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

Take-away from last lecture?

disk access — expensive.

read → 1 $I/O

write → 2
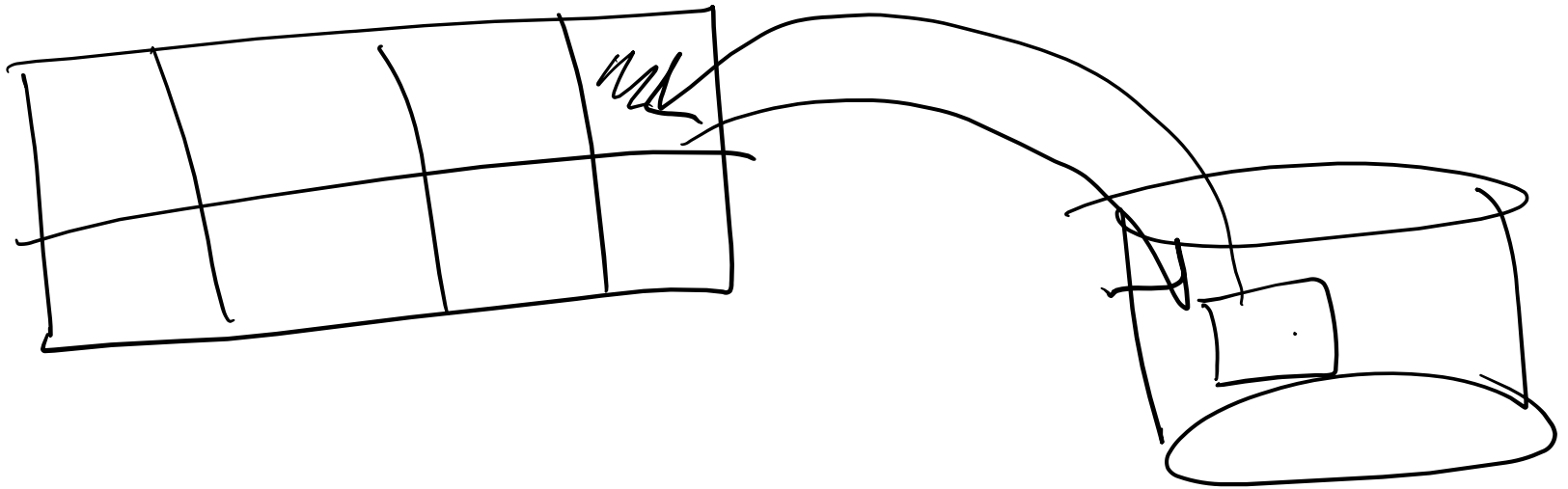
Today's lecture

- Index

- Dense vs. Sparse
- Clustered vs. unclustered
- Primary vs. secondary

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*;

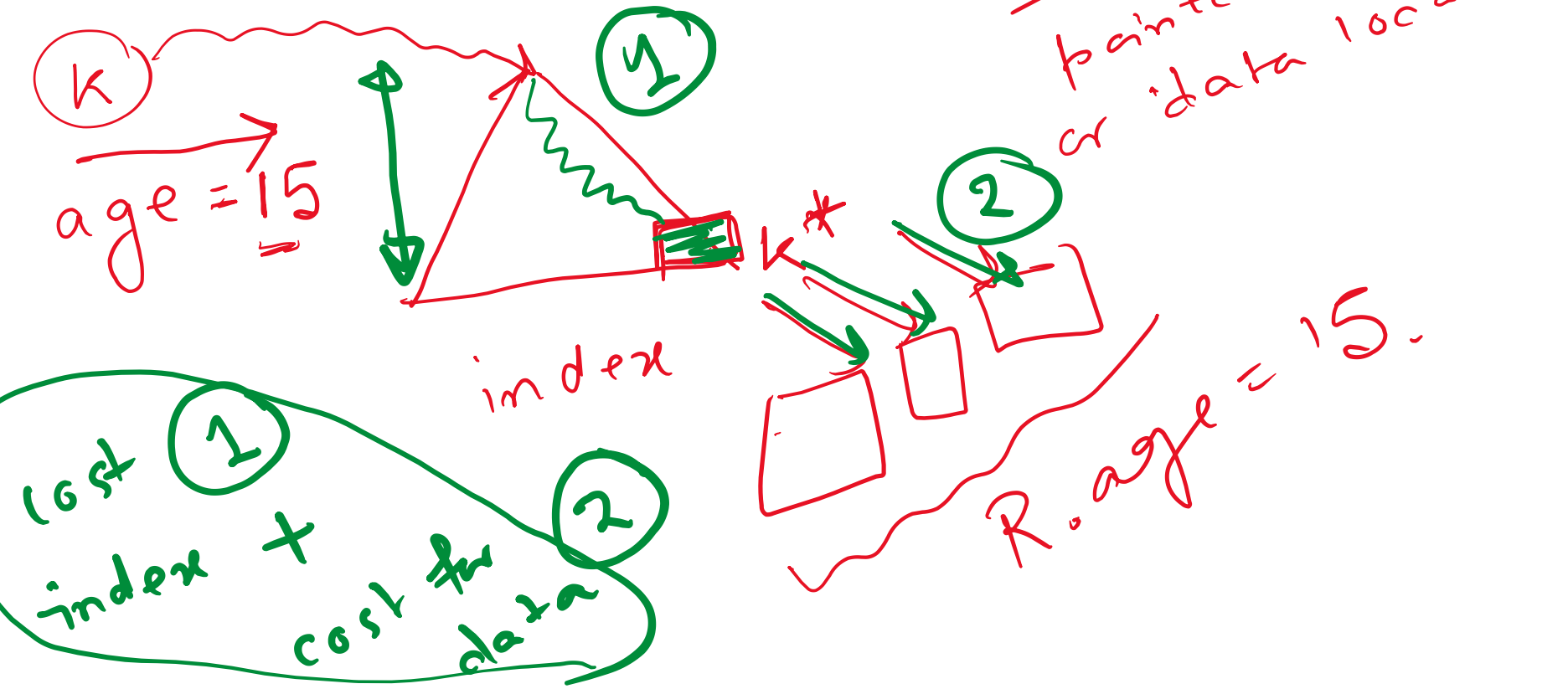 - Keyword search

Focus of this lecture

| database indexing | Search |

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?

R

id	age	name

pointer to tuples
or data locator.

k

age = 15

①

index

k*

②

R.age = 15.

cost ①
index + cost for data ②

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

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

Sparse index
on *uid*

123
456
857

Dense index
on *name*

Bart
Jessica
Lisa
Martin
Milhouse
Nelson
Ralph
Sherri
Terri
Windel

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
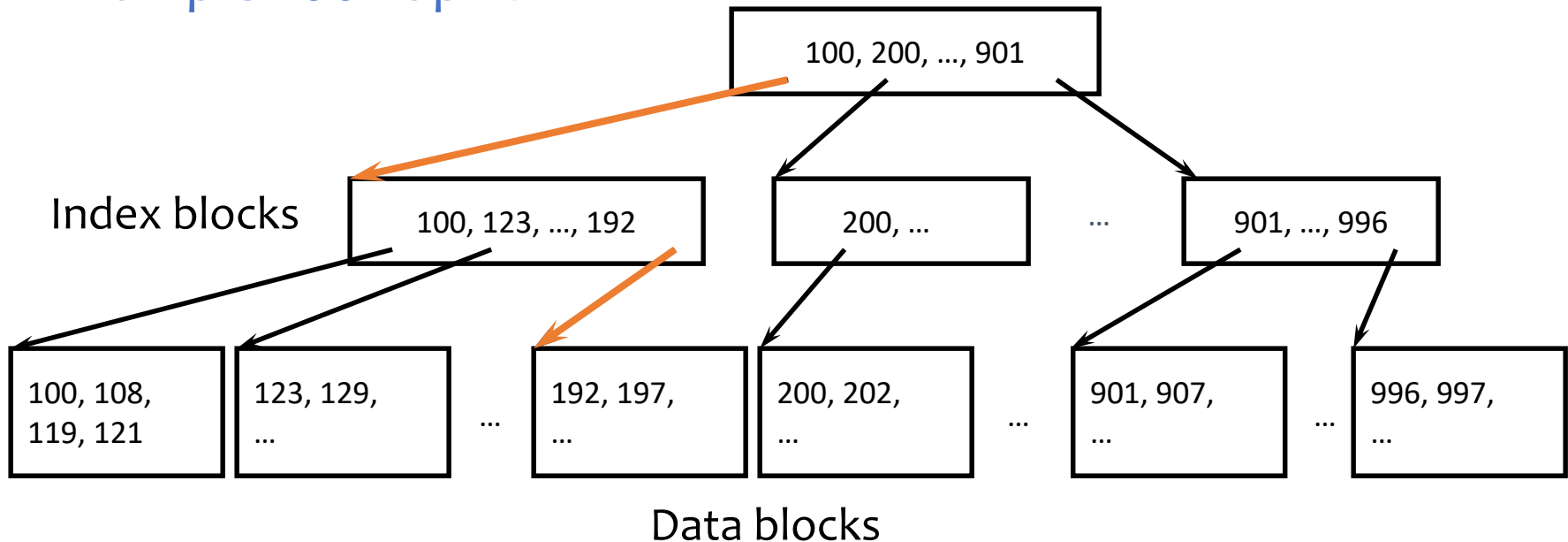 - Records are usually clustered by the primary key
  - Can be sparse
- Secondary index
  - Usually dense
- SQL
  - PRIMARY KEY declaration automatically creates a primary index, UNIQUE key automatically creates a secondary index
  - Additional secondary index can be created on non-key attribute(s):
    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
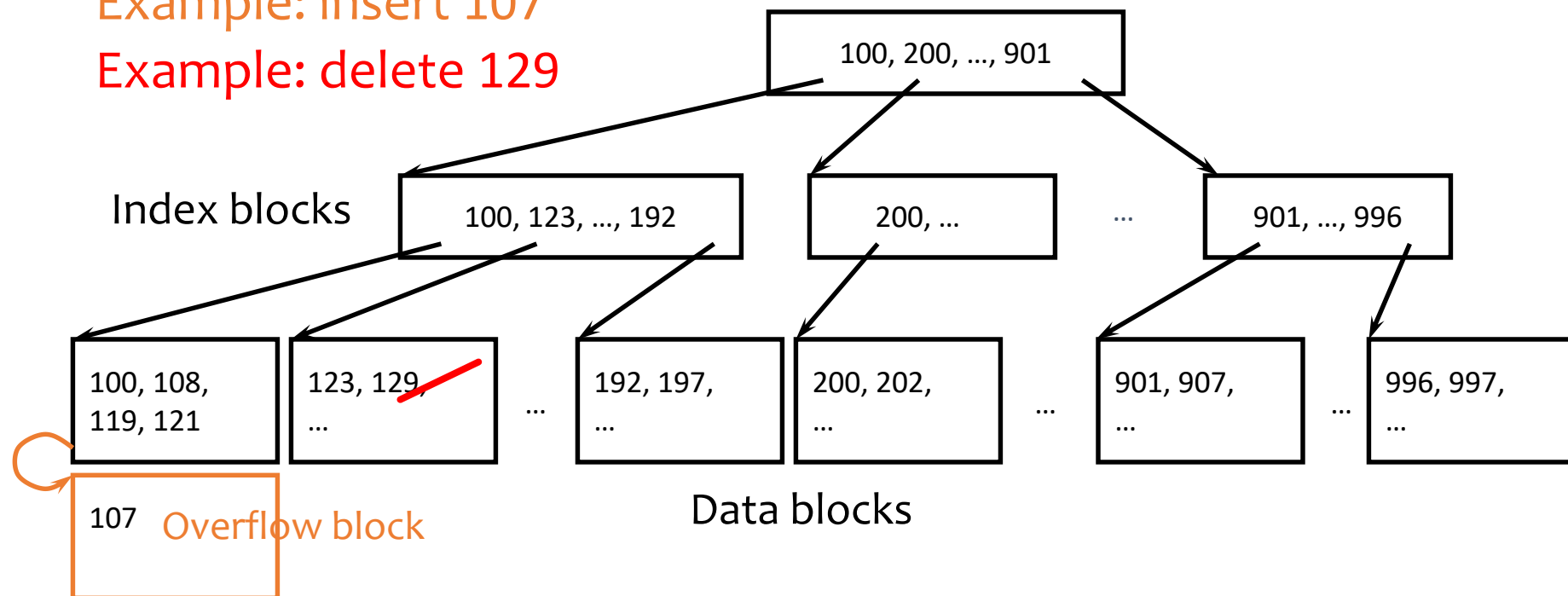
Example: look up 197

