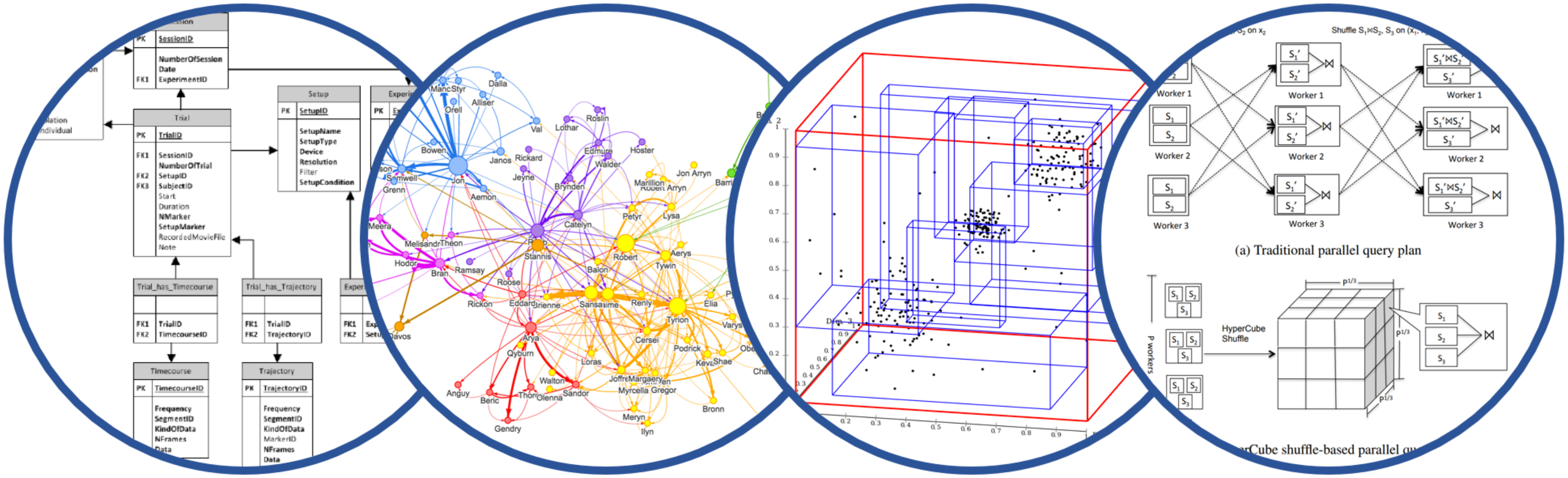


Database System Internals

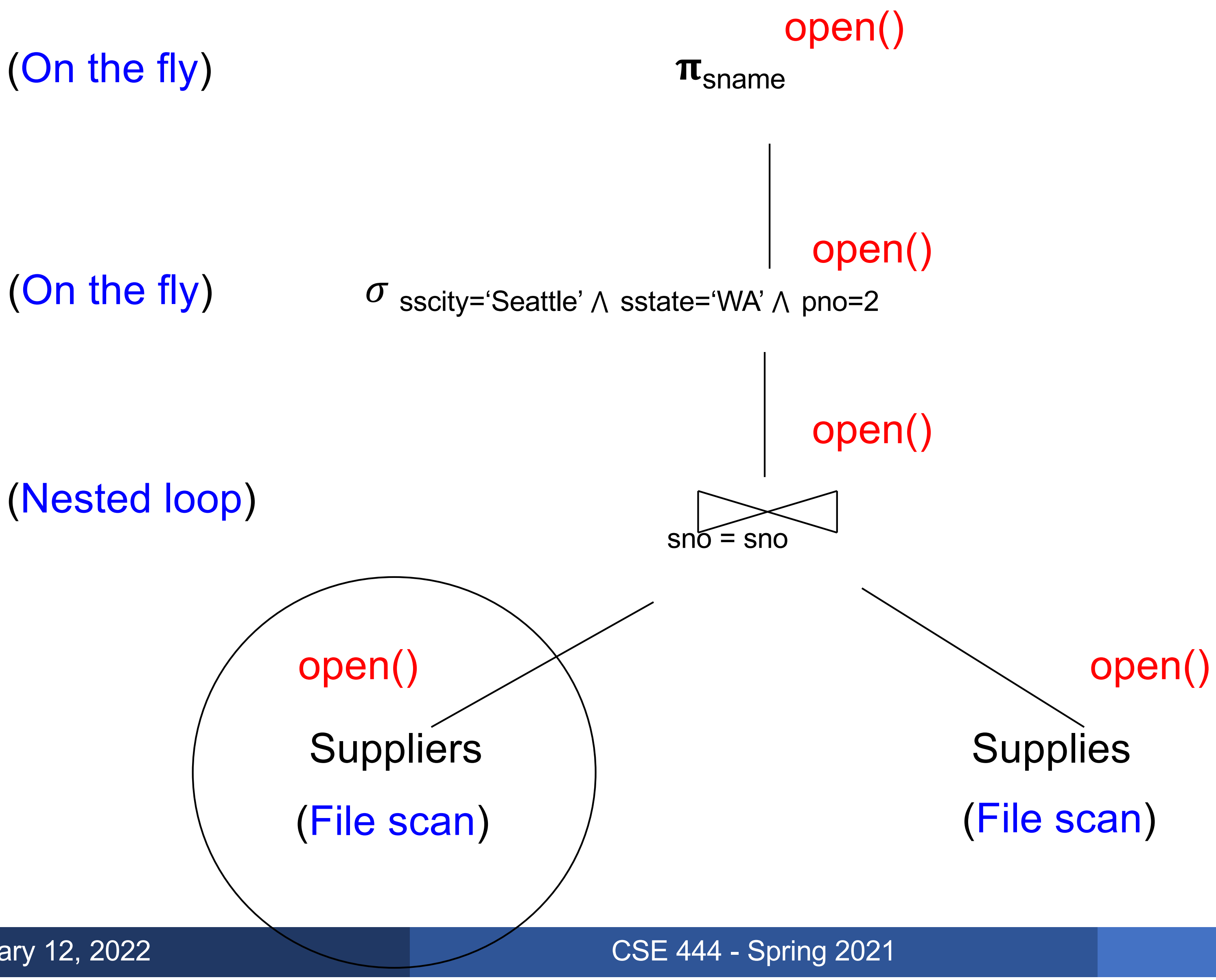
Indexing

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle

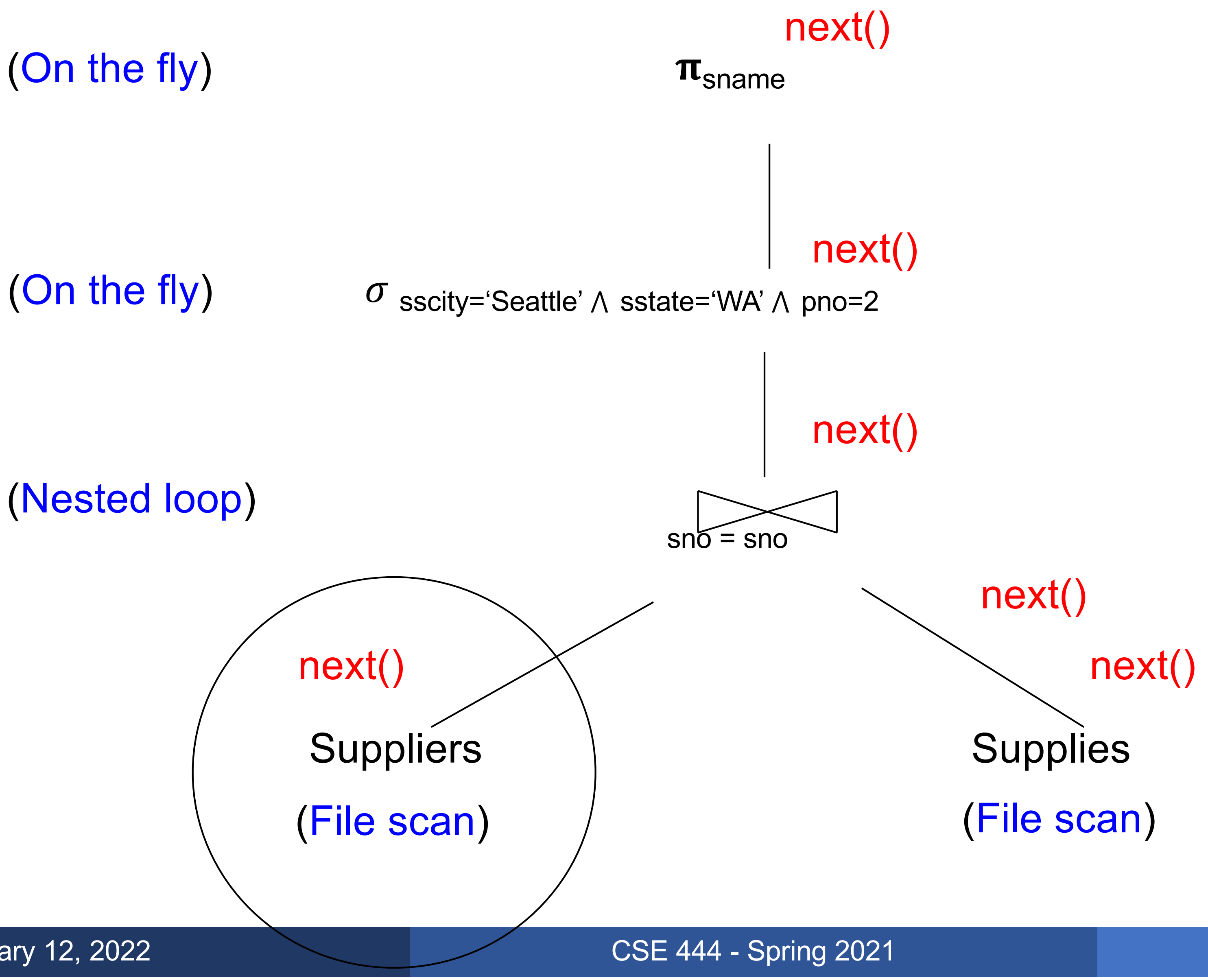
Heap File Access Method API

- **Create** or **destroy** a file
- **Insert** a record
- **Delete** a record with a given rid (rid)
  - rid: unique tuple identifier (more later)
- **Get** a record with a given rid
  - Not necessary for sequential scan operator
  - But used with indexes (more next lecture)
- **Scan** all records in the file

Query Execution How it all Fits



Query Execution How it all Fits



Query Execution In SimpleDB

open()

next()

SeqScan

Operator at bottom of plan

open()

next()

Heap File Access Method

In SimpleDB, SeqScan can find HeapFile in Catalog

Offers iterator interface
open()
next()
close()
Knows how to read/write pages from disk

But if Heap File reads data directly from disk, it will not stay cached in Buffer Pool!

Iterators in SimpleDB

- SeqScan.java

- DbFileIterator.java

- Both have this method:
  public Tuple next()

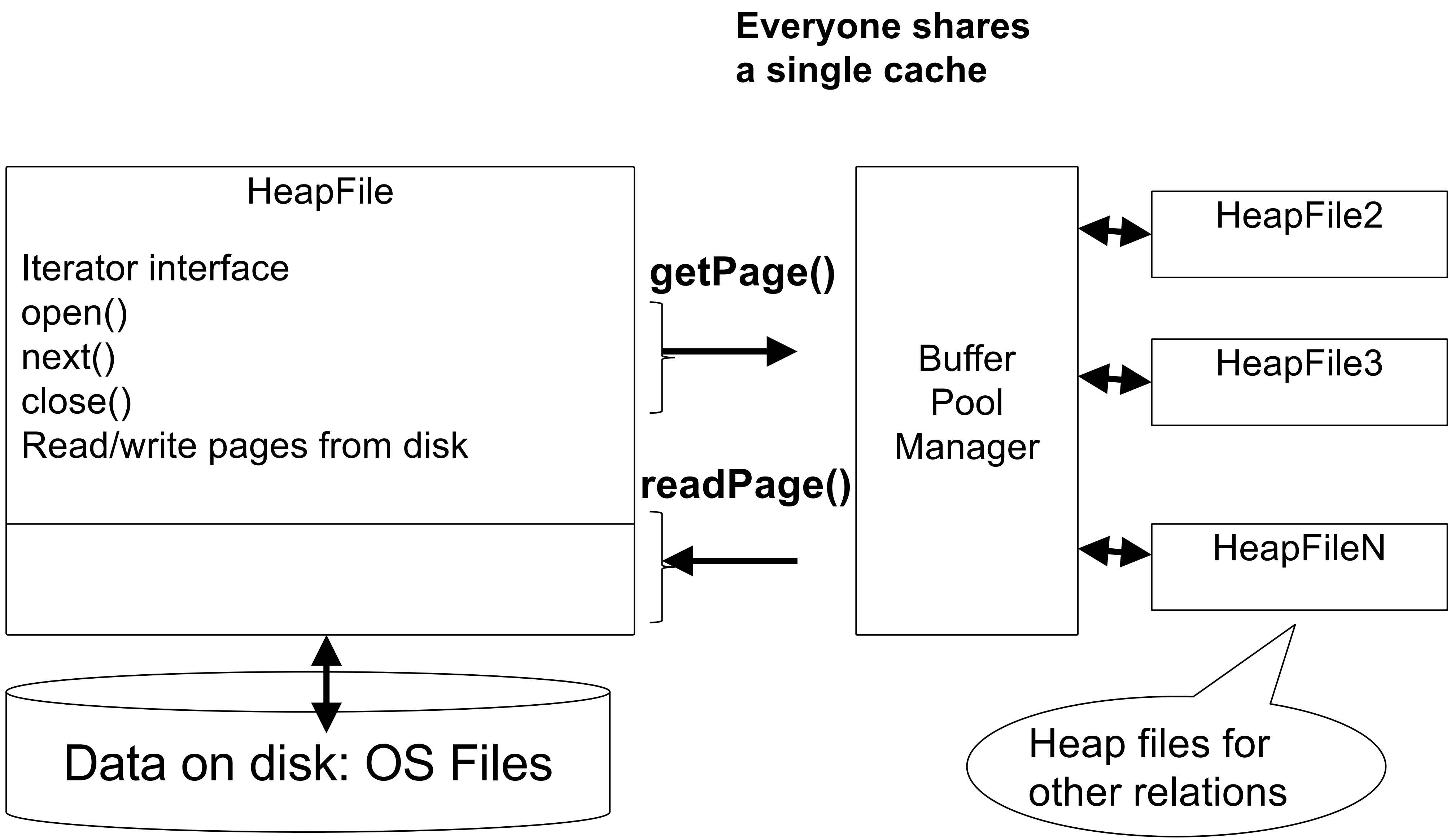
Iterators in SimpleDB

- How does DbFileIterator.java get its tuples?

