Index

Introduction to Databases

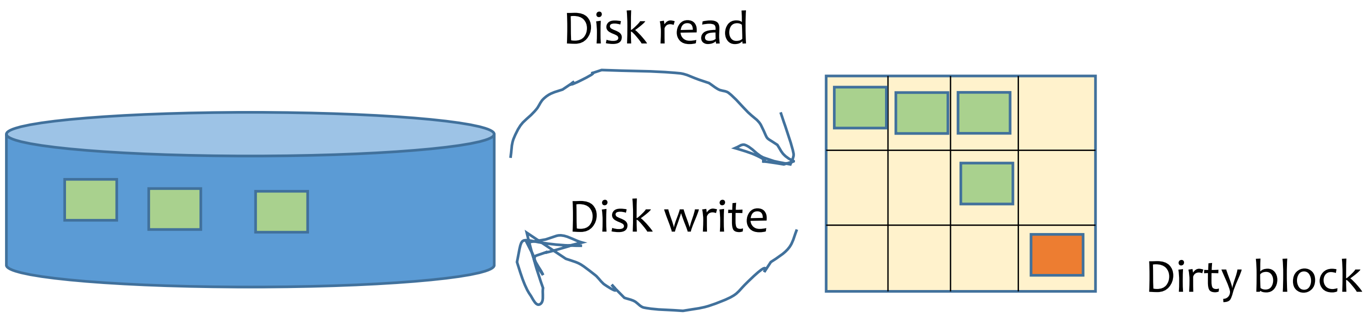
CompSci 316 Fall 2022

DUKE
COMPUTER SCIENCE

Announcements (Thu. Oct 13)

- Project, project, & project!
  - MS-2 due today (10/13)
  - HW4 due 10/20 - group submission per project team
  - DS7 – team work for project & HW4

Recall the Disk-Main Memory diagram!



Topics

- 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

Focus of this lecture

Dense and sparse indexes

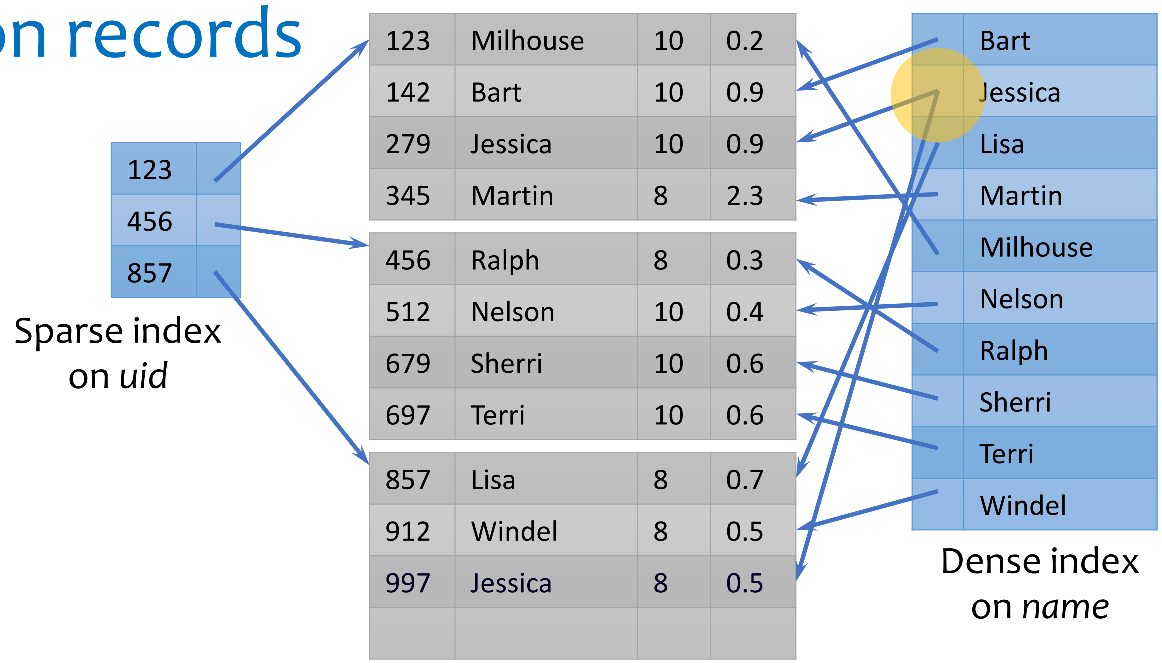
When are these possible?

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
  - ??



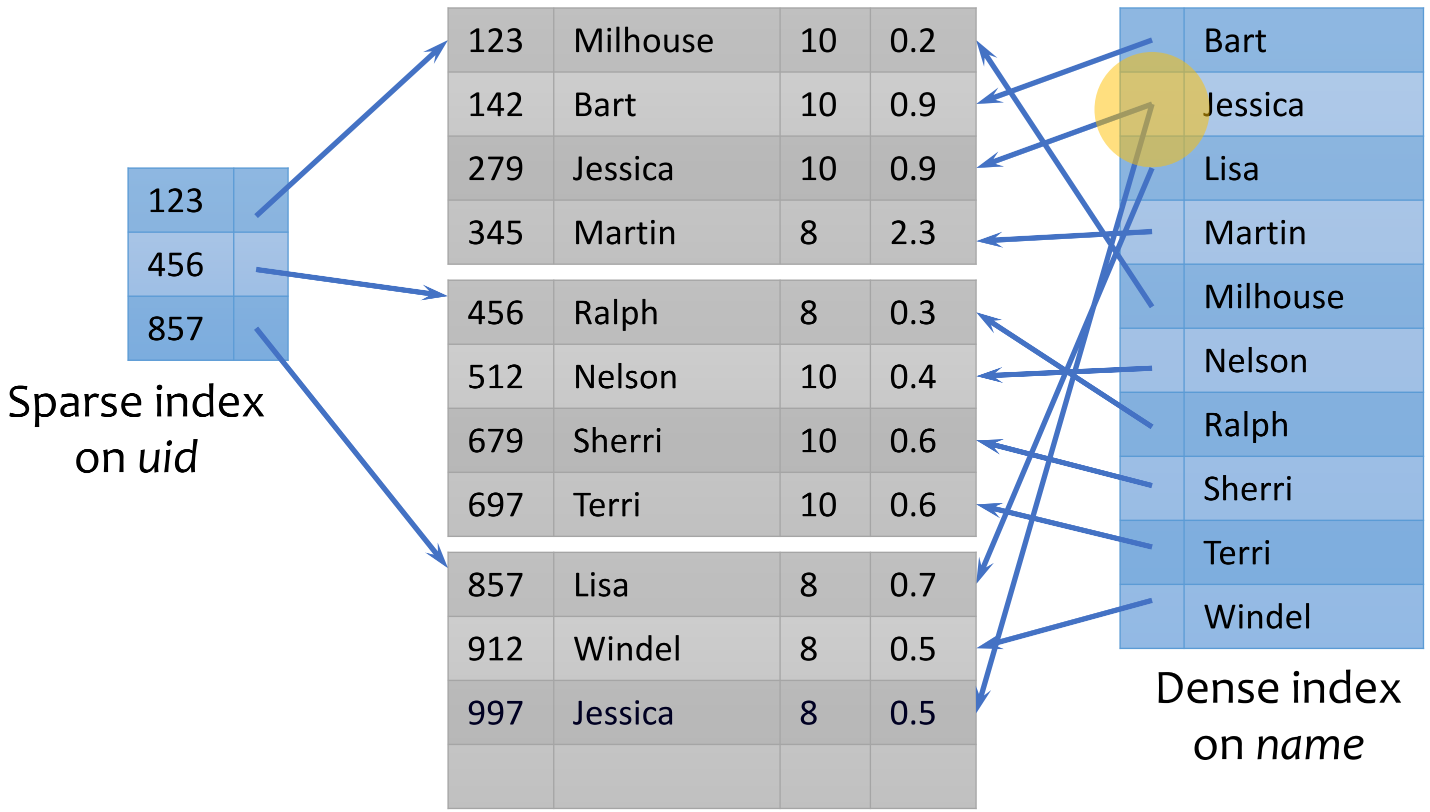
Dense versus sparse indexes

- 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?

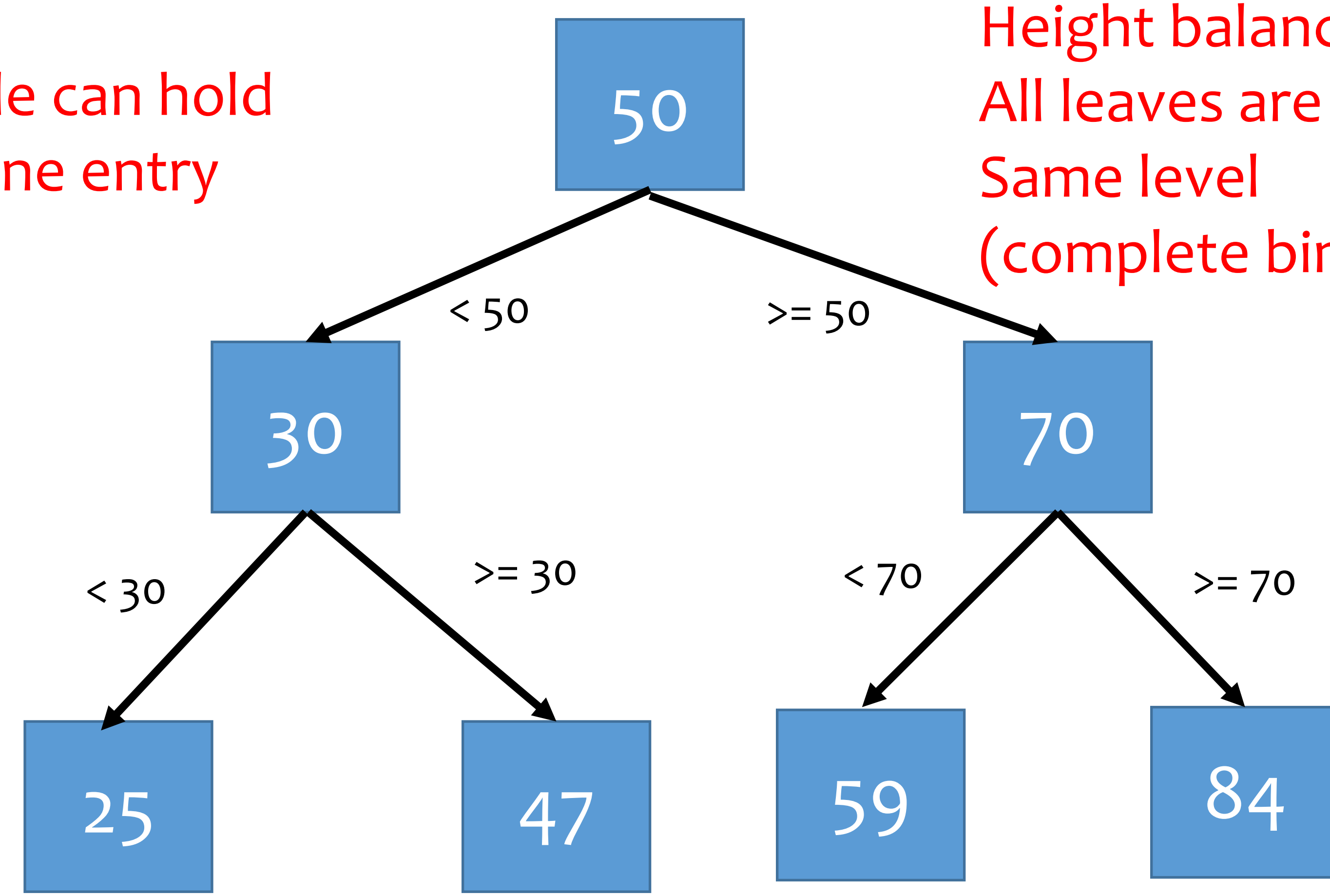
Put a another (sparse) index on top of that!

