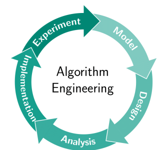
Predecessor Data Structures: Lower Bounds and Beating Them

Stefan Walzer, Ragnar Groot Koerkamp, Stefan Hermann | compiled on 15. Juni 2026



KIT

1. Lower Bounds

2. Predecessor Search for Structured Inputs

Uniform Random Input Data
Learned Index Structures

Lower Bounds for Static Predecessor Search

Assume $w = \log u$.

Fusion Trees vs. vEB Trees: Which is better?

[1] Stronger and more detailed version of theorem in “Time-space trade-offs for predecessor search” [Pătraşcu Thorup, 2006].

[2] *Much* weaker! Trivial for $w = \operatorname{poly}\log n$. Non-trivial only for $\log w = \omega(\log\log n)$, e.g. $w = 2^{\sqrt{\log n}}$. But this is all we can prove here.

Lower Bounds for Static Predecessor Search

Assume $w = \log u$.

Fusion Trees vs. vEB Trees: Which is better?

FT: $\mathcal{O}(\log_w n)$ better when $\log n \ll \log^2 w$

vEB: $\mathcal{O}(\log w)$ better when $\log n \gg \log^2 w$

[1] Stronger and more detailed version of theorem in "Time-space trade-offs for predecessor search" [Pătraşcu Thorup, 2006].

[2] *Much* weaker! Trivial for $w = \text{poly}\log n$. Non-trivial only for $\log w = \omega(\log\log n)$, e.g. $w = 2^{\sqrt{\log n}}$. But this is all we can prove here.

Lower Bounds for Static Predecessor Search

Assume $w = \log u$.

Fusion Trees vs. vEB Trees: Which is better?

FT: $\mathcal{O}(\log_w n)$ better when $\log n \ll \log^2 w$
vEB: $\mathcal{O}(\log w)$ better when $\log n \gg \log^2 w$

Can min(vEB,FT) be beaten? // static case

- **YES**, if $n = \Theta(2^w)$ // use bitvector
- **YES**, if we waste lots of space // use bitvector
- **YES**, we can get: $\mathcal{O}\left(\frac{\log w}{\log \frac{\log w}{\log\log n}}\right)$ // don't ask

[1] Stronger and more detailed version of theorem in "Time-space trade-offs for predecessor search" [Pătraşcu Thorup, 2006].
[2] *Much* weaker! Trivial for $w = \text{poly}\log n$. Non-trivial only for $\log w = \omega(\log\log n)$, e.g. $w = 2^{\sqrt{\log n}}$. But this is all we can prove here.

Lower Bounds for Static Predecessor Search

Assume $w = \log u$.

Fusion Trees vs. vEB Trees: Which is better?

FT: $\mathcal{O}(\log_w n)$ better when $\log n \ll \log^2 w$
vEB: $\mathcal{O}(\log w)$ better when $\log n \gg \log^2 w$

Can min(vEB,FT) be beaten? // static case

- **YES**, if $n = \Theta(2^w)$ // use bitvector
- **YES**, if we waste lots of space // use bitvector
- **YES**, we can get: $\mathcal{O}\left(\frac{\log w}{\log \frac{\log w}{\log\log n}}\right)$ // don't ask

Theorem[1]: min(vEB,FT) only barely beatable

Every static predecessor data structure with space $\mathcal{O}(n)$ has query time

$$\Omega\left(\min\left(\frac{\log w}{\log\log w}, \frac{\log n}{\log w}\right)\right).$$

The "$\log\log w$" vanishes if $w = \log^{\mathcal{O}(1)} n \rightsquigarrow$ then vEB is optimal.

[1] Stronger and more detailed version of theorem in "Time-space trade-offs for predecessor search" [Pătraşcu Thorup, 2006].
[2] *Much* weaker! Trivial for $w = \text{poly}\log n$. Non-trivial only for $\log w = \omega(\log\log n)$, e.g. $w = 2^{\sqrt{\log n}}$. But this is all we can prove here.

Lower Bounds for Static Predecessor Search

Assume $w = \log u$.

Fusion Trees vs. vEB Trees: Which is better?

FT: $\mathcal{O}(\log_w n)$ better when $\log n \ll \log^2 w$
vEB: $\mathcal{O}(\log w)$ better when $\log n \gg \log^2 w$

Can min(vEB,FT) be beaten? // static case

- **YES**, if $n = \Theta(2^w)$ // use bitvector
- **YES**, if we waste lots of space // use bitvector
- **YES**, we can get: $\mathcal{O}\left(\frac{\log w}{\log \frac{\log w}{\log\log n}}\right)$ // don't ask

Theorem[1]: min(vEB,FT) only barely beatable

Every static predecessor data structure with space $\mathcal{O}(n)$ has query time

$$\Omega\left(\min\left(\frac{\log w}{\log\log w}, \frac{\log n}{\log w}\right)\right).$$

The "$\log\log w$" vanishes if $w = \log^{\mathcal{O}(1)} n \rightsquigarrow$ then vEB is optimal.

Theorem[2] (simplified)

Every static predecessor data structure with space $\mathcal{O}(n)$ has query time

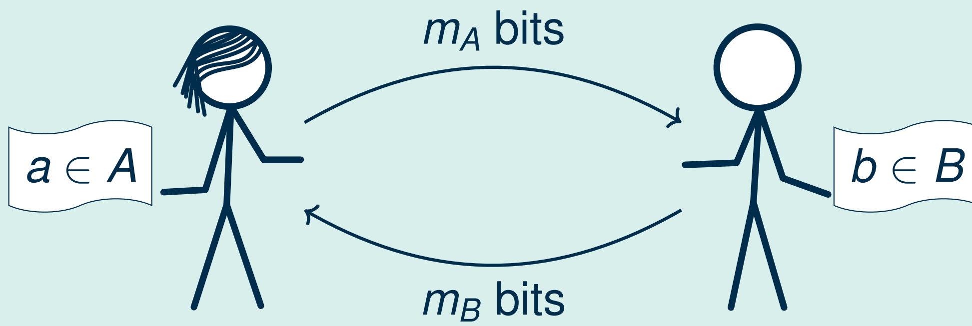
$$\Omega\left(\min\left(\frac{\log w}{\log\log n}, \frac{\log n}{\log w}\right)\right)$$

[1] Stronger and more detailed version of theorem in "Time-space trade-offs for predecessor search" [Pătraşcu Thorup, 2006].
[2] *Much* weaker! Trivial for $w = \text{poly}\log n$. Non-trivial only for $\log w = \omega(\log\log n)$, e.g. $w = 2^{\sqrt{\log n}}$. But this is all we can prove here.

Communication Complexity

Setting

Given function $f : A \times B \to R$ and $(a, b) \in A \times B$.



Definition: Communication Complexity of f

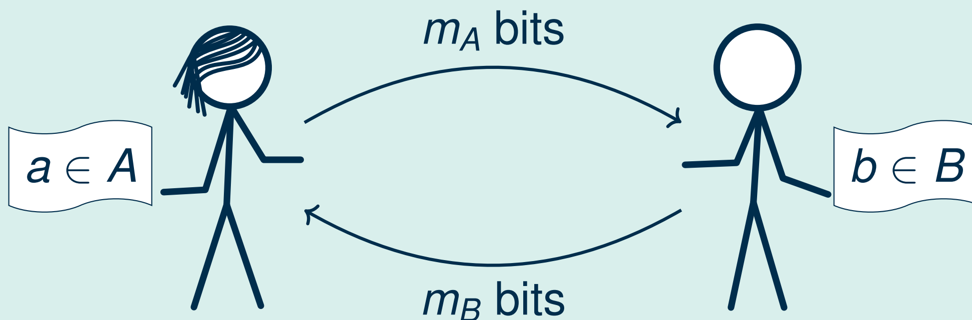
Number of messages until $f(a, b)$ is known (to Alice or Bob)

- $(a, b) \in A \times B$ is *worst case*
- optimal protocol is fixed in advanced

Communication Complexity

Setting

Given function $f : A \times B \to R$ and $(a, b) \in A \times B$.



Definition: Communication Complexity of f

Number of messages until $f(a, b)$ is known (to Alice or Bob)

- $(a, b) \in A \times B$ is *worst case*
- optimal protocol is fixed in advanced

Remark

Interesting only if $m_A \ll \log_2 |A|$ and $m_B \ll \log_2 |B|$

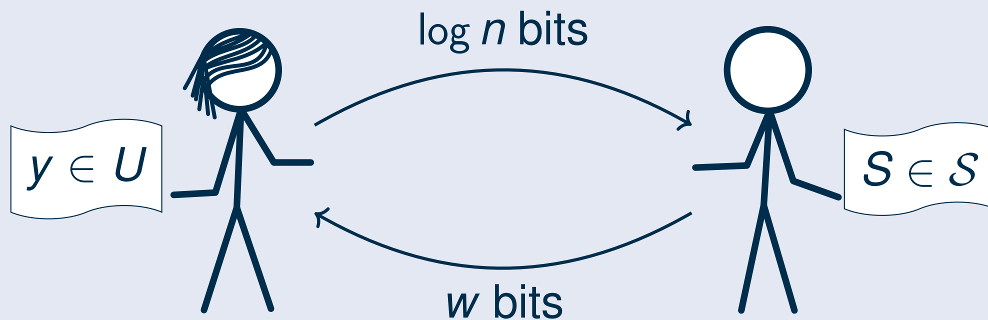
// otherwise players can share their inputs in $\mathcal{O}(1)$ rounds

Communication Complexity LB implies Cell Probe LB

Let $U = \{0, \ldots, u-1\}$, $\mathcal{S} = \{S \subseteq U \mid |S| = n\}$ and $q : U \times \mathcal{S} \to U \cup \{\text{None}\}$ such that $q(y, S)$ is pred(y) for input S.

Theorem (Communication Complexity LB) // todo

For $m_A = \log n$ and $m_B = w$ the communication complexity of q is $t = \Omega\Big(\min\big(\frac{\log w}{\log\log n}, \frac{\log n}{\log w}\big)\Big)$.

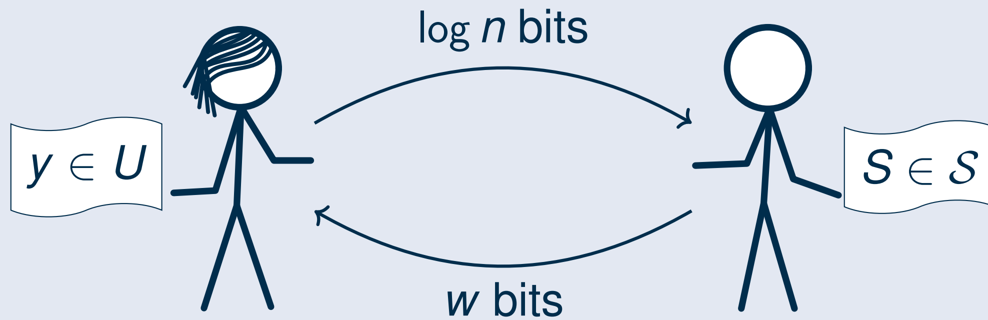


Communication Complexity LB implies Cell Probe LB

Let $U = \{0, \ldots, u-1\}$, $\mathcal{S} = \{S \subseteq U \mid |S| = n\}$ and $q : U \times \mathcal{S} \to U \cup \{\text{None}\}$ such that $q(y, S)$ is pred(y) for input S.

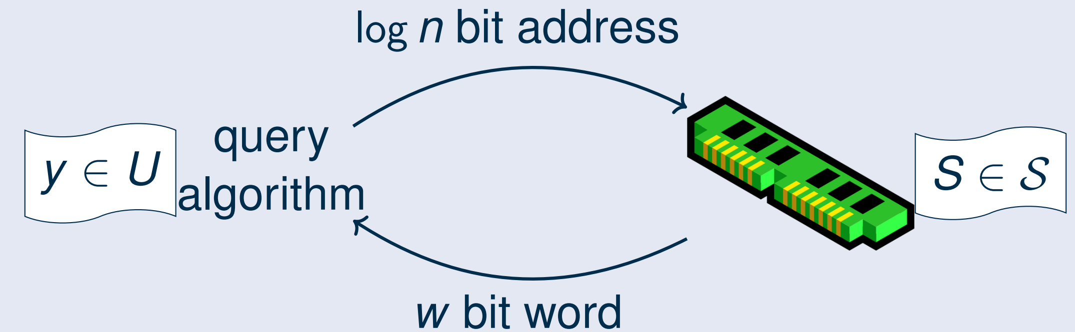
Theorem (Communication Complexity LB) // todo

For $m_A = \log n$ and $m_B = w$ the communication complexity of q is $t = \Omega\Big(\min\big(\frac{\log w}{\log\log n}, \frac{\log n}{\log w}\big)\Big)$.



