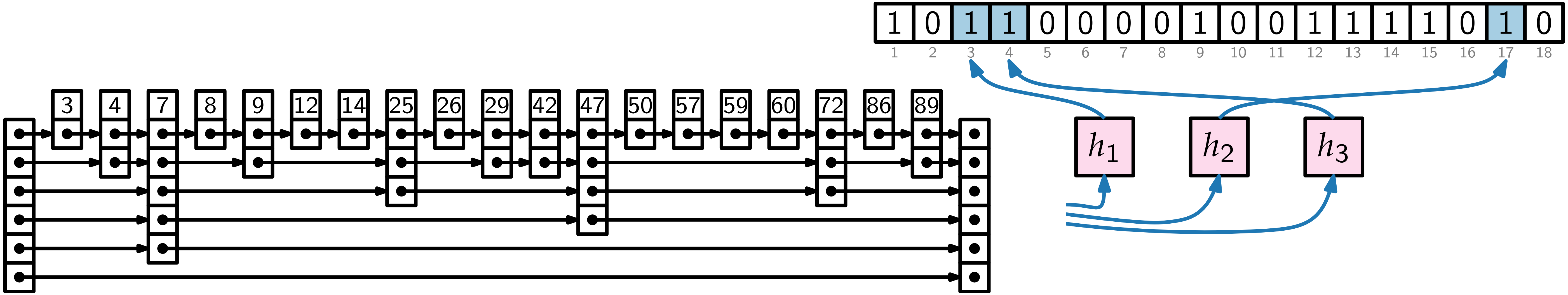
Advanced Algorithms

Randomized and Probabilistic Data Structures

Skip Lists & Bloom Filters

Johannes Zink · WS22



Data Structures and Randomization

- We have seen that (in expectation) some **randomized algorithms** beat all deterministic approaches in terms of running time.

Data Structures and Randomization

- We have seen that (in expectation) some **randomized algorithms** beat all deterministic approaches in terms of running time.
- Moreover, randomized approaches are often simpler/more elegant.

Data Structures and Randomization

- We have seen that (in expectation) some **randomized algorithms** beat all deterministic approaches in terms of running time.
- Moreover, randomized approaches are often simpler/more elegant.
- Also data structures may use randomization for these reasons.

Data Structures and Randomization

- We have seen that (in expectation) some **randomized algorithms** beat all deterministic approaches in terms of running time.
- Moreover, randomized approaches are often simpler/more elegant.
- Also data structures may use randomization for these reasons.
- A data structure that uses randomization (e.g. for a better expected runtime/space consumption or a simpler implementation) is a **randomized data structure**.

Data Structures and Randomization

- We have seen that (in expectation) some **randomized algorithms** beat all deterministic approaches in terms of running time.
- Moreover, randomized approaches are often simpler/more elegant.
- Also data structures may use randomization for these reasons.
- A data structure that uses randomization (e.g. for a better expected runtime/space consumption or a simpler implementation) is a **randomized data structure**.
  - Randomized skip lists (in this lecture!)

Data Structures and Randomization

- We have seen that (in expectation) some **randomized algorithms** beat all deterministic approaches in terms of running time.
- Moreover, randomized approaches are often simpler/more elegant.
- Also data structures may use randomization for these reasons.
- A data structure that uses randomization (e.g. for a better expected runtime/space consumption or a simpler implementation) is a **randomized data structure**.
  - Randomized skip lists (in this lecture!)
  - Randomized binary search trees and treaps (= tree + heap)

Data Structures and Randomization

- We have seen that (in expectation) some **randomized algorithms** beat all deterministic approaches in terms of running time.
- Moreover, randomized approaches are often simpler/more elegant.
- Also data structures may use randomization for these reasons.
- A data structure that uses randomization (e.g. for a better expected runtime/space consumption or a simpler implementation) is a **randomized data structure**.
  - Randomized skip lists (in this lecture!)
  - Randomized binary search trees and treaps (= tree + heap)
  - Hashing when choosing a random hash function

Data Structures and Randomization

- We have seen that (in expectation) some **randomized algorithms** beat all deterministic approaches in terms of running time.
- Moreover, randomized approaches are often simpler/more elegant.
- Also data structures may use randomization for these reasons.
- A data structure that uses randomization (e.g. for a better expected runtime/space consumption or a simpler implementation) is a **randomized data structure**.
  - Randomized skip lists (in this lecture!)
  - Randomized binary search trees and treaps (= tree + heap)
  - Hashing when choosing a random hash function
- A data structure that answers correctly according to some probabilty distribution is a **probabilistic data structure**.

Data Structures and Randomization

- We have seen that (in expectation) some **randomized algorithms** beat all deterministic approaches in terms of running time.
- Moreover, randomized approaches are often simpler/more elegant.
- Also data structures may use randomization for these reasons.
- A data structure that uses randomization (e.g. for a better expected runtime/space consumption or a simpler implementation) is a **randomized data structure**.
  - Randomized skip lists (in this lecture!)
  - Randomized binary search trees and treaps (= tree + heap)
  - Hashing when choosing a random hash function
- A data structure that answers correctly according to some probabilty distribution is a **probabilistic data structure**.
  - Bloom filters (in this lecture!)

Data Structures and Randomization

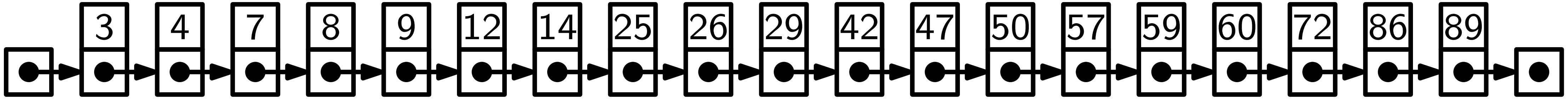
- We have seen that (in expectation) some **randomized algorithms** beat all deterministic approaches in terms of running time.
- Moreover, randomized approaches are often simpler/more elegant.
- Also data structures may use randomization for these reasons.
- A data structure that uses randomization (e.g. for a better expected runtime/space consumption or a simpler implementation) is a **randomized data structure**.
  - Randomized skip lists (in this lecture!)
  - Randomized binary search trees and treaps (= tree + heap)
  - Hashing when choosing a random hash function
- A data structure that answers correctly according to some probabilty distribution is a **probabilistic data structure**.
  - Bloom filters (in this lecture!)
  - Count–min sketch (estimates the frequency of different events in a data stream)

Deterministic Skip Lists

What time is needed to search an element in a sorted linked list with n elements?

Deterministic Skip Lists

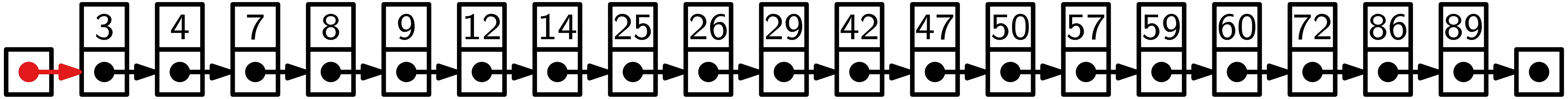
What time is needed to search an element in a sorted linked list with n elements?



e.g. search 42

Deterministic Skip Lists

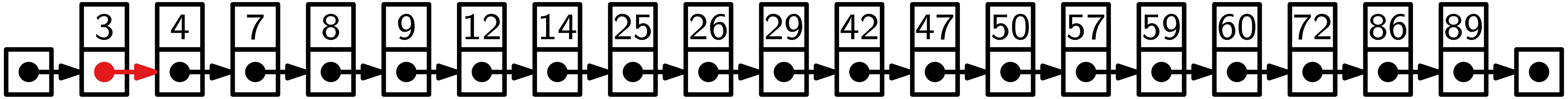
What time is needed to search an element in a sorted linked list with n elements?



e.g. search 42

Deterministic Skip Lists

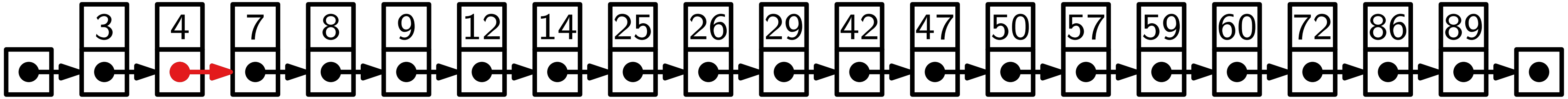
What time is needed to search an element in a sorted linked list with n elements?



e.g. search 42

Deterministic Skip Lists

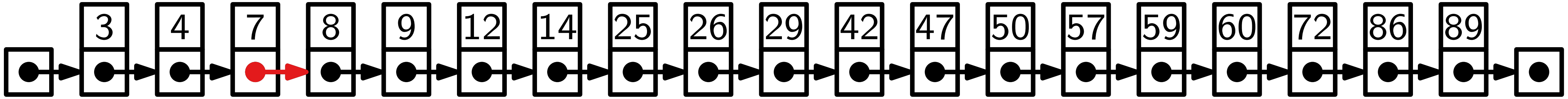
What time is needed to search an element in a sorted linked list with n elements?



e.g. search 42

Deterministic Skip Lists

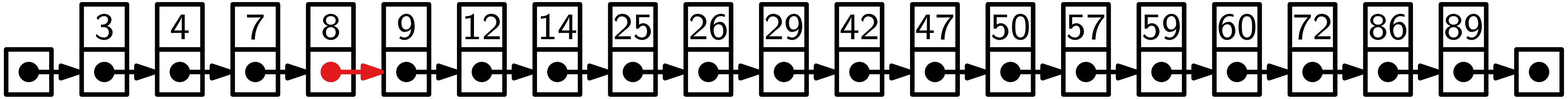
What time is needed to search an element in a sorted linked list with n elements?



e.g. search 42

Deterministic Skip Lists

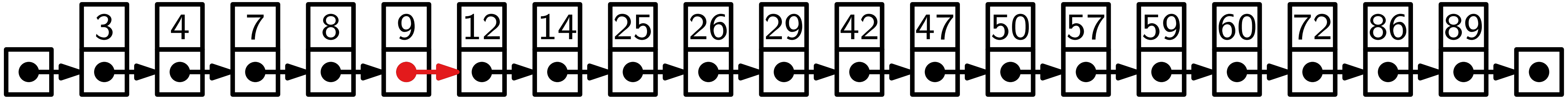
What time is needed to search an element in a sorted linked list with n elements?



e.g. search 42

Deterministic Skip Lists

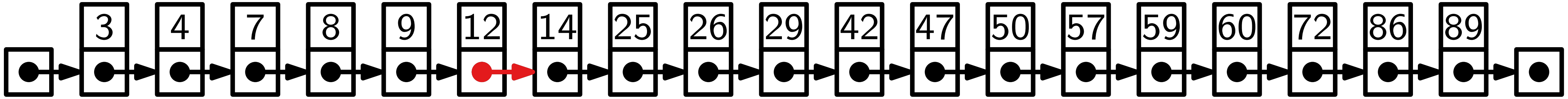
What time is needed to search an element in a sorted linked list with n elements?



e.g. search 42

Deterministic Skip Lists

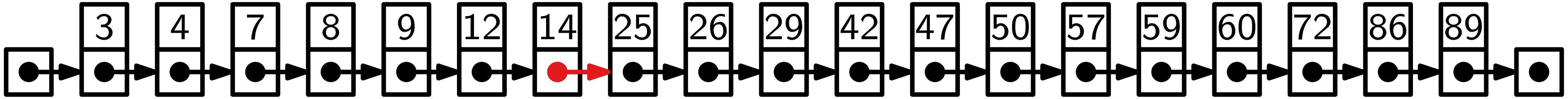
What time is needed to search an element in a sorted linked list with n elements?



e.g. search 42

Deterministic Skip Lists

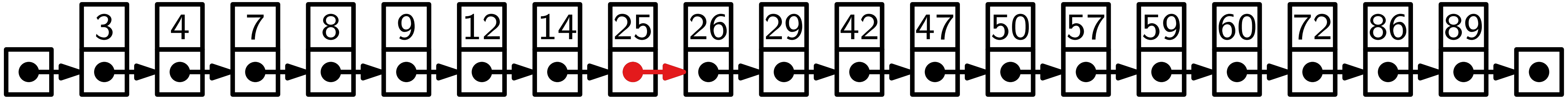
What time is needed to search an element in a sorted linked list with n elements?



e.g. search 42

Deterministic Skip Lists

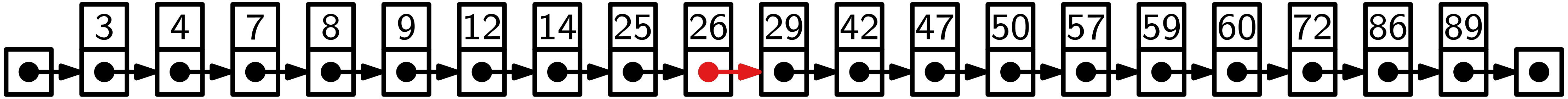
What time is needed to search an element in a sorted linked list with n elements?



e.g. search 42

Deterministic Skip Lists

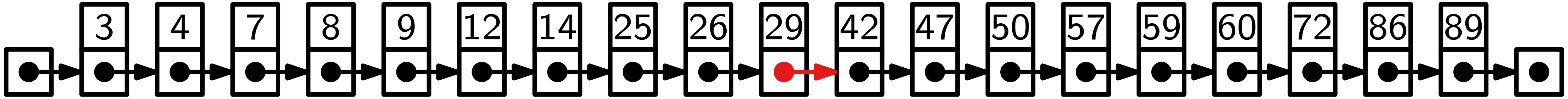
What time is needed to search an element in a sorted linked list with n elements?



e.g. search 42

Deterministic Skip Lists

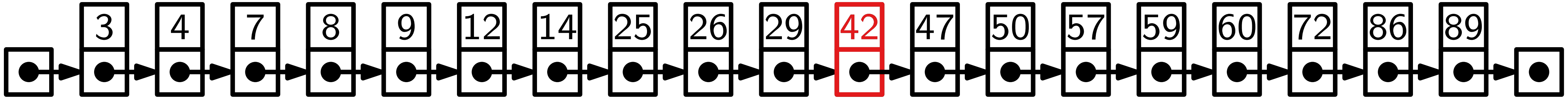
What time is needed to search an element in a sorted linked list with n elements?



e.g. search 42

Deterministic Skip Lists

What time is needed to search an element in a sorted linked list with n elements?

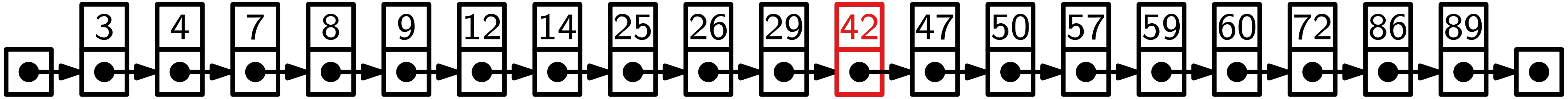


e.g. search 42

Deterministic Skip Lists

What time is needed to search an element in a sorted linked list with n elements?

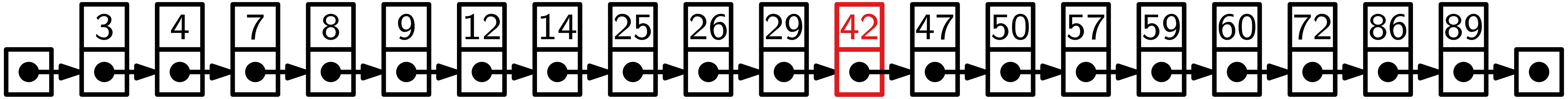
$\Theta(n)$



e.g. search 42

Deterministic Skip Lists

What time is needed to search an element in a sorted linked list with n elements?



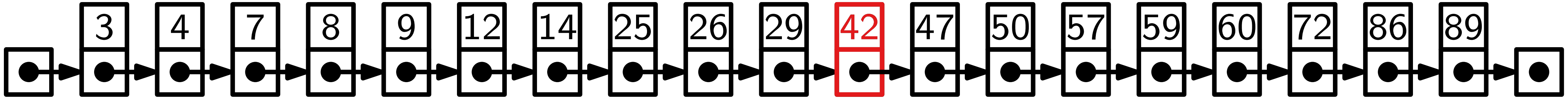
$\Theta(n)$

e.g. search 42

We know that there are data structures like balanced binary search trees that allow for searching in $\Theta(\log n)$ time. However, they are more complicated than linked lists.

Deterministic Skip Lists

What time is needed to search an element in a sorted linked list with n elements?



e.g. search 42

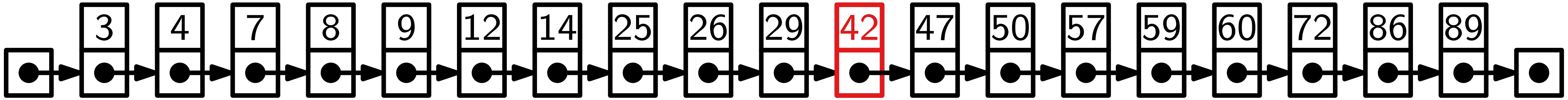
$\Theta(n)$

We know that there are data structures like balanced binary search trees that allow for searching in $\Theta(\log n)$ time. However, they are more complicated than linked lists.

Idea: Keep a linked list but add "shortcuts" (or more lists) to skip 1, 2, 4, 8, . . . entries.

Deterministic Skip Lists

What time is needed to search an element in a sorted linked list with n elements?

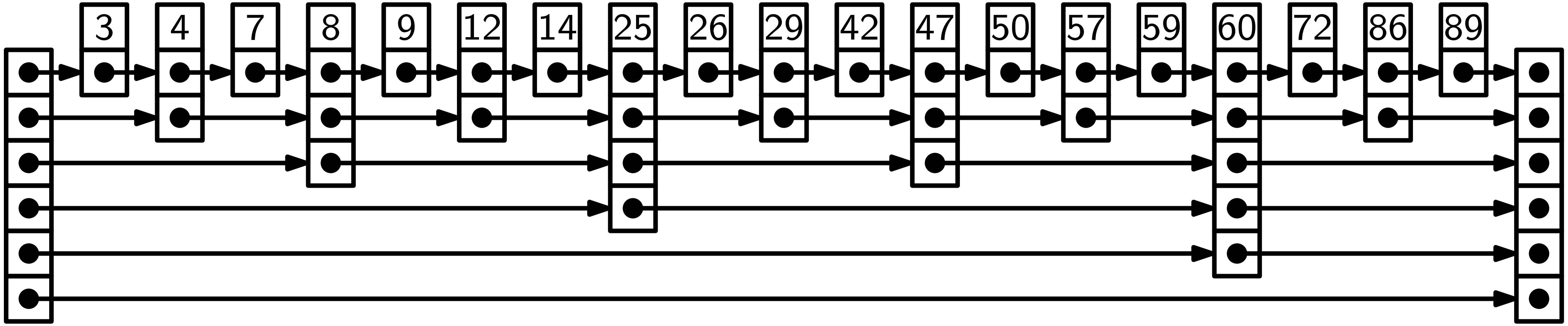


$\Theta(n)$

e.g. search 42

We know that there are data structures like balanced binary search trees that allow for searching in $\Theta(\log n)$ time. However, they are more complicated than linked lists.

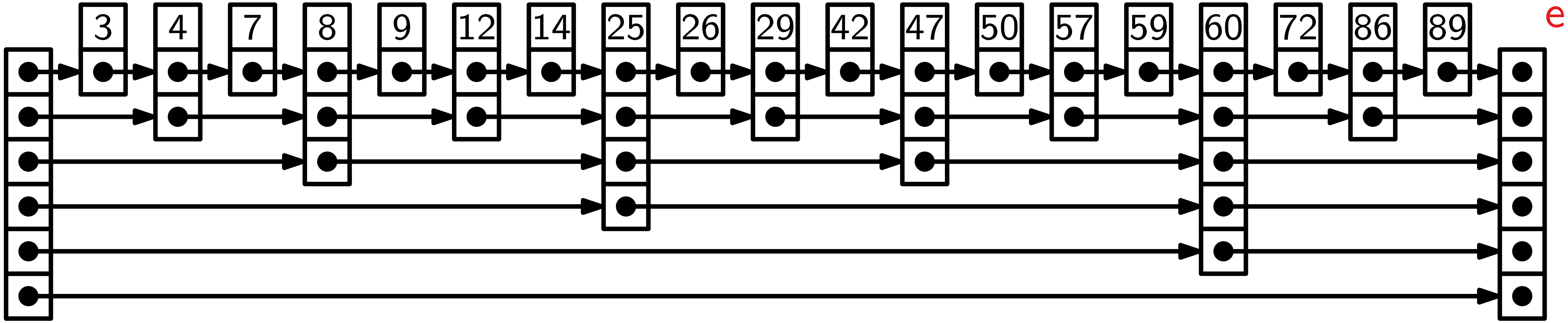
Idea: Keep a linked list but add "shortcuts" (or more lists) to skip 1, 2, 4, 8, . . . entries.



