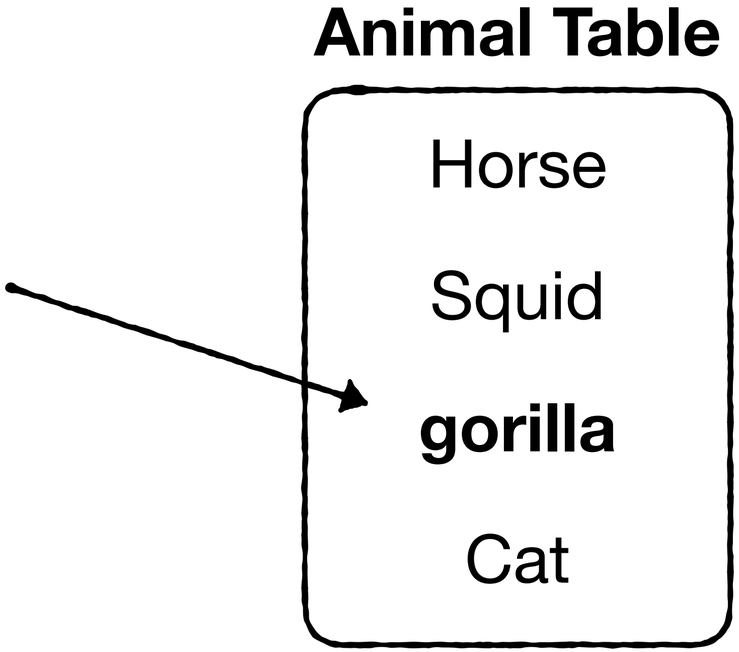
In-Memory Indexing

Niv Dayan - CSC2525 Research Topics in Database Management

Indexes

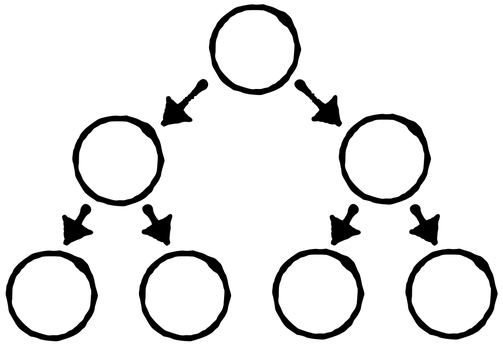
Select * from animals where name = “**gorilla**”

How to find quickly without a full scan?

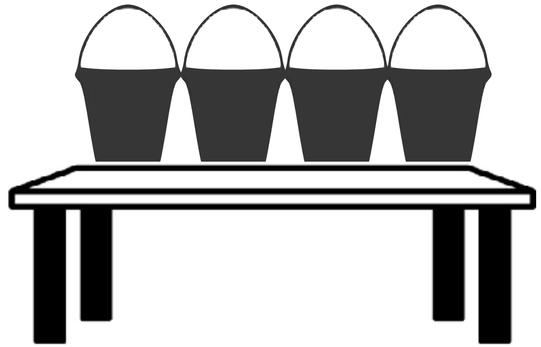


Indexes

You have learned about



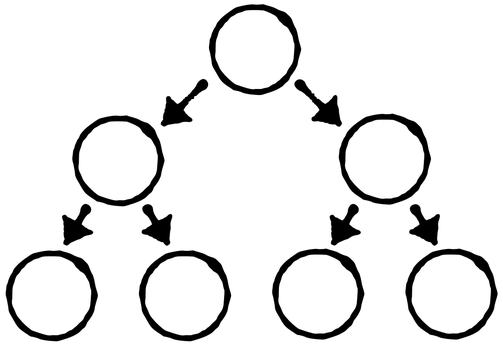
Binary trees



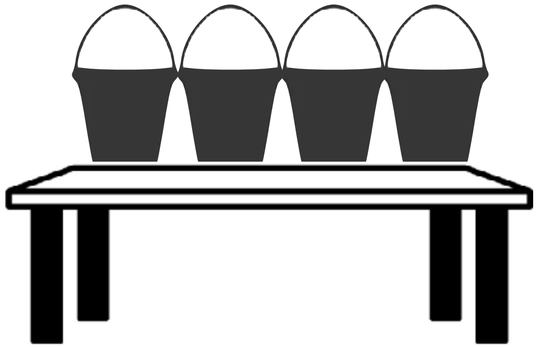
Hash tables

Indexes

You have learned about



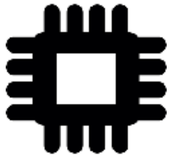
Binary trees



Hash tables

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

The memory Hierarchy



CPU caches

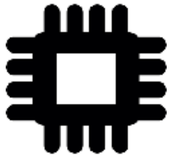
memory

SSD

disk

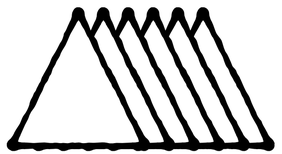
Expensive & fast

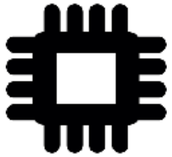
Slow & cheap



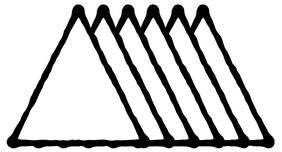
Not enough space

Indexes stored here

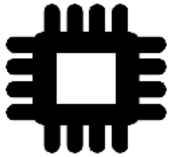




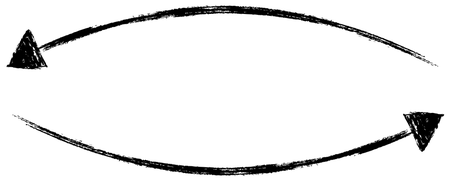
Indexes



Block-addressable
4-16 KB

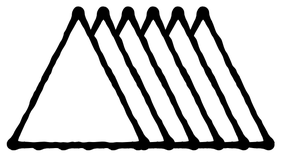


Read I/O



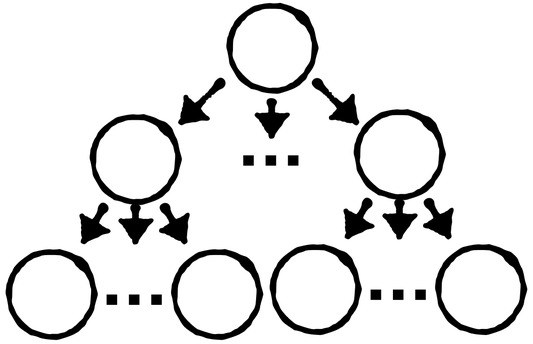
Write I/O

Indexes



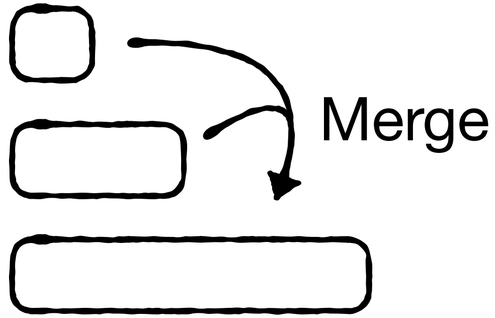
Goal: minimize block I/Os

Indexes for Storage
(in CSC443)



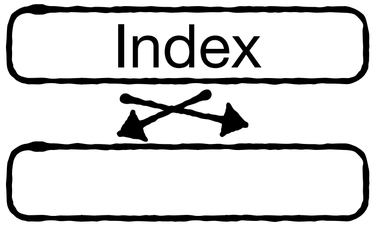
B-Trees

1970



Log-Structured Merge-Trees

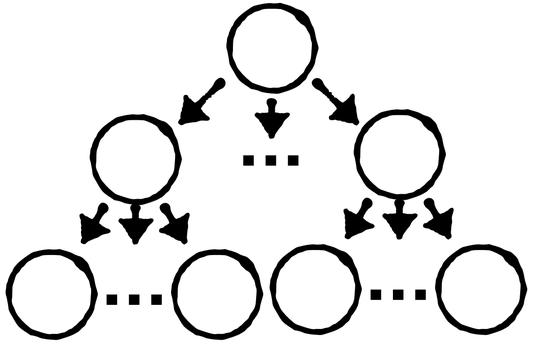
1996



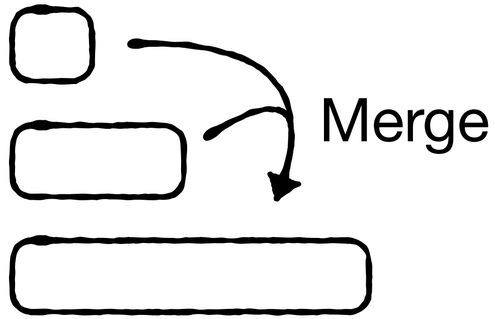
Circular Log

2000’s

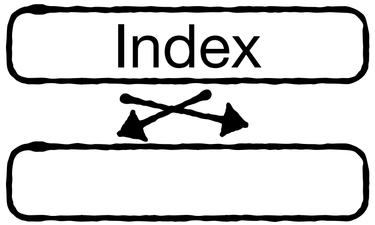
Indexes for Storage
(in CSC443)



B-Trees



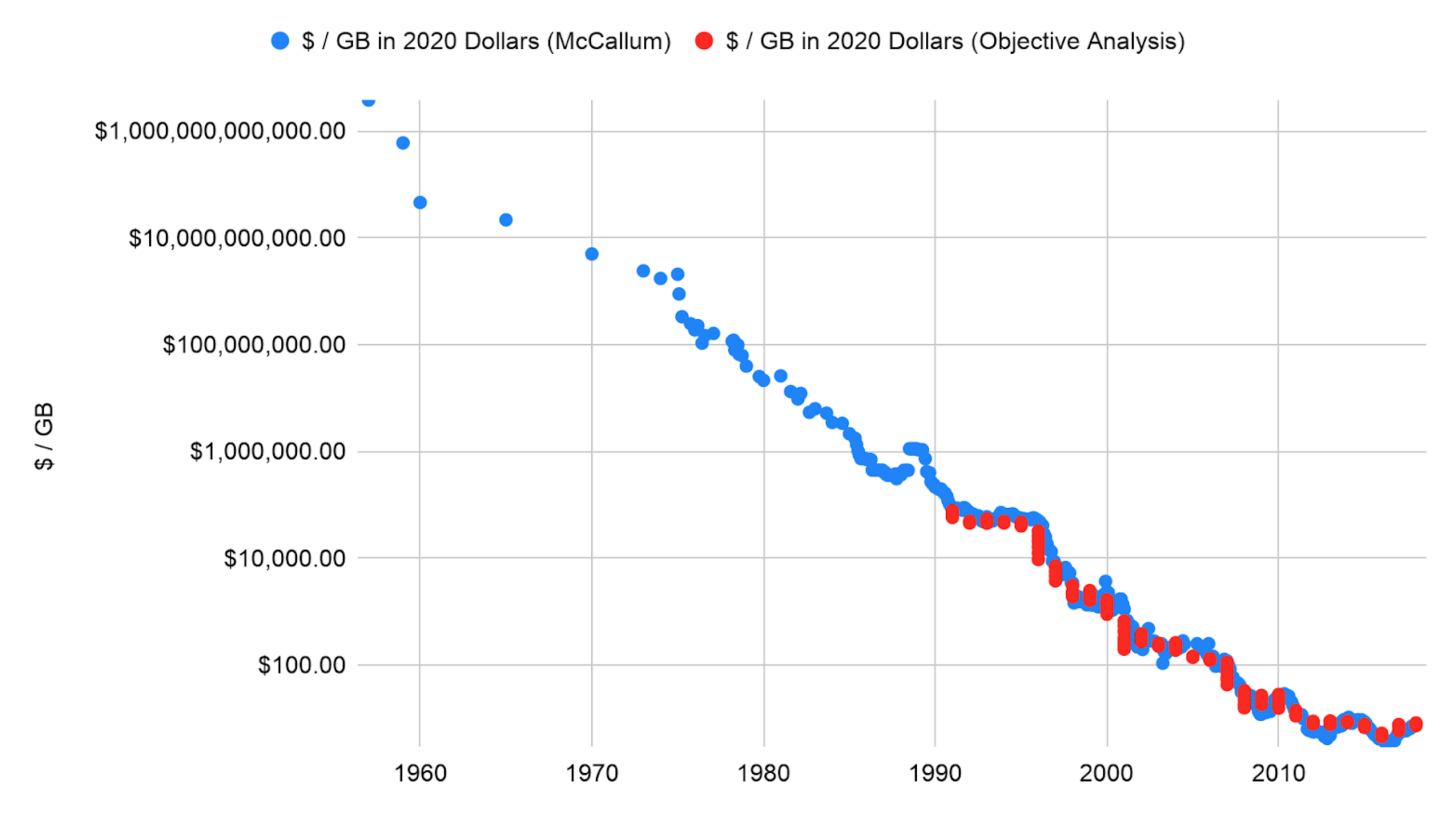
Log-Structured
Merge-Trees

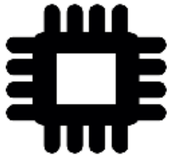


Circular Log

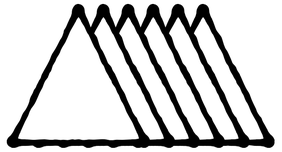
Cheaper queries

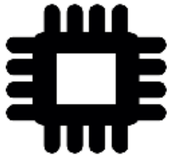
More memory
Cheaper writes


$ / 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

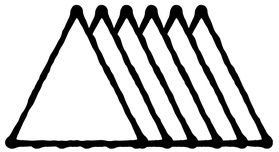



SSD

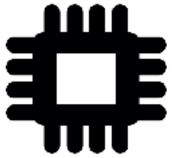




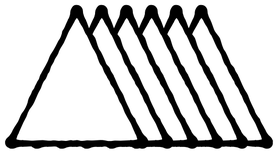
Indexes can fit here

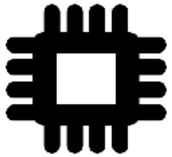


New Design Goals!

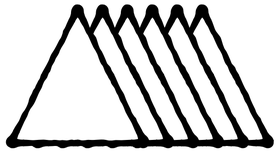


Indexes can fit here



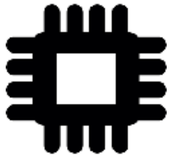


Cache miss
(≈100 cycles)



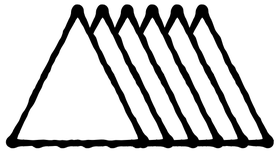
CPU register

Goal 1: Minimize Cache Misses



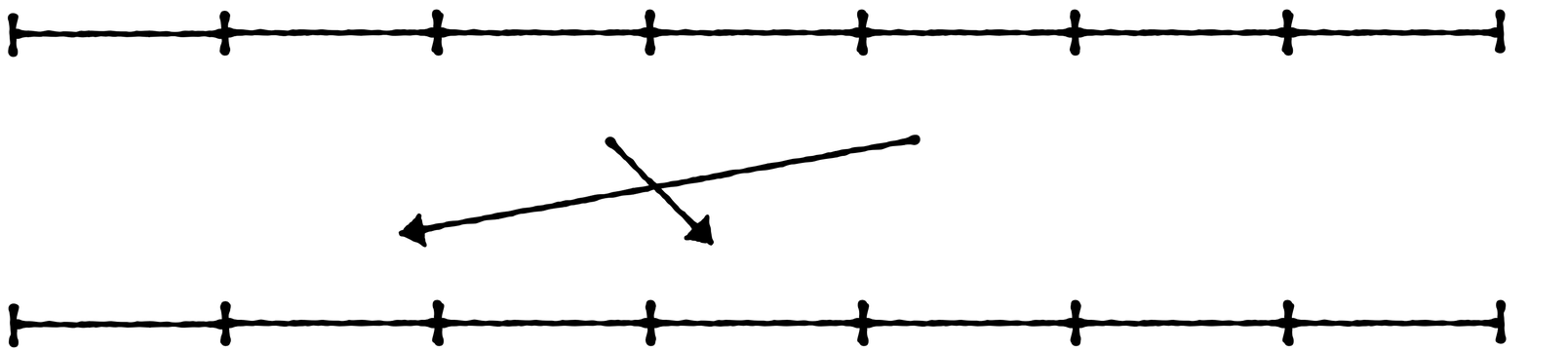
Cache miss
(≈100 cycles)

CPU register

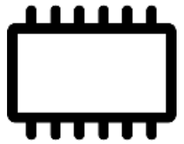



Physical DRAM 4KB pages

Virtual Memory in 4KB pages

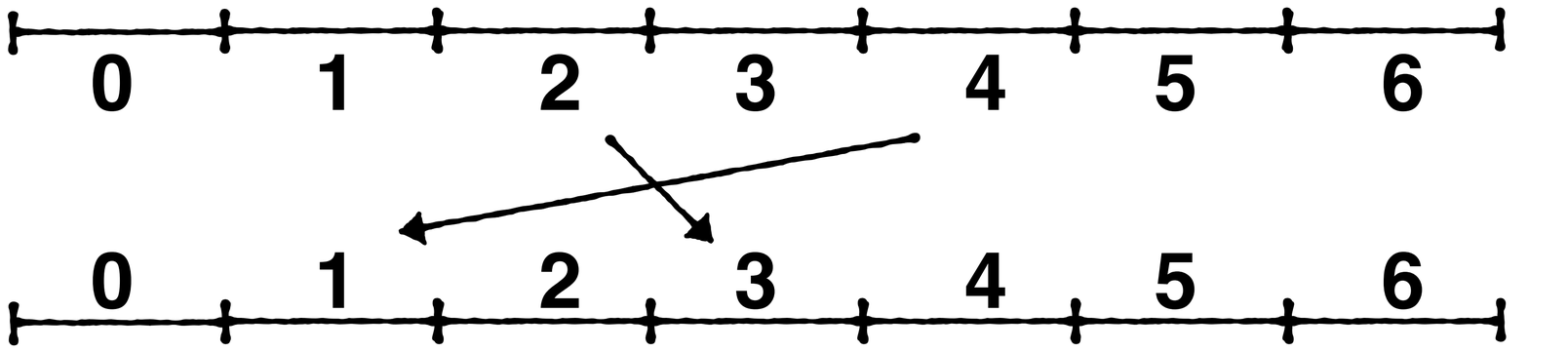


Physical DRAM 4KB pages



Virtual Memory in 4KB pages

Mapping



Physical DRAM 4KB pages

Mapping

…

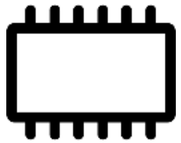
2 → **3**

…

4 → **1**

…

Stored in known location in physical DRAM



…

2 → 3

…

Translation Lookaside Buffer (TLB) in SRAM

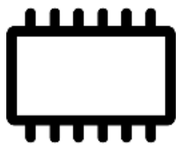
Mapping

…

4 → 1

…

Stored in known location in physical DRAM



…

2 → 3

…

Translation Lookaside Buffer (TLB) in SRAM

↑ **miss** (≈100 cycles)

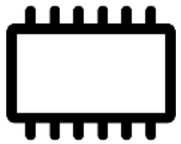
Mapping

…

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)

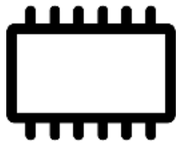
Mapping

…

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 64B, key is 4B, and a pointer is 4B

B = 8

Terms

N: # entries in dataset

B: # entries per cache line

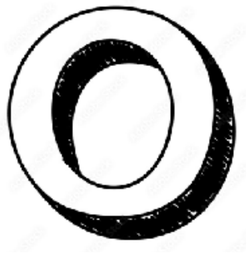
Assume a cache line is 64B, key is 4B, and a pointer is 4B

B = 8

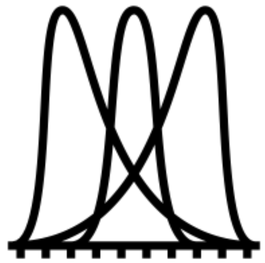
Block size is far smaller than in disk access model

Improvements Along Two Areas

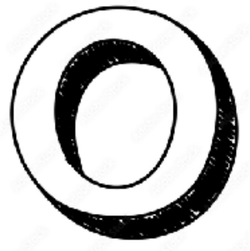
Better Worst-Case



Exploiting Data Distribution



Better Worst-Case



AVL-Tree
1962

B-Tree
1970

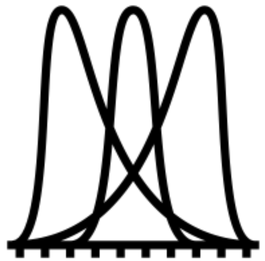
T-Tree
1985

CSS-Tree
1998

CSB-Tree
2000

ART
2013

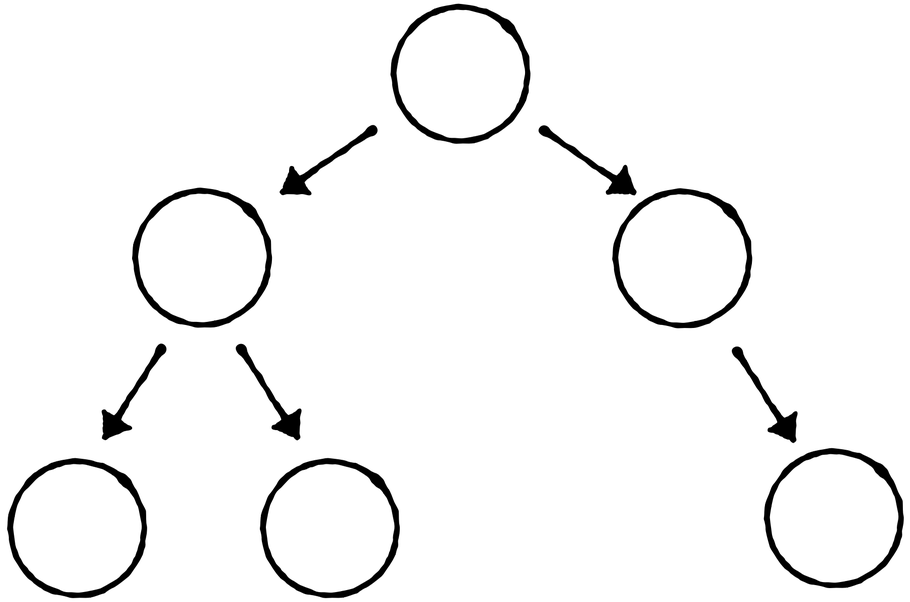
Exploiting Data Distribution



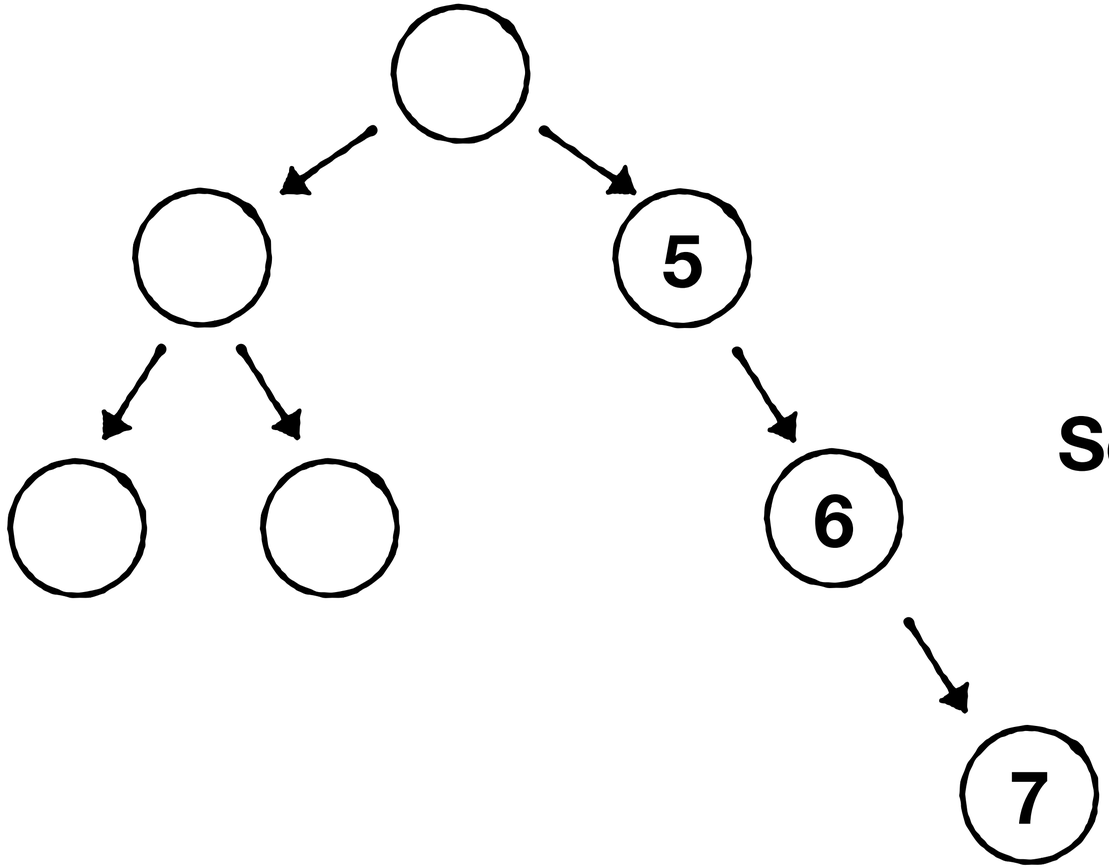
Interpolation search
1959

FITing-Tree
2019

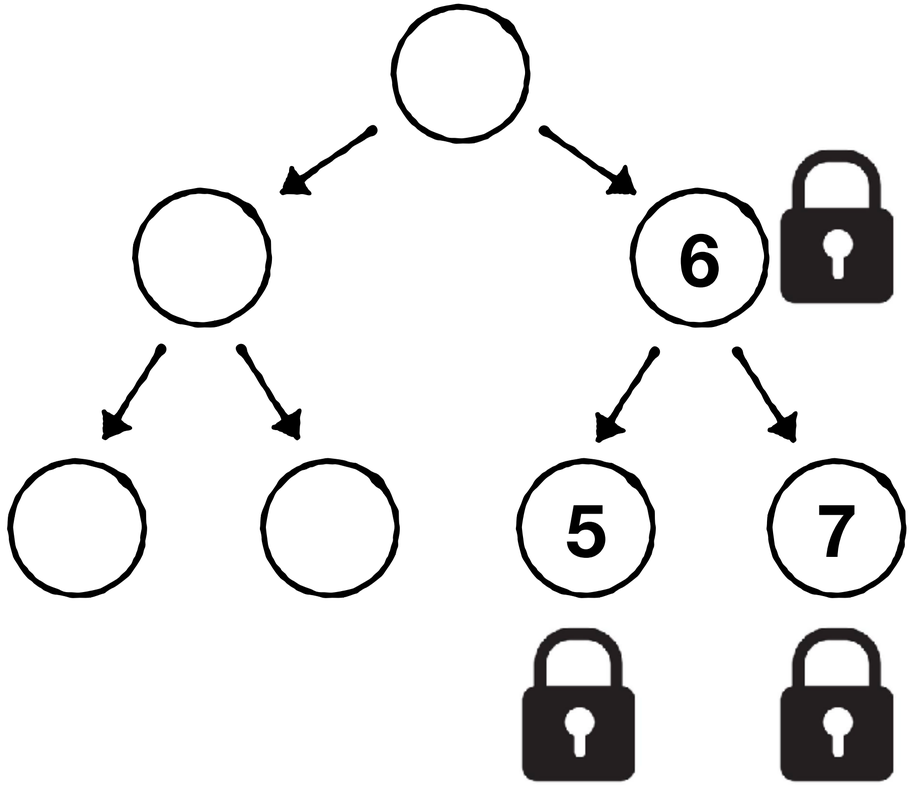
AVL-Tree



AVL-Tree

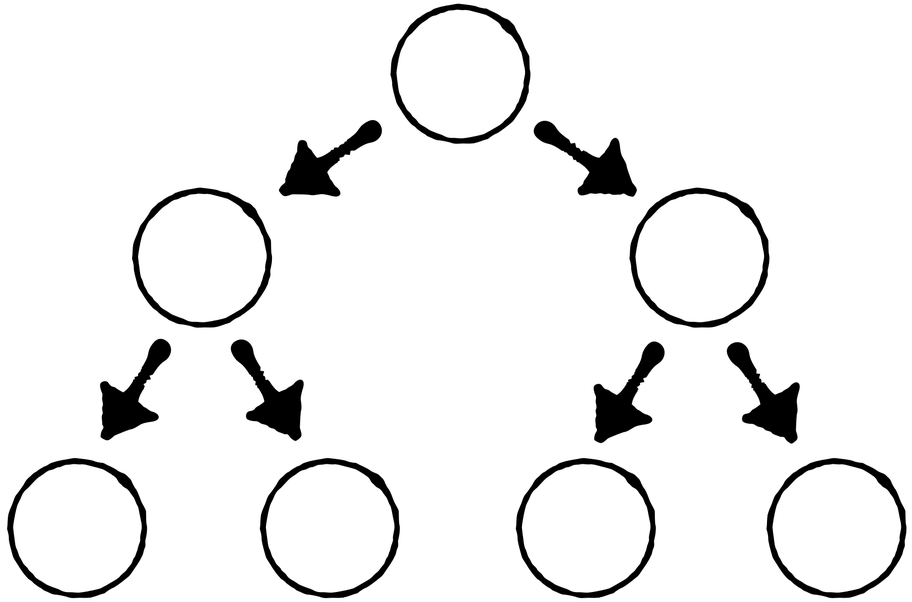


Self-balance on inserts

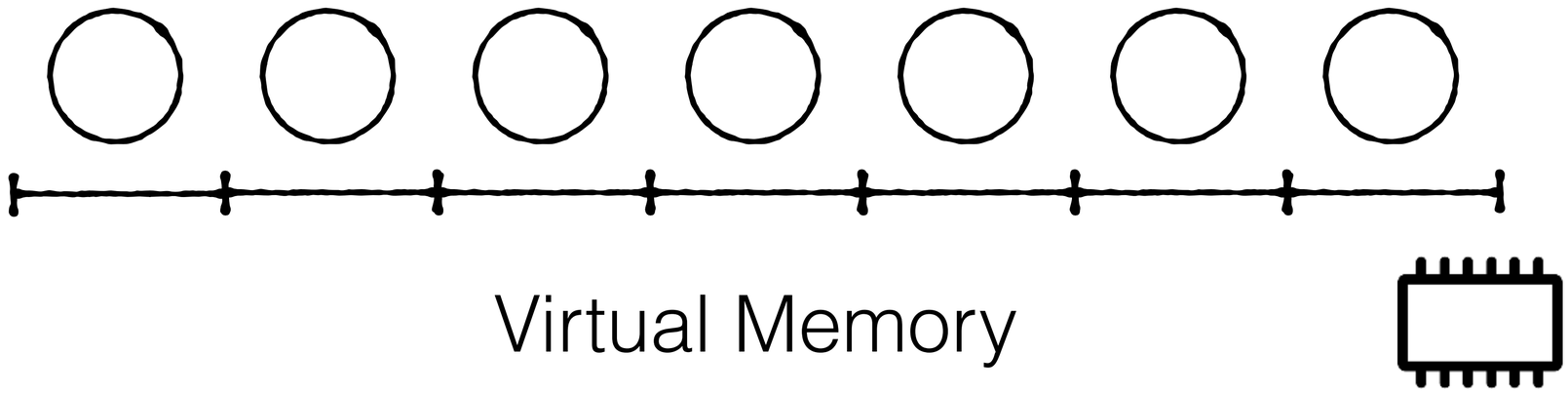


Requires holding multiple latches (damages concurrency)

Pointers create >3X metadata overhead

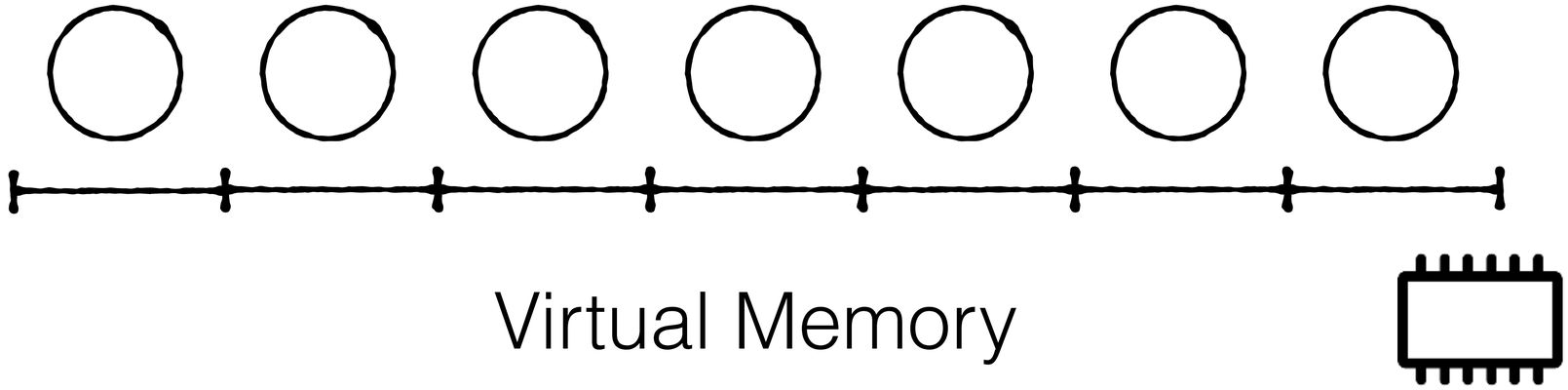


Each node may be on a different virtual page

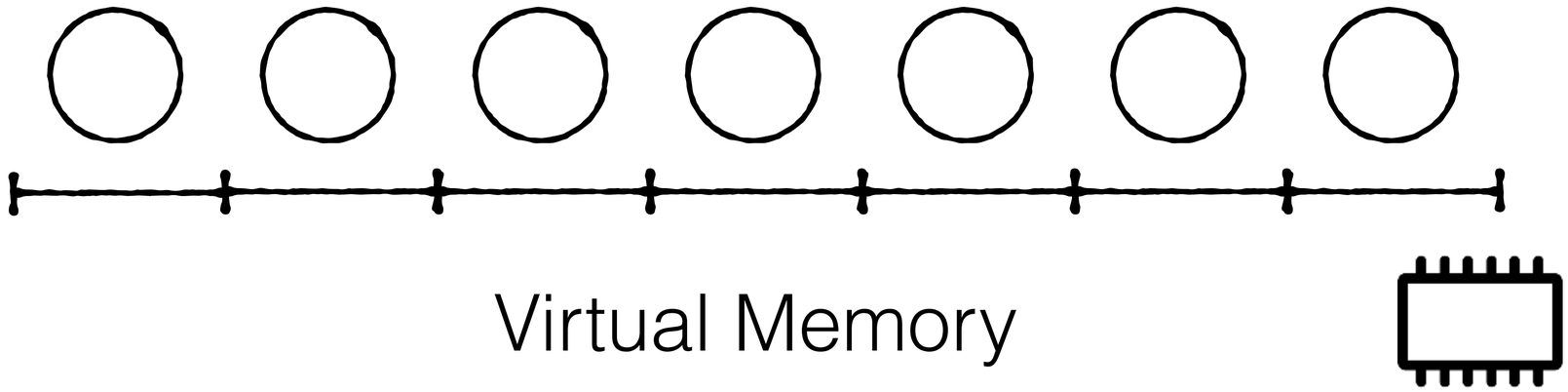


Each node may be on a different virtual page

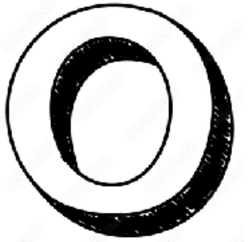
Worst case 1 TLB miss and 1 cache miss per node (≈200 cycles)



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

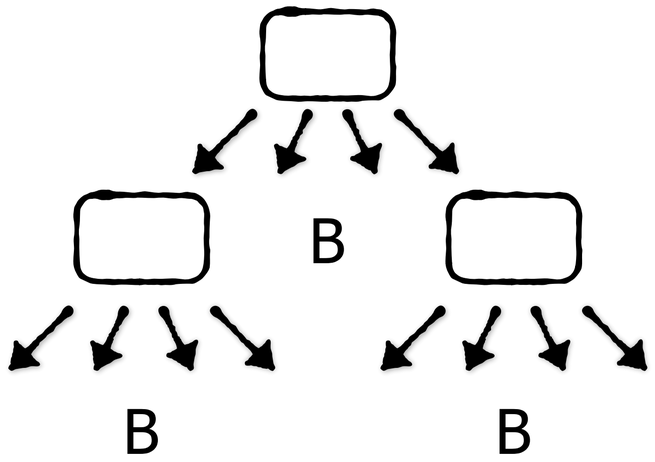


Better Worst-Case

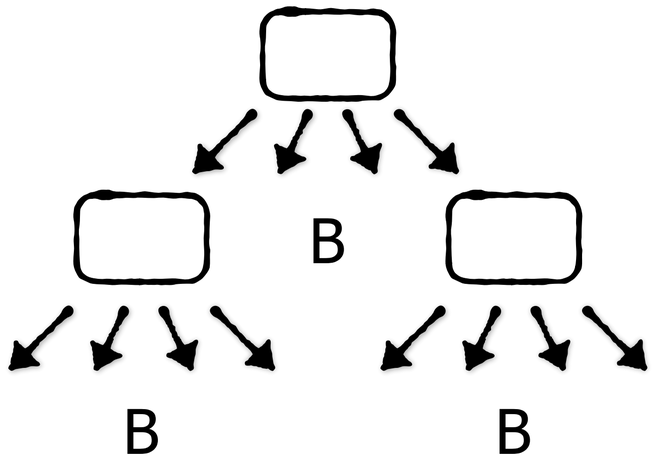


AVL-Tree	**B-Tree**	T-Tree	CSS-Tree	CSB-Tree	HOT
1962	1970	1985	1998	2000	2018

B-Tree

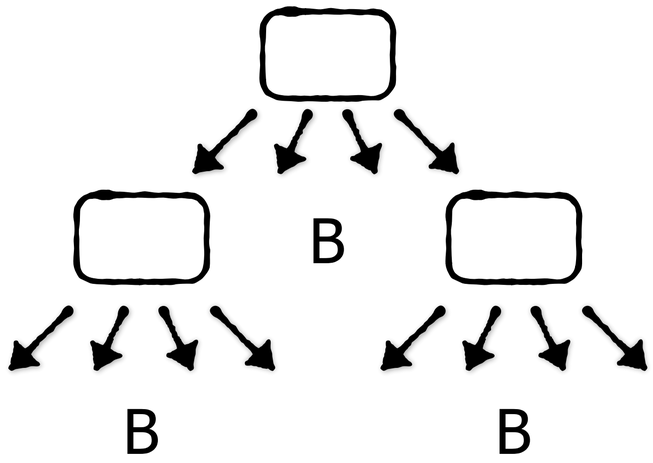


B-Tree



Set each node to be the size of a cache line

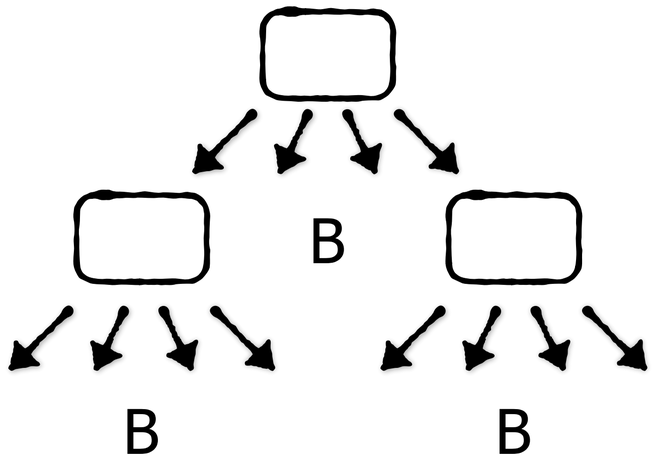
B-Tree



Set each node to be the size of a cache line

We expect $O(\log_B N)$

B-Tree

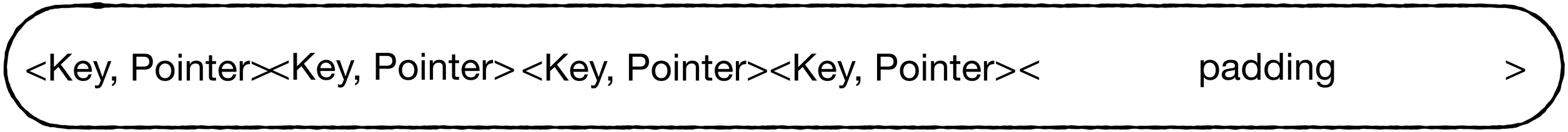


Set each node to be the size of a cache line

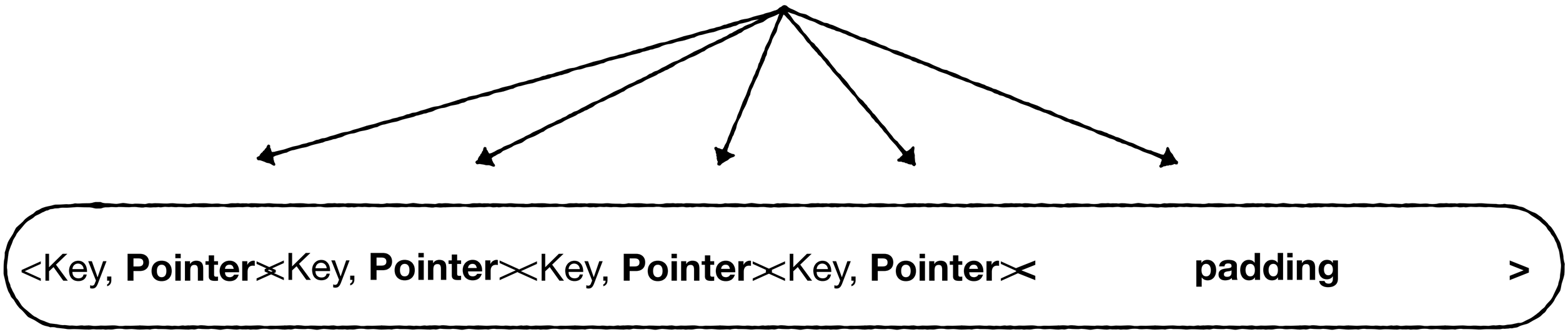
We expect $O(\log_B N)$

Do we get it? Why or why not?

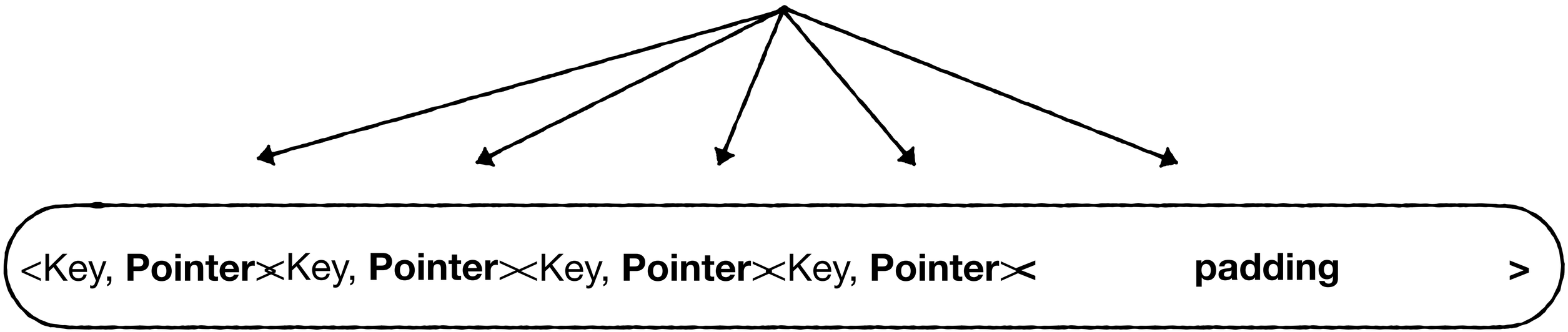
B-trees Page Organization



Space overheads

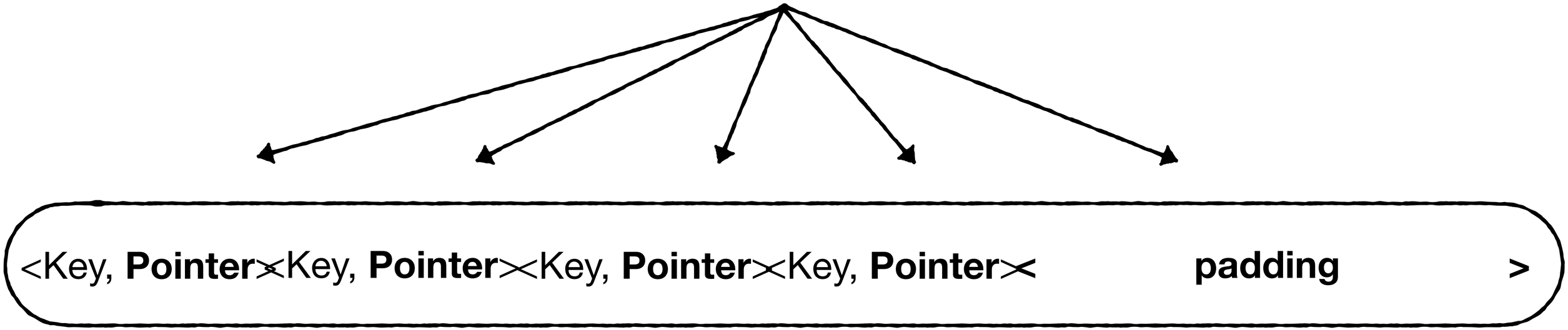


Space overheads



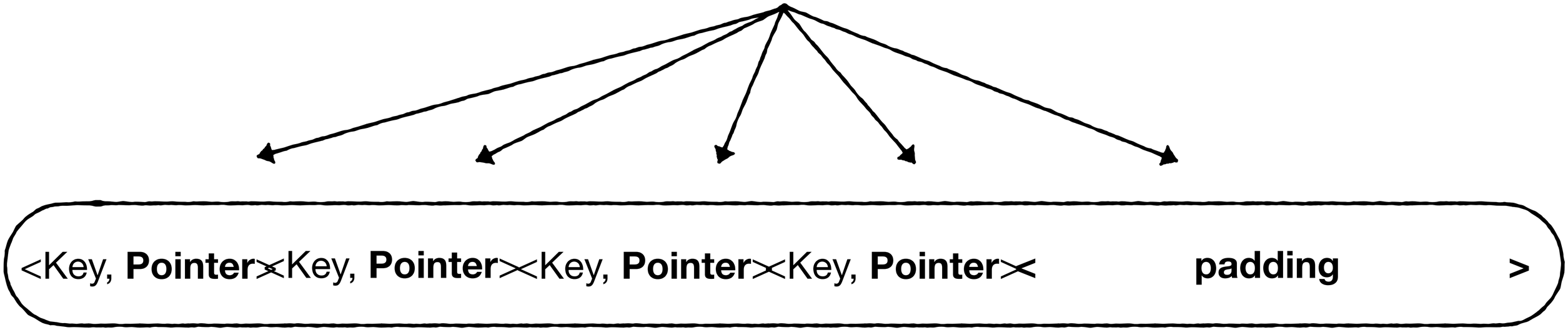
Real fanout: B/4 **(e.g., 2 rather than 8)**

Space overheads



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

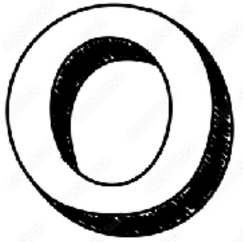
Space overheads



log $_{8/4}$ (N) cache misses

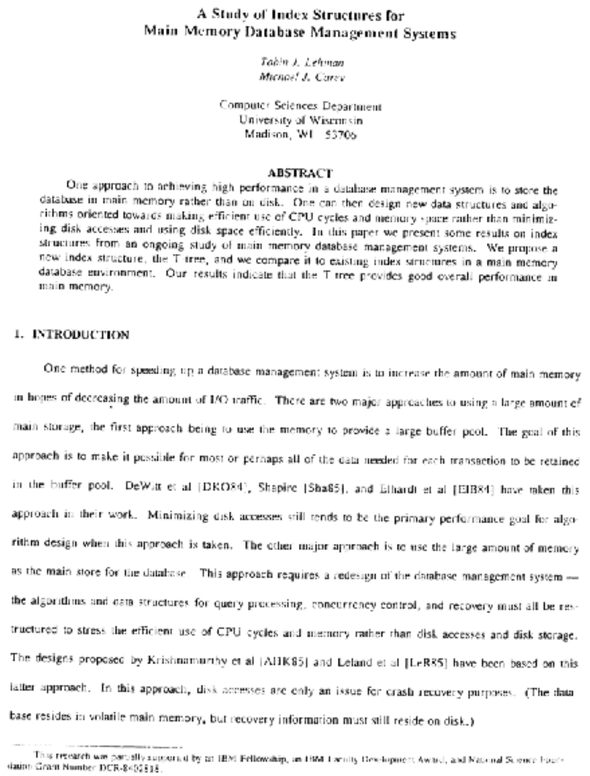
Space overheads also harm scans

Better Worst-Case



AVL-Tree	B-Tree	**T-Tree**	CSS-Tree	CSB-Tree	ART
1962	1970	1985	1998	2000	2013

T-Tree



A Study of Index Structures for
Main Memory Database Management Systems

Tobin J. Lehman
Michael J. Carey

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

ABSTRACT

One approach to achieving high performance in a database management system is to store the database in main memory rather than on disk. One can then design new data structures and algorithms oriented towards making efficient use of CPU cycles and memory space rather than minimizing disk accesses and using disk space efficiently. In this paper we present some results on index structures from an ongoing study of main memory database management systems. We propose a new index structure, the T tree, and we compare it to existing index structures in a main memory database environment. Our results indicate that the T tree provides good overall performance in main memory.

