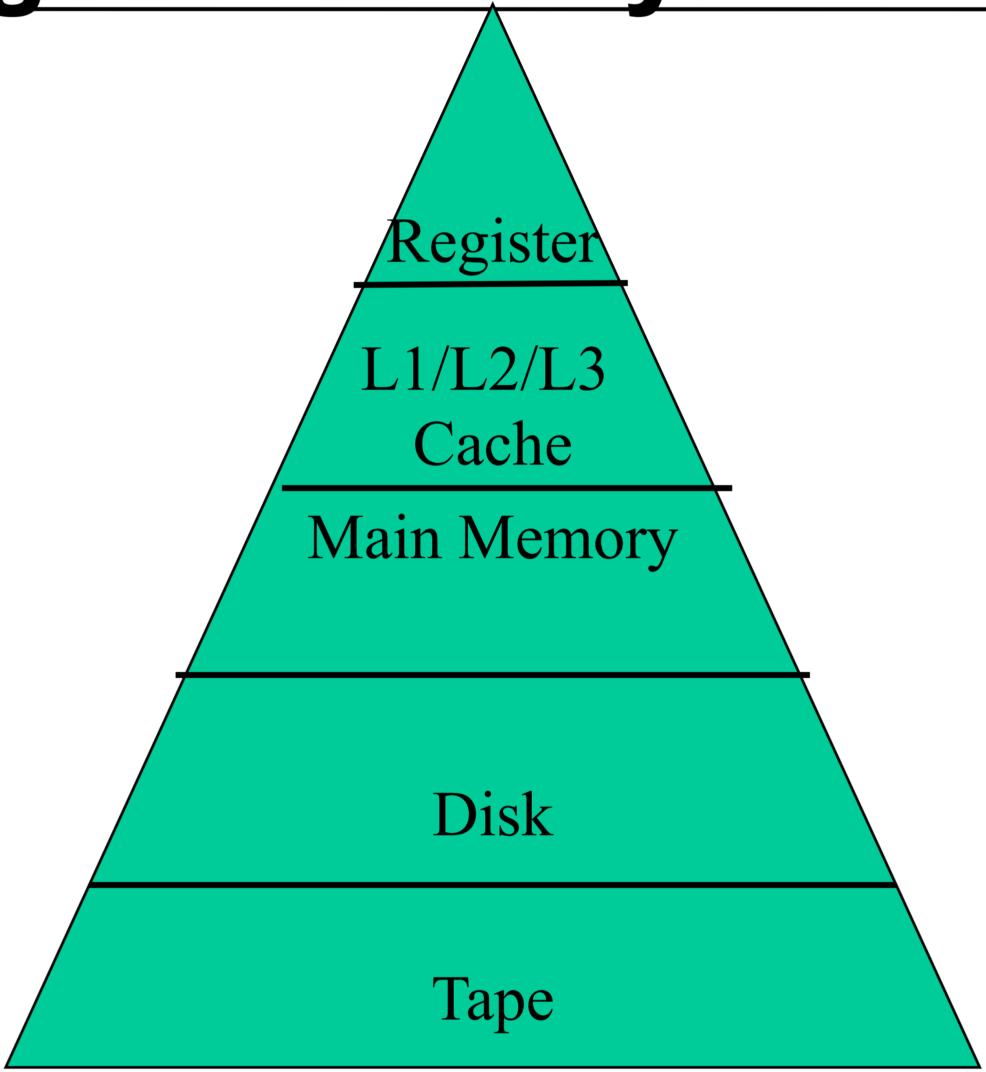
# Physical Data Organisation

Topics:

- Storage hierarchy
- Storage structures
- ISAM
- B-Trees
- Hashing
- Clustering

# Storage Hierarchy



Register

L1/L2/L3
Cache

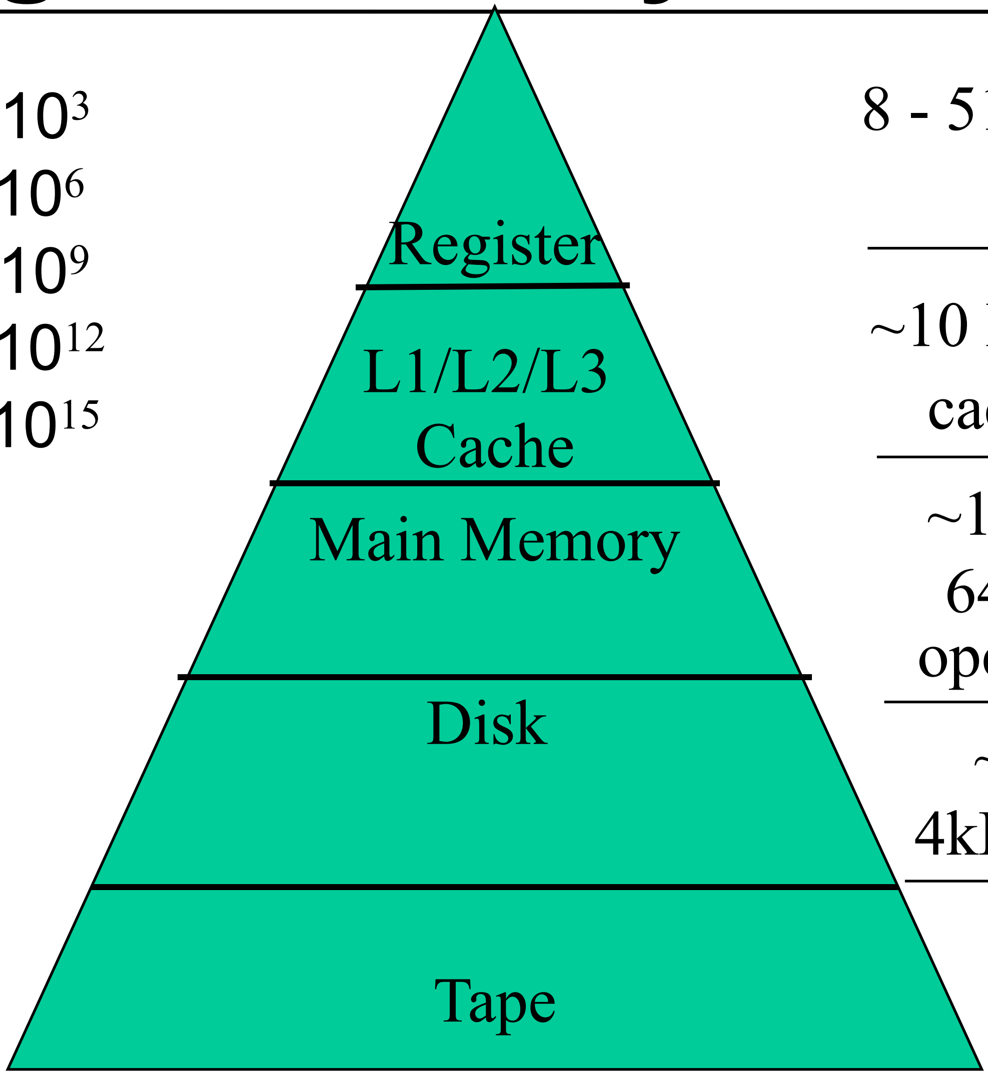
Main Memory

Disk

Tape

# Storage Hierarchy

1 K (Kilo) = $10^3$
1 M (Mega) = $10^6$
1 G (Giga) = $10^9$
1 T (Tera) = $10^{12}$
1 P (Peta) = $10^{15}$



8 - 512 Byte/Register
Compiler

~10 MB Byte/Cache
cache-controller

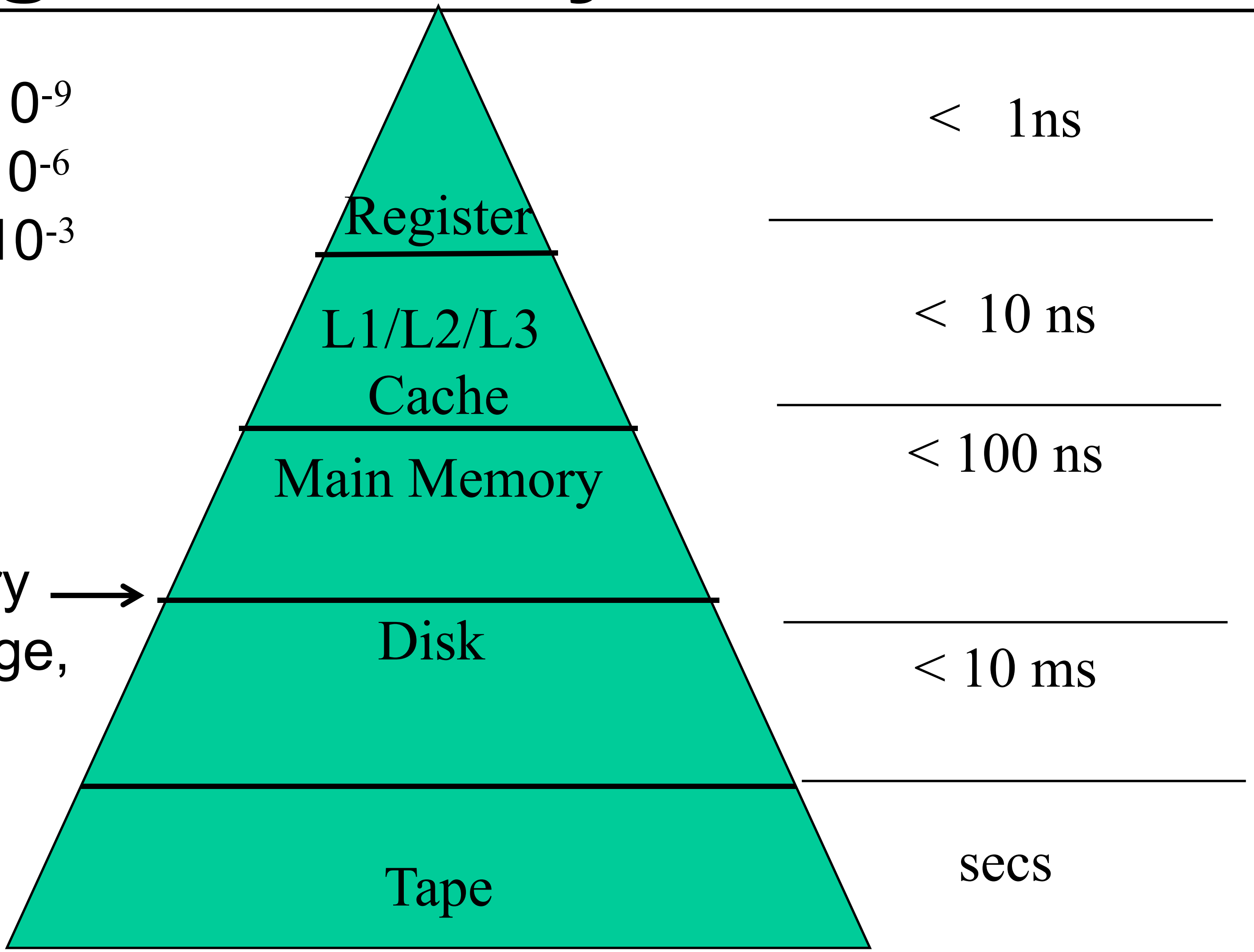
~100 GB-range,
64B block size
operating system

~1 TB-range
4kB blocks | user

PB-range
user

# Storage Hierarchy

1 n (nano) = $10^{-9}$
1 μ (micro) = $10^{-6}$
1 m (milli) = $10^{-3}$



(Flash-Memory →
Lower TB-range,
< 100 μs)

# Storage Hierarchy

Room (1min)

Building (10min)

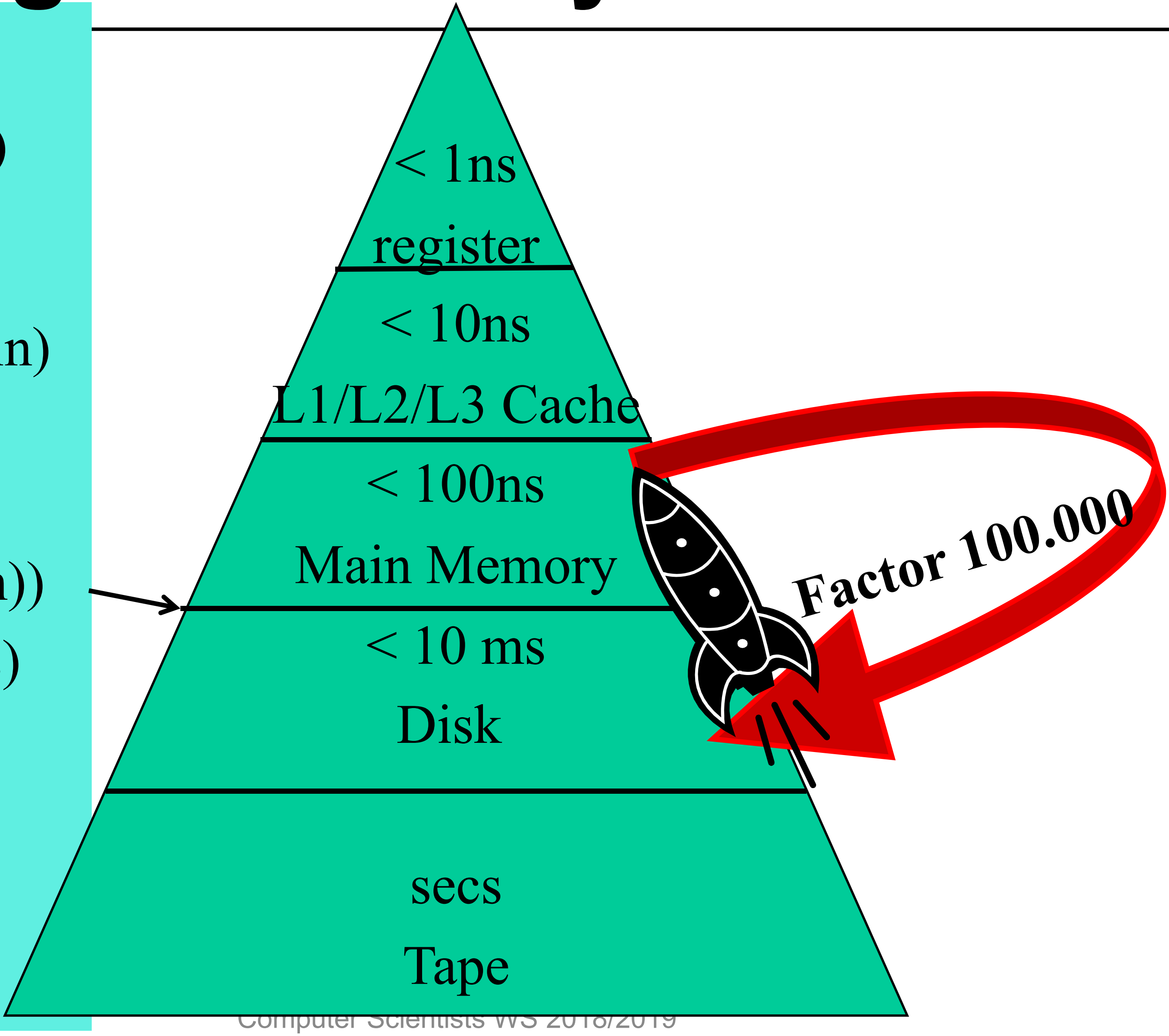
City (1.5h)

