Storage and Indexing

Introduction to Databases

CompSci 316 Fall 2020

DUKE
COMPUTER SCIENCE

Announcements (Thu. Oct 1)

- Keep working on your project!
  - MS-2 due in two weeks (10/15)
  - Need to submit a basic working version of your website (all functionalities not needed, but interactions from/to UI and databases should be there)+ other things

- HW-5/Gradiance-3 to be released today
  - Due in a week 10/8 (Thu)

Where are we now?

Relational model and queries
- Relational Model
- Query in SQL
- Query in RA

Database Design
- E/R diagram (design from scratch)
- Normal Forms (refine design)

DBMS Internals and Query Processing
- Storage
- Index
- Join algo/Sorting
- Execution/ Optimization

Beyond Relational Model
- XML
- NOSQL JSON/MongoDB

Transactions
- Basics
- Concurrency Control
- Recovery

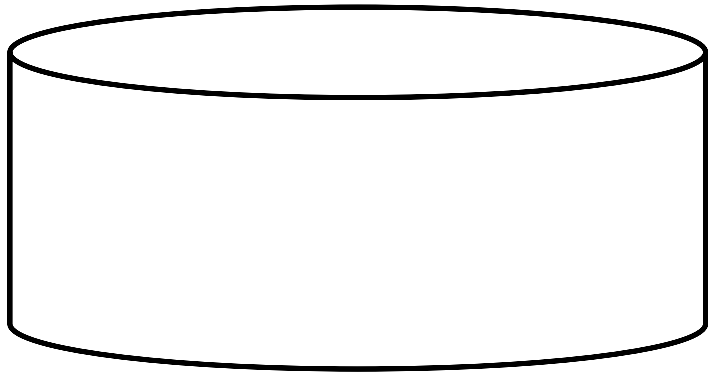
(Basic) Big Data Processing
- Map-Reduce
- Parallel DBMS

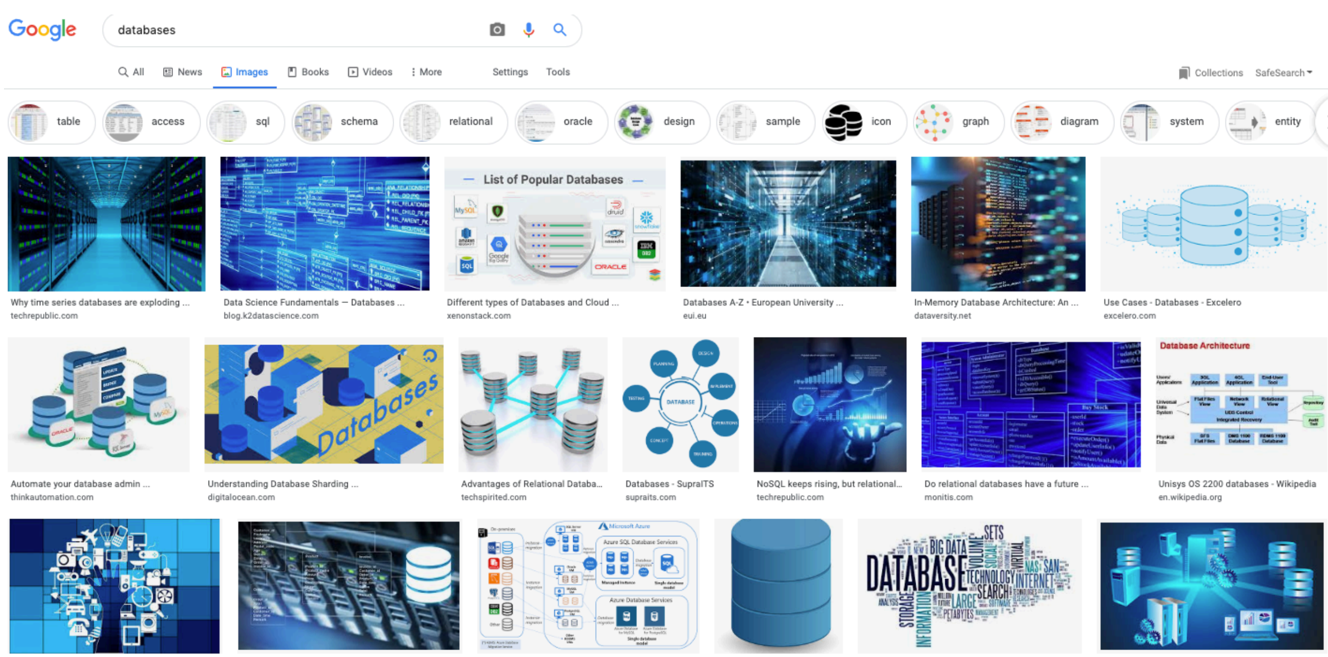
Covered

To be covered

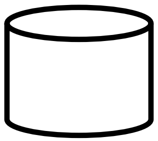
Next

Why do we draw databases like this?

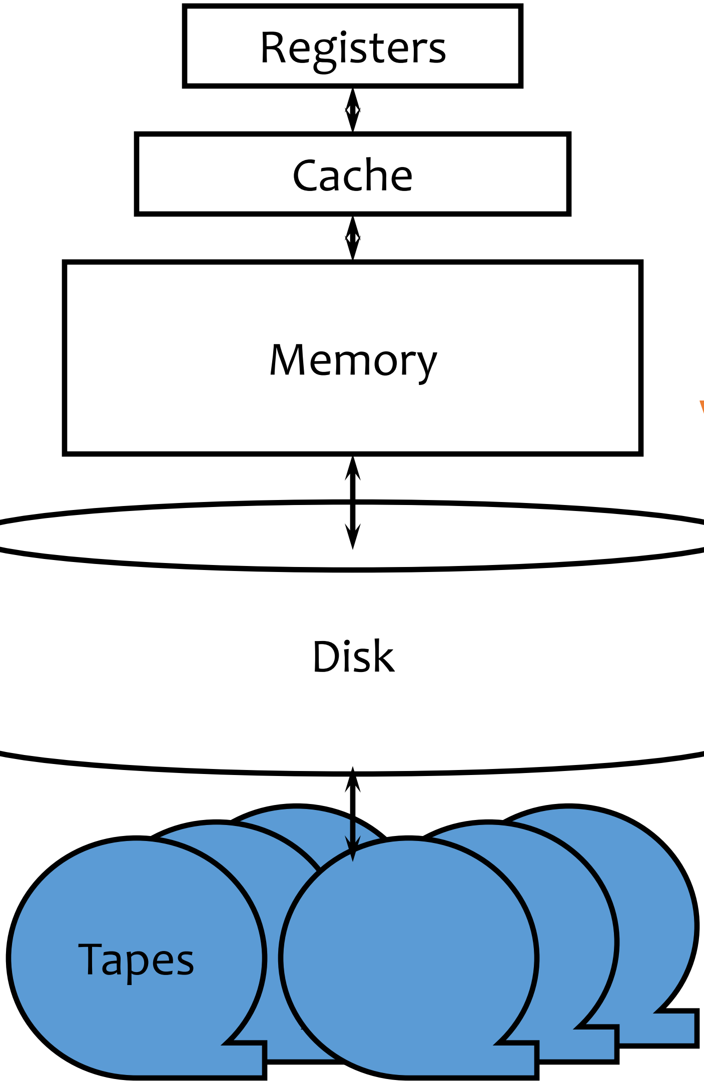




Outline

- It’s all about disks!
  - That’s why we always draw databases as 
  - And why the single most important metric in database processing is (oftentimes) the number of disk I/O’s performed

Storage hierarchy



Why a hierarchy?

How far away is data?

Location	Cycles	Location	Time
Registers	1	My head	1 min.
On-chip cache	2	This room	2 min.
On-board cache	10	Duke campus	10 min.
Memory	100	Washington D.C.	1.5 hr.
Disk	10^6	Pluto	2 yr.
Tape	10^9	Andromeda	2000 yr.

(Source: AlphaSort paper, 1995)
The gap has been widening!

☞ I/O dominates—design your algorithms to reduce I/O!

Latency Numbers Every Programmer Should Know

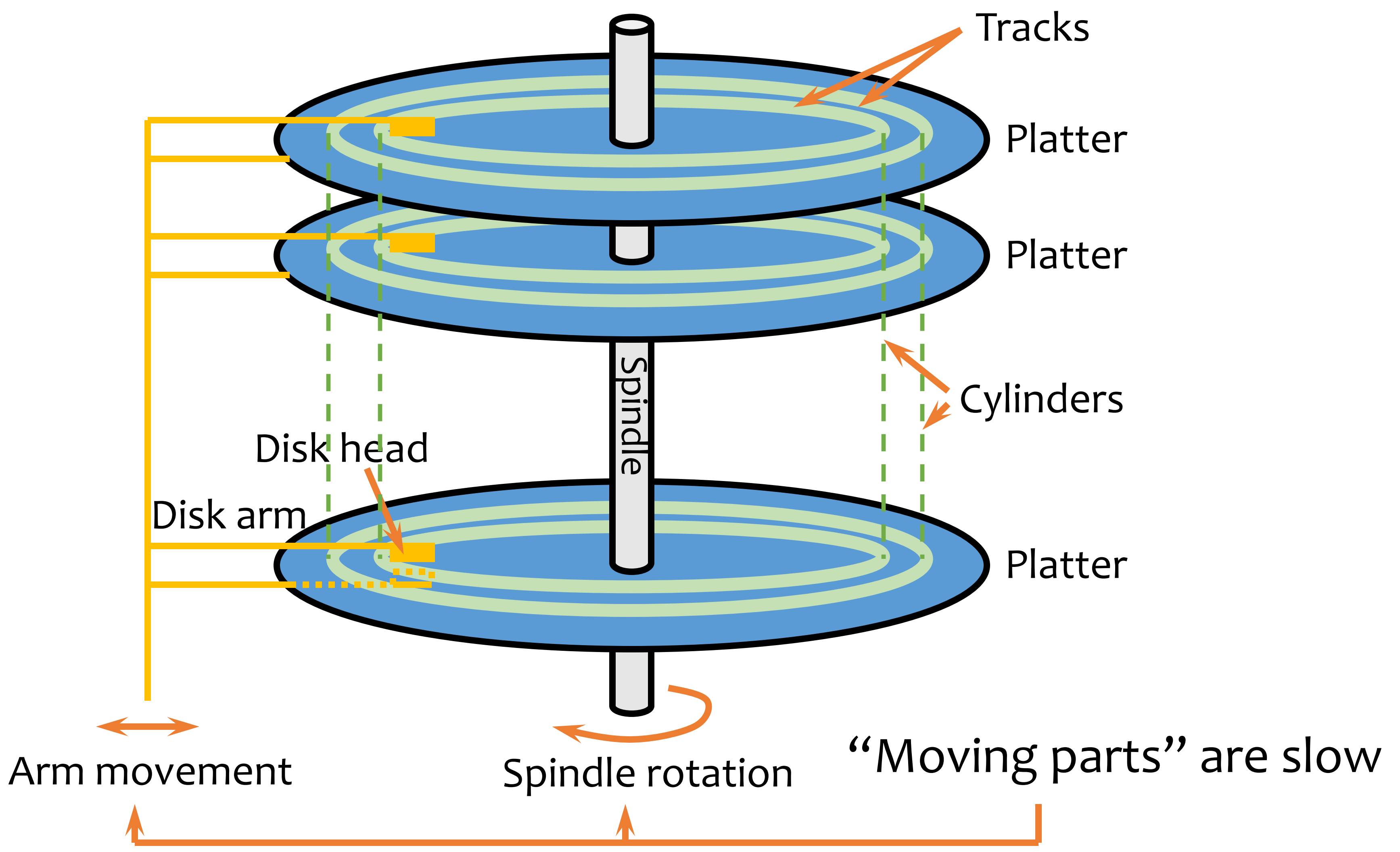
Just FYI –
Take a look yourself!

```
Latency Comparison Numbers
--------------------------
L1 cache reference                           0.5 ns
Branch mispredict                            5   ns
L2 cache reference                           7   ns                      14x L1 cache
Mutex lock/unlock                           25   ns
Main memory reference                      100   ns                      20x L2 cache, 200x L1 cache
Compress 1K bytes with Zippy             3,000   ns        3 us
Send 1K bytes over 1 Gbps network       10,000   ns       10 us
Read 4K randomly from SSD*             150,000   ns      150 us          ~1GB/sec SSD
Read 1 MB sequentially from memory     250,000   ns      250 us
Round trip within same datacenter      500,000   ns      500 us
Read 1 MB sequentially from SSD*     1,000,000   ns    1,000 us    1 ms  ~1GB/sec SSD, 4X memory
Disk seek                           10,000,000   ns   10,000 us   10 ms  20x datacenter roundtrip
Read 1 MB sequentially from disk    20,000,000   ns   20,000 us   20 ms  80x memory, 20X SSD
Send packet CA->Netherlands->CA    150,000,000   ns  150,000 us  150 ms

Notes
-----
1 ns = 10^-9 seconds
1 us = 10^-6 seconds = 1,000 ns
1 ms = 10^-3 seconds = 1,000 us = 1,000,000 ns

Credit
------
By Jeff Dean:               http://research.google.com/people/jeff/
Originally by Peter Norvig: http://norvig.com/21-days.html#answers
```

