Lecture 6: Storage/indexes, Tuning, Security

Tuesday, May 5, 2009

Outline

- Storage and indexing: Chapter 8, 10
- Database Tuning: Chapter 20
- Security in SQL: Chapter 21

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

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

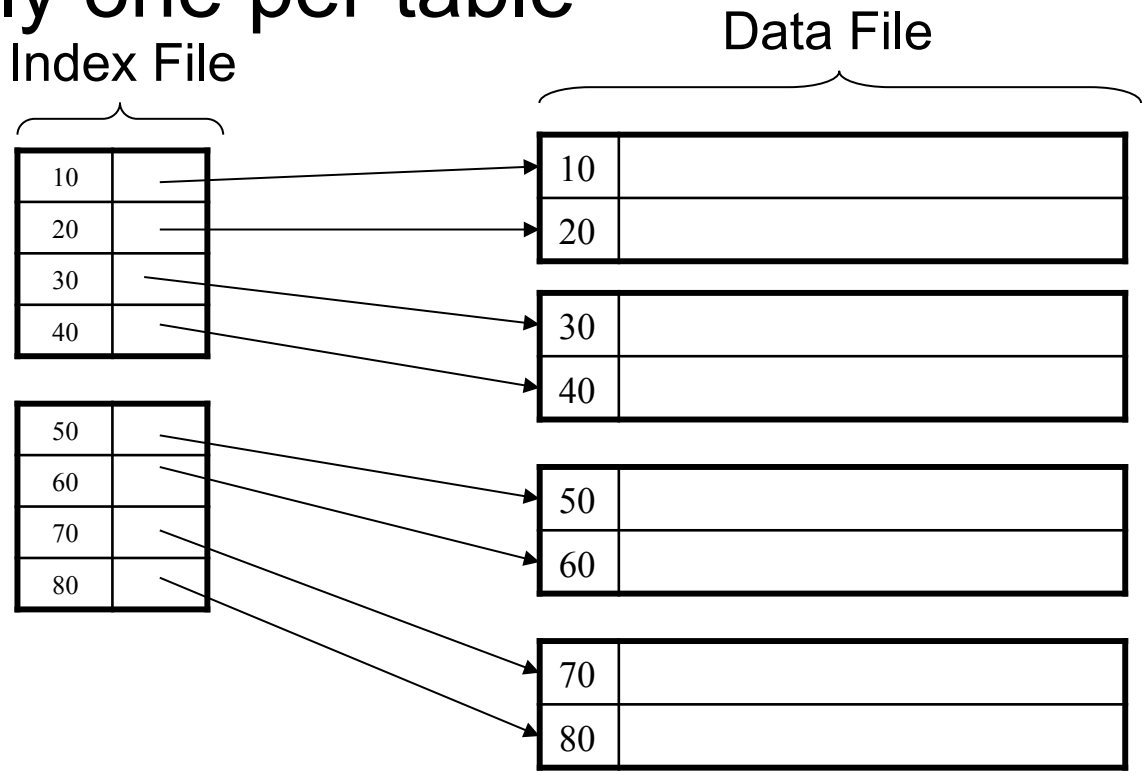
- **Organization**: B+ tree or Hash table

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

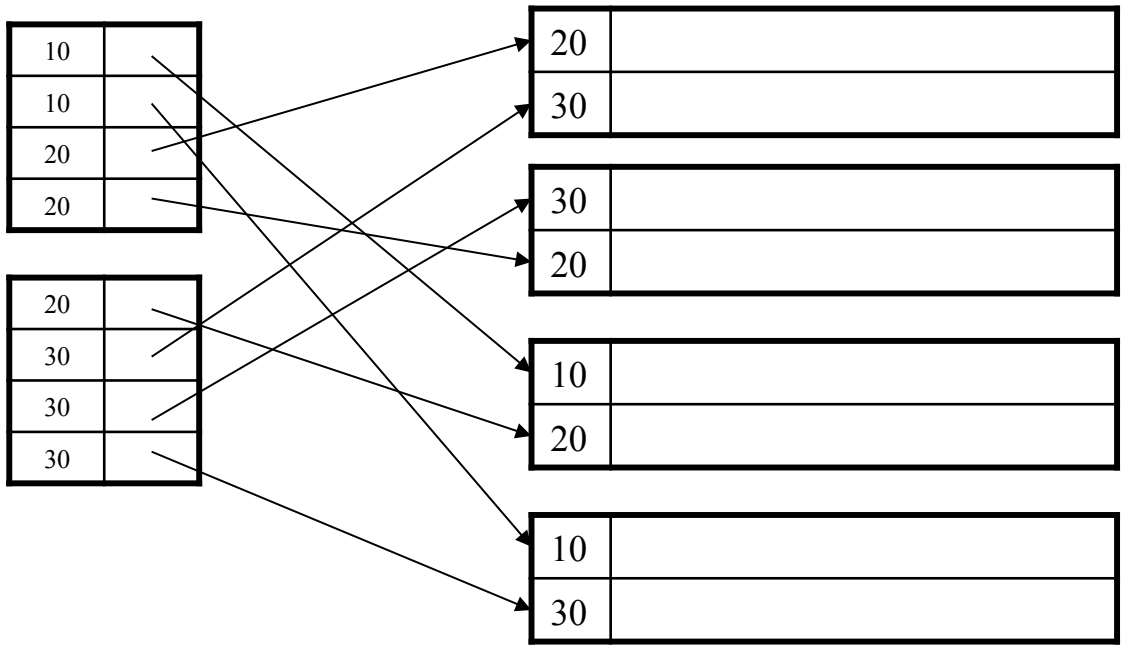
Clustered Index

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



Unclustered Index

- Several per table

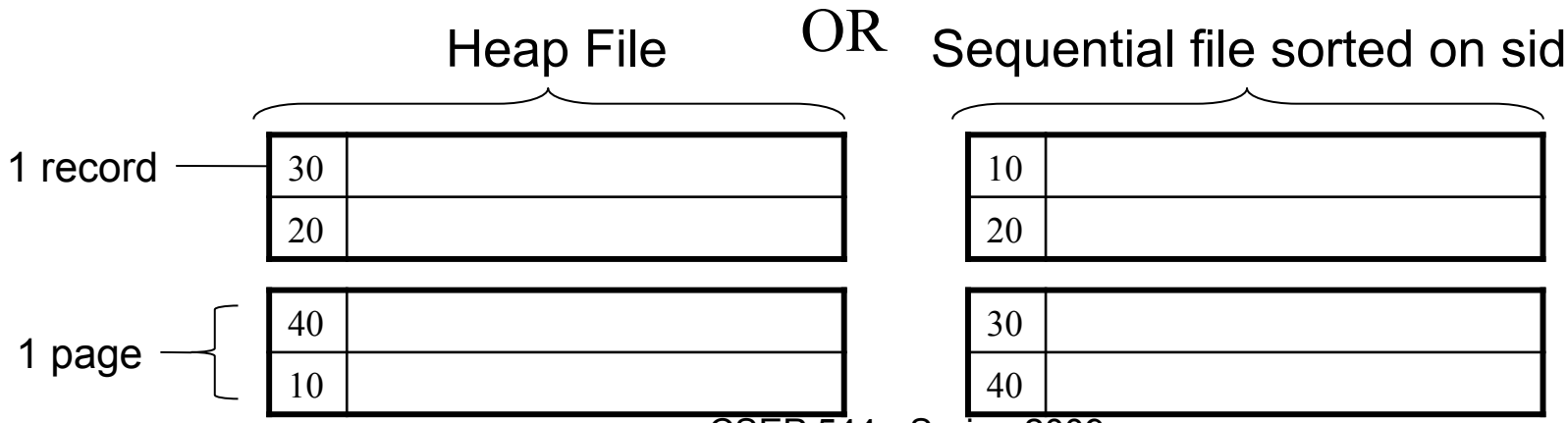


Example

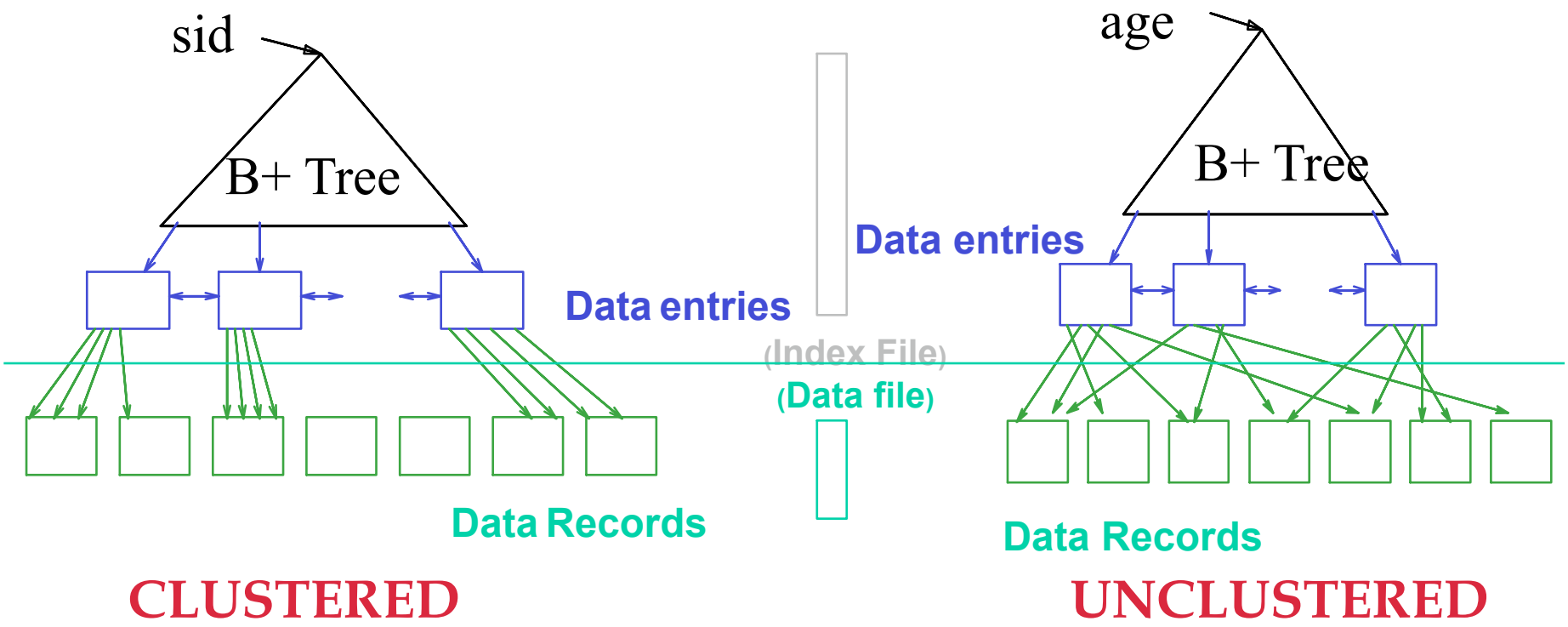
Student(sid: int, age: int, …)

Typically one relation = one file

- Heap file: tuples inside file are not sorted
- Sequential file: tuples sorted on a key