Deterministic skip list

Deterministic Skip Lists

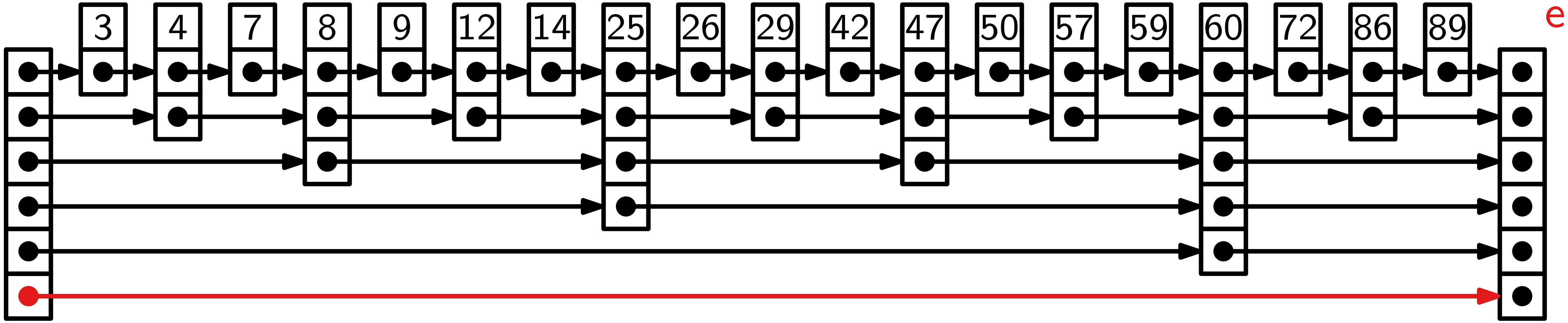
What time is needed to search an element in a deterministic skip list with n elements?



e.g. search 42

Deterministic Skip Lists

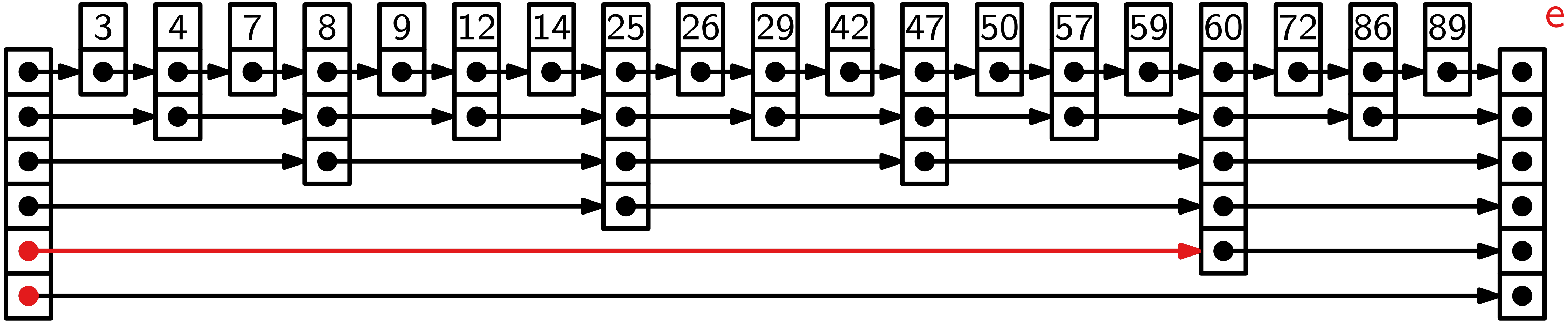
What time is needed to search an element in a deterministic skip list with n elements?



e.g. search 42

Deterministic Skip Lists

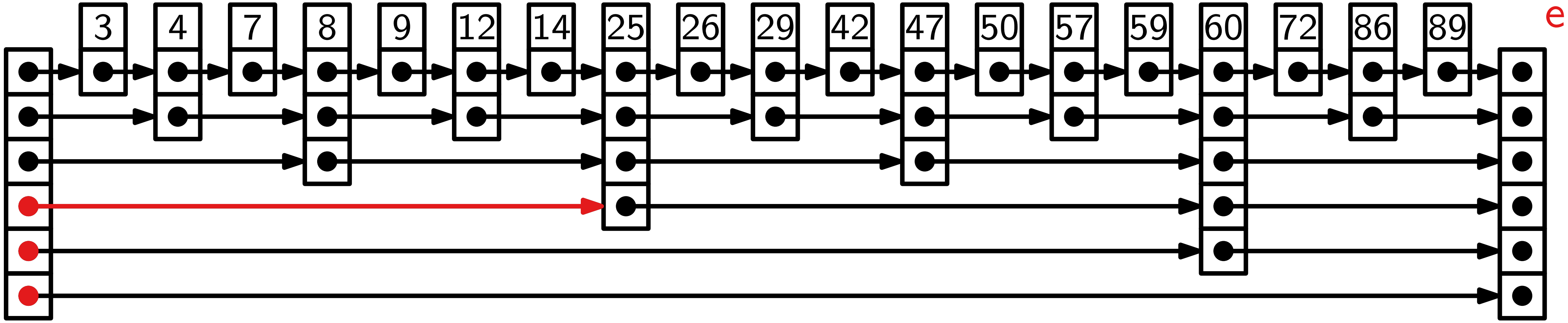
What time is needed to search an element in a deterministic skip list with n elements?



e.g. search 42

Deterministic Skip Lists

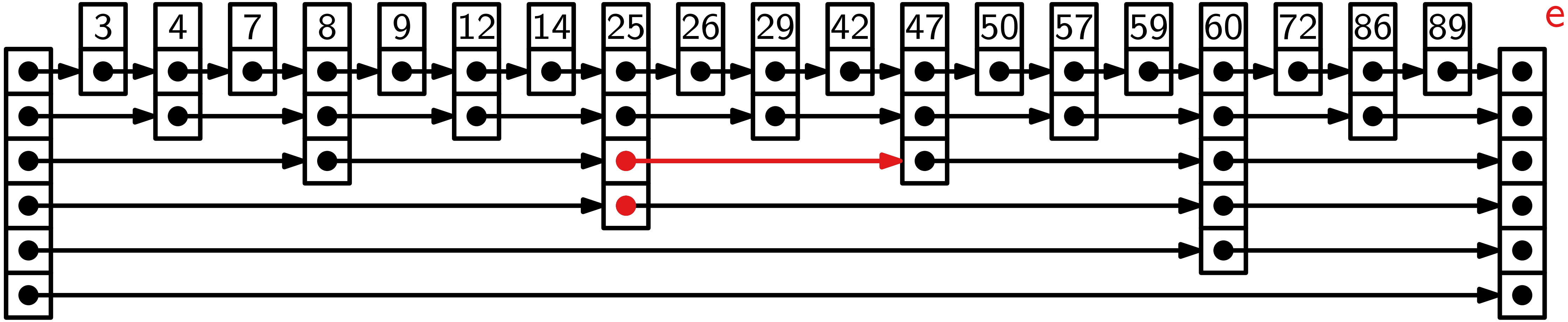
What time is needed to search an element in a deterministic skip list with n elements?



e.g. search 42

Deterministic Skip Lists

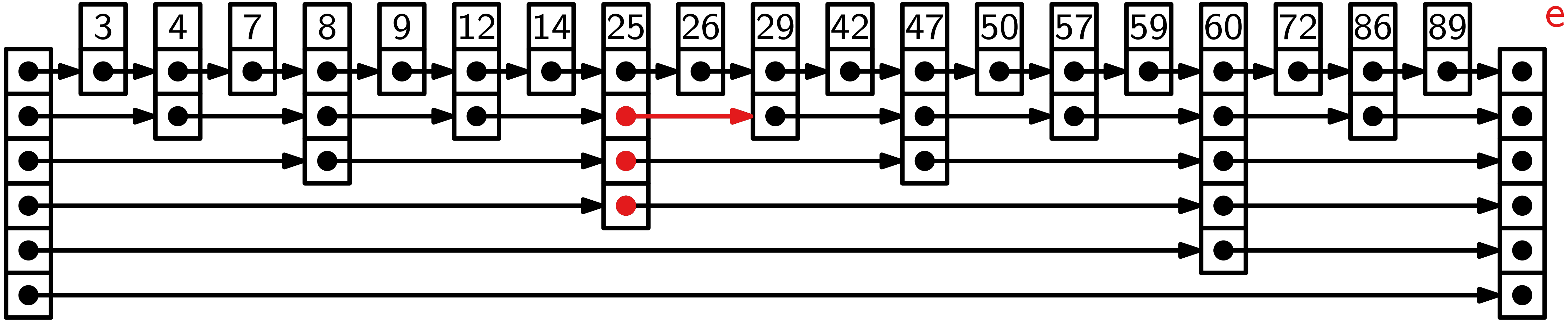
What time is needed to search an element in a deterministic skip list with n elements?



e.g. search 42

Deterministic Skip Lists

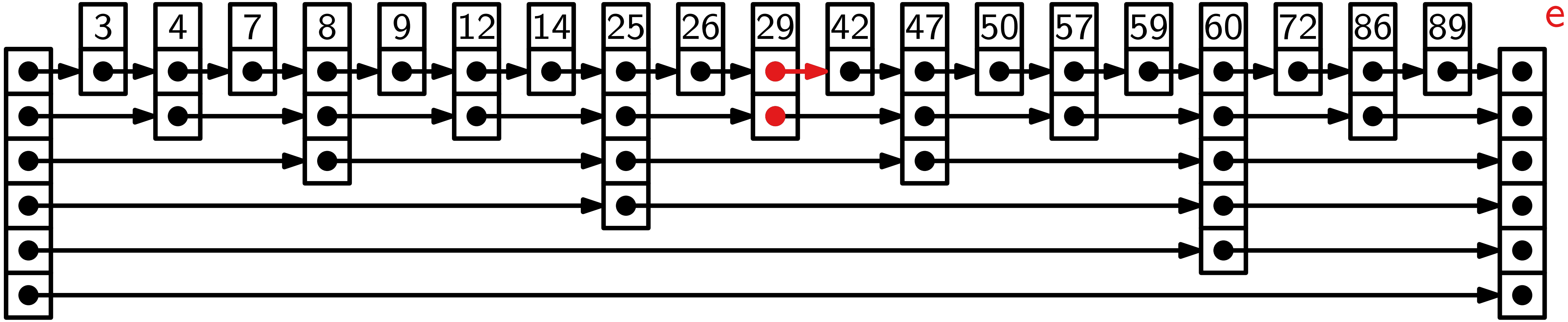
What time is needed to search an element in a deterministic skip list with n elements?



e.g. search 42

Deterministic Skip Lists

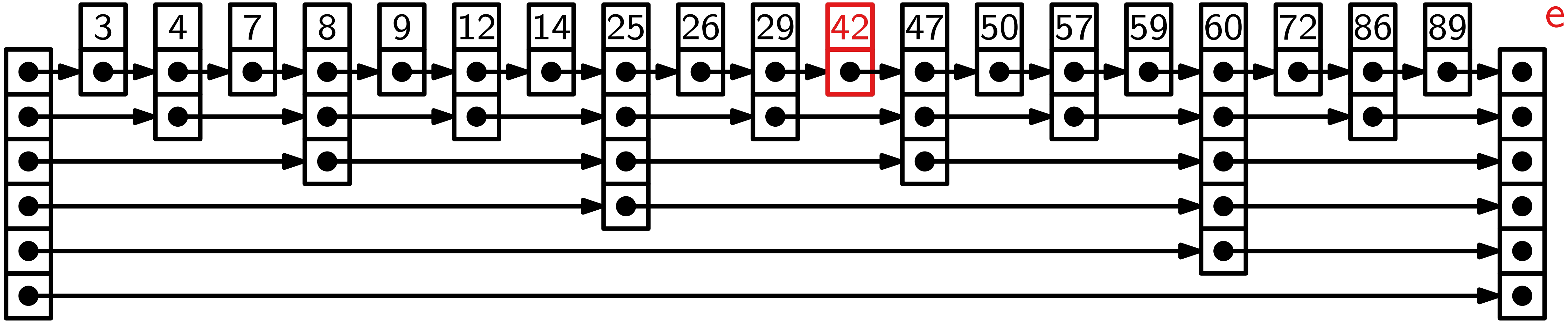
What time is needed to search an element in a deterministic skip list with n elements?



e.g. search 42

Deterministic Skip Lists

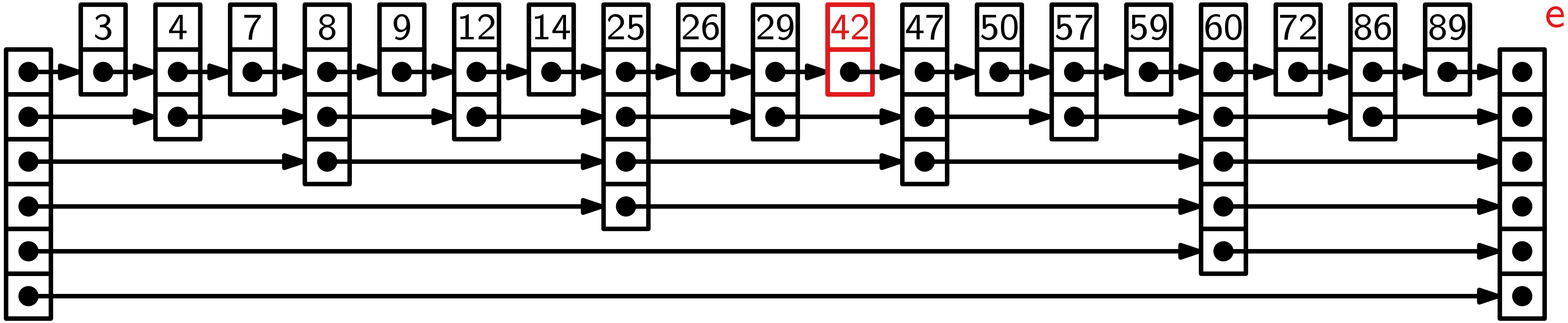
What time is needed to search an element in a deterministic skip list with n elements?



e.g. search 42

Deterministic Skip Lists

What time is needed to search an element in a deterministic skip list with n elements?



e.g. search 42

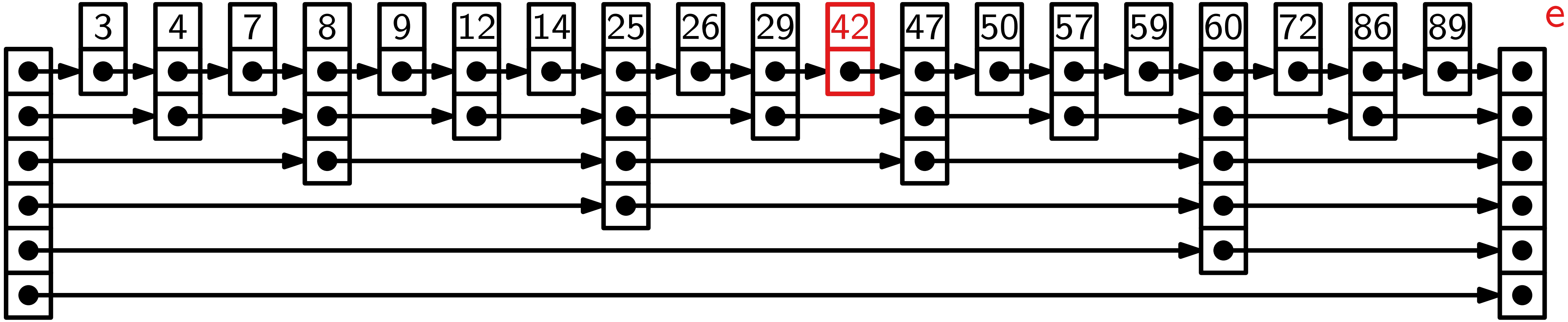
$\Theta(\log n)$

(We check only constantly many entries per level and there are $O(\log n)$ levels.)

What time is needed to insert/delete an element in a deterministic skip list?

Deterministic Skip Lists

What time is needed to search an element in a deterministic skip list with n elements?



e.g. search 42

$\Theta(\log n)$

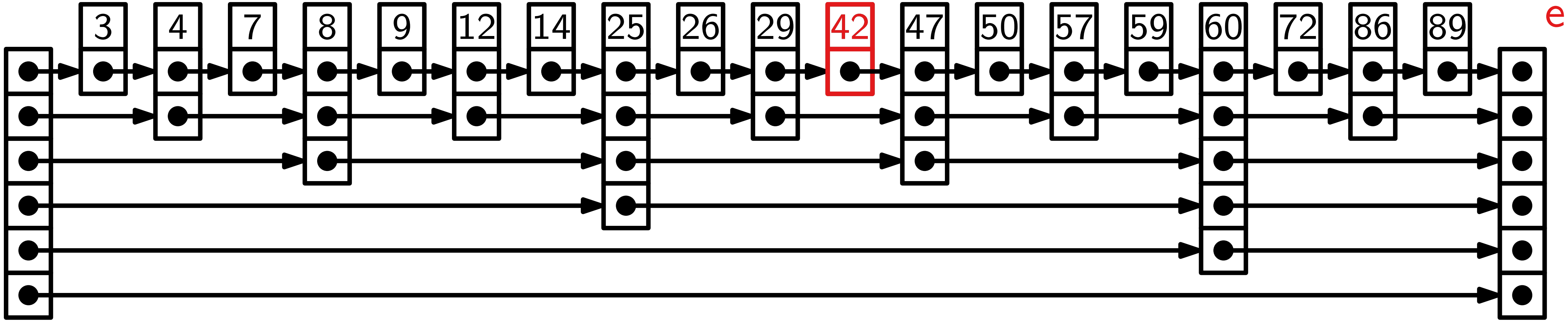
(We check only constantly many entries per level and there are $O(\log n)$ levels.)

What time is needed to insert/delete an element in a deterministic skip list?

$\Theta(n)$ (We need to re-build large parts of the data structure if we remove or add an element)

Deterministic Skip Lists

What time is needed to search an element in a deterministic skip list with n elements?



e.g. search 42

$\Theta(\log n)$

(We check only constantly many entries per level and there are $O(\log n)$ levels.)

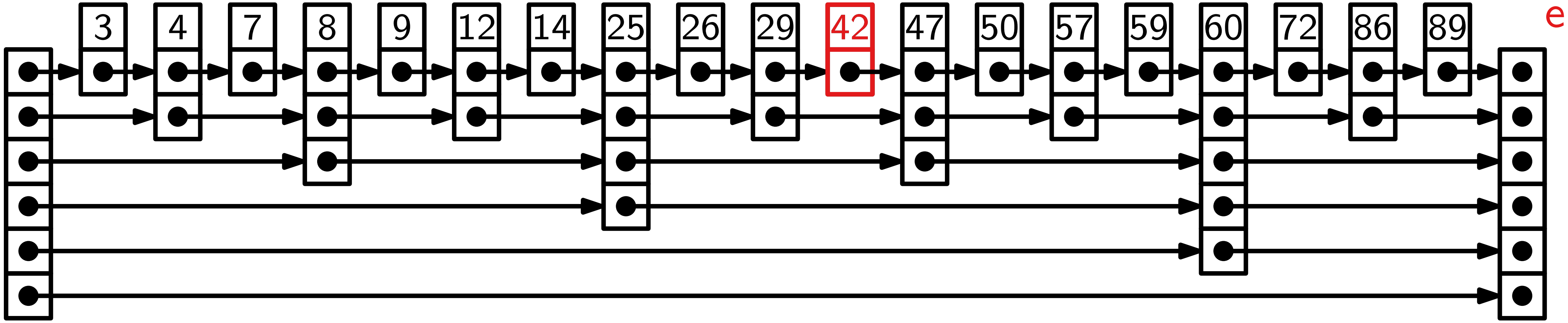
What time is needed to insert/delete an element in a deterministic skip list?

$\Theta(n)$ (We need to re-build large parts of the data structure if we remove or add an element)

We know that there are data structures like balanced binary search trees that allow for insertion/deletion in $\Theta(\log n)$ time. However, they are more complicated than skip lists.

Deterministic Skip Lists

What time is needed to search an element in a deterministic skip list with n elements?



e.g. search 42

$\Theta(\log n)$

(We check only constantly many entries per level and there are $O(\log n)$ levels.)

What time is needed to insert/delete an element in a deterministic skip list?

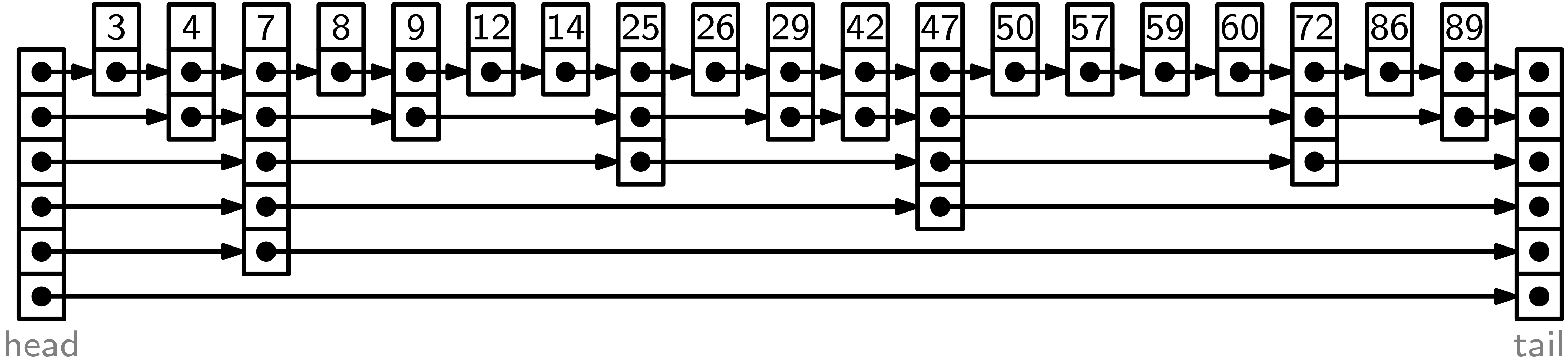
$\Theta(n)$ (We need to re-build large parts of the data structure if we remove or add an element)

We know that there are data structures like balanced binary search trees that allow for insertion/deletion in $\Theta(\log n)$ time. However, they are more complicated than skip lists.

