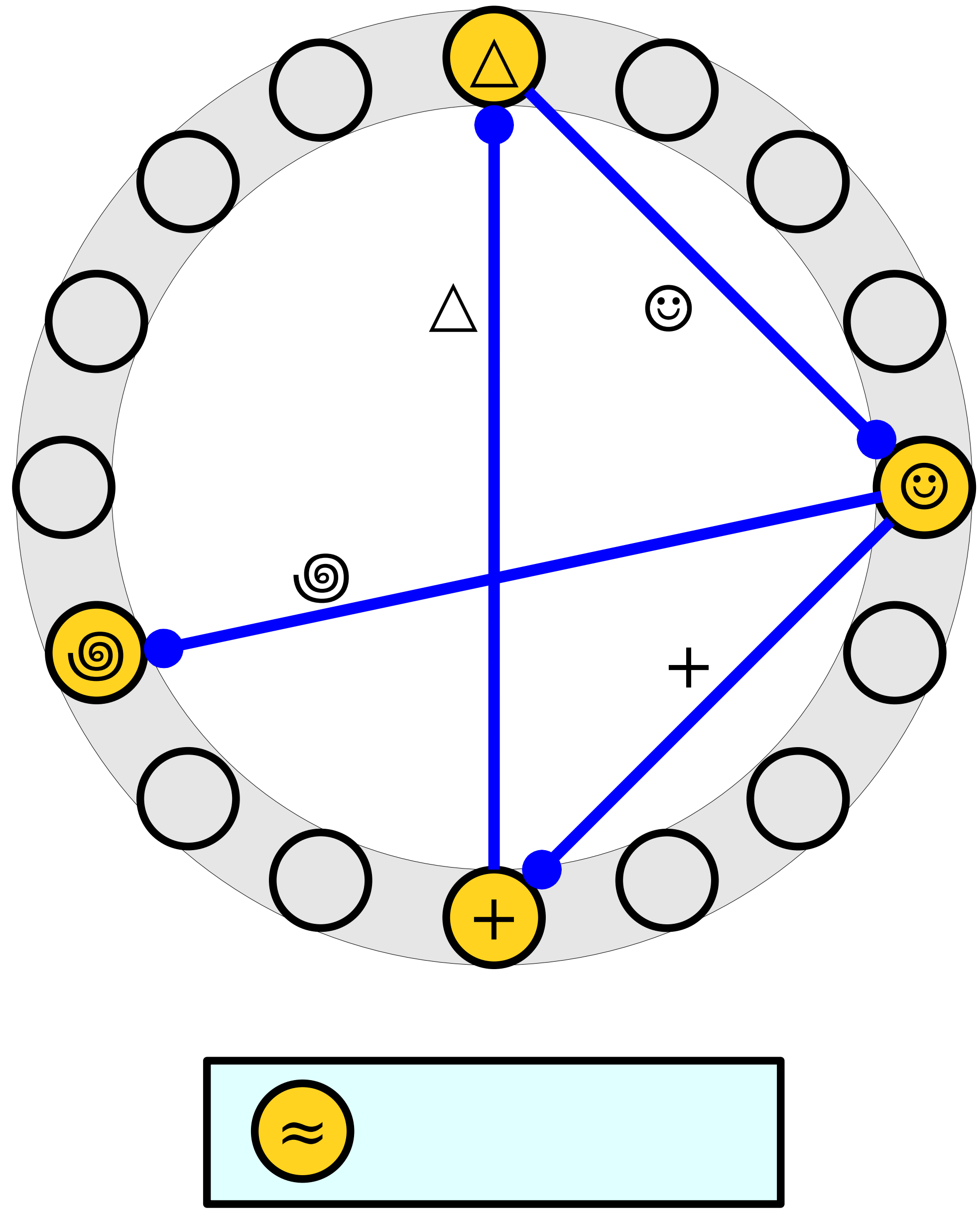
Hashing With Stashing

- Insertions fail when the newly-added node adds a second cycle into a connected component in the cuckoo graph.
- ***Idea:*** When this happens, pick an edge in that connected component that forms a cycle and store it in a secondary hash table called the ***stash***.