1. INTRODUCTION

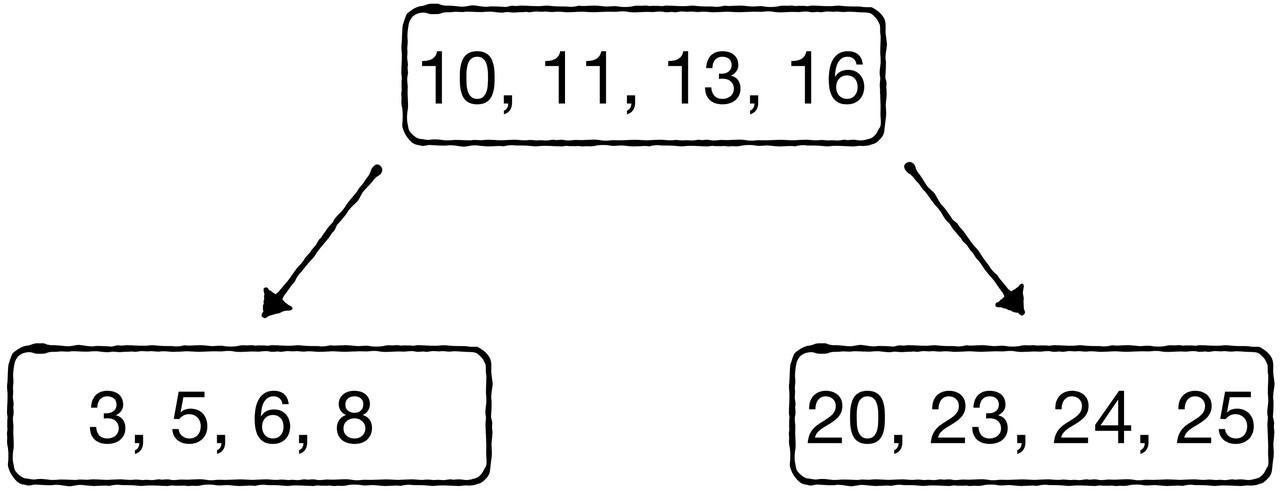
One method for speeding up a database management system is to increase the amount of main memory in hopes of decreasing the amount of I/O traffic. There are two major approaches to using a large amount of main storage, the first approach being to use the memory to provide a large buffer pool. The goal of this approach is to make it possible for most or perhaps all of the data needed for each transaction to be retained in the buffer pool. DeWitt et al [DKO84], Shapiro [Sha85], and Eliardt et al [EIB84] have taken this approach in their work. Minimizing disk accesses still tends to be the primary performance goal for algorithm design when this approach is taken. The other major approach is to use the large amount of memory as the main store for the database. This approach requires a redesign of the database management system — the algorithms and data structures for query processing, concurrency control, and recovery must all be restructured to stress the efficient use of CPU cycles and memory rather than disk accesses and disk storage. The designs proposed by Krishnamurthy et al [AHK85] and Leland et al [LeR85] have been based on this latter approach. In this approach, disk accesses are only an issue for crash recovery purposes. (The database resides in volatile main memory, but recovery information must still reside on disk.)

This research was partially supported by an IBM Fellowship, an IBM Faculty Development Award, and National Science Foundation Grant Number DCR-8402818.

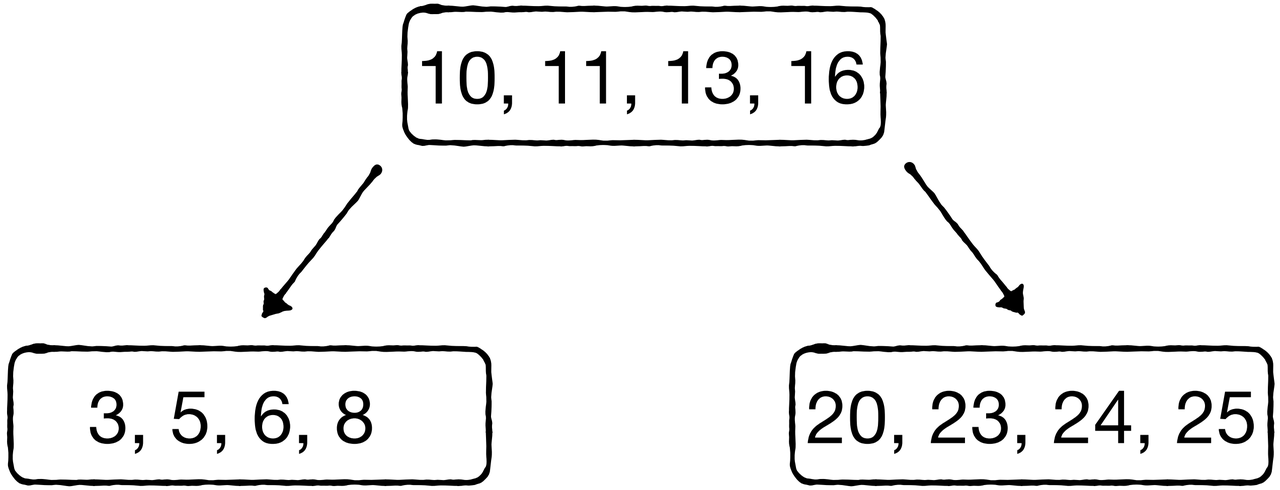
A study of index structures for main memory database management systems

Tobin J. Lehman Michael J. Carey. Technical Report. 1985

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

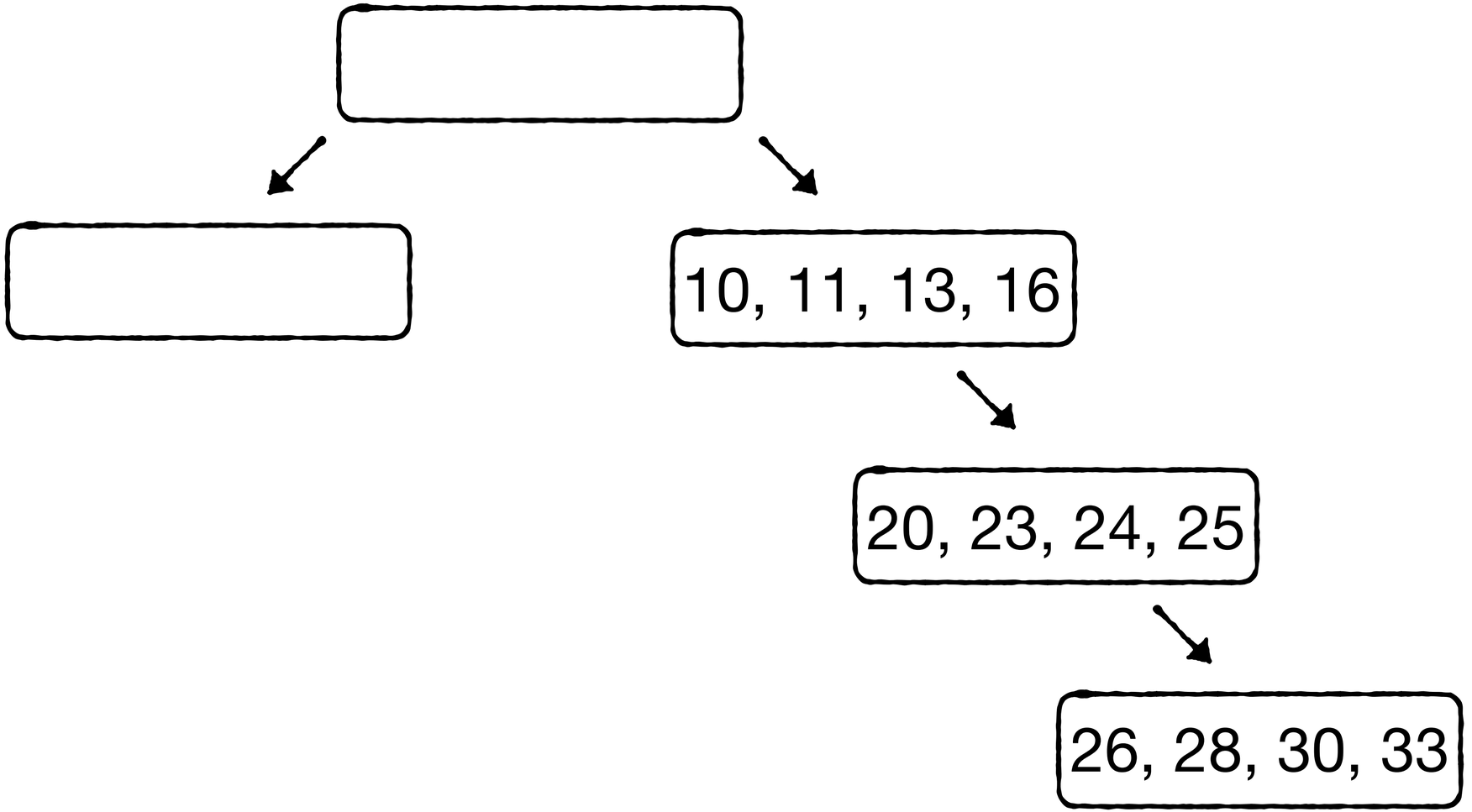


T-Tree: an AVL-Tree with Fat Nodes

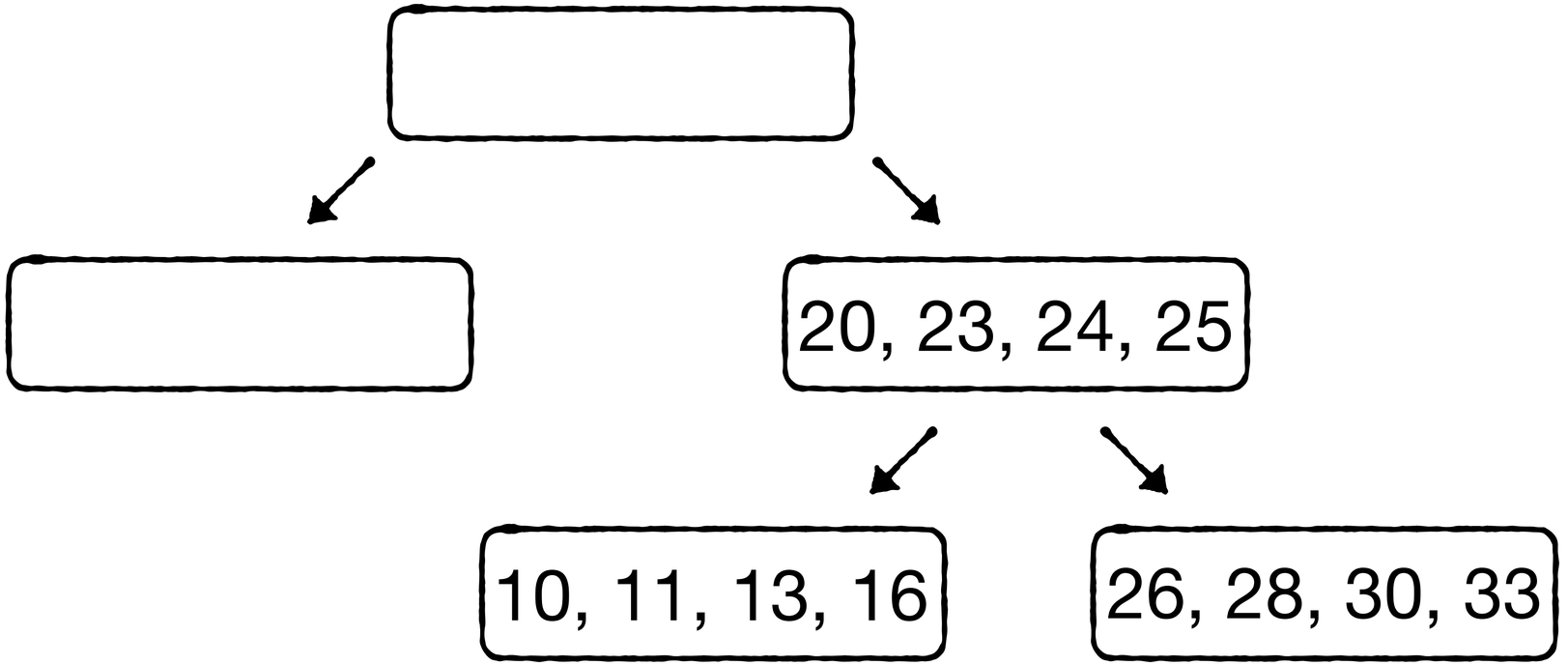


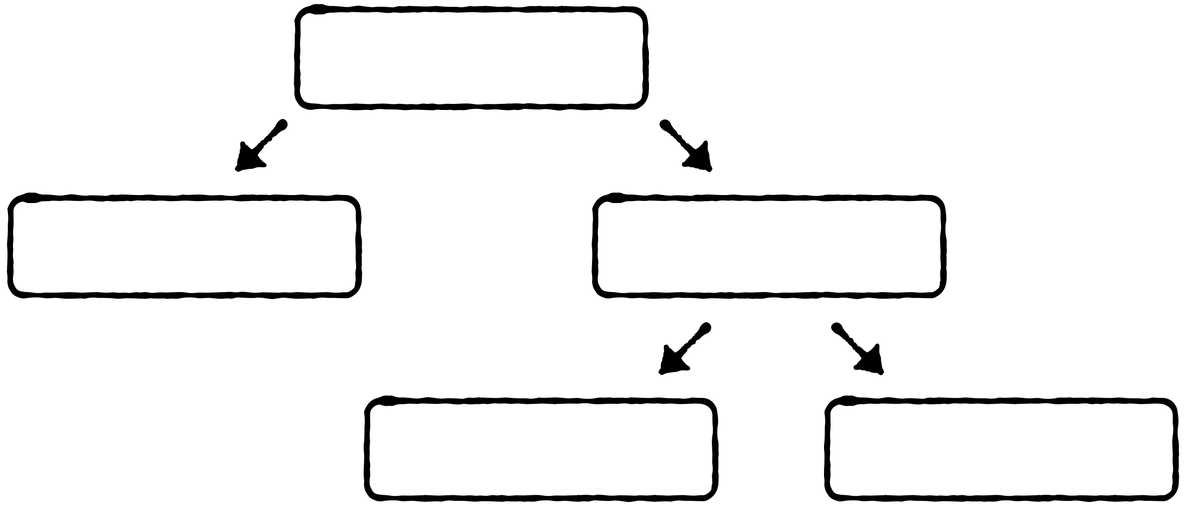
A node can’t intersect with its sub-trees in key range

Self-Balancing is Identical to AVL-Tree

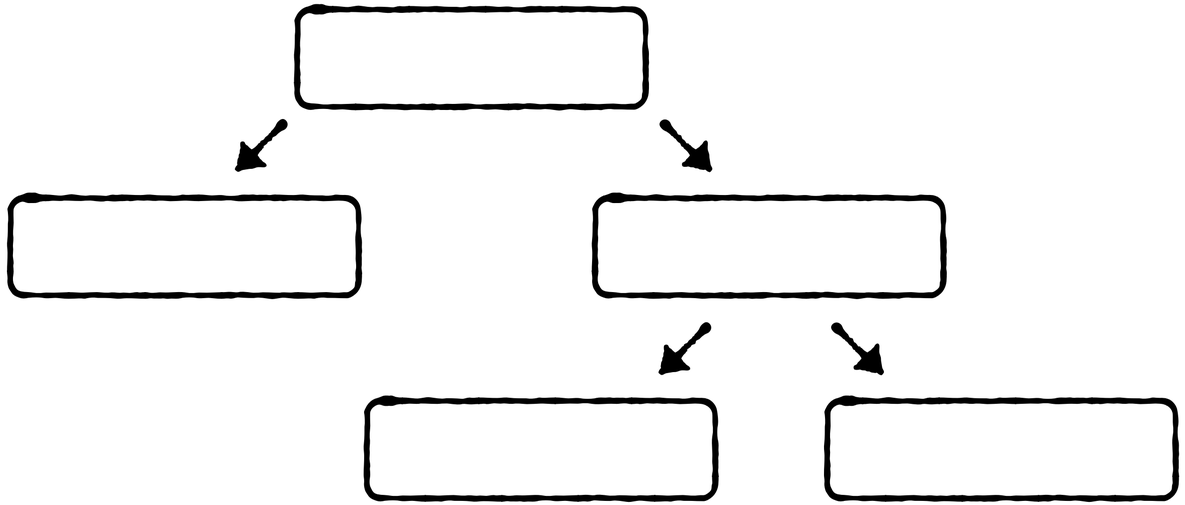


Self-Balancing is Identical to AVL-Tree



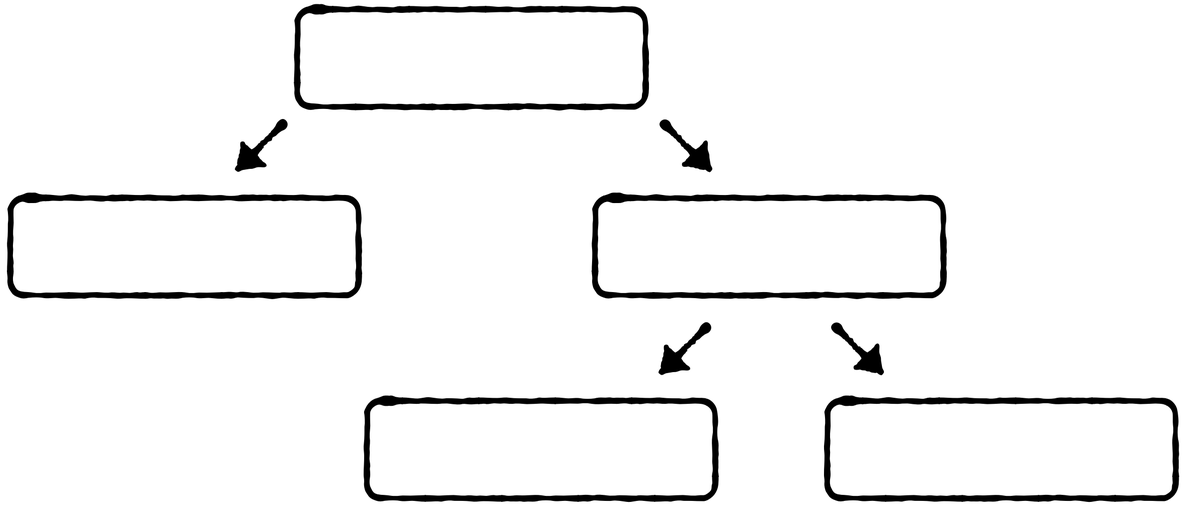


Pros: (1) Less metadata relative to data



Pros: (1) Less metadata relative to data

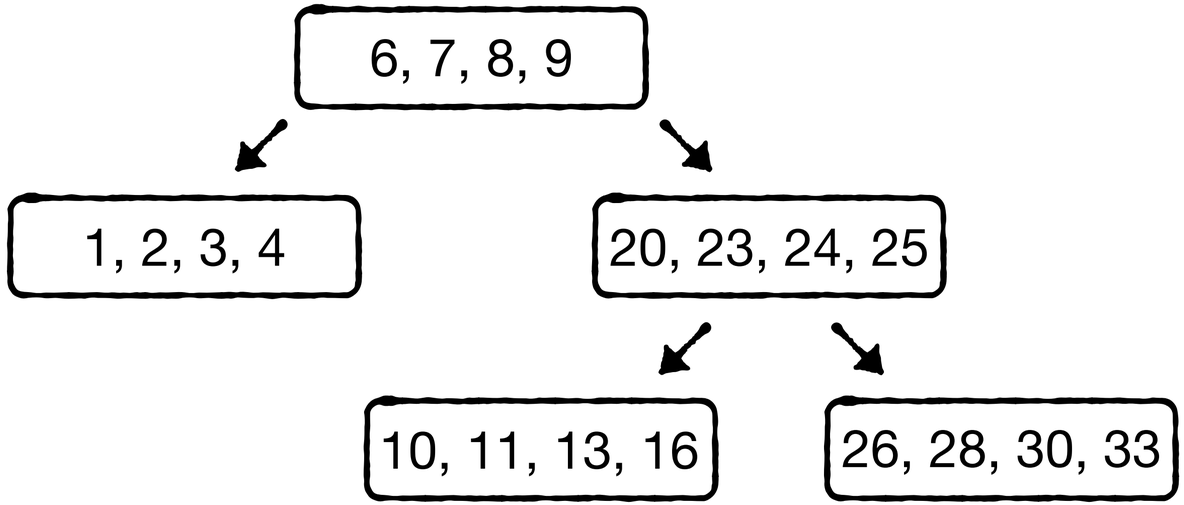
(2) Fewer rotations



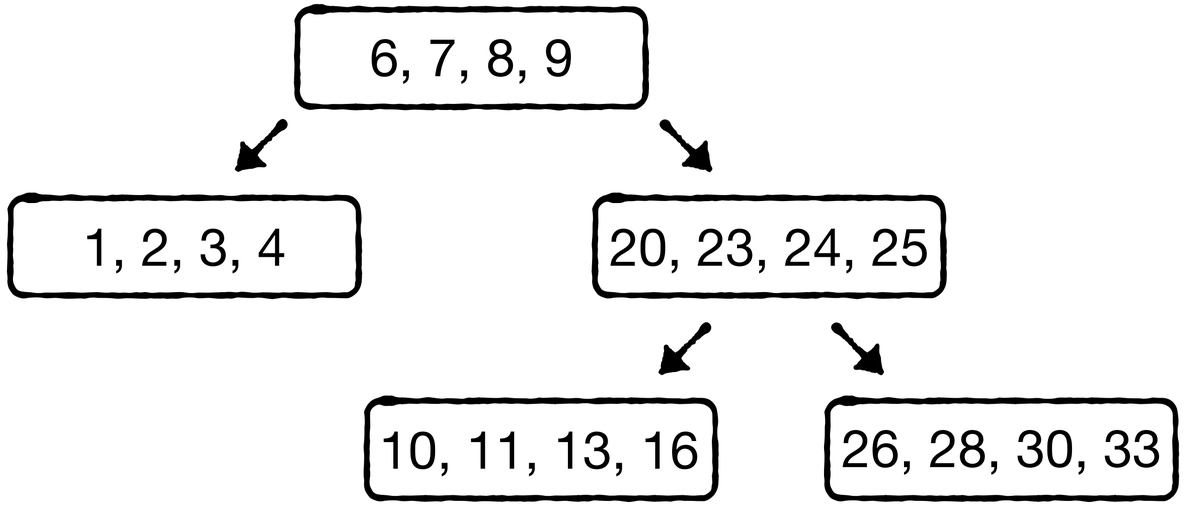
Pros: (1) Less metadata relative to data

(2) Fewer rotations → **less locking**

search cost? (In terms of B and N)

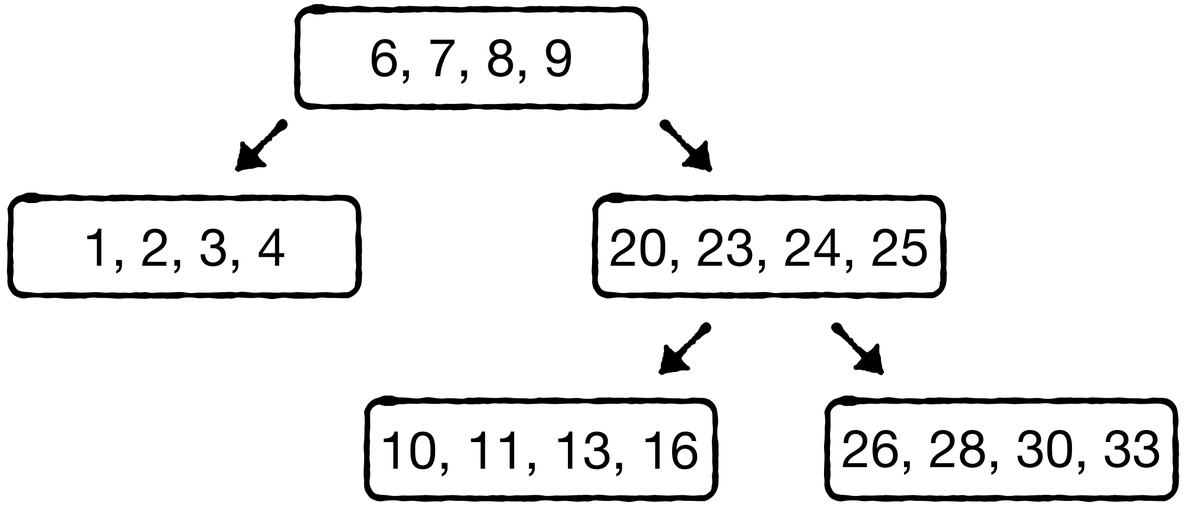


search cost? (In terms of B and N)



We only prune the search space by 2x per node

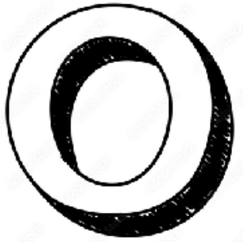
search cost? (In terms of B and N)



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	ART
1962	1970	1985	1998	2000	2013

Cache-Sensitive Search-Tree (CSS-Tree)

Cache-Sensitive Search-Tree (CSS-Tree)

1 Introduction

As random access memory gets cheaper it becomes increasingly affordable to build computers with large main memories. It is possible to configure machines with gigabytes of main memory for just a few thousand dollars. Thus, one begins to ask whether some data intensive applications such as decision-support query processing can take advantage of the speed of main memory to provide rapid answers to complex queries. In this paper we work with databases that fit entirely into main memory. There are many applications with large datasets of the order of several gigabytes for which this limitation is feasible [GMS92].

Index structures are important even in main memory database systems. Although there are no more disk accesses, indexes can be used to reduce overall computation time without using too much extra space. Past work on measuring the performance of indexing in main-memory databases includes [LC86b, WK90], with [LC86b] being the most comprehensive on the specific issue of indexing. In the twelve years since [LC86b] was published, there have been substantial changes in the architecture of computer chips. The most relevant change is that CPU speeds have been increasing at a much faster rate (60% per year) than memory speeds (10% per year) as shown in Figure 1 (borrowed from [CLH98]). Thus the *relative* cost of a cache miss has increased by two orders of magnitude since 1986. As a result, we cannot assume that the ranking of indexing algorithms given in [LC86b] would be valid on today's architectures. In fact, our experimental results indicate very different relative outcomes from [LC86b] for lookup speed.

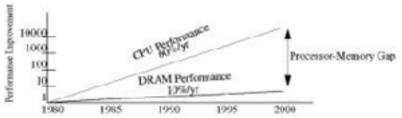


Figure 1: Processor-memory performance imbalance

A second recent development has been the explosion of interest in On-Line Analytical Processing (OLAP). OLAP workloads are query-intensive, and have infrequent batch updates. In an OLAP context, it is more important to optimize query performance than update performance. In a main-memory system, it may be relatively cheap to rebuild an index from scratch after a batch of updates. With OLAP as our focus, we can design algorithms for query performance, perhaps at the expense of update performance.

Two important criteria for the selection of index structures are space and time. Space is critical in a main memory database and we may have a limited amount of space available for precomputed structures such as indexes. Given space constraints, we try to optimize the time taken by index lookups. In a main-memory database there are several factors influencing the speed of database operations. An important factor is the degree of locality in data references for a given algorithm. Good data locality leads to fewer (expensive) cache misses and better performance.

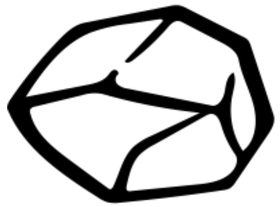
We study a variety of existing techniques, including hash indexes, binary search on a sorted list of record identifiers, binary trees, B+-trees [Com79], T-trees [LC86a], and interpolation search. We also introduce a new technique called "Cache-Sensitive Search Trees" (CSS-trees). CSS-trees augment binary search by storing a directory structure on top of the sorted list of elements. CSS-trees differ from B+-trees by avoiding storing the child pointers in each node. The CSS-tree is organized in such a way that traversing the tree yields good data reference locality (unlike binary search), and hence relatively few cache misses.

We measure the time and space requirements of each algorithm. When range queries or sequential access are needed on an attribute, we keep a list of record-identifiers that is sorted by that attribute. (If the underlying table is clustered by that attribute, then such a list is not necessary.) Our major conclusions are as follows:

- Hash indices have a high space overhead (the hash table is at least as large as the list of record-

Cache Conscious Indexing for Decision-Support in Main Memory
Jun Rao, Kenneth A. Ross. VLDB 1999.

Cache-Sensitive Search-Tree (CSS-Tree)



static data

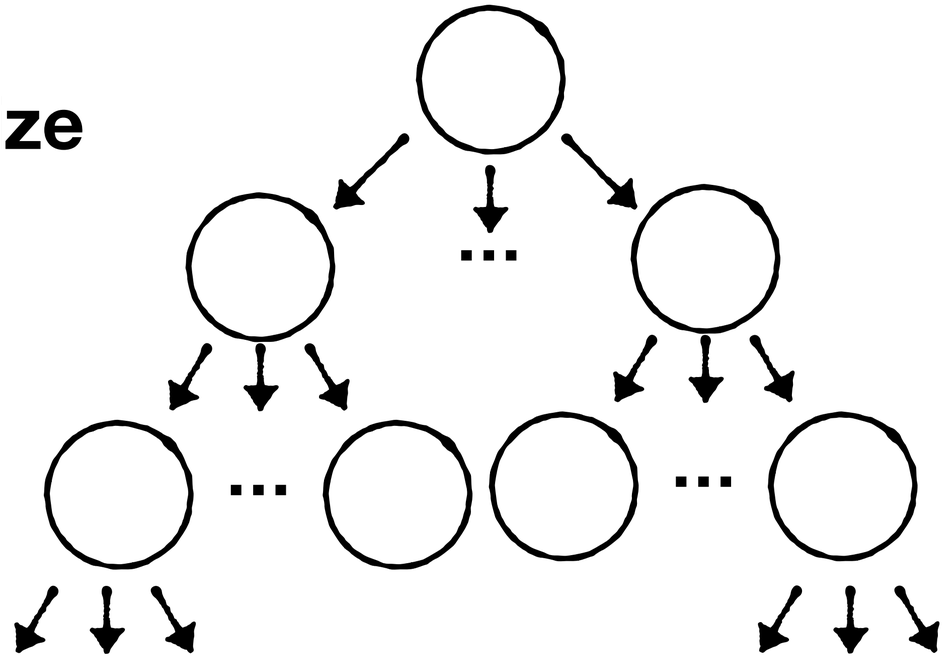
Cache Conscious Indexing for Decision-Support in Main Memory
Jun Rao, Kenneth A. Ross. VLDB 1999.

Cache-Sensitive Search-Tree (CSS-Tree)

static sorted array

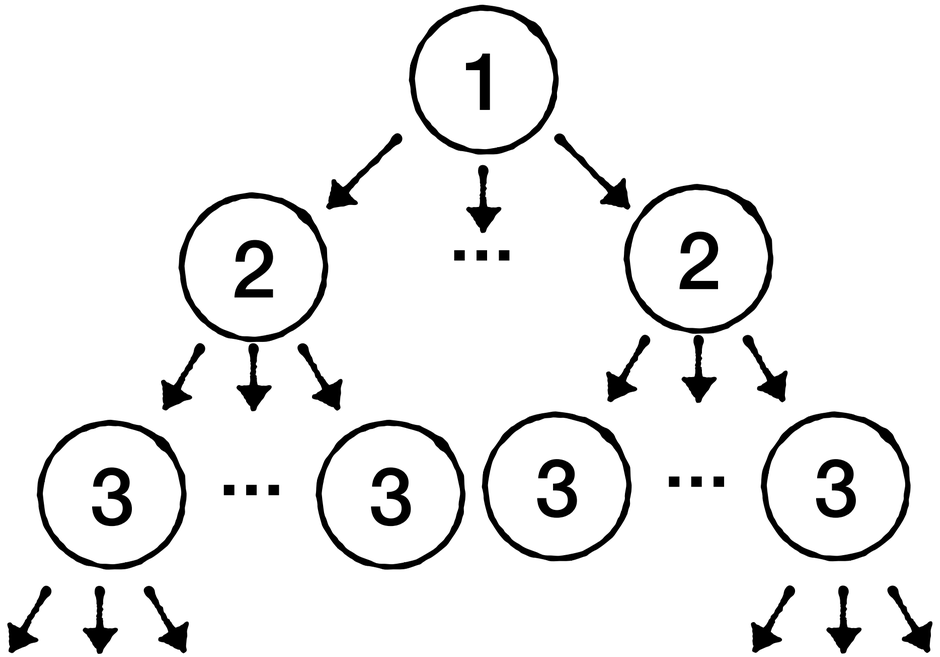
Cache-Sensitive Search-Tree (CSS-Tree)

Each node the size of a cache line



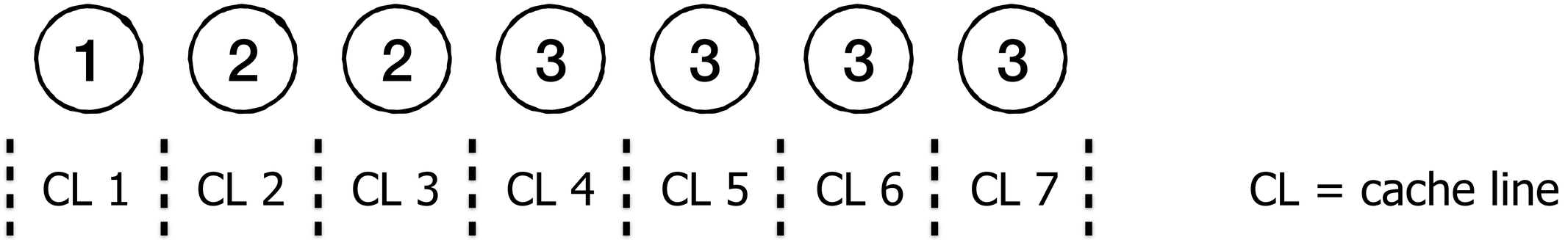
static sorted array

Cache-Sensitive Search-Tree (CSS-Tree)



static sorted array

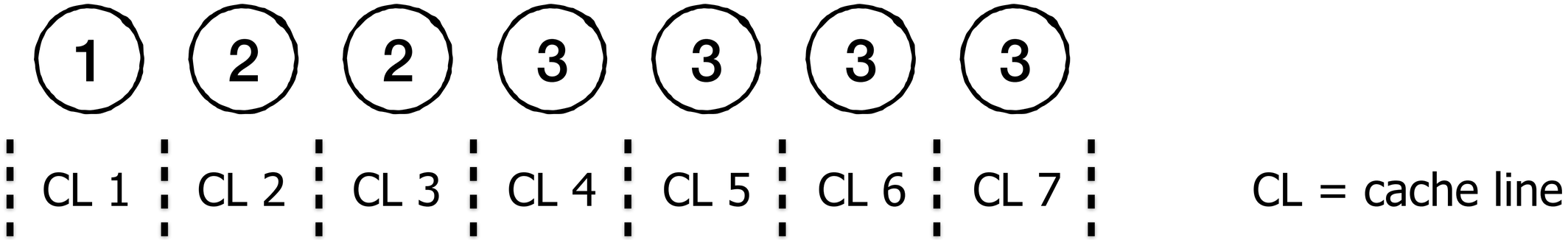
Stored as dense array in breadth-first order



static sorted array

Stored as dense array in breadth-first order

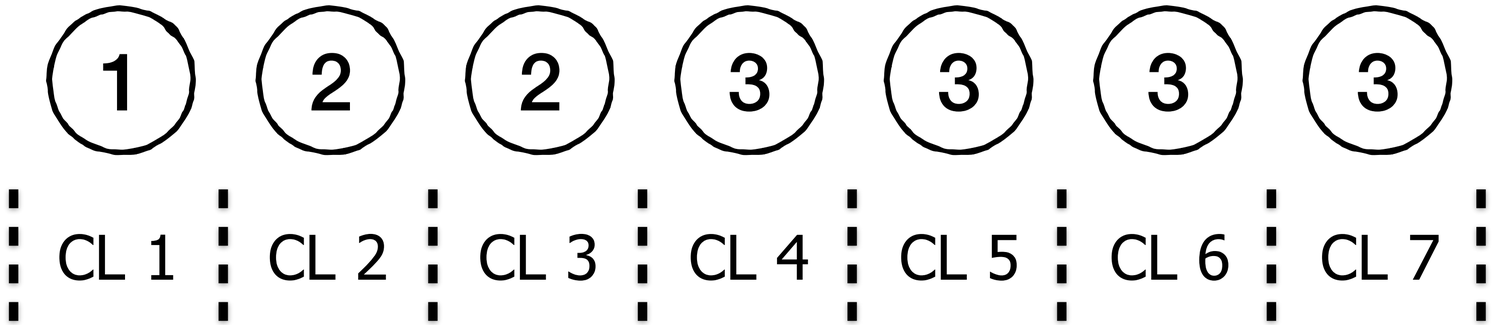
No padding or pointers among nodes (arithmetic is used to find child)



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

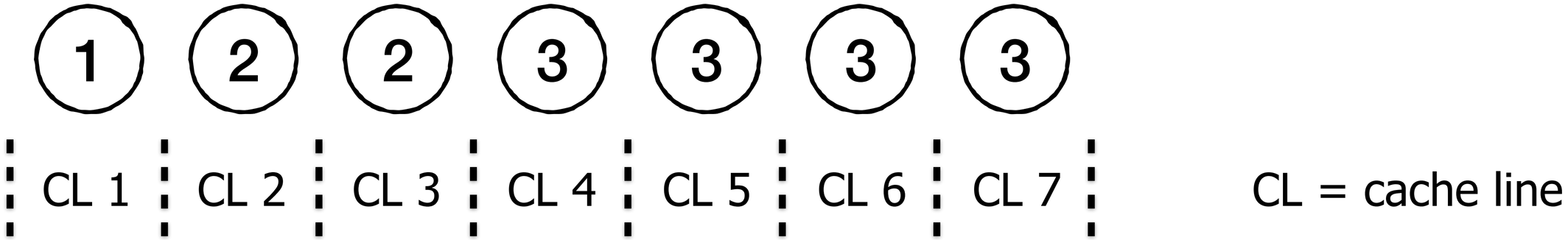


CL = cache line

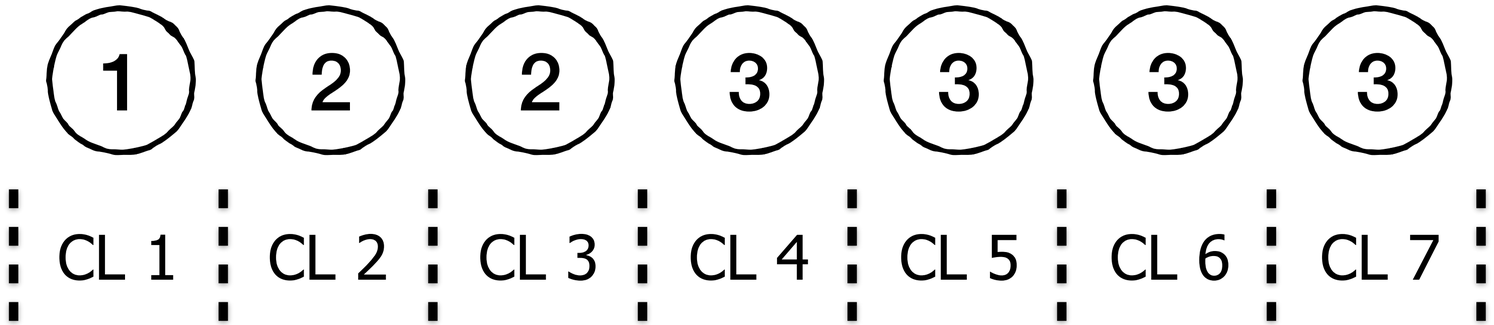
Stored as dense array in breadth-first order

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

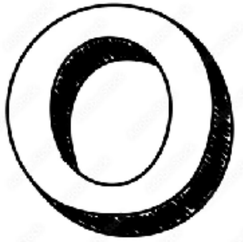
$\text{Log}_8\, N$ Cache misses



Requires full reconstruction to handle updates



Better Worst-Case



AVL-Tree 1962

B-Tree 1970

T-Tree 1985

CSS-Tree 1998

CSB-Tree 2000

ART 2013

Cache-Sensitive B-Tree (CSB-Tree)

Making B+-Trees Cache Conscious in Main Memory

Jun Rao
Columbia University
jun@cs.columbia.edu

Kenneth A. Ross*
Columbia University
kar@cs.columbia.edu

Abstract

Previous research has shown that cache behavior is important for main memory index structures. Cache conscious index structures such as Cache Sensitive Search Trees (CSS-Trees) perform lookups much faster than binary search and T-Trees. However, CSS-Trees are designed for decision support workloads with relatively static data. Although B+-Trees are more cache conscious than binary search and T-Trees, their utilization of a cache line is low since half of the space is used to store child pointers. Nevertheless, for applications that require incremental updates, traditional B+-Trees perform well.

Our goal is to make B+-Trees as cache conscious as CSS-Trees without increasing their update cost too much. We propose a new indexing technique called "Cache Sensitive B+-Trees" (CSB+-Trees). It is a variant of B+-Trees that stores all the child nodes of any given node contiguously, and keeps only the address of the first child in each node. The rest of the children can be found by adding an offset to that address. Since only one child pointer is stored explicitly, the utilization of a cache line is high. CSB+-Trees support incremental updates in a way similar to B+-Trees.

We also introduce two variants of CSB+-Trees. Segmented CSB+-Trees divide the child nodes into segments. Nodes within the same segment are stored contiguously and only pointers to the beginning of each segment are stored explicitly in each node. Segmented CSB+-Trees can reduce the copying cost when there is a split since only one segment needs to be moved. Full CSB+-Trees preallocate space for the full node group and thus reduce the split cost. Our performance studies show that CSB+-Trees are useful for a wide range of applications.

*This research was supported by a David and Lucile Packard Foundation Fellowship in Science and Engineering, by an NSF Young Investigator Award, by NSF grant number IIS-98-12014, and by NSF CISE award CDA-9625374.

1 Introduction

Figure 1. CPU-memory Performance Imbalance

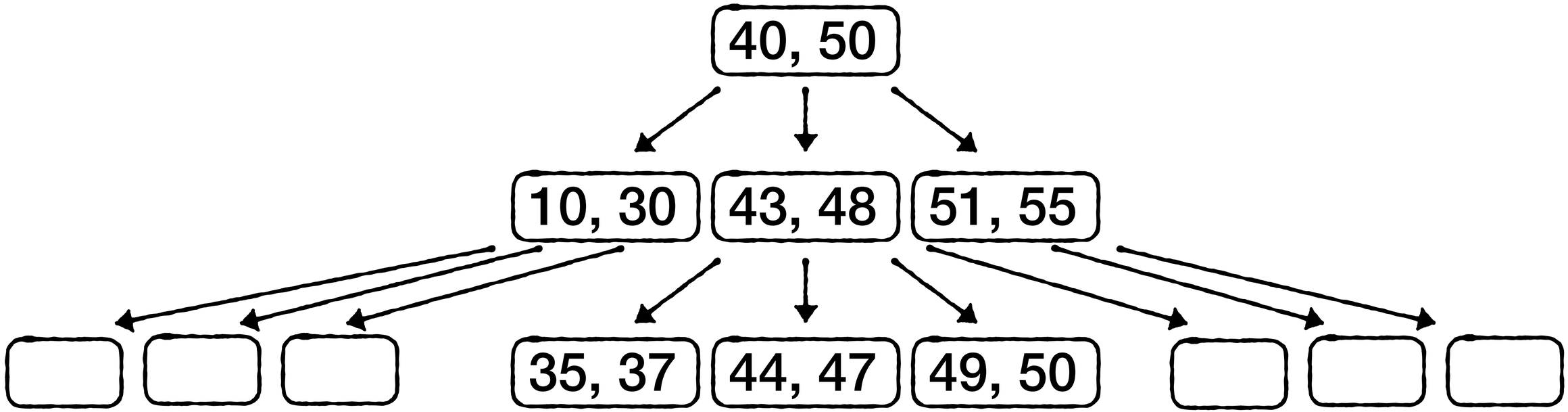
As random access memory gets cheaper, it becomes increasingly affordable to build computers with large main memories. The recent "Asilomar Report" ([BBC+98]) predicts: "Within ten years, it will be common to have a terabyte of main memory serving as a buffer pool for a hundred-terabyte database. All but the largest database tables will be resident in main memory." But main memory data processing is not as simple as increasing the buffer pool size. An important issue is cache behavior. The traditional assumption that memory references have uniform cost is no longer valid given the current speed gap between cache access and main memory access. [ADW99] studied the performance of several commercial database management systems in main memory. The conclusions they reached is that a significant portion of execution time is spent on second level data cache misses and first level instruction cache misses. Further more, CPU speeds have been increasing at a much faster rate (60% per year) than memory speeds (10% per year) as shown in Figure 1. So, improving cache behavior is going to be an imperative task in main memory data processing.