Binary Search Tree

End of Lecture 10/6
(some earlier slides were skipped
That will be covered next weeks)

Each node can hold
Exactly one entry

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



Leaves are sorted

Remember Terminology

- Index search key (key): k
  - Used to search a record

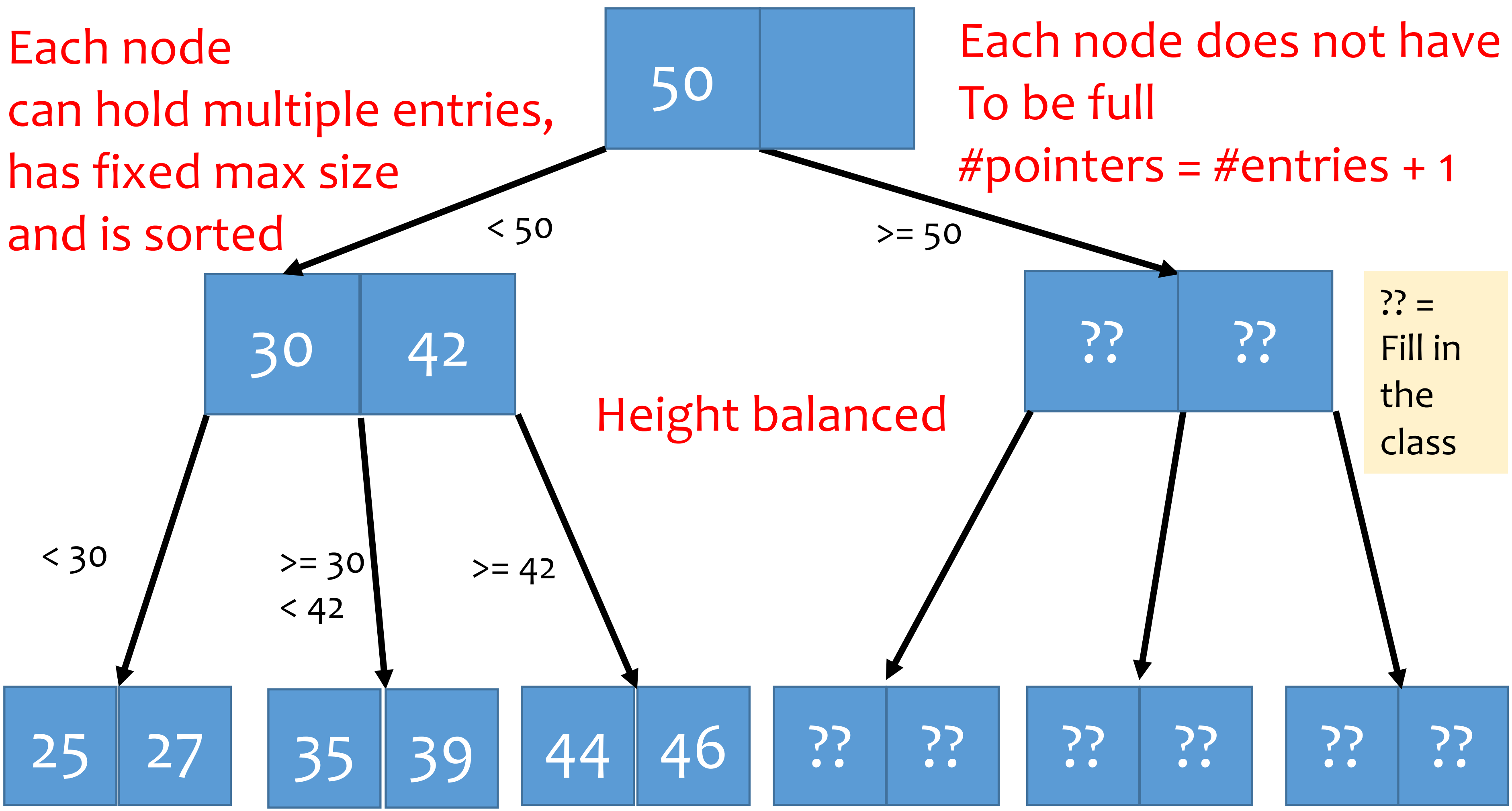
INDEX
does this

- Data entry : k*
  - Pointed to by k
  - Contains record id(s)
    - (-) another level of indirection (+) small and fixed length entries
  - or record itself
    - (-) can be large, cannot be stored in memory, (+) saves some disk access

