Database Systems I

CMPT 354 Summer 2024

Zhengjie Miao

July 10 2024

Announcements (Wed. July 10)

- Update on Exam I solution
  - Grades will also be updated
- Makeup OH slots
  - Thu. July 11, 12:45pm - 1:30pm @TASC I 9407
  - Fri. July 12, 11:15am - 12:00pm @TASC I 9407

Examples of using indexes

- `SELECT * FROM User WHERE name = 'Bart’;`
- How is the query processed?
- Without an index on *User.name:* must scan the entire table if we store *User* as a flat file of unordered rows
- With index: go “directly” to rows with `name='Bart'`

Examples of using indexes

- ```
  SELECT * FROM User, Member
  WHERE User.uid = Member.uid AND Member.gid = 'cks’;
  ```
- How to find relevant *Member* rows directly?
  - With an index on *Member.gid* or (*gid*, *uid*):
- For each relevant *Member* row, how to directly look up *User* rows with matching *uid*?
  - With an index on *User.uid*
  - Without it: for each *Member* row, scan the entire *User* table for matching *uid*
    - Sorting could help
  ```

Indexes

- An index is an auxiliary persistent data structure
  - Search tree (e.g., B^+-tree), lookup table (e.g., hash table), etc.
- Creating and dropping indexes in SQL:
  - `CREATE [UNIQUE] INDEX` $indexname$ `ON` $tablename(columnname_1, \ldots, columnname_n);$
    - With `UNIQUE`, the DBMS will also enforce that $\{columnname_1, \ldots, columnname_n\}$ is a key of $tablename$
  - `DROP INDEX` $indexname;$

You will not have to create indices on these columns after using the unique keyword.

  - Typically, the DBMS will automatically create indexes for `PRIMARY KEY` and `UNIQUE` constraint declarations
- Can have many indexes for one table

Indexes

- An index on $R.A$ can speed up accesses of the form
  - $R.A = value$
  - $R.A > value$ (sometimes; depending on the index type)
- An index on $(R.A_1, \dots, R.A_n)$ can speed up
  - $R.A_1 = value_1 \wedge \cdots \wedge R.A_n = value_n$
  - $(R.A_1, \dots, R.A_n) > (value_1, \dots, value_n)$ (again depends)
- Ordering of index columns is important—is an index on $(R.A, R.B)$ equivalent to one on $(R.B, R.A)$?
- How about an index on $R.A$ plus another on $R.B$?

For the first case, if you access the A column it can be obtained easily. However, if you access the B column, accessing the data will not be as fast.

If you are checking that R.A equals some value and R.B equals another value, then you can have separate indices.

Ask professor to go over the difference of using separate vs joined indices

Choosing indexes to create

More indexes = better performance?

- Indexes take space
- Indexes need to be maintained when data is updated
- Indexes have one more level of indirection

- Optimal index selection depends on both query and update workload and the size of tables
  - Automatic index selection is now featured in some commercial DBMS

Automatic index problem is common

Choosing indexes to create

- Make some attribute K a search key if the WHERE clause contains:
  - An exact match on K
  - A range predicate on K
  - A join on K

The index selection problem 1

- Your workload is

Since name will have less duplication, the indices will serve more for optimizing the query

Age will have at most 100 values. So having a index of that would not be as beneficial compared to name

100000 queries

```
SELECT uid
FROM User
WHERE name = ?
```

100000 queries

```
SELECT uid
FROM User
WHERE age = ?
```

Which one is better?

A. Index on name

B. Index on age

The index selection problem 2

- Your workload is

100000 queries

```
SELECT uid
FROM User
WHERE name = ?
```

100000 queries

```
SELECT uid
FROM User
WHERE name = ? AND age > ?
```

For the first query, the first index is better to allow you quick access to the names

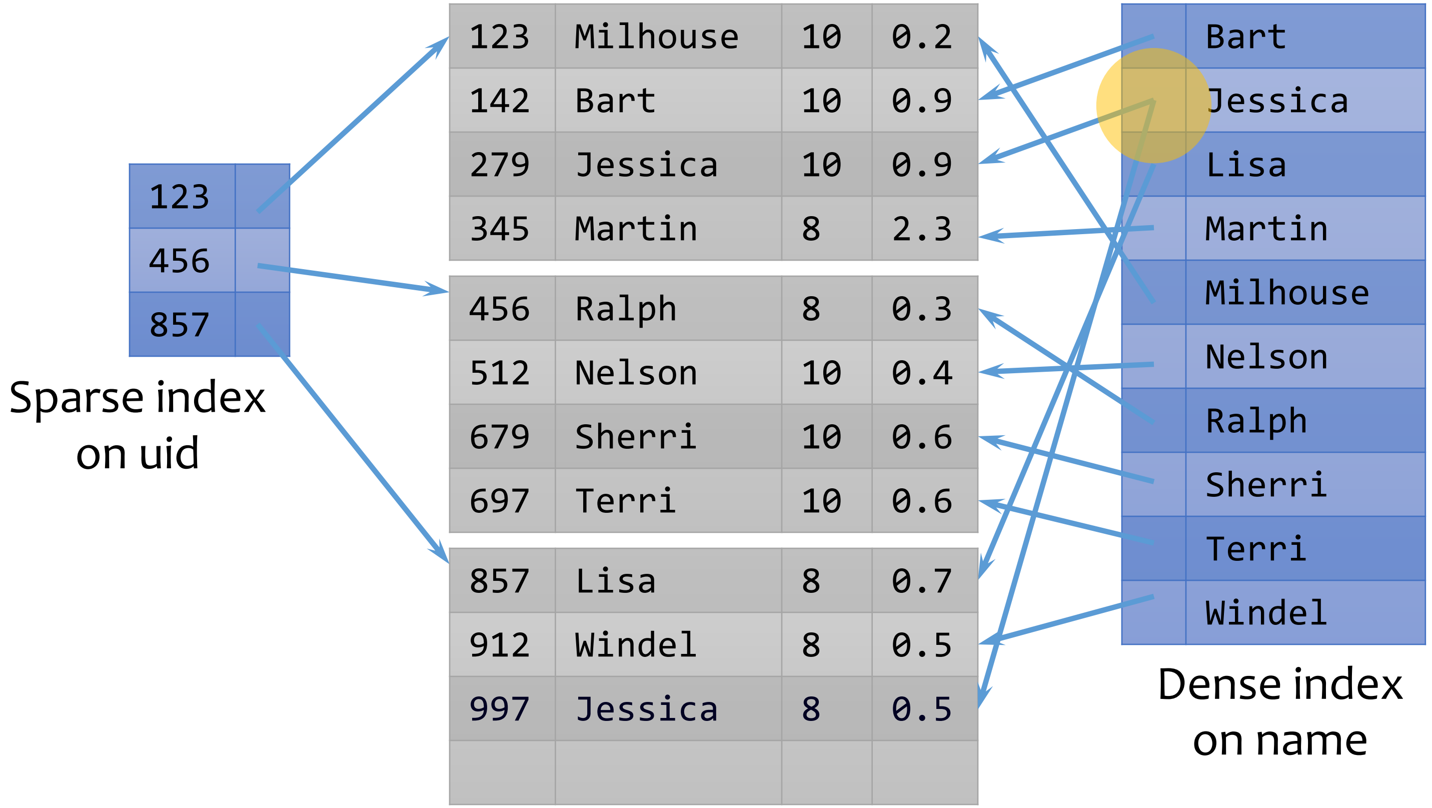
Which one is better?

