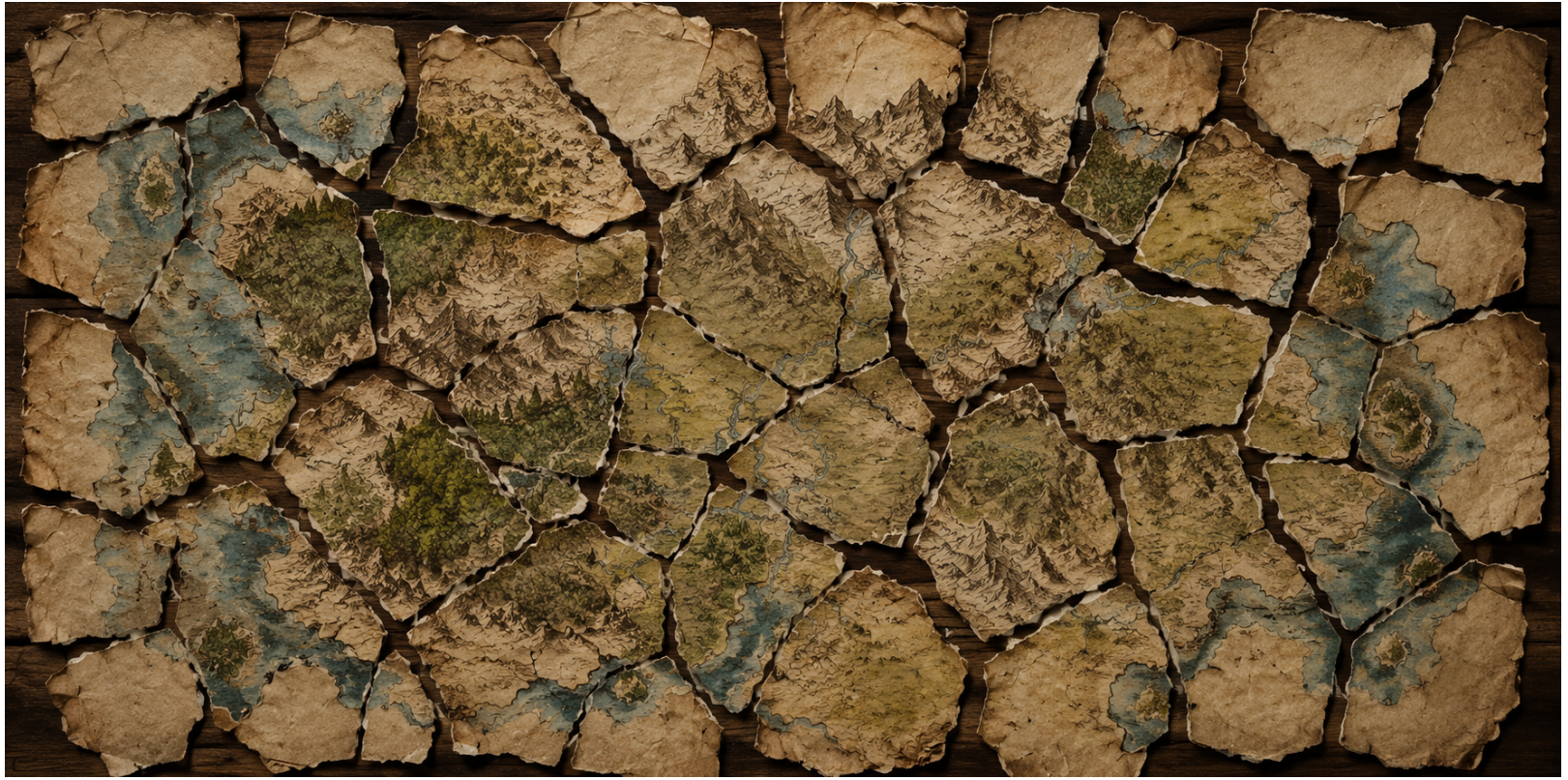
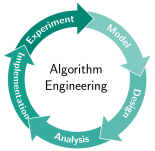


Hash Tables II

Stefan Walzer, Ragnar Groot Koerkamp, Stefan Hermann | compiled on 27. Juli 2026



Previously

Technique	Avg. Positive Query	Negative Query	Insertion	Bits per Key[1]	Comment
Chaining	$\mathcal{O}(1)$ expected	$1+\alpha$ expected	$=$ neg. query [2]	$p/\alpha + w + p$	Stable
Cuckoo Hashing	$\leqslant 2$	2	$\mathcal{O}(1)$ expected	w/α	worst-case $\mathcal{O}(1)$ queries!
Linear Probing	$\frac{1}{2}(1+\frac{1}{1-\alpha})$ expected	$\frac{1}{2}(1+\frac{1}{(1-\alpha)^2})$ expected	$=$ neg. query	w/α	cache friendliness
Quadratic Probing	$\mathcal{O}(\log \frac{1}{1-\alpha})$ [3]	$\mathcal{O}(\frac{1}{1-\alpha})$ [3]	$\mathcal{O}(\frac{1}{1-\alpha})$ [3]	w/α	current research (for decades)

[1] w bits per element
[2] even in the worst-case if we can assume the key does not exist yet then we can insert at the beginning of the list
[3] Conjectured

Previously

Technique	Avg. Positive Query	Negative Query	Insertion	Bits per Key[1]	Comment
Chaining	$\mathcal{O}(1)$ expected	$1+\alpha$ expected	= neg. query [2]	$p/\alpha + w + p$	Stable
Cuckoo Hashing	$\leqslant 2$	2	$\mathcal{O}(1)$ expected	w/α	worst-case $\mathcal{O}(1)$ queries!
Linear Probing	$\frac{1}{2}(1+\frac{1}{1-\alpha})$ expected	$\frac{1}{2}(1+\frac{1}{(1-\alpha)^2})$ expected	= neg. query	w/α	cache friendliness
Quadratic Probing	$\mathcal{O}(\log \frac{1}{1-\alpha})$ [3]	$\mathcal{O}(\frac{1}{1-\alpha})$ [3]	$\mathcal{O}(\frac{1}{1-\alpha})$ [3]	w/α	current research (for decades)
Today	$\mathcal{O}(1)$ worst-case	$\mathcal{O}(1)$ worst-case	$\mathcal{O}(1)$ whp [4]	$(1+o(1))w$	strong simultaneous guarantees

[1] w bits per element
[2] even in the worst-case if we can assume the key does not exist yet then we can insert at the beginning of the list
[3] Conjectured
[4] with high probability (whp) defined later

This lecture is based on slides by

Martín Farach-Colton
Rutgers University, USA

Michael Bender
Stony Brook University, USA

John Kuszmaul
Yale University, USA

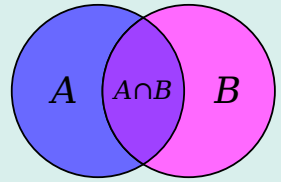
William Kuszmaul
MIT, USA

For details, refer to the paper
Bender, M. A., Farach-Colton, M., Kuszmaul, J., & Kuszmaul, W. (2024). Modern hashing made simple.

Preliminaries (Basic Probability)

Union Bound

We know $Pr[A \cup B] = Pr[A] + Pr[B] - Pr[A \cap B]$



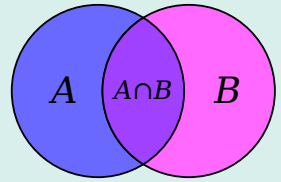
[5] sometimes $Pr[A]$ depends on constants that have to be chosen in dependence on c to achieve this bound

[6] Unless the expected time in the failure event is $\omega(\text{poly}(n))$

Preliminaries (Basic Probability)

Union Bound

We know $Pr[A \cup B] = Pr[A] + Pr[B] - Pr[A \cap B]$
Therefore, $Pr[A \cup B] \leqslant Pr[A] + Pr[B]$

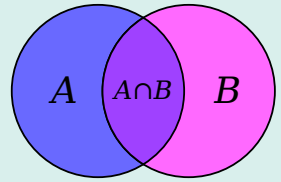


[5] sometimes $Pr[A]$ depends on constants that have to be chosen in dependence on c to achieve this bound
[6] Unless the expected time in the failure event is $\omega(\text{poly}(n))$

Preliminaries (Basic Probability)

Union Bound

We know $Pr[A \cup B] = Pr[A] + Pr[B] - Pr[A \cap B]$
Therefore, $Pr[A \cup B] \leqslant Pr[A] + Pr[B]$
And $Pr\left[\bigcup_i A_i\right] \leqslant \sum_i Pr[A_i]$

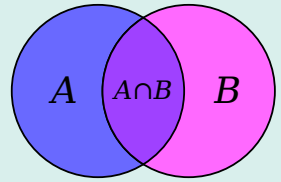


[5] sometimes $Pr[A]$ depends on constants that have to be chosen in dependence on c to achieve this bound
[6] Unless the expected time in the failure event is $\omega(\text{poly}(n))$

Preliminaries (Basic Probability)

Union Bound

We know $Pr[A \cup B] = Pr[A] + Pr[B] - Pr[A \cap B]$
Therefore, $Pr[A \cup B] \leqslant Pr[A] + Pr[B]$
And $Pr\left[\bigcup_i A_i\right] \leqslant \sum_i Pr[A_i]$



With High Probability

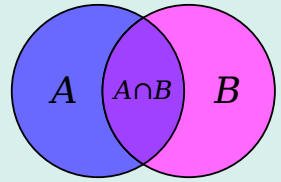
We say that an event A succeeds *with high probability* (whp) if $Pr[A] = 1 - \mathcal{O}\left(\frac{1}{n^c}\right)$ for **any** constant c. [5]

[5] sometimes $Pr[A]$ depends on constants that have to be chosen in dependence on c to achieve this bound
[6] Unless the expected time in the failure event is $\omega(\text{poly}(n))$

Preliminaries (Basic Probability)

Union Bound

We know $Pr[A \cup B] = Pr[A] + Pr[B] - Pr[A \cap B]$
Therefore, $Pr[A \cup B] \leqslant Pr[A] + Pr[B]$
And $Pr\left[\bigcup_i A_i\right] \leqslant \sum_i Pr[A_i]$



With High Probability

We say that an event A succeeds *with high probability* (whp) if $Pr[A] = 1 - \mathcal{O}\left(\frac{1}{n^c}\right)$ for **any** constant c. [5]

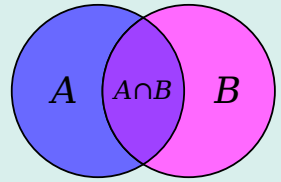
1. $\mathcal{O}(1)$ time whp is stronger than $\mathcal{O}(1)$ expected. [6]

[5] sometimes $Pr[A]$ depends on constants that have to be chosen in dependence on c to achieve this bound
[6] Unless the expected time in the failure event is $\omega(\text{poly}(n))$

Preliminaries (Basic Probability)

Union Bound

We know $Pr[A \cup B] = Pr[A] + Pr[B] - Pr[A \cap B]$
Therefore, $Pr[A \cup B] \leqslant Pr[A] + Pr[B]$
And $Pr\left[\bigcup_i A_i\right] \leqslant \sum_i Pr[A_i]$



With High Probability

We say that an event A succeeds *with high probability* (whp) if $Pr[A] = 1 - \mathcal{O}\left(\frac{1}{n^c}\right)$ for **any** constant c. [5]

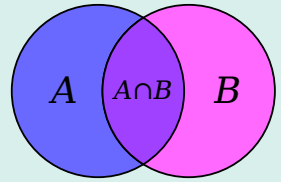
1. $\mathcal{O}(1)$ time whp is stronger than $\mathcal{O}(1)$ expected. [6]
2. If events $A_1, \ldots A_n$ succeed whp then $\bigcap_i A_i$ succeeds whp using a union bound

[5] sometimes $Pr[A]$ depends on constants that have to be chosen in dependence on c to achieve this bound
[6] Unless the expected time in the failure event is $\omega(\text{poly}(n))$

Preliminaries (Basic Probability)

Union Bound

We know $Pr[A \cup B] = Pr[A] + Pr[B] - Pr[A \cap B]$
Therefore, $Pr[A \cup B] \leqslant Pr[A] + Pr[B]$
And $Pr\left[\bigcup_i A_i\right] \leqslant \sum_i Pr[A_i]$



With High Probability

We say that an event A succeeds *with high probability* (whp) if $Pr[A] = 1 - \mathcal{O}\left(\frac{1}{n^c}\right)$ for **any** constant c. [5]

1. $\mathcal{O}(1)$ time whp is stronger than $\mathcal{O}(1)$ expected. [6]
2. If events $A_1, \ldots A_n$ succeed whp then $\bigcap_i A_i$ succeeds whp using a union bound

A Chernoff Bound

Consider a sum of independent $[0,1]$-valued random variables $X = \sum_{i=1}^{n} X_i$. Then $X \leqslant \mathbb{E}[X] + \mathcal{O}(\sqrt{\mathbb{E}[X] \log n})$ whp.

[5] sometimes $Pr[A]$ depends on constants that have to be chosen in dependence on c to achieve this bound
[6] Unless the expected time in the failure event is $\omega(\text{poly}(n))$

Preliminaries (A first whp result)

Lemma

Given a geometrically distributed random variable $X \sim \text{Geo}(p)$, then $X \leqslant c \log n$ whp, for constants p and c.

Preliminaries (A first whp result)

Lemma

Given a geometrically distributed random variable $X \sim \text{Geo}(p)$, then $X \leqslant c \log n$ whp, for constants p and c.

Proof.

$$Pr[X > c \ln n] = (1-p)^{c \ln n}$$

Preliminaries (A first whp result)

Lemma

Given a geometrically distributed random variable $X \sim \text{Geo}(p)$, then $X \leqslant c \log n$ whp, for constants p and c.

Proof.

$$Pr[X > c \ln n] = (1-p)^{c \ln n}$$
$$\ln Pr[X > c \ln n] = c \ln n \ln(1-p)$$

Preliminaries (A first whp result)

Lemma

Given a geometrically distributed random variable $X \sim \text{Geo}(p)$, then $X \leqslant c \log n$ whp, for constants p and c.