- Records or data
  - Actual tuples
  - Pointed to by record ids

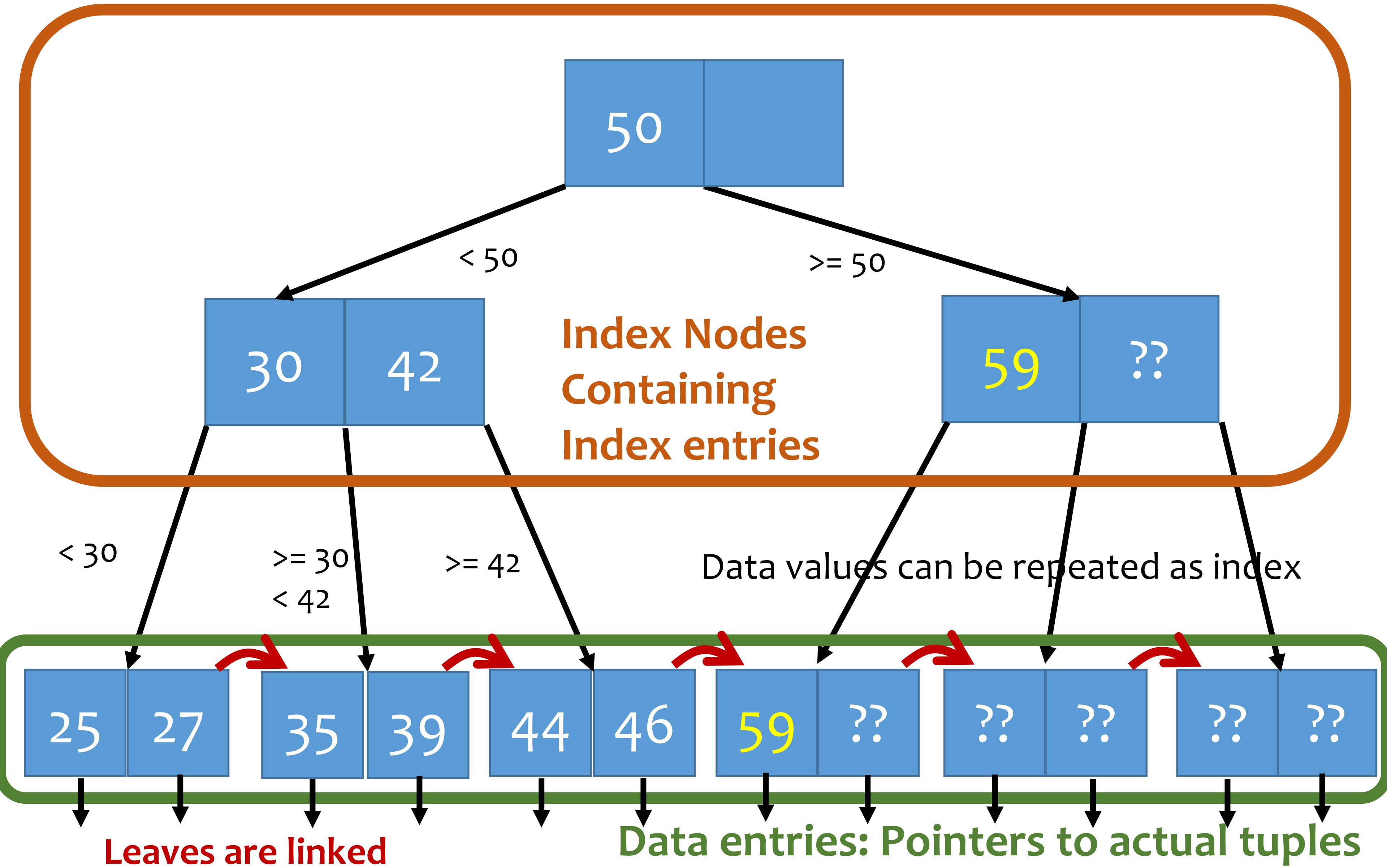
On disk

B-tree: Generalizing Binary Search Trees



Leaves are sorted

B+-tree: Data only at leaves



B+-tree balancing properties

Check yourself

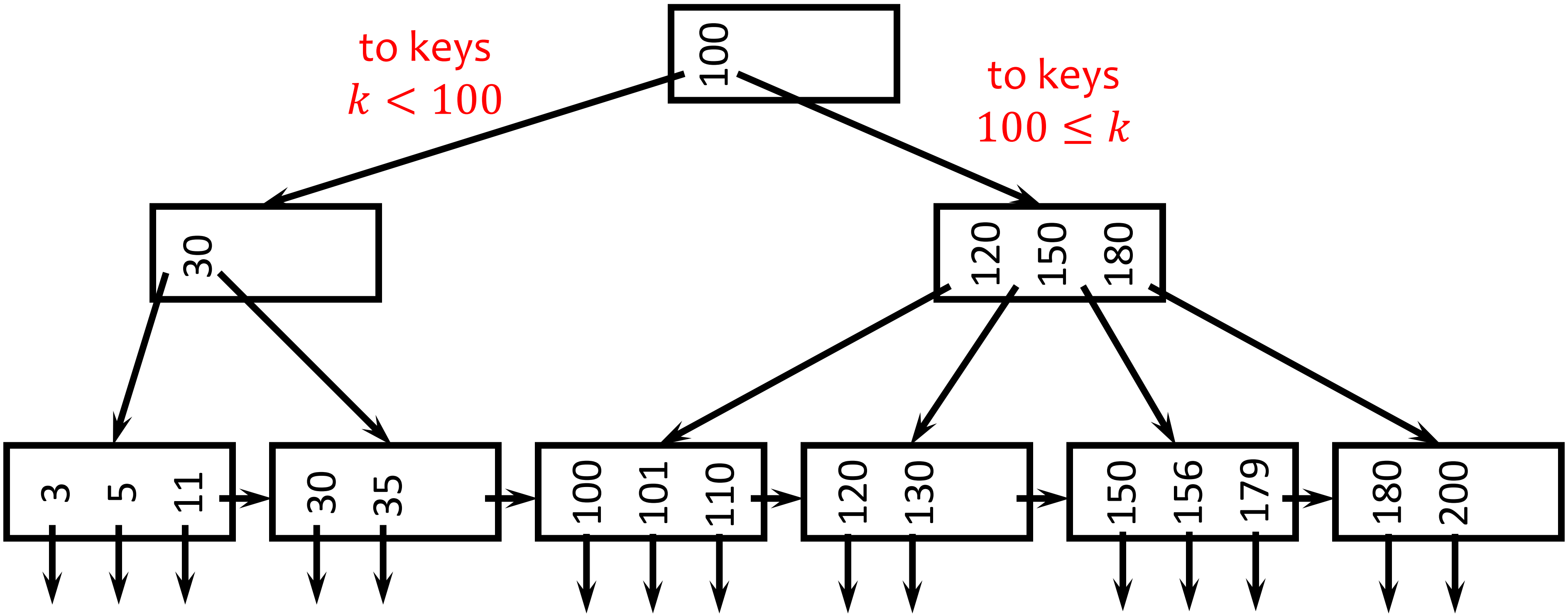
- Height constraint: all leaves at the same lowest level
- Fan-out constraint: all nodes at least half full (except root)

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

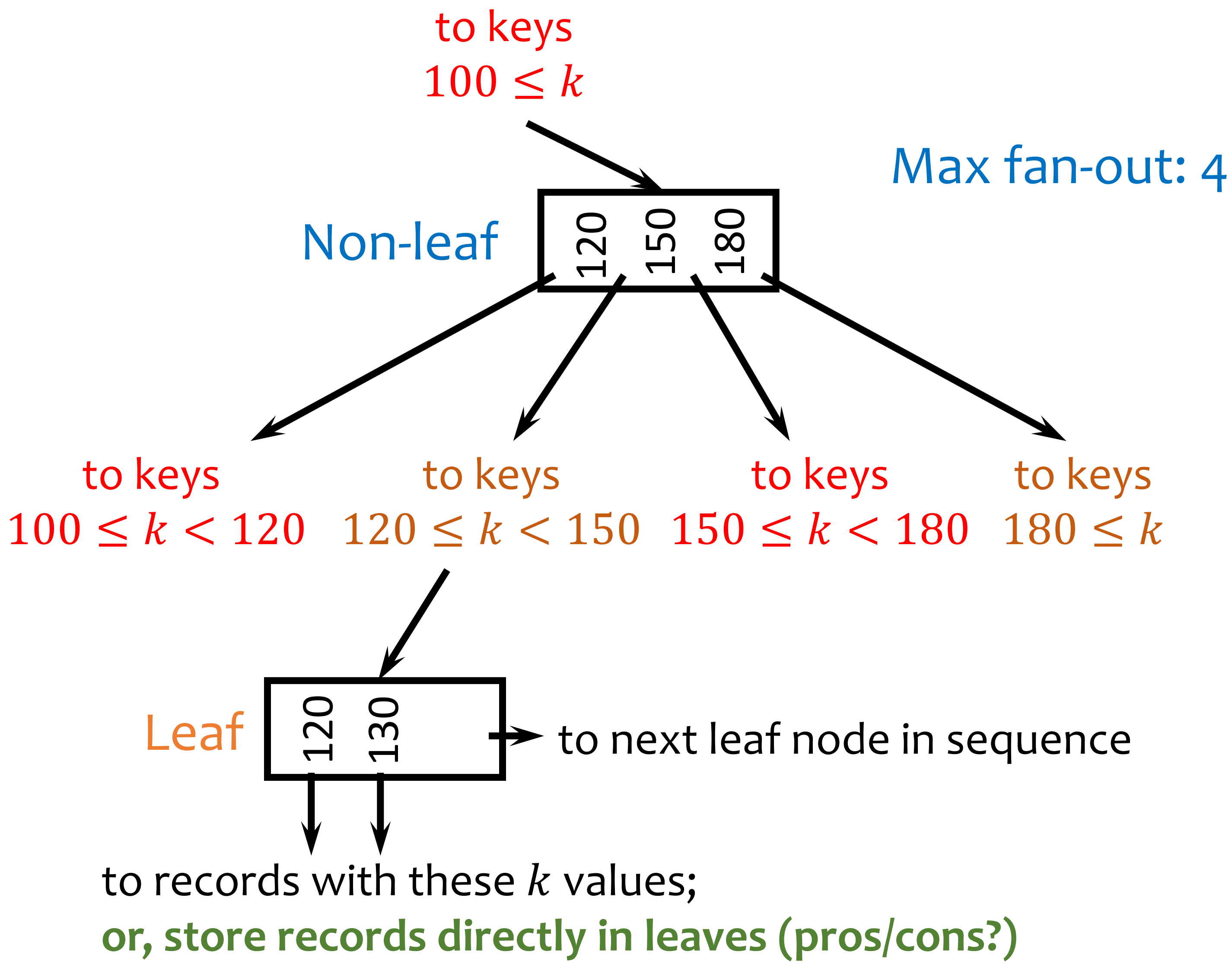
B+-tree: Closer Look

Max fan-out: 4

- 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



- Questions

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

vs.

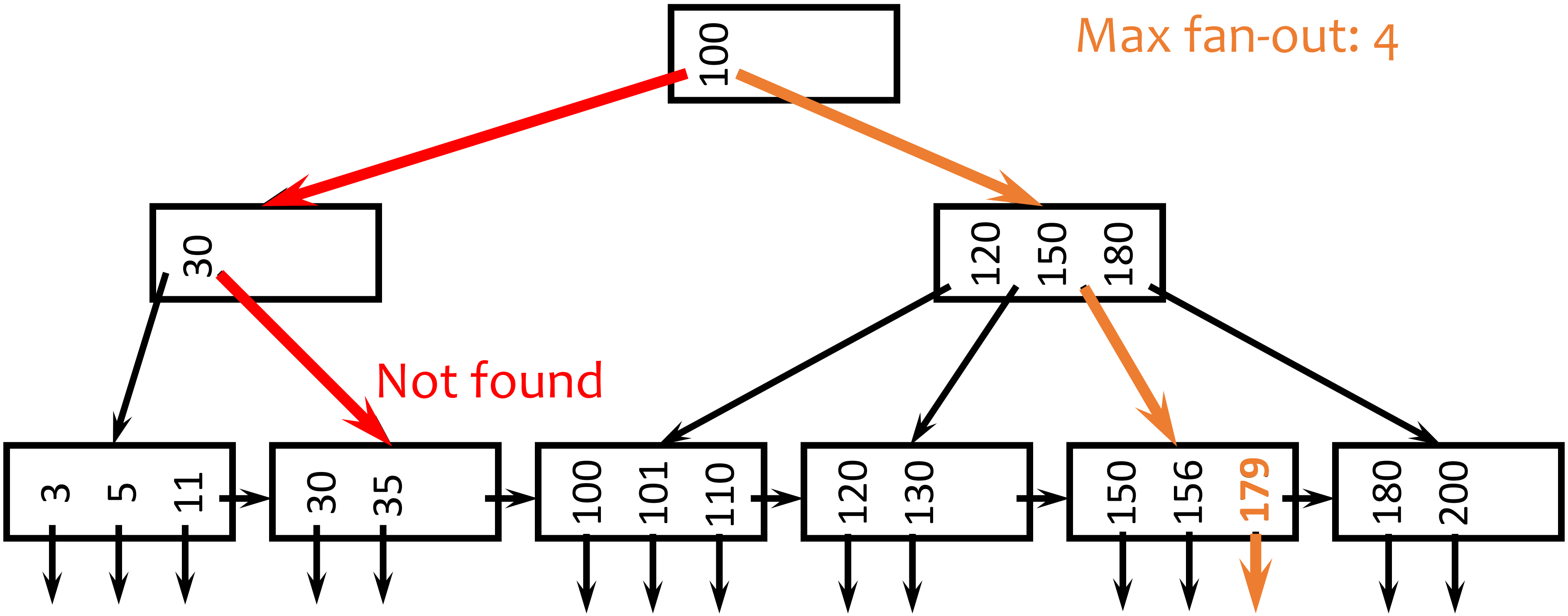
- Why do we use B^+-tree as database index instead of B-trees (next slide)?
  - What are the differences/pros/cons of B-trees vs. B^+-tree as index?

