Indexing

Introduction to Databases

CompSci 316 Spring 2020

DUKE
COMPUTER SCIENCE

Announcements (Thu., Feb 27)

- Private project threads created on piazza
  - Please check you are on your thread
  - Primary and secondary project mentors to be assigned soon
  - They will give you feedback on MS1, check updates, and help you as needed
  - Feel free to discuss projects in all TA office hours

- Project updates to be posted **every Monday**
  - Starts Monday 03/02: each member should say what you are supposed to do for the rest of the semester and also for MS2
  - Make sure that your primary TA says "sounds good" in the response to each update, otherwise do as they suggest
  - Other team members will also check the updates and respond to the threads as needed if there are confusions or clarifications
  - Try to resolve conflicts/concerns within group whenever possible, otherwise reach out to your TAs and me early

Announcements - contd (Thu., Feb 27)

- Homework #4 published due next Wednesday 03/04
  - One submission per project group
  - See updates on piazza about submitting the link to your website

- Let me know if you want to meet me about midterm or anything else
  - Final exam will be similar in nature (problem-solving based), but the length/difficulty/question types may vary
  - Comprehensive with more focus on topics after midterm
  - How to prepare? Think about what we discuss in class and ask me tough questions!

- Heads up: (almost) weekly quiz or lab **every Thursday**
  - For practicing problems for the final
  - In groups, but individual submissions
  - Quizzes are shorter and discussed in class, Labs are longer with extra time after class (extra credit for submitting within the class)

Today's lecture

- Index

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

Related

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

database indexing

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

123	
456	
857	

Sparse index on *uid*

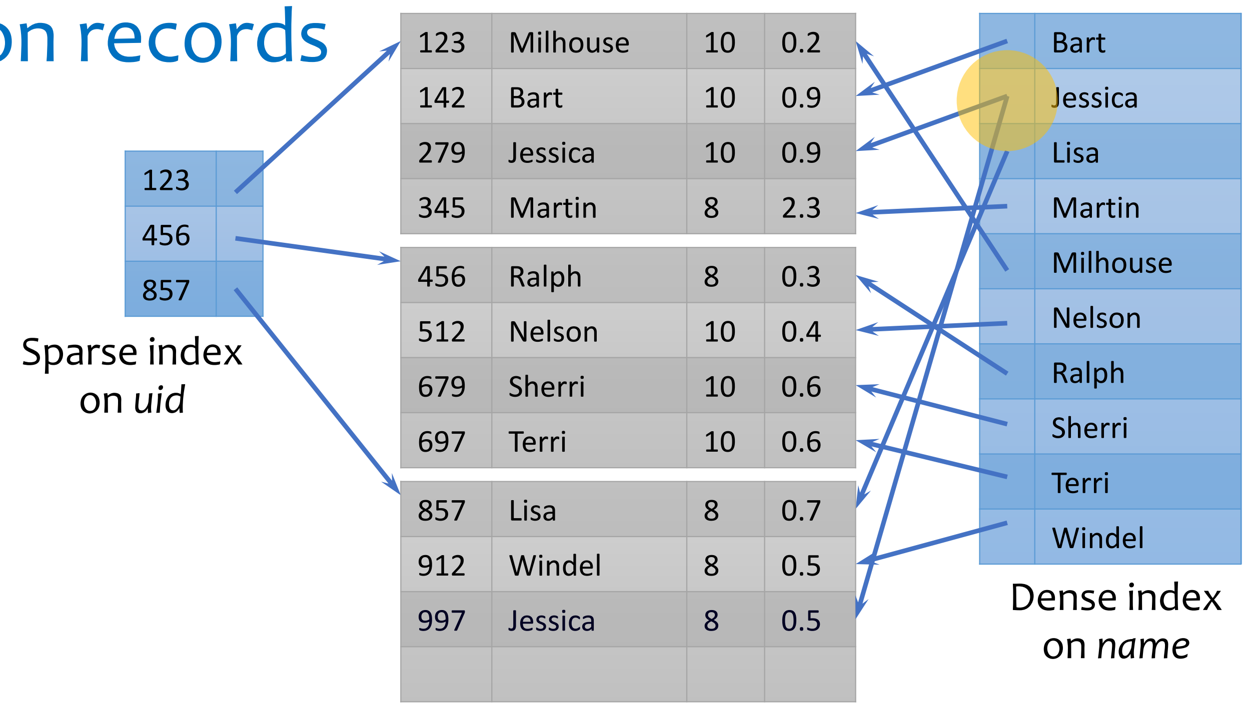
123	Milhouse	10	0.2
142	Bart	10	0.9
279	Jessica	10	0.9
345	Martin	8	2.3
456	Ralph	8	0.3
512	Nelson	10	0.4
679	Sherri	10	0.6
697	Terri	10	0.6
857	Lisa	8	0.7
912	Windel	8	0.5
997	Jessica	8	0.5

	Bart
	Jessica
	Lisa
	Martin
	Milhouse
	Nelson
	Ralph
	Sherri
	Terri
	Windel

Dense index on *name*

Dense versus sparse indexes

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



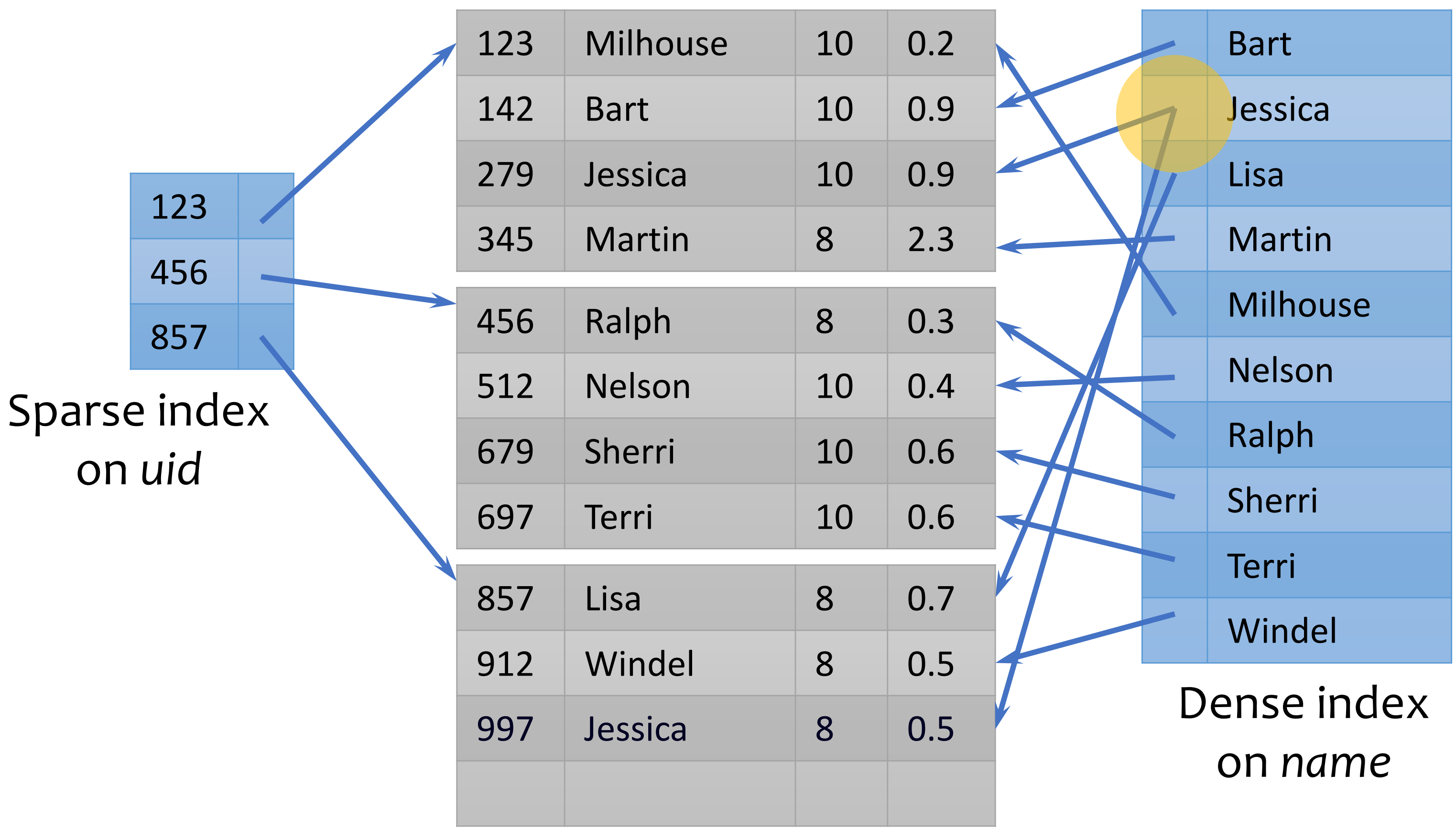
Dense versus sparse indexes

- 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
  - Easier for sparse index

