Introduction to Database Systems
CSE 444

Lecture 16: Data Storage and Indexes

About the Midterm

- Open book and open notes
  - But you won’t have time to read during midterm!
  - No laptops, no mobile devices

- Four questions:
  1. SQL
  2. ER Diagrams / Database design
  3. Transactions - recovery
  4. Transactions - concurrency control

More About the Midterm

- Review Lectures 1 through 15
  - Read the lecture notes carefully
  - Read the book for extra details, extra explanations

- Review Project 1 (Project 2 not on any exam)
- Review HW1 and HW2

- Take a look at sample midterms & finals

Where We Are

- How to use a DBMS as a:
  - Data analyst: SQL, SQL, SQL,…
  - Application programmer: JDBC, XML,…
  - Database administrator: tuning, triggers, security
  - Massive-scale data analyst: Pig/MapReduce
- How DBMSs work:
  - Transactions
  - Data storage and indexing
  - Query execution
- Databases as a service

Outline

- Storage model
- Index structures (Section 14.1)
  - [Old edition: 13.1 and 13.2]
- B-trees (Section 14.2)
  - [Old edition: 13.3]

Storage Model

- DBMS needs spatial and temporal control over storage
  - Spatial control for performance
  - Temporal control for correctness and performance
    - Solution: Buffer manager inside DBMS (see past lectures)

- For spatial control, two alternatives
  - Use "raw" disk device interface directly
  - Use OS files

Spatial Control
Using "Raw" Disk Device Interface

- **Overview**
  - DBMS issues low-level storage requests directly to disk device
- **Advantages**
  - DBMS can ensure that important queries access data sequentially
  - Can provide highest performance
- **Disadvantages**
  - Requires devoting entire disks to the DBMS
  - Reduces portability as low-level disk interfaces are OS specific
  - Many devices are in fact "virtual disk devices"

Spatial Control Using OS Files

- **Overview**
  - DBMS creates one or more very large OS files
- **Advantages**
  - Allocating large file on empty disk can yield good physical locality
- **Disadvantages**
  - OS can limit file size to a single disk
  - OS can limit the number of open file descriptors
  - But these drawbacks have mostly been overcome by modern OSs

Commercial Systems

- Most commercial systems offer both alternatives
  - Raw device interface for peak performance
  - OS files more commonly used
- In both cases, we end-up with a DBMS file abstraction implemented on top of OS files or raw device interface

Outline

- Storage model
- Index structures (Section 14.1)
  - [Old edition: 13.1 and 13.2]
- B-trees (Section 14.2)
  - [Old edition: 13.3]

Database File Types

The data file can be one of:

- Heap file
  - Set of records, partitioned into blocks
  - Unsorted
- Sequential file
  - Sorted according to some attribute(s) called *key*

"key" here means something else than "primary key"

Index

- A (possibly separate) file, that allows fast access to records in the data file given a search key
- The index contains (key, value) pairs:
  – The key = an attribute value
  – The value = either a pointer to the record, or the record itself

"key" (aka "search key") again means something else

Index Classification

- **Clustered/unclustered**
  - Clustered = records close in index are close in data
  - Unclustered = records close in index may be far in data

- **Primary/secondary**
  - Meaning 1:
    - Primary = is over attributes that include the primary key
    - Secondary = otherwise
  - Meaning 2: means the same as clustered/unclustered

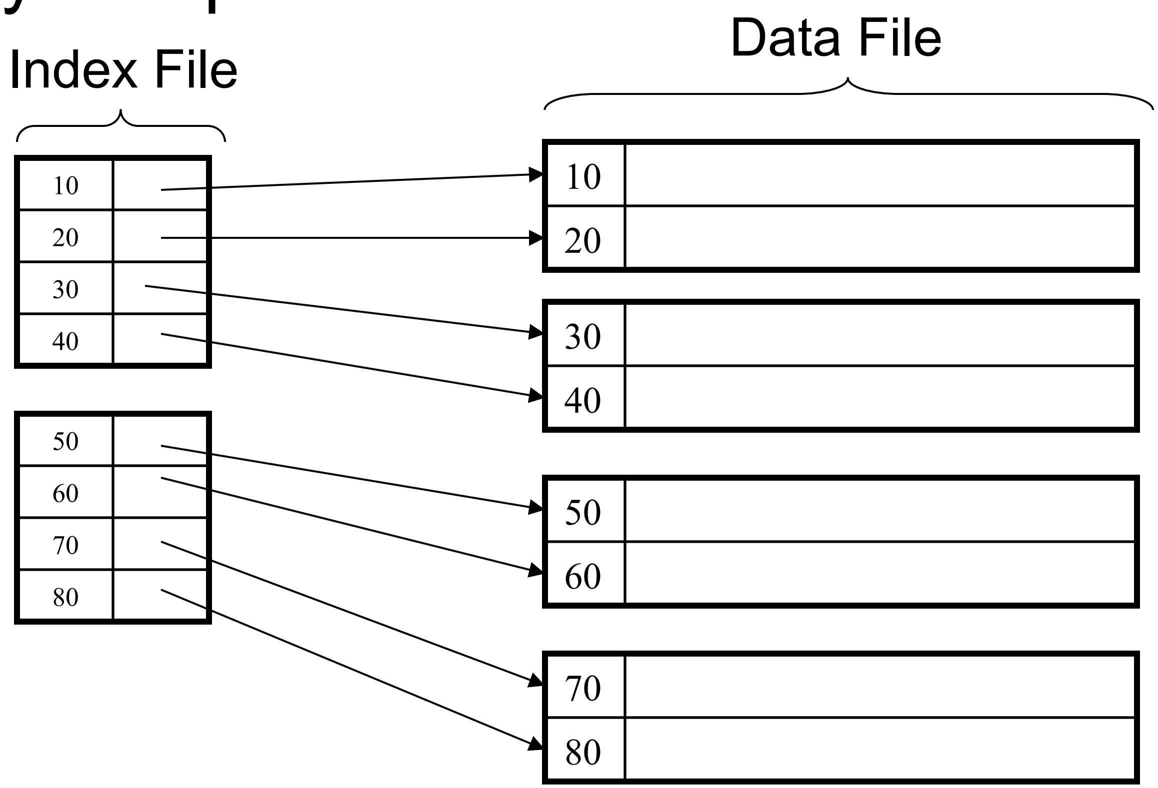
- **Organization**: B+ tree or Hash table

Clustered/Unclustered

- Clustered
  - Index determines the location of indexed records
  - Typically, clustered index is one where values are data records (but not necessary)
- Unclustered
  - Index cannot reorder data, does not determine data location
  - In these indexes: value = pointer to data record