Idea: Keep a skip list, but assign each entry a random height (number of lists it occurs in) s.t. lower heights are more likely to occur.

(Randomized) Skip Lists

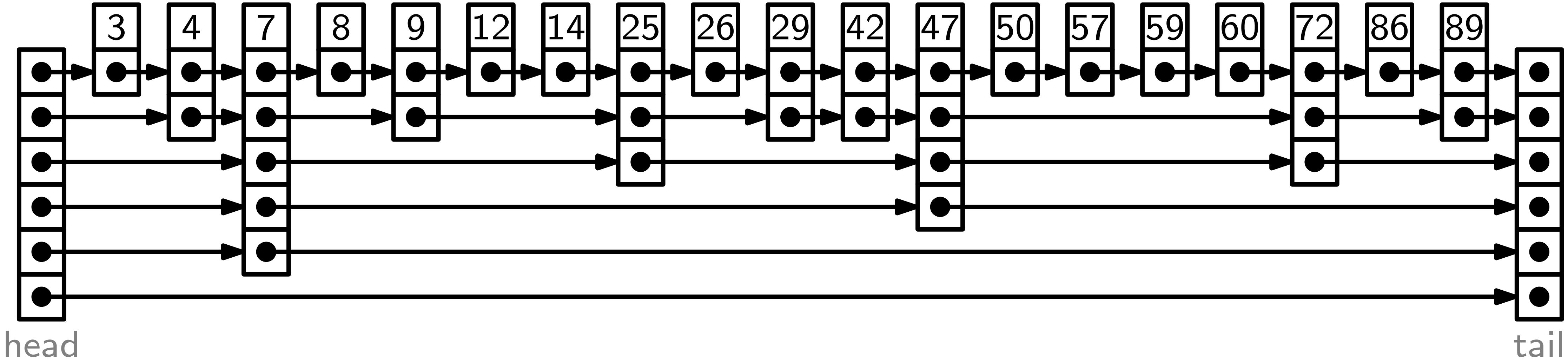
For a new entry, flip a coin until it shows HEAD. The number of flips will be its height.



(We give
the head/tail
of the list
a height of
$\lfloor \log_2 n \rfloor + 1$.)

(Randomized) Skip Lists

For a new entry, flip a coin until it shows HEAD. The number of flips will be its height.

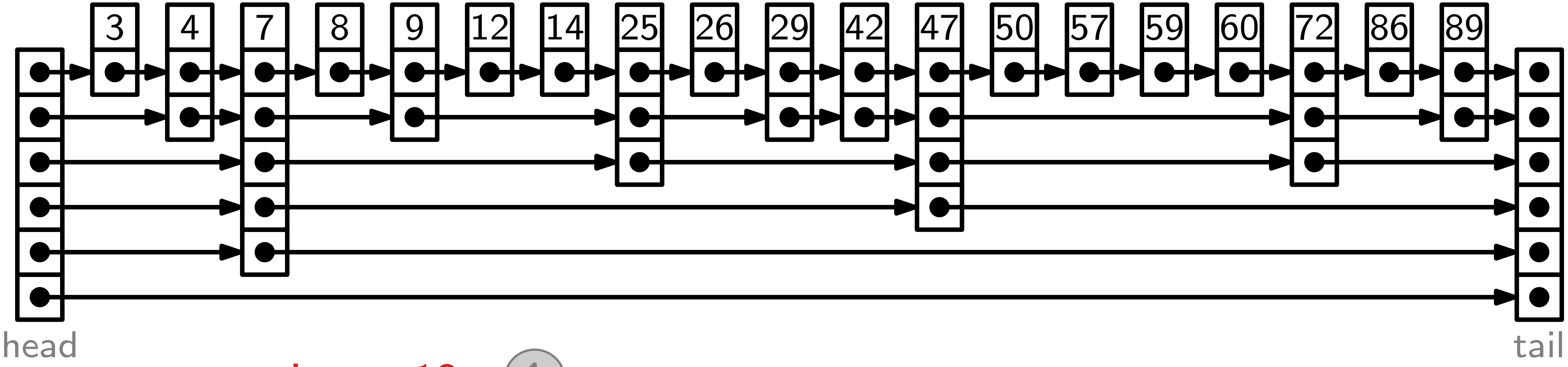


(We give
the head/tail
of the list
a height of
$\lfloor \log_2 n \rfloor + 1$.)

e.g. insert 13

(Randomized) Skip Lists

For a new entry, flip a coin until it shows HEAD. The number of flips will be its height.

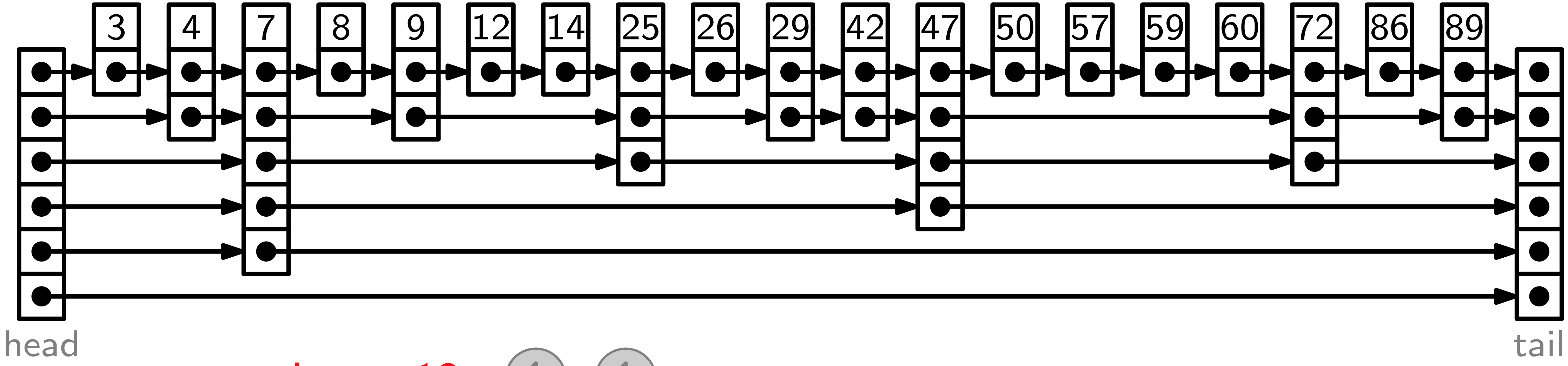


(We give
the head/tail
of the list
a height of
$\lfloor \log_2 n \rfloor + 1$.)

e.g. insert 13

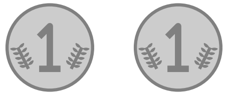
(Randomized) Skip Lists

For a new entry, flip a coin until it shows HEAD. The number of flips will be its height.



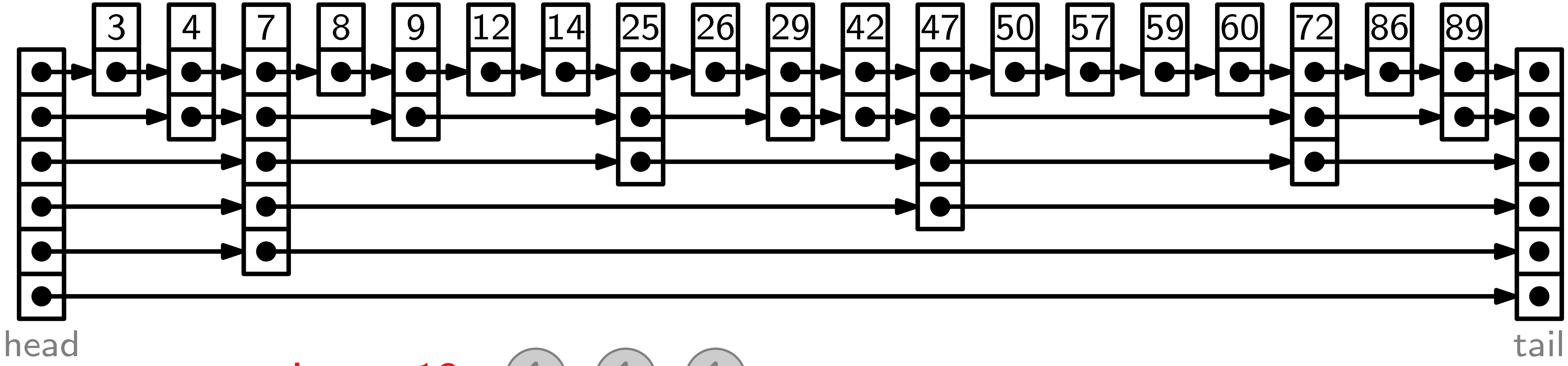
(We give
the head/tail
of the list
a height of
$\lfloor \log_2 n \rfloor + 1$.)

e.g. insert 13



(Randomized) Skip Lists

For a new entry, flip a coin until it shows HEAD. The number of flips will be its height.

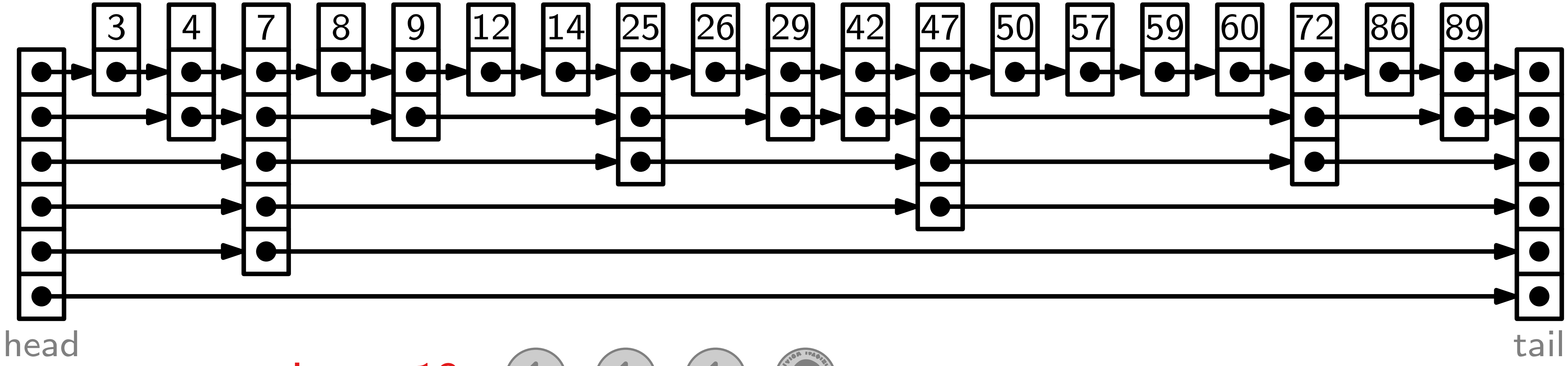


(We give
the head/tail
of the list
a height of
$\lfloor \log_2 n \rfloor + 1$.)

e.g. insert 13

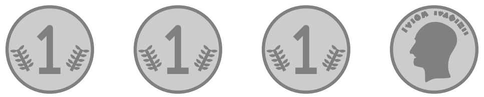
(Randomized) Skip Lists

For a new entry, flip a coin until it shows HEAD. The number of flips will be its height.



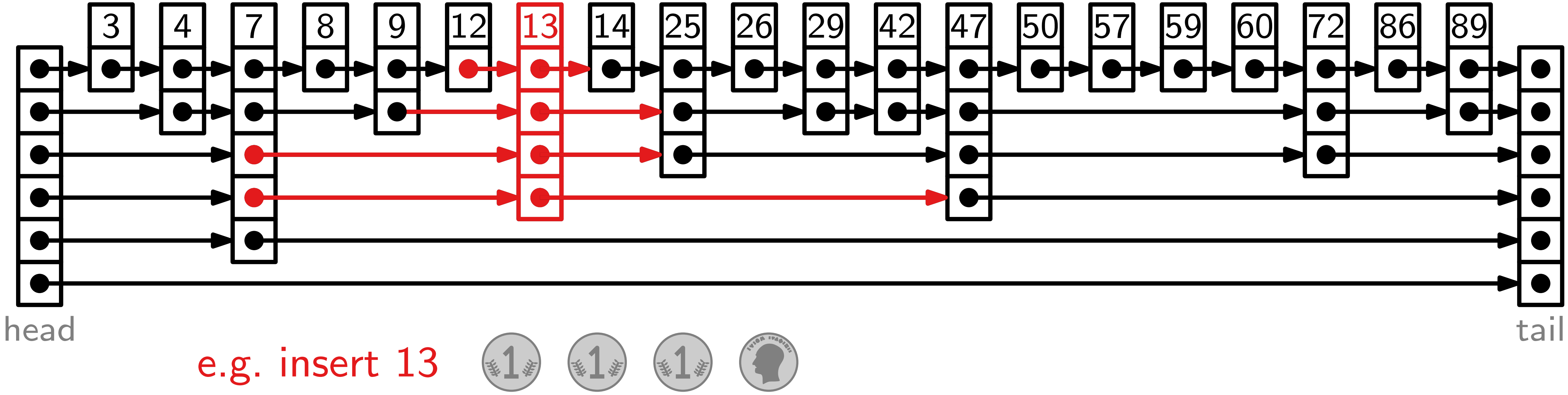
(We give
the head/tail
of the list
a height of
$\lfloor \log_2 n \rfloor + 1$.)

e.g. insert 13



(Randomized) Skip Lists

For a new entry, flip a coin until it shows HEAD. The number of flips will be its height.

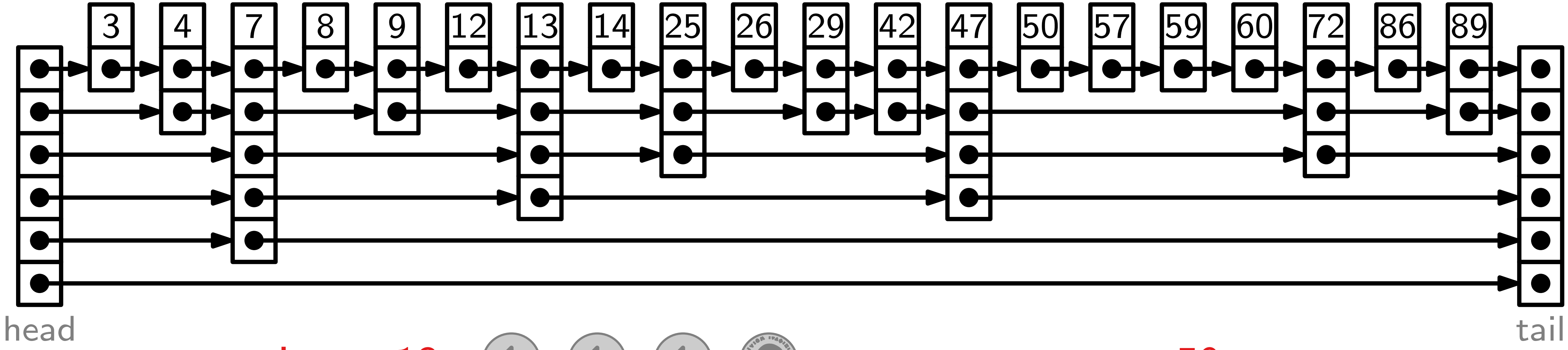


e.g. insert 13

(We give the head/tail of the list a height of $\lfloor \log_2 n \rfloor + 1$.)

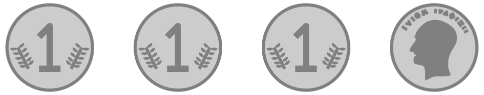
(Randomized) Skip Lists

For a new entry, flip a coin until it shows HEAD. The number of flips will be its height.



(We give the head/tail of the list a height of $\lfloor \log_2 n \rfloor + 1$.)

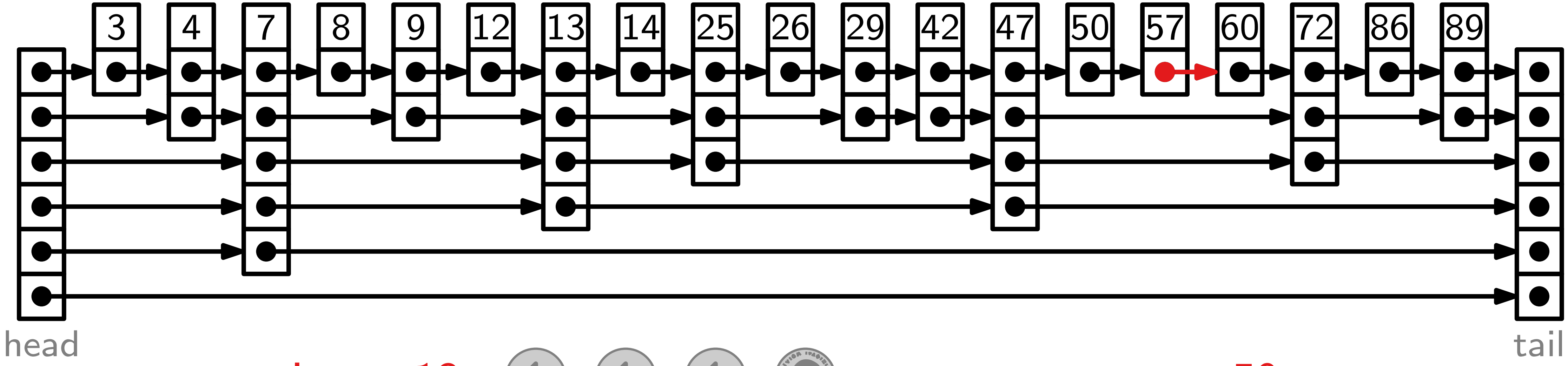
e.g. insert 13



e.g. remove 59

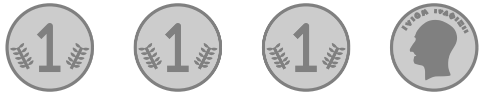
(Randomized) Skip Lists

For a new entry, flip a coin until it shows HEAD. The number of flips will be its height.



(We give
the head/tail
of the list
a height of
$\lfloor \log_2 n \rfloor + 1$.)

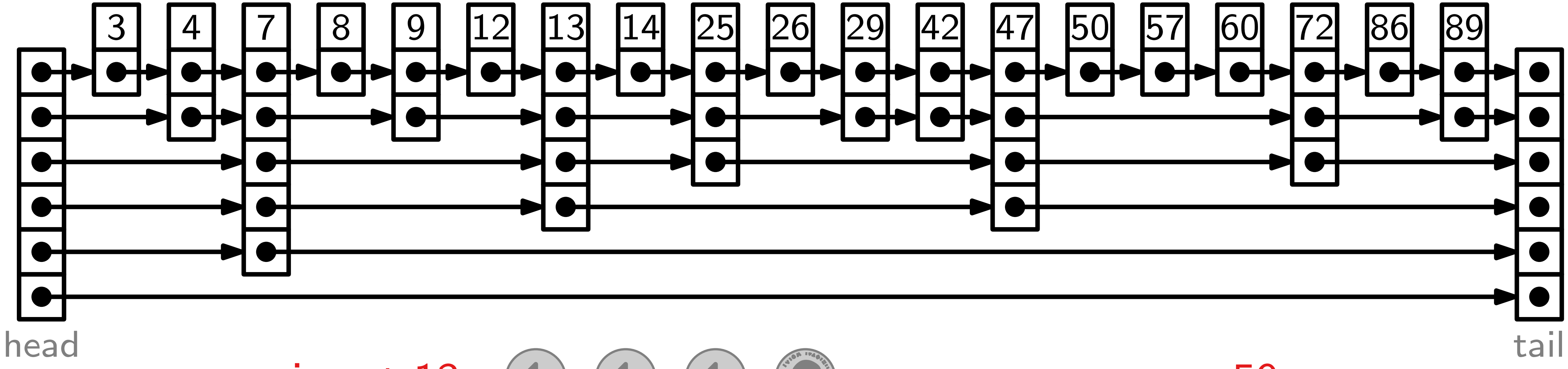
e.g. insert 13



e.g. remove 59

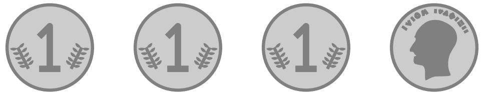
(Randomized) Skip Lists

For a new entry, flip a coin until it shows HEAD. The number of flips will be its height.