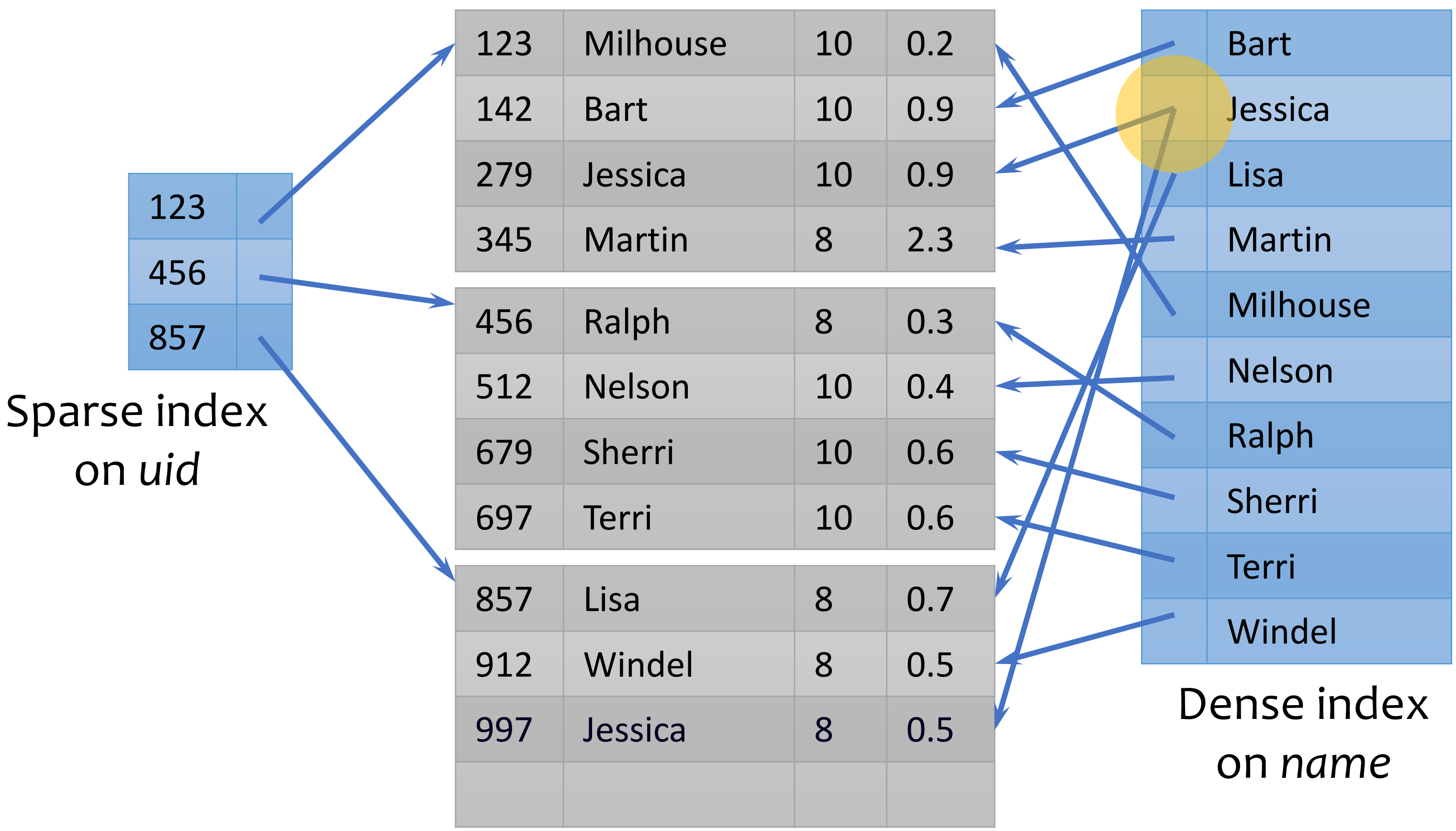
Primary and secondary indexes

- 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?

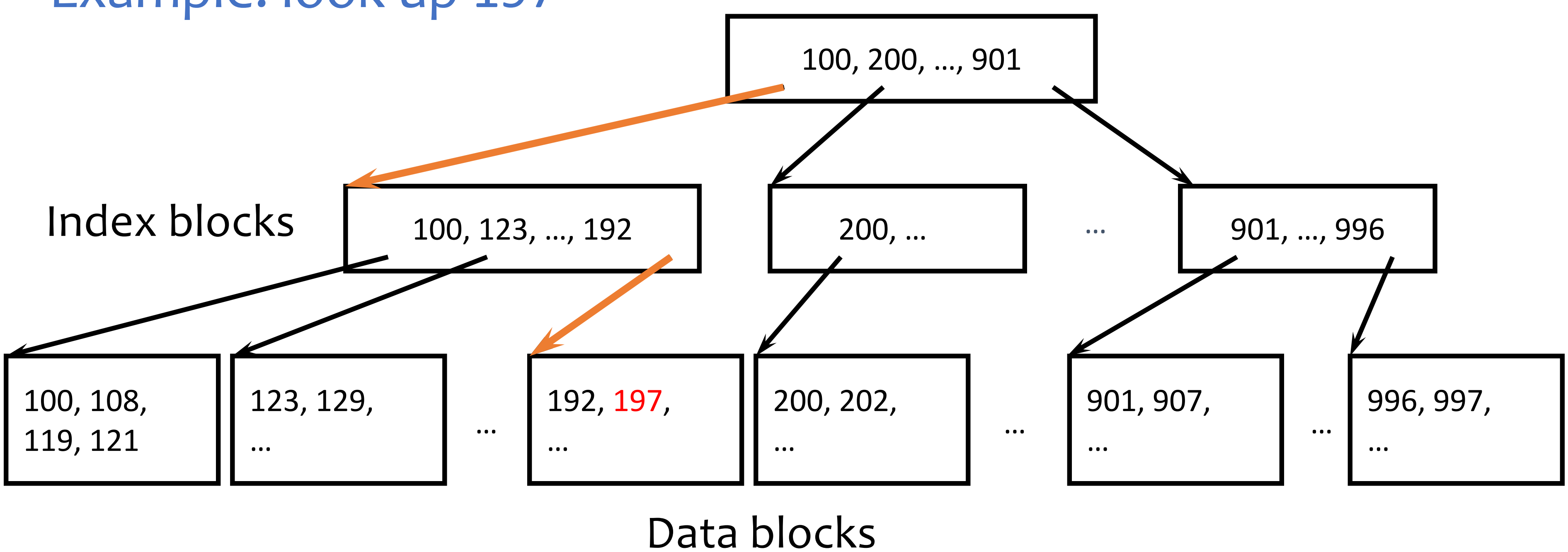


Put a another (sparse) index on top of that!

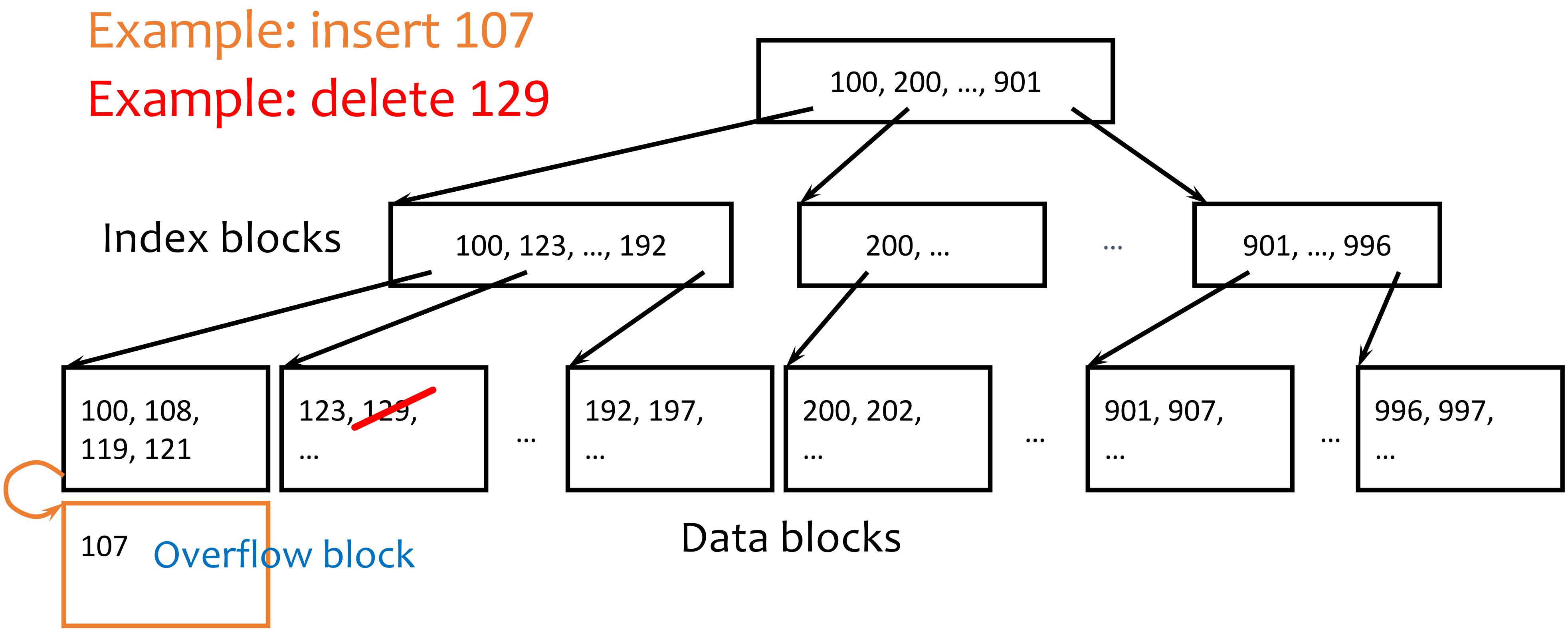
ISAM

- 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



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

Start: Tuesday 03/03

Announcements (Tue., Mar 03)