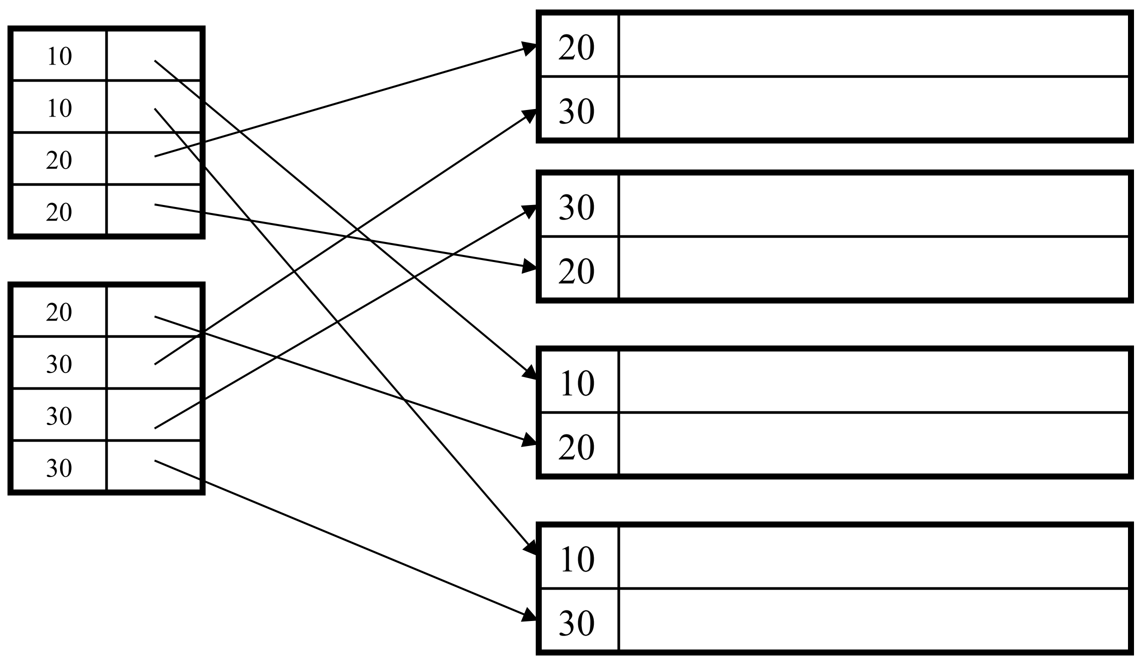
Clustered Index

- File is sorted on the index attribute
- Only one per table

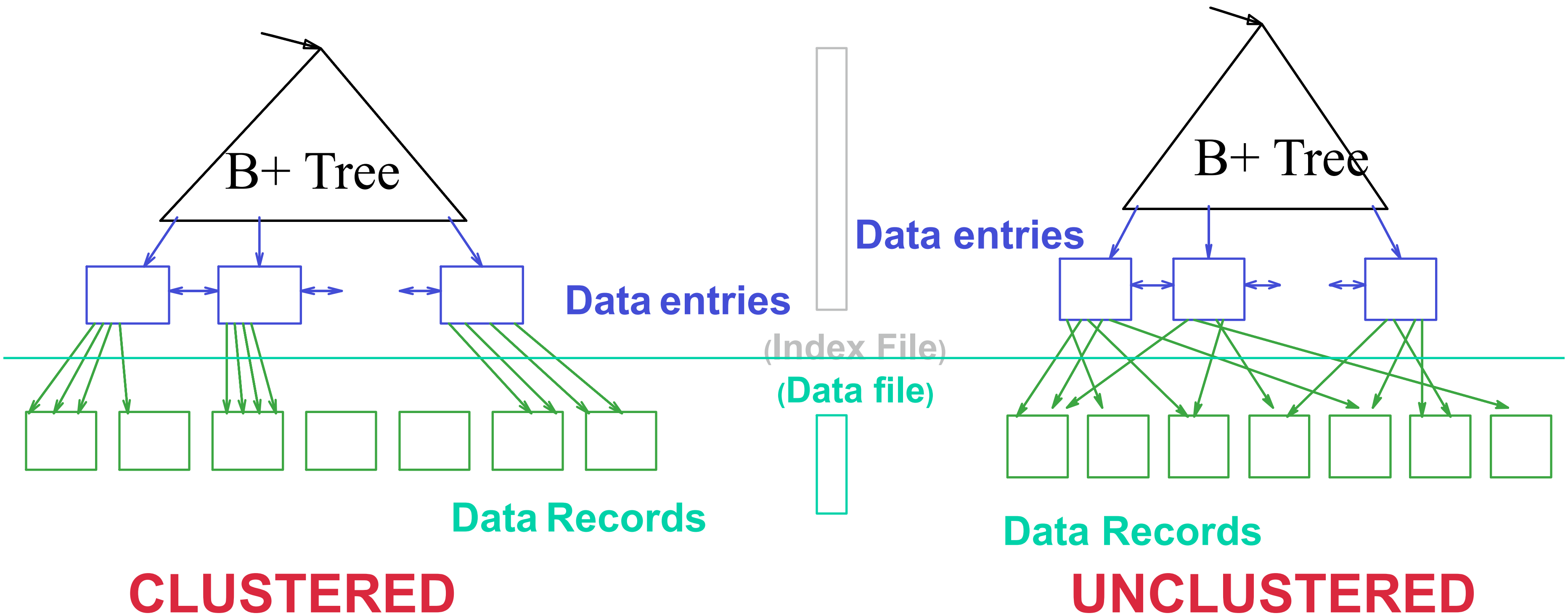


Unclustered Index

- Several per table



Clustered vs. Unclustered Index



- More commonly, in a clustered B+ Tree index, **data entries are data records**

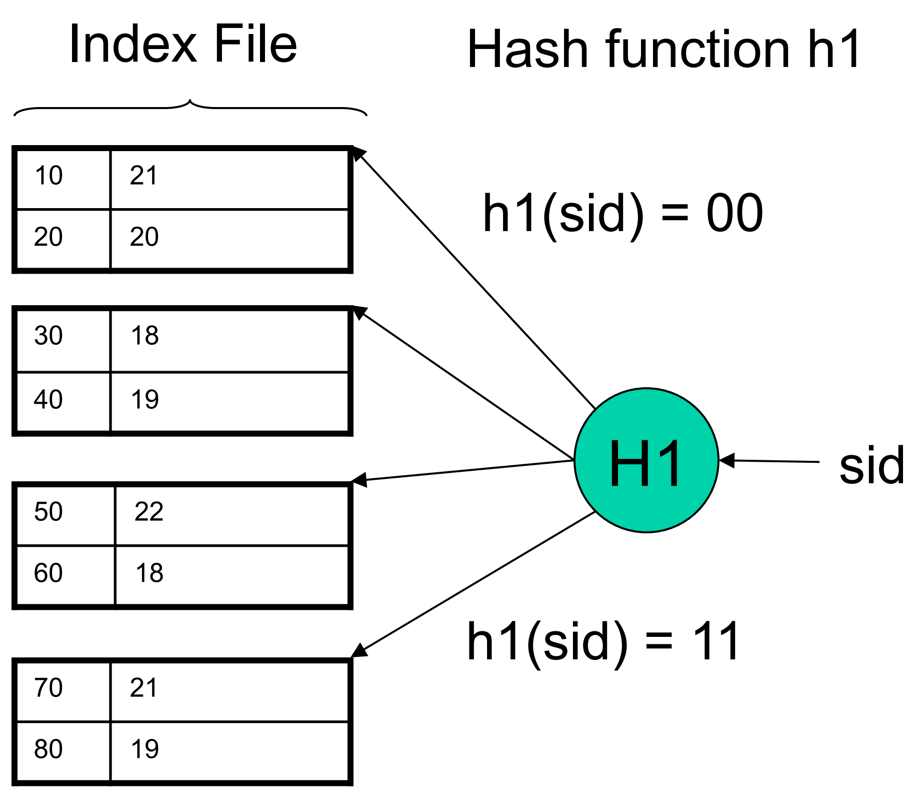
Hash-Based Index Example

Example hash-based index
on sid (student id)

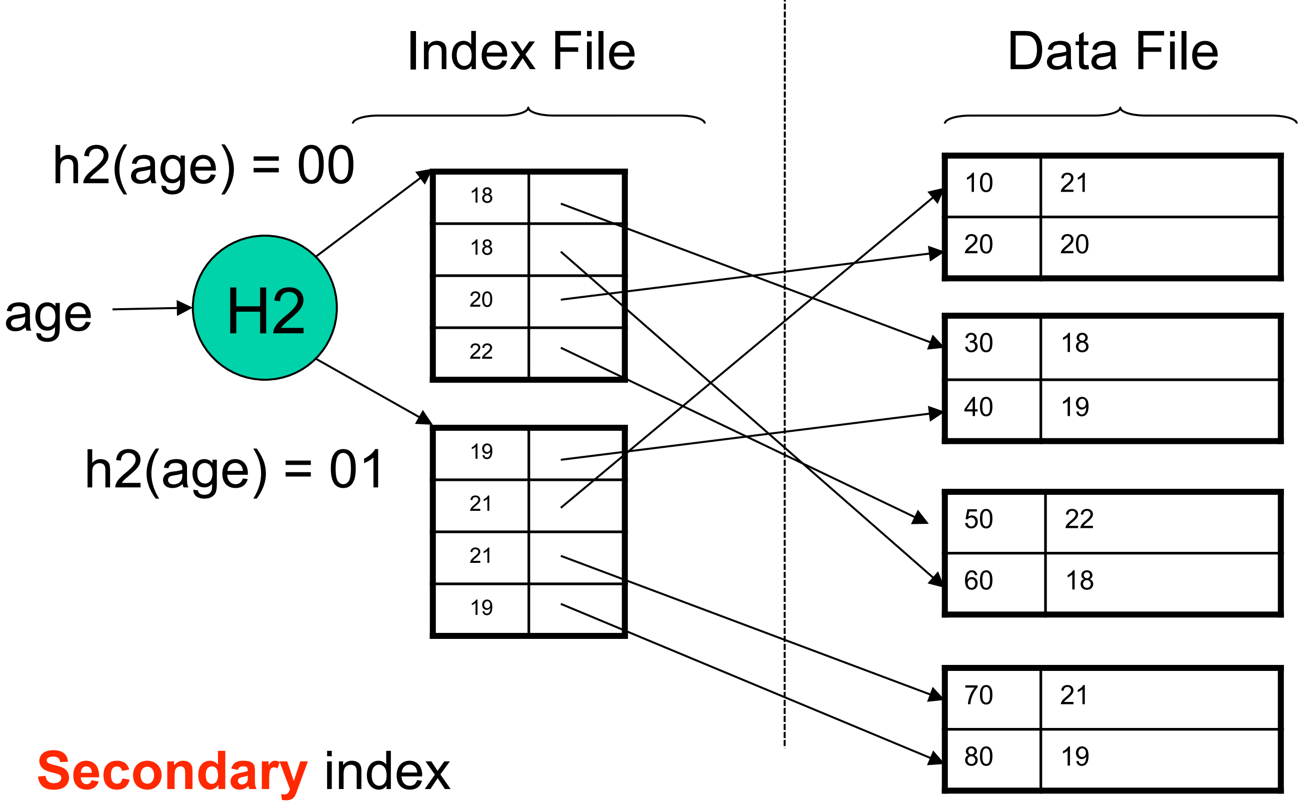
This is a **primary** index
because it determines the location
of indexed records

In this case, data entries in the index
are actual data records
There is no separate data file