(Mars (2 month))

Pluto (9 years)

Andromeda

(2000 years)



< 1ns

register

< 10ns

L1/L2/L3 Cache

< 100ns

Main Memory

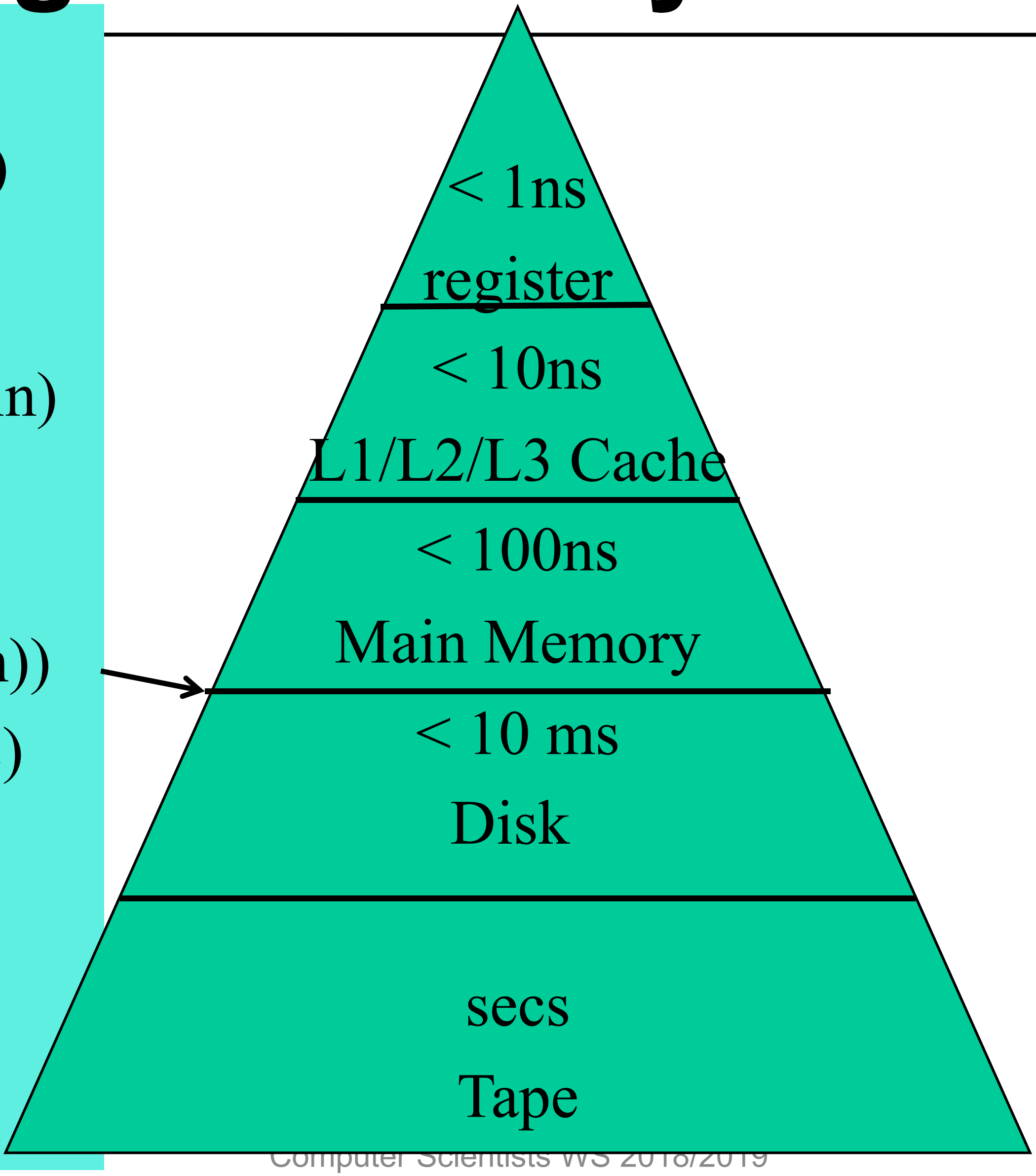
< 10 ms

Disk

secs

Tape

# Storage Hierarchy



Room (1min)

Building (10min)

City (1.5h)

(Mars (2 month))

Pluto (9 years)

Andromeda

(2000 years)

< 1ns

register

< 10ns

L1/L2/L3 Cache

< 100ns

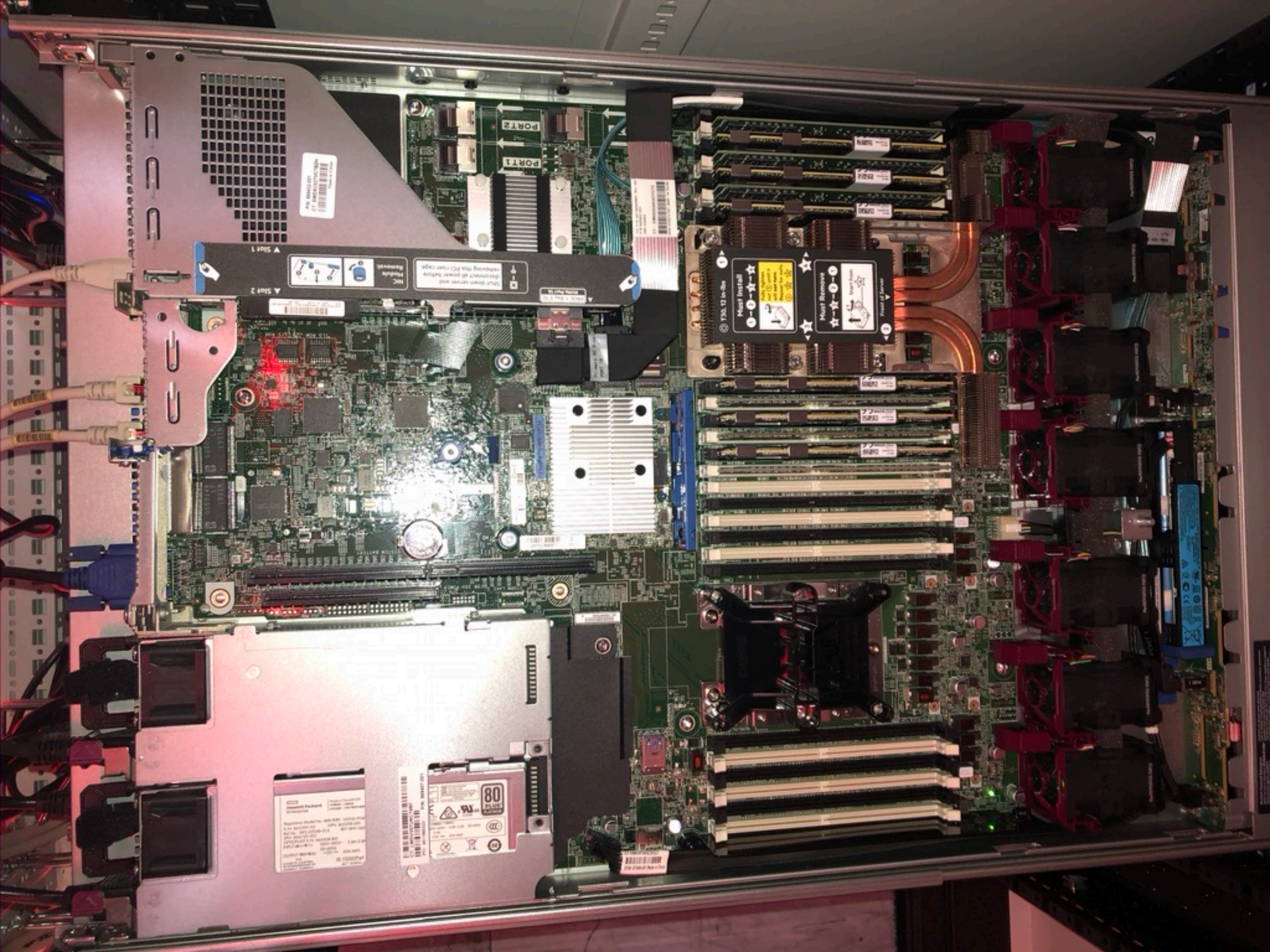
Main Memory

< 10 ms

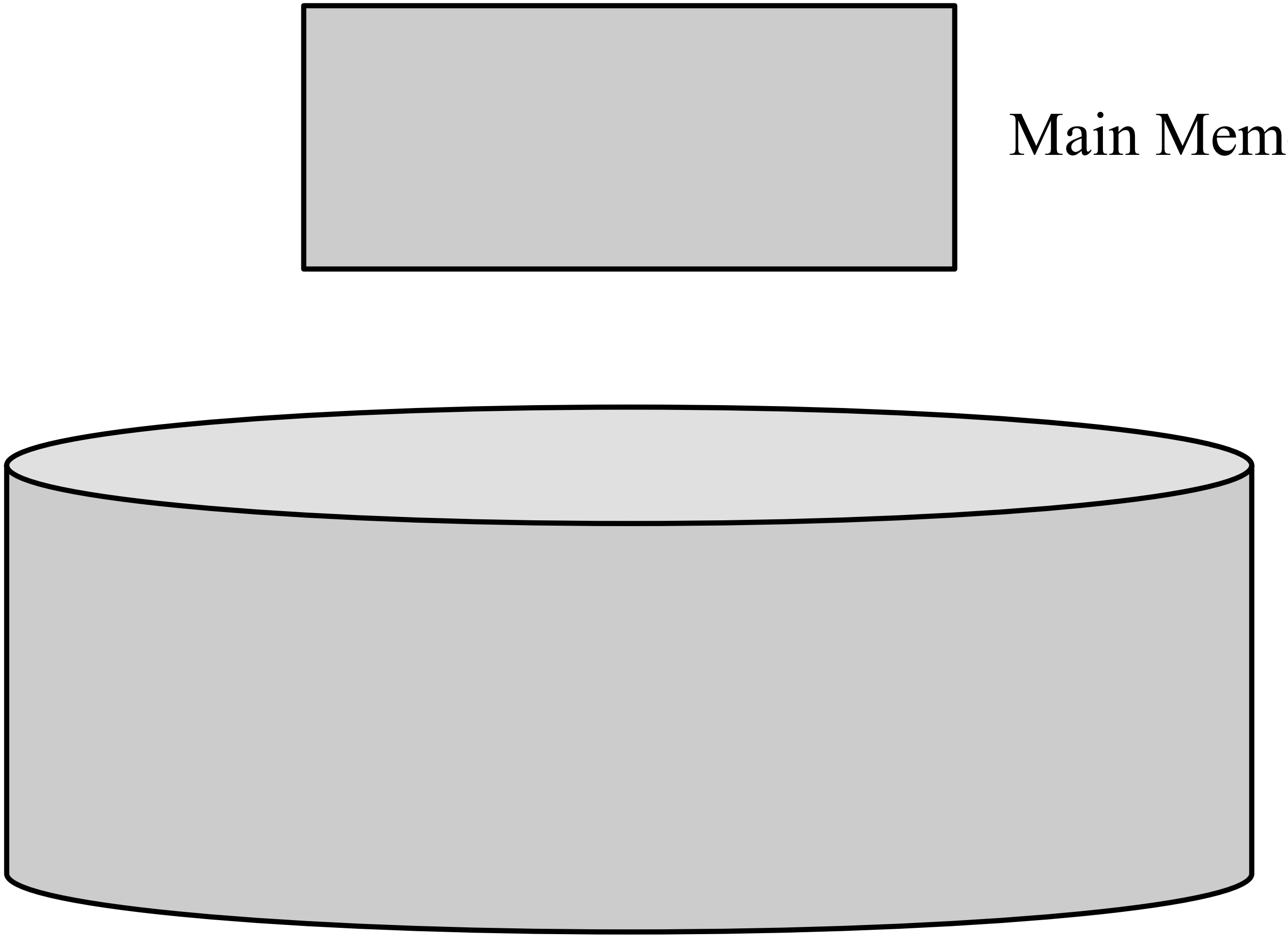
Disk

secs

Tape


PORT2
PORT1
Slot 1
Slot 2
Shut down server and disconnect all power before removing the PCI riser cage.
NIC Module Removal
Must Install
Must Remove
T30, 12 in-lbs
Front of Server
80 PLUS

