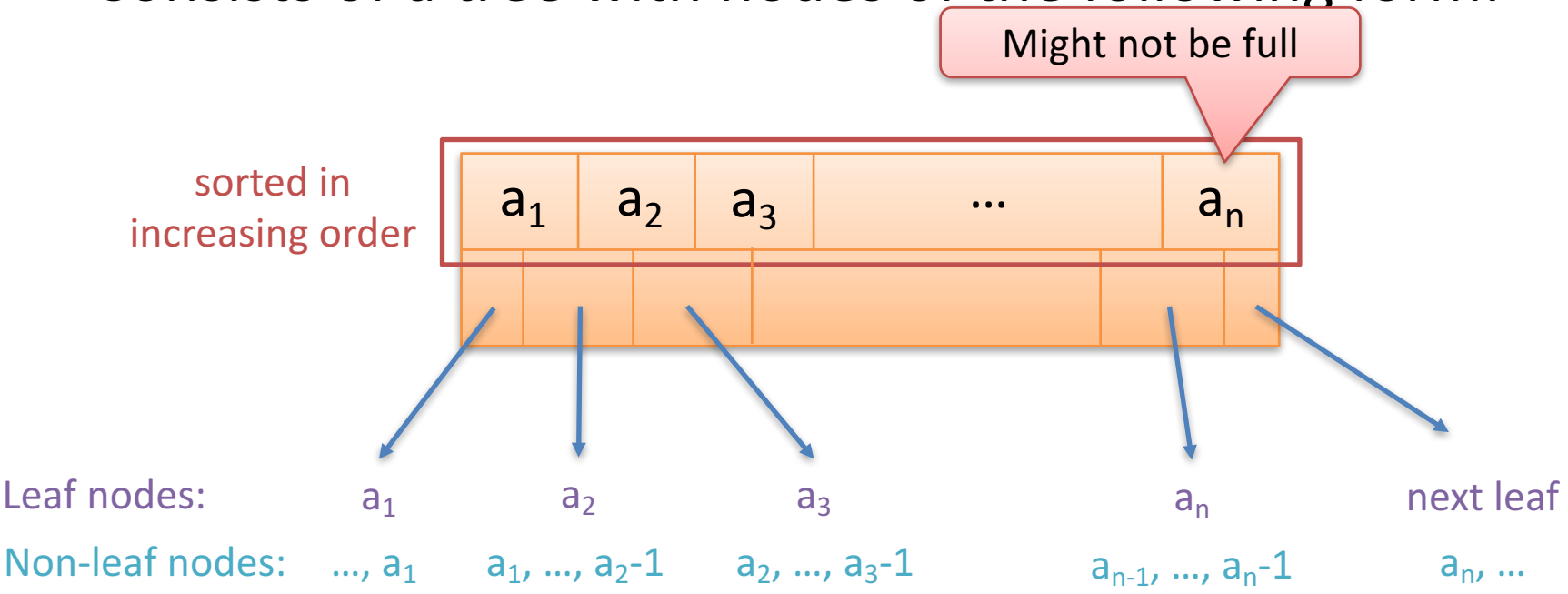
COMP207
Database Development

Lecture 16

Query Processing:
Optimisation of Query Plans

Review: B+ Tree Index

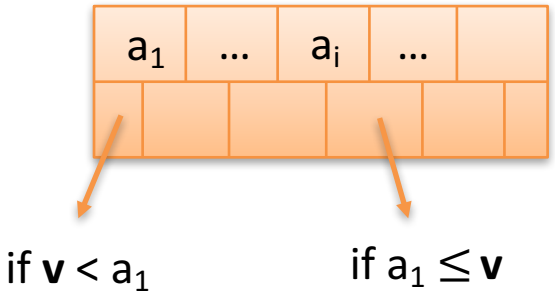
- Particular implementation of an index
- Consists of a tree with nodes of the following form:



- **n** = chosen such that node fits into a single disk block

Review: Finding value

- Goal: find the pointer to the rows with value **v**
- Procedure:
  - Start at the root of the B+ tree
  - While the current node is a non-leaf node:
    - If **v** < a_1, proceed to the first child of the node
    - Otherwise find the **largest i** with $a_i \leq$ **v** and proceed to the associated child node
  - If the current node is a leaf:
    - If **v** occurs in the leaf, follow the associated pointer
    - If **v** does not occur in the leaf, return "**v** does not exist in index"



- Running time: $O(h \times \log_2 n)$ "real" running time $O(h \times D)$

Height of the B+ tree

Time for a disk operation

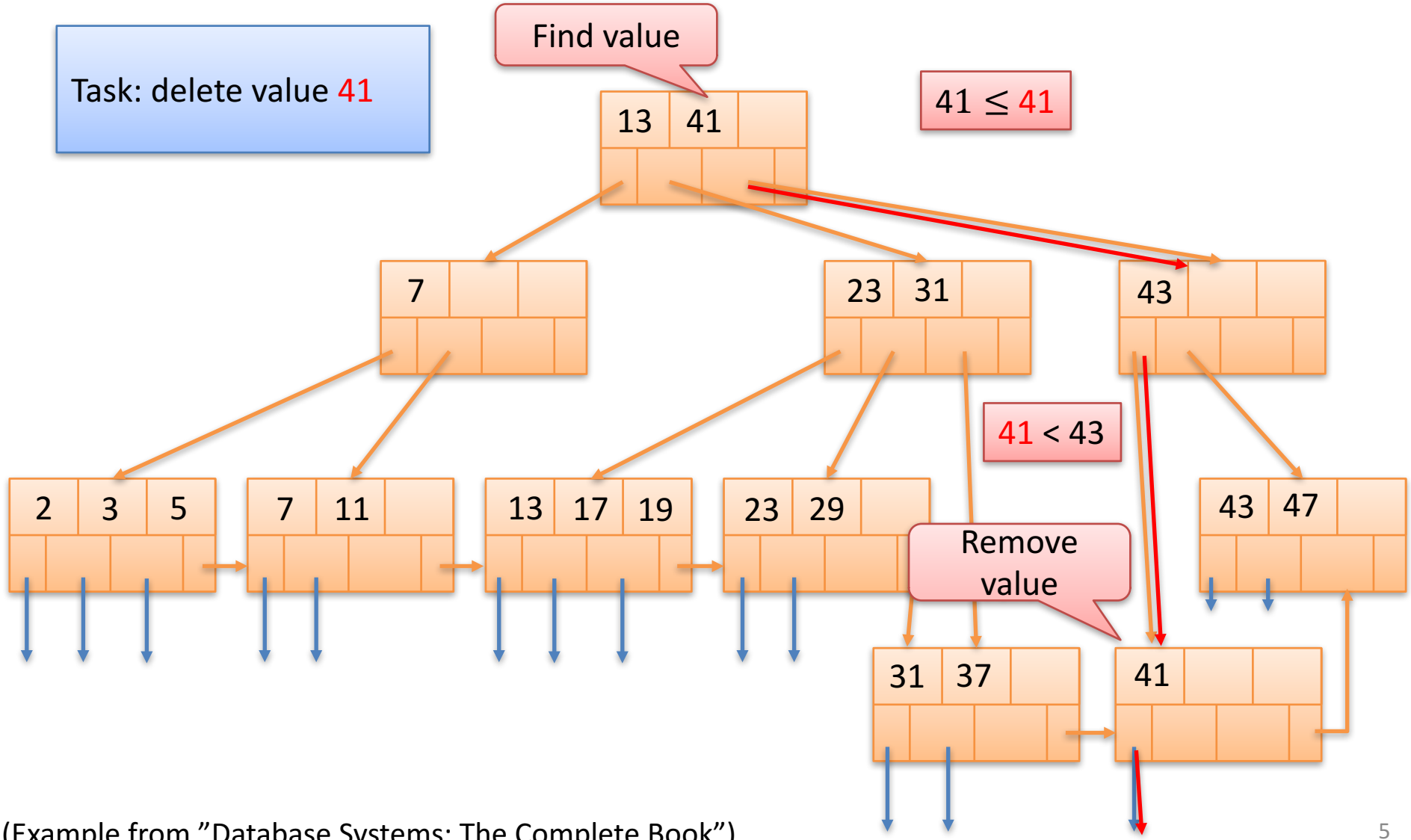
Review: Insertion

- Goal: insert a new value/pointer pair
- Procedure:
  - Find the leaf that should contain the value
  - If the leaf is not full, insert the key value pair at a suitable location
  - If the leaf is full:
    - Split the leaf to make space for the new value/pointer pair
    - Insert the value/pointer pair
    - Connect the leaf to a suitable parent node (which might incur the creation of a new node etc.)
- The B+ tree **remains balanced**!
- Running time: $O(h \times \log_2 n)$ "real" running time $O(h \times D)$