This index is also **clustered**



Hash-Based Index Example 2

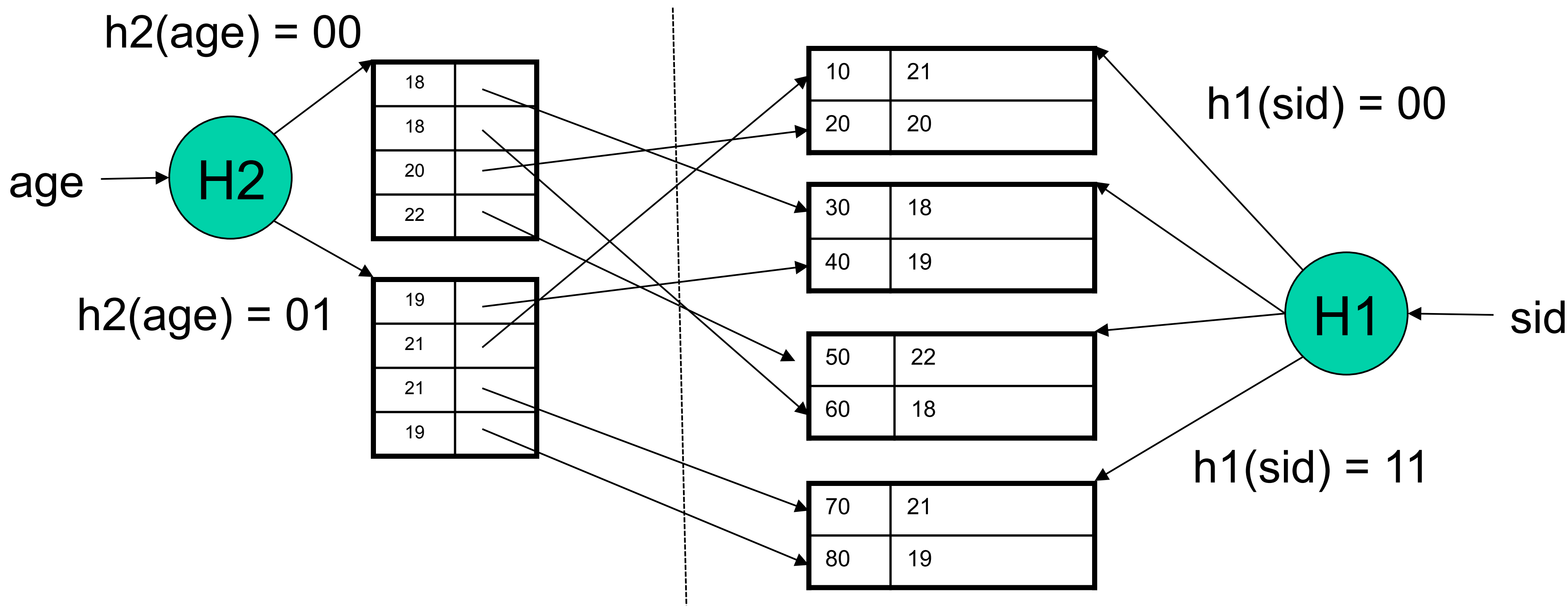


Secondary index
Data entries in index are (key,RID) pairs

Unclustered index

Hash-Based Index

Good for point queries but not range queries



Another example
of unclustered/secondary index

Another example of
clustered/primary index

Outline

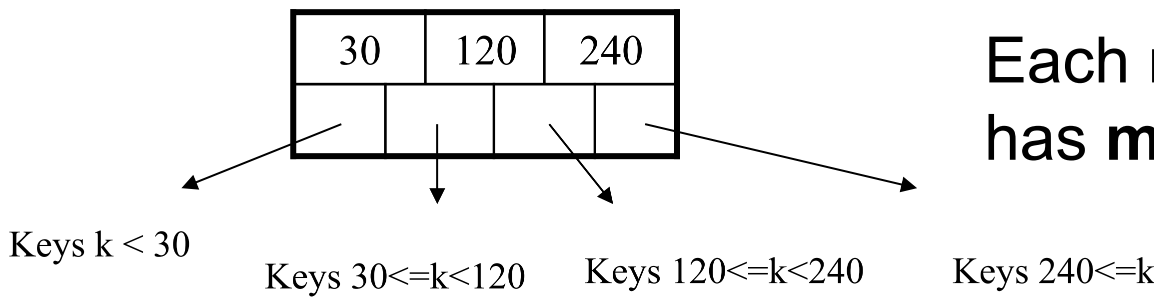
- Storage model
- Index structures (Section 14.1)
  – [Old edition: 13.1 and 13.2]
- B-trees (Section 14.2)
  – [Old edition: 13.3]

B+ Trees

- Search trees
- Idea in B Trees
  - Make 1 node = 1 block
  - Keep tree balanced in height
- Idea in B+ Trees
  - Make leaves into a linked list: facilitates range queries

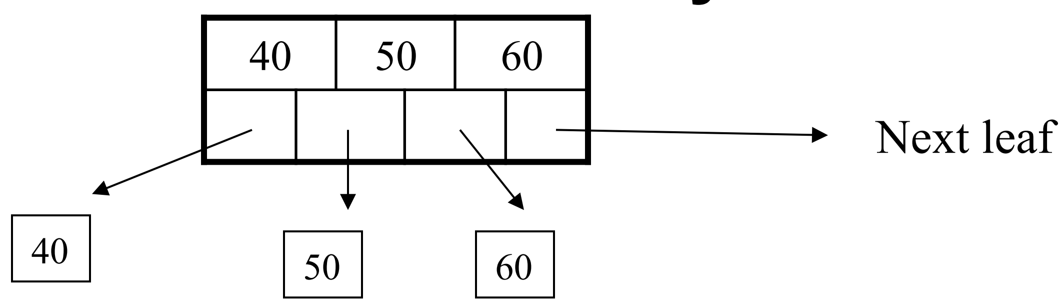
B+ Trees Basics

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



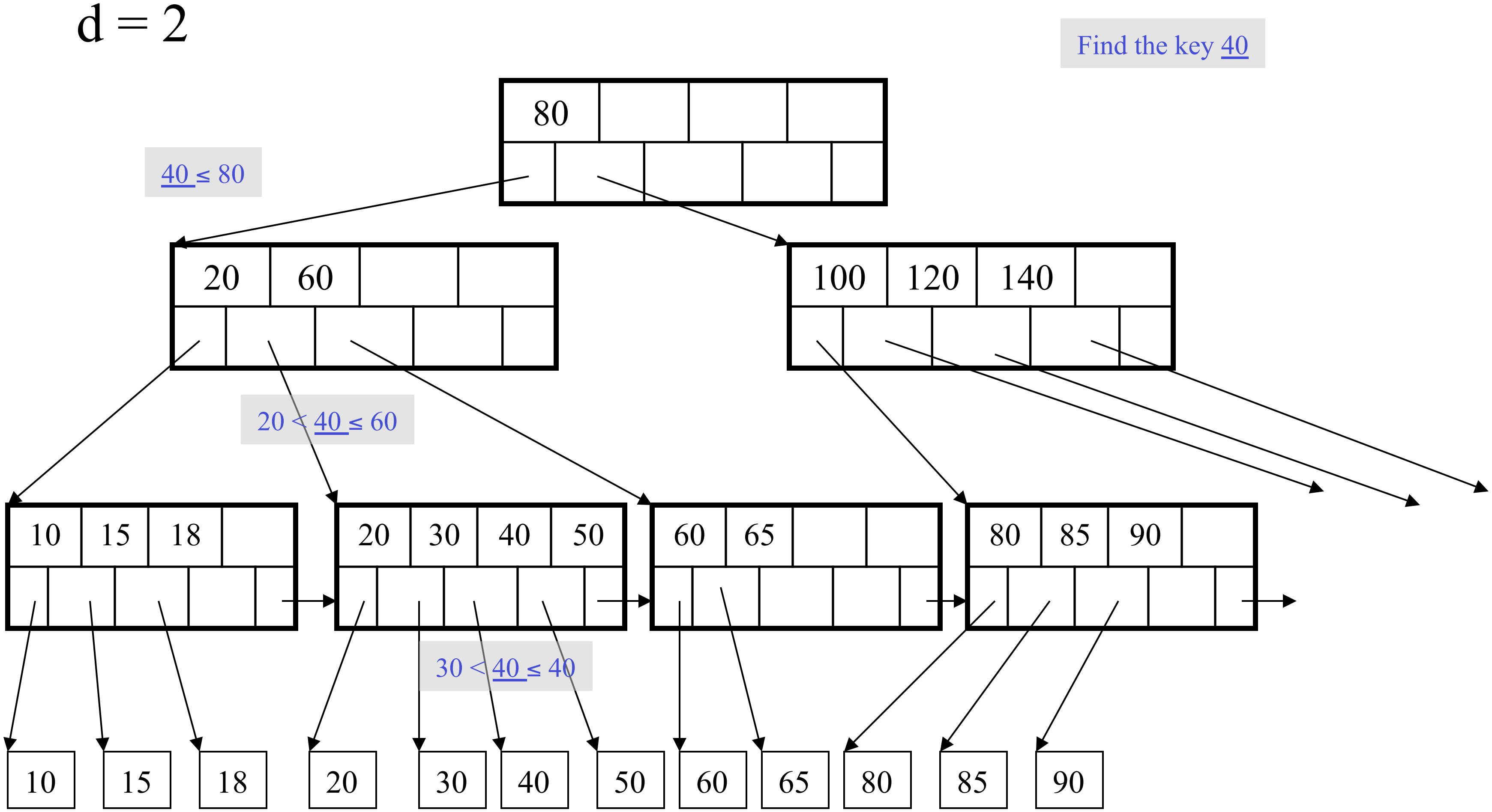
Each node also has **m+1 pointers**

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



B+ Tree Example

d = 2



B+ Tree Design

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

Searching a B+ Tree

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

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

- Range queries:
  - As above
  - Then sequential traversal

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

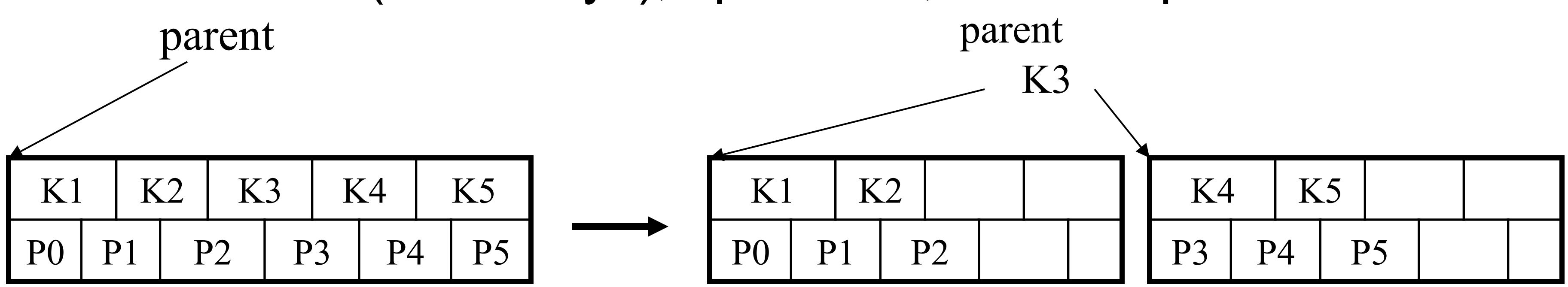
B+ Trees in Practice

- 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)

