Storing & Indexing Structured Data

Asterios Katsifodimos

Who am I?













BSc+MSc

PhD

Postdoc

SW. Engineer

Ass. Prof.

Data-intensive Systems Group

ING ICAI NWO adyen aws ASML

Fintech | Cloud Data Management | Data Marketplaces | Industry 4.0

Resilient Systems | Data Systems | Graph ML | Federated ML | Decentralized Systems



Jérémie Decouchant

Resilient Distributed Systems



Asterios Katsifodimos
Group Chair

Scalable Data Systems



Kubilay Atasu

Scalable Graph Learning



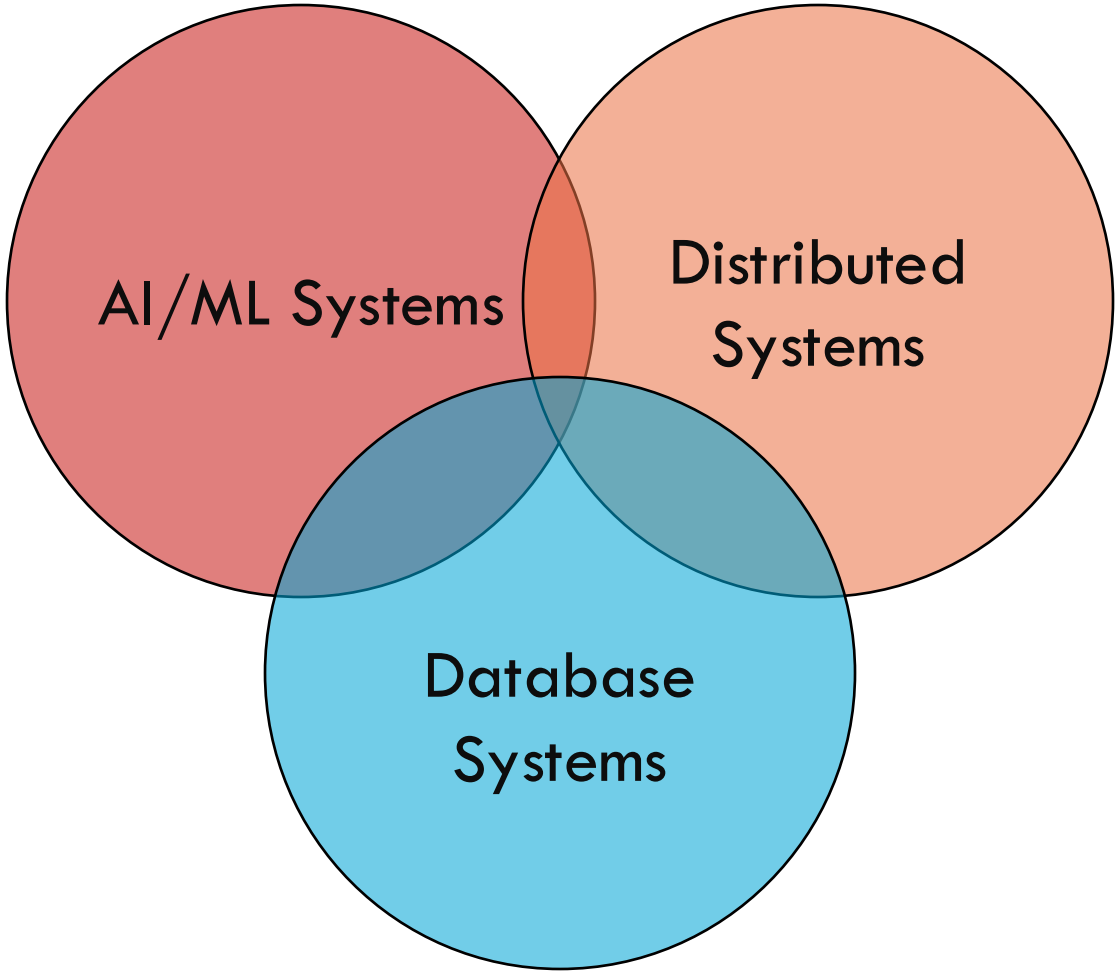
Lydia Chen

Distributed ML Systems



Johan Pouwelse

Decentralized Systems & Blockchain



Data integration for AI
Data-intensive ML
Federated Learning
Graph Learning
…

Parallel data processing
Cloud Computing
Blockchain
High-Performance Computing
Distributed algorithms
Decentralized systems
…

Programming/query models
Indexing
Partitioning
Database Transactions
…

Courses by Data-intensive Systems Group

BSc

MSc

Information & Data Management



Operating Systems



Distributed Algorithms (Q2)



Web-scale Data Management (Q2, core)



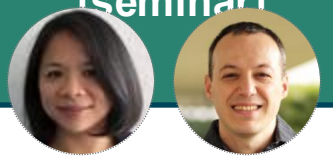
Distributed Data Systems (Q3)



Blockchain Engineering (Q4)



Scalable Learning Systems (Q4) [seminar]



Data Management and Engineering [DSAIT]

Rihan, Avishek, Asterios, Sole, …

Information Management & DSAIT

Distributed Systems Engineering Theme

BSc Classes

Today‘s lecture

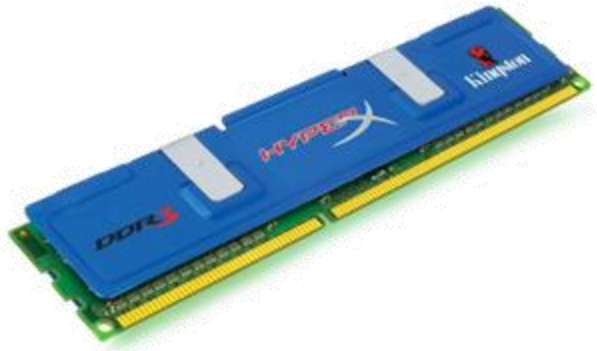
- This lecture: how to find data fast
- How **storage media** work
- **Indexes**: how to index information in hard disks
- Three Index Structures
  - Hash Indexes for point queries
  - B-Trees for range queries
  - KD-Trees for multimensional indexing

Why should I care about this lecture?

- Data management is the most important part of AI/ML pipelines.
- Everything starts with storing and querying of training and testing data.
- Scalability matters, in 2025.

Physical Storage Introduction

- Data is stored on a **storage media**. Media highly differ in terms of
  - **Random Access** Speed
  - **Random/Sequential Read/Write** speed
  - **Capacity**
  - **Cost** per Capacity







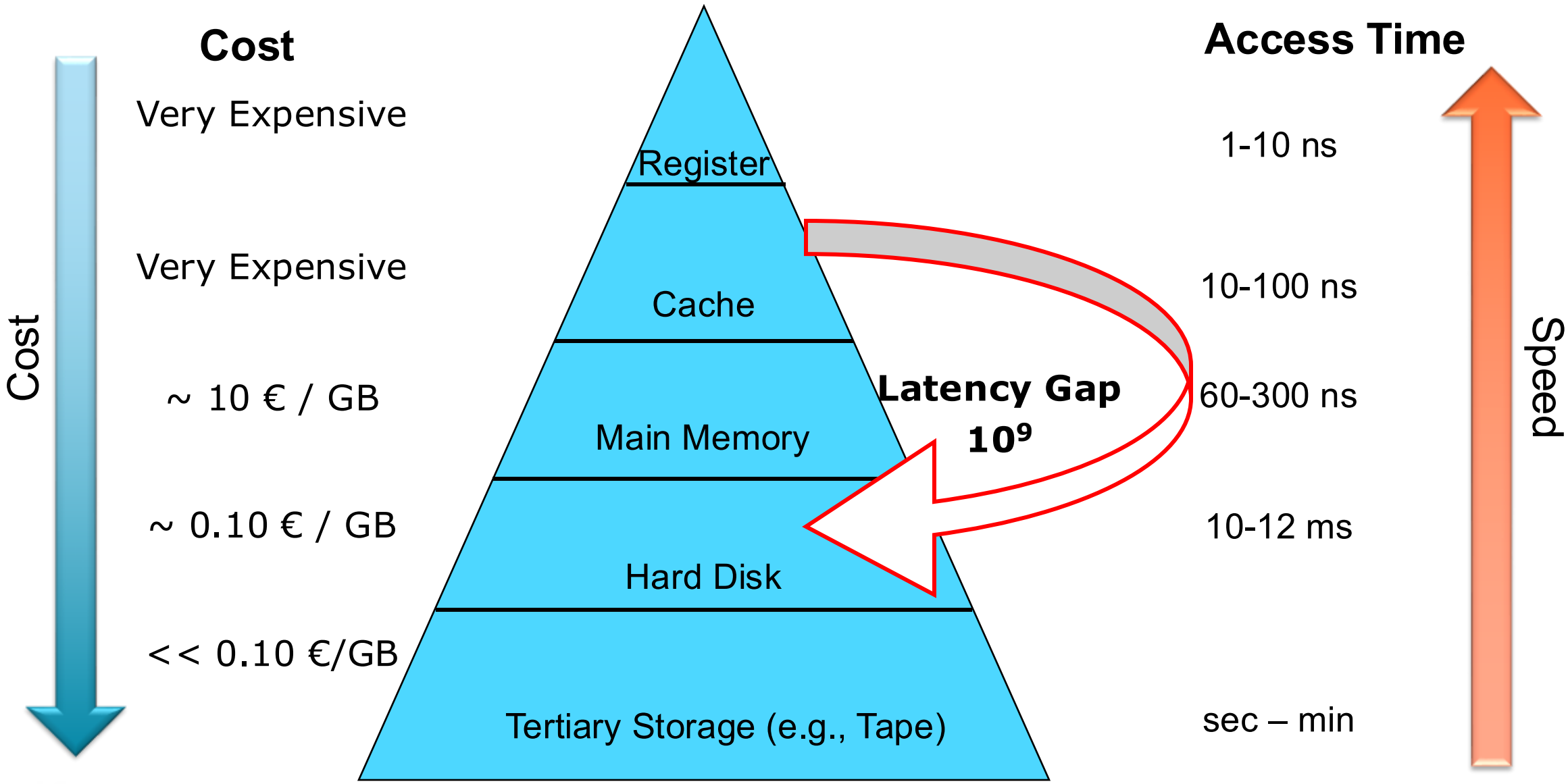
Memory Hierarchy

a.k.a. the Storage Pyramid

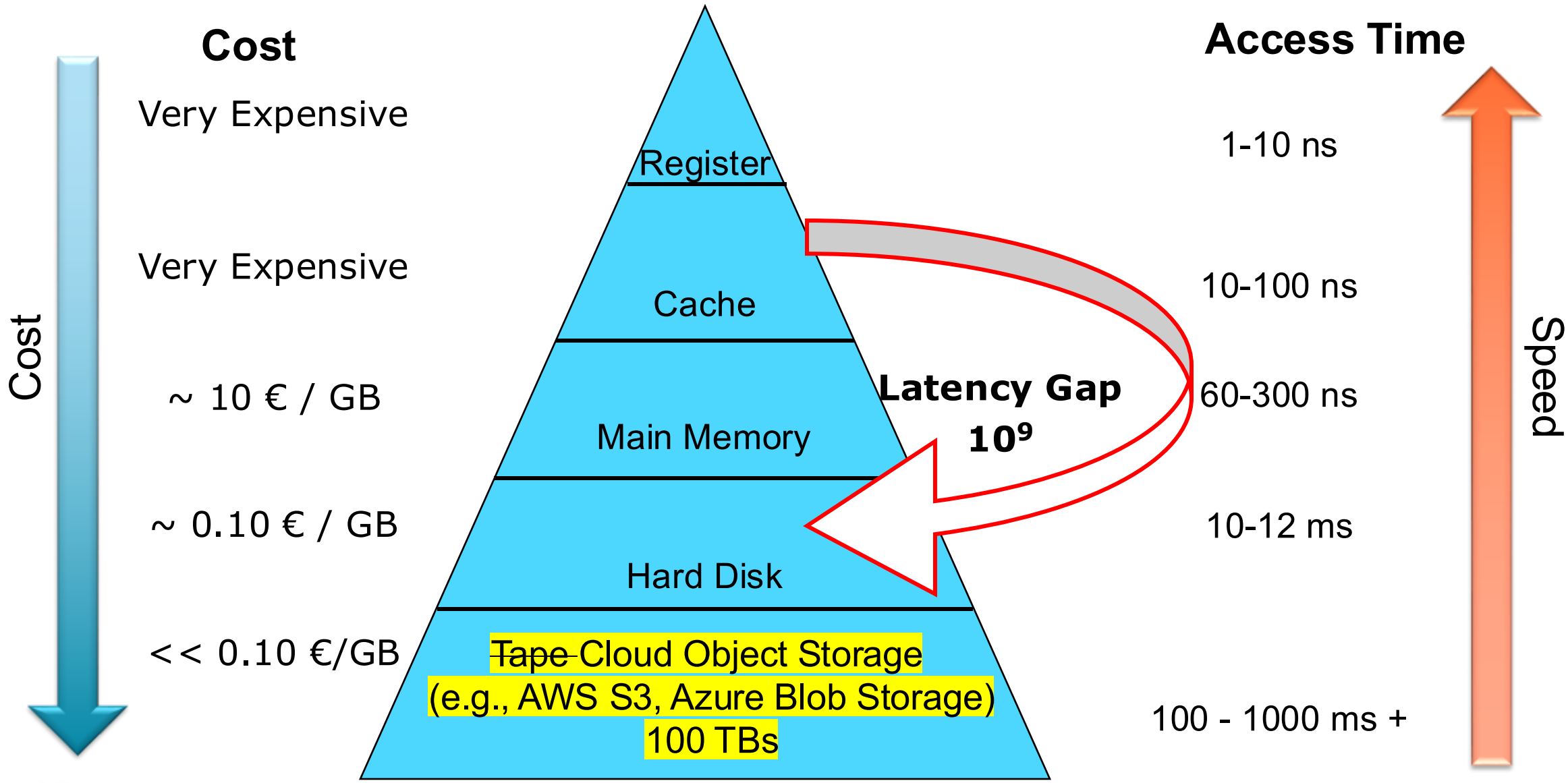
The Storage Hierarchy

- Media characteristics result in a **storage hierarchy**
- DBMS optimize data distribution among the storage levels
  - **Primary Storage (e.g., RAM):** Fast, limited capacity, high price, usually volatile electronic storage
    - Frequently used data / current work data
  - **Secondary Storage (Hard Disk / SSD):** Slower, large capacity, lower price
    - Main stored data
  - **Tertiary Storage (Tape):** Even slower, huge capacity, even lower price, usually offline
    - Backup and long-term storage of not frequently used data