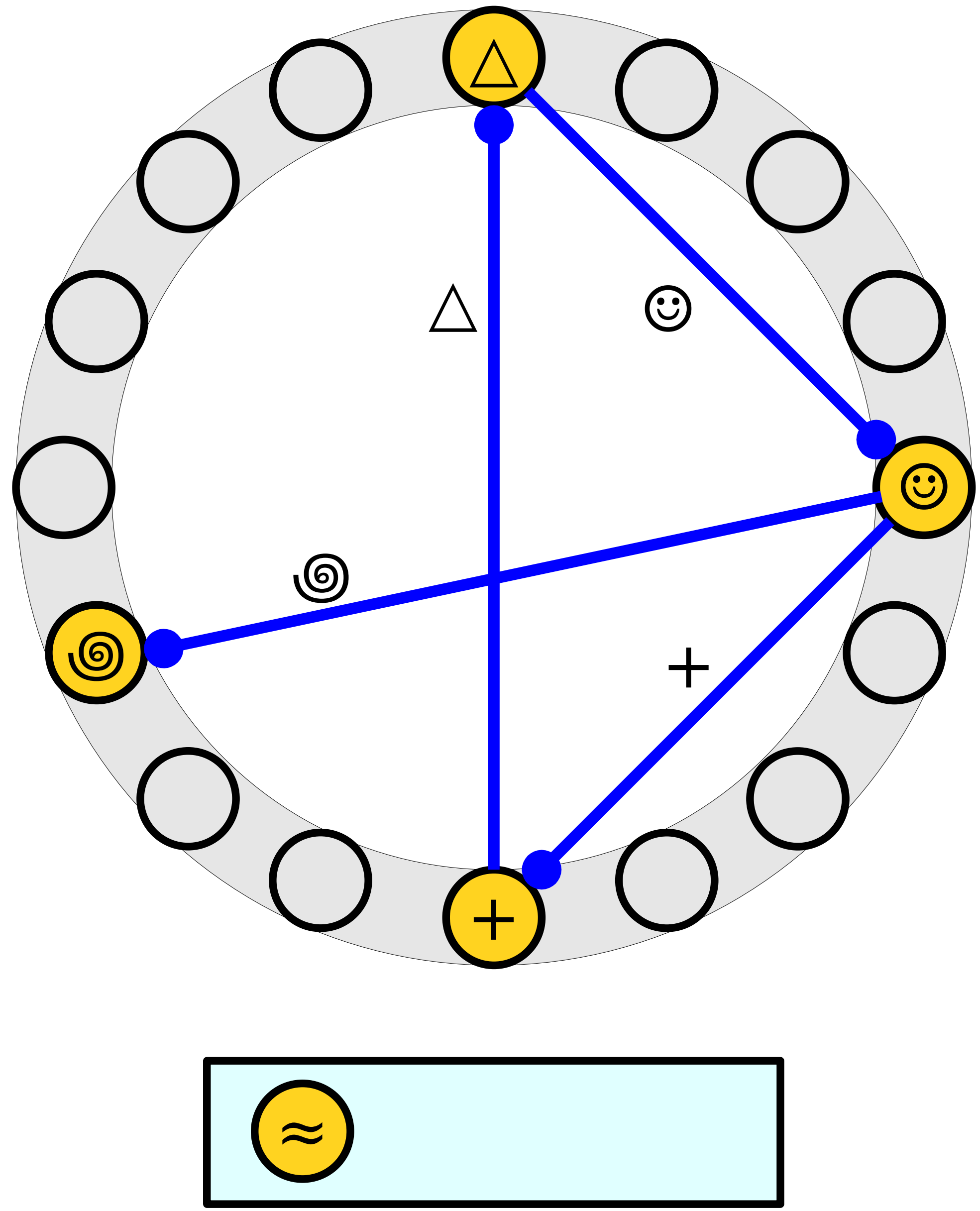
Hashing With Stashing

- Insertions fail when the newly-added node adds a second cycle into a connected component in the cuckoo graph.
- ***Idea:*** When this happens, pick an edge in that connected component that forms a cycle and store it in a secondary hash table called the ***stash***.