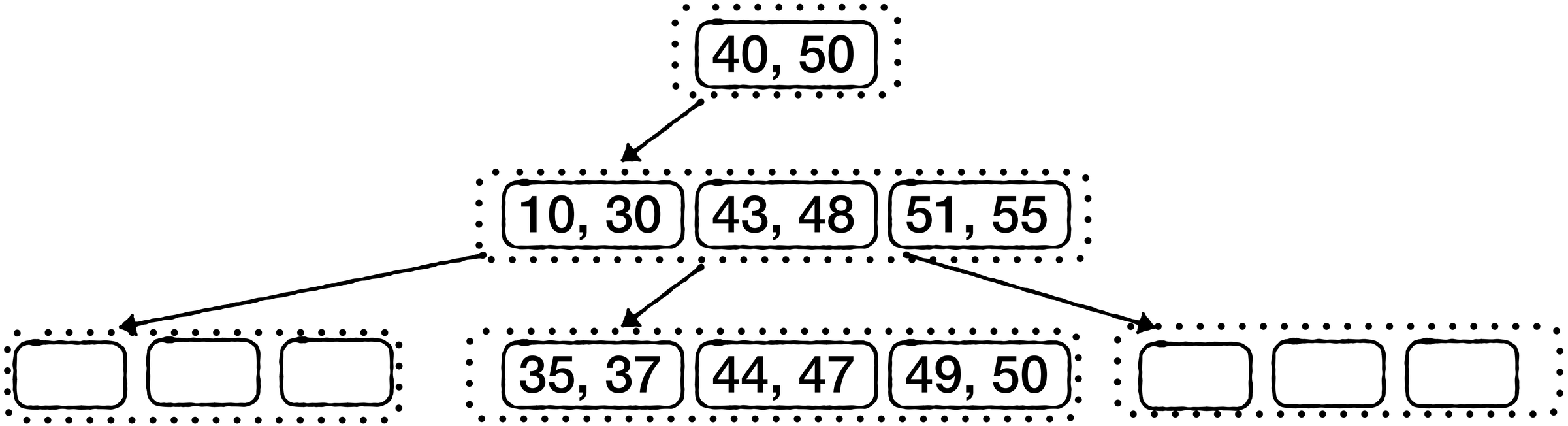
Index structures are important even in main

Making B+trees cache conscious in main memory

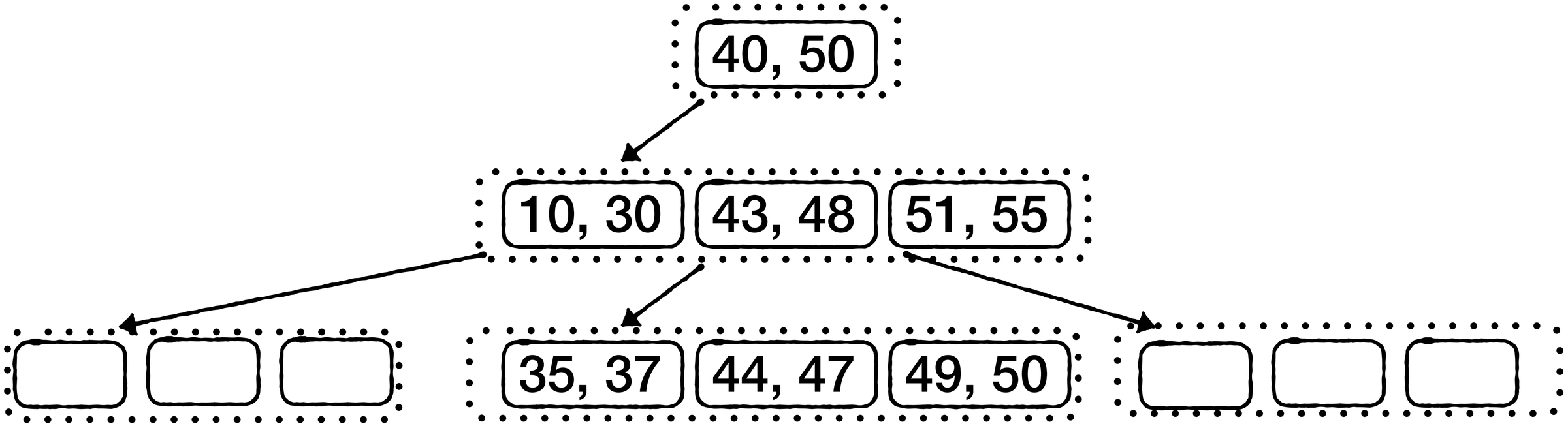
Jun Rao, Kenneth A. Ross SIGMOD 2000

Cache-Sensitive B-Tree (CSB-Tree)



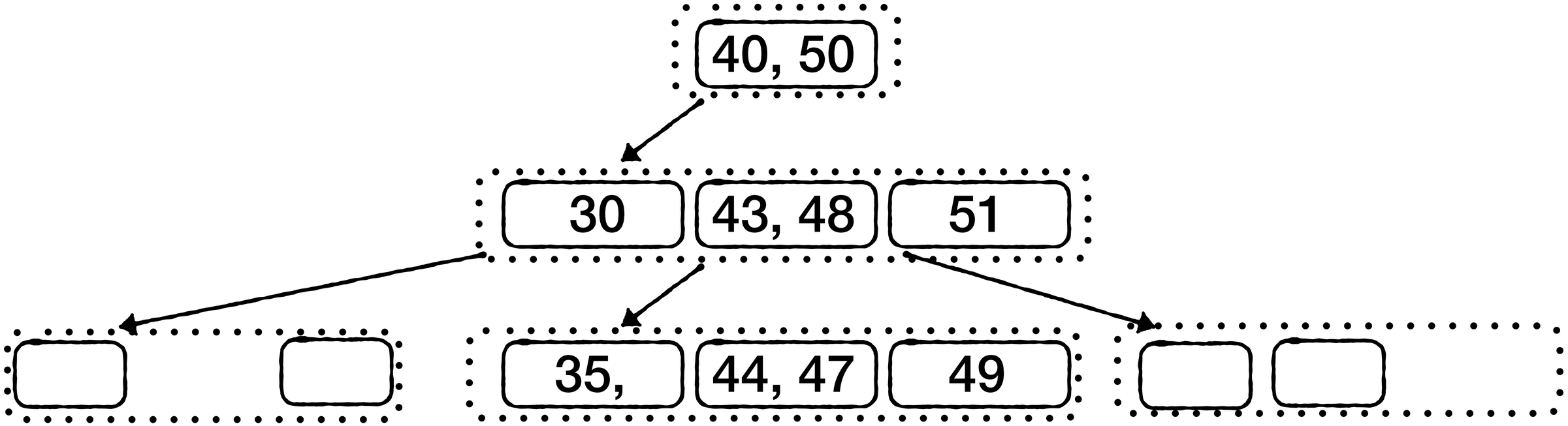


All children of a node are stored contiguously in “node group”



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

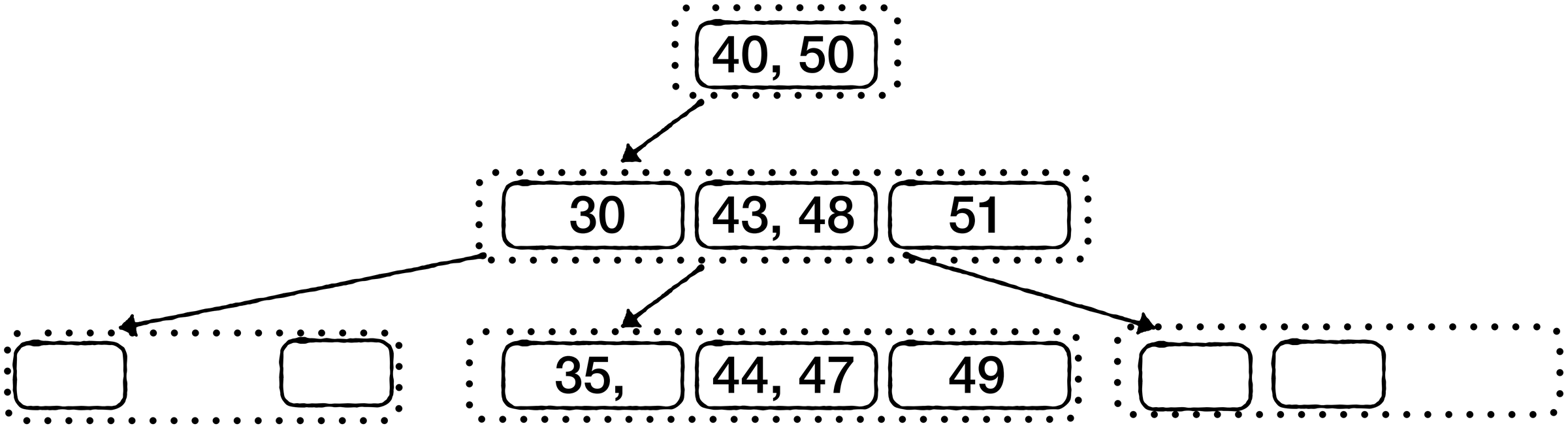
Most pointers are eliminated (only one needed per node group)



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

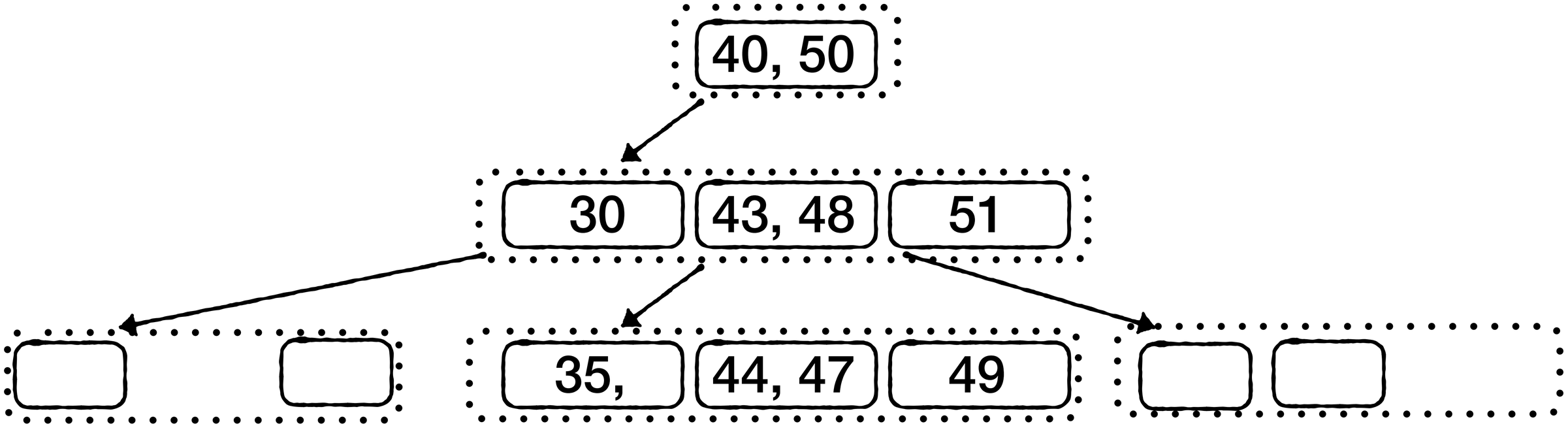


All children of a node are stored contiguously in “node group”

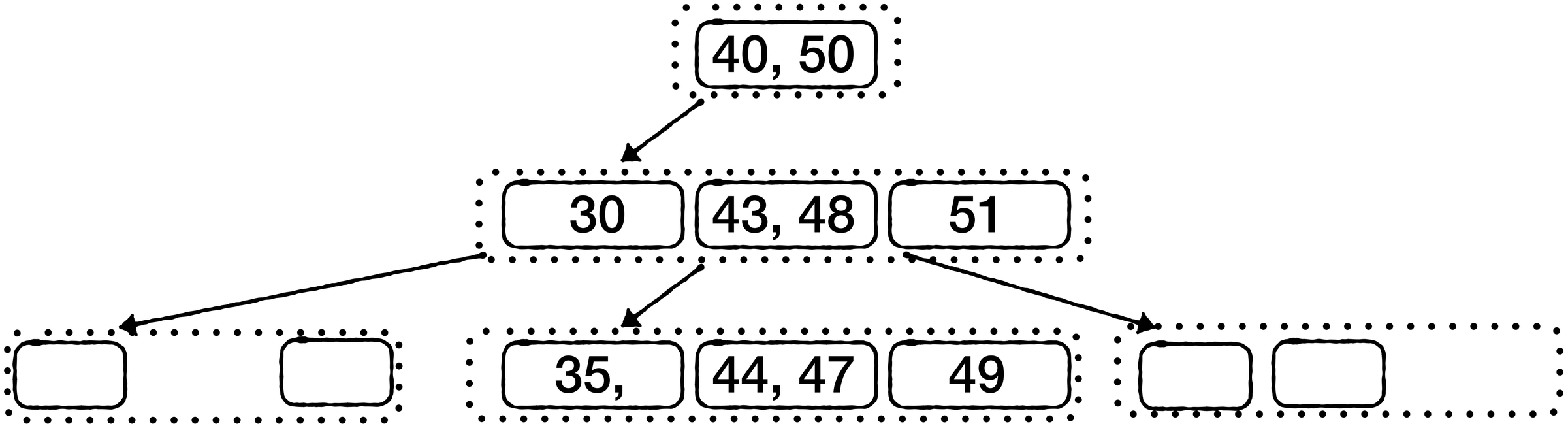
Most pointers are eliminated (only one needed per node group)

Padding is still needed to absorb insertions

B nodes per node group due to fanout of tree



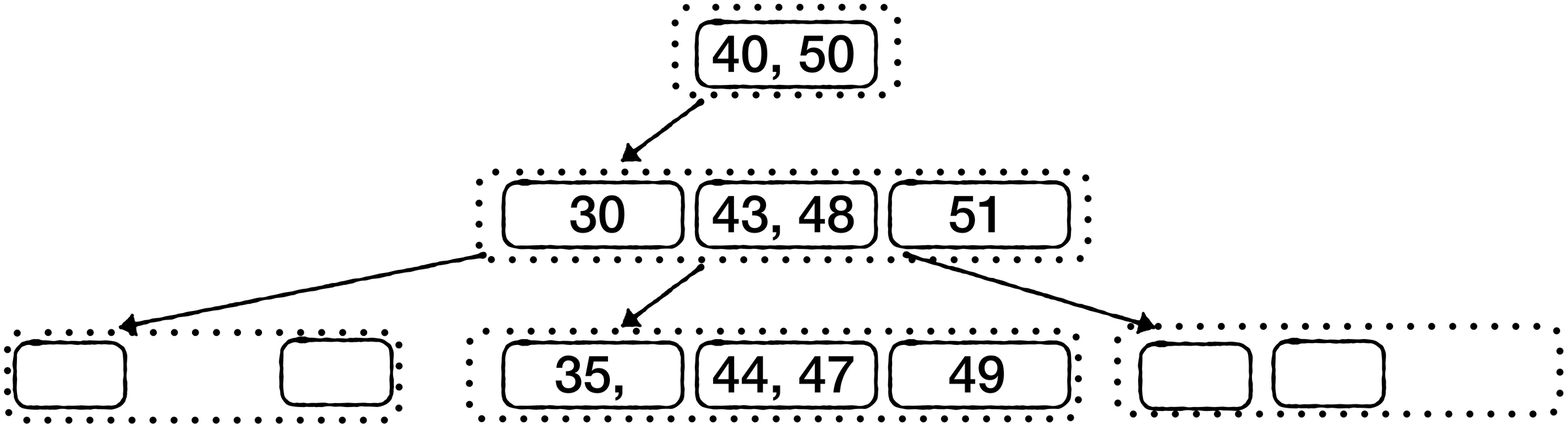
Fanout is B/2 rather than B/4



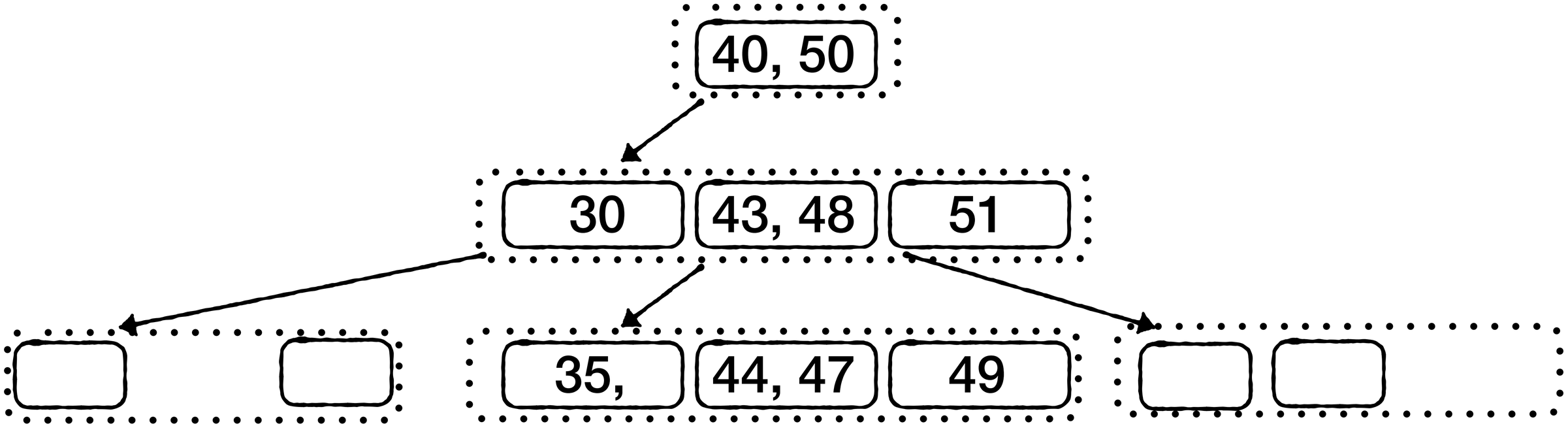
Fanout is B/2 rather than B/4

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

Write cost?

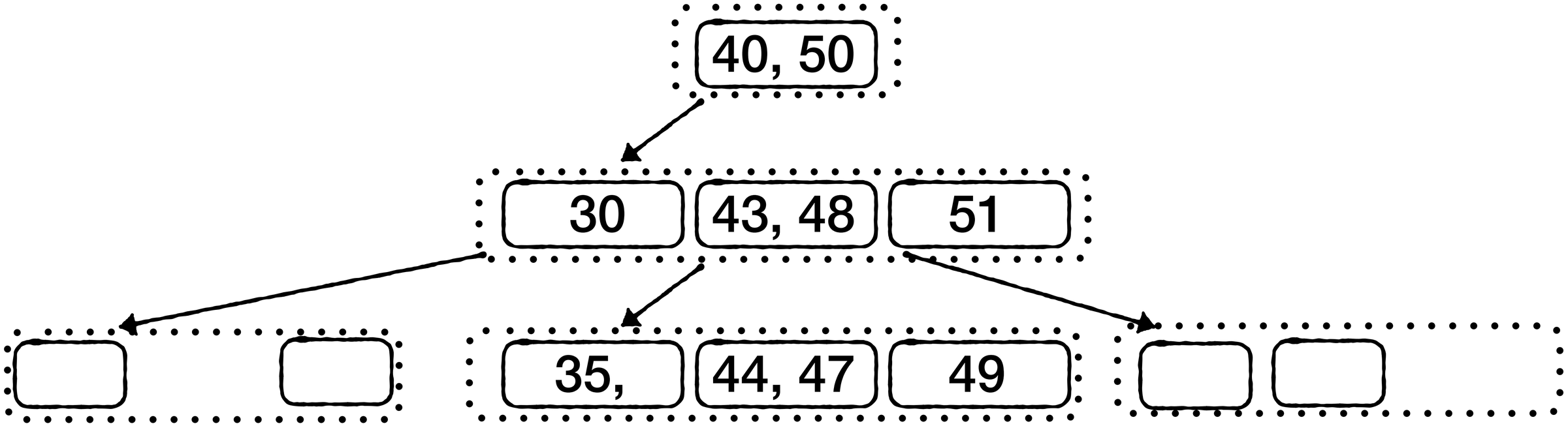


Write cost?



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

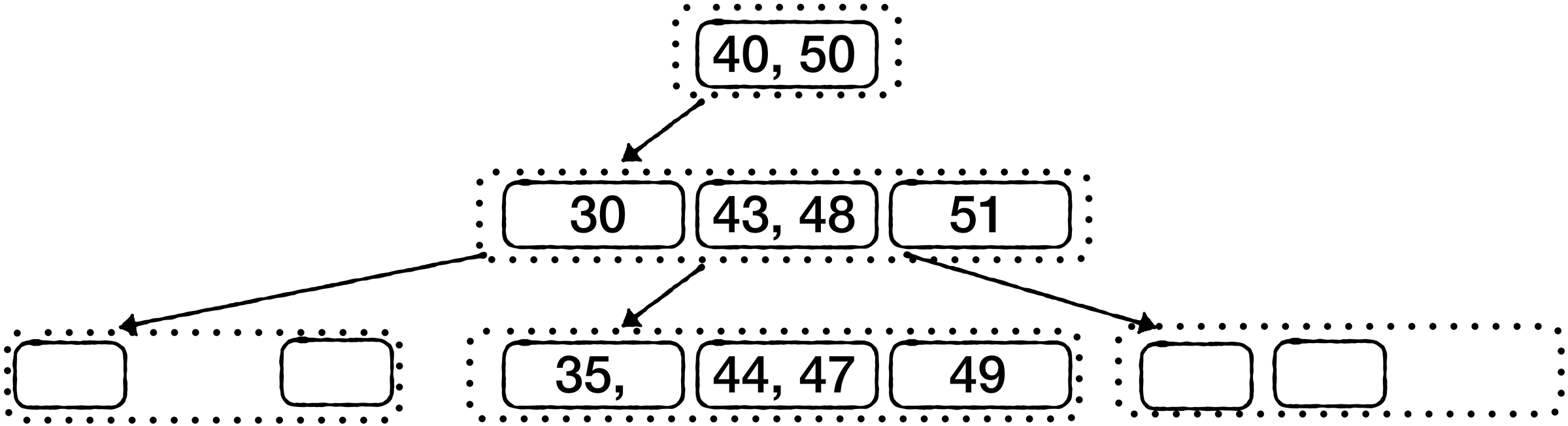
Write cost?



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

Which costs O(B) cache misses

Write cost?

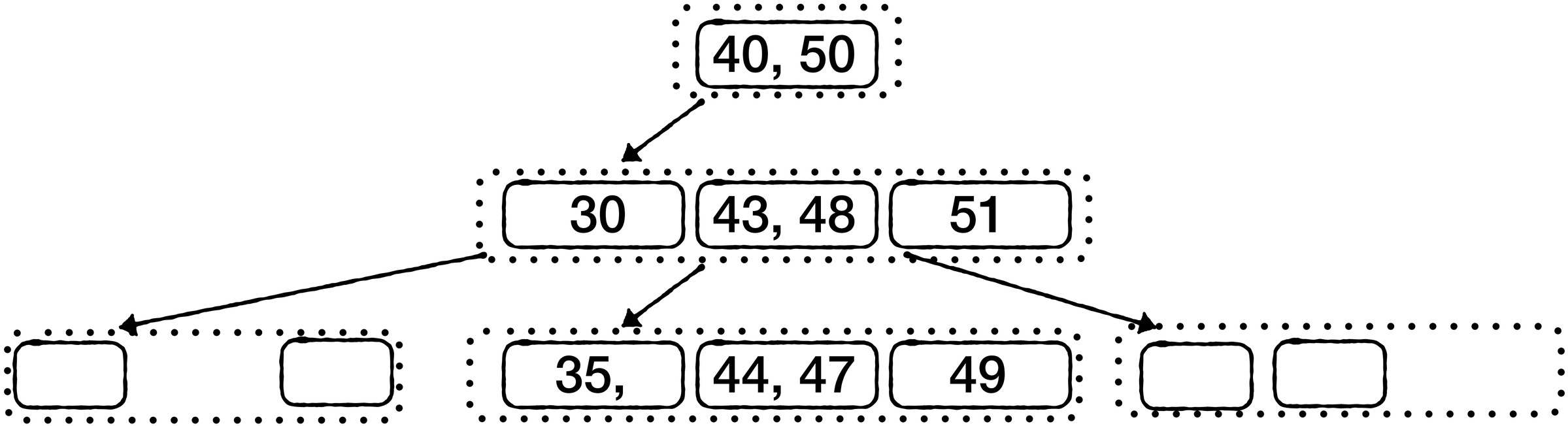


Every B/2 insertions, a leaf splits, causing us to rewrite a node group

Which costs O(B) cache misses

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

Write cost?

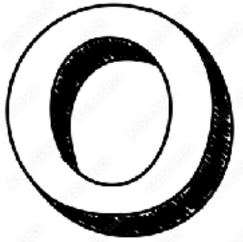


Every B/2 insertions, a leaf splits, causing us to rewrite a node group

Which costs O(B) cache misses

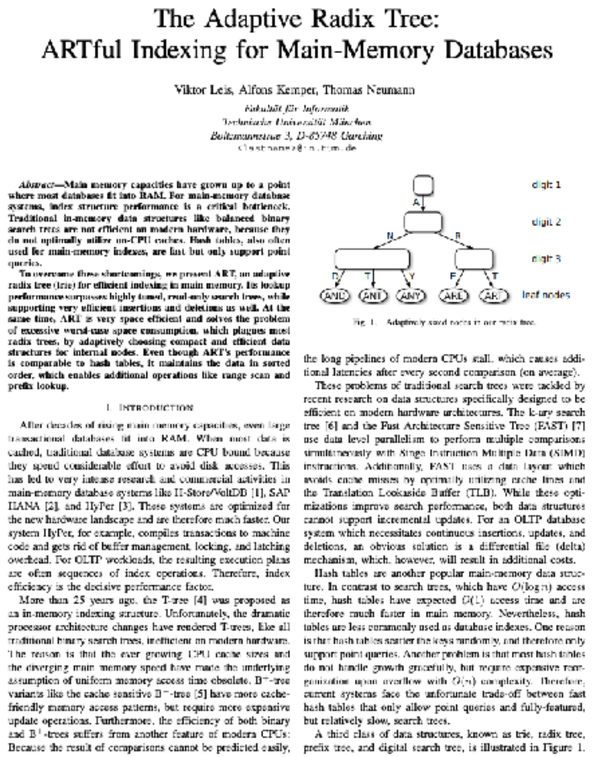
$O(\log_{B/2}(N))$

Better Worst-
Case



AVL-Tree	**B-Tree**	**T-Tree**	**CSS-Tree**	**CSB-Tree**	**ART**
1962	1970	1985	1998	2000	2013

Adaptive Radix Tree (ART)



The Adaptive Radix Tree:
ARTful Indexing for Main-Memory Databases

Viktor Leis, Alfons Kemper, Thomas Neumann
Fakultät für Informatik
Technische Universität München
Boltzmannstrasse 3, D-85748 Garching
<lastname>@in.tum.de

Abstract—Main memory capacities have grown up to a point where most databases fit into RAM. For main-memory database systems, index structure performance is a critical bottleneck. Traditional in-memory data structures like balanced binary search trees are not efficient on modern hardware, because they do not optimally utilize on-CPU caches. Hash tables, also often used for main-memory indexes, are fast but only support point queries.

To overcome these shortcomings, we present ART, an adaptive radix tree (trie) for efficient indexing in main memory. Its lookup performance surpasses highly tuned, read-only search trees, while supporting very efficient insertions and deletions as well. At the same time, ART is very space efficient and solves the problem of excessive worst-case space consumption, which plagues most radix trees, by adaptively choosing compact and efficient data structures for internal nodes. Even though ART's performance is comparable to hash tables, it maintains the data in sorted order, which enables additional operations like range scan and prefix lookup.

I. Introduction

After decades of rising main memory capacities, even large transactional databases fit into RAM. When most data is cached, traditional database systems are CPU bound because they spend considerable effort to avoid disk accesses. This has led to very intense research and commercial activities in main-memory database systems like H-Store/VoltDB [1], SAP HANA [2], and HyPer [3]. These systems are optimized for the new hardware landscape and are therefore much faster. Our system HyPer, for example, compiles transactions to machine code and gets rid of buffer management, locking, and latching overhead. For OLTP workloads, the resulting execution plans are often sequences of index operations. Therefore, index efficiency is the decisive performance factor.

More than 25 years ago, the T-tree [4] was proposed as an in-memory indexing structure. Unfortunately, the dramatic processor architecture changes have rendered T-trees, like all traditional binary search trees, inefficient on modern hardware. The reason is that the ever growing CPU cache sizes and the emerging main memory speed have made the underlying assumption of uniform memory access time obsolete. B$^+$-tree variants like the cache sensitive B$^+$-tree [5] have more cache-friendly memory access patterns, but require more expensive update operations. Furthermore, the efficiency of both binary and B$^+$-trees suffers from another feature of modern CPUs: Because the result of comparisons cannot be predicted easily, the long pipelines of modern CPUs stall, which causes additional latencies after every second comparison (on average).

These problems of traditional search trees were tackled by recent research on data structures specifically designed to be efficient on modern hardware architectures. The k-ary search tree [6] and the Fast Architecture Sensitive Tree (FAST) [7] use data level parallelism to perform multiple comparisons simultaneously with Single Instruction Multiple Data (SIMD) instructions. Additionally, FAST uses a data layout which avoids cache misses by optimally utilizing cache lines and the Translation Lookaside Buffer (TLB). While these optimizations improve search performance, both data structures cannot support incremental updates. For an OLTP database system which necessitates continuous insertions, updates, and deletions, an obvious solution is a differential file (delta) mechanism, which, however, will result in additional costs.

Hash tables are another popular main-memory data structure. In contrast to search trees, which have $O(\log n)$ access time, hash tables have expected $O(1)$ access time and are therefore much faster in main memory. Nevertheless, hash tables are less commonly used as database indexes. One reason is that hash tables scatter the keys randomly, and therefore only support point queries. Another problem is that most hash tables do not handle growth gracefully, but require expensive reorganizations upon overflow with $O(n)$ complexity. Therefore, current systems face the unfortunate trade-off between fast hash tables that only allow point queries and fully-featured, but relatively slow, search trees.

A third class of data structures, known as trie, radix tree, prefix tree, and digital search tree, is illustrated in Figure 1.

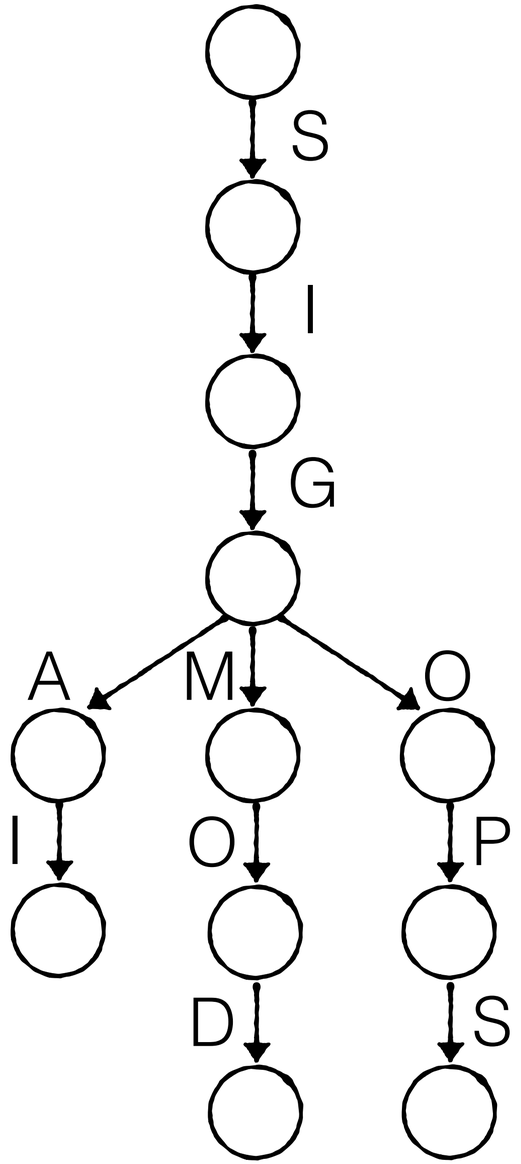
The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases

Viktor Leis, Alfons Kemper, Thomas Neumann ICDE 2013

Radix Tree

Keys

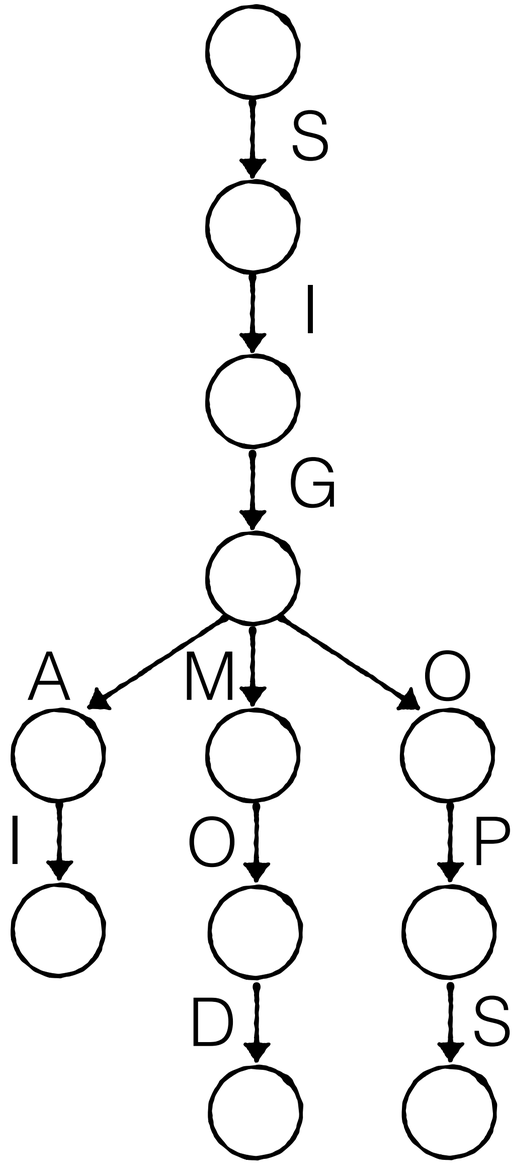
SIGAI
SIGMOD
SIGOPS



Radix Tree

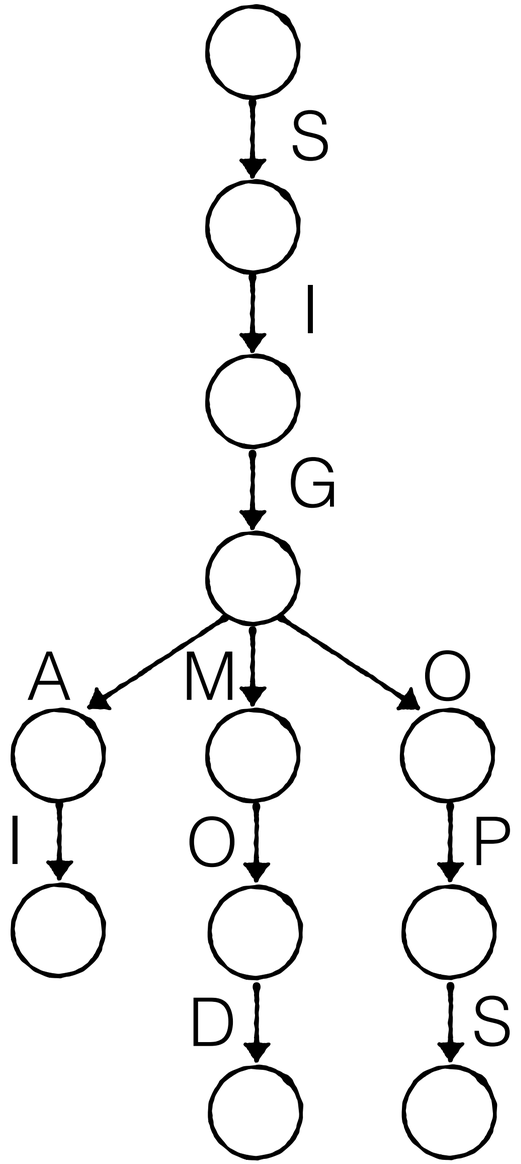
Keys

SIGAI
SIGMOD
SIGOPS



Fanout - 256
(1 byte)

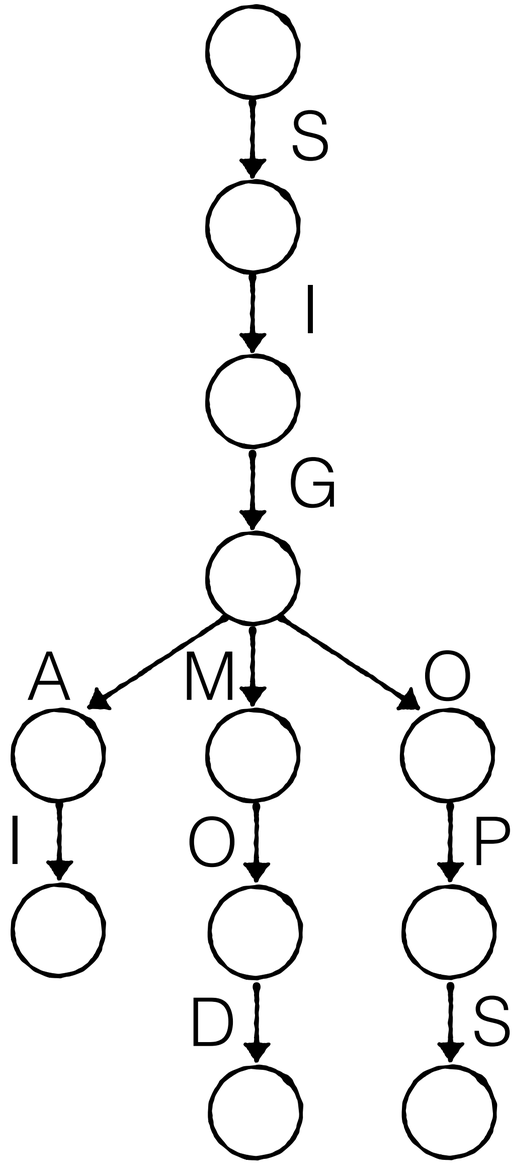
Promises?



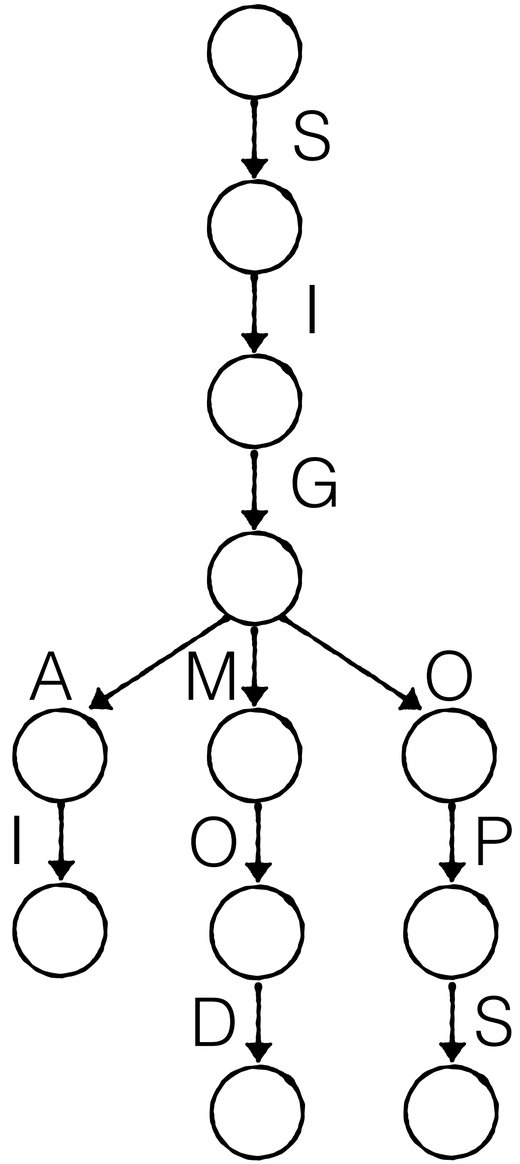
Promises?

No rotations or splitting

Easier for concurrency



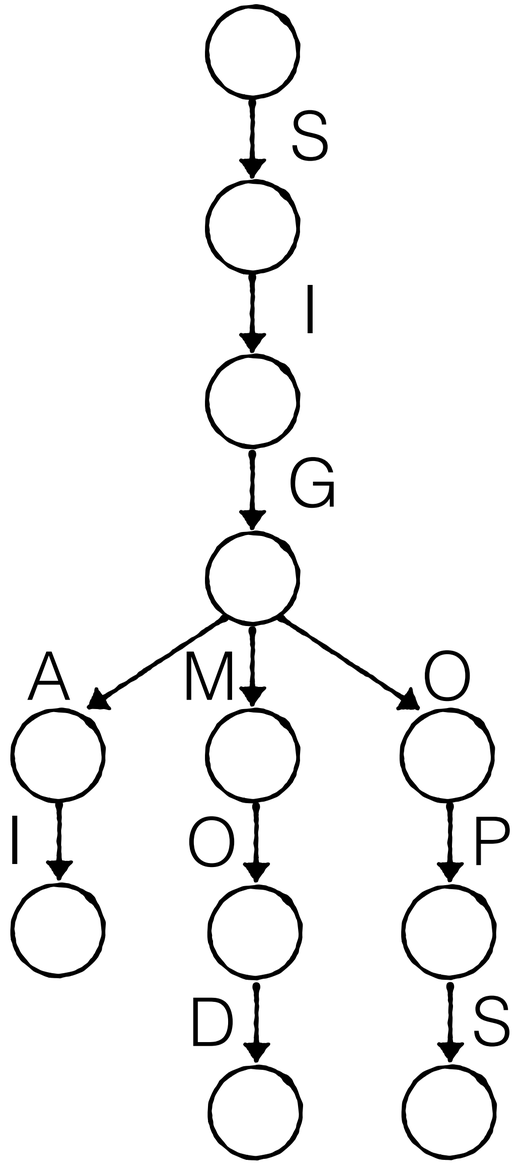
Promises?



Search time

O(key length * work per node)

Promises?



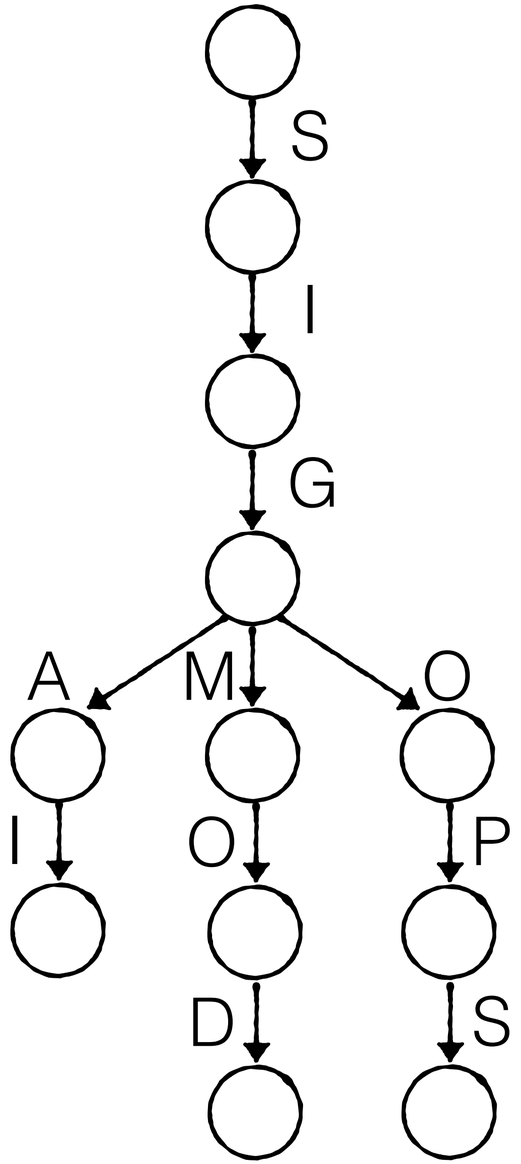
Search time

O(**key length** * work per node)

↑

In bytes

Promises?



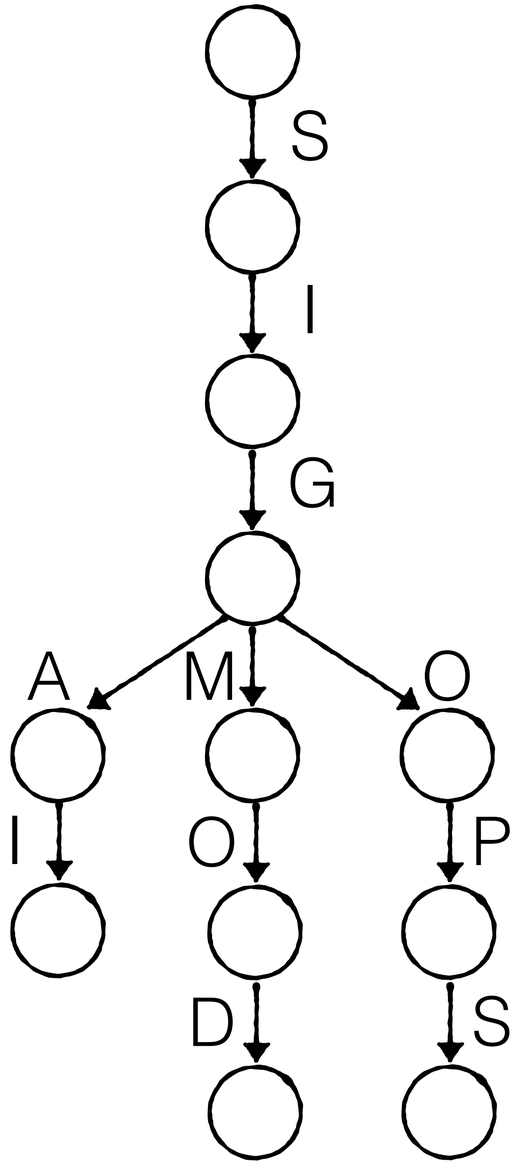
Search time

O(**key length** * work per node)

In bytes

4 for integers

Promises?



Search time

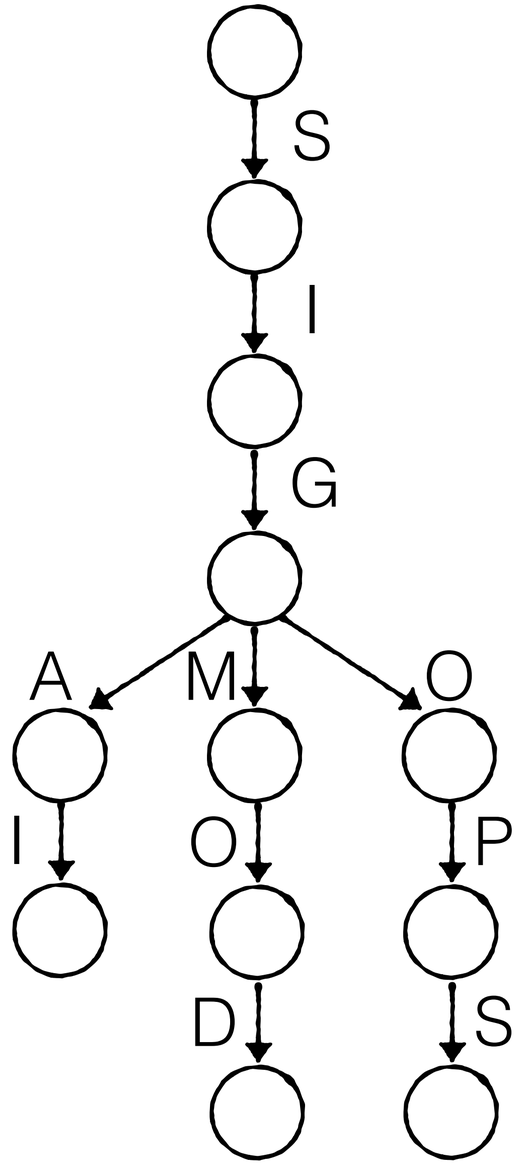
O(**key length** * work per node)

In bytes

4 for integers

Independent of N

Promises?



Search time

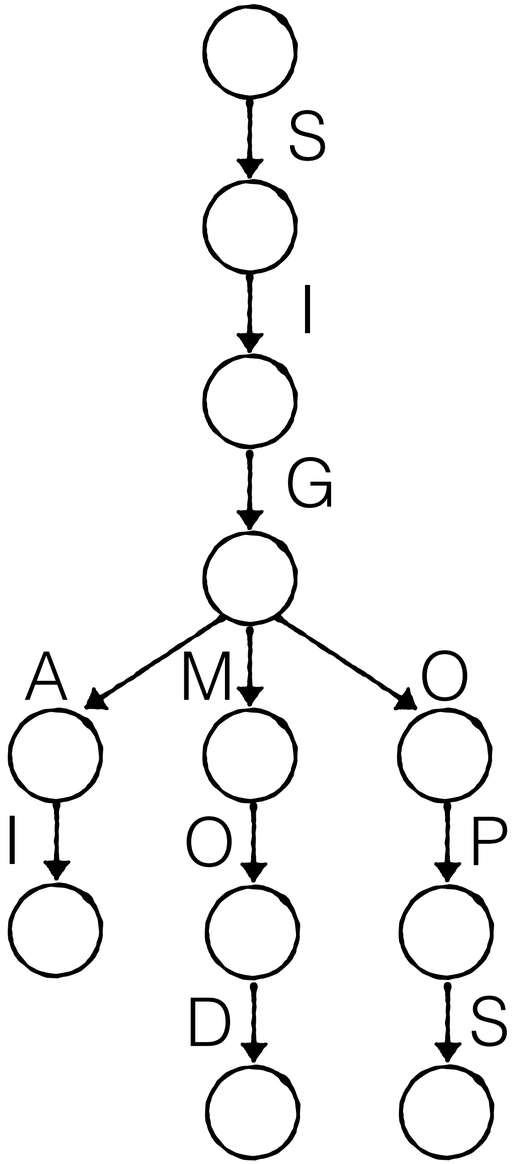
O(key length * **work per node**)

↑

How to structure each node to keep this small?

