Bloom Filters

Benjamin Sach
(based on slides by Ashley Montanaro)
Introduction

In this lecture we are interested in space efficient data structures for storing a set $S$ which support only two, basic operations:

$\text{INSERT}(k)$ - inserts the key $k$ from $U$ into $S$

$\text{MEMBER}(k)$ - output ‘yes’ if $k \in S$ and ‘no’ otherwise

$U$ is the universe, containing all possible keys

Let $n$ be an upper bound on the number of keys that will ever be in $S$

Our motivation comes from applications where the size of the universe $U$ is much much larger than $n$
Introduction

In this lecture we are interested in space efficient data structures for storing a set $S$ which support only two, basic operations:

$\text{INSERT}(k)$ - inserts the key $k$ from $U$ into $S$

$\text{MEMBER}(k)$ - output ‘yes’ if $k \in S$ 

and ‘no’ otherwise

$U$ is the universe, containing all possible keys

Let $n$ be an upper bound on the number of keys that will ever be in $S$

Our motivation comes from applications where the size of the universe $U$ is much much larger than $n$.
In this lecture we are interested in space efficient data structures for storing a set $S$ which support only two, basic operations:

- **INSERT($k$)** - inserts the key $k$ from $U$ into $S$

- **MEMBER($k$)** - output ‘yes’ if $k \in S$
  
  *and ‘no’ otherwise*

$U$ is the universe, containing all possible keys.

Let $n$ be an upper bound on the number of keys that will ever be in $S$.

Our motivation comes from applications where the size of the universe $U$ is much much larger than $n$.
Introduction

In this lecture we are interested in space efficient data structures for storing a set $S$ which support only two, basic operations:

**INSERT**($k$) - inserts the key $k$ from $U$ into $S$

**MEMBER**($k$) - output ‘yes’ if $k \in S$ and ‘no’ otherwise

$U$ is the universe, containing all possible keys

Let $n$ be an upper bound on the number of keys that will ever be in $S$

Our motivation comes from applications where the size of the universe $U$ is *much much* larger than $n$

**Important**: You cannot ask “which keys are in $S$?”, only “is this key in $S$?”
Example and Motivation

Imagine you are attempting to build a **blacklist** of unsafe URLs that users should not visit.

The universe contains all possible URLs.

Whenever a new unsafe URL is discovered it is inserted into the data structure.
Whenever we want to visit a URL we check the data structure.
Example and Motivation

Imagine you are attempting to build a **blacklist** of unsafe URLs that users should not visit.

The universe contains all possible URLs.

Whenever a new unsafe URL is discovered it is inserted into the data structure. Whenever we want to visit a URL we check the data structure.

```
INSERT(www.AwfulVirus.com)
```
Example and Motivation

Imagine you are attempting to build a **blacklist** of unsafe URLs that users should not visit.

The universe contains all possible URLs.

Whenever a new unsafe URL is discovered it is inserted into the data structure. Whenever we want to visit a URL we check the data structure.

\[ \text{INSERT}(\text{www.AwfulVirus.com}) \]

\[ \text{INSERT}(\text{www.VirusStore.com}) \]
Example and Motivation

Imagine you are attempting to build a **blacklist** of unsafe URLs that users should not visit.

The universe contains all possible URLs.

Whenever a new unsafe URL is discovered it is inserted into the data structure.
Whenever we want to visit a URL we check the data structure.

**INSERT** (www.AwfulVirus.com)
**INSERT** (www.VirusStore.com)

Disclaimer: I take no responsibility for the contents of these websites.
Example and Motivation

Imagine you are attempting to build a **blacklist** of unsafe URLs that users should not visit.

The universe contains all possible URLs.

Whenever a new unsafe URL is discovered it is inserted into the data structure.
Whenever we want to visit a URL we check the data structure.

\[
\text{\textbf{INSERT(www.AwfulVirus.com)}}
\]
\[
\text{\textbf{INSERT(www.VirusStore.com)}}
\]
\[
\text{\textbf{MEMBER(www.BBC.co.uk)} - returns ‘no’}
\]

Disclaimer: I take no responsibility for the contents of these websites.
Example and Motivation

Imagine you are attempting to build a **blacklist** of unsafe URLs that users should not visit.

The universe contains all possible URLs.

Whenever a new unsafe URL is discovered it is inserted into the data structure.

Whenever we want to visit a URL we check the data structure.

\[ \text{INSERT(www.AwfulVirus.com)} \]

\[ \text{INSERT(www.VirusStore.com)} \]

\[ \text{MEMBER(www.BBC.co.uk)} - \text{returns 'no'} \]

\[ \text{MEMBER(www.VirusStore.com)} - \text{returns 'yes'} \]

Disclaimer: I take no responsibility for the contents of these websites.
Example and Motivation

Imagine you are attempting to build a **blacklist** of unsafe URLs that users should not visit.

The universe contains all possible URLs.

Whenever a new unsafe URL is discovered it is inserted into the data structure.

Whenever we want to visit a URL we check the data structure.

\[
\text{INSERT} \left( \text{www.AwfulVirus.com} \right) \\
\text{INSERT} \left( \text{www.VirusStore.com} \right) \\
\text{MEMBER} \left( \text{www.BBC.co.uk} \right) - \text{returns ‘no’} \\
\text{MEMBER} \left( \text{www.VirusStore.com} \right) - \text{returns ‘yes’}
\]
Example and Motivation

Imagine you are attempting to build a **blacklist** of unsafe URLs that users should not visit.

The universe contains all possible URLs.

Whenever a new unsafe URL is discovered it is inserted into the data structure.
Whenever we want to visit a URL we check the data structure.

\[
\text{INSERT(www.AwfulVirus.com)}
\]

\[
\text{MEMBER(www.BBC.co.uk)} \text{ - returns 'no'}
\]

\[
\text{MEMBER(www.VirusStore.com)} \text{ - returns 'yes'}
\]

\[
\text{INSERT(www.CleanUpPC.com)}
\]
Imagine you are attempting to build a **blacklist** of unsafe URLs that users should not visit.

The universe contains all possible URLs.

Whenever a new unsafe URL is discovered it is inserted into the data structure. Whenever we want to visit a URL we check the data structure.

- **INSERT**(`www.AwfulVirus.com`) - returns ‘no’
- **INSERT**(`www.VirusStore.com`) - returns ‘yes’
- **MEMBER**(`www.BBC.co.uk`) - returns ‘yes’
Example and Motivation

Imagine you are attempting to build a **blacklist** of unsafe URLs that users should not visit. The universe contains all possible URLs.

Whenever a new unsafe URL is discovered it is inserted into the data structure. Whenever we want to visit a URL we check the data structure.

\[
\text{**INSERT**} (\text{www.AwfulVirus.com})
\]

\[
\text{**INSERT**} (\text{www.VirusStore.com})
\]

\[
\text{**MEMBER**} (\text{www.BBC.co.uk}) - \text{returns ‘no’}
\]

\[
\text{**MEMBER**} (\text{www.VirusStore.com}) - \text{returns ‘yes’}
\]

\[
\text{**INSERT**} (\text{www.CleanUpPC.com})
\]

\[
\text{**MEMBER**} (\text{www.BBC.co.uk}) - \text{returns ‘yes’}
\]
Example and Motivation

Imagine you are attempting to build a **blacklist** of unsafe URLs that users should not visit.

