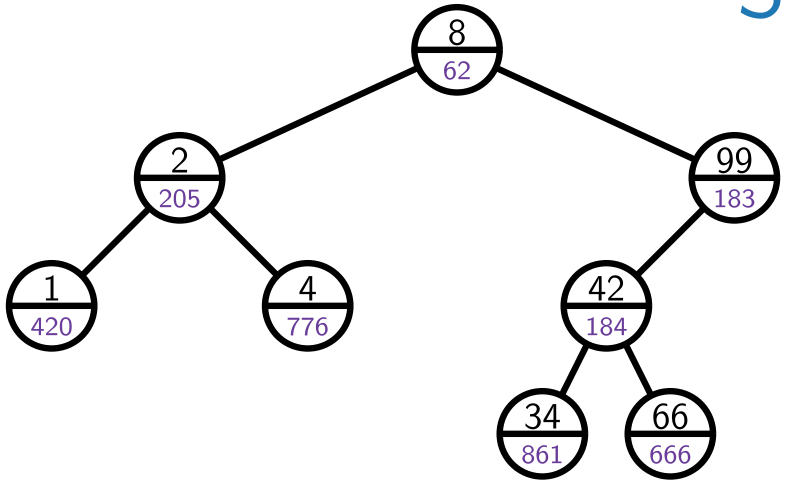
Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

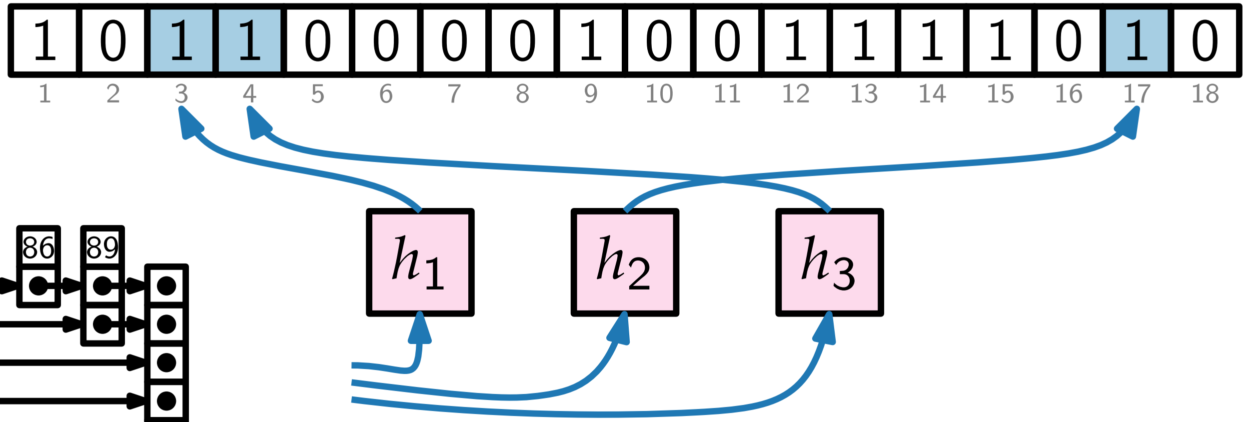
# Advanced Algorithms

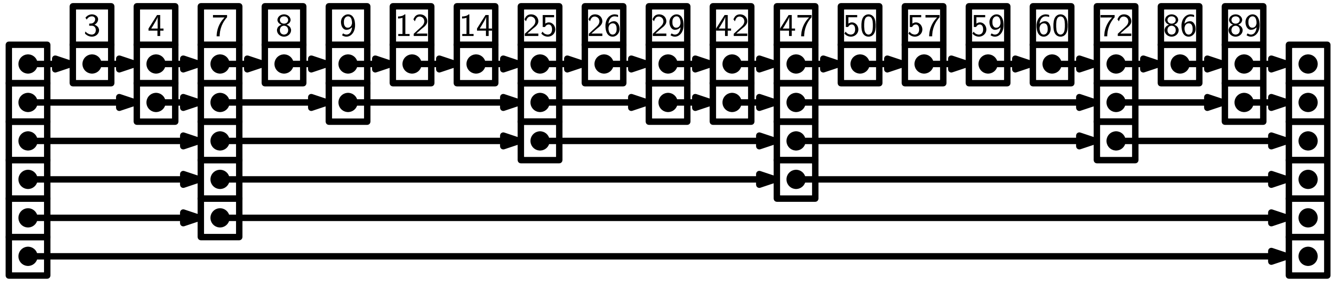
## Randomized and Probabilistic Data Structures

### Skip Lists, Treaps & Bloom Filters




Johannes Zink · WS23/24



# 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**.
  - 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**.
  - Hashing when choosing a random hash function
  - 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**.
  - Hashing when choosing a random hash function
  - Randomized skip lists (in this lecture!)
  - Treaps (in this lecture!) and randomized binary search trees

# 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**.
  - Hashing when choosing a random hash function
  - Randomized skip lists (in this lecture!)
  - Treaps (in this lecture!) and randomized binary search trees
- A data structure that answers correctly according to some probability 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**.
  - Hashing when choosing a random hash function
  - Randomized skip lists (in this lecture!)
  - Treaps (in this lecture!) and randomized binary search trees
- A data structure that answers correctly according to some probability distribution is a **probabilistic data structure**.
  - Bloom filters (in this lecture!)

# Data Structures and Randomization

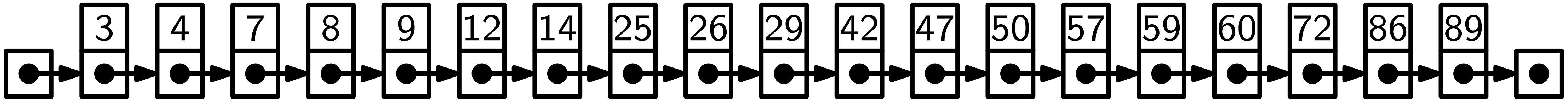
- We have seen that (in expectation) some **randomized algorithms** beat all deterministic approaches in terms of running time.
- Moreover, randomized approaches are often simpler/more elegant.
- Also data structures may use randomization for these reasons.
- A data structure that uses randomization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

# Sorted Linked Lists

What time is needed to search a key in a sorted linked list with $n$ entries?

# Sorted Linked Lists

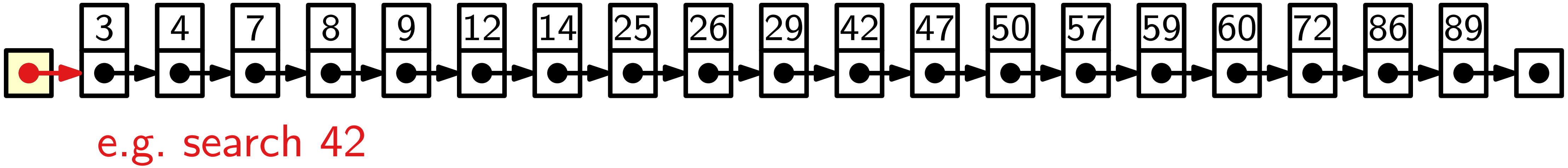
What time is needed to search a key in a sorted linked list with $n$ entries?




e.g. search 42

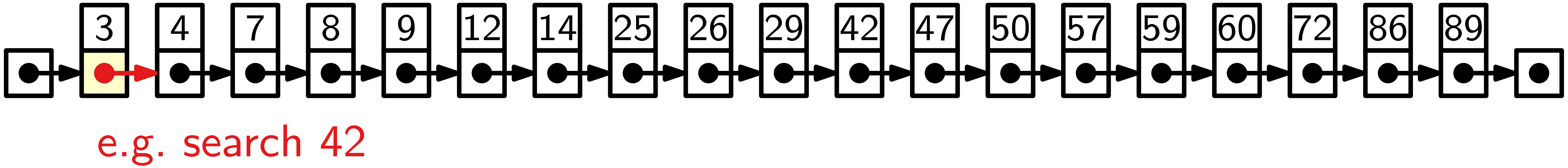
# Sorted Linked Lists

What time is needed to search a key in a sorted linked list with $n$ entries?



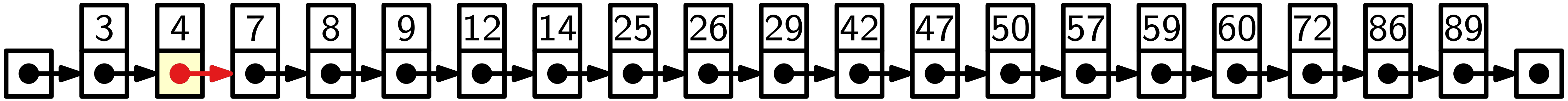
# Sorted Linked Lists

What time is needed to search a key in a sorted linked list with $n$ entries?



# Sorted Linked Lists

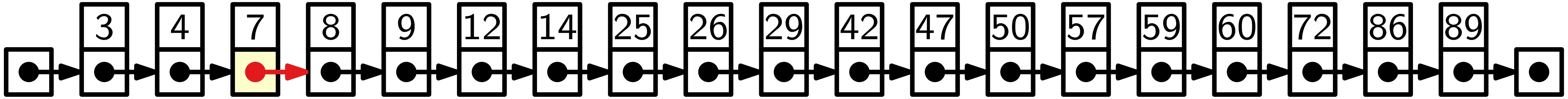
What time is needed to search a key in a sorted linked list with $n$ entries?




e.g. search 42

# Sorted Linked Lists

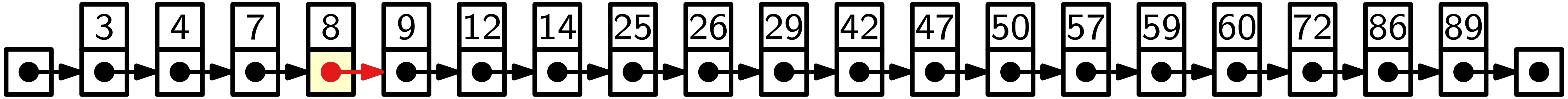
What time is needed to search a key in a sorted linked list with $n$ entries?




e.g. search 42

# Sorted Linked Lists

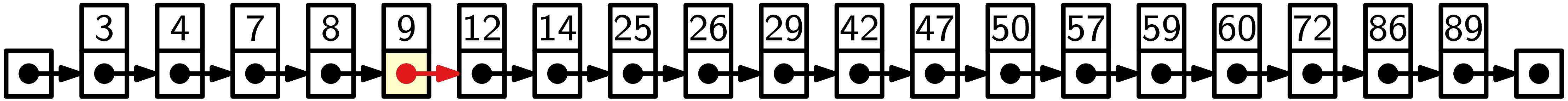
What time is needed to search a key in a sorted linked list with $n$ entries?




e.g. search 42

# Sorted Linked Lists

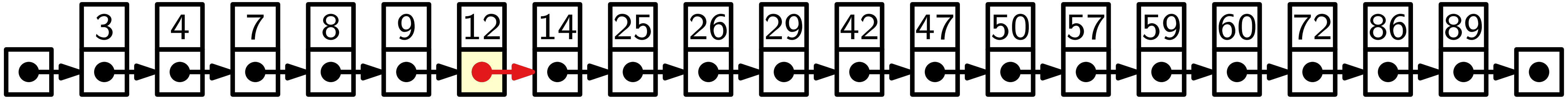
What time is needed to search a key in a sorted linked list with $n$ entries?




e.g. search 42

# Sorted Linked Lists

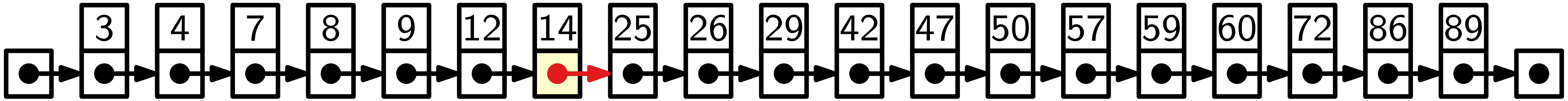
What time is needed to search a key in a sorted linked list with $n$ entries?




e.g. search 42

# Sorted Linked Lists

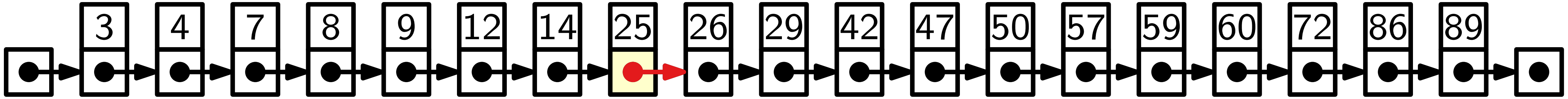
What time is needed to search a key in a sorted linked list with $n$ entries?




e.g. search 42

# Sorted Linked Lists

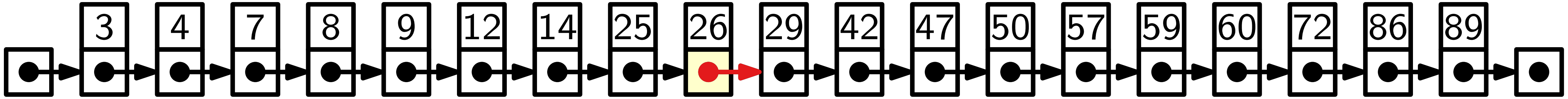
What time is needed to search a key in a sorted linked list with $n$ entries?




e.g. search 42

# Sorted Linked Lists

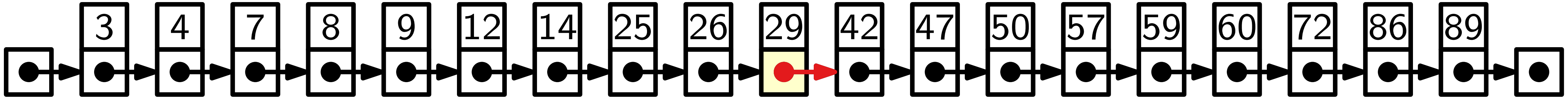
What time is needed to search a key in a sorted linked list with $n$ entries?




e.g. search 42

# Sorted Linked Lists

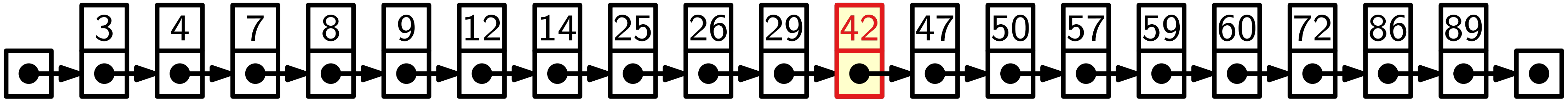
What time is needed to search a key in a sorted linked list with $n$ entries?




e.g. search 42

# Sorted Linked Lists

What time is needed to search a key in a sorted linked list with $n$ entries?

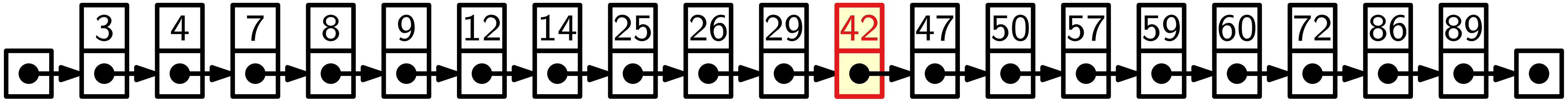



e.g. search 42

# Sorted Linked Lists

What time is needed to search a key in a sorted linked list with $n$ entries?

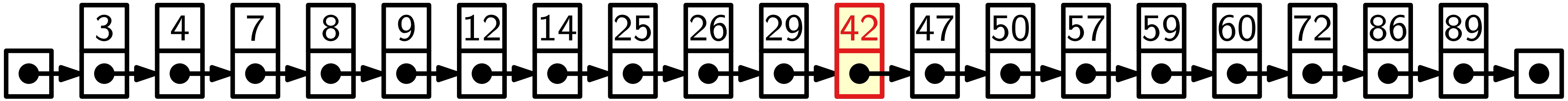
$\Theta(n)$




e.g. search 42

# Sorted Linked Lists

What time is needed to search a key in a sorted linked list with $n$ entries?



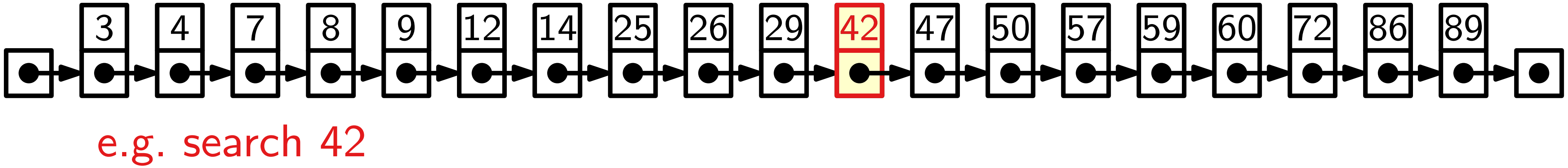

$\Theta(n)$

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.

# Sorted Linked Lists

What time is needed to search a key in a sorted linked list with $n$ entries?



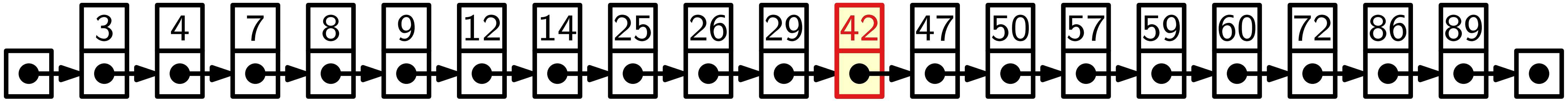

$\Theta(n)$

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, ... keys.

# Deterministic Skip Lists

What time is needed to search a key in a sorted linked list with $n$ entries?

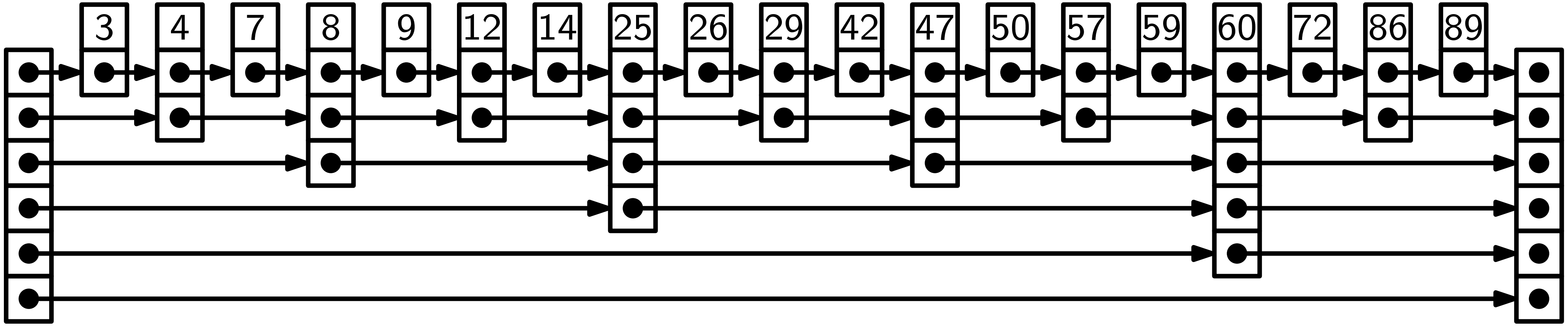



$\Theta(n)$

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, . . . keys.

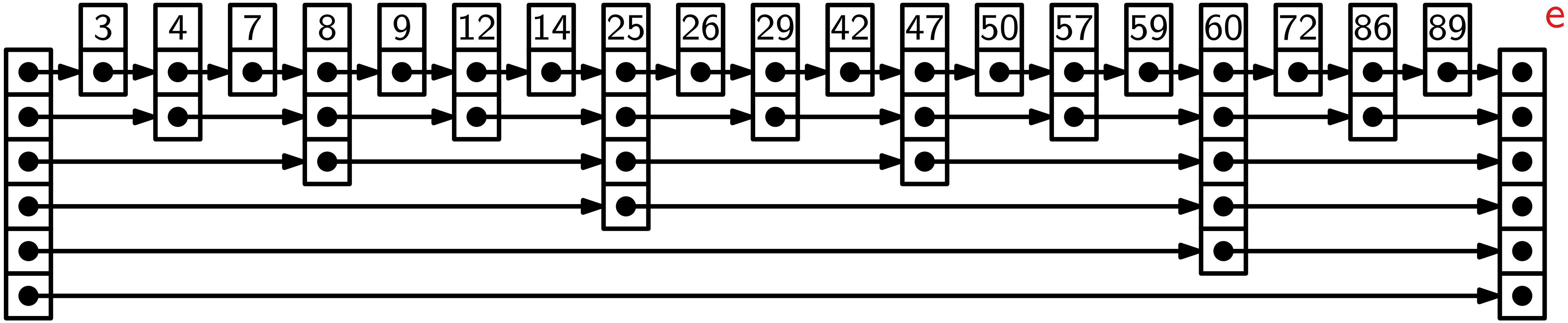



**deterministic skip list**

# Deterministic Skip Lists

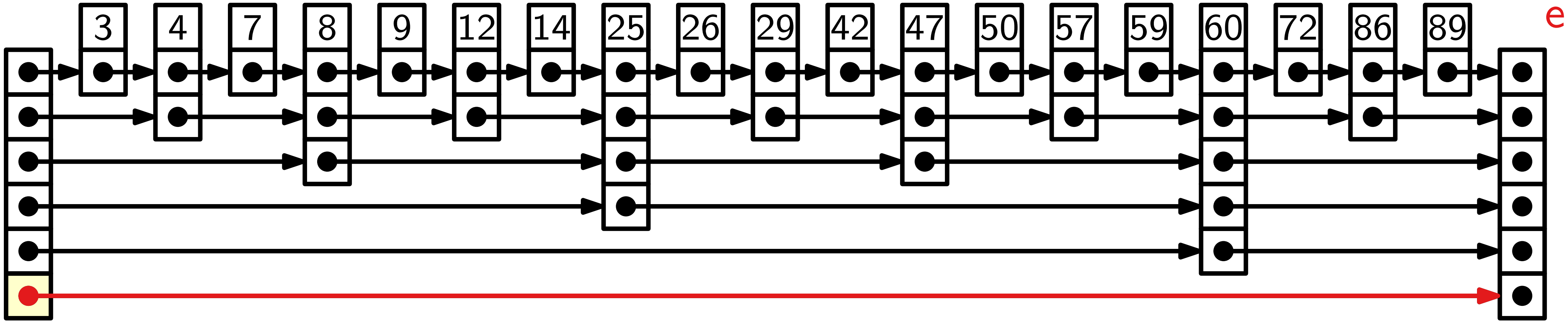
What time is needed to search a key in a deterministic skip list with $n$ entries?

e.g. search 42



# Deterministic Skip Lists

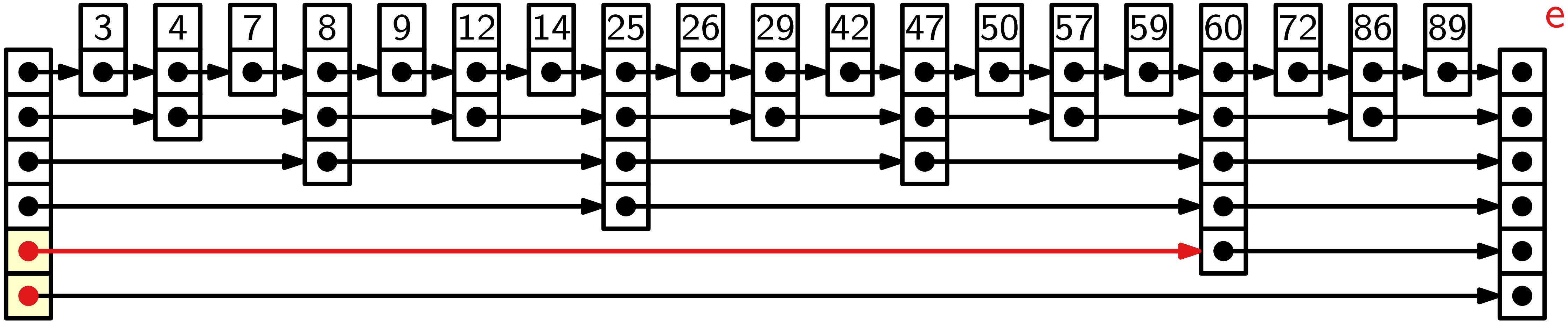
What time is needed to search a key in a deterministic skip list with $n$ entries?




e.g. search 42

# Deterministic Skip Lists

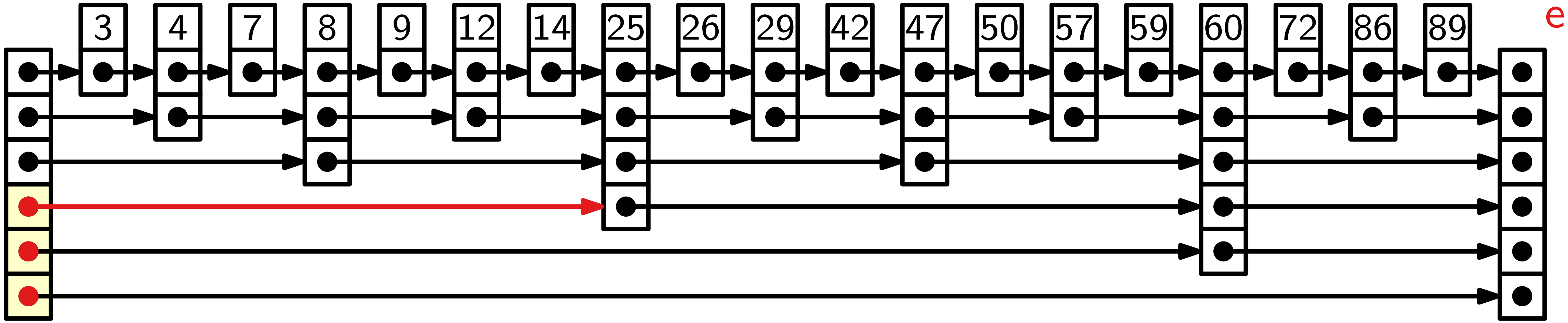
What time is needed to search a key in a deterministic skip list with $n$ entries?