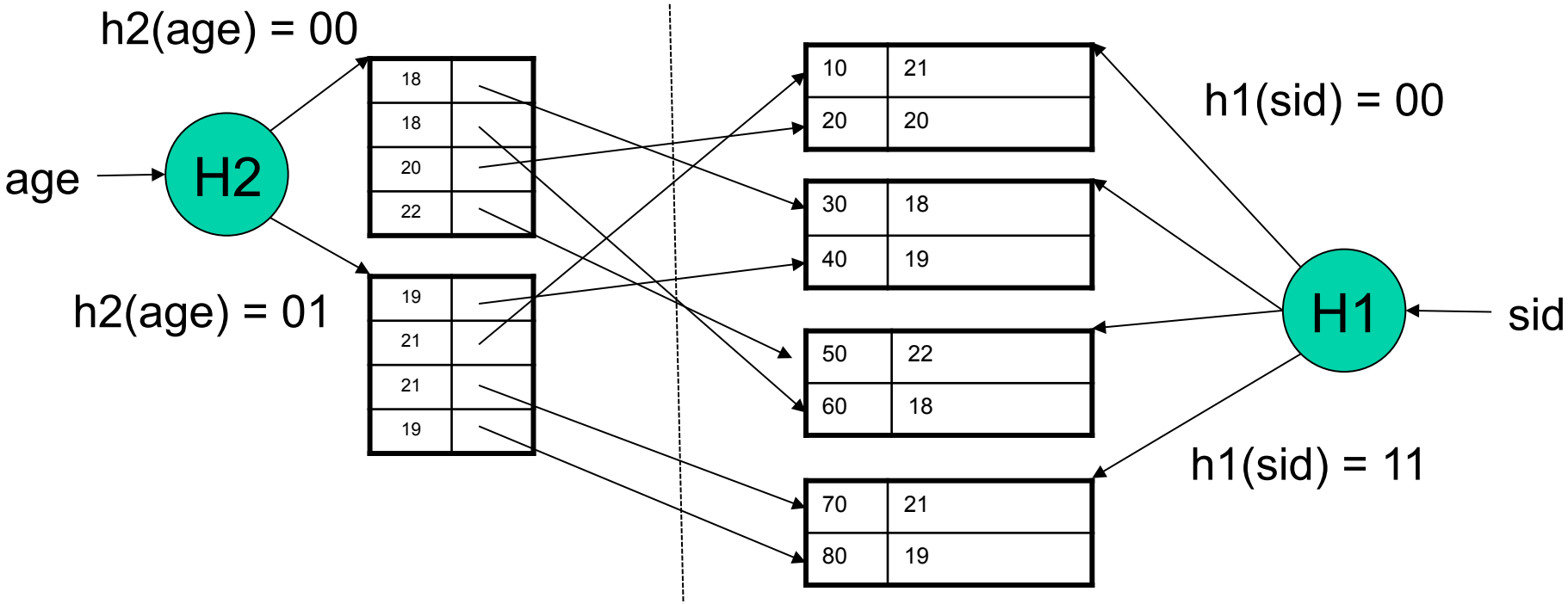
Clustered vs. Unclustered



Student(sid: int, age: int, …)

Hash-Based Index

Good for point queries but not range queries



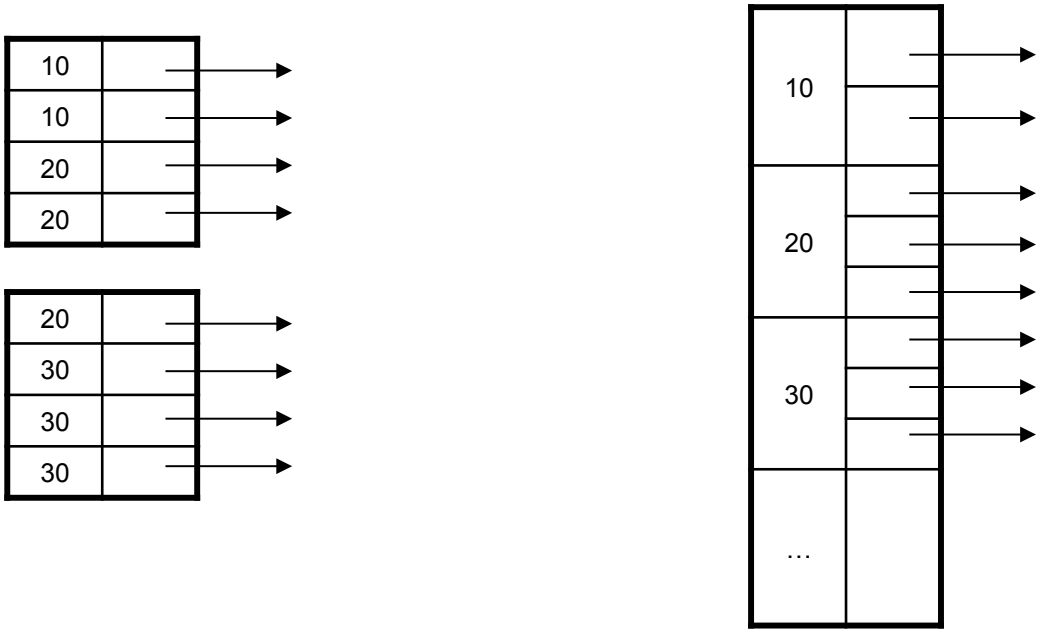
Another example
of unclustered/secondary index

Another example of
clustered/primary index

Alternatives for Data Entry k* in Index

- Three alternatives for **k***:
  - Data record with key value **k**
  - <**k**, **rid** of data record with key = **k**>
  - <**k**, list of **rids** of data records with key = **k**>
- Last two choices are orthogonal to the indexing technique used to locate data entries with a given key value k.

