Indexing

Introduction to Databases

CompSci 316 Spring 2019

DUKE
COMPUTER SCIENCE

Announcements (Tue., Mar. 26)

- Homework #3 due tomorrow 03/27
  - 5% per hour late penalty
- Project milestone #2 due Friday 03/29
  - one report per group
- HW4:
  - one problem (similar to exam problems) on every week's lectures due in 7 days (see piazza post)
  - gradiance problems are due in two weeks
- Short weekly update required from all project group members by each Friday on your piazza threads
  - see piazza

Today’s lecture

- Index

$$\left.\begin{array}{l}\text{• Dense vs. Sparse}\\ \text{• Clustered vs. unclustered}\\ \text{• Primary vs. secondary}\end{array}\right\}\text{Related}$$

- Tree-based vs. Hash-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*;

  - Keyword search

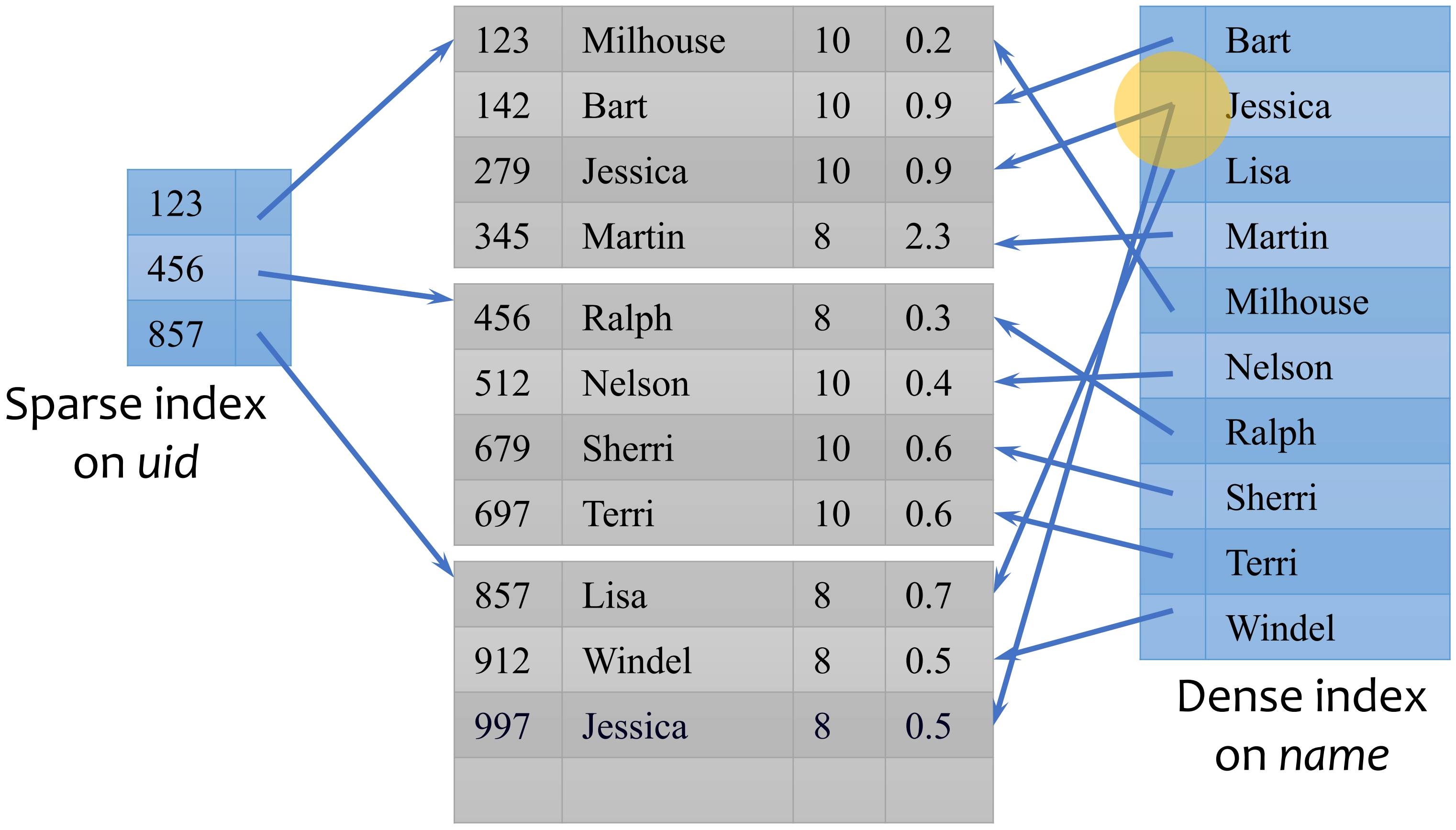
Focus of this lecture

High level structure of indexes

- (in class)
- what is a search key k?
- what is data entry (index entry) k*?
- how do we access a record?

Dense and sparse indexes

- 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
    - 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