e.g. search 42

# Deterministic Skip Lists

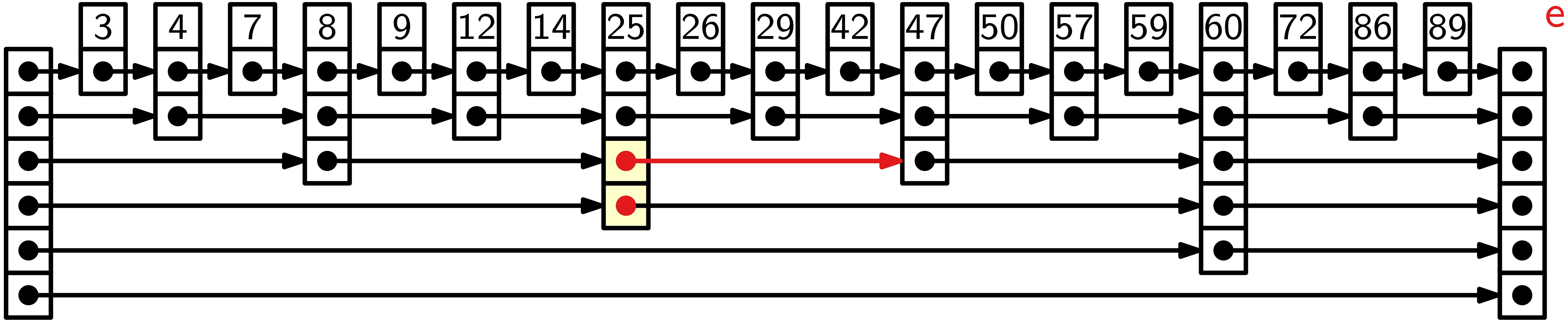
What time is needed to search a key in a deterministic skip list with $n$ entries?




e.g. search 42

# Deterministic Skip Lists

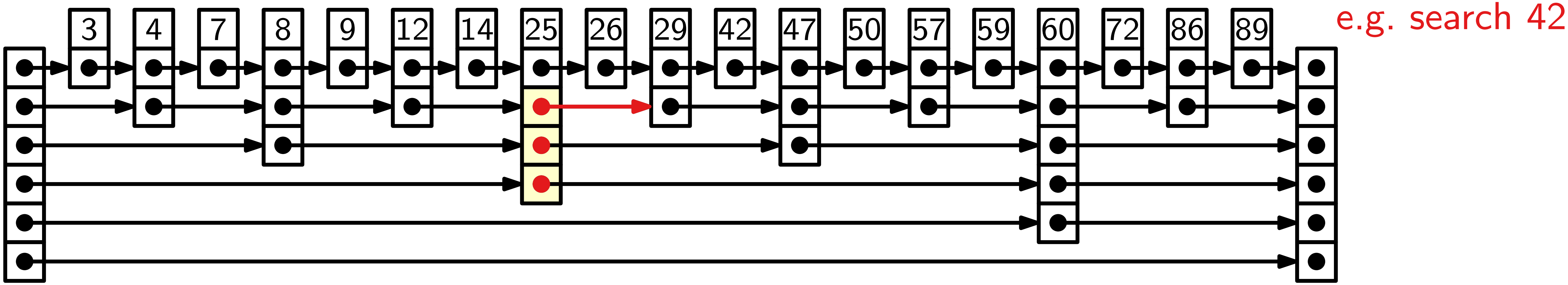
What time is needed to search a key in a deterministic skip list with $n$ entries?




e.g. search 42

# Deterministic Skip Lists

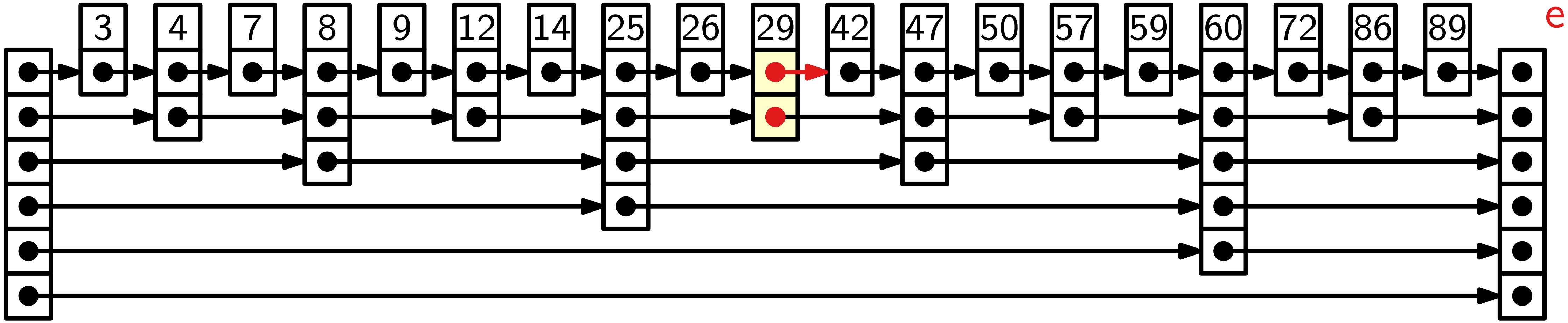
What time is needed to search a key in a deterministic skip list with $n$ entries?



# Deterministic Skip Lists

What time is needed to search a key in a deterministic skip list with $n$ entries?

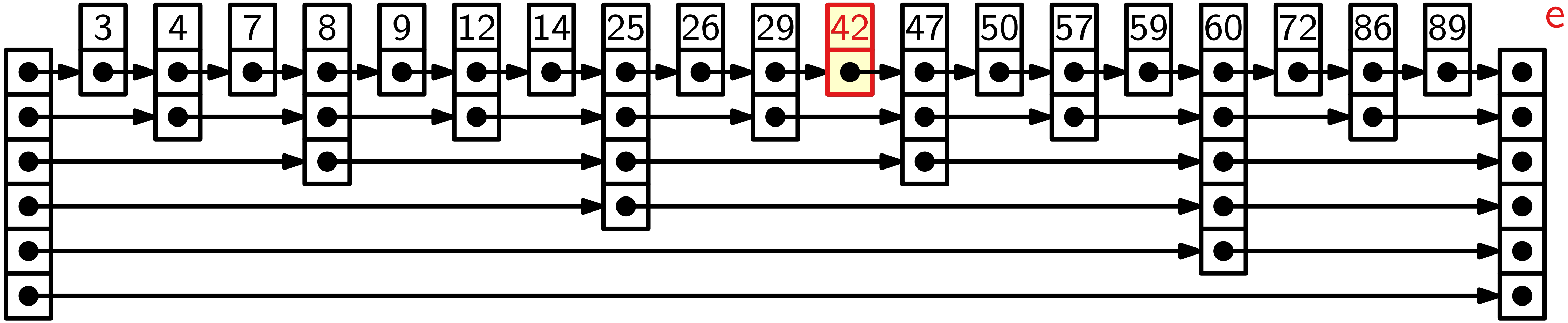
e.g. search 42



# Deterministic Skip Lists

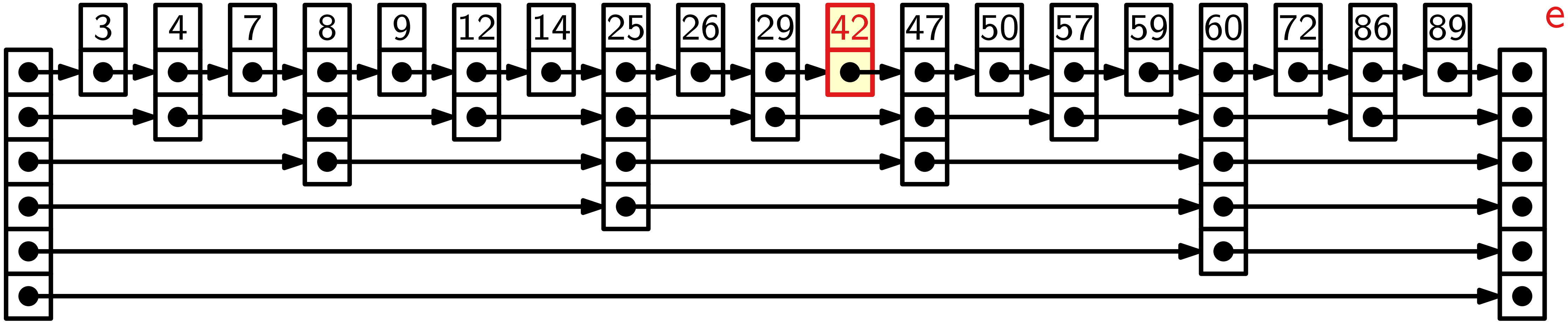
What time is needed to search a key in a deterministic skip list with $n$ entries?

e.g. search 42



# Deterministic Skip Lists

What time is needed to search a key in a deterministic skip list with $n$ entries?



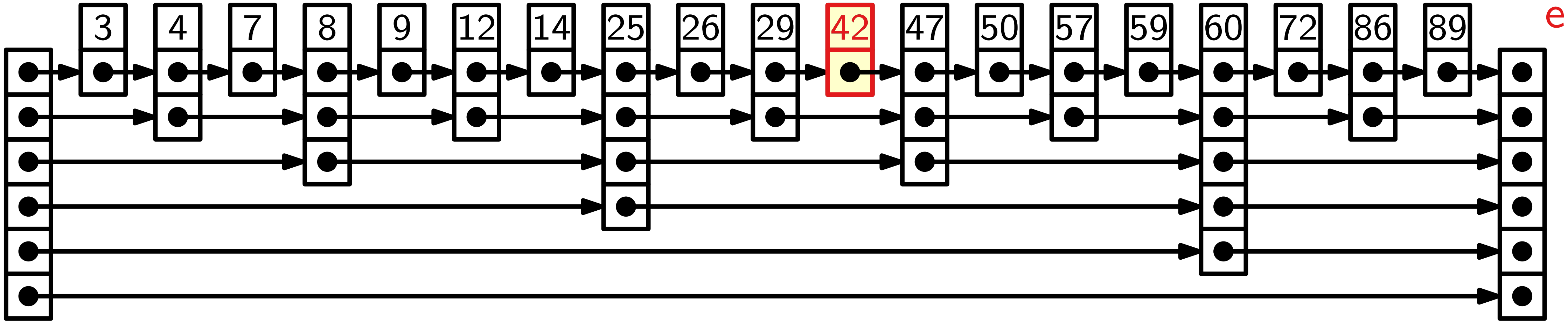

e.g. search 42

$\Theta(\log n)$

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

# Deterministic Skip Lists

What time is needed to search a key in a deterministic skip list with $n$ entries?




e.g. search 42

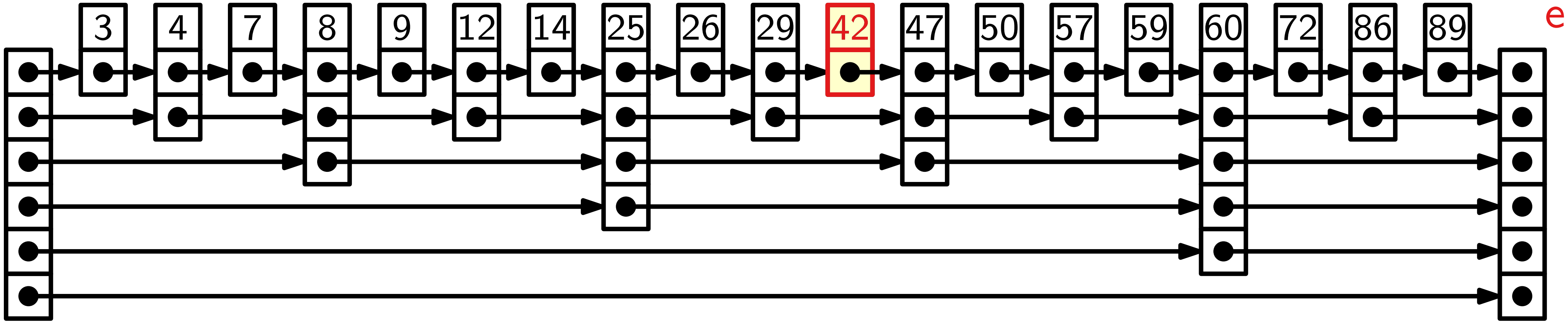
$\Theta(\log n)$

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

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

# Deterministic Skip Lists

What time is needed to search a key in a deterministic skip list with $n$ entries?




e.g. search 42

$\Theta(\log n)$

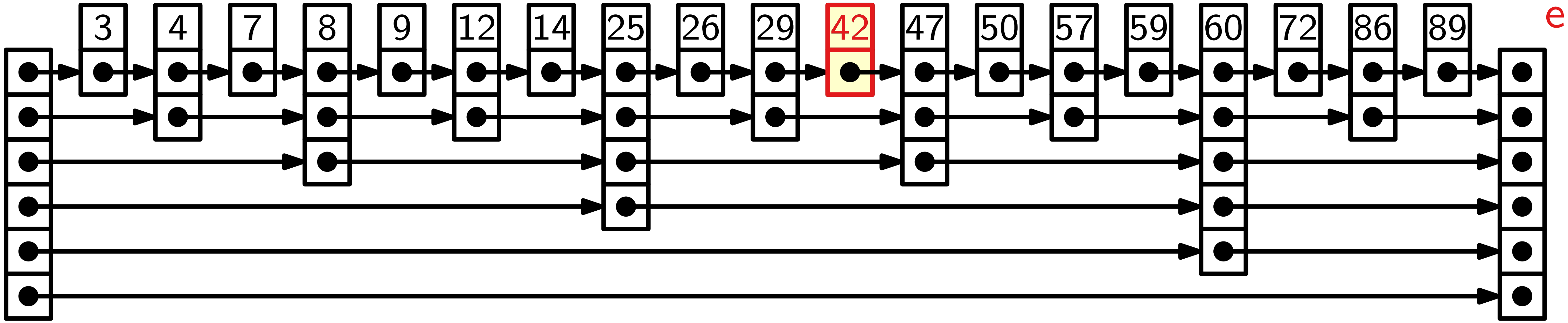
(We check only constantly many entries per level and there are $O(\log n)$ levels.)

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

$\Theta(n)$ (We need to swap multiple keys and re-build some parts)

# Deterministic Skip Lists

What time is needed to search a key in a deterministic skip list with $n$ entries?




e.g. search 42

$\Theta(\log n)$

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

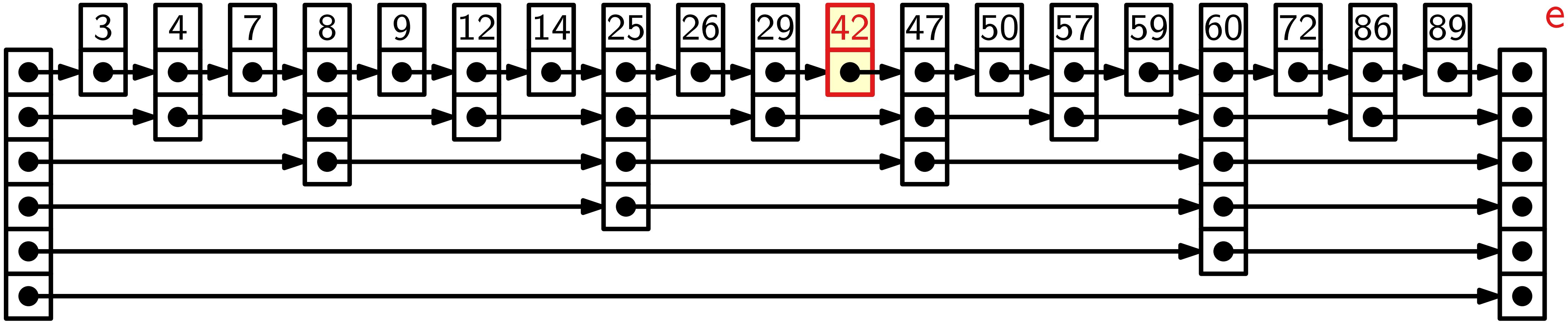
What time is needed to insert/delete a key in a deterministic skip list?

$\Theta(n)$ (We need to swap multiple keys and re-build some parts)

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 a key in a deterministic skip list with $n$ entries?




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 a key in a deterministic skip list?

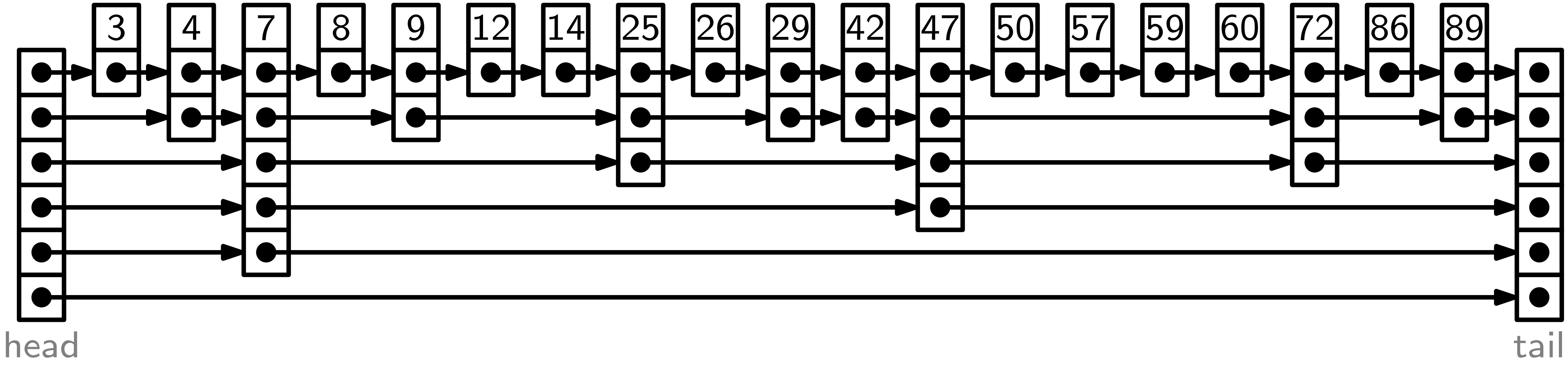
$\Theta(n)$ (We need to swap multiple keys and re-build some parts)

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

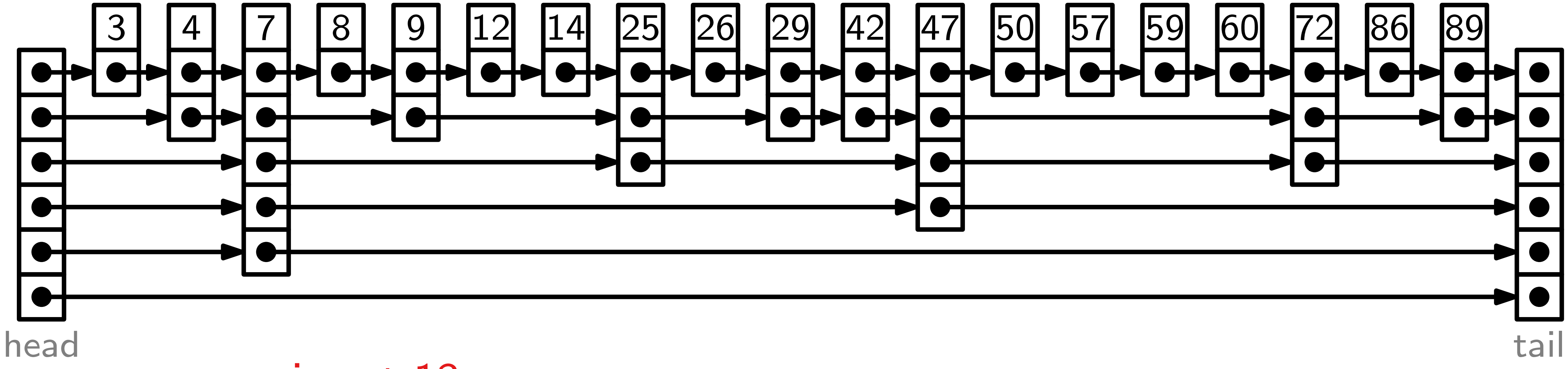
For a new entry, flip a coin until it shows HEADS. The number of flips will be its height.




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

# Randomized Skip Lists

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

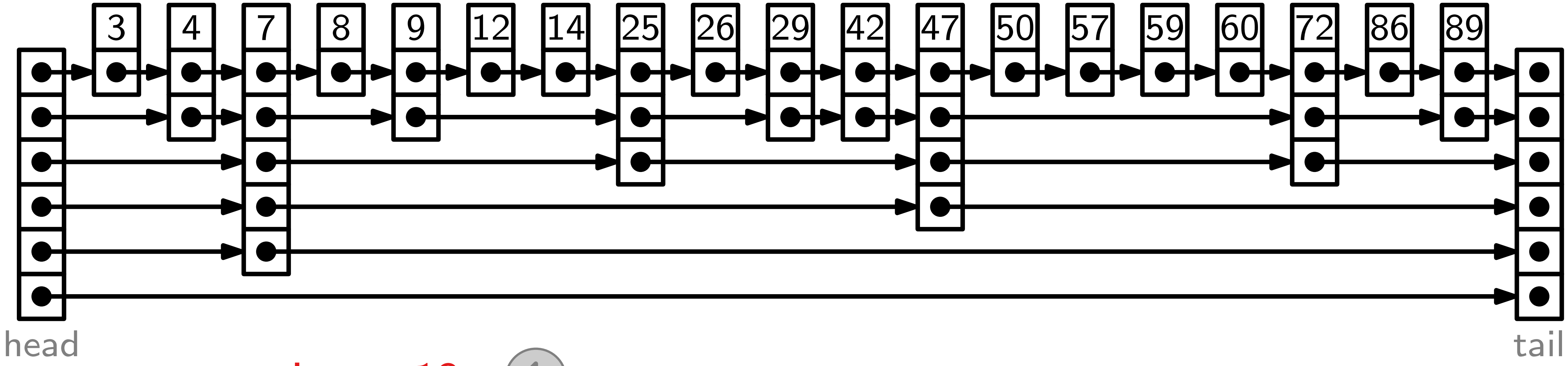



(We let the head/tail of the list have a height of $\lfloor \log_2 n \rfloor + 1$.)

e.g. insert 13

# Randomized Skip Lists

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

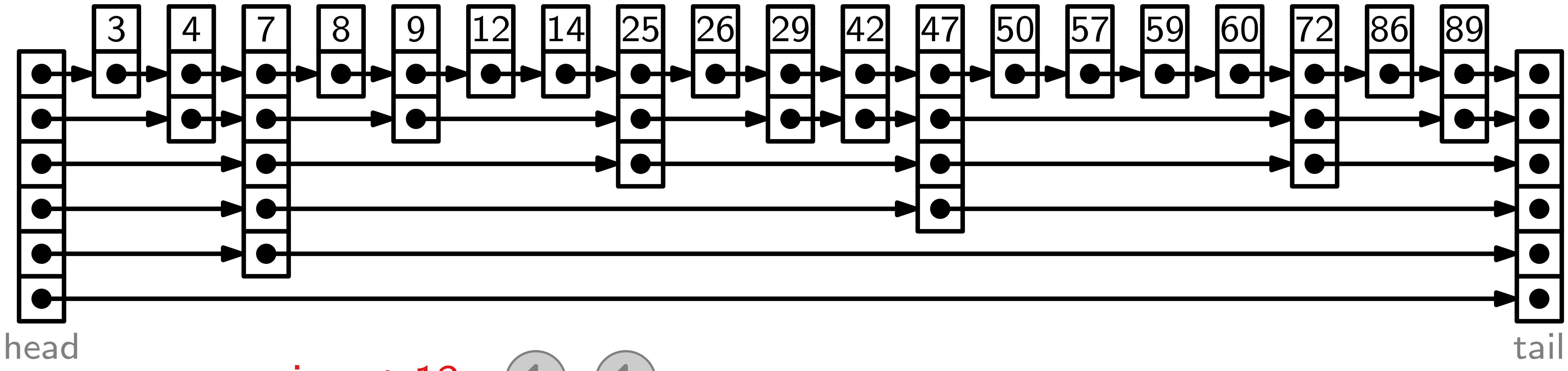



(We let the head/tail of the list have a height of $\lfloor \log_2 n \rfloor + 1$.)

e.g. insert 13

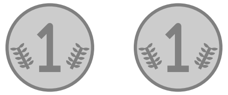
# Randomized Skip Lists

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



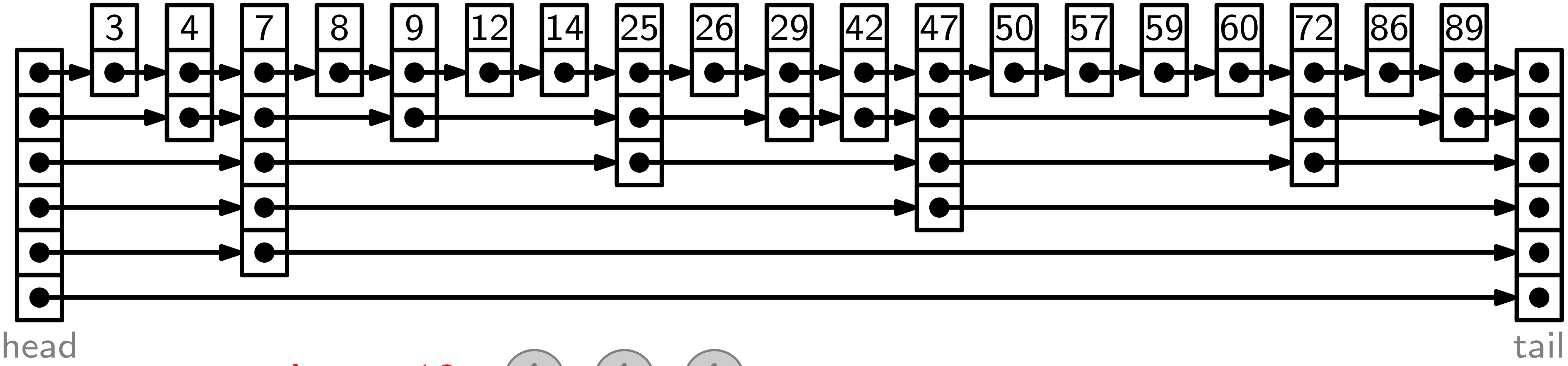

(We let the head/tail of the list have a height of $\lfloor \log_2 n \rfloor + 1$.)

e.g. insert 13



# Randomized Skip Lists

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

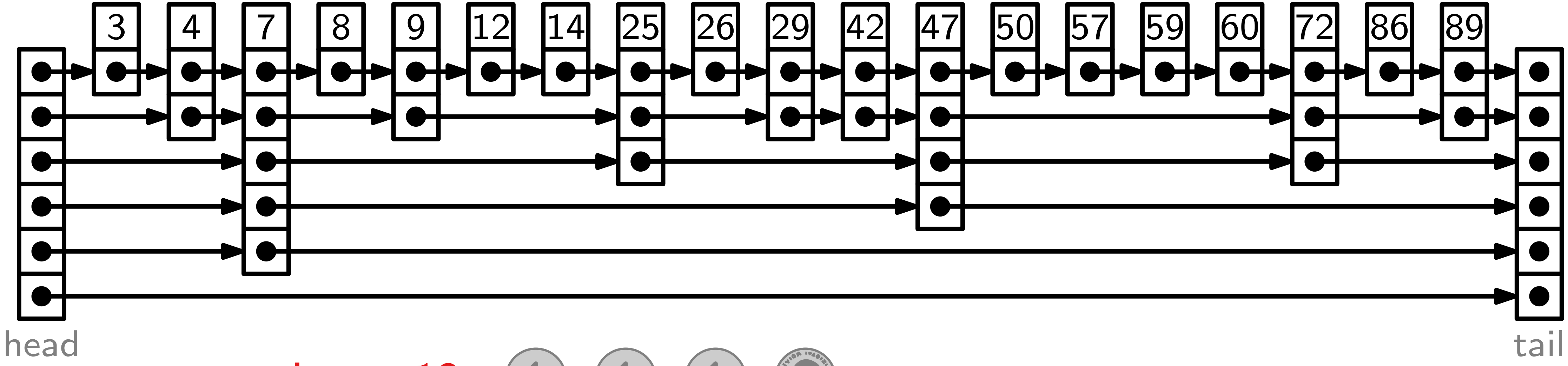



(We let the head/tail of the list have a height of $\lfloor \log_2 n \rfloor + 1$.)