Corollary (Cell Probe LB)

Any query algorithm for the static predecessor problem must perform $t/2$ memory accesses in the worst case.

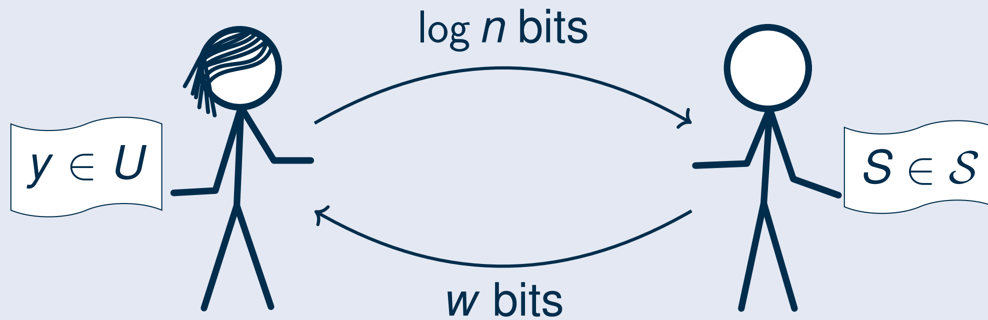


Communication Complexity LB implies Cell Probe LB

Let $U = \{0, \dots, u-1\}$, $\mathcal{S} = \{S \subseteq U \mid |S| = n\}$ and $q : U \times \mathcal{S} \to U \cup \{\text{None}\}$ such that $q(y, S)$ is pred(y) for input S.

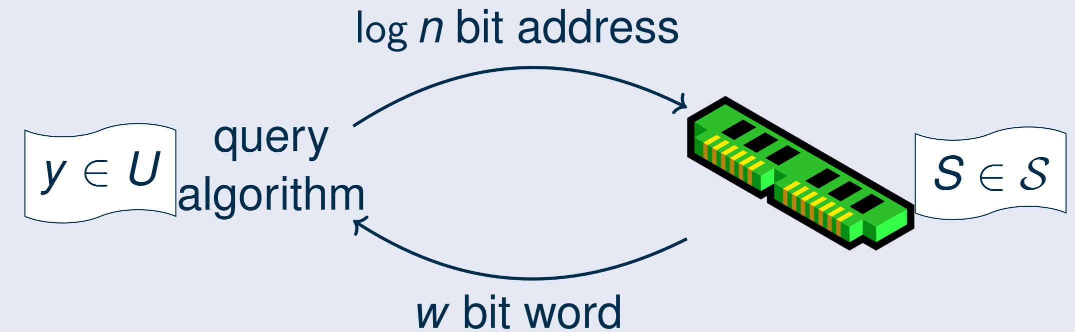
Theorem (Communication Complexity LB) // todo

For $m_A = \log n$ and $m_B = w$ the communication complexity of q is $t = \Omega\left(\min\left(\frac{\log w}{\log\log n}, \frac{\log n}{\log w}\right)\right)$.



Corollary (Cell Probe LB)

Any query algorithm for the static predecessor problem must perform $t/2$ memory accesses in the worst case.



Corollary (Time Lower Bound)

Any query algorithm for the static predecessor problem has worst case running time $\Omega(t)$.

Comm. Compl. Bound via *Round Elimination* (Sketch)

Setting

Consider *instances* (N, W, y, S) where $N \leq n$, $W \leq w$, $y \in \{0,1\}^W$ is uniformly random and $S \subseteq \{0,1\}^W$ with $|S| = N$ is random.

Comm. Compl. Bound via *Round Elimination* (Sketch)

Setting

Consider *instances* (N, W, y, S) where $N \le n$, $W \le w$, $y \in \{0,1\}^W$ is uniformly random and $S \subseteq \{0,1\}^W$ with $|S| = N$ is random.

Lemma: "Eliminate" an Alice Message

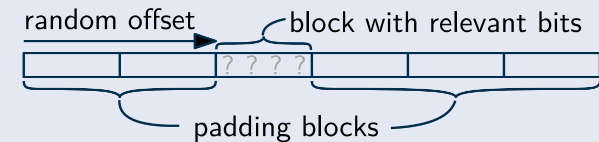
Every instance (N, W, y, S) can be "embedded" into an instance (N, W', y', S') s.t.

- $W' = kW$ for $k = \mathcal{O}(\log^2 n)$
- if Bob starts, the communication complexity is that of the embedded instance
- if Alice starts, her first message is "probably useless"

Proof Sketch: Eliminating Alice's Message

Generate new instance using

- let $l \sim \text{Unif}([k])$
- let $p_\ell \sim \text{Unif}(\{0,1\}^{(l-1)W})$
- let $p_r \sim \text{Unif}(\{0,1\}^{(k-l)W})$
- let $y' = p_\ell \circ y \circ p_r$
- let $S' = \{p_\ell \circ x \circ 0^{(k-l)W} \mid x \in S\}$ // see footnote [a]



- when Bob speaks he can reveal l // note $\log l \le \log w \le w$
  - $\Rightarrow$ both parties can discard the padding
- Alice's first message contains, *in expectation*, m_A/k bits about x.
  - $\Rightarrow$ "likely useless" since $\frac{m_A}{k} = \frac{\log n}{k} \ll 1$.

[a] $S' = \{p_\ell \circ x \circ p_r \mid x \in S\}$ would fail: Alice could send something clever. Can you see why?

Comm. Compl. Bound via *Round Elimination* (Sketch)

Lemma: "Eliminate" an Alice Message

Every instance (N, W, y, S) can be "embedded" into an instance (N, W', y', S') s.t.

- $W' = kW$ for $k = \mathcal{O}(\log^2 n)$
- if Bob starts, the communication complexity is that of the embedded instance
- if Alice starts, her first message is "probably useless"

Lemma: "Eliminate" a Bob Message

Every instance (N, W, y, S) can be "embedded" into an instance (N', W', y', S') s.t.

- $N' = kN$ for $k = \mathcal{O}(w^2)$
- $W' = W + \mathcal{O}(\log w)$
- if Alice starts, the communication complexity is that of the embedded instance
- if Bob starts, his first message is "probably useless"

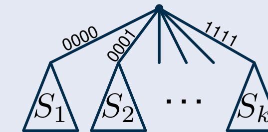
Setting

Consider *instances* (N, W, y, S) where $N \leq n$, $W \leq w$, $y \in \{0,1\}^W$ is uniformly random and $S \subseteq \{0,1\}^W$ with $|S| = N$ is random.

Proof Sketch: Eliminating Bob's Message

Generate new instance:

- let $(y_1, S_1), \dots (y_k, S_k)$ be independent and distributed like (y, S)
- let $I \sim \mathrm{Unif}([k])$ // assume k is power of 2, hence $I \sim \mathrm{Unif}(\{0,1\}^{\log k})$
- let $y' = I \circ y_I$
- let $S = \bigcup_{i \in [k]} \{ i \circ y \mid y \in S_i \}$



- when Alice speaks she can reveal I // note $\log I \leq \log n$
  - $\Rightarrow$ then both parties can focus on (x_I, S_I)
- Bob's first message contains, *in expectation*, m_B/k bits about S_I.
  - $\Rightarrow$ "likely useless" since $\frac{m_B}{k} = \frac{w}{k} \ll 1$.

Comm. Compl. Bound via *Round Elimination* (Sketch)

Lemma: "Eliminate" an Alice Message

Every instance (N, W, y, S) can be "embedded" into an instance (N, W', y', S') s.t.

- $W' = kW$ for $k = \mathcal{O}(\log^2 n)$
- if Bob starts, the communication complexity is that of the embedded instance
- if Alice starts, her first message is "probably useless"

Lemma: "Eliminate" a Bob Message

Every instance (N, W, y, S) can be "embedded" into an instance (N', W', y', S') s.t.

- $N' = kN$ for $k = \mathcal{O}(w^2)$
- $W' = W + \mathcal{O}(\log w)$
- if Alice starts, the communication complexity is that of the embedded instance
- if Bob starts, his first message is "probably useless"

Theorem: Iterated Round Elimination

There exists an instance with $W \leq w$, $N \leq n$ that requires at least t messages with

$$t = \Theta\Big(\min\big(\frac{\log w}{\log\log n}, \frac{\log n}{\log w}\big)\Big).$$

Setting

Consider *instances* (N, W, y, S) where $N \leq n$, $W \leq w$, $y \in \{0,1\}^W$ is uniformly random and $S \subseteq \{0,1\}^W$ with $|S| = N$ is random.

Proof Sketch: Iterated Round Elimination

- Begin with instance with $N = W = 1$ that requires ≥ 1 message.
- alternatingly apply the lemmas for $2t$ steps
  - $\Rightarrow$ first t message pairs useless for base instance
  - $\Rightarrow$ communication complexity $\geq t$
- Final instance must respect $N \leq n$ and $W \leq w$:
  - $N \approx (w^2)^t \overset{!}{\leq} n$, so we need $t = \mathcal{O}(\frac{\log n}{\log w})$
  - $W \approx (\log^2 n)^t \overset{!}{\leq} w$, so we need $t = \mathcal{O}(\frac{\log w}{\log\log n})$

KIT

1. Lower Bounds

2. Predecessor Search for Structured Inputs

Uniform Random Input Data
Learned Index Structures

Can we beat Binary Search on Random Data?

Context: Sorted array $A[1..n]$ of *uniformly random* numbers from $[0, 1]$.
Input: $p \in [0, 1]$. // not random, but chosen oblivious of the random numbers
Output: Rank $R = |\{i \in [n] \mid A[i] < p\}|$.

Can we beat Binary Search on Random Data?

Context: Sorted array $A[1..n]$ of *uniformly random* numbers from $[0, 1]$.
Input: $p \in [0, 1]$. // not random, but chosen oblivious of the random numbers
Output: Rank $R = |\{i \in [n] \mid A[i] < p\}|$.

Intuition: Interpolation Search achieves $\mathcal{O}(\log \log n)$ expected time

Probabilistic observations:

- $R \sim \text{Bin}(n, p)$
- $\mathbb{E}[R] = np$
- $\sigma^2 = \text{Var}(R) = np(1 - p) \leq n$
  $\Rightarrow \sigma = \sqrt{n}$

Therefore

- The guess $R \approx \mathbb{E}[np]$ is typically off by only $\pm\sqrt{n}$
  $\hookrightarrow$ recurse on subinterval of size $\mathcal{O}(\sqrt{n})$
- The sequence $n, \sqrt{n}, \sqrt{\sqrt{n}}, \ldots$ reaches $\mathcal{O}(1)$ after $\mathcal{O}(\log \log n)$ steps.

$\mathbb{E}[R] - \mathcal{O}(\sigma)$ $\mathcal{O}(\sqrt{n})$ $\mathbb{E}[R] + \mathcal{O}(\sigma)$

A:	0	?	?	?	?	?	0.19	?	?	?	?	?	?	0.33	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	1

$p = 0.3$ in here?

