Lecture 5
Bloom filters

Markus Jalsenius
Simpler operations

Many dictionaries, including hashing, support the following operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add(x, v)</code></td>
<td>Insert the key <code>x</code> together with the satellite data <code>v</code>, i.e. data associated with <code>x</code>.</td>
</tr>
<tr>
<td><code>lookup(x)</code></td>
<td>Return the value <code>v</code> that was stored with the key <code>x</code>.</td>
</tr>
<tr>
<td><code>delete(x)</code></td>
<td>Remove key <code>x</code>.</td>
</tr>
</tbody>
</table>
Simpler operations

Many dictionaries, including hashing, support the following operations:

- **add**$(x, v)$: Insert the key $x$ together with the satellite data $v$, i.e. data associated with $x$.
- **lookup**$(x)$: Return the value $v$ that was stored with the key $x$.
- **delete**$(x)$: Remove key $x$.

Suppose we do not care about satellite data, and we do not want to remove keys. We only care about membership queries:

- **add**$(x)$: Insert the key $x$.
- **lookup**$(x)$: Return TRUE if $x$ has been inserted, otherwise FALSE.
Simpler operations

Many dictionaries, including hashing, support the following operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add($x, v$)</td>
<td>Insert the key $x$ together with the <em>satellite data</em> $v$, i.e. data associated with $x$.</td>
</tr>
<tr>
<td>lookup($x$)</td>
<td>Return the value $v$ that was stored with the key $x$.</td>
</tr>
<tr>
<td>delete($x$)</td>
<td>Remove key $x$.</td>
</tr>
</tbody>
</table>

Suppose we do not care about satellite data, and we do not want to remove keys. We only care about *membership* queries:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add($x$)</td>
<td>Insert the key $x$.</td>
</tr>
<tr>
<td>lookup($x$)</td>
<td>Return <code>TRUE</code> if $x$ has been inserted, otherwise <code>FALSE</code>.</td>
</tr>
</tbody>
</table>

We can use hashing still, but there are drawbacks:

- The actual keys themselves are stored in the hash table.
- Satellite data is unnecessary overhead.
- The size of a hash table, including linked lists and other overhead, may be relatively large, depending on performance.
Encoding method

**Example**

Universe \( U = \{1, 2, 3, 4\} \)

- Suppose we want to insert \( n = 2 \) keys into the dictionary.
Encoding method

**Example**

Universe $U = \{1, 2, 3, 4\}$

- Suppose we want to insert $n = 2$ keys into the dictionary.
- There are $\binom{4}{2} = 6$ possibilities:

<table>
<thead>
<tr>
<th>Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2</td>
</tr>
<tr>
<td>1, 3</td>
</tr>
<tr>
<td>1, 4</td>
</tr>
<tr>
<td>2, 3</td>
</tr>
<tr>
<td>2, 4</td>
</tr>
<tr>
<td>3, 4</td>
</tr>
</tbody>
</table>
Encoding method

**Example**

Universe $U = \{1, 2, 3, 4\}$

- Suppose we want to insert $n = 2$ keys into the dictionary.
- There are $\binom{4}{2} = 6$ possibilities:

<table>
<thead>
<tr>
<th>Keys</th>
<th>Encoding (3 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2</td>
<td>000</td>
</tr>
<tr>
<td>1, 3</td>
<td>001</td>
</tr>
<tr>
<td>1, 4</td>
<td>010</td>
</tr>
<tr>
<td>2, 3</td>
<td>011</td>
</tr>
<tr>
<td>2, 4</td>
<td>100</td>
</tr>
<tr>
<td>3, 4</td>
<td>101</td>
</tr>
</tbody>
</table>

- The dictionary consists of *only three bits*. If the bits are, say 011, then this means that keys 2 and 3 are in the dictionary.
Encoding method

Let us generalise…

Universe $U$ containing $u$ keys.

- Suppose we want to insert $n$ keys. There are $\binom{u}{n}$ possibilities.
Encoding method

Let us generalise... 

Universe $\mathcal{U}$ containing $u$ keys.

- Suppose we want to insert $n$ keys. There are $\binom{u}{n}$ possibilities.
- We can uniquely encode any set of $n$ keys using $\log_2 \binom{u}{n}$ bits. This is also the number of bits necessary to support lookups of $n$ inserted keys.
Encoding method

Let us generalise...

Universe $U$ containing $u$ keys.

- Suppose we want to insert $n$ keys. There are $\binom{u}{n}$ possibilities.
- We can uniquely encode any set of $n$ keys using $\log_2 \binom{u}{n}$ bits. This is also the number of bits necessary to support lookups of $n$ inserted keys.
- The size of the dictionary is $\log_2 \binom{u}{n}$ bits.
Encoding method

Let us generalise...