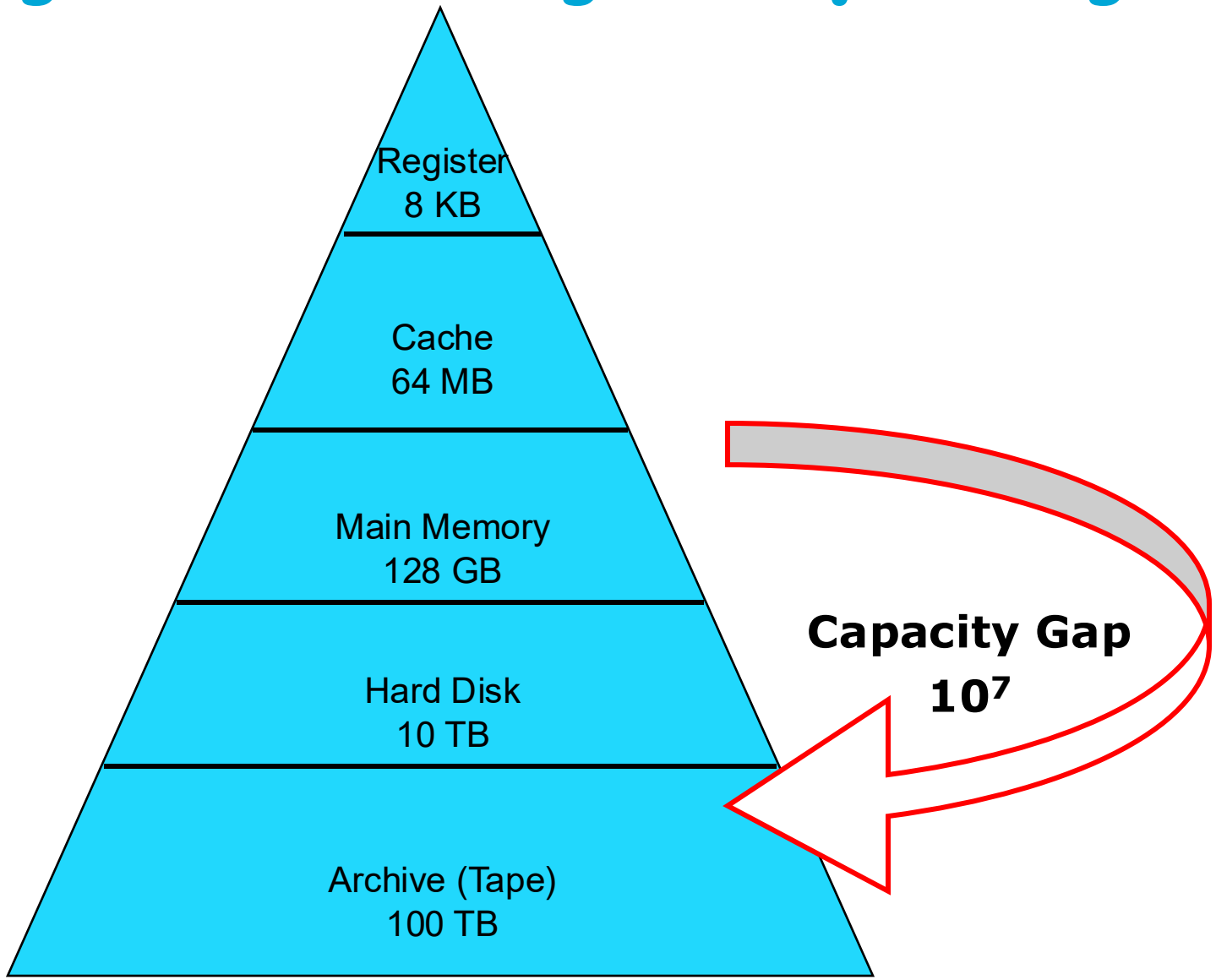
Memory Hierarchy: Cost vs. Access Time



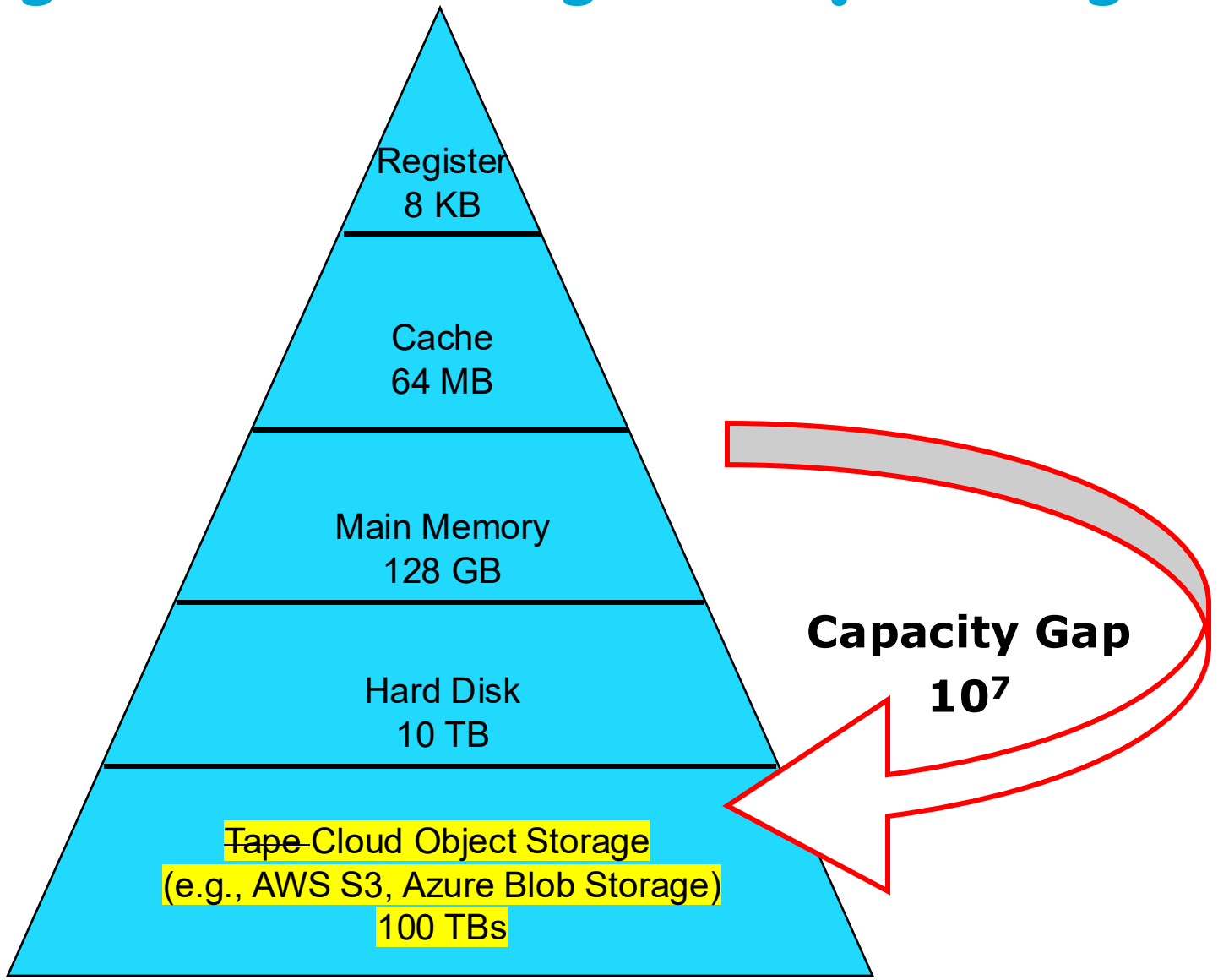
Memory Hierarchy: Cost vs. Access Time



Memory Hierarchy: Capacity



Memory Hierarchy: Capacity



(Spinning) Hard Disks

How they work, and why you should care.

Magnetic Disk Storage – HDDs

- Hard disc drives are (still) the standard for large, cheap and persistent storage
  - Usually used as the **main storage media** for most data in a DB



- DBMSs need to be **optimized** for efficient disk storage and access
  - **Data access**: as fast as possible
  - **Often-used data**: should be accessible with highest speed,
  - **Rarely-used data**: may take longer
  - **Data items used together** (e.g., two tables for a join): should also be stored/accessed together

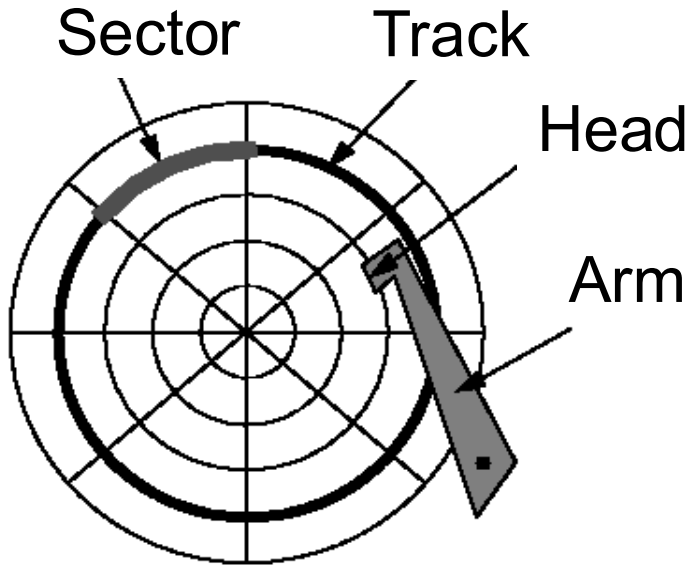
Demonstration: Hard Disk Spinning

- https://www.youtube.com/watch?v=3owqvmMf6No

HDD: Disk Structure



- **Tracks:** each surface is divided into circular tracks
- **Sector** (e.g., 1-8 KB) is the smallest **physical transfer unit**
  - Size is fixed by manufacturer
  - More sectors on outer tracks of disk
- **Gaps** between sectors make 10% of a track
  - Not magnetized
  - Serve to find the start of a sector
- **Platters** are fixed on the main spindle
  - Rotate at constant speed (5400 rpm - 10000 rpm)
- **Head** is attached to an arm
- **Arms** can position heads along the surface
- **Blocks** are logical transfer unit (*more later*)
  - Can consist of one or more sectors



Here: every track has same amount of sectors (simplification, different in reality)

Disk-Access Characteristics

- **Sector access** (read or write)
  - R/W-head positioned such that
    - While disk rotates sectors appear under R/W-head
- **Latency**
  - Time between request to read a block and arrival of the data block in main memory

Latency

Latency consists of 4 components:

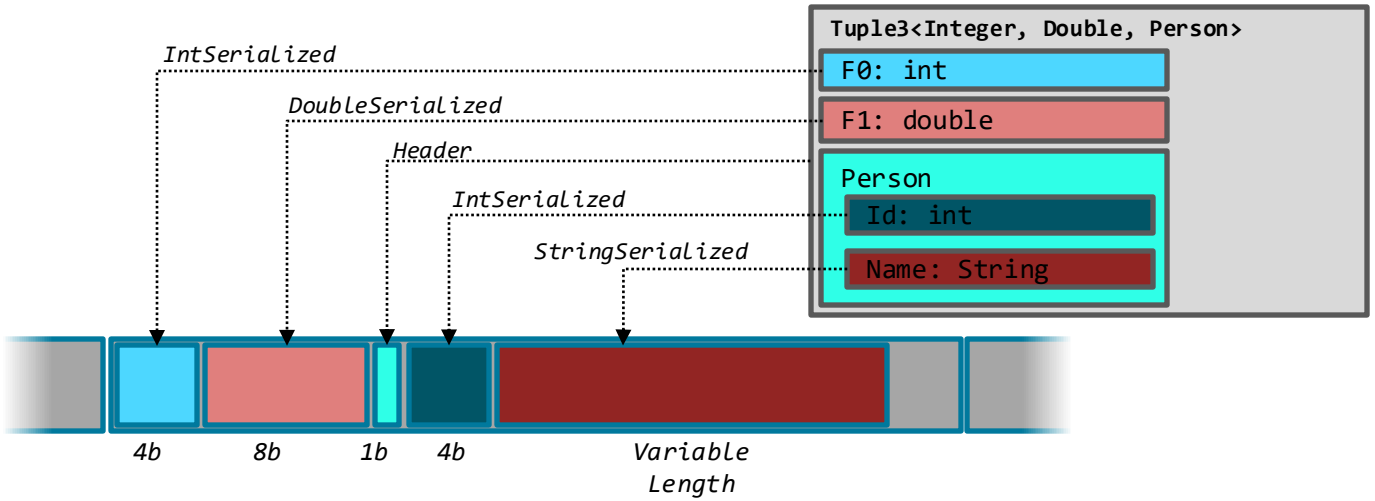
1. **Communication time** between processor and disk controller
   - Fraction of a millisecond → can be ignored in this course
2. **Seektime** to position R/W-head at the right cylinder
   - Between 0 and 42 ms → proportional to distance
   - Start time (1 ms), movement time (10 – 40 ms), Stop time (1 ms)
3. **Rotational delay** until first sector of the block appears under R/W-Head
   - On average ½ rotation (5 ms)
   - Optimization by track cache in disk controller possible
4. **Transfer time** during disk rotation until all sectors and gaps of the block have passed the R/W head
   - Ca. 16 KB-Block in ½ ms

Buffers / Pages

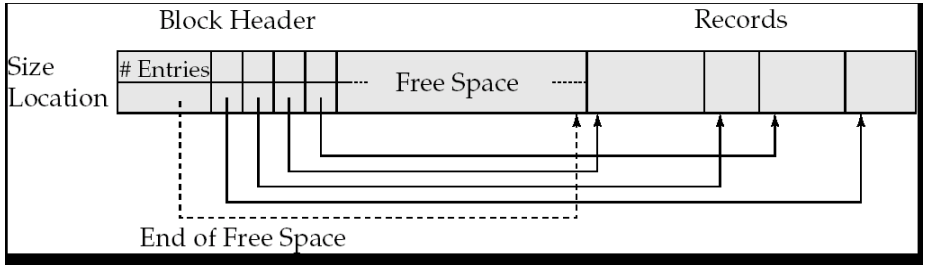
Possibly the most important topic today.

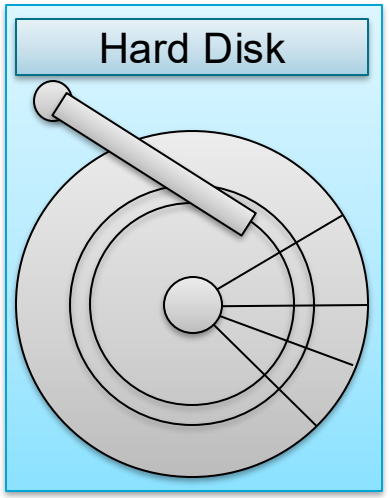
Data Storage on Hard Disks

Anatomy of a Record (in Row format)



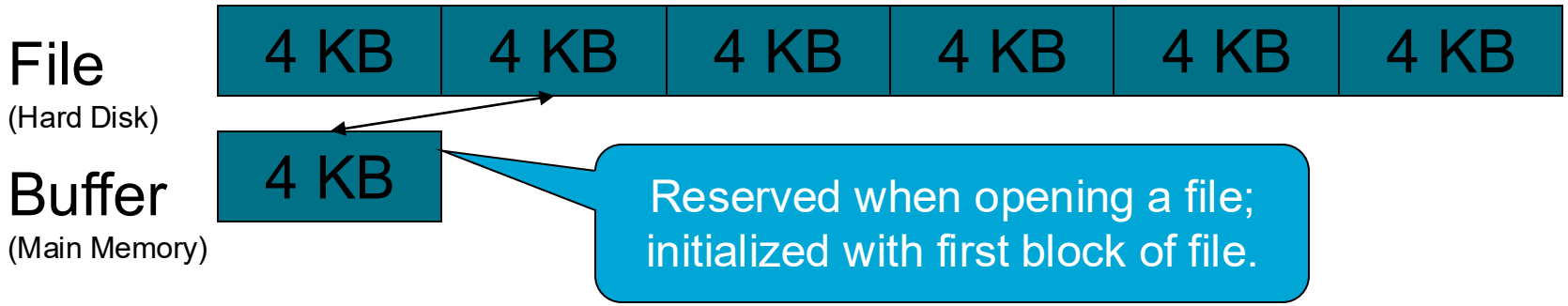
Anatomy of a Block





Hard Disk Buffers

- Main memory buffers parts of files
  - In block size (e.g., 4 KB)



- **DBMS do their own block management**
  - vs. operating system that does the same thing.
- Time for reading/writing a block: 10 – 30 ms (huge!)
  - During this time, CPU can execute many million instructions
  - I/O-time dominates overall cost
  - Ideally: Block should already be in main memory!

HDDs – Take away

- **Locating data on a disk is a major bottleneck**
- **Take-Away-Message HDD**
  - **Cheap and Large** Secondary Storage Medium
  - **Avoid** Random Accesses
    - Reposition head for each access
    - Seek Time + Rotational Delay + Block Transfer
  - **Aim** for sequential access
    - Position head once, and then just read sectors & transfer
  - **Aim** for re-using data already stored in the Buffers (RAM)
  - **Disks are mechanical** devices, and prone to mechanical failure

Efficient HDD Operations

I/O Costs Dominate

Disclaimer: if you replace **HDD** with **Cloud object storage**, the following slides still hold.



Writing and Changing of Blocks

- Writing of blocks
  - Algorithm and latency: like reading

- Changing of blocks
  - Not possible to do it without RAM (main memory)
    1. Read block into main memory
    2. Modify the data
    3. Write block back to disk
    4. Check for success
  - Latency: t_read + t_write
  - But with luck (i.e., nobody reads anything in the meantime)
    - R/W-head is still close to cylinder (t_write is cheaper then)

Algorithms Course vs. DBMS Course

- Usual assumption when analyzing algorithms:
  - RAM model of computation
    - All data fits into main memory
    - Data is already in main memory
- Assumption when implementing a database
  - I/O-Model: The whole data set does not fit in the main memory
    - Although lots of datasets do fit in memory nowadays (more later…)
- Algorithms for external storage (AES) are often different
  - Good AES algorithm need not be the best RAM model algorithm

- **Design goal: Avoid I/O**

I/O-Model of Computation

- Example: Simple DBMS
  - Too large for main memory
  - One disk, one processor, many concurrent queries/users

- Disk controller maintains queue with access requests
  - Processing principle: First-come-first-served (FCFS)
  - Requests appear in random order
    - R/W-head is at random position over disk

Definition (Dominance of I/O-cost)

- **The cost of transferring a block from disk (or cloud storage) to main memory is orders of magnitude larger than the cost of the operations in main memory**

→ **Corrolary:** the number of (read/write) block accesses is a good approximation of overall cost. Databases aim at minimizing block accesses

Two Ways to Avoid Random I/O

- Write data on disk in the order of reading
  - Not covered

- Use indexes for quickly finding data on disk
  - This lecture

Index Structures

Index Structures

- A good index structure should support queries as efficiently as possible
  - Avoid **block access** to secondary storage wherever possible!!
  - Query types:
    - Single Equality Condition "… WHERE x = 5"
      - Does such a record exist?
      - If it exists, where can we find it?
      - Hash Indexes and B^+ Trees do well
    - Range Conditions "… WHERE x > 5 AND x < 42"
      - Traverse index efficiently
        - » Maybe it is even better to not use the index at all? → Next week!
      - $^+$ Trees do well

Index Structures

- What about other queries?
  - Queries involving multiple attributes?
    - "… WHERE x = 5 AND y = 42"
    - "… WHERE x > 5 AND x < 999 AND y > 4 AND y < 1234"
    - We have multi-dimensional trees!
      - R-Tree!
      - K-D-Tree!
      - M-Tree!
      - …
  - Systems which have very low write load, but high read load?
    - Bitmap indexes!
  - Queries where you only need to check if some records exist?
    - Bloom filters!
  - Systems which operate in main memory only?
    - CST tree!