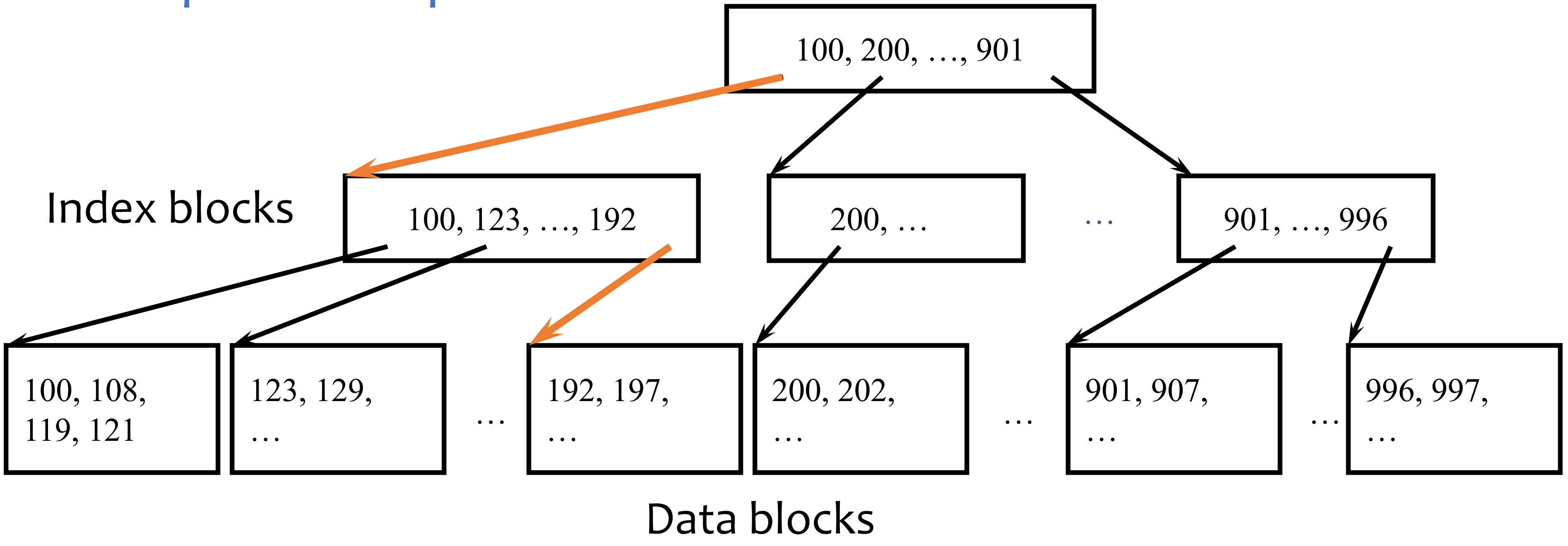
Primary and secondary indexes

- 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);`

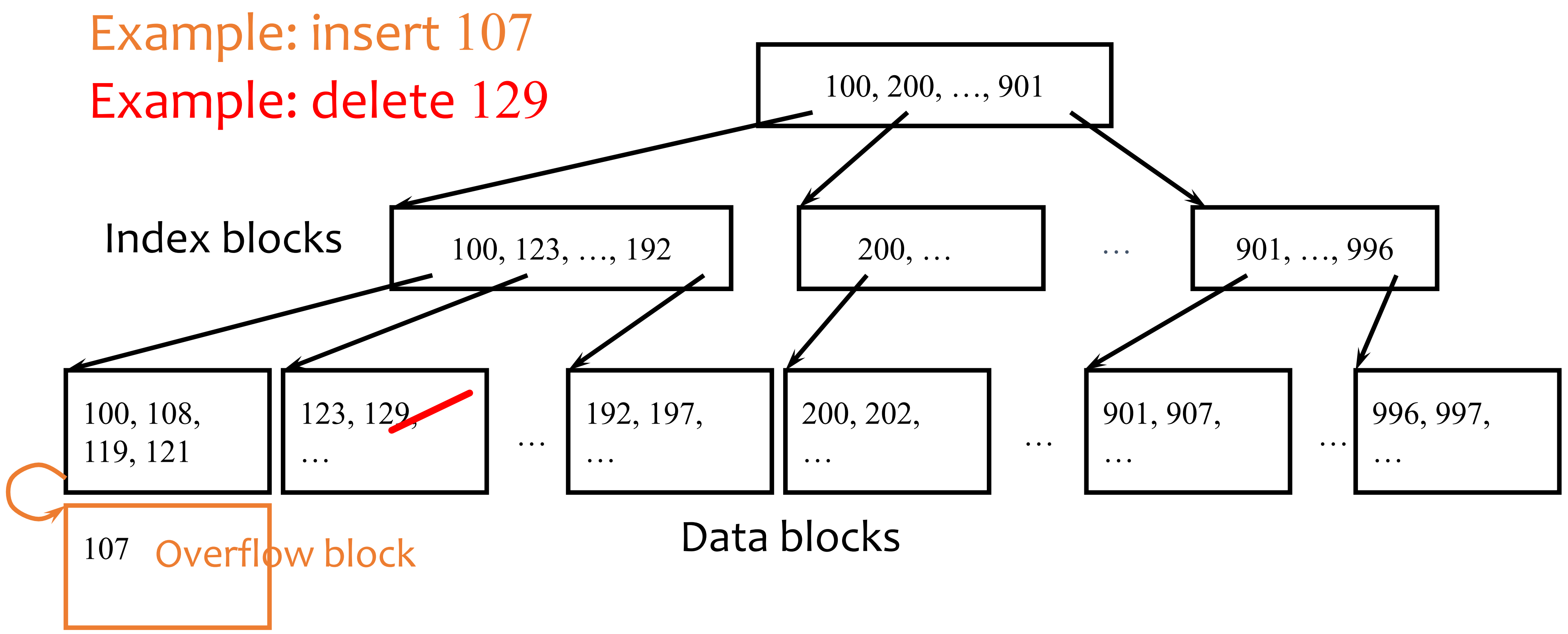
ISAM

- 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



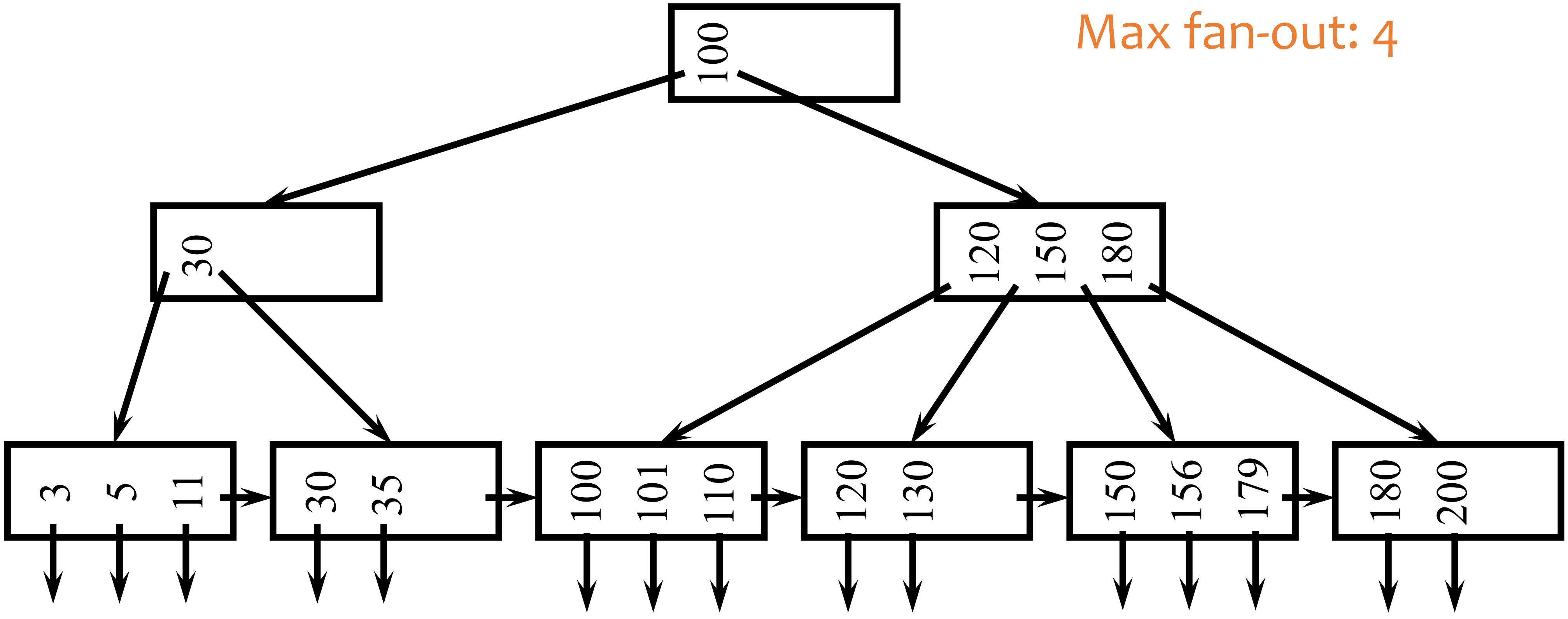
Updates with ISAM



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

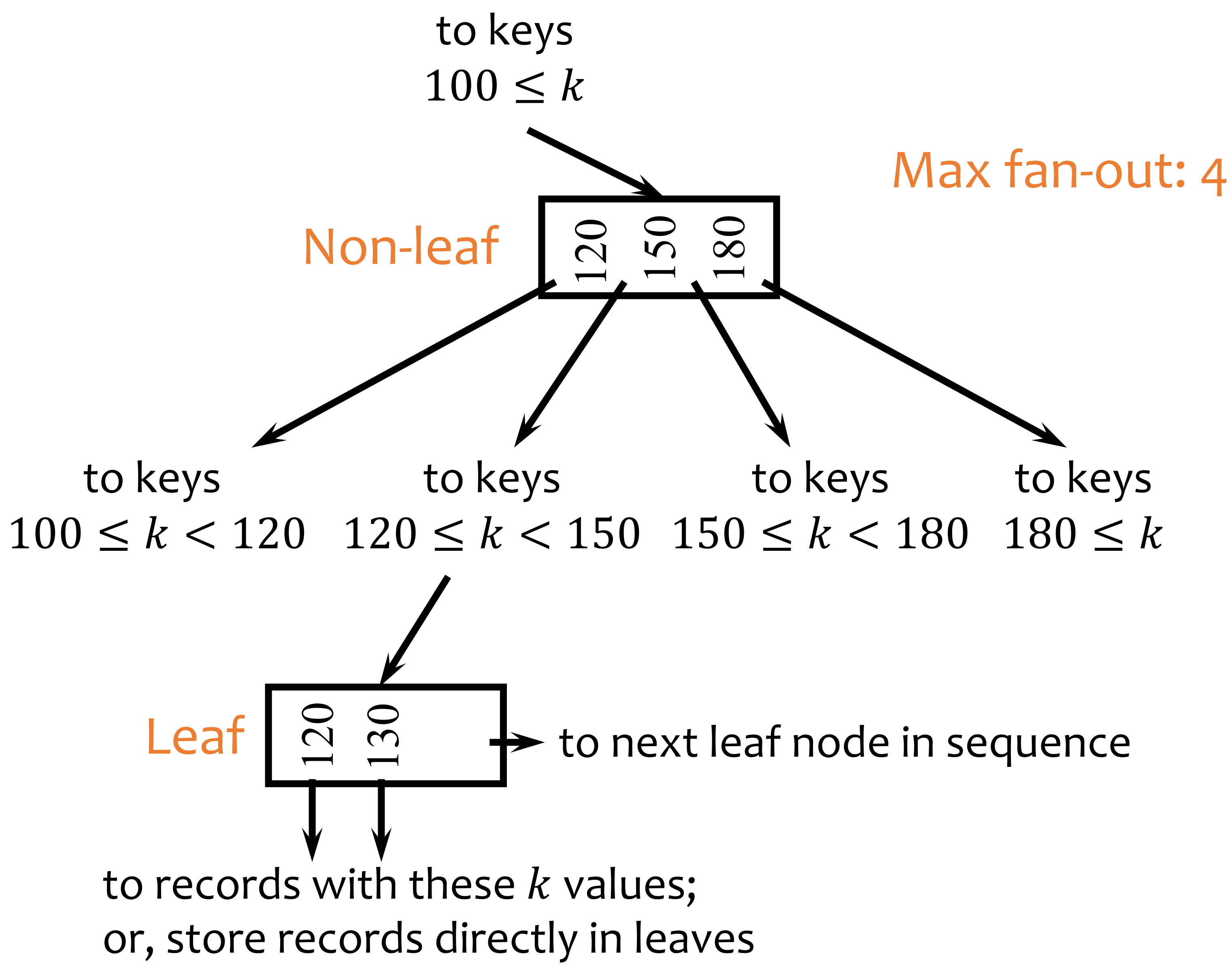
B+-tree

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



Max fan-out: 4