Universe $\mathcal{U}$ containing $u$ keys.

- Suppose we want to insert $n$ keys. There are $\binom{u}{n}$ possibilities.
- We can uniquely encode any set of $n$ keys using $\log_2 \binom{u}{n}$ bits. This is also the number of bits necessary to support lookups of $n$ inserted keys.
- The size of the dictionary is $\log_2 \binom{u}{n}$ bits.

**Observe**

It could take a long time to perform a lookup. Why?
Encoding method

Let us generalise…

Universe $U$ containing $u$ keys.

- Suppose we want to insert $n$ keys. There are $\binom{u}{n}$ possibilities.
- We can uniquely encode any set of $n$ keys using $\log_2 \binom{u}{n}$ bits. This is also the number of bits necessary to support lookups of $n$ inserted keys.
- The size of the dictionary is $\log_2 \binom{u}{n}$ bits.

**Observe**

It could take a long time to perform a lookup. Why?

To determine whether key $x$ is in the dictionary we must *decode* the bit string and work out what keys it represents. Although we can accurately do this, it is not obvious how to do it quickly without using a translation table (like in the previous example), which itself uses a lot of space.
Encoding method

How much is $\log_2 \binom{u}{n}$ anyway?
Encoding method

► How much is $\log_2 \binom{u}{n}$ anyway?

► Recall from the first lecture that $\binom{u}{n} \geq \left( \frac{u}{n} \right)^n$. 
Encoding method

How much is \( \log_2 \binom{u}{n} \) anyway?

Recall from the first lecture that \( \binom{u}{n} \geq \left( \frac{u}{n} \right)^n \).

Thus,

\[
\log_2 \binom{u}{n} \geq \log_2 \left( \frac{u}{n} \right)^n = n \log_2 u - n \log_2 n \approx n \log_2 u
\]

when \( u \) is much larger than \( n \).
How much is \( \log_2 \left( \frac{u}{n} \right) \) anyway?

Recall from the first lecture that \( \binom{u}{n} \geq \left( \frac{u}{n} \right)^n \).

Thus,

\[
\log_2 \binom{u}{n} \geq \log_2 \left( \frac{u}{n} \right)^n = n \log_2 u - n \log_2 n \approx n \log_2 u
\]

when \( u \) is much larger than \( n \).

If the universe is very, very large, then even \( \log_2 u \) is relatively big, and we have the problem with the lookup time pointed out previously.
Encoding method

- How much is $\log_2 \left( \binom{u}{n} \right)$ anyway?

- Recall from the first lecture that $\binom{u}{n} \geq \left( \frac{u}{n} \right)^n$.

- Thus,

\[
\log_2 \left( \binom{u}{n} \right) \geq \log_2 \left( \frac{u}{n} \right)^n = n \log_2 u - n \log_2 n \approx n \log_2 u
\]

when $u$ is much larger than $n$.

- If the universe is very, very large, then even $\log_2 u$ is relatively big, and we have the problem with the lookup time pointed out previously.

- We will shortly see how a dictionary of size $c \cdot n$ bits, where $c$ is a constant, can give good performance.
Encoding method

► How much is $\log_2 \left( \binom{u}{n} \right)$ anyway?

► Recall from the first lecture that $\binom{u}{n} \geq \left( \frac{u}{n} \right)^n$.

► Thus,

$$\log_2 \left( \binom{u}{n} \right) \geq \log_2 \left( \frac{u}{n} \right)^n = n \log_2 u - n \log_2 n \approx n \log_2 u$$

when $u$ is much larger than $n$.

► If the universe is very, very large, then even $\log_2 u$ is relatively big, and we have the problem with the lookup time pointed out previously.

► We will shortly see how a dictionary of size $c \cdot n$ bits, where $c$ is a constant, can give good performance.

Observe

Since we need $\log_2 \left( \binom{u}{n} \right)$ bits, a dictionary of size $c \cdot n$, where $c$ is a constant, must make errors with certain probability.
Hashing

As mentioned, there are drawbacks:

- The actual keys themselves are stored in the hash table.
- Satellite data is unnecessary overhead.
- The size of a hash table, including linked lists and other overhead, may be relatively large, depending on performance.
As mentioned, there are drawbacks:
- The actual keys themselves are stored in the hash table.
- Satellite data is unnecessary overhead.
- The size of a hash table, including linked lists and other overhead, may be relatively large, depending on performance.