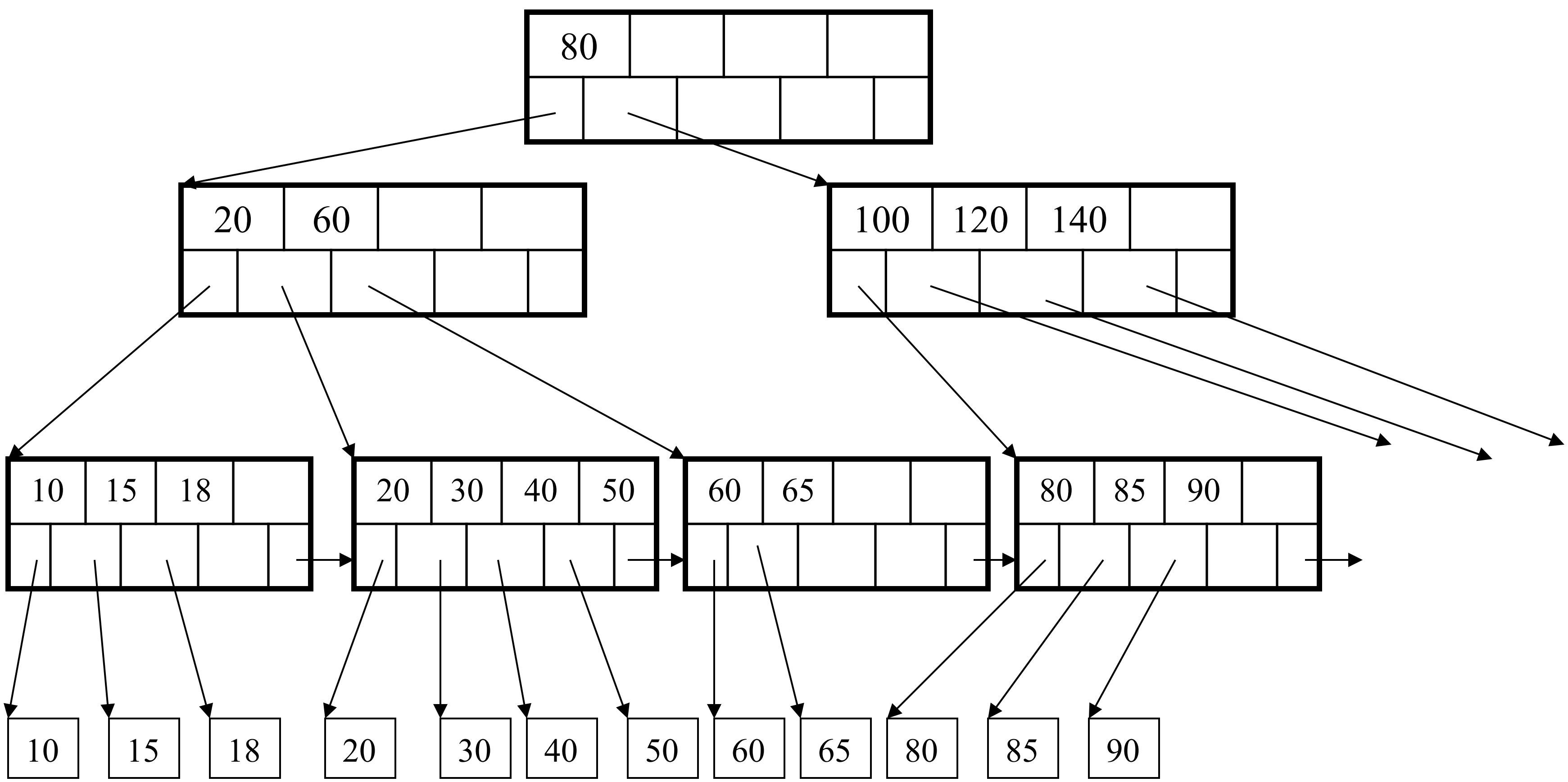
- Find leaf where K belongs, insert
- If no overflow (2d keys or less), halt
- If overflow (2d+1 keys), split node, insert in parent:



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

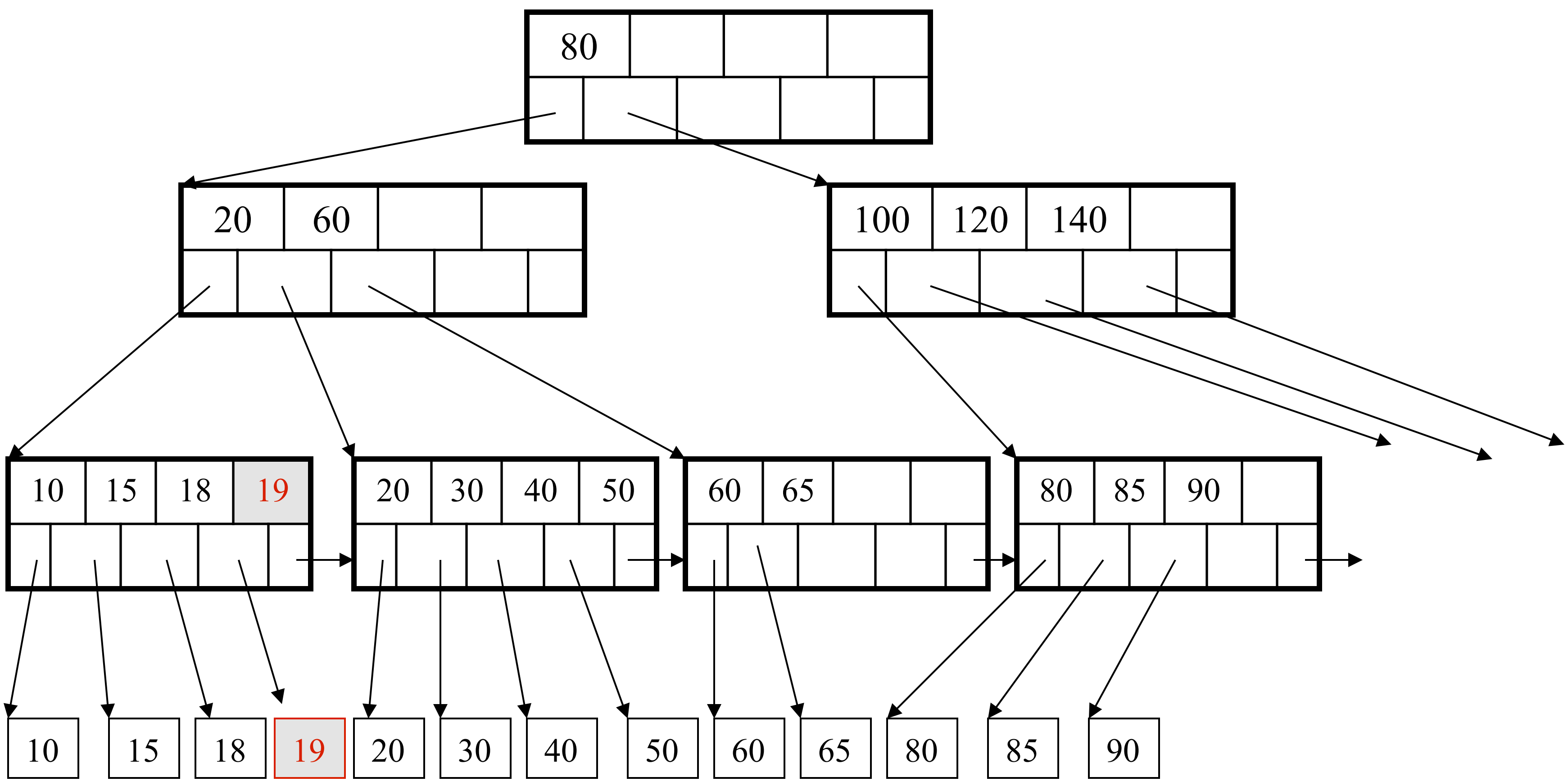
Insertion in a B+ Tree

Insert K=19



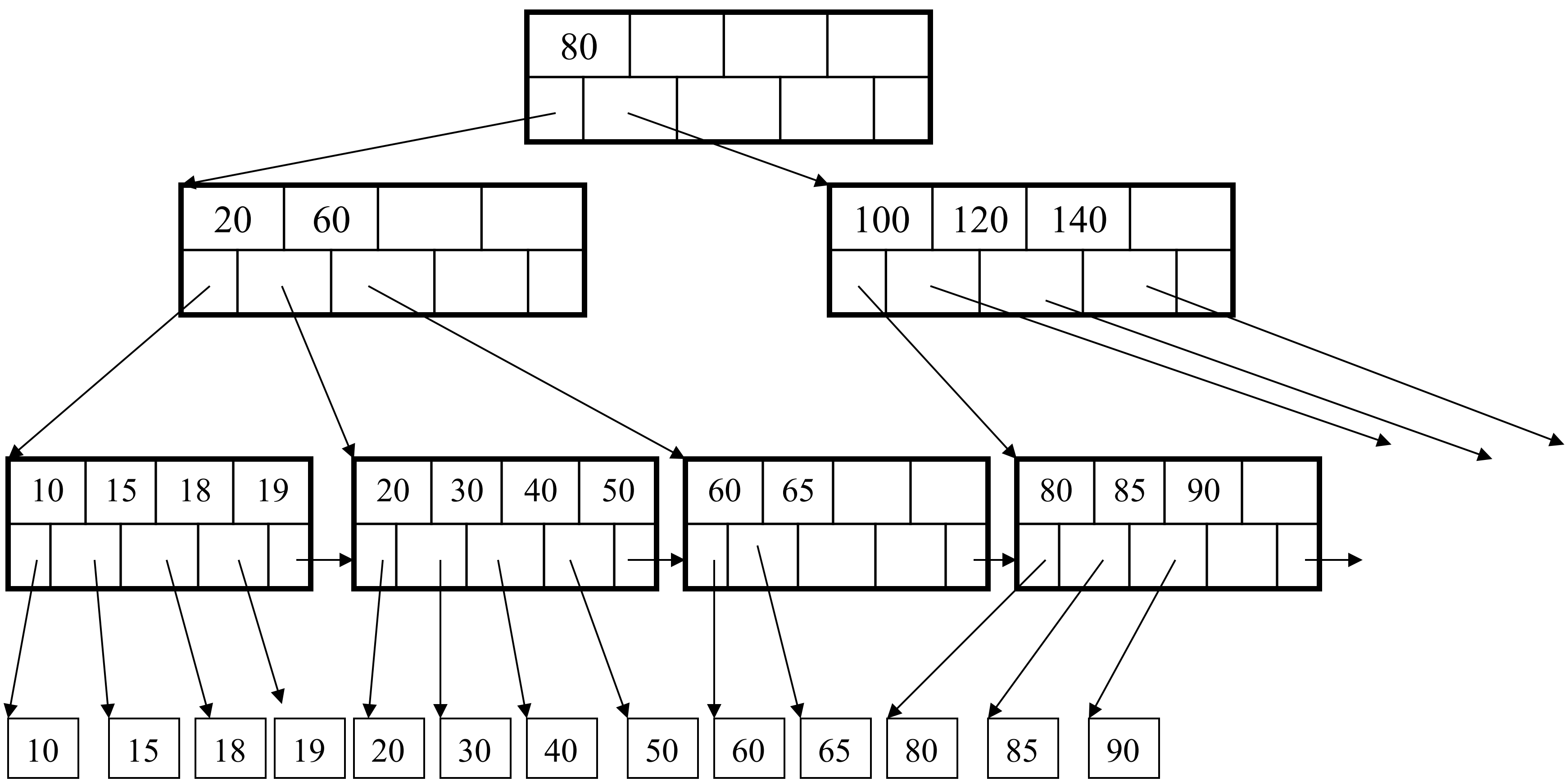
Insertion in a B+ Tree

After insertion