Alternatives 2 and 3



B+ Trees

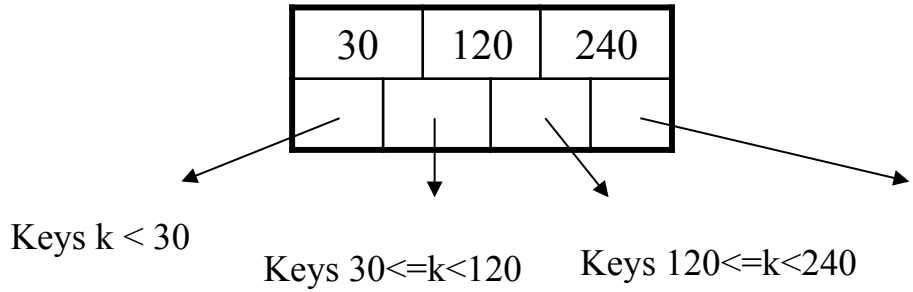
- Search trees

- 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

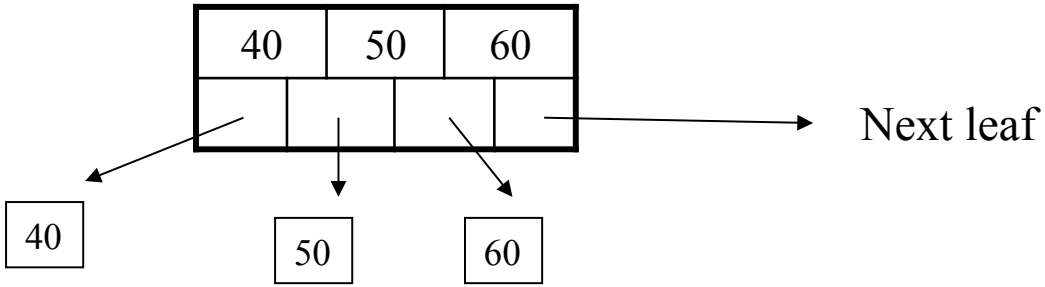
B+ Trees Basics

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

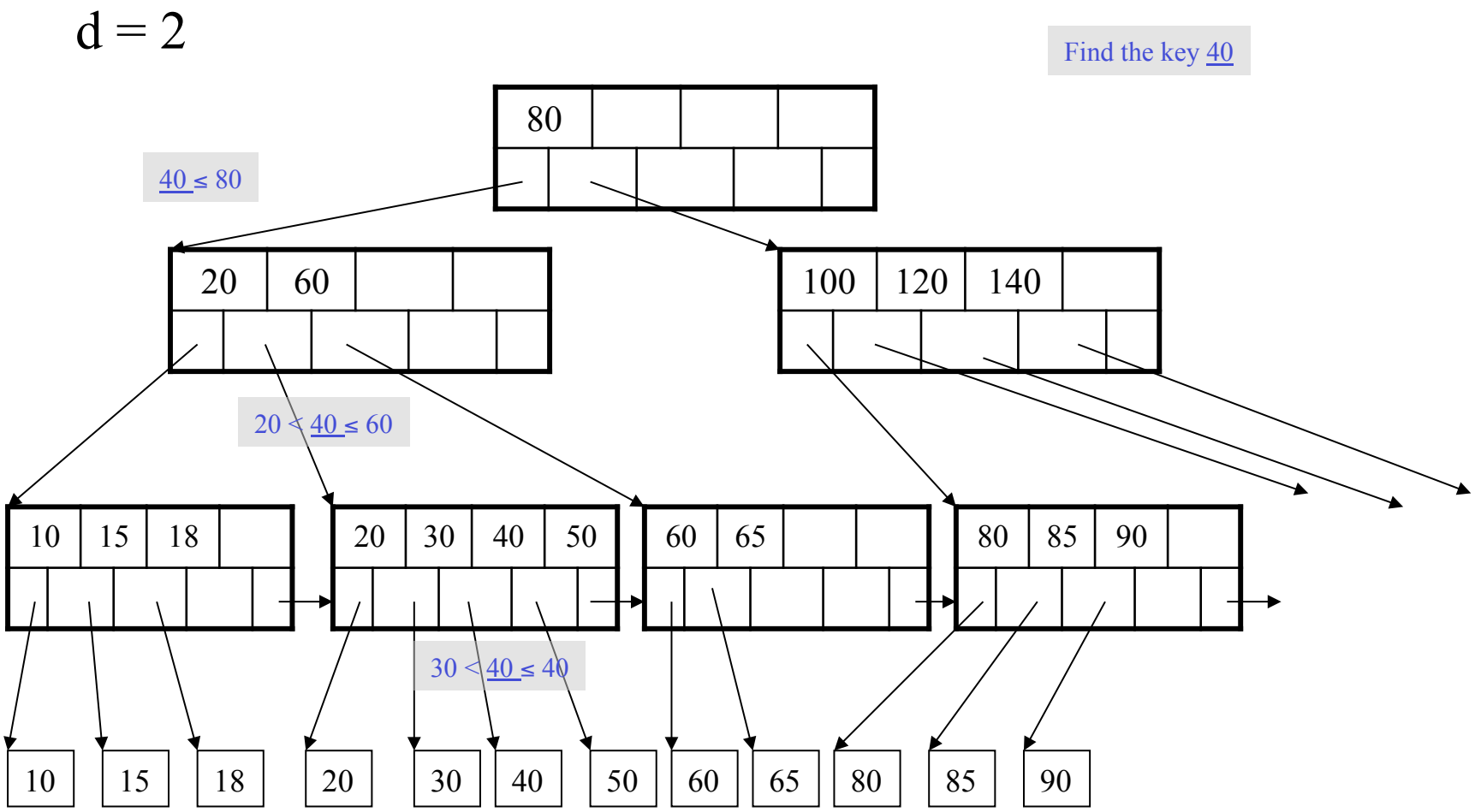


Each node also has **m+1 pointers**

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



B+ Tree Example



B+ Tree Design

- How large d ?
- Example:
  - Key size = 4 bytes
  - Pointer size = 8 bytes
