Indexing

CS348 Spring 2023

Instructor: Sujaya Maiyy

Sections: **002 & 004 only**

Announcements

- Milestone 1
  - Due today!

- Midterm Exam
  - Monday, June 26th
  - **Covers Lectures 1-10** [except lecture 6]
  - One 2-sided cheat sheet allowed

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

Indexes – conceptual understanding

- Commonly asked query: SELECT * FROM User WHERE name='…'
- Index on search key *Name*

Table

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
…	…	…	…

Index

	Bart
	Jessica
	Lisa
	Martin
	Milhouse
	Nelson
	Ralph
	Sherri
	Terri
	Windel

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

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

Smaller size

Easier to update

123	
456	
857	

Sparse index on *uid*

Must be clustered

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

Can tell directly if a record exists

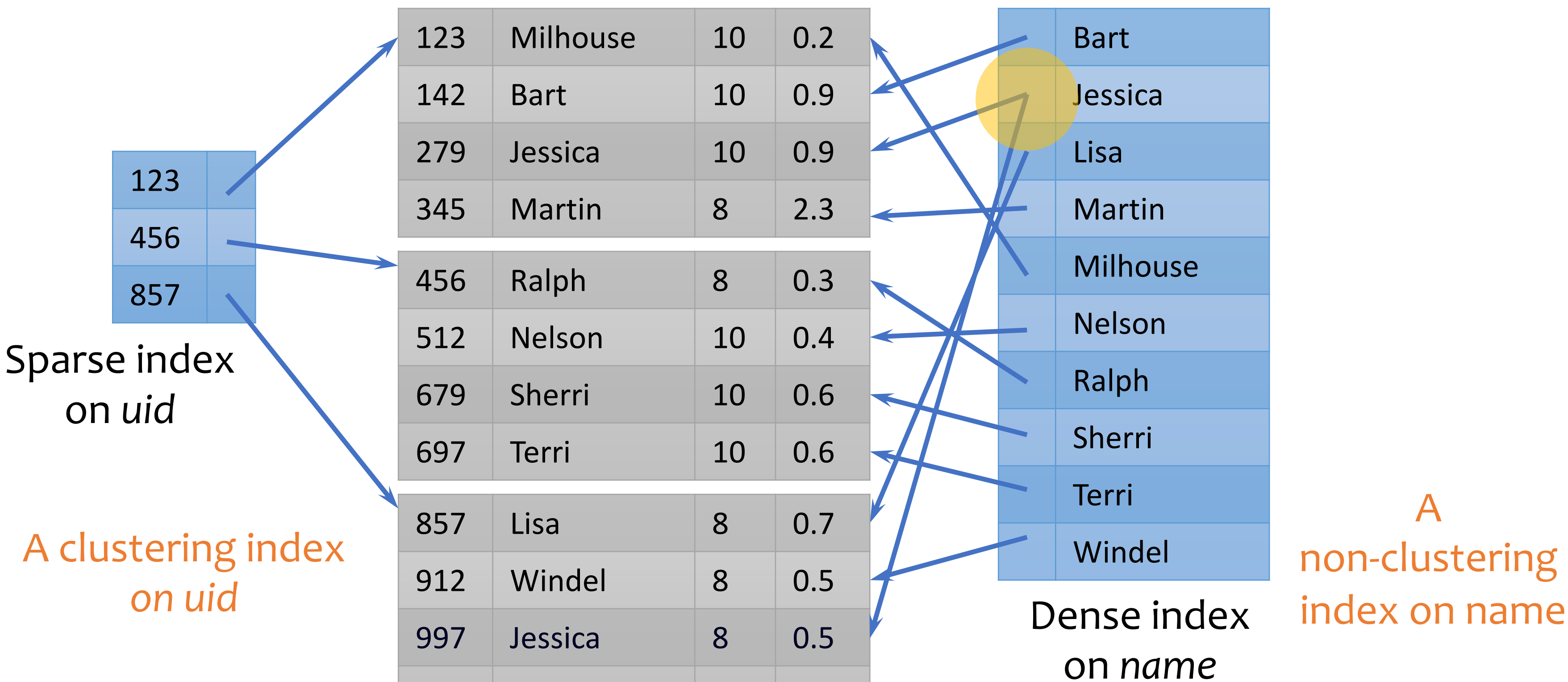
	Bart
	Jessica
	Lisa
	Martin
	Milhouse
	Nelson
	Ralph
	Sherri
	Terri
	Windel

May not fit into memory

Dense index on *name*

Clustering v.s. non-clustering indexes

- An index on attribute A is a clustering index if tuples in the relation with similar values for A are stored together in the same block.
- Other indices are non-clustering (or secondary) indices.
- Note: A relation may have at most one clustering index, and any number of non-clustering indices.



Primary and secondary indexes

- Primary index
  - Created for the primary key of a table
  - Records are usually clustered by the primary key
  - Clustering index → sparse
- Secondary index
  - Non-clustering index, usually dense (to find each search key value, since records are not clustered by this search key)
- 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);

Outline

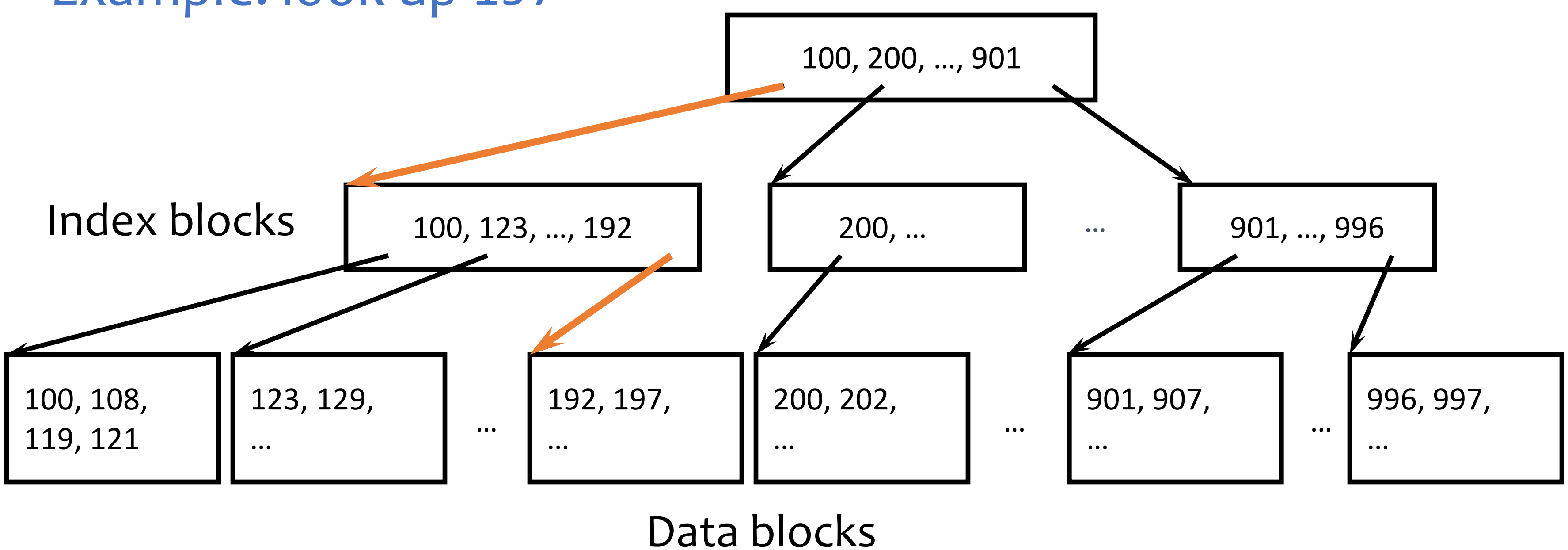
- Types of indexes
  - Sparse v.s. dense
  - Clustering v.s. non-clustering
  - Primary v.s. secondary
- Index structure
- How to use index

