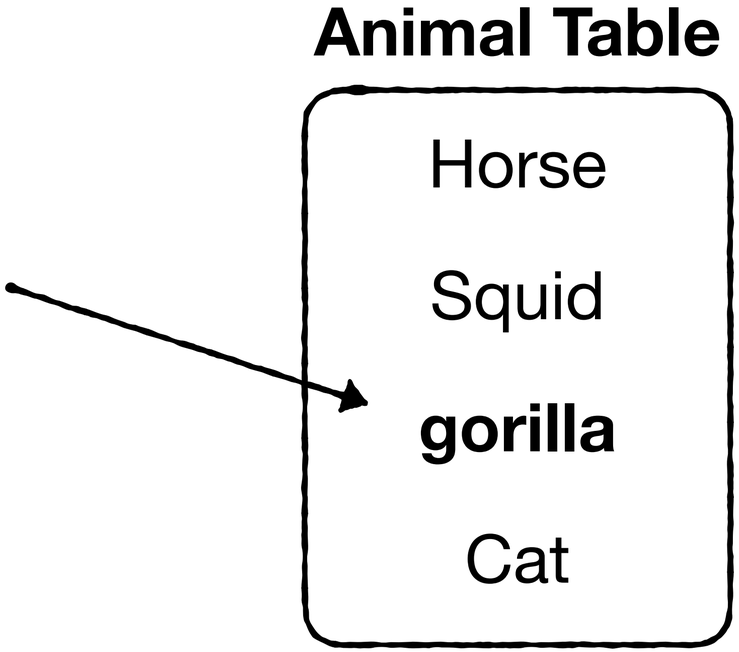
In-Memory Indexing

Niv Dayan - CSC2525 Research Topics in Database Management

Indexes

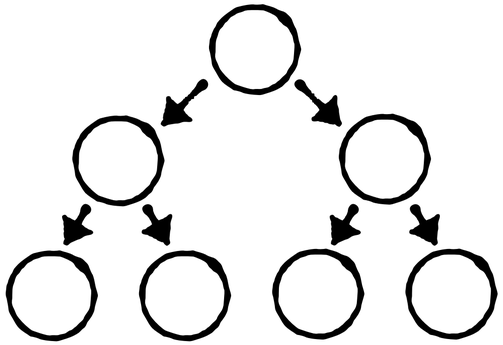
Select * from animals where name = “**gorilla**”

How to find quickly without a full scan?

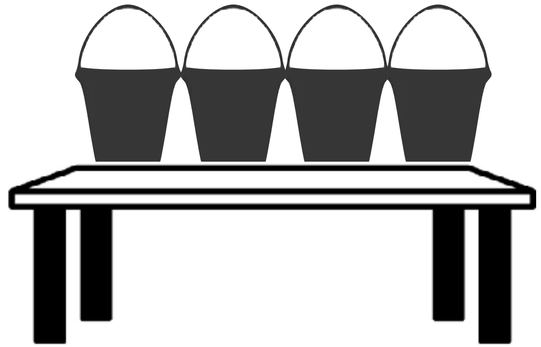


Indexes

You have learned about



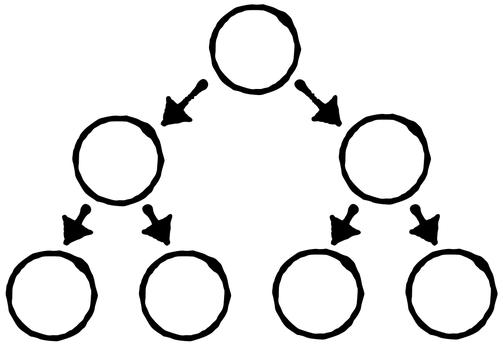
Binary trees



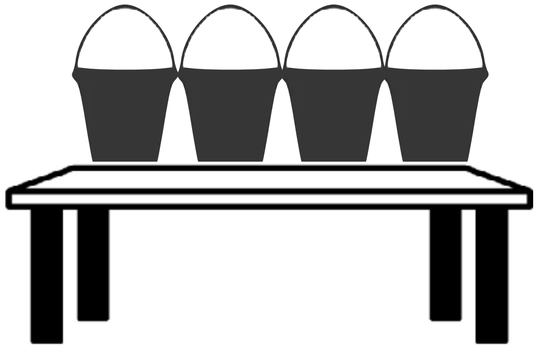
Hash tables

Indexes

You have learned about



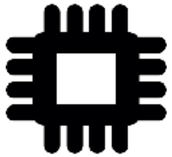
Binary trees



Hash tables

Not a good fit for disks or SSDs (from CSC443)

The memory Hierarchy



CPU caches

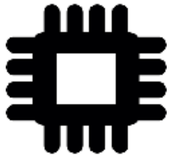
memory

SSD

disk

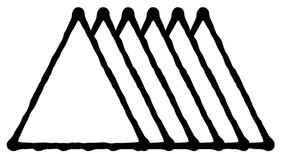
Expensive & fast

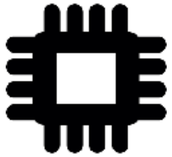
Slow & cheap



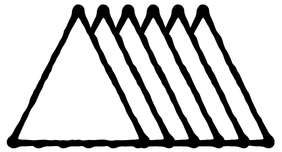
Not enough space

Indexes stored here

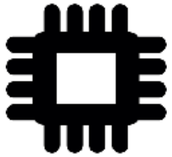




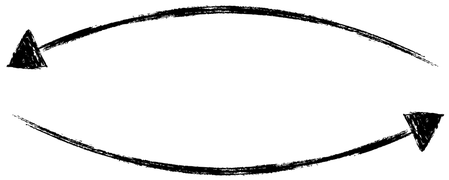
Indexes



Block-addressable
4-16 KB

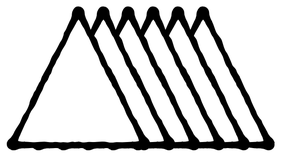


Read I/O



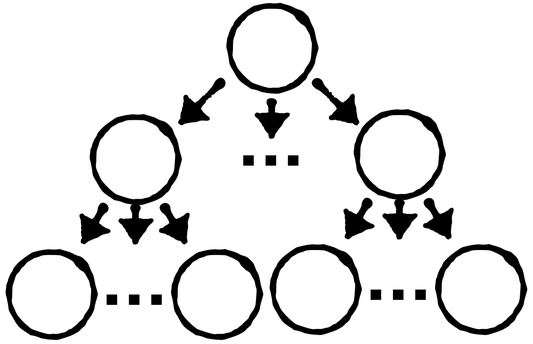
Write I/O

Indexes



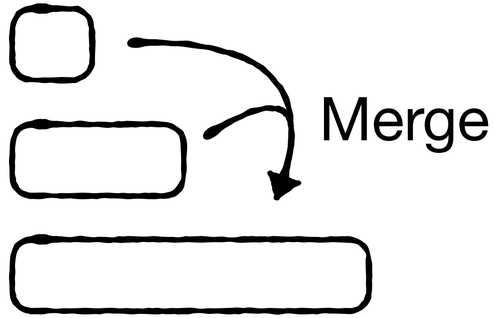
Goal: minimize block I/Os

Indexes for Storage
(in CSC443)



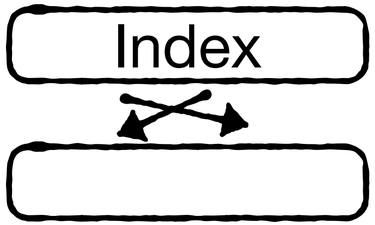
B-Trees

1970



Log-Structured Merge-Trees

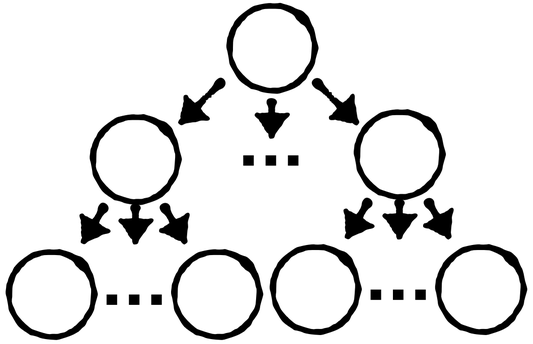
1996



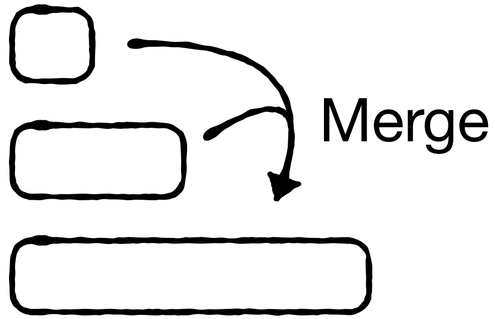
Circular Log

2000’s

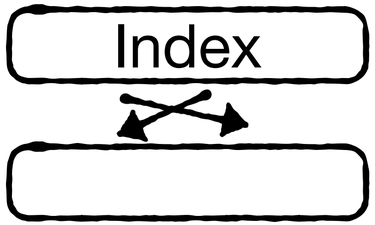
Indexes for Storage
(in CSC443)



B-Trees



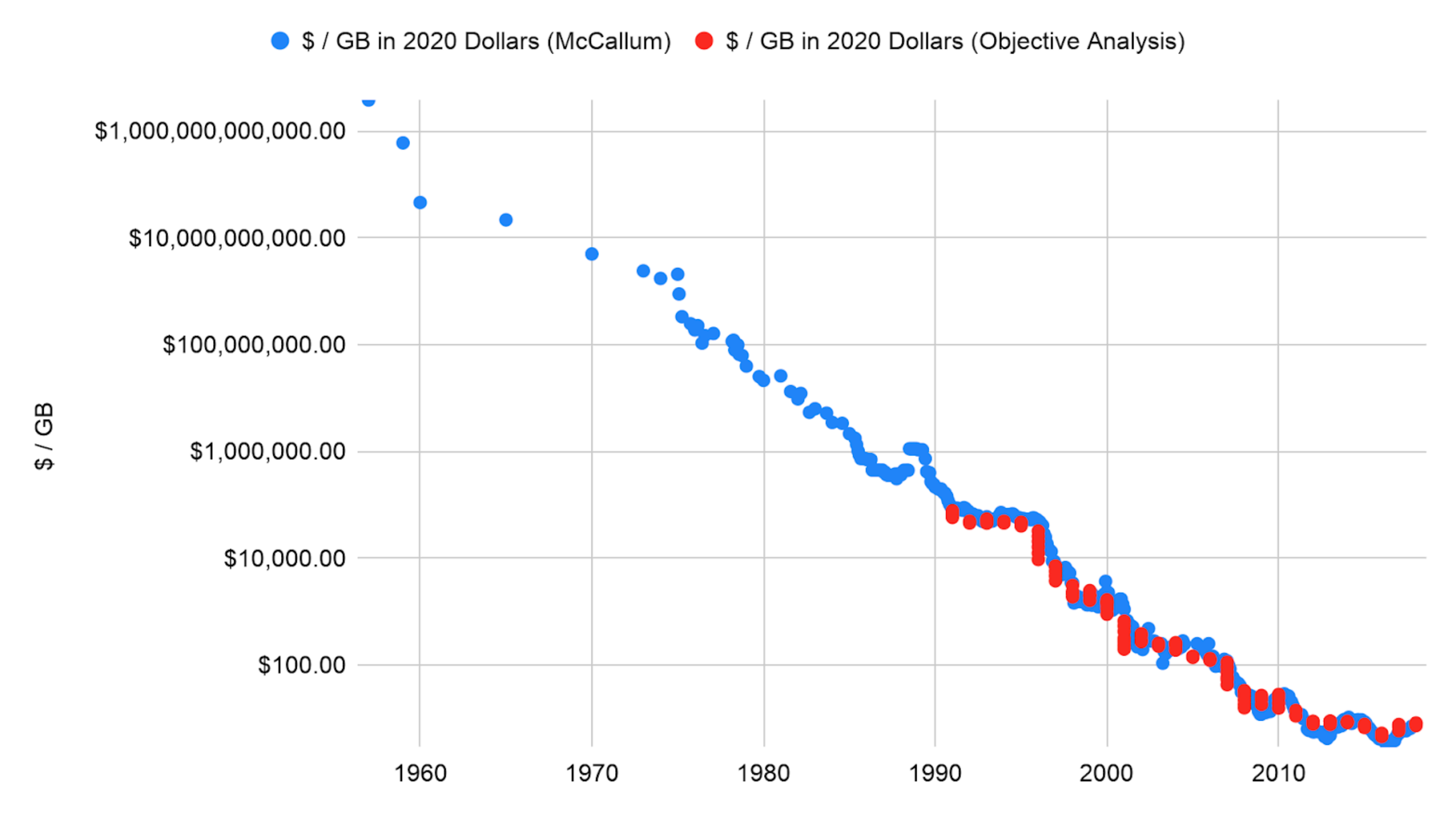
Log-Structured
Merge-Trees

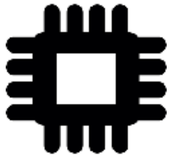


Circular Log

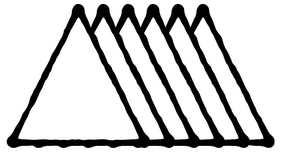
Cheaper queries

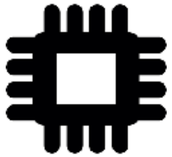
More memory
Cheaper writes


$ / GB in 2020 Dollars (McCallum)
$ / GB in 2020 Dollars (Objective Analysis)
$1,000,000,000,000.00
$10,000,000,000.00
$100,000,000.00
$1,000,000.00
$10,000.00
$100.00
$ / GB
1960
1970
1980
1990
2000
2010

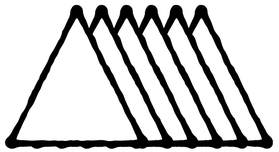



SSD

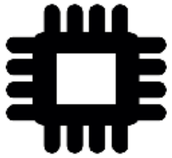




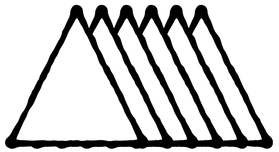
Indexes can fit here

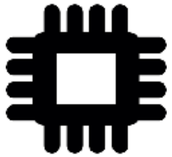


New Design Goals!

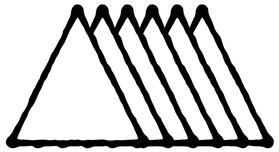


Indexes can fit here



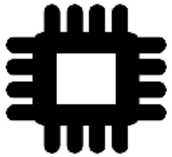


Cache miss
(≈100 cycles)



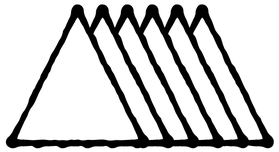
CPU register

Goal 1: Minimize Cache Misses



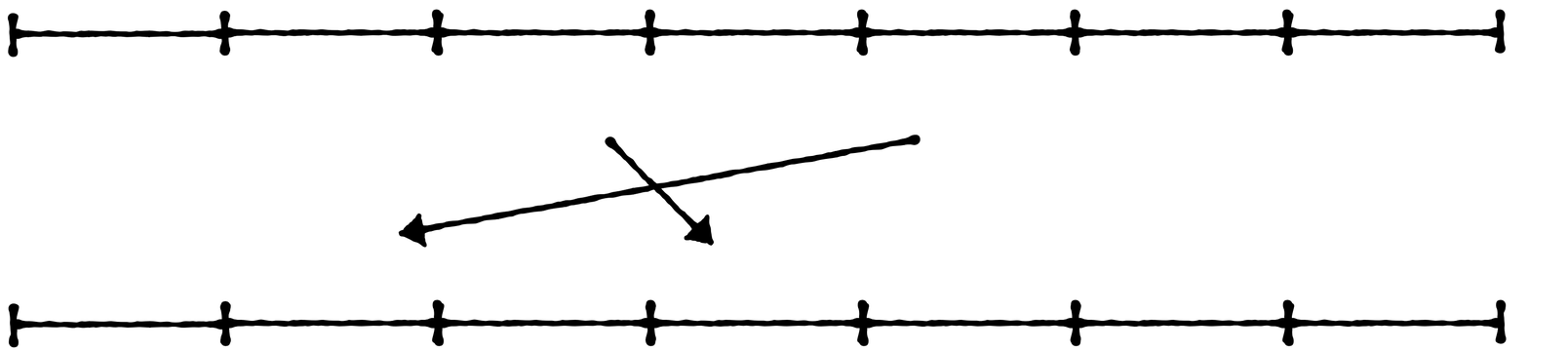
Cache miss
(≈100 cycles)

CPU register

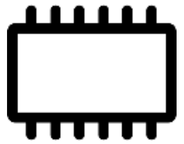



Physical DRAM 4KB pages

Virtual Memory in 4KB pages

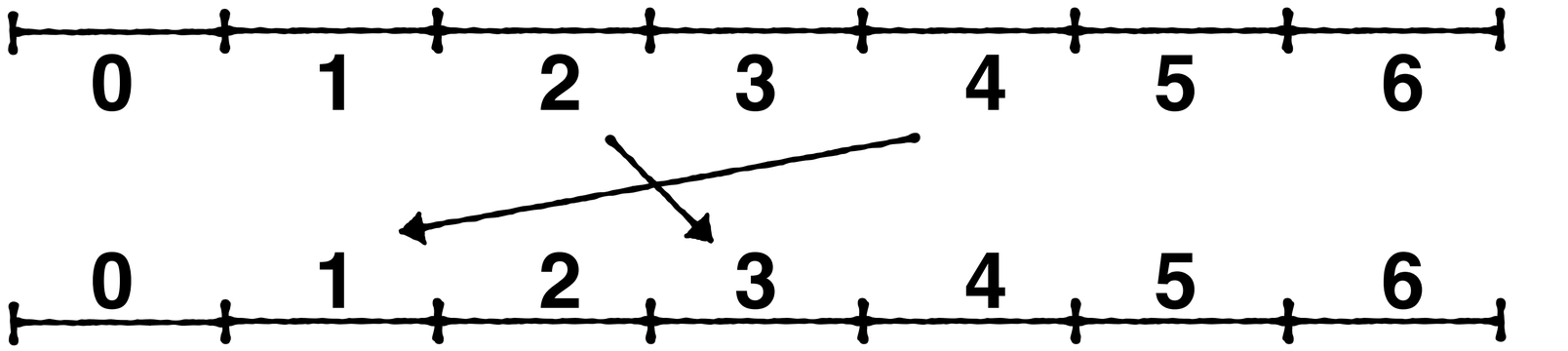


Physical DRAM 4KB pages



Virtual Memory in 4KB pages

Mapping



Physical DRAM 4KB pages

Mapping

…

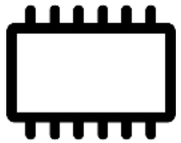
2 → **3**

…

4 → **1**

…

Stored in known location in physical DRAM



…

2 → 3

…

Translation Lookaside Buffer (TLB) in SRAM

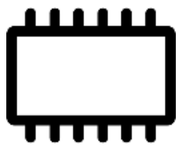
Mapping

…

4 → 1

…

Stored in known location in physical DRAM



…

2 → 3

…

Translation Lookaside Buffer (TLB) in SRAM

↑ **miss** (≈100 cycles)

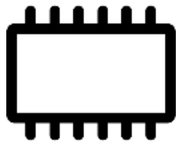
Mapping

…

4 → 1

…

Stored in known location in physical DRAM



Goal 2: Minimize TLB Misses

…

2 → **3**

…

Translation Lookaside Buffer (TLB) in SRAM

miss (≈100 cycles)

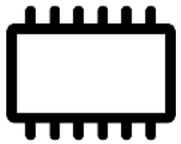
Mapping

…

4 → 1

…

Stored in known location in physical DRAM



Design goals

Minimize Cache Misses

Minimize TLB Misses

Design goals

Minimize Cache Misses

Minimize TLB Misses

Maximize Parallelism (SIMD & multi-threading)

Design goals

Minimize Cache Misses

Minimize TLB Misses

Maximize Parallelism

Minimize Space overheads

Terms

N: # entries in dataset

B: # entries per cache line

Terms

N: # entries in dataset

B: # entries per cache line

Assume a cache line is 128B and a key is 4B

B = 32

Terms

N: # entries in dataset

B: # entries per cache line

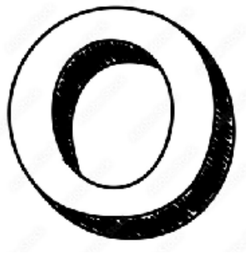
Assume a cache line is 128B and a key is 4B

B = 32

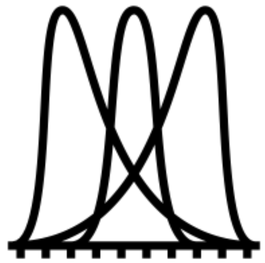
Block size is far smaller than in disk access model

Improvements Along Two Areas

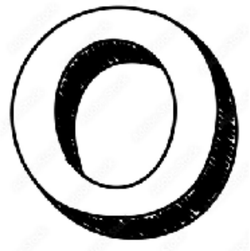
Better Worst-Case



Exploiting Data Distribution



Better Worst-Case



AVL-Tree
1962

B-Tree
1970

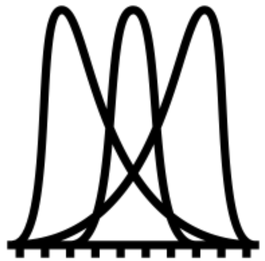
T-Tree
1985

CSS-Tree
1998

CSB-Tree
2000

HOT
2018

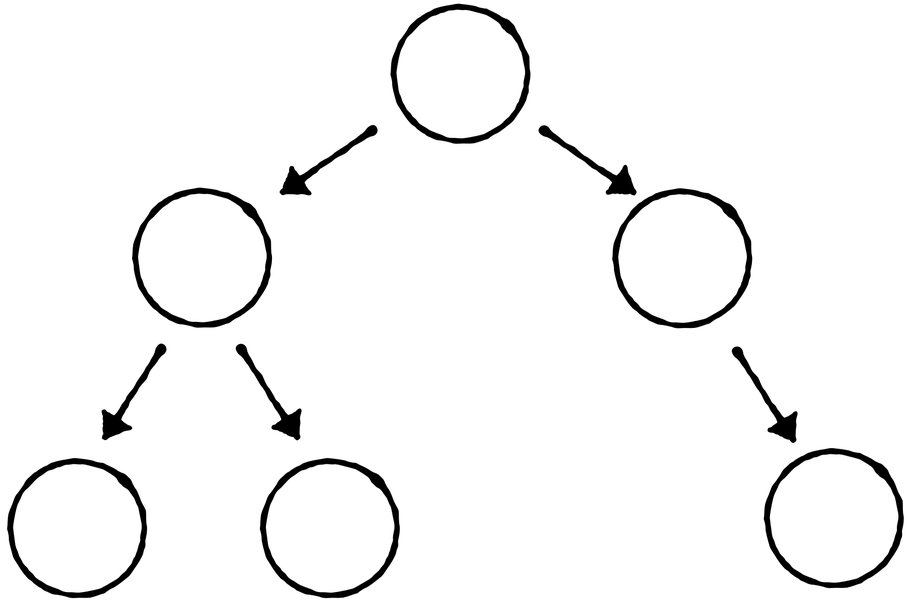
Exploiting Data Distribution



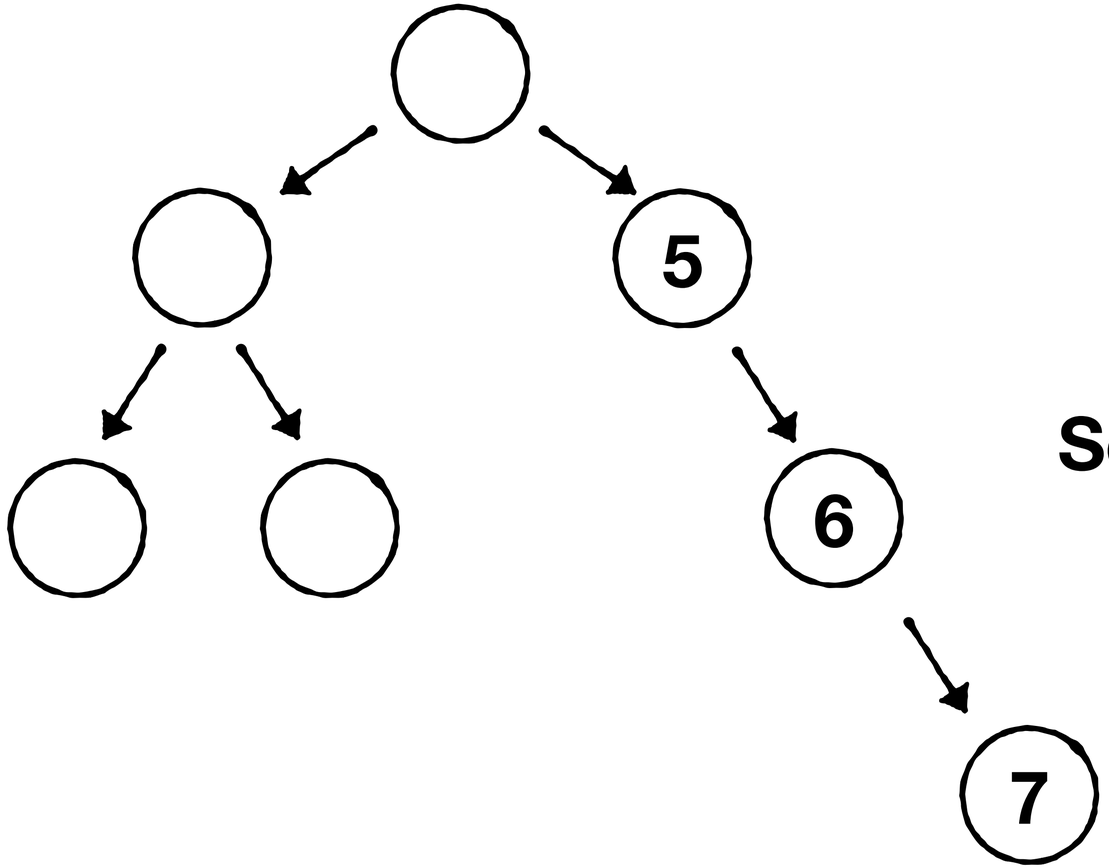
Interpolation search
1959

FITing-Tree
2019

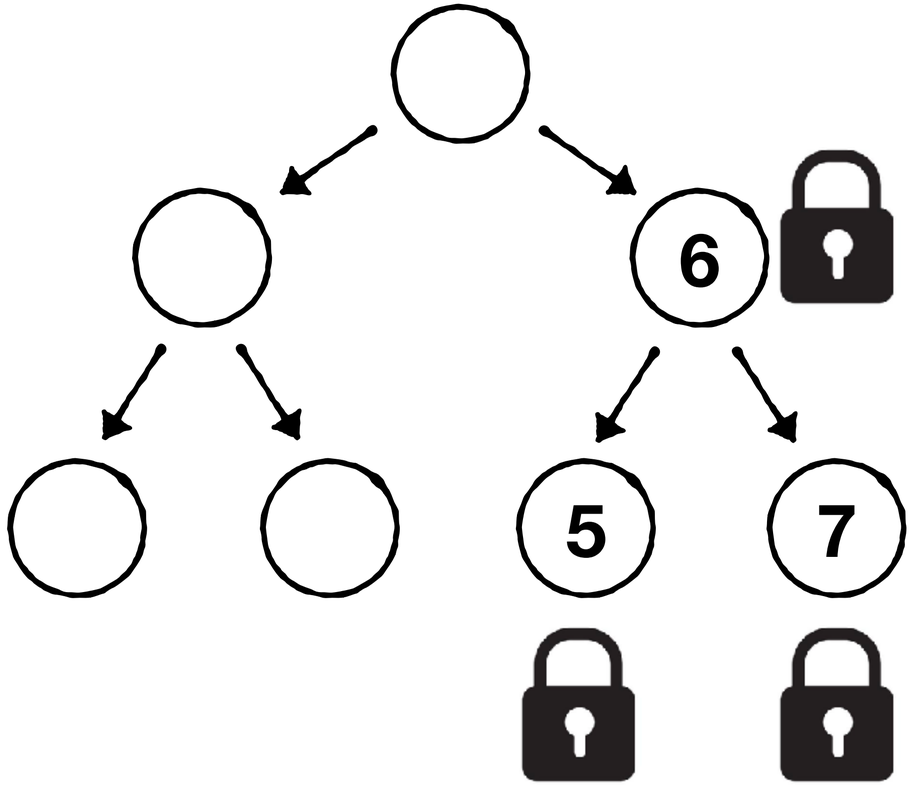
AVL-Tree



AVL-Tree

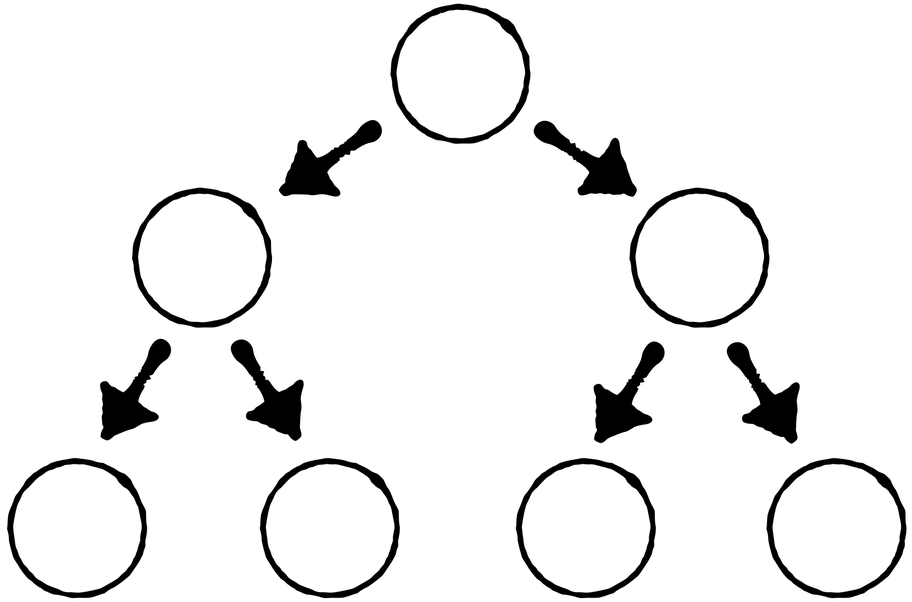


Self-balance on inserts

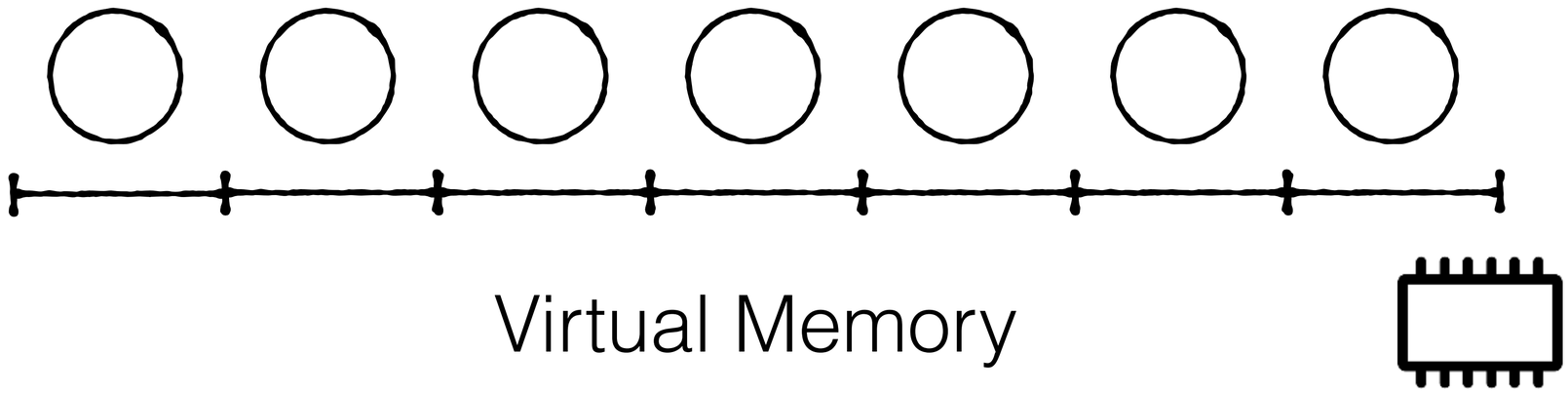


Requires holding multiple latches (damages concurrency)

Pointers create >3X metadata overhead

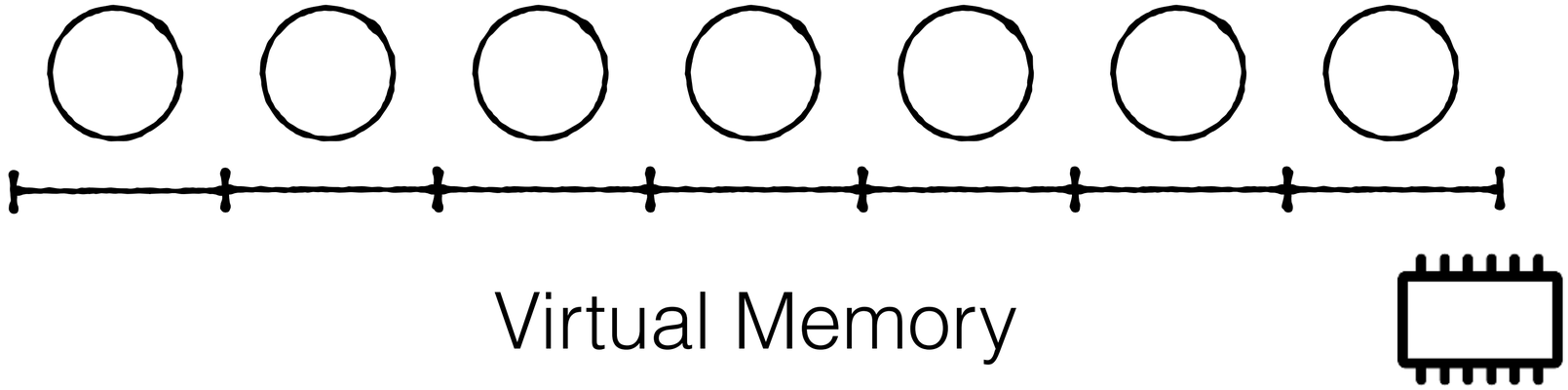


Each node may be on a different virtual page

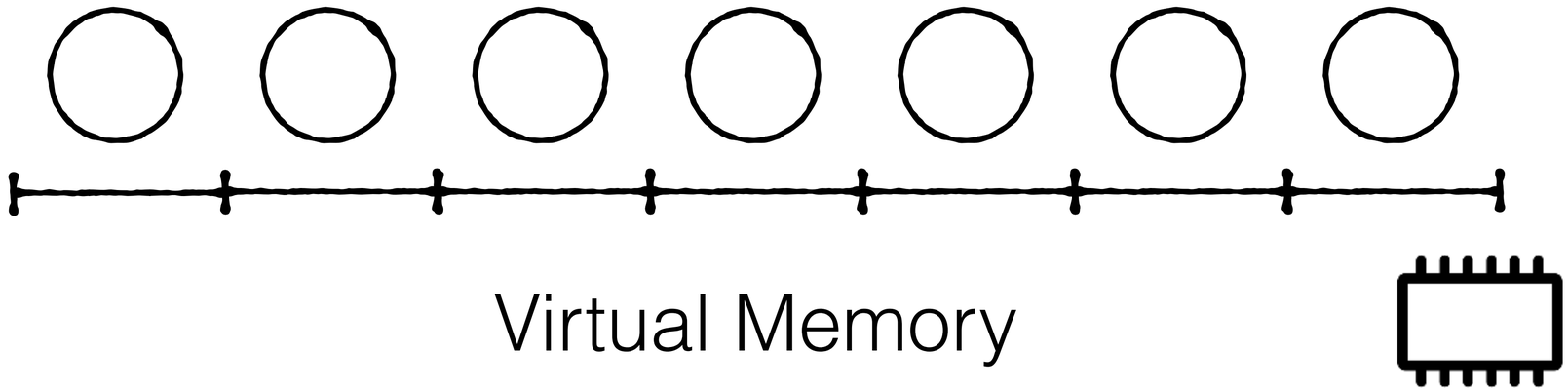


Each node may be on a different virtual page

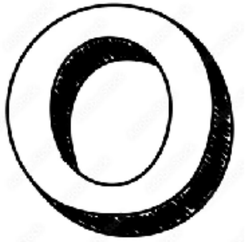
Worst case 1 TLB miss and 1 cache miss per node (≈200 cycles)