(We give the head/tail of the list a height of $\lfloor \log_2 n \rfloor + 1$.)

e.g. insert 13

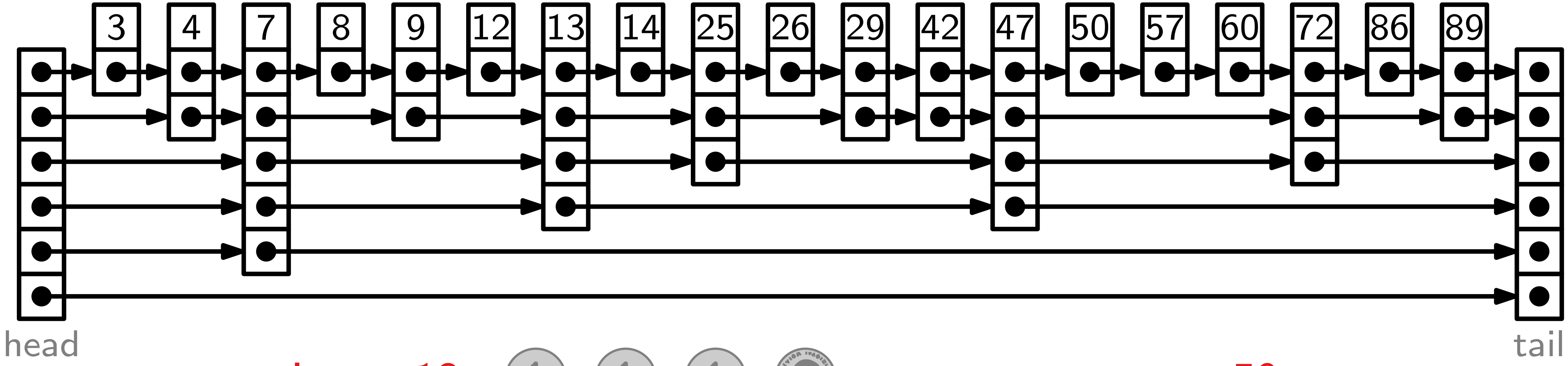


e.g. remove 59

Insertion and deletion works in $O(\log n)$ time + the time to search an element.
(We store the $O(\log n)$ pointers that need to be updated while searching.
Searching works in the same way as for deterministic skip lists.)

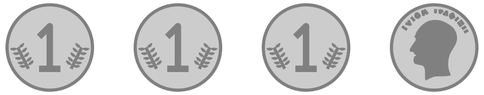
(Randomized) Skip Lists

For a new entry, flip a coin until it shows HEAD. The number of flips will be its height.



(We give
the head/tail
of the list
a height of
$\lfloor \log_2 n \rfloor + 1$.)

e.g. insert 13



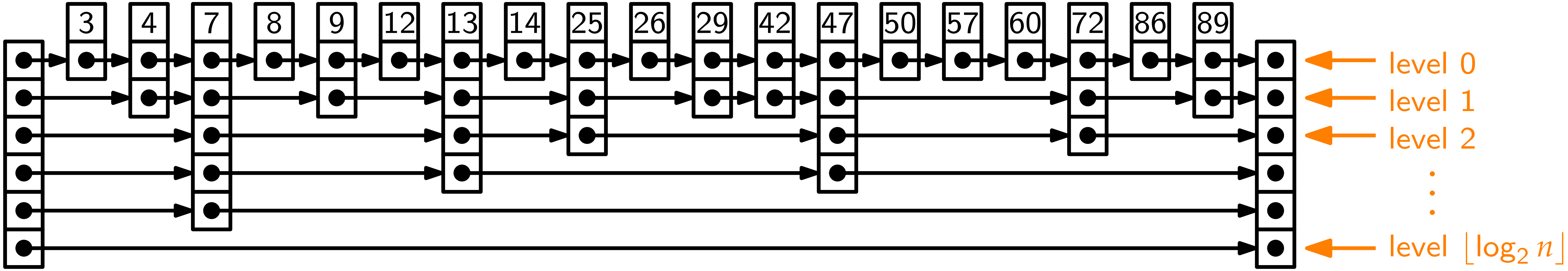
e.g. remove 59

Insertion and deletion works in $O(\log n)$ time + the time to search an element.
(We store the $O(\log n)$ pointers that need to be updated while searching.
Searching works in the same way as for deterministic skip lists.)

Theorem 1. Searching in a (rand.) skip list can be done in expected $O(\log n)$ time.

(Randomized) Skip Lists

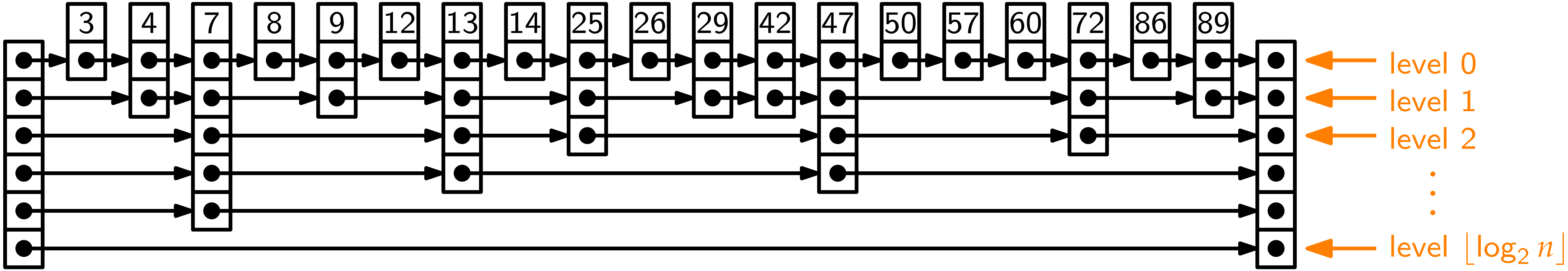
Proof of Theorem 1.



Theorem 1. Searching in a (rand.) skip list can be done in expected $O(\log n)$ time.

(Randomized) Skip Lists

Proof of Theorem 1.

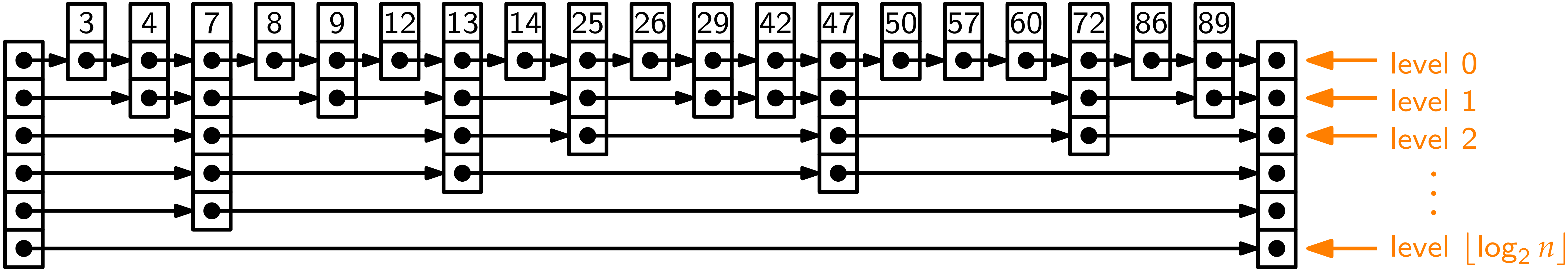


How long is the search path to reach the element we search for?

Theorem 1. Searching in a (rand.) skip list can be done in expected $O(\log n)$ time.

(Randomized) Skip Lists

Proof of Theorem 1.



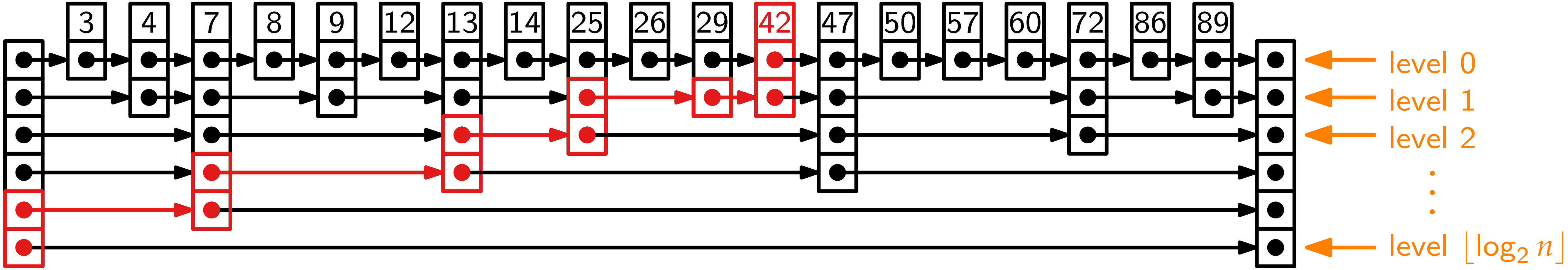
How long is the search path to reach the element we search for?

We do backwards analysis (→ see lecture on rand. algorithms) on the search path.

Theorem 1. Searching in a (rand.) skip list can be done in expected $O(\log n)$ time.

(Randomized) Skip Lists

Proof of Theorem 1.



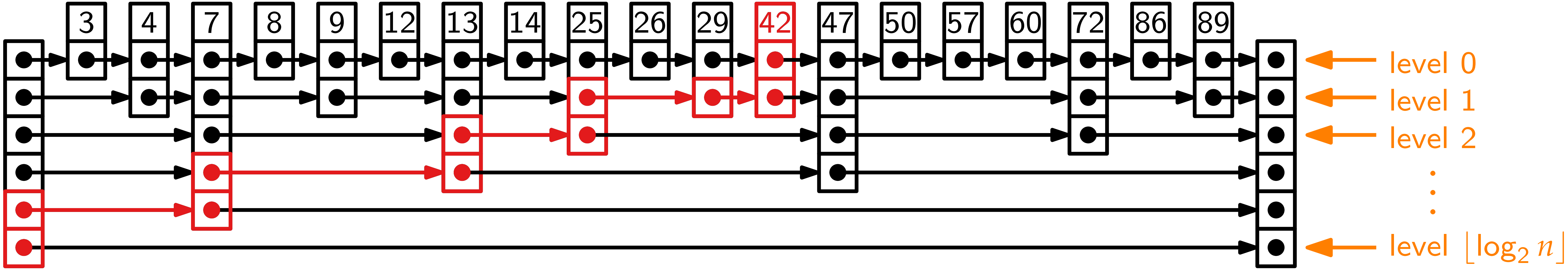
How long is the search path to reach the element we search for?

We do backwards analysis (→ see lecture on rand. algorithms) on the search path.

Theorem 1. Searching in a (rand.) skip list can be done in expected $O(\log n)$ time.

(Randomized) Skip Lists

Proof of Theorem 1.



How long is the search path to reach the element we search for?

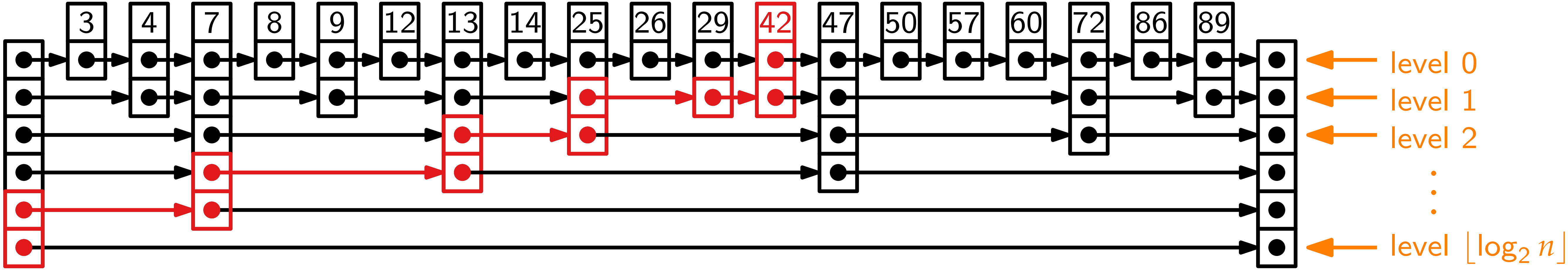
We do backwards analysis (→ see lecture on rand. algorithms) on the search path.

In the reverse search path, we always go to the next greater level if possible, otherwise we follow the (reverse) pointer to the left.

Theorem 1. Searching in a (rand.) skip list can be done in expected $O(\log n)$ time.

(Randomized) Skip Lists

Proof of Theorem 1.



How long is the search path to reach the element we search for?

We do backwards analysis (→ see lecture on rand. algorithms) on the search path.

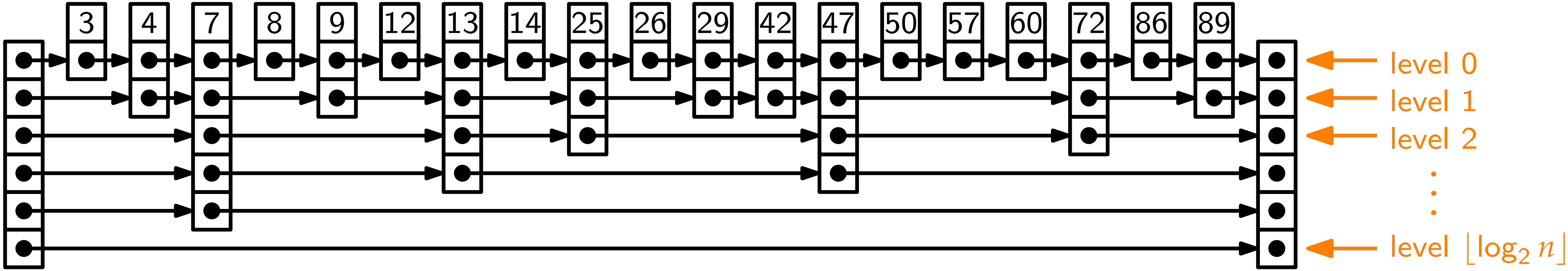
In the reverse search path, we always go to the next greater level if possible, otherwise we follow the (reverse) pointer to the left.

If we are at level i, the probability that we can go a level up is $1/2$ by construction.

Theorem 1. Searching in a (rand.) skip list can be done in expected $O(\log n)$ time.

(Randomized) Skip Lists

Proof of Theorem 1.

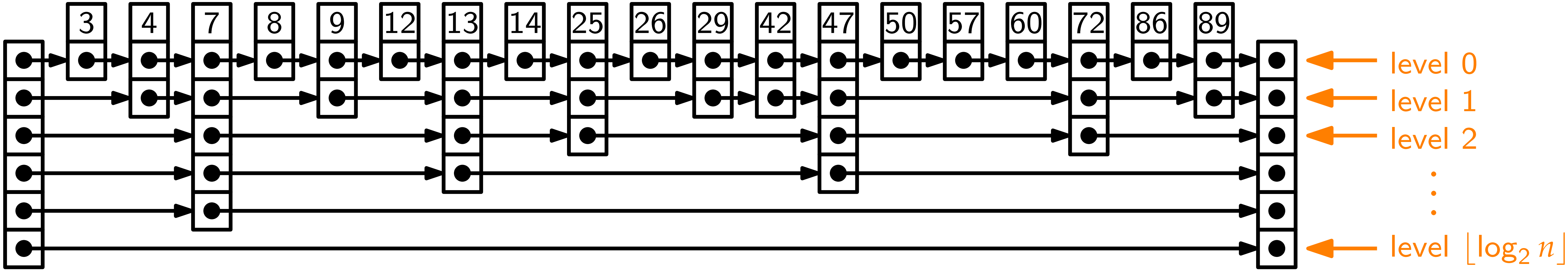


Let X_i be a random variable denoting the number of steps we take on level i or lower.

Theorem 1. Searching in a (rand.) skip list can be done in expected $O(\log n)$ time.

(Randomized) Skip Lists

Proof of Theorem 1.



Let X_i be a random variable denoting the number of steps we take on level i or lower.

$$E[X_i] = 1 + \tfrac{1}{2}E[X_{i-1}] + \tfrac{1}{2}E[X_i]$$

(for $i < 0$: $E[X_i] = 0$)

in the previous step we used a (reverse) pointer to the left

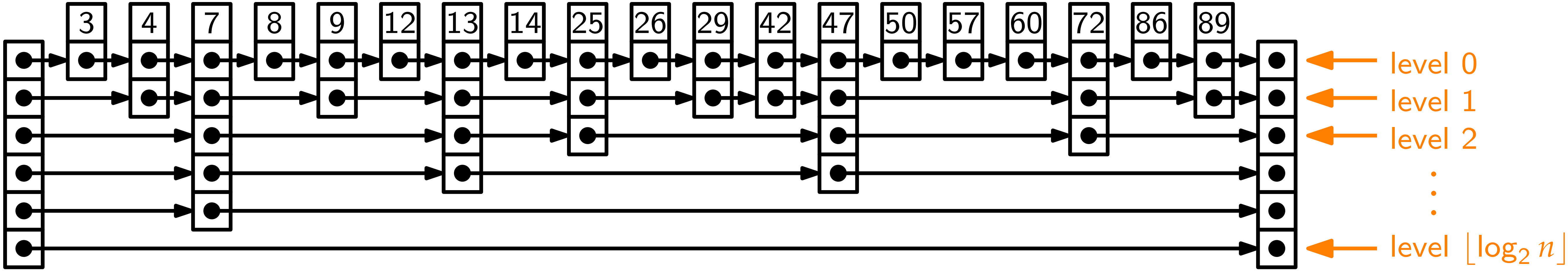
in the previous step we went a level up

current step we take on level i (start with the last step we take on level i; we don't skip levels)

Theorem 1. Searching in a (rand.) skip list can be done in expected $O(\log n)$ time.

(Randomized) Skip Lists

Proof of Theorem 1.



Let X_i be a random variable denoting the number of steps we take on level i or lower.

$$E[X_i] = 1 + \tfrac{1}{2}E[X_{i-1}] + \tfrac{1}{2}E[X_i] \quad \Leftrightarrow \quad E[X_i] = 2 + E[X_{i-1}] \qquad (\text{for } i < 0\text{: } E[X_i] = 0)$$

in the previous step we used a (reverse) pointer to the left

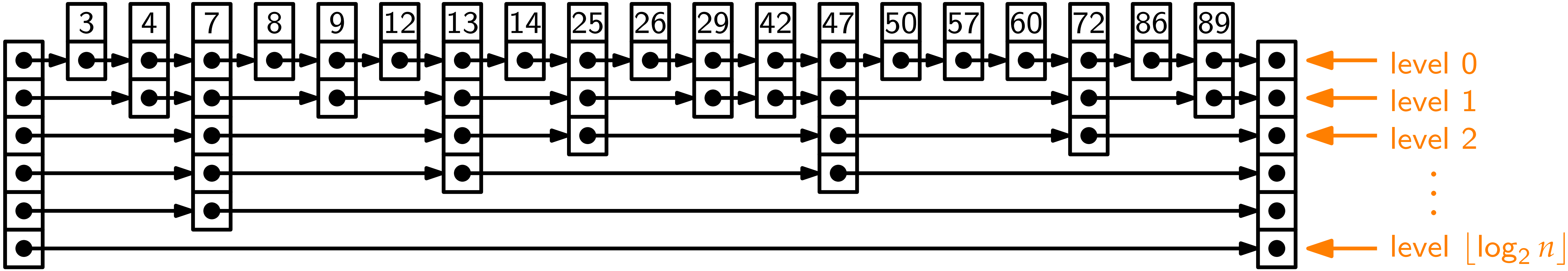
in the previous step we went a level up

current step we take on level i (start with the last step we take on level i; we don't skip levels)

Theorem 1. Searching in a (rand.) skip list can be done in expected $O(\log n)$ time.

(Randomized) Skip Lists

Proof of Theorem 1.



Let X_i be a random variable denoting the number of steps we take on level i or lower.

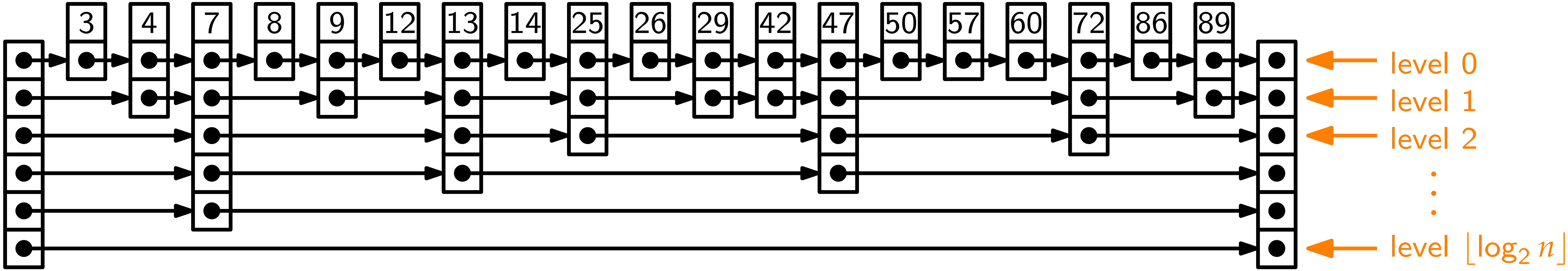
$$E[X_i] = 1 + \tfrac{1}{2}E[X_{i-1}] + \tfrac{1}{2}E[X_i] \quad \Leftrightarrow \quad E[X_i] = 2 + E[X_{i-1}] \qquad (\text{for } i < 0\text{: } E[X_i] = 0)$$

$$\Rightarrow \quad E[X_i] = 2 + 2 + E[X_{i-2}] = 6 + E[X_{i-3}] = \ldots = 2i + E[X_0] = 2i + 2$$

Theorem 1. Searching in a (rand.) skip list can be done in expected $O(\log n)$ time.

(Randomized) Skip Lists

Proof of Theorem 1.



Let X_i be a random variable denoting the number of steps we take on level i or lower.

$$E[X_i] = 1 + \tfrac{1}{2}E[X_{i-1}] + \tfrac{1}{2}E[X_i] \;\Leftrightarrow\; E[X_i] = 2 + E[X_{i-1}] \qquad (\text{for } i < 0\colon E[X_i] = 0)$$

$$\Rightarrow \quad E[X_i] = 2 + 2 + E[X_{i-2}] = 6 + E[X_{i-3}] = \ldots = 2i + E[X_0] = 2i + 2$$

$$\Rightarrow \quad E[X_{\lfloor \log_2 n \rfloor}] = 2\lfloor \log_2 n \rfloor + 2 \in O(\log n) \qquad \square$$

Theorem 1. Searching in a (rand.) skip list can be done in expected $O(\log n)$ time.