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

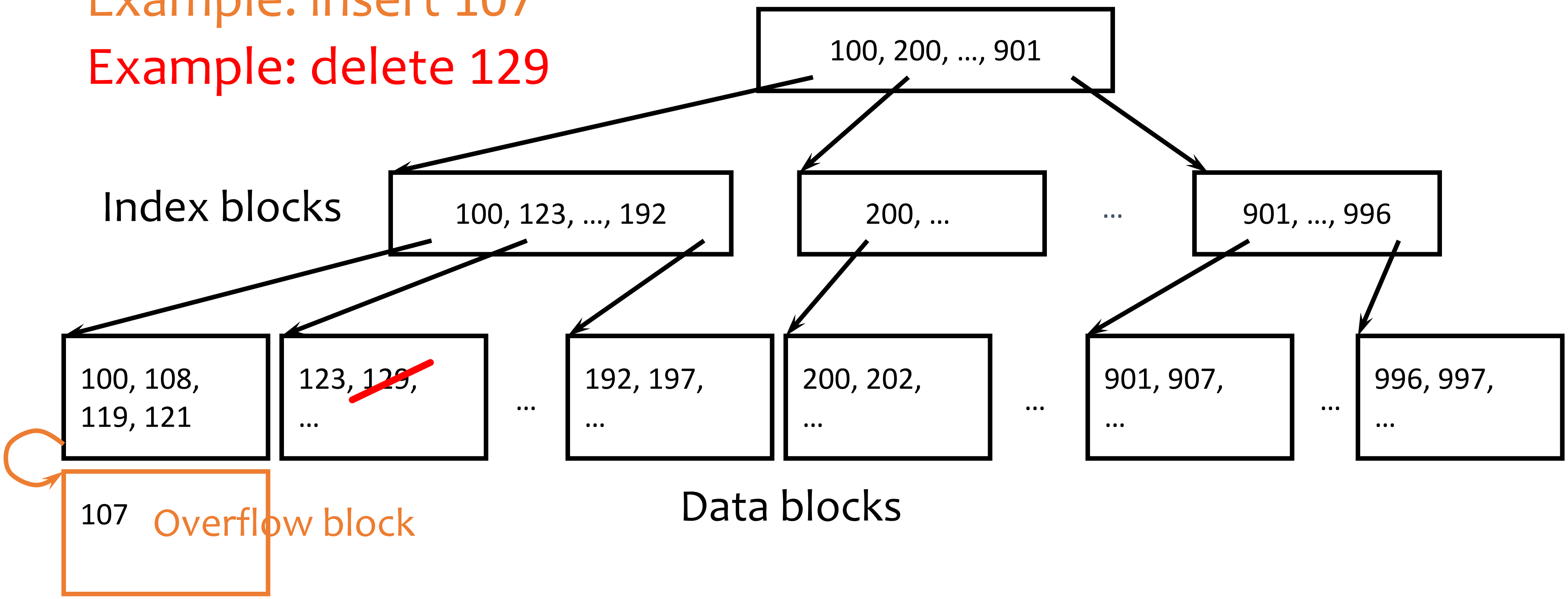
Example: look up 197



Updates with ISAM

Example: insert 107

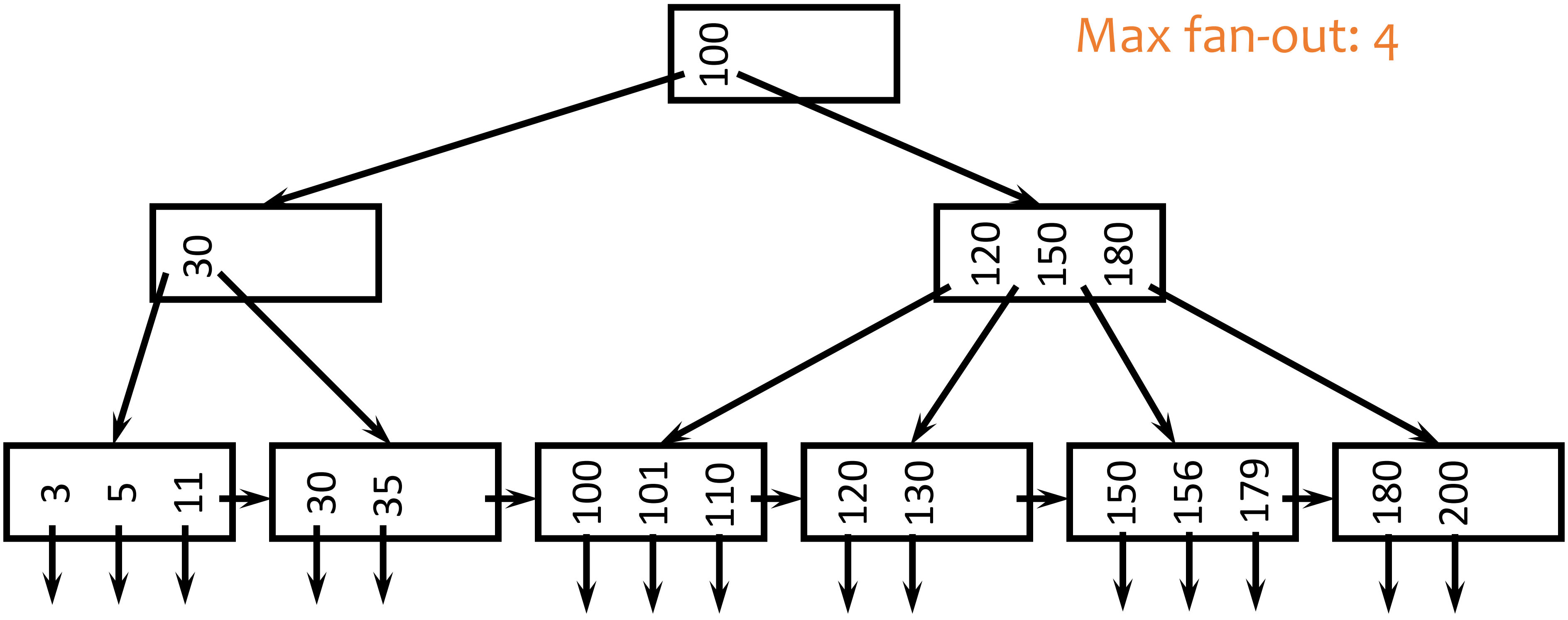
Example: delete 129



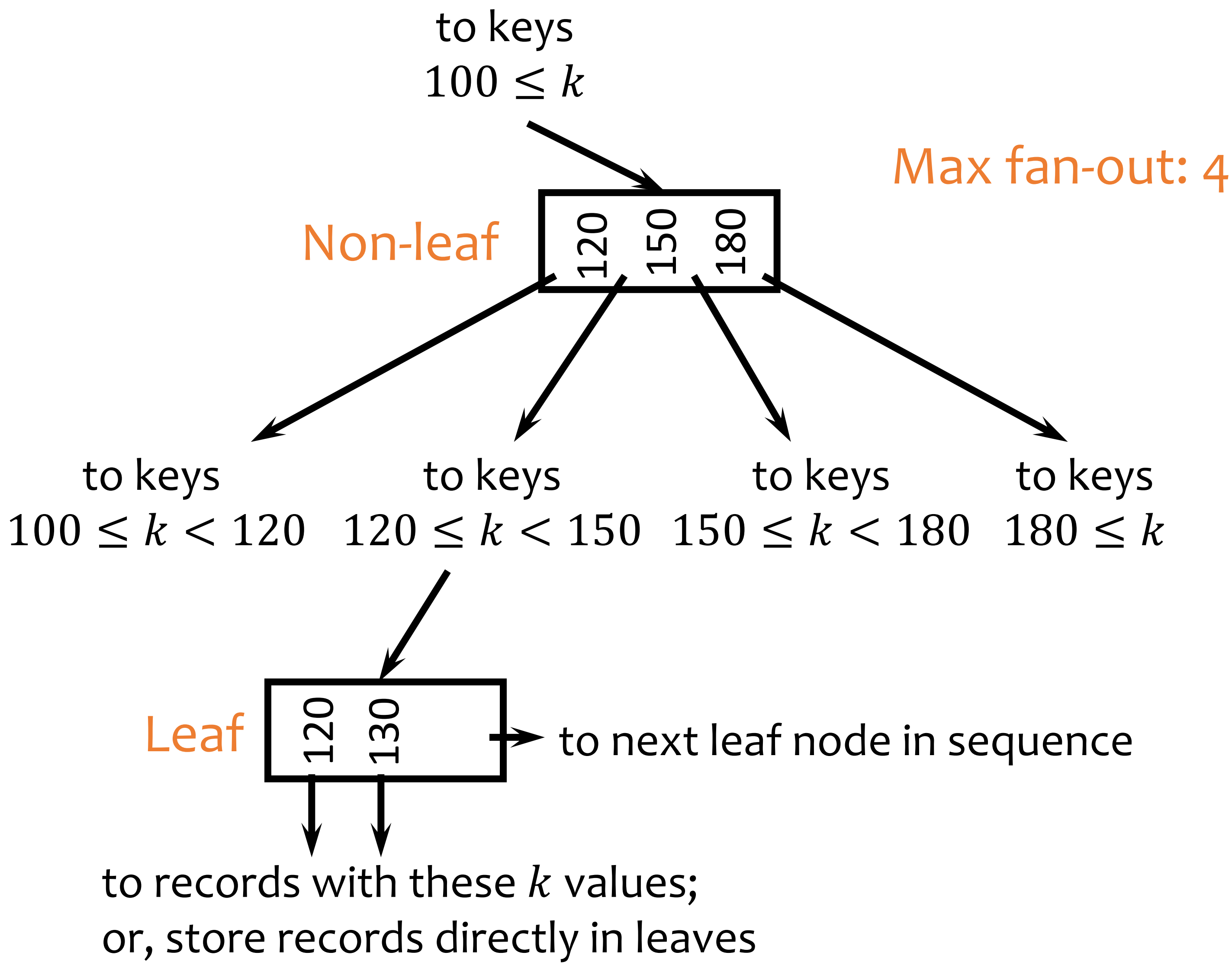
- Overflow chains and empty data blocks degrade performance
  - Worst case: most records go into one long chain, so lookups require scanning all data!