Interpolation Search: Algorithm and Analysis

Algorithm $\text{interpRank}_a^b(x, A[1..n] \in [a,b]^n)$**:**

- **if** $n = 0$ *or* $x = a$ **then return** 0
- let $p \leftarrow \frac{x-a}{b-a} \in (0,1]$ and $i = \lceil pn \rceil \in \{1,\dots,n\}$
- **if** $A[i] < x$ **then**
  - **return** $i + \text{interpRank}_{A[i]}^b(x, A[i+1..n])$
- **else**
  - **return** $\text{interpRank}_a^{A[i]}(x, A[1..i-1])$

Interpolation Search: Algorithm and Analysis

Algorithm $\text{interpRank}_a^b(x, A[1..n] \in [a,b]^n)$**:**

- **if** $n = 0$ *or* $x = a$ **then return** 0
- let $p \leftarrow \frac{x-a}{b-a} \in (0,1]$ and $i = \lceil pn \rceil \in \{1, \dots, n\}$
- **if** $A[i] < x$ **then**
  - **return** $i + \text{interpRank}_{A[i]}^b(x, A[i+1..n])$
- **else**
  - **return** $\text{interpRank}_a^{A[i]}(x, A[1..i-1])$

Theorem

Let T be the running time of a call to interpRank_a^b where A arises by sorting random $X_1, \dots, X_n \sim \text{Unif}([a,b])$ and $x \in [a,b]$. Let $p = \frac{x-a}{b-a} \in [0,1]$ and $\tilde{p} = \min(p, 1-p)$. Then

$$\mathbb{E}[T] = \mathcal{O}(\max(1, \log\log \tilde{p}n)).$$

Interpolation Search: Algorithm and Analysis

Algorithm interpRank$_a^b(x, A[1..n] \in [a,b]^n)$**:**

- **if** $n = 0$ *or* $x = a$ **then return** 0
- let $p \leftarrow \frac{x-a}{b-a} \in (0,1]$ and $i = \lceil pn \rceil \in \{1,\dots,n\}$
- **if** $A[i] < x$ **then**
  - **return** $i +$ interpRank$_{A[i]}^b(x, A[i+1..n])$
- **else**
  - **return** interpRank$_a^{A[i]}(x, A[1..i-1])$

Theorem

Let T be the running time of a call to interpRank$_a^b$ where A arises by sorting random $X_1,\dots,X_n \sim \mathrm{Unif}([a,b])$ and $x \in [a,b]$. Let $p = \frac{x-a}{b-a} \in [0,1]$ and $\tilde{p} = \min(p, 1-p)$. Then

$$\mathbb{E}[T] = \mathcal{O}(\max(1, \log\log \tilde{p}n)).$$

Corollary: $\mathbb{E}[T] = \mathcal{O}(\log\log n)$.

Interpolation Search: Algorithm and Analysis

Algorithm $\text{interpRank}_a^b(x, A[1..n] \in [a,b]^n)$**:**

- **if** $n = 0$ *or* $x = a$ **then return** 0
- let $p \leftarrow \frac{x-a}{b-a} \in (0,1]$ and $i = \lceil pn \rceil \in \{1,\dots,n\}$
- **if** $A[i] < x$ **then**
  - **return** $i + \text{interpRank}_{A[i]}^b(x, A[i+1..n])$
- **else**
  - **return** $\text{interpRank}_a^{A[i]}(x, A[1..i-1])$

Theorem

Let T be the running time of a call to interpRank_a^b where A arises by sorting random $X_1, \dots, X_n \sim \text{Unif}([a,b])$ and $x \in [a,b]$. Let $p = \frac{x-a}{b-a} \in [0,1]$ and $\tilde{p} = \min(p, 1-p)$. Then

$$\mathbb{E}[T] = \mathcal{O}(\max(1, \log\log \tilde{p}n)).$$

Corollary: $\mathbb{E}[T] = \mathcal{O}(\log\log n)$.

Proof Strategy

- **Case 1:** $\tilde{p}n = \mathcal{O}(1)$. // includes $\tilde{p}n = o(1)$ when $\log\log \tilde{p}n < 0$
- **Case 2:** $\tilde{p}n = \omega(1)$.

  For convenience assume $pn \in \mathbb{N}$ and $(a,b) = (0,n)$. Then $i = x = pn$. Let $\tilde{p}'$ and n' be the values of $\tilde{p}$ and n in the recursive call. We show:
  - Step 1: $\tilde{p}'n' \le \tilde{p}n$ // always
  - Step 2: If $A[\tilde{p}n] \in \tilde{p}n \pm \mathcal{O}(\sqrt{\tilde{p}n})$ then $\tilde{p}'n' \le \mathcal{O}(\sqrt{\tilde{p}n})$.
  - Step 3: The condition of Step 2 holds with probability $1-\epsilon$.

  Most iterations reduce $\tilde{p}n$ to $\sqrt{\tilde{p}n}$. The remaining iterations "don't hurt".
  $\Rightarrow \mathbb{E}[T] = \log\log \tilde{p}n$.

A : array with positions 1, $i = pn$, n; $\in [0,n]^n$

assume without loss of generality: $\tilde{p} = p$.

Interpolation Search: Algorithm and Analysis

Algorithm $\text{interpRank}_a^b(x, A[1..n] \in [a,b]^n)$**:**

- **if** $n = 0$ *or* $x = a$ **then return** 0
- let $p \leftarrow \frac{x-a}{b-a} \in (0,1]$ and $i = \lceil pn \rceil \in \{1, \dots, n\}$
- **if** $A[i] < x$ **then**
  - **return** $i + \text{interpRank}_{A[i]}^b(x, A[i+1..n])$
- **else**
  - **return** $\text{interpRank}_a^{A[i]}(x, A[1..i-1])$

Theorem

Let T be the running time of a call to interpRank_a^b where A arises by sorting random $X_1, \dots, X_n \sim \text{Unif}([a,b])$ and $x \in [a,b]$. Let $p = \frac{x-a}{b-a} \in [0,1]$ and $\tilde{p} = \min(p, 1-p)$. Then

$$\mathbb{E}[T] = \mathcal{O}(\max(1, \log\log \tilde{p}n)).$$

Corollary: $\mathbb{E}[T] = \mathcal{O}(\log\log n)$.

Proof Strategy

- **Case 1:** $\tilde{p}n = \mathcal{O}(1)$. // includes $\tilde{p}n = o(1)$ when $\log\log \tilde{p}n < 0$
- **Case 2:** $\tilde{p}n = \omega(1)$.

  For convenience assume $pn \in \mathbb{N}$ and $(a,b) = (0,n)$. Then $i = x = pn$.
  Let $\tilde{p}'$ and n' be the values of $\tilde{p}$ and n in the recursive call. We show:
  - Step 1: $\tilde{p}'n' \leq \tilde{p}n$ // always
  - Step 2: If $A[\tilde{p}n] \in \tilde{p}n \pm \mathcal{O}(\sqrt{\tilde{p}n})$ then $\tilde{p}'n' \leq \mathcal{O}(\sqrt{\tilde{p}n})$.
  - Step 3: The condition of Step 2 holds with probability $1 - \epsilon$.

  Most iterations reduce $\tilde{p}n$ to $\sqrt{\tilde{p}n}$. The remaining iterations “don't hurt”.
  $\Rightarrow \mathbb{E}[T] = \log\log \tilde{p}n$.

$A :$ [array with positions 1, $i = pn$, n] $\in [0,n]^n$

assume without loss of generality: $\tilde{p} = p$.

Case 1: $pn = \mathcal{O}(1)$.

For $pn \leq 1$ the algorithm does linear search:

- We get $i = 1$ and recurse on $A[2..n]$ (with $p'n' < pn$) or are done.
- If the result is R we take time $T = \mathcal{O}(R)$.
- Since $R \sim \text{Bin}(n,p)$ we have $\mathbb{E}[R] = np \leq 1$ so $\mathbb{E}[T] = \mathcal{O}(1)$. ✓

For larger $pn = \mathcal{O}(1)$ the same idea works.

Interpolation Search: Algorithm and Analysis

Algorithm interpRank$_a^b(x, A[1..n] \in [a,b]^n)$**:**

```
if n = 0 or x = a then return 0
let p ← (x−a)/(b−a) ∈ (0,1] and i = ⌈pn⌉ ∈ {1,...,n}
if A[i] < x then
    return i + interpRank^b_{A[i]}(x, A[i+1..n])
else
    return interpRank^{A[i]}_a(x, A[1..i−1])
```

Theorem

Let T be the running time of a call to interpRank$_a^b$ where A arises by sorting random $X_1, \dots, X_n \sim \text{Unif}([a,b])$ and $x \in [a,b]$. Let $p = \frac{x-a}{b-a} \in [0,1]$ and $\tilde{p} = \min(p, 1-p)$. Then

$$\mathbb{E}[T] = \mathcal{O}(\max(1, \log\log \tilde{p}n)).$$

Corollary: $\mathbb{E}[T] = \mathcal{O}(\log\log n)$.

Proof Strategy

- **Case 1:** $\tilde{p}n = \mathcal{O}(1)$. // includes $\tilde{p}n = o(1)$ when $\log\log \tilde{p}n < 0$
- **Case 2:** $\tilde{p}n = \omega(1)$.

  For convenience assume $pn \in \mathbb{N}$ and $(a,b) = (0,n)$. Then $i = x = pn$.
  Let $\tilde{p}'$ and n' be the values of $\tilde{p}$ and n in the recursive call. We show:
  - Step 1: $\tilde{p}'n' \le \tilde{p}n$ // always
  - Step 2: If $A[\tilde{p}n] \in \tilde{p}n \pm \mathcal{O}(\sqrt{\tilde{p}n})$ then $\tilde{p}'n' \le \mathcal{O}(\sqrt{\tilde{p}n})$.
  - Step 3: The condition of Step 2 holds with probability $1-\epsilon$.

  Most iterations reduce $\tilde{p}n$ to $\sqrt{\tilde{p}n}$. The remaining iterations "don't hurt".
  $\Rightarrow \mathbb{E}[T] = \log\log \tilde{p}n$.

A : [array with positions 1, $i = pn$, n] $\in [0,n]^n$

assume without loss of generality: $\tilde{p} = p$.

Case 2, Step 1.

- If we recurse left then $\tilde{p}'n' \le n' = i - 1 < i = pn$.
- If we recurse right then $n' \le n$ and $p' \le p$, hence $\tilde{p}'n' \le p'n' \le pn$.

Interpolation Search: Algorithm and Analysis

Algorithm interpRank$_a^b(x, A[1..n] \in [a,b]^n)$**:**

- **if** $n = 0$ *or* $x = a$ **then return** 0
- let $p \leftarrow \frac{x-a}{b-a} \in (0,1]$ and $i = \lceil pn \rceil \in \{1,\dots,n\}$
- **if** $A[i] < x$ **then**
  - **return** $i +$ interpRank$_{A[i]}^b(x, A[i+1..n])$
- **else**
  - **return** interpRank$_a^{A[i]}(x, A[1..i-1])$

Theorem

Let T be the running time of a call to interpRank$_a^b$ where A arises by sorting random $X_1,\dots,X_n \sim \text{Unif}([a,b])$ and $x \in [a,b]$. Let $p = \frac{x-a}{b-a} \in [0,1]$ and $\tilde{p} = \min(p, 1-p)$. Then

$$\mathbb{E}[T] = \mathcal{O}(\max(1, \log\log \tilde{p}n)).$$

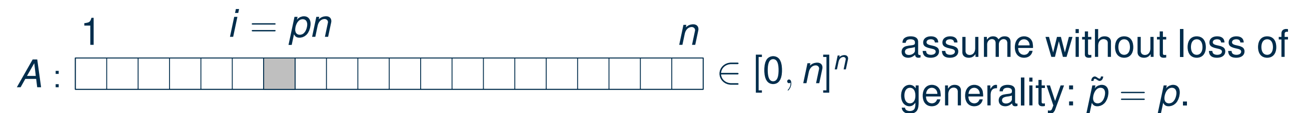
Corollary: $\mathbb{E}[T] = \mathcal{O}(\log\log n)$.

Proof Strategy

- **Case 1:** $\tilde{p}n = \mathcal{O}(1)$. // includes $\tilde{p}n = o(1)$ when $\log\log\tilde{p}n < 0$
- **Case 2:** $\tilde{p}n = \omega(1)$.

  For convenience assume $pn \in \mathbb{N}$ and $(a,b) = (0,n)$. Then $i = x = pn$. Let $\tilde{p}'$ and n' be the values of $\tilde{p}$ and n in the recursive call. We show:
  - Step 1: $\tilde{p}'n' \le \tilde{p}n$ // always
  - Step 2: If $A[\tilde{p}n] \in \tilde{p}n \pm \mathcal{O}(\sqrt{\tilde{p}n})$ then $\tilde{p}'n' \le \mathcal{O}(\sqrt{\tilde{p}n})$.
  - Step 3: The condition of Step 2 holds with probability $1-\epsilon$.

  Most iterations reduce $\tilde{p}n$ to $\sqrt{\tilde{p}n}$. The remaining iterations "don't hurt".
  $\Rightarrow \mathbb{E}[T] = \log\log\tilde{p}n$.



assume without loss of generality: $\tilde{p} = p$.

