CS11921 - Fall 2024

Algorithm Design & Analysis

Bloom Filters

Ibrahim Albluwi

Did I see you before?

Problem. Given n elements (where n is large), answer queries of the form:
Was element k seen before?

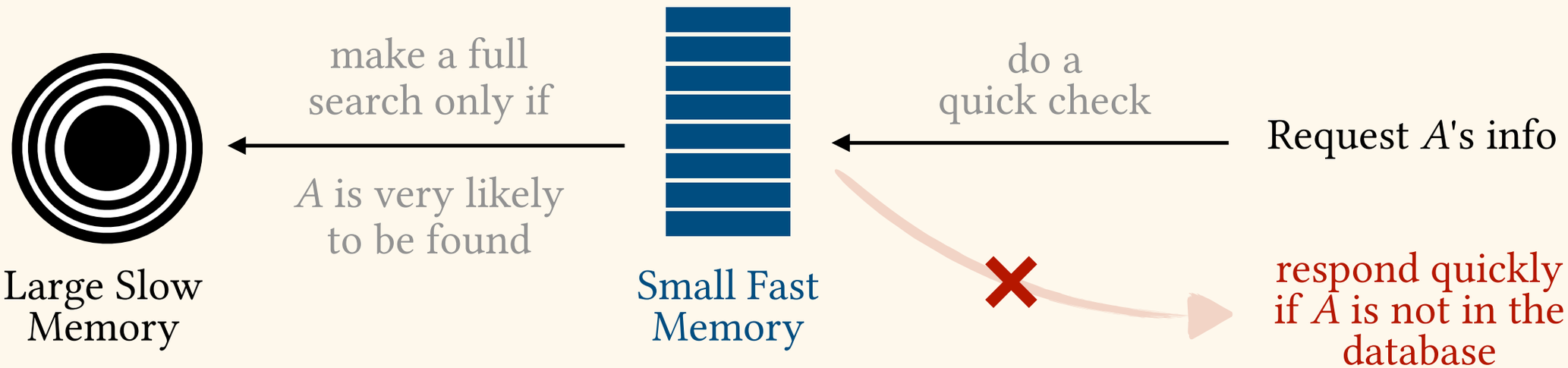
Requirement. Be as memory efficient as possible.
Assumption. A small possibility of incorrect YES answers is acceptable.

Problem. Design a *memory efficient* data structure to perform the following operations:

INSERT(x): Adds x to the set.

CONTAINS(x): Returns FALSE only if x is not in the set.
Returns TRUE if x is *very likely* to be in the set.

Application.



Did I see you before?

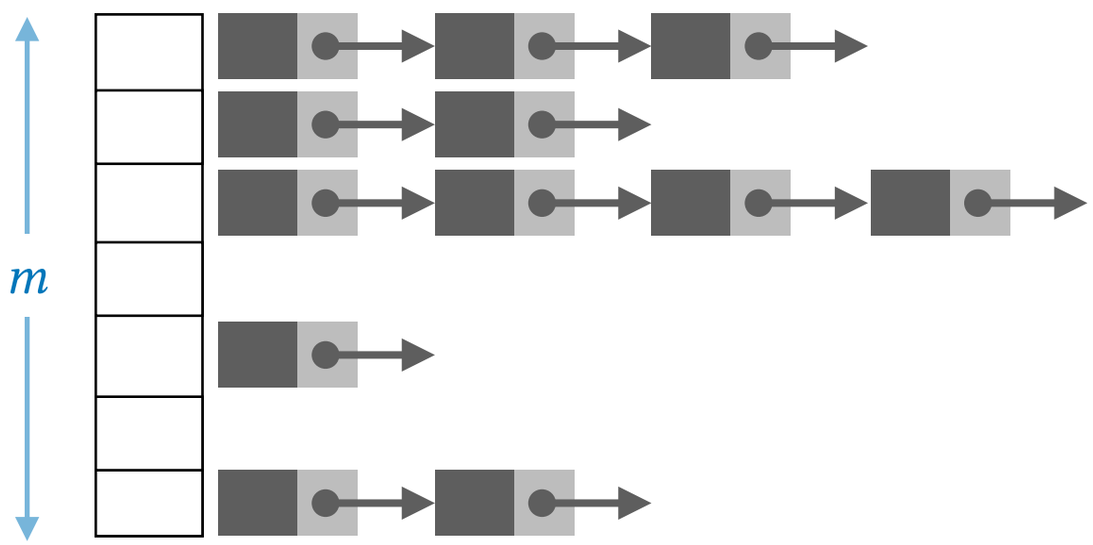
Problem. Design a *memory efficient* data structure to perform the following operations:

`INSERT(x)`: Adds x to the set.

`CONTAINS(x)`: Returns `FALSE` only if x is not in the set.
Returns `TRUE` if x is *very likely* to be in the set.

Solution `#0`. Use a hash table.

- Quick and always returns a correct answer!
- Not memory efficient!



A **C++** implementation requires at least:

- $m \times 64$ bits for each linked list object.
- $n \times 64$ bits for each next pointer.
- $n \times 32$ bits for each element.
  (assuming the elements are integers)

Assuming $m = n/4$, the total is at least:

$\frac{64}{4}n + 96n = 112n$ bits to store n integers.