$2 \cdot \log_2(N)$ **Cache misses** per search

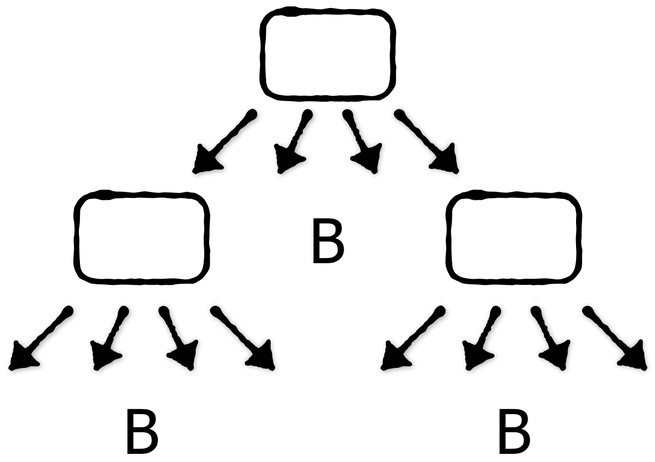


Better Worst-
Case

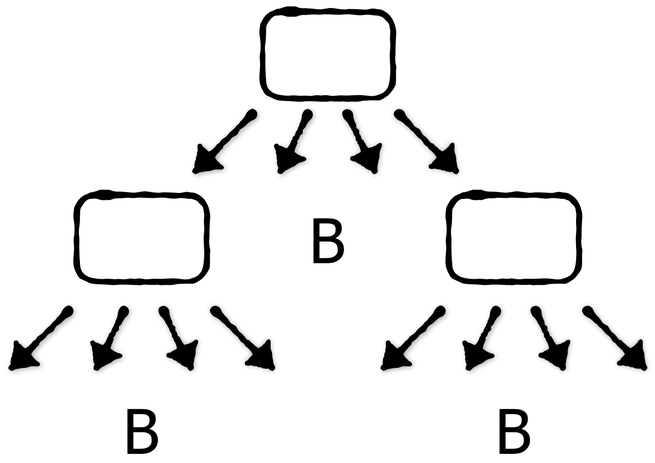


AVL-Tree	**B-Tree**	**T-Tree**	**CSS-Tree**	**CSB-Tree**	**HOT**
1962	1970	1985	1998	2000	2018

B-Tree

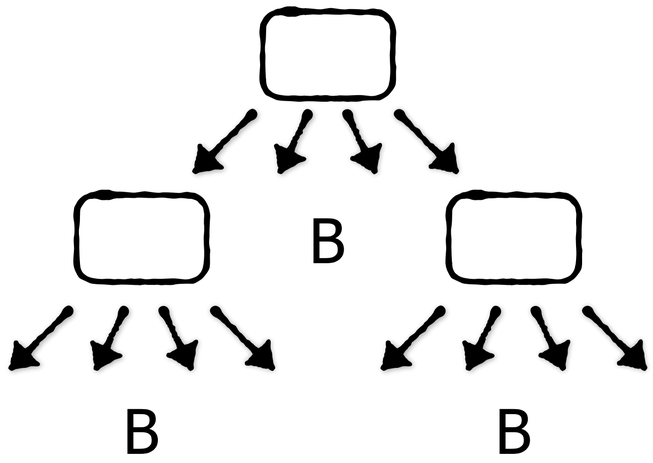


B-Tree



Set each node to be the size of a cache line

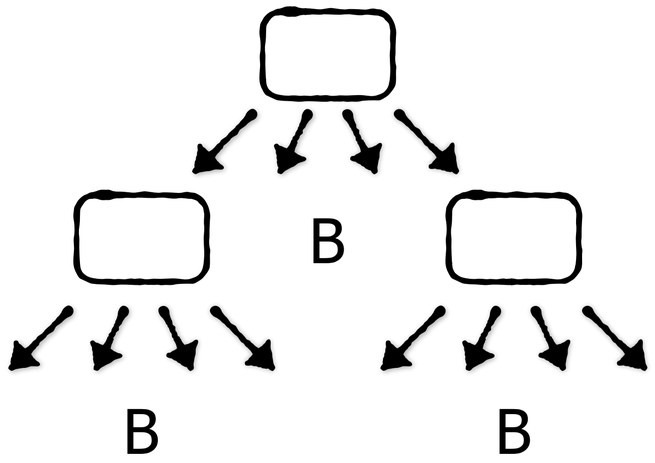
B-Tree



Set each node to be the size of a cache line

We expect $O(\log_B N)$

B-Tree

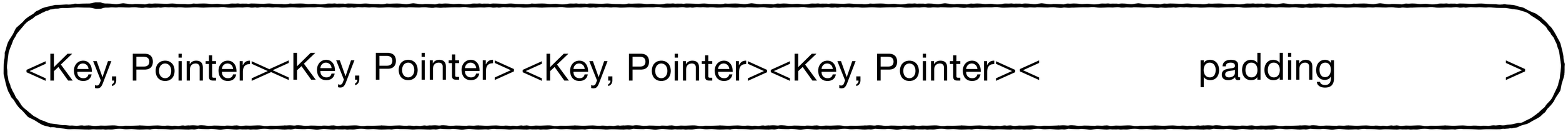


Set each node to be the size of a cache line

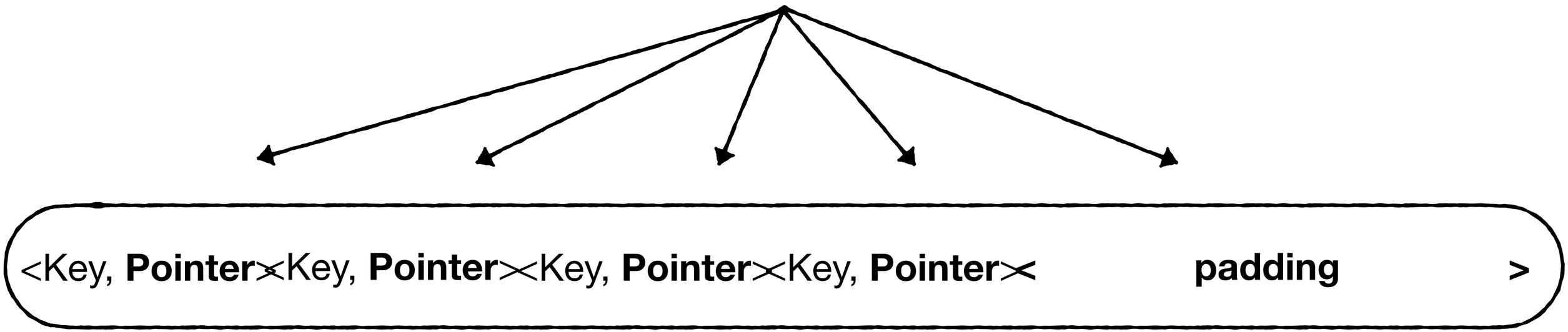
We expect $O(\log_B N)$

Do we get it? Why or why not?

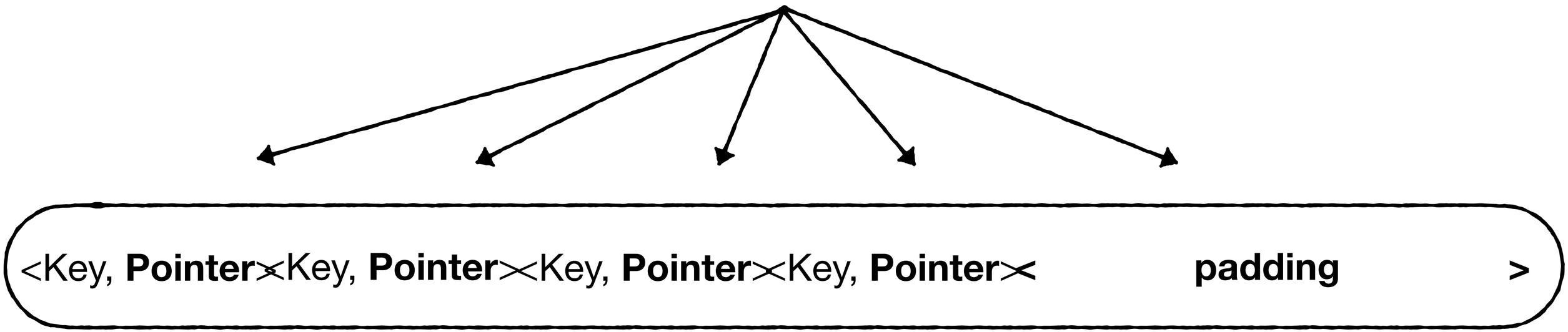
B-trees Page Organization



Space overheads

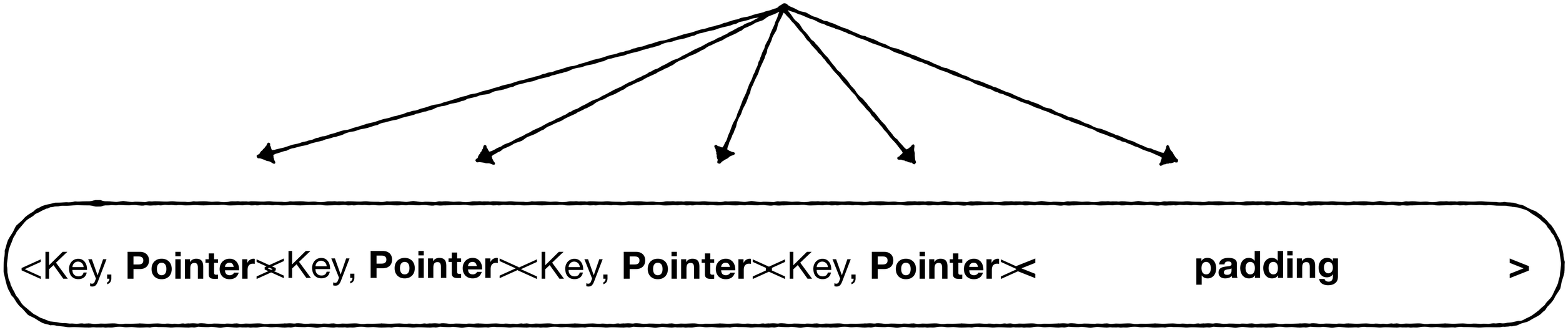


Space overheads



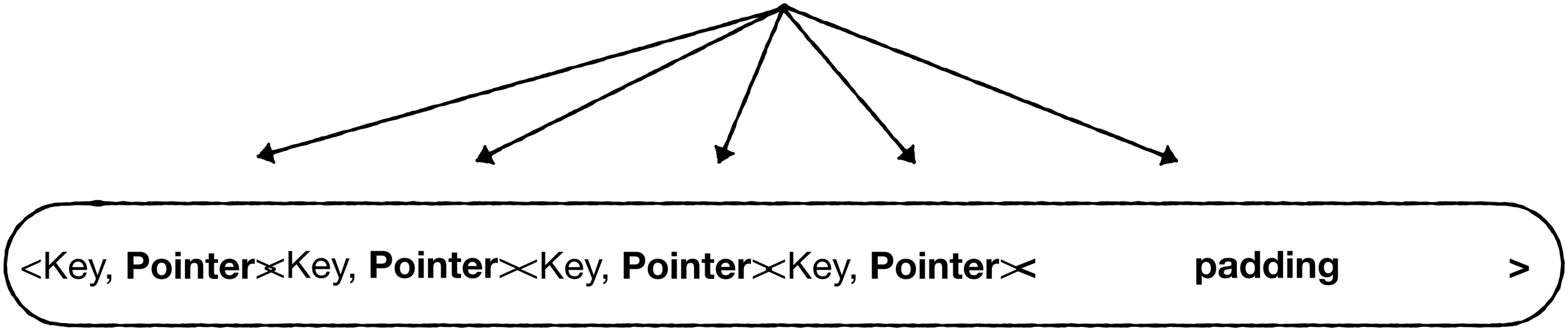
Real fanout: B/4 **(e.g., 8 rather than 32)**

Space overheads



log $_{B/4}$ (N) cache misses

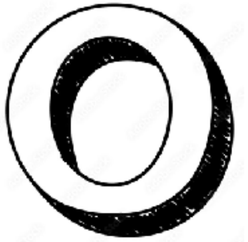
Space overheads



$\log_{B/4}(N)$ cache misses

Space overheads also harm scans

Better Worst-
Case



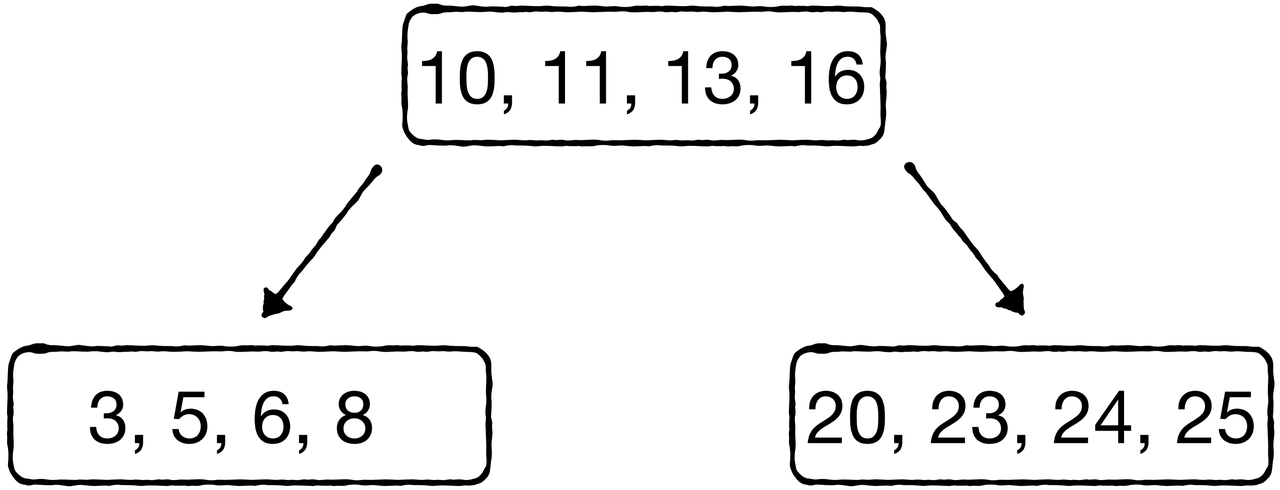
AVL-Tree	**B-Tree**	**T-Tree**	**CSS-Tree**	**CSB-Tree**	**HOT**
1962	1970	1985	1998	2000	2018

T-Tree

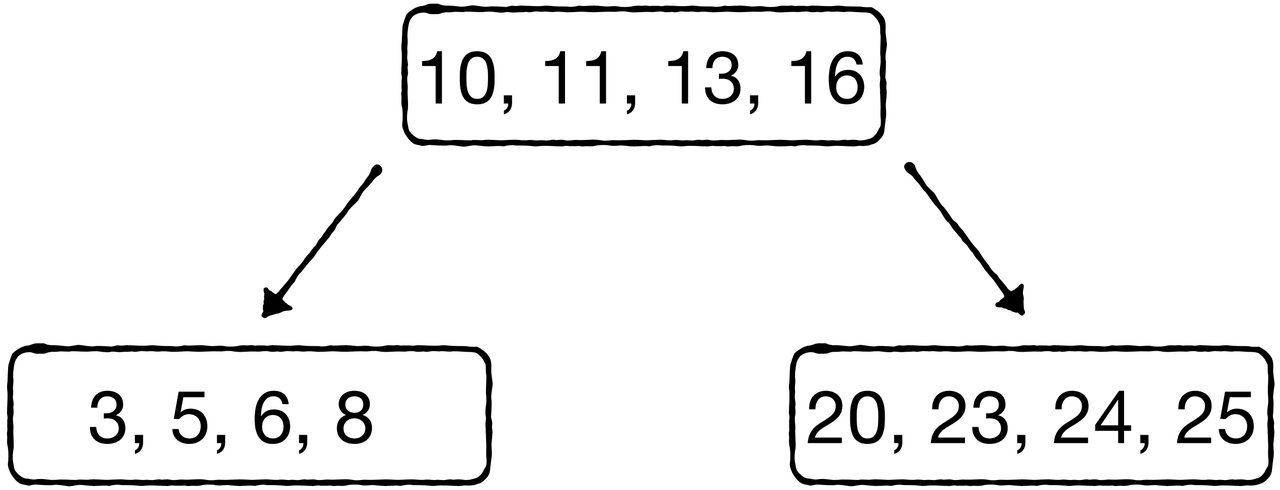
A study of index structures for main memory database management systems

Technical Report. 1985

T-Tree: **an AVL-Tree with Fat Nodes**

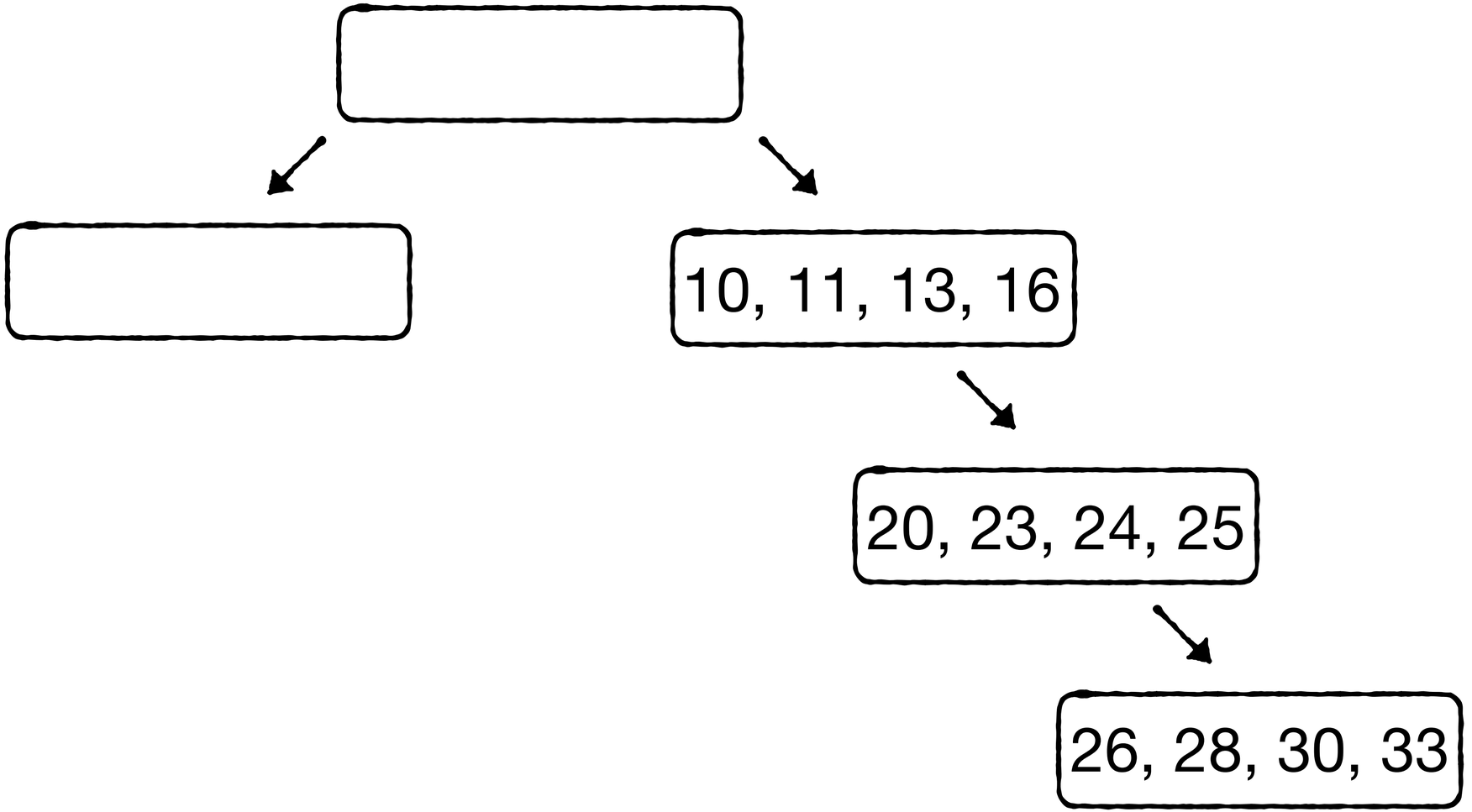


T-Tree: an AVL-Tree with Fat Nodes

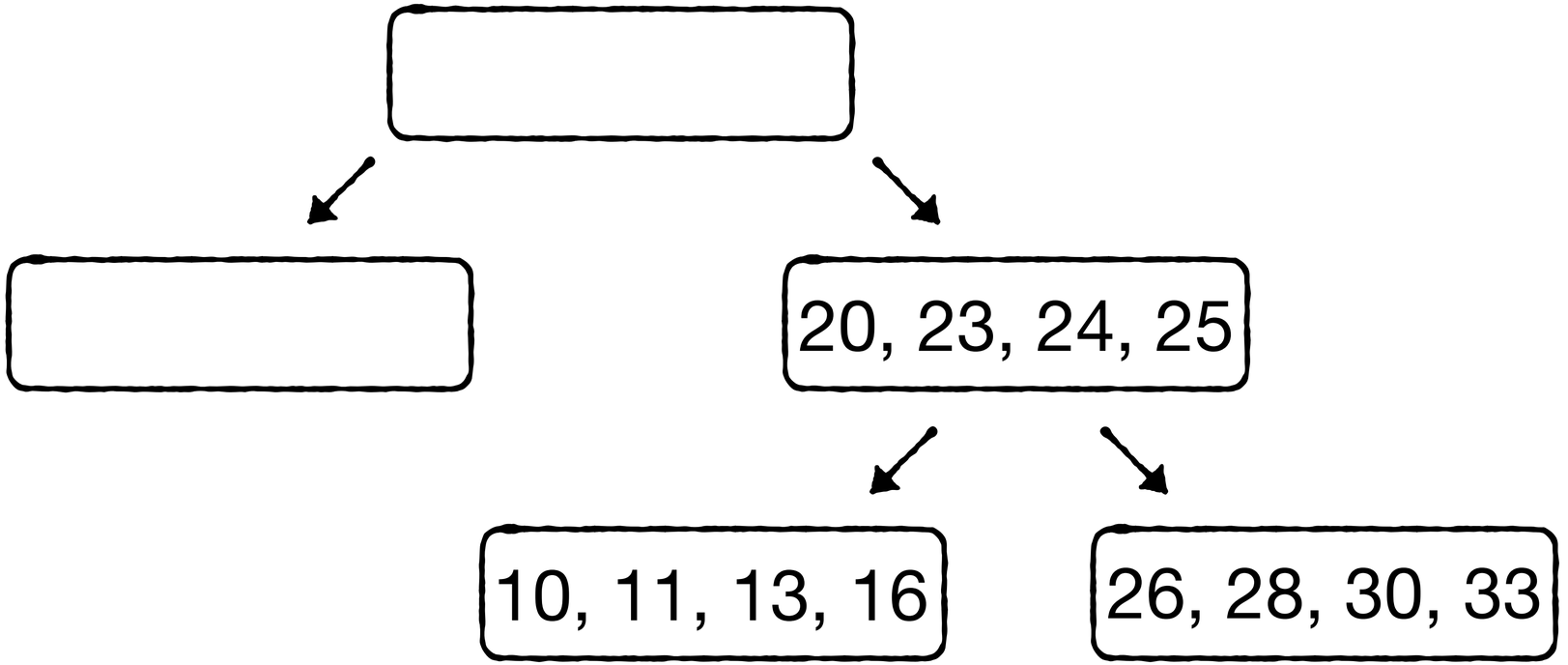


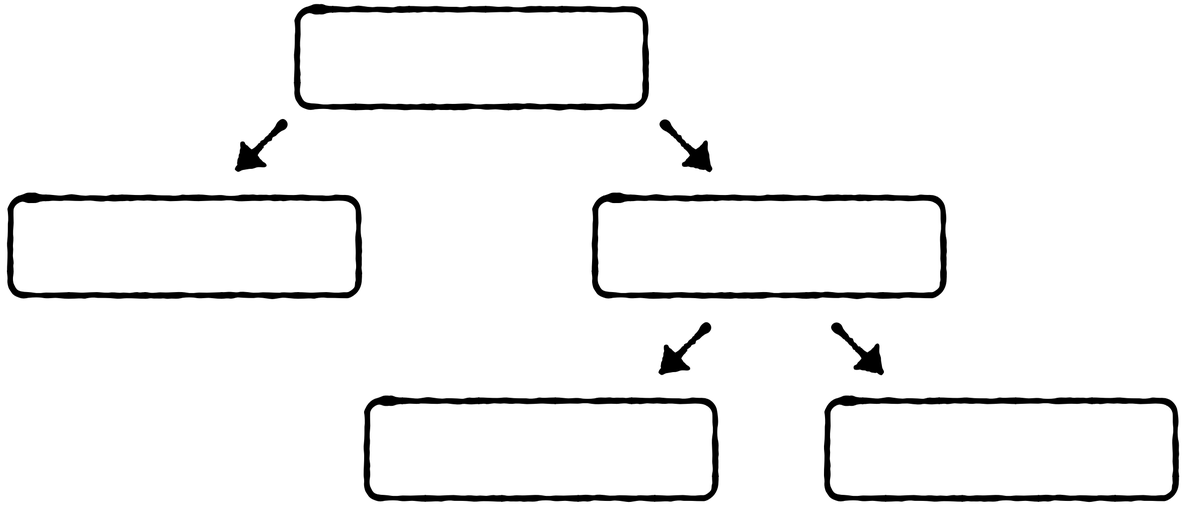
A node can’t intersect with its sub-trees in key range

Self-Balancing is Identical to AVL-Tree

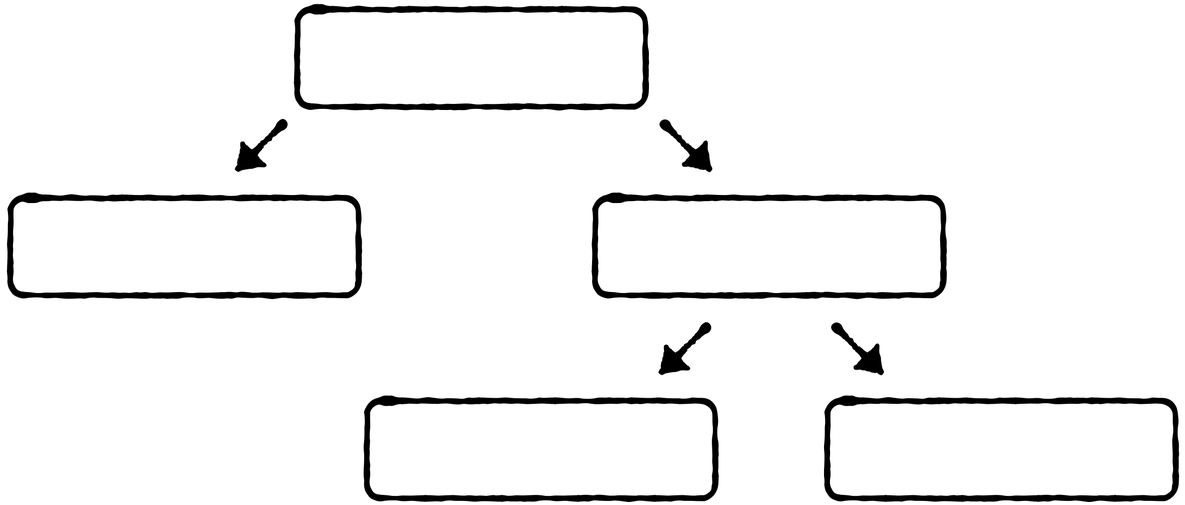


Self-Balancing is Identical to AVL-Tree





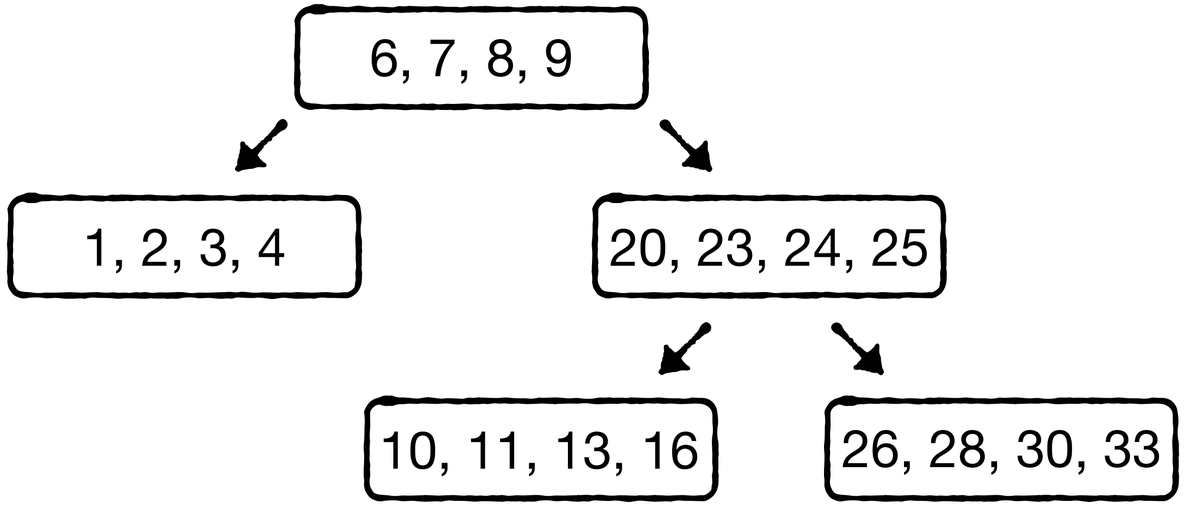
Pros: (1) Less metadata relative to data



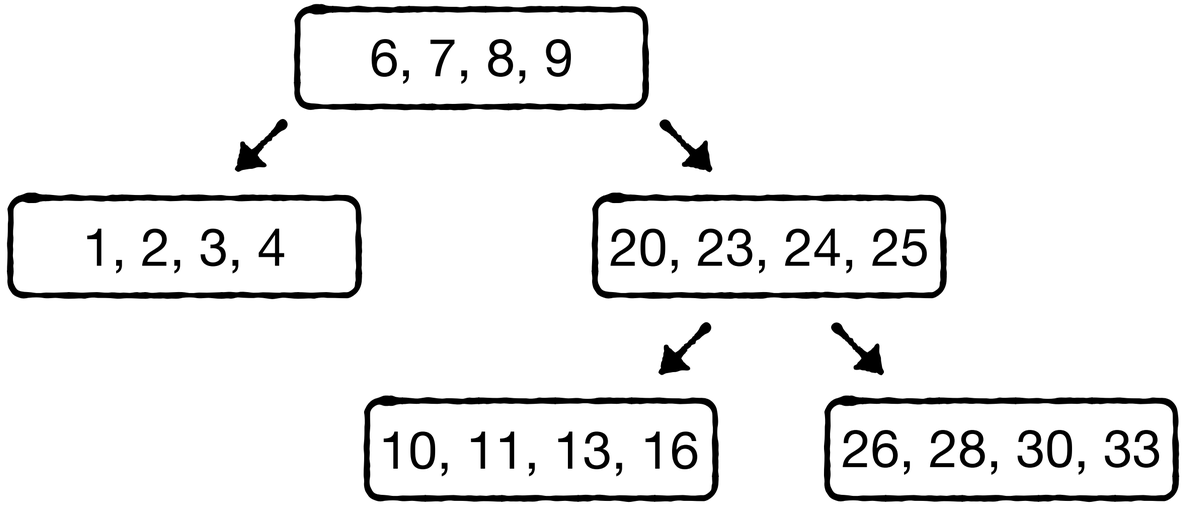
Pros: (1) Less metadata relative to data

(2) Fewer rotations

search cost? (In terms of B and N)

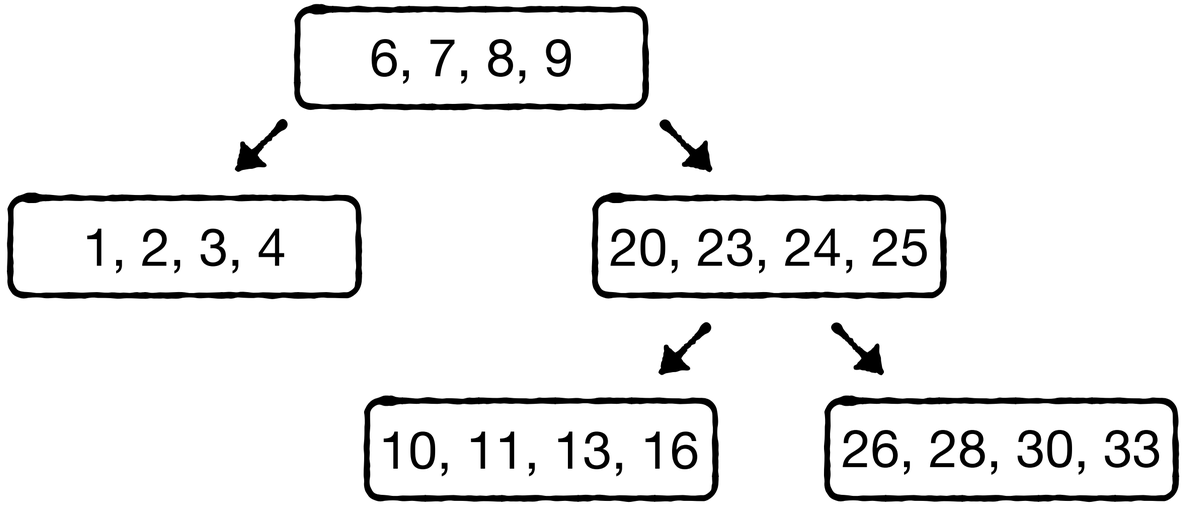


search cost? (In terms of B and N)



We only prune the search space by 2x per node

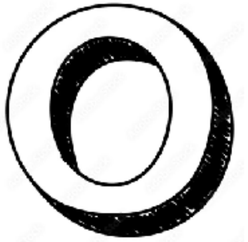
search cost? (In terms of B and N)



We only prune the search space by 2x per node

$O(\log_2(N/B))$ cache misses

Better Worst-
Case



AVL-Tree	**B-Tree**	**T-Tree**	**CSS-Tree**	**CSB-Tree**	**HOT**
1962	1970	1985	1998	2000	2018

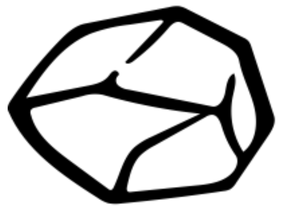
Cache-Sensitive Search-Tree (CSS-Tree)

Cache-Sensitive Search-Tree (CSS-Tree)

Cache conscious indexing for decision-support in main memory
VLDB 1999.

Cache-Sensitive Search-Tree (CSS-Tree)

Cache conscious indexing for decision-support in main memory
VLDB 1999.



