Big Data Infrastructures

Philippe Cudré-Mauroux

Fall 2018

Lecture 2 - Storage and Indexing

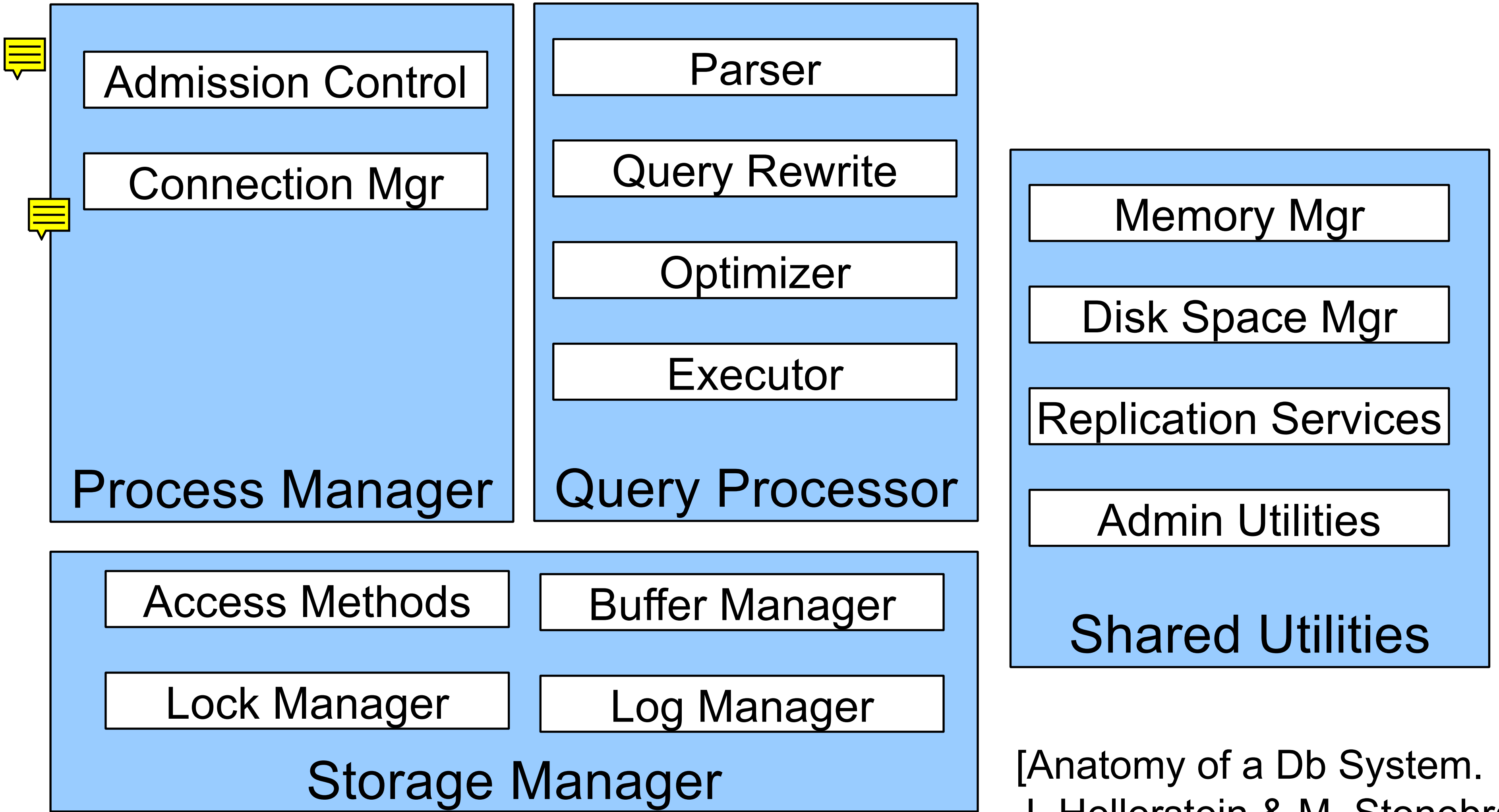
Outline

- **Case-Study**
- **Data storage**
  - Disk and files
  - Operations on files
- **Indices**
  - Index structures
  - Hash-based indices
  - B+ trees

Why is this important?

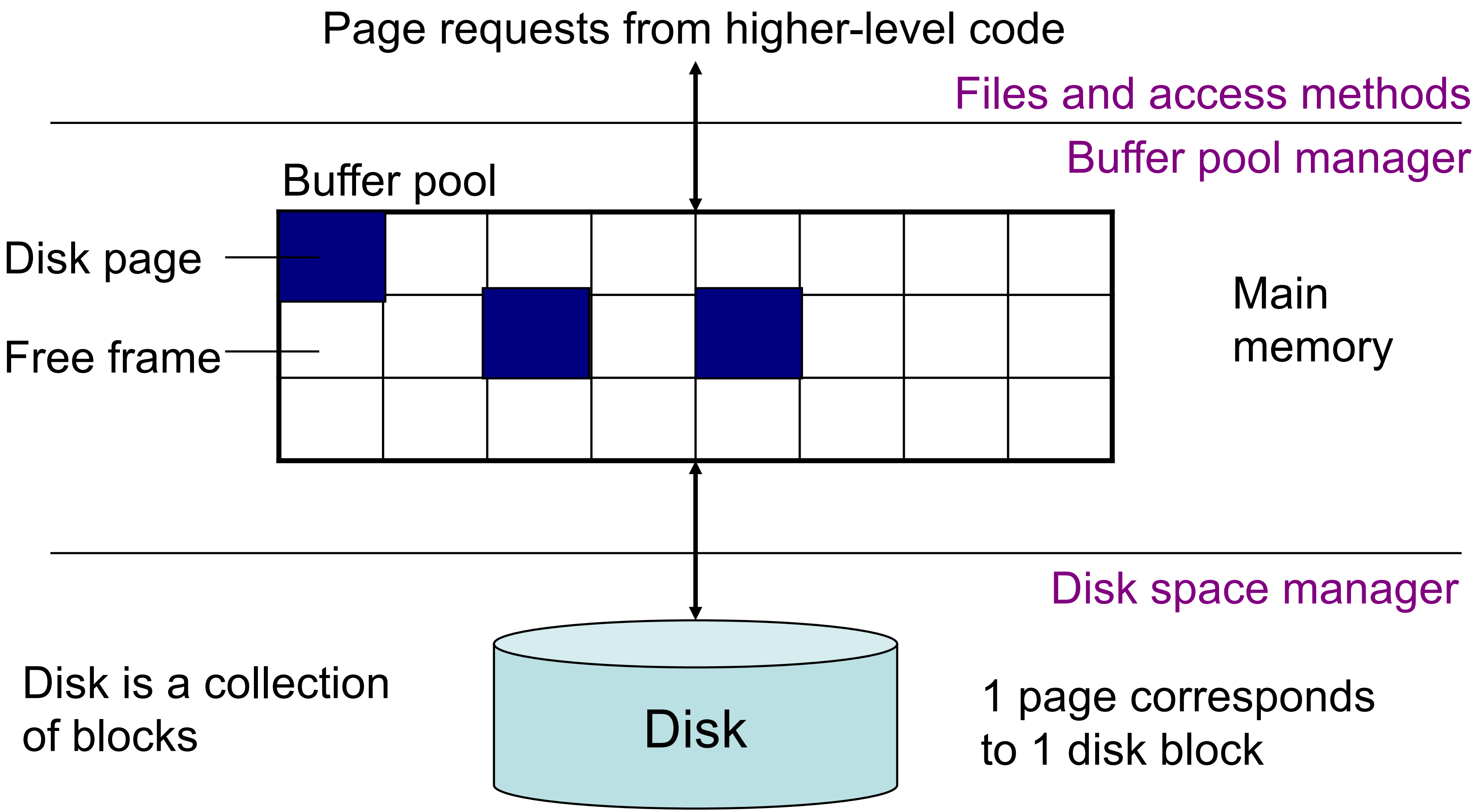
- **Data storage**
- **Indexes**

DBMS Architecture



[Anatomy of a Db System.
J. Hellerstein & M. Stonebraker.
Red Book. 4ed.]

Buffer Manager



Data Storage

- Basic **abstraction**
  - *Collection of records* or *file*
  - Typically, 1 relation = 1 file
  - A file consists of one or more *pages*

- How to organize pages into files?
- How to organize records inside a file?

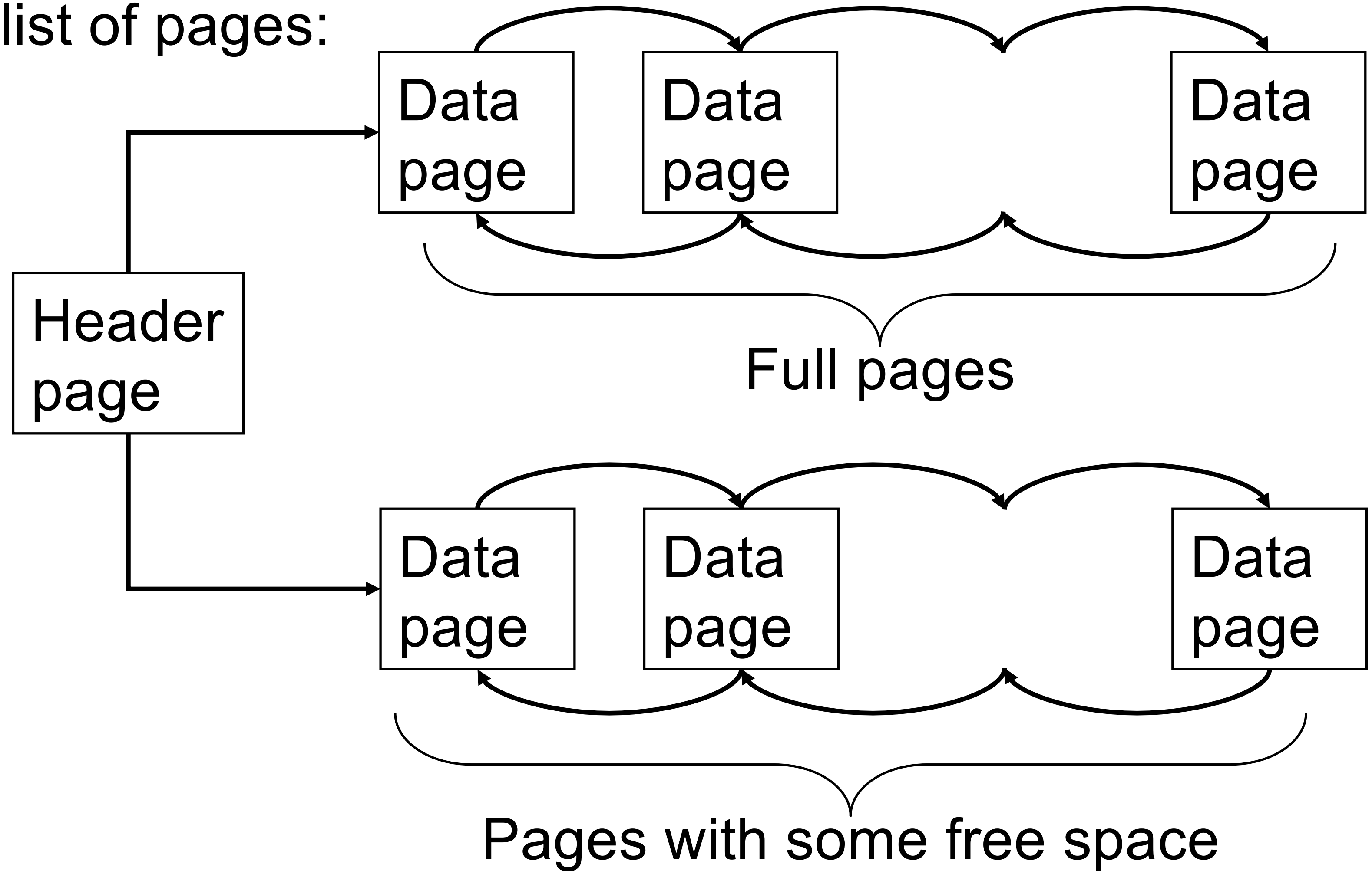
- Simplest approach: **heap file** (unordered)
  - Further approaches: clustered file or sorted file

Heap File Operations

- **Create** or **destroy** a file
- **Insert** a record
- **Delete** a record with a given record id (rid)
  - rid: unique tuple identifier
  - can identify disk address of page containing record by using rid
- **Get** a record with a given rid
- **Scan** all records in the file