Checking Containment in a Set

Say you are given a (large) set of n (long) keys, which are represented as numbers. What would you do to answer queries of whether a key is contained in your set quickly?

Checking Containment in a Set

Say you are given a (large) set of n (long) keys, which are represented as numbers. What would you do to answer queries of whether a key is contained in your set quickly?

Store the keys in . . .

- array or linked list:
  - $(-)$ $\Theta(n)$ time for containment check ($\Theta(\log n)$ for a sorted array)
  - $(\circ)$ simple data structure with low space consumption
  - $(-)$ adding/removing keys takes $\Theta(n)$ time

Checking Containment in a Set

Say you are given a (large) set of n (long) keys, which are represented as numbers. What would you do to answer queries of whether a key is contained in your set quickly?

Store the keys in . . .

- array or linked list:
  - $(-)$ $\Theta(n)$ time for containment check ($\Theta(\log n)$ for a sorted array)
  - $(\circ)$ simple data structure with low space consumption
  - $(-)$ adding/removing keys takes $\Theta(n)$ time
- balanced binary search tree or skip list:
  - $(\circ)$ $\Theta(\log n)$ time for containment check
  - $(\circ)$ not too complicated data structure and moderate space consumption
  - $(+)$ adding/removing keys takes $\Theta(\log n)$ time

Checking Containment in a Set

Say you are given a (large) set of n (long) keys, which are represented as numbers.
What would you do to answer queries of whether a key is contained in your set quickly?
Store the keys in . . .

- array or linked list:
  - $(-)$ $\Theta(n)$ time for containment check ($\Theta(\log n)$ for a sorted array)
  - $(\circ)$ simple data structure with low space consumption
  - $(-)$ adding/removing keys takes $\Theta(n)$ time
- hash table:
  - $(+)$ usually $\Theta(1)$ time for containment check
  - $(-)$ more complicated and maybe a higher space consumption
  - $(+)$ adding/removing keys takes usually $\Theta(1)$ time
- balanced binary search tree or skip list:
  - $(\circ)$ $\Theta(\log n)$ time for containment check
  - $(\circ)$ not too complicated data structure and moderate space consumption
  - $(+)$ adding/removing keys takes $\Theta(\log n)$ time

Checking Containment in a Set

Say you are given a (large) set of n (long) keys, which are represented as numbers. What would you do to answer queries of whether a key is contained in your set quickly? Store the keys in . . .

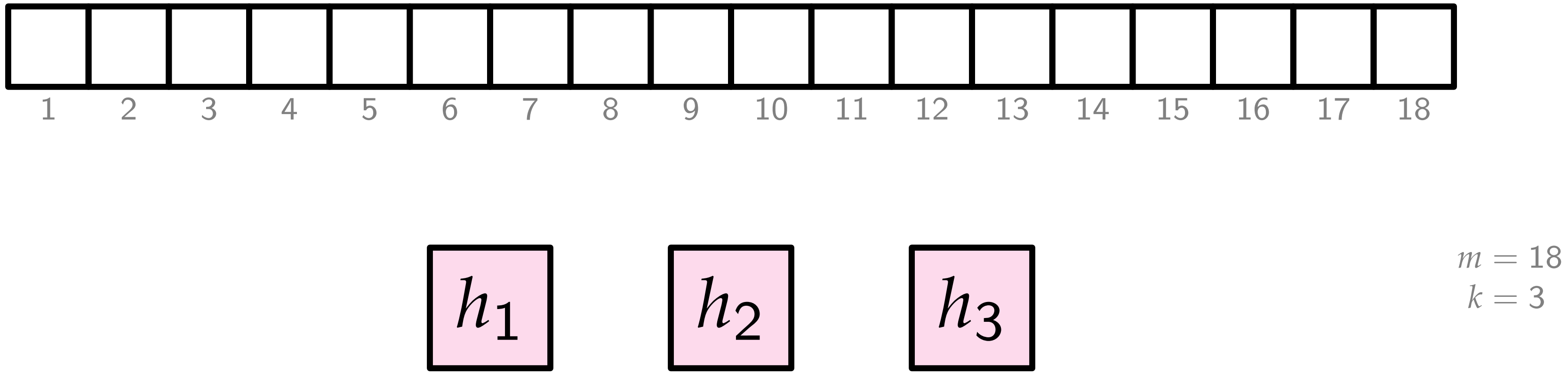
- array or linked list:
  - (−) $\Theta(n)$ time for containment check ($\Theta(\log n)$ for a sorted array)
  - (∘) simple data structure with low space consumption
  - (−) adding/removing keys takes $\Theta(n)$ time
- hash table:
  - (+) usually $\Theta(1)$ time for containment check
  - (−) more complicated and maybe a higher space consumption
  - (+) adding/removing keys takes usually $\Theta(1)$ time
- balanced binary search tree or skip list:
  - (∘) $\Theta(\log n)$ time for containment check
  - (∘) not too complicated data structure and moderate space consumption
  - (+) adding/removing keys takes $\Theta(\log n)$ time
- Bloom filter:
  - (+) $\Theta(1)$ time for containment check
  - (−) may produce false positives
  - (+) very low space consumption that does not depend on the lengths of the keys
  - (−) allows adding keys (in $\Theta(1)$ time), but not removing keys

Bloom Filters

A Bloom filter is a **bit array** of m bits & a set of k different **hash functions** $h_1, \ldots, h_k$. Each hash function h_i generates a uniform random distribution in the range $\{1, \ldots, m\}$.

Bloom Filters

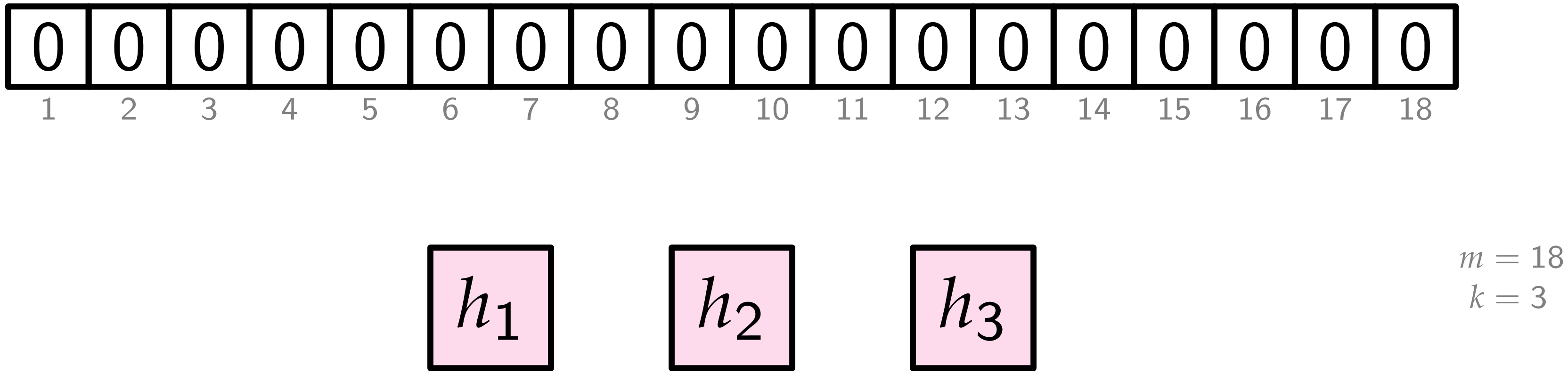
A Bloom filter is a **bit array** of m bits & a set of k different **hash functions** $h_1, \ldots, h_k$. Each hash function h_i generates a uniform random distribution in the range $\{1, \ldots, m\}$.



Bloom Filters

A Bloom filter is a **bit array** of m bits & a set of k different **hash functions** $h_1, \ldots, h_k$.
Each hash function h_i generates a uniform random distribution in the range $\{1, \ldots, m\}$.

Initially the array contains only 0s. Such a Bloom filter represents the empty set.

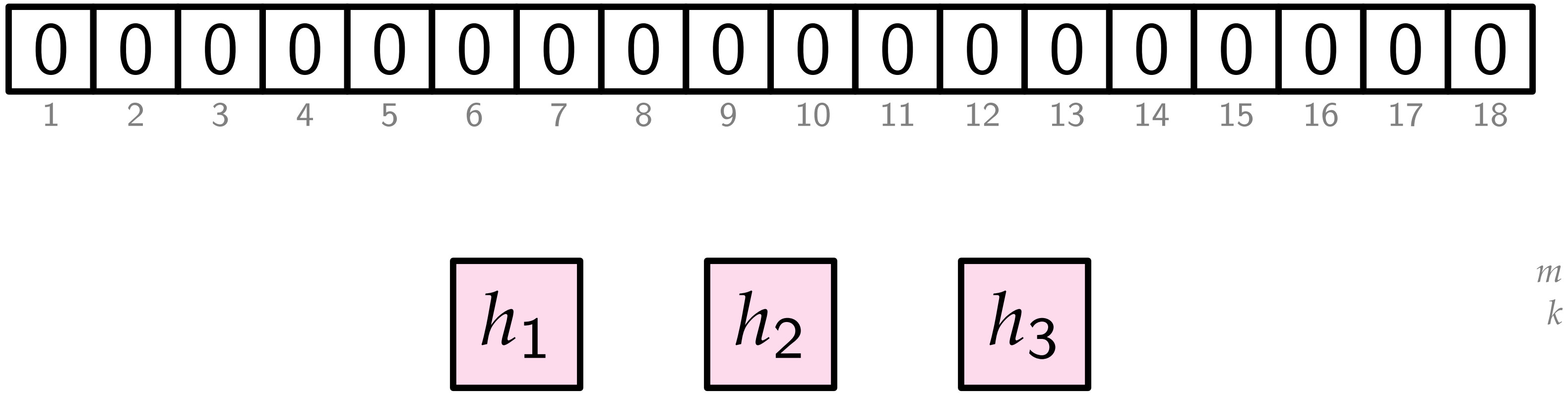


Bloom Filters

A Bloom filter is a **bit array** of m bits & a set of k different **hash functions** $h_1, \dots, h_k$. Each hash function h_i generates a uniform random distribution in the range $\{1, \dots, m\}$.

Initially the array contains only 0s. Such a Bloom filter represents the empty set.

For a set S of numbers, we insert each $s \in S$ to the Bloom filter by setting all bits at the positions $h_1(s)$, $h_2(s)$, . . . , $h_k(s)$ to 1.



0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

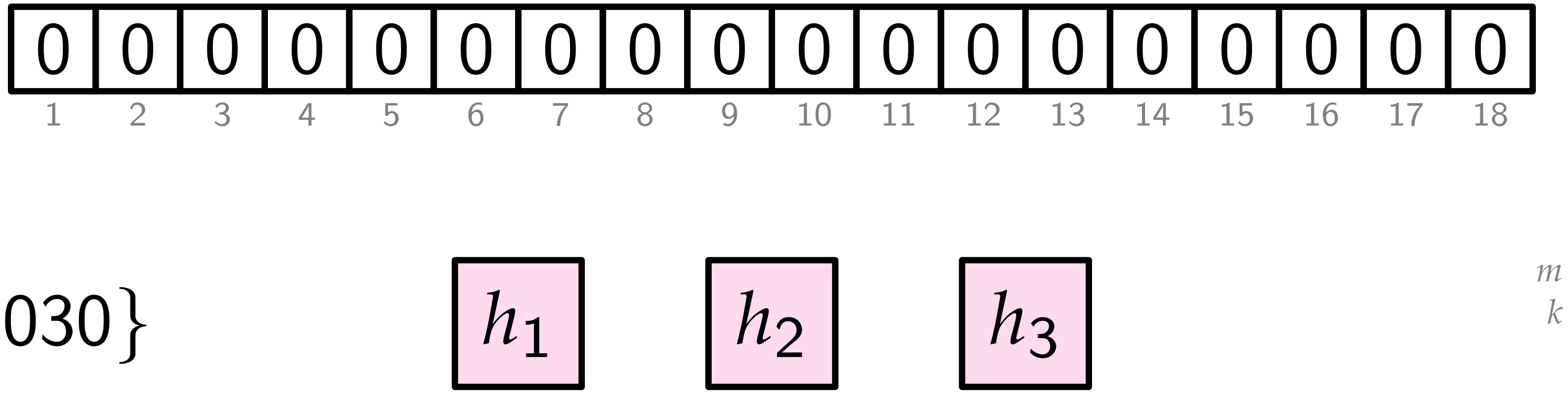
$m = 18$
$k = 3$

Bloom Filters

A Bloom filter is a **bit array** of m bits & a set of k different **hash functions** $h_1, \ldots, h_k$. Each hash function h_i generates a uniform random distribution in the range $\{1, \ldots, m\}$.

Initially the array contains only 0s. Such a Bloom filter represents the empty set.

For a set S of numbers, we insert each $s \in S$ to the Bloom filter by setting all bits at the positions $h_1(s)$, $h_2(s)$, . . . , $h_k(s)$ to 1.



0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

$S = \{2345, 8234, 12492, 34030\}$

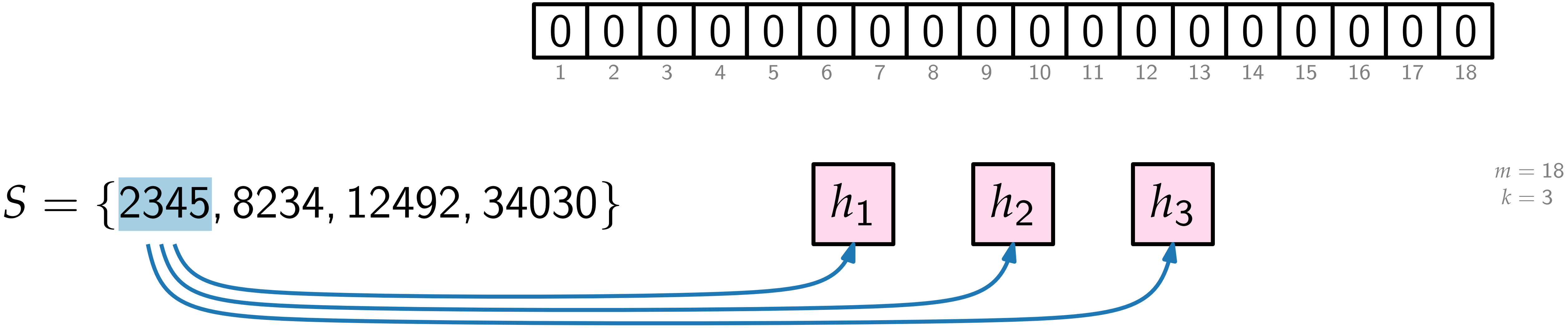
$m = 18$
$k = 3$

Bloom Filters

A Bloom filter is a **bit array** of m bits & a set of k different **hash functions** $h_1, \ldots, h_k$. Each hash function h_i generates a uniform random distribution in the range $\{1, \ldots, m\}$.

Initially the array contains only 0s. Such a Bloom filter represents the empty set.

For a set S of numbers, we insert each $s \in S$ to the Bloom filter by setting all bits at the positions $h_1(s)$, $h_2(s)$, …, $h_k(s)$ to 1.



0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

$S = \{2345, 8234, 12492, 34030\}$

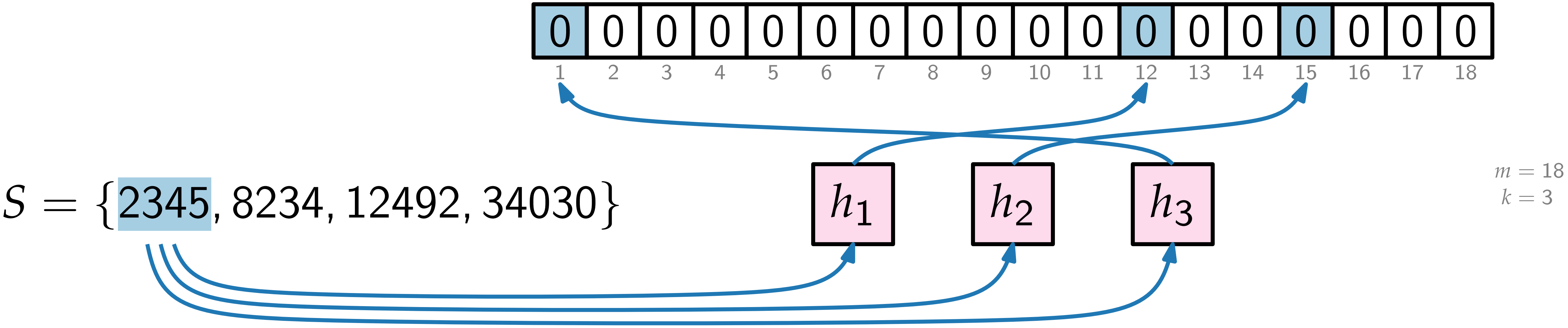
$m = 18$
$k = 3$

Bloom Filters

A Bloom filter is a **bit array** of m bits & a set of k different **hash functions** $h_1, \ldots, h_k$. Each hash function h_i generates a uniform random distribution in the range $\{1, \ldots, m\}$.

Initially the array contains only 0s. Such a Bloom filter represents the empty set.

For a set S of numbers, we insert each $s \in S$ to the Bloom filter by setting all bits at the positions $h_1(s)$, $h_2(s)$, ..., $h_k(s)$ to 1.

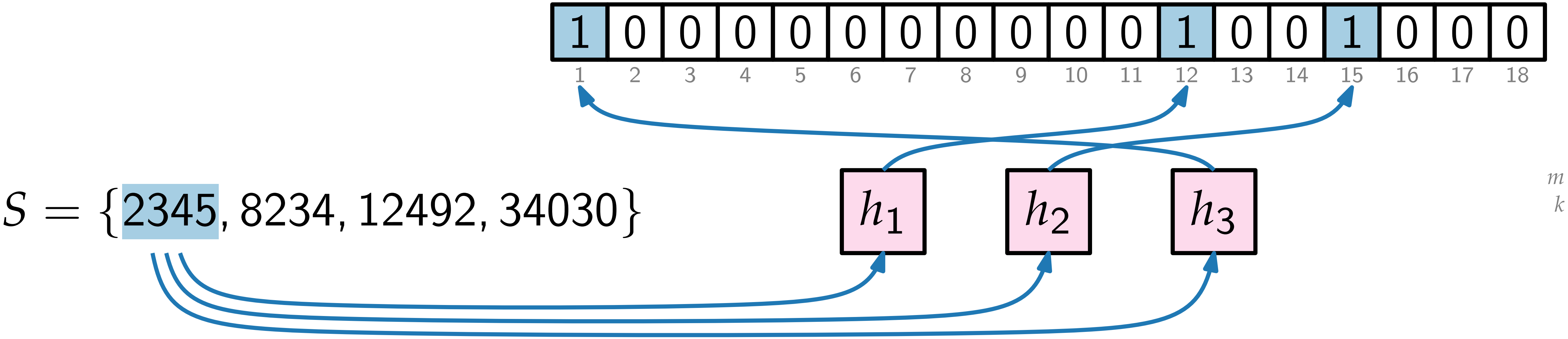


Bloom Filters

A Bloom filter is a **bit array** of m bits & a set of k different **hash functions** $h_1, \dots, h_k$. Each hash function h_i generates a uniform random distribution in the range $\{1, \dots, m\}$.

Initially the array contains only 0s. Such a Bloom filter represents the empty set.

For a set S of numbers, we insert each $s \in S$ to the Bloom filter by setting all bits at the positions $h_1(s)$, $h_2(s)$, …, $h_k(s)$ to 1.



1	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

$S = \{2345, 8234, 12492, 34030\}$

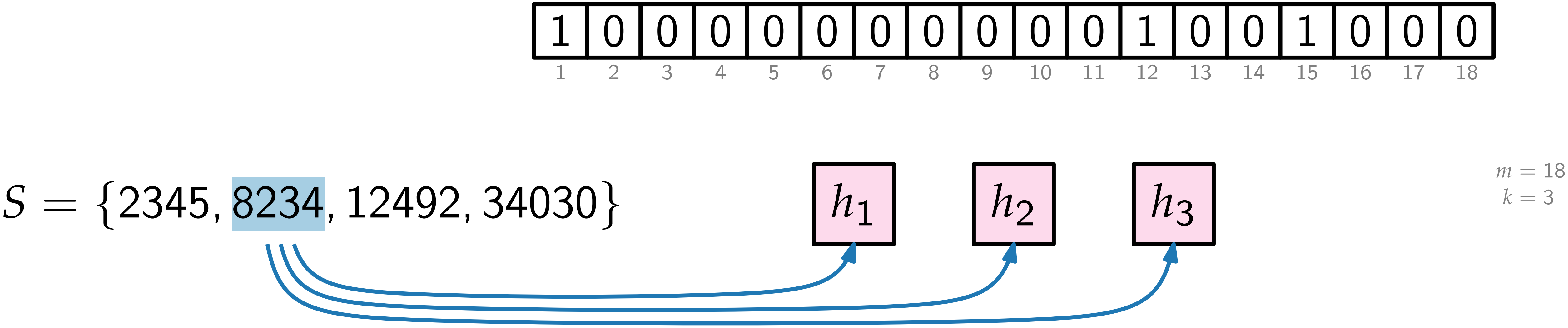
$m = 18$
$k = 3$

Bloom Filters

A Bloom filter is a **bit array** of m bits & a set of k different **hash functions** $h_1, \ldots, h_k$. Each hash function h_i generates a uniform random distribution in the range $\{1, \ldots, m\}$.