- Needs pages from buffer pool

- Buffer pool has this method:
  getPage()

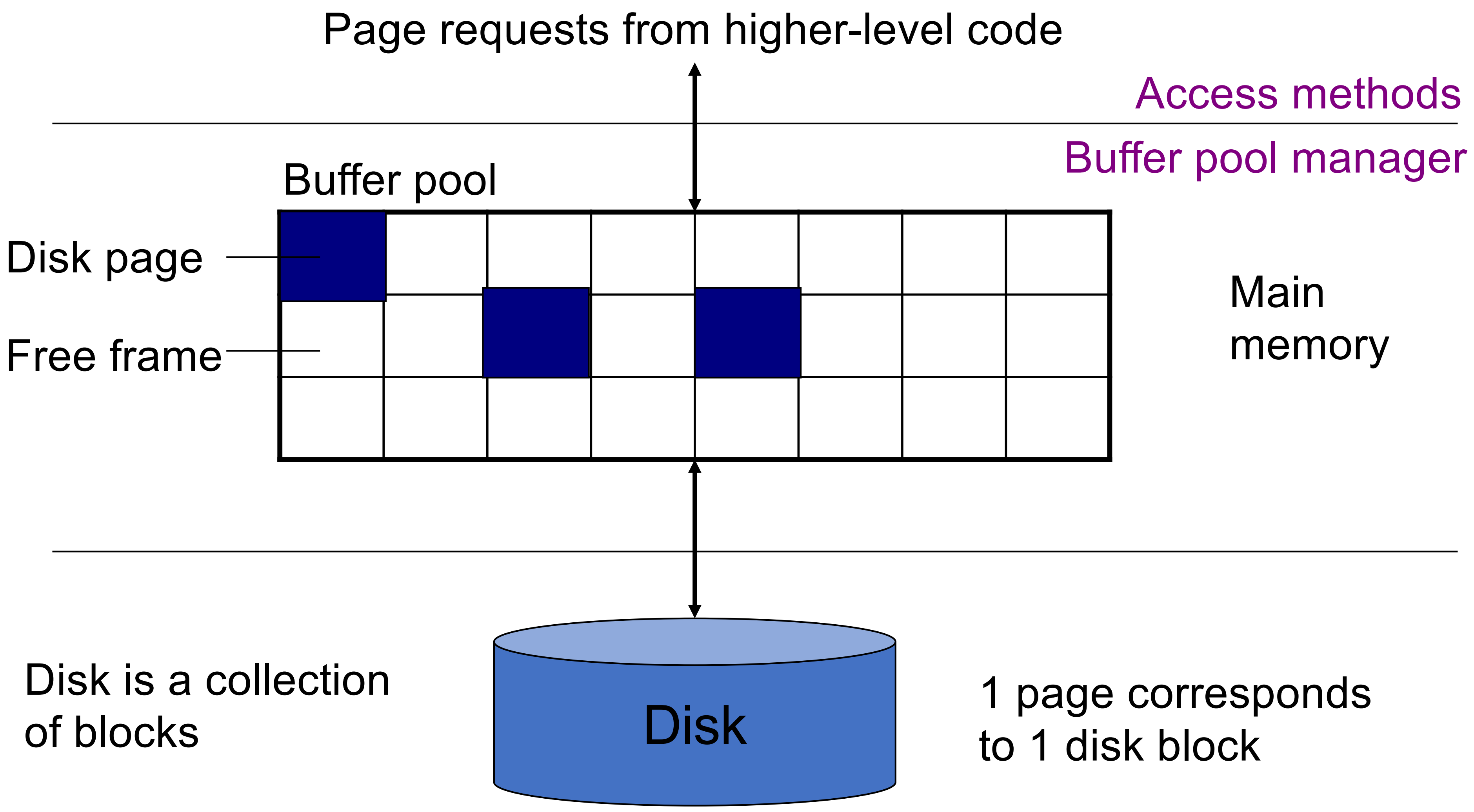
Query Execution In SimpleDB



Buffer Manager

- Brings pages in from memory and caches them
- Eviction policies
  - Random page (ok for SimpleDB)
  - Least-recently used
  - The “clock” algorithm
- Keeps track of which **pages are dirty**
  - A dirty page has changes not reflected on disk
  - Implementation: Each page includes a dirty bit

Buffer Manager



Basic Access Method: Heap File

API

- **Create** or **destroy** a file
- **Insert** a record
- **Delete** a record with a given rid (rid)
  - rid: unique tuple identifier (more later)
- **Get** a record with a given rid
  - Not necessary for sequential scan operator
  - But used with indexes
- **Scan** all records in the file

But Often Also Want….

- **Scan** all records in the file that match a **predicate** of the form **attribute op value**
  - Example: Find all students with GPA > 3.5

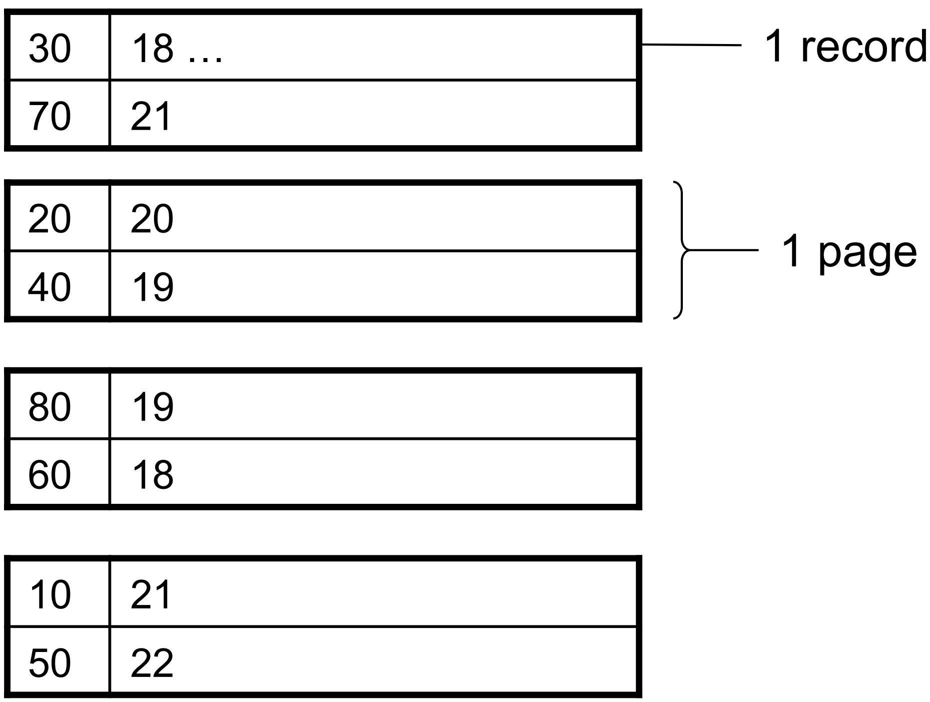
- Critical to support such requests efficiently
  - Why read all data form disk when we only need a small fraction of that data?

- This lecture and next, we will learn how

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?

- Note: Sorted files are inefficient for inserts/deletes

Creating Indexes in SQL

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

```
CREATE INDEX V1 ON V(N)
```

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

```
select *
from V
where P=55 and M=77
```

```
select *
from V
where P=55
```

Outline

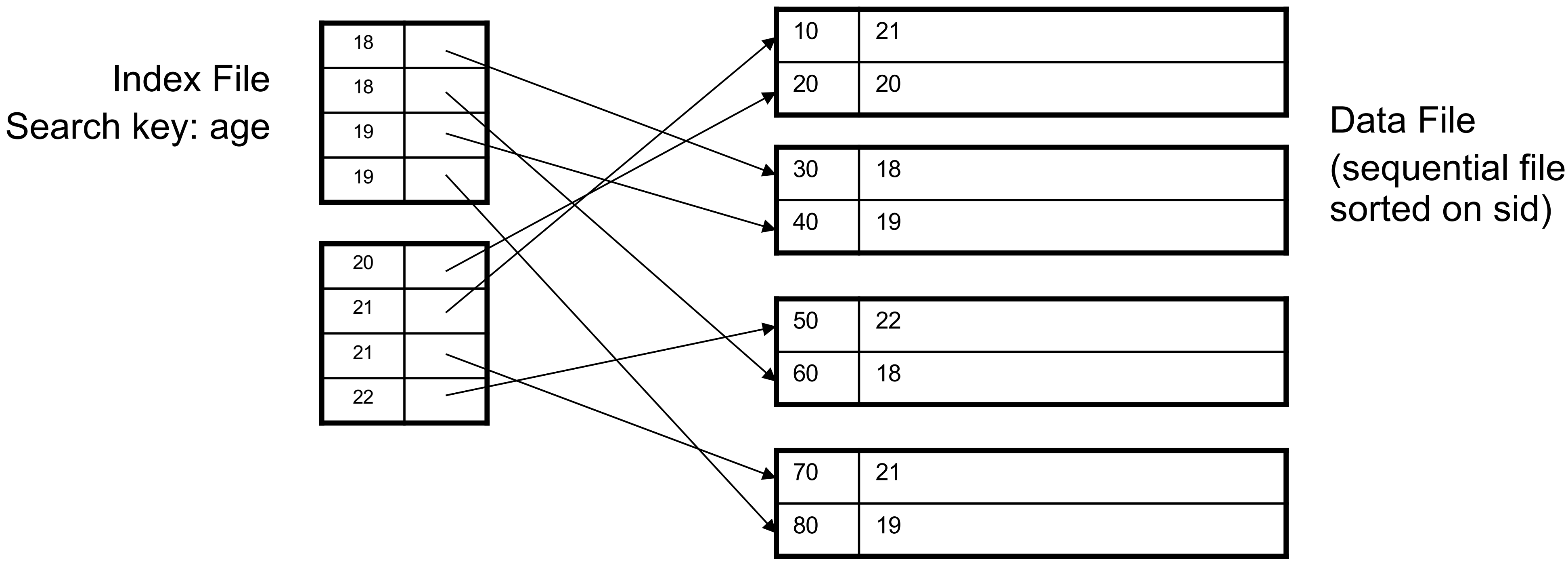
- Index structures
- Hash-based indexes
- B+ trees

Today (Index structures, Hash-based indexes)

Next time (B+ trees)

Indexes

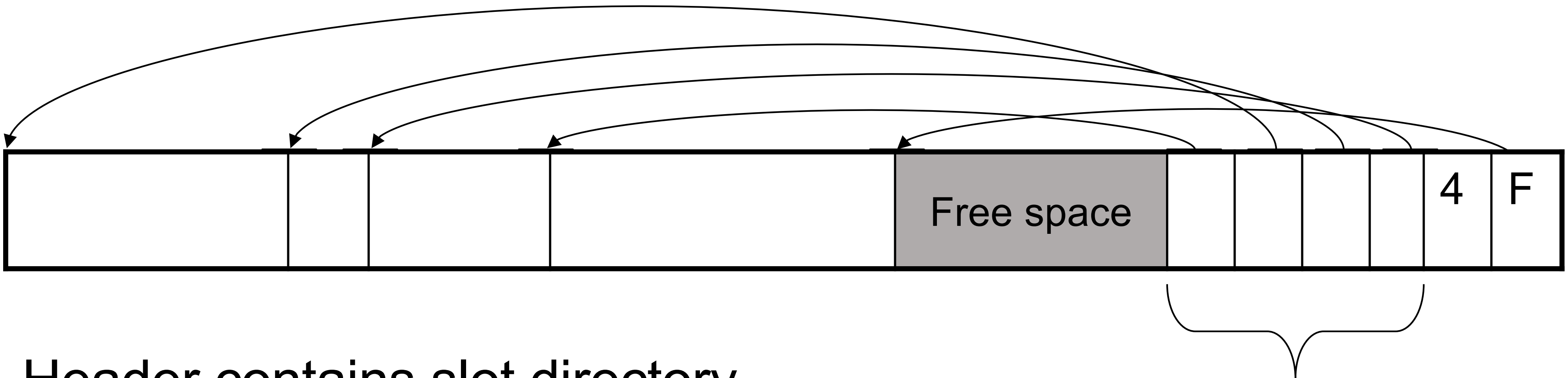
- **Index**: data structure that organizes data records on disk to optimize selections on the ***search key fields*** for the index
- An index contains a collection of ***data entries***, and supports efficient retrieval of all data entries with a given search key value **k**
- Indexes are also access methods!
  - So they provide the same API as we have seen for Heap Files
  - And efficiently support scans over tuples matching predicate on search key



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:
  - (k, RID)
  - (k, list-of-RIDs)
  - The actual record with key k
    - In this case, **the index is also a special file organization**
    - Called: “indexed file organization”

Page Format Approach 2



Header contains slot directory

+ Need to keep track of # of slots
+ Also need to keep track of free space (F)

Slot directory

Each slot contains
<record offset, record length>

Can handle variable-length records

Can move tuples inside a page without changing RIDs

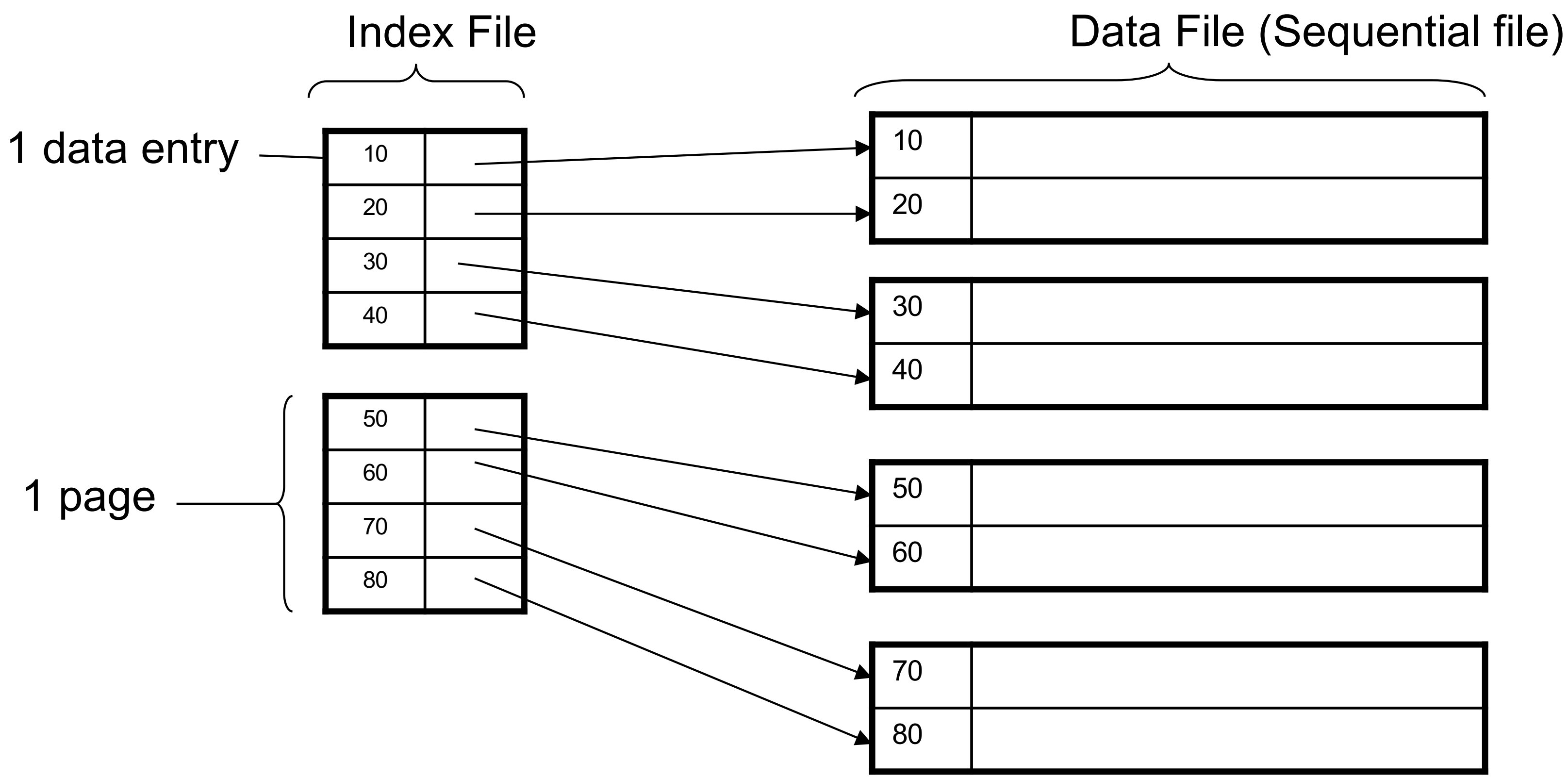
RID is (PageID, SlotID) combination

Different Types of Files

- For the data inside base relations:
  - Heap file (tuples stored without any order)
  - Sequential file (tuples sorted on some attribute(s))
  - Indexed file (tuples organized following an index)
- Then we can have additional index files that store (key,rid) pairs
- Index can also be a "covering index"
  - Index contains (search key + other attributes, rid)
  - Index suffices to answer some queries