Proof.

$$
\begin{aligned}
Pr[X > c \ln n] &= (1-p)^{c \ln n} \\
\ln Pr[X > c \ln n] &= c \ln n \ln(1-p) \\
Pr[X > c \ln n] &= e^{c \ln n \ln(1-p)} \\
&= n^{c \ln(1-p)}
\end{aligned}
$$

Preliminaries (A first whp result)

Lemma

Given a geometrically distributed random variable $X \sim \text{Geo}(p)$, then $X \leqslant c \log n$ whp, for constants p and c.

Proof.

$$
\begin{aligned}
Pr[X > c \ln n] &= (1-p)^{c \ln n} \\
\ln Pr[X > c \ln n] &= c \ln n \ln(1-p) \\
Pr[X > c \ln n] &= e^{c \ln n \ln(1-p)} \\
&= n^{c \ln(1-p)}
\end{aligned}
$$

Since $\ln(1-p) < 0$ we have

$$
Pr[X \leqslant c \ln n] = 1 - \frac{1}{n^{\Theta(c)}}
$$

Preliminaries (Model)

Model of Computation

We assume the RAM model: $\mathcal{O}(1)$ operations on $w := (1 + \Theta(1)) \log n$ bits (see first lecture of this course)

Preliminaries (Model)

Model of Computation

We assume the RAM model: $\mathcal{O}(1)$ operations on $w := (1 + \Theta(1)) \log n$ bits (see first lecture of this course)

Exhaustive subtabulation (aka Four Russians)
Given a function $T(x)$, where x can be specified using $\frac{\log n}{2}$ bits
Construct a lookup table of $2^{\frac{\log n}{2}} = \sqrt{n}$ entries
$\mathcal{O}(1)$ evaluation of $T(x)$!

Preliminaries (Model)

Model of Computation

We assume the RAM model: $\mathcal{O}(1)$ operations on $w := (1 + \Theta(1)) \log n$ bits (see first lecture of this course)

Exhaustive subtabulation (aka Four Russians)
Given a function $T(x)$, where x can be specified using $\frac{\log n}{2}$ bits
Construct a lookup table of $2^{\frac{\log n}{2}} = \sqrt{n}$ entries
$\mathcal{O}(1)$ evaluation of $T(x)$!

Hash Table

The table has capacity for n keys.
Each key is represented by w bits.

Preliminaries (Model)

Model of Computation

We assume the RAM model: $\mathcal{O}(1)$ operations on $w := (1 + \Theta(1)) \log n$ bits (see first lecture of this course)

Exhaustive subtabulation (aka Four Russians)
Given a function $T(x)$, where x can be specified using $\frac{\log n}{2}$ bits
Construct a lookup table of $2^{\frac{\log n}{2}} = \sqrt{n}$ entries
$\mathcal{O}(1)$ evaluation of $T(x)$!

Hash Table

The table has capacity for n keys.
Each key is represented by w bits.
We say that a hash table is space efficient if it needs $(1 + o(1))nw$ bits.

Preliminaries (Model)

Model of Computation

We assume the RAM model: $\mathcal{O}(1)$ operations on $w := (1 + \Theta(1)) \log n$ bits (see first lecture of this course)

Exhaustive subtabulation (aka Four Russians)
Given a function $T(x)$, where x can be specified using $\frac{\log n}{2}$ bits
Construct a lookup table of $2^{\frac{\log n}{2}} = \sqrt{n}$ entries
$\mathcal{O}(1)$ evaluation of $T(x)$!

Hash Table

The table has capacity for n keys.
Each key is represented by w bits.
We say that a hash table is space efficient if it needs $(1 + o(1))nw$ bits.
E.g. a hash table using $nw + n \log \log n$ bits is space efficient.

Overview

	Array	Resizable
Query	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Insert	$\mathcal{O}(n)$	$\mathcal{O}(1)$ whp

$(1 + o(1))nw$ bits

$(1 + o(1))\bar{n}w$ bits

A terrible hash table

Array hash table

Simply store all keys in an array.

- Insertion: scan for an empty slot
- Query: scan the whole array
- Deletion: scan for item and remove it

A terrible hash table

Array hash table

Simply store all keys in an array.

- Insertion: scan for an empty slot
- Query: scan the whole array
- Deletion: scan for item and remove it

Analysis

Space: nw
Time: $\mathcal{O}(n)$ for all operations

Overview

	Array	Resizable
Query	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Insert	$\mathcal{O}(n)$	$\mathcal{O}(1)$ whp

$(1 + o(1))nw$ bits

$(1 + o(1))\bar{n}w$ bits

Overview

	Array	Partitioning	Resizable
Query	$\mathcal{O}(n)$	$\mathcal{O}(\log^5 n)$	$\mathcal{O}(1)$
Insert	$\mathcal{O}(n)$	$\mathcal{O}(\log^5 n)$ whp	$\mathcal{O}(1)$ whp

$(1 + o(1))nw$ bits

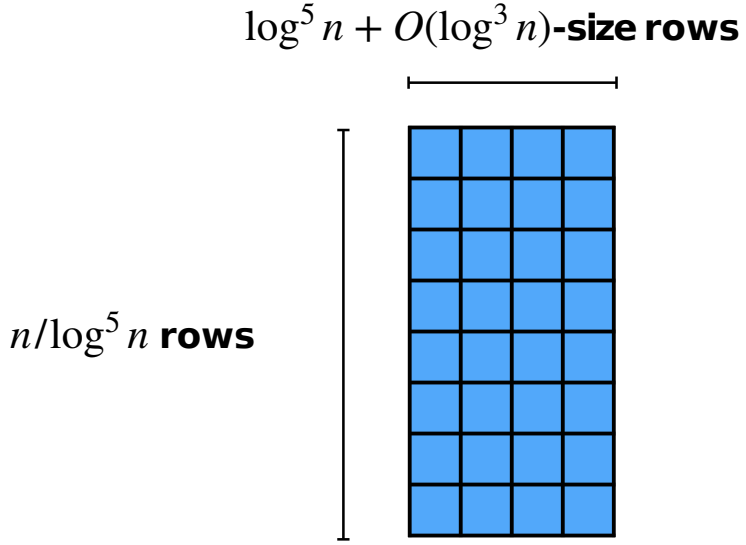
$(1 + o(1))\bar{n}w$ bits

Speeding up by partitioning

Idea

Break the array into $n/\log^5 n$ rows. The capacity of each row is $\log^5 n + \mathcal{O}(\log^3 n)$.

- Query: hash to a row & scan the row
- Deletion: hash to a row & find item & empty the slot
- Insertion: hash to a row & insert into empty slot & rebuild the entire table if there is no free slot



Speeding up by partitioning

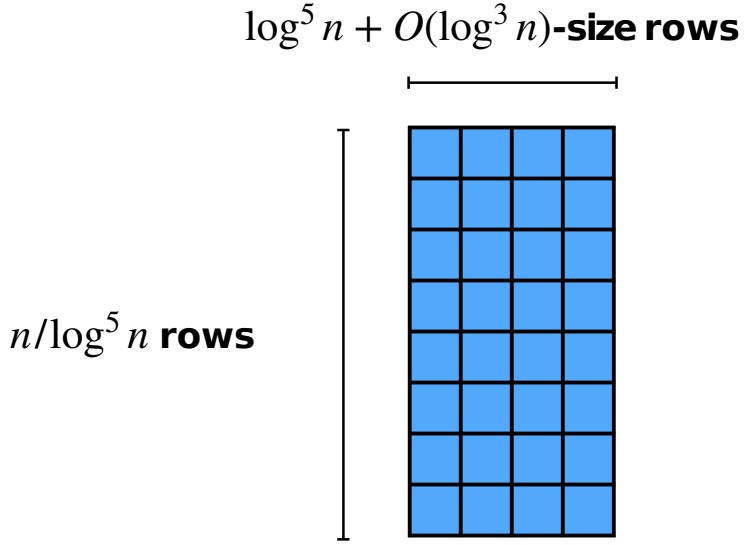
Idea

Break the array into $n/\log^5 n$ rows. The capacity of each row is $\log^5 n + \mathcal{O}(\log^3 n)$.

- Query: hash to a row & scan the row
- Deletion: hash to a row & find item & empty the slot
- Insertion: hash to a row & insert into empty slot & rebuild the entire table if there is no free slot

Row overflow

When a row overflows we have to rebuild everything using a new hash function. We show that this does not happen whp.



Speeding up by partitioning

Idea

Break the array into $n/\log^5 n$ rows. The capacity of each row is $\log^5 n + \mathcal{O}(\log^3 n)$.

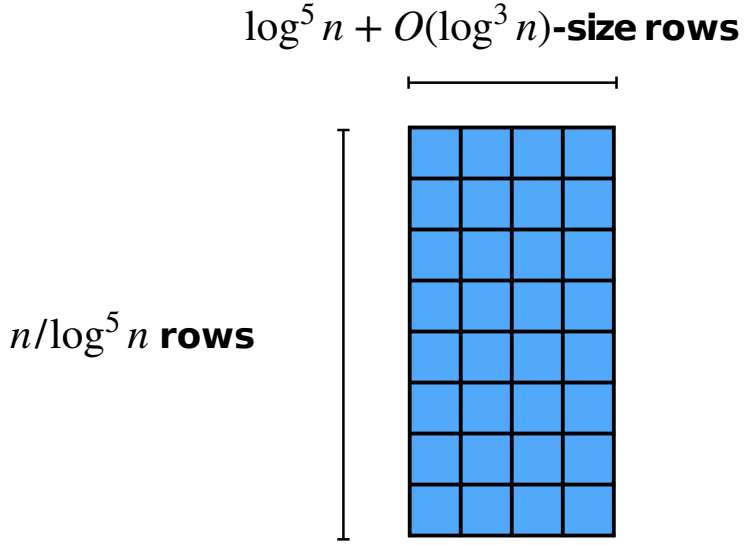
- Query: hash to a row & scan the row
- Deletion: hash to a row & find item & empty the slot
- Insertion: hash to a row & insert into empty slot & rebuild the entire table if there is no free slot

Row overflow

When a row overflows we have to rebuild everything using a new hash function. We show that this does not happen whp.

This gives us

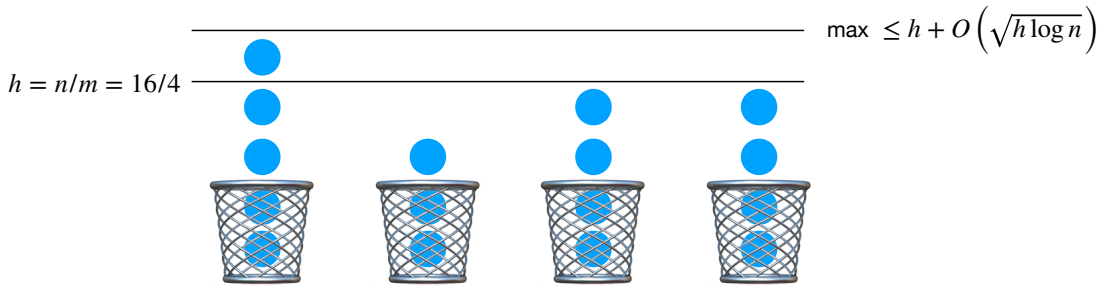
$\mathcal{O}(\log^5 n)$ operations (insertions only with high probability)
Space of $wn(1 + \mathcal{O}(1/\log^2 n))$ bits.



Speeding up by partitioning (Balls into Bins)

Lemma: Row overflows are unlikely

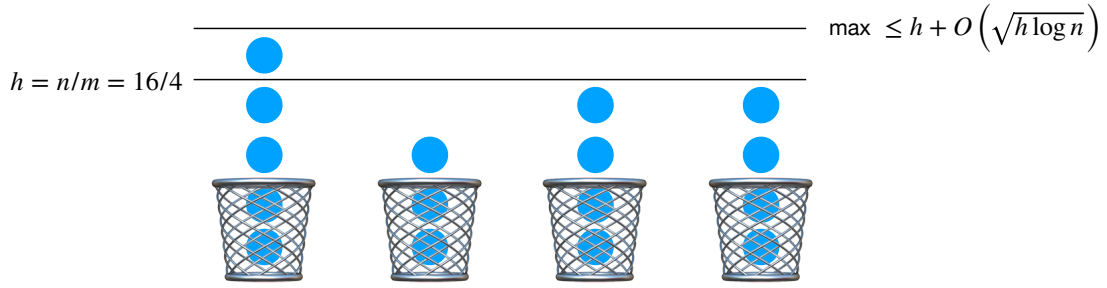
Suppose n balls are thrown randomly into m bins, $h = n/m \geqslant \log n$. Then the fullest bin has at most $h + \mathcal{O}(\sqrt{h \log n})$ balls whp



Speeding up by partitioning (Balls into Bins)

Lemma: Row overflows are unlikely

Suppose n balls are thrown randomly into m bins, $h = n/m \geqslant \log n$. Then the fullest bin has at most $h + \mathcal{O}(\sqrt{h \log n})$ balls whp



Proof

We bound the load in bin 1 and then union bound over all bins. Define the random variable

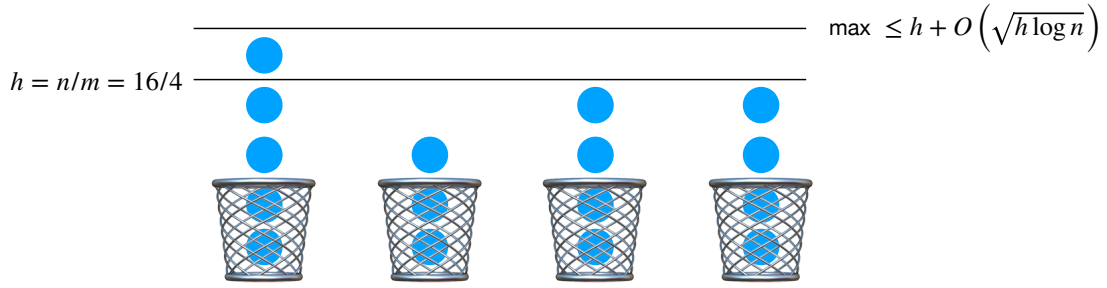
$$X_i = \begin{cases} 1, \text{if ball } i \text{ lands in bin 1} \\ 0, \text{otherwise} \end{cases}$$

and let $X = \sum_{i=1}^{n} X_i$. Then $\mathbb{E}[X] = h$.

