Indexing

CS348

Instructor: Sujaya Maiyya

Hash based indexes with range queries

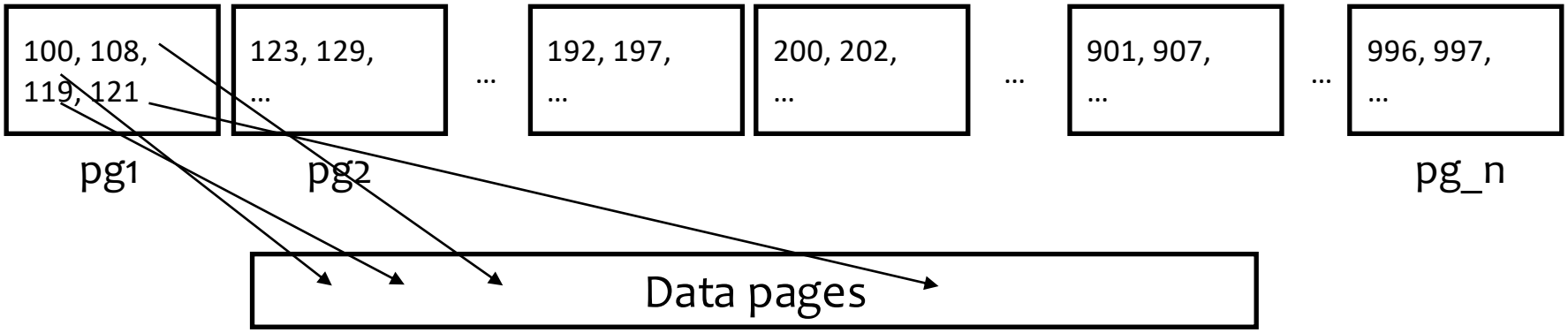
- Select * Where age > 10 and age <= 20;
- Index lookup for 11, 12,… 20 and union/merge the results
- Can we do better? Pay the index lookup cost only once?

Outline

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

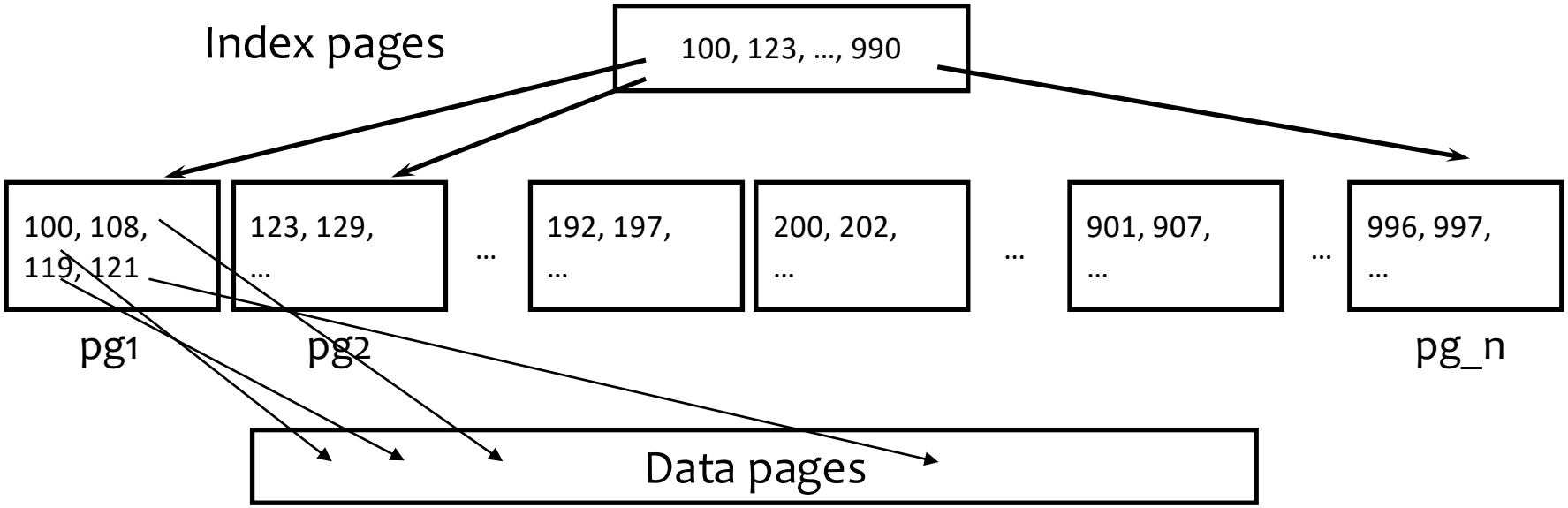
Layered indexing

- Goal of index: minimize I/O when retrieving data pages
- Keep the index keys in a page in sorted order
- What if an index is page is too big?
  - ☞Store index info across multiple pages
- But we don't want to scan all n index pages!



Layered indexing

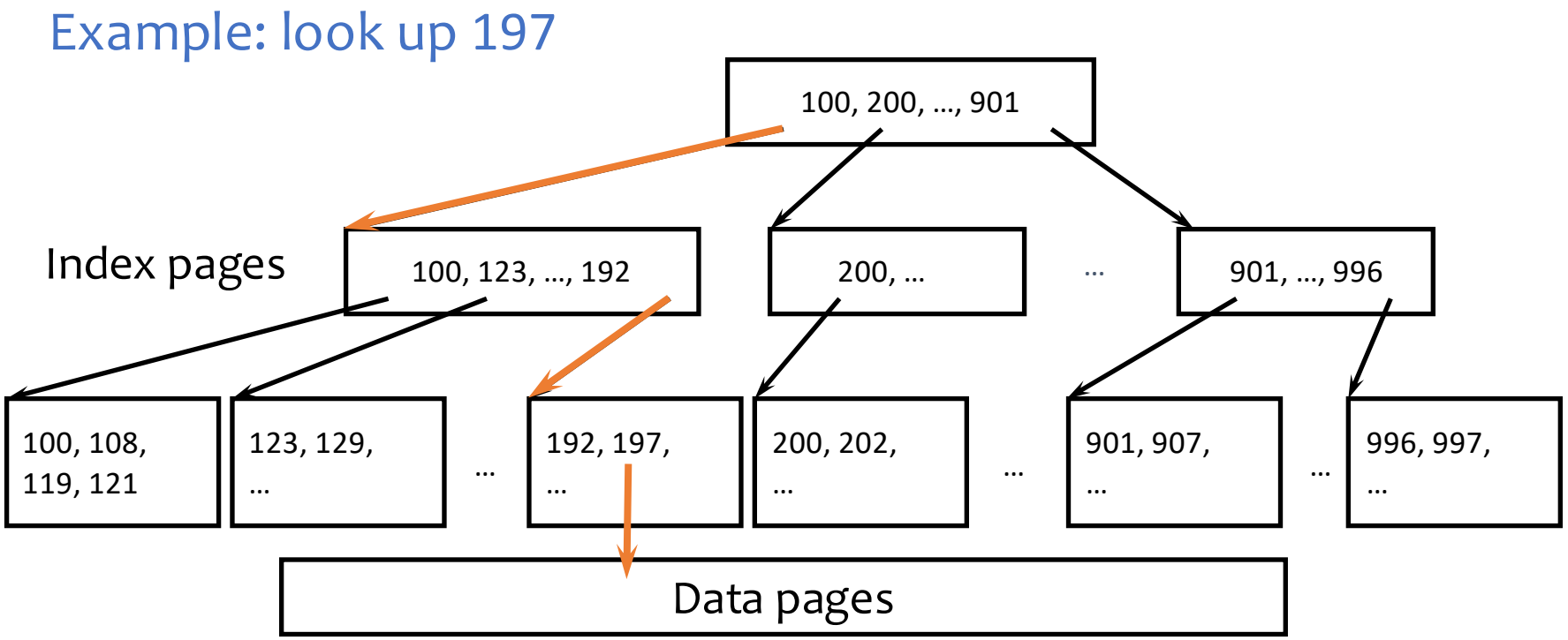
- Put a sparse index on top!
- If top layer doesn't fit in one page?