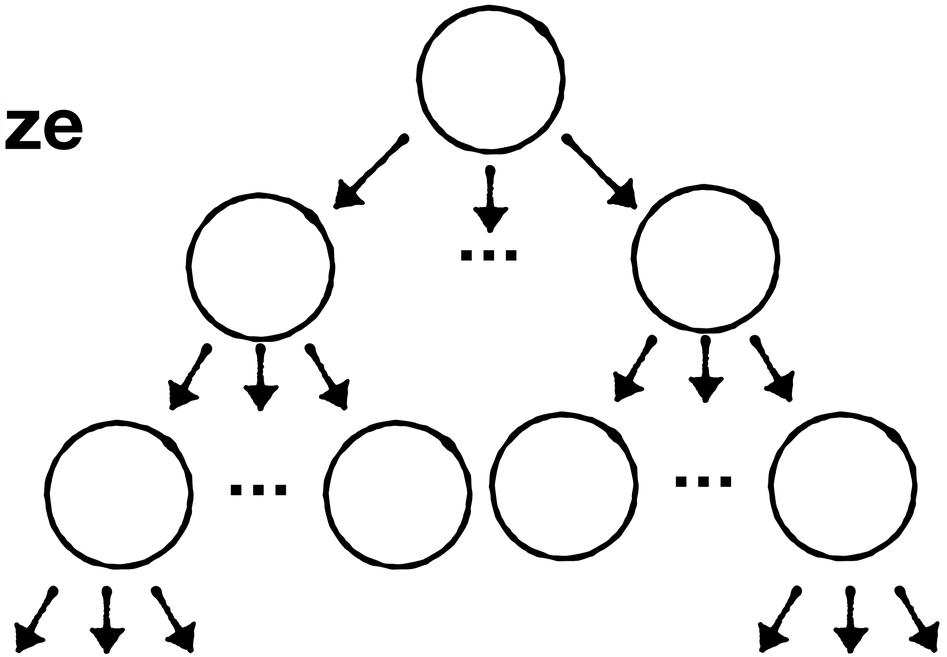
static data

Cache-Sensitive Search-Tree (CSS-Tree)

static sorted array

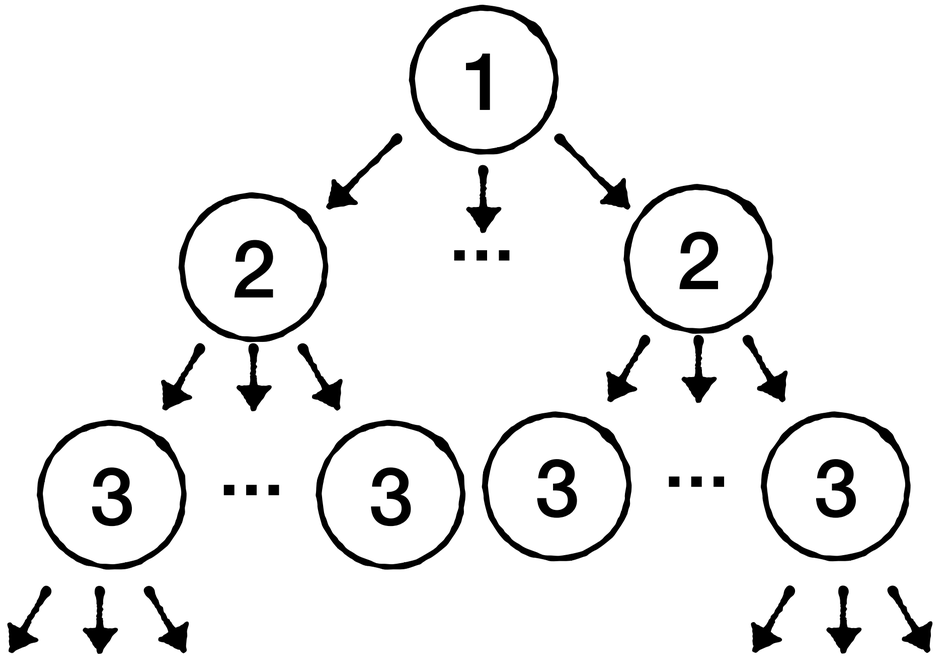
Cache-Sensitive Search-Tree (CSS-Tree)

Each node the size of a cache line



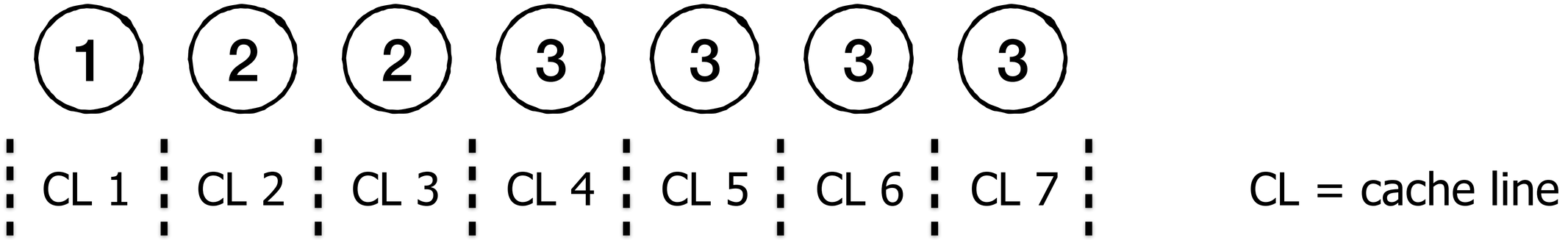
static sorted array

Cache-Sensitive Search-Tree (CSS-Tree)



static sorted array

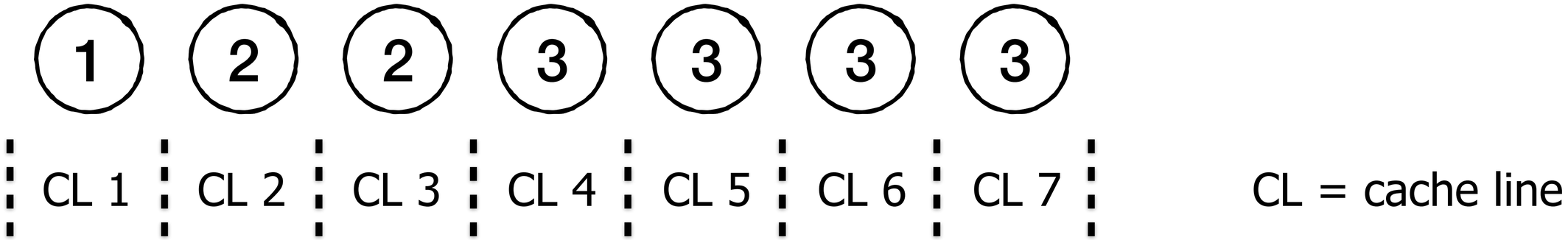
Stored as dense array in breadth-first order



static sorted array

Stored as dense array in breadth-first order

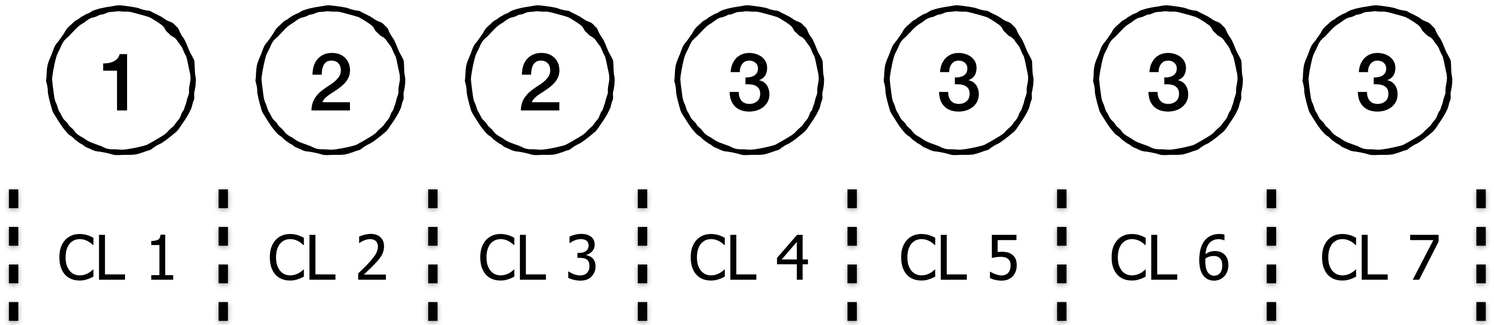
No padding or pointers among nodes (arithmetic is used to find child)



Stored as dense array in breadth-first order

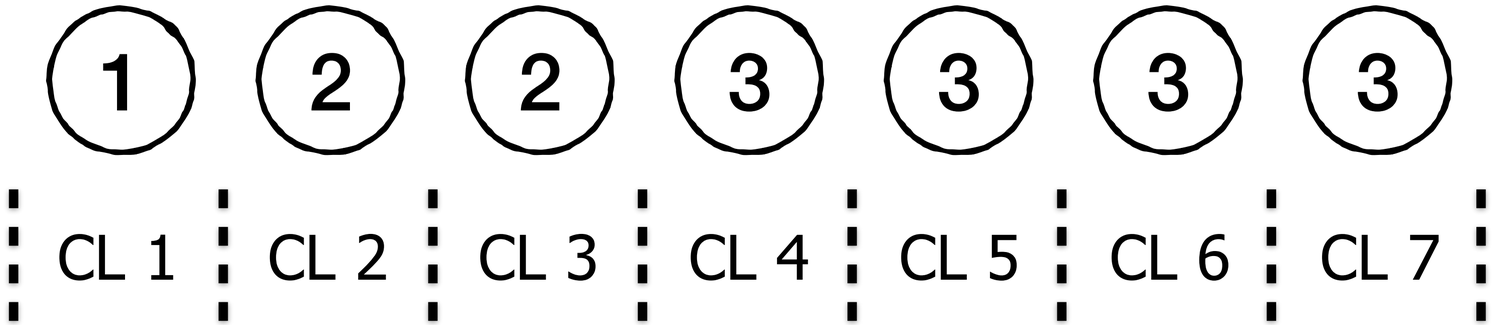
No padding or pointers among nodes (arithmetic is used to find child)

Log_B N Cache misses

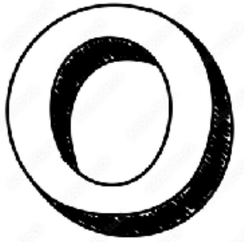


CL = cache line

Requires full reconstruction to handle updates

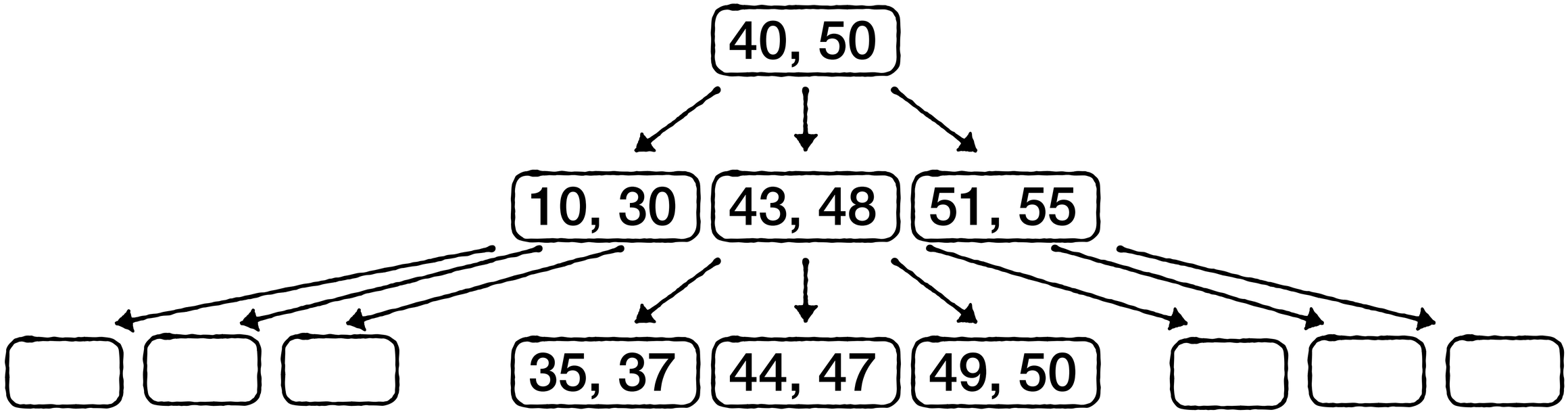


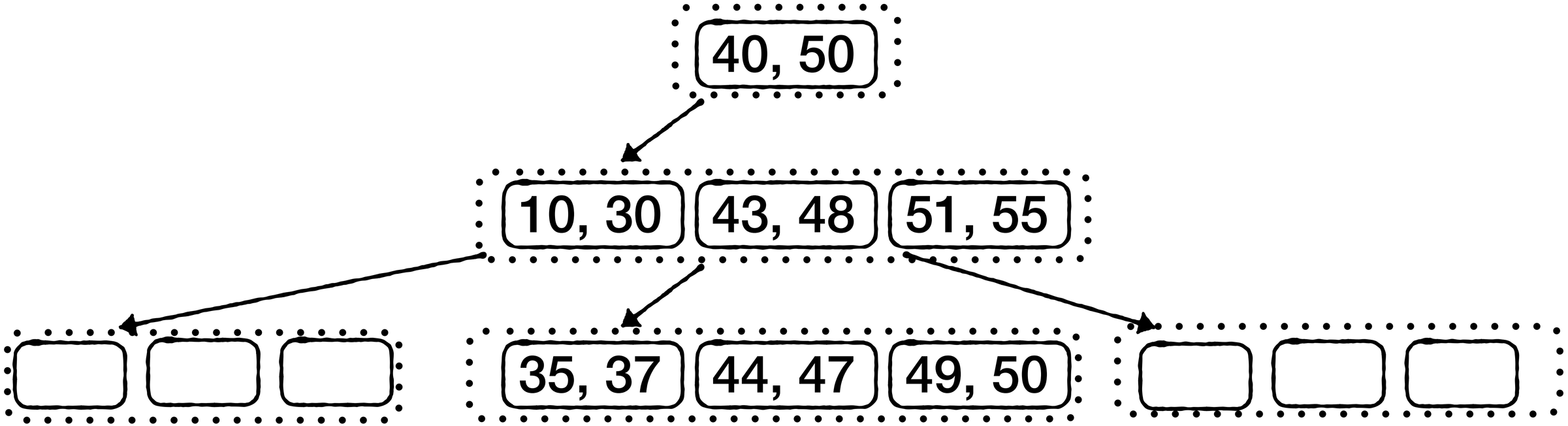
Better Worst-
Case



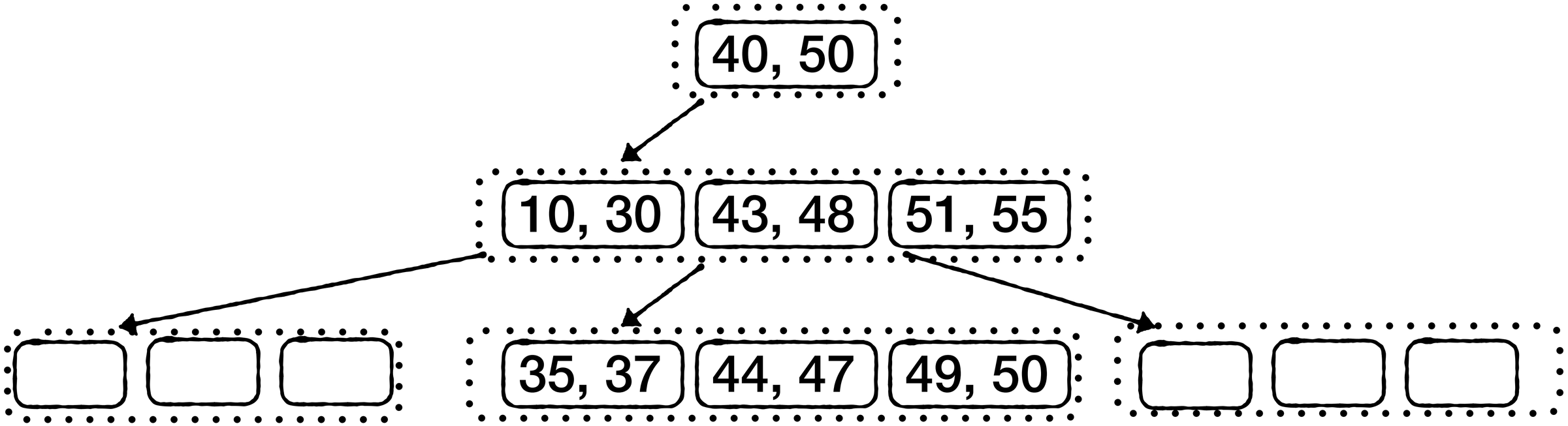
AVL-Tree	**B-Tree**	**T-Tree**	**CSS-Tree**	**CSB-Tree**	**HOT**
1962	1970	1985	1998	2000	2018

Cache-Sensitive B-Tree (CSB-Tree)



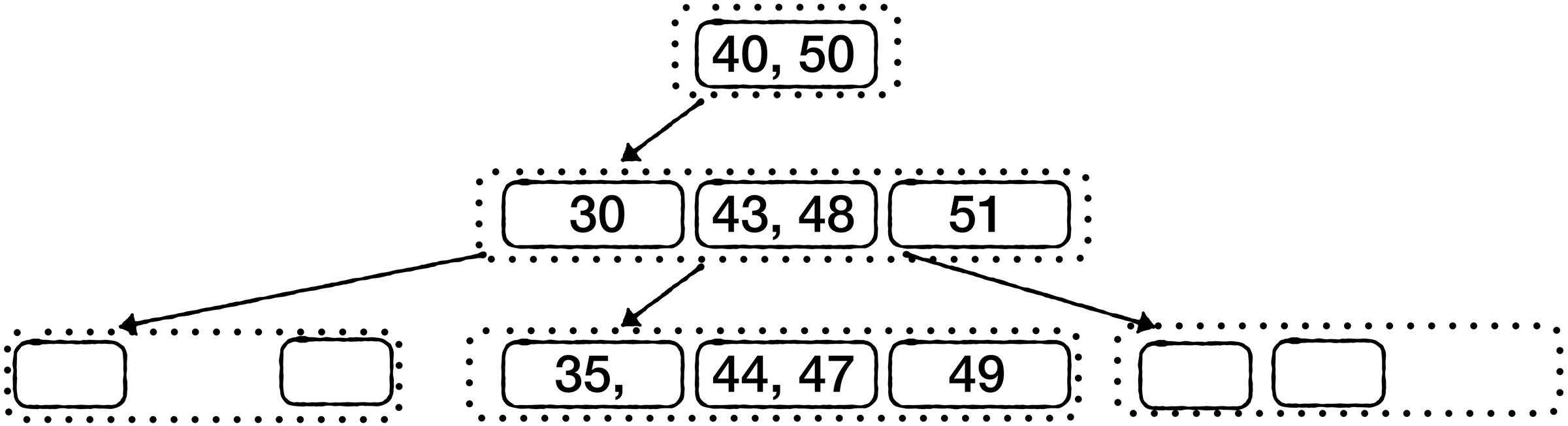


All children of a node are stored contiguously in “node group”



All children of a node are stored contiguously in “node group”

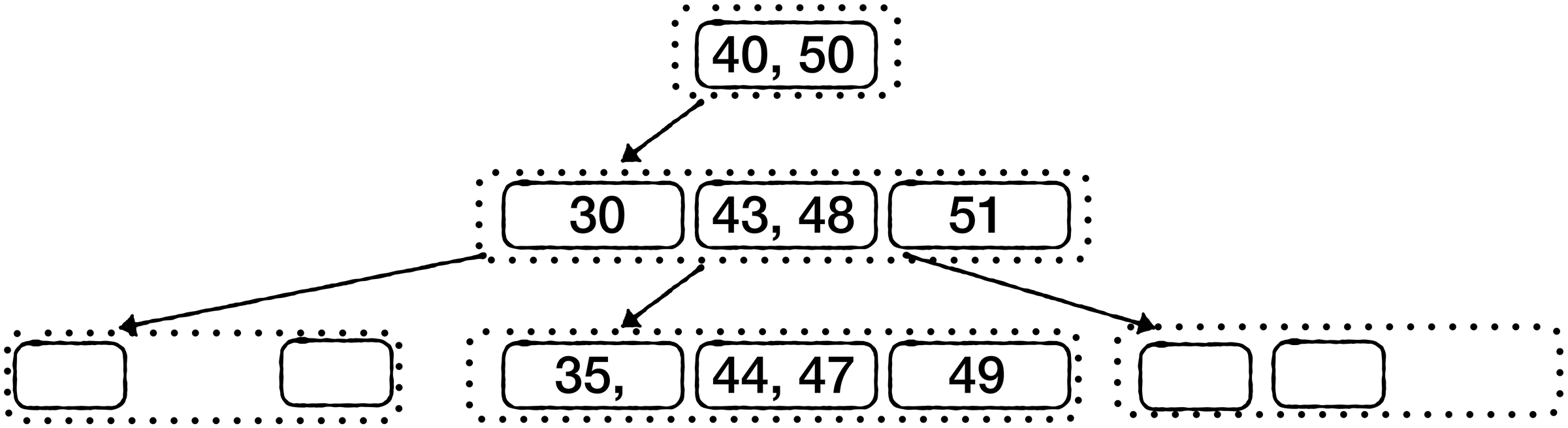
Most pointers are eliminated (only one needed per node group)



All children of a node are stored contiguously in “node group”

Most pointers are eliminated (only one needed per node group)

Padding is still needed to absorb insertions

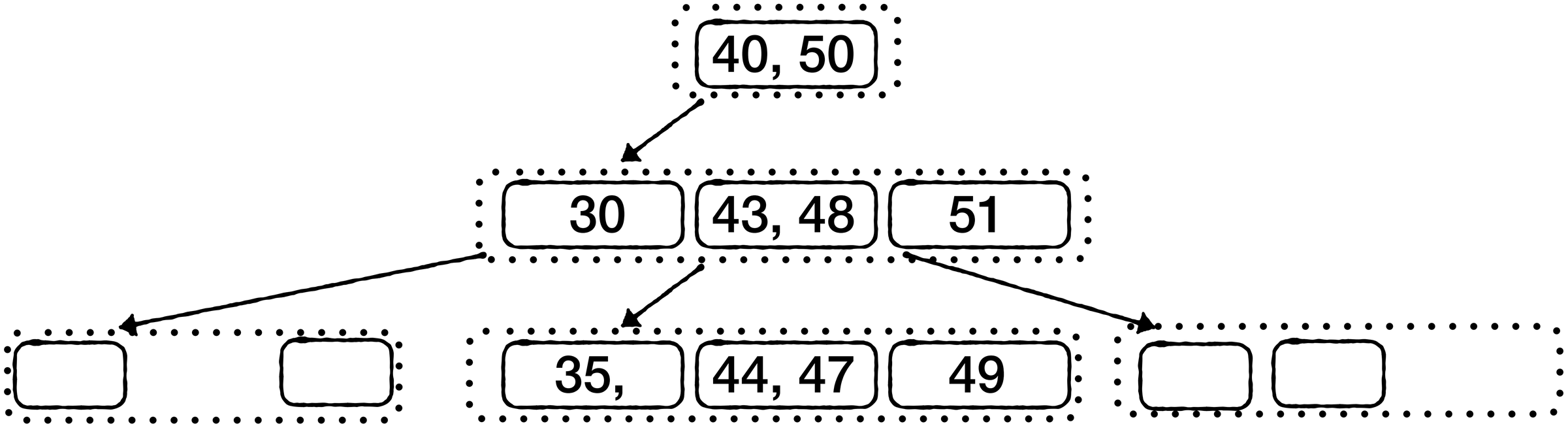


All children of a node are stored contiguously in “node group”

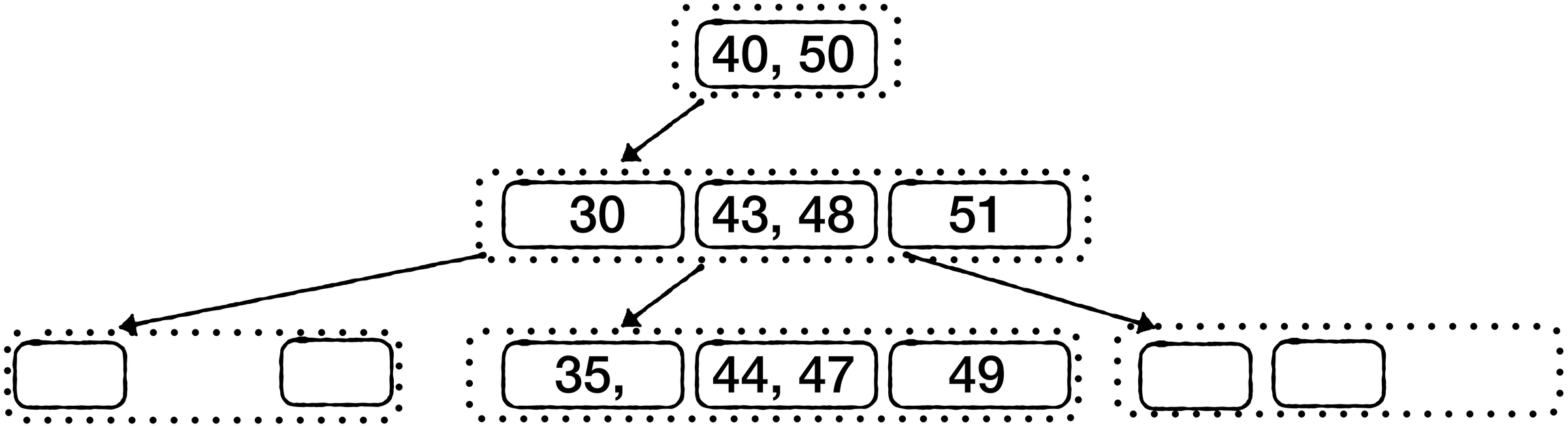
Most pointers are eliminated (only one needed per node group)

Padding is still needed to absorb insertions

B nodes per node group due to fanout of tree



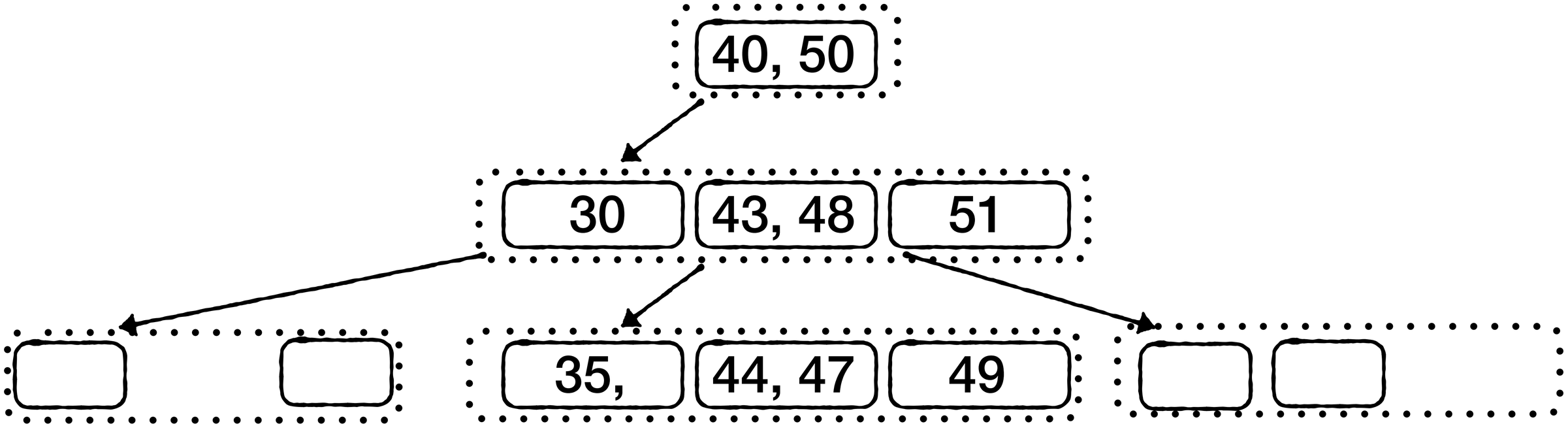
Fanout is B/2 rather than B/4



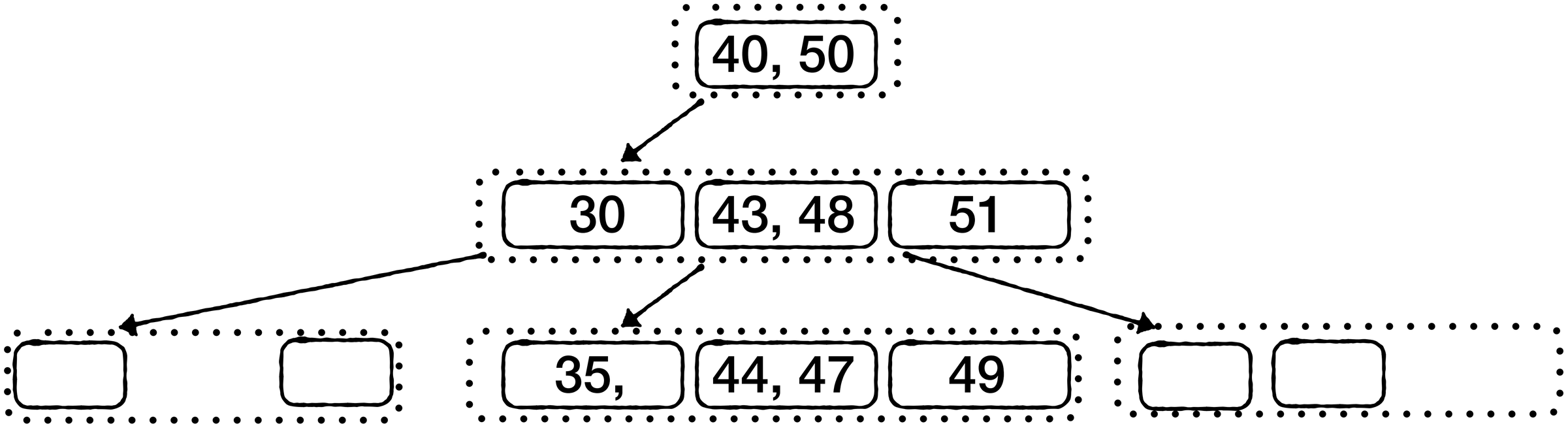
Fanout is B/2 rather than B/4

Depth: $O(\log_{B/2}(N))$

Write cost?

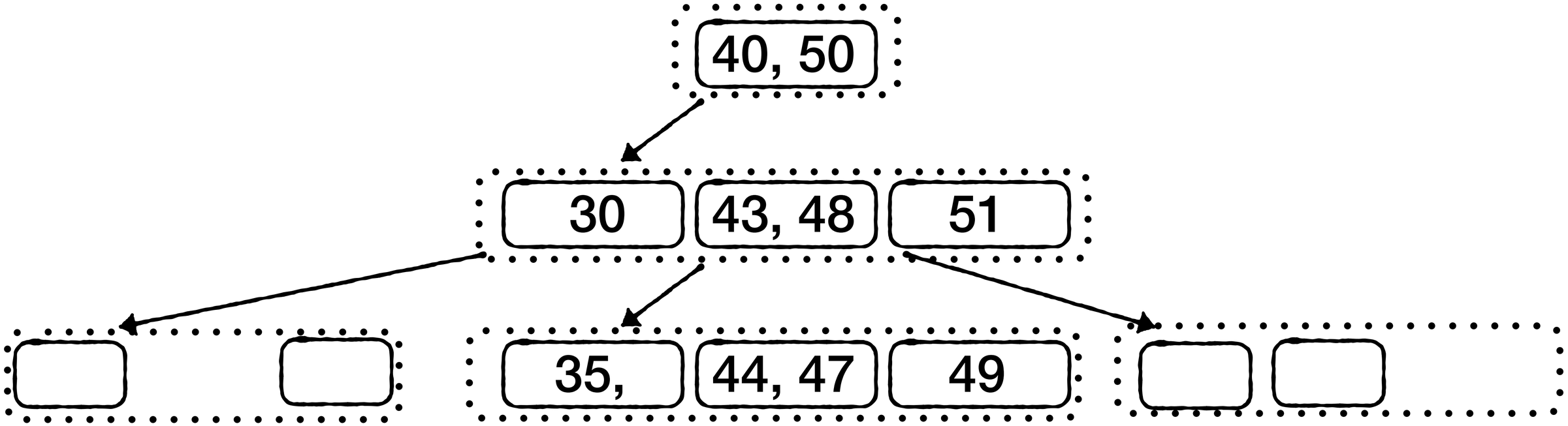


Write cost?



Every B/2 insertions, a leaf splits, causing us to rewrite a node group

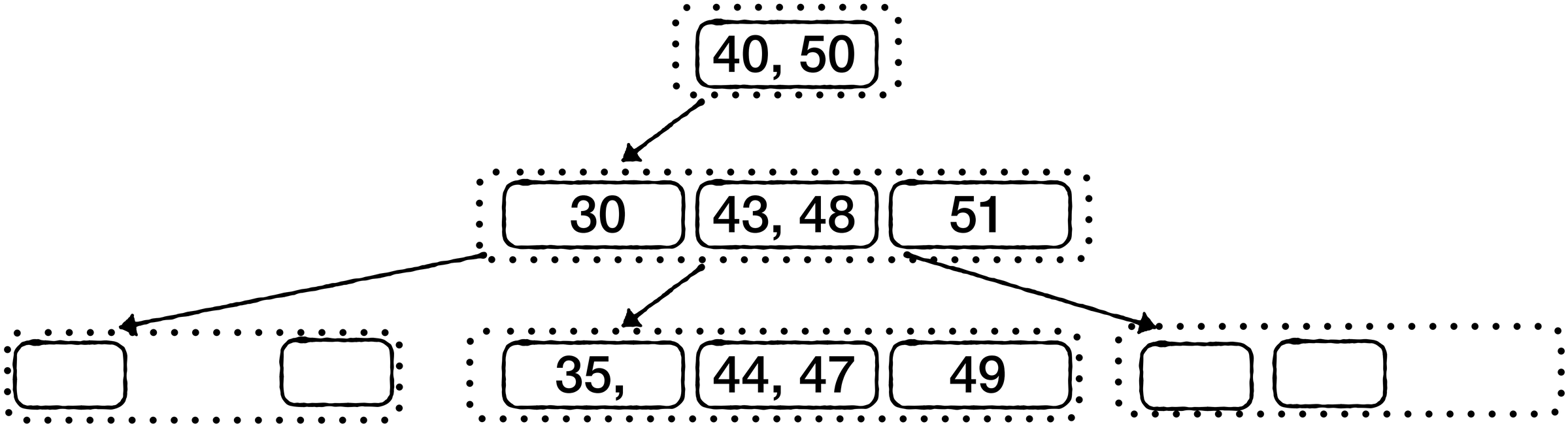
Write cost?



Every B/2 insertions, a leaf splits, causing us to rewrite a node group

Which costs O(B) cache misses

Write cost?

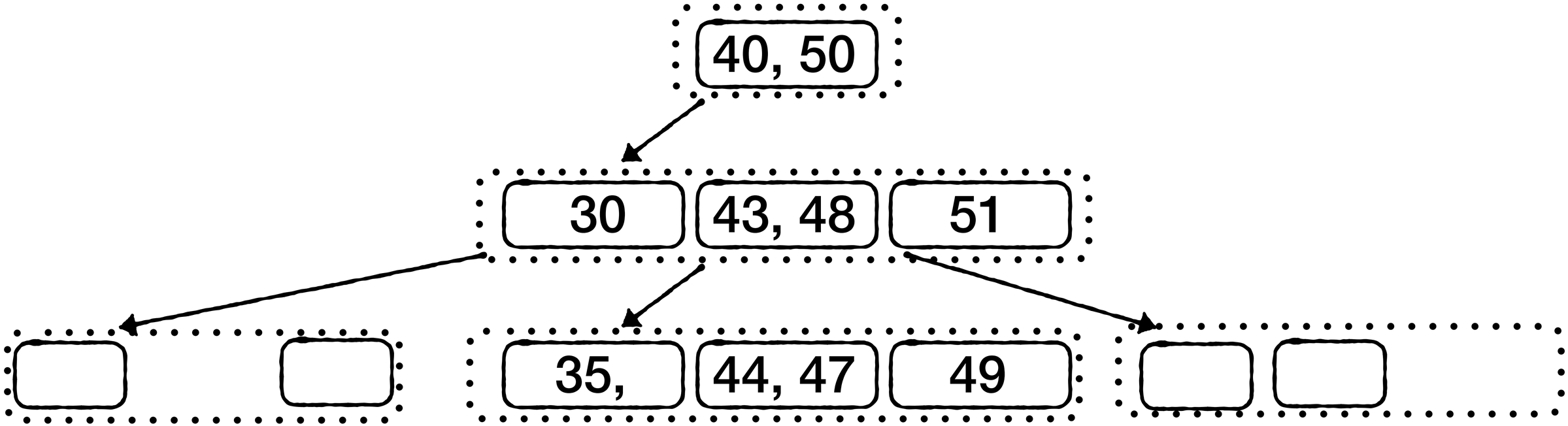


Every B/2 insertions, a leaf splits, causing us to rewrite a node group

Which costs O(B) cache misses

$O(\log_{B/2}(N) + B / (B/2))$

Write cost?