Speeding up by partitioning (Balls into Bins)

Lemma: Row overflows are unlikely

Suppose n balls are thrown randomly into m bins, $h = n/m \geqslant \log n$. Then the fullest bin has at most $h + \mathcal{O}(\sqrt{h \log n})$ balls whp



Proof

We bound the load in bin 1 and then union bound over all bins. Define the random variable

$$X_i = \begin{cases} 1, \text{if ball } i \text{ lands in bin 1} \\ 0, \text{otherwise} \end{cases}$$

and let $X = \sum_{i=1}^{n} X_i$. Then $\mathbb{E}[X] = h$. By a Chernoff bound $X \leqslant h + \mathcal{O}(\sqrt{h \log n})$ whp.

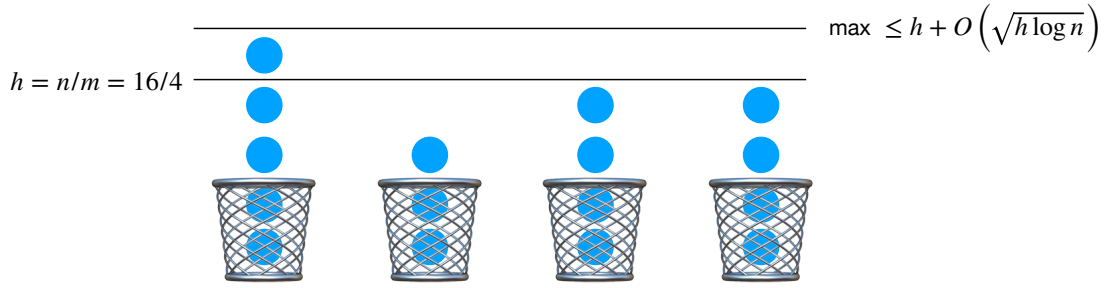
Reminder: A Chernoff Bound

Consider a sum of independent $[0, 1]$-valued random variables $X = \sum_{i=1}^{n} X_i$. Then $X \leqslant \mathbb{E}[X] + \mathcal{O}(\sqrt{\mathbb{E}[X] \log n})$ whp.

Speeding up by partitioning (Balls into Bins)

Lemma: Row overflows are unlikely

Suppose n balls are thrown randomly into m bins, $h = n/m \geqslant \log n$. Then the fullest bin has at most $h + \mathcal{O}(\sqrt{h \log n})$ balls whp



Proof

We bound the load in bin 1 and then union bound over all bins. Define the random variable

$$X_i = \begin{cases} 1, \text{if ball } i \text{ lands in bin 1} \\ 0, \text{otherwise} \end{cases}$$

and let $X = \sum_{i=1}^{n} X_i$. Then $\mathbb{E}[X] = h$. By a Chernoff bound $X \leqslant h + \mathcal{O}(\sqrt{h \log n})$ whp. Therefore, by the union bound, no bin has more than $h + \mathcal{O}(\sqrt{h \log n})$ whp

Reminder: A Chernoff Bound

Consider a sum of independent $[0, 1]$-valued random variables $X = \sum_{i=1}^{n} X_i$. Then $X \leqslant \mathbb{E}[X] + \mathcal{O}(\sqrt{\mathbb{E}[X] \log n})$ whp.

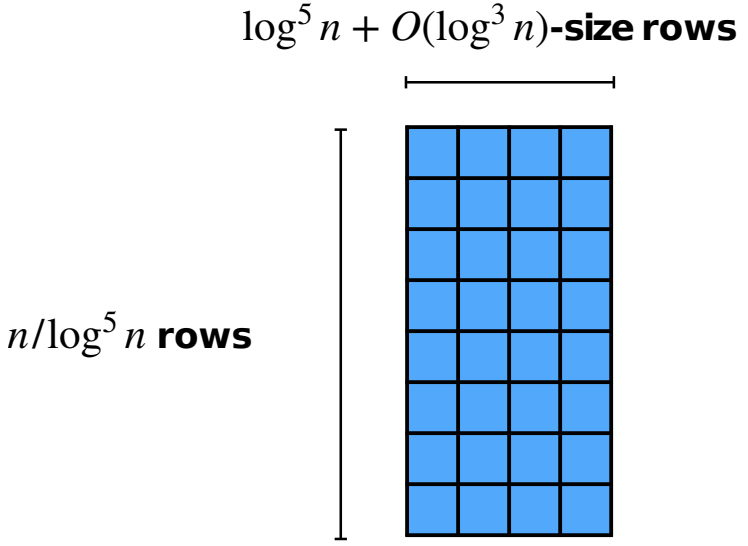
Speeding up by partitioning (Balls into Bins)

Lemma: Row overflows are unlikely

Suppose n balls are thrown randomly into m bins, $h = n/m \geqslant \log n$. Then the fullest bin has at most $h + \mathcal{O}(\sqrt{h \log n})$ balls whp

Example

If $m = n/\log^5 n$ bins (rows), the fullest bin has $\log^5 n + \mathcal{O}(\log^3 n)$ balls (keys) whp.



Overview

	Array	Partitioning	Resizable
Query	$\mathcal{O}(n)$	$\mathcal{O}(\log^5 n)$	$\mathcal{O}(1)$
Insert	$\mathcal{O}(n)$	$\mathcal{O}(\log^5 n)$ whp	$\mathcal{O}(1)$ whp

$(1 + o(1))nw$ bits

$(1 + o(1))\bar{n}w$ bits

Overview

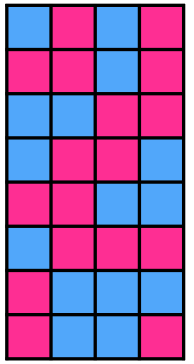
	Array	Partitioning	Left Justification	Resizable
Query	$\mathcal{O}(n)$	$\mathcal{O}(\log^5 n)$	$\mathcal{O}(\log^5 n)$	$\mathcal{O}(1)$
Insert	$\mathcal{O}(n)$	$\mathcal{O}(\log^5 n)$ whp	$\mathcal{O}(1)$ whp	$\mathcal{O}(1)$ whp

$(1 + o(1))nw$ bits

$(1 + o(1))\bar{n}w$ bits

Left justified rows

Instead of this:



Left justified rows

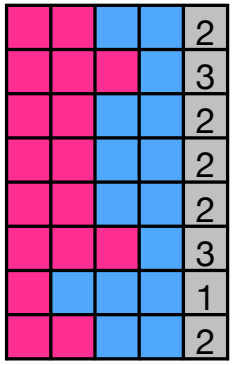
Insertions are easy

In each row: Keep items left justified and maintain a counter

Operations

$\mathcal{O}(1)$ whp **Insertion**: increment counter i & place into location i & rebuild table if row overflows
$\mathcal{O}(\log^5 n)$ **Query**: still has to search the entire array
$\mathcal{O}(\log^5 n)$ **Deletions**: Search & swap with rightmost & decrement counter

Keep items arranged like this:



Metadata = count for each row

Left justified rows

Insertions are easy

In each row: Keep items left justified and maintain a counter

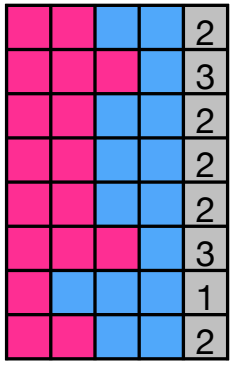
Operations

$\mathcal{O}(1)$ whp **Insertion**: increment counter i & place into location i & rebuild table if row overflows
$\mathcal{O}(\log^5 n)$ **Query**: still has to search the entire array
$\mathcal{O}(\log^5 n)$ **Deletions**: Search & swap with rightmost & decrement counter

Space

We have $n/\log^5 n$ counters, each requires $\mathcal{O}(\log\log n)$ bits. The space overhead is therefore $\mathcal{O}(\frac{n \log\log n}{\log^5 n}) < o(nw)$. Our hash table remains space-efficient.

Keep items arranged like this:



Metadata = count for each row

Overview

	Array	Partitioning	Left Justification	Resizable
Query	$\mathcal{O}(n)$	$\mathcal{O}(\log^5 n)$	$\mathcal{O}(\log^5 n)$	$\mathcal{O}(1)$
Insert	$\mathcal{O}(n)$	$\mathcal{O}(\log^5 n)$ whp	$\mathcal{O}(1)$ whp	$\mathcal{O}(1)$ whp

$(1 + o(1))nw$ bits

$(1 + o(1))\bar{n}w$ bits

Overview

	Array	Partitioning	Left Justification	Query Router	Resizable
Query	$\mathcal{O}(n)$	$\mathcal{O}(\log^5 n)$	$\mathcal{O}(\log^5 n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Insert	$\mathcal{O}(n)$	$\mathcal{O}(\log^5 n)$ whp	$\mathcal{O}(1)$ whp	$\mathcal{O}(1)$ expected	$\mathcal{O}(1)$ whp

$(1 + o(1))nw$ bits

$(1 + o(1))\bar{n}w$ bits

Query router

Goal: $\mathcal{O}(1)$ queries

Can we locate a key using a data structure that maps keys to their locations? But isn't this the problem we are trying to solve in the first place?

Query router

Goal: $\mathcal{O}(1)$ queries

Can we locate a key using a data structure that maps keys to their locations? But isn't this the problem we are trying to solve in the first place?

An $\mathcal{O}(1)$ B-Tree !?

In each row, use a B-Tree to map a key to its location.

1. We need $\mathcal{O}(1)$ height
   Use a fanout of $\sqrt{\log n} + 1$ and height of ≈ 10 as $\log^5 n = \sqrt{\log n}^{10}$

Query router

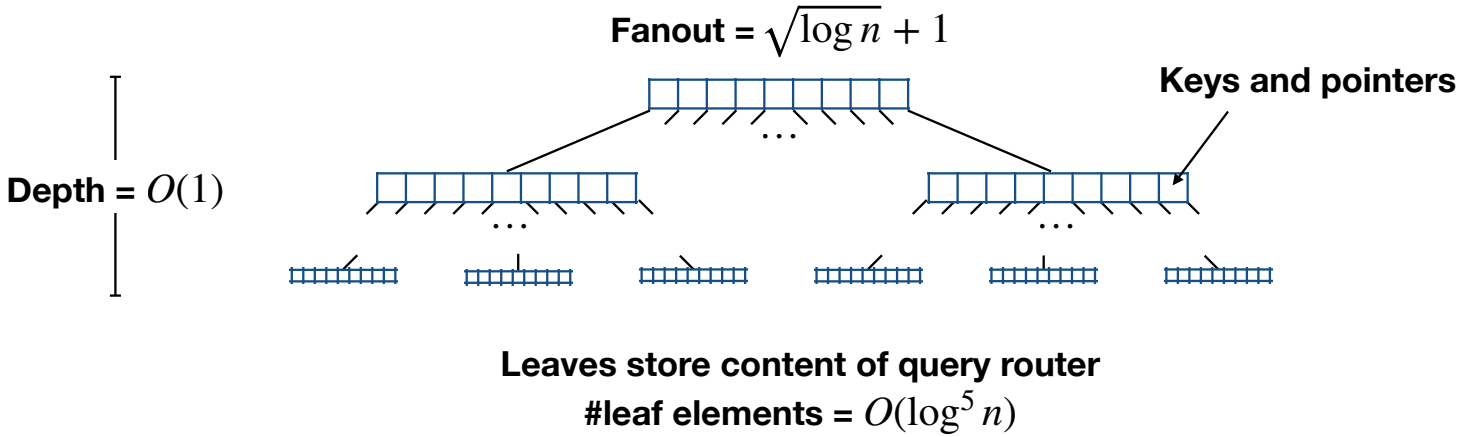
Goal: $\mathcal{O}(1)$ queries

Can we locate a key using a data structure that maps keys to their locations? But isn't this the problem we are trying to solve in the first place?

An $\mathcal{O}(1)$ B-Tree !?

In each row, use a B-Tree to map a key to its location.

1. We need $\mathcal{O}(1)$ height
   Use a fanout of $\sqrt{\log n} + 1$ and height of ≈ 10 as $\log^5 n = \sqrt{\log n}^{10}$



Query router

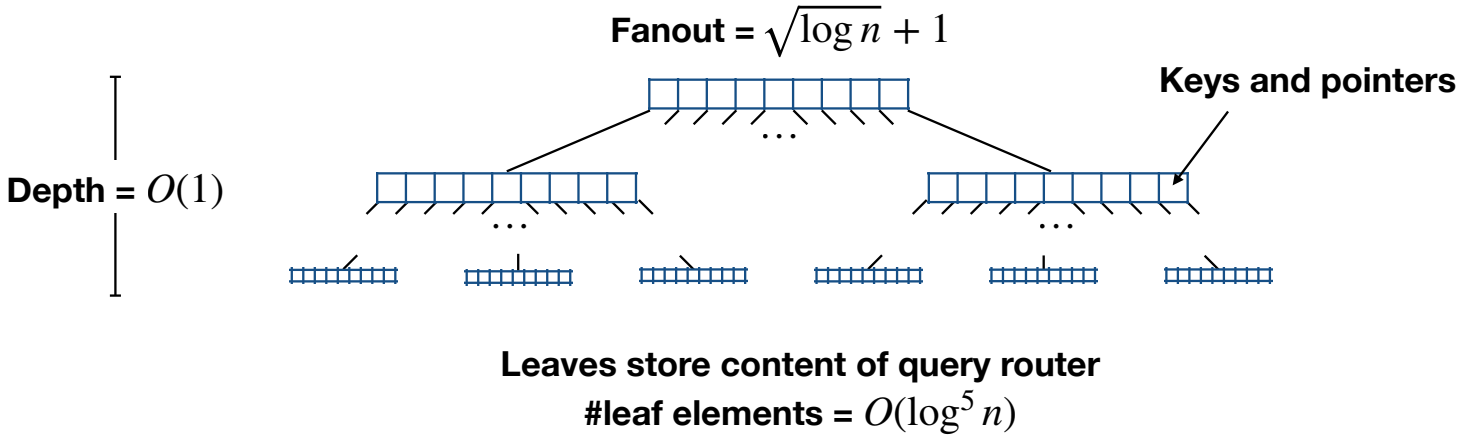
Goal: $\mathcal{O}(1)$ queries

Can we locate a key using a data structure that maps keys to their locations? But isn't this the problem we are trying to solve in the first place?

