CSE544
Data Management

Lectures 5: Storage + Indexes

Announcements

- HW1 due on Friday
- Review 3 due on Wednesday

Where We are

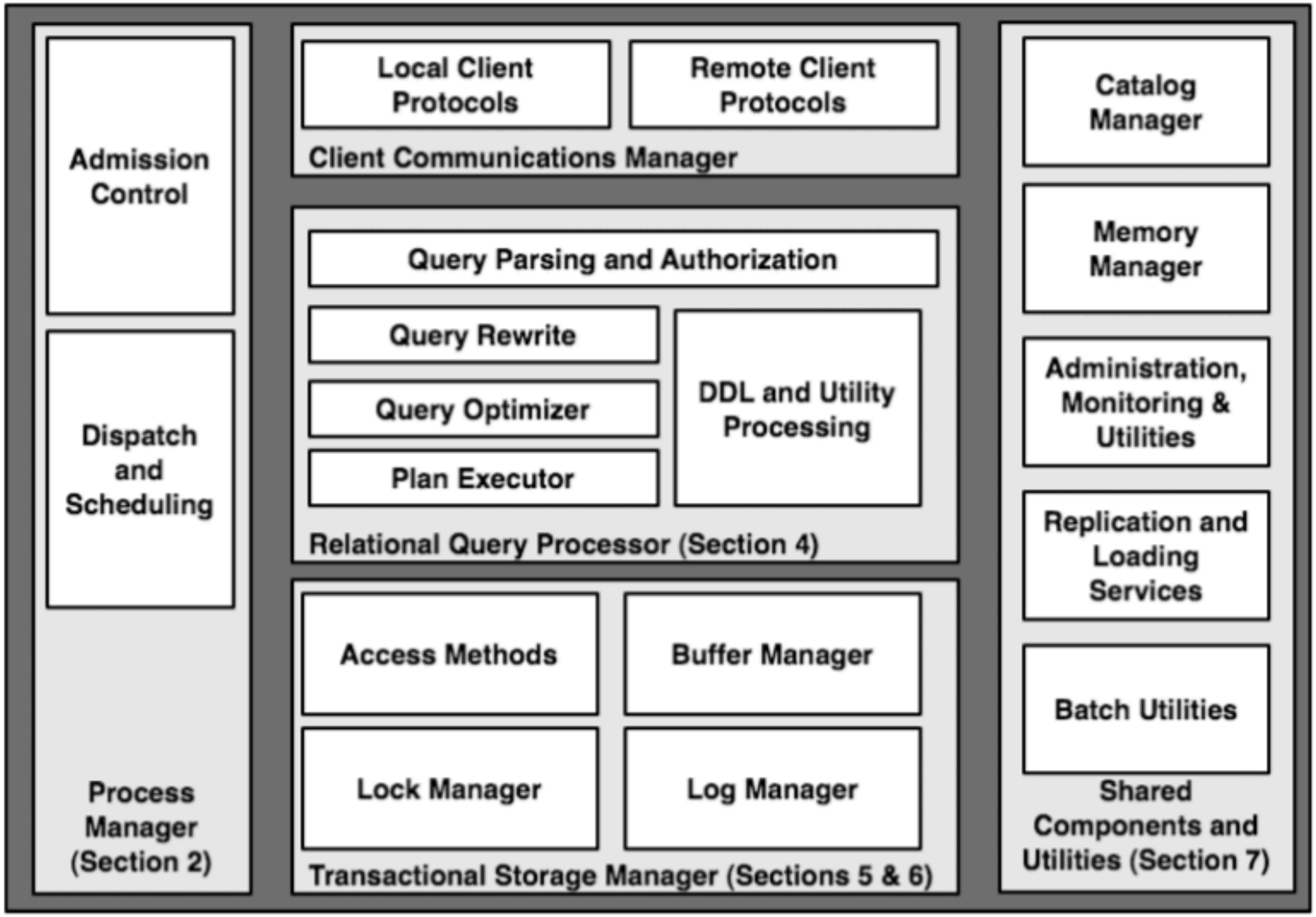
- SQL+RA
- Relational data model
- Query Processor
  - Storage/Indexes
  - Execution
  - Optimization
  - Recursive queries: Datalog
  - Advanced techniques (Bloom, LSM)
- Distributed Query Processing
- TXNs

Where We are

- SQL+RA
- Relational data model
- Query Processor
  - Storage/Indexes
  - Execution
  - Optimization
  - Recursive queries: Datalog
  - Advanced techniques (Bloom, LSM)
- Distributed Query Processing
- TXNs

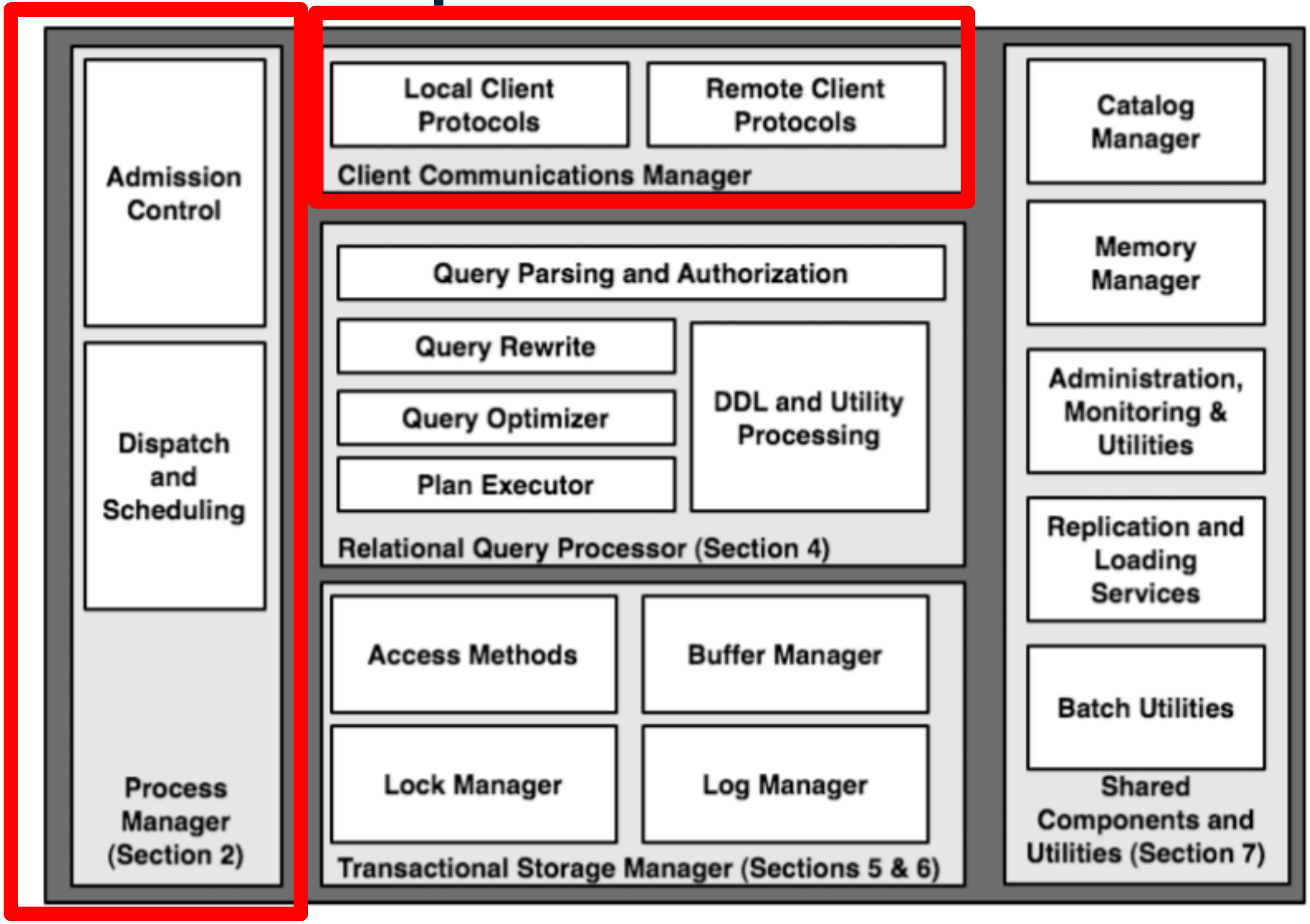
[Architecture of a Database System, Hellerstein, Stonebraker, Hamilton]

Architecture of DBMS



[Architecture of a Database System, Hellerstein, Stonebraker, Hamilton]

Multiple Processes



Why Multiple Processes

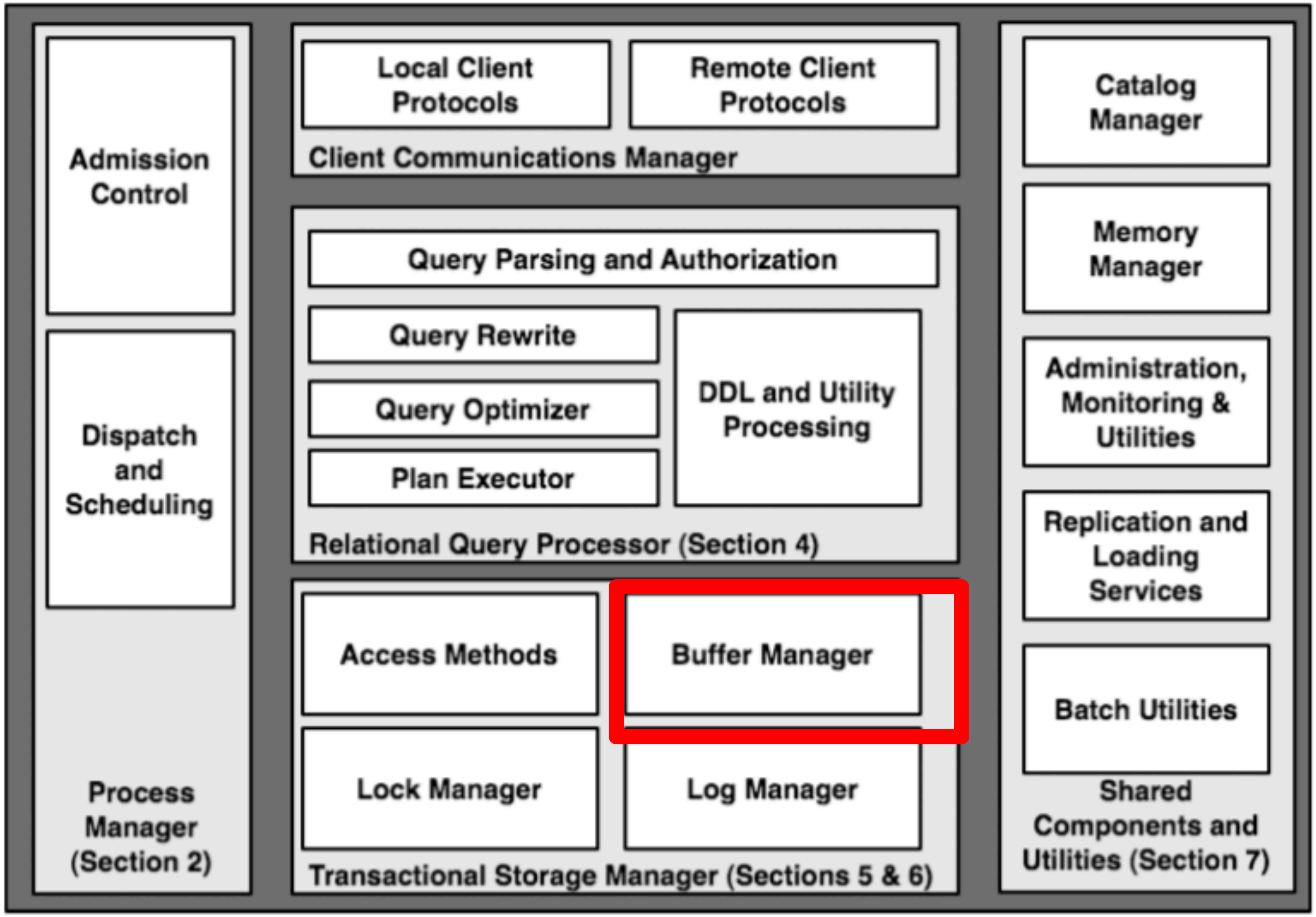
- DBMS listens to requests from clients
- Each request = one SQL command
- Handles multiple requests concurrently; multiple processes

Process Models

- Process per DBMS worker
- Thread per DBMS worker
- Process pool

[Architecture of a Database System, Hellerstein, Stonebraker, Hamilton]

Storage



The Mechanics of Disk

Mechanical characteristics:

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

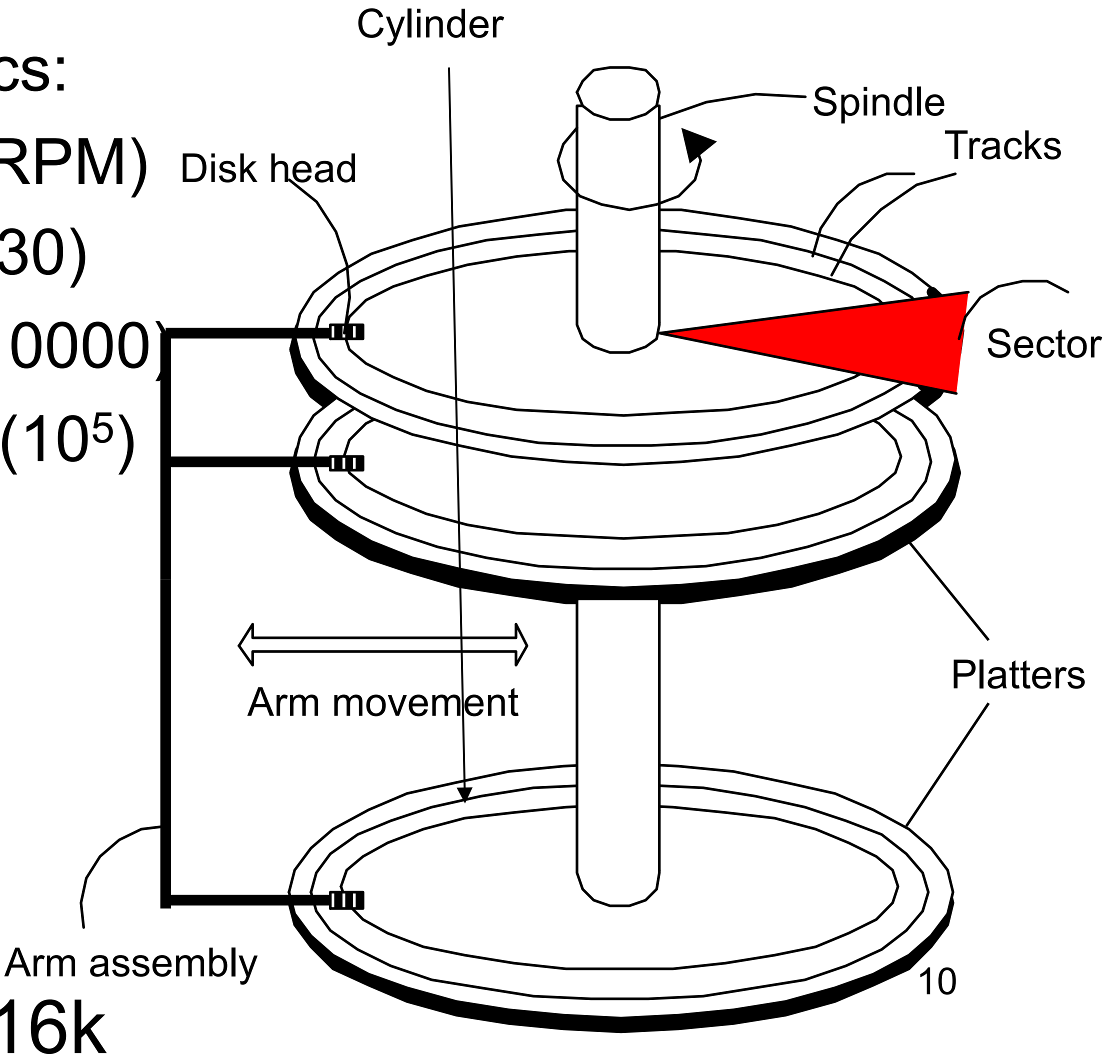
Unit of read or write:

disk block

Once in memory:

page

Typically: 4k or 8k or 16k



Disk Access Characteristics

- Disk latency
  - Time between request and when data is in memory
    = seek time + rotational latency
- Seek time = time for the head to reach cylinder
  - 10ms – 40ms
- Rotational latency = time for sector to rotate
  - Rotation time = 10ms
  - Average latency = 10ms / 2
- Transfer time = typically 40-80MB/s

Disks access MUCH slower than main memory

Storage Technologies

Architecture: buffer pool

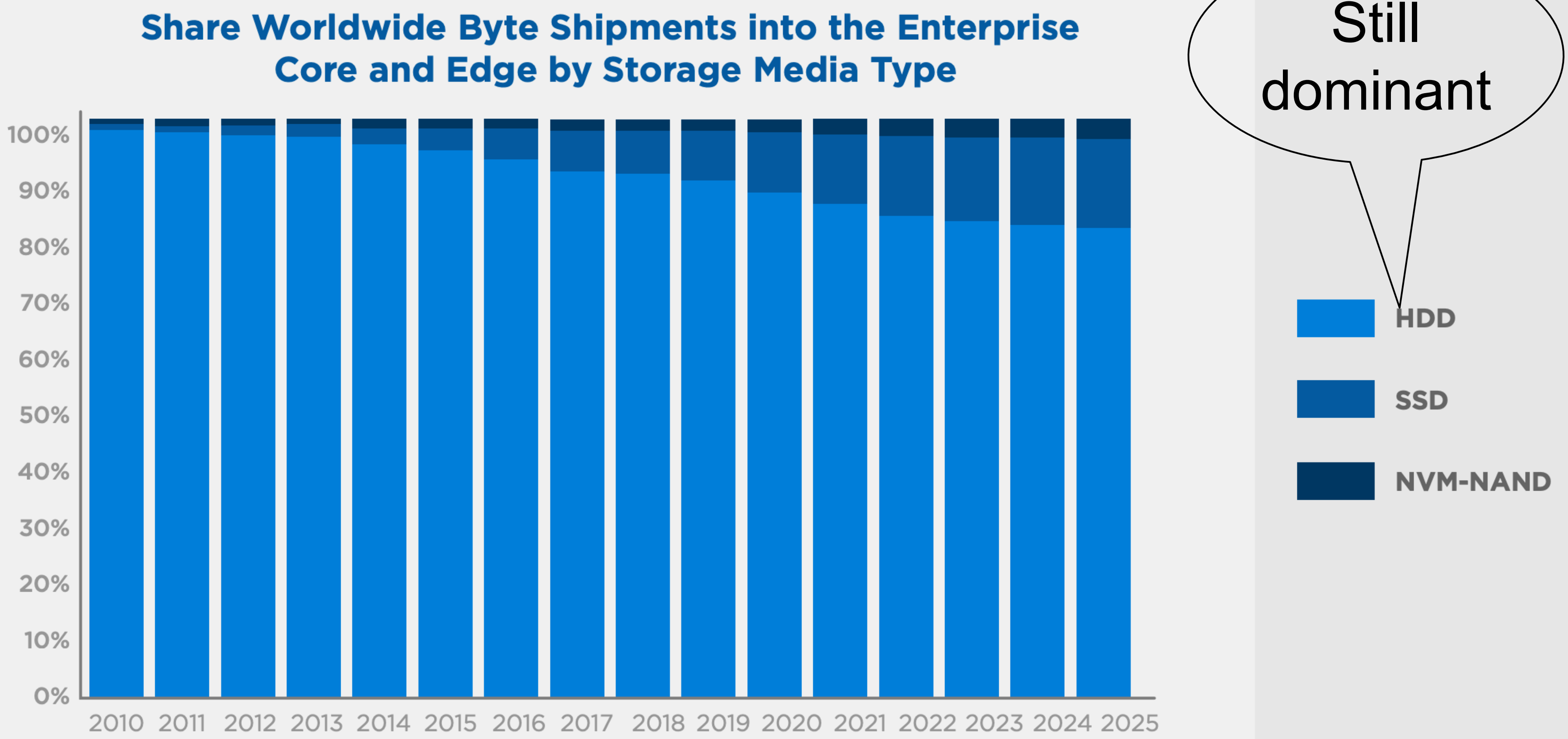
- Hard Drive Disk HDD
  - $
  - Latency << main memory
  - Block addressable
  - Random >> sequential
- Solid State Drive SDD
  - $$
  - Latency < main memory
  - Block addressable (at least for writes)
  - Random > sequential

