CSE544: Principles of Database Systems

Lectures 6

Database Architecture

Storage and Indexes

Announcements

- Project
  - Choose a topic. Set *limited* goals!
  - Sign up (doodle) to meet with me next Wednesday
- Homework 1
  - Due on Monday
- Paper review
  - Due next Wednesday

Where We Are

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

Outline

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

The Mechanics of Disk

Mechanical characteristics:

- Rotation speed (5400RPM)
- Number of platters (1-30)
- Number of tracks (<=10000)
- Number of bytes/track(10^5)

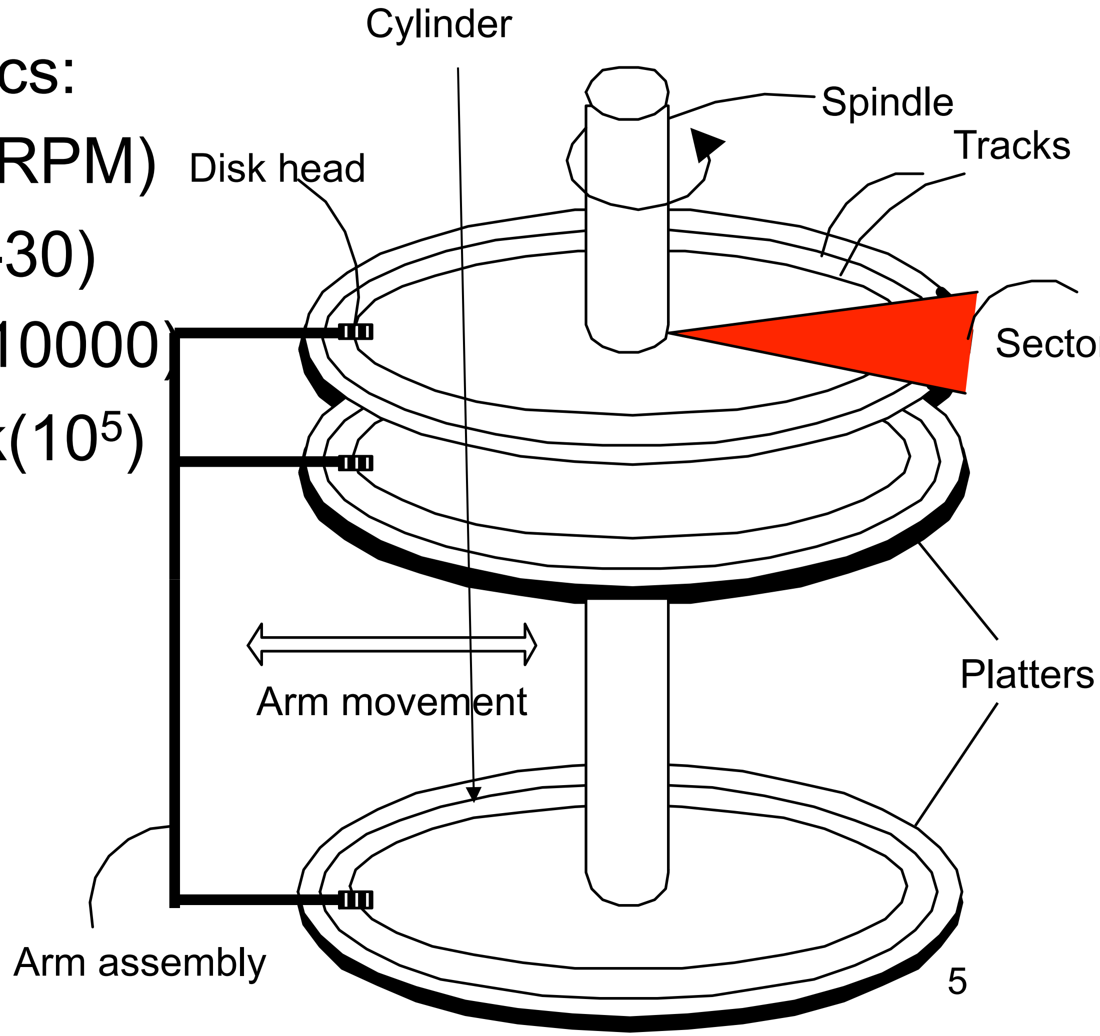
Unit of read or write:

disk block

Once in memory:

page

Typically: 4k or 8k or 16k



Disk Access Characteristics

- Disk latency
  - Time between when command is issued and when data is in memory
  - Equals = seek time + rotational latency
- Seek time = time for the head to reach cylinder
  - 10ms – 40ms
- Rotational latency = time for the sector to rotate
    - Rotation time = 10ms
    - Average latency = 10ms/2
- Transfer time = typically 40MB/s

Basic factoid: disks always read/write an entire block at a time

RAID

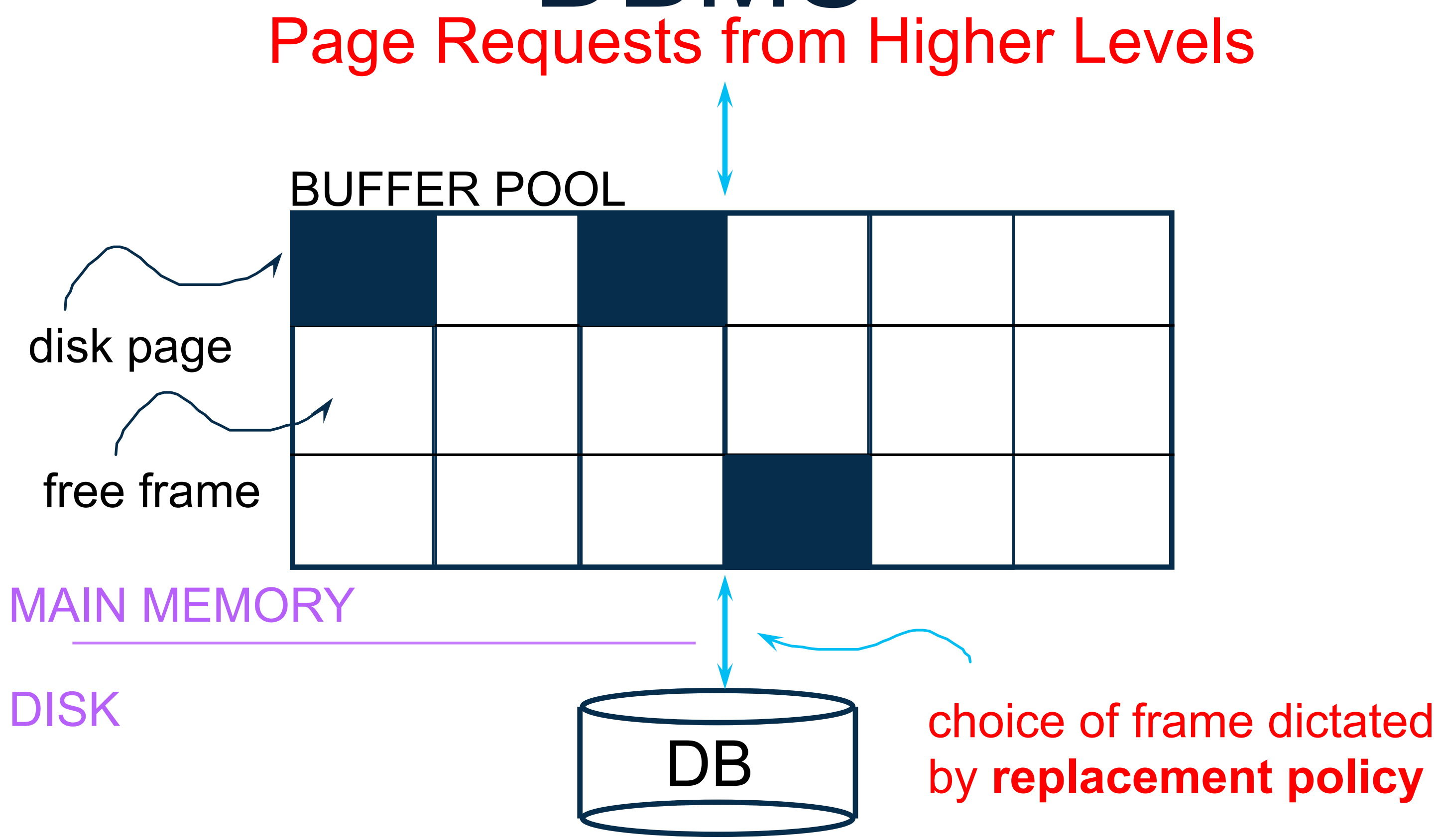
Several disks that work in parallel

- Redundancy: use parity to recover from disk failure
- Speed: read from several disks at once

Various configurations (called *levels*):

- RAID 1 = mirror
- RAID 4 = n disks + 1 parity disk
- RAID 5 = n+1 disks, assign parity blocks round robin
- RAID 6 = “Hamming codes”

Buffer Management in a DBMS



- Data must be in RAM for DBMS to operate on it!
- Table of <frame#, pageid> pairs is maintained

Buffer Manager

Needs to decide on page replacement policy

- LRU
- Clock algorithm

Both work well in OS, but not always in DB

Enables the higher levels of the DBMS to assume that the needed data is in main memory.

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

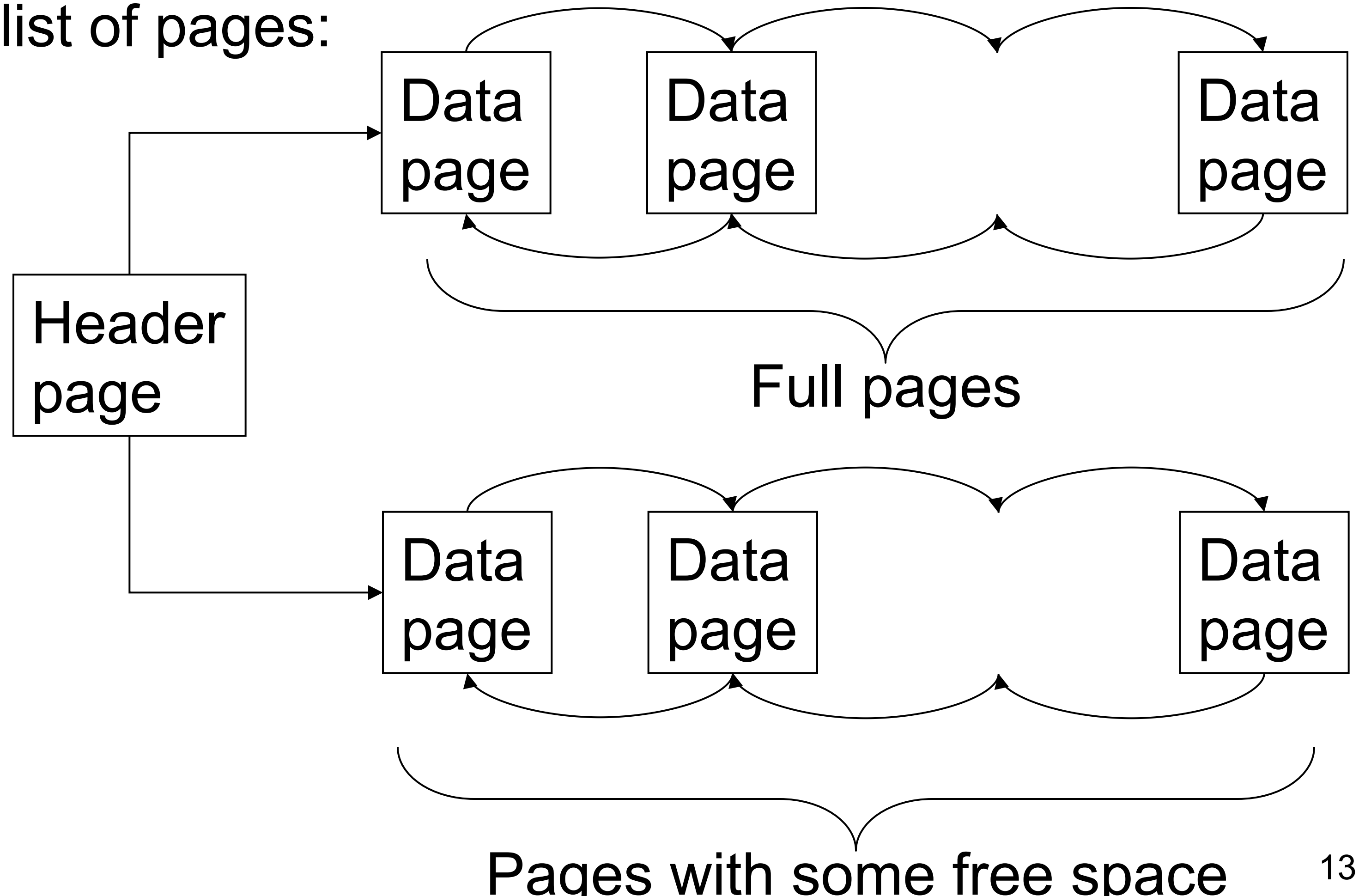
- Managing free blocks
- File Organization
- Represent the records inside the blocks
- Represent attributes inside the records