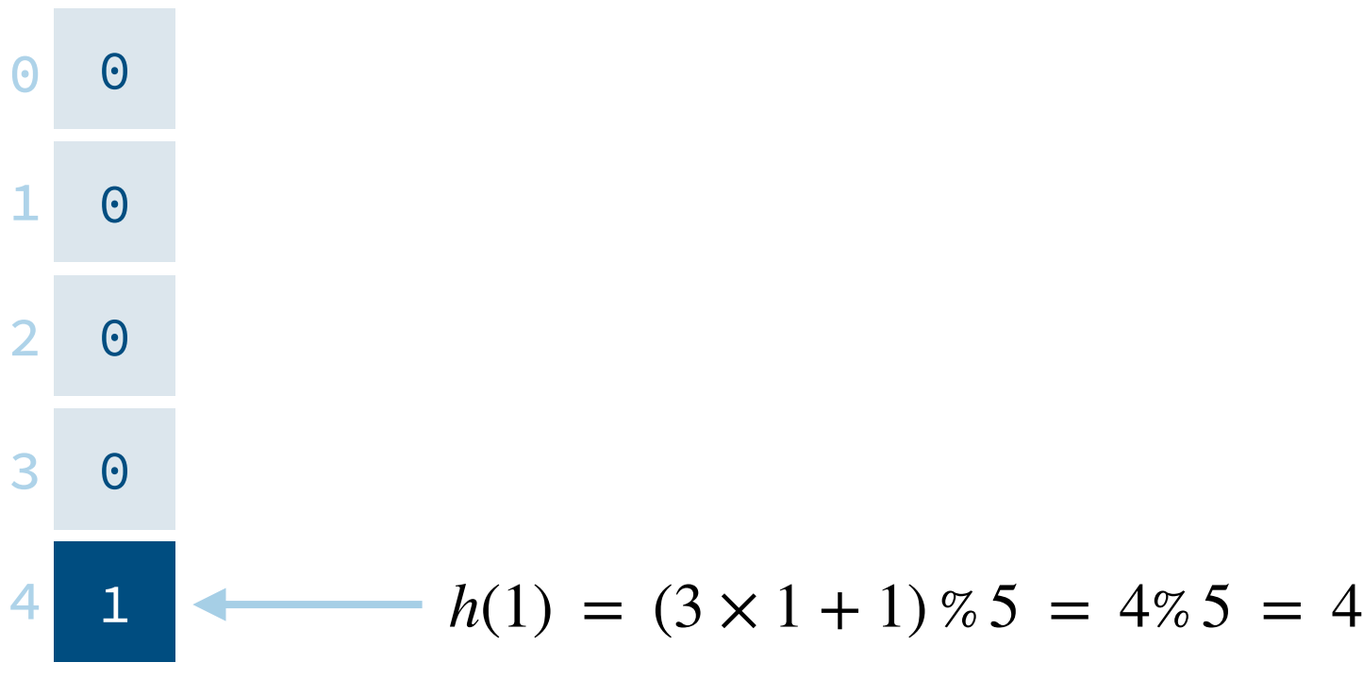
(much more than that is needed
if the elements are strings, for example)

Did I see you before?

Solution # 1. Use a hash table, but don't store the elements, and don't resolve collisions!

```
INSERT(x):     Set    TABLE[ h(x) ]  to  1
CONTAINS(x):   Return TABLE[ h(x) ]  ==  1
```

Example. Insert $\{1, 3, 4\}$ using $h(x) = (3x + 1) \% m$



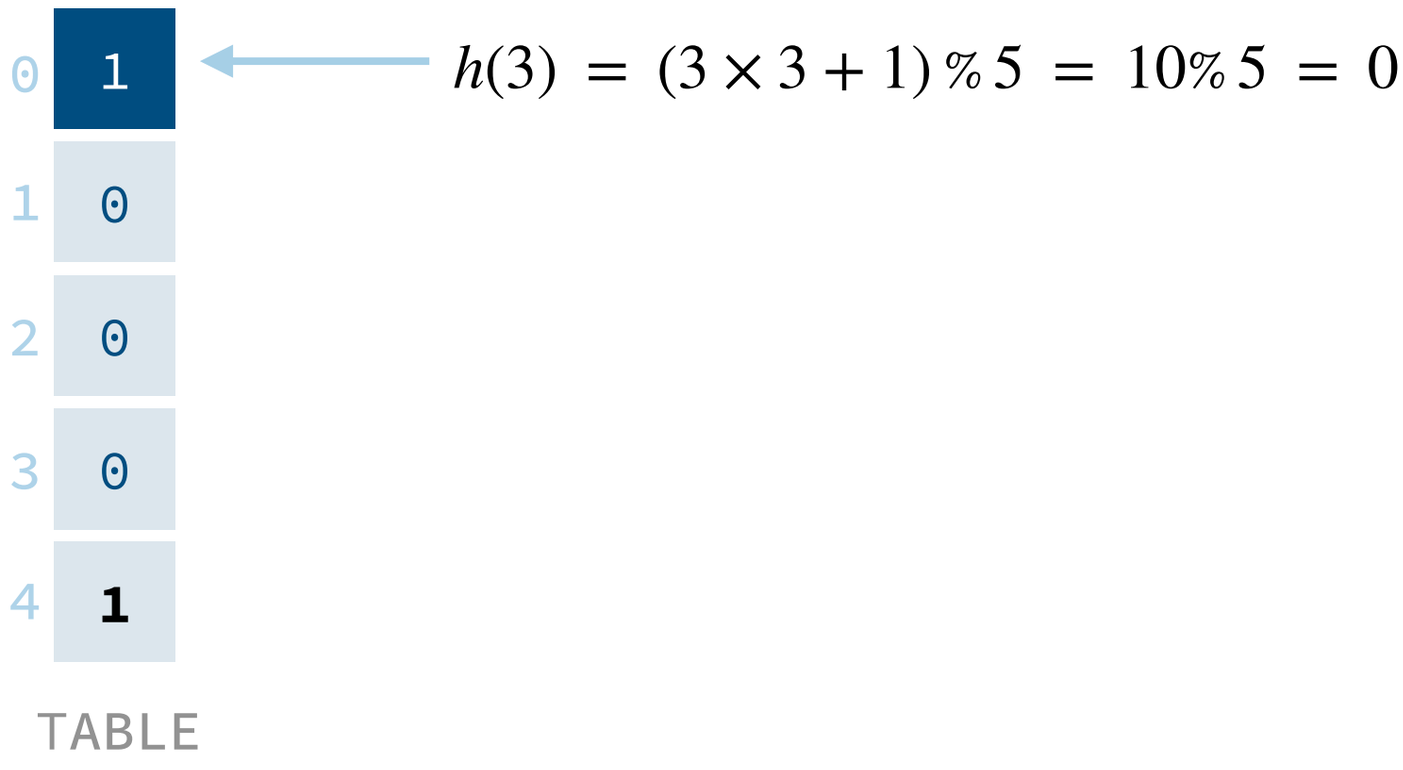
Did I see you before?

Solution # 1. Use a hash table, but don't store the elements, and don't resolve collisions!