**Example**

- Suppose the universe $U$ contains $10^9$ strings, each of length $10^5$ bits.
Hashing

As mentioned, there are drawbacks:

- The actual keys themselves are stored in the hash table.
- Satellite data is unnecessary overhead.
- The size of a hash table, including linked lists and other overhead, may be relatively large, depending on performance.

Example

- Suppose the universe $U$ contains $10^9$ strings, each of length $10^5$ bits.
- We want to store $n = 100$ of the strings in a dictionary.
- Using the encoding argument from before means the dictionary must have size around $n \log_2 u \approx 100 \cdot 30 = 3000$ bits.
Hashing

As mentioned, there are drawbacks:

- The actual keys themselves are stored in the hash table.
- Satellite data is unnecessary overhead.
- The size of a hash table, including linked lists and other overhead, may be relatively large, depending on performance.

**Example**

- Suppose the universe $U$ contains $10^9$ strings, each of length $10^5$ bits.
- We want to store $n = 100$ of the strings in a dictionary.
- Using the encoding argument from before means the dictionary must have size around
  \[ n \log_2 u \approx 100 \cdot 30 = 3000 \text{ bits}. \]
- Using a hash table of size $m = n = 100$ say, we still need at least
  \[ 100 \cdot 10^5 = 10^7 \text{ bits} \] as we also store the keys in the hash table.
Hashing

- As mentioned, there are drawbacks:
  - The actual keys themselves are stored in the hash table.
  - Satellite data is unnecessary overhead.
  - The size of a hash table, including linked lists and other overhead, may be relatively large, depending on performance.

**Example**

- Suppose the universe $U$ contains $10^9$ strings, each of length $10^5$ bits.
- We want to store $n = 100$ of the strings in a dictionary.
- Using the encoding argument from before means the dictionary must have size around

  $$n \log_2 u \approx 100 \cdot 30 = 3000$$

  bits.

- Using a hash table of size $m = n = 100$ say, we still need at least

  $$100 \cdot 10^5 = 10^7$$

  bits as we also store the keys in the hash table.

- Thus, hashing can be very costly space-wise.
Succinct data structure

- A **succinct data structure** uses close to minimum number of bits, i.e. it is very space efficient.
- We want to use less than $n \log_2 u$ bits for our dictionary that supports only insertions and lookups.
- Therefore we will introduce errors!
Succinct data structure

- A **succinct data structure** uses close to minimum number of bits, i.e. it is very space efficient.
- We want to use less than $n \log_2 u$ bits for our dictionary that supports only insertions and lookups.
- Therefore we will introduce errors!
- The solution is hashing, but in a slightly different way.
Back to hashing

Universe $U$ containing $u$ keys. Bit string $T$ of size $m$.

A hash function $h : U \rightarrow [m]$ maps a key to a position in $T$. 
Back to hashing

Universe $U$ containing $u$ keys.

Bit string $T$ of size $m$.

A hash function $h : U \rightarrow [m]$ maps a key to a position in $T$.

Initially the bit string $T$ contains only zeros.

- **add($x$)** Set $T[h(x)]$ to 1.
- **lookup($x$)** Return **TRUE** if $T[h(x)] = 1$, otherwise return **FALSE**.
Back to hashing

Universe $U$ containing $n$ keys. Bit string $T$ of size $m$.

- A **hash function** $h : U \rightarrow [m]$ maps a key to a position in $T$.
- Initially the bit string $T$ contains only zeros.
  - **add($x$)** Set $T[h(x)]$ to 1.
  - **lookup($x$)** Return **TRUE** if $T[h(x)] = 1$, otherwise return **FALSE**.

**Observe**

Only false positive errors are possible. **Why?**
Back to hashing

**Example**

- Suppose the hash function $h$ is chosen uniformly at random from the set of all hash functions (i.e. we assume true randomness).
- Let $m = 2n$ and suppose $n$ keys have been inserted.
Back to hashing

**Example**

- Suppose the hash function $h$ is chosen uniformly at random from the set of all hash functions (i.e. we assume true randomness).
- Let $m = 2n$ and suppose $n$ keys have been inserted.
- For any position $i$, use the union bound over all keys:

\[
\Pr(T[i] = 1) \leq \sum_{\text{n keys}} \frac{1}{m} = \frac{n}{m} = \frac{1}{2}.
\]
Suppose the hash function $h$ is chosen uniformly at random from the set of all hash functions (i.e. we assume true randomness).

Let $m = 2n$ and suppose $n$ keys have been inserted.