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!

Lookups

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



Practice Problems

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

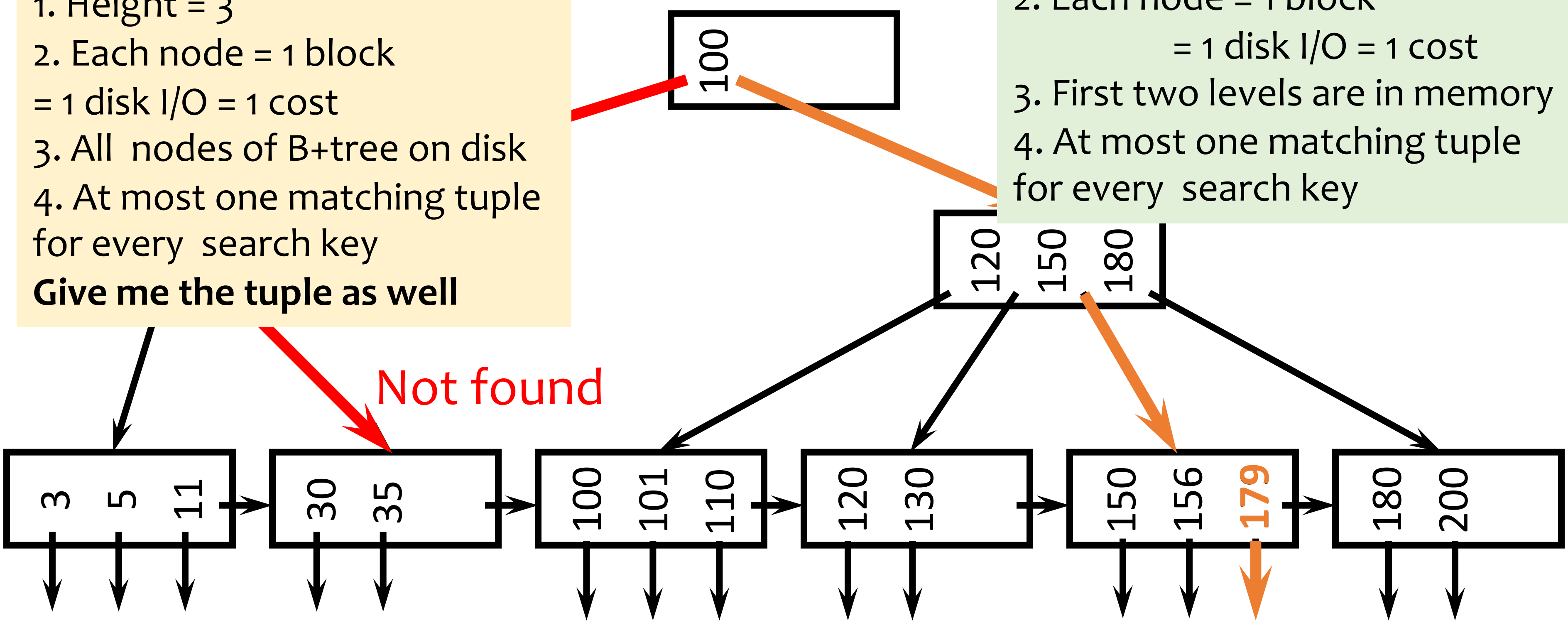
Asssumptions: Cost = 3
1. Height = 3
2. Each node = 1 block
   = 1 disk I/O = 1 cost
3. All nodes of B+tree on disk
4. At most one matching tuple for every search key

Just find if such a tuple exist

Asssumptions: Cost = 1
1. Height = 3
2. Each node = 1 block
   = 1 disk I/O = 1 cost
3. First two levels are in memory
4. At most one matching tuple for every search key

Asssumptions: Cost <= 3 +1 = 4
1. Height = 3
2. Each node = 1 block
   = 1 disk I/O = 1 cost
3. All nodes of B+tree on disk
4. At most one matching tuple for every search key

Give me the tuple as well

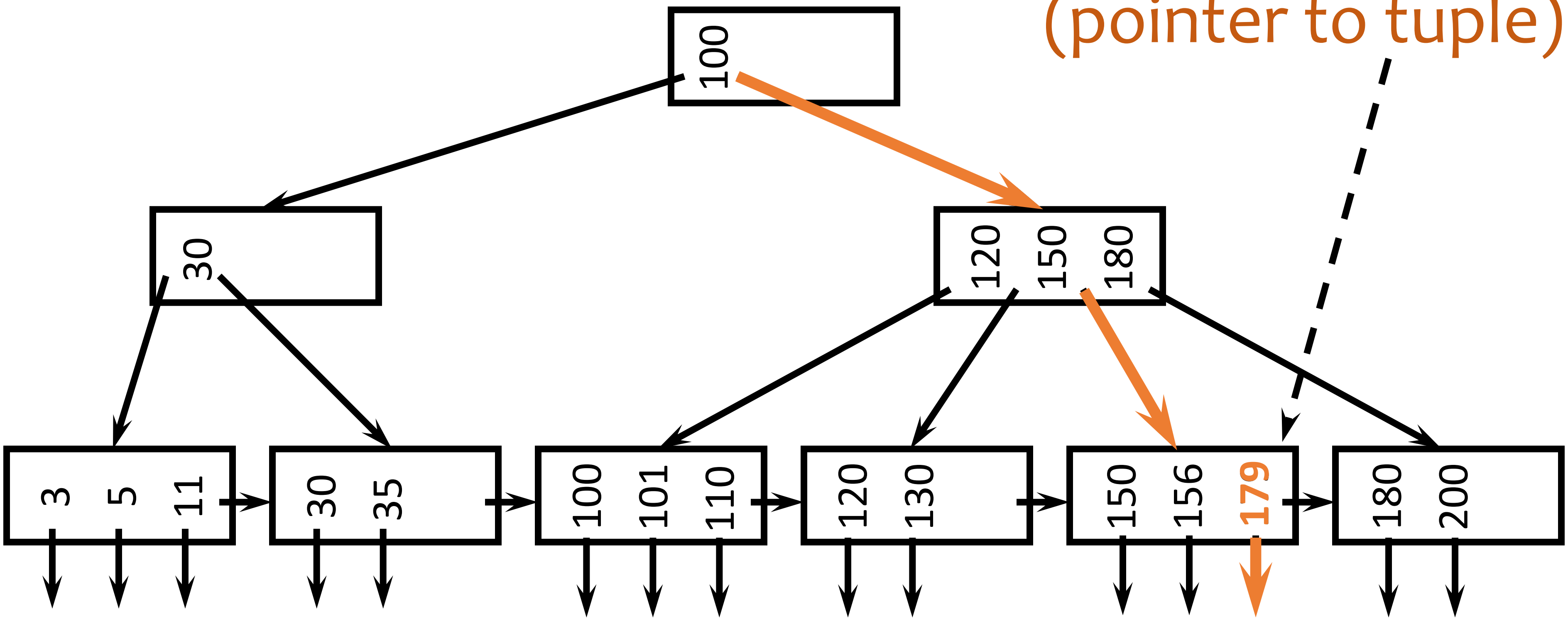


Recap: Search key and Data entry

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

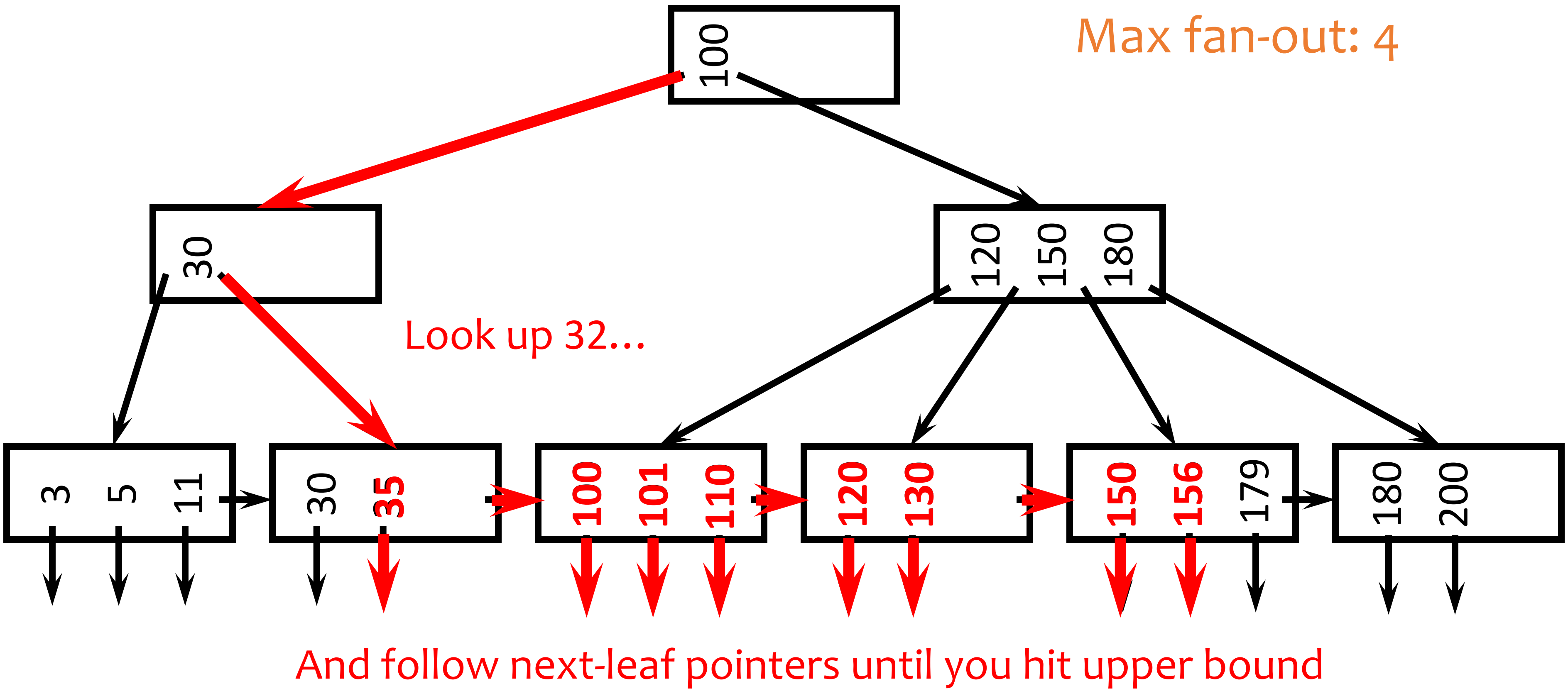
Search key (value)