A. Index on (name, age)

B. Index on (age, name)

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

One entry corresponds to one block

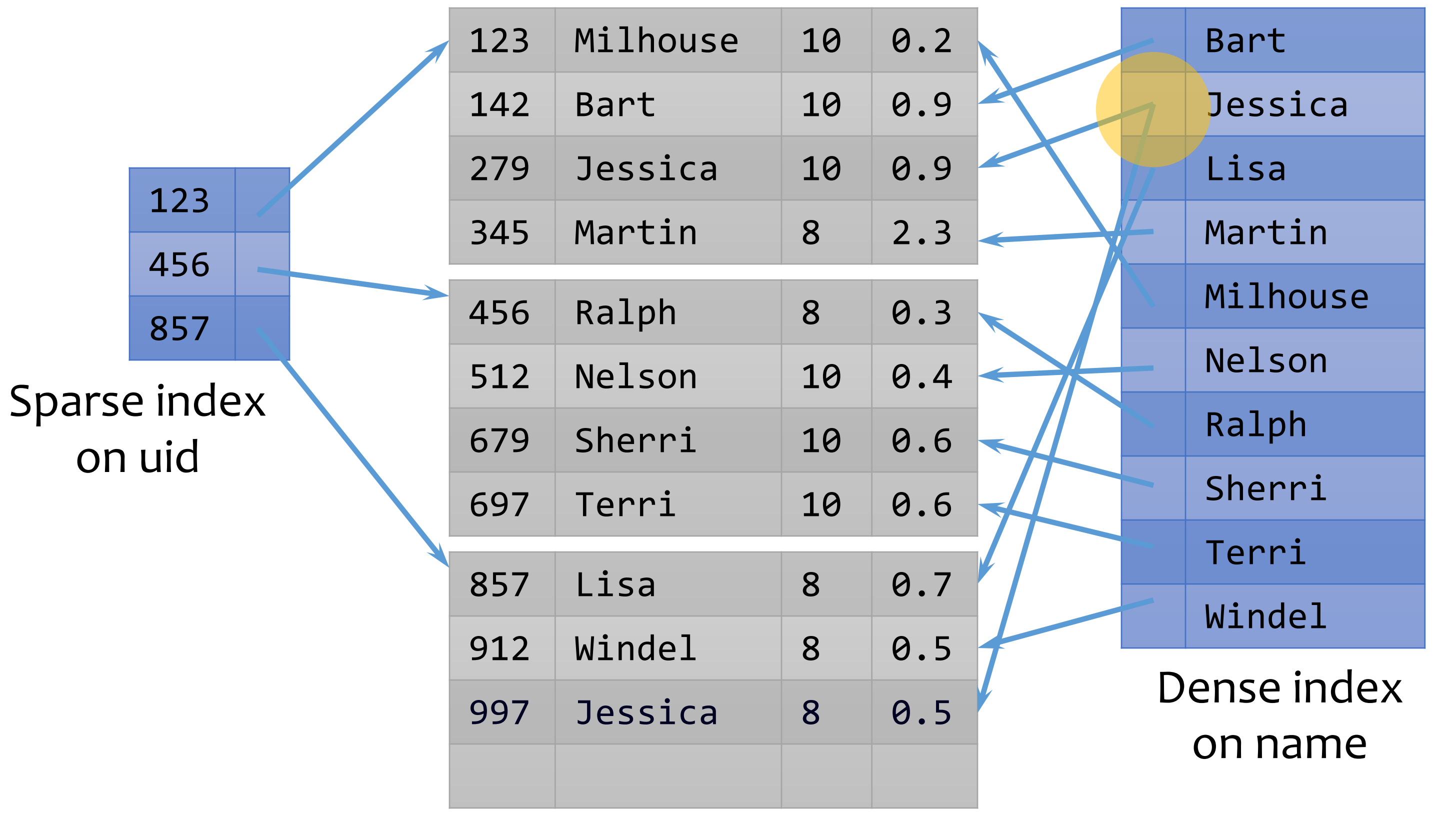
- Requirement on records
  - Records must be clustered for sparse index
- Lookup
  - Sparse index is smaller and may fit in memory
  - Dense index can directly tell if a record exists
- Update
  - Easier for sparse index

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
- In 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) too
    - `CREATE INDEX UserPopIndex ON User(pop);`

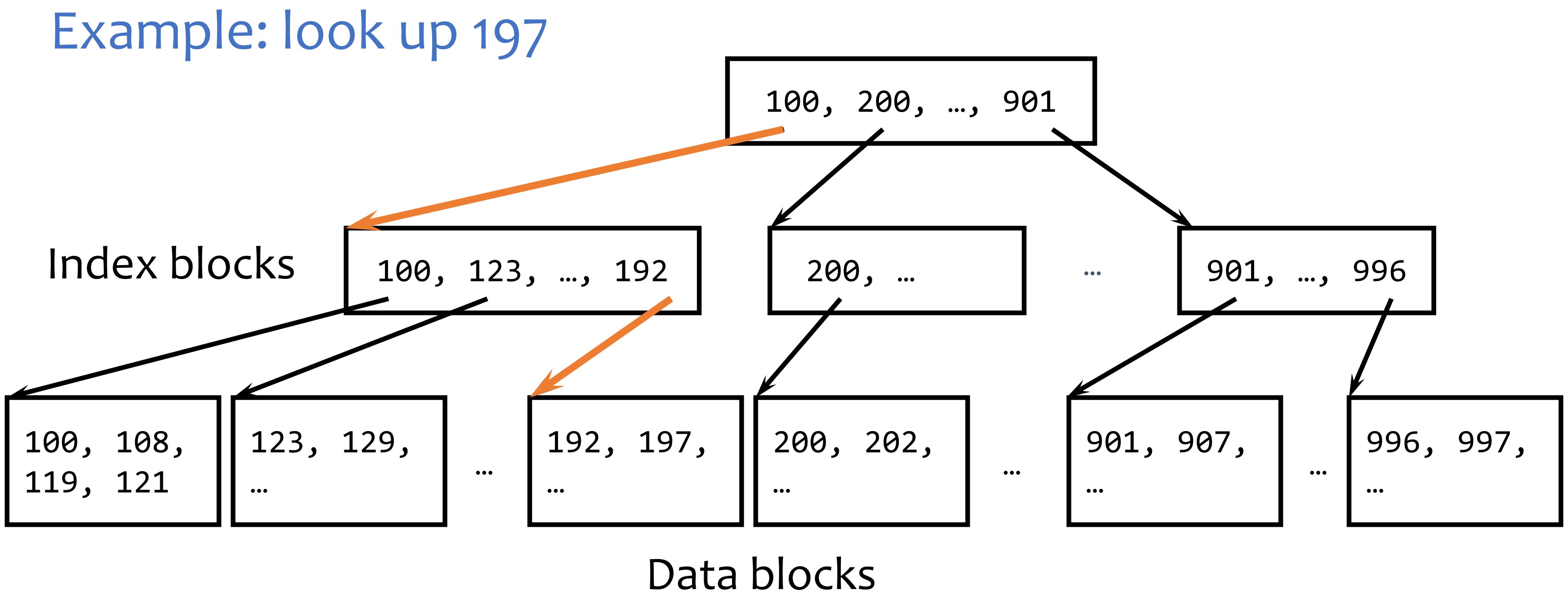
What if the index is too big as well?