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



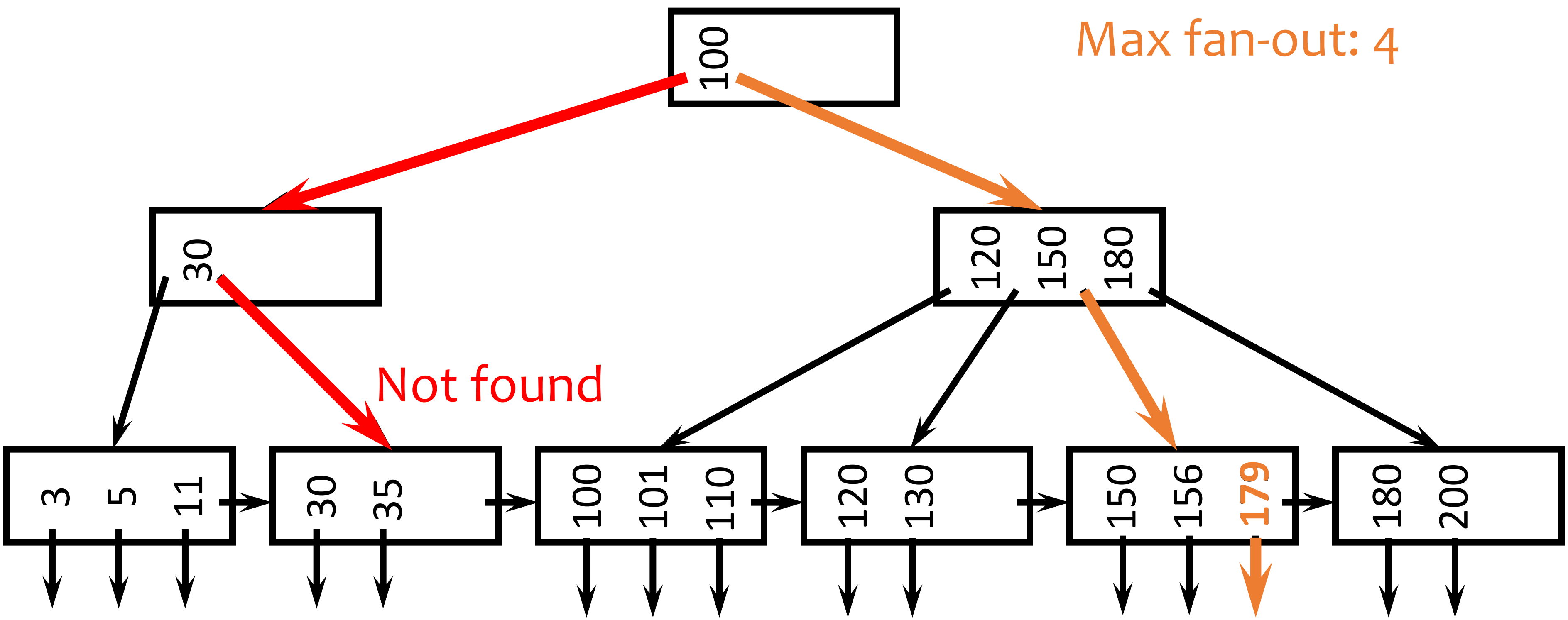
B+-tree balancing properties

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

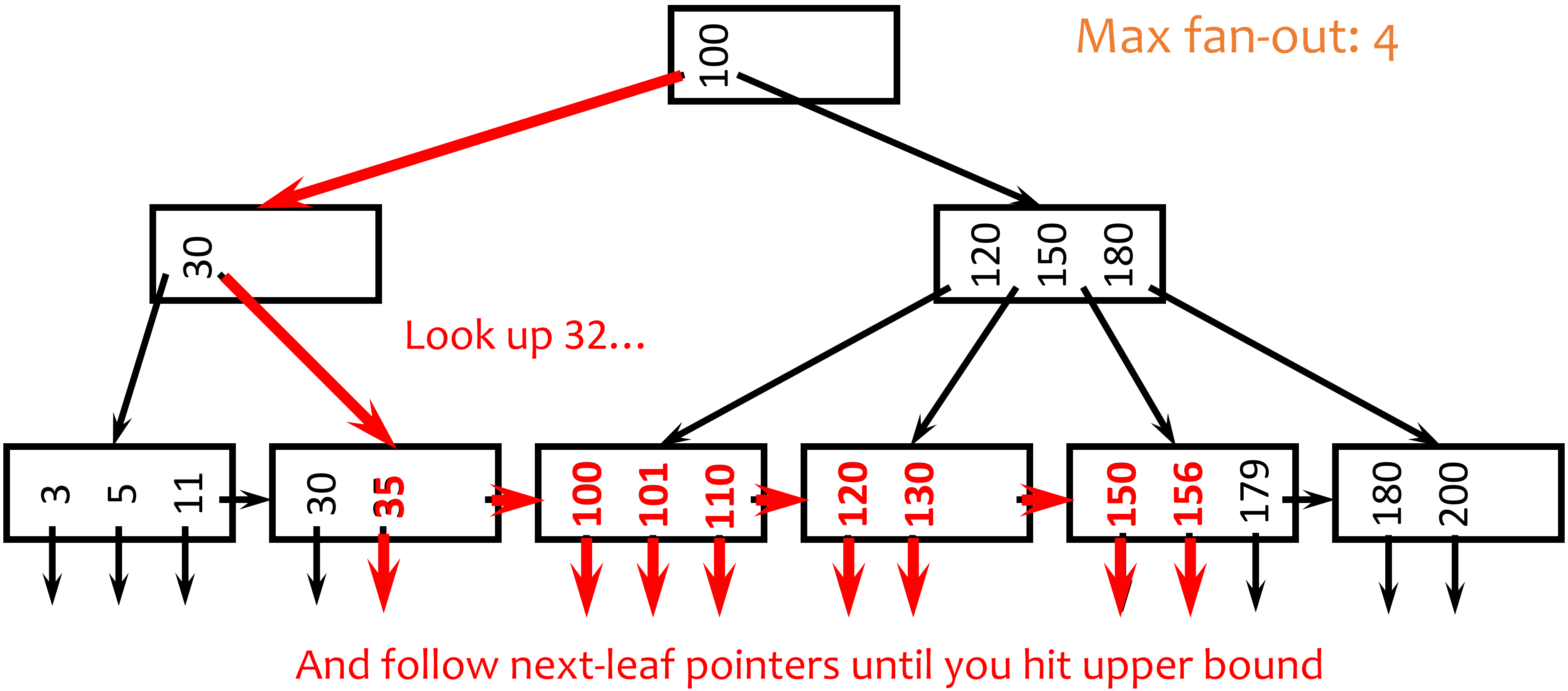
Lookups

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



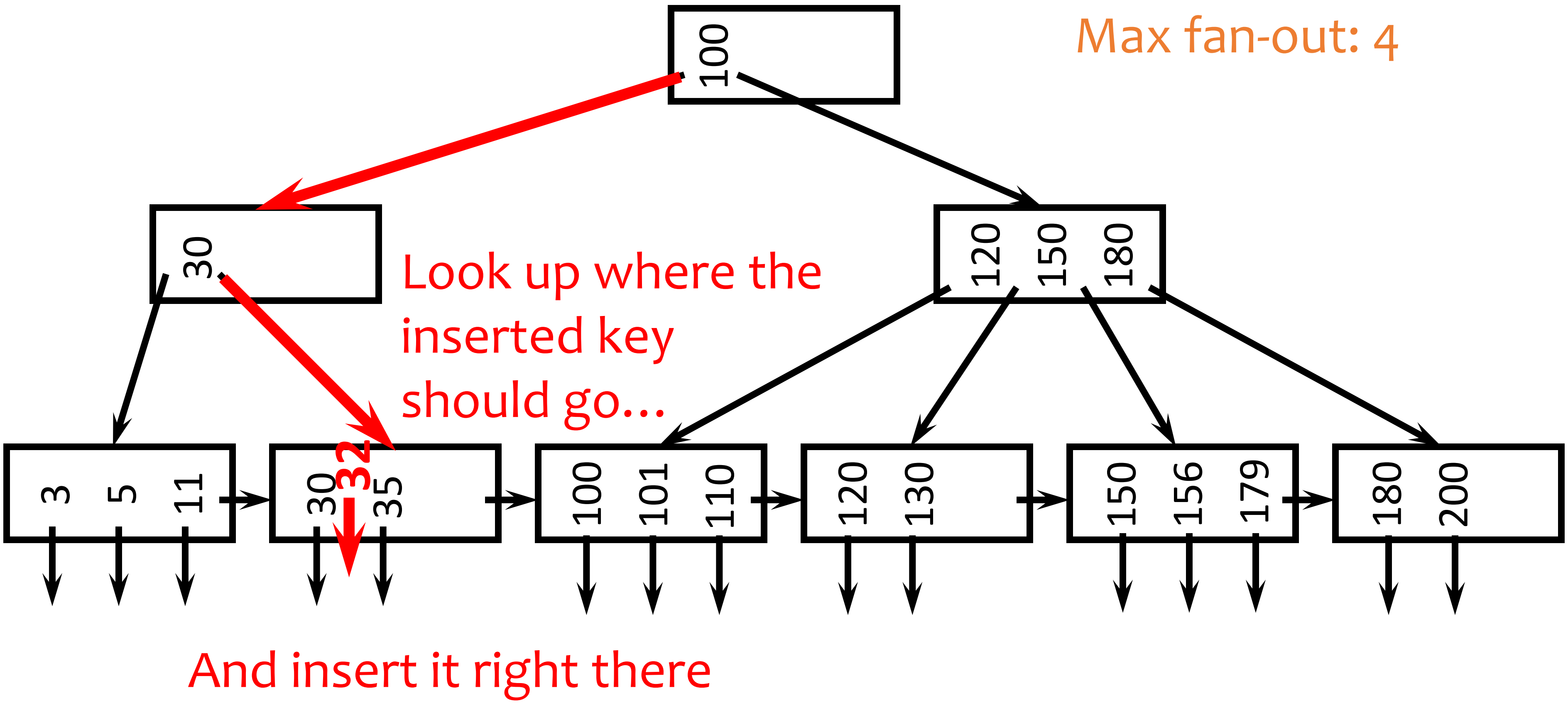
Range query

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



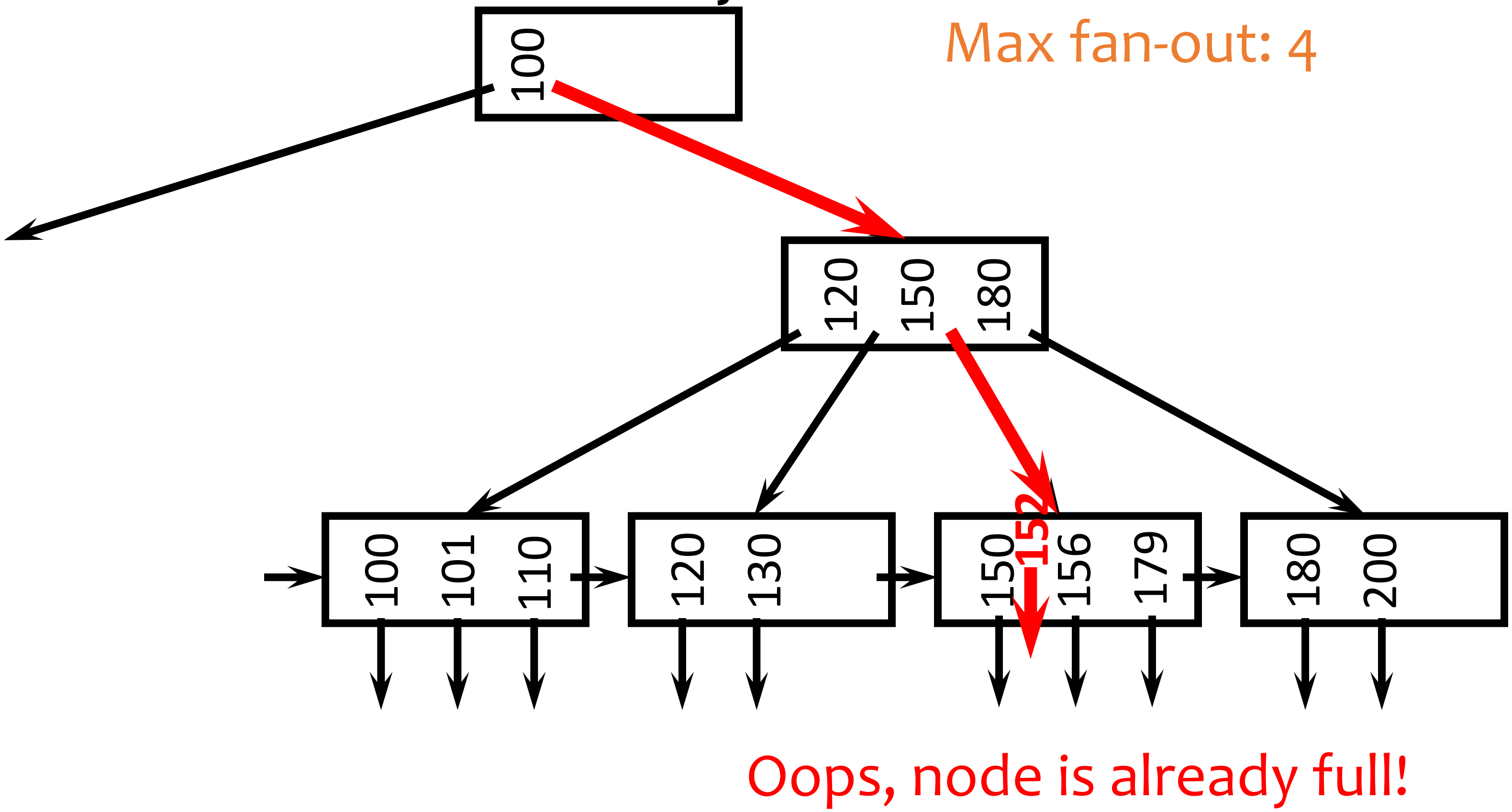
Insertion

- Insert a record with search key value 32



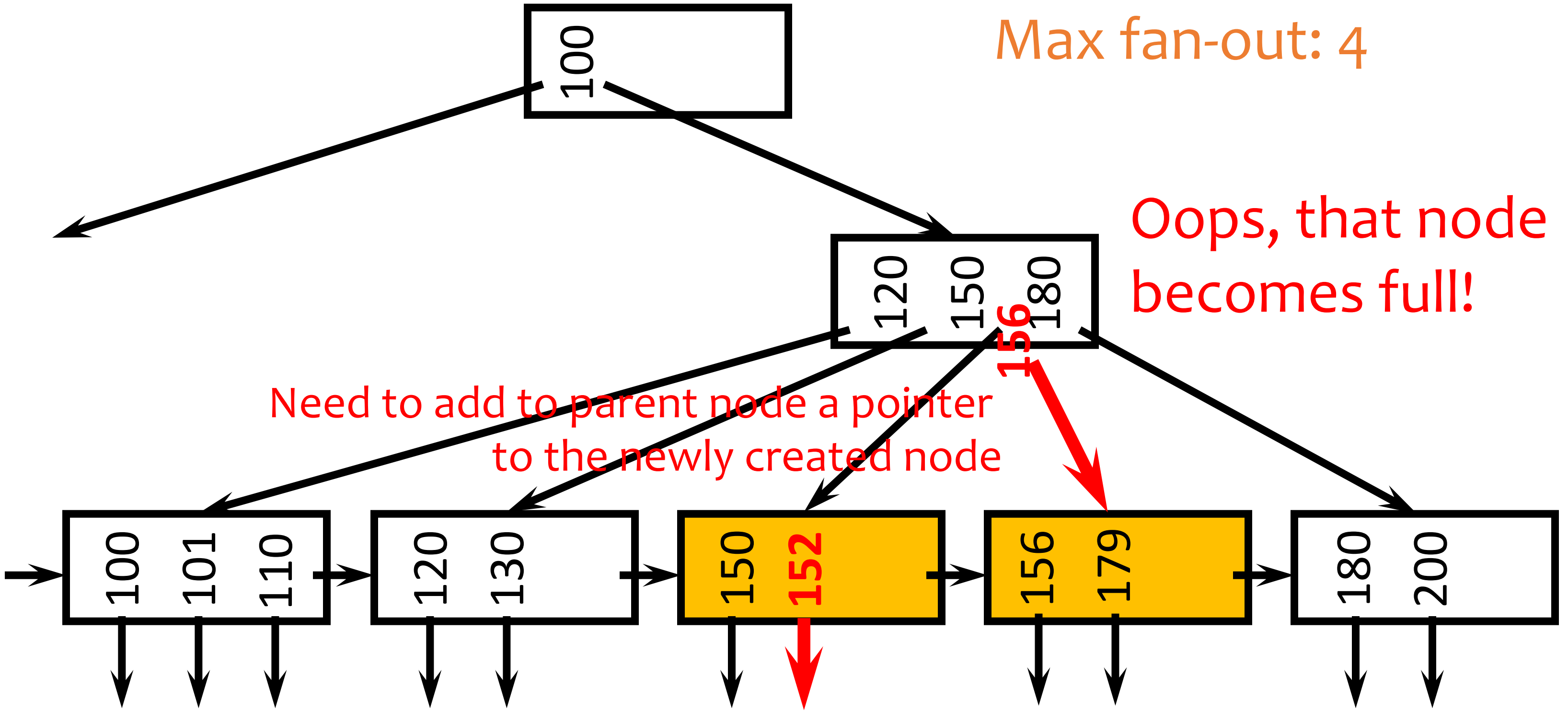
Another insertion example

- Insert a record with search key value 152

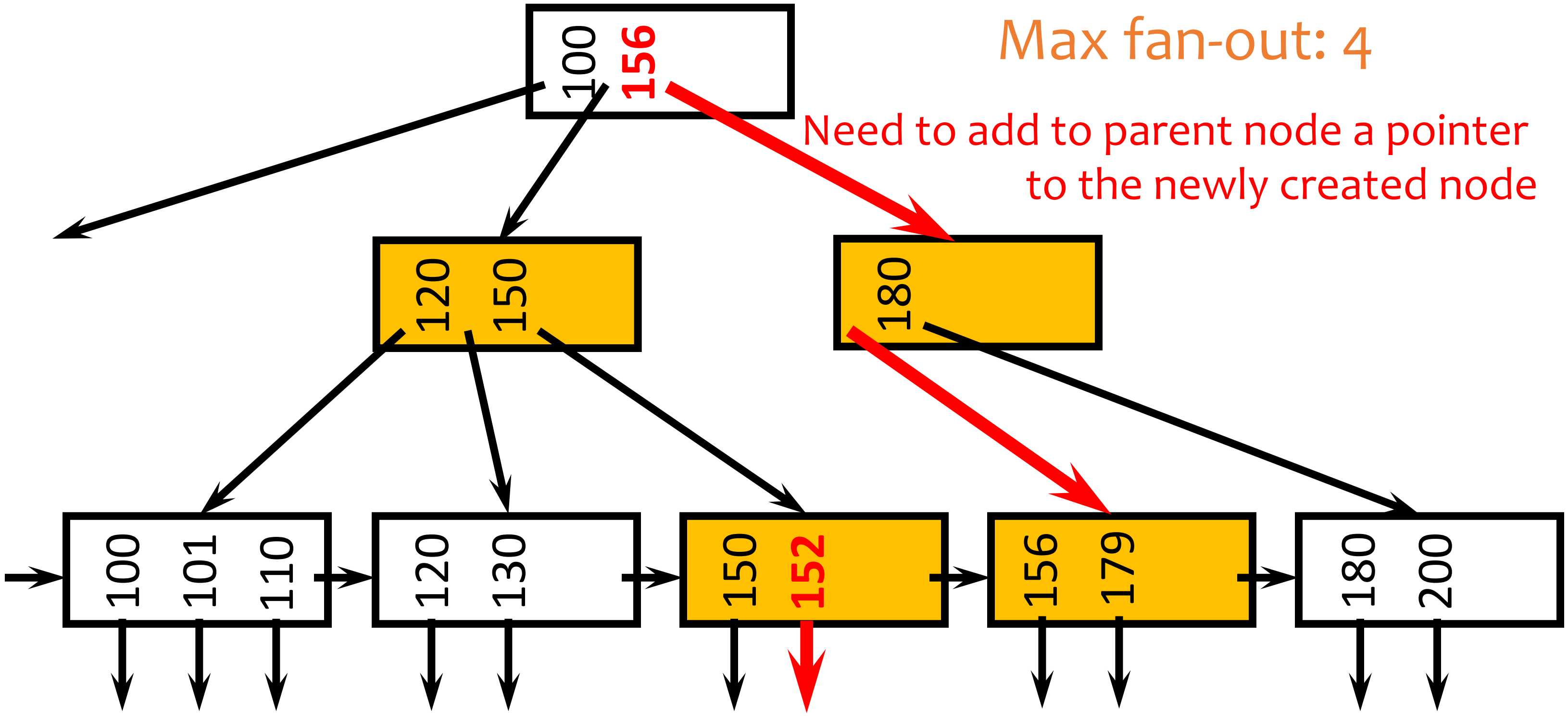


Node splitting

Max fan-out: 4



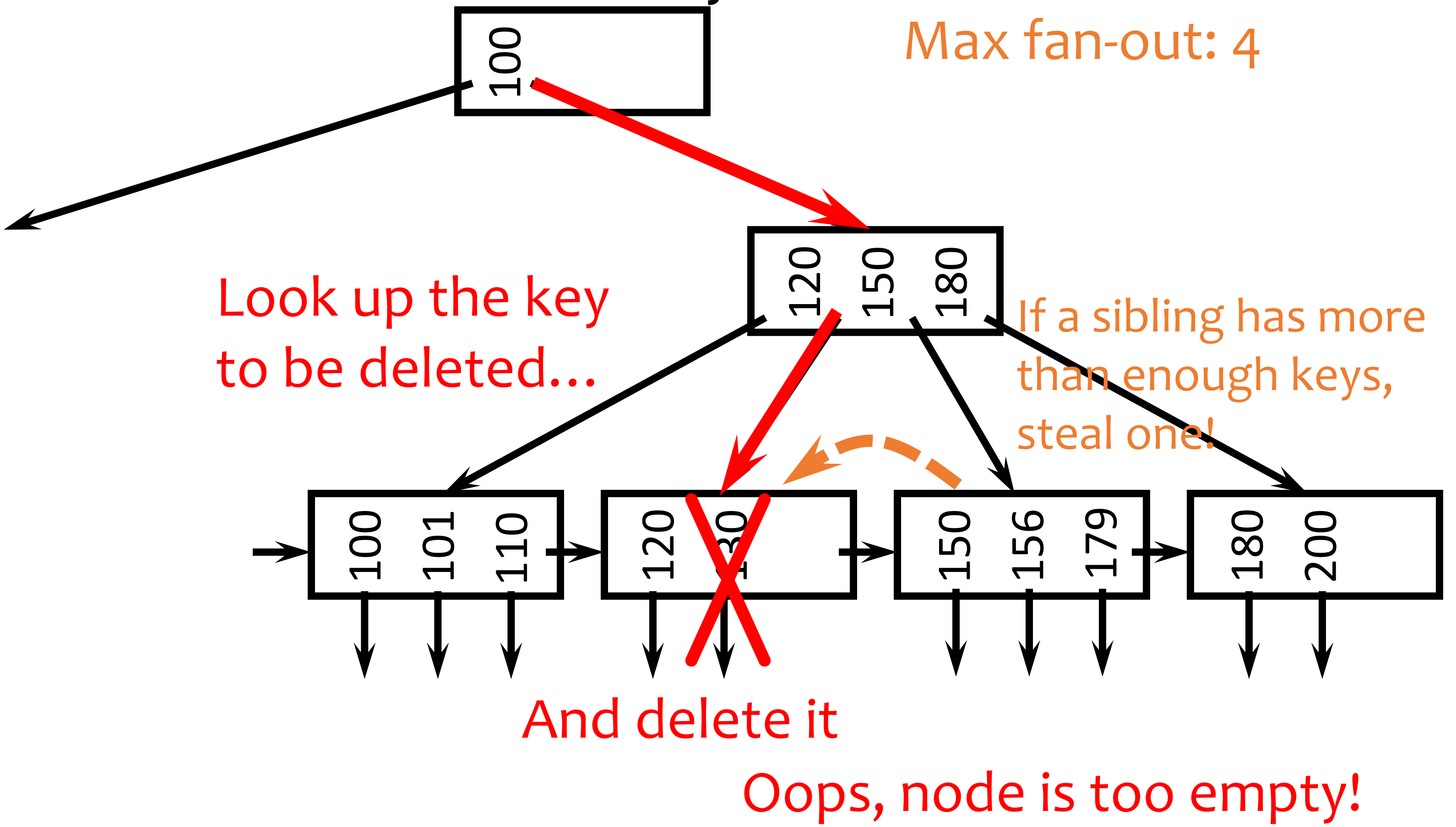
More node splitting



- 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

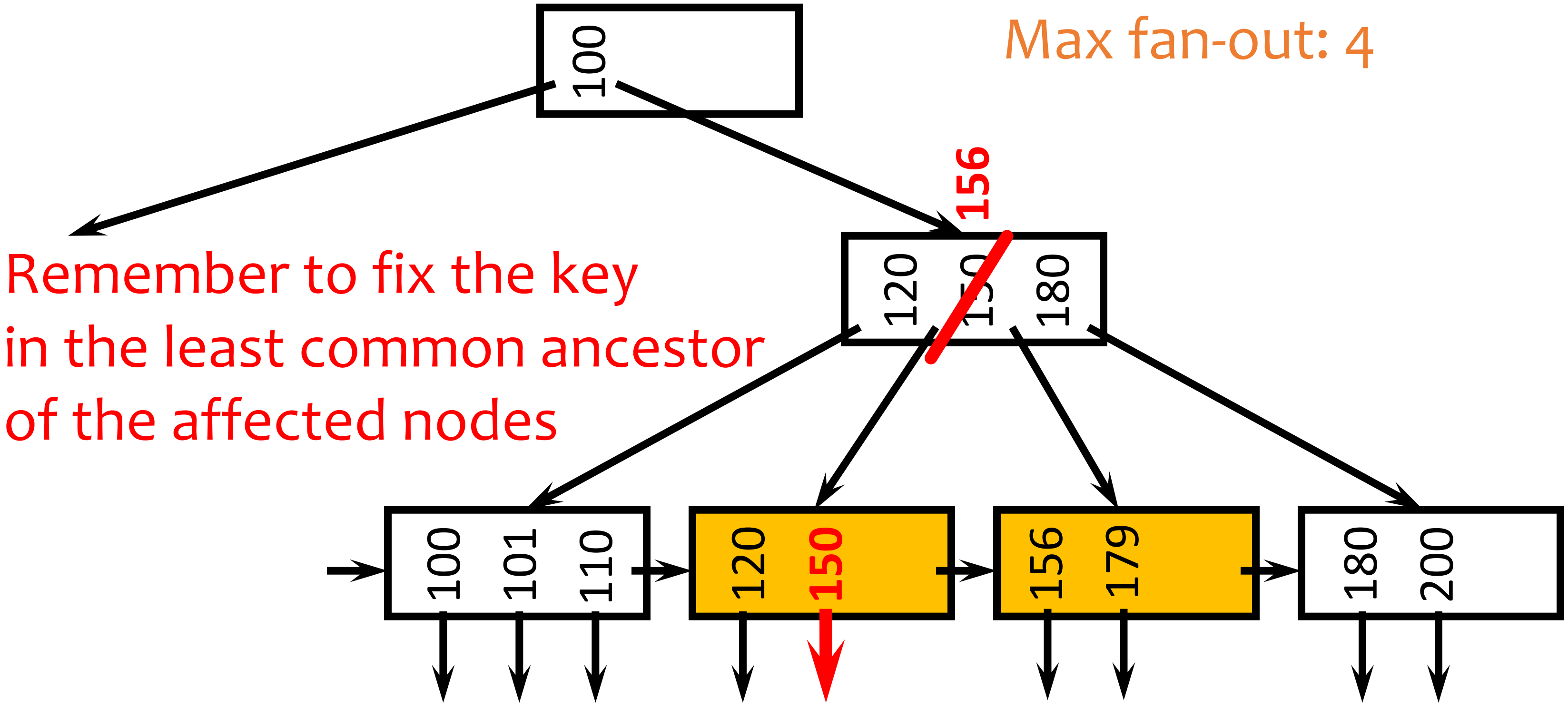
Deletion

- Delete a record with search key value 130



Stealing from a sibling

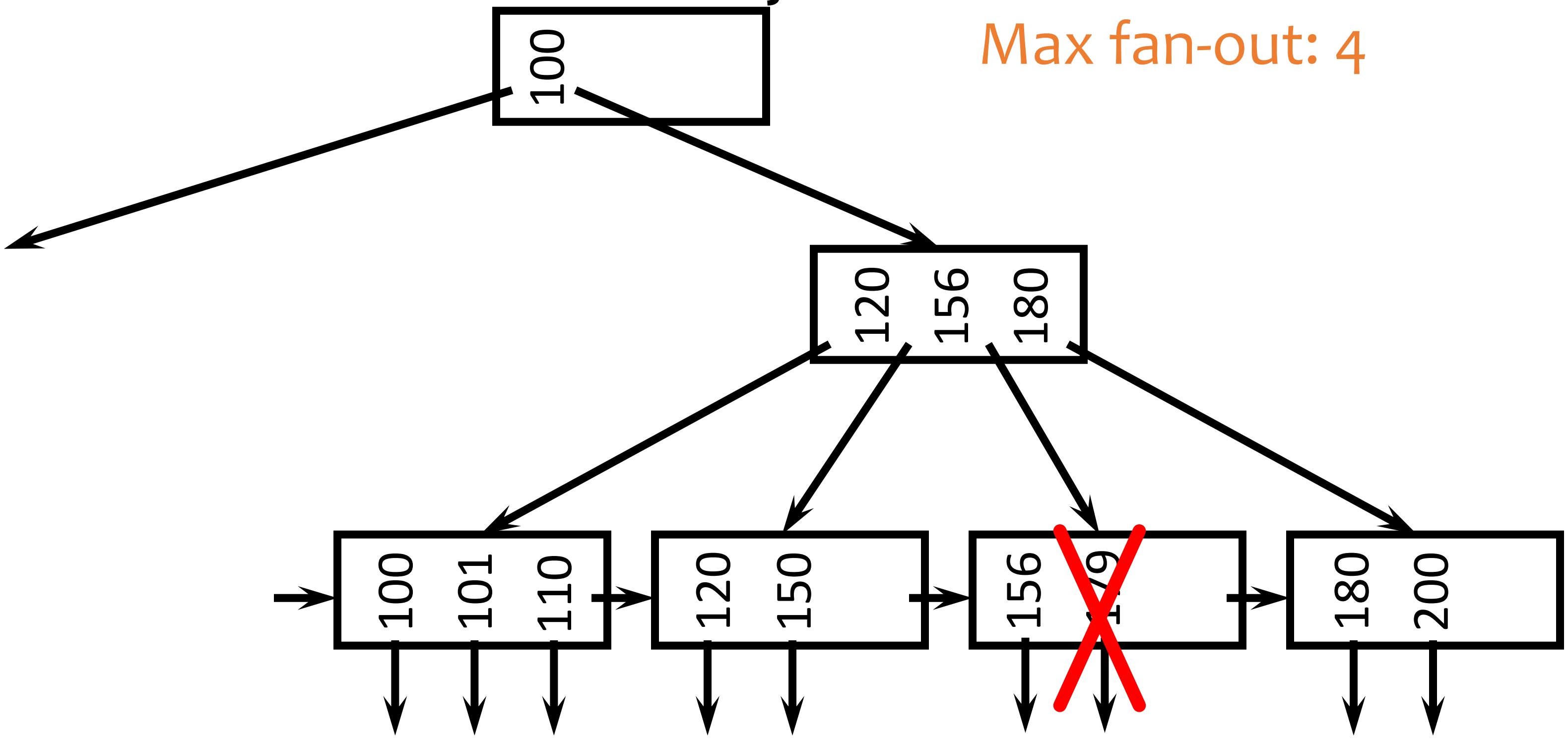
Max fan-out: 4



Another deletion example

- Delete a record with search key value 179

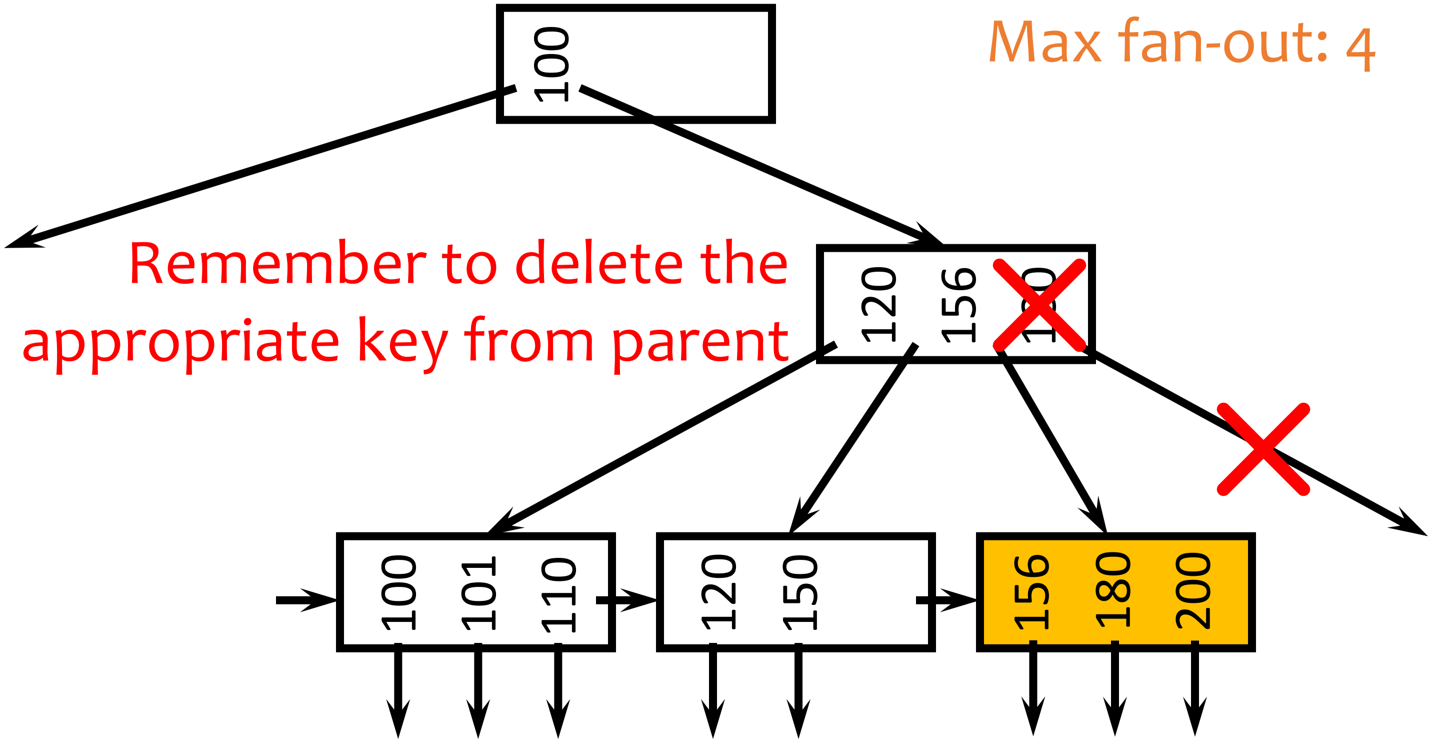
Max fan-out: 4



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 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

- How to use index

Multi-attribute indices

- Index on several attributes of the same relation.
  - CREATE INDEX NameIndex ON User(LastName,FirstName);