  - Block size = 4096 bytes
- 2d x 4 + (2d+1) x 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
```

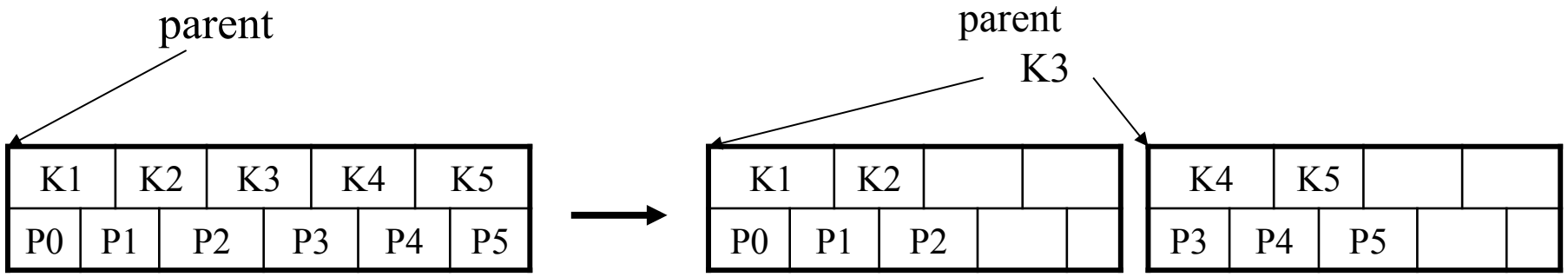
B+ Trees in Practice

- 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)

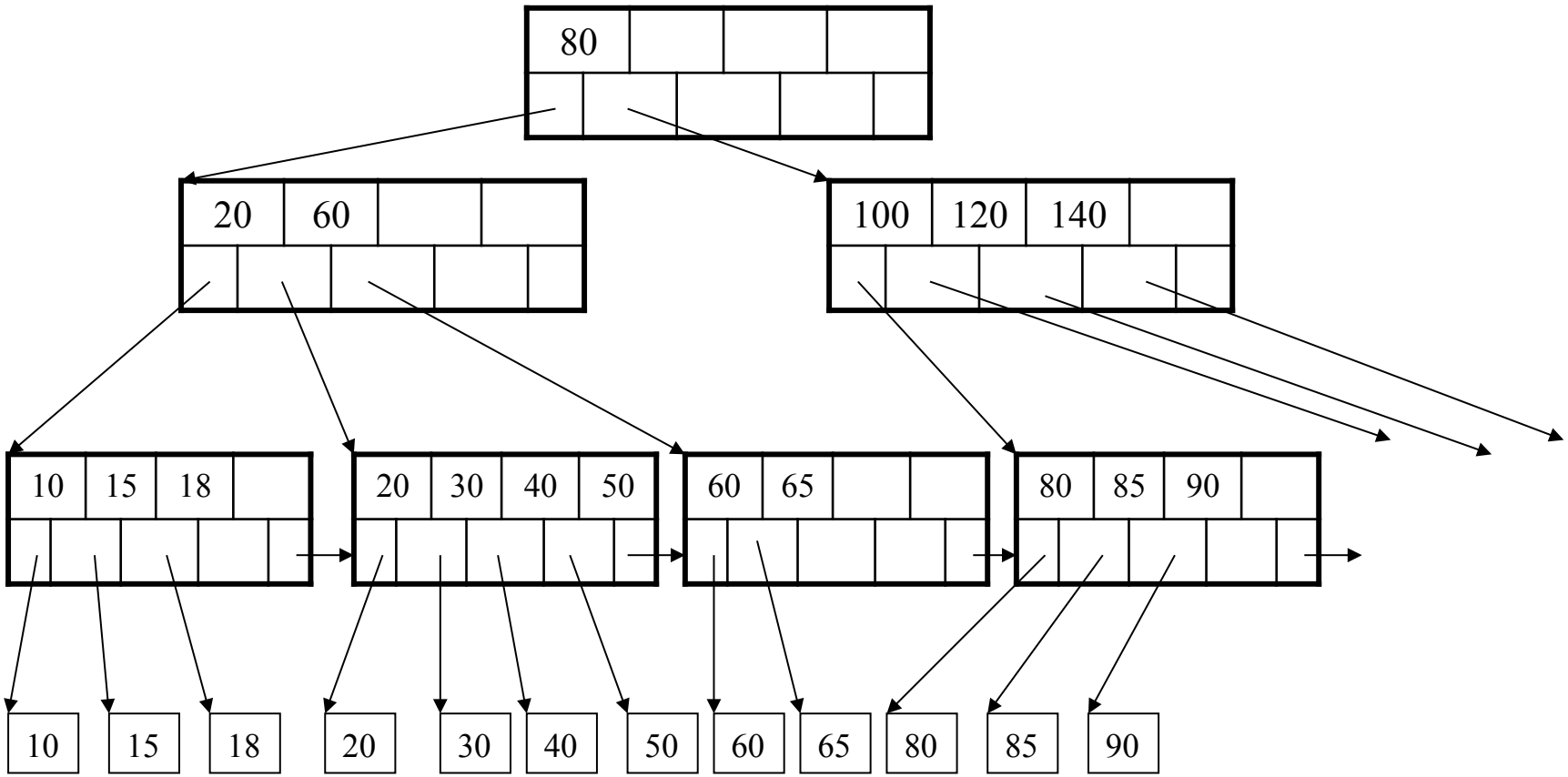
- Find leaf where K belongs, insert
- If no overflow (2d keys or less), halt
- If overflow (2d+1 keys), split node, insert in parent:



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

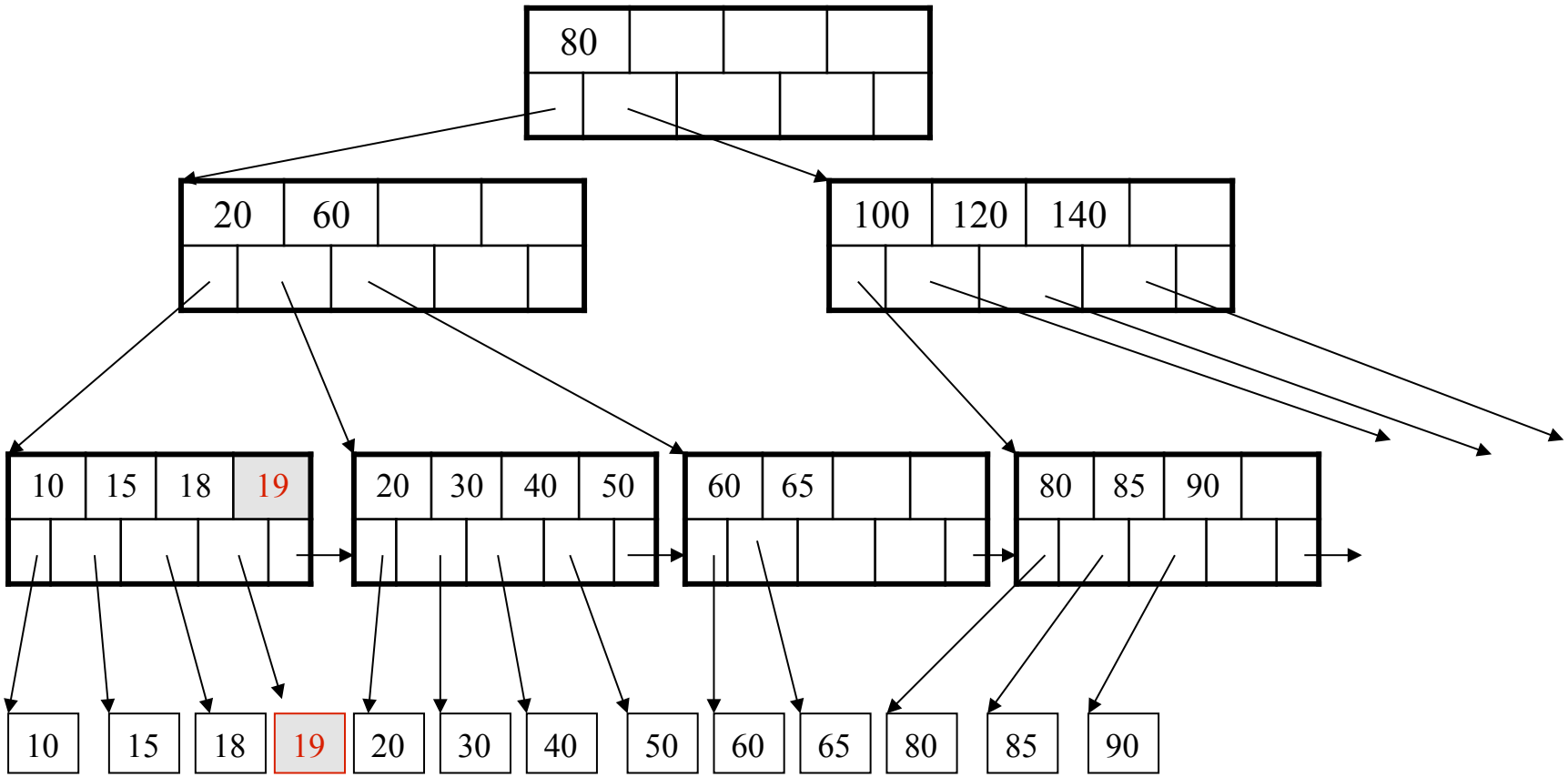