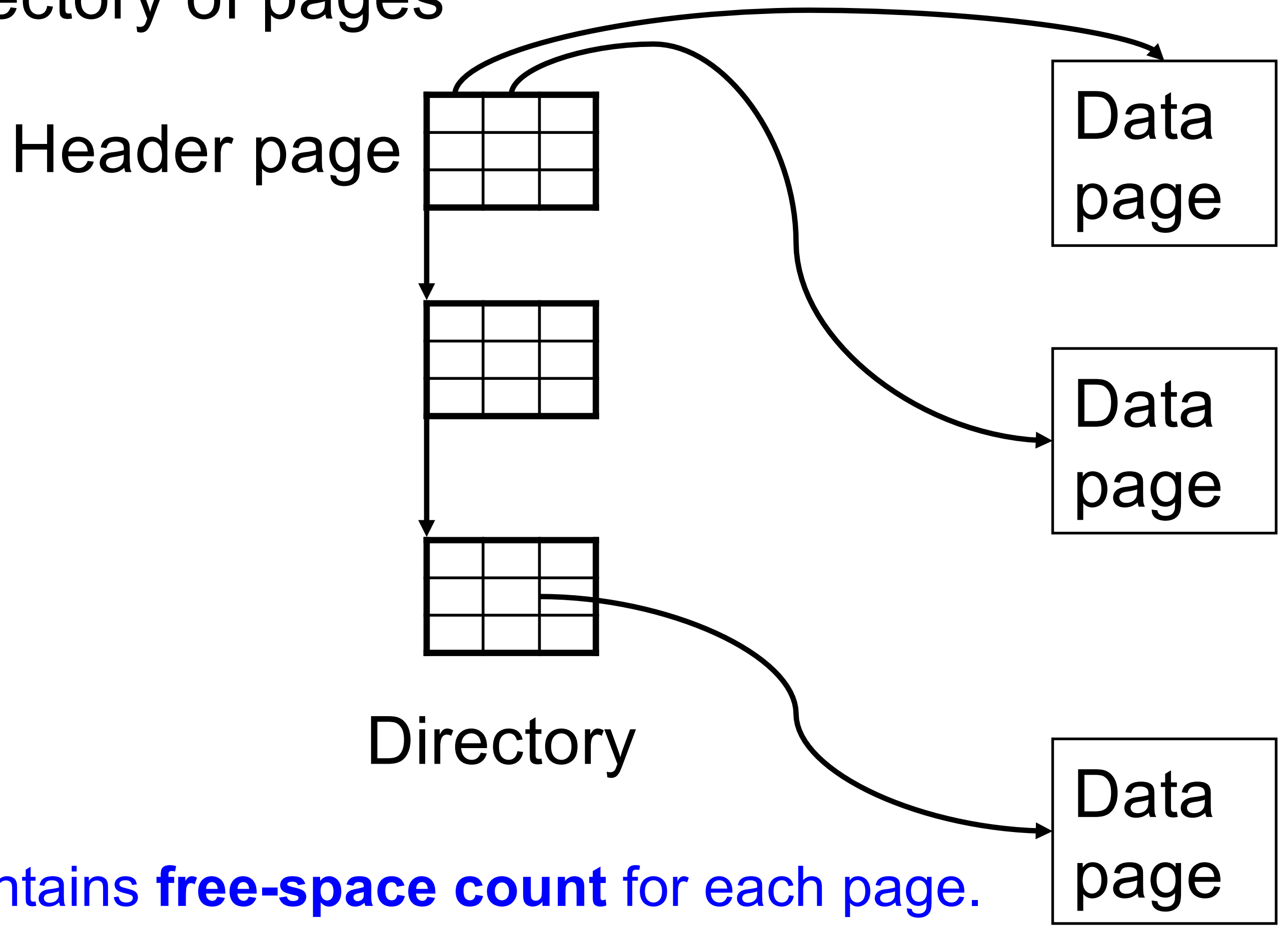
Heap File Implementation 1

Linked list of pages:



Heap File Implementation 2

Better: directory of pages



Directory contains **free-space count** for each page.
Faster inserts for variable-length records

Page Formats

Issues to consider

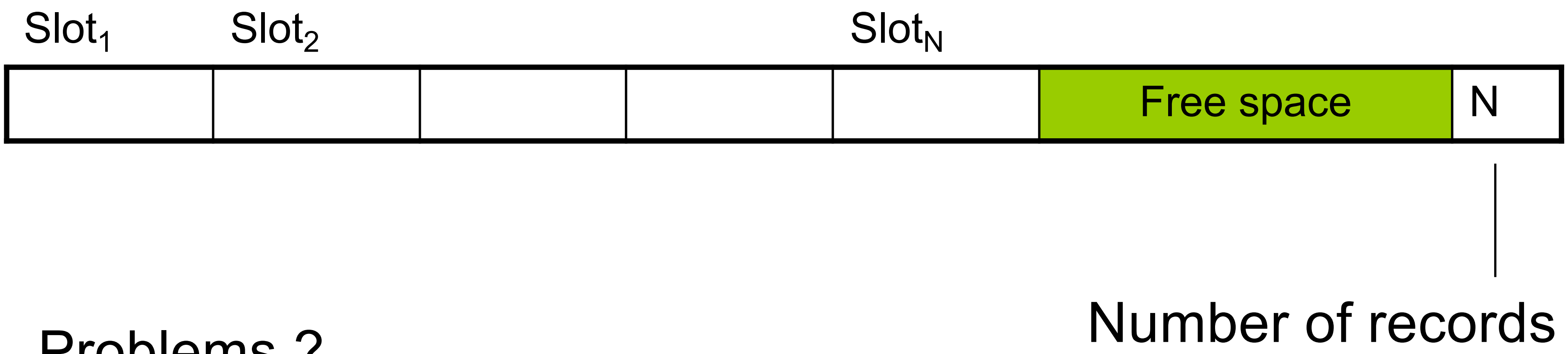
- 1 page = 1 disk block = fixed size (e.g. 8KB)
- Records:
  - Fixed length
  - Variable length
- Record id = RID
  - For example RID = (PageID, SlotNumber)

Why do we need RIDs in a relational DBMS?

Is the RID typically known to the end-user?

Page Format Approach 1

Fixed-length records: packed representation

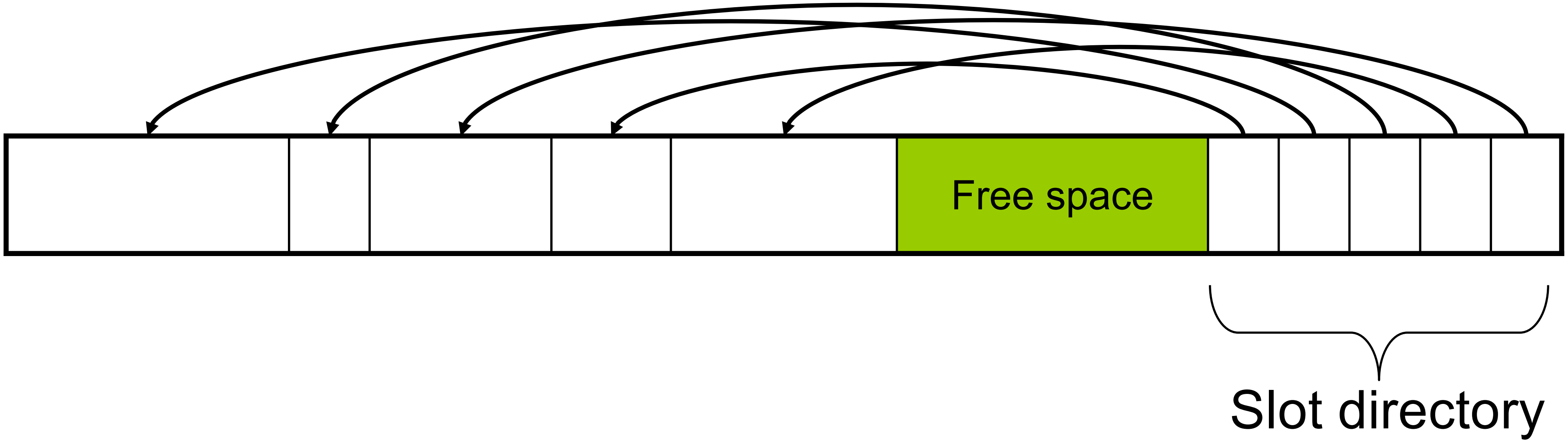


Problems ?

How to handle variable-length records?

Need to move records for each deletion, changing RIDs

Page Format Approach 2



Each slot contains
<record offset, record length>

Can handle variable-length records
Can move tuples inside a page without changing RIDs

Record Formats

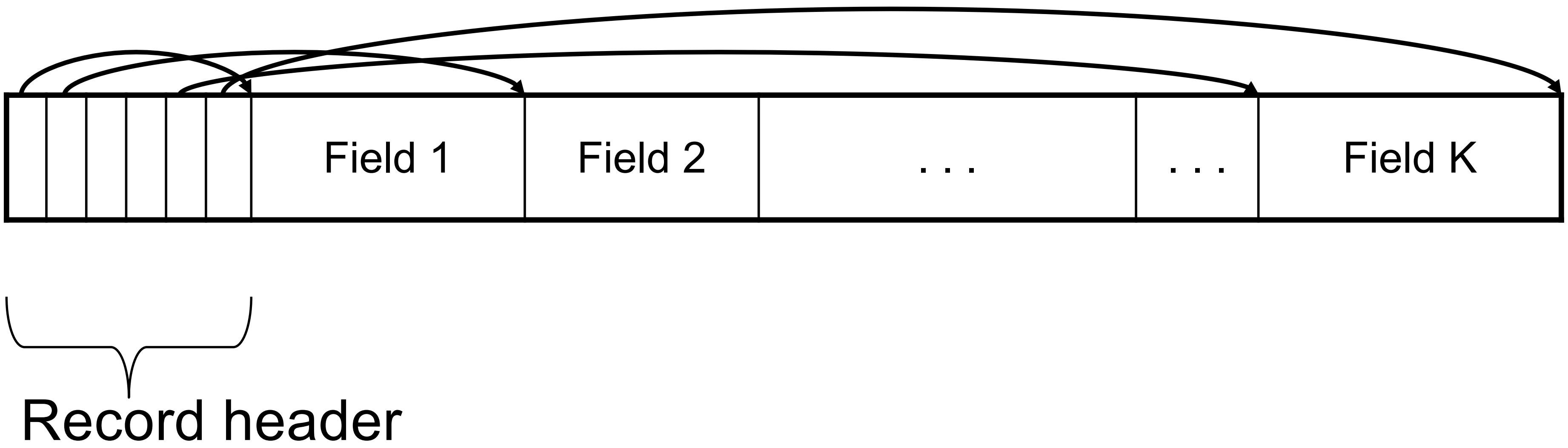
Fixed-length records → Each field has a fixed length (i.e., it has the same length in all the records)

Field 1	Field 2	. . .	. . .	Field K

Information about field lengths and types is in the catalog

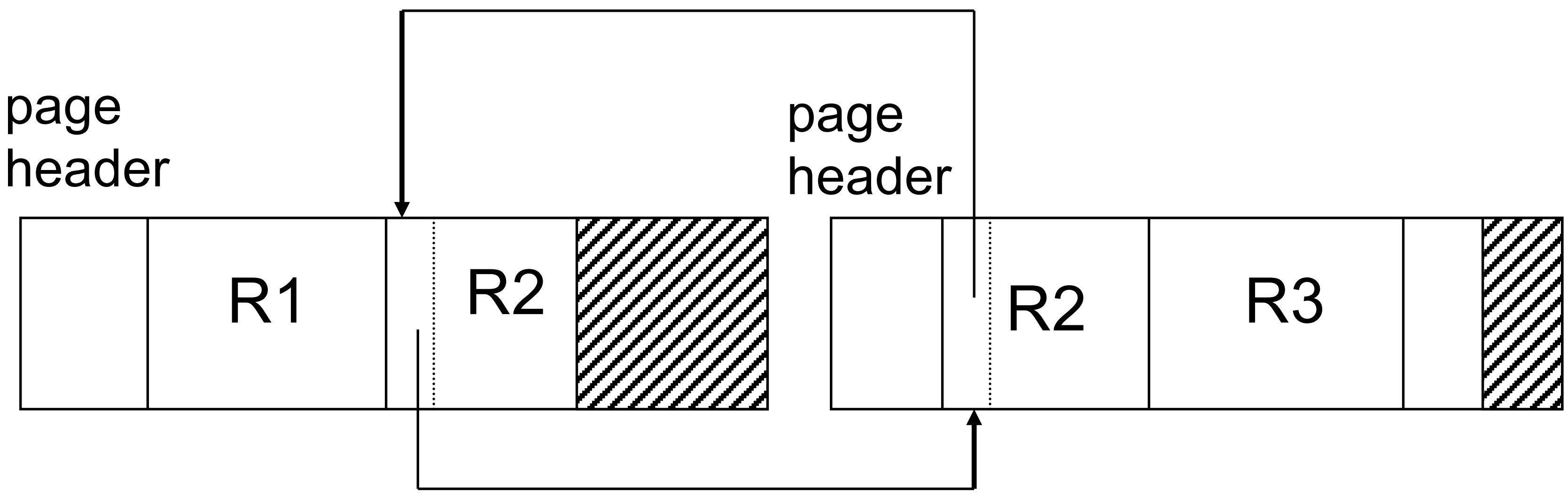
Record Formats

Variable length records



Remark: NULLS require no space at all (why ?)

Long Records Across Pages



- When records are very large
- Or even medium size: saves space in blocks
- Commercial RDBMSs avoid this

LOB

- Large objects
  - Binary large object: BLOB
  - Character large object: CLOB

- Supported by modern database systems
- E.g. images, sounds, texts, etc.