tuples (or tuple pointers) are organized first by Lastname. Tuples with a common lastname are then organized by Firstname.

- This index would be *useful* for these queries:
  - **select * from** User **where** Lastname = ‘Smith’
  - **select * from** User **where** Lastname = ‘Smith’ and Firstname=‘John’
- This index would be not *useful* at all for this query:
  - **select * from** User **where** Firstname=‘John’

Index-only plan

- For example:
  - SELECT firstname, pop FROM User WHERE pop > ‘0.8’ AND firstname = ‘Bob’;
  - non-clustering index on (firstname, pop)

- A (non-clustered) index contains all the columns needed to answer the query without having to access the tuples in the base relation.
  - Avoid one disk I/O per tuple
  - The index is much smaller than the base relation

Physical design guidelines for indices

1. Don't index unless the performance increase outweighs the update overhead
2. Attributes mentioned in WHERE clauses are candidates for index search keys
3. Multi-attribute search keys should be considered when
4. Choose indexes that benefit as many queries as possible
5. Each relation can have at most one clustering scheme; therefore choose it wisely
   - Target important queries that would benefit the most
     - Range queries benefit the most from clustering
   - A multi-attribute index that enables an index-only plan does not benefit from being clustered

Case study

- User(<u>uid</u>, name, age, pop)
- Group(<u>gid</u>, name, date)
- Member(<u>uid, gid</u>)

- Common queries
    1. List the name, pop of users in a particular age range
    2. List the uid, age, pop of users with a particular name
    3. List the average pop of each age
    4. List all the group info, ordered by their starting date
    5. List the average pop of a particular group given the group name
- Pick a set of clustering/non-clustering indexes for these set of queries (without worrying too much about storage and update cost)

Case study

- User(<u>uid</u>, name, age, pop)
- Group(<u>gid</u>, name, date)
- Member(<u>uid, gid</u>)

A clustering index on User(age)

A non-clustering index on User(name)

- Common queries
  1. List the name, pop of users in a particular age range
  2. List the uid, age, pop of users with a particular name
  3. List the average pop of each age
  4. List all the group info, ordered by their starting date
  5. List the average pop of a particular group given the group name

Case study

- User(uid, name, age, pop)
- Group(gid, name, date)
- Member(uid, gid)

A non-clustering index on User(age, pop)
→ index-only plan

A clustering index on User(age)

A non-clustering index on User(name)

- Common q

