Recap: Fields

A **Field** is a set of elements $F$ with **two** binary operators $\ast$ and $+$ such that

1. $(F, +)$ is an **abelian group**
2. $(F \setminus \{1\}, \ast)$ is an **abelian group**
   the “multiplicative group”
3. **Distribution**: $a \ast (b+c) = a \ast b + a \ast c$
4. **Cancellation**: $a \ast 1 = 1$

The **order (or size)** of a field is the number of elements. A field of finite order is a **finite field**.

Simple examples to keep mind:
- Binary field $= \{0,1\}$ with binary $+$ and $\ast$
- $\mathbb{Z}_p$ (p prime) with $+$ and $\ast \mod p$
Some constructions: 2-wise independent

Construction 3 (Using finite fields)
Consider $\text{GF}(2^u)$

Pick two random numbers $a, b \in \text{GF}(2^u)$. For any $x \in U$, define
$h(x) := ax + b$
where the calculations are done over the field $\text{GF}(2^u)$.

Q: What is the domain and range of this mapping?
$[U]$ to $[U]$

Q: Is it 2-wise independent?
Yes (write as a matrix and invert) <board>
Some constructions: 2-wise independent

Construction 3 (Using finite fields)
Consider $\text{GF}(2^u)$. Pick two random numbers $a, b \in \text{GF}(2^u)$. For any $x \in U$, define $h(x) := ax + b$ where the calculations are done over the field $\text{GF}(2^u)$.

Q: What is the domain and range of this mapping?
$[U]$ to $[U]$

Q: Is it 2-wise independent?
Yes

Q: How change the range to $[M]$?
Truncate last $u=m$ bits. Still is 2-wise independent.
Some constructions: k-wise independent

Construction 4 (k-wise independence using finite fields):

Q: Any ideas based on the previous construction?
Hint: Going to higher degree polynomial instead of linear.

Consider GF(2^u).
Pick k random numbers $a_0, a_1, \ldots, a_{k-1} \in GF(2^u)$

$$h(x) = a_0 + a_1 x + \ldots + a_{k-1} x^{k-1}$$

where the calculations are done over the field GF(2^u).

Similar proof as before.
Other hashing schemes with good properties

Simple Tabulation Hashing:
Initialize a 2-dimensional \( u \times k \) array \( T \) with each of the \( u \times k \) entries having a random \( m \)-bit string. For the key \( x = x_1 x_2 \ldots x_u \), define its hash as
\[
h(x) := T[1, x_1] \oplus T[2, x_2] \oplus \ldots \oplus T[u, x_u].
\]
Q: How many random bits?
\( u k m \)
Q: Size of the hash family?
\( 2^{ukm} \)

Theorem. Tabulation hashing is 3-wise independent but not 4-wise independent.
15-750: Graduate Algorithms

Hashing:
Hash function basics and some constructions
Hash tables:
  Separate chaining
  Open addressing
Application: Other approaches to collision handling

Open addressing:
  No separate structures
  All keys stored in a single array

Linear probing:
  When inserting x and h(x) is occupied, look for the smallest index i such that (h(x) + 1) mod M is free, and store h(x) there.
  When querying for q, look at h(q) and scan linearly until you find q or an empty space.
Application: Other approaches to collision handling

Linear probing (cont.):

- Deletions are not quite as simple any more.
- It is known that linear probing can also be done in expected constant time, but universal hashing does not suffice to prove this bound: 5-wise independent hashing is necessary [PT10] and sufficient [PPR11].

Other probe sequences:

Using a step-size

Quadratic probing

[Mihai Patrascu and Mikkel Thorup, 2010]
[Anna Pagh, Rasmus Pagh, and Milan Ruzic, 2011]
15-750: Graduate Algorithms

Hashing:
Hash function basics and some constructions
Hash tables:
  Separate chaining
  Open addressing
  Cuckoo hashing
Bloom filters
Application: Cuckoo hashing


Take two tables $T_1$ and $T_2$, both of size $M = O(N)$.

Take two hash functions $h_1, h_2: U \rightarrow [M]$ from hash family $H$. Let $H$ be fully-random ($O(\log N)$-wise independence suffices).
Cuckoo hashing

Insertion:
When an element x is inserted, if either $T_1[h_1(x)]$ or $T_2[h_2(x)]$ is empty, put the element x in that location.

If not bump out the element (say y) in either of these locations and put x in.

When an element gets bumped out, place it in the other possible location. If that is empty then done. If not, bump the element in that location and place y there.

If more than $6 \times \log N$ bumps occur then rehash everything by picking a new pair of hash functions.

Query/delete:
An element x will be either in $T_1[h_1(x)]$ or $T_2[h_2(x)]$.

O(1) operations
Cuckoo hashing

**Theorem.** The expected time to perform an insert operation is $O(1)$ if $M \geq 4N$.

**Proof sketch.**

Assume completely random hash functions (ideal).