Case 2, Step 2.

- If we recurse right, then $\tilde{p}' \le p' = \frac{x-a'}{b'-a'} = \frac{pn-A[pn]}{n-A[pn]} \le \frac{\mathcal{O}(\sqrt{pn})}{\mathcal{O}(n)} = \mathcal{O}(\sqrt{p/n})$.
  Hence $\tilde{p}'n' \le \tilde{p}'n = \mathcal{O}(\sqrt{pn})$.
- If we recurse left, then $\tilde{p}' \le 1 - p' = \frac{b'-x}{b'-a'} = \frac{A[pn]-pn}{A[pn]} \le \frac{\mathcal{O}(\sqrt{pn})}{\mathcal{O}(pn)} = \mathcal{O}(\frac{1}{\sqrt{pn}})$.
  Hence $\tilde{p}'n' \le \tilde{p}'pn = \mathcal{O}(\sqrt{pn})$.

Interpolation Search: Algorithm and Analysis

Algorithm $\text{interpRank}_a^b(x, A[1..n] \in [a,b]^n)$**:**

- **if** $n = 0$ *or* $x = a$ **then return** 0
- let $p \leftarrow \frac{x-a}{b-a} \in (0,1]$ and $i = \lceil pn \rceil \in \{1, \dots, n\}$
- **if** $A[i] < x$ **then**
  - **return** $i + \text{interpRank}_{A[i]}^b(x, A[i+1..n])$
- **else**
  - **return** $\text{interpRank}_a^{A[i]}(x, A[1..i-1])$

Theorem

Let T be the running time of a call to interpRank_a^b where A arises by sorting random $X_1, \dots, X_n \sim \text{Unif}([a,b])$ and $x \in [a,b]$. Let $p = \frac{x-a}{b-a} \in [0,1]$ and $\tilde{p} = \min(p, 1-p)$. Then

$$\mathbb{E}[T] = \mathcal{O}(\max(1, \log\log \tilde{p}n)).$$

Corollary: $\mathbb{E}[T] = \mathcal{O}(\log\log n)$.

Proof Strategy

- **Case 1:** $\tilde{p}n = \mathcal{O}(1)$. // includes $\tilde{p}n = o(1)$ when $\log\log \tilde{p}n < 0$
- **Case 2:** $\tilde{p}n = \omega(1)$.

  For convenience assume $pn \in \mathbb{N}$ and $(a,b) = (0,n)$. Then $i = x = pn$. Let $\tilde{p}'$ and n' be the values of $\tilde{p}$ and n in the recursive call. We show:

  - Step 1: $\tilde{p}'n' \le \tilde{p}n$ // always
  - Step 2: If $A[\tilde{p}n] \in \tilde{p}n \pm \mathcal{O}(\sqrt{\tilde{p}n})$ then $\tilde{p}'n' \le \mathcal{O}(\sqrt{\tilde{p}n})$.
  - Step 3: The condition of Step 2 holds with probability $1 - \epsilon$.

  Most iterations reduce $\tilde{p}n$ to $\sqrt{\tilde{p}n}$. The remaining iterations "don't hurt".

  $\Rightarrow \mathbb{E}[T] = \log\log \tilde{p}n$.

A : [array of cells indexed 1 … $i = pn$ … n] $\in [0,n]^n$

assume without loss of generality: $\tilde{p} = p$.

Case 2, Step 3: $A[pn] \ge pn - \mathcal{O}(\sqrt{pn})$. // "$\ldots \le \ldots + \ldots$" is analogous

Let $L = pn - C\sqrt{pn}$ and $n_{<L}$ the number of elements in range $[0, L]$.

- We have $\mathbb{E}[n_{<L}] = L$ and $\sigma^2 = \text{Var}(n_{<L}) \le L \le pn$.
- Hence $\Pr[n_{<L} \ge pn] = \Pr[n_{<L} \ge \mathbb{E}[n_{<L}] + C\sigma] < \epsilon$ // if $C(\epsilon)$ large enough
- In the likely case of $n_{<L} < pn$ we get $A[pn] \ge L$. ✓

Interpolation Search: Algorithm and Analysis

Algorithm $\text{interpRank}_a^b(x, A[1..n] \in [a,b]^n)$**:**

- **if** $n = 0$ *or* $x = a$ **then return** 0
- let $p \leftarrow \frac{x-a}{b-a} \in (0,1]$ and $i = \lceil pn \rceil \in \{1,\dots,n\}$
- **if** $A[i] < x$ **then**
  - **return** $i + \text{interpRank}_{A[i]}^b(x, A[i+1..n])$
- **else**
  - **return** $\text{interpRank}_a^{A[i]}(x, A[1..i-1])$

Theorem

Let T be the running time of a call to interpRank_a^b where A arises by sorting random $X_1,\dots,X_n \sim \text{Unif}([a,b])$ and $x \in [a,b]$. Let $p = \frac{x-a}{b-a} \in [0,1]$ and $\tilde{p} = \min(p, 1-p)$. Then

$$\mathbb{E}[T] = \mathcal{O}(\max(1, \log\log \tilde{p}n)).$$

Corollary: $\mathbb{E}[T] = \mathcal{O}(\log\log n)$.

Exercise

Find a worst case instance for interpolation search!

Interpolation Search: Algorithm and Analysis

Algorithm $\text{interpRank}_a^b(x, A[1..n] \in [a,b]^n)$**:**

- **if** $n = 0$ *or* $x = a$ **then return** 0
- let $p \leftarrow \frac{x-a}{b-a} \in (0,1]$ and $i = \lceil pn \rceil \in \{1,\dots,n\}$
- **if** $A[i] < x$ **then**
  - **return** $i + \text{interpRank}_{A[i]}^b(x, A[i+1..n])$
- **else**
  - **return** $\text{interpRank}_a^{A[i]}(x, A[1..i-1])$

Theorem

Let T be the running time of a call to interpRank_a^b where A arises by sorting random $X_1,\dots,X_n \sim \text{Unif}([a,b])$ and $x \in [a,b]$. Let $p = \frac{x-a}{b-a} \in [0,1]$ and $\tilde{p} = \min(p, 1-p)$. Then

$$\mathbb{E}[T] = \mathcal{O}(\max(1, \log\log \tilde{p}n)).$$

Corollary: $\mathbb{E}[T] = \mathcal{O}(\log\log n)$.

Exercise

Find a worst case instance for interpolation search!

$A:$

0	0	0	0	0	0	0	0	$\cdots$	0	n

$x = 1 \rightsquigarrow T = \mathcal{O}(n)$.

Interpolation Search: Algorithm and Analysis

Algorithm interpRank$_a^b(x, A[1..n] \in [a,b]^n)$**:**

- **if** $n = 0$ *or* $x = a$ **then return** 0
- let $p \leftarrow \frac{x-a}{b-a} \in (0,1]$ and $i = \lceil pn \rceil \in \{1,\dots,n\}$
- **if** $A[i] < x$ **then**
  - **return** $i +$ interpRank$_{A[i]}^b(x, A[i+1..n])$
- **else**
  - **return** interpRank$_a^{A[i]}(x, A[1..i-1])$

Theorem

Let T be the running time of a call to interpRank$_a^b$ where A arises by sorting random $X_1,\dots,X_n \sim \text{Unif}([a,b])$ and $x \in [a,b]$. Let $p = \frac{x-a}{b-a} \in [0,1]$ and $\tilde{p} = \min(p, 1-p)$. Then

$$\mathbb{E}[T] = \mathcal{O}(\max(1, \log\log \tilde{p}n)).$$

Corollary: $\mathbb{E}[T] = \mathcal{O}(\log\log n)$.

Exercise

Find a worst case instance for interpolation search!

A : | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ⋯ | 0 | n | $x = 1 \rightsquigarrow T = \mathcal{O}(n)$.

Exercise

Propose a modification to simultaneously achieve

- worst case running time $\mathcal{O}(\log n)$ and
- expected running time $\mathcal{O}(\log\log n)$ for random inputs.

Interpolation Search: Algorithm and Analysis

Algorithm $\text{interpRank}_a^b(x, A[1..n] \in [a,b]^n)$**:**

- **if** $n = 0$ *or* $x = a$ **then return** 0
- let $p \leftarrow \frac{x-a}{b-a} \in (0,1]$ and $i = \lceil pn \rceil \in \{1, \ldots, n\}$
- **if** $A[i] < x$ **then**
  - **return** $i + \text{interpRank}_{A[i]}^b(x, A[i+1..n])$
- **else**
  - **return** $\text{interpRank}_a^{A[i]}(x, A[1..i-1])$

Theorem

Let T be the running time of a call to interpRank_a^b where A arises by sorting random $X_1, \ldots, X_n \sim \text{Unif}([a,b])$ and $x \in [a,b]$. Let $p = \frac{x-a}{b-a} \in [0,1]$ and $\tilde{p} = \min(p, 1-p)$. Then

$$\mathbb{E}[T] = \mathcal{O}(\max(1, \log\log \tilde{p}n)).$$

Corollary: $\mathbb{E}[T] = \mathcal{O}(\log\log n)$.

Exercise

Find a worst case instance for interpolation search!

A : | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ⋯ | 0 | n | $x = 1 \rightsquigarrow T = \mathcal{O}(n)$.

Exercise

Propose a modification to simultaneously achieve

- worst case running time $\mathcal{O}(\log n)$ and
- expected running time $\mathcal{O}(\log\log n)$ for random inputs.

Solution: Alternate interpolation steps and regular binary search steps.

Dynamic Predecessors for Fully Random Data

Theorem

For any $n \in \mathbb{N}$ there exists a dynamic predecessor data structure that after storing random $X_1, \ldots, X_n \sim \text{Unif}\big((0,1)\big)$

- for any $x \in [0,1]$, finding pred(x) takes expected time $\mathcal{O}(1)$,
- for any $x \in [0,1]$, inserting x takes time $\mathcal{O}(1)$,
- for any $i \in [n]$, deleting X_i takes expected time $\mathcal{O}(1)$,
- and the data structure takes space $\mathcal{O}(n)$.

Proof.

Dynamic Predecessors for Fully Random Data

Theorem

For any $n \in \mathbb{N}$ there exists a dynamic predecessor data structure that after storing random $X_1, \ldots, X_n \sim \text{Unif}((0,1))$

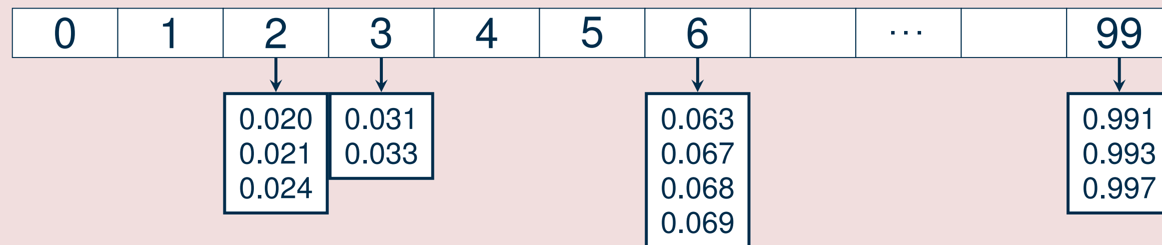
- for any $x \in [0,1]$, finding pred(x) takes expected time $\mathcal{O}(1)$,
- for any $x \in [0,1]$, inserting x takes time $\mathcal{O}(1)$,
- for any $i \in [n]$, deleting X_i takes expected time $\mathcal{O}(1)$,
- and the data structure takes space $\mathcal{O}(n)$.

Proof.

Use hash table with chaining with n buckets and deterministic "hash function" h

$$h : \begin{cases} [0,1] & \to \{0, \ldots, n-1\} \\ x & \mapsto \lfloor xn \rfloor. \end{cases}$$

$\Rightarrow$ Every bucket has expected size $\mathcal{O}(1)$.



Warning

If the data is imperfectly random this may fail *catastrophically*.

