# CSE544: Principles of Database Systems

Lectures 5-6

Database Architecture

Storage and Indexes

# Announcements

- Project
  - Choose a topic. Set *limited* goals!
  - Sign up (doodle) to meet with me this week
- Homework 1
  - A few people have not turned in yet: will do by tomorrow. Then we will post solutions
- Homework 2
  - Will be posted today; you will receive email
- Paper review for Wednesday
  - Join processing

# Where We Are

- Part 1: The relational data model
- Part 2: Database Systems
- Part 3: Database Theory
- Part 4: Miscellaneous

# Where We Are

- **The relational data model**
  - **Motivation of the relational model**
    - Older data models and the need for *data independence*
    - Relational model, E/R model, Normal Forms (we skipped them)
  - **Query Languages**
    - SQL
    - Relational algebra
    - Relational calculus
    - Non-recursive datalog with negation

- **Database Systems**
  - **How can we efficiently implement this model?**

# Outline

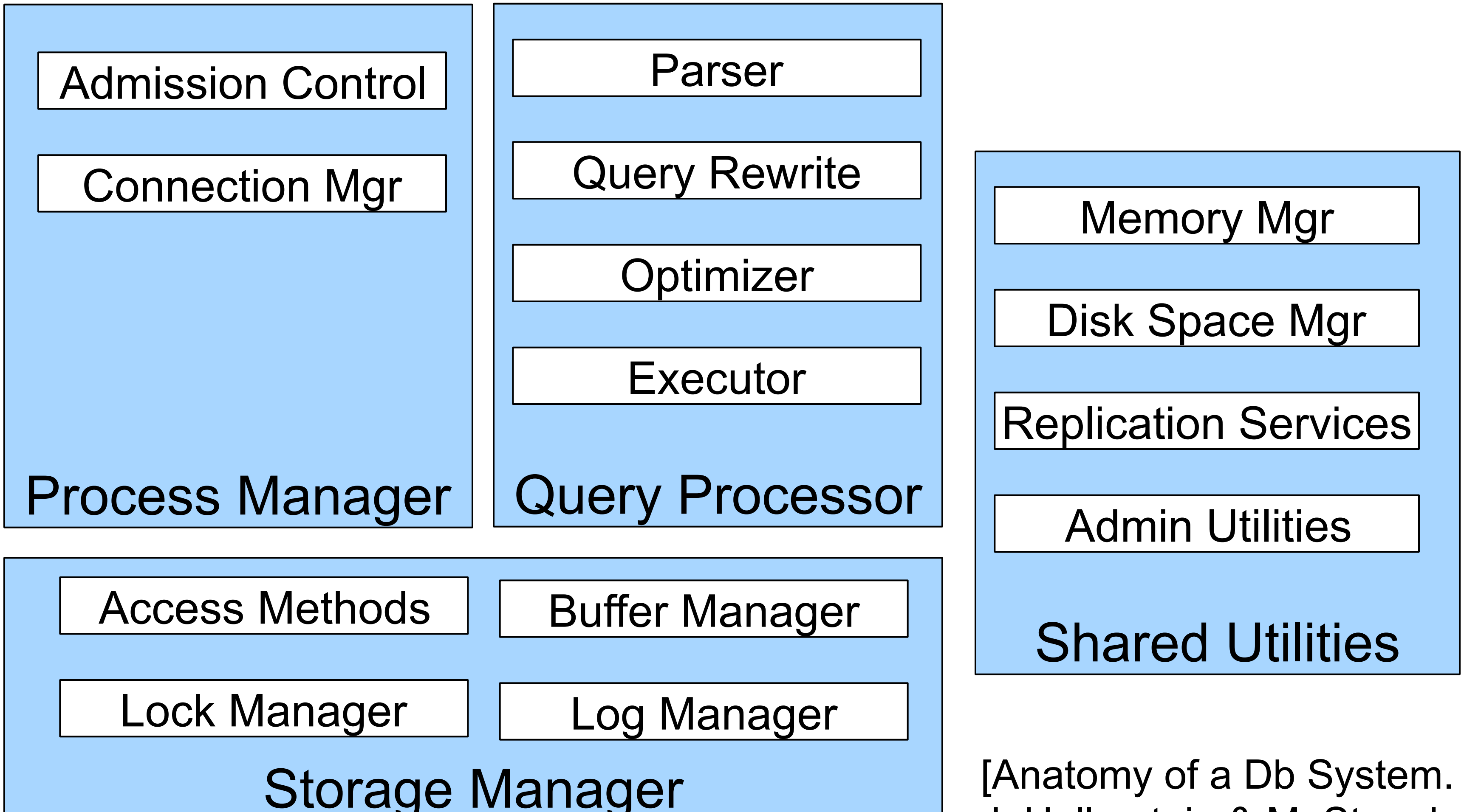
- DBMS Architecture
  - Anatomy of a database system. J. Hellerstein and M. Stonebraker. In Red Book (4th ed).

- Storage and Indexes
  - Book: Ch. 8-11, and 20

# DMBS Architecture: Outline

- Main components of a modern DBMS
- Process models
- Storage models
- Query processor

# DBMS Architecture




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

# DMBS Architecture: Outline

- Main components of a modern DBMS
- Process models
- Storage models
- Query processor

# Process Model

Q: Why not simply queue all user requests, and serve them one at the time?

# Process Model

Q: Why not simply queue all user requests, and serve them one at the time?

A: Because of the high disk I/O latency

Corollary: in a main memory db you can service transactions sequentially!

Alternatives

1. **Process per connection**
2. **Server process** (thread per connection)
   - OS threads or DBMS threads
3. **Server process with I/O process**

# Process Per Connection

- **Overview**
  - DB server forks one process for each client connection
- **Advantages**
  - ?
- **Drawbacks**
  - ?

# Process Per Connection

- **Overview**
  - DB server forks one process for each client connection
- **Advantages**
  - Easy to implement (OS time-sharing, OS isolation, debuggers, etc.)
- **Drawbacks**
  - Need OS-supported “shared memory” (for lock table, buffer pool)
  - Not scalable: memory overhead and expensive context switches

# Server Process

- **Overview**
  - *Dispatcher* thread listens to requests, dispatches *worker* threads

- **Advantages**
  - ?
  - ?

- **Drawbacks**
  - ?

# Server Process

- **Overview**
  - *Dispatcher* thread listens to requests, dispatches *worker* threads
- **Advantages**
  - Shared structures can simply reside on the heap
  - Threads are lighter weight than processes: memory, context switching
- **Drawbacks**
  - Concurrent programming is hard to get right (race conditions, deadlocks)
  - Subtle API thread differences across different operating systems make portability difficult

# Sever Process with I/O Process

Problem: entire process blocks on synchronous I/O calls

- Solution 1: Use separate process(es) for I/O tasks
- Solution 2: Modern OS provide asynchronous I/O

# DBMS Threads vs OS Threads

- **Why do DBMSs implement their own threads?**

# DBMS Threads vs OS Threads

- **Why do DBMSs implement their own threads?**
  - Legacy: originally, there were no OS threads
  - Portability: OS thread packages are not completely portable
  - Performance: fast task switching

- **Drawbacks**
  - Replicating a good deal of OS logic
  - Need to manage thread state, scheduling, and task switching

- **How to map DBMS threads onto OS threads or processes?**
  - Rule of thumb: one OS-provided dispatchable unit per physical device
  - See page 9 and 10 of Hellerstein and Stonebraker's paper

# Historical Perspective (1981)

In 1981:

- No OS threads
- No shared memory between processes
  - Makes one process per user hard to program
- Some OSs did not support many to one communication
  - Thus forcing the one process per user model
- No asynchronous I/O
  - But inter-process communication expensive
  - Makes the use of I/O processes expensive

- Common original design: DBMS threads, frequently yielding control to a scheduling routine

# Commercial Systems

- Oracle
  - Unix default: process-per-user mode
  - Unix: DBMS threads multiplexed across OS processes
  - Windows: DBMS threads multiplexed across OS threads
- DB2
  - Unix: process-per-user mode
  - Windows: OS thread-per-user
- SQL Server
  - Windows default: OS thread-per-user
  - Windows: DBMS threads multiplexed across OS threads

# DMBS Architecture: Outline

- Main components of a modern DBMS
- Process models
- Storage models
- Query processor

# Storage Model

- **Problem**: DBMS needs spatial and temporal control over storage
  - Spatial control for performance
  - Temporal control for correctness and performance
- **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**
  - ?
  - ?
- **Disadvantages**
  - ?

# 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"
    - SAN = storage area network; NAS = network attached device

# Spatial Control Using OS Files

- **Overview**
  - DBMS creates one or more very large OS files
- **Advantages**
  - ?
- **Disadvantages**
  - ?

# 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**
  - Must control the timing of writes for *correctness* and *performance*
  - OS may further delay writes
  - OS may lead to double buffering, leading to unnecessary copying
  - DB must fine tune when the log tail is flushed to disk

# Historical Perspective (1981)

- Recognizes mismatch problem between OS files and DBMS needs
  - If DBMS uses OS files and OS files grow with time, blocks get scattered
  - OS uses tree structure for files but DBMS needs its own tree structure

- Other proposals at the time
  - Extent-based file systems
  - Record management inside OS

# 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

# Temporal Control
# Buffer Manager

- Correctness problems
  - DBMS needs to control when data is written to disk in order to provide **transactional semantics** (we will study transactions later)
  - OS buffering can **delay writes**, causing problems when crashes occur

- Performance problems
  - OS optimizes buffer management for general workloads
  - DBMS understands its workload and can do better
  - Areas of possible optimizations
    - Page replacement policies
    - Read-ahead algorithms (physical vs logical)
    - Deciding when to flush tail of write-ahead log to disk

# Historical Perspective (1981)

- Problems with OS buffer pool management long recognized
  - Accessing OS buffer pool involves an expensive system call
  - Faster to access a DBMS buffer pool in user space
  - LRU replacement does not match DBMS workload
  - DBMS can do better
  - OS can do only sequential prefetching, DBMS knows which page it needs next and that page may not be sequential
  - DBMS needs ability to control when data is written to disk

# Commercial Systems

- DBMSs implement their own buffer pool managers
- Modern filesystems provide good support for DBMSs
  - Using large files provides good spatial control
  - Using interfaces like the mmap suite
    - Provides good temporal control
    - Helps avoid double-buffering at DBMS and OS levels

# DMBS Architecture: Outline

- Main components of a modern DBMS
- Process models
- Storage models
- Query processor  (will go over the query processor in lectures 6-7)

# Outline

- DBMS Architecture
  - Anatomy of a database system. J. Hellerstein and M. Stonebraker. In Red Book (4th ed).
- Storage and Indexes
  - Book: Ch. 8-11, and 20

# Arranging Pages on Disk

A disk is organized into blocks (a.k.a. pages)

- blocks on same track, followed by
- blocks on same cylinder, followed by
- blocks on adjacent cylinder

A file should (ideally) consists of sequential blocks on disk, to minimize seek and rotational delay.

For a sequential scan, pre-fetching several pages at a time is a big win!

# Issues

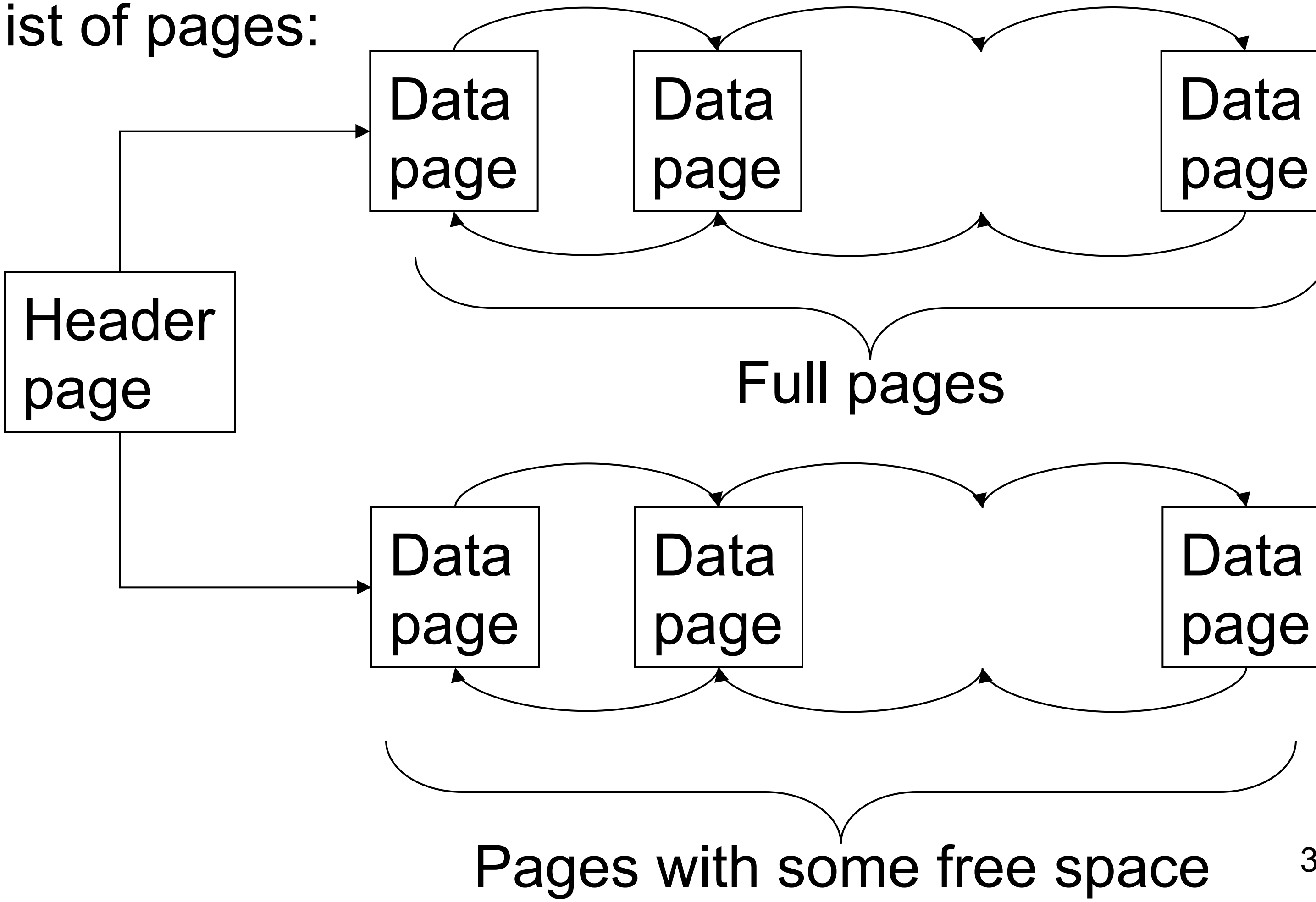
- Managing free blocks
- Represent the records inside the blocks
- Represent attributes inside the records