Hashing With Stashing

- Adding a stash doesn't increase the maximum load factor we can use.
- However, it significantly decreases the likelihood that we need to rehash.
- ***Theorem:*** With a stash of size s, the probability of having to rehash after inserting n items is $O(n^{-s})$.



Cuckoo Hashing Revisited

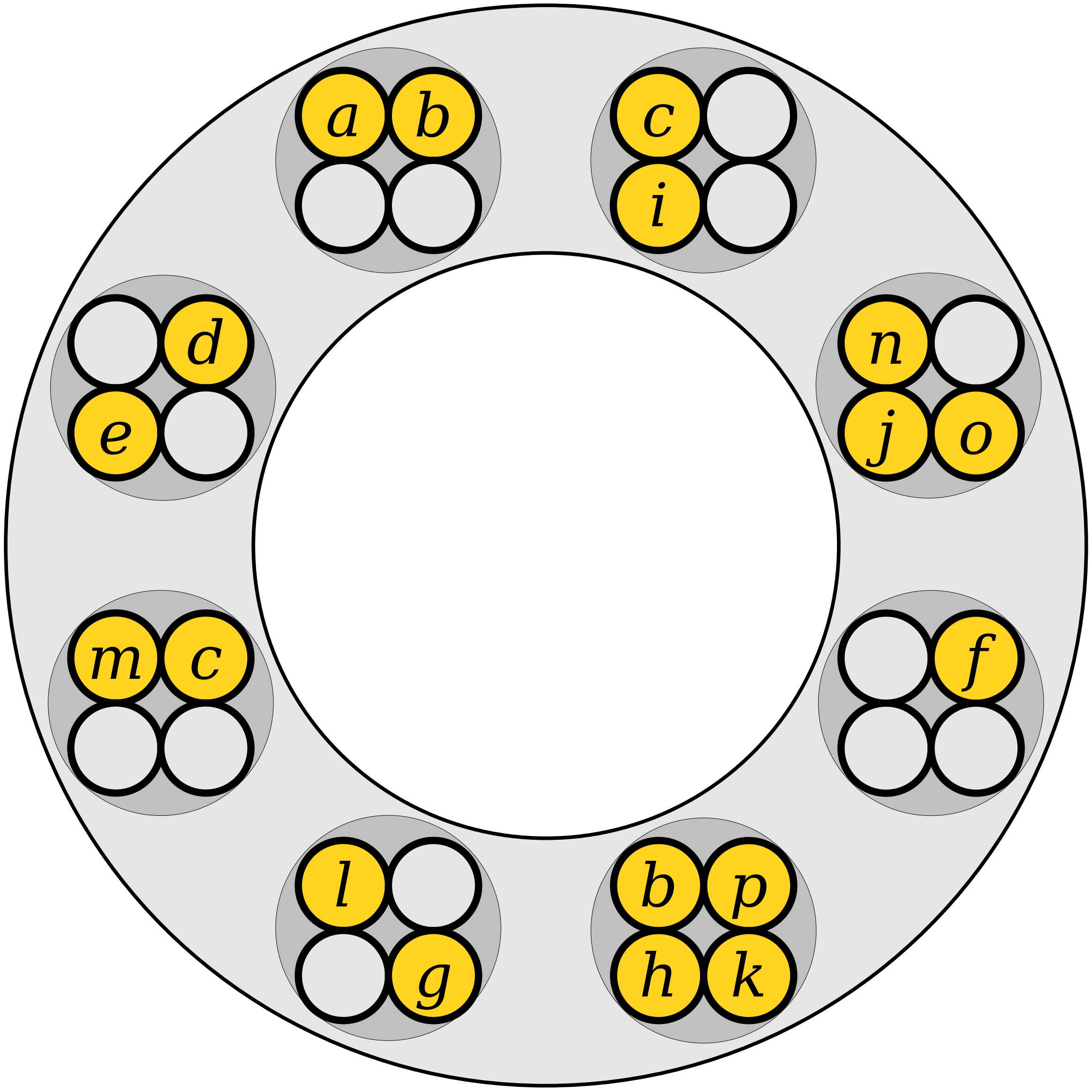
- We now have three ideas for improving our basic cuckoo hash table.
  - ***Idea 1:*** Use multiple hash functions to give items more wiggle room about where to go.
  - ***Idea 2:*** Don't place all items in the table; let some of them go somewhere else.
  - ***Idea 3:*** Allow for multiple items to be placed in each slot.
- Each of these ideas has been explored. We'll do a quick survey of what these options look like.