INSERT(x): Set TABLE[$h(x)$] to 1

CONTAINS(x): Return TABLE[$h(x)$] == 1

Example. Insert $\{1, 3, 4\}$ using $h(x) = (3x + 1) \% m$

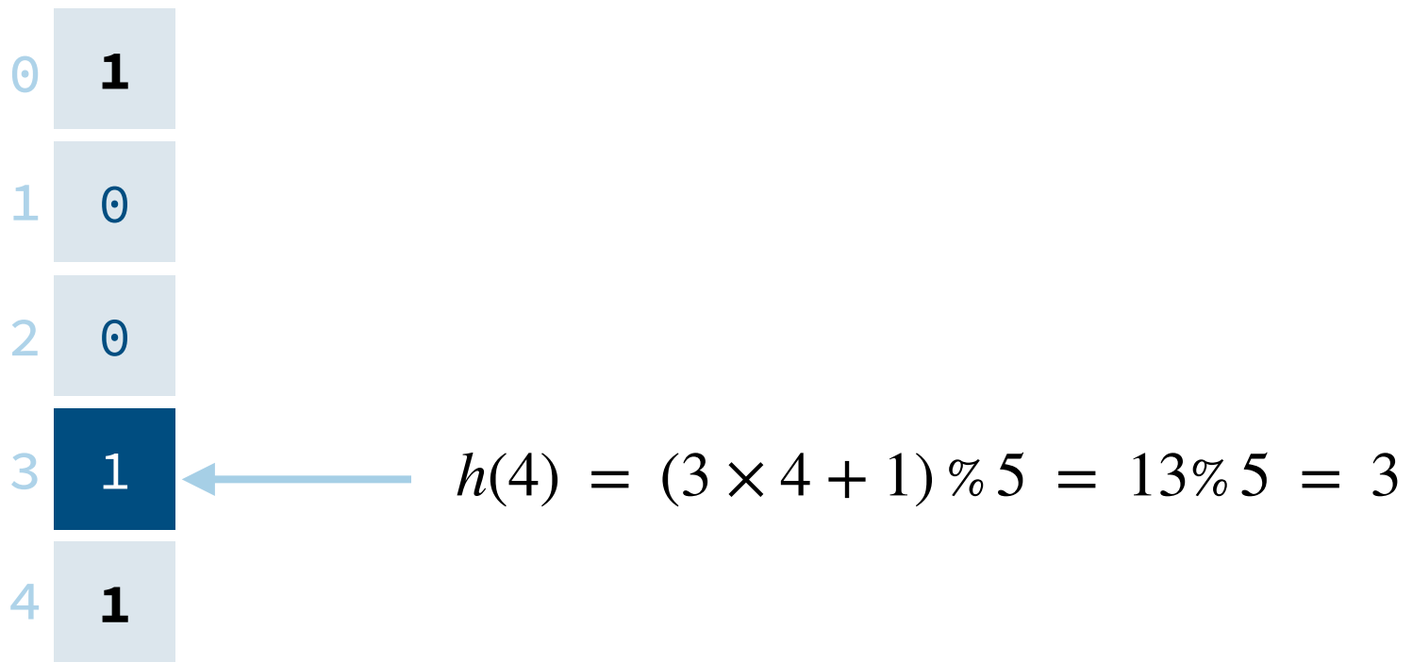


Did I see you before?

Solution # 1. Use a hash table, but don't store the elements, and don't resolve collisions!

```
INSERT(x):     Set    TABLE[ h(x) ]  to  1
CONTAINS(x):   Return TABLE[ h(x) ]  ==  1
```

Example. Insert $\{1, 3, 4\}$ using $h(x) = (3x + 1) \% m$



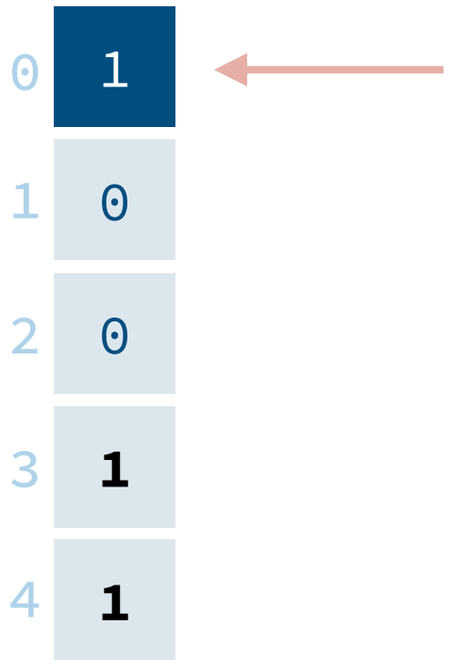
Did I see you before?

Solution # 1. Use a hash table, but don't store the elements, and don't resolve collisions!

INSERT(x): Set TABLE[$h(x)$] to 1

CONTAINS(x): Return TABLE[$h(x)$] == 1

Example. Insert $\{1, 3, 4\}$ using $h(x) = (3x + 1) \% m$



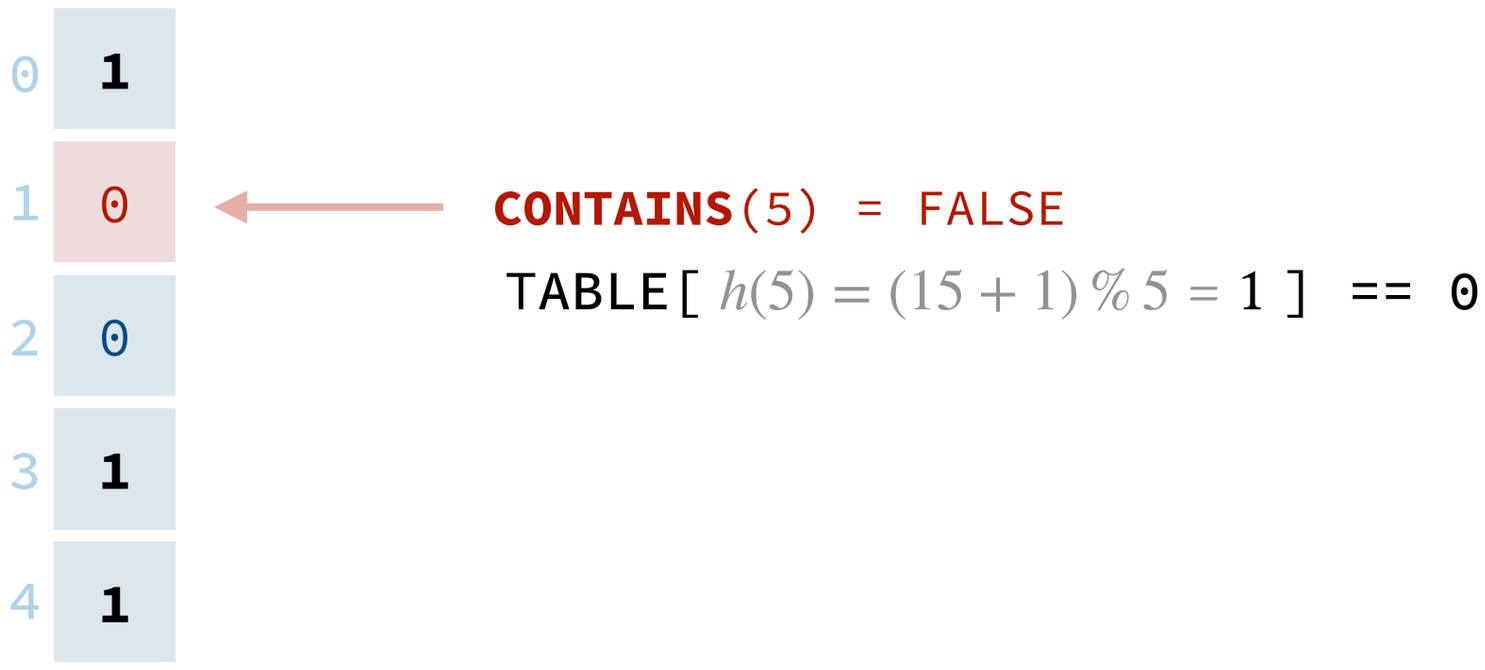
Did I see you before?