A typical hard drive

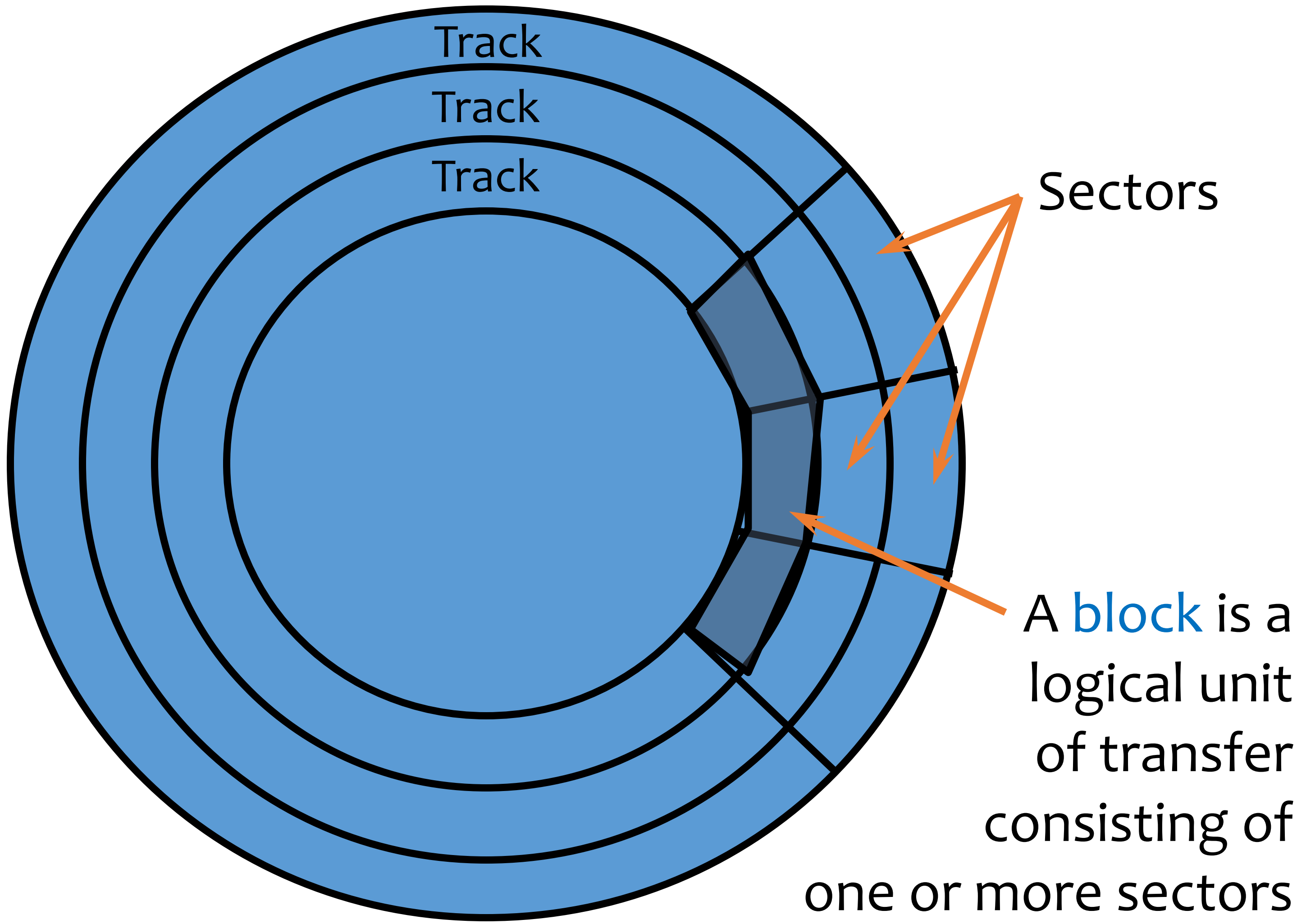
http://upload.wikimedia.org/wikipedia/commons/f/f8/Laptop-hard-drive-exposed.jpg

A typical hard drive



Top view

"Zoning": more sectors/data on outer tracks



Disk access time

Sum of:

- Seek time: time for disk heads to move to the correct cylinder
- Rotational delay: time for the desired block to rotate under the disk head
- Transfer time: time to read/write data in the block (= time for disk to rotate over the block)

Sequential vs. Random disk access

Seek time + rotational delay + transfer time

- Average seek time
  - Sequential: 0
  - Random: "Typical" value: 5 ms

- Average rotational delay
  - Sequential: 0
  - Random: "Typical" value: 4.2 ms (7200 RPM)

- Transfer time
  - Thee same for sequential and random

- Sequential is an order of magnitude faster!

Important consequences

- It's all about reducing I/O's!
- Cache blocks from stable storage in memory
  - DBMS maintains a memory buffer pool of blocks
  - Reads/writes operate on these memory blocks
  - Dirty (updated) memory blocks are "flushed" back to stable storage

Picture on board that we will use again and again!

Performance tricks

- Disk layout strategy
  - Keep related things (what are they?) close together: same sector/block → same track → same cylinder → adjacent cylinder
- Prefetching
  - While processing the current block in memory, fetch the next block from disk (overlap I/O with processing)
- Parallel I/O
  - More disk heads working at the same time
- Disk scheduling algorithm
  - Example: "elevator" algorithm
- Track buffer
  - Read/write one entire track at a time

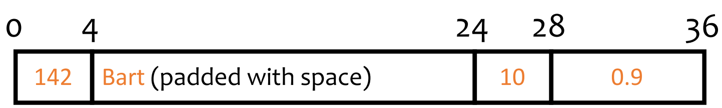
Not covered in detail

Data layout on disk

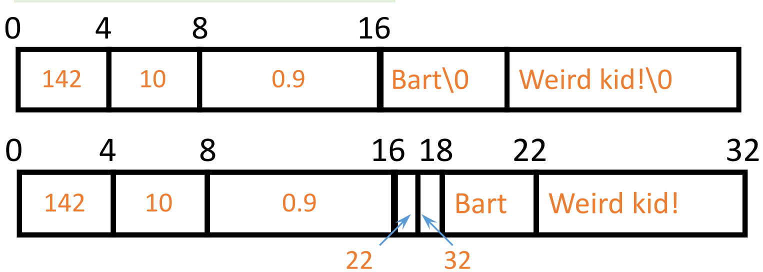
How each component is stored in the parent

Table → Pages/Blocks → Records/Tuples/Rows → Attributes

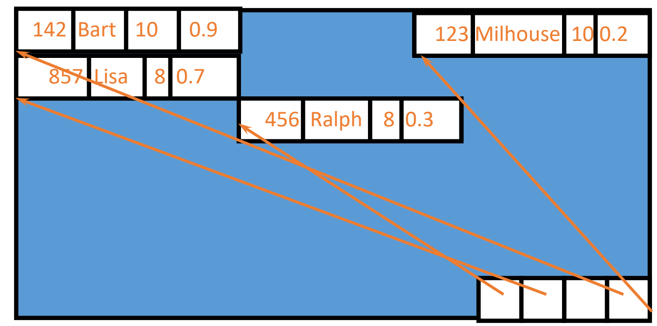
Examples:



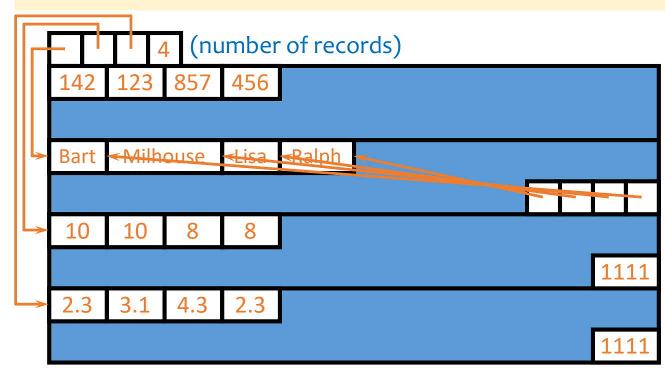
Fixed-length fields



Variable-length fields (delimiter or offset array)



N-ary storage model/NSM
"Row-major", directory at the end
Reorganization needed after updates



PAX (Partition Attributes Across) "Column-major" -> Column store

End of Lecture 10/1

Take-away

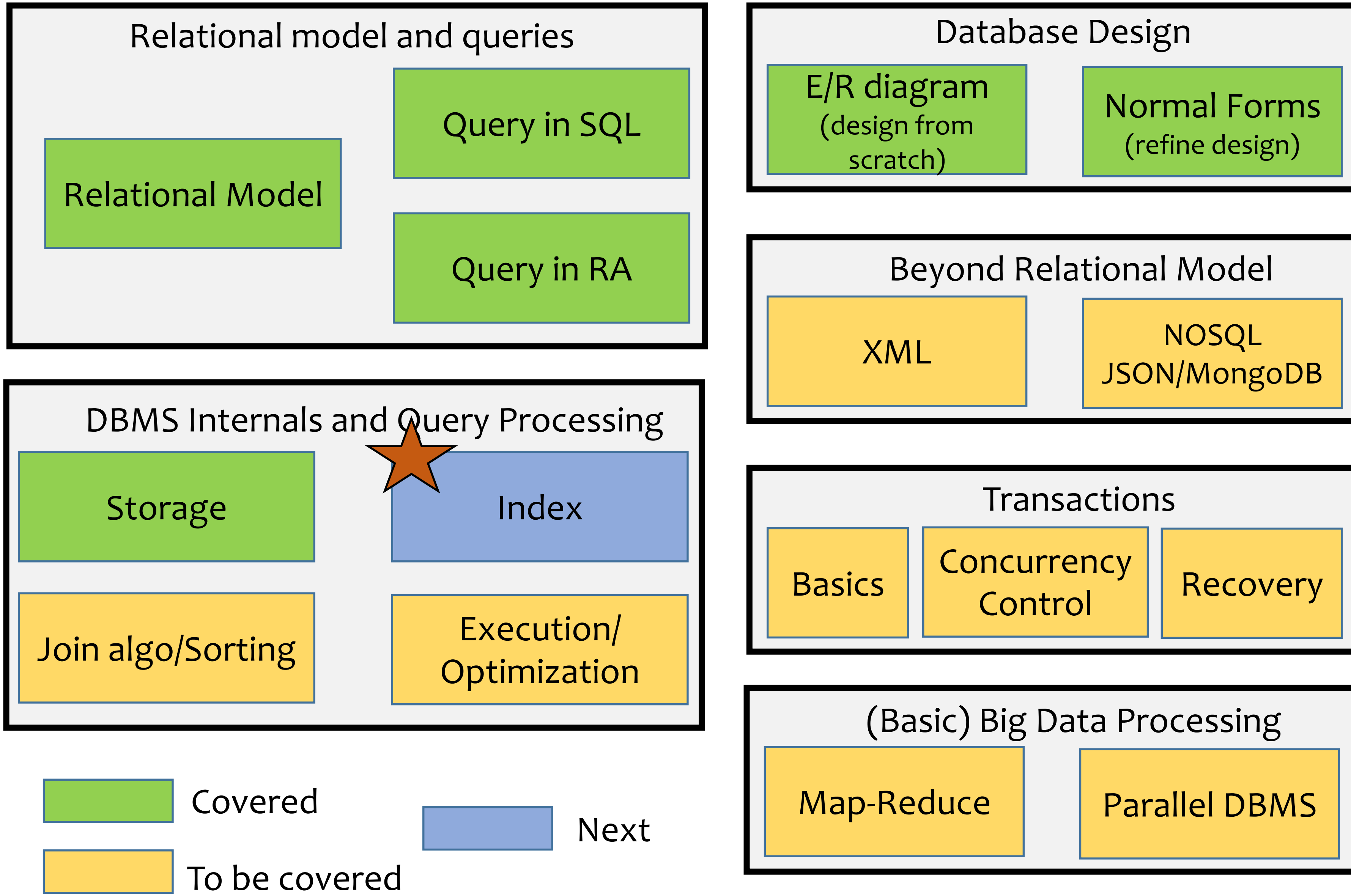
- Storage hierarchy
  - Why I/O’s dominate the cost of database operations
- Disk
  - Steps in completing a disk access
  - Sequential versus random accesses
- Disk is slower than Main memory = Buffer Pool
  - Minimize the number of transfers to/from Disk
  - Our unit of cost!
    - All computation cost ignored by default

Index

Announcements (Tue. Oct 6)

- HW-5 + Gradiance-3 (Constraints/Triggers)
  - Due this Friday 10/9
- Keep working on your project!
  - MS-2 due next week (10/15)
  - Need to submit a basic working version of your website (all functionalities not needed, but interactions from/to UI and databases should be there) + other things
- If you would like to meet me one-one, please email Yesenia and me ASAP
  - By tomorrow (Wed 10/7)

Where are we now?



Recall the Disk-Main Memory diagram!

Topics

- Index

- Dense vs. Sparse
- Clustered vs. unclustered
- Primary vs. secondary
- Tree-based vs. Hash-index

Related (Dense vs. Sparse; Clustered vs. unclustered; Primary vs. secondary)

What are indexes for?