Sample B⁺-tree nodes



to keys $100 \leq k$
Max fan-out: 4
Non-leaf
120
150
180
to keys $100 \leq k < 120$
to keys $120 \leq k < 150$
to keys $150 \leq k < 180$
to keys $180 \leq k$
Leaf
120
130
to next leaf node in sequence
to records with these k values;
or, store records directly in leaves

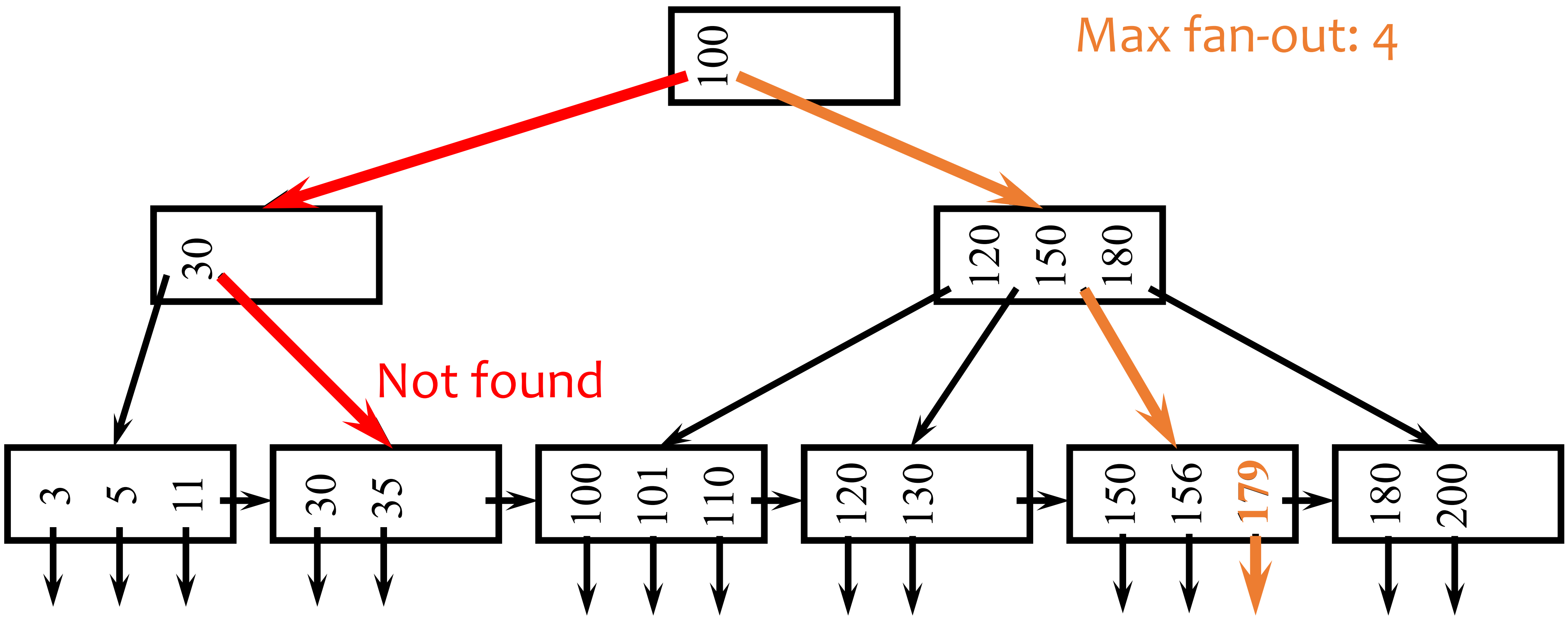
B$^+$-tree balancing properties

- 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	$f-1$	$\lfloor f/2 \rfloor$	$\lfloor f/2 \rfloor$