Insertion in a B+ Tree

Insert K=19



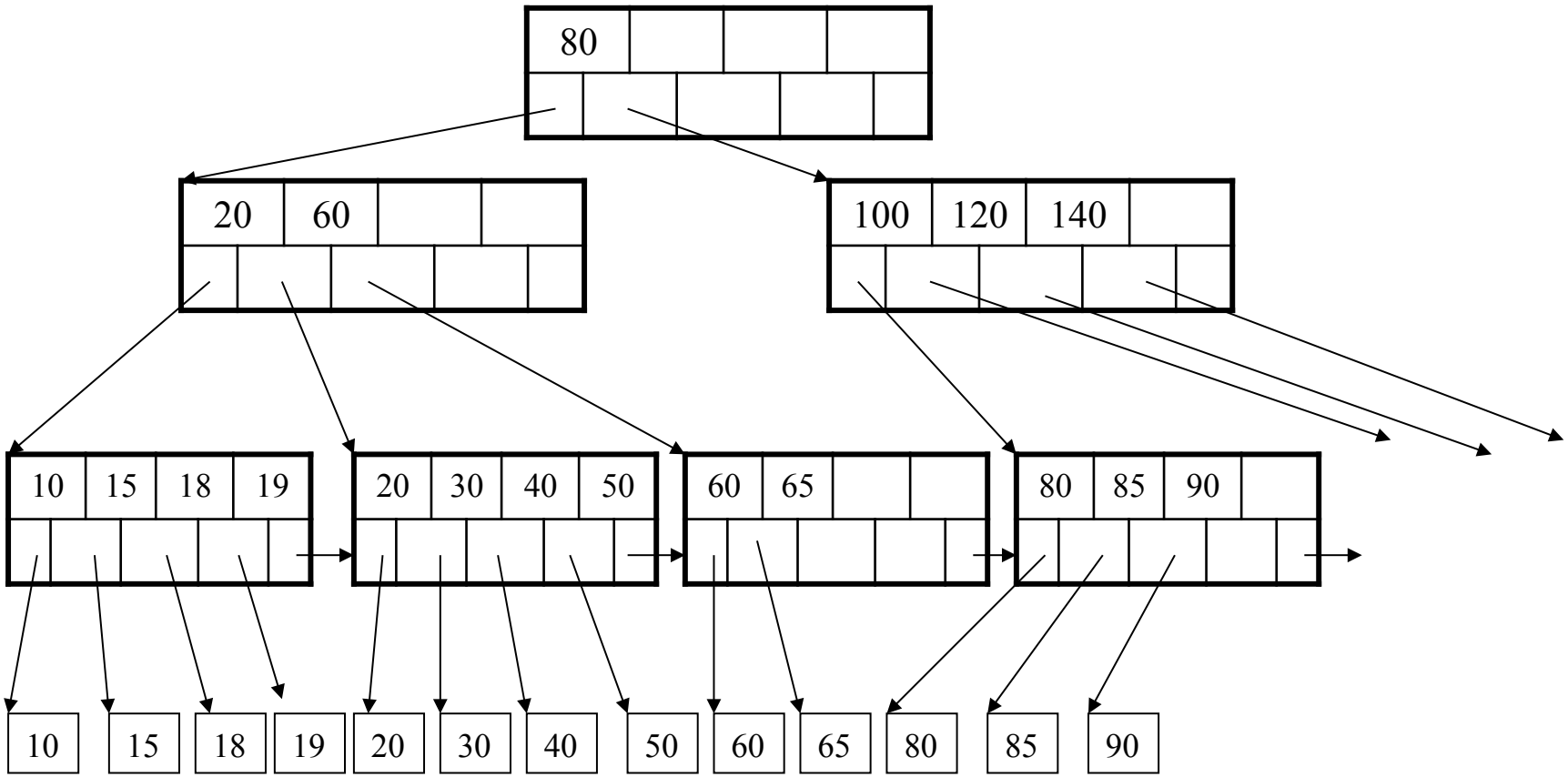
Insertion in a B+ Tree

After insertion



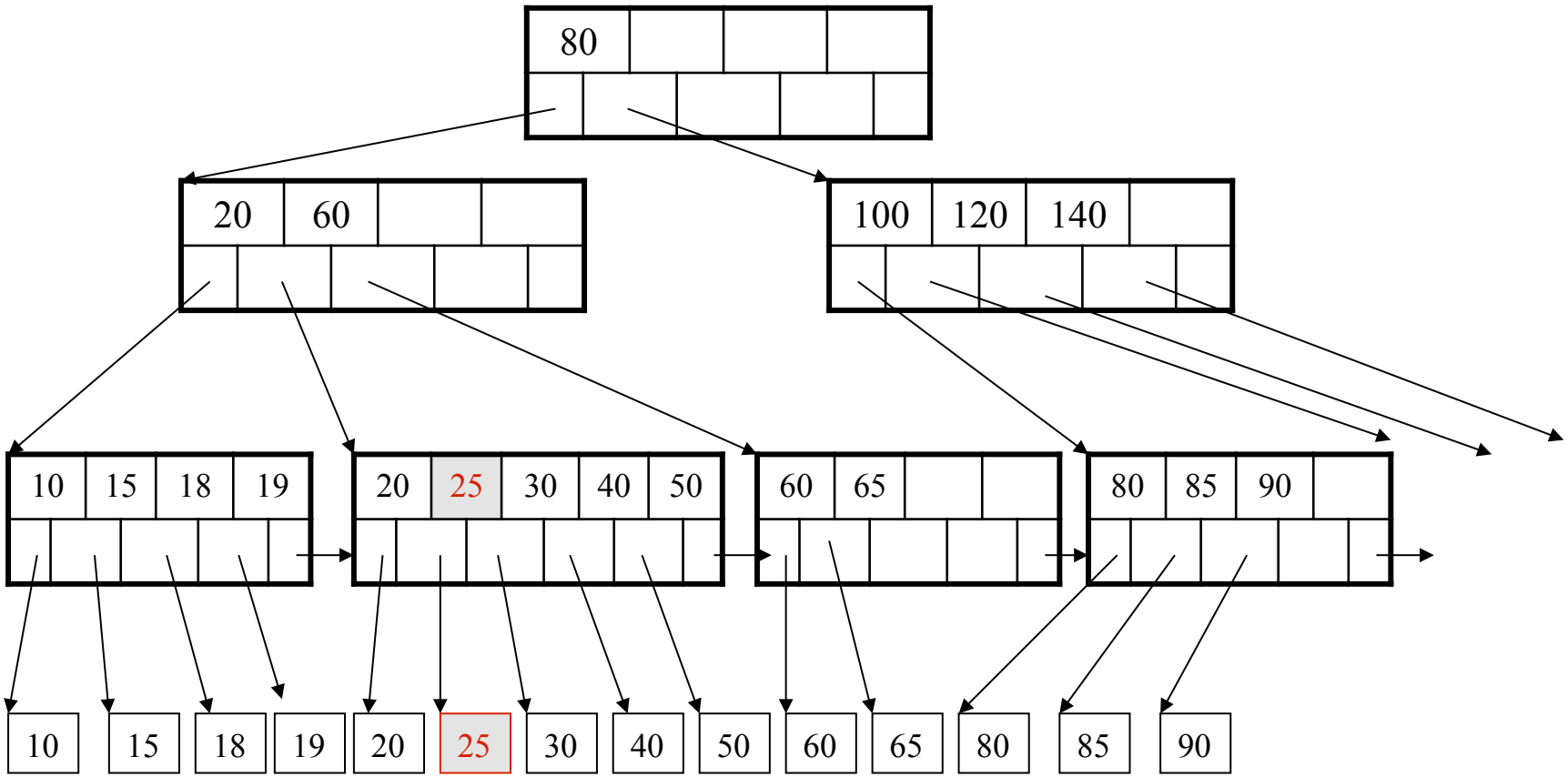
Insertion in a B+ Tree

Now insert 25



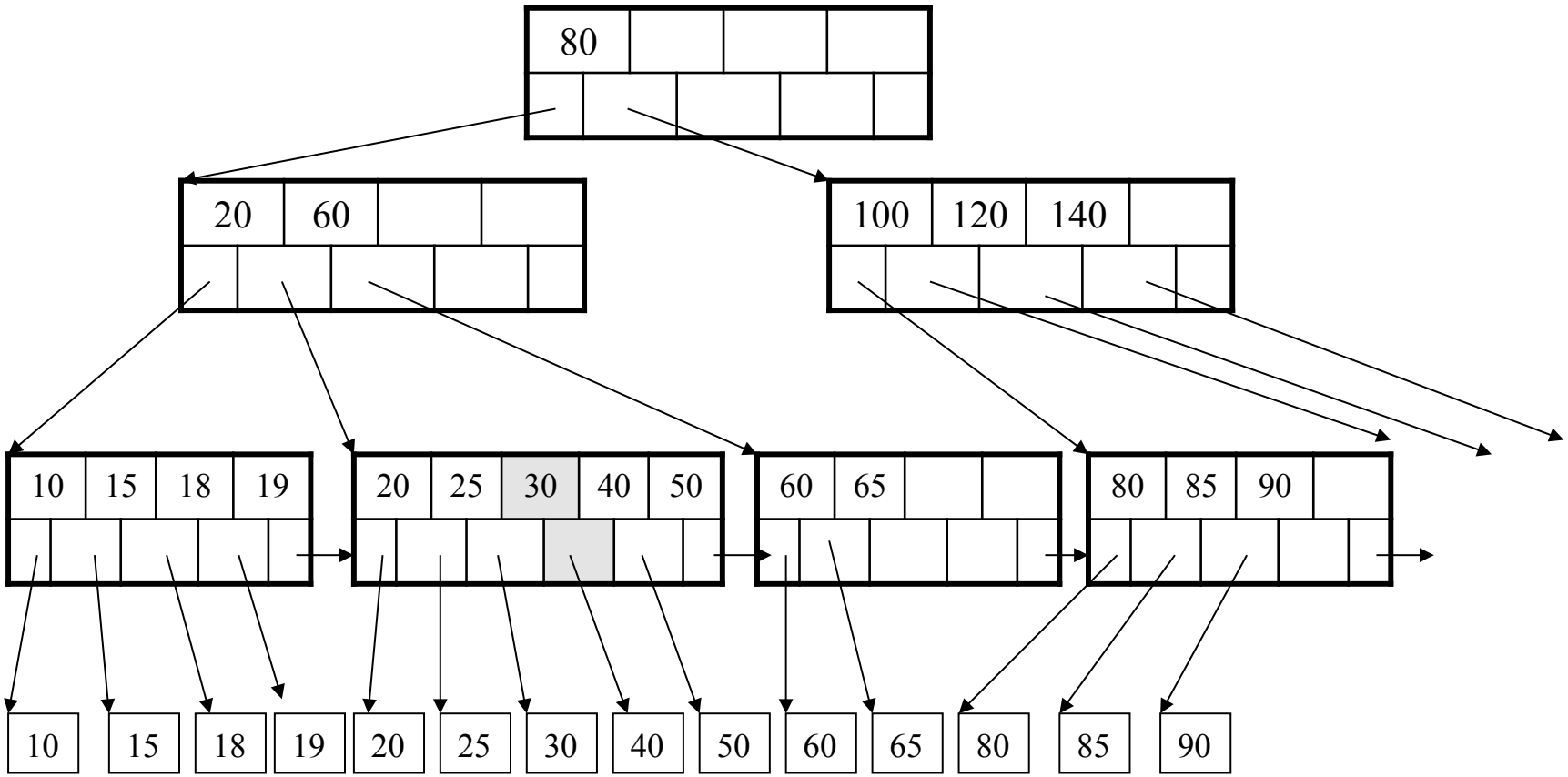
Insertion in a B+ Tree

After insertion



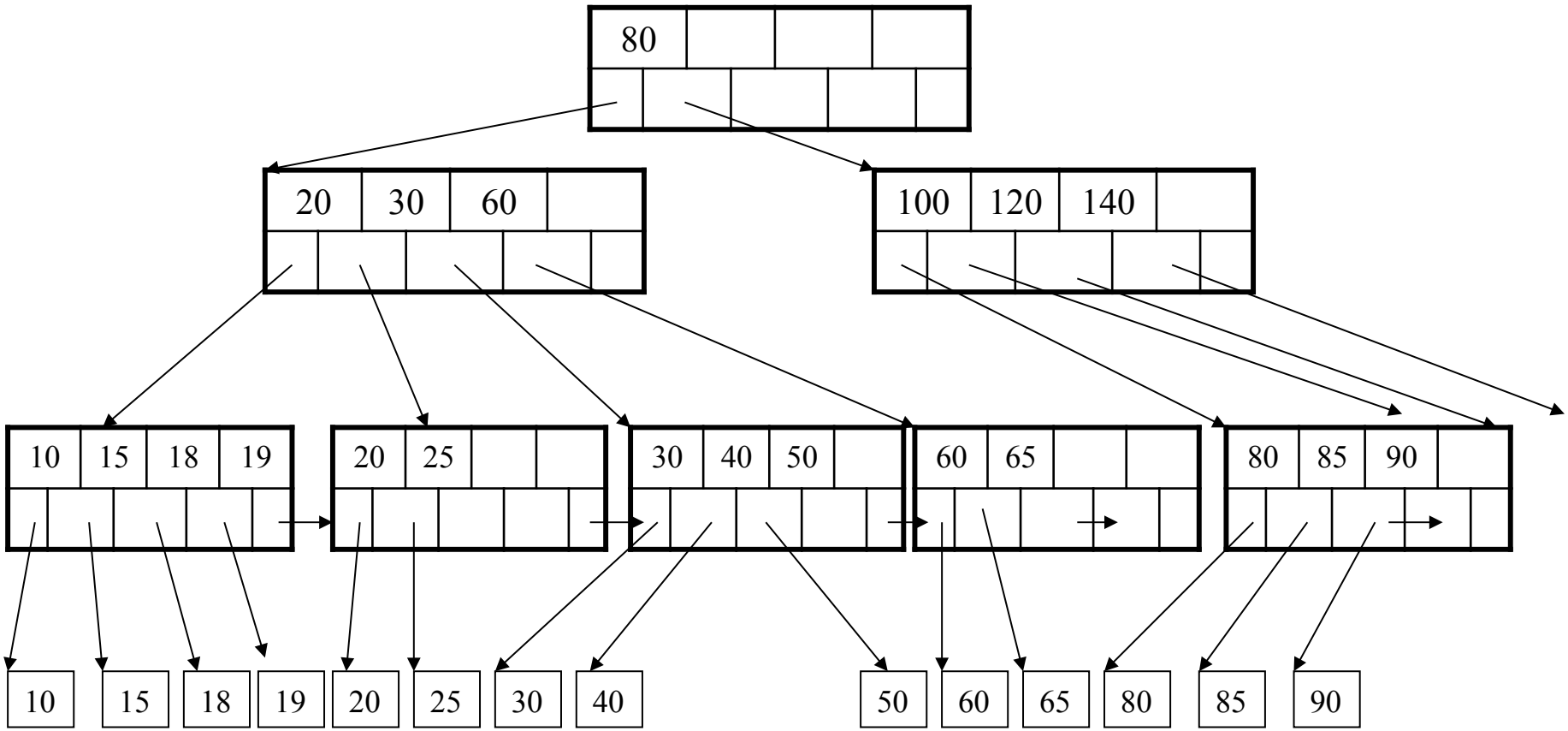
Insertion in a B+ Tree

But now have to split !



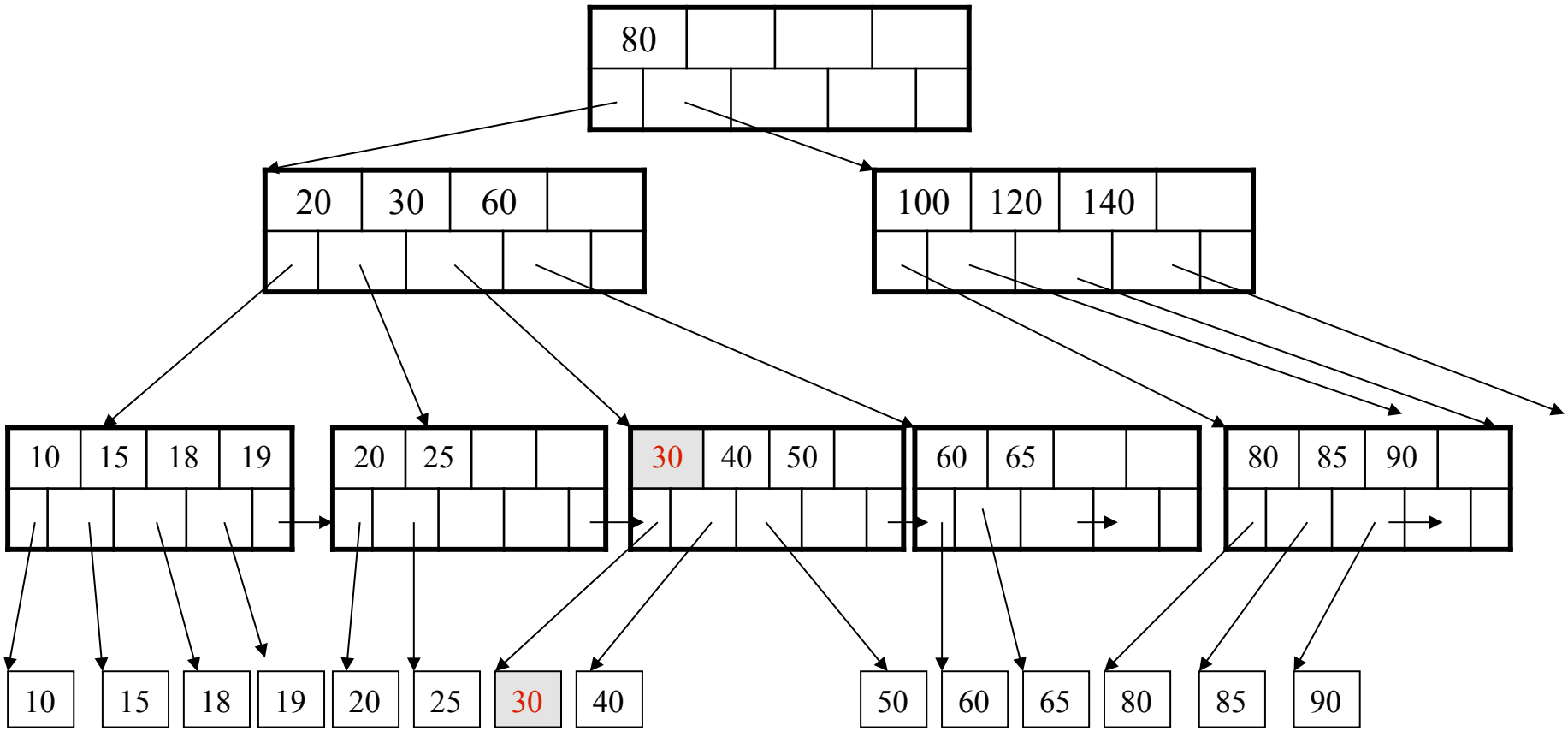
Insertion in a B+ Tree

After the split



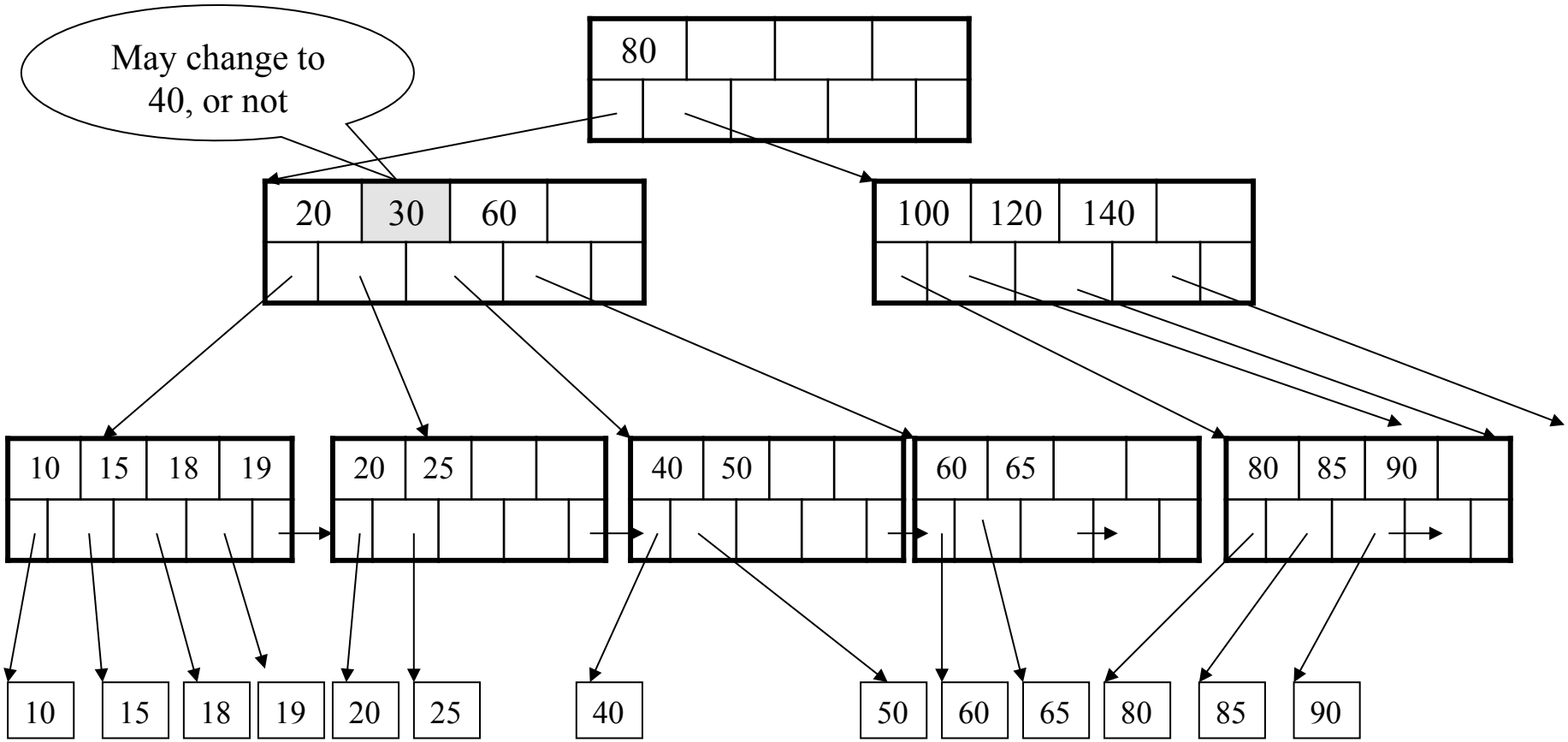
Deletion from a B+ Tree