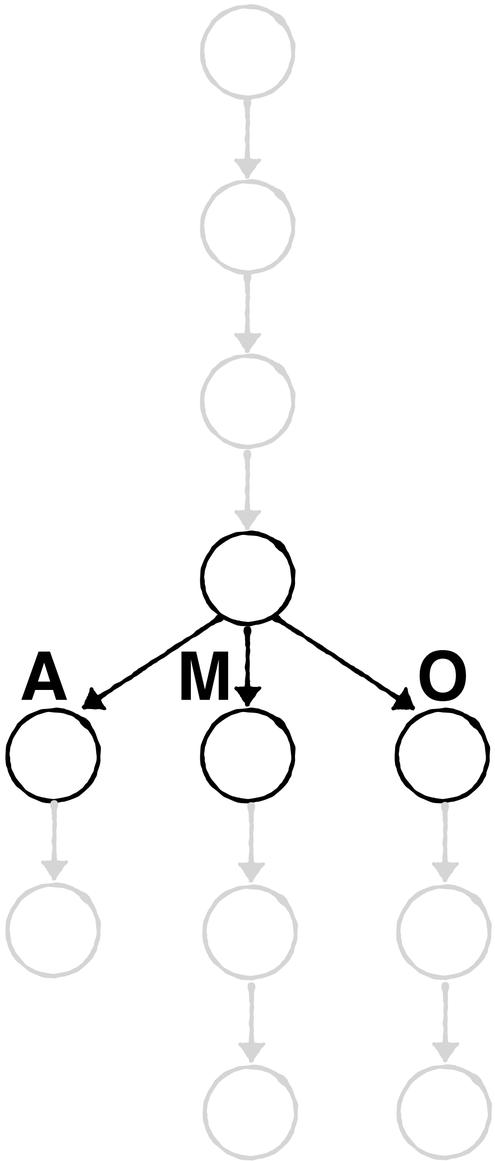
Pointers Array

Each node consists of 256 pointers



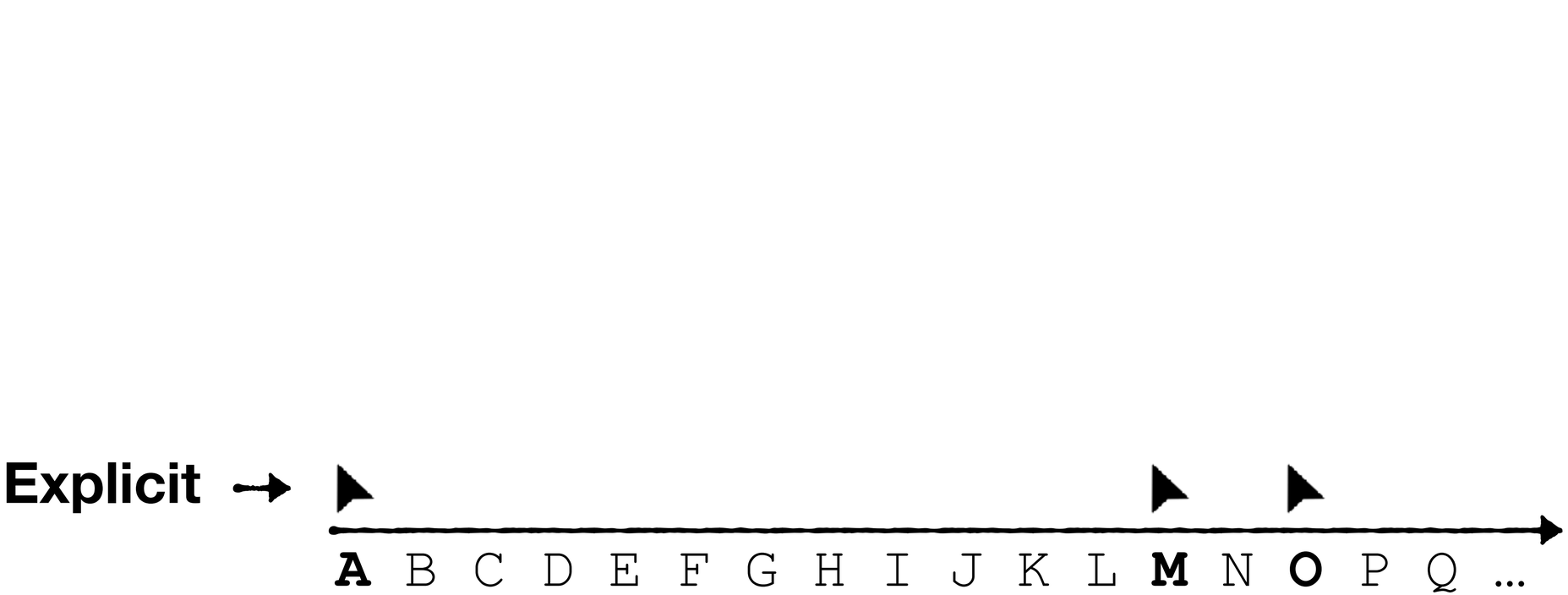
Pointers Array

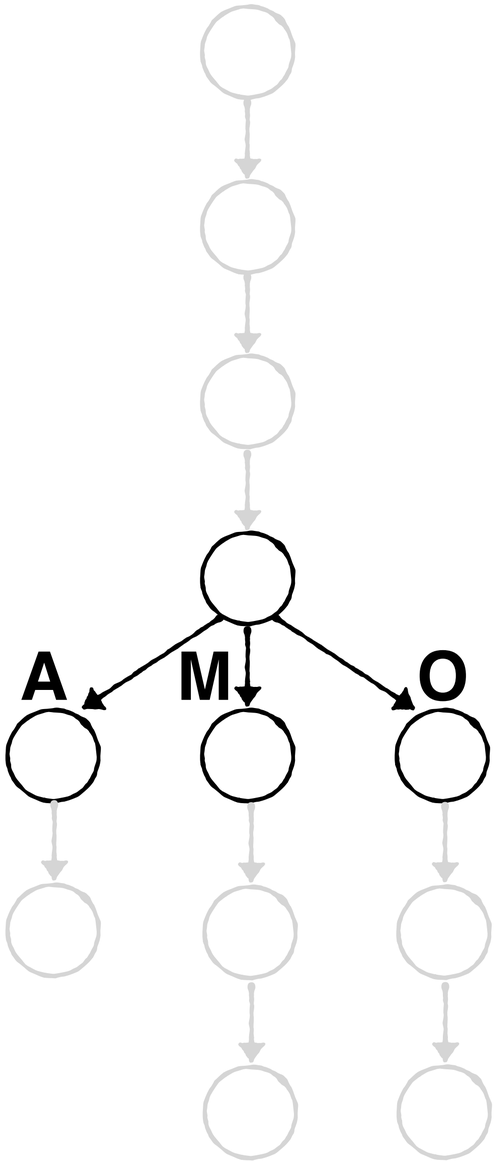
Each node consists of 256 pointers



Pointers Array

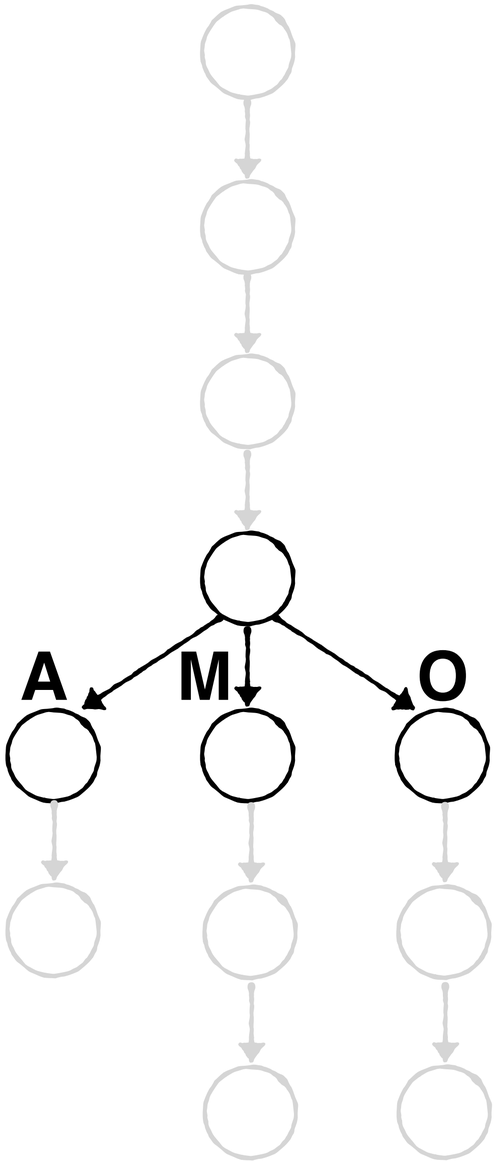
Each node consists of 256 pointers



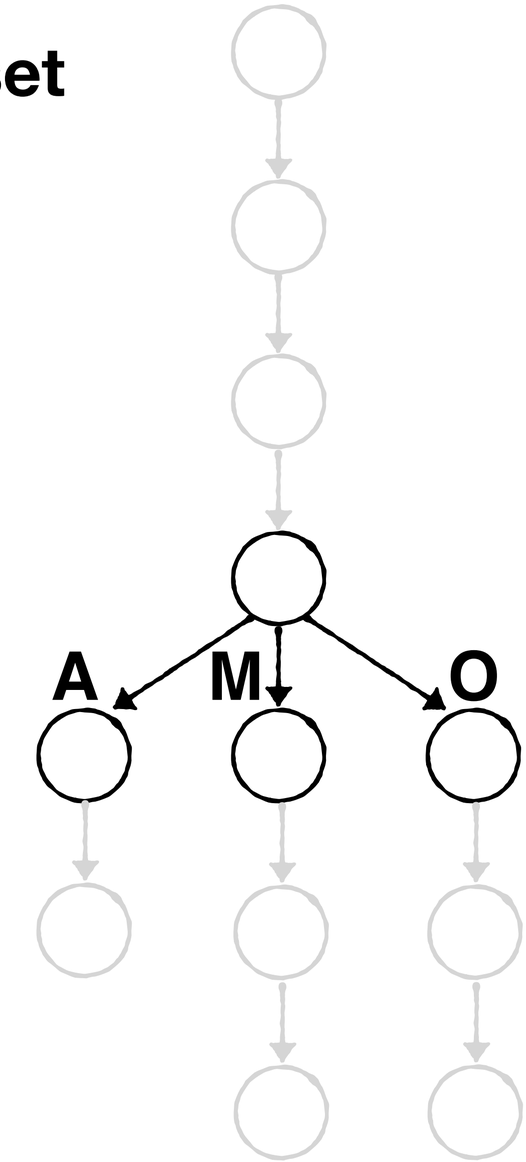


Pointers Array

Each node consists of 256 pointers

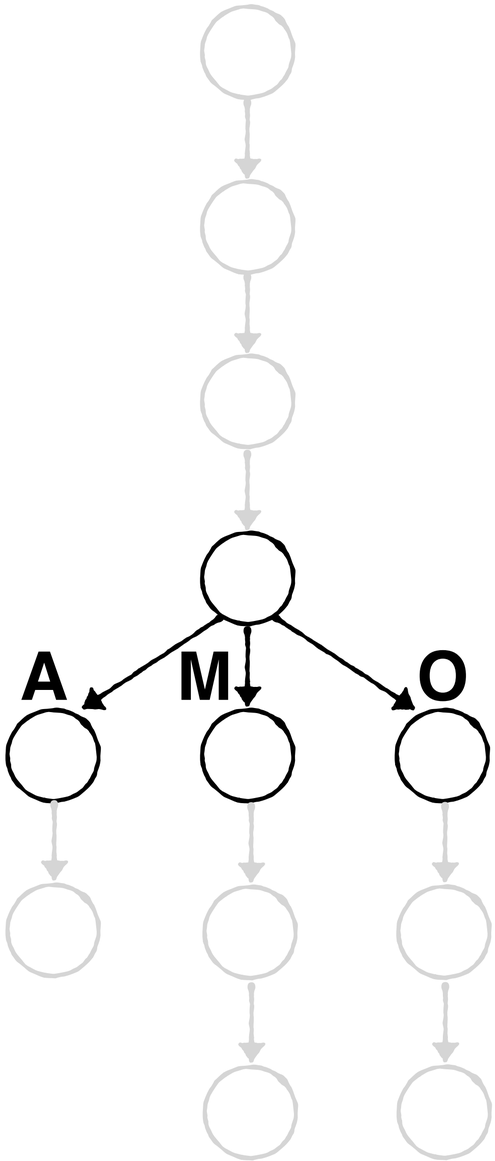


Query knows starting point and can skip to target offset



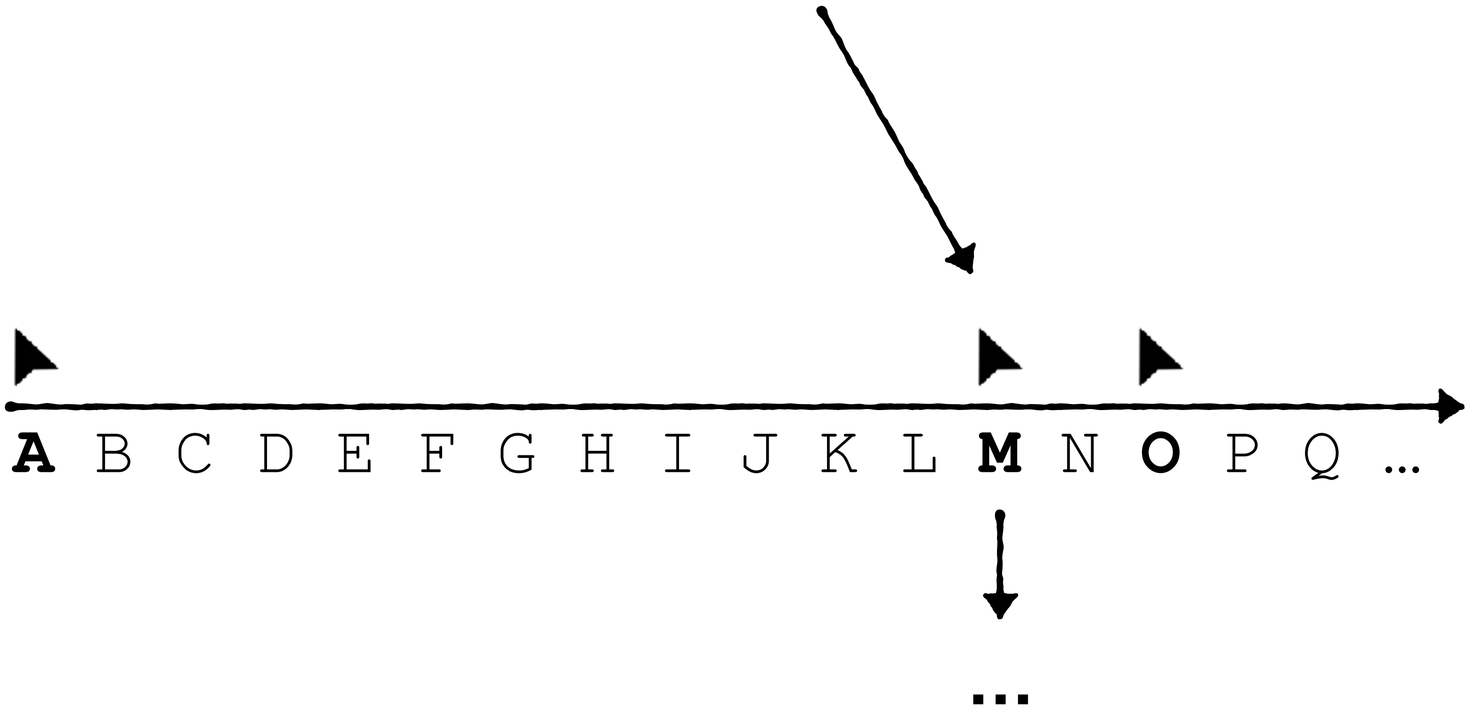
Query knows starting point and can skip to target offset

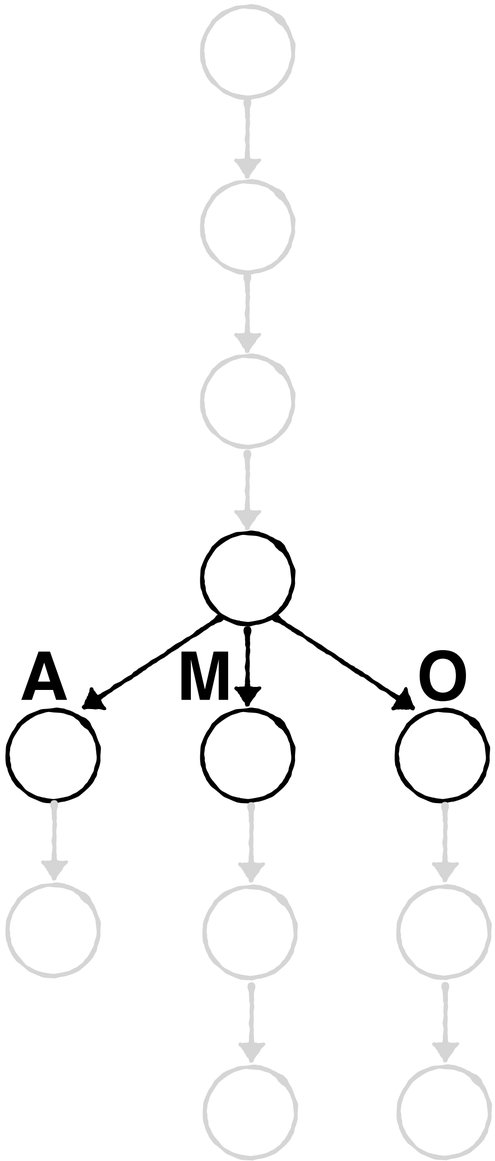
e.g., SIGMOD



Query knows starting point and can skip to target offset

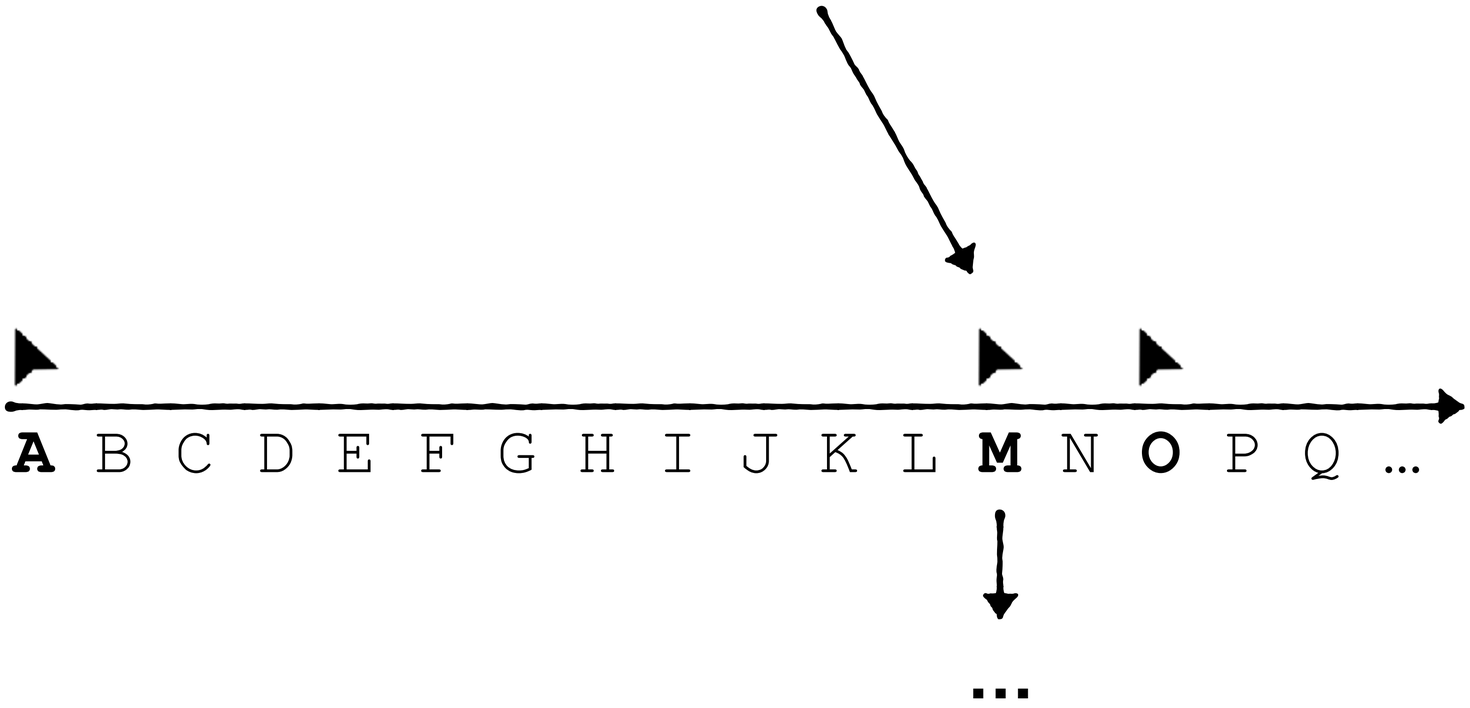
e.g., SIG**M**OD

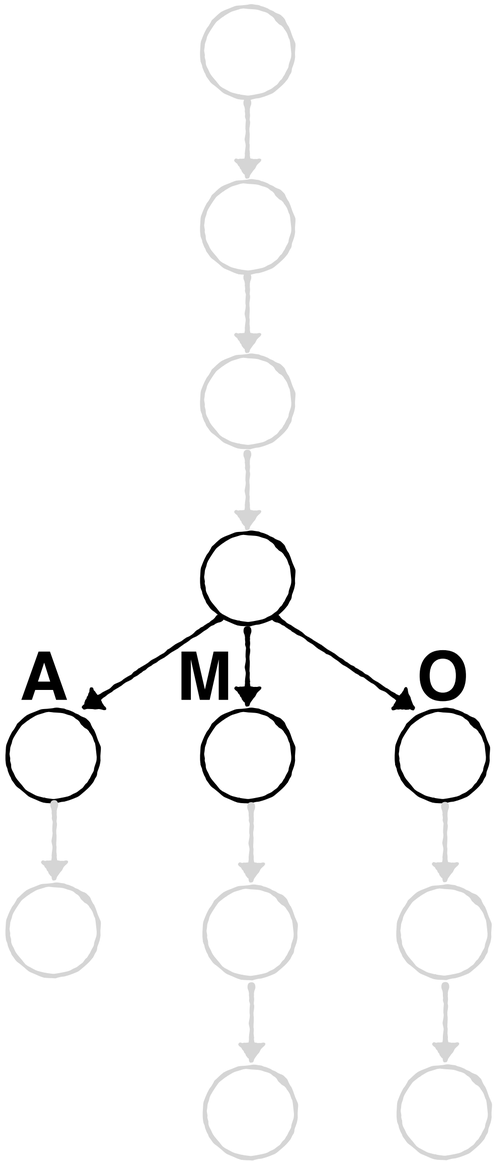




Query knows starting point and can skip to target offset

e.g., SIG**M**OD

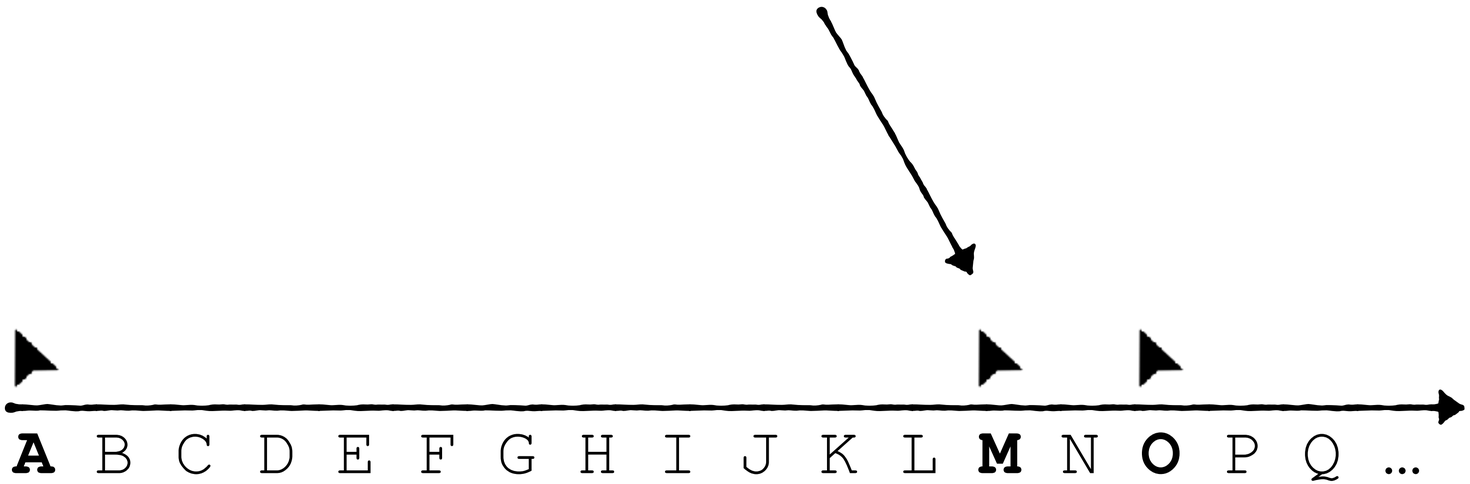


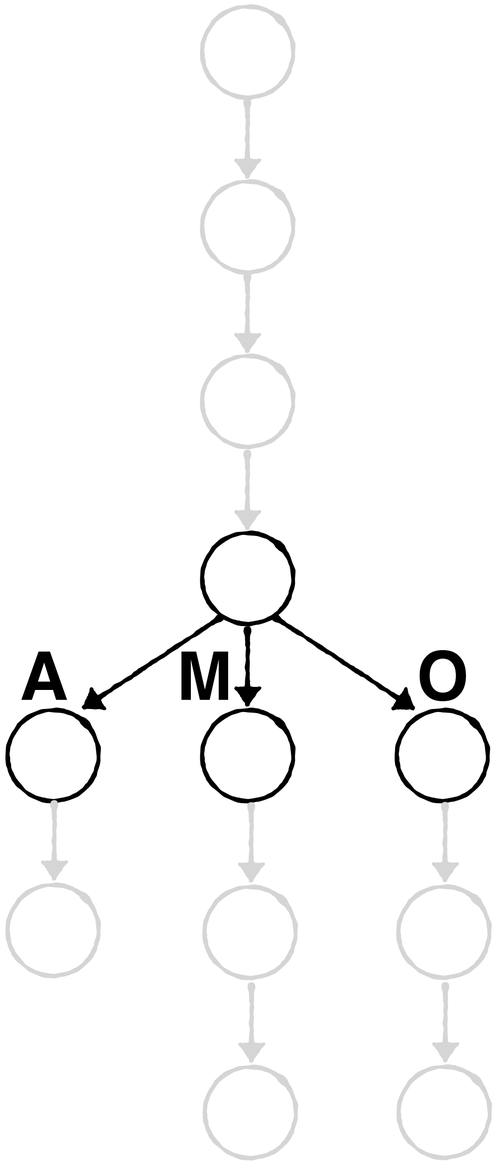


Single access per node :)

Query knows starting point and can skip to target offset

e.g., SIG**M**OD

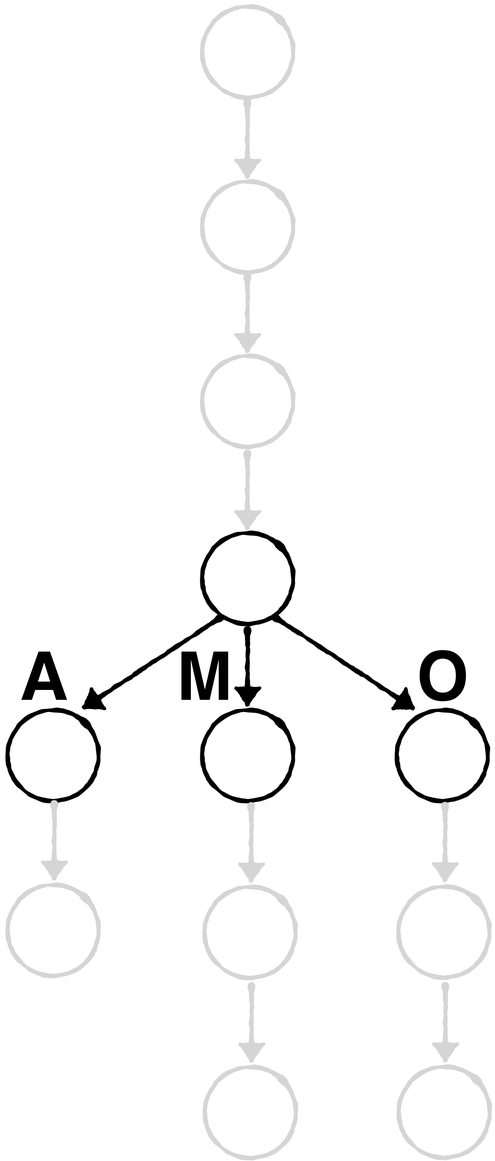




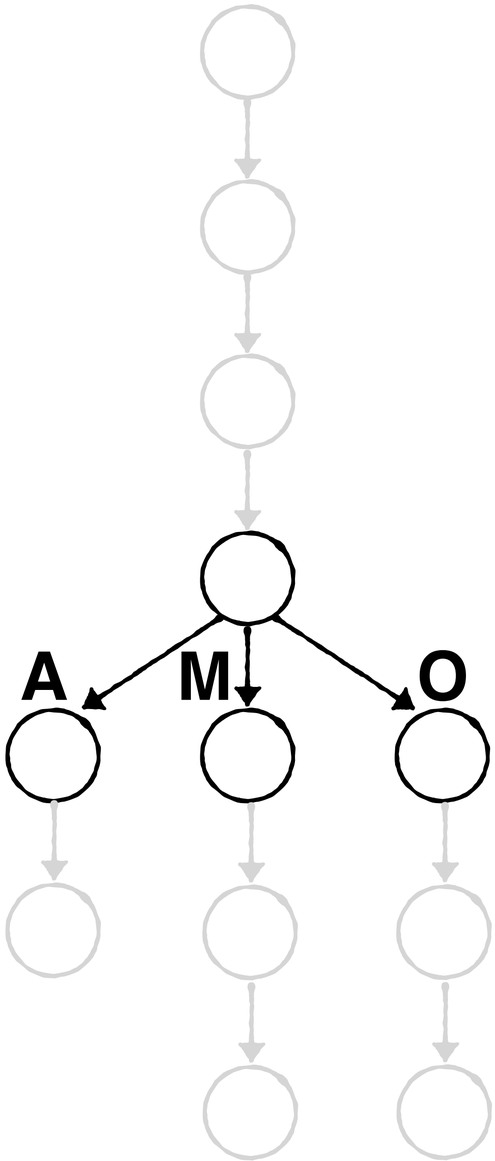
Single access per node :)

Problems?

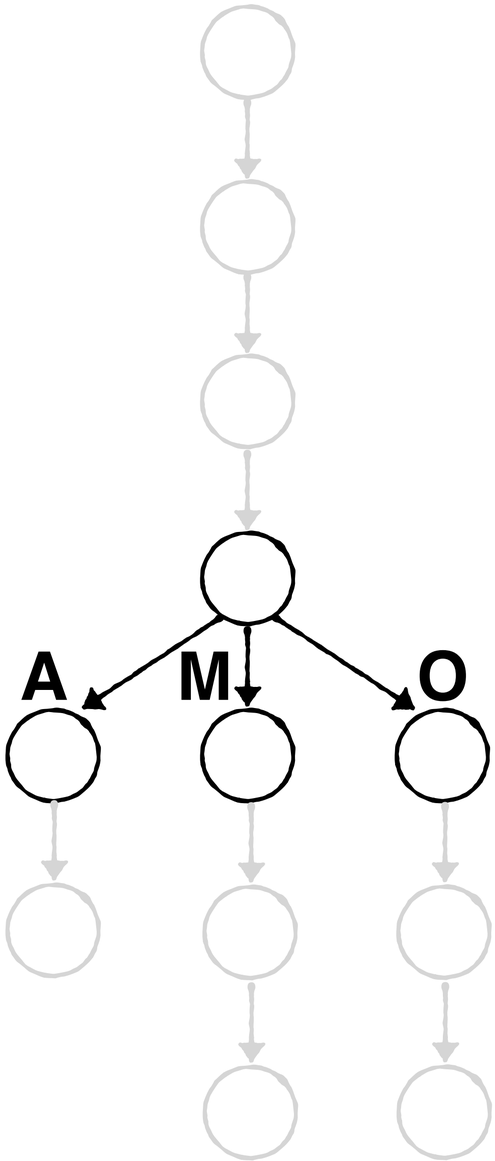

4 bytes
A B C D E F G H I J K L M N O P Q ...


A
M
O

Each node is 1 KB



4 bytes



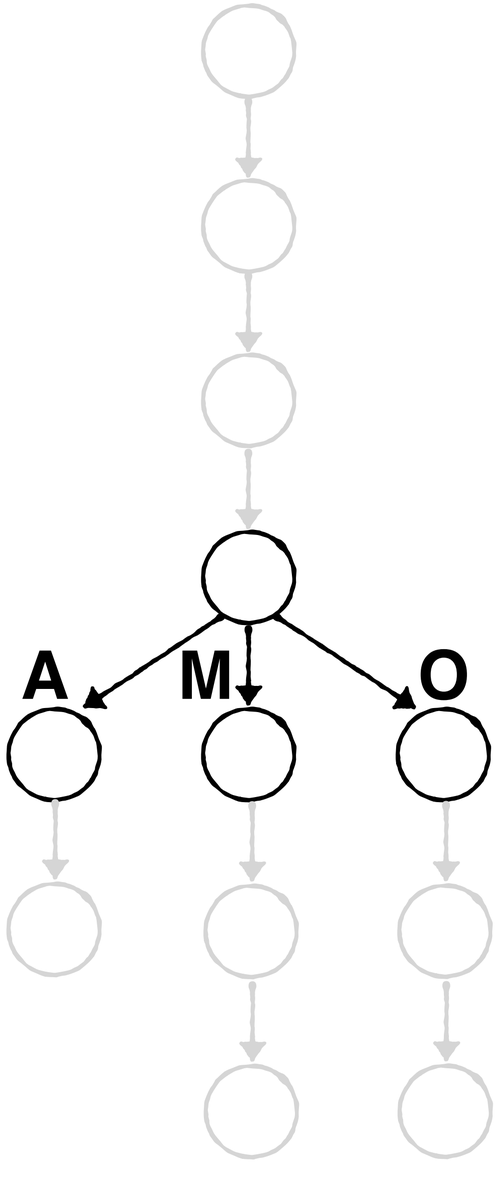
Each node is 1 KB

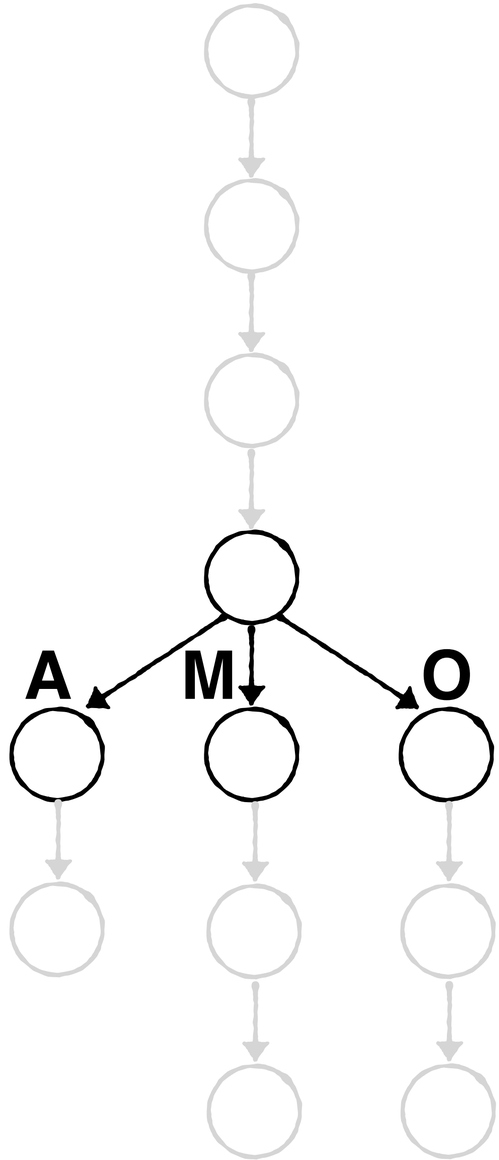
Fine if node is dense, but bad if sparse

Each node is 1 KB

Fine if node is dense, but bad if sparse

Could decrease fanout. Issue?





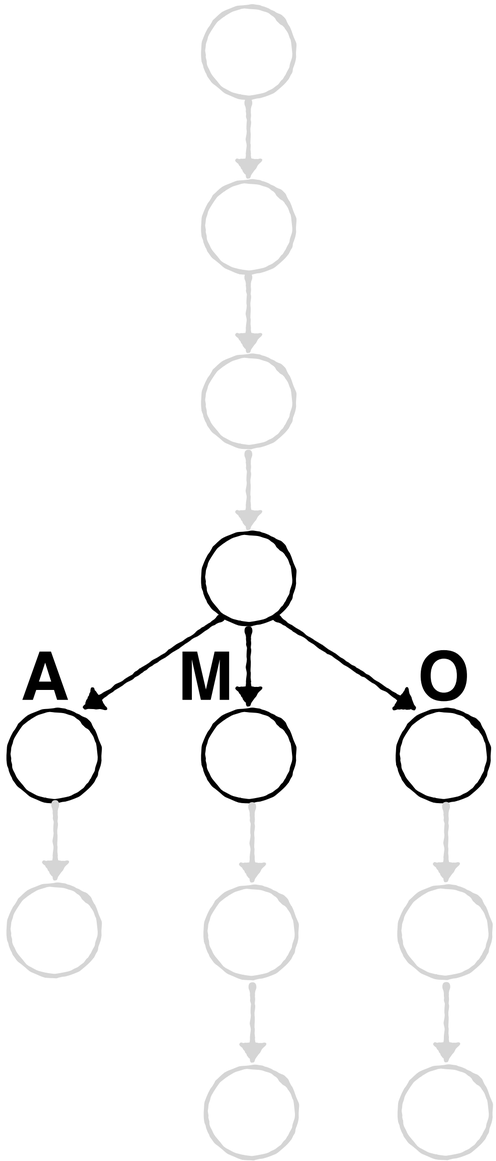
Each node is 1 KB

Fine if node is dense, but bad if sparse

Could decrease fanout. Issue? **Worse runtime.**

Can we get around this runtime vs. space contention?

4 bytes



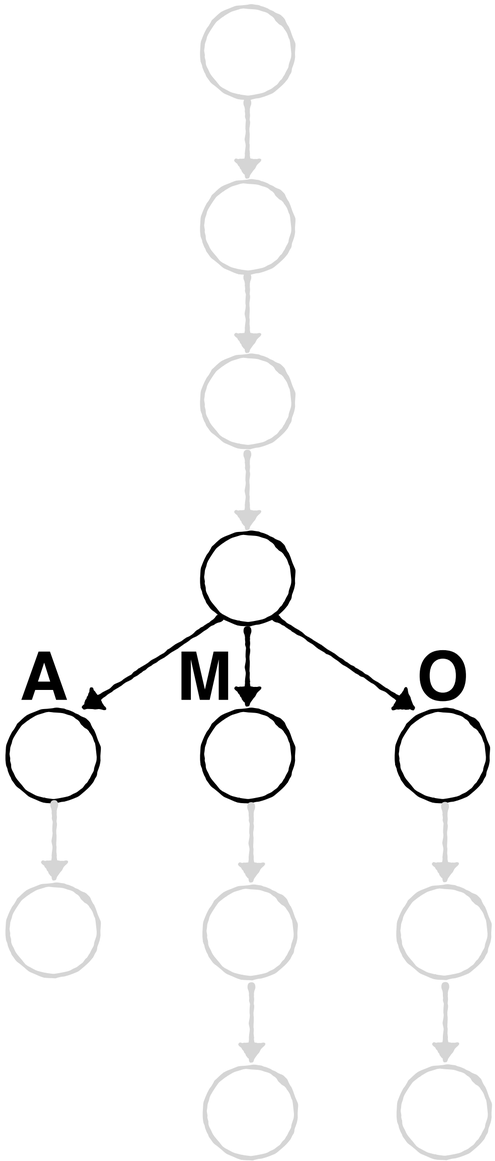
Each node is 1 KB

Fine if node is dense, but bad if sparse

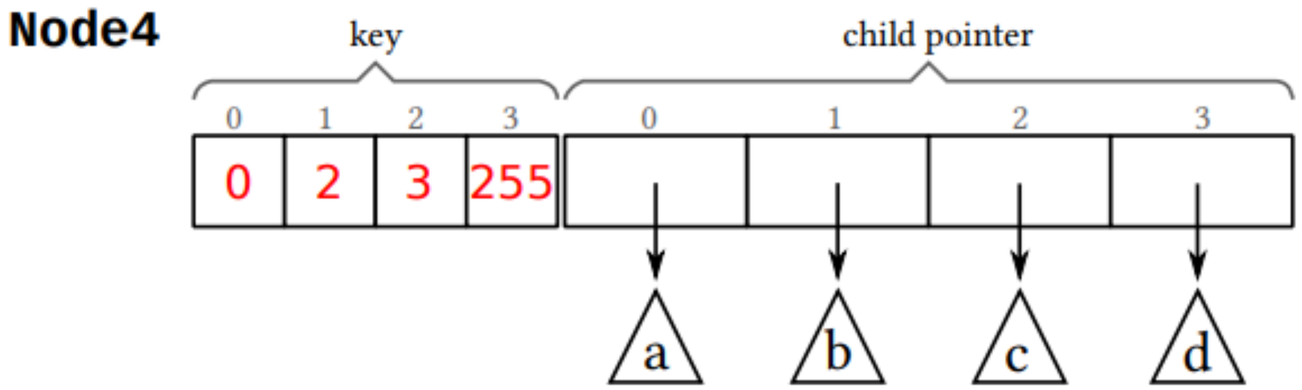
Could decrease fanout. Issue? Worse runtime.

Can we get around this runtime vs. space contention?

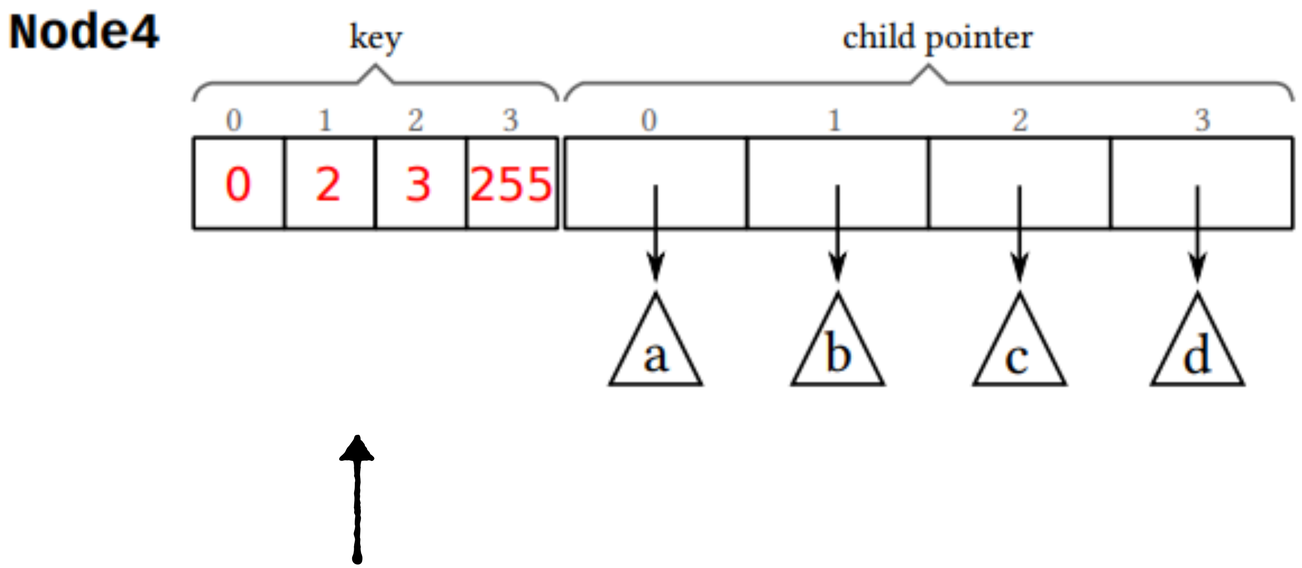
Structure a node based on its number of children