Time for a disk operation

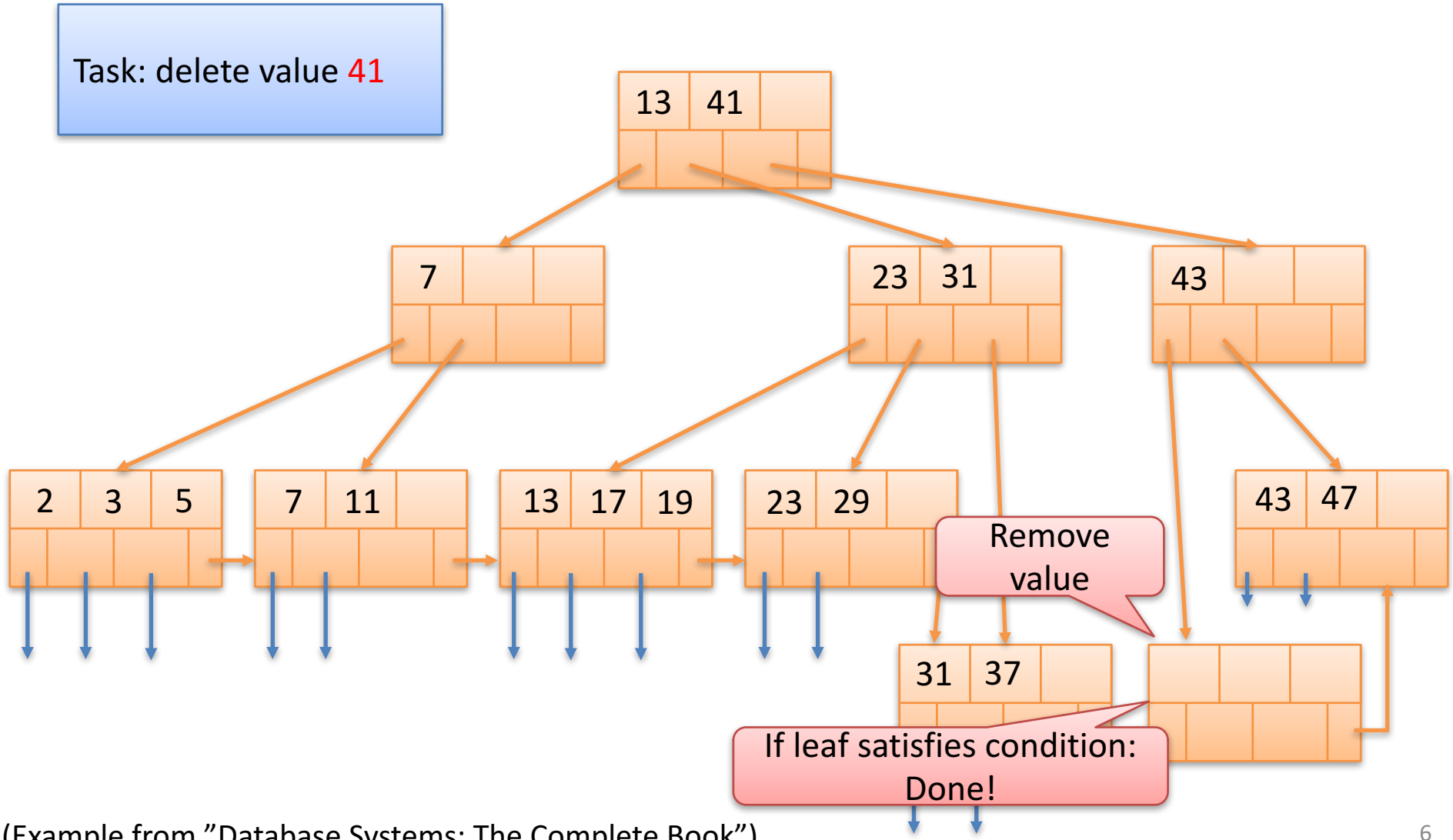
Height of the B+ tree

Deletion part 2



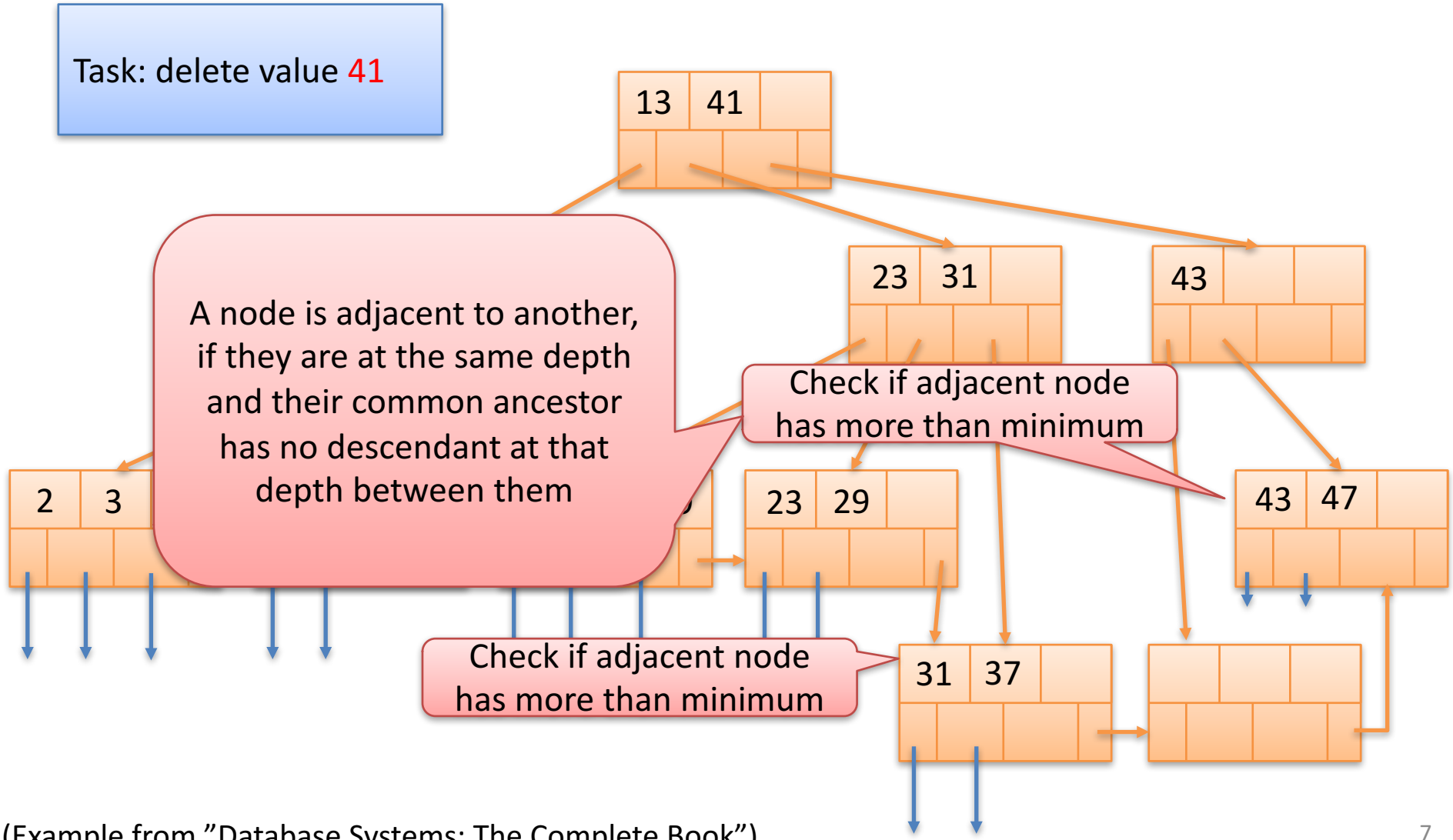
(Example from ”Database Systems: The Complete Book”)

Deletion part 2



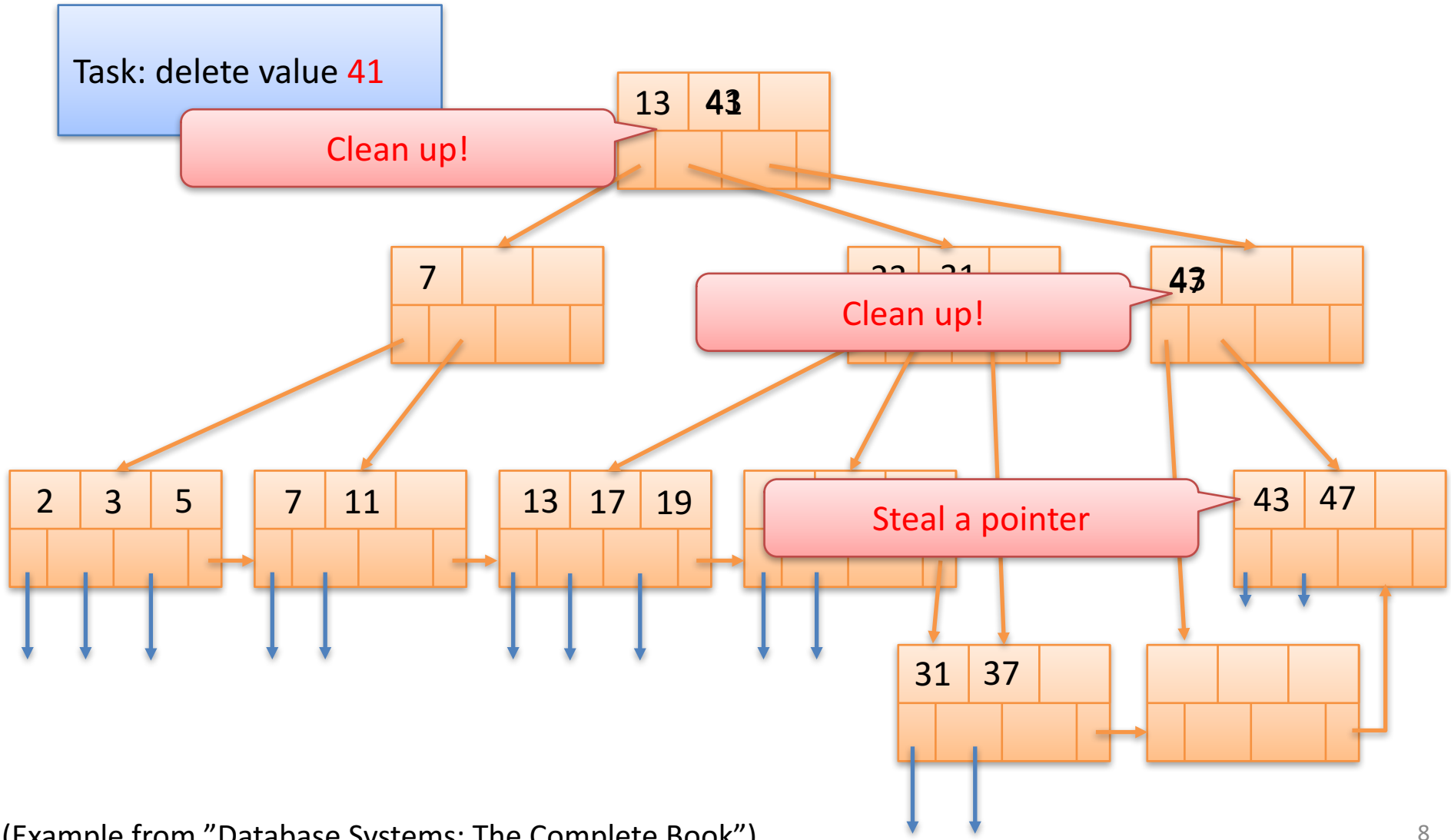
(Example from ”Database Systems: The Complete Book”)

Deletion part 2



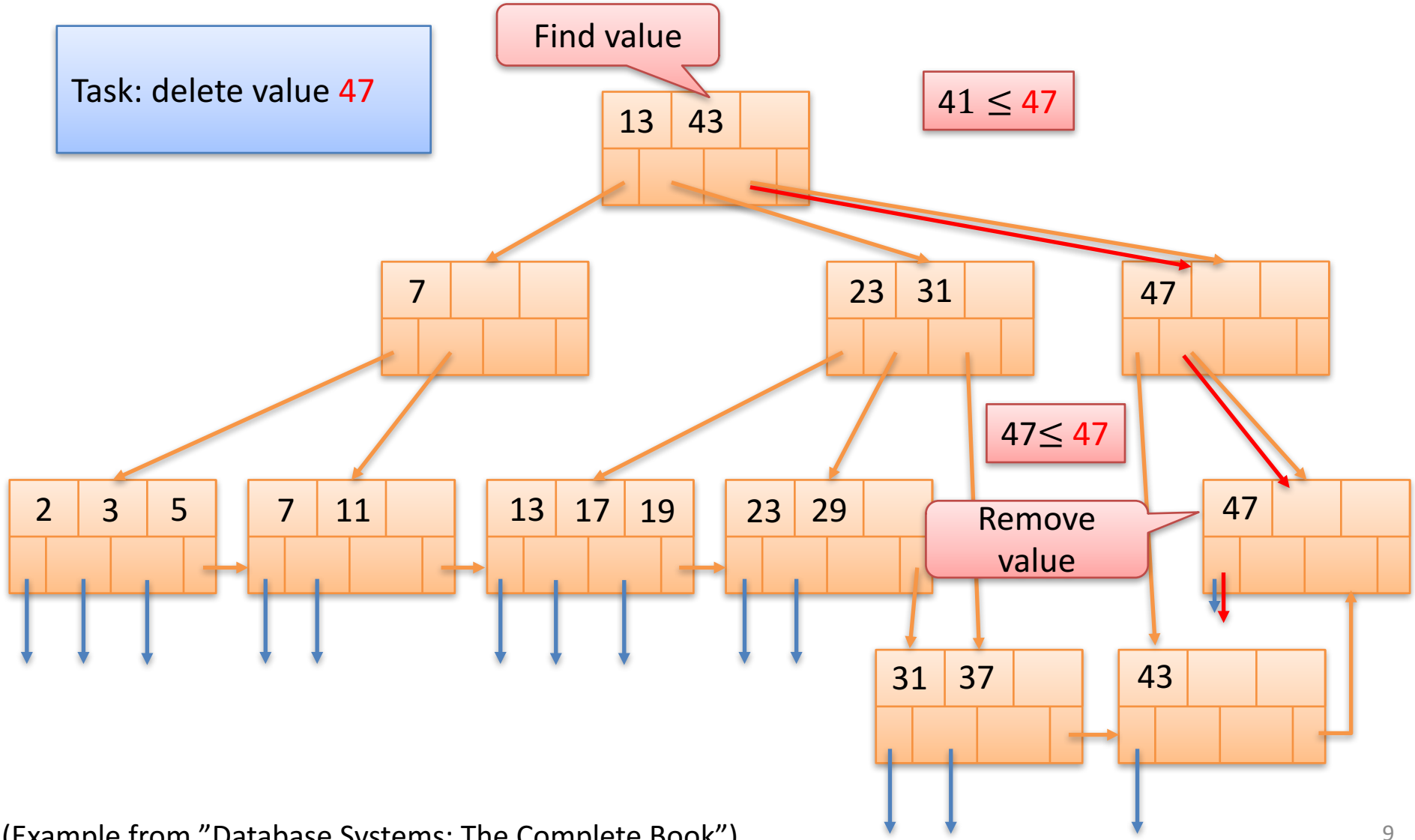
(Example from ”Database Systems: The Complete Book”)