Delete 30



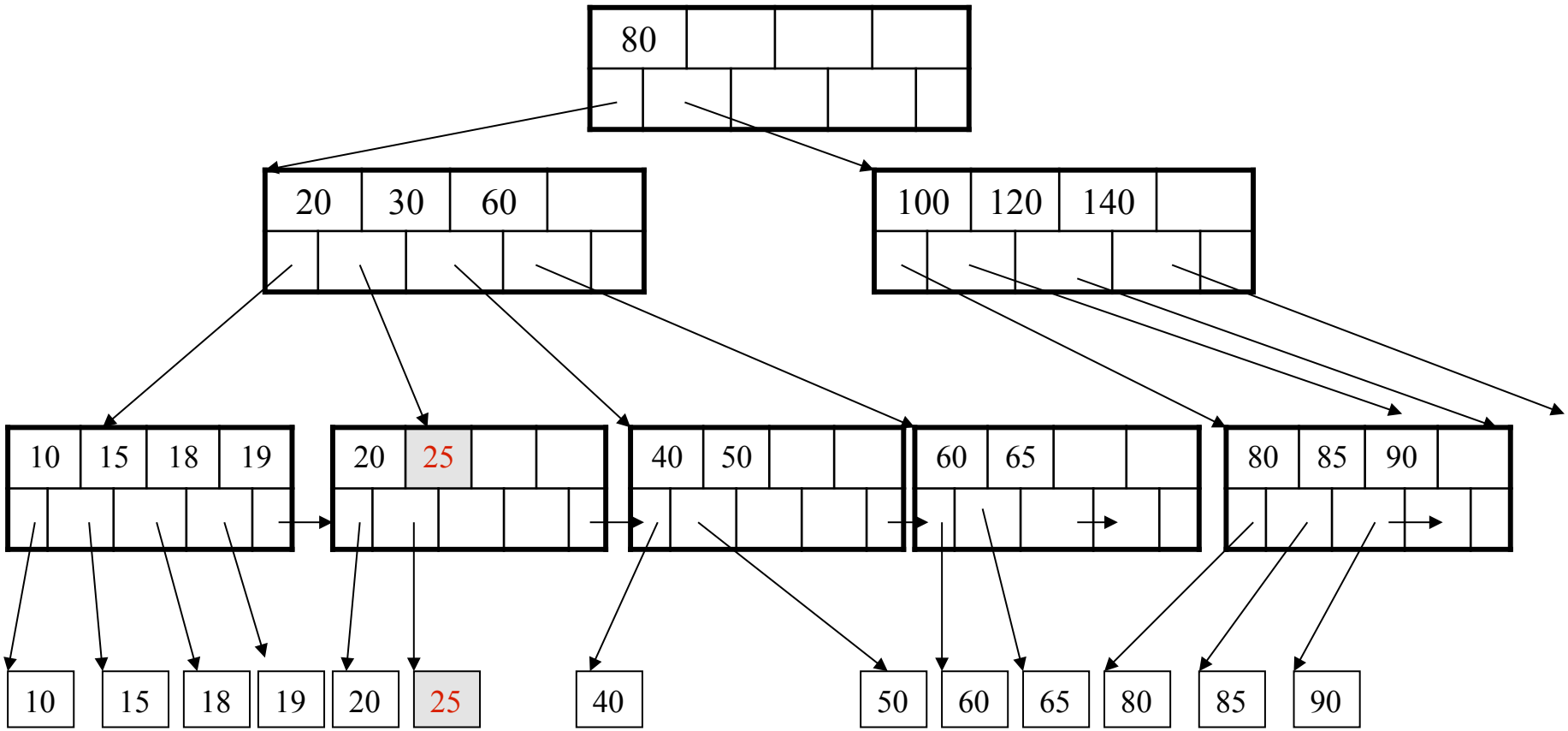
Deletion from a B+ Tree

After deleting 30



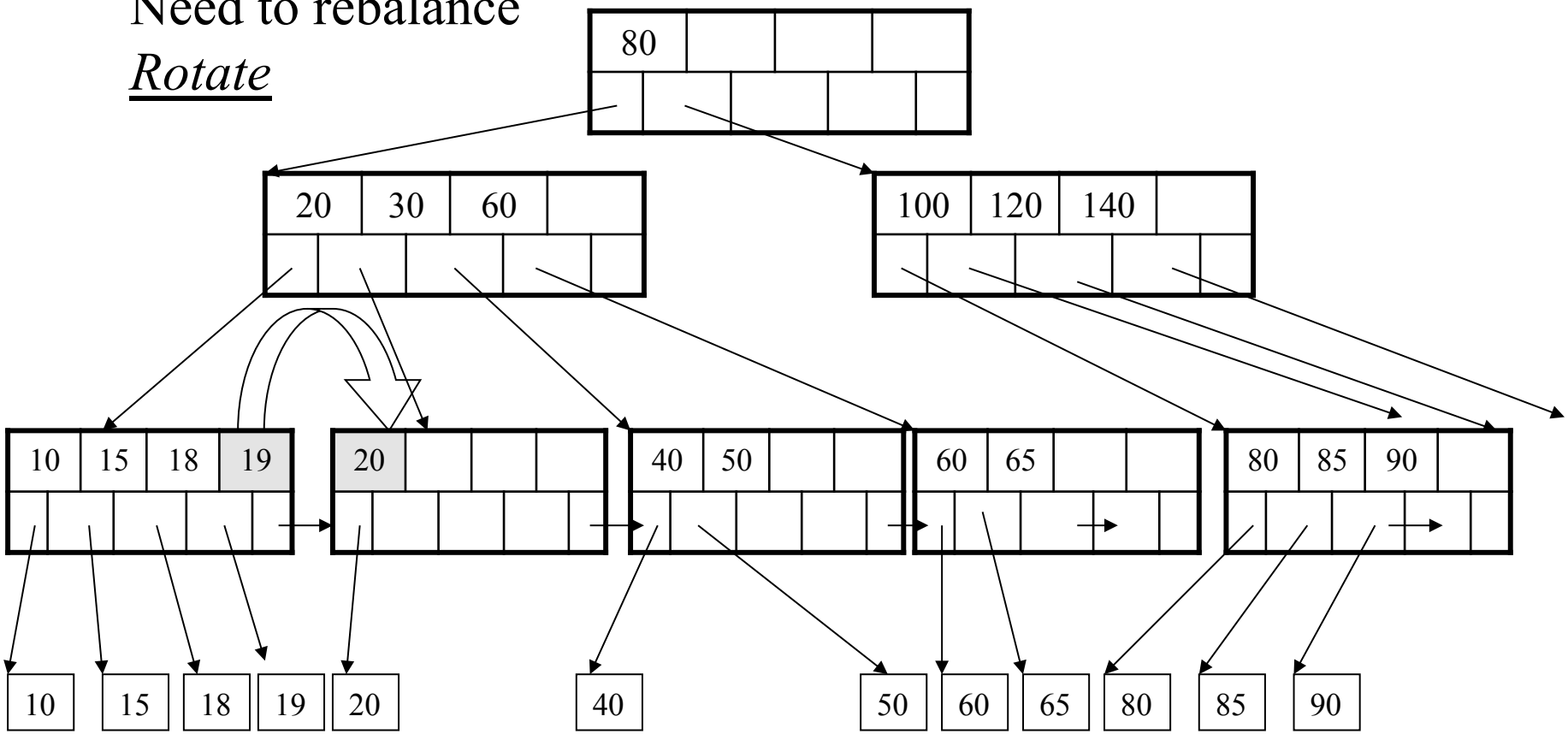
Deletion from a B+ Tree

Now delete 25



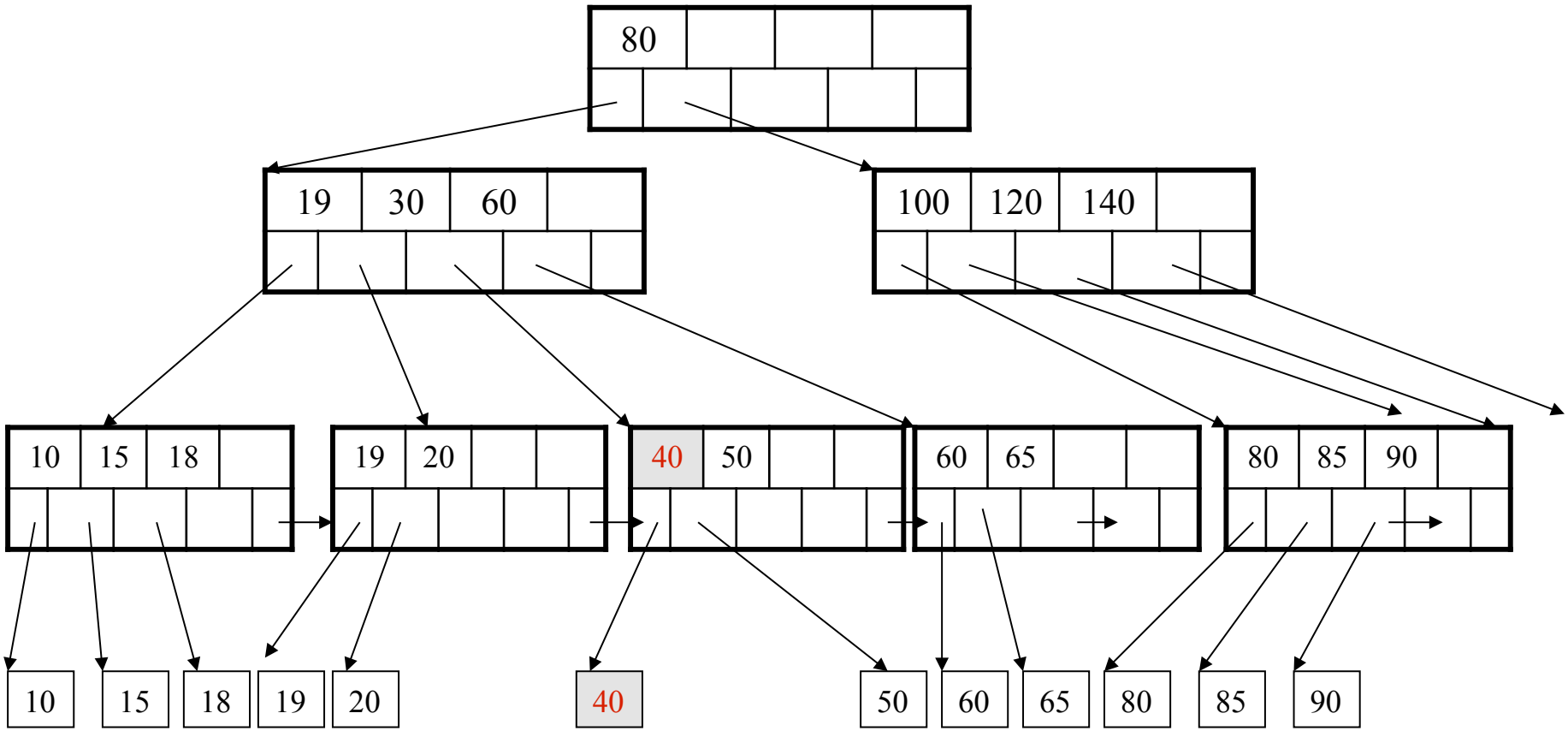
Deletion from a B+ Tree

After deleting 25
Need to rebalance
Rotate



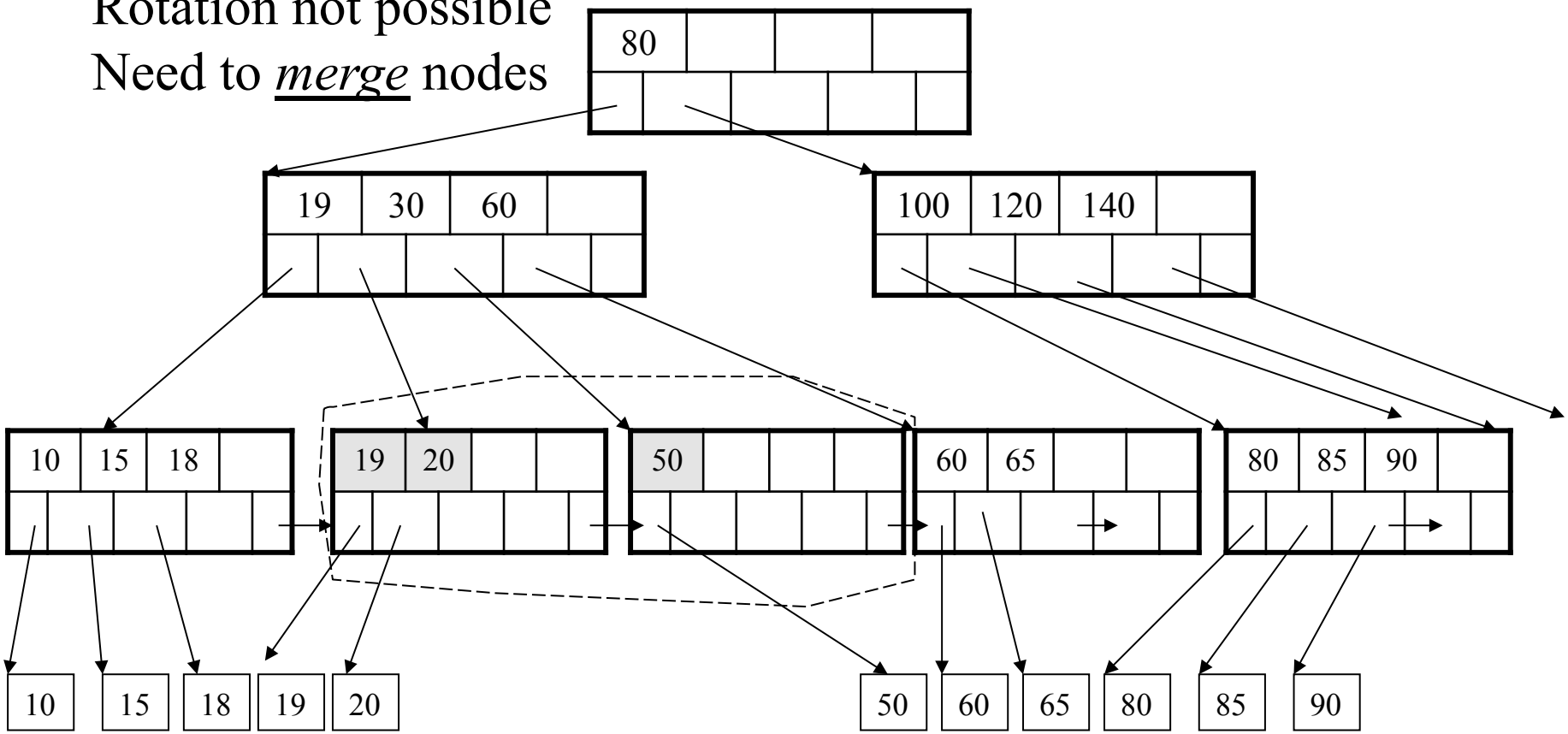
Deletion from a B+ Tree

Now delete 40



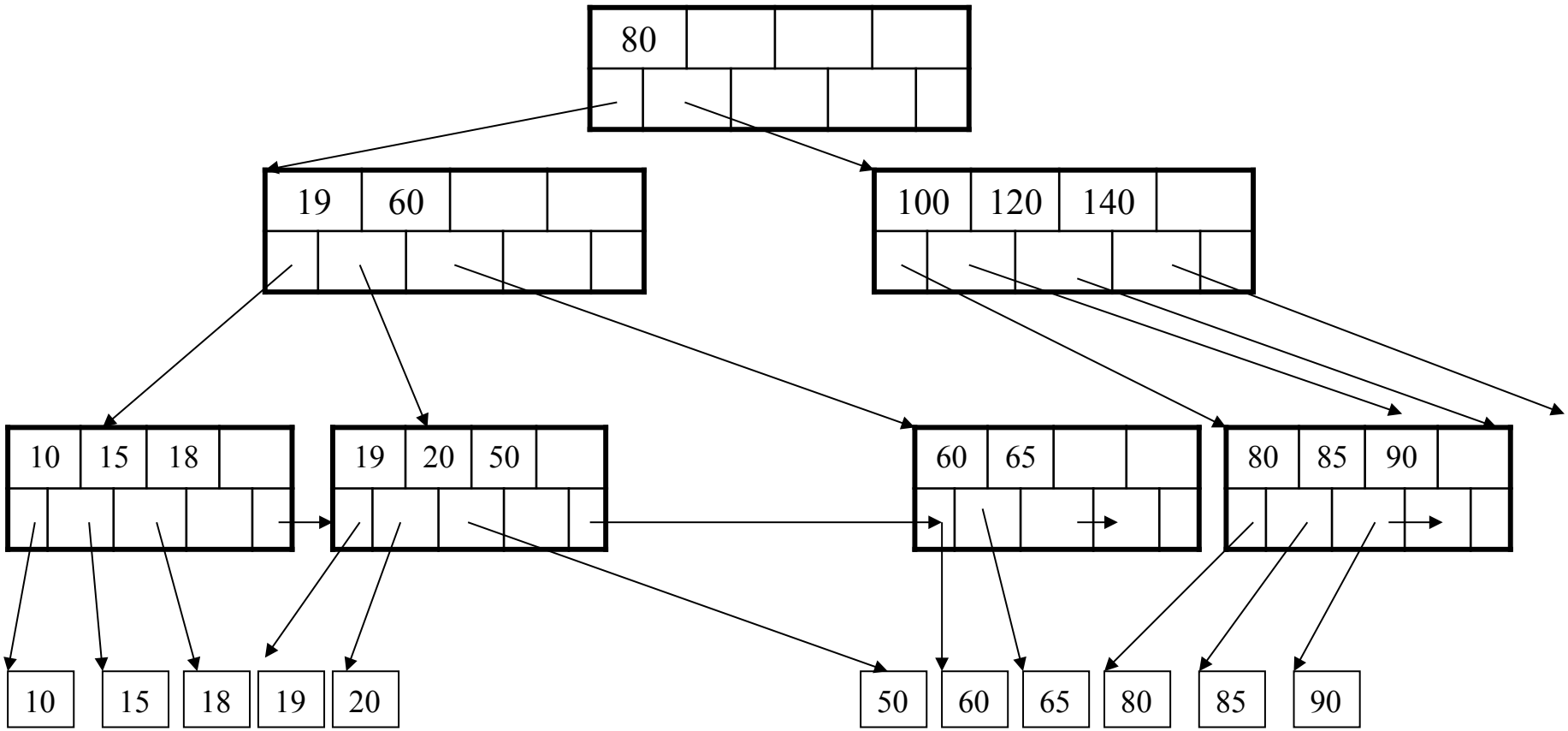
Deletion from a B+ Tree

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)
```