This Lecture: Indexing

- Hash Indexes
- **B+Trees**
  - Structure
  - Search
  - Insert/Delete
  - Efficiency
  - Variations

B-Trees

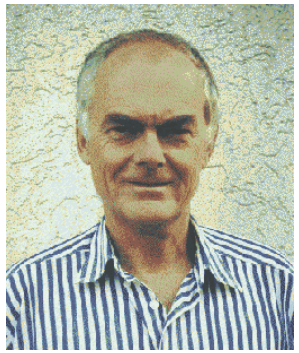
B-Trees

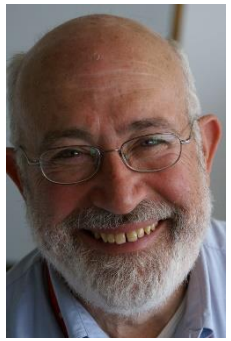
- B-Trees adapt concepts and techniques learned for binary trees and **optimize** them for **hard disk storage**
- **Basic Ideas**:
  - **Searching within** a DB/disk block is very efficient
    - Take advantage of the static nature within a block
    - Search can be performed in memory with **bisection search**
    - Treat entire **blocks** as **tree nodes**
  - **Reading blocks** from the disk is expensive
    - Reduce block reads
    - Most data resides in the **leaf nodes**
  - Thus **minimize the height** of the tree
    - Dramatically increase fan-out factor
    - Tree becomes "**bushy**"
    - Smaller search path length

B-Trees

- **B-Tree** are specialized **Block Search Trees** for disk-based **Indexing**
  - Invented by Rudolf Bayer and Ed McCreight in 1971
  - Keys may be **non-unique**
  - Tree is **self-balancing**
    - No degenerated cases anymore

- R. Bayer, E. McCreight, E. (1972), "Organization and Maintenance of Large Ordered Indexes", Acta Informatica 1 (3): 173–189





B^+-Trees

(from this moment, B-Tree will refer to a **B^+-Tree**)

B+-Trees

- Summary
  - **B+ Tree** is **THE** super index structure for disk-based databases
  - Improved over B-Tree
    - Improved traversal performance
    - Increased search efficiency
    - Increased memory efficiency



Safety Bike (1876)



vs.

Road Bike (2024)



Working principles are the same!

This Lecture: Indexing

- **Hash Indexes**
- **B^+-Trees** **(the "safety bike" of indexing structures)**
  - **Structure**
  - Search
  - Insert/Delete
  - Efficiency

B+-Trees

- In general: B+-Trees
  - As many levels as necessary
  - Blocks are at least 50% full
  - No overflow blocks needed

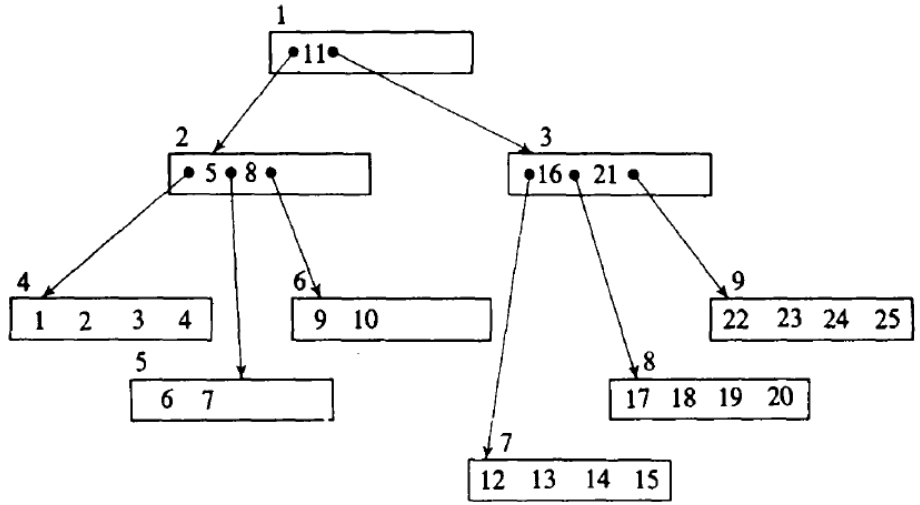


Fig. 2. A data structure in $\tau(2, 3)$ for an index

Structure

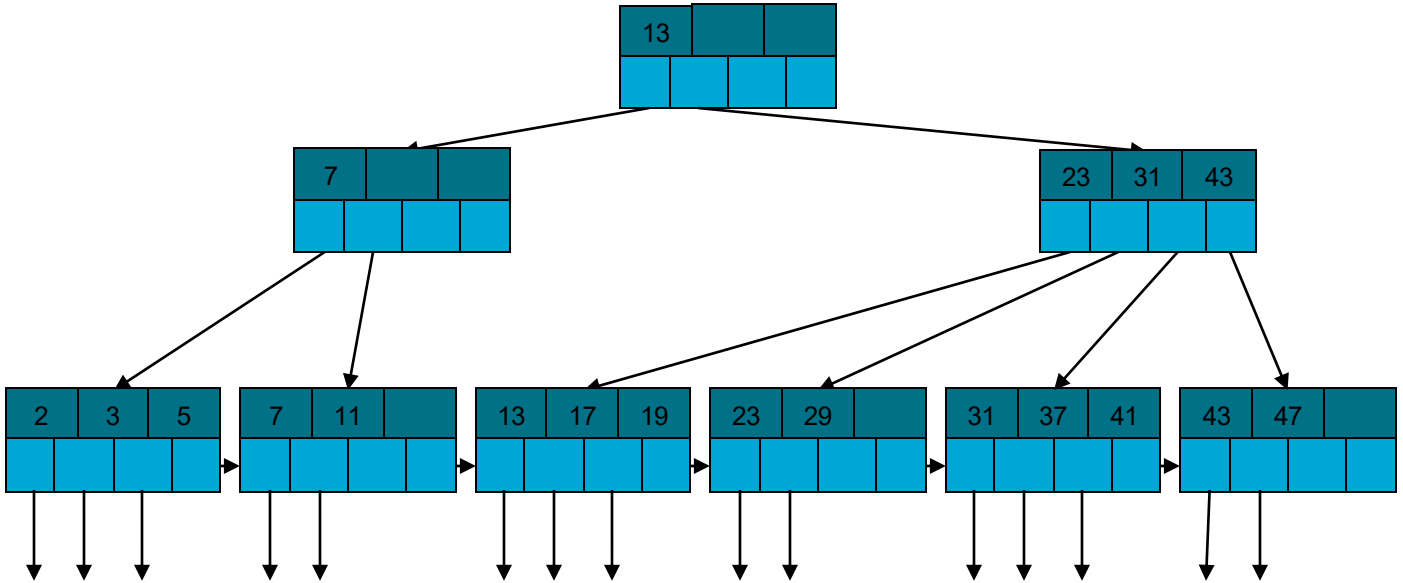
- Index blocks organized in a tree
- Balanced
  - Each path from root to leaf has the same length
- Parameter n
  - Every block holds up to n search keys
  - Every block holds up to n+1 pointers
  - Just like the index block before, but one additional pointer
- Choice of n
  - n as large as possible with respect to block size
  - 4096 Bytes per block; 4 Bytes per key; 8 Bytes per pointer
  - $4n + 8(n+1) \leq 4096 \Rightarrow n = 340$

Definition of B^+-Trees

- Keys in the leaves are keys from the data
  - Sorted across all leaves (from left to right)
- Root: At least two used pointers.
  - All pointers point to B^+-tree block one level below
- Leaves: The last pointer points to the next leaf (on the right)
  - From the other n pointers at least $\lfloor (n+1)/2 \rfloor$ are used
  - Point to data blocks
- Inner Nodes: Pointers point to B^+-tree blocks of level below
  - At least $\lceil (n+1)/2 \rceil$ are used
  - If j pointers are used, there are j-1 keys in the block
    - $K_1, \ldots, K_{j-1}$
  - First pointer points to subtree with key values < K_1
  - Second pointer points to subtree with key values between K_1 and K_2.

Example Calculations

- n = 3
  - All nodes: At most 3 keys and 4 pointers
  - Root: At least 1 key and 2 pointers
  - Inner Nodes: At least 1 key and 2 pointers
  - Leaves: At least 2 keys and 3 pointers

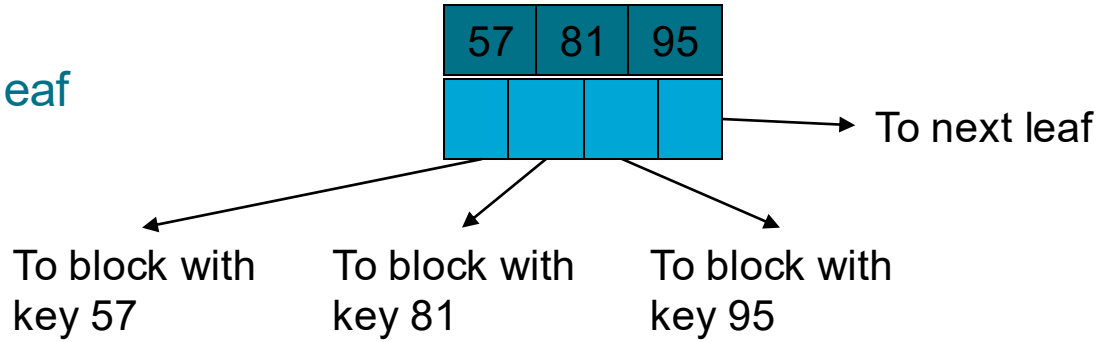


Example Calculations

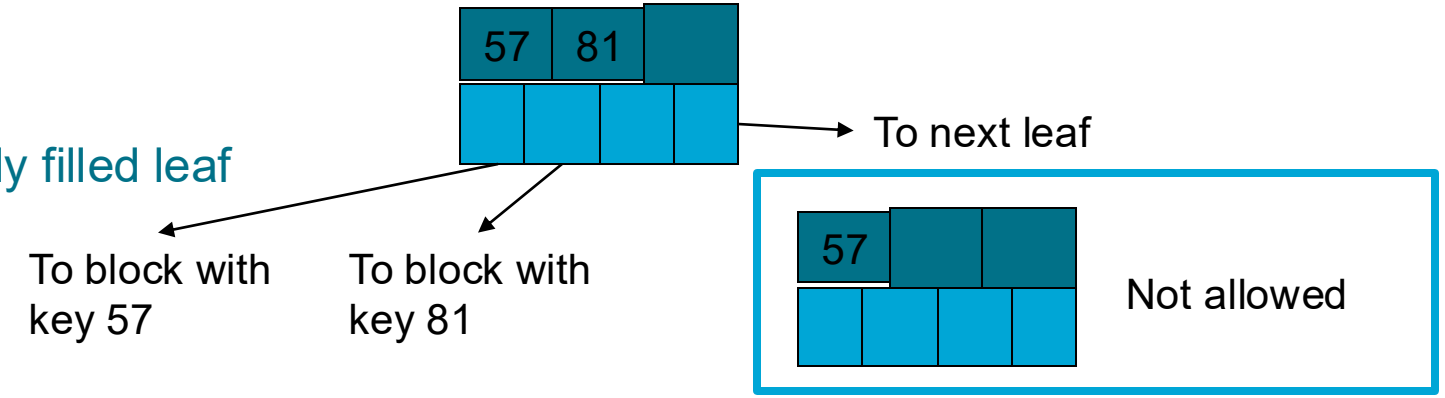
- n = 4
  - All nodes: At most 4 keys and 5 pointers
  - Root: At least 1 key and 2 pointers
  - Inner nodes: At least $\lceil (n+1)/2 \rceil$ pointers = 3 pointers
    => At least 2 keys
  - Leaves:
    - 1 pointer to next leaf + at least $\lfloor (n+1)/2 \rfloor$ further pointers =
      = 3 pointers
      => At least 2 keys
- n = 5
  - All nodes: At most 5 keys and 6 pointers
  - Root: At least 1 key and 2 pointers
  - Inner Nodes: At least $\lceil (n+1)/2 \rceil$ pointers = 3 pointers
    => At least 2 keys
  - Leaves:
    - 1 pointer to next leaf + at least $\lfloor (n+1)/2 \rfloor$ further pointers =
      = 4 pointers
      => At least 3 keys

