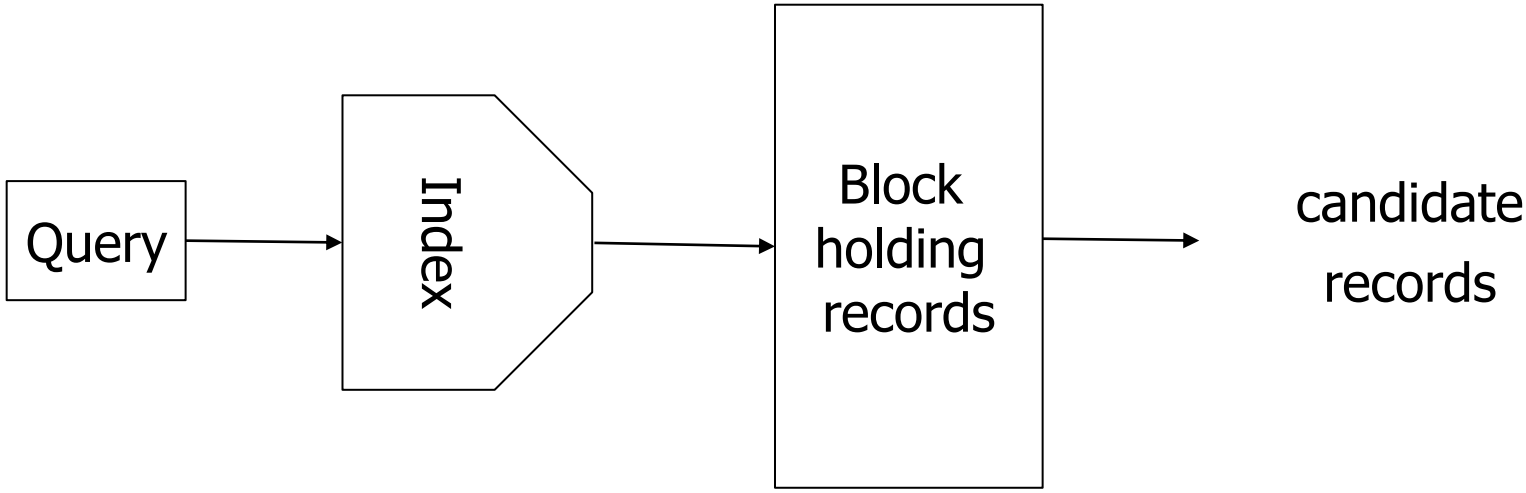
Indexing

Hector Garcia-Molina
Mahmoud Sakr

Indexing



Topics

- Conventional indexes
- B-trees
- Hashing schemes

Sequential File

10	
20	

30	
40	

50	
60	

70	
80	

90	
100	

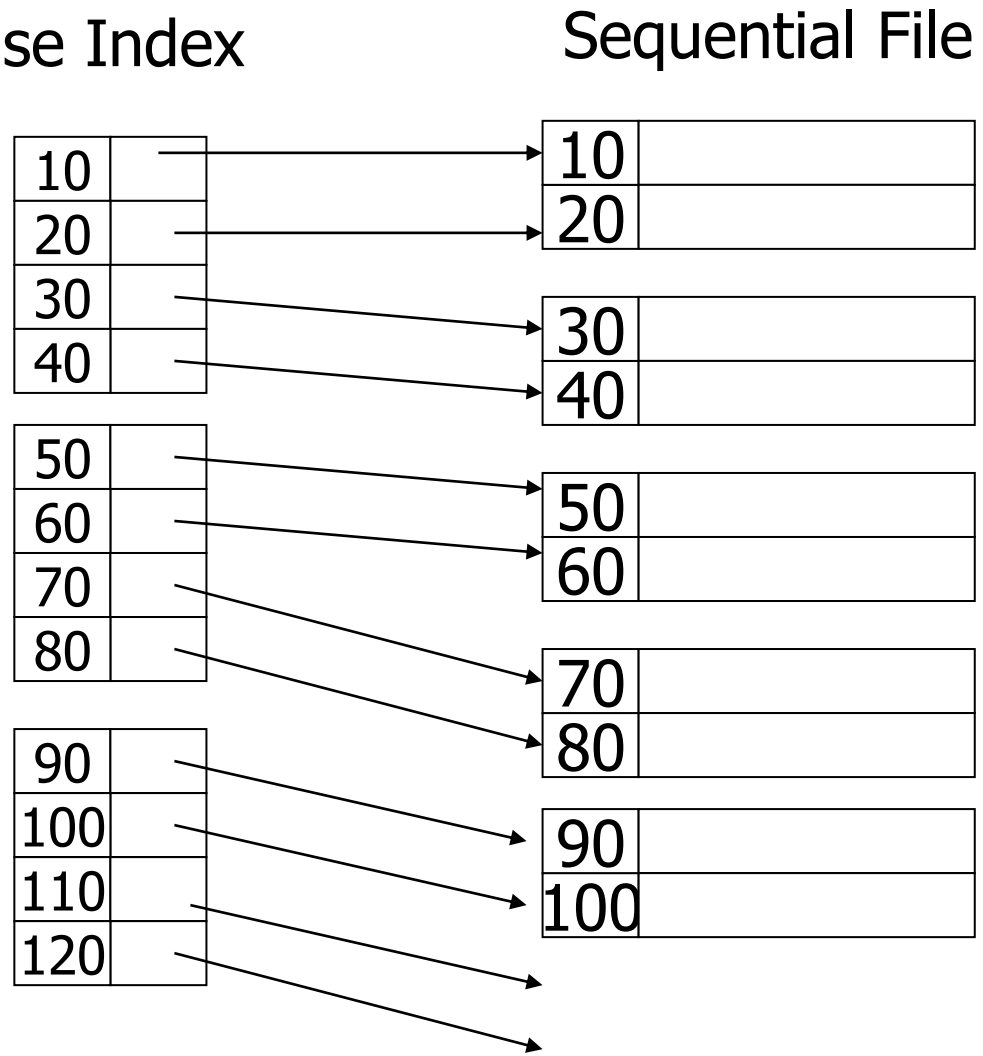
Dense Index = a pointer per key

How to search for a key= 30 ?

How to search for a key= 25 ?

Can we use a dense index on a non-sequential file ? yes

Why querying a dense index is more efficient than querying the sequential file ? memory / disk



