Hash Tables

```c
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
    // guaranteed to be random.
}
```

xkcd. http://xkcd.com/221/. “Random Number.” Used with permission under Creative Commons 2.5 License.
The Search Problem

- Find items with keys matching a given search key
  - Given an array $A$, containing $n$ keys, and a search key $x$, find the index $i$ such as $x = A[i]$
  - As in the case of sorting, a key could be part of a large record.

example of a record

<table>
<thead>
<tr>
<th>Key</th>
<th>other data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Special Case: Dictionaries

- **Dictionary** = data structure that supports mainly two basic operations: **insert** a new item and **return an item with a given key**.
  - Queries: return information about the set $S$:
    - get $(S, k)$
  - Modifying operations: change the set
    - put $(S, k)$: insert new or update the item of key $k$.
    - remove $(S, k)$ – **not very often**
Direct Addressing

- Assumptions:
  - Key values are distinct
  - Each key is drawn from a universe $U = \{0, 1, \ldots, N - 1\}$

- Idea:
  - Store the items in an array, indexed by keys

- **Direct-address table** representation:
  - An array $T[0 \ldots N - 1]$
  - Each *slot*, or position, in $T$ corresponds to a key in $U$
  - For an element $x$ with key $k$, a pointer to $x$ (or $x$ itself) will be placed in location $T[k]$
  - If there are no elements with key $k$ in the set, $T[k]$ is empty, represented by NIL
Direct Addressing (cont’d)

(insert/delete in $O(1)$ time)
Comparing Different Implementations

- Implementing dictionaries using:
  - Direct addressing
  - Ordered/unordered arrays
  - Ordered linked lists
  - Balanced search trees

<table>
<thead>
<tr>
<th></th>
<th>put</th>
<th>get</th>
</tr>
</thead>
<tbody>
<tr>
<td>direct addressing</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>ordered array</td>
<td>O(N)</td>
<td>O(lgN)</td>
</tr>
<tr>
<td>unordered array</td>
<td>O(1)</td>
<td>O(N)</td>
</tr>
<tr>
<td>ordered list</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>balance search tree</td>
<td>O(lgN)</td>
<td>O(lgN)</td>
</tr>
</tbody>
</table>
Hash Tables

- When \( n \) is much smaller than \( \max(U) \), where \( U \) is the set of all keys, a **hash table** requires much less space than a **direct-address table**
  - Can reduce storage requirements to \( O(n) \)
  - Can still get \( O(1) \) search time, but on the **average case**, not the **worst case**
Hash Tables

- Use a function $h$ to compute the slot for each key
- Store the element in slot $h(k)$

- A **hash function** $h$ transforms a key into an index in a hash table $T[0...N-1]$: $h : U \rightarrow \{0, 1, \ldots, N - 1\}$

- We say that $k$ hashes to $h(k)$, hash value of $k$.

- Advantages:
  - Reduce the range of array indices handled: $N$ instead of $\text{max}(U)$
  - Storage is also reduced
Example: HASH TABLES

**U**
(universe of keys)

**K**
(actual keys)

\[ h(k_1) \]
\[ h(k_2) = h(k_5) \]
\[ h(k_3) \]
\[ m - 1 \]
Example

Suppose that the keys are nine-digit social security numbers

Possible hash function

\[ h(ssn) = sss \mod 100 \text{ (last 2 digits of ssn)} \]

e.g., if \( ssn = 10123411 \) then \( h(10123411) = 11 \)
Do you see any problems with this approach?