Index blocks

Data blocks

Updates with ISAM

Example: insert 107
Example: delete 129

Index blocks

| 100, 200, ..., 901 |

| 100, 123, ..., 192 | 200, ... | ... | 901, ..., 996 |

| 100, 108, 119, 121 | 123, 129 ... | ... | 192, 197, ... | 200, 202, ... | ... | 901, 907, ... | ... | 996, 997, ... |

107   Overflow block

Data blocks

- 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
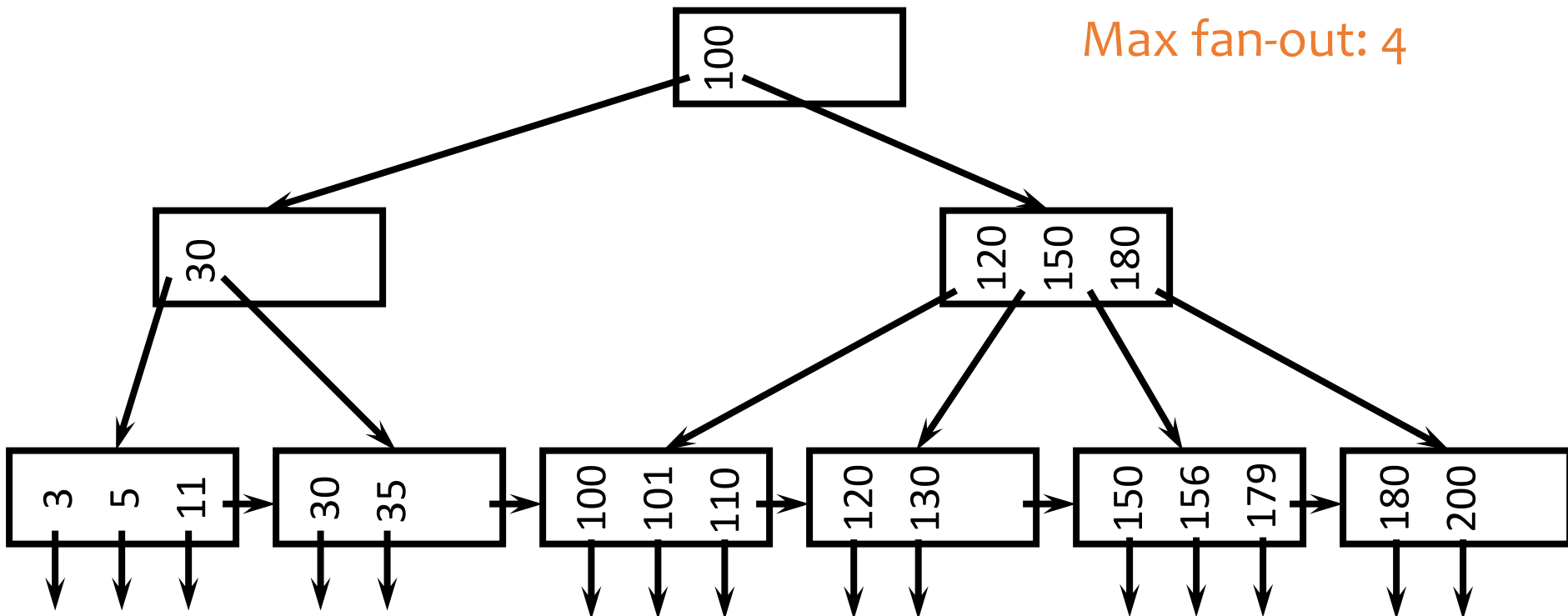
to records with these k values;
or, store records directly in leaves
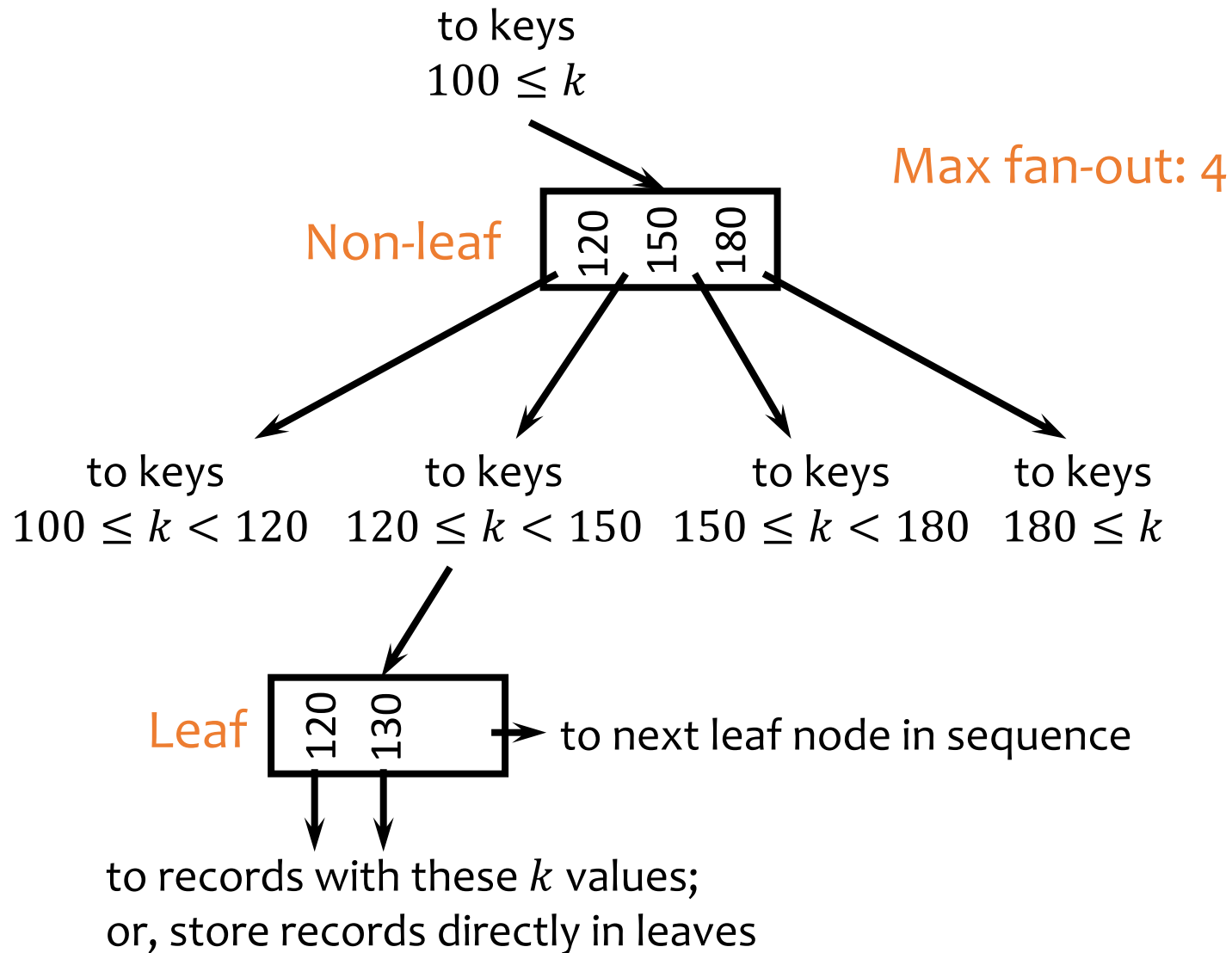
B⁺-tree balancing properties

- 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;