Data Entry (pointer to tuple)



Range query

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



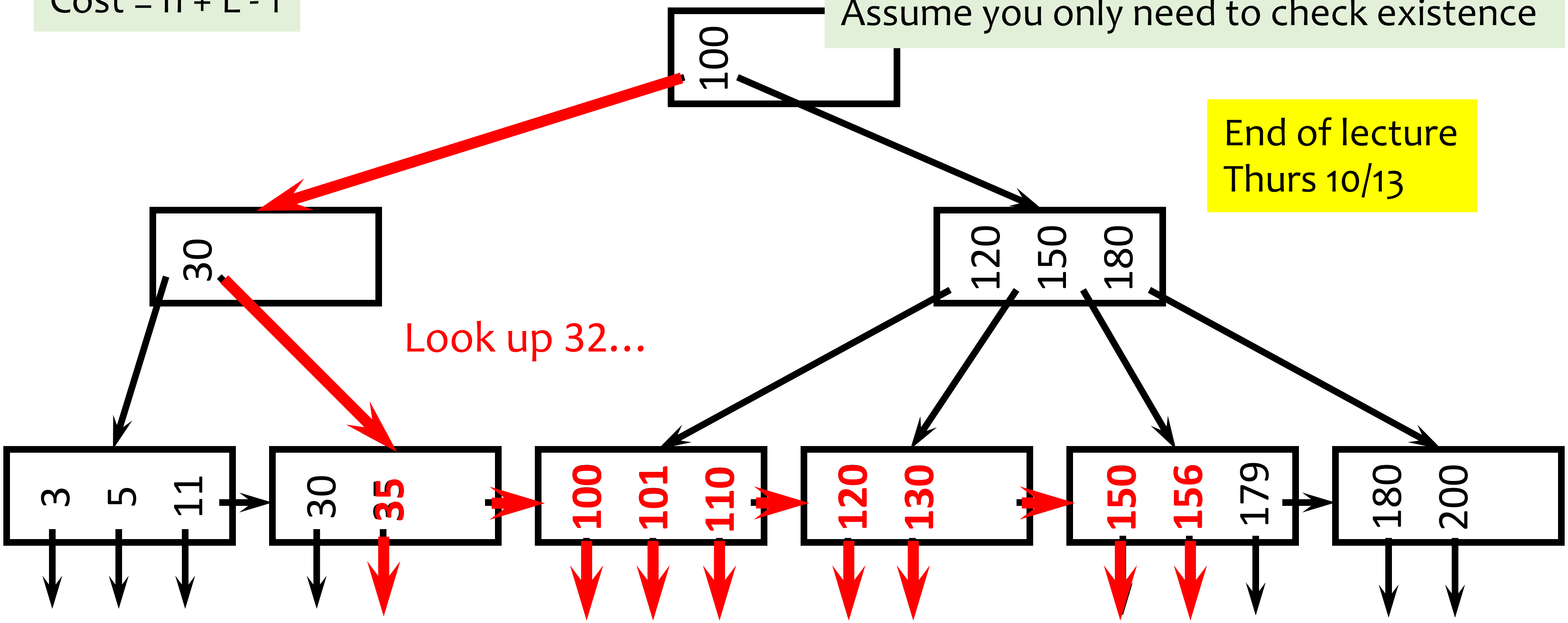
Practice Problem 2

Assume height = h
Assume there are L matching leaves
= matching data entries reside on L leaves
Assume one node = one block
= 1 cost = 1I/O
Everything on disk
Assume you only need to check existence

- SELECT * FROM *R* WHERE *k* >

Cost = h + L - 1

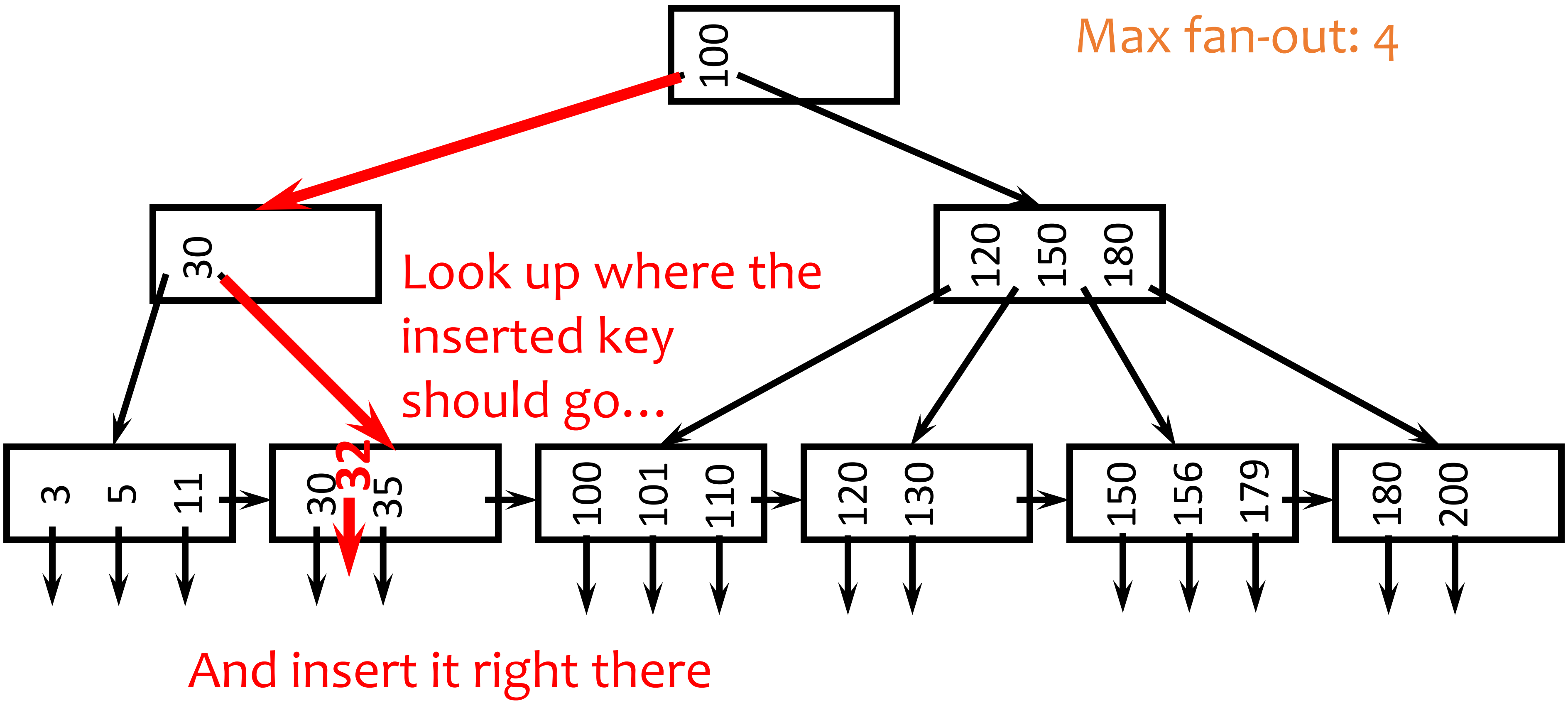
End of lecture
Thurs 10/13



And follow next-leaf pointers until you hit upper bound

Insertion

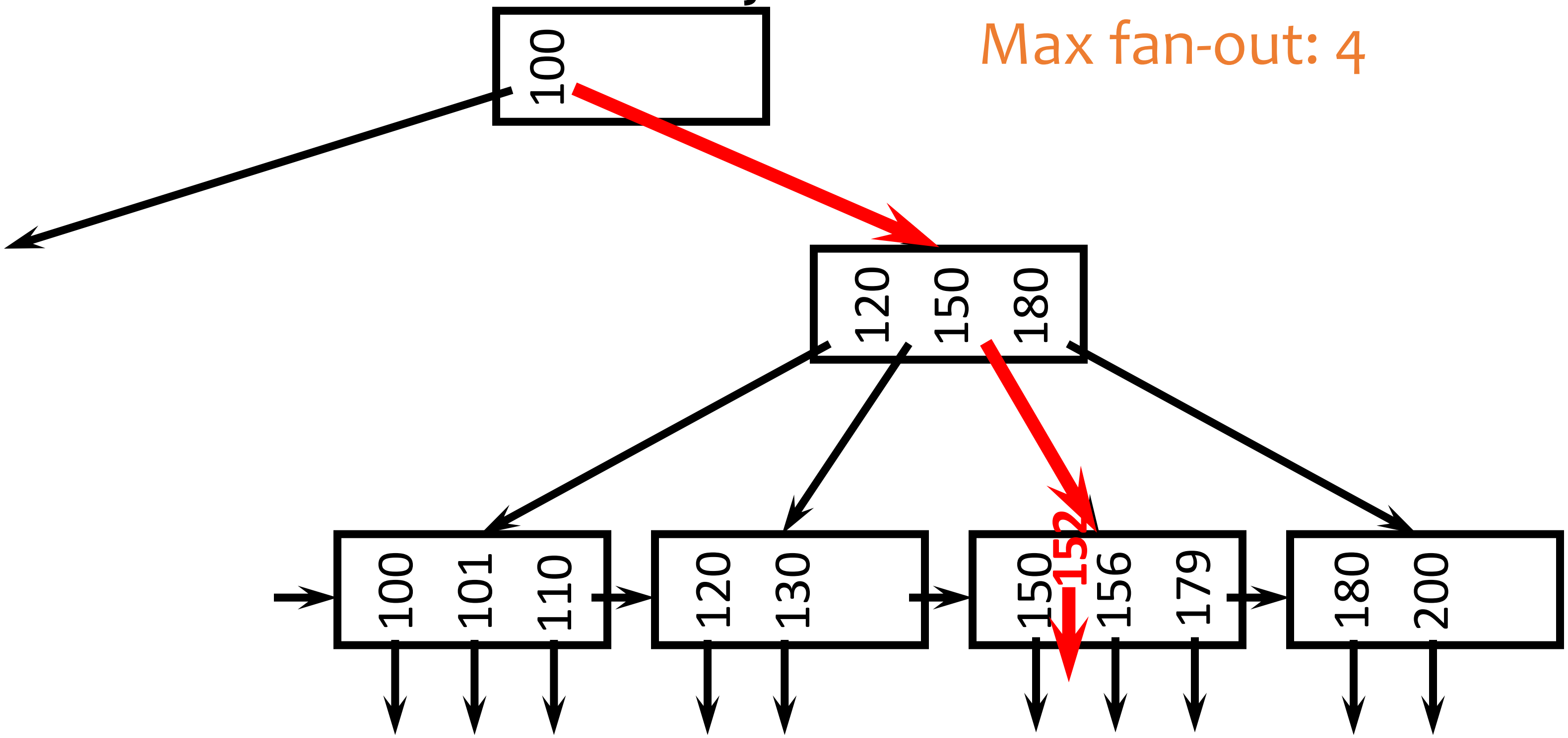
- Insert a record with search key value 32



