Lecture 14: Indexing

CS348 Spring 2025:
Introduction to Database Management

Guest Lecture: **Chao Zhang**
Sections: 001, 002, 003

Announcements

- Assignment 2
  - Due today!

Outline

- Types of indexes
- Index structure
- How to use index

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*;

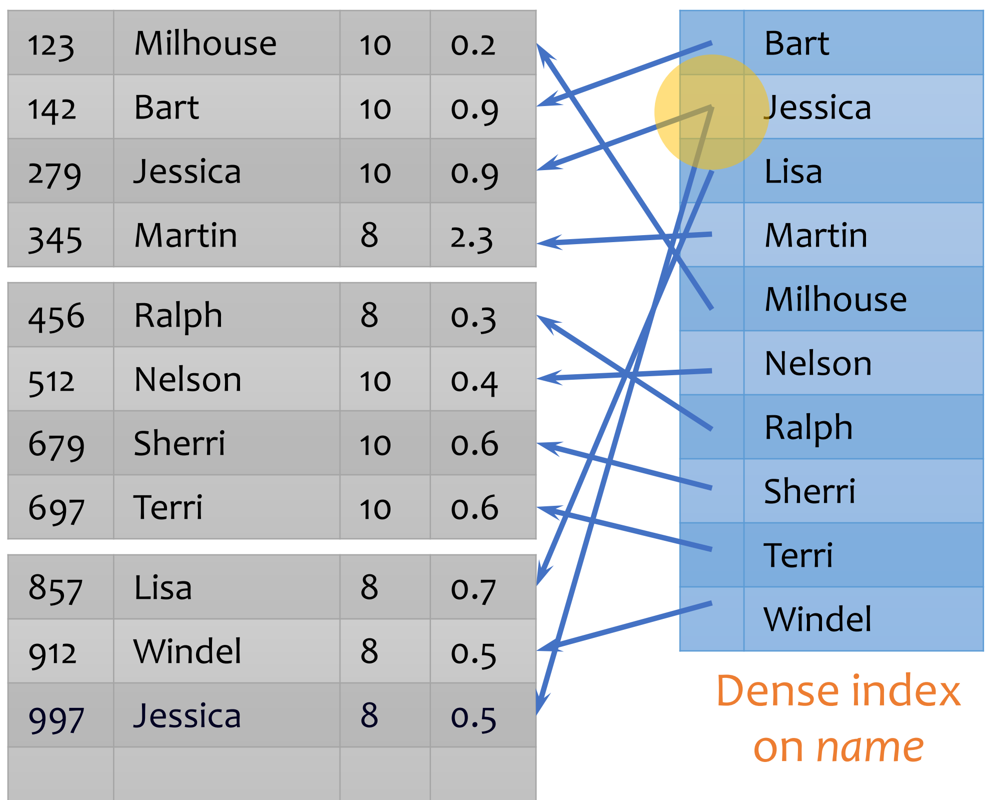
- Find data by other search criteria, e.g. range search

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

- We call A in the above example a *search key*
  - The attribute whose values will be indexed

Dense v.s. Sparse indexes

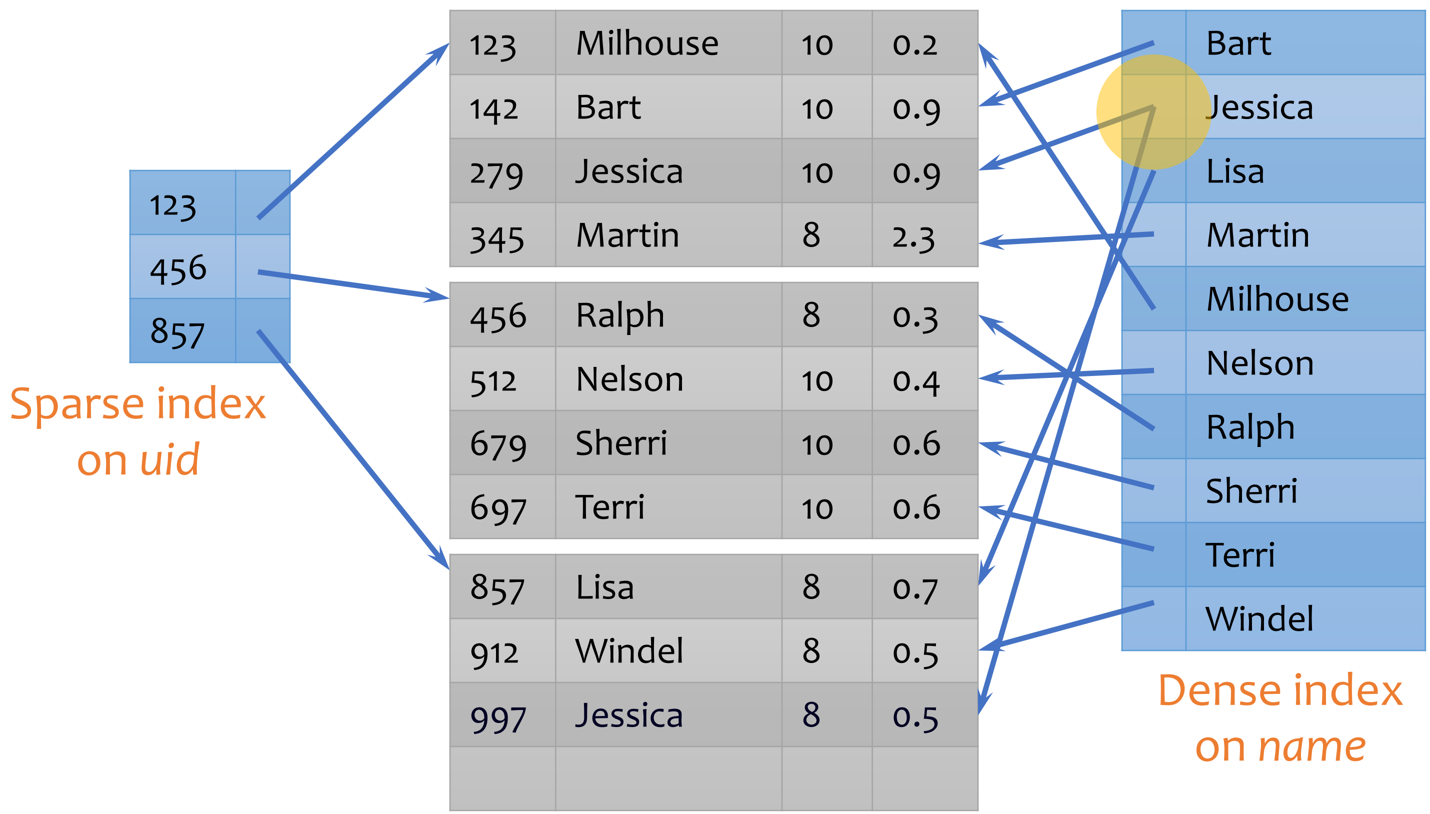
- Dense: one index entry for each search key value
  - One entry may "point" to multiple records (e.g., two users named Jessica)