U (universe of keys)

K (actual keys)

\[ h(k_1) = h(k_5) \]

Collisions!
Collisions

- Two or more keys hash to the same slot!!
- For a given set of \( n \) keys
  - If \( n \leq N \), collisions may or may not happen, depending on the hash function
  - If \( n > N \), collisions will definitely happen (i.e., there must be at least two keys that have the same hash value)
- Avoiding collisions completely is hard, even with a good hash function
Hash Functions

- A hash function transforms a key into a table address

**What makes a good hash function?**

1. Easy to compute
2. Approximates a random function: for every input, every output is equally likely *(simple uniform hashing)*

- In practice, it is very hard to satisfy the simple uniform hashing property
  - i.e., we don’t know in advance the probability distribution that keys are drawn from
Good Approaches for Hash Functions

- Minimize the chance that closely related keys hash to the same slot
  - Strings such as *stop*, *tops*, and *pots* should hash to different slots
- Derive a hash value that is independent from any patterns that may exist in the distribution of the keys.
The Division Method

- **Idea:**
  - Map a key $k$ into one of the $N$ slots by taking the remainder of $k$ divided by $N$
  - $h(k) = k \mod N$

- **Advantage:**
  - fast, requires only one operation

- **Disadvantage:**
  - Certain values of $N$ are bad, e.g.,
    - power of 2
    - non-prime numbers
Example - The Division Method

- If $N = 2^p$, then $h(k)$ is just the least significant $p$ bits of $k$
  - $p = 1 \Rightarrow N = 2$
    - $h(k) = \{0, 1\}$, least significant 1 bit of $k$
  - $p = 2 \Rightarrow N = 4$
    - $h(k) = \{0, 1, 2, 3\}$, least significant 2 bits of $k$

- Choose $N$ to be a prime, not close to a power of 2
  - Column 2: $k \mod 97$
  - Column 3: $k \mod 100$

<table>
<thead>
<tr>
<th>$N$</th>
<th>97</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>16838</td>
<td>57</td>
<td>38</td>
</tr>
<tr>
<td>5758</td>
<td>35</td>
<td>58</td>
</tr>
<tr>
<td>10113</td>
<td>25</td>
<td>13</td>
</tr>
<tr>
<td>17515</td>
<td>55</td>
<td>15</td>
</tr>
<tr>
<td>31051</td>
<td>11</td>
<td>51</td>
</tr>
<tr>
<td>5627</td>
<td>1</td>
<td>27</td>
</tr>
<tr>
<td>23010</td>
<td>21</td>
<td>10</td>
</tr>
<tr>
<td>7419</td>
<td>47</td>
<td>19</td>
</tr>
<tr>
<td>16212</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>4086</td>
<td>12</td>
<td>86</td>
</tr>
<tr>
<td>2749</td>
<td>33</td>
<td>49</td>
</tr>
<tr>
<td>12767</td>
<td>60</td>
<td>67</td>
</tr>
<tr>
<td>9084</td>
<td>63</td>
<td>84</td>
</tr>
<tr>
<td>12060</td>
<td>32</td>
<td>60</td>
</tr>
<tr>
<td>32225</td>
<td>21</td>
<td>25</td>
</tr>
<tr>
<td>17543</td>
<td>83</td>
<td>43</td>
</tr>
<tr>
<td>25089</td>
<td>63</td>
<td>89</td>
</tr>
<tr>
<td>21183</td>
<td>37</td>
<td>83</td>
</tr>
<tr>
<td>25137</td>
<td>14</td>
<td>37</td>
</tr>
<tr>
<td>25566</td>
<td>55</td>
<td>66</td>
</tr>
<tr>
<td>26966</td>
<td>0</td>
<td>66</td>
</tr>
<tr>
<td>4978</td>
<td>31</td>
<td>78</td>
</tr>
<tr>
<td>20495</td>
<td>28</td>
<td>95</td>
</tr>
<tr>
<td>10311</td>
<td>29</td>
<td>11</td>
</tr>
<tr>
<td>11367</td>
<td>18</td>
<td>67</td>
</tr>
</tbody>
</table>
The Multiplication Method

**Idea:**
- Multiply key \( k \) by a constant \( A \), where \( 0 < A < 1 \)
- Extract the fractional part of \( kA \)
- Multiply the fractional part by \( N \)
- Take the floor of the result

\[
h(k) = \lfloor N (kA - \lfloor kA \rfloor) \rfloor
\]

**Disadvantage:** A little slower than division method

**Advantage:** Value of \( N \) is not critical, e.g., typically \( 2^p \)
Hash Functions

- A hash function is usually specified as the composition of two functions:

  **Hash code:**
  \( h_1: \text{keys} \rightarrow \text{integers} \)

  **Compression function:**
  \( h_2: \text{integers} \rightarrow [0, N - 1] \)

  Typically, \( h_2 \) is mod N.