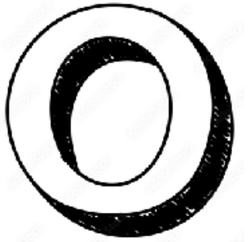


Every B/2 insertions, a leaf splits, causing us to rewrite a node group

Which costs O(B) cache misses

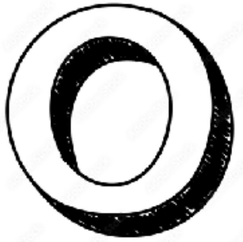
$O(\log_{B/2}(N))$

Better Worst-
Case



AVL-Tree	**B-Tree**	**T-Tree**	**CSS-Tree**	**CSB-Tree**	**HOT**
1962	1970	1985	1998	2000	2018

Better Worst-Case



AVL-Tree 1962

B-Tree 1970

T-Tree 1985

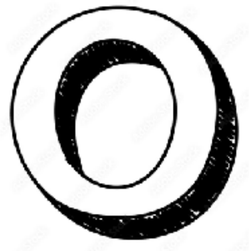
CSS-Tree 1998

CSB-Tree 2000

HOT 2018

Pres after

Better Worst-Case



AVL-Tree
1962

B-Tree
1970

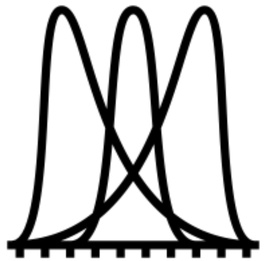
T-Tree
1985

CSS-Tree
1998

CSB-Tree
2000

HOT
2018

Exploiting Data Distribution



Interpolation search
1959

FITing-Tree
2019


Universe U

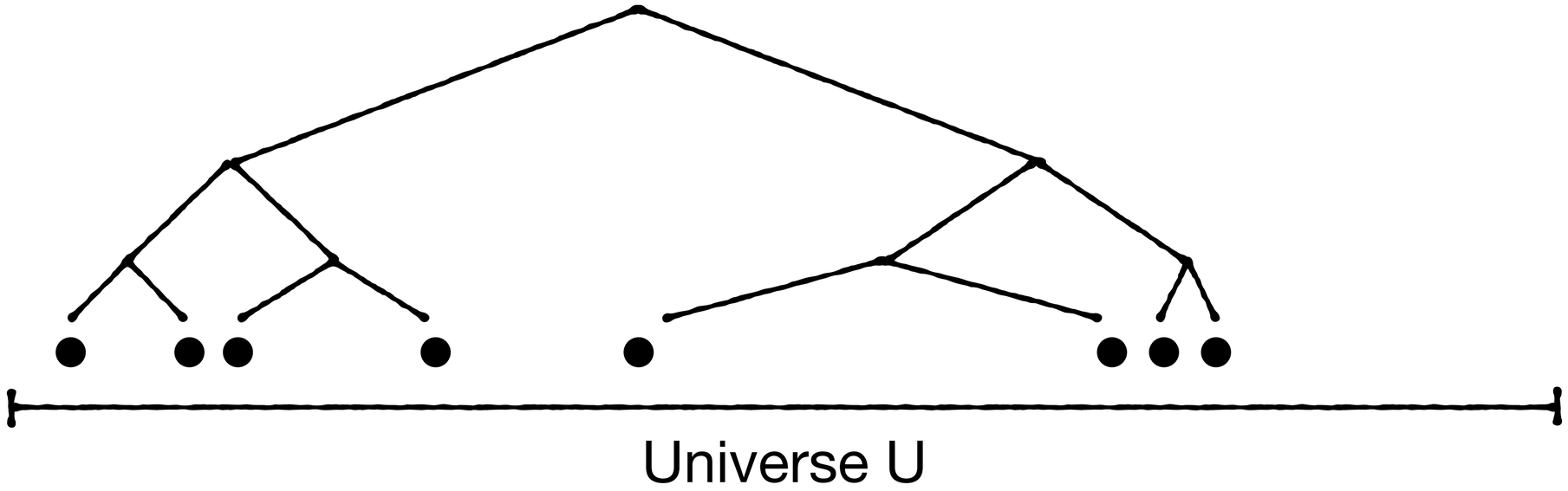
Binary search

$O(\log_2 N)$

Tree search

$O(\log_2 N)$ cycles

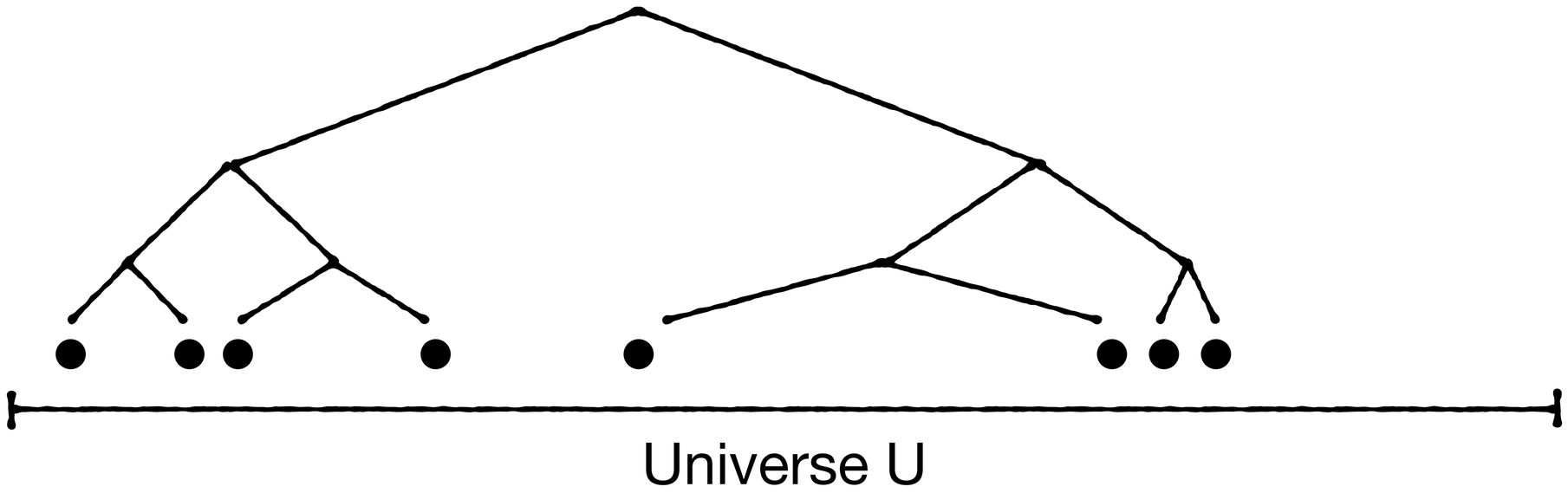
$O(\log_B N)$ I/O



These methods' performance is independent of data distribution

Tree search

Binary search



Can we exploit the data distribution to speed up search?

Can we exploit the data distribution to speed up search?

Fixed intervals

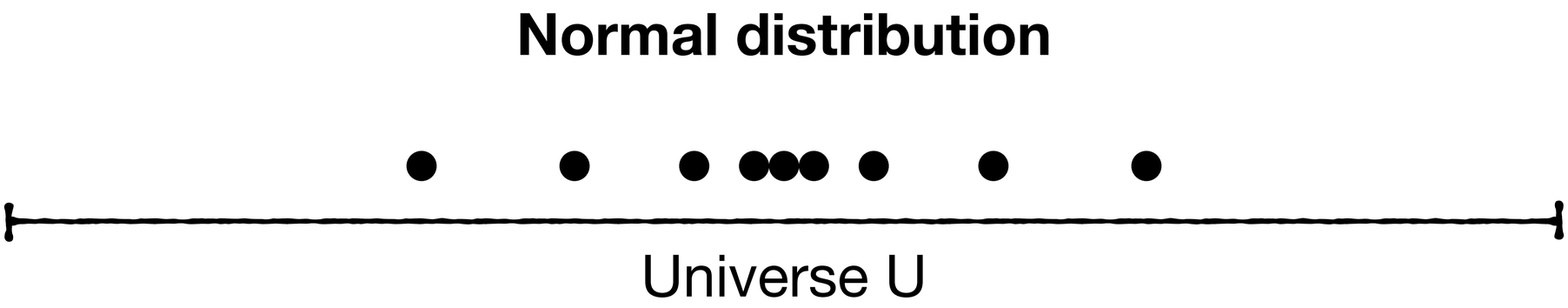
Universe U

Can we exploit the data distribution to speed up search?

Uniform distribution

Universe U

Can we exploit the data distribution to speed up search?



Can we exploit the data distribution to speed up search?

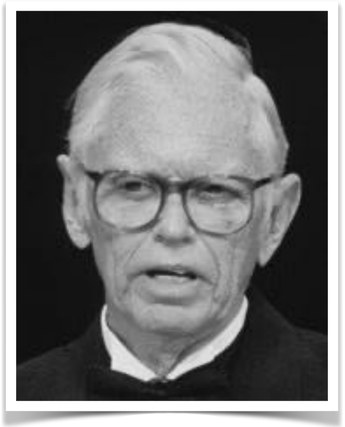
Bi-modal distribution

Universe U

Interpolation Search

Addressing for Random-Access Storage

IBM Journal of Research and Development. 1959



W. Wesley Peterson

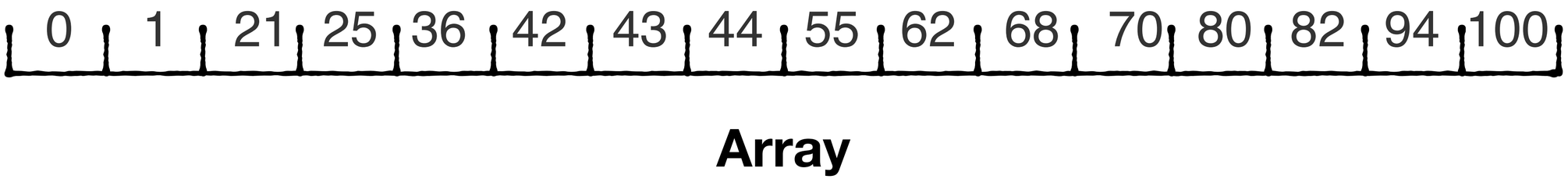
Interpolation Search

Uniform distribution

Interpolation Search

Uniform distribution - **e.g., sorted array of hash values**

Interpolation Search



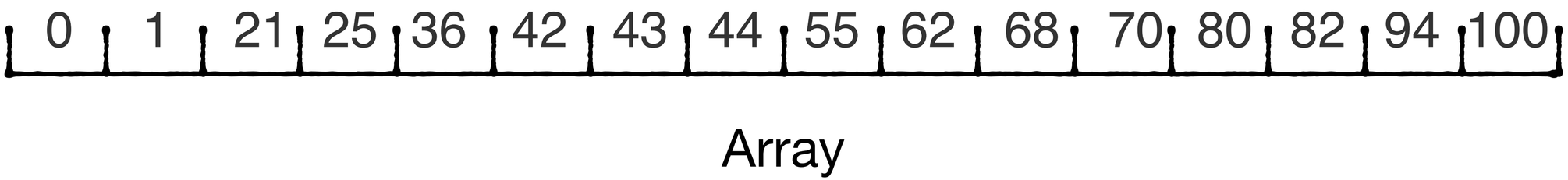
Interpolation Search

get(X)

Array

get(X)

$$\text{Estimated location} = \left\lfloor \frac{X - \min}{\max - \min} \cdot (\#\text{slots} - 1) \right\rfloor$$

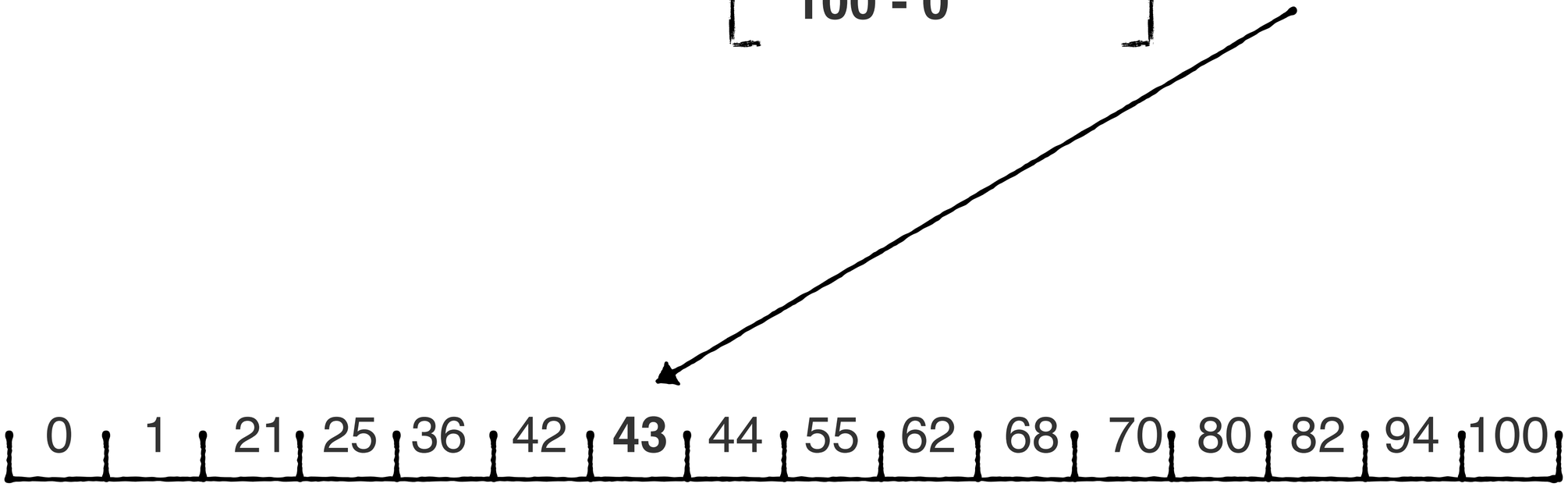


get(42)

$$\text{Estimated location} = \left\lfloor \frac{\mathbf{42 - 0}}{\mathbf{100 - 0}} \cdot \mathbf{15} \right\rfloor =$$

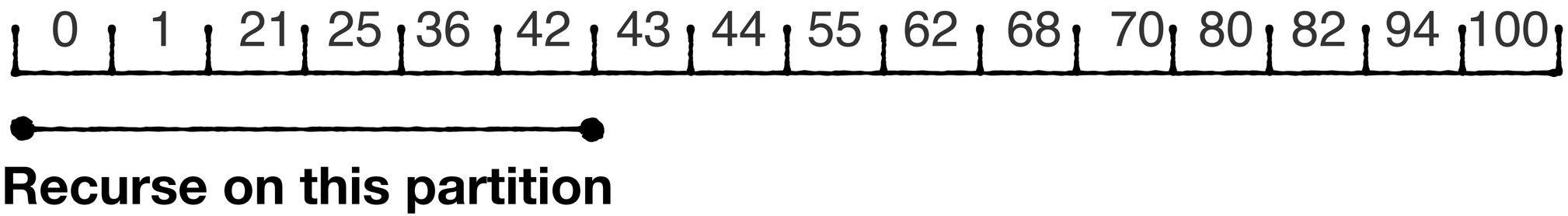
get(42)

$$\text{Estimated location} = \left\lfloor \frac{42 - 0}{100 - 0} \cdot 15 \right\rfloor = 6$$



get(42)

$$\text{Estimated location} = \left\lfloor \frac{42 - 0}{100 - 0} \cdot 15 \right\rfloor = 6$$



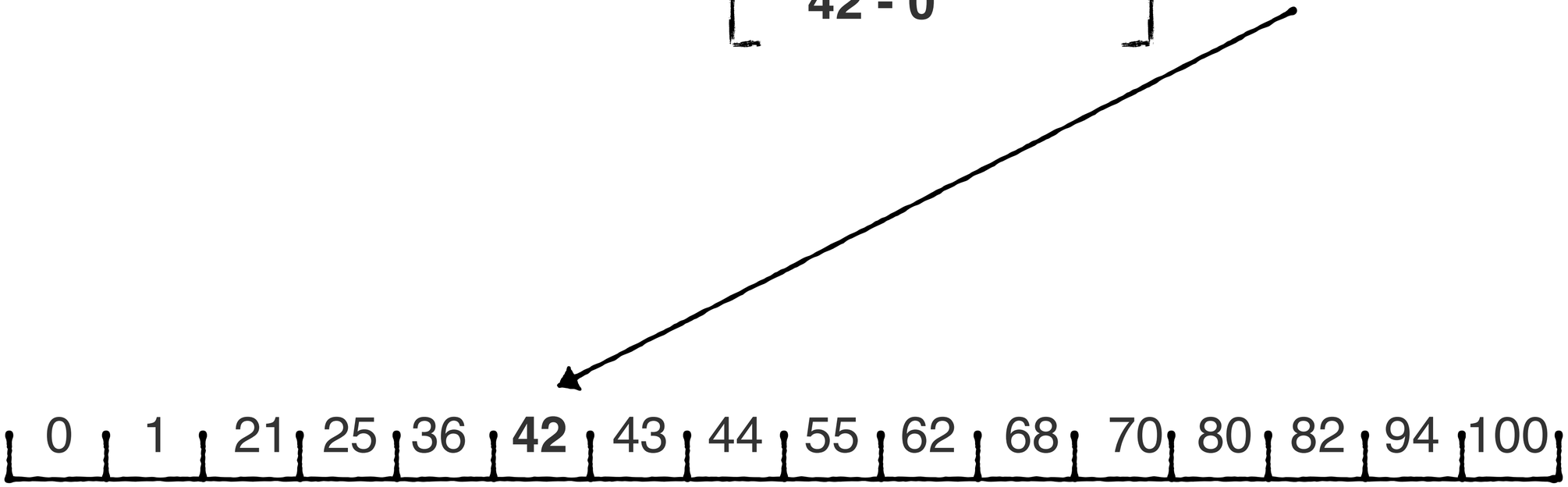
get(42)

$$\text{Estimated location} = \left\lfloor \frac{42 - 0}{42 - 0} \cdot 5 \right\rfloor$$

Recurse on this partition

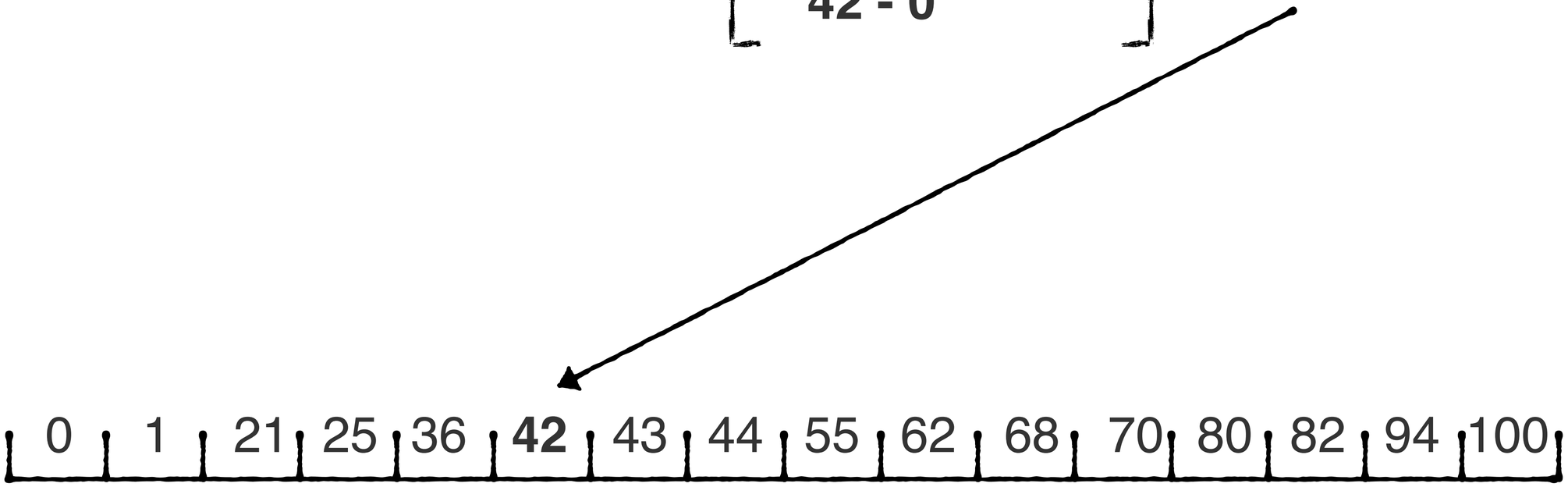
get(42)

$$\text{Estimated location} = \left\lfloor \frac{42 - 0}{42 - 0} \cdot 5 \right\rfloor = 5$$



get(42)

$$\text{Estimated location} = \left\lfloor \frac{42 - 0}{42 - 0} \cdot 5 \right\rfloor = 5$$



Found in two steps! **As opposed to $\log_2(16) = 4$ steps with binary search**

For uniformly distributed data:

Interpolation search $O(\log_2 \log_2 N)$

Binary search $O(\log_2 N)$

Interpolation search $O(\log_2 \log_2 N)$

Intuition: each iteration prunes the search space by $\sqrt{N}$

Interpolation search $O(\log_2 \log_2 N)$

Intuition: each iteration prunes the search space by $\sqrt{N}$

$$\mathbf{T(n) < C + T(\sqrt{N})}$$

Interpolation search $O(\log_2 \log_2 N)$

Intuition: each iteration prunes the search space by $\sqrt{N}$

$$T(n) < C \cdot \mathbf{\log_2 \log_2 N}$$

Interpolation search $O(\log_2 \log_2 N)$

Intuition: each iteration prunes the search space by $\sqrt{N}$

$$T(n) < \mathbf{C} \cdot \log_2 \log_2 N$$

$$\uparrow$$

$$\approx \mathbf{2}$$

Interpolation search $O(\log_2 \log_2 N)$

Intuition: each iteration prunes the search space by $\sqrt{N}$

$$T(n) < \mathbf{C} \cdot \log_2 \log_2 N$$

$$\uparrow$$

$$\approx \mathbf{2}$$

For details, check our reading for this week.

Now suppose the data is geometrically increasing

Now suppose the data is geometrically increasing

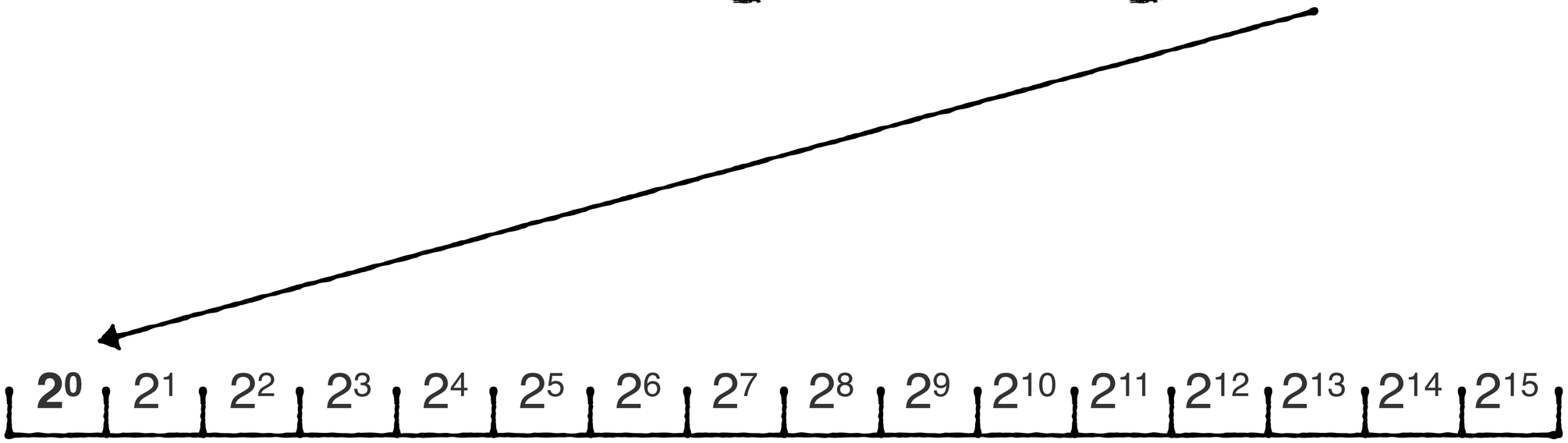
$$\textbf{Estimated location} = \left\lfloor \frac{\textbf{X - min}}{\textbf{max - min}} \cdot \textbf{(\#slots - 1)} \right\rfloor$$

get(2^{11})

$$\text{Estimated location} = \left\lfloor \frac{\text{X - min}}{\text{max - min}} \cdot (\#\text{slots - 1}) \right\rfloor$$

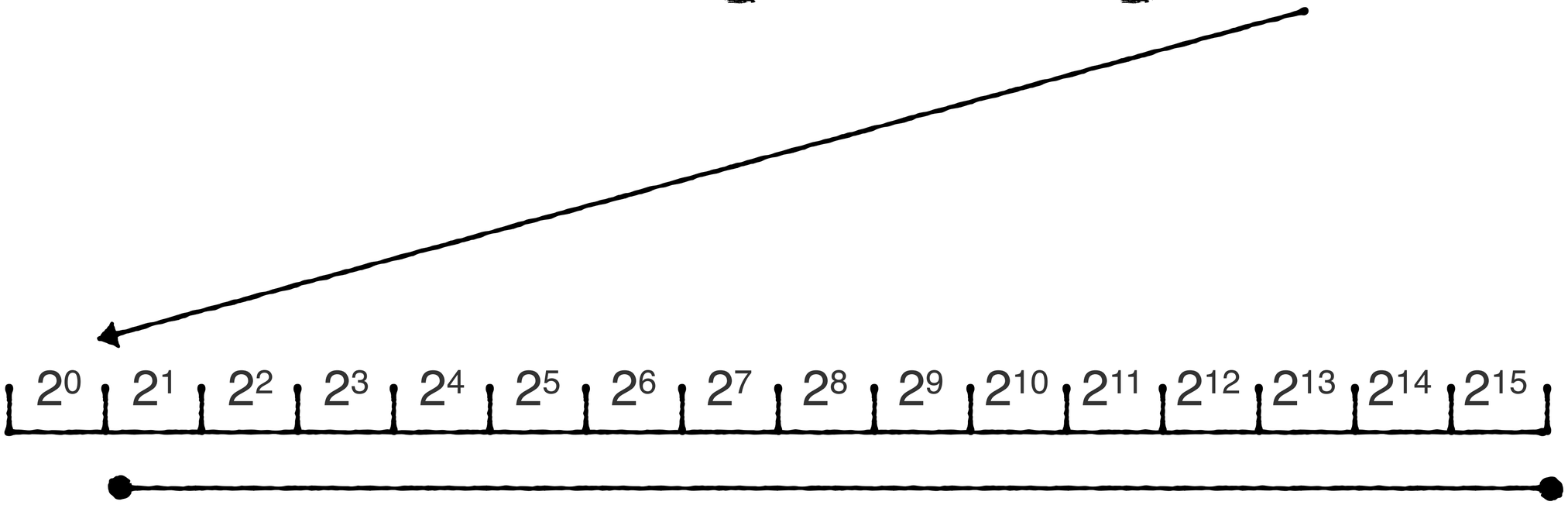
get(2^{11})

$$\text{Estimated location} = \left\lfloor \frac{2^{11} - 2^{0}}{2^{15} - 2^{0}} \cdot 15 \right\rfloor = 0$$



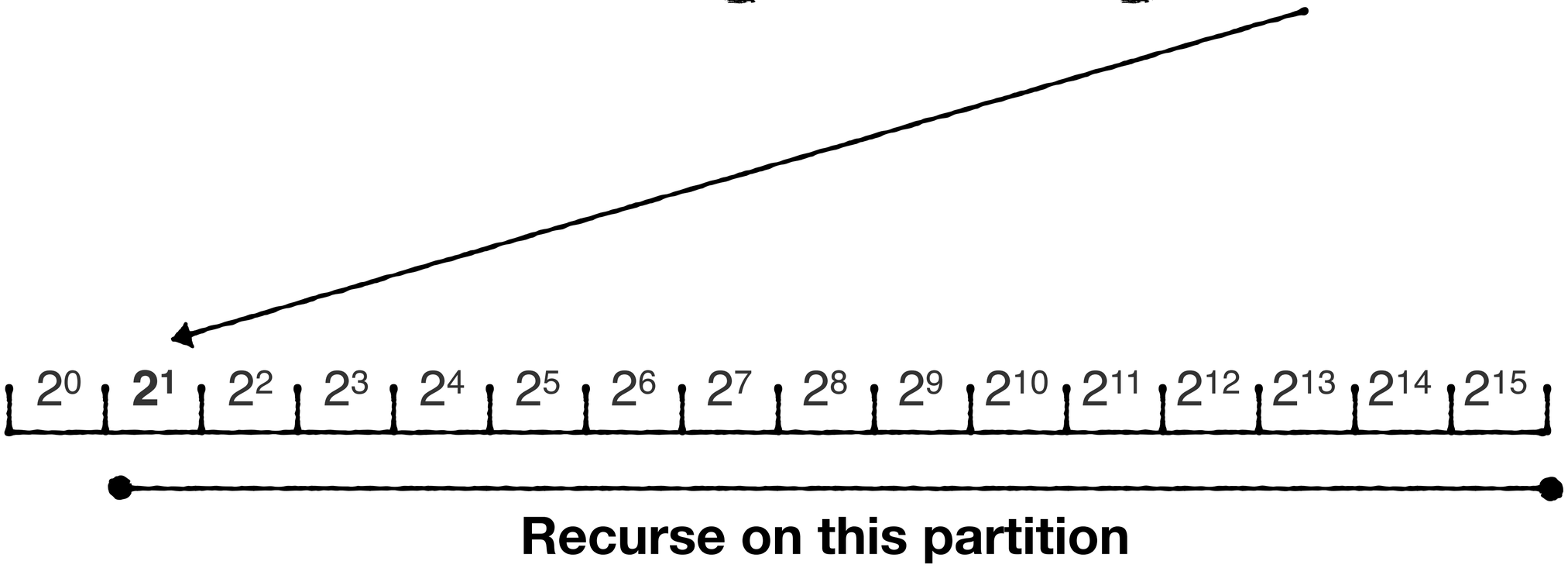
get(2^{11})

$$\text{Estimated location} = \left\lfloor \frac{2^{11} - 2^{0}}{2^{15} - 2^{0}} \cdot 15 \right\rfloor = 0$$



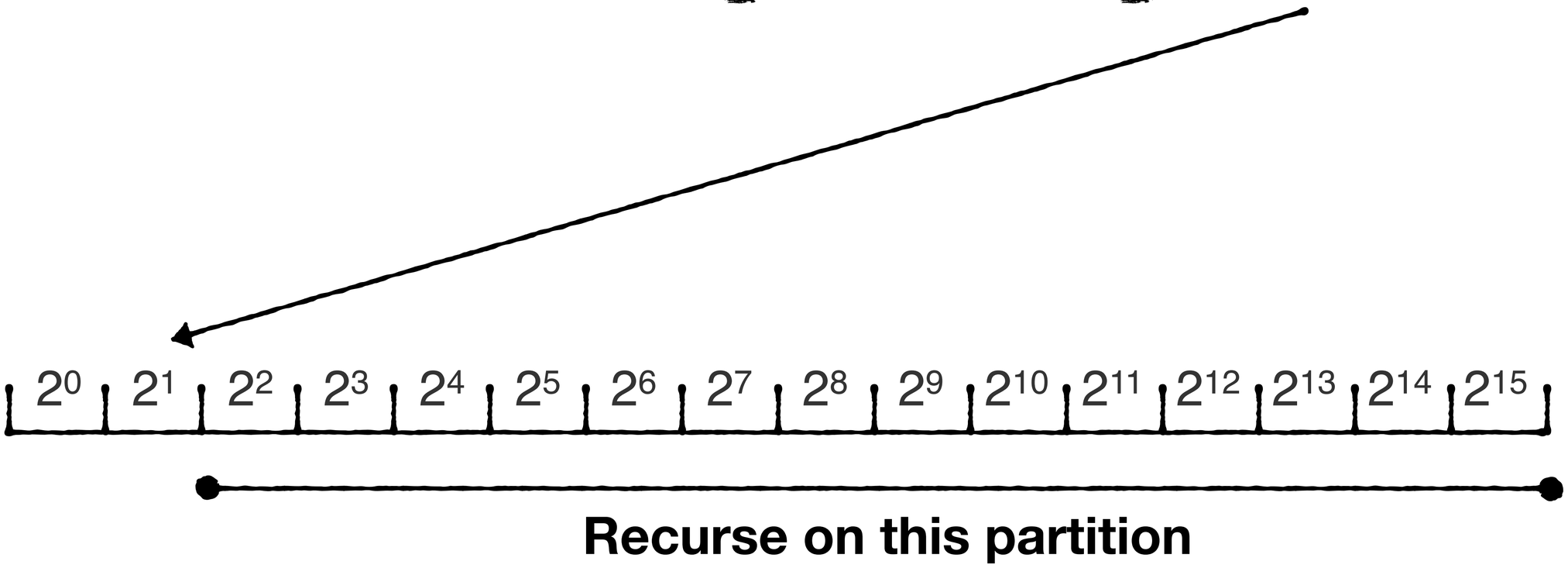
get(2^{11})

$$\text{Estimated location} = \left\lfloor \frac{2^{11} - 2^{1}}{2^{15} - 2^{1}} \cdot 14 \right\rfloor = 0$$



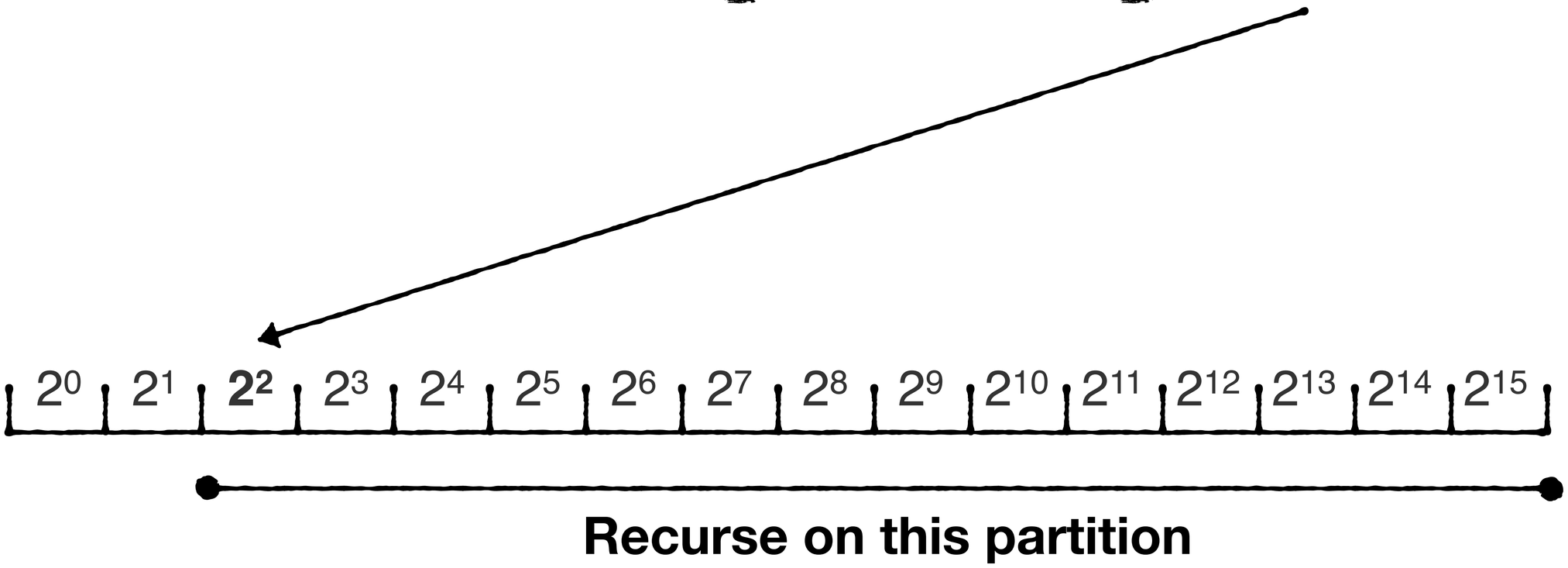
get(2^{11})

$$\text{Estimated location} = \left\lfloor \frac{2^{11} - 2^{1}}{2^{15} - 2^{1}} \cdot 14 \right\rfloor = 0$$