Layered indexing: ISAM

- Add more pages
- Put another (sparse) index on top of that!

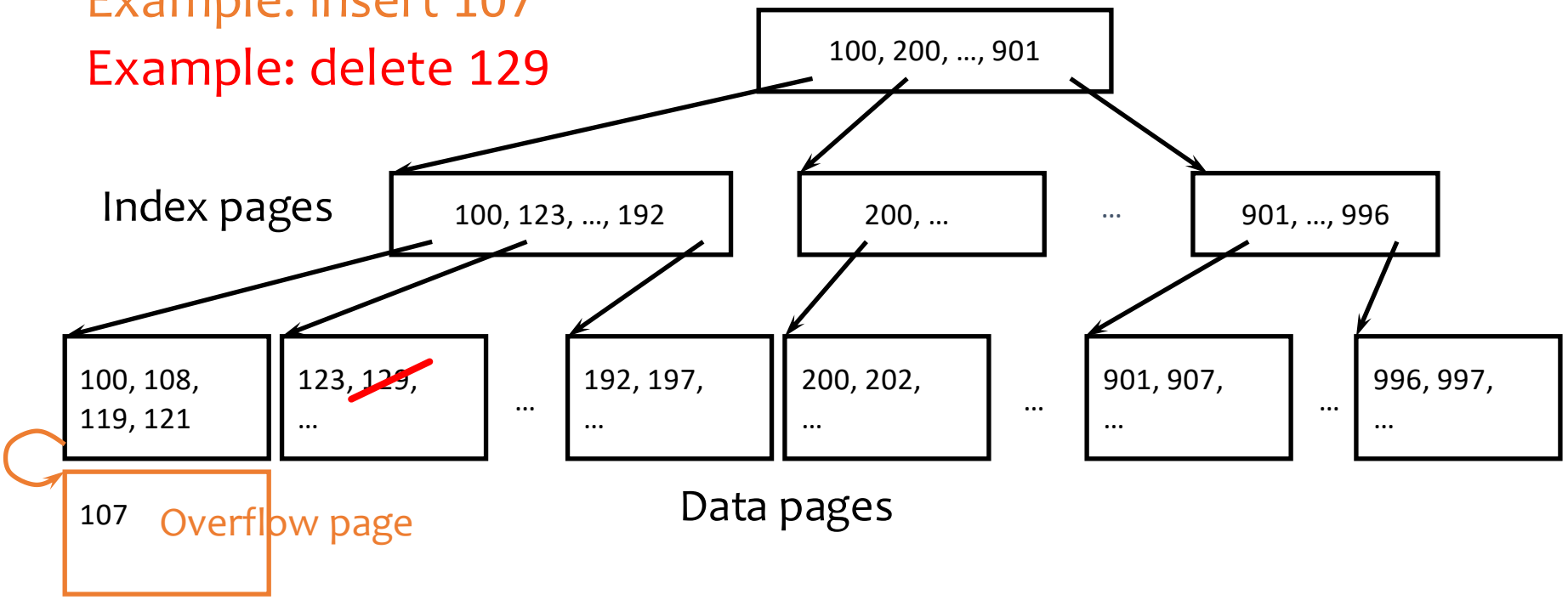
☞ISAM (Index Sequential Access Method)



Updates with ISAM

Example: insert 107

Example: delete 129

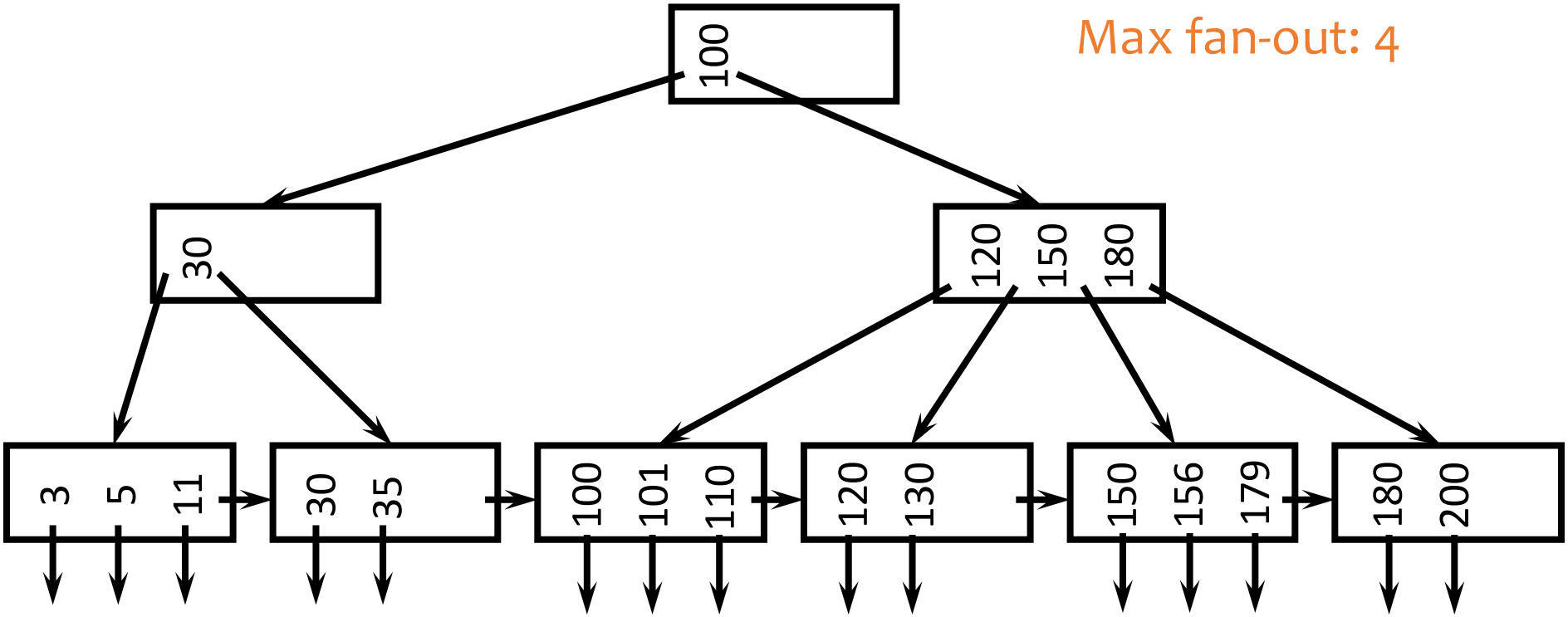


- Overflow chains and empty data pages degrade performance
  - Worst case: most records go into one long chain, so lookups require scanning all data!