# Managing Free Blocks

- Linked list of free blocks
- Or bit map

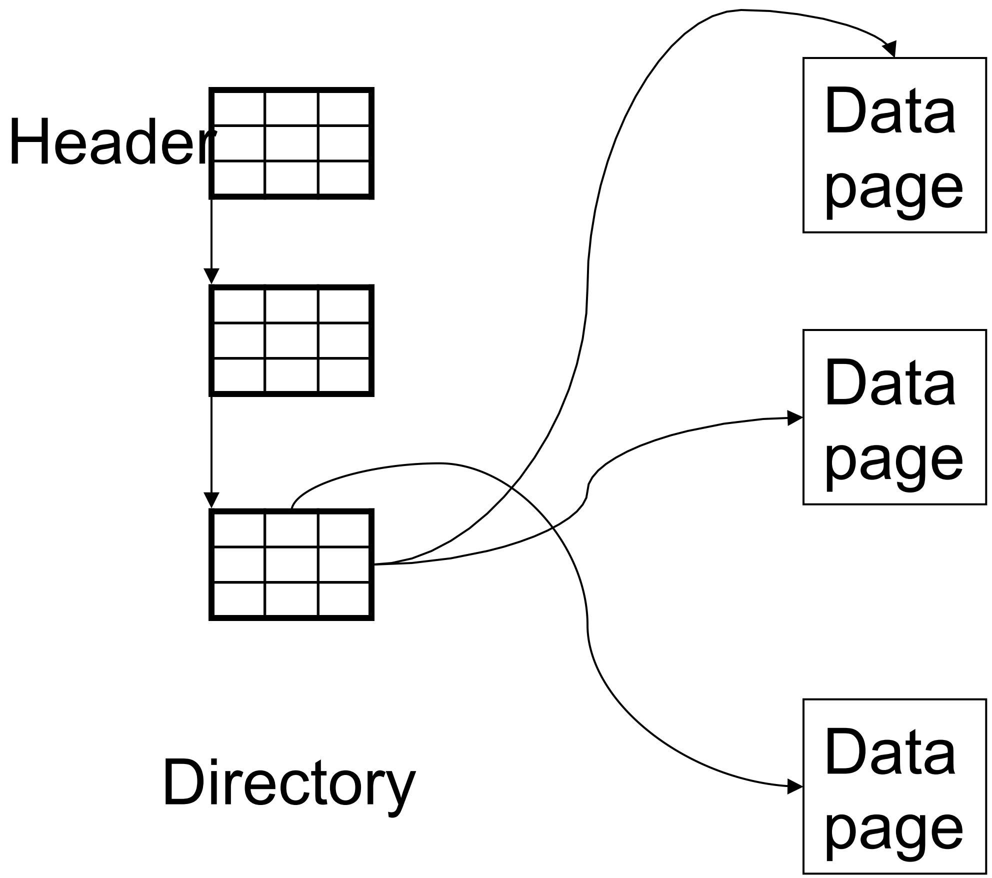
# File Organization

Linked list of pages:



# File Organization

Better: directory of pages



# Page Formats

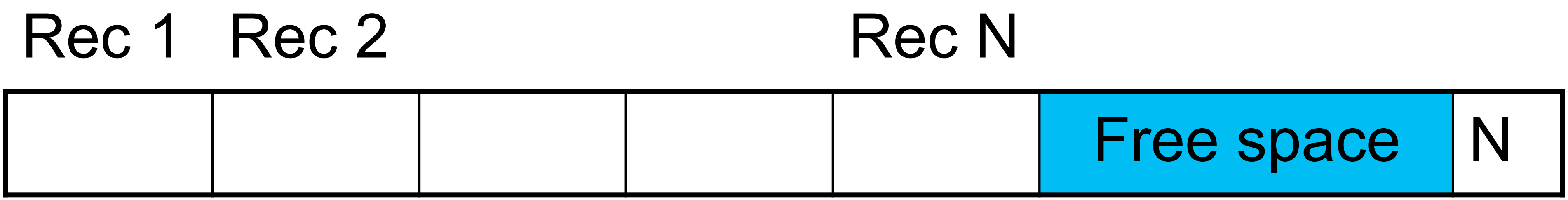
Issues to consider

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

Why do we need RID's in a relational DBMS ?

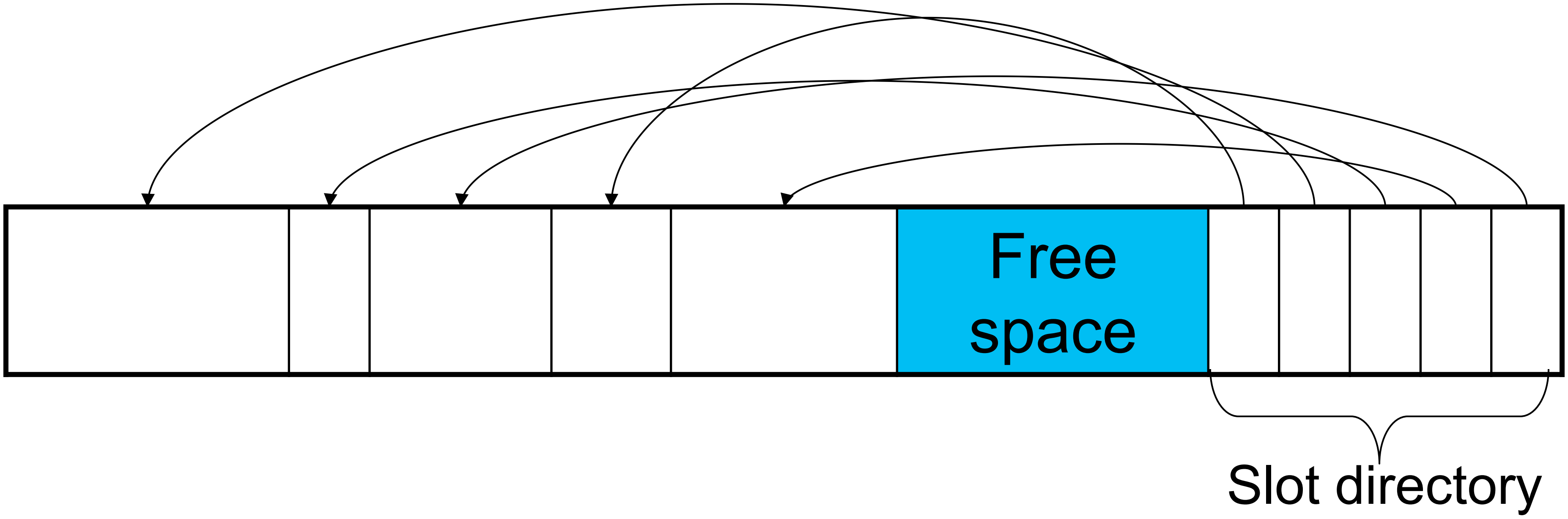
# Page Formats

Fixed-length records: packed representation




Problems ?

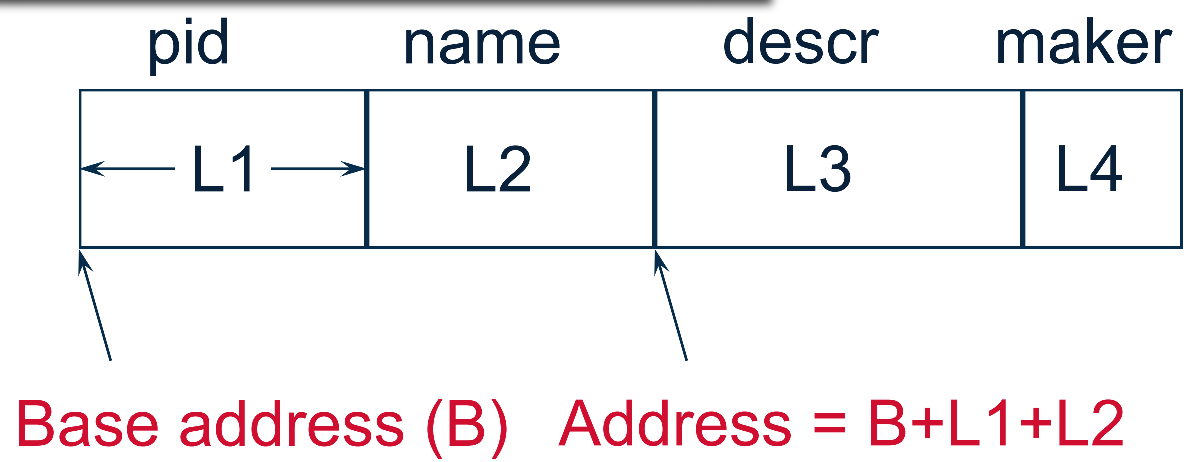
# Page Formats




Variable-length records

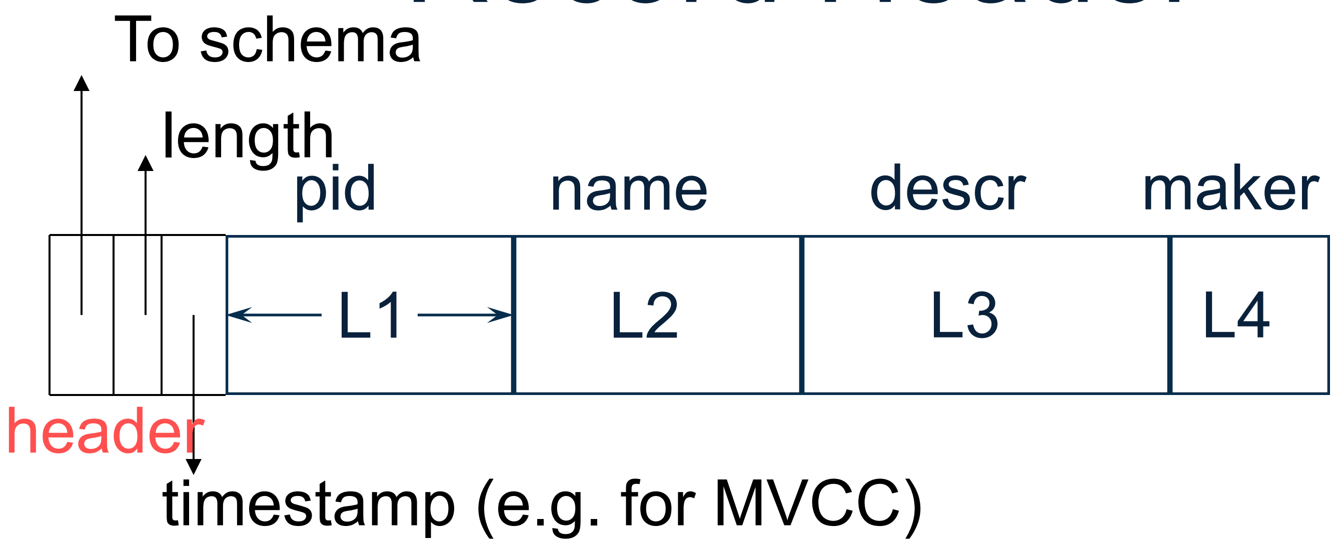
# Record Formats:  Fixed Length

Product(pid, name, descr, maker)




- Information about field types same for all records in a file; stored in *system catalogs.*
- Finding *i'th* field requires scan of record.
- Note the importance of schema information!

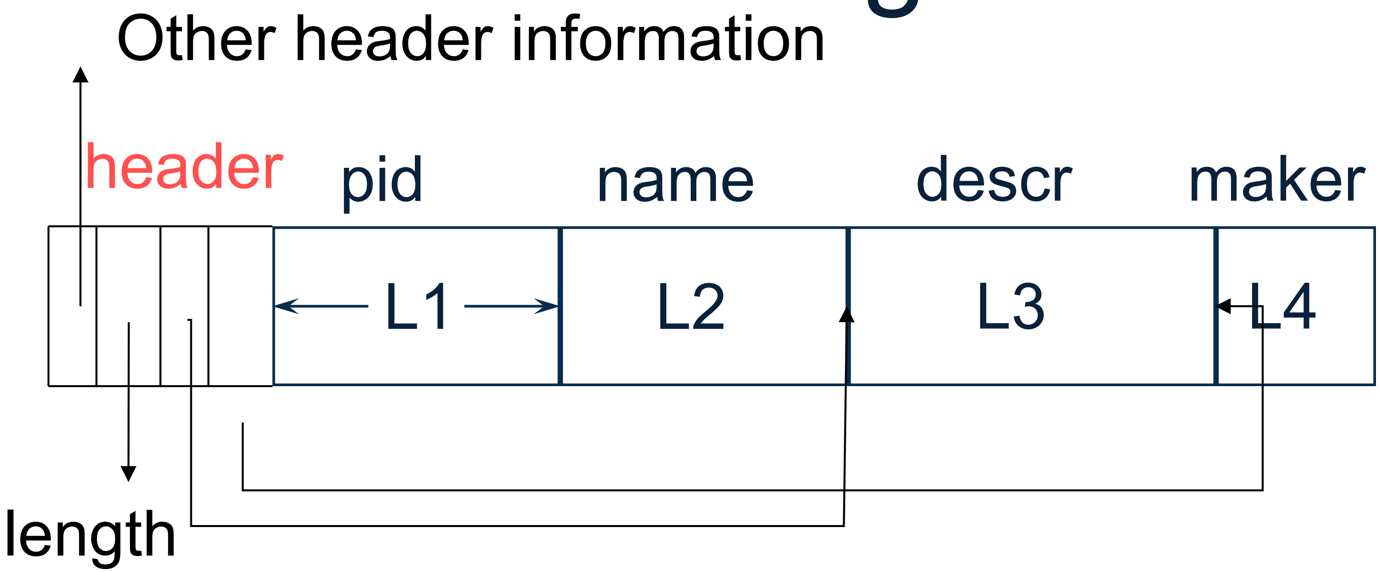
# Record Header




Need the header because:

- The schema may change
  for a while new+old may coexist
- Records from different relations may coexist

# Variable Length Records




Place the fixed fields first: F1

Then the variable length fields: F2, F3, F4

Null values take 2 bytes only

Sometimes they take 0 bytes (when at the end)

# BLOB

- Binary large objects
- Supported by modern database systems
- E.g. images, sounds, etc.
- Storage: attempt to cluster blocks together

CLOB = character large object

- Supports only restricted operations

# File Organizations

- Heap (random order) files: Suitable when typical access is a file scan retrieving all records.
- Sorted Files Best if records must be retrieved in some order, or only a `range' of records is needed.
- Indexes Data structures to organize records via trees or hashing.
  - Like sorted files, they speed up searches for a subset of records, based on values in certain ("search key") fields
  - Updates are much faster than in sorted files.