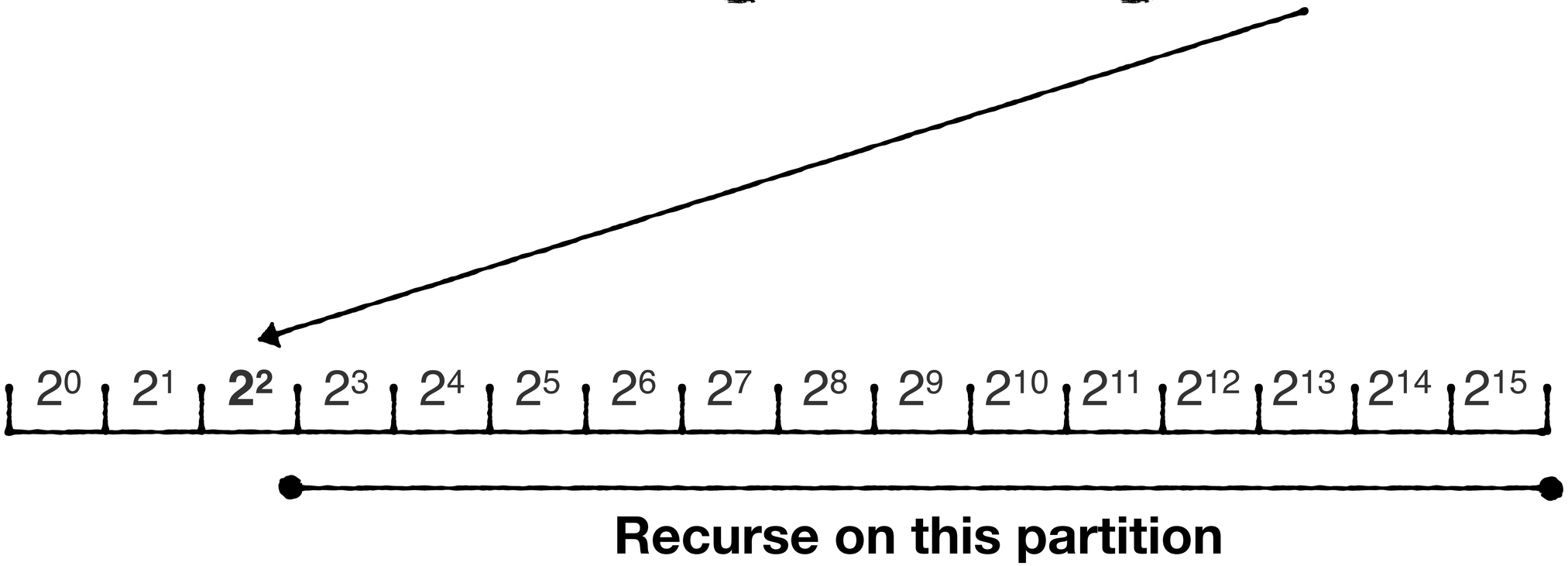
get(2^{11})

$$\text{Estimated location} = \left\lfloor \frac{2^{11} - 2^2}{2^{15} - 2^2} \cdot 13 \right\rfloor = 0$$



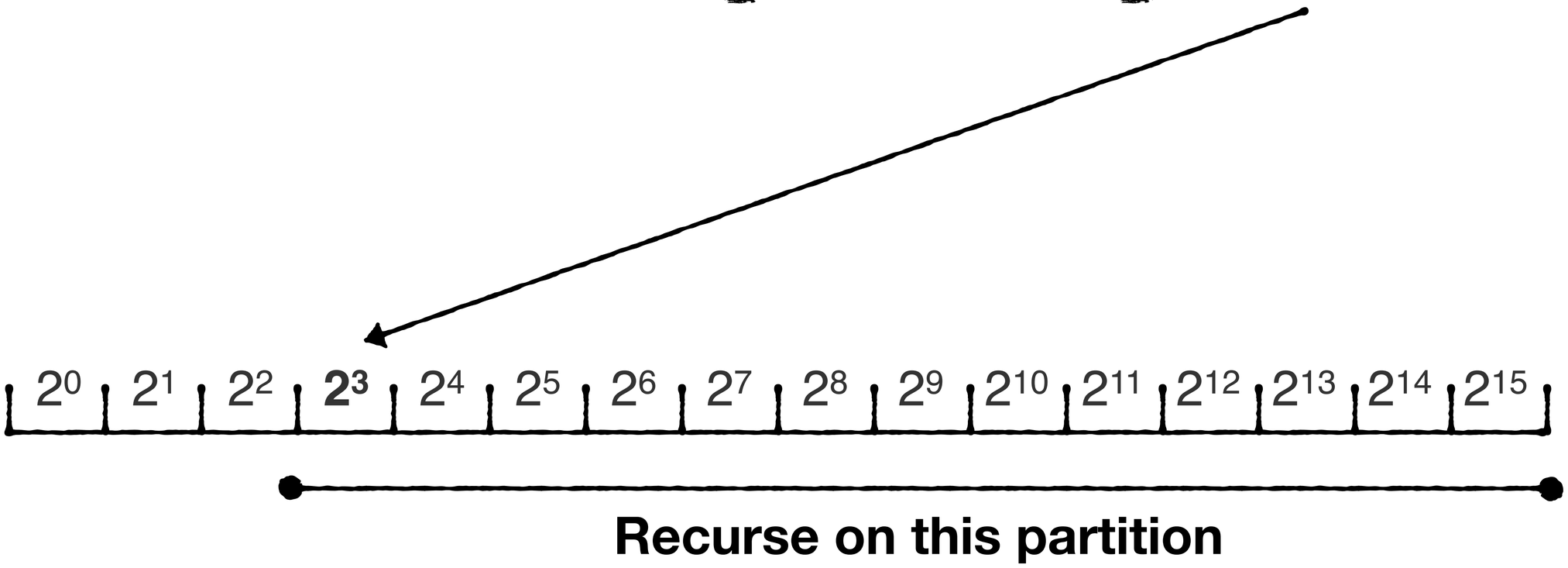
get(2^{11})

$$\text{Estimated location} = \left\lfloor \frac{2^{11} - 2^2}{2^{15} - 2^2} \cdot 13 \right\rfloor = 0$$



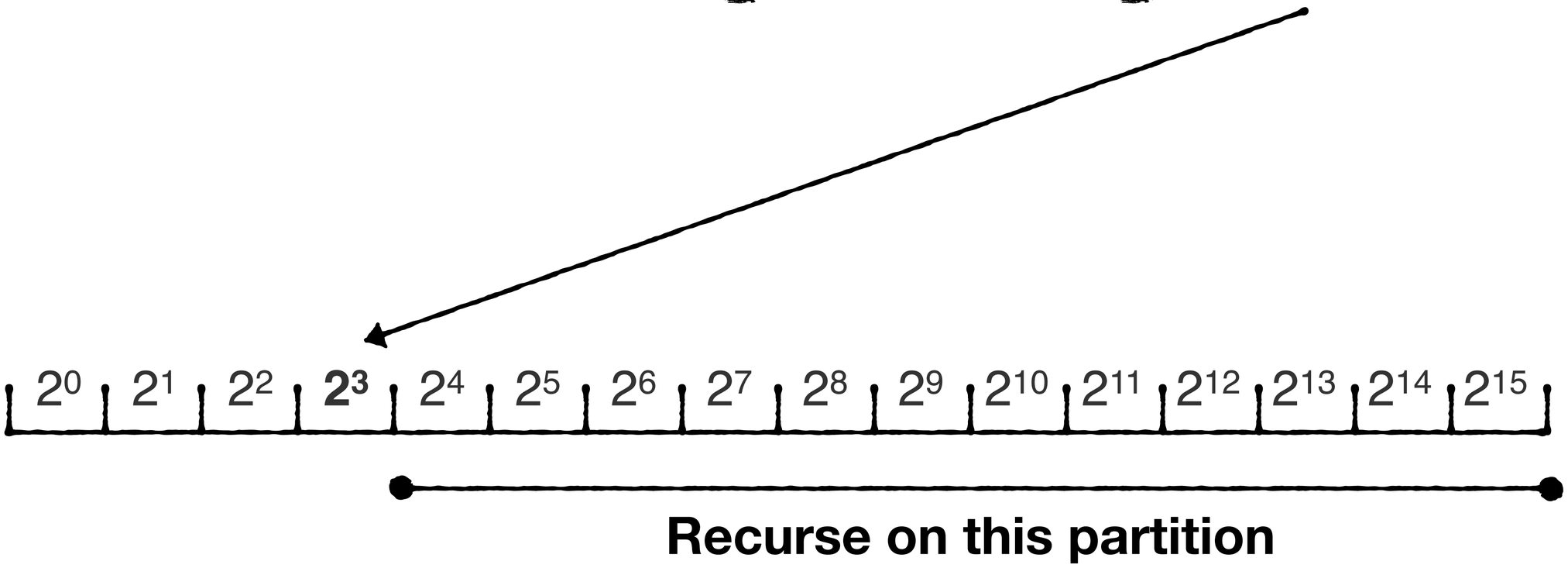
get(2^{11})

$$\text{Estimated location} = \left\lfloor \frac{2^{11} - 2^{3}}{2^{15} - 2^{3}} \cdot 12 \right\rfloor = 0$$



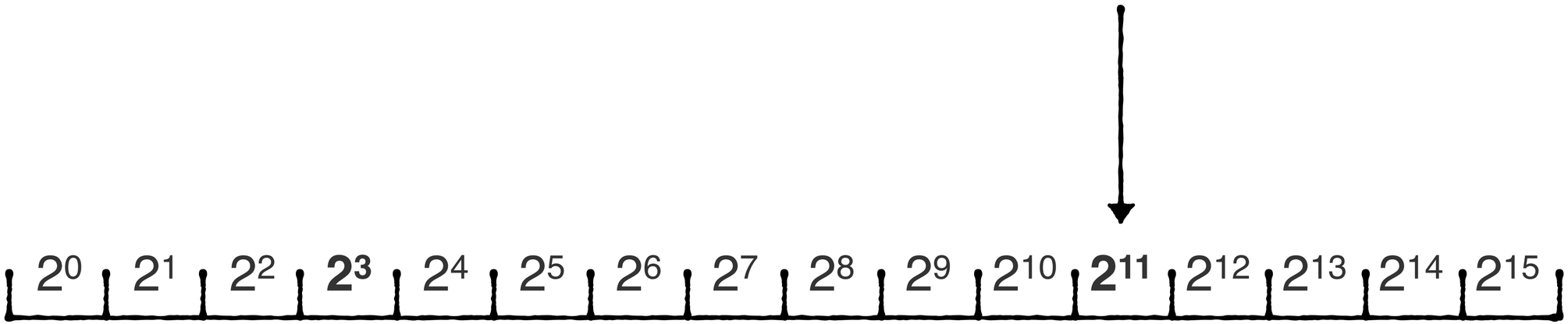
get(2^{11})

$$\text{Estimated location} = \left\lfloor \frac{2^{11} - 2^{3}}{2^{15} - 2^{3}} \cdot 12 \right\rfloor = 0$$



get(2^{11})

Find target after O(N) iterations



Interpolation search

Uniform

$O(\log_2 \log_2 N)$

Worst-case

$O(N)$

CPU overheads?

CPU overheads

$$\textbf{Estimated location} = \left\lfloor \frac{\textbf{X - min}}{\textbf{max - min}} \cdot \textbf{(\#slots - 1)} \right\rfloor$$

CPU overheads

$$\text{Estimated location} = \left\lfloor \frac{X - \min}{\max - \min} \cdot (\#\text{slots} - 1) \right\rfloor$$

3 subtractions
1 multiplication
1 division

CPU overheads

$$\text{Estimated location} = \left\lfloor \frac{X - \min}{\max - \min} \cdot (\#\text{slots} - 1) \right\rfloor$$

3 subtractions	**≈ 6 cycles each**
1 multiplication	**≈ 6 cycles**
1 division	**≈ 30 - 60 cycles**

CPU overheads

$$\text{Estimated location} = \left\lfloor \frac{X - \min}{\max - \min} \cdot (\#\text{slots} - 1) \right\rfloor$$

3 subtractions	≈ 6 cycles each
1 multiplication	≈ 6 cycles
1 division	≈ 30 - 60 cycles

For small data, binary search may win as there is no division.

CPU overheads

$$\text{Estimated location} = \left\lfloor \frac{X - \min}{\max - \min} \cdot (\#\text{slots} - 1) \right\rfloor$$

3 subtractions	≈ 6 cycles each
1 multiplication	≈ 6 cycles
1 division	≈ 30 - 60 cycles

For small data, binary search may win as there is no division.

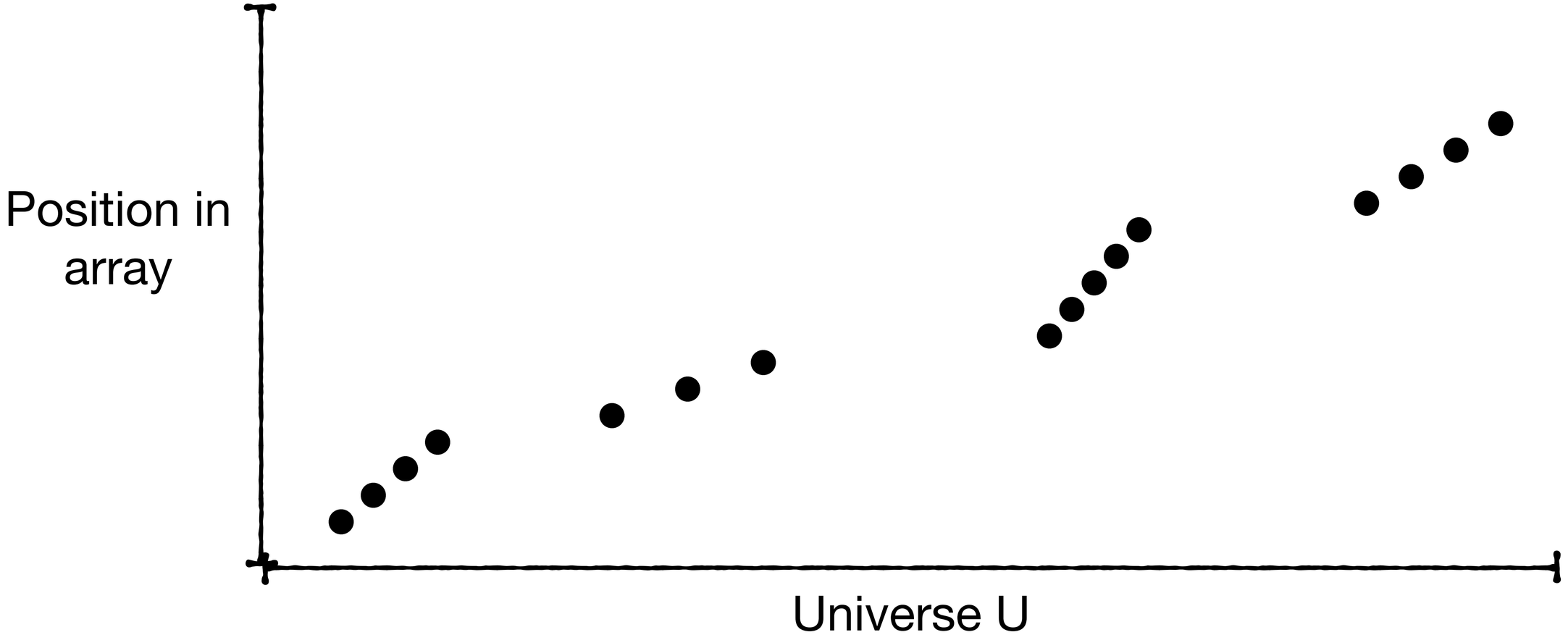
As the data grows, interpolation search is likely faster.

Interpolation search works well for uniform data

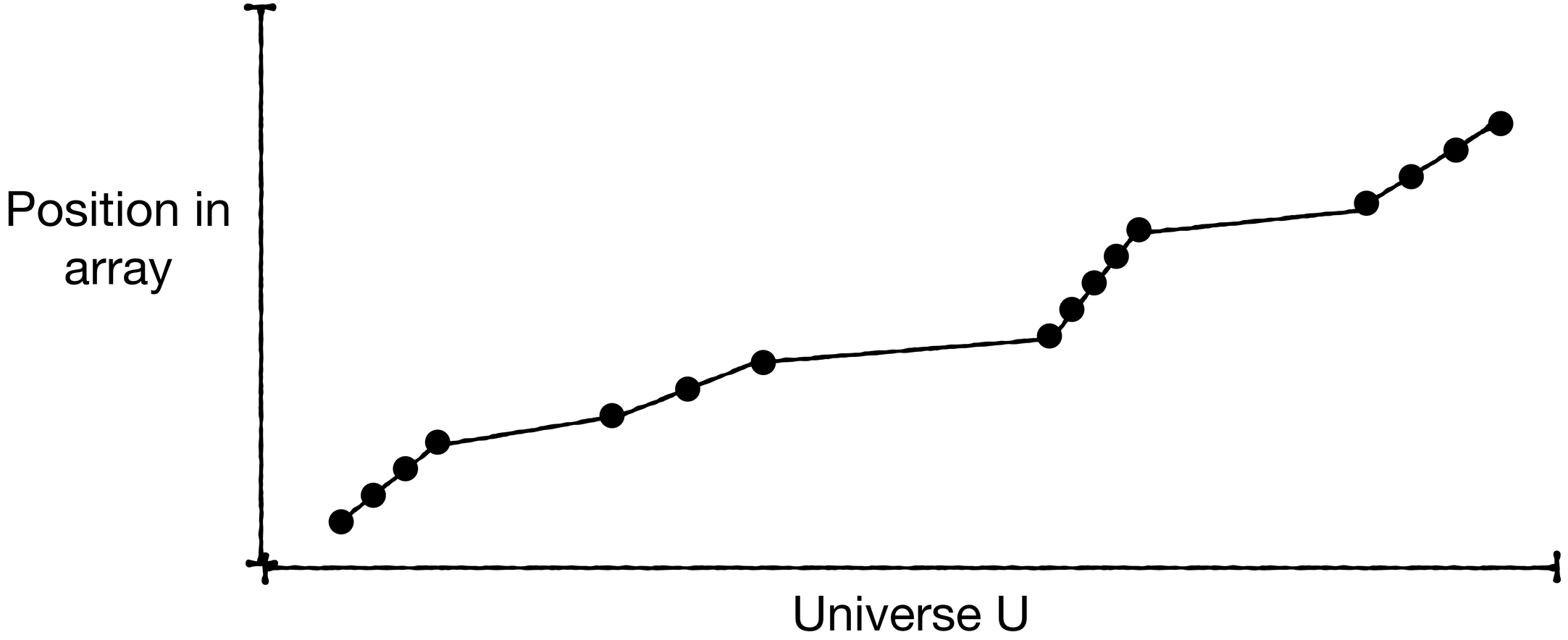
Interpolation search works well for uniform data

Insight: when data is predictable, we can tailor better access methods

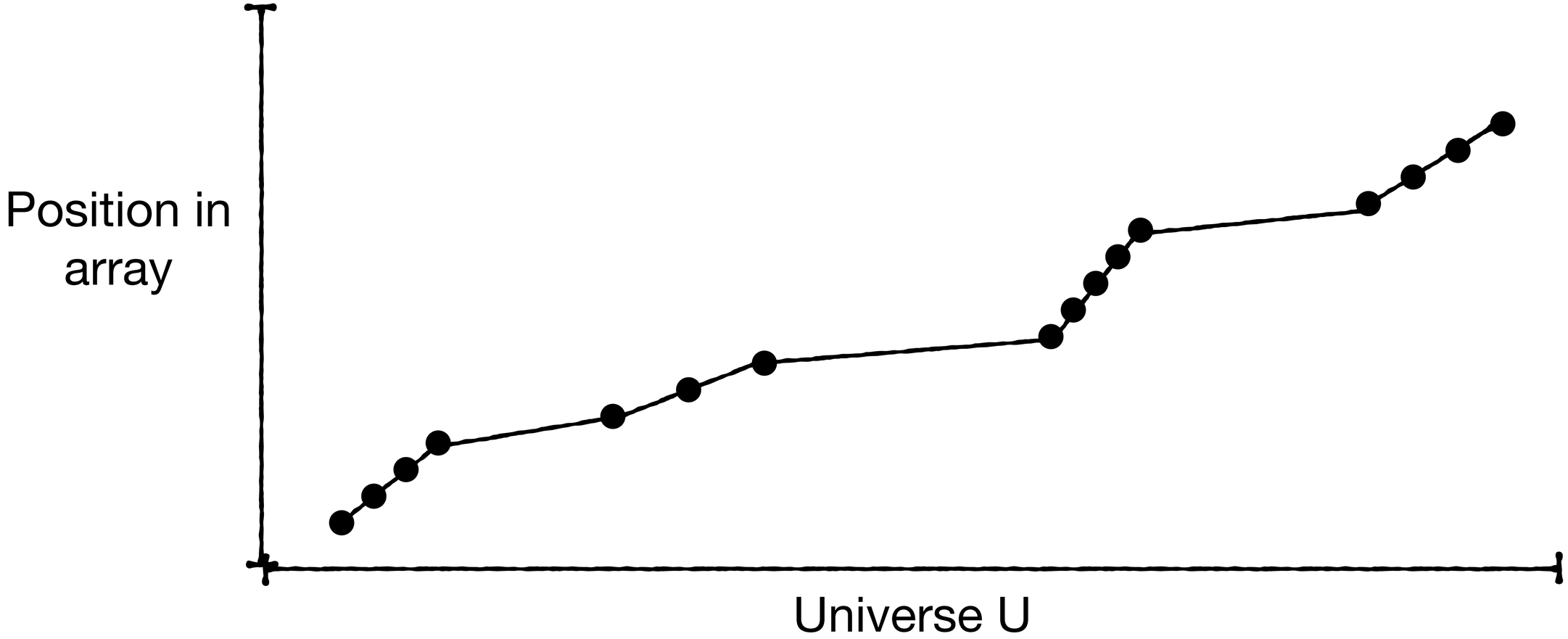
But data often follows idiosyncratic patterns


Position in array
Universe U

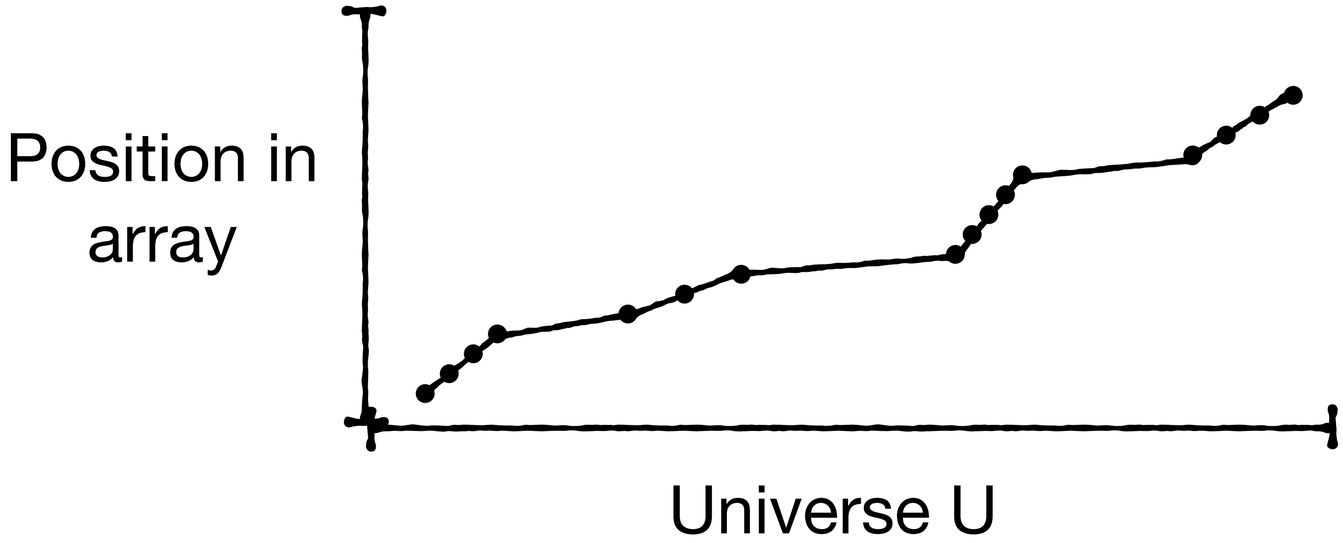
cumulative distribution function (CDF)

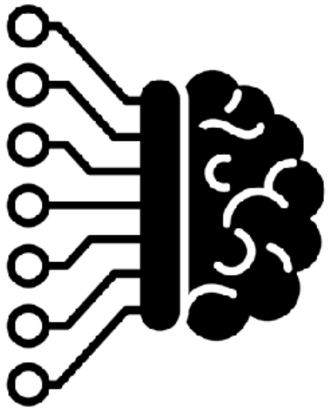


Insight: at each segment, it's easy to predict an entry's position

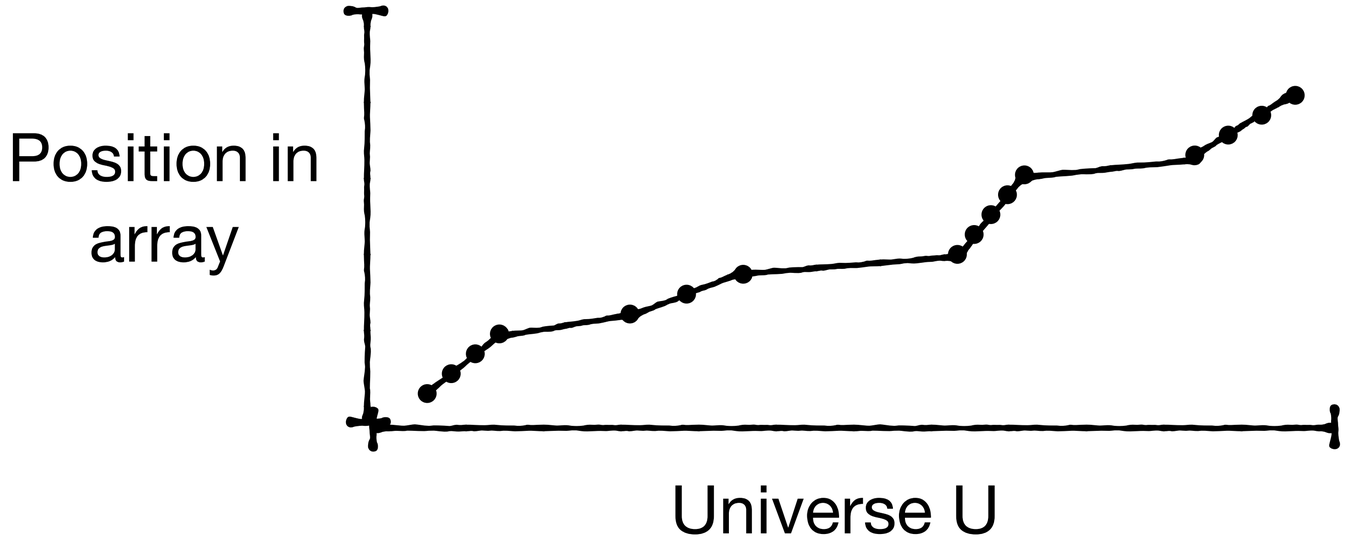


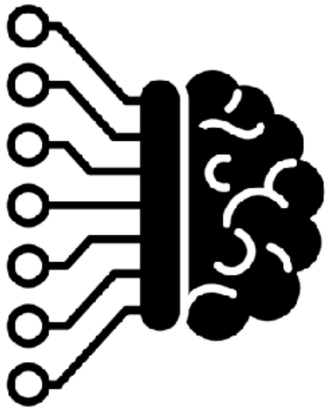
Learned Indexes: Learn the data distribution to predict an entry's position





Learned Indexes: Learn the data distribution to predict an entry's position



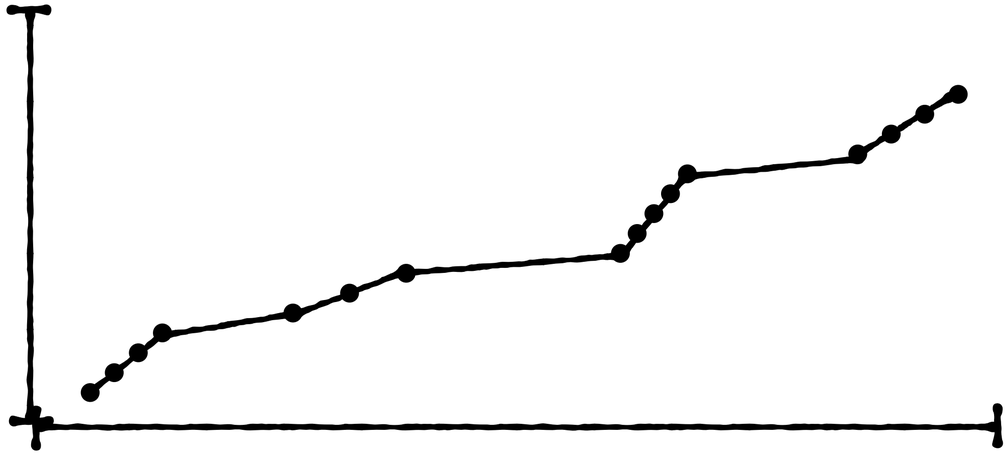


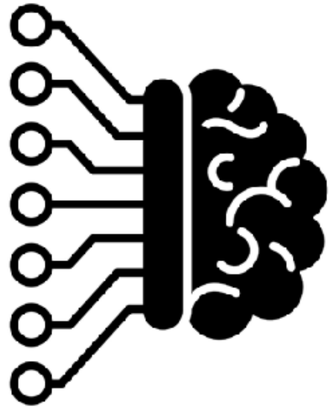
Partition the data into predictable segments

The Case for Learned Index Structures

Tim Kraska , Alex Beutel , Ed H. Chi , Jeffrey Dean , Neoklis Polyzotis

SIGMOD 2018

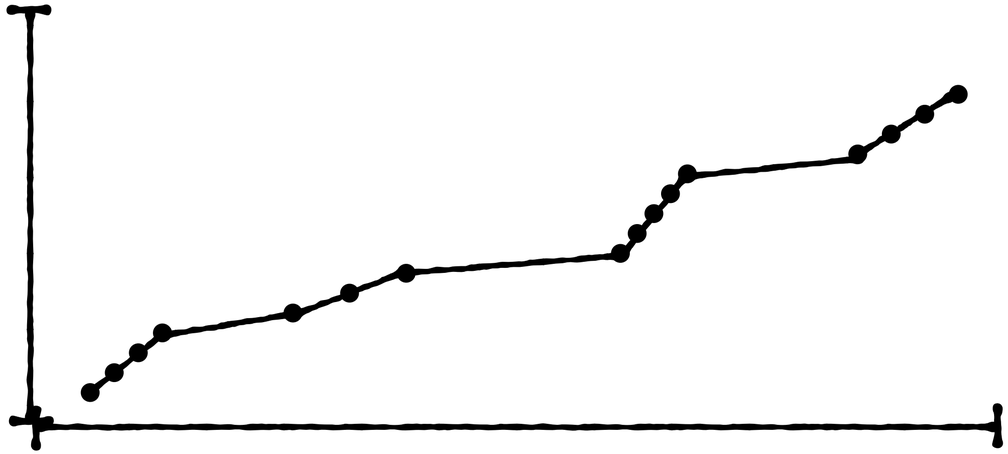


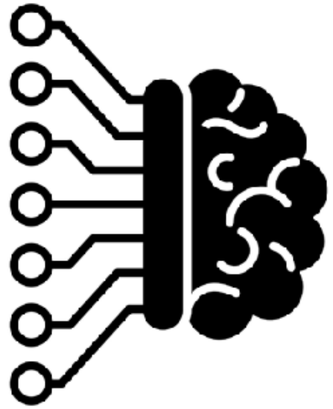


The Case for Learned Index Structures

Tim Kraska , Alex Beutel , Ed H. Chi , Jeffrey Dean , Neoklis Polyzotis

SIGMOD 2018



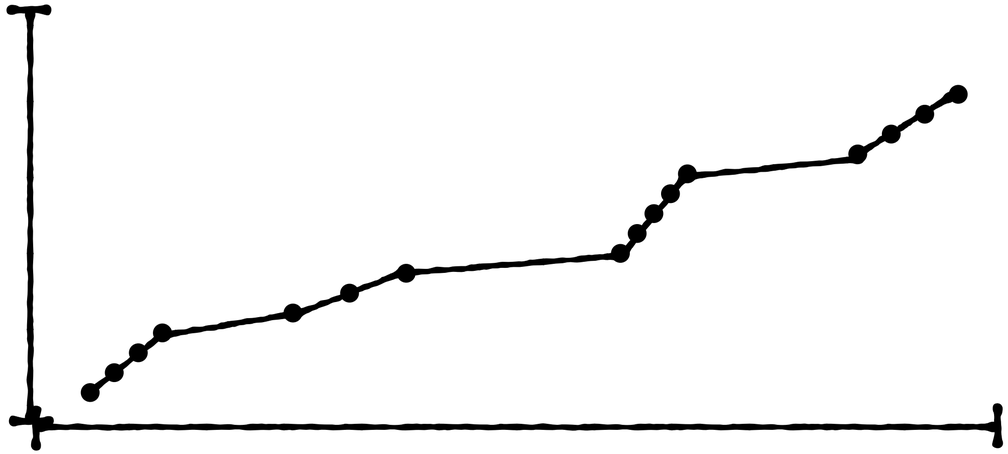


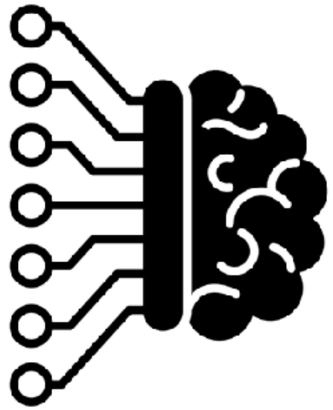
Vision paper, and algorithmic details can be vague

FITing-Tree: A Data-aware Index Structure

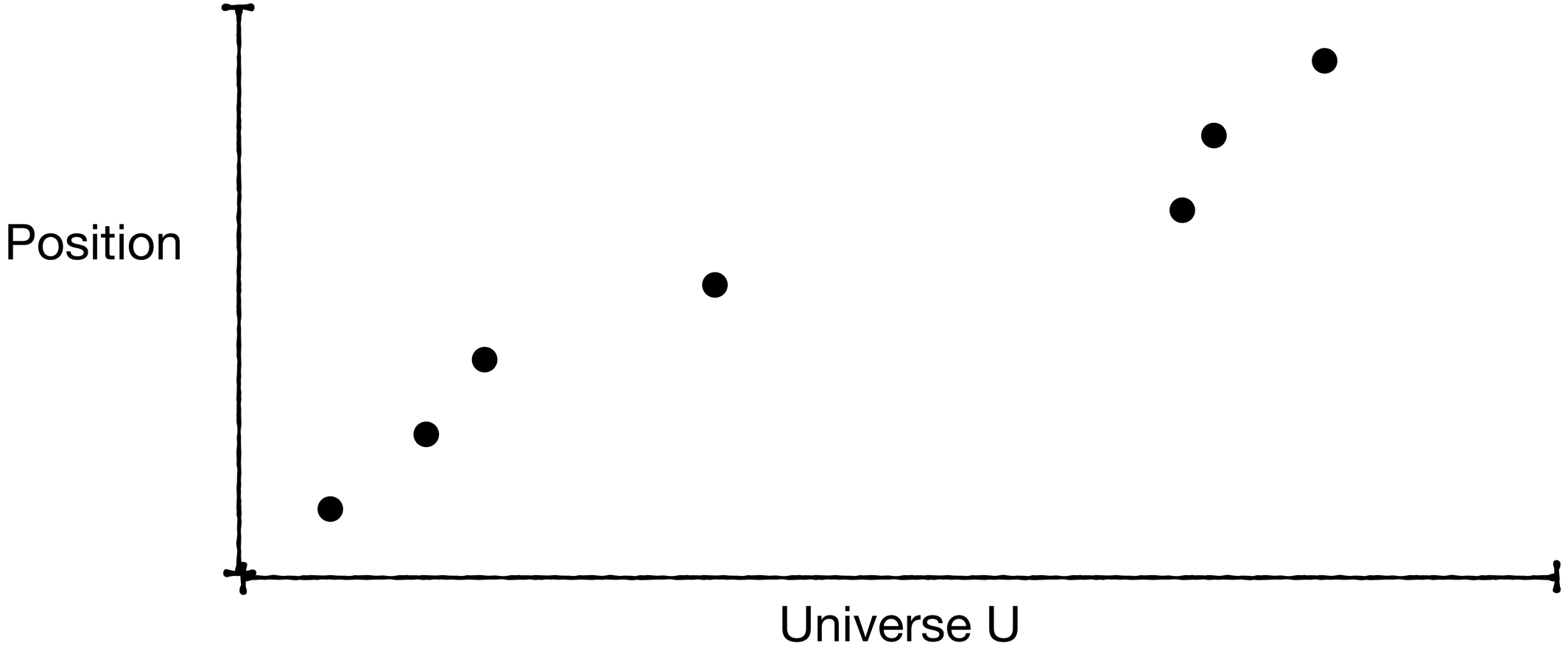
Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, Tim Kraska