Managing Free Blocks

- Linked list of free blocks
- Or bit map

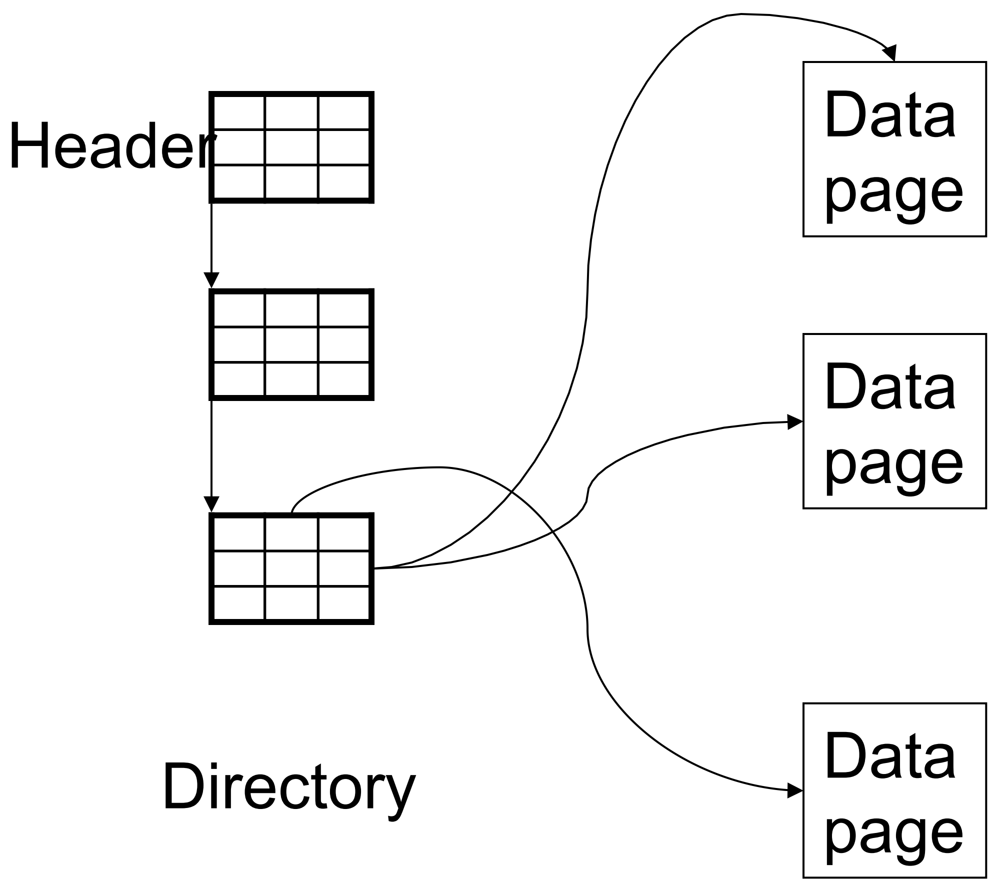
File Organization

Linked list of pages:



File Organization

Better: directory of pages



Page Formats

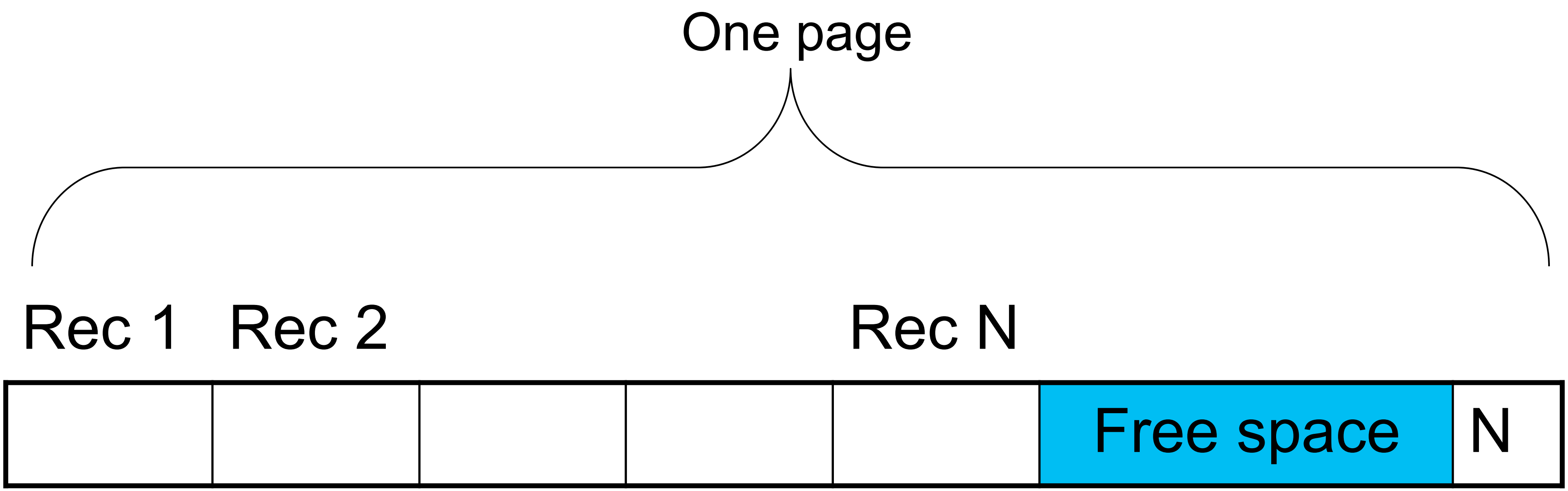
Issues to consider

- 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 ?

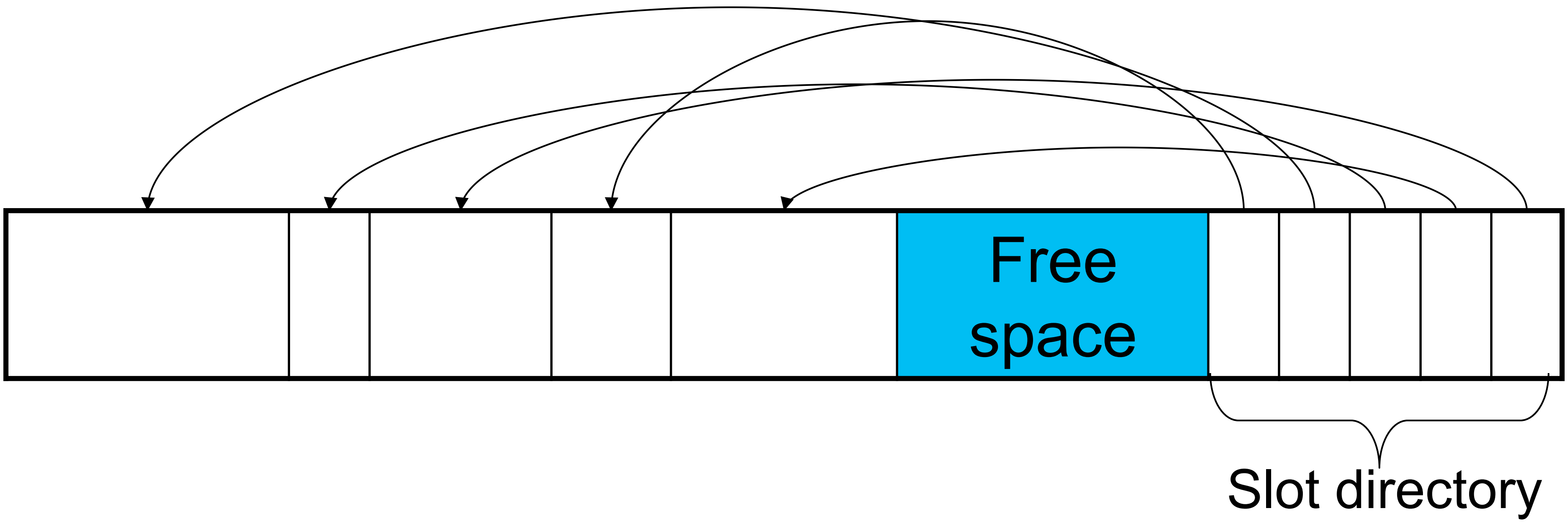
Page Formats

Fixed-length records: packed representation



Problems ?

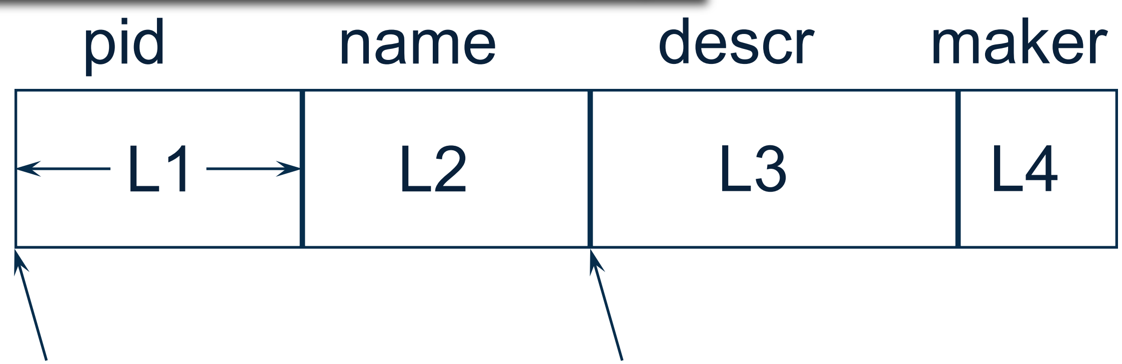
Page Formats



Variable-length records

Record Formats: Fixed Length

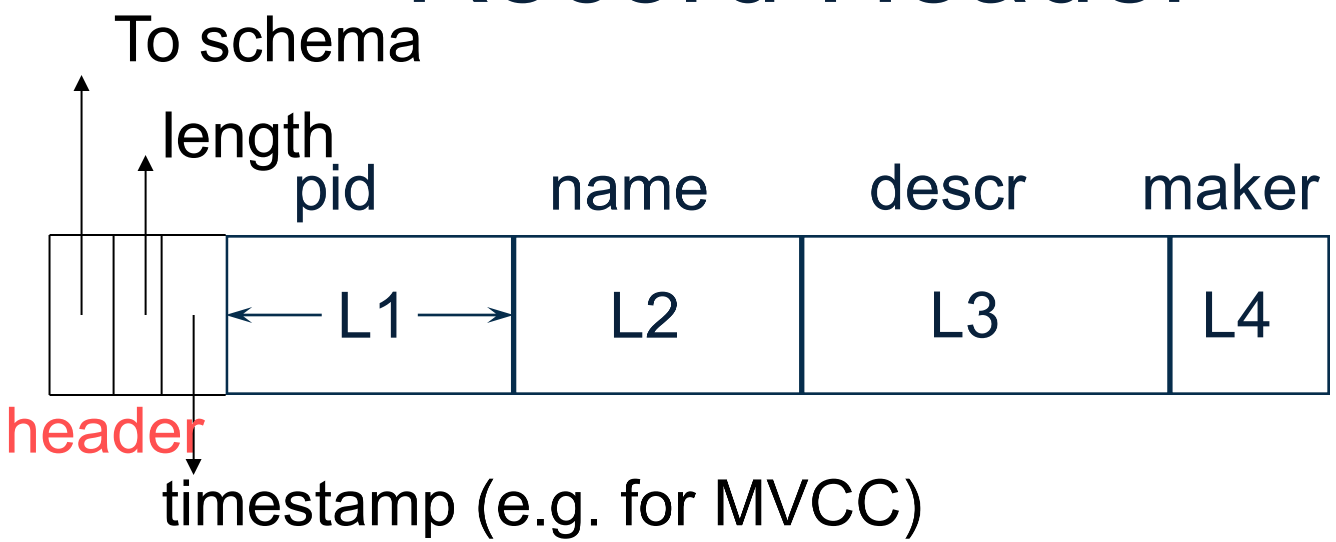
Product(pid, name, descr, maker)



Base address (B)   Address = B+L1+L2

- 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!