The universe contains all possible URLs.

Whenever a new unsafe URL is discovered it is inserted into the data structure. Whenever we want to visit a URL we check the data structure.

```
INSERT(www.AwfulVirus.com)
INSERT(www.VirusStore.com)
MEMBER(www.BBC.co.uk) - returns ‘no’
MEMBER(www.VirusStore.com) - returns ‘yes’
INSERT(www.CleanUpPC.com)
MEMBER(www.BBC.co.uk) - returns ‘yes’
```

A **Bloom filter** is a *randomised* data structure - sometimes it gets the answer wrong.
Bloom filters

A Bloom filter is a randomised data structure for storing a set $S$ which supports two operations
Bloom filters

A Bloom filter is a *randomised* data structure for storing a set $S$ which supports two operations.

The \textsc{Insert}$(k)$ operation inserts the key $k$ from $U$ into $S$. 
Bloom filters

A Bloom filter is a *randomised* data structure for storing a set $S$ which supports two operations

The $\text{INSERT}(k)$ operation inserts the key $k$ from $U$ into $S$ *(it never does this incorrectly)*
Bloom filters

A Bloom filter is a *randomised* data structure for storing a set \( S \) which supports two operations

The `INSERT(\( k \))` operation inserts the key \( k \) from \( U \) into \( S \) 

(it never does this incorrectly)

In a bloom filter, the `MEMBER(\( k \))` operation
Bloom filters

A Bloom filter is a *randomised* data structure for storing a set $S$ which supports two operations

The $\text{INSERT}(k)$ operation inserts the key $k$ from $U$ into $S$ *(it never does this incorrectly)*

In a bloom filter, the $\text{MEMBER}(k)$ operation

always returns ‘yes’ if $k \in S$
Bloom filters

A Bloom filter is a randomised data structure for storing a set $S$ which supports two operations.

The $\text{INSERT}(k)$ operation inserts the key $k$ from $U$ into $S$ (it never does this incorrectly).

In a bloom filter, the $\text{MEMBER}(k)$ operation always returns ‘yes’ if $k \in S$.

However, if $k$ is not in $S$ there is a small chance (say 1%) that it will still say ‘yes’.
Bloom filters

A Bloom filter is a randomised data structure for storing a set \( S \) which supports two operations.

The \texttt{INSERT}(k) operation inserts the key \( k \) from \( U \) into \( S \) (it never does this incorrectly).

In a bloom filter, the \texttt{MEMBER}(k) operation:

- always returns 'yes' if \( k \in S \)
- however, if \( k \) is not in \( S \):
  - there is a small chance (say 1\%) that it will still say 'yes'

**Why use a Bloom filter then?**
Bloom filters

A **Bloom filter** is a *randomised* data structure for storing a set $S$ which supports two operations

The **INSERT($k$)** operation inserts the key $k$ from $U$ into $S$ *(it never does this incorrectly)*

In a bloom filter, the **MEMBER($k$)** operation

always returns ‘yes’ if $k \in S$

however, if $k$ is not in $S$

there is a small chance (say 1%) that it will still say ‘yes’

*Why use a Bloom filter then?*

Both operations run in $O(1)$ time and the space used is *very very good*
Bloom filters

A Bloom filter is a *randomised* data structure for storing a set \( S \) which supports two operations.

The \textsc{Insert}(\( k \)) operation inserts the key \( k \) from \( U \) into \( S \) (it never does this incorrectly).

In a bloom filter, the \textsc{Member}(\( k \)) operation:

- always returns ‘yes’ if \( k \in S \)
- however, if \( k \) is not in \( S \):
  - there is a small chance (say 1\%) that it will still say ‘yes’

Why use a Bloom filter then?

Both operations run in \( O(1) \) time and the space used is very very good.

It will use \( O(n) \) bits of space to store up to \( n \) keys.
Bloom filters

A Bloom filter is a randomised data structure for storing a set \( S \) which supports two operations.

The \text{INSERT}(k) operation inserts the key \( k \) from \( U \) into \( S \) (it never does this incorrectly).

In a bloom filter, the \text{MEMBER}(k) operation

always returns ‘yes’ if \( k \in S \)

however, if \( k \) is not in \( S \)

there is a small chance (say 1\%) that it will still say ‘yes’

Why use a Bloom filter then?

Both operations run in \( O(1) \) time and the space used is very very good.

It will use \( O(n) \) bits of space to store up to \( n \) keys

- the exact number of bits will depend on the failure probability
Bloom filters

A Bloom filter is a *randomised* data structure for storing a set \( S \) which supports two operations:

The `INSERT(k)` operation inserts the key \( k \) from \( U \) into \( S \) (it never does this incorrectly).

In a bloom filter, the `MEMBER(k)` operation:

- always returns ‘yes’ if \( k \in S \)
- however, if \( k \) is not in \( S \), there is a small chance (say 1%) that it will still say ‘yes’

*Why use a Bloom filter then?*

Both operations run in \( O(1) \) time and the space used is *very very good*.

It will use \( O(n) \) bits of space to store up to \( n \) keys:

- the exact number of bits will depend on the failure probability *we’ll come back to this at the end*
Approach 1: build an array

Before discussing Bloom filters, let's consider a naive approach using an array...

For simplicity, let us think of the universe $U$ as containing numbers $1, 2, 3 \ldots |U|$. 
Approach 1: build an array

Before discussing Bloom filters, let's consider a naive approach using an array...

For simplicity, let us think of the universe $U$ as containing numbers $1, 2, 3 \ldots |U|$.

We could maintain a bit string $B$.
Approach 1: build an array

Before discussing Bloom filters, let's consider a naive approach using an array...

For simplicity, let us think of the universe $U$ as containing numbers $1, 2, 3 \ldots |U|$.

We could maintain a bit string $B$

Example:

$$
\begin{array}{ccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
B & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
\end{array}
$$
Approach 1: build an array

Before discussing Bloom filters, let’s consider a naive approach using an array...

For simplicity, let us think of the universe $U$ as containing numbers $1, 2, 3 \ldots |U|$.

We could maintain a bit string $B$

where $B[k] = 1$ if $k \in S$ and $B[k] = 0$ otherwise

Example:

\[
\begin{array}{ccccccccccc}
    & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
B & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
\end{array}
\]
Approach 1: build an array

Before discussing Bloom filters, let's consider a naive approach using an array...

For simplicity, let us think of the universe $U$ as containing numbers $1, 2, 3 \ldots |U|$. 

We could maintain a bit string $B$

where $B[k] = 1$ if $k \in S$ and $B[k] = 0$ otherwise

**Example:**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

here $|U| = 10$ and $S$ contains 3, 6 and 8
Approach 1: build an array

Before discussing Bloom filters, let's consider a naive approach using an array...

For simplicity, let us think of the universe $U$ as containing numbers $1, 2, 3 \ldots |U|$.