SIGMOD 2019

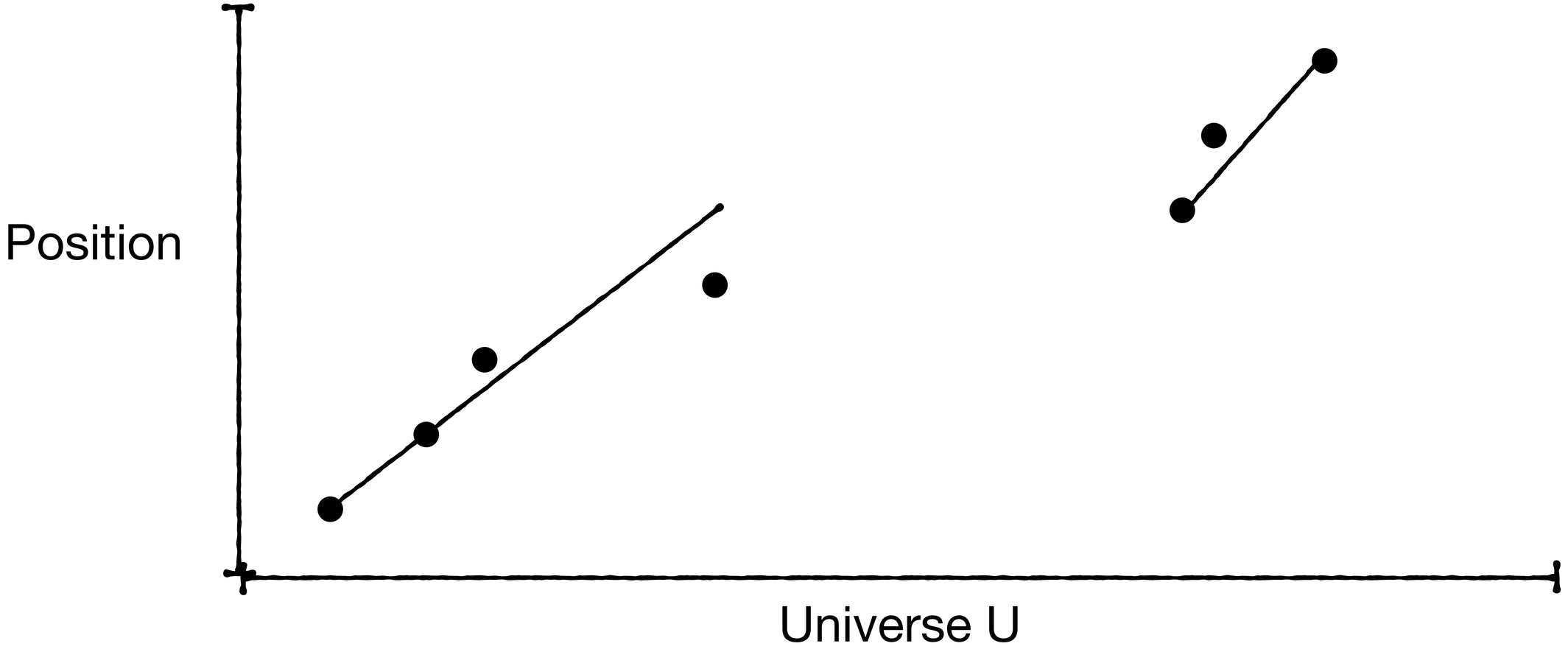




FITing-Tree: A Data-aware Index Structure

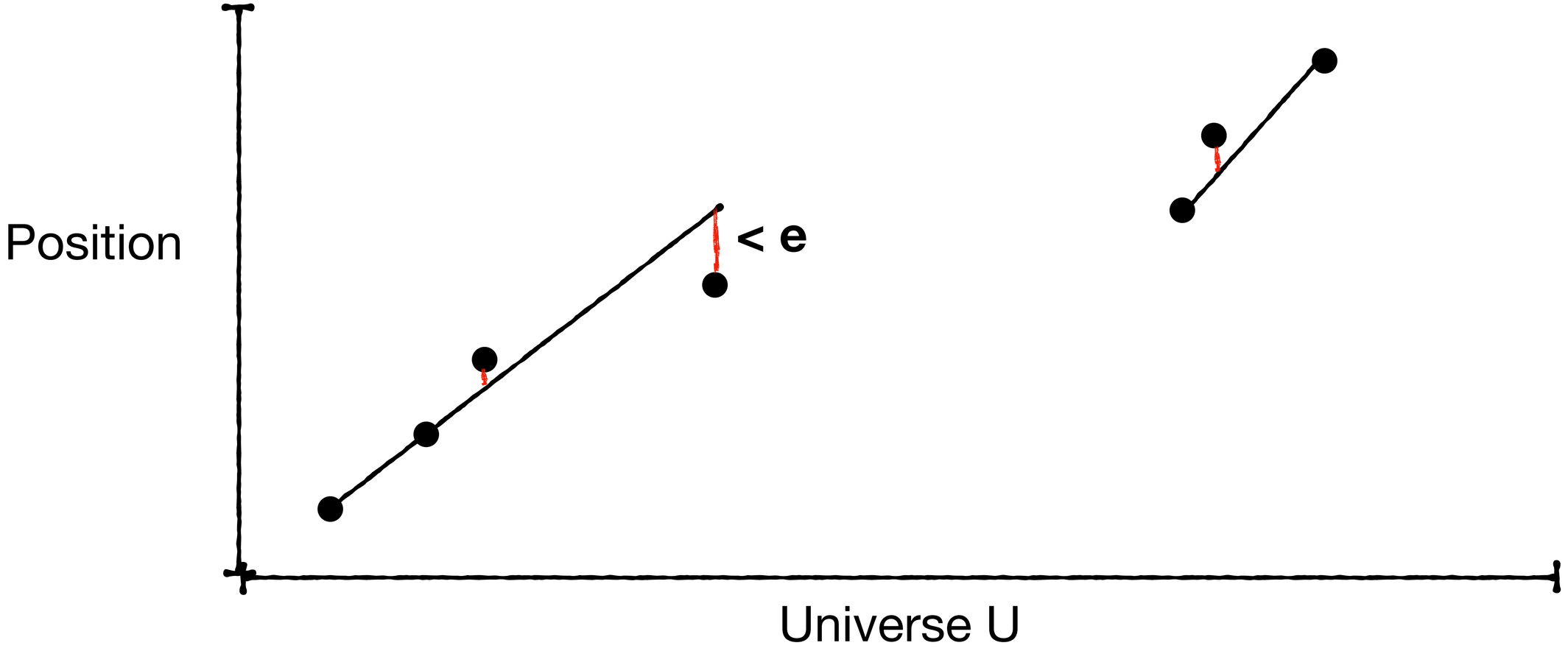


approximate distribution using piecewise linear functions



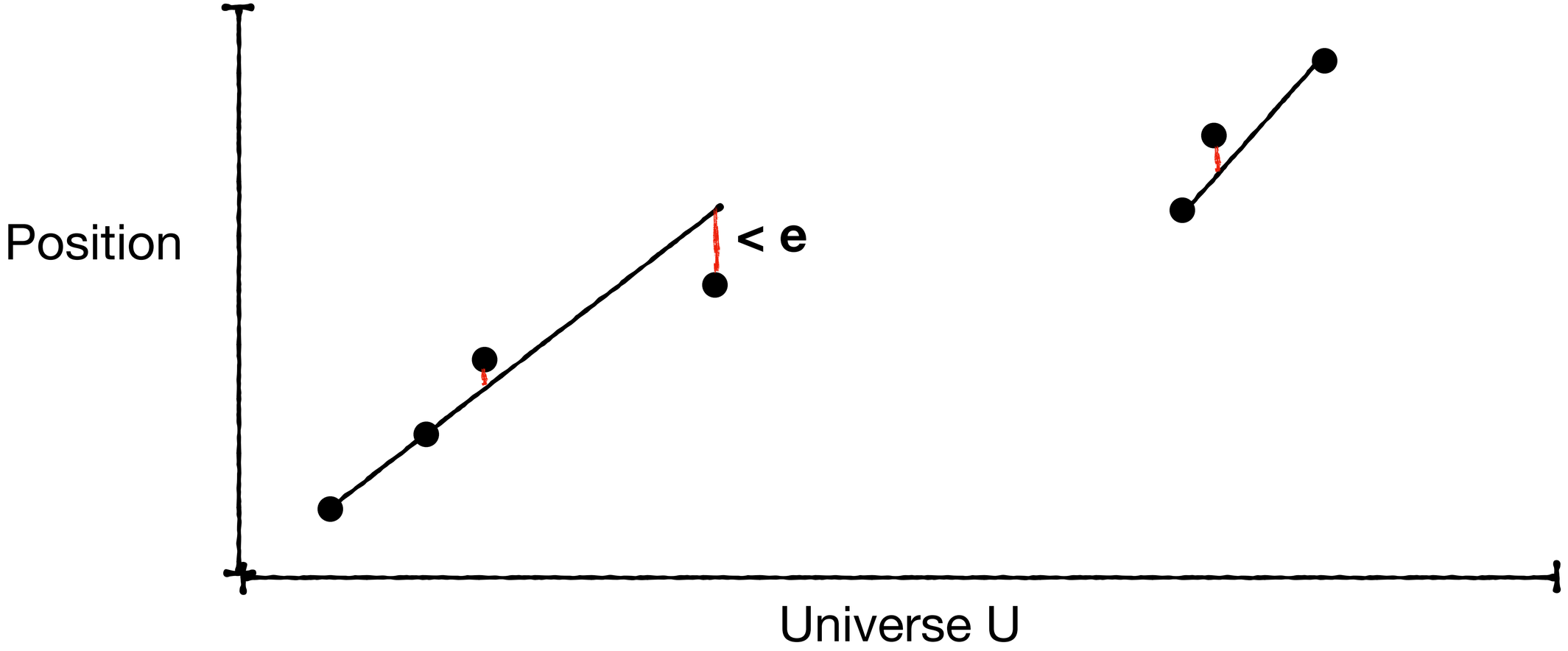
approximate distribution using piecewise linear functions

Ensure each point has max distance of e from its function

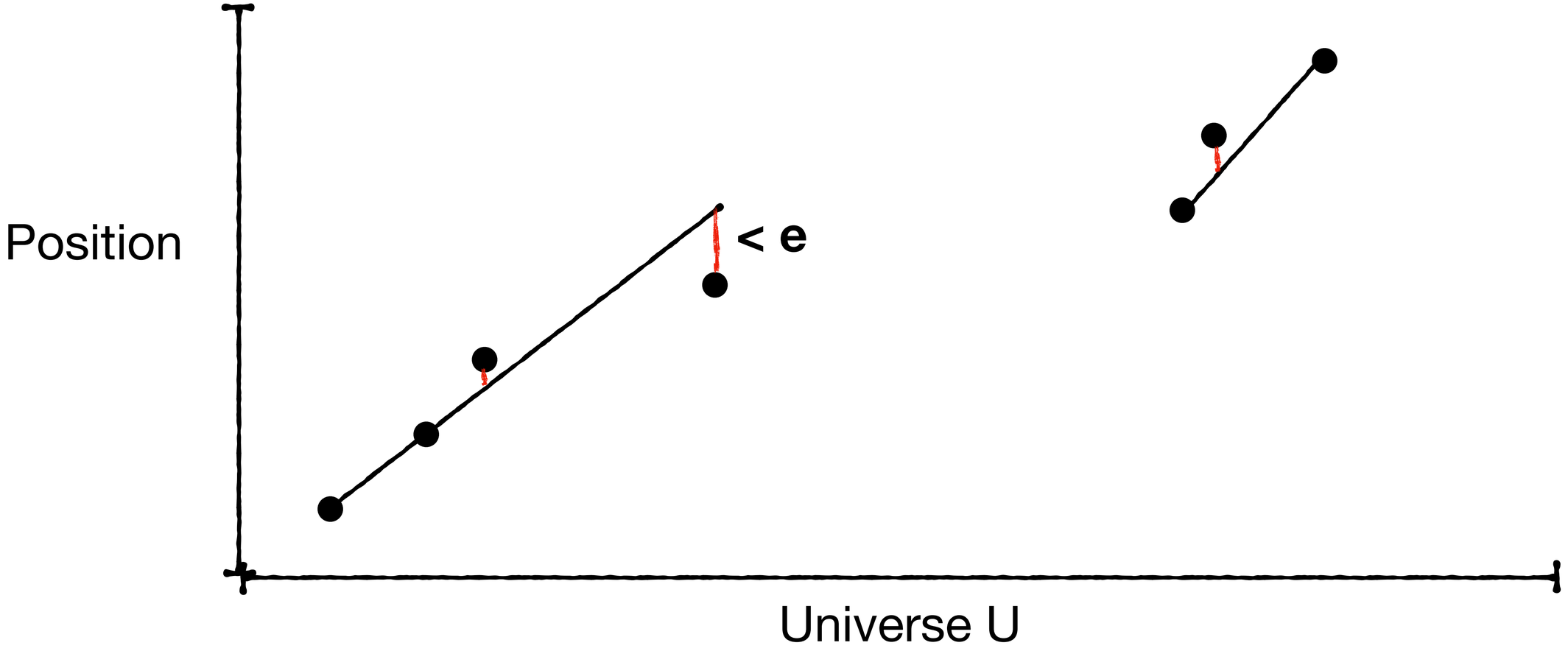


Ensure each point has max distance of e from its function

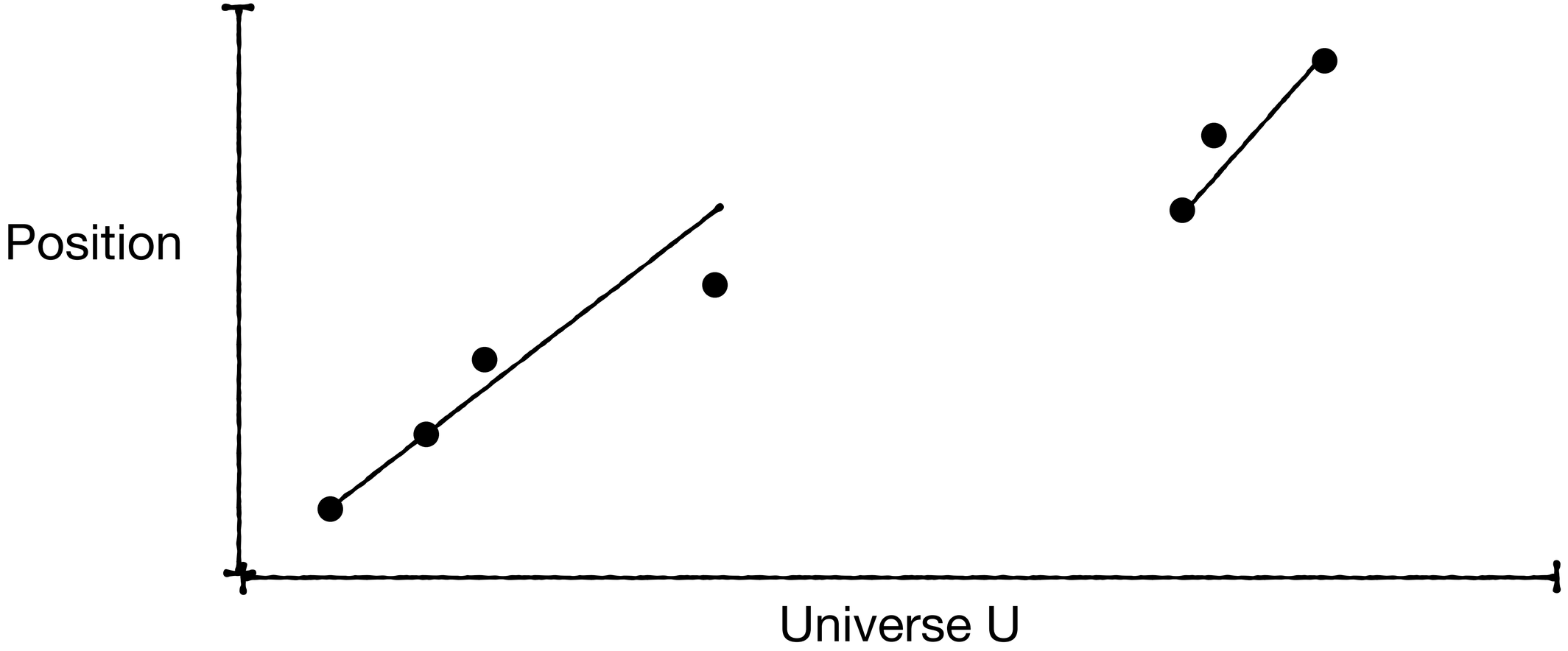
Why?



Why? **To bound the search space size due to an inaccurate prediction**

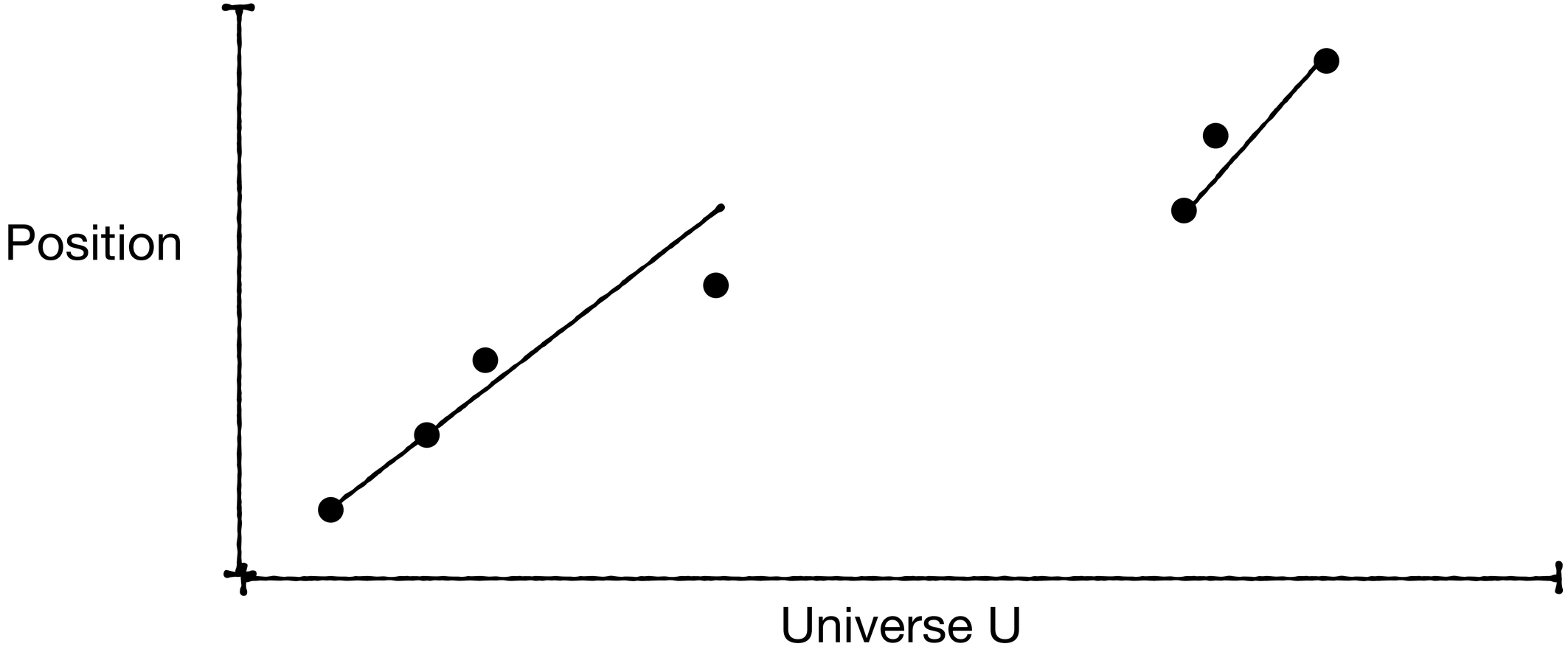


constrained optimization problem



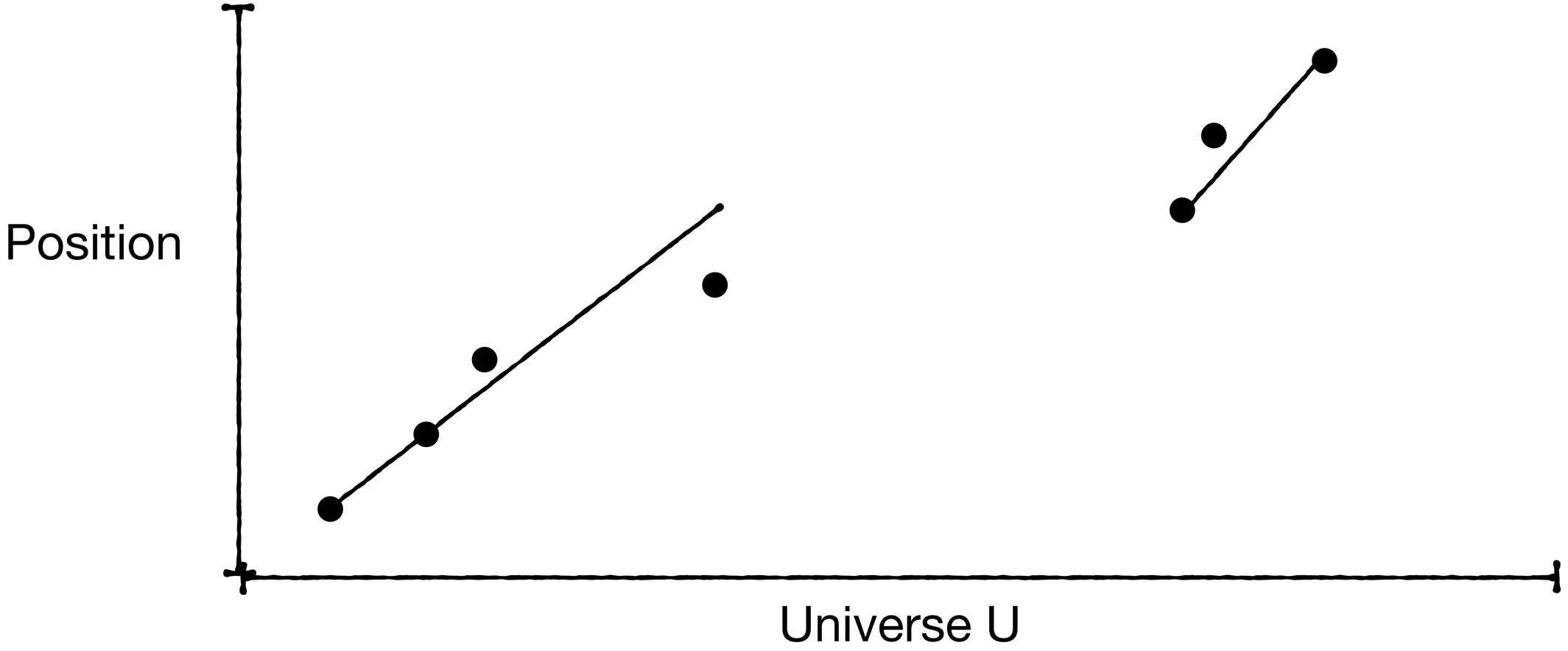
constrained optimization problem:

Model distribution using least number of segments while ensuring all points are within e positions of prediction

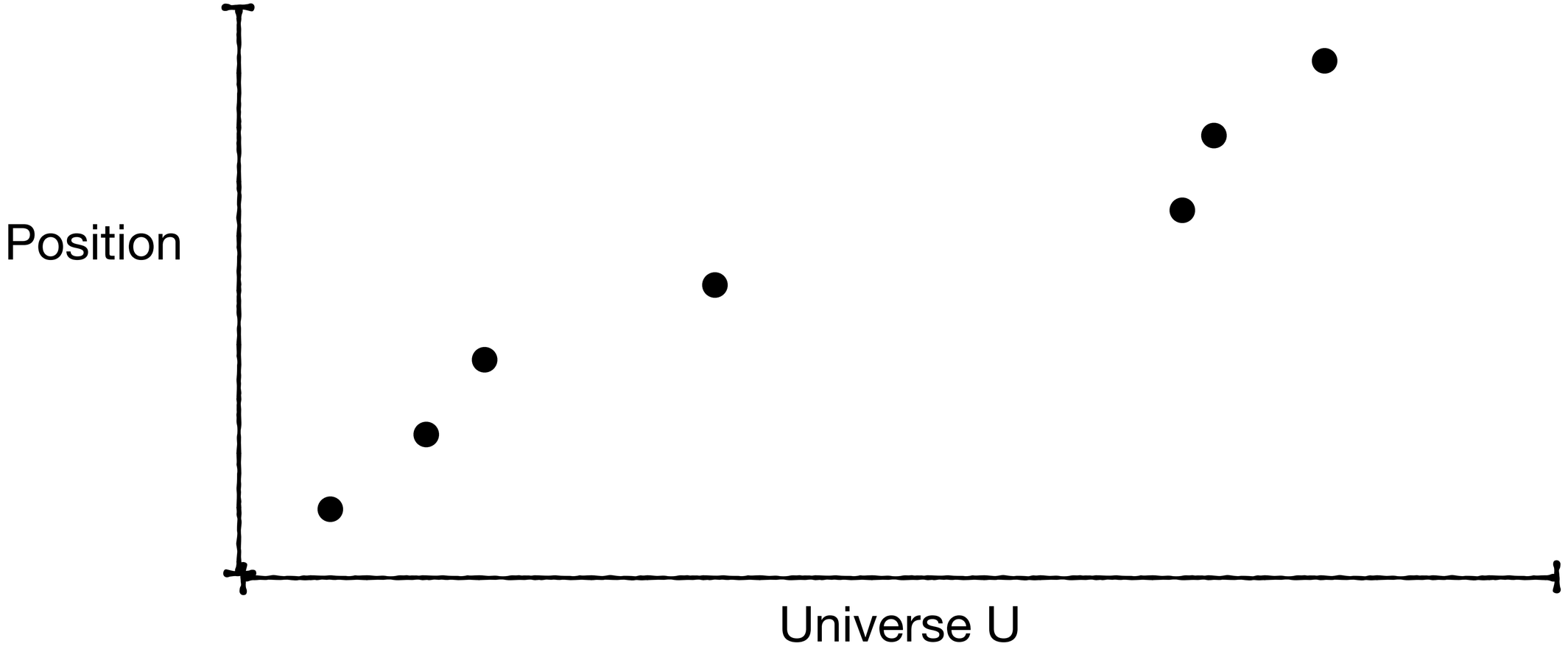


Model distribution using least number of segments while ensuring all points are within e positions of prediction

propose an algorithm for this!

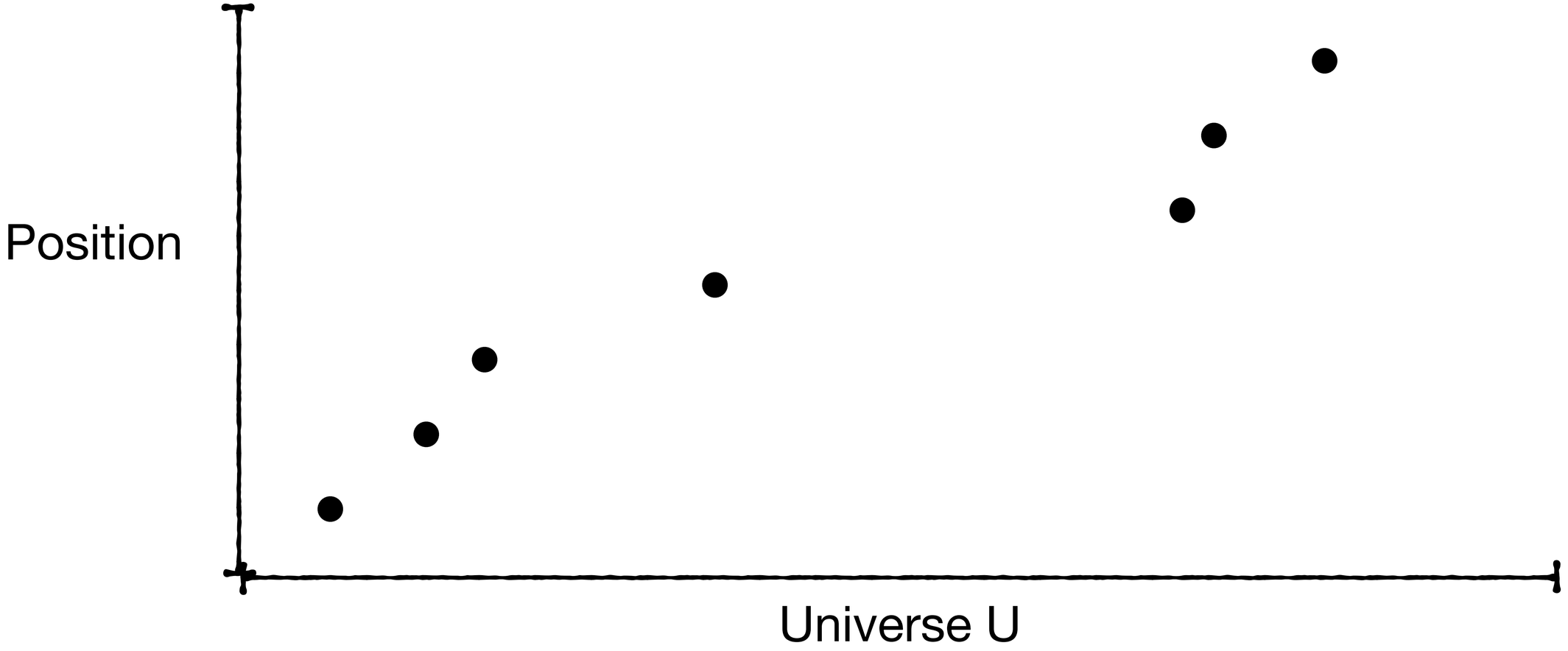


There is a $O(N^2)$ optimal algorithm - too expensive

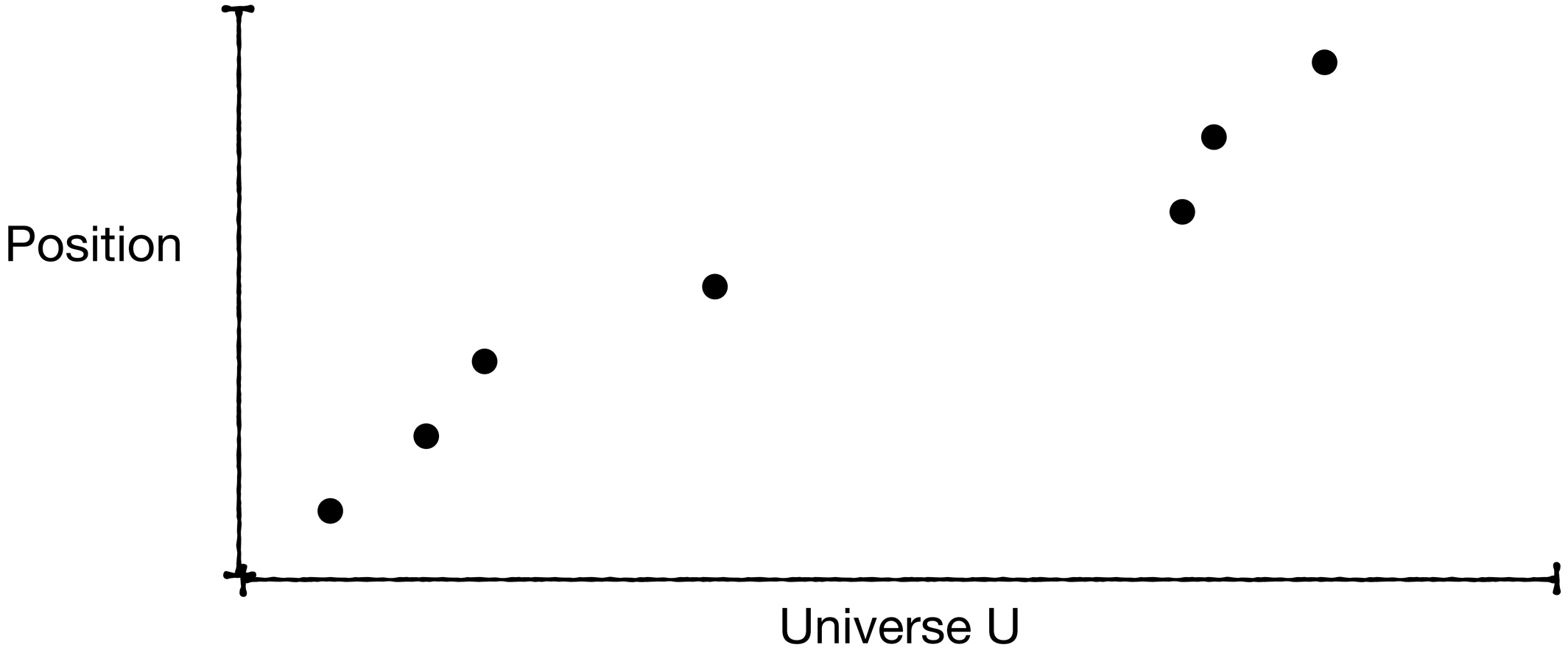


There is a $O(N^2)$ optimal algorithm - too expensive

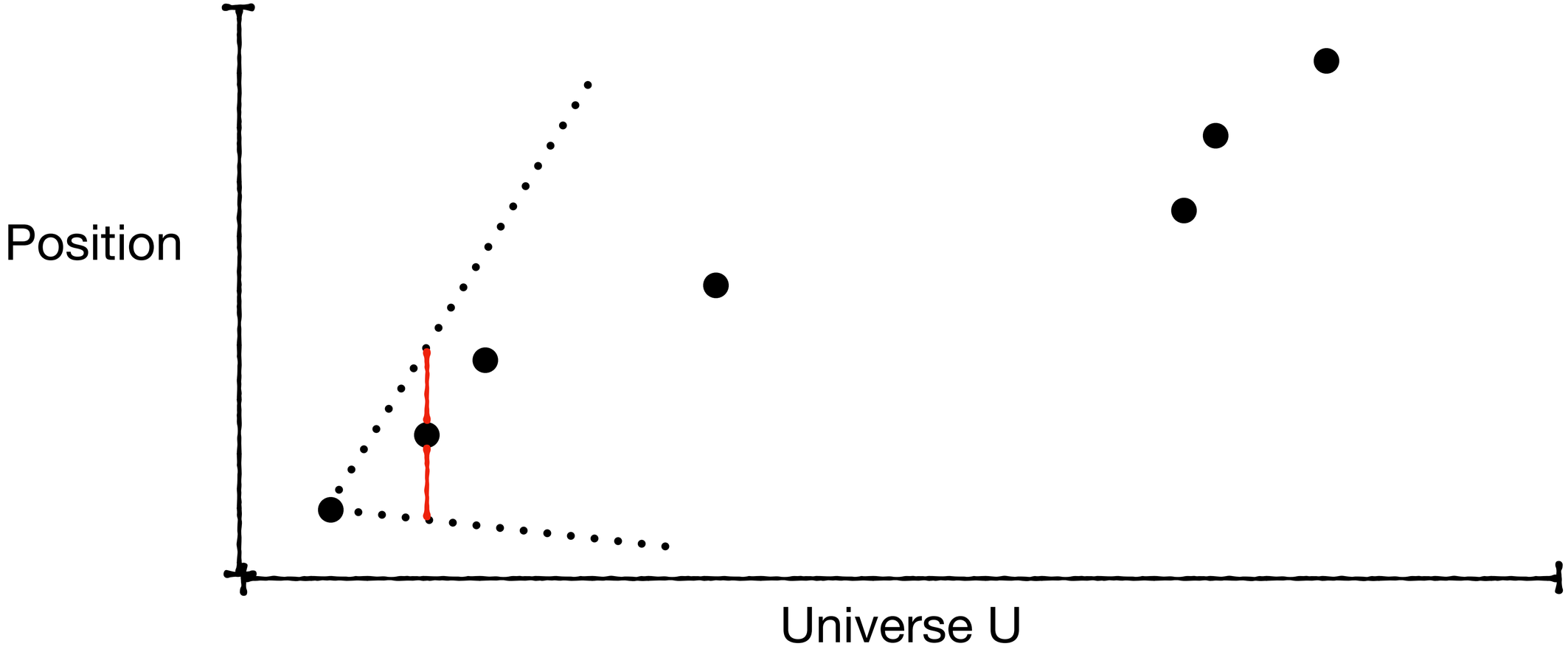
How about a O(N) approximate algorithm?