Solution # 1. Use a hash table, but don't store the elements, and don't resolve collisions!

INSERT(x): Set TABLE[$h(x)$] to 1

CONTAINS(x): Return TABLE[$h(x)$] == 1

Example. Insert $\{1, 3, 4\}$ using $h(x) = (3x + 1) \% m$



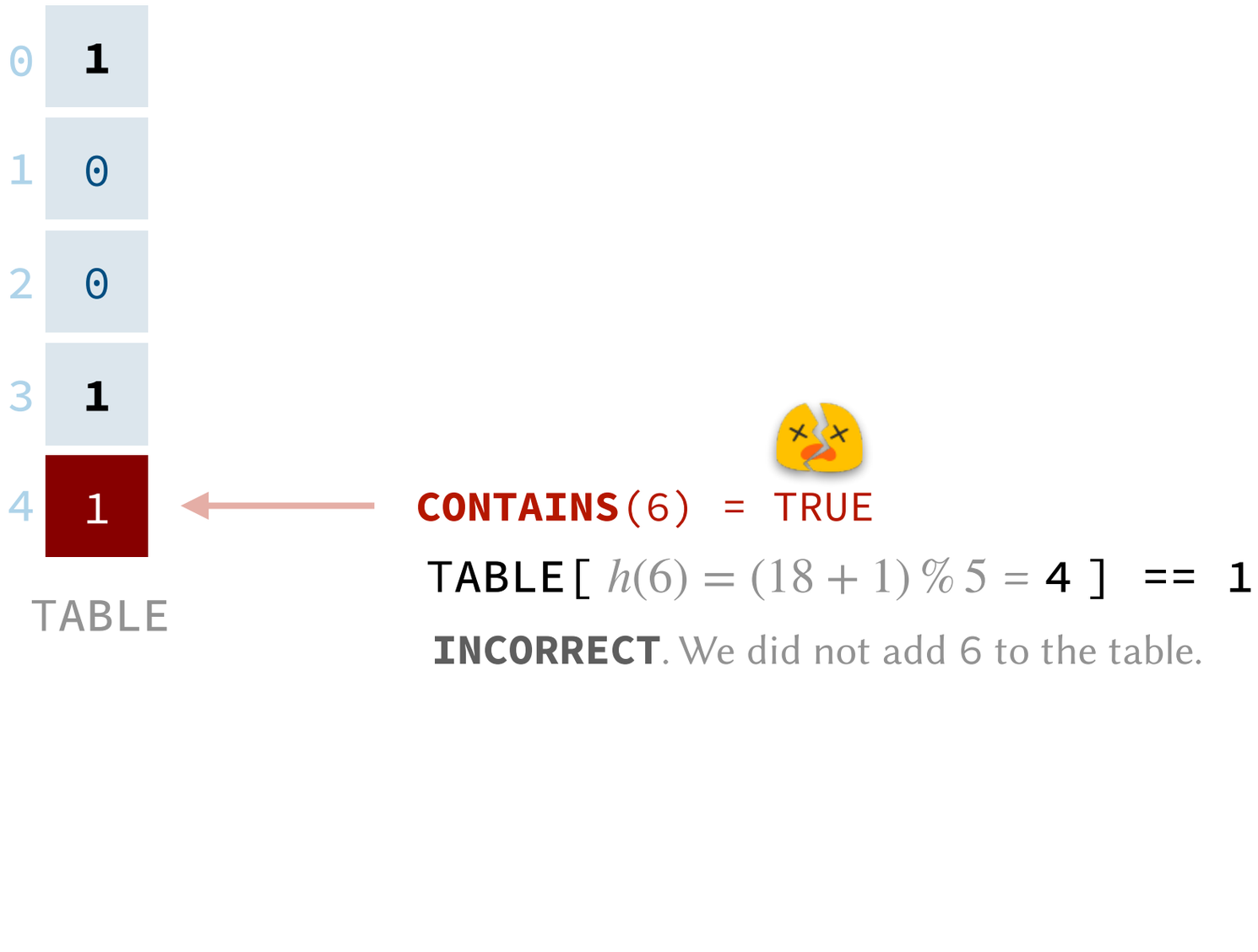
Did I see you before?

Solution # 1. Use a hash table, but don't store the elements, and don't resolve collisions!

INSERT(x): Set TABLE[$h(x)$] to 1

CONTAINS(x): Return TABLE[$h(x)$] == 1

Example. Insert $\{1, 3, 4\}$ using $h(x) = (3x + 1) \% m$



Did I see you before?

Solution # 1. Use a hash table, but don't store the elements, and don't resolve collisions!

INSERT(x): Set TABLE[$h(x)$] to 1

CONTAINS(x): Return TABLE[$h(x)$] == 1

Example. Insert $\{1, 3, 4\}$ using $h(x) = (3x + 1) \% m$

0	**1**
1	0
2	0
3	**1**
4	**1**

TABLE

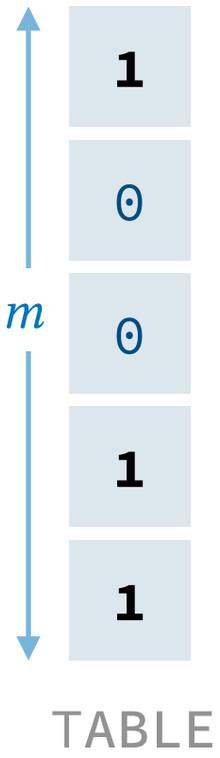
IF TABLE[$h(x)$] == **0**:
x definitely was **not** seen before
(otherwise, it would have been set to 1).