Examples of Leaves

- n = 3
  - 3 keys and 4 pointers
- Full leaf

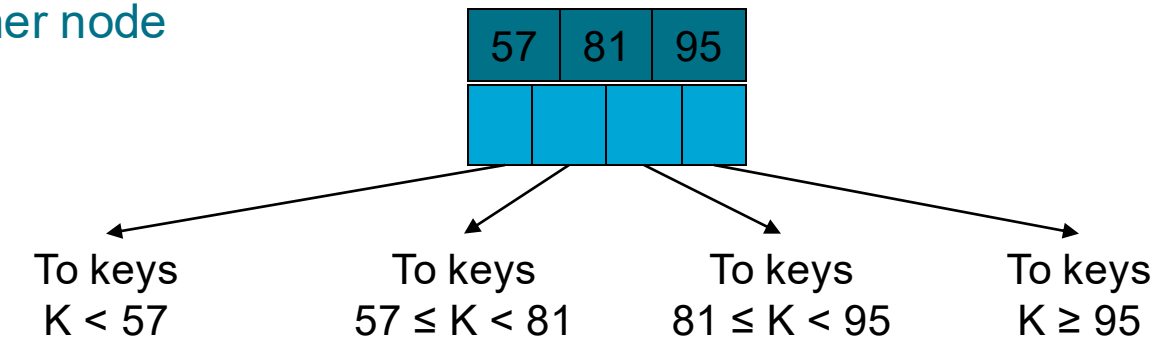


- Partially filled leaf

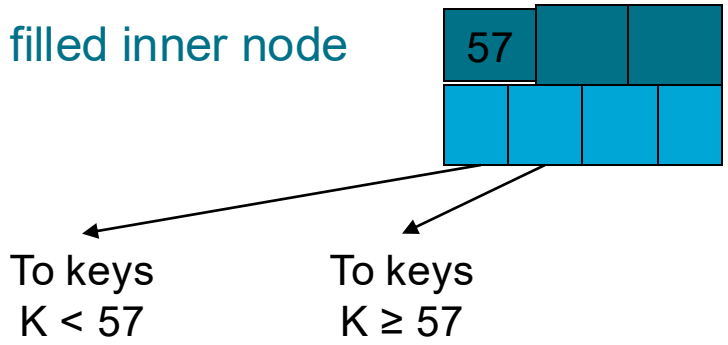


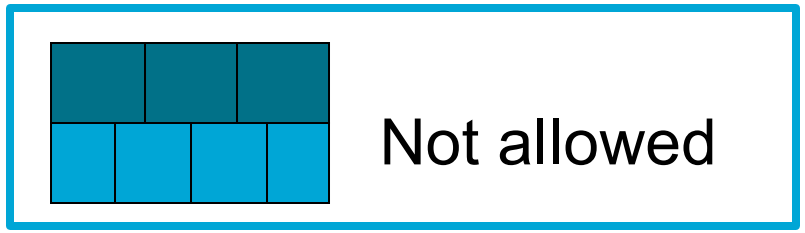
Examples of Inner Nodes

– Full inner node

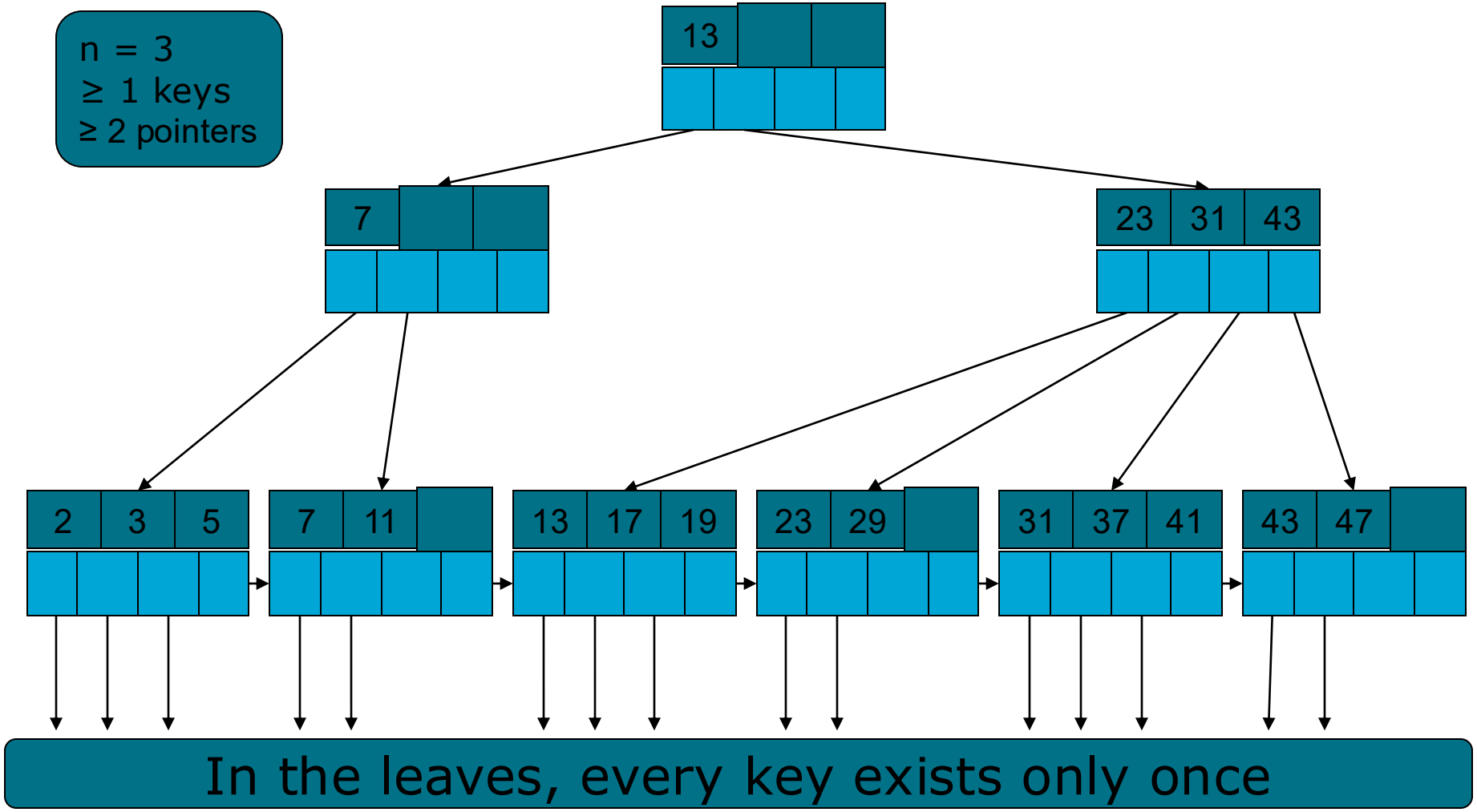


– Partially filled inner node





Example B+-Tree



Applications of B$^+$-Trees

B$^+$-trees can assume different index roles:

- Search key is the primary key
  - Dense index
    - Order of data file is not important
  - Sparse index
    - Data file must be sorted by primary key
- Search key is NOT the primary key
  - Search key is not unique
  - Data file is sorted by search key
    - Sparse index possible
  - Data is NOT sorted by search key
    - Dense index required

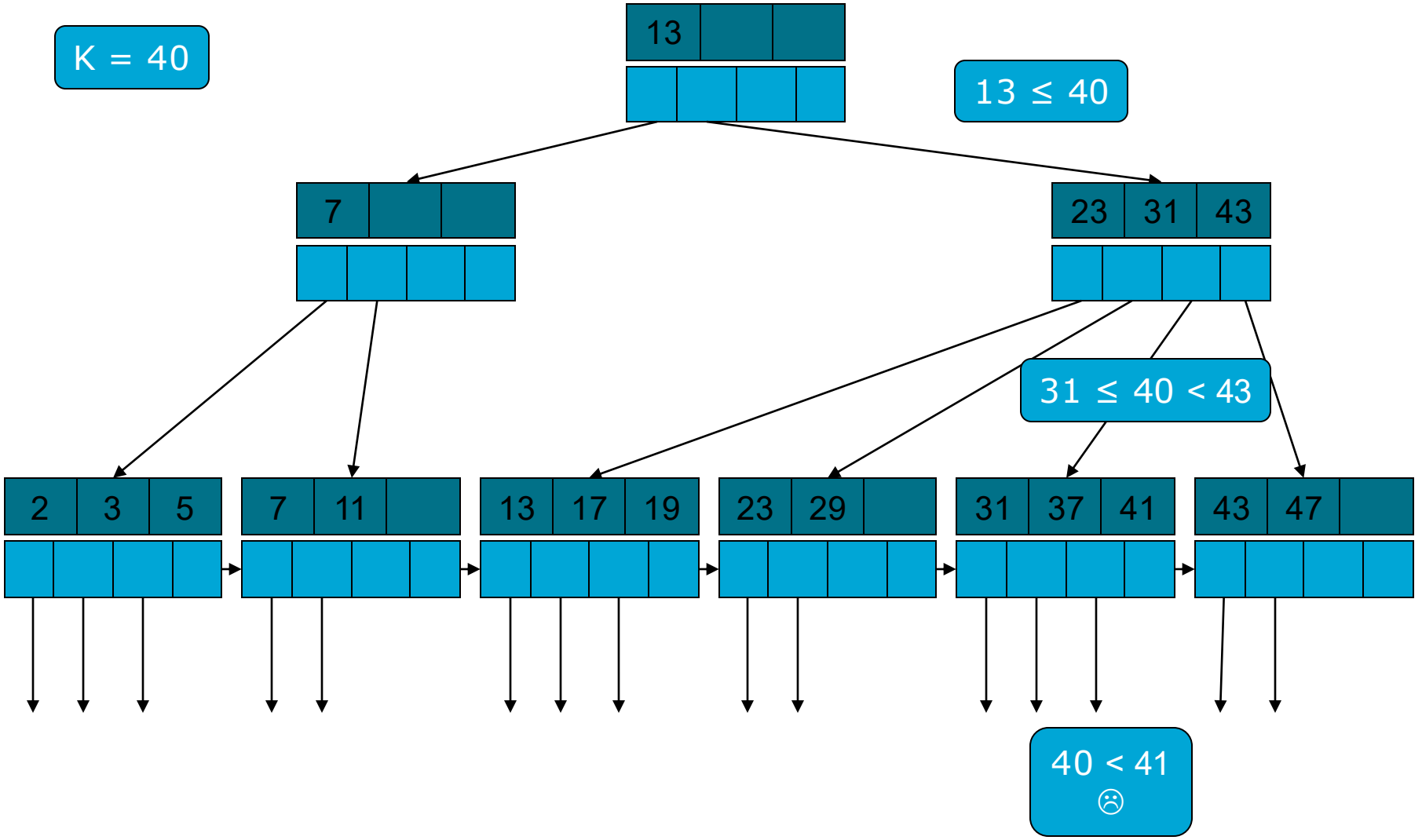
This Lecture: Indexing

- **Hash Indexes**
- **B$^+$-Trees**
  - Structure
  - **Search**
  - Insert/Delete
  - Efficiency

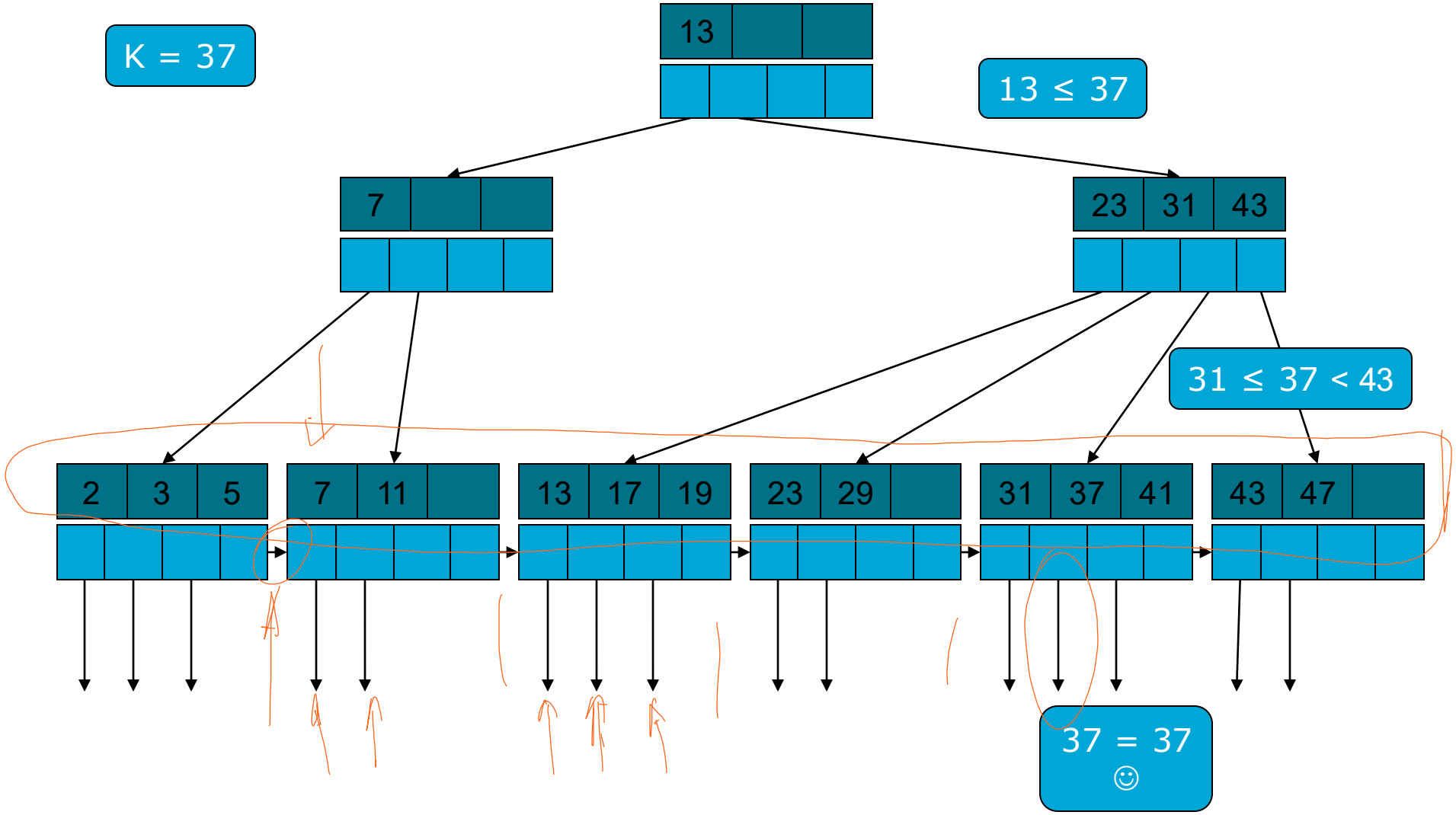
Searching in B$^+$-Trees

- Now again: Search key = primary key
- Dense index
  - Operations for sparse indexes similar
- Find K
  - If we are at a leaf:
    - Search K on the leaf
  - If we are at an inner node with keys K_1, K_2, …, K_n:
    - If $K < K_1$ go to first child
    - If $K_1 \leq K < K_2$ go to second child
    - …
    - If $K_n \leq K$ go to the last child

Example: Searching in B+-Trees



Example: Searching in B+-Trees



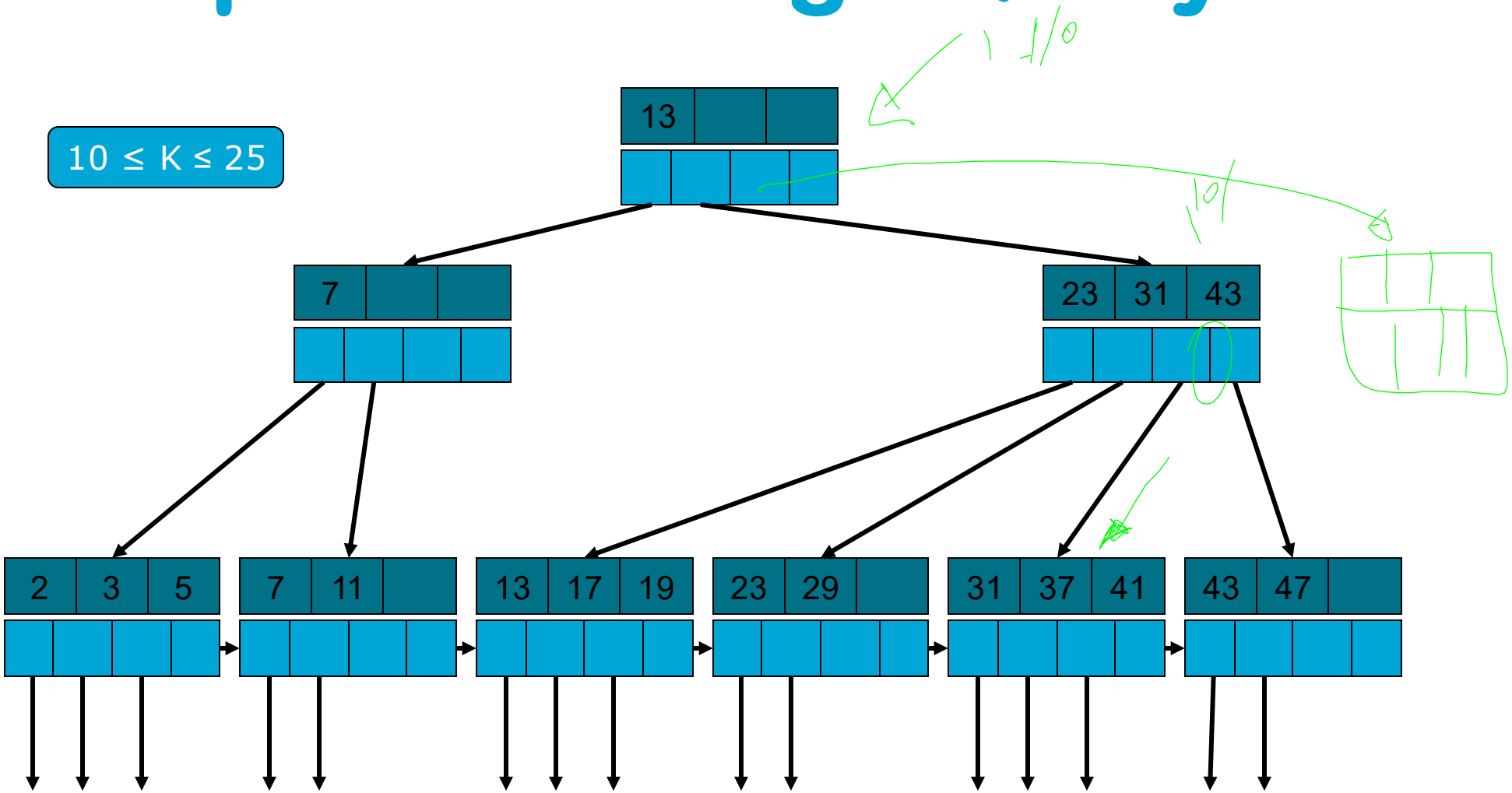
Range Queries

- Queries with inequality in WHERE-clause
  - SELECT * FROM R
    WHERE R.k > 40
  - SELECT * FROM R
    WHERE R.k >= 10 AND R.k <= 25

- Searching for a range [a, b] (BETWEEN)
  - Find a in B^+-tree:
    - Corresponding leaf could contain a.
    - Find all keys on the leaf satisfying the range predicate.
  - If no key on leaf > b:
    - Follow the pointer to the next node.