- Storage: attempt to cluster blocks together

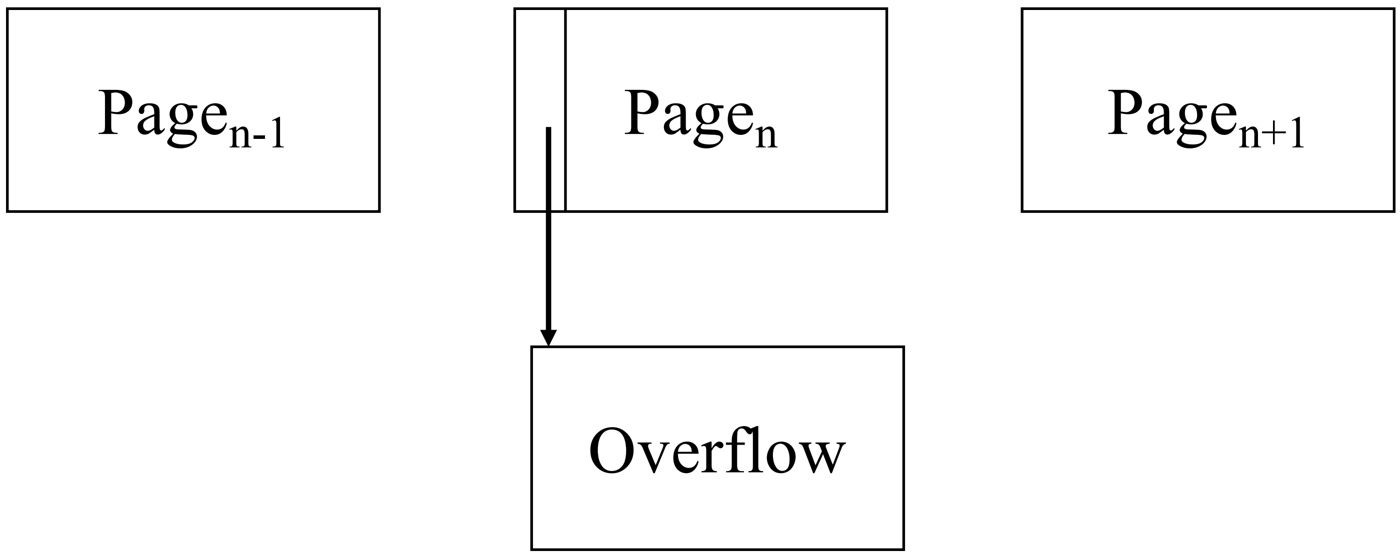
Outline

- **Data storage**
  - Disk and files
  - Operations on files
- **Indexes**
  - Index structures
  - Hash-based indexes
  - B+ trees

Modifications: Insertion

- File is unsorted (= ***heap file***)
  - add it wherever there is space (easy ☺)
- File is sorted
  - Is there space on the right page ?
    - Yes: we are lucky, store it there
  - Is there space in a neighboring page ?
    - Look 1-2 pages to the left/right, shift records
  - If anything else fails, create ***overflow page***

Overflow Pages



- After a while the file starts being dominated by overflow pages: time to reorganize

Modifications: Deletions

- Free space in page, shift records
  - Be careful with slots
  - RIDs for remaining tuples must NOT change
- May be able to eliminate an overflow page

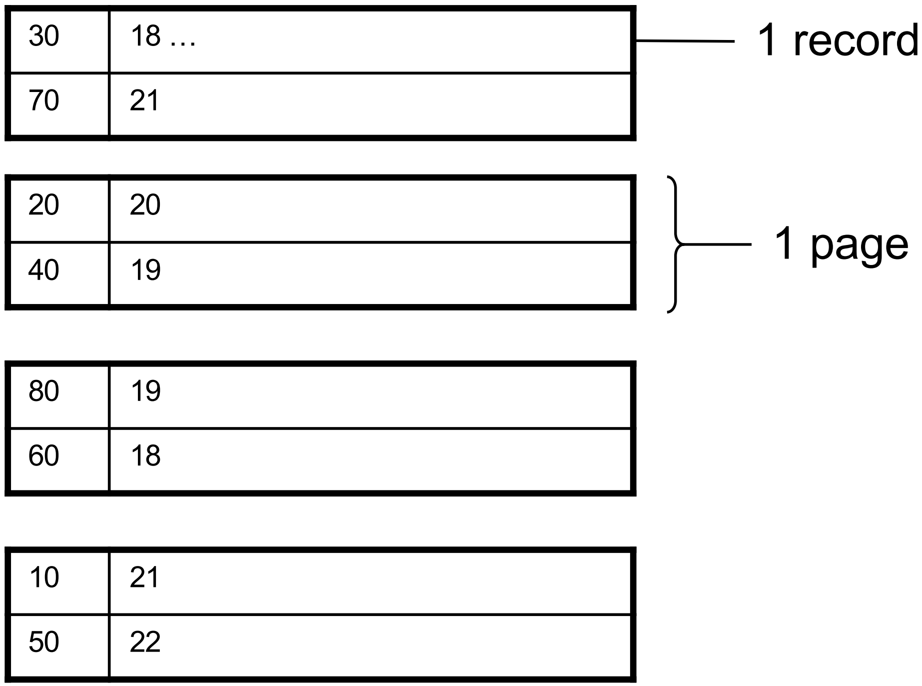
Modifications: Updates

- If new record is shorter than previous, easy ☺
- If it is longer, need to shift records
  - May have to create overflow pages

Searching in a Heap File

File is not sorted on any attribute

`Student(sid: int, age: int, …)`



Heap File Search Example

- 10,000 students
- 10 student records per page
- Total number of pages: 1,000 pages
- Find student whose sid is 80
  - Must read on average 500 pages
- Find all students older than 20
  - Must read all 1,000 pages
- Can we do better?

Sequential File

File sorted on an attribute, usually on primary key

`Student(sid: int, age: int, …)`

10	21 …
20	20

30	18
40	19

50	22
60	18

70	21
80	19

Sequential File Example

- Total number of pages: 1,000 pages
- Find student whose sid is 80
  – Could do binary search, read $\log_2(1{,}000) \approx 10$ pages
- Find all students older than 20
  – Must still read all 1,000 pages
- Can we do even better?

Outline

- **Data storage**
  - Disk and files
  - Operations on files
- **Indexes**
  - Index structures
  - Hash-based indexes
  - B+ trees

Indexes

- **Index:** data structure that organizes data records on disk to optimize selections on the ***search key fields*** for the index

- An index contains a collection of ***data entries***, and supports efficient retrieval of all data entries with a given search key value **k**

Index Classification Overview

- Primary/secondary
  – Primary = determines the location of indexed records
  – Secondary = cannot reorder data, does not determine data location
- Dense/sparse
  – Dense = every key in the data appears in the index
  – Sparse = the index contains only some keys
- Clustered/unclustered
  – Clustered = records close in index are close in data
  – Unclustered = records close in index may be far in data
- B+ tree / Hash table / …

Indexes

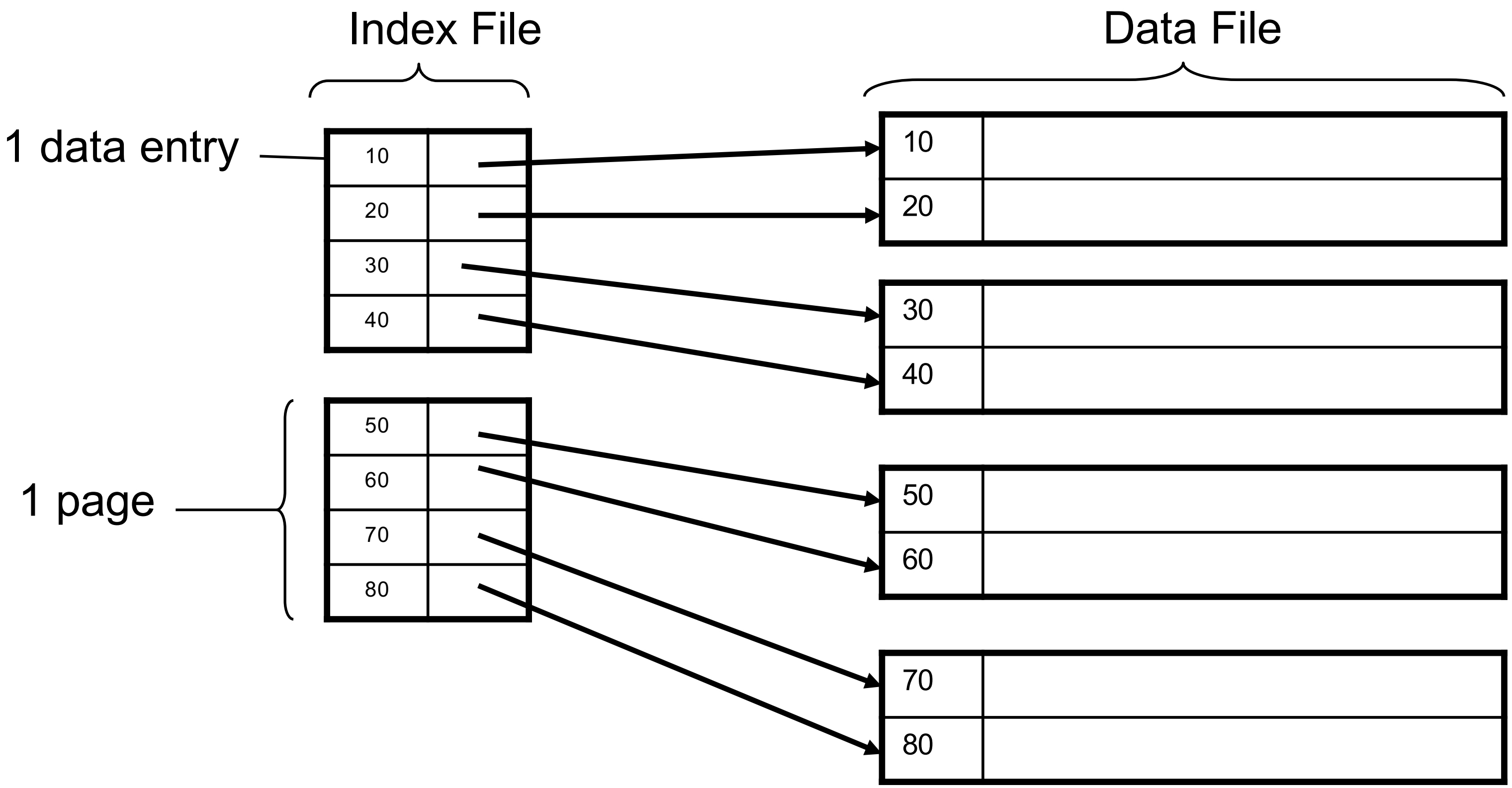
- **Search key** = can be any set of fields
  - not the same as the primary key, nor a key

- **Index** = collection of data entries

- **Data entry** for key k can be:
  - The actual record with key k
    - In this case, **the index is also a special file organization**
    - This type of index is also called the **primary index** of a file
  - (k, RID)
  - (k, list-of-RIDs)

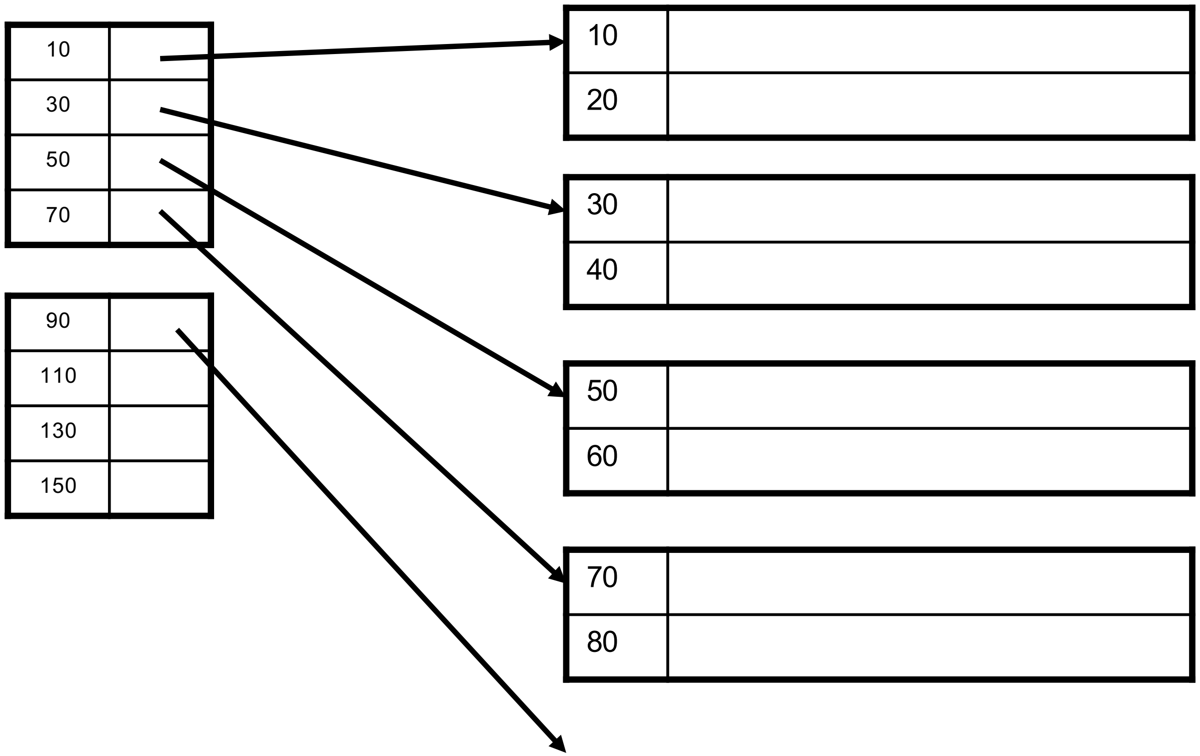
Primary Index

- Index determines the location of indexed records
- *Dense* index: sequence of (key,pointer) pairs



Primary Index

- *Sparse* index



Primary Index Example

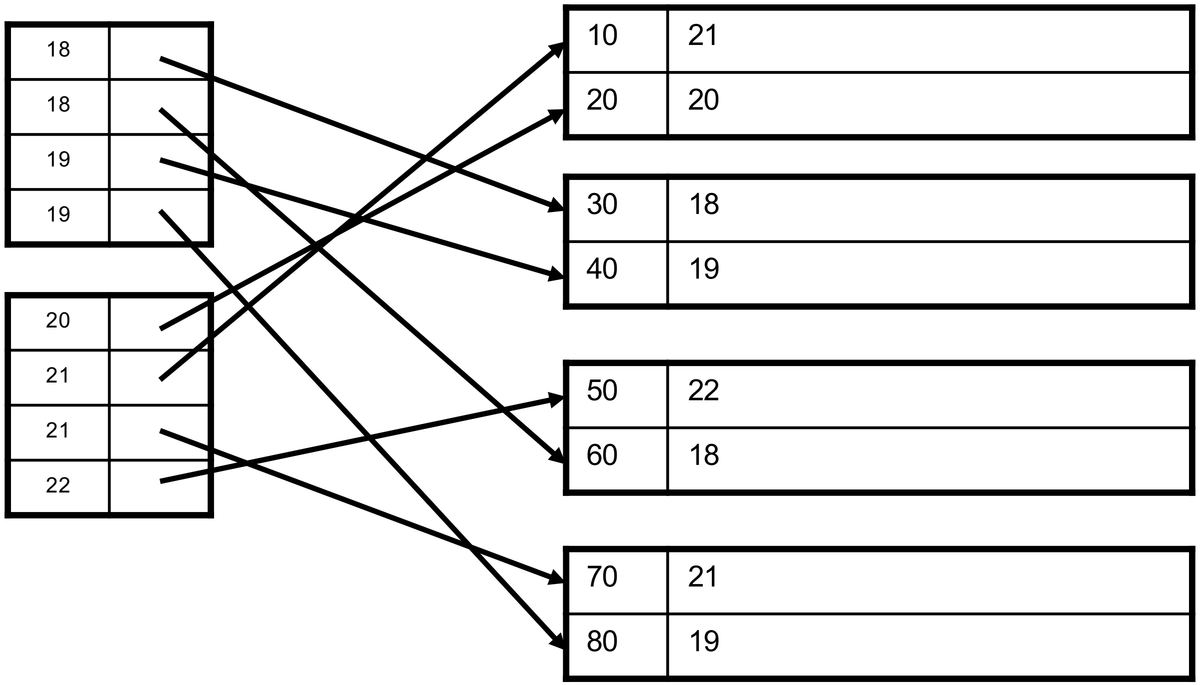
- Let's assume all pages of index fit in memory

- Find student whose sid is 80
  - Index (dense or sparse) points directly to the page
  - Only need to read 1 page from disk.
- Find all students older than 20
  - Must still read all 1,000 pages.