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

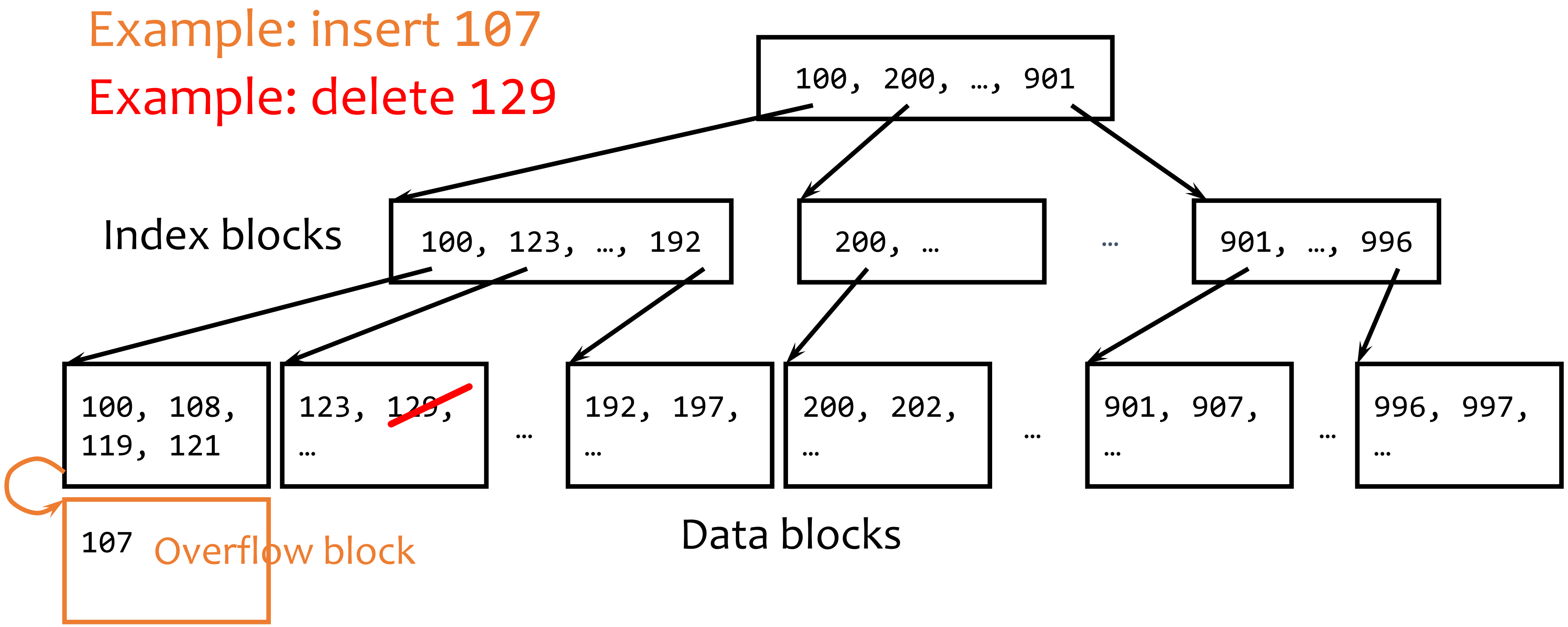


ISAM

- 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



Updates with ISAM

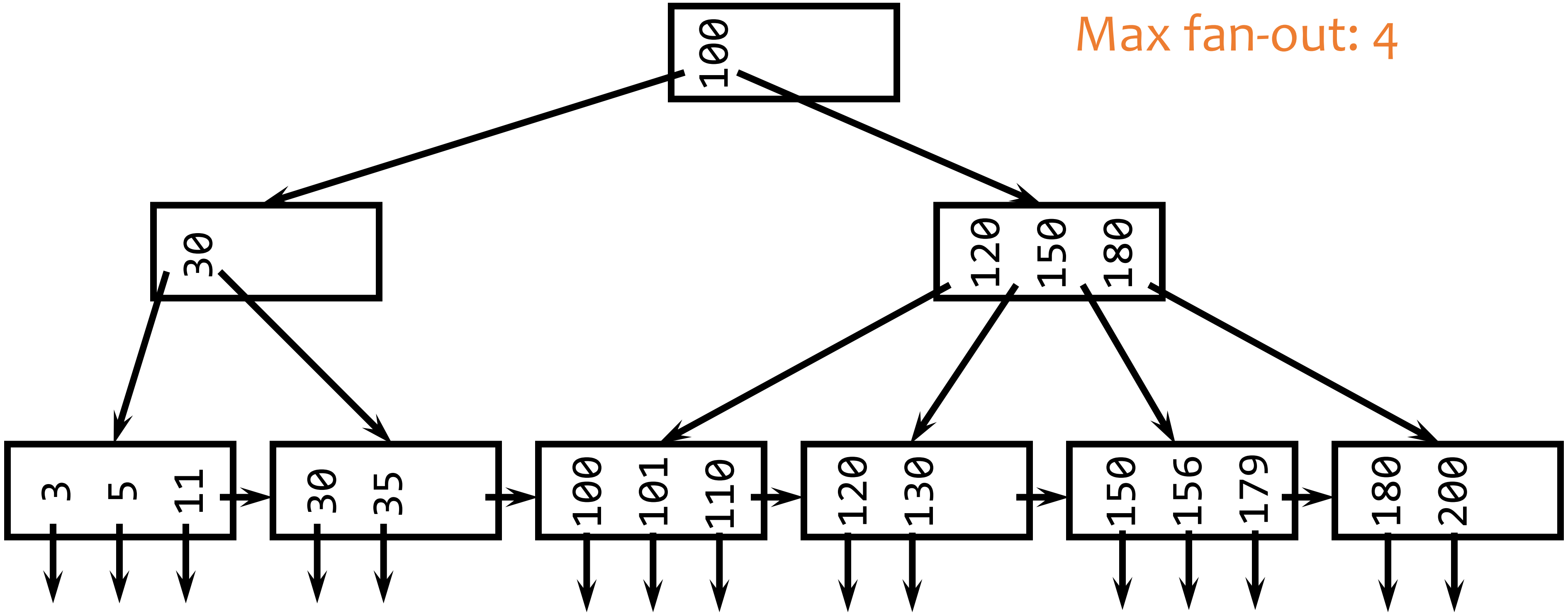


- Overflow chains and empty data blocks degrade performance
  - Worst case: most records go into one long chain, so lookups require scanning all data!

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

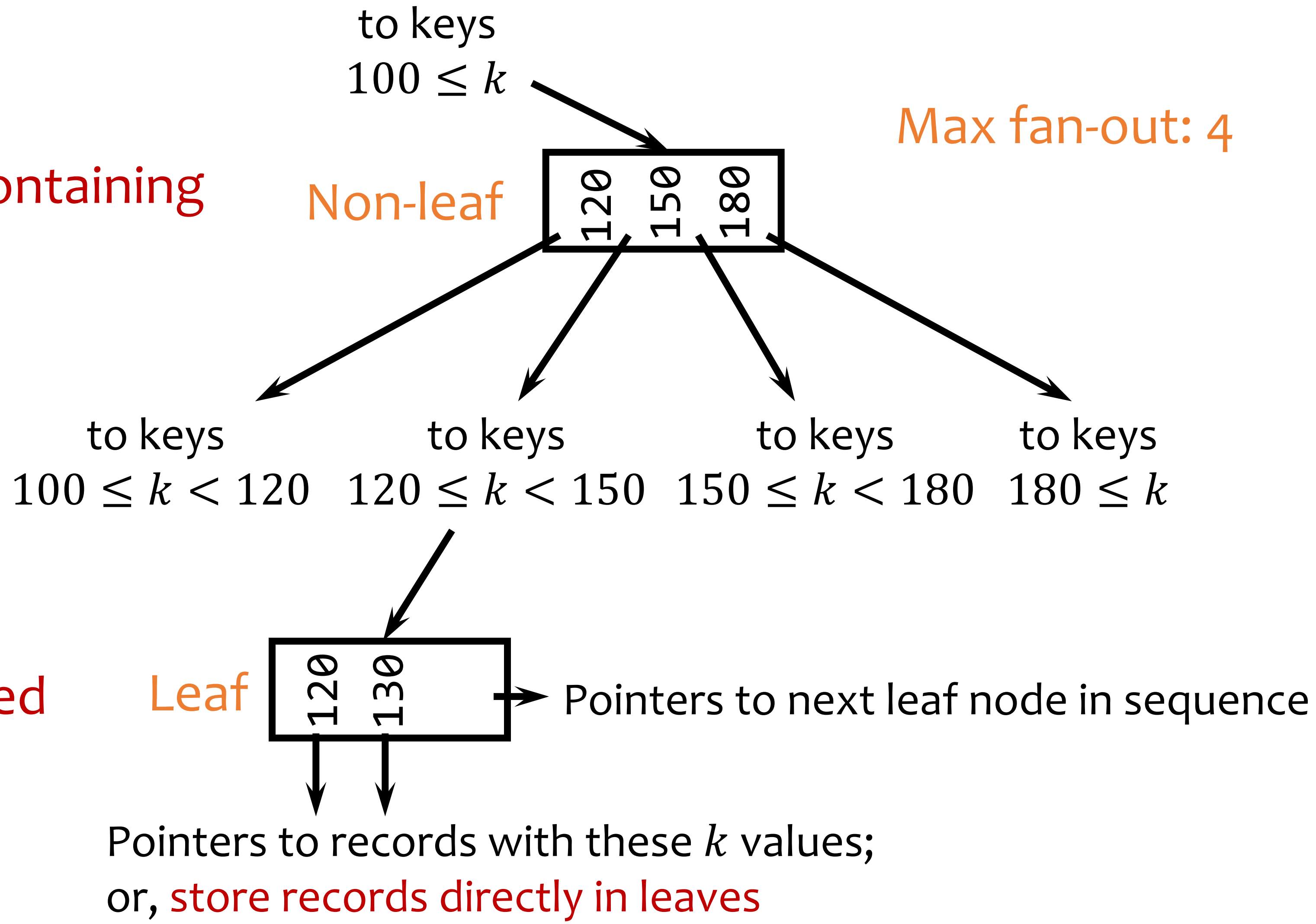