For analysis we will use “cuckoo graph” $G$

- M vertices corresponding to hashtable locations
- Edges correspond to the items to be inserted.
  - For all $x$ in $S$, $e_x = (h1(x), h2(x))$ will be in the edge set
Cuckoo hashing

Proof sketch continued.

Q: When can element $y$ get bumped out from it location when inserting a new element $x$?

A: When $y$ falls in a path in cuckoo graph starting from $h_1(x)$ or $h_2(x)$

Define: Bucket of $x$, $B(x) =$ set of nodes of $G$ reachable from $h_1(x)$ or $h_2(x)$

- Connected component of $G$ with edge $e_x$
Cuckoo hashing

Proof sketch (cont.):

\( E[\text{Insertion time for } x] = E[|B(x)|] \)

Goal: To show \( E[|B(x)|] \leq O(1) \)
Cuckoo hashing

Proof sketch (cont.):
Goal: To show $E[|B(x)|] \leq O(1)$

\[
E[|B(x)|] = \sum_{y \in S, y \neq x} P \left[ e_y \in B(x) \right] \\
\leq N \cdot P \left[ e_y \in B(x) \right]
\]

Sufficient to show

\[
P \left[ e_y \in B(x) \right] \leq O \left( \frac{1}{m} \right)
\]
Cuckoo hashing

Proof sketch (cont.):
Goal: To show \( P \left[ e_y \in B(z) \right] \leq O \left( \frac{1}{M} \right) \)

Lemma. For any \( i, j \) in \([M]\),
\( P[\text{there exists a path of length } \ell \text{ between } i \text{ and } j \text{ in the cuckoo graph}] \leq \frac{1}{2^\ell M} \)

Proof. For \( \ell = 1 \), \( P[\text{edge } i \text{ between } j] \)

\[ = P \left[ \exists y \text{ s.t. } e_y \text{ exist in } E \right] \leq N \cdot \frac{2}{M^2} \leq \frac{1}{2} \cdot \frac{1}{M} \]

Then induction on \( \ell \).
(Exercise)
Cuckoo hashing

Proof sketch (cont.):
Goal: To show \( P \left[ e_y \in B(x) \right] \leq O \left( \frac{1}{m} \right) \)

Proof. Using the Lemma,

\[
P \left[ e_y \in B(x) \right] \leq \sum_{k=1}^{\infty} \frac{1}{2^k m} = O \left( \frac{1}{m} \right)
\]

- This proof for Cuckoo hashing is by Rasmus Pagh and a very nice explanation of this proof can be found at: http://www.cs.toronto.edu/~wgeorge/csc265/2013/10/17/tutorial-5-cuckoo-hashing.html
- A different proof can be found at: http://courses.csail.mit.edu/6.851/spring12/SCRIBE/lec10.pdf
Cuckoo hashing: space efficiency

One of the key metrics for hash tables is the space efficiency, measured by “occupancy rate”. Corresponds to the space overhead needed

With $M \geq 4N$ we have only 25% occupancy!

Can we do better?

Turns out that you can get close to 50% occupancy with good probability of success, but not for better than 50% occupancy

-> If one uses $d$ hash functions instead of 2?

    With $d = 3$, experimentally > 90% occupancy

-> Put more items in a location (say, 2 to 4 items) in each location? Mostly, only experimental conjectures and theory still open.
15-750: Graduate Algorithms

Hashing:
- Hash function basics and some constructions
- Hash tables:
  - Separate chaining
  - Open addressing
  - Cuckoo hashing

Bloom filters

Load balancing (balls and bins)
- Concentration bounds
- Power-of-two choices
Application: Bloom filter

Representing a dictionary with far fewer bits when only need membership query.


• Restricted set of operations:
  • Only membership queries
  • No deletions
• Allow mistakes on membership queries
• Only false positives; no false negatives
  • may report that a key is present when it is not
• Very useful for “filtering out”: scenario where most keys will not belong to the dictionary.
  • E.g: malicious/blocked websites in web browser
• If the answer is “Yes” then you can use a slow data structure
Bloom filter

Space efficient data structure for *approximate* membership queries.

- Keep an array $T$ of $M$ bits
  - initially all entries are zero.
- $k$ hash functions: $h_1, h_2, \ldots, h_k: U \rightarrow [M]$
  - Assume completely random hash functions for analysis

Adding a key:
- To add a key $x \in S \subseteq U$, set bits $T[h_1(x)], T[h_2(x)], \ldots, T[h_k(x)]$ to 1
**Bloom filter**

Membership query:
- For a query for key \( x \in U \): check if all the entries \( T[h_i(x)] \) are set to 1
- If so, answer Yes else answer No.

Q: Why no false negatives?
If an item \( x \) is present, then corresponding bits will be set.

Q: Why false positives?
Other elements could have set the same bits.

Let’s analyze the probability of false positives.