e.g. insert 13

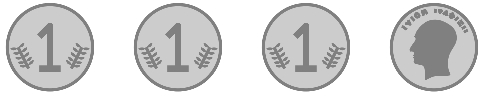
# Randomized Skip Lists

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



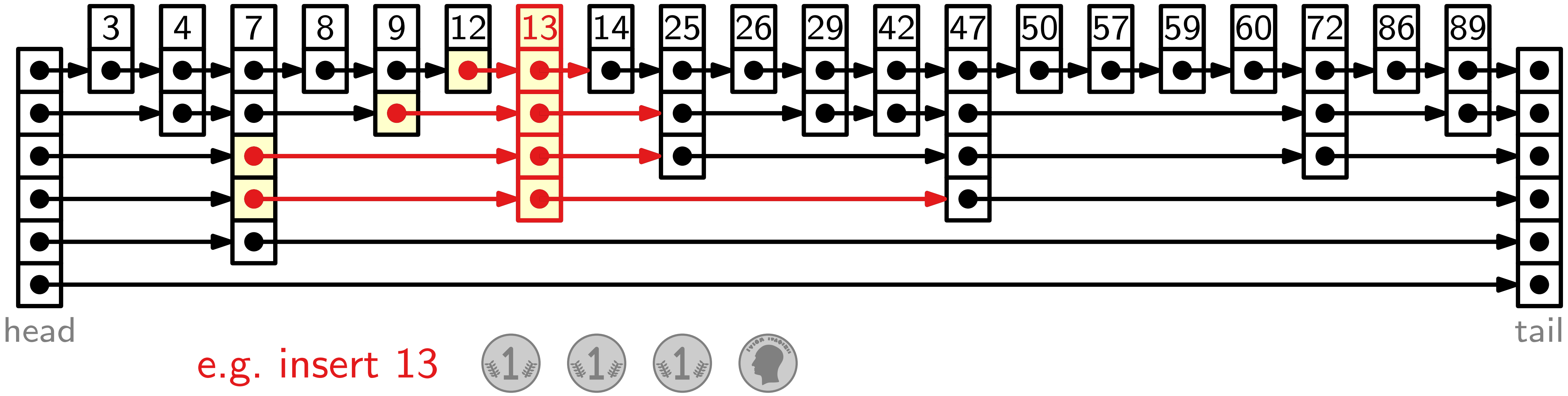

(We let the head/tail of the list have a height of $\lfloor \log_2 n \rfloor + 1$.)

e.g. insert 13



# Randomized Skip Lists

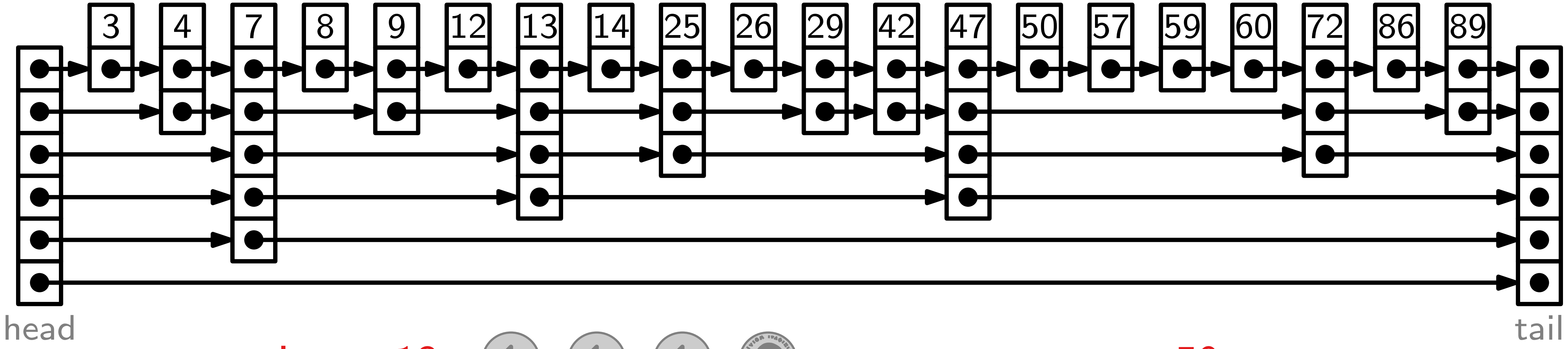
For a new entry, flip a coin until it shows HEADS. The number of flips will be its height.




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

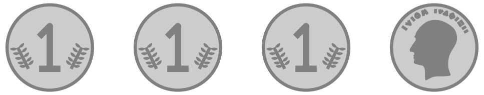
# Randomized Skip Lists

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




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

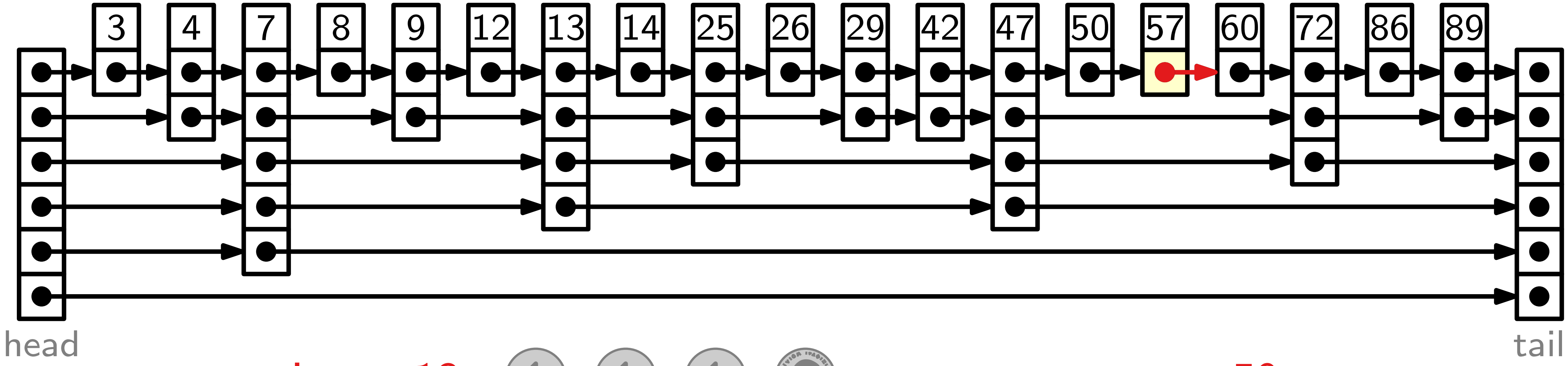
e.g. insert 13



e.g. remove 59

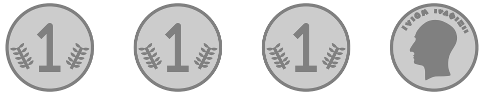
# Randomized Skip Lists

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




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

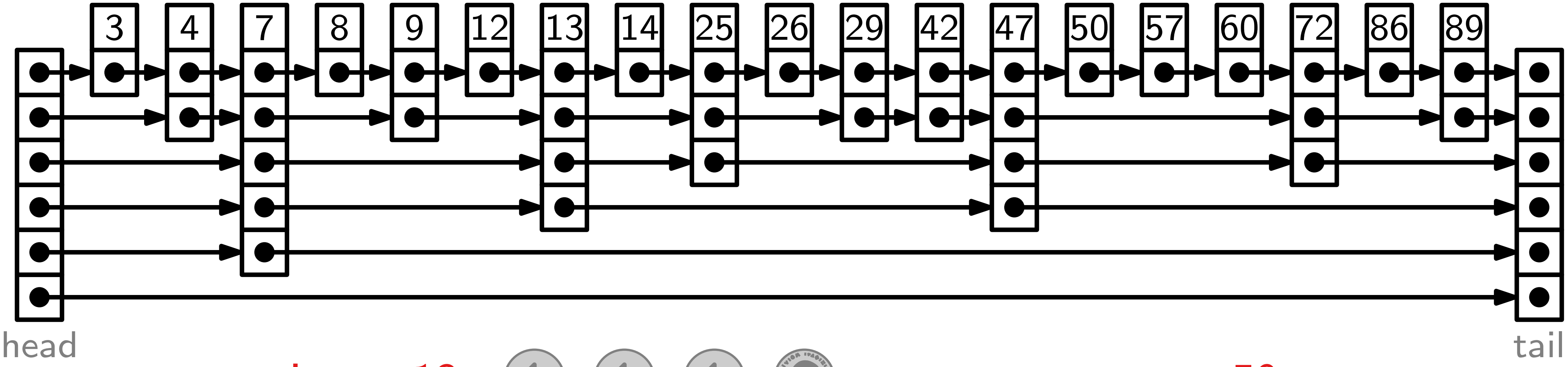
e.g. insert 13



e.g. remove 59

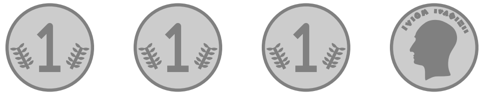
# Randomized Skip Lists

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




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

e.g. insert 13

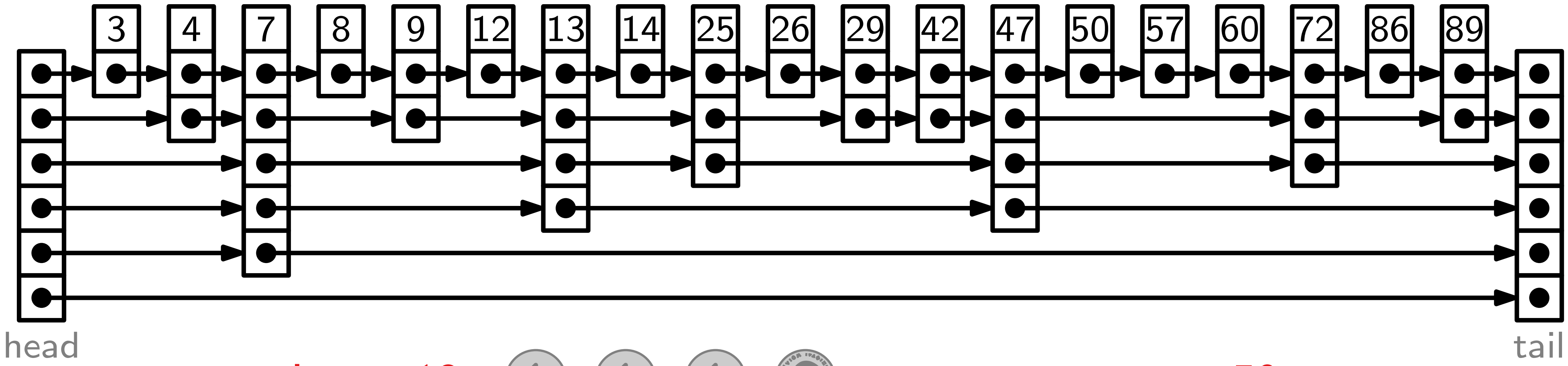



e.g. remove 59

Insertion and deletion works in $O(\log n)$ time + the time to search a key.
(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.)

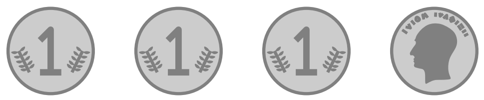
# Randomized Skip Lists

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




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

e.g. insert 13



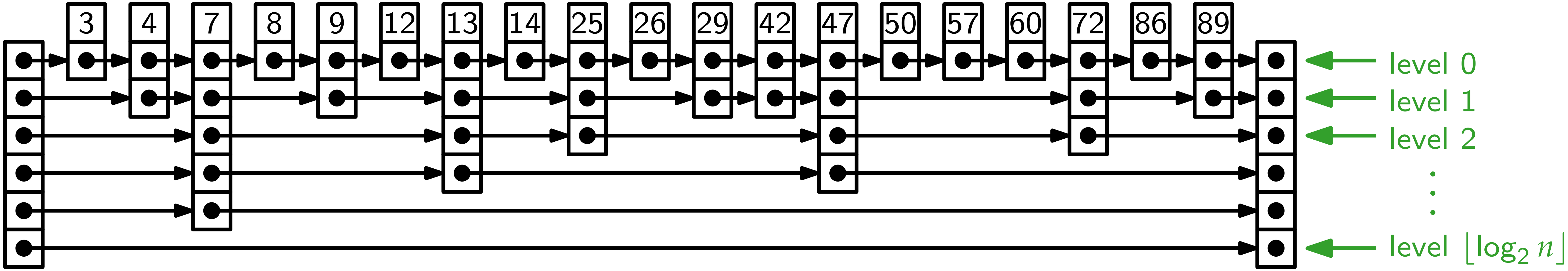
e.g. remove 59

Insertion and deletion works in $O(\log n)$ time + the time to search a key.
(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

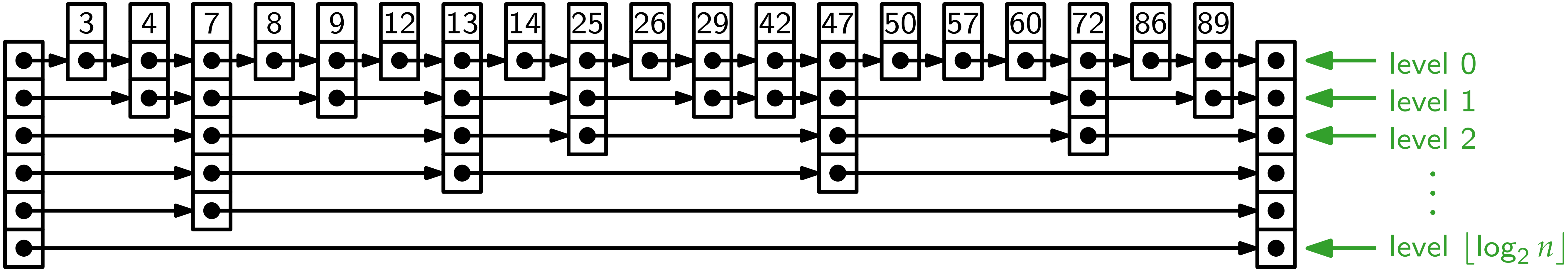
## Proof of Theorem 1.




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

# Randomized Skip Lists

## Proof of Theorem 1.

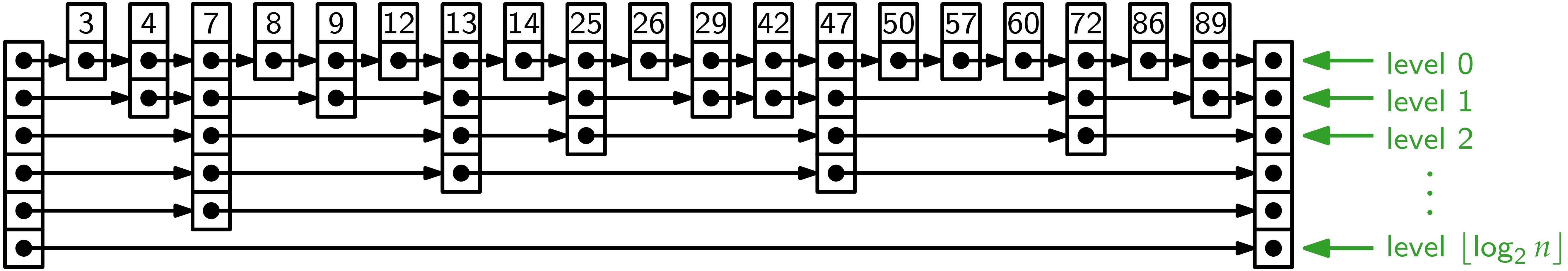



How long is the search path to reach the key we search for?

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

# Randomized Skip Lists

## Proof of Theorem 1.



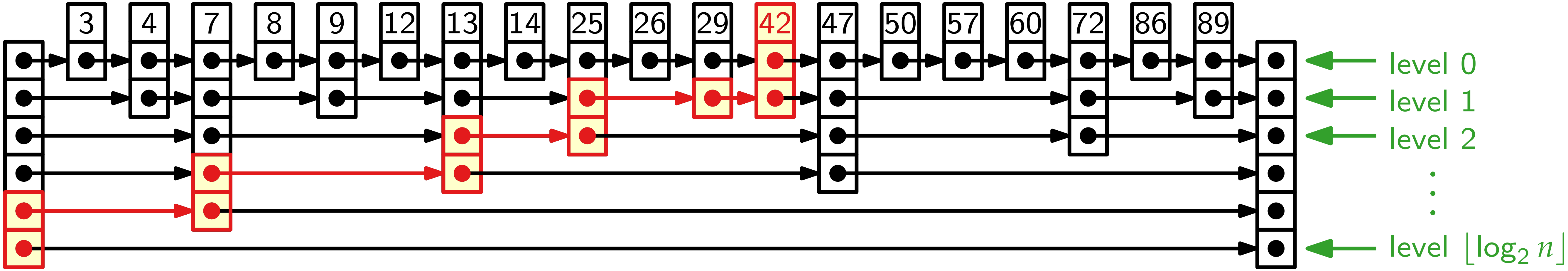

How long is the search path to reach the key we search for?

We do backwards analysis ($\rightarrow$ 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.**



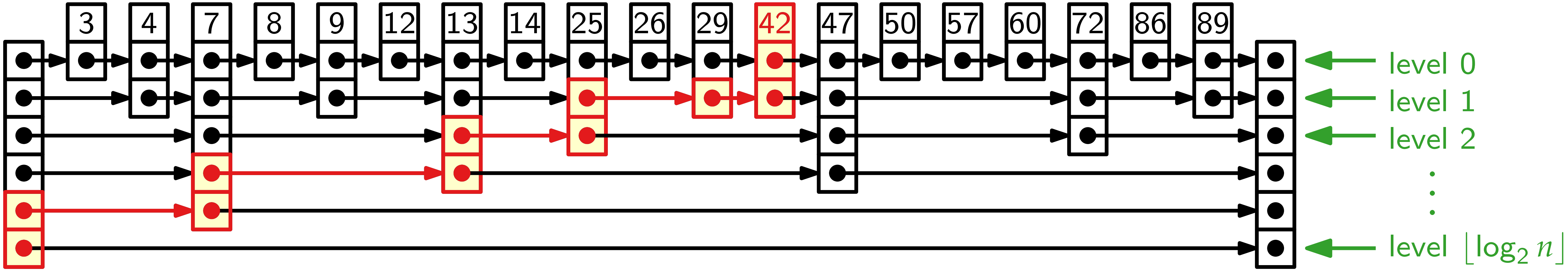

How long is the search path to reach the key we search for?

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 key we search for?

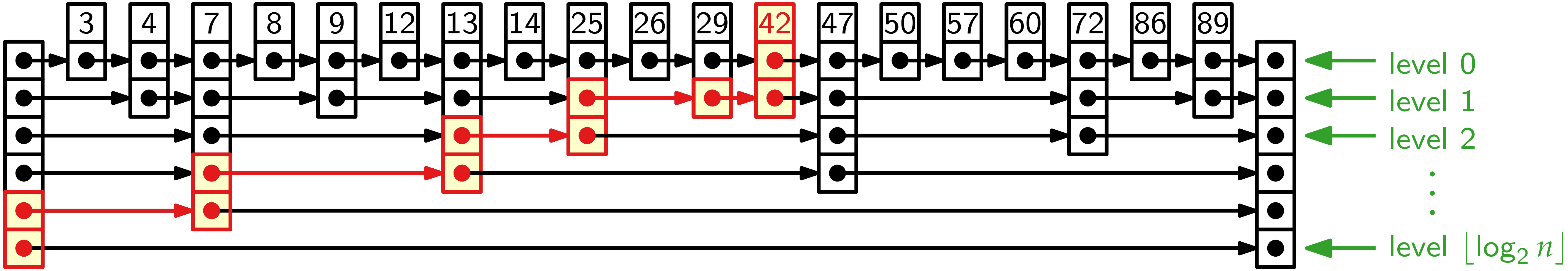
We do backwards analysis (→ see lecture on rand. algorithms) on the search path.

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 key we search for?

We do backwards analysis ($\to$ see lecture on rand. algorithms) on the search path.

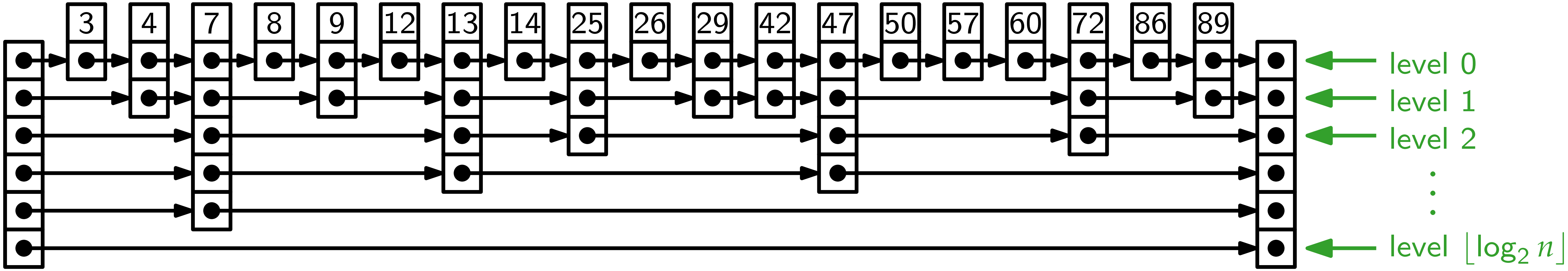
In the reverse search path, we always go to the next greater level if possible, otherwise we follow the (reverse) pointer to the left.

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.

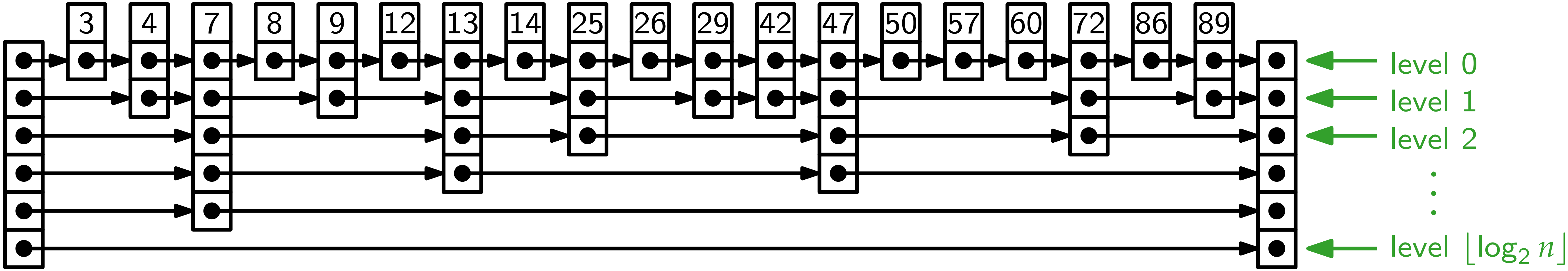



Let $X_i$ be a random variable denoting the number of steps we take on level $i$ or lower.

**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

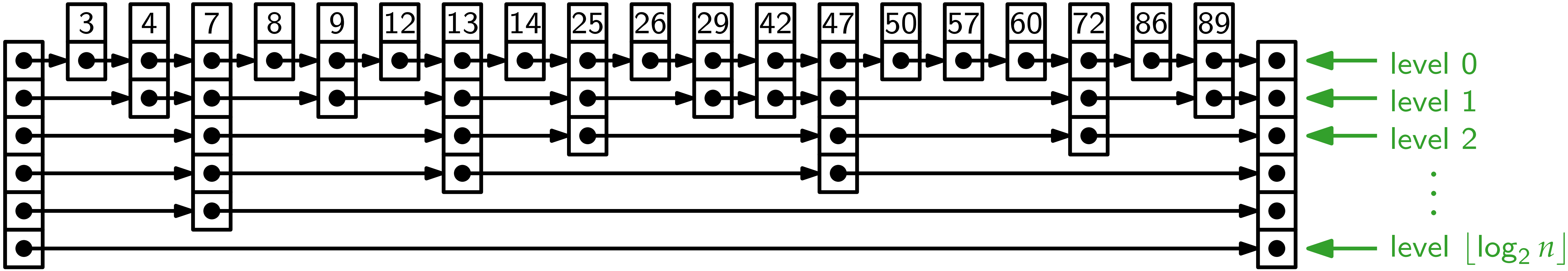
in the previous step we went a level up

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

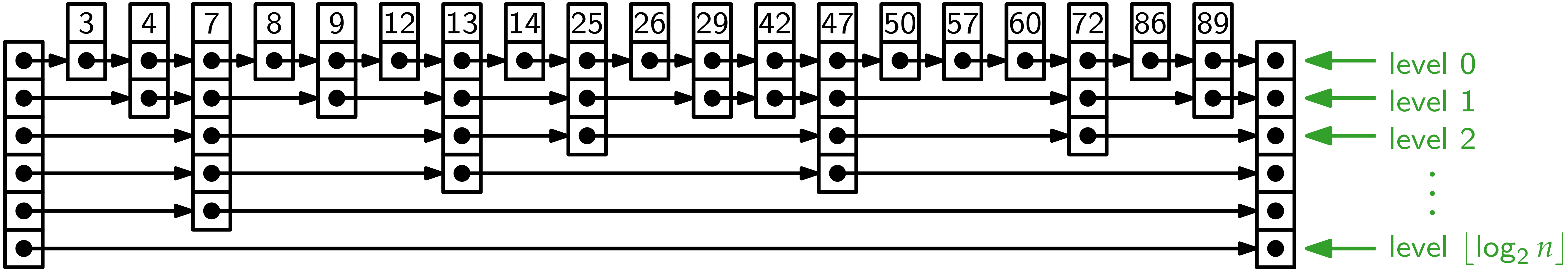
in the previous step we went a level up

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.