Same here

- Non-volatile memory NVM
  - $$$
  - Latency ~ main memory
  - Byte addressable
  - Random ~ sequential

Requires new architecture (still research) https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf

Figure 11 - Share Worldwide Byte Shipments into the Enterprise Core and Edge by Storage Media Type



Still dominant

Data Storage

Student

ID	fName	lName
10	Tom	Hanks
20	Amy	Hanks
…		

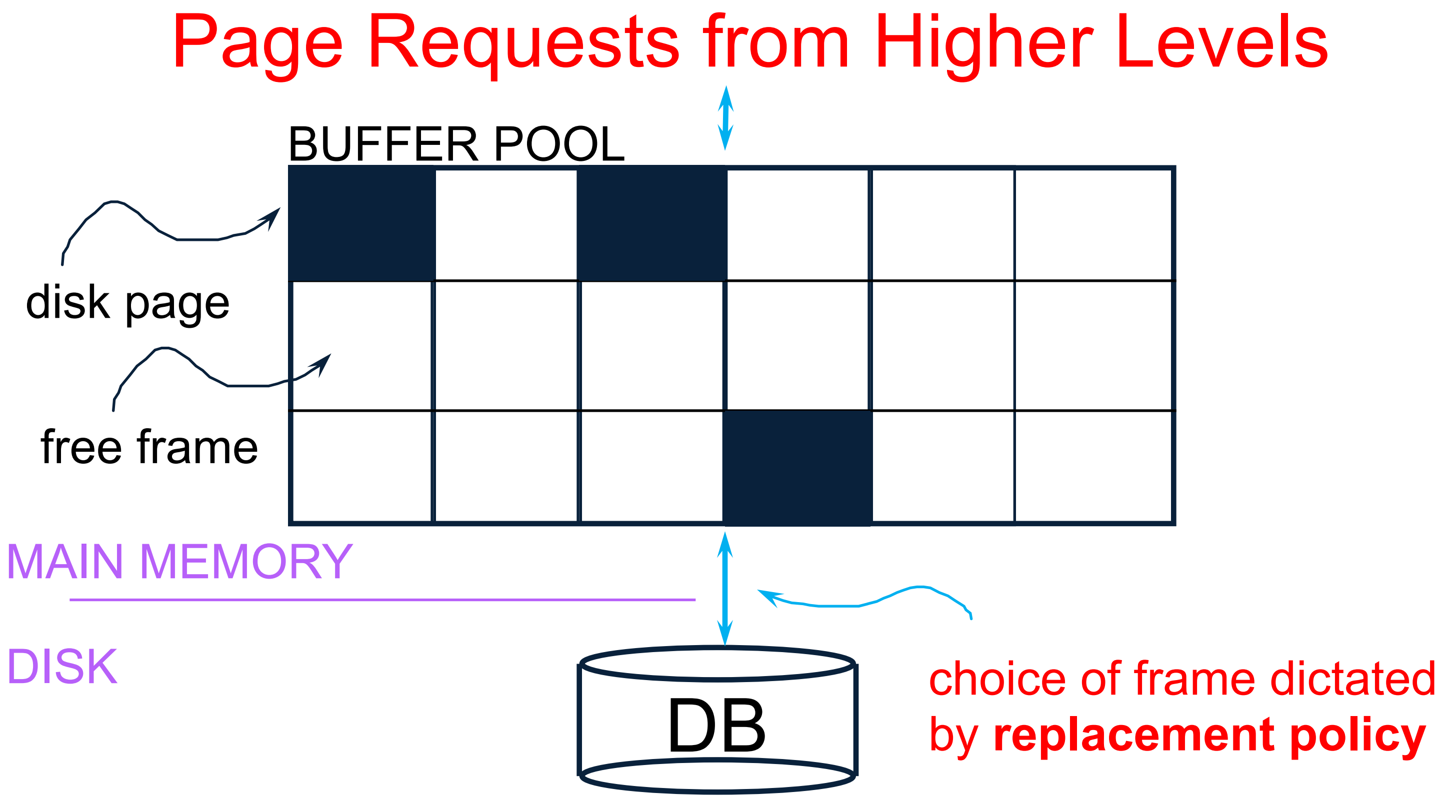
- DBMSs store data in **files**
- Most common organization is row-wise storage
- On disk, a file is split into blocks
- Each block contains a set of tuples

10	Tom	Hanks	block 1
20	Amy	Hanks	
50	…	…	block 2
200	…		
220			block 3
240			
420			
800			

In the example, we have 4 blocks with 2 tuples each

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

Buffer Manager



- 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

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!

Storing Records On Disk

- Page format: records inside a page
- Record format: attributes inside a record
- File Organization

Page Format

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

Need RID’s for indexes and for transactions

Page Format Approach 1

Fixed-length records: packed representation

Divide page into **slots**. Each slot can hold one tuple

Record ID (RID) for each tuple is (PageID,SlotNb)

Number of records

How do we insert a new record?

Page Format Approach 1

Fixed-length records: packed representation

Divide page into **slots**. Each slot can hold one tuple

Record ID (RID) for each tuple is (PageID,SlotNb)

Number of records

How do we insert a new record?

Page Format Approach 1

Fixed-length records: packed representation
Divide page into **slots**. Each slot can hold one tuple
Record ID (RID) for each tuple is (PageID,SlotNb)

Slot$_1$ Slot$_2$ Slot$_N$ Slot$_{N+1}$

Free Sp. N

Number of records

How do we insert a new record?

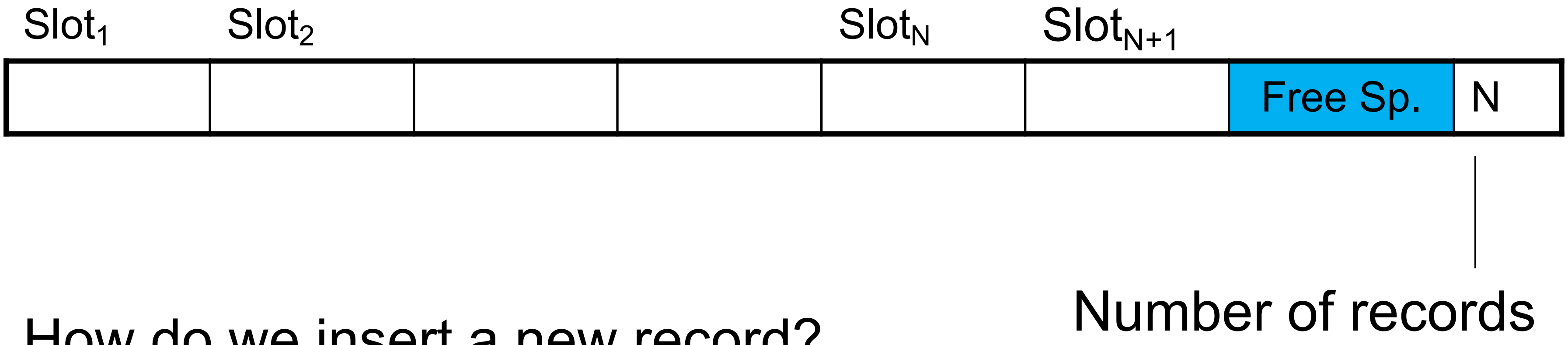
How do we delete a record?

Page Format Approach 1

Fixed-length records: packed representation

Divide page into **slots**. Each slot can hold one tuple

Record ID (RID) for each tuple is (PageID,SlotNb)



Slot$_1$ Slot$_2$ Slot$_N$ Slot$_{N+1}$

Free Sp.

N

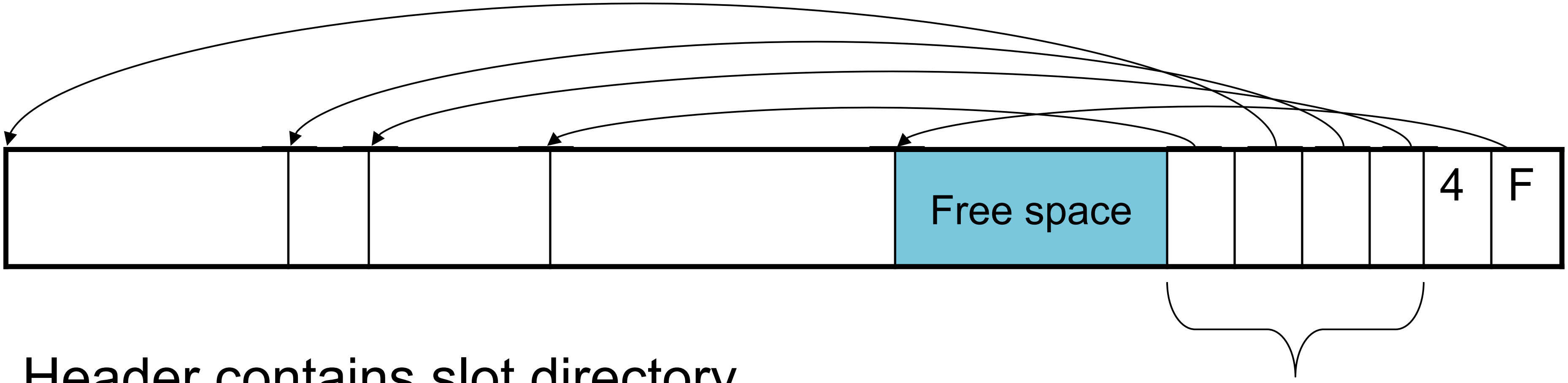
Number of records

How do we insert a new record?

How do we delete a record? Cannot remove record (why?)

How do we handle variable-length records?

Page Format Approach 2



Slot directory

Header contains slot directory
+ Need to keep track of nb of slots
+ Also need to keep track of free space (F)

Can handle variable-length records
Can move tuples inside a page without changing RIDs
RID is (PageID, SlotID) combination

Record Formats

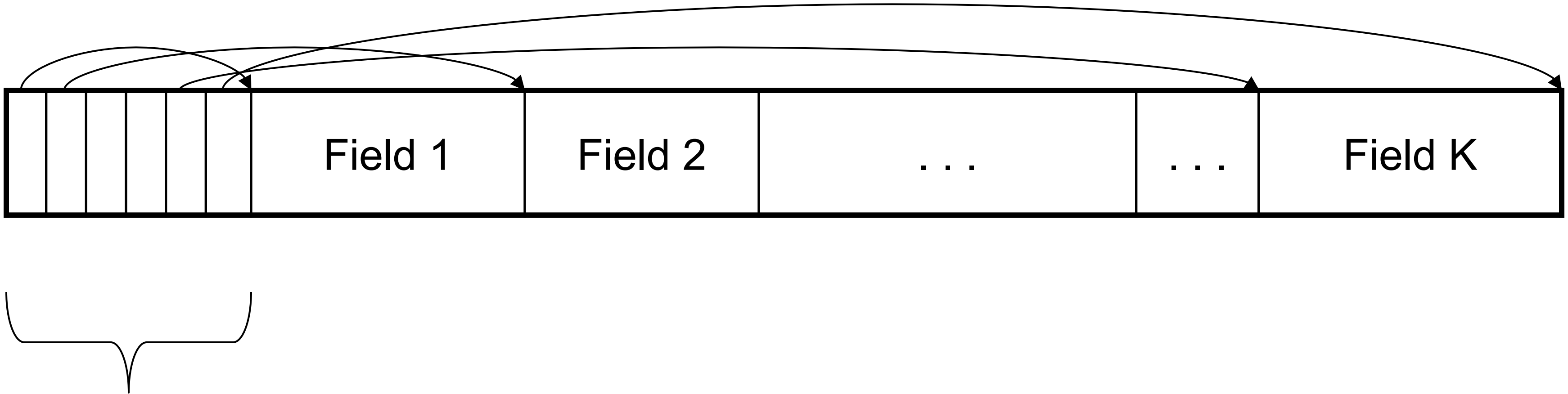
Fixed-length records => Each field has a fixed length (i.e., it has the same length in all the records)

Field 1	Field 2	. . .	. . .	Field K

Information about field lengths and types is in the catalog

Record Formats

Variable length records



Field 1

Field 2

. . .

. . .

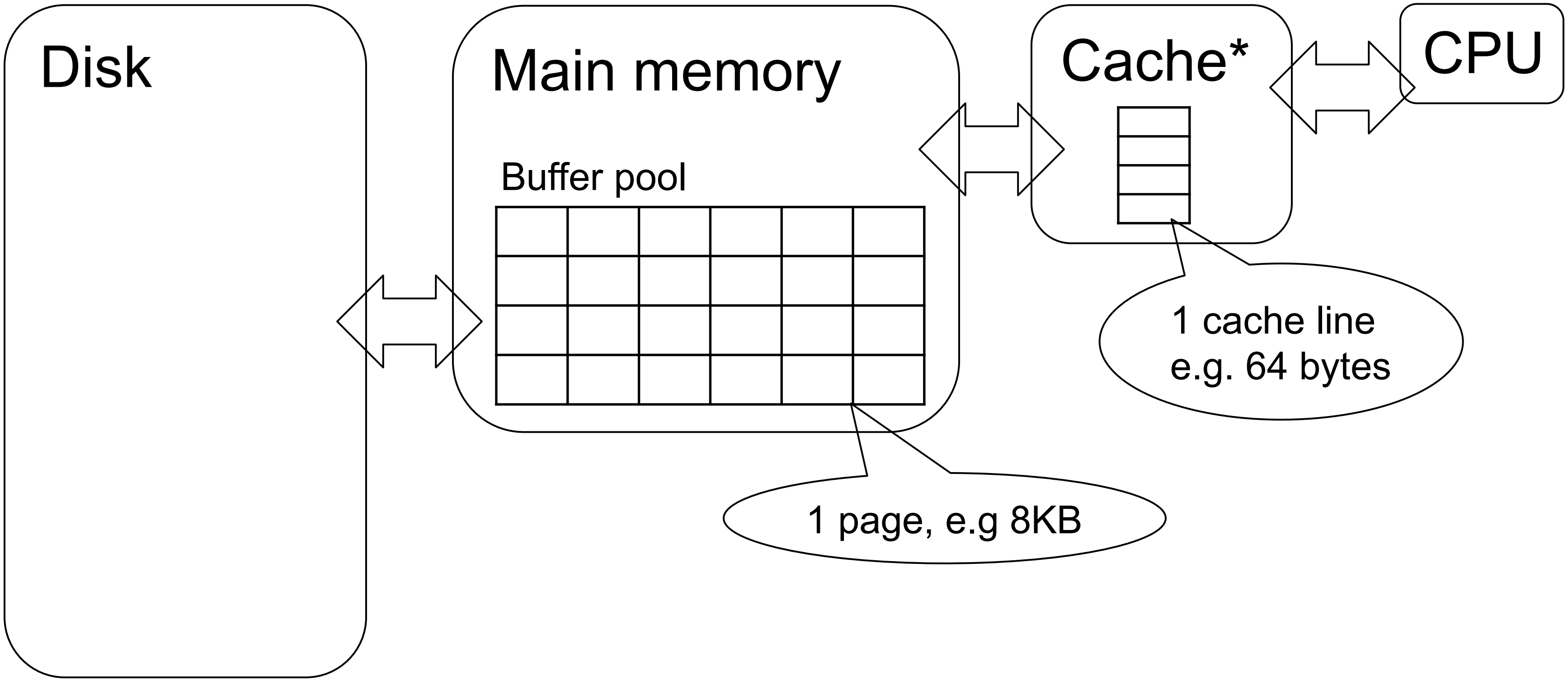
Field K

Record header

Remark: NULLS require no space at all (why ?)

Notes for the PAX paper

Memory hierarchies:



*aka CPU cache; several! L3, L2, L1 cache

File Organizations

- Heap (random order) files: Suitable when typical access is a file scan retrieving all records.
- Sequential file (sorted): Best if records must be retrieved in some order, or by a `range’
- Index: Data structures to organize records via trees or hashing.