- The hash code is applied first, and the compression function is applied next on the result, i.e.,
  \[ h(x) = h_2(h_1(x)) \]

- The goal of the hash function is to “disperse” the keys in an apparently random way.
Typical Function for $H_1$

- **Polynomial accumulation:**
  - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits)
    $$a_0 a_1 \ldots a_{n-1}$$
  - We evaluate the polynomial $p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_{n-1}z^{n-1}$
    at a fixed value $z$, ignoring overflows
  - Especially suitable for strings (e.g., the choice $z = 33$ gives at most 6 collisions on a set of 50,000 English words)

- Polynomial $p(z)$ can be evaluated in $O(n)$ time using Horner’s rule:
  - The following polynomials are successively computed, each from the previous one in $O(1)$ time
    $$p_0(z) = a_{n-1}$$
    $$p_i(z) = a_{n-i-1} + zp_{i-1}(z) \quad (i = 1, 2, \ldots, n - 1)$$
  - We have $p(z) = p_{n-1}(z)$
  - Good values for $z$: 33, 37, 39, and 41.
Tabulation-Based Hashing

- Suppose each key can be viewed as a tuple, \( k = (x_1, x_2, \ldots, x_d) \), for a fixed \( d \), where each \( x_i \) is in the range \([0, M - 1]\).
- There is a class of hash functions we can use, which involve simple table lookups, known as tabulation-based hashing.
- We can initialize \( d \) tables, \( T_1, T_2, \ldots, T_d \), of size \( M \) each, so that each \( T_i[j] \) is a uniformly chosen independent random number in the range \([0, N - 1]\).
- We then compute the hash function, \( h(k) \), as
  \[
  h(k) = T_1[x_1] \oplus T_2[x_2] \oplus \cdots \oplus T_d[x_d],
  \]
  where \( \oplus \) denotes the bitwise exclusive-or function.
- Because the values in the tables are themselves chosen at random, such a function is itself fairly random. For instance, it can be shown that such a function will cause two distinct keys to collide at the same hash value with probability \( 1/N \), which is what we would get from a perfectly random function.
Compression Functions

- **Division**:
  - \( h_2(y) = y \mod N \)
  - The size \( N \) of the hash table is usually chosen to be a prime
  - The reason has to do with number theory and is beyond the scope of this course

- **Random linear hash function**:
  - \( h_2(y) = (ay + b) \mod N \)
  - \( a \) and \( b \) are random nonnegative integers such that \( a \mod N \neq 0 \)
  - Otherwise, every integer would map to the same value \( b \)
Handling Collisions

- We will review the following methods:
  - Separate Chaining
  - Open addressing
    - Linear probing
    - Quadratic probing
    - Double hashing
Handling Collisions Using Chaining

**Idea:**

- Put all elements that hash to the same slot into a linked list

- Slot $j$ contains a pointer to the head of the list of all elements that hash to $j$
Collision with Chaining

- Choosing the size of the table
  - Small enough not to waste space
  - Large enough such that lists remain short
  - Typically 1/5 or 1/10 of the total number of elements

- How should we keep the lists: ordered or not?
  - Not ordered!
    - Insert is fast
    - Can easily remove the most recently inserted elements
Insert in Hash Tables

Algorithm `put(k, v):`  // k is a new key
    `t = A[h(k)].put(k,v)`
    `n = n + 1`
    `return t`

- Worst-case running time is $O(1)$
- Assumes that the element being inserted isn’t already in the list
- It would take an additional search to check if it was already inserted
Deletion in Hash Tables

**Algorithm** `remove(k)`:

- `t = A[h(k)].remove(k)`
- `if t ≠ null then` {k was found}
  - `n = n - 1`
- `return t`

- Need to find the element to be deleted.
- **Worst-case running time:**
  - Deletion depends on searching the corresponding list
Searching in Hash Tables