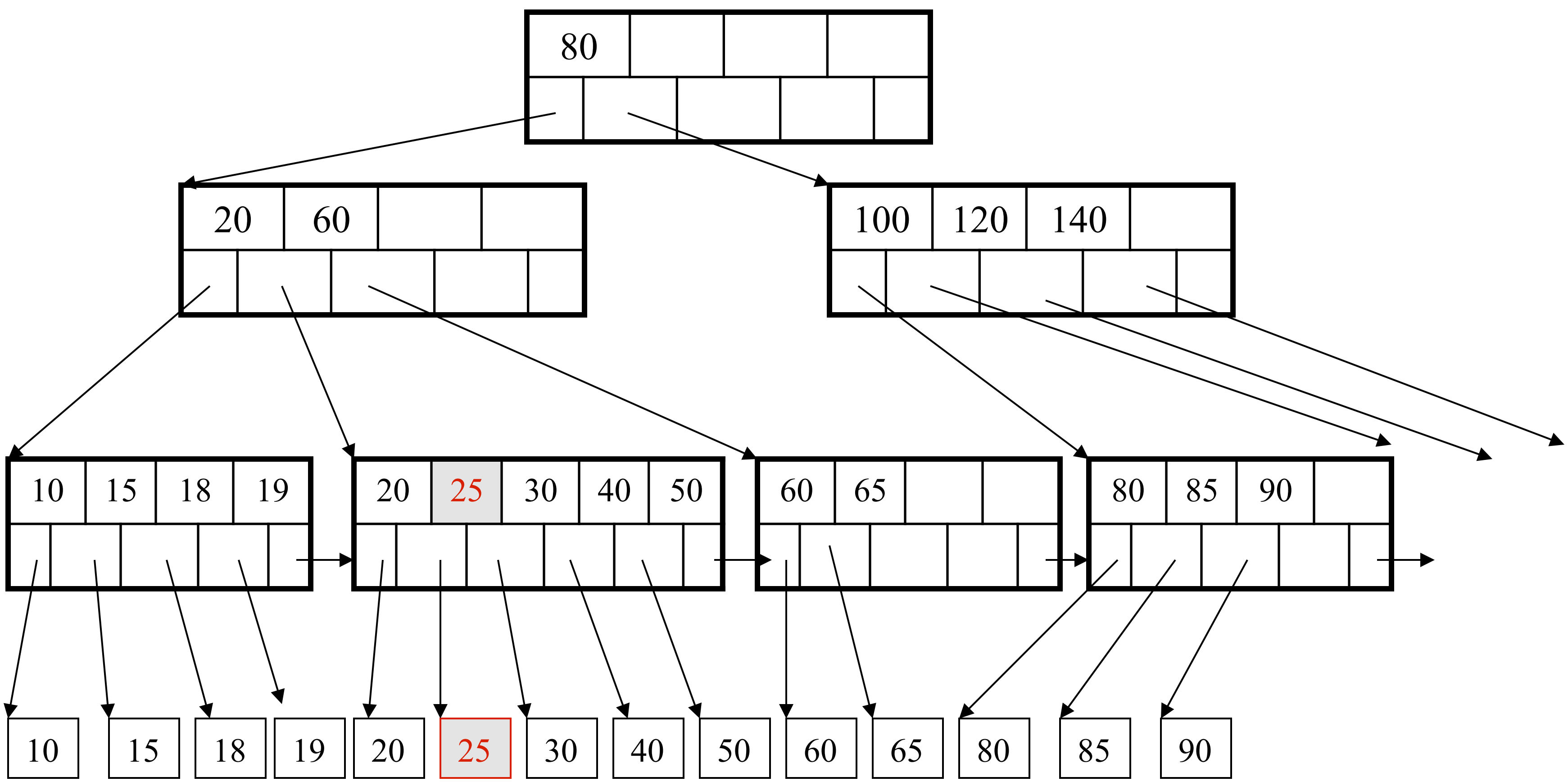
Insertion in a B+ Tree

Now insert 25



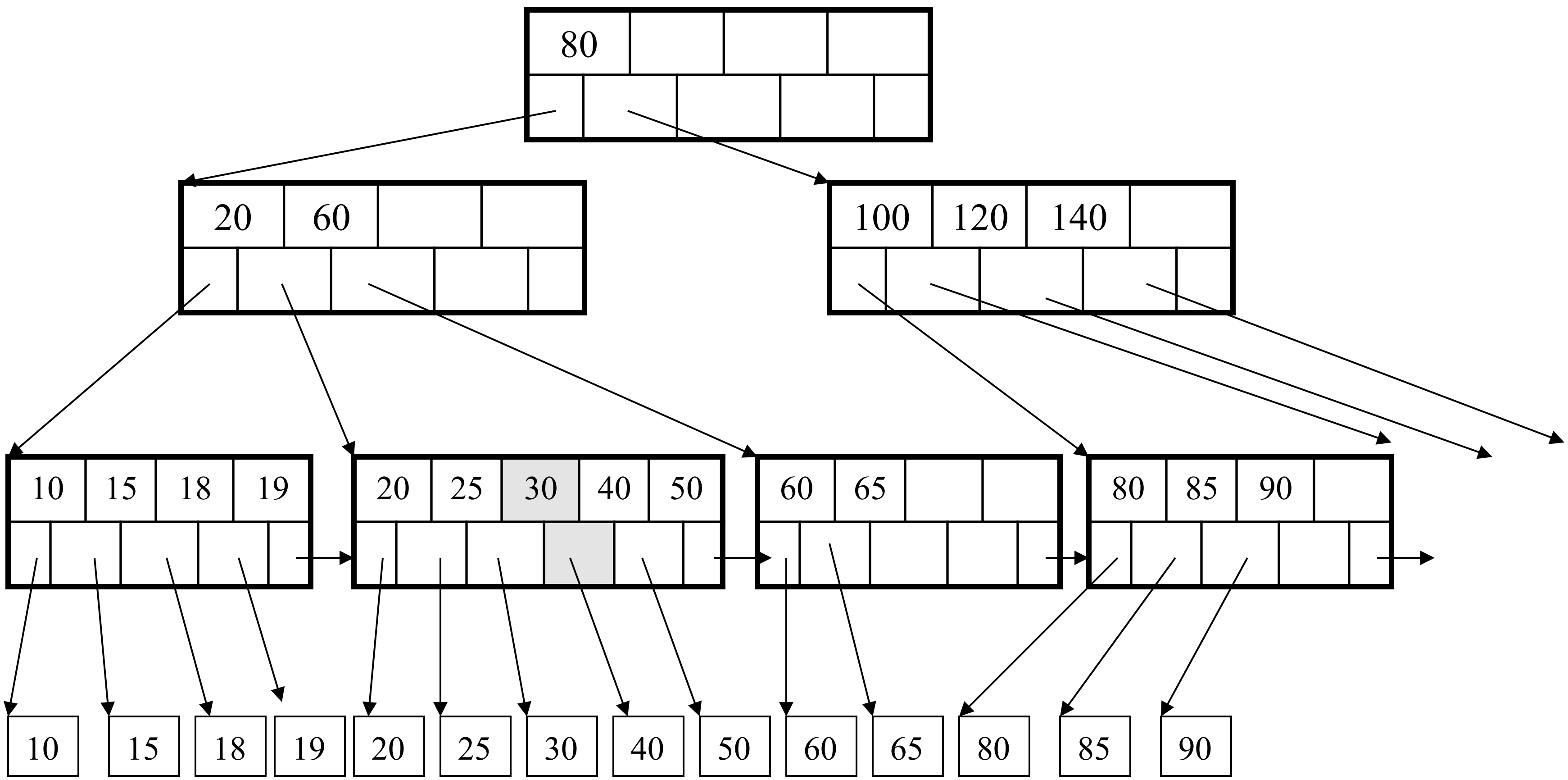
Insertion in a B+ Tree

After insertion



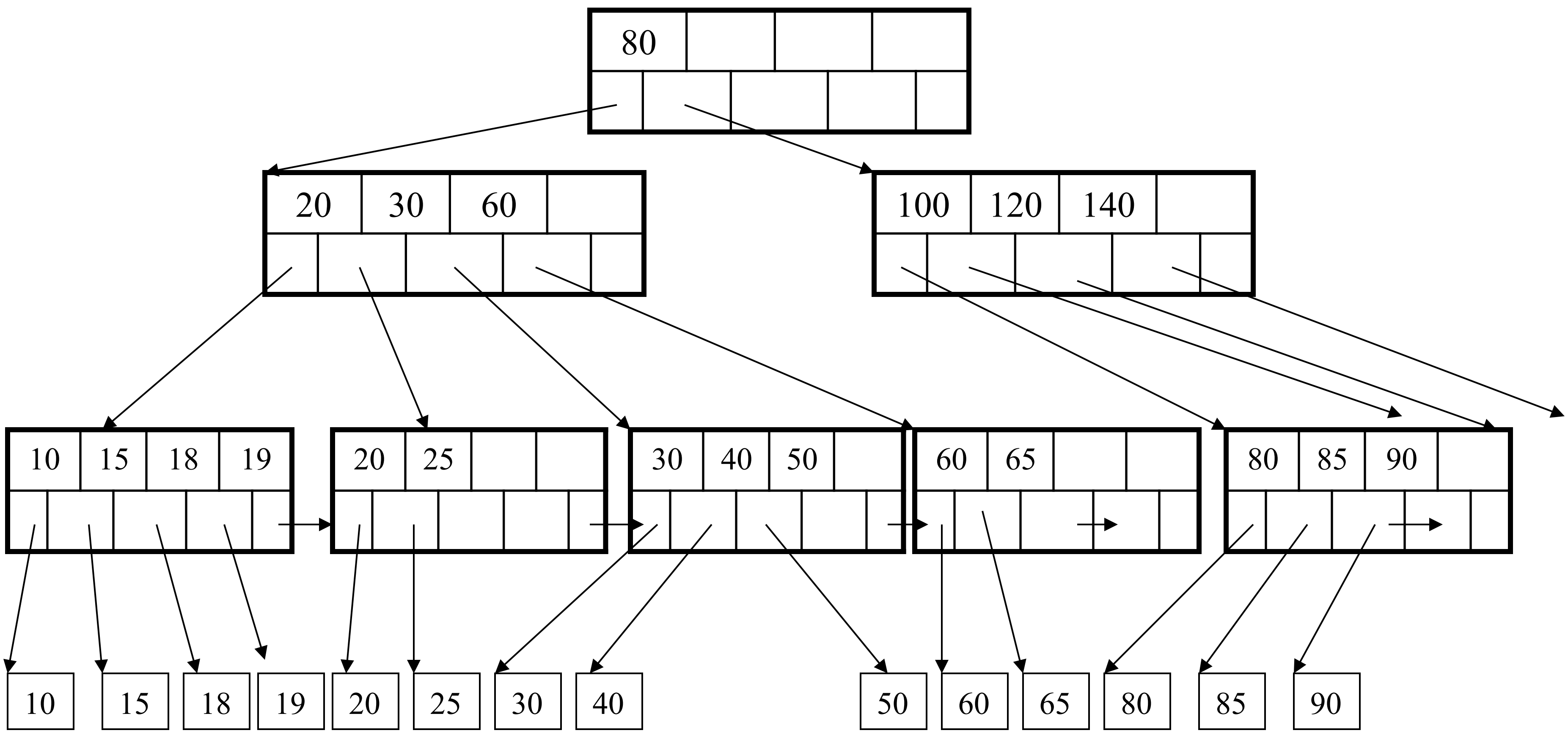
Insertion in a B+ Tree

But now have to split !



