Hash Functions

- Challenge in dictionary implementation: among a large universe U of words, we need to maintain a subset S, with fast Insert, Delete, and Lookup operations (ideally in $O(1)$ time).

Hash Functions

- Challenge in dictionary implementation: among a large universe U of words, we need to maintain a subset S, with fast Insert, Delete, and Lookup operations (ideally in $O(1)$ time).
- Maintaining an array of size $|U|$ is too expensive.

Hash Functions

- Challenge in dictionary implementation: among a large universe U of words, we need to maintain a subset S, with fast Insert, Delete, and Lookup operations (ideally in $O(1)$ time).
- Maintaining an array of size $|U|$ is too expensive.
- Maintaining a linked list of size $|S|$ makes lookup too slow.

Hash Functions

- Challenge in dictionary implementation: among a large universe U of words, we need to maintain a subset S, with fast Insert, Delete, and Lookup operations (ideally in $O(1)$ time).
- Maintaining an array of size $|U|$ is too expensive.
- Maintaining a linked list of size $|S|$ makes lookup too slow.
- *Hash*: Maintain an array of size (roughly) $n = |S|$, and use function $h : U \to \{0, 1, \ldots, n-1\}$. Ideally, we'd like $h(u) \neq h(v)$ whenever $u, v \in S$ are not equal.

Hash Functions

- Challenge in dictionary implementation: among a large universe U of words, we need to maintain a subset S, with fast Insert, Delete, and Lookup operations (ideally in $O(1)$ time).
- Maintaining an array of size $|U|$ is too expensive.
- Maintaining a linked list of size $|S|$ makes lookup too slow.
- *Hash*: Maintain an array of size (roughly) $n = |S|$, and use function $h : U \to \{0, 1, \ldots, n-1\}$. Ideally, we'd like $h(u) \neq h(v)$ whenever $u, v \in S$ are not equal.
- No deterministic function h can work for all inputs.

Hash Functions

- Challenge in dictionary implementation: among a large universe U of words, we need to maintain a subset S, with fast Insert, Delete, and Lookup operations (ideally in $O(1)$ time).
- Maintaining an array of size $|U|$ is too expensive.
- Maintaining a linked list of size $|S|$ makes lookup too slow.
- *Hash*: Maintain an array of size (roughly) $n = |S|$, and use function $h : U \to \{0, 1, \ldots, n-1\}$. Ideally, we'd like $h(u) \neq h(v)$ whenever $u, v \in S$ are not equal.
- No deterministic function h can work for all inputs.
- A completely random mapping has collision rate $\frac{1}{n}$, but memorizing the mapping is exactly the problem we started with!

Universal Hash Functions

- Instead of using one hash function, let's construct a *family* of hash functions, each of them compactly representable and efficiently computable. We then *randomly* one to use.

Universal Hash Functions

- Instead of using one hash function, let's construct a *family* of hash functions, each of them compactly representable and efficiently computable. We then *randomly* one to use.
- A family of hash functions $\mathcal{H}$ is *universal* if for any $u, v \in U$ that are not equal, the probability that a randomly chosen $h \in \mathcal{H}$ with $h(u) = h(v)$ is at most $1/n$.

Universal Hash Functions

- Instead of using one hash function, let's construct a *family* of hash functions, each of them compactly representable and efficiently computable. We then *randomly* one to use.
- A family of hash functions $\mathcal{H}$ is *universal* if for any $u, v \in U$ that are not equal, the probability that a randomly chosen $h \in \mathcal{H}$ with $h(u) = h(v)$ is at most $1/n$.

Lemma

Let $\mathcal{H}$ be a universal class of hash functions mapping a universe U to $\{0, 1, \ldots, n-1\}$, let $S \subseteq U$ be of size at most n, and let u be any element in U. Define X to be a random variable equal to the numbe rof elements $s \in S$ for which $h(s) = h(u)$, for a random choice of $h \in \mathcal{H}$. Then $\mathbf{E}[X] \leq 1$.

An implementation of universal hash functions

- Let p be a prime number approximately equal to n (e.g., $n \leq p \leq 2n$.

An implementation of universal hash functions

- Let p be a prime number approximately equal to n (e.g., $n \leq p \leq 2n$.
- Represent each element of U by a sequence $x_1 x_2 \ldots x_r$, where x_i is an integer from $\{0, 1, \ldots, p-1\}$.