Add first two points into segment while maintaining “maximal cone”

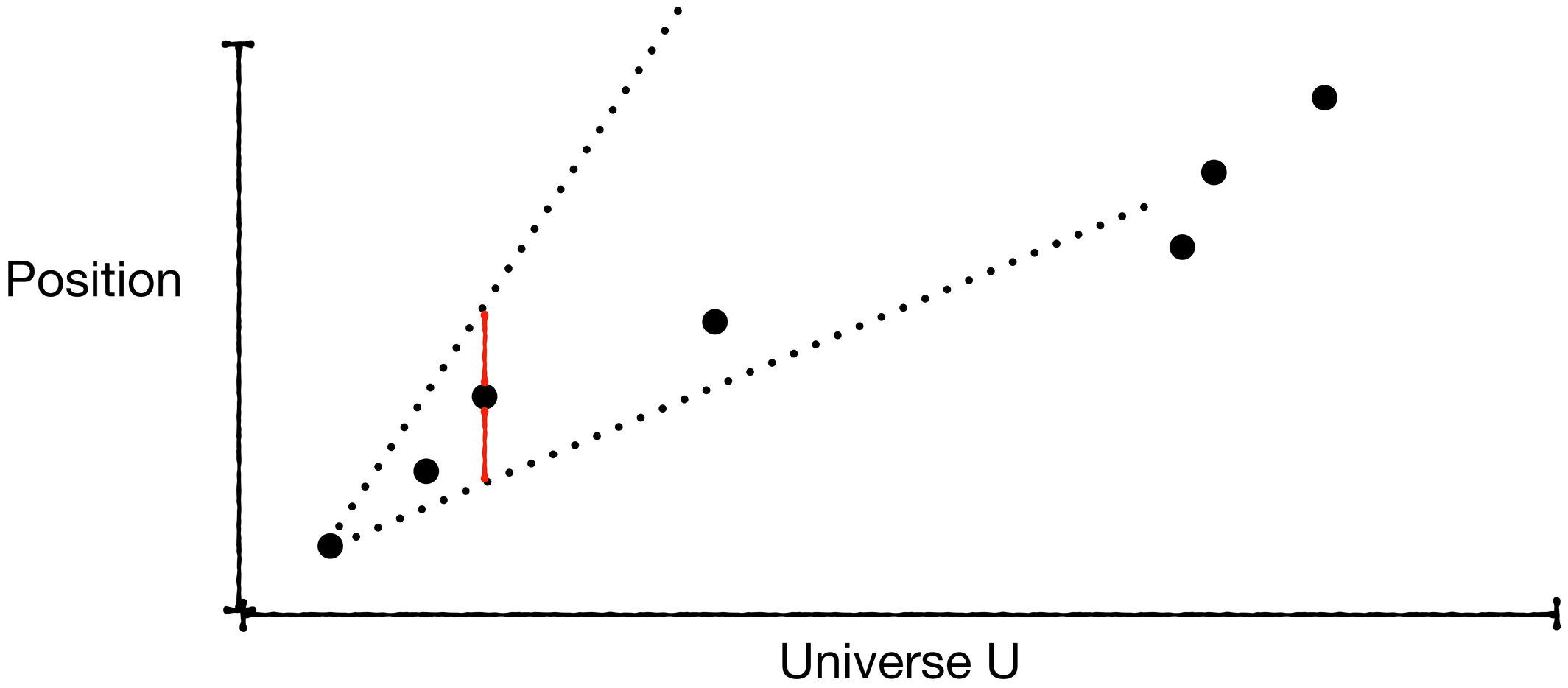


Add first two points into segment while maintaining “maximal cone”



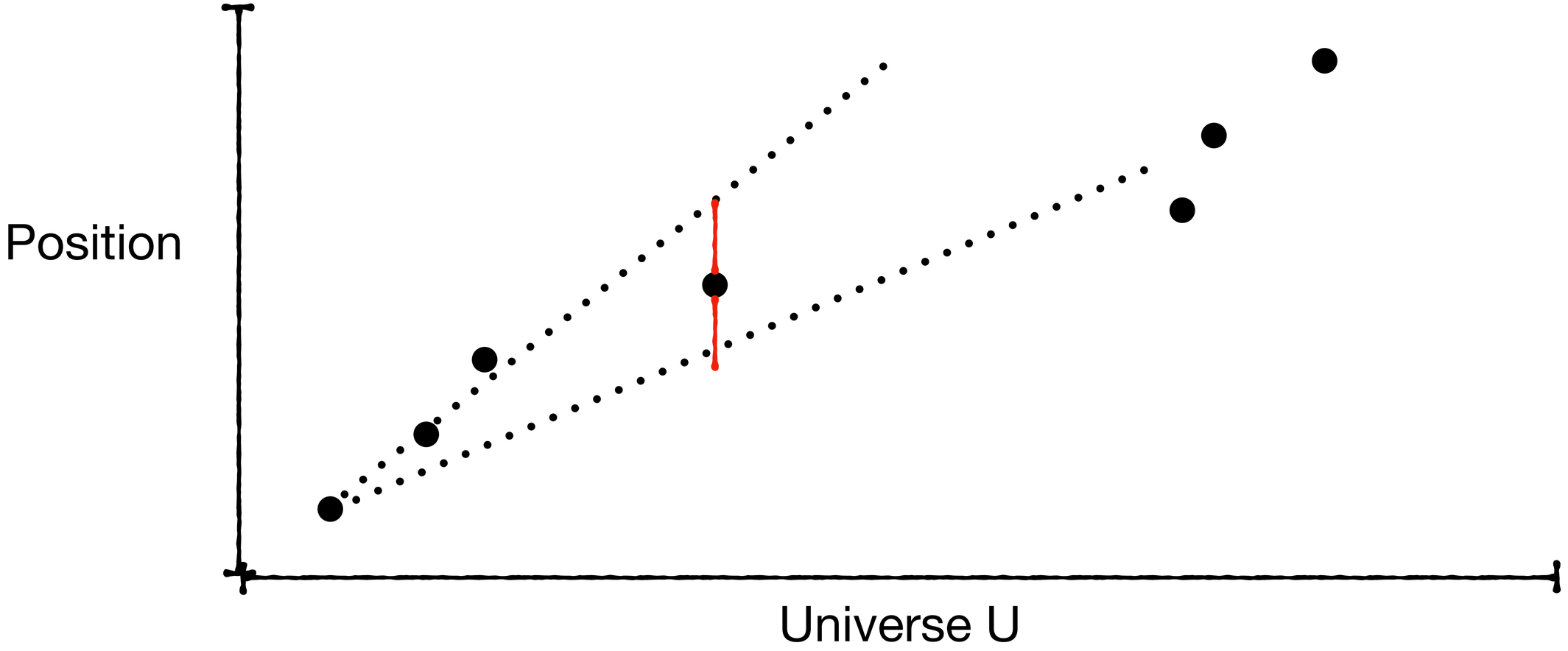
Add first two points into segment while maintaining “maximal cone”

If next point is within cone, add it to segment & narrow cone accordingly



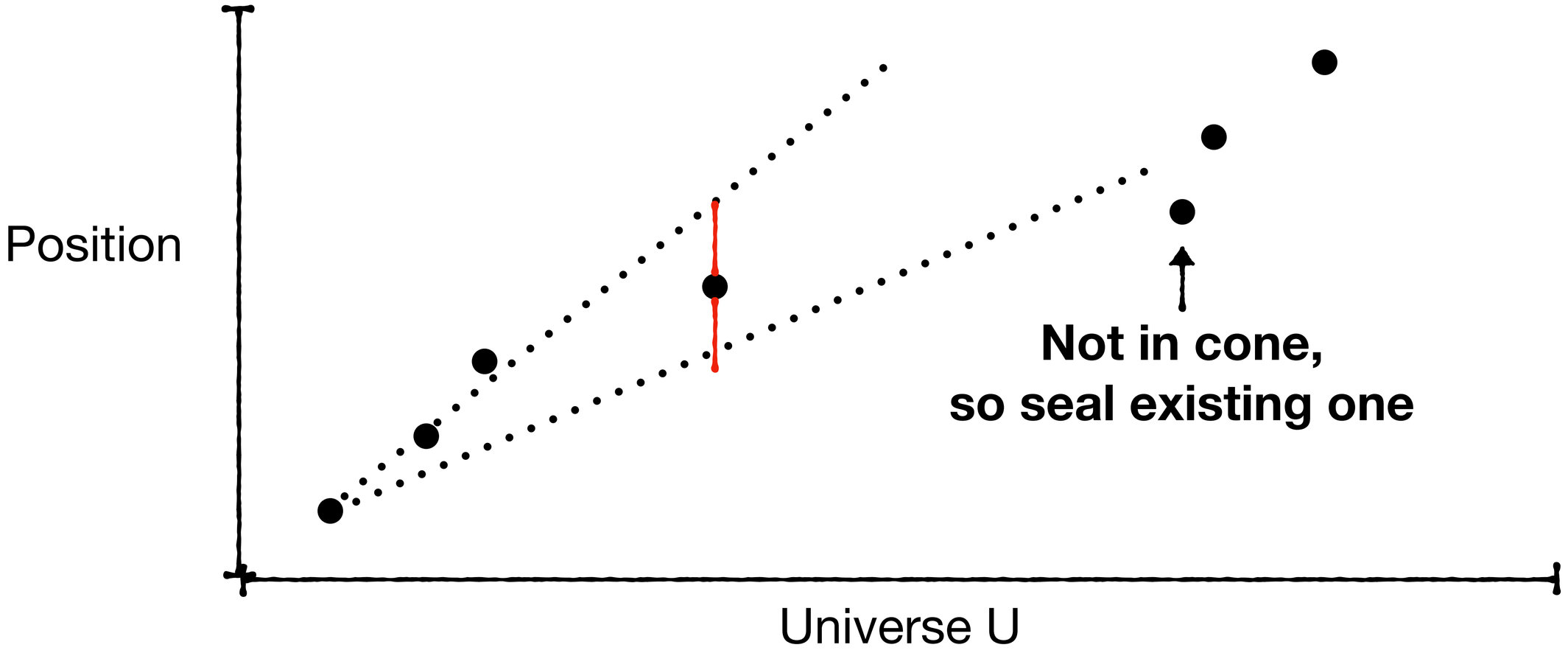
Add first two points into segment while maintaining “maximal cone”

If next point is within cone, add it to segment & narrow cone accordingly



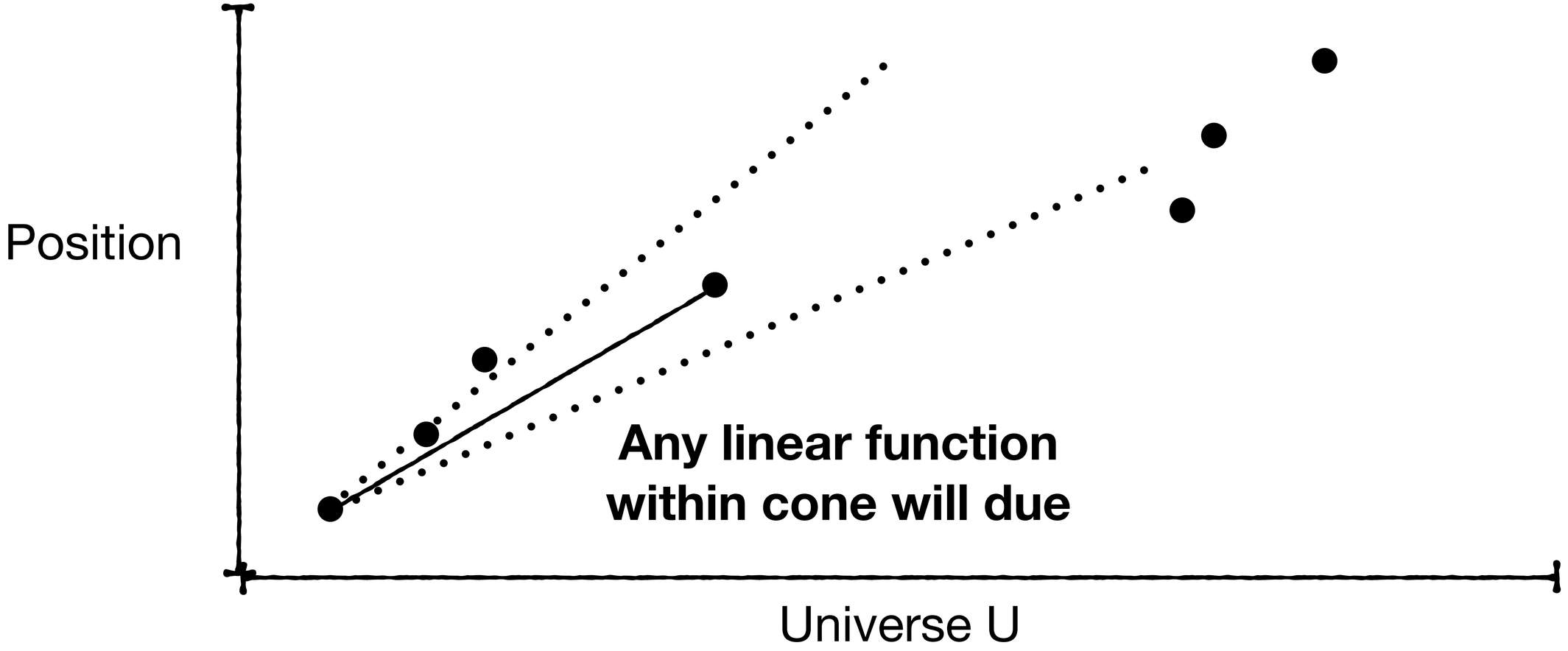
Add first two points into segment while maintaining “maximal cone”

If next point is within cone, add it to segment & narrow cone accordingly



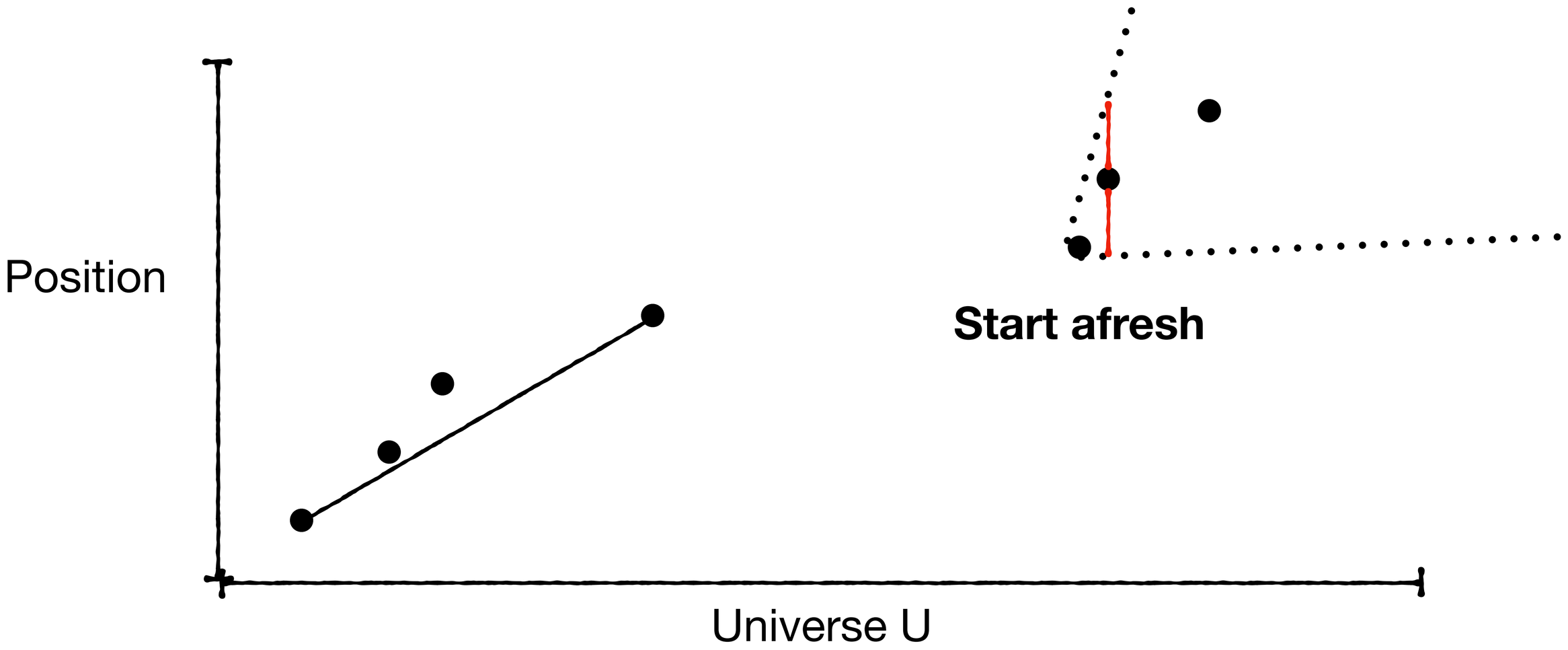
Add first two points into segment while maintaining “maximal cone”

If next point is within cone, add it to segment & narrow cone accordingly



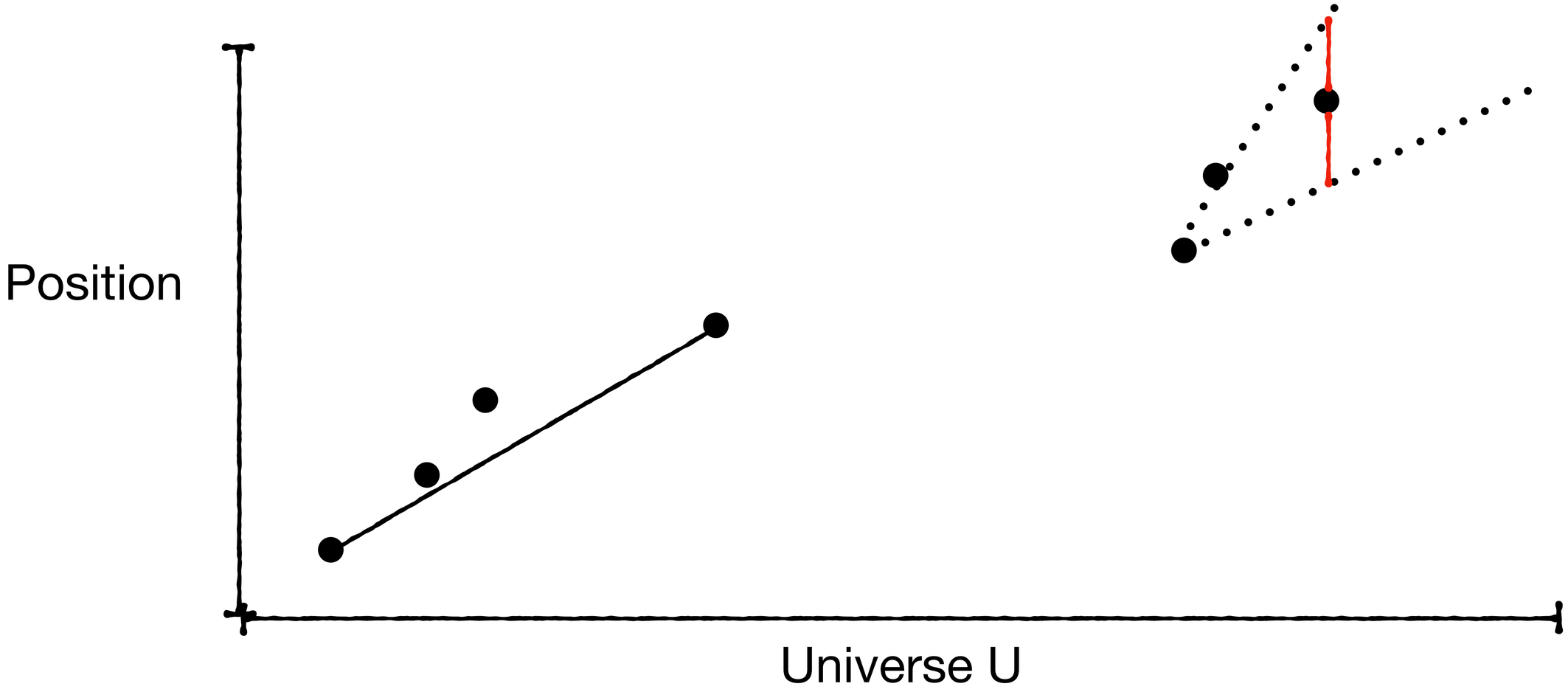
Add first two points into segment while maintaining “maximal cone”

If next point is within cone, add it to segment & narrow cone accordingly



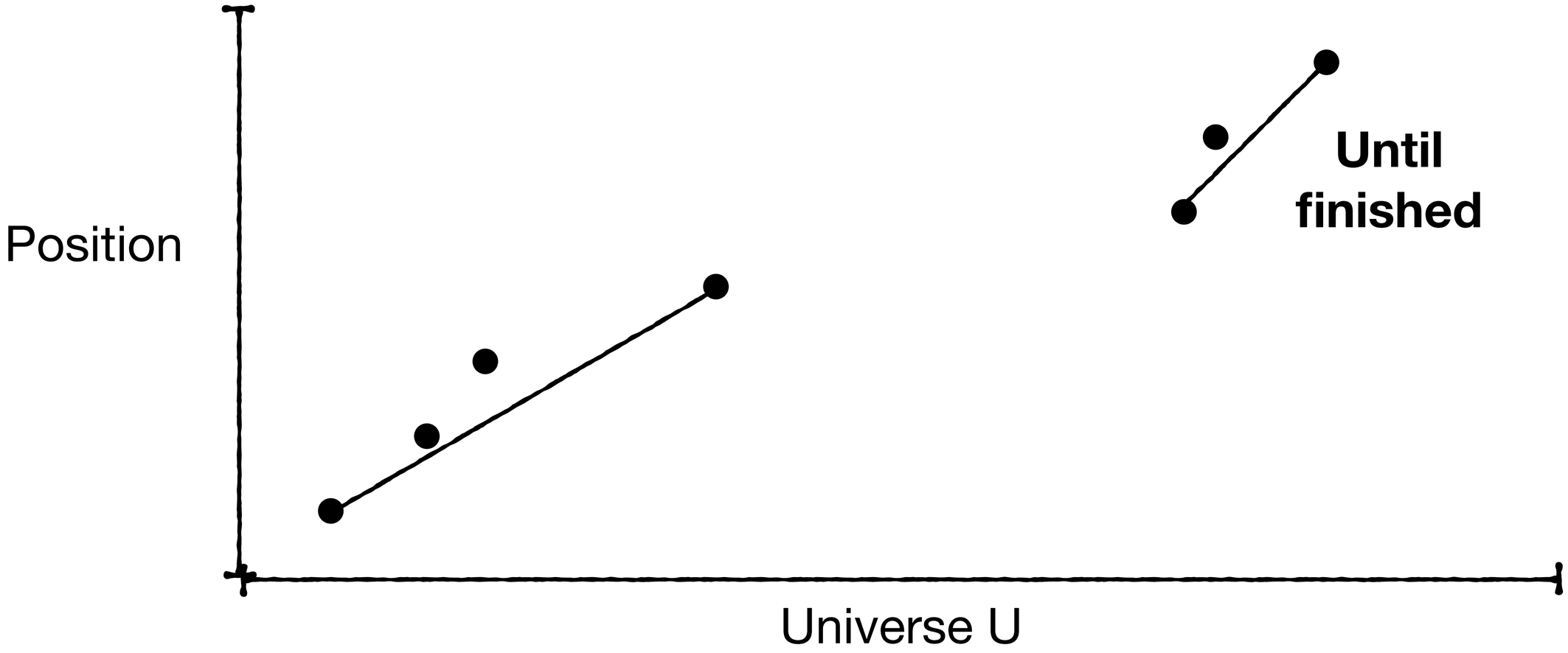
Add first two points into segment while maintaining “maximal cone”

If next point is within cone, add it to segment & narrow cone accordingly

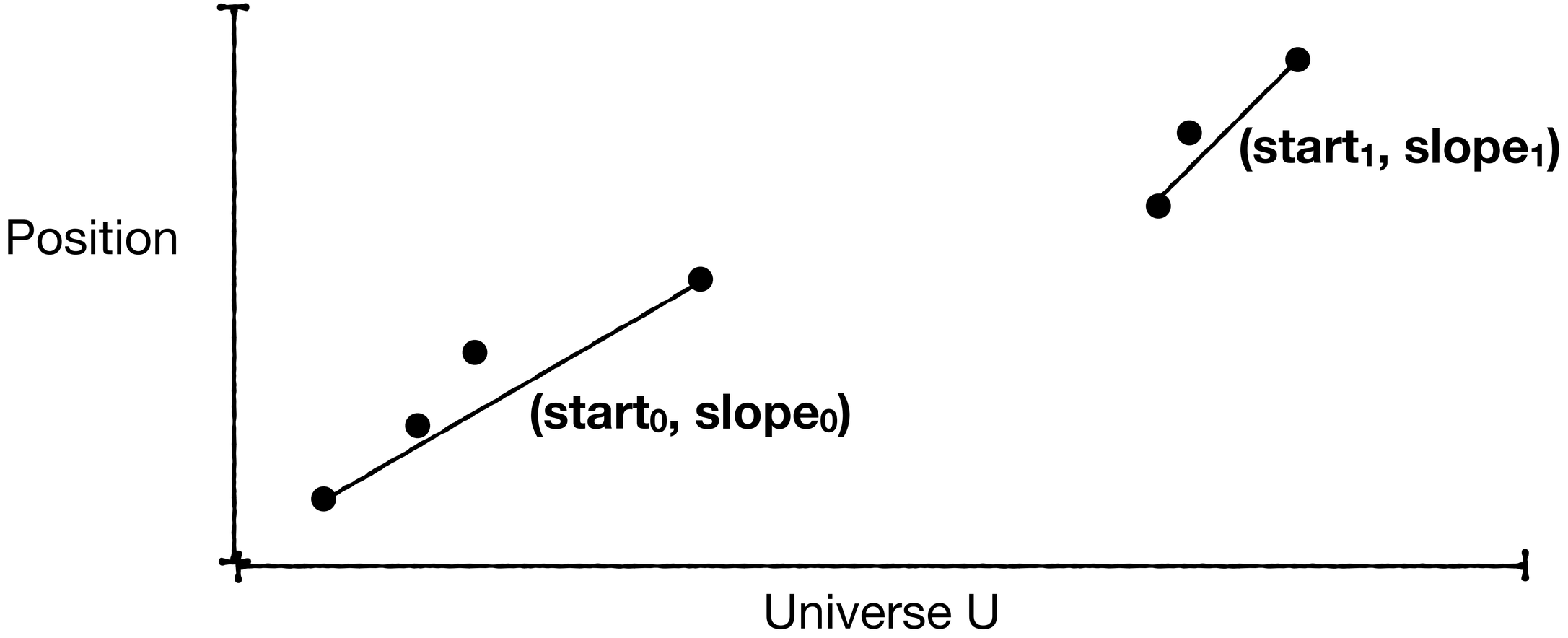


Add first two points into segment while maintaining “maximal cone”

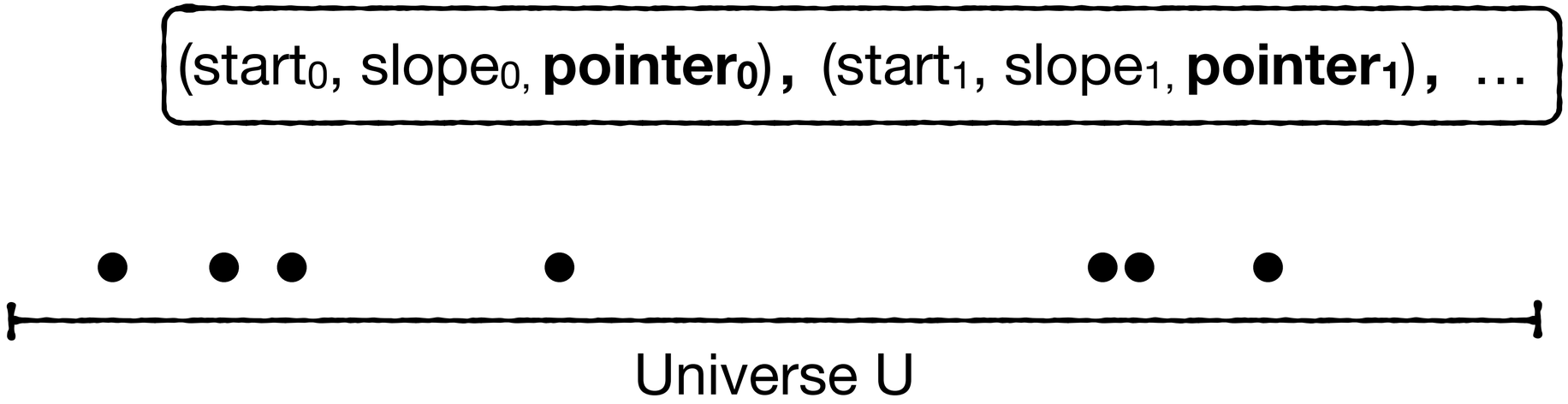
If next point is within cone, add it to segment & narrow cone accordingly



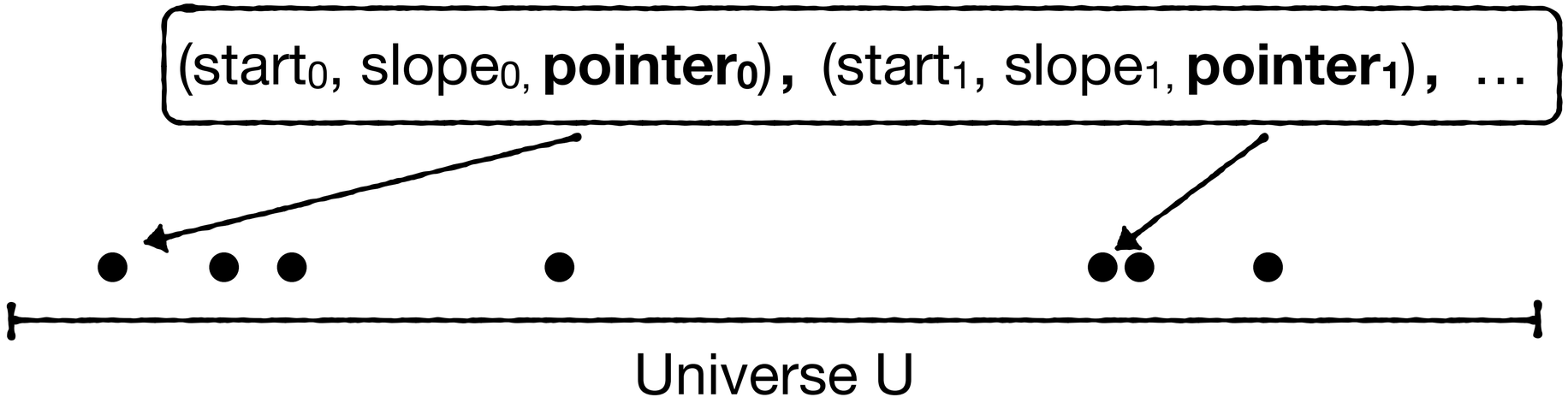
Each segment is characterized by starting point and slope



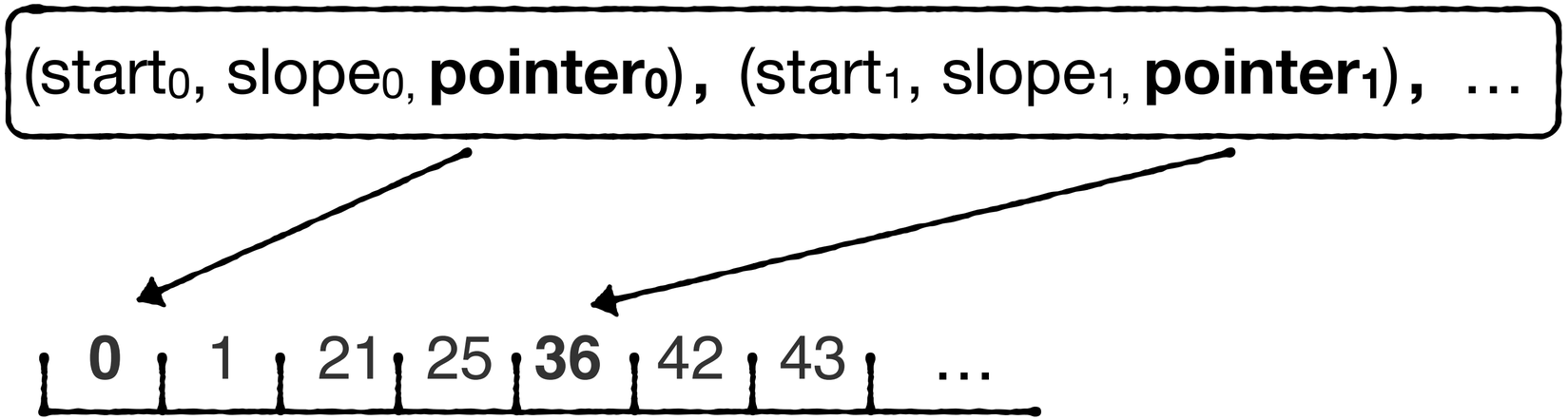
Place in B-tree node, sorted by starting point



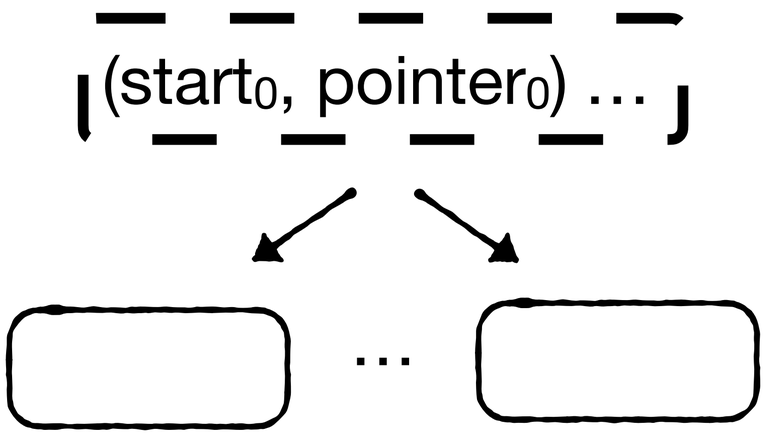
Place in B-tree node, sorted by starting point