An $\mathcal{O}(1)$ B-Tree !?

In each row, use a B-Tree to map a key to its location.

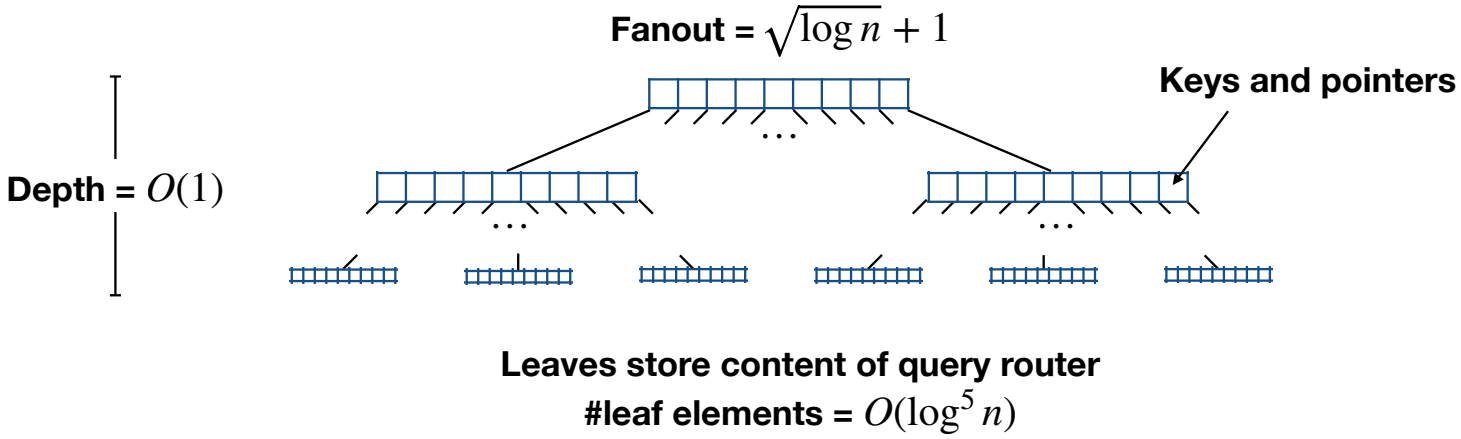
1. We need $\mathcal{O}(1)$ height
   Use a fanout of $\sqrt{\log n} + 1$ and height of ≈ 10 as $\log^5 n = \sqrt{\log n}^{10}$
2. We need $\mathcal{O}(1)$ operations within the nodes.
   Idea: Use exhaustive subtabulation within the nodes



Query router

Exhaustive Subtabulation

Use exhaustive subtabulation to implement any modification / query on a node.
Table is indexed by the node and the key that is modified / queried.
Node and key need to be described in $o(\log n)$ bits.

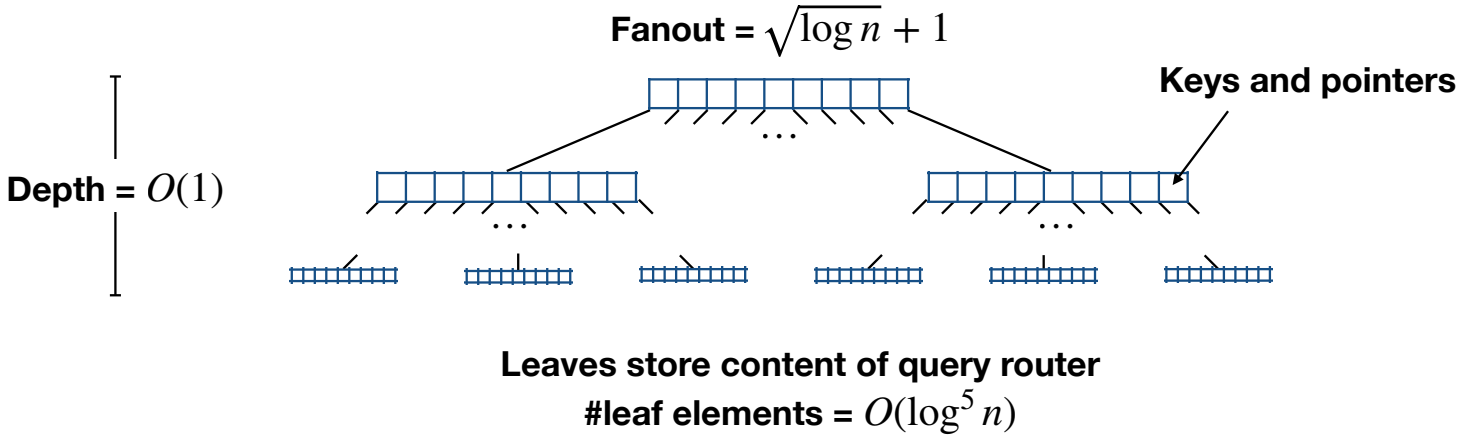


Query router

Exhaustive Subtabulation

Use exhaustive subtabulation to implement any modification / query on a node.
Table is indexed by the node and the key that is modified / queried.
Node and key need to be described in $o(\log n)$ bits. A node stores:

1. $\sqrt{\log n} + 1$ pointers to nodes: $\mathcal{O}(\log \log n)$ bits ($\mathcal{O}(\log^5 n)$ nodes). ☺

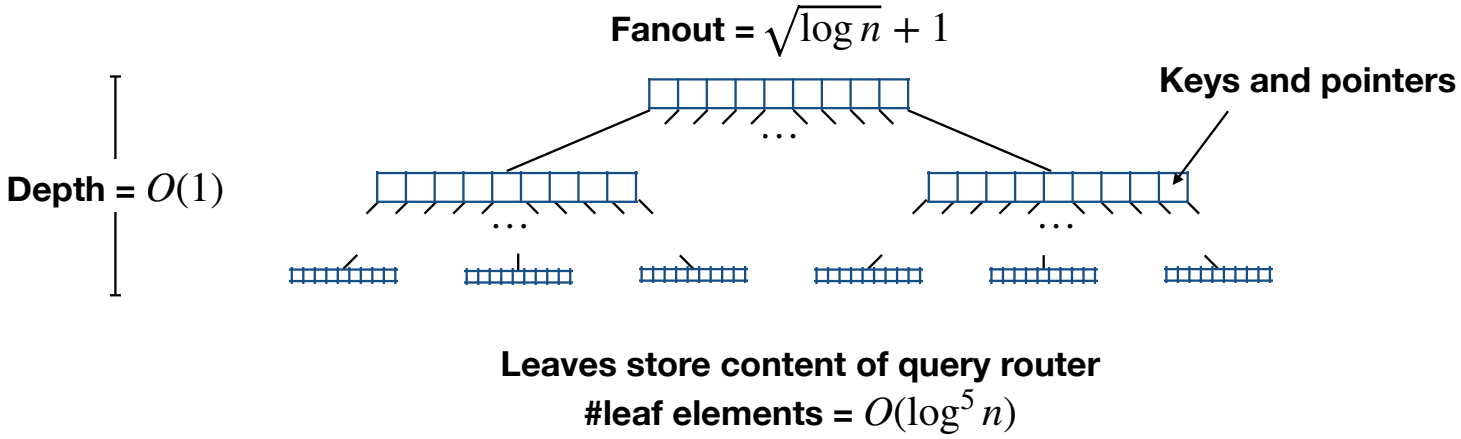


Query router

Exhaustive Subtabulation

Use exhaustive subtabulation to implement any modification / query on a node.
Table is indexed by the node and the key that is modified / queried.
Node and key need to be described in $o(\log n)$ bits. A node stores:

1. $\sqrt{\log n} + 1$ pointers to nodes: $\mathcal{O}(\log \log n)$ bits ($\mathcal{O}(\log^5 n)$ nodes). ☺
2. $\sqrt{\log n}$ pointers to array locations: $\mathcal{O}(\log \log n)$ bits ($\mathcal{O}(\log^5 n)$ locations). ☺

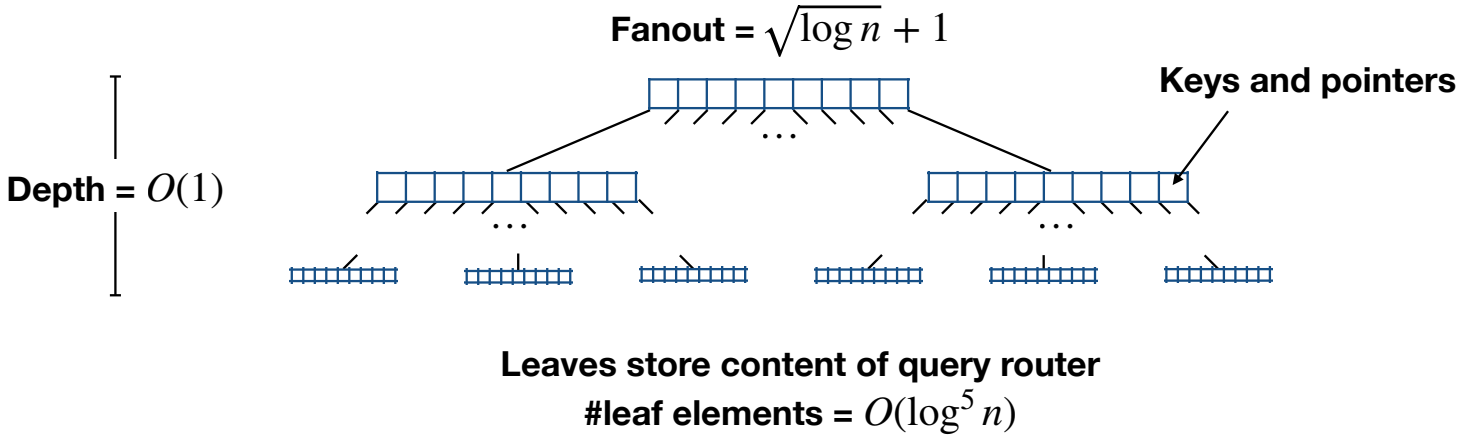


Query router

Exhaustive Subtabulation

Use exhaustive subtabulation to implement any modification / query on a node.
Table is indexed by the node and the key that is modified / queried.
Node and key need to be described in $o(\log n)$ bits. A node stores:

1. $\sqrt{\log n} + 1$ pointers to nodes: $\mathcal{O}(\log \log n)$ bits ($\mathcal{O}(\log^5 n)$ nodes). ☺
2. $\sqrt{\log n}$ pointers to array locations: $\mathcal{O}(\log \log n)$ bits ($\mathcal{O}(\log^5 n)$ locations). ☺
3. $\sqrt{\log n}$ keys: w bits and $w\sqrt{\log n}$ exceeds $o(\log n)$ by far. ☹



Query router

Exhaustive Subtabulation

Use exhaustive subtabulation to implement any modification / query on a node.
Table is indexed by the node and the key that is modified / queried.
Node and key need to be described in $o(\log n)$ bits. A node stores:

1. $\sqrt{\log n} + 1$ pointers to nodes: $\mathcal{O}(\log \log n)$ bits ($\mathcal{O}(\log^5 n)$ nodes). ☺
2. $\sqrt{\log n}$ pointers to array locations: $\mathcal{O}(\log \log n)$ bits ($\mathcal{O}(\log^5 n)$ locations). ☺
3. $\sqrt{\log n}$ keys: w bits and $w\sqrt{\log n}$ exceeds $o(\log n)$ by far. ☹

Fingerprints

- Observation: No need to store entire key, we only need unique identifiers/fingerprints.

Query router

Exhaustive Subtabulation

Use exhaustive subtabulation to implement any modification / query on a node.
Table is indexed by the node and the key that is modified / queried.
Node and key need to be described in $o(\log n)$ bits. A node stores:

1. $\sqrt{\log n} + 1$ pointers to nodes: $\mathcal{O}(\log \log n)$ bits ($\mathcal{O}(\log^5 n)$ nodes). ☺
2. $\sqrt{\log n}$ pointers to array locations: $\mathcal{O}(\log \log n)$ bits ($\mathcal{O}(\log^5 n)$ locations). ☺
3. $\sqrt{\log n}$ keys: w bits and $w\sqrt{\log n}$ exceeds $o(\log n)$ by far. ☹

Fingerprints

- Observation: No need to store entire key, we only need unique identifiers/fingerprints.
- Idea: Use a hash function to map a key to a $\log \log n$ bit fingerprint $f : S \to \{0,1\}^{\log \log n}$.

Query router

Exhaustive Subtabulation

Use exhaustive subtabulation to implement any modification / query on a node.
Table is indexed by the node and the key that is modified / queried.
Node and key need to be described in $o(\log n)$ bits. A node stores:

1. $\sqrt{\log n} + 1$ pointers to nodes: $\mathcal{O}(\log \log n)$ bits ($\mathcal{O}(\log^5 n)$ nodes). ☺
2. $\sqrt{\log n}$ pointers to array locations: $\mathcal{O}(\log \log n)$ bits ($\mathcal{O}(\log^5 n)$ locations). ☺
3. $\sqrt{\log n}$ keys: w bits and $w\sqrt{\log n}$ exceeds $o(\log n)$ by far. ☹

Fingerprints

- Observation: No need to store entire key, we only need unique identifiers/fingerprints.
- Idea: Use a hash function to map a key to a $\log \log n$ bit fingerprint $f : S \to \{0,1\}^{\log \log n}$.
- Now, within the query router use unique fingerprints $f(x)$ instead of full keys x.