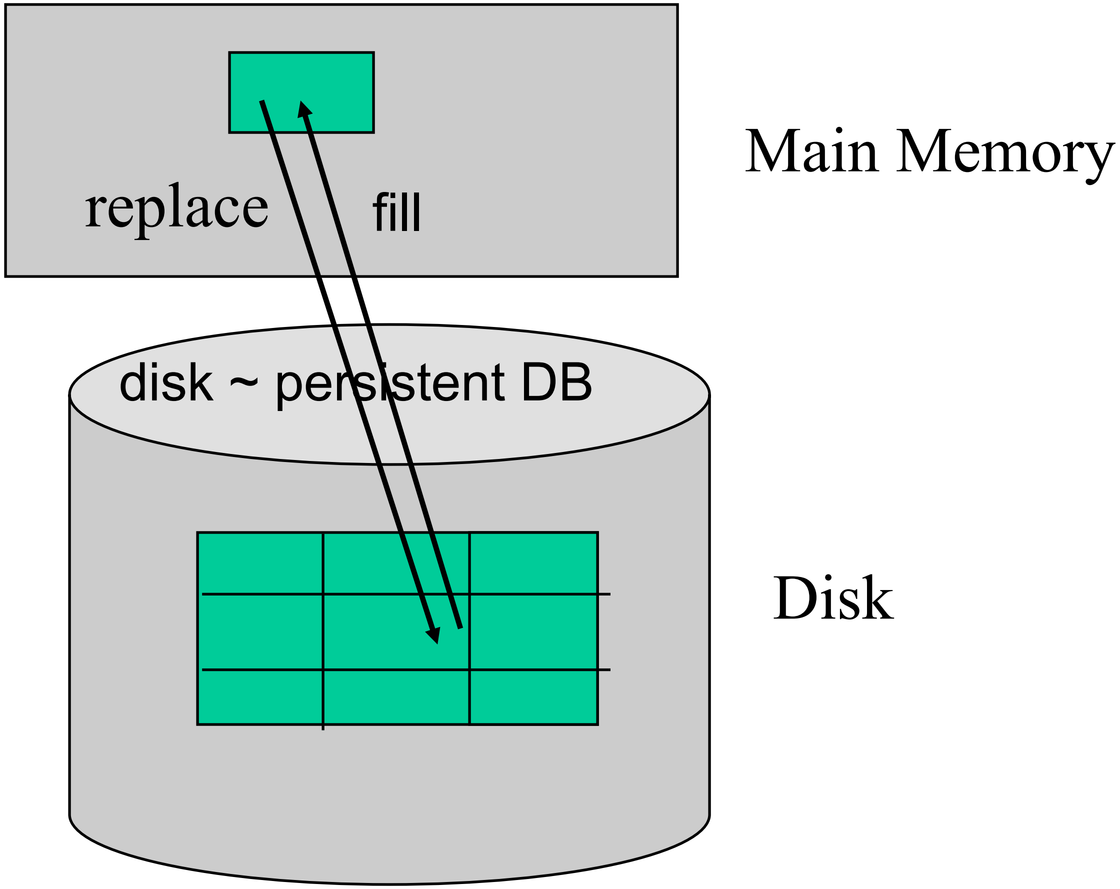
# Buffer Management



Main Memory

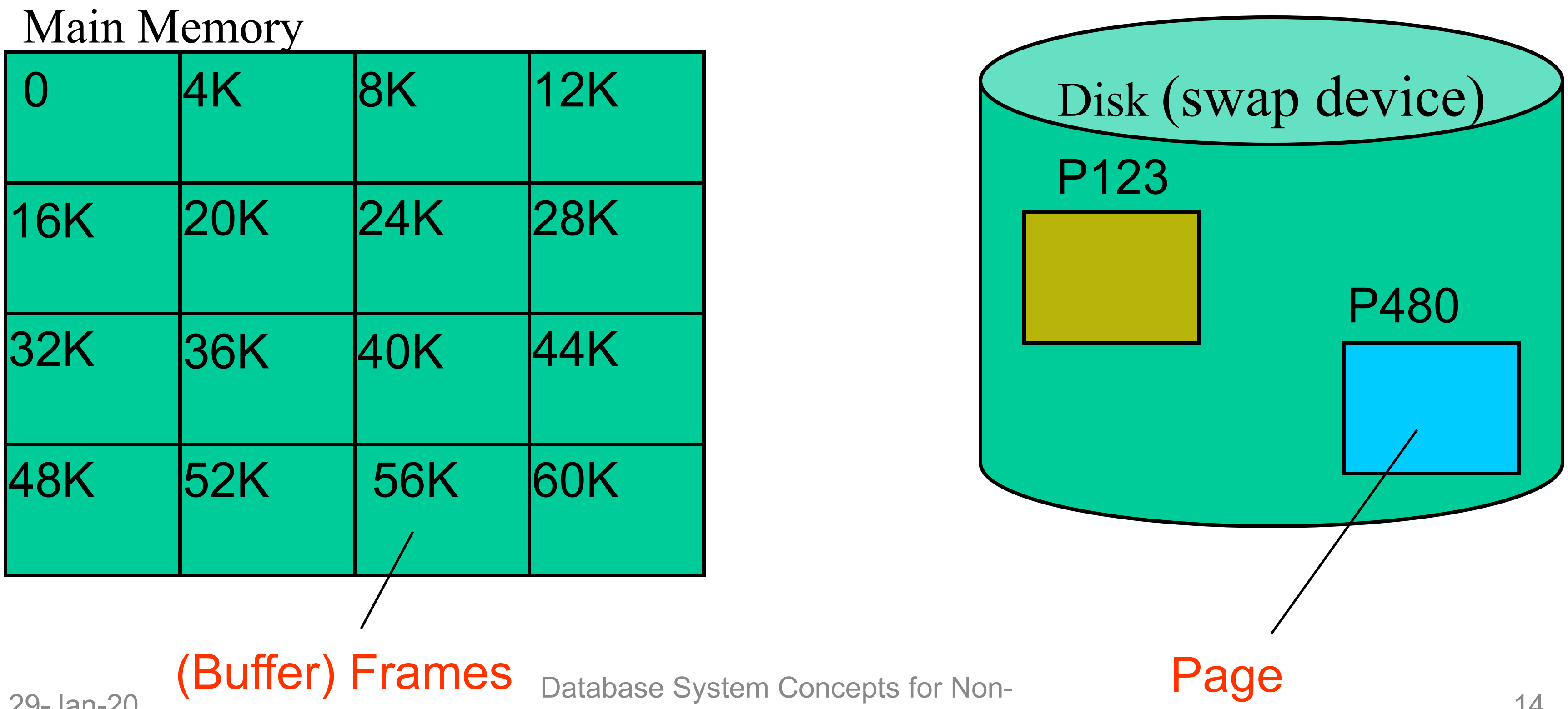
Disk

# Buffer Management

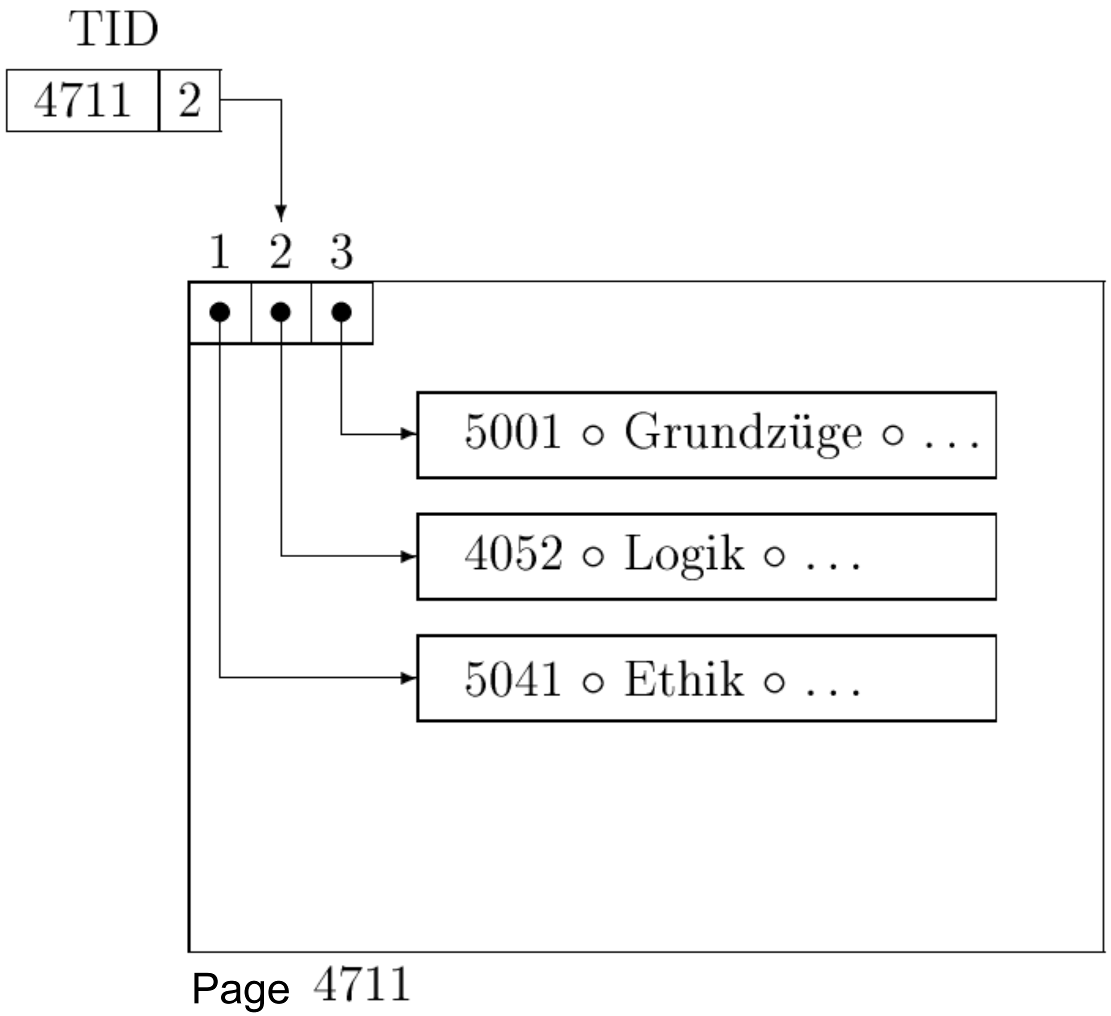


# Fill and replace pages

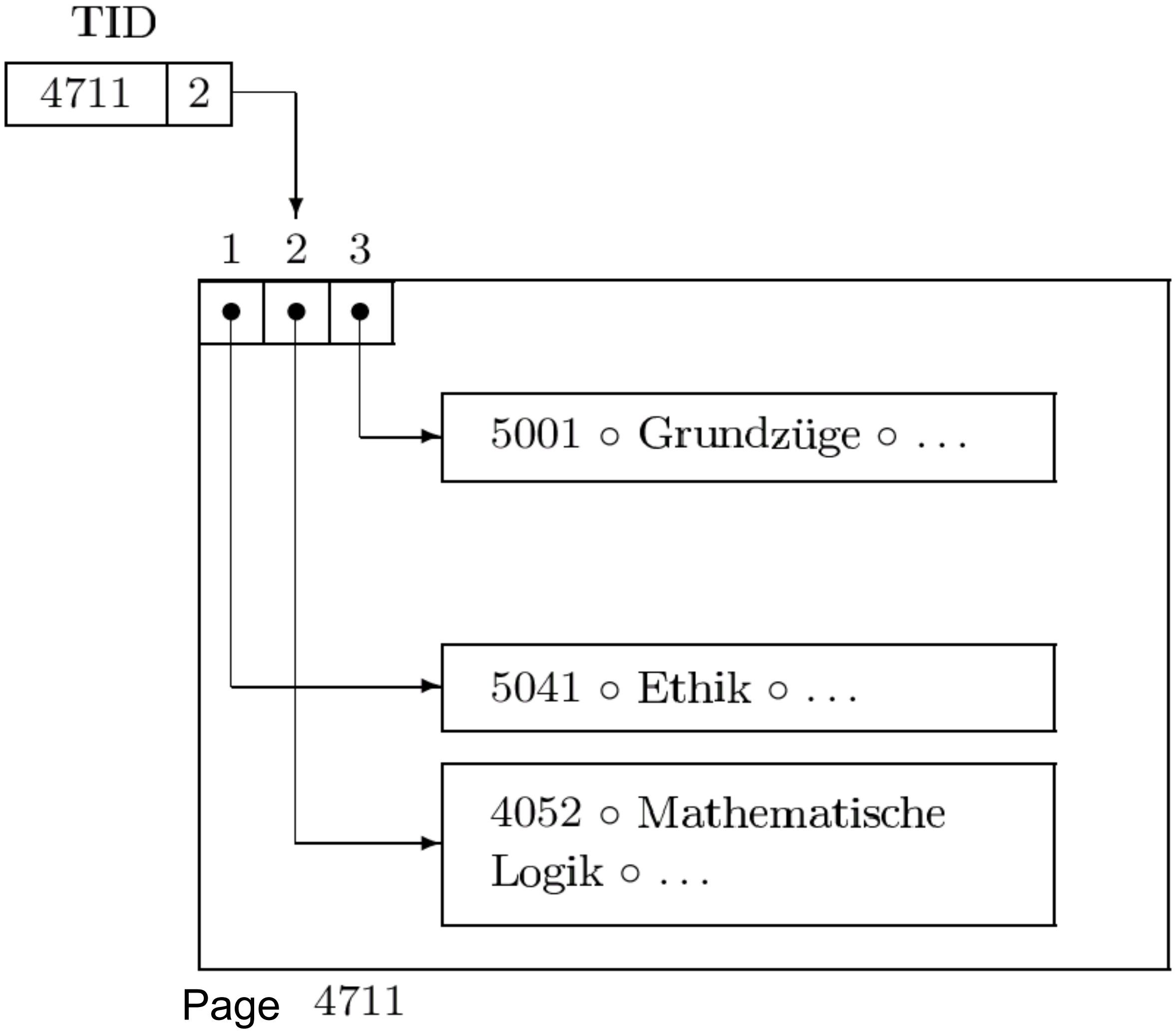
- System buffer is divided in frames of equal size
- A frame can be filled with one page
- Overflow pages are swapped on disk