If node has to up 4 children

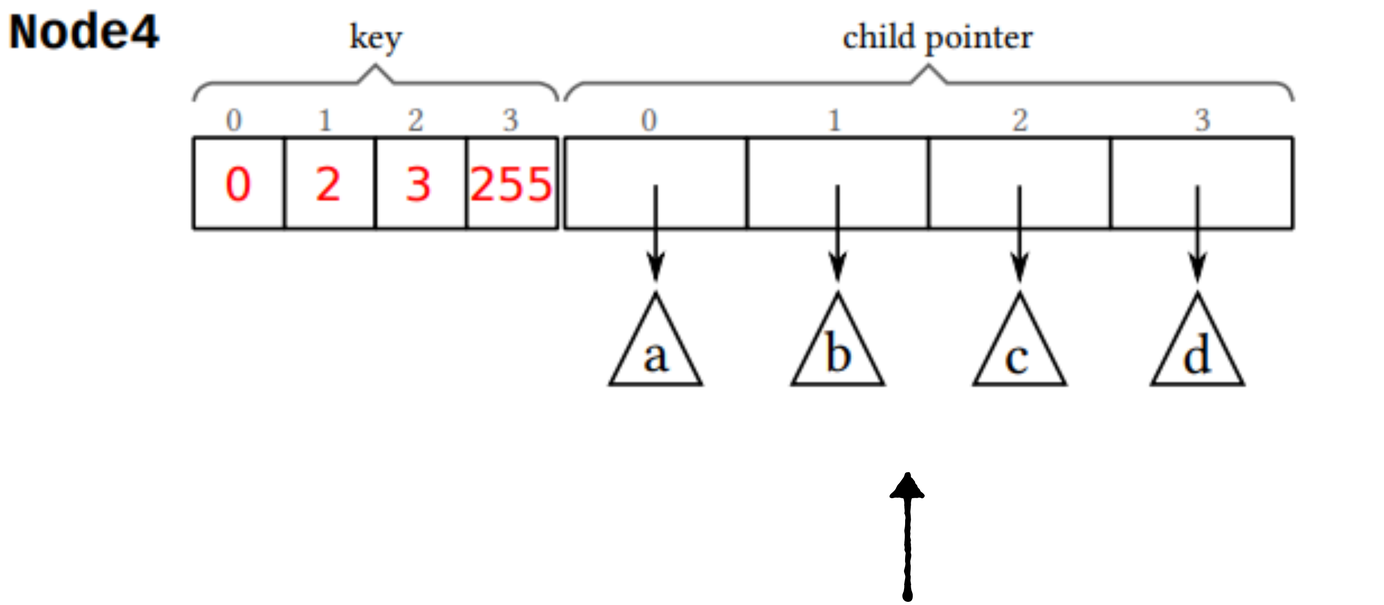


If node has to up 4 children



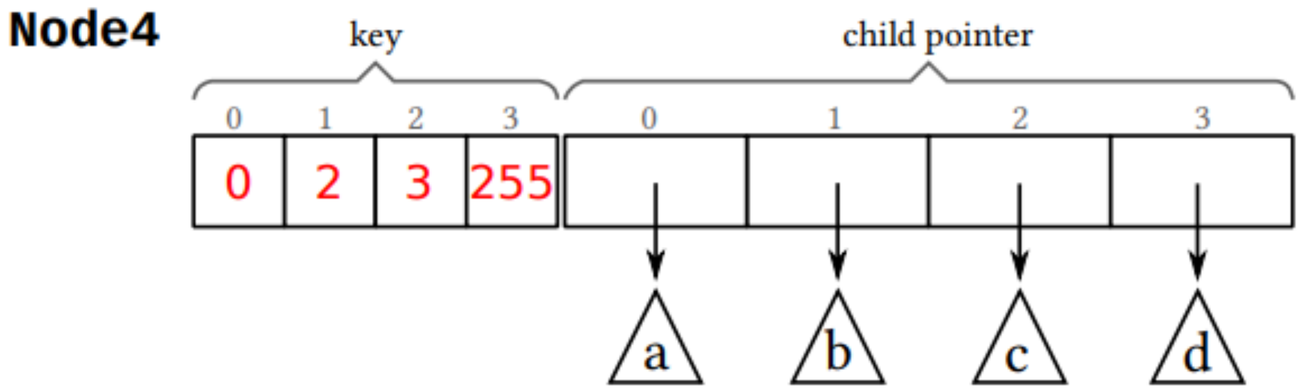
Search using SIMD

If node has to up 4 children



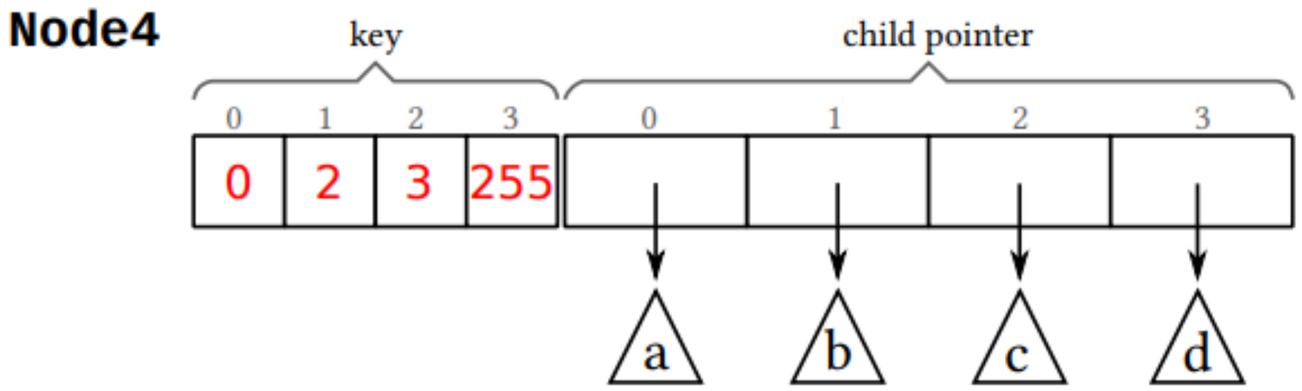
Go to corresponding pointer

If node has to up 4 children



4 + 4 * 8 = 36 bytes

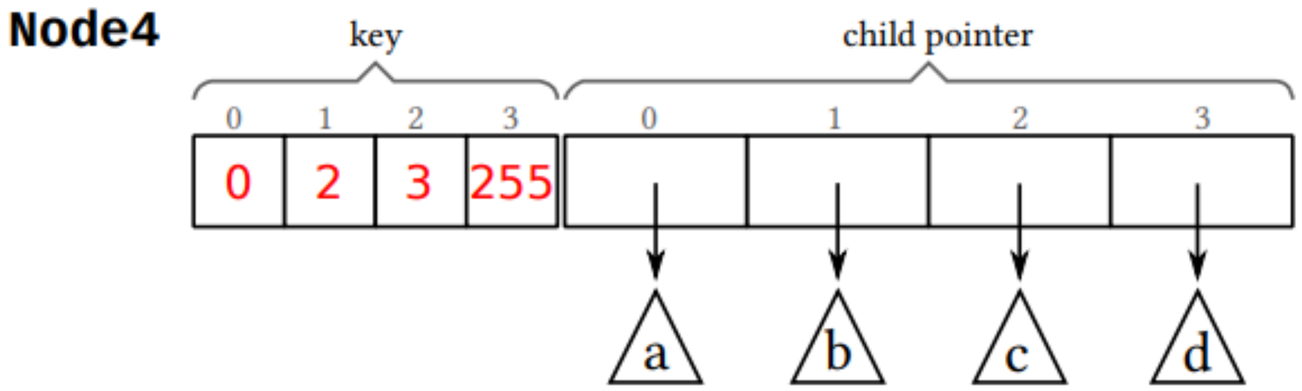
If node has to up 4 children



4 + 4 * 8 = 36 bytes

9 - 18 bytes per key

If node has to up 4 children



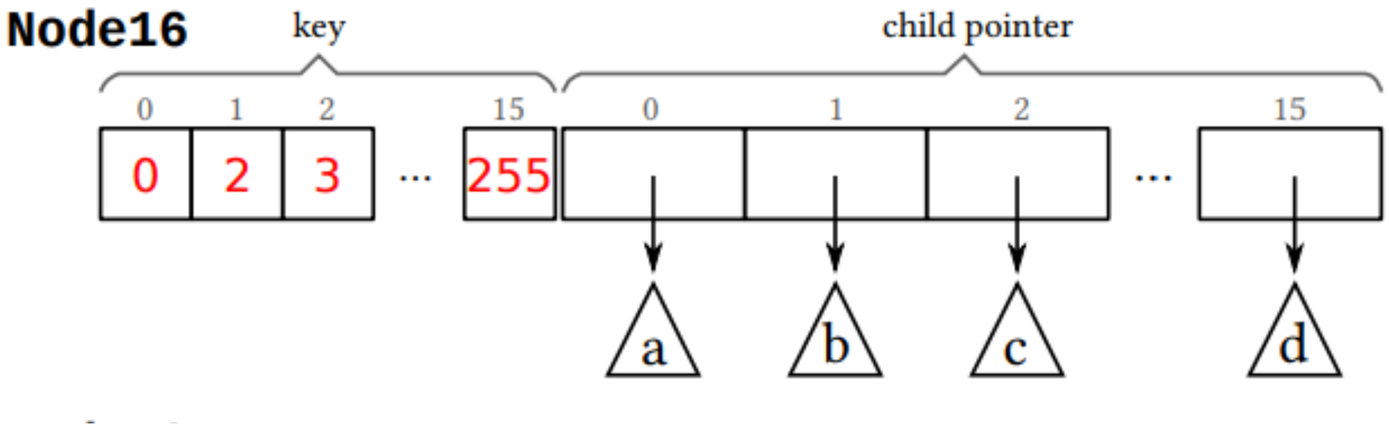
4 + 4 * 8 = 36 bytes

9 - 18 bytes per key

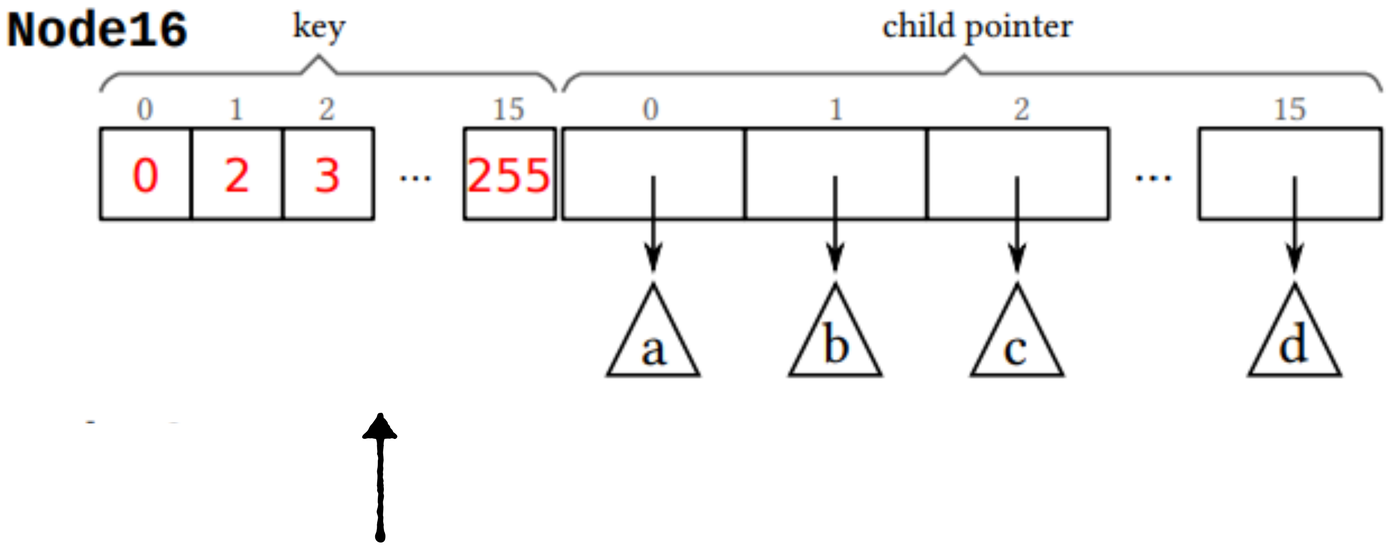
1-2 cache misses per node

If node has 5-16 children

If node has 5-16 children

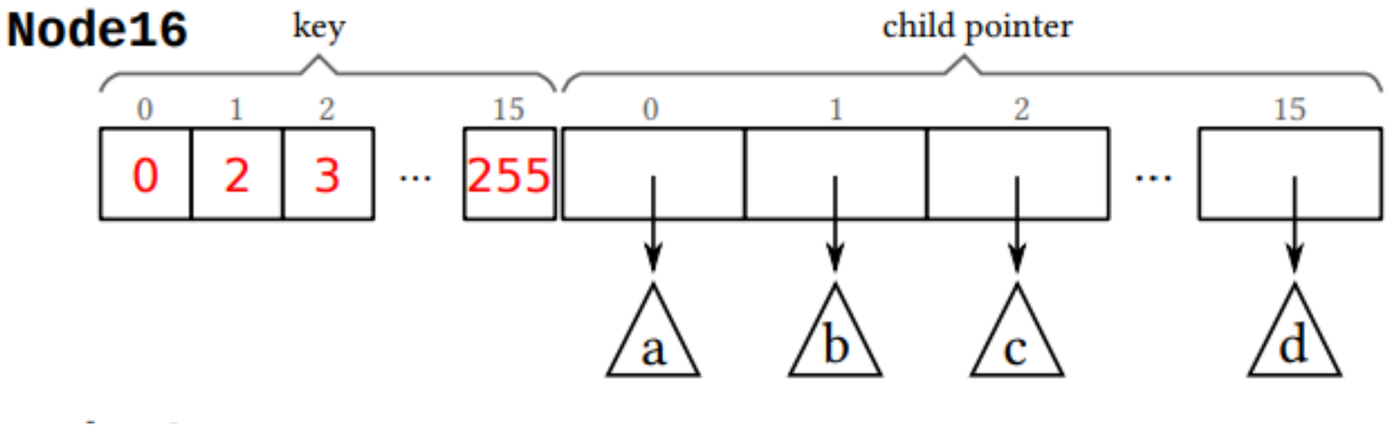


If node has 5-16 children



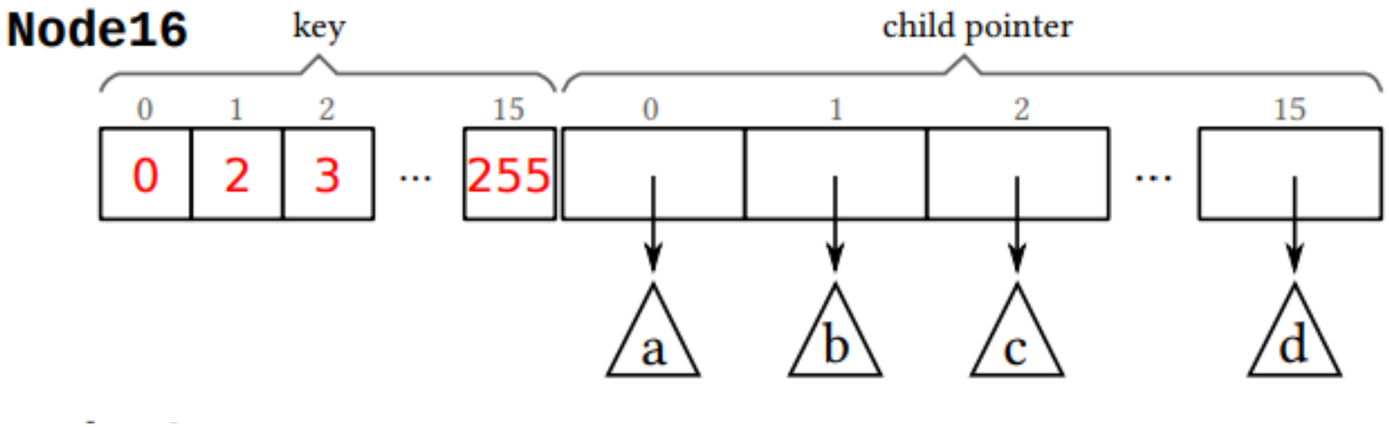
Search using SIMD

If node has 5-16 children



16 + 16 * 8 = 144 B

If node has 5-16 children

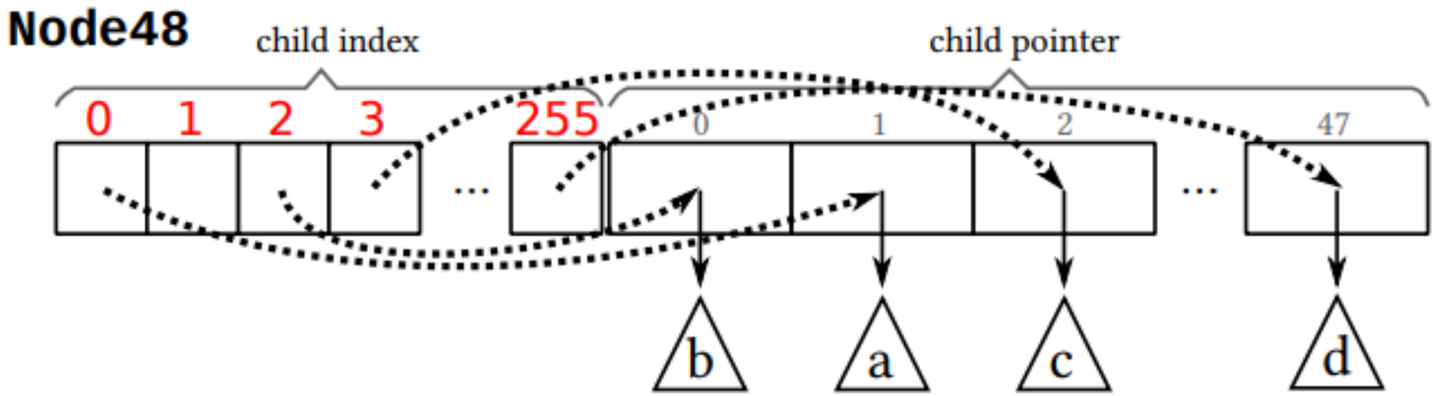


16 + 16 * 8 = 144 B

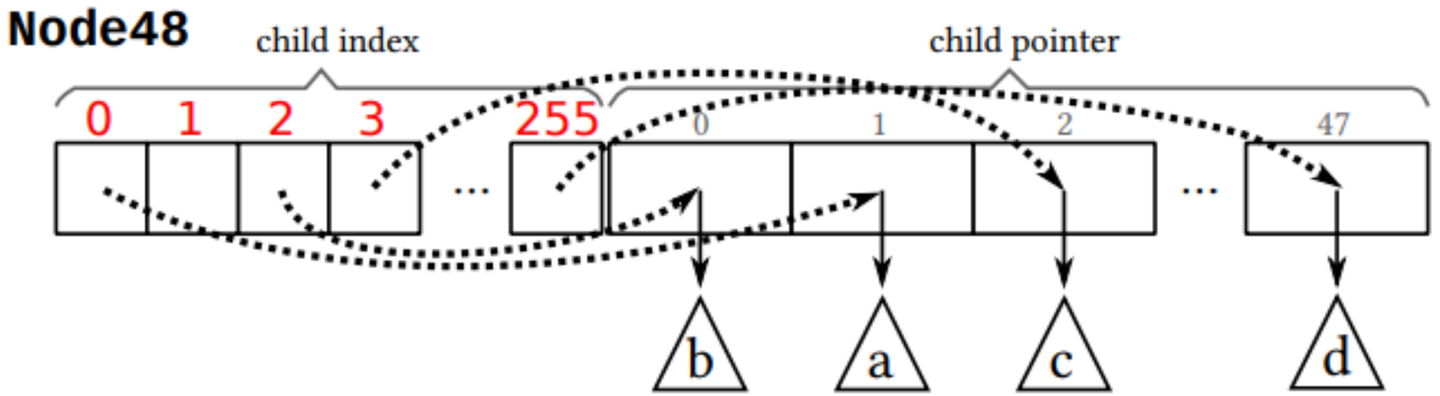
9 - 29 bytes per key

If node has 17-48 children

If node has 17-48 children

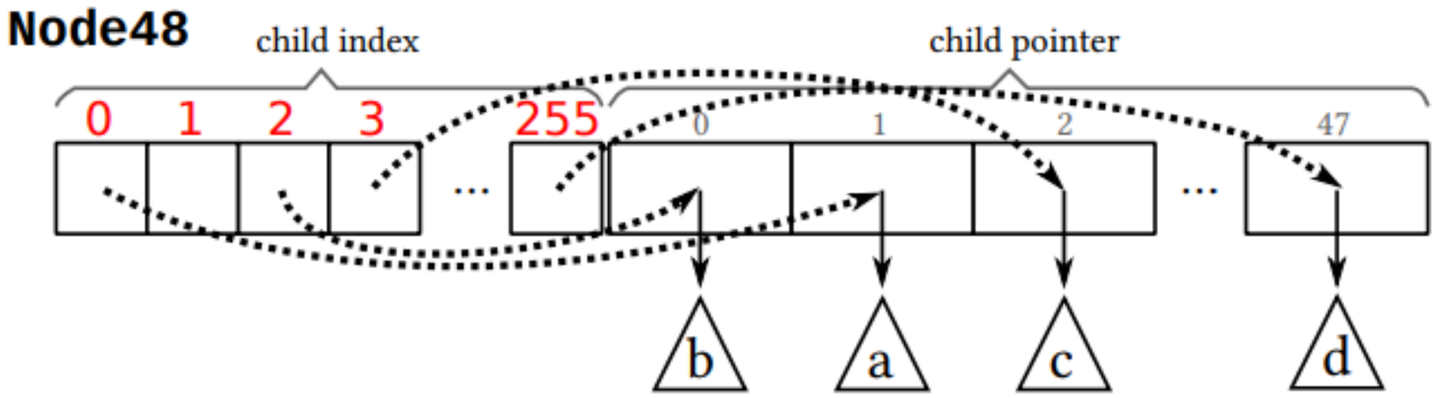


If node has 17-48 children



Too big for SIMD

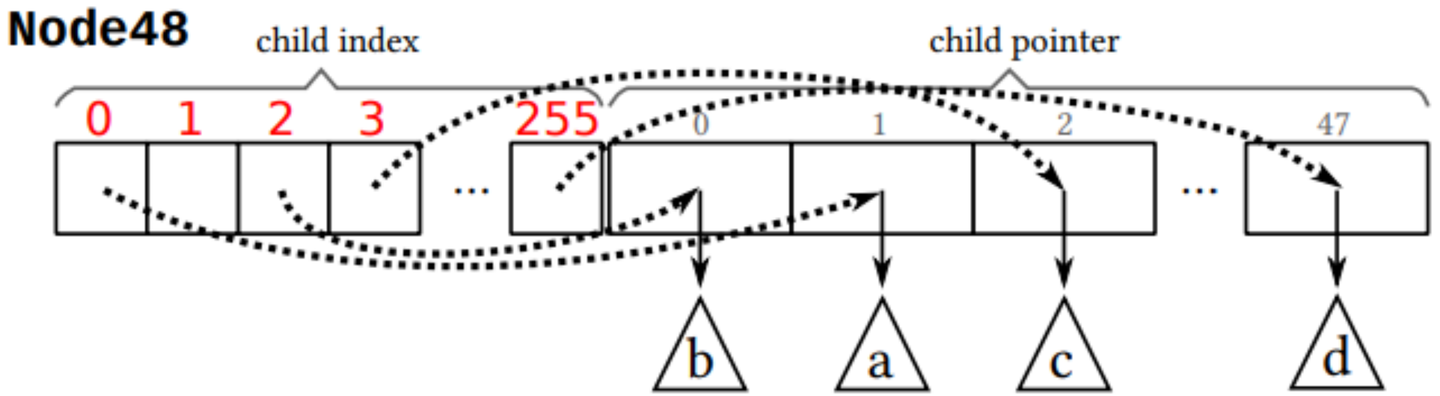
If node has 17-48 children



Too big for SIMD

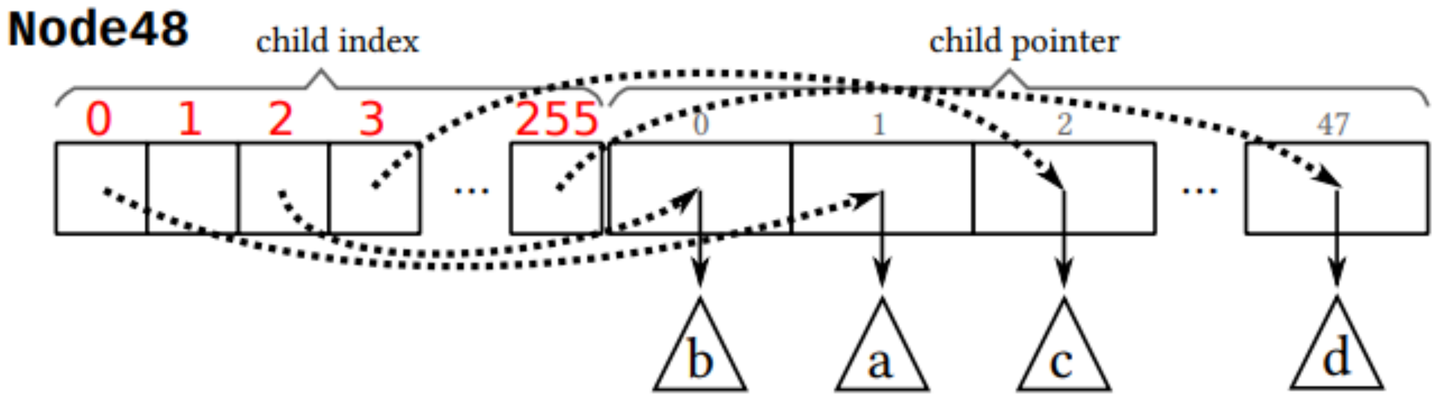
Use direct access

If node has 17-48 children



256 + 48 * 8 = 640 B

If node has 17-48 children

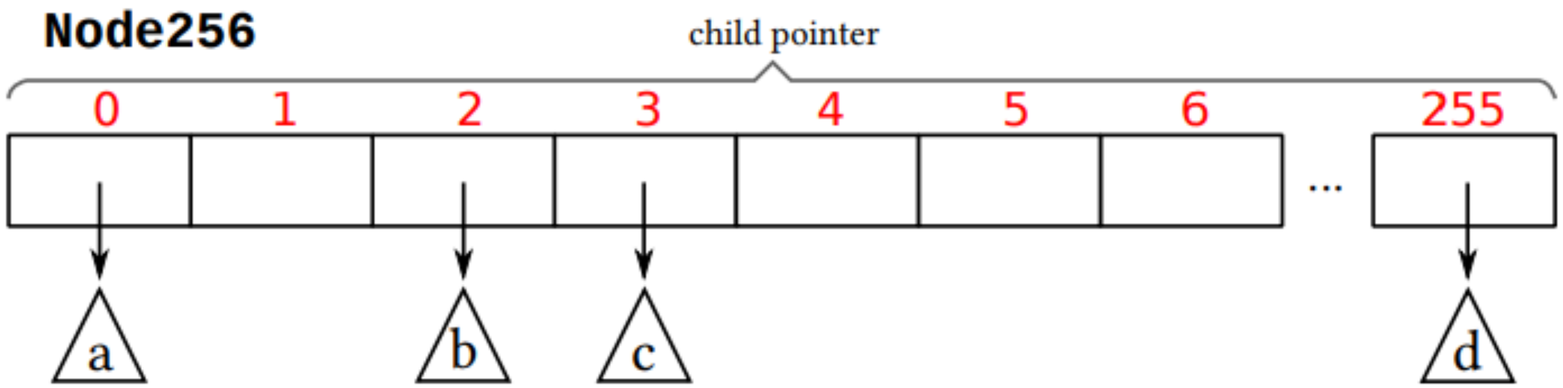


256 + 48 * 8 = 640 B

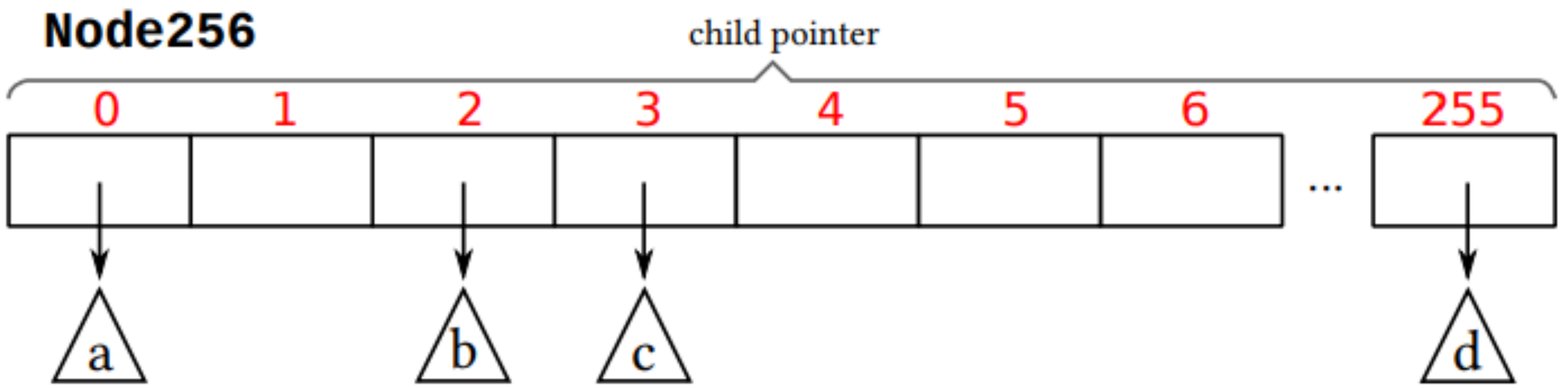
13 - 38 bytes per key

If node has 49-256 children

If node has 49-256 children

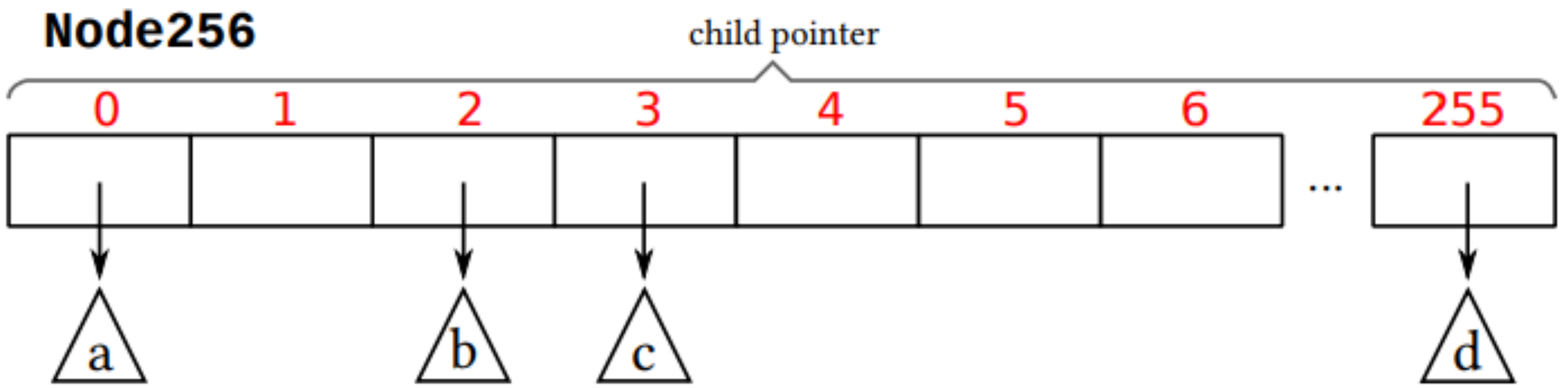


If node has 49-256 children



256 * 8 = 2048 B

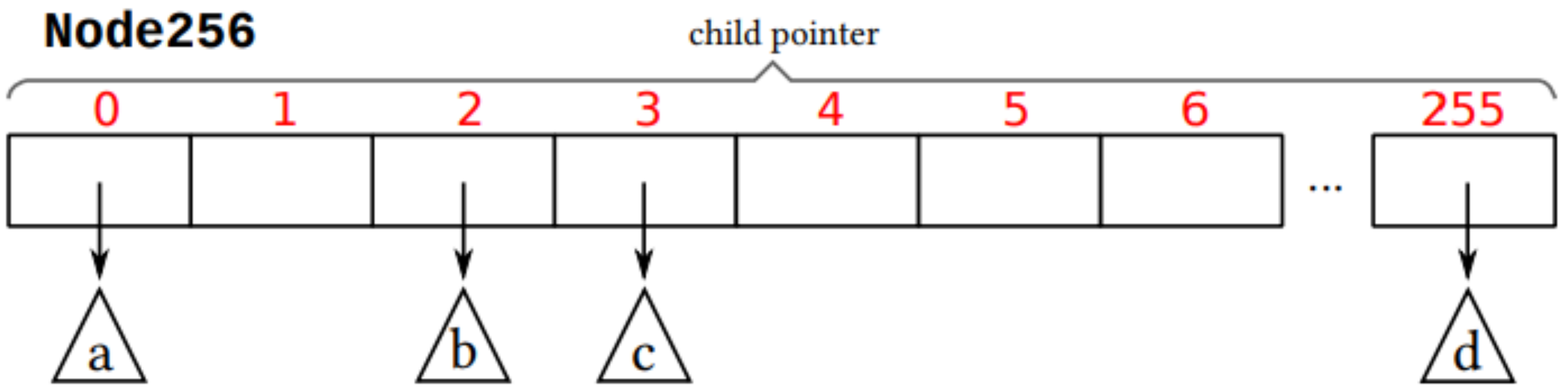
If node has 49-256 children



256 * 8 = 2048 B

8 - 42 bytes per key

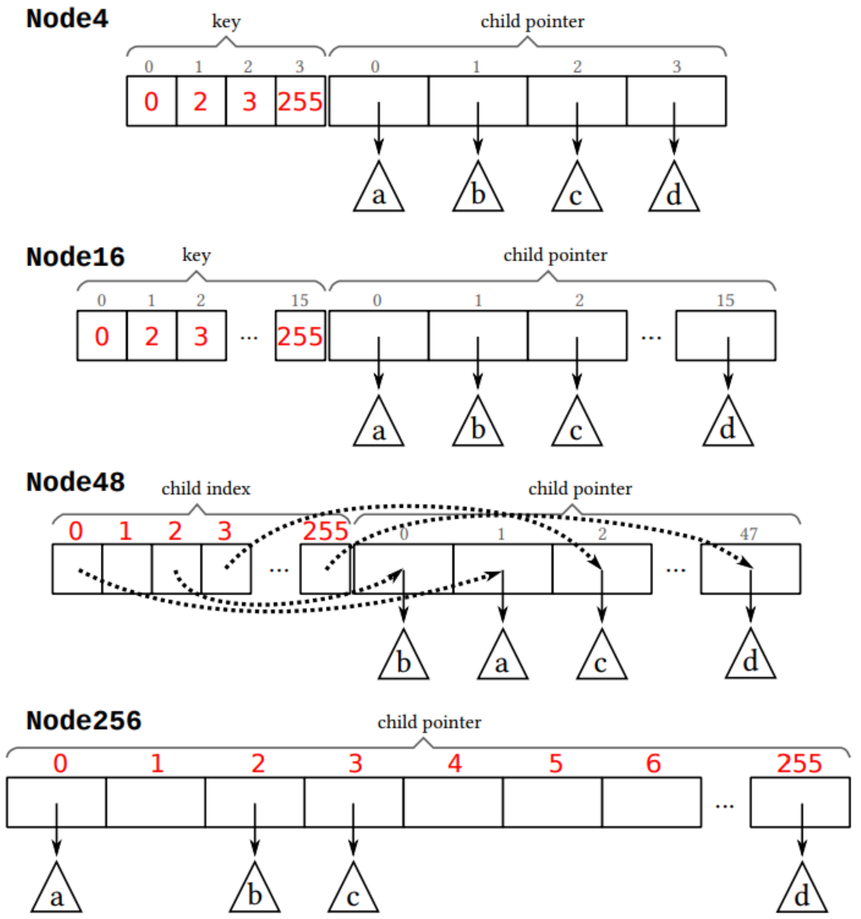
If node has 49-256 children



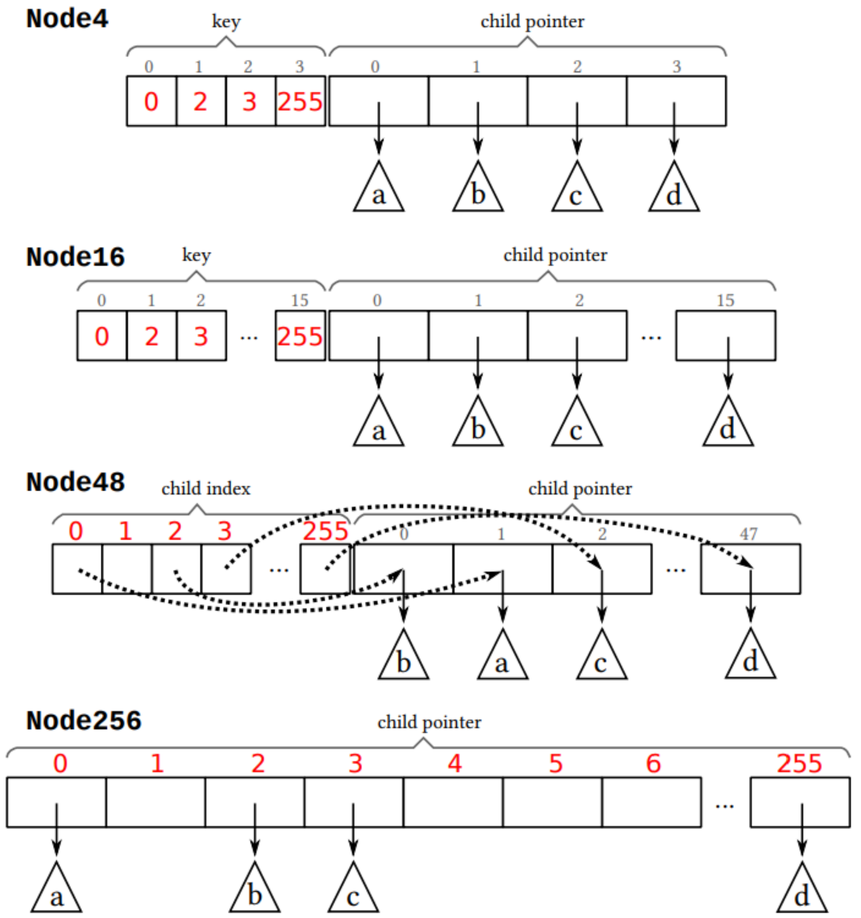
256 * 8 = 2048 B

8 - 42 bytes per key

Why only 4 node types?

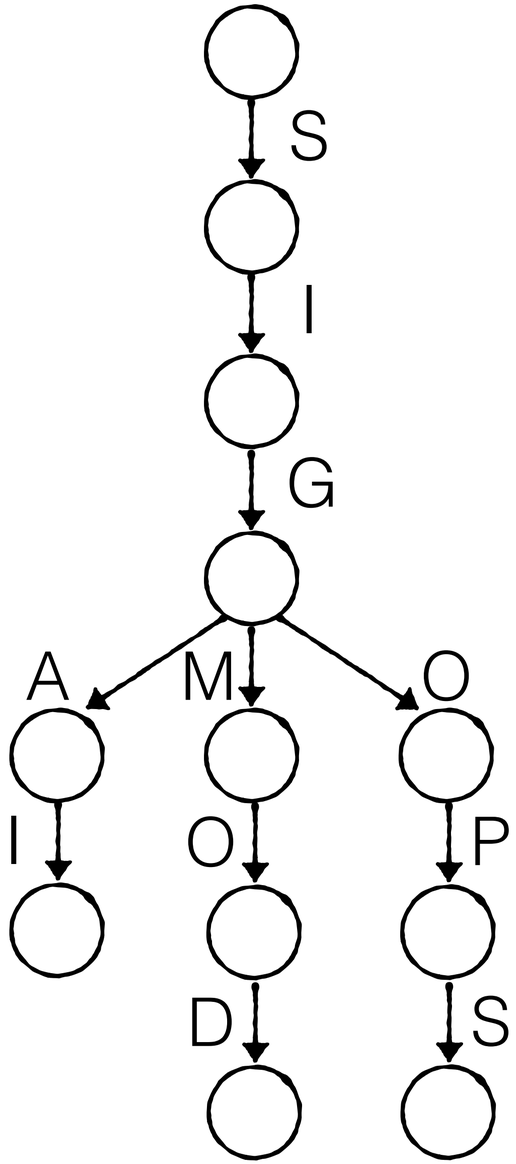


Why only 4 node types?

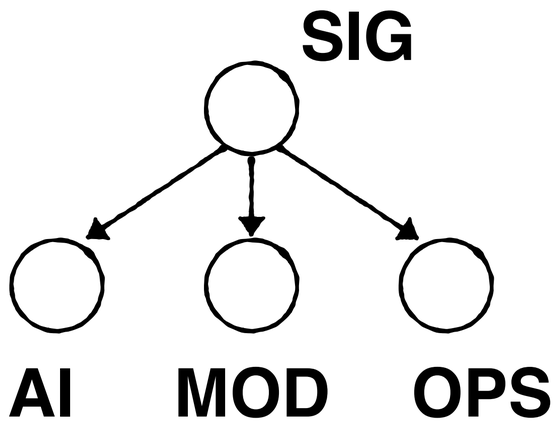


Switching a node type takes work. We want to do it sparingly.

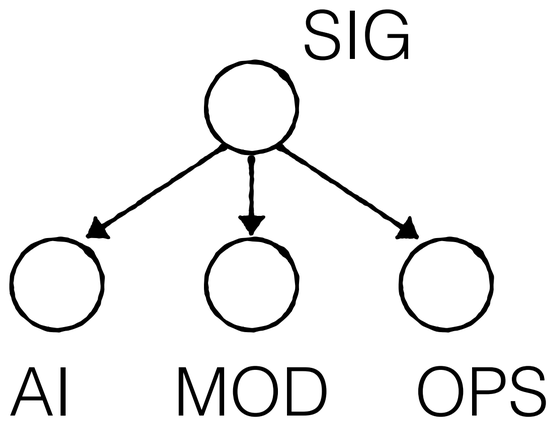
Path compression



Path compression

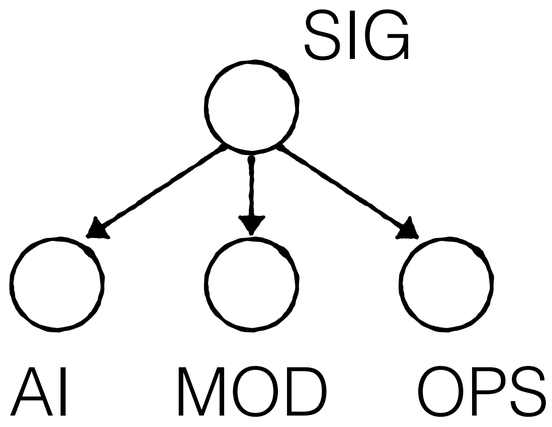


Path compression



How to store paths for a node?

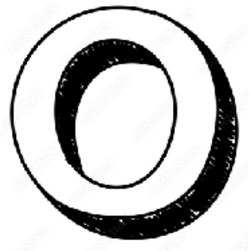
Path compression



How to store paths for a node?

In a 16 header at the start of each node

Better Worst-Case



AVL-Tree
1962

B-Tree
1970

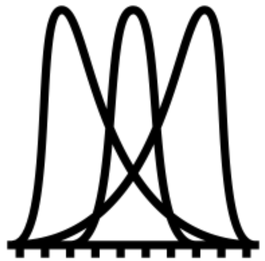
T-Tree
1985

CSS-Tree
1998

CSB-Tree
2000

ART
2013

Exploiting Data Distribution



Interpolation search
1959

FITing-Tree
2019


Universe U

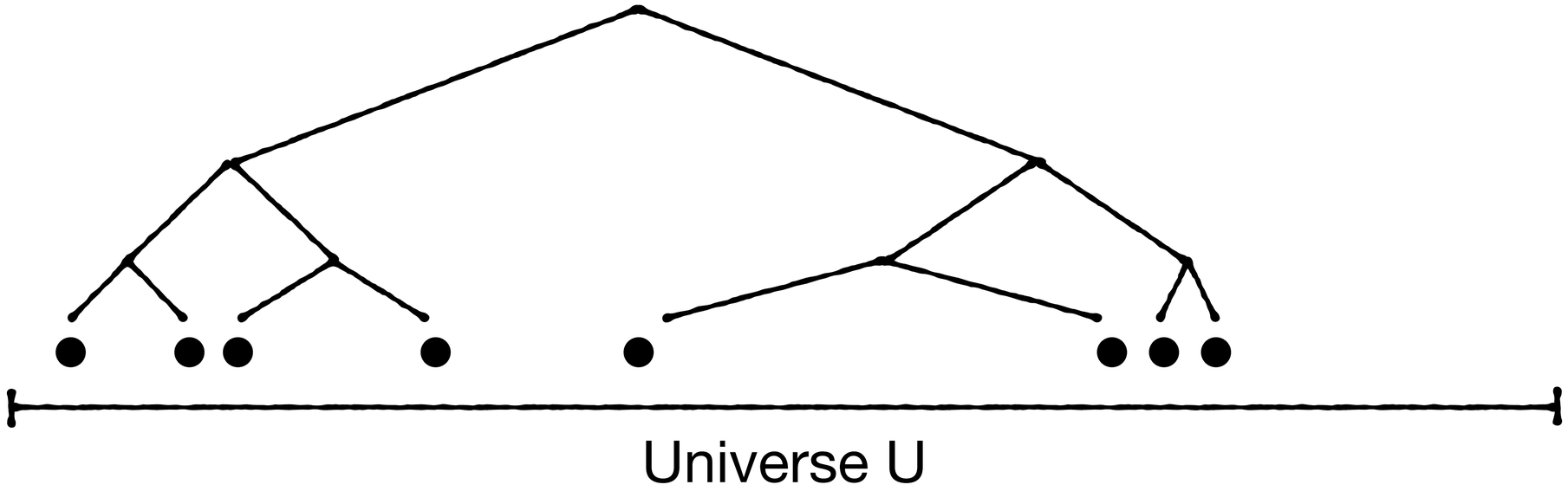
Binary search

$O(\log_2 N)$

Tree search

$O(\log_2 N)$ cycles

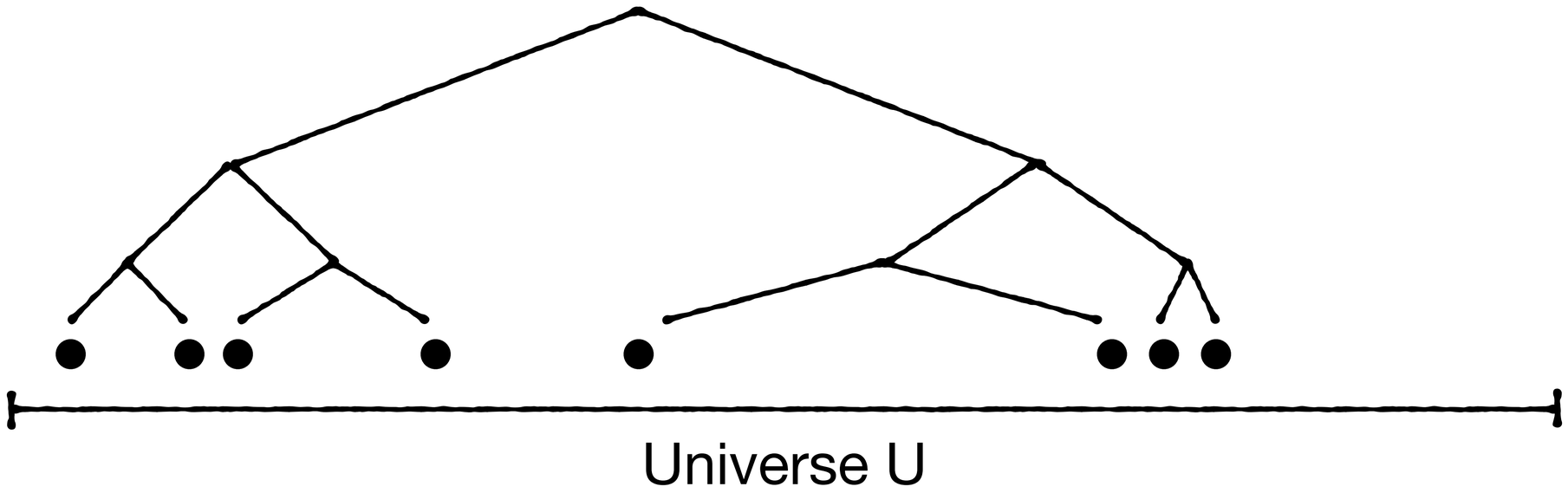
$O(\log_B N)$ I/O



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

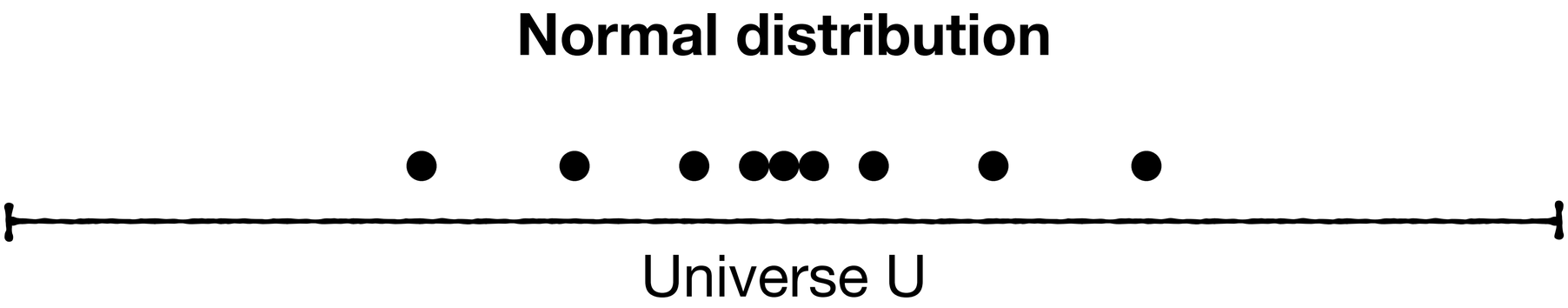
Universe U

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?

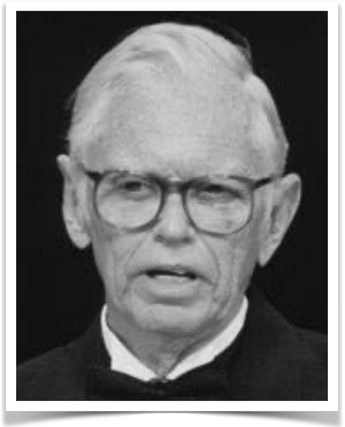
Bi-modal distribution

Universe U

Interpolation Search

Addressing for Random-Access Storage

IBM Journal of Research and Development. 1959



W. Wesley Peterson

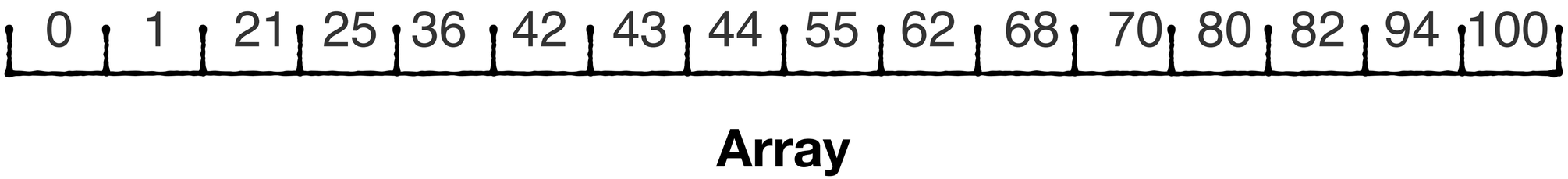
Interpolation Search

Uniform distribution

Interpolation Search

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

Interpolation Search



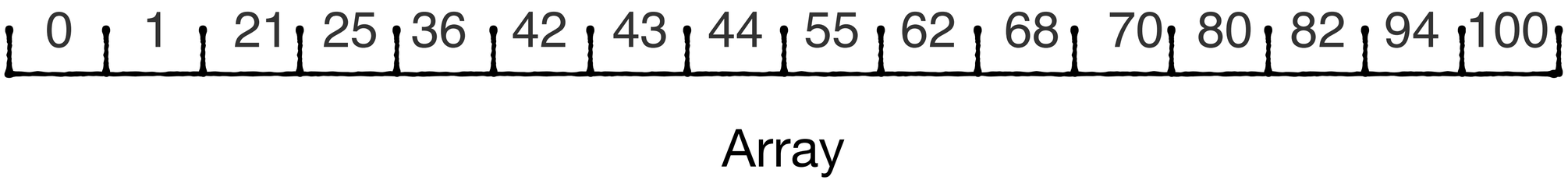
Interpolation Search

get(X)

Array

get(X)

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

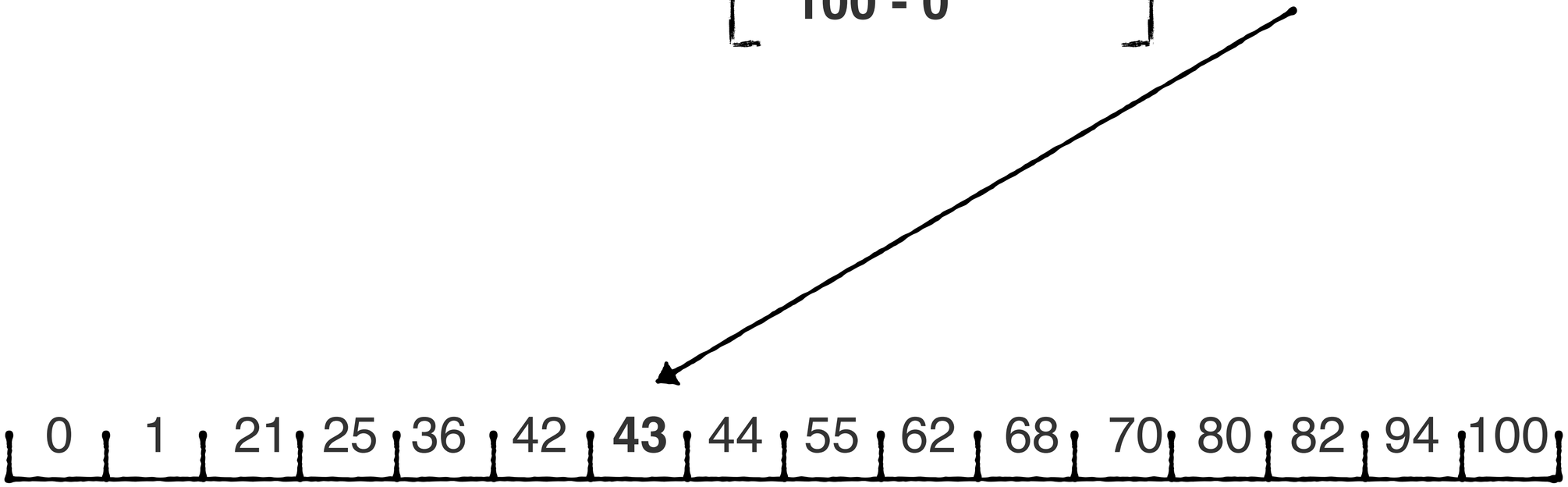


get(42)

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

get(42)

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



get(42)

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

Recurse on this partition

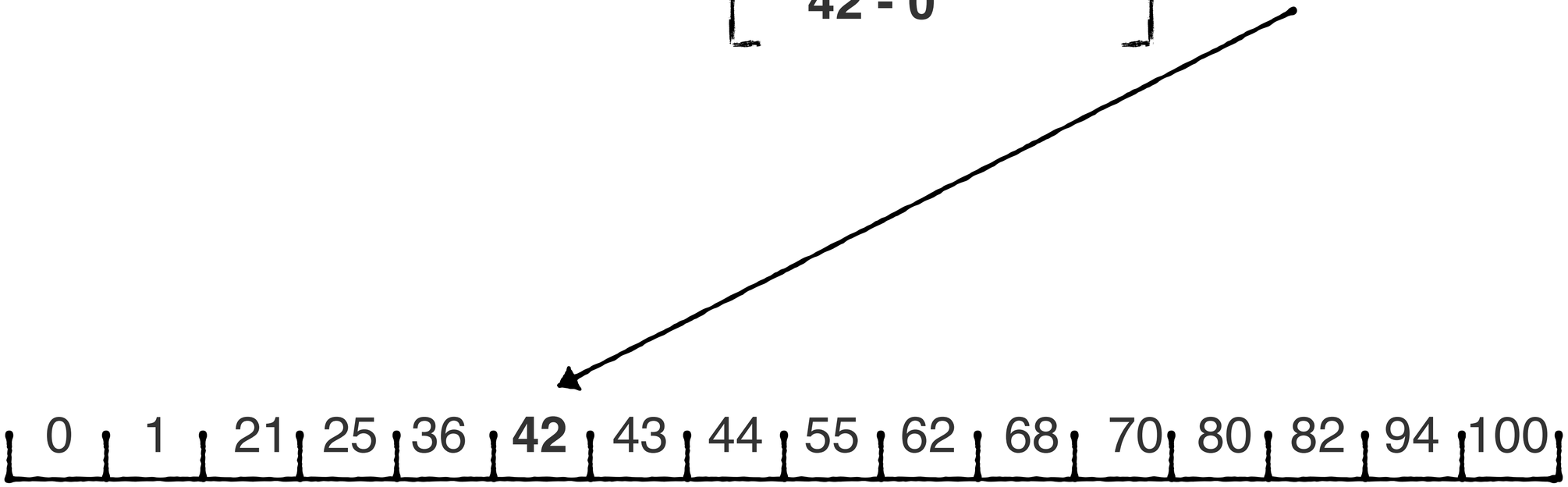
get(42)

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

Recurse on this partition

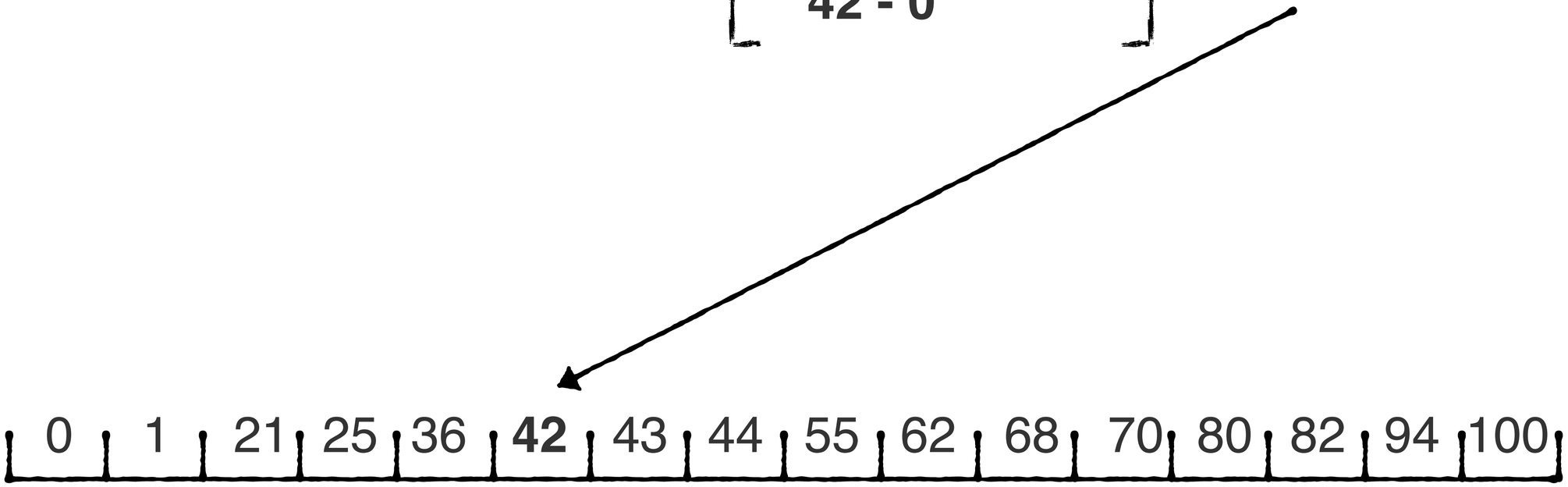
get(42)

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



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

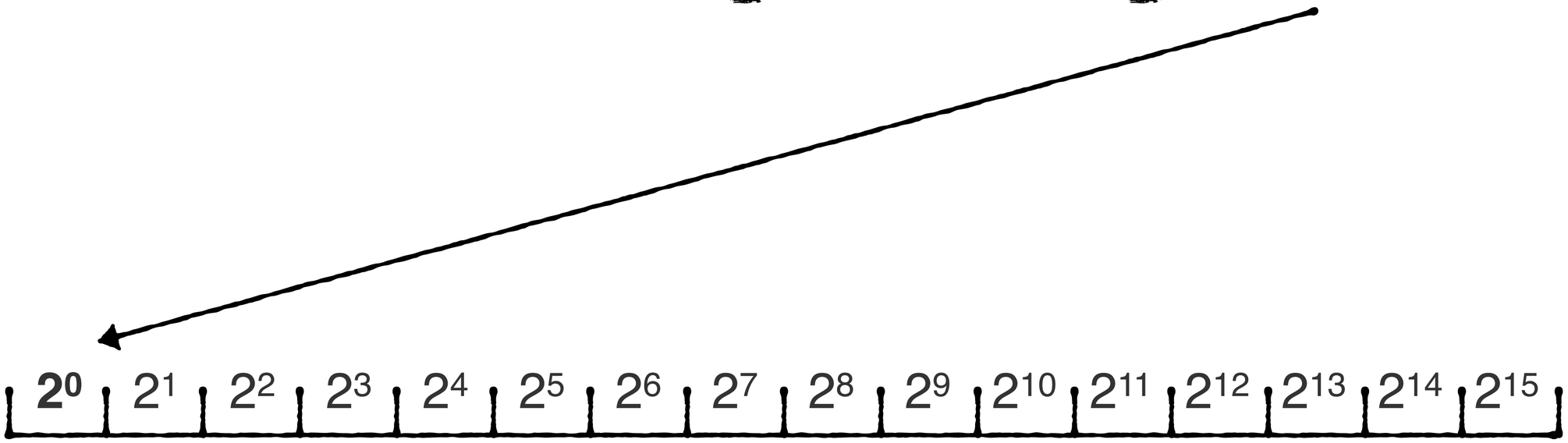
$$\textbf{Estimated location} = \left\lfloor \frac{\textbf{X - min}}{\textbf{max - min}} \cdot \textbf{(\#slots - 1)} \right\rfloor$$

get(2^{11})