Query router

Exhaustive Subtabulation

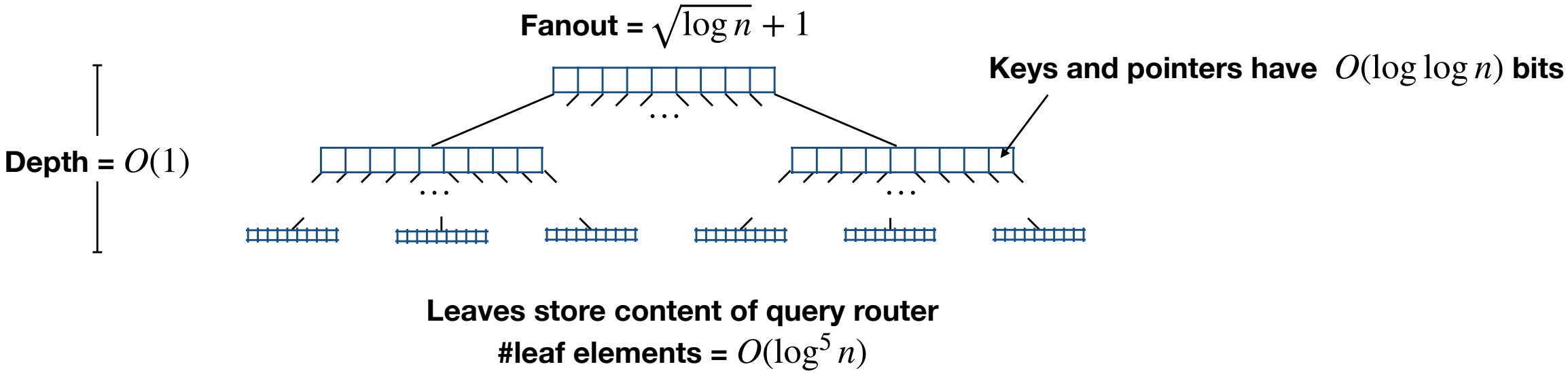
Use exhaustive subtabulation to implement any modification / query on a node.
Table is indexed by the node and the key that is modified / queried.
Node and key need to be described in $o(\log n)$ bits. A node stores:

1. $\sqrt{\log n} + 1$ pointers to nodes: $\mathcal{O}(\log \log n)$ bits ($\mathcal{O}(\log^5 n)$ nodes). ☺
2. $\sqrt{\log n}$ pointers to array locations: $\mathcal{O}(\log \log n)$ bits ($\mathcal{O}(\log^5 n)$ locations). ☺
3. $\sqrt{\log n}$ keys: w bits and $w\sqrt{\log n}$ exceeds $o(\log n)$ by far. ☹

Fingerprints

- Observation: No need to store entire key, we only need unique identifiers/fingerprints.
- Idea: Use a hash function to map a key to a $\log \log n$ bit fingerprint $f : S \to \{0,1\}^{\log \log n}$.
- Now, within the query router use unique fingerprints $f(x)$ instead of full keys x.
- Next: Need to handle fingerprint collisions.

Query router

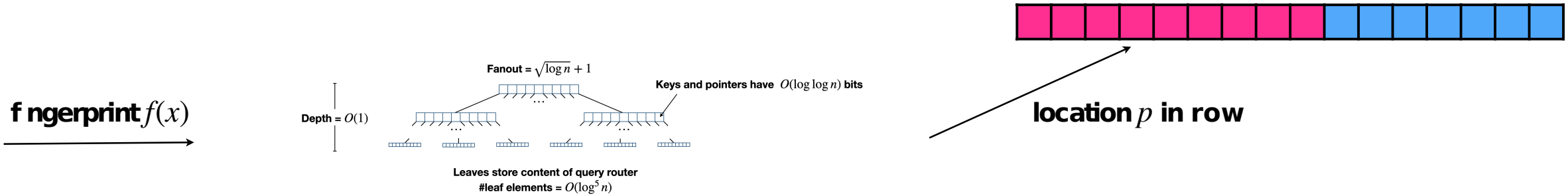


Within this context the B-Tree has $\mathcal{O}(1)$ operations!

Query router

Query(x)

1. Hash x to find the row.
2. Compute x's fingerprint $f(x)$.
3. Use the query router to convert $f(x)$ to the location p in the row.
4. Check whether x is stored at location p. If not, x is not in the hash table.

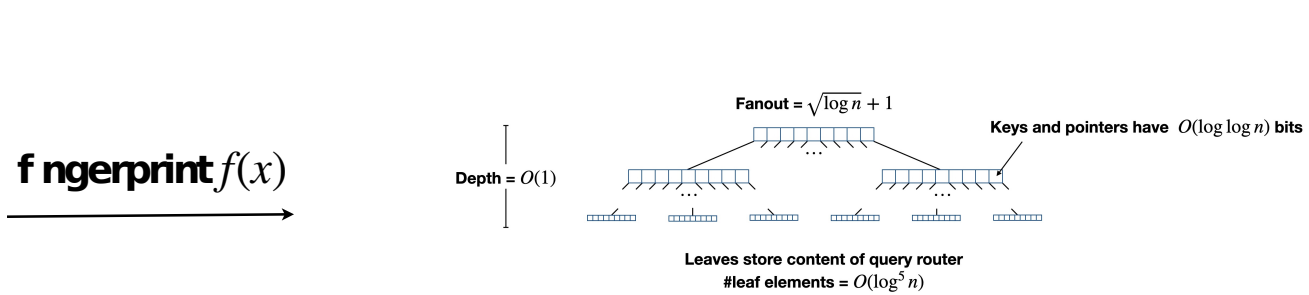


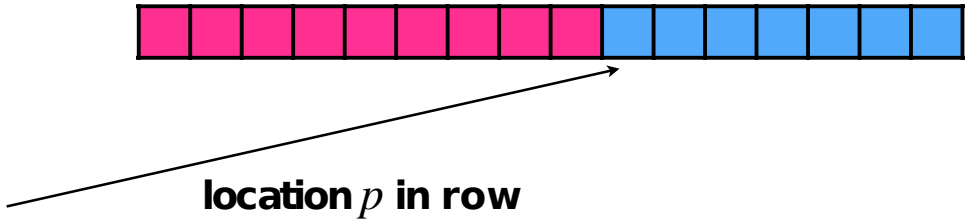
Query router

Insert(x)

(Query x to check whether it is already present.)

1. Hash x to a row.
2. Assign x to a location p in the row.
3. Compute x's fingerprint $f(x)$. Place the pair $(f(x), p)$ in the query router.
4. If a fingerprint collision occurs, rebuild the row with a new fingerprint hash function. [7]





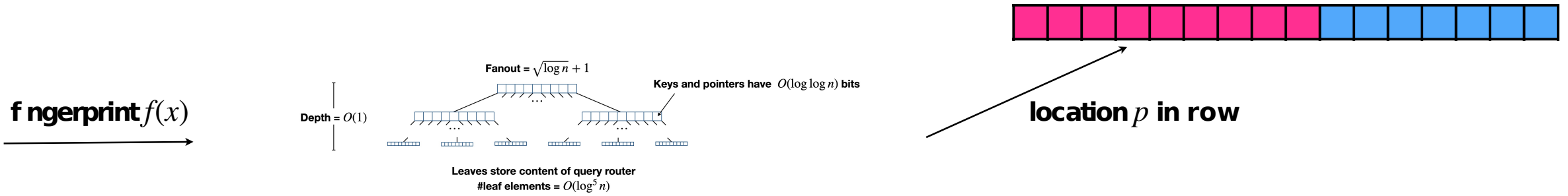
[7] For simplicity, we assume that representing the hash function takes no space

Query router

Delete(x)

(Query x to check whether it is already present.)

1. Delete the fingerprint $f(x)$ from the query router.
2. Delete the key from the array.



Query router (Analysis)

Space

Each key uses $\mathcal{O}(\log \log n)$ bits in some query router:

1. 1 fingerprint,
2. 1 pointer to array location,
3. $\mathcal{O}(1)$ B-Tree pointers, since every B-Tree node is at least half full [8].

Hence, the query routers need $\mathcal{O}(n \log \log n) < o(nw)$ bits (we are still space efficient).

[8] except for root nodes, but this is a lower order term

Query router (Analysis)

Space

Each key uses $\mathcal{O}(\log \log n)$ bits in some query router:

1. 1 fingerprint,
2. 1 pointer to array location,
3. $\mathcal{O}(1)$ B-Tree pointers, since every B-Tree node is at least half full [8].

Hence, the query routers need $\mathcal{O}(n \log \log n) < o(nw)$ bits (we are still space efficient).

Time

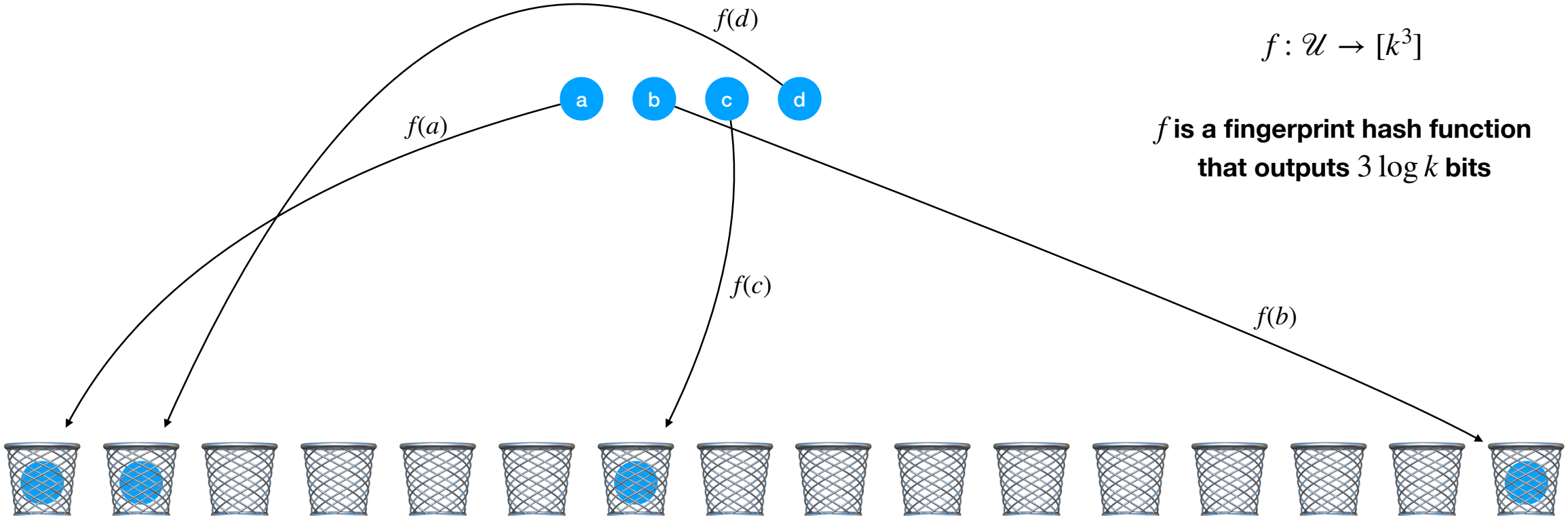
- Query / Delete: still $\mathcal{O}(1)$
- Insert: If fingerprint collision occurs, pick new fingerprint function and reconstruct row.
  - This is slow: Takes time $\mathcal{O}(\log^5 n)$
  - We need to show that this is rare: $< \mathcal{O}(1/\log^5 n)$
  - $\mathcal{O}(1)$ in expectation

[8] except for root nodes, but this is a lower order term

Query router (Analysis)

Lemma

If k balls are thrown randomly into $m = k^3$ bins, the probability that every ball lands in a bin by itself is at least $1 - \mathcal{O}(1/k)$.



$f : \mathcal{U} \to [k^3]$

f is a fingerprint hash function that outputs $3 \log k$ bits

Query router (Analysis)

Lemma

If k balls are thrown randomly into $m = k^3$ bins, the probability that every ball lands in a bin by itself is at least $1 - \mathcal{O}(1/k)$.

Proof.

$$Pr\left[\bigcup_{x \neq y} [f(x) = f(y)]\right]$$

Query router (Analysis)

Lemma

If k balls are thrown randomly into $m = k^3$ bins, the probability that every ball lands in a bin by itself is at least $1 - \mathcal{O}(1/k)$.

Proof.

$$Pr\left[\bigcup_{x \neq y} [f(x) = f(y)]\right] \overset{\text{union bound}}{\leqslant} \sum_{x \neq y} Pr[f(x) = f(y)]$$

Reminder: Union Bound

$Pr\left[\bigcup_i A_i\right] \leqslant \sum_i Pr[A_i]$

Query router (Analysis)

Lemma

If k balls are thrown randomly into $m = k^3$ bins, the probability that every ball lands in a bin by itself is at least $1 - \mathcal{O}(1/k)$.

Proof.

$$Pr\left[\bigcup_{x \neq y} [f(x) = f(y)]\right] \overset{\text{union bound}}{\leqslant} \sum_{x \neq y} Pr[f(x) = f(y)] = \frac{\binom{k}{2}}{k^3} \leqslant \mathcal{O}(1/k)$$

Reminder: Union Bound

$Pr\left[\bigcup_i A_i\right] \leqslant \sum_i Pr[A_i]$

Query router (Analysis)

Lemma

If k balls are thrown randomly into $m = k^3$ bins, the probability that every ball lands in a bin by itself is at least $1 - \mathcal{O}(1/k)$.

Proof.

$$Pr\left[\bigcup_{x \neq y} [f(x) = f(y)]\right] \overset{\text{union bound}}{\leqslant} \sum_{x \neq y} Pr[f(x) = f(y)] = \frac{\binom{k}{2}}{k^3} \leqslant \mathcal{O}(1/k)$$

Reminder: Union Bound

$Pr\left[\bigcup_i A_i\right] \leqslant \sum_i Pr[A_i]$

Applying the Lemma

We have $k = \Theta(\log^5 n)$ keys.
If we use $m = k^3 = \Theta(\log^{15} n)$ distinct fingerprints then the probability of collision is $\mathcal{O}(1/\log^5 n)$.

Query router (Analysis)

Lemma

If k balls are thrown randomly into $m = k^3$ bins, the probability that every ball lands in a bin by itself is at least $1 - \mathcal{O}(1/k)$.