Dynamic Predecessors for Fully Random Data

Theorem

For any $n \in \mathbb{N}$ there exists a dynamic predecessor data structure that after storing random $X_1, \ldots, X_n \sim \text{Unif}((0,1))$

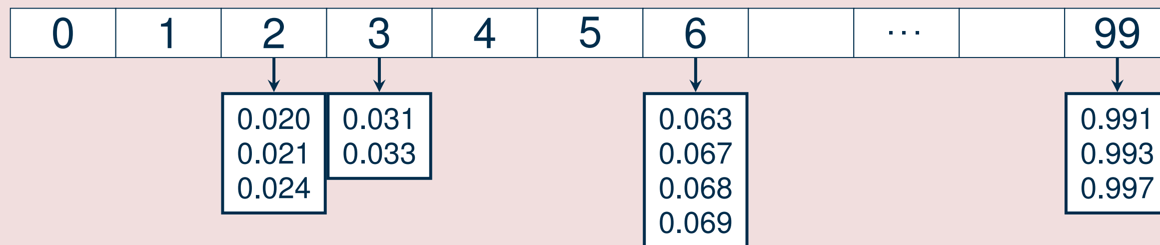
- for any $x \in [0,1]$, finding pred(x) takes expected time $\mathcal{O}(1)$,
- for any $x \in [0,1]$, inserting x takes time $\mathcal{O}(1)$,
- for any $i \in [n]$, deleting X_i takes expected time $\mathcal{O}(1)$,
- and the data structure takes space $\mathcal{O}(n)$.

Proof.

Use hash table with chaining with n buckets and deterministic "hash function" h

$$h : \begin{cases} [0,1] & \to \{0, \ldots, n-1\} \\ x & \mapsto \lfloor xn \rfloor. \end{cases}$$

$\Rightarrow$ Every bucket has expected size $\mathcal{O}(1)$.



Warning

If the data is imperfectly random this may fail *catastrophically*.

Simple Failure Case

Each key arises from $b_1, \ldots, b_k \sim \text{Unif}(\{0,1\})$ as $X = b_1 0 b_2 0 b_3 0 \ldots b_k 0$.

1. Lower Bounds

2. Predecessor Search for Structured Inputs

Uniform Random Input Data

Learned Index Structures

Jargon

What is an "Index"?

- for us: synonymous with *predecessor data structure*
- often: acceleration data structure built on top of existing data structure (e.g. a sorted array)

What is "Learned"?

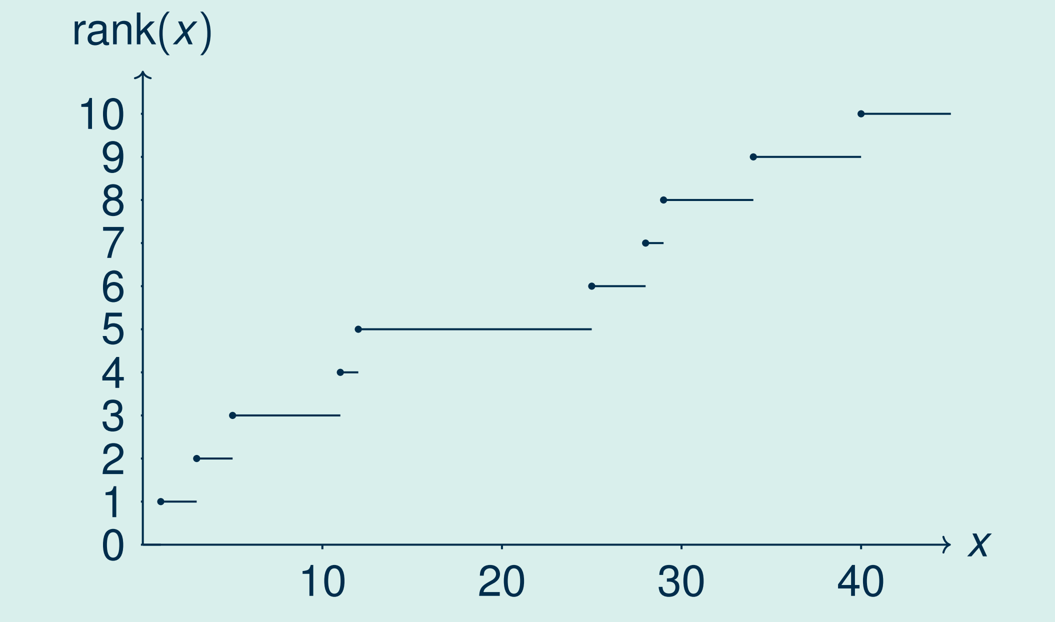
- refers to *machine learning*
- think "linear regression", not "LLM"

“The Case for Learned Index Structures” (Kraska et al, 2018)

Assume a sorted array A stores $a_1 \leq \ldots \leq a_n$. Define

$$\text{rank} : \begin{cases} \mathbb{R} & \to \{0, \ldots, n\} \\ x & \mapsto |\{i \in [n] \mid a_i \leq x\}| \end{cases}$$

Example: $A = [1, 3, 5, 11, 12, 25, 28, 29, 34, 40]$

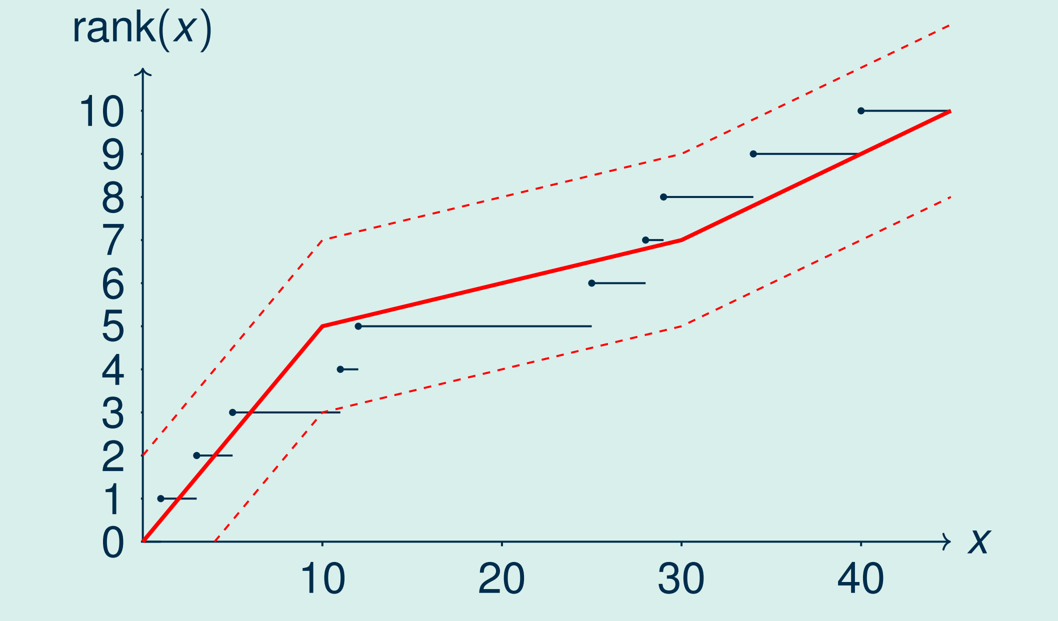


“The Case for Learned Index Structures” (Kraska et al, 2018)

Assume a sorted array A stores $a_1 \leq \ldots \leq a_n$. Define

$$\text{rank} : \begin{cases} \mathbb{R} & \to \{0,\ldots,n\} \\ x & \mapsto |\{i \in [n] \mid a_i \leq x\}| \end{cases}$$

Example: $A = [1, 3, 5, 11, 12, 25, 28, 29, 34, 40]$



Observation: A good predictor makes for a good index

Assume $\widehat{\text{rank}}$ is a function with evaluation time T. We can accelerate the computation of rank as follows.

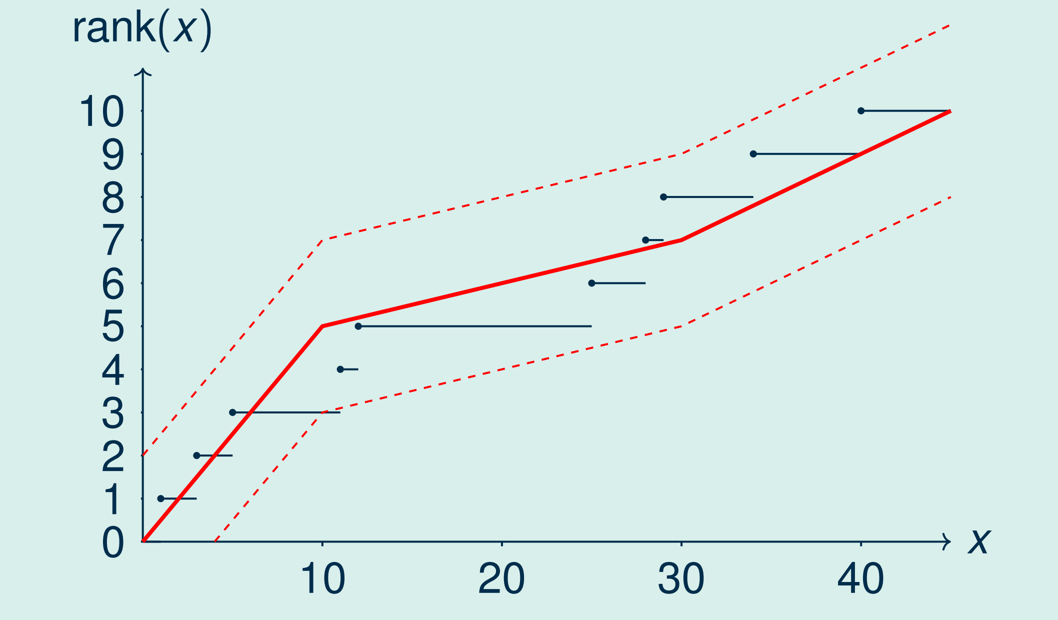
1. Given $\epsilon \in \mathbb{N}$ such that $\|\widehat{\text{rank}} - \text{rank}\|_\infty \leq \epsilon$, we can compute $\text{rank}(x)$ in time $T + \mathcal{O}(\log \epsilon)$.
2. We can compute $\text{rank}(x)$ in time $T + \mathcal{O}(\log |\widehat{\text{rank}}(x) - \text{rank}(x)|)$.

“The Case for Learned Index Structures” (Kraska et al, 2018)

Assume a sorted array A stores $a_1 \leq \ldots \leq a_n$. Define

$$\text{rank} : \begin{cases} \mathbb{R} & \to \{0, \ldots, n\} \\ x & \mapsto |\{i \in [n] \mid a_i \leq x\}| \end{cases}$$

Example: $A = [1, 3, 5, 11, 12, 25, 28, 29, 34, 40]$



Observation: A good predictor makes for a good index

Assume $\widehat{\text{rank}}$ is a function with evaluation time T. We can accelerate the computation of rank as follows.

1. Given $\epsilon \in \mathbb{N}$ such that $\|\widehat{\text{rank}} - \text{rank}\|_\infty \leq \epsilon$, we can compute $\text{rank}(x)$ in time $T + \mathcal{O}(\log \epsilon)$.
2. We can compute $\text{rank}(x)$ in time $T + \mathcal{O}(\log |\widehat{\text{rank}}(x) - \text{rank}(x)|)$.

Proof

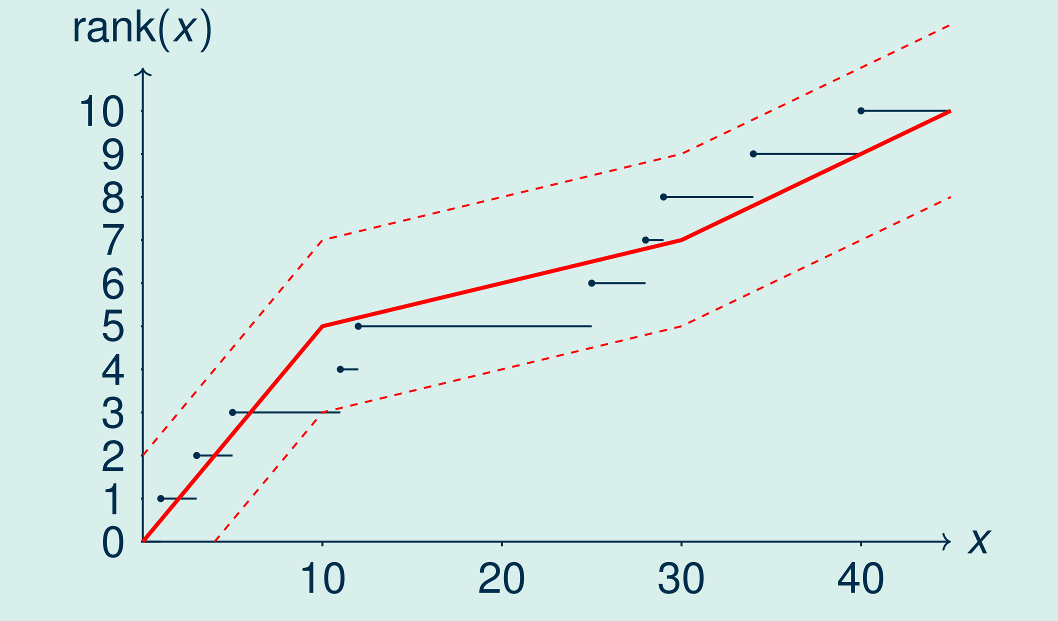
1. Binary search $A[\widehat{\text{rank}}(x) - \epsilon, \widehat{\text{rank}}(x) + \epsilon]$.
2. Start exponential search (see below) from $\widehat{\text{rank}}(x)$.