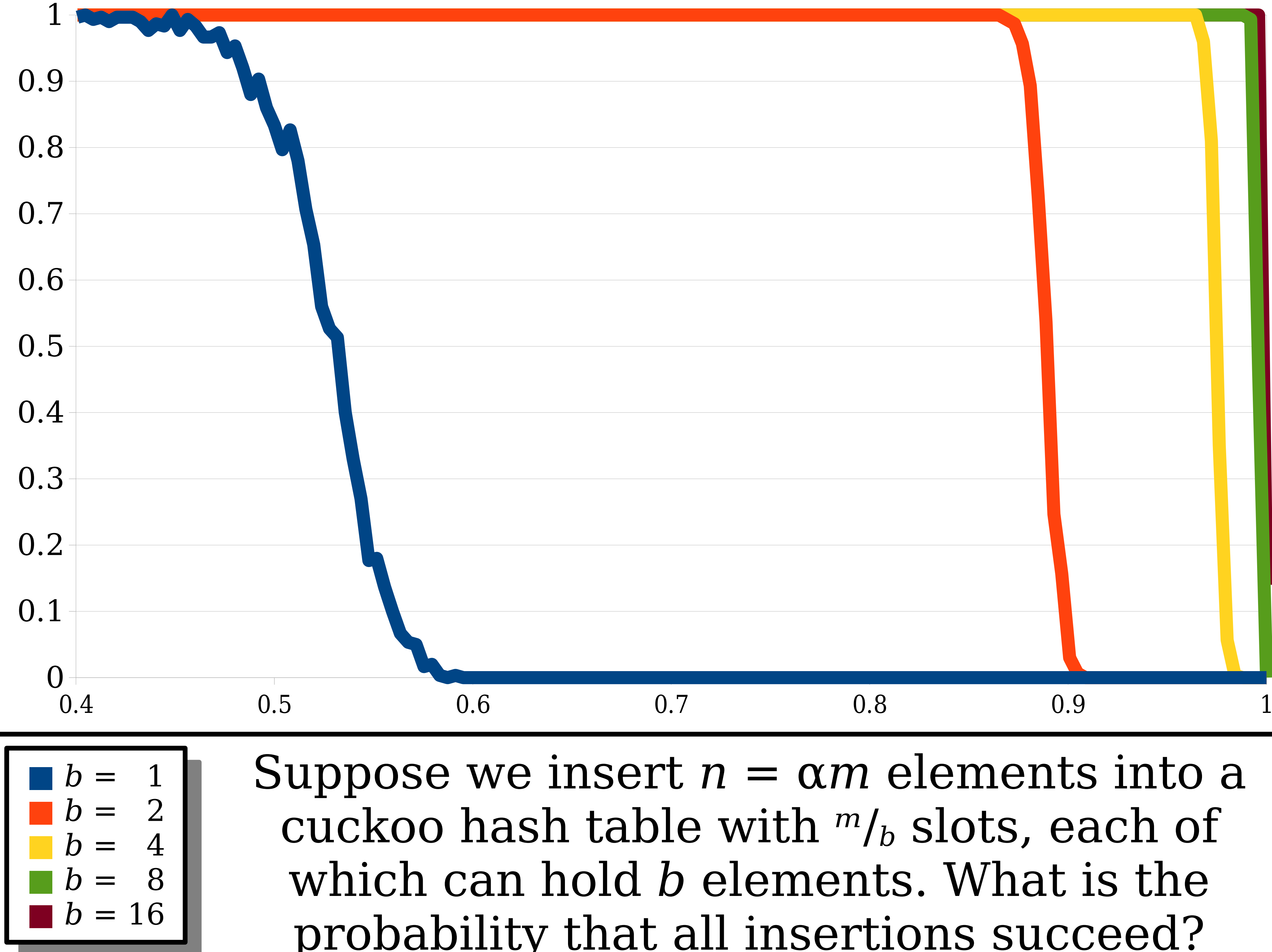
Cuckoo Hashing Revisited

- We now have three ideas for improving our basic cuckoo hash table.
  - ***Idea 1:*** Use multiple hash functions to give items more wiggle room about where to go.
  - ***Idea 2:*** Don't place all items in the table; let some of them go somewhere else.
  - ***Idea 3:*** Allow for multiple items to be placed in each slot.
- Each of these ideas has been explored. We'll do a quick survey of what these options look like.

Blocked Cuckoo Hashing

- In ***blocked cuckoo hashing***, each table slot can hold $b \geq 1$ items.
- The parameter b is often chosen so that one block fits perfectly into a cache line, improving locality of reference.
- When inserting an item, place it in one of the two slots it hashes to if there's free space in either.
- If there's no room left, displace an element. Use either a random walk or BFS to select which item to kick.
- Increasing b decreases the likelihood that insertions fail, but increases the cost of lookups and deletions.





Suppose we insert $n = \alpha m$ elements into a cuckoo hash table with $^m/_b$ slots, each of which can hold b elements. What is the probability that all insertions succeed?

Blocked Cuckoo Hashing

- Suppose we have a table with m/b slots, each of which hold b items. Assume $n = m\alpha$ and we use two hash functions.
- ***Theorem:*** Blocked cuckoo hashing succeeds with high probability as long as