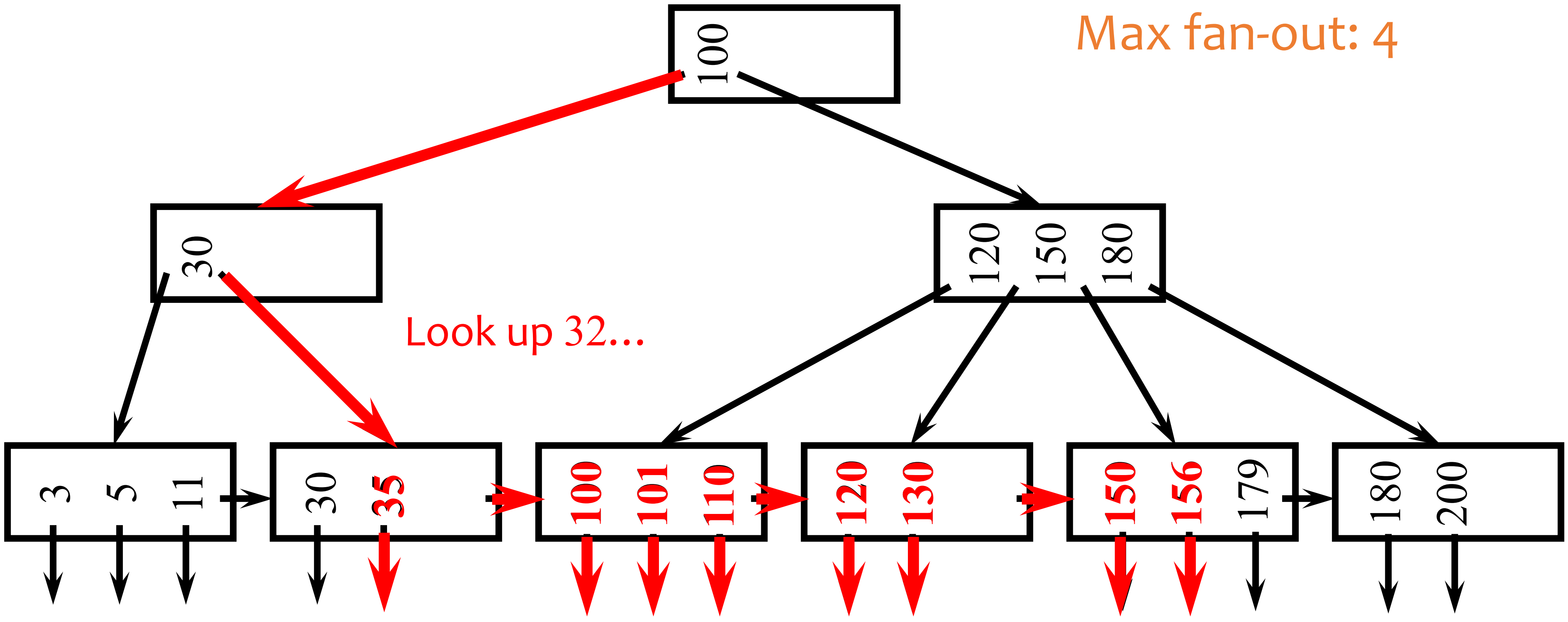
Lookups

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



Range query

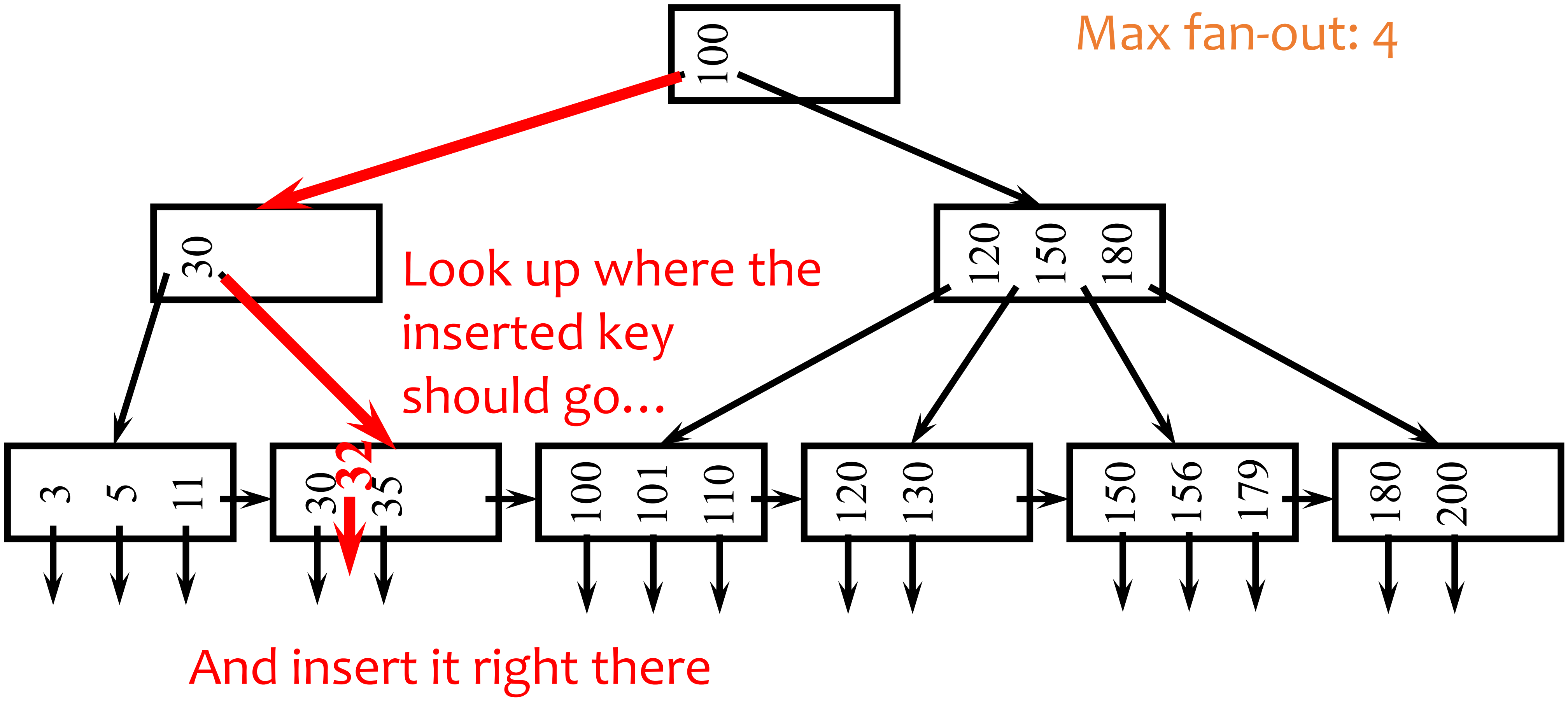
- SELECT * FROM *R* WHERE $k > 32$ AND $k < 179$;