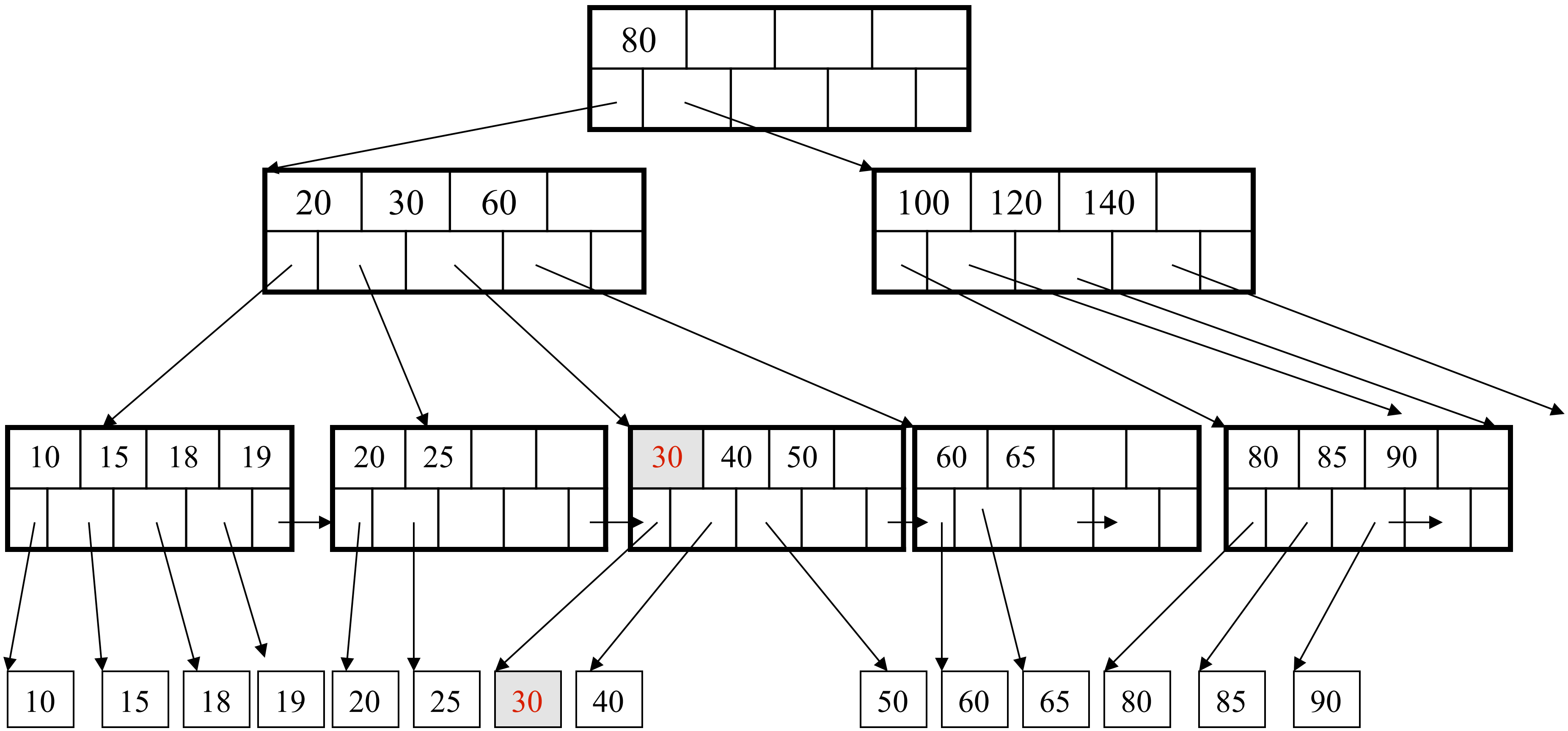
Insertion in a B+ Tree

After the split



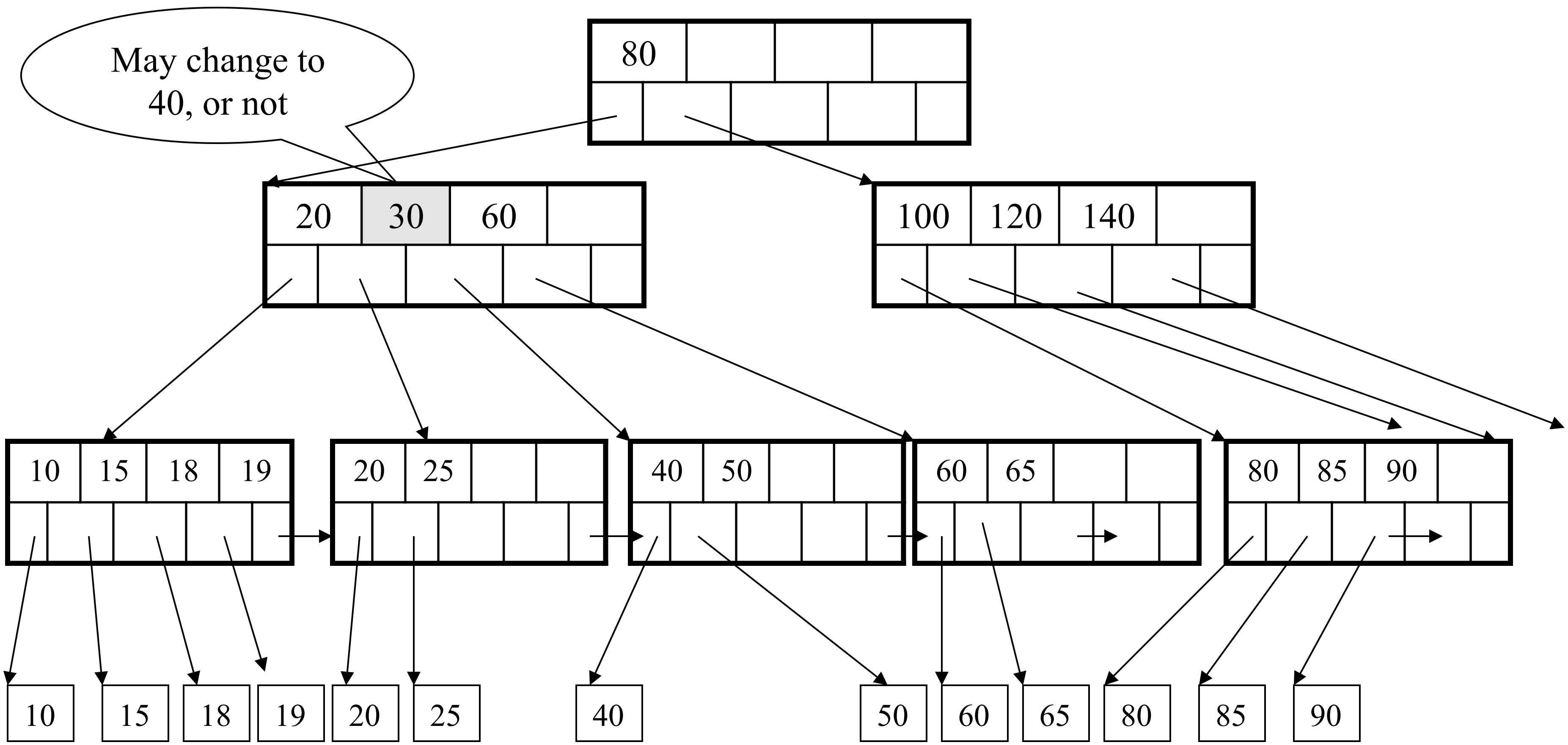
Deletion from a B+ Tree

Delete 30



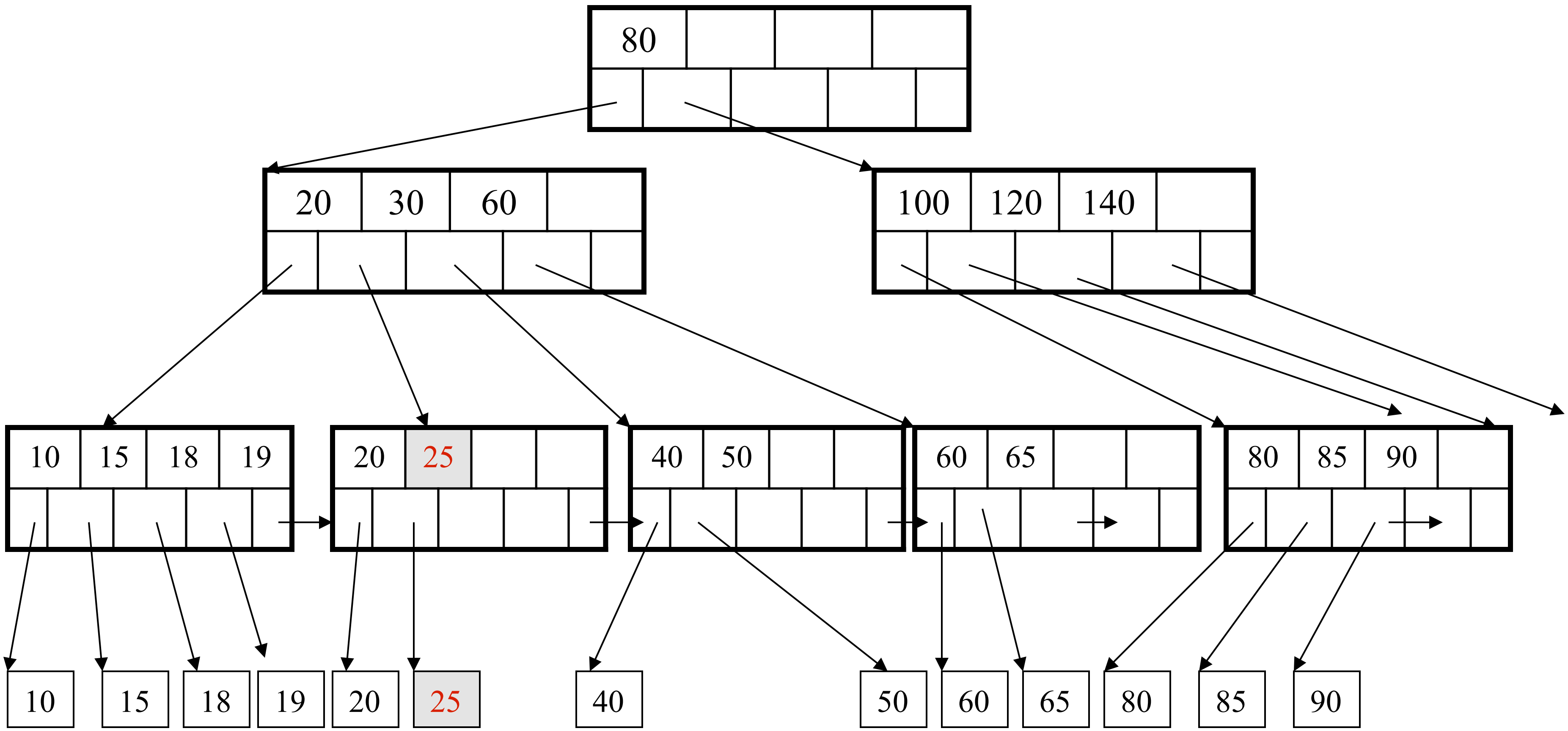
Deletion from a B+ Tree

After deleting 30



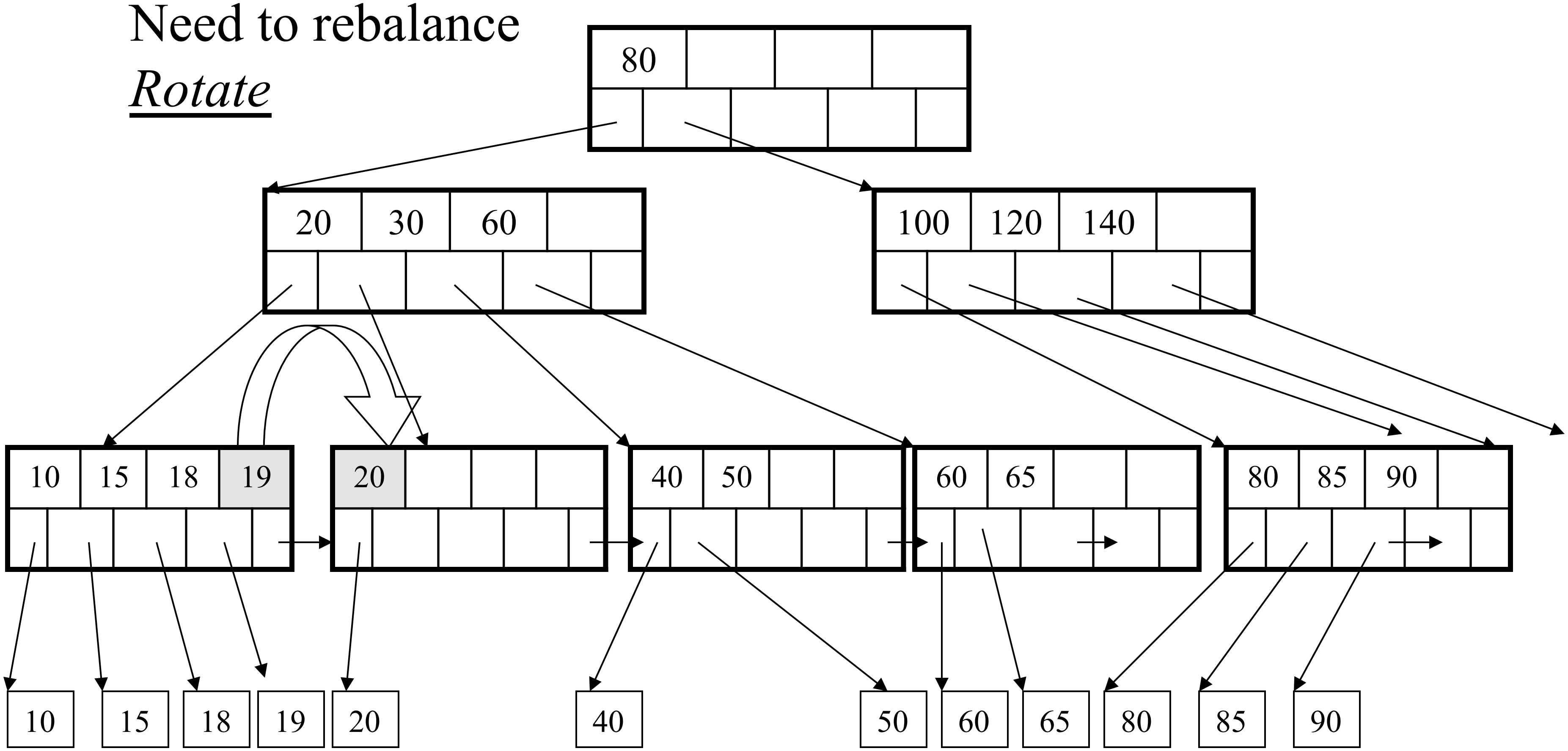
Deletion from a B+ Tree

Now delete 25



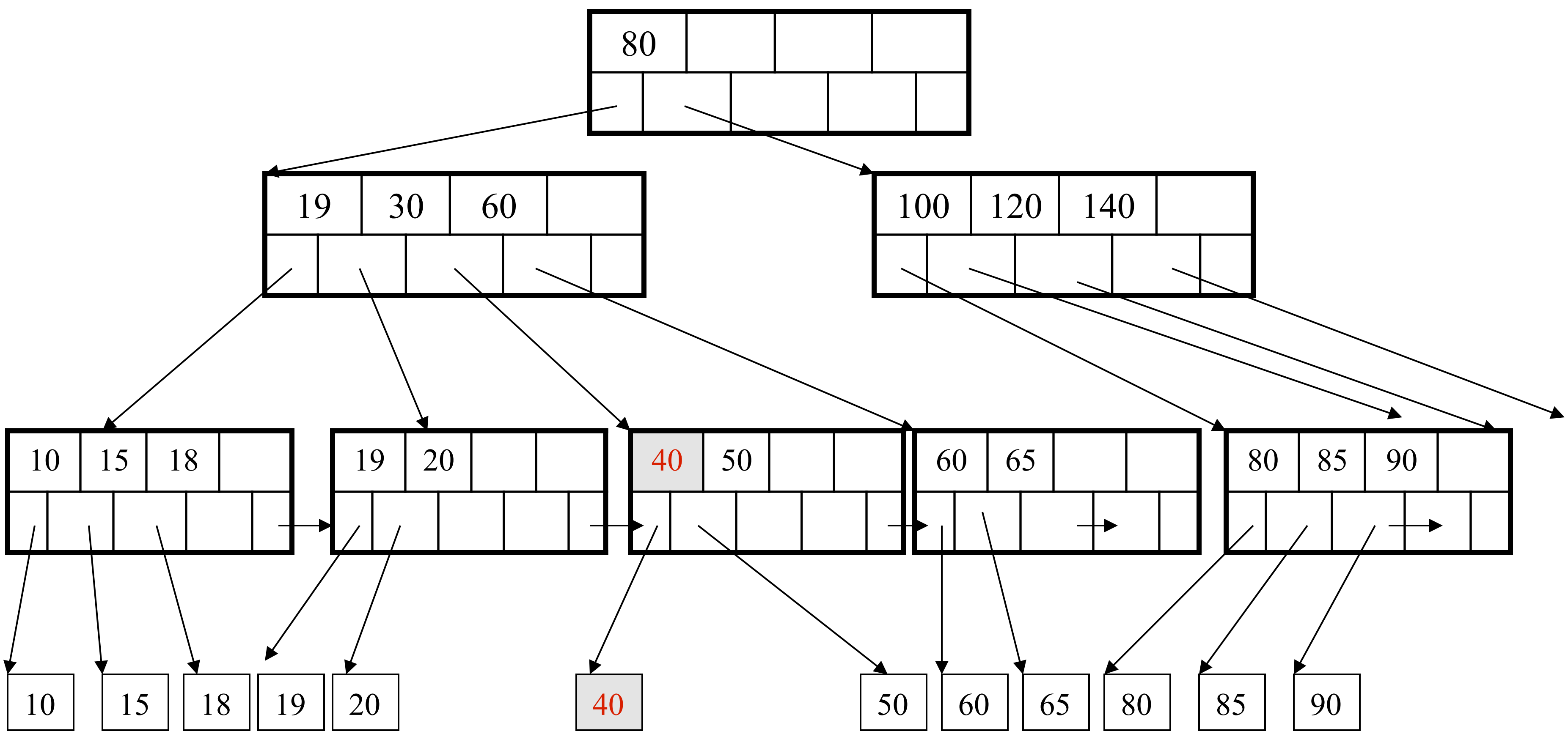
Deletion from a B+ Tree

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



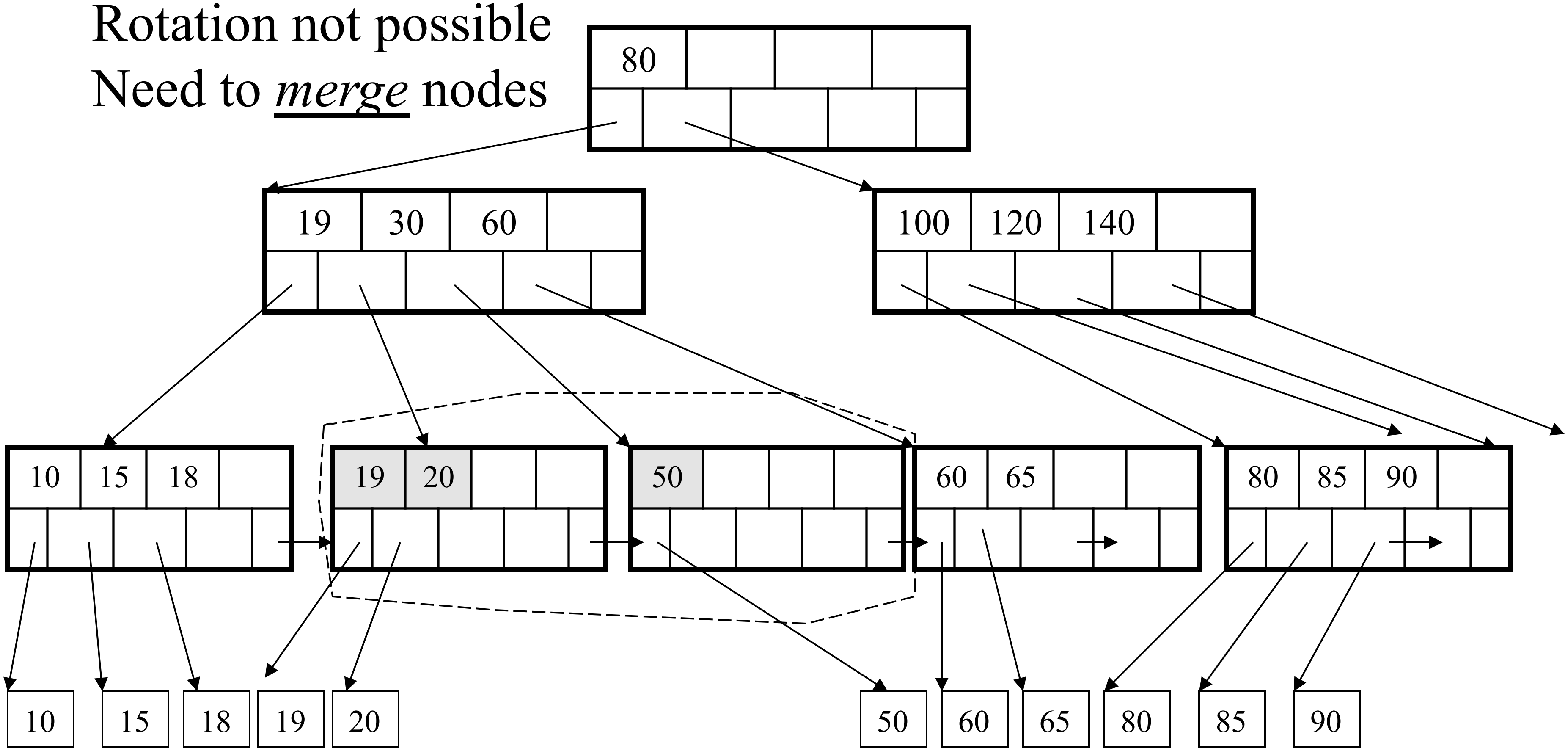
Deletion from a B+ Tree