# Index

- A (possibly separate) file, that allows fast access to records in the data file
- The index contains (key, value) pairs:
  - The key = an attribute value
  - The value = one of:
    - pointer to the record *secondary index*
    - or the record itself *primary index*

Note: “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

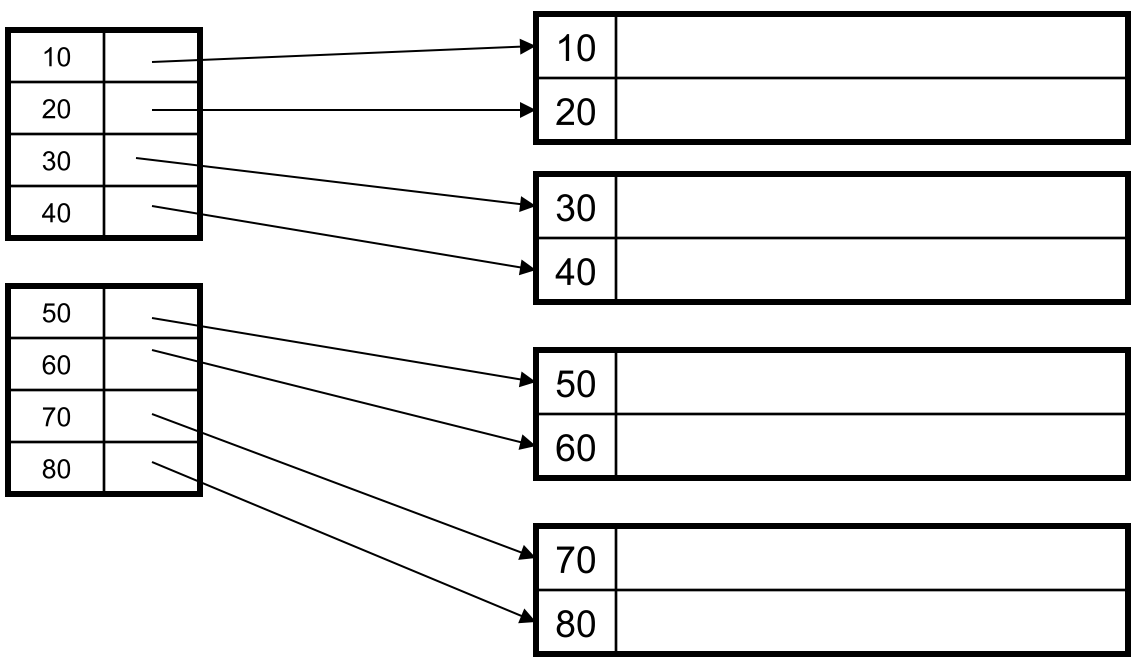
# Clustered/Unclustered

- 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

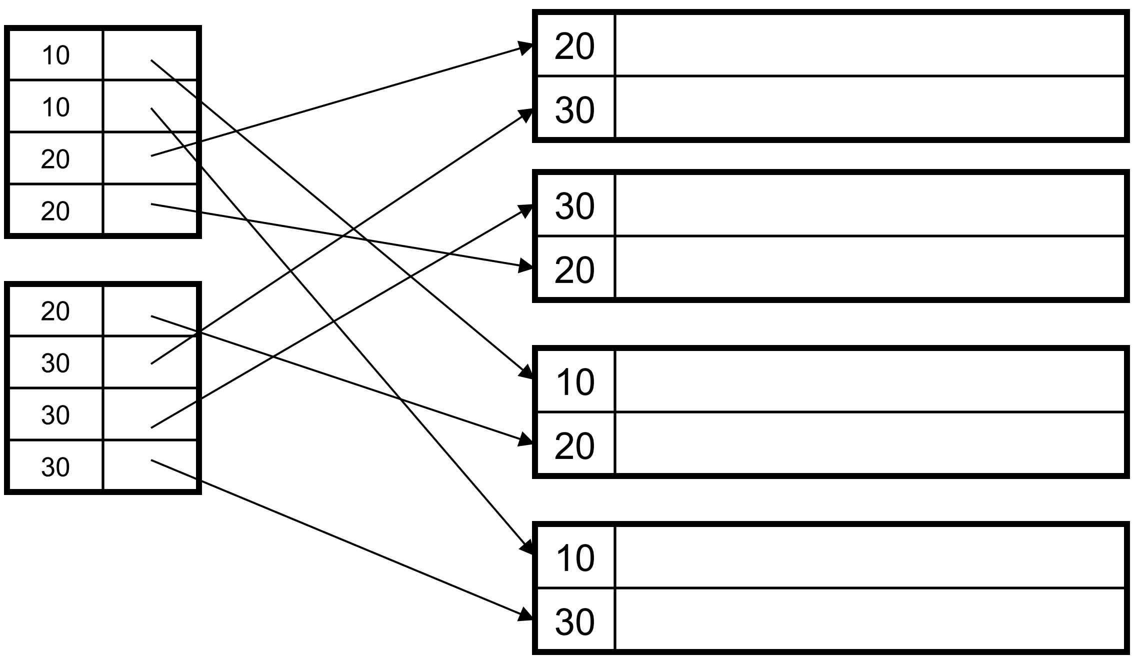
# Clustered Index

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

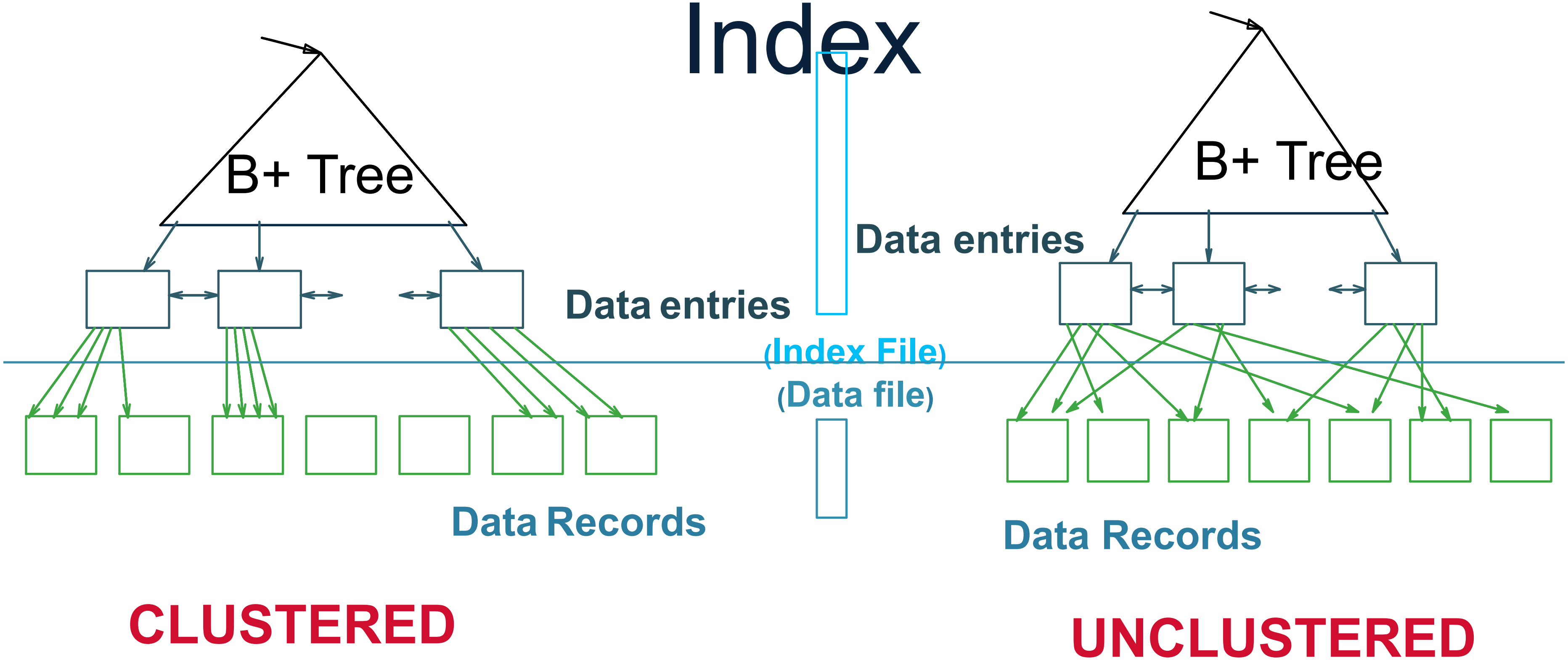


# Unclustered Index

- Several per table

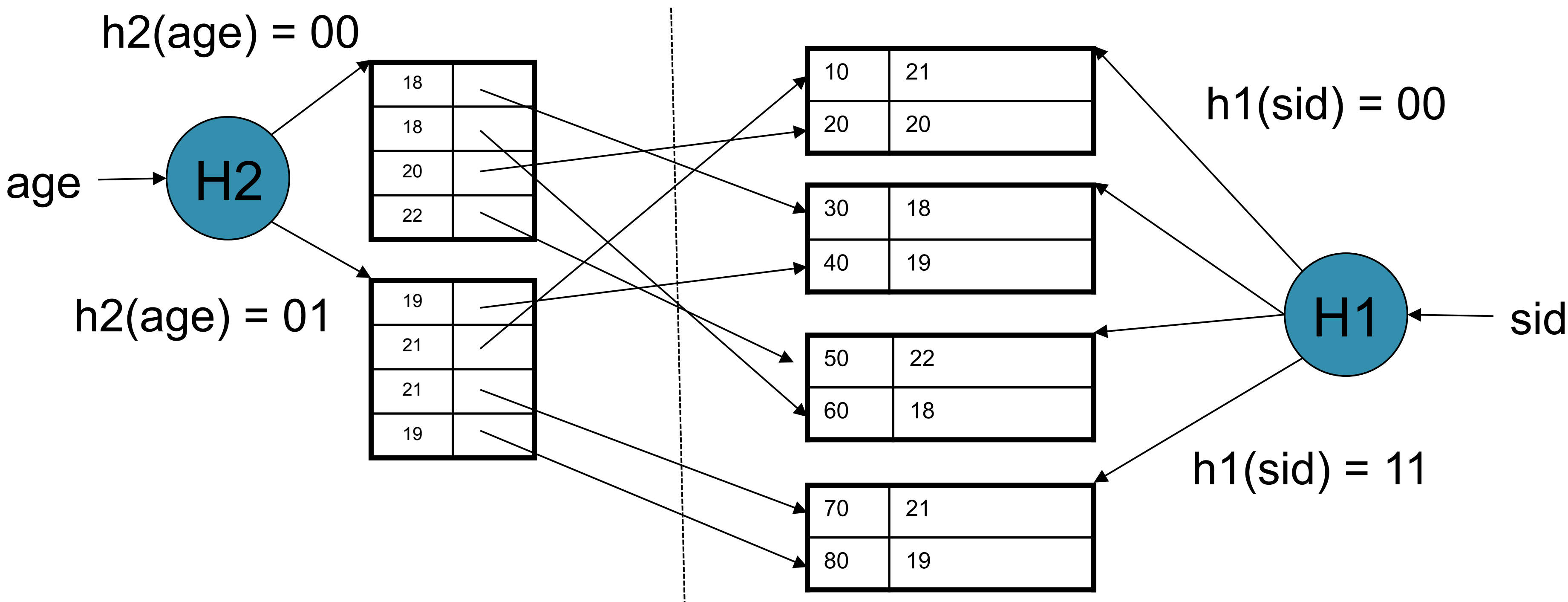


# Clustered vs. Unclustered Index



# Hash-Based Index

Good for point queries but not range queries




Another example
of unclustered/secondary index

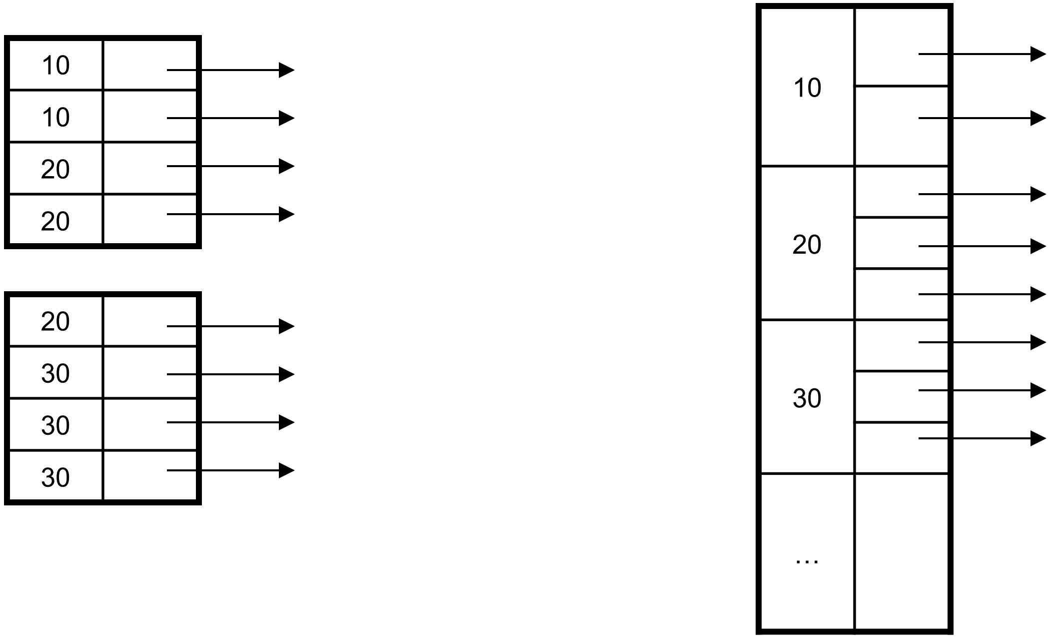
Another example of
clustered/primary index

# 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**>

# Alternatives 2 and 3



# B+ Trees

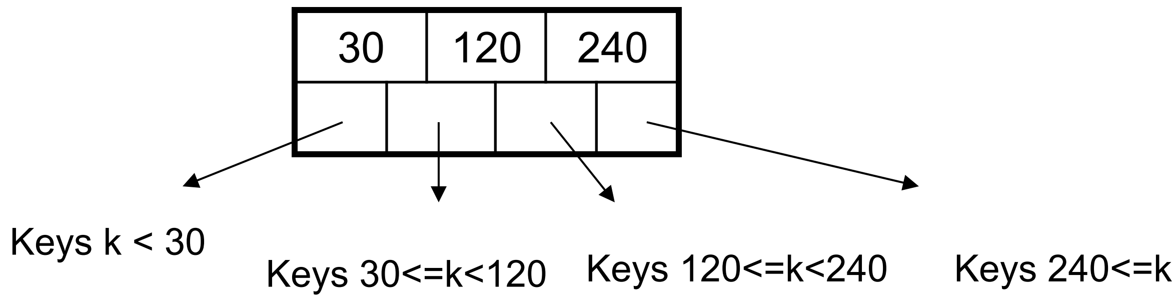
- Search trees

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

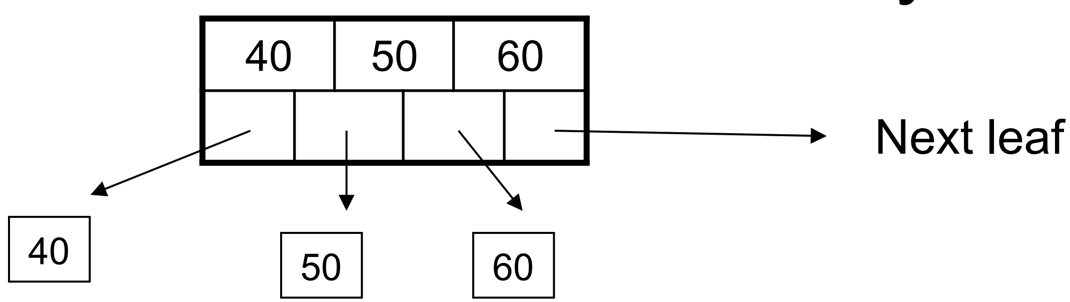
- Idea in B+ Trees
  - Make leaves into a linked list: facilitates range queries