An implementation of universal hash functions

- Let p be a prime number approximately equal to n (e.g., $n \leq p \leq 2n$.
- Represent each element of U by a sequence $x_1 x_2 \ldots x_r$, where x_i is an integer from $\{0, 1, \ldots, p-1\}$.
- Let $\mathcal{A}$ be $\{0, 1, \ldots, p-1\}^r$. For each $\mathbf{a} \in \mathcal{A}$, define

$$h_{\mathbf{a}}(\mathbf{x}) = \left(\sum_i a_i x_i\right) \mod p, \quad \mathbf{x} \in U.$$

An implementation of universal hash functions

- Let p be a prime number approximately equal to n (e.g., $n \leq p \leq 2n$.
- Represent each element of U by a sequence $x_1 x_2 \ldots x_r$, where x_i is an integer from $\{0, 1, \ldots, p-1\}$.
- Let $\mathcal{A}$ be $\{0, 1, \ldots, p-1\}^r$. For each $\mathbf{a} \in \mathcal{A}$, define

$$h_{\mathbf{a}}(\mathbf{x}) = \left(\sum_i a_i x_i\right) \mod p, \quad \mathbf{x} \in U.$$

Theorem

The family $\{h_{\mathbf{a}} : \mathbf{a} \in \mathcal{A}\}$ is universal family of hash functions.

An implementation of universal hash functions

- Let p be a prime number approximately equal to n (e.g., $n \le p \le 2n$.
- Represent each element of U by a sequence $x_1 x_2 \dots x_r$, where x_i is an integer from $\{0, 1, \dots, p-1\}$.
- Let $\mathcal{A}$ be $\{0, 1, \dots, p-1\}^r$. For each $\mathbf{a} \in \mathcal{A}$, define

$$h_{\mathbf{a}}(\mathbf{x}) = \left(\sum_i a_i x_i \right) \mod p, \quad \mathbf{x} \in U.$$

Theorem

The family $\{h_{\mathbf{a}} : \mathbf{a} \in \mathcal{A}\}$ is universal family of hash functions.

Lemma

For any prime p and any integer $z \neq 0 \mod p$, and any two integers α, β, if $\alpha z = \beta z \mod p$, then $\alpha = \beta \mod p$.

k-wise Independent hash functions

Definition

A family $\mathcal{H}$ of hash functions is k-wise independent if, for any $u_1, u_2, \ldots, u_k \in U$, and any hash values $z_1, \ldots, z_k \in \{0, 1, \ldots, n-1\}$, we have

$$\mathbf{Pr}_{h \sim \mathcal{H}}[h(u_1) = z_1, \ldots, h(u_k) = z_k] = \frac{1}{n^k}.$$

k-wise Independent hash functions

Definition

A family $\mathcal{H}$ of hash functions is k-wise independent if, for any $u_1, u_2, \ldots, u_k \in U$, and any hash values $z_1, \ldots, z_k \in \{0, 1, \ldots, n-1\}$, we have

$$\mathbf{Pr}_{h \sim \mathcal{H}}\left[h(u_1) = z_1, \ldots, h(u_k) = z_k\right] = \frac{1}{n^k}.$$

- Depending on the context, sometimes we only require the bound to be $O(\frac{1}{n^k})$.

Definition

A family $\mathcal{H}$ of hash functions is k-wise independent if, for any $u_1, u_2, \ldots, u_k \in U$, and any hash values $z_1, \ldots, z_k \in \{0, 1, \ldots, n-1\}$, we have

$$\mathbf{Pr}_{h \sim \mathcal{H}}\left[h(u_1) = z_1, \ldots, h(u_k) = z_k\right] = \frac{1}{n^k}.$$

- Depending on the context, sometimes we only require the bound to be $O(\frac{1}{n^k})$.
- Pairwise independence is sometimes known as *strong universality*.

k-wise Independent hash functions

Definition

A family $\mathcal{H}$ of hash functions is k-wise independent if, for any $u_1, u_2, \ldots, u_k \in U$, and any hash values $z_1, \ldots, z_k \in \{0, 1, \ldots, n-1\}$, we have

$$\mathbf{Pr}_{h \sim \mathcal{H}}[h(u_1) = z_1, \ldots, h(u_k) = z_k] = \frac{1}{n^k}.$$

- Depending on the context, sometimes we only require the bound to be $O(\frac{1}{n^k})$.
- Pairwise independence is sometimes known as *strong universality*.

Bloom filters

- Scenario: web caching.

Bloom filters

- Scenario: web caching.
- Operations we'd like to support:
  - Insert
  - Lookup: One-sided fault tolerant: if the page is in the cache, we should answer "yes"; if the page is in the cache, we should answer "no" with good probability.

Bloom filters

- Scenario: web caching.
- Operations we'd like to support:
  - Insert
  - Lookup: One-sided fault tolerant: if the page is in the cache, we should answer "yes"; if the page is in the cache, we should answer "no" with good probability.
  - (This is not an operation but a desired property): the data structure should be (much) smaller than the dictionary so we can easily send it around.