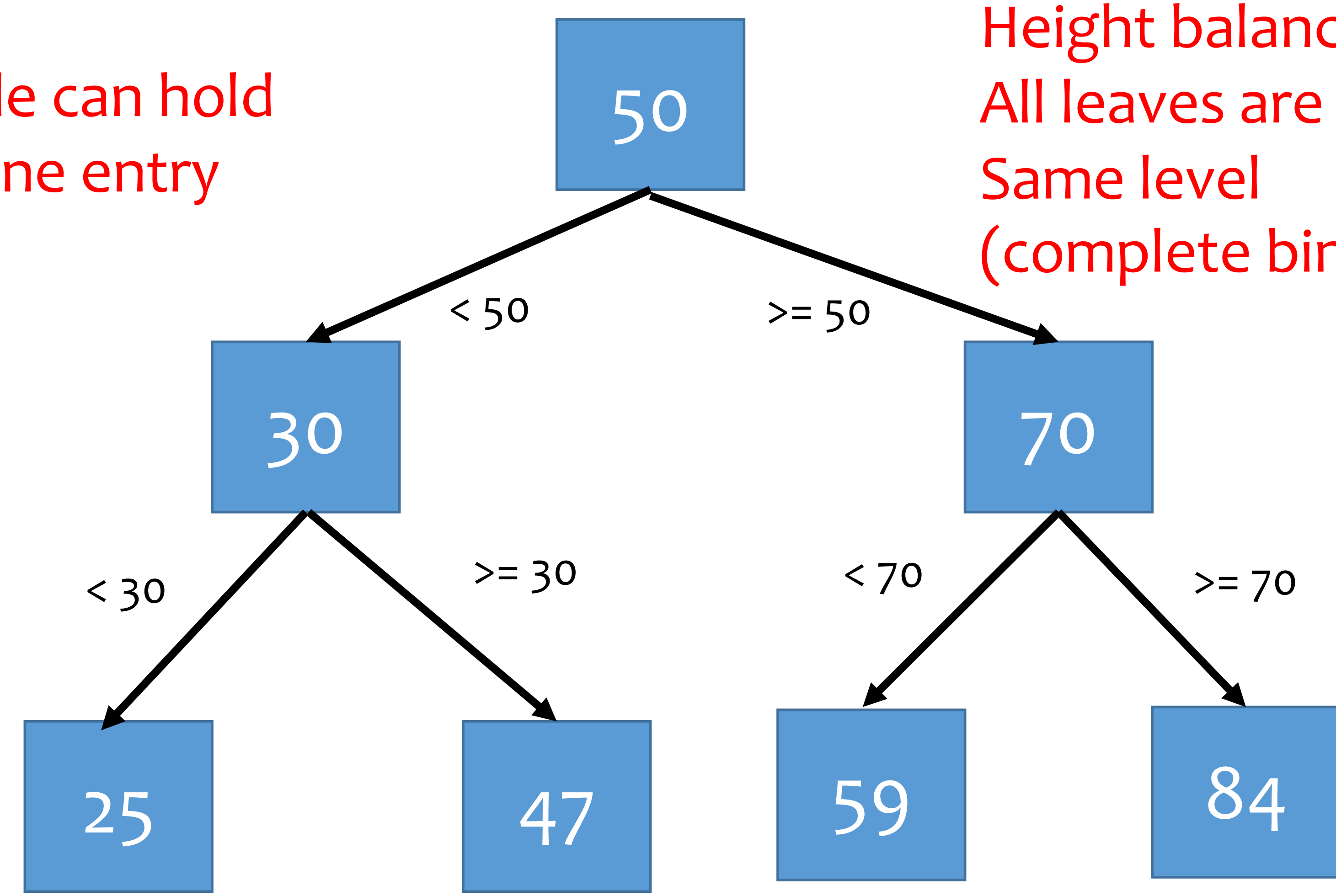
- Reminder: HW4 due tomorrow (Wed 03/04)
  - One group submission per group on gradescope
  - Mention all group members' names

- Reminder: Lab-2 (index) on Thursday in class

Binary Search Tree

Each node can hold
Exactly one entry

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

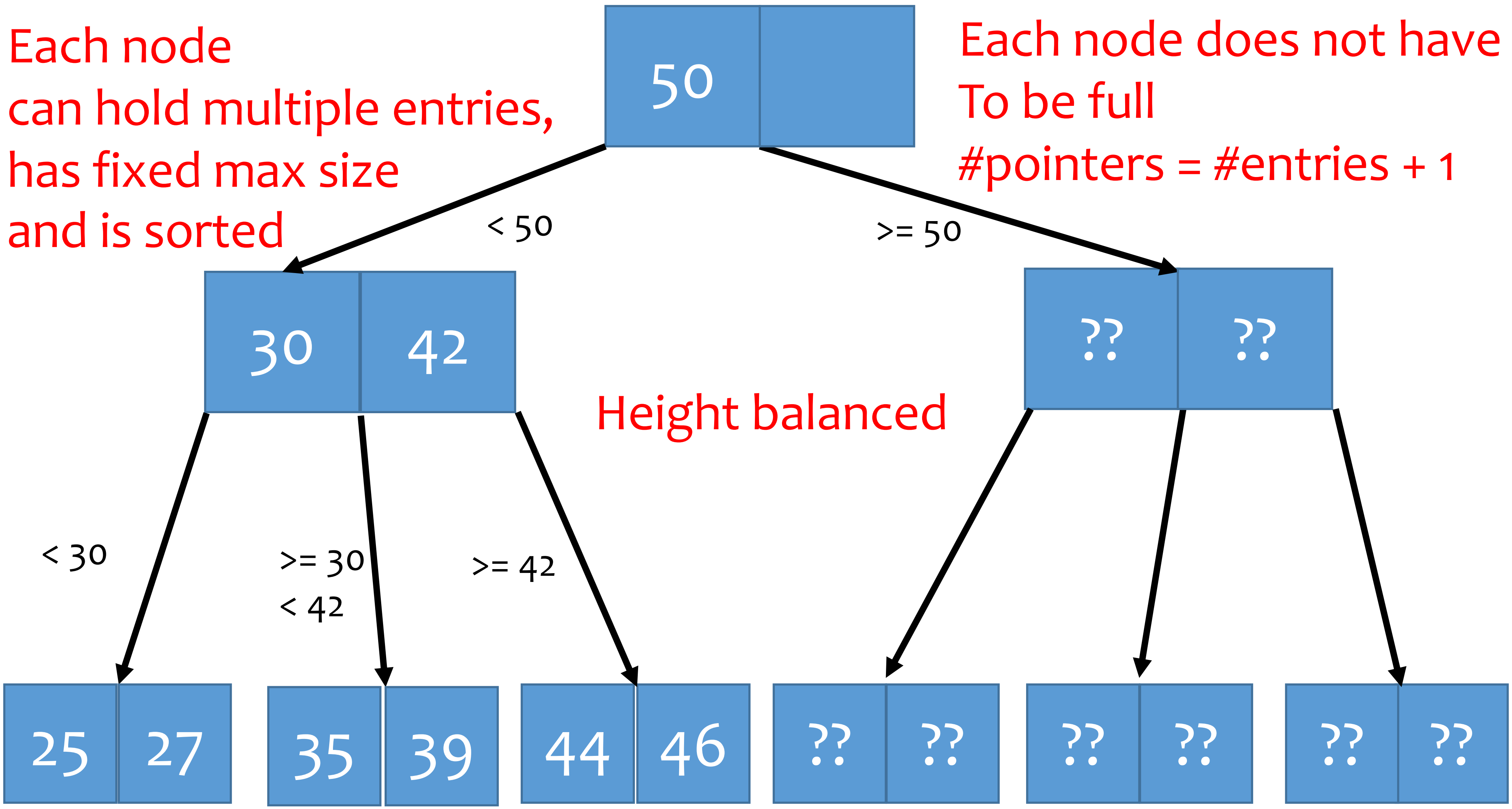


Leaves are sorted

B-tree: Generalizing Binary Search Trees

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

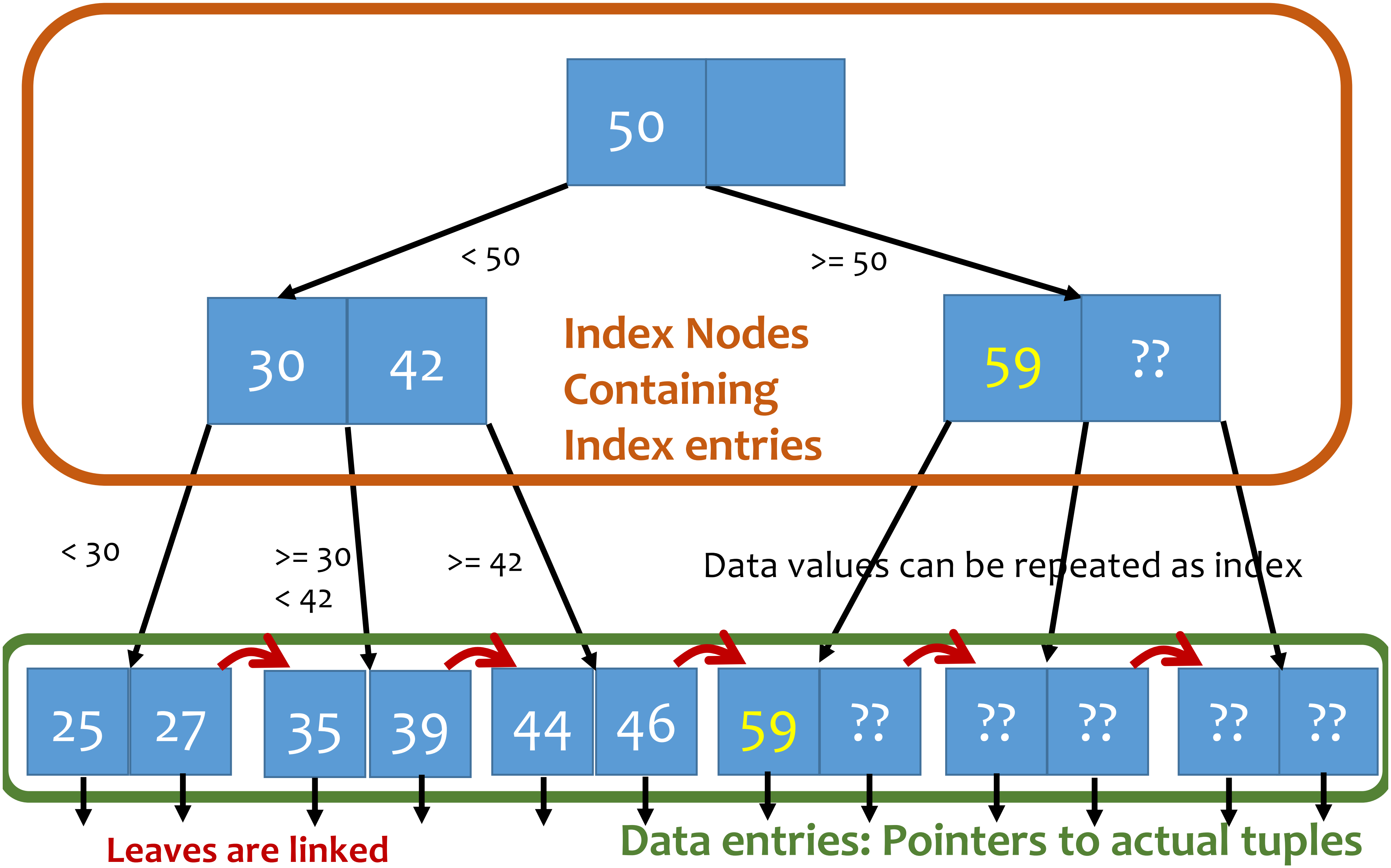
Each node does not have
To be full
#pointers = #entries + 1