IF TABLE[$h(x)$] == **1**:
x was seen before or another element y
was seen before, where $h(x) = h(y)$.

After n insertions, what is the probability that **CONTAINS**(x) finds TABLE[$h(x)$] to be 1?

Solution # 1 Analysis

Probability of Error. After n insertions, what is the probability that **`CONTAINS(x)`** finds `TABLE[` $h(x)$ `]` to be 1?



After 1 insertion:

$$P(\texttt{TABLE[}\ i\ \texttt{]}\text{ is }1) = \frac{1}{m}$$

$$P(\texttt{TABLE[}\ i\ \texttt{]}\text{ is }\textbf{not}\ 1) = 1-\frac{1}{m}$$

After n insertions:

$$P(\texttt{TABLE[}\ i\ \texttt{]}\text{ is still }\textbf{not}\ 1) = (1-\frac{1}{m})^n = ((1-\frac{1}{m})^m)^{\frac{n}{m}}$$

knowing that $\lim_{x\to\infty}(1-\frac{1}{x})^x = e^{-1}$

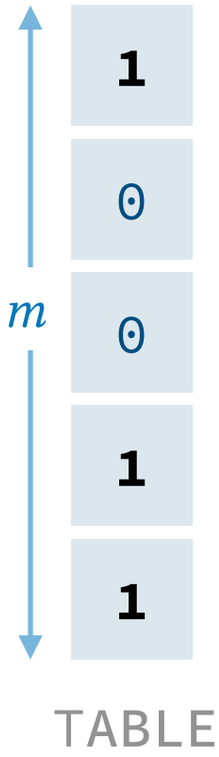
$$P(\texttt{TABLE[}\ i\ \texttt{]}\text{ is still }\textbf{not}\ 1) = e^{-\frac{n}{m}}$$

$$P(\texttt{TABLE[}\ i\ \texttt{]}\text{ is }1) = 1 - e^{-\frac{n}{m}}$$

Is this low enough?

Solution # 1 Analysis

Probability of Error. After n insertions, what is the probability that **`CONTAINS(x)`** finds `TABLE[` $h(x)$ `]` to be 1?



After 1 insertion:

$$P(\texttt{TABLE[}\ i\ \texttt{]}\text{ is }1) = \frac{1}{m}$$

$$P(\texttt{TABLE[}\ i\ \texttt{]}\text{ is }\textbf{not}\ 1) = 1-\frac{1}{m}$$

After n insertions:

$$P(\texttt{TABLE[}\ i\ \texttt{]}\text{ is still }\textbf{not}\ 1) = (1-\frac{1}{m})^n = ((1-\frac{1}{m})^m)^{\frac{n}{m}}$$

knowing that $\lim_{x\to\infty}(1-\frac{1}{x})^x = e^{-1}$

$$P(\texttt{TABLE[}\ i\ \texttt{]}\text{ is still }\textbf{not}\ 1) = e^{-\frac{n}{m}}$$

$$P(\texttt{TABLE[}\ i\ \texttt{]}\text{ is }1) = 1 - e^{-\frac{n}{m}}$$

Assuming $n = 10{,}000$

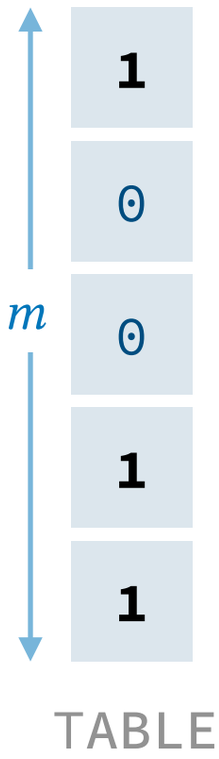
the probability of a wrong `YES` is

$$1 - e^{-\frac{10000}{1000000}} = 1 - e^{-\frac{1}{100}} \approx 0.01$$

m	probability of error
100	≈ 1
1000	≈ 1
2000	≈ 0.993
5000	≈ 0.865
10,000	≈ 0.632
50,000	≈ 0.181
100,000	≈ 0.095
500,000	≈ 0.02
1000,000	≈ 0.01

Solution # 1 Analysis

Probability of Error. After n insertions, what is the probability that **`CONTAINS(x)`** finds `TABLE[ h(x) ]` to be 1?



After 1 insertion:

$$P(\texttt{TABLE[} i \texttt{]} \text{ is } 1) = \frac{1}{m}$$

$$P(\texttt{TABLE[} i \texttt{]} \text{ is } \textbf{not } 1) = 1 - \frac{1}{m}$$

After n insertions:

$$P(\texttt{TABLE[} i \texttt{]} \text{ is still } \textbf{not } 1) = (1 - \frac{1}{m})^n = ((1 - \frac{1}{m})^m)^{\frac{n}{m}}$$

knowing that $\lim_{x \to \infty} (1 - \frac{1}{x})^x = e^{-1}$

$$P(\texttt{TABLE[} i \texttt{]} \text{ is still } \textbf{not } 1) = e^{-\frac{n}{m}}$$

$$P(\texttt{TABLE[} i \texttt{]} \text{ is } 1) = 1 - e^{-\frac{n}{m}}$$

Assuming $n = 10{,}000$

We need 1 million bits ($100n$ bits) to get a 1% error rate.

Not much better than normal hash tables (require $\geq 112n$ bits)