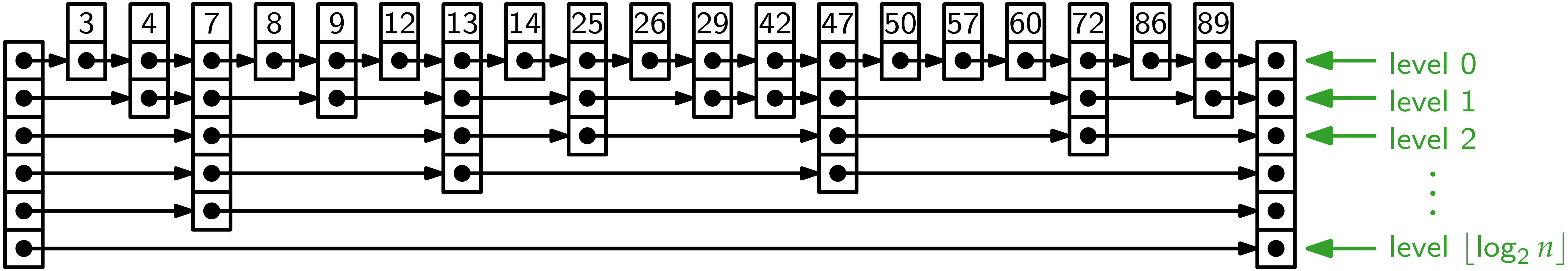
$E[X_i] = 1 + \frac{1}{2}E[X_{i-1}] + \frac{1}{2}E[X_i] \iff E[X_i] = 2 + E[X_{i-1}]$ (for $i < 0$: $E[X_i] = 0$)

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

**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.

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

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

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

□

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

**Binary-search-tree property:**
Let $x$ be a key in the tree.
For every key $y$ in the left (right) subtree of $x$ it holds that $y \leq x$ ($y \geq x$).

**Binary-search-tree property:**
Let $x$ be a key in the tree.
For every key $y$ in the left (right) subtree of $x$ it holds that $y \leq x$ ($y \geq x$).

**Heap property:**
Let $a$ be a key in the tree.
For every child $b$ of $a$ it holds that $b \geq a$.

# Treaps

**Binary-search-tree property:**
Let $x$ be a key in the tree.
For every key $y$ in the left (right) subtree of $x$ it holds that $y \leq x$ $(y \geq x)$.

**Heap property:**
Let $a$ be a key in the tree.
For every child $b$ of $a$ it holds that $b \geq a$.

Combine both properties $\Rightarrow$ **Treap**

# Treaps

**Binary-search-tree property:**
Let $x$ be a key in the tree.
For every key $y$ in the left (right) subtree of $x$ it holds that $y \leq x$ $(y \geq x)$.

**Heap property:**
Let $a$ be a key in the tree.
For every child $b$ of $a$ it holds that $b \geq a$.

Combine both properties $\Rightarrow$ **Treap**

- A treap is a randomized tree data structure to store a set of keys.

# Treaps

**Binary-search-tree property:**
Let $x$ be a key in the tree.
For every key $y$ in the left (right) subtree of $x$ it holds that $y \leq x$ ($y \geq x$).

**Heap property:**
Let $a$ be a key in the tree.
For every child $b$ of $a$ it holds that $b \geq a$.

Combine both properties $\Rightarrow$ **Treap**

- A treap is a randomized tree data structure to store a set of keys.
- Every node of the tree contains one of the keys and a randomly chosen priority.

# Treaps

**Binary-search-tree property:**
Let $x$ be a key in the tree.
For every key $y$ in the left (right) subtree of $x$ it holds that $y \leq x$ ($y \geq x$).

**Heap property:**
Let $a$ be a key in the tree.
For every child $b$ of $a$ it holds that $b \geq a$.

Combine both properties $\Rightarrow$ **Treap**

- A treap is a randomized tree data structure to store a set of keys.
- Every node of the tree contains one of the keys and a randomly chosen priority.
- The keys in the tree fulfill the binary-search-tree property.

# Treaps

**Binary-search-tree property:**
Let $x$ be a key in the tree.
For every key $y$ in the left (right) subtree of $x$ it holds that $y \le x$ ($y \ge x$).

**Heap property:**
Let $a$ be a ~~key~~ priority in the tree.
For every child $b$ of $a$ it holds that $b \ge a$.

Combine both properties $\Rightarrow$ **Treap**

- A treap is a randomized tree data structure to store a set of keys.
- Every node of the tree contains one of the keys and a randomly chosen priority.
- The keys in the tree fulfill the binary-search-tree property.
- The priorities in the tree fulfill the heap property.

# Building Treaps

We build a treap for the key set $S = \{34, 8, 99, 1, 4, 2, 42, 66\}$ by inserting each key.

# Building Treaps

We build a treap for the key set $S = \{34, 8, 99, 1, 4, 2, 42, 66\}$ by inserting each key.

we start with the empty tree

# Building Treaps

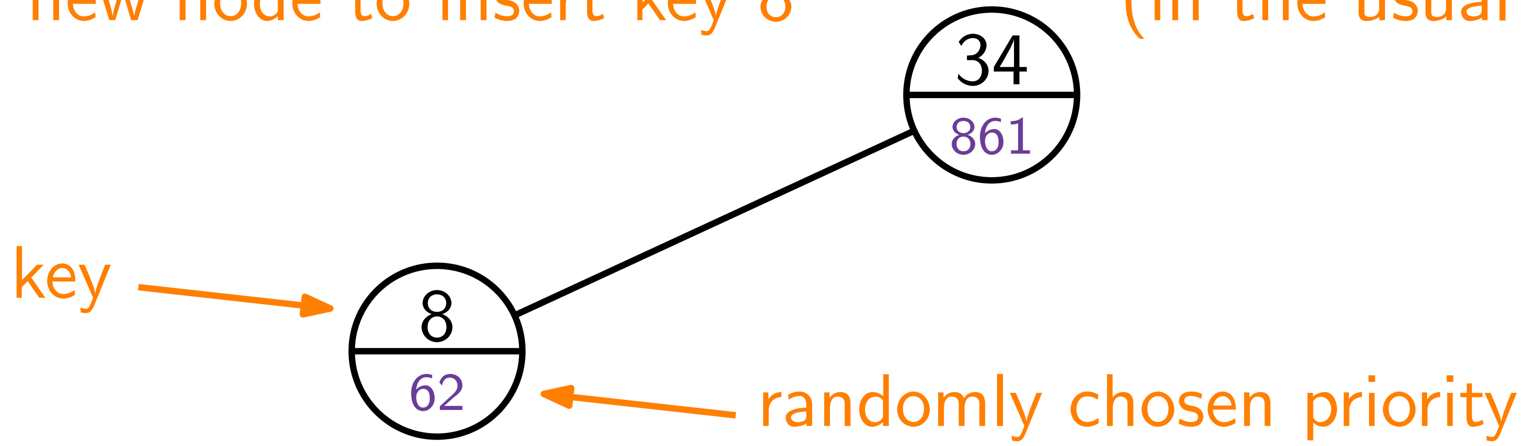
We build a treap for the key set $S = \{34, 8, 99, 1, 4, 2, 42, 66\}$ by inserting each key.

create a new node to insert key 34

# Building Treaps

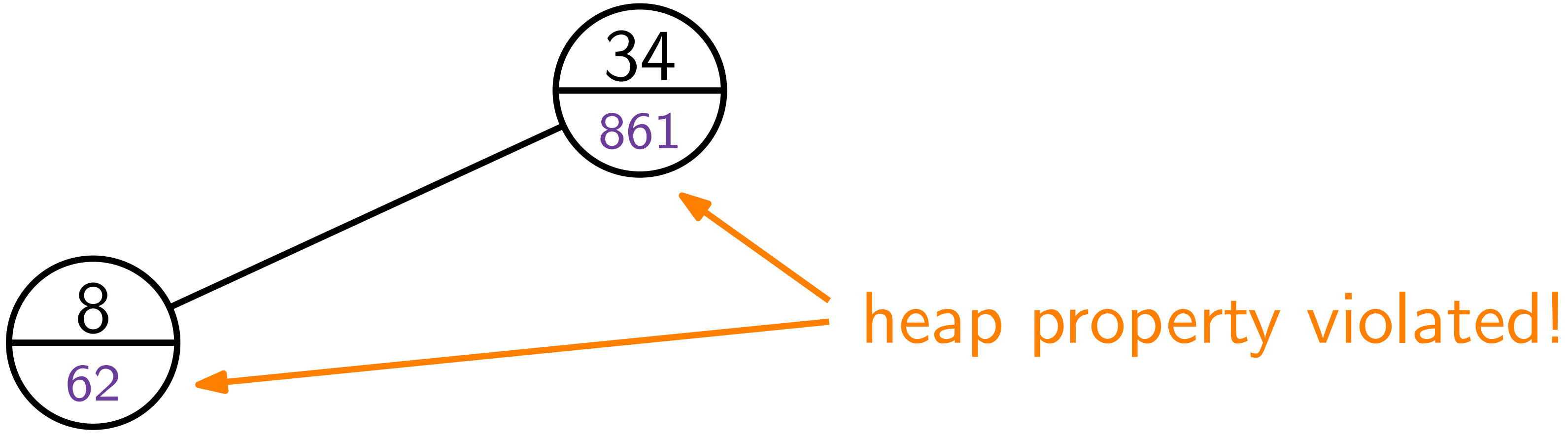
We build a treap for the key set $S = \{34, 8, 99, 1, 4, 2, 42, 66\}$ by inserting each key.

create a new node to insert key 8 (in the usual way for a binary search tree)



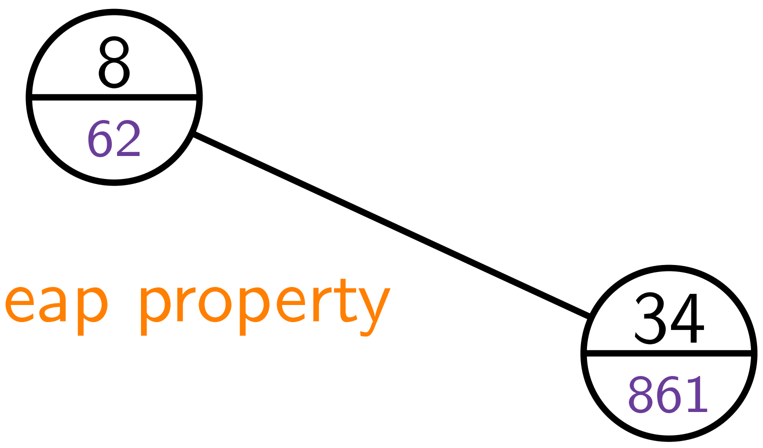
# Building Treaps

We build a treap for the key set $S = \{34, 8, 99, 1, 4, 2, 42, 66\}$ by inserting each key.



# Building Treaps

We build a treap for the key set $S = \{34, 8, 99, 1, 4, 2, 42, 66\}$ by inserting each key.

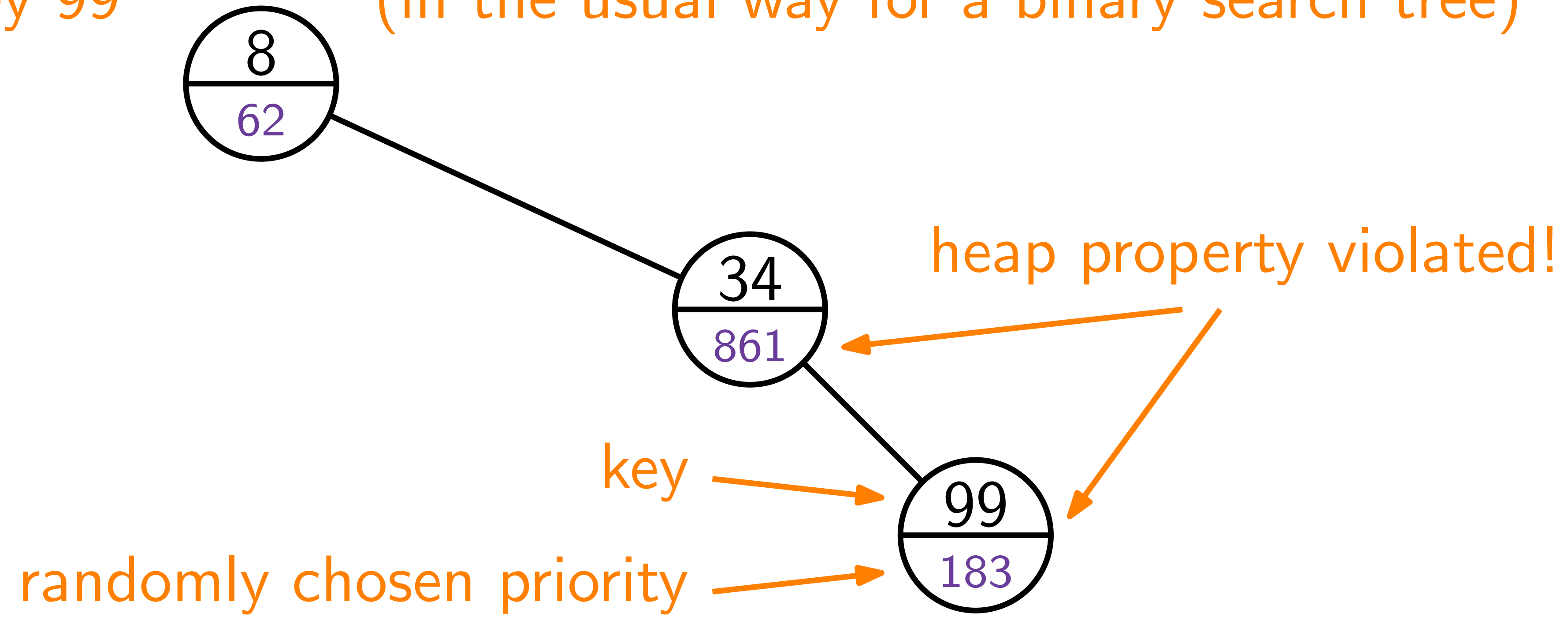



do a right rotation to fix heap property

# Building Treaps

We build a treap for the key set $S = \{34, 8, 99, 1, 4, 2, 42, 66\}$ by inserting each key.

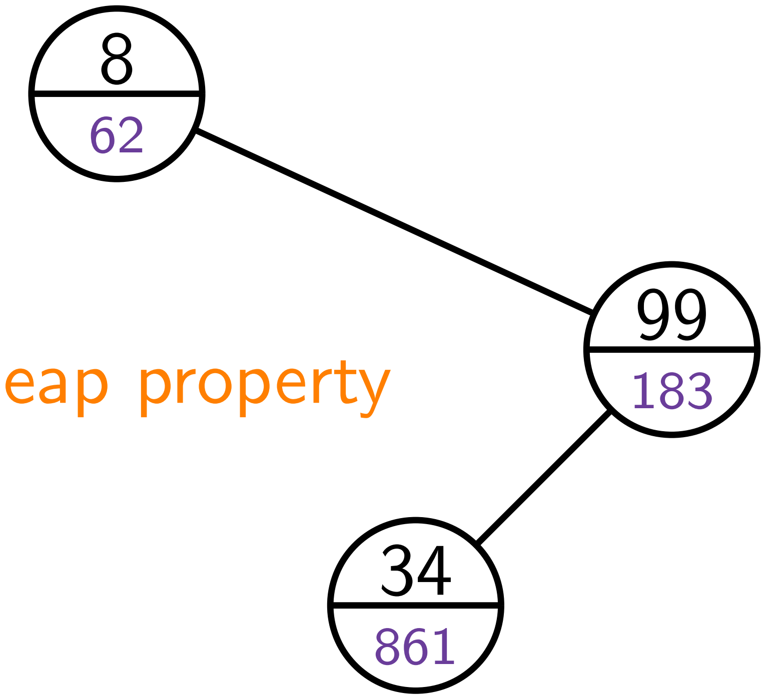
create a new node to insert key 99 (in the usual way for a binary search tree)



# Building Treaps

We build a treap for the key set $S = \{34, 8, 99, 1, 4, 2, 42, 66\}$ by inserting each key.

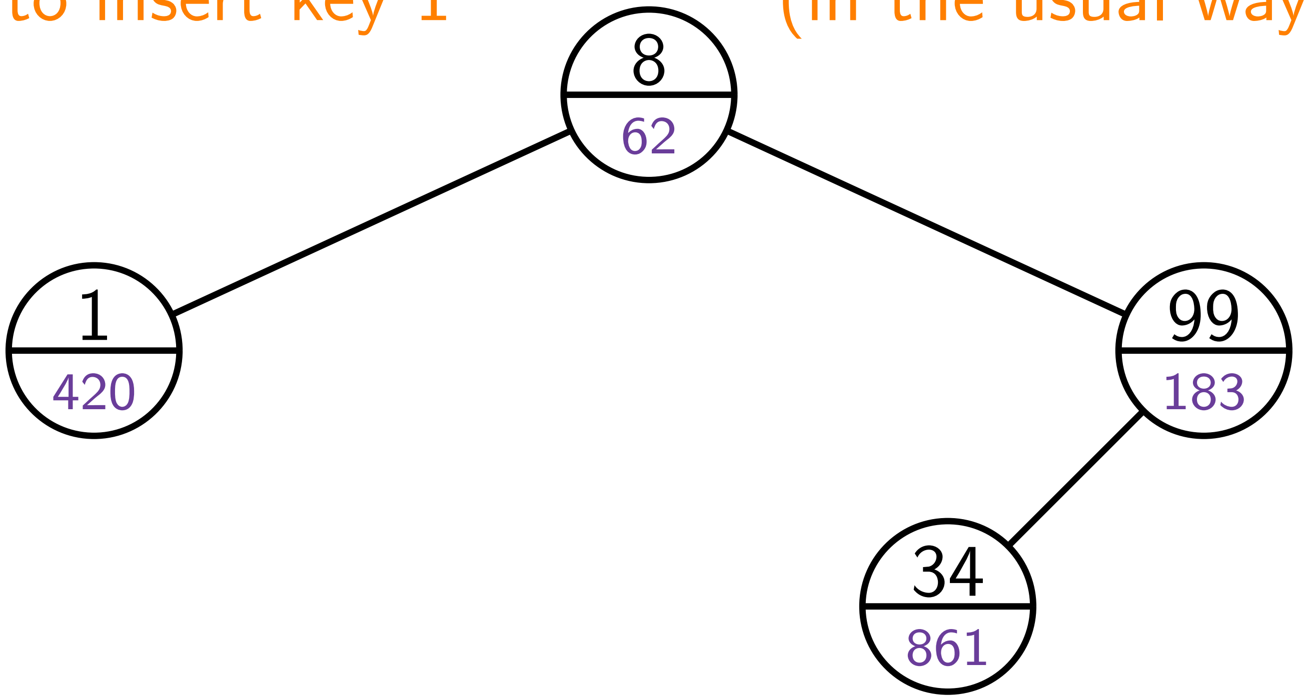
do a left rotation to fix heap property



# Building Treaps

We build a treap for the key set $S = \{34, 8, 99, 1, 4, 2, 42, 66\}$ by inserting each key.

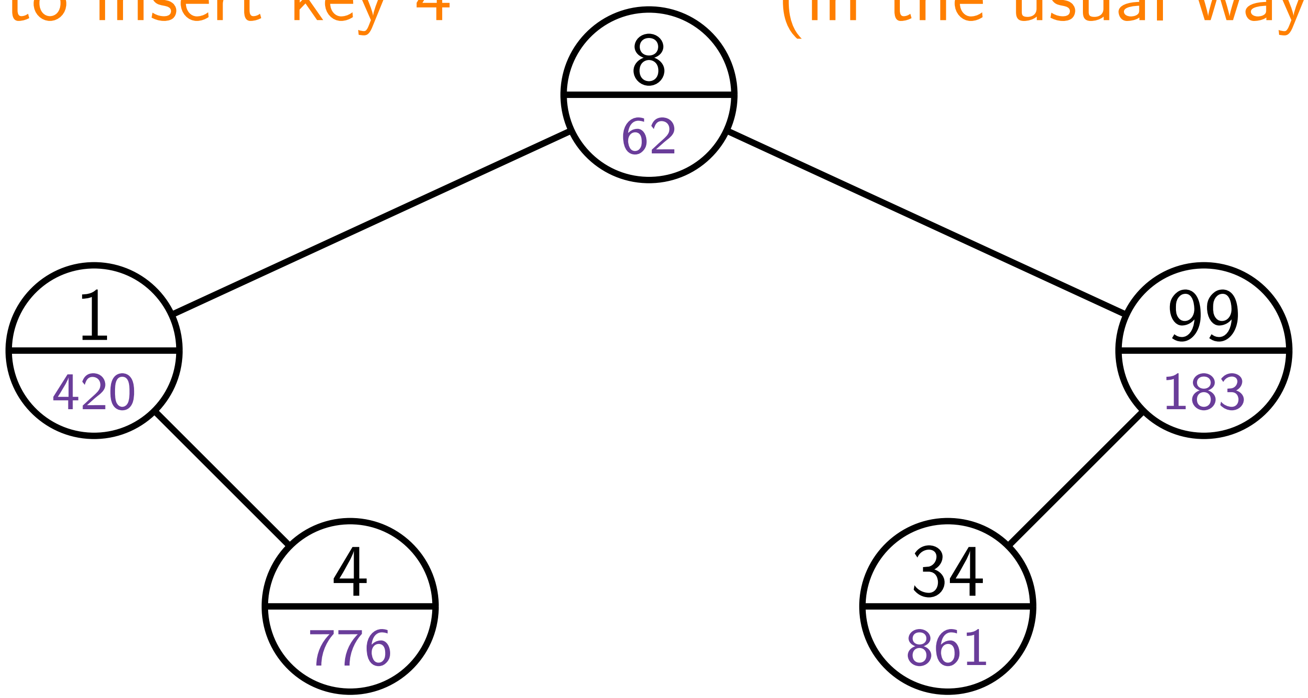
create a new node to insert key 1 (in the usual way for a binary search tree)



# Building Treaps

We build a treap for the key set $S = \{34, 8, 99, 1, 4, 2, 42, 66\}$ by inserting each key.

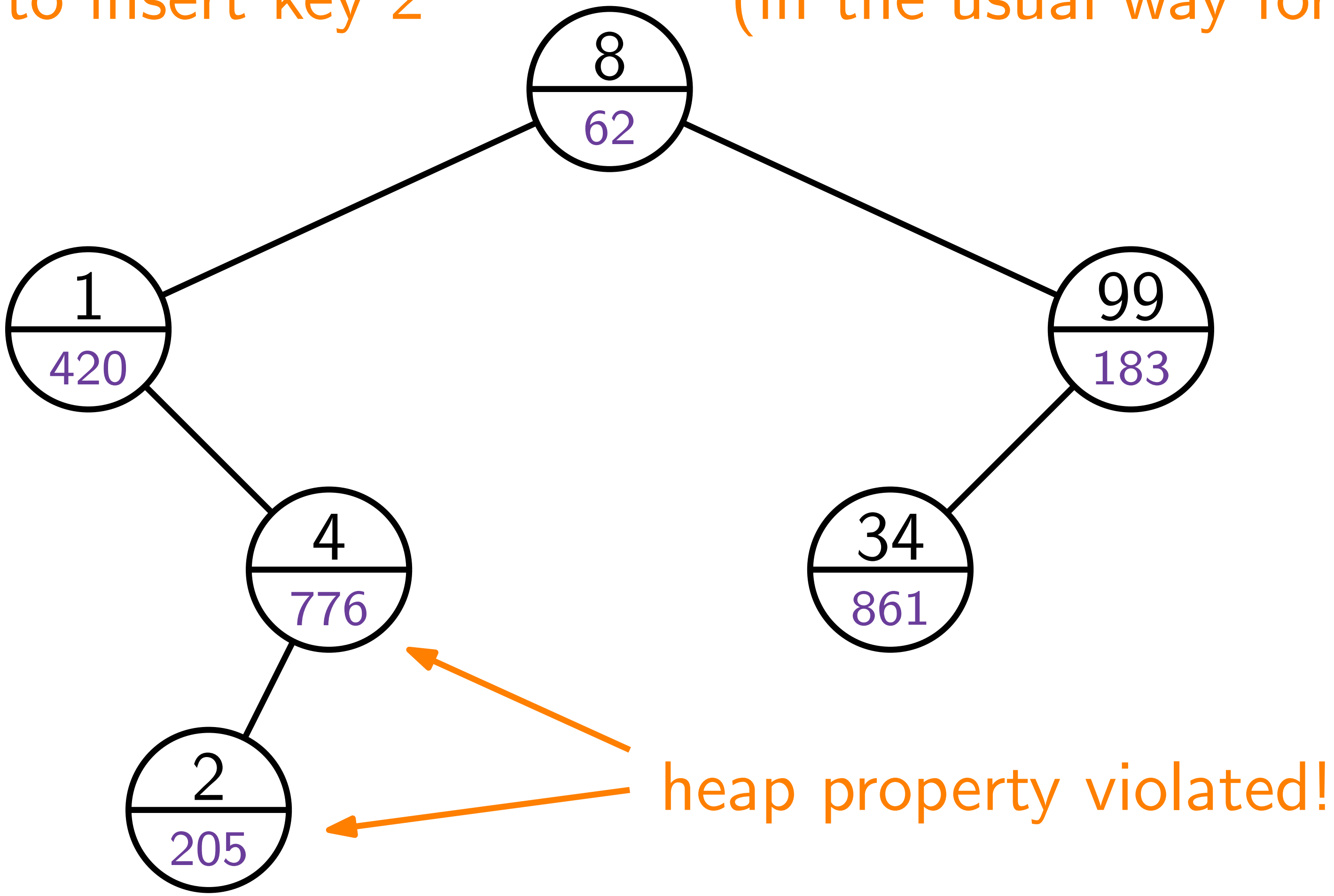
create a new node to insert key 4 (in the usual way for a binary search tree)