"The Case for Learned Index Structures" (Kraska et al, 2018)

Assume a sorted array A stores $a_1 \leq \ldots \leq a_n$. Define

$$\text{rank} : \begin{cases} \mathbb{R} & \to \{0, \ldots, n\} \\ x & \mapsto |\{i \in [n] \mid a_i \leq x\}| \end{cases}$$

Example: $A = [1, 3, 5, 11, 12, 25, 28, 29, 34, 40]$



Observation: A good predictor makes for a good index

Assume $\widehat{\text{rank}}$ is a function with evaluation time T. We can accelerate the computation of rank as follows.

1. Given $\epsilon \in \mathbb{N}$ such that $\|\widehat{\text{rank}} - \text{rank}\|_\infty \leq \epsilon$, we can compute $\text{rank}(x)$ in time $T + \mathcal{O}(\log \epsilon)$.
2. We can compute $\text{rank}(x)$ in time $T + \mathcal{O}(\log |\widehat{\text{rank}}(x) - \text{rank}(x)|)$.

Proof

1. Binary search $A[\widehat{\text{rank}}(x) - \epsilon, \widehat{\text{rank}}(x) + \epsilon]$.
2. Start exponential search (see below) from $\widehat{\text{rank}}(x)$.

Observation: Exponential Search

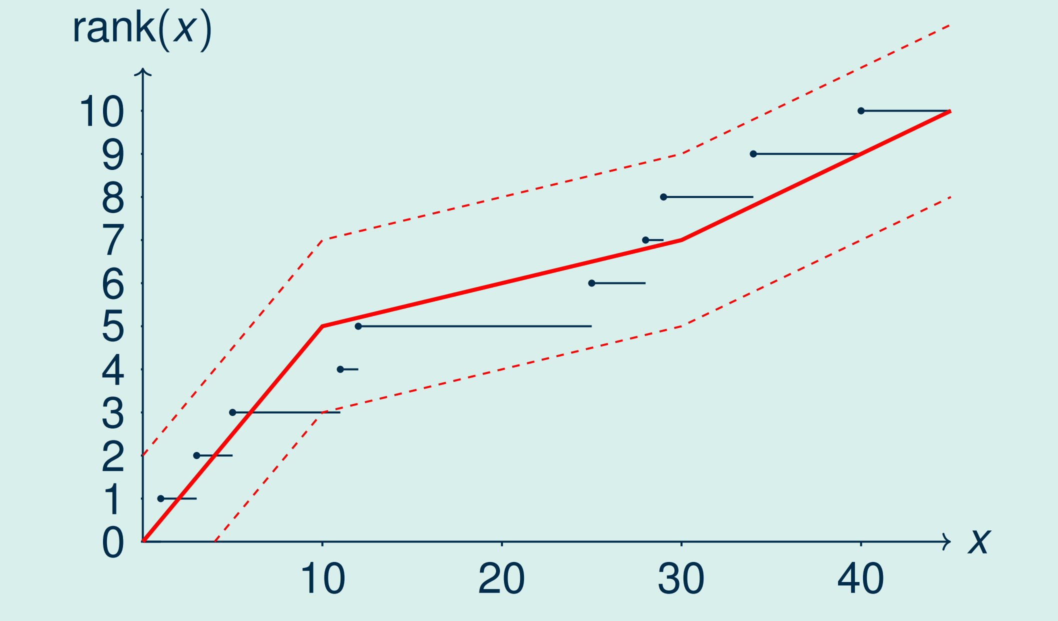
Even for unbounded sorted A, there exists an algorithm that, given $x \in \mathbb{R}$ with $\text{rank}(x) < \infty$, computes $\text{rank}(x)$ in time $\mathcal{O}(\log |\text{rank}(x)|)$.

“The Case for Learned Index Structures” (Kraska et al, 2018)

Assume a sorted array A stores $a_1 \le \dots \le a_n$. Define

$$\text{rank}: \begin{cases} \mathbb{R} & \to \{0,\dots,n\} \\ x & \mapsto |\{i \in [n] \mid a_i \le x\}| \end{cases}$$

Example: $A = [1,3,5,11,12,25,28,29,34,40]$



Observation: A good predictor makes for a good index

Assume $\widehat{\text{rank}}$ is a function with evaluation time T. We can accelerate the computation of rank as follows.

1. Given $\epsilon \in \mathbb{N}$ such that $\|\widehat{\text{rank}} - \text{rank}\|_\infty \le \epsilon$, we can compute $\text{rank}(x)$ in time $T + \mathcal{O}(\log \epsilon)$.
2. We can compute $\text{rank}(x)$ in time $T + \mathcal{O}(\log |\widehat{\text{rank}}(x) - \text{rank}(x)|)$.

Proof

1. Binary search $A[\widehat{\text{rank}}(x) - \epsilon, \widehat{\text{rank}}(x) + \epsilon]$.
2. Start exponential search (see below) from $\widehat{\text{rank}}(x)$.

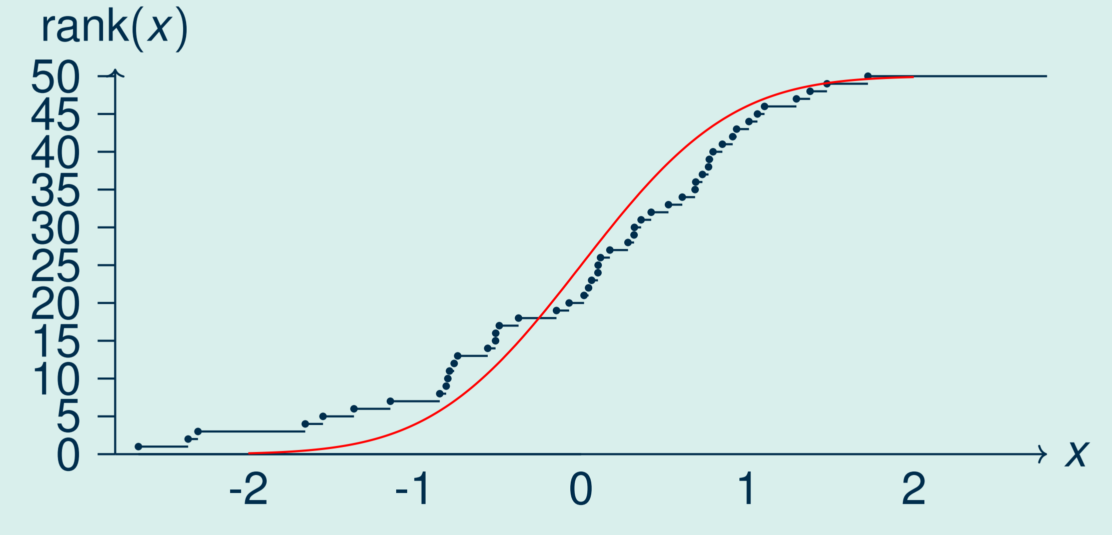
Observation: Exponential Search

Even for unbounded sorted A, there exists an algorithm that, given $x \in \mathbb{R}$ with $\text{rank}(x) < \infty$, computes $\text{rank}(x)$ in time $\mathcal{O}(\log |\text{rank}(x)|)$.

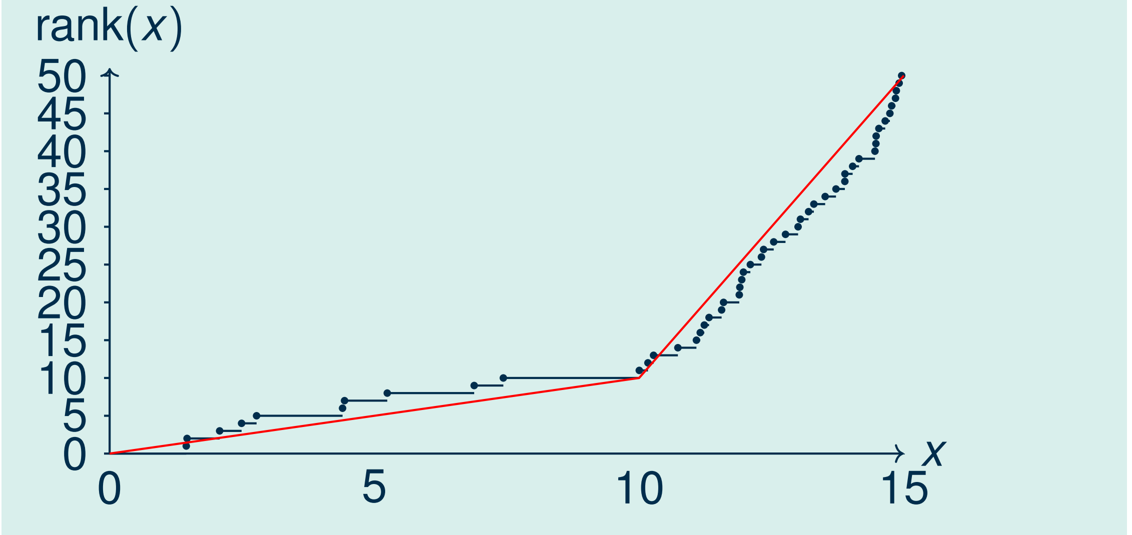
$r \leftarrow 1$
while $x \ge A[r]$ **do**
  $r \leftarrow 2r$
binary search $A[1..r]$

Simple Distributions that Occur in Practice

Gaussian Distribution



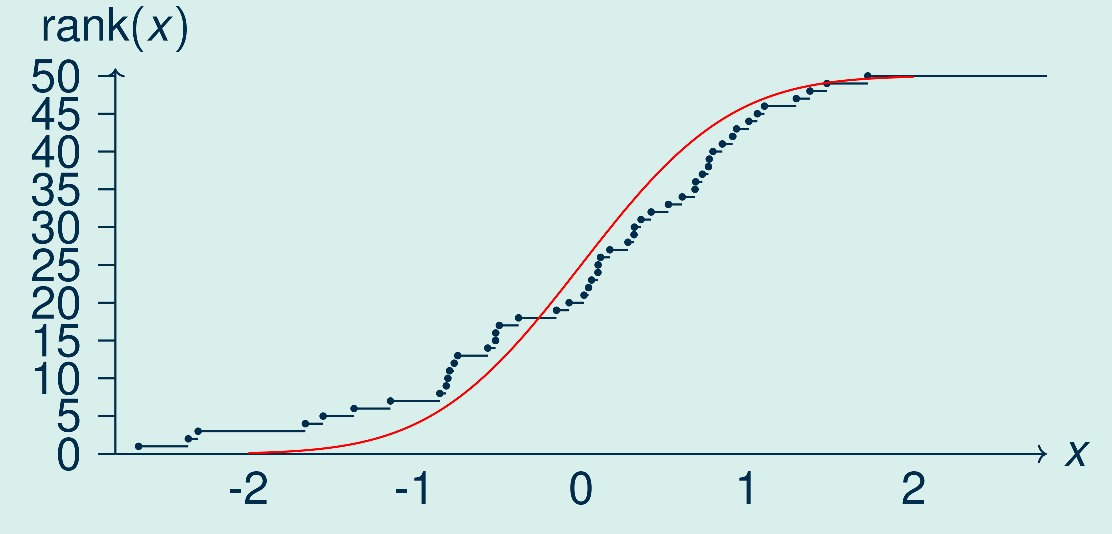
Piece-Wise Linear Distribution Function[a]