Height balanced

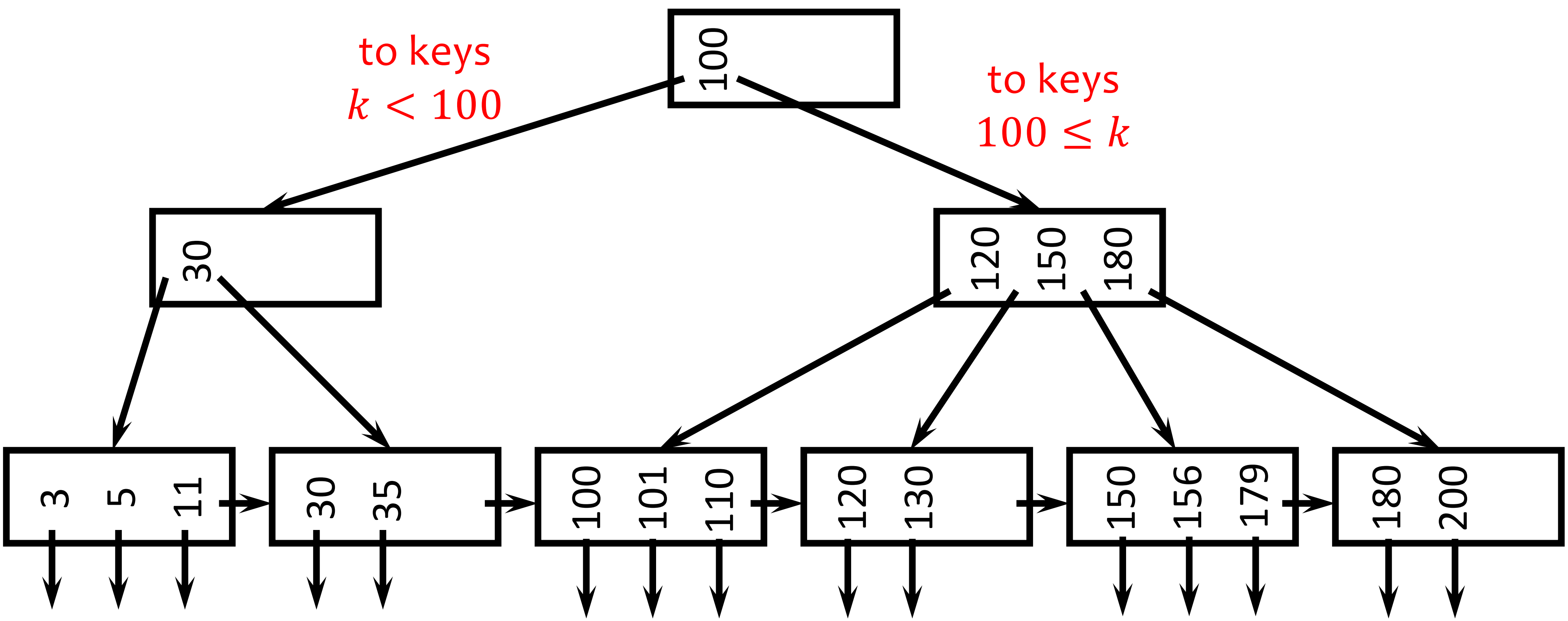
Leaves are sorted

B+-tree: Data only at leaves

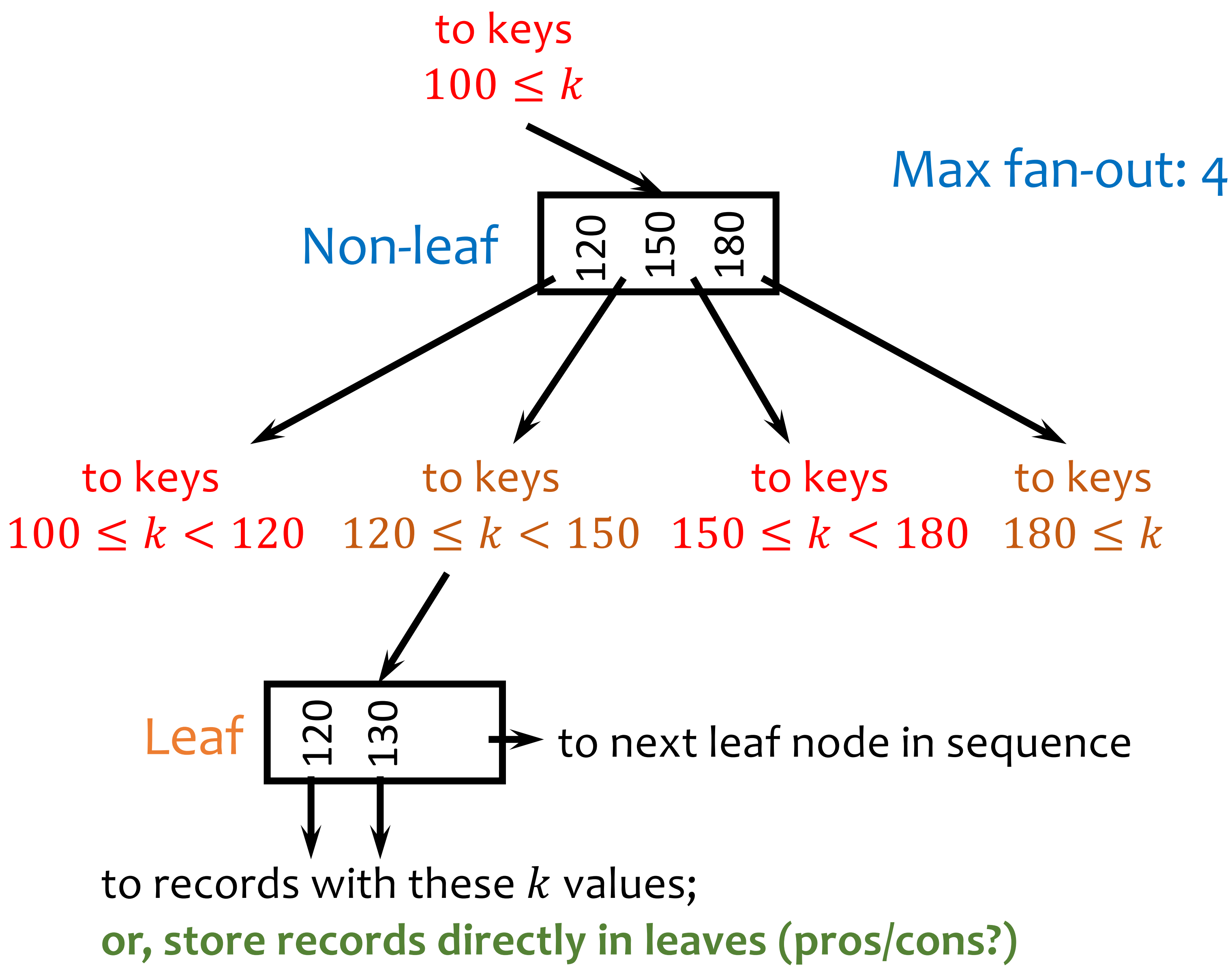


B+-tree: Closer Look

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



Sample B$^+$-tree nodes



- Question (discuss with your neighbor):