Sample B+-tree nodes

Index Nodes Containing
Index entries

Max fan-out: 4

Leaves are linked



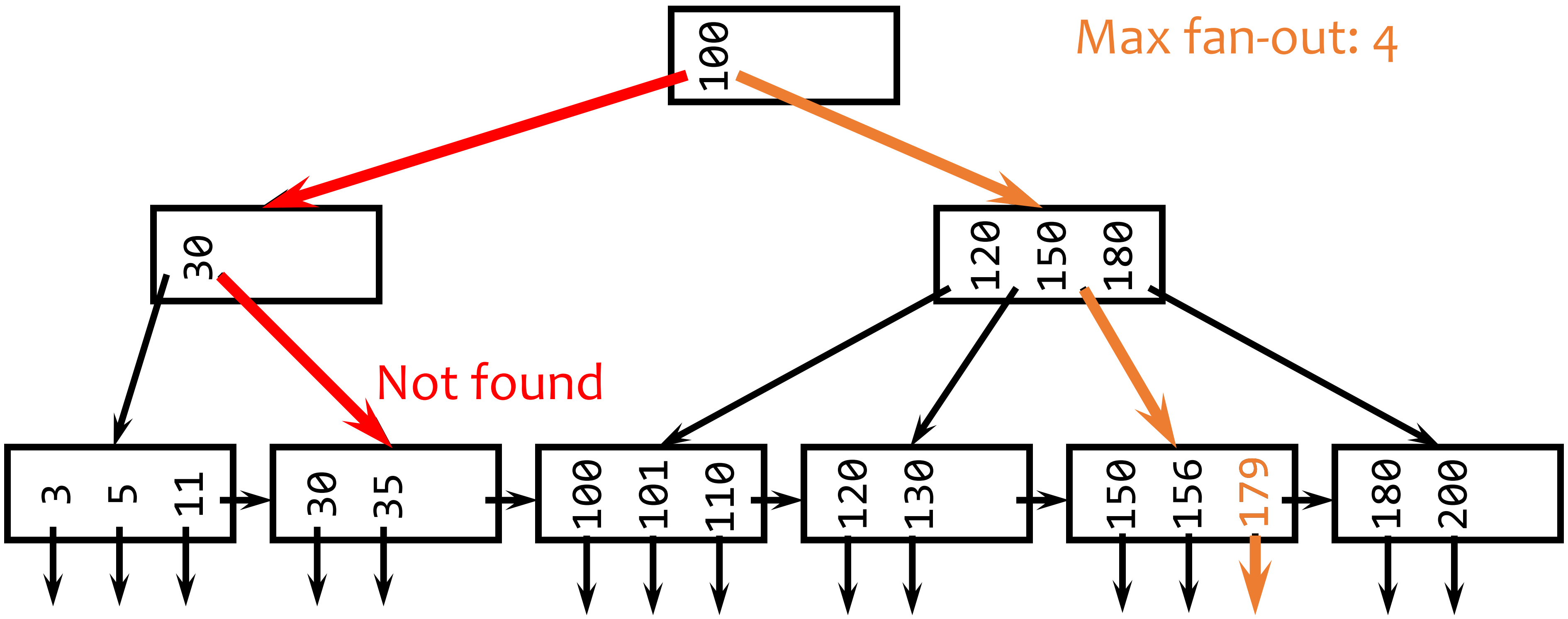
B^+-tree balancing properties

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

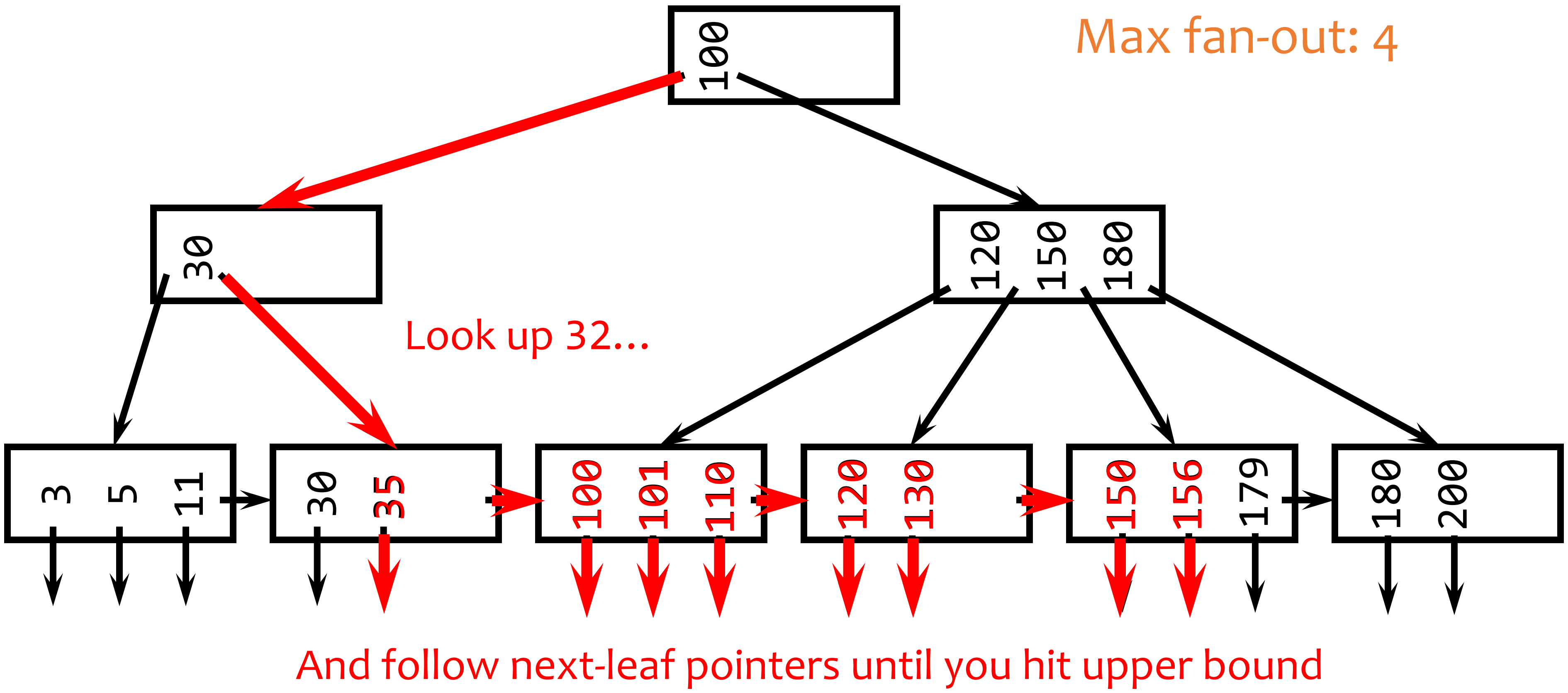
Lookups

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



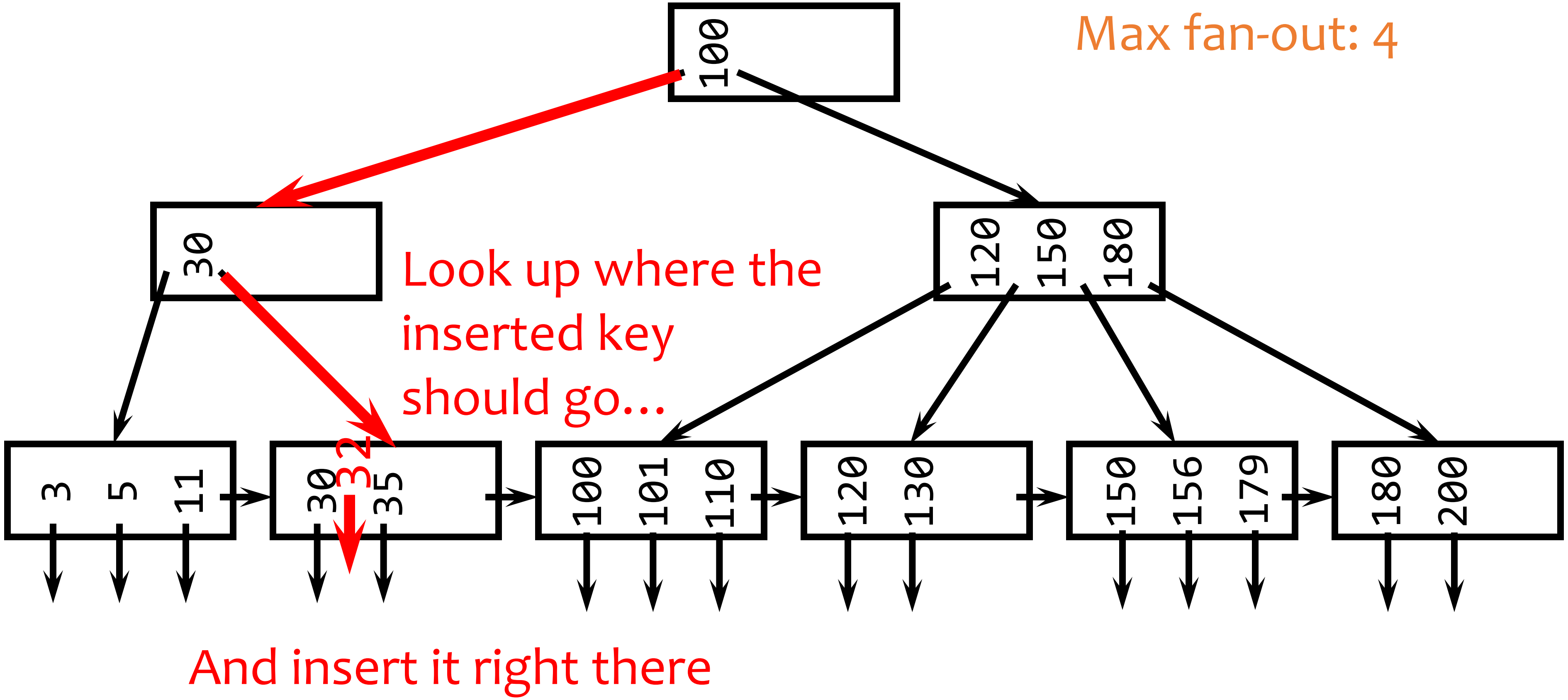
Range query

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



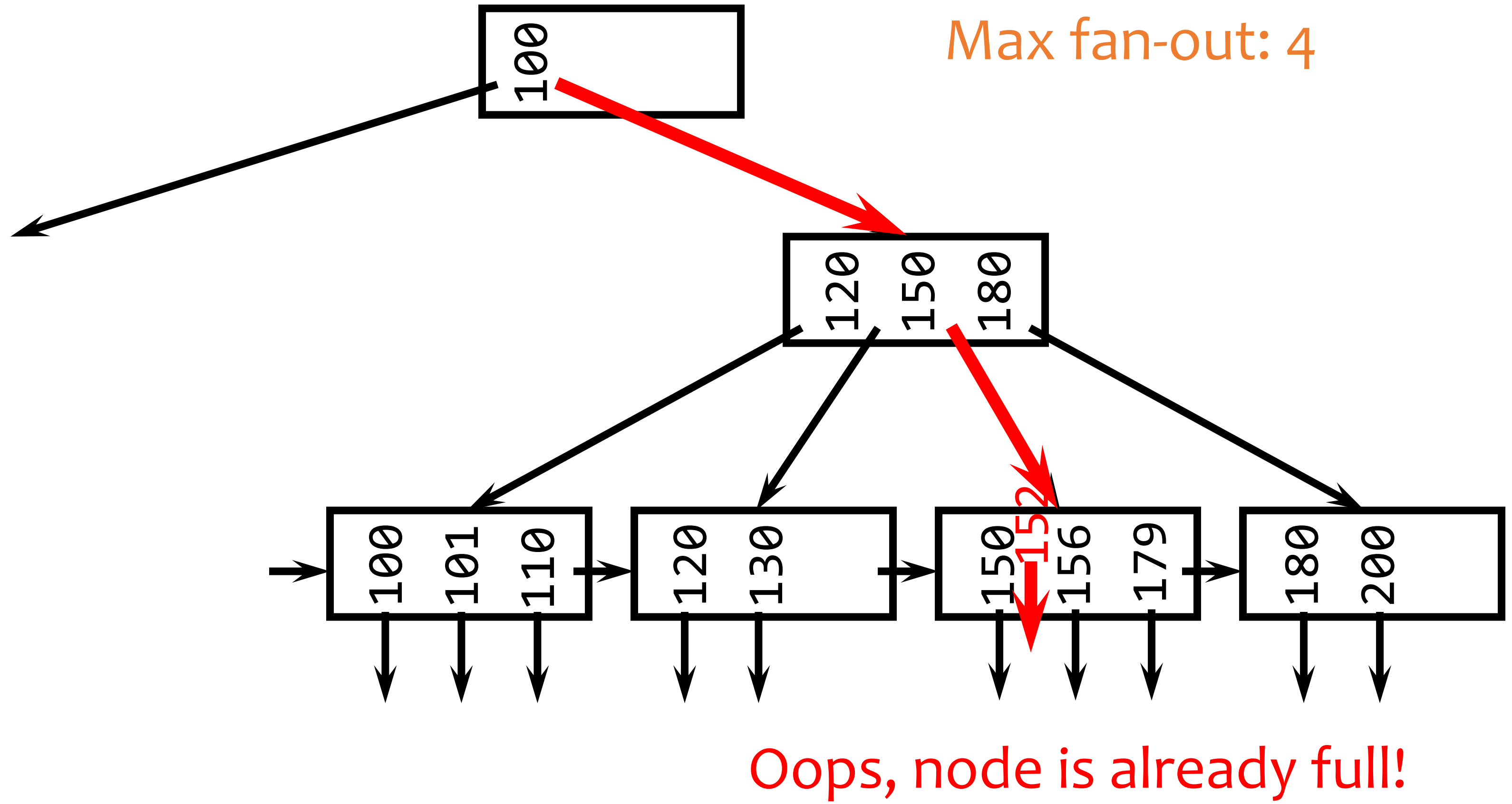