- Given a value, locate the record(s) with this value

  SELECT * FROM *R* WHERE *A = value*;

  SELECT * FROM *R*, *S* WHERE *R.A = S.B*;

- Find data by other search criteria, e.g.
  - Range search

    SELECT * FROM *R* WHERE *A > value*;

  - Keyword search

Focus of this lecture

Dense and sparse indexes

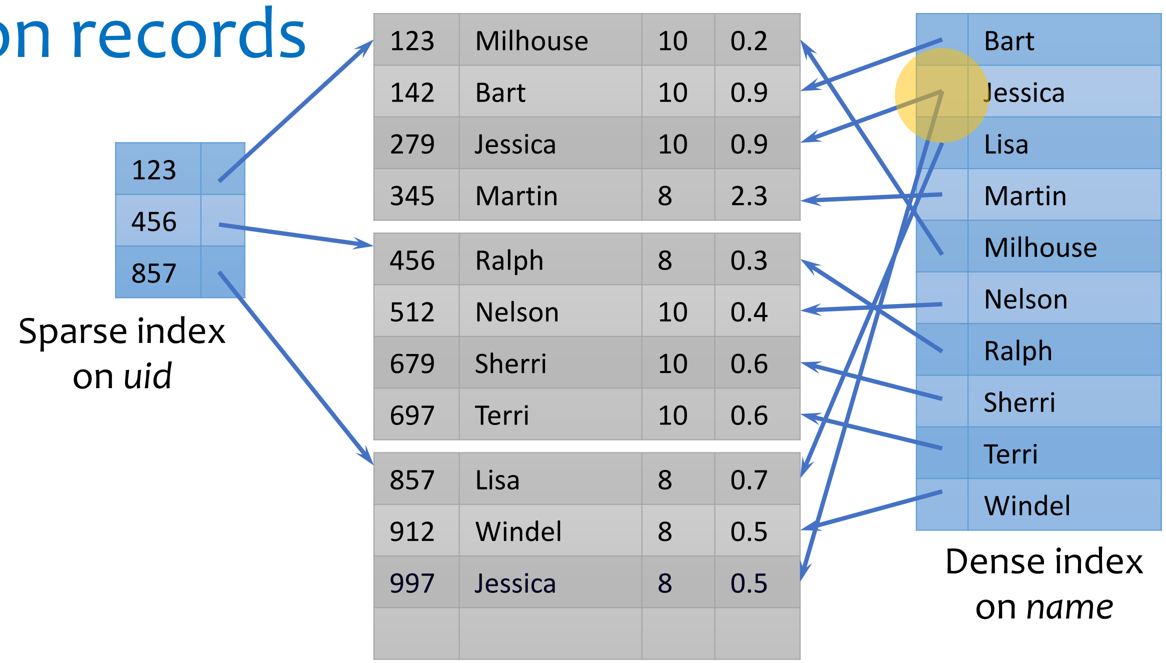
When are these possible?

Comparison?

- Dense: one index entry for each search key value
  - One entry may "point" to multiple records (e.g., two users named Jessica)
- Sparse: one index entry for each block
  - Records must be clustered according to the search key

Dense versus sparse indexes

- Index size
  - ??
- Requirement on records
  - ??
- Lookup
  - ??
- Update
  - ??



Dense versus sparse indexes

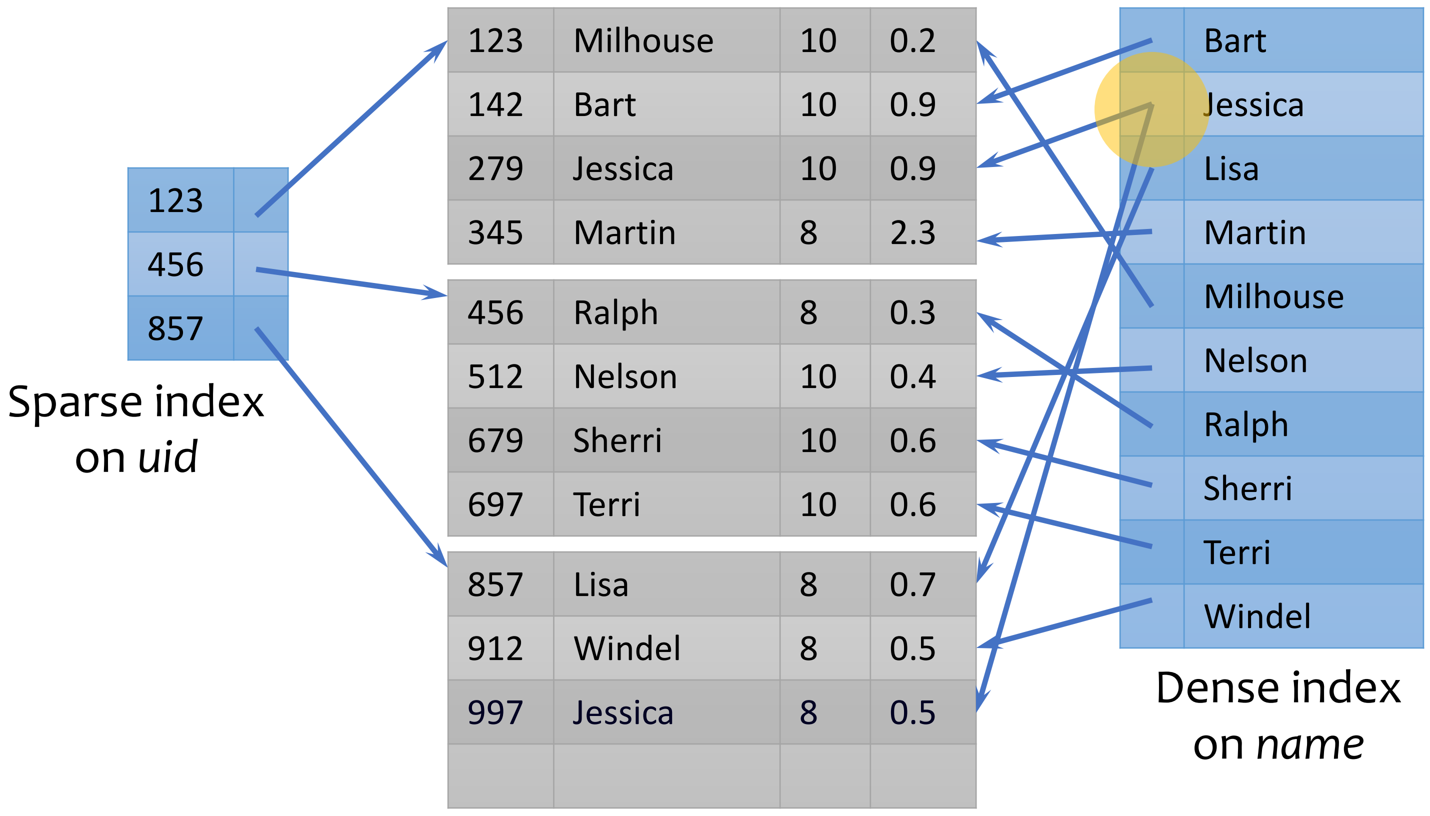
- Index size
  - Sparse index is smaller
- Requirement on records
  - Records must be clustered for sparse index
- Lookup
  - Sparse index is smaller and may fit in memory
  - Dense index can directly tell if a record exists
- Update
  - May be easier for sparse index (less movement for updates)

Primary and secondary indexes

- Primary index
  - Created for the primary key of a table
  - Records are usually clustered by the primary key
  - Can be sparse
- Secondary index
  - Usually dense
- SQL
  - PRIMARY KEY declaration automatically creates a primary index, UNIQUE key automatically creates a secondary index
  - Additional secondary index can be created on non-key attribute(s):
    `CREATE INDEX UserPopIndex ON User(pop);`

What if the index is too big as well?

What if the index is too big as well?

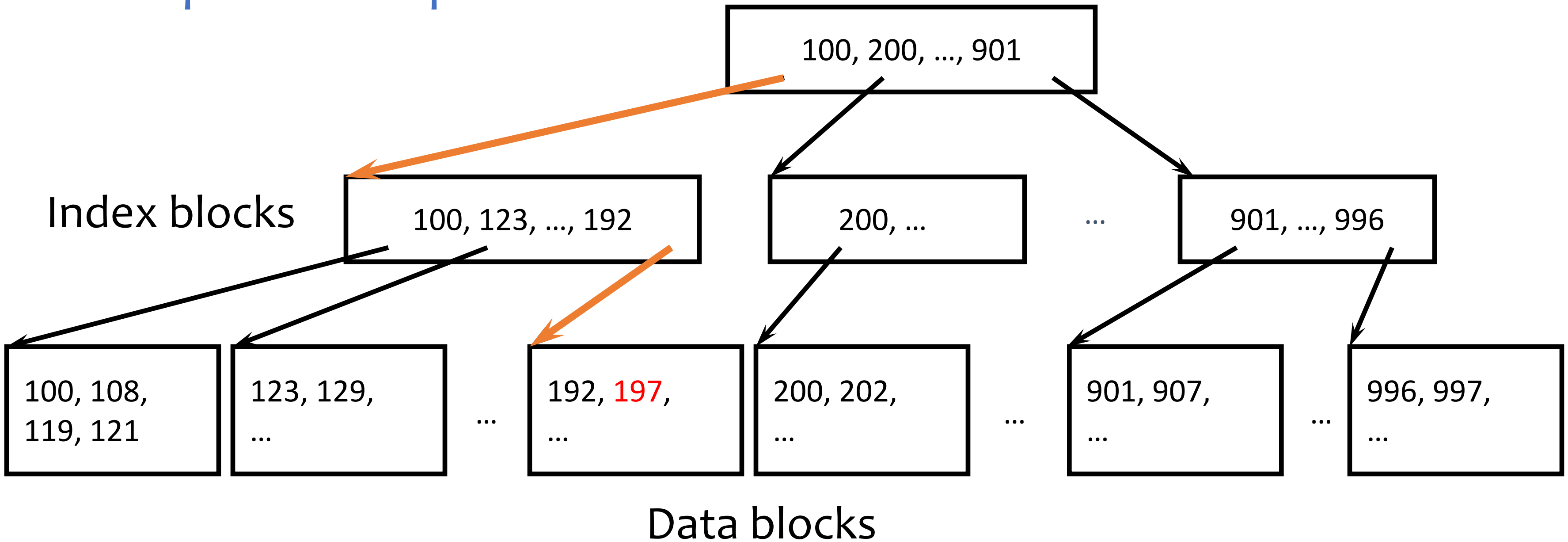


Put a another (sparse) index on top of that!

ISAM

- What if an index is still too big?
  - Put a another (sparse) index on top of that!
  - ☞ISAM (Index Sequential Access Method), more or less