- 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

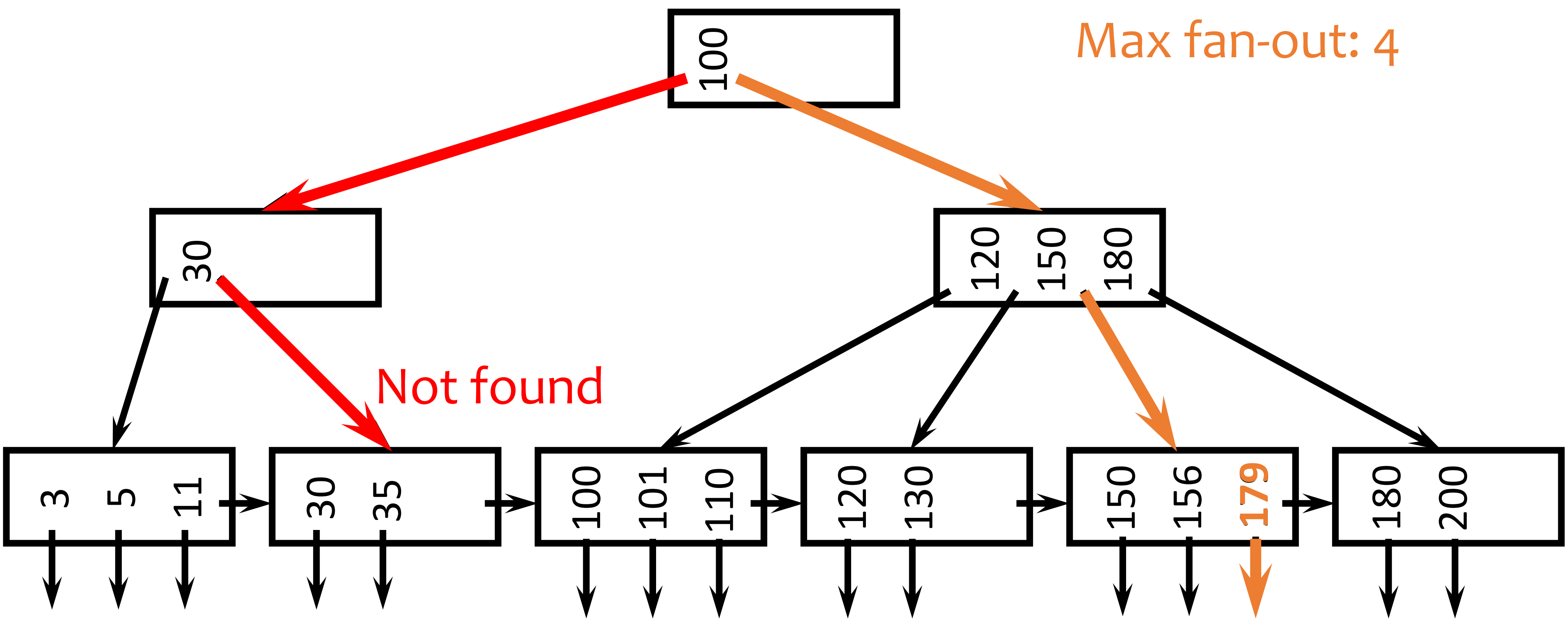
Check yourself

- 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

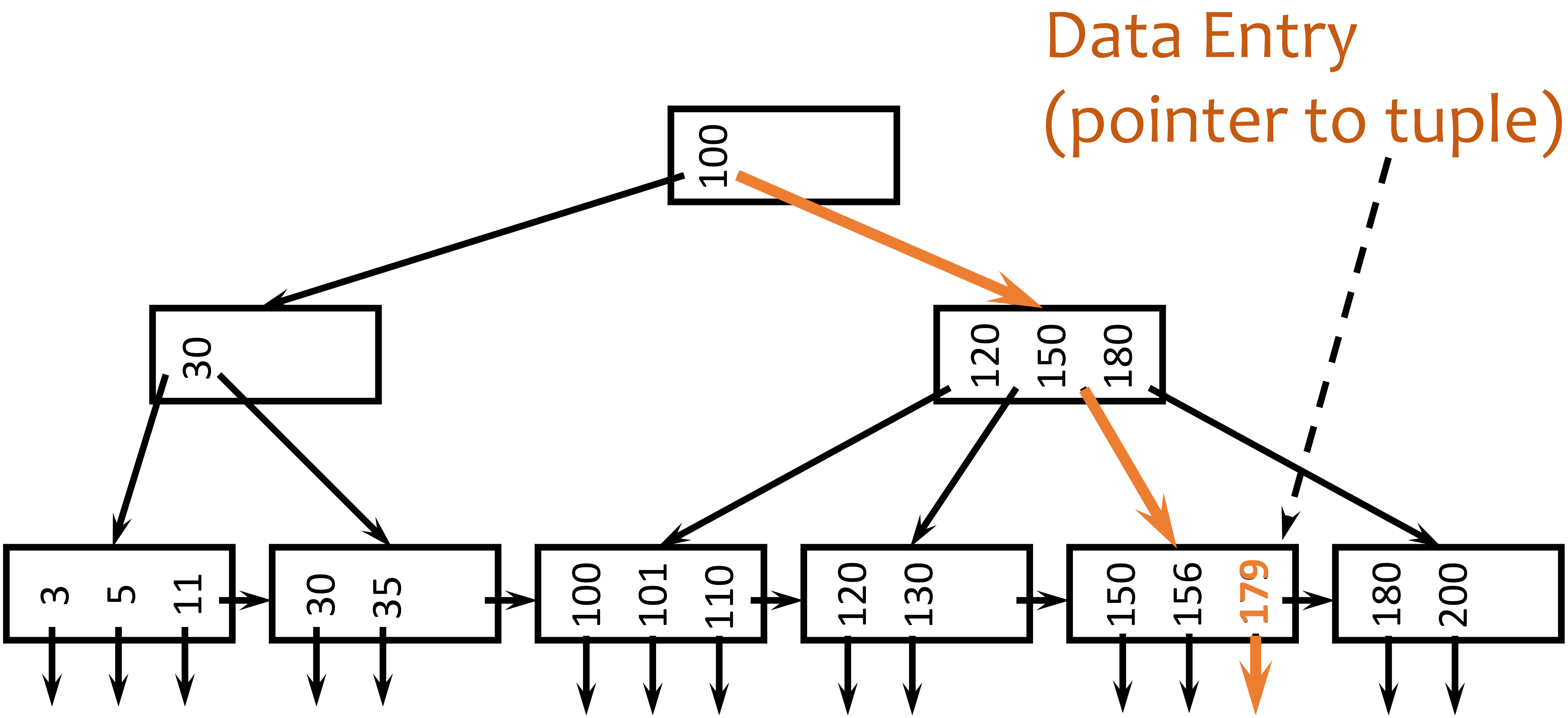
- SELECT * FROM *R* WHERE *k* = 179;
- SELECT * FROM *R* WHERE *k* = 32;



Search key and Data entry

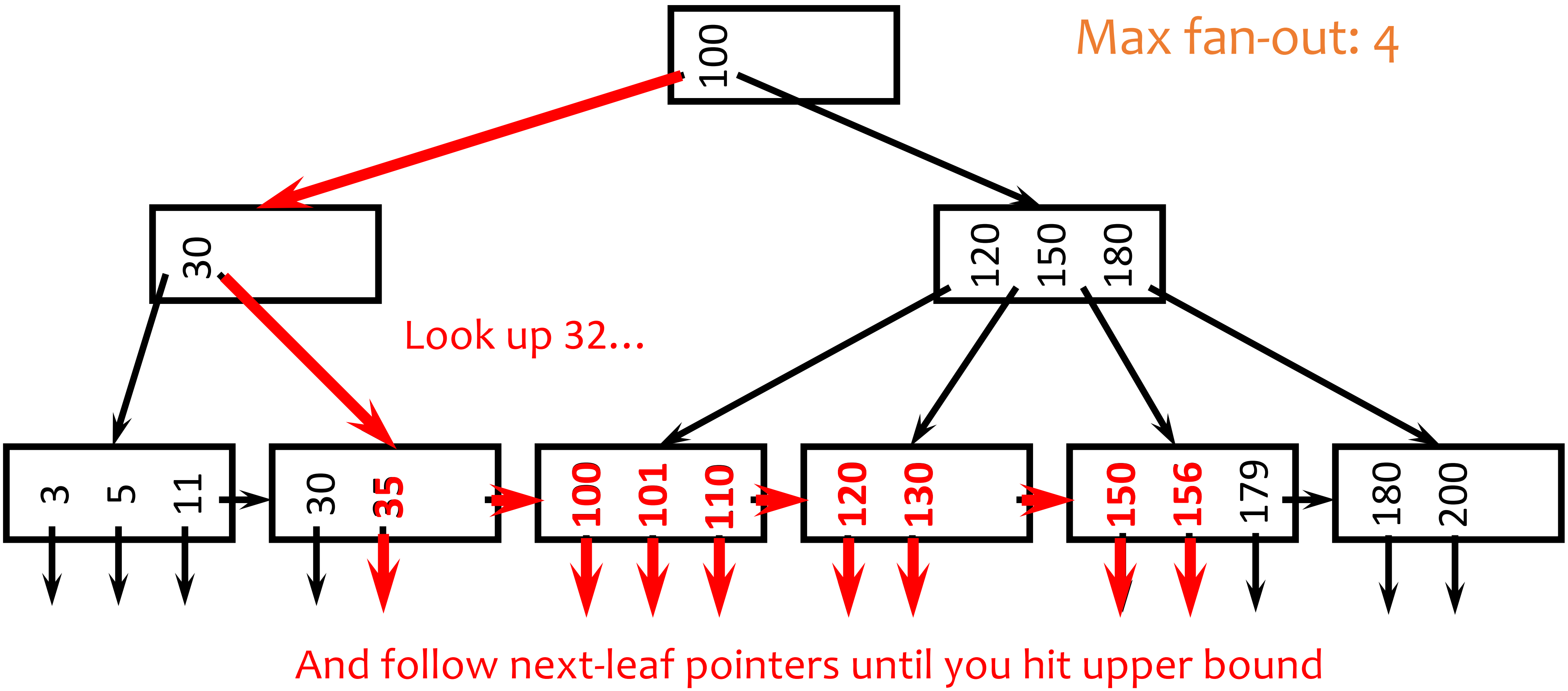
- SELECT * FROM *R* WHERE *k* = 179;

Search key (value)