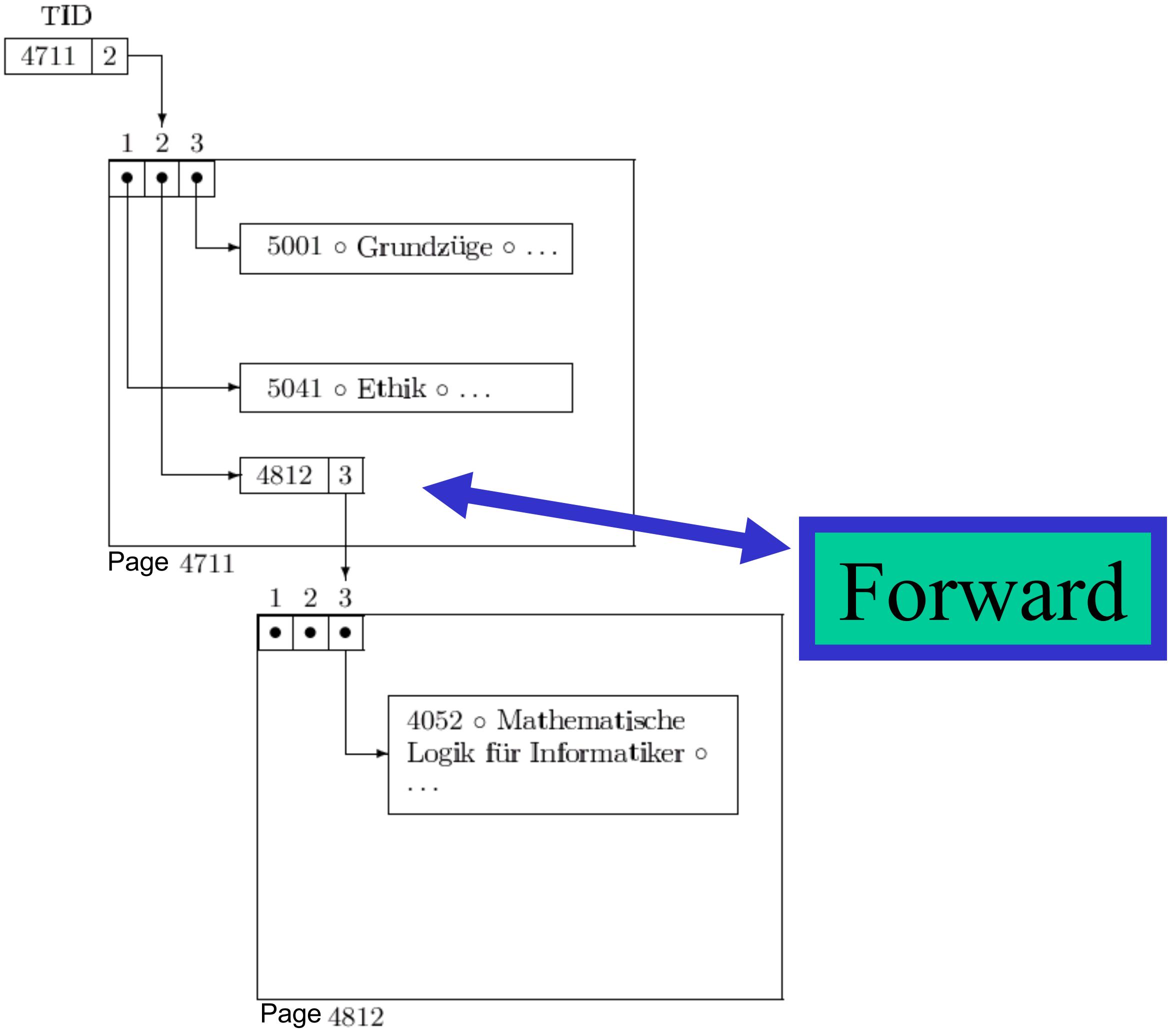
# Addressing tuples on disk



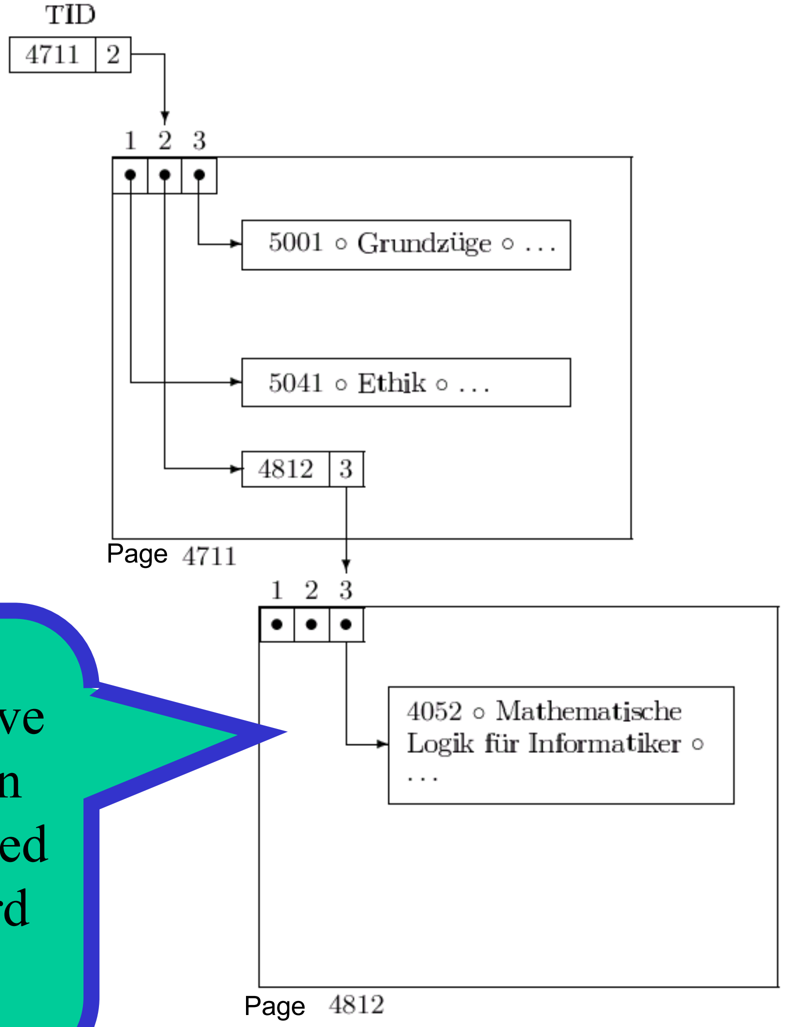
# Moving within a page



# Moving from one page to another

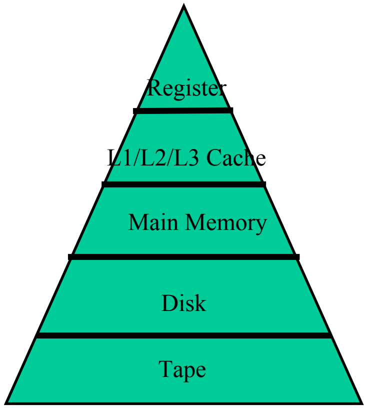


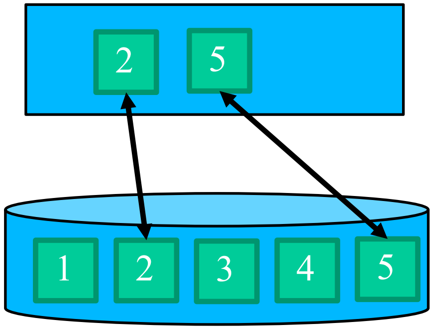
# Moving from one page to another

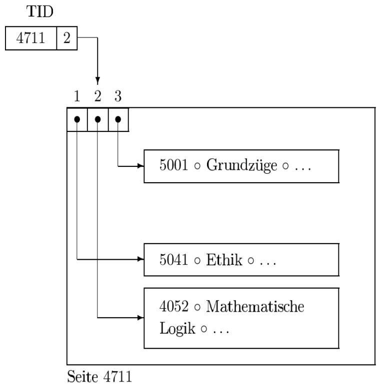


With the next move the „Forward“ on page 4711 is altered (no more Forward to page 4812)

# Storage Summary







## Storage Hierarchy

- Huge/slow storage vs small/fast memory
- Very important for DBMSs design
- Algorithms need to be aware of performance difference and place data optimally

## Buffer Management

- A component of the database
- Migrates pages between disc and main memory
- Keeps hot pages in DRAM and cold ones on disc

## Tuple IDs

- Used to locate a tuple
- Composed of page identifier and a page-local record identifier

# Storage Operations

Full table scan: Retrieve all tuples from a table

```
select * from students;
```

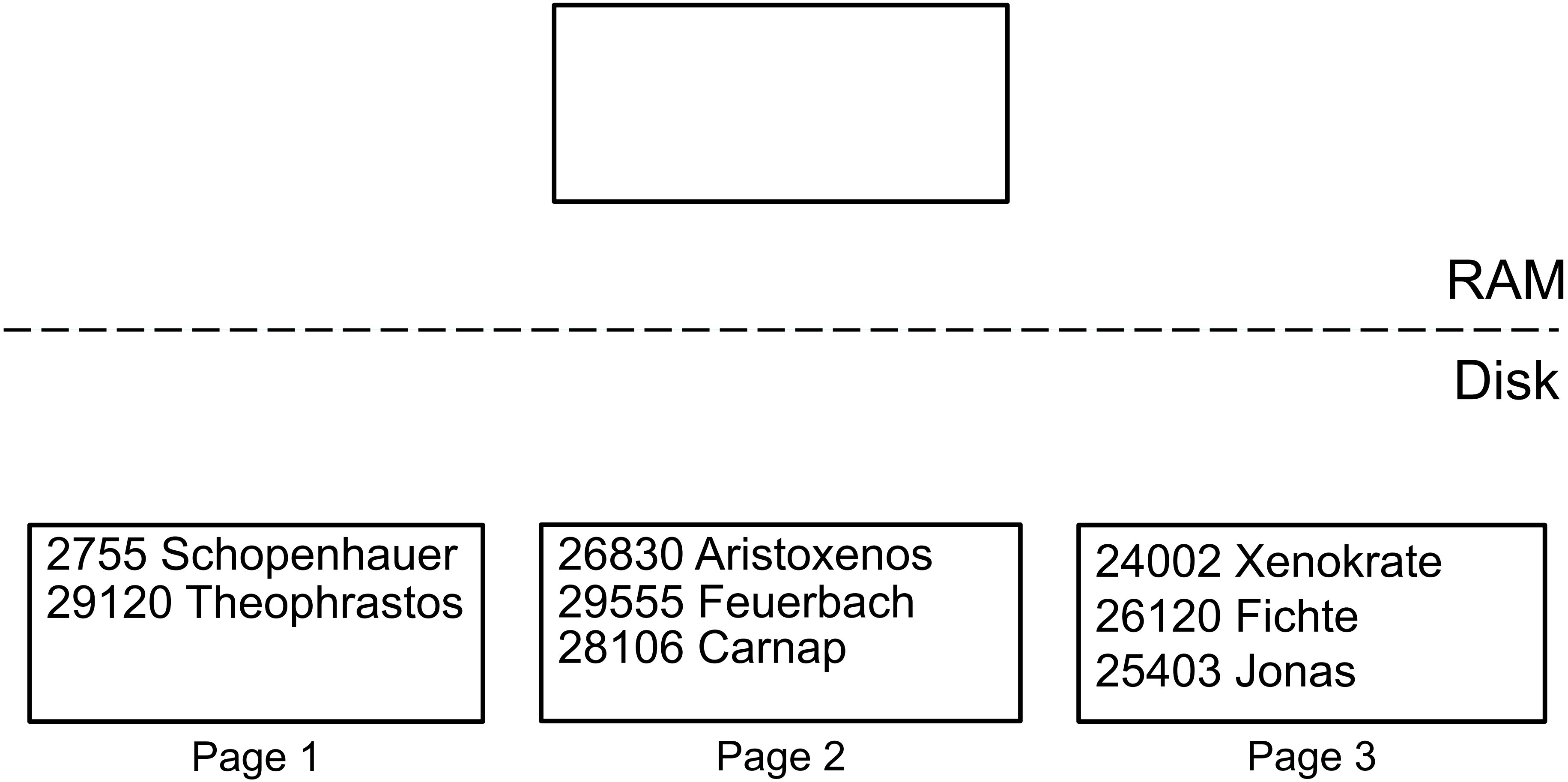
Point query: Find one specific tuple

```
select * from students
         where studNr =26120;
```