Dense index on *name*

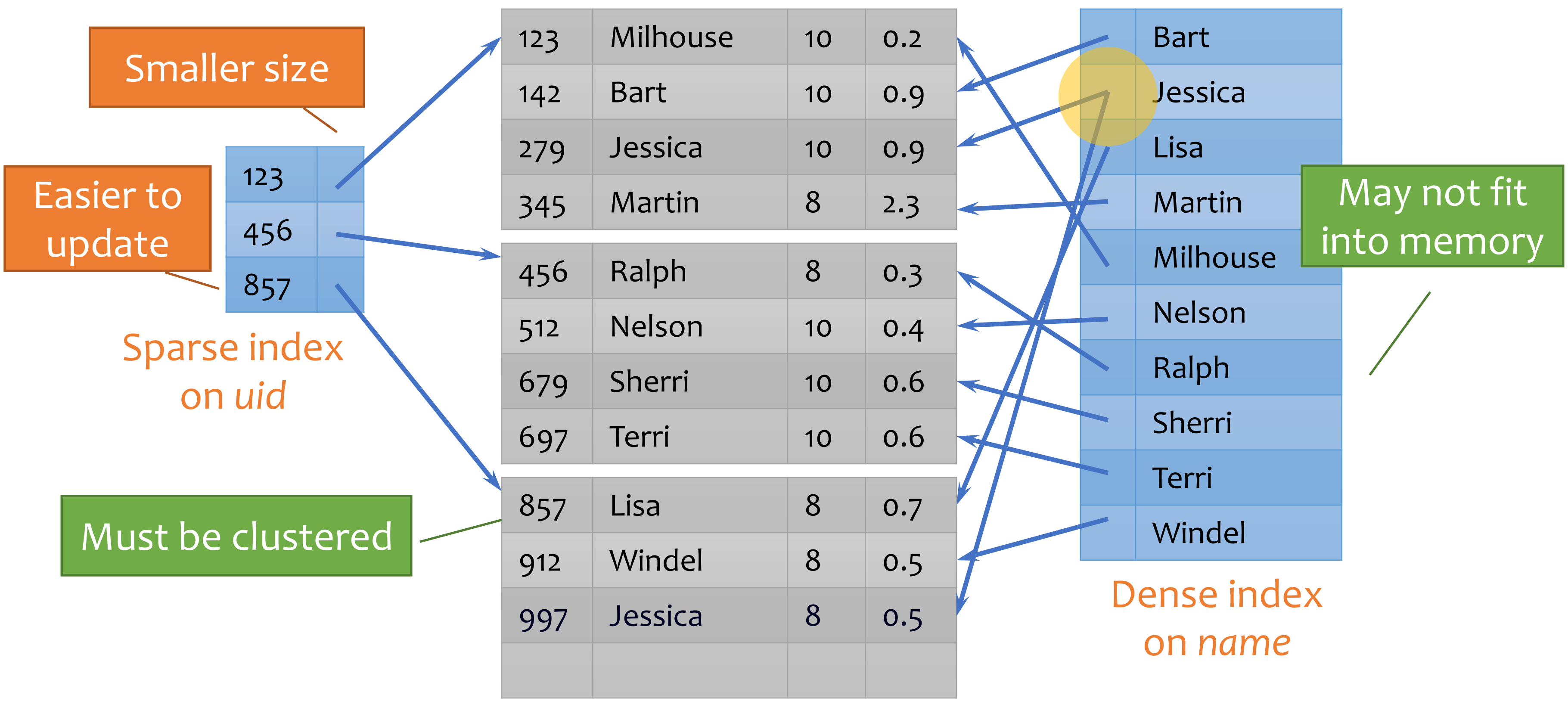
Dense v.s. Sparse indexes

- Sparse: one index entry for each block
  - Records must be clustered according to the search key on the disk



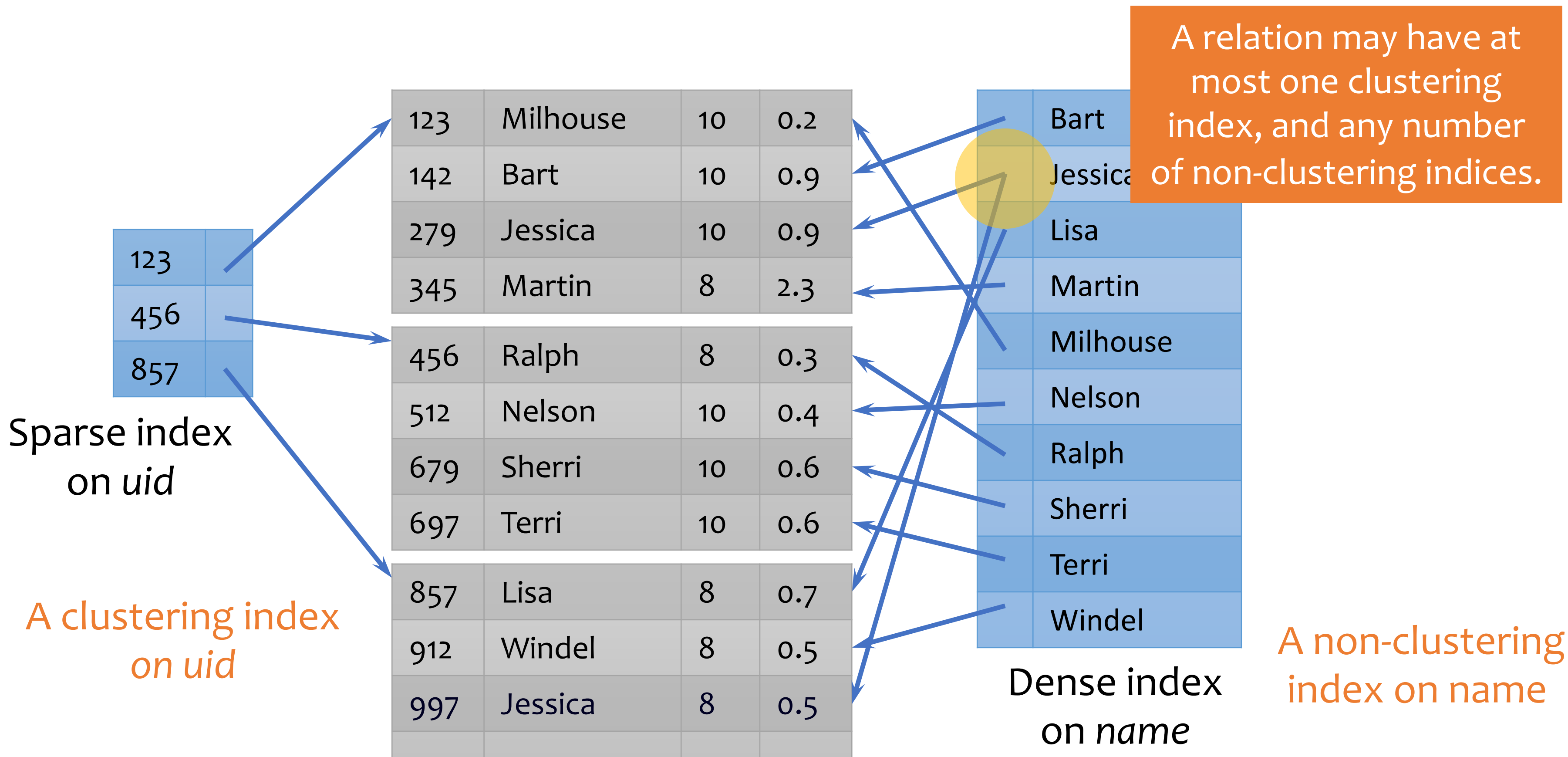
Dense v.s. Sparse indexes

- Dense: one index entry for each search key value
- Sparse: one index entry for each block
  - Records must be clustered according to the search key



Clustering v.s. Non-Clustering indexes

- An index on attribute A is a clustering index if tuples with similar A-values are stored together in the same block, and non-clustering otherwise.



Primary v.s. Secondary indexes

- Primary index
  - Typically created for the primary key of a table
  - Records are usually clustered by the primary key
  - Clustering index, so sparse
- Secondary index
  - Non-clustering index, usually dense (why?)