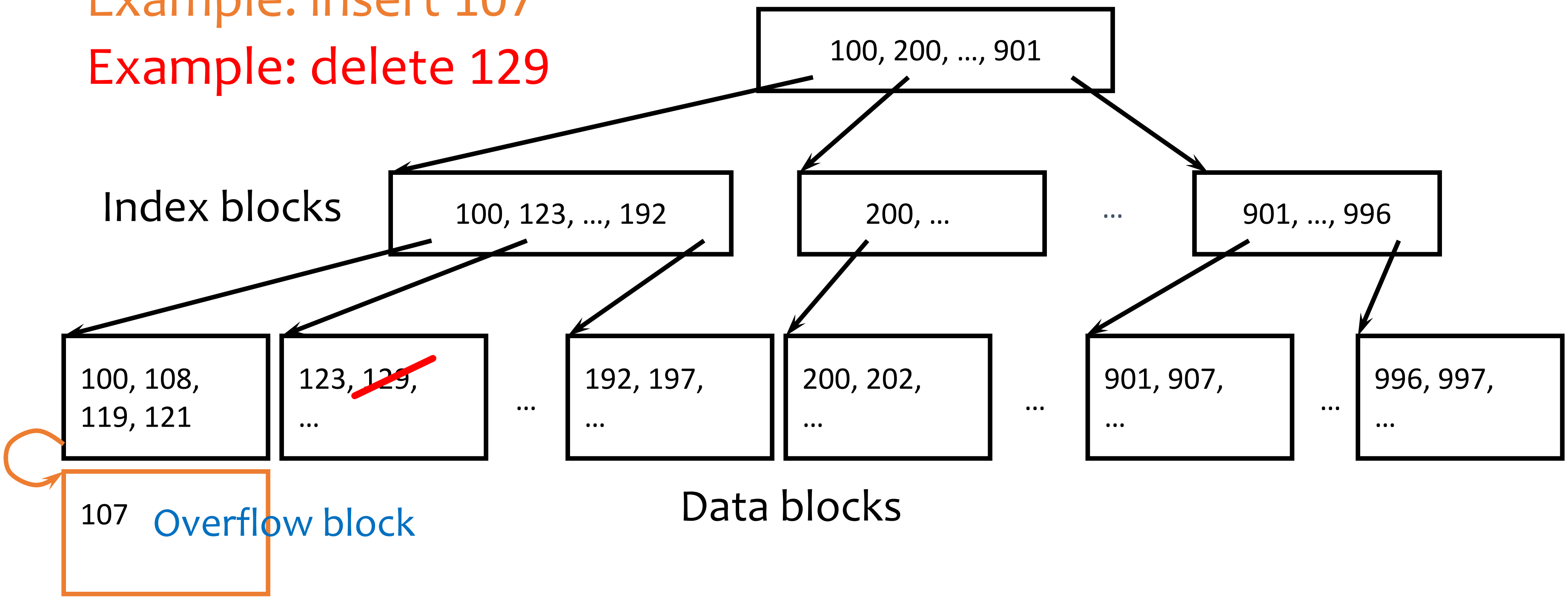
Example: look up 197



Updates with ISAM

Example: insert 107

Example: delete 129

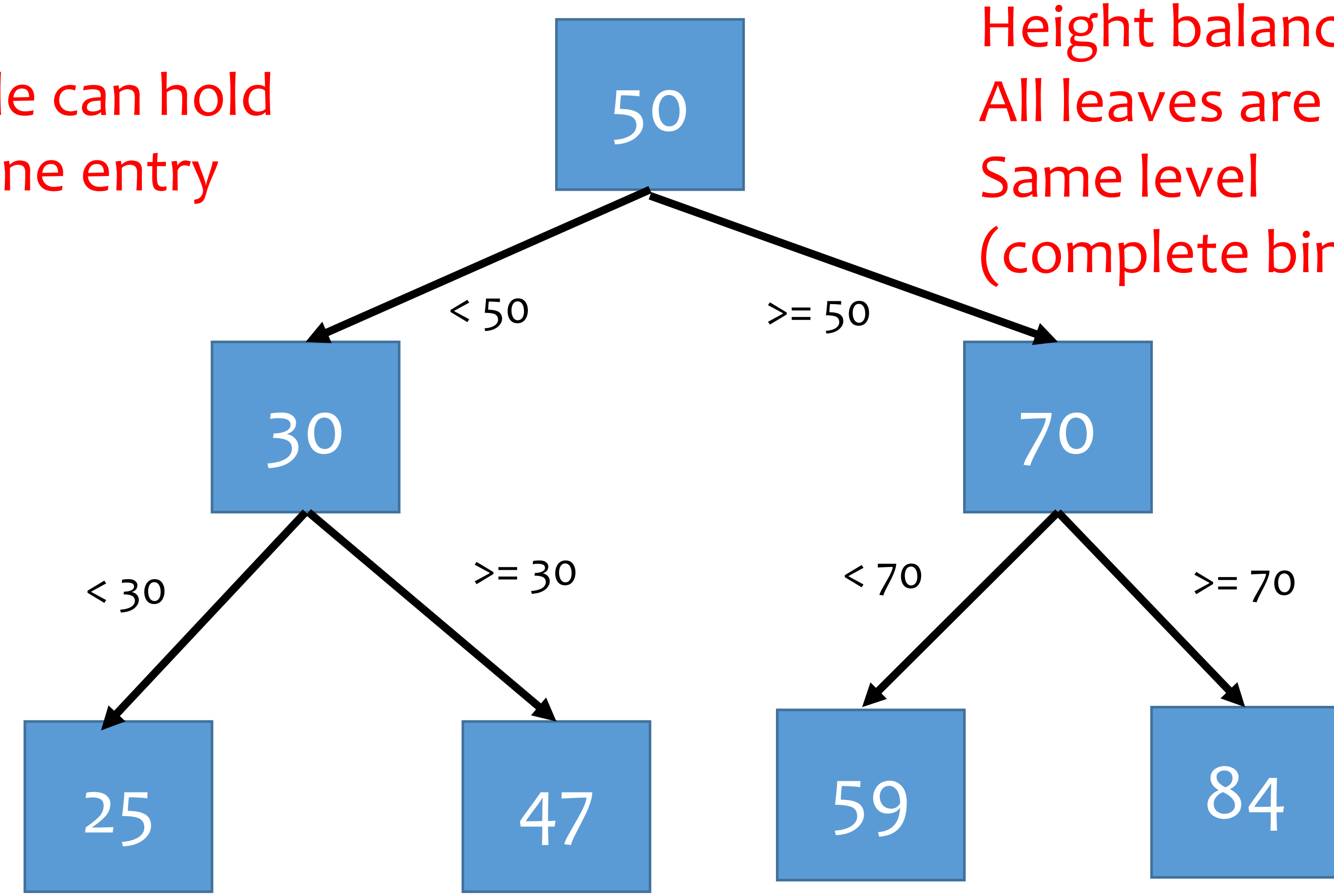


- Overflow chains and empty data blocks degrade performance
  - Worst case: most records go into one long chain, so lookups require scanning all data!

Binary Search Tree

Each node can hold
Exactly one entry

Height balanced:
All leaves are at the
Same level
(complete binary tree)

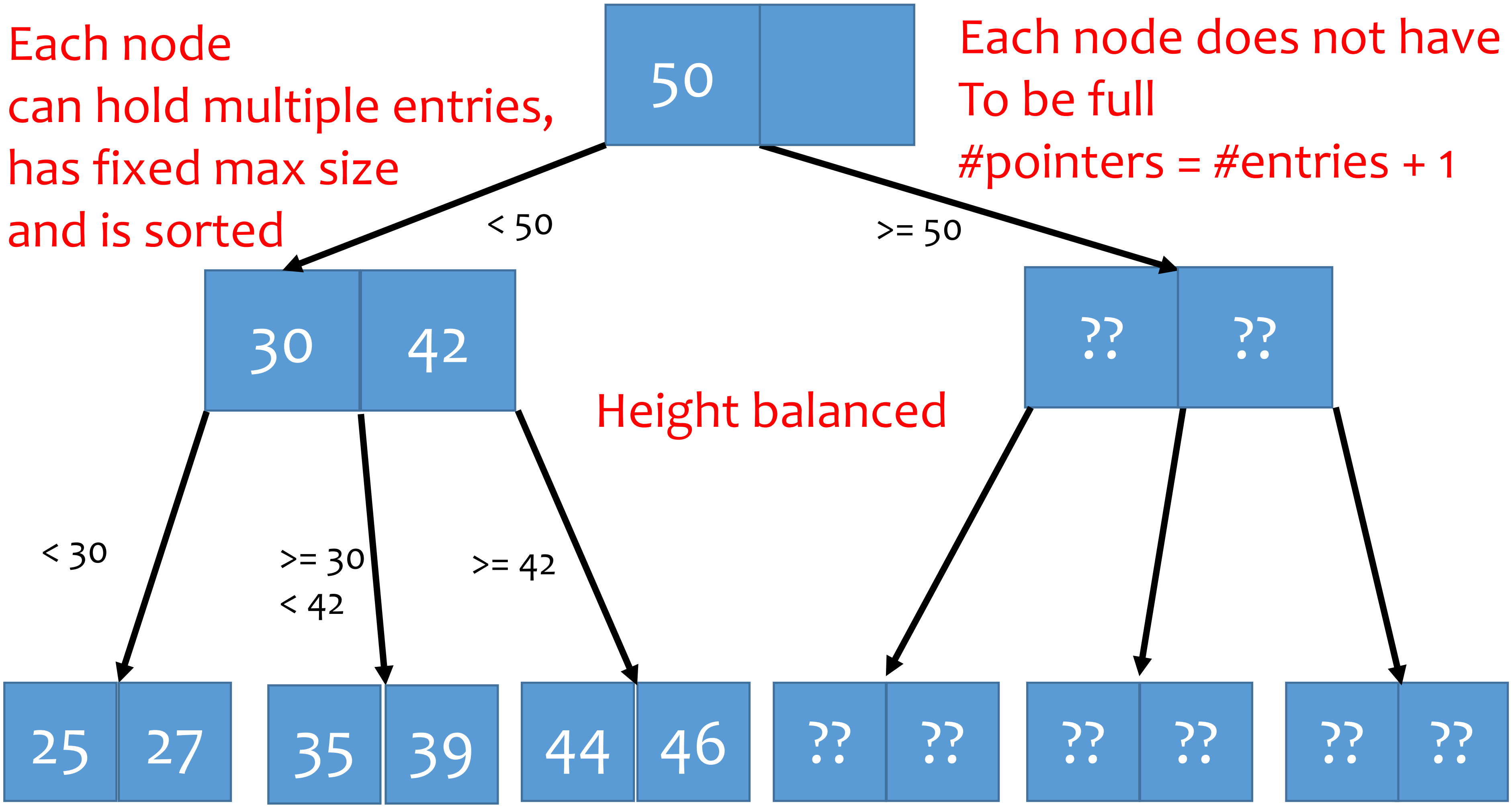


Leaves are sorted

B-tree: Generalizing Binary Search Trees

Each node
can hold multiple entries,
has fixed max size
and is sorted

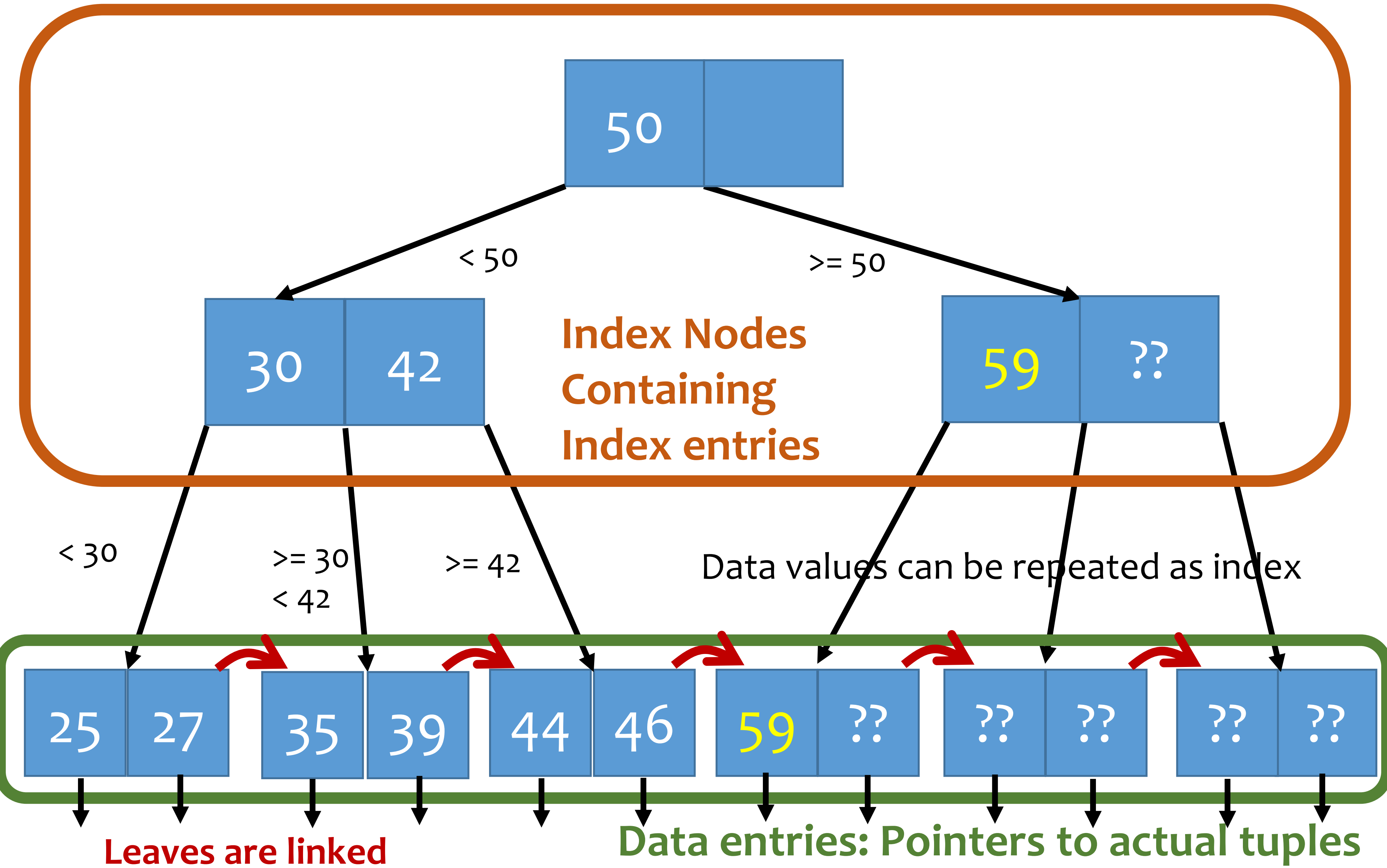
Each node does not have
To be full
#pointers = #entries + 1