**Algorithm** `get(k):
    return A[h(k)].get(k)

- Running time is proportional to the length of the list of elements in slot `h(k)`
Analysis of Hashing with Chaining: Worst Case

- How long does it take to search for an element with a given key?
- Worst case:
  - All \( n \) keys hash to the same slot
  - Worst-case time to search is \( \Theta(n) \), plus time to compute the hash function
Analysis of Hashing with Chaining: Average Case

- **Average case**
  - depends on how well the hash function distributes the \(n\) keys among the \(N\) slots

- **Simple uniform hashing** assumption:
  - Any given element is equally likely to hash into any of the \(N\) slots (i.e., probability of collision \(Pr(h(x)=h(y))\), is 1/\(N\))

- **Length of a list**:
  \(T[j].\text{size} = n_j, \quad j = 0, 1, \ldots, N - 1\)

- **Number of keys in the table**:
  \(n = n_0 + n_1 + \cdots + n_{N-1}\)

- **Load factor**: Average value of \(n_j\):
  \(E[n_j] = \alpha = n/N\)
Load Factor of a Hash Table

- Load factor of a hash table $T$: 
  $$\alpha = \frac{n}{N}$$ 
  
  - $n$ = # of elements stored in the table 
  - $N$ = # of slots in the table = # of linked lists 
- $\alpha$ encodes the average number of elements stored in a chain 
- $\alpha$ can be $<, =, > 1$
Case 1: Unsuccessful Search  
(i.e., item not stored in the table)

**Theorem** An unsuccessful search in a hash table takes expected time  
\( \Theta(1 + \alpha) \) under the assumption of simple uniform hashing  
(i.e., probability of collision \( \Pr(h(x) = h(y)) \), is \( 1/N \))

**Proof**
- Searching unsuccessfully for any key \( k \)
  - need to search to the end of the list \( T[h(k)] \)
- Expected length of the list: \( E[n_{h(k)}] = \alpha = n/N \)
- Expected number of elements examined in this case is \( \alpha \)
- Total time required is:
  - \( O(1) \) (for computing the hash function) + \( \alpha \)  \( \Rightarrow \)  \( \Theta(1 + \alpha) \)
Case 2: Successful Search

Successful search: $\Theta(1 + \frac{a}{2}) = \Theta(1 + a)$ time on the average

(search half of a list of length $a$ plus $O(1)$ time to compute $h(k)$)
Analysis of Search in Hash Tables

- If \( N \) (\# of slots) is proportional to \( n \) (\# of elements in the table):
  - \( n = \Theta(N) \)
  - \( \alpha = n/N = \Theta(N)/N = O(1) \)

\[ \Rightarrow \text{Searching takes constant time on average} \]
Open Addressing

- If we have enough contiguous memory to store all the keys ⇒ store the keys in the table itself
- No need to use linked lists anymore
- Basic idea:
  - **put**: if a slot is full, try another one, until you find an empty one
  - **get**: follow the same sequence of probes
  - **remove**: more difficult ... (we’ll see why)
- Search time depends on the length of the probe sequence!

\[ h(k) = k \mod 13 \]  

e.g., insert 14
Generalize hash function notation:

- A hash function contains two arguments now: (i) Key value, and (ii) Probe number
  
  \[ h(k, p), \quad p=0,1,...,N-1 \]

- Probe sequences
  
  \[ <h(k,0), h(k,1), ..., h(k,N-1)> \]

  - Must be a permutation of \(<0,1,...,N-1>\)
  - There are \(N!\) possible permutations
  - Good hash functions should be able to produce all \(N!\) probe sequences

Example:

\[ <1, 5, 9> \]
Common Open Addressing Methods

- Linear probing
- Quadratic probing
- Double hashing

Note: None of these methods can generate more than $N^2$ different probing sequences!
Linear probing

- Idea: when there is a collision, check the next available position in the table (i.e., probing)
  
  \[ h(k,i) = (h_1(k) + a*i) \mod N \]
  
  \[ i=0,1,2,... \]

