Cuckoo hashing
Main ideas

Cuckoo = bird that lays eggs in other birds’ nests when it hatches, the baby pushes other eggs and chicks out

Cuckoo hashing = open addressing, only two possible cells per key
Invented by Rasmus Pagh and Flemming Friche Rodler in 2001

Lookup: always $O(1)$ time – just look in those two cells

Deletion: easy, just blank your cell

Insertion: if a key is already in your cell, push it out (and recursively insert it into its other cell)
Comparison with other methods

When an adversary can pick a bad key, other methods can be slow:

$\Theta(\log n / \log \log n)$ for hash chaining

$\Theta(\log n)$ for linear probing

In contrast, cuckoo hash lookup is always $O(1)$

Cuckoo hashing is useful when keys can be adversarial or when lookups must always be fast

  e.g. internet packet filter

But for fast average case, linear probing may be better
Requirements and variations

Sometimes the two cells for each key are in two different arrays but it works equally well for both to be in a single array (We will use the single-array version)

Basic method needs load factor $\alpha < 1/2$ to avoid infinite recursion

More complicated variations can handle any $\alpha < 1$

- Store a fixed number of keys per cell, not just one
- Keep a “stash” of $O(1)$ keys that don’t fit
Easy operations: Lookup and delete

With array $A$, hash functions $h_1$ and $h_2$ such that $h_1(k) \neq h_2(k)$:

To look up value for key $k$:

- if $A[h_1(k)]$ contains $k$:
  - return value in $A[h_1(k)]$
- if $A[h_2(k)]$ contains $k$:
  - return value in $A[h_2(k)]$
- raise exception

To delete key $k$:

- if $A[h_1(k)]$ contains $k$:
  - empty $A[h_1(k)]$ and return
- if $A[h_2(k)]$ contains $k$:
  - empty $A[h_2(k)]$ and return
- raise exception
To insert key $k$ with value $x$:

$$i = h_1(k)$$

while $A[i]$ is non-empty:

- swap $k, x$ with $A[i]$
- $i = h_1(k) + h_2(k) - i$

$A[i] = k, x$

Not shown: test for infinite loop

If we ever return to a state where we are trying to re-insert the original key $k$ into its original cell $h_1(k)$, the insertion fails (put $k$ into the stash or rebuild the whole data structure)
### Example

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
<th>h1</th>
<th>h2</th>
<th>A:</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>7</td>
<td>2</td>
<td>4</td>
<td>i = 2 3 4 5</td>
</tr>
<tr>
<td>Y</td>
<td>9</td>
<td>3</td>
<td>4</td>
<td>X,7</td>
</tr>
<tr>
<td>Z</td>
<td>8</td>
<td>2</td>
<td>3</td>
<td>X,7 Y,9</td>
</tr>
<tr>
<td>W</td>
<td>6</td>
<td>2</td>
<td>5</td>
<td>Z,8 Y,9 X,7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X,7 Z,8 Y,9 W,6</td>
</tr>
</tbody>
</table>
Graph view of cuckoo state

Vertices = array cells

Edges = pairs of cells used by the same key
Directed away from cell where the key is stored

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
<th>h1</th>
<th>h2</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>7</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Y</td>
<td>9</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Z</td>
<td>8</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>W</td>
<td>6</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

A:

\[A_i = \{X, 7, Z, 8, Y, 9, W, 6\} \]

Key properties: Each vertex has at most one outgoing edge
Each component is a pseudotree (cycle with trees attached)
Add edge $h_1(k) \rightarrow h_2(k)$ to the graph.

If $h_1(k)$ already had an outgoing edge:

Follow path of outgoing edges, reversing each edge in the path.

Case analysis:

- $h_1(k)$ was in a tree: path stops at previous root.
- $h_1(k)$ was in a pseudotree and $h_2(k)$ was in a tree: path loops through cycle, comes back out, reverses new edge, and stops at root of tree.
- $h_1(k)$ and $h_2(k)$ are both in pseudotrees (or same pseudotree): fails, too many edges for all vertices to have only one outgoing edge.
Summary of graph interpretation

The insertion algorithm

- Works correctly when the undirected graph with edges $h_1(k) - h_2(k)$ has at most one cycle per component.
- Fails when any component has two cycles (equivalently: more edges than vertices).
- Takes time proportional to the component size.