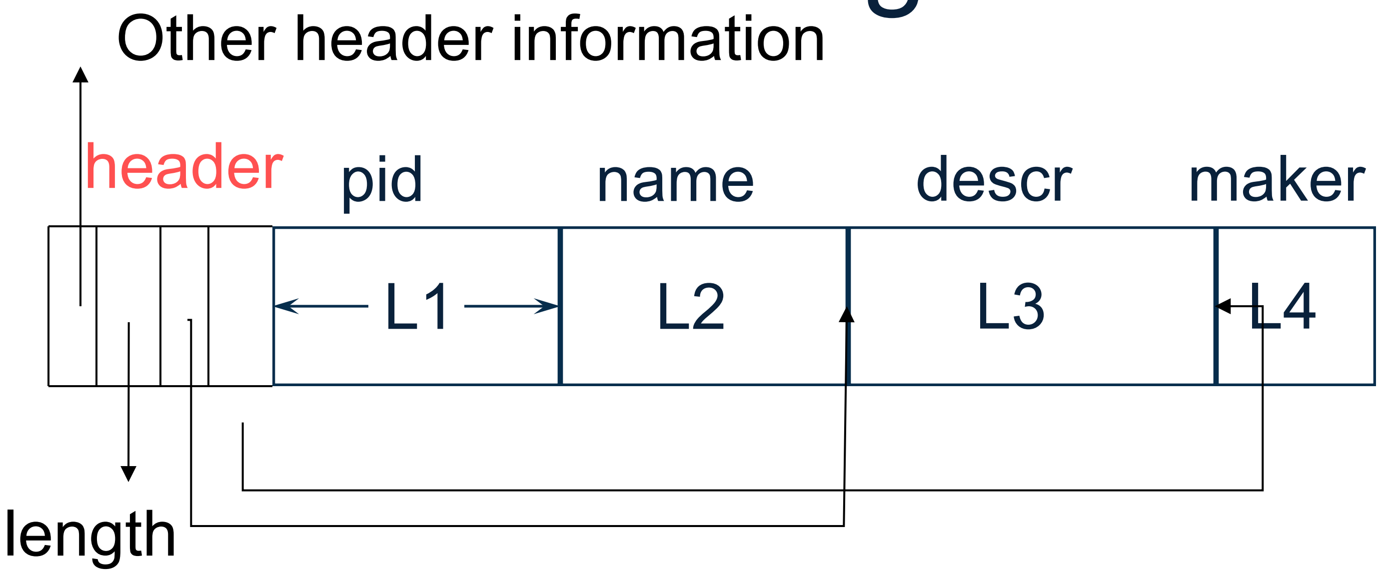
Record Header



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*

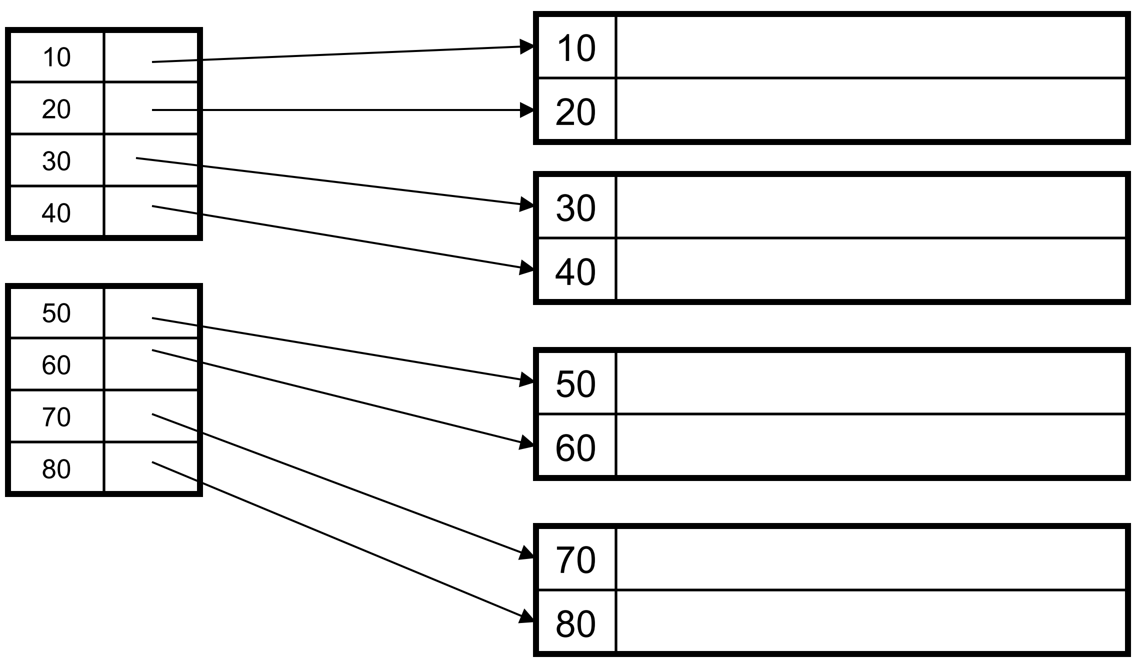
Note: “key” (aka “search key”) again means something else

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

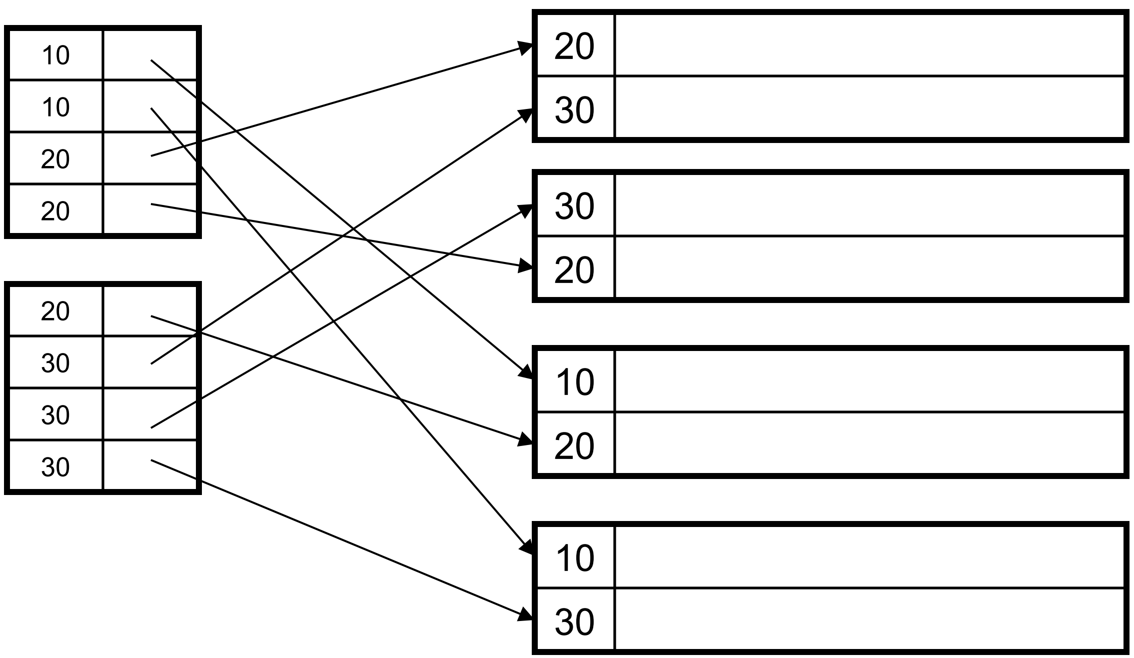
Clustered Index

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

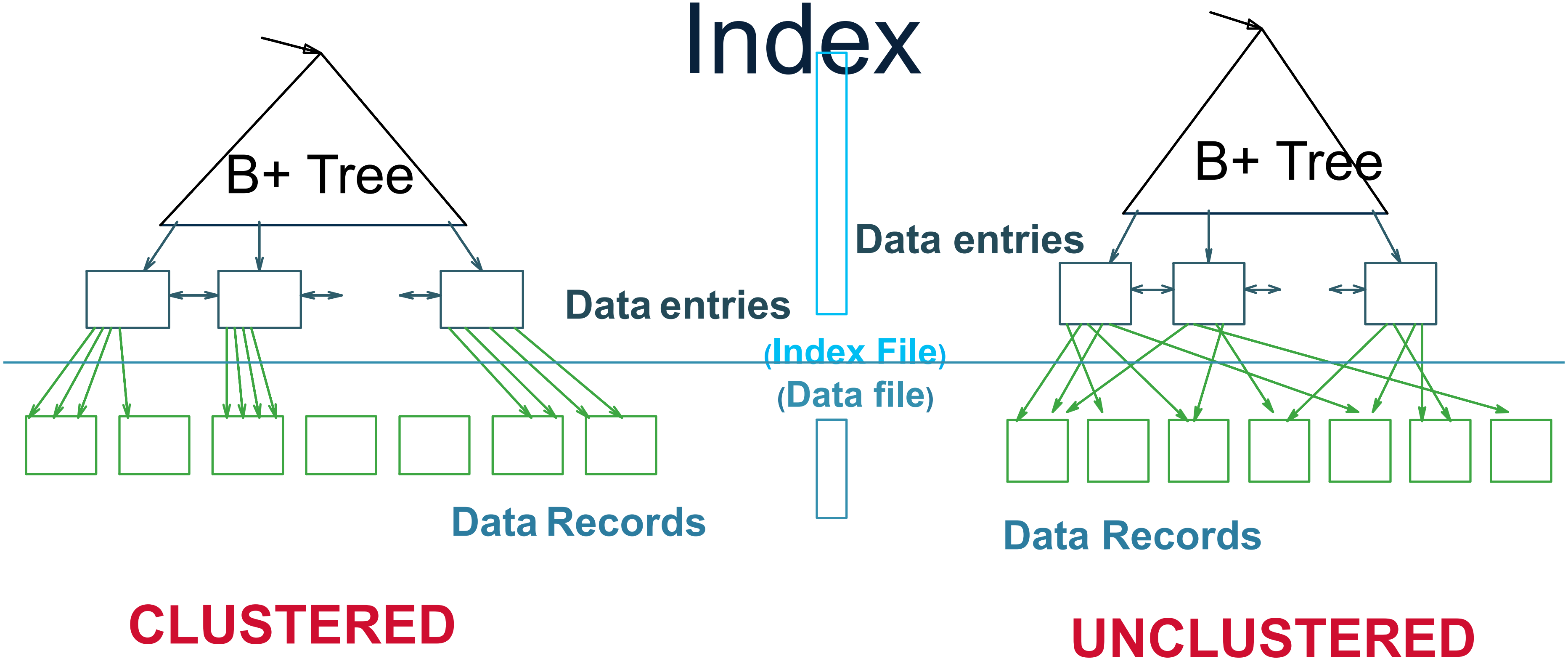


Unclustered Index

- Several per table

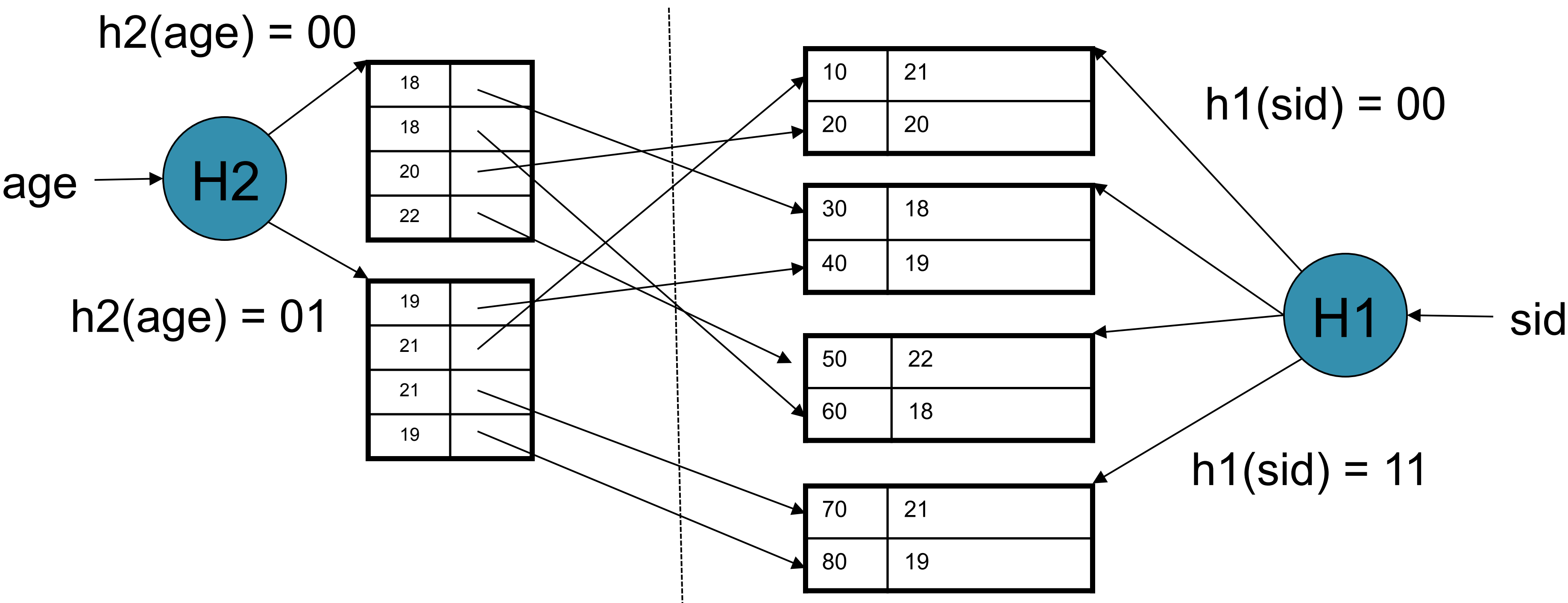


Clustered vs. Unclustered Index



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

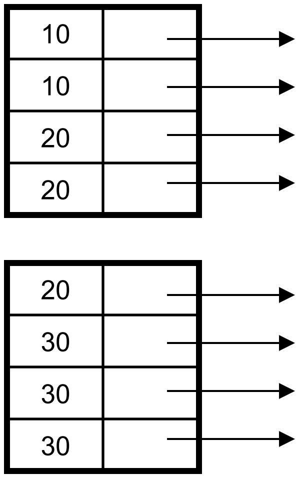
Three alternatives for **k***:

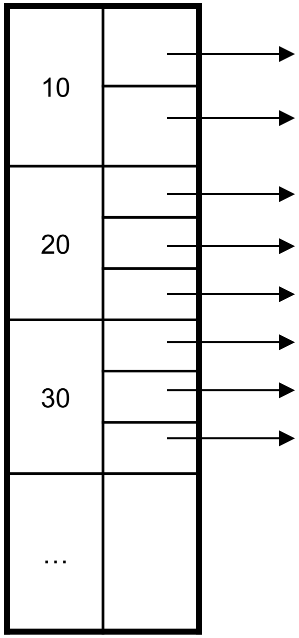
- Data record with key value **k**
- <**k**, **rid** of data record with key = **k**>
- <**k**, list of **rids** of data records with key = **k**>