Deletion part 2



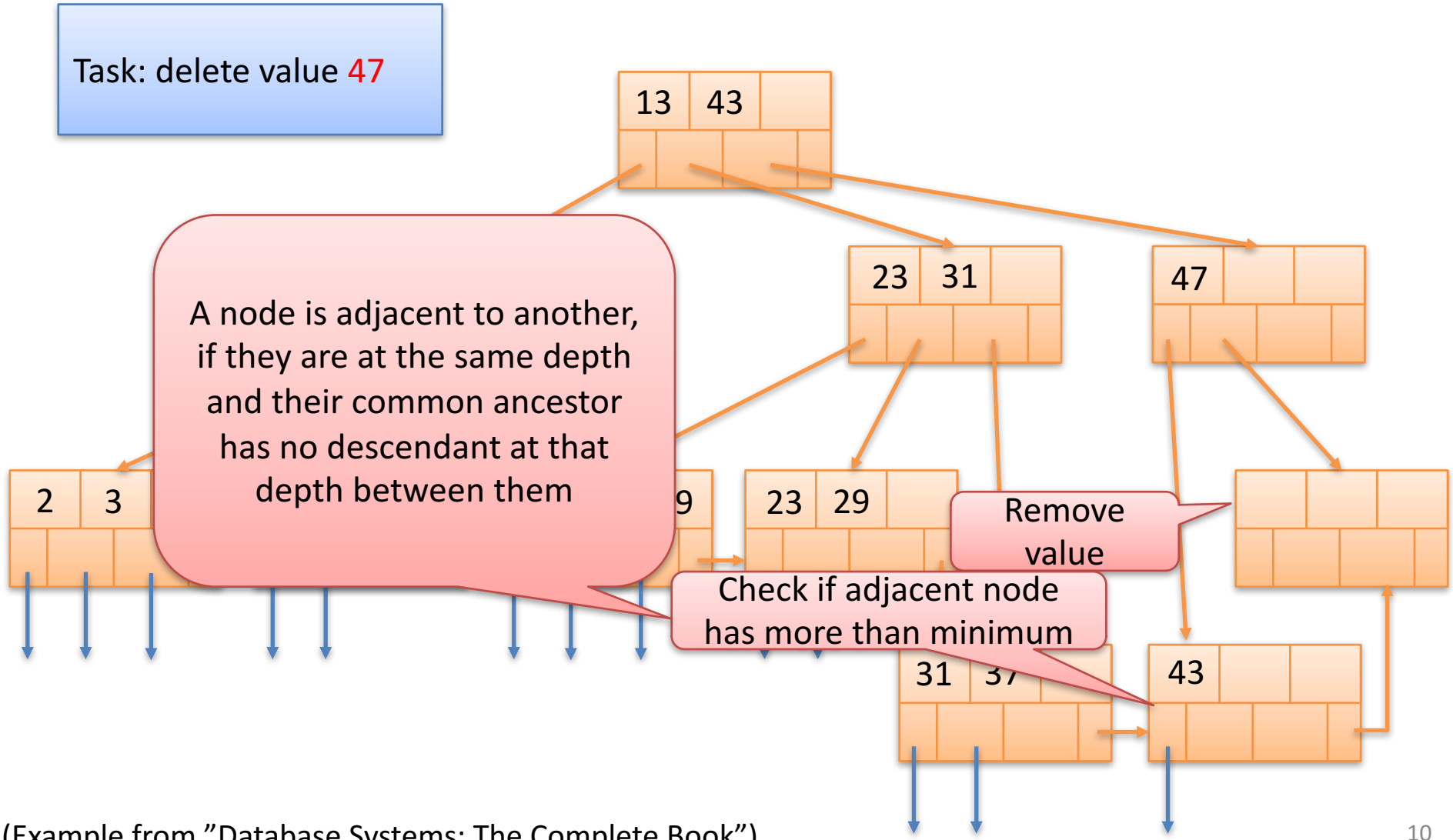
(Example from ”Database Systems: The Complete Book”)

Deletion part 3



(Example from ”Database Systems: The Complete Book”)

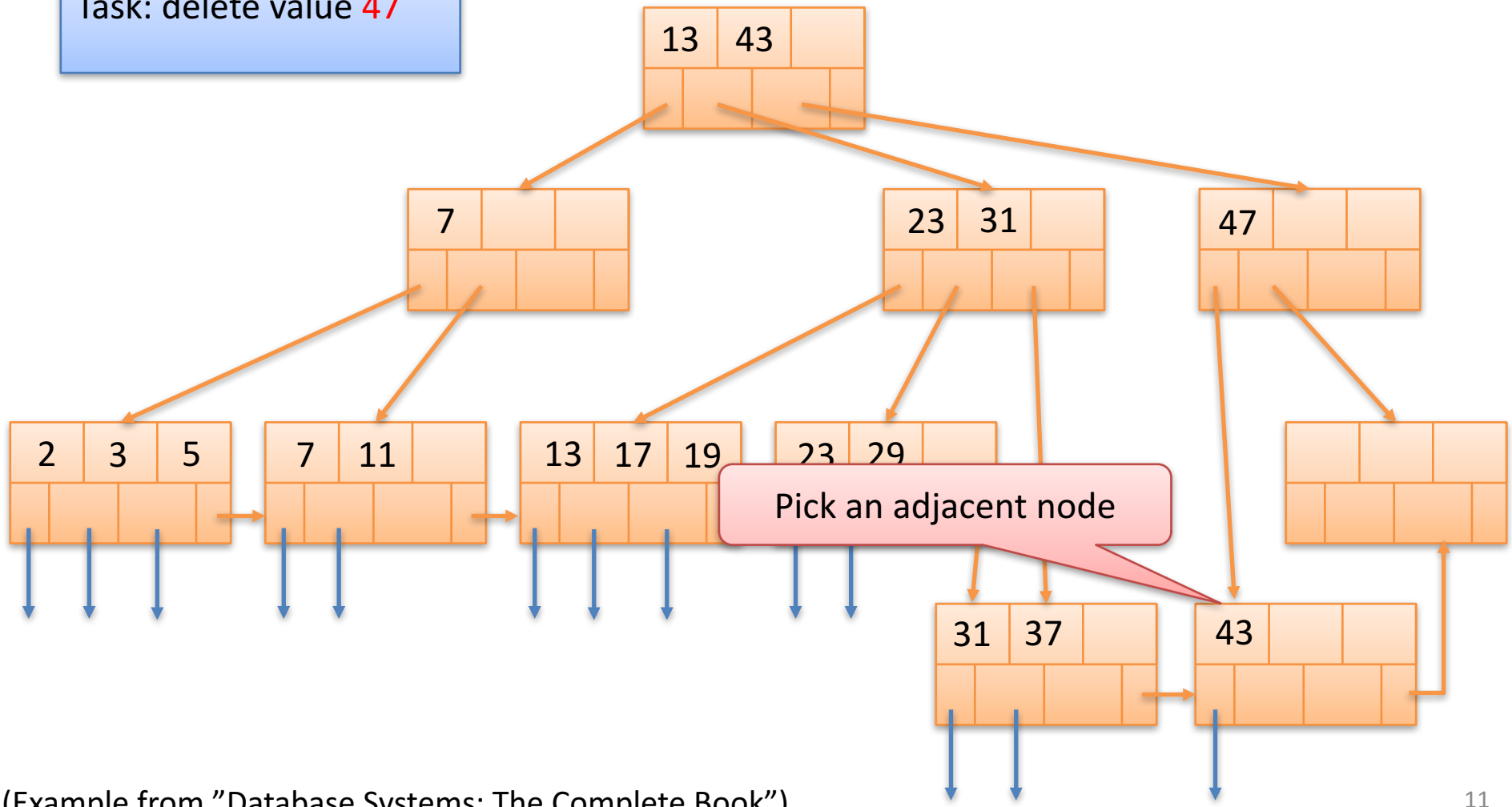
Deletion part 3



(Example from ”Database Systems: The Complete Book”)

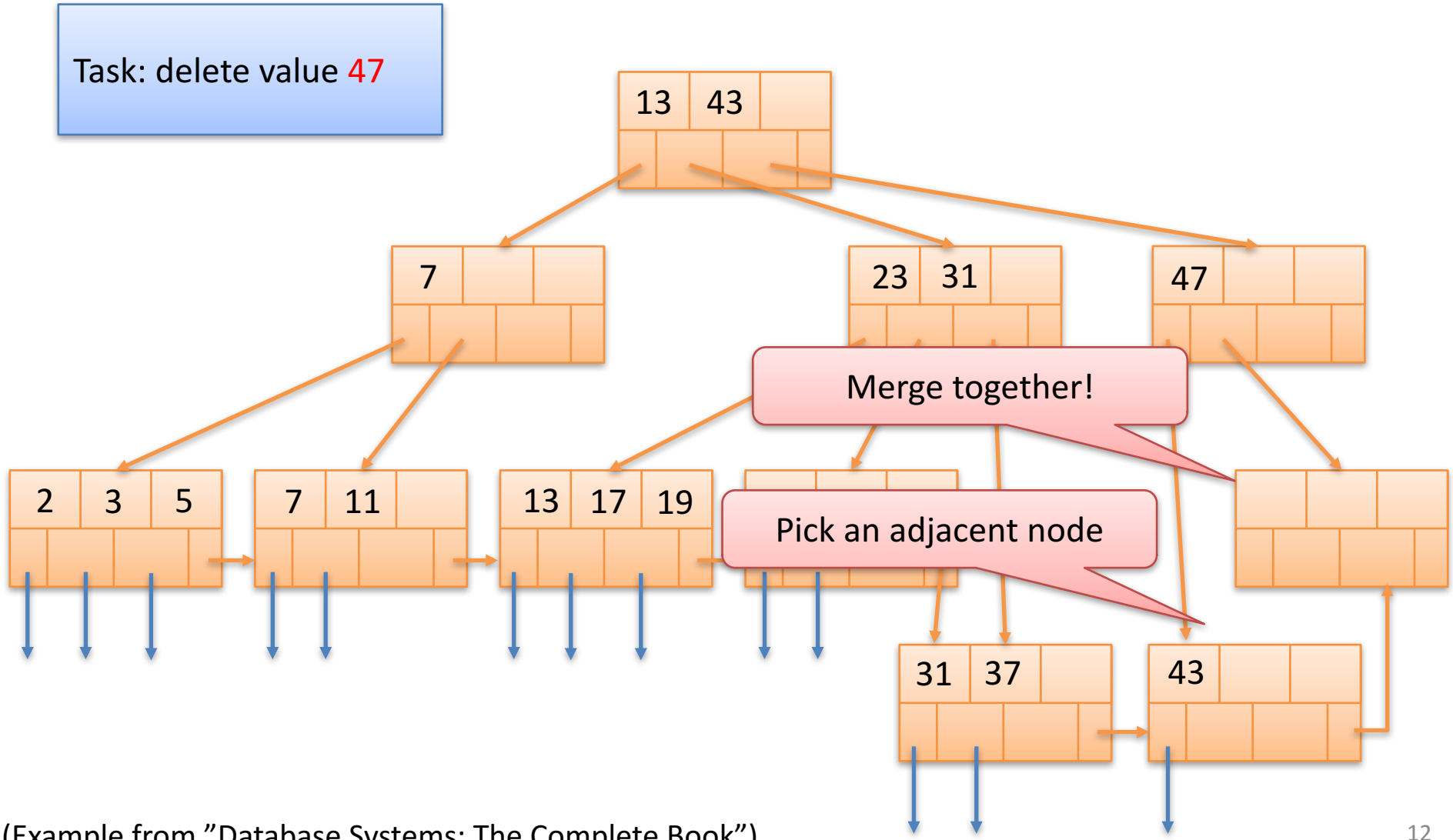
Deletion part 3

Task: delete value 47



(Example from ”Database Systems: The Complete Book”)

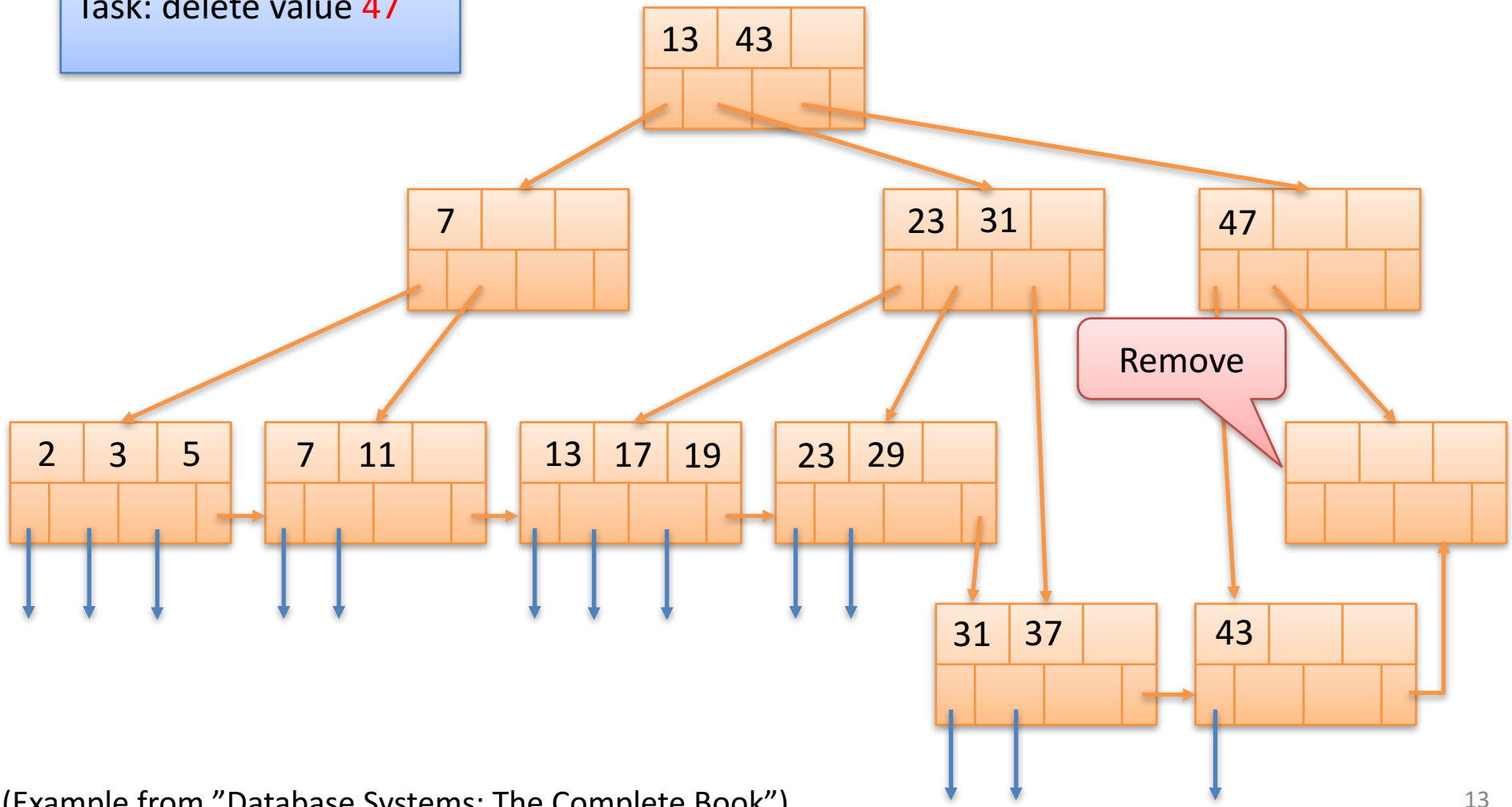
Deletion part 3



(Example from ”Database Systems: The Complete Book”)