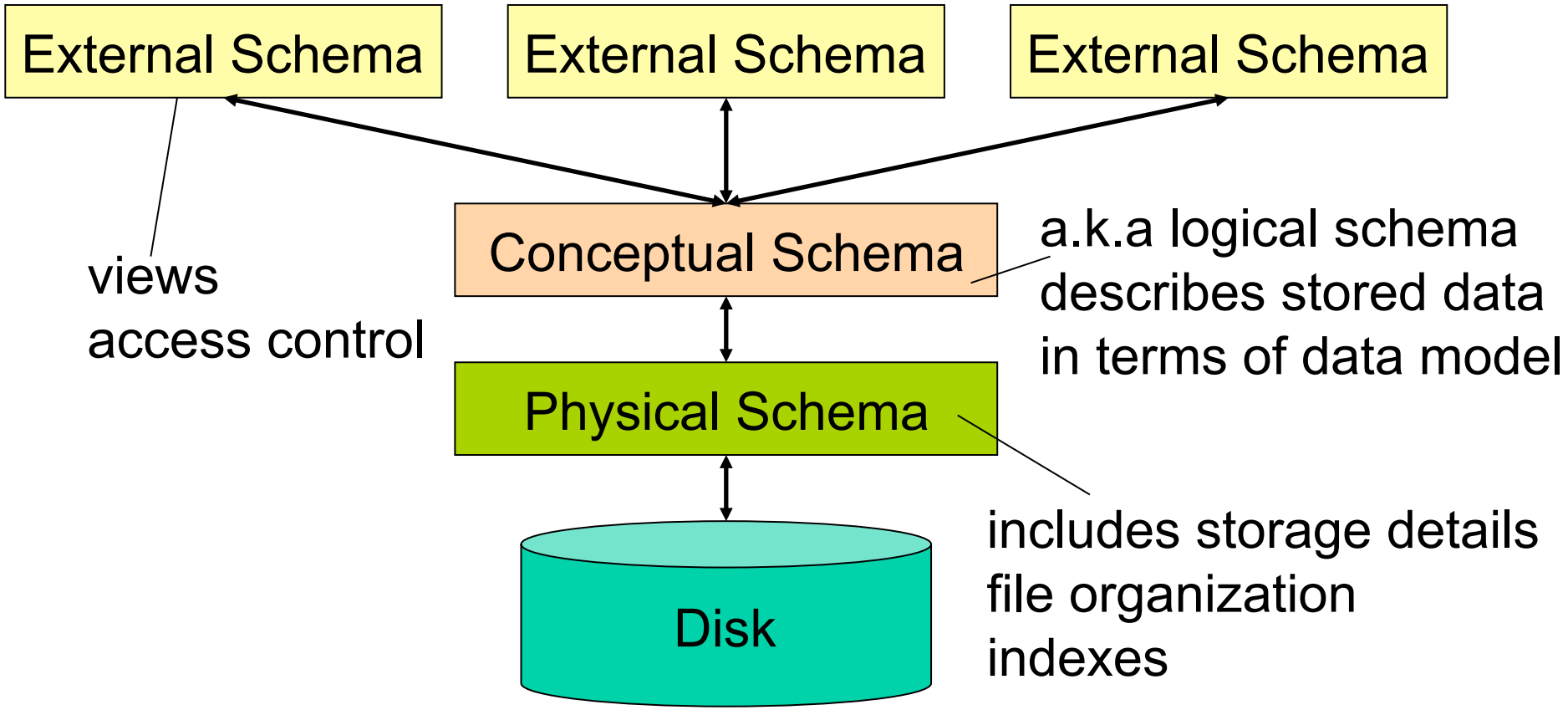
```
CLUSTER V USING V2
```

Makes V2 clustered

Database Tuning Overview

- The database tuning problem
- Index selection (discuss in detail)
- Horizontal/vertical partitioning (see lecture 3)
- Denormalization (discuss briefly)

Levels of Abstraction in a DBMS



The Database Tuning Problem

- We are given a workload description
  - List of queries and their frequencies
  - List of updates and their frequencies
  - Performance goals for each type of query
- Perform *physical database design*
  - Choice of indexes
  - Tuning the conceptual schema
    - Denormalization, vertical and horizontal partition
  - Query and transaction tuning

The Index Selection Problem

- Given a database schema (tables, attributes)
- Given a "query workload":
  - Workload = a set of (query, frequency) pairs
  - The queries may be both SELECT and updates
  - Frequency = either a count, or a percentage
- Select a set of indexes that optimizes the workload

In general this is a very hard problem

Index Selection: Which Search Key

- Make some attribute K a search key if the WHERE clause contains:
  - An exact match on K
  - A range predicate on K
  - A join on K

The Index Selection Problem 1

```
V(M, N, P);
```

Your workload is this

100000 queries:

```
SELECT *
FROM V
WHERE N=?
```

100 queries:

```
SELECT *
FROM V
WHERE P=?
```

What indexes ?

The Index Selection Problem 1

```
V(M, N, P);
```

Your workload is this

100000 queries:

```
SELECT *
FROM V
WHERE N=?
```

100 queries:

```
SELECT *
FROM V
WHERE P=?
```

A: V(N) and V(P) (hash tables or B-trees)

The Index Selection Problem 2

V(M, N, P);

Your workload is this

100000 queries:

```
SELECT *
FROM V
WHERE N>? and N<?
```

100 queries:

```
SELECT *
FROM V
WHERE P=?
```

100000 queries:

```
INSERT INTO V
VALUES (?, ?, ?)
```

What indexes ?

The Index Selection Problem 2

```
V(M, N, P);
```

Your workload is this

100000 queries:

```
SELECT *
FROM V
WHERE N>? and N<?
```

100 queries:

```
SELECT *
FROM V
WHERE P=?
```

100000 queries:

```
INSERT INTO V
VALUES (?, ?, ?)
```

A: definitely V(N) (must B-tree); unsure about V(P)

The Index Selection Problem 3

```
V(M, N, P);
```

Your workload is this

100000 queries:

```
SELECT *
FROM V
WHERE N=?
```

1000000 queries:

```
SELECT *
FROM V
WHERE N=? and P>?
```

100000 queries:

```
INSERT INTO V
VALUES (?, ?, ?)
```

What indexes ?

The Index Selection Problem 3

V(M, N, P);

Your workload is this

100000 queries:

```
SELECT *
FROM V
WHERE N=?
```

1000000 queries:

```
SELECT *
FROM V
WHERE N=? and P>?
```

100000 queries:

```
INSERT INTO V
VALUES (?, ?, ?)
```

A: V(N, P)

The Index Selection Problem 4

```
V(M, N, P);
```

Your workload is this

1000 queries:

```
SELECT *
FROM V
WHERE N>? and N<?
```

100000 queries:

```
SELECT *
FROM V
WHERE P>? and P<?
```

What indexes ?

The Index Selection Problem 4

V(M, N, P);

Your workload is this

1000 queries:

```
SELECT *
FROM V
WHERE N>? and N<?
```

100000 queries:

```
SELECT *
FROM V
WHERE P>? and P<?
```

A: V(N) secondary, V(P) primary index

The Index Selection Problem

- SQL Server
  - Automatically, thanks to *AutoAdmin* project
  - Much acclaimed successful research project from mid 90's, similar ideas adopted by the other major vendors
- PostgreSQL
  - You will do it manually, part of homework 5
  - But tuning wizards also exist