Alternatives 1, 2, 3

10	ssn	age	…
10	ssn	age	…
20	ssn	age	…
20	ssn	age	…

20	ssn	age	…
30	ssn	age	…
30	ssn	age	…
30	ssn	age	…



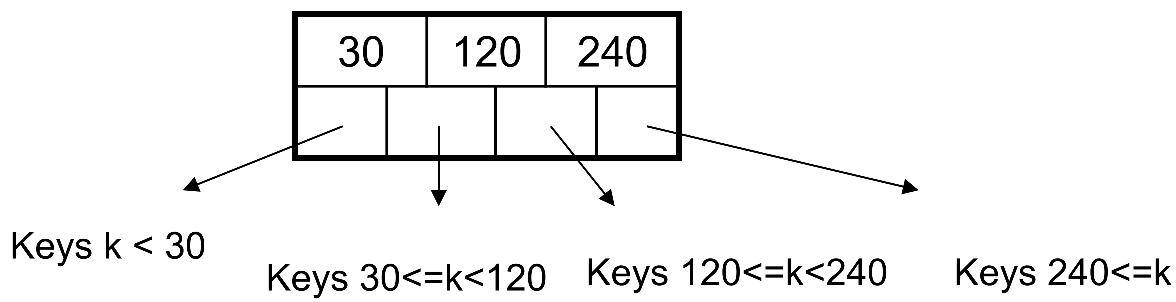


B+ Trees

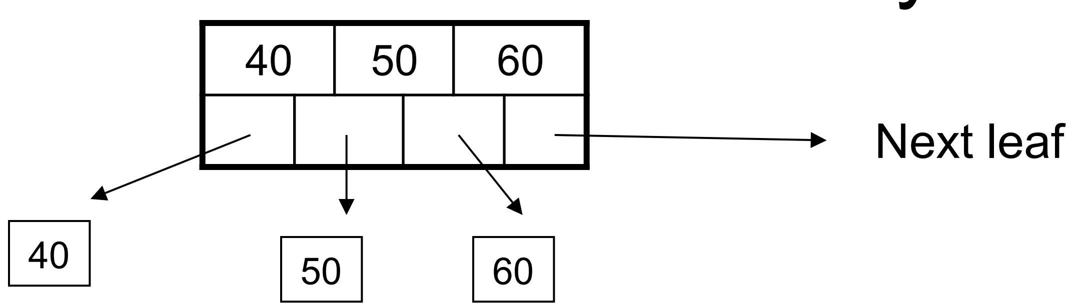
- Search trees
- Idea in B Trees
  - Make 1 node = 1 block
  - Keep tree balanced in height
- Idea in B+ Trees
  - Make leaves into a linked list: facilitates range queries

B+ Trees Basics

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



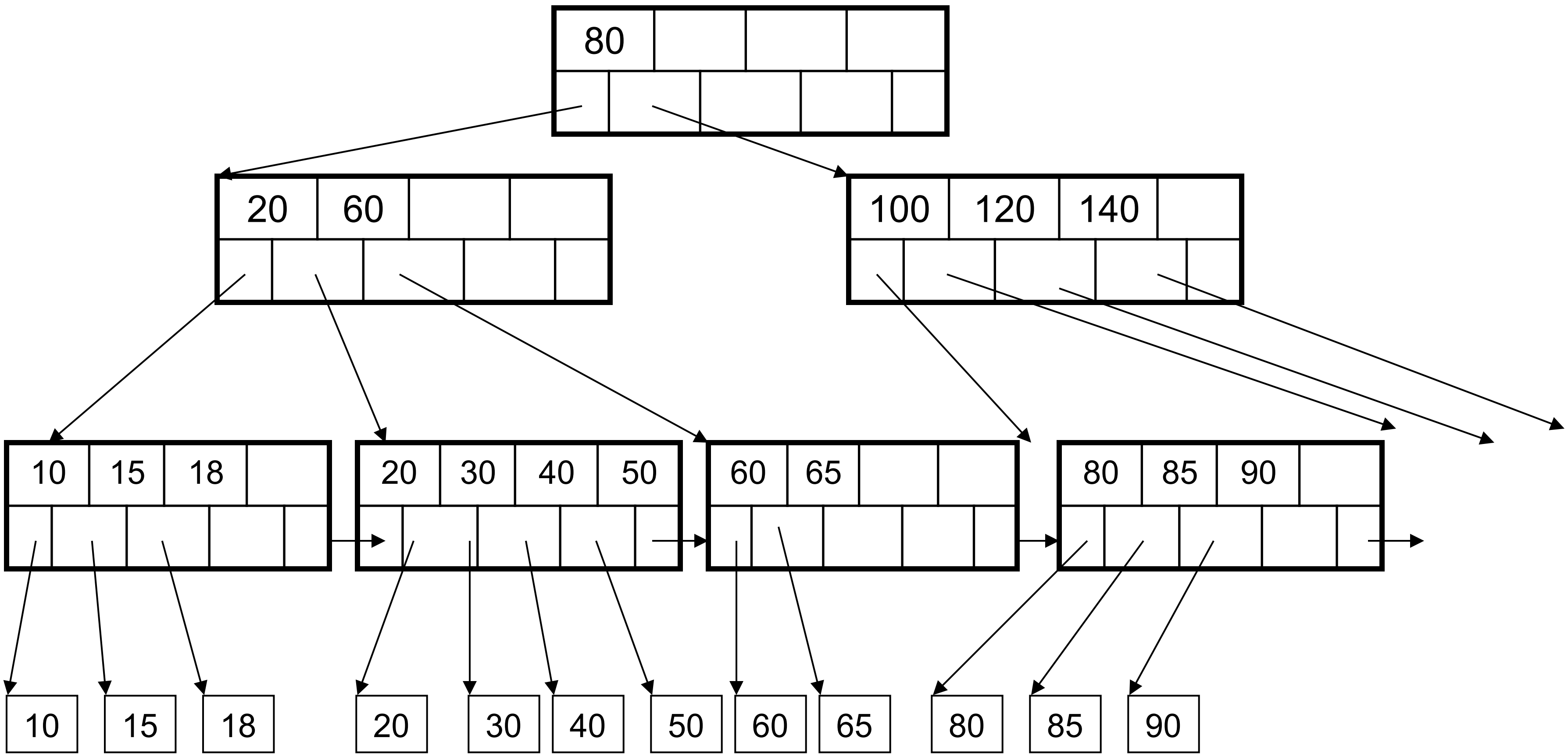
- Each leaf has >=d and <= 2d keys:



B+ Tree Example

d = 2

Find the key 40

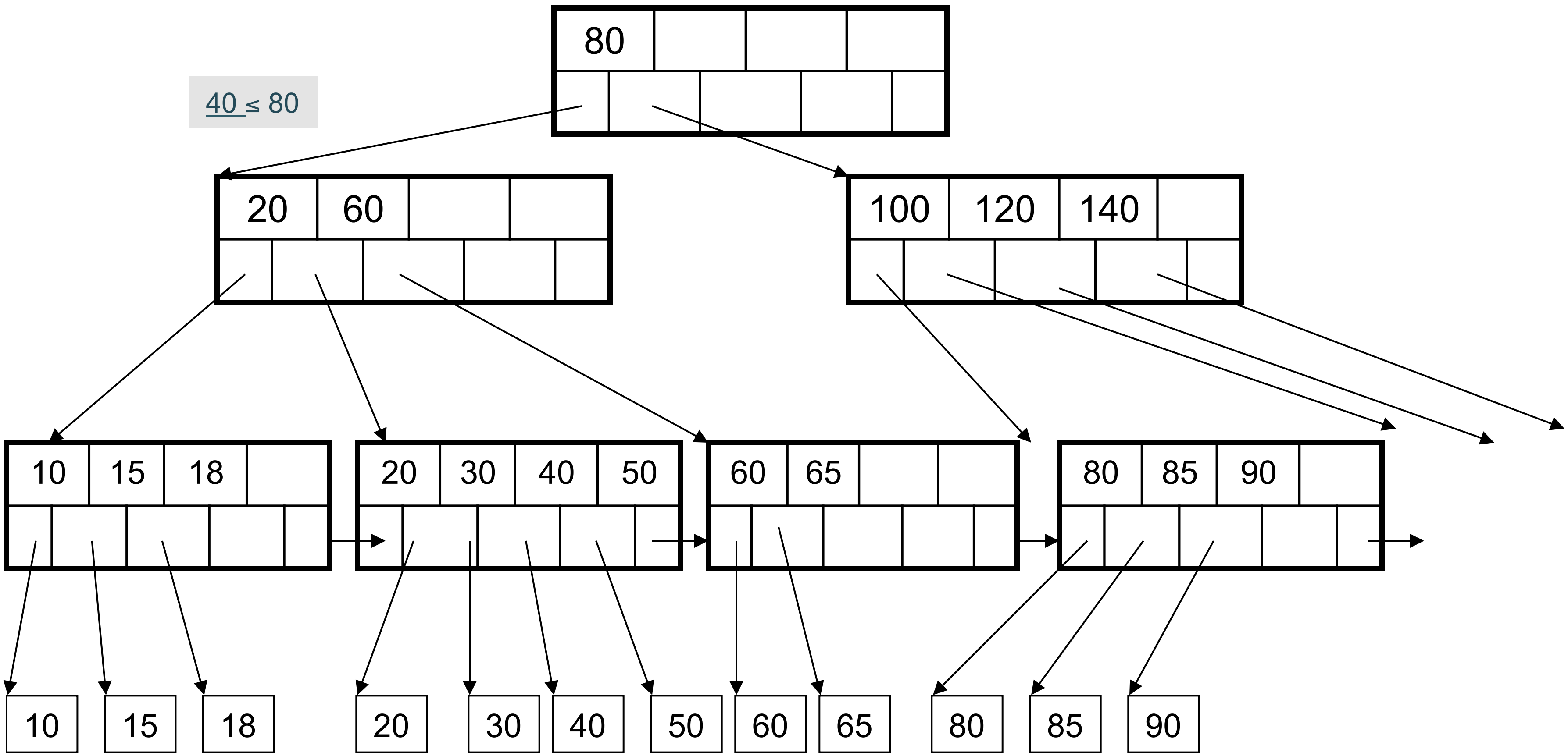


B+ Tree Example

d = 2

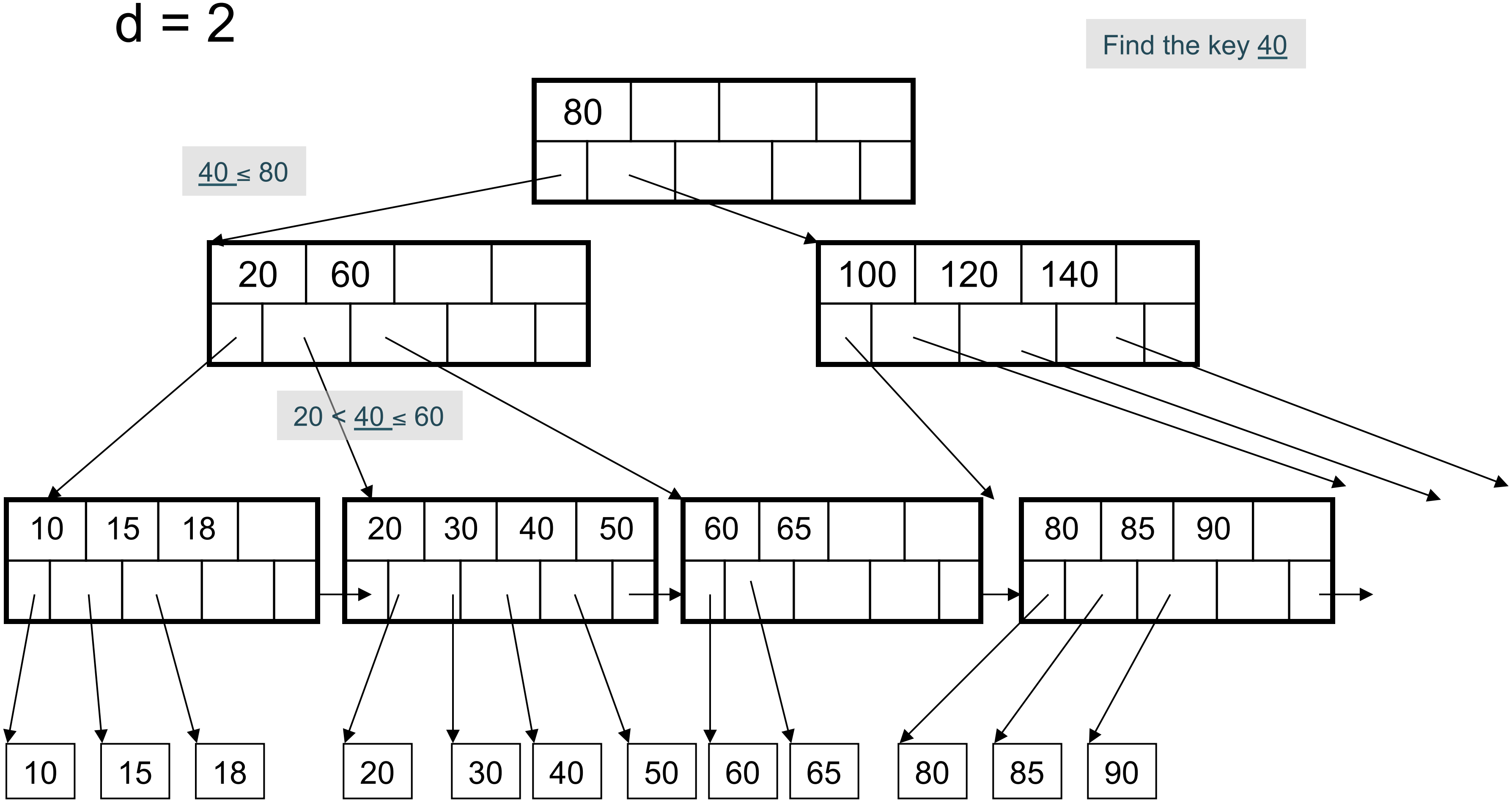
Find the key 40

40 ≤ 80



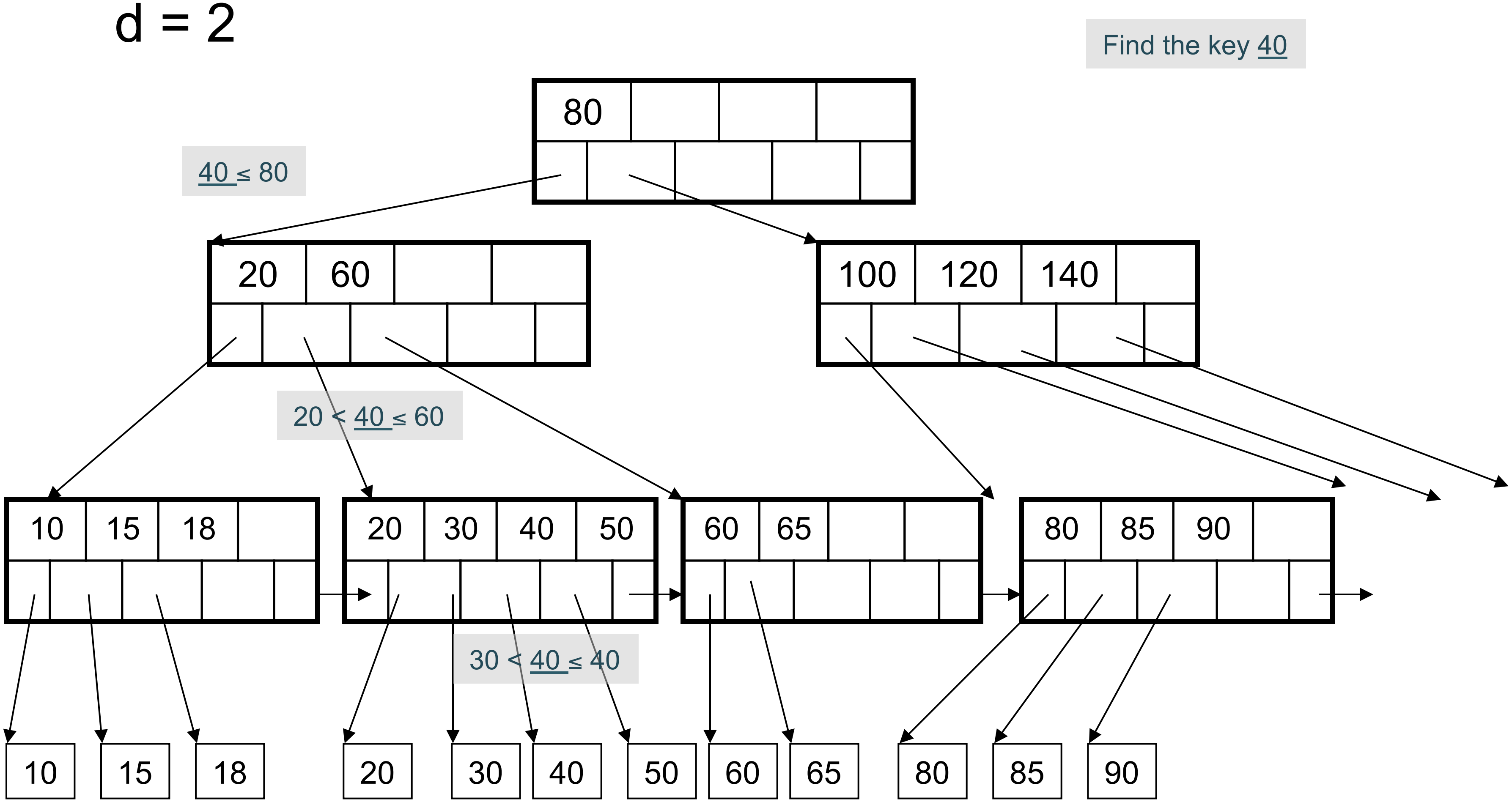
B+ Tree Example

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
```

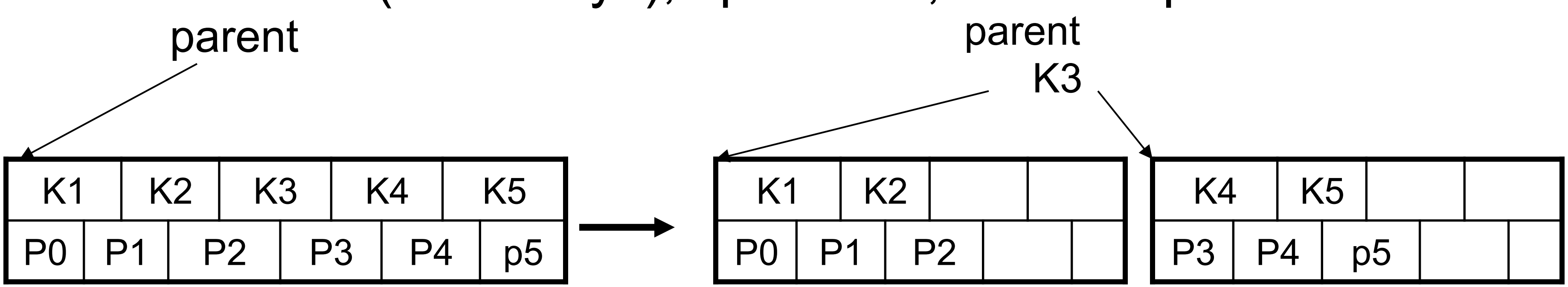
```
SELECT *
FROM People
WHERE name = ‘Smith’
```

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

Insertion in a B+ Tree

Insert (K, P)