Initially the array contains only 0s. Such a Bloom filter represents the empty set.

For a set S of numbers, we insert each $s \in S$ to the Bloom filter by setting all bits at the positions $h_1(s)$, $h_2(s)$, …, $h_k(s)$ to 1.



1	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

$S = \{2345, 8234, 12492, 34030\}$

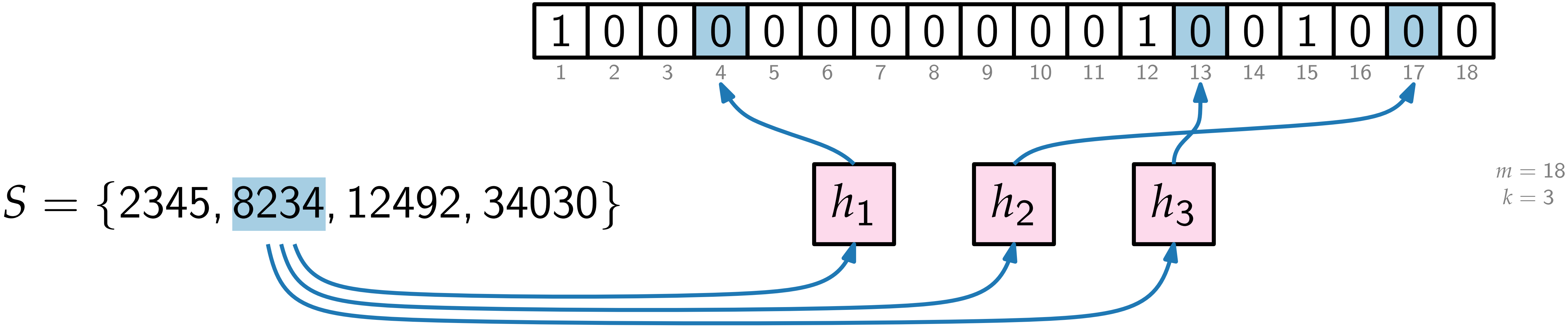
$m = 18$
$k = 3$

Bloom Filters

A Bloom filter is a **bit array** of m bits & a set of k different **hash functions** $h_1, \ldots, h_k$. Each hash function h_i generates a uniform random distribution in the range $\{1, \ldots, m\}$.

Initially the array contains only 0s. Such a Bloom filter represents the empty set.

For a set S of numbers, we insert each $s \in S$ to the Bloom filter by setting all bits at the positions $h_1(s)$, $h_2(s)$, $\ldots$, $h_k(s)$ to 1.



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0

$S = \{2345, 8234, 12492, 34030\}$

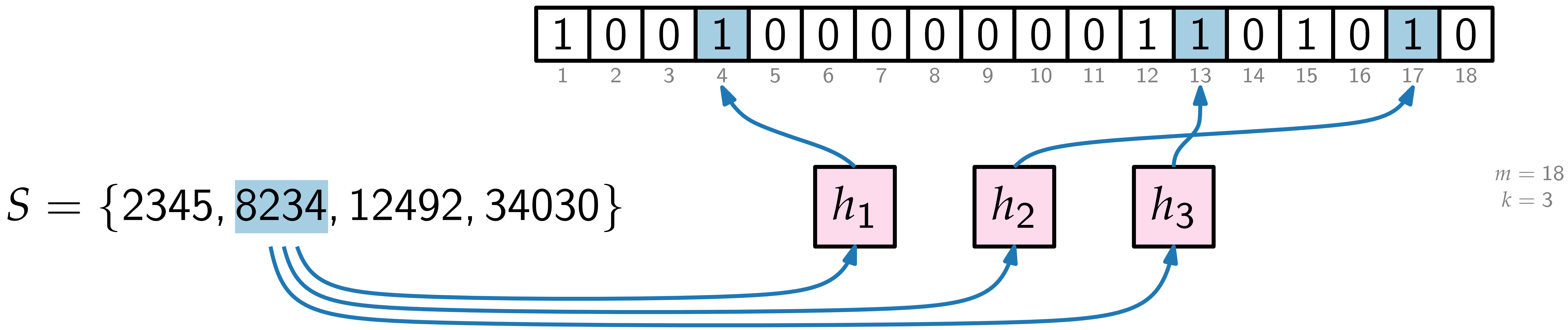
$m = 18$
$k = 3$

Bloom Filters

A Bloom filter is a **bit array** of m bits & a set of k different **hash functions** $h_1, \dots, h_k$. Each hash function h_i generates a uniform random distribution in the range $\{1, \dots, m\}$.

Initially the array contains only 0s. Such a Bloom filter represents the empty set.

For a set S of numbers, we insert each $s \in S$ to the Bloom filter by setting all bits at the positions $h_1(s)$, $h_2(s)$, …, $h_k(s)$ to 1.



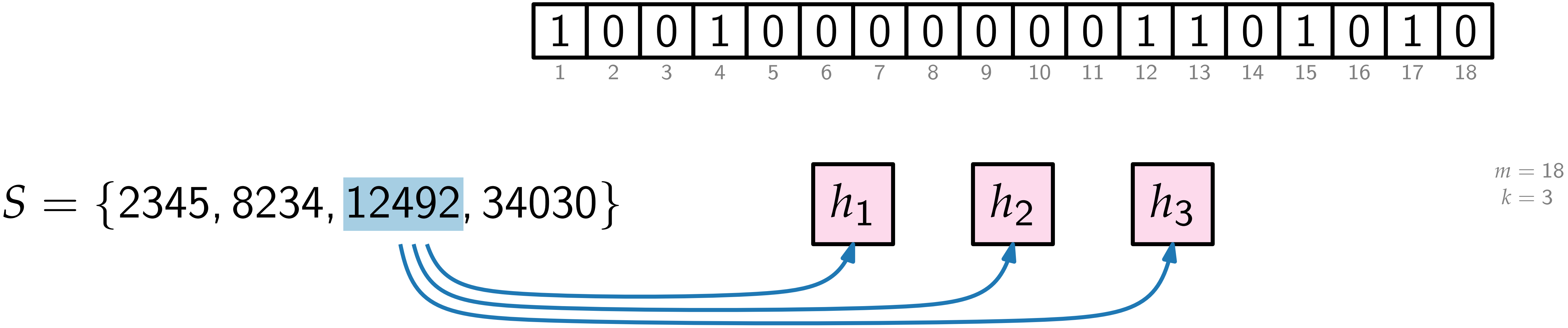
$S = \{2345, 8234, 12492, 34030\}$

Bloom Filters

A Bloom filter is a **bit array** of m bits & a set of k different **hash functions** $h_1, \dots, h_k$. Each hash function h_i generates a uniform random distribution in the range $\{1, \dots, m\}$.

Initially the array contains only 0s. Such a Bloom filter represents the empty set.

For a set S of numbers, we insert each $s \in S$ to the Bloom filter by setting all bits at the positions $h_1(s)$, $h_2(s)$, …, $h_k(s)$ to 1.



1	0	0	1	0	0	0	0	0	0	0	1	1	0	1	0	1	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

$S = \{2345, 8234, 12492, 34030\}$

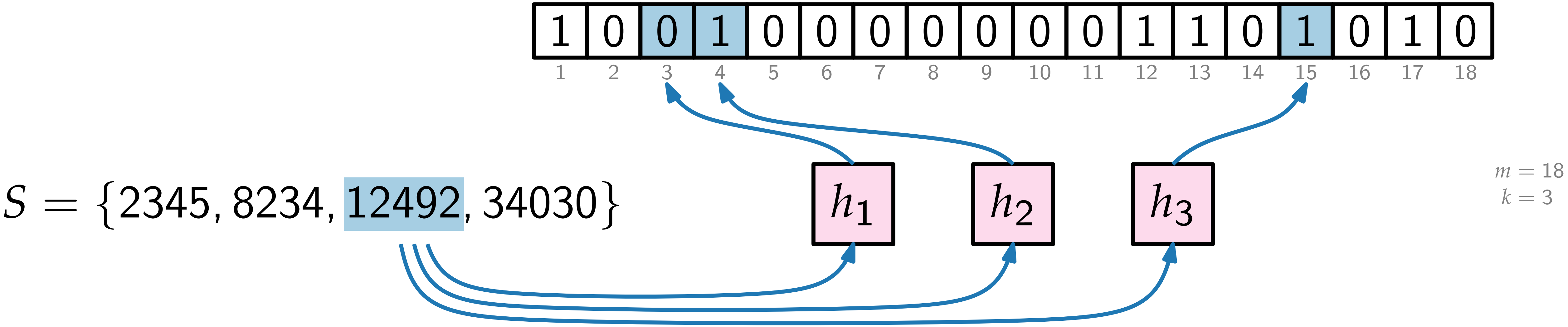
$m = 18$
$k = 3$

Bloom Filters

A Bloom filter is a **bit array** of m bits & a set of k different **hash functions** $h_1, \ldots, h_k$. Each hash function h_i generates a uniform random distribution in the range $\{1, \ldots, m\}$.

Initially the array contains only 0s. Such a Bloom filter represents the empty set.

For a set S of numbers, we insert each $s \in S$ to the Bloom filter by setting all bits at the positions $h_1(s)$, $h_2(s)$, ..., $h_k(s)$ to 1.



$S = \{2345, 8234, 12492, 34030\}$

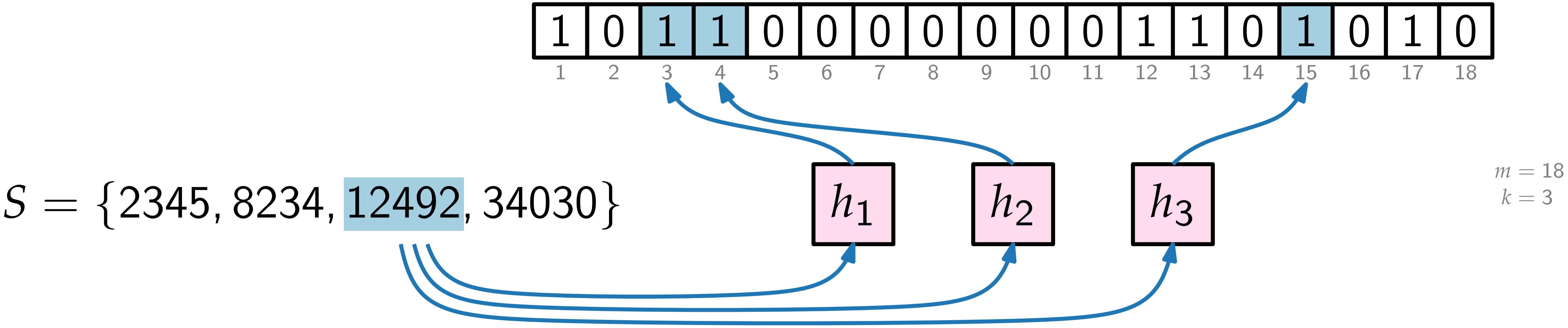
$m = 18$
$k = 3$

Bloom Filters

A Bloom filter is a **bit array** of m bits & a set of k different **hash functions** $h_1, \ldots, h_k$. Each hash function h_i generates a uniform random distribution in the range $\{1, \ldots, m\}$.

Initially the array contains only 0s. Such a Bloom filter represents the empty set.

For a set S of numbers, we insert each $s \in S$ to the Bloom filter by setting all bits at the positions $h_1(s)$, $h_2(s)$, . . . , $h_k(s)$ to 1.



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	0	1	1	0	0	0	0	0	0	0	1	1	0	1	0	1	0

$S = \{2345, 8234, 12492, 34030\}$

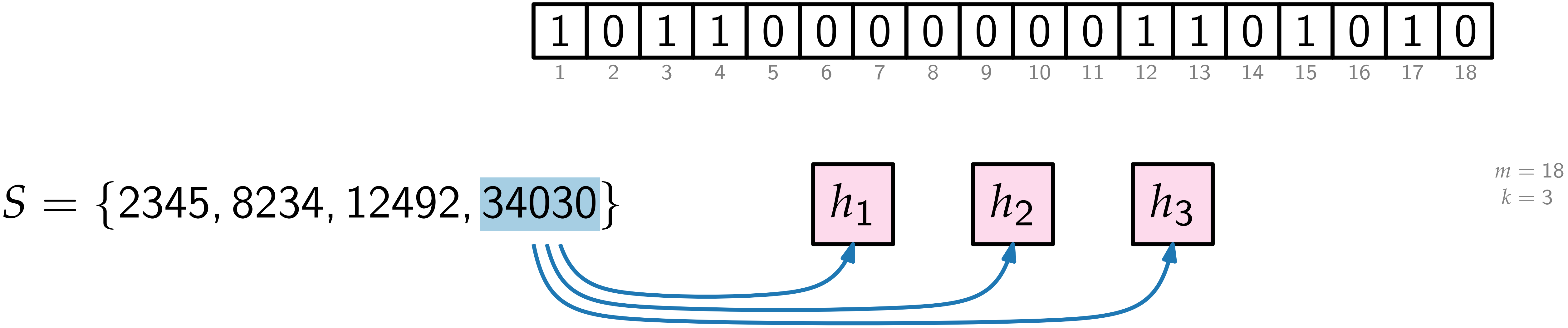
$m = 18$
$k = 3$

Bloom Filters

A Bloom filter is a **bit array** of m bits & a set of k different **hash functions** $h_1, \ldots, h_k$. Each hash function h_i generates a uniform random distribution in the range $\{1, \ldots, m\}$.

Initially the array contains only 0s. Such a Bloom filter represents the empty set.

For a set S of numbers, we insert each $s \in S$ to the Bloom filter by setting all bits at the positions $h_1(s)$, $h_2(s)$, ..., $h_k(s)$ to 1.



1	0	1	1	0	0	0	0	0	0	0	1	1	0	1	0	1	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

$S = \{2345, 8234, 12492, 34030\}$

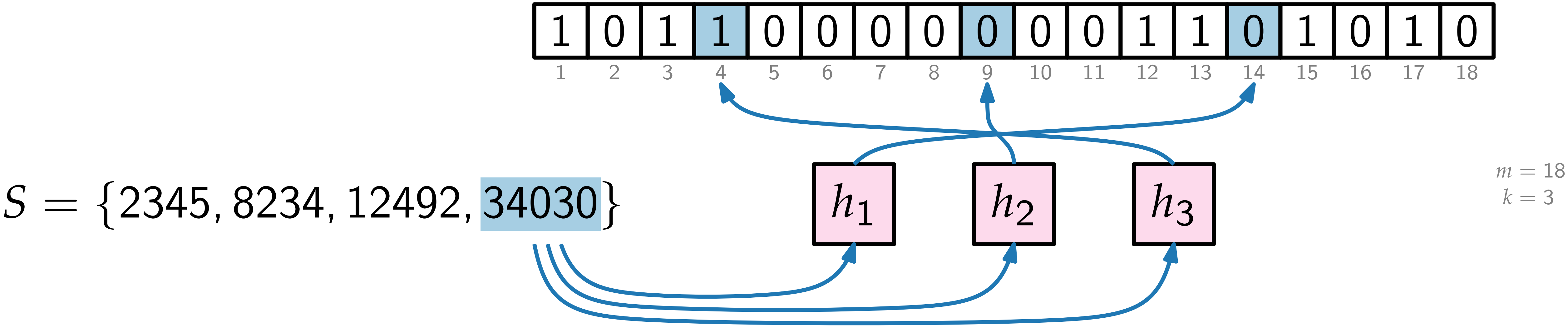
$m = 18$
$k = 3$

Bloom Filters

A Bloom filter is a **bit array** of m bits & a set of k different **hash functions** $h_1, \dots, h_k$. Each hash function h_i generates a uniform random distribution in the range $\{1, \dots, m\}$.

Initially the array contains only 0s. Such a Bloom filter represents the empty set.

For a set S of numbers, we insert each $s \in S$ to the Bloom filter by setting all bits at the positions $h_1(s)$, $h_2(s)$, ..., $h_k(s)$ to 1.



$S = \{2345, 8234, 12492, 34030\}$

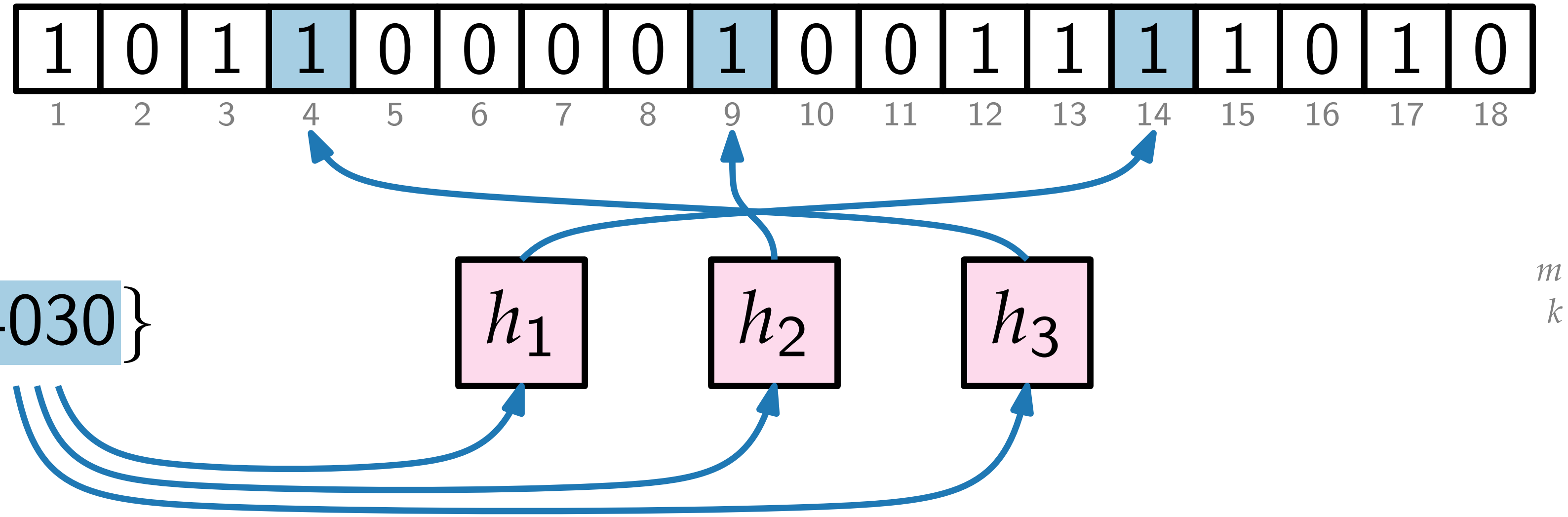
$m = 18$
$k = 3$

Bloom Filters

A Bloom filter is a **bit array** of m bits & a set of k different **hash functions** $h_1, \ldots, h_k$. Each hash function h_i generates a uniform random distribution in the range $\{1, \ldots, m\}$.

Initially the array contains only 0s. Such a Bloom filter represents the empty set.

For a set S of numbers, we insert each $s \in S$ to the Bloom filter by setting all bits at the positions $h_1(s)$, $h_2(s)$, …, $h_k(s)$ to 1.



1	0	1	1	0	0	0	0	1	0	0	1	1	1	1	0	1	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

$S = \{2345, 8234, 12492, 34030\}$

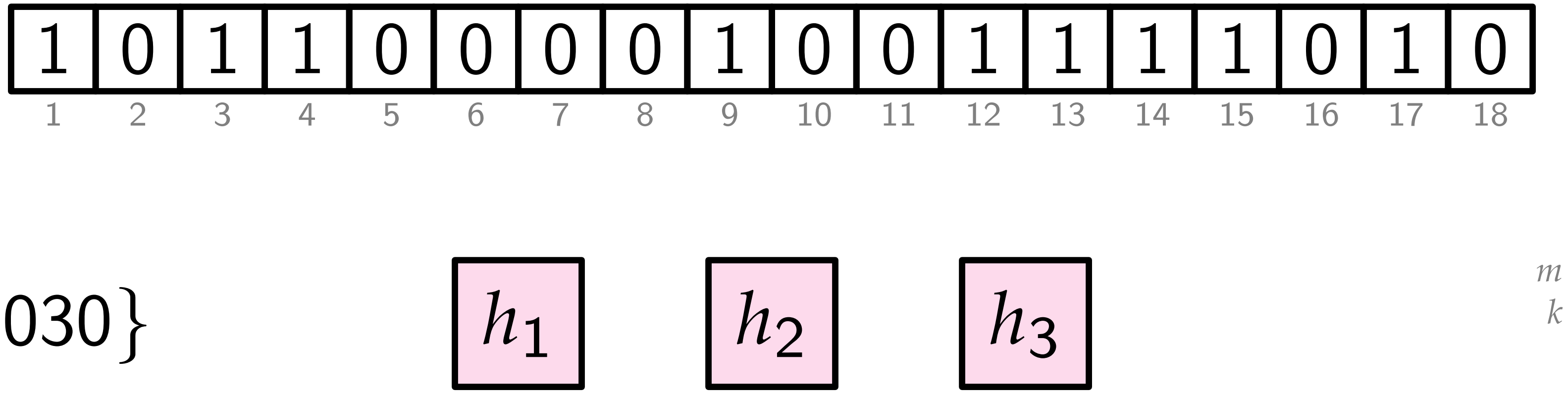
$m = 18$
$k = 3$

Bloom Filters

A Bloom filter is a **bit array** of m bits & a set of k different **hash functions** $h_1, \ldots, h_k$. Each hash function h_i generates a uniform random distribution in the range $\{1, \ldots, m\}$.