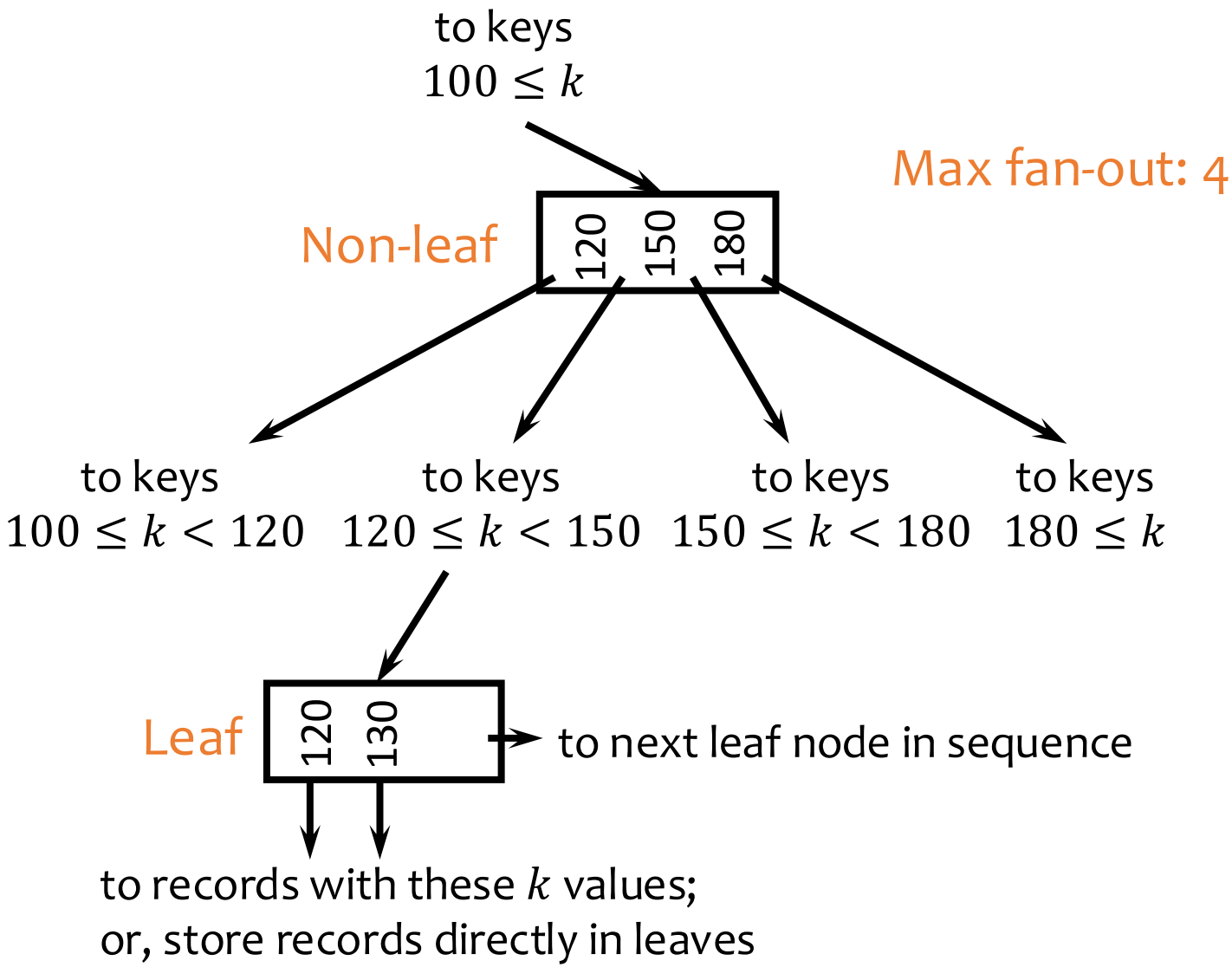
In class exercises

B$^+$-tree

- A hierarchy of nodes with intervals
- Balanced: good performance guarantee
- Disk-based: one node per page; large fan-out



Sample B⁺-tree nodes



B⁺-tree balancing properties

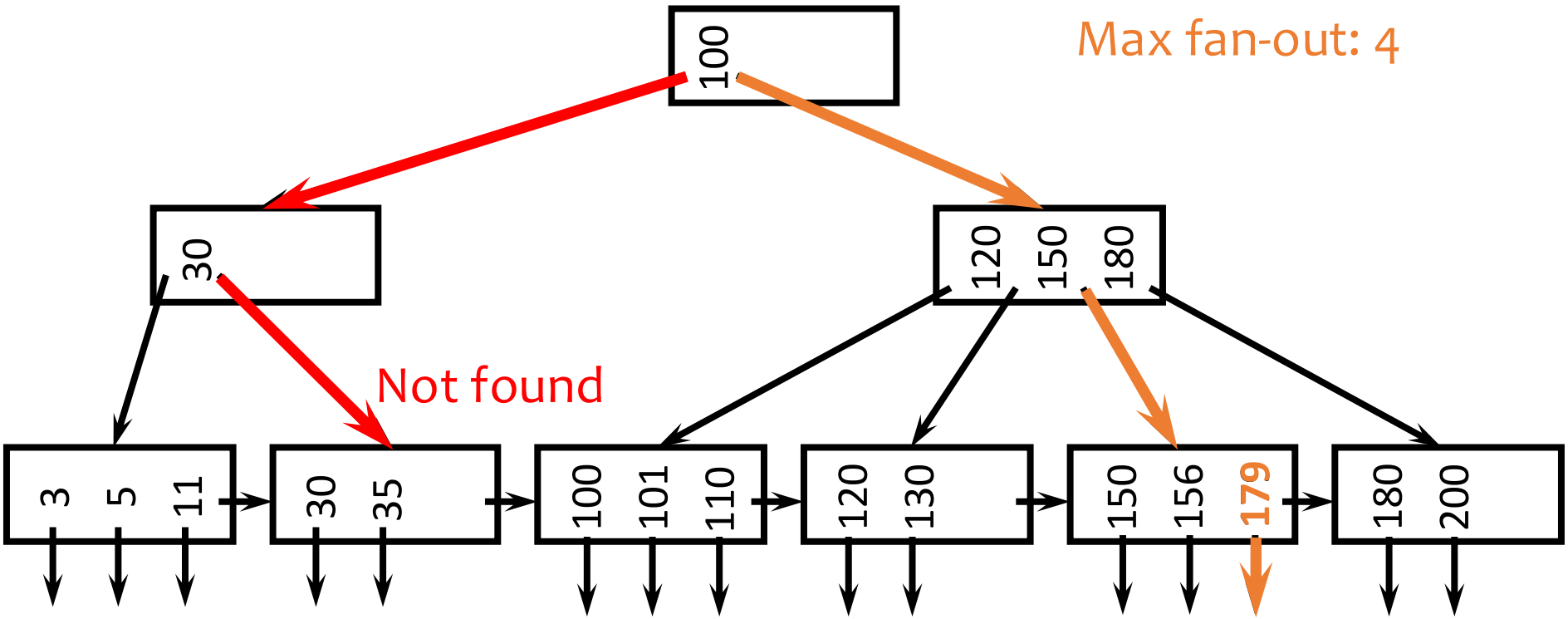
- Height (h) constraint: all leaves at the same lowest level
- Fan-out (f) constraint: all nodes at least half full (except root)