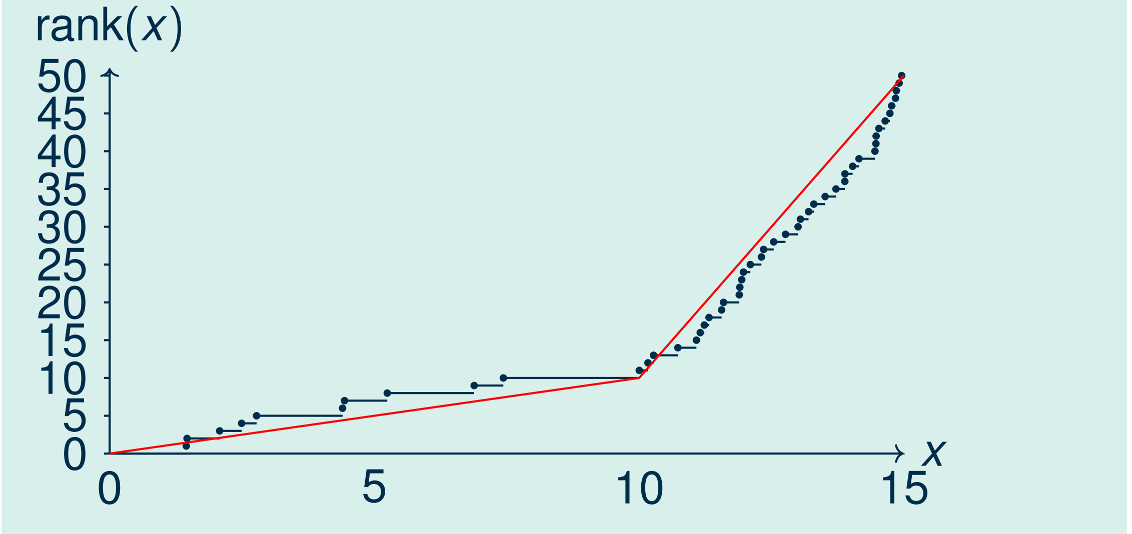
[a]Meaning piece-wise constant density function.

Simple Distributions that Occur in Practice

Gaussian Distribution



Piece-Wise Linear Distribution Function[a]

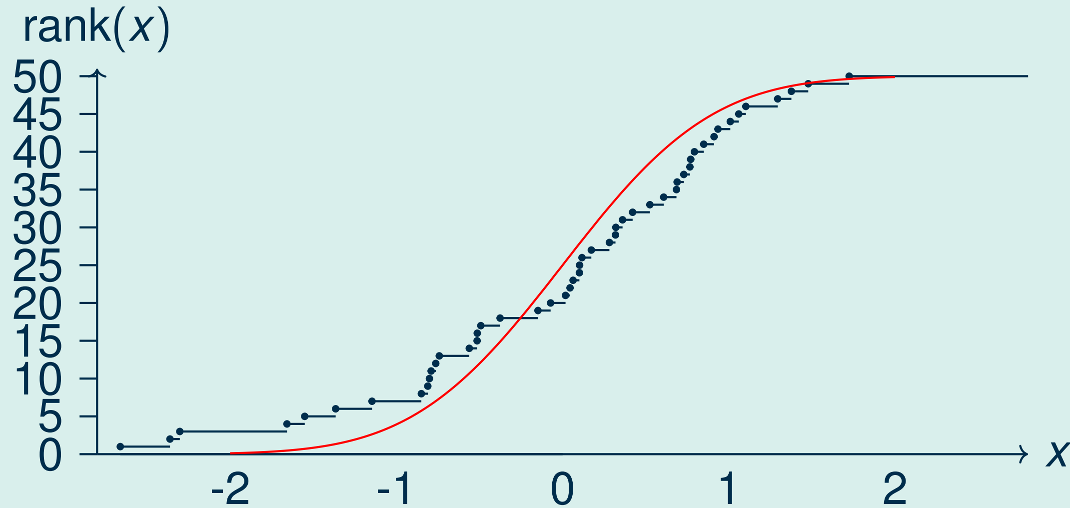


[a] Meaning piece-wise constant density function.

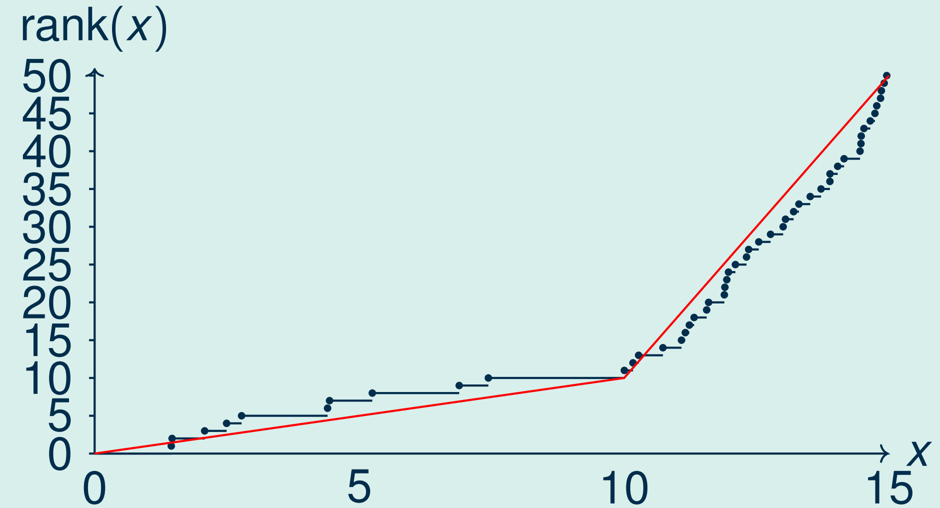
Discuss: How would our approaches for uniformly random data perform?

Simple Distributions that Occur in Practice

Gaussian Distribution



Piece-Wise Linear Distribution Function[a]



[a] Meaning piece-wise constant density function.

Discuss: How would our approaches for uniformly random data perform?

Hope: Machine learning can detect such patterns

But recall: we want to beat B-Trees. Sophisticated prediction methods are too slow.

PGM Index (PGM = Paolo & Giorgio Model)

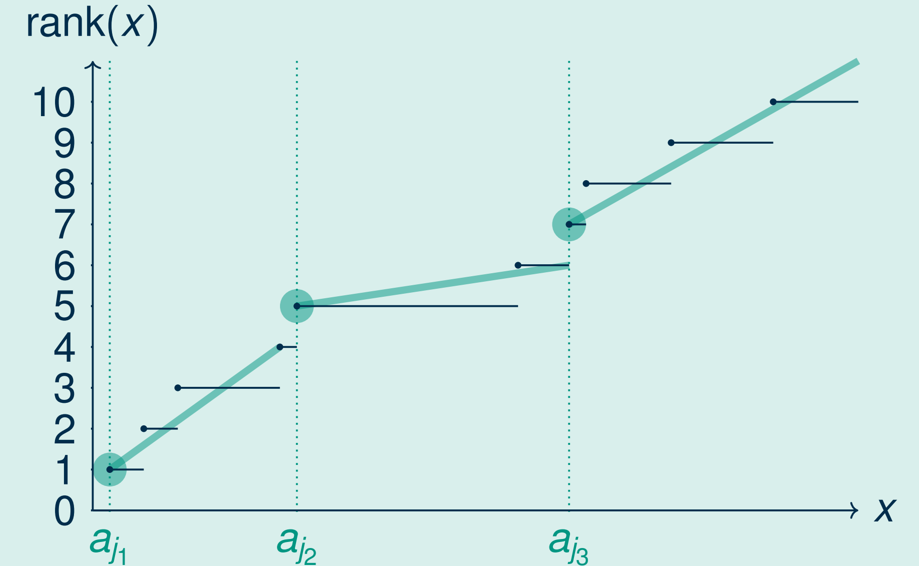
PGM Index (PGM = ~~Paolo & Giorgio Model~~ Piece-wise Geometric Model)

PGM Index (PGM = ~~Paolo & Giorgio Model~~ Piece-wise Geometric Model)

Construct($A = [a_1, \dots, a_n]$)

- Compute ϵ-approximation of rank using s segments
  - goal: minimise s while respecting error bound ϵ
  - ith segment covers range $a_{j_i}, \dots, a_{j_{i+1}-1}$ of keys
  - ith segment interpolates (a_{j_i}, j_i) // we're storing a_{j_i} anyway
- store (a_{j_i}, j_i) and slope for each segment
- recursively compute PGM index for $(a_{j_1}, \dots, a_{j_s})$.

Example



PGM Index (PGM = ~~Paolo & Giorgio Model~~ Piece-wise Geometric Model)

Construct($A = [a_1, \ldots, a_n]$)

- Compute ϵ-approximation of rank using s segments
  - goal: minimise s while respecting error bound ϵ
  - ith segment covers range $a_{j_i}, \ldots, a_{j_{i+1}-1}$ of keys
  - ith segment interpolates (a_{j_i}, j_i) // we're storing a_{j_i} anyway
- store (a_{j_i}, j_i) and slope for each segment
- recursively compute PGM index for $(a_{j_1}, \ldots, a_{j_s})$.

Computing rank(x)

- query recursively PGM to find segment index i
- evaluate ith segment to compute estimate $\widehat{\text{rank}}(x)$
- binary search $A[\widehat{\text{rank}}(x) - \epsilon..\widehat{\text{rank}}(x) + \epsilon]$

Example

PGM Index (PGM = ~~Paolo & Giorgio Model~~ Piece-wise Geometric Model)

Construct($A = [a_1, \dots, a_n]$)

- Compute ϵ-approximation of rank using s segments
  - goal: minimise s while respecting error bound ϵ
  - ith segment covers range $a_{j_i}, \dots, a_{j_{i+1}-1}$ of keys
  - ith segment interpolates (a_{j_i}, j_i) // we're storing a_{j_i} anyway
- store (a_{j_i}, j_i) and slope for each segment
- recursively compute PGM index for $(a_{j_1}, \dots, a_{j_s})$.

Computing rank(x)

- query recursively PGM to find segment index i
- evaluate ith segment to compute estimate $\widehat{\text{rank}}(x)$
- binary search $A[\widehat{\text{rank}}(x) - \epsilon..\widehat{\text{rank}}(x) + \epsilon]$

Worst Case:

Example

PGM Index (PGM = ~~Paolo & Giorgio Model~~ Piece-wise Geometric Model)

Construct($A = [a_1, \ldots, a_n]$)

- Compute ϵ-approximation of rank using s segments
  - goal: minimise s while respecting error bound ϵ
  - ith segment covers range $a_{j_i}, \ldots, a_{j_{i+1}-1}$ of keys
  - ith segment interpolates (a_{j_i}, j_i) // we're storing a_{j_i} anyway
- store (a_{j_i}, j_i) and slope for each segment
- recursively compute PGM index for $(a_{j_1}, \ldots, a_{j_s})$.