# B+ Trees Basics

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



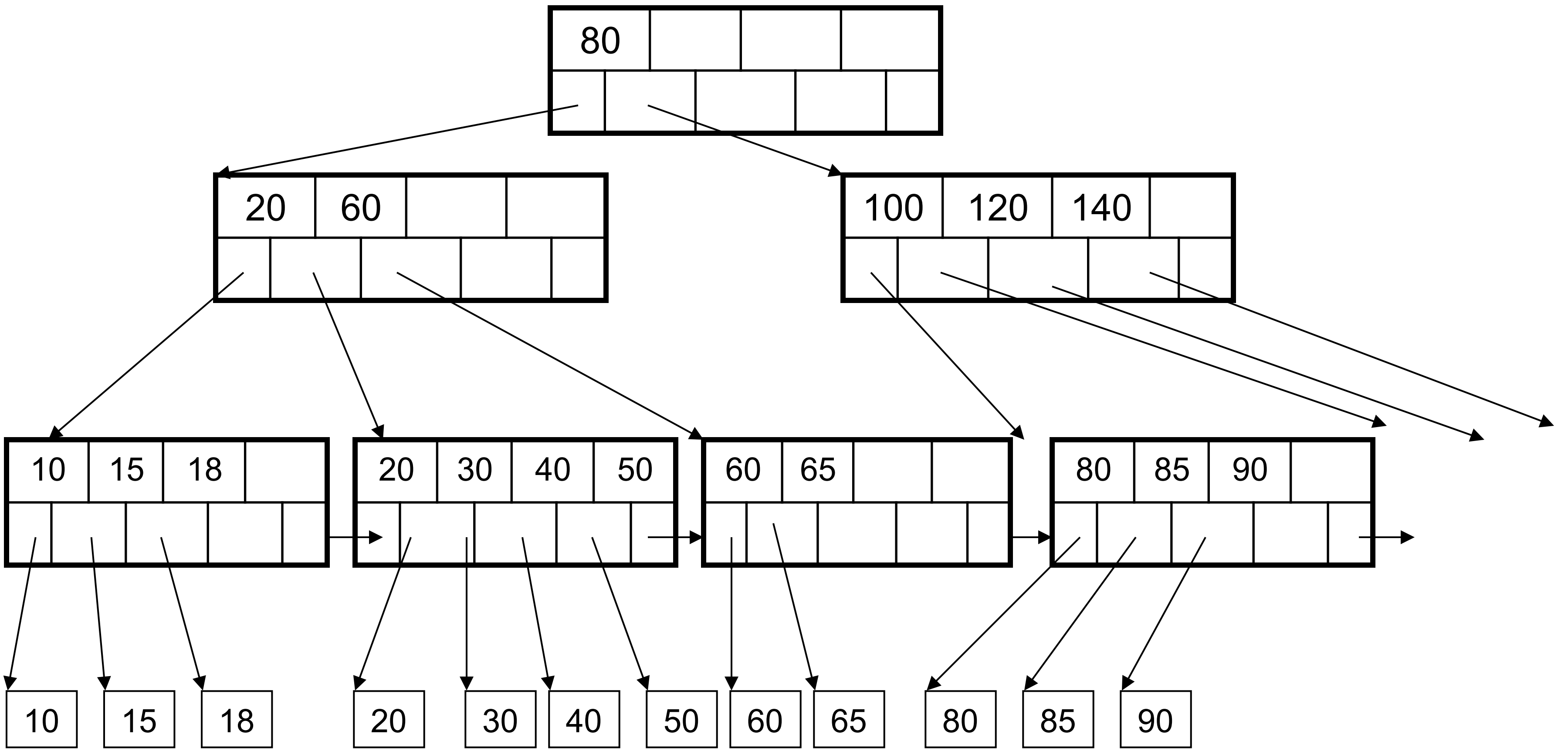

- Each leaf has >=d and <= 2d keys:



# B+ Tree Example

d = 2

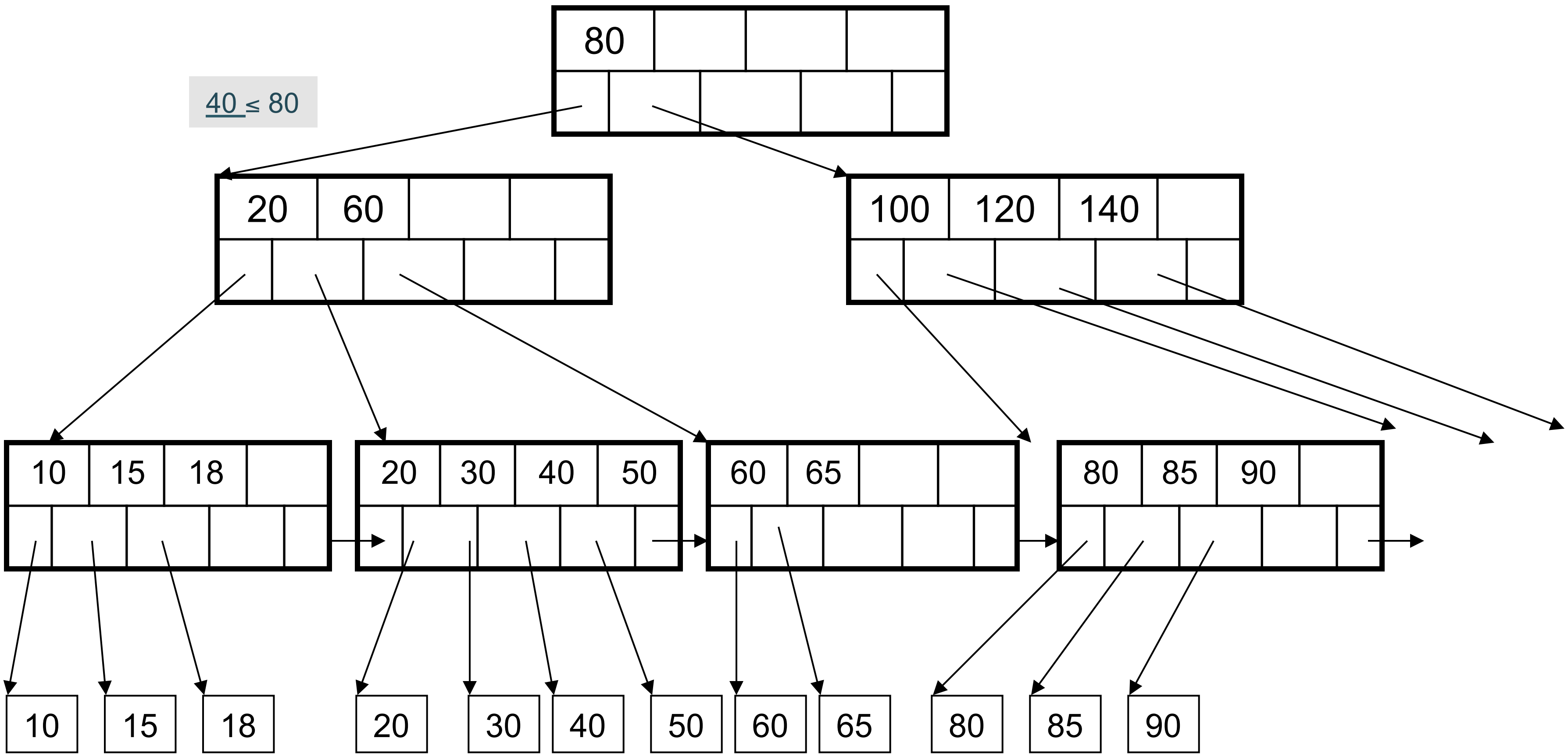
Find the key 40



# B+ Tree Example

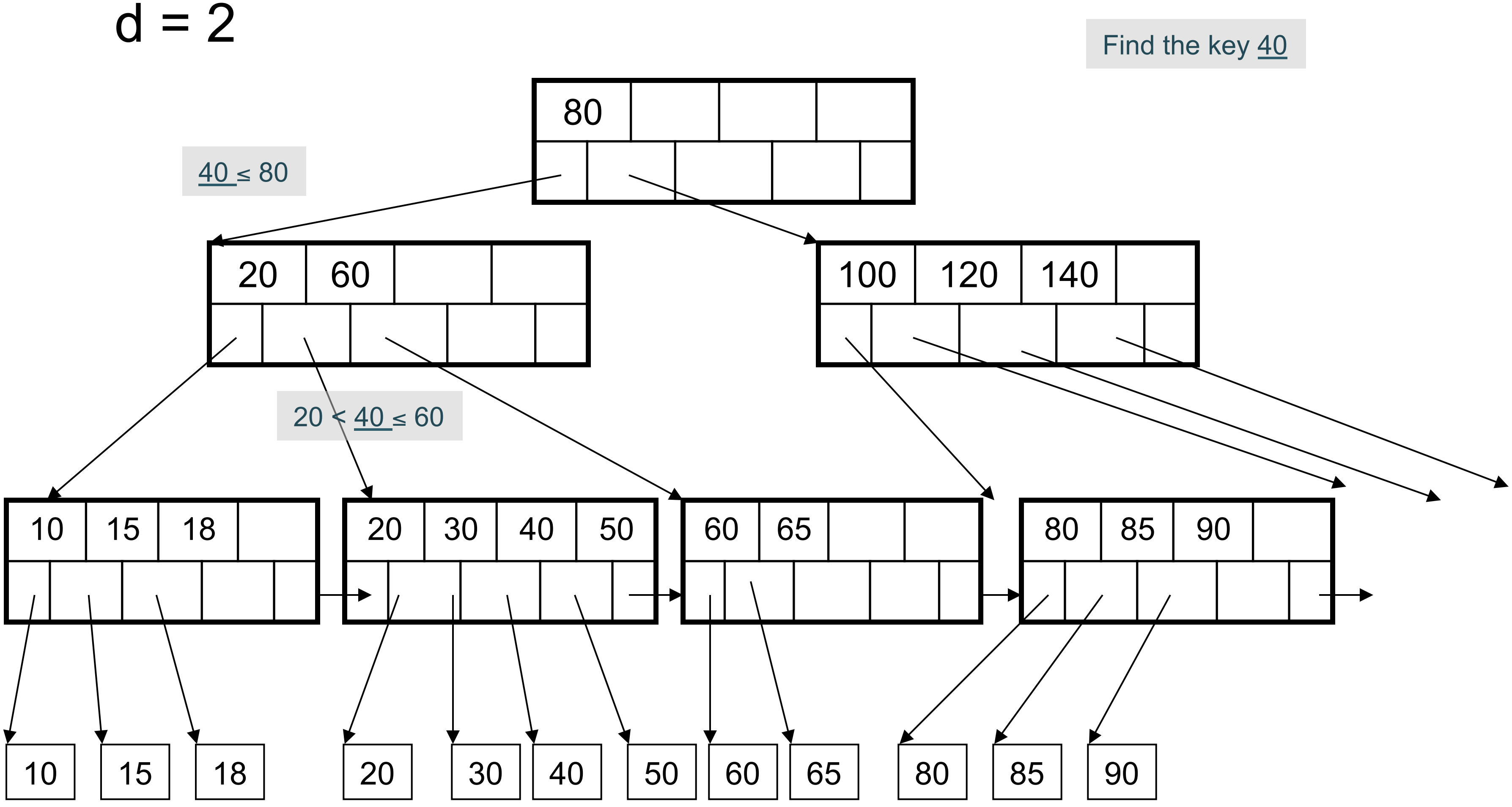
d = 2

Find the key 40



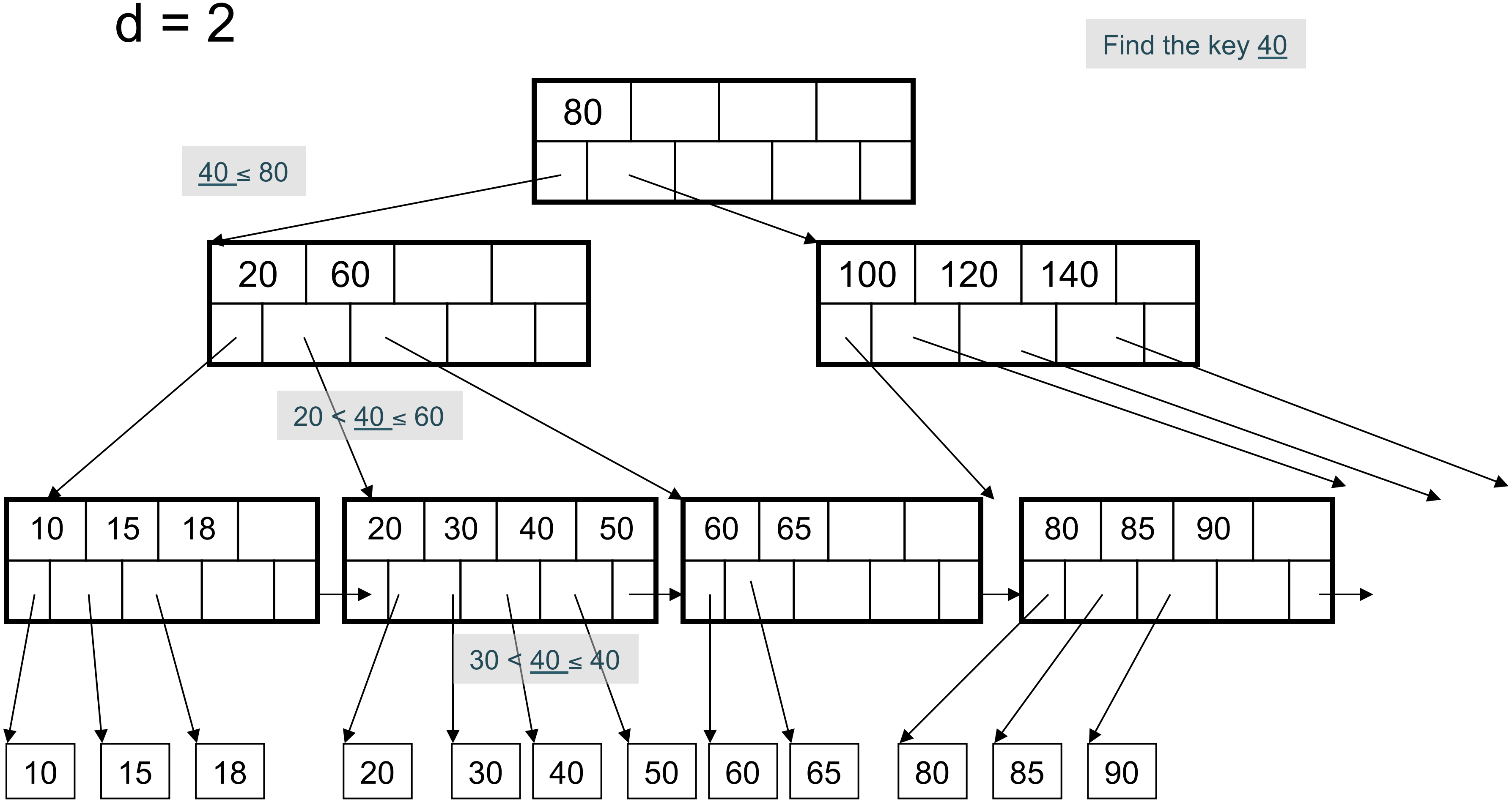
# B+ Tree Example

d = 2



# B+ Tree Example

d = 2



# Using a B+ Tree

Index on People(age)