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

minimum	$2 \cdot \lceil f/2 \rceil^{h-2} \cdot \lfloor f/2 \rfloor \approx 2 \cdot \lceil f/2 \rceil^{h-1}$
maximum	$f^{h-1} \cdot (f-1) \approx f^{h}$

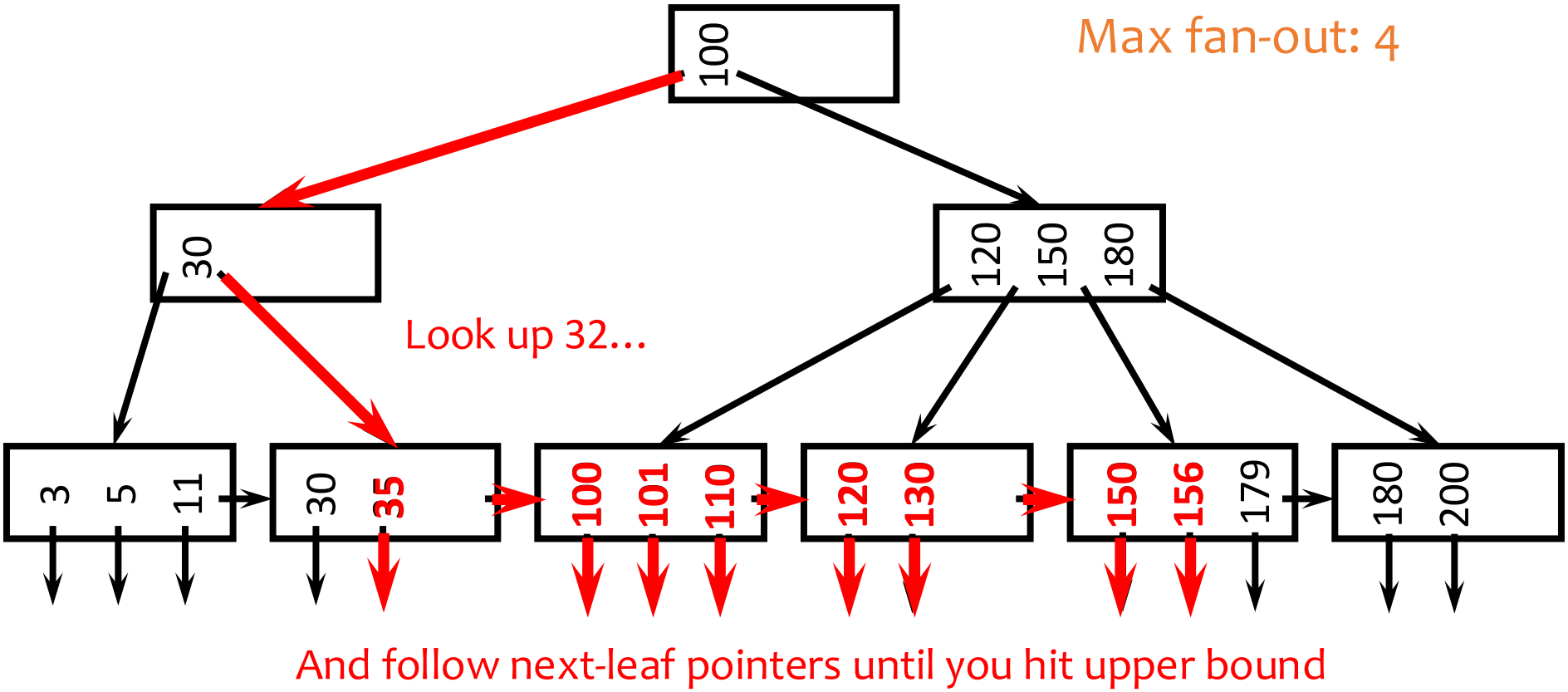
Lookups

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



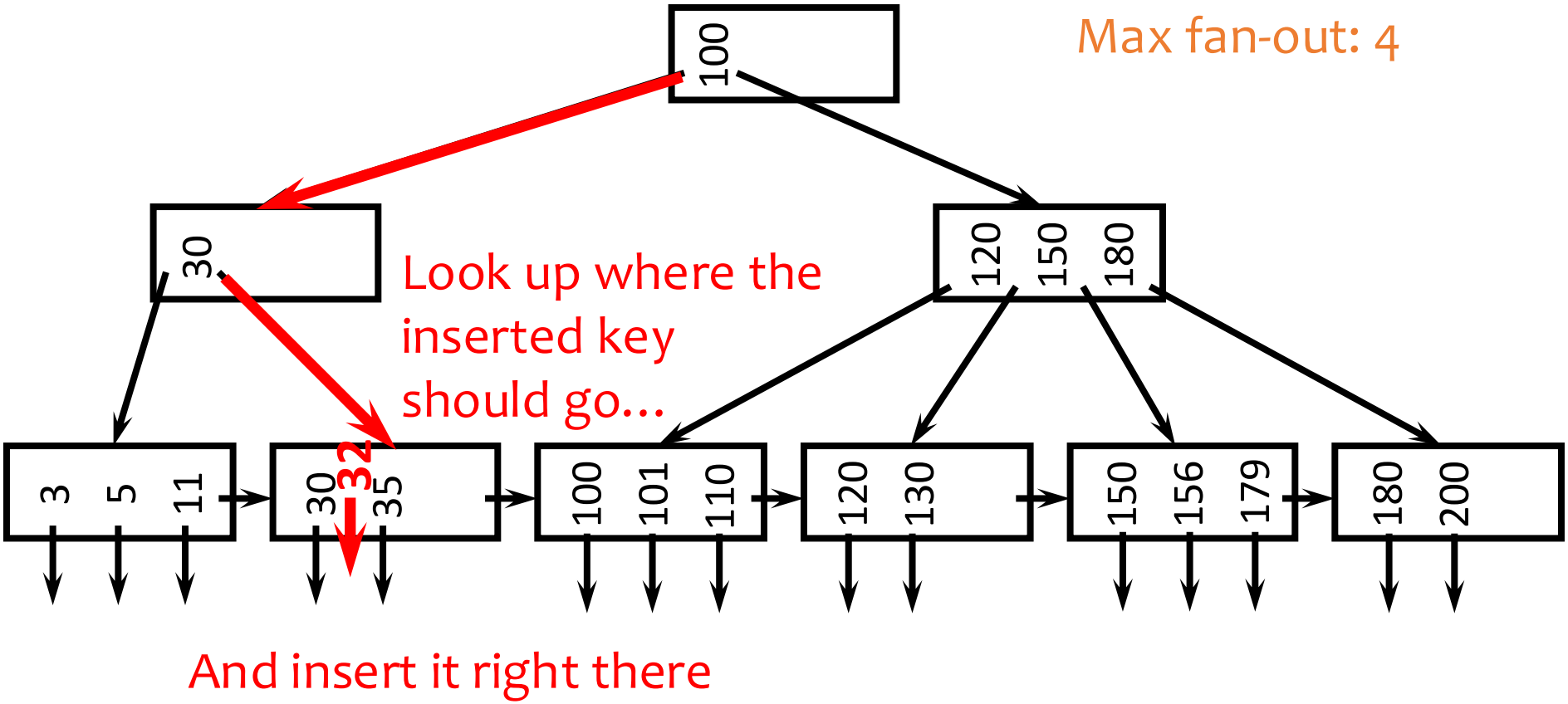
Range query

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



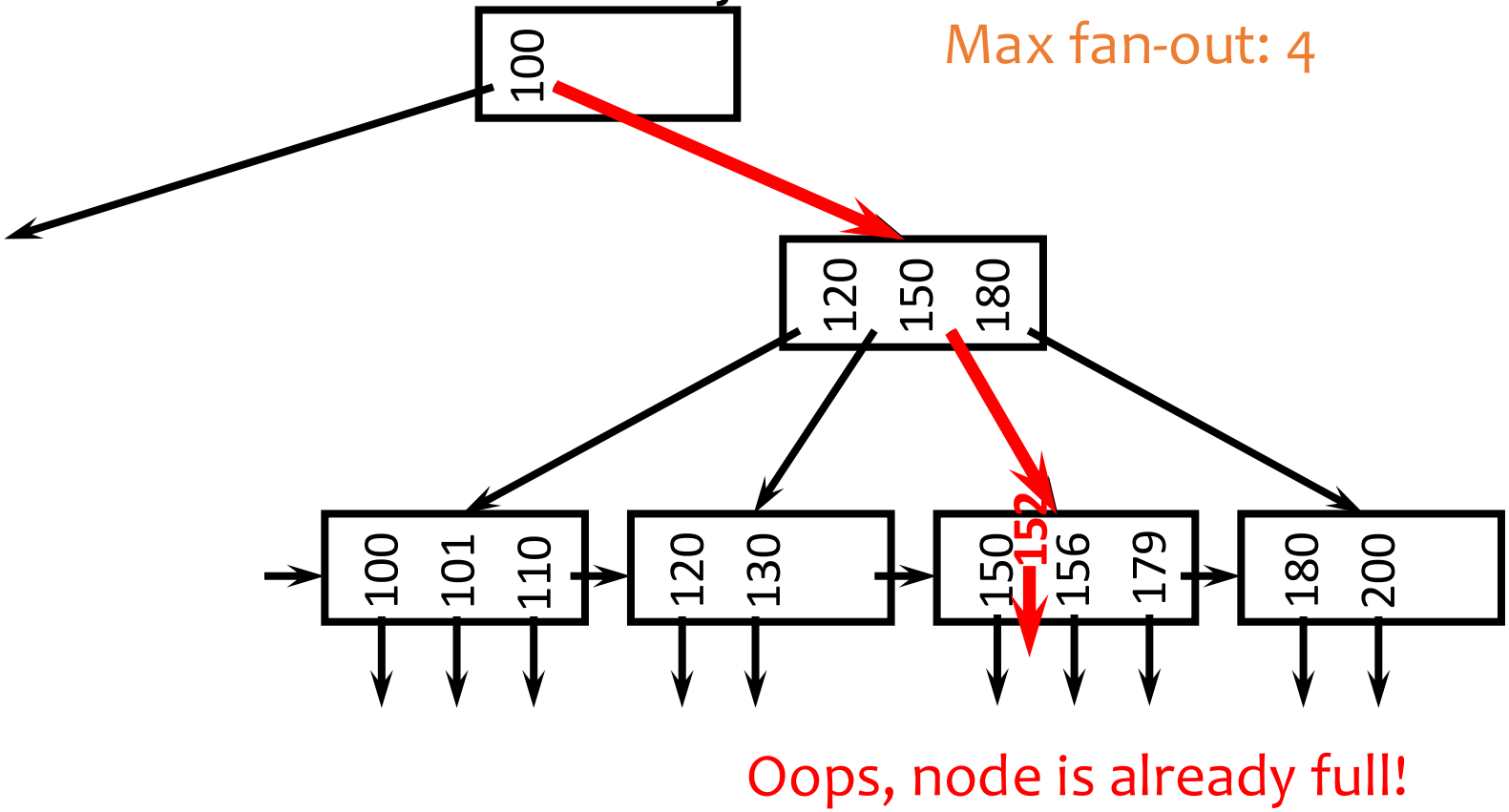