(Recap) Indexes in SQL

- PRIMARY KEY declaration automatically creates a primary index
- UNIQUE key declaration automatically creates a secondary index
- Additional secondary index can be created on non-key attribute(s)
  - CREATE INDEX UserPopIndex ON User(pop)

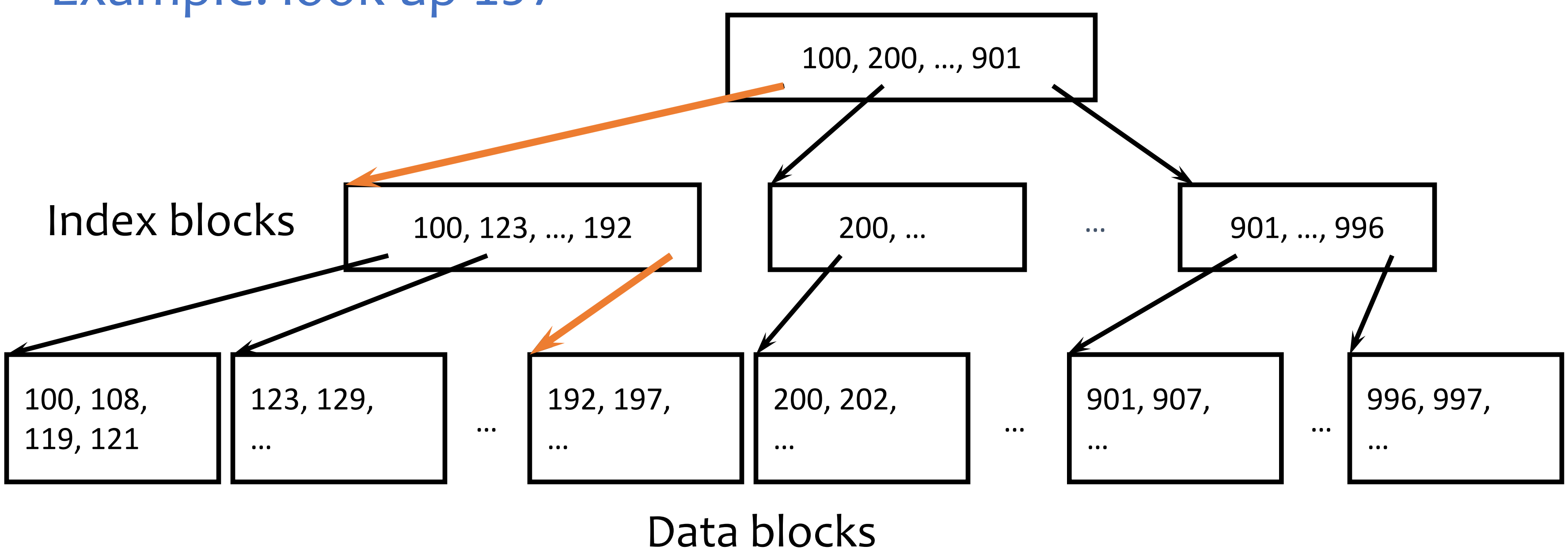
Outline

- Types of indexes
  - Sparse v.s. dense
  - Clustering v.s. non-clustering
  - Primary v.s. secondary
- Index structure
  - ISAM
  - B-tree
- How to use index