Initially the array contains only 0s. Such a Bloom filter represents the empty set.

For a set S of numbers, we insert each $s \in S$ to the Bloom filter by setting all bits at the positions $h_1(s)$, $h_2(s)$, . . . , $h_k(s)$ to 1.



1	0	1	1	0	0	0	0	1	0	0	1	1	1	1	0	1	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

$S = \{2345, 8234, 12492, 34030\}$

$m = 18$
$k = 3$

Bloom Filters

A Bloom filter is a **bit array** of m bits & a set of k different **hash functions** $h_1, \ldots, h_k$. Each hash function h_i generates a uniform random distribution in the range $\{1, \ldots, m\}$.

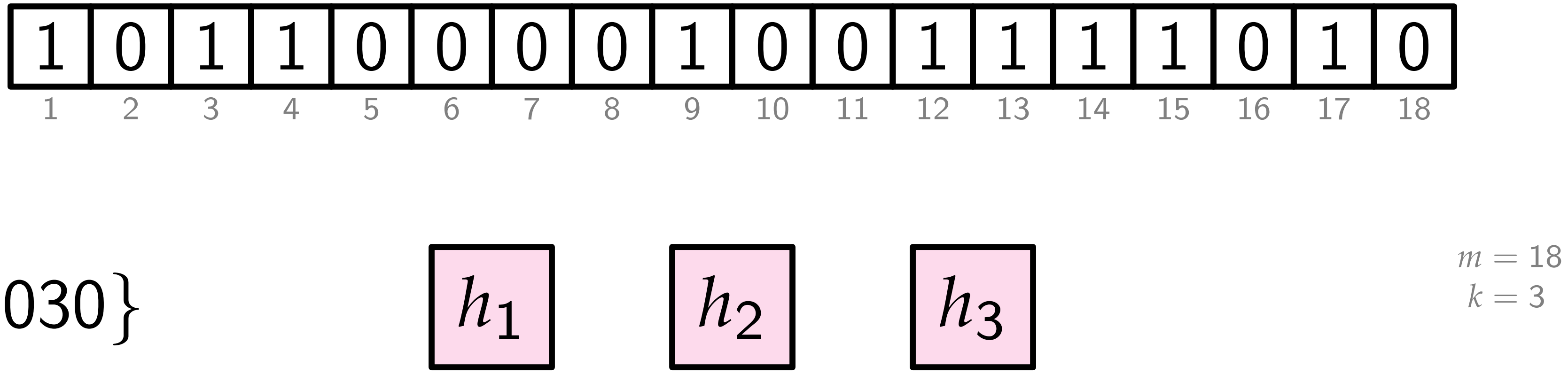
Initially the array contains only 0s. Such a Bloom filter represents the empty set.

For a set S of numbers, we insert each $s \in S$ to the Bloom filter by setting all bits at the positions $h_1(s)$, $h_2(s)$, . . . , $h_k(s)$ to 1.

Containment check of a number a: check the bits at positions $h_1(a), h_2(a), \ldots, h_k(a)$:

- if there is a 0, a is for sure not in S.
- if there are only 1s, a may be in S (it is in S with a small error probability).

$S = \{2345, 8234, 12492, 34030\}$



1	0	1	1	0	0	0	0	1	0	0	1	1	1	1	0	1	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

$m = 18$
$k = 3$

Bloom Filters

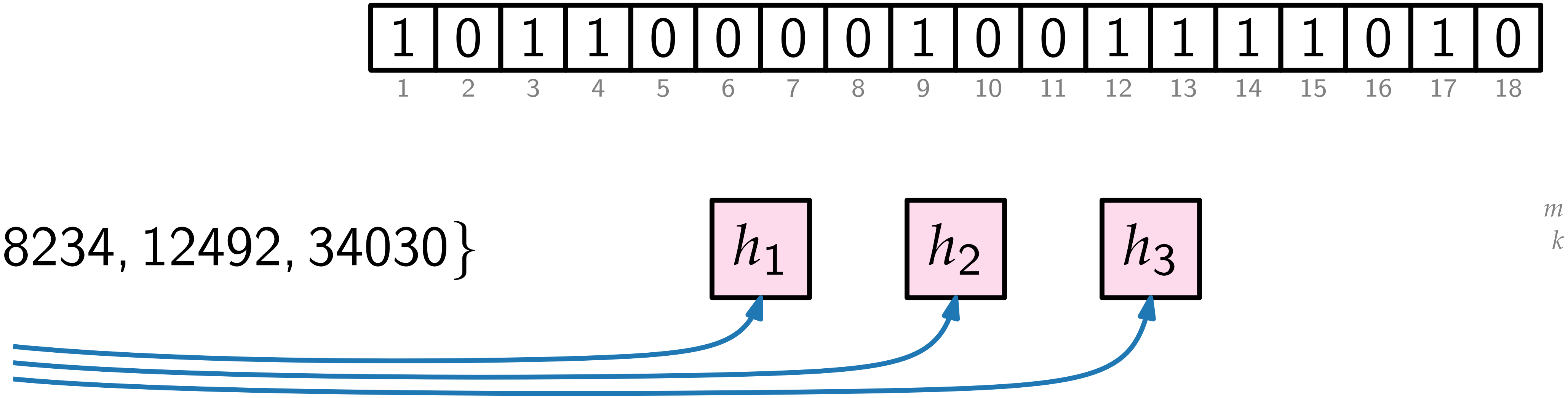
A Bloom filter is a **bit array** of m bits & a set of k different **hash functions** $h_1, \ldots, h_k$. Each hash function h_i generates a uniform random distribution in the range $\{1, \ldots, m\}$.

Initially the array contains only 0s. Such a Bloom filter represents the empty set.

For a set S of numbers, we insert each $s \in S$ to the Bloom filter by setting all bits at the positions $h_1(s)$, $h_2(s)$, . . . , $h_k(s)$ to 1.

Containment check of a number a: check the bits at positions $h_1(a), h_2(a), \ldots, h_k(a)$:

- if there is a 0, a is for sure not in S.
- if there are only 1s, a may be in S (it is in S with a small error probability).



1	0	1	1	0	0	0	0	1	0	0	1	1	1	1	0	1	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

$m = 18$
$k = 3$

$S = \{2345, 8234, 12492, 34030\}$

$a = 2346$

Bloom Filters

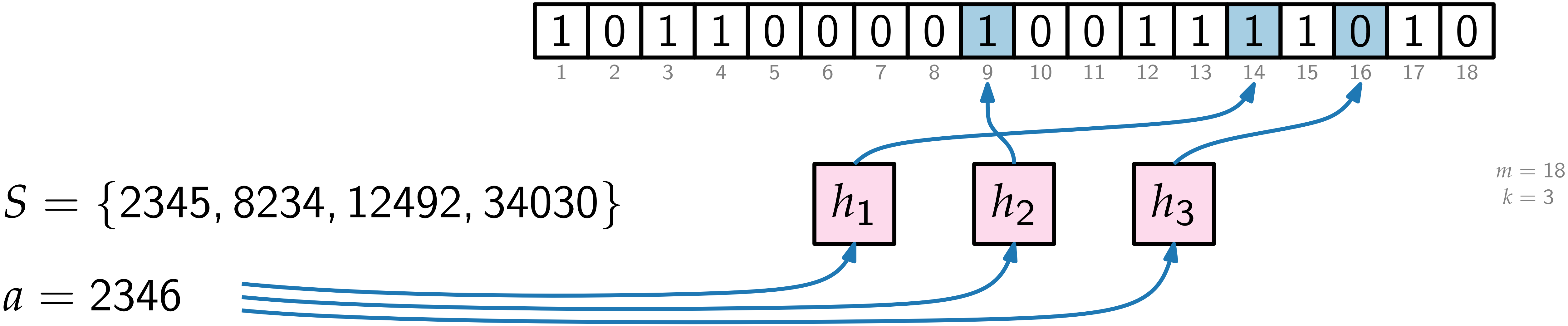
A Bloom filter is a **bit array** of m bits & a set of k different **hash functions** $h_1, \ldots, h_k$. Each hash function h_i generates a uniform random distribution in the range $\{1, \ldots, m\}$.

Initially the array contains only 0s. Such a Bloom filter represents the empty set.

For a set S of numbers, we insert each $s \in S$ to the Bloom filter by setting all bits at the positions $h_1(s)$, $h_2(s)$, ..., $h_k(s)$ to 1.

Containment check of a number a: check the bits at positions $h_1(a), h_2(a), \ldots, h_k(a)$:

- if there is a 0, a is for sure not in S.
- if there are only 1s, a may be in S (it is in S with a small error probability).



$S = \{2345, 8234, 12492, 34030\}$

$a = 2346$

Bloom Filters

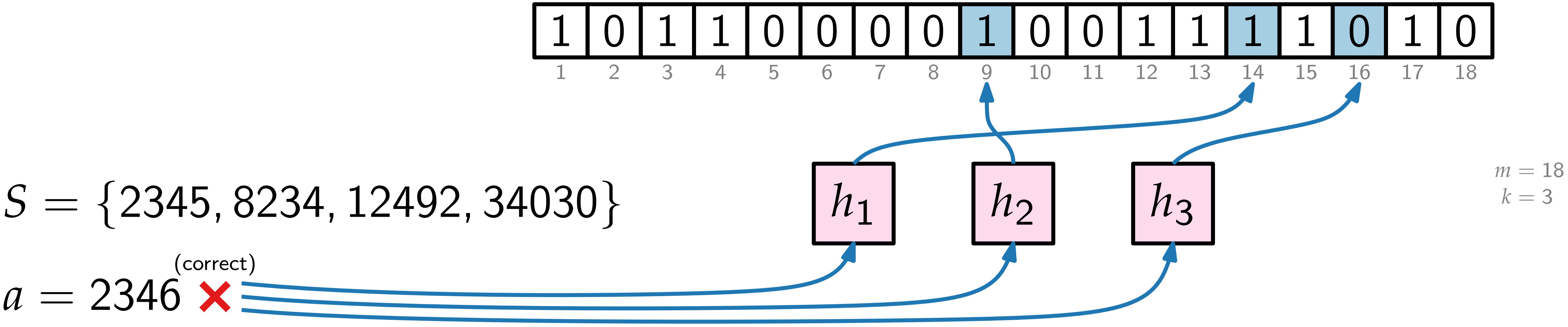
A Bloom filter is a **bit array** of m bits & a set of k different **hash functions** $h_1, \ldots, h_k$. Each hash function h_i generates a uniform random distribution in the range $\{1, \ldots, m\}$.

Initially the array contains only 0s. Such a Bloom filter represents the empty set.

For a set S of numbers, we insert each $s \in S$ to the Bloom filter by setting all bits at the positions $h_1(s)$, $h_2(s)$, …, $h_k(s)$ to 1.

Containment check of a number a: check the bits at positions $h_1(a), h_2(a), \ldots, h_k(a)$:

- if there is a 0, a is for sure not in S.
- if there are only 1s, a may be in S (it is in S with a small error probability).



1	0	1	1	0	0	0	0	1	0	0	1	1	1	1	0	1	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

$S = \{2345, 8234, 12492, 34030\}$

$a = 2346$ ✗ (correct)

$m = 18$
$k = 3$

Bloom Filters

A Bloom filter is a **bit array** of m bits & a set of k different **hash functions** $h_1, \dots, h_k$. Each hash function h_i generates a uniform random distribution in the range $\{1, \dots, m\}$.

Initially the array contains only 0s. Such a Bloom filter represents the empty set.

For a set S of numbers, we insert each $s \in S$ to the Bloom filter by setting all bits at the positions $h_1(s)$, $h_2(s)$, ..., $h_k(s)$ to 1.

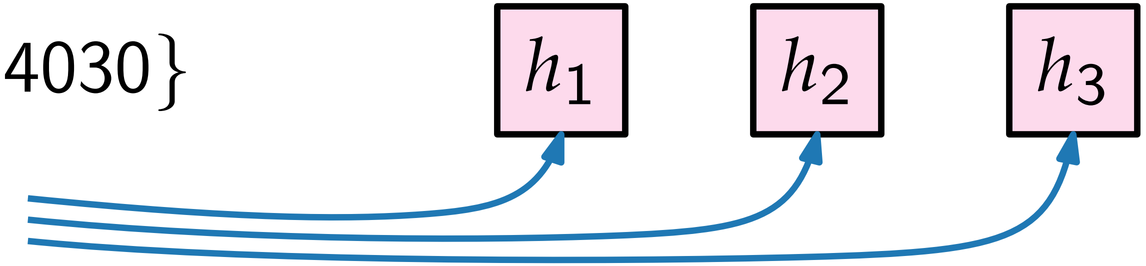
Containment check of a number a: check the bits at positions $h_1(a), h_2(a), \dots, h_k(a)$:

- if there is a 0, a is for sure not in S.
- if there are only 1s, a may be in S (it is in S with a small error probability).

1	0	1	1	0	0	0	0	1	0	0	1	1	1	1	0	1	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

$m = 18$
$k = 3$

$S = \{2345, 8234, 12492, 34030\}$



$a = 2346$ ✗ (correct) $b = 8234$

Bloom Filters

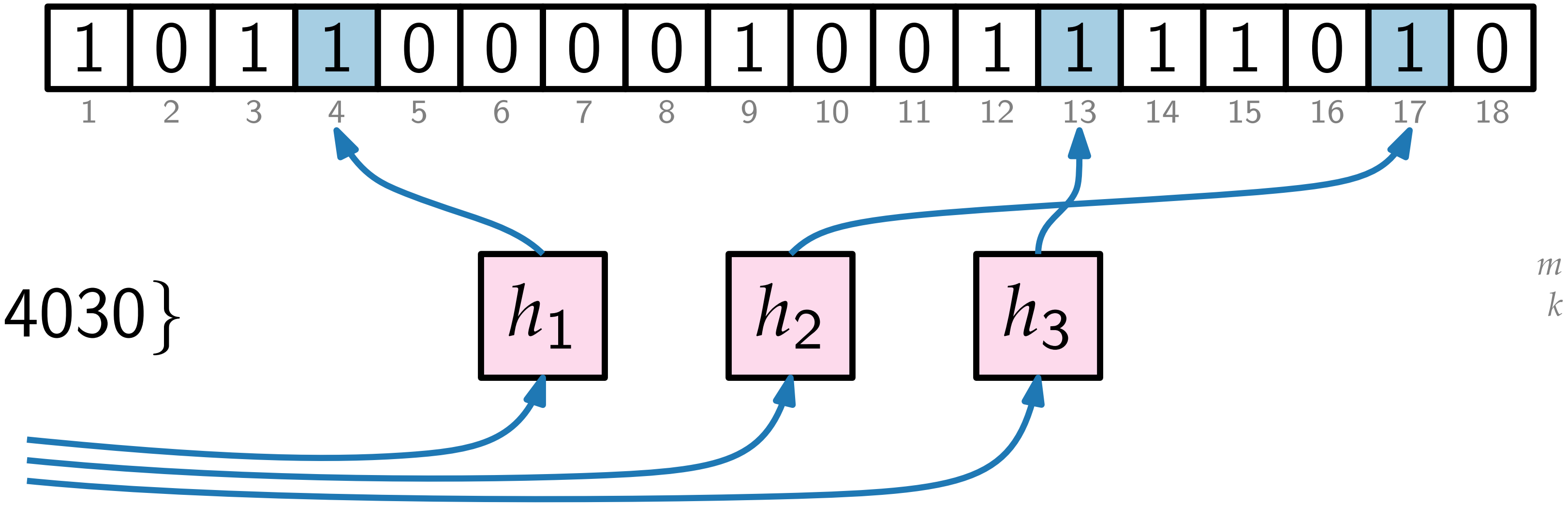
A Bloom filter is a **bit array** of m bits & a set of k different **hash functions** $h_1, \ldots, h_k$. Each hash function h_i generates a uniform random distribution in the range $\{1, \ldots, m\}$.

Initially the array contains only 0s. Such a Bloom filter represents the empty set.

For a set S of numbers, we insert each $s \in S$ to the Bloom filter by setting all bits at the positions $h_1(s)$, $h_2(s)$, …, $h_k(s)$ to 1.

Containment check of a number a: check the bits at positions $h_1(a), h_2(a), \ldots, h_k(a)$:

- if there is a 0, a is for sure not in S.
- if there are only 1s, a may be in S (it is in S with a small error probability).



1	0	1	1	0	0	0	0	1	0	0	1	1	1	1	0	1	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

$m = 18$
$k = 3$

$S = \{2345, 8234, 12492, 34030\}$

$a = 2346$ ✗ (correct) $b = 8234$

Bloom Filters

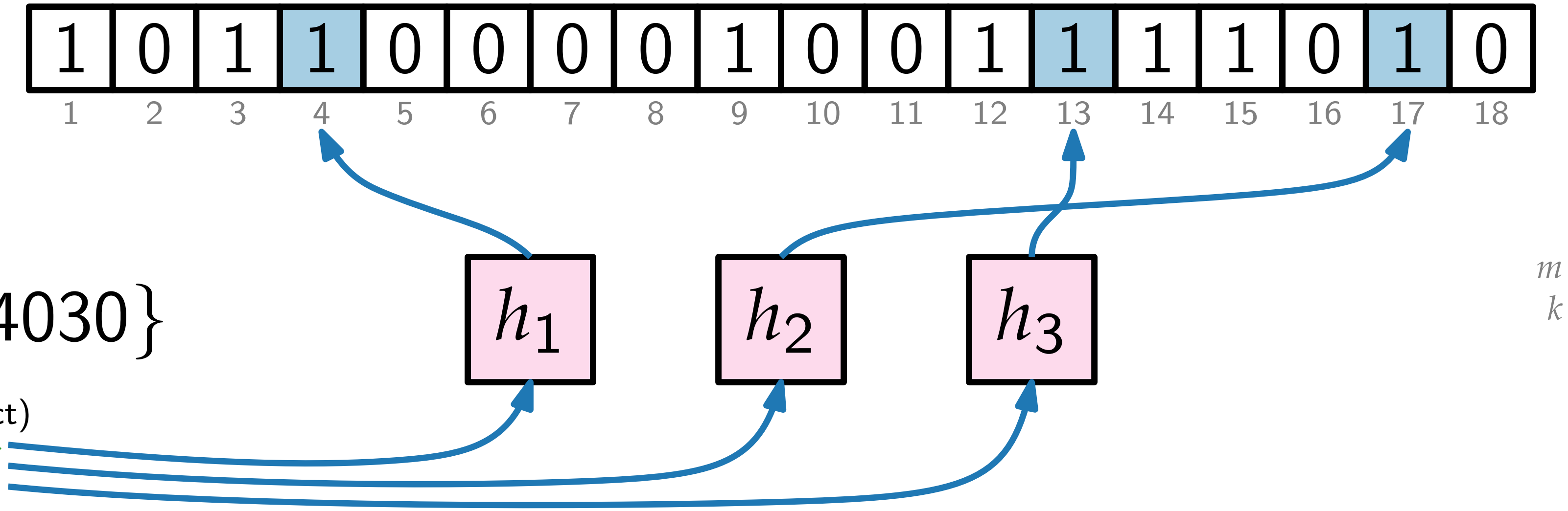
A Bloom filter is a **bit array** of m bits & a set of k different **hash functions** $h_1, \dots, h_k$. Each hash function h_i generates a uniform random distribution in the range $\{1, \dots, m\}$.

Initially the array contains only 0s. Such a Bloom filter represents the empty set.

For a set S of numbers, we insert each $s \in S$ to the Bloom filter by setting all bits at the positions $h_1(s)$, $h_2(s)$, ..., $h_k(s)$ to 1.

Containment check of a number a: check the bits at positions $h_1(a), h_2(a), \dots, h_k(a)$:

- if there is a 0, a is for sure not in S.
- if there are only 1s, a may be in S (it is in S with a small error probability).



1	0	1	1	0	0	0	0	1	0	0	1	1	1	1	0	1	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

$S = \{2345, 8234, 12492, 34030\}$

$a = 2346$ ✗ (correct) $b = 8234$ ✓ (correct)

$m = 18$
$k = 3$

Bloom Filters

A Bloom filter is a **bit array** of m bits & a set of k different **hash functions** $h_1, \ldots, h_k$. Each hash function h_i generates a uniform random distribution in the range $\{1, \ldots, m\}$.

Initially the array contains only 0s. Such a Bloom filter represents the empty set.

For a set S of numbers, we insert each $s \in S$ to the Bloom filter by setting all bits at the positions $h_1(s)$, $h_2(s)$, …, $h_k(s)$ to 1.

Containment check of a number a: check the bits at positions $h_1(a), h_2(a), \ldots, h_k(a)$:

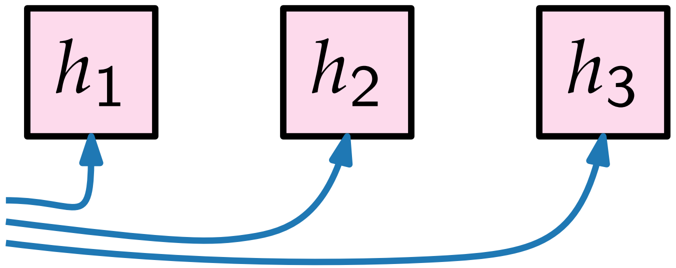
- if there is a 0, a is for sure not in S.
- if there are only 1s, a may be in S (it is in S with a small error probability).