We could maintain a bit string $B$

where $B[k] = 1$ if $k \in S$ and $B[k] = 0$ otherwise

Example:

```
B
```

```
<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

Here $|U| = 10$ and $S$ contains 3, 6 and 8

While the operations take $O(1)$ time, this array is $|U|$ bits long!
Approach 1: build an array

Before discussing Bloom filters, let's consider a naive approach using an array...

For simplicity, let us think of the universe $U$ as containing numbers $1, 2, 3 \ldots |U|$.

We could maintain a bit string $B$ where $B[k] = 1$ if $k \in S$ and $B[k] = 0$ otherwise.

Example:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Here $|U| = 10$ and $S$ contains 3, 6 and 8.

While the operations take $O(1)$ time, this array is $|U|$ bits long!

*It certainly isn't suitable for our application*
Approach 2: build a hash table

We could solve the problem by hashing...

We now maintain a *much shorter* bit string $B$ of some length $m < |U|$ (to be determined later)

Example:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Approach 2: build a hash table

We could solve the problem by hashing...

We now maintain a *much shorter* bit string $B$ of some length $m < |U|$ (to be determined later)

Assume we have access to a hash function $h$ which maps each key $k \in U$ to an integer $h(k)$ between 1 and $m$

Example:

```
  1 2 3
B 0 0 0
```
Approach 2: build a hash table

We could solve the problem by hashing...

We now maintain a *much shorter* bit string \( B \) of some length \( m < |U| \)

(to be determined later)

Assume we have access to a hash function \( h \) which maps each key \( k \in U \)
to an integer \( h(k) \) between 1 and \( m \)

Example:

Imagine that \( m = 3 \) and

\[
\begin{align*}
B & \quad \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \\
& \quad \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}
\end{align*}
\]

\[
\begin{align*}
h(www.AwfulVirus.com) &= 2 \\
h(www.VirusStore.com) &= 3 \\
h(www.BBC.co.uk) &= 3
\end{align*}
\]
Approach 2: build a hash table

We could solve the problem by hashing...

We now maintain a *much shorter* bit string $B$ of some length $m < |U|$ (to be determined later)

Assume we have access to a hash function $h$ which maps each key $k \in U$ to an integer $h(k)$ between 1 and $m$

**$\text{INSERT}(k)$** sets $B[h(k)] = 1$

<table>
<thead>
<tr>
<th>Example:</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Imagine that $m = 3$ and

- $h(\text{www.AwfulVirus.com}) = 2$
- $h(\text{www.VirusStore.com}) = 3$
- $h(\text{www.BBC.co.uk}) = 3$
Approach 2: build a hash table

We could solve the problem by hashing...

We now maintain a *much shorter* bit string $B$ of some length $m < |U|$

(to be determined later)

Assume we have access to a hash function $h$ which maps each key $k \in U$

to an integer $h(k)$ between 1 and $m$

$\text{INSERT}(k)$ sets $B[h(k)] = 1$  \hspace{1cm}  $\text{MEMBER}(k)$ returns ‘yes’ if $B[h(k)] = 1$

and ‘no’ if $B[h(k)] = 0$

---

**Example:**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Imagine that $m = 3$ and

$h(\text{www.AwfulVirus.com}) = 2$

$h(\text{www.VirusStore.com}) = 3$

$h(\text{www.BBC.co.uk}) = 3$
Approach 2: build a hash table

We could solve the problem by hashing...

We now maintain a much shorter bit string $B$ of some length $m < |U|$ (to be determined later)

Assume we have access to a hash function $h$ which maps each key $k \in U$ to an integer $h(k)$ between 1 and $m$

$\text{INSERT}(k)$ sets $B[h(k)] = 1$  
$\text{MEMBER}(k)$ returns ‘yes’ if $B[h(k)] = 1$ and ‘no’ if $B[h(k)] = 0$

Example:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Imagine that $m = 3$ and $h(www.AwfulVirus.com) = 2$  
$h(www.VirusStore.com) = 3$  
$h(www.BBC.co.uk) = 3$
Approach 2: build a hash table

We could solve the problem by hashing...

We now maintain a *much shorter* bit string $B$ of some length $m < |U|$ (to be determined later)

Assume we have access to a hash function $h$ which maps each key $k \in U$ to an integer $h(k)$ between 1 and $m$

**INSERT**($k$) sets $B[h(k)] = 1$  
**MEMBER**($k$) returns ‘yes’ if $B[h(k)] = 1$ and ‘no’ if $B[h(k)] = 0$

Example: 

$B$  

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Imagine that $m = 3$ and  

$h(www.AwfulVirus.com) = 2$  
$h(www.VirusStore.com) = 3$  
$h(www.BBC.co.uk) = 3$
Approach 2: build a hash table

We could solve the problem by hashing...

We now maintain a much shorter bit string $B$ of some length $m < |U|$ (to be determined later)

Assume we have access to a hash function $h$ which maps each key $k \in U$ to an integer $h(k)$ between 1 and $m$

$\text{INSERT}(k)$ sets $B[h(k)] = 1$  \hspace{1cm} $\text{MEMBER}(k)$ returns 'yes' if $B[h(k)] = 1$ and 'no' if $B[h(k)] = 0$

Example:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B$</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Imagine that $m = 3$ and

$h(\text{www.AwfulVirus.com}) = 2$

$h(\text{www.VirusStore.com}) = 3$

$h(\text{www.BBC.co.uk}) = 3$
Approach 2: build a hash table

We could solve the problem by hashing...

We now maintain a *much shorter* bit string $B$ of some length $m < |U|$ (to be determined later)

Assume we have access to a hash function $h$ which maps each key $k \in U$ to an integer $h(k)$ between 1 and $m$

**Insert**($k$) sets $B[h(k)] = 1$  
**Member**($k$) returns ‘yes’ if $B[h(k)] = 1$ and ‘no’ if $B[h(k)] = 0$

**Example:**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B$</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Imagine that $m = 3$ and

$h$(www.AwfulVirus.com) = 2
$h$(www.VirusStore.com) = 3
$h$(www.BBC.co.uk) = 3

**Insert**(www.AwfulVirus.com)
**Insert**(www.VirusStore.com)
Approach 2: build a hash table

We could solve the problem by hashing...

We now maintain a *much shorter* bit string $B$ of some length $m < |U|$ (to be determined later)

Assume we have access to a hash function $h$ which maps each key $k \in U$ to an integer $h(k)$ between 1 and $m$

**Example:**

Imagine that $m = 3$ and $h(www.AwfulVirus.com) = 2$

$\text{INSERT}(www.AwfulVirus.com)$

$\text{INSERT}(www.VirusStore.com)$

$\text{MEMBER}(www.BBC.co.uk)$ - returns ‘yes’
Approach 2: build a hash table

We could solve the problem by hashing...

We now maintain a *much shorter* bit string $B$ of some length $m < |U|$ (to be determined later)

Assume we have access to a hash function $h$ which maps each key $k \in U$ to an integer $h(k)$ between 1 and $m$

**INSERT**($k$) sets $B[h(k)] = 1$  
**MEMBER**($k$) returns ‘yes’ if $B[h(k)] = 1$ and ‘no’ if $B[h(k)] = 0$

**Example:**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B$</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Imagine that $m = 3$ and

- $h$(www.AwfulVirus.com) = 2