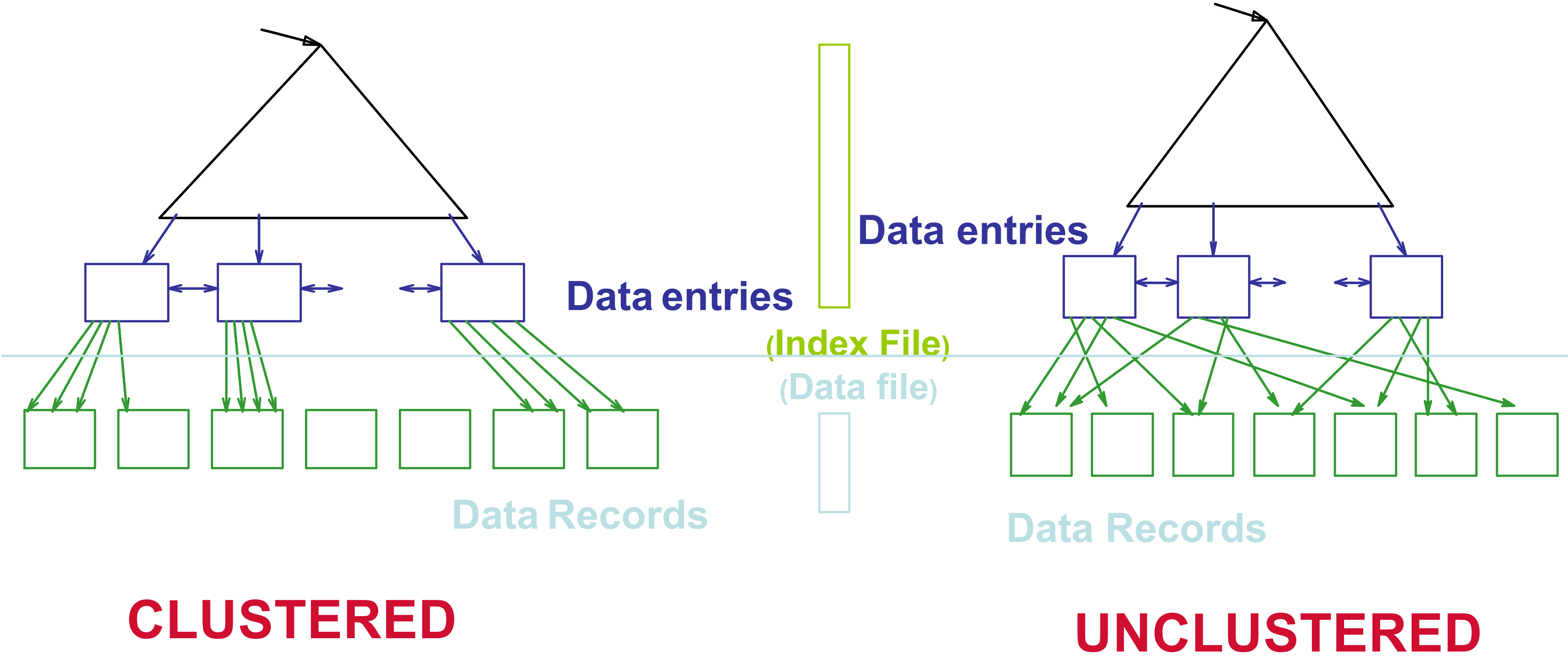
- How can we make *both* queries fast?

Secondary Indexes

- To index other attributes than primary key
- Always dense (why ?)



Clustered vs. Unclustered Index



Clustered = records close in index are close in data

Clustered/Unclustered

- Primary index = clustered by definition
- Secondary indexes = usually unclustered

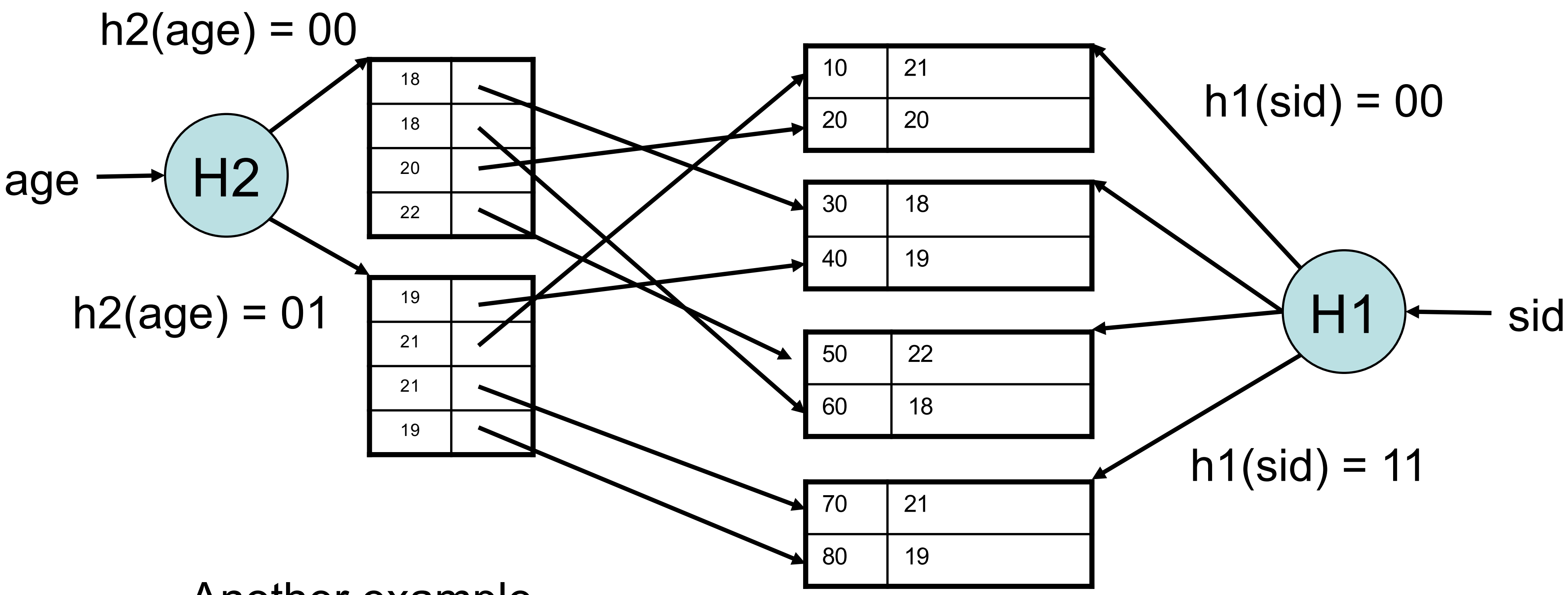
Large Indexes

- What if index does not fit in memory?

- Would like to index the index itself
  - Hash-based index
  - Tree-based index

Hash-Based Index

Good for point queries but not range queries



Another example
of secondary index

Another example of primary index

Tree-Based Index

- How many index levels do we need?
- Can we create them automatically? Yes!
- Can do something even more powerful!

B+ Trees

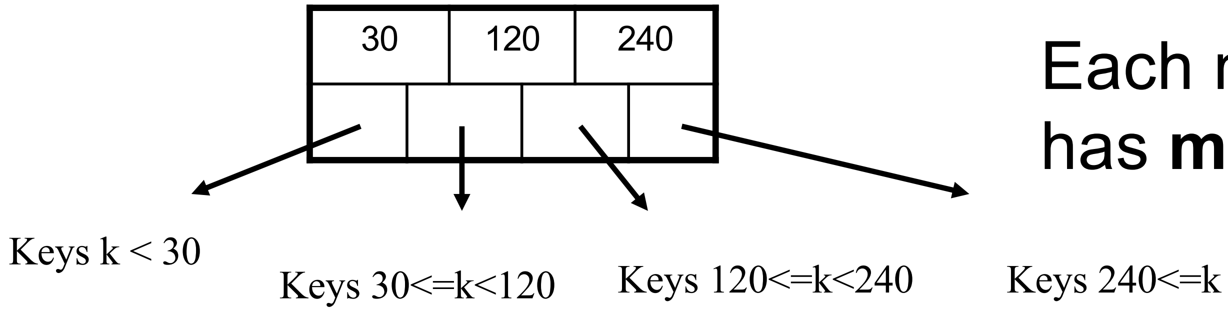
- Search trees

- Idea in B Trees
  - Make 1 node = 1 page (= 1 block)
  - Keep tree balanced in height

- Idea in B+ Trees
  - Make leaves into a linked list : facilitates range queries

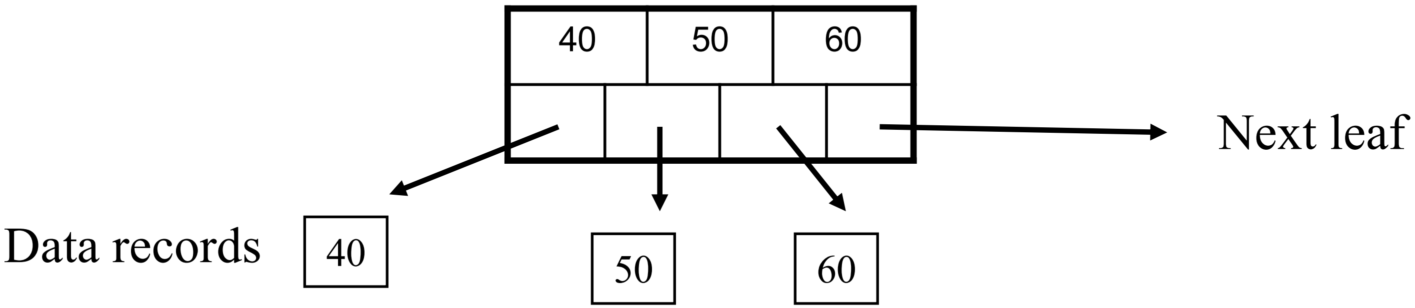
B+ Trees Basics

- Parameter d = the *degree*
- Each node has **d <= m <= 2d keys** (except root)



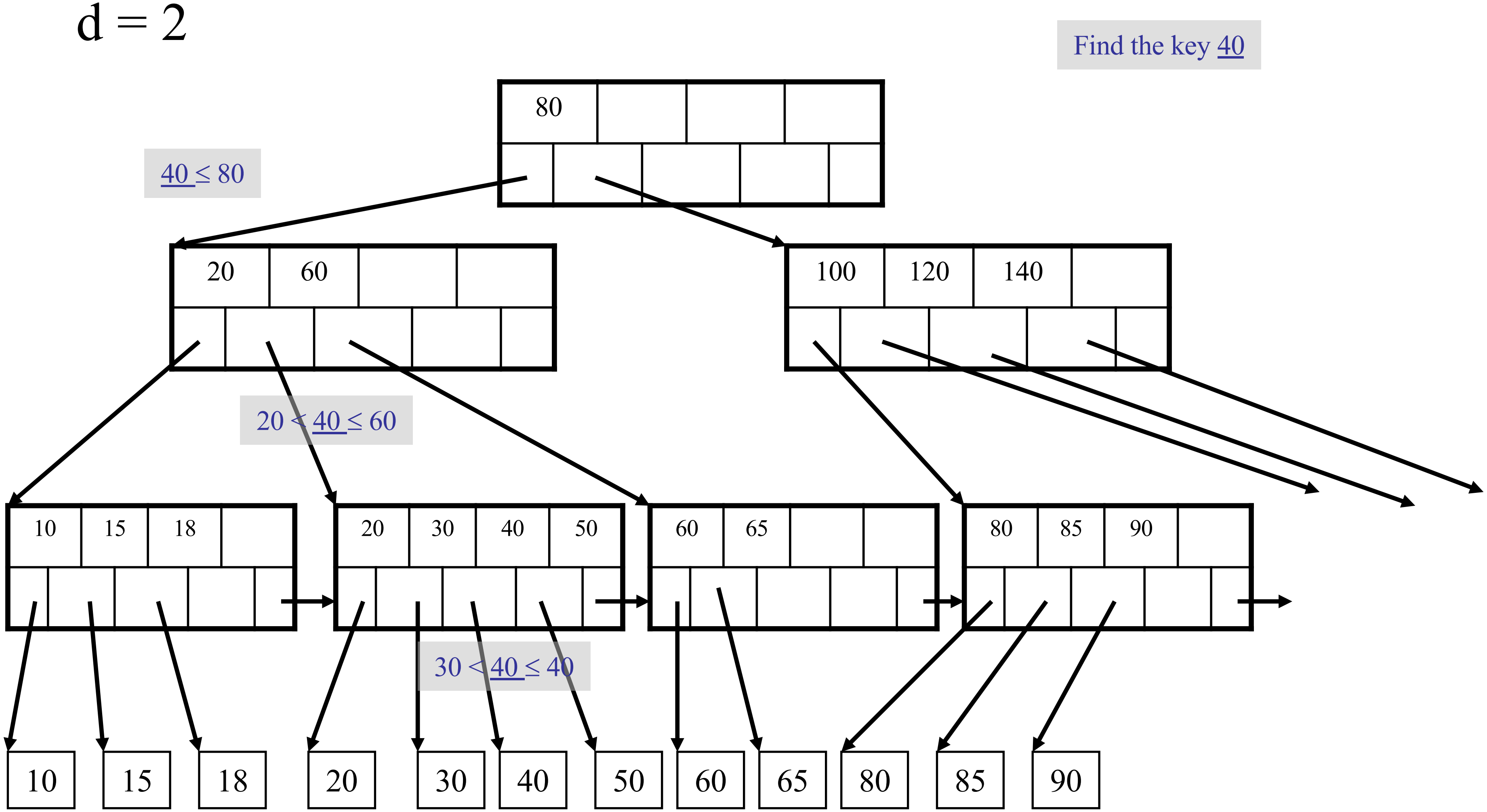
Each node also has **m+1 pointers**

- Each leaf has **d <= m <= 2d keys**:



B+ Tree Example

d = 2



Searching a B+ Tree

- Exact key values:
  - Start at the root
  - Proceed down, to the leaf

- Range queries:
  - Find lowest bound as above
  - Then sequential traversal

```
Select name
From Student
Where age = 25
```

```
Select name
From Student
Where 20 <= age
  and  age <= 30
```

B+ Tree Design

- How large d ?
- Example:
  - Key size = 4 bytes
  - Pointer size = 8 bytes
  - Block size = 4096 bytes
- $2d \times 4 + (2d+1) \times 8 \leq 4096$
- $d = 170$

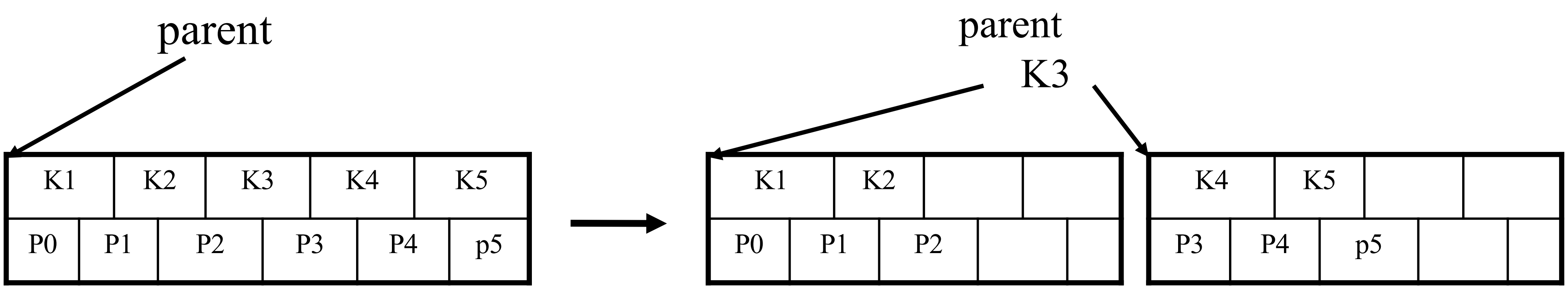
B+ Trees in Practice

- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities
  - Height 4: 133^4 = 312,900,700 records
  - Height 3: 133^3 = 2,352,637 records
- Can often hold top levels in buffer pool
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 Mbytes

Insertion in a B+ Tree

Insert (K, P)

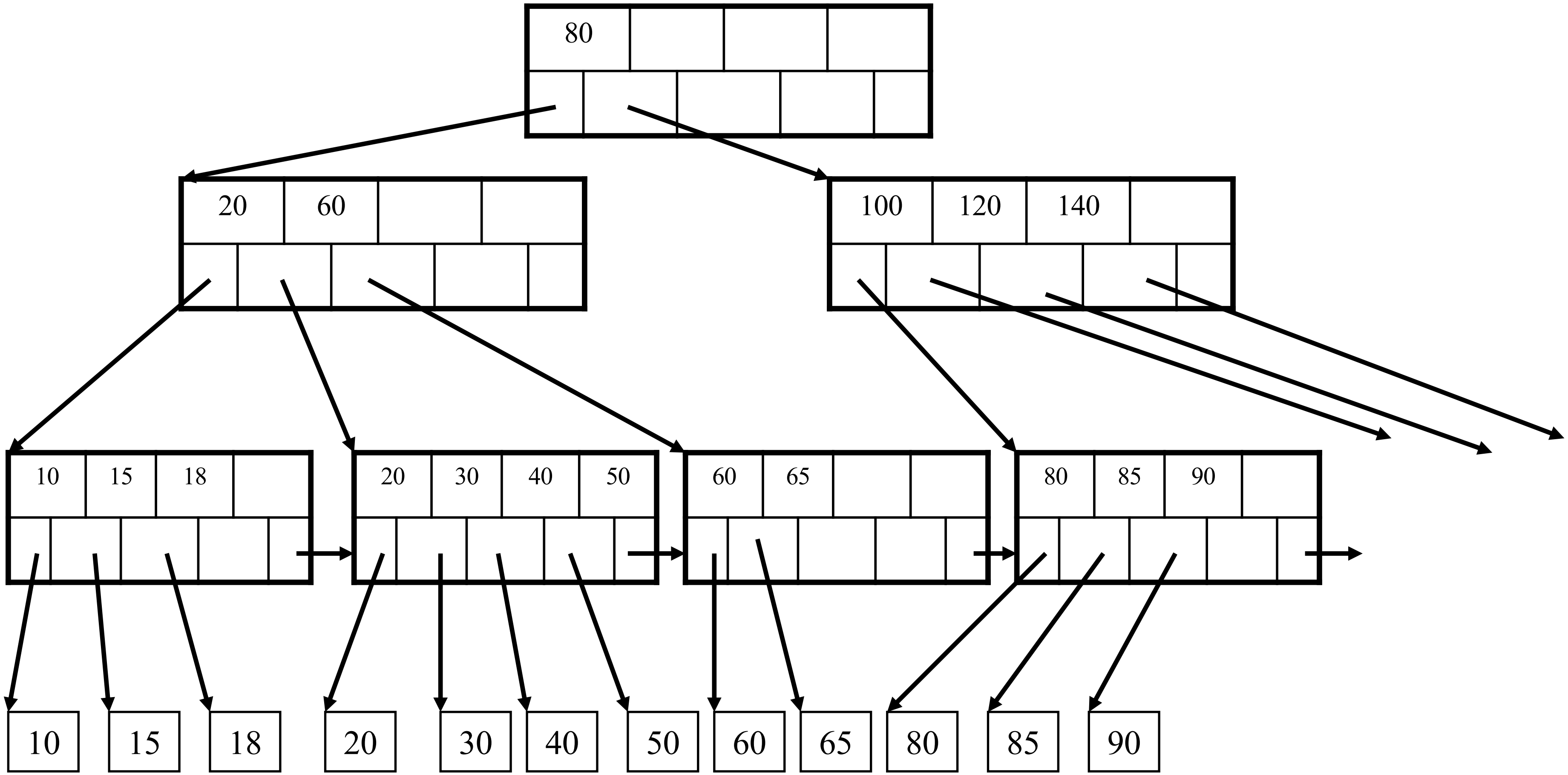
- Find leaf where K belongs, insert
- If no overflow (2d keys or less), halt
- If overflow (2d+1 keys), split node, insert in parent:



- If leaf, also keep K3 in right node
- When root splits, new root has 1 key only