Max fan-out: 4

Not found
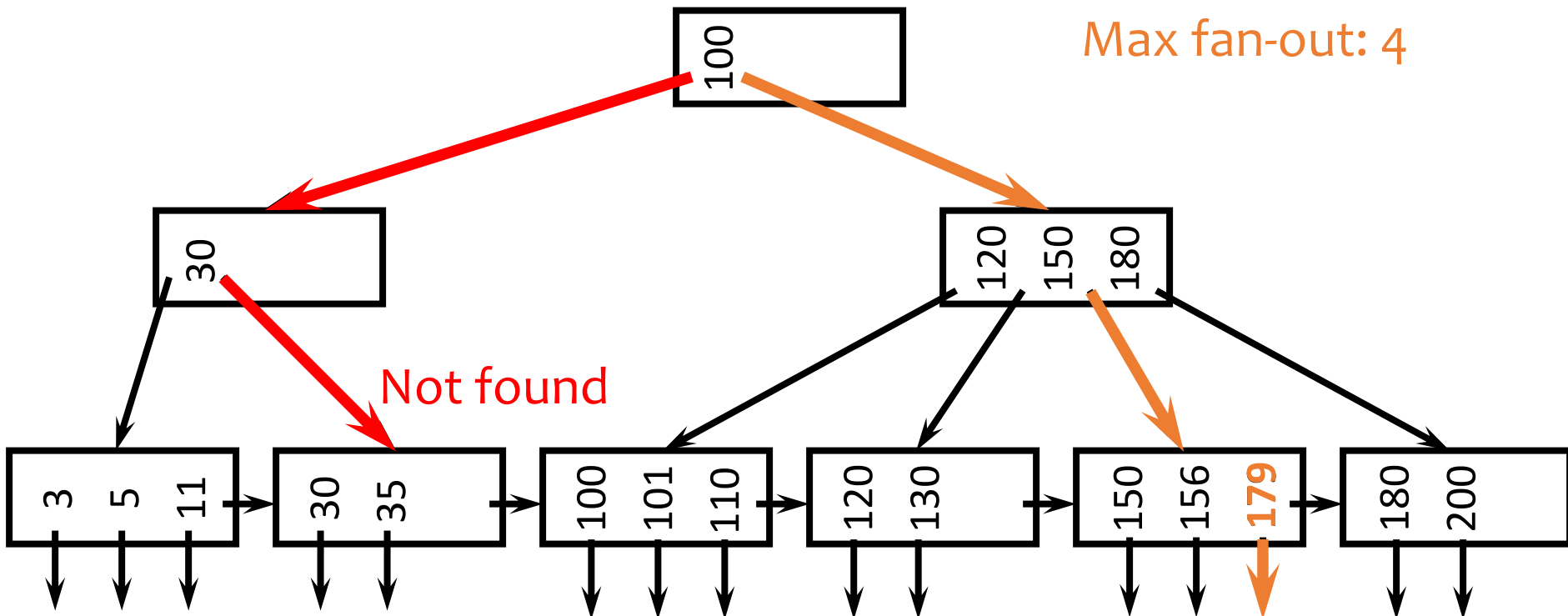
Range query

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

Max fan-out: 4

Look up 32...
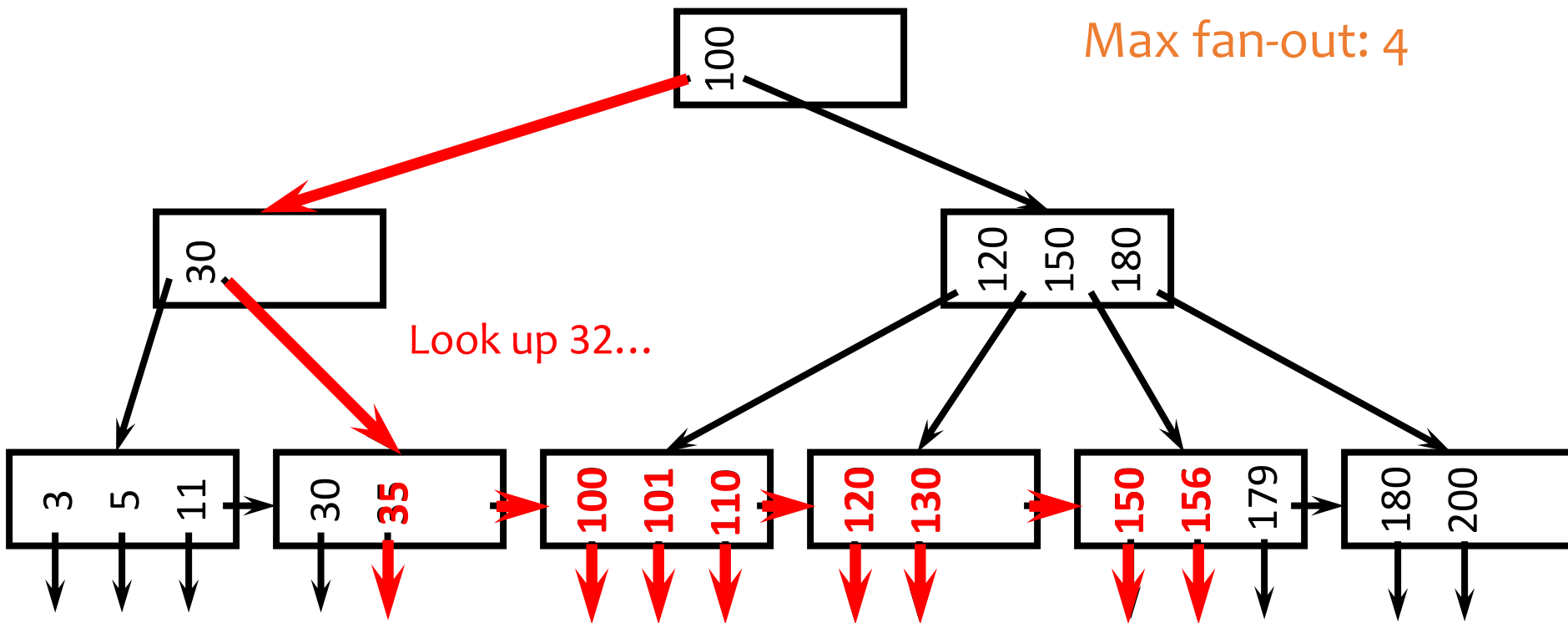
And follow next-leaf pointers until you hit upper bound

Tree structure:
- Root: 100
- Level 2: [30] and [120, 150, 180]
- Leaves: [3, 5, 11] → [30, 35] → [100, 101, 110] → [120, 130] → [150, 156, 179] → [180, 200]

Insertion

- Insert a record with search key value 32

Max fan-out: 4

Look up where the inserted key should go...

And insert it right there
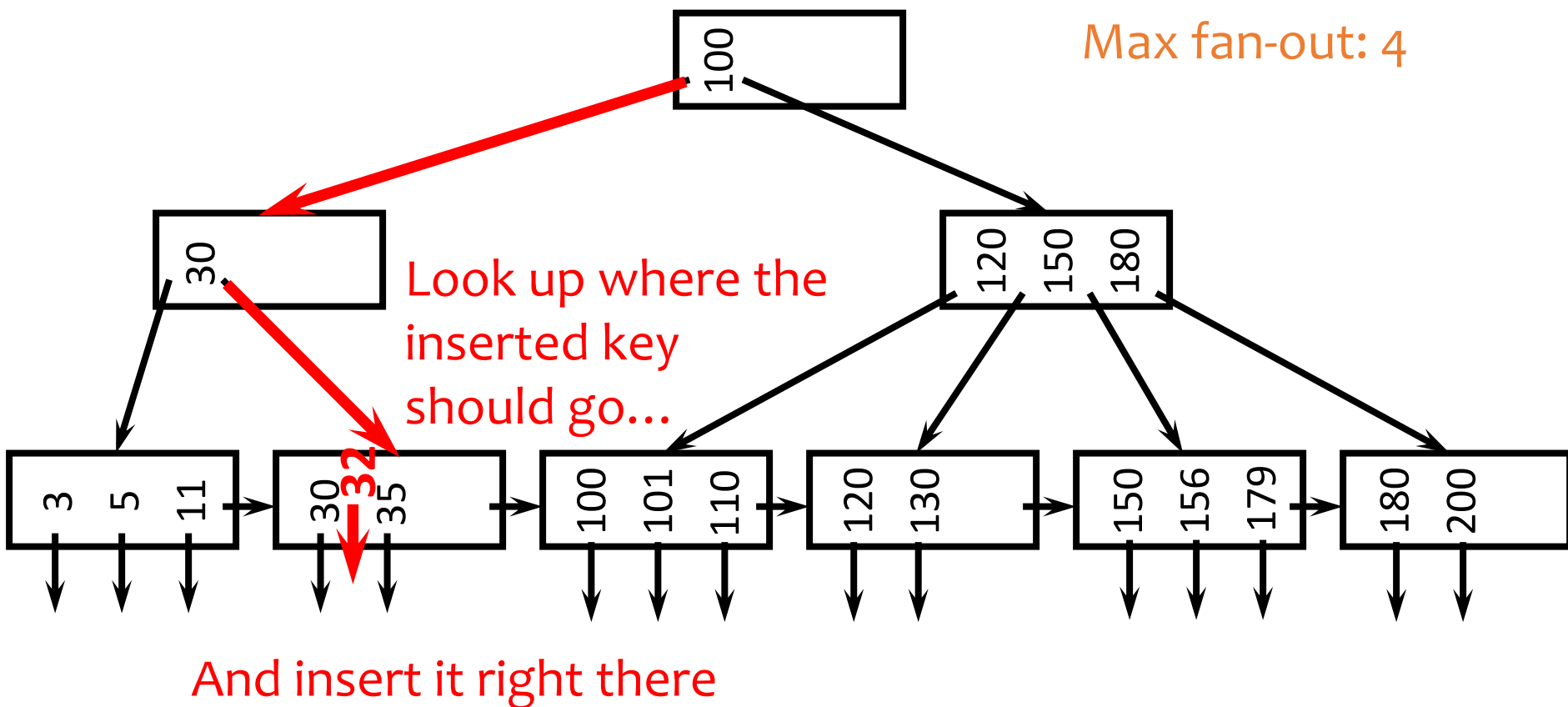
Another insertion example

- Insert a record with search key value 152

Max fan-out: 4
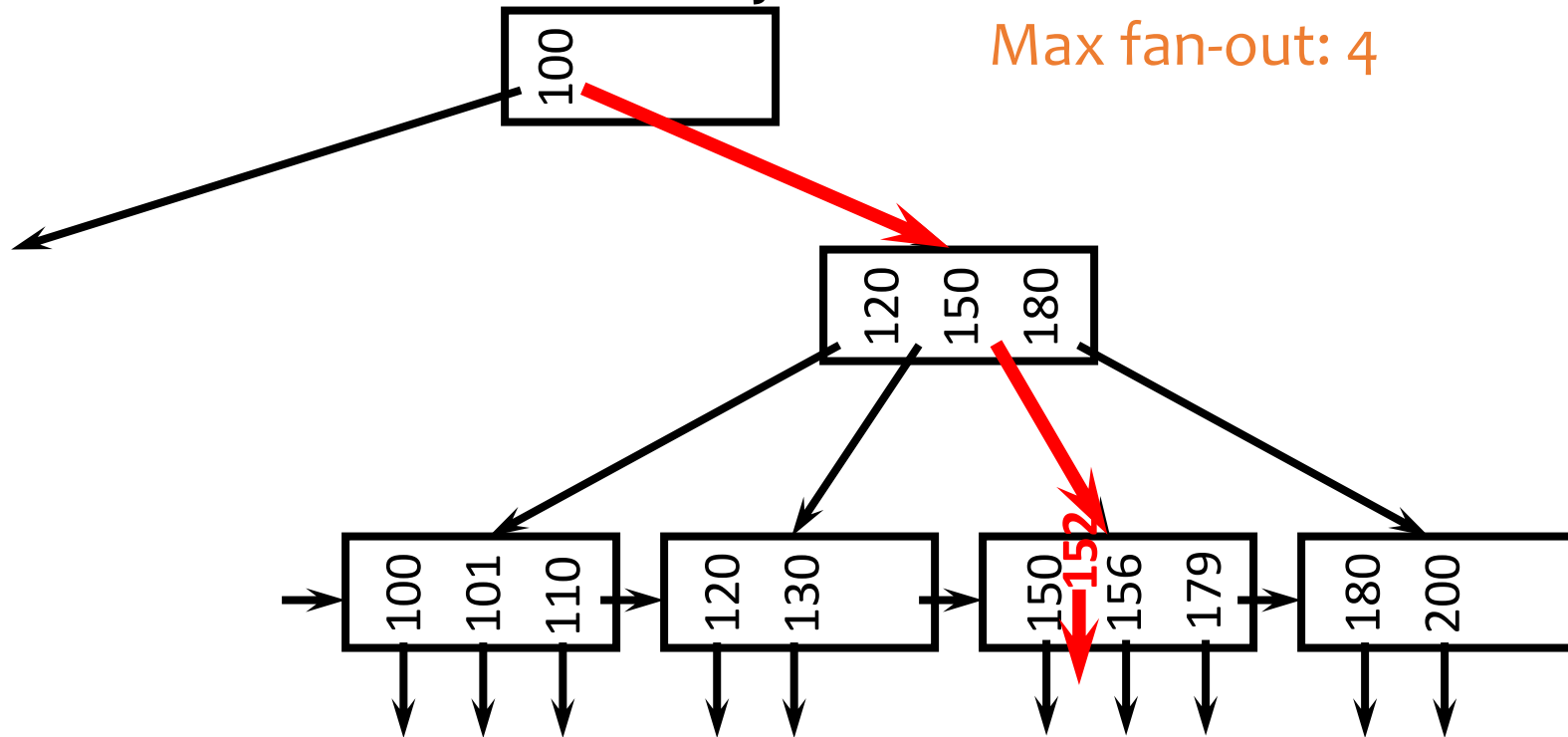
Oops, node is already full!