Range query: Find one specific tuple

```
select * from students
         where 26000 <= studNr
           and studNr < 27000;
```

# Index Motivation



RAM

Disk

2755 Schopenhauer
29120 Theophrastos

Page 1

26830 Aristoxenos
29555 Feuerbach
28106 Carnap

Page 2

24002 Xenokrate
26120 Fichte
25403 Jonas

Page 3

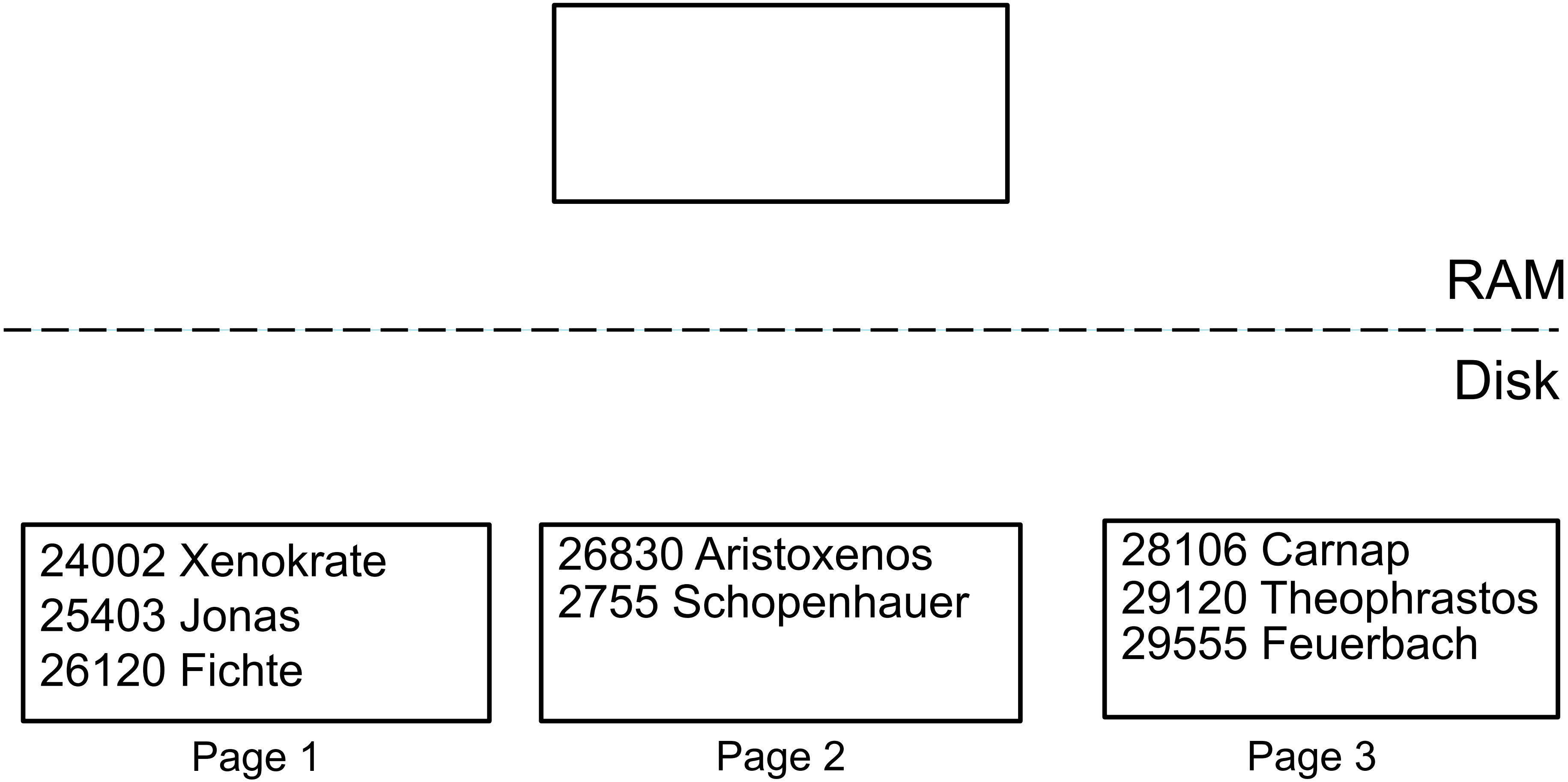
# Index Motivation

Full table scan: Load all pages from disk, one by one.

Point query: When we sort the data, we can find a key more easily. Compare to a dictionary: You can look at the page in the middle to determine in which half the word you are looking for is and then continue this process with the new found half. This way you need to look at much less pages then reading it front to back.

Range query: When the data is stored in a sorted fashion, a range query can be processed as a combination of a point query (to find the starting point) and then a scan.

# Index Motivation



RAM

Disk

24002 Xenokrate
25403 Jonas
26120 Fichte

Page 1

26830 Aristoxenos
2755 Schopenhauer

Page 2

28106 Carnap
29120 Theophrastos
29555 Feuerbach

Page 3

# Data Transfer

Simple query execution:

**select** * **from** students **where** studNr=26120;

Get one tuple/page after the other to the main memory and evaluate predicates.

→ Most expensive way ☹

→ Mostly only a small fraction of the tuples fulfills the query

# Index Structures

- Index structures are used to keep the data volume to be transferred from disk to main memory small
- Only that part of the data which is really needed to answer the query is transferred
- Two main indexing methods:
  - Hierarchical (trees)
  - Partitioning (Hashing)

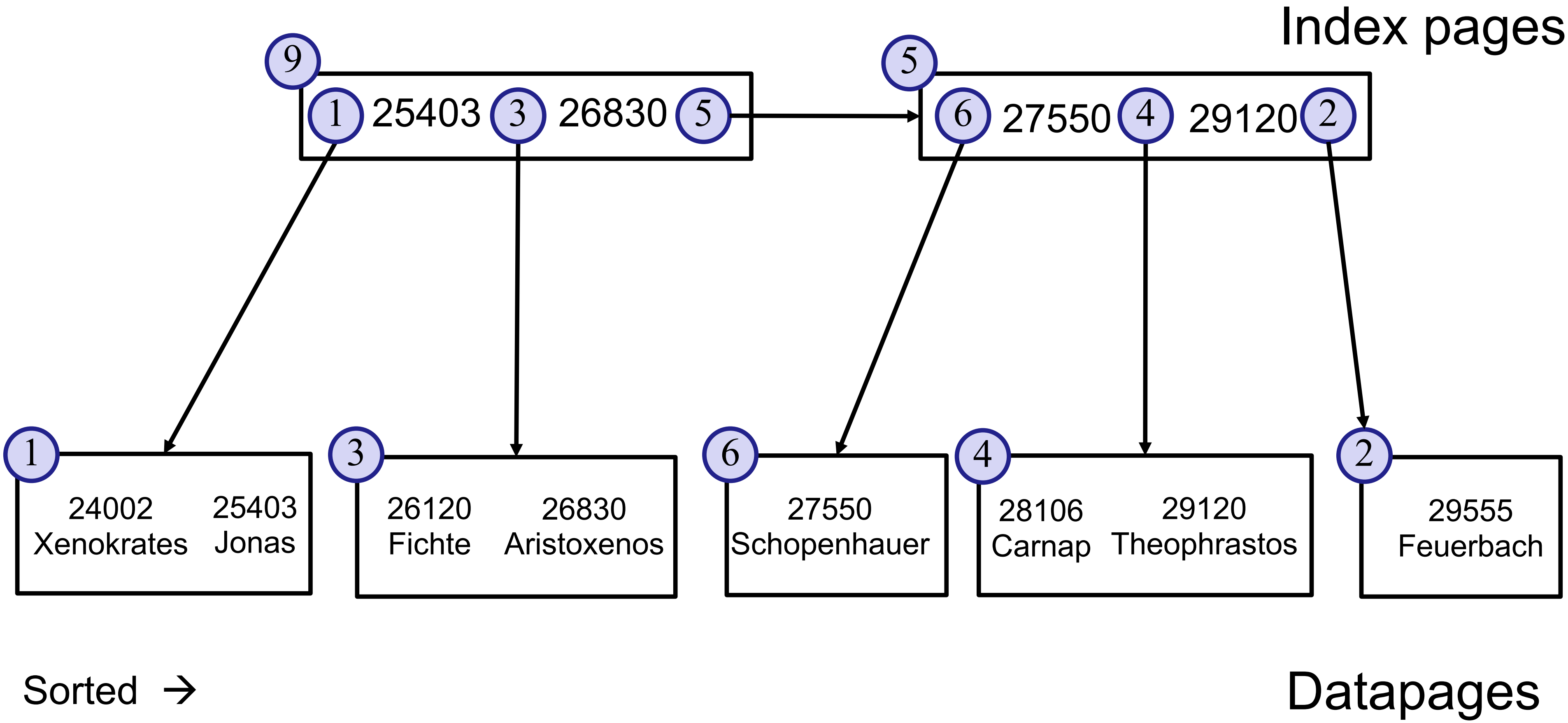
# Hierarchical Indexes

We consider two hierarchical index structures:

- ISAM (Index-Sequential Access Method)
- B-Trees

- ISAM is the predecessor of B-Trees
- Main idea: sort tuples on the indexed attribute and create an index file on it
- Similar to a thumb index in a book

# ISAM Example



Index pages

9

1 25403 3 26830 5

5

6 27550 4 29120 2

1

24002 Xenokrates

25403 Jonas

3

26120 Fichte

26830 Aristoxenos

6

27550 Schopenhauer

4

28106 Carnap

29120 Theophrastos

2

29555 Feuerbach

Sorted →

Datapages

# Example cont.

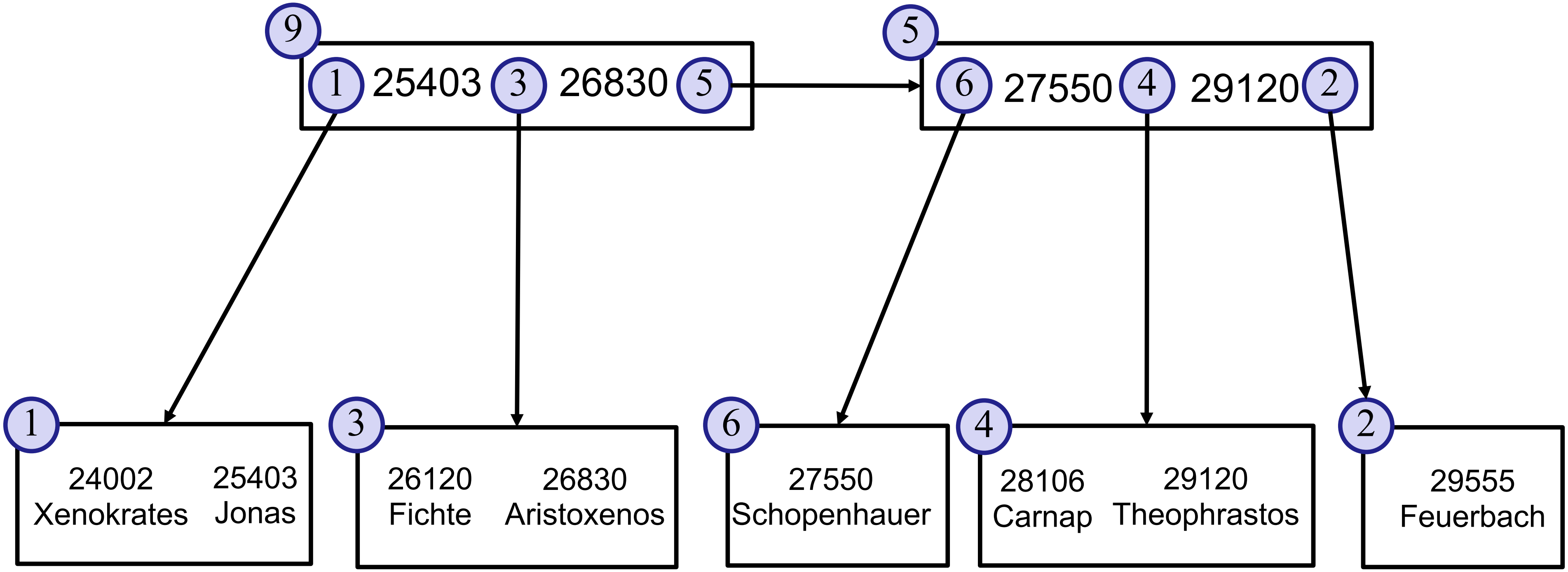
- Student with student number 28106 is searched
- Go through the **index pages** (starting at ⑨) and look where 28106 fits => the first larger (or equal) key is 29120 (on page ⑤)
- From there you get the referenced **data page** => ④
- **Advantage**: Number of index pages is **much** less than number of data pages, i.e. you save I/O
- You can also answer **range queries**, e.g. all StudNr between 26830 and 29120 : find as a start the first fitting data page for 26830 and from there on you can go **sequentially** through the data pages until StudNr 29120

# Problems with ISAM

Simple and fast search but **maintenance of index** is expensive:

- Inserting a tuple in a full data page: need to make room in **dividing data page into two** (because we need to keep the tuples sorted)
- This creates a **new entry** on an **index page**
- Inserting an entry in a full index page leads to **shifting the entries** to make room
- Although the number of index pages is smaller than the number of data pages **going through the index pages** can nevertheless **take a long time**

# Exercise 1


9
1
25403
3
26830
5
5
6
27550
4
29120
2
1
24002
Xenokrates
25403
Jonas
3
26120
Fichte
26830
Aristoxenos
6
27550
Schopenhauer
4
28106
Carnap
29120
Theophrastos
2
29555
Feuerbach

# Exercise 2

# Advancement

Idea:

Why not have **index pages for the index pages**?