- $h$(www.VirusStore.com) = 3
- $h$(www.BBC.co.uk) = 3

This is called a collision
Approach 2: build a hash table

The problem with hashing is that if $m < |U|$ then there will be some keys that hash to the same positions

(\textit{these are called collisions})
Approach 2: build a hash table

The problem with hashing is that if $m < |U|$ then there will be some keys that hash to the same positions (these are called collisions).

If we call MEMBER($k$) for some key $k$ not in $S$ but there is a key $k' \in S$ with $h(k) = h(k')$ we will incorrectly output ‘yes’.
Approach 2: build a hash table

The problem with hashing is that if \( m < |U| \) then

there will be some keys that hash to the same positions

*(these are called collisions)*

If we call \( \text{MEMBER}(k) \) for some key \( k \) *not* in \( S \)

but there is a key \( k' \in S \) with \( h(k) = h(k') \)

we will incorrectly output ‘yes’

To make sure that the probability of an error is low for *every operation sequence*,

we pick each \( h(k) \) at random
Approach 2: build a hash table

The problem with hashing is that if $m < |U|$ then

there will be some keys that hash to the same positions

(these are called collisions)

If we call MEMBER($k$) for some key $k$ not in $S$

but there is a key $k' \in S$ with $h(k) = h(k')$

we will incorrectly output ‘yes’

To make sure that the probability of an error is low for every operation sequence,

we pick each $h(k)$ at random

For every key $k \in U$, the value of $h(k)$ is chosen uniformly at random:

that is, the probability that $h(k) = j$ is $\frac{1}{m}$ for all $j$ between 1 and $m$

(all positions are equally likely)
What is the probability of an error?

Assume we have already \textit{INSERTED} \( n \) keys into the structure

Further, we have just called

\texttt{MEMBER}(k) \textit{for some key} \( k \) \textit{not in} \( S \)

(which will check whether \( B[h(k)] = 1 \))
What is the probability of an error?

Assume we have already \textbf{INSERTED} \textit{n} keys into the structure

Further, we have just called

\texttt{MEMBER}(k) for some key \texttt{k not in } S

\text{(which will check whether } B[h(k)] = 1)\text{)

We want to know the probability that the answer returned is ‘yes’ (which would be bad)
What is the probability of an error?

Assume we have already INSERTED \( n \) keys into the structure

Further, we have just called

\[
\text{MEMBER}(k) \text{ for some key } k \text{ not in } S
\]

(which will check whether \( B[h(k)] = 1 \))

We want to know the probability that the answer returned is ‘yes’ (which would be bad)

The bit-string \( B \) contains at most \( n \) 1’s among the \( m \) positions
What is the probability of an error?

Assume we have already \textbf{INSERTED} \textit{n} keys into the structure

Further, we have just called

\begin{center}
\textbf{MEMBER}(k) \text{ for some key } k \text{ not in } S
\end{center}

(which will check whether \( B[h(k)] = 1 \))

We want to know the probability that the answer returned is ‘yes’ (which would be bad)

The bit-string \( B \) contains at most \( n \) 1’s among the \( m \) positions

\[
B = \begin{array}{cccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\hline
m
\end{array}
\]
What is the probability of an error?

Assume we have already **INSERTED** \( n \) keys into the structure

Further, we have just called

\[ \text{MEMBER}(k) \text{ for some key } k \text{ not in } S \]

(which will check whether \( B[h(k)] = 1 \))

We want to know the probability that the answer returned is ‘yes’ (which would be bad)

The bit-string \( B \) contains at most \( n \) 1’s among the \( m \) positions

\[
B = \begin{array}{cccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1
\end{array}
\]

By definition, \( h(k) \) is equally likely to be any position between 1 and \( m \)
What is the probability of an error?

Assume we have already \textbf{INSERTED} \( n \) keys into the structure

Further, we have just called

\begin{equation*}
\text{MEMBER}(k) \text{ for some key } k \text{ not in } S
\end{equation*}

(which will check whether \( B[h(k)] = 1 \))

We want to know the probability that the answer returned is ‘yes’ (which would be bad)

The bit-string \( B \) contains at most \( n \) 1’s among the \( m \) positions

By definition, \( h(k) \) is equally likely to be any position between 1 and \( m \)
What is the probability of an error?

Assume we have already **INSERTED** \( n \) keys into the structure

Further, we have just called

\[ \text{MEMBER}(k) \text{ for some key } k \text{ not in } S \]

(which will check whether \( B[h(k)] = 1 \))

We want to know the probability that the answer returned is ‘yes’ (which would be bad)

The bit-string \( B \) contains at most \( n \) 1’s among the \( m \) positions

\[
\begin{array}{ccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 1 \downarrow \\
\hline
m
\end{array}
\]

By definition, \( h(k) \) is equally likely to be any position between 1 and \( m \)
What is the probability of an error?

Assume we have already **inserted** \( n \) keys into the structure

Further, we have just called

\[ \text{MEMBER}(k) \] for some key \( k \) **not** in \( S \)

(which will check whether \( B[h(k)] = 1 \))

We want to know the probability that the answer returned is ‘yes’ (which would be bad)

The bit-string \( B \) contains at most \( n \) 1’s among the \( m \) positions

\[
B = \begin{array}{ccccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\hline
& & & & & & & & m
\end{array}
\]

By definition, \( h(k) \) is equally likely to be any position between 1 and \( m \)

Therefore the probability that \( B[h(k)] = 1 \) is at most \( \frac{n}{m} \)
What is the probability of an error?

Assume we have already inserted \( n \) keys into the structure.

Further, we have just called
\[
\text{MEMBER}(k) \quad \text{for some key } k \text{ not in } S
\]
(which will check whether \( B[h(k)] = 1 \))

We want to know the probability that the answer returned is ‘yes’ (which would be bad).

The bit-string \( B \) contains at most \( n \) 1’s among the \( m \) positions.

By definition, \( h(k) \) is equally likely to be any position between 1 and \( m \).

Therefore the probability that \( B[h(k)] = 1 \) is at most \( \frac{n}{m} \)

If we choose \( m = 100n \) then we get a failure probability of at most 1\%.
Approach 2: build a hash table

We have developed a *randomised* data structure for storing a set $S$ which supports two operations.
Approach 2: build a hash table

We have developed a *randomised* data structure for storing a set $S$ which supports two operations.

The $\text{INSERT}(k)$ operation inserts the key $k$ from $U$ into $S$. 
Approach 2: build a hash table