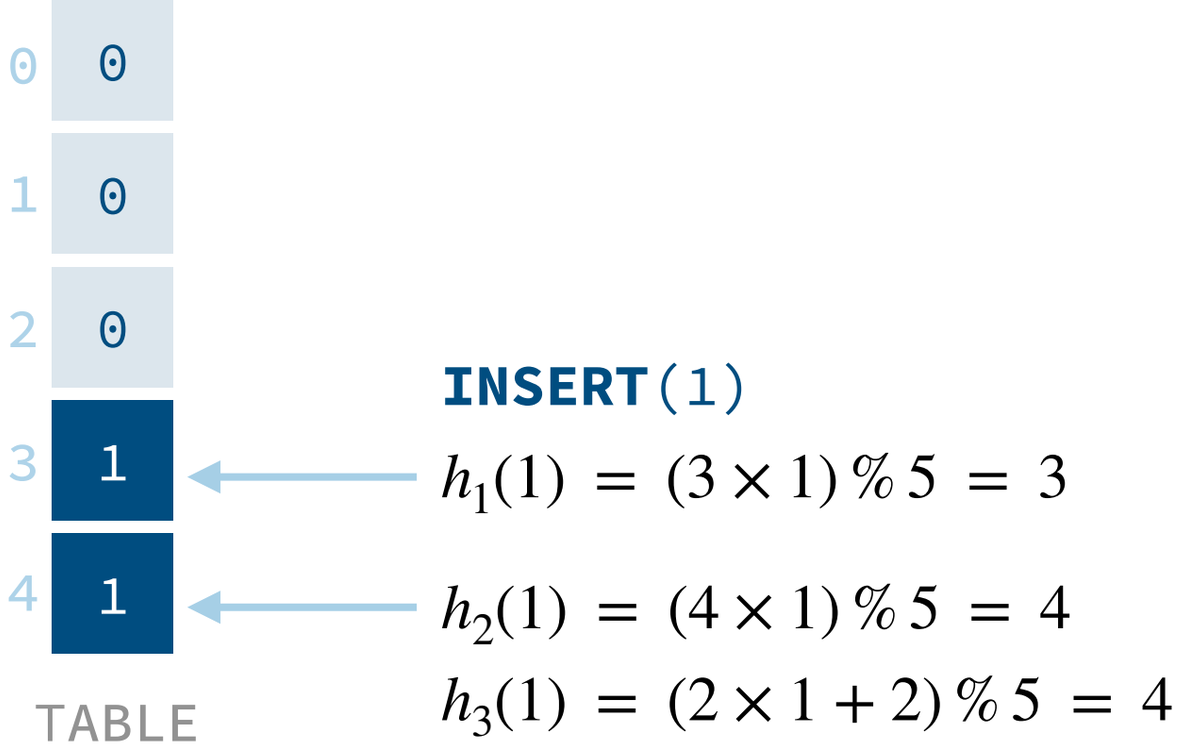
m	probability of error
100	≈ 1
1000	≈ 1
2000	≈ 0.993
5000	≈ 0.865
10,000	≈ 0.632
50,000	≈ 0.181
100,000	≈ 0.095
500,000	≈ 0.02
1000,000	≈ 0.01

Solution # 2 : Bloom Filters

Solution # 2. Use multiple hash functions: $h_1(x)$, $h_2(x)$, …, $h_k(x)$

```
INSERT(x):    Set    T[h1(x)]=1    T[h2(x)]=1    ... T[hk(x)]=1
CONTAINS(x):  Return T[h1(x)]==1 AND T[h2(x)]==1 AND ... T[hk(x)]==1
```

Example. $k = 3$, $h_1(x) = 3x \,\%\, m$, $h_2(x) = 4x \,\%\, m$, $h_3(x) = (2x + 2) \,\%\, m$.
Assume that $m = 5$, and we insert the elements $\{1, 3, 5\}$.



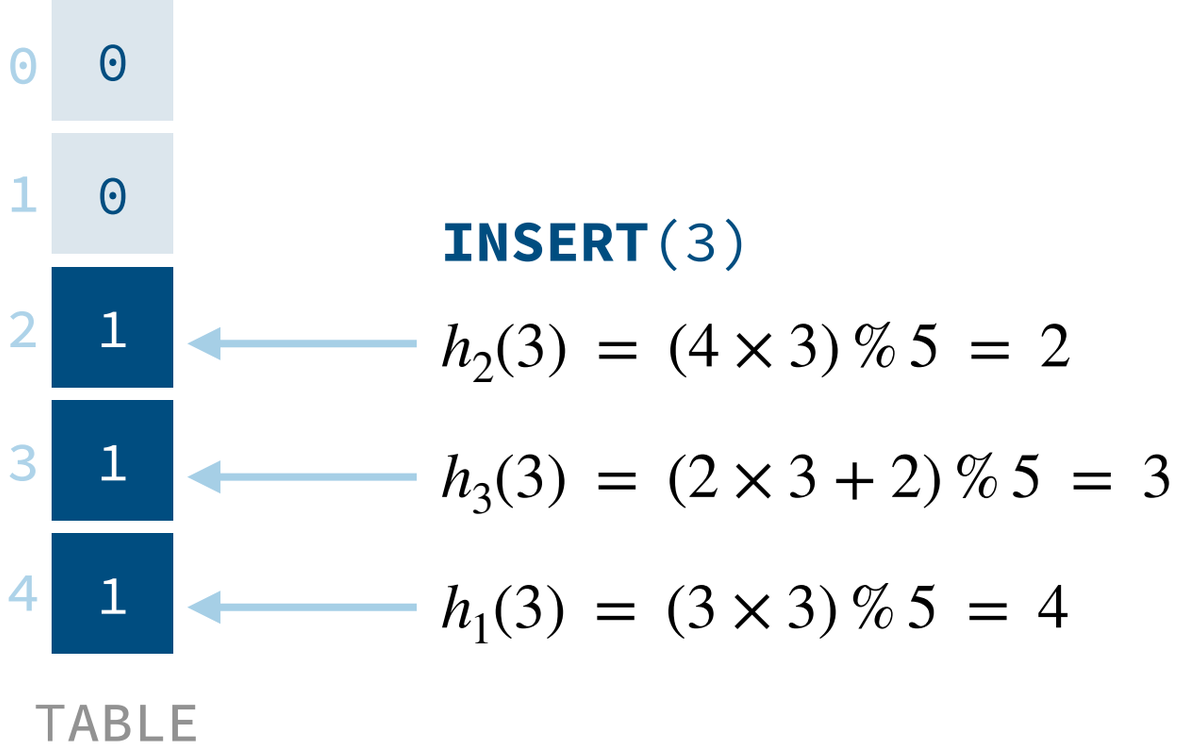
Solution # 2 : Bloom Filters

Solution # 2. Use multiple hash functions: $h_1(x)$, $h_2(x)$, …, $h_k(x)$

INSERT(x): Set `T[`$h_1(x)$`]=1` `T[`$h_2(x)$`]=1` … `T[`$h_k(x)$`]=1`

CONTAINS(x): Return `T[`$h_1(x)$`]==1` **AND** `T[`$h_2(x)$`]==1` **AND** … `T[`$h_k(x)$`]==1`

Example. $k = 3$, $h_1(x) = 3x \,\%\, m$, $h_2(x) = 4x \,\%\, m$, $h_3(x) = (2x + 2) \,\%\, m$.
Assume that $m = 5$, and we insert the elements $\{1, 3, 5\}$.