Insertion

- Insert a record with search key value 32

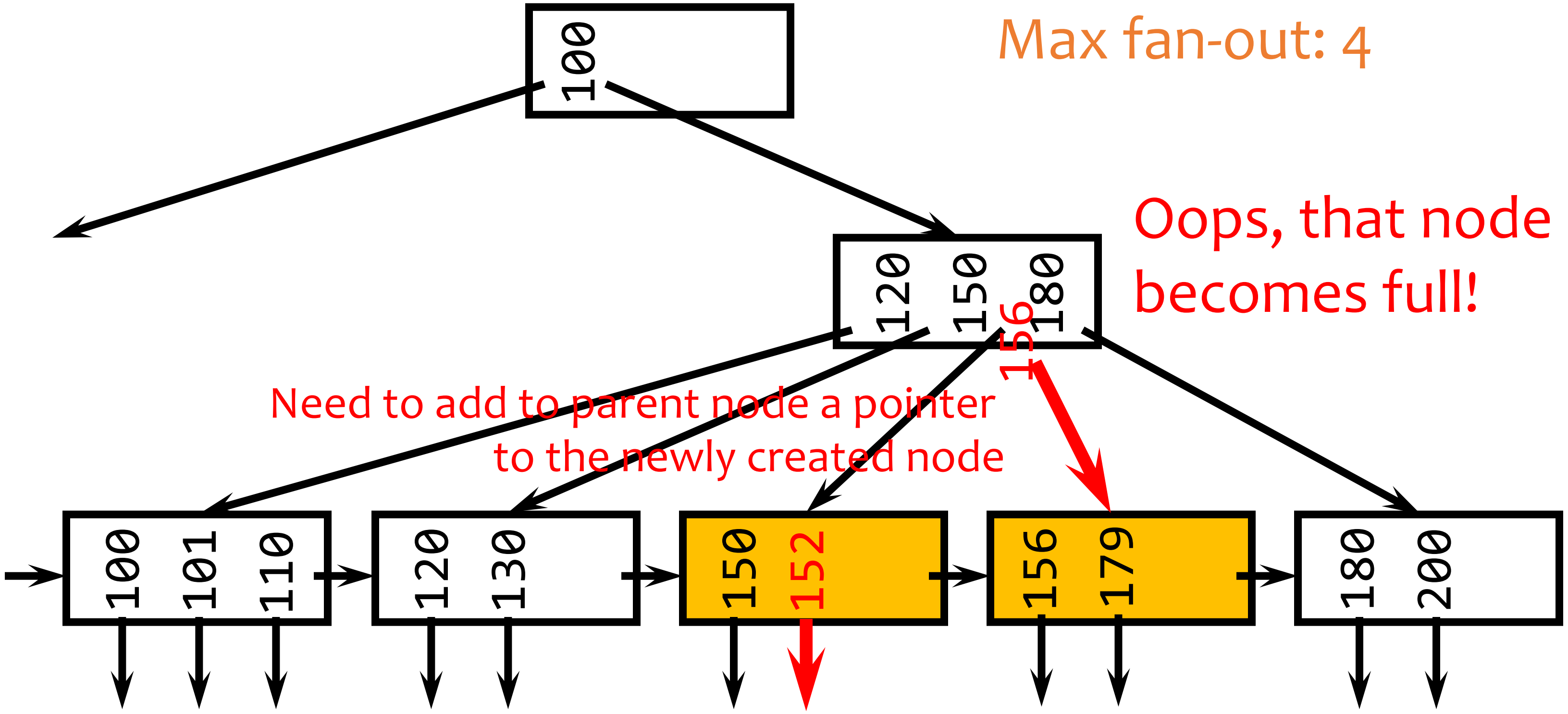


Another insertion example

- Insert a record with search key value 152



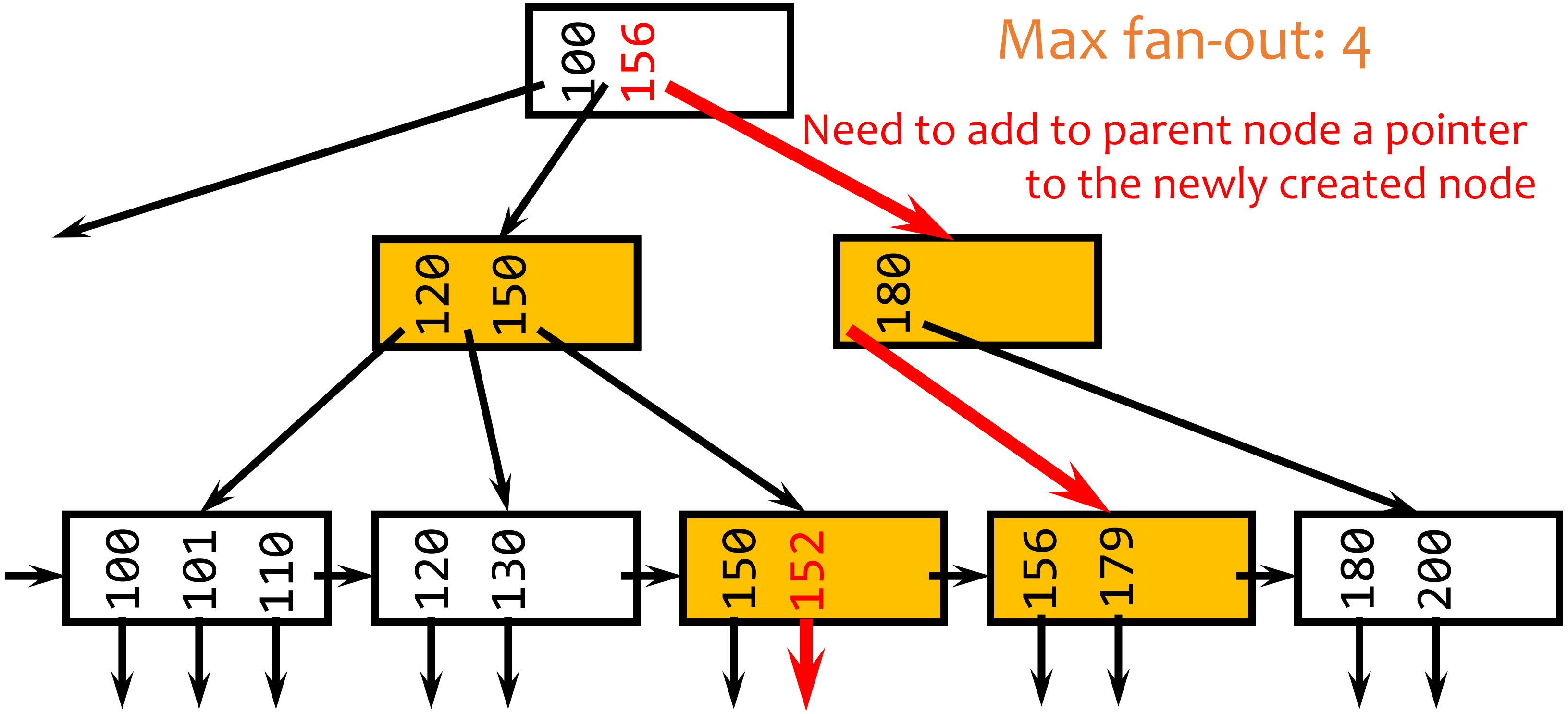
Node splitting



1. we "copy up" while splitting leaves – Insertion both at leaf and parent
2. the value inserted at parent may *not* be the new value we are inserting

More node splitting

We "push up" while splitting non-leaves, insertion ONLY at the parent node (from the middle)