We have developed a *randomised* data structure for storing a set $S$ which supports two operations:

The $\text{INSERT}(k)$ operation inserts the key $k$ from $U$ into $S$ *(it never does this incorrectly)*
Approach 2: build a hash table

We have developed a *randomised* data structure for storing a set $S$ which supports two operations.

The $\text{INSERT}(k)$ operation inserts the key $k$ from $U$ into $S$ *(it never does this incorrectly)*.

Like in a bloom filter, the $\text{MEMBER}(k)$ operation...
Approach 2: build a hash table

We have developed a *randomised* data structure for storing a set $S$ which supports two operations

The $\text{INSERT}(k)$ operation inserts the key $k$ from $U$ into $S$ *(it never does this incorrectly)*

Like in a bloom filter, the $\text{MEMBER}(k)$ operation

    always returns ‘yes’ if $k \in S$
Approach 2: build a hash table

We have developed a randomised data structure for storing a set $S$ which supports two operations.

The $\text{INSERT}(k)$ operation inserts the key $k$ from $U$ into $S$ (it never does this incorrectly).

Like in a bloom filter, the $\text{MEMBER}(k)$ operation always returns ‘yes’ if $k \in S$.

However, if $k$ is not in $S$ there is a small chance (in fact 1%) that it will still say ‘yes’.
Approach 2: build a hash table

We have developed a \textit{randomised} data structure for storing a set $S$ which supports two operations.

The $\text{INSERT}(k)$ operation inserts the key $k$ from $U$ into $S$

(it never does this incorrectly)

Like in a bloom filter, the $\text{MEMBER}(k)$ operation

always returns ‘yes’ if $k \in S$

however, if $k$ is not in $S$

there is a small chance (in fact 1\%) that it will still say ‘yes’

Both operations run in $O(1)$ time and the space used is $100n$ bits.
Approach 2: build a hash table

We have developed a \textit{randomised} data structure for storing a set $S$ which supports two operations

The \texttt{INSERT}(k) operation inserts the key $k$ from $U$ into $S$\textit{ (it never does this incorrectly)}.

Like in a bloom filter, the \texttt{MEMBER}(k) operation

always returns ‘yes’ if $k \in S$

however, if $k$ is not in $S$

there is a small chance (in fact 1\%) that it will still say ‘yes’

Both operations run in $O(1)$ time and the space used is $100n$ bits \textit{when storing up to $n$ keys}.
Approach 2: build a hash table

We have developed a *randomised* data structure for storing a set $S$ which supports two operations

The $\text{INSERT}(k)$ operation inserts the key $k$ from $U$ into $S$ *(it never does this incorrectly)*

Like in a bloom filter, the $\text{MEMBER}(k)$ operation

always returns ‘yes’ if $k \in S$

however, if $k$ is not in $S$

there is a small chance (in fact 1%) that it will still say ‘yes’

Both operations run in $O(1)$ time and the space used is $100n$ bits *when storing up to* $n$ *keys*

neither the space nor the failure probability depend on $|U|$
Approach 2: build a hash table

We have developed a *randomised* data structure for storing a set \( S \) which supports two operations.

The \texttt{INSERT}(\( k \)) operation inserts the key \( k \) from \( U \) into \( S \) (it never does this incorrectly).

Like in a bloom filter, the \texttt{MEMBER}(\( k \)) operation always returns ‘yes’ if \( k \in S \).

However, if \( k \) is not in \( S \), there is a small chance (in fact 1%) that it will still say ‘yes’.

Both operations run in \( O(1) \) time and the space used is 100\( n \) bits
when storing up to \( n \) keys.

Neither the space nor the failure probability depend on \(|U|\).

If we wanted a better probability, we could use more space.
Approach 2: build a hash table

We have developed a *randomised* data structure for storing a set $S$ which supports two operations:

The **INSERT**($k$) operation inserts the key $k$ from $U$ into $S$ *(it never does this incorrectly)*.

Like in a bloom filter, the **MEMBER**($k$) operation:

- always returns ‘yes’ if $k \in S$
- however, if $k$ is not in $S$
  - there is a small chance (in fact 1%) that it will still say ‘yes’

Both operations run in $O(1)$ time and the space used is $100n$ bits when storing up to $n$ keys.

Neither the space nor the failure probability depend on $|U|$.

If we wanted a better probability, we could use more space.

*Why use a Bloom filter then?*
Approach 2: build a hash table

We have developed a *randomised* data structure for storing a set \( S \) which supports two operations.

The \texttt{INSERT}(k) operation inserts the key \( k \) from \( U \) into \( S \)

\textit{(it never does this incorrectly)}

Like in a bloom filter, the \texttt{MEMBER}(k) operation

\begin{itemize}
  \item always returns ‘yes’ if \( k \in S \)
  \item however, if \( k \) is not in \( S \)
    \begin{itemize}
      \item there is a small chance (in fact 1\%) that it will still say ‘yes’
    \end{itemize}
\end{itemize}

Both operations run in \( O(1) \) time and the space used is 100\( n \) bits when storing up to \( n \) keys

neither the space nor the failure probability depend on \(|U|\)

\textit{if we wanted a better probability, we could use more space}

\textit{Why use a Bloom filter then?}

we will get \textit{much better} space usage for the same probability
Approach 3: build a bloom filter

We still maintain a bit string $B$ of some length $m < |U|$

Now we have $r$ hash functions: $h_1, h_2, \ldots, h_r$ (we will choose $r$ and $m$ later)

Each hash function $h_i$ maps a key $k$, to an integer $h_i(k)$ between 1 and $m$
Approach 3: build a bloom filter

We still maintain a bit string $B$ of some length $m < |U|$

Now we have $r$ hash functions: $h_1, h_2, \ldots, h_r$ \hspace{1cm} (we will choose $r$ and $m$ later)

Each hash function $h_i$ maps a key $k$, to an integer $h_i(k)$ between 1 and $m$

Imagine that $m = 4$, $r = 2$ and

Example: $\begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 0 & 0 & 0 \end{bmatrix}$

$h_1(\text{AwVi.com}) = 2$ \hspace{1cm} $h_2(\text{AwVi.com}) = 1$

$h_1(\text{ViSt.com}) = 3$ \hspace{1cm} $h_2(\text{ViSt.com}) = 2$

$h_1(\text{BBC.com}) = 2$ \hspace{1cm} $h_2(\text{BBC.com}) = 4$
Approach 3: build a bloom filter

We still maintain a bit string $B$ of some length $m < |U|$

Now we have $r$ hash functions: $h_1, h_2, \ldots, h_r$ (we will choose $r$ and $m$ later)

Each hash function $h_i$ maps a key $k$, to an integer $h_i(k)$ between 1 and $m$

**INSERT**($k$) sets $B[h_i(k)] = 1$

for all $i$ between 1 and $r$

**MEMBER**($k$) returns ‘yes’ if and only if

for all $i$, $B[h_i(k)] = 1$

Imagine that $m = 4$, $r = 2$ and

<table>
<thead>
<tr>
<th>Example:</th>
<th>$h_1(AwVi.com) = 2$</th>
<th>$h_2(AwVi.com) = 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>$h_1(ViSt.com) = 3$</th>