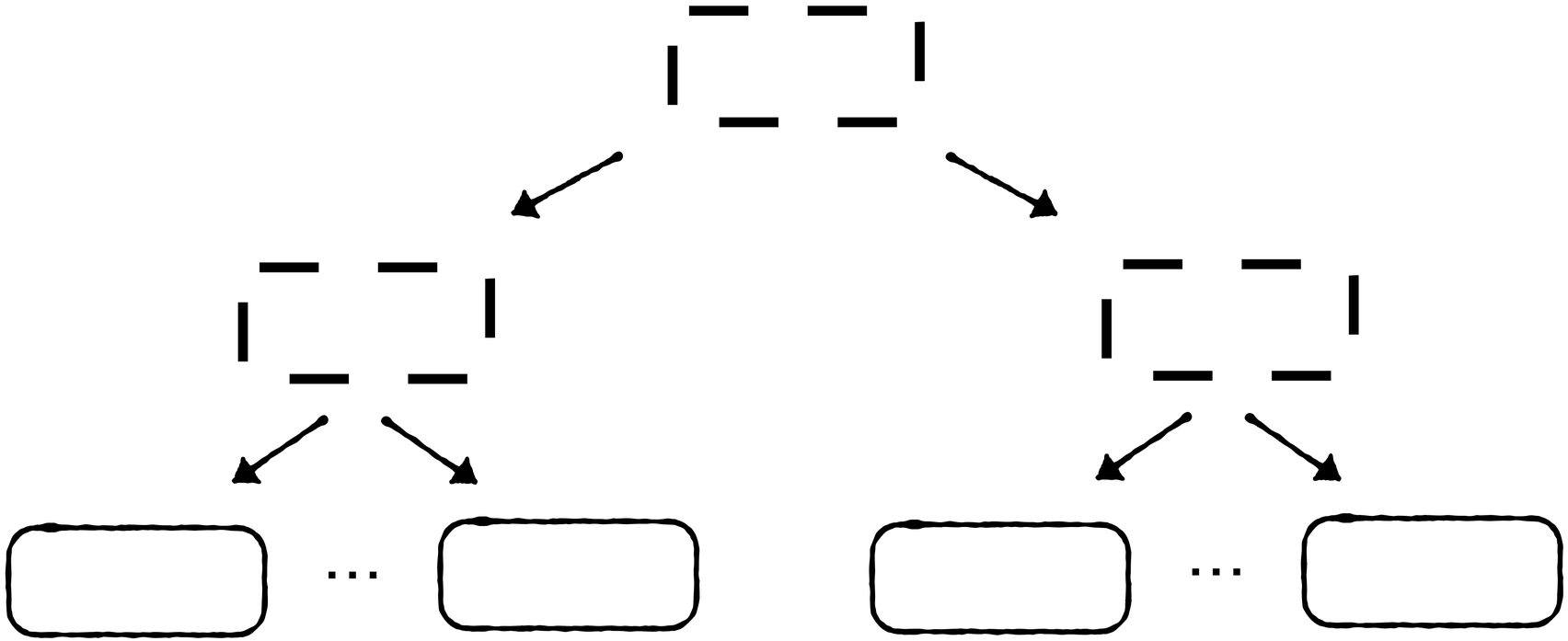
Place in B-tree node, sorted by starting point



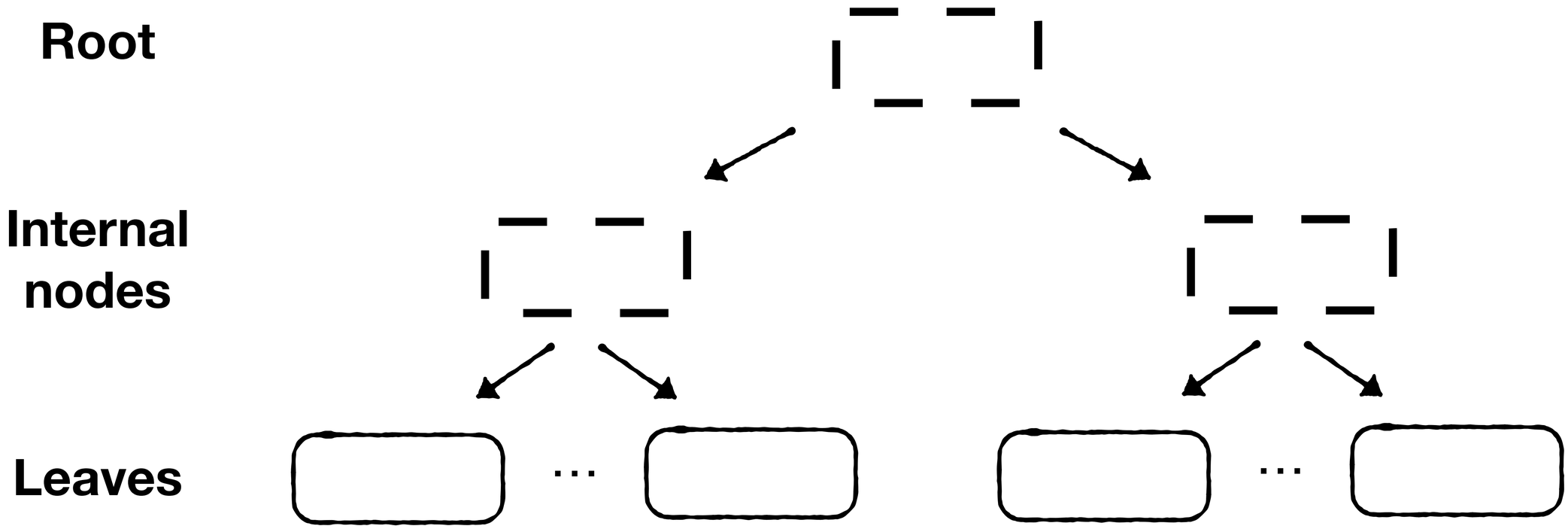
Store as a B-tree



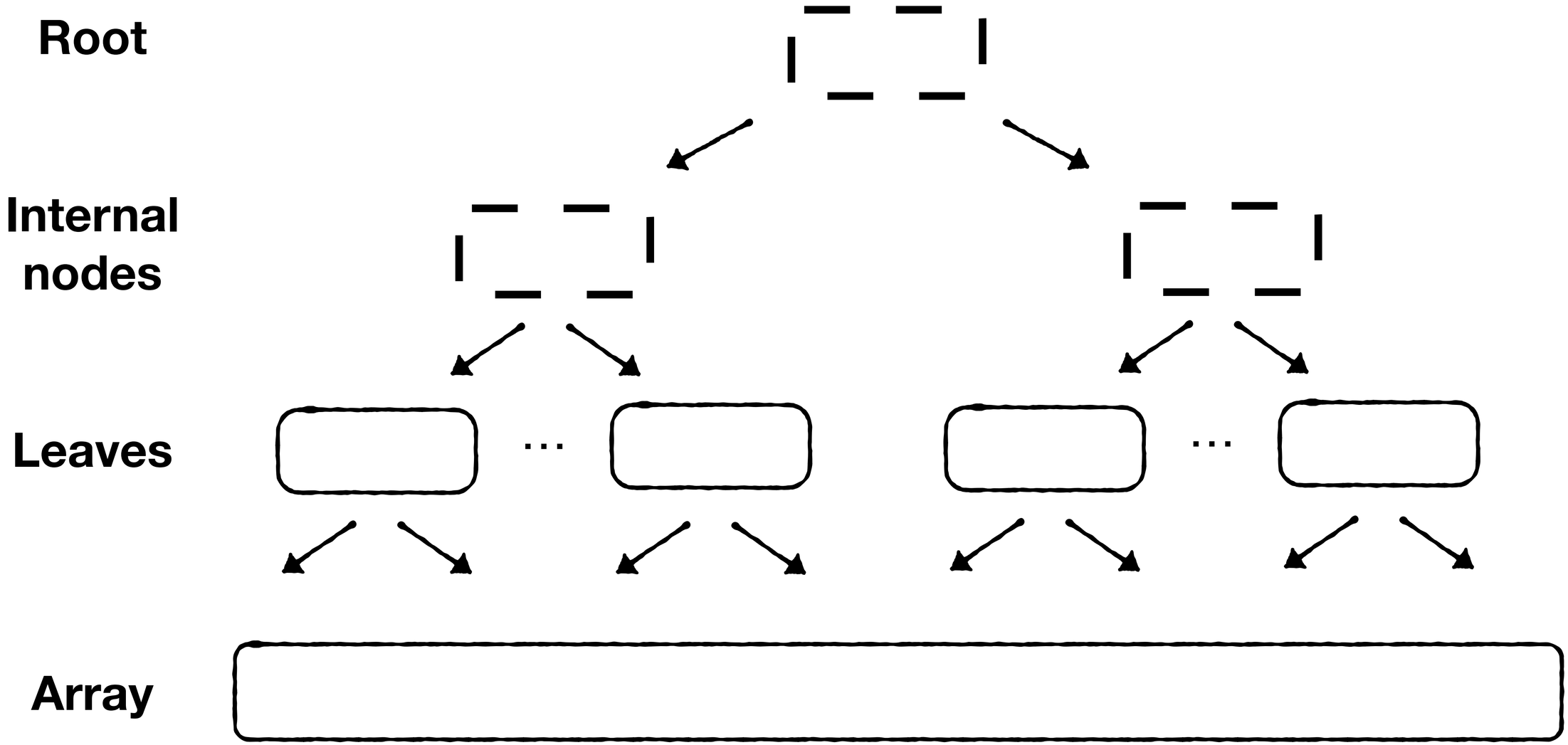
Store as a B-tree



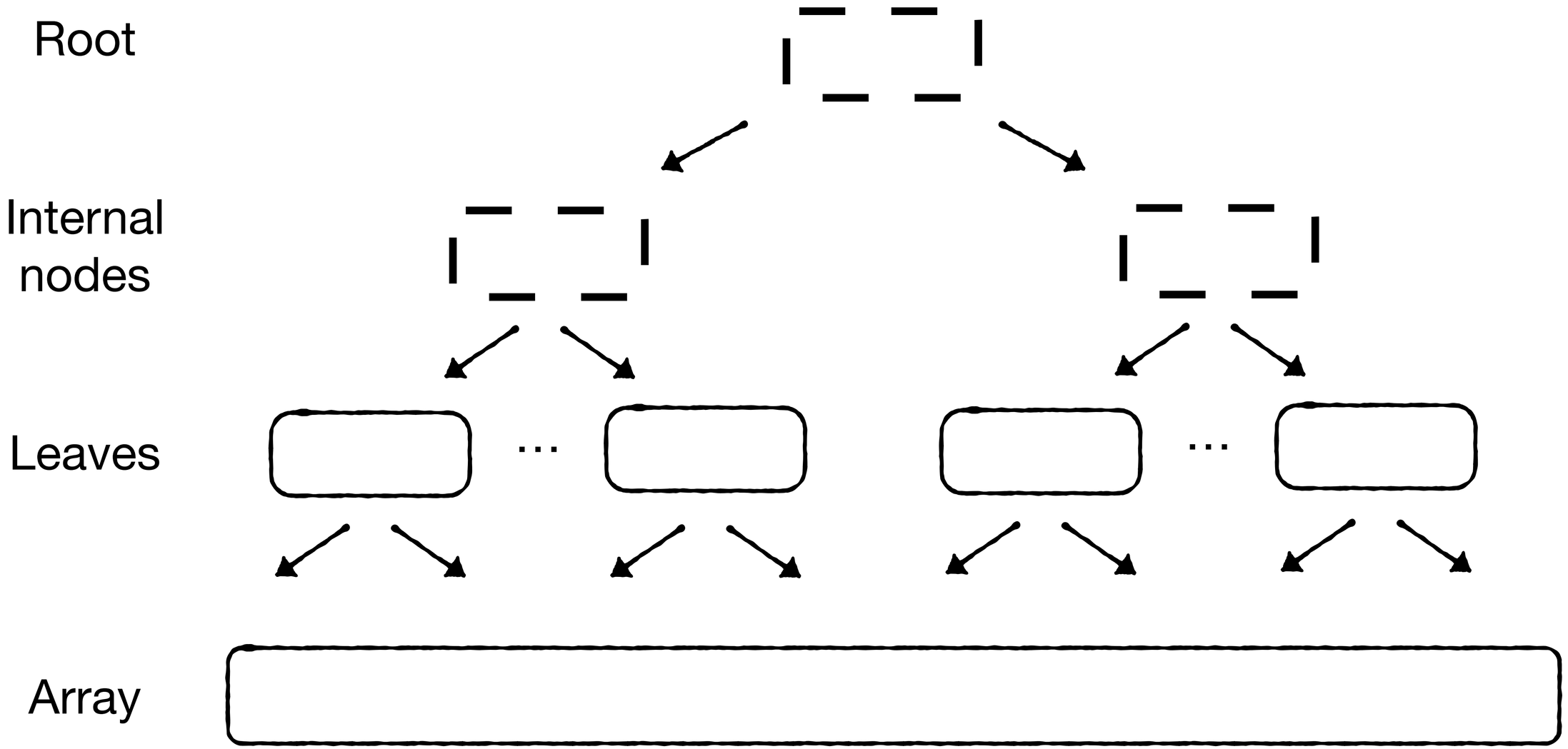
Store as a B-tree



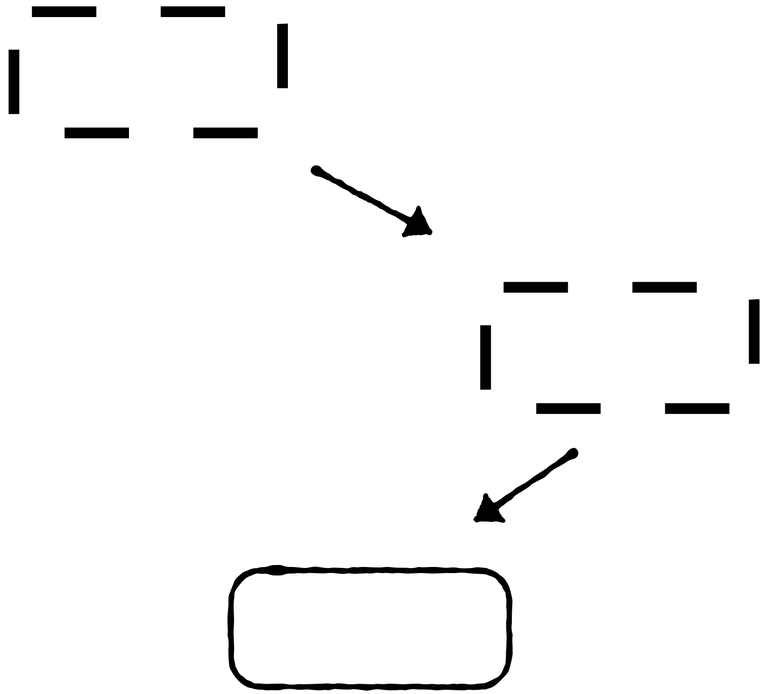
Store as a B-tree



How to query this tree? e.g., get(15)



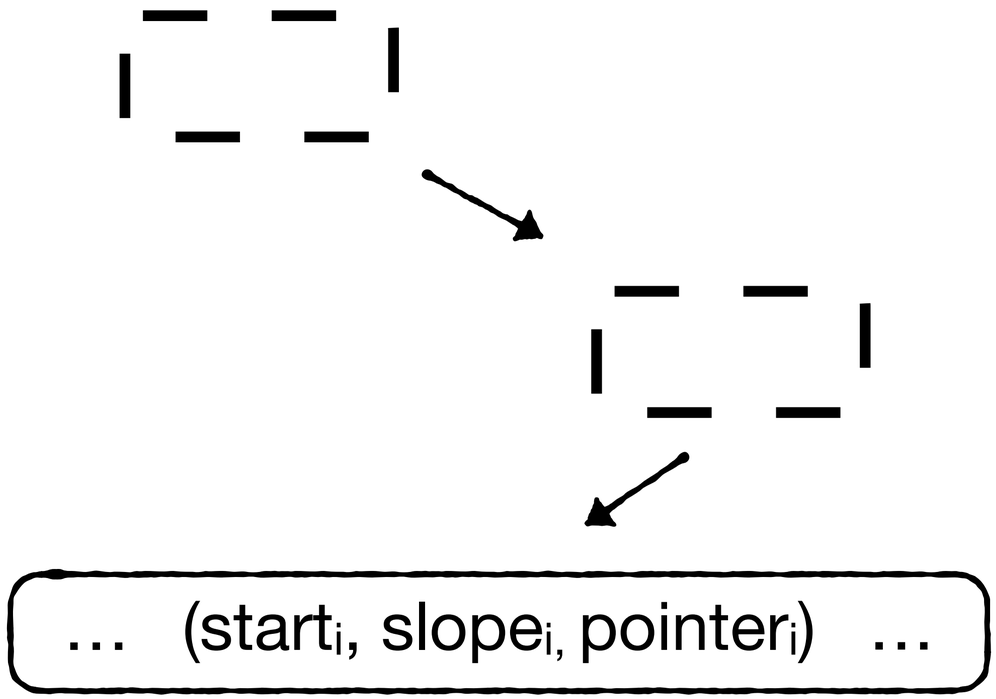
How to query this tree? **(1) traverse B-tree**



How to query this tree?

(1) traverse B-tree

(2) Find starting segment



How to query this tree?

(1) traverse B-tree

(2) Find starting segment

(3) interpolate using slope

Position prediction = start + (key - start) / slope

… ($start_1$, $slope_1$, $pointer_1$) …

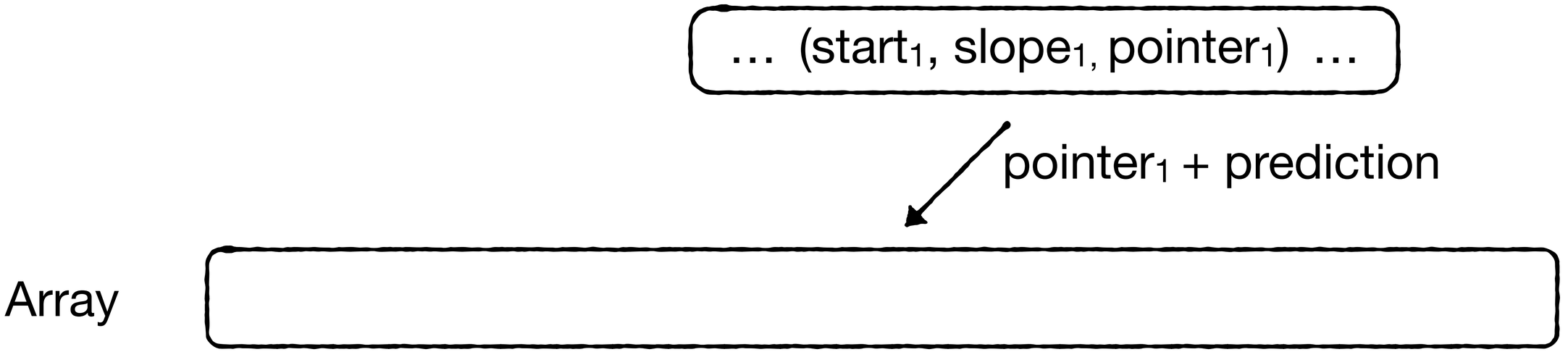
How to query this tree?

(1) traverse B-tree

(2) Find starting segment

(3) interpolate using slope

(4) add prediction to pointer and access array



How to query this tree?

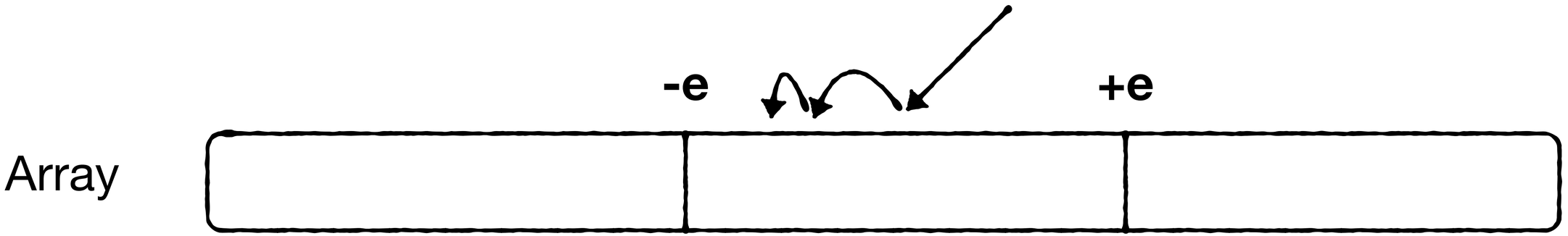
(1) traverse B-tree

(2) Find starting segment

(3) interpolate using slope

(4) add prediction to pointer and access array

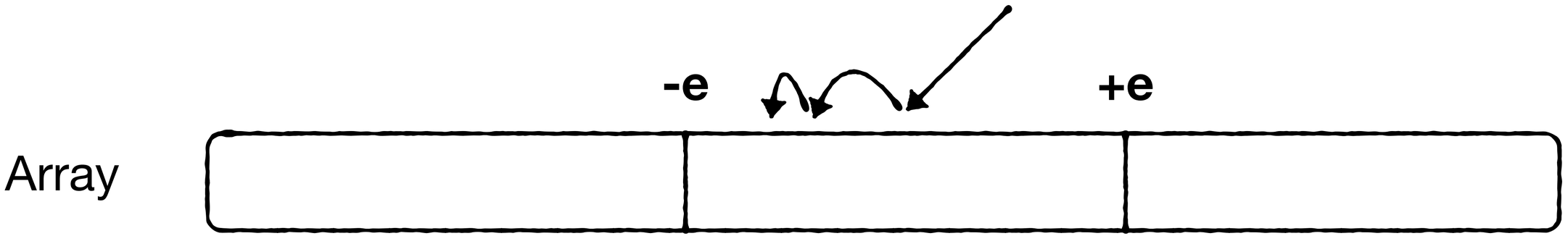
(5) binary search within max error bounds



How to query this tree?

(1) traverse B-tree
(2) Find starting segment
(3) interpolate using slope
(4) add prediction to pointer and access array
(5) binary search within max error bounds

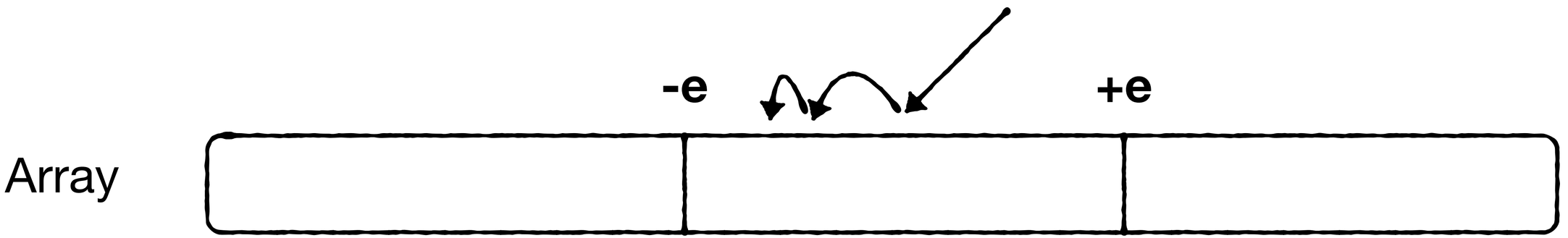
Query cost?



How to query this tree?

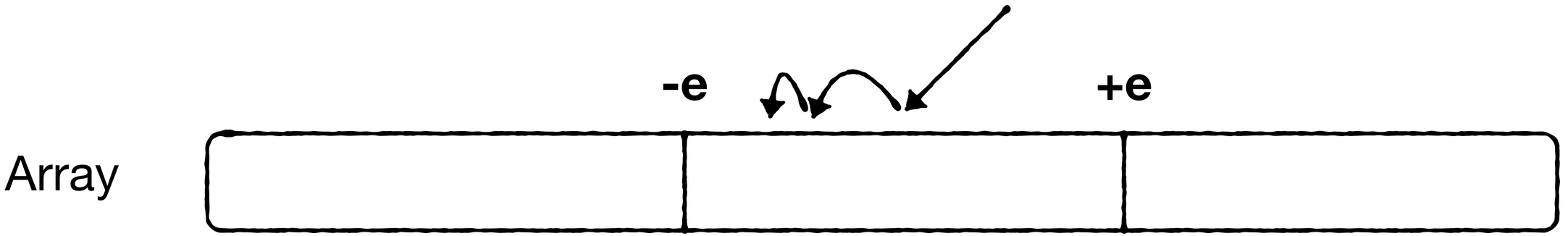
(1) traverse B-tree
(2) Find starting segment
(3) interpolate using slope
(4) add prediction to pointer and access array
(5) binary search within max error bounds

Query cost? **O(log(#segments) + log(e))**

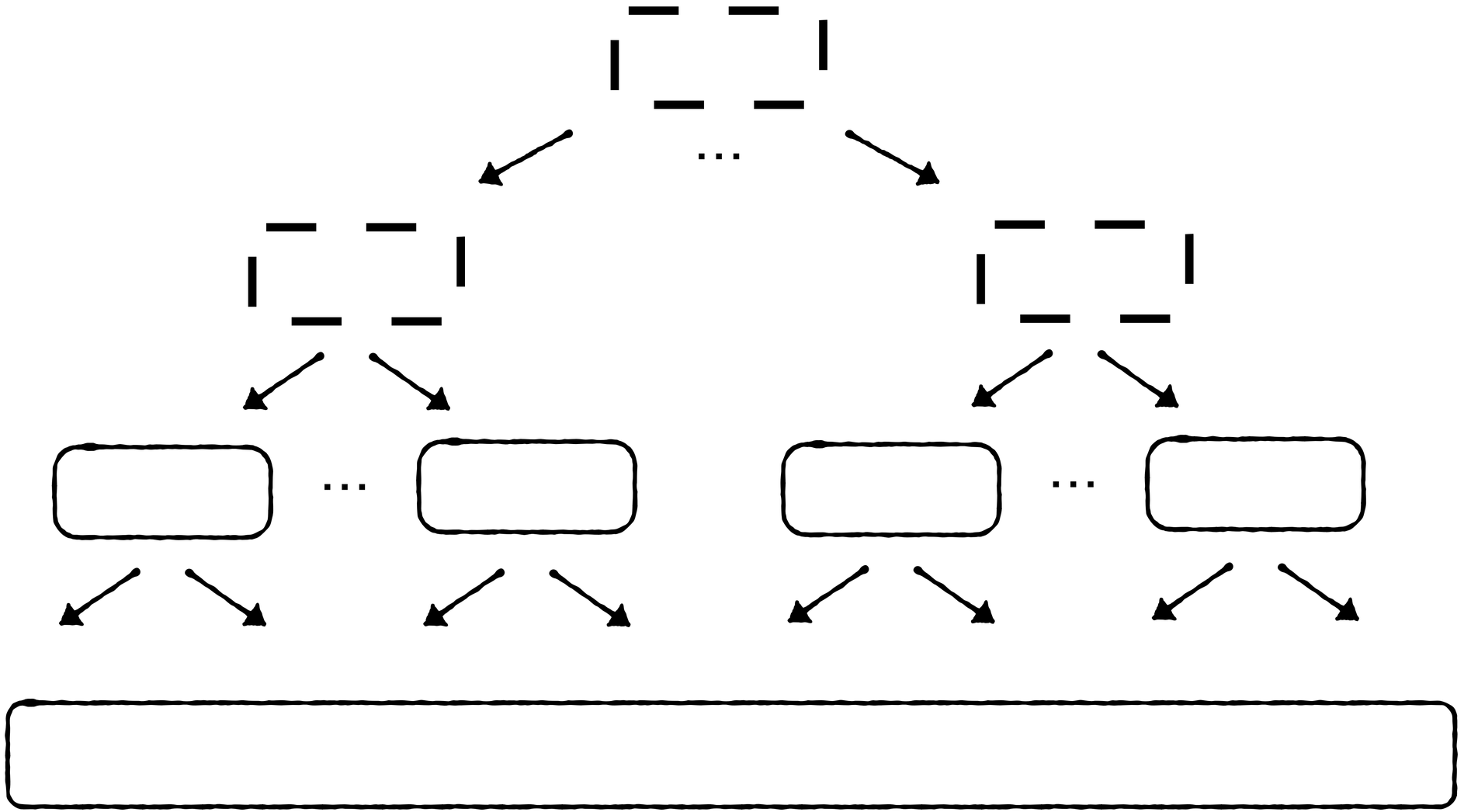


Query cost? O(log(**#segments**) + log(**e**))

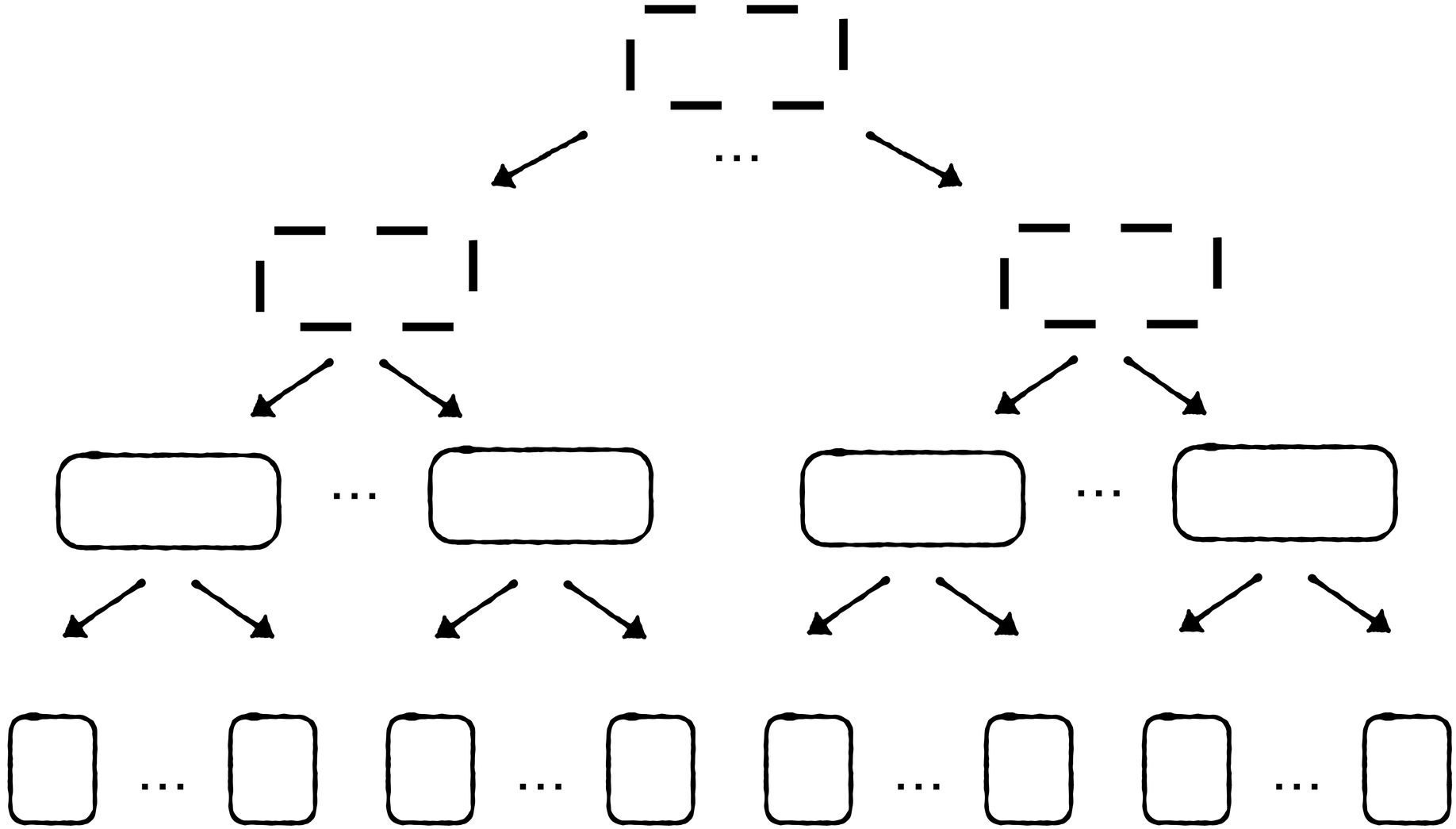
The more predictable the data is, the more the cost drops



How to handle updates?



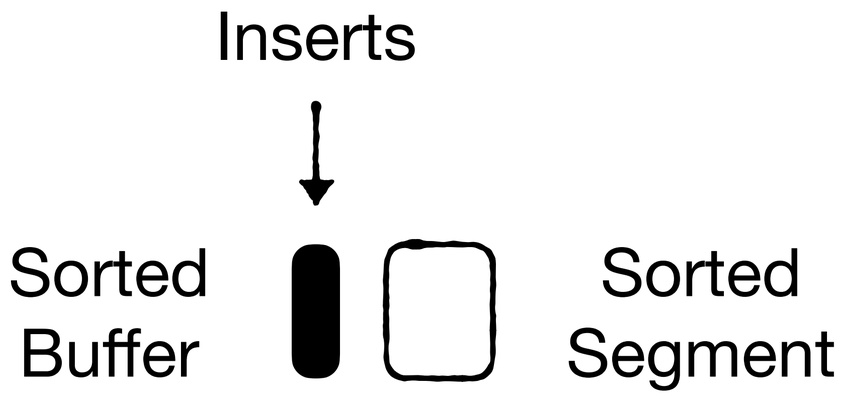
How to handle updates? **(1) separate segments into contiguous chunks**



How to handle updates?

(1) separate segments into contiguous chunks

(2) employ sorted insert buffer for each segment

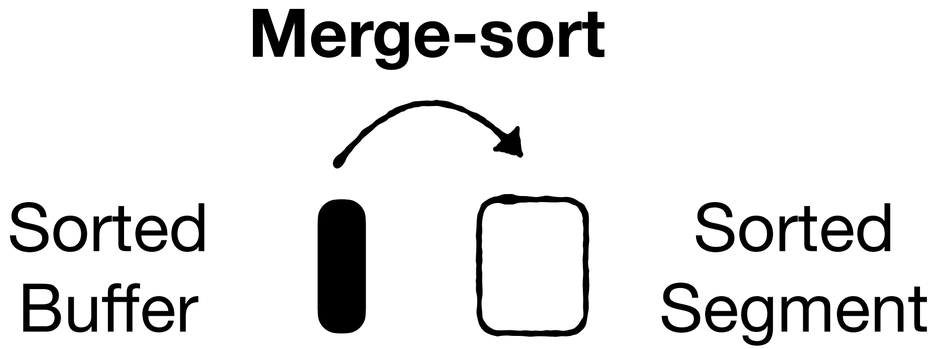


How to handle updates?

(1) separate segments into contiguous chunks

(2) employ sorted insert buffer for each segment

(3) merge buffer into segment at threshold size



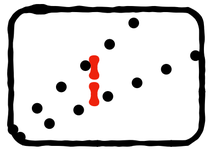
How to handle updates?

(1) separate segments into contiguous chunks

(2) employ sorted insert buffer for each segment

(3) merge buffer into segment at threshold size

(4) rerun segmentation (cone) algorithm



Sorted
Segment

How to handle updates?

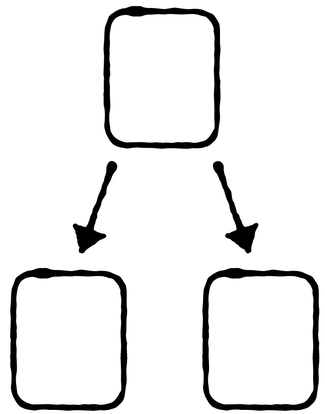
(1) separate segments into contiguous chunks

(2) employ sorted insert buffer for each segment

(3) merge buffer into segment at threshold size

(4) rerun segmentation (cone) algorithm

(5) if segment splits, update parent/s



Sorted
Segment

How to handle updates?

(1) separate segments into contiguous chunks

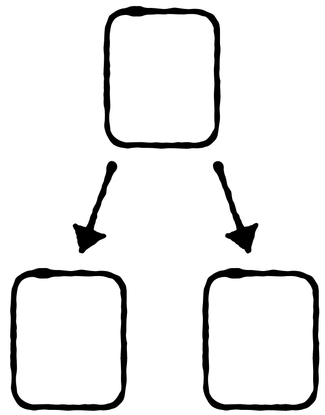
(2) employ sorted insert buffer for each segment

(3) merge buffer into segment at threshold size

(4) rerun segmentation (cone) algorithm

(5) if segment splits, update parent/s

Thus, FITing tree depends heavily on insertion order

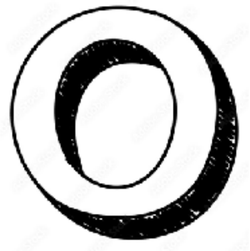


Sorted
Segment

Flaw 1: Thus, FITing tree depends heavily on insertion order

Flaw 2: **Worst-case query cost depends on O(log(#segments))**

Better Worst-Case



AVL-Tree
1962

B-Tree
1970

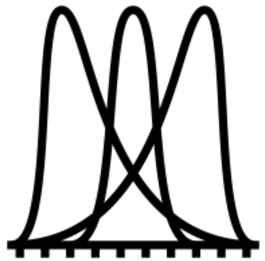
T-Tree
1985

CSS-Tree
1998

CSB-Tree
2000

HOT
2018

Exploiting Data Distribution



Interpolation search
1959

FITing-Tree
2019