Data Structures and Algorithms – COMS21103

Bloom Filters

Raphaël Clifford

(Slides by Benjamin Sach and Ashley Montanaro)

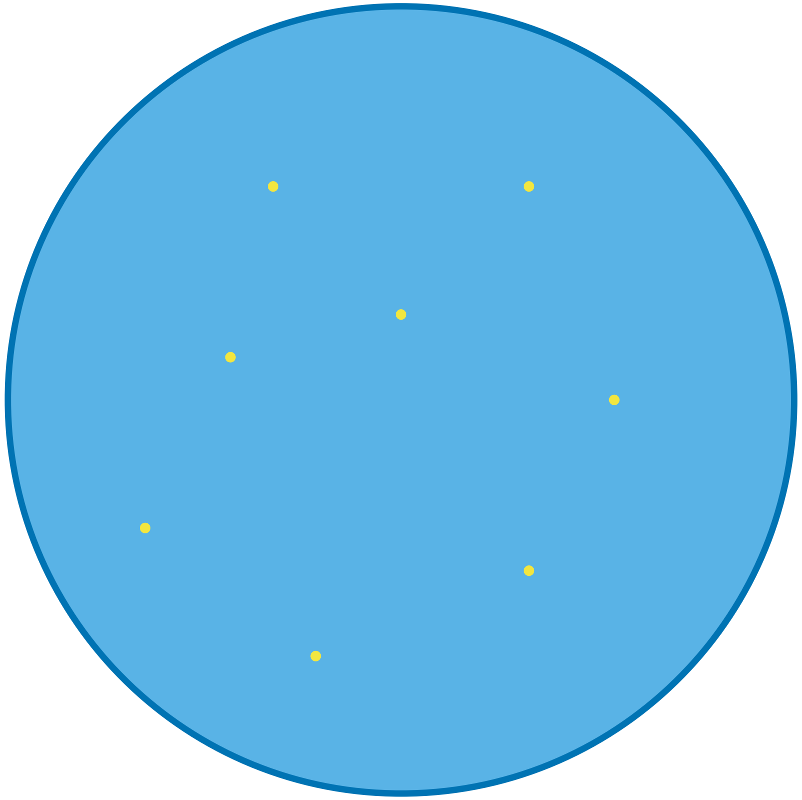
University of BRISTOL

Introduction

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

$\textsc{Insert}(k)$ - inserts the key k from U into S

$\textsc{Member}(k)$ - output ‘yes’ if $k \in S$ *and ‘no’ otherwise*



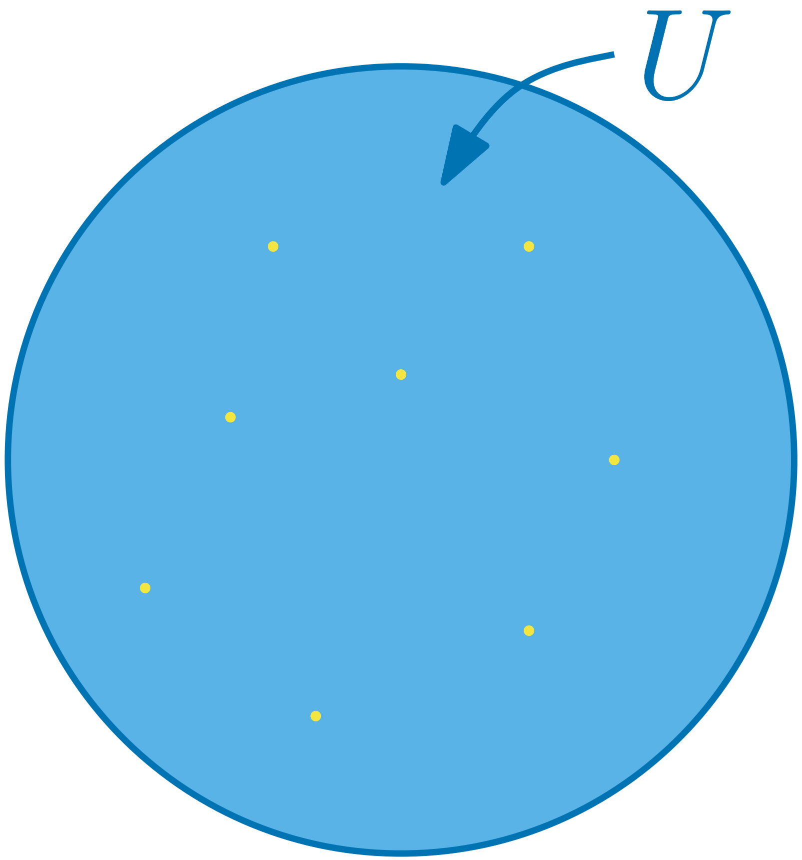
U is the universe, containing *all possible keys*