Height balanced

Leaves are sorted

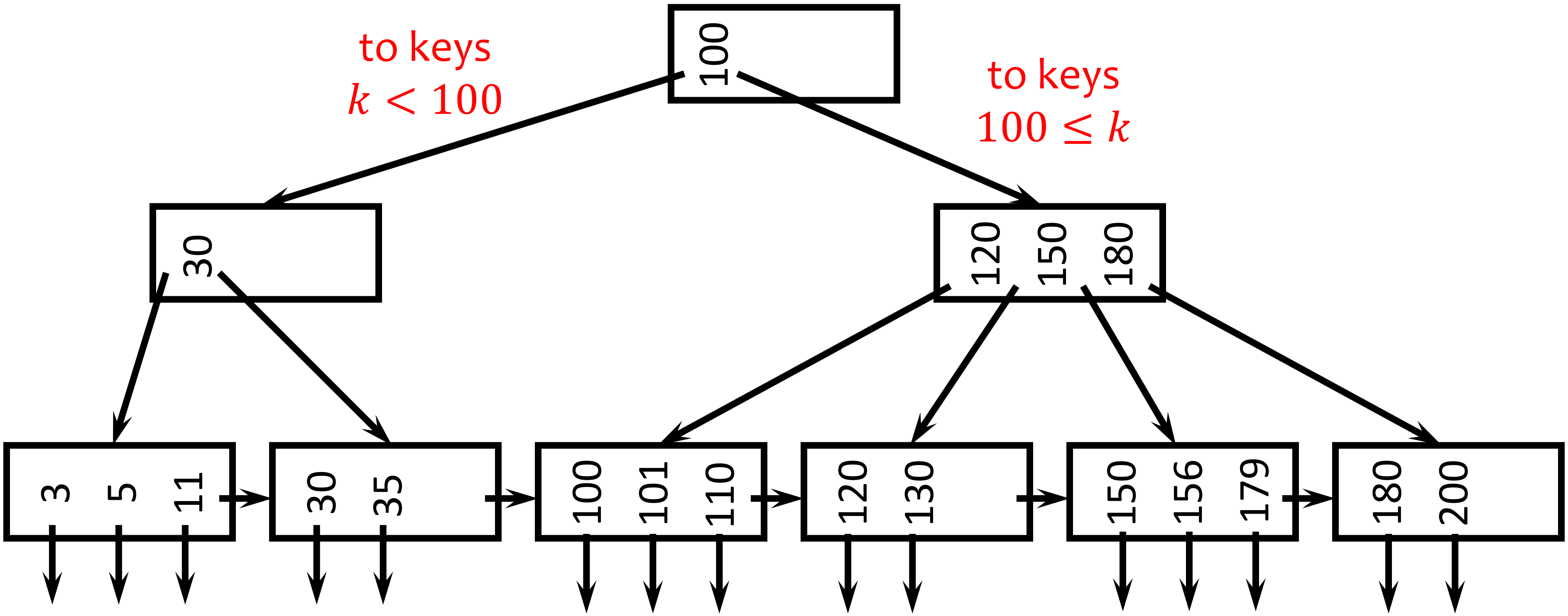
B+-tree: Data only at leaves



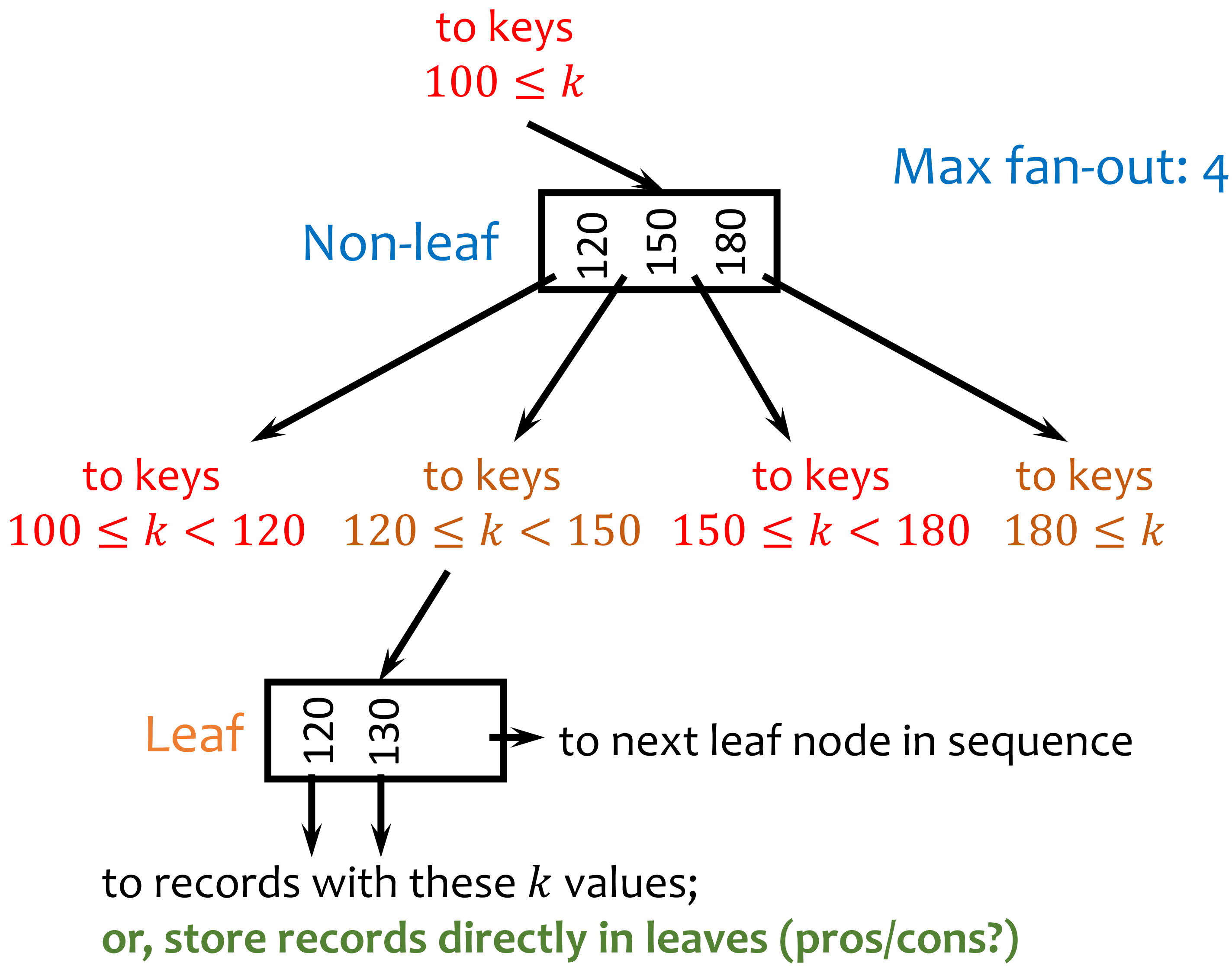
B+-tree: Closer Look

- A hierarchy of nodes with intervals
- Balanced (more or less): good performance guarantee
- Disk-based: one node per block; large fan-out

Max fan-out: 4



Sample B+-tree nodes



- Questions

- Why do we use B$^+$-tree as database index instead of binary trees?

vs.

- Why do we use B$^+$-tree as database index instead of B-trees?
  - What are the differences/pros/cons of B-trees vs. B$^+$-tree as index?

B^+-tree versus B-tree

- B-tree: why not store records (or record pointers) in non-leaf nodes?
  - These records can be accessed with fewer I/O's
- Problems?
  - Storing more data in a node decreases fan-out and increases h
  - Records in leaves require more I/O's to access
  - Vast majority of the records live in leaves!

B+-tree balancing properties

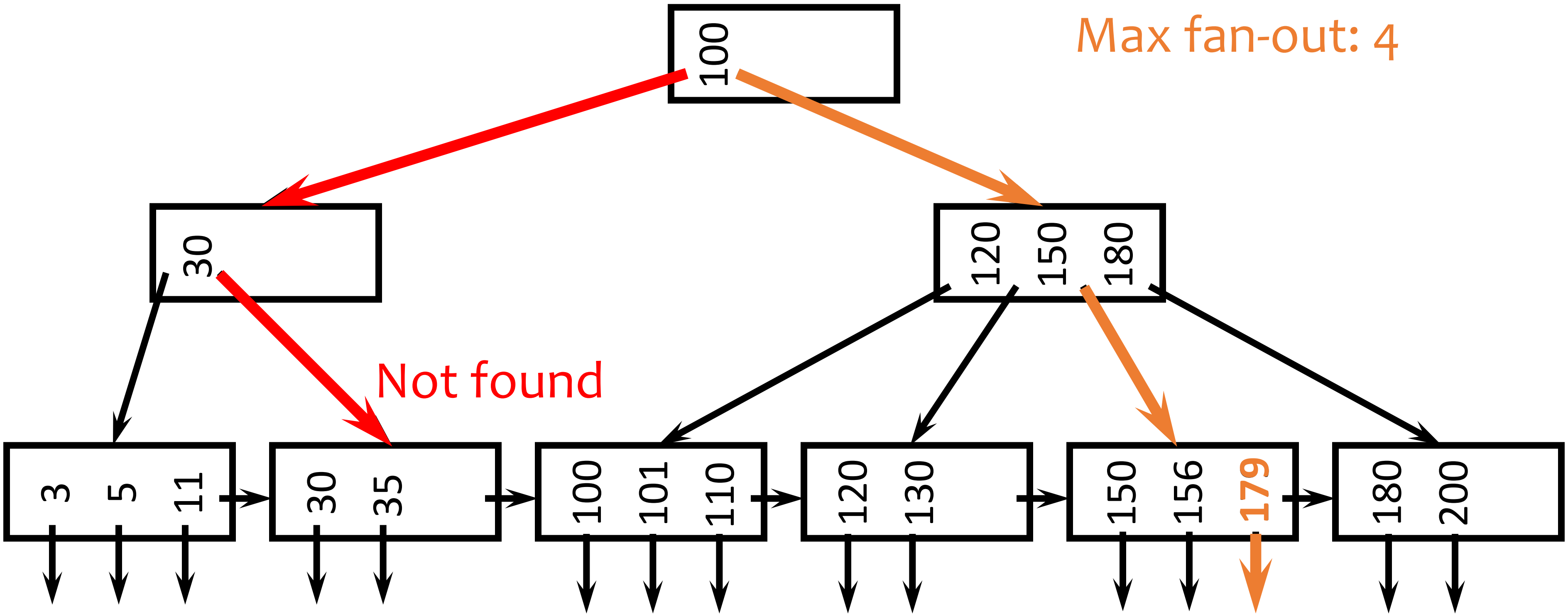
Check yourself

- Height constraint: all leaves at the same lowest level
- Fan-out constraint: all nodes at least half full (except root)

	Max # pointers	Max # keys	Min # active pointers	Min # keys
Non-leaf	f	$f-1$	$\lceil f/2 \rceil$	$\lceil f/2 \rceil - 1$
Root	f	$f-1$	2	1
Leaf	f	$f-1$	$\lfloor f/2 \rfloor$	$\lfloor f/2 \rfloor$

Lookups

- SELECT * FROM *R* WHERE *k* = 179;
- SELECT * FROM *R* WHERE *k* = 32;

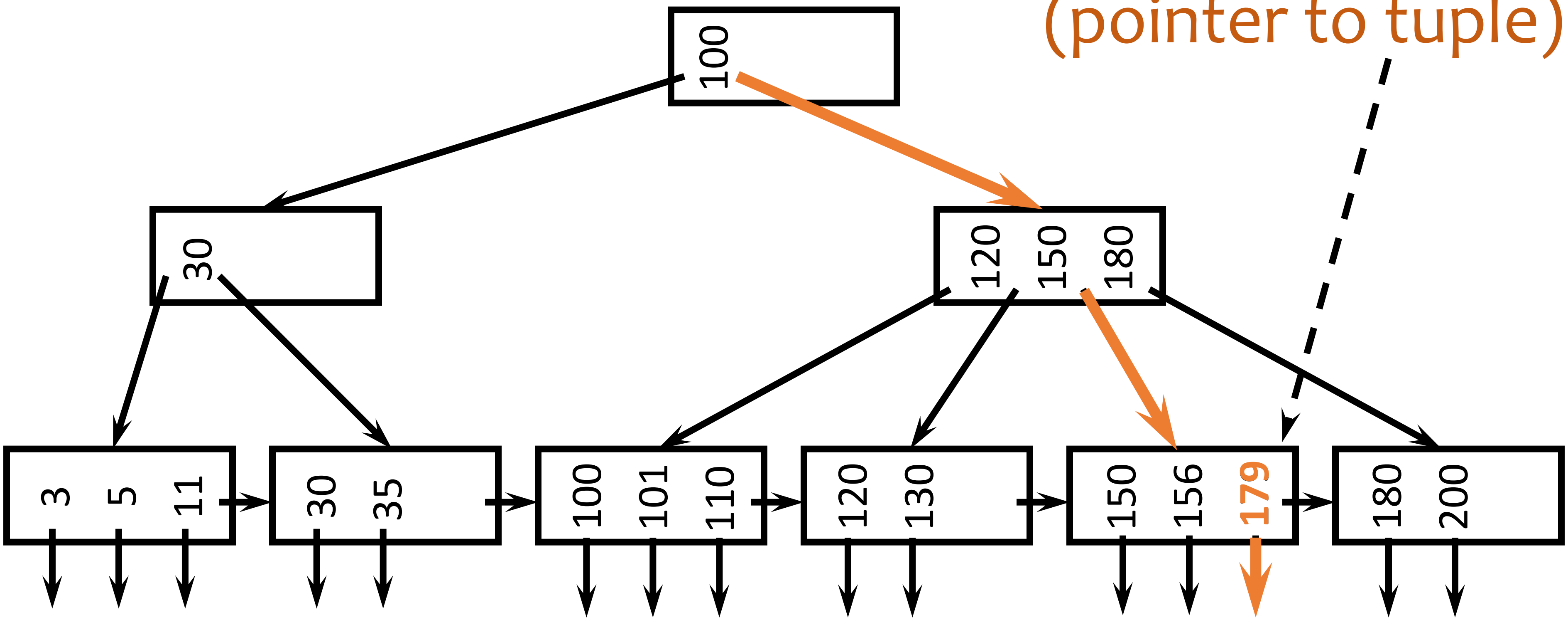


Search key and Data entry

- SELECT * FROM *R* WHERE *k* = 179;