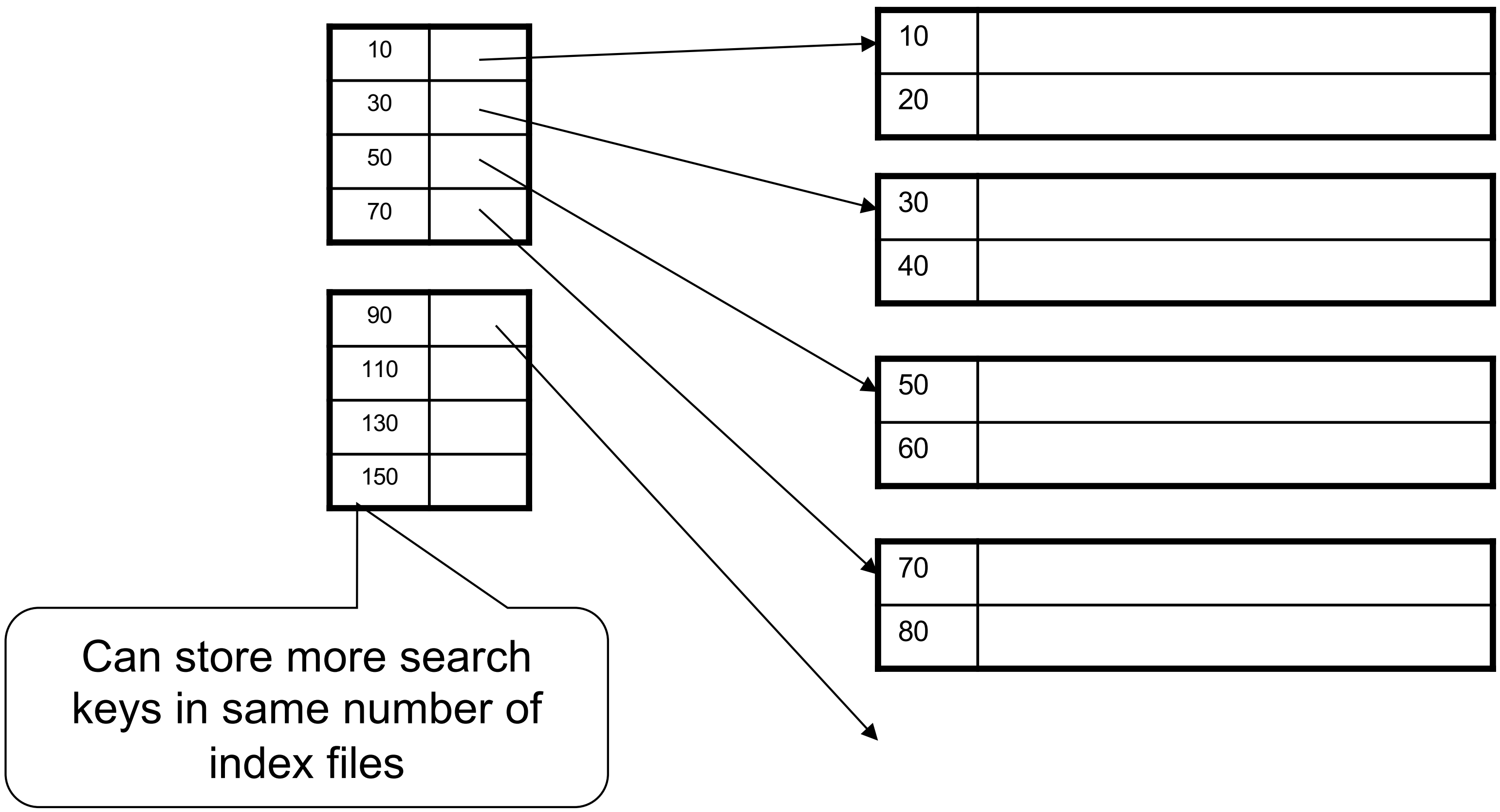
Primary Index

- Primary index determines location of indexed records
- *Dense* index: sequence of (key,rid) pairs



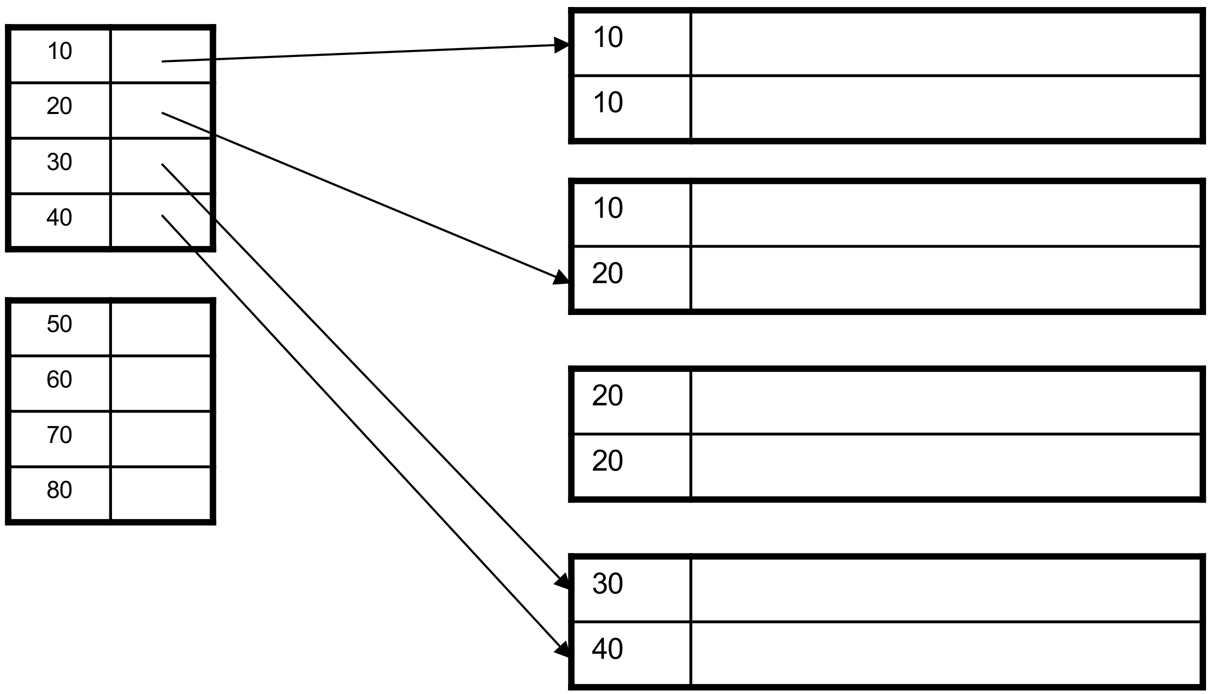
Primary Index

- *Sparse* index



Primary Index with Duplicate Keys

- Dense index:

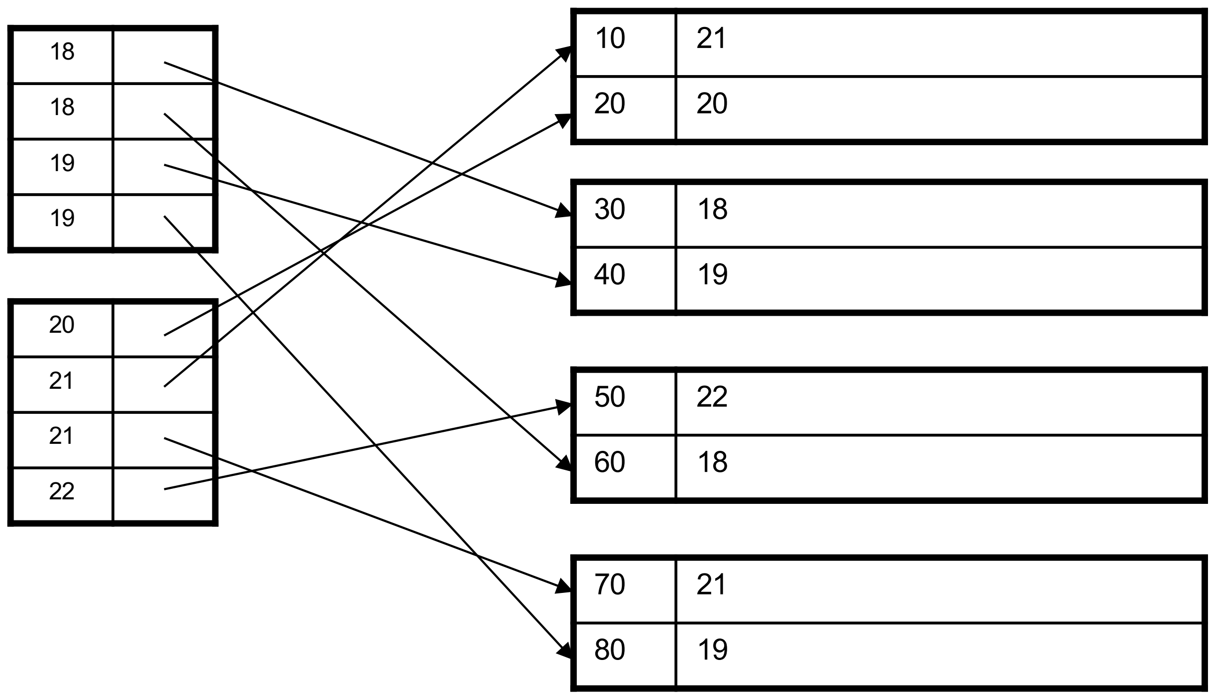


Primary Index: Back to Example

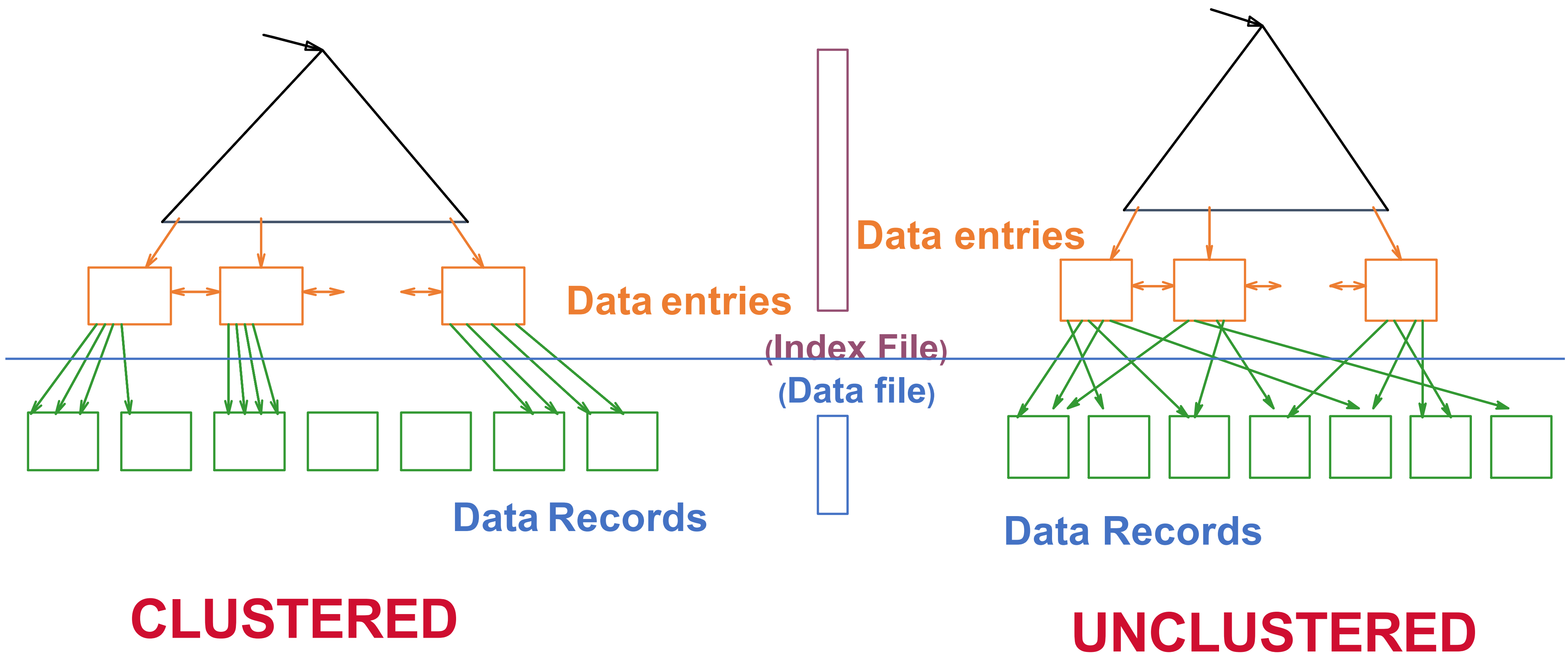
- Let's assume all pages of index fit in memory
- Find student whose sid is 80
  - Index (dense or sparse) points directly to the page
  - Only need to read 1 page from disk.
- Find all students older than 20
- How can we make *both* queries fast?