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:



$\textsc{Insert}(k)$ - inserts the key k from U into S

$\textsc{Member}(k)$ - 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

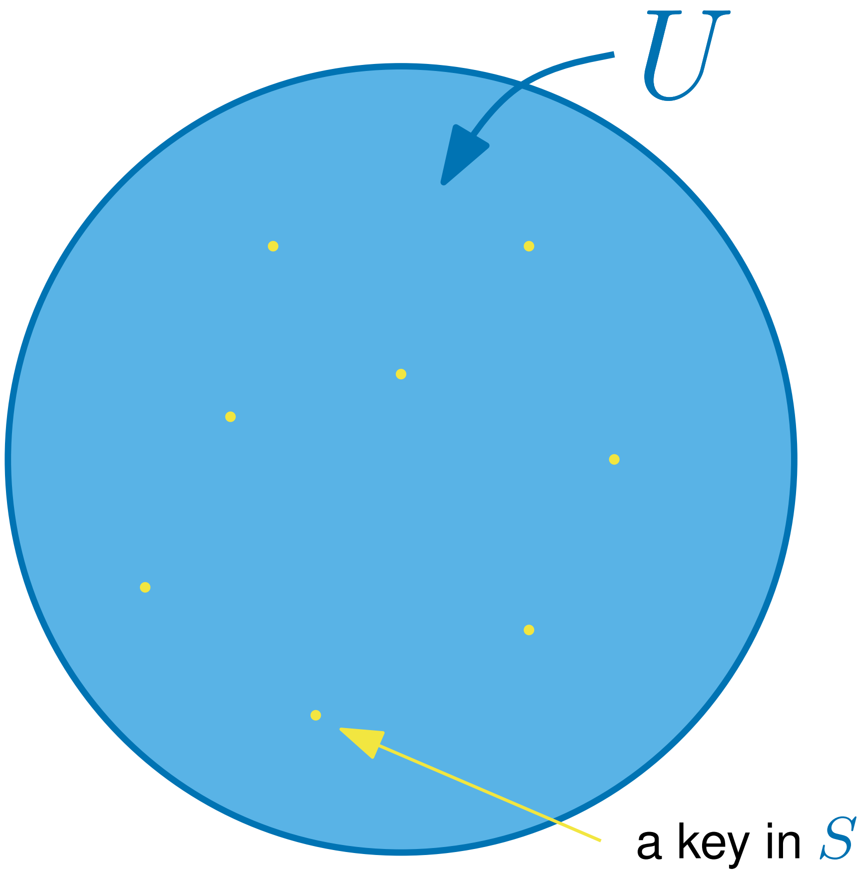
In this lecture we are interested in space efficient data structures for storing a set S which support only two, basic operations:

$\textsc{Insert}(k)$ - inserts the key k from U into S

$\textsc{Member}(k)$ - 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

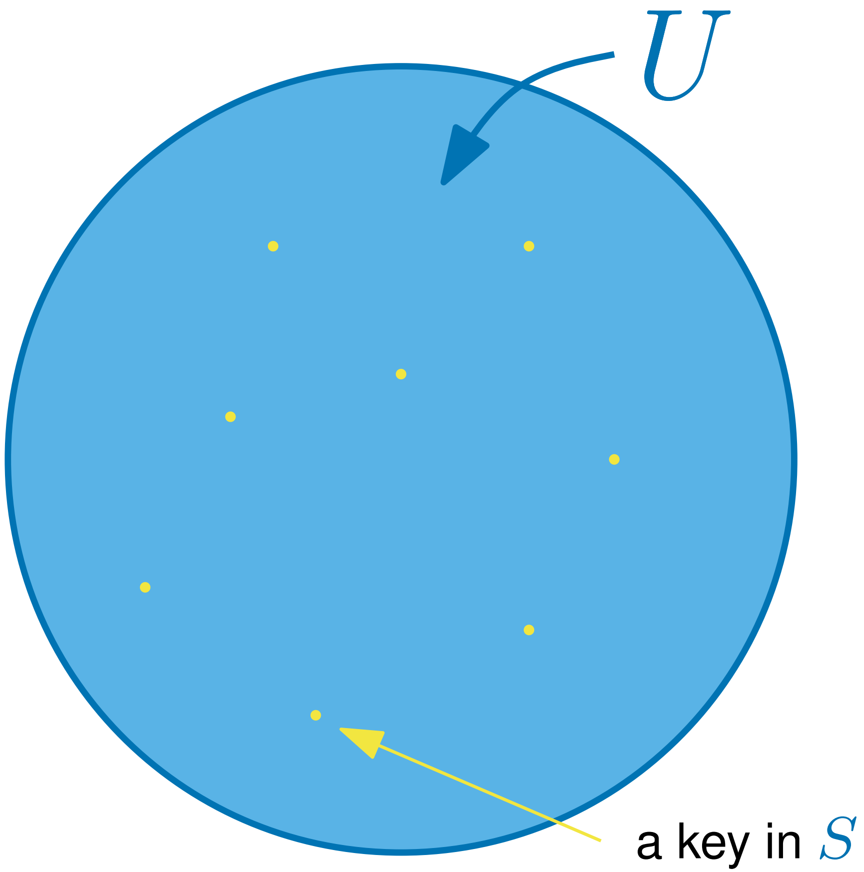
In this lecture we are interested in space efficient data structures for storing a set S which support only two, basic operations:

$\textsc{Insert}(k)$ - inserts the key k from U into S

$\textsc{Member}(k)$ - 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.

$\textsc{Insert}(\texttt{www.AwfulVirus.com})$

$\textsc{Insert}(\texttt{www.VirusStore.com})$

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.

$\textsc{Insert}(\texttt{www.AwfulVirus.com})$

$\textsc{Insert}(\texttt{www.VirusStore.com})$

Disclaimer: I take no responsability 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.

INSERT(www.AwfulVirus.com)

INSERT(www.VirusStore.com)

MEMBER(www.BBC.co.uk) - returns 'no'

Disclaimer: I take no responsability 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.

INSERT(`www.AwfulVirus.com`)

INSERT(`www.VirusStore.com`)

MEMBER(`www.BBC.co.uk`) - returns 'no'

MEMBER(`www.VirusStore.com`) - returns 'yes'

Disclaimer: I take no responsability 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.

INSERT(`www.AwfulVirus.com`)

INSERT(`www.VirusStore.com`)

MEMBER(`www.BBC.co.uk`) - returns ‘no’

MEMBER(`www.VirusStore.com`) - 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`)

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’

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' ?!

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

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

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 $k \in S$

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'

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?

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*

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 $\textsc{Insert}(k)$ 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

- 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 $\textsc{Insert}(k)$ 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

- 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, lets 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, lets 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, lets 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:

	1	2	3	4	5	6	7	8	9	10
B	0	0	1	0	0	1	0	1	0	0

$|U|$

Approach 1: build an array

Before discussing Bloom filters, lets 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:

	1	2	3	4	5	6	7	8	9	10
B	0	0	1	0	0	1	0	1	0	0

$|U|$

Approach 1: build an array

Before discussing Bloom filters, lets 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:

	1	2	3	4	5	6	7	8	9	10
B	0	0	1	0	0	1	0	1	0	0

$|U|$

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

Approach 1: build an array

Before discussing Bloom filters, lets 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:

	1	2	3	4	5	6	7	8	9	10
B	0	0	1	0	0	1	0	1	0	0

$|U|$

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, lets consider a naive approach using an array. . .