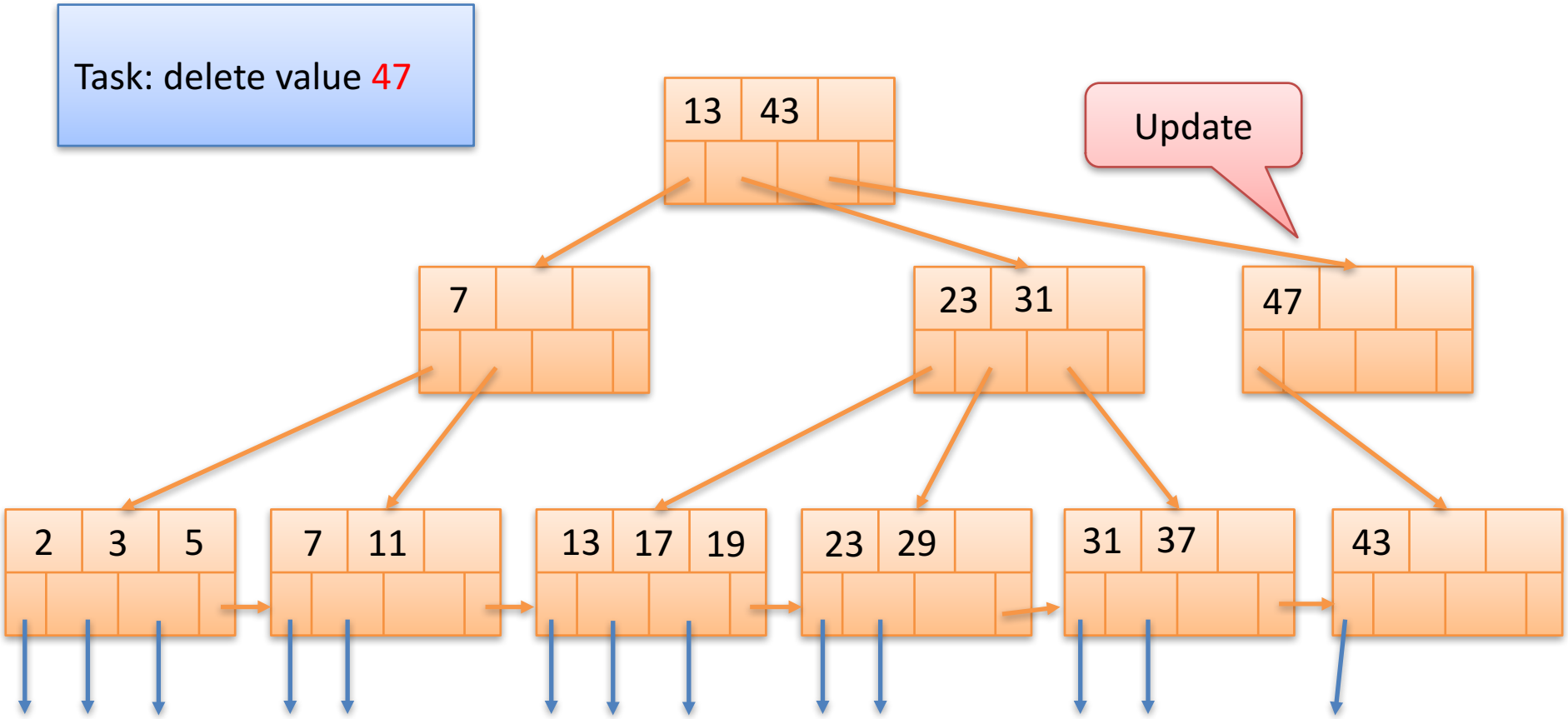
Deletion part 3

Task: delete value 47



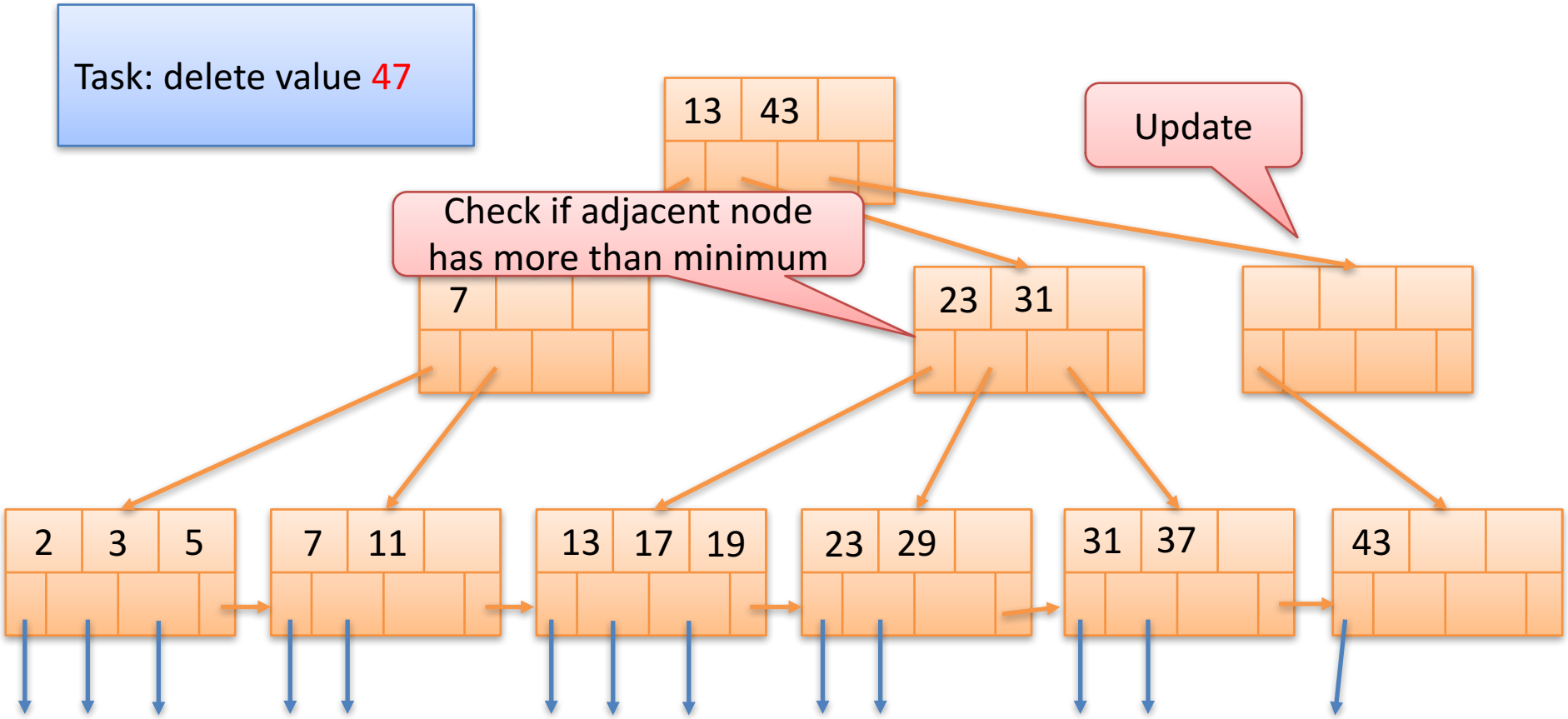
(Example from ”Database Systems: The Complete Book”)

Deletion part 3



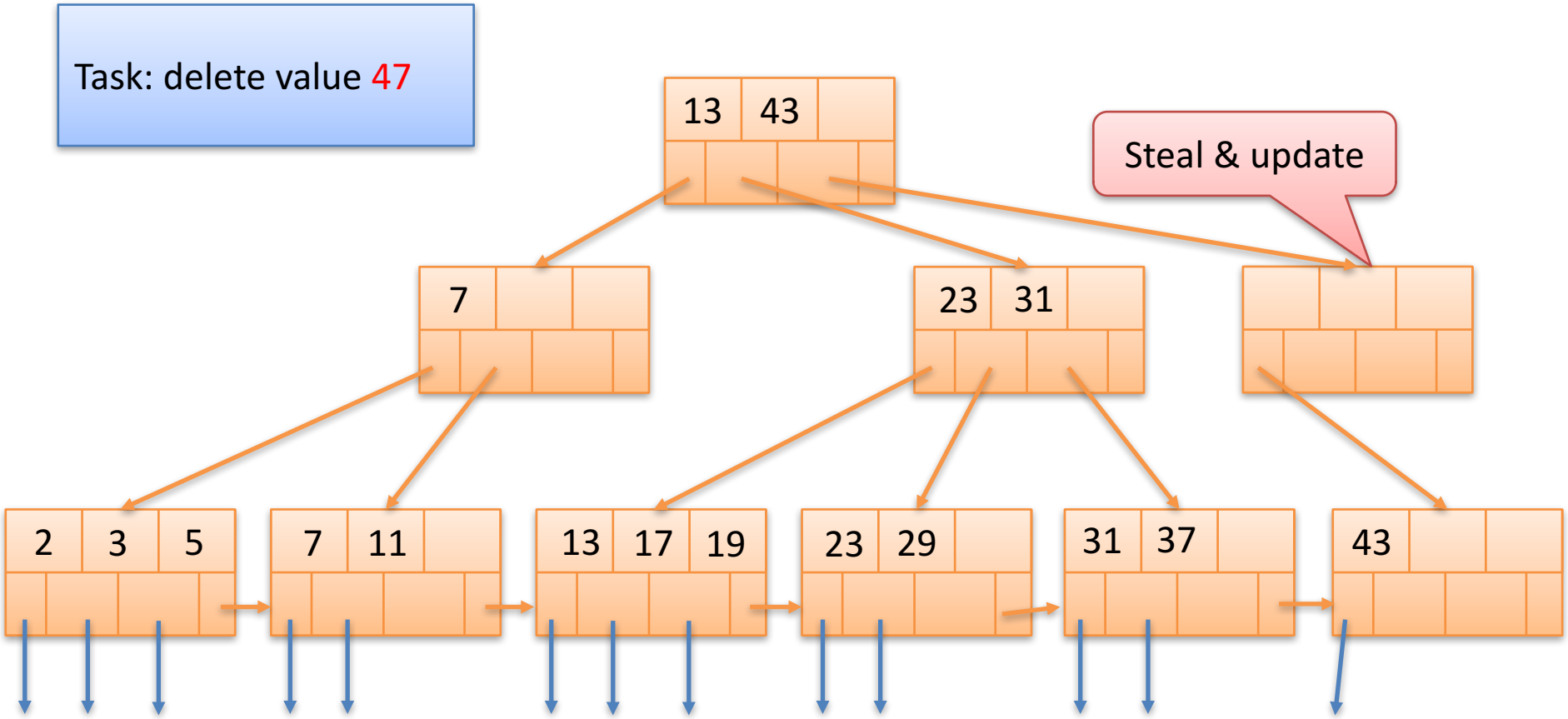
(Example from ”Database Systems: The Complete Book”)