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

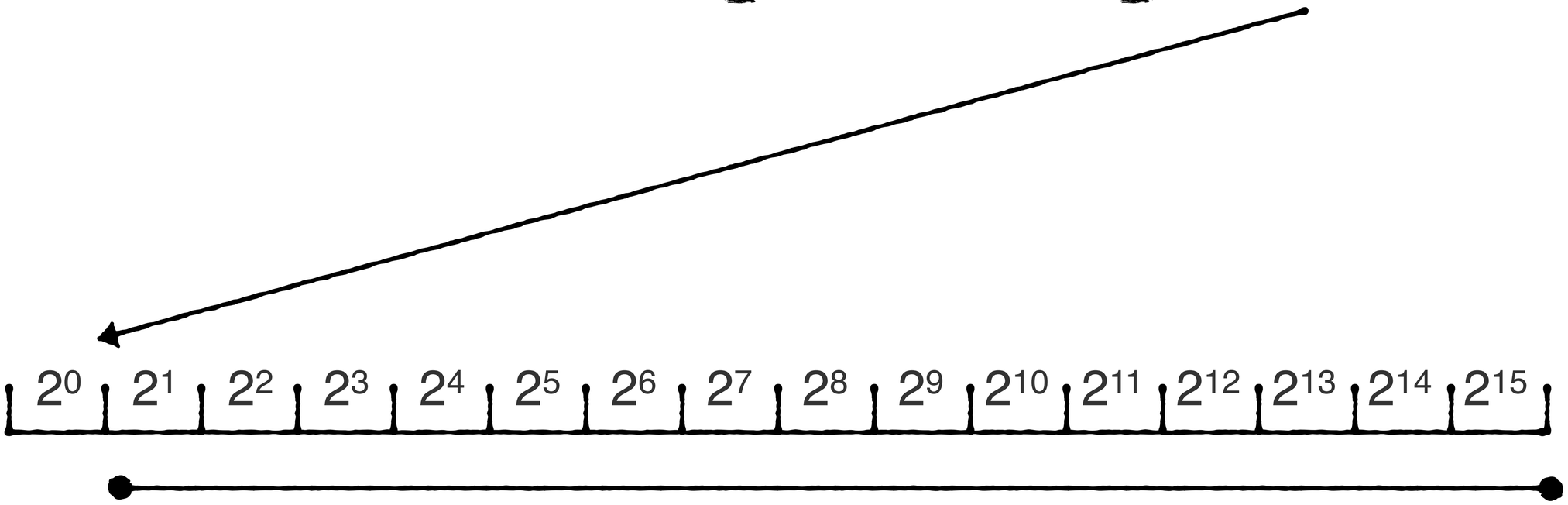
get(2^{11})

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



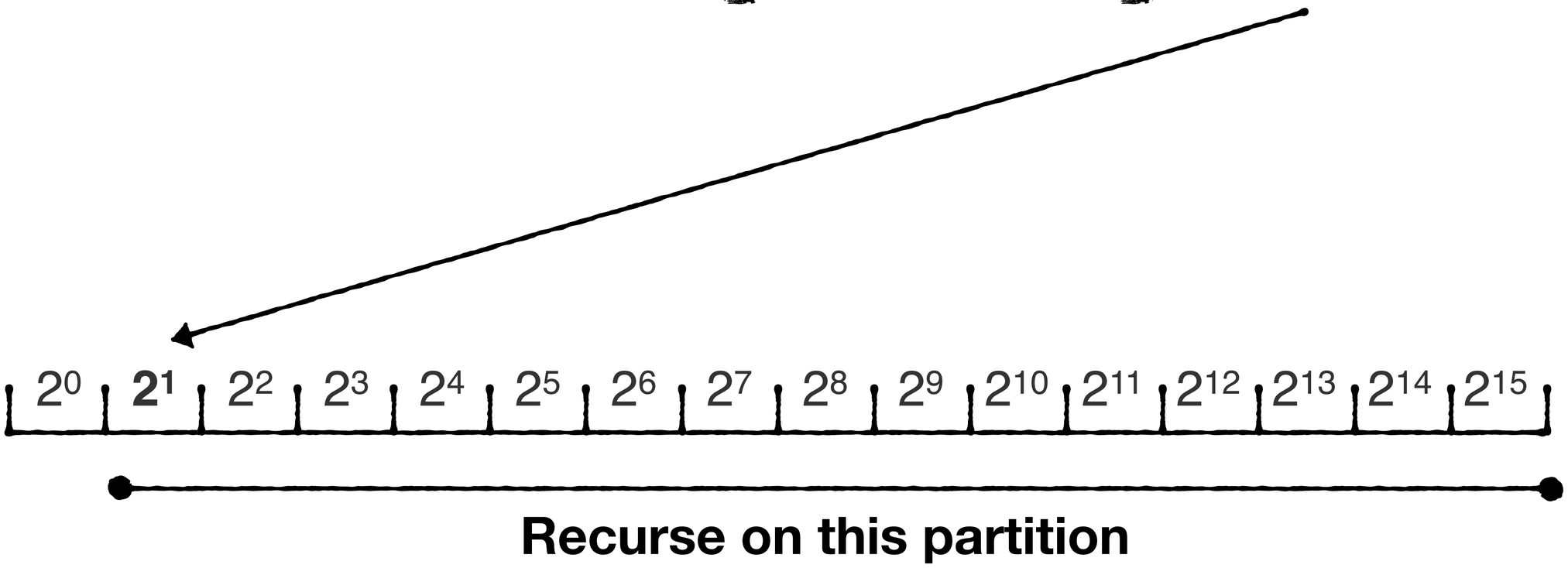
get(2^{11})

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



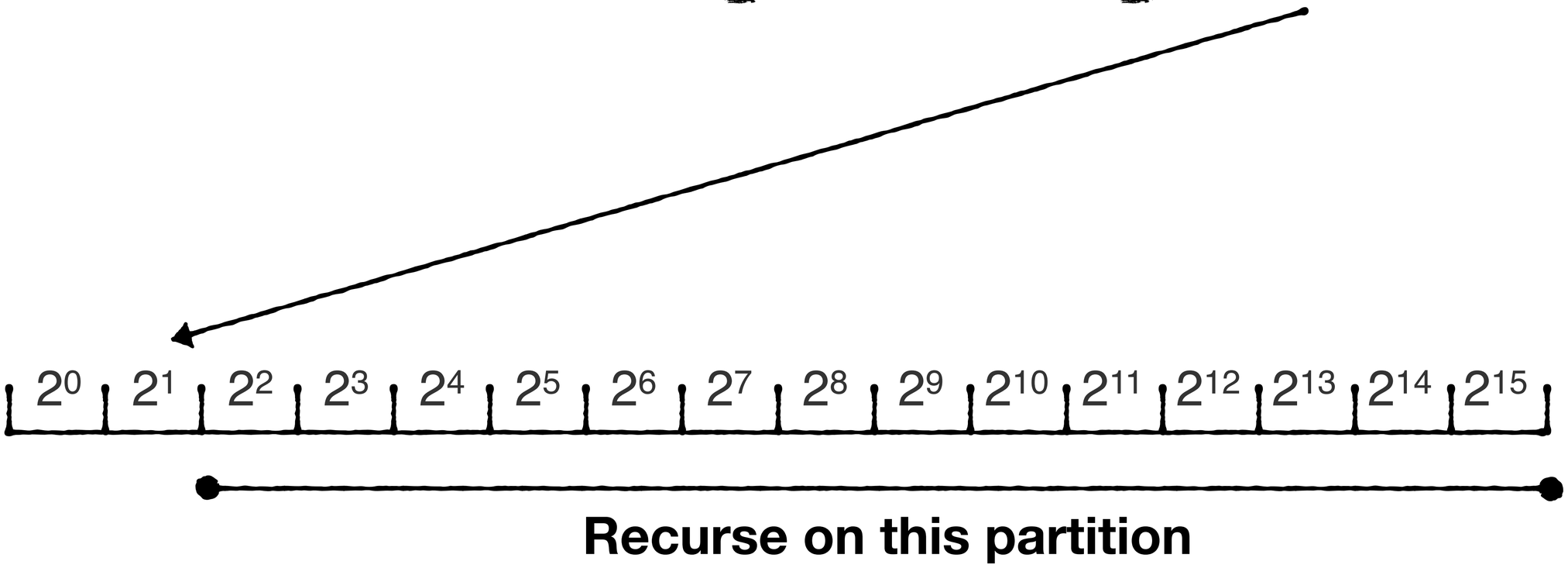
get(2^{11})

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



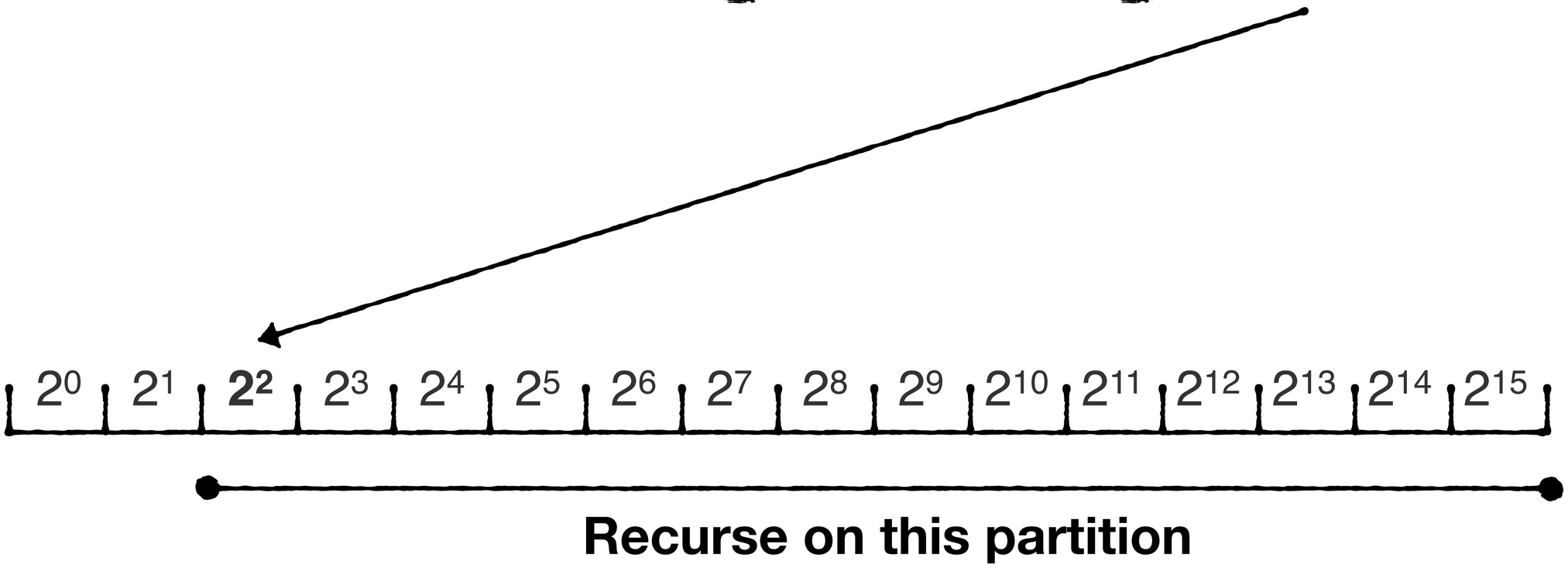
get(2^{11})

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



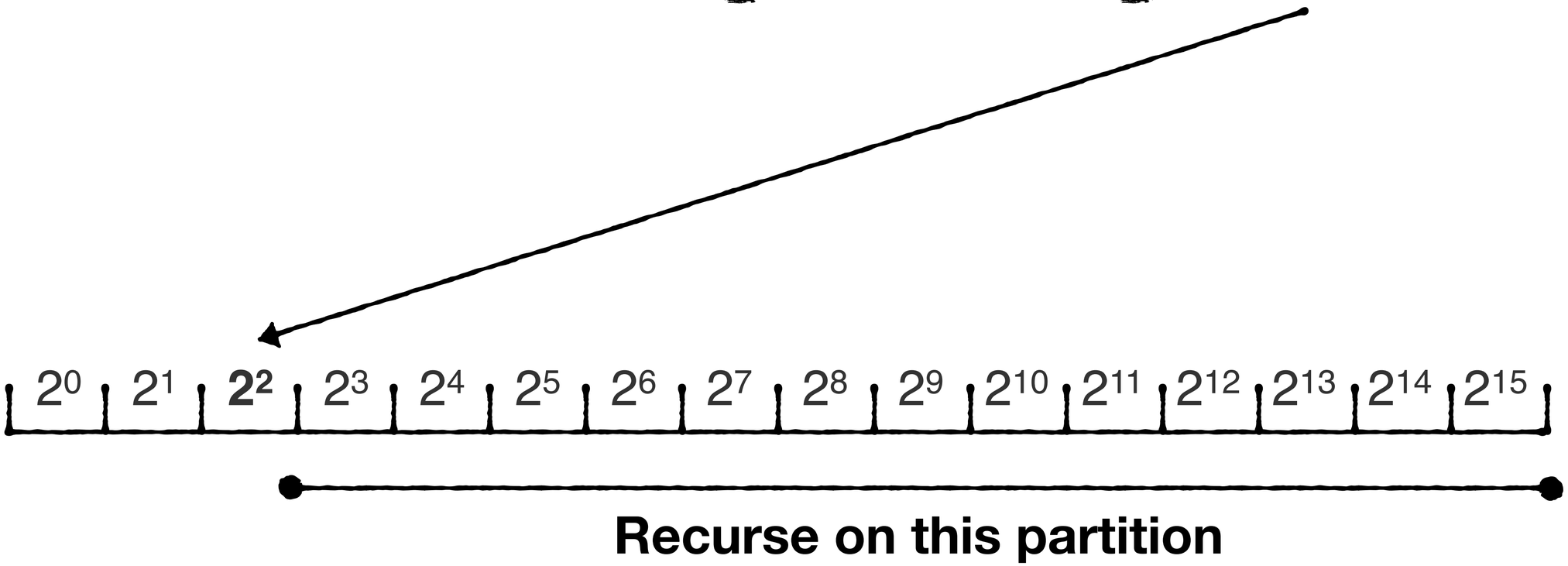
get(2^{11})

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



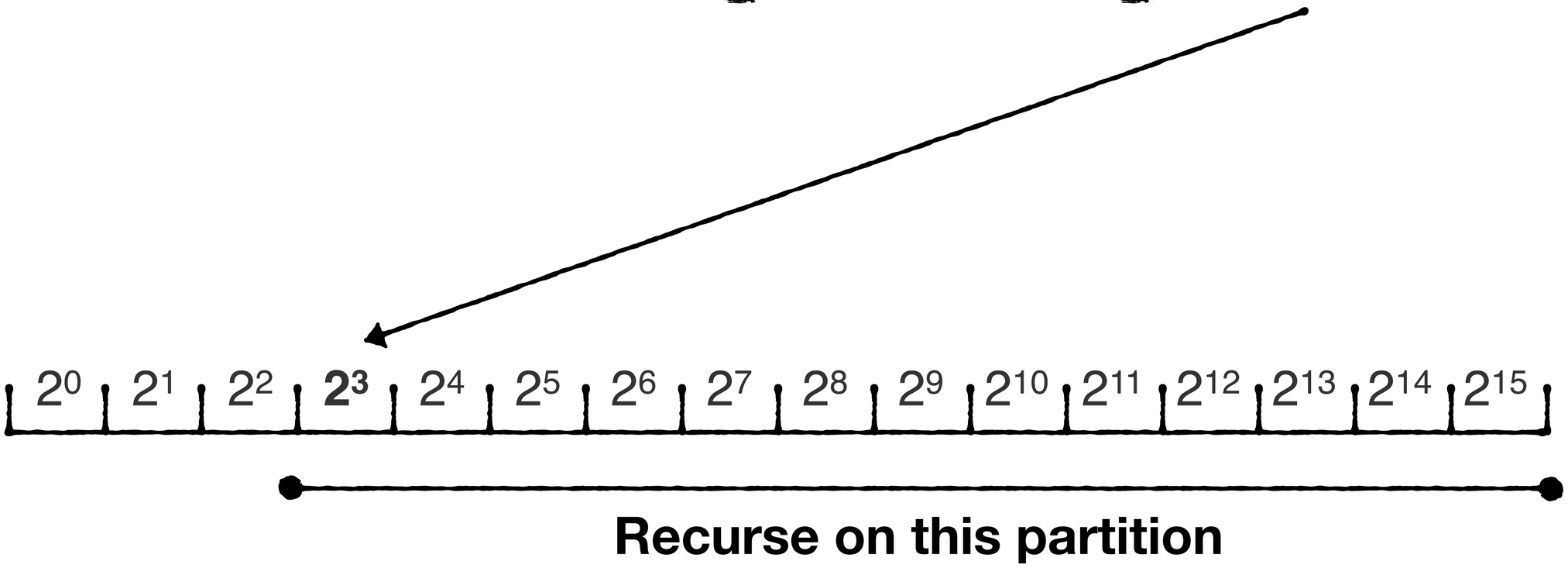
get(2^{11})

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



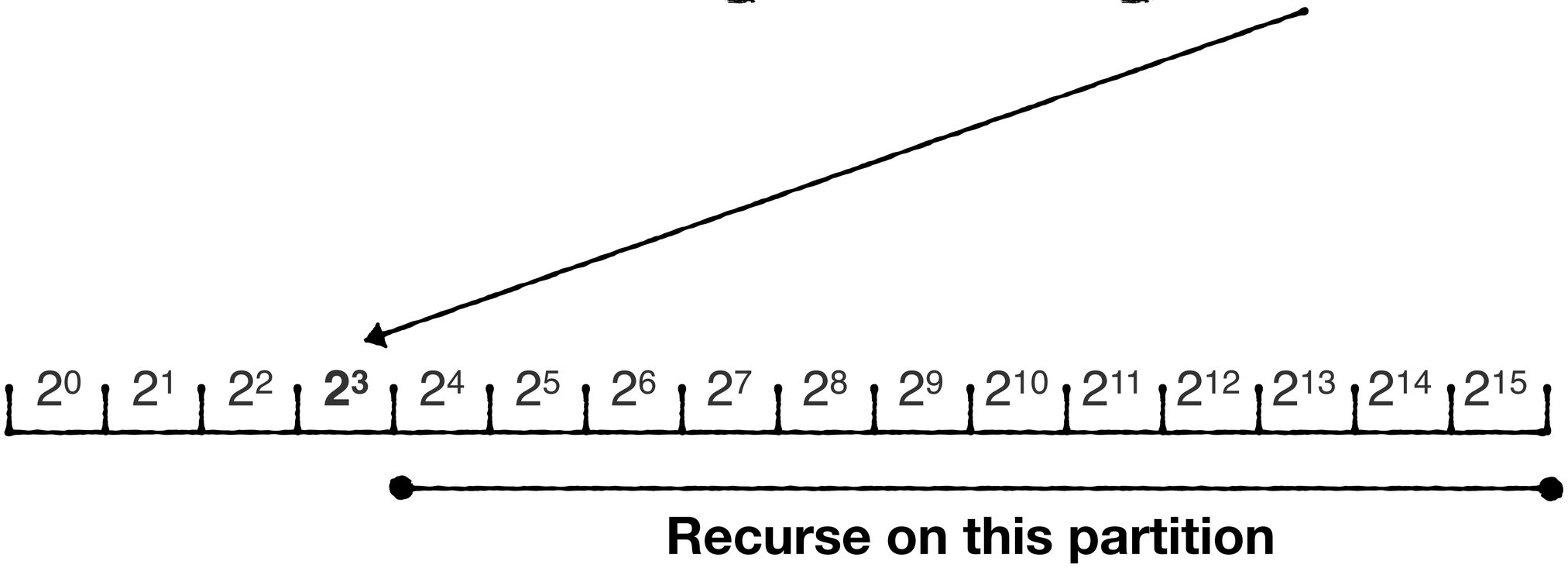
get(2^{11})

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



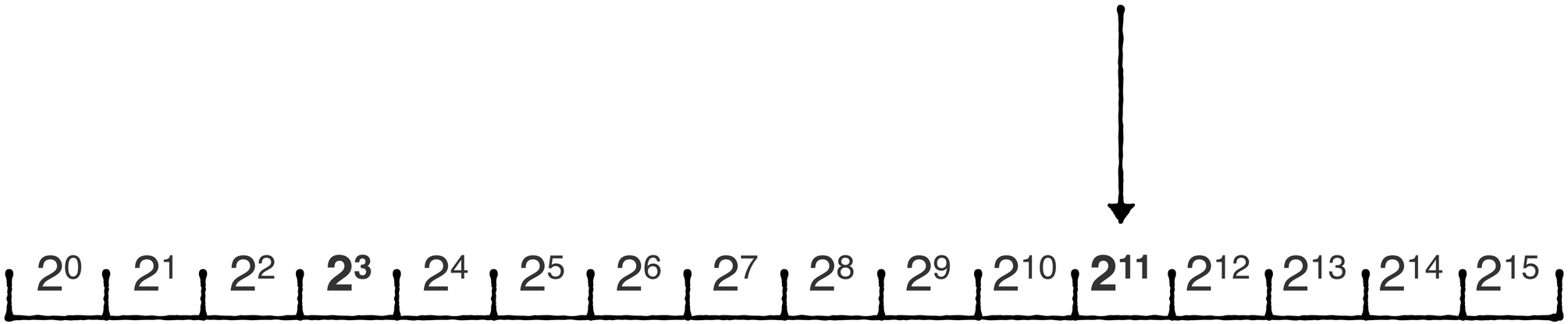
get(2^{11})

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



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.

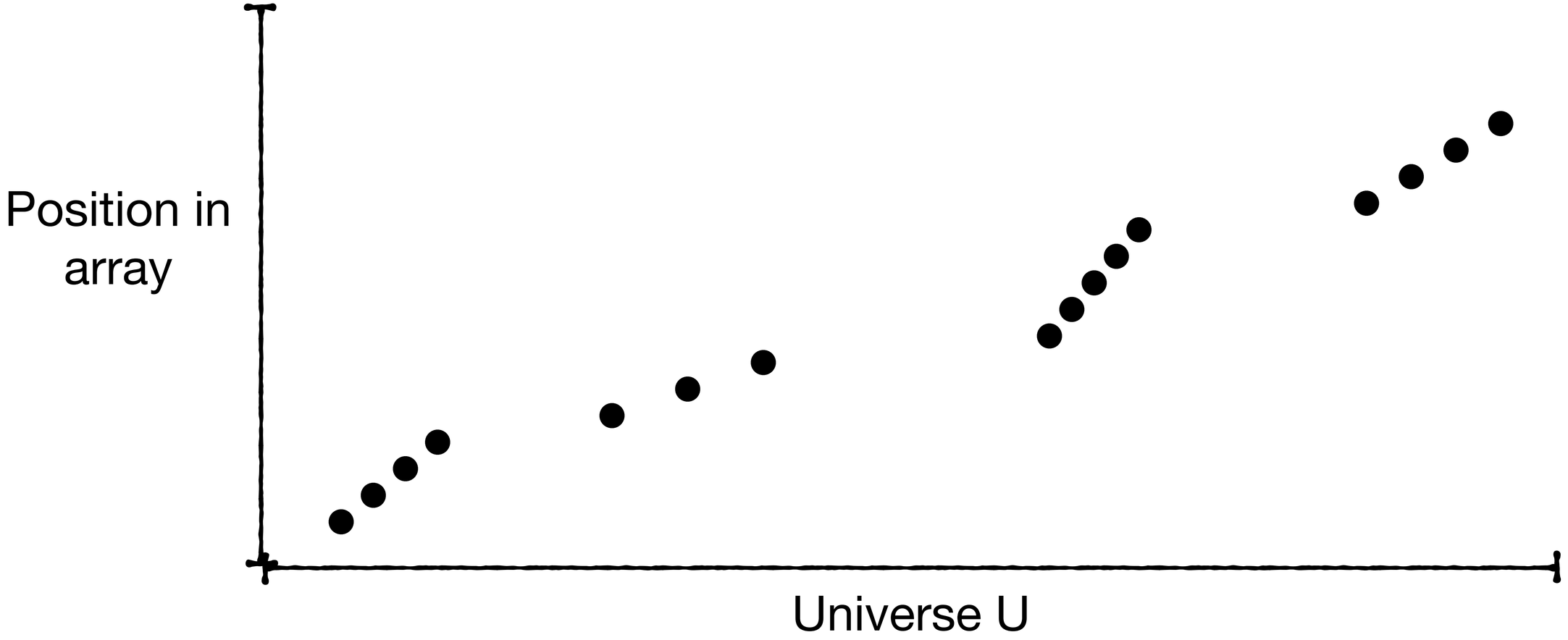
As the data grows, interpolation search is likely faster.

Interpolation search works well for uniform data

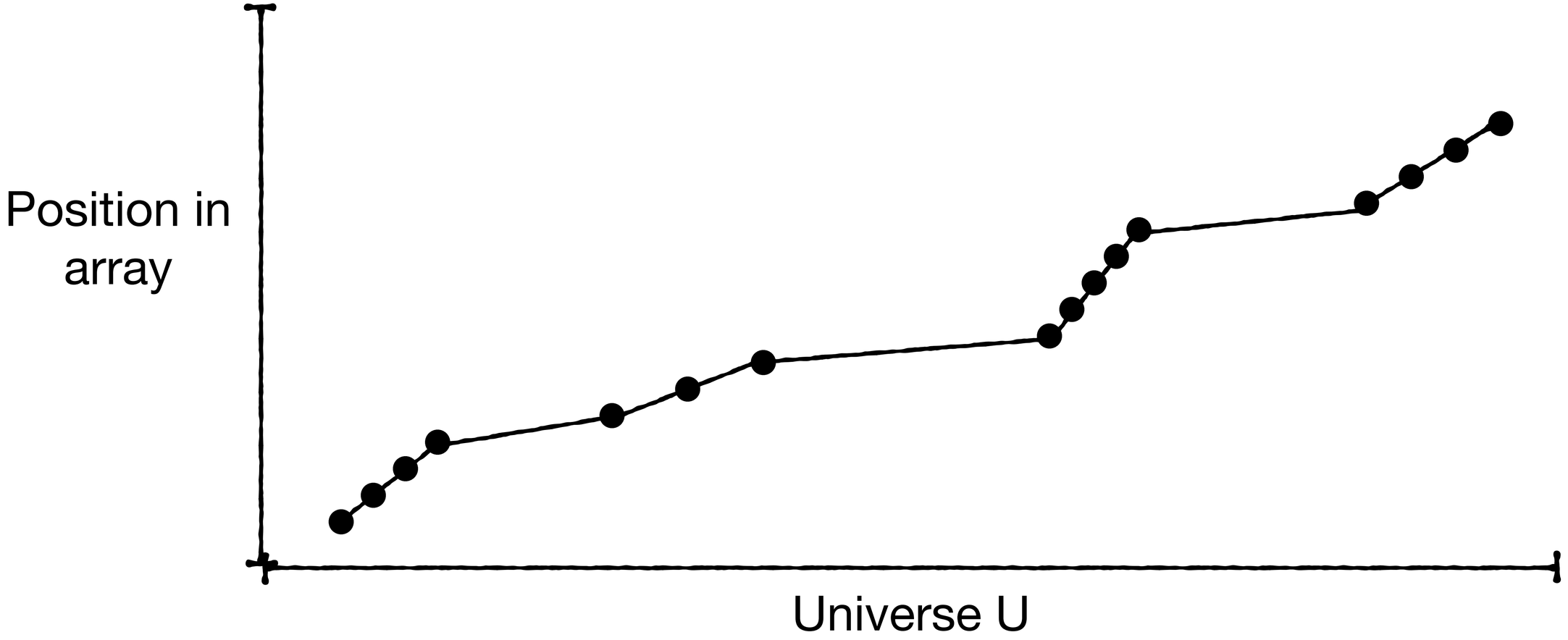
Interpolation search works well for uniform data

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

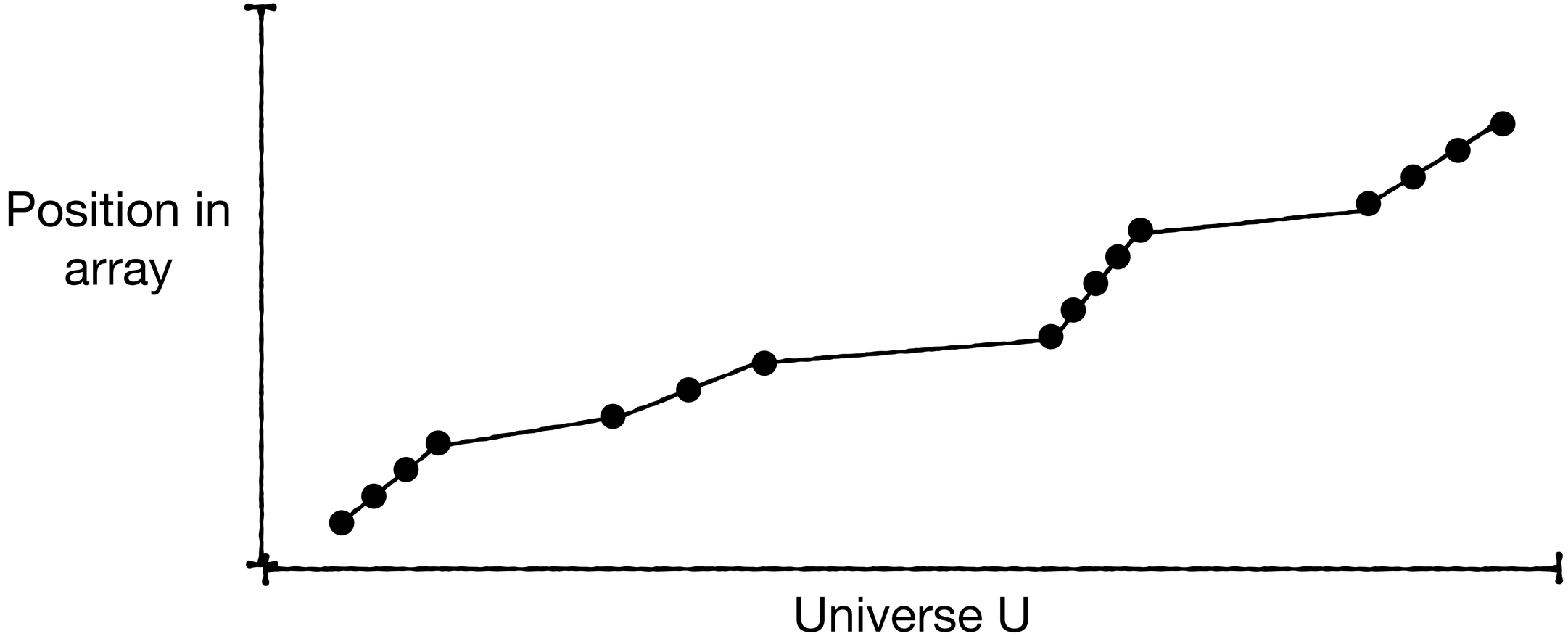
But data often follows idiosyncratic patterns


Position in array
Universe U

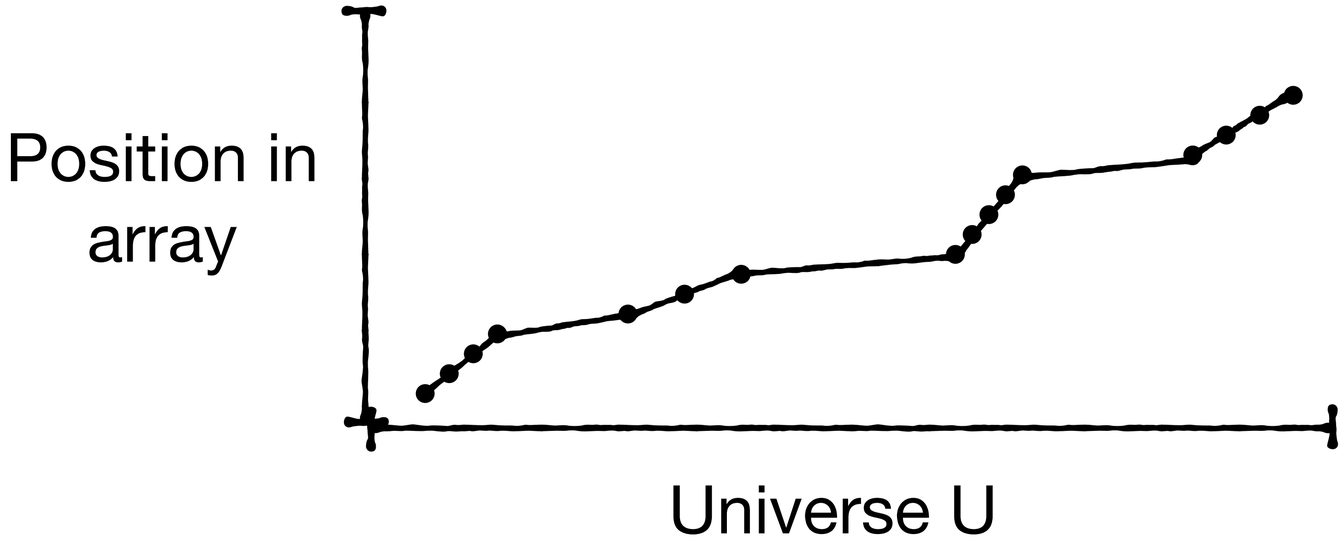
cumulative distribution function (CDF)

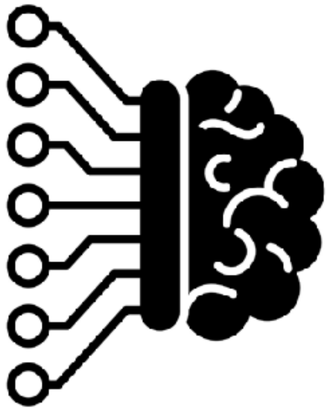


Insight: at each segment, it's easy to predict an entry's position

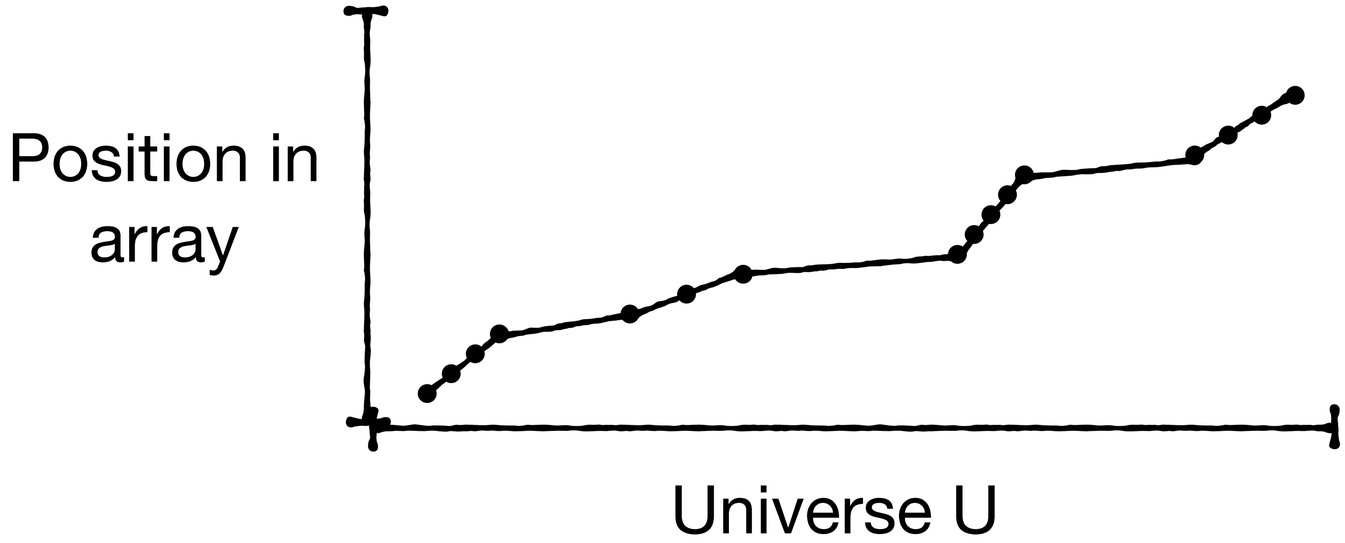


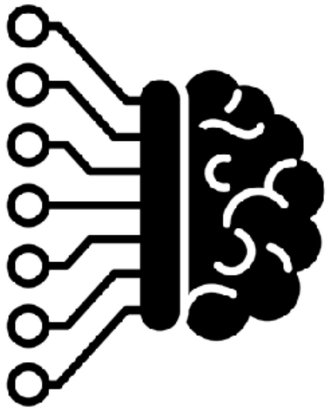
Learned Indexes: Learn the data distribution to predict an entry's position





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



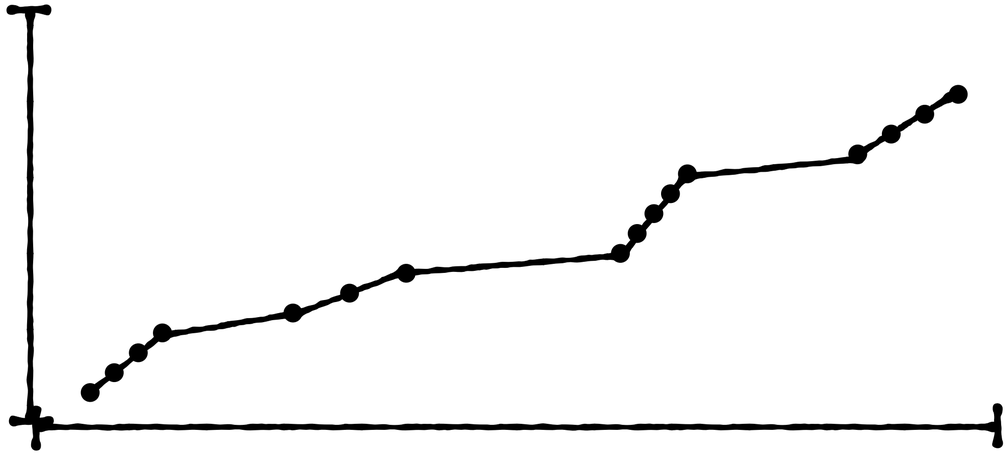


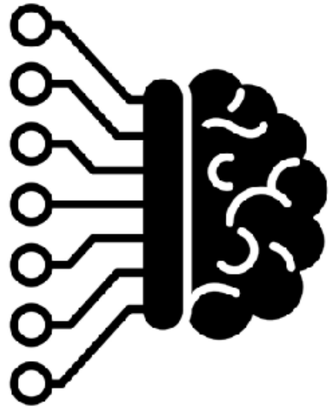
Partition the data into predictable segments

The Case for Learned Index Structures

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

SIGMOD 2018

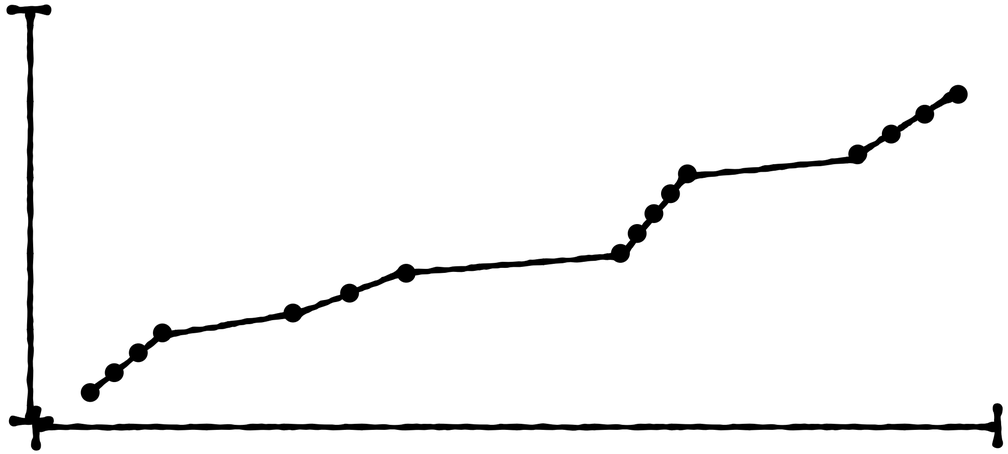


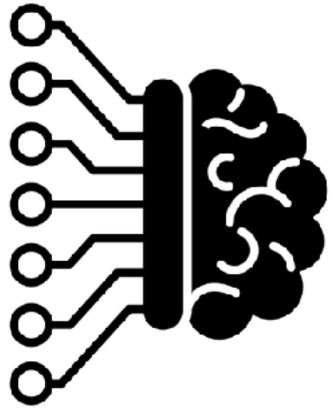


The Case for Learned Index Structures

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

SIGMOD 2018



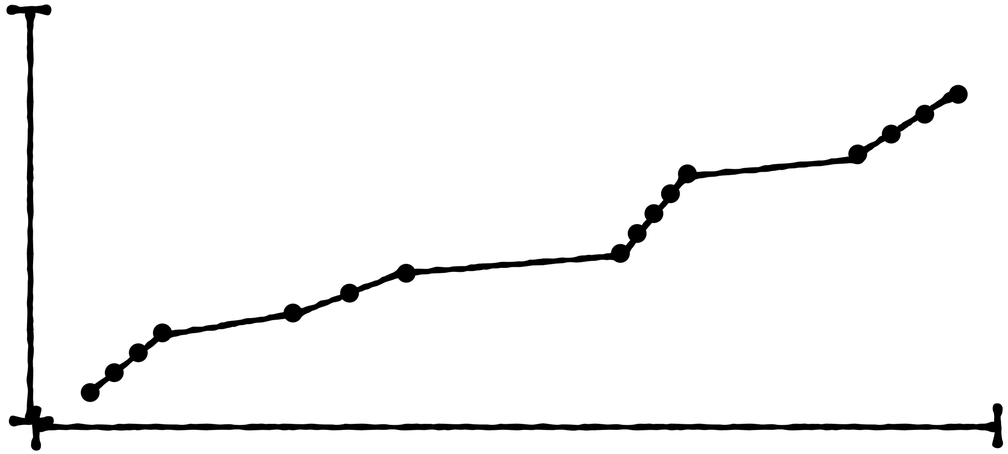


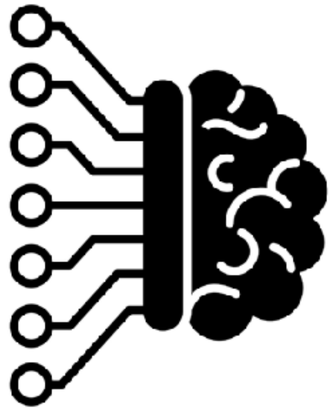
Vision paper, and algorithmic details can be vague

FITing-Tree: A Data-aware Index Structure

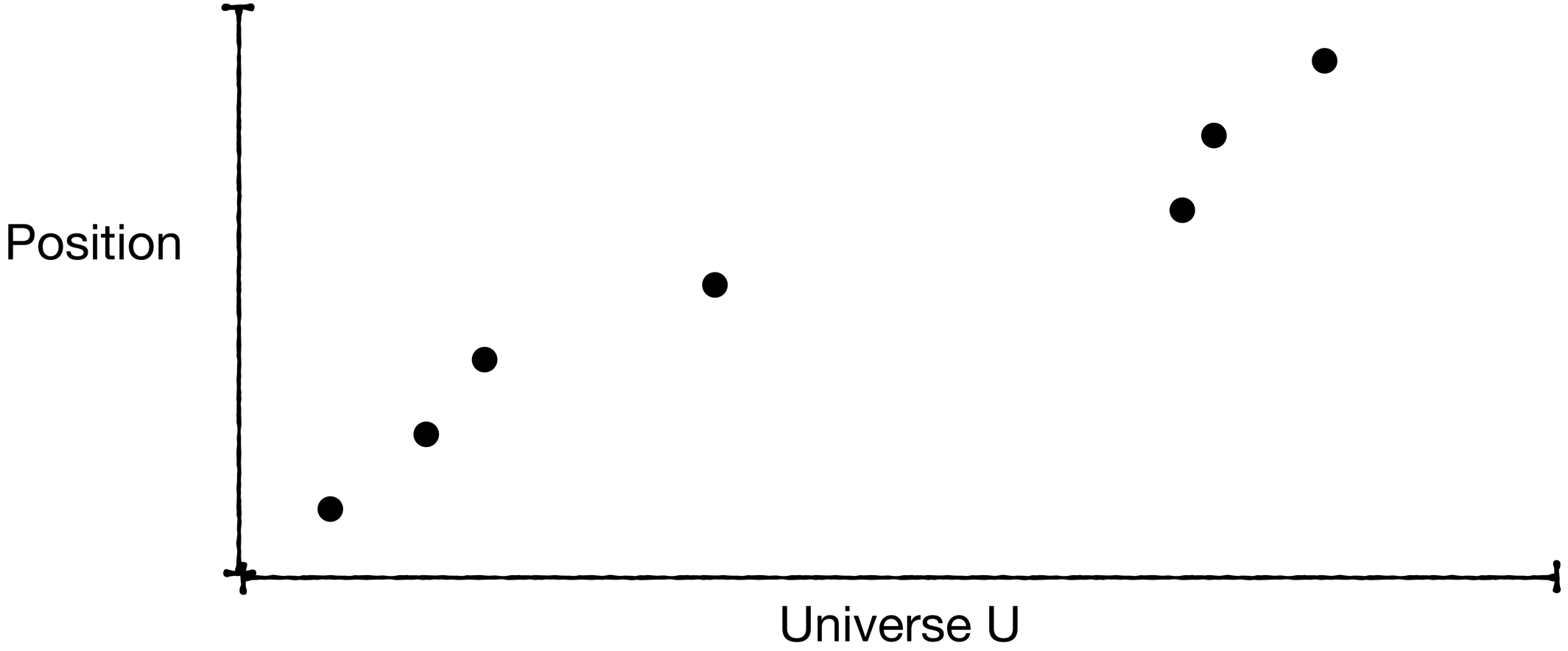
Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, Tim Kraska