# Building Treaps

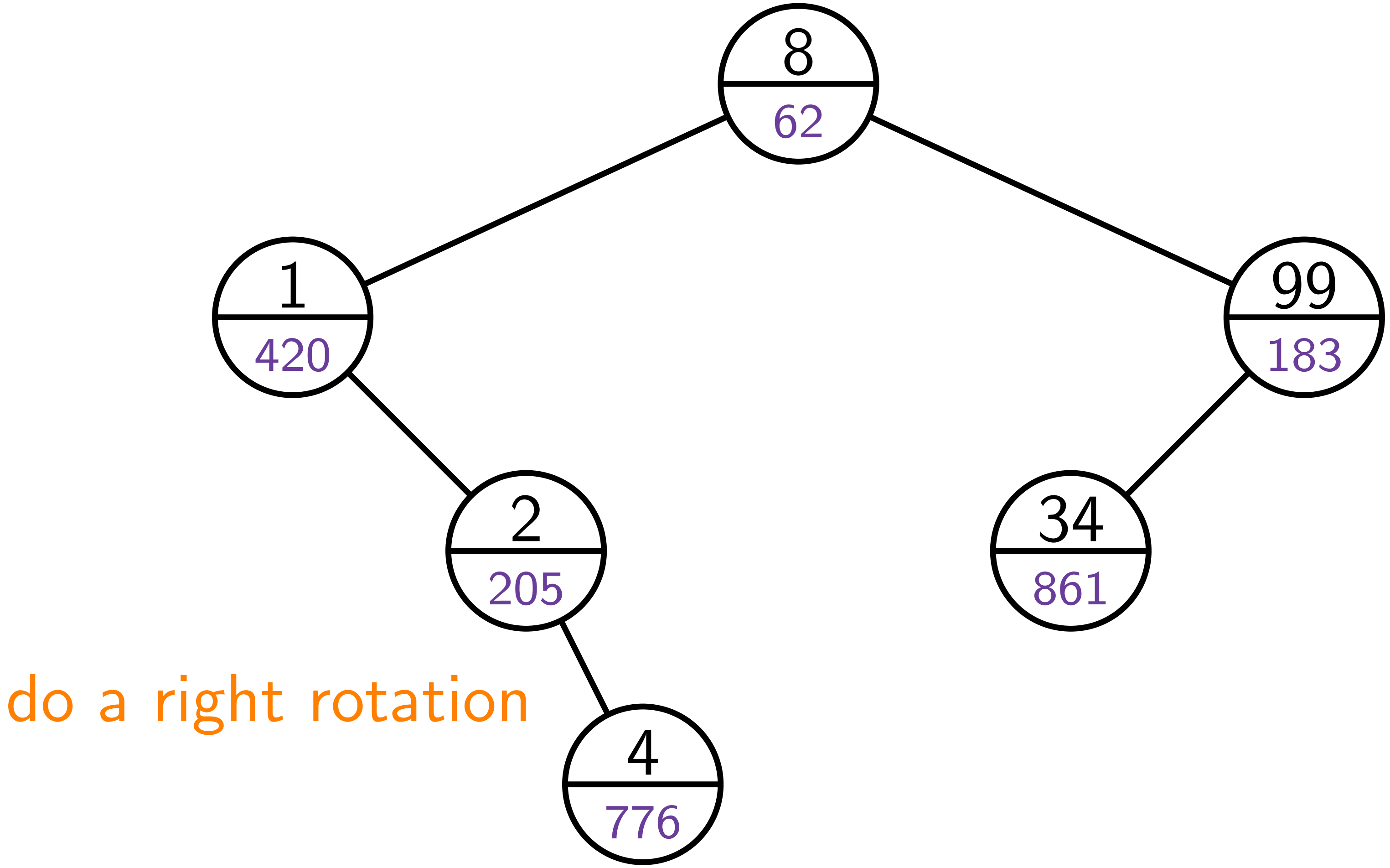
We build a treap for the key set $S = \{34, 8, 99, 1, 4, 2, 42, 66\}$ by inserting each key.

create a new node to insert key 2 (in the usual way for a binary search tree)



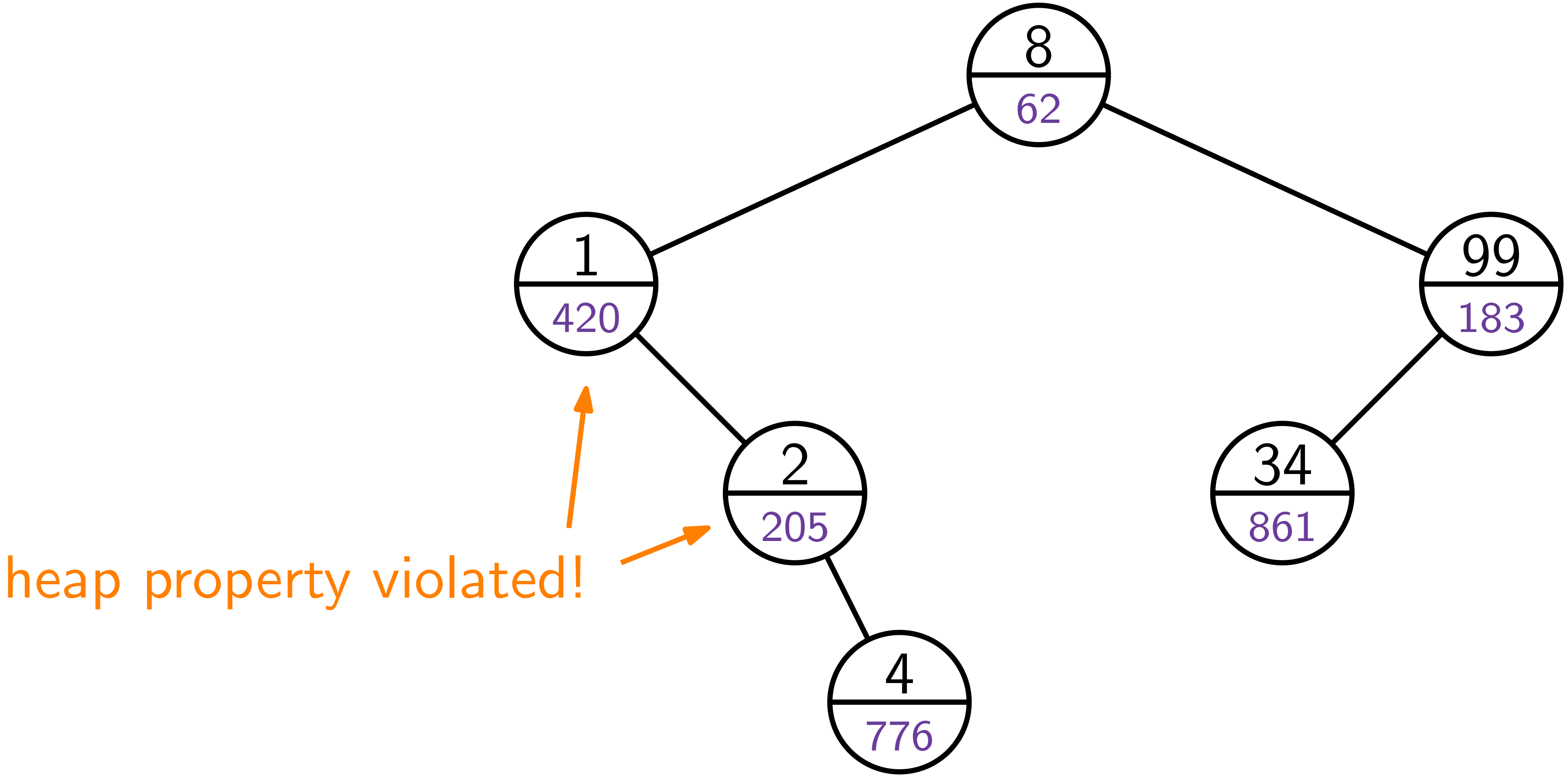
# Building Treaps

We build a treap for the key set $S = \{34, 8, 99, 1, 4, 2, 42, 66\}$ by inserting each key.



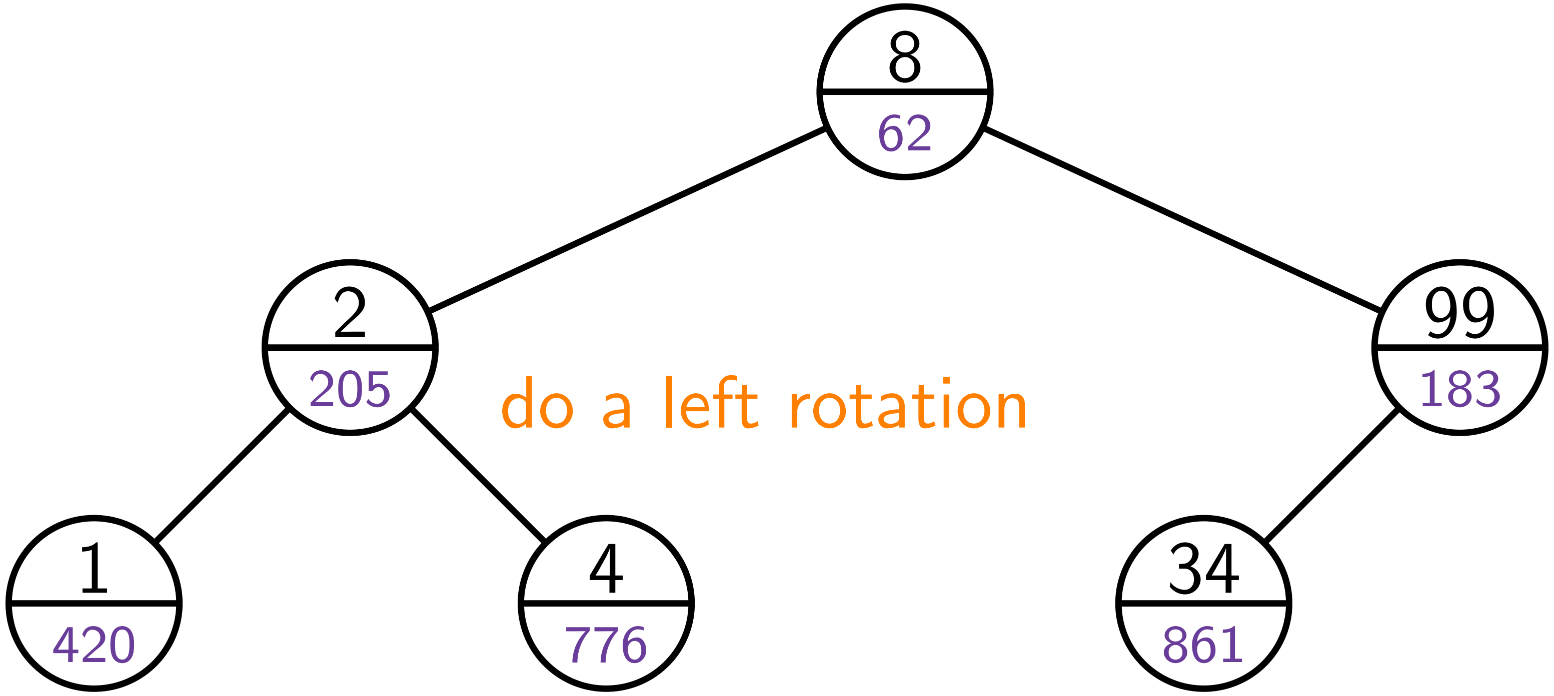
# Building Treaps

We build a treap for the key set $S = \{34, 8, 99, 1, 4, 2, 42, 66\}$ by inserting each key.



# Building Treaps

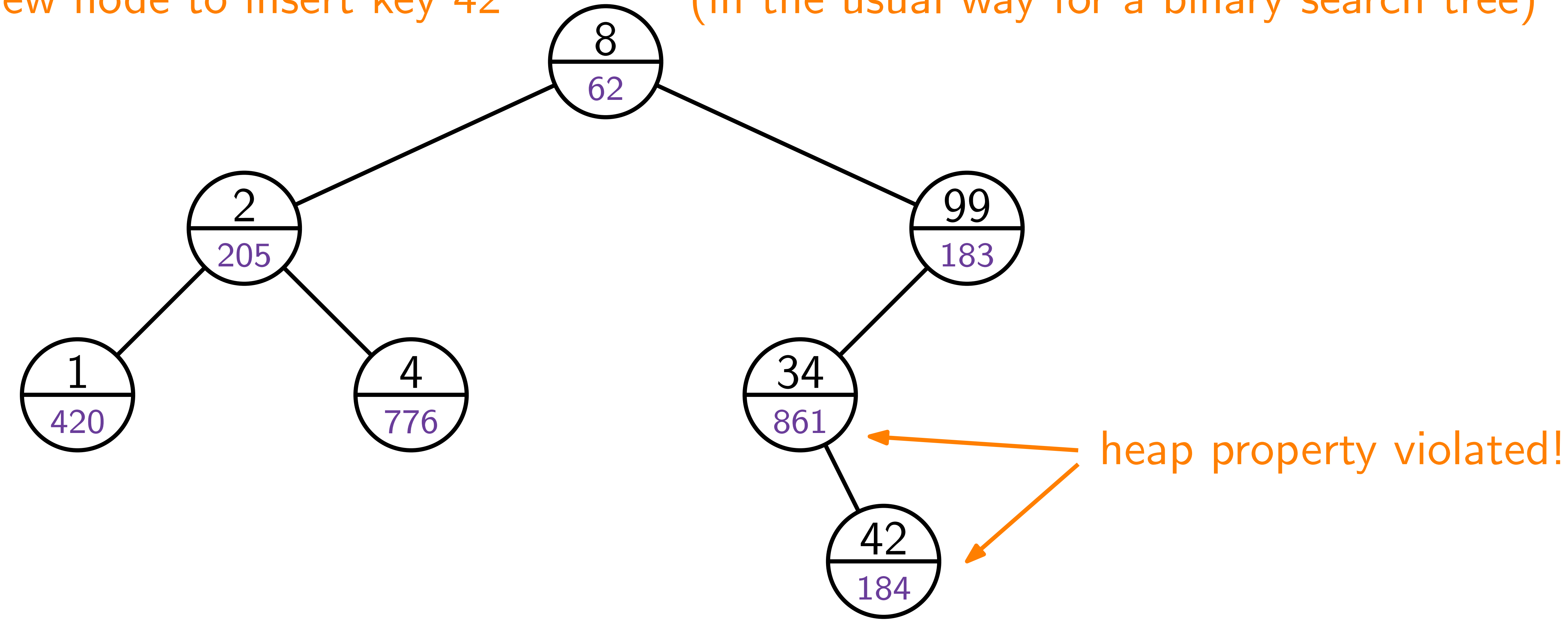
We build a treap for the key set $S = \{34, 8, 99, 1, 4, 2, 42, 66\}$ by inserting each key.



# Building Treaps

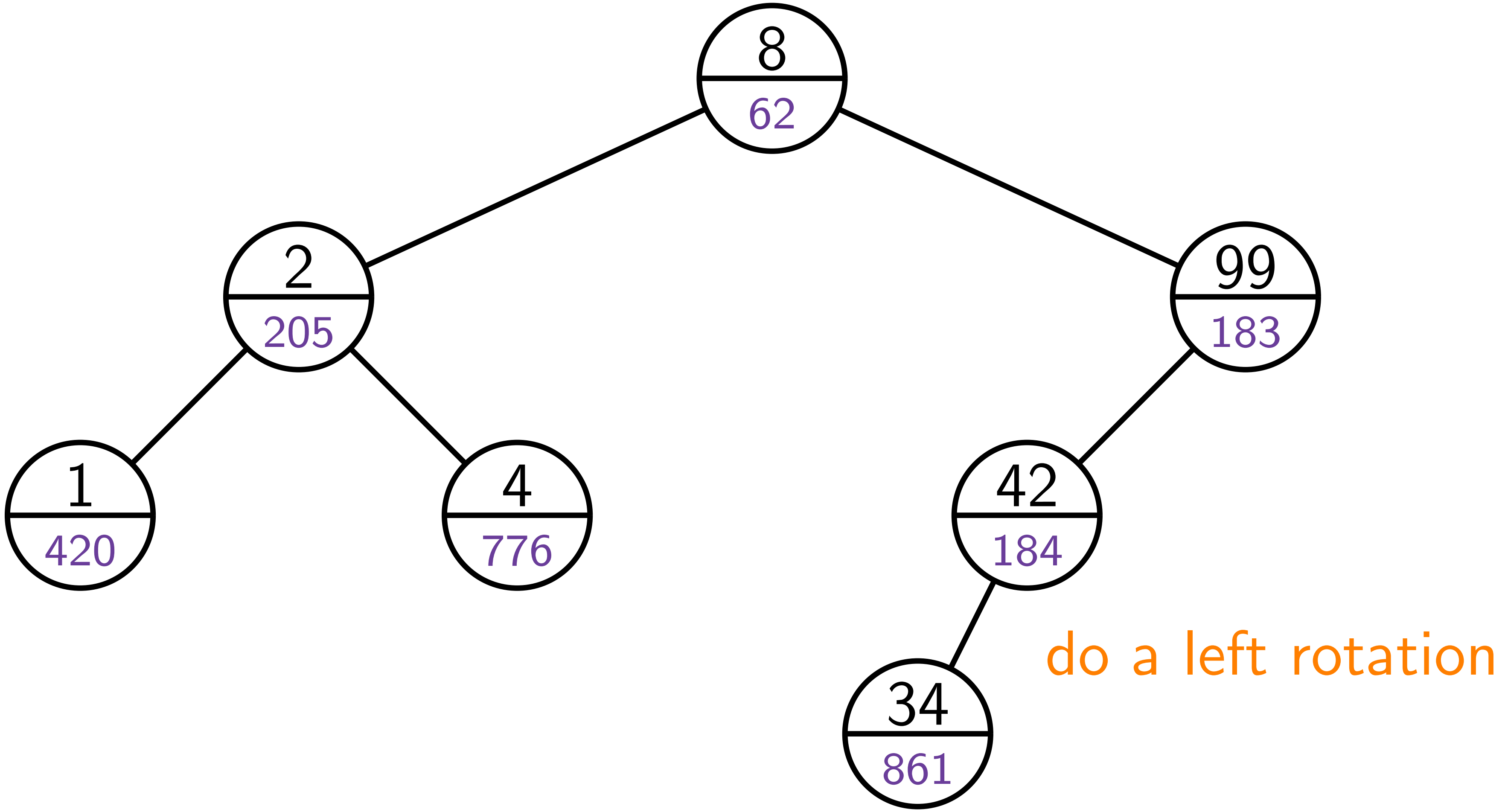
We build a treap for the key set $S = \{34, 8, 99, 1, 4, 2, 42, 66\}$ by inserting each key.

create a new node to insert key 42 (in the usual way for a binary search tree)



# Building Treaps

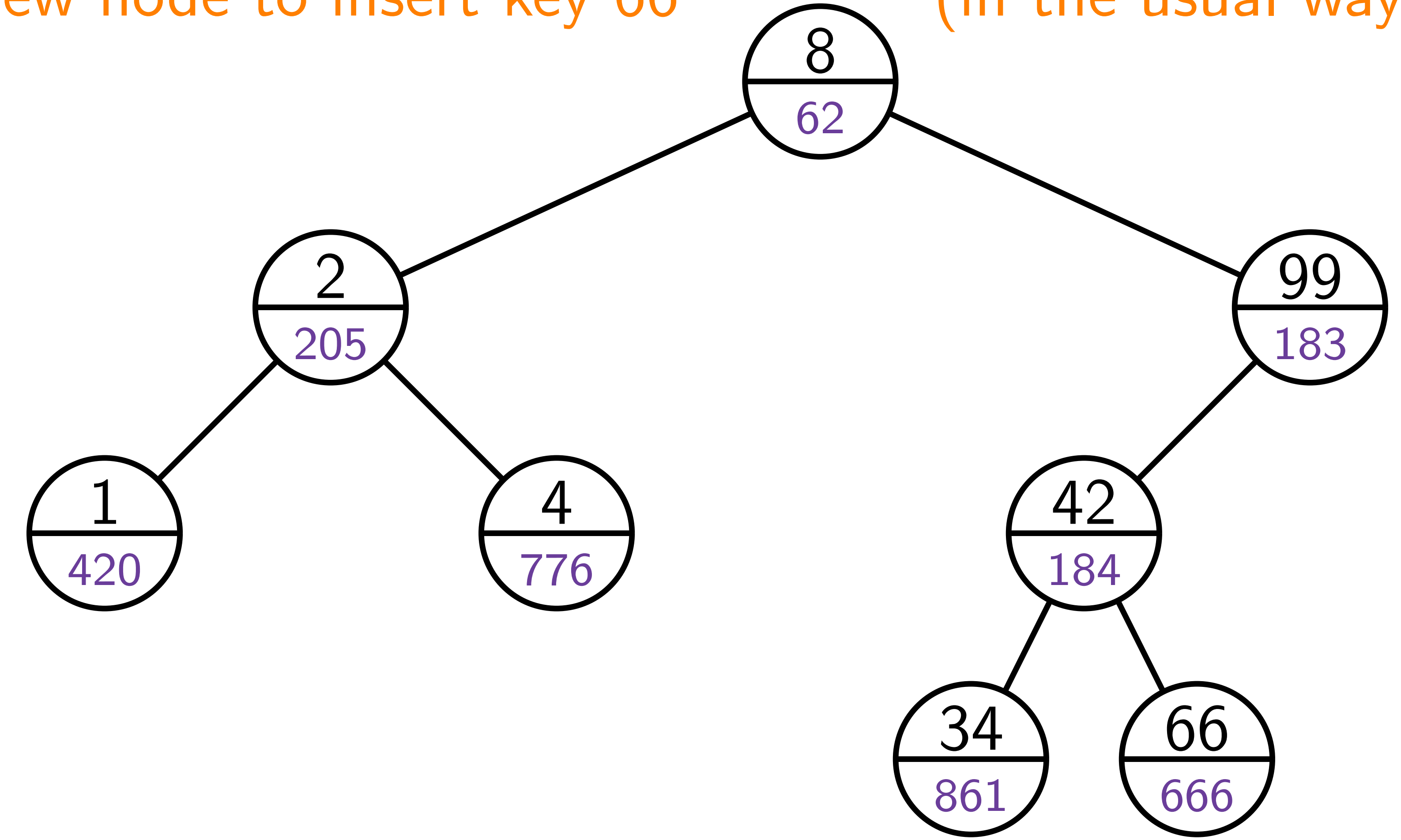
We build a treap for the key set $S = \{34, 8, 99, 1, 4, 2, 42, 66\}$ by inserting each key.



# Building Treaps

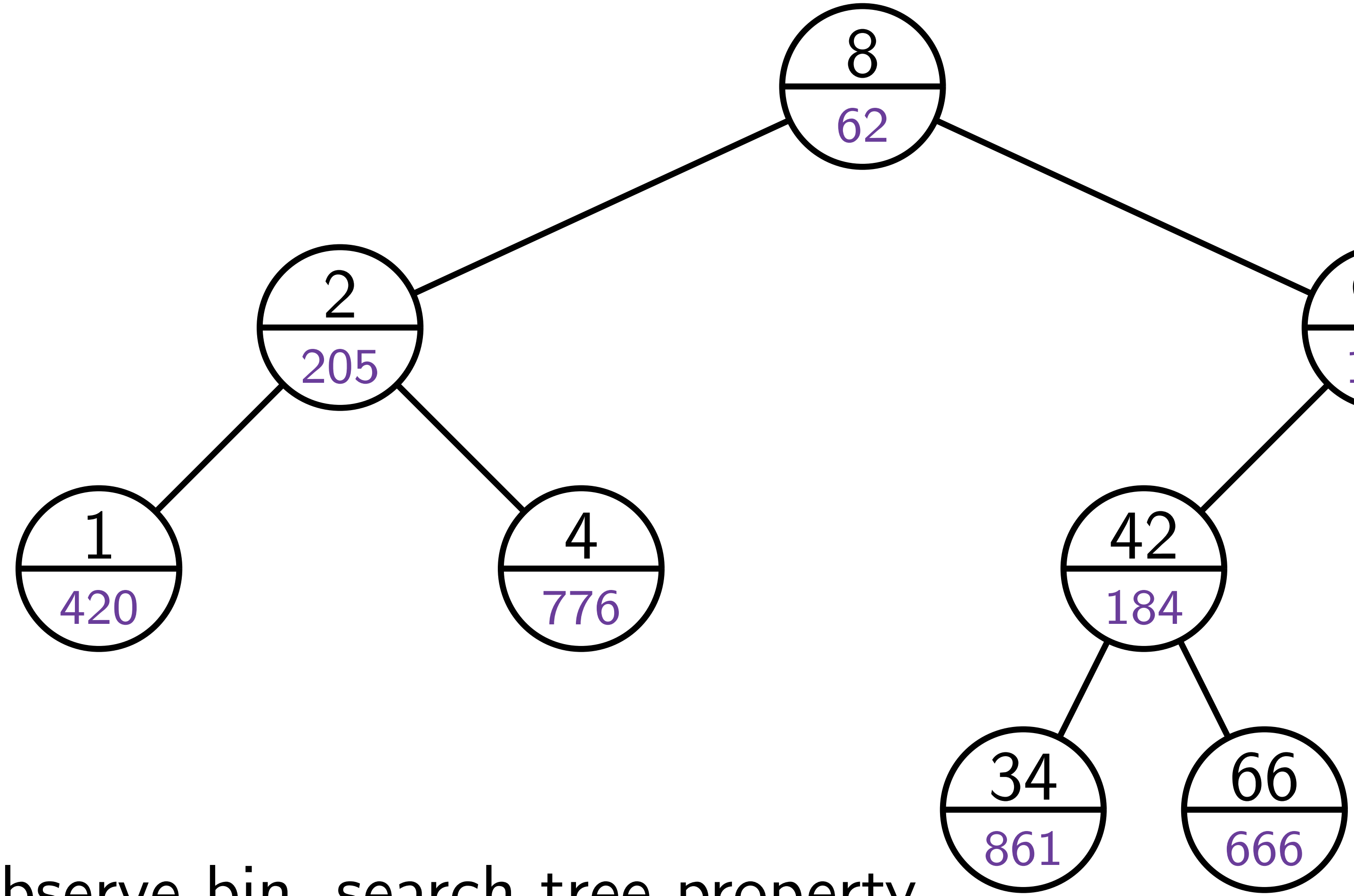
We build a treap for the key set $S = \{34, 8, 99, 1, 4, 2, 42, 66\}$ by inserting each key.

create a new node to insert key 66 (in the usual way for a binary search tree)



# Building Treaps

We build a treap for the key set $S = \{34, 8, 99, 1, 4, 2, 42, 66\}$ by inserting each key.