File Organizations

Example: table STUDENT

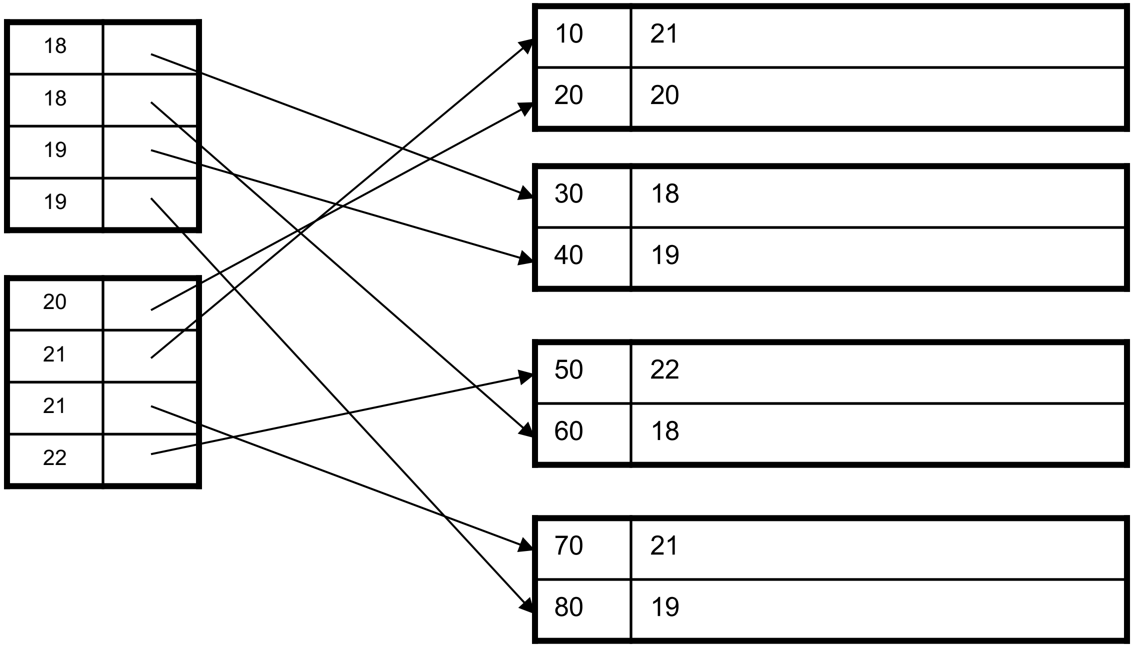
- The STUDENT file can be:
  - Heap file (tuples stored without any order)
  - Sequential file (tuples sorted on some attribute(s))
  - Clustered (primary) index file (relation+index)

- There can be several unclustered (secondary) index files that store (key,rid) pairs

Indexes

- **Index:** separate file with fast access by “key” value
- Contains pairs of the form (key, RID)

Index File
Search key: age



18
18
19
19

20
21
21
22

10 21
20 20

30 18
40 19

50 22
60 18

70 21
80 19

Data File
(sequential file sorted on sid)

Indexes

- **Search key** = can attribute or set of attributes
  - not the same as the primary key; not a key

- **Index** = collection of data entries

- **Data entry** for key k can be:
  - (k, RID)
  - (k, list-of-RIDs)
  - Record with key k; "clustered" or "primary" index

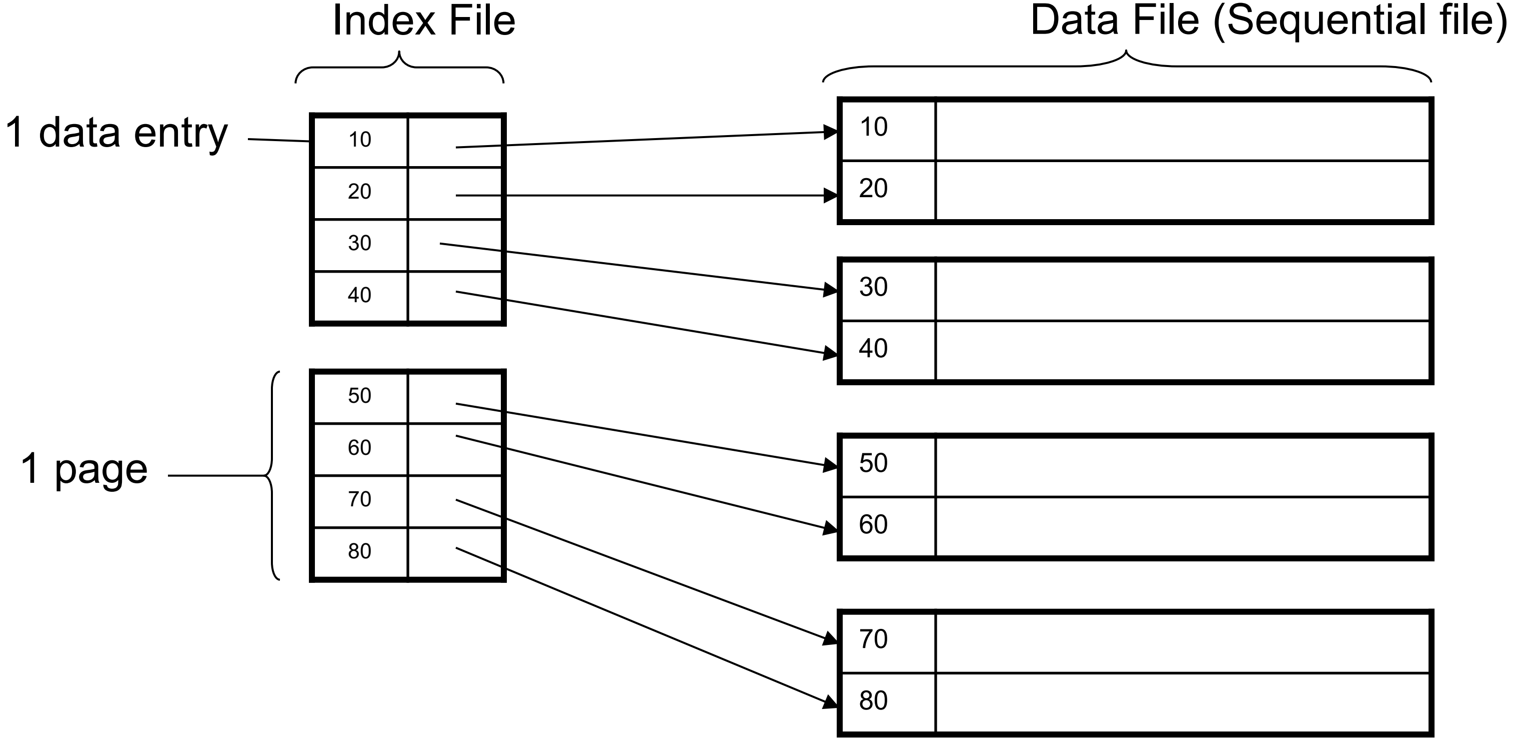
How Indexes Help

We want to support these kinds of queries
Assume Student = a heap file

- Find student where sid=12345
  - Use an index on Student(sid)
- Find students where age > 20
  - Use an index on Student(age)
- Insert a new student
  - Insert in the Student heap file -- easy
  - Insert in indexes Student(sid), Student(age) – will discuss

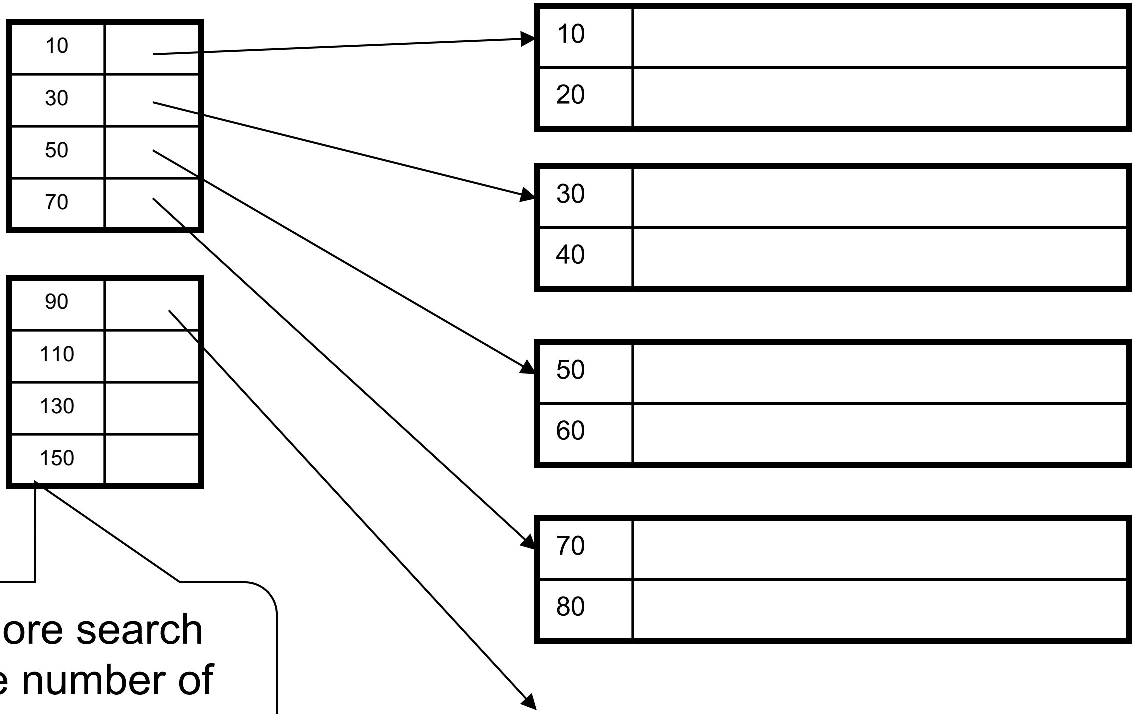
Clustered (aka Primary) Index

- Records in data file have same order as in index
- *Dense* index: sequence of (key,rid) pairs



Clustered (aka Primary) Index

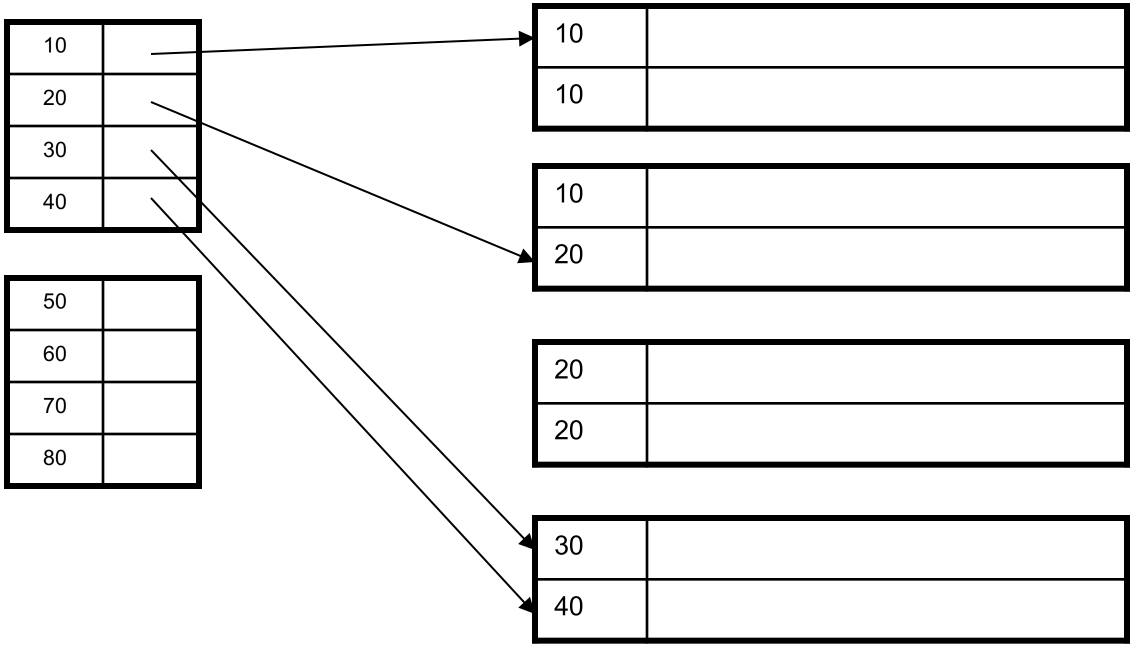
- Records in data file have same order as in index
- *Sparse* index: store a subset of (key,rid) pairs



Can store more search keys in same number of index files

Clustered Index with Duplicate Keys

- Dense index:



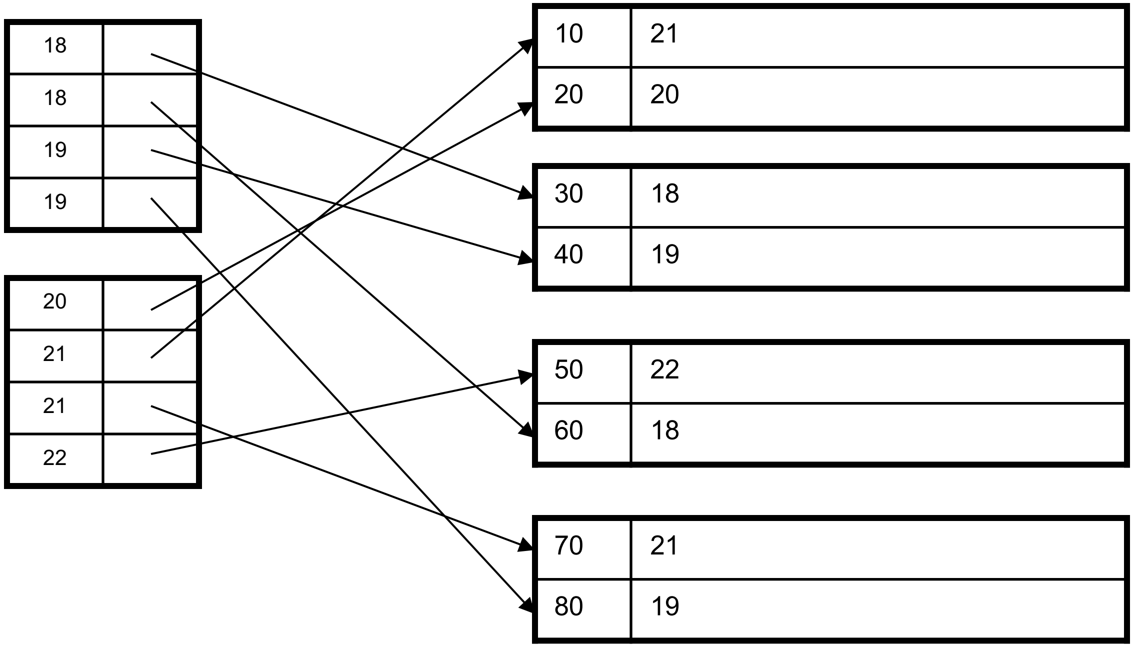
Clustered Index: Back to Example

- Assume entire index fits in main memory

- Find student where sid=12345
  - Index (dense or sparse) points directly to the page
  - Read only 1 page from disk
- Find all students where age > 20
  - Add a second index…

Secondary Indexes

- Do not determine placement of records in data files
- Always dense (why ?)



The Confusing Terminology of Indexes…

- Clustered index:
  - Means: keys close in the index are also close in the data
  - Can co-exists with the data file (quite common)
  - Can have only one clustered index (obviously!!)
  - Sometimes called “primary index”

The Confusing Terminology of Indexes…

- Clustered index:
  - Means: keys close in the index are also close in the data
  - Can co-exists with the data file (quite common)
  - Can have only one clustered index (obviously!!)
  - Sometimes called “primary index”
- Unclustered index:
  - Means: order in the index and order in the data differ
  - Always a separate file
  - Can have as many unclustered indexes as we want
  - Sometimes called “secondary index”

The Confusing Terminology of Indexes…

- Clustered index:
  - Means: keys close in the index are also close in the data
  - Can co-exists with the data file (quite common)
  - Can have only one clustered index (obviously!!)
  - Sometimes called “primary index”
- Unclustered index:
  - Means: order in the index and order in the data differ
  - Always a separate file
  - Can have as many unclustered indexes as we want
  - Sometimes called “secondary index”
- Some people use *different convetion*:
  - Primary index = index on the primary key
  - Secondary index = everything else

Index Organization

- The index is a collection of (key, RID(s)) pairs
- Needs to support efficiently:
  - **Find** the entry where key=[some value]
  - **Insert** a new (key, RID)
  - **Delete** a (key, RID)
- How would you design the index data structure?

Index Organization

- The index is a collection of (key, RID(s)) pairs
- Needs to support efficiently:
  - **Find** the entry where key=[some value]
  - **Insert** a new (key, RID)
  - **Delete** a (key, RID)
- How would you design the index data structure?
  - Ordered file – problem here (why?)

Index Organization

- The index is a collection of (key, RID(s)) pairs
- Needs to support efficiently:
  - **Find** the entry where key=[some value]
  - **Insert** a new (key, RID)
  - **Delete** a (key, RID)
- How would you design the index data structure?
  - Ordered file – problem here (why?)
  - Hash table

Index Organization

- The index is a collection of (key, RID(s)) pairs
- Needs to support efficiently:
  - **Find** the entry where key=[some value]
  - **Insert** a new (key, RID)
  - **Delete** a (key, RID)
- How would you design the index data structure?
  - Ordered file – problem here (why?)
  - Hash table
  - B+ tree

BRIEF Review of Hash Tables

Arrays are *very* efficient:

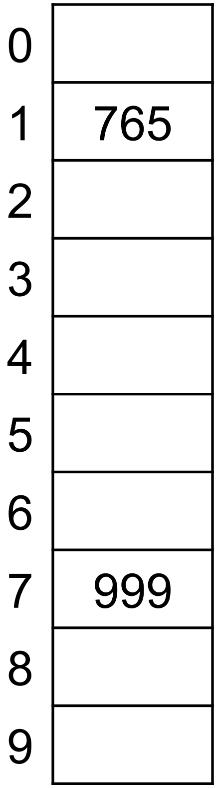
- Find(T[7])

0	
1	765
2	
3	
4	
5	
6	
7	999
8	
9	

BRIEF Review of Hash Tables

Arrays are *very* efficient:

- Find(T[7])
- Set T[3] := 234

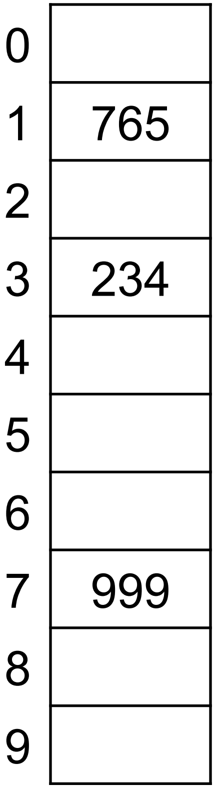


0	
1	765
2	
3	
4	
5	
6	
7	999
8	
9	

BRIEF Review of Hash Tables

Arrays are *very* efficient:

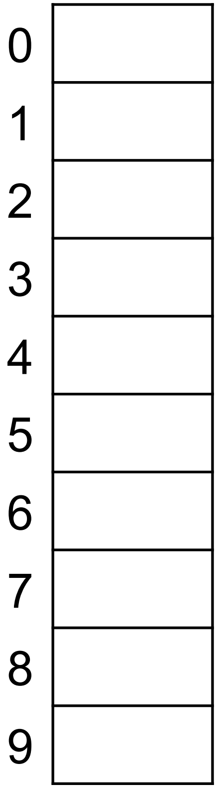
- Find(T[7])
- Set T[3] := 234



0	
1	765
2	
3	234
4	
5	
6	
7	999
8	
9	

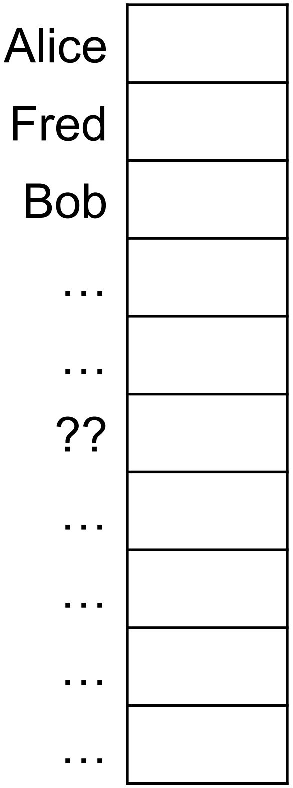
BRIEF Review of Hash Tables

Problem: the key is not 0,1,2,…9 but is a string k



BRIEF Review of Hash Tables

Problem: the key is not 0,1,2,…9 but is a string k

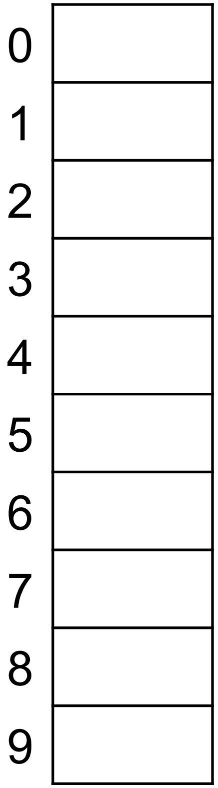


BRIEF Review of Hash Tables

Problem: the key is not 0,1,2,…9 but is a string k

A (naïve) hash function:

$$h(k) = \mathrm{sum}(k) \bmod 10$$

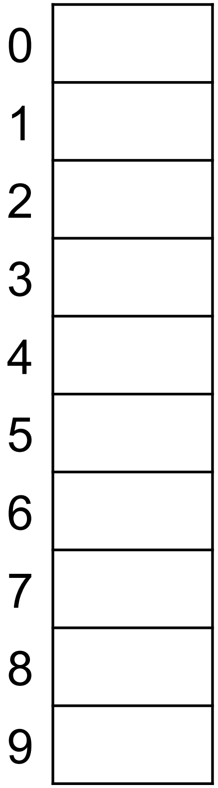


BRIEF Review of Hash Tables

Problem: the key is not 0,1,2,…9 but is a string k

A (naïve) hash function:

h(k) = sum(k) mod 10



Example: h("Fred") =
= (ascii("F")+ascii("r")+…)
mod 10
= (70 + 114 + 101 + 100)
mod 10
= 5