<th>$h_2(ViSt.com) = 2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$h_1(BBC.com) = 2$</td>
<td>$h_2(BBC.com) = 4$</td>
</tr>
</tbody>
</table>
Approach 3: build a bloom filter

We still maintain a bit string $B$ of some length $m < |U|$.

Now we have $r$ hash functions: $h_1, h_2, \ldots, h_r$ (we will choose $r$ and $m$ later).

Each hash function $h_i$ maps a key $k$, to an integer $h_i(k)$ between 1 and $m$.

**Example:** Imagine that $m = 4$, $r = 2$ and

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**INSERT**($k$) sets $B[h_i(k)] = 1$ for all $i$ between 1 and $r$.

**MEMBER**($k$) returns ‘yes’ if and only if for all $i$, $B[h_i(k)] = 1$.

$\begin{align*}
\text{INSERT}(\text{AwVi.com}) & \quad h_1(\text{AwVi.com}) = 2 \quad h_2(\text{AwVi.com}) = 1 \\
\text{INSERT}(\text{ViSt.com}) & \quad h_1(\text{ViSt.com}) = 3 \quad h_2(\text{ViSt.com}) = 2 \\
\text{INSERT}(\text{BBC.com}) & \quad h_1(\text{BBC.com}) = 2 \quad h_2(\text{BBC.com}) = 4
\end{align*}$
Approach 3: build a bloom filter

We still maintain a bit string $B$ of some length $m < |U|$

Now we have $r$ hash functions: $h_1, h_2, \ldots, h_r$ (we will choose $r$ and $m$ later)

Each hash function $h_i$ maps a key $k$, to an integer $h_i(k)$ between 1 and $m$

<table>
<thead>
<tr>
<th>$i$</th>
<th>$h_i(k)$</th>
<th>$B[h_i(k)]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$h_1(AwVi.com) = 2$</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>$h_2(AwVi.com) = 1$</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>$h_1(ViSt.com) = 3$</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>$h_2(ViSt.com) = 2$</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>$h_1(BBC.com) = 2$</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>$h_2(BBC.com) = 4$</td>
<td>0</td>
</tr>
</tbody>
</table>

Imagine that $m = 4$, $r = 2$ and

**Example:**

Imagine that $m = 4$, $r = 2$ and

Imagine that $m = 4$, $r = 2$ and
Approach 3: build a bloom filter

We still maintain a bit string $B$ of some length $m < |U|$

Now we have $r$ hash functions: $h_1, h_2, \ldots, h_r$ (we will choose $r$ and $m$ later)

Each hash function $h_i$ maps a key $k$, to an integer $h_i(k)$ between 1 and $m$

**INSERT**($k$) sets $B[h_i(k)] = 1$ for all $i$ between 1 and $r$

**MEMBER**($k$) returns ‘yes’ if and only if for all $i$, $B[h_i(k)] = 1$

Imagine that $m = 4$, $r = 2$ and

Example:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**INSERT**(AwVi.com)

$h_1$(AwVi.com) = 2

$h_2$(AwVi.com) = 1

**INSERT**(ViSt.com)

$h_1$(ViSt.com) = 3

$h_2$(ViSt.com) = 2

$h_1$(BBC.com) = 2

$h_2$(BBC.com) = 4
Approach 3: build a bloom filter

We still maintain a bit string $B$ of some length $m < |U|$

Now we have $r$ hash functions: $h_1, h_2, \ldots, h_r$  
(we will choose $r$ and $m$ later)

Each hash function $h_i$ maps a key $k$, to an integer $h_i(k)$ between 1 and $m$

$\text{INSERT}(k)$ sets $B[h_i(k)] = 1$ for all $i$ between 1 and $r$

$\text{MEMBER}(k)$ returns ‘yes’ if and only if for all $i, B[h_i(k)] = 1$

Imagine that $m = 4, r = 2$ and

Example:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

$\text{INSERT}(\text{AwVi.com})$

$\text{INSERT}(\text{ViSt.com})$

$h_1(\text{AwVi.com}) = 2 \quad h_2(\text{AwVi.com}) = 1$

$h_1(\text{ViSt.com}) = 3 \quad h_2(\text{ViSt.com}) = 2$

$h_1(\text{BBC.com}) = 2 \quad h_2(\text{BBC.com}) = 4$
Approach 3: build a bloom filter

We still maintain a bit string $B$ of some length $m < |U|$

Now we have $r$ hash functions: $h_1, h_2, \ldots, h_r$ (we will choose $r$ and $m$ later)

Each hash function $h_i$ maps a key $k$, to an integer $h_i(k)$ between 1 and $m$

**INSERT**$(k)$ sets $B[h_i(k)] = 1$ for all $i$ between 1 and $r$
**MEMBER**$(k)$ returns ‘yes’ if and only if for all $i$, $B[h_i(k)] = 1$

Imagine that $m = 4$, $r = 2$ and

**Example:**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**INSERT***(AwVi.com)*

**INSERT***(ViSt.com)*

**MEMBER**(BBC.com) - returns ‘no’

$h_1$(AwVi.com) = 2   $h_2$(AwVi.com) = 1
$h_1$(ViSt.com) = 3   $h_2$(ViSt.com) = 2
$h_1$(BBC.com) = 2    $h_2$(BBC.com) = 4
Approach 3: build a bloom filter

We still maintain a bit string $B$ of some length $m < |U|$

Now we have $r$ hash functions: $h_1, h_2, \ldots, h_r$ (we will choose $r$ and $m$ later)

Each hash function $h_i$ maps a key $k$, to an integer $h_i(k)$ between 1 and $m$

**Example:**

Imagine that $m = 4$, $r = 2$ and

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**INSERT(AwVi.com)**

**INSERT(ViSt.com)**

**MEMBER(BBC.com)** - returns ‘no’

Much better!
Approach 3: build a bloom filter

We still maintain a bit string $B$ of some length $m < |U|$

Now we have $r$ hash functions: $h_1, h_2, \ldots, h_r$ (we will choose $r$ and $m$ later)

Each hash function $h_i$ maps a key $k$, to an integer $h_i(k)$ between 1 and $m$

\[\text{INSERT}(k) \text{ sets } B[h_i(k)] = 1\]
for all $i$ between 1 and $r$

\[\text{MEMBER}(k) \text{ returns ‘yes’ if and only if}\]
for all $i$, $B[h_i(k)] = 1$

Imagine that $m = 4$, $r = 2$ and

Example:

\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
1 & 1 & 1 & 0 \\
\end{array}
\]

\[\text{INSERT(AwVi.com)} \]
\[\text{INSERT(ViSt.com)} \]

\[\text{MEMBER(BBC.com)} - \text{returns ‘no’} \quad \text{Much better! (not convinced?)}\]
Approach 3: build a bloom filter

We still maintain a bit string $B$ of some length $m < |U|$

Now we have $r$ hash functions: $h_1, h_2, \ldots, h_r$ (we will choose $r$ and $m$ later)

Each hash function $h_i$ maps a key $k$, to an integer $h_i(k)$ between 1 and $m$

$\textsc{Insert}(k)$ sets $B[h_i(k)] = 1$ for all $i$ between 1 and $r$

$\textsc{Member}(k)$ returns ‘yes’ if and only if for all $i$, $B[h_i(k)] = 1$

For every key $k \in U$, the value of each $h_i(k)$ is chosen uniformly at random:

that is, the probability that $h_i(k) = j$ is $\frac{1}{m}$ for all $j$ between 1 and $m$

(all positions are equally likely)
Approach 3: build a bloom filter

We still maintain a bit string $B$ of some length $m < |U|$

Now we have $r$ hash functions: $h_1, h_2, \ldots, h_r$ (we will choose $r$ and $m$ later)

Each hash function $h_i$ maps a key $k$, to an integer $h_i(k)$ between 1 and $m$

<table>
<thead>
<tr>
<th>INSERT($k$) sets $B[h_i(k)] = 1$</th>
<th>MEMBER($k$) returns ‘yes’ if and only if for all $i$, $B[h_i(k)] = 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>for all $i$ between 1 and $r$</td>
<td></td>
</tr>
</tbody>
</table>

For every key $k \in U$, the value of each $h_i(k)$ is chosen uniformly at random:

that is, the probability that $h_i(k) = j$ is $\frac{1}{m}$ for all $j$ between 1 and $m$

(all positions are equally likely)

but what is the probability of a wrong answer?
What is the probability of an error?