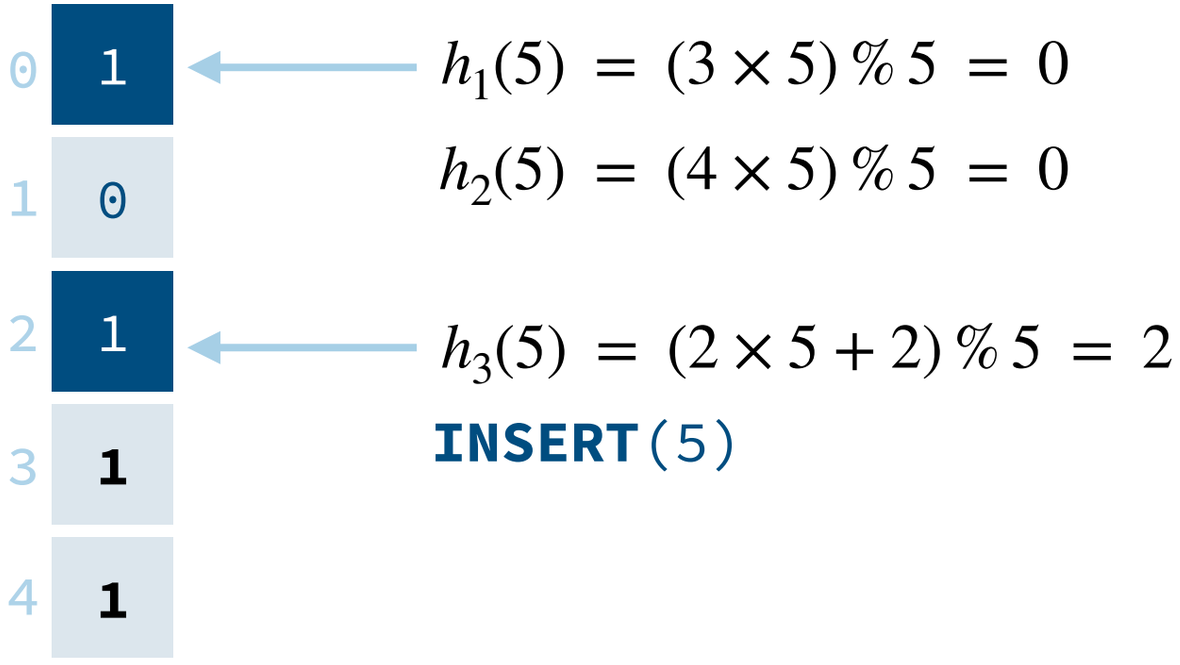


Solution # 2 : Bloom Filters

Solution # 2. Use multiple hash functions: $h_1(x)$, $h_2(x)$, …, $h_k(x)$

```
INSERT(x):    Set    T[h1(x)]=1    T[h2(x)]=1    ...  T[hk(x)]=1
CONTAINS(x):  Return T[h1(x)]==1 AND T[h2(x)]==1 AND ...  T[hk(x)]==1
```

Example. $k = 3$, $h_1(x) = 3x \,\%\, m$, $h_2(x) = 4x \,\%\, m$, $h_3(x) = (2x + 2) \,\%\, m$.
Assume that $m = 5$, and we insert the elements $\{1, 3, 5\}$.



Solution # 2 : Bloom Filters

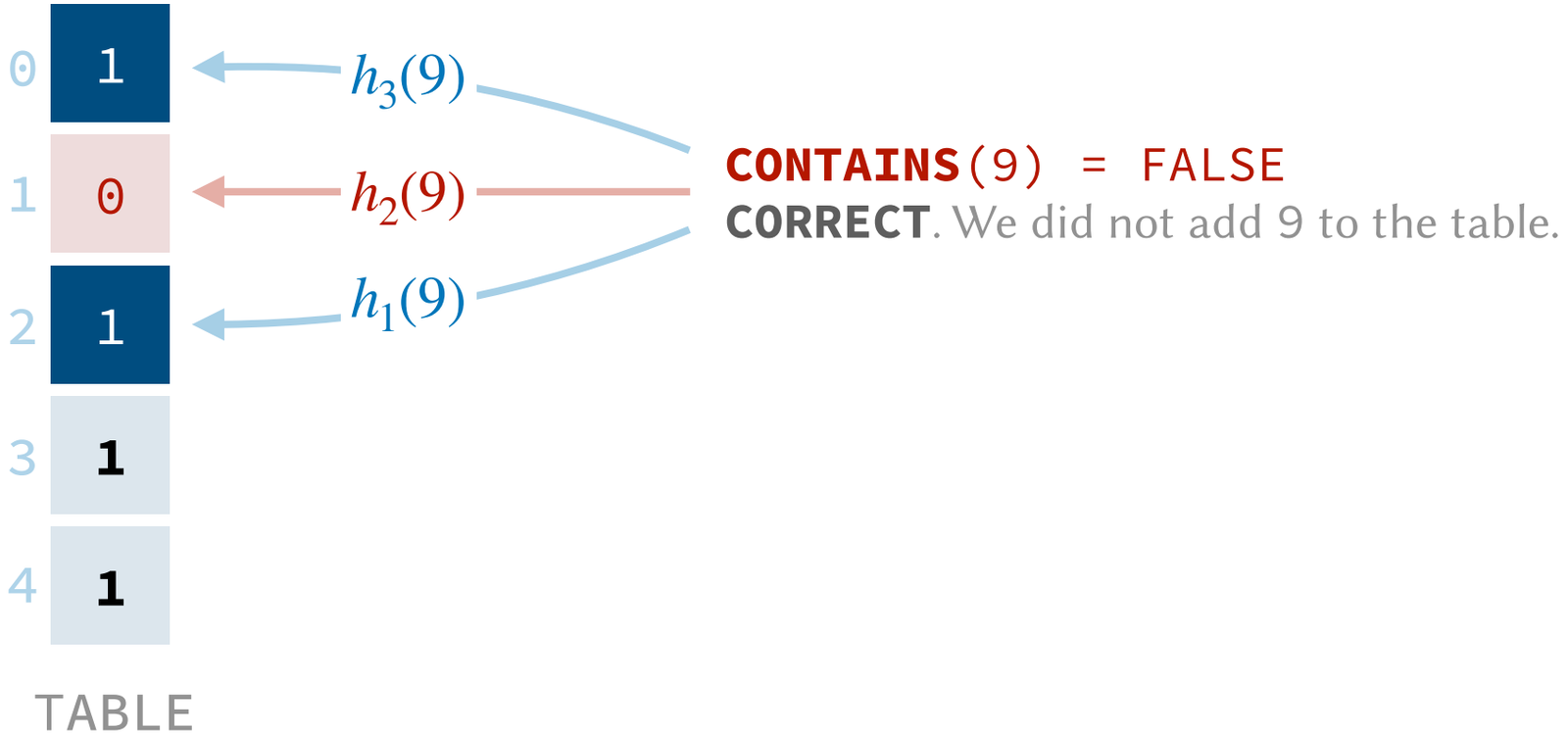
Solution # 2. Use multiple hash functions: $h_1(x)$, $h_2(x)$, …, $h_k(x)$

INSERT(x): Set `T[`$h_1(x)$`]=1` `T[`$h_2(x)$`]=1` … `T[`$h_k(x)$`]=1`

CONTAINS(x): Return `T[`$h_1(x)$`]==1` **AND** `T[`$h_2(x)$`]==1` **AND** … `T[`$h_k(x)$`]==1`

Example. $k = 3$, $h_1(x) = 3x \,\%\, m$, $h_2(x) = 4x \,\%\, m$, $h_3(x) = (2x + 2) \,\%\, m$.