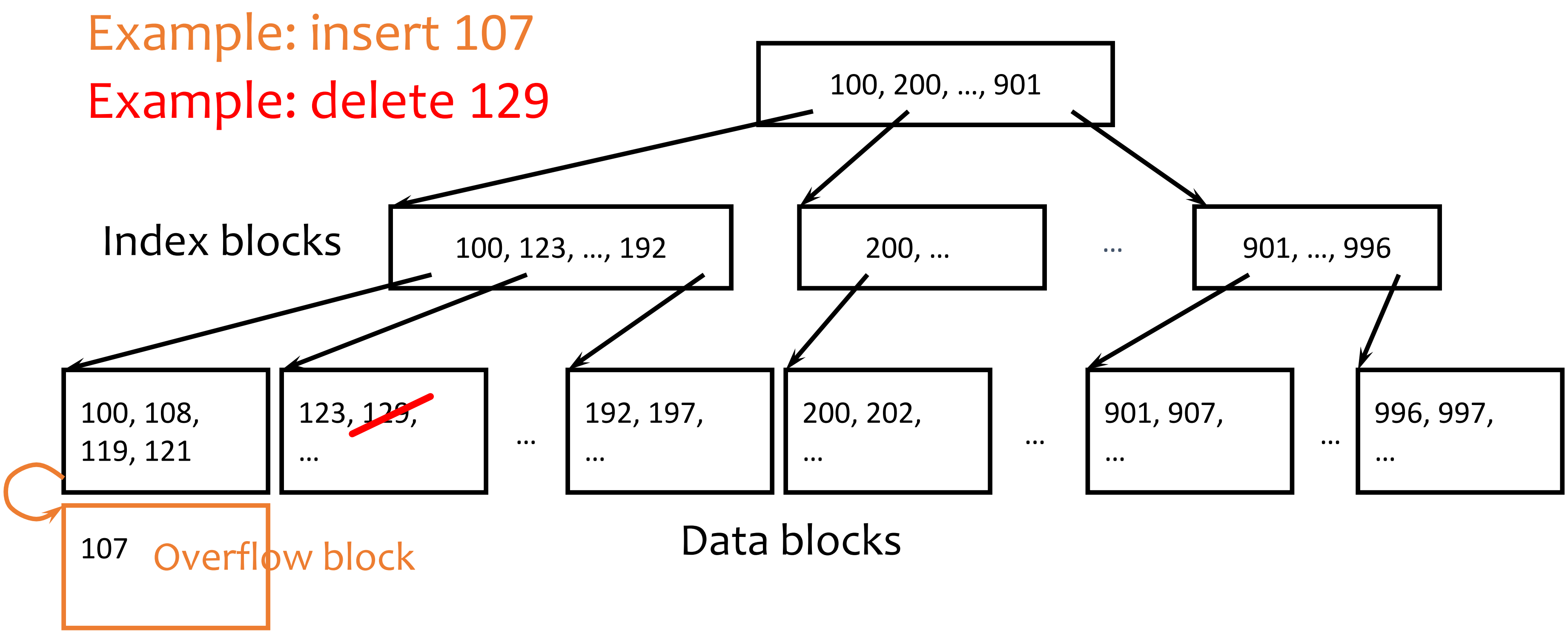
ISAM

- 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



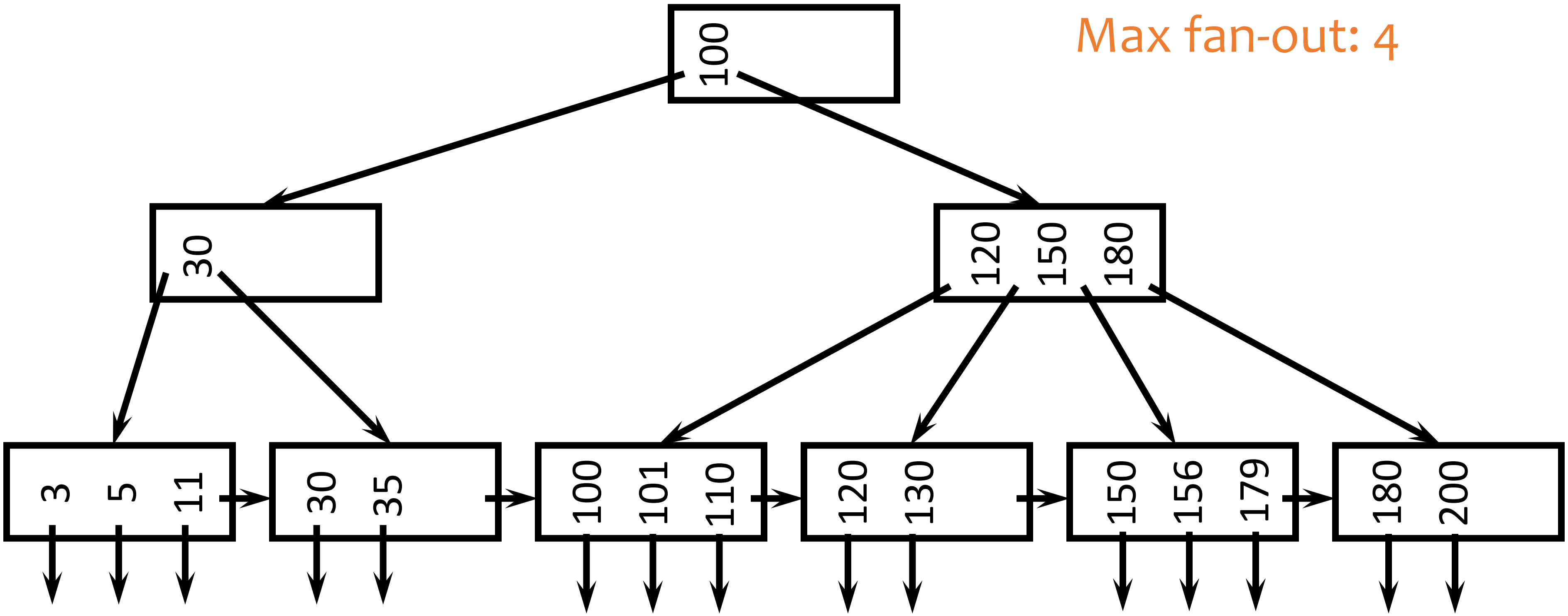
Updates with ISAM



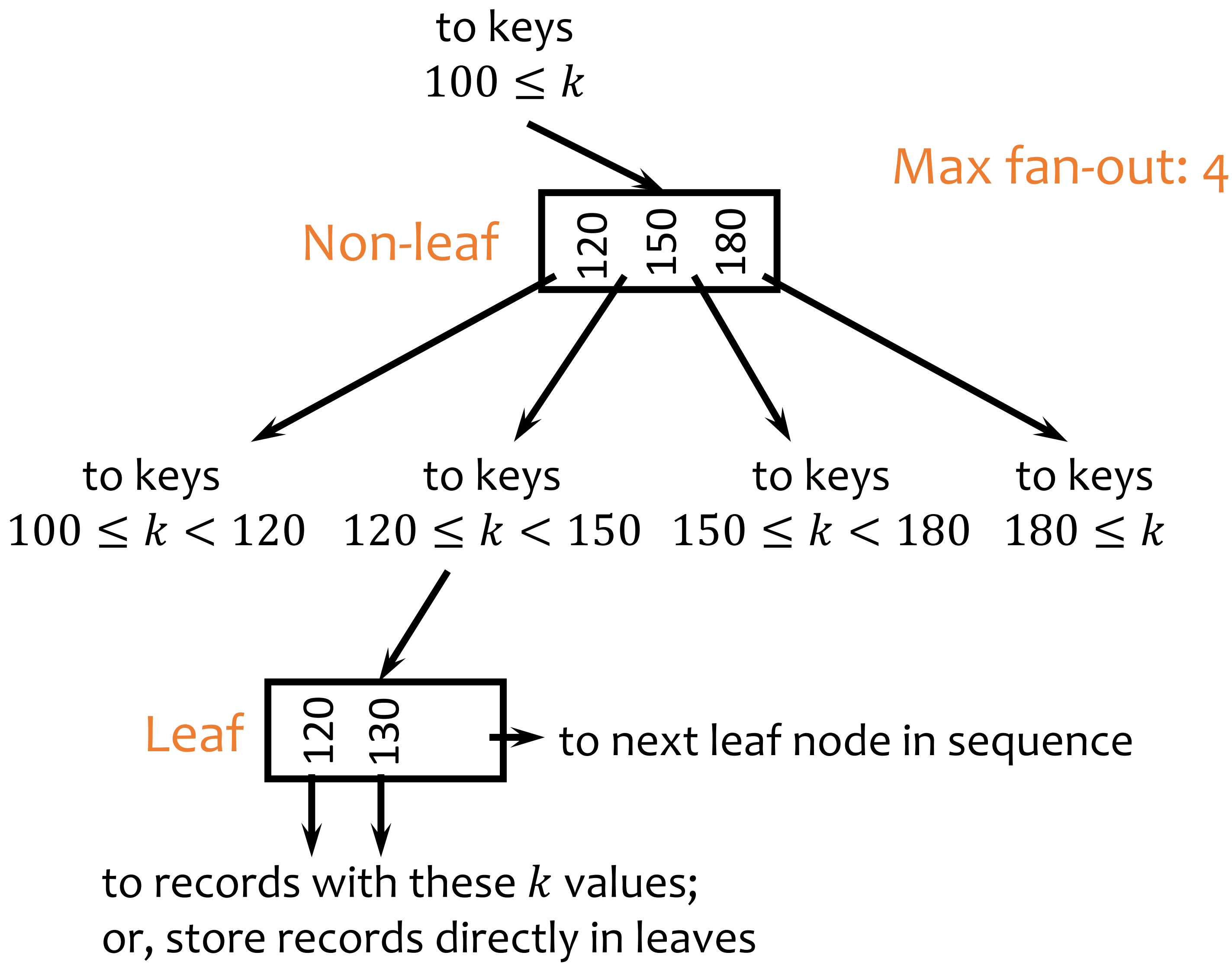
- Overflow chains and empty data blocks degrade performance
  - Worst case: most records go into one long chain, so lookups require scanning all data!