- Similar for open intervals [-∞ , b] respectively [a, ∞]

Example of a Range Query



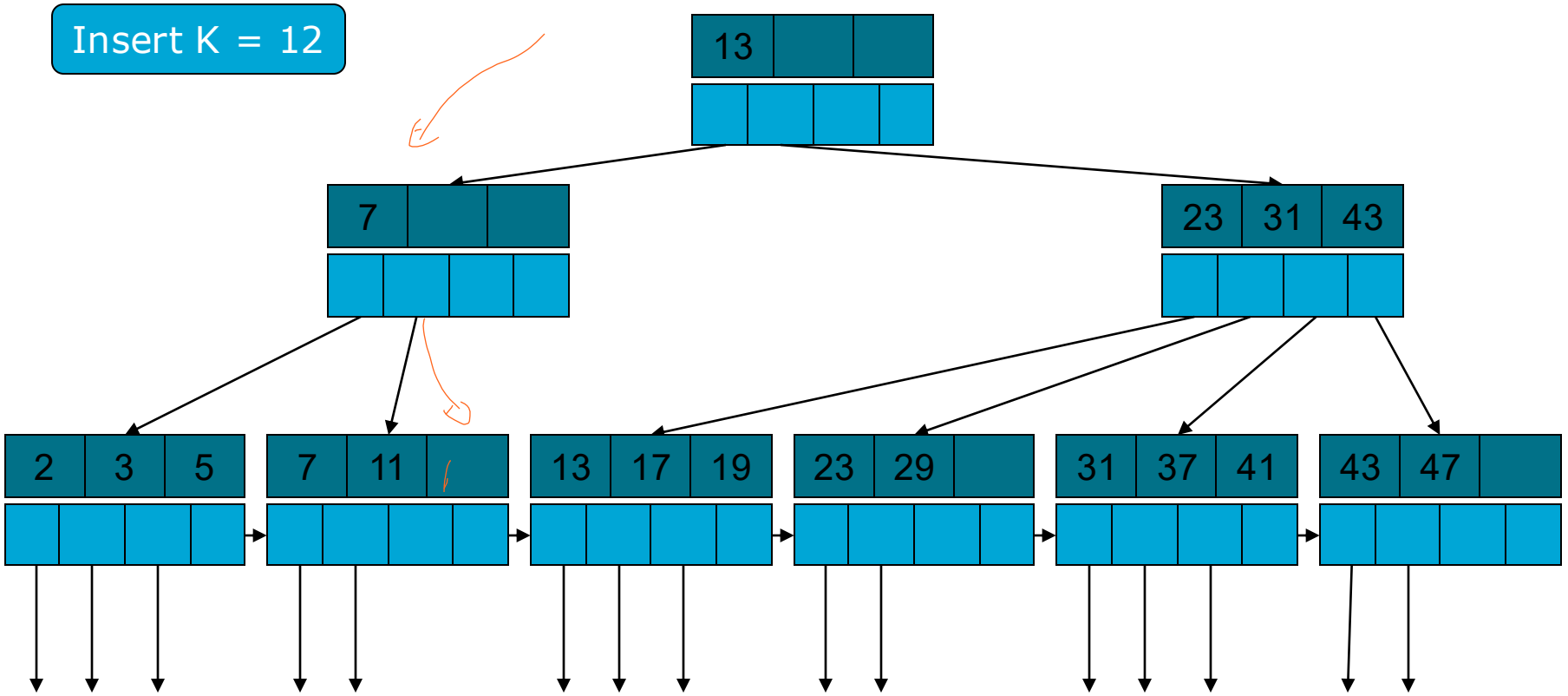
This Lecture: Indexing

- B^+-Trees
  - Structure
  - Search
  - **Inserting/Deleting**
  - Efficiency

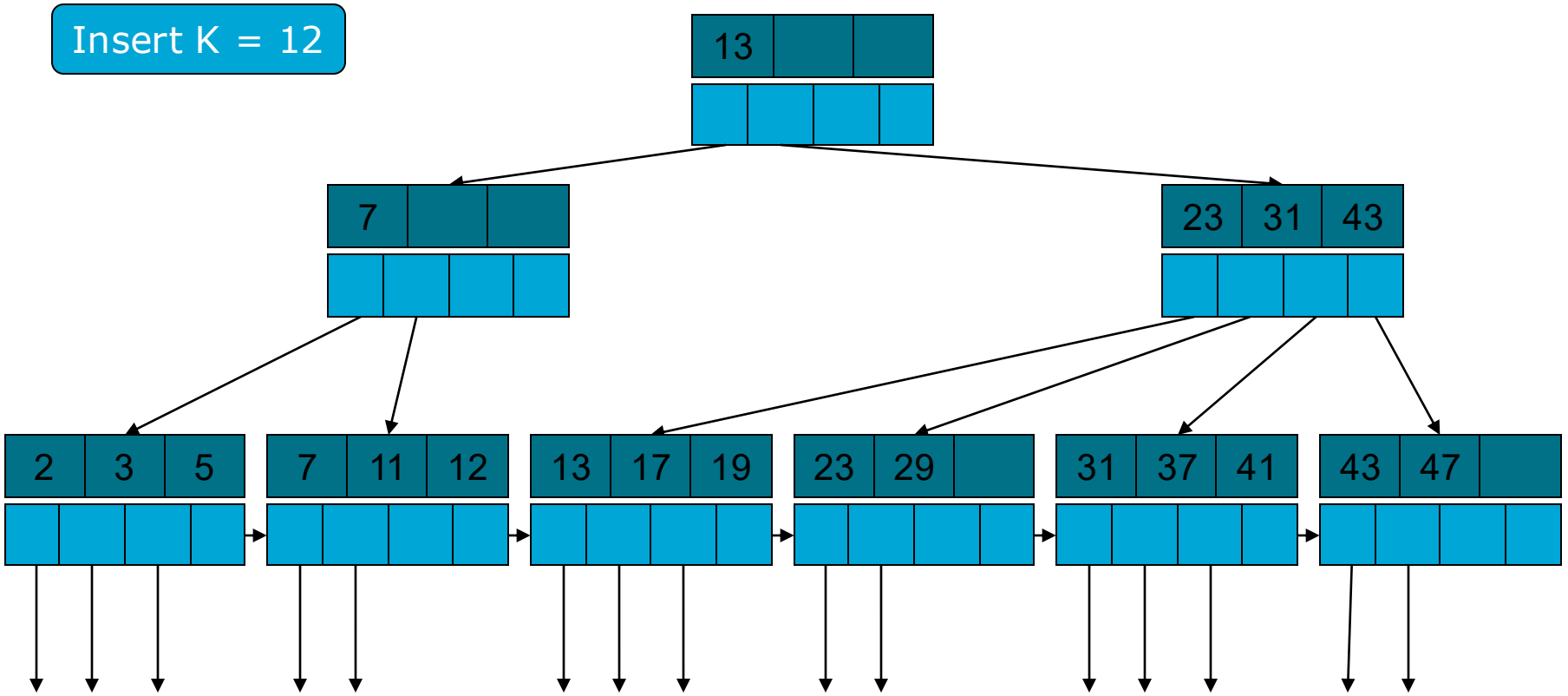
Inserting into a B+-Tree

- Recursive Algorithm:
  - Search the corresponding leaf
    - If room, insert key and pointer
  - If no room: Overflow
    - Split the leaf in two parts and distribute the keys equally
  - Split requires inserting a new key/pointer pair in the parent node
    - Recursively ascend the tree
  - Exception: If no space in root
    - Split root
    - Create a new root (with only one key)