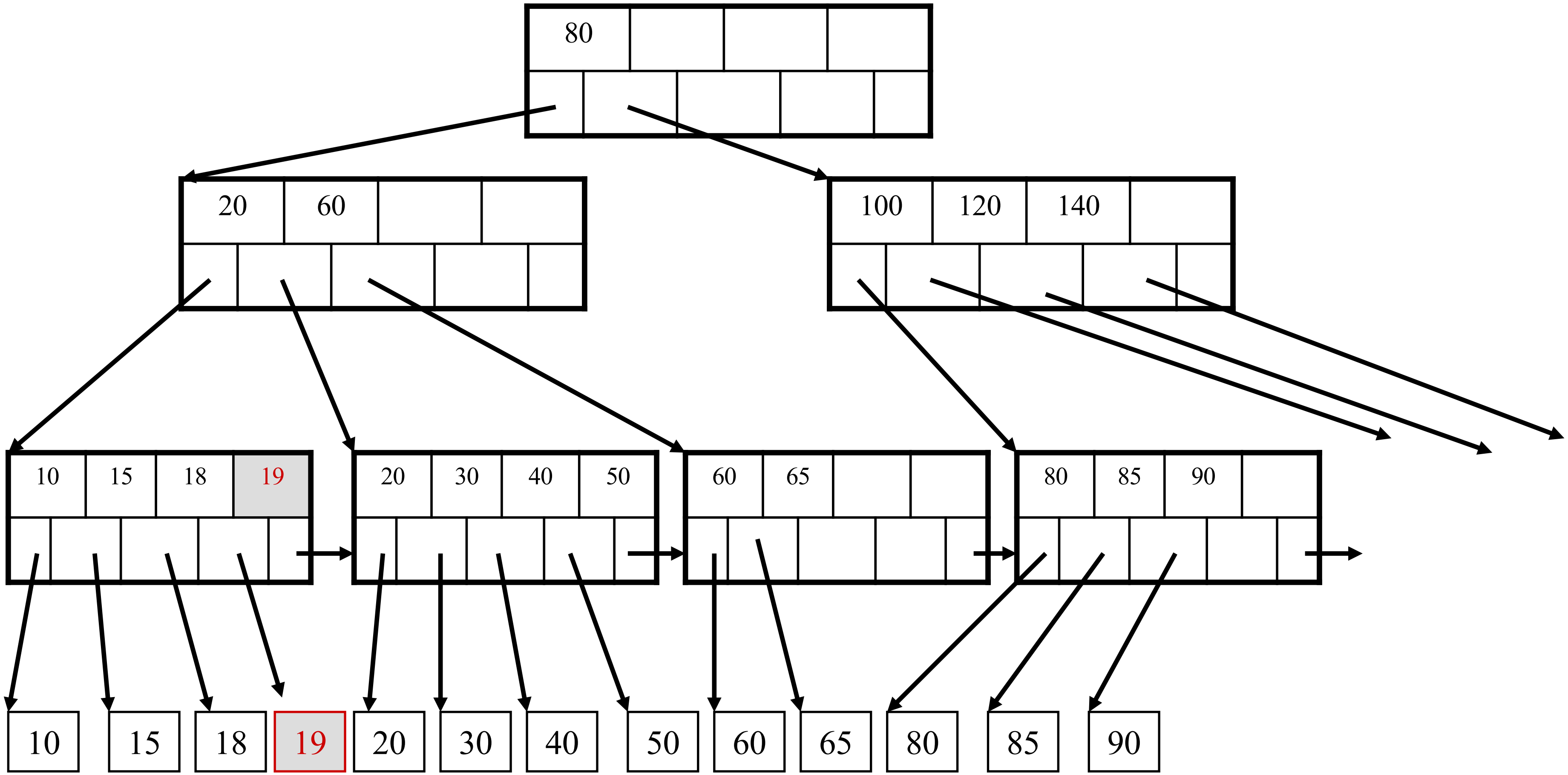
Insertion in a B+ Tree

Insert K=19



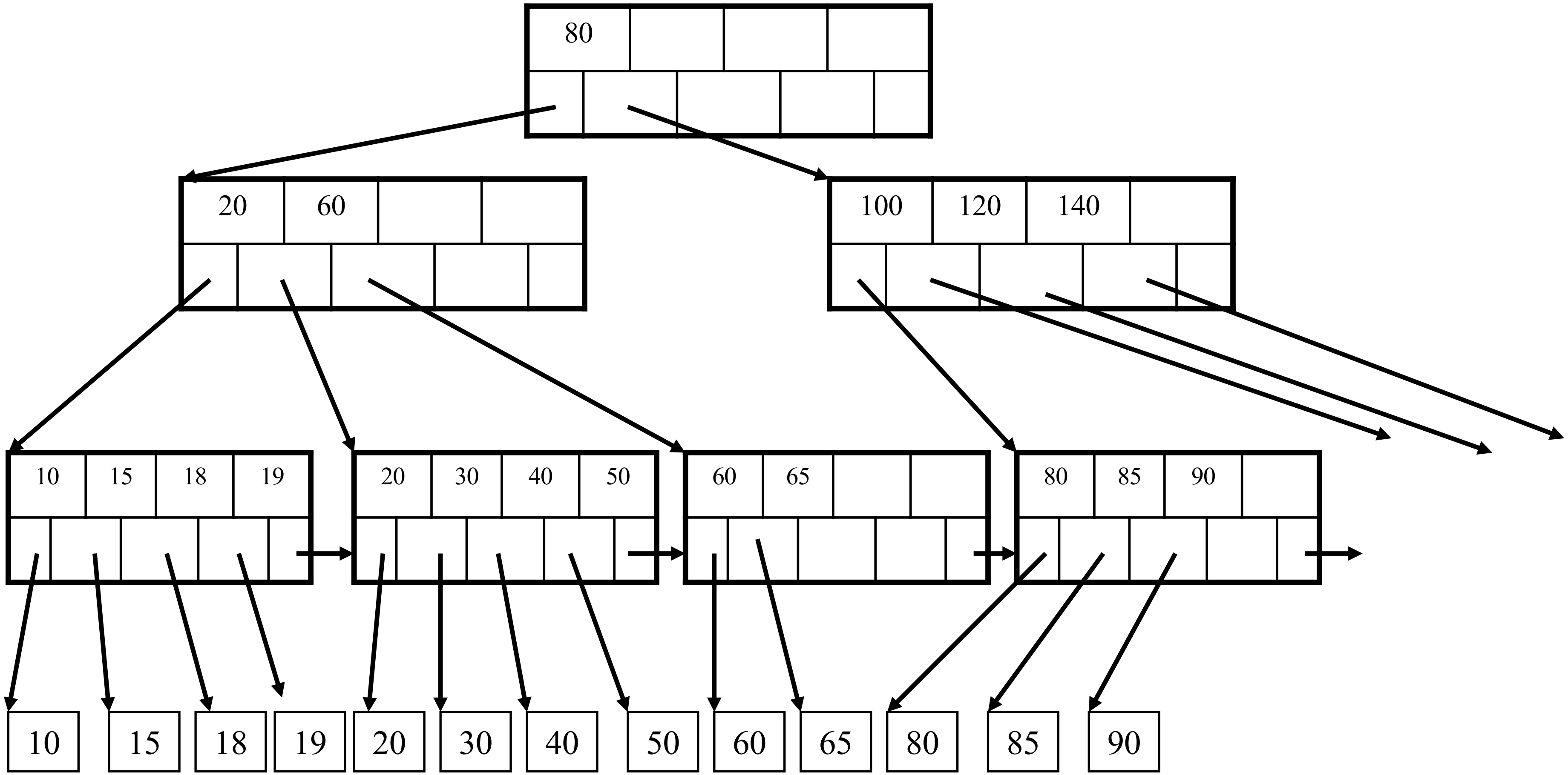
Insertion in a B+ Tree

After insertion



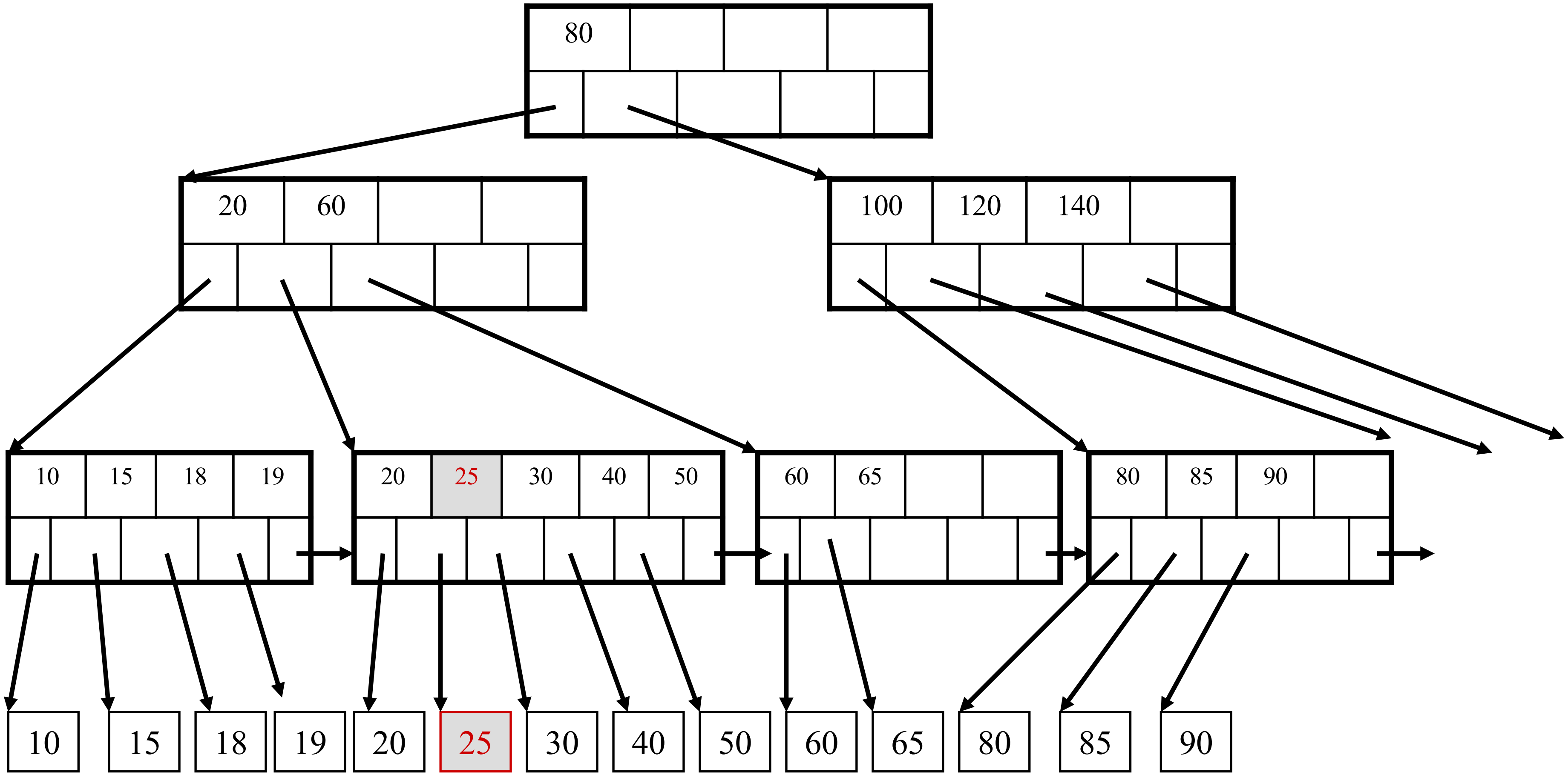
Insertion in a B+ Tree

Now insert 25



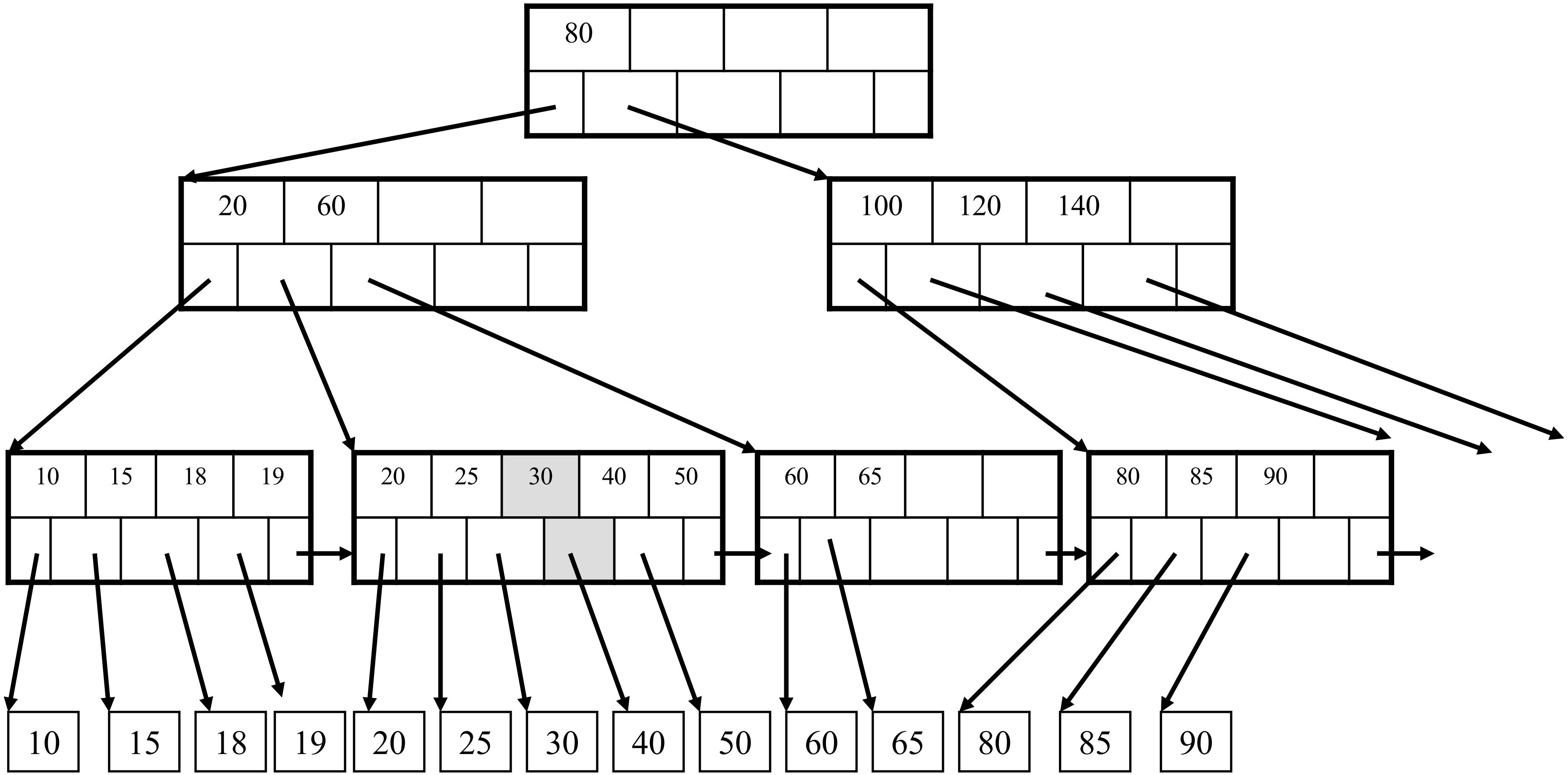
Insertion in a B+ Tree

After insertion



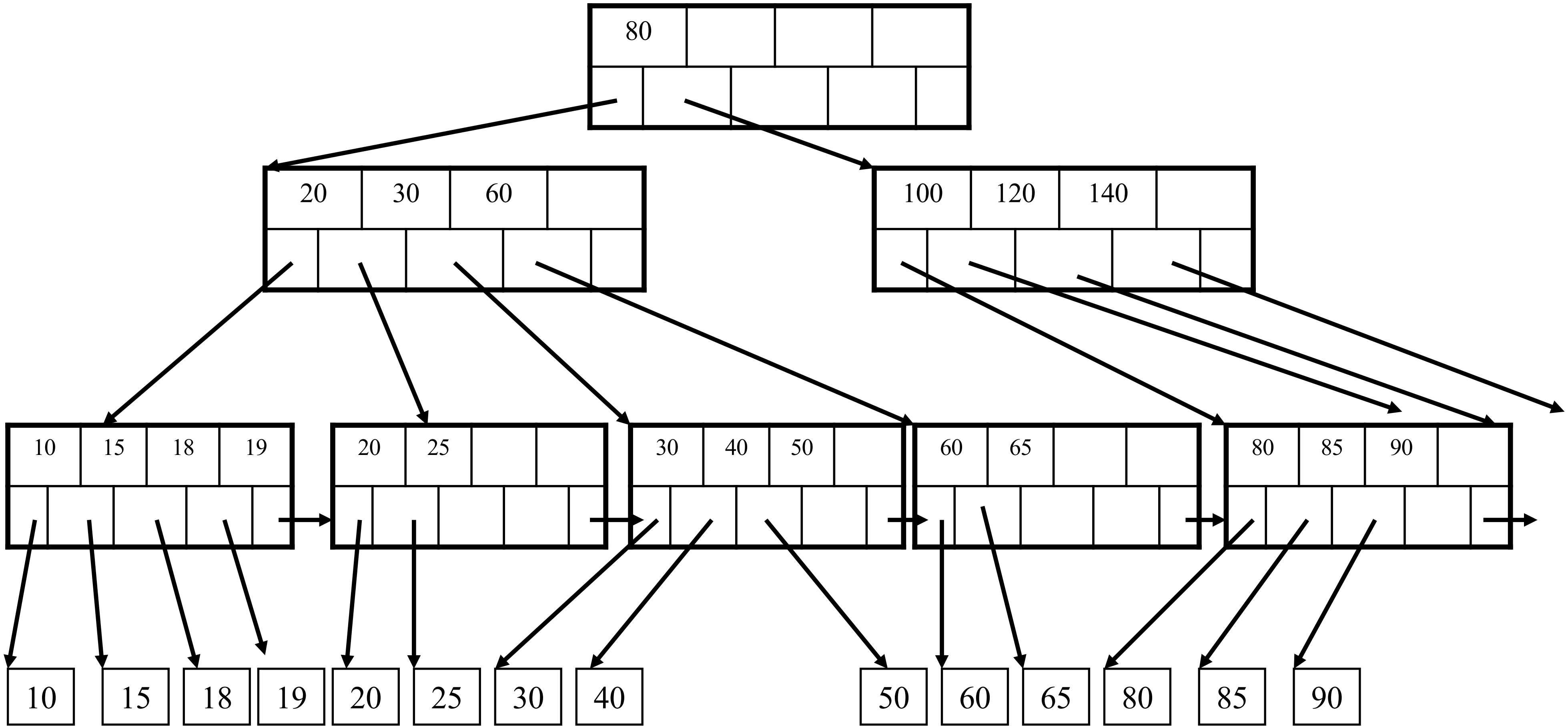
Insertion in a B+ Tree

But now have to split !