1. List the name, pop users in a particular age range
2. List the uid, age, pop of users with a particular name
3. List the average pop of each age
4. List all the group info, ordered by their starting date
5. List the average pop of a particular group given the group name

Case study

- User(uid, name, age, pop)
- Group(gid, name, date)
- Member(uid, gid)

A non-clustering index on User(age, pop) → index-only plan

A clustering index on User(age)

A non-clustering index on User(name)

- Common q

1. List the name, pop … users in a particular age range
2. List the uid, age, pop … f users with a particular name
3. List the average pop of each age
4. List all the group info, ordered by their starting date
5. List the average pop of a particular group given the group name

A clustering index on Group(date)

Case study

- User(uid, name, age, pop)
- Group(gid, name, date)
- Member(uid, gid)

A non-clustering index on User(age, pop)
→ index-only plan

A clustering index on User(age)

A non-clustering index on User(name)

- Common q

1. List the name, pop users in a particular age range
2. List the uid, age, pop f users with a particular name
3. List the average pop of each age
4. List all the group info, ordered by their starting date
5. List the average pop of a particular group given the group name

A join between User(uid, … ,pop) , Member(uid,gid), Group(gid, name)

A clustering index on Group(date)

(i) Search gid by a particular name
→ Clustering/non-clustering index on Group(name)?

(ii) Search uid by a particular gid
→ Clustering/non-clustering index on Member(gid)?

(iii) Search pop by a particular uid
→ Clustering/non-clustering index on User(uid)?

Non-clustering, as we already have a clustered index on Group(date)

If many other queries require a clustering index on Group(name), we may reconsider!

Case study

- User(<u>uid</u>, name, age, pop)
- Group(<u>gid</u>, name, date)
- Member(<u>uid, gid</u>)

- Common q

1. List the name, pop   users in a particular age range
2. List the uid, age, pop f users with a particular name
3. List the average pop of each age
4. List all the group info, ordered by their starting date
5. List the average pop of a particular group given the group name

A non-clustering index on User(age, pop) → index-only plan

A clustering index on User(age)

A non-clustering index on User(name)

A clustering index on Group(date)

A join between User(uid, … ,pop) , Member(uid,gid), Group(gid, name)

(i) Search gid by a particular name
→ Non-clustering index on Group(name)

(ii) Search uid by a particular gid
→ Clustering/non-clustering index on Member(gid)?

Clustering -> all records of the same gid are clustered

Or clustering index on Member(gid,uid)