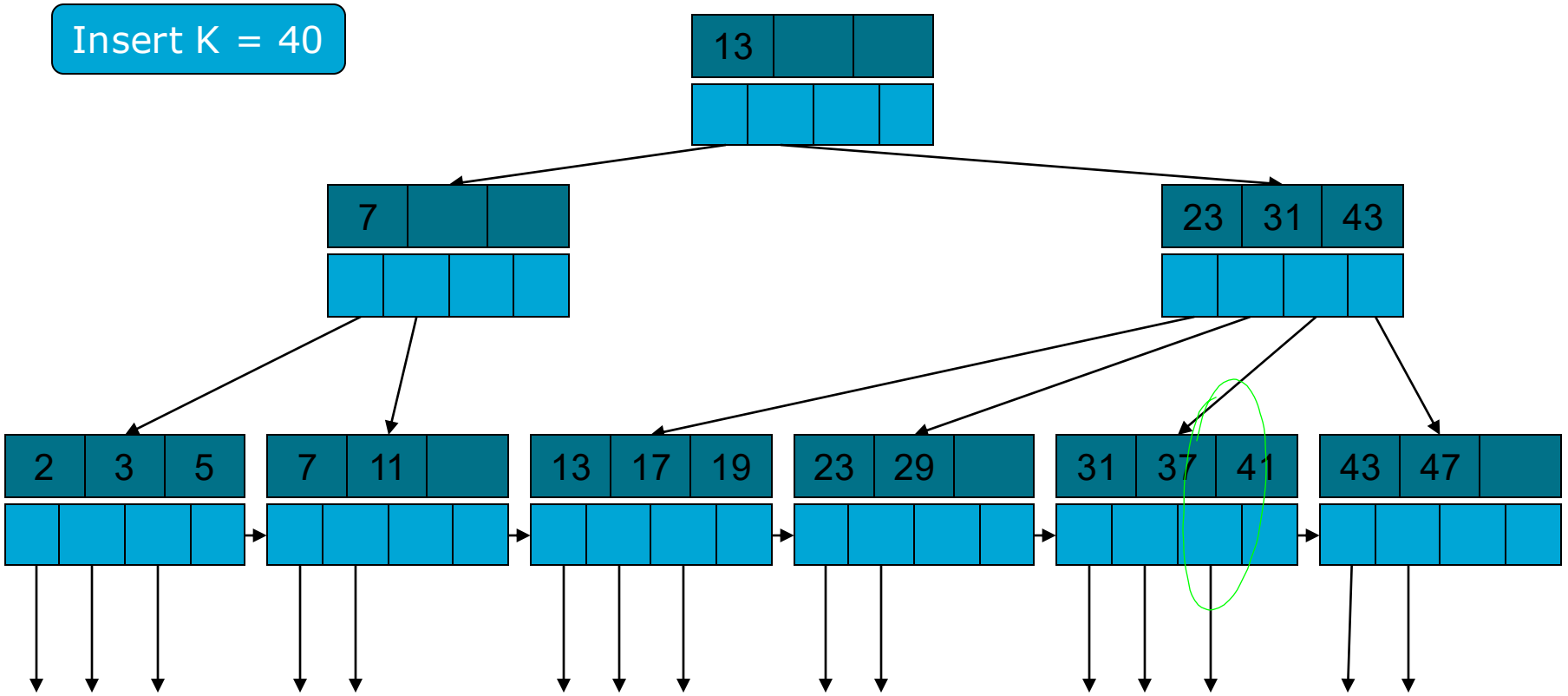
Example of B$^+$-Tree Insertion



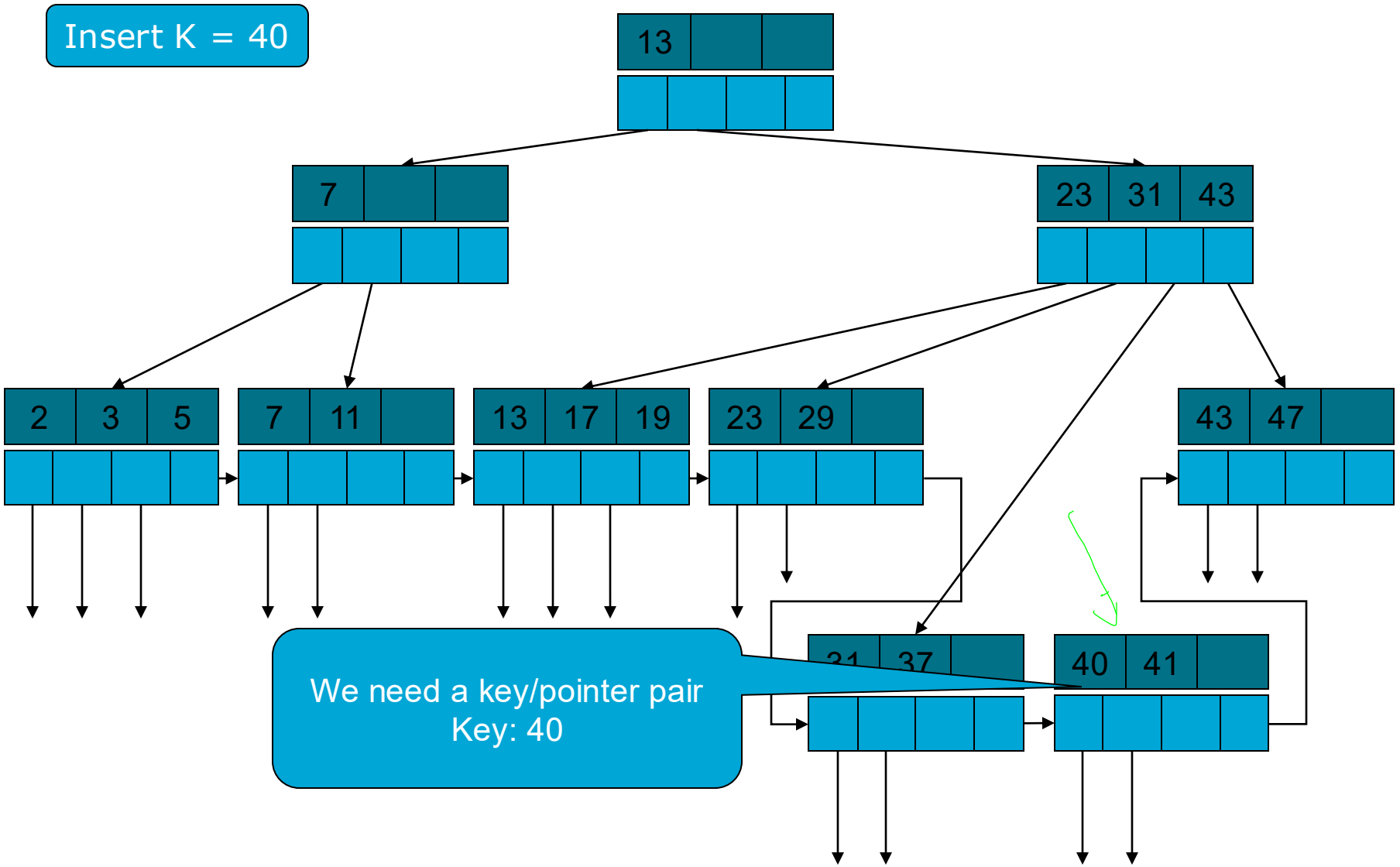
Example of B$^+$-Tree Insertion



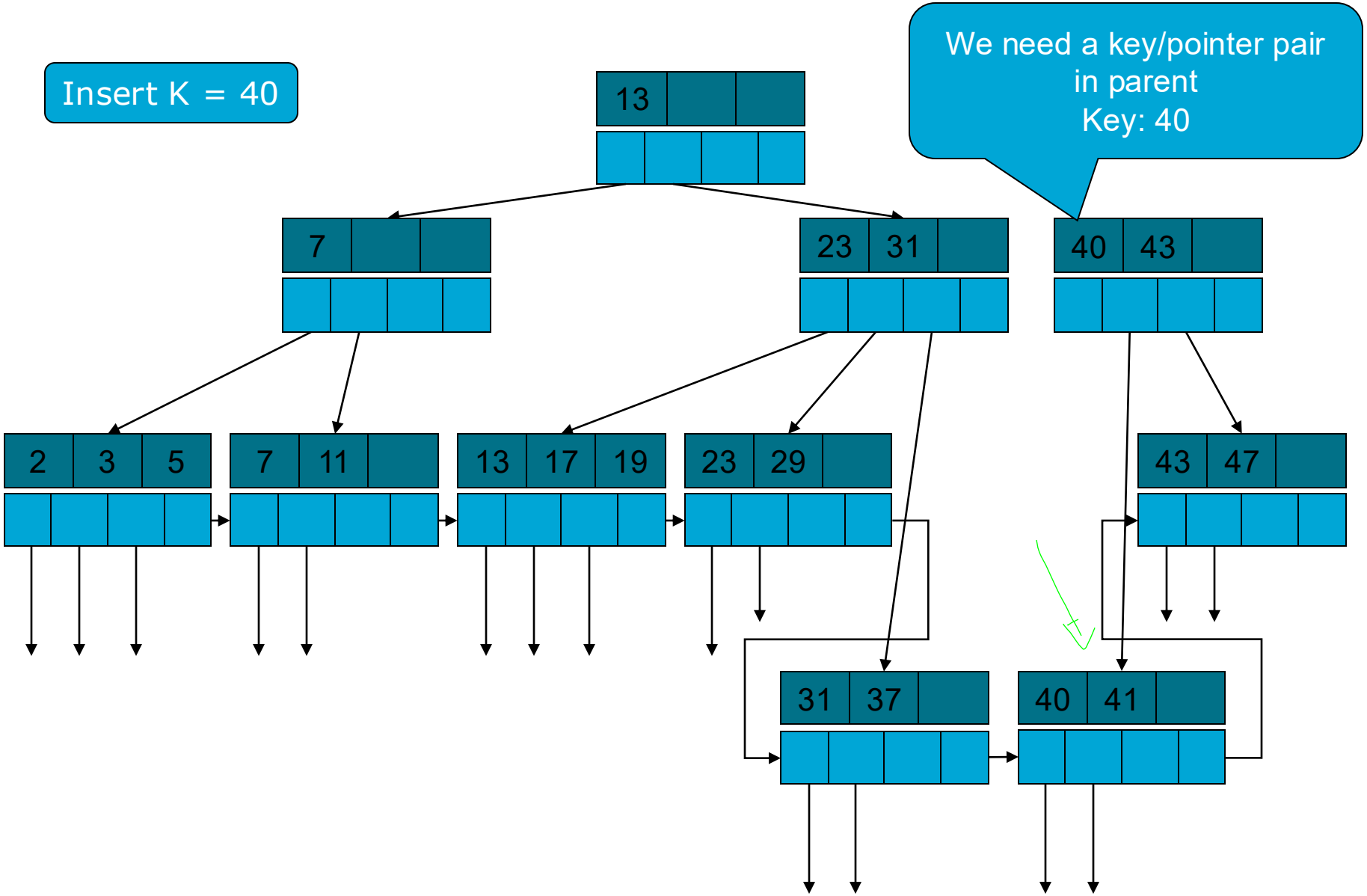
Example of B+-Tree Insertion



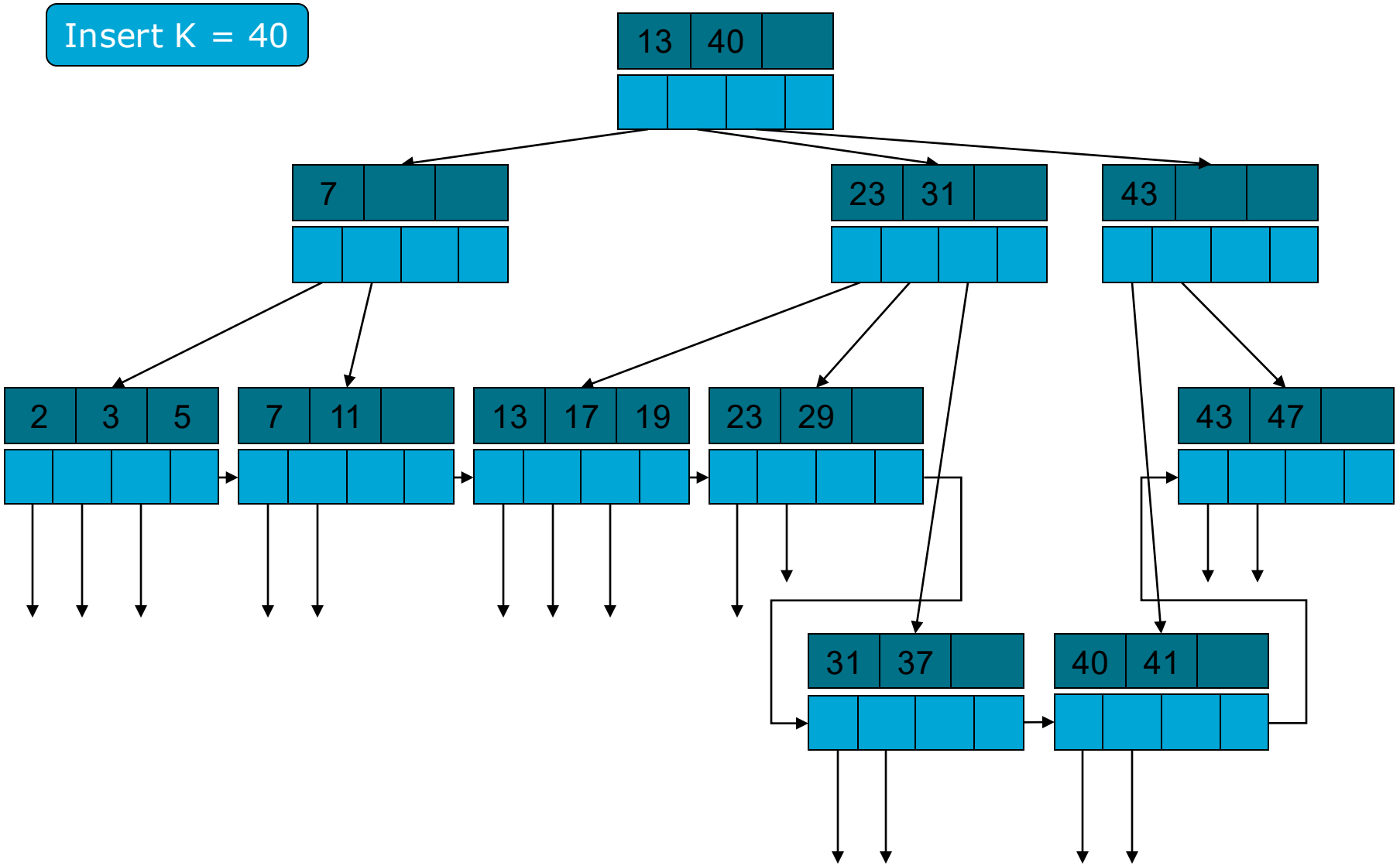
Example of B$^+$-Tree Insertion



Example of B+-Tree Insertion



Example of B$^+$-Tree Insertion



Insertion Cost

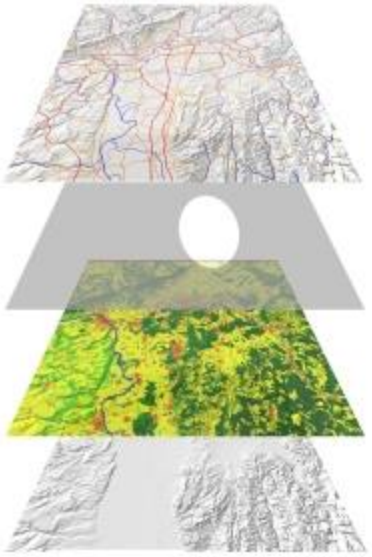
- Let h be the height of the B^+-Tree
  - In practice often: h = 3
- Search for leaf node: h
- If no split is needed: Total cost h + 1
  - h block reads, 1 block write
- If a split is needed
  - Worst-case: Ascends to root
  - Even caching useless: Nodes need to be written to disk
  - Total cost: 3 h + 1
    - On every level search and overflow handling

Multidimensional Indexes

Geographical Information Systems (GIS)



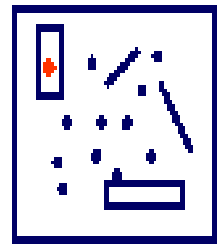
- Usually 2 Dimensions
  - Points or areas on a plain
  - Houses, Roads, Bridges, Pipes, …
- Also:
  - Mechanical CAD
  - VLSI
  - Circuit design
  - Windows
  - …



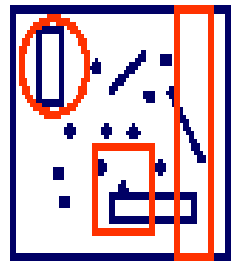
- Typical queries:
  - Partial queries: Only some of the dimensions are restricted
  - Range queries: Range specifies area and searches data in it
    - Subset, touches, overlaps
  - Nearest neighbor: Nearest point to a given point
    - E.g., next large city
  - Where-am-I-queries: Given a point,on which plain is it located?
    - E.g., windows-clicks, mouse-over

For simplicity: two-dimensions

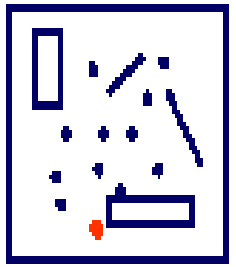
- Queries:
  - Point queries
  - Range Query: “Find all restaurants within a given region.”
  - Nearest Neighbor Query: “Find the nearest restaurant to my house.”
    - What images are similar to this one?
    - LLM: What is the most relevant answer in a document, given a question?



Point Query



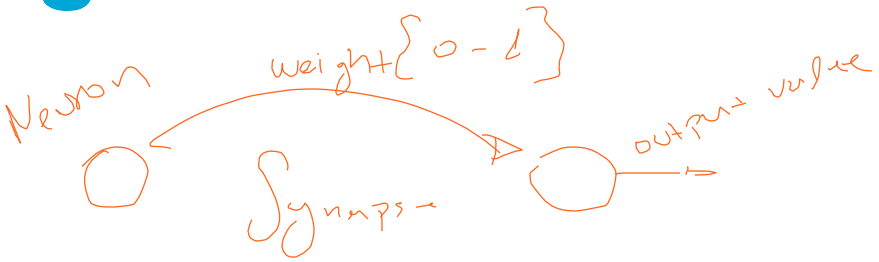
Range Query

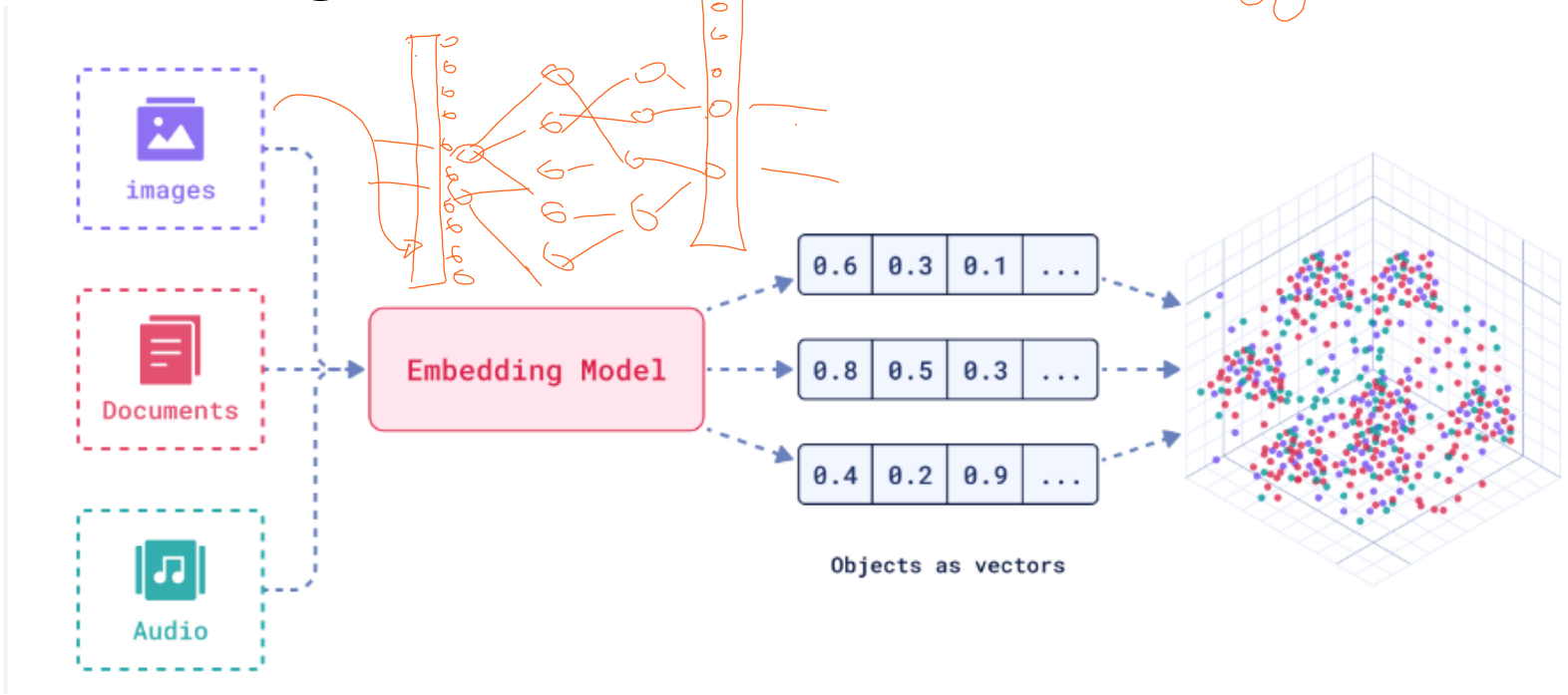


NN Query

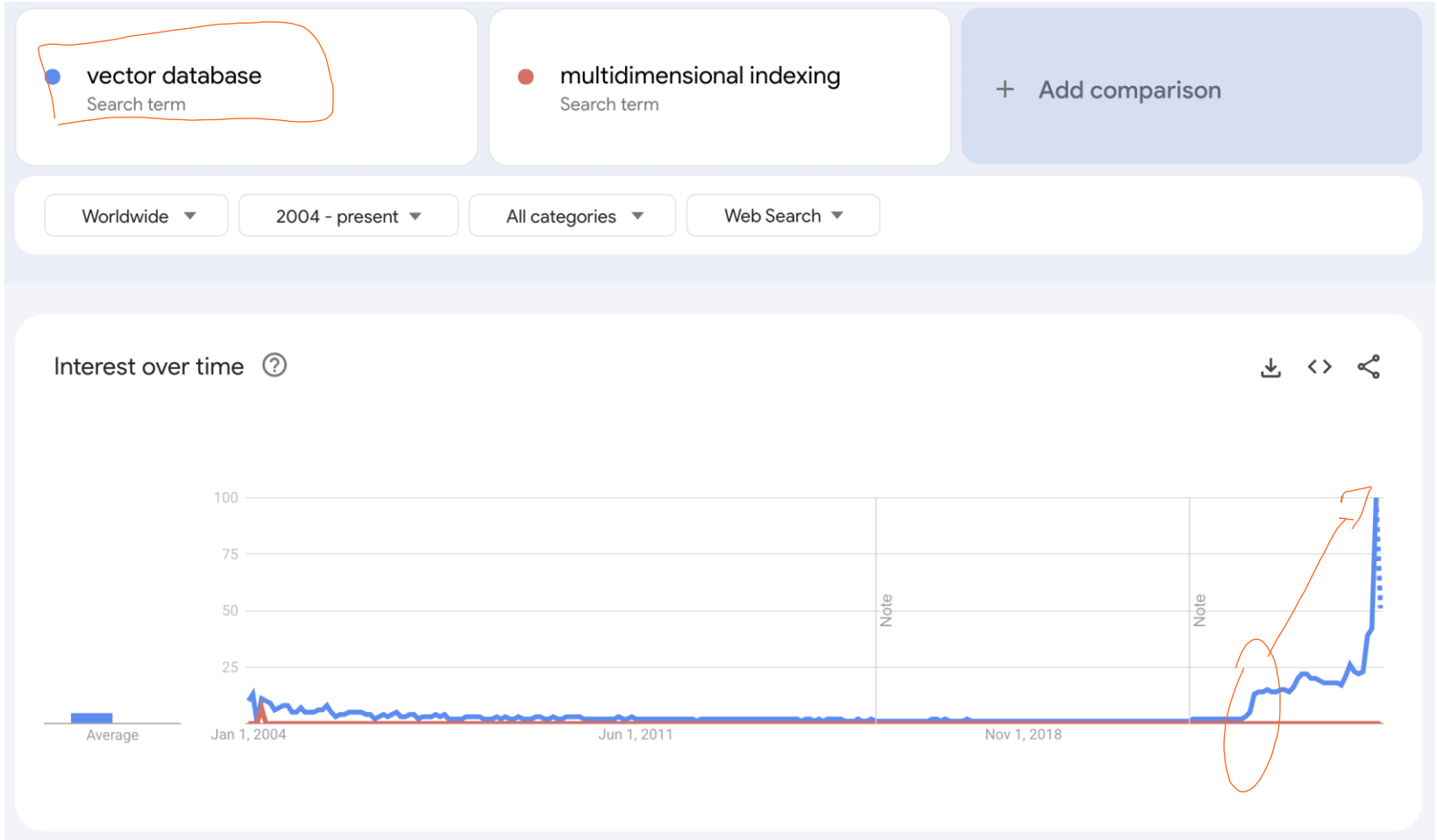
Same story in Deep Learning & LLMS

- Embeddings search.





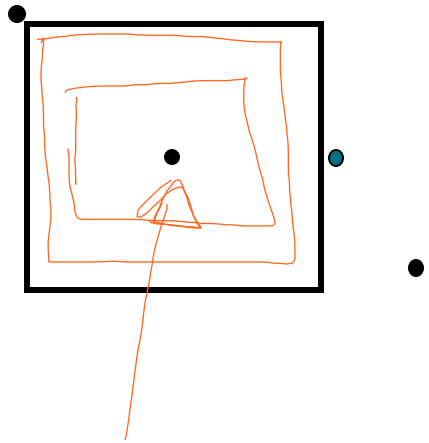
Marketed as “vector databases”



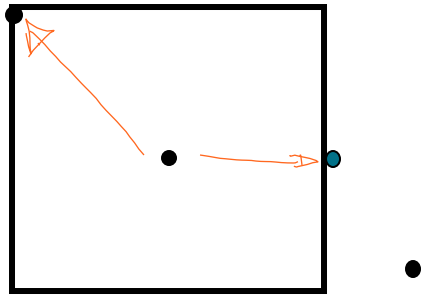
In reality: good old multidimensional indexing.

Difference nowadays: approximate indexing also ok.