B^+-tree

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



Sample B+-tree nodes



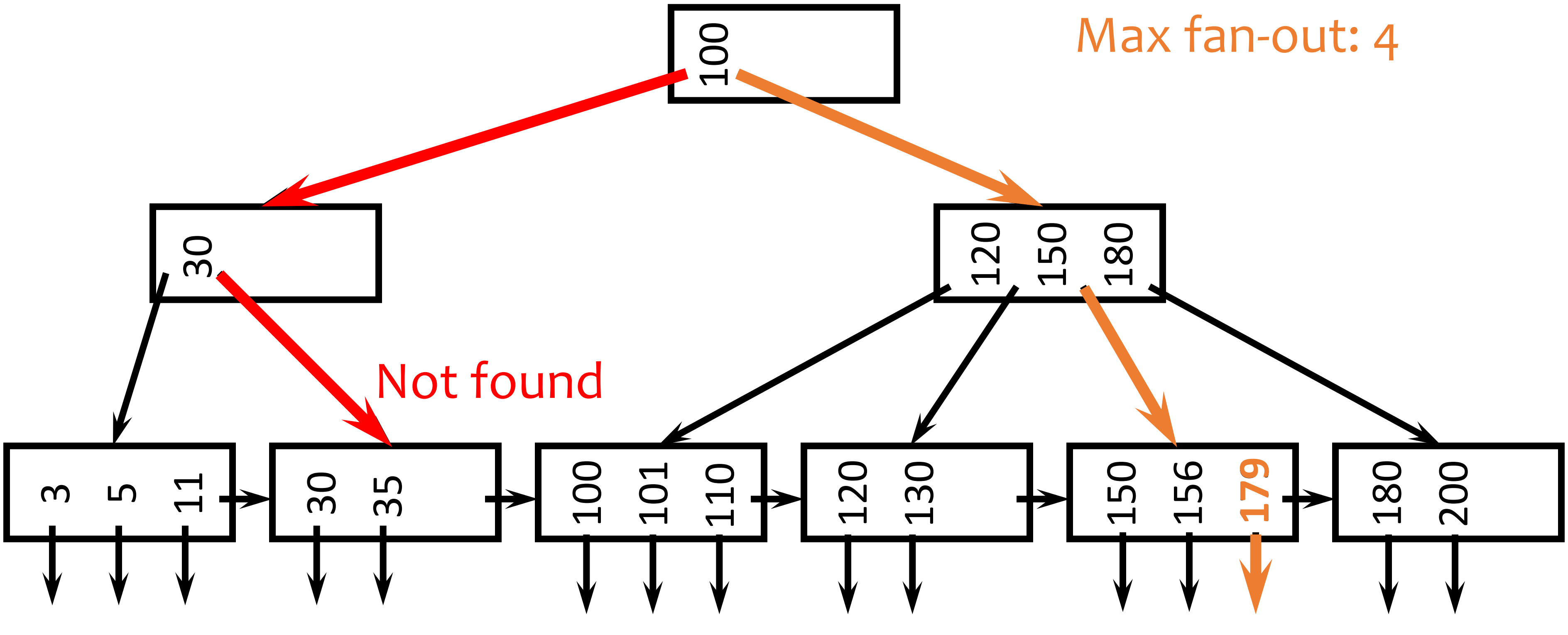
B^+-tree balancing properties

- 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

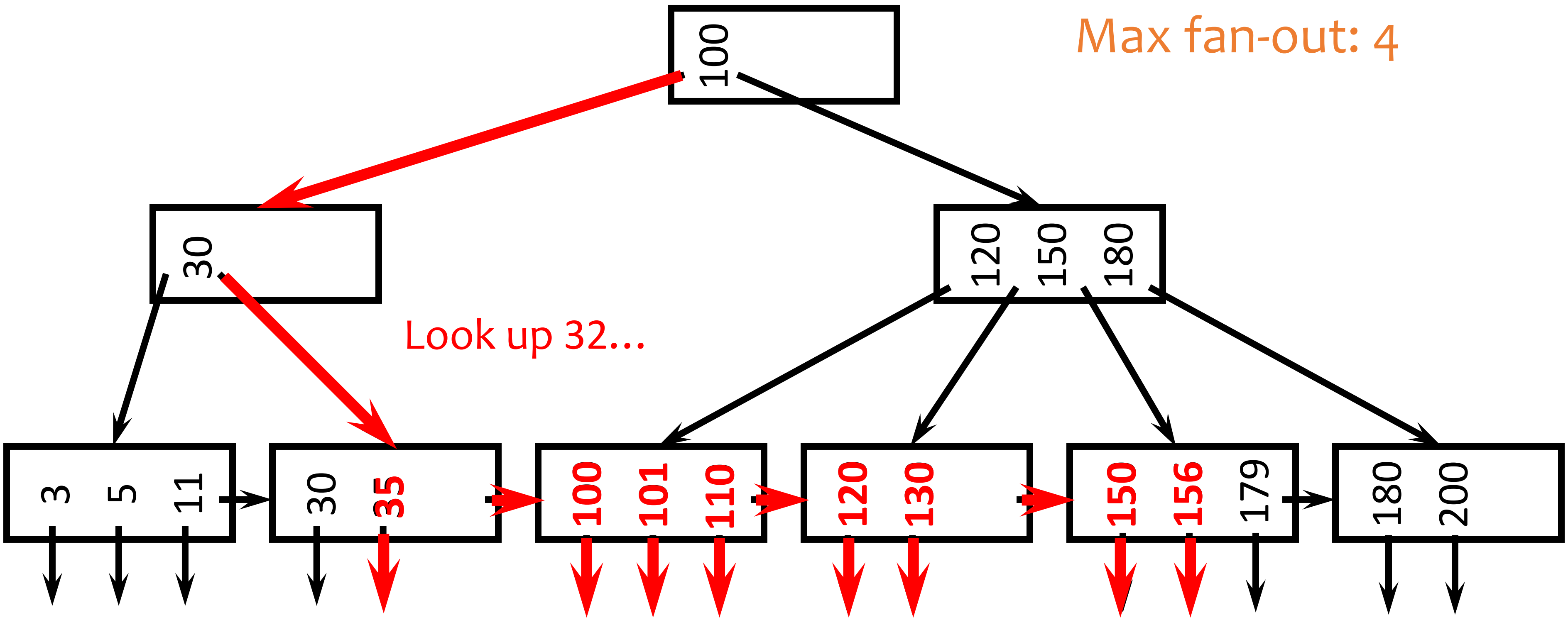
- SELECT * FROM *R* WHERE *k* = 179;
- SELECT * FROM *R* WHERE *k* = 32;



Range query

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

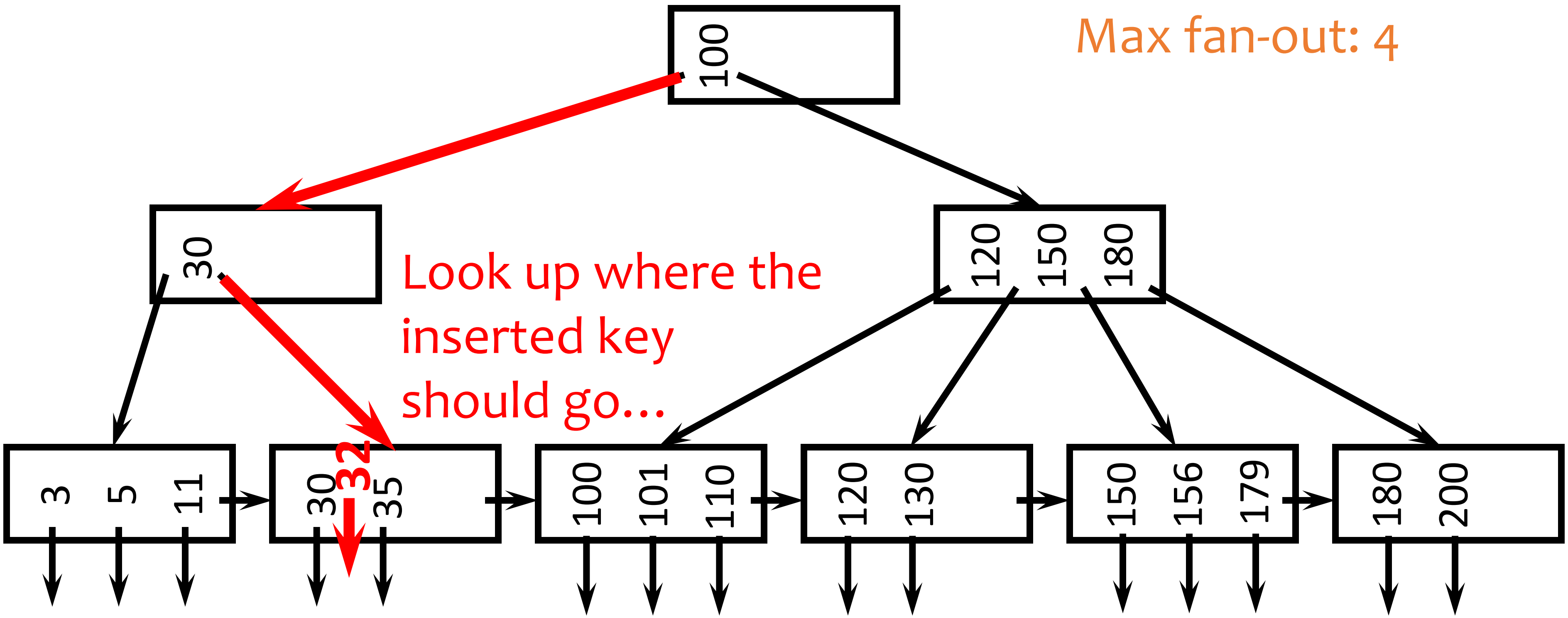
Max fan-out: 4



And follow next-leaf pointers until you hit upper bound

Insertion

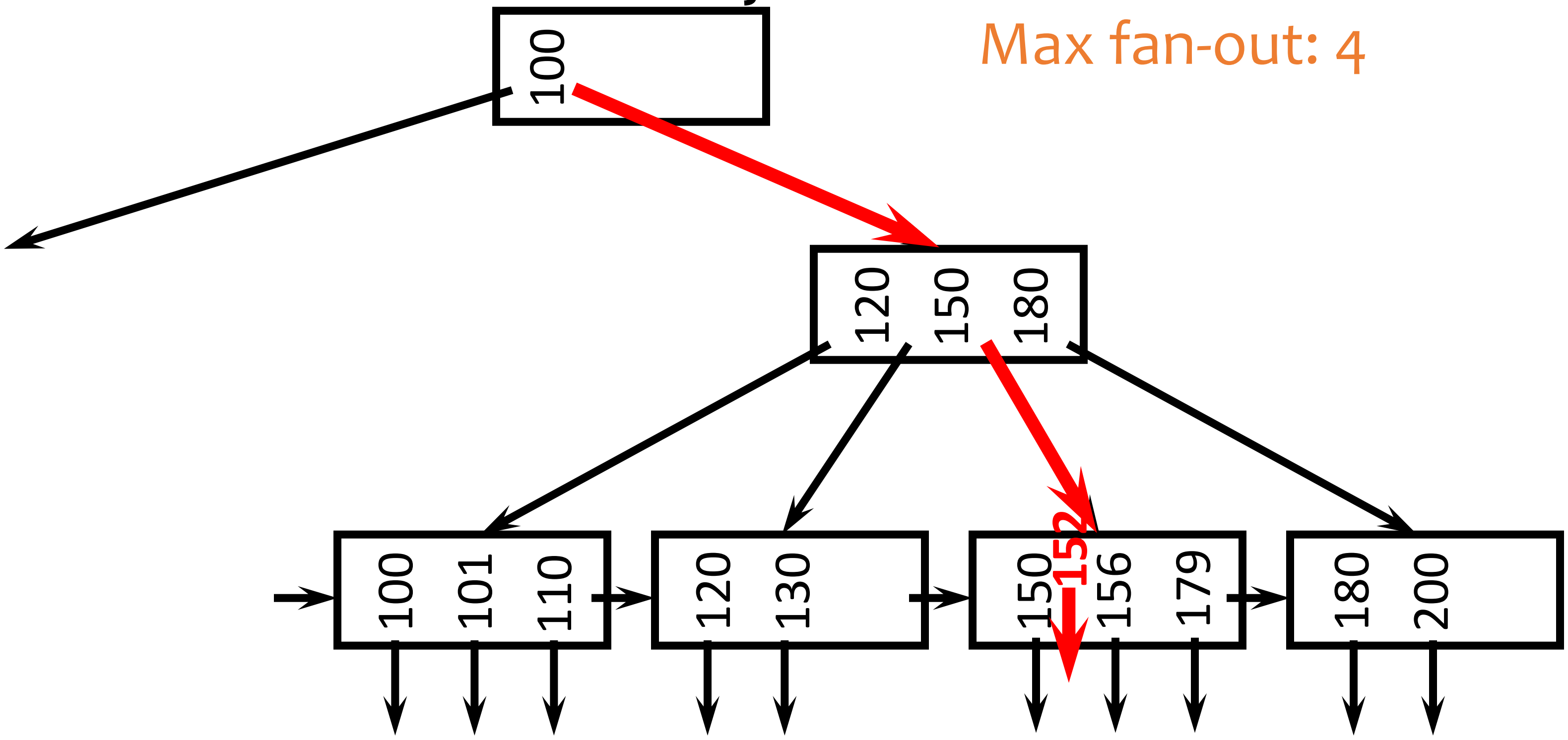
- Insert a record with search key value 32