SIGMOD 2019

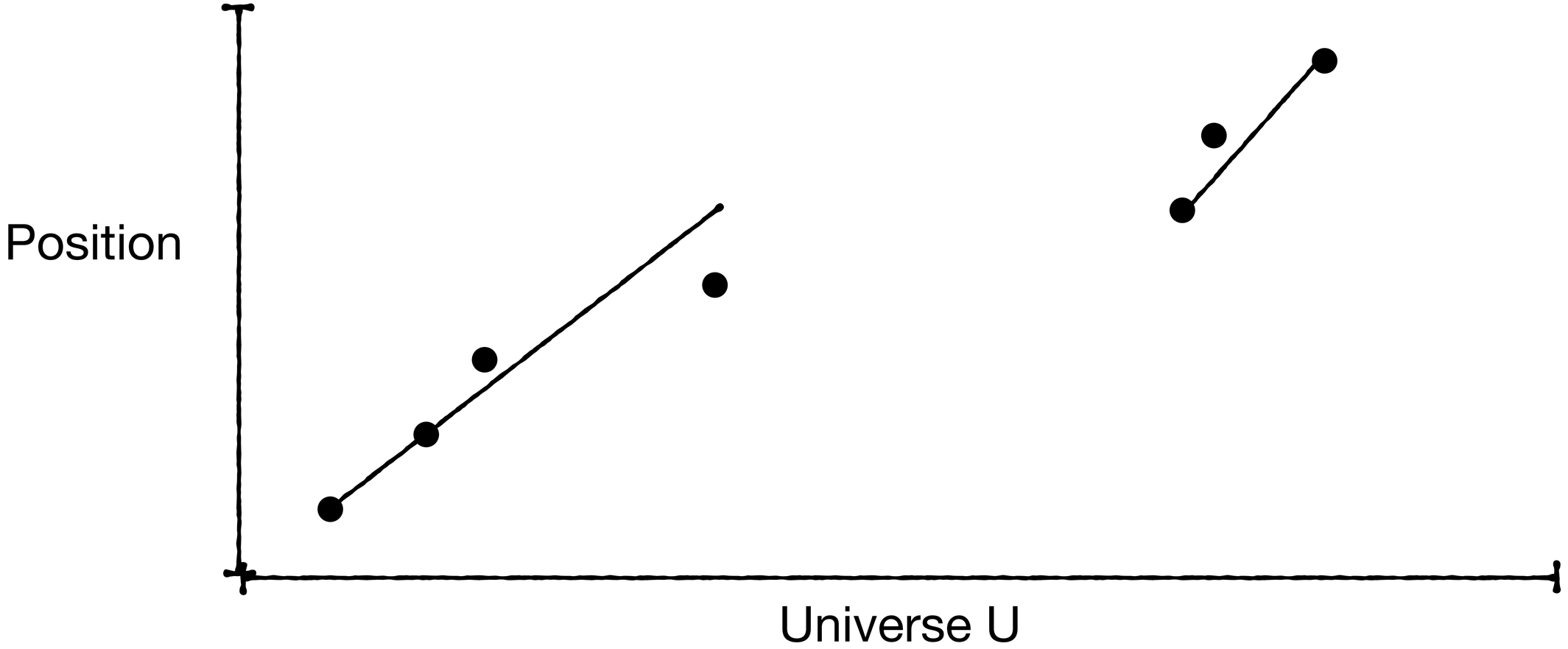




FITing-Tree: A Data-aware Index Structure

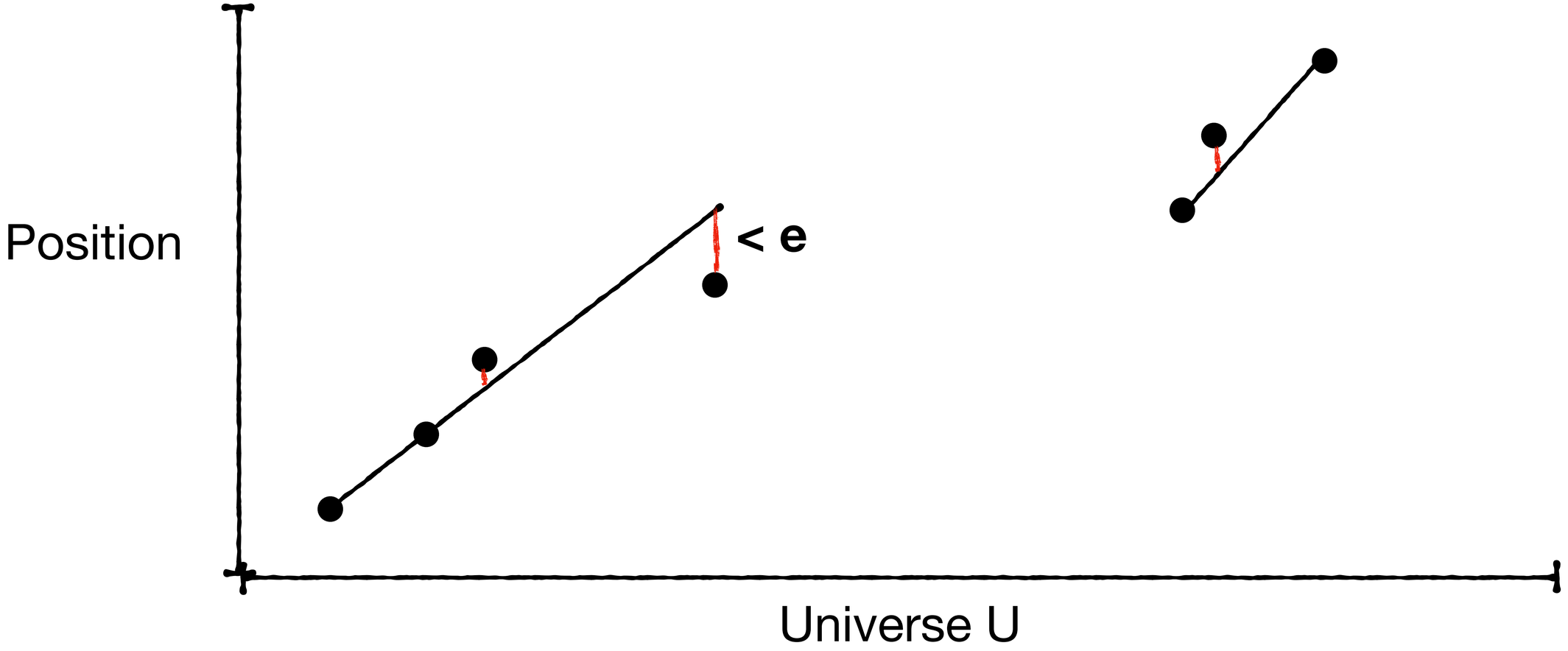


approximate distribution using piecewise linear functions



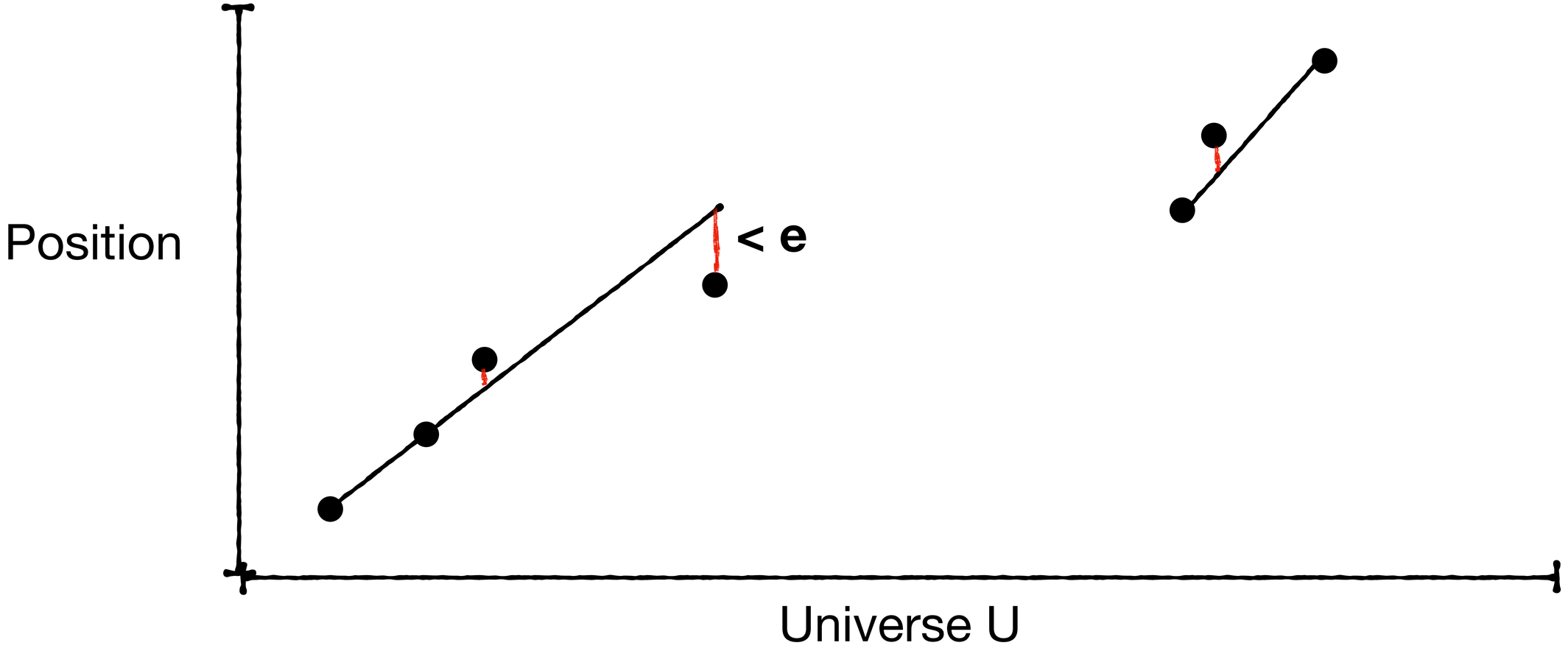
approximate distribution using piecewise linear functions

Ensure each point has max distance of e from its function

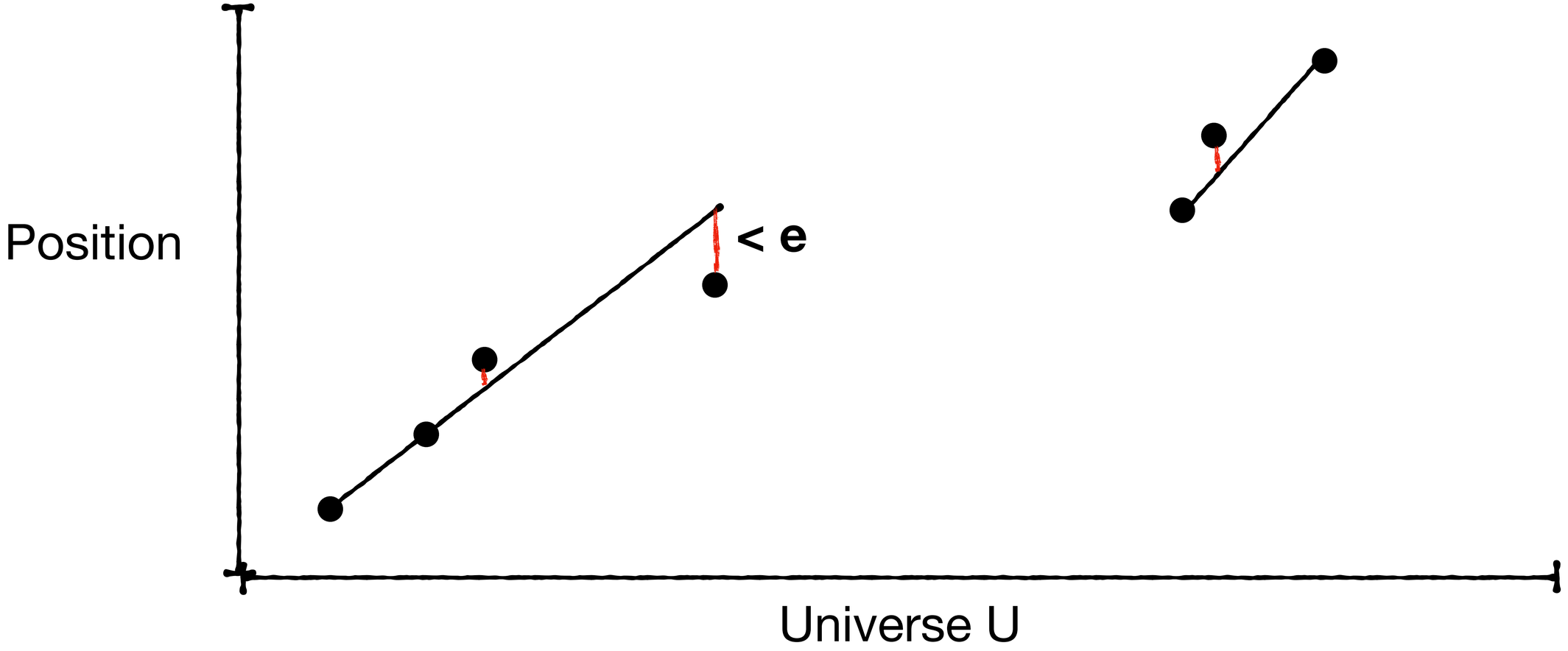


Ensure each point has max distance of e from its function

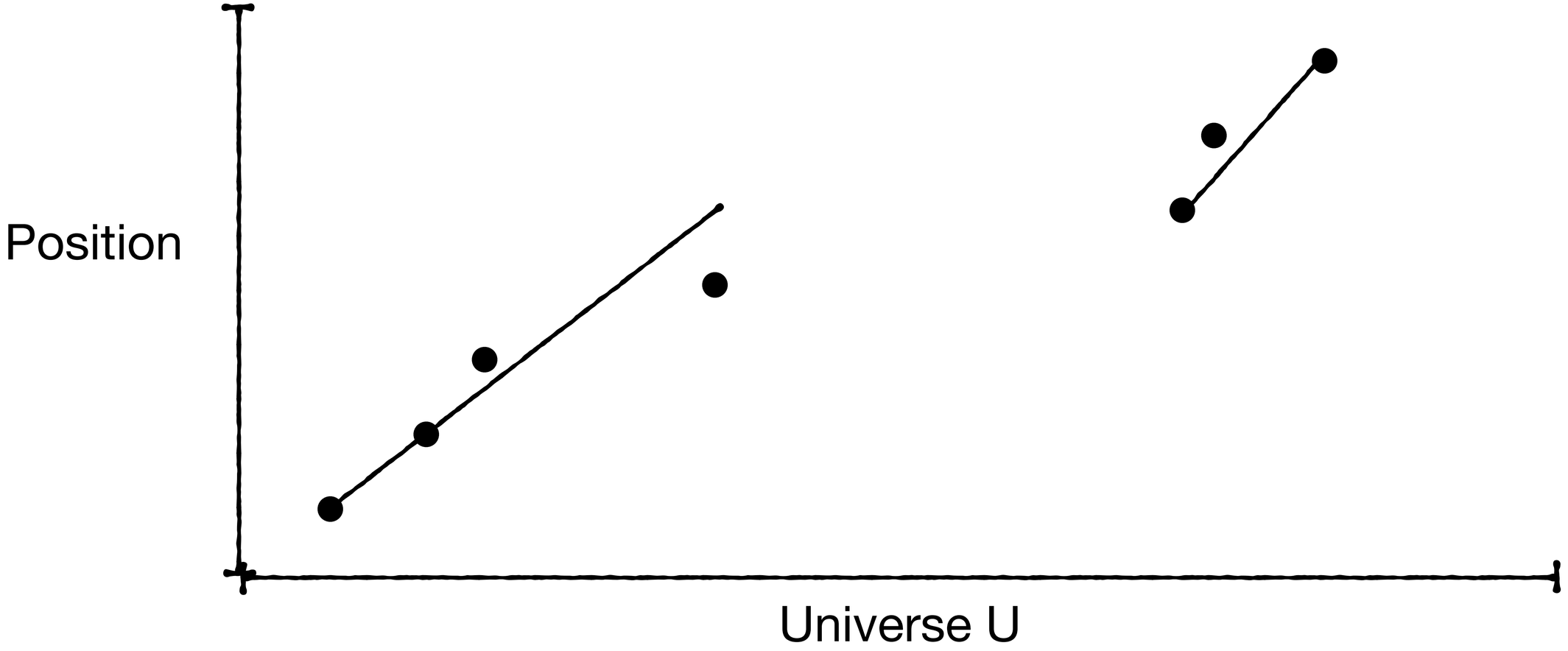
Why?



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

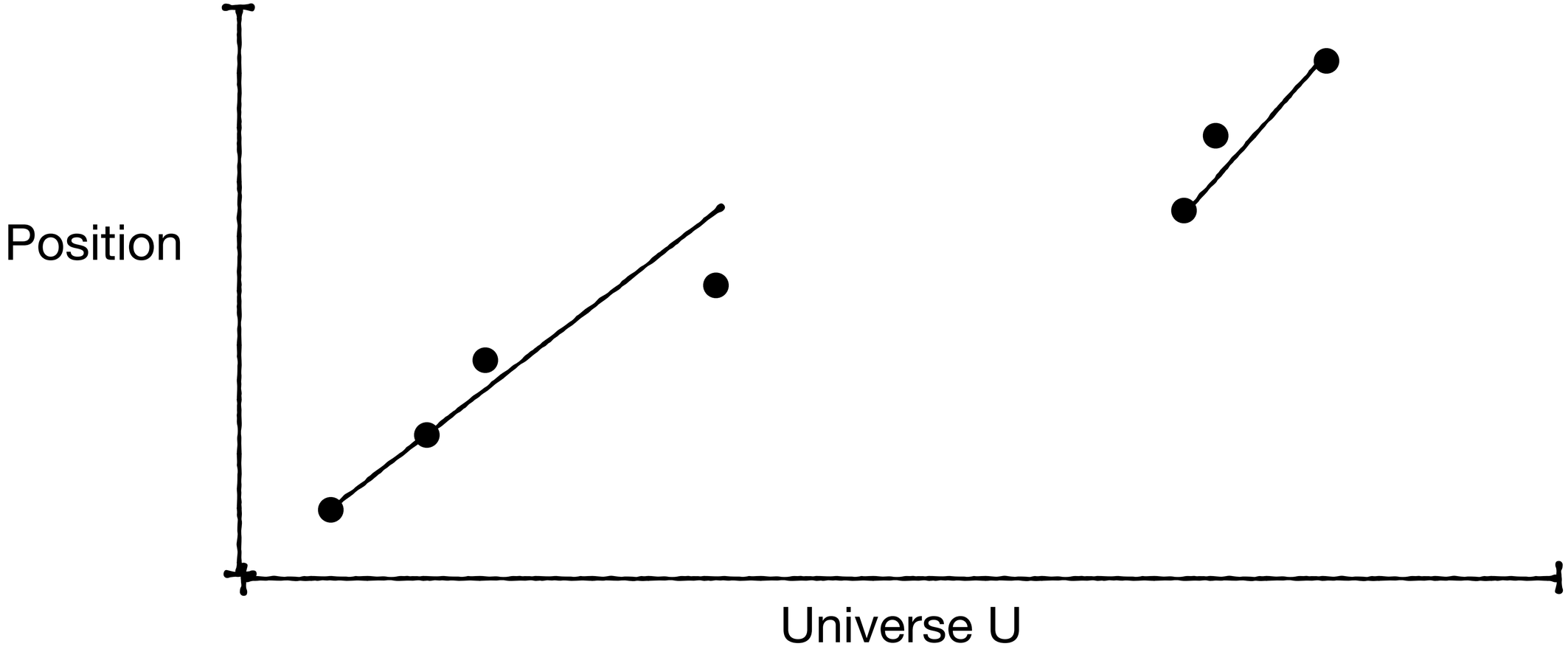


constrained optimization problem



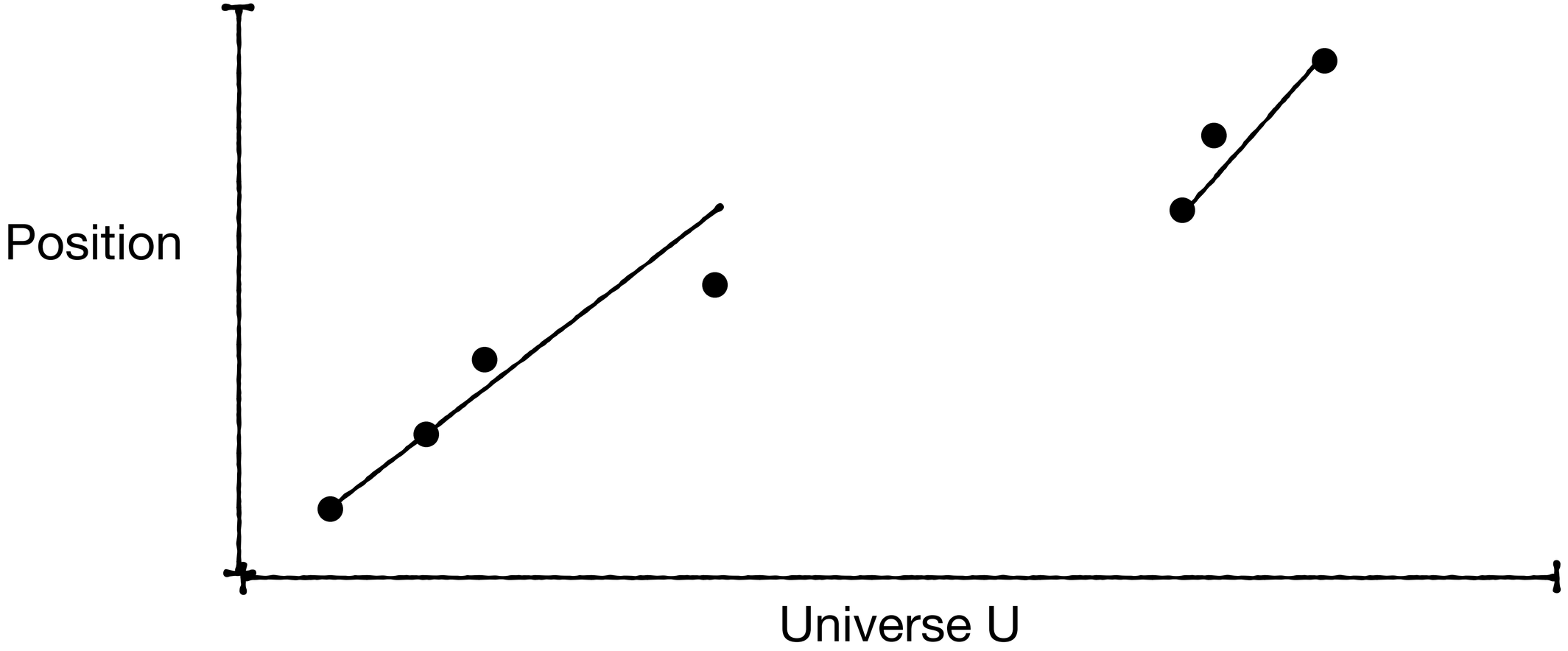
constrained optimization problem:

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

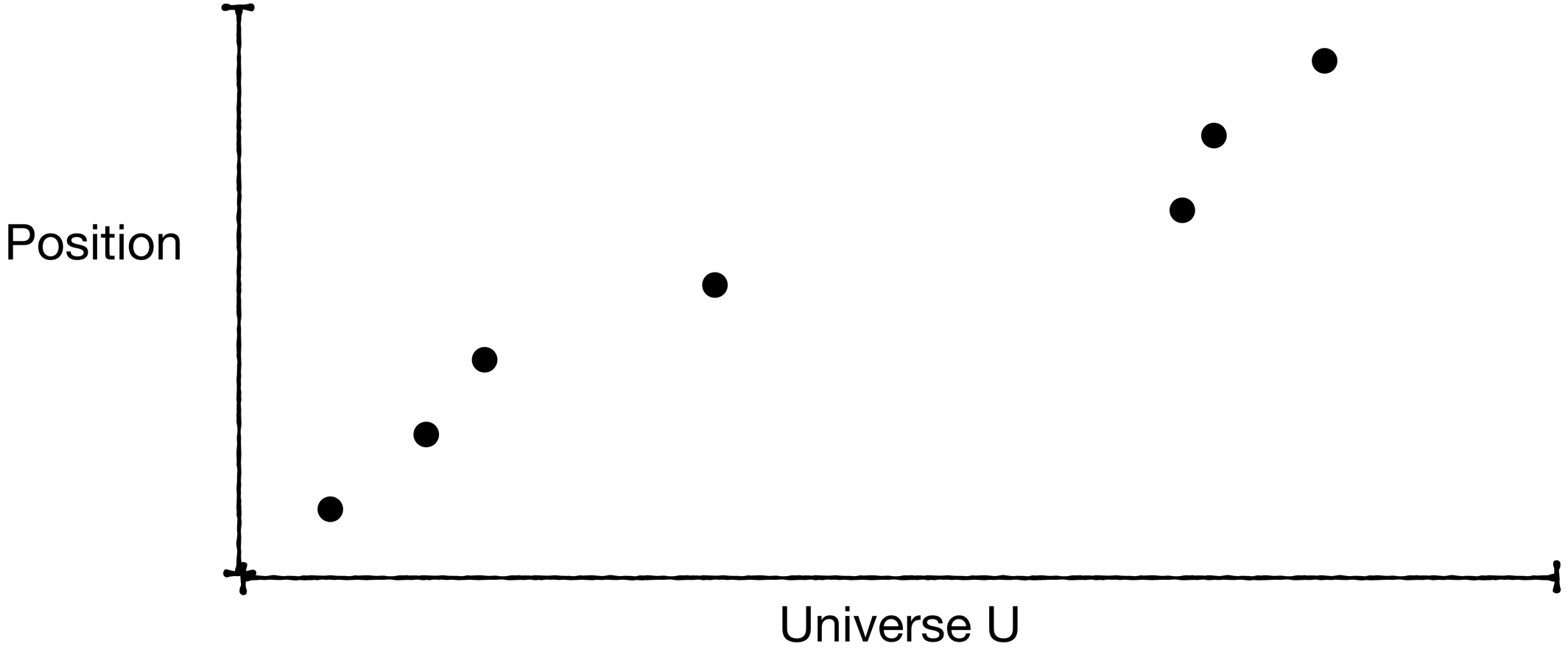


Model distribution using least number of segments while ensuring all points are within e positions of prediction

propose an algorithm for this!

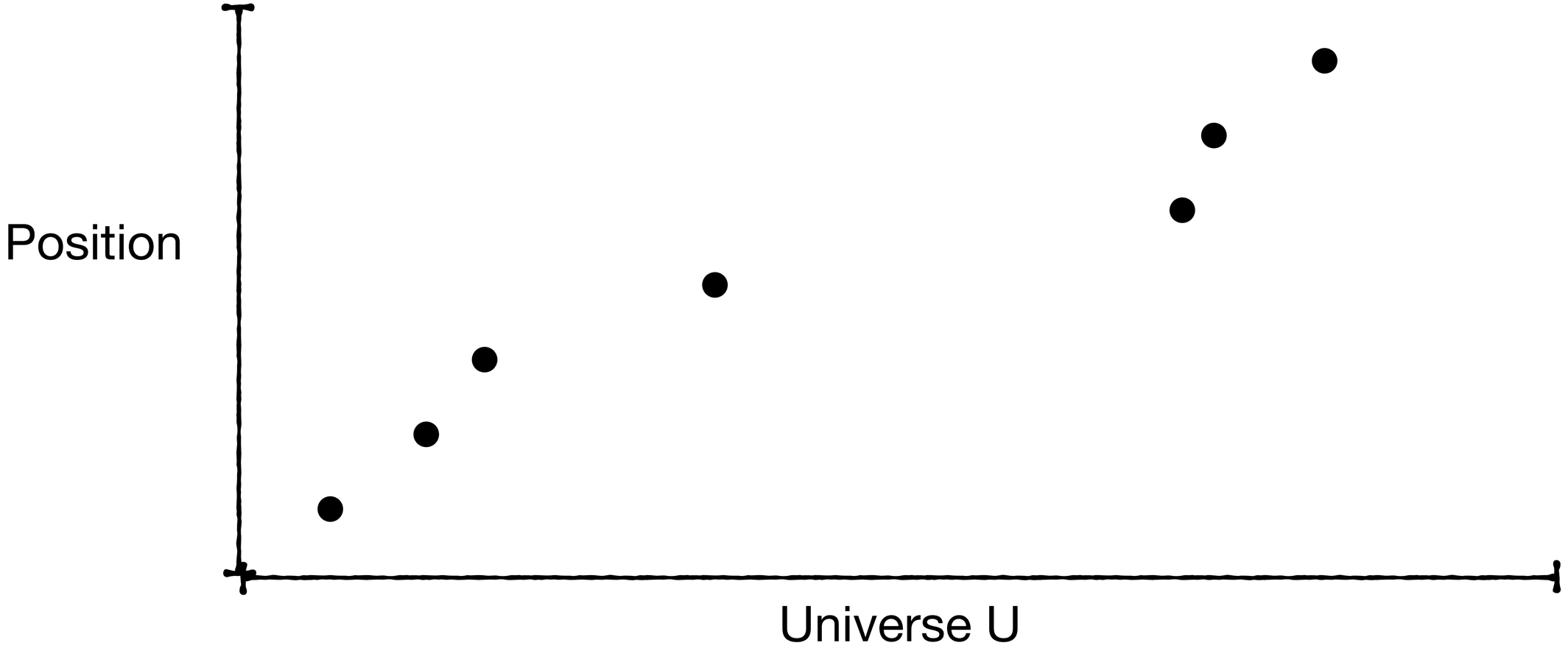


There is a $O(N^2)$ optimal algorithm - too expensive

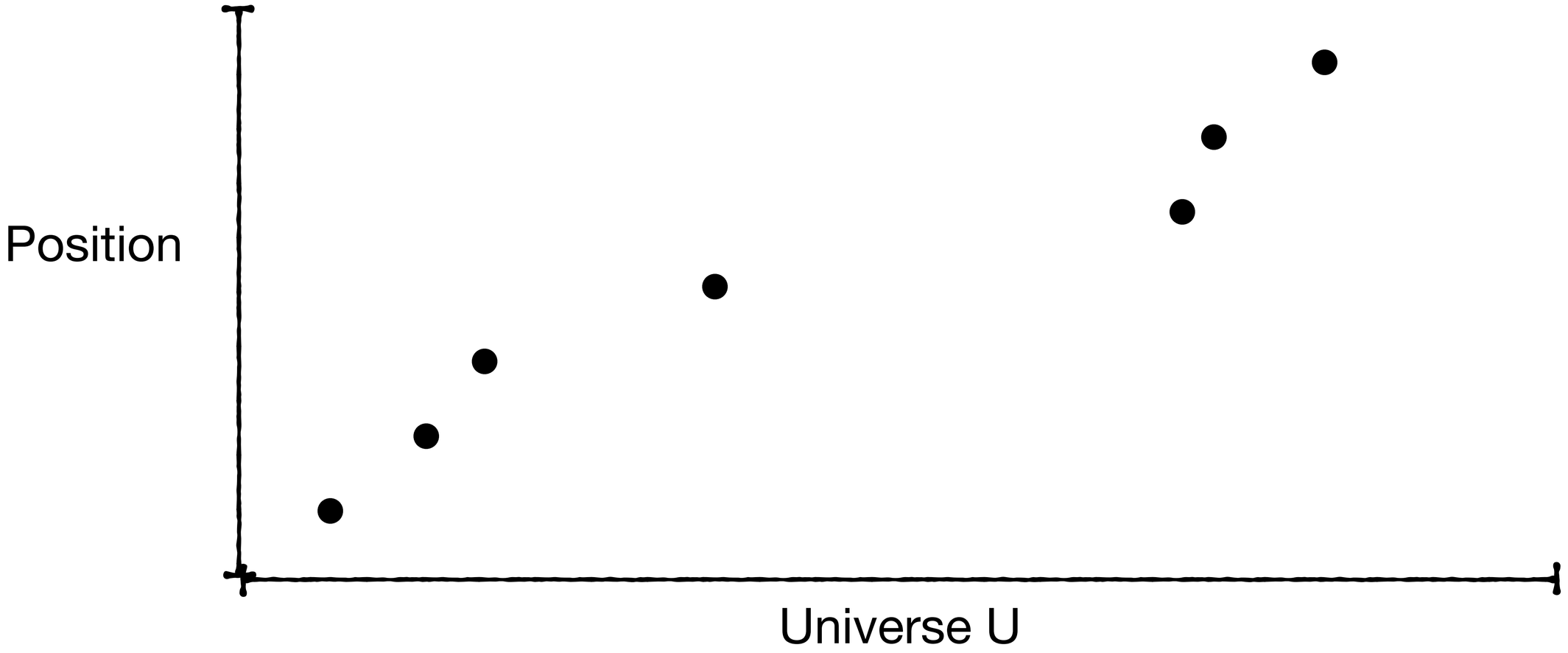


There is a $O(N^2)$ optimal algorithm - too expensive

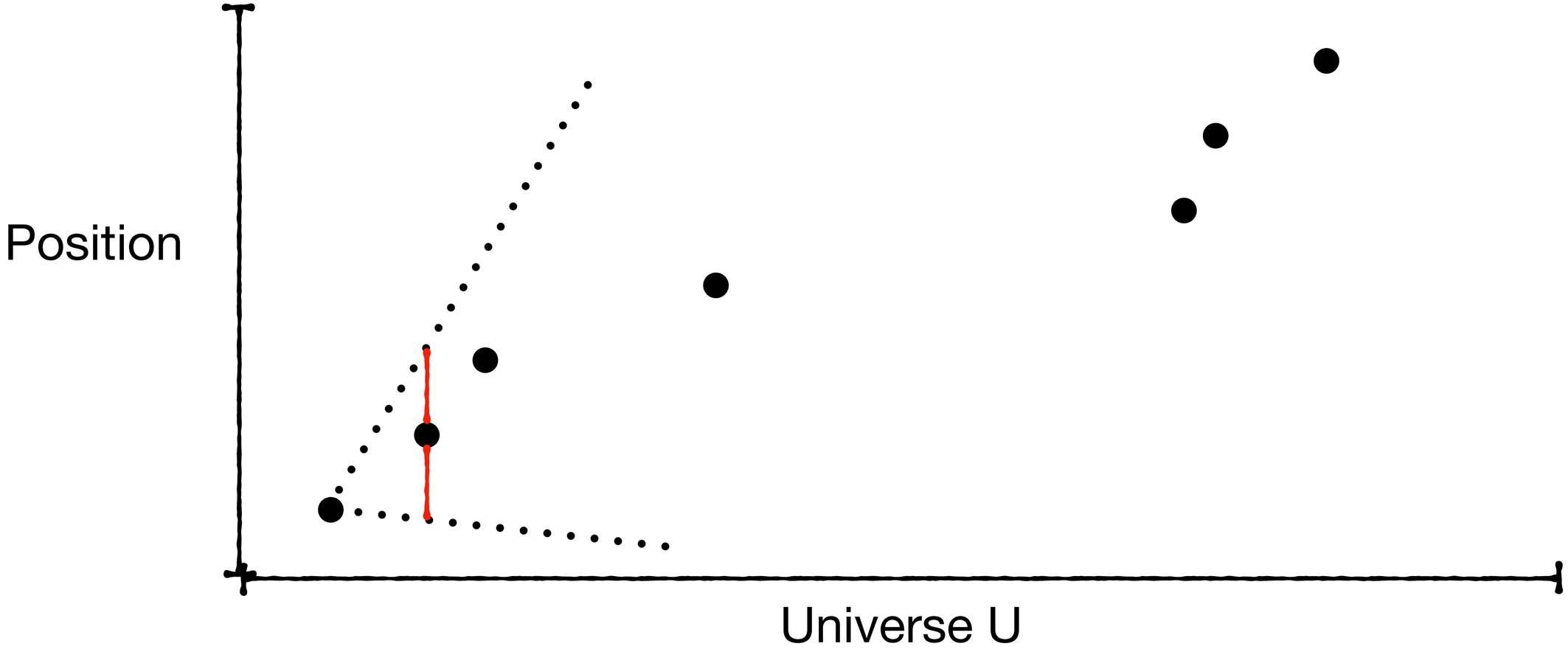
How about a O(N) approximate algorithm?



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

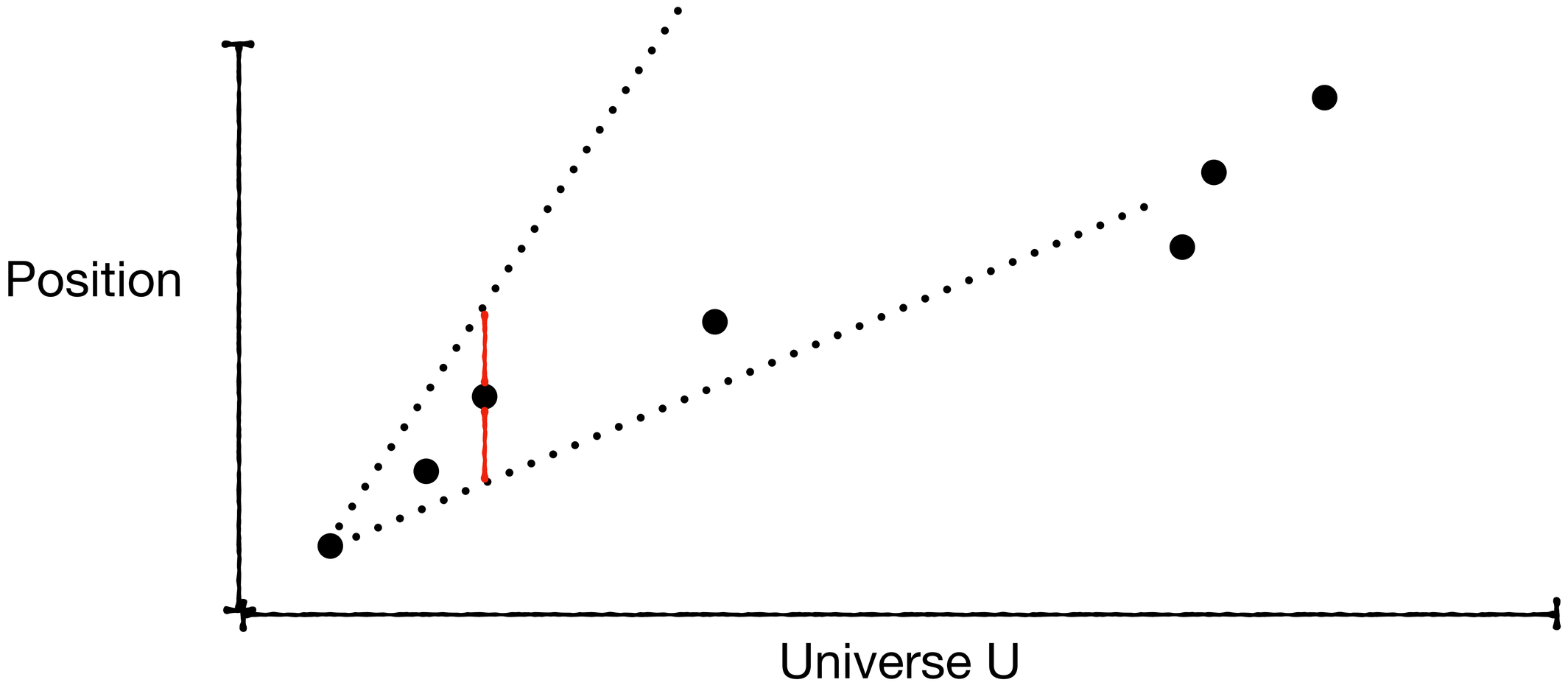


Add first two points into segment while maintaining “maximal cone”



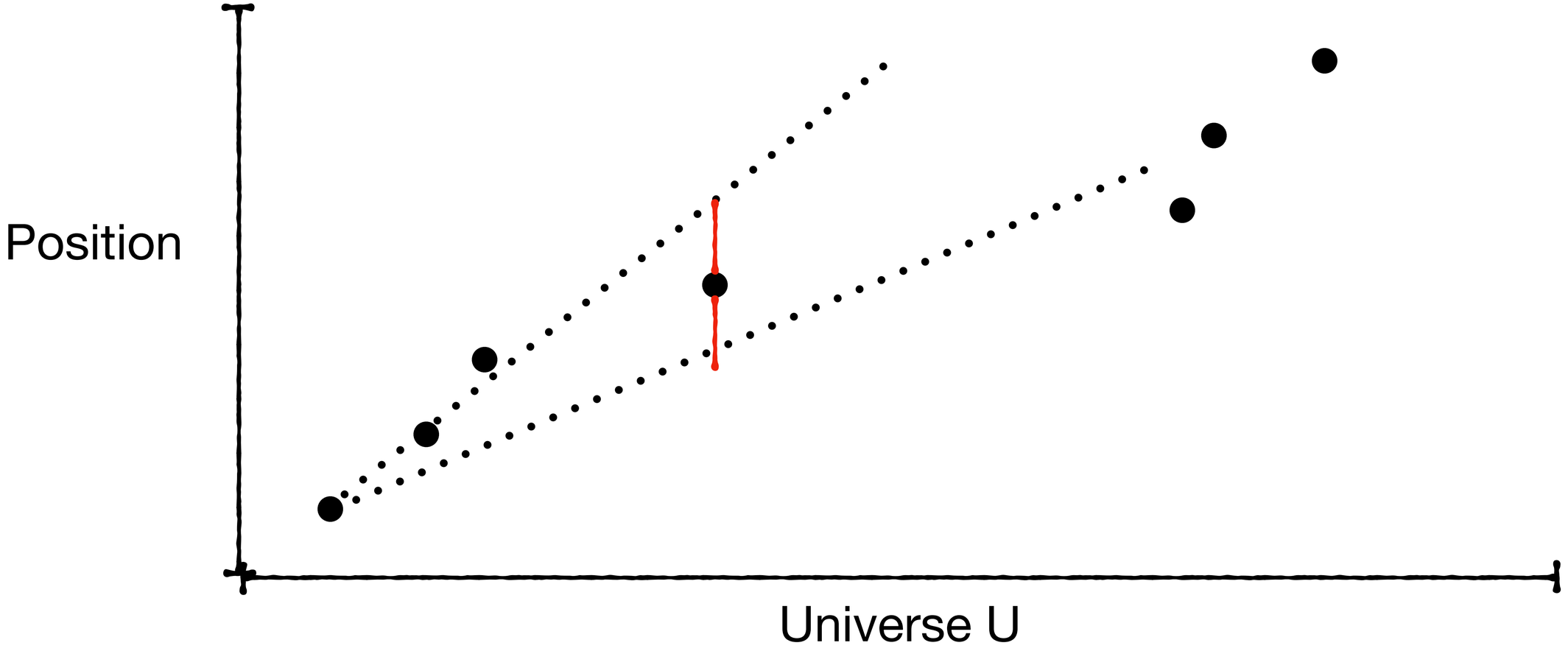
Add first two points into segment while maintaining “maximal cone”

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



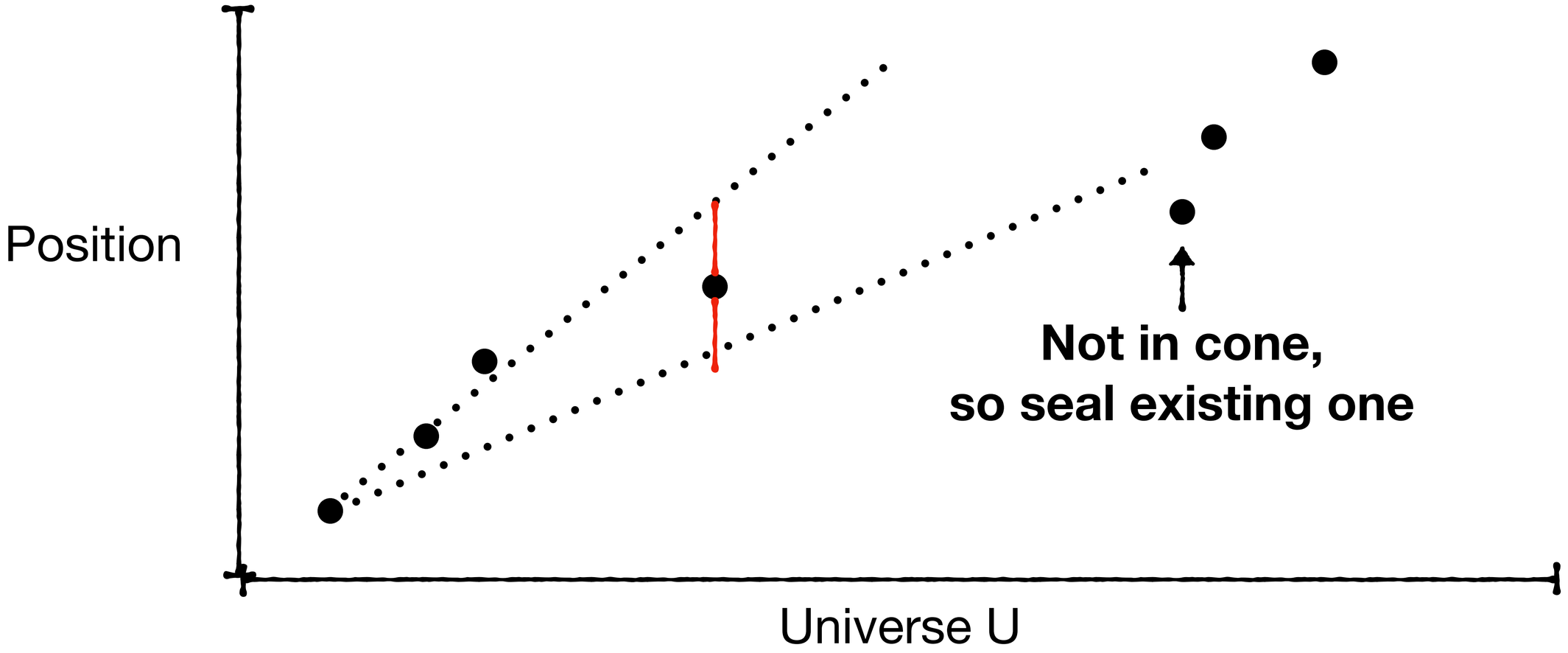
Add first two points into segment while maintaining “maximal cone”

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



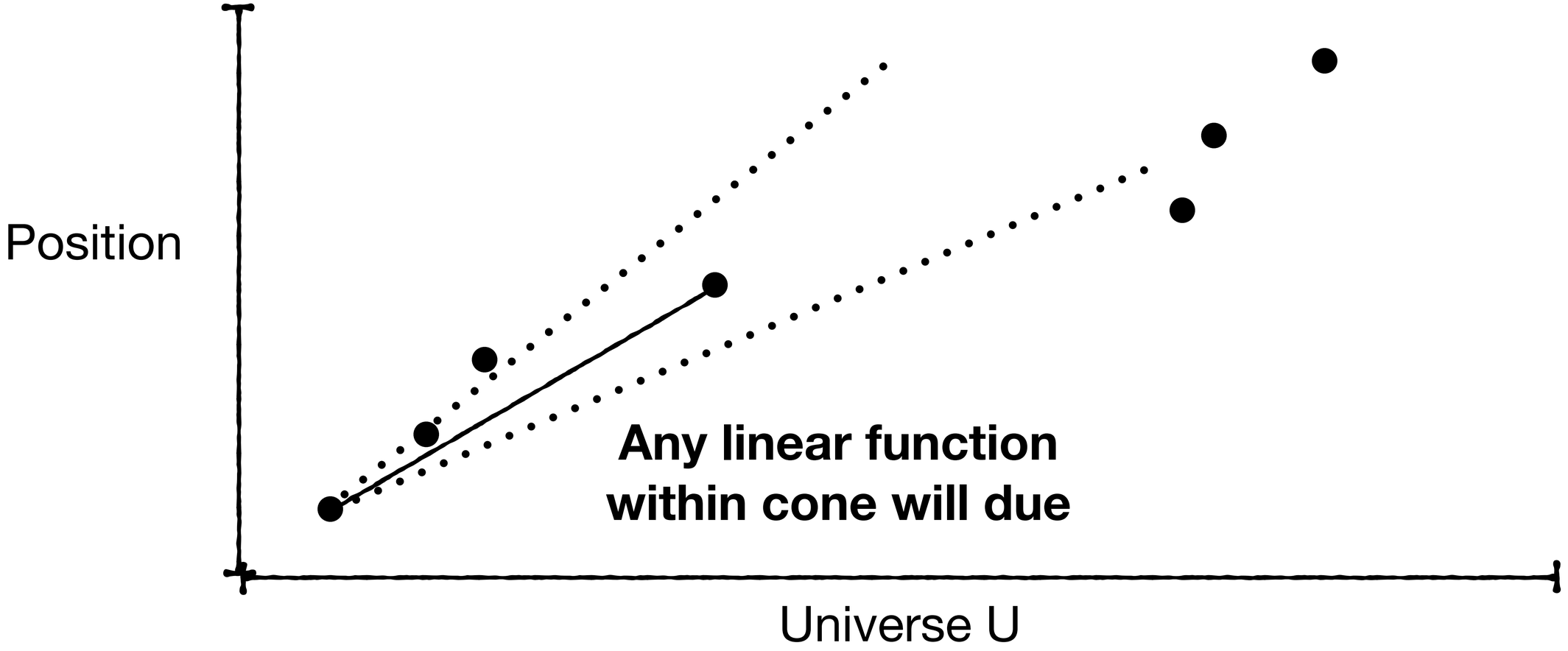
Add first two points into segment while maintaining “maximal cone”

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



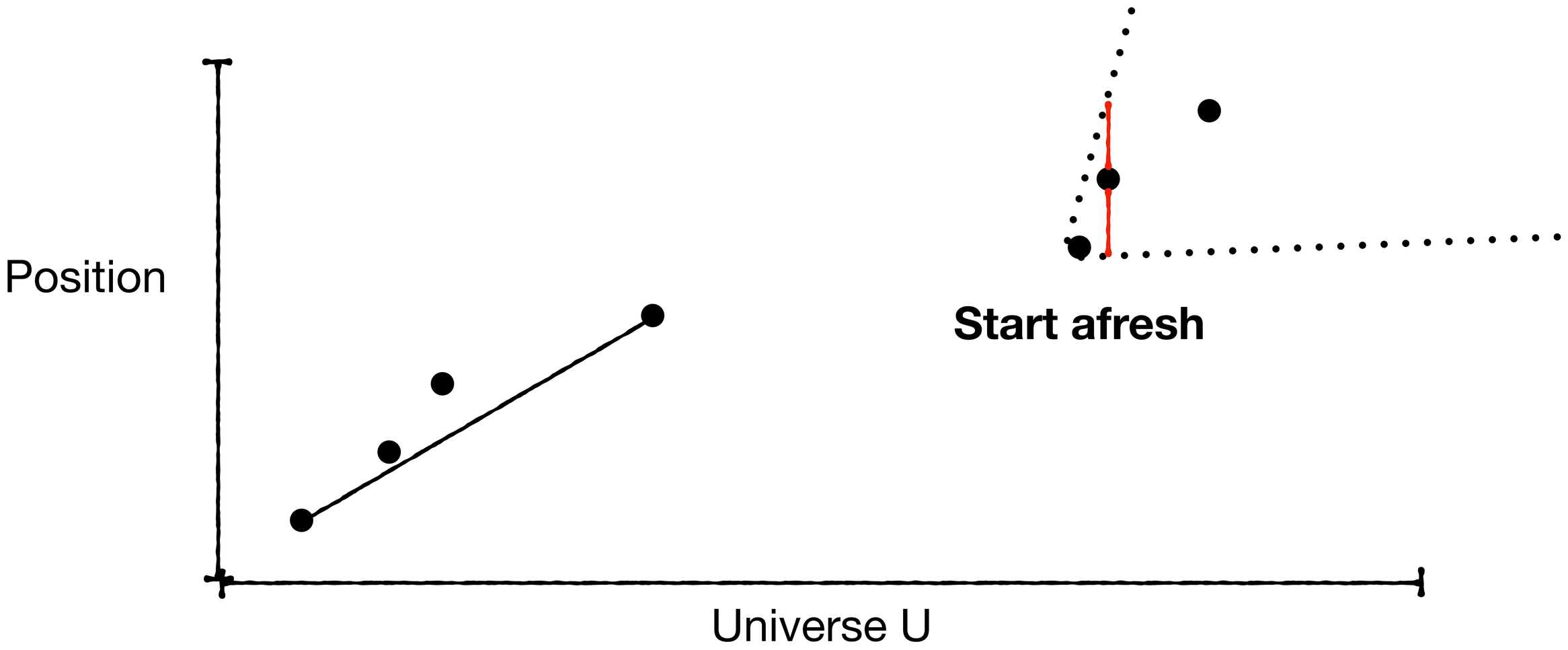
Add first two points into segment while maintaining “maximal cone”

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



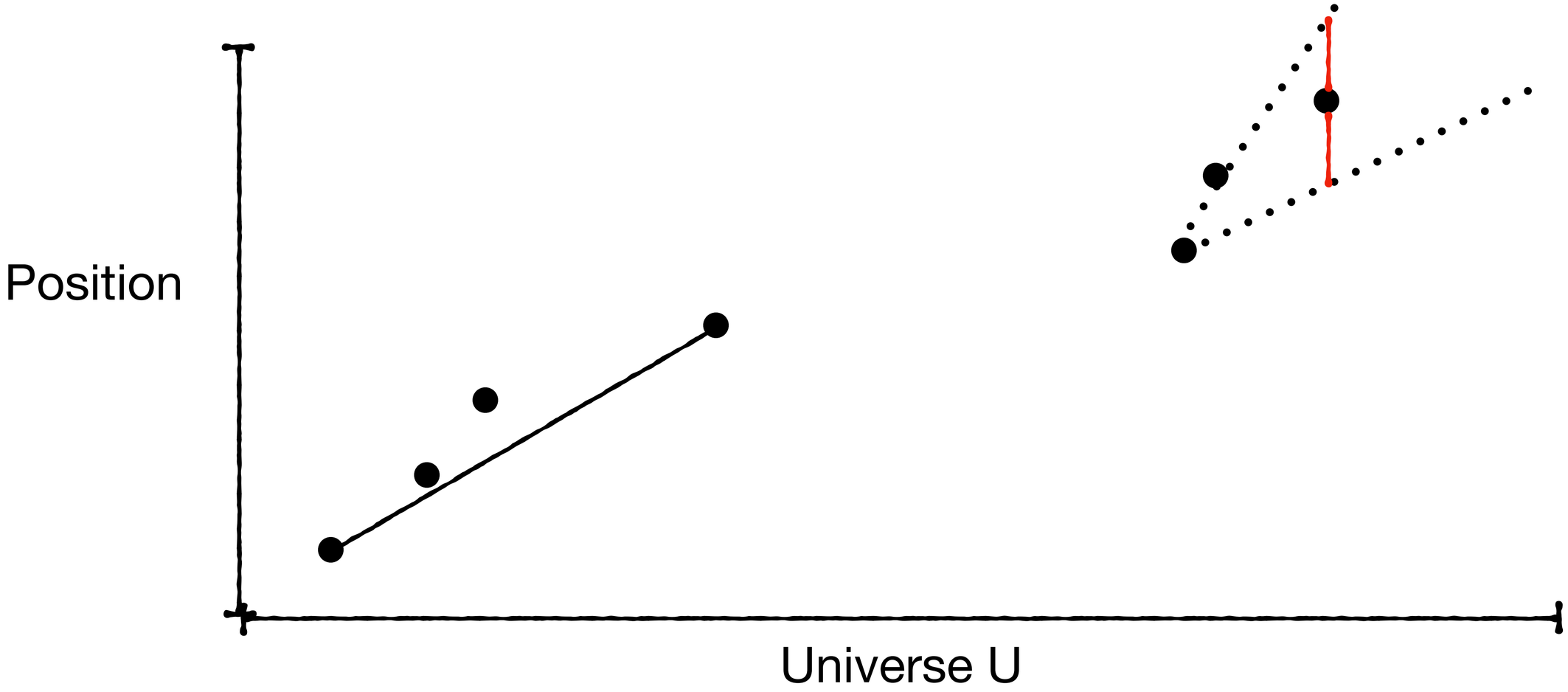
Add first two points into segment while maintaining “maximal cone”