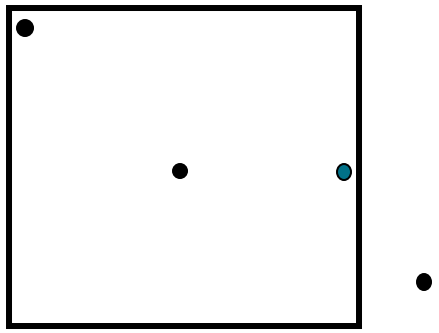
Nearest Neighbor Queries with Regular Indexes



Nearest Neighbor Queries with Regular Indexes



Nearest Neighbor Queries with Regular Indexes

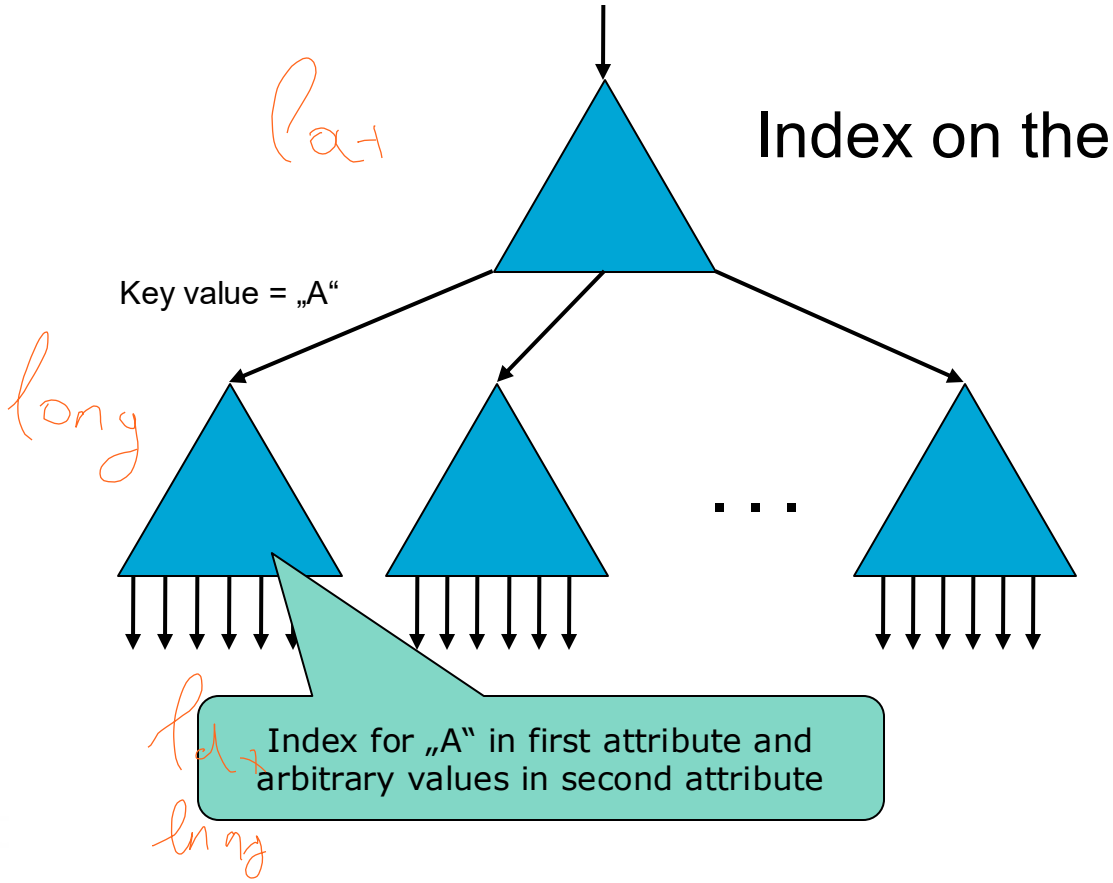


Multidimensional Index Structures

- For point sets
  - kd-Trees: Inner Nodes separate point set
  - Covered in this lecture
- Additional for sets of *regions or areas*
  - R-Trees: Separation with Overlaps –
  - Not covered in this lecture
- HNSW (Hierarchical Navigable Small World)
  - Used for high-dimensions like embeddings
  - Not covered in this lecture

Multiple-Key Indexes

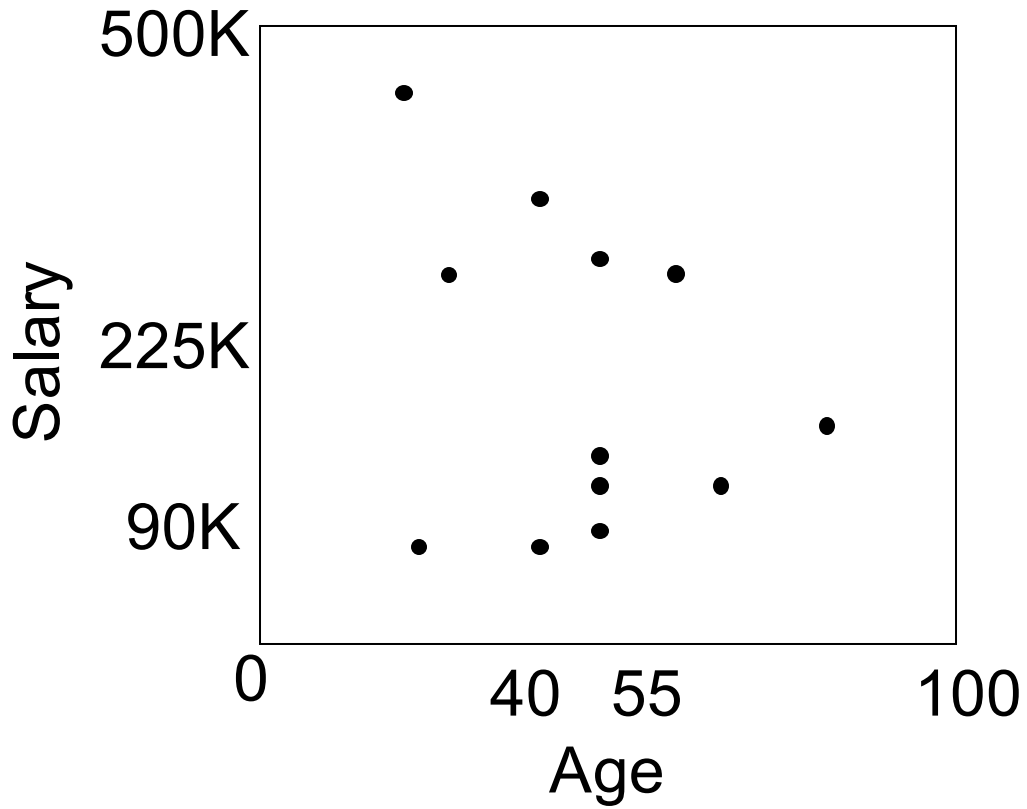
- Idea: Index on Indexes
  - Nodes on every level are Indexes on single attributes.

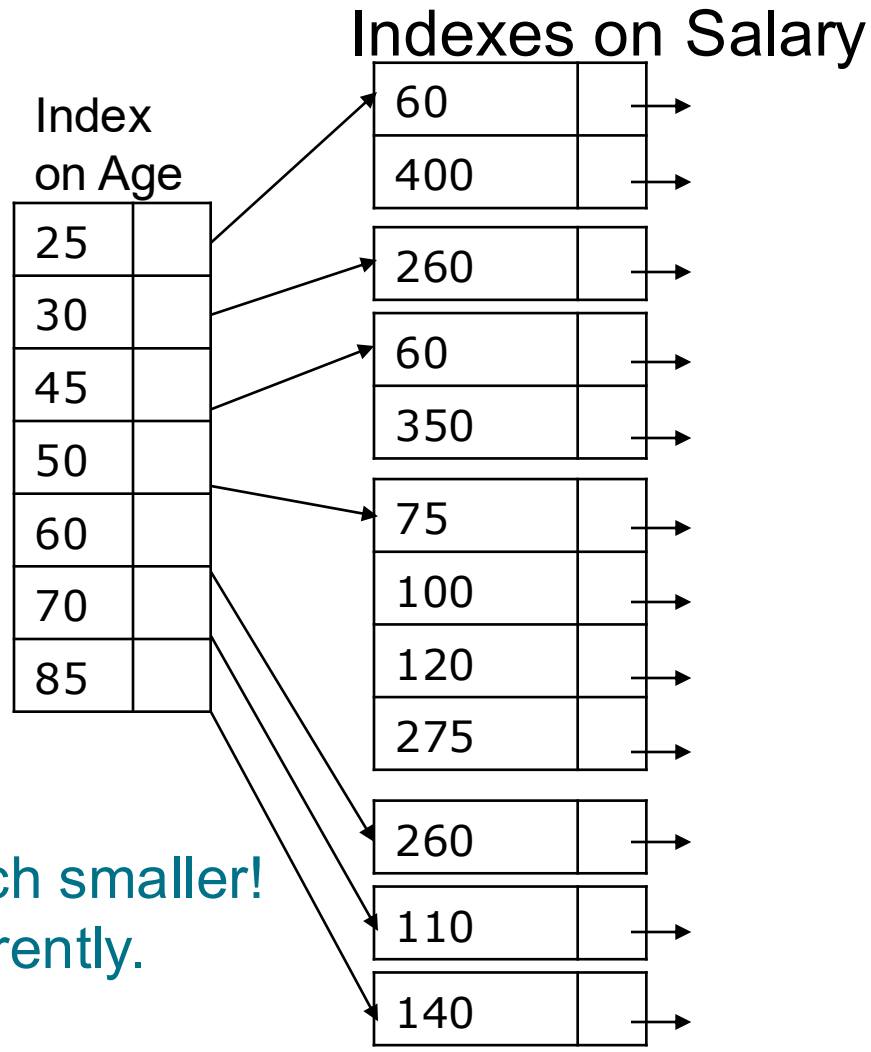


Index on the first attribute (B-tree, Hash table, …)

Indexes on second attribute (Subindexes)the

Multiple-Key Indexes - Example





- Consider:
  - Second-level indexes are usually much smaller!
  - Very small indexes implemented differently.
  - Index structure is irrelevant.

Multiple-Key Indexes – Performance

- Partial match queries (only one attribute specified)
  - If the first attribute
    - Great: Search first B-tree, then scan the entire qualifying second index
  - If other attribute
    - Bad luck: Search all sub Indexes

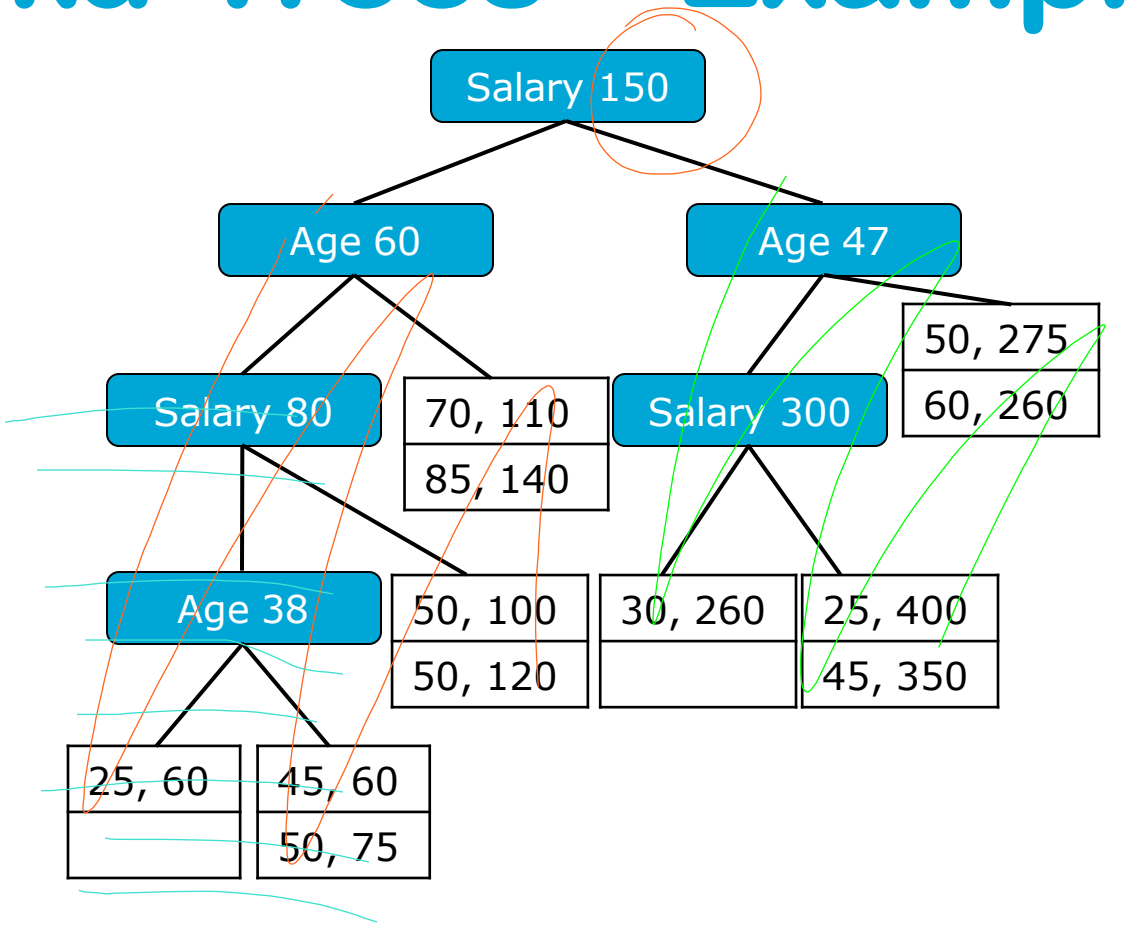
- Range queries
  - Work well, if indexes support range queries well (i.e., B-tree instead of a hash index).
  - Example: $35 \leq \texttt{Age} \leq 55$ **AND** $100 \leq \texttt{Salary} \leq 200$

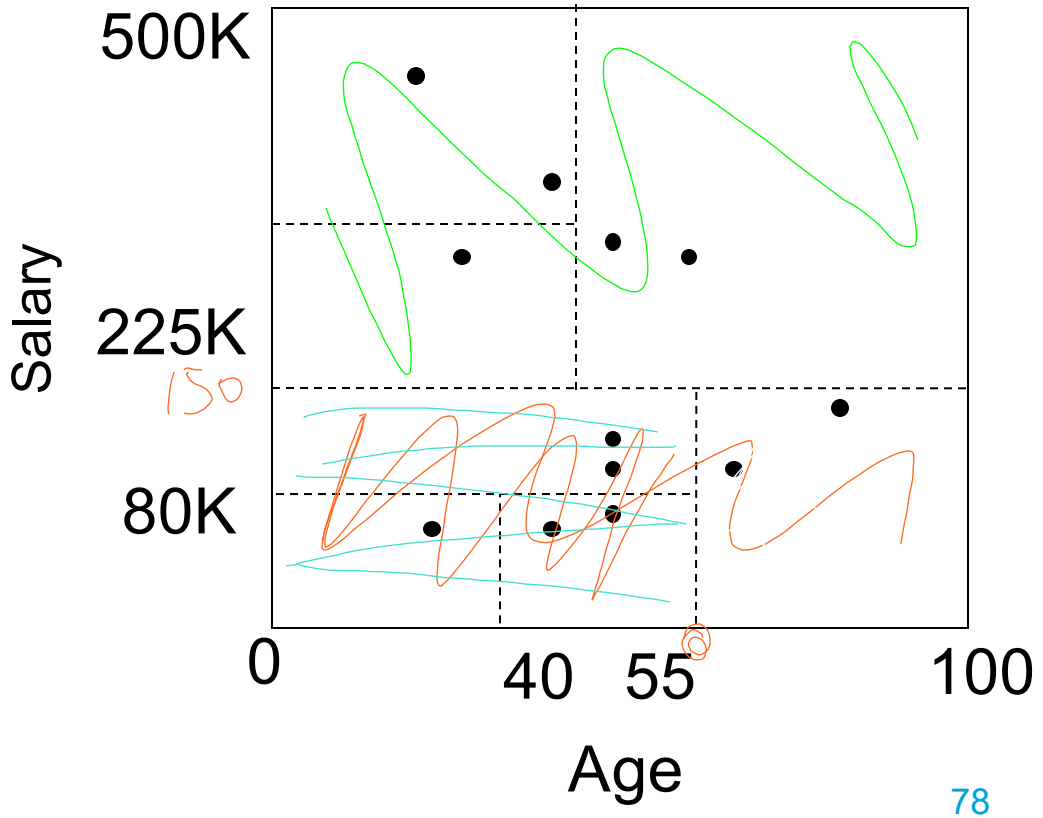
- Nearest-Neighbor-Queries (from point (x_0, y_0))
  - As before: select *d*
  - Range query: $x_0-d \leq x \leq x_0+d$ **AND** $y_0-d \leq x \leq y_0+d$
  - Potentially enlarge *d*. When?
    - If no result or nearest point has a larger distance than *d*

kd-Trees

- *k*-dimensional-trees
- Generalization of binary search trees for more (namely *k*) dimensions
- Inner nodes
  - Attribute and separator that partitions values for the attribute
    - Left: smaller values
    - Right : larger values
  - Attributes alternate through levels (every level is different attribute)
  - Two pointers to children (left and right child)
- Leaf nodes
  - Disk blocks with (many) records