And follow next-leaf pointers until you hit upper bound

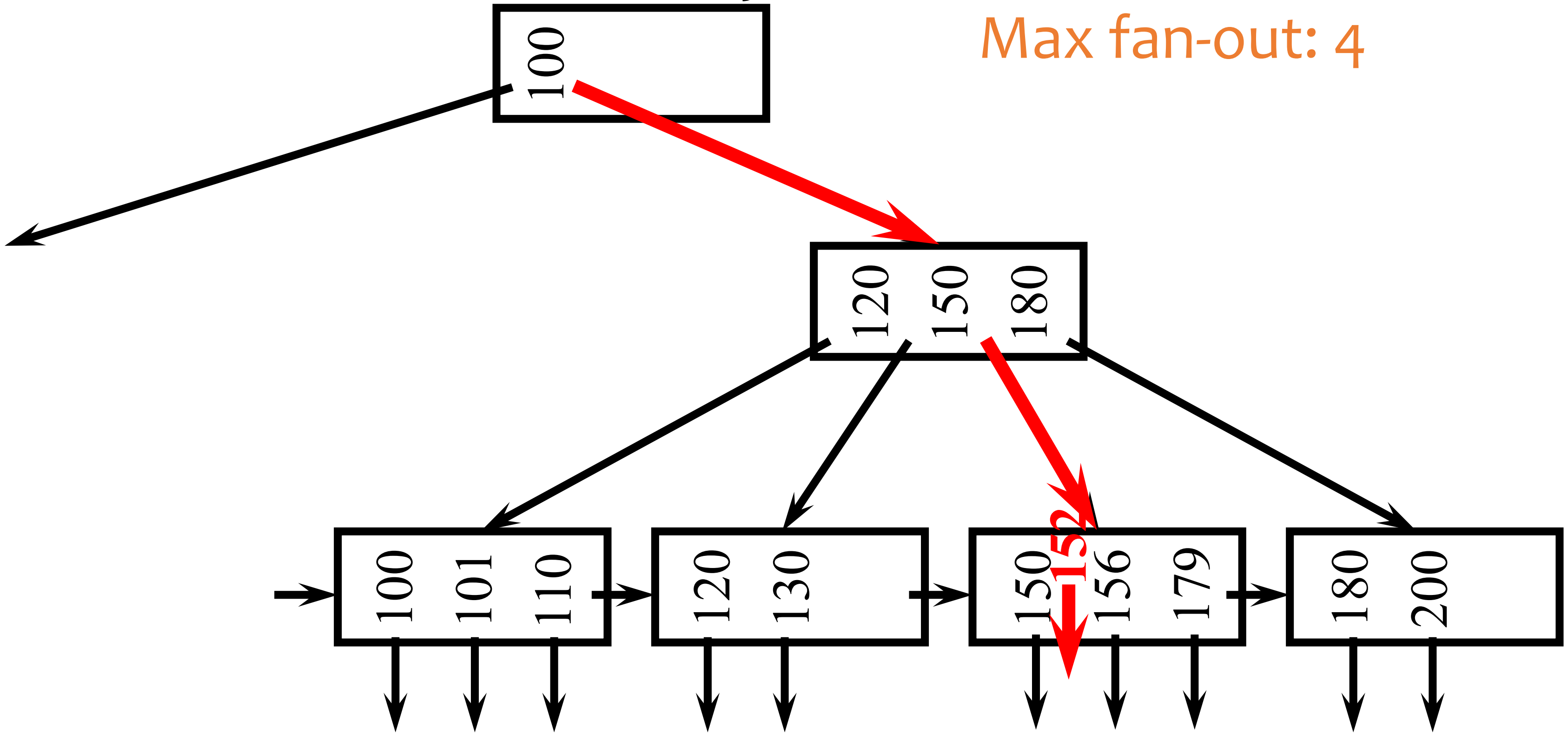
Insertion

- Insert a record with search key value 32



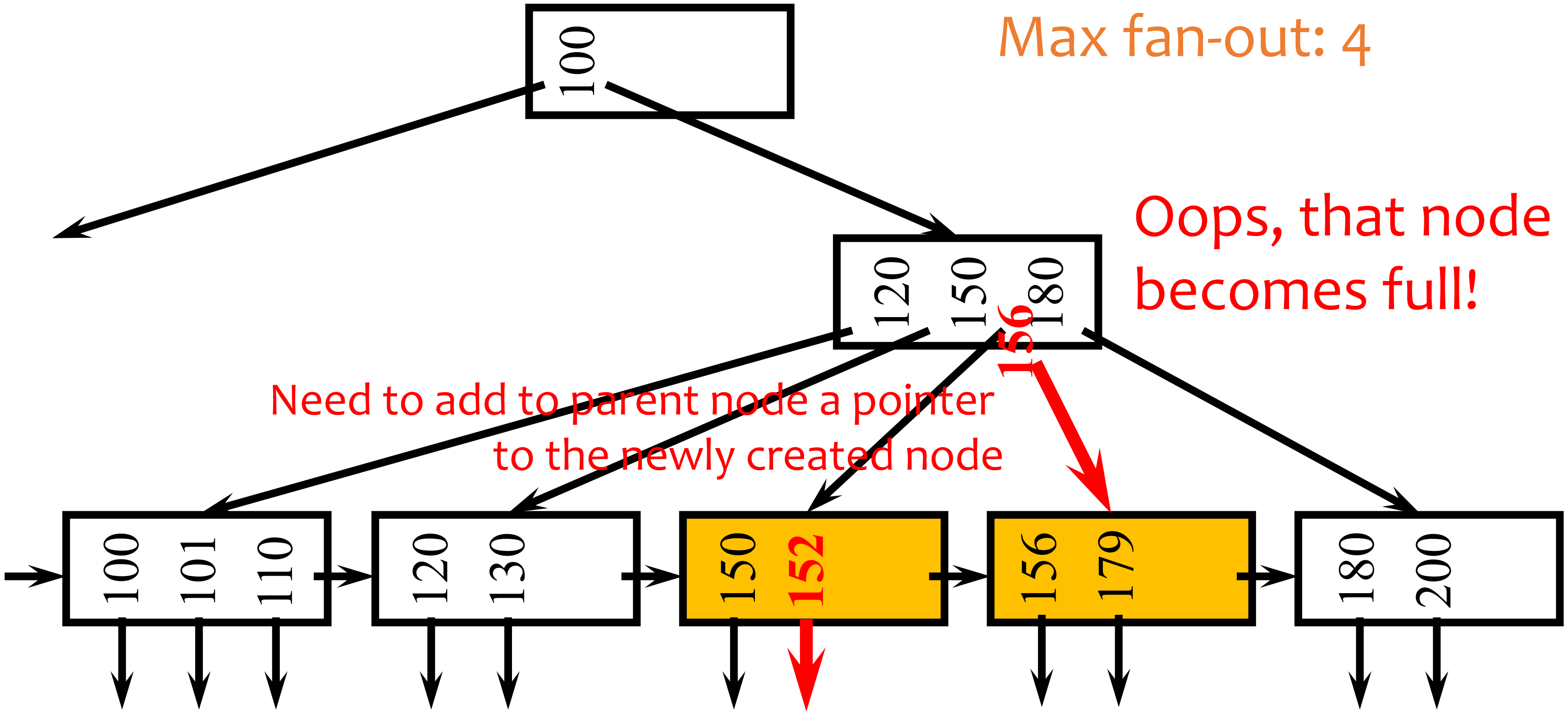
Another insertion example

- Insert a record with search key value 152

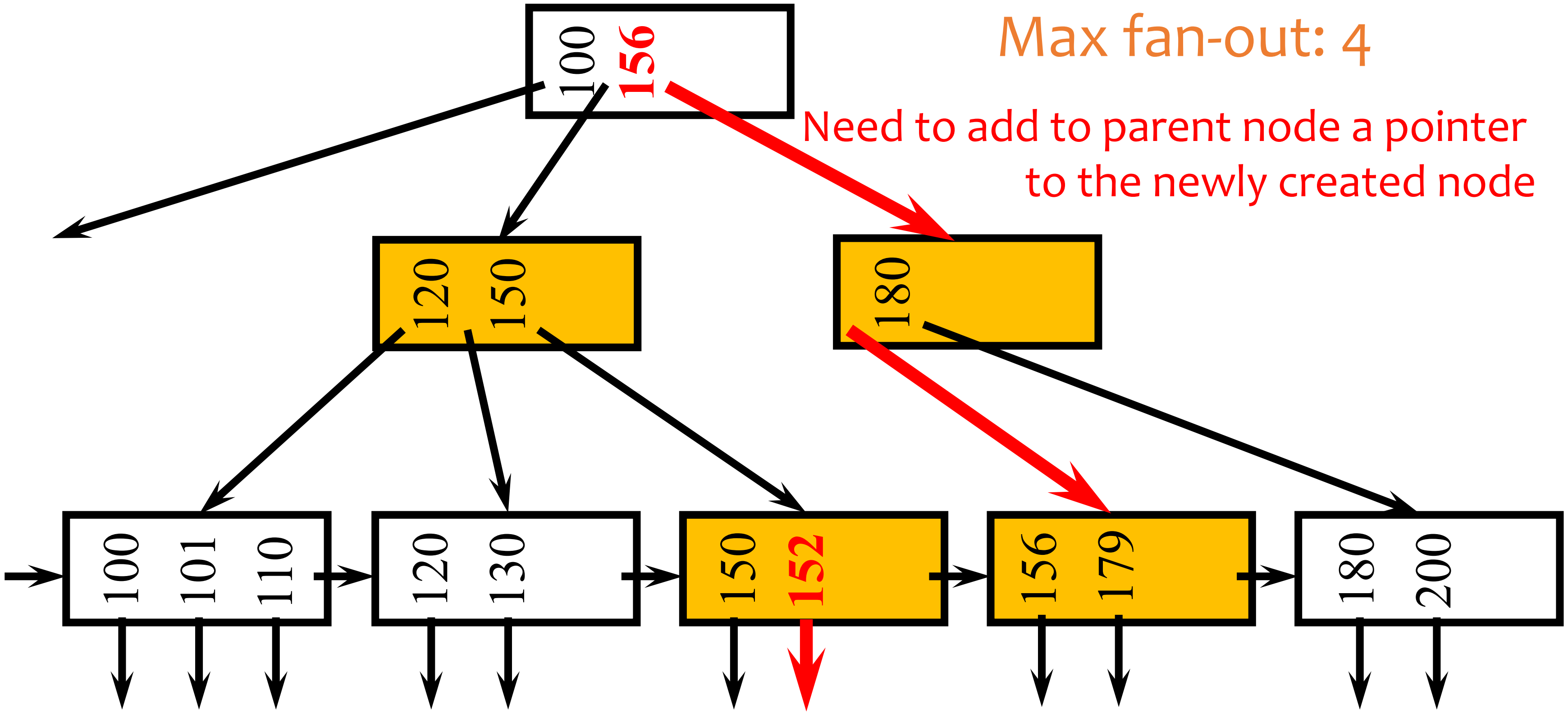


Oops, node is already full!

Node splitting


Max fan-out: 4
Oops, that node becomes full!
Need to add to parent node a pointer to the newly created node
100
120