Node splitting

Max fan-out: 4

Oops, that node becomes full!

Need to add to parent node a pointer
to the newly created node

More node splitting

Max fan-out: 4

Need to add to parent node a pointer to the newly created node
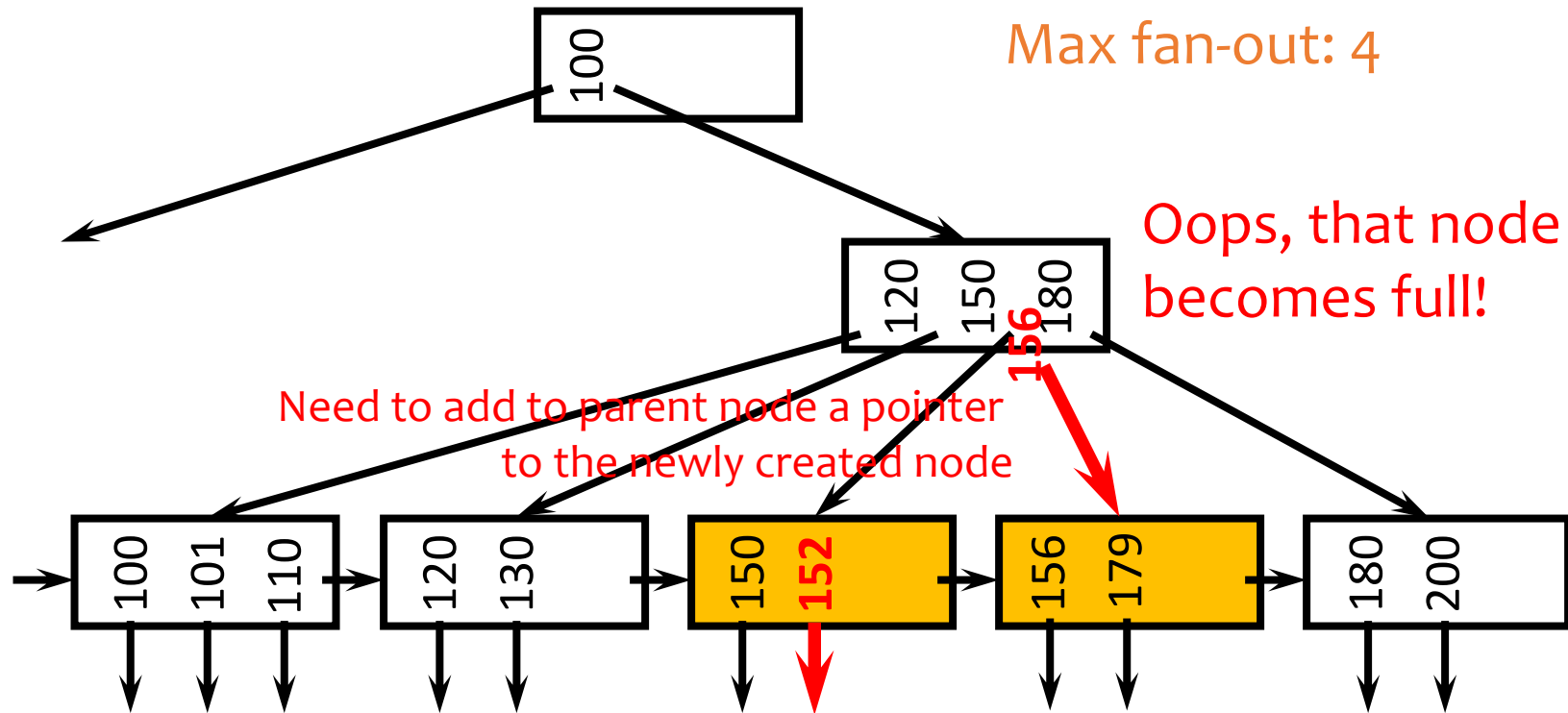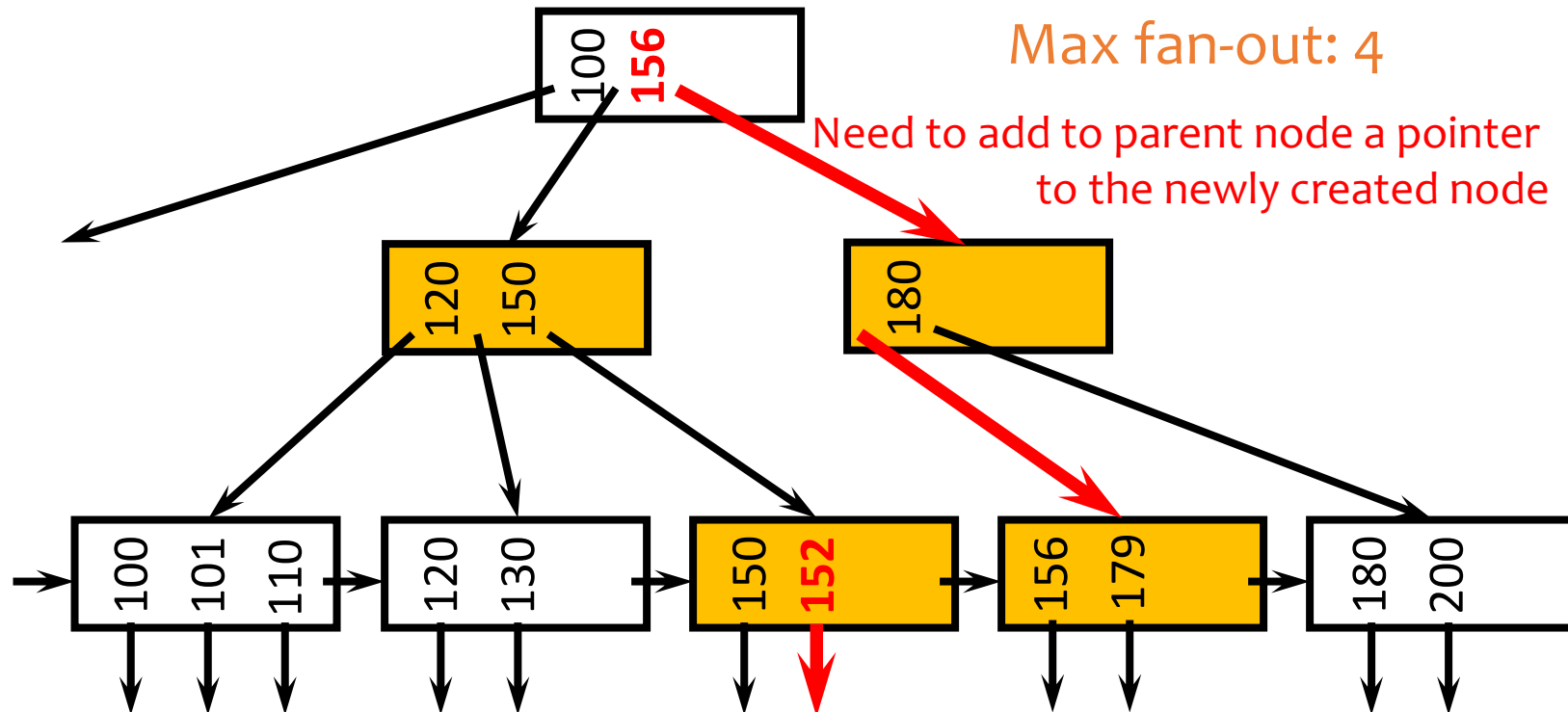
- In the worst case, node splitting can "propagate" all the way up to the root of the tree (not illustrated here)
  - Splitting the root introduces a new root of fan-out 2 and causes the tree to grow "up" by one level

Deletion

- Delete a record with search key value 130

Max fan-out: 4

Look up the key to be deleted…

If a sibling has more than enough keys, steal one!

100

120	150	180

100	101	110

120	130

150	156	179

180	200

And delete it

Oops, node is too empty!