Bloom filters

- Scenario: web caching.
- Operations we'd like to support:
  - Insert
  - Lookup: One-sided fault tolerant: if the page is in the cache, we should answer "yes"; if the page is in the cache, we should answer "no" with good probability.
  - (This is not an operation but a desired property): the data structure should be (much) smaller than the dictionary so we can easily send it around.
- Operations we do not need to support:
  - Retrieve the entry associated with a given key value.

Bloom filters

- Scenario: web caching.
- Operations we'd like to support:
  - Insert
  - Lookup: One-sided fault tolerant: if the page is in the cache, we should answer "yes"; if the page is in the cache, we should answer "no" with good probability.
  - (This is not an operation but a desired property): the data structure should be (much) smaller than the dictionary so we can easily send it around.
- Operations we do not need to support:
  - Retrieve the entry associated with a given key value.
- Solution: Bloom filter (Burton Bloom, 1970)

Bloom filters

- Scenario: web caching.
- Operations we'd like to support:
  - Insert
  - Lookup: One-sided fault tolerant: if the page is in the cache, we should answer "yes"; if the page is in the cache, we should answer "no" with good probability.
  - (This is not an operation but a desired property): the data structure should be (much) smaller than the dictionary so we can easily send it around.
- Operations we do not need to support:
  - Retrieve the entry associated with a given key value.
- Solution: Bloom filter (Burton Bloom, 1970)
- Another application: web browser cache for email contact list

Bloom Filter: Implementation

- Set up a bit vector B of length m (we will decide m later; think of it as $O(n)$); let $h_1, \ldots, h_k$ be hash functions, which we think of as independent random functions mapping U to $\{0, 1, \ldots, m\}$.

Bloom Filter: Implementation

- Set up a bit vector B of length m (we will decide m later; think of it as $O(n)$); let $h_1, \ldots, h_k$ be hash functions, which we think of as independent random functions mapping U to $\{0, 1, \ldots, m\}$.
- Insert $u \in U$: set $B[h_1(u)] = B[h_2(u)] = \cdots = B[h_k(u)] = 1$.

Bloom Filter: Implementation

- Set up a bit vector B of length m (we will decide m later; think of it as $O(n)$); let $h_1, \ldots, h_k$ be hash functions, which we think of as independent random functions mapping U to $\{0, 1, \ldots, m\}$.
- Insert $u \in U$: set $B[h_1(u)] = B[h_2(u)] = \cdots = B[h_k(u)] = 1$.
- Query if u is in the set: If $B[h_1(u)] = \cdots = B[h_k(u)] = 1$, answer "yes" else answer "no"

Bloom Filter: Implementation

- Set up a bit vector B of length m (we will decide m later; think of it as $O(n)$); let $h_1, \ldots, h_k$ be hash functions, which we think of as independent random functions mapping U to $\{0, 1, \ldots, m\}$.
- Insert $u \in U$: set $B[h_1(u)] = B[h_2(u)] = \cdots = B[h_k(u)] = 1$.
- Query if u is in the set: If $B[h_1(u)] = \cdots = B[h_k(u)] = 1$, answer "yes" else answer "no"

- If u is in the set, we obviously always answer "yes"

Bloom Filter: Implementation

- Set up a bit vector B of length m (we will decide m later; think of it as $O(n)$); let $h_1, \ldots, h_k$ be hash functions, which we think of as independent random functions mapping U to $\{0, 1, \ldots, m\}$.
- Insert $u \in U$: set $B[h_1(u)] = B[h_2(u)] = \cdots = B[h_k(u)] = 1$.
- Query if u is in the set: If $B[h_1(u)] = \cdots = B[h_k(u)] = 1$, answer "yes" else answer "no"

- If u is in the set, we obviously always answer "yes"
- If u is not in the set, what is the probability we answer "yes"?

Bloom Filter: Analysis

- If u is not in the set, what is the probability we answer "yes"?

Bloom Filter: Analysis

- If u is not in the set, what is the probability we answer "yes"?

We need to bound the probability of the following "bad event": given a word $u \notin S$, $h_1(u) = h_{j_1}(u_1), \cdots, h_k(u) = h_{j_k}(u_k)$ for $u_1, \ldots, u_k \in S$, and $j_1, \ldots, j_k \in \{1, 2, \cdots, k\}$.

Bloom Filter: Analysis

- If u is not in the set, what is the probability we answer "yes"?

We need to bound the probability of the following “bad event”: given a word $u \notin S$, $h_1(u) = h_{j_1}(u_1), \cdots, h_k(u) = h_{j_k}(u_k)$ for $u_1, \ldots, u_k \in S$, and $j_1, \ldots, j_k \in \{1, 2, \cdots, k\}$.
Let E_i denote the event that there eixts $u_i \in S$ and $j_i \in \{1, \cdots, k\}$ such that $h_i(u) = h_{j_i}(u_i)$. Then we need to bound $\mathbf{Pr}[E_1 \cap E_2 \cdots \cap E_k]$.