1	0	1	1	0	0	0	0	1	0	0	1	1	1	1	0	1	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

$m = 18$
$k = 3$

$S = \{2345, 8234, 12492, 34030\}$

$a = 2346$ (correct) ✗ $b = 8234$ (correct) ✓ $c = 7042$



Bloom Filters

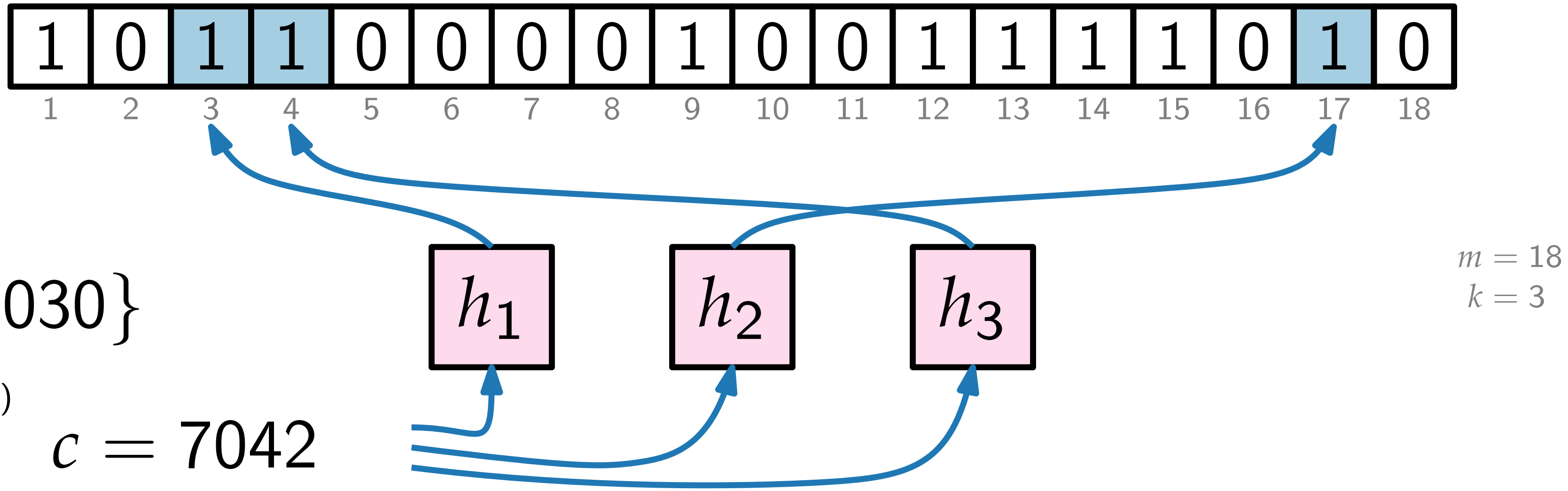
A Bloom filter is a **bit array** of m bits & a set of k different **hash functions** $h_1, \ldots, h_k$.
Each hash function h_i generates a uniform random distribution in the range $\{1, \ldots, m\}$.

Initially the array contains only 0s. Such a Bloom filter represents the empty set.

For a set S of numbers, we insert each $s \in S$ to the Bloom filter by setting all bits at the positions $h_1(s)$, $h_2(s)$, ..., $h_k(s)$ to 1.

Containment check of a number a: check the bits at positions $h_1(a), h_2(a), \ldots, h_k(a)$:

- if there is a 0, a is for sure not in S.
- if there are only 1s, a may be in S (it is in S with a small error probability).



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	0	1	1	0	0	0	0	1	0	0	1	1	1	1	0	1	0

$S = \{2345, 8234, 12492, 34030\}$

$a = 2346$ (correct) ✗ $b = 8234$ (correct) ✓ $c = 7042$

Bloom Filters

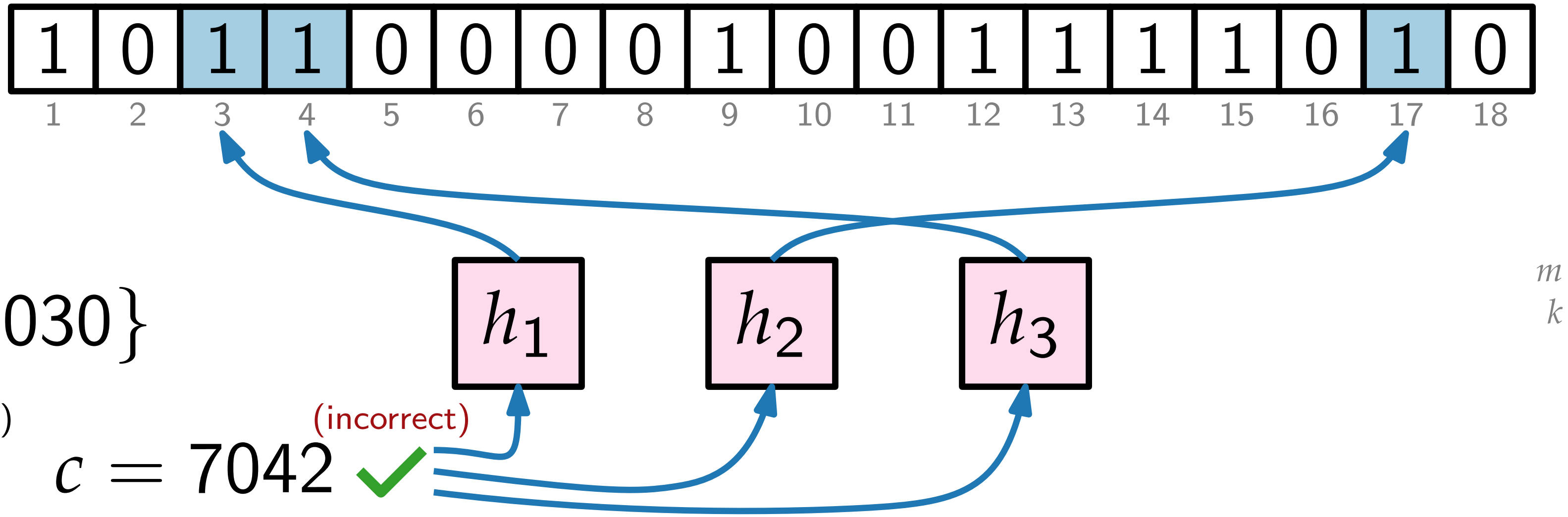
A Bloom filter is a **bit array** of m bits & a set of k different **hash functions** $h_1, \ldots, h_k$. Each hash function h_i generates a uniform random distribution in the range $\{1, \ldots, m\}$.

Initially the array contains only 0s. Such a Bloom filter represents the empty set.

For a set S of numbers, we insert each $s \in S$ to the Bloom filter by setting all bits at the positions $h_1(s)$, $h_2(s)$, …, $h_k(s)$ to 1.

Containment check of a number a: check the bits at positions $h_1(a), h_2(a), \ldots, h_k(a)$:

- if there is a 0, a is for sure not in S.
- if there are only 1s, a may be in S (it is in S with a small error probability).



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	0	1	1	0	0	0	0	1	0	0	1	1	1	1	0	1	0

$m = 18$
$k = 3$

$S = \{2345, 8234, 12492, 34030\}$

$a = 2346$ (correct) ✗ $b = 8234$ (correct) ✓ $c = 7042$ (incorrect) ✓

Error Probability of Bloom Filters

We start with an empty array and insert a number s.

Error Probability of Bloom Filters

We start with an empty array and insert a number s.

- The probability that a specific bit is kept as 0 by h_i is $1 - \frac{1}{m}$.

Error Probability of Bloom Filters

We start with an empty array and insert a number s.

- The probability that a specific bit is kept as 0 by h_i is $1 - \frac{1}{m}$.
- The probability that a specific bit is kept as 0 by all k hash functions is $\left(1 - \frac{1}{m}\right)^k$.

Error Probability of Bloom Filters

We start with an empty array and insert a number s.

- The probability that a specific bit is kept as 0 by h_i is $1 - \frac{1}{m}$.
- The probability that a specific bit is kept as 0 by all k hash functions is $\left(1 - \frac{1}{m}\right)^k$.
- For large m, we have $\left(1 - \frac{1}{m}\right)^k = \left(\left(1 - \frac{1}{m}\right)^m\right)^{k/m} \approx e^{-k/m}$ since $\lim_{m\to\infty} \left(1 - \frac{1}{m}\right)^m = \frac{1}{e}$.

Error Probability of Bloom Filters

We start with an empty array and insert a number s.

- The probability that a specific bit is kept as 0 by h_i is $1 - \frac{1}{m}$.
- The probability that a specific bit is kept as 0 by all k hash functions is $\left(1 - \frac{1}{m}\right)^k$.
- For large m, we have $\left(1 - \frac{1}{m}\right)^k = \left(\left(1 - \frac{1}{m}\right)^m\right)^{k/m} \approx e^{-k/m}$ since $\lim_{m\to\infty} \left(1 - \frac{1}{m}\right)^m = \frac{1}{e}$.

After having inserted all n numbers, the probability that a specific bit is kept as 0 is $\left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}$.

Error Probability of Bloom Filters

We start with an empty array and insert a number s.

- The probability that a specific bit is kept as 0 by h_i is $1 - \frac{1}{m}$.
- The probability that a specific bit is kept as 0 by all k hash functions is $\left(1 - \frac{1}{m}\right)^k$.
- For large m, we have $\left(1 - \frac{1}{m}\right)^k = \left(\left(1 - \frac{1}{m}\right)^m\right)^{k/m} \approx e^{-k/m}$ since $\lim_{m \to \infty} \left(1 - \frac{1}{m}\right)^m = \frac{1}{e}$.

After having inserted all n numbers, the probability that a specific bit is kept as 0 is $\left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}$.

Now, check containment for a number $a \notin S$.

Error Probability of Bloom Filters

We start with an empty array and insert a number s.

- The probability that a specific bit is kept as 0 by h_i is $1 - \frac{1}{m}$.
- The probability that a specific bit is kept as 0 by all k hash functions is $\left(1 - \frac{1}{m}\right)^k$.
- For large m, we have $\left(1 - \frac{1}{m}\right)^k = \left(\left(1 - \frac{1}{m}\right)^m\right)^{k/m} \approx e^{-k/m}$ since $\lim_{m\to\infty} \left(1 - \frac{1}{m}\right)^m = \frac{1}{e}$.

After having inserted all n numbers, the probability that a specific bit is kept as 0 is $\left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}$.

Now, check containment for a number $a \notin S$.

The probabilities that all bits at positions $h_1(a), \ldots, h_k(a)$ are set to 1 are not independent. However, one can still show that the error probability ε for a false positive is relatively close to $\varepsilon \approx \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k$.

Parameters of Bloom Filters

$$\varepsilon \approx \left(1 - e^{-kn/m}\right)^k$$

So what number k of hash functions should we use?

Parameters of Bloom Filters

$$\varepsilon \approx \left(1 - e^{-kn/m}\right)^k$$

So what number k of hash functions should we use?

The error probability ε is minimized if $k \approx \frac{m}{n} \ln 2$.

Parameters of Bloom Filters

$\varepsilon \approx \left(1 - e^{-kn/m}\right)^k$

So what number k of hash functions should we use?

The error probability ε is minimized if $k \approx \frac{m}{n} \ln 2$.

If we only use the optimal k, the error probibilty $\varepsilon \approx \left(\frac{1}{2}\right)^{m \ln 2 / n}$

Parameters of Bloom Filters

$$\varepsilon \approx \left(1 - e^{-kn/m}\right)^k$$

So what number k of hash functions should we use?

The error probability ε is minimized if $k \approx \frac{m}{n} \ln 2$.

If we only use the optimal k, the error probibilty $\varepsilon \approx \left(\frac{1}{2}\right)^{m \ln 2/n}$

Thus, the optimal number of bits per element in our set is $\frac{m}{n} = -\frac{\log_2 \varepsilon}{\ln 2} \approx -1.44 \log_2 \varepsilon$.

Parameters of Bloom Filters

$$\varepsilon \approx \left(1 - e^{-kn/m}\right)^k$$

So what number k of hash functions should we use?

The error probability ε is minimized if $k \approx \frac{m}{n} \ln 2$.

If we only use the optimal k, the error probibilty $\varepsilon \approx \left(\frac{1}{2}\right)^{m \ln 2/n}$

Thus, the optimal number of bits per element in our set is $\frac{m}{n} = -\frac{\log_2 \varepsilon}{\ln 2} \approx -1.44 \log_2 \varepsilon$.

So, the number of bits in our array depends on the desired error probabilty,

Parameters of Bloom Filters

$$\varepsilon \approx \left(1 - e^{-kn/m}\right)^{k}$$

So what number k of hash functions should we use?

The error probability ε is minimized if $k \approx \frac{m}{n} \ln 2$.

If we only use the optimal k, the error probibilty $\varepsilon \approx \left(\frac{1}{2}\right)^{m \ln 2/n}$

Thus, the optimal number of bits per element in our set is $\frac{m}{n} = -\frac{\log_2 \varepsilon}{\ln 2} \approx -1.44 \log_2 \varepsilon$.

So, the number of bits in our array depends on the desired error probabilty,

error probabiltiy ε	
bits per element $\frac{m}{n}$	
# hash functions k	

Parameters of Bloom Filters

$$\varepsilon \approx \left(1 - e^{-kn/m}\right)^k$$

So what number k of hash functions should we use?

The error probability ε is minimized if $k \approx \frac{m}{n} \ln 2$.

If we only use the optimal k, the error probibilty $\varepsilon \approx \left(\frac{1}{2}\right)^{m \ln 2 / n}$

Thus, the optimal number of bits per element in our set is $\frac{m}{n} = -\frac{\log_2 \varepsilon}{\ln 2} \approx -1.44 \log_2 \varepsilon$.

So, the number of bits in our array depends on the desired error probabilty,

10 %

error probabiltiy ε	0.1
bits per element $\frac{m}{n}$	12
# hash functions k	4

Parameters of Bloom Filters

$$\varepsilon \approx \left(1 - e^{-kn/m}\right)^k$$

So what number k of hash functions should we use?

The error probability ε is minimized if $k \approx \frac{m}{n} \ln 2$.

If we only use the optimal k, the error probibilty $\varepsilon \approx \left(\frac{1}{2}\right)^{m \ln 2/n}$

Thus, the optimal number of bits per element in our set is $\frac{m}{n} = -\frac{\log_2 \varepsilon}{\ln 2} \approx -1.44 \log_2 \varepsilon$.

So, the number of bits in our array depends on the desired error probabilty,

1 %

error probabiltiy ε	0.1	0.01
bits per element $\frac{m}{n}$	12	23
# hash functions k	4	7

Parameters of Bloom Filters

$$\varepsilon \approx \left(1 - e^{-kn/m}\right)^k$$

So what number k of hash functions should we use?

The error probability ε is minimized if $k \approx \frac{m}{n} \ln 2$.

If we only use the optimal k, the error probibilty $\varepsilon \approx \left(\frac{1}{2}\right)^{m \ln 2 / n}$

Thus, the optimal number of bits per element in our set is $\frac{m}{n} = -\frac{\log_2 \varepsilon}{\ln 2} \approx -1.44 \log_2 \varepsilon$.

So, the number of bits in our array depends on the desired error probabilty,

1 ‰

error probabiltiy ε	0.1	0.01	0.001
bits per element $\frac{m}{n}$	12	23	34
$\#$ hash functions k	4	7	10

Parameters of Bloom Filters

$$\varepsilon \approx \left(1 - e^{-kn/m}\right)^k$$

So what number k of hash functions should we use?

The error probability ε is minimized if $k \approx \frac{m}{n} \ln 2$.

If we only use the optimal k, the error probibilty $\varepsilon \approx \left(\frac{1}{2}\right)^{m \ln 2/n}$

Thus, the optimal number of bits per element in our set is $\frac{m}{n} = -\frac{\log_2 \varepsilon}{\ln 2} \approx -1.44 \log_2 \varepsilon$.

So, the number of bits in our array depends on the desired error probabilty,

1 in a million (10^6)

error probabiltiy ε	0.1	0.01	0.001	0.000001
bits per element $\frac{m}{n}$	12	23	34	67
# hash functions k	4	7	10	20

Parameters of Bloom Filters

$$\varepsilon \approx \left(1 - e^{-kn/m}\right)^k$$

So what number k of hash functions should we use?

The error probability ε is minimized if $k \approx \frac{m}{n} \ln 2$.

If we only use the optimal k, the error probibilty $\varepsilon \approx \left(\frac{1}{2}\right)^{m \ln 2/n}$

Thus, the optimal number of bits per element in our set is $\frac{m}{n} = -\frac{\log_2 \varepsilon}{\ln 2} \approx -1.44 \log_2 \varepsilon$.

So, the number of bits in our array depends on the desired error probabilty,

1 in a billion (10^9)

error probabiltiy ε	0.1	0.01	0.001	0.000001	0.000000001
bits per element $\frac{m}{n}$	12	23	34	67	100
# hash functions k	4	7	10	20	30

Parameters of Bloom Filters

$$\varepsilon \approx \left(1 - e^{-kn/m}\right)^k$$

So what number k of hash functions should we use?

The error probability ε is minimized if $k \approx \frac{m}{n} \ln 2$.

If we only use the optimal k, the error probibilty $\varepsilon \approx \left(\frac{1}{2}\right)^{m \ln 2 / n}$

Thus, the optimal number of bits per element in our set is $\frac{m}{n} = -\frac{\log_2 \varepsilon}{\ln 2} \approx -1.44 \log_2 \varepsilon$.

So, the number of bits in our array depends on the desired error probabilty,

1 in a trillion (10^{12}) ↙

error probabiltiy ε	0.1	0.01	0.001	0.000001	0.000000001	0.000000000001
bits per element $\frac{m}{n}$	12	23	34	67	100	133
# hash functions k	4	7	10	20	30	40

Parameters of Bloom Filters

$$\varepsilon \approx \left(1 - e^{-kn/m}\right)^k$$

So what number k of hash functions should we use?

The error probability ε is minimized if $k \approx \frac{m}{n} \ln 2$.

If we only use the optimal k, the error probibilty $\varepsilon \approx \left(\frac{1}{2}\right)^{m \ln 2/n}$

Thus, the optimal number of bits per element in our set is $\frac{m}{n} = -\frac{\log_2 \varepsilon}{\ln 2} \approx -1.44 \log_2 \varepsilon$.

So, the number of bits in our array depends on the desired error probabilty,

error probabiltiy ε	0.1	0.01	0.001	0.000001	0.000000001	0.000000000001
bits per element $\frac{m}{n}$	12	23	34	67	100	133
# hash functions k	4	7	10	20	30	40

. . . but not on the lengths of the numbers.
(We could check for whole documents whether they are there or not.)

Discussion

- (Randomized) skip list provide a simpler alternative to balanced binary search trees with (in expectation) the same asymptotic time complexities for the basic operations.

Discussion

- (Randomized) skip list provide a simpler alternative to balanced binary search trees with (in expectation) the same asymptotic time complexities for the basic operations.
- However, the constant factors may differ and if running time and space consumption are the most important factors, binary search trees might still be the better choice.

Discussion

- (Randomized) skip list provide a simpler alternative to balanced binary search trees with (in expectation) the same asymptotic time complexities for the basic operations.
- However, the constant factors may differ and if running time and space consumption are the most important factors, binary search trees might still be the better choice.
- Bloom filters provide a very space-efficient and time-efficient tool to handle requests on large data sets. They should be applied where the disadvantages (no removals, potentially wrong output) can be tolerated.

Discussion

- (Randomized) skip list provide a simpler alternative to balanced binary search trees with (in expectation) the same asymptotic time complexities for the basic operations.
- However, the constant factors may differ and if running time and space consumption are the most important factors, binary search trees might still be the better choice.
- Bloom filters provide a very space-efficient and time-efficient tool to handle requests on large data sets. They should be applied where the disadvantages (no removals, potentially wrong output) can be tolerated.
- There are some refinements of classical Bloom filters to overcome some disadvantages.

Discussion

- (Randomized) skip list provide a simpler alternative to balanced binary search trees with (in expectation) the same asymptotic time complexities for the basic operations.
- However, the constant factors may differ and if running time and space consumption are the most important factors, binary search trees might still be the better choice.
- Bloom filters provide a very space-efficient and time-efficient tool to handle requests on large data sets. They should be applied where the disadvantages (no removals, potentially wrong output) can be tolerated.
- There are some refinements of classical Bloom filters to overcome some disadvantages.
- Bloom filters are used for
  - Internet search engines
  - caching objects in Internet applications (is an image or a digest in the cache?)
  - databases (Google Bigtable, Apache HBase, Apache Cassandra, PostgreSQL)
  - web browsers (Google Chrome used one to identify malicious URLs)
  - crypto currencies (finding logs in Ethereum)

Literature

Skip lists:

- [Pugh '90] William W. Pugh, "Skip Lists: A Probabilistic Alternative to Balanced Trees" in *Communications of the ACM*, 33(6):668–676, 1990
- Sabine Storandt's lecture script "Randomized Algorithms" (2016–2017)

Bloom filters:

- [Bloom '70] Burton H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors" in *Communications of the ACM*, 13(7):422–426, 1970
- [Mitzenmacher, Upfal '05] Michael Mitzenmacher and Eli Upfal, "Probability and Computing: Randomized Algorithms and Probabilistic Analysis", Cambridge University Press, 2005
- [Bose et al. '08] Prosenjit Bose, Hua Guo, Evangelos Kranakis, Anil Maheshwari, Pat Morin, Jason Morrison, Michiel H. M. Smid, and Yihui Tang, "On the false-positive rate of Bloom filters" in *Information Processing Letters*, 108(4):210–213, 2008