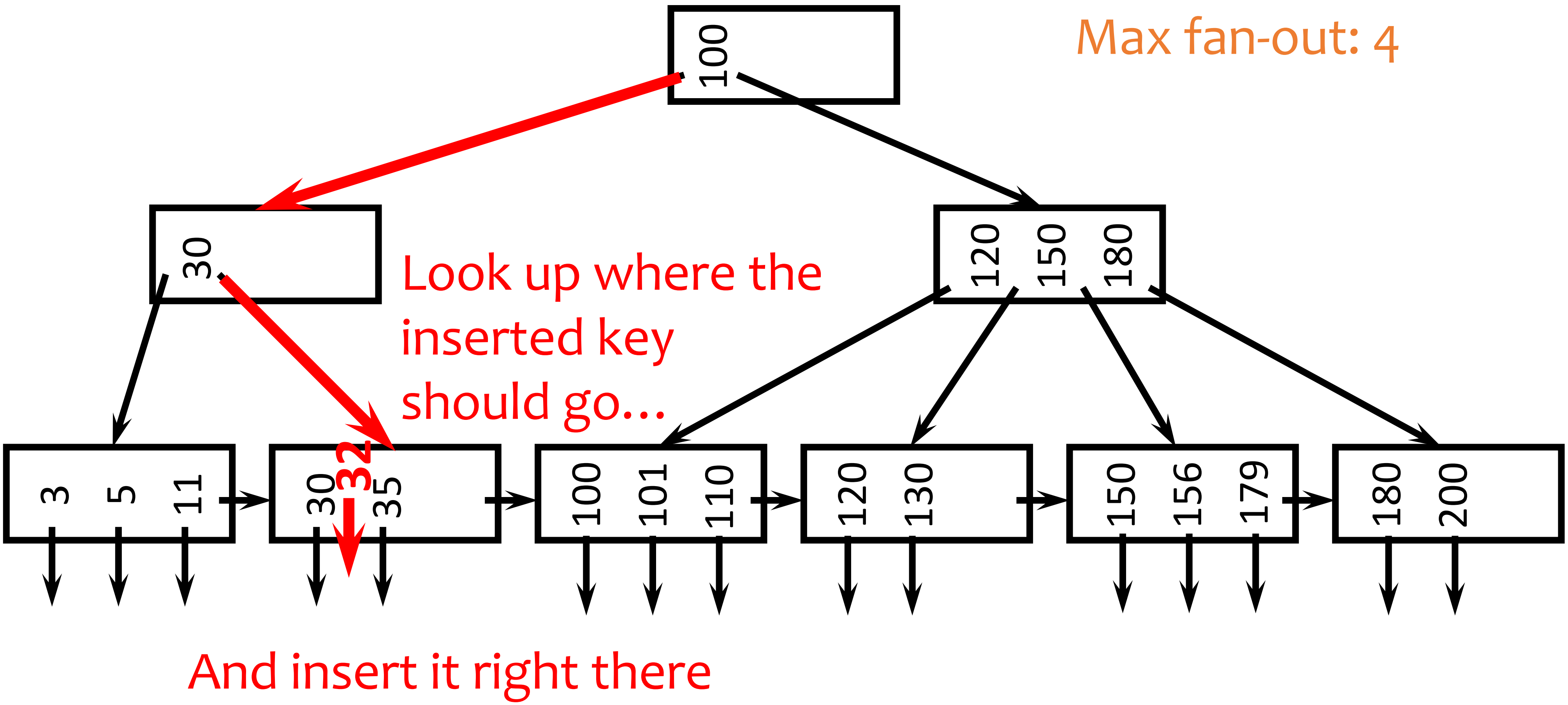
Range query

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



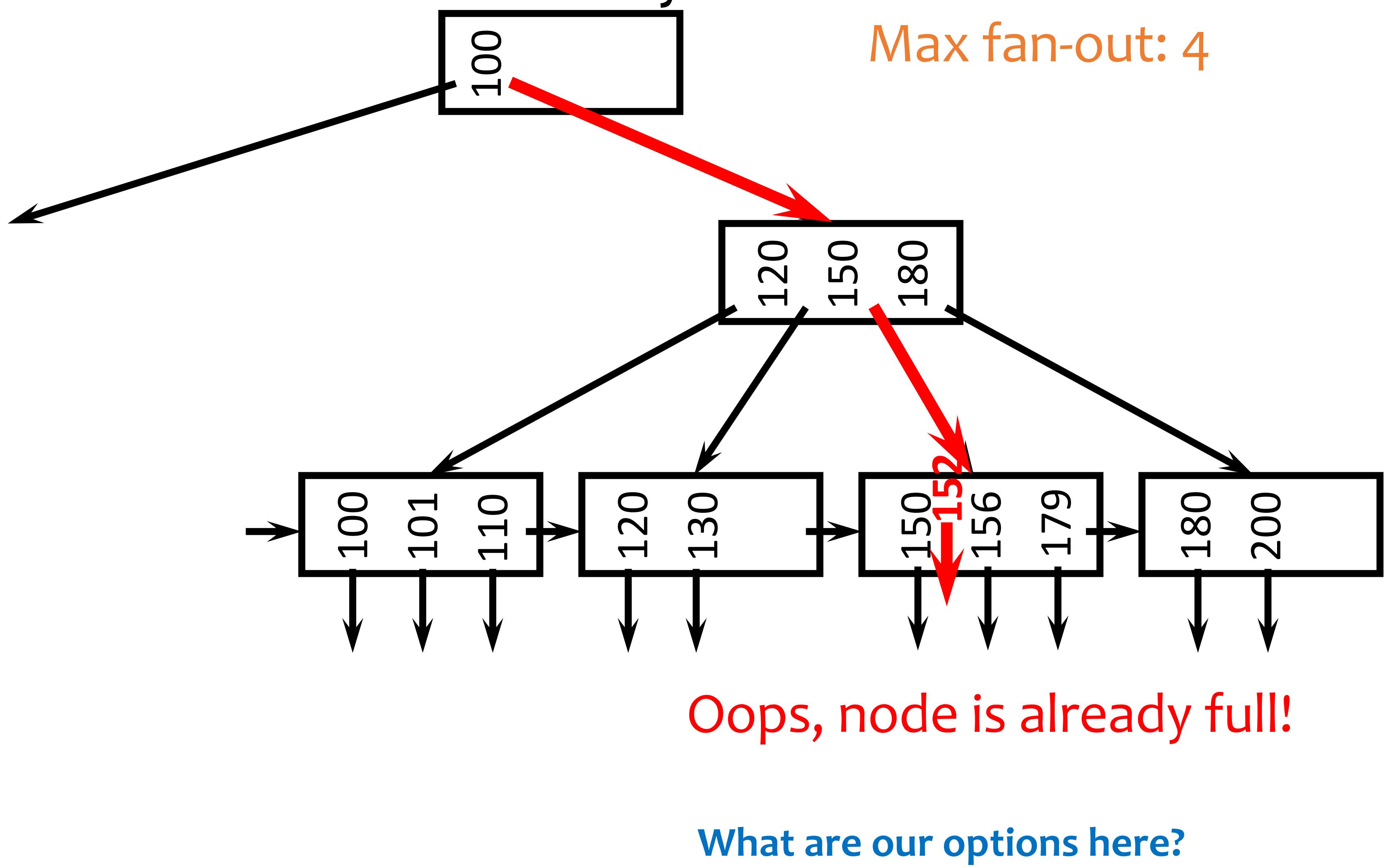
Insertion

- Insert a record with search key value 32

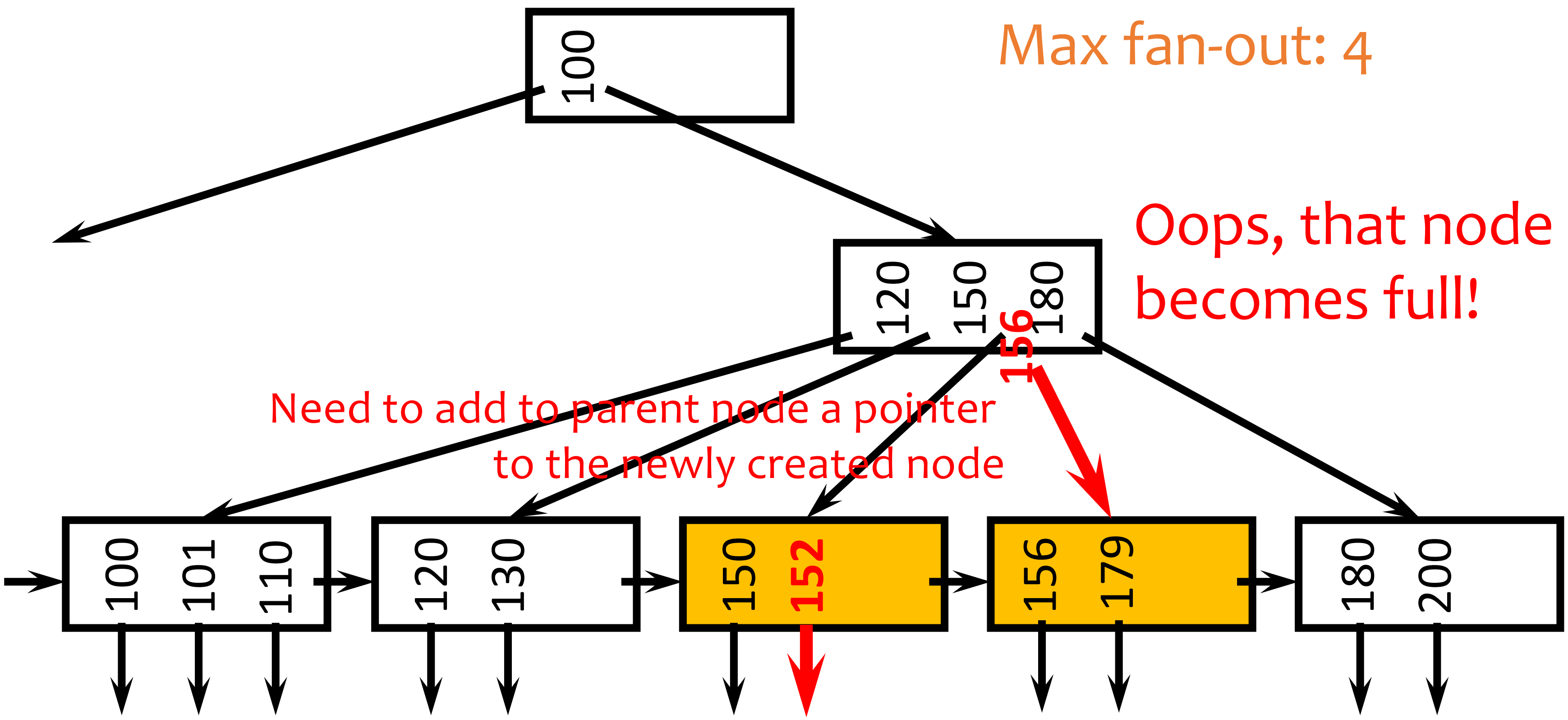


Another insertion example

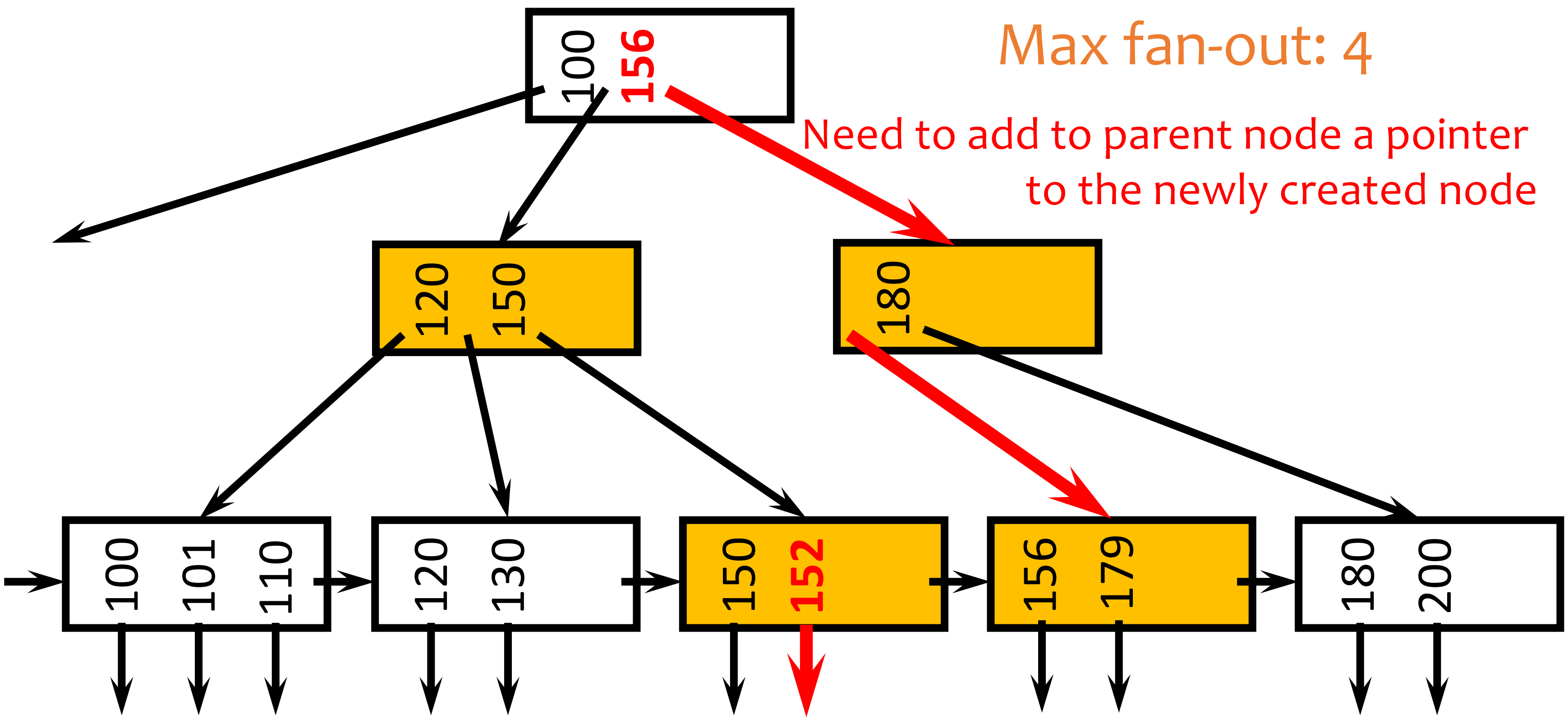
- Insert a record with search key value 152