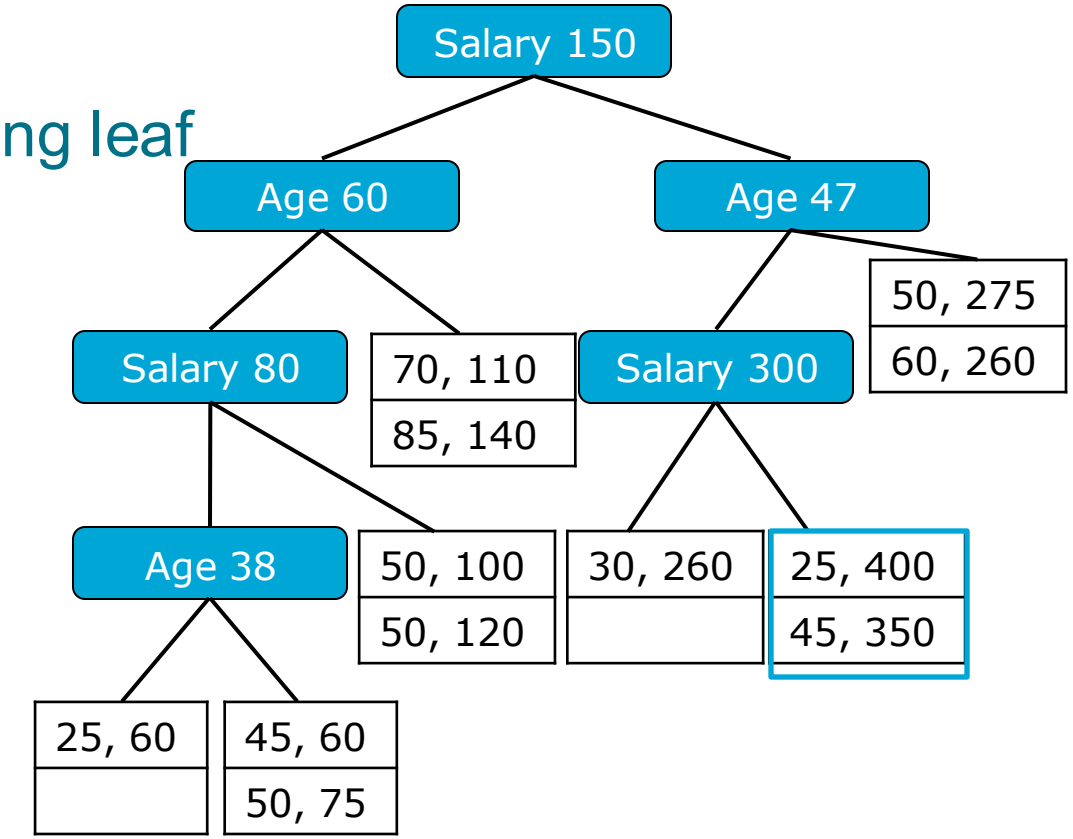
kd-Trees – Example (2d-Tree)





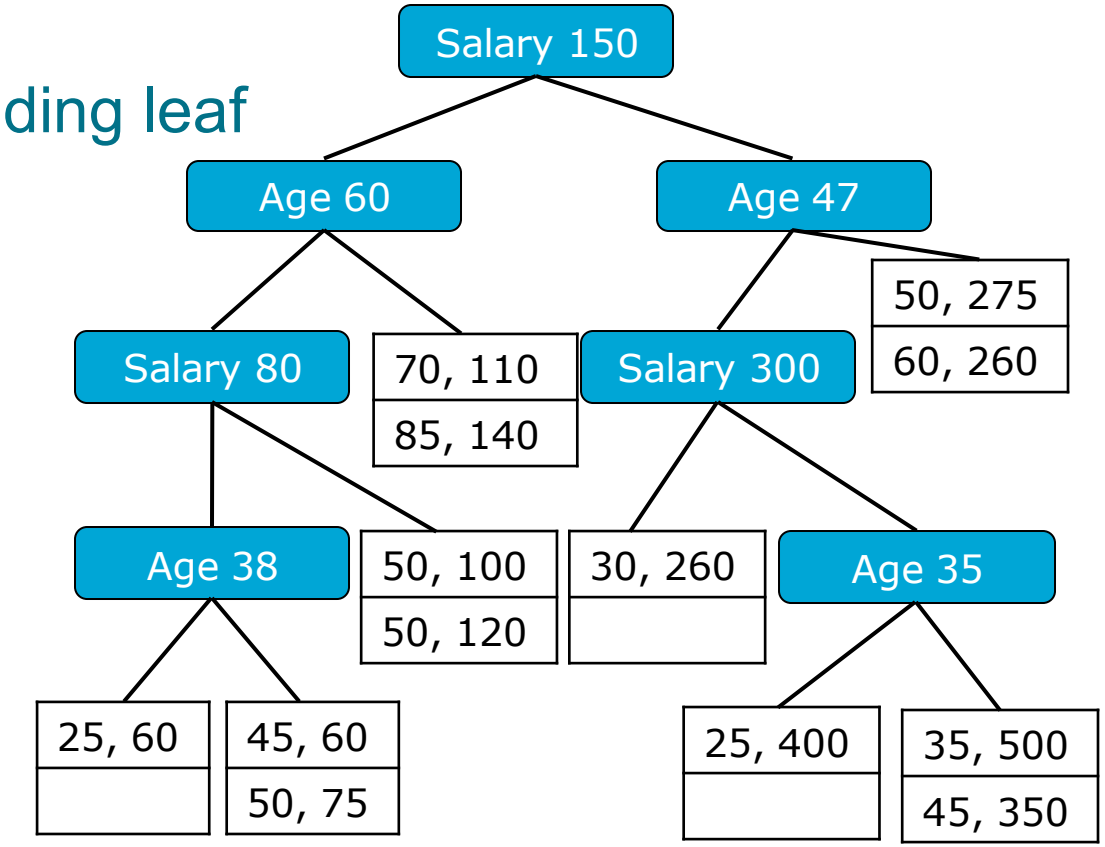
Operations on kd-Trees

- Search like in binary search trees
- Insert
  - Search corresponding leaf
  - If space: Insert
  - If no space: Split
- Example:
  - Insert (35, 500)
  - Split necessary
  - Split criteria?



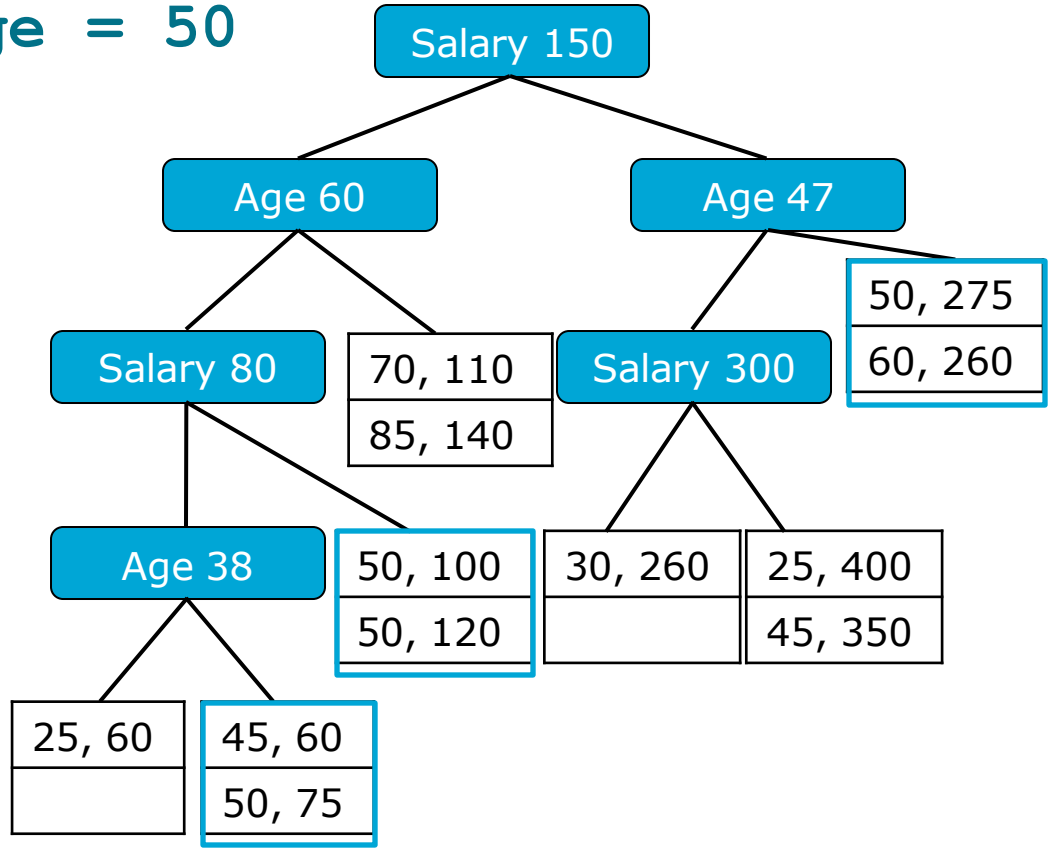
Operations on kd-Trees

- Search like in binary search trees
- Insert
  - Search the corresponding leaf
  - If space: Insert
  - If no space: Split
- Example:
  - Insert (35, 500)



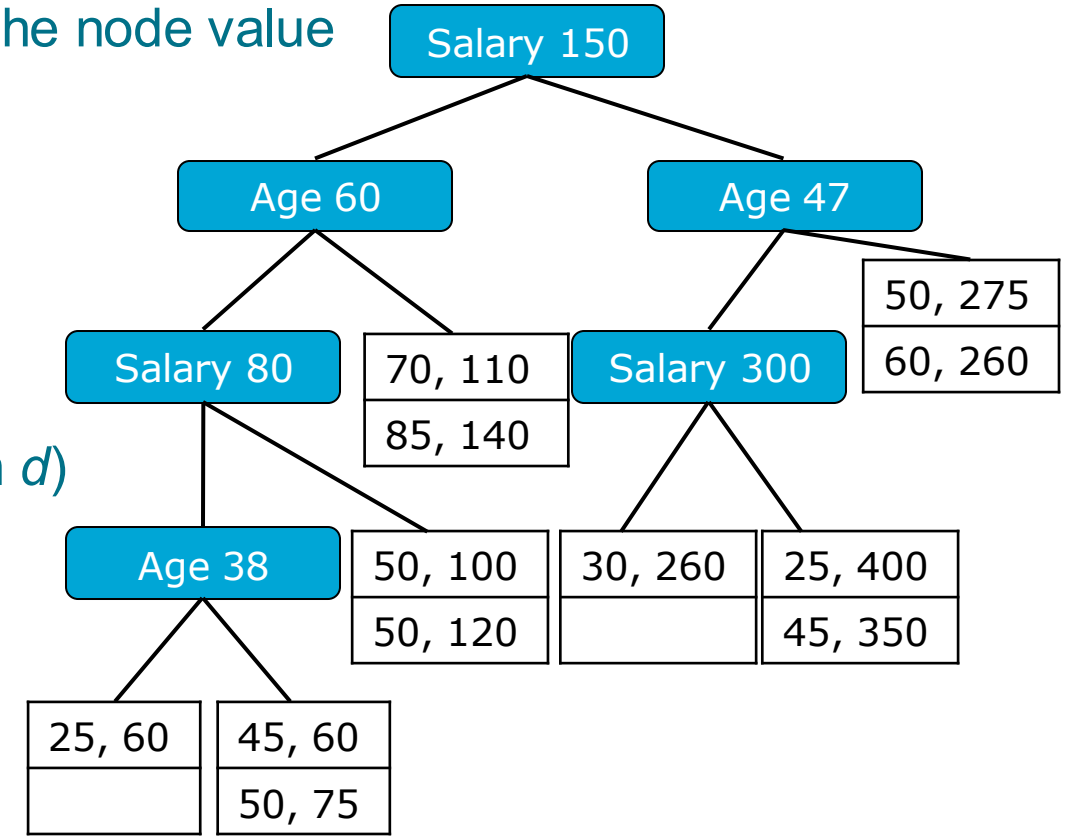
Queries with kd-Trees

- Partial match query
  - Descend to both or only one child
  - Example: **`Age = 50`**



Queries with kd-Trees

- Range queries
  - Good luck: Complete range on one side of the node value
  - Bad luck: The range covers the node value
  - `35 ≤ Age ≤ 55`
    `AND`
    `100 ≤ Salary ≤ 200`
- Nearest Neighbor Query
  - Same as a range query (with *d*)



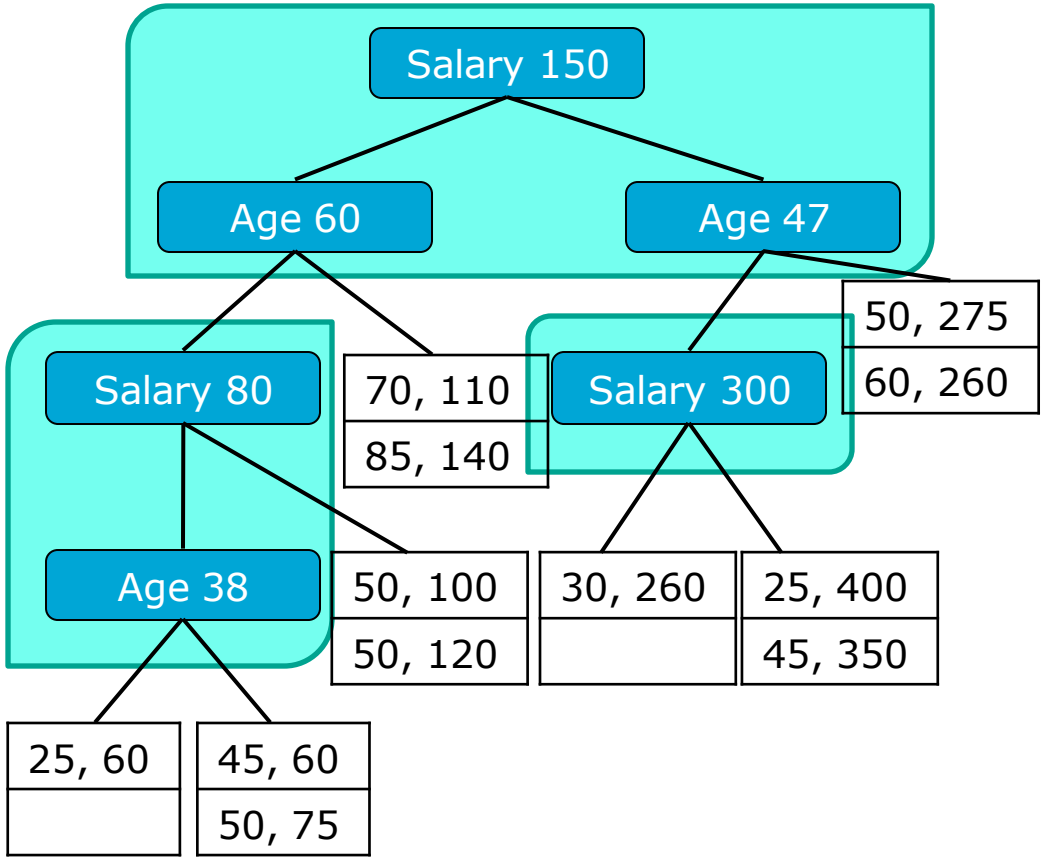
kd-Trees on Disk

- Problem 1: Path length to leaf much longer than in B-trees
  - Binary vs. n-ary
- Problem 2: Inner nodes of kd-tree are tiny
  - Only one (attribute, value) pair

- Idea 1: n-ary inner nodes
  - Each inner node stores n values and partitions value range into n+1 ranges
  - Problems: Insertion, particularly split
    - Easier in B-trees, since every level is partitioned according to the same criterion
- Idea 2: Several inner nodes in one disk block
  - One node and all descendants up to a certain level

kd-Trees on Disk

- Assumption: 3 inner nodes per block
  - 3 disk blocks



Summary

Index Structures

- A good index structure should support queries as efficiently as possible
  - Avoid block access to secondary storage wherever possible!!
  - Query types:
    - Single Equality Condition "… WHERE x = 5"
      - Does such a record exist?
      - If it exists, where can we find it?
      - Hash Indexes and B^+-Trees do well
    - Range Conditions "… WHERE x > 5 AND x < 42"
      - Traverse index efficiently
        - » Maybe it is even better to not use the index at all? → Next week!
      - + Trees do well

Index Structures

- Queries involving multiple attributes?
  - "… WHERE x = 5 AND y = 42"
  - "… WHERE x > 5 AND x < 999 AND y > 4 AND y < 1234"
  - We also have multi-dimensional trees (look them up)!
    - K-D-Tree*!*
    - R-Tree!
    - …

Hashing vs. B$^+$-Tree

- Hashing efficient to determine the existence of a value
- B$^+$-tree efficient for range queries

- Creating an index in SQL:
  - `CREATE [ UNIQUE ] INDEX indexname ON table ( column [ ASC | DESC ] [ , column [ ASC | DESC ] ... ]`
    - UNIQUE does not permit NULL
  - No specification of the type of key
  - No specification of parameters
  - Sometimes vendor-specific syntax for parameters
  - Sometimes one can specify type: hash vs. B$^+$-tree

Friday

- Parallel Database & Matrix Operations



Fin



TU Delft WIS Web Information Systems