Deletion part 3



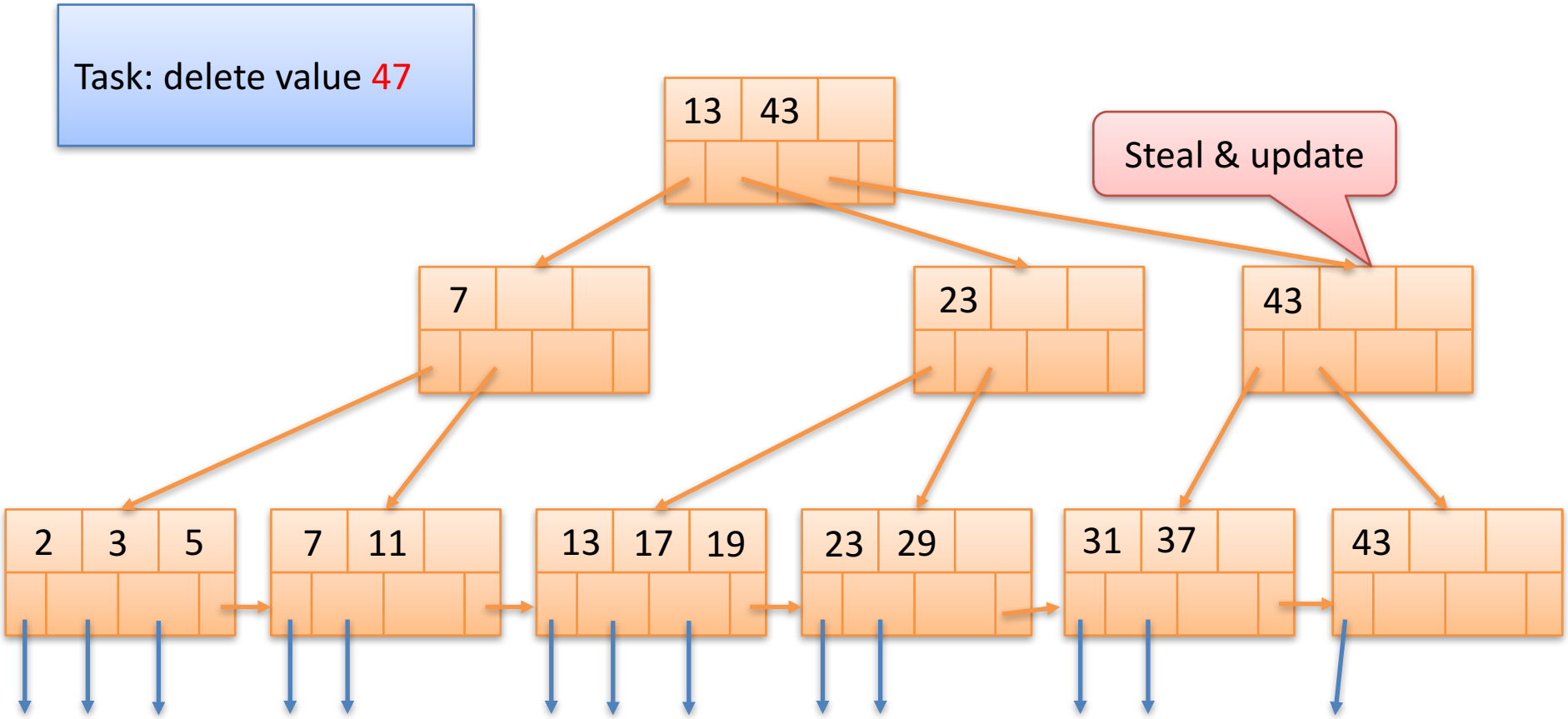
(Example from ”Database Systems: The Complete Book”)

Deletion part 3



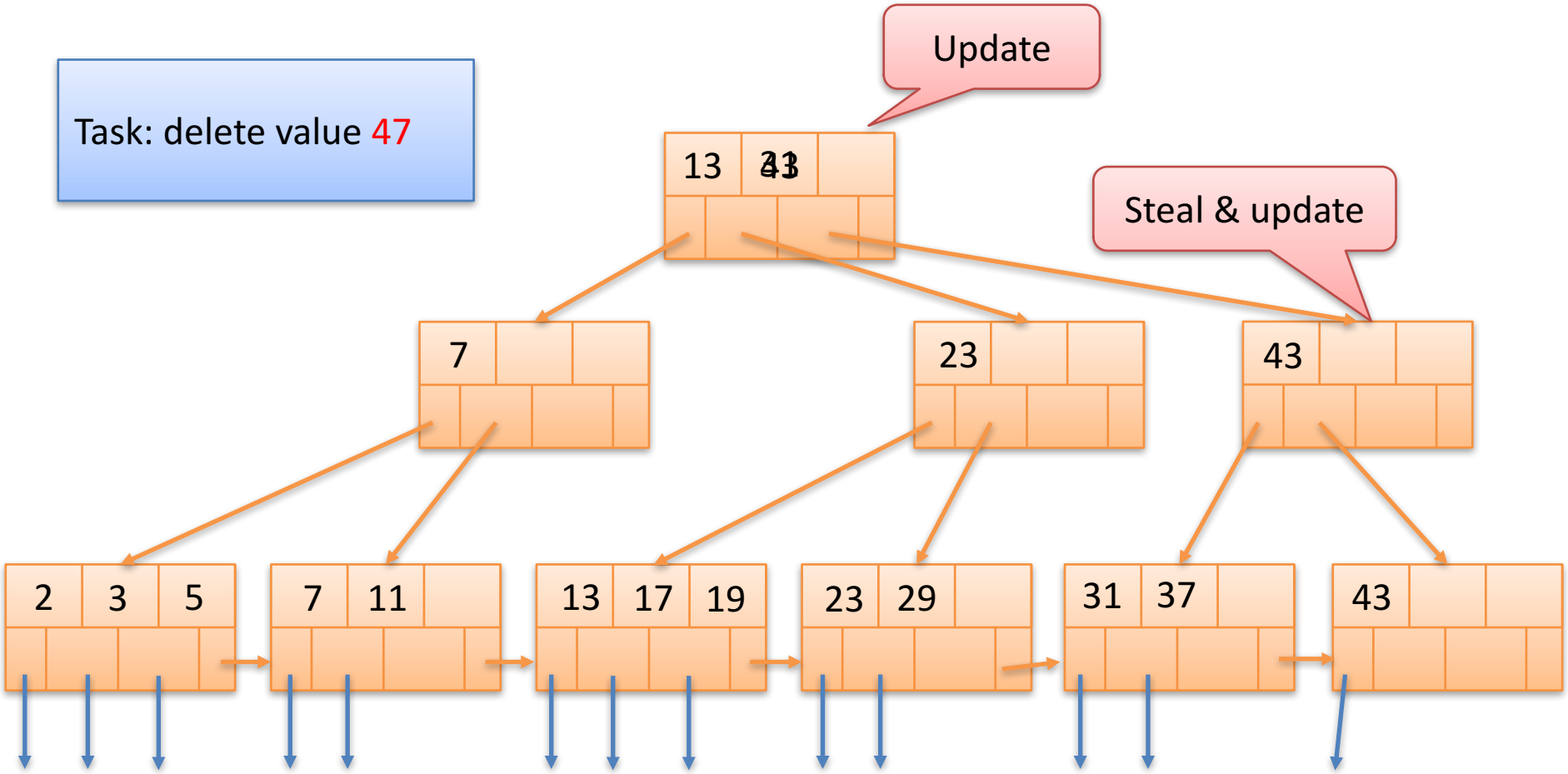
(Example from ”Database Systems: The Complete Book”)

Deletion part 3



(Example from ”Database Systems: The Complete Book”)

Deletion part 3



(Example from ”Database Systems: The Complete Book”)

Review: Deletion

- Goal: delete a value/pointer pair
- Procedure:
  - Find the leaf that should contain the value
    - If not there: Done
  - Remove the value/pointer pair
  - Let the current node C
  - Let x be $\begin{cases} 2 & if\ C\ is\ root \\ \left\lceil \frac{n+1}{2} \right\rceil & if\ C\ is\ internal\ node \\ \left\lfloor \frac{n+1}{2} \right\rfloor & if\ c\ is\ leaf \end{cases}$
  - If C has above x pointers: Fix ancestors (if necessary) and you are done
  - If C is the root but not a leaf: Remove it (and let the child of the root be the new root)
  - Otherwise, check if an adjacent node has at least $x + 1$ pointers
  - If so: take one, fix ancestors (if necessary) and you are done
  - Otherwise, merge with sibling and go to line 3 with the parent as current node
- The B+ tree **remains balanced**!
- Running time: O(h × $\log_2$ n)

See definition slide 25

Time for a disk operation

"real" running time O(h × D)

Height of the B+ tree

Properties of B+ Tree Indexes

- **Fast lookups, insertions, deletions** in time:

$n \approx B$

$$O(\text{height of B+ tree} \times \log_2 n) = O(\log_n N \times \log_2 n)$$
$$= O(\log_2 N)$$

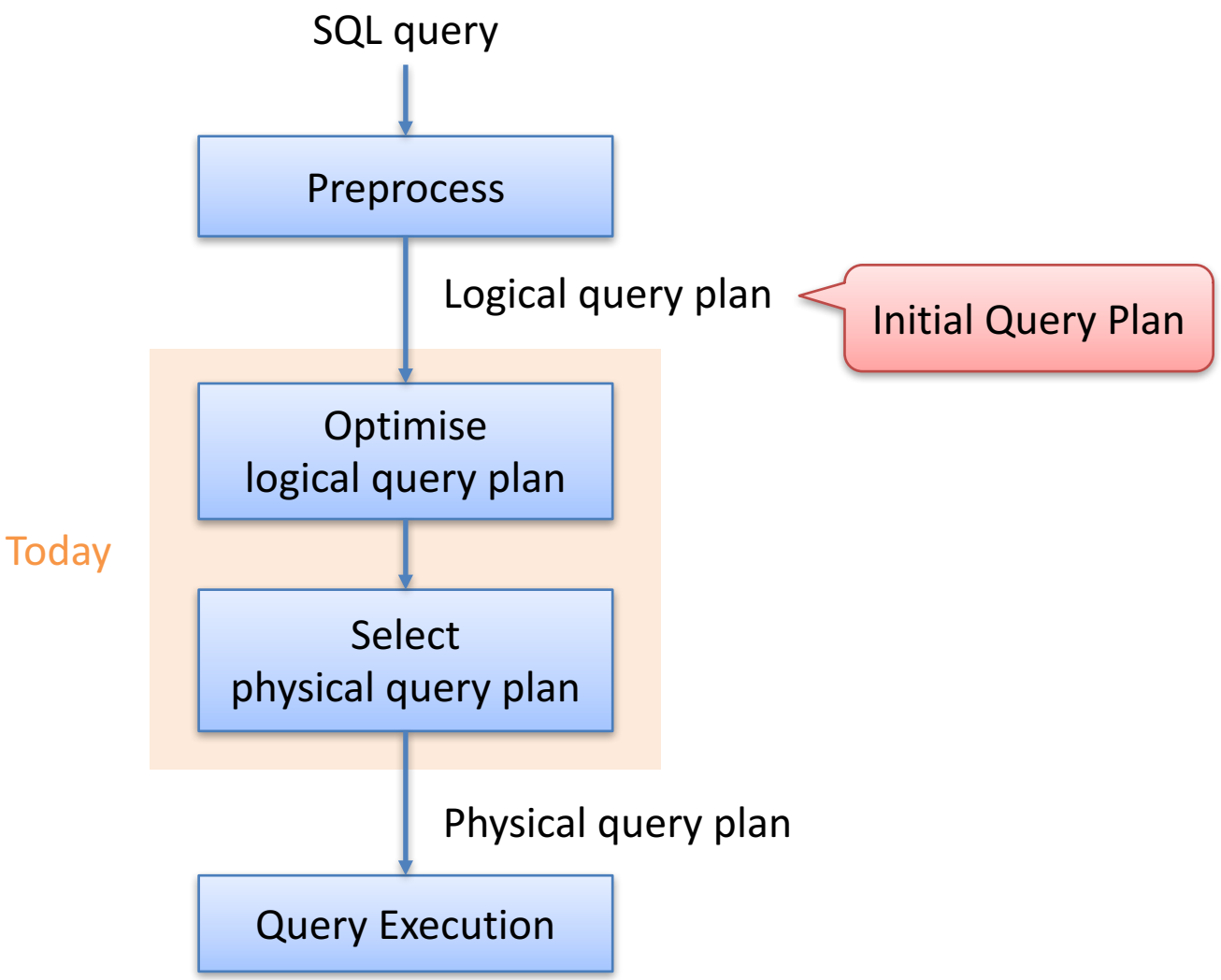
- Remain **balanced**
- **Huge capacity** even with height 3 if blocks large enough
  - Block size: 16386 bytes (16 kilobytes)
  - Values stored in index: 4 bytes
  - Pointers: 8 bytes
  - Largest n so that each B+ tree node fits into a block (i.e., $4n + 8(n+1) \leq 16386$) is 1364
  - B+ trees with height 3 can store > n^3 = 2537716544 values