Secondary Indexes

- Do not determine placement of records in data files
- 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
  Possible that sorted order of the secondary index matches that of primary index, but hardly every the case

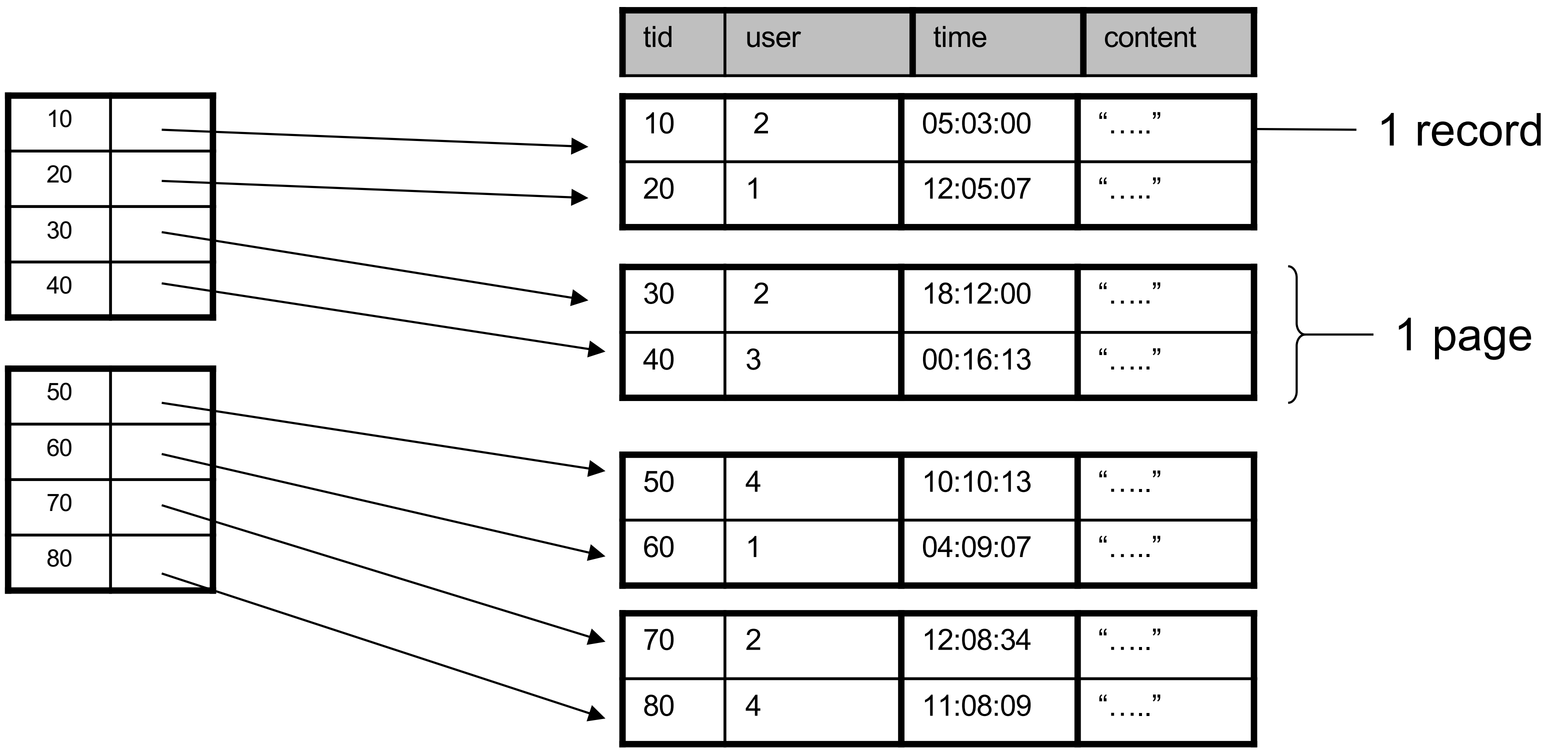
Secondary Indexes

- Applications
  - Index unsorted files (heap files)
  - When necessary to have multiple indexes
  - Index files that hold data from two relations

Index Classification Summary

- 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 / …

Ex1. Primary Dense Index (tid)



- Dense: an "index key" for every database record
  - (In this case) every "database key" appears as an "index key"
- Primary: determines the location of indexed records
- Also, Clustered: records close in index are close in data

Improve from Primary Clustered Index?

Clustered Index can be made Sparse

(normally one key per page)

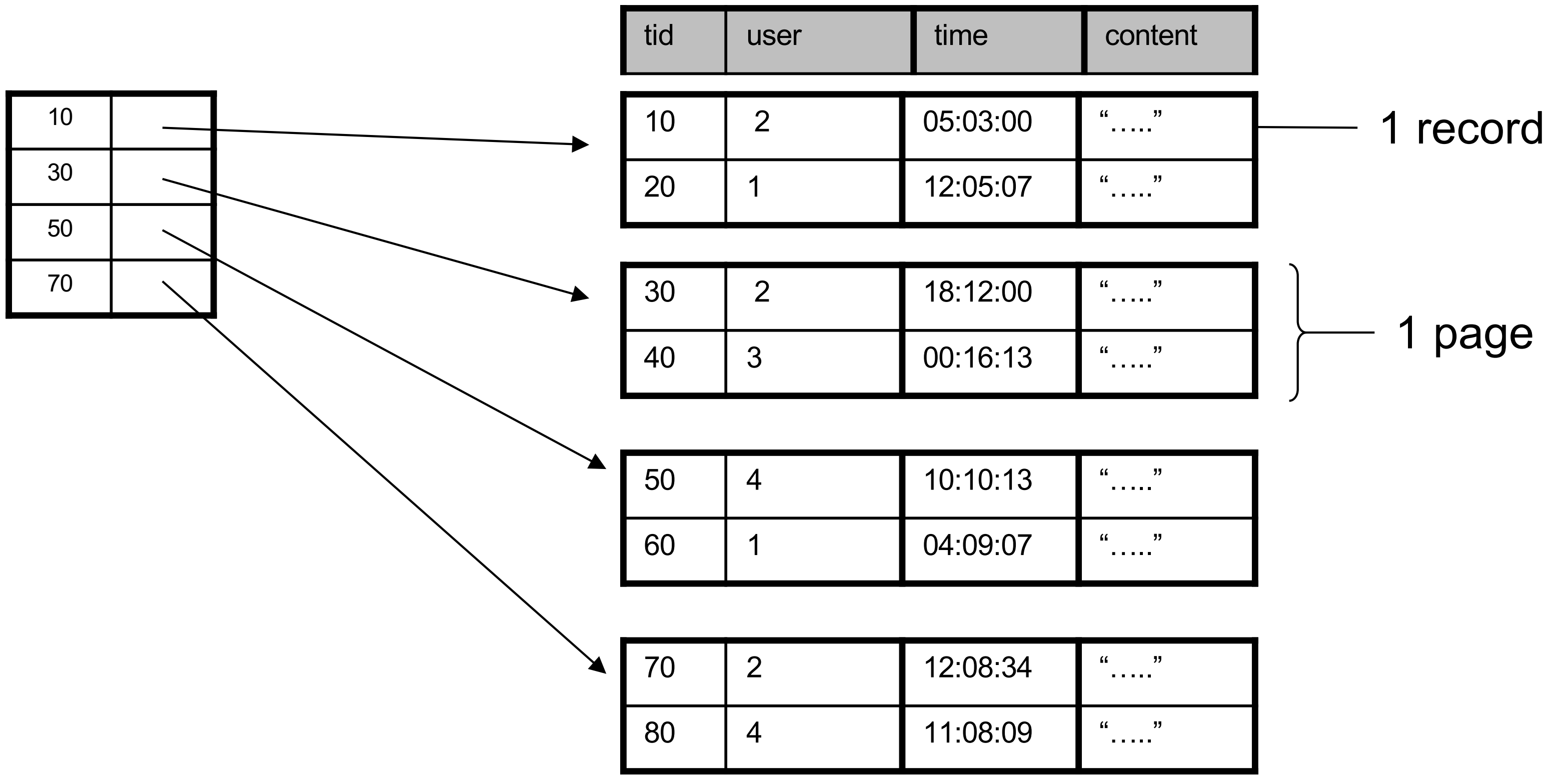
Ex2. Draw a primary sparse index on "tid"

tid	user	time	content
10	2	05:03:00	"….."
20	1	12:05:07	"….."
30	2	18:12:00	"….."
40	3	00:16:13	"….."
50	4	10:10:13	"….."
60	1	04:09:07	"….."
70	2	12:08:34	"….."
80	4	11:08:09	"….."

1 record

1 page

Ex2. Primary Sparse Index (tid)



- Only one index file page instead of two

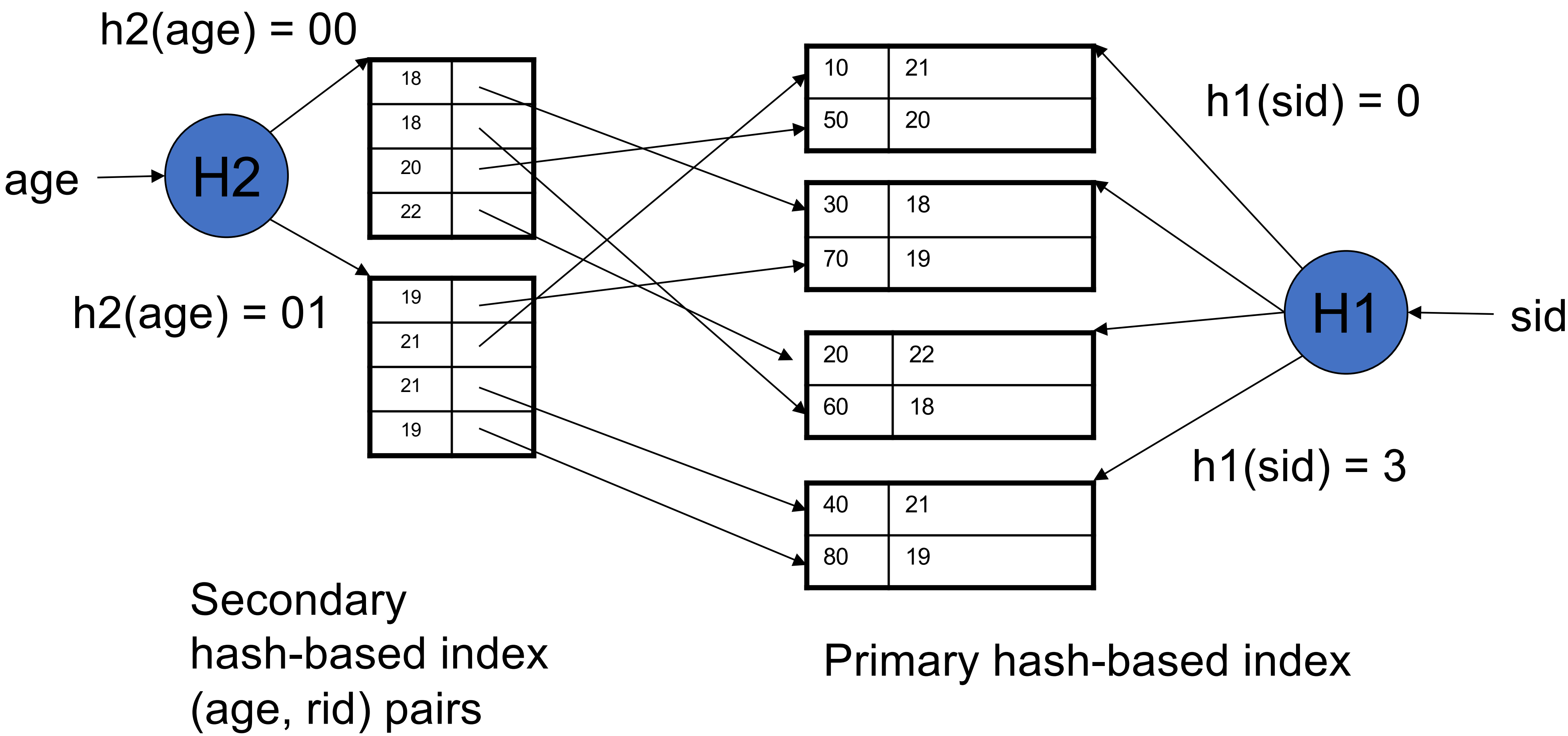
Large Indexes

- 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



Tree-Based Index

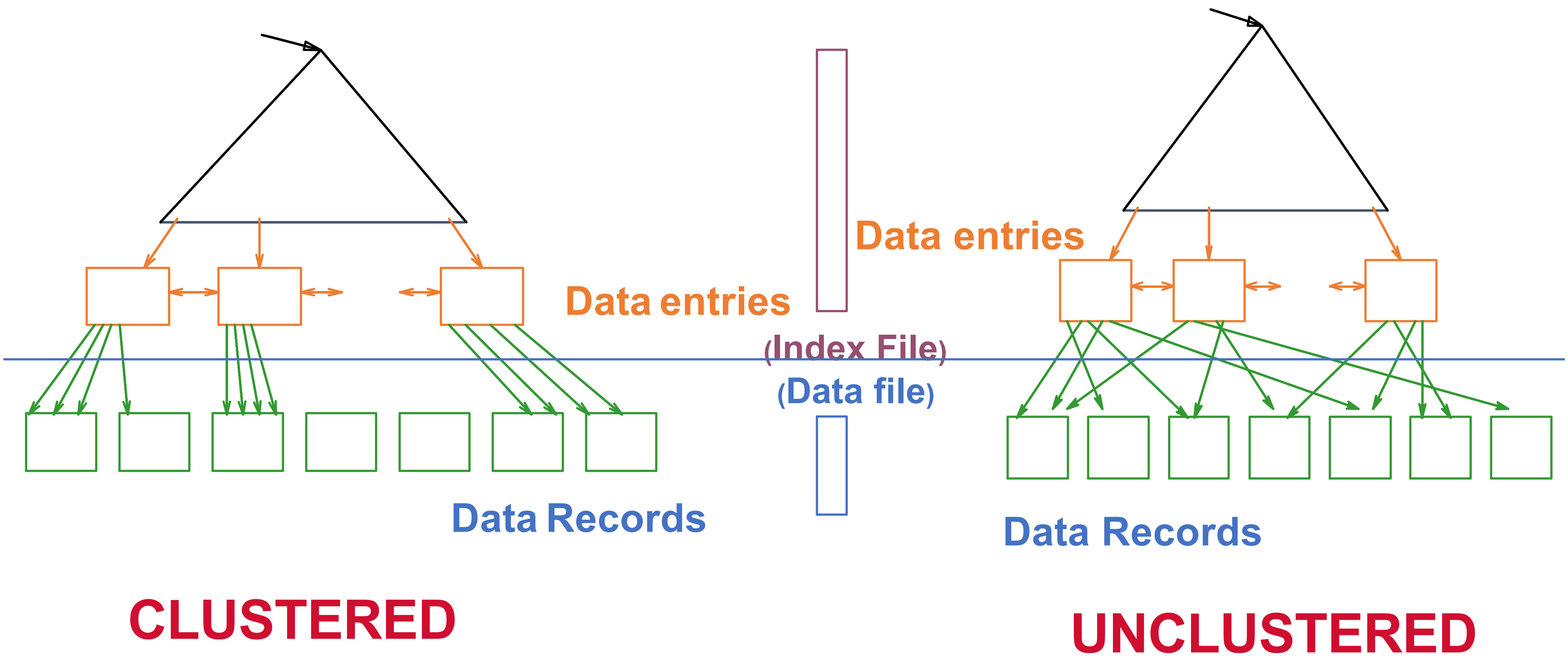
- How many index levels do we need?
- Can we create them automatically? Yes!
- Can do something even more powerful!