→ This is in principle the idea of a **B-Tree**

# Trees

All trees in computer science

… have nodes

… have edges

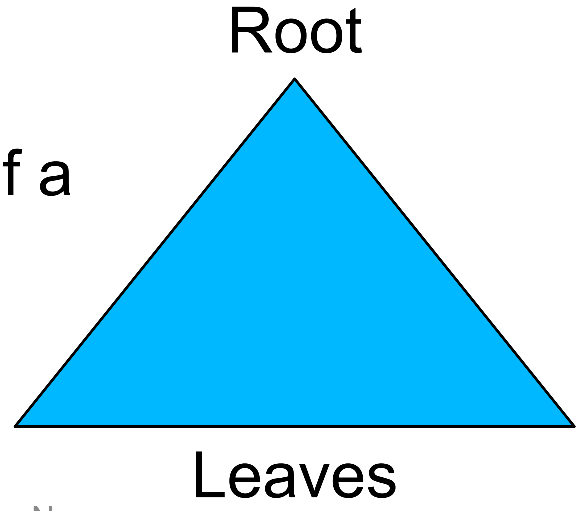
… have a root (at the top!)

… have leaves (at the bottom!)

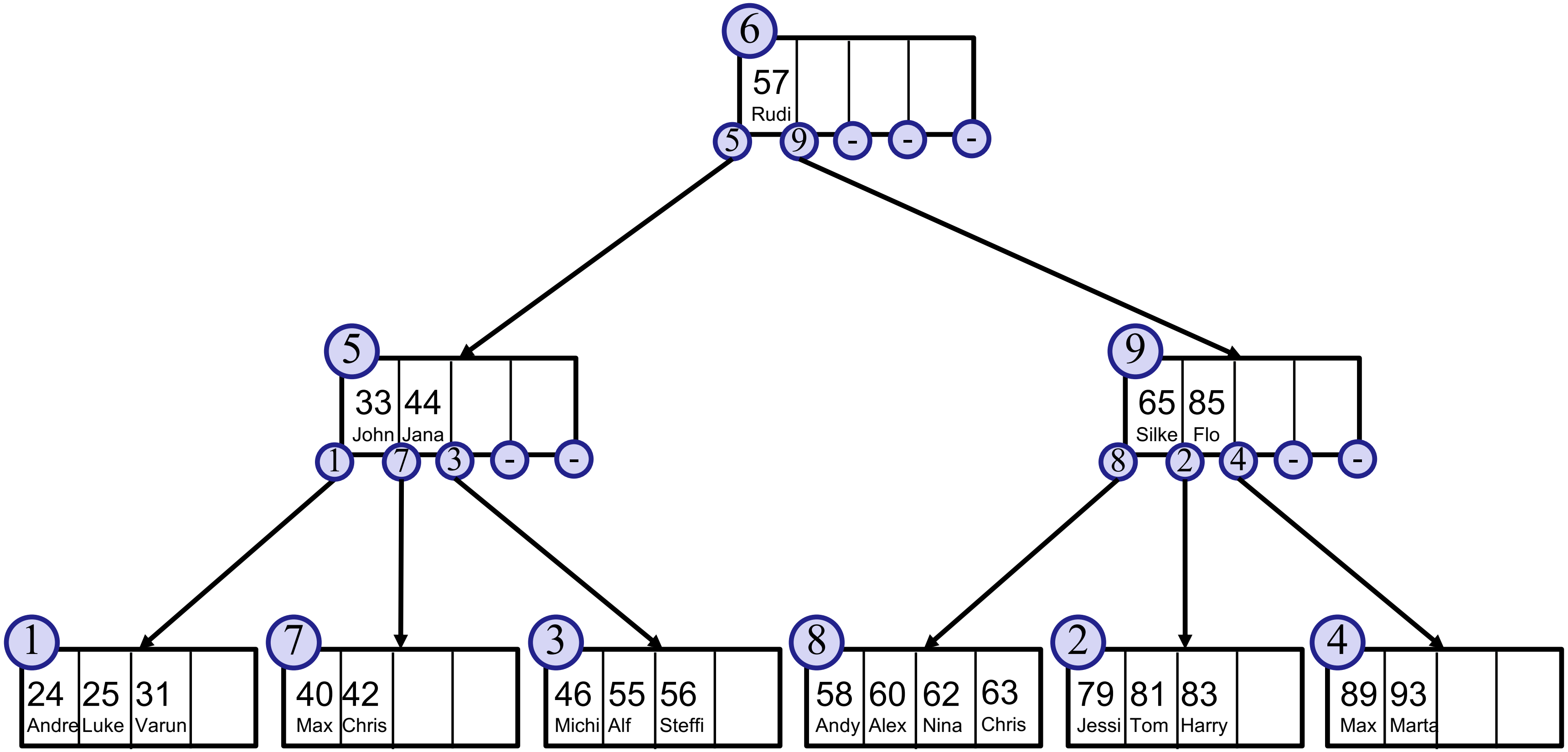
… are often balanced

(otherwise in extreme cases rather a chain)

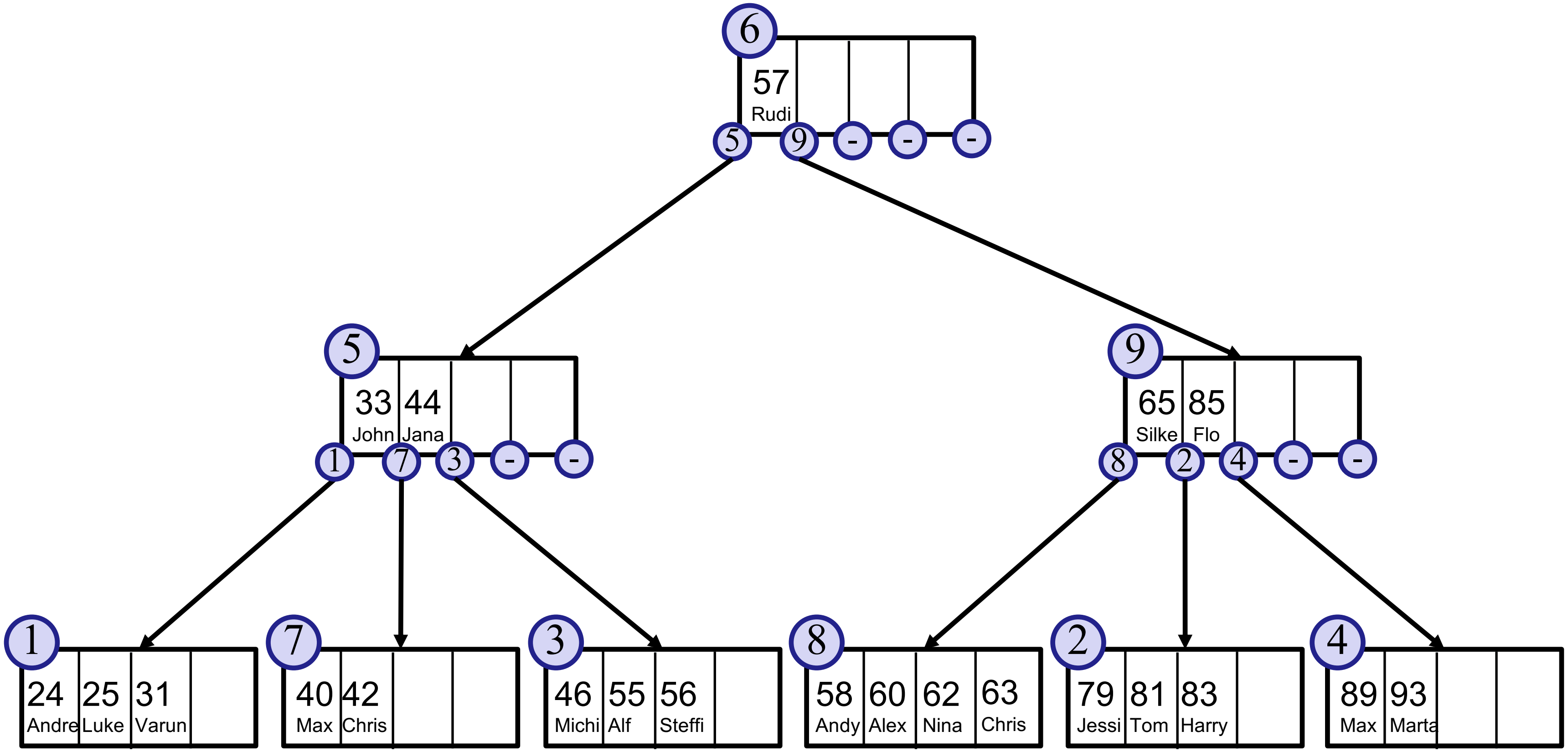
Schematic depiction of a balanced tree:



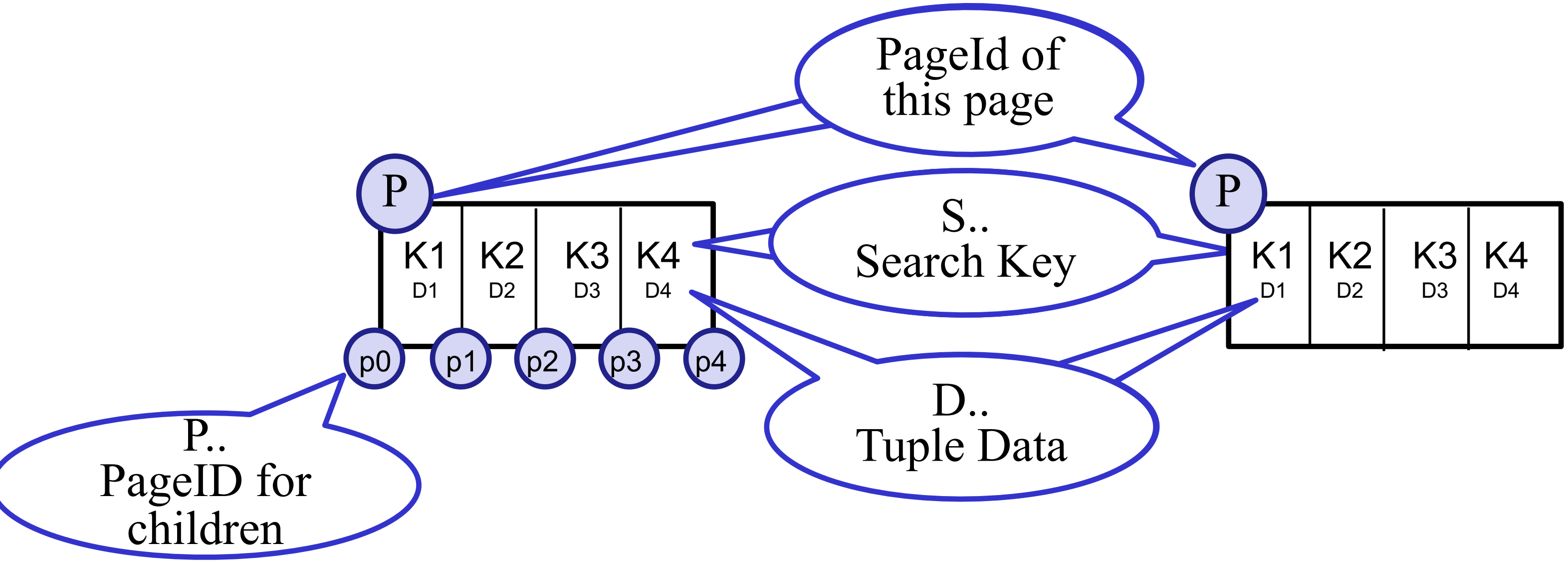
# B-Tree (degree = 2)



# B-Tree (degree = 2)



# Node Structure



Inner nodes

Leaf nodes

Tree properties:

- One node is one page
- Tree is balanced
- Node utilization at least 50%

# Properties of a B-Tree

B-Tree of degree *i* has following properties:

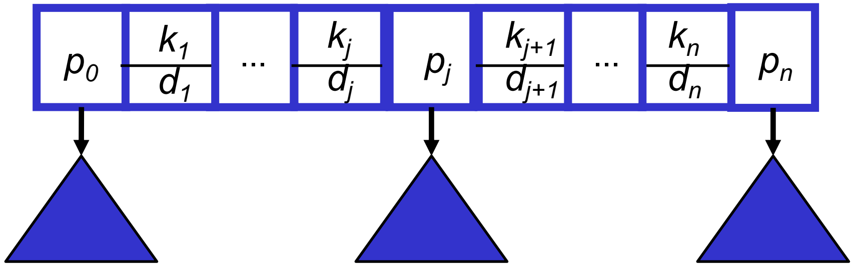
- Every path from the root to a leaf has the same length
- Every node – except the root – has at least *i* and at most *2i entries* (in the example above *i*=*2*)
- Entries in every node are sorted
- Every node – except the leaves – with *n* entries has *n+1* children