150
180
156
100
101
110
120
130
150
152
156
179
180
200

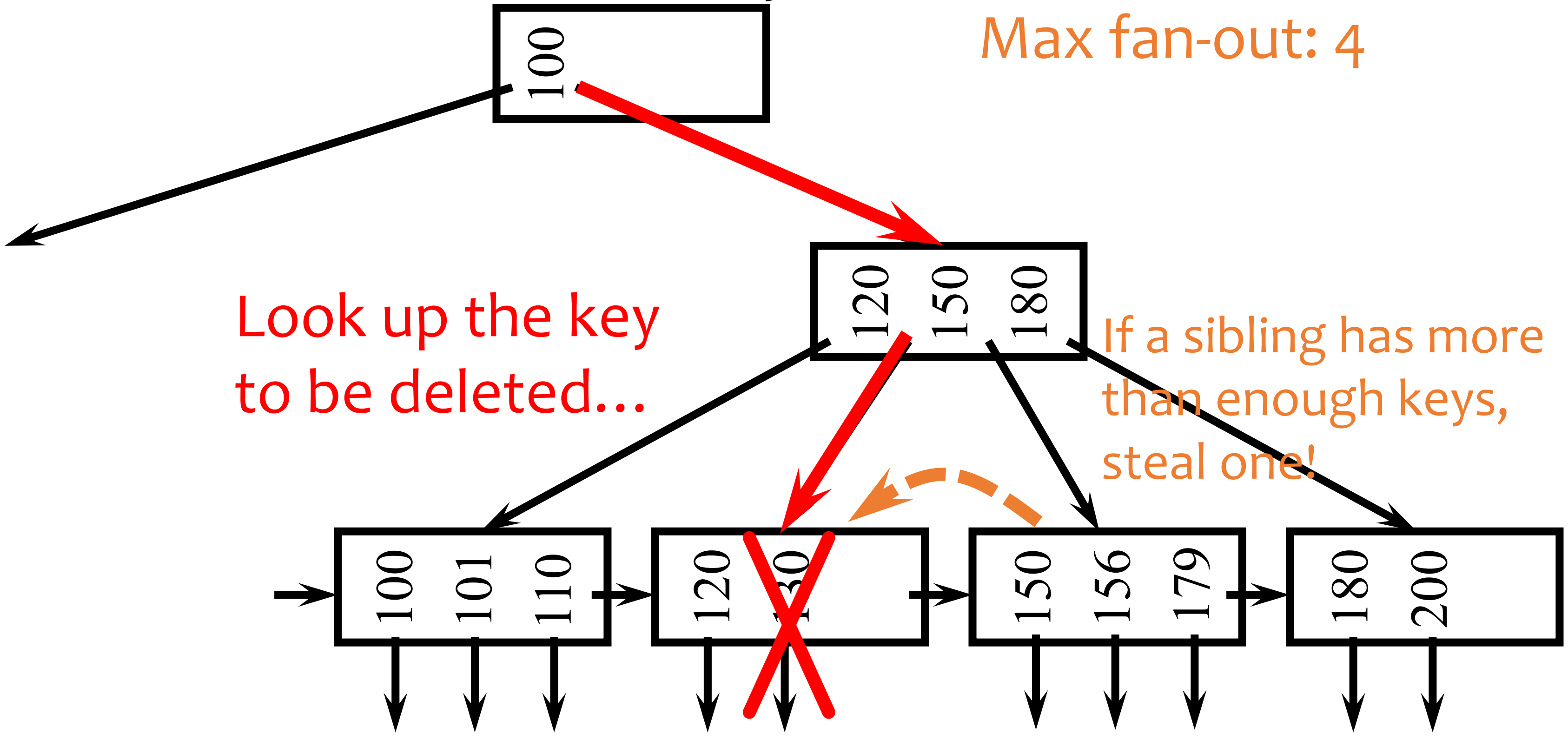
More node splitting


100
156
120
150
180
100
101
110
120
130
150
152
156
179
180
200
Max fan-out: 4
Need to add to parent node a pointer to the newly created node

- 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


100
120 150 180
100 101 110
120 130
150 156 179
180 200
Max fan-out: 4
Look up the key to be deleted…
If a sibling has more than enough keys, steal one!