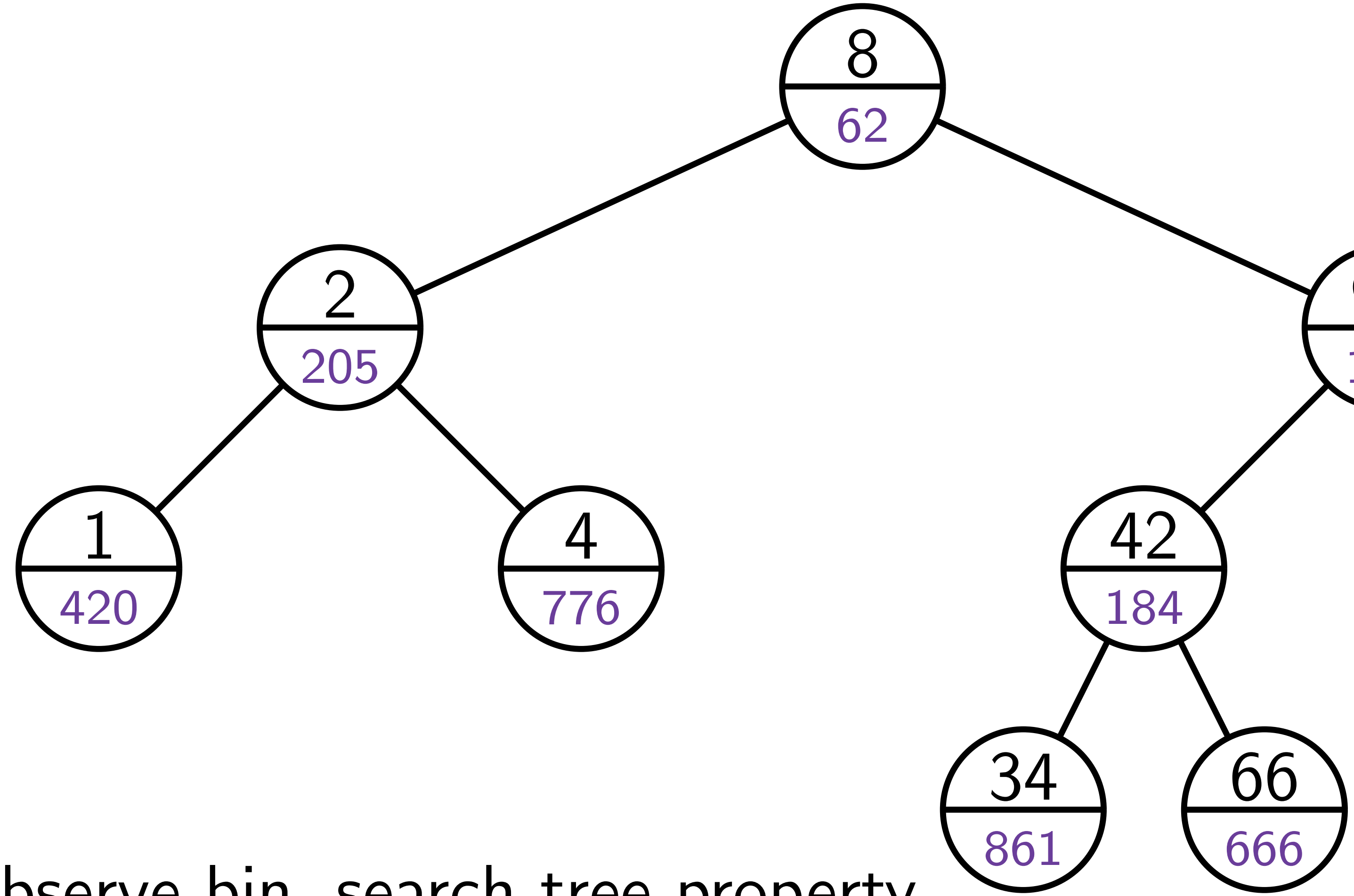



$\Rightarrow$ Keys observe bin.-search-tree property.

$\Rightarrow$ Random priorities observe heap property.

# Building Treaps

We build a treap for the key set $S = \{34, 8, 99, 1, 4, 2, 42, 66\}$ by inserting each key.

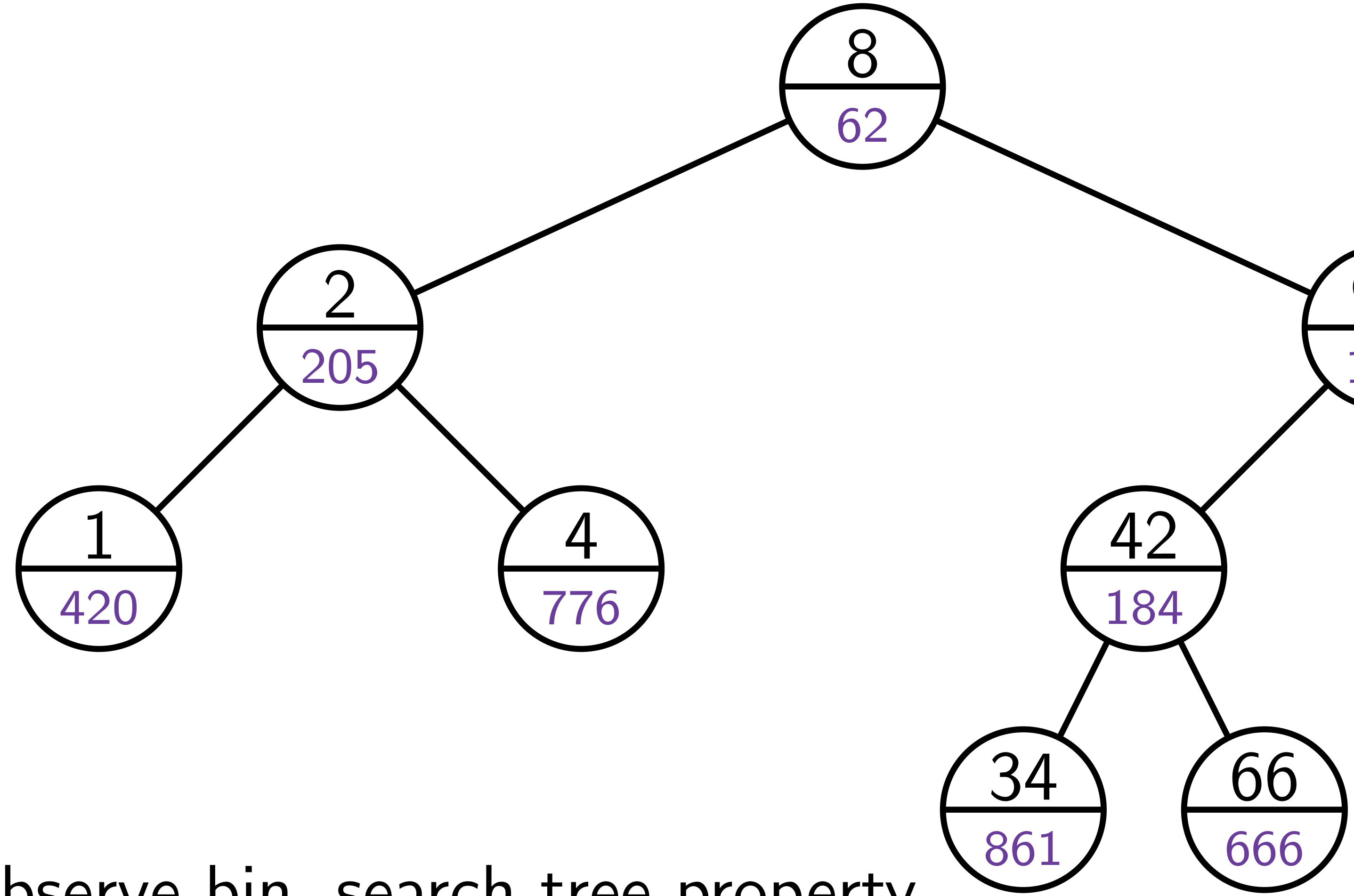



- We assume that all priorities are distinct.

$\Rightarrow$ Keys observe bin.-search-tree property.
$\Rightarrow$ Random priorities observe heap property.

# Building Treaps

We build a treap for the key set $S = \{34, 8, 99, 1, 4, 2, 42, 66\}$ by inserting each key.

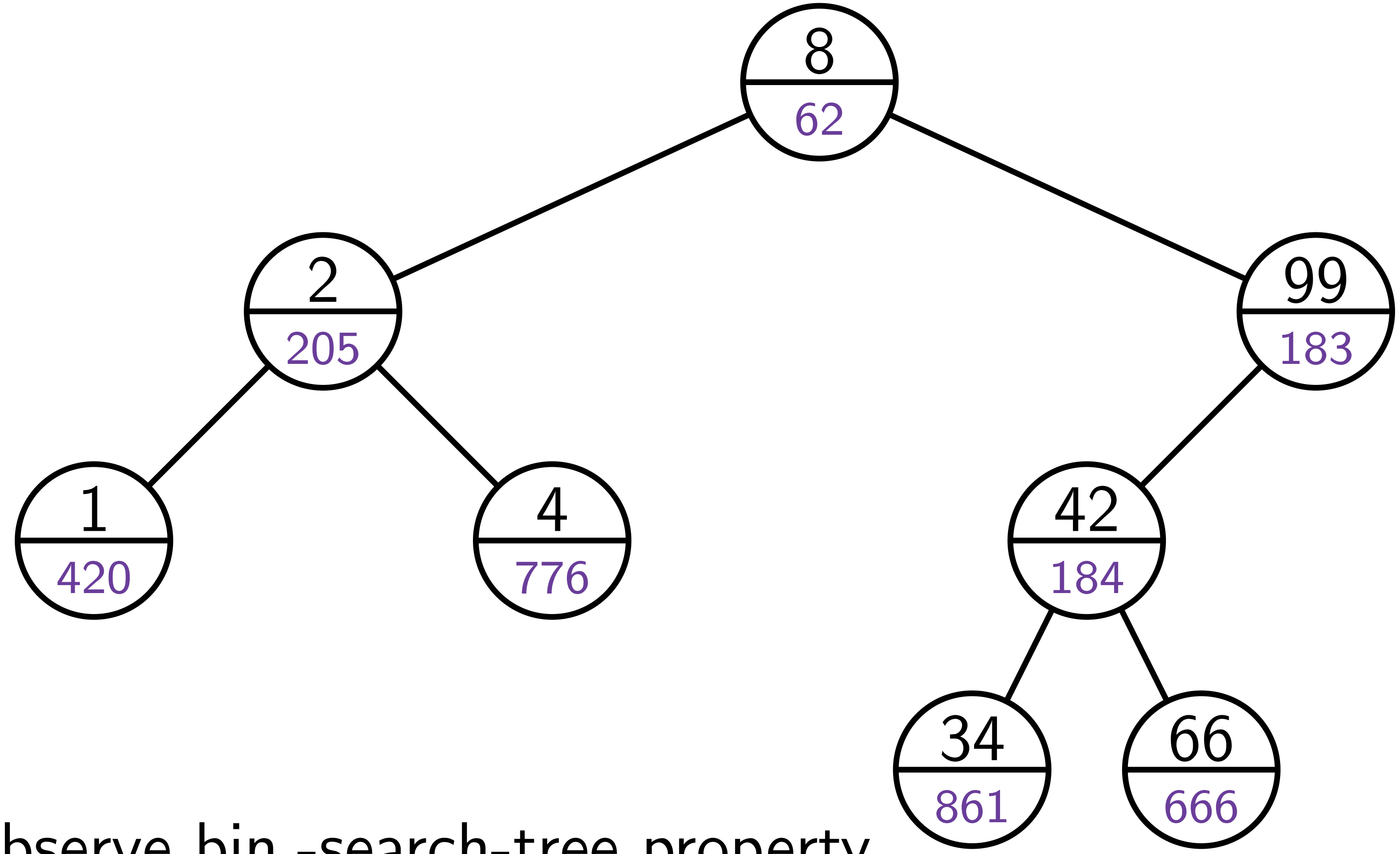



- We assume that all priorities are distinct.
- Note that the resulting tree is not necessarily balanced (in w.c. it has linear depth, but we'll see later that in expectation it is better).

$\Rightarrow$ Keys observe bin.-search-tree property.
$\Rightarrow$ Random priorities observe heap property.

# Building Treaps

We build a treap for the key set $S = \{34, 8, 99, 1, 4, 2, 42, 66\}$ by inserting each key.




- We assume that all priorities are distinct.
- Note that the resulting tree is not necessarily balanced (in w.c. it has linear depth, but we'll see later that in expectation it is better).
- Deletion of an element works similar to insertion. $\rightarrow$ **Exercise**

$\Rightarrow$ Keys observe bin.-search-tree property.
$\Rightarrow$ Random priorities observe heap property.

# Properties of Treaps

**Theorem 2.** Given the pairs of keys and priorities, the structure of a treap is unique.

# Properties of Treaps

**Theorem 2.** Given the pairs of keys and priorities, the structure of a treap is unique.

**Proof.**

# Properties of Treaps

**Theorem 2.** Given the pairs of keys and priorities, the structure of a treap is unique.

**Proof.**

- The node with lowest priority is in the root.

# Properties of Treaps

**Theorem 2.** Given the pairs of keys and priorities, the structure of a treap is unique.

**Proof.**

- The node with lowest priority is in the root.
- Due to the binary-search-tree property, all other nodes are uniquely assigned to the left or the right subtree of the root.

# Properties of Treaps

**Theorem 2.** Given the pairs of keys and priorities, the structure of a treap is unique.

**Proof.**

- The node with lowest priority is in the root.
- Due to the binary-search-tree property, all other nodes are uniquely assigned to the left or the right subtree of the root.
- Recursively apply this argument to the left and to the right subtree.

$\square$

# Properties of Treaps

Let $x_1, x_2, \ldots, x_n$ be the keys in a treap in increasing order.

# Properties of Treaps

Let $x_1, x_2, \ldots, x_n$ be the keys in a treap in increasing order.

Let $Y_{ij}$ be a random variable that is 1 if $x_j$ is an ancestor of $x_i$, and 0 otherwise.

# Properties of Treaps

Let $x_1, x_2, \ldots, x_n$ be the keys in a treap in increasing order.

Let $Y_{ij}$ be a random variable that is 1 if $x_j$ is an ancestor of $x_i$, and 0 otherwise.

**Lemma 3.** $E[Y_{ij}] = \frac{1}{|i-j|+1}$

# Properties of Treaps

Let $x_1, x_2, \ldots, x_n$ be the keys in a treap in increasing order.

Let $Y_{ij}$ be a random variable that is 1 if $x_j$ is an ancestor of $x_i$, and 0 otherwise.

**Lemma 3.** $E[Y_{ij}] = \frac{1}{|i-j|+1}$

**Proof.**

# Properties of Treaps

Let $x_1, x_2, \ldots, x_n$ be the keys in a treap in increasing order.

Let $Y_{ij}$ be a random variable that is 1 if $x_j$ is an ancestor of $x_i$, and 0 otherwise.

**Lemma 3.** $E[Y_{ij}] = \frac{1}{|i-j|+1}$

**Proof.**

- We assume that $i < j$; the other case is symmetric.

# Properties of Treaps

Let $x_1, x_2, \ldots, x_n$ be the keys in a treap in increasing order.

Let $Y_{ij}$ be a random variable that is 1 if $x_j$ is an ancestor of $x_i$, and 0 otherwise.

**Lemma 3.** $E[Y_{ij}] = \frac{1}{|i-j|+1}$

**Proof.**

- We assume that $i < j$; the other case is symmetric.
- Let $r$ be the lowest common ancestor of $x_i, x_{i+1}, \ldots, x_j$ and let $T_r$ be the subtree rooted at $r$. Clearly, $r$ has the lowest priority within $T_r$.

# Properties of Treaps

Let $x_1, x_2, \ldots, x_n$ be the keys in a treap in increasing order.

Let $Y_{ij}$ be a random variable that is 1 if $x_j$ is an ancestor of $x_i$, and 0 otherwise.

**Lemma 3.** $E[Y_{ij}] = \frac{1}{|i-j|+1}$

**Proof.**

- We assume that $i < j$; the other case is symmetric.
- Let $r$ be the lowest common ancestor of $x_i, x_{i+1}, \ldots, x_j$ and let $T_r$ be the subtree rooted at $r$. Clearly, $r$ has the lowest priority within $T_r$.
- $r \in \{x_i, x_{i+1}, \ldots, x_j\}$ as otherwise $r$ would not be the lowest common ancestor.

# Properties of Treaps

Let $x_1, x_2, \ldots, x_n$ be the keys in a treap in increasing order.

Let $Y_{ij}$ be a random variable that is 1 if $x_j$ is an ancestor of $x_i$, and 0 otherwise.

**Lemma 3.** $E[Y_{ij}] = \frac{1}{|i-j|+1}$

**Proof.**

- We assume that $i < j$; the other case is symmetric.
- Let $r$ be the lowest common ancestor of $x_i, x_{i+1}, \ldots, x_j$ and let $T_r$ be the subtree rooted at $r$. Clearly, $r$ has the lowest priority within $T_r$.
- $r \in \{x_i, x_{i+1}, \ldots, x_j\}$ as otherwise $r$ would not be the lowest common ancestor.
- If $r = x_j$, then $Y_{ij} = 1$; otherwise $x_j$ cannot be an ancestor of $x_i$ since $x_i$ and $x_j$ are in different subtrees of $r$ (or $r = x_i$).

# Properties of Treaps

Let $x_1, x_2, \ldots, x_n$ be the keys in a treap in increasing order.

Let $Y_{ij}$ be a random variable that is 1 if $x_j$ is an ancestor of $x_i$, and 0 otherwise.

**Lemma 3.** $E[Y_{ij}] = \frac{1}{|i-j|+1}$

**Proof.**

- We assume that $i < j$; the other case is symmetric.
- Let $r$ be the lowest common ancestor of $x_i, x_{i+1}, \ldots, x_j$ and let $T_r$ be the subtree rooted at $r$. Clearly, $r$ has the lowest priority within $T_r$.
- $r \in \{x_i, x_{i+1}, \ldots, x_j\}$ as otherwise $r$ would not be the lowest common ancestor.
- If $r = x_j$, then $Y_{ij} = 1$; otherwise $x_j$ cannot be an ancestor of $x_i$ since $x_i$ and $x_j$ are in different subtrees of $r$ (or $r = x_i$).
- Hence, $Y_{ij} = 1$ if and only if $x_j$ has the lowest priority among $\{x_i, x_{i+1}, \ldots, x_j\}$. Since all priorities are chosen uniformly at random, this has prob. $1/(j-i+1)$. $\square$

# Properties of Treaps

**Lemma 3.** $E[Y_{ij}] = \frac{1}{|i-j|+1}$

Let $x_1, x_2, \ldots, x_n$ be the keys in a treap in increasing order.

Let $Y_{ij}$ be a random variable that is 1 if $x_j$ is an ancestor of $x_i$, and 0 otherwise.

Let $Z_i$ be a random variable that denotes the number of vtcs. on path from root to $x_i$.

# Properties of Treaps

**Lemma 3.** $E[Y_{ij}] = \frac{1}{|i-j|+1}$

Let $x_1, x_2, \ldots, x_n$ be the keys in a treap in increasing order.

Let $Y_{ij}$ be a random variable that is 1 if $x_j$ is an ancestor of $x_i$, and 0 otherwise.

Let $Z_i$ be a random variable that denotes the number of vtcs. on path from root to $x_i$.

**Lemma 4.** $E[Z_i] = H_i + H_{n-i+1} - 1 \in O(\log n)$, where $H_k = \sum_{j=1}^{k} 1/k$.

$k$-th harmonic number

# Properties of Treaps

**Lemma 3.** $E[Y_{ij}] = \frac{1}{|i-j|+1}$

Let $x_1, x_2, \ldots, x_n$ be the keys in a treap in increasing order.

Let $Y_{ij}$ be a random variable that is 1 if $x_j$ is an ancestor of $x_i$, and 0 otherwise.

Let $Z_i$ be a random variable that denotes the number of vtcs. on path from root to $x_i$.

**Lemma 4.** $E[Z_i] = H_i + H_{n-i+1} - 1 \in O(\log n)$, where $H_k = \sum_{j=1}^{k} 1/k$.

$k$-th harmonic number

**Proof.**

# Properties of Treaps

**Lemma 3.** $E[Y_{ij}] = \frac{1}{|i-j|+1}$

Let $x_1, x_2, \ldots, x_n$ be the keys in a treap in increasing order.

Let $Y_{ij}$ be a random variable that is 1 if $x_j$ is an ancestor of $x_i$, and 0 otherwise.

Let $Z_i$ be a random variable that denotes the number of vtcs. on path from root to $x_i$.

**Lemma 4.** $E[Z_i] = H_i + H_{n-i+1} - 1 \in O(\log n)$, where $H_k = \sum_{j=1}^{k} 1/k$.

$k$-th harmonic number

**Proof.**

- On the path from the root to $x_i$, there are precisely the ancestors of $x_i$ and $x_i$ itself.

# Properties of Treaps

**Lemma 3.** $E[Y_{ij}] = \frac{1}{|i-j|+1}$

Let $x_1, x_2, \ldots, x_n$ be the keys in a treap in increasing order.

Let $Y_{ij}$ be a random variable that is 1 if $x_j$ is an ancestor of $x_i$, and 0 otherwise.

Let $Z_i$ be a random variable that denotes the number of vtcs. on path from root to $x_i$.

**Lemma 4.** $E[Z_i] = H_i + H_{n-i+1} - 1 \in O(\log n)$, where $H_k = \sum_{j=1}^{k} 1/k$.

$k$-th harmonic number

**Proof.**

- On the path from the root to $x_i$, there are precisely the ancestors of $x_i$ and $x_i$ itself.
- Hence, $Z_i = \sum_{j=1}^{n} Y_{ij}$, where $Y_{ii} = 1$.

# Properties of Treaps

**Lemma 3.** $E[Y_{ij}] = \frac{1}{|i-j|+1}$

Let $x_1, x_2, \ldots, x_n$ be the keys in a treap in increasing order.

Let $Y_{ij}$ be a random variable that is 1 if $x_j$ is an ancestor of $x_i$, and 0 otherwise.

Let $Z_i$ be a random variable that denotes the number of vtcs. on path from root to $x_i$.

**Lemma 4.** $E[Z_i] = H_i + H_{n-i+1} - 1 \in O(\log n)$, where $H_k = \sum_{j=1}^{k} 1/k$.

$k$-th harmonic number

**Proof.**

- On the path from the root to $x_i$, there are precisely the ancestors of $x_i$ and $x_i$ itself.
- Hence, $Z_i = \sum_{j=1}^{n} Y_{ij}$, where $Y_{ii} = 1$.
- $E[Z_i] =$

# Properties of Treaps

**Lemma 3.** $E[Y_{ij}] = \frac{1}{|i-j|+1}$

Let $x_1, x_2, \ldots, x_n$ be the keys in a treap in increasing order.

Let $Y_{ij}$ be a random variable that is 1 if $x_j$ is an ancestor of $x_i$, and 0 otherwise.

Let $Z_i$ be a random variable that denotes the number of vtcs. on path from root to $x_i$.

**Lemma 4.** $E[Z_i] = H_i + H_{n-i+1} - 1 \in O(\log n)$, where $H_k = \sum_{j=1}^{k} 1/k$.

$k$-th harmonic number

**Proof.**

- On the path from the root to $x_i$, there are precisely the ancestors of $x_i$ and $x_i$ itself.
- Hence, $Z_i = \sum_{j=1}^{n} Y_{ij}$, where $Y_{ii} = 1$.
- $E[Z_i] = E\left[\sum_{j=1}^{n} Y_{ij}\right] =$

# Properties of Treaps

**Lemma 3.** $E[Y_{ij}] = \frac{1}{|i-j|+1}$

Let $x_1, x_2, \ldots, x_n$ be the keys in a treap in increasing order.

Let $Y_{ij}$ be a random variable that is 1 if $x_j$ is an ancestor of $x_i$, and 0 otherwise.

Let $Z_i$ be a random variable that denotes the number of vtcs. on path from root to $x_i$.

**Lemma 4.** $E[Z_i] = H_i + H_{n-i+1} - 1 \in O(\log n)$, where $H_k = \sum_{j=1}^{k} 1/k$.

$k$-th harmonic number

**Proof.**

- On the path from the root to $x_i$, there are precisely the ancestors of $x_i$ and $x_i$ itself.
- Hence, $Z_i = \sum_{j=1}^{n} Y_{ij}$, where $Y_{ii} = 1$.
- $E[Z_i] = E\left[\sum_{j=1}^{n} Y_{ij}\right] = \sum_{j=1}^{n} E[Y_{ij}] =$

# Properties of Treaps

**Lemma 3.** $E[Y_{ij}] = \frac{1}{|i-j|+1}$

Let $x_1, x_2, \ldots, x_n$ be the keys in a treap in increasing order.

Let $Y_{ij}$ be a random variable that is 1 if $x_j$ is an ancestor of $x_i$, and 0 otherwise.

Let $Z_i$ be a random variable that denotes the number of vtcs. on path from root to $x_i$.

**Lemma 4.** $E[Z_i] = H_i + H_{n-i+1} - 1 \in O(\log n)$, where $H_k = \sum_{j=1}^{k} 1/k$.

$k$-th harmonic number

**Proof.**

- On the path from the root to $x_i$, there are precisely the ancestors of $x_i$ and $x_i$ itself.
- Hence, $Z_i = \sum_{j=1}^{n} Y_{ij}$, where $Y_{ii} = 1$.
- $E[Z_i] = E\left[\sum_{j=1}^{n} Y_{ij}\right] = \sum_{j=1}^{n} E[Y_{ij}] = \sum_{j=1}^{n} \frac{1}{|i-j|+1} =$

# Properties of Treaps

**Lemma 3.** $E[Y_{ij}] = \frac{1}{|i-j|+1}$

Let $x_1, x_2, \ldots, x_n$ be the keys in a treap in increasing order.

Let $Y_{ij}$ be a random variable that is 1 if $x_j$ is an ancestor of $x_i$, and 0 otherwise.

Let $Z_i$ be a random variable that denotes the number of vtcs. on path from root to $x_i$.

**Lemma 4.** $E[Z_i] = H_i + H_{n-i+1} - 1 \in O(\log n)$, where $H_k = \sum_{j=1}^{k} 1/k$.

$k$-th harmonic number

**Proof.**

- On the path from the root to $x_i$, there are precisely the ancestors of $x_i$ and $x_i$ itself.
- Hence, $Z_i = \sum_{j=1}^{n} Y_{ij}$, where $Y_{ii} = 1$.
- $E[Z_i] = E\left[\sum_{j=1}^{n} Y_{ij}\right] = \sum_{j=1}^{n} E[Y_{ij}] = \sum_{j=1}^{n} \frac{1}{|i-j|+1} = \sum_{j=2}^{i} \frac{1}{j} + \sum_{j=2}^{n-i+1} \frac{1}{j} + 1$

# Properties of Treaps

**Lemma 3.** $E[Y_{ij}] = \frac{1}{|i-j|+1}$

Let $x_1, x_2, \dots, x_n$ be the keys in a treap in increasing order.

Let $Y_{ij}$ be a random variable that is 1 if $x_j$ is an ancestor of $x_i$, and 0 otherwise.

Let $Z_i$ be a random variable that denotes the number of vtcs. on path from root to $x_i$.

**Lemma 4.** $E[Z_i] = H_i + H_{n-i+1} - 1 \in O(\log n)$, where $H_k = \sum_{j=1}^{k} 1/k$.

$k$-th harmonic number

**Proof.**

- On the path from the root to $x_i$, there are precisely the ancestors of $x_i$ and $x_i$ itself.
- Hence, $Z_i = \sum_{j=1}^{n} Y_{ij}$, where $Y_{ii} = 1$.
- $E[Z_i] = E\left[\sum_{j=1}^{n} Y_{ij}\right] = \sum_{j=1}^{n} E[Y_{ij}] = \sum_{j=1}^{n} \frac{1}{|i-j|+1} = \sum_{j=2}^{i} \frac{1}{j} + \sum_{j=2}^{n-i+1} \frac{1}{j} + 1$

$= H_i + H_{n-i+1} - 1$ $\square$

# Properties of Treaps

Let $x_1, x_2, \ldots, x_n$ be the keys in a treap in increasing order.

Let $Y_{ij}$ be a random variable that is 1 if $x_j$ is an ancestor of $x_i$, and 0 otherwise.

Let $Z_i$ be a random variable that denotes the number of vtcs. on path from root to $x_i$.

**Lemma 4.** $E[Z_i] = H_i + H_{n-i+1} - 1 \in O(\log n)$, where $H_k = \sum_{j=1}^{k} 1/k$.

$k$-th harmonic number

**Theorem 5.** Searching, inserting, and deleting a key in a treap 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 ...