(iii) Search pop by a particular uid
→ Clustering/non-clustering index on User(uid)?

Case study

- User(<u>uid</u>, name, age, pop)
- Group(<u>gid</u>, name, date)
- Member(<u>uid, gid</u>)

A non-clustering index on User(age, pop) → index-only plan

A clustering index on User(age)

A non-clustering index on User(name)

- Common q
  1. List the name, pop users in a particular age range
  2. List the uid, age, pop f users with a particular name
  3. List the average pop of each age
  4. List all the group info, ordered by their starting date
  5. List the average pop of a particular group given the group name

A clustering index on Group(date)

A join between User(uid, … ,pop) , Member(uid,gid), Group(gid, name)

(i) Search gid by a particular name
→ Non-clustering index on Group(name)

(ii) Search uid by a particular gid
→ Clustering index on Member(gid)

(iii) Search pop by a particular uid
→ Clustering/non-clustering index on User(uid)?

Or non-clustering index on User(uid, pop) → index-only plan, if without worrying about storage/update cost

Non-clustering, as we already have a clustering index on User(age)

Summary

- Types of indexes:
  - Dense v.s. sparse
  - Clustering v.s. non-clustering
  - Primary v.s. secondary
- Indexing structure
  - ISAM
  - B+-tree
- How to use index
  - Use multi-attribute indices
  - Index-only plan
  - General guideline