- First slot probed: \( h_1(k) \)
- Second slot probed: \( h_1(k) + 1 \) (\( a = 1 \))
- Third slot probed: \( h_1(k)+2 \), and so on

- Can generate \( N \) probe sequences maximum, why?
  
  probe sequence: \(< h1(k), h1(k)+1 , h1(k)+2 , ....>\)
Linear probing: Searching for a key

- **Three cases:**
  1. Position in table is occupied with an element of equal key
  2. Position in table is empty
  3. Position in table occupied with a different element

- **Case 3:** probe the next index until the element is found or an empty position is found

- The process wraps around to the beginning of the table

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>h(k₁)</th>
<th>h(k₄)</th>
<th>h(k₂) = h(k₅)</th>
<th>h(k₃)</th>
<th>N - 1</th>
</tr>
</thead>
</table>
Search with Linear Probing

- Consider a hash table $A$ that uses linear probing
- $\text{get}(k)$
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following occurs
    - An item with key $k$ is found, or
    - An empty cell is found, or
    - $N$ cells have been unsuccessfully probed

Algorithm $\text{get}(k)$

\[
i \leftarrow h(k) \\
p \leftarrow 0 \\
\text{repeat} \\
c \leftarrow A[i] \\
\text{if } c = \emptyset \\
\quad \text{return null} \\
\text{else if } c\.getKey() = k \\
\quad \text{return } c\.getValue() \\
\text{else} \\
\quad i \leftarrow (i + 1) \bmod N \\
\quad p \leftarrow p + 1 \\
\text{until } p = N \\
\text{return null}
\]
Quadratic Probing

\[ h(k, i) = (h_1(k) + i^2) \mod N \]

- Probe sequence:
  
  0\textsuperscript{th} probe = \( h(k) \mod N \)
  
  1\textsuperscript{st} probe = \( (h(k) + 1) \mod N \)
  
  2\textsuperscript{nd} probe = \( (h(k) + 4) \mod N \)
  
  3\textsuperscript{rd} probe = \( (h(k) + 9) \mod N \)
  
  \ldots
  
  i\textsuperscript{th} probe = \( (h(k) + i^2) \mod N \)

Less likely to encounter Primary Clustering
Quadratic Probing Example

- **insert(76)**
  - $76 \mod 7 = 6$
  - Insert at index 6

- **insert(40)**
  - $40 \mod 7 = 5$
  - Insert at index 5

- **insert(48)**
  - $48 \mod 7 = 6$
  - Insert at index 6

- **insert(5)**
  - $5 \mod 7 = 5$
  - Insert at index 5

- **insert(55)**
  - $55 \mod 7 = 6$
  - Insert at index 6

But...

- **insert(47)**
  - $47 \mod 7 = 5$
  - Insert at index 5
Quadratic Probing:
Success guarantee for $\alpha < \frac{1}{2}$

- If $N$ is prime and $\alpha < \frac{1}{2}$, then quadratic probing will find an empty slot in $N/2$ probes or fewer, because each probe checks a different slot.
  - Show for all $0 \leq i, j \leq N/2$ and $i \neq j$
    
    $$(h(x) + i^2) \mod N \neq (h(x) + j^2) \mod N$$

  - By contradiction: suppose that for some $i \neq j$:
    
    $$(h(x) + i^2) \mod N = (h(x) + j^2) \mod N$$
    
    $$i^2 \mod N = j^2 \mod N$$
    
    $$\implies (i^2 - j^2) \mod N = 0$$
    
    $$\implies [(i + j)(i - j)] \mod N = 0$$

Because $N$ is prime, $(i-j)$ or $(i+j)$ must be zero, and neither can be, a contradiction.