The graph is random (each two vertices equally likely to be edge).
We know a lot about random graphs!

For $n$ vertices and $\alpha n$ edges, $\alpha < 1/2$:
- With high probability, all components are trees or pseudotrees
- Prob. component of $v$ has size $\ell$ is exponentially small in $\ell$
- Expected size of component containing $v$ is $O(1)$
- But expected largest component size $\Theta(\log n)$
- Expected number of pseudotrees is $O(1)$

For $\alpha = 1/2$:
- There is a giant component with $\Theta(n^{2/3})$ vertices (with high probability)
- The rest of the graph is still small trees and pseudotrees
What random graphs imply for cuckoo hashing

For load factor $\alpha < 1/2$:

- With high probability, all keys can be inserted
- Expected time per insertion is $O(1)$
- Expected time of slowest insertion is $O(\log n)$

For load factor $\alpha \geq 1/2$:

- It doesn’t work (in simplest variation)
$k$-independent hash functions
The problem

All analysis so far has assumed hash function is random

But that is rarely achievable in practice
  ▶ Cryptographic functions act like random but too slow
  ▶ IdentityHashMap not usable in all applications and doesn’t allow changing to a new hash function

Many software libraries use ad-hoc hash functions that are arbitrary, but not random
  ▶ We can’t prove anything about how well they work!

Instead, we want a function that
  ▶ Can be constructed using only a small seed of randomness
  ▶ Is fast (theoretically and in practice) to evaluate
  ▶ Can be proven to work well with hashing algorithms
Choose function $h$ randomly from a bigger family $H$ of functions.

If $H = \text{all functions}$, $h$ is uniformly random (previous assumption).

If $H$ is smaller, $h$ will be less random.

Define $H$ to be \textit{k-independent} if every $k$-tuple of keys has independent outputs (every tuple of outputs is equally likely).

Bigger values of $k$ give stronger independence guarantees.

An example of a (bad) 1-independent hash function: choose one random number $r$ and define $h_r$ to ignore its argument and return $r$.

So we are selecting a function randomly from the set $H = \{ h_r \}$. 

Is $k$-independence enough?

Expected-time analysis of hash chaining only considers pairs of keys (expected time is sum of probabilities that another key collides with given key).

If we use a 2-independent hash function, these probabilities are the same as for a fully independent hash function.

Expected-time analysis of linear probing has been done with 5-independent hashing [Pagh, Pagh & Ružić].

But there exist 4-independent hash functions designed to make linear probing bad [Pătrașcu & Thorup].

Cuckoo hashing requires $\Theta(\log n)$-independence.
Algebraic method for $k$-independence

From $b$-bit numbers (that is, $0 \leq \text{value} < 2^b$) to the range $[0, N-1]$

Choose (nonrandom) prime number $p > 2^b$

Choose $k$ random coefficients $a_0, a_1, \ldots a_{k-1} \pmod{p}$

$$h(x) = \left( \left( \sum_i a_i x^i \right) \pmod{p} \right) \pmod{N}$$

Works because, for every $k$-tuple of keys and every $k$-tuple of outputs, exactly one polynomial mod $p$ produces that output

$O(k)$ arithmetic operations but multiplications can be slow
Tabulation hashing

Represent key $x$ as a base-$B$ number for an appropriate base $B$

$$x = \sum_i x_i B^i \text{ with } 0 \leq x_i < B$$

E.g. for $B = 256$, $x_i$ can be calculated by $(x\gg(i<<3))\&0xff$ (using only shifts and masks, no multiplication)

Let $d$ be number of digits ($d = 4$ for 32-bit keys and $B = 256$)

Initialize: fill a $d \times B$ table $T$ with random numbers

$$h(x) = \text{bitwise exclusive or of } T[i, x_i] \ (i = 0, 1, \ldots d - 1)$$

3-independent but not 4-independent; works anyway for linear probing and static cuckoo hashing [Pătraşcu & Thorup]
Summary
Summary

- Dictionary problem: Updating and looking up key–value pairs
- In standard libraries (e.g. Java HashMap, Python dict)
- Hash tables: direct addressing + hash function
- Hash functions: perfect, cryptographic, random, $k$-independent (algebraic and tabulation)
- Hash chaining, expected time, expected worst case
- Linear probing, expected time, expected worst case
- Cuckoo hashing and random graphs
- Reviewed basic probability theory + Chernoff bound