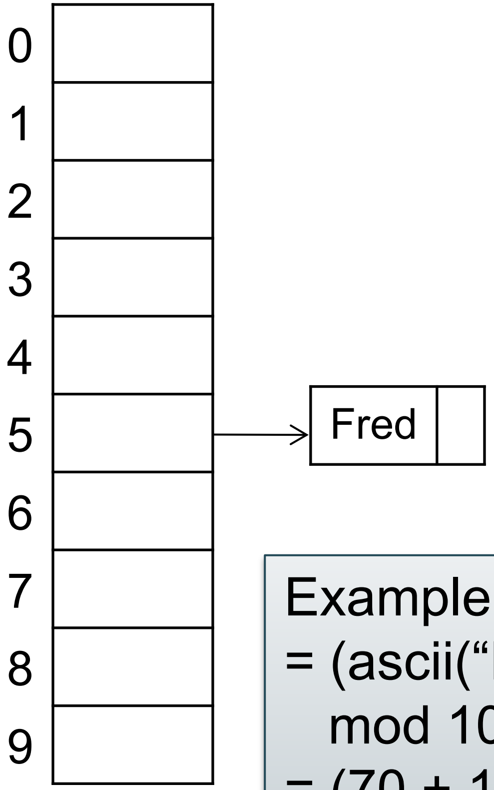
BRIEF Review of Hash Tables

Problem: the key is not 0,1,2,…9 but is a string k

A (naïve) hash function:

h(k) = sum(k) mod 10

Separate chaining:



Example: h("Fred") =
= (ascii("F")+ascii("r")+…)
mod 10
= (70 + 114 + 101 + 100)
mod 10
= 5

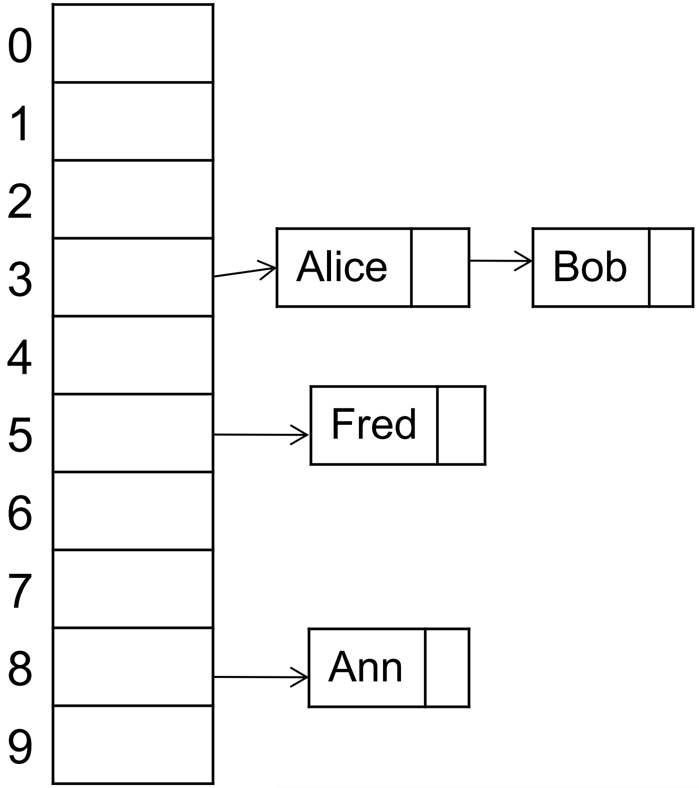
BRIEF Review of Hash Tables

Problem: the key is not 0,1,2,…9 but is a string k

A (naïve) hash function:

h(k) = sum(k) mod 10

Separate chaining:



h("Alice") = h("Bob") = 3
Called *collisions*

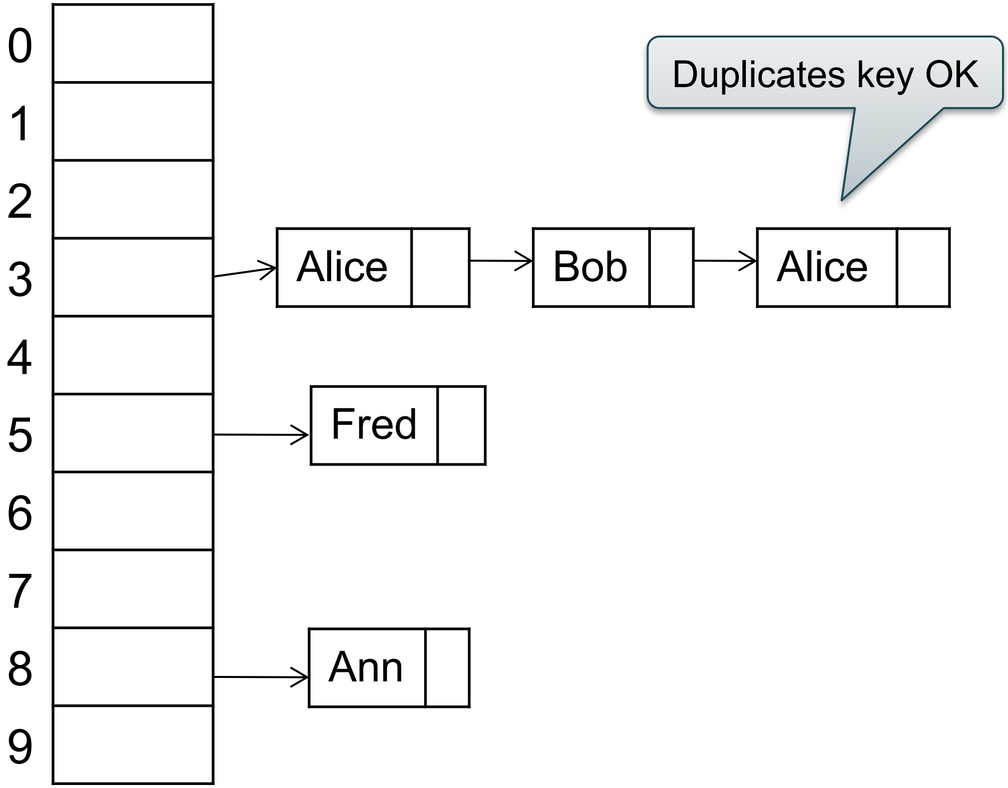
BRIEF Review of Hash Tables

Problem: the key is not 0,1,2,…9 but is a string k

A (naïve) hash function:

h(k) = sum(k) mod 10

Separate chaining:



Duplicates key OK

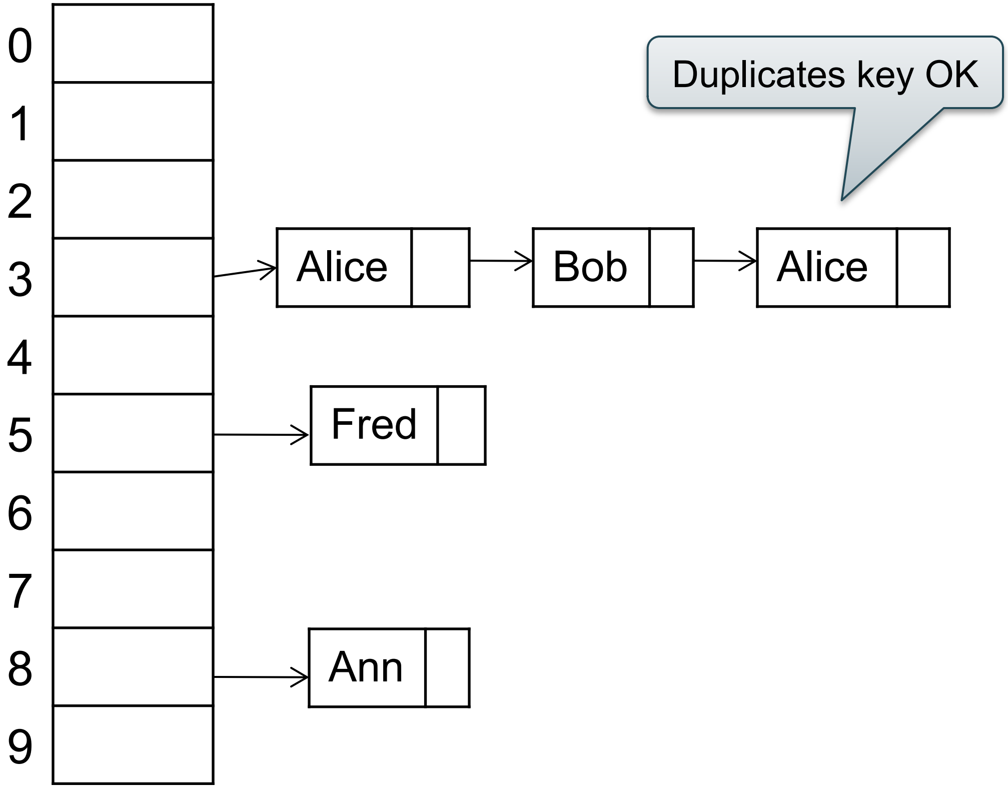
BRIEF Review of Hash Tables

Problem: the key is not 0,1,2,…9 but is a string k

A (naïve) hash function:

h(k) = sum(k) mod 10

Separate chaining:



Duplicates key OK

Operations:

find(Bob) = ??

BRIEF Review of Hash Tables

Problem: the key is not 0,1,2,…9 but is a string k

A (naïve) hash function:

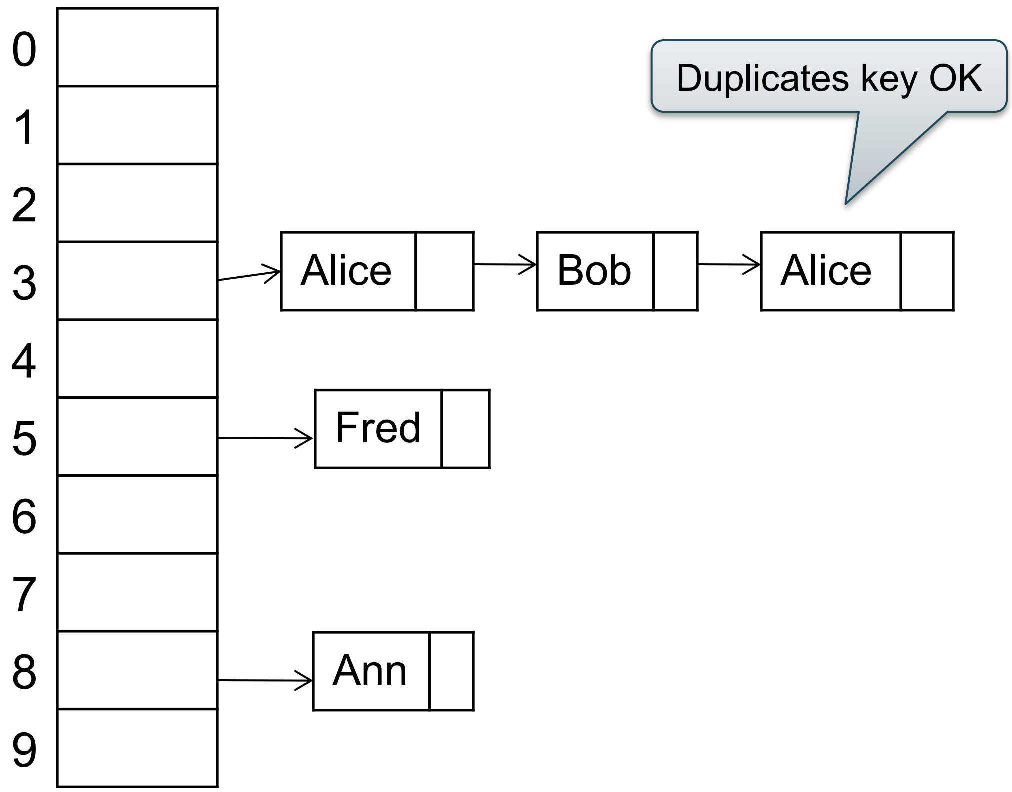
$h(k) = sum(k) \bmod 10$

Operations:

find(Bob) = ??

insert(Jon) = ??

Separate chaining:



BRIEF Review of Hash Tables

Problem: the key is not 0,1,2,…9 but is a string k

A (naïve) hash function:

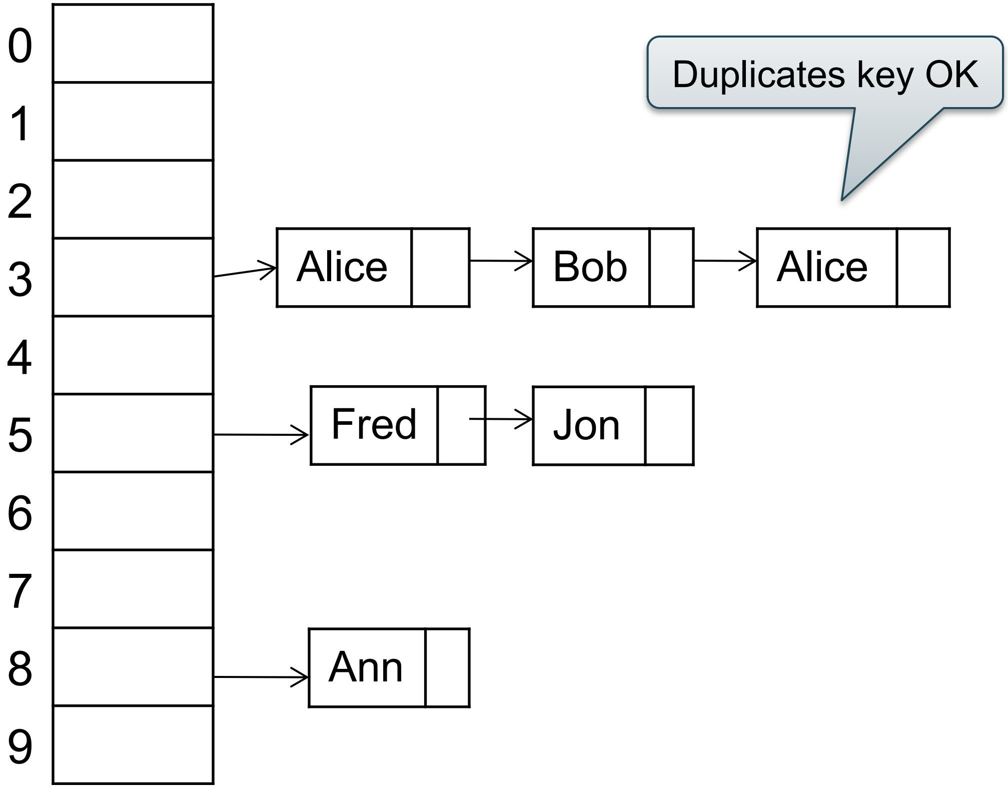
h(k) = sum(k) mod 10

Operations:

find(Bob) = ??

insert(Jon) = ??

Separate chaining:



Duplicates key OK

BRIEF Review of Hash Tables

Problem: the key is not 0,1,2,…9 but is a string k

A (naïve) hash function:

h(k) = sum(k) mod 10

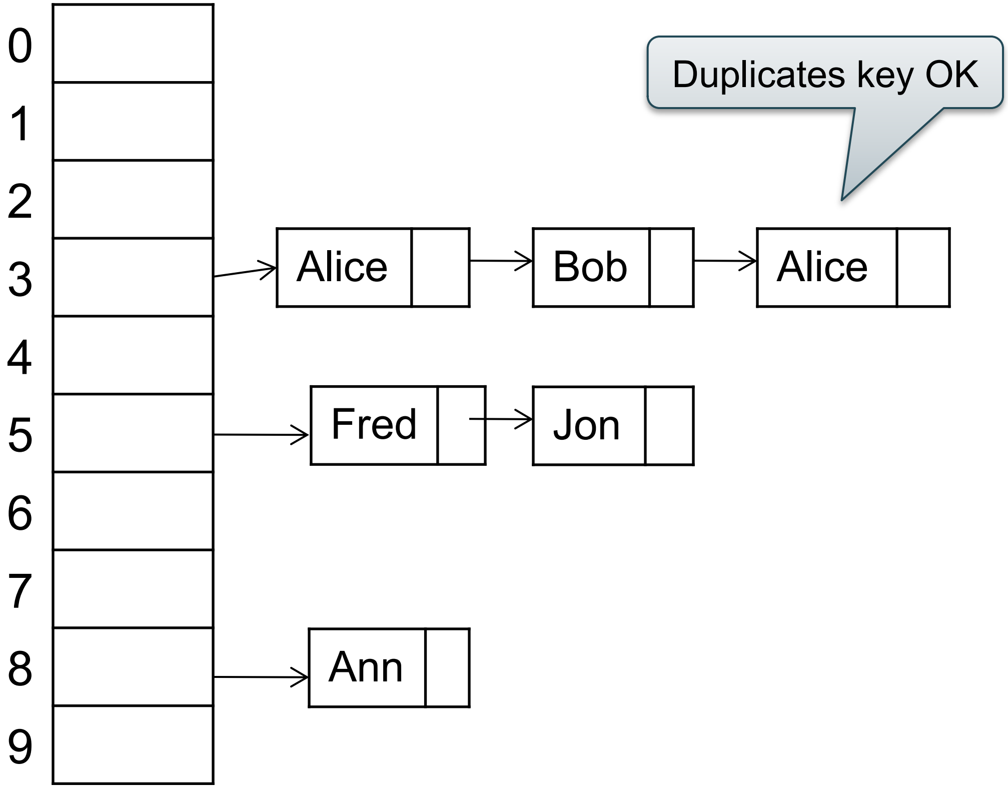
Operations:

find(Bob) = ??

insert(Jon) = ??

delete(Ann) = ??