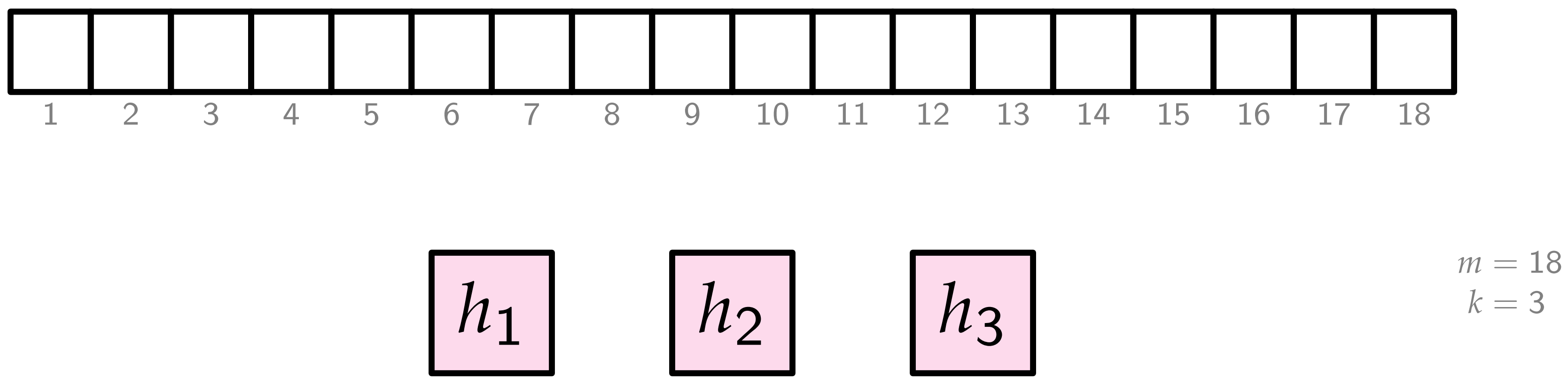
- array or linked list:
  - $(-)$ $\Theta(n)$ time for containment check ($\Theta(\log n)$ for a sorted array)
  - $(\circ)$ simple data structure with low space consumption
  - $(-)$ adding/removing keys takes $\Theta(n)$ time
- hash table:
  - $(+)$ usually $\Theta(1)$ time for containment check
  - $(-)$ 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
- 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

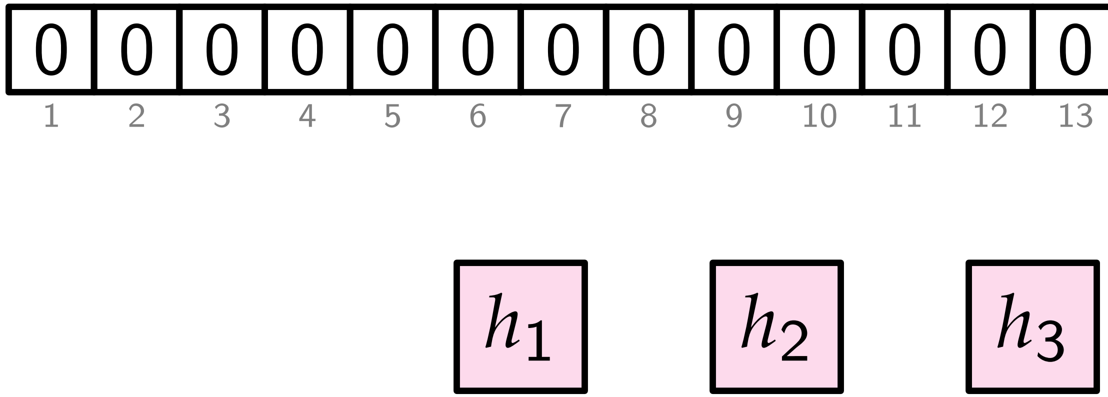
A Bloom filter is a **bit array** of $m$ bits & a set of $k$ different **hash functions** $h_1, \ldots, h_k$.
Each hash function $h_i$ generates a uniform random distribution in the range $\{1, \ldots, m\}$.



# 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.



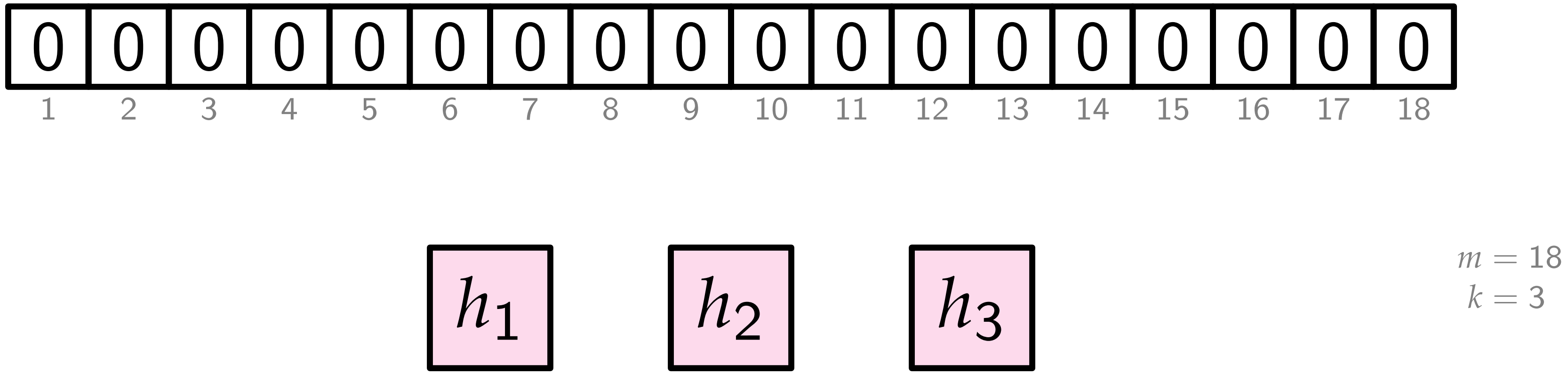

$m = 18$
$k = 3$

# 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 keys, 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.

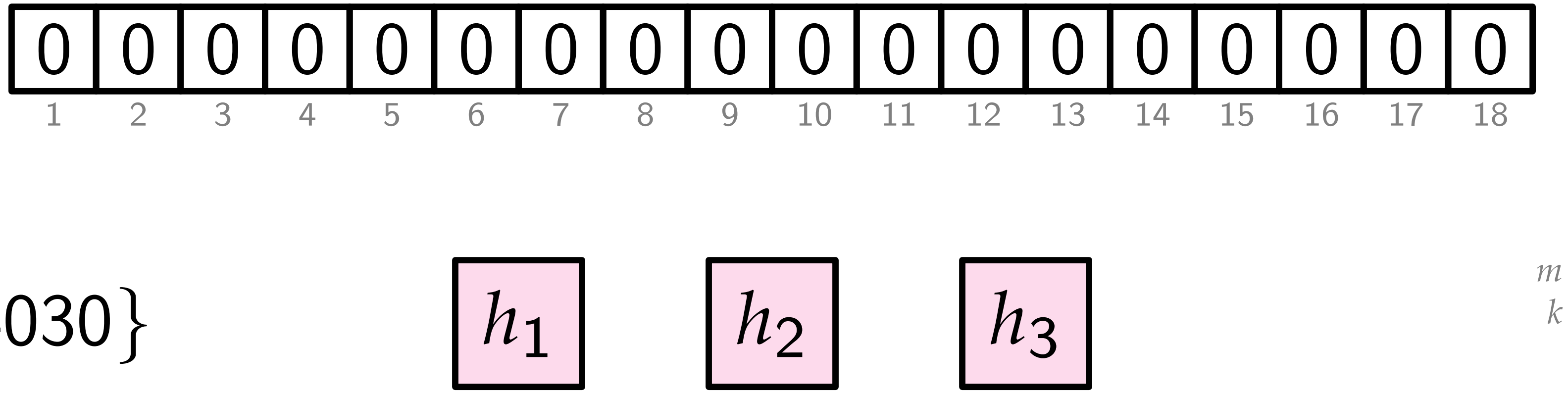


# Bloom Filters

A Bloom filter is a **bit array** of $m$ bits & a set of $k$ different **hash functions** $h_1, \ldots, h_k$.
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 keys, 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.




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

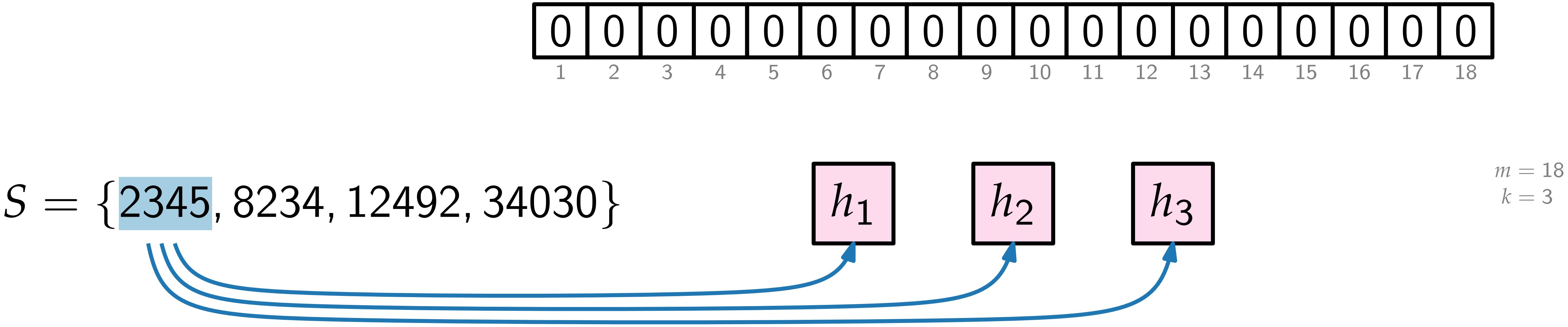
$m = 18$
$k = 3$

# 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 keys, 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.

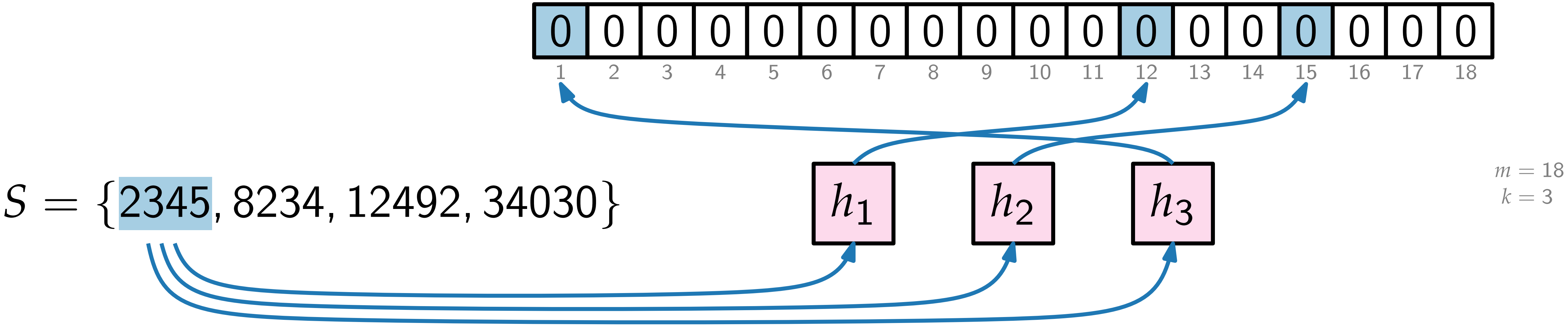


# Bloom Filters

A Bloom filter is a **bit array** of $m$ bits & a set of $k$ different **hash functions** $h_1, \ldots, h_k$.
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 keys, 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.

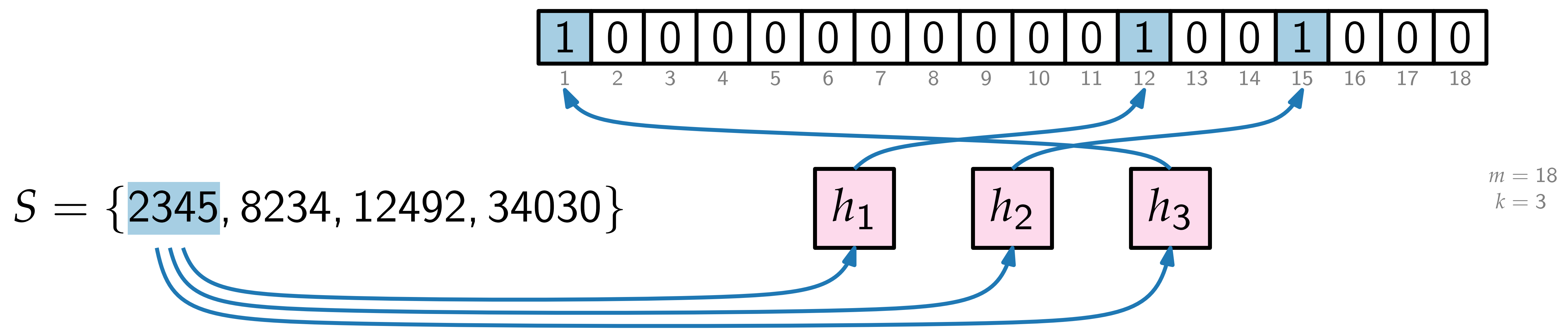


# Bloom Filters

A Bloom filter is a **bit array** of $m$ bits & a set of $k$ different **hash functions** $h_1, \ldots, h_k$.
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 keys, 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.

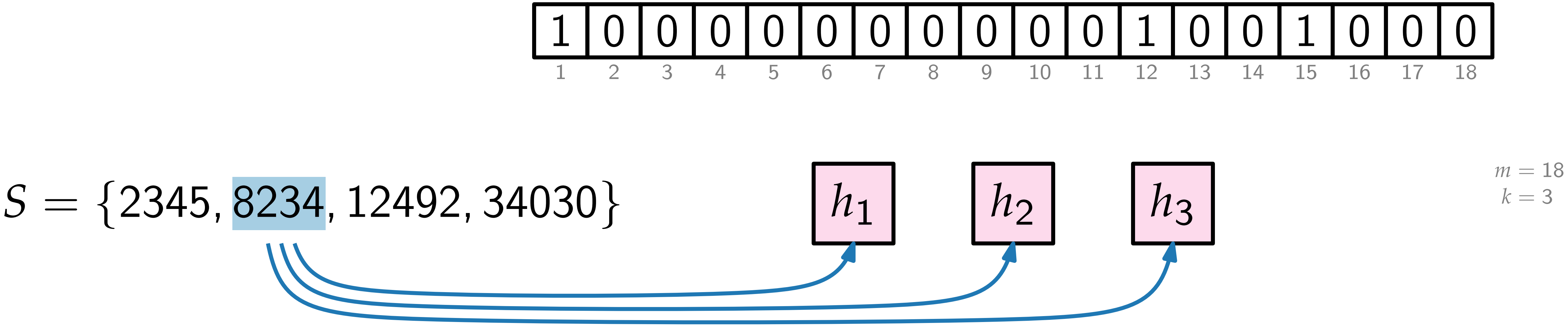


# Bloom Filters

A Bloom filter is a **bit array** of $m$ bits & a set of $k$ different **hash functions** $h_1, \ldots, h_k$. 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 keys, 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.

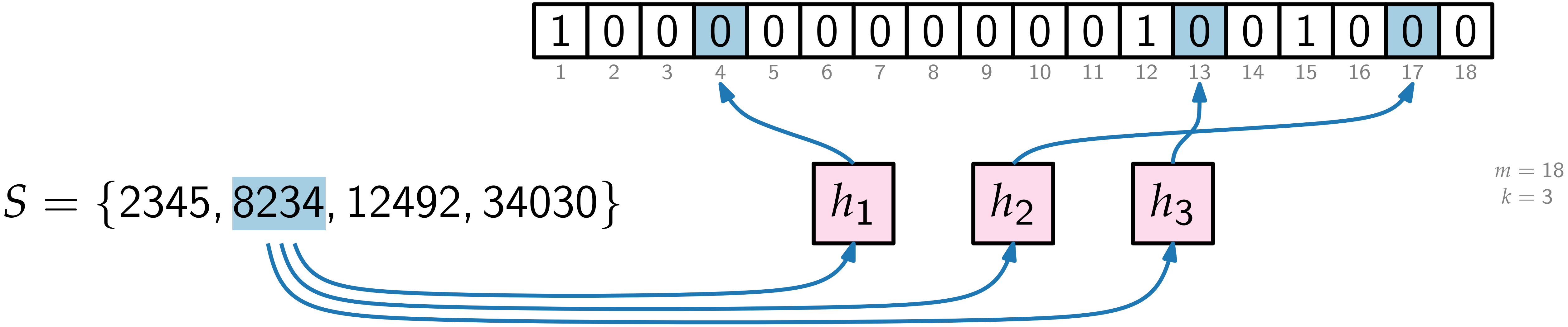


# Bloom Filters

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 keys, 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.

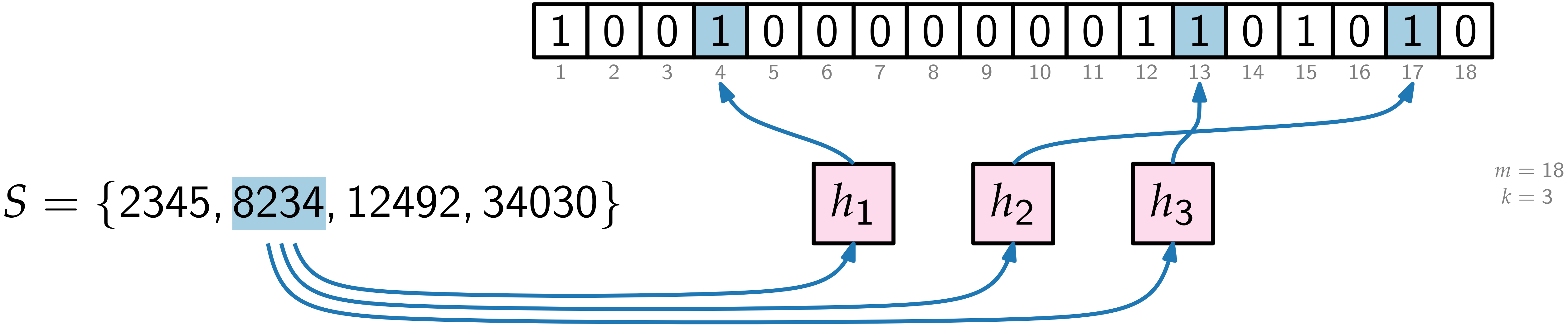


# Bloom Filters

A Bloom filter is a **bit array** of $m$ bits & a set of $k$ different **hash functions** $h_1, \ldots, h_k$. 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 keys, 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

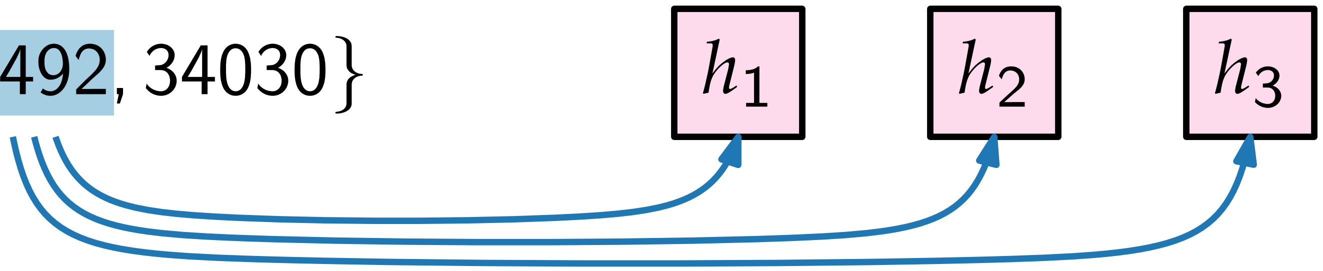
A Bloom filter is a **bit array** of $m$ bits & a set of $k$ different **hash functions** $h_1, \ldots, h_k$. Each hash function $h_i$ generates a uniform random distribution in the range $\{1, \ldots, m\}$.

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

For a set $S$ of keys, 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 |

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



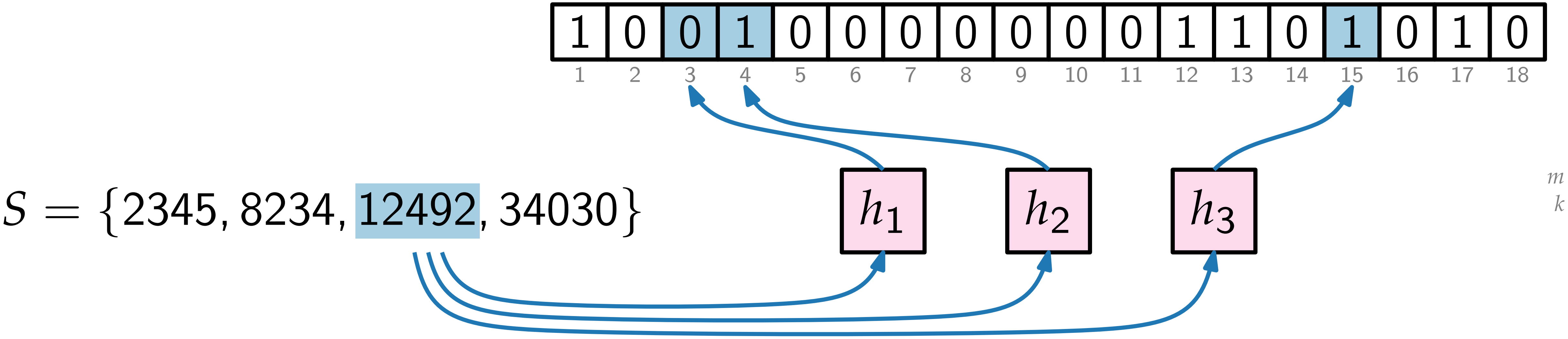

$m = 18$
$k = 3$

# 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 keys, 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.



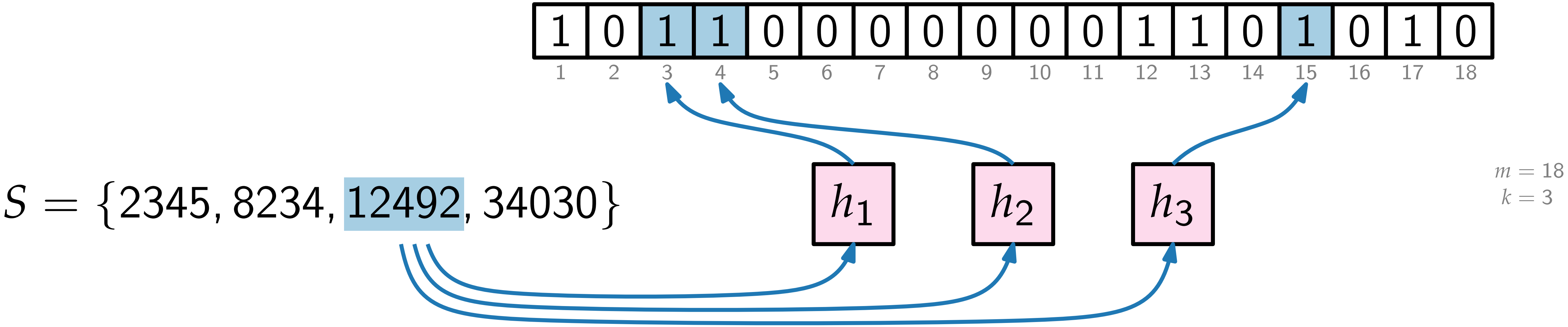

$m = 18$
$k = 3$

# 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 keys, 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** $h_1, \ldots, h_k$. 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 keys, 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 |

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

$h_1$ $h_2$ $h_3$

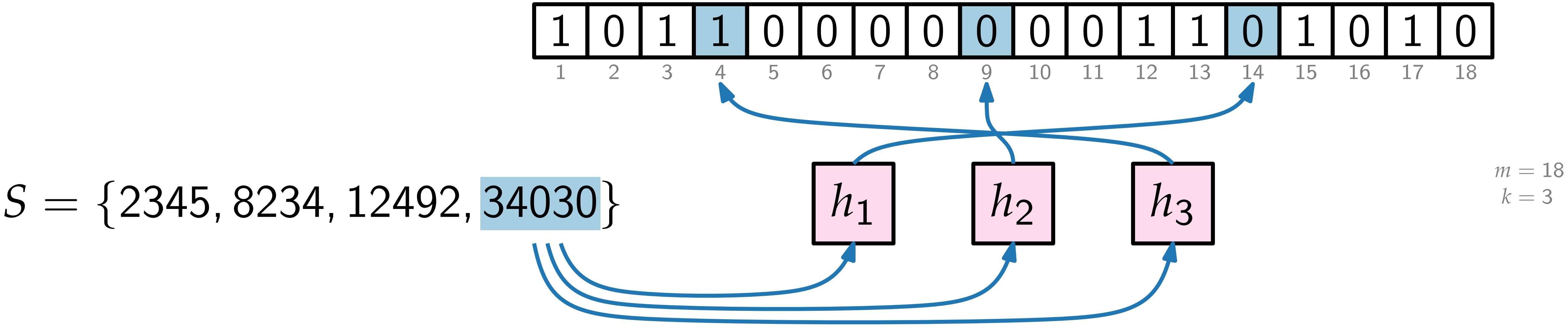
$m = 18$
$k = 3$

# 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 keys, 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.

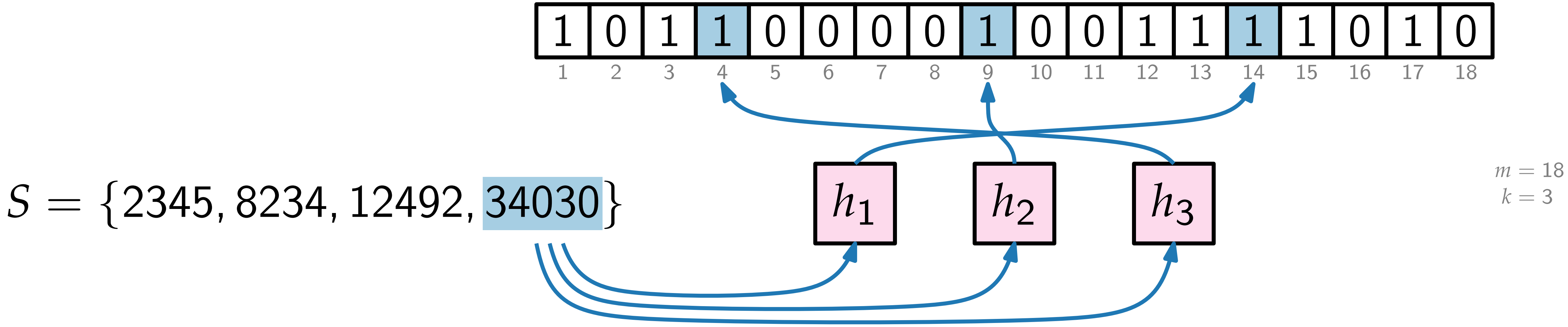


# Bloom Filters

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 keys, 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 keys, 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 |