Stealing from a sibling

Max fan-out: 4

100

156

120 150 180

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

100 101 110

120 150

156 179

180 200
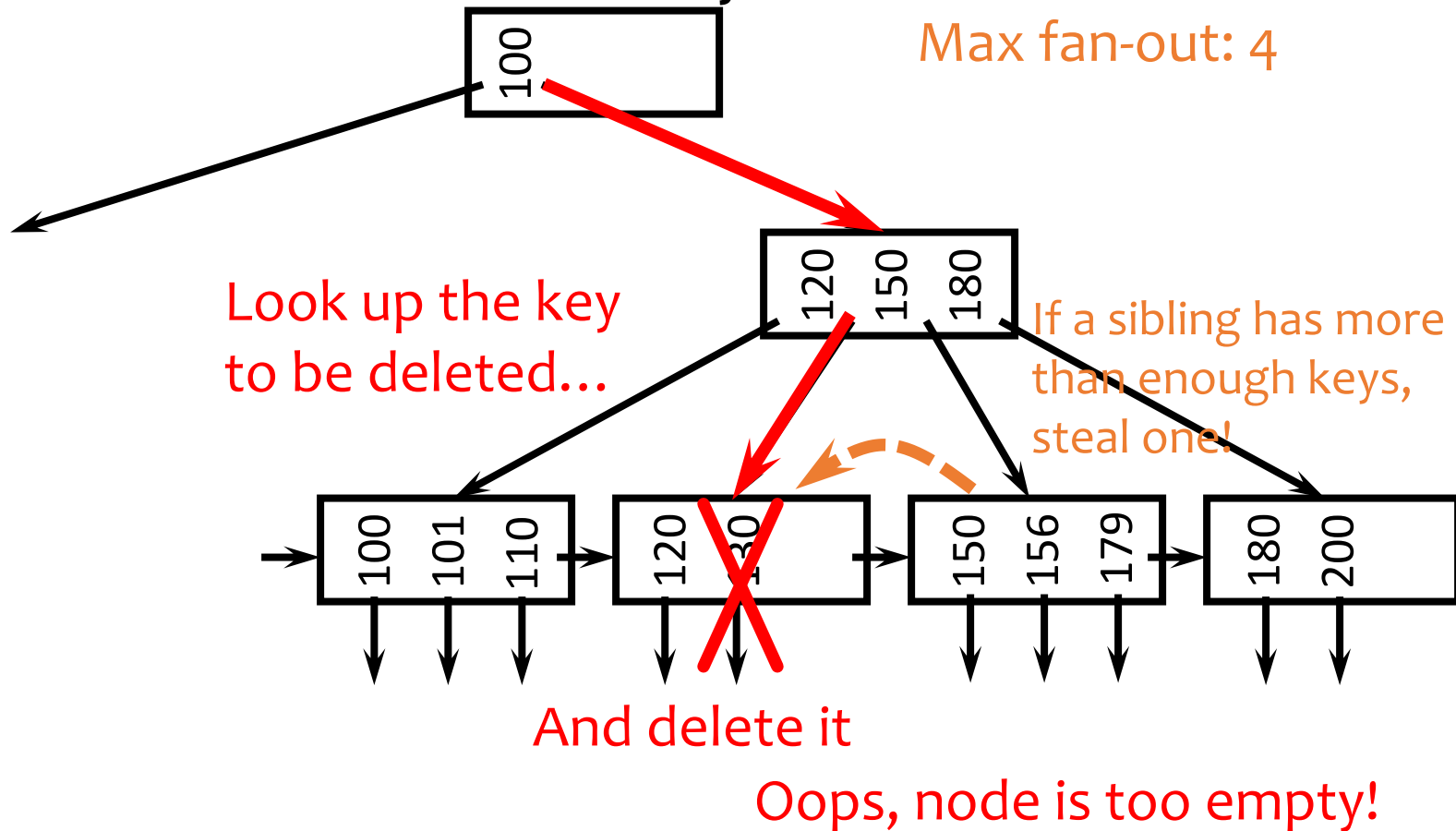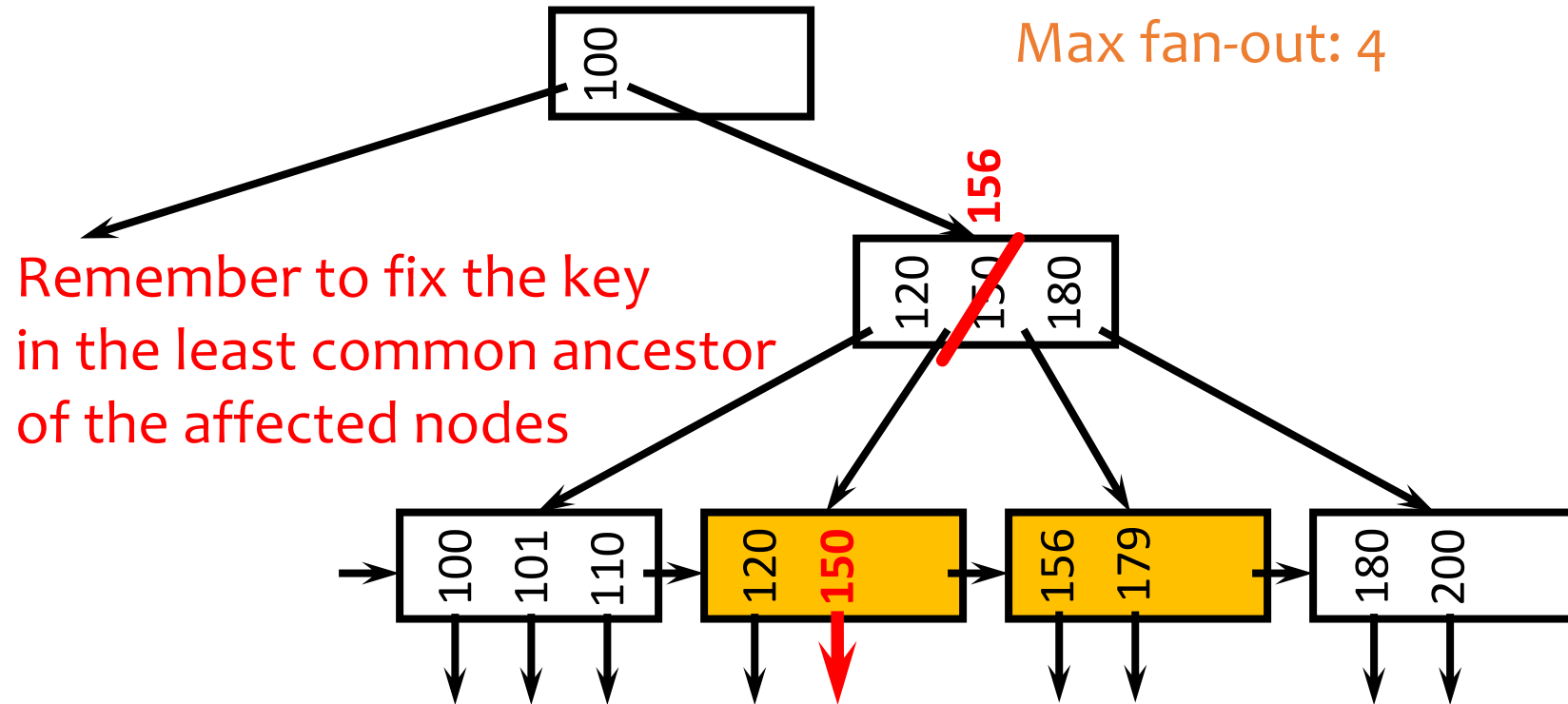
Another deletion example

- Delete a record with search key value 179

Max fan-out: 4
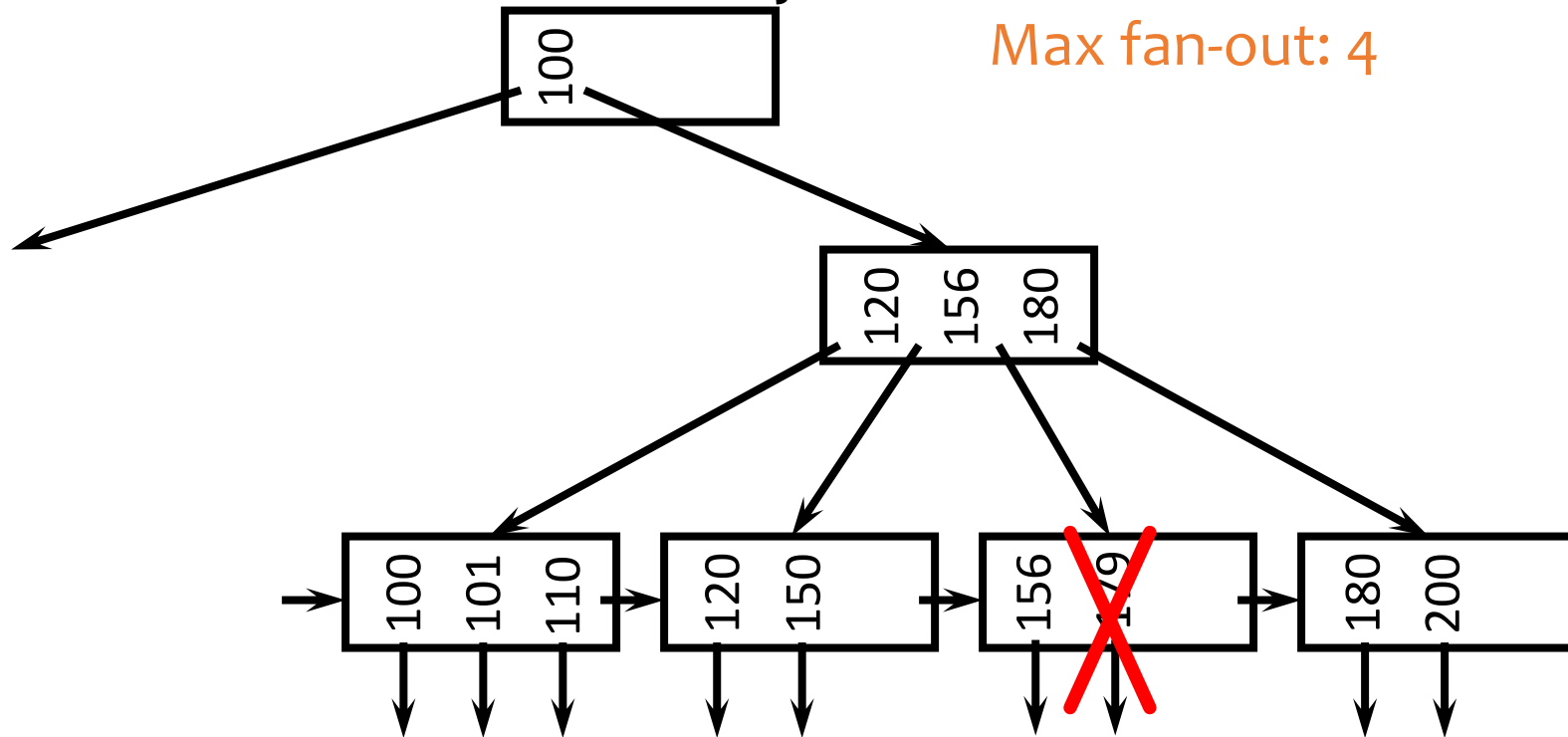
Cannot steal from siblings

Then coalesce (merge) with a sibling!

Coalescing

Max fan-out: 4

Remember to delete the appropriate key from parent
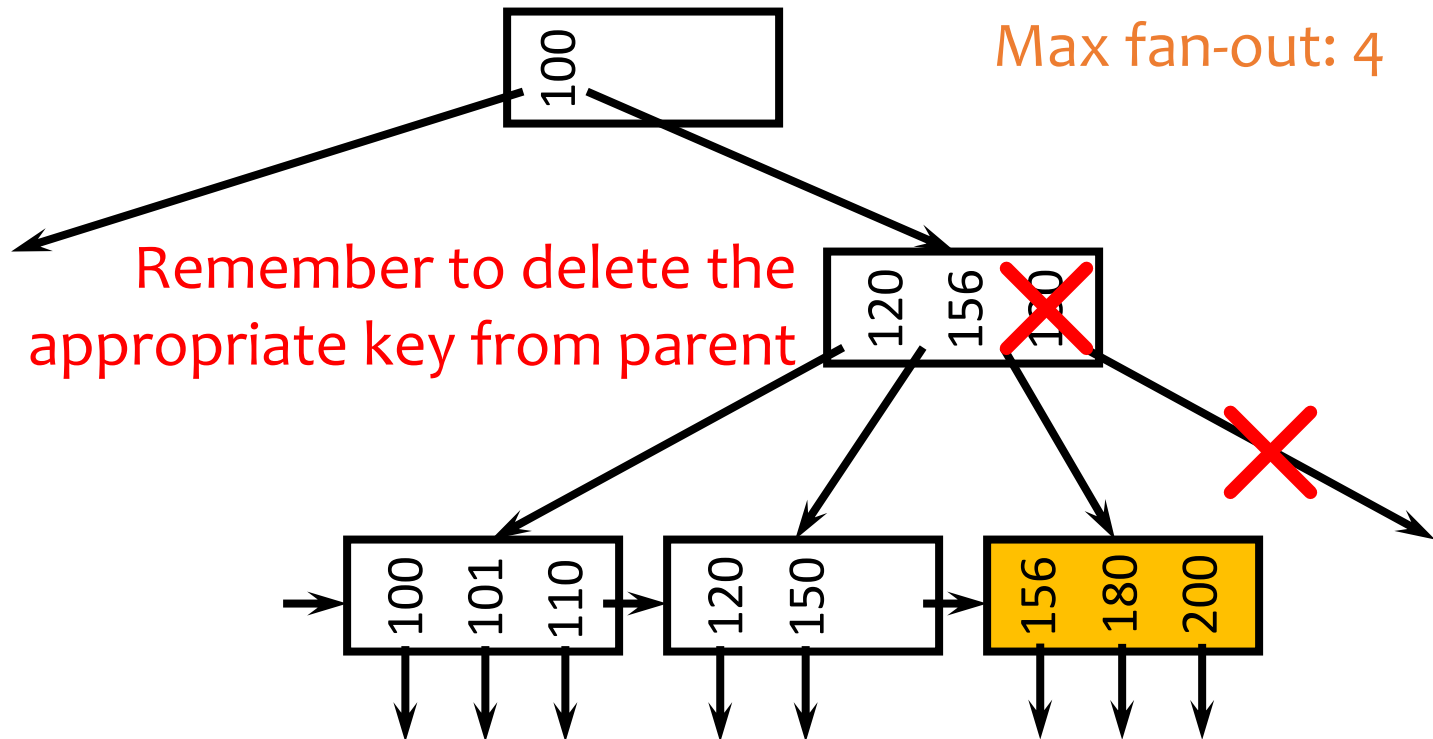
- Deletion can "propagate" all the way up to the root of the tree (not illustrated here)
  - When the root becomes empty, the tree "shrinks" by one level