Assume we have already inserted $n$ keys into the bloom filter.

Further, we have just called $\text{MEMBER}(k)$ for some key $k$ not in $S$.

This will check whether $B[h_i(k)] = 1$ for all $j = 1, 2, \ldots r$. 
What is the probability of an error?

Assume we have already **INSERTED** \( n \) keys into the bloom filter.

Further, we have just called **MEMBER** \(( k )\) for some key \( k \) **not** in \( S \)

this will check whether \( B[h_i(k)] = 1 \) for all \( j = 1, 2, \ldots r \)

*This is the same as checking whether \( r \) randomly chosen bits of \( B \) all equal 1*
What is the probability of an error?

Assume we have already INSERTED $n$ keys into the bloom filter.

Further, we have just called MEMBER$(k)$ for some key $k$ not in $S$.

This will check whether $B[h_i(k)] = 1$ for all $j = 1, 2, \ldots r$.

This is the same as checking whether $r$ randomly chosen bits of $B$ all equal 1.

We will now show that there is only a small probability of this happening.
What is the probability of an error?

Assume we have already **INSERTED** $n$ keys into the bloom filter

Further, we have just called MEMBER($k$) for some key $k$ **not** in $S$
	his will check whether $B[h_i(k)] = 1$ for all $j = 1, 2, \ldots r$

*This is the same as checking whether $r$ randomly chosen bits of $B$ all equal 1*

We will now show that there is only a small probability of this happening

As there are at most $n$ keys in the filter,

at most $nr$ bits of $B$ are set to 1
What is the probability of an error?

Assume we have already \textbf{INSERTED} \( n \) keys into the bloom filter

Further, we have just called \textbf{MEMBER}(\( k \)) for some key \( k \) \textbf{not} in \( S \)

this will check whether \( B[h_j(k)] = 1 \) for all \( j = 1, 2, \ldots r \)

\textit{This is the same as checking whether \( r \) randomly chosen bits of \( B \) all equal 1}

We will now show that there is only a small probability of this happening

As there are at most \( n \) keys in the filter,

at most \( nr \) bits of \( B \) are set to 1

\textit{(each \textbf{INSERT} sets at most \( r \) bits to 1)}
What is the probability of an error?

Assume we have already **INSERTED** \( n \) keys into the bloom filter.

Further, we have just called **MEMBER**(\( k \)) for some key \( k \) **not** in \( S \)

this will check whether \( B[h_i(k)] = 1 \) for all \( j = 1, 2, \ldots r \)

*This is the same as checking whether \( r \) randomly chosen bits of \( B \) all equal 1*

We will now show that there is only a small probability of this happening.

As there are at most \( n \) keys in the filter,

at most \( nr \) bits of \( B \) are set to 1

*each **INSERT** sets at most \( r \) bits to 1*
What is the probability of an error?

Assume we have already INSERTED \( n \) keys into the bloom filter.

Further, we have just called MEMBER(\( k \)) for some key \( k \) not in \( S \).

This will check whether \( B[h_i(k)] = 1 \) for all \( j = 1, 2, \ldots r \).

This is the same as checking whether \( r \) randomly chosen bits of \( B \) all equal 1.

We will now show that there is only a small probability of this happening.

As there are at most \( n \) keys in the filter, at most \( nr \) bits of \( B \) are set to 1.

(Each INSERT sets at most \( r \) bits to 1)

So the fraction of bits set to 1 is at most \( \frac{nr}{m} \).
What is the probability of an error?

Assume we have already inserted \( n \) keys into the bloom filter.

Further, we have just called \( \text{MEMBER}(k) \) for some key \( k \) not in \( S \) this will check whether \( B[h_i(k)] = 1 \) for all \( j = 1, 2, \ldots r \)

\[ \text{This is the same as checking whether } \text{ } r \text{ random chosen bits of } B \text{ all equal } 1 \]

We will now show that there is only a small probability of this happening.

As there are at most \( n \) keys in the filter,

\( \text{at most } nr \text{ bits of } B \text{ are set to } 1 \)

\( (\text{each INSERT sets at most } r \text{ bits to } 1) \)

So the fraction of bits set to 1 is at most \( \frac{nr}{m} \)

so the probability that a randomly chosen bit is 1 is at most \( \frac{nr}{m} \)
What is the probability of an error?

Assume we have already **INSERTED** \( n \) keys into the bloom filter

Further, we have just called **MEMBER**(\( k \)) for some key \( k \) **not** in \( S \)

this will check whether \( B[h_i(k)] = 1 \) for all \( j = 1, 2, \ldots r \)

*This is the same as checking whether \( r \) randomly chosen bits of \( B \) all equal 1*

We will now show that there is only a small probability of this happening

As there are at most \( n \) keys in the filter,

at most \( nr \) bits of \( B \) are set to 1

*(each **INSERT** sets at most \( r \) bits to 1)*

So the fraction of bits set to 1 is at most \( \frac{nr}{m} \)

so the probability that a randomly chosen bit is 1 is at most \( \frac{nr}{m} \)
What is the probability of an error?

Assume we have already **INSERTED** \( n \) keys into the bloom filter

Further, we have just called **MEMBER**\((k)\) for some key \( k \) not in \( S \)

this will check whether \( B[h_i(k)] = 1 \) for all \( j = 1, 2, \ldots r \)

*This is the same as checking whether \( r \) randomly chosen bits of \( B \) all equal 1*

We will now show that there is only a small probability of this happening

As there are at most \( n \) keys in the filter,

at most \( nr \) bits of \( B \) are set to 1

(each **INSERT** sets at most \( r \) bits to 1)

So the fraction of bits set to 1 is at most \( \frac{nr}{m} \)

so the probability that a randomly chosen bit is 1 is at most \( \frac{nr}{m} \)

so the probability that \( r \) randomly chosen bits all equal 1 is at most \( \left( \frac{nr}{m} \right)^r \)
What is the probability of an error?

Assume we have already INSERTED $n$ keys into the bloom filter.

Further, we have just called MEMBER($k$) for some key $k$ not in $S$.

this will check whether $B[h_i(k)] = 1$ for all $j = 1, 2, \ldots r$

*This is the same as checking whether $r$ randomly chosen bits of $B$ all equal 1*

We will now show that there is only a small probability of this happening.

As there are at most $n$ keys in the filter,

at most $nr$ bits of $B$ are set to 1

*(each INSERT sets at most $r$ bits to 1)*

So the fraction of bits set to 1 is at most $\frac{nr}{m}$

so the probability that a randomly chosen bit is 1 is at most $\frac{nr}{m}$

so the probability that $r$ randomly chosen bits all equal 1 is at most $(\frac{nr}{m})^r$
What is the probability of a collision?

We now choose $r$ to minimise this probability...
What is the probability of a collision?

We now choose $r$ to minimise this probability . . .

By differentiating, we can find that $\left( \frac{nr}{m} \right)^r$ is minimised by