B+ Trees

- Search trees

- Idea in B Trees
  - Make 1 node = 1 page (= 1 block)

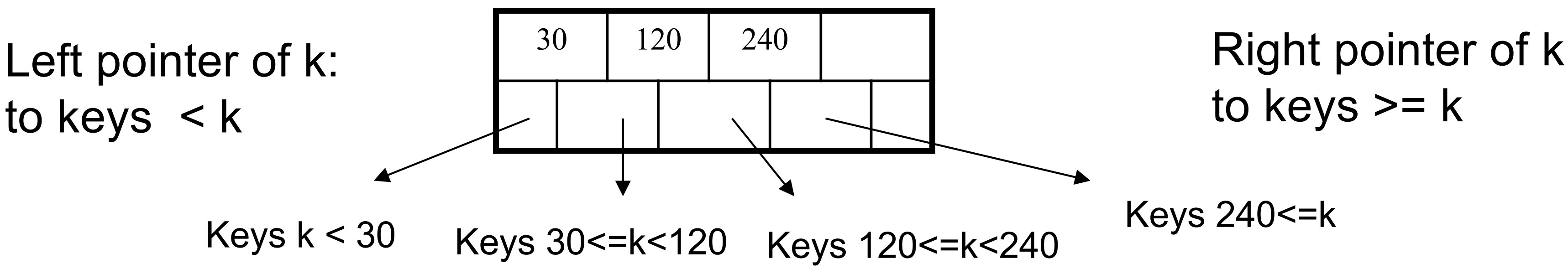
- Idea in B+ Trees
  - Keep tree balanced in height – dynamic rather than static
  - Make leaves into a linked list : facilitates range queries



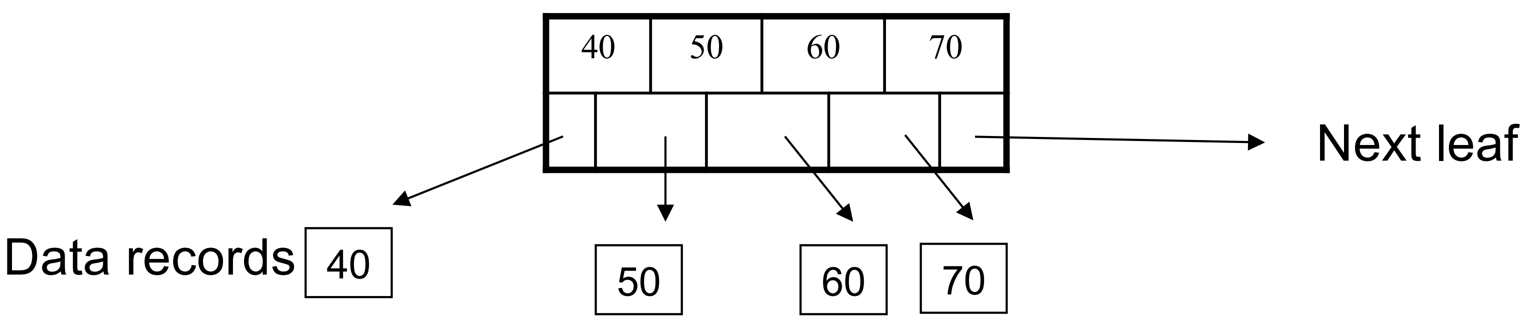
Note: can also store data records directly as data entries

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+ Trees Properties

- For each node except the root, maintain 50% occupancy of keys
- Insert and delete must rebalance to maintain constraints

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

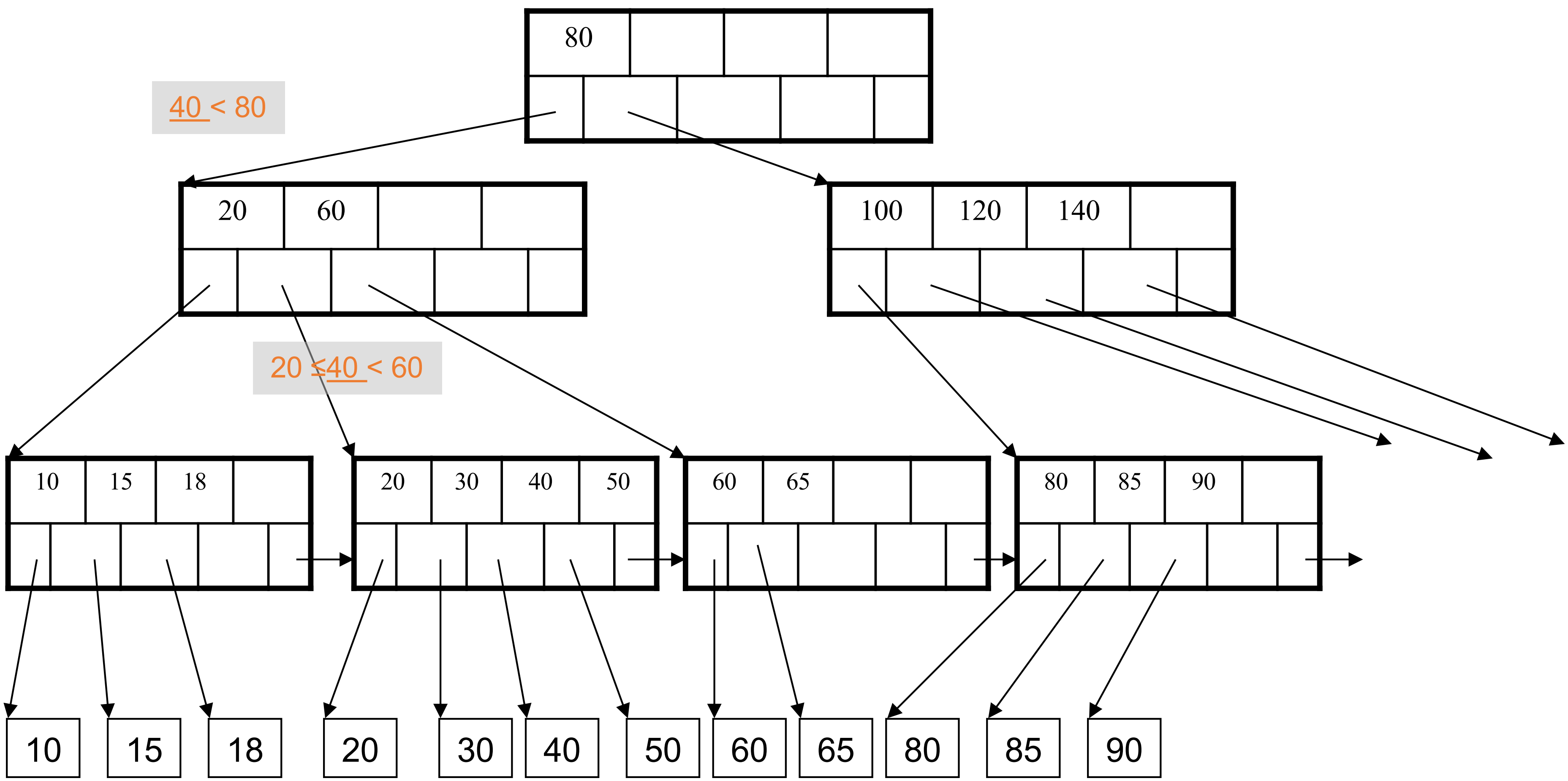
```
Select name
From Student
Where age = 25
```

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

B+ Tree Example

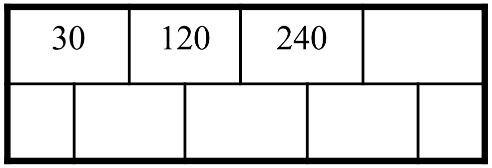
d = 2

Find the key 40



B+ Tree Design

- How large d ?   Make one node fit on one block
- Example:
  - Key size = 4 bytes
  - Pointer size = 8 bytes
  - Block size = 4096 bytes



(e.g. d = 2)

- 2d x 4 + (2d+1) x 8 <= 4096
- d = 170

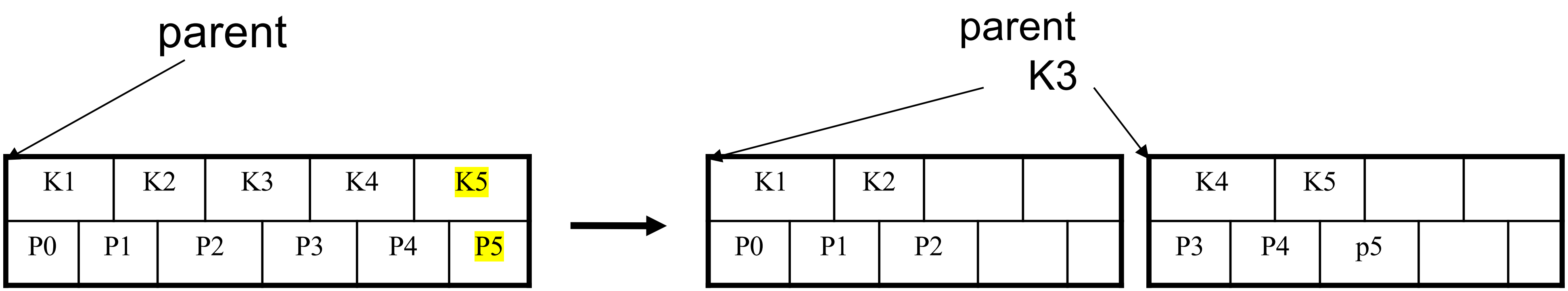
B+ Trees in Practice

- 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)

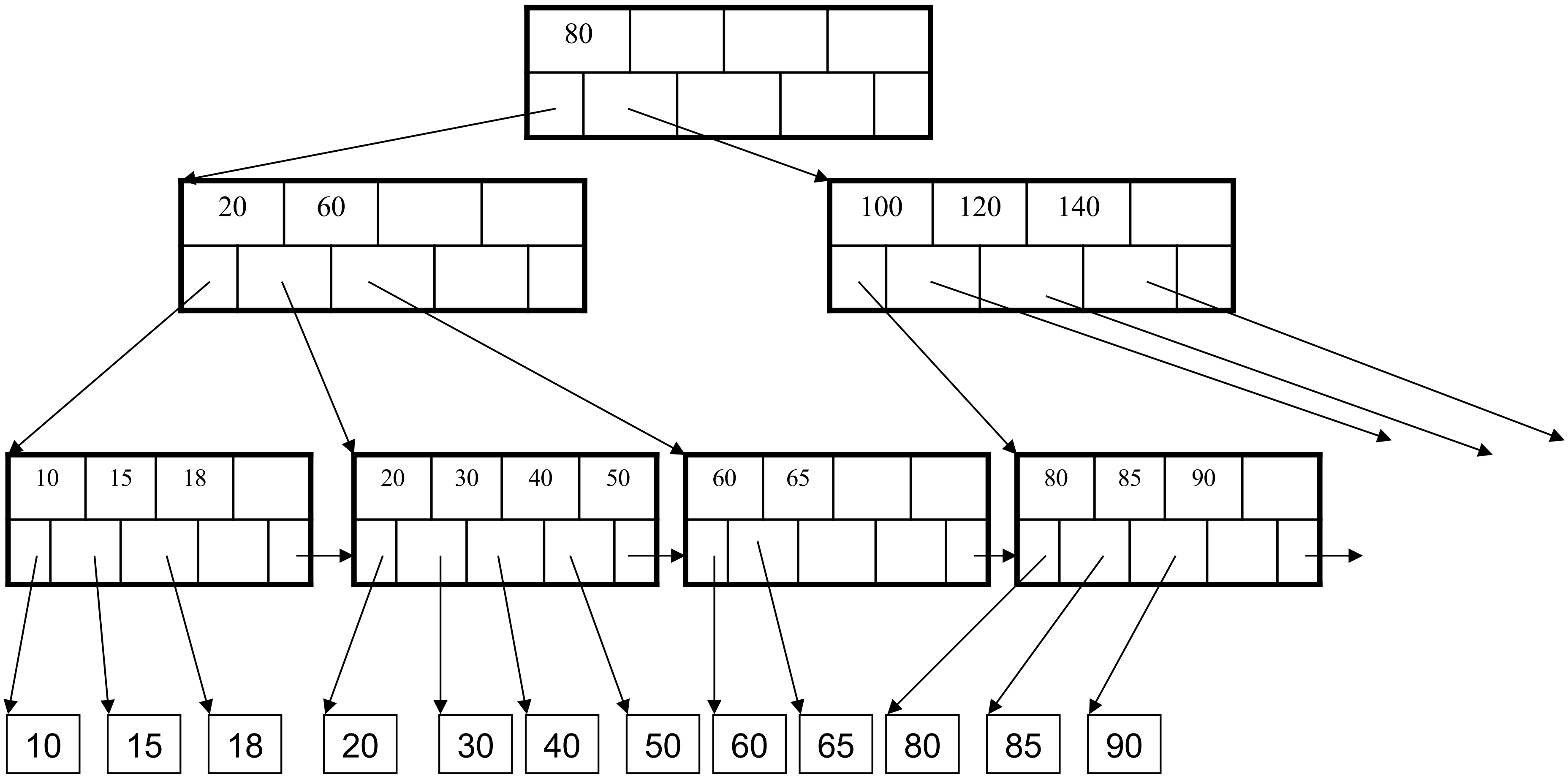
- Find leaf where K belongs, insert
- If no overflow (2d keys or less), halt
- If overflow (2d+1 keys), split node, insert in parent:



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

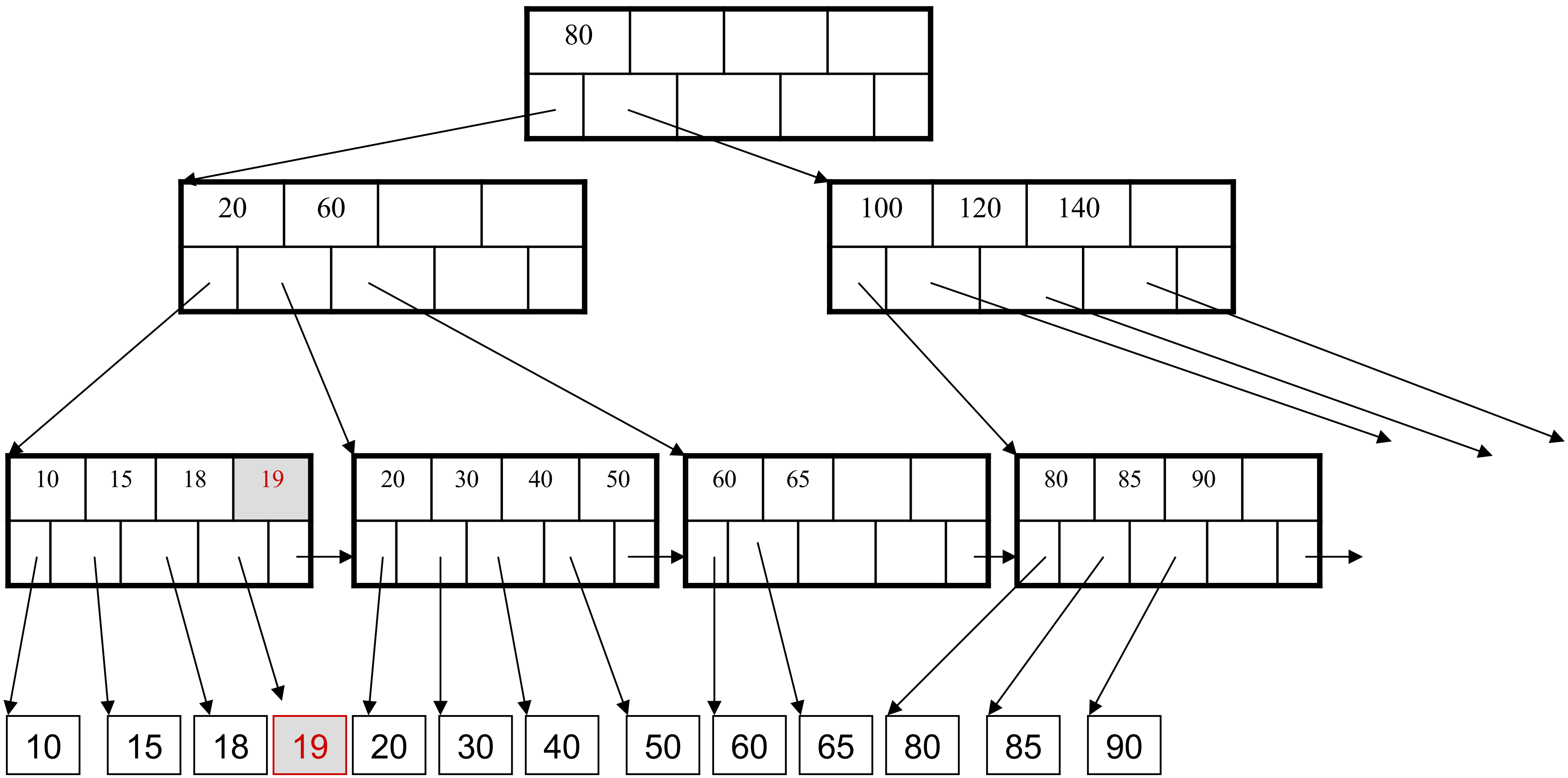
Insertion in a B+ Tree

Insert K=19



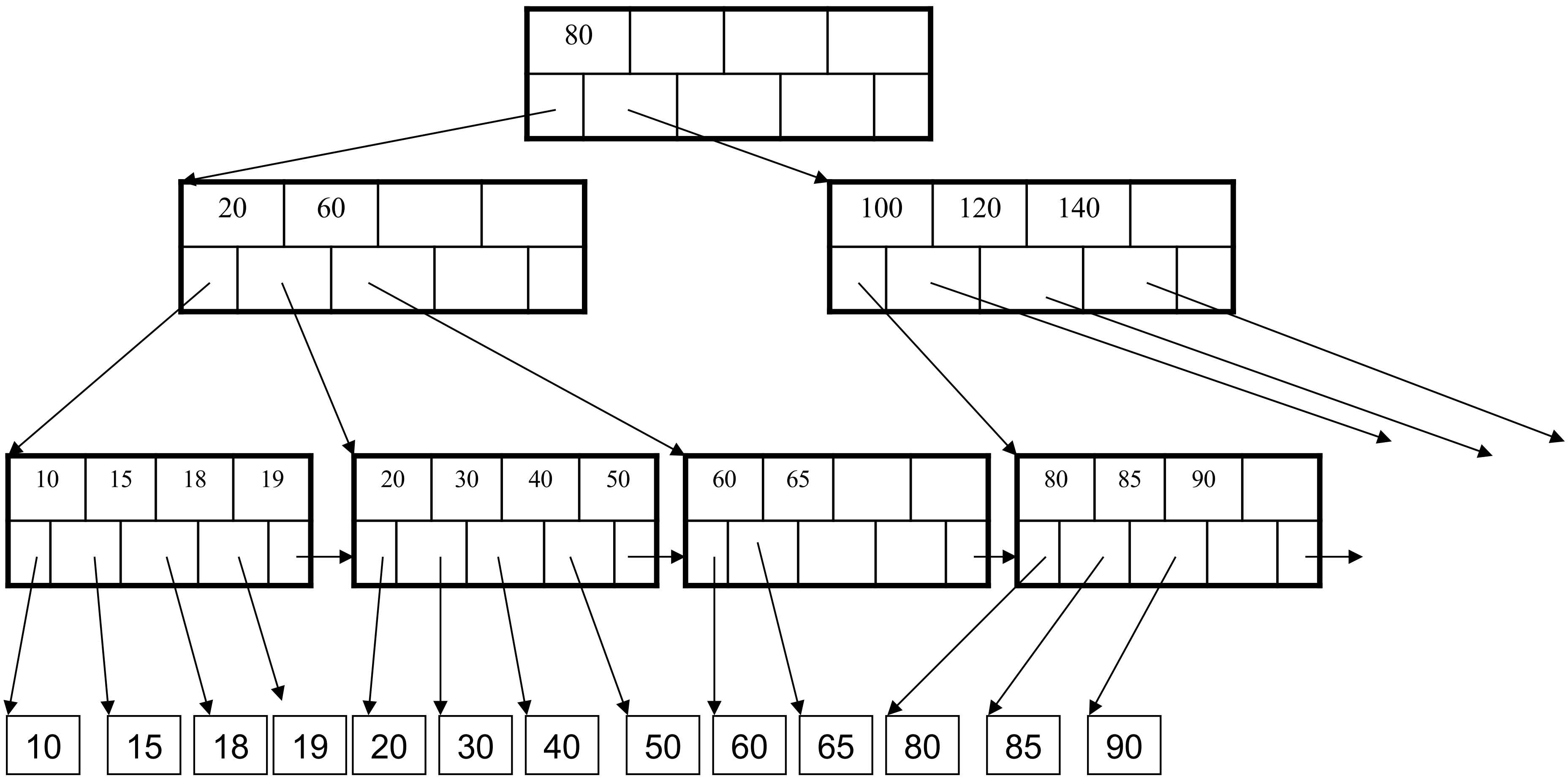
Insertion in a B+ Tree

After insertion



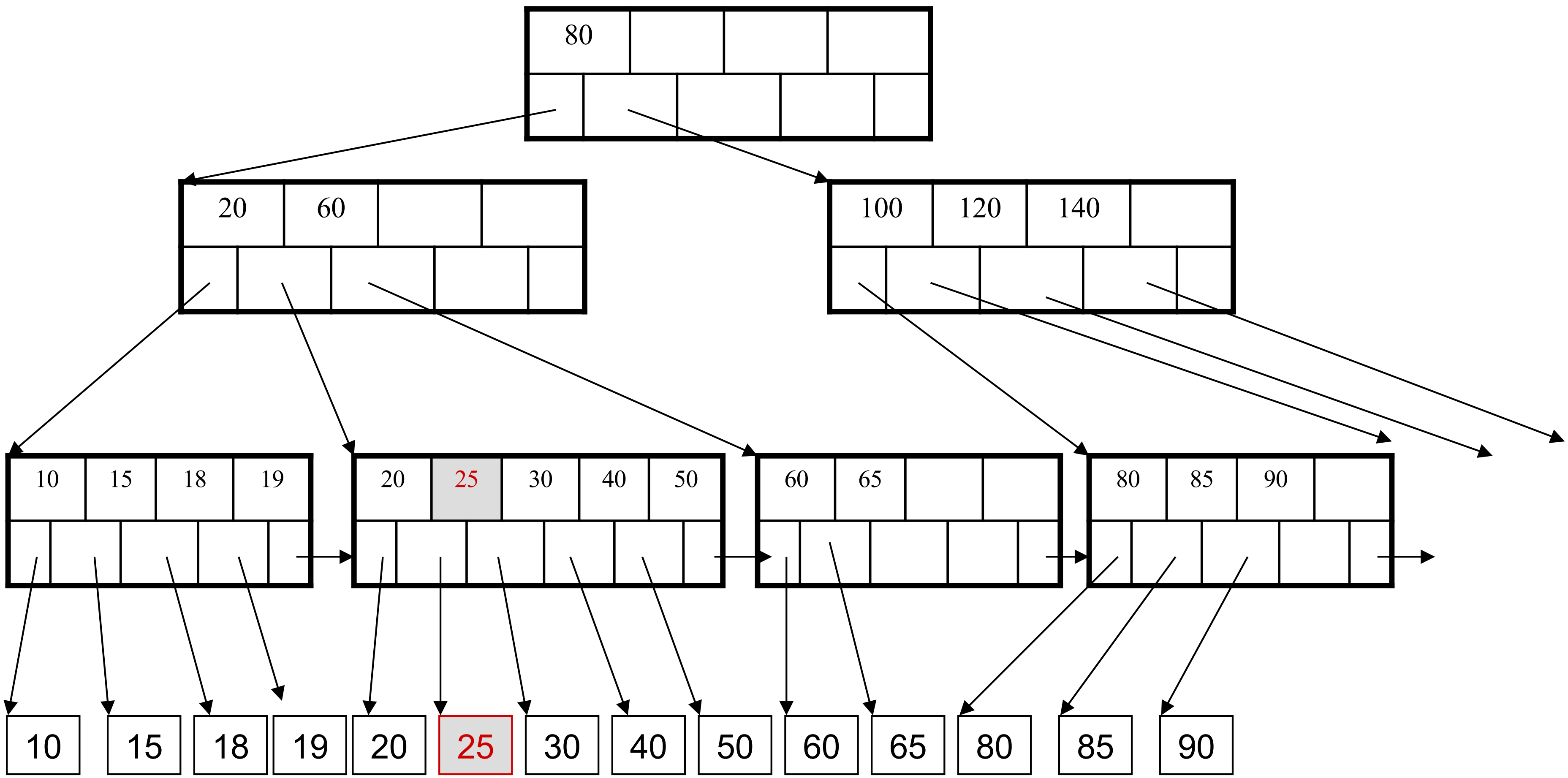
Insertion in a B+ Tree

Now insert 25



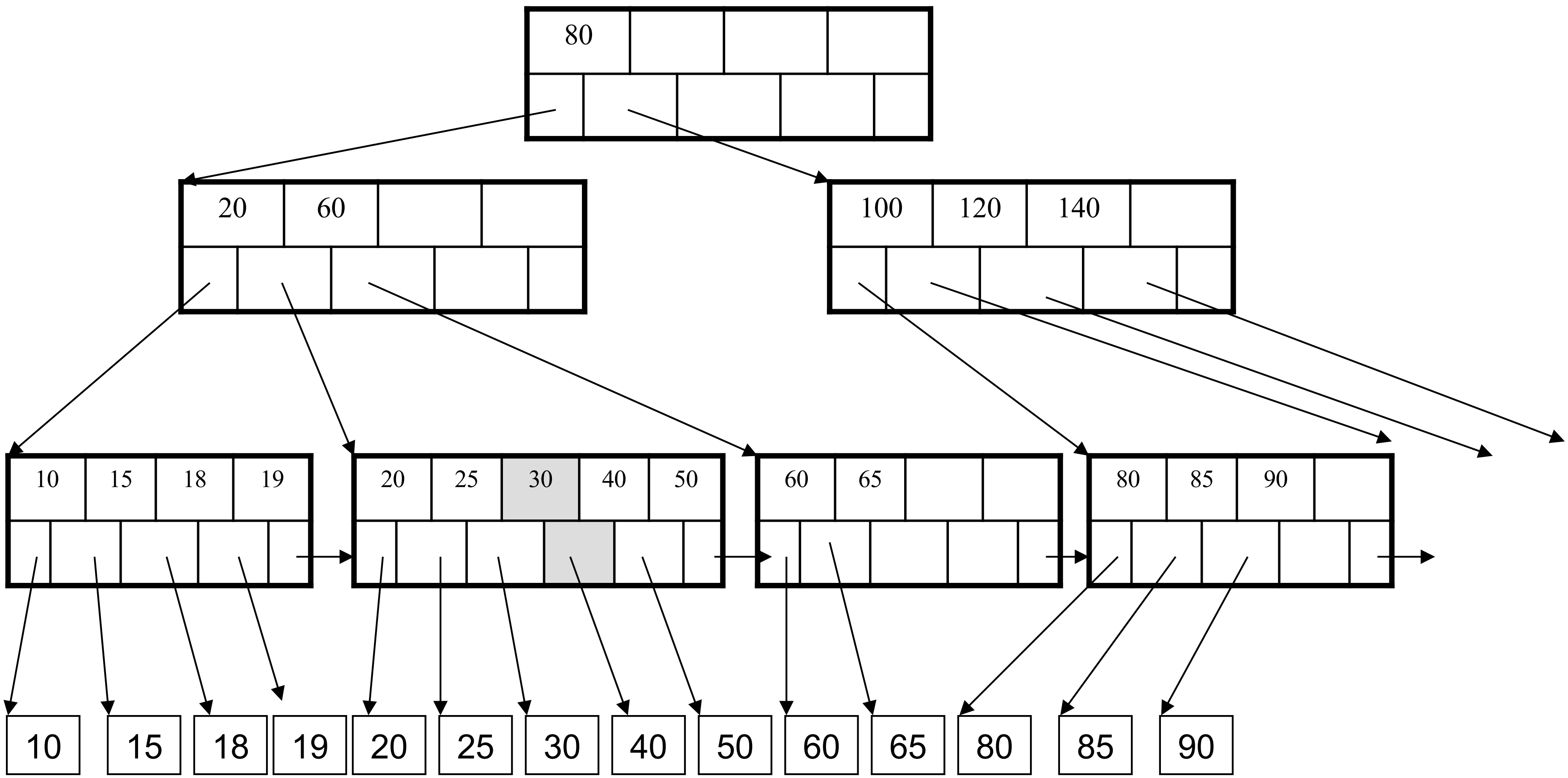
Insertion in a B+ Tree

After insertion



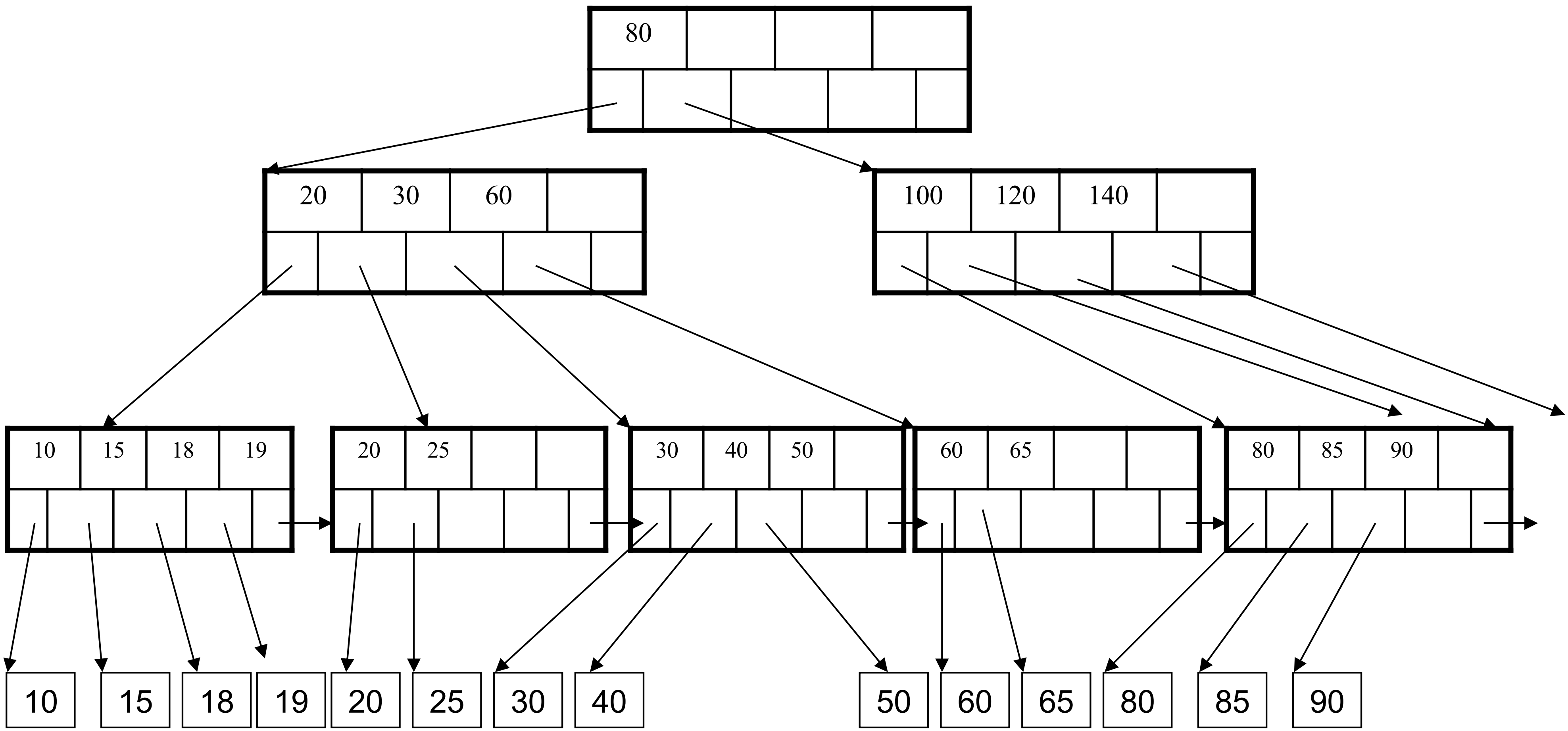