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

$\boxed{h_1}$ $\boxed{h_2}$ $\boxed{h_3}$

$m = 18$
$k = 3$

# 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 keys, 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 |

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

$h_1$ $h_2$ $h_3$

$m = 18$
$k = 3$

# 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 keys, 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 $h_1(a), h_2(a), \ldots, h_k(a)$:

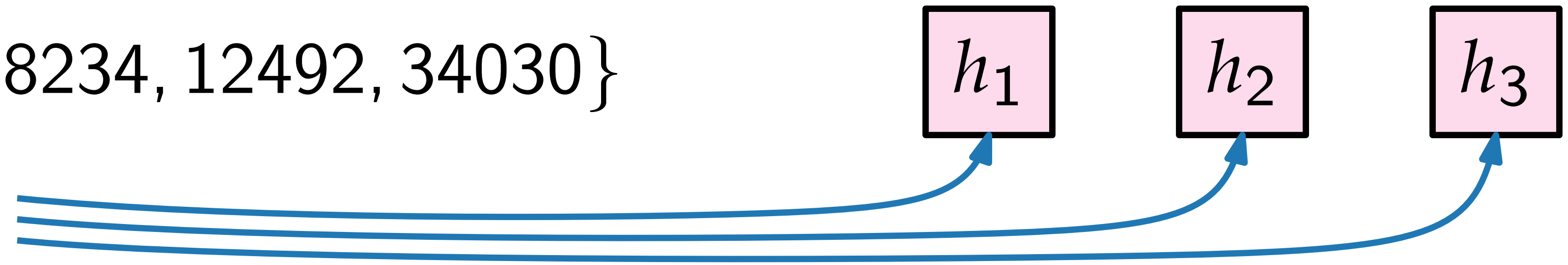
- if there is a 0, $a$ is for sure not in $S$.
- if there are only 1s, $a$ may be in $S$ (it is in $S$ with a small error probability).

| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 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

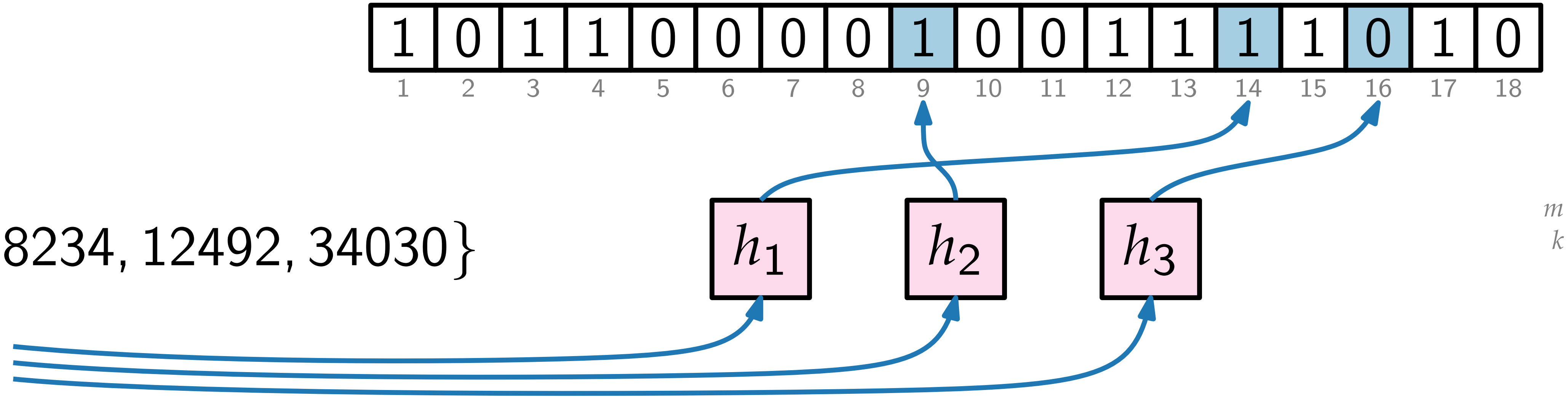
A Bloom filter is a **bit array** of $m$ bits & a set of $k$ different **hash functions** $h_1, \dots, h_k$. Each hash function $h_i$ generates a uniform random distribution in the range $\{1, \dots, m\}$.

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

For a set $S$ of keys, 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).




$m = 18$
$k = 3$

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

$a = 2346$

# Bloom Filters

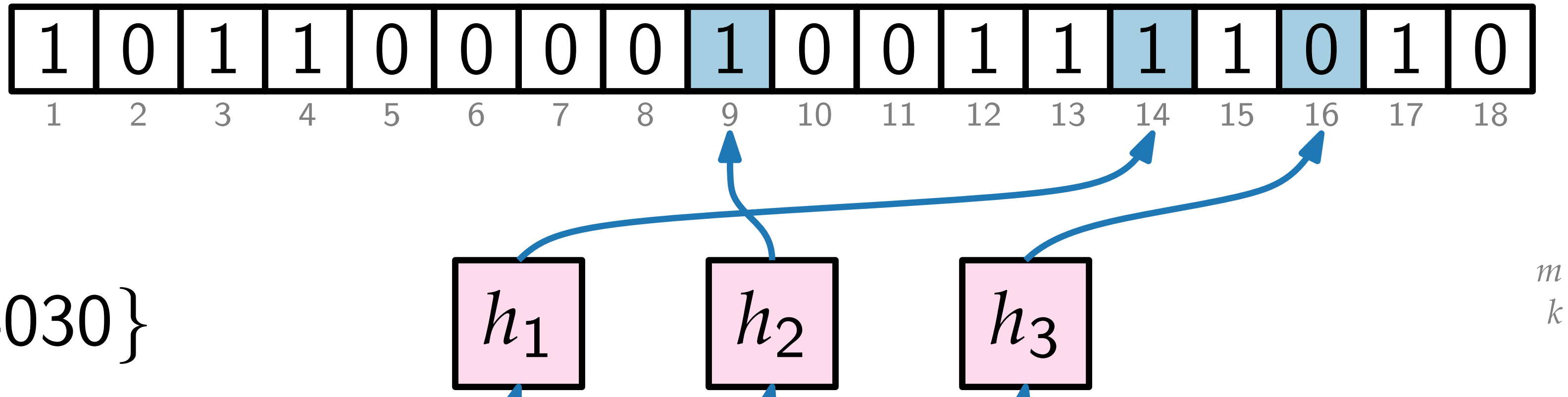
A Bloom filter is a **bit array** of $m$ bits & a set of $k$ different **hash functions** $h_1, \dots, h_k$. Each hash function $h_i$ generates a uniform random distribution in the range $\{1, \dots, m\}$.

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

For a set $S$ of keys, 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).




$m = 18$
$k = 3$

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

$a = 2346$ (correct) ✗

# 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 keys, 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 $h_1(a), h_2(a), \dots, h_k(a)$:

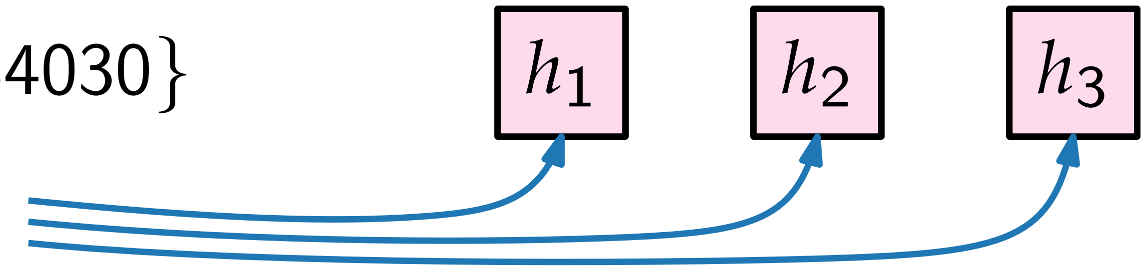
- if there is a 0, $a$ is for sure not in $S$.
- if there are only 1s, $a$ may be in $S$ (it is in $S$ with a small error probability).

| 1 | 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) ✗ $\quad b = 8234$



# Bloom Filters

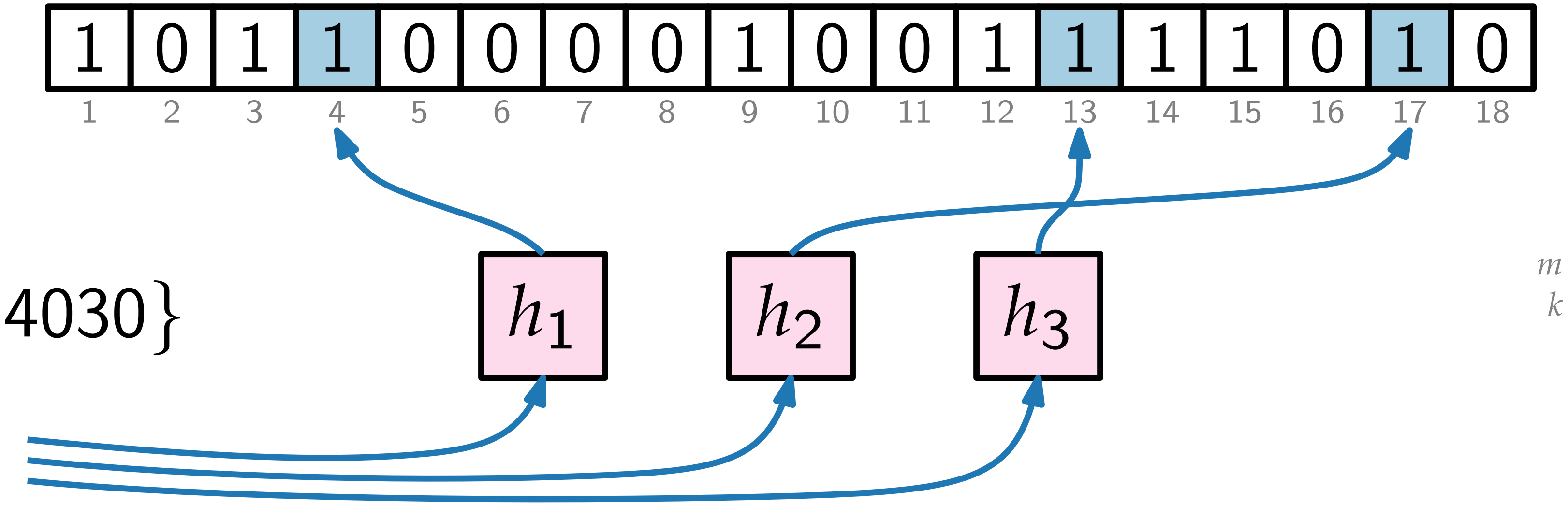
A Bloom filter is a **bit array** of $m$ bits & a set of $k$ different **hash functions** $h_1, \ldots, h_k$.
Each hash function $h_i$ generates a uniform random distribution in the range $\{1, \ldots, m\}$.

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

For a set $S$ of keys, 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).




$m = 18$
$k = 3$

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

$a = 2346$ ✗ (correct) $b = 8234$

# Bloom Filters

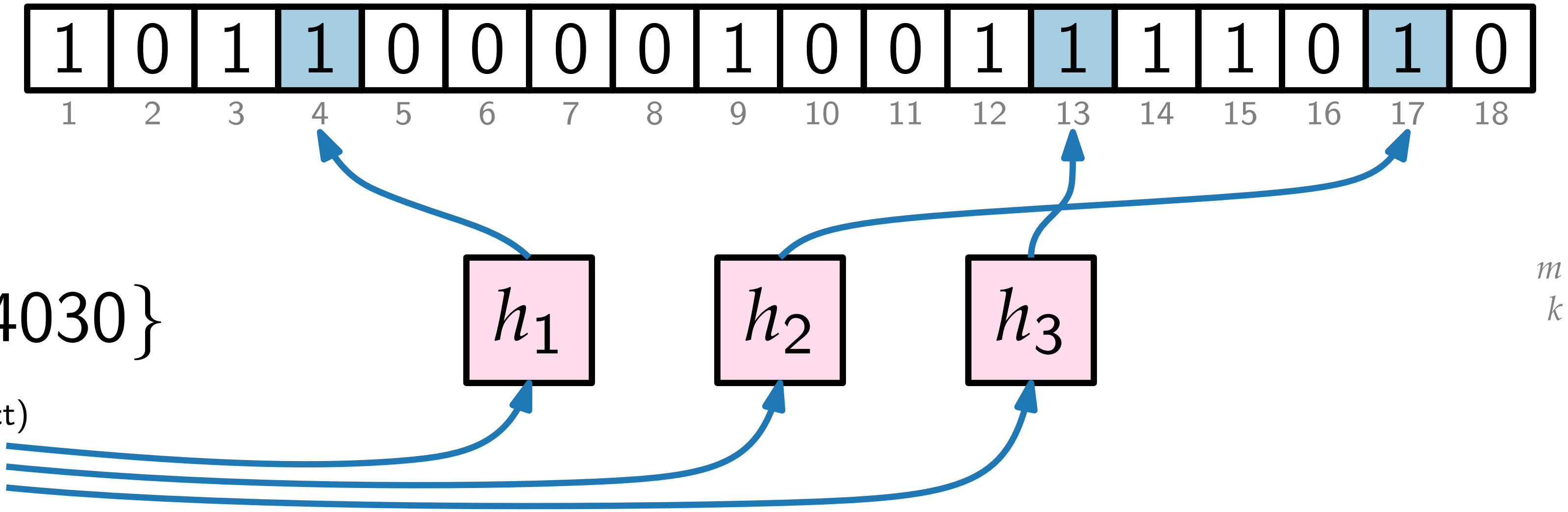
A Bloom filter is a **bit array** of $m$ bits & a set of $k$ different **hash functions** $h_1, \ldots, h_k$.
Each hash function $h_i$ generates a uniform random distribution in the range $\{1, \ldots, m\}$.

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

For a set $S$ of keys, 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).




$m = 18$
$k = 3$

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

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

# 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 keys, 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 $h_1(a), h_2(a), \ldots, h_k(a)$:

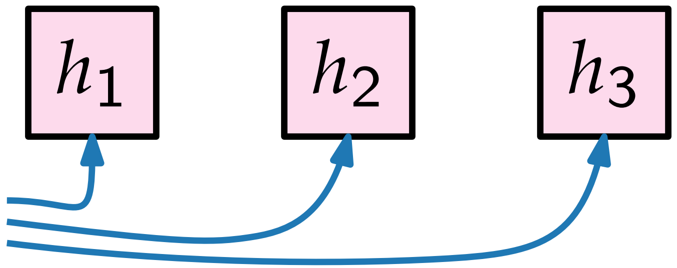
- if there is a 0, $a$ is for sure not in $S$.
- if there are only 1s, $a$ may be in $S$ (it is in $S$ with a small error probability).

| 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

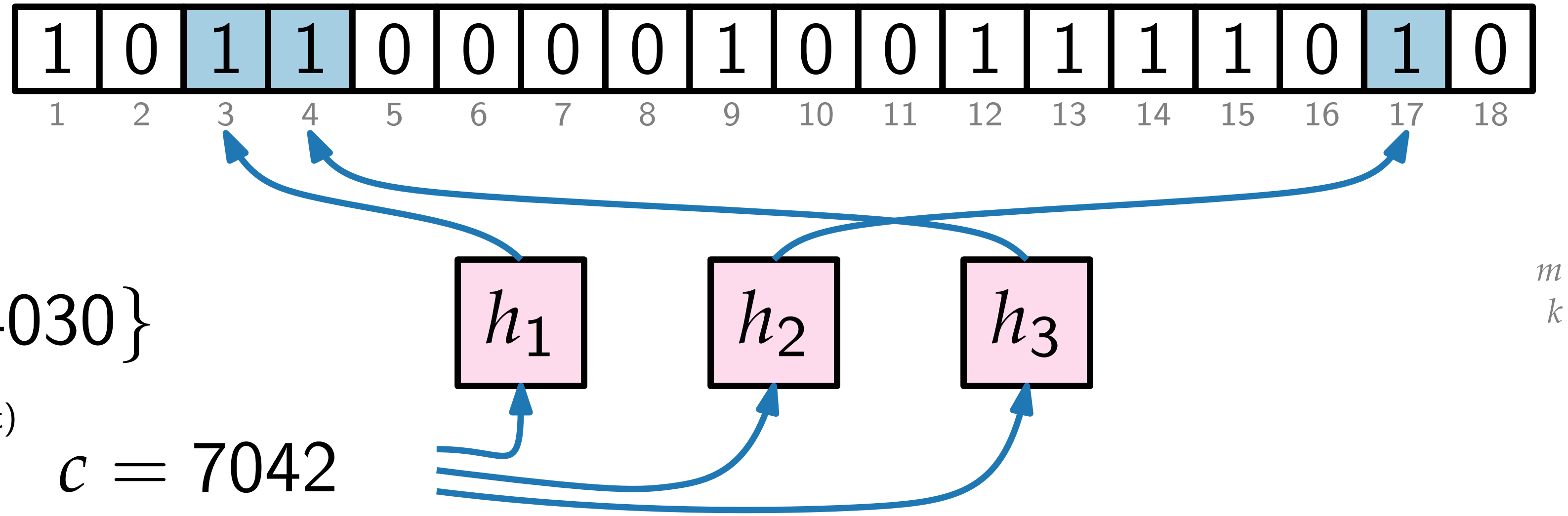
A Bloom filter is a **bit array** of $m$ bits & a set of $k$ different **hash functions** $h_1, \dots, h_k$.
Each hash function $h_i$ generates a uniform random distribution in the range $\{1, \dots, m\}$.

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

For a set $S$ of keys, 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).




$m = 18$
$k = 3$

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

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

# Bloom Filters

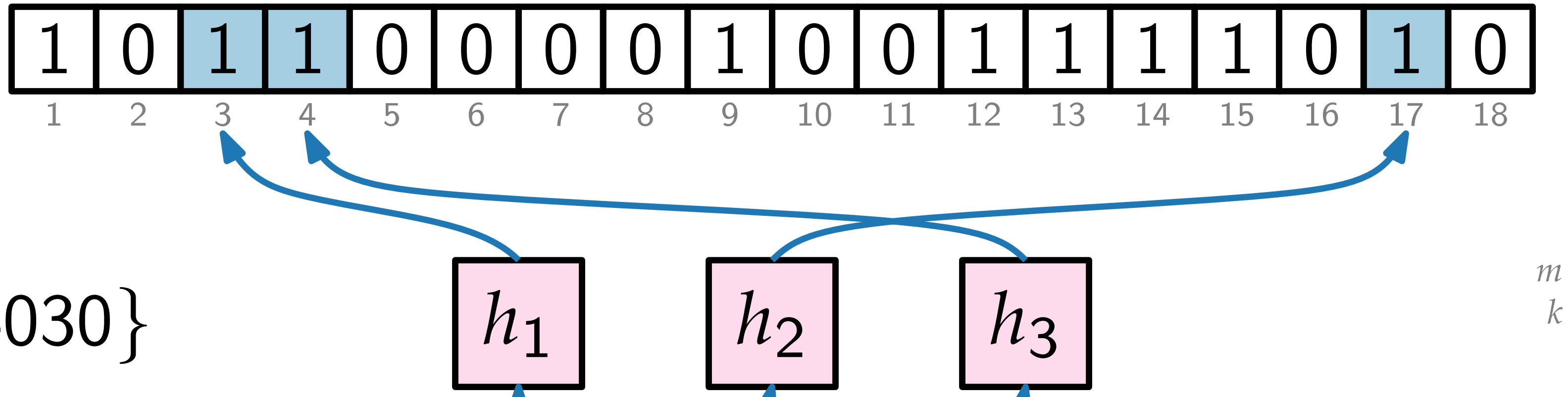
A Bloom filter is a **bit array** of $m$ bits & a set of $k$ different **hash functions** $h_1, \ldots, h_k$.
Each hash function $h_i$ generates a uniform random distribution in the range $\{1, \ldots, m\}$.

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

For a set $S$ of keys, 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).




$m = 18$
$k = 3$

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

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

# Error Probability of Bloom Filters

We start with an empty array and insert a key $s$.

# Error Probability of Bloom Filters

We start with an empty array and insert a key $s$.

- 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 key $s$.

- 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 key $s$.

- 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 key $s$.

- 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 $n$ keys, 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 key $s$.

- 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 $n$ keys, 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 key $s$.

- 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 $n$ keys, 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 $\varepsilon$ 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 $\varepsilon$ 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 $\varepsilon$ 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 $\varepsilon$ 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 key in our set is $\frac{m}{n} \approx -\frac{\log_2 \varepsilon}{\ln 2} \approx -1.44 \log_2 \varepsilon$.

# 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 $\varepsilon$ 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 key in our set is $\frac{m}{n} \approx -\frac{\log_2 \varepsilon}{\ln 2} \approx -1.44 \log_2 \varepsilon$.

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 $\varepsilon$ 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 key in our set is $\frac{m}{n} \approx -\frac{\log_2 \varepsilon}{\ln 2} \approx -1.44 \log_2 \varepsilon$.

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

| |
|---|
| error probabiltiy $\varepsilon$ |
| bits per key $\frac{m}{n}$ |
| # 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 $\varepsilon$ 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 key in our set is $\frac{m}{n} \approx -\frac{\log_2 \varepsilon}{\ln 2} \approx -1.44 \log_2 \varepsilon$.

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

10 %

| | |
|---|---|
| error probabiltiy $\varepsilon$ | 0.1 |
| bits per key $\frac{m}{n}$ | 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 $\varepsilon$ 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 key in our set is $\frac{m}{n} \approx -\frac{\log_2 \varepsilon}{\ln 2} \approx -1.44 \log_2 \varepsilon$.

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

| error probabiltiy $\varepsilon$ | 0.1 | 0.01 (1 %) |
|---|---|---|
| bits per key $\frac{m}{n}$ | 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 $\varepsilon$ 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 key in our set is $\frac{m}{n} \approx -\frac{\log_2 \varepsilon}{\ln 2} \approx -1.44 \log_2 \varepsilon$.

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

1 ‰

| error probabiltiy $\varepsilon$ | 0.1 | 0.01 | 0.001 |
|---|---|---|---|
| bits per key $\frac{m}{n}$ | 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 $\varepsilon$ 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 key in our set is $\frac{m}{n} \approx -\frac{\log_2 \varepsilon}{\ln 2} \approx -1.44 \log_2 \varepsilon$.

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

1 in a million ($10^6$)

| error probabiltiy $\varepsilon$ | 0.1 | 0.01 | 0.001 | 0.000001 |
|---|---|---|---|---|
| bits per key $\frac{m}{n}$ | 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 $\varepsilon$ 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 key in our set is $\frac{m}{n} \approx -\frac{\log_2 \varepsilon}{\ln 2} \approx -1.44 \log_2 \varepsilon$.

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

1 in a billion ($10^9$)

| error probabiltiy $\varepsilon$ | 0.1 | 0.01 | 0.001 | 0.000001 | 0.000000001 |
|---|---|---|---|---|---|
| bits per key $\frac{m}{n}$ | 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 $\varepsilon$ 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 key in our set is $\frac{m}{n} \approx -\frac{\log_2 \varepsilon}{\ln 2} \approx -1.44 \log_2 \varepsilon$.

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

1 in a trillion ($10^{12}$)

| error probabiltiy $\varepsilon$ | 0.1 | 0.01 | 0.001 | 0.000001 | 0.000000001 | 0.000000000001 |
|---|---|---|---|---|---|---|
| bits per key $\frac{m}{n}$ | 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 $\varepsilon$ 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 key in our set is $\frac{m}{n} \approx -\frac{\log_2 \varepsilon}{\ln 2} \approx -1.44 \log_2 \varepsilon$.

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

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

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

# Discussion of Skip Lists and Treaps

- (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 of Skip Lists and Treaps

- (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 of Skip Lists and Treaps

- (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.
- Similarly, treaps are a simpler alternative to deterministic binary search trees.

# Discussion of Skip Lists and Treaps

- (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.
- Similarly, treaps are a simpler alternative to deterministic binary search trees.
- Beside treaps, there is a random. data structure called *randomized binary search tree*.

# Discussion of Skip Lists and Treaps

- (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.
- Similarly, treaps are a simpler alternative to deterministic binary search trees.
- Beside treaps, there is a random. data structure called *randomized binary search tree*.
- A randomized binary search tree does not store priorities, instead, when a key is inserted, it replaces with probability $1/(n+1)$ the current root; if it does not replace the root, this process is repeated in the corresponding subtree.

# Discussion of Skip Lists and Treaps

- (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.
- Similarly, treaps are a simpler alternative to deterministic binary search trees.
- Beside treaps, there is a random. data structure called *randomized binary search tree*.
- A randomized binary search tree does not store priorities, instead, when a key is inserted, it replaces with probability $1/(n+1)$ the current root; if it does not replace the root, this process is repeated in the corresponding subtree.
- Hence, a randomized binary search tree stores less information (the size of the subtree instead of a priority) and there is no risk of a collision between priorities. However, there are more requests to the random number generator.

# Discussion of Bloom Filters

- 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 can be tolerated.

# Discussion of Bloom Filters

- 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 can be tolerated.
- There are some refinements of classical Bloom filters to overcome some disadvantages, e.g., *counting Bloom filters*:
  - Instead of $m$ bits, $m$ counters are stored.
  - To insert a key, the counters determined by the hash functions are incremented.
  - To remove a key, the counters determined by the hash functions are decremented.

# Discussion of Bloom Filters

- 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 can be tolerated.
- There are some refinements of classical Bloom filters to overcome some disadvantages, e.g., *counting Bloom filters*:
  - Instead of $m$ bits, $m$ counters are stored.
  - To insert a key, the counters determined by the hash functions are incremented.
  - To remove a key, the counters determined by the hash functions are decremented.
- 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)
  - hiding real data, while indicating if an object is in a set (e.g., database of criminals)

# Literature

## Skip lists:


- [Pug '90] William W. Pugh, "Skip Lists: A Probabilistic Alternative to Balanced Trees" in *Communications of the ACM*, 33(6):668–676, 1990 (preliminary version: WADS 1989)
- [Sto '17] Sabine Storandt's lecture script "Randomized Algorithms" (2016–2017)


## Treaps:


- [SA '96] Raimund Seidel, Cecilia R. Aragon, "Randomized Search Trees" in *Algorithmica*, 16(4/5):464–497, 1996 (preliminary version: FOCS 1989)


## Bloom filters:


- [Blo '70] Burton H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors" in *Communications of the ACM*, 13(7):422–426, 1970
- [MU '05] Michael Mitzenmacher and Eli Upfal, "Probability and Computing: Randomized Algorithms and Probabilistic Analysis", Cambridge University Press, 2005
- [BGK+ '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