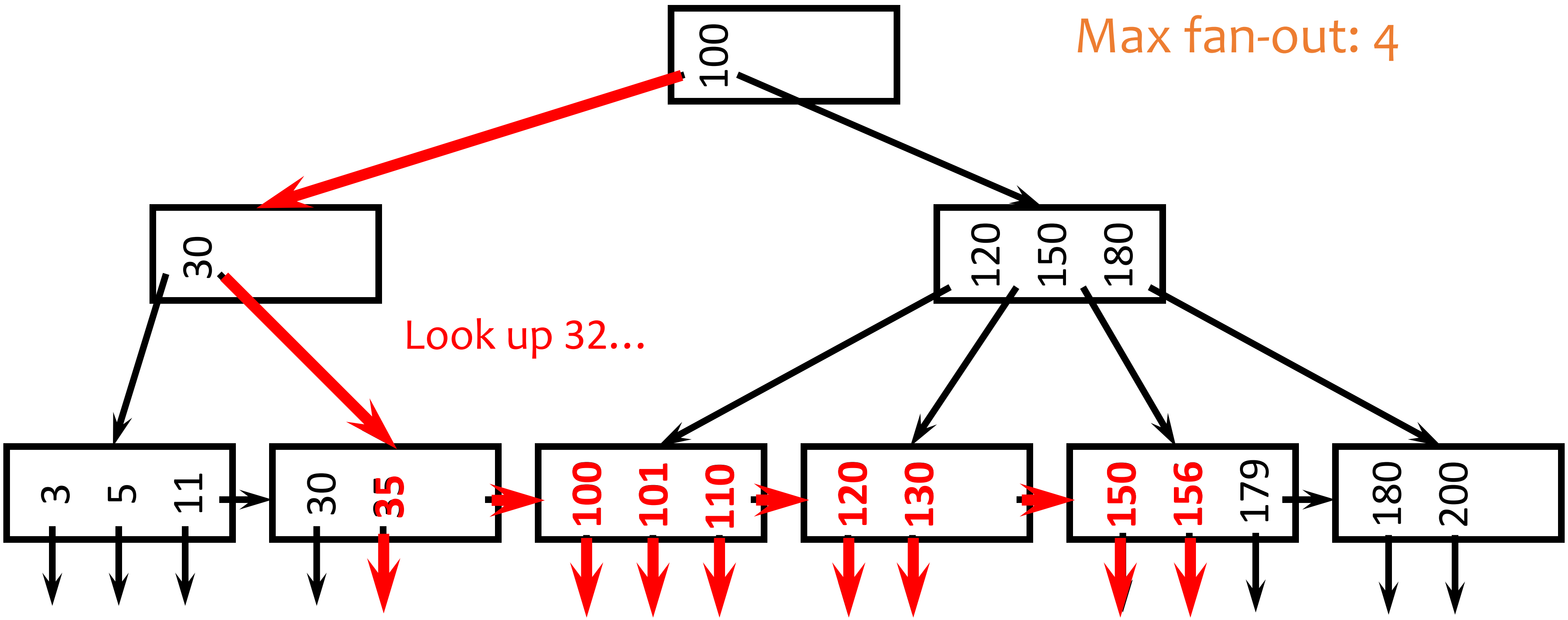
Search key (value)

Data Entry (pointer to tuple)



Range query

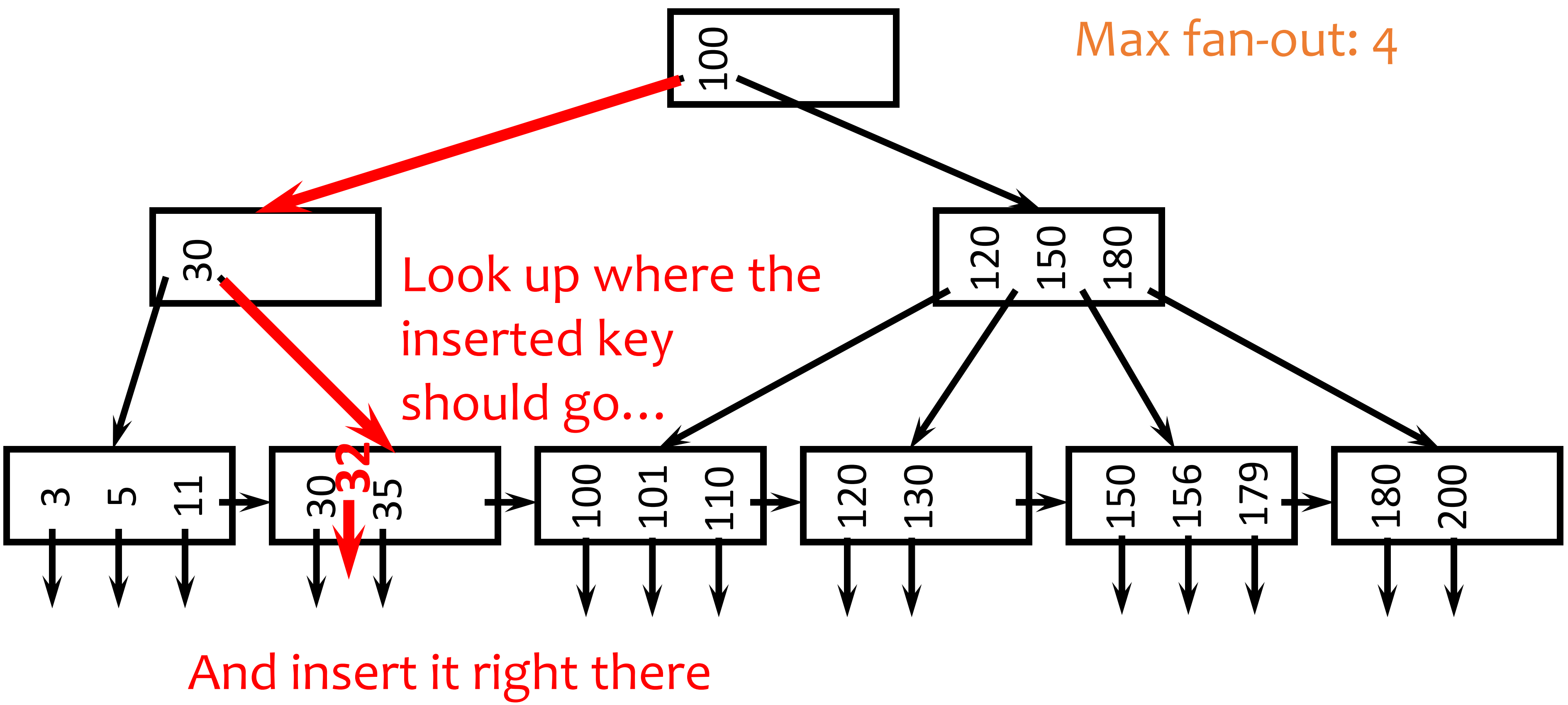
- SELECT * FROM *R* WHERE *k* > 32 AND *k* < 179;



And follow next-leaf pointers until you hit upper bound

Insertion

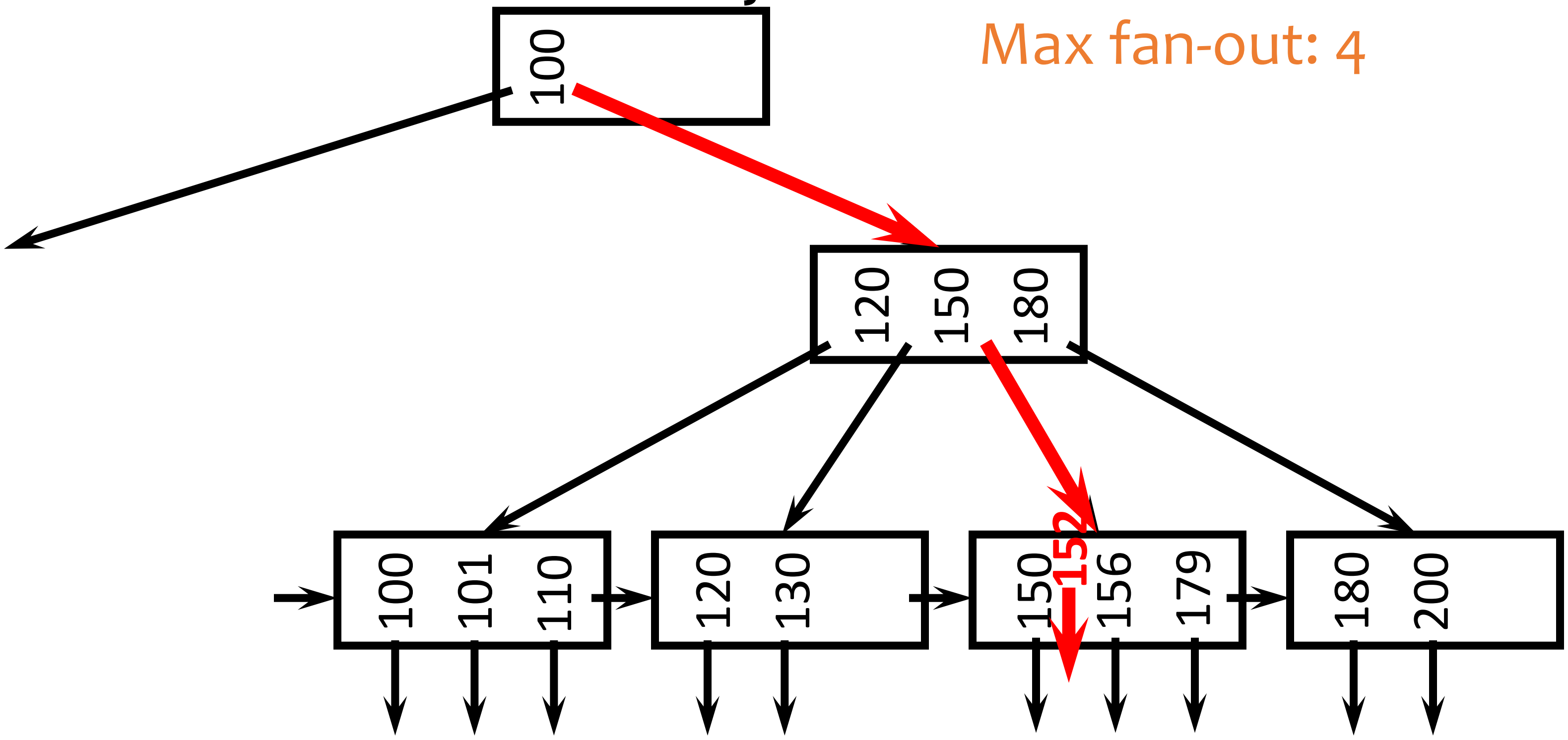
- Insert a record with search key value 32



Another insertion example

- Insert a record with search key value 152

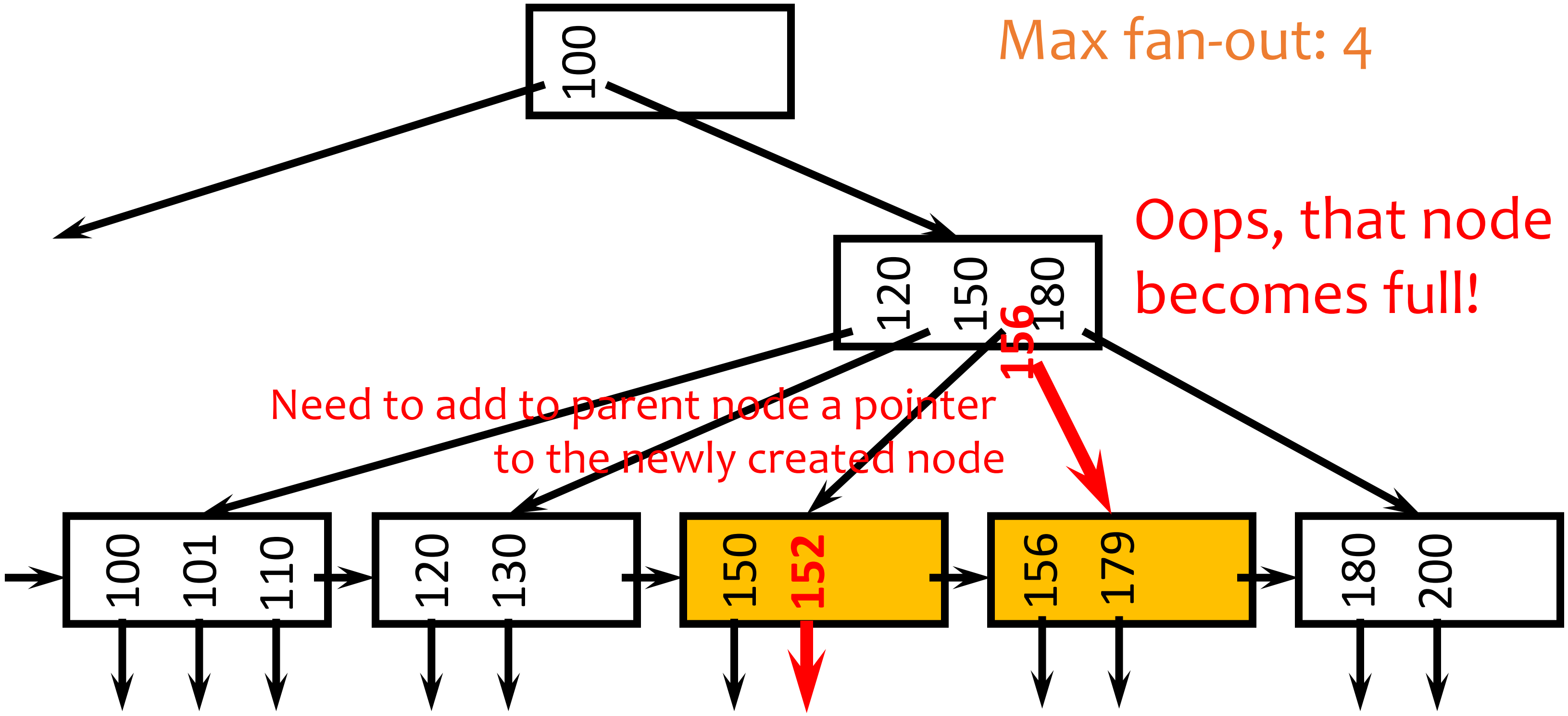
Max fan-out: 4



Oops, node is already full!