so that we do not have a dangling pointer at non-leaves



- 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

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

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!

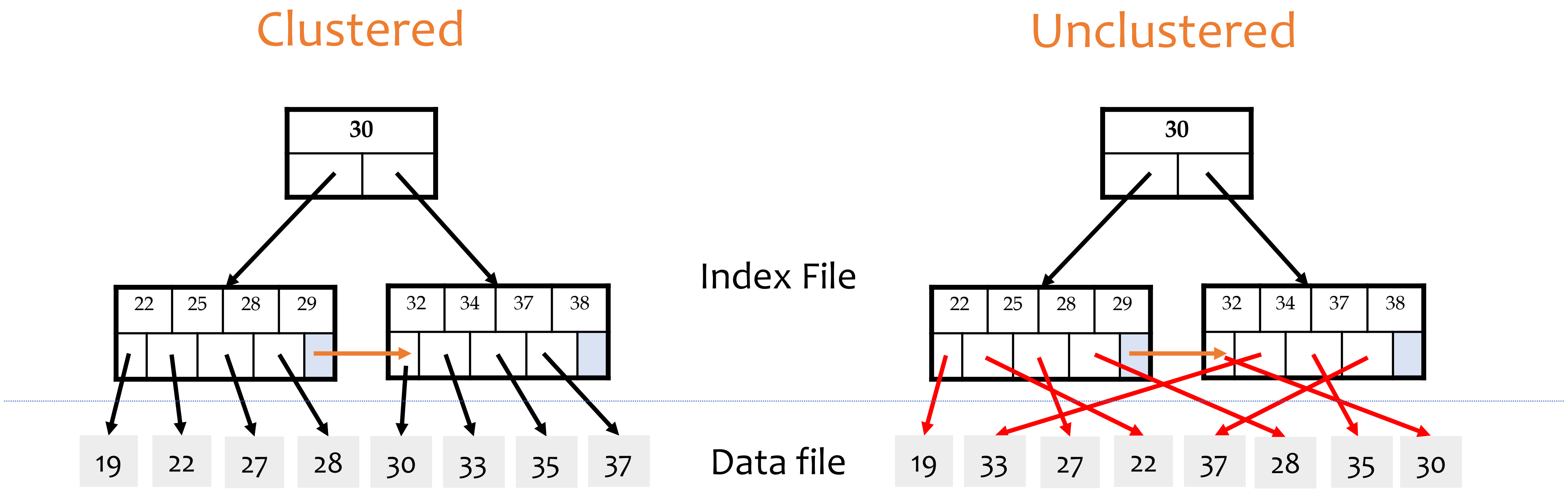
Beyond ISAM, B-, and B$^+$-trees (skip)

- Other tree-based indexes: R-trees, 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, vector database index, etc.