Insertion in a B+ Tree

But now have to split !



Insertion in a B+ Tree

After the split



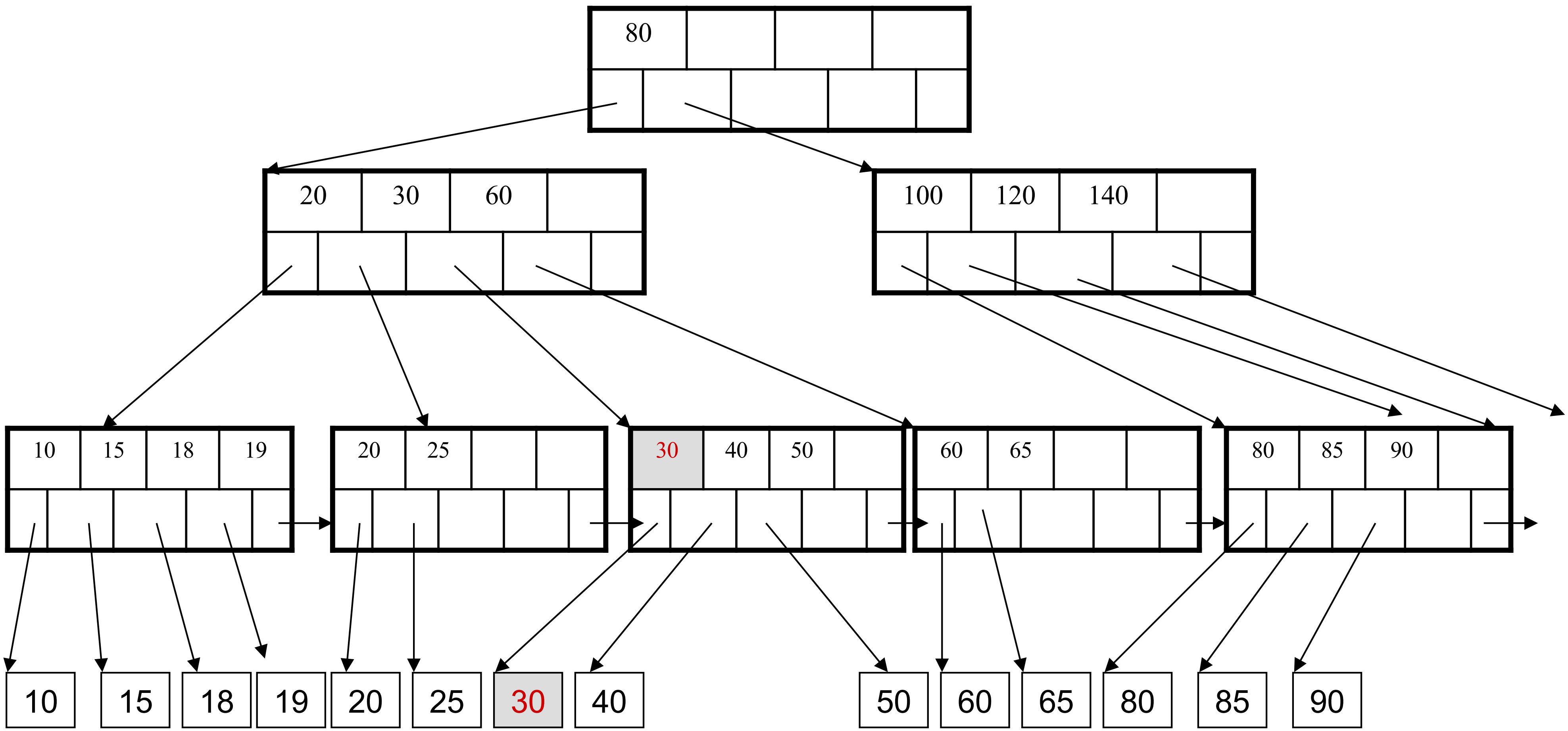
Deletion in a B+ Tree

Delete (K, P)

- Find leaf where K belongs, delete
- Check for capacity
- If leaf below capacity, search adjacent nodes (left first, then right) for extra tuples and rotate them to new leaf
- If adjacent nodes at 50% full, merge
- Update and repeat algorithm on parent nodes if necessary