- 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
```

# Which queries can use this index ?

Index on People(name, zipcode)

```
SELECT *
FROM People
WHERE name = 'Smith'
   and zipcode = 12345
```

```
SELECT *
FROM People
WHERE name = 'Smith'
```

```
SELECT *
FROM People
WHERE zipcode = 12345
```

# B+ Tree Design

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

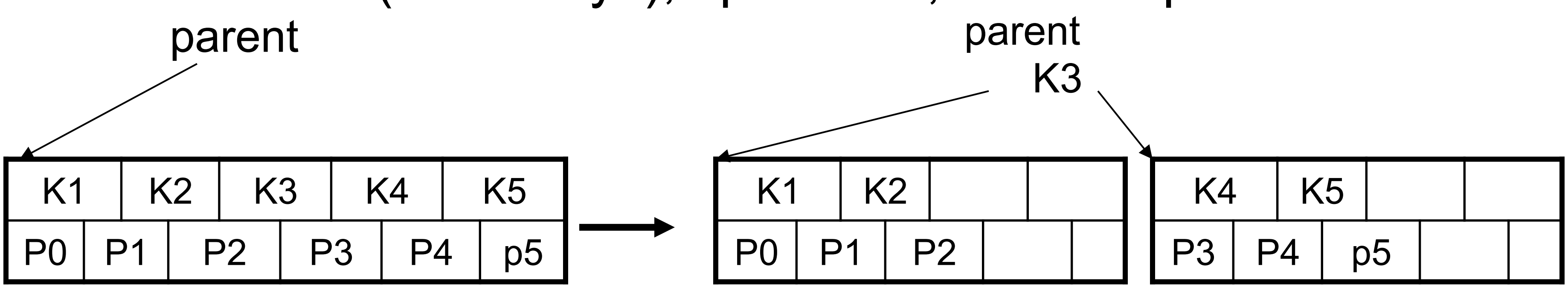
# B+ Trees in Practice

- 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)

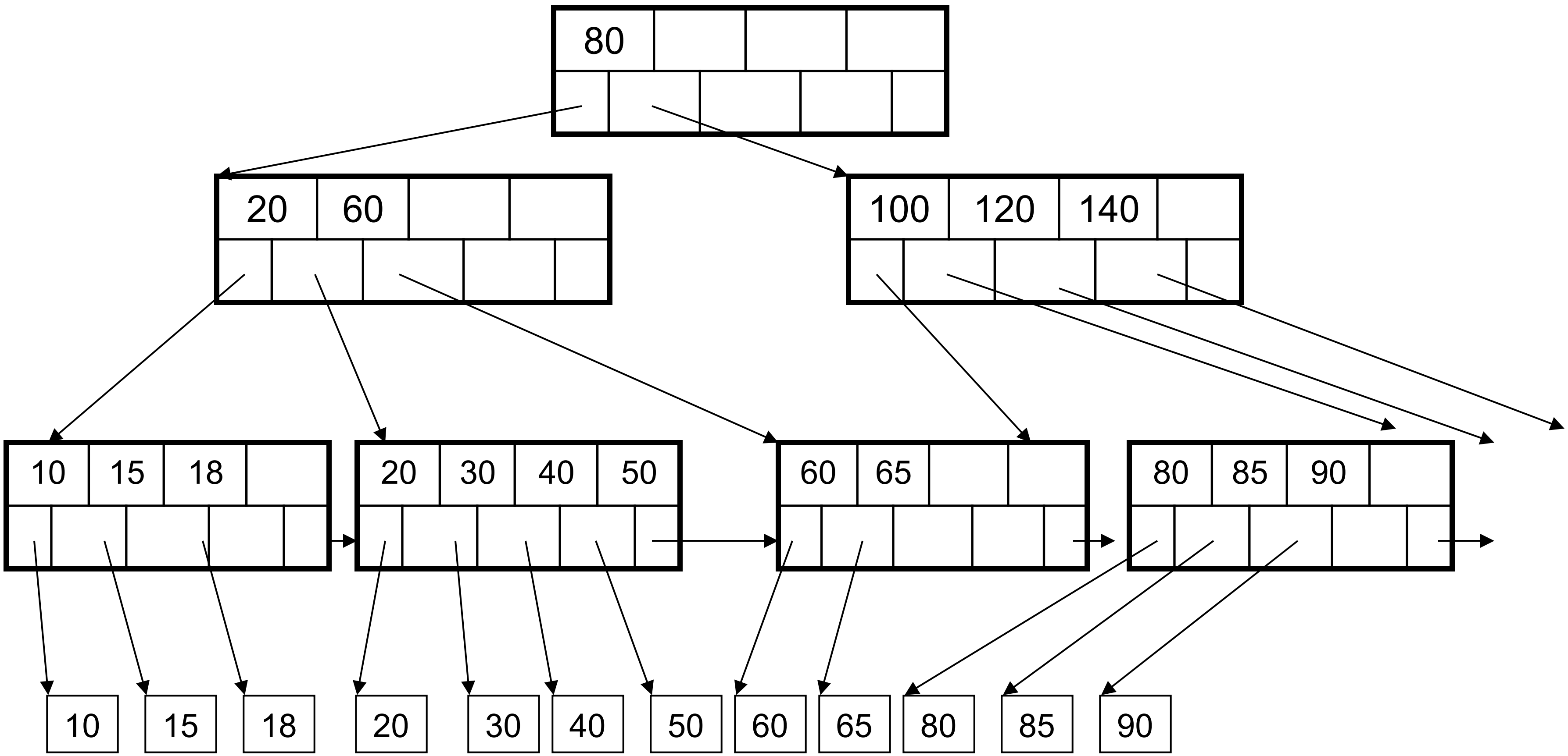
- Find leaf where K belongs, insert
- If no overflow (2d keys or less), halt
- If overflow (2d+1 keys), split node, insert in parent:




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

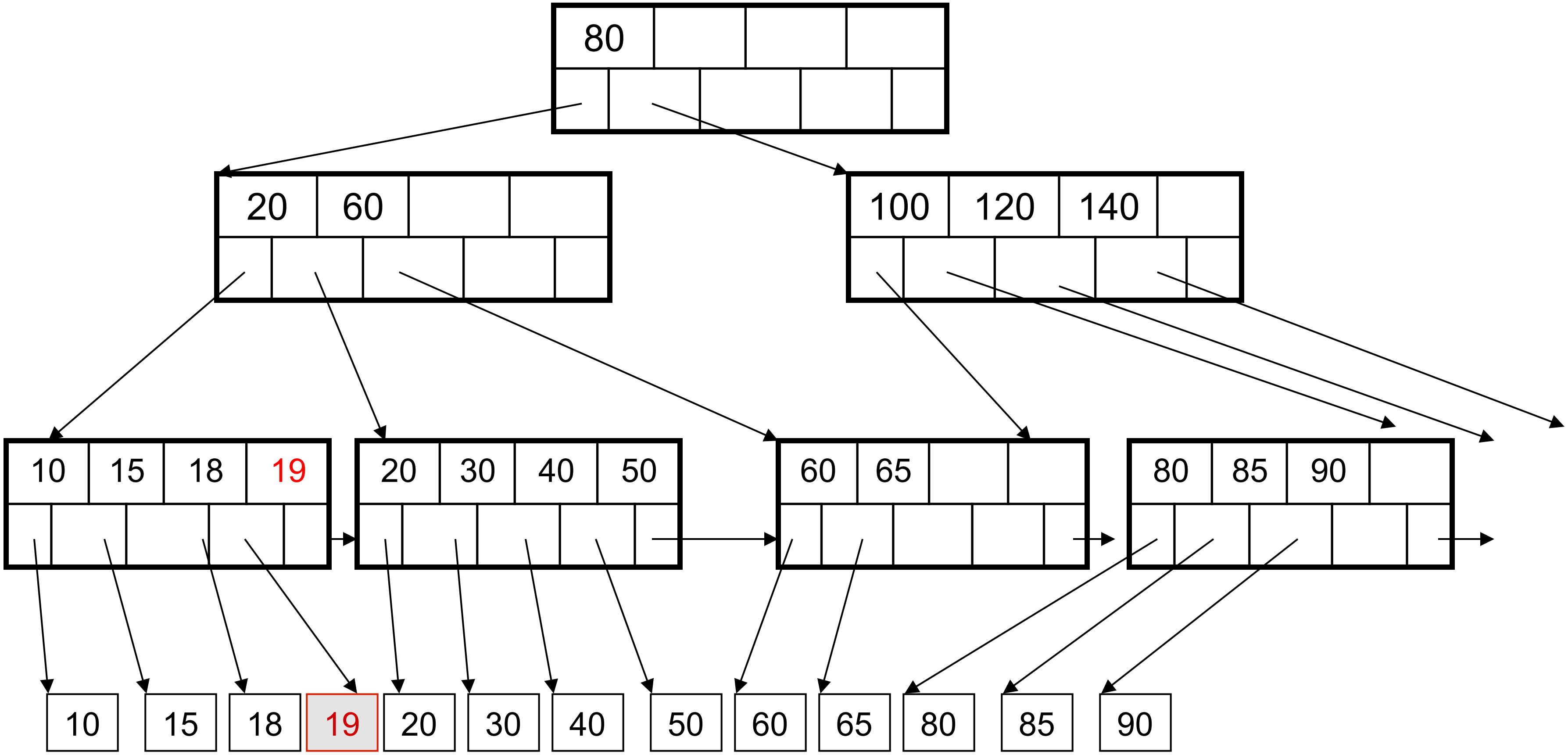
# Insertion in a B+ Tree

Insert K=19



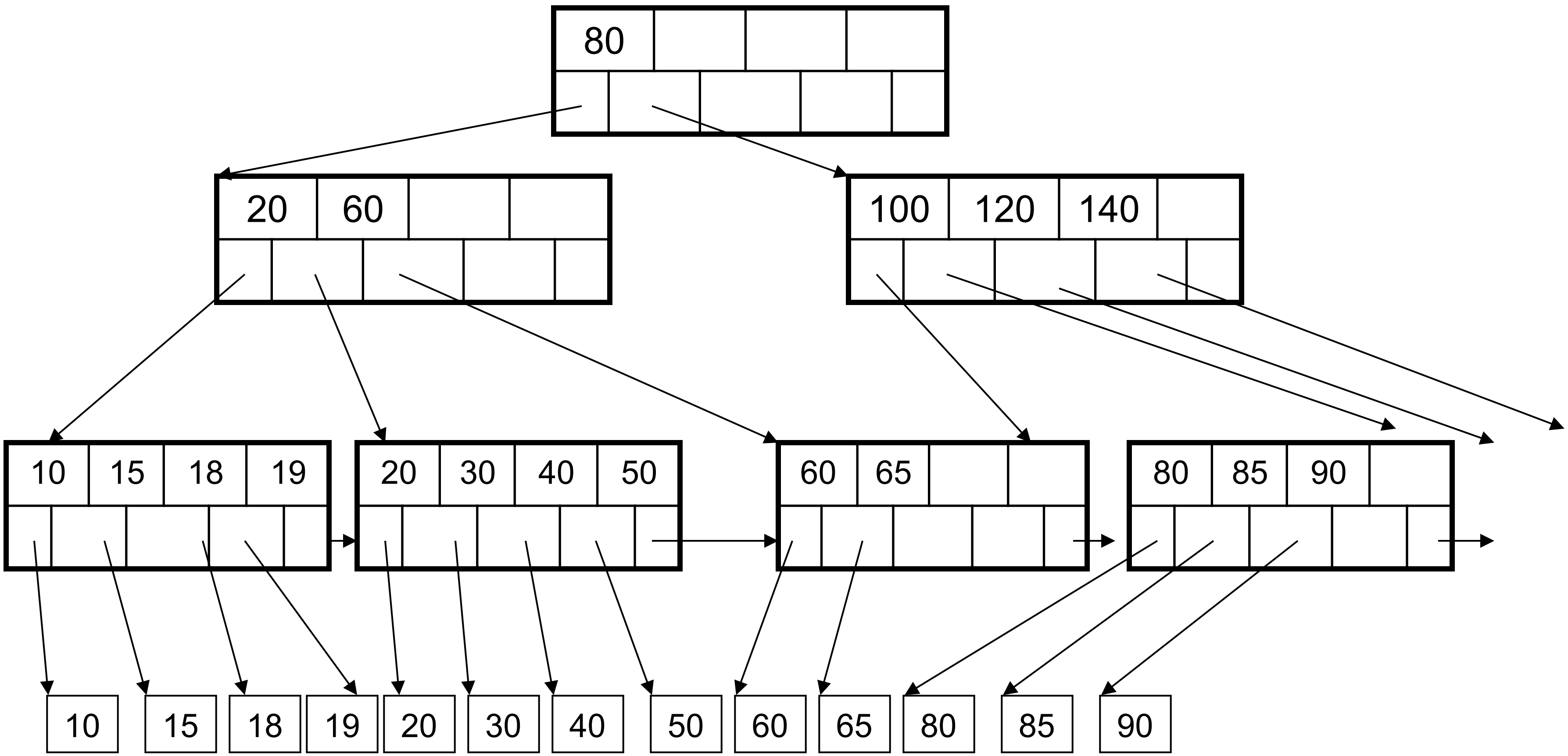
# Insertion in a B+ Tree

After insertion



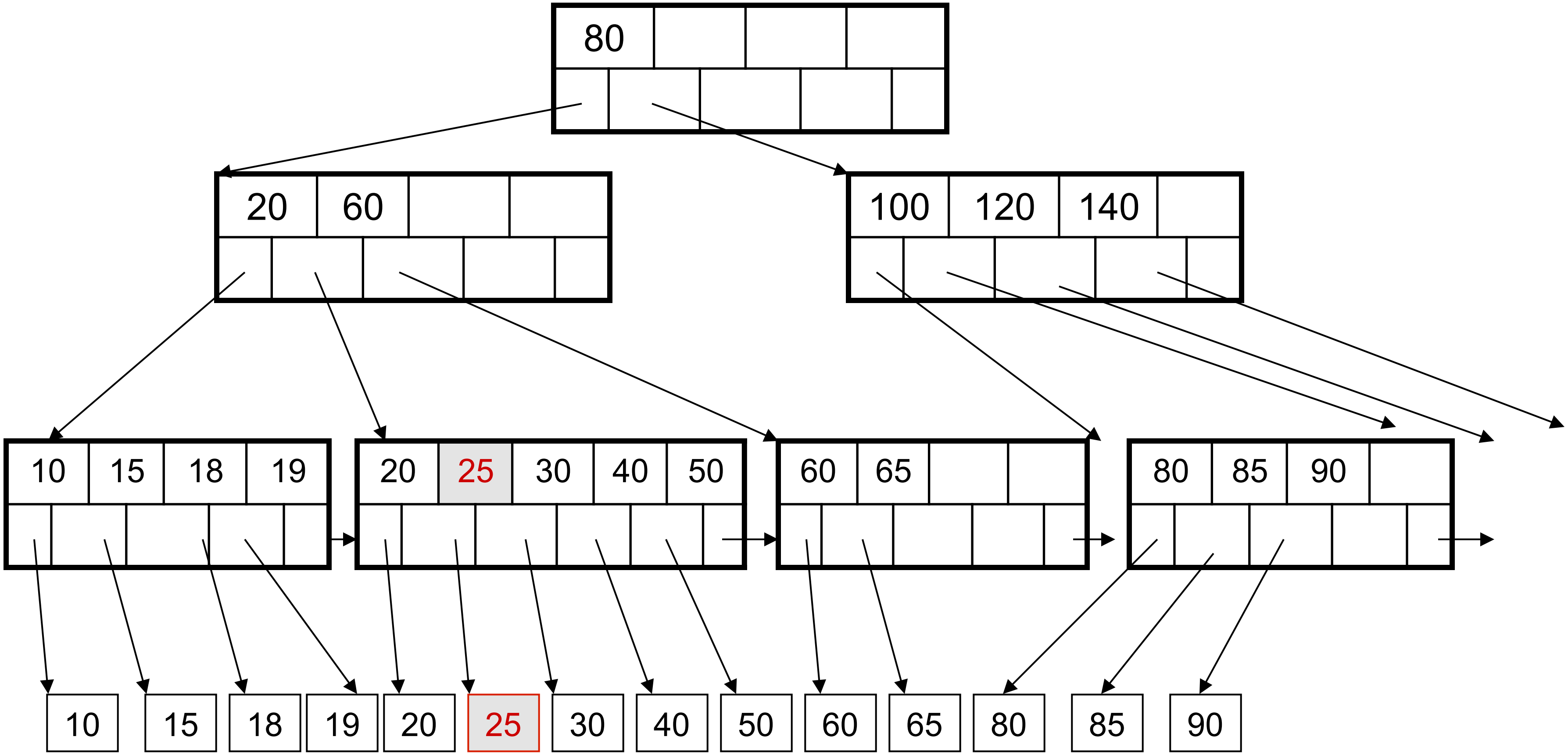
# Insertion in a B+ Tree

Now insert 25



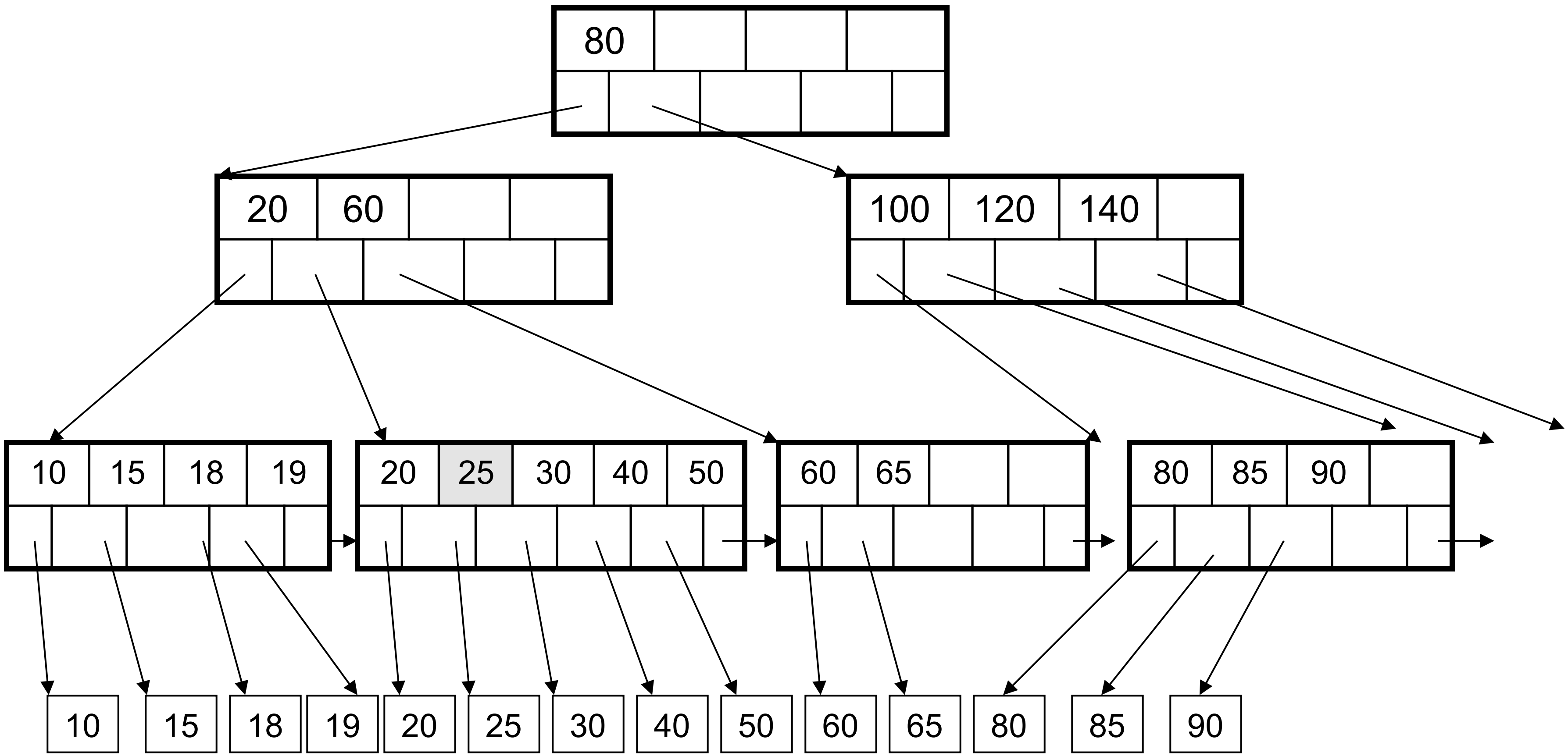
# Insertion in a B+ Tree

After insertion



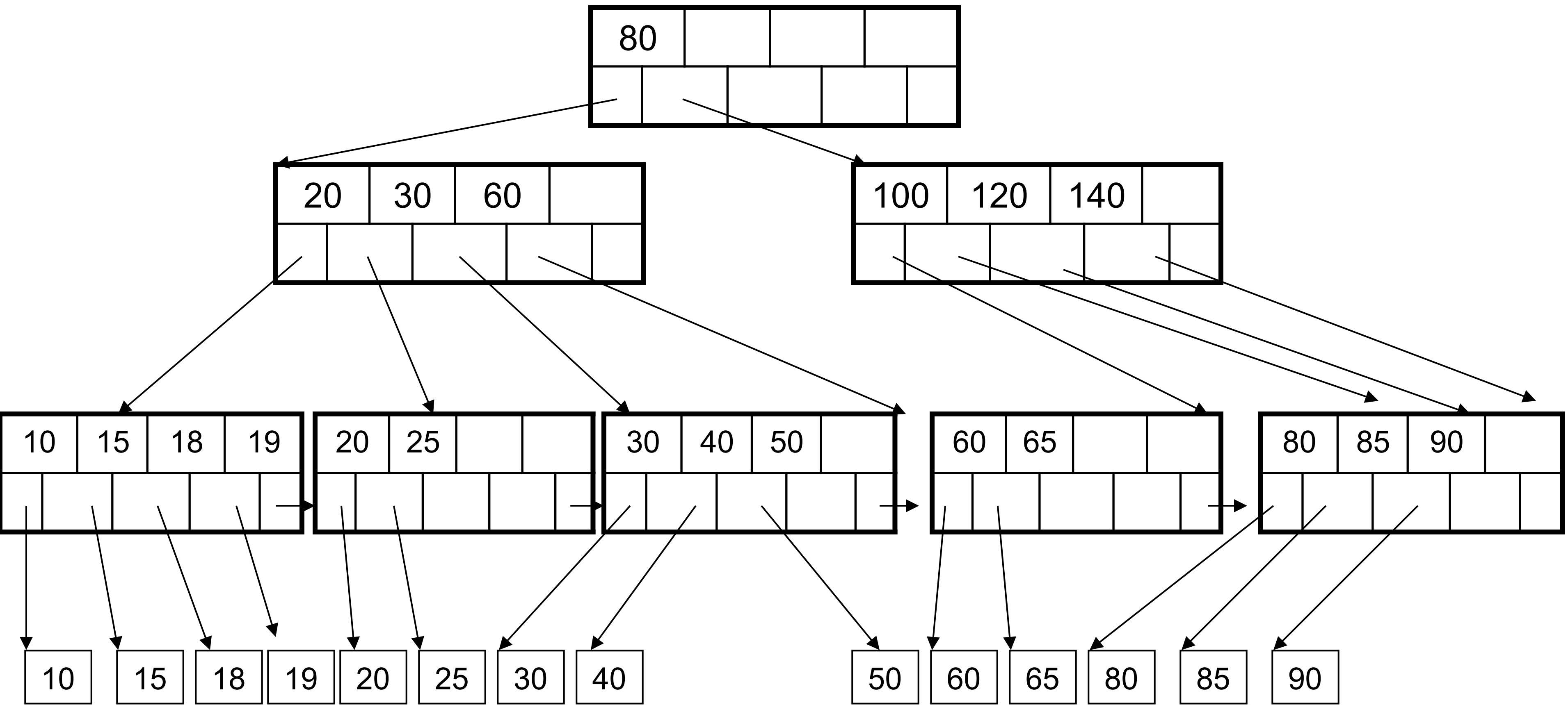
# Insertion in a B+ Tree

But now have to split !



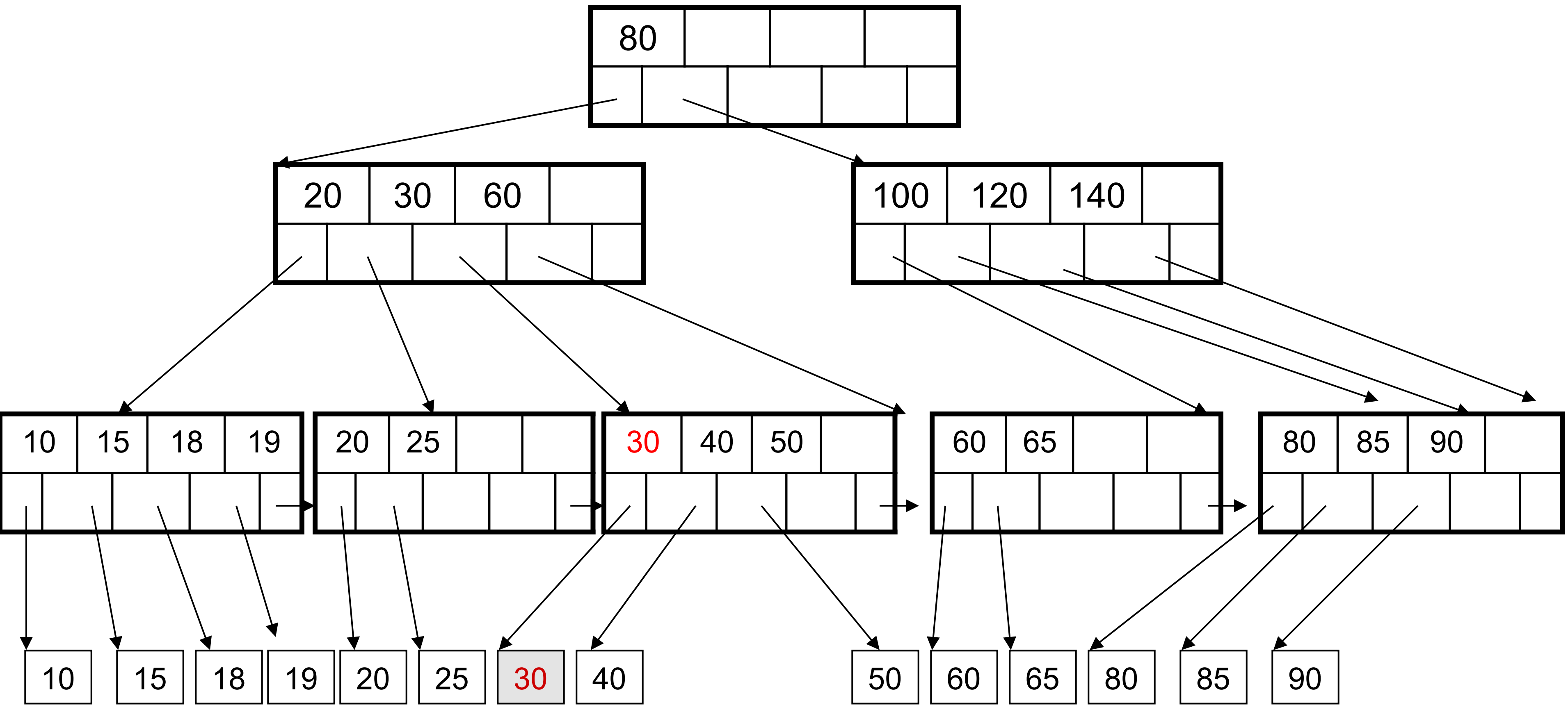
# Insertion in a B+ Tree

After the split



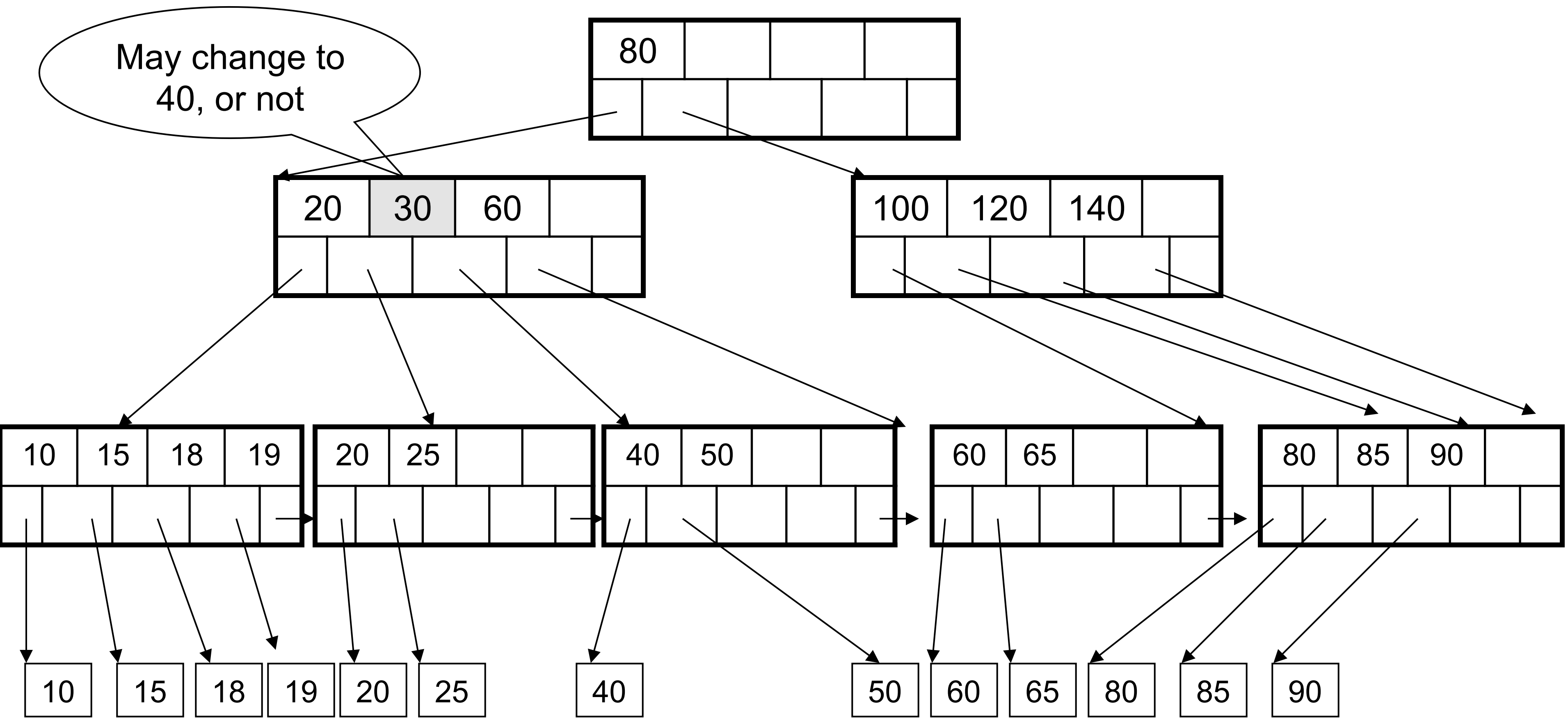
# Deletion from a B+ Tree

Delete 30