Now delete 40



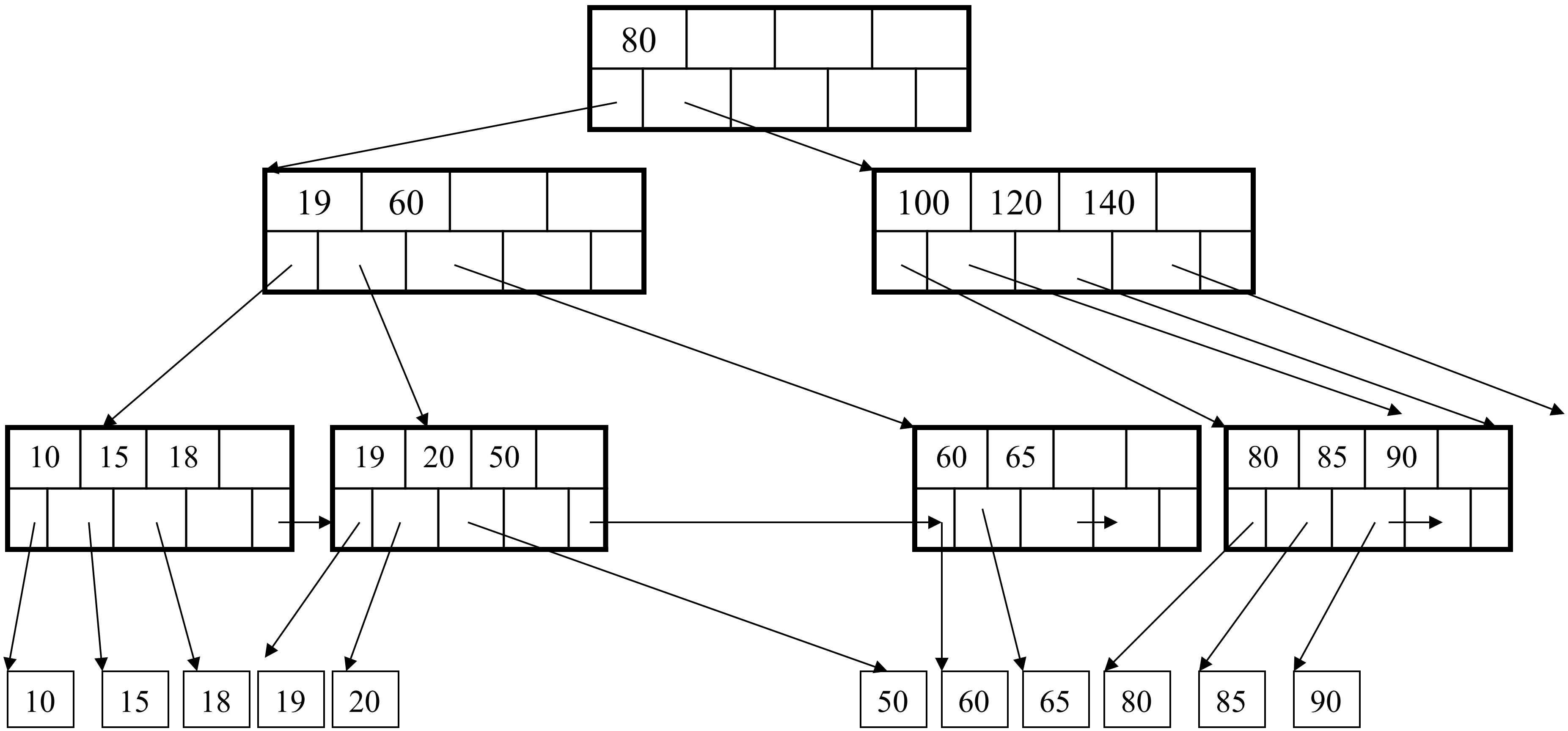
Deletion from a B+ Tree

After deleting 40
Rotation not possible
Need to *merge* nodes



Deletion from a B+ Tree

Final tree



Summary of B+ Trees

- Default index structure on most DBMS
- 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

Indexes in PostgreSQL

```
CREATE TABLE V(M int, N varchar(20), P int);
```

```
CREATE INDEX V1_N ON V(N)
```

```
CREATE INDEX V2 ON V(P, M)
```

```
CREATE INDEX VVV ON V(M, N)
```

```
CLUSTER V USING V2
```

Makes V2 clustered