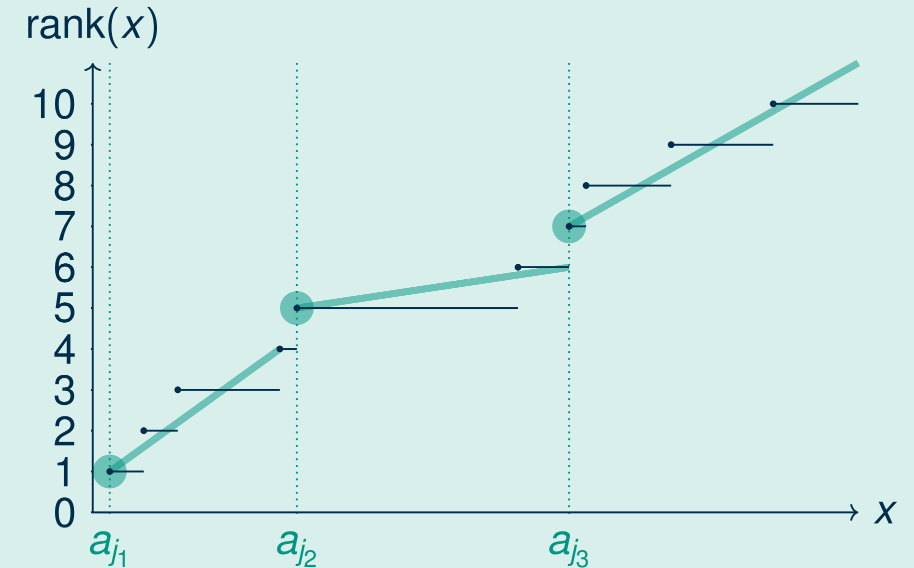
Computing rank(x)

- query recursively PGM to find segment index i
- evaluate ith segment to compute estimate $\widehat{\text{rank}}(x)$
- binary search $A[\widehat{\text{rank}}(x) - \epsilon..\widehat{\text{rank}}(x) + \epsilon]$

Worst Case: Behaves like a B-Tree (with $B = \epsilon$)

Segment covers $\approx \epsilon$ points
$\hookrightarrow \approx \log_\epsilon n$ levels of recursion
$\hookrightarrow$ binary search on $\mathcal{O}(\epsilon)$ keys per level

Example



PGM Index (PGM = ~~Paolo & Giorgio Model~~ Piece-wise Geometric Model)

Construct($A = [a_1, \dots, a_n]$)

- Compute ϵ-approximation of rank using s segments
  - goal: minimise s while respecting error bound ϵ
  - ith segment covers range $a_{j_i}, \dots, a_{j_{i+1}-1}$ of keys
  - ith segment interpolates (a_{j_i}, j_i) // we're storing a_{j_i} anyway
- store (a_{j_i}, j_i) and slope for each segment
- recursively compute PGM index for $(a_{j_1}, \dots, a_{j_s})$.

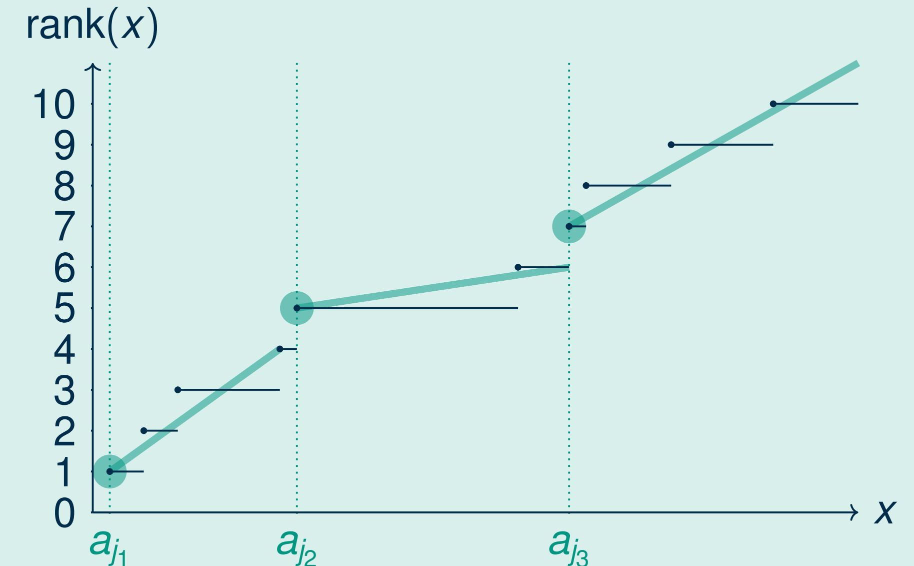
Computing rank(x)

- query recursively PGM to find segment index i
- evaluate ith segment to compute estimate $\widehat{\text{rank}}(x)$
- binary search $A[\widehat{\text{rank}}(x) - \epsilon..\widehat{\text{rank}}(x) + \epsilon]$

Worst Case: Behaves like a B-Tree (with $B = \epsilon$)

Segment covers $\approx \epsilon$ points
$\hookrightarrow \approx \log_\epsilon n$ levels of recursion
$\hookrightarrow$ binary search on $\mathcal{O}(\epsilon)$ keys per level

Example



Average Case

PGM Index (PGM = ~~Paolo & Giorgio Model~~ Piece-wise Geometric Model)

Construct($A = [a_1, \ldots, a_n]$)

- Compute ϵ-approximation of rank using s segments
  - goal: minimise s while respecting error bound ϵ
  - ith segment covers range $a_{j_i}, \ldots, a_{j_{i+1}-1}$ of keys
  - ith segment interpolates (a_{j_i}, j_i) // we're storing a_{j_i} anyway
- store (a_{j_i}, j_i) and slope for each segment
- recursively compute PGM index for $(a_{j_1}, \ldots, a_{j_s})$.

Computing rank(x)

- query recursively PGM to find segment index i
- evaluate ith segment to compute estimate $\widehat{\text{rank}}(x)$
- binary search $A[\widehat{\text{rank}}(x) - \epsilon..\widehat{\text{rank}}(x) + \epsilon]$

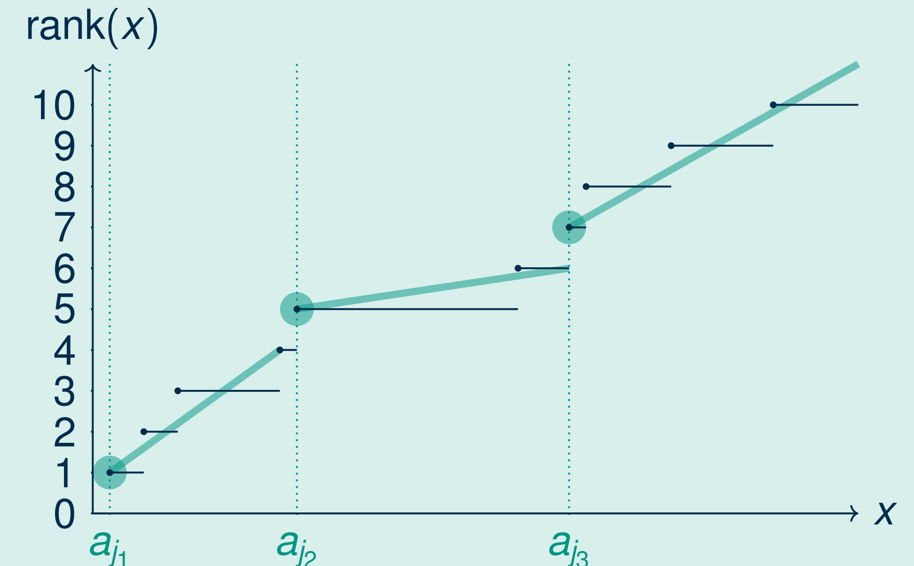
Worst Case: Behaves like a B-Tree (with $B = \epsilon$)

Segment covers $\approx \epsilon$ points

$\hookrightarrow \approx \log_\epsilon n$ levels of recursion

$\hookrightarrow$ binary search on $\mathcal{O}(\epsilon)$ keys per level

Example



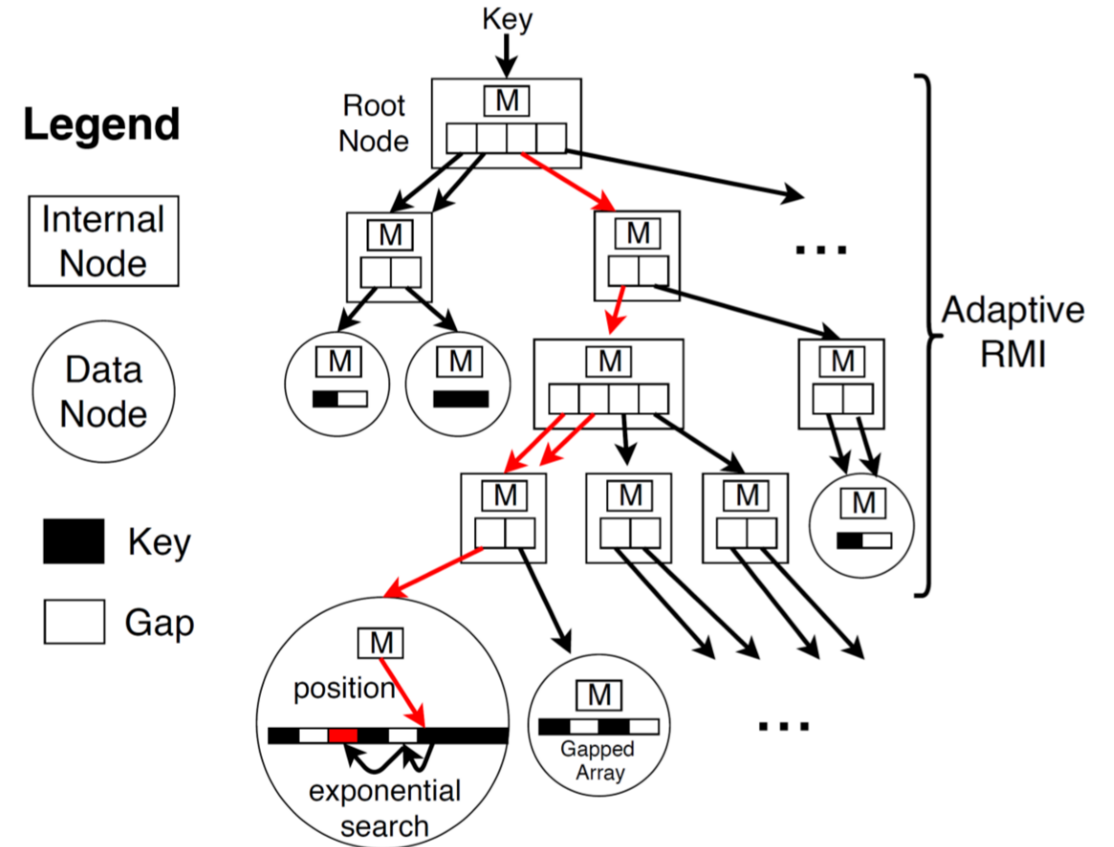
Average Case

- For "locally random" data, a segment covers $\mathcal{O}(\epsilon^2)$ points in expectation.
- $\approx \log_{\epsilon^2} n = \frac{1}{2} \log_\epsilon n$ levels of recursion

ALEX: Adaptive learned index for dynamic workloads

ALEX: B^+-Trees with several twists

- larger nodes
- linear model in each node
- gaps in node arrays
- advanced splitting heuristics
- *no proven performance guarantees*



Picture taken from a slide of the authors.

Conference Talk at SIGMOD 2020: https://www.youtube.com/watch?v=wVxbOcwYZ8I

Summary

Lower Bounds

No algorithm can beat $\min(\frac{\text{vEB}}{\log\log w}, FT)$.

// we only proved $\min(\frac{\text{vEB}}{\log\log n}, FT)$

Proof techniques:

- communication complexity
- cell probe lower bound
- nested round elimination

Interpolation Search

Achieves $\mathcal{O}(\log\log n)$ expected search times on sorted array with random data.

"Hash" Table

Achieves $\mathcal{O}(1)$ expected search times and $\mathcal{O}(n)$ space for dynamic predecessor problem, but is not robust at all.

Learned Index Structures

Try to improve over the worst case for many practical data distributions.

- PGM index uses recursively constructed linear approximation.
- ALEX maintains linear models in B-Tree-like dynamic data structure.

Possible Exam Questions

- regarding lower bounds:
  - Compare fusion trees and van Emde Boas trees. Which is better (under what conditions)?
  - How close is $\min(\text{FT}, \text{vEB})$ to optimal? What did we prove?
  - Define the communication complexity of a function.
    - In what sense is a word RAM algorithm a communication protocol?
    - What do we mean by a cell-probe lower bound?
  - Summarise the technique of round elimination. How did we eliminate a message by Alice? How did we eliminate a message by Bob?
- regarding structured inputs:
  - Can the lower bound be beaten with additional assumptions on the input data?
  - Explain interpolation search.
    - Explain intuitively how it achieves running time $\mathcal{O}(\log \log n)$.
    - What is the formal result we have proven?
    - What is the worst-case for interpolation search and how can it be mitigated?
  - Describe a dynamic predecessor data structure for random data.
    - What is its performance?
    - What is a major caveat?
  - Learned Index Structures:
    - In what sense does a predictor for a function give rise to a predecessor data structure? (Also explain interpolation search.)
    - Describe the PGM Index.
    - ↪ What is its worst-case behaviour and why?
    - ↪ What is its behaviour on random data and why?