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!
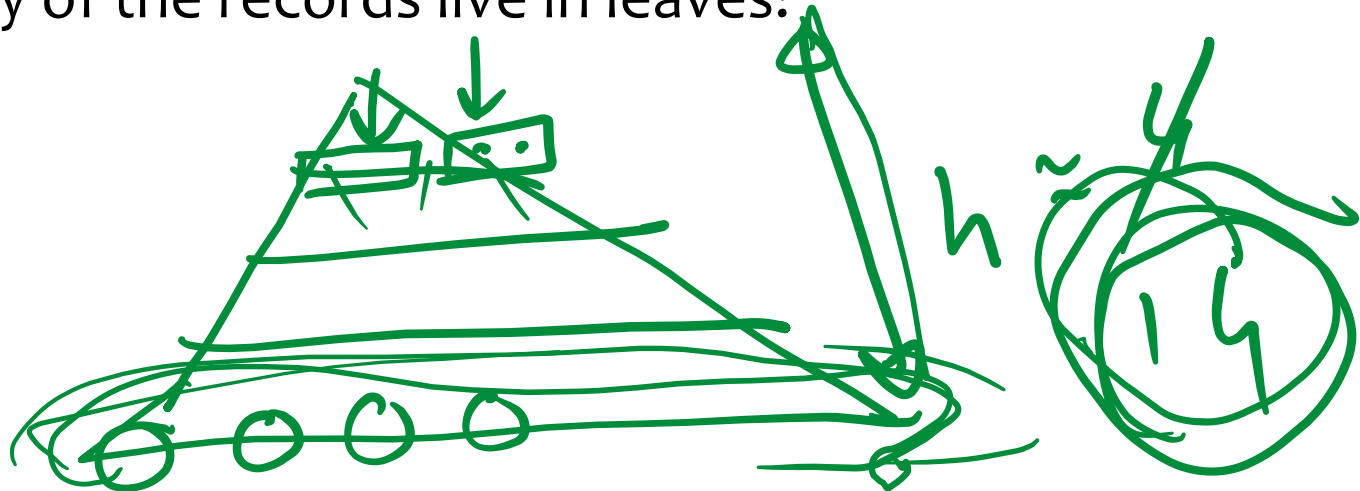
Beyond ISAM, B-, and B⁺-trees

- 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.

age = 15

index.

① data entries.

k
search key

age = 15

k*

②

15
15   13   15
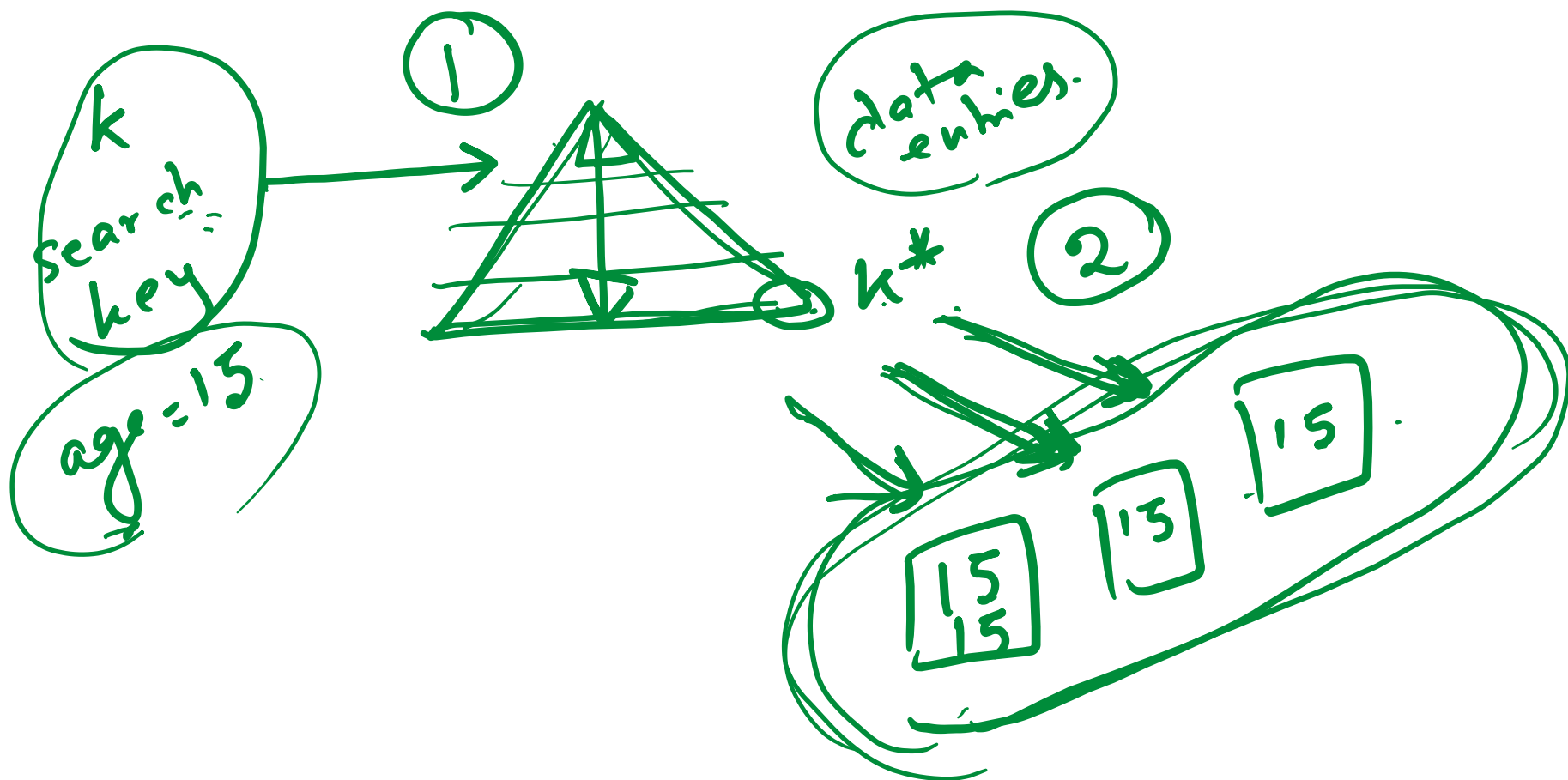
Clustered vs. Unclustered Index

- 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

Index entries
direct search for
data entries

Data entries

UNCLUSTERED

Data entries

10 11   11 12   12 12

(Index File)

(Data file)

10 11   11 12   12 12   12 13

Data Records

10 15   12 35   12   11
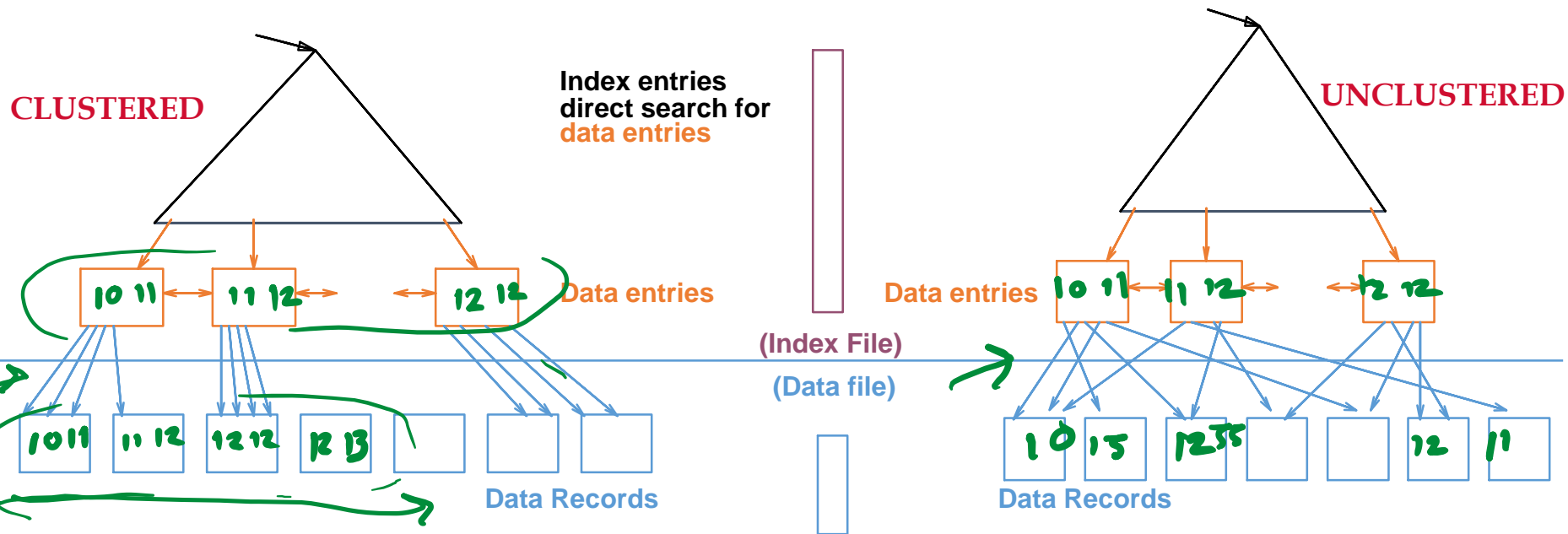
Data Records

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 tree node)

12

50 50 50 . . . 50

3
+
(12)

Uncl.

3+4

Cl.

12

user records.

3

| Bert |
| Alice. |
| Tom |
| Bam |

500

3 + 12

15
96
5

= 15

3.3 + 2

15
15

Uncl.