Proof.

$$Pr\left[\bigcup_{x \neq y} [f(x) = f(y)]\right] \overset{\text{union bound}}{\leqslant} \sum_{x \neq y} Pr[f(x) = f(y)] = \frac{\binom{k}{2}}{k^3} \leqslant \mathcal{O}(1/k)$$

Reminder: Union Bound

$Pr\left[\bigcup_i A_i\right] \leqslant \sum_i Pr[A_i]$

Applying the Lemma

We have $k = \Theta(\log^5 n)$ keys.
If we use $m = k^3 = \Theta(\log^{15} n)$ distinct fingerprints then the probability of collision is $\mathcal{O}(1/\log^5 n)$.
I.e. use fingerprints of $\Theta(\log m) = \Theta(\log \log n)$ bits.

Overview

	Array	Partitioning	Left Justification	Query Router	Resizable
Query	$\mathcal{O}(n)$	$\mathcal{O}(\log^5 n)$	$\mathcal{O}(\log^5 n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Insert	$\mathcal{O}(n)$	$\mathcal{O}(\log^5 n)$ whp	$\mathcal{O}(1)$ whp	$\mathcal{O}(1)$ expected	$\mathcal{O}(1)$ whp

$(1 + o(1))nw$ bits

$(1 + o(1))\bar{n}w$ bits

Overview

	Array	Partitioning	Left Justification	Query Router	Deamortization	Resizable
Query	$\mathcal{O}(n)$	$\mathcal{O}(\log^5 n)$	$\mathcal{O}(\log^5 n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Insert	$\mathcal{O}(n)$	$\mathcal{O}(\log^5 n)$ whp	$\mathcal{O}(1)$ whp	$\mathcal{O}(1)$ expected	$\mathcal{O}(1)$ whp	$\mathcal{O}(1)$ whp

$(1 + o(1))nw$ bits

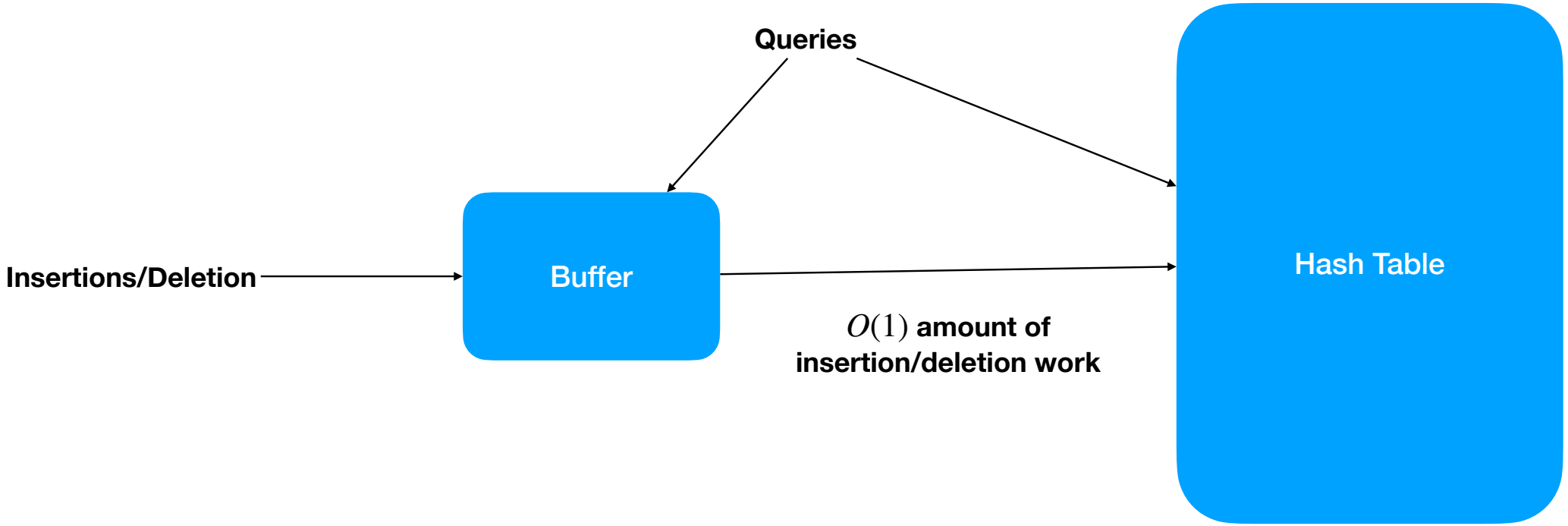
$(1 + o(1))\bar{n}w$ bits

Deamortization

Buffer-based deamortization

Maintain a buffer for insertions and deletions to spread the cost over a longer time. The buffer needs to:

1. Support deterministic $\mathcal{O}(1)$ buffer operations.
   At the cost of space efficiency: capacity should be small.
2. Large capacity, so that it does not fall behind and overflow.



Deamortization

Buffer-based deamortization

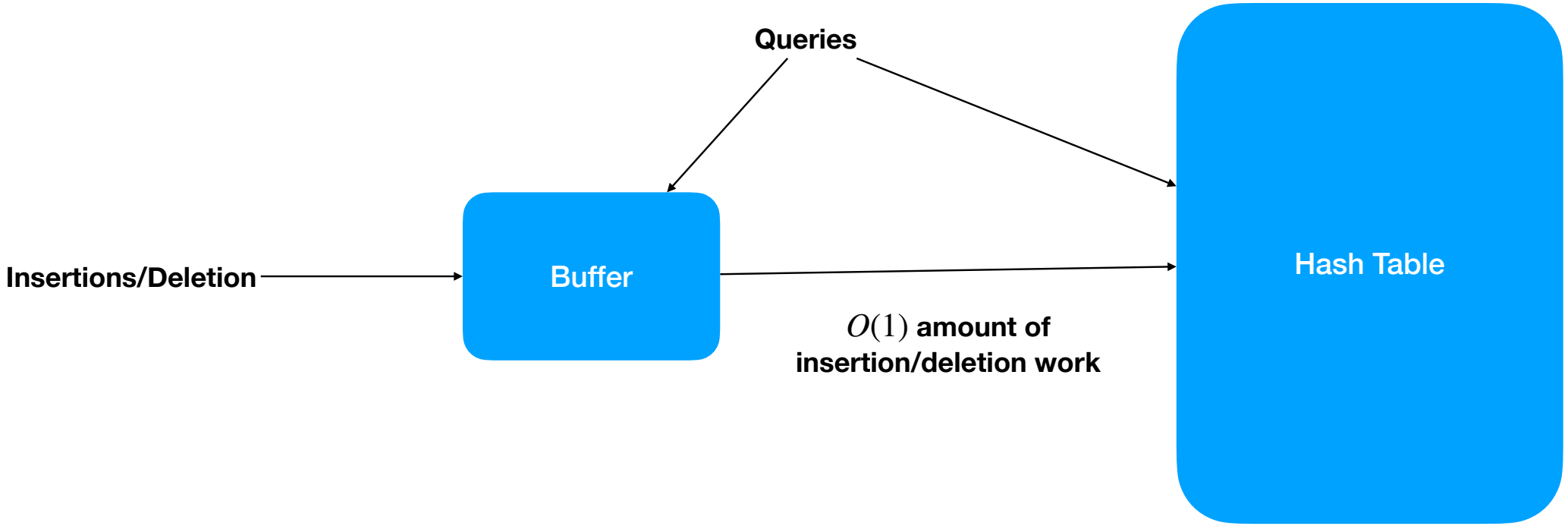
Maintain a buffer for insertions and deletions to spread the cost over a longer time. The buffer needs to:

1. Support deterministic $\mathcal{O}(1)$ buffer operations.
   At the cost of space efficiency: capacity should be small.
2. Large capacity, so that it does not fall behind and overflow.

Lemma: Buffer capacity

A batch of $\sqrt{n}$ insertions takes $\mathcal{O}(\sqrt{n})$ time **whp**.

This implies that a buffer of capacity of $2\sqrt{n}$ will suffice: Once the buffer contains $\sqrt{n}$ elements, at least $\sqrt{n}$ elements are processed after the next $\sqrt{n}$ elements have arrived.



Deamortization

Lemma: Buffer capacity

Any batch of $\sqrt{n}$ insertions takes $\mathcal{O}(\sqrt{n})$ time **whp**.

Proof

1. $A_1, \ldots, A_{n/\log^5 n}$: A_i #number of insertions mapped to the i-th row.
2. $X_1, \ldots, X_{n/\log^5 n}$: X_i time needed for the A_i operations in the i-th row

Deamortization

Lemma: Buffer capacity

Any batch of $\sqrt{n}$ insertions takes $\mathcal{O}(\sqrt{n})$ time **whp**.

Proof

1. $A_1, \ldots, A_{n/\log^5 n}$: A_i #number of insertions mapped to the i-th row.
2. $X_1, \ldots, X_{n/\log^5 n}$: X_i time needed for the A_i operations in the i-th row

Whp there are no overflows: $A_i < \mathcal{O}(\log^5 n)$ insertions per row

Deamortization

Lemma: Buffer capacity

Any batch of $\sqrt{n}$ insertions takes $\mathcal{O}(\sqrt{n})$ time **whp**.

Proof

1. $A_1, \ldots, A_{n/\log^5 n}$: A_i #number of insertions mapped to the i-th row.
2. $X_1, \ldots, X_{n/\log^5 n}$: X_i time needed for the A_i operations in the i-th row

Whp there are no overflows: $A_i < \mathcal{O}(\log^5 n)$ insertions per row
Rebuild time on fingerprint collision: $\mathcal{O}(\log^5 n)$

Deamortization

Lemma: Buffer capacity

Any batch of $\sqrt{n}$ insertions takes $\mathcal{O}(\sqrt{n})$ time **whp**.

Proof

1. $A_1, \ldots, A_{n/\log^5 n}$: A_i #number of insertions mapped to the i-th row.
2. $X_1, \ldots, X_{n/\log^5 n}$: X_i time needed for the A_i operations in the i-th row

Whp there are no overflows: $A_i < \mathcal{O}(\log^5 n)$ insertions per row
Rebuild time on fingerprint collision: $\mathcal{O}(\log^5 n)$
Number rebuilds per insertion is geometrically distributed and therefore at most $\mathcal{O}(\log n)$ whp (see Preliminaries).

Deamortization

Lemma: Buffer capacity

Any batch of $\sqrt{n}$ insertions takes $\mathcal{O}(\sqrt{n})$ time **whp**.

Proof

1. $A_1, \ldots, A_{n/\log^5 n}$: A_i #number of insertions mapped to the i-th row.
2. $X_1, \ldots, X_{n/\log^5 n}$: X_i time needed for the A_i operations in the i-th row

Whp there are no overflows: $A_i < \mathcal{O}(\log^5 n)$ insertions per row
Rebuild time on fingerprint collision: $\mathcal{O}(\log^5 n)$
Number rebuilds per insertion is geometrically distributed and therefore at most $\mathcal{O}(\log n)$ whp (see Preliminaries).
Hence, $X_i < \mathcal{O}(\log^{11} n)$ whp.

Deamortization

Lemma: Buffer capacity

Any batch of $\sqrt{n}$ insertions takes $\mathcal{O}(\sqrt{n})$ time **whp**.

Proof

1. $A_1, \ldots, A_{n/\log^5 n}$: A_i #number of insertions mapped to the i-th row.
2. $X_1, \ldots, X_{n/\log^5 n}$: X_i time needed for the A_i operations in the i-th row

Whp there are no overflows: $A_i < \mathcal{O}(\log^5 n)$ insertions per row
Rebuild time on fingerprint collision: $\mathcal{O}(\log^5 n)$
Number rebuilds per insertion is geometrically distributed and therefore at most $\mathcal{O}(\log n)$ whp (see Preliminaries).
Hence, $X_i < \mathcal{O}(\log^{11} n)$ whp.
We need to bound $X := \sum_i X_i$.

Deamortization

Lemma: Buffer capacity

Any batch of $\sqrt{n}$ insertions takes $\mathcal{O}(\sqrt{n})$ time **whp**.

Proof

1. $A_1, \ldots, A_{n/\log^5 n}$: A_i #number of insertions mapped to the i-th row.
2. $X_1, \ldots, X_{n/\log^5 n}$: X_i time needed for the A_i operations in the i-th row

Whp there are no overflows: $A_i < \mathcal{O}(\log^5 n)$ insertions per row
Rebuild time on fingerprint collision: $\mathcal{O}(\log^5 n)$
Number rebuilds per insertion is geometrically distributed and therefore at most $\mathcal{O}(\log n)$ whp (see Preliminaries).
Hence, $X_i < \mathcal{O}(\log^{11} n)$ whp.
We need to bound $X := \sum_i X_i$.
Consider $X/\Theta(\log^{11} n)$, the sum of independent $[0,1]$ random variables, with $\mu := \mathbb{E}[X/\Theta(\log^{11} n)] = \Theta(n^{1/2}/\log^{11} n)$:

Deamortization

Lemma: Buffer capacity

Any batch of $\sqrt{n}$ insertions takes $\mathcal{O}(\sqrt{n})$ time **whp**.

Chernoff Bound

Consider a sum of independent $[0,1]$-valued random variables $X = \sum_{i=1}^{n} X_i$. Then $X \leqslant \mathbb{E}[X] + \mathcal{O}(\sqrt{\mathbb{E}[X] \log n})$ whp.

Proof

1. $A_1, \ldots, A_{n/\log^5 n}$: A_i #number of insertions mapped to the i-th row.
2. $X_1, \ldots, X_{n/\log^5 n}$: X_i time needed for the A_i operations in the i-th row

Whp there are no overflows: $A_i < \mathcal{O}(\log^5 n)$ insertions per row
Rebuild time on fingerprint collision: $\mathcal{O}(\log^5 n)$
Number rebuilds per insertion is geometrically distributed and therefore at most $\mathcal{O}(\log n)$ whp (see Preliminaries).
Hence, $X_i < \mathcal{O}(\log^{11} n)$ whp.
We need to bound $X := \sum_i X_i$.
Consider $X/\Theta(\log^{11} n)$, the sum of independent $[0,1]$ random variables, with $\mu := \mathbb{E}[X/\Theta(\log^{11} n)] = \Theta(n^{1/2}/\log^{11} n)$:

Deamortization

Lemma: Buffer capacity

Any batch of $\sqrt{n}$ insertions takes $\mathcal{O}(\sqrt{n})$ time **whp**.