Bloom Filter: Analysis

- If u is not in the set, what is the probability we answer "yes"?

We need to bound the probability of the following "bad event": given a word $u \notin S$, $h_1(u) = h_{j_1}(u_1), \cdots, h_k(u) = h_{j_k}(u_k)$ for $u_1, \ldots, u_k \in S$, and $j_1, \ldots, j_k \in \{1, 2, \cdots, k\}$.
Let E_i denote the event that there eixts $u_i \in S$ and $j_i \in \{1, \cdots, k\}$ such that $h_i(u) = h_{j_i}(u_i)$. Then we need to bound $\mathbf{Pr}[E_1 \cap E_2 \cdots \cap E_k]$.
For a particular hash value $z \in \{0, 1, \ldots, m\}$, and any word u in our set S, and a hash function h_i,

$$\mathbf{Pr}\left[h_i(u) \neq z\right] = 1 - \frac{1}{m};$$

Bloom Filter: Analysis

- If u is not in the set, what is the probability we answer "yes"?

We need to bound the probability of the following "bad event": given a word $u \notin S$, $h_1(u) = h_{j_1}(u_1), \cdots, h_k(u) = h_{j_k}(u_k)$ for $u_1, \ldots, u_k \in S$, and $j_1, \ldots, j_k \in \{1, 2, \cdots, k\}$.
Let E_i denote the event that there eixts $u_i \in S$ and $j_i \in \{1, \cdots, k\}$ such that $h_i(u) = h_{j_i}(u_i)$. Then we need to bound $\mathbf{Pr}[E_1 \cap E_2 \cdots \cap E_k]$.
For a particular hash value $z \in \{0, 1, \ldots, m\}$, and any word u in our set S, and a hash function h_i,

$$\mathbf{Pr}\left[h_i(u) \neq z\right] = 1 - \frac{1}{m};$$

For $z \in \{0, 1, \ldots, m\}$, the probability that none of the words in S are mapped to z by any of the k hash functions is

$$\left(1 - \frac{1}{m}\right)^{kn}.$$

Bloom Filter: Analysis continued

- If u is not in the set, what is the probability we answer "yes"?

Let E_i denote the event that there eixts $u_i \in S$ and $j_i \in \{1, \cdots, k\}$ such that $h_i(u) = h_{j_i}(u_i)$. Then we need to bound $\mathbf{Pr}[E_1 \cap E_2 \cdots \cap E_k]$.

Bloom Filter: Analysis continued

- If u is not in the set, what is the probability we answer "yes"?

Let E_i denote the event that there eixts $u_i \in S$ and $j_i \in \{1, \cdots, k\}$ such that $h_i(u) = h_{j_i}(u_i)$. Then we need to bound $\mathbf{Pr}[E_1 \cap E_2 \cdots \cap E_k]$. Therefore

$$\mathbf{Pr}\left[E_i\right] = 1 - \left(1 - \frac{1}{m}\right)^{kn}$$

Bloom Filter: Analysis continued

- If u is not in the set, what is the probability we answer "yes"?

Let E_i denote the event that there eixts $u_i \in S$ and $j_i \in \{1, \cdots, k\}$ such that $h_i(u) = h_{j_i}(u_i)$. Then we need to bound $\mathbf{Pr}[E_1 \cap E_2 \cdots \cap E_k]$. Therefore

$$\mathbf{Pr}\left[E_i\right] = 1 - \left(1 - \frac{1}{m}\right)^{kn}$$

Since the events are negatively correlated,

$$\mathbf{Pr}\left[\cap_i E_i\right] \leq \left[1 - \left(1 - \frac{1}{m}\right)^{kn}\right]^k \approx \left(1 - e^{-kn/m}\right)^k.$$

Bloom Filter: Analysis continued

- If u is not in the set, what is the probability we answer "yes"?

Let E_i denote the event that there eixts $u_i \in S$ and $j_i \in \{1, \cdots, k\}$ such that $h_i(u) = h_{j_i}(u_i)$. Then we need to bound $\mathbf{Pr}[E_1 \cap E_2 \cdots \cap E_k]$. Therefore

$$\mathbf{Pr}\left[E_i\right] = 1 - \left(1 - \frac{1}{m}\right)^{kn}$$

Since the events are negatively correlated,

$$\mathbf{Pr}\left[\cap_i E_i\right] \leq \left[1 - \left(1 - \frac{1}{m}\right)^{kn}\right]^k \approx \left(1 - e^{-kn/m}\right)^k.$$

This is the probability of a false positive, and we would like to minimize this. Taking the derivative of the logarithm of this quantity, we get $k = \ln 2 \cdot \frac{m}{n}$; when using this value of k, the false positive rate is

$$\left(\frac{1}{2}\right)^k = (0.6185)^{\frac{m}{n}}.$$