Insertion

- Insert a record with search key value 32

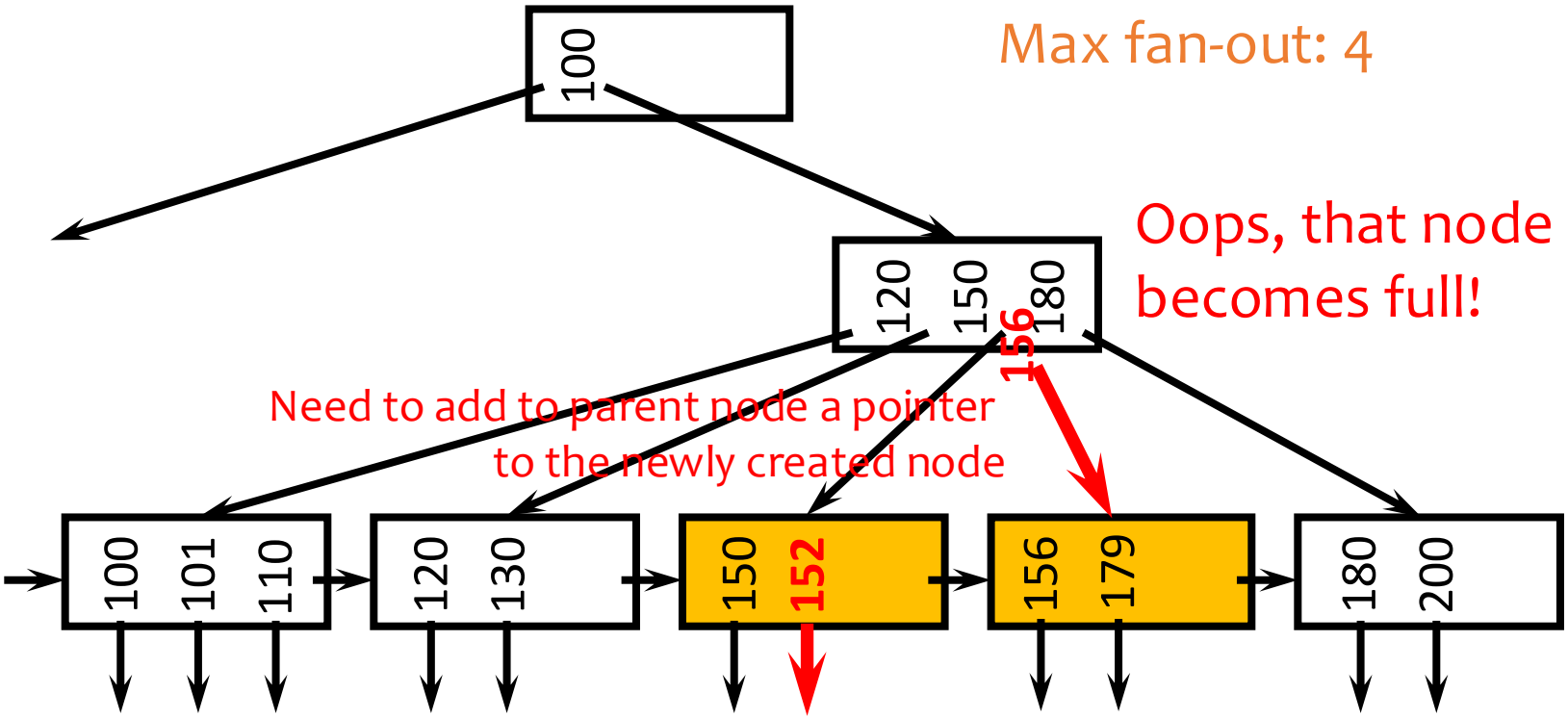


Another insertion example

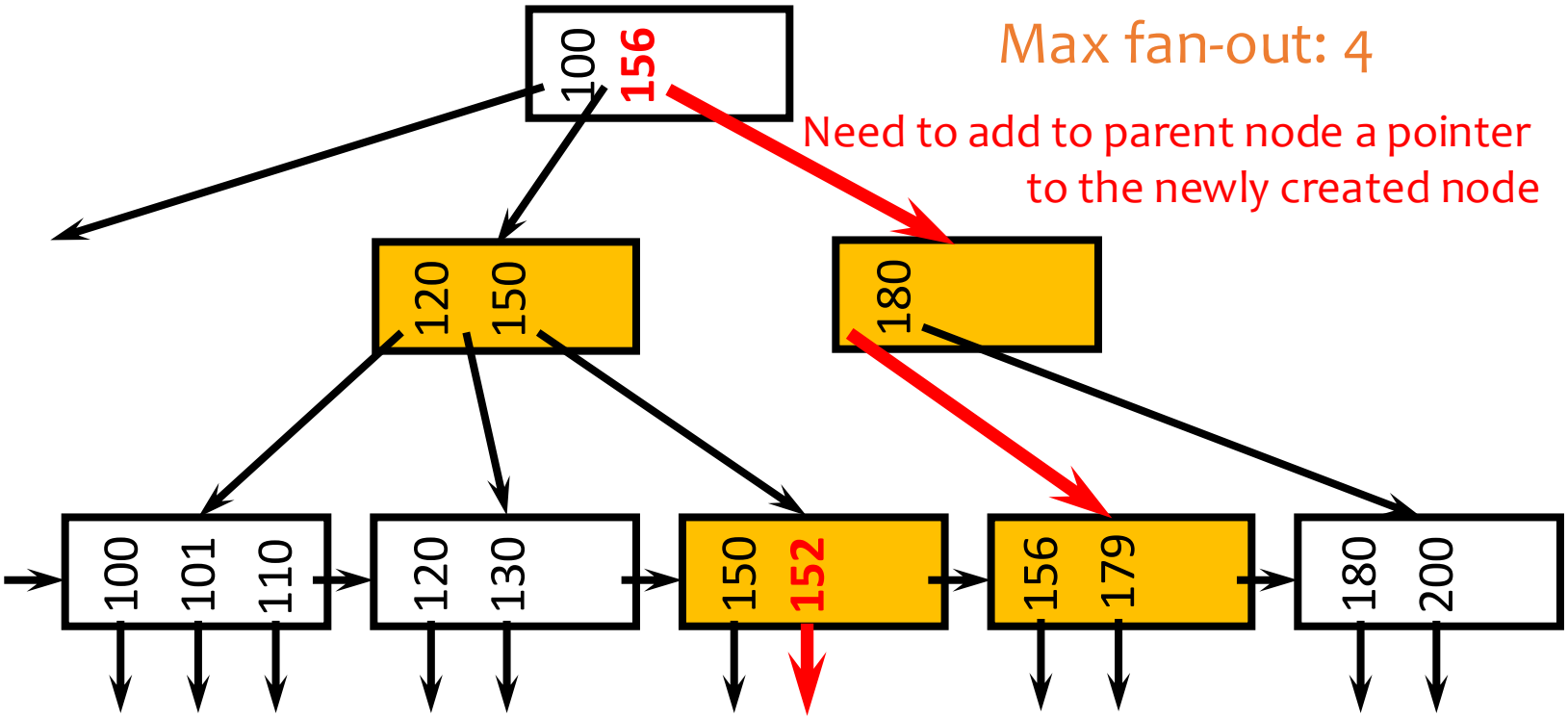
- Insert a record with search key value 152