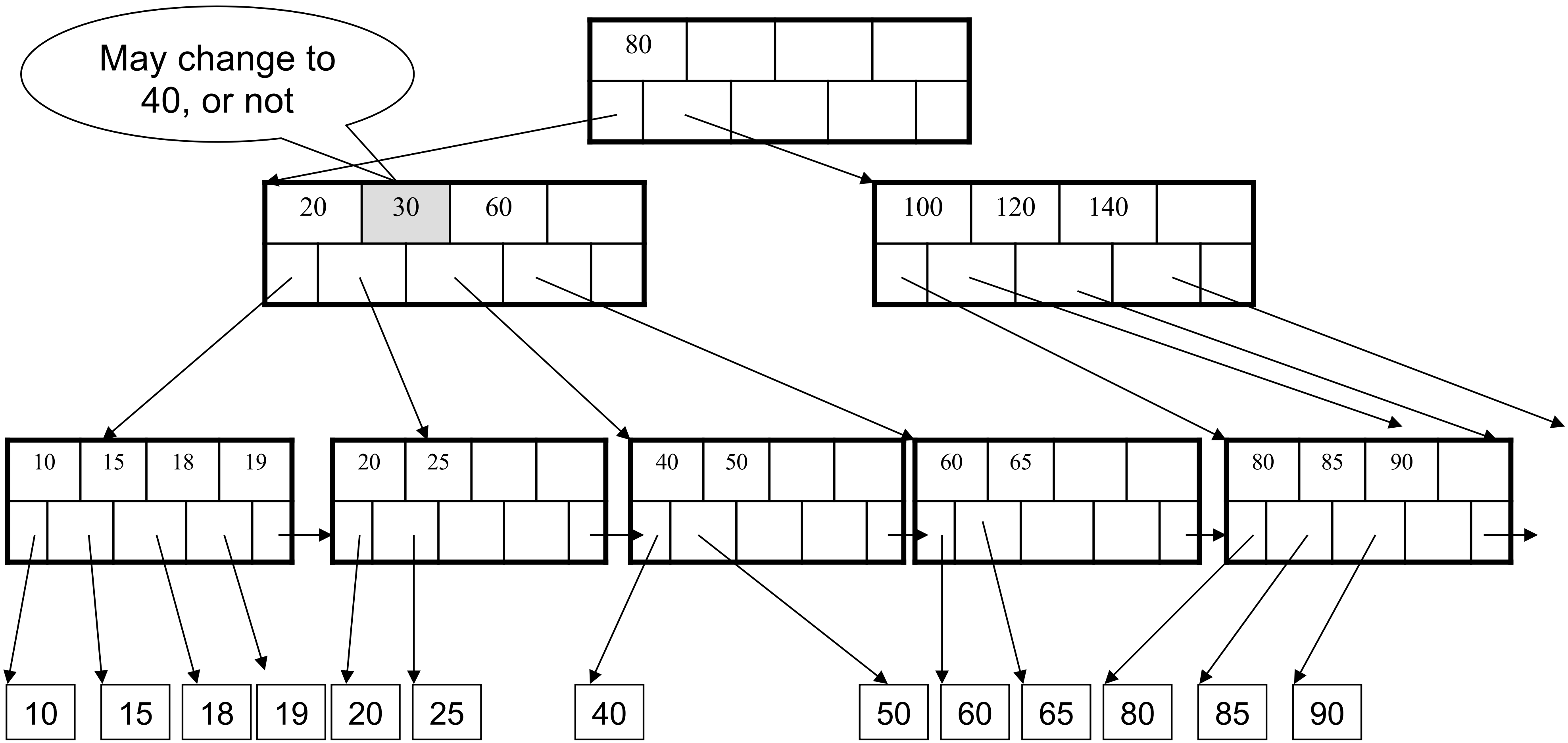
Deletion from a B+ Tree

Delete 30



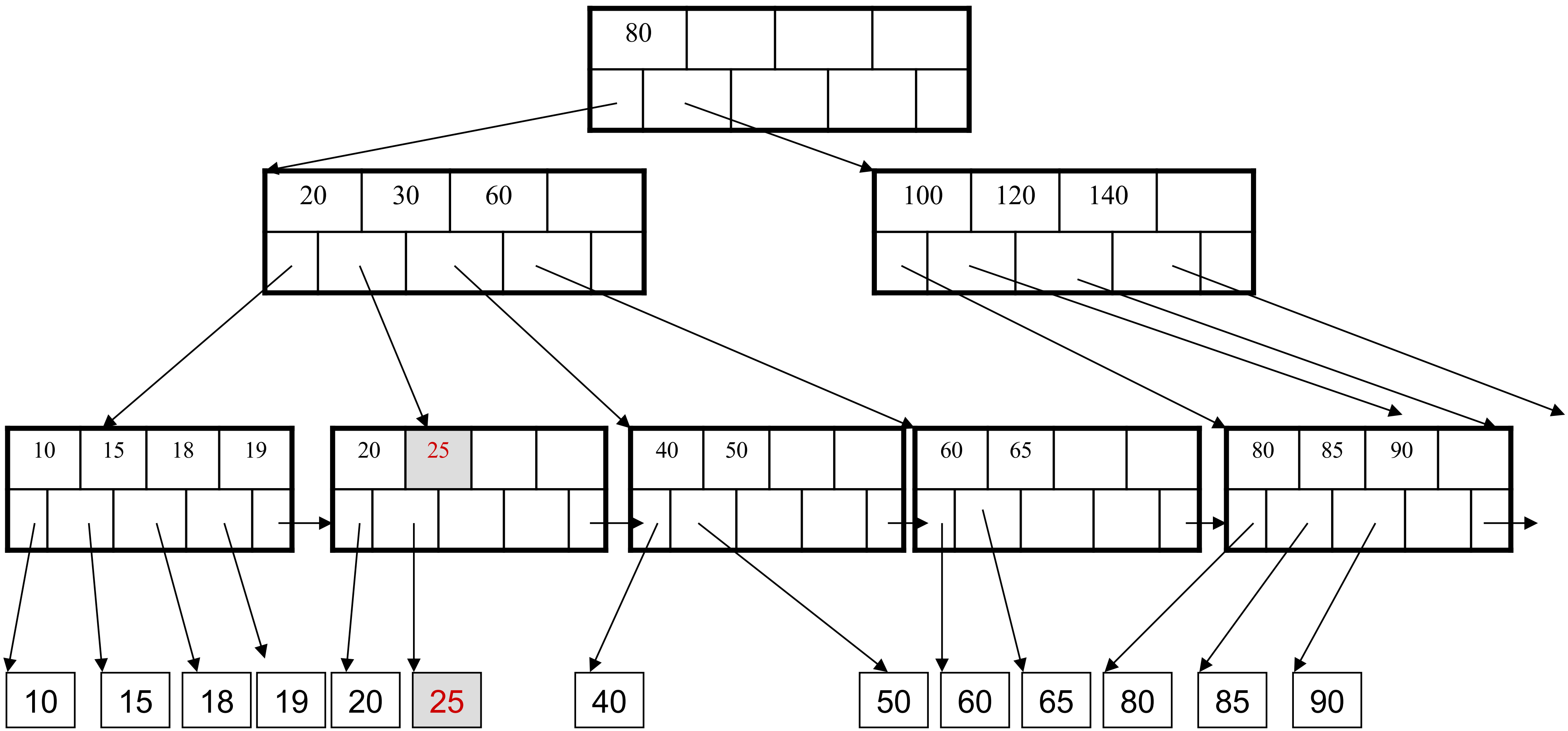
Deletion from a B+ Tree

After deleting 30



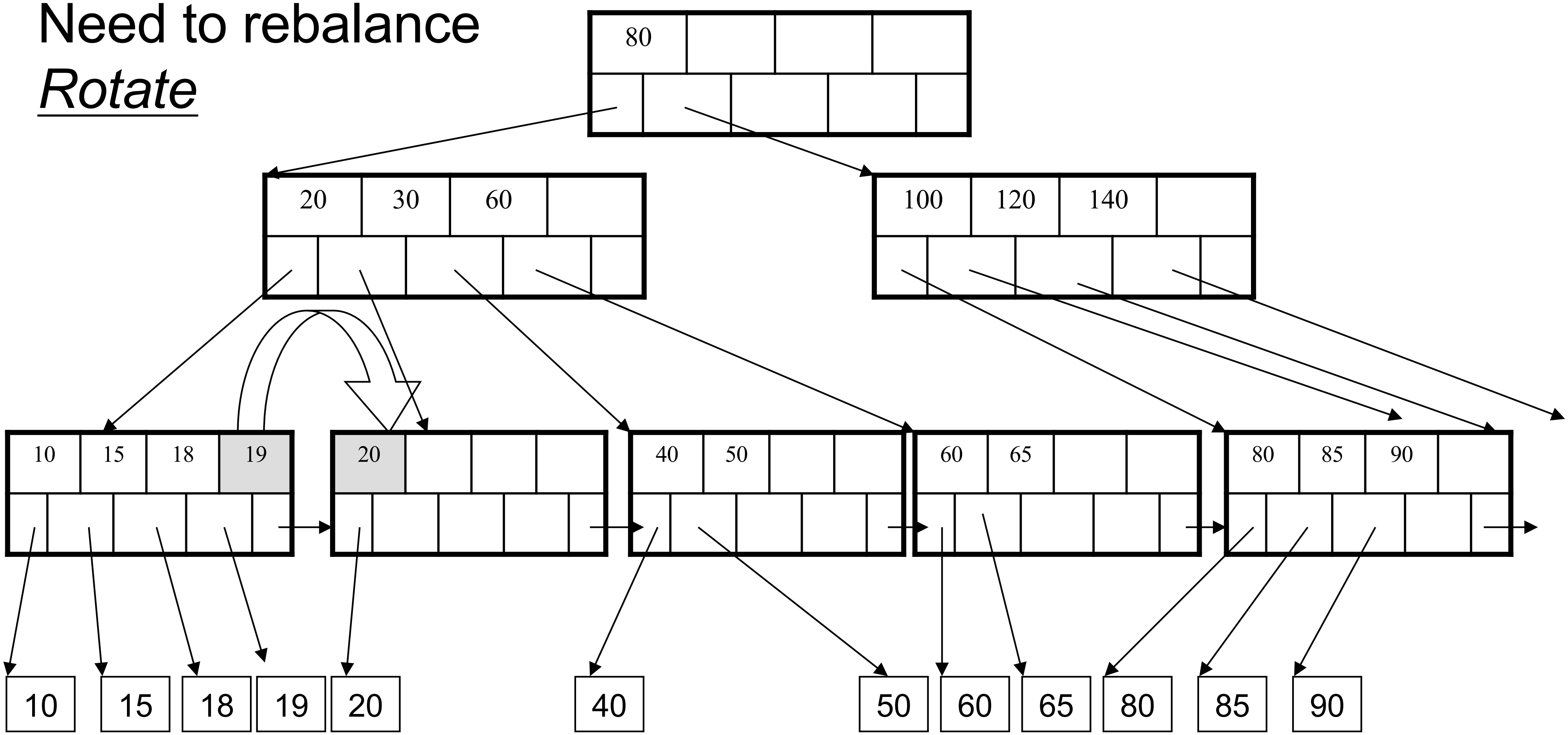
Deletion from a B+ Tree

Now delete 25



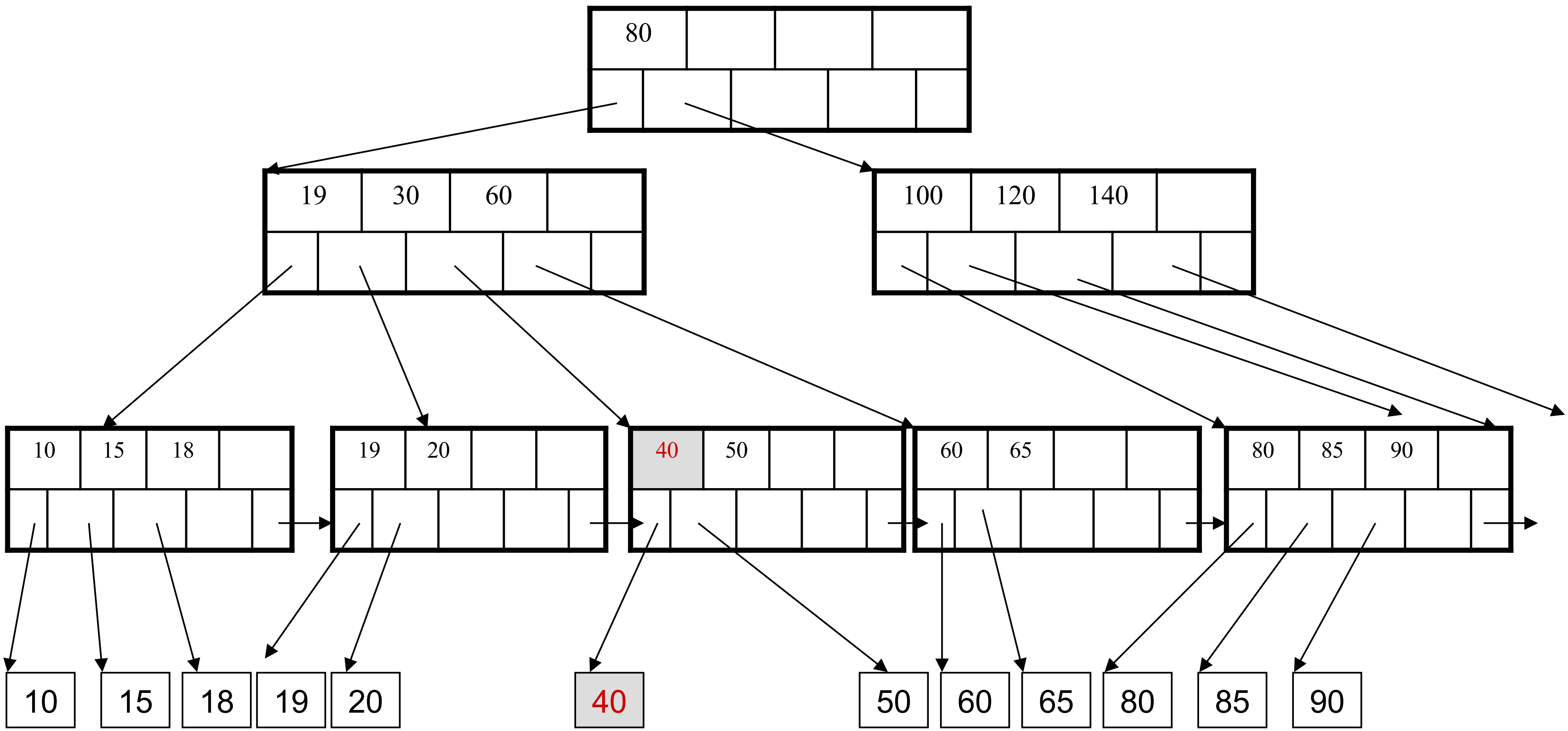
Deletion from a B+ Tree

After deleting 25
Need to rebalance
Rotate



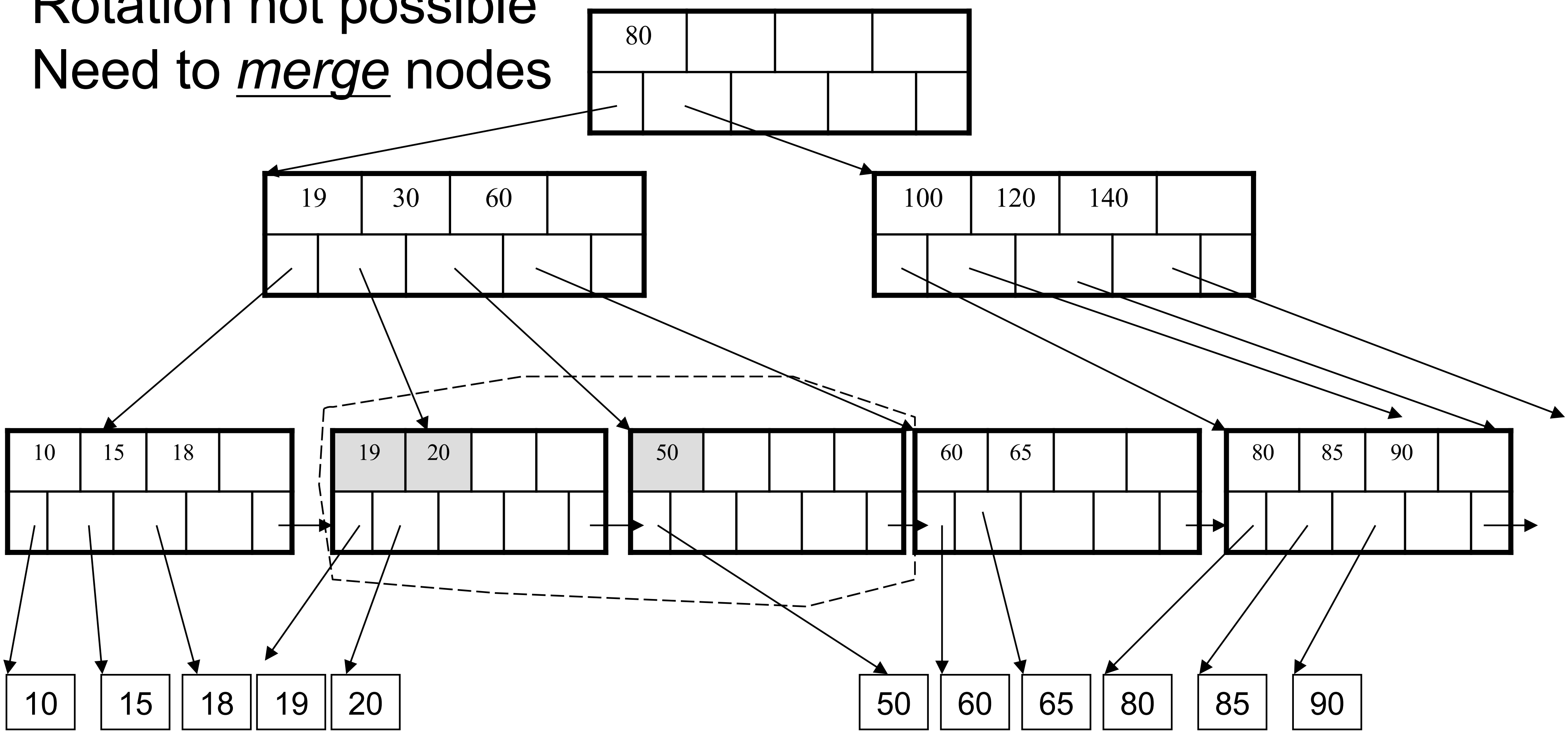
Deletion from a B+ Tree

Now delete 40



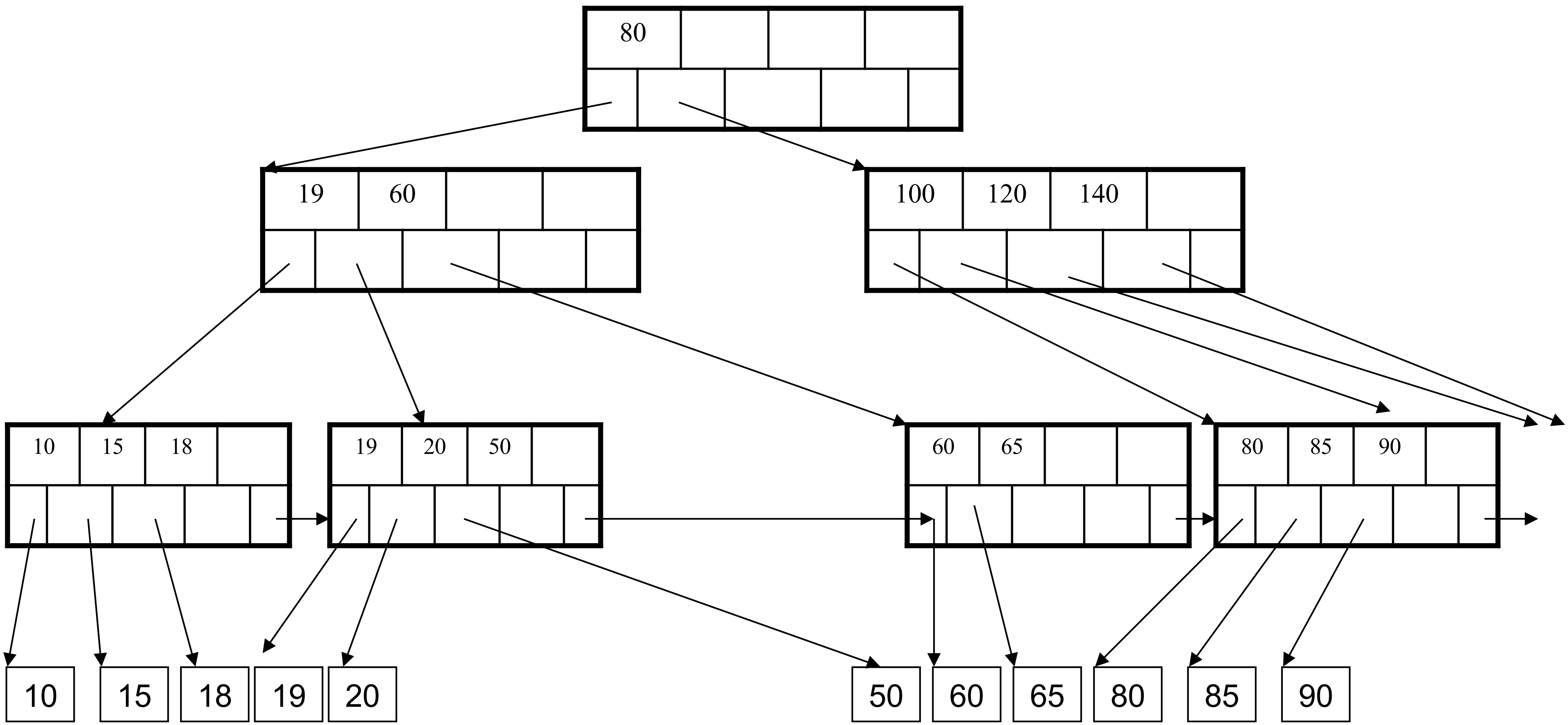
Deletion from a B+ Tree

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



Deletion from a B+ Tree

Final tree



Summary 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