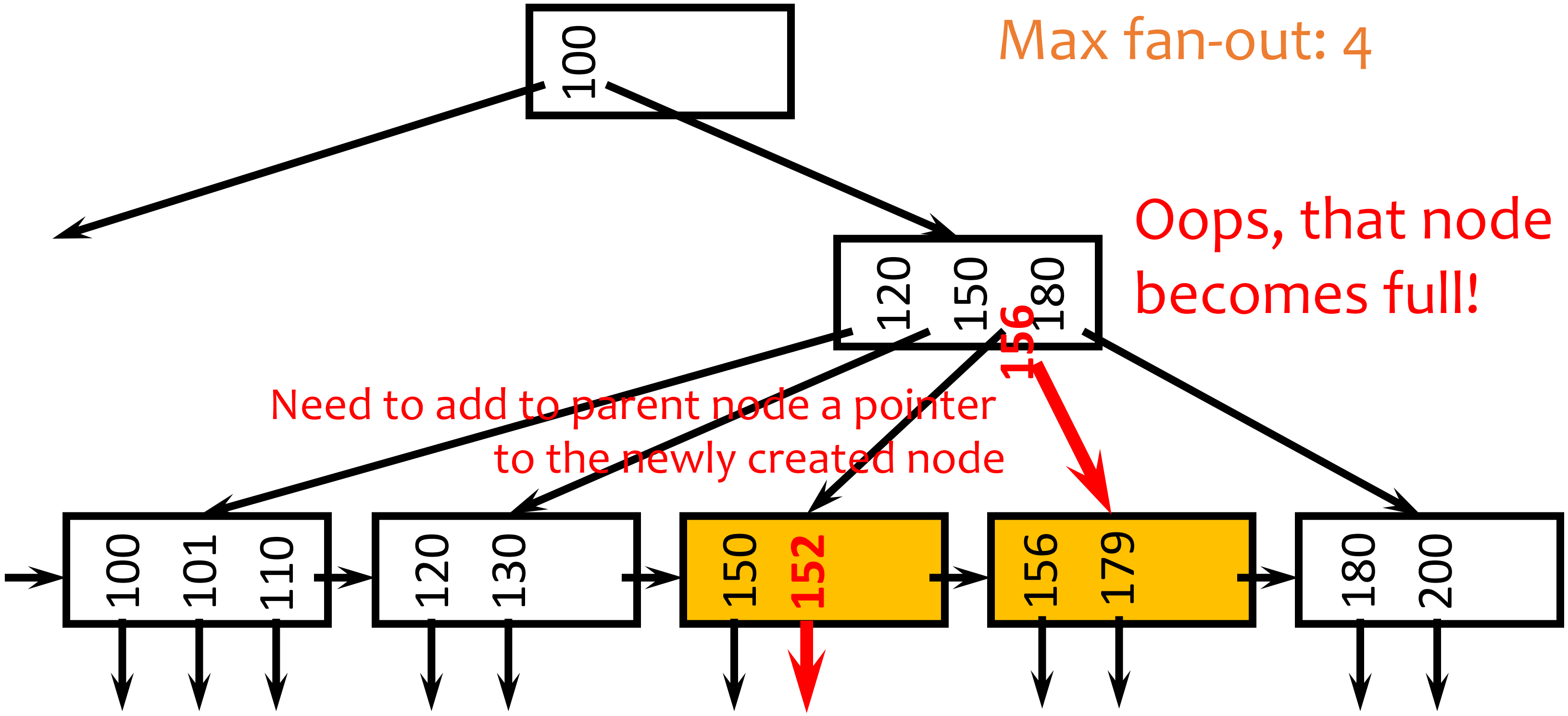
And insert it right there

Another insertion example

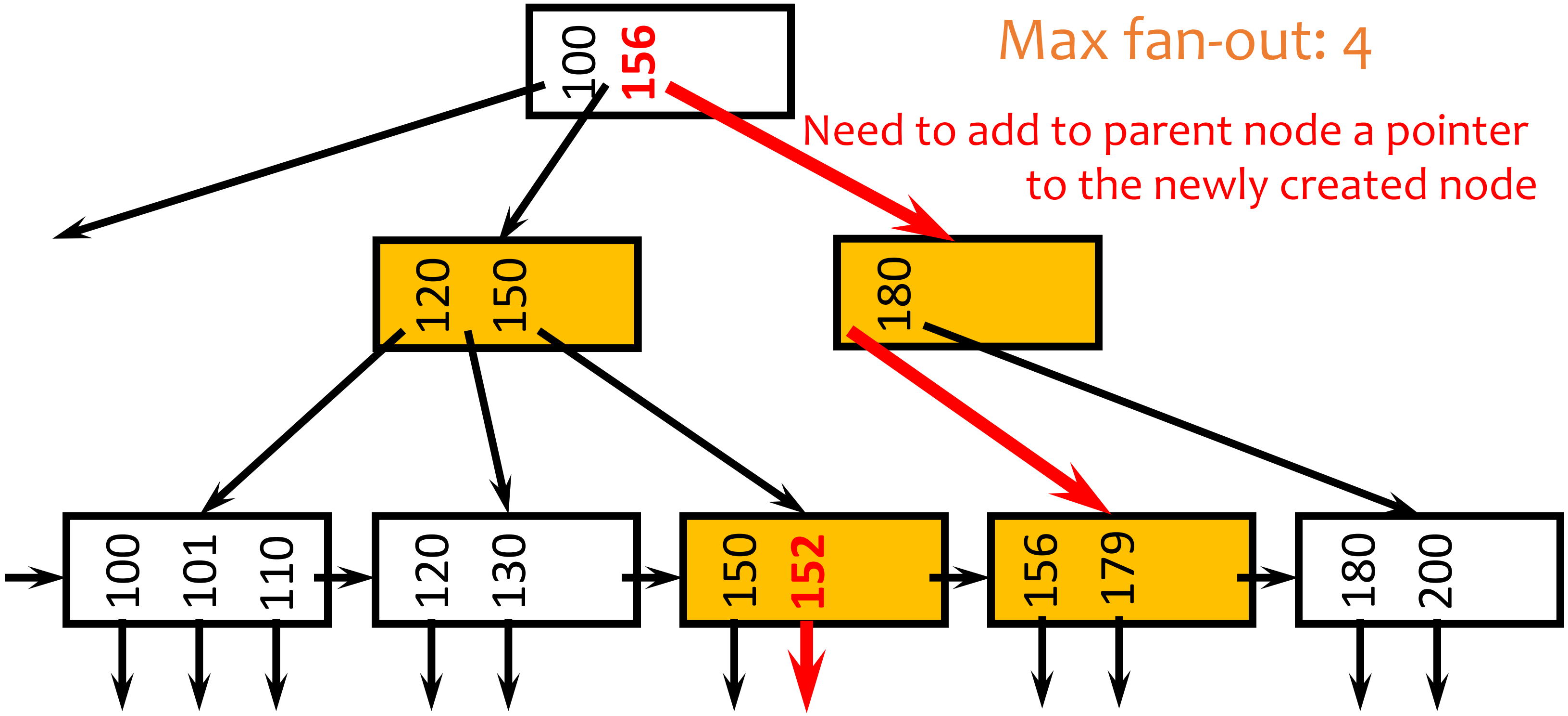
- Insert a record with search key value 152