Node splitting



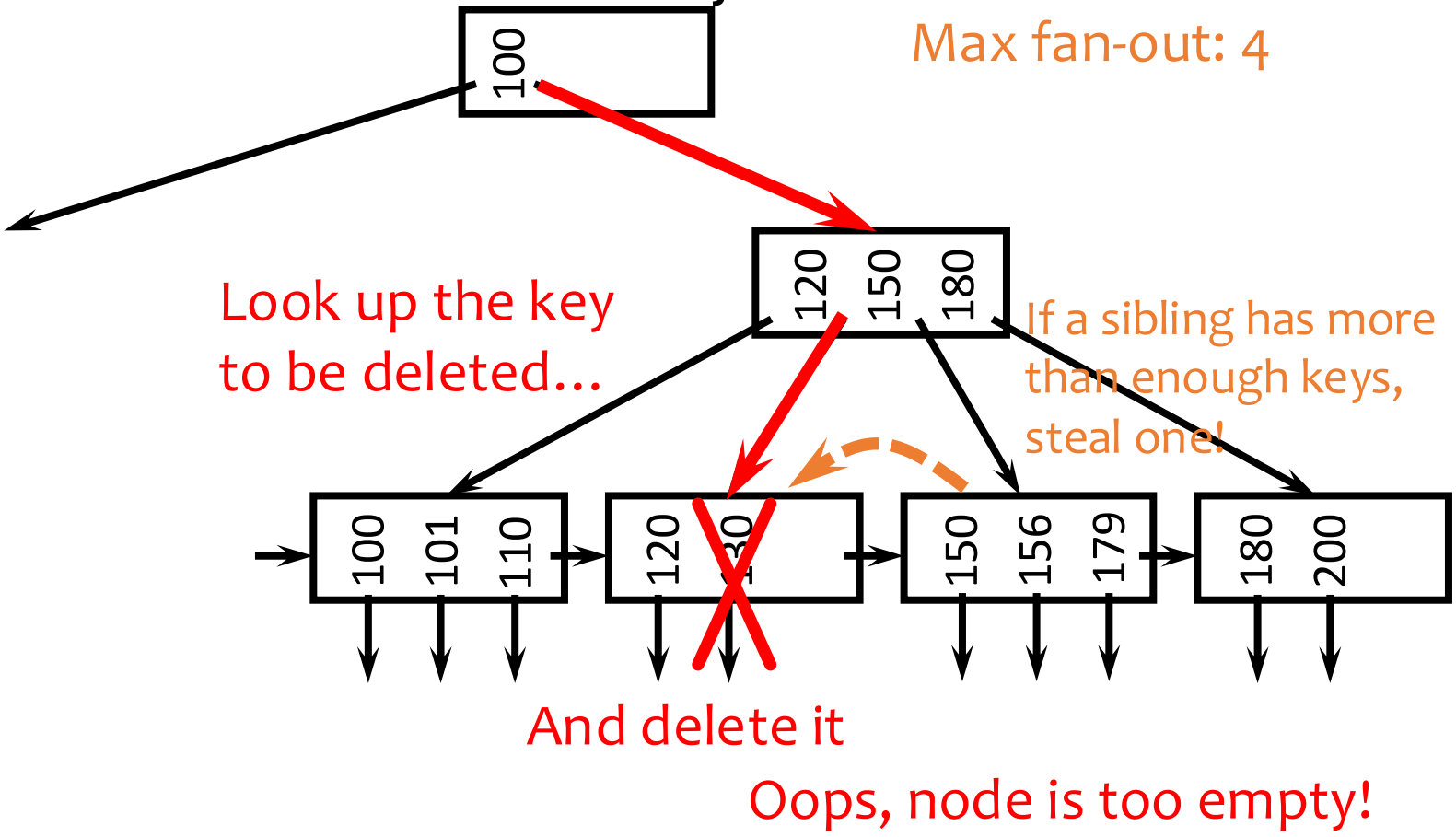
More node splitting



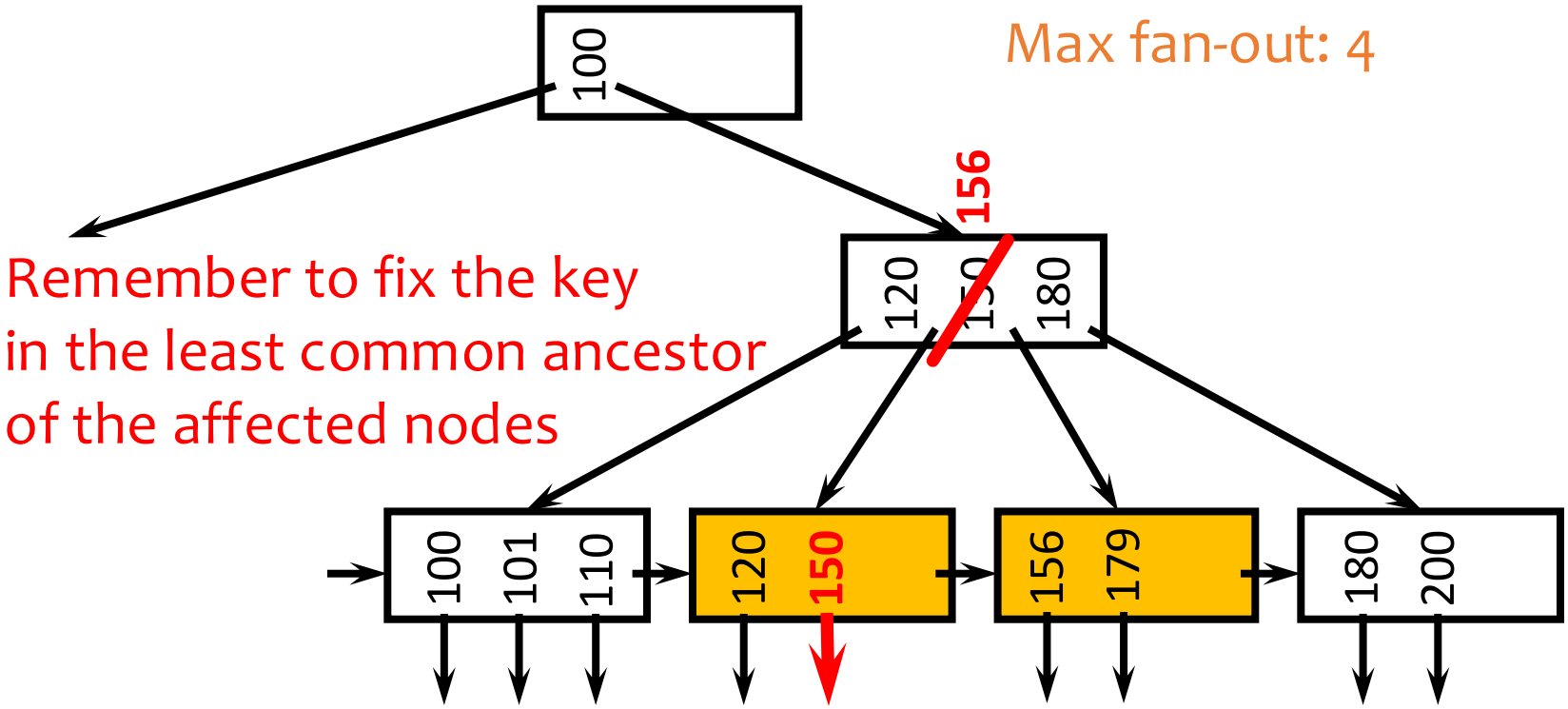
- In the worst case, node splitting can "propagate" all the way up to the root of the tree (not illustrated here)
  - Splitting the root introduces a new root of fan-out 2 and causes the tree to grow "up" by one level