Chernoff Bound

Consider a sum of independent $[0,1]$-valued random variables $X = \sum_{i=1}^{n} X_i$. Then $X \leqslant \mathbb{E}[X] + \mathcal{O}(\sqrt{\mathbb{E}[X] \log n})$ whp.

Proof

1. $A_1, \ldots, A_{n/\log^5 n}$: A_i #number of insertions mapped to the i-th row.
2. $X_1, \ldots, X_{n/\log^5 n}$: X_i time needed for the A_i operations in the i-th row

Whp there are no overflows: $A_i < \mathcal{O}(\log^5 n)$ insertions per row
Rebuild time on fingerprint collision: $\mathcal{O}(\log^5 n)$
Number rebuilds per insertion is geometrically distributed and therefore at most $\mathcal{O}(\log n)$ whp (see Preliminaries).
Hence, $X_i < \mathcal{O}(\log^{11} n)$ whp.
We need to bound $X := \sum_i X_i$.
Consider $X/\Theta(\log^{11} n)$, the sum of independent $[0,1]$ random variables, with $\mu := \mathbb{E}[X/\Theta(\log^{11} n)] = \Theta(n^{1/2}/\log^{11} n)$:
By a Chernoff bound $X/\Theta(\log^{11} n)$, exceeds μ by at most $\mathcal{O}(\sqrt{\mu \log n})$ whp.

Deamortization

Lemma: Buffer capacity

Any batch of $\sqrt{n}$ insertions takes $\mathcal{O}(\sqrt{n})$ time **whp**.

Chernoff Bound

Consider a sum of independent $[0,1]$-valued random variables $X = \sum_{i=1}^{n} X_i$. Then $X \leqslant \mathbb{E}[X] + \mathcal{O}(\sqrt{\mathbb{E}[X] \log n})$ whp.

Proof

1. $A_1, \ldots, A_{n/\log^5 n}$: A_i #number of insertions mapped to the i-th row.
2. $X_1, \ldots, X_{n/\log^5 n}$: X_i time needed for the A_i operations in the i-th row

Whp there are no overflows: $A_i < \mathcal{O}(\log^5 n)$ insertions per row
Rebuild time on fingerprint collision: $\mathcal{O}(\log^5 n)$
Number rebuilds per insertion is geometrically distributed and therefore at most $\mathcal{O}(\log n)$ whp (see Preliminaries).
Hence, $X_i < \mathcal{O}(\log^{11} n)$ whp.
We need to bound $X := \sum_i X_i$.
Consider $X/\Theta(\log^{11} n)$, the sum of independent $[0,1]$ random variables, with $\mu := \mathbb{E}[X/\Theta(\log^{11} n)] = \Theta(n^{1/2}/\log^{11} n)$:
By a Chernoff bound $X/\Theta(\log^{11} n)$, exceeds μ by at most $\mathcal{O}(\sqrt{\mu \log n})$ whp.
Unfolding gives us that X exceeds its expectation by at most $\mathcal{O}(\log^{11}(n)\sqrt{\mu \log n}) = \mathcal{O}(n^{1/4} \log^6(n)) < \mathcal{O}(n^{1/2})$.

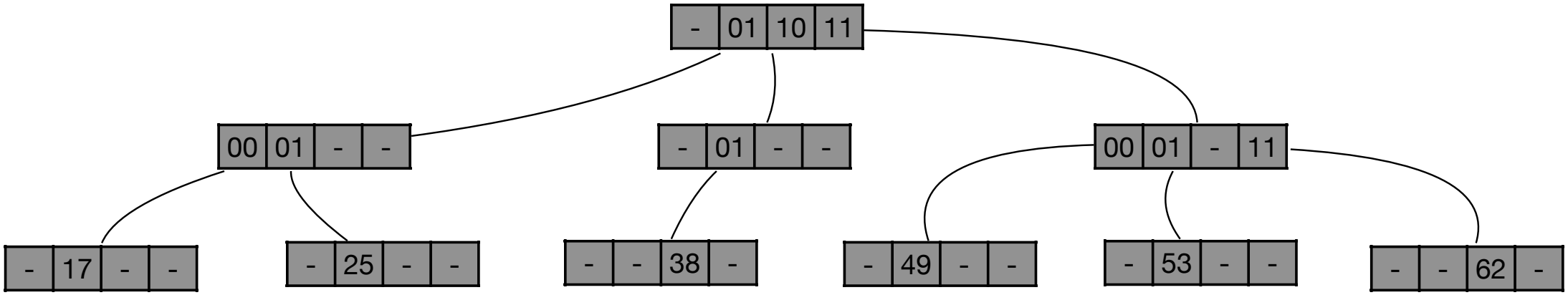
Deamortization

The buffer data structure

Use a **radix tree**, i.e. a fixed arity (max #children) search tree with evenly spaced pivots.
Maintain keys that need to be inserted / deleted in the radix tree.

Example: arity 4 radix tree for

$$\{17,25,38,49,53,62\} = \{01\ 00\ 01,\ 01\ 10\ 01,\ 10\ 01\ 10,\ 11\ 00\ 01,\ 11\ 01\ 01,\ 11\ 11\ 10\}$$

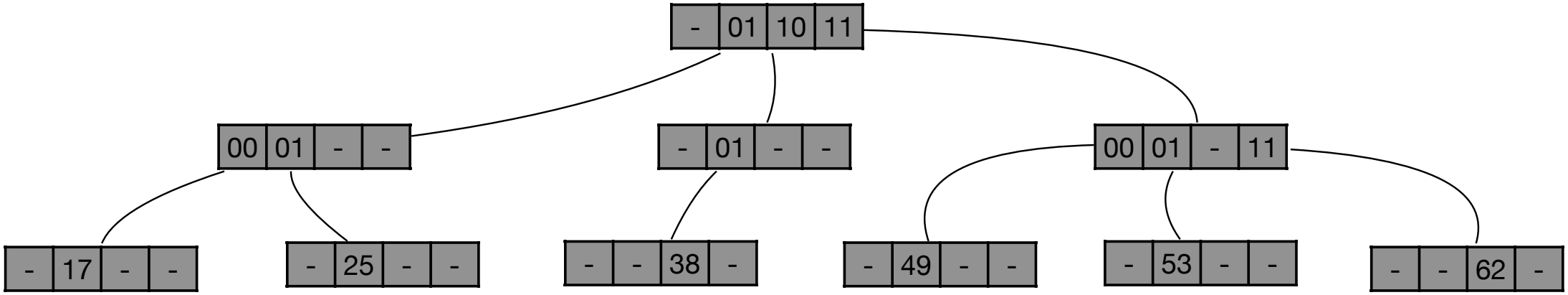


Correction: 25 is incorrectly placed in this figure

Deamortization

Choosing the arity parameter *a*

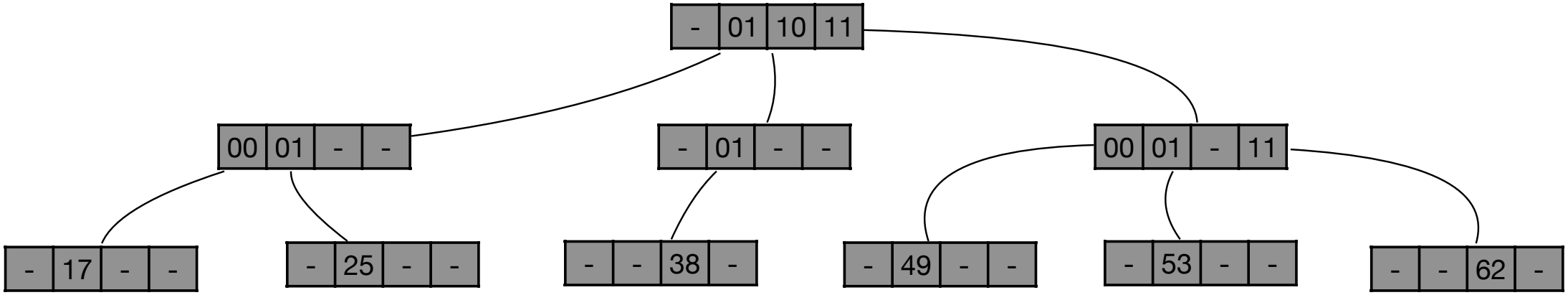
1. Choosing *a* too small makes the height *h* of the tree too large and therefore the queries too slow.



Deamortization

Choosing the arity parameter a

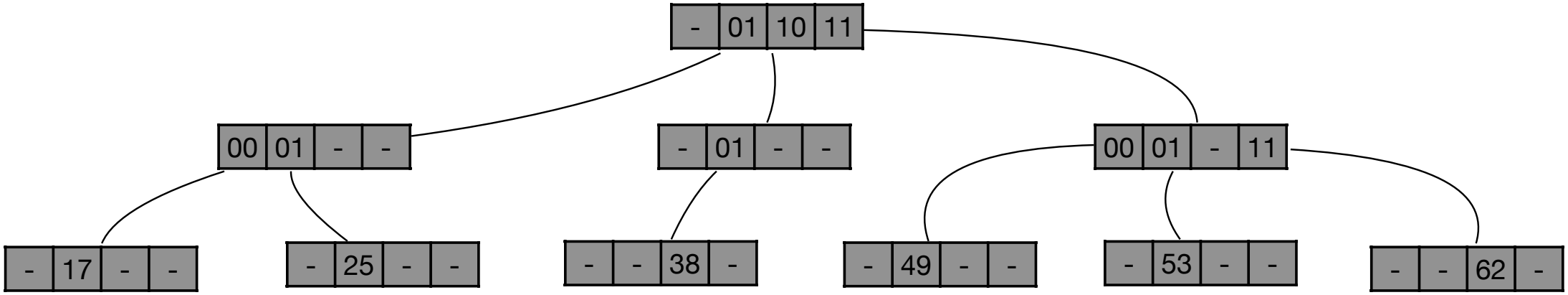
1. Choosing a too small makes the height h of the tree too large and therefore the queries too slow.
   We have $w = h \log a$, because each node takes away $\log a$ bits of the w bit key.
   Rearranging gives: $a = 2^{w/h} = (2^w)^{1/h} = n^{\Theta(1/h)}$.
   Hence, $h \in \mathcal{O}(1)$ needs $a \in \Omega(\text{poly}(n))$



Deamortization

Choosing the arity parameter a

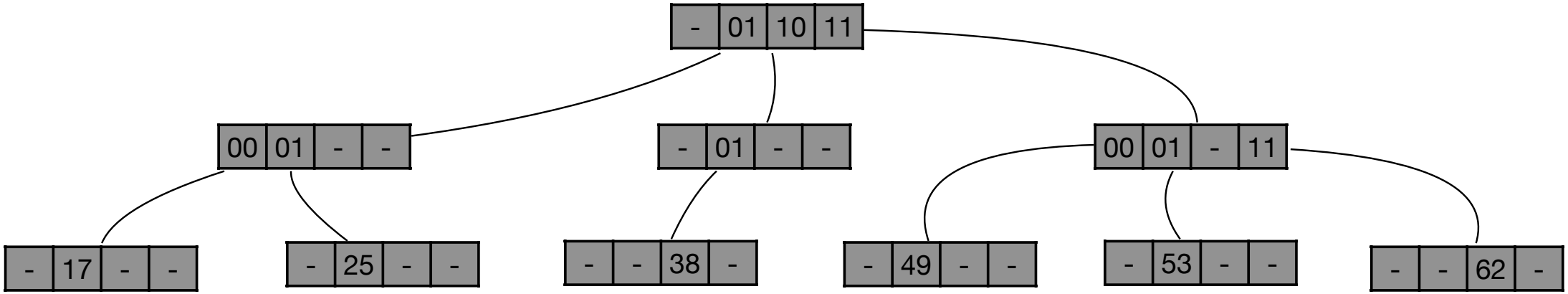
1. Choosing a too small makes the height h of the tree too large and therefore the queries too slow.
   We have $w = h \log a$, because each node takes away $\log a$ bits of the w bit key.
   Rearranging gives: $a = 2^{w/h} = (2^w)^{1/h} = n^{\Theta(1/h)}$.
   Hence, $h \in \mathcal{O}(1)$ needs $a \in \Omega(\text{poly}(n))$
2. Choosing a too large requires too much space:
   Each node stores a pointers of $\log n$ bits.
   Each of the $\sqrt{n}$ elements requires at most h internal nodes.
   This gives us at most $\sqrt{n}ha \log n$ bits.



Deamortization

Choosing the arity parameter a

1. Choosing a too small makes the height h of the tree too large and therefore the queries too slow.
   We have $w = h \log a$, because each node takes away $\log a$ bits of the w bit key.
   Rearranging gives: $a = 2^{w/h} = (2^w)^{1/h} = n^{\Theta(1/h)}$.
   Hence, $h \in \mathcal{O}(1)$ needs $a \in \Omega(\text{poly}(n))$
2. Choosing a too large requires too much space:
   Each node stores a pointers of $\log n$ bits.
   Each of the $\sqrt{n}$ elements requires at most h internal nodes.
   This gives us at most $\sqrt{n}ha \log n$ bits.
   Choosing $a = n^{\frac{1}{4}}$ gives $\mathcal{O}(n^{\frac{3}{4}} \log n)$ bits
   This is a lower order term in the overall space of at least $n \log n$ bits.



Overview

	Array	Partitioning	Left Justification	Query Router	Deamortization	Resizable
Query	$\mathcal{O}(n)$	$\mathcal{O}(\log^5 n)$	$\mathcal{O}(\log^5 n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Insert	$\mathcal{O}(n)$	$\mathcal{O}(\log^5 n)$ whp	$\mathcal{O}(1)$ whp	$\mathcal{O}(1)$ expected	$\mathcal{O}(1)$ whp	$\mathcal{O}(1)$ whp

$(1 + o(1))nw$ bits

$(1 + o(1))\bar{n}w$ bits

Resizing

Definition

We now need to keep track of different n.