Clustered vs. Unclustered Index



How does it affect # of page accesses?

Recall that for a disk with block access, sequential IO is much faster than random IO

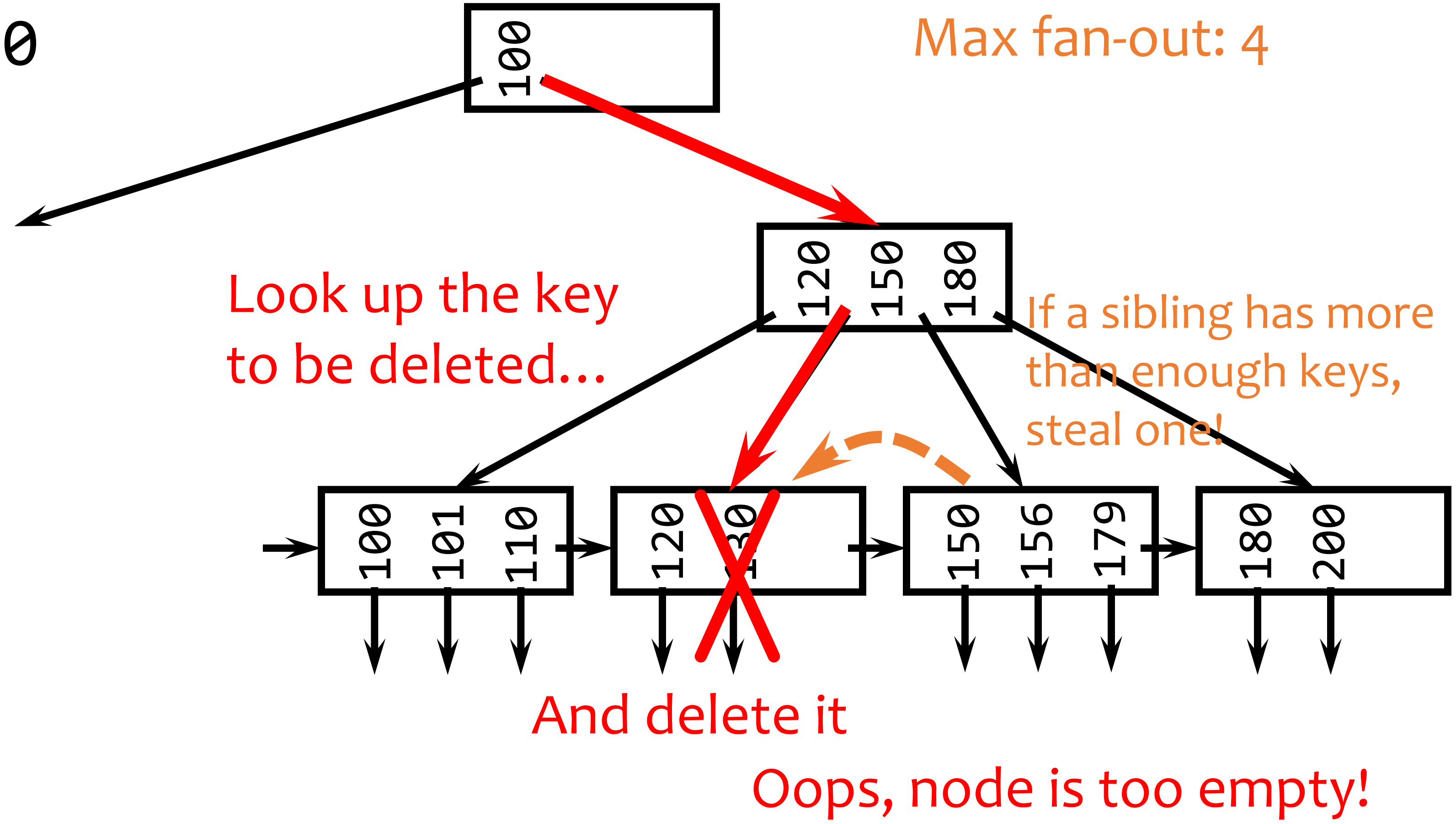
Data is sorted on search key   Data can be in any order

Clustered vs. Unclustered Index

- For range search over n values:
  - 1 random IO + n sequential IO vs. n random IO
- 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 = data entries are in the same node of B+-tree)
  - What happens if the index is unclustered? (cost = 3+12)
  - What happens if the index is clustered? (cost <= 3 +(3 +1))