For any position $i$, use the union bound over all keys:

$$\Pr(T[i] = 1) \leq \sum_{n \text{ keys}} \frac{1}{m} = \frac{n}{m} = \frac{1}{2}.$$

Thus,

$$\Pr(\text{false positive}) \leq \frac{1}{2},$$

where false positive means that $\text{lookup}(x)$ returns TRUE even though $x$ has not been inserted. That is, $T[h(x)] = 1$. 
Back to hashing

**Example**

- Suppose the hash function $h$ is chosen uniformly at random from the set of all hash functions (i.e. we assume true randomness).
- Let $m = 2n$ and suppose $n$ keys have been inserted.
- For any position $i$, use the union bound over all keys:
  \[
  \Pr(T[i] = 1) \leq \sum_{\text{n keys}} \frac{1}{m} = \frac{n}{m} = \frac{1}{2}.
  \]
- Thus,
  \[
  \Pr(\text{false positive}) \leq \frac{1}{2},
  \]
  where *false positive* means that lookup$(x)$ returns **true** even though $x$ has not been inserted. That is, $T[h(x)] = 1$.

**Observe**

We could increase $m$ to get a better (smaller) probability of error, but this would require $m$ to be quite large if we want a *small* risk of error.
Observe

In the previous example we used the assumption of true randomness, which is not realistic. However, one can show that a weakly universal set of hash functions will give the same bound.

As an exercise, modify the proof in the previous example to hold for weakly universal hashing.
Multiple hash functions

Instead of using only one hash function we use multiple hash functions. Universe $U$ containing $u$ keys. Bit string $T$ of size $m$.

We use $k$ independent hash functions $h_1, \ldots, h_k$.

Still only one bit string $T$ of length $m$. 
Multiple hash functions

Instead of using only one hash function we use multiple hash functions. Universe $U$ containing $u$ keys. Bit string $T$ of size $m$.

- We use $k$ independent hash functions $h_1, \ldots, h_k$.
- Still only one bit string $T$ of length $m$.

- add($x$) For each $i \in \{1, \ldots, k\}$, set $T[h_i(x)]$ to 1.
- lookup($x$) Return TRUE if, for all $i \in \{1, \ldots, k\}$, $T[h_i(x)] = 1$. Otherwise return FALSE.
Multiple hash functions

Instead of using only one hash function we use multiple hash functions.

Universe $U$ containing $u$ keys.  Bit string $T$ of size $m$.

- We use $k$ independent hash functions $h_1, \ldots, h_k$.
- Still only one bit string $T$ of length $m$.

1. **add($x$)**  For each $i \in \{1, \ldots, k\}$, set $T[h_i(x)]$ to 1.
2. **lookup($x$)**  Return TRUE if, for all $i \in \{1, \ldots, k\}$, $T[h_i(x)] = 1$. Otherwise return FALSE.

Again, only false positives are possible.
Multiple hash functions

Instead of using only one hash function we use multiple hash functions. Universe $U$ containing $u$ keys. Bit string $T$ of size $m$.

- We use $k$ independent hash functions $h_1, \ldots, h_k$.
- Still only one bit string $T$ of length $m$.

- **add**($x$) For each $i \in \{1, \ldots, k\}$, set $T[h_i(x)]$ to 1.
- **lookup**($x$) Return TRUE if, for all $i \in \{1, \ldots, k\}$, $T[h_i(x)] = 1$. Otherwise return FALSE.

**Observe**

- Again, only false positives are possible.
- It is possible to parallelise the computation of the $k$ hash functions.
Bloom filters

The hashing scheme we just described is called a Bloom filter.
Bloom filters

The hashing scheme we just described is called a Bloom filter.

**Theorem**

Using a bloom filter with \( k \) hash functions, there is a \( k \) (that depends on \( m \) and \( n \)) such that the probability of a false positive (i.e. a lookup returns TRUE instead of FALSE) is at most

\[
0.7 \left( \frac{m}{n} \right).
\]
Bloom filters

The hashing scheme we just described is called a **Bloom filter**.

**Theorem**

Using a bloom filter with \( k \) hash functions, there is a \( k \) (that depends on \( m \) and \( n \)) such that the probability of a false positive (i.e. a lookup returns \text{TRUE} instead of \text{FALSE}) is at most

\[
0.7 \left( \frac{m}{n} \right). 
\]

**Observe**

The probability of error does not depend on the universe size \( u \).

**Observe**

The proof of the theorem assumes true randomness. It is not clear if the same result holds when the hash functions are chosen from a weakly universal set of hash functions.

As an exercise, work out what part of the proof breaks if weakly universal hashing is used.
Let $s$ be the number of ones in the bit string ("hash table") after $n$ insertions.