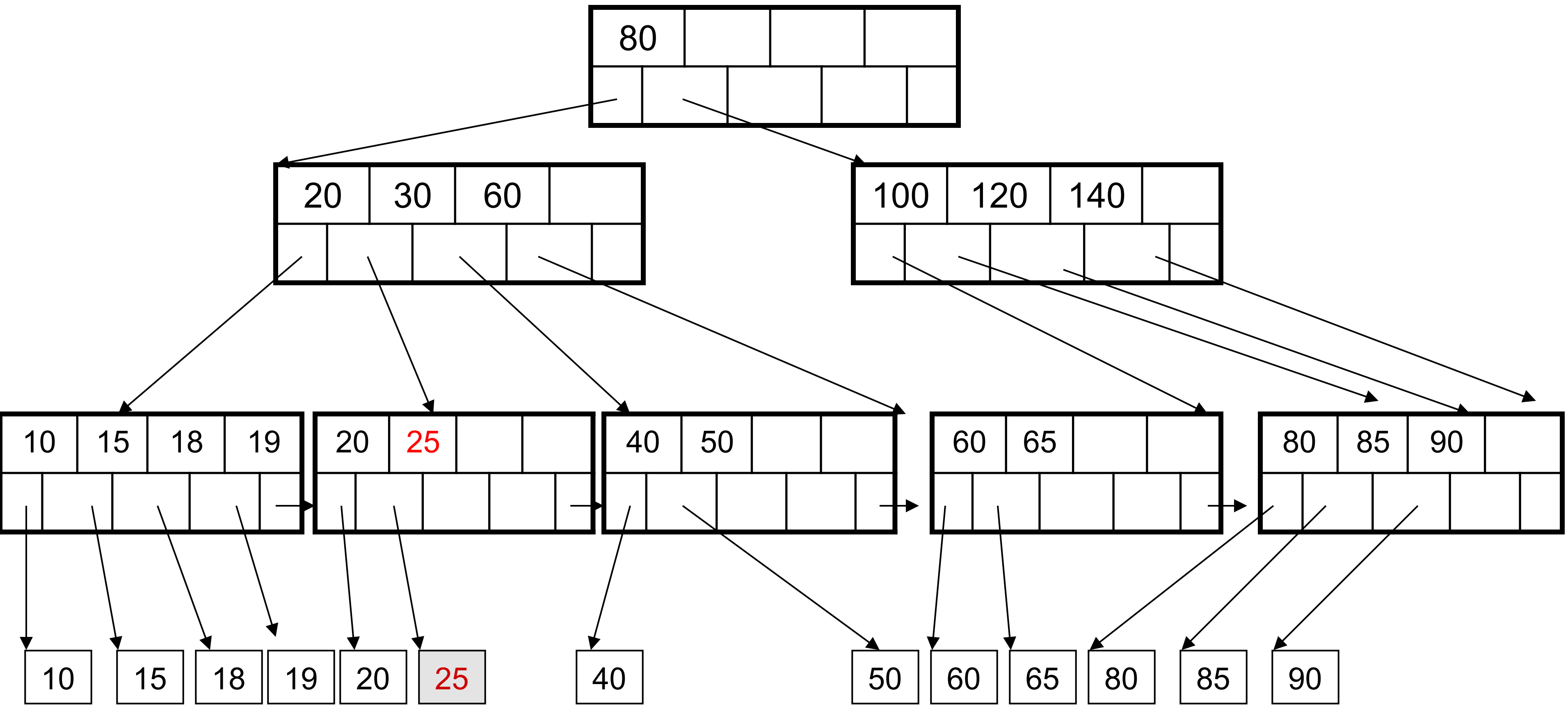
# Deletion from a B+ Tree

After deleting 30



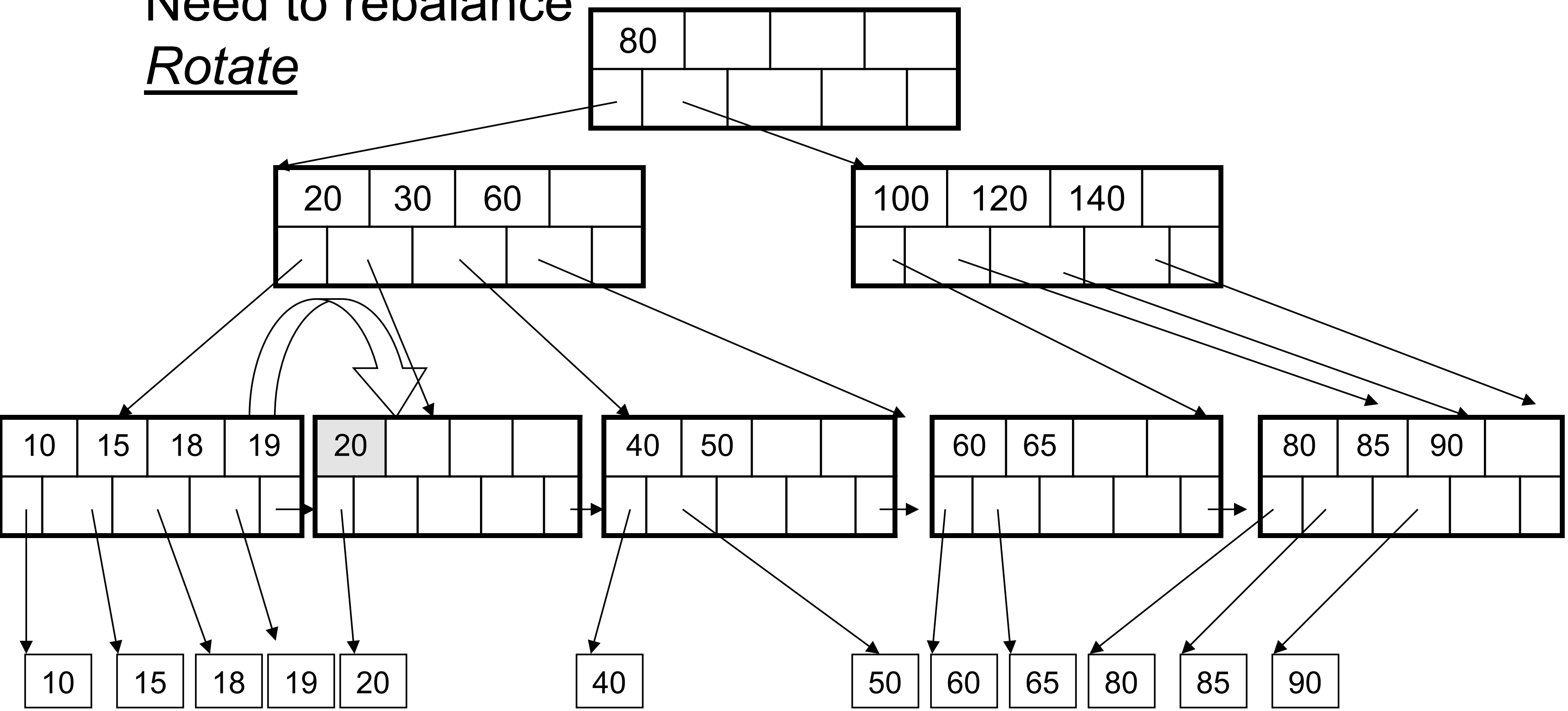
# Deletion from a B+ Tree

Now delete 25



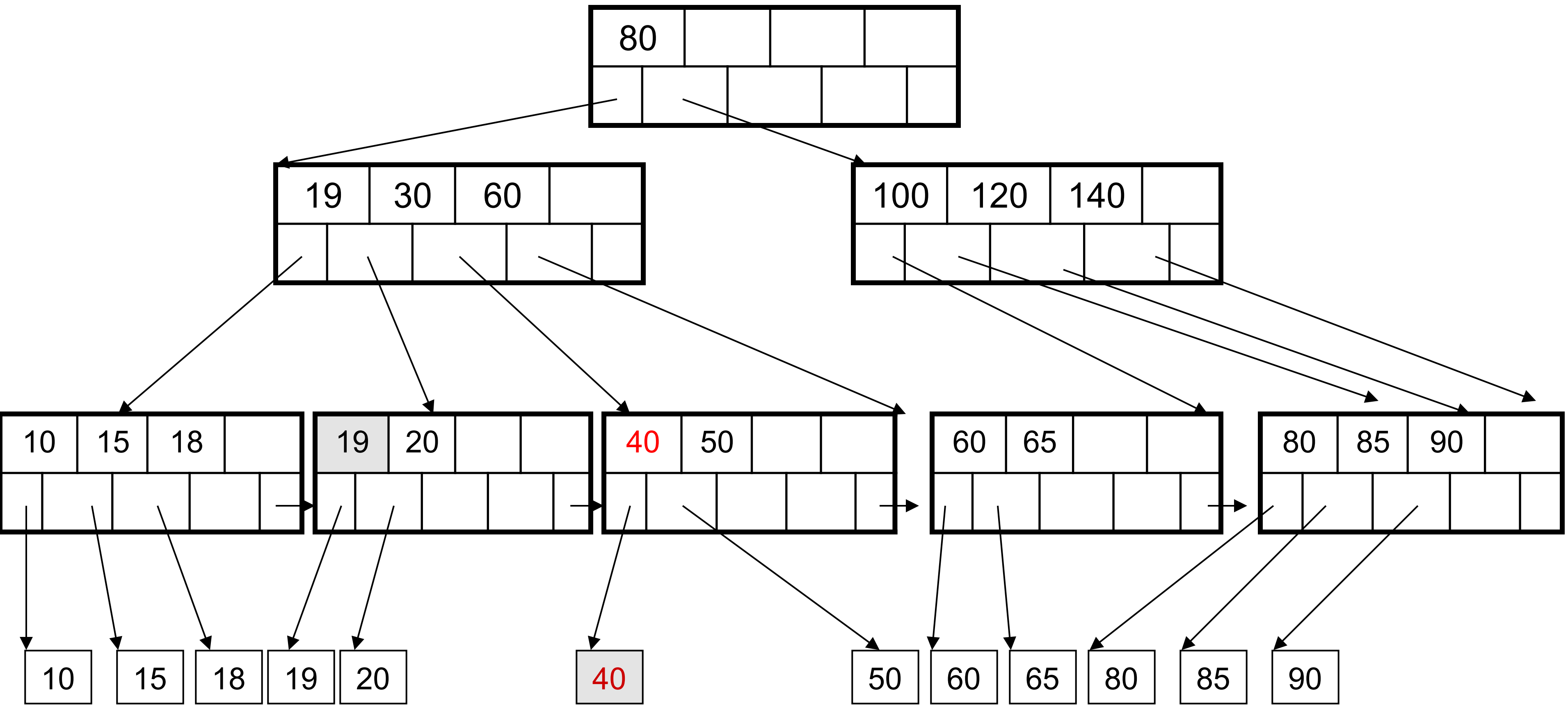
# Deletion from a B+ Tree

After deleting 25
Need to rebalance
*Rotate*



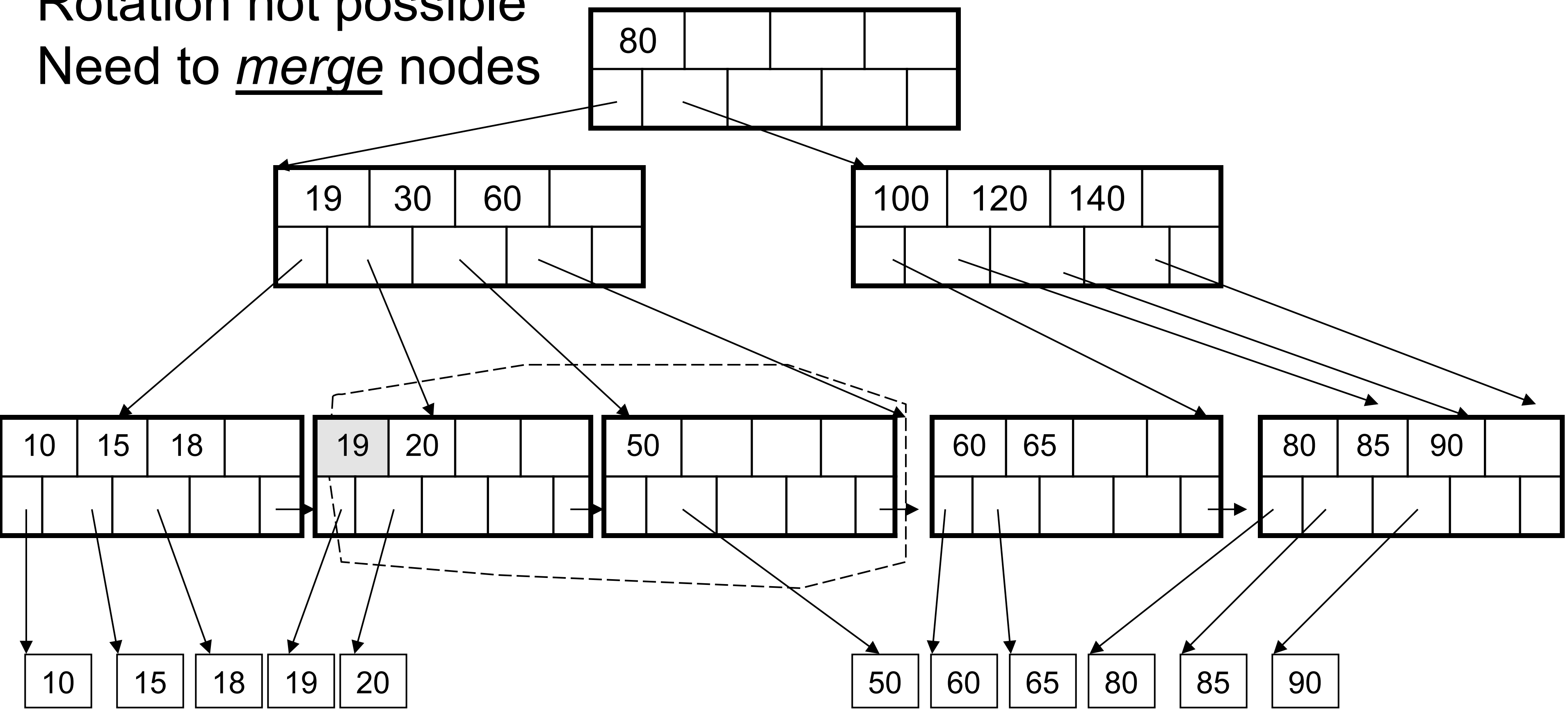
# Deletion from a B+ Tree

Now delete 40



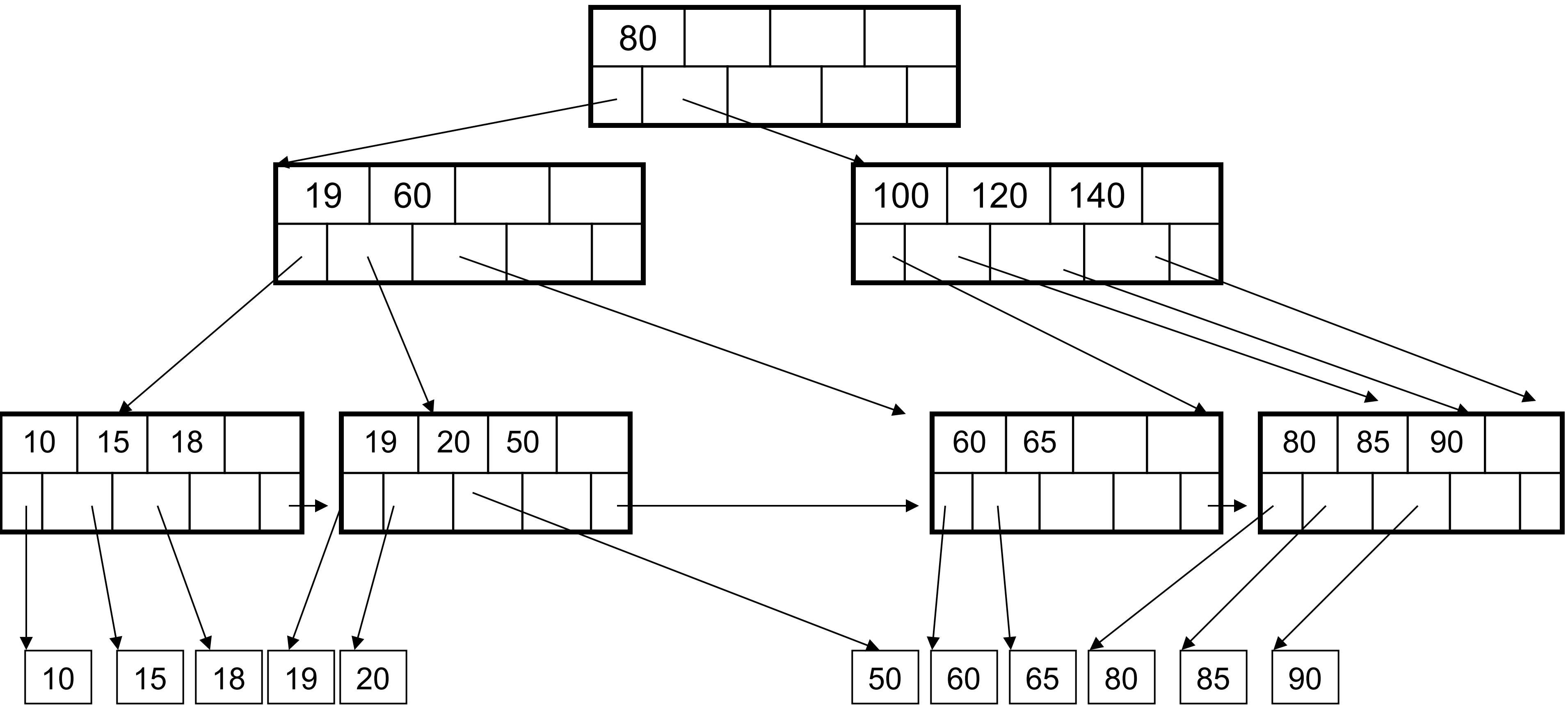
# Deletion from a B+ Tree

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



# Deletion from a B+ Tree

Final tree



# Practical Aspects of B+ Trees

Key compression:

- Each node keeps only the from parent keys
- Jonathan, John, Johnsen, Johnson … →
  - Parent: Jo
  - Child: nathan, hn, hnsen, hnson, …

# Practical Aspects of B+ Trees

Bulk insertion

- When a new index is created there are two options:
  - Start from empty tree, insert each key one-by-one
  - Do *bulk insertion* – what does that mean ?

# Practical Aspects of B+ Trees

Concurrency control

- The root of the tree is a “hot spot”
  - Leads to lock contention during insert/ delete
- Solution: do proactive split during insert, or proactive merge during delete
  - Insert/delete now require only one traversal, from the root to a leaf
  - Use the “tree locking” protocol

# Summary on 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 Postgres