Node splitting



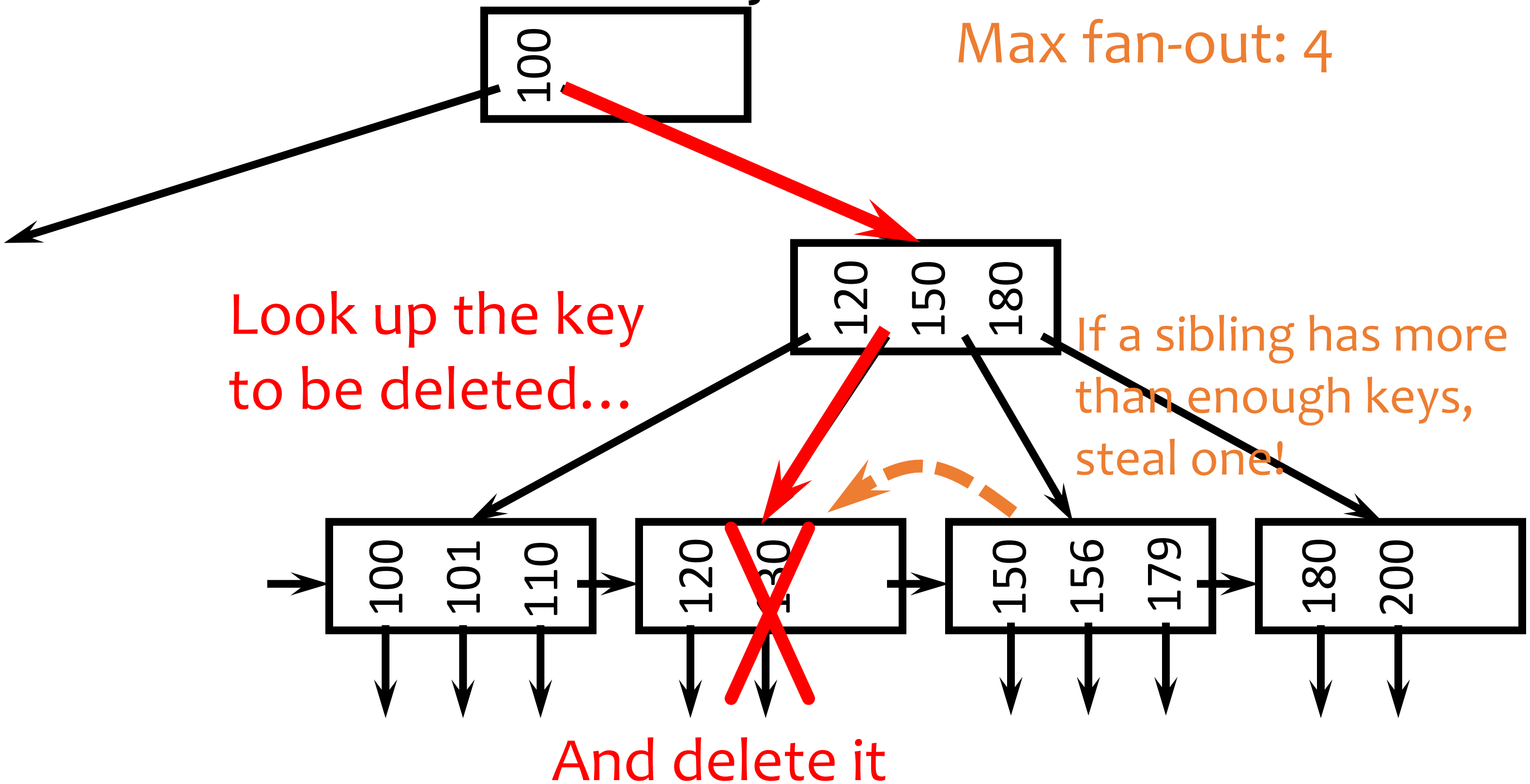
More node splitting



- 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

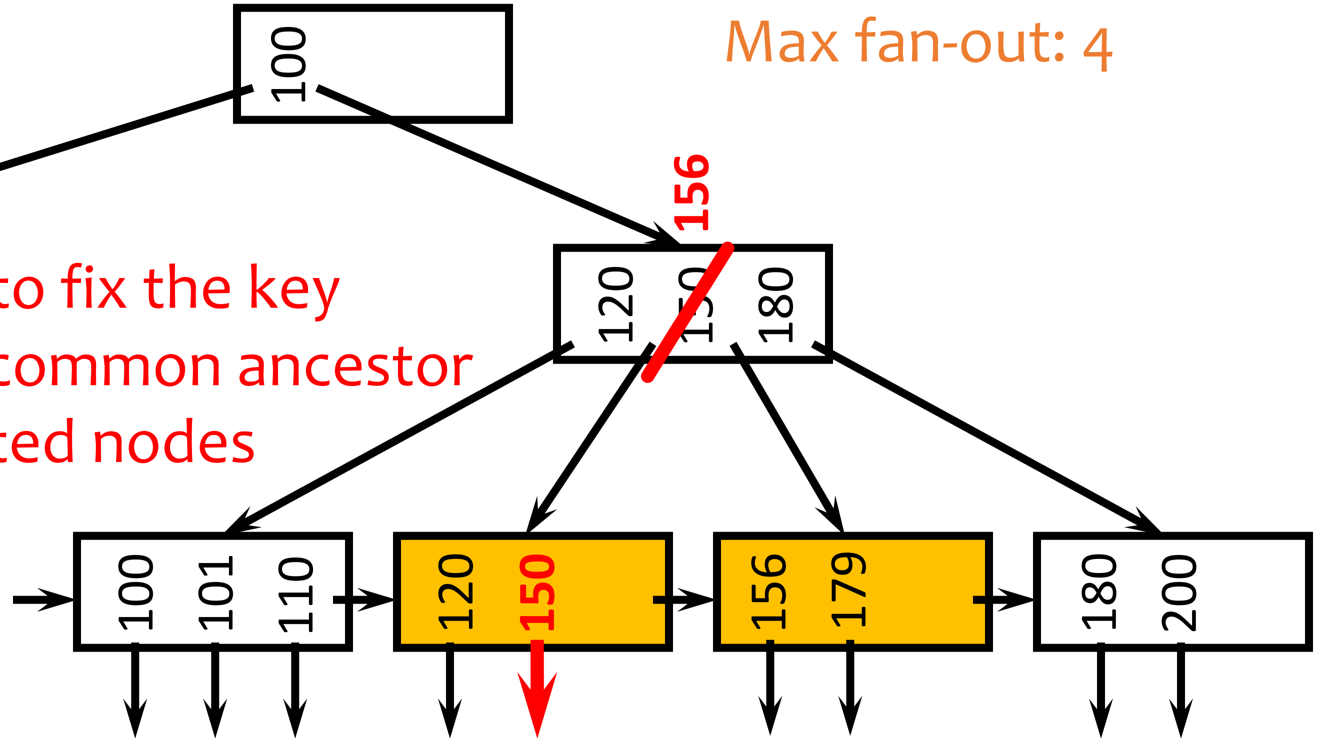


Oops, node is too empty!