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

We could maintain a bit string B

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

Example:

	1	2	3	4	5	6	7	8	9	10
B	0	0	1	0	0	1	0	1	0	0

$|U|$

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 the application we have seen

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:

$$B \quad \begin{array}{ccc} 1 & 2 & 3 \\ \hline \multicolumn{1}{|c|}{0} & \multicolumn{1}{c|}{0} & \multicolumn{1}{c|}{0} \\ \hline \end{array}$$

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:

	1	2	3
B	0	0	0

Imagine that $m = 3$ and

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

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

$h(\texttt{www.BBC.co.uk}) = 3$

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

$\textsc{Insert}(k)$ sets $B[h(k)] = 1$

Example:

$$B \quad \begin{array}{ccc} 1 & 2 & 3 \\ \hline \boxed{0} & \boxed{0} & \boxed{0} \end{array}$$

Imagine that $m = 3$ and

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

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

$h(\texttt{www.BBC.co.uk}) = 3$

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

$\textsc{Insert}(k)$ sets $B[h(k)] = 1$

$\textsc{Member}(k)$ returns 'yes' if $B[h(k)] = 1$ and 'no' if $B[h(k)] = 0$

Example:

$$B \quad \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{array}{|c|c|c|} \hline 0 & 0 & 0 \\ \hline \end{array} \end{array}$$

Imagine that $m = 3$ and

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

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

$h(\texttt{www.BBC.co.uk}) = 3$

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

$\textsc{Insert}(k)$ sets $B[h(k)] = 1$

$\textsc{Member}(k)$ returns 'yes' if $B[h(k)] = 1$

and 'no' if $B[h(k)] = 0$

Example:

	1	2	3
B	0	0	0

$\textsc{Insert}(\texttt{www.AwfulVirus.com})$

Imagine that $m = 3$ and

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

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

$h(\texttt{www.BBC.co.uk}) = 3$

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

$\textsc{Insert}(k)$ sets $B[h(k)] = 1$

$\textsc{Member}(k)$ returns 'yes' if $B[h(k)] = 1$
and 'no' if $B[h(k)] = 0$

Example:

	1	2	3
B	0	1	0

$\textsc{Insert}(\texttt{www.AwfulVirus.com})$

Imagine that $m = 3$ and

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

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

$h(\texttt{www.BBC.co.uk}) = 3$

University of BRISTOL

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

$\textsc{Insert}(k)$ sets $B[h(k)] = 1$

$\textsc{Member}(k)$ returns 'yes' if $B[h(k)] = 1$
and 'no' if $B[h(k)] = 0$

Example:

	1	2	3
B	0	1	0

$\textsc{Insert}(\texttt{www.AwfulVirus.com})$

$\textsc{Insert}(\texttt{www.VirusStore.com})$

Imagine that $m = 3$ and

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

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

$h(\texttt{www.BBC.co.uk}) = 3$

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

$\textsc{Insert}(k)$ sets $B[h(k)] = 1$ $\qquad$ $\textsc{Member}(k)$ returns ‘yes’ if $B[h(k)] = 1$

and ‘no’ if $B[h(k)] = 0$

Example:

$$B \quad \begin{array}{ccc} 1 & 2 & 3 \\ \hline \multicolumn{1}{|c|}{0} & \multicolumn{1}{c|}{1} & \multicolumn{1}{c|}{1} \\ \hline \end{array}$$

$\textsc{Insert}(\texttt{www.AwfulVirus.com})$

$\textsc{Insert}(\texttt{www.VirusStore.com})$

Imagine that $m = 3$ and

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

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

$h(\texttt{www.BBC.co.uk}) = 3$

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

$\textsc{Insert}(k)$ sets $B[h(k)] = 1$ $\qquad$ $\textsc{Member}(k)$ returns 'yes' if $B[h(k)] = 1$

and 'no' if $B[h(k)] = 0$

Example:

	1	2	3
B	0	1	1

Imagine that $m = 3$ and

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

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

$h(\texttt{www.BBC.co.uk}) = 3$

$\textsc{Insert}(\texttt{www.AwfulVirus.com})$

$\textsc{Insert}(\texttt{www.VirusStore.com})$

$\textsc{Member}(\texttt{www.BBC.co.uk})$ - 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

$\textsc{Insert}(k)$ sets $B[h(k)] = 1$

$\textsc{Member}(k)$ returns 'yes' if $B[h(k)] = 1$ and 'no' if $B[h(k)] = 0$

Example:

	1	2	3
B	0	1	1

$\textsc{Insert}(\texttt{www.AwfulVirus.com})$

$\textsc{Insert}(\texttt{www.VirusStore.com})$

$\textsc{Member}(\texttt{www.BBC.co.uk})$ - returns 'yes'

Imagine that $m = 3$ and

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

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

$h(\texttt{www.BBC.co.uk}) = 3$

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

(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 $\textsc{Member}(k)$ 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 $\textsc{Member}(k)$ 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 the hash function h 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 $\textsc{Member}(k)$ 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 the hash function h at random

Important: *h is chosen before any operations happen and never changes*

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 $\textsc{Member}(k)$ 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 the hash function h at random

Important: *h is chosen before any operations happen and never changes*

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

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

(each position is equally likely)

What is the probability of an error?

Assume we have already $\textsc{Inserted}$ n keys into the structure

Further, we have just called

$\textsc{Member}(k)$ for some key k **not** in S

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

What is the probability of an error?

Assume we have already $\textsc{Inserted}$ n keys into the structure

Further, we have just called

$\textsc{Member}(k)$ 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)

What is the probability of an error?

Assume we have already INSERTED n keys into the structure

Further, we have just called

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

What is the probability of an error?

Assume we have already INSERTED n keys into the structure

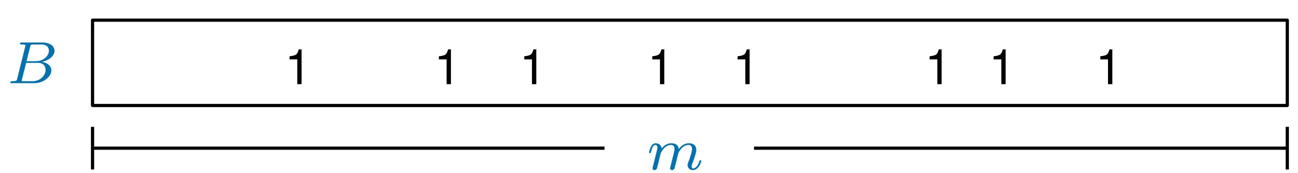
Further, we have just called

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



What is the probability of an error?