Separate chaining:



BRIEF Review of Hash Tables

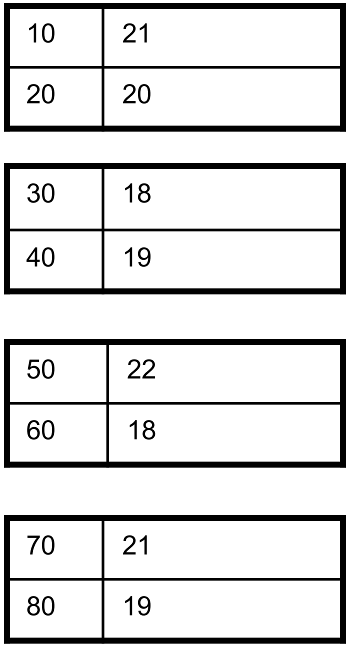
- insert(k, v) = inserts a key k with value v
  - Duplicate k's may be OK or may not be OK
- find(k) = returns the value v associated to k, or the ***list*** of all values associated to k
- delete(k)

Discussion of Hash Tables

- Hash function:
  - Should distribute values uniformly
  - Never write your own! (why is x mod 10 bad?) Use a standard library function
  - Best: concatenate with fixed, random seed (in class)
- Hash table:
  - Size of table: large enough to avoid collisions
  - Typically: size of table ≈ size of data
  - Why not make it small? Why not make it big?
  - Problem: hash table allocated statically, at creation
  - Book describes solutions to increase size dynamically

Hash-Based Index

Good for point queries but not range queries



10	21
20	20

30	18
40	19

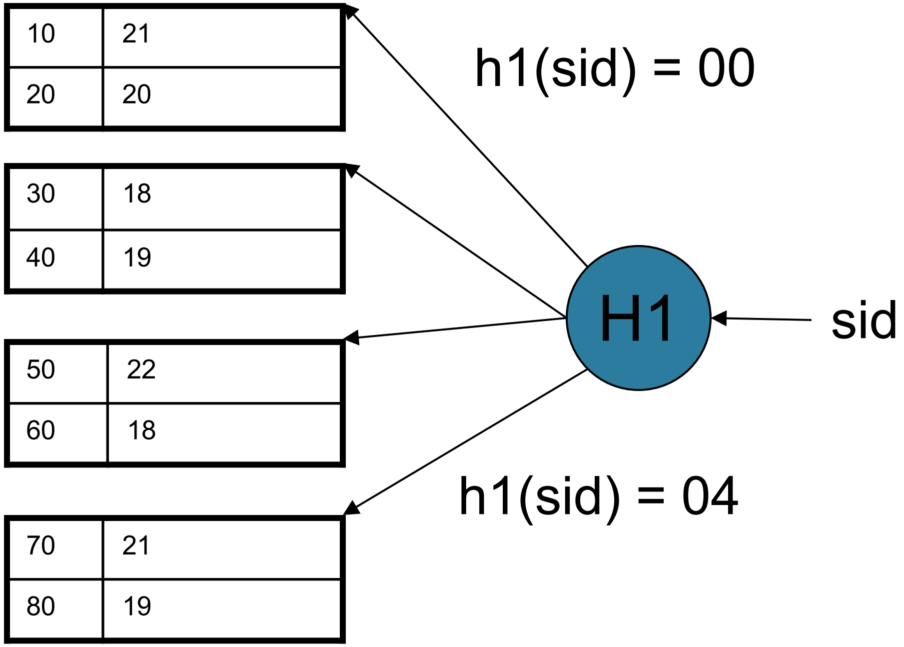
50	22
60	18

70	21
80	19

Data File

Hash-Based Index

Good for point queries but not range queries



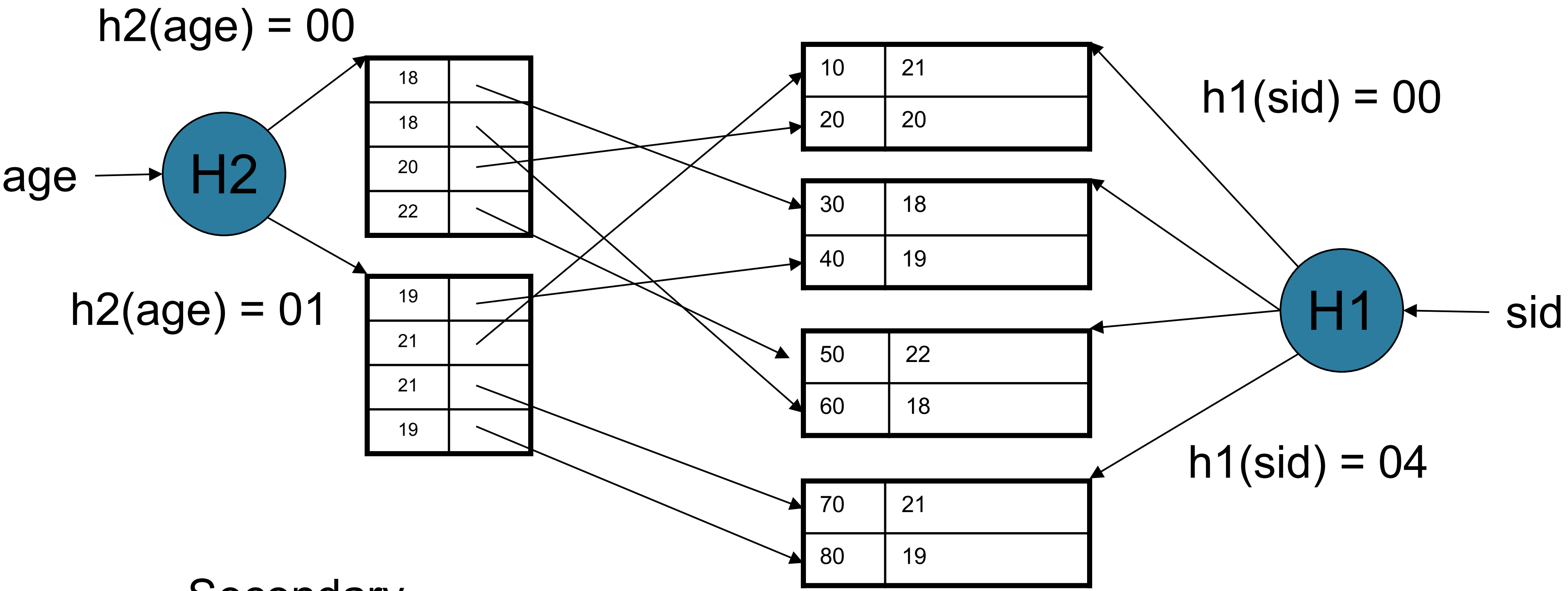
10	21
20	20
30	18
40	19
50	22
60	18
70	21
80	19

Data File

Primary hash-based index

Hash-Based Index

Good for point queries but not range queries



Secondary
hash-based index
(age, rid) pairs

Data File
Primary hash-based index

B+ Trees

- Search trees (quick review in class)

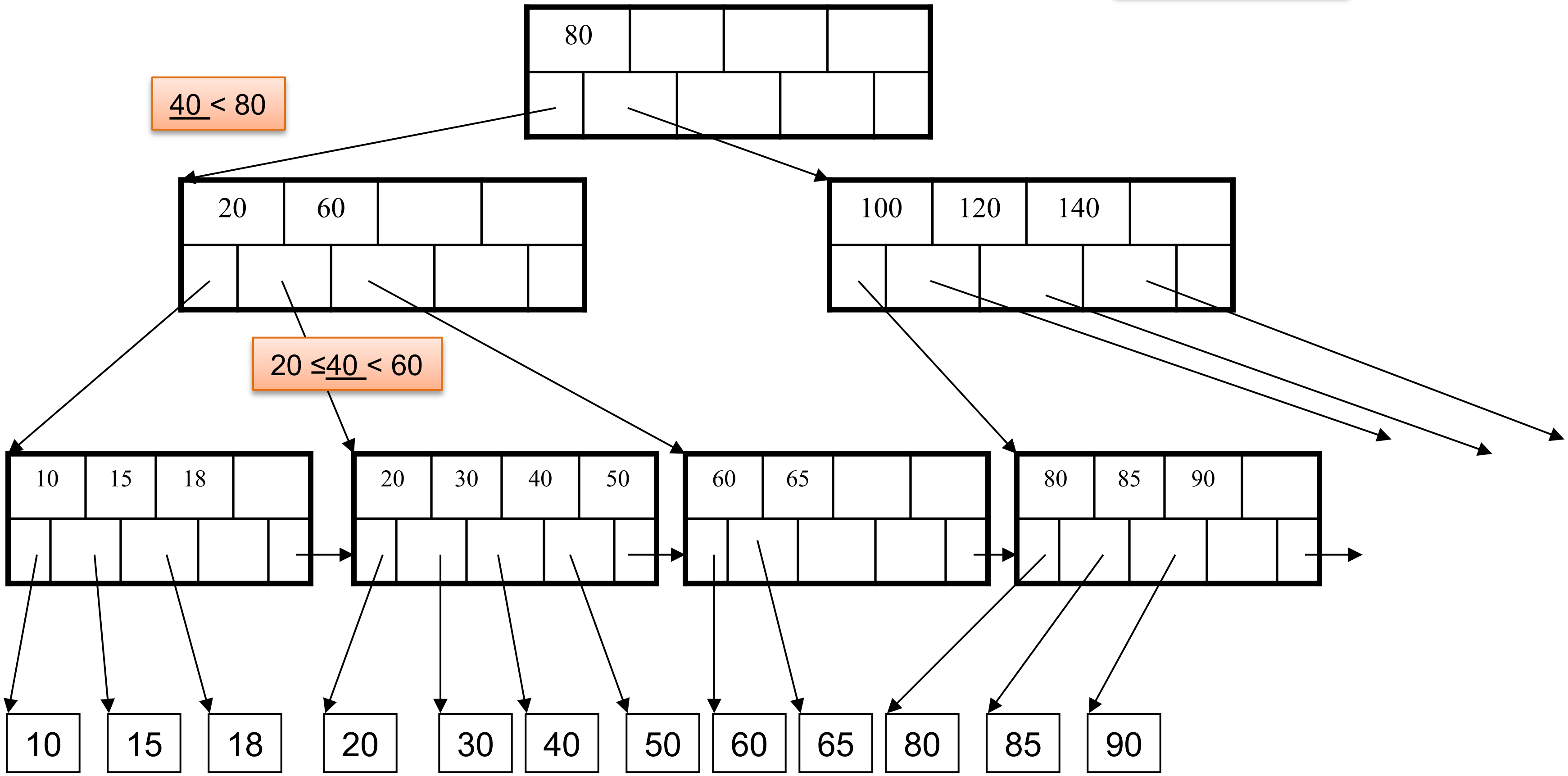
- Idea in B Trees
  - Make 1 node = 1 page (= 1 block)

- Idea in B+ Trees
  - Keys are stored on the leaves (not internal nodes)
  - Leaves are linked in a list, for range queries

B+ Tree Example

d = 2

Find the key 40



B+ Trees Properties

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

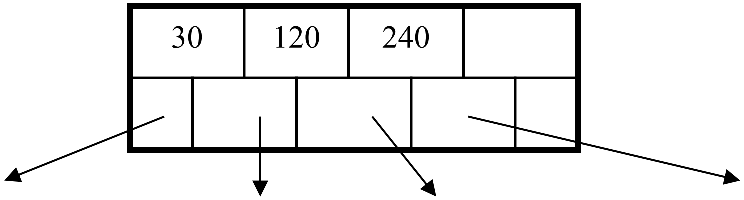
B+ Trees Details

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

B+ Trees Details

- Parameter d = the *degree*
- Each node has **d <= m <= 2d keys** (except root)
- Each node also has **m+1 pointers**

Left pointer of k:
to keys < k



Right pointer of k:
to keys >= k

Keys k < 30

Keys 30<=k<120

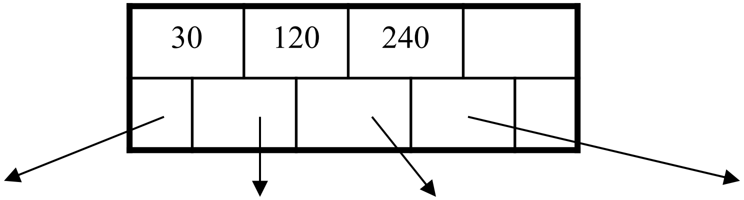
Keys 120<=k<240

Keys 240<=k

B+ Trees Details

- Parameter d = the *degree*
- Each node has **d <= m <= 2d keys** (except root)
- Each node also has **m+1 pointers**

Left pointer of k:
to keys < k



Right pointer of k:
to keys >= k

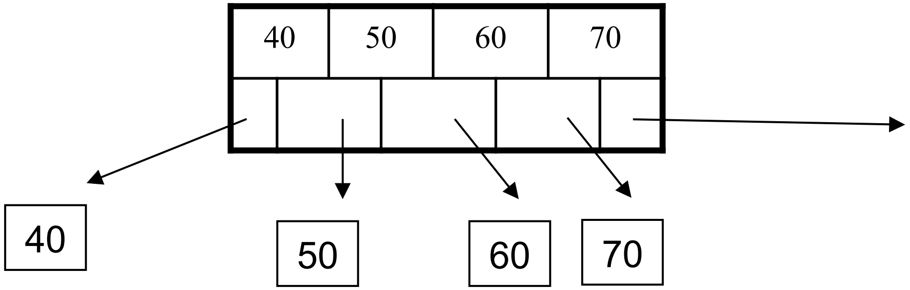
Keys k < 30

Keys 30<=k<120

Keys 120<=k<240

Keys 240<=k

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

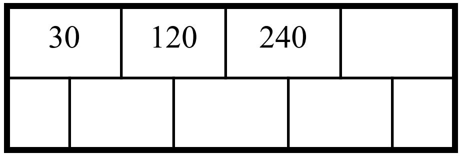


Next leaf

Data records

B+ Tree Design

- How large d ? Make one node fit on one block



(e.g. d = 2)

- Example:
  - Key size = 4 bytes
  - Pointer size = 8 bytes
  - Block size = 4096 bytes
- 2d x 4 + (2d+1) x 8 <= 4096
- d = 170

B+ Trees in Practice

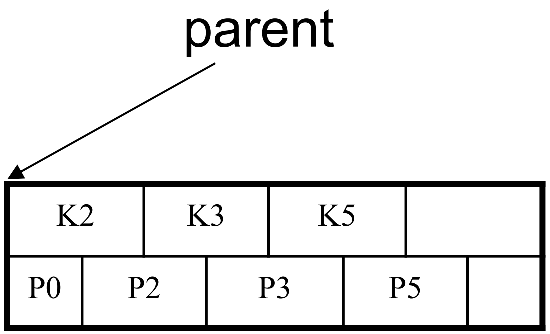
- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities
  - Height 4: 133^4 = 312,900,700 records
  - Height 3: 133^3 = 2,352,637 records
- Can often hold top levels in buffer pool
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 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

Insert k1

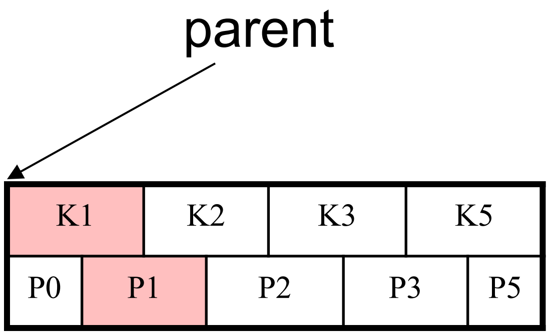


Insertion in a B+ Tree

Insert (K, P)

- Find leaf where K belongs, insert
- If no overflow (2d keys or less), halt

Insert k1

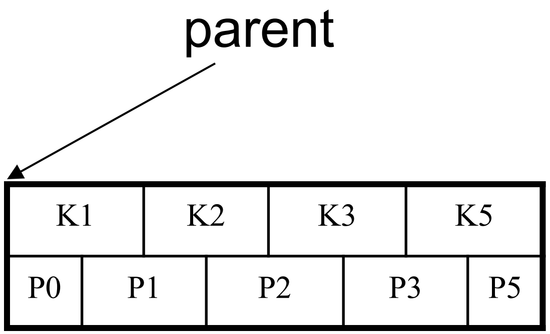


Insertion in a B+ Tree

Insert (K, P)

- Find leaf where K belongs, insert
- If no overflow (2d keys or less), halt

Insert k4



parent

K1 K2 K3 K5

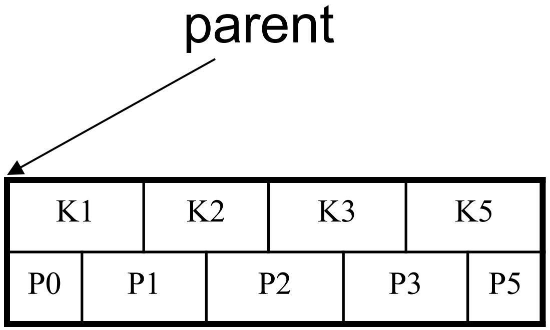
P0 P1 P2 P3 P5

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:

Insert k4



parent

K1 K2 K3 K5

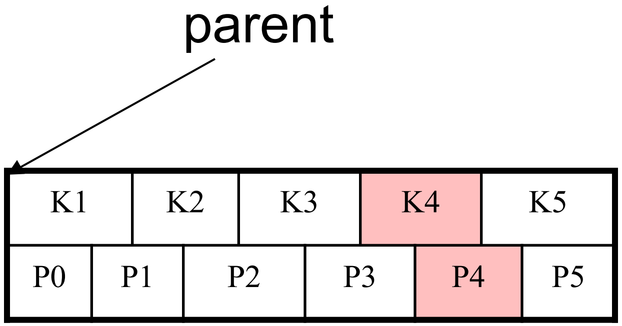
P0 P1 P2 P3 P5

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:

Insert k4



parent

K1 K2 K3 K4 K5

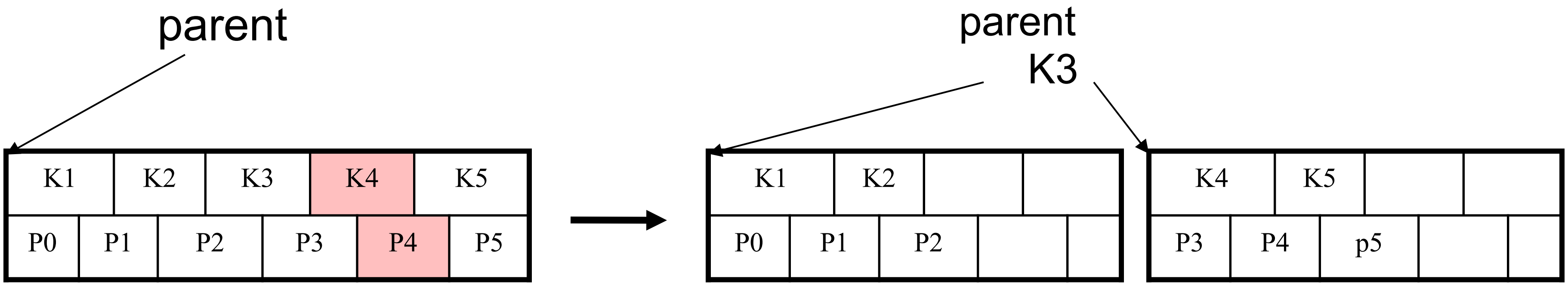
P0 P1 P2 P3 P4 P5

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:

Insert k4

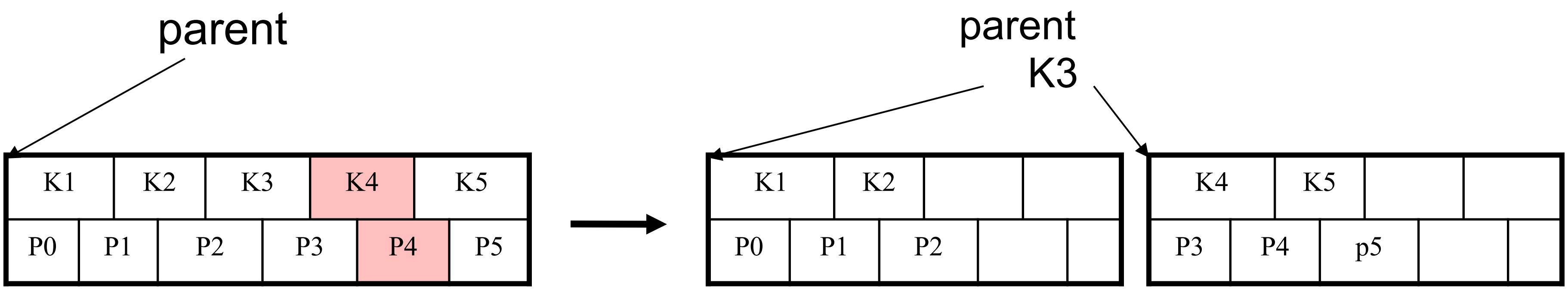


Insertion in a B+ Tree

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:

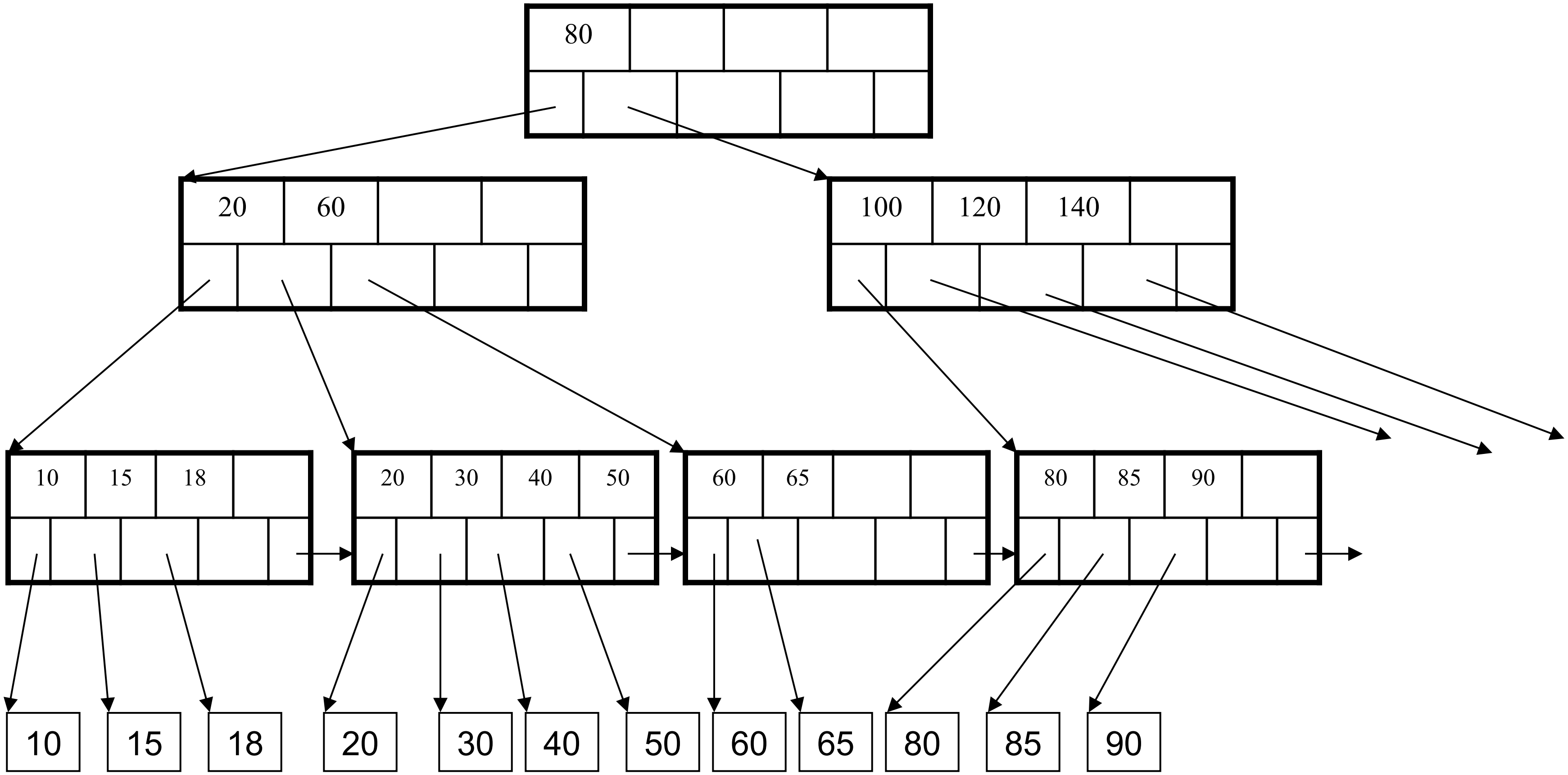
Insert k4



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

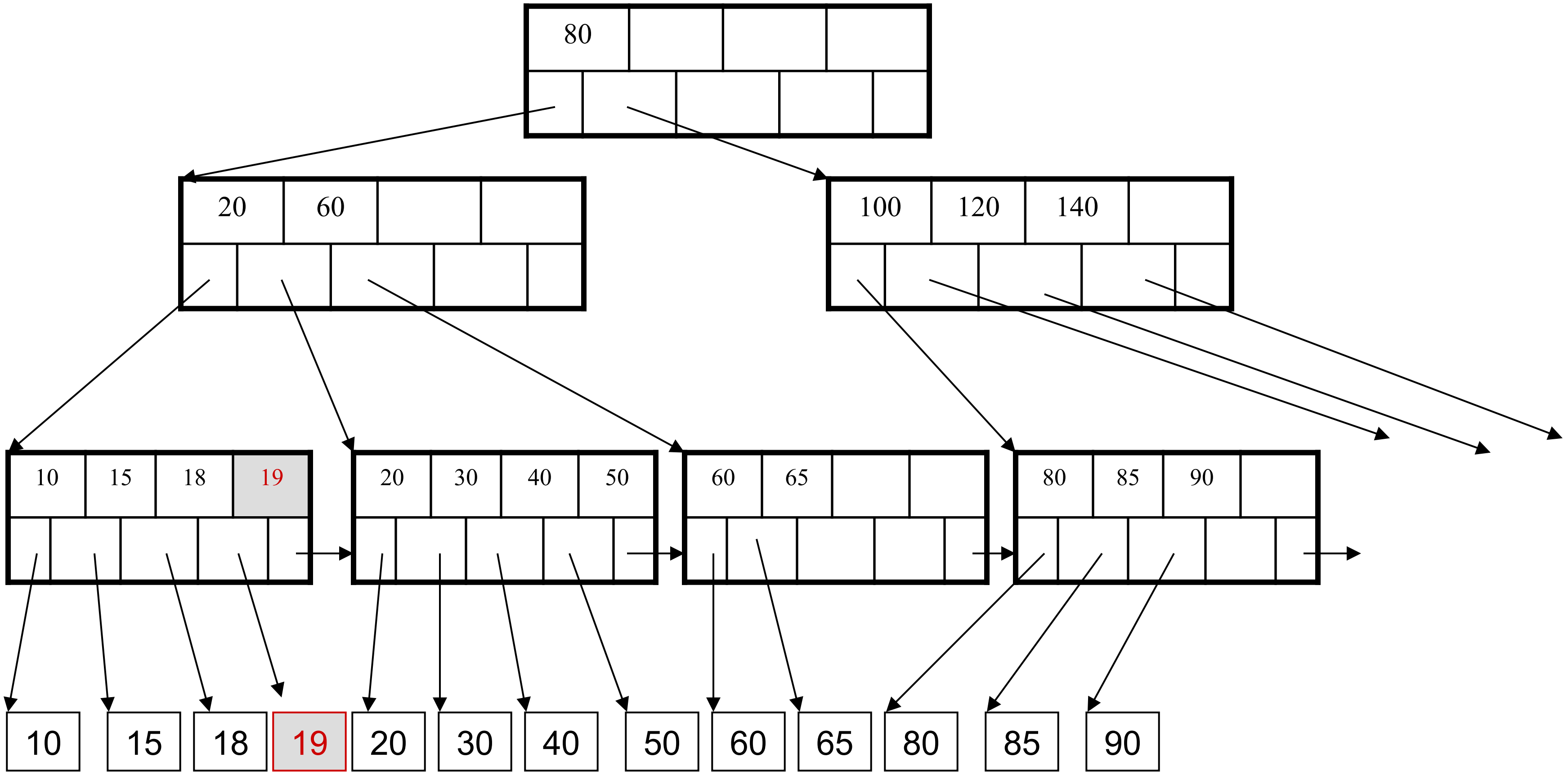
Insertion in a B+ Tree

Insert K=19



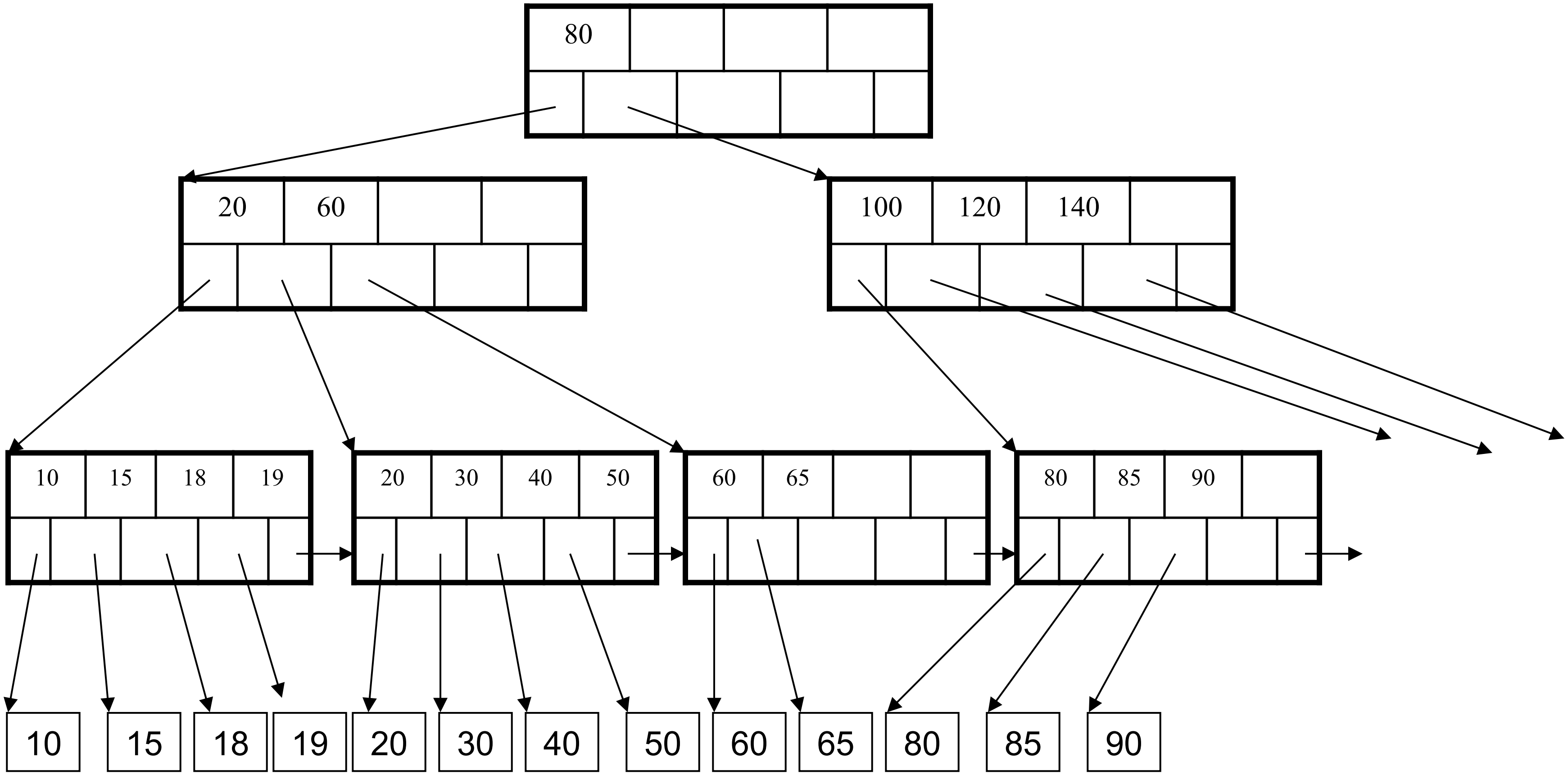
Insertion in a B+ Tree

After insertion



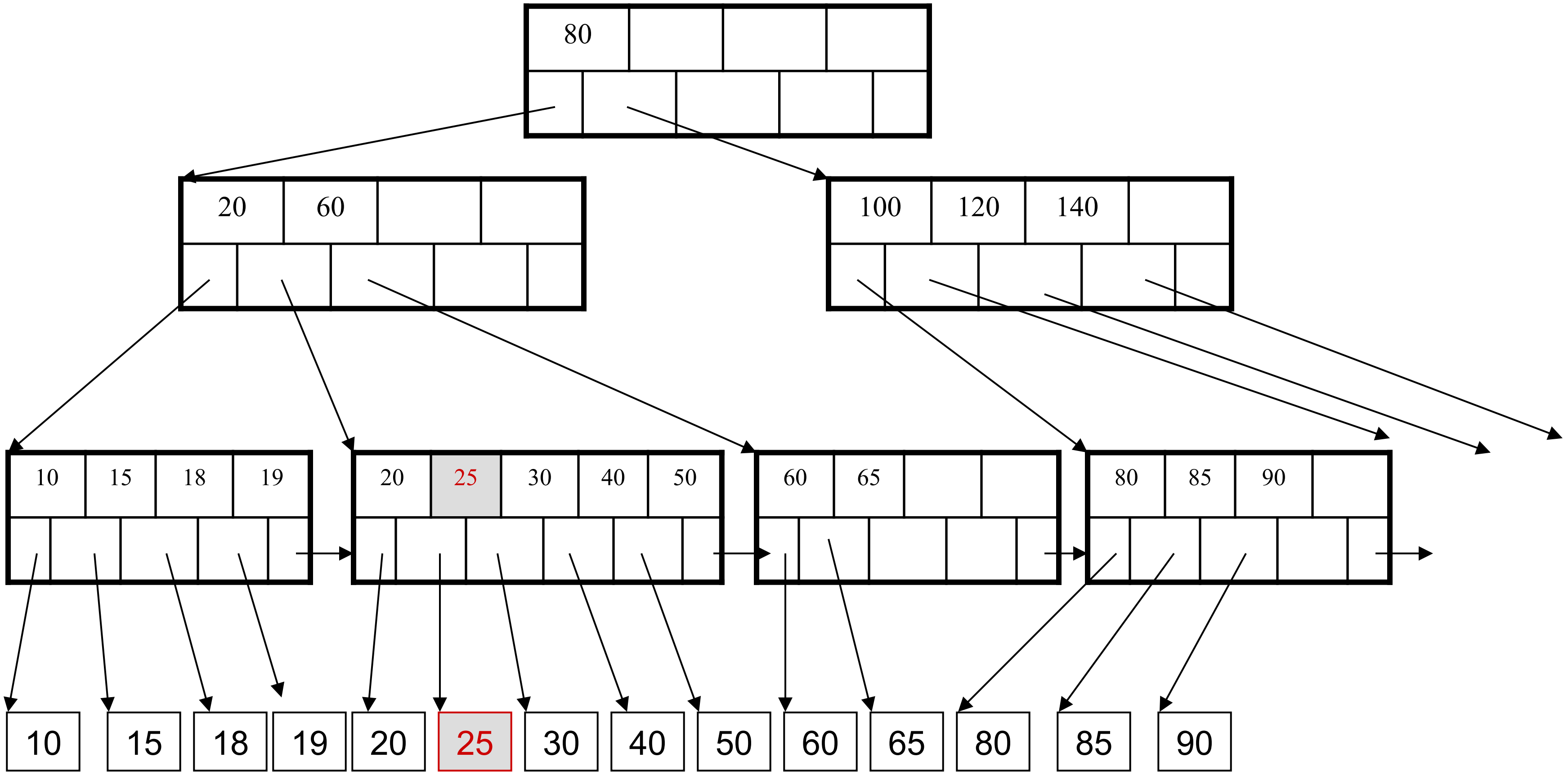
Insertion in a B+ Tree

Now insert 25



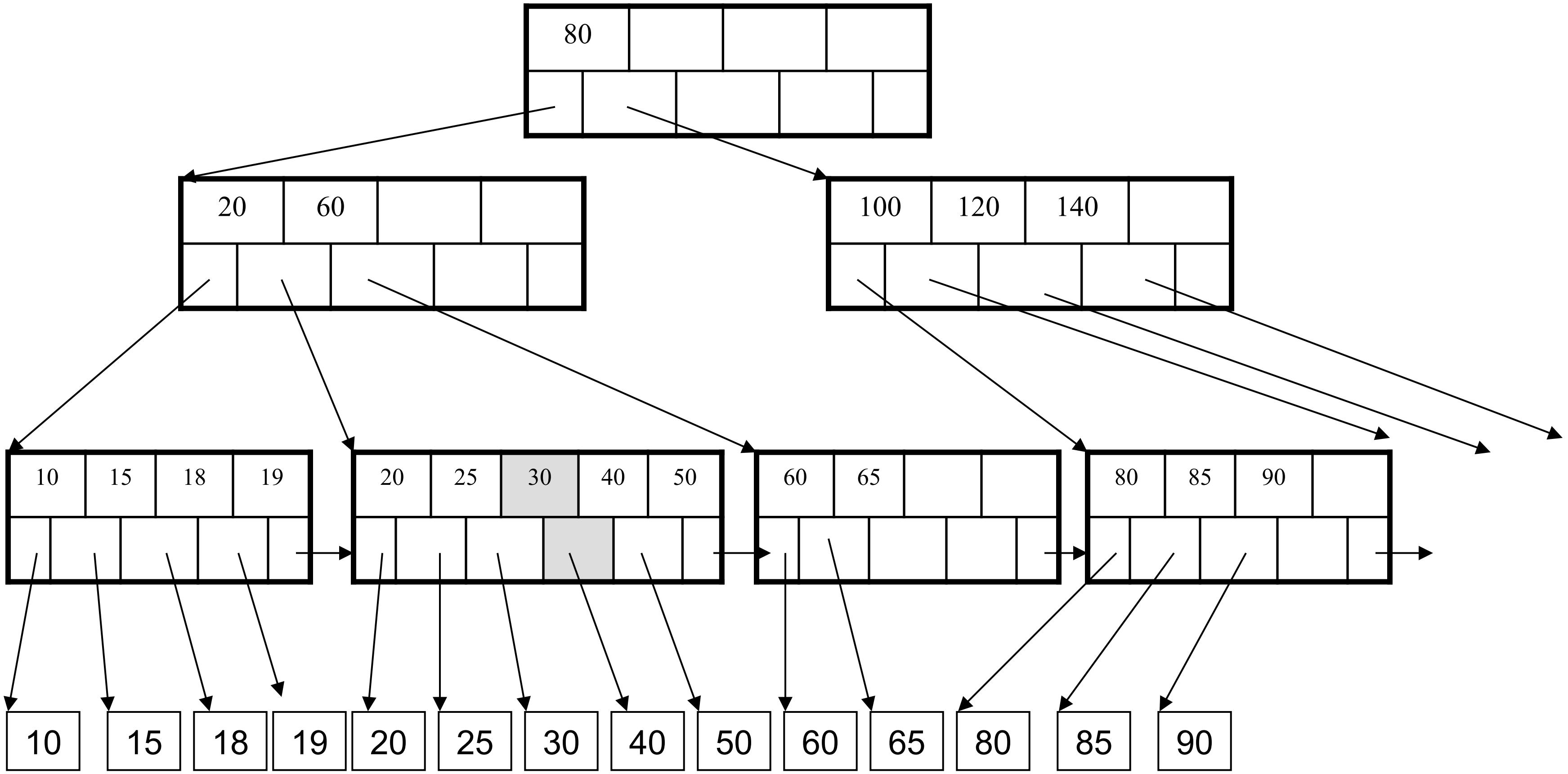
Insertion in a B+ Tree

After insertion



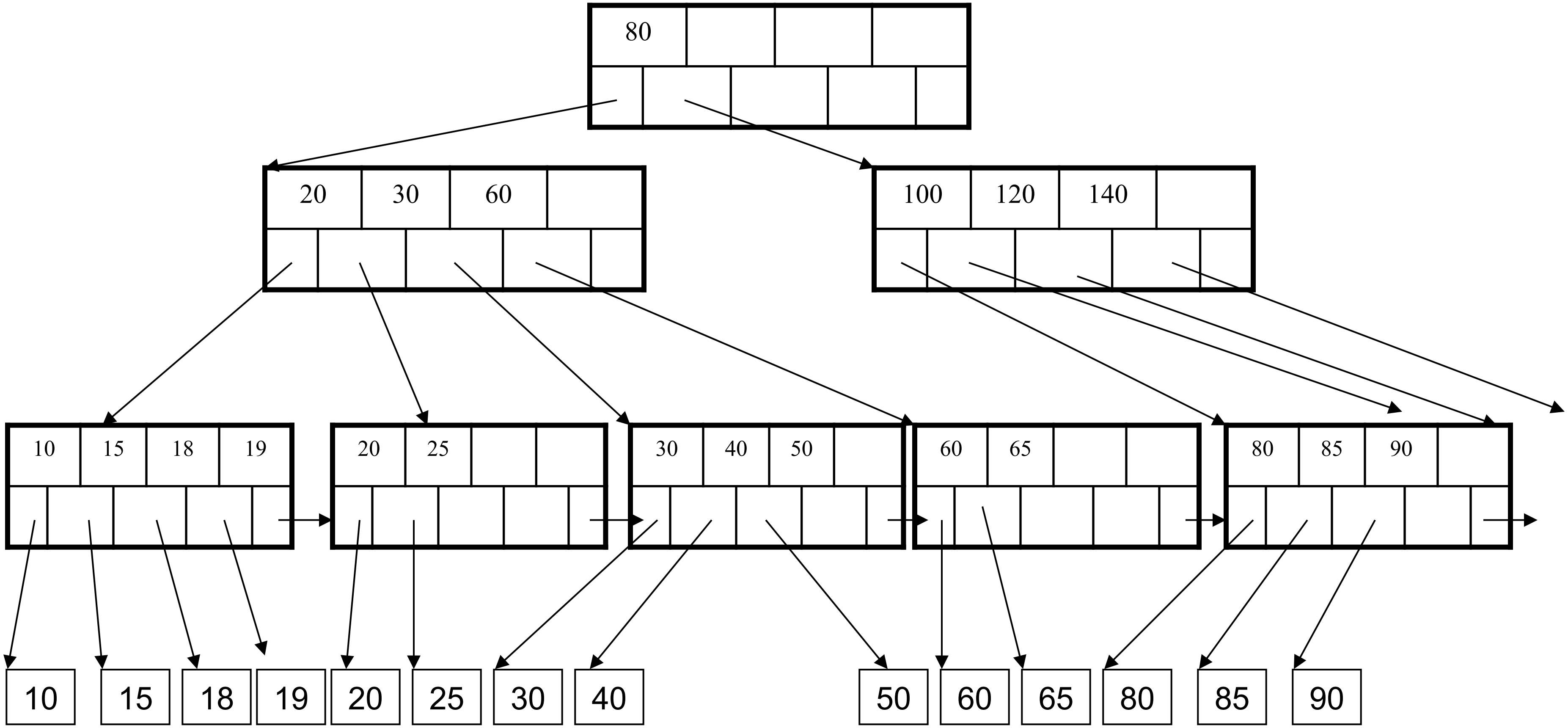
Insertion in a B+ Tree

But now have to split !



Insertion in a B+ Tree

After the split



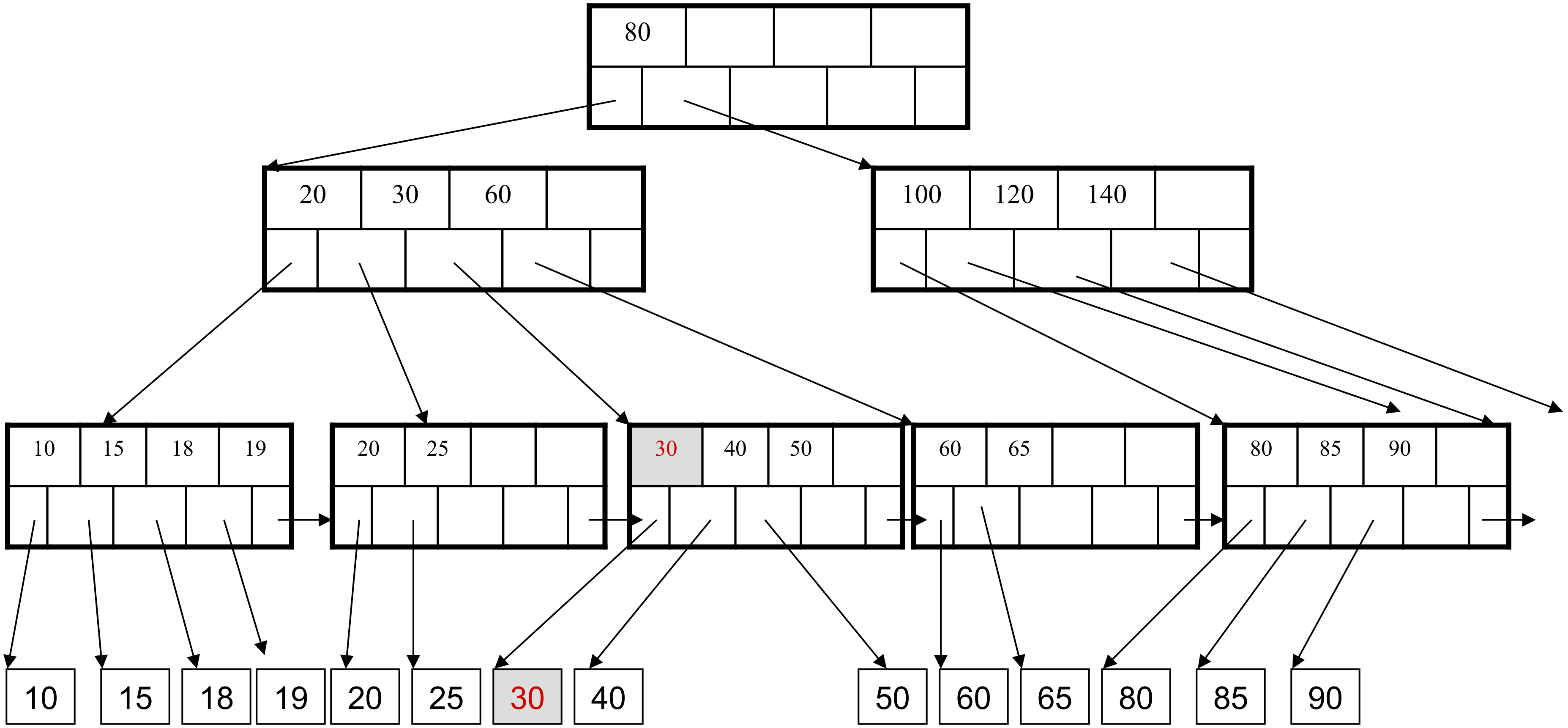
Deletion in a B+ Tree

Delete (K, P)

- Find leaf node where K belongs, delete
- Check for capacity; if above min capacity: **Stop**
- If node below capacity, search adjacent nodes (left, then right) for extra key and rotate key(s) to current node. **Stop**
- If adjacent nodes 50% full, merge with on adjacent node
  This removes a key/child from parent;
  Repeat algorithm on parent node