Another insertion example

- Insert a record with search key value 152

Max fan-out: 4



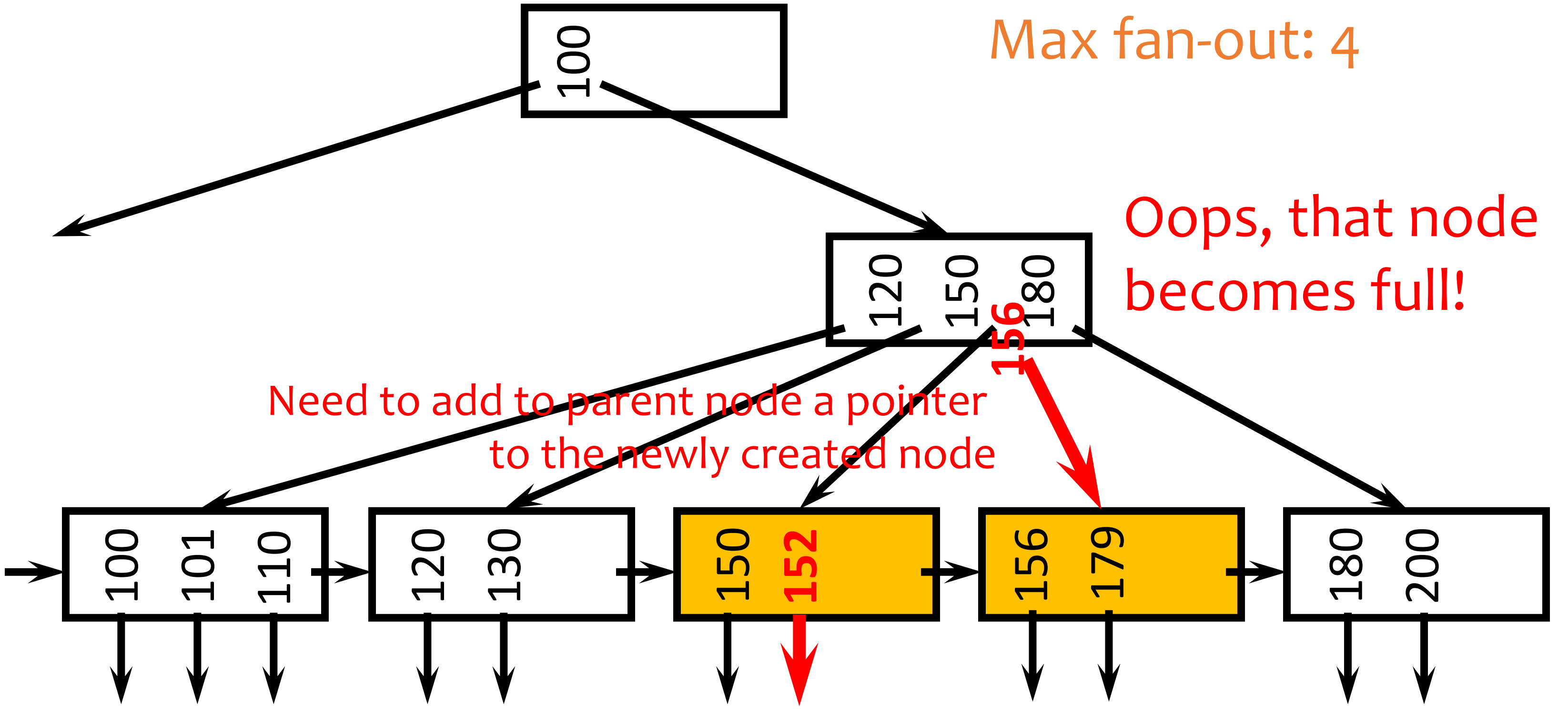
You could reorganize with a sibling here since they have space
e.g., (150, 152, 156) and (179, 180, 200)
The parent should be (120, 150, 179)

Oops, node is already full!

What are our options here?