Node splitting



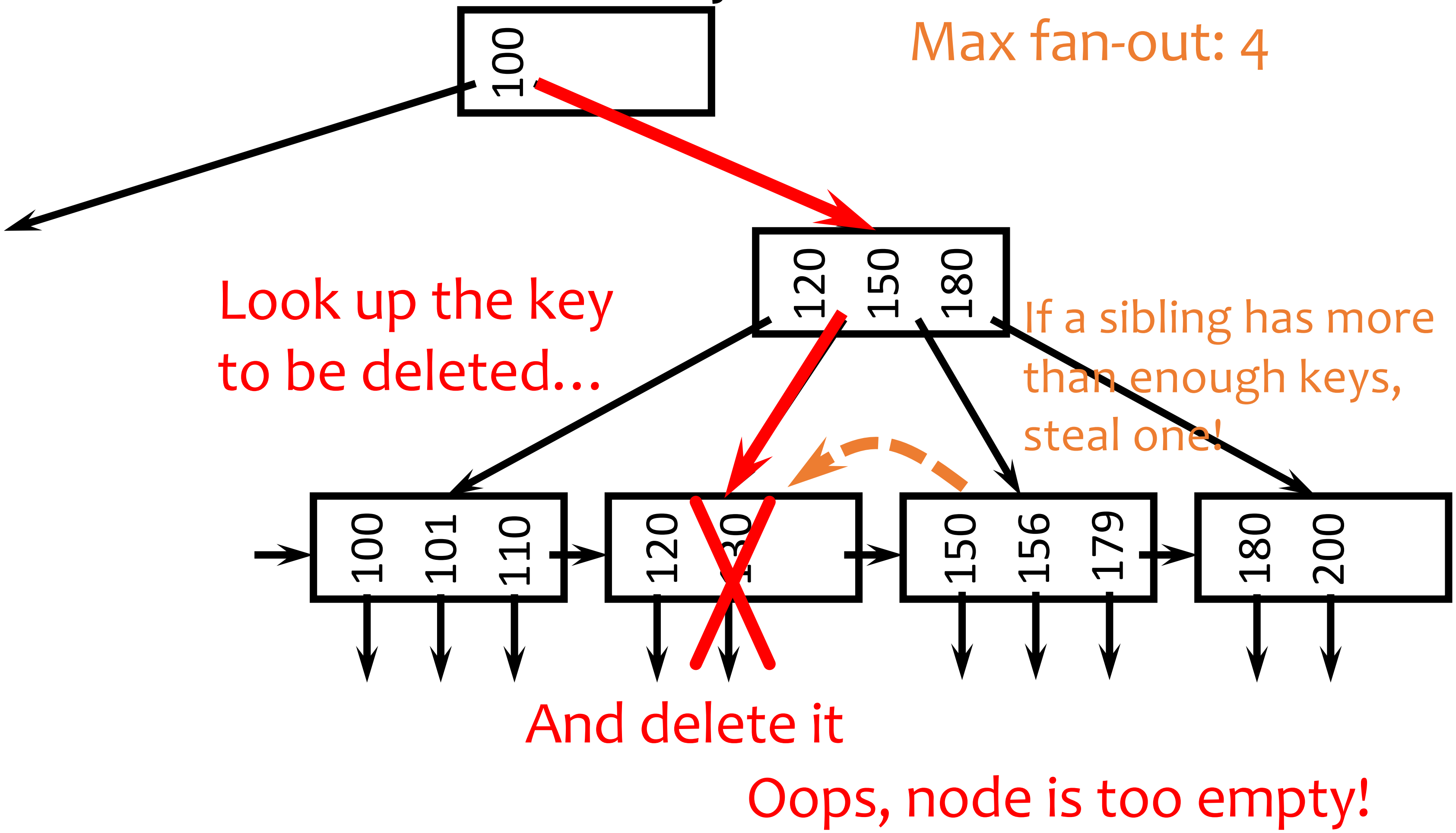
More node splitting



- 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

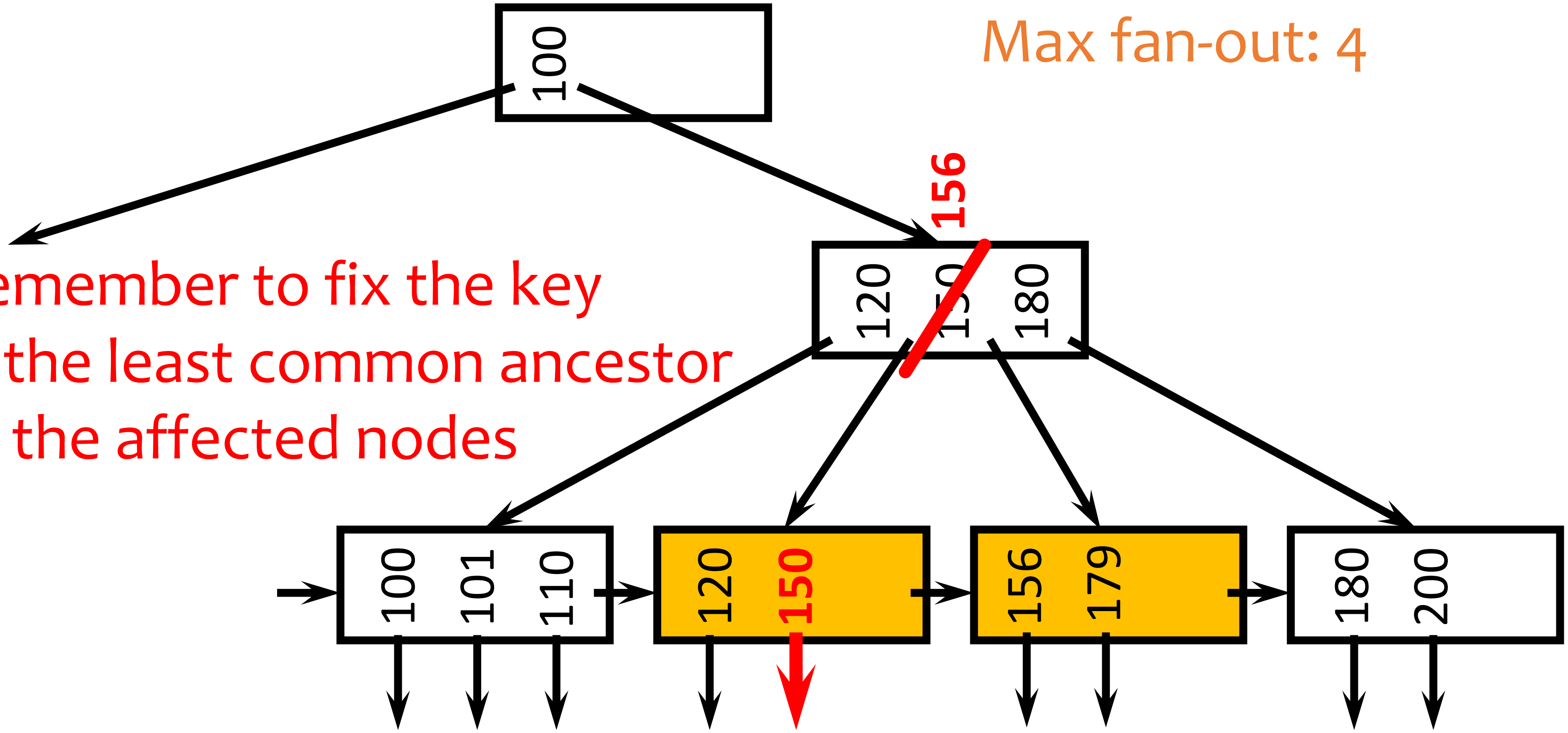
Deletion

- Delete a record with search key value 130



Stealing from a sibling

Max fan-out: 4

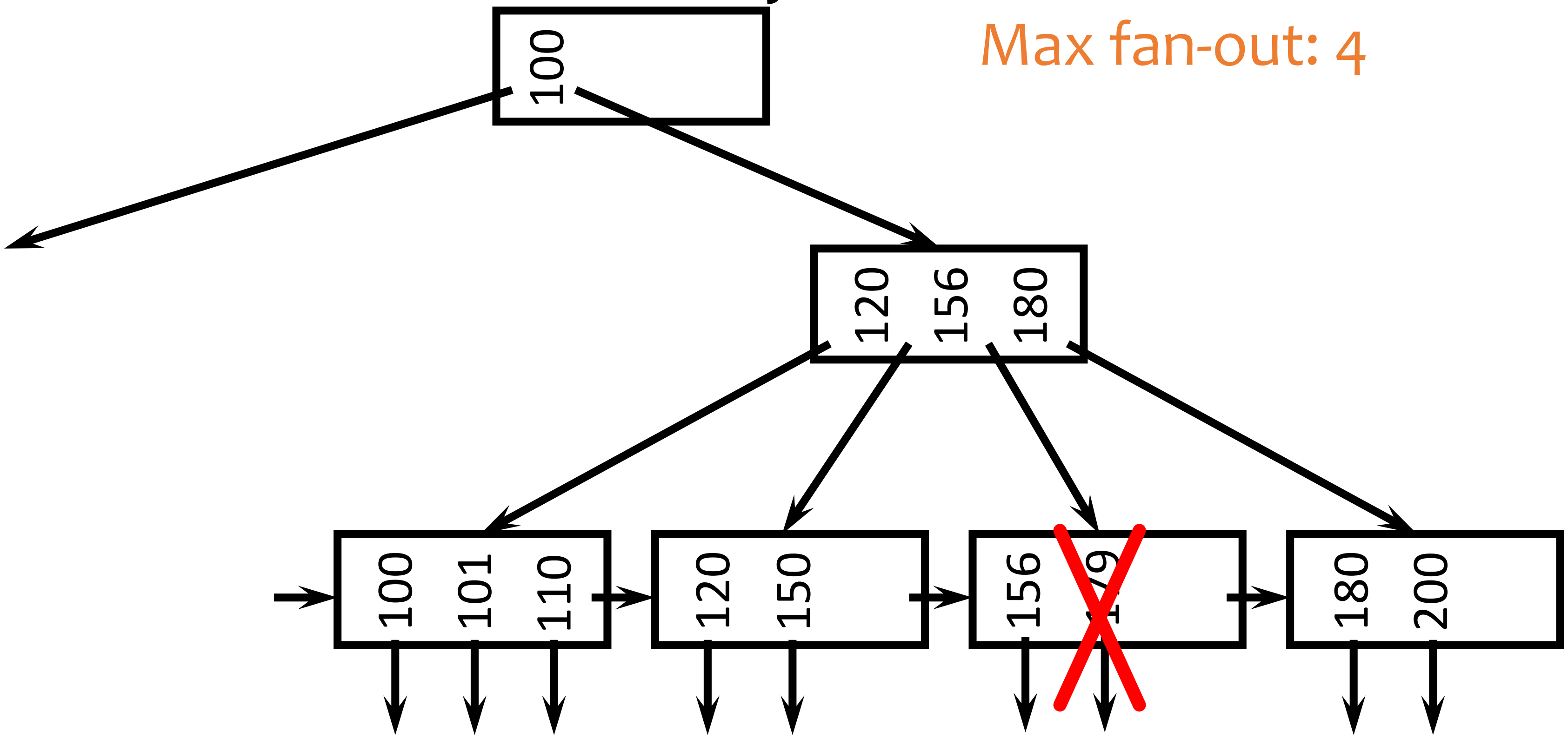


Remember to fix the key
in the least common ancestor
of the affected nodes

Another deletion example

- Delete a record with search key value 179

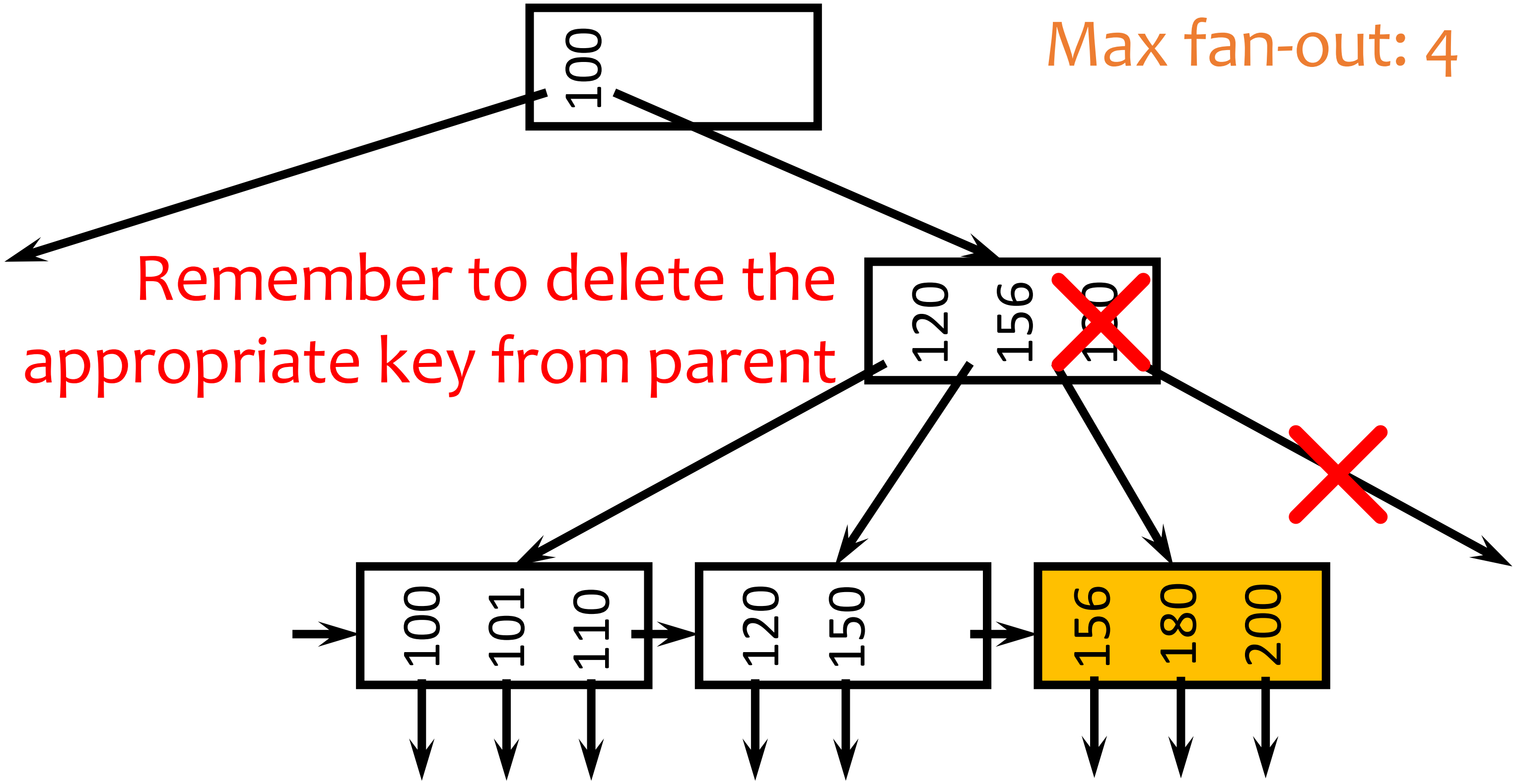
Max fan-out: 4



Cannot steal from siblings