Assume we have already INSERTED n keys into the structure

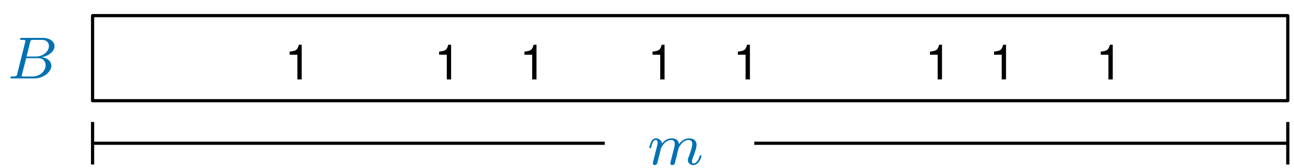
Further, we have just called

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



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

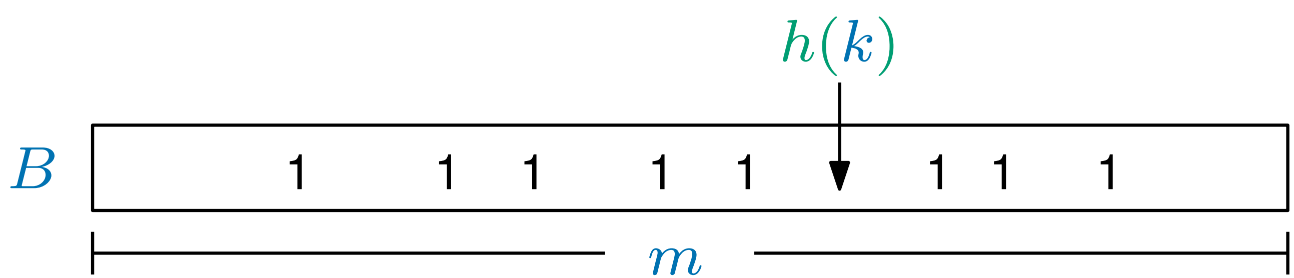
Further, we have just called

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



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

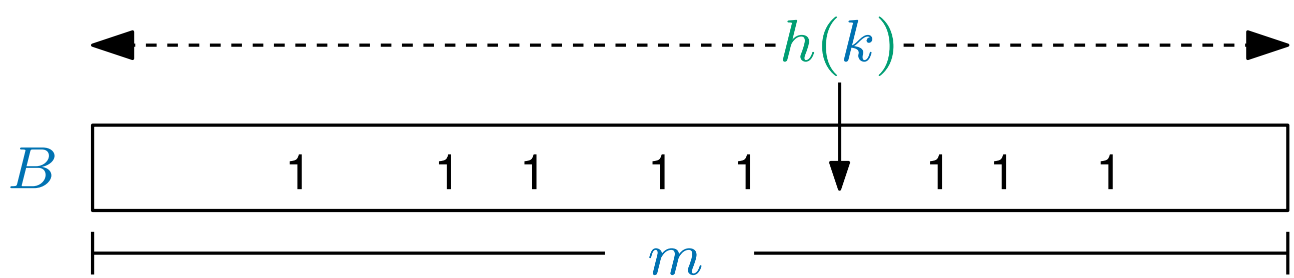
Further, we have just called

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



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

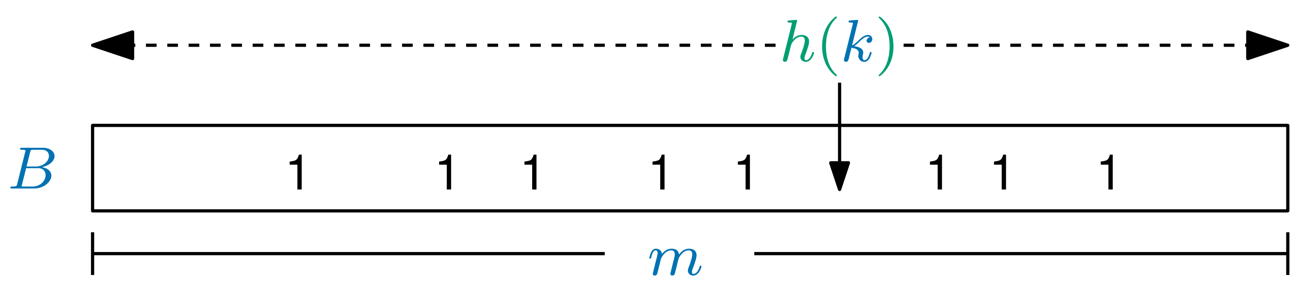
Further, we have just called

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



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

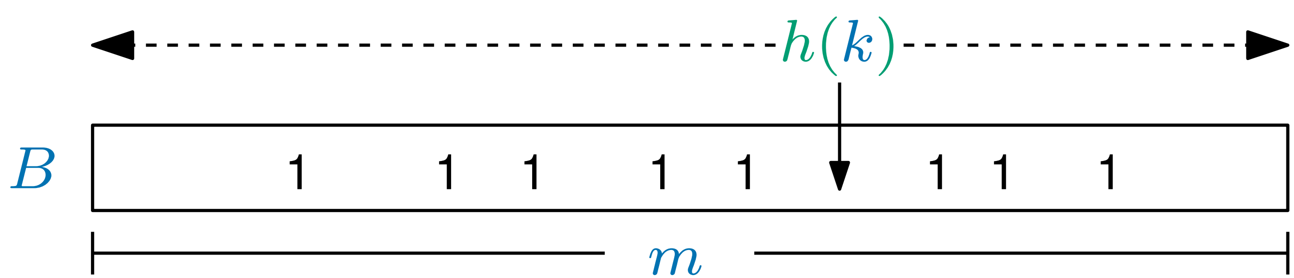
Further, we have just called

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



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

Like in a bloom filter, the $\textsc{Member}(k)$ operation

Approach 2: build a hash table

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

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

Like in a bloom filter, the $\textsc{Member}(k)$ 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 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'

Approach 2: build a hash table

We have developed 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)

Like 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 (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 *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)

Like 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 (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

Approach 2: build a hash table

We have developed 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)

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

Like 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 (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

University of BRISTOL

Approach 2: build a hash table

We have developed 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)

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

Like 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 (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?

we will get *much better* space usage for the same probability

University of BRISTOL

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: $h_1, h_2, \ldots, h_r$ *(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:

1	2	3	4
0	0	0	0

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

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

$h_1(\texttt{BBC.com}) = 2$ $\quad h_2(\texttt{BBC.com}) = 4$

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

---

$\textsc{Insert}(k)$ 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, $B[h_i(k)] = 1$

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

Example:

1	2	3	4
0	0	0	0

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

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

$h_1(\texttt{BBC.com}) = 2$ $\quad h_2(\texttt{BBC.com}) = 4$

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

---

$\textsc{Insert}(k)$ 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, $B[h_i(k)] = 1$

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

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

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

$h_1(\texttt{BBC.com}) = 2$ $h_2(\texttt{BBC.com}) = 4$

Example:

1	2	3	4
0	0	0	0

$\textsc{Insert}(\texttt{AwVi.com})$

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, $B[h_i(k)] = 1$

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

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

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

$h_1(\texttt{BBC.com}) = 2$ $h_2(\texttt{BBC.com}) = 4$

Example:

1	2	3	4
1	1	0	0

$\textsc{Insert}(\texttt{AwVi.com})$

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, $B[h_i(k)] = 1$

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

Example:

1	2	3	4
1	1	0	0

$\textsc{Insert}(\texttt{AwVi.com})$

$\textsc{Insert}(\texttt{ViSt.com})$

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

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

$h_1(\texttt{BBC.com}) = 2$ $\quad h_2(\texttt{BBC.com}) = 4$

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

---

$\textsc{Insert}(k)$ 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, $B[h_i(k)] = 1$

Example:

1	2	3	4
1	1	1	0

$\textsc{Insert}(\texttt{AwVi.com})$

$\textsc{Insert}(\texttt{ViSt.com})$

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

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

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

$h_1(\texttt{BBC.com}) = 2 \quad h_2(\texttt{BBC.com}) = 4$

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

---

$\textsc{Insert}(k)$ 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, $B[h_i(k)] = 1$

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

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

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

$h_1(\texttt{BBC.com}) = 2$ $\quad h_2(\texttt{BBC.com}) = 4$

Example:

1	2	3	4
1	1	1	0

$\textsc{Insert}(\texttt{AwVi.com})$

$\textsc{Insert}(\texttt{ViSt.com})$

$\textsc{Member}(\texttt{BBC.com})$ - returns 'no'

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, $B[h_i(k)] = 1$

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

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

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

$h_1(\texttt{BBC.com}) = 2 \quad h_2(\texttt{BBC.com}) = 4$

Example:

1	2	3	4
1	1	1	0

$\textsc{Insert}(\texttt{AwVi.com})$

$\textsc{Insert}(\texttt{ViSt.com})$

$\textsc{Member}(\texttt{BBC.com})$ - 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

---

$\textsc{Insert}(k)$ 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, $B[h_i(k)] = 1$

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

Example:

1	2	3	4
1	1	1	0

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

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

$h_1(\texttt{BBC.com}) = 2$ $h_2(\texttt{BBC.com}) = 4$

$\textsc{Insert}(\texttt{AwVi.com})$

$\textsc{Insert}(\texttt{ViSt.com})$

$\textsc{Member}(\texttt{BBC.com})$ - returns 'no'

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

---

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, $B[h_i(k)] = 1$

For every key $k \in U$,

the value of each $h_i(k)$ is chosen independently and uniformly at random:

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

(each position is 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

---

$\textsc{Insert}(k)$ 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 independently and uniformly at random:

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

(each position is equally likely)

but what is the probability of a wrong answer?

University of BRISTOL

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 $j = 1, 2, \ldots r$

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

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

University of BRISTOL

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 $j = 1, 2, \ldots r$

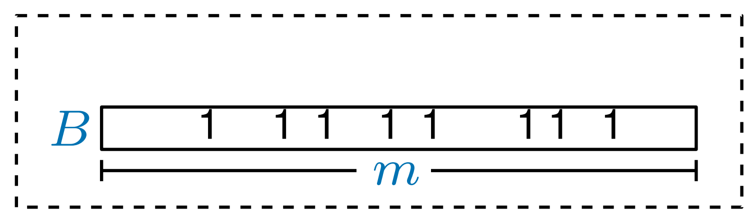
This is the same as checking whether r randomly chosen bits of B all equal 1