$$b \geq 1 + \frac{\ln\left(\frac{\alpha}{\alpha-1}\right)}{1 - \ln 2}.$$

- There is a phase transition result here, but, empirically, the theory hasn't fully captured it.

	$b = 1$	$b = 2$	$b = 4$	$b = 8$	$b = 16$
Predicted max α	**0.500**	**0.576**	**0.715**	**0.895**	**0.990**
Empirical max α	**0.453**	**0.870**	**0.966**	**0.992**	**0.997**

To Summarize

Summary of Cuckoo Hashing

- Cuckoo hashing is a fast and powerful way to build perfect hash tables.
- We can increase the number of hash functions to increase the load factor, though at a cost to lookup and insert times.
- We can use a stash to hold "meddlesome" items, which increases the likelihood that the table can be built.
- We can increase the number of items per slot to increase the load factor, though at a cost to lookup and insert times.

Major Ideas for Today

- Randomized data structures using multiple hash functions can often be analyzed from a graph-theoretic perspective.
- Many properties of random graphs exhibit sharp phase transitions.
- Running experiments is a great way to learn more about randomized data structures.
- Modeling multiple hashes into a table as a bipartite graph is a useful tool.

Next Time

- ***k-Independent Hash Functions***
  - Quantifying how random we need our hash functions to be.
- ***Frequency Estimation***
  - Estimating frequency counts in sublinear space.
- ***Count-Min Sketches***
  - Finding frequent items without actually storing frequencies.