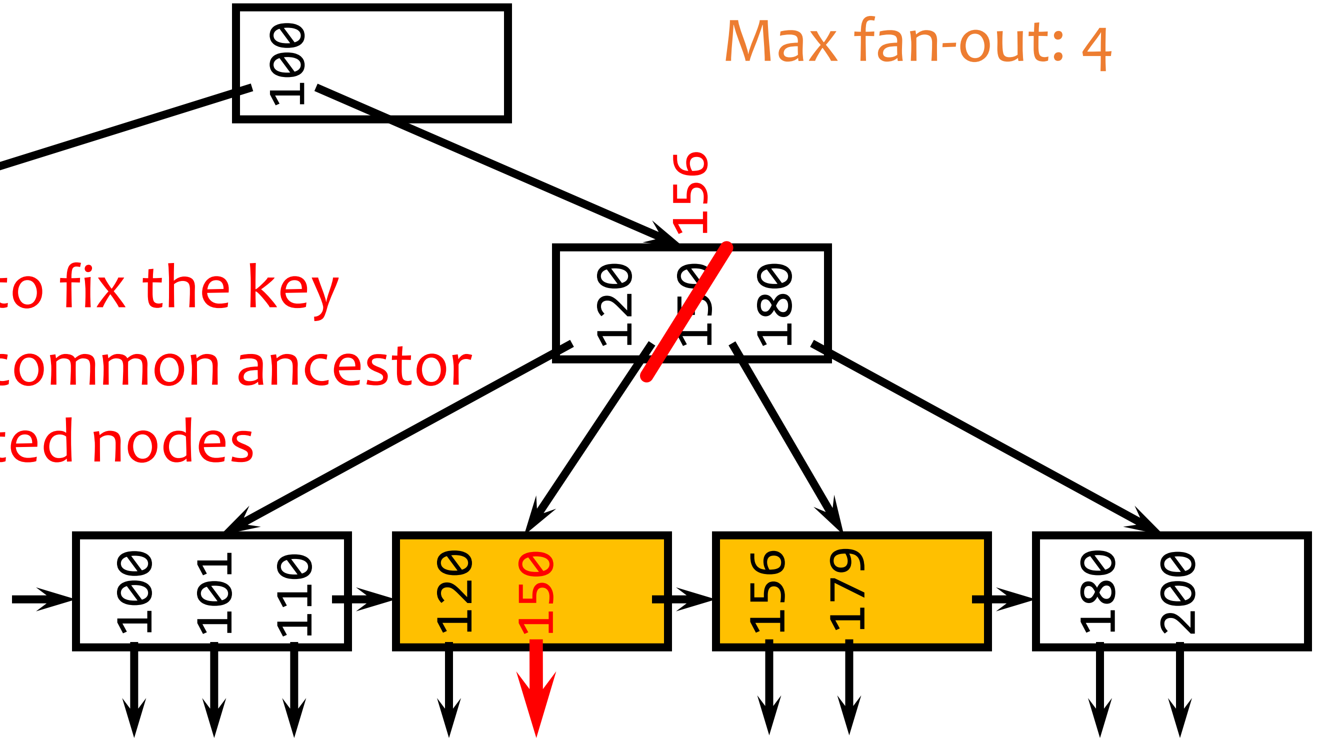
Deletion (skip)

- Delete a record with search key value `130`



Stealing from a sibling (skip)

Max fan-out: 4

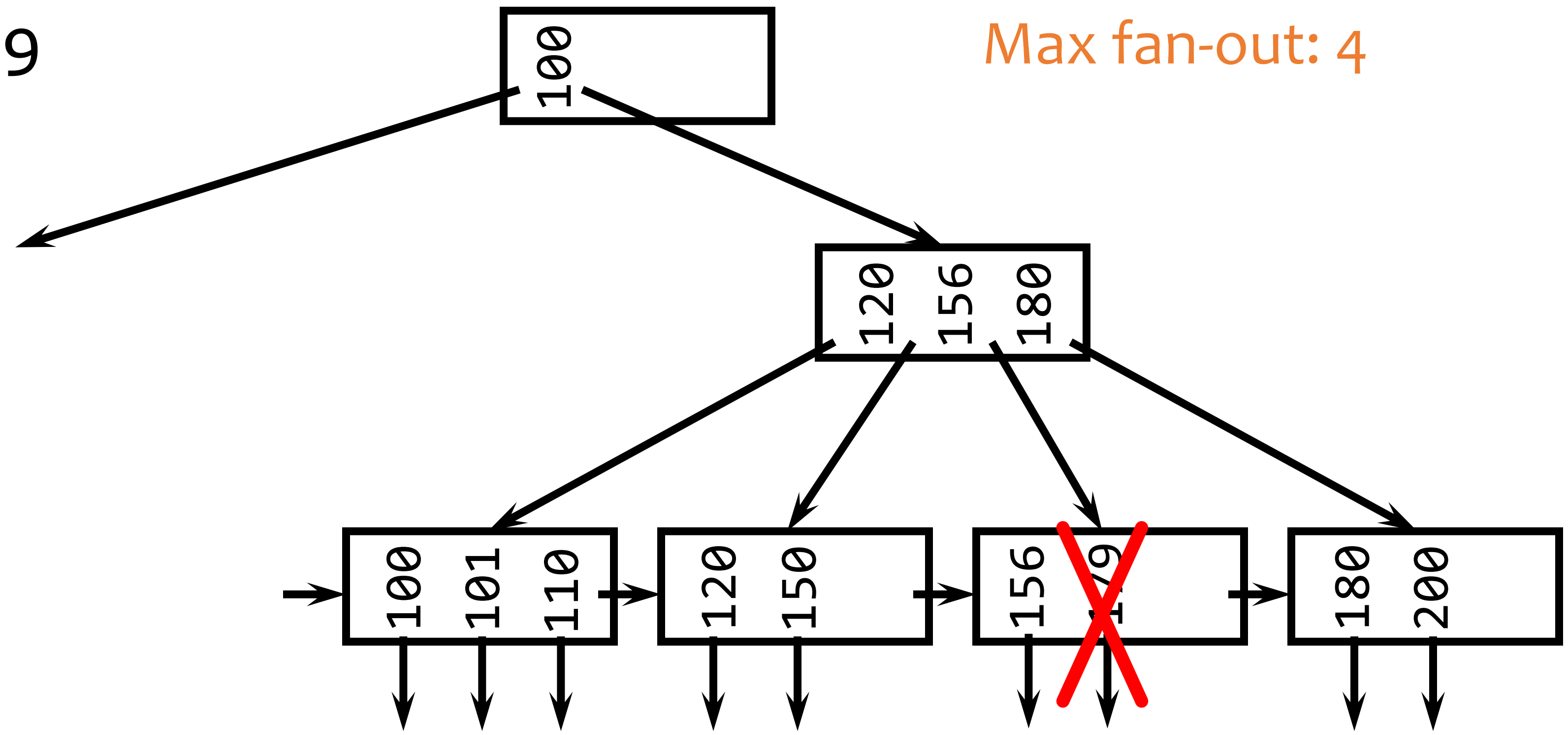


Remember to fix the key
in the least common ancestor
of the affected nodes

Another deletion example (skip)

- Delete a record with search key value 179

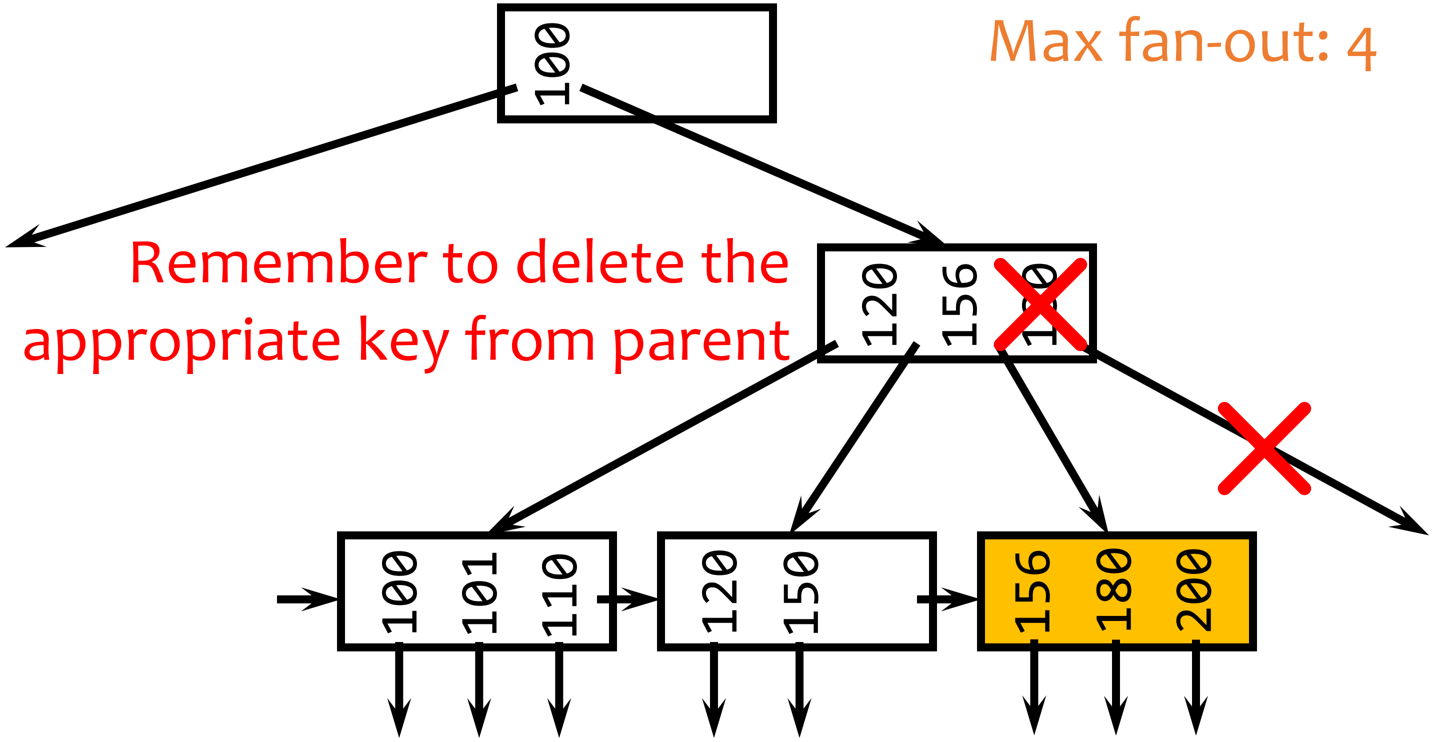
Max fan-out: 4



Cannot steal from siblings

Then coalesce (merge) with a sibling!

Coalescing (skip)



- 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