letting $r = m/(ne)$ where $e = 2.7813 . . .$
What is the probability of a collision?

We now choose $r$ to minimise this probability.

By differentiating, we can find that $\left( \frac{nr}{m} \right)^r$ is minimised by

letting $r = m/(ne)$ where $e = 2.7813 \ldots$

If we plug this in we get that, the probability of failure, is at most $\left( \frac{1}{e} \right) \frac{m}{n} e \approx (0.69) \frac{m}{n}$.
What is the probability of a collision?

We now choose $r$ to minimise this probability...

By differentiating, we can find that $\left( \frac{nr}{m} \right)^r$ is minimised by

letting $r = m/(ne)$ where $e = 2.7813\ldots$

If we plug this in we get that, the probability of failure, is at most

$\left( \frac{1}{e} \right) \frac{m}{ne} \approx (0.69) \frac{m}{n}$

In particular to achieve a 1% failure probability,

we can set $m \approx 12.52n$ bits
What is the probability of a collision?

We now choose $r$ to minimise this probability...

By differentiating, we can find that $\left( \frac{nr}{m} \right)^r$ is minimised by

letting $r = \frac{m}{(ne)}$ where $e = 2.7813\ldots$

If we plug this in we get that, the probability of failure, is at most $\left( \frac{1}{e} \right) \frac{m}{ne} \approx (0.69) \frac{m}{n}$

In particular to achieve a 1% failure probability,

we can set $m \approx 12.52n$ bits

neither the space nor the failure probability depend on $|U|$
What is the probability of a collision?

We now choose $r$ to minimise this probability...

By differentiating, we can find that \( \left( \frac{nr}{m} \right)^r \) is minimised by

letting \( r = \frac{m}{(ne)} \) where \( e = 2.7813 \ldots \)

If we plug this in we get that, the probability of failure, is at most
\[
\left( \frac{1}{e} \right) \frac{m}{ne} \approx (0.69) \frac{m}{n}
\]

In particular to achieve a 1\% failure probability,

we can set \( m \approx 12.52n \) bits

neither the space nor the failure probability depend on \( |U| \)

if we wanted a better probability, we could use more space
What is the probability of a collision?

We now choose $r$ to minimise this probability…

By differentiating, we can find that $\left( \frac{nr}{m} \right)^r$ is minimised by

letting $r = \frac{m}{(ne)}$ where $e = 2.7813 \ldots$

If we plug this in we get that, the probability of failure, is at most

$$\left( \frac{1}{e} \right)^{\frac{m}{ne}} \approx \left( 0.69 \right)^{\frac{m}{n}}$$

In particular to achieve a 1% failure probability,

we can set $m \approx 12.52n$ bits

neither the space nor the failure probability depend on $|U|$

if we wanted a better probability, we could use more space

This is much better than the $100n$ bits we needed with a single hash function
to achieve the same probability
Bloom filter summary

A **Bloom filter** is a *randomised* data structure for storing a set \( S \) which supports two operations, each in \( O(1) \) time.

The **INSERT** \((k)\) operation inserts the key \( k \) from \( U \) into \( S \) *(it never does this incorrectly)*.

In a bloom filter, the **MEMBER** \((k)\) operation
- always returns ‘yes’ if \( k \in S \)
- however, if \( k \) is not in \( S \)
  - there is a small chance, \( \epsilon \), that it will still say ‘yes’

We have seen that if \( \epsilon = 0.01 \) (1%) the space used is \( m \approx 12.52n \) bits when storing up to \( n \) keys.

By improving the analysis, one can show that only \( \approx 1.44 \log_2(1/\epsilon) \) bits are needed
- \( \approx 9.57n \) bits when \( \epsilon = 0.01 \)
Practical hash functions

We made the unrealistic assumption that each hash function $h_i$ maps a key $k$ to a uniformly random integer between 1 and $m$. 
Practical hash functions

We made the unrealistic assumption that each hash function $h_i$ maps a key $k$ to a uniformly random integer between 1 and $m$.

In practice, we pick each hash function $h_i$ randomly from a fixed set of hash functions.
Practical hash functions

We made the unrealistic assumption that each hash function $h_i$ maps a key $k$ to a uniformly random integer between 1 and $m$.

In practice, we pick each hash function $h_i$ randomly from a fixed set of hash functions.

One way of doing this for integer keys $k$ (see CLRS 11.3.3) is the following:

For each $i$:

1. Pick a prime number $p > |U|$.
2. Pick random integers $a \in \{1, \ldots, p - 1\}, b \in \{0, \ldots, p - 1\}$.
3. Let $h_i$ be defined by $h_i(k) = 1 + ((ak + b) \mod p) \mod m$. 
Practical hash functions

We made the unrealistic assumption that each hash function $h_i$ maps a key $k$ to a uniformly random integer between 1 and $m$.

In practice, we pick each hash function $h_i$ randomly from a fixed set of hash functions.

One way of doing this for integer keys $k$ (see CLRS 11.3.3) is the following:

For each $i$:

1. Pick a prime number $p > |U|$.
2. Pick random integers $a \in \{1, \ldots, p - 1\}$, $b \in \{0, \ldots, p - 1\}$.
3. Let $h_i$ be defined by $h_i(k) = 1 + ((ak + b) \mod p) \mod m$.

Some number theory can be used to prove that this set of hash functions is “pseudorandom” in some sense; however, technically they are not “random enough” for our analysis above to go through.
Practical hash functions

We made the unrealistic assumption that each hash function $h_i$ maps a key $k$ to a uniformly random integer between 1 and $m$.

In practice, we pick each hash function $h_i$ randomly from a fixed set of hash functions.

One way of doing this for integer keys $k$ (see CLRS 11.3.3) is the following:

For each $i$:

1. Pick a prime number $p > |U|$.
2. Pick random integers $a \in \{1, \ldots, p - 1\}, b \in \{0, \ldots, p - 1\}$.
3. Let $h_i$ be defined by $h_i(k) = 1 + ((ak + b) \mod p) \mod m$.

Some number theory can be used to prove that this set of hash functions is “pseudorandom” in some sense; however, technically they are not “random enough” for our analysis above to go through.

Nevertheless, in practice hash functions like this are very effective.
Bloom filter summary

A **Bloom filter** is a *randomised* data structure for storing a set \( S \) which supports two operations, each in \( O(1) \) time.

The **INSERT\( (k) \)** operation inserts the key \( k \) from \( U \) into \( S \)

\(\text{(it never does this incorrectly)}\)

In a bloom filter, the **MEMBER\( (k) \)** operation

always returns ‘yes’ if \( k \in S \)

however, if \( k \) is not in \( S \)

there is a small chance, \( \epsilon \), that it will still say ‘yes’

We have seen that if \( \epsilon = 0.01 \) (1\%) the the space used is \( m \approx 12.52n \) bits when storing up to \( n \) keys

By improving the analysis, one can show that only \( \approx 1.44 \log_2(1/\epsilon) \) bits are needed

\( \approx 9.57n \) bits when \( \epsilon = 0.01 \)}