```
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)
```

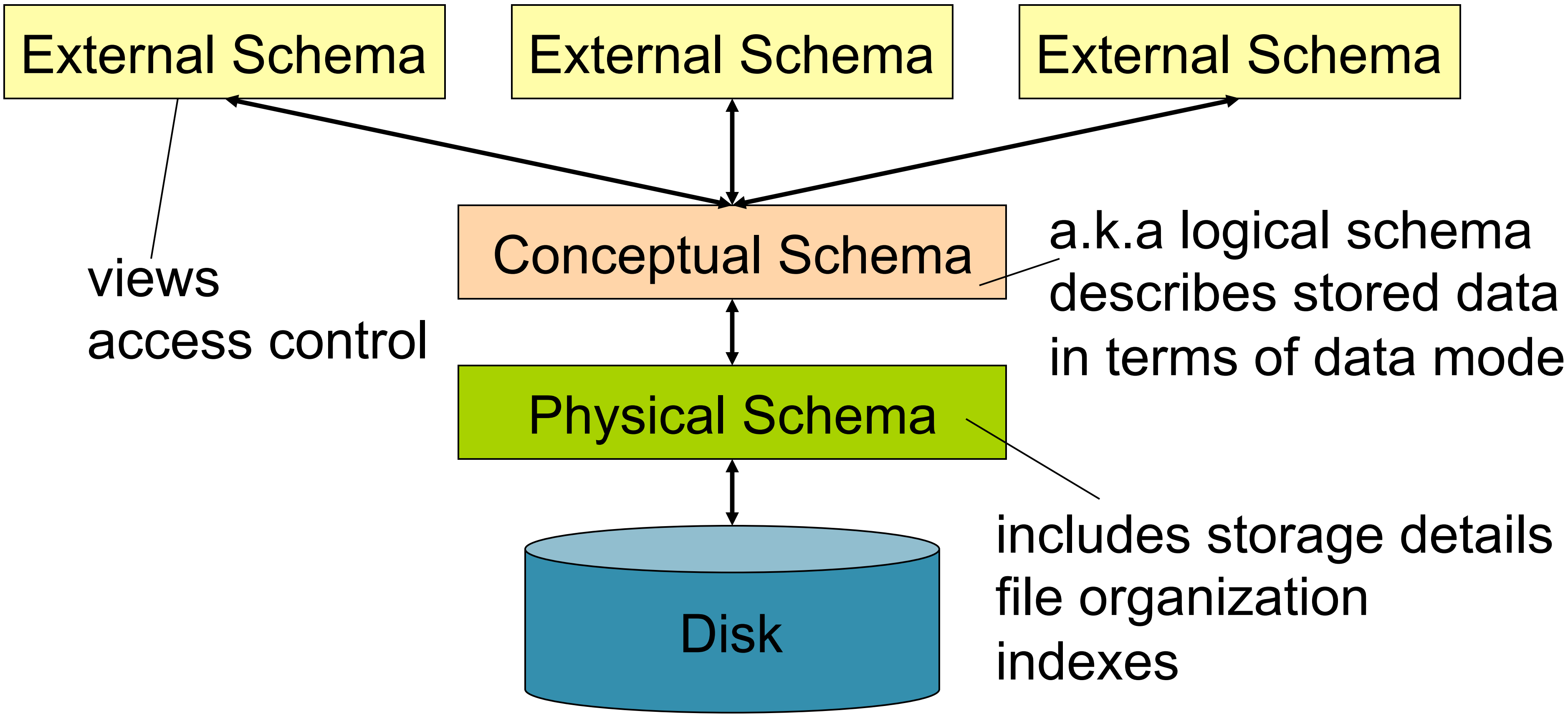
```
CLUSTER V USING V2
```

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

# 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=?
```