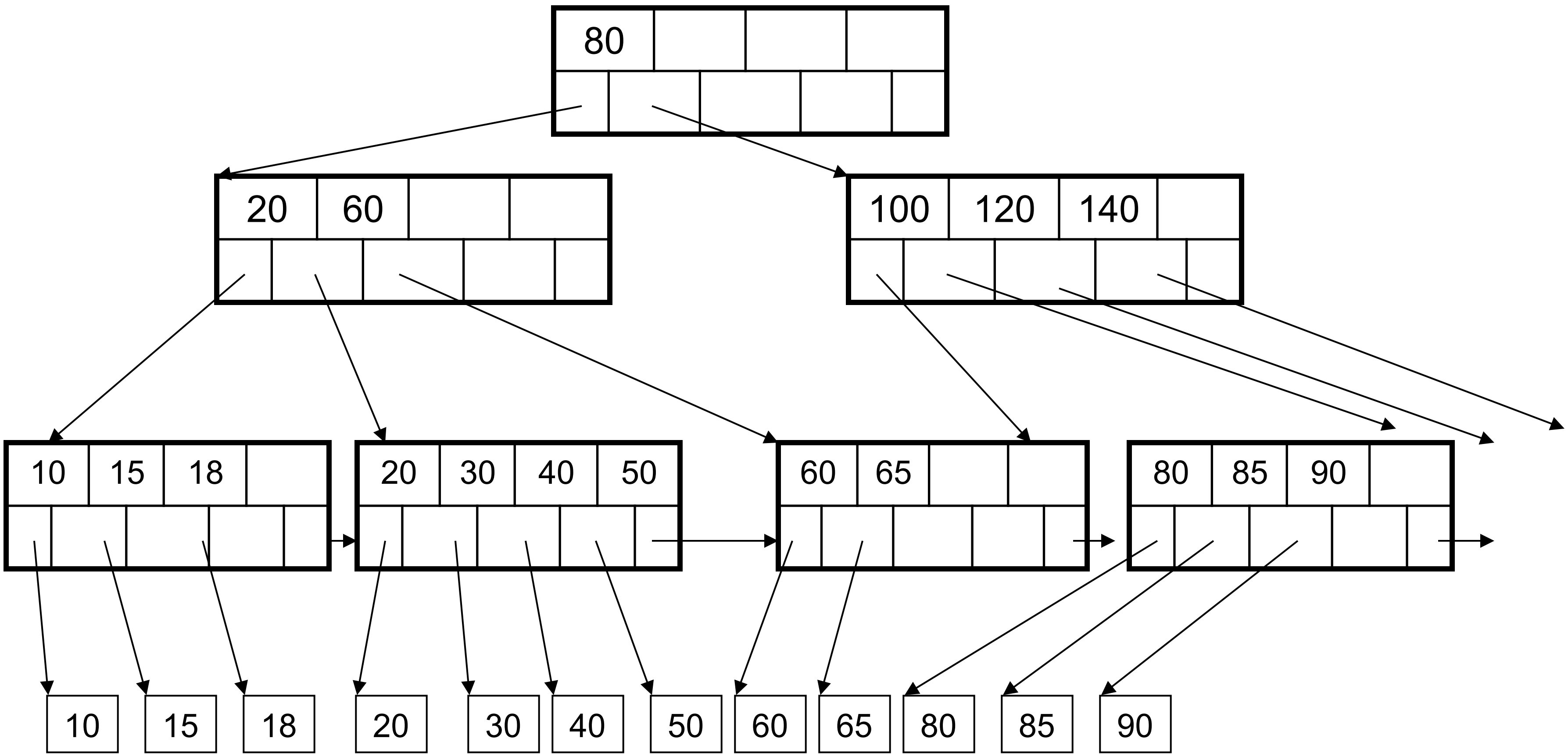
- Find leaf where K belongs, insert
- If no overflow (2d keys or less), halt
- If overflow (2d+1 keys), split node, insert in parent:



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

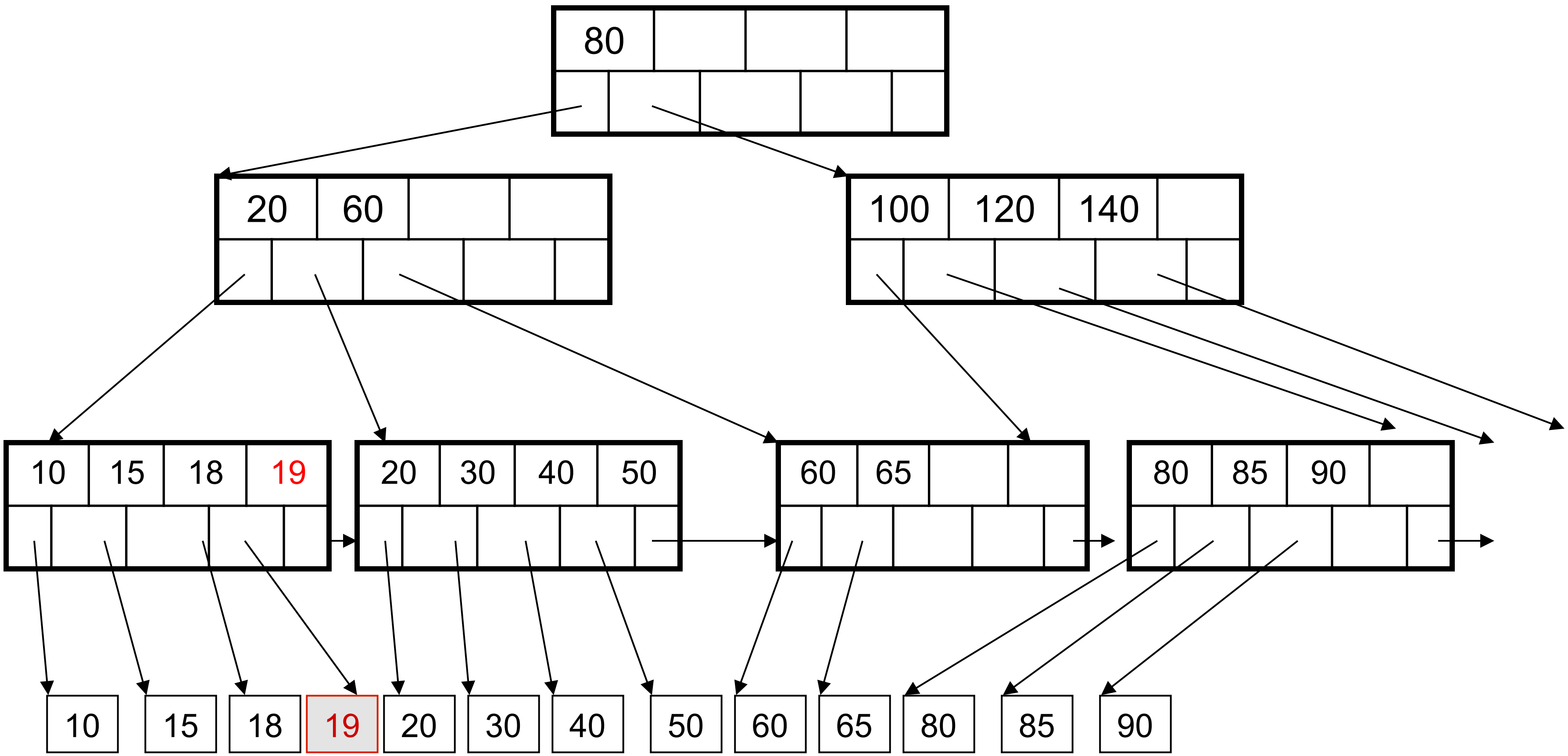
Insertion in a B+ Tree

Insert K=19



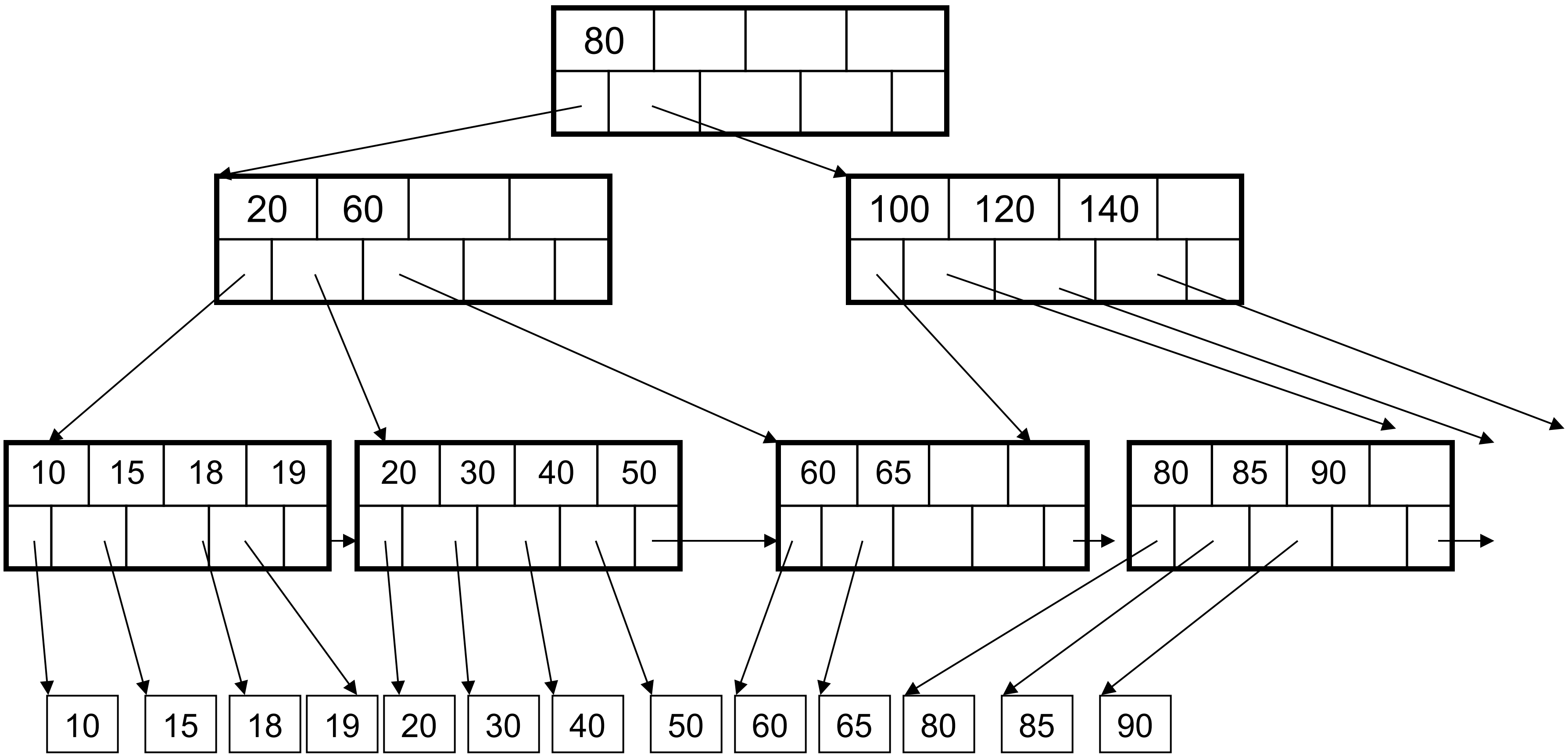
Insertion in a B+ Tree

After insertion



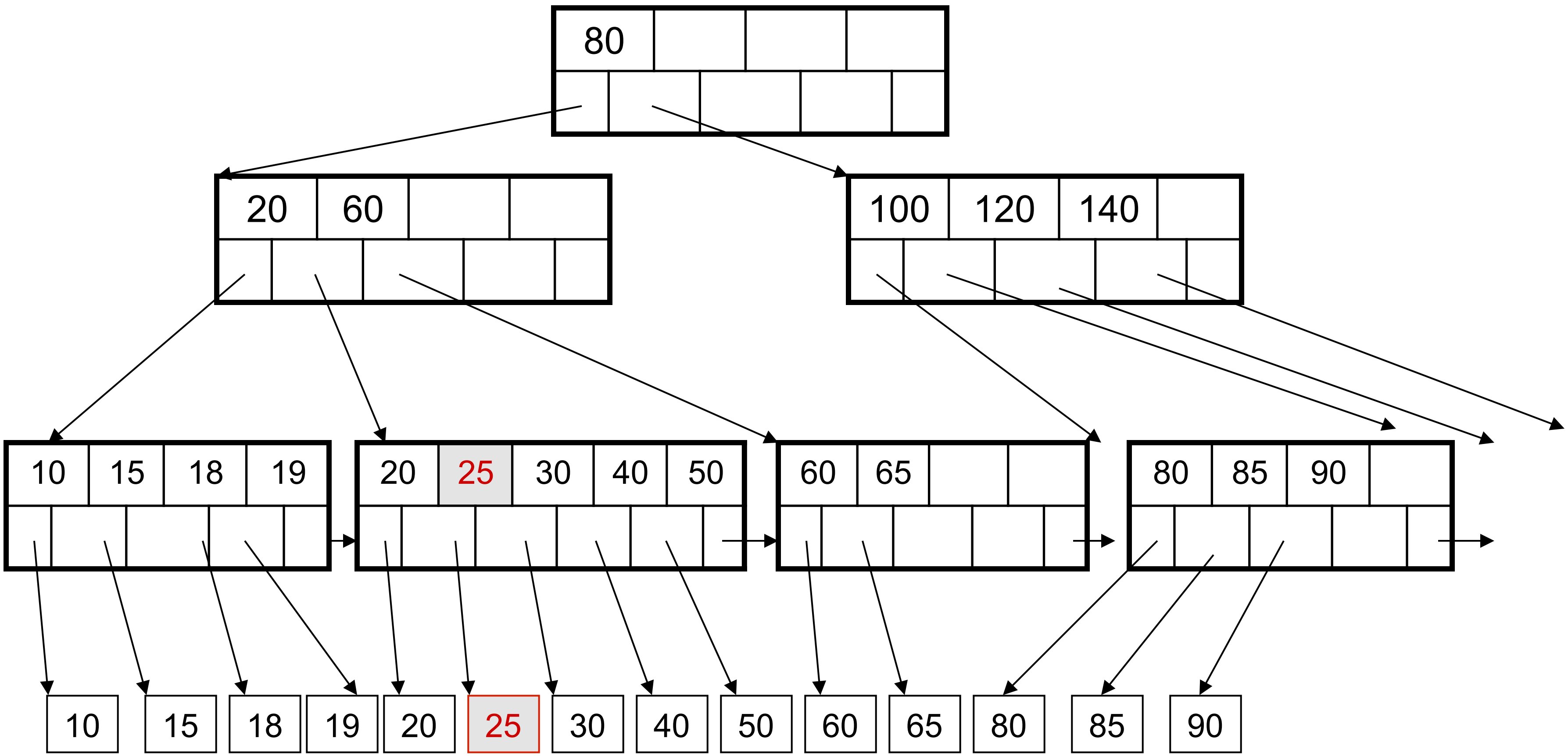
Insertion in a B+ Tree

Now insert 25



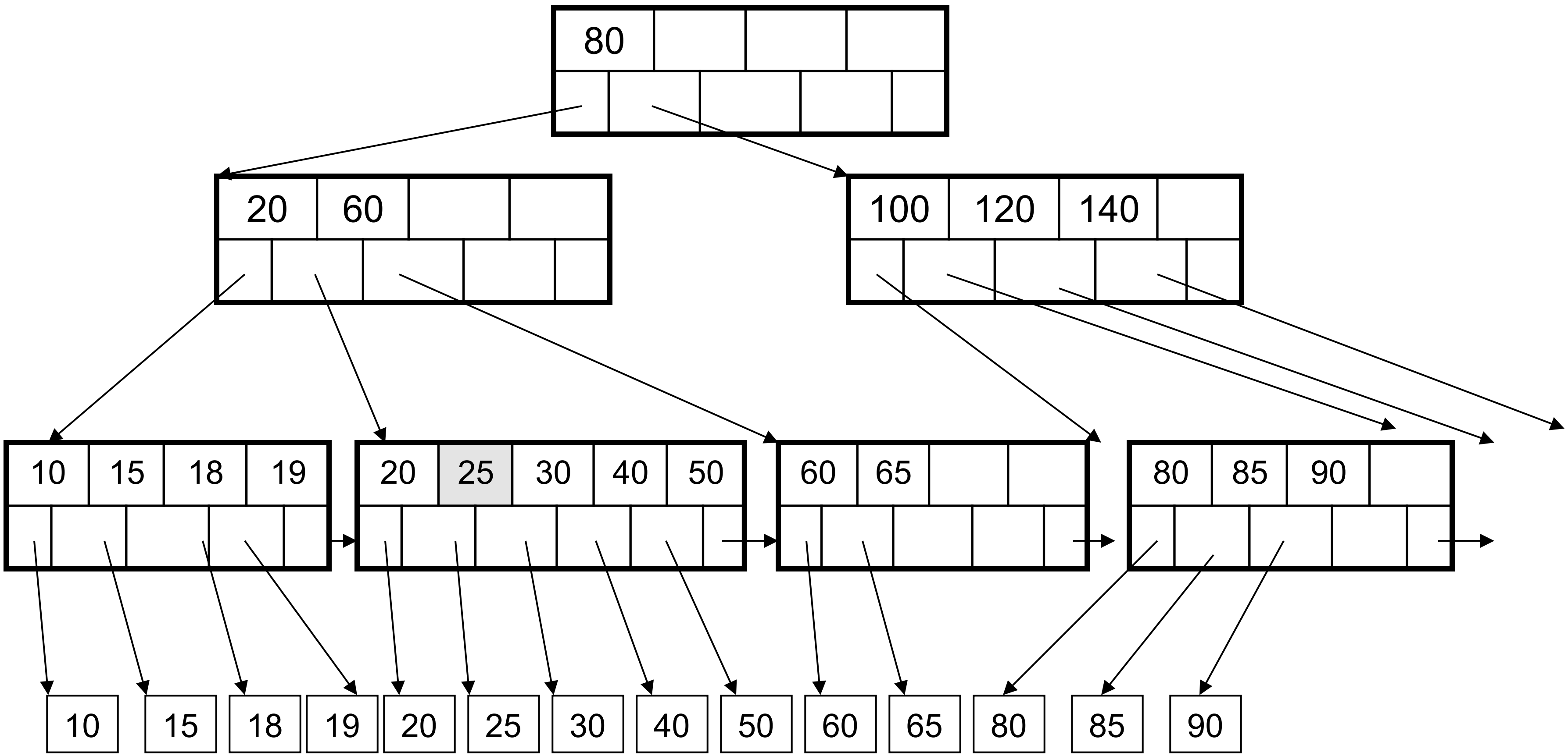