Stealing from a sibling

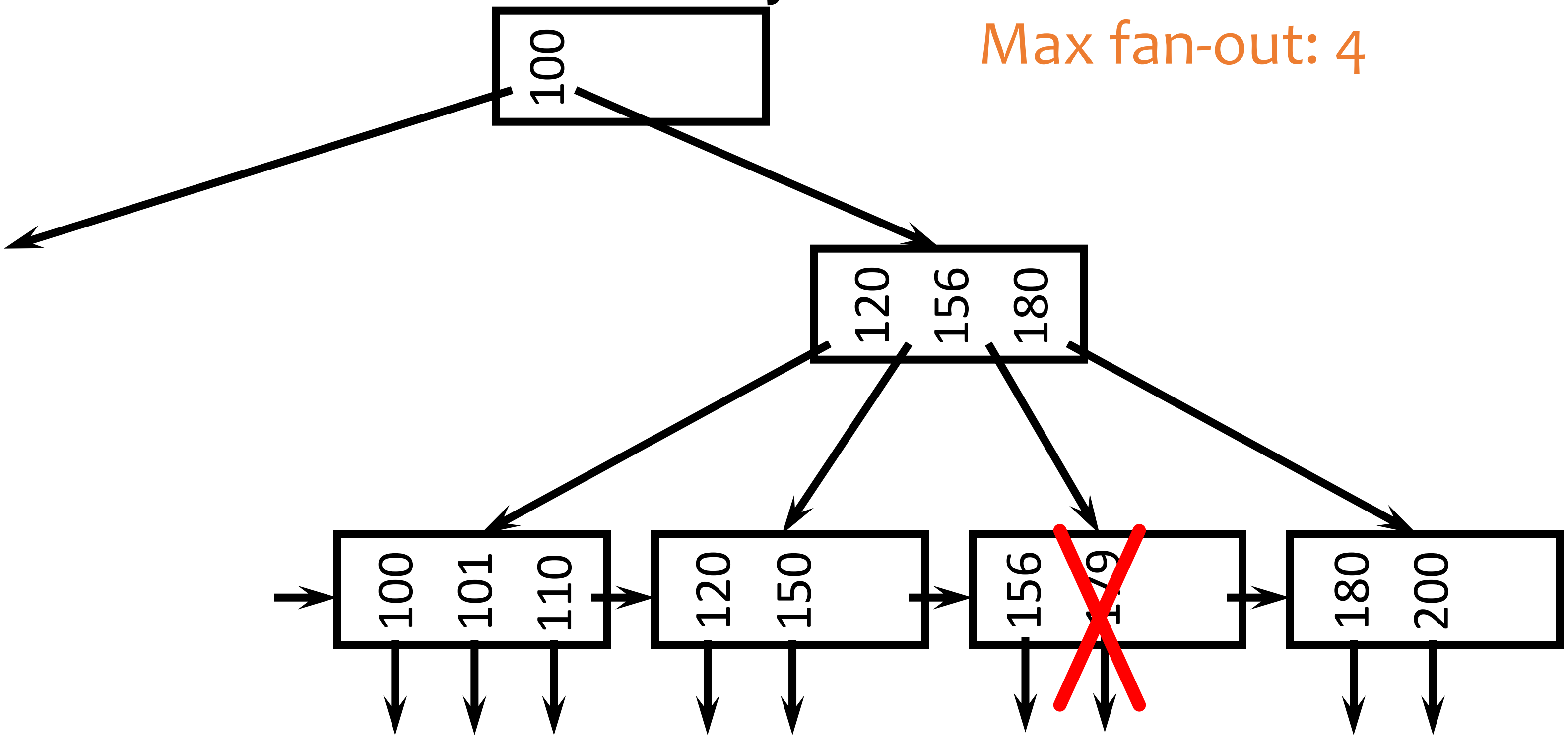
Max fan-out: 4

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



Another deletion example

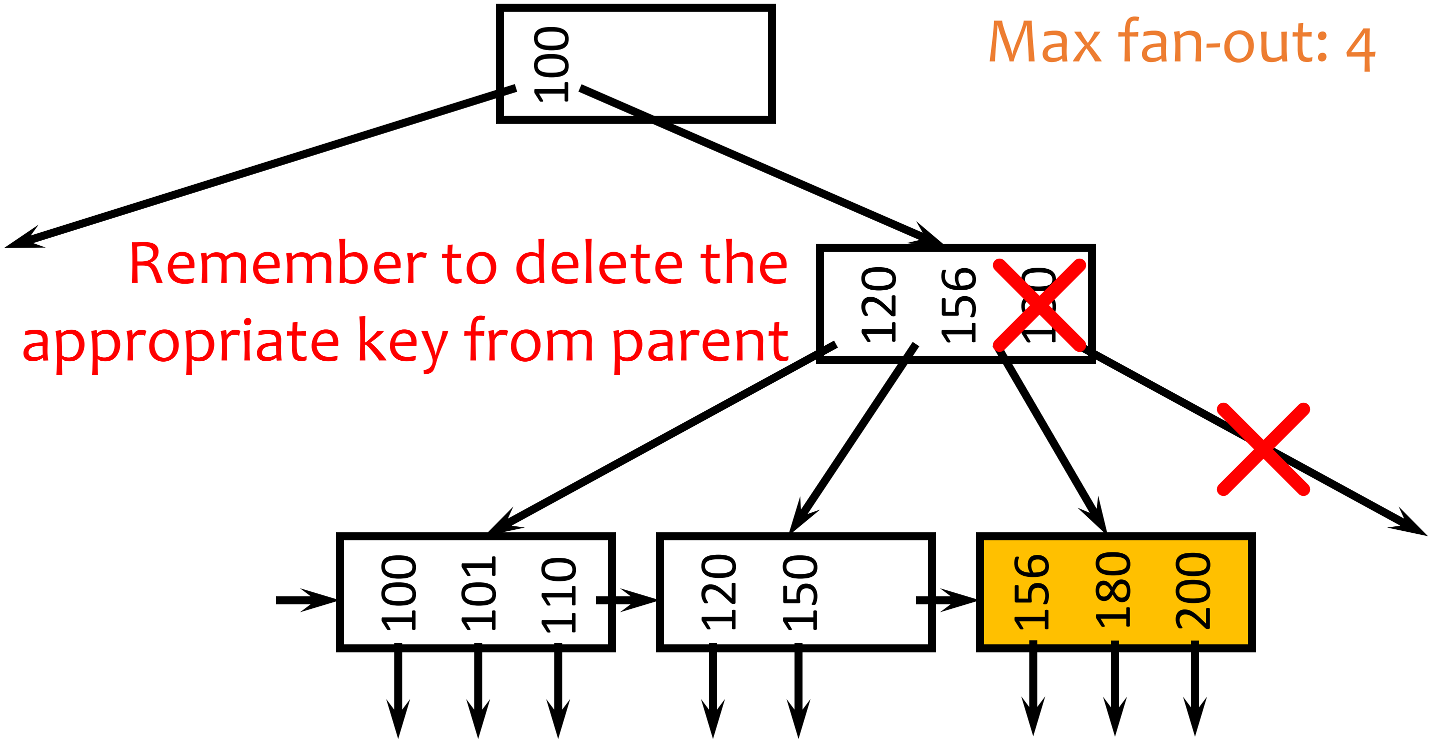
- Delete a record with search key value 179



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 of B+-tree

- How many I/O's are required for each operation?
  - h, the height of the tree
  - 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
  - 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
  - $h(value)\ mod\ f$: bucket/block to which data entry with search key value belongs

B^+-tree versus ISAM

- ISAM is more static; B^+-tree is more dynamic
- ISAM can be more compact (at least initially)
    - Fewer levels and I/O's than B^+-tree
- Overtime, ISAM may not be balanced
    - Cannot provide guaranteed performance as B^+-tree does

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 requiring more I/O on average
  - Deletions are hard since search keys cannot be repeated
  - Range queries can become less efficient

Outline

- Types of indexes:
  - Dense v.s. sparse
  - Clustering v.s. non-clustering
  - Primary v.s. secondary
- Indexing structure
  - ISAM
  - B+-tree
  - Hashing
- How to use index