**Which indexes should we create?**

# 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)

# 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 (?, ?, ?)
```

Which indexes should we create?

# 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)

# 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 (?, ?, ?)
```

Which indexes should we create?

# 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)

# 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<?
```

Which indexes should we create?

# 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

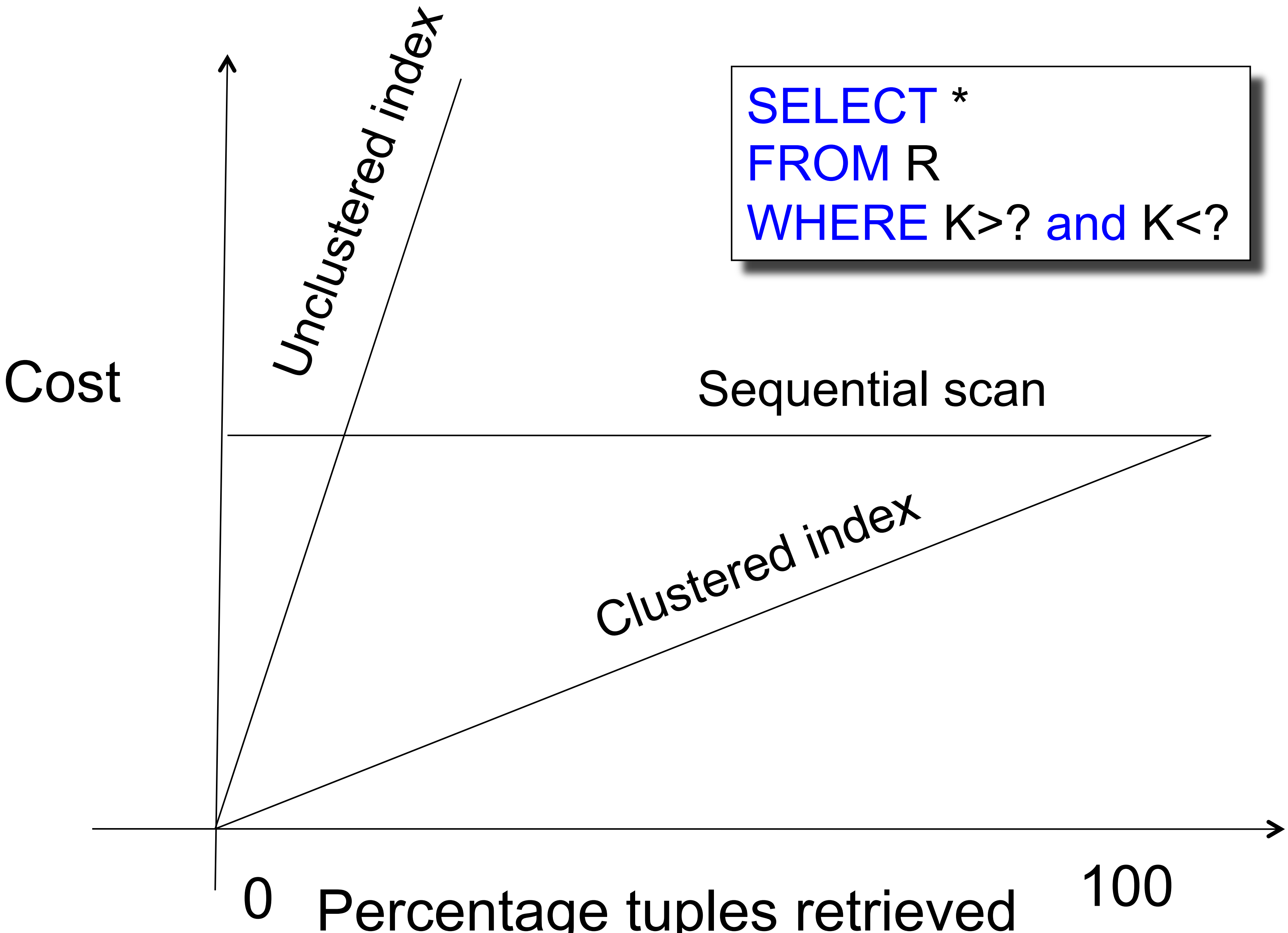
Consider creating a multi-attribute key on K1, K2, … if

- 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

```
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<?
```

# 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, DELECTE, 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