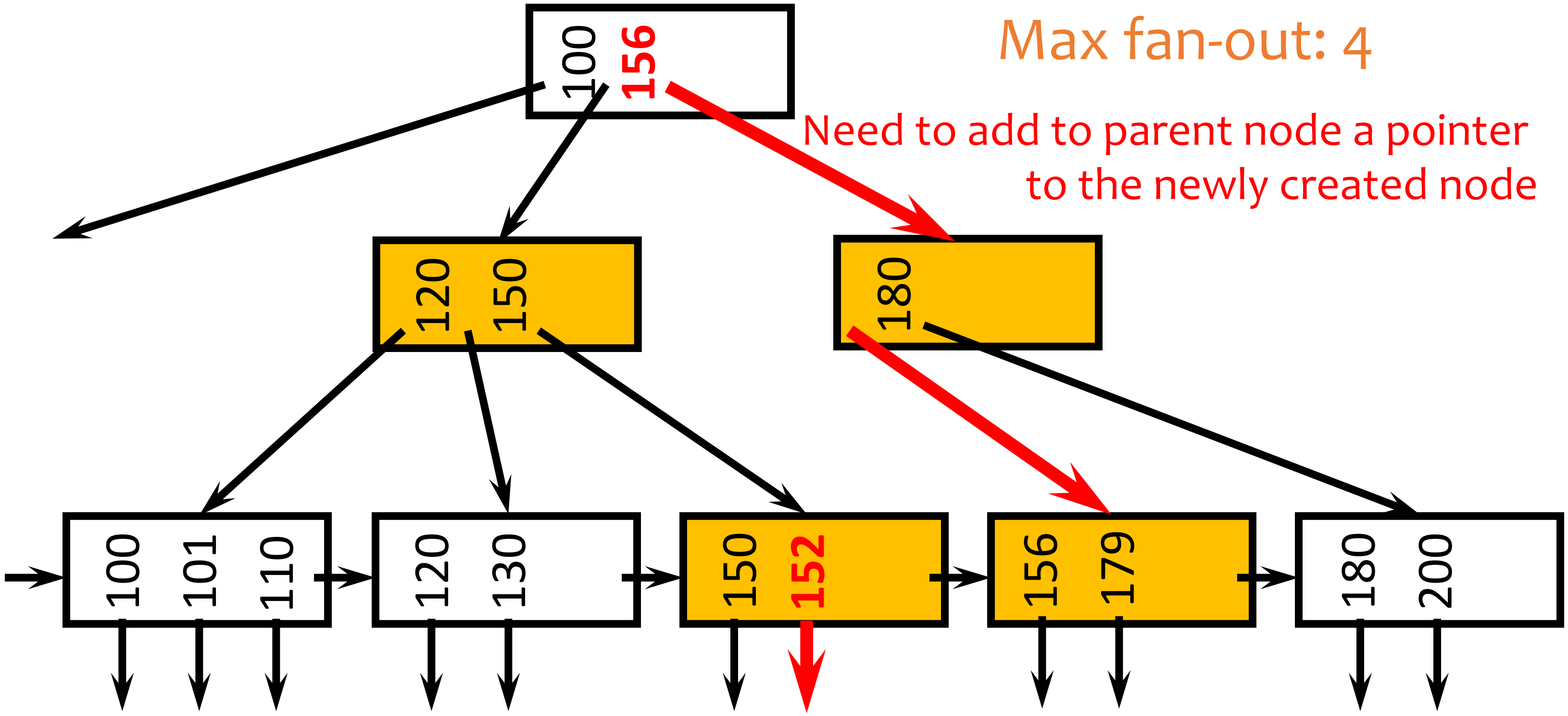
Node splitting

Note:
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

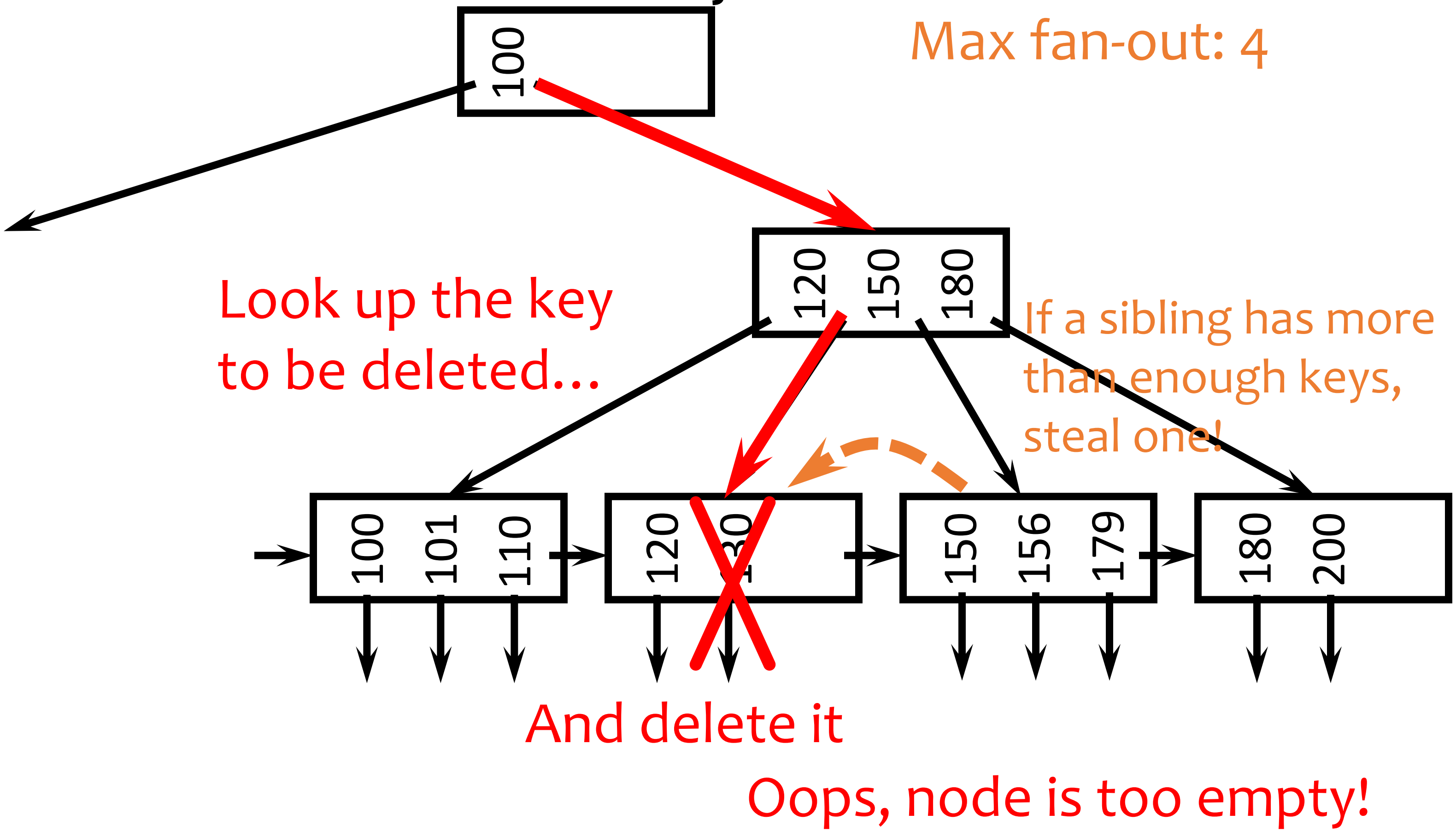
Note:
We "push up" while splitting non-leaves, insertion ONLY at the parent node (from the middle)
This is so that we do not have a dangling pointer at non-leaves