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)*



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 $j = 1, 2, \ldots r$

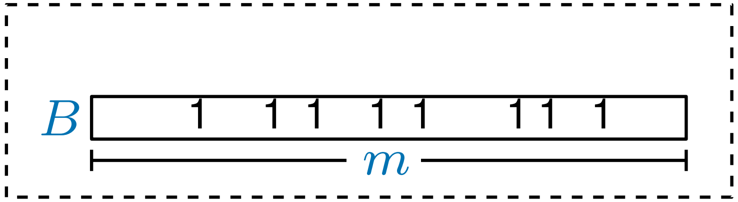
This is the same as checking whether r randomly chosen bits of B all equal 1

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 $\dfrac{nr}{m}$

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 $j = 1, 2, \ldots r$

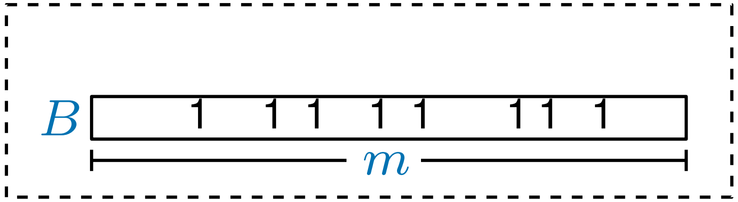
This is the same as checking whether r randomly chosen bits of B all equal 1

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 $\dfrac{nr}{m}$

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

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 $j = 1, 2, \ldots r$

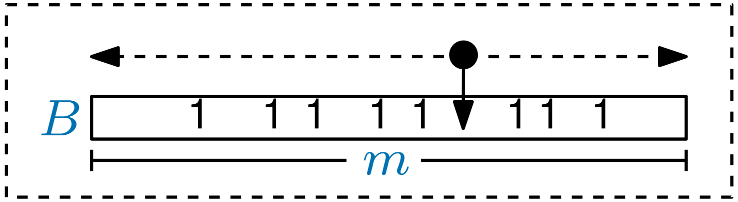
This is the same as checking whether r randomly chosen bits of B all equal 1

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 $\dfrac{nr}{m}$

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

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 $j = 1, 2, \ldots r$

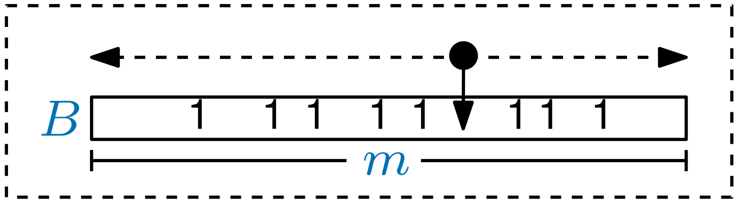
This is the same as checking whether r randomly chosen bits of B all equal 1

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 $j = 1, 2, \ldots r$

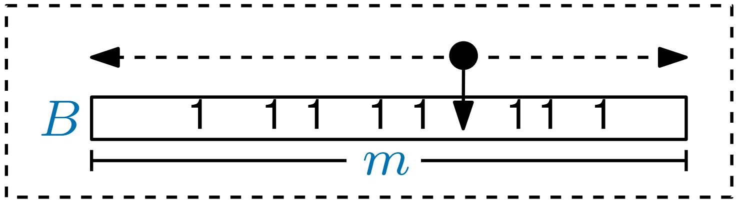
This is the same as checking whether r randomly chosen bits of B all equal 1

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)*



(do this independently r times)

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

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

so the probability that r randomly chosen bits all equal 1 is at most $\left(\dfrac{nr}{m}\right)^r$

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 $e = 2.7813\ldots$

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

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 $r = m/(ne)$ 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 $r = m/(ne)$ 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 $r = m/(ne)$ 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

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 $\textsc{Insert}(k)$ 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, ϵ, 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 impoving 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 is the following: (see CLRS 11.3.3)

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 is the following: (see CLRS 11.3.3)

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 is the following: (see CLRS 11.3.3)

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.

University of BRISTOL

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 $\textsc{Insert}(k)$ 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, ϵ, 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 impoving the analysis, one can show that only $\approx 1.44 \log_2(1/\epsilon)$ bits are needed
($\approx 9.57n$ bits when $\epsilon = 0.01$)