Then $s \leq kn$. We will decide the value of $k$ later.
Bloom filters

**Proof**

- Let $s$ be the number of ones in the bit string ("hash table") after $n$ insertions.
- Then $s \leq kn$. We will decide the value of $k$ later.
- Suppose we lookup a key $x$ than has not been inserted.
- We assume true randomness, so for each of the $k$ hash functions $h_i (i \in \{1, \ldots, k\})$, the probability that $T[h_i(x)] = 1$ is $\frac{s}{m} \leq \frac{kn}{m}$. 

Bloom filters

**Proof**

- Let $s$ be the number of ones in the bit string ("hash table") after $n$ insertions.
- Then $s \leq kn$. We will decide the value of $k$ later.
- Suppose we lookup a key $x$ than has not been inserted.
- We assume true randomness, so for each of the $k$ hash functions $h_i$ ($i \in \{1, \ldots, k\}$), the probability that $T[h_i(x)] = 1$ is $\frac{s}{m} \leq \frac{kn}{m}$.
- Since the hash functions are independent, the probability that all of them map $x$ onto a 1 in the bit string is at most $\left(\frac{kn}{m}\right)^k$. 
**Proof**

- Let $s$ be the number of ones in the bit string (“hash table”) after $n$ insertions.
- Then $s \leq kn$. We will decide the value of $k$ later.
- Suppose we lookup a key $x$ than has not been inserted.
- We assume true randomness, so for each of the $k$ hash functions $h_i$ ($i \in \{1, \ldots, k\}$), the probability that $T[h_i(x)] = 1$ is $\frac{s}{m} \leq \frac{kn}{m}$.
- Since the hash functions are independent, the probability that all of them map $x$ onto a 1 in the bit string is at most $\left(\frac{kn}{m}\right)^k$.
- Thus, the probability of a false positive is upper bounded by

\[
\left(\frac{kn}{m}\right)^k.
\]
Let $s$ be the number of ones in the bit string ("hash table") after $n$ insertions.

Then $s \leq kn$. We will decide the value of $k$ later.

Suppose we lookup a key $x$ than has not been inserted.

We assume true randomness, so for each of the $k$ hash functions $h_i$ ($i \in \{1, \ldots, k\}$), the probability that $T[h_i(x)] = 1$ is $\frac{s}{m} \leq \frac{kn}{m}$.

Since the hash functions are independent, the probability that all of them map $x$ onto a 1 in the bit string is at most $\left(\frac{kn}{m}\right)^k$.

Thus, the probability of a false positive is upper bounded by $\left(\frac{kn}{m}\right)^k$.

We now want to choose $k$ such that this error probability is minimised.
Minimise \( \left( \frac{kn}{m} \right)^k \) with respect to \( k \).
Minimise \( \left( \frac{kn}{m} \right)^k \) with respect to \( k \).

After some maths, involving differentiation, we find that \( k = \frac{m}{ne} \) minimises the formula.
Bloom filters

**Proof continued...**

1. Minimise \( \left( \frac{kn}{m} \right)^k \) with respect to \( k \).
2. After some maths, involving differentiation, we find that \( k = \frac{m}{ne} \) minimises the formula.
3. Thus, using this \( k \), the probability of a false positive error is at most

\[
\left( \frac{kn}{m} \right)^k = \left( \frac{m}{ne} \cdot \frac{n}{m} \right) \frac{m}{ne} = \left( \frac{1}{e^e} \right) \frac{m}{n} \approx (0.6922 \ldots) \frac{m}{n} \leq 0.7 \frac{m}{n}.
\]
Minimise \((\frac{kn}{m})^k\) with respect to \(k\).

\[
\frac{kn}{m} = \frac{m}{ne} = \left(\frac{1}{e}\right) \frac{m}{n} 
\approx (0.6922 \ldots) \frac{m}{n} \leq 0.7 \frac{m}{n}.
\]

**Example**

<table>
<thead>
<tr>
<th>(k)</th>
<th>(m) and (n)</th>
<th>Approx probability of a false positive</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>(m \approx 5.4 \cdot n)</td>
<td>0.14</td>
</tr>
<tr>
<td>5</td>
<td>(m \approx 14 \cdot n)</td>
<td>0.0067</td>
</tr>
<tr>
<td>10</td>
<td>(m \approx 27 \cdot n)</td>
<td>0.000045</td>
</tr>
<tr>
<td>15</td>
<td>(m \approx 41 \cdot n)</td>
<td>3.1 \cdot 10^{-7}</td>
</tr>
<tr>
<td>20</td>
<td>(m \approx 54 \cdot n)</td>
<td>2.0 \cdot 10^{-9}</td>
</tr>
</tbody>
</table>