Insertion in a B+ Tree

After insertion



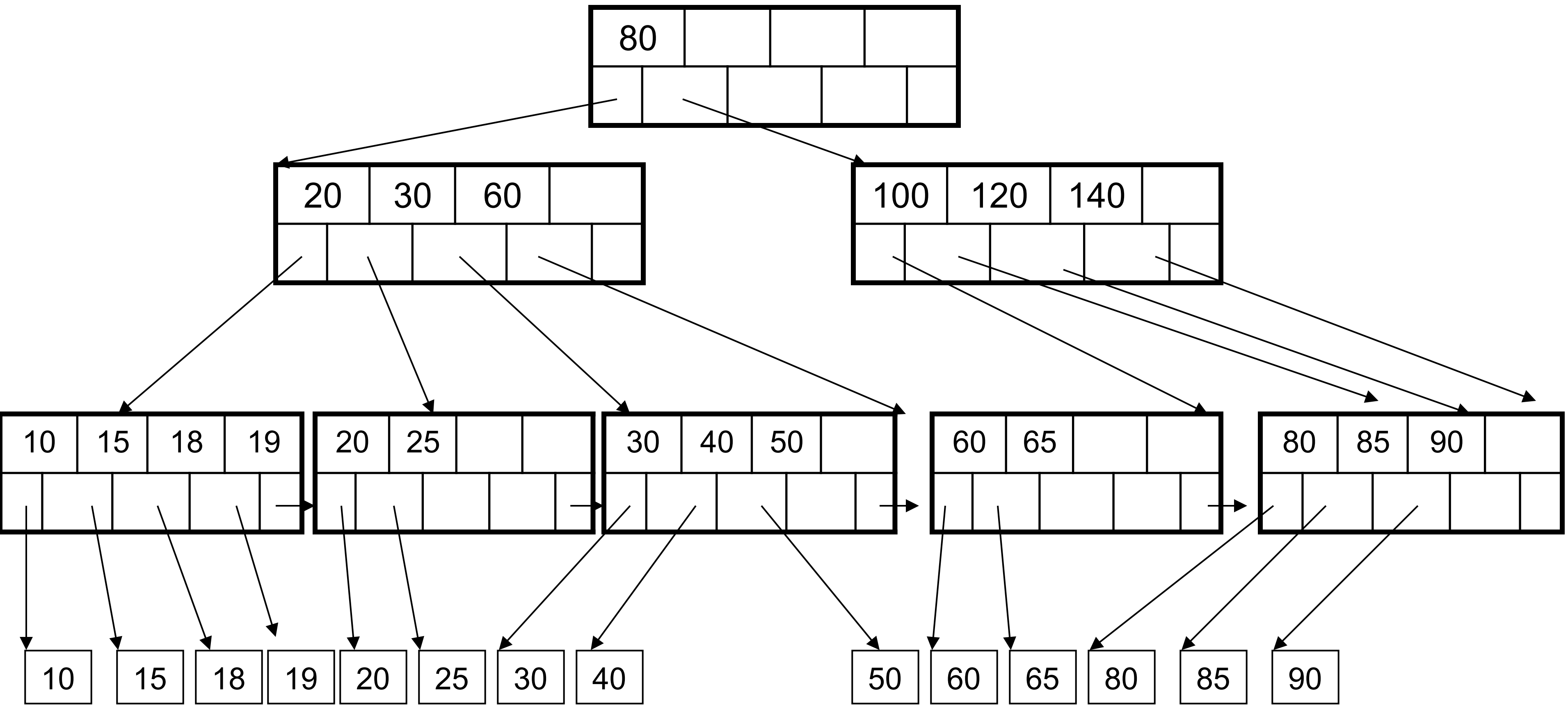
Insertion in a B+ Tree

But now have to split !



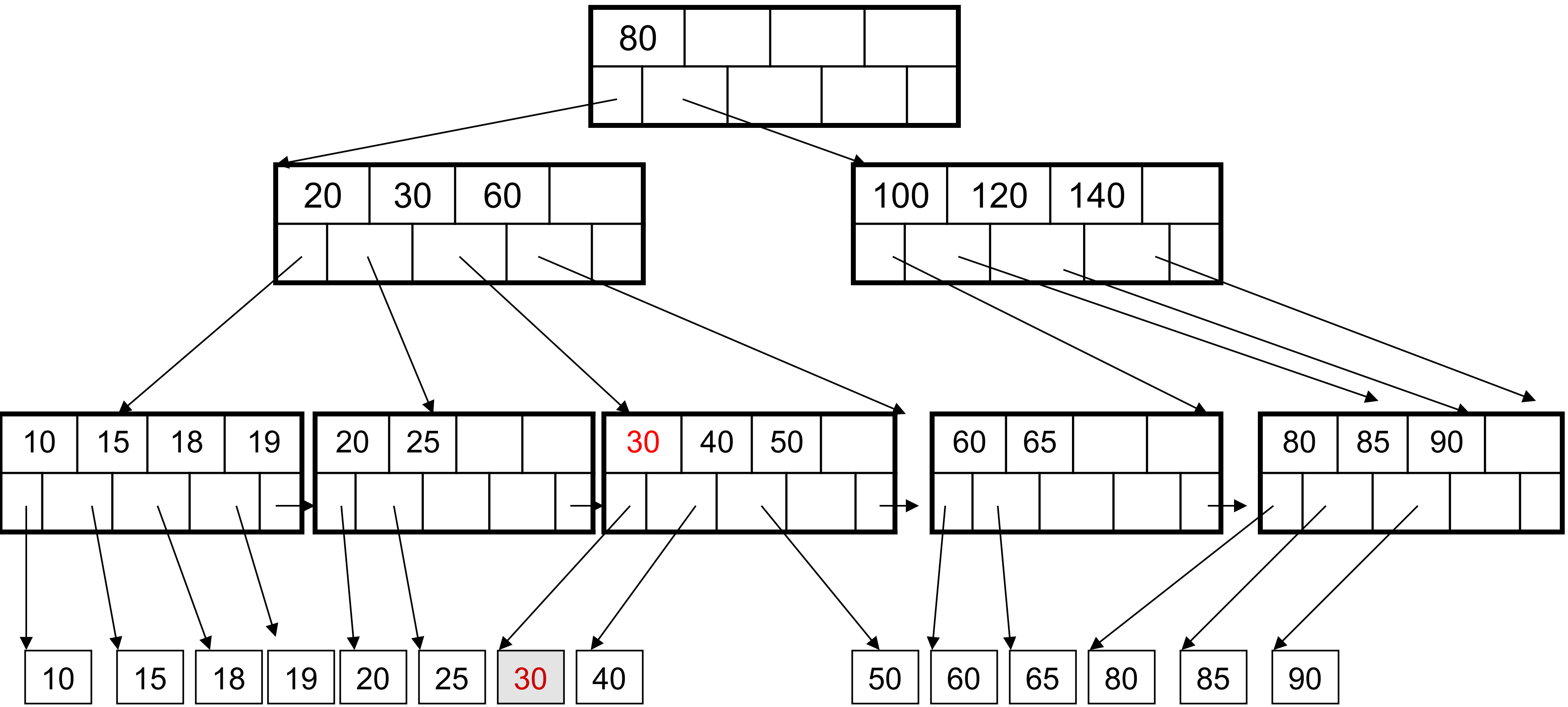
Insertion in a B+ Tree

After the split



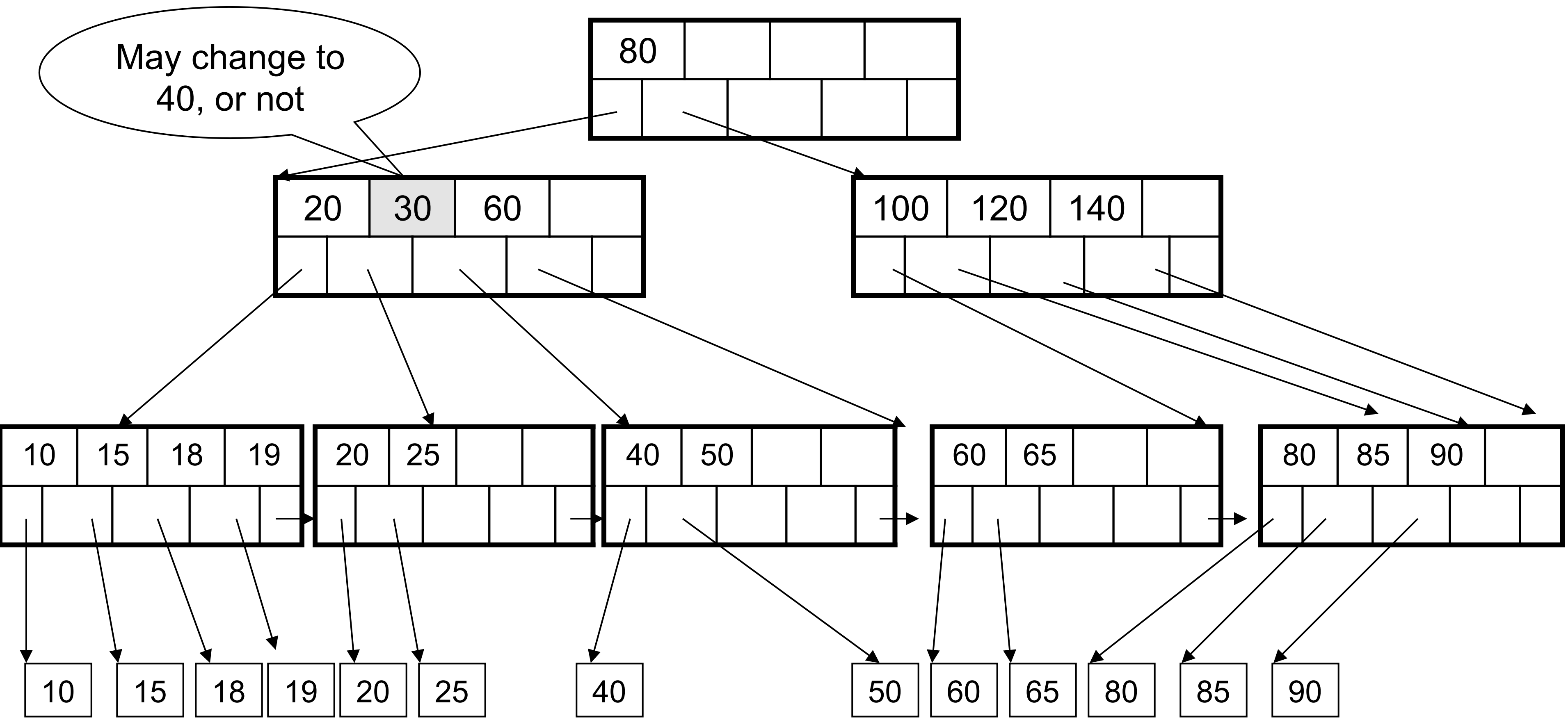
Deletion from a B+ Tree

Delete 30



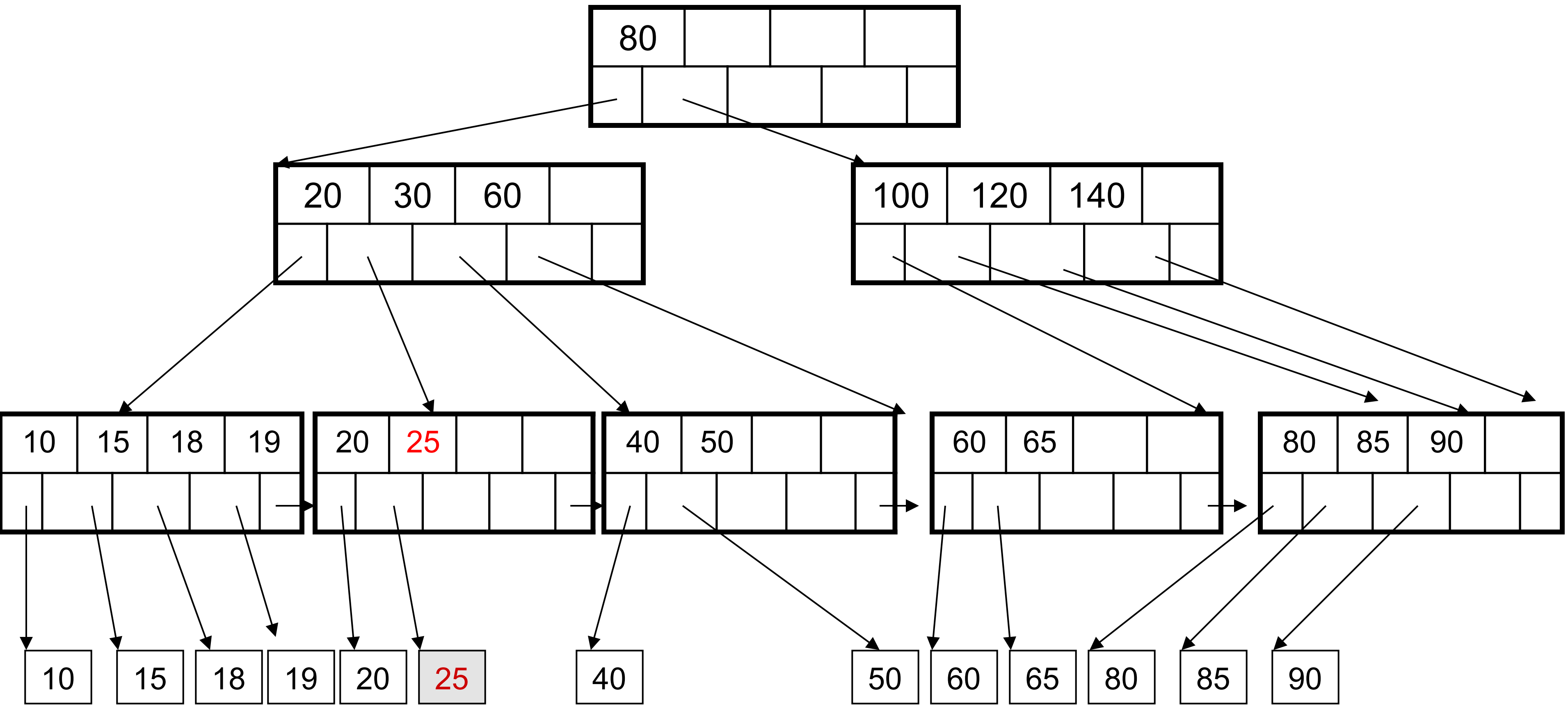
Deletion from a B+ Tree

After deleting 30



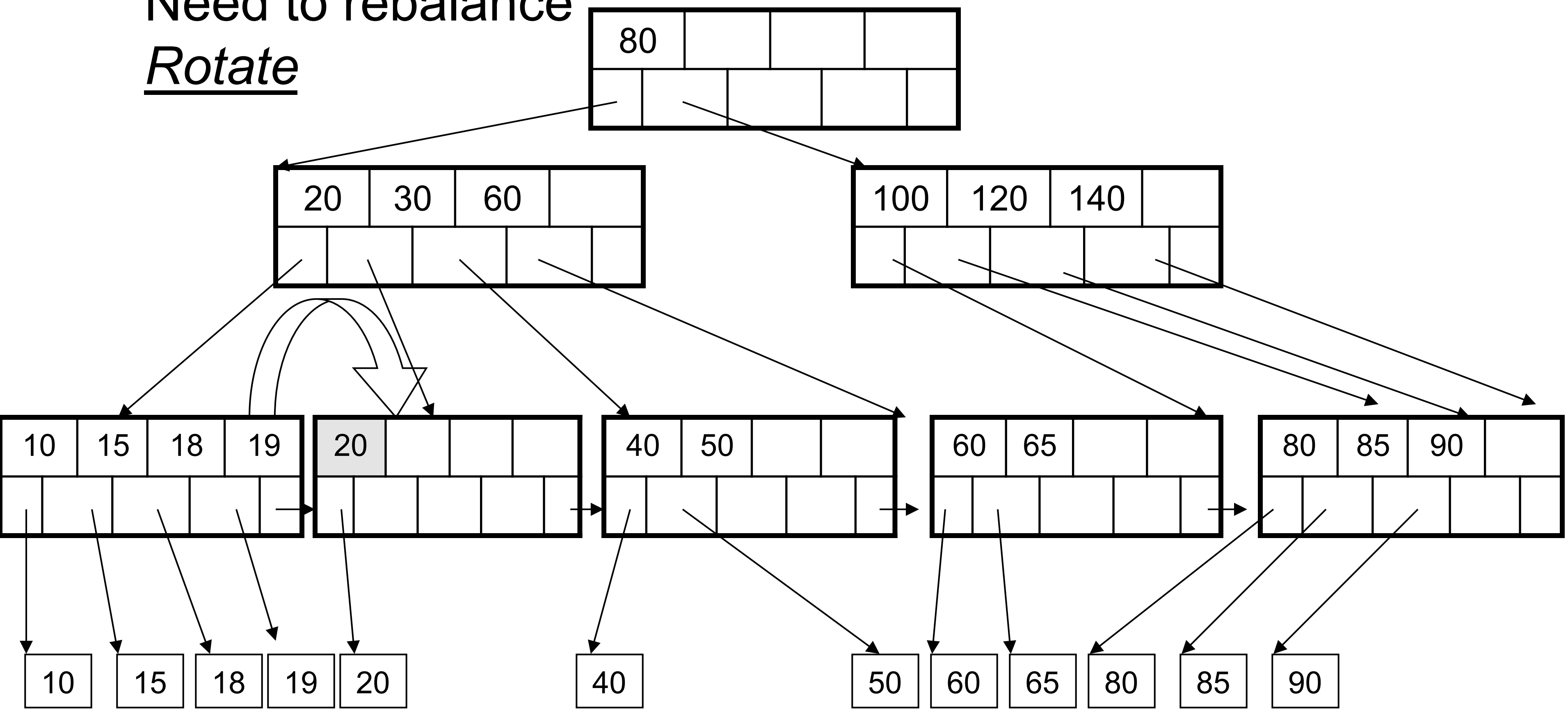
Deletion from a B+ Tree

Now delete 25



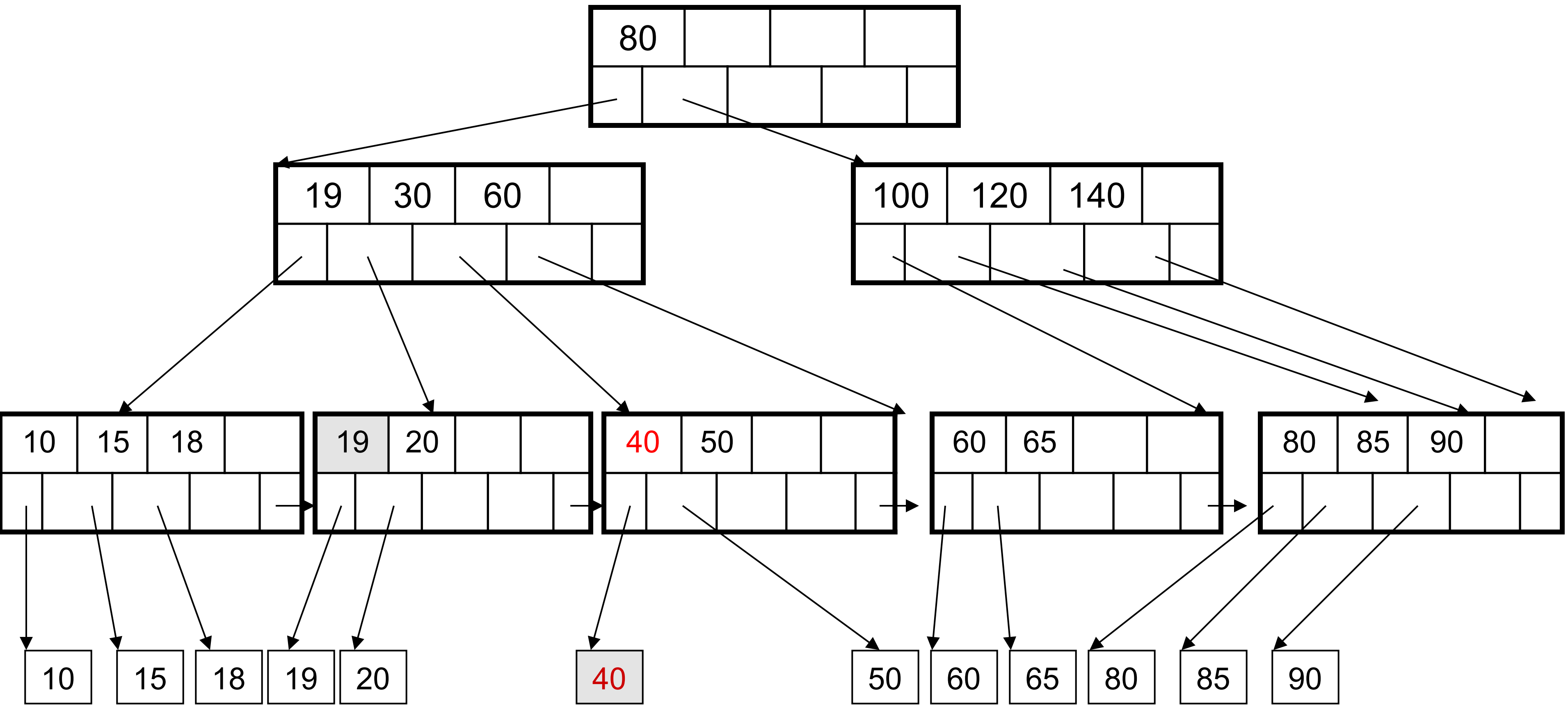