Index Selection: Multi-attribute Keys

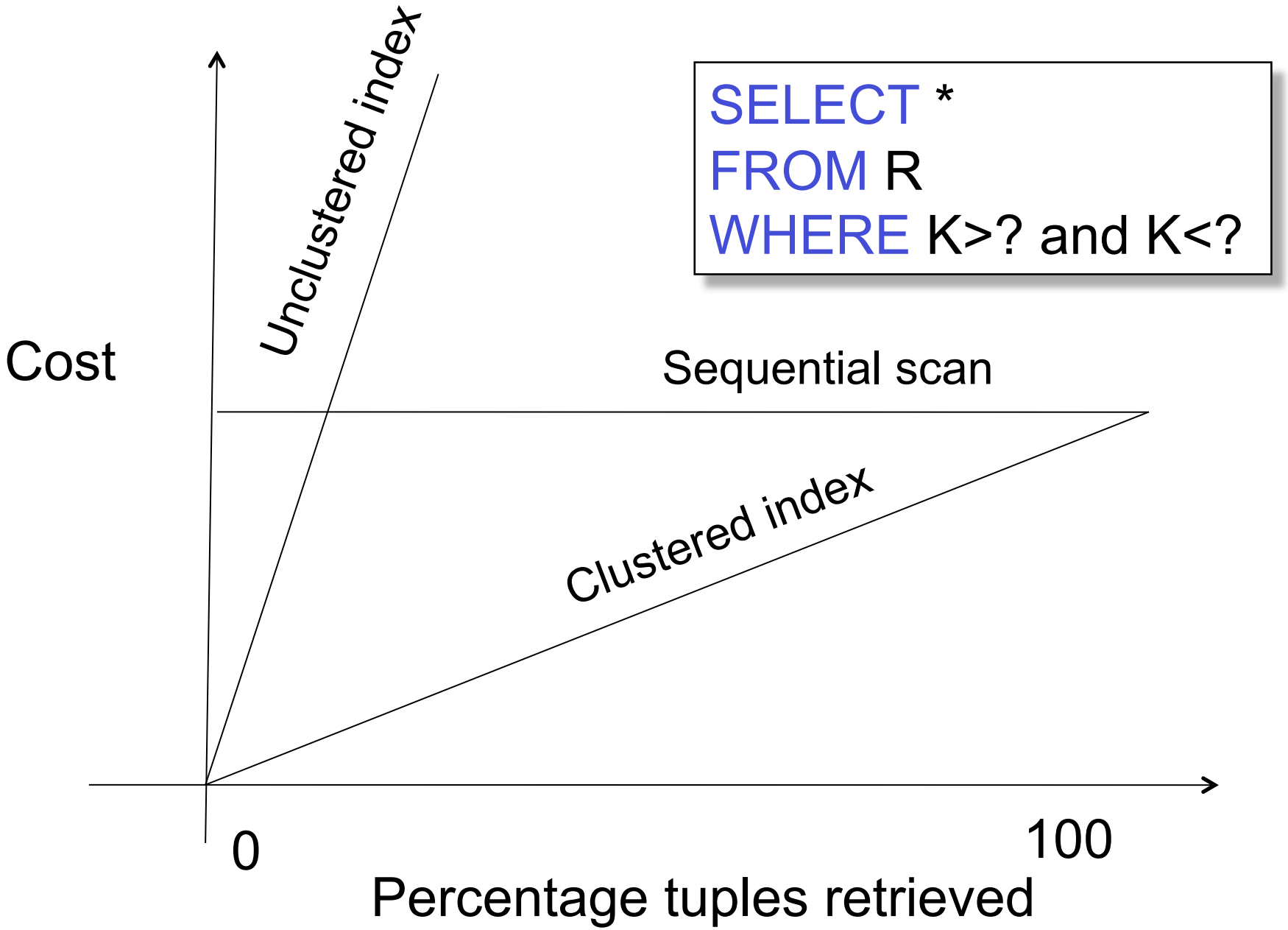
Consider creating a multi-attribute key on K1, K2, … if

- WHERE clause has matches on K1, K2, …
  - But also consider separate indexes
- SELECT clause contains only K1, K2, ..
  - A *covering index* is one that can be used exclusively to answer a query, e.g. index R(K1,K2) covers the query:

```
SELECT K2 FROM R WHERE K1=55
```

To Cluster or Not

- Range queries benefit mostly from clustering
- Covering indexes do *not* need to be clustered: they work equally well unclustered


SELECT *
FROM R
WHERE K>? and K<?
Cost
Unclustered index
Sequential scan
Clustered index
0
100
Percentage tuples retrieved

Hash Table v.s. B+ tree

- Rule 1: always use a B+ tree ☺
- Rule 2: use a Hash table on K when:
  - There is a very important selection query on equality (WHERE K=?), and no range queries
  - You know that the optimizer uses a nested loop join where K is the join attribute of the inner relation (you will understand that in a few lectures)

Balance Queries v.s. Updates

- Indexes speed up queries
  - SELECT FROM WHERE
- But they usually slow down updates:
  - INSERT, DELETE, UPDATE
  - However some updates benefit from indexes

```
UPDATE R
  SET A = 7
  WHERE K=55
```

Tools for Index Selection

- SQL Server 2000 Index Tuning Wizard
- DB2 Index Advisor

- How they work:
  - They walk through a large number of configurations, compute their costs, and choose the configuration with minimum cost

Denormalization

Product(**pid**, pname, price, cid)
Company(**cid**, cname, city)

A very frequent query:

```
SELECT x.pid, x.pname
FROM Product x, Company y
WHERE x.cid = y.cid and x.price < ? and y.city = ?
```

How can we speed up this query workload ?

Denormalization

Product(**pid**, pname, price, cid)
Company(**cid**, cname, city)

Denormalize:

ProductCompany(**pid**, pname, price, cname, city)

```
INSERT INTO ProductCompany
  SELECT x.pid, x.pname, x.price, y.cname, y.city
  FROM Product x, Company y
  WHERE x.cid = y.cid
```

Denormalization

Next, replace the query

```
SELECT x.pid, x.pname
FROM Product x, Company y
WHERE x.cid = y.cid and x.price < ? and y.city = ?
```

```
SELECT pid, pname
FROM ProductCompany
WHERE price < ? and city = ?
```

Issues with Denormalization

- It is no longer in BCNF
  - We have the hidden FD: cid → cname, city
- When Product or Company are updated, we need to propagate updates to ProductCompany
  - Use a TRIGGER in PostgreSQL (see PostgreSQL doc.)
- Sometimes cannot modify the query
  - What do we do then ?

Denormalization Using Views

```
INSERT INTO ProductCompany
   SELECT x.pid, x.pname,.price, y.cid, y.cname, y.city
   FROM Product x, Company y
   WHERE x.cid = y.cid;

DROP Product; DROP Company;

CREATE VIEW Product AS
   SELECT pid, pname, price, cid FROM ProductCompany

CREATE VIEW Company AS
   SELECT DISTINCT cid, cname, city FROM ProductCompany
```

Security in SQL

- Discretionary access control in SQL
- Using views for security

Discretionary Access Control in SQL

```
GRANT privileges
    ON object
    TO users
    [WITH GRANT OPTIONS]
```

privileges = SELECT |
INSERT(column-name) |
UPDATE(column-name) |
DELETE |
REFERENCES(column-name)

object = table | attribute

Examples

```
GRANT INSERT, DELETE ON Customers
TO Yuppy WITH GRANT OPTIONS
```

Queries allowed to Yuppy:

```
INSERT INTO Customers(cid, name, address)
      VALUES(32940, 'Joe Blow', 'Seattle')

DELETE Customers
      WHERE LastPurchaseDate < 1995
```

Queries denied to Yuppy:

```
SELECT Customer.address
FROM Customer
WHERE name = 'Joe Blow'
```

Examples

GRANT SELECT ON Customers TO **Michael**

Now **Michael** can SELECT, but not INSERT or DELETE

Examples

```
GRANT SELECT ON Customers
   TO Michael WITH GRANT OPTIONS
```

Michael can say this:

GRANT SELECT ON Customers TO **Yuppi**

Now **Yuppi** can SELECT on Customers

Examples

```
GRANT UPDATE (price)  ON Product TO Leah
```

Leah can update, but only Product.price, but not Product.name

Examples

Customer(cid, name, address, balance)
Orders(oid, cid, amount) cid= foreign key

Bill has INSERT/UPDATE rights to Orders.
BUT HE CAN'T INSERT ! (why ?)

GRANT REFERENCES (cid) ON Customer TO **Bill**

Now **Bill** can INSERT tuples into Orders

Views and Security

David owns

Customers:

Name	Address	Balance
Mary	Huston	450.99
Sue	Seattle	-240
Joan	Seattle	333.25
Ann	Portland	-520

Fred is not allowed to see this

David says

```
CREATE VIEW PublicCustomers
        SELECT Name, Address
        FROM Customers
GRANT SELECT ON PublicCustomers TO Fred
```

David owns

Views and Security

Customers:

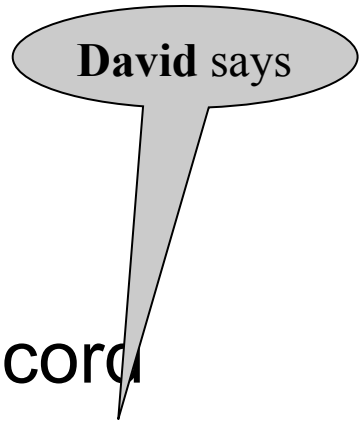
Name	Address	Balance
Mary	Huston	450.99
Sue	Seattle	-240
Joan	Seattle	333.25
Ann	Portland	-520

John is allowed to see only <0 balances

David says

```
CREATE VIEW BadCreditCustomers
        SELECT *
        FROM Customers
        WHERE Balance < 0
GRANT SELECT ON BadCreditCustomers TO John
```

Views and Security



- Each customer should see only her/his record

Name	Address	Balance
Mary	Huston	450.99
Sue	Seattle	-240
Joan	Seattle	333.25
Ann	Portland	-520

```
CREATE VIEW CustomerMary
        SELECT *  FROM Customers
        WHERE name = 'Mary'
GRANT SELECT
ON CustomerMary TO Mary
```

```
CREATE VIEW CustomerSue
        SELECT *  FROM Customers
        WHERE name = 'Sue'
GRANT SELECT
ON CustomerSue TO Sue
```

Doesn't scale.

Need *row-level* access control !

• • •

Revocation

```
REVOKE   [GRANT OPTION FOR]   privileges
         ON object FROM users  {  RESTRICT  |  CASCADE   }
```

Administrator says:

```
REVOKE SELECT ON Customers  FROM David CASCADE
```

John loses SELECT privileges on BadCreditCustomers

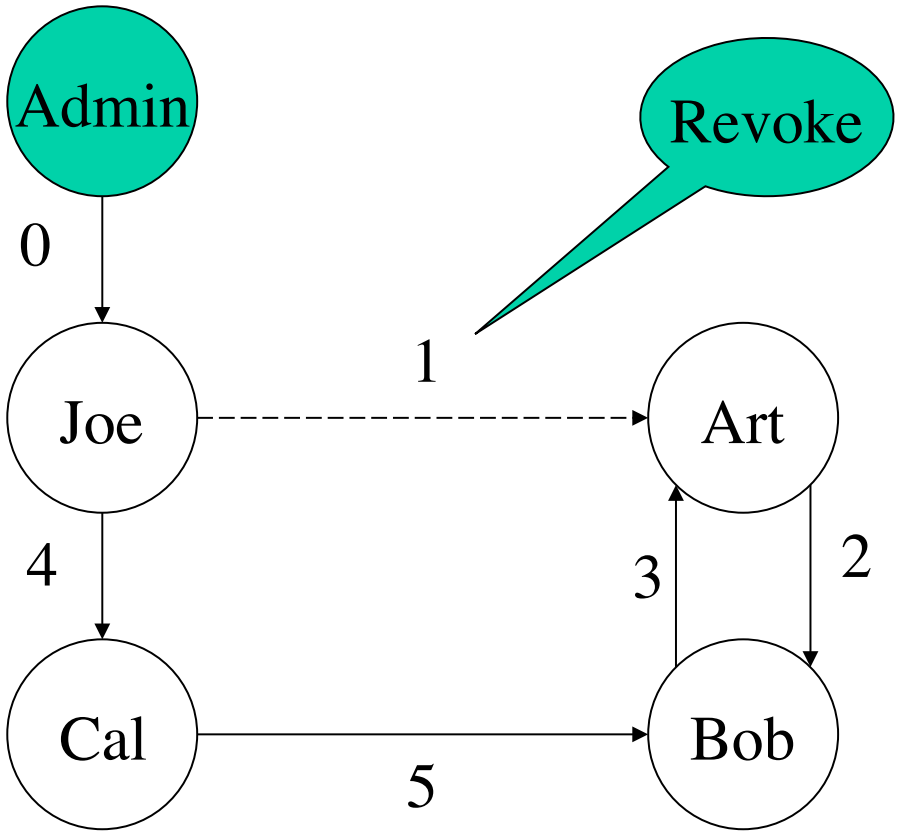
Revocation

Joe: GRANT [….] TO Art …
Art: GRANT [….] TO Bob …
Bob: GRANT [….] TO Art …
Joe: GRANT [….] TO Cal …
Cal: GRANT [….] TO Bob …
Joe: REVOKE [….] FROM Art CASCADE

Same privilege, same object, GRANT OPTION

What happens ??

Revocation



According to SQL everyone keeps the privilege

Summary of SQL Security

Limitations:

- No row level access control
- Table creator owns the data: that's unfair !
- Today the database is not at the center of the policy administration universe