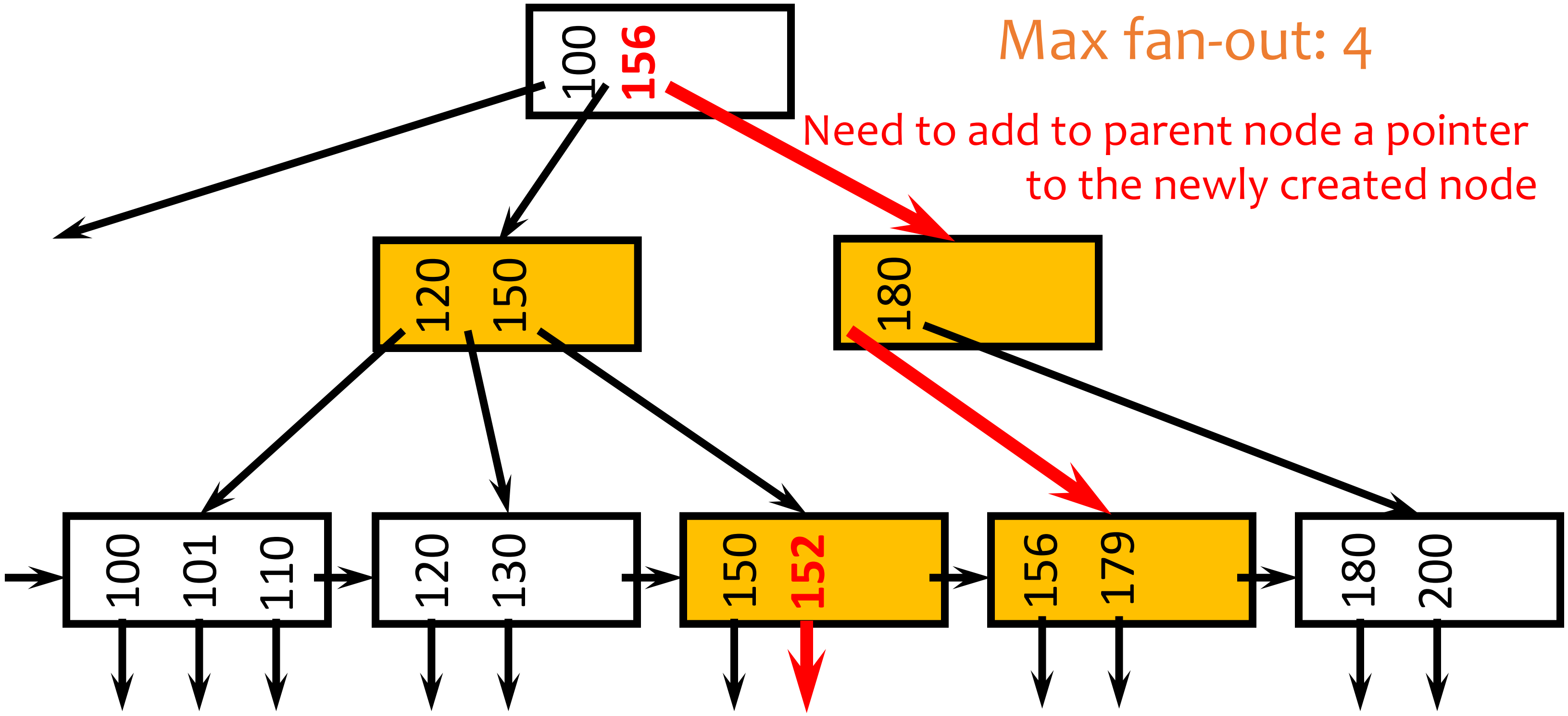
What are our options here?

Node splitting



End of Lecture 10/6

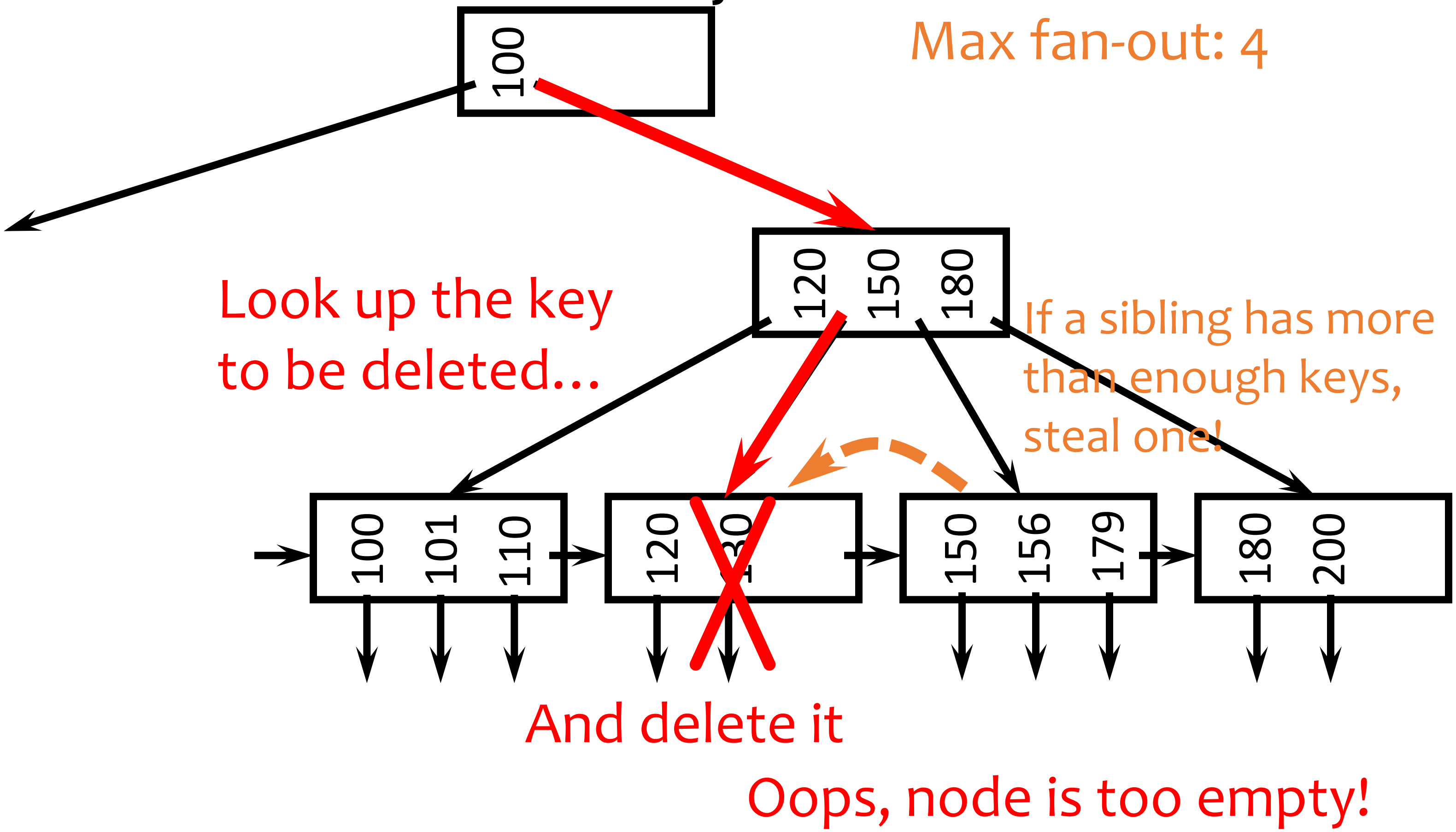
More node splitting



- In the worst case, node splitting can "propagate" all the way up to the root of the tree (not illustrated here)
  - Splitting the root introduces a new root of fan-out 2 and causes the tree to grow "up" by one level

Deletion

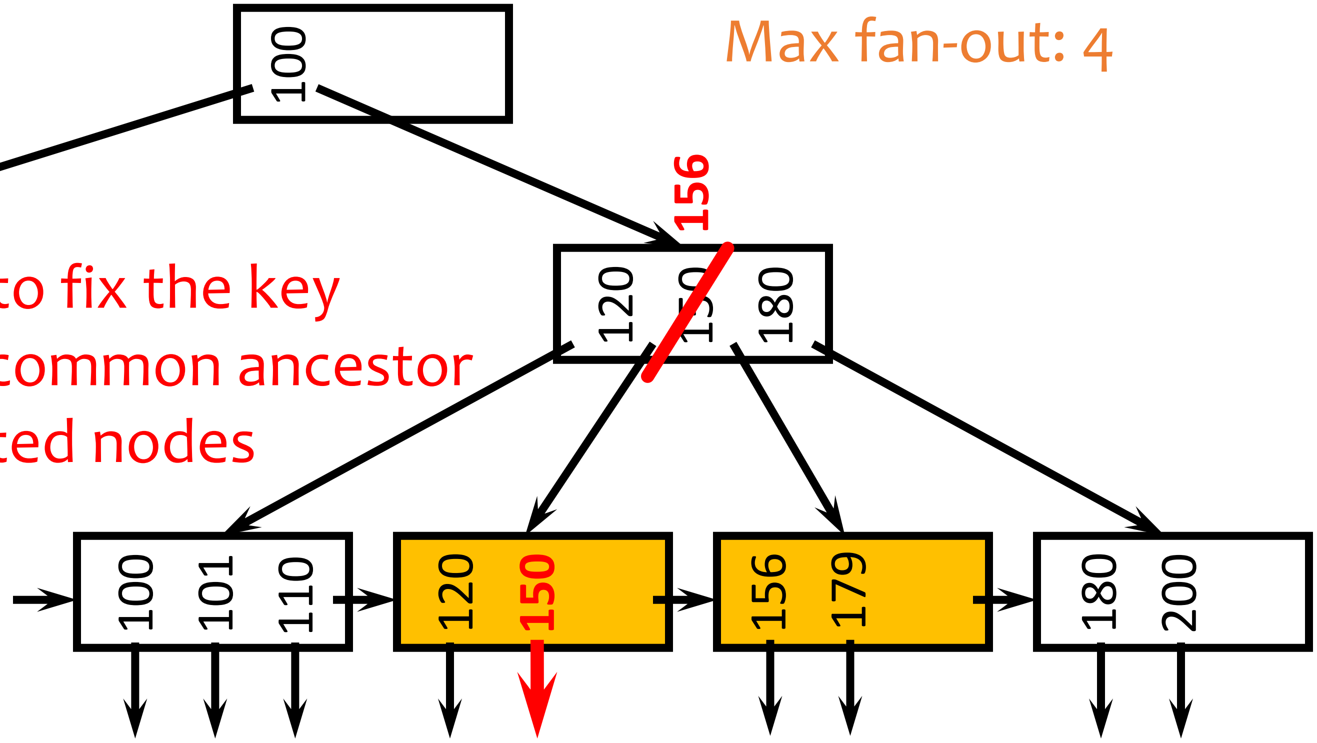
- Delete a record with search key value 130