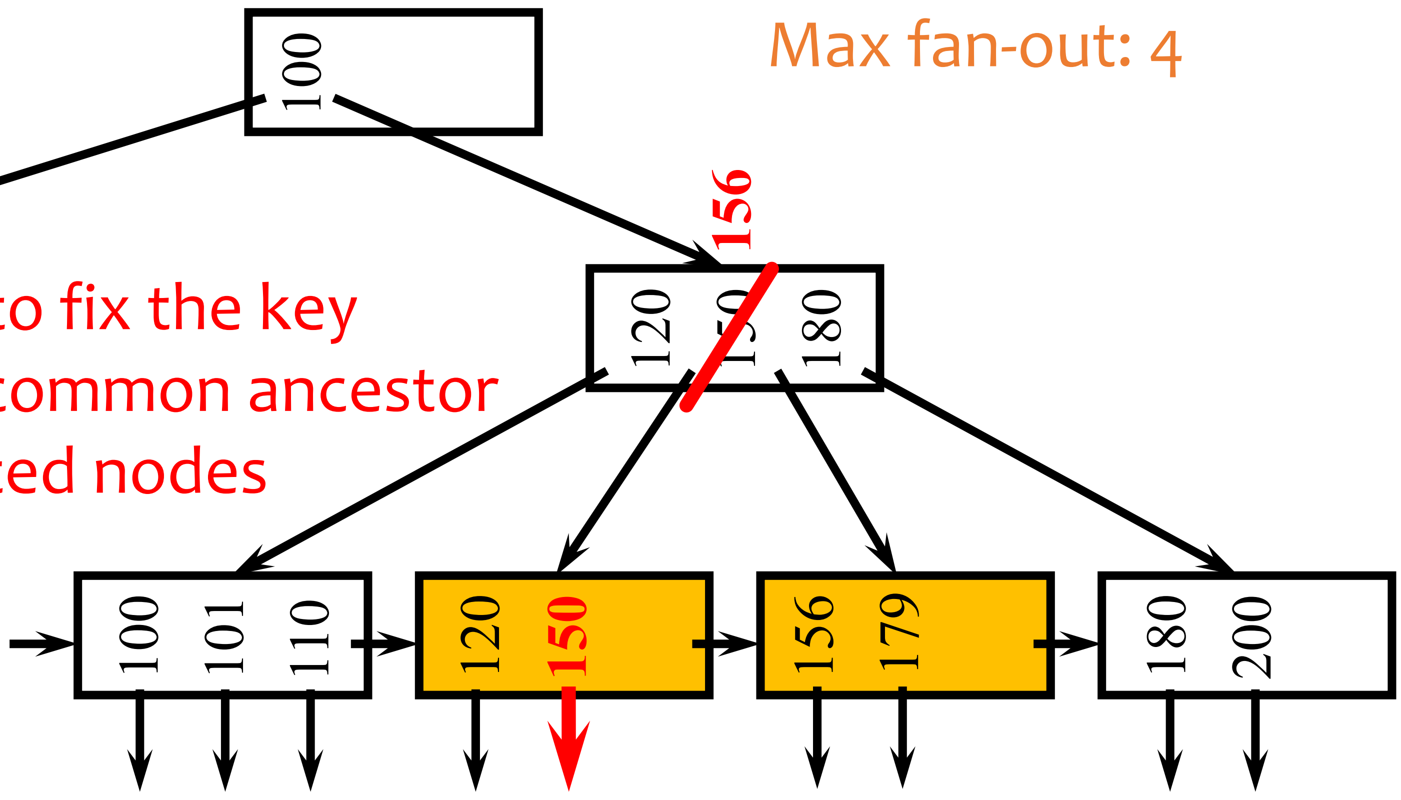
And delete it

Oops, node is too empty!

Stealing from a sibling

Max fan-out: 4

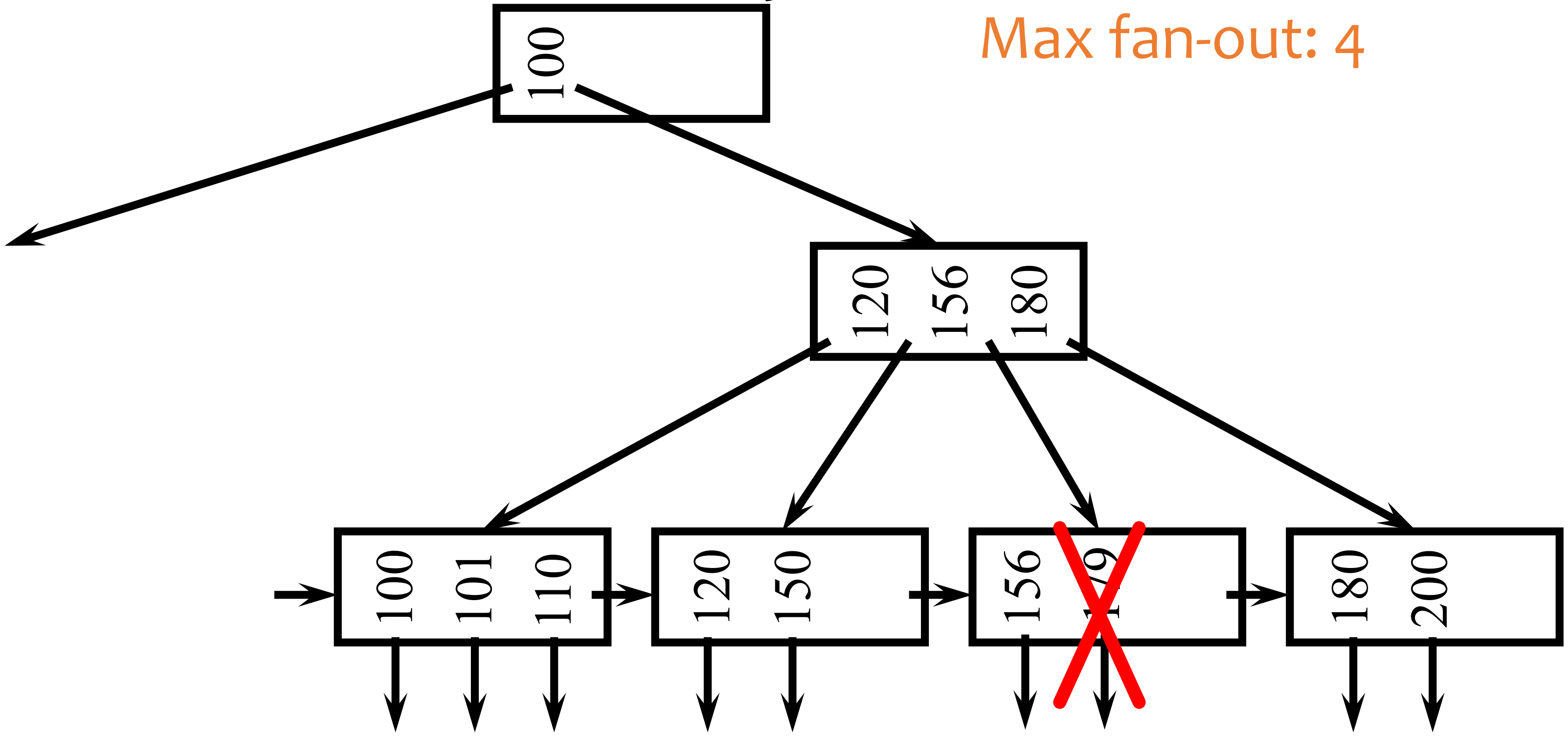
Remember to fix the key in the least common ancestor of the affected nodes