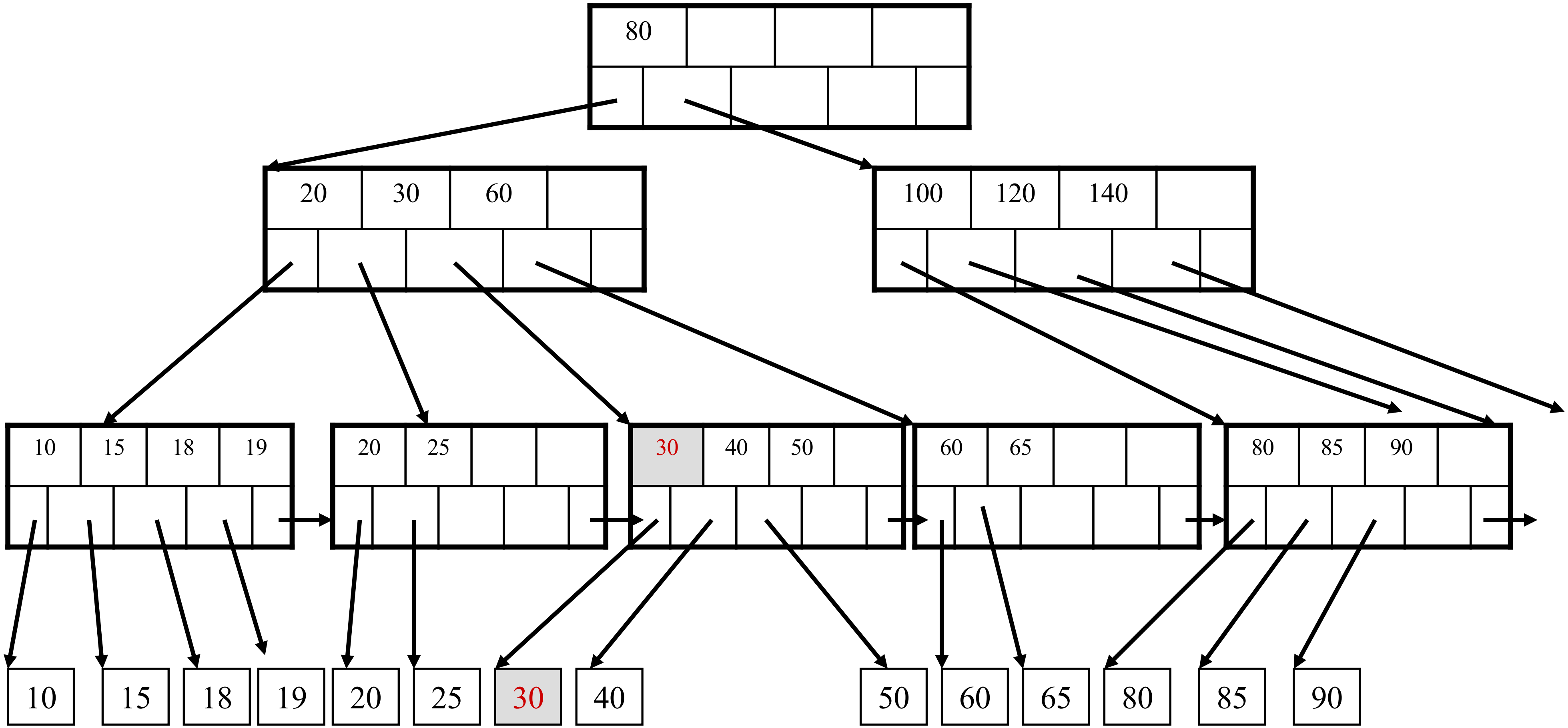
Insertion in a B+ Tree

After the split



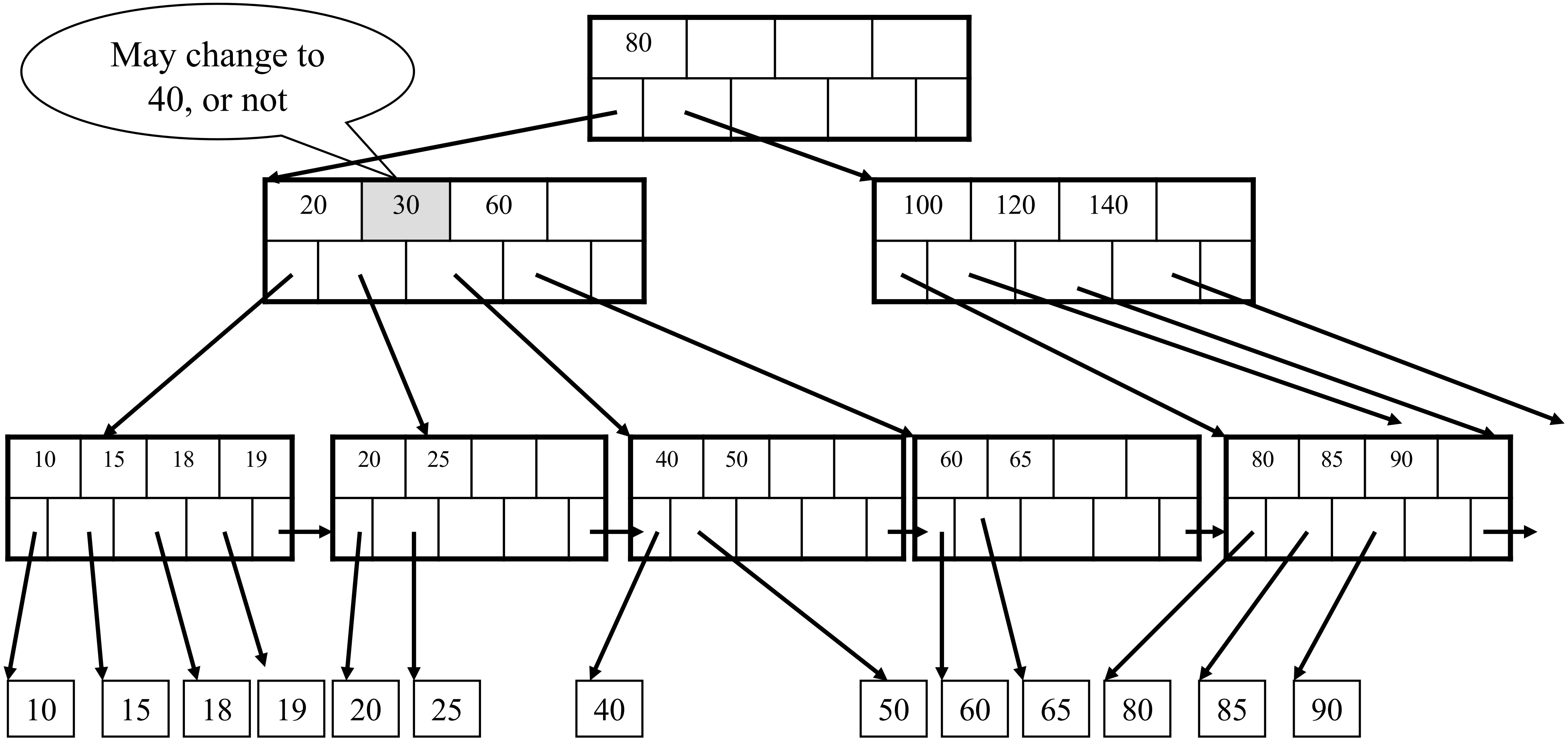
Deletion from a B+ Tree

Delete 30



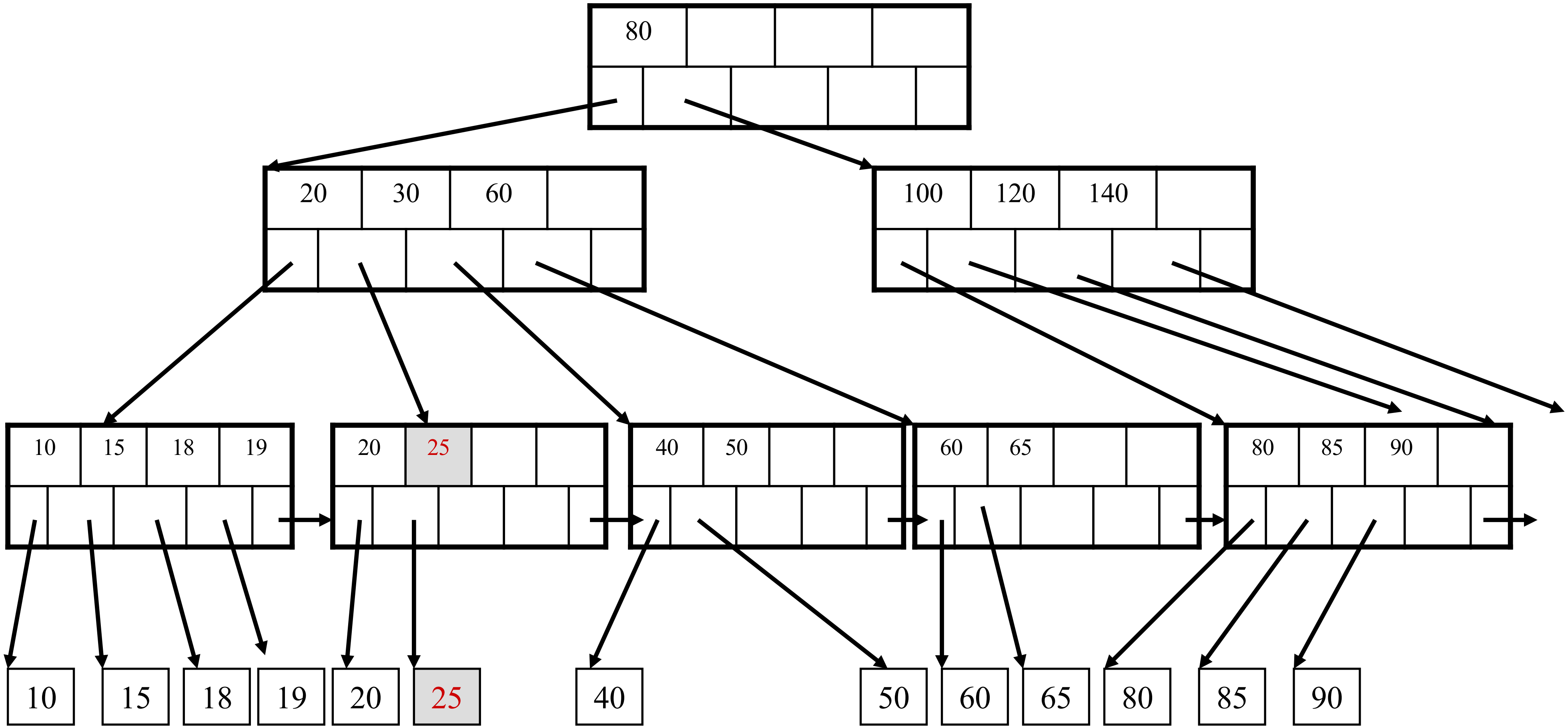
Deletion from a B+ Tree

After deleting 30



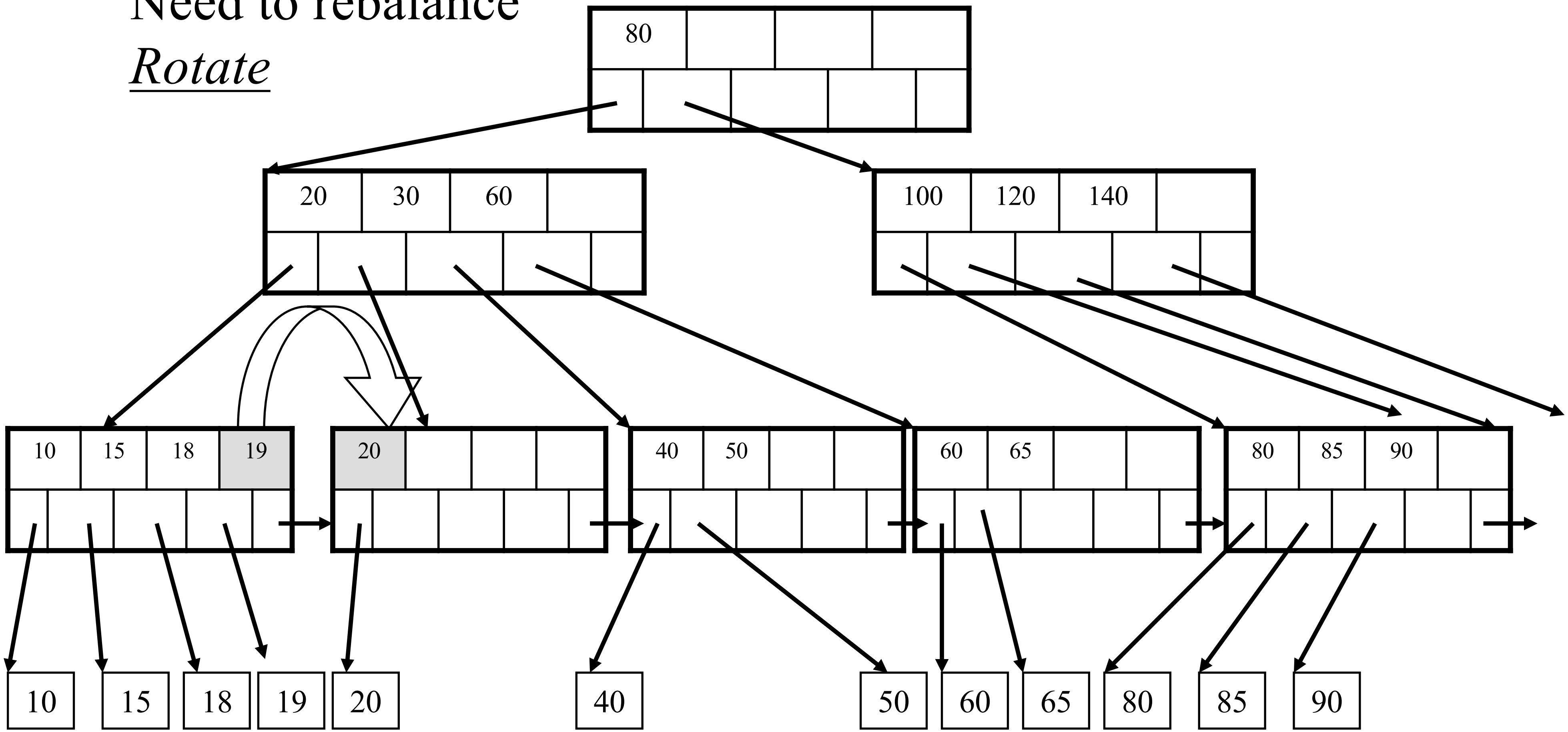
Deletion from a B+ Tree

Now delete 25



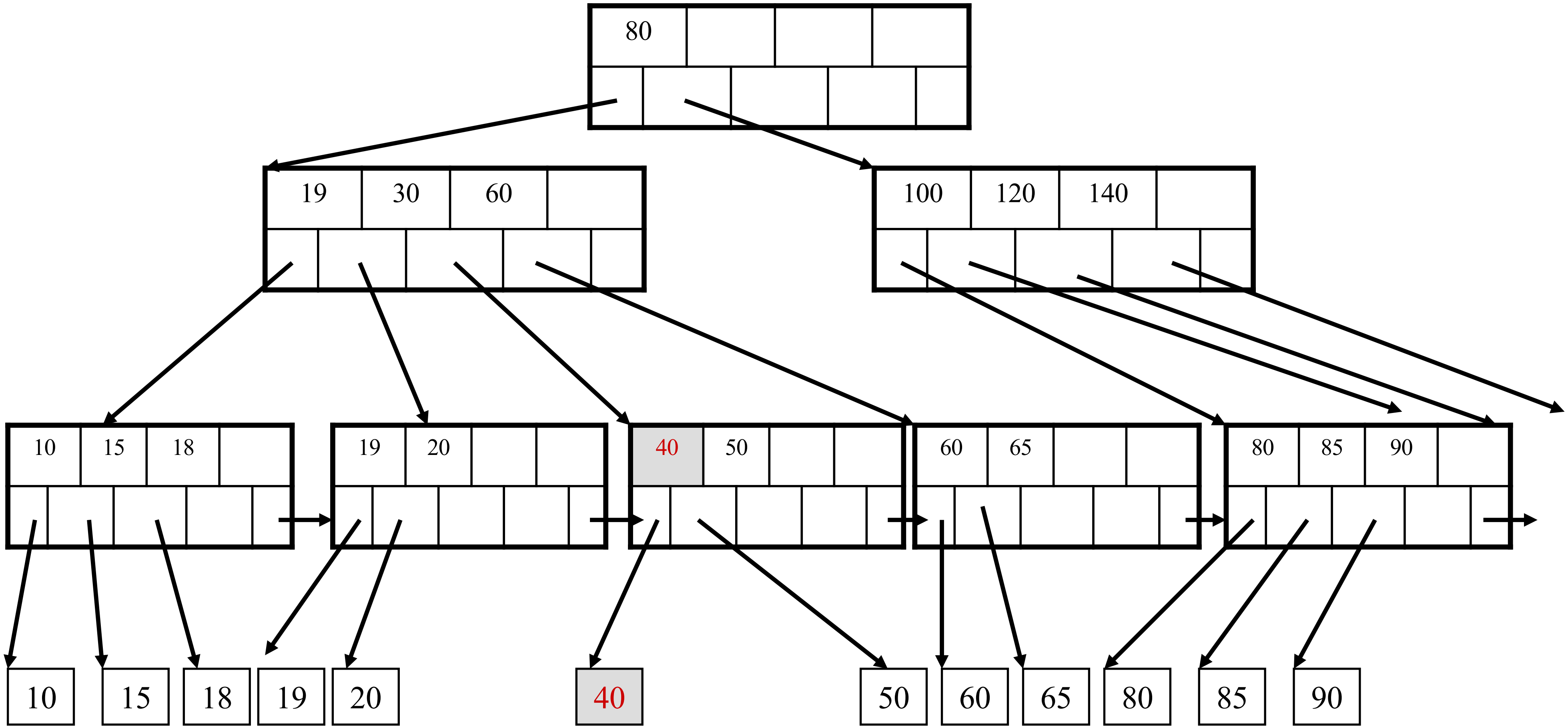
Deletion from a B+ Tree

After deleting 25
Need to rebalance
<u>*Rotate*</u>



Deletion from a B+ Tree

Now delete 40

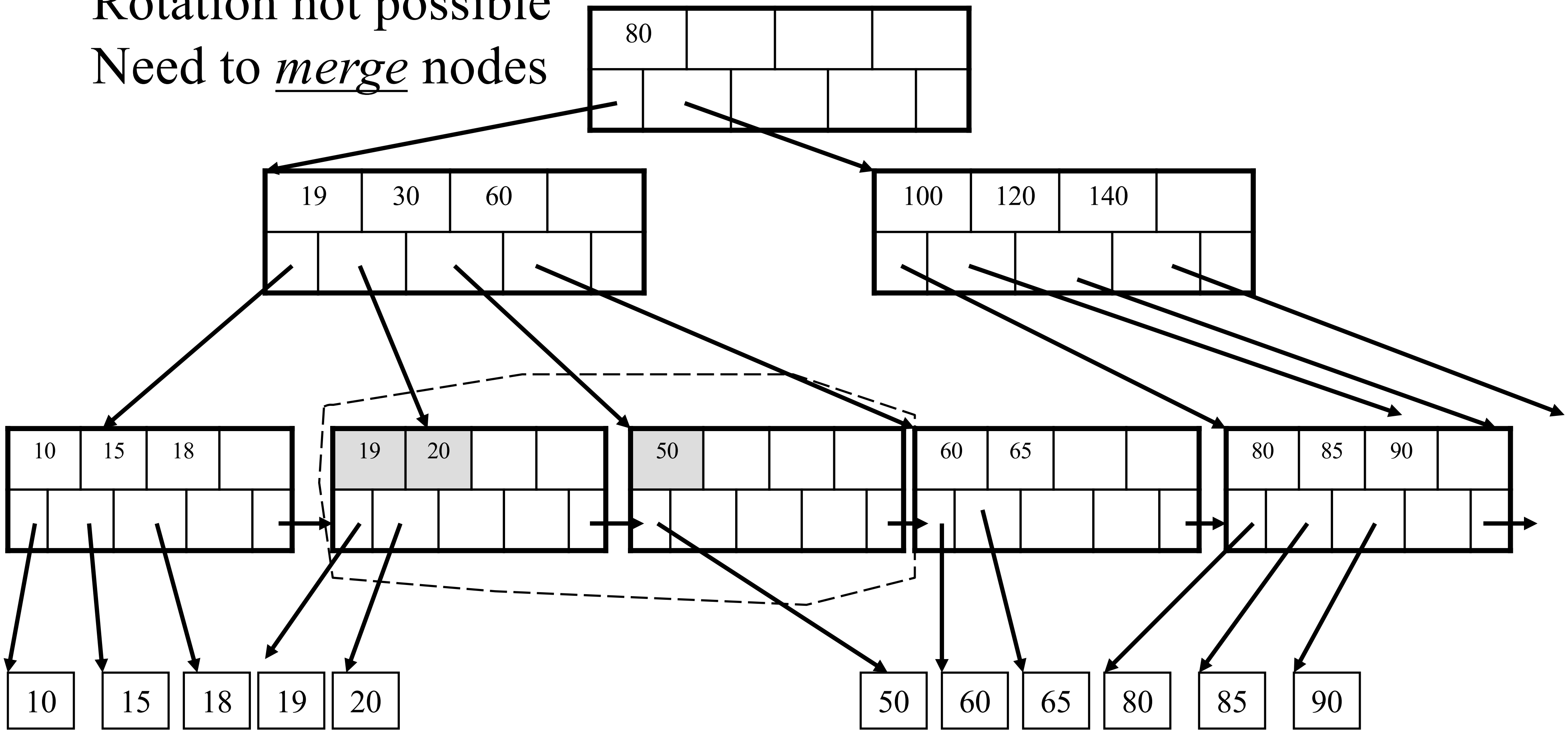


Deletion from a B+ Tree

After deleting 40

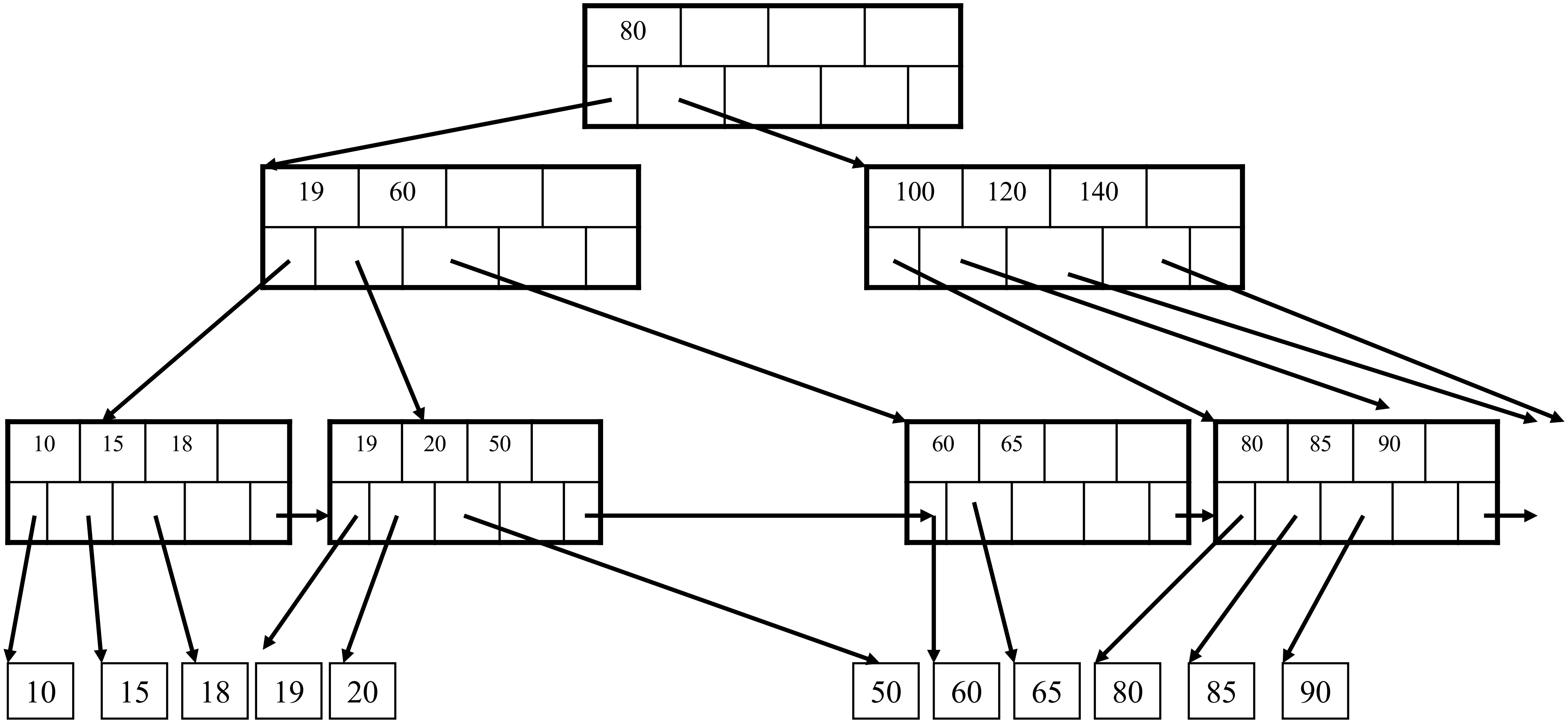
Rotation not possible