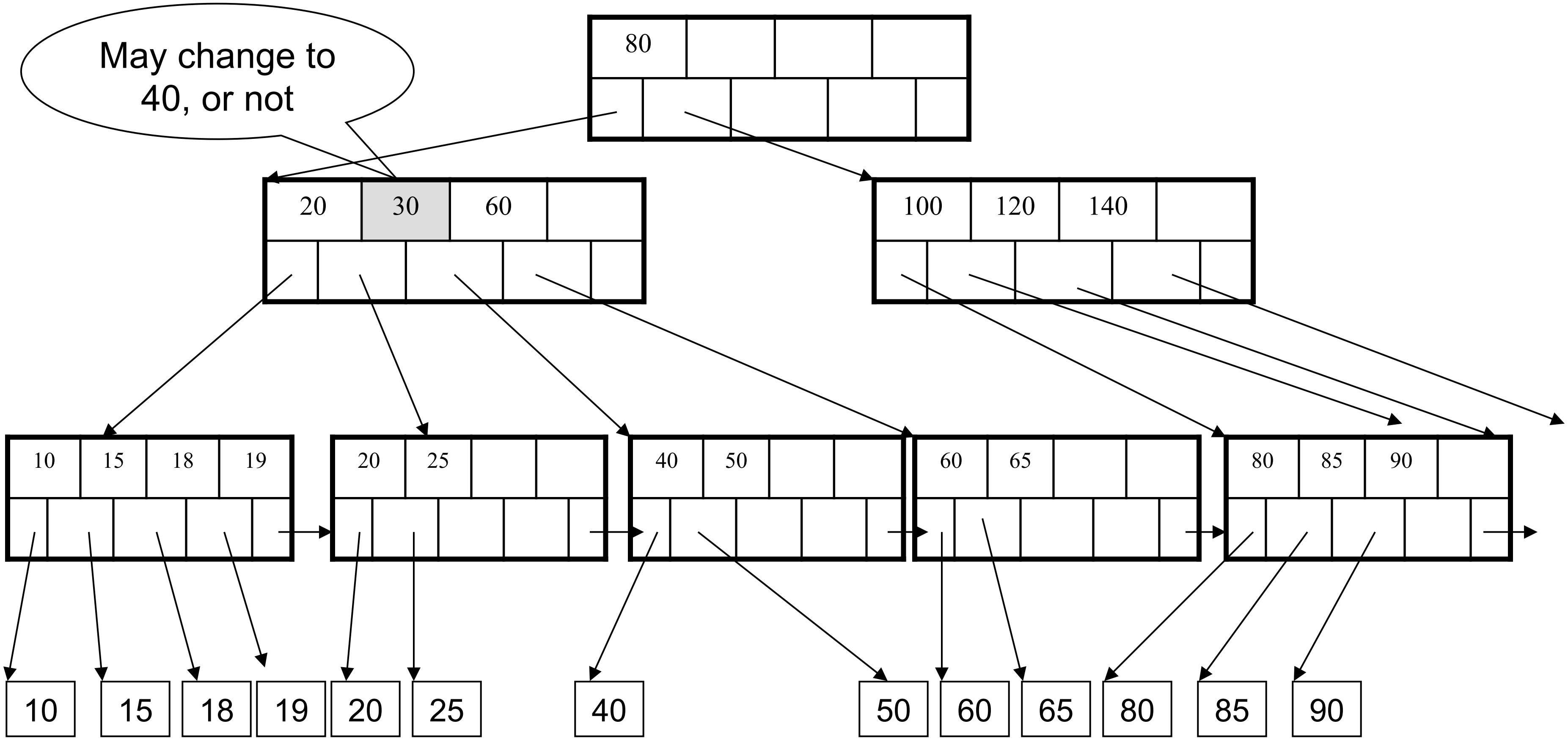
Deletion from a B+ Tree

Delete 30



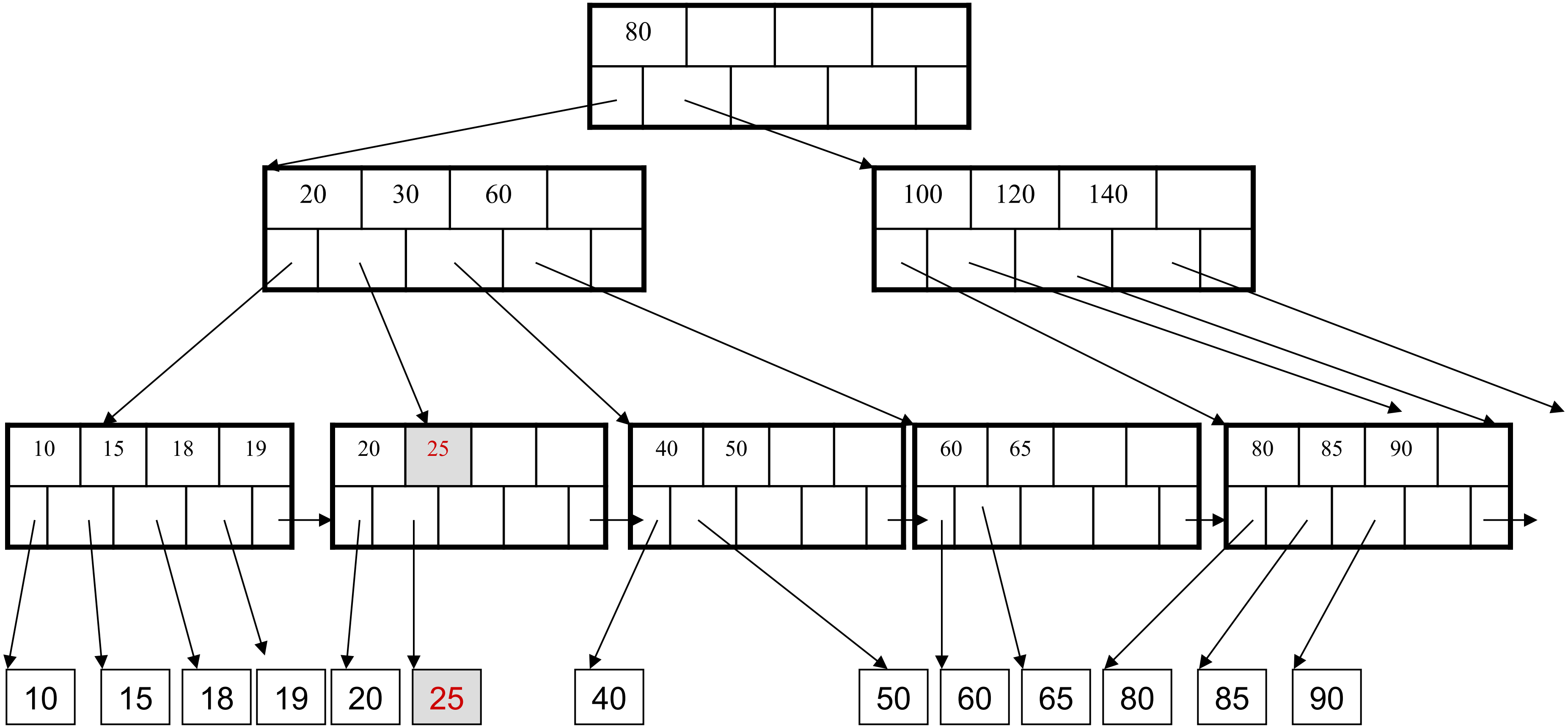
Deletion from a B+ Tree

After deleting 30



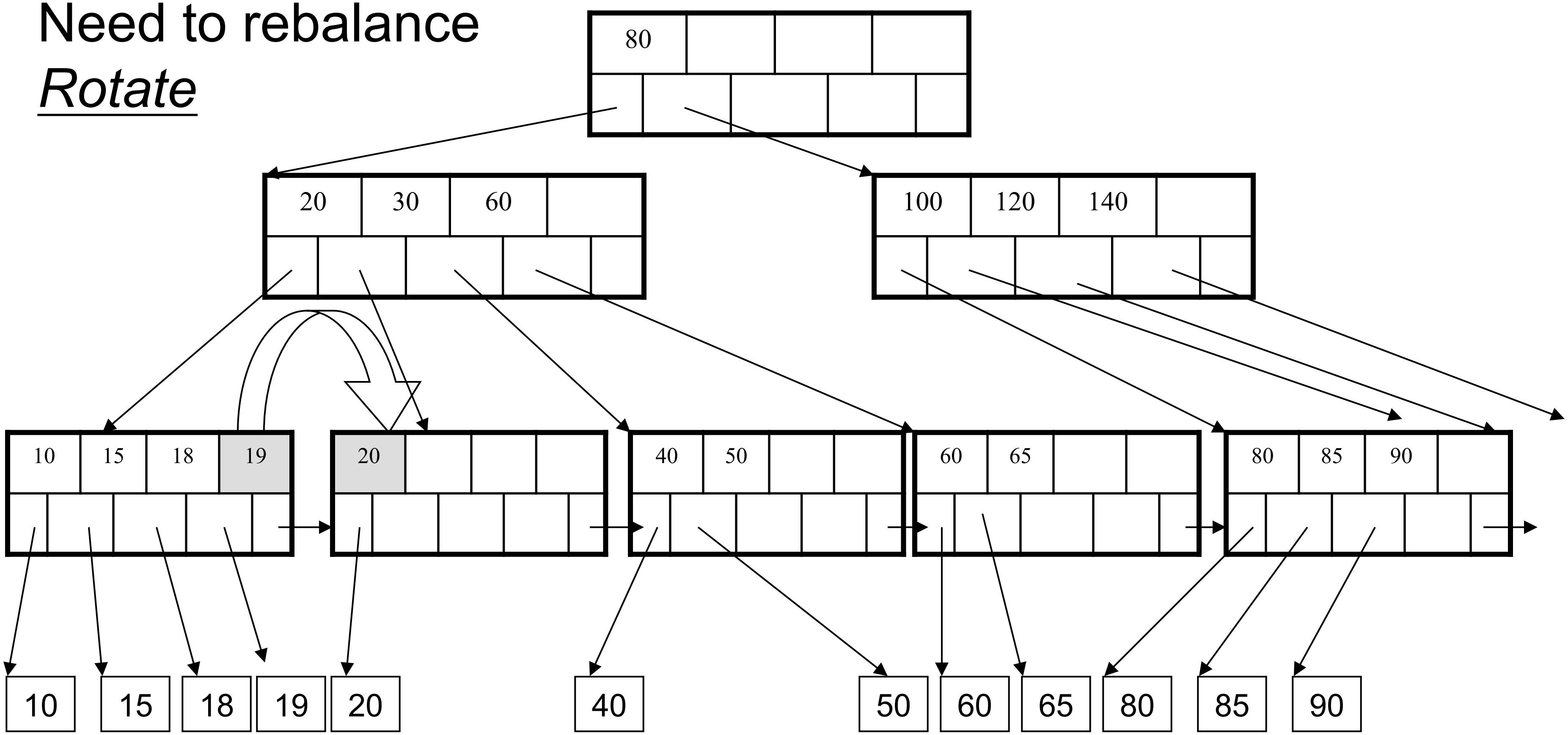
Deletion from a B+ Tree

Now delete 25

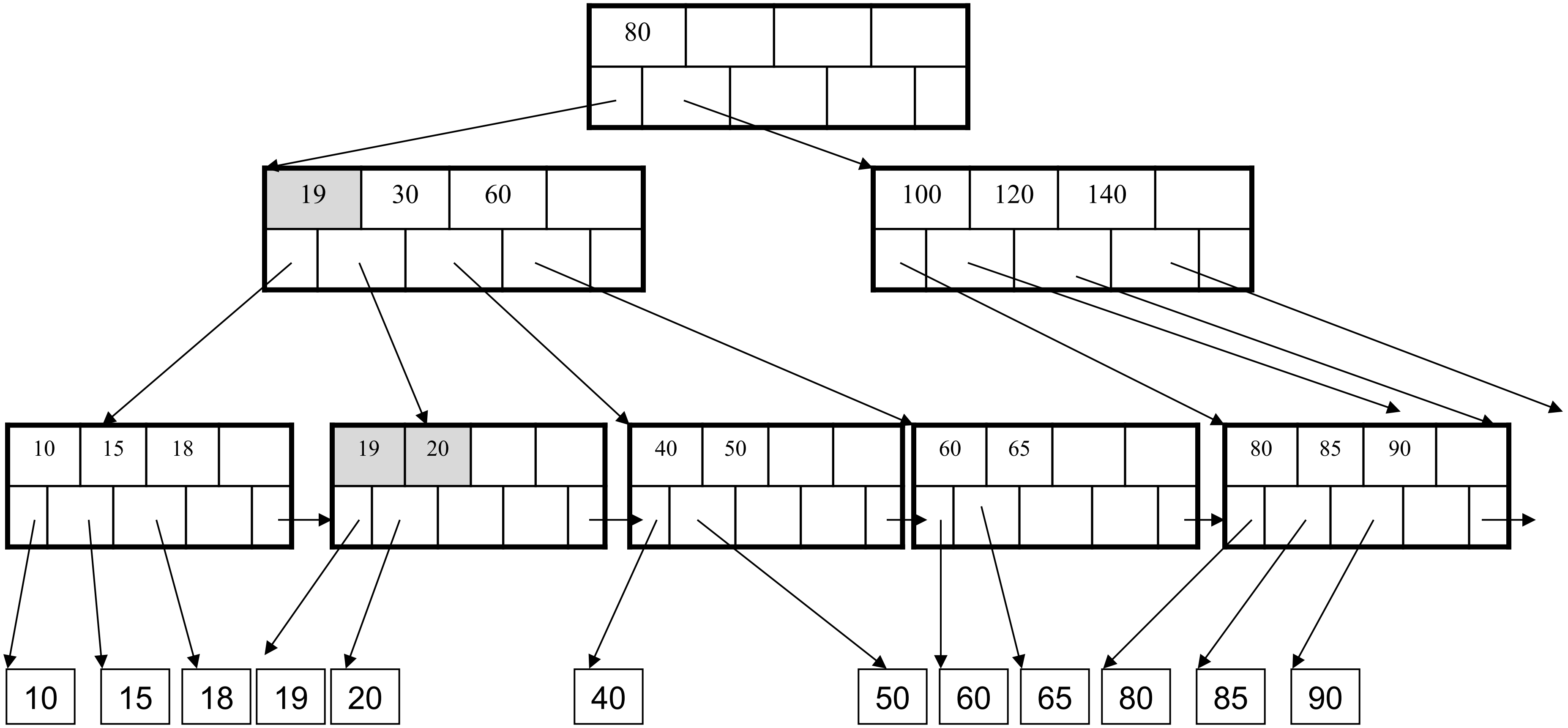


Deletion from a B+ Tree

After deleting 25
Need to rebalance
Rotate

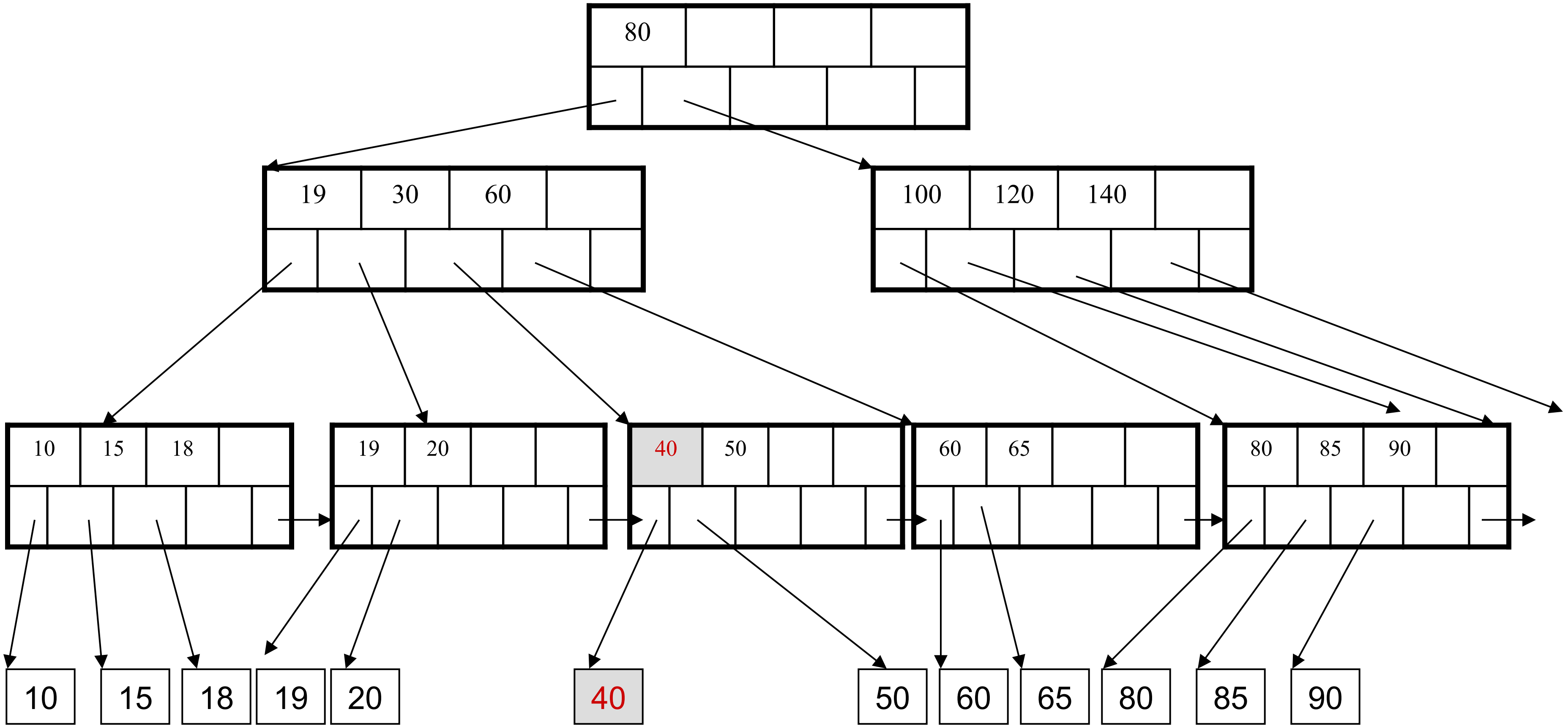


Deletion from a B+ Tree



Deletion from a B+ Tree

Now delete 40

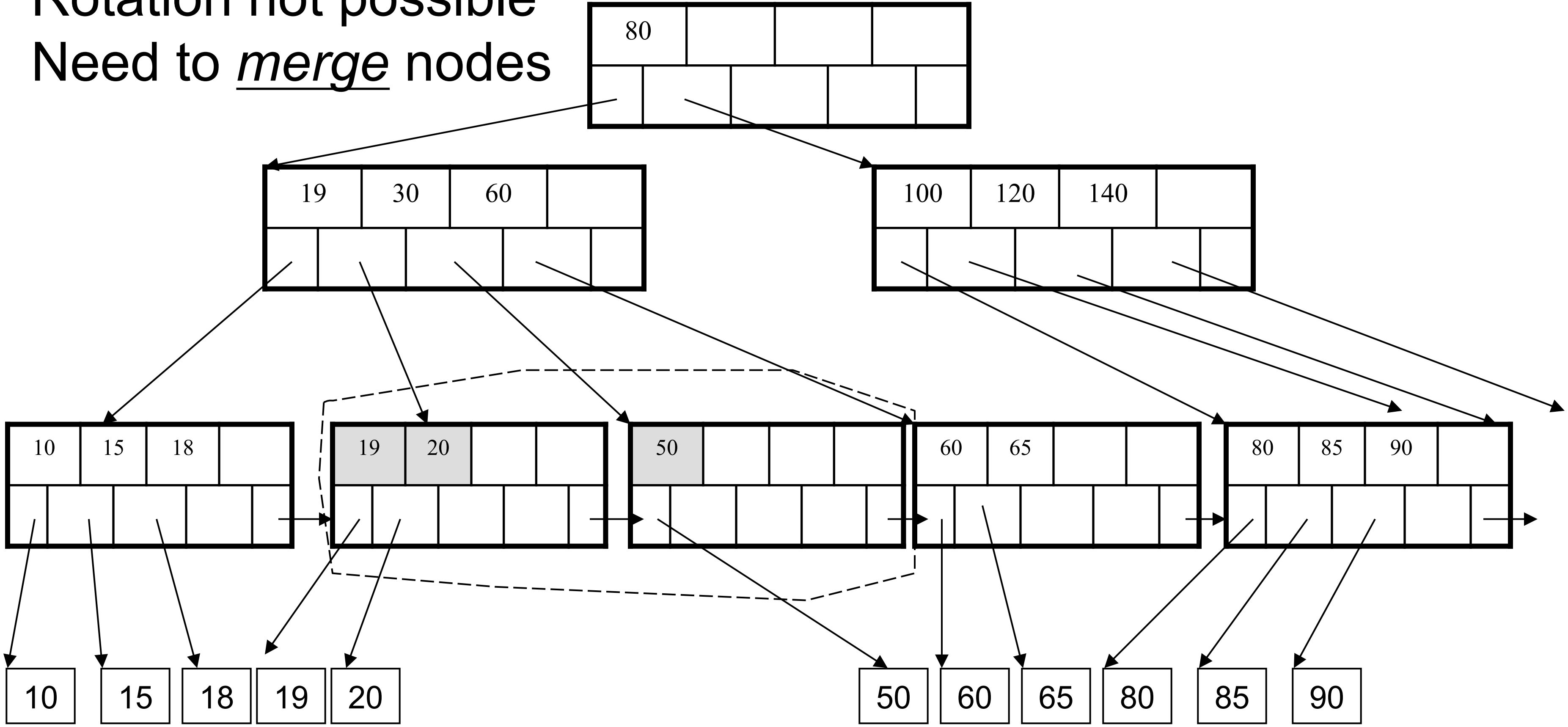


Deletion from a B+ Tree

After deleting 40

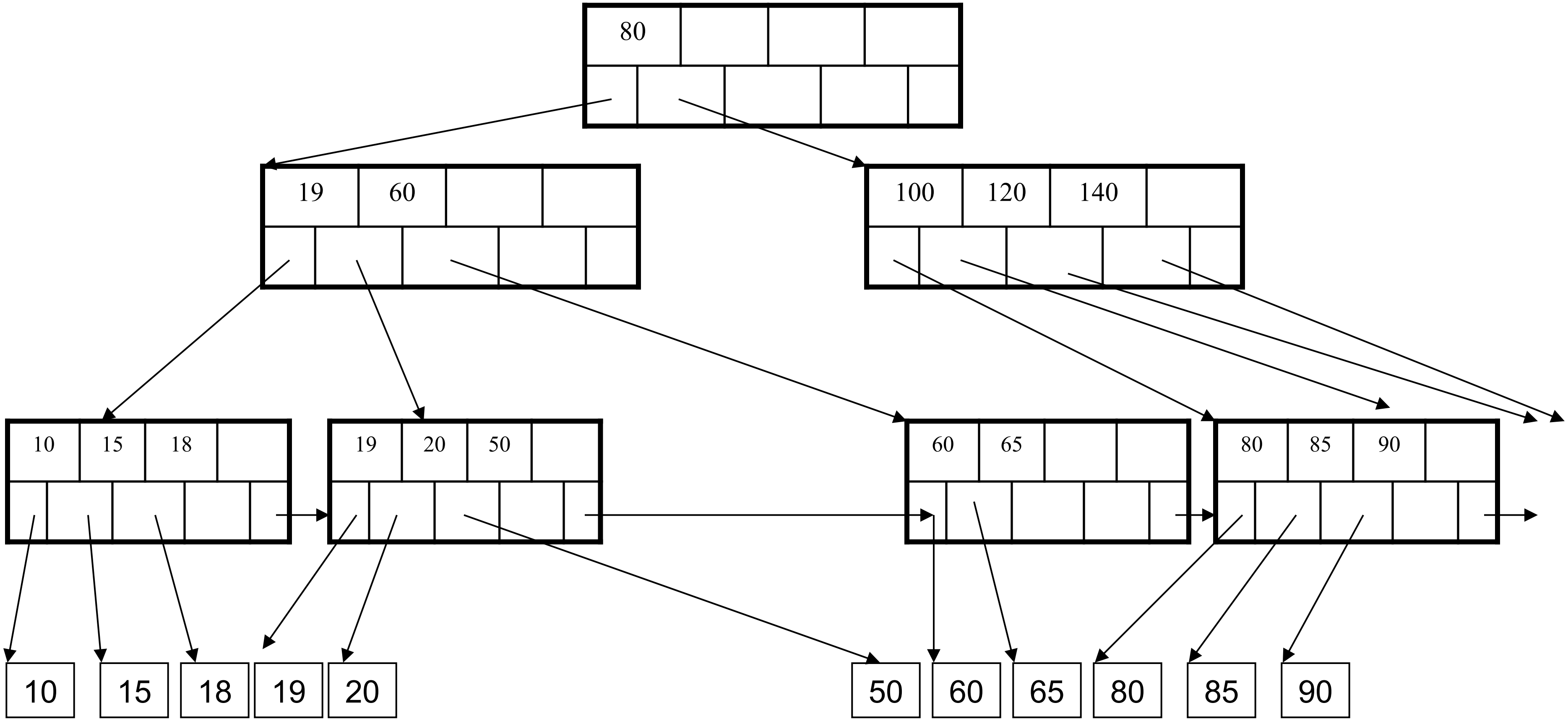
Rotation not possible

Need to *merge* nodes



Deletion from a B+ Tree

Final tree



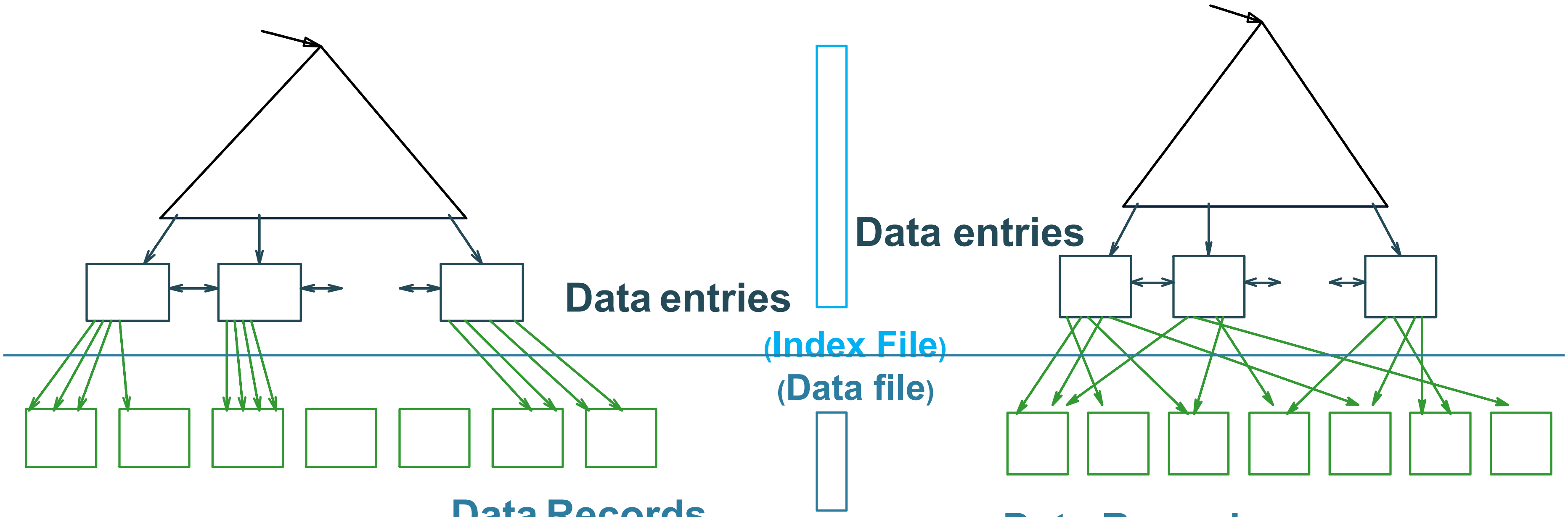
Deletion: Summary

- If capacity ≥ min-capacity: **Stop**
- If neighbor capacity > min-capacity: rotate, then **Stop**
- Merge with a neighbor (choose right or left) and steal a key from parent
  - Parent has one fewer keys: **Repeat** process on the parent
  - What if the parent was the root?

Discussion

- Reads are very fast
- Inserts are slow, in the sense that they requires several block writes
- LSM trees speed up writes, with only minor penalty for reads (to discuss later)

Clustered v.s. Unclustered B+ Trees



Data entries

Data entries

(Index File)

(Data file)

Data Records

Data Records

CLUSTERED

UNCLUSTERED

Note: can also store data records directly as data entries

Searching a B+ Tree

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

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

- Range queries:
  - Find lowest bound as above
  - Then sequential traversal

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

- Less effective for multi-range
  - Can only use one B+ tree, ignore the other(s)
  - Called *access path selection*

```
Select name
From Student
Where age = 25
    and GPA = 3.5
```