**Conclusion:** For any $\alpha < \frac{1}{2}$, quadratic probing will find an empty slot; for bigger $\alpha$, quadratic probing may find a slot
Double Hashing

(1) Use one hash function to determine the first slot
(2) Use a second hash function to determine the increment for the probe sequence

\[ h(k,i) = (h_1(k) + i h_2(k)) \mod N, \quad i=0,1,\ldots \]

- Initial probe: \( h_1(k) \)
- Second probe is offset by \( h_2(k) \mod N \), so on ...
- **Advantage**: avoids clustering
- **Disadvantage**: harder to delete an element
- Can generate \( N^2 \) probe sequences maximum
Double Hashing: Example

\[ h_1(k) = k \mod 13 \]
\[ h_2(k) = 1 + (k \mod 11) \]
\[ h(k, i) = (h_1(k) + i h_2(k)) \mod 13 \]

- Insert key 14:
  \[ h_1(14, 0) = 14 \mod 13 = 1 \]
  \[ h(14, 1) = (h_1(14) + h_2(14)) \mod 13 \]
  \[ = (1 + 4) \mod 13 = 5 \]
  \[ h(14, 2) = (h_1(14) + 2 h_2(14)) \mod 13 \]
  \[ = (1 + 8) \mod 13 = 9 \]
Analysis of Open Addressing

- Ignore the problem of clustering and assume that all probe sequences are equally likely

Unsuccessful retrieval:

\[
\text{Prob(probe hits an occupied cell)} = a \quad \text{(load factor)}
\]

\[
\text{Prob(probe hits an empty cell)} = 1 - a
\]

probability that a probe terminates in 2 steps: \(a(1 - a)\)

probability that a probe terminates in \(k\) steps: \(a^{k-1}(1 - a)\)

What is the average number of steps in a probe?

\[
E(\#\text{steps}) = \sum_{k=1}^{m} ka^{k-1}(1 - a) \leq \sum_{k=0}^{\infty} ka^{k-1}(1 - a) = (1 - a) \frac{1}{(1 - a)^2} = \frac{1}{1 - a}
\]
Rehashing

**Idea:** When the table gets too full, create a bigger table (usually 2x as large) and hash all the items from the original table into the new table.

- **When to rehash?**
  - half full ($\alpha = 0.5$)
  - when an insertion fails
  - some other threshold

- **Cost of rehashing?**
Cuckoo hashing

Idea: The hash function provides *two* possible locations.
Cuckoo hashing insertion

New information is inserted by, if necessary, kicking out old information.

0603751133 Harry
0601739822 Pia
0601758473 Mikael
0602760666 Petra
0602728741 Benno

0602754287 Holger
0602761985 Jens
0602754811 Børge
0602753211 Bengt
0602738432 Alice
The Power of Two Choices

- In the cuckoo hashing scheme, we use two lookup tables, $T_0$ and $T_1$, each of size $N$, where $N$ is greater than $n$, the number of items in the map, by at least a constant factor, say, $N \geq 2n$.
- We use a hash function, $h_0$, for $T_0$, and a different hash function, $h_1$, for $T_1$.
- For any key, $k$, there are only two possible places where we are allowed to store an item with key $k$, namely, either in $T_0[h_0(k)]$ or $T_1[h_1(k)]$.
- This freedom turns out to allow us to achieve $O(1)$ worst-case running time for table lookups.
An Example of Cuckoo Hashing

- Each of the keys in the set \( S = \{2, 3, 5, 8, 9\} \) has two possible locations it can go, one in the table \( T_1 \) and one in the table \( T_2 \).
- Note that there is a collision of 2 and 8 in \( T_2 \), but that is okay, since there is no collision for 2 in its alternative location, in \( T_1 \).
Pseudo-code for get and remove

- The algorithms for get and remove are simple and run in $O(1)$ time in the worst case.

- get$(k)$:
  
  ```
  if $T_0[h_0(k)] \neq \text{NULL}$ and $T_0[h_0(k)].key = k$ then
    return $T_0[h_0(k)]$
  if $T_1[h_1(k)] \neq \text{NULL}$ and $T_1[h_1(k)].key = k$ then
    return $T_1[h_1(k)]$
  return NULL
  ```

- remove$(k)$:
  
  ```
  if $T_0[h_0(k)] \neq \text{NULL}$ and $T_0[h_0(k)].key = k$ then
    temp $\leftarrow$ $T_0[h_0(k)]$
    $T_0[h_0(k)] \leftarrow \text{NULL}$
    return temp
  if $T_1[h_1(k)] \neq \text{NULL}$ and $T_1[h_1(k)].key = k$ then
    temp $\leftarrow$ $T_1[h_1(k)]$
    $T_1[h_1(k)] \leftarrow \text{NULL}$
    return temp
  return NULL
  ```
Intuition for the Name

- The name “cuckoo hashing” comes from the way the put(k, v) operation is performed in this scheme, because it mimics the breeding habits of the Common Cuckoo bird.