Then coalesce (merge) with a sibling!

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 >= 100000;

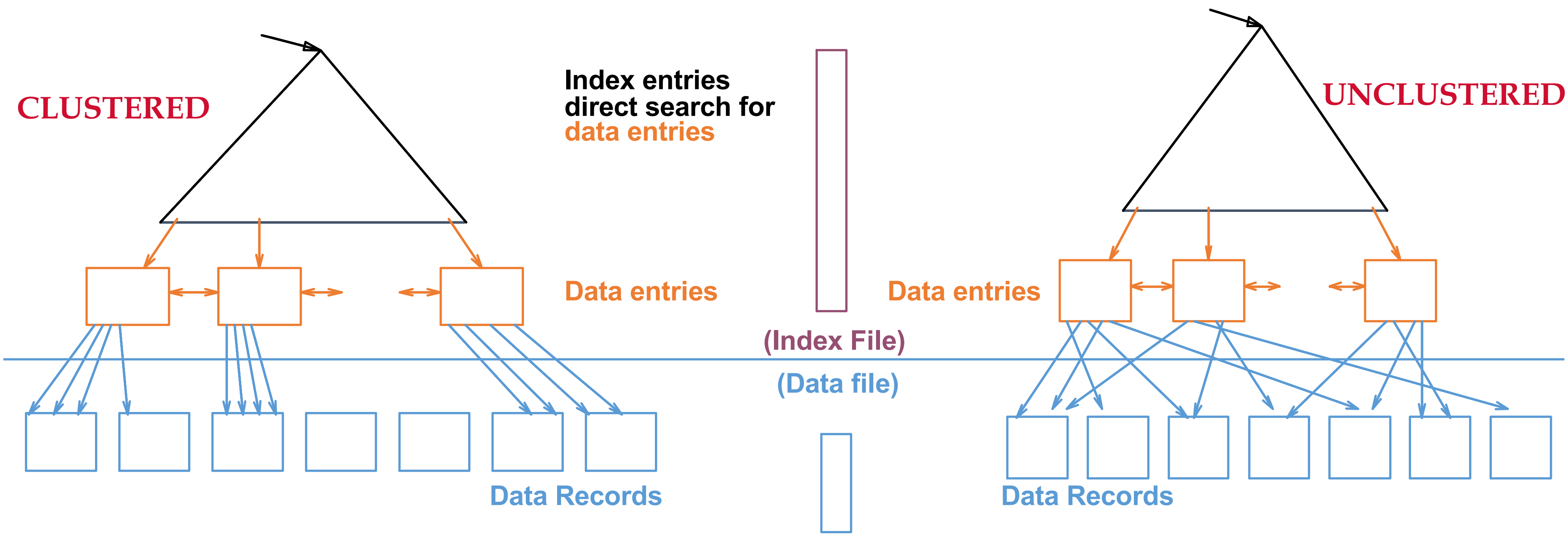
  - There is a B^+-tree index on *Payroll*(*salary*)
  - The update never stopped (why?)
- Solutions?
  - Scan index in reverse, or
  - Before update, scan index to create a "to-do" list, or
  - During update, maintain a "done" list, or
  - Tag every row with transaction/statement id

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?
- (in class – discuss with your neighbors)

- 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?