Deletion from a B+ Tree

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



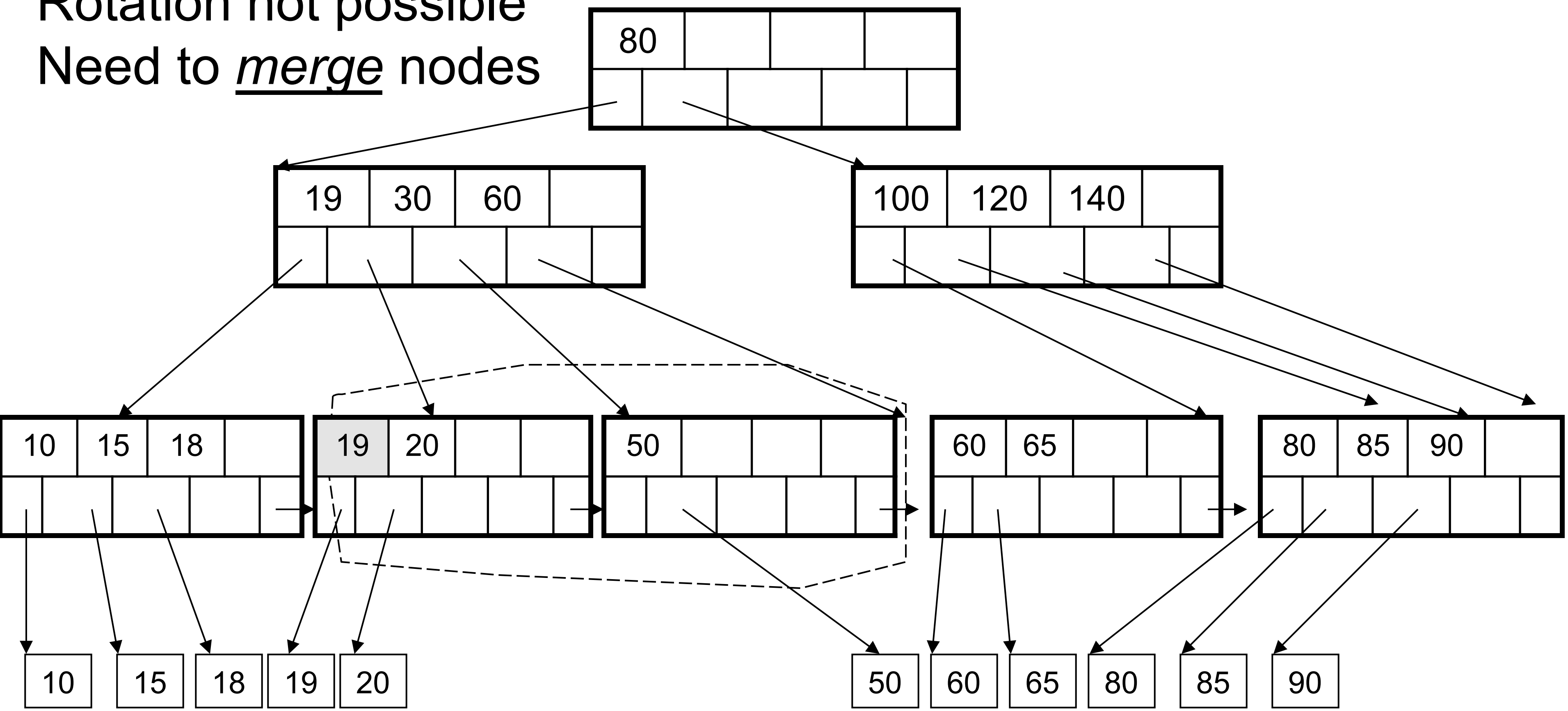
Deletion from a B+ Tree

Now delete 40



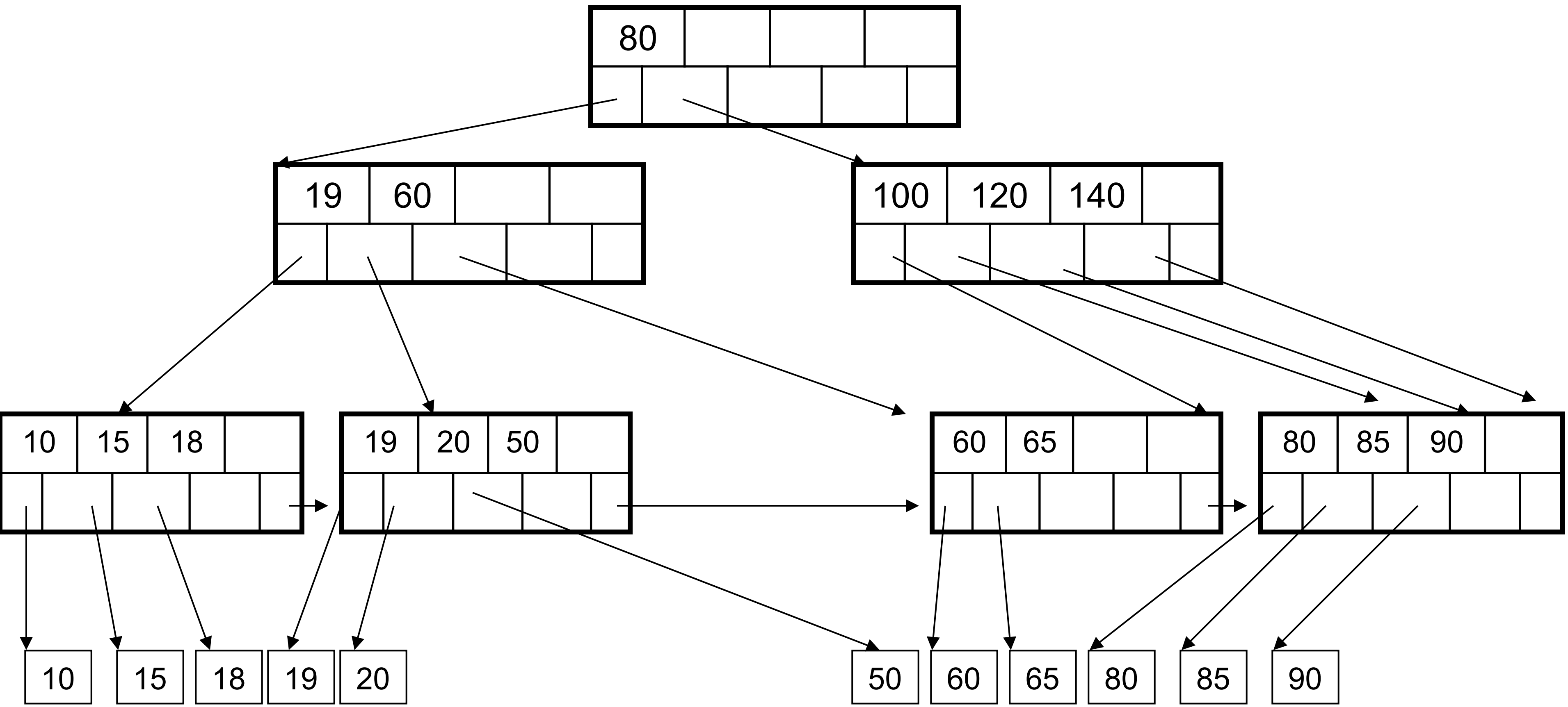
Deletion from a B+ Tree

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



Deletion from a B+ Tree

Final tree



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

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

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