Need to <u>*merge*</u> nodes



Deletion from a B+ Tree

Final tree



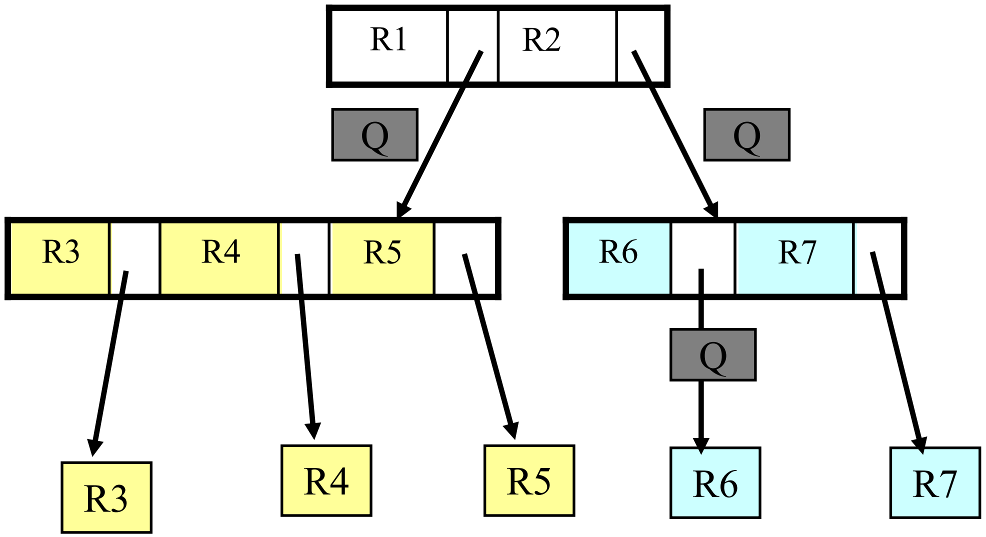
Summary on B+ Trees

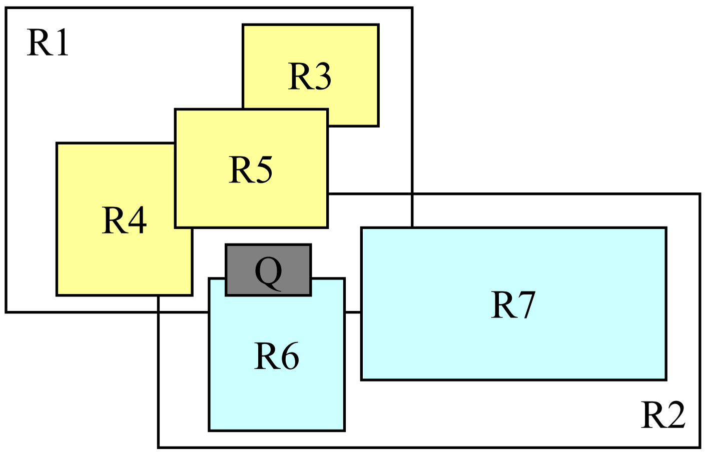
- Default index structure on most DBMSs
- Very effective at answering 'point' queries:
  productName = 'gizmo'
- Effective for range queries:
  50 < price AND price < 100
- Less effective for multirange:
  50 < price < 100 AND 2 < quant < 20

R-Tree: a multidimensional B-Tree

Designed for spatial data

Search key values are bounding boxes





For insertion: at each level, choose child whose bounding box needs least enlargement (in terms of area)