- 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

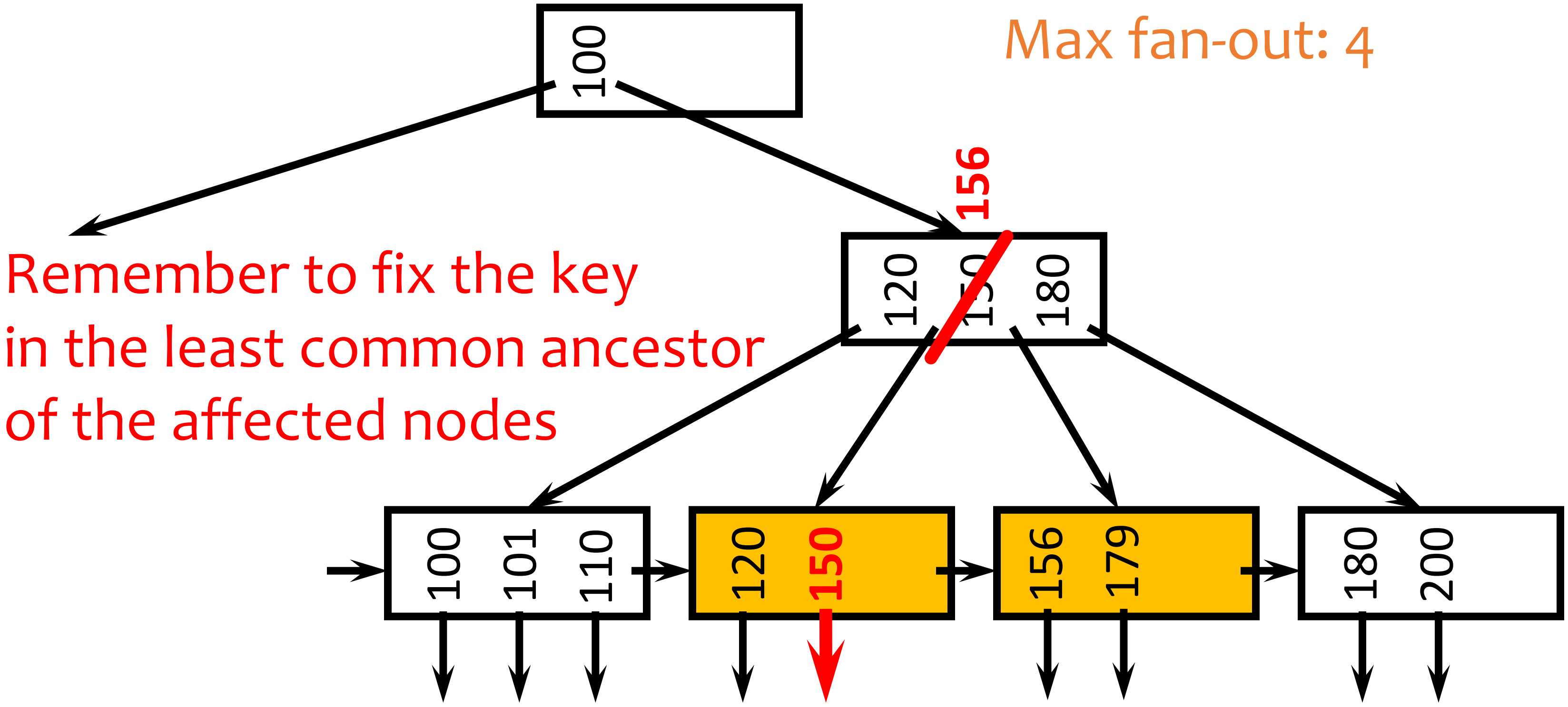
Deletion

- Delete a record with search key value 130



Stealing from a sibling

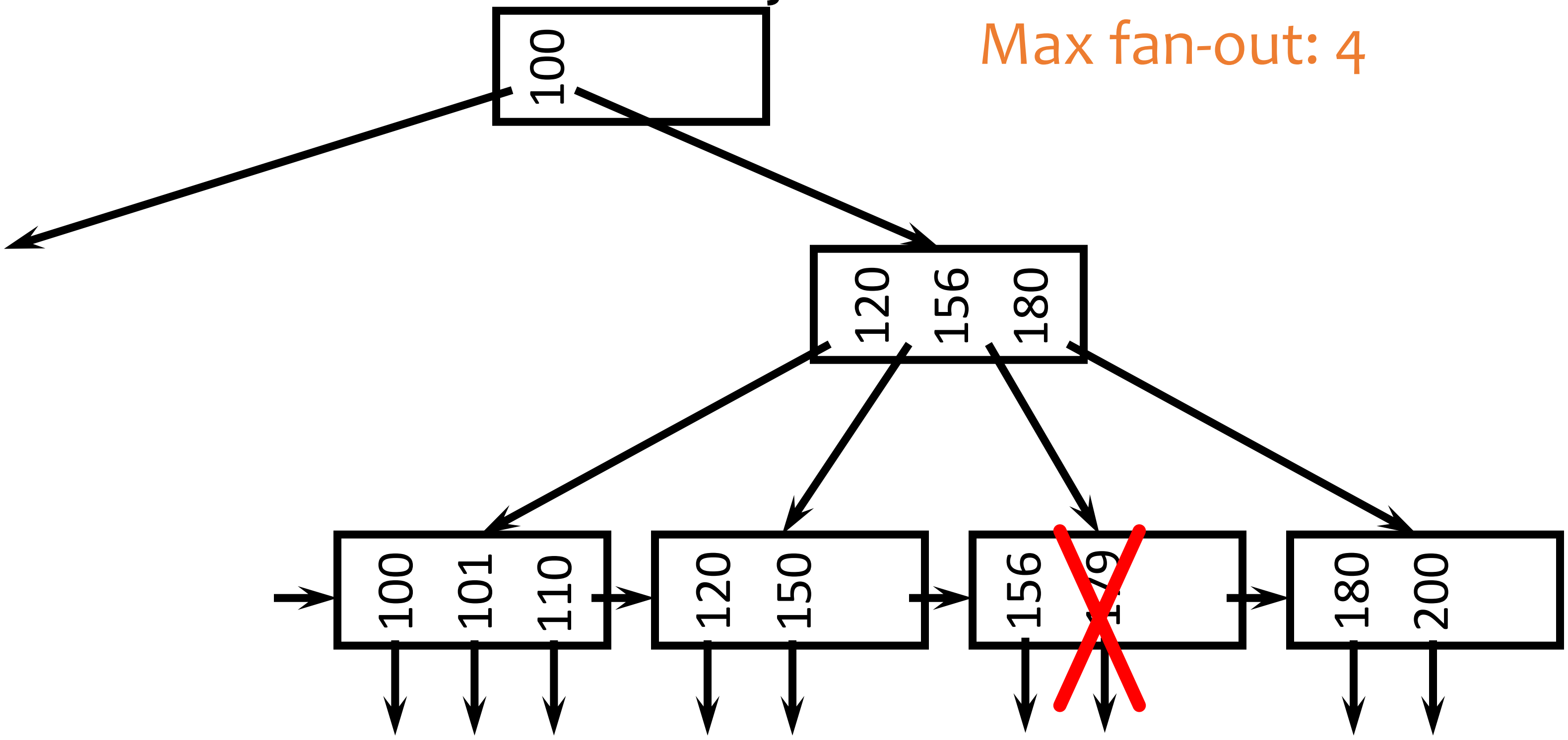
Max fan-out: 4



Another deletion example

- Delete a record with search key value 179

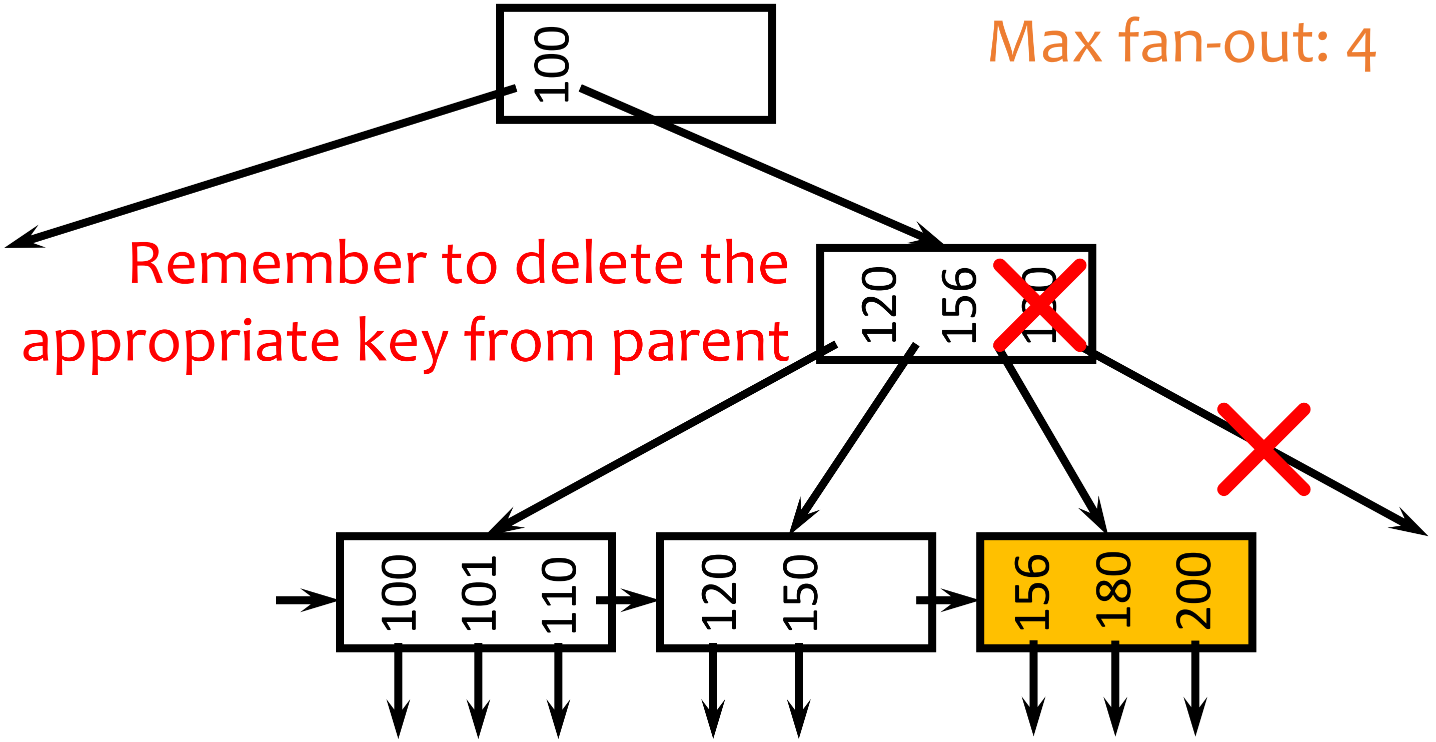
Max fan-out: 4



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 $\log_{\mathrm{fanout}} N$, 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

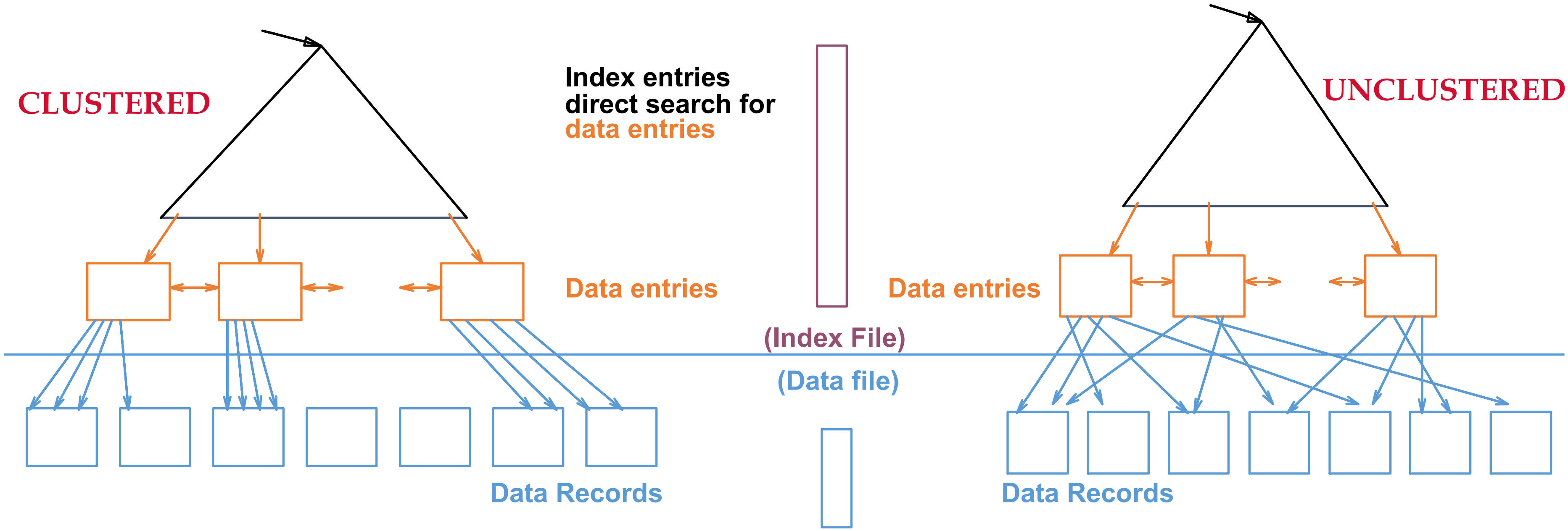
B^+-tree in practice

- 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!

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 = 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))
  - +1 for page boundary

The Halloween Problem

- Story from the early days of System R…

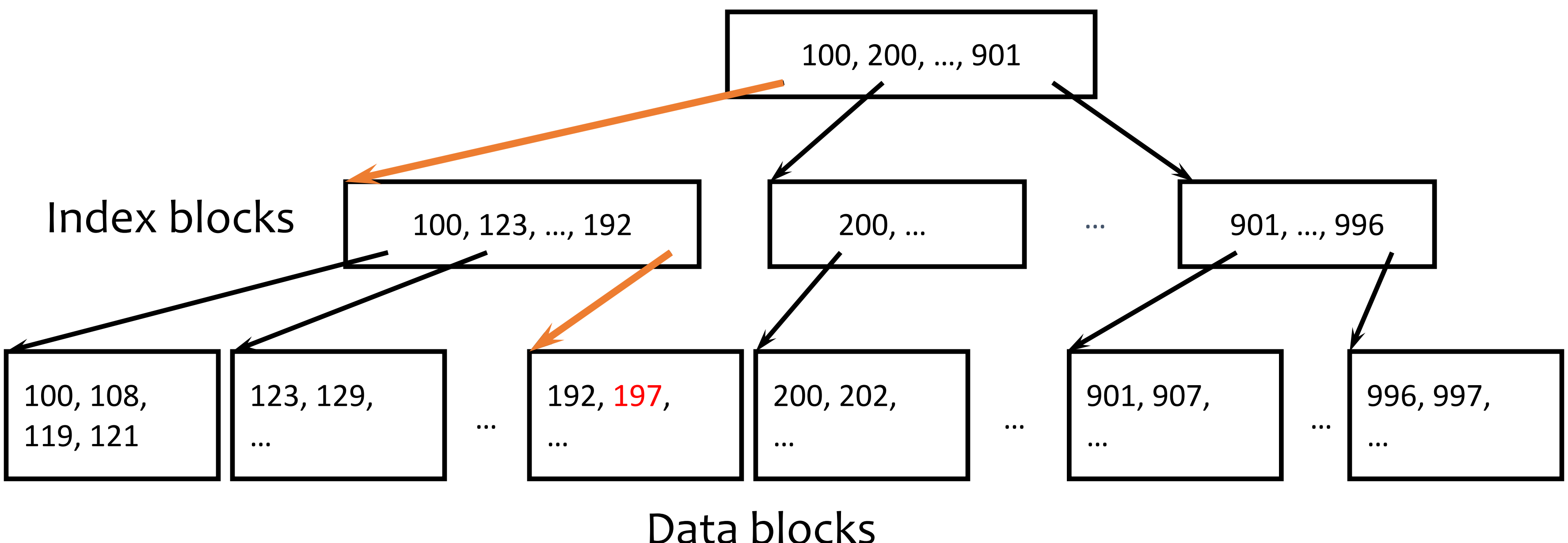
```
UPDATE Payroll
SET salary = salary * 1.1
WHERE salary <= 25000;
```

  - 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

ISAM

- ☞ ISAM (Index Sequential Access Method), static version of B+-tree
- ☞ Updates are handled by (long) overflow chains
- Overflow chains and empty data blocks degrade performance
  - Worst case: most records go into one long chain, so lookups require scanning all data!



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.

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

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))
- Index on prefixes: There can be "composite" hash or tree index on a set of attributes.
  - E.g., a tree-index on composite attributes (A, B) may have values as data entries (2, 1), (2, 2) (3, 1), (3, 5), (3, 7), (4, 1), (4, 5), …
  - Like "lexicographic order" – when same value of A, sort by B
- **(-) Hash index** 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 index on only (age) works, hash index on (age, name) does not work
- (+) 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?

End of lecture on 10/18

Trade-offs for Indexes

- Should we use as many indexes as possible?
- Indexes can make
  - queries go faster
  - updates slower
- Require disk space, too