Stealing from a sibling

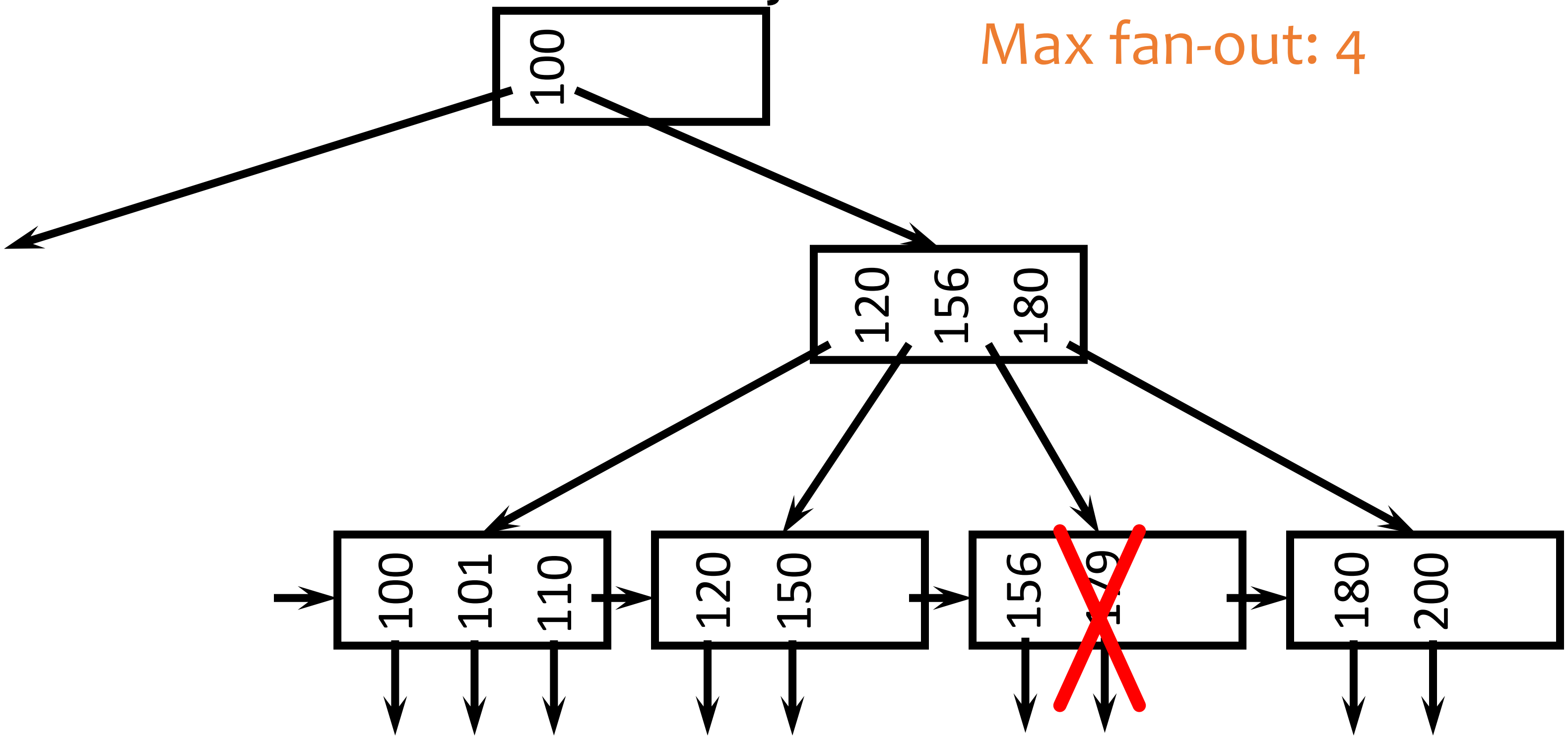
Max fan-out: 4

Remember to fix the key
in the least common ancestor
of the affected nodes

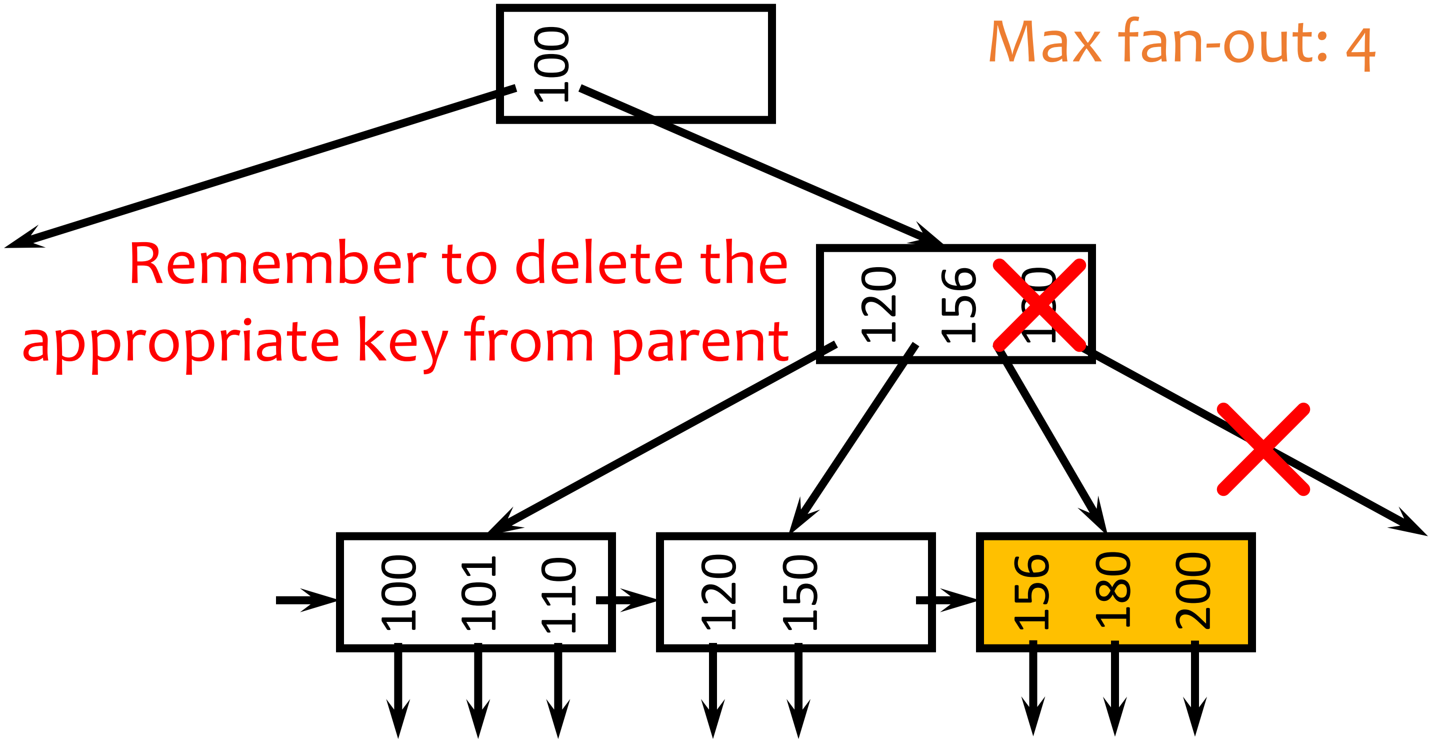


Another deletion example

- Delete a record with search key value 179



Coalescing



- Deletion can "propagate" all the way up to the root of the tree (not illustrated here)
  - When the root becomes empty, the tree "shrinks" by one level

Performance analysis

- How many I/O's are required for each operation?
    - h, the height of the tree (more or less)
    - Plus one or two to manipulate actual records
    - Plus $O(h)$ for reorganization (rare if f is large)
    - Minus one if we cache the root in memory
- How big is h?
    - Roughly $\log_{\text{fanout}} N$, where N is the number of records
    - B+-tree properties guarantee that fan-out is least $f/2$ for all non-root nodes
    - Fan-out is typically large (in hundreds)—many keys and pointers can fit into one block
    - A 4-level B+-tree is enough for "typical" tables

B^+-tree in practice

- Complex reorganization for deletion often is not implemented (e.g., Oracle)
  - Leave nodes less than half full and periodically reorganize
- Most commercial DBMS use B^+-tree instead of hashing-based indexes because B^+-tree handles range queries
  - A key difference between hash and tree indexes!

The Halloween Problem

- Story from the early days of System R…

```
UPDATE Payroll
SET salary = salary * 1.1
WHERE salary <= 25000;
```

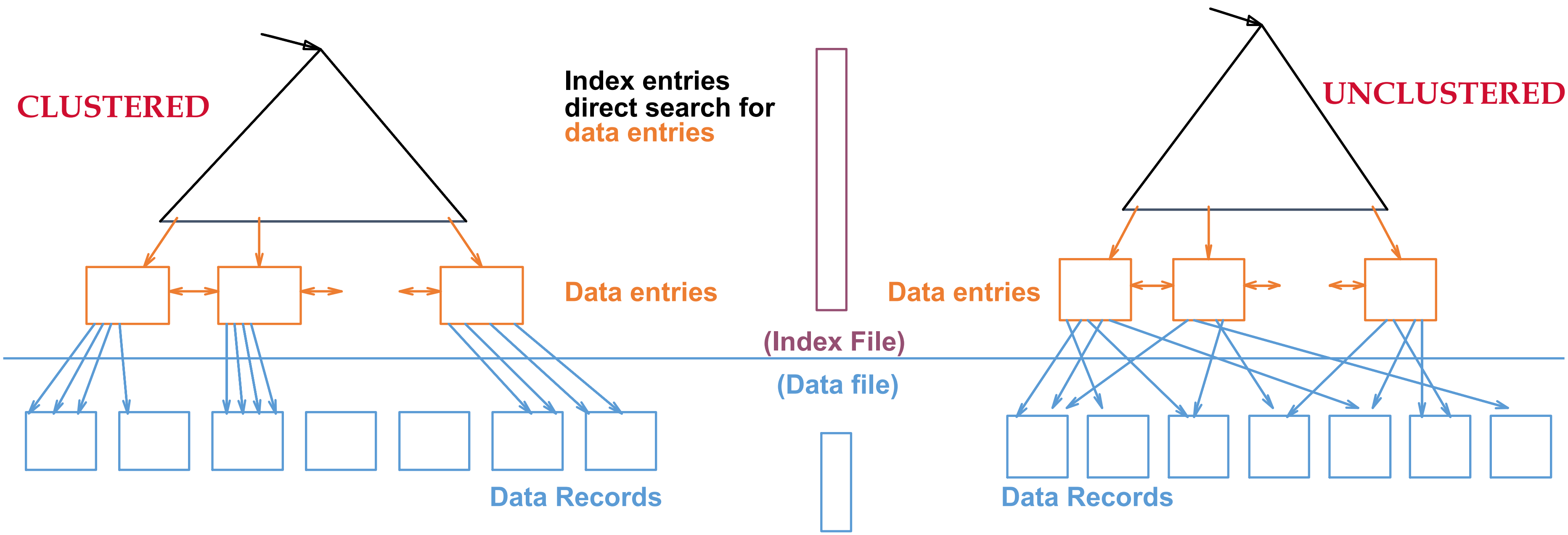
  - There is a B^+-tree index on *Payroll*(*salary*)
  - All employees end up earning >= 25000 (why?)
- Solutions?
  - Scan index in reverse, or
  - Before update, scan index to create a "to-do" list, or
  - During update, maintain a "done" list, or
  - Tag every row with transaction/statement id

https://en.wikipedia.org/wiki/Halloween_Problem

Clustered vs. Unclustered Index

- If order of data records in a file is the same as, or `close to', order of data entries in an index, then clustered, otherwise unclustered

- How does it affect # of page accesses? (in class)



Data is sorted on search key   Data can be anywhere

Clustered vs. Unclustered Index

- How does it affect # of page accesses?
  - Recall disk-memory diagram!

- SELECT * FROM USER WHERE age = 50
  - Assume 12 users with age = 50
  - Assume one data page can hold 4 User tuples
  - Suppose searching for a data entry requires 3 IOs in a B+-tree, which contain pointers to the data records (assume all matching pointers are in the same node of B+-tree)

  - What happens if the index is unclustered?
  - What happens if the index is clustered?

FYI – not covered in this class

Beyond ISAM, B-trees, and B^+-trees

- Other tree-based indexes: R-trees and variants, GiST, etc.
- Hashing-based indexes: extensible hashing, linear hashing, etc.
- Text indexes: inverted-list index, suffix arrays, etc.
- Other tricks: bitmap index, bit-sliced index, etc.

Hash vs. Tree Index

Need to know only this much
About hash indexes in this class

- Hash indexes can only handle equality queries
  - SELECT * FROM R WHERE age = 5 (requires hash index on (age))
  - SELECT * FROM R, S WHERE R.A = S.A (requires hash index on R.A or S.A)
  - SELECT * FROM R WHERE age = 5 and name = ‘Bart’ (requires hash index on (age, name))

- **(-)** Cannot handle range queries or prefixes
  - SELECT * FROM R WHERE age >= 5
  - need to use tree indexes (more common)
  - Tree index on (age), or (age, name) works, but not (name, age) – why?

- **(+)** Hash-indexes are more amenable to parallel processing
  - Will learn more in hash-based join

- Performance depends on how good the hash function is (whether the hash function distributes data uniformly and whether data has skew)

Trade-offs for Indexes

- Should we use as many indexes as possible?

Trade-offs for Indexes

- Should we use as many indexes as possible?

- Indexes can make
  - queries go faster
  - updates slower

- Require disk space, too

Index-Only Plans

- A number of queries can be answered without retrieving any tuples from one or more of the relations involved if a suitable index is available

```
SELECT  E.dno, COUNT(*)
FROM  Emp E
GROUP BY  E.dno
```

<E.dno>

```
SELECT  E.dno, MIN(E.sal)
FROM  Emp E
GROUP BY  E.dno
```

<E.dno,E.sal>

Tree index!

<E. age,E.sal>

Tree index!

```
SELECT AVG(E.sal)
FROM  Emp E
WHERE  E.age=25 AND
  E.sal BETWEEN 3000 AND 5000
```

- If you have an index on E.dno in the above query, no need to access data
- For index-only strategies, clustering is not important