Another deletion example

- Delete a record with search key value 179

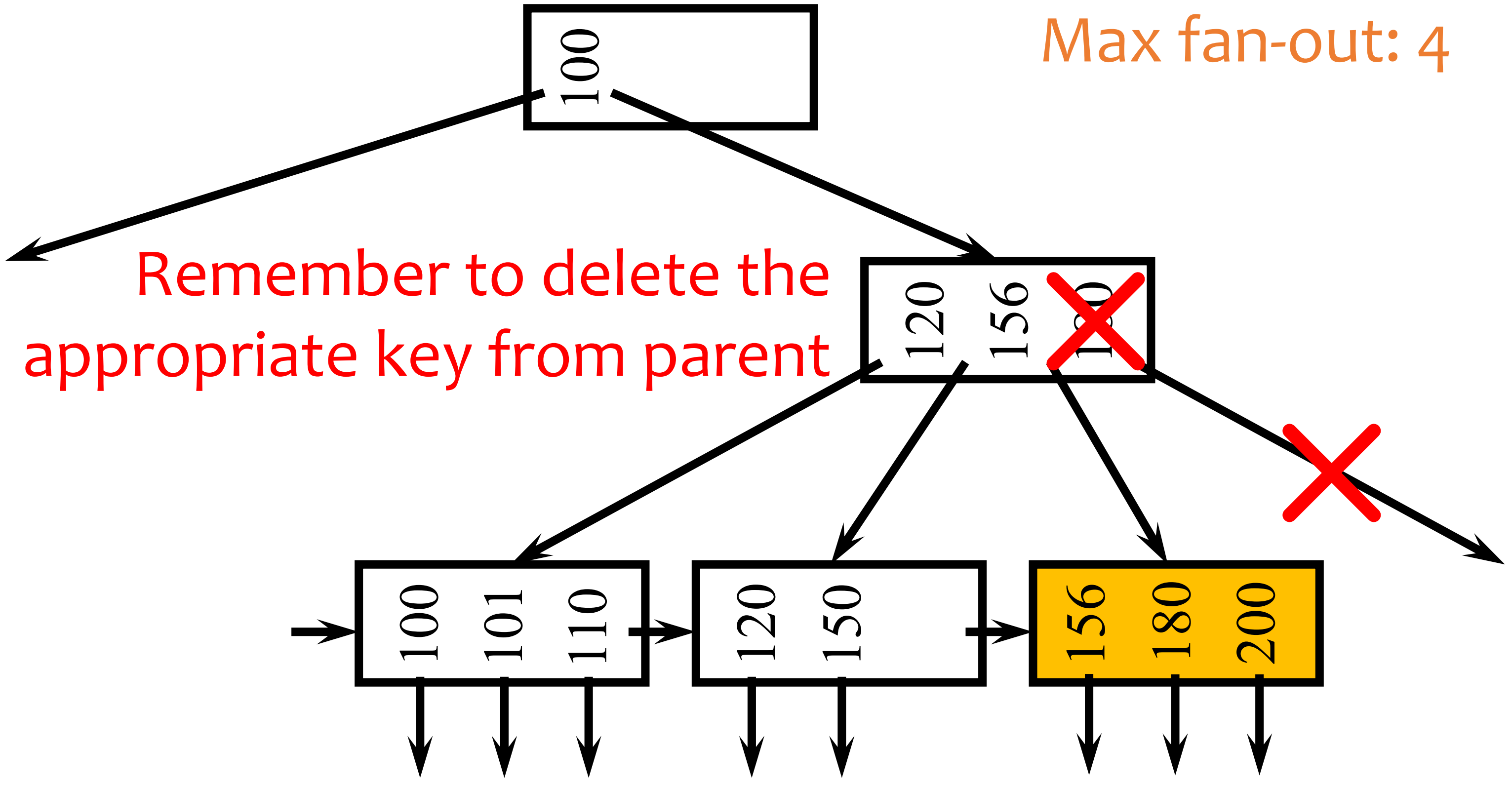
Max fan-out: 4



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

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

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
  - Records in leaves require more I/O’s to access
  - Vast majority of the records live in leaves!

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

- Other tree-based indexes: R-trees and variants, GiST, etc.
  - How about binary tree?

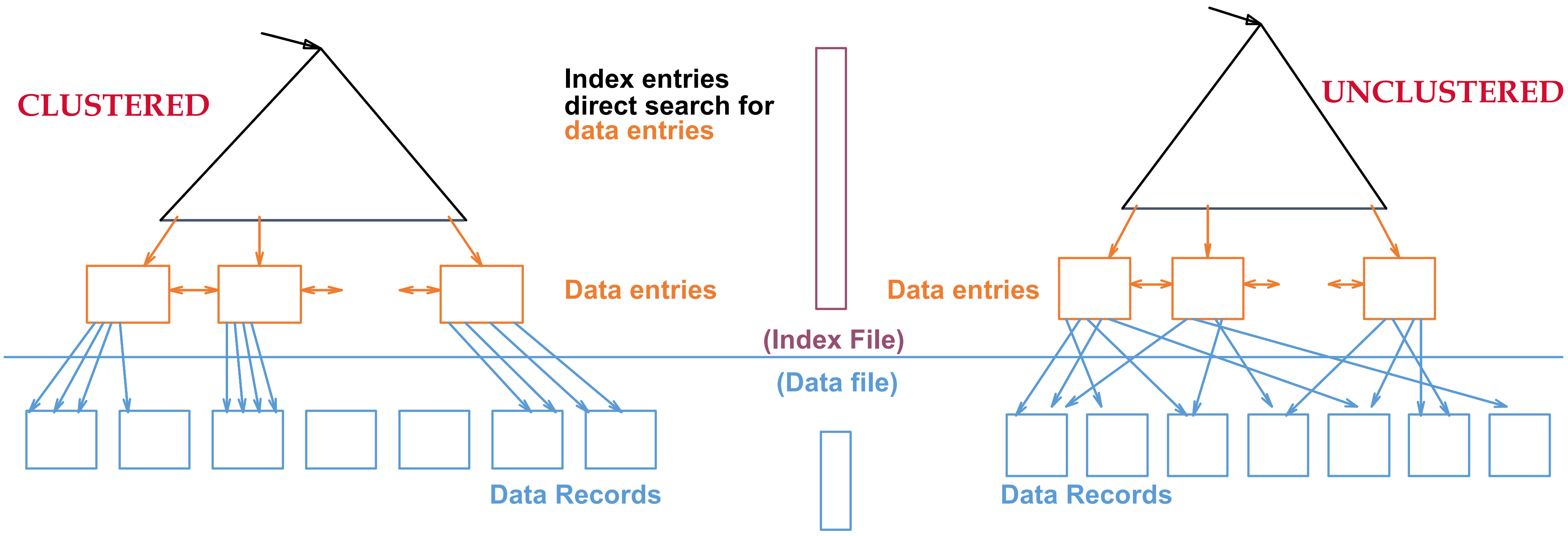
vs.

- Hashing-based indexes: extensible hashing, linear hashing, etc.
- Text indexes: inverted-list index, suffix arrays, etc.
- Other tricks: bitmap index, bit-sliced index, etc.

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)



Clustered vs. Unclustered Index

- How does it affect # of page accesses? (in class)
- SELECT * FROM USER WHERE age = 50
  - Assume 12 users with age = 50
  - Assume one page can hold 4 User records
  - Suppose accessing the data entry (-ies) require 3 IOs in a B+-tree, which contain pointers to the data records (all pointers in the same node)

Hash vs. Tree Index

- 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
  - 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?

- + But are more amenable to parallel processing
  - late hash-based join

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

- Details of hash-based dynamic index (extendible hashing, linear hashing) not covered in this class

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

<E.dno,E.sal>

Tree index!

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

<E.dno>

<E. age,E.sal>

Tree index!

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

- For index-only strategies, clustering is not important