Properties of B+ Tree Indexes

- Can be implemented **efficiently with respect to number of disk accesses**
  - Number of disk accesses typically ≈ 2 + height of B+ tree
- Most of the B+ tree can be kept in memory
  - Upper levels
  - Even with block size of 16384 bytes and n = 1364:
    - Level 1 (root) ≈ 16 KB
    - Level 2 (children of root) ≈ (n+1) × 16 KB ≈ 21 MB
    - Level 3 ≈ (n+1) × (n+1) × 16 KB ≈ 28 GB

Typically, these are the leaf nodes and can be loaded from disk on demand

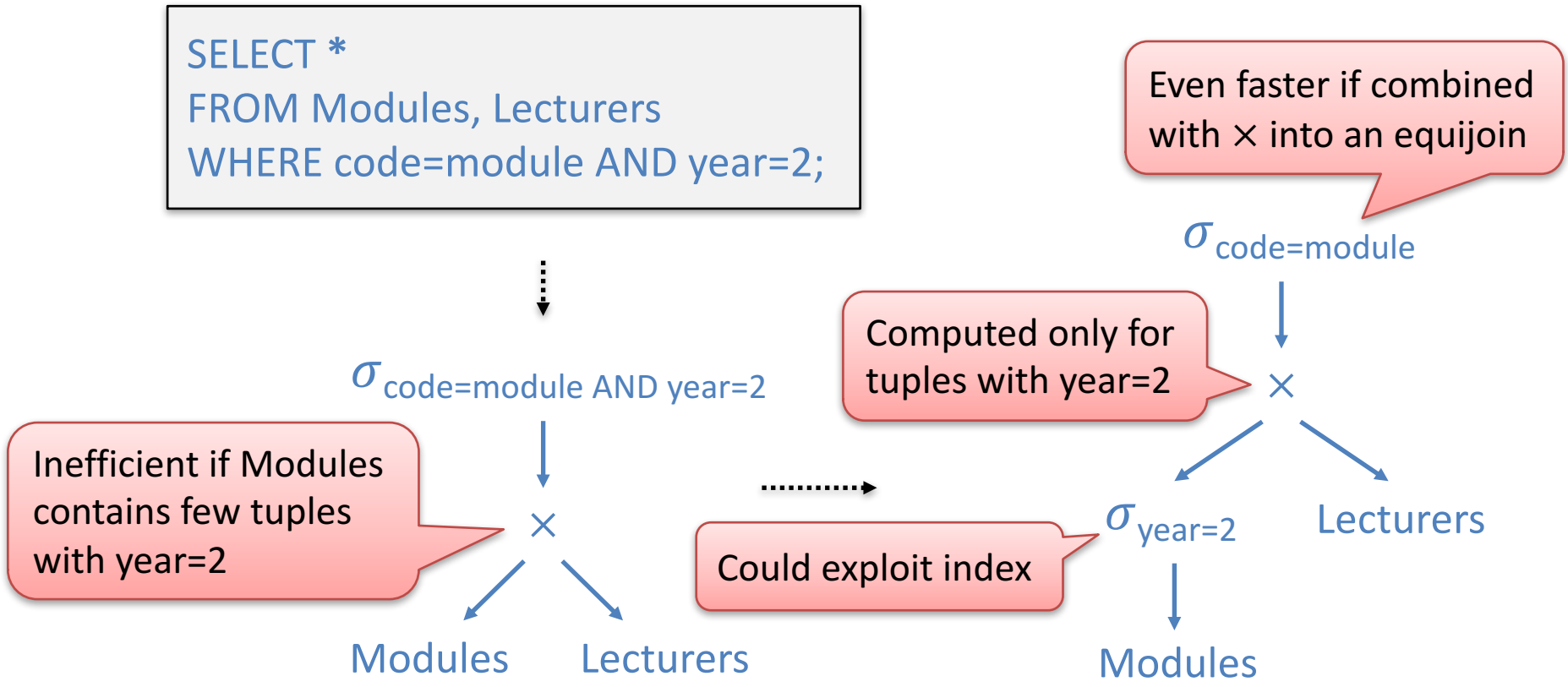
Final Bit…


SQL query
Preprocess
Logical query plan
Initial Query Plan
Optimise
logical query plan
Today
Select
physical query plan
Physical query plan
Query Execution

How to optimise the initial query plan?

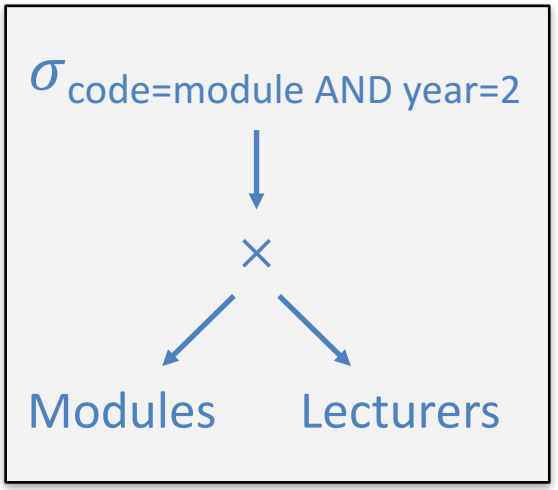
Why Optimise Query Plans?

- The initial query plan might not be the optimal one



How to Optimise?

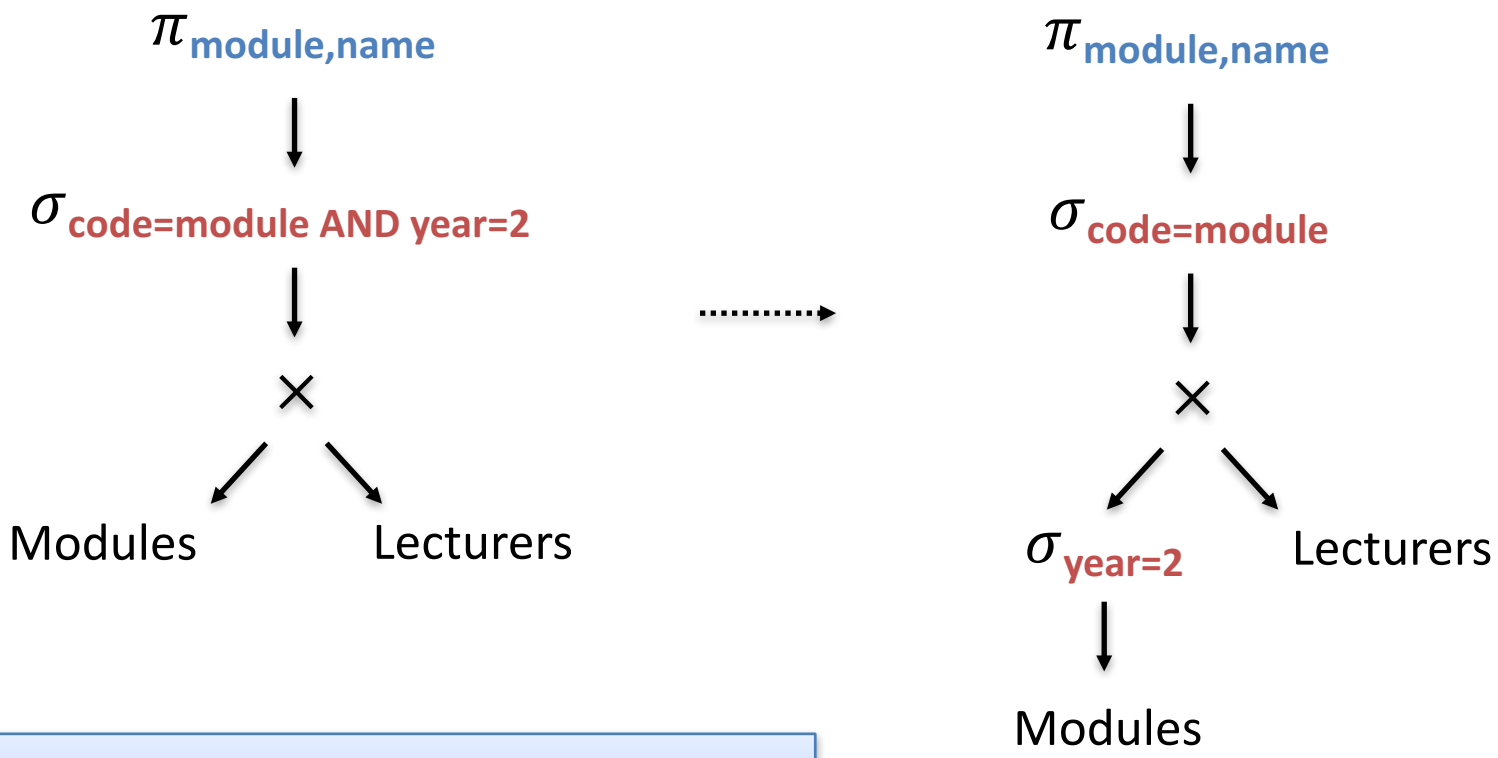
- Evaluation of query plans:
  - Bottom-up
  - Efficiency depends on size of intermediate results



- Rewrite the initial query plan so that intermediate results will be smaller
- Based on equivalence laws of relational algebra
  - Many laws (see references for this lecture on Vital)
  - Examples:
    - $\sigma_{A=a \text{ AND } B=b}(R) = \sigma_{A=a}(\sigma_{B=b}(R))$
    - $\sigma_{A=a}(R \times S) = \sigma_{A=a}(R) \times S$ — If A is on R
    - $\sigma_{A=B}(R \times S) = R \bowtie_{A=B} S$ — If A is on R and B is on S

Heuristics

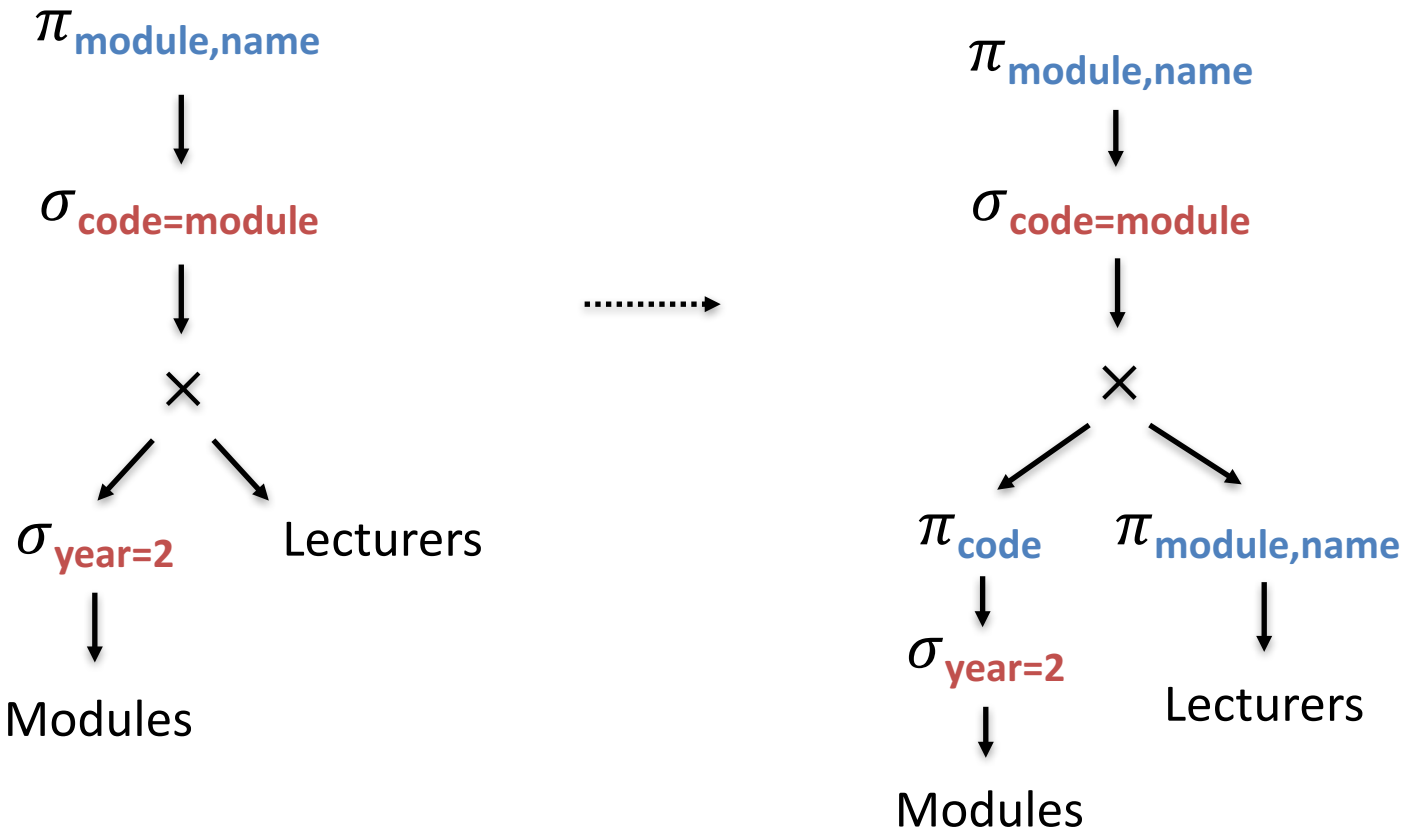
#1: Push selections as far down the tree as possible



Intuition: This gets rid of many irrelevant tuples very early during execution.