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



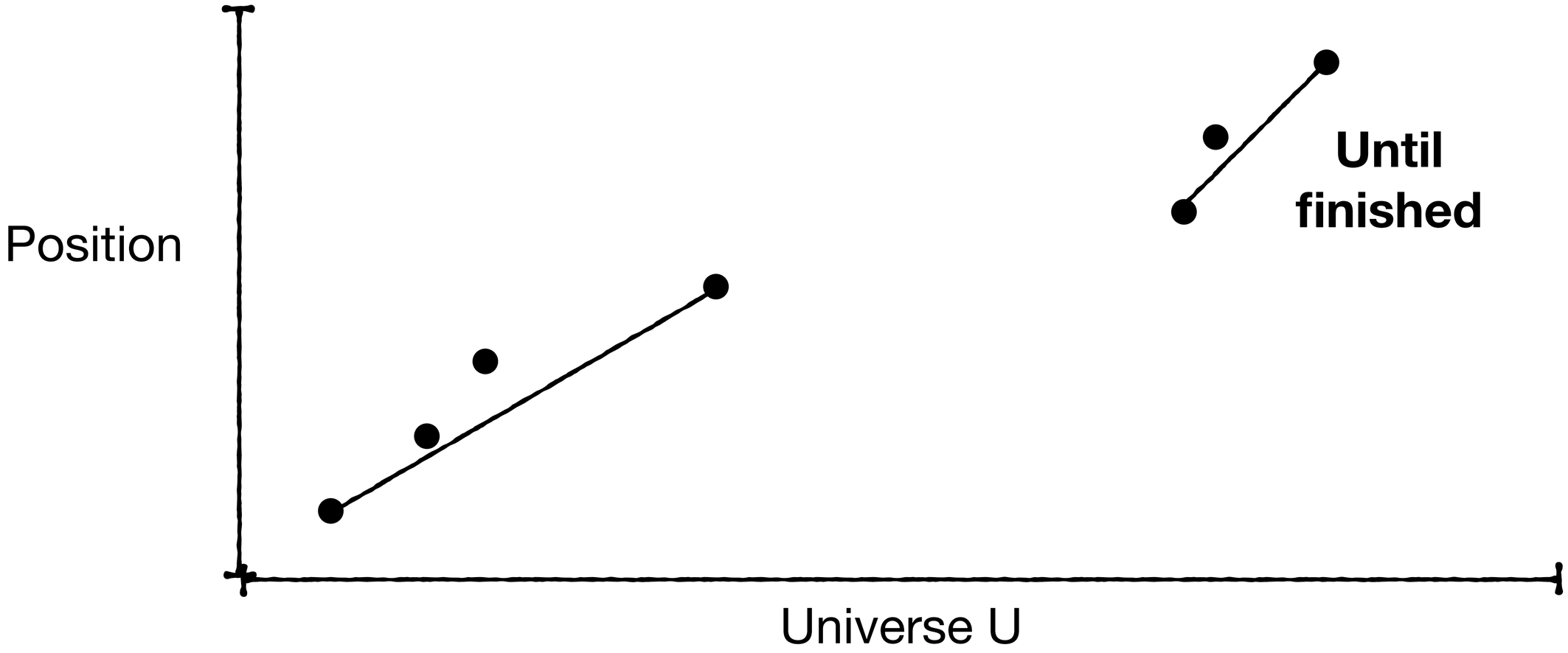
Add first two points into segment while maintaining “maximal cone”

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

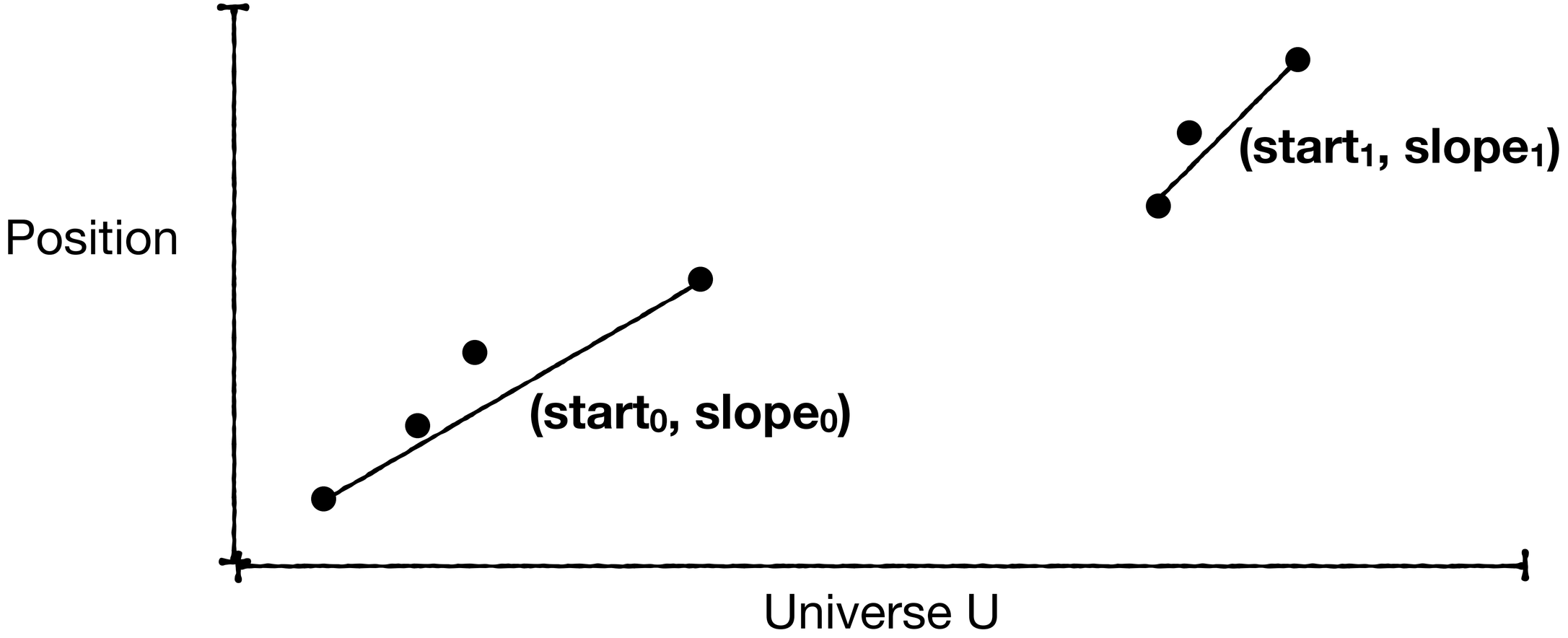


Add first two points into segment while maintaining “maximal cone”

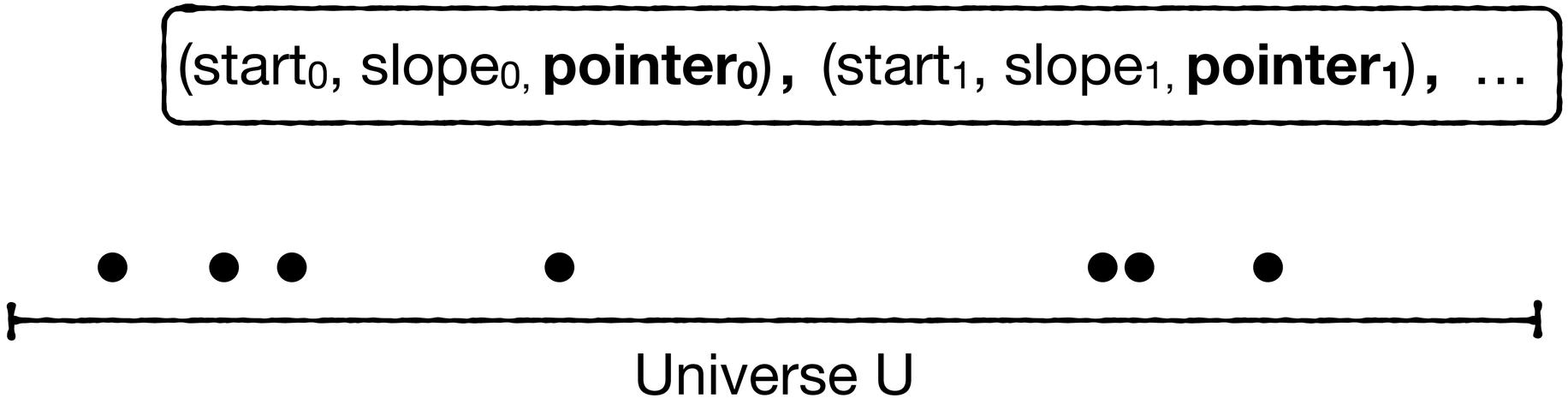
If next point is within cone, add it to segment & narrow cone accordingly



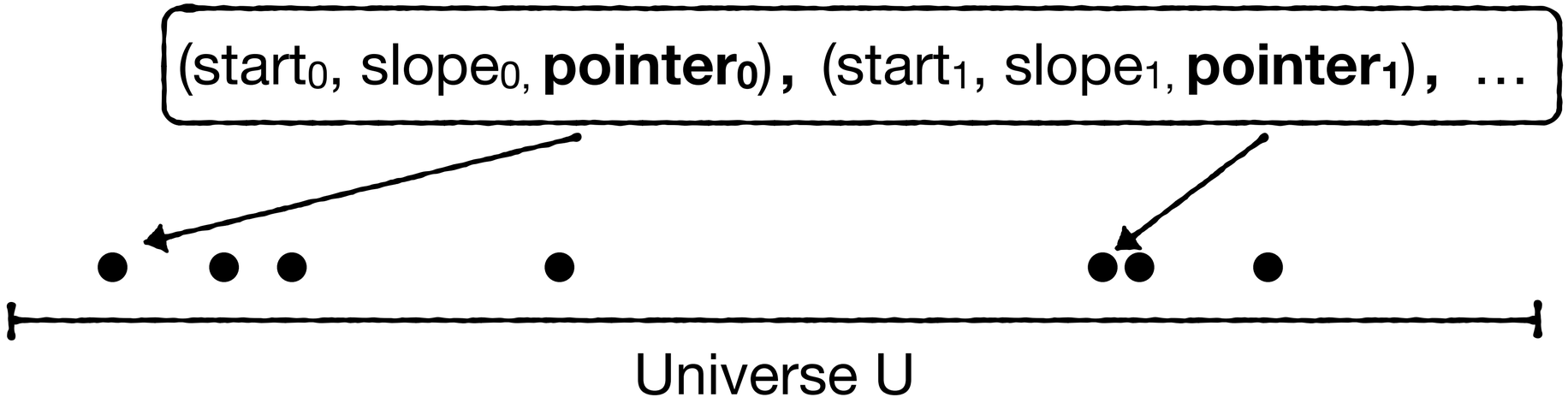
Each segment is characterized by starting point and slope



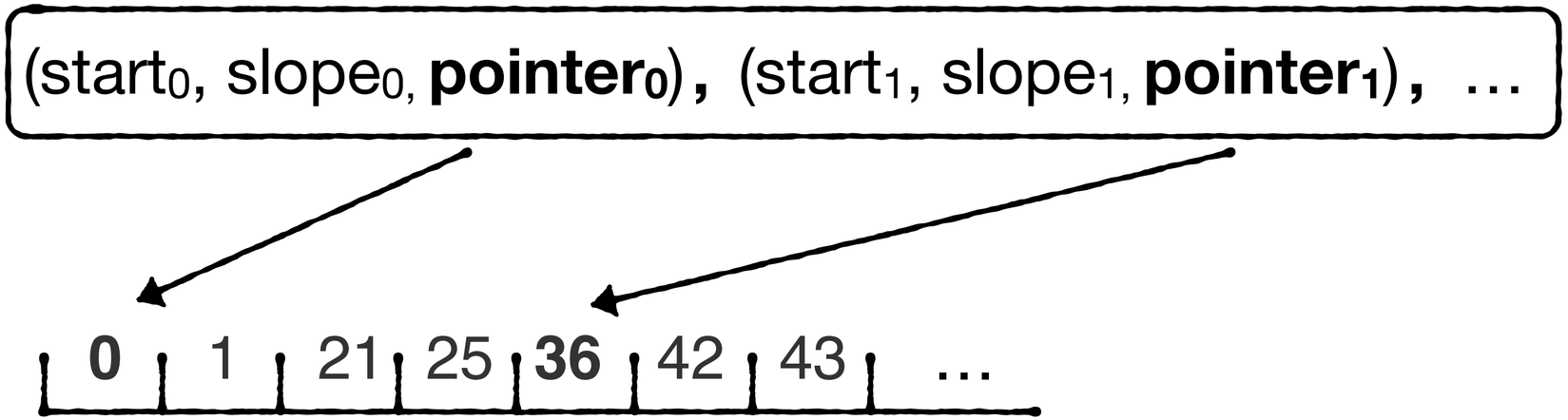
Place in B-tree node, sorted by starting point



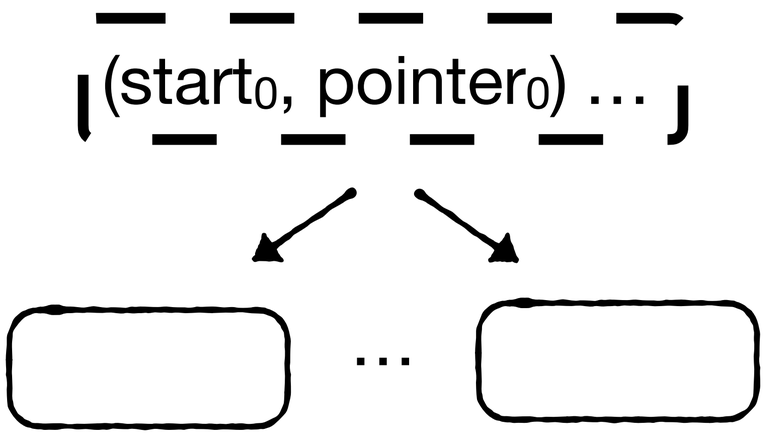
Place in B-tree node, sorted by starting point



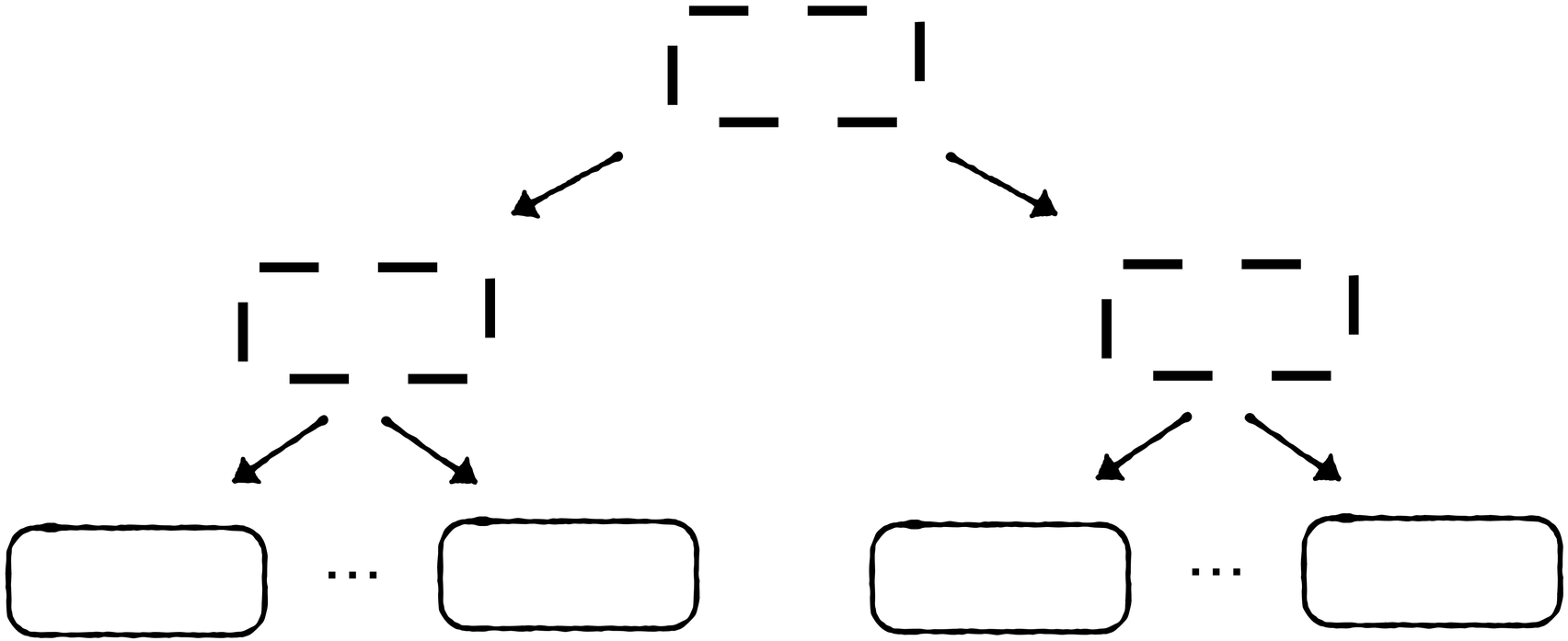
Place in B-tree node, sorted by starting point



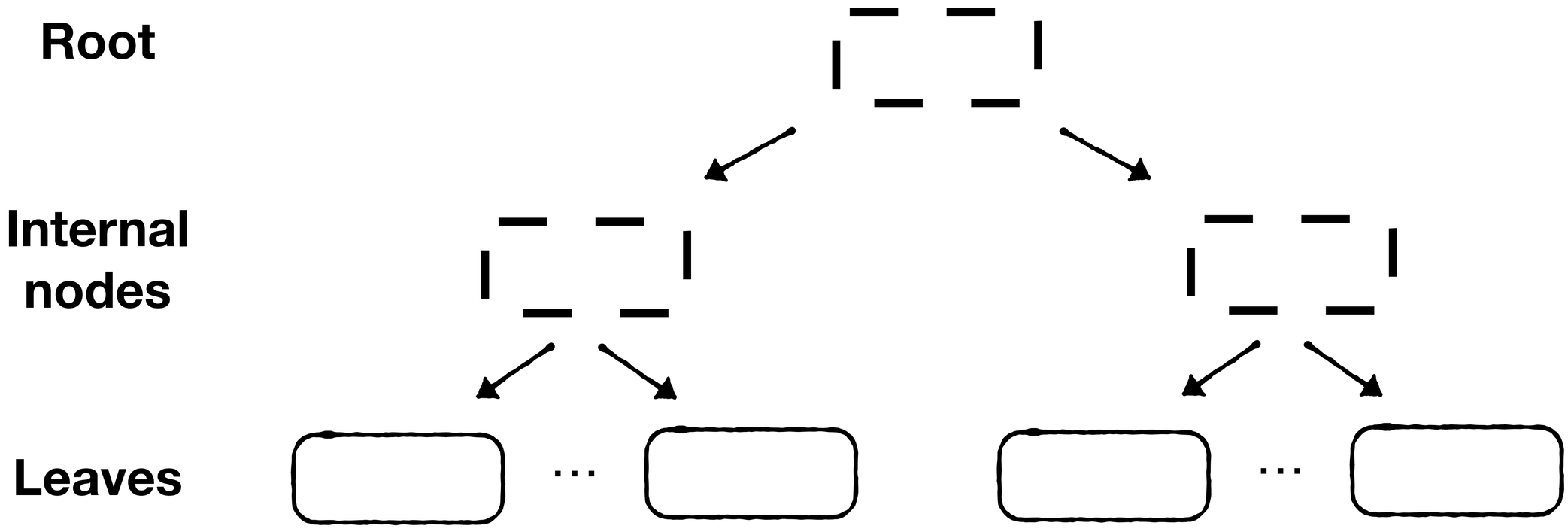
Store as a B-tree



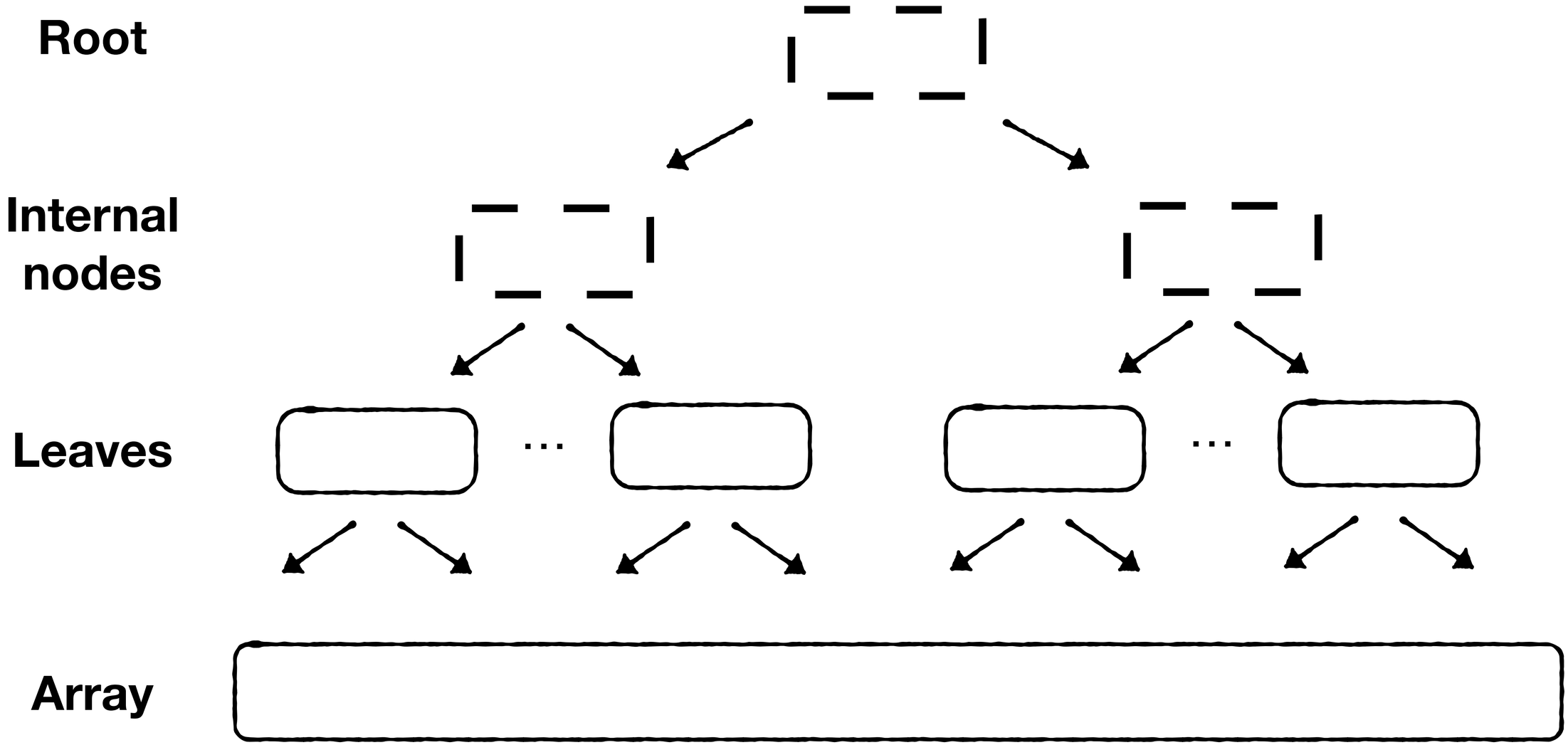
Store as a B-tree



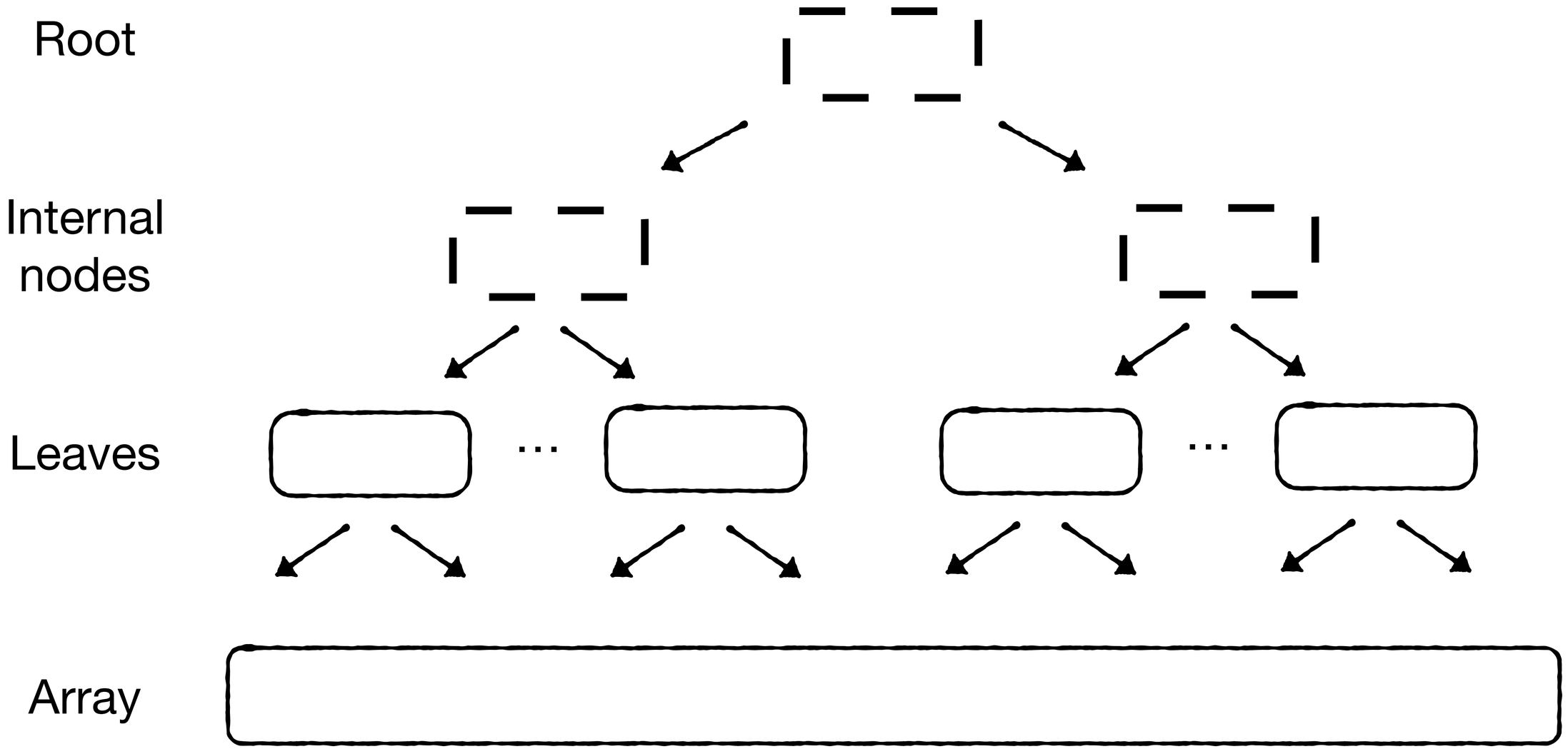
Store as a B-tree



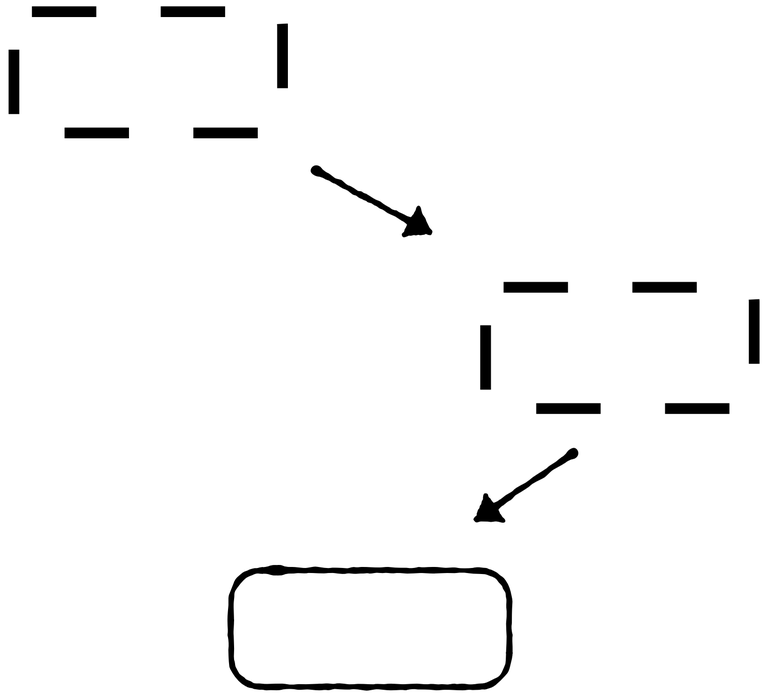
Store as a B-tree



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



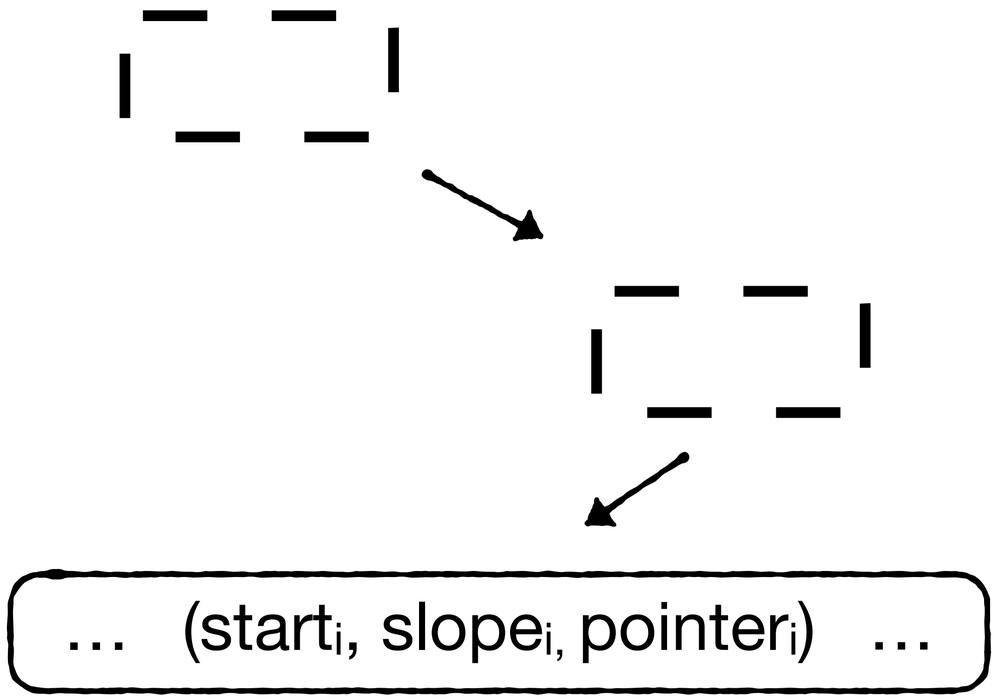
How to query this tree? **(1) traverse B-tree**



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

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

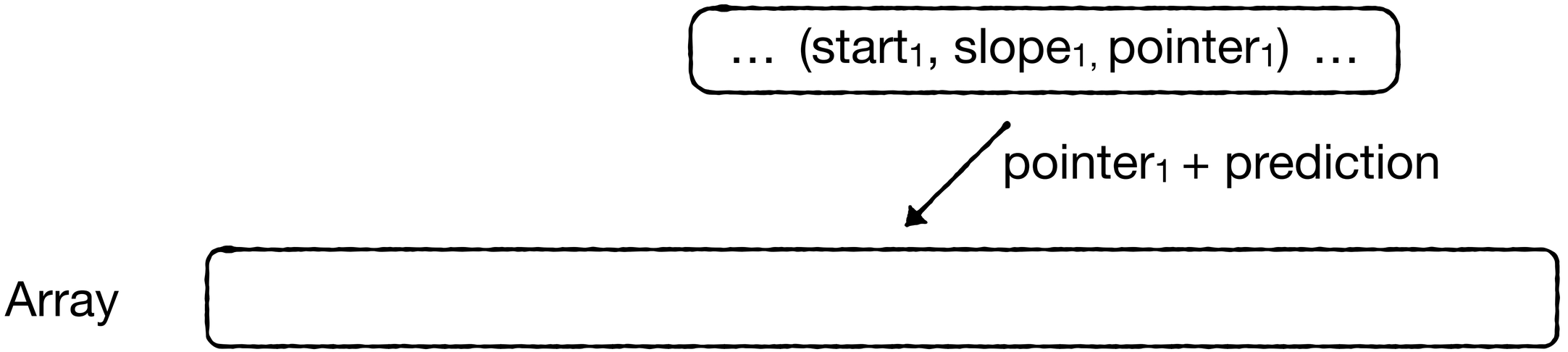
How to query this tree?

(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?

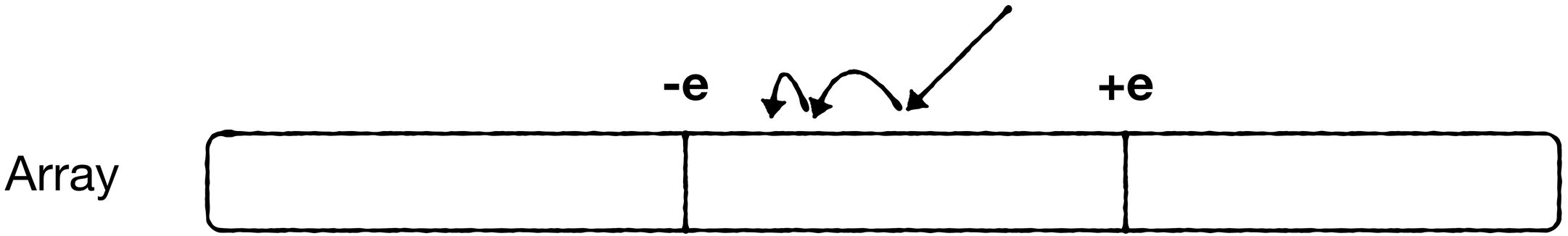
(1) traverse B-tree

(2) Find starting segment

(3) interpolate using slope

(4) add prediction to pointer and access array

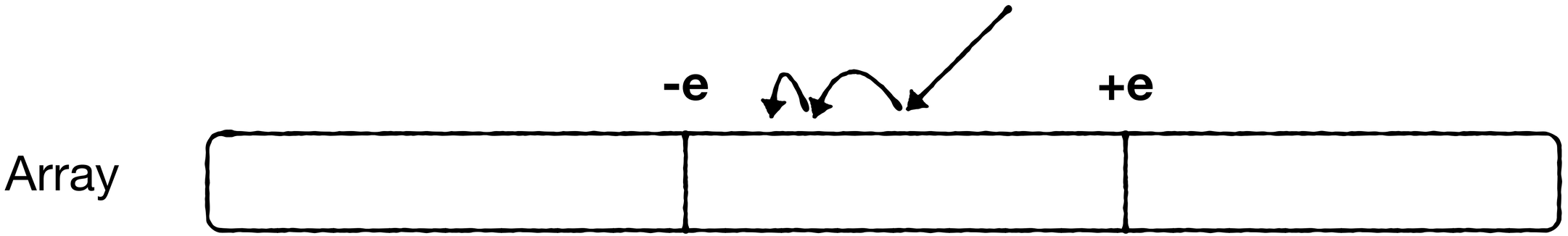
(5) binary search within max error bounds



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

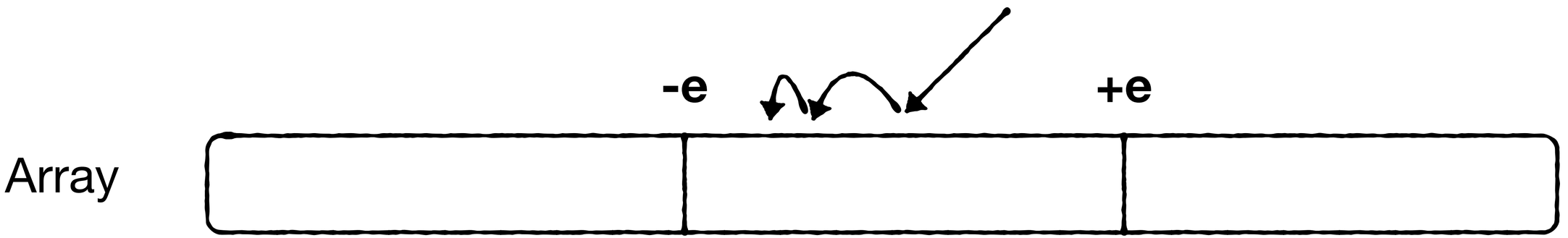
Query cost?



How to query this tree?

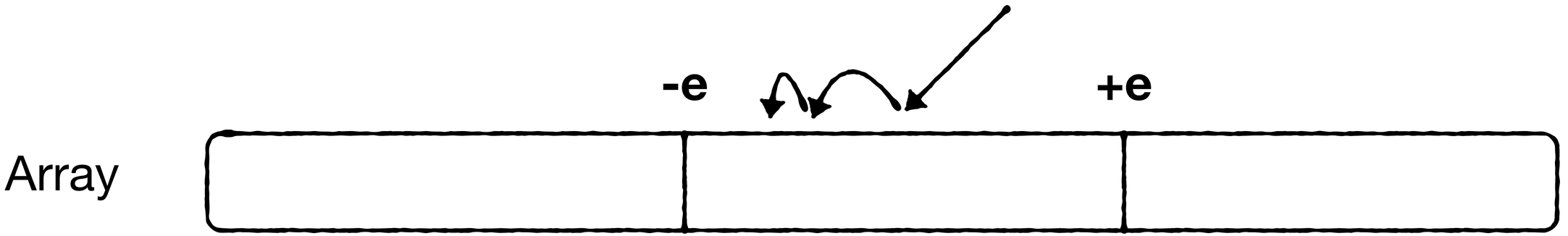
(1) traverse B-tree
(2) Find starting segment
(3) interpolate using slope
(4) add prediction to pointer and access array
(5) binary search within max error bounds

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

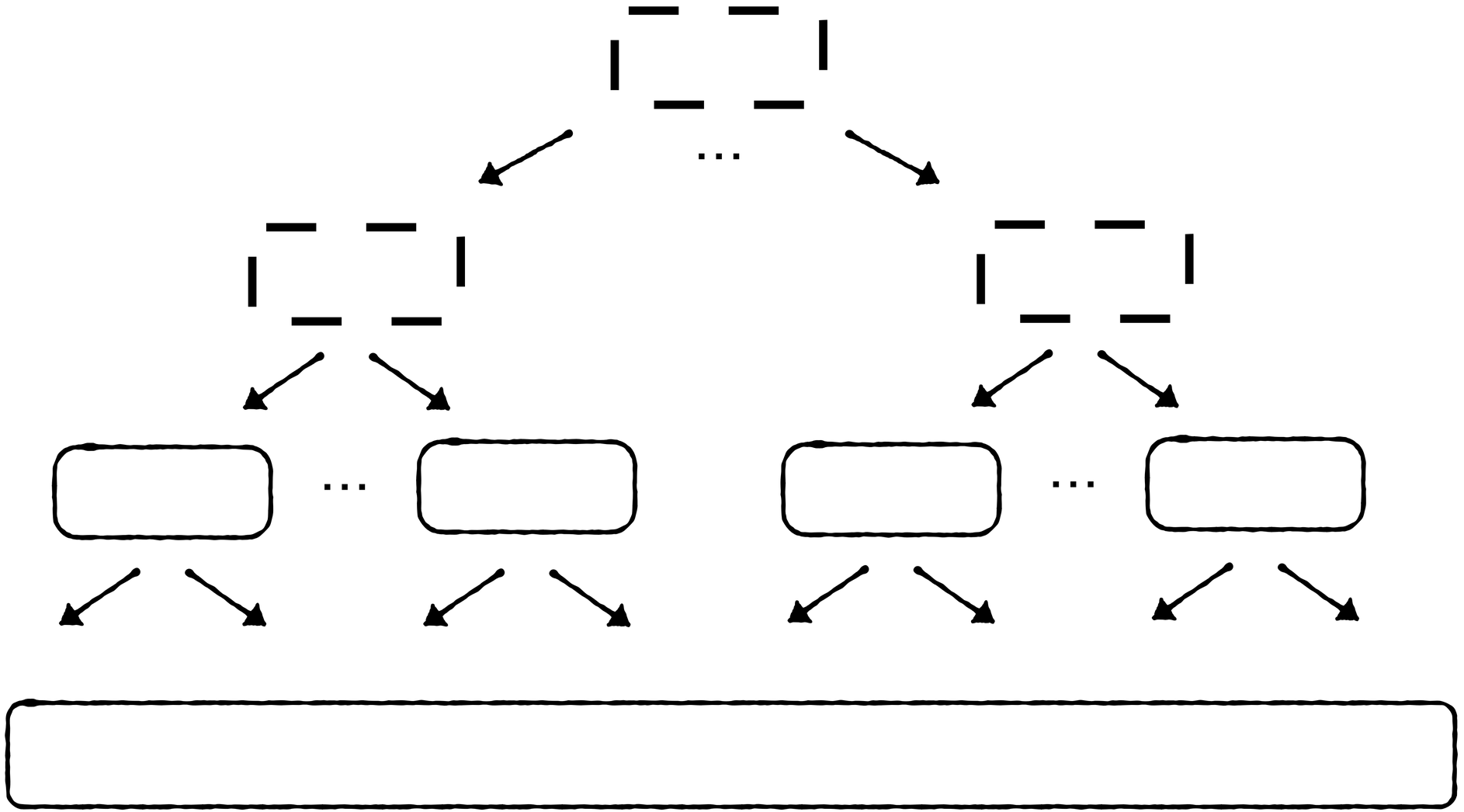


Query cost? O(log(**#segments**) + log(**e**))

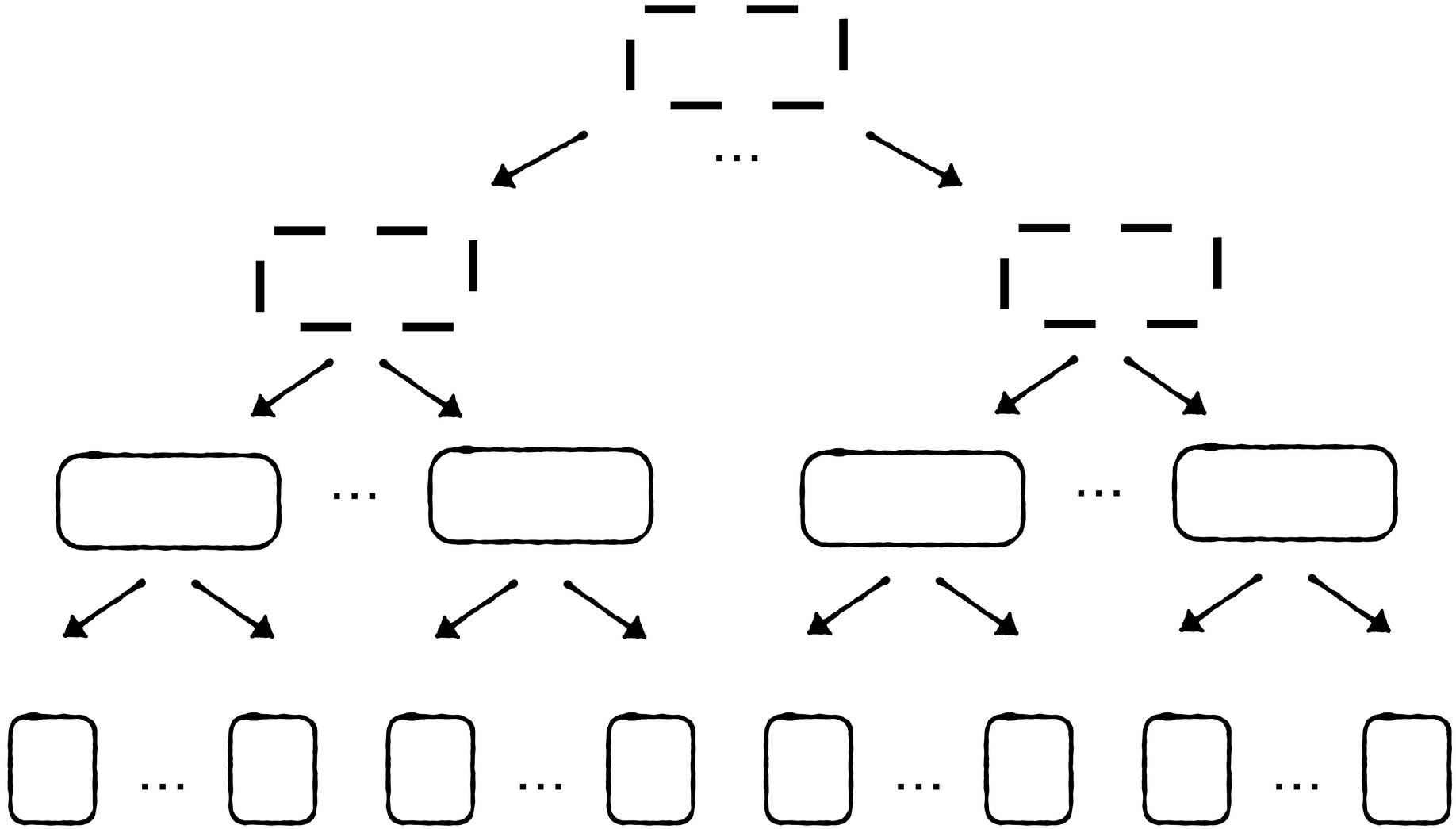
The more predictable the data is, the more the cost drops



How to handle updates?



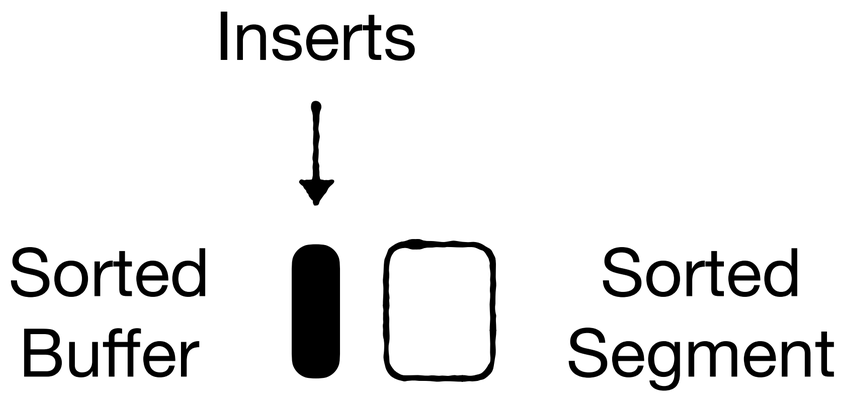
How to handle updates? **(1) separate segments into contiguous chunks**



How to handle updates?

(1) separate segments into contiguous chunks

(2) employ sorted insert buffer for each segment

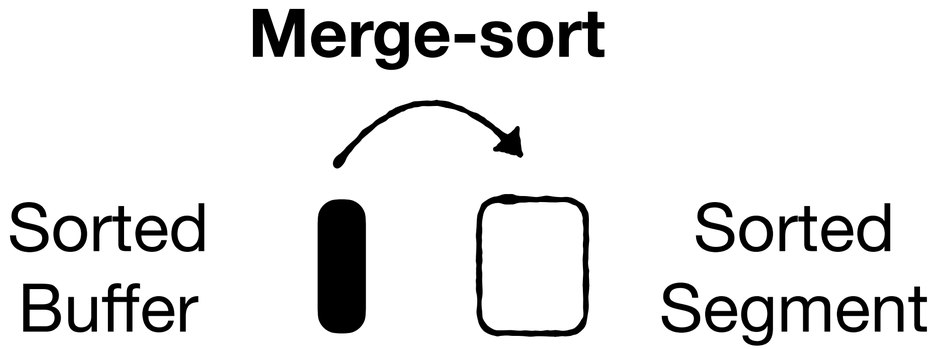


How to handle updates?

(1) separate segments into contiguous chunks

(2) employ sorted insert buffer for each segment

(3) merge buffer into segment at threshold size



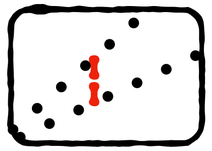
How to handle updates?

(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?

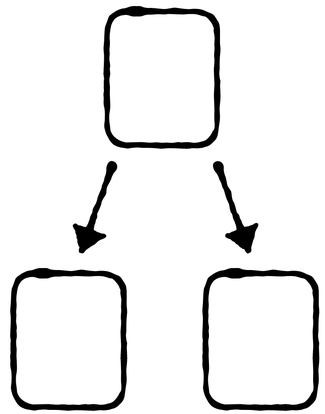
(1) separate segments into contiguous chunks

(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

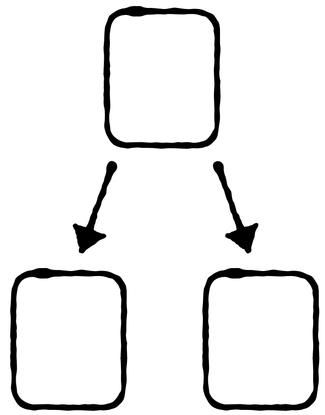
(2) employ sorted insert buffer for each segment

(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

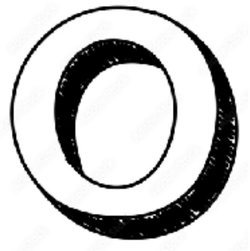


Sorted
Segment

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

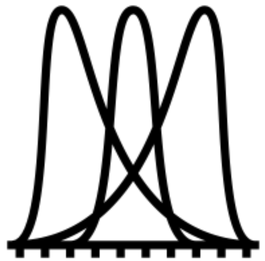
T-Tree
1985

CSS-Tree
1998

CSB-Tree
2000

HOT
2018

Exploiting Data Distribution



Interpolation search
1959

FITing-Tree
2019