# Properties of a B-Tree

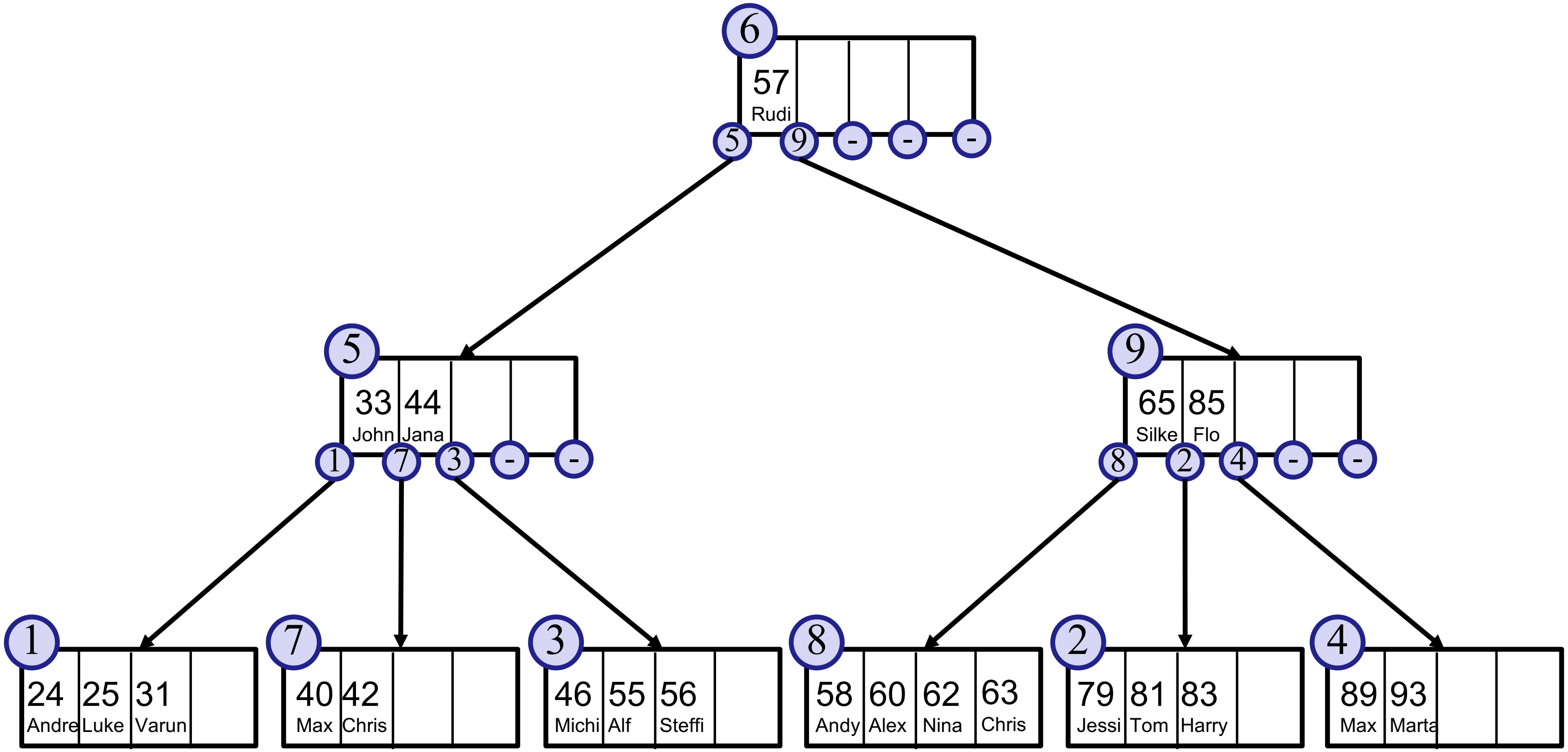
- *Let*
  $p_0, k_1, p_1, k_2, \ldots k_n, p_n$
  be entries in a node ($p_j$ are page identifier, $k_j$ keys)

Then the following holds:

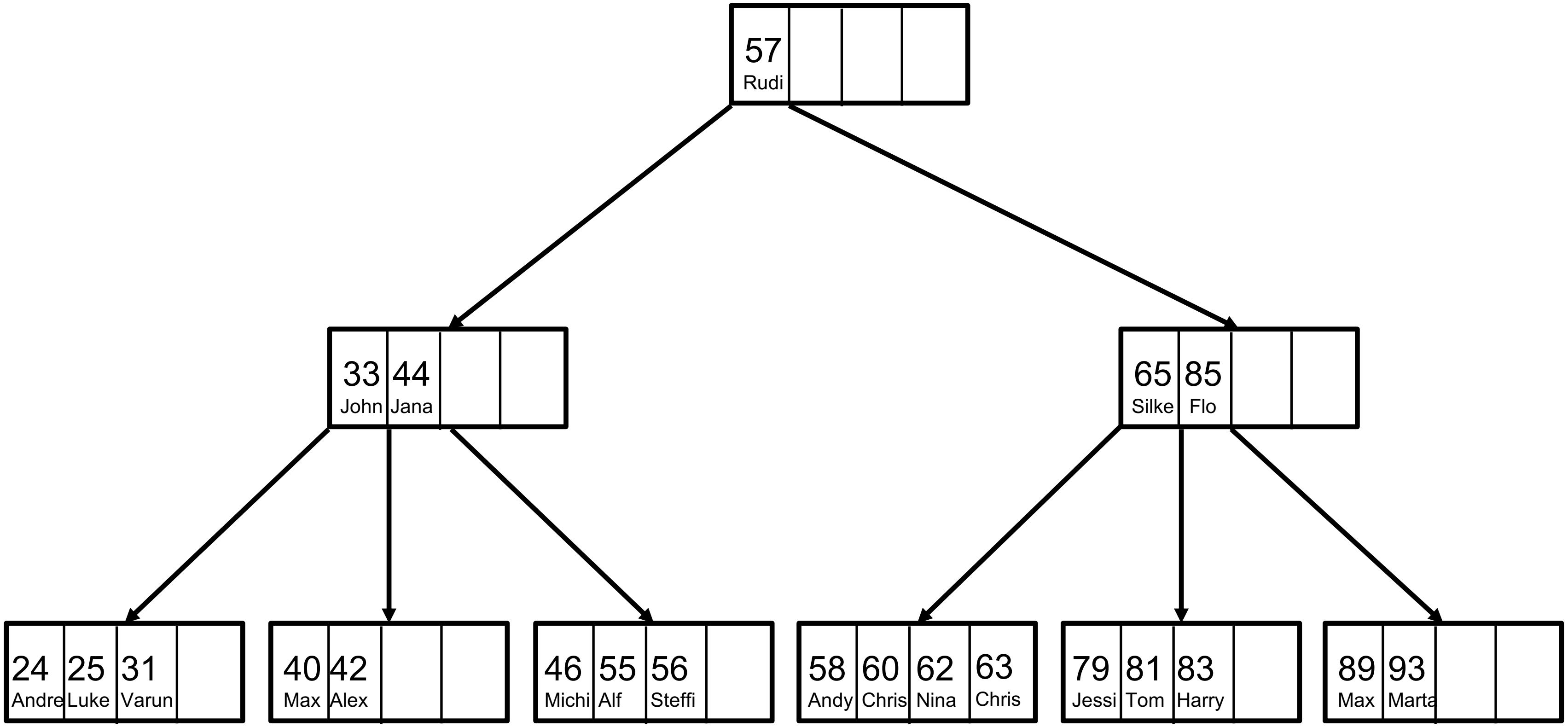
1. Sub-tree in $p_0$ contains only keys smaller than $k_1$
2. $p_j$ has a sub-tree with keys between $k_j$ and $k_j+1$
3. Sub-tree being referenced by $p_n$ contains only keys greater than $k_n$



# B-Tree (degree = 2)



# B-Tree (degree = 2)

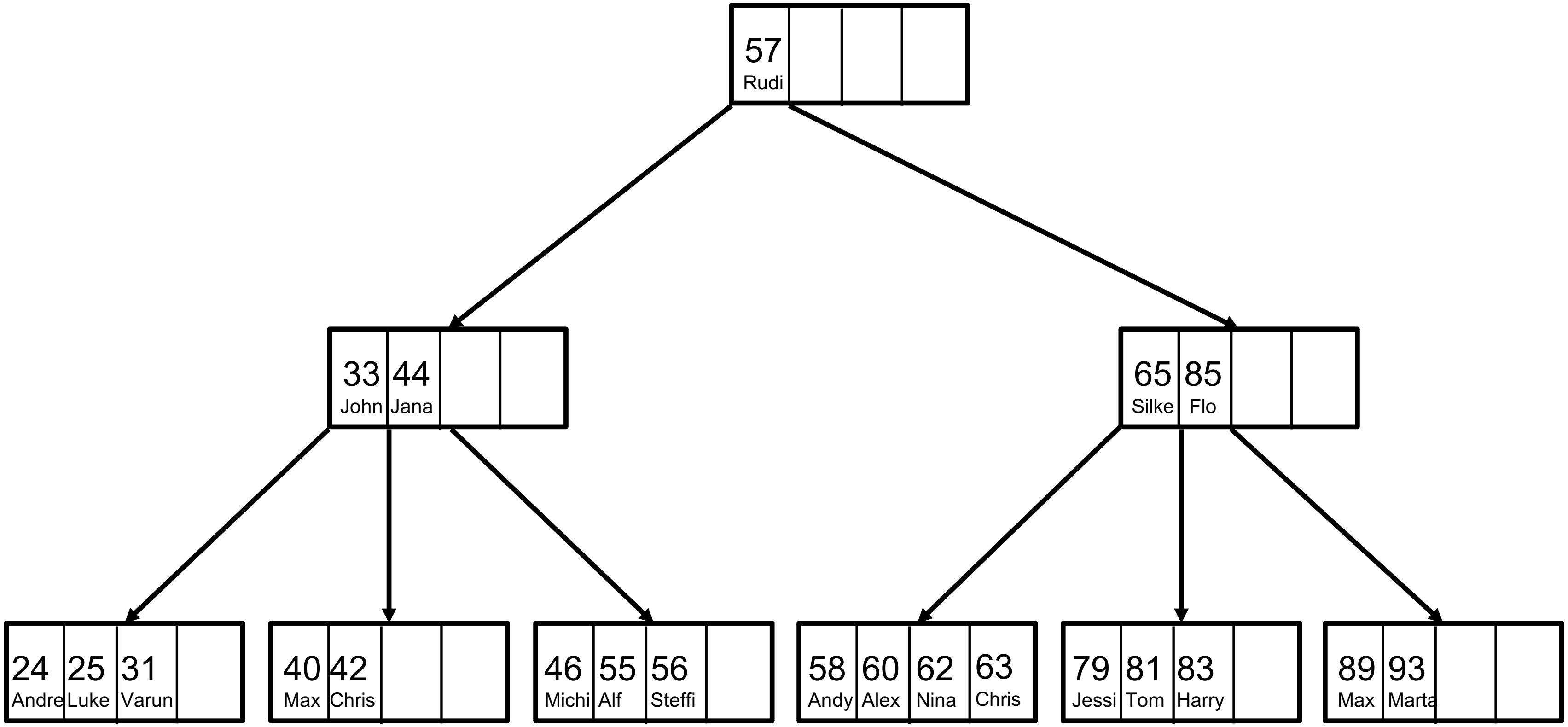


# B-Tree Size

# Insert Algorithm

1. Find the proper leaf node to insert new key
2. Insert key there
3. If node full
   i. Divide node into two and extract median
   ii. Insert all keys smaller than median into left node, all keys greater than median into right node
   iii. Insert median in parent node and adapt pointers
4. If parent node full
   i. If root node then create new root node, insert median, and adapt pointers
   ii. Otherwise repeat 3. with parent node

# B-Tree (degree = 2)



# Gradual Assembly of a B-Tree of Degree i=2

See:
https://db.in.tum.de/teaching/ws1819/DBSandere/BTreeExample.pdf

In the internet there are a number of animation programs for B-Trees – **no warranty!**

https://www.cs.usfca.edu/~galles/visualization/BTree.html
looks quite good, but uses a different notation for the maximal node size and does not handle node underflows.

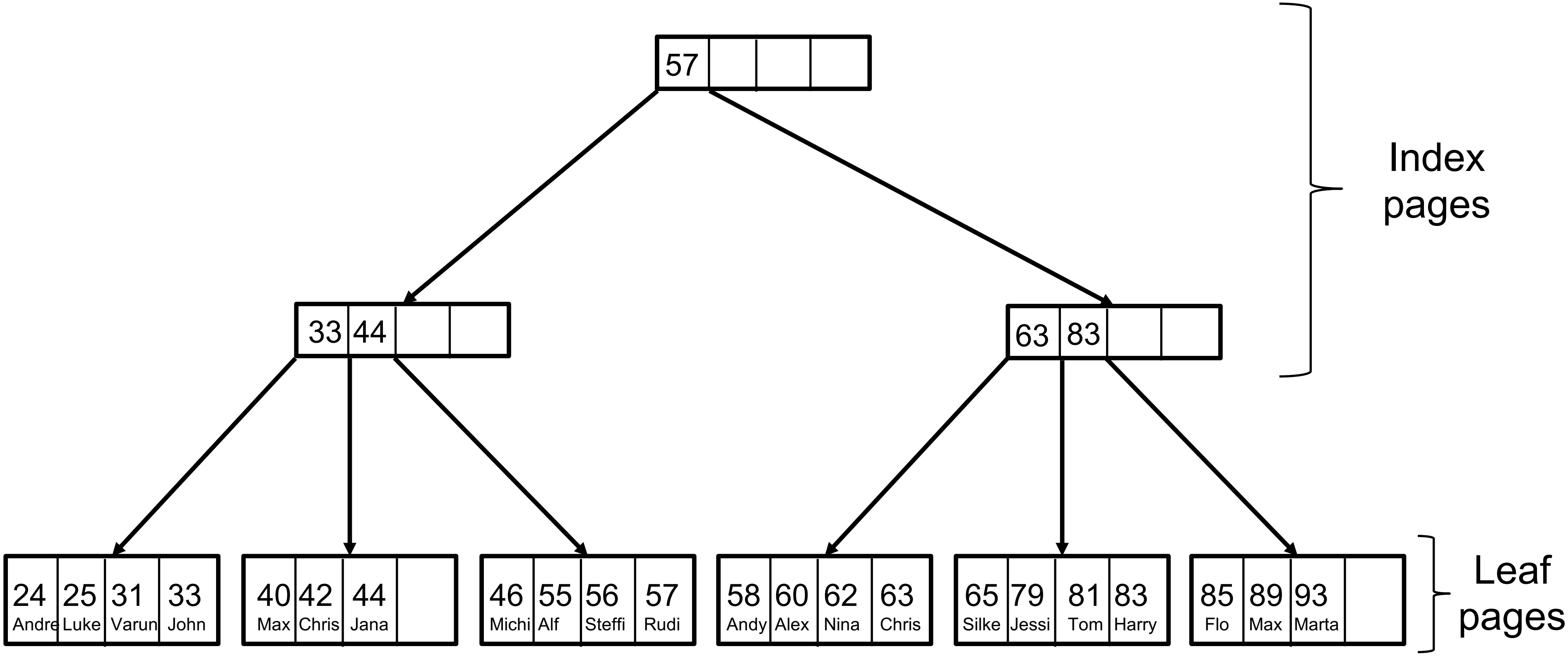
# Delete Algorithm

Read the literature or example on lecture website

# B+-Trees

- Performance of a B-Tree heavily depends on height: on average $\log_k(n)$ page accesses to read one data element
  (k=degree of branching, n=number of indexed data elements)
  → preferably high degree of branching of the inner nodes
- Storing data in the inner nodes reduces branching degree
- B+-Trees only store reference keys in inner nodes – data itself is stored in leaf nodes
- Usually leaf nodes are bidirectionally linked in order to enable fast sequential search