Heuristics

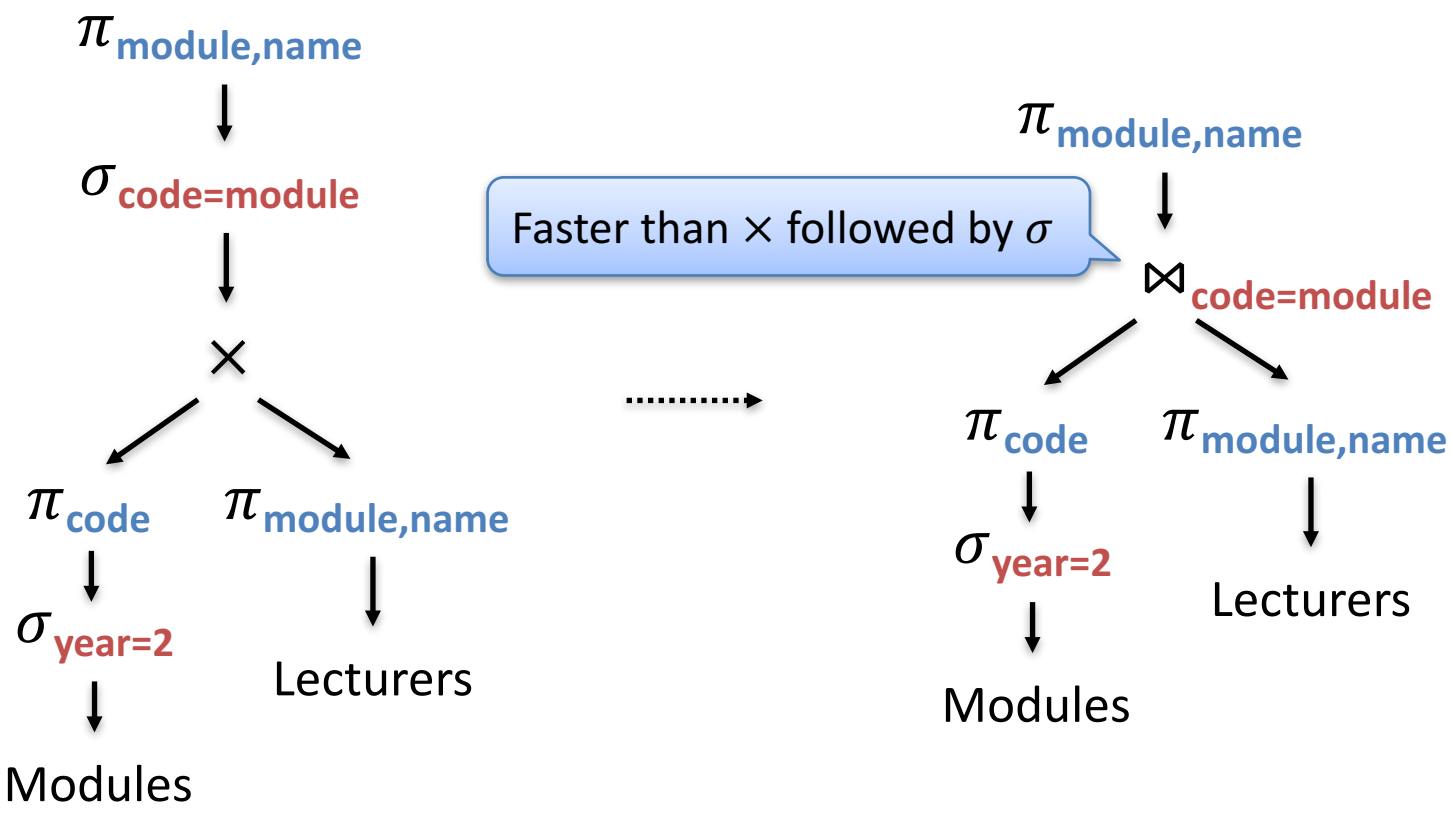
#2: "Push" projections as far down the tree as possible, or insert projections where appropriate



Heuristics

Many more heuristics…

#3: If possible, introduce equijoins for × followed by σ

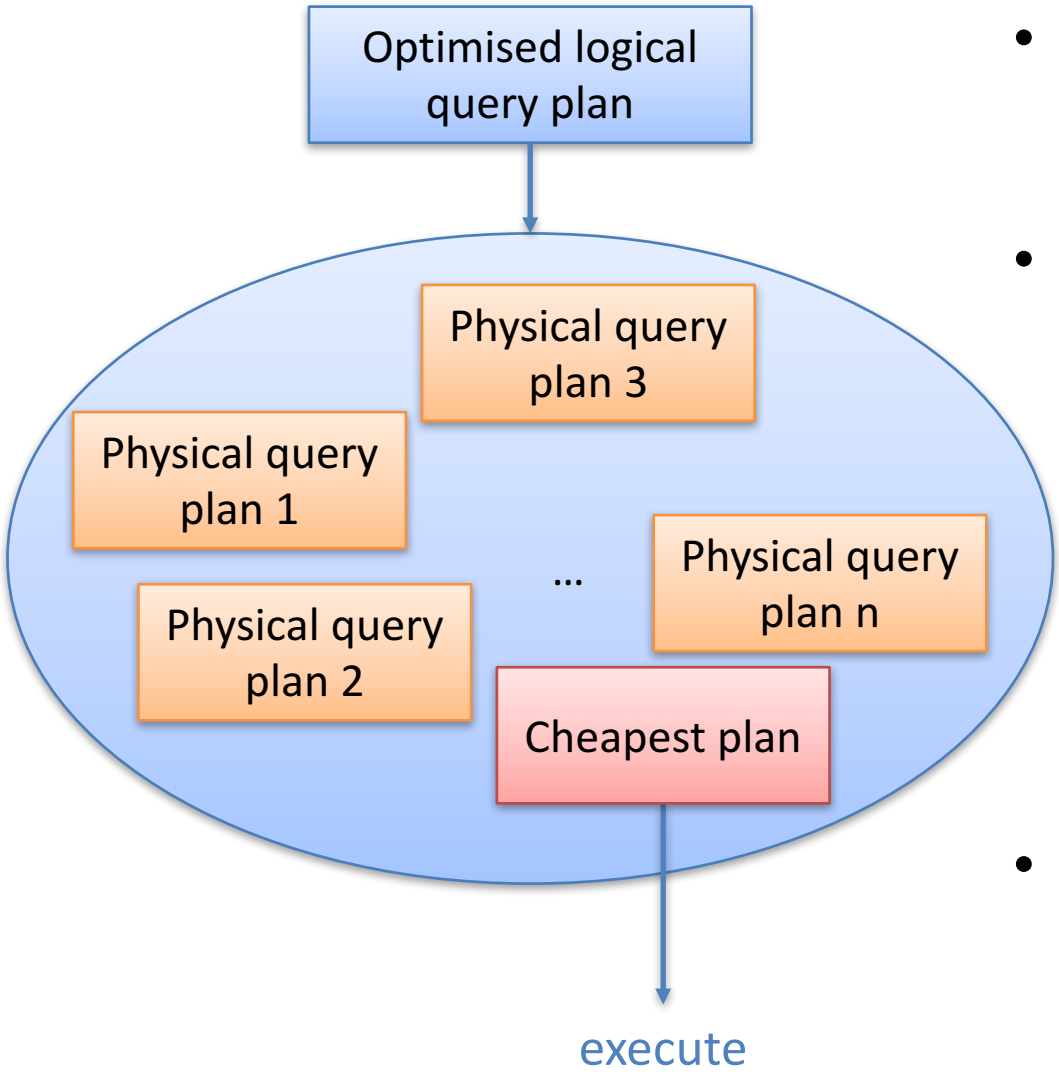


Recap

- So far, **optimisation of query plans at the logical level**
  - Started with initial query plan
  - Applied equivalence laws of relational algebra to optimise the plan
- A **physical query plan** adds information required to execute the optimised query plan
  - Which **algorithm** to use for execution of operators?
    - Naïve selection or selection with an index?
    - Nested Block Join or Sort Join or Hash Join etc.?
  - How to **pass information** from one operator to the other?
    - Write to disk, keep in memory, pipelining operators, etc.?
  - Good **order** for computing joins, unions, etc.?
  - Additional operations such as sorting

How to select a physical query plan?

From Logical Plans To Physical Plans



- **Generate** many different physical query plans
- **Estimate cost** of execution for each plan
  - Time
  - Disk accesses
  - Memory
  - Communication
  - …
- Select physical plan with **lowest cost estimate**

Estimating the Cost of Execution

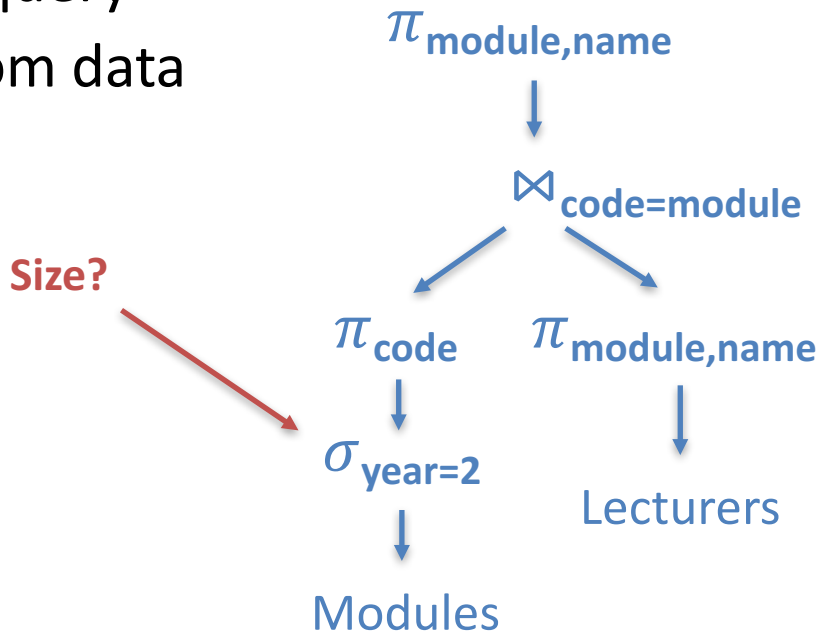
- Here: **number of disk access operations**
- Number of disk accesses influenced by many factors:
  - Selection of algorithms for the individual operators
  - Method for passing information
  - **Size of intermediate results**

One of the most critical factors

- Estimated from parameters of the database
  - Important paramters:
    - **Size of relations**
    - **Number of distinct items per attribute per relation**
  - Computed exactly from the database or are estimated ("statistics gathering")

Estimating Intermediate Result Sizes

- One of the most challenging tasks of a DBMS
  - Difficult even for nodes close to the leaves
  - Cannot afford executing the query
  - Rely on statistics gathered from data



- Many different approaches
  - Some easier than others
  - With join size estimation, we enter active research terrain…

Estimating the Size of a Selection

- Estimate for the size of $\sigma_{A=a}(R)$:

Recall: |R| = number of tuples in R

$$\frac{|R|}{\text{number of distinct values in column } A \text{ of relation } R}$$

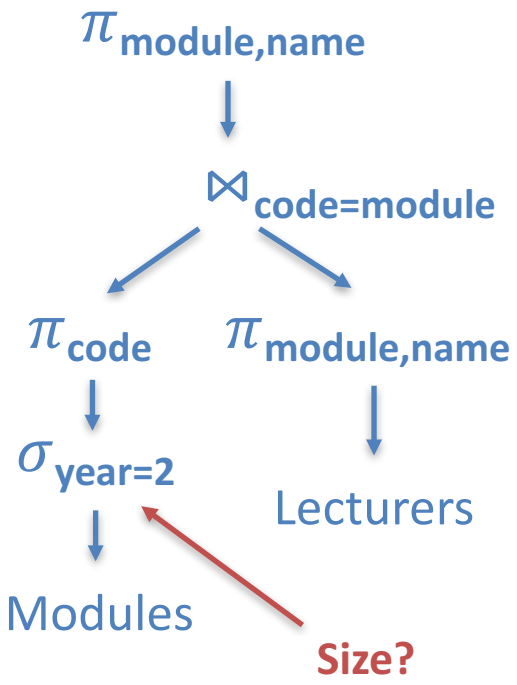
- Estimate for the # of blocks required to store $\sigma_{A=a}(R)$:

$$\frac{\text{number of blocks for } R}{\text{number of distinct values in column } A \text{ of relation } R}$$

- Good if values in column **A** of **R** occur equally often, but can be bad

Example

- Assume:
  - Modules contains **80 tuples**, stored in **20 blocks**
  - There are exactly **4 distinct values** for the year attribute of Modules
- Estimate for size of $\sigma_{year=2}$(Modules): 80/4 = 20 tuples
- Estimate for number of blocks that are required to store $\sigma_{year=2}$(Modules): 20/4 = 5 blocks



Joins

- How to estimate R ⋈ S?