- n is the current capacity of the hash table.
- $\bar{n}$ is the number of items in the hash table at any time.

Resizing

Definition

We now need to keep track of different n.

- n is the current capacity of the hash table.
- $\bar{n}$ is the number of items in the hash table at any time.

Goal

Space efficiency: $(1 + o(1))\bar{n}w$ instead of $(1 + o(1))nw$

Resizing

Definition

We now need to keep track of different n.

- n is the current capacity of the hash table.
- $\bar{n}$ is the number of items in the hash table at any time.

Goal

Space efficiency: $(1 + o(1))\bar{n}w$ instead of $(1 + o(1))nw$

Resizeability in two steps

1. We'll first show how to remain space-efficient when $\bar{n} \in [n/2, n]$.
2. Later we'll let $\bar{n}$ range more widely.

Resizing in $\bar{n} \in [n/2, n]$

Observation

The scaffolding (counters, query routers, buffer) needs $\mathcal{O}(n \log \log n)$ bits. Since, $\bar{n} \in [n/2, n]$ this is also $\mathcal{O}(\bar{n} \log \log \bar{n}) < o(\bar{n}w)$ bits, so still lower order.
We just need to dynamically resize the arrays.

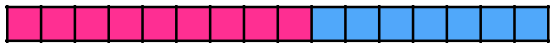
Resizing in $\bar{n} \in [n/2, n]$

Observation

The scaffolding (counters, query routers, buffer) needs $\mathcal{O}(n \log \log n)$ bits. Since, $\bar{n} \in [n/2, n]$ this is also $\mathcal{O}(\bar{n} \log \log \bar{n}) < o(\bar{n}w)$ bits, so still lower order.
We just need to dynamically resize the arrays.

Previously

Each row is an array of length $\log^5 n + \mathcal{O}(\log^3 n)$



Resizing in $\bar{n} \in [n/2, n]$

Observation

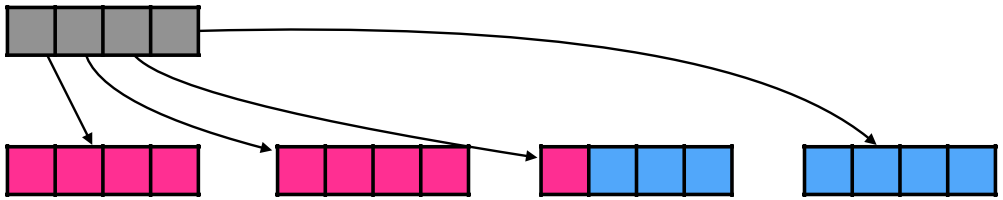
The scaffolding (counters, query routers, buffer) needs $\mathcal{O}(n \log\log n)$ bits. Since, $\bar{n} \in [n/2, n]$ this is also $\mathcal{O}(\bar{n} \log\log \bar{n}) < o(\bar{n}w)$ bits, so still lower order.
We just need to dynamically resize the arrays.

Previously

Each row is an array of length $\log^5 n + \mathcal{O}(\log^3 n)$

Now

Replace this array with an array of $\mathcal{O}(\log^{2.5} n)$ pointers, each pointing to an array of capacity $\log^{2.5} n$.



Resizing in $\bar{n} \in [n/2, n]$

Observation

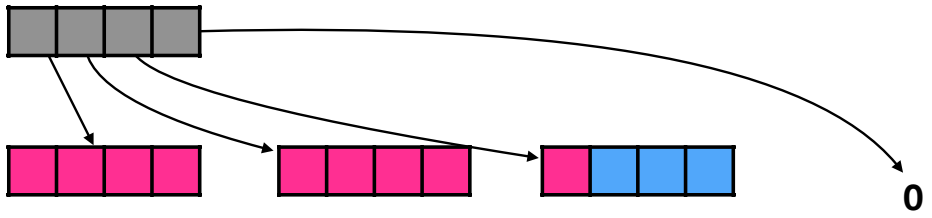
The scaffolding (counters, query routers, buffer) needs $\mathcal{O}(n \log\log n)$ bits. Since, $\bar{n} \in [n/2, n]$ this is also $\mathcal{O}(\bar{n} \log\log \bar{n}) < o(\bar{n}w)$ bits, so still lower order.
We just need to dynamically resize the arrays.

Previously

Each row is an array of length $\log^5 n + \mathcal{O}(\log^3 n)$

Now

Replace this array with an array of $\mathcal{O}(\log^{2.5} n)$ pointers, each pointing to an array of capacity $\log^{2.5} n$.
We only keep the non-empty arrays allocated.
That's why we kept the arrays left aligned!



Resizing in $\bar{n} \in [n/2, n]$

Observation

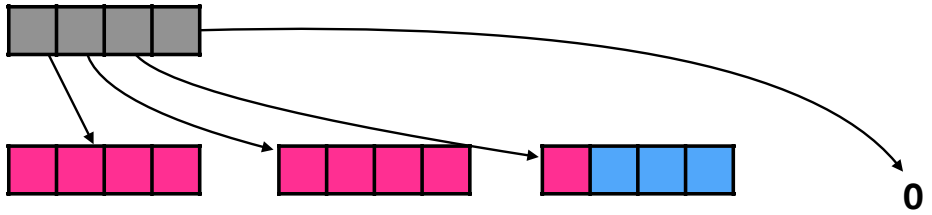
The scaffolding (counters, query routers, buffer) needs $\mathcal{O}(n \log\log n)$ bits. Since, $\bar{n} \in [n/2, n]$ this is also $\mathcal{O}(\bar{n} \log\log \bar{n}) < o(\bar{n}w)$ bits, so still lower order.
We just need to dynamically resize the arrays.

Previously

Each row is an array of length $\log^5 n + \mathcal{O}(\log^3 n)$

Now

Replace this array with an array of $\mathcal{O}(\log^{2.5} n)$ pointers, each pointing to an array of capacity $\log^{2.5} n$.
We only keep the non-empty arrays allocated.
That's why we kept the arrays left aligned!



Analysis

The space across all $n/\log^5 n$ rows is therefore:

1. at most $\bar{n}w$ bits for the full arrays

Resizing in $\bar{n} \in [n/2, n]$

Observation

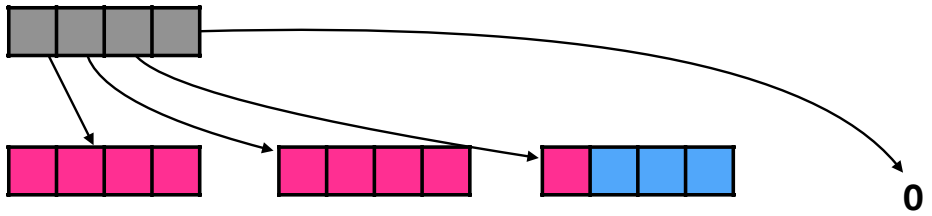
The scaffolding (counters, query routers, buffer) needs $\mathcal{O}(n \log\log n)$ bits. Since, $\bar{n} \in [n/2, n]$ this is also $\mathcal{O}(\bar{n} \log\log \bar{n}) < o(\bar{n}w)$ bits, so still lower order.
We just need to dynamically resize the arrays.

Previously

Each row is an array of length $\log^5 n + \mathcal{O}(\log^3 n)$

Now

Replace this array with an array of $\mathcal{O}(\log^{2.5} n)$ pointers, each pointing to an array of capacity $\log^{2.5} n$.
We only keep the non-empty arrays allocated.
That's why we kept the arrays left aligned!



Analysis

The space across all $n/\log^5 n$ rows is therefore:

1. at most $\bar{n}w$ bits for the full arrays
2. in each row at most one partially filled array using $w \log^{2.5} n$ bits

Resizing in $\bar{n} \in [n/2, n]$

Observation

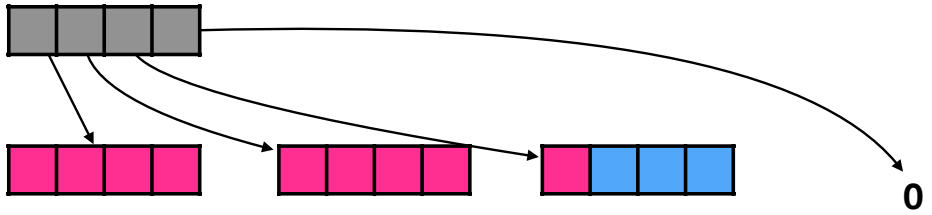
The scaffolding (counters, query routers, buffer) needs $\mathcal{O}(n \log\log n)$ bits. Since, $\bar{n} \in [n/2, n]$ this is also $\mathcal{O}(\bar{n} \log\log \bar{n}) < o(\bar{n}w)$ bits, so still lower order.
We just need to dynamically resize the arrays.

Previously

Each row is an array of length $\log^5 n + \mathcal{O}(\log^3 n)$

Now

Replace this array with an array of $\mathcal{O}(\log^{2.5} n)$ pointers, each pointing to an array of capacity $\log^{2.5} n$.
We only keep the non-empty arrays allocated.
That's why we kept the arrays left aligned!



Analysis

The space across all $n/\log^5 n$ rows is therefore:

1. at most $\bar{n}w$ bits for the full arrays
2. in each row at most one partially filled array using $w \log^{2.5} n$ bits
3. in each row $\mathcal{O}(w \log^{2.5} n)$ bits for pointers

Resizing in $\bar{n} \in [n/2, n]$

Observation

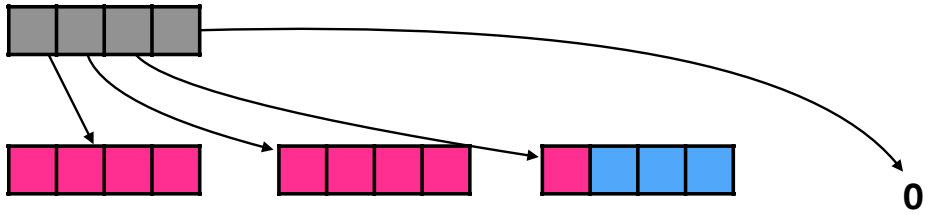
The scaffolding (counters, query routers, buffer) needs $\mathcal{O}(n \log\log n)$ bits. Since, $\bar{n} \in [n/2, n]$ this is also $\mathcal{O}(\bar{n} \log\log \bar{n}) < o(\bar{n}w)$ bits, so still lower order.
We just need to dynamically resize the arrays.

Previously

Each row is an array of length $\log^5 n + \mathcal{O}(\log^3 n)$

Now

Replace this array with an array of $\mathcal{O}(\log^{2.5} n)$ pointers, each pointing to an array of capacity $\log^{2.5} n$.
We only keep the non-empty arrays allocated.
That's why we kept the arrays left aligned!



Analysis

The space across all $n/\log^5 n$ rows is therefore:

1. at most $\bar{n}w$ bits for the full arrays
2. in each row at most one partially filled array using $w \log^{2.5} n$ bits
3. in each row $\mathcal{O}(w \log^{2.5} n)$ bits for pointers
4. $\mathcal{O}(n \log\log n)$ bits for the scaffolding

Resizing in $\bar{n} \in [n/2, n]$

Observation

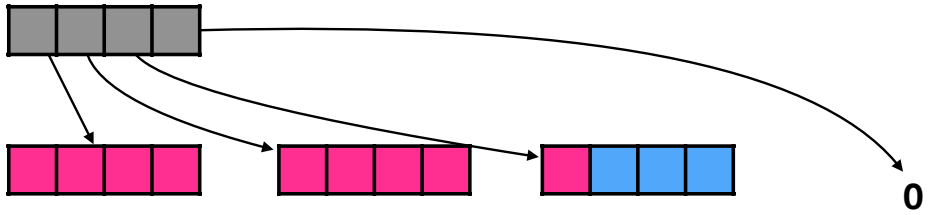
The scaffolding (counters, query routers, buffer) needs $\mathcal{O}(n \log\log n)$ bits. Since, $\bar{n} \in [n/2, n]$ this is also $\mathcal{O}(\bar{n} \log\log \bar{n}) < o(\bar{n}w)$ bits, so still lower order.
We just need to dynamically resize the arrays.

Previously

Each row is an array of length $\log^5 n + \mathcal{O}(\log^3 n)$

Now

Replace this array with an array of $\mathcal{O}(\log^{2.5} n)$ pointers, each pointing to an array of capacity $\log^{2.5} n$.
We only keep the non-empty arrays allocated.
That's why we kept the arrays left aligned!



Analysis

The space across all $n/\log^5 n$ rows is therefore:

1. at most $\bar{n}w$ bits for the full arrays
2. in each row at most one partially filled array using $w \log^{2.5} n$ bits
3. in each row $\mathcal{O}(w \log^{2.5} n)$ bits for pointers
4. $\mathcal{O}(n \log\log n)$ bits for the scaffolding

In total $\bar{n}w + \mathcal{O}(n/\log^{1.5} n) + \mathcal{O}(n \log\log n)$ bits.
This is space efficient as $n \in \Theta(\bar{n})$

Resizing in full range

$\bar{n}$ in the range $[n/2, n]$

Our method allows us to be space efficient in this range because the scaffolding requires $\mathcal{O}(n \log \log n)$ bits.

Full range resizing

What happens when the table gets bigger or smaller than this range?
Standard deamortization works:

1. When the table approaches overfilling, build scaffolding for the next size up in a deamortized fashion
2. When it hits the max size, new insertions go into new table, and two items get moved from old table to new
3. Space overhead is now twice the scaffolding, still negligible

Possible Exam Questions

- How did we proof that no row overflows whp?
- Why did we use left justification to achieve constant insertion time (instead of e.g. a list that maintains free locations)?
- What is the biggest downside of using the query router?
- What modifications were needed to make the B-Tree constant time?
- How did we proof that fingerprint collisions are unlikely?
- How do all the operations work with the deamortization buffer?
- How did we construct the buffer itself as a constant time hash table? Why can we not use this technique for the entire hash table?
- What tradeoff is involved in choosing the arity of the radix tree?
- How did we achieve space efficiency in terms of $\bar{n}$?
- Considering the final hash table, why is partitioning still needed?