Cl.

4

≤ 3 + 4 = 7

15
15

15

15
15

15
15
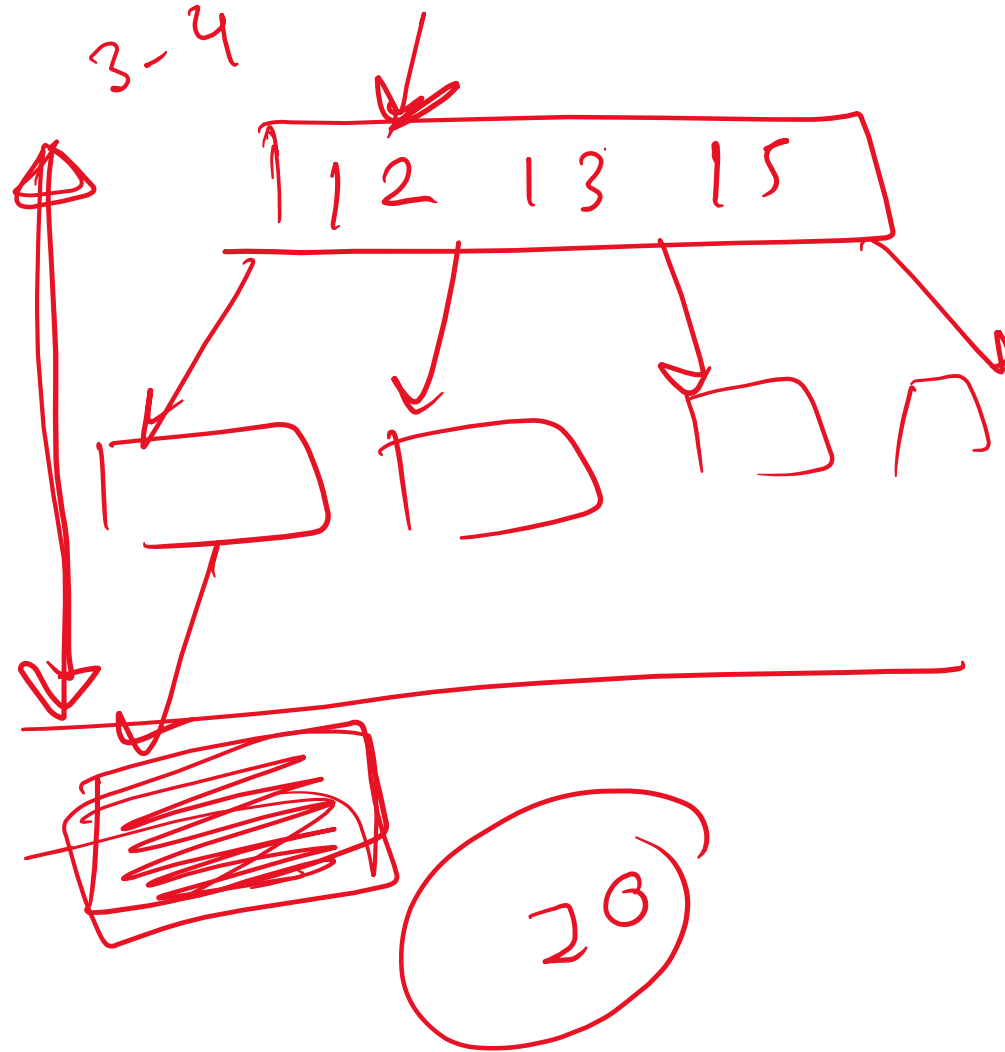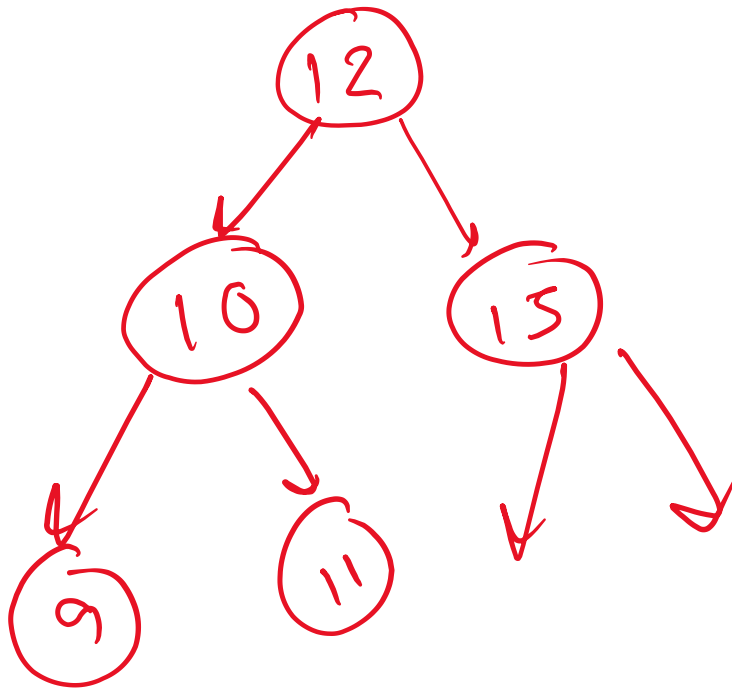
3-4

12 13 15
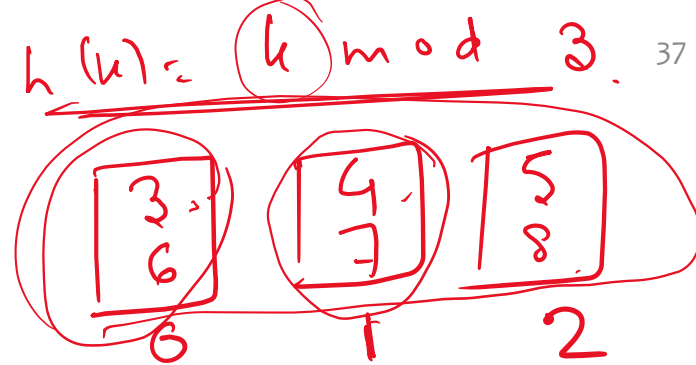
20
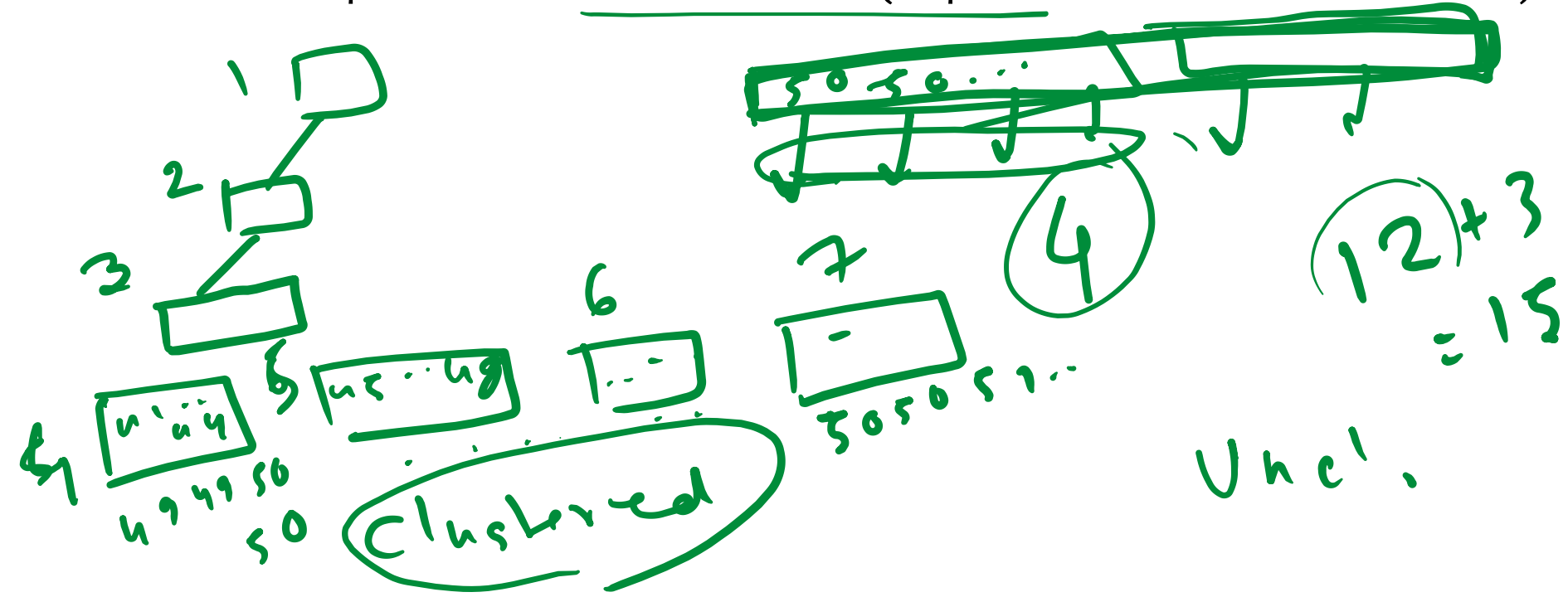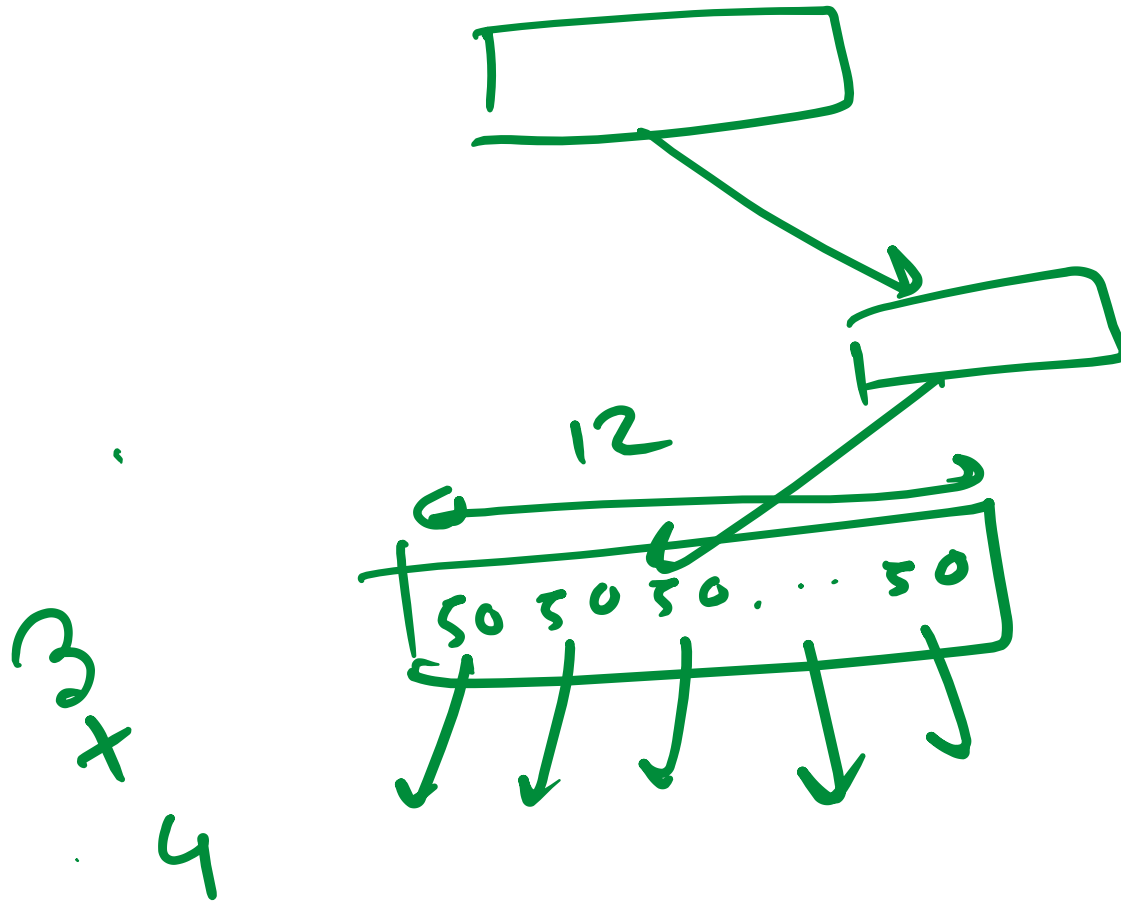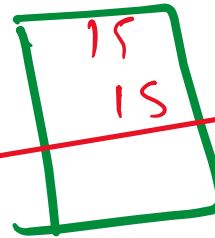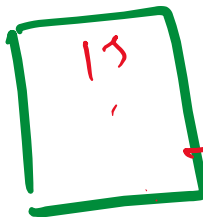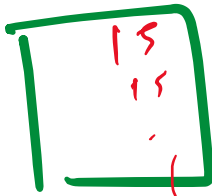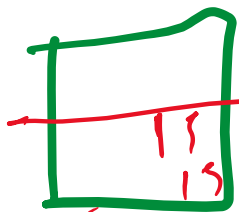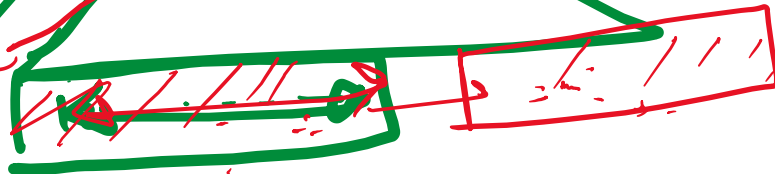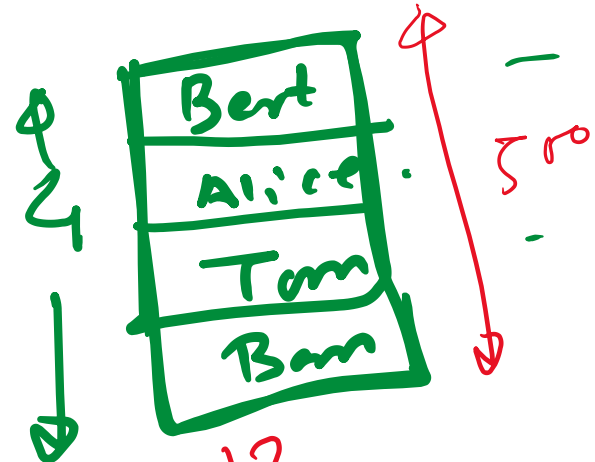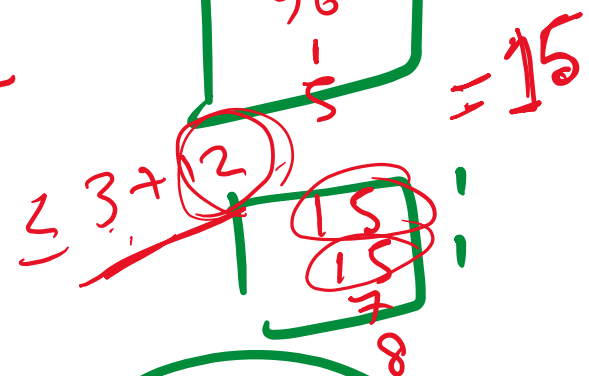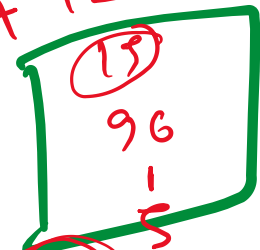
Hash vs. Tree Index