Assume A is the only common attribute.

- Simple estimate based on size of R & S and number of distinct values in common attributes

$$\frac{|R| \times |S|}{\text{max. number of distinct values for } A \text{ in } R \text{ or } S}$$

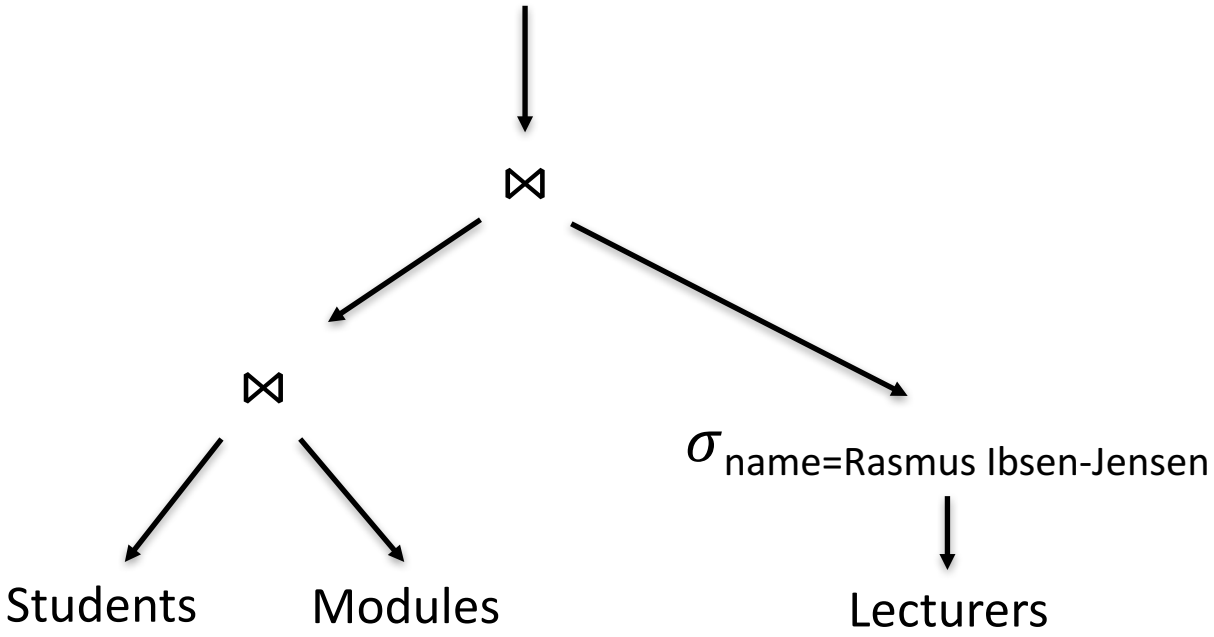
- As for selection, based on assumptions that might not always lead to good estimates
- More sophisticated methods:
  - Many and still a topic of active research
  - See, e.g., SIGMOD/PODS/VLDB conferences

Other Issues

- **How to generate** physical query plans?
  - Explore all?
  - More sensible approaches: top-down/bottom-up
- Selection of a **suitable algorithm** for each operator
  - based on size of intermediate result
- Selection of a **good join order**
  - also based on size of intermediate results
- How to **pass information** from one operator to another?

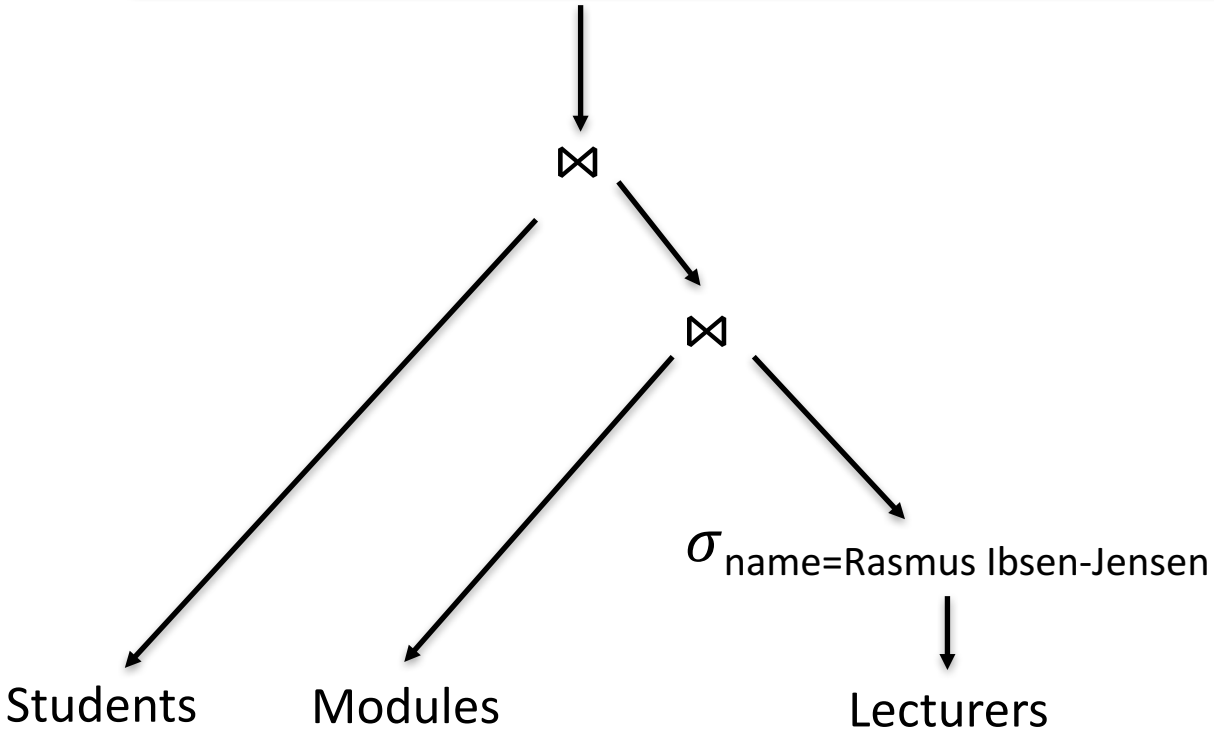
Example where join order matters

```
SELECT *
FROM Lecturers NATURAL JOIN Modules NATURAL JOIN Students
WHERE Lecturers.name = Rasmus Ibsen-Jensen
```



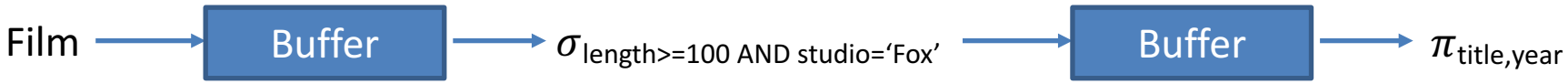
Example where join order matters

```
SELECT *
FROM Lecturers NATURAL JOIN Modules NATURAL JOIN Students
WHERE Lecturers.name = Rasmus Ibsen-Jensen
```

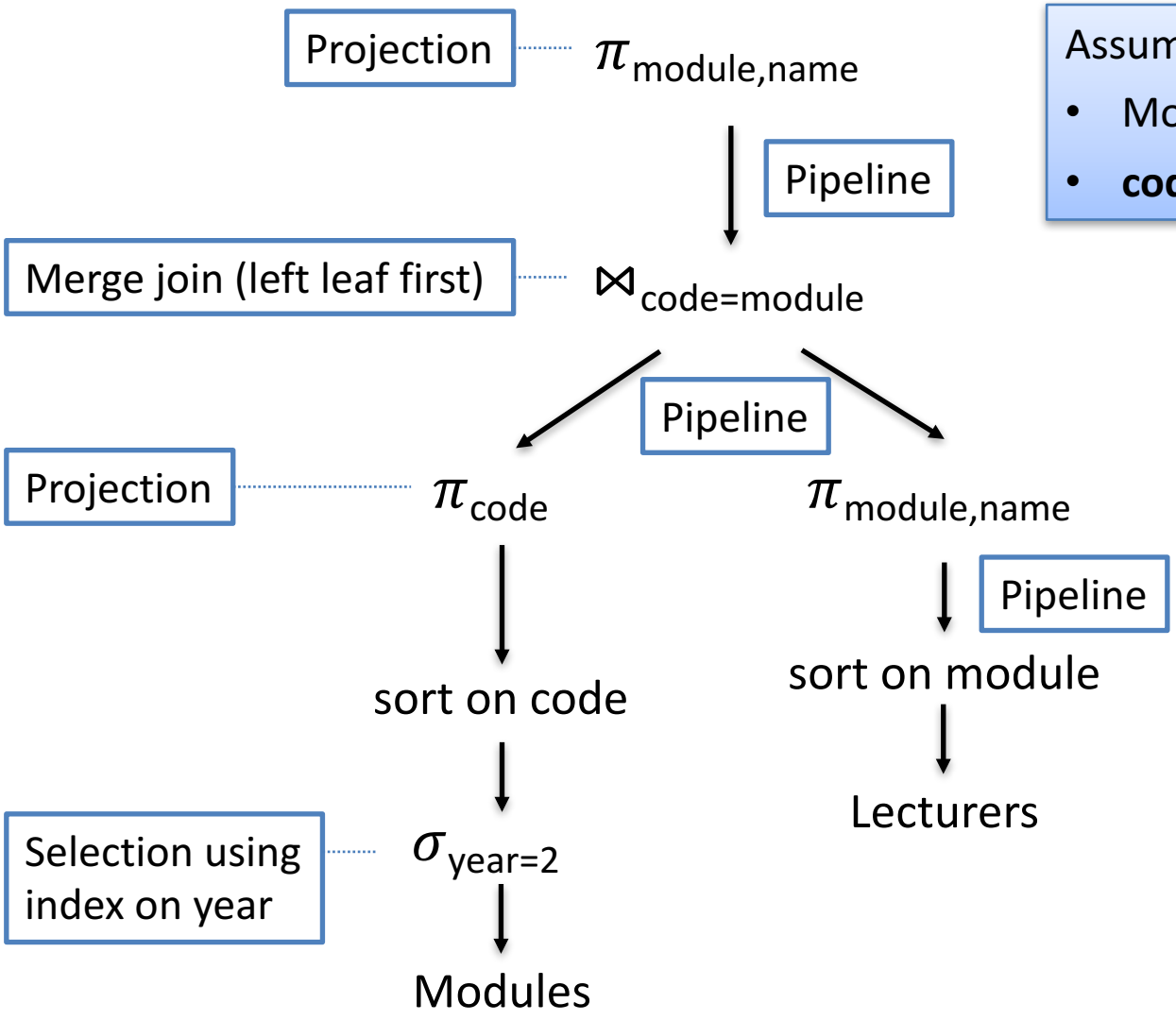


Passing Information

- Materialisation: write intermediate results to disk
- Pipelining ("stream-based processing")
  - Passes the tuples of one operation directly to the next operation without using disk
  - Extra buffer for each pair of adjacent operations to hold tuples passing from one relation to the other
  - Example:
    - $\pi_{\text{title,year}}(\sigma_{\text{length>=100 AND studio='Fox'}}(\text{Film}))$
    - With pipelining, the intermediate result of the selection will be written into a buffer in memory, from which the projection operator will read and process these tuples directly



A Physical Query Plan

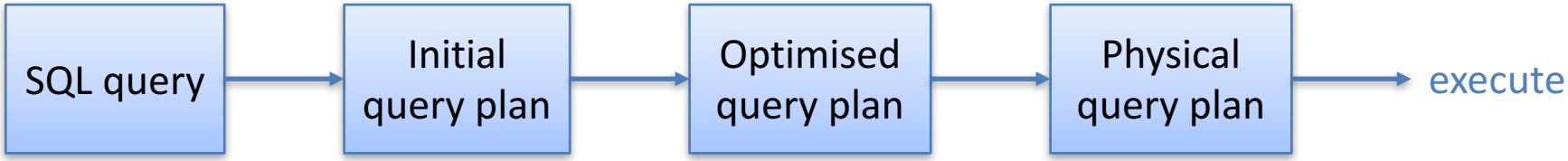


Assumption:

- Modules has index on **year**
- **code** is a key of Modules

Summary

- Query processing is the main task of DBMS
- Process:



- Main work: constructing the physical query plan
  - Heart of a DBMS
  - Has to select a good order of executing operators in query plan, algorithms for executing these operators, compute estimates on the size of intermediate results, etc.