# B+-Tree


57
33 44
63 83
24 Andre
25 Luke
31 Varun
33 John
40 Max
42 Chris
44 Jana
46 Michi
55 Alf
56 Steffi
57 Rudi
58 Andy
60 Alex
62 Nina
63 Chris
65 Silke
79 Jessi
81 Tom
83 Harry
85 Flo
89 Max
93 Marta
Index pages
Leaf pages

# Prefix B+-Trees

- Further Improvement by use of prefixes of reference keys, e.g. with long strings as keys
- You only have to find a reference key which separates the left and the right sub-tree:
  - Disestablishment <= E < Incomprehensibility
  - Systemprogram <= ? < Systemprogrammer

# Several Indexes on the same Data

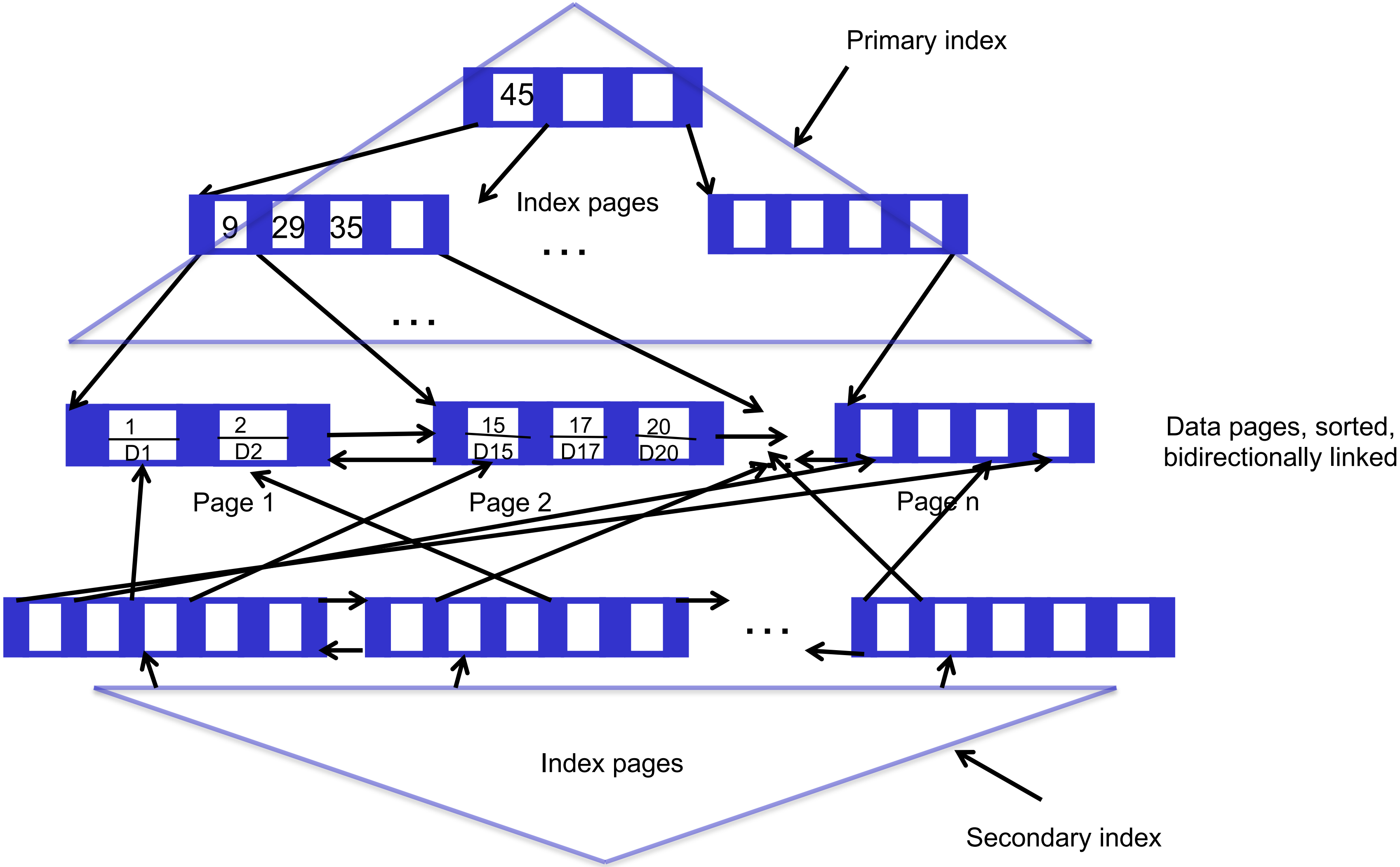
Primary index – Secondary index

| Students | | |
|---|---|---|
| StudNr | Name | Semester |
| 25403 | Jonas | 12 |
| 29120 | Theophrastos | 2 |
| 29555 | Feuerbach | 2 |
| 27550 | Schopenhauer | 6 |
| ⋮ | ⋮ | ⋮ |

When

- Index on StudNr?
- Index on Name?
- Index on Semester?

# Secondary indexes



# DDL: Create Index

```
CREATE [UNIQUE] INDEX index_name
ON table_name (column_name1 [, column_name2, …])
```

Example:

```
CREATE INDEX full_name
ON Person (Last_Name, First_Name)
```

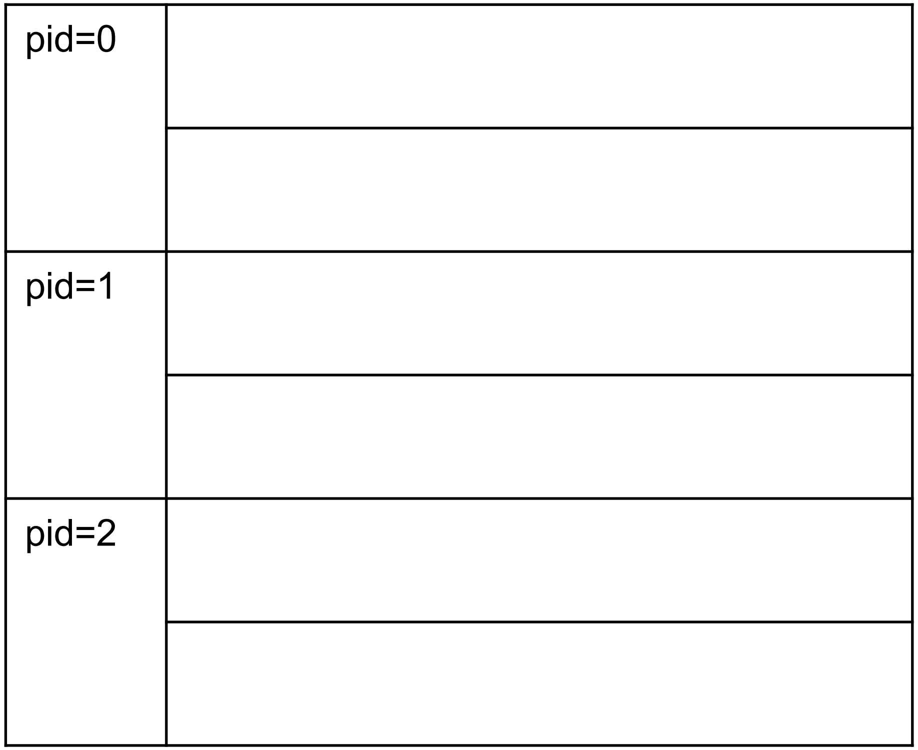
# Partitioning
# What is Hashing?

- (to hash = zerhacken)
- Storing tuples in a defined memory area
- Hash function: mapping tuples (key values) to a fixed set of function values (memory area)
- Optimal hash function:
  - injective (no identical function values for different arguments)
  - surjective (no waste of memory)
- Typical hash function h: h (x) = x mod N set of function values thereby {0,..., N-1}

# Hashing Example

*hash(x) = x mod 3*

25403, ‘Jonas’, 12
27550, ‘Schopenhauer‘, 3
26120, ‘Fichte’, 10



pid=0

pid=1

pid=2

# Hashing Example

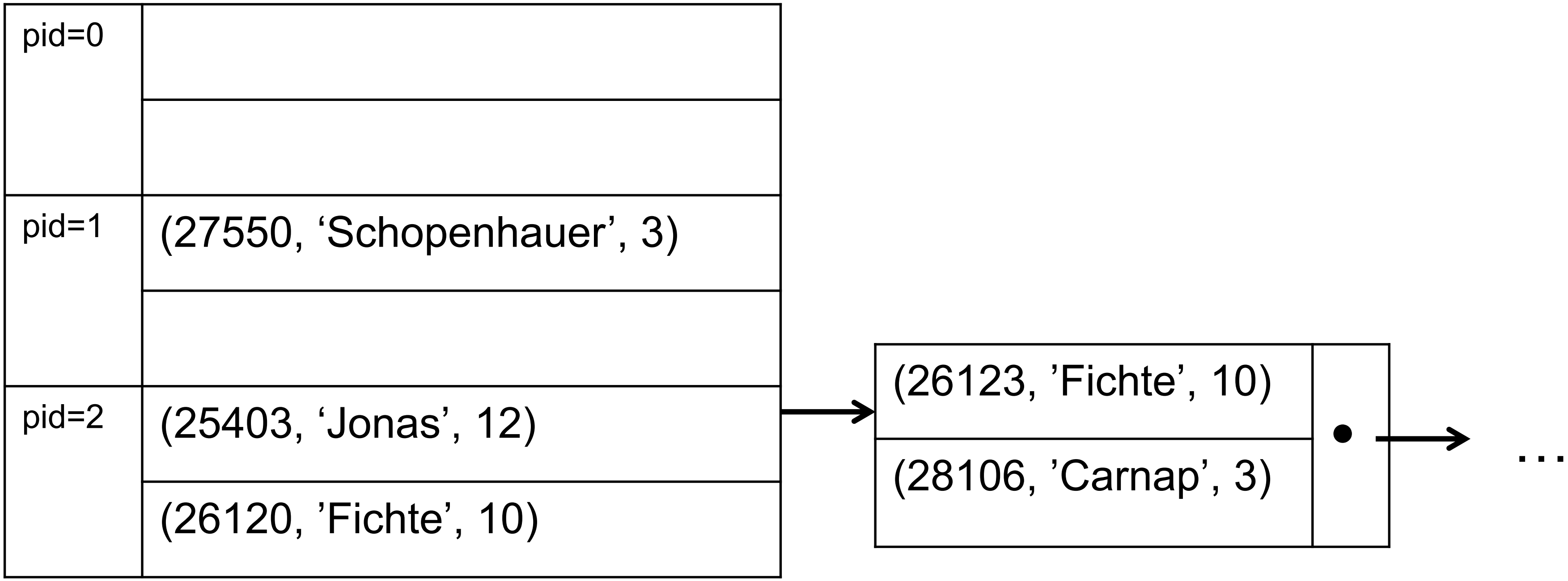
*hash(x) = x mod 3*

25403, ‘Jonas’, 12
27550, ‘Schopenhauer‘, 3
26120, ‘Fichte’, 10

| | |
|---|---|
| pid=0 | |
| | |
| pid=1 | (27550, ‘Schopenhauer’, 3) |
| | |
| pid=2 | (25403, ‘Jonas’, 12) |
| | (26120, ’Fichte’, 10) |

# Collisions

Collision handling



pid=0

pid=1

(27550, ‘Schopenhauer’, 3)

pid=2

(25403, ‘Jonas’, 12)

(26120, ’Fichte’, 10)

(26123, ’Fichte’, 10)

(28106, ’Carnap’, 3)

…

Could be inefficiently: hash table is too small, a bad hash function is used, unlucky, adversary input

Way out: extensible (dynamic) Hashing

# Advantages / Disadvantages Hashing

+ Few accesses to external storage
  constant cost: O(1), generally 1-2
+ Simple implementation

- Collision handling necessary
- Pre-allocation of memory area
- Not dynamic resp. only with adjustment
- **No range queries, only point queries**