Assume that $m = 5$, and we insert the elements $\{1, 3, 5\}$.



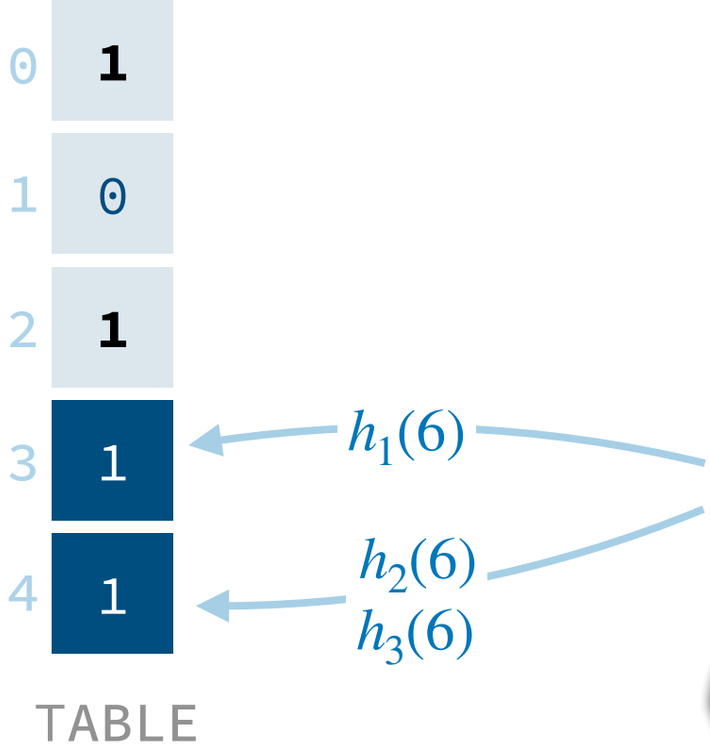
Solution # 2 : Bloom Filters

Solution # 2. Use multiple hash functions: $h_1(x)$, $h_2(x)$, …, $h_k(x)$

INSERT(x): Set `T[`$h_1(x)$`]=1` `T[`$h_2(x)$`]=1` … `T[`$h_k(x)$`]=1`

CONTAINS(x): Return `T[`$h_1(x)$`]==1` **AND** `T[`$h_2(x)$`]==1` **AND** … `T[`$h_k(x)$`]==1`

Example. $k = 3$, $h_1(x) = 3x \,\%\, m$, $h_2(x) = 4x \,\%\, m$, $h_3(x) = (2x + 2) \,\%\, m$.
Assume that $m = 5$, and we insert the elements $\{1, 3, 5\}$.

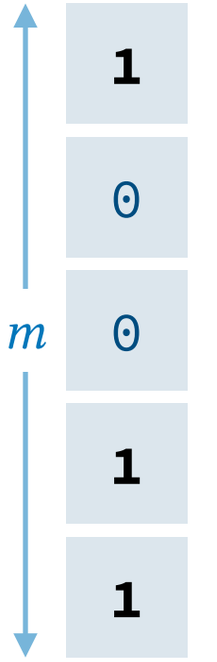


CONTAINS(6) = TRUE
WRONG. We did not add 6 to the table.

After n insertions, what is the probability that **CONTAINS**(x) finds `T[` $h_1(x)$ `]` and `T[` $h_2(x)$ `]` … and `T[` $h_k(x)$ `]` to be 1?

Solution # 2 : Analysis

Probability of Error. After n insertions, what is the probability that **`CONTAINS(x)`** finds `T[` $h_1(x)$ `]` and `T[` $h_2(x)$ `]` ... and `T[` $h_k(x)$ `]` to be 1?



After 1 insertion: $P(\texttt{T[}\ i\ \texttt{]}$ is **not** $1) = (1-\frac{1}{m})^k$

After n insertions: $P(\texttt{T[}\ i\ \texttt{]}$ is still **not** $1) = (1-\frac{1}{m})^{kn} = ((1-\frac{1}{m})^m)^{\frac{kn}{m}}$

knowing that $\lim_{x\to\infty}(1-\frac{1}{x})^x = e^{-1}$

$P(\texttt{T[}\ i\ \texttt{]}$ is still **not** $1) = e^{-\frac{kn}{m}}$

$P(\texttt{T[}\ i\ \texttt{]}$ is $1) = 1 - e^{-\frac{kn}{m}}$

$P(\texttt{T[}\ h_1(x)\ \texttt{]}$ and `T[` $h_2(x)$ `]` ... and `T[` $h_k(x)$ `]` $= 1) =$

$$\epsilon = (1 - e^{-\frac{kn}{m}})^k$$

Fixing n and m, we find (using differentiation) that the optimal value of k that minimizes the probability of error is:

$$k = \frac{m}{n}\ln 2$$

Is this low enough?

Solution # 2 : Analysis

Probability of Error. After n insertions, what is the probability that **`CONTAINS(x)`** finds `T[` $h_1(x)$ `]` and `T[` $h_2(x)$ `]` ... and `T[` $h_k(x)$ `]` to be 1?

$$\epsilon = (1 - e^{-\frac{kn}{m}})^k$$

$$k = \frac{m}{n} \ln 2$$

Assuming $n = 10{,}000$

m	optimal k	probability of error
10,000	< 1	≈ 0.67
20,000	≈ 1.386	≈ 0.48
30,000	≈ 2.079	≈ 0.33
40,000	≈ 2.773	≈ 0.23
50,000	≈ 3.466	≈ 0.16
100,000	≈ 6.931	≈ 0.026
125,000	≈ 8.664	≈ 0.01

We need 125,000 bits ($12.5n$ bits) to get a 1% error rate (using 8-9 hash functions)

Much better than normal hash tables (require $\geq 112n$ bits)