Deletion

- Delete a record with search key value 130

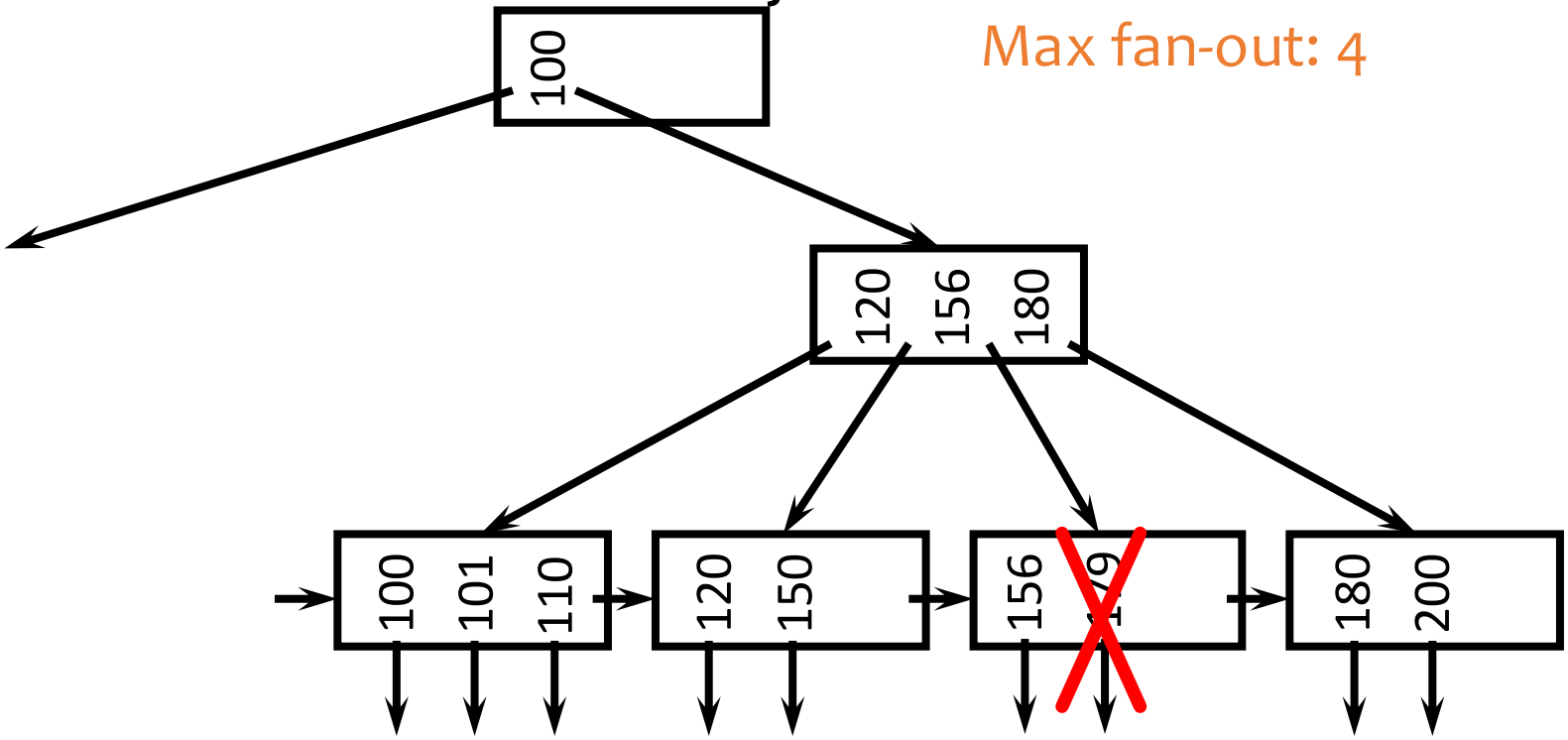


Stealing from a sibling

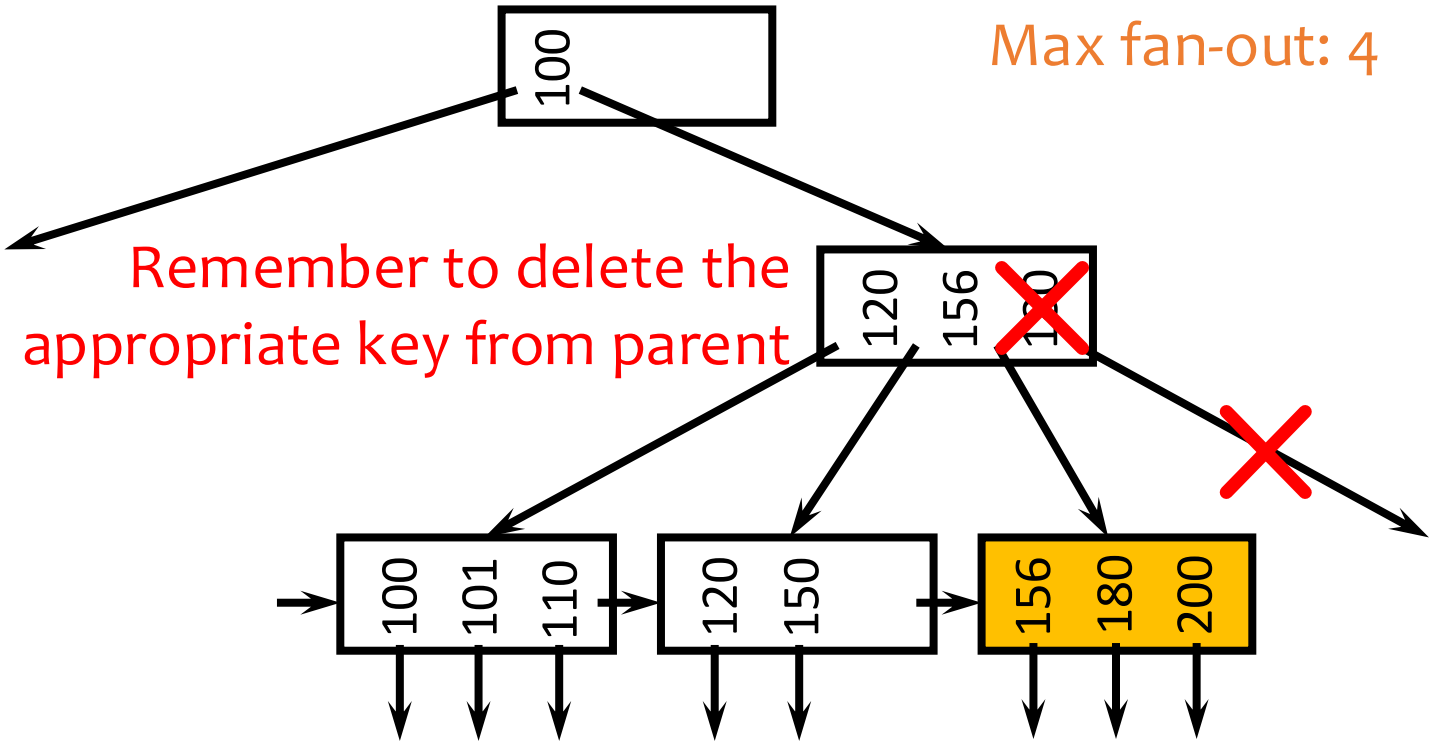


Another deletion example

- Delete a record with search key value 179



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

In class exercises

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)

- 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: can store up to *f^level* number of keys (typical fanout is 200)

In class ex.

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

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
  - Range queries can become less efficient
  - Records in leaves require more I/O's to access
  - Vast majority of the records live in leaves!

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 a WHERE clause contains several conditions; or it enables index-only plans
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(uid, name, age, pop)
- Group(gid, name, date)
- Member(uid, gid)

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

- 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 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 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

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(<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 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(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 → 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>)

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

A clustering index on Group(date)

(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
  - Hash based
  - Layered
- How to use index
  - Use multi-attribute indices
  - Index-only plan
  - General guideline