- The Common Cuckoo is a brood parasite—it lays its egg in the nest of another bird after first evicting an egg out of that nest.
Intuition

- The name “cuckoo hashing” comes from the way the put(k, v) operation is performed in this scheme, as it mimics the breeding habits of the Common Cuckoo bird.
- The Common Cuckoo is a brood parasite, laying its egg in the nest of another bird after evicting an egg out of that nest.
Cuckoo Hashing Examples

Arrows indicate alternative positions
Cuckoo Hashing Examples

Both positions for F are available.
Cuckoo Hashing Examples
Cuckoo Hashing Examples

Both positions for G are unavailable.

If G evicts A, then A will evict E, and E changes position.
Cuckoo Hashing Examples

Both positions for G are unavailable.

If G evicts A, then A will evict E, and E changes position.
Both positions for G are unavailable.

If G evicts B, then B evict D, and D evicts B, and loops.
If G evicts F, then F evicts C; C evicts E; E evicts A; A evicts F, ...
The insertion of G will fail.
Put in Cuckoo Hashing

- If a collision occurs in the insertion operation in the cuckoo hashing scheme, then we evict the previous item in that cell and insert the new one in its place.
- This forces the evicted item to go to its alternate location in the other table and be inserted there, which may repeat the eviction process with another item, and so on.
- Eventually, we either find an empty cell and stop or we repeat a previous eviction, which indicates an eviction cycle.
- If we discover an eviction cycle, then we stop the insertion process and rehash all the items in the two tables using new, hopefully better, hash functions.
Pseudo-code for put

- put\((k, v)\):
  
  ```
  if T_0[h_0(k)] \neq \text{NULL} \text{ and } T_0[h_0(k)].key = k then
    T_0[h_0(k)] \leftarrow (k, v)
  return
  if T_1[h_1(k)] \neq \text{NULL} \text{ and } T_1[h_1(k)].key = k then
    T_1[h_1(k)] \leftarrow (k, v)
  return
  i \leftarrow 0
  repeat
    if T_i[h_i(k)] = \text{NULL} then
      T_i[h_i(k)] \leftarrow (k, v)
    return
    temp \leftarrow T_i[h_i(k)]
    T_i[h_i(k)] \leftarrow (k, v)  \quad  // \text{cuckoo eviction}
    (k, v) \leftarrow \text{temp}
    i \leftarrow (i + 1) \mod 2
  until \text{a cycle occurs}
  ```

- Note that the above pseudo-code has a condition for detecting a cycle in the sequence of insertions. There are several ways to formulate this condition. For example, we could count the number of iterations for this loop and consider there to be a cycle if we go over a certain threshold, like \(n\) or \(\log n\).
Good Properties of Cuckoo Hashing

- \textit{Worst case constant lookup time.}
- \textit{Simple to build, design.}
Cuckoo Hashing Failures

- Bad case 1: inserted element runs into cycles.
- Bad case 2: inserted element has very long path before insertion completes.
  - Could be on a long cycle.
- Bad cases occur with very small probability when load is sufficiently low, say 1/2.
- Standard solution: re-hash everything with new hashing functions if a failure occurs.
Performance of Cuckoo Hashing

- We can also show that, since long eviction sequences are rare, we will have to rehash only with very low probability.
- Thus, the expected amortized time to perform any single insertion in a cuckoo table is $O(1)$.
- So a cuckoo hash table achieves worst-case $O(1)$ time for lookups and removals, and expected $O(1)$ time for insertions.
- This performance is primarily because there are exactly two possible places for any item to be in a cuckoo hash table, which shows the power of two choices.