- 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
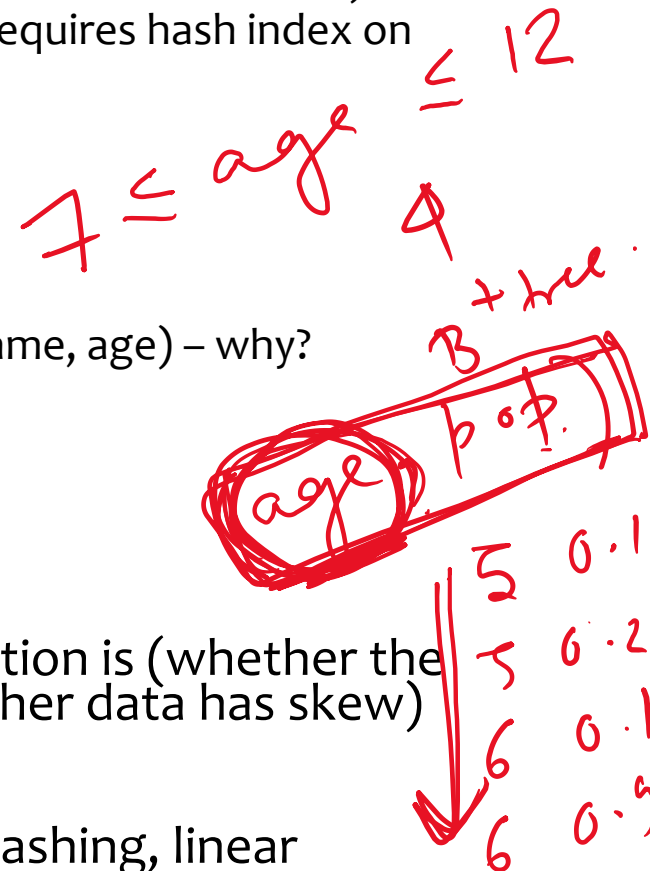
Hash

Tree

$=$

$=$ $\geq$

①

prefix

② $\times$

(age , p o p)

↳

age

③ parallel
   proc.

$\times$

Trade-offs for Indexes

- Should we use as many indexes as possible?
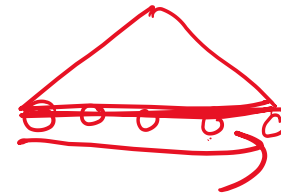
User name | id | age | bob | group

$$2^5 - 1$$

Trade-offs for Indexes

- Should we use as many indexes as possible?

- Indexes can make
  - queries go faster
  - updates slower

- Require disk space, too
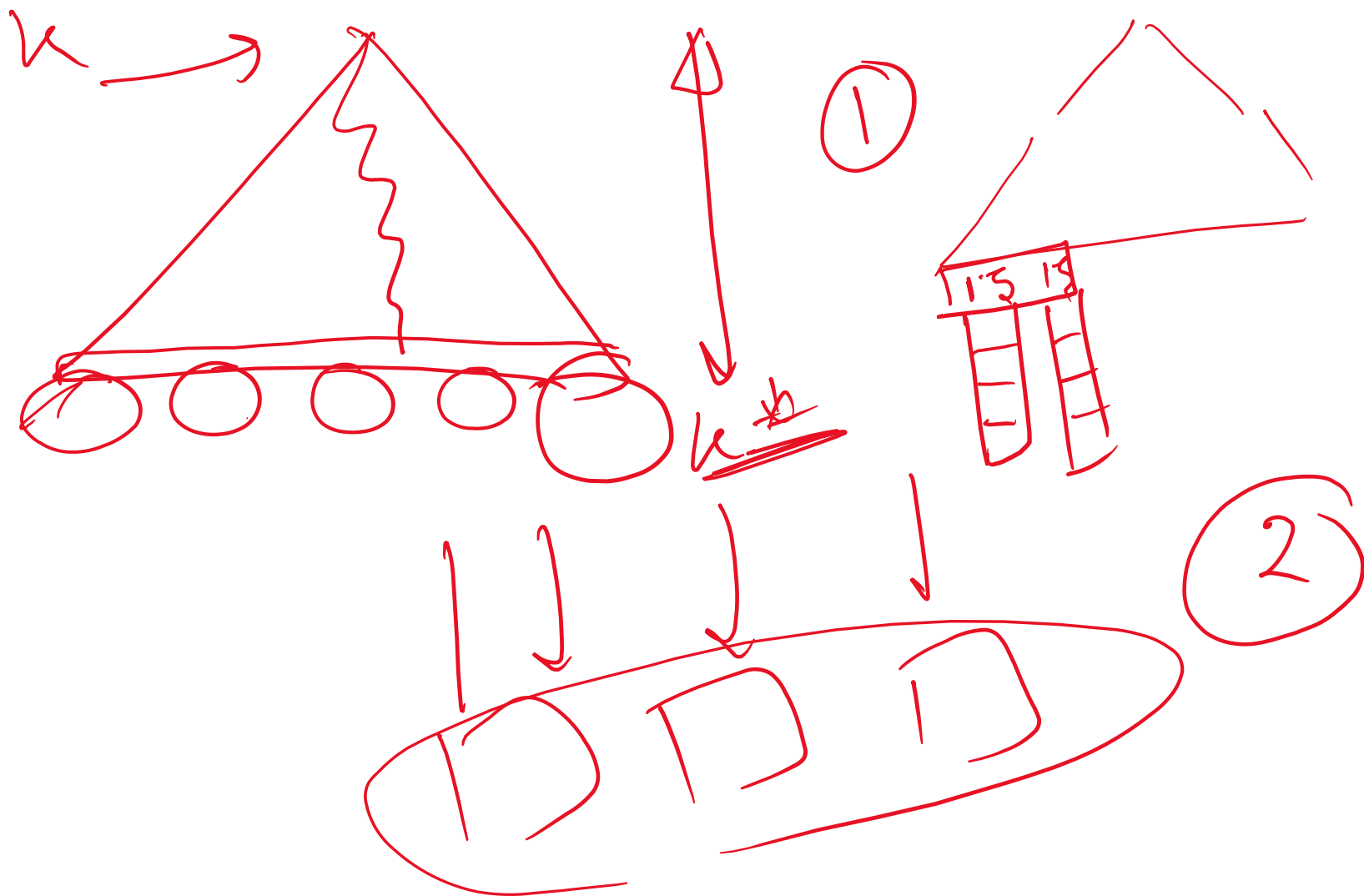
Index-Only Plans

- 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

age = 50 ^
name =
(name ^ age ^ name)

Alice   16
Bon   16
Bob   15
           19
Carol

(15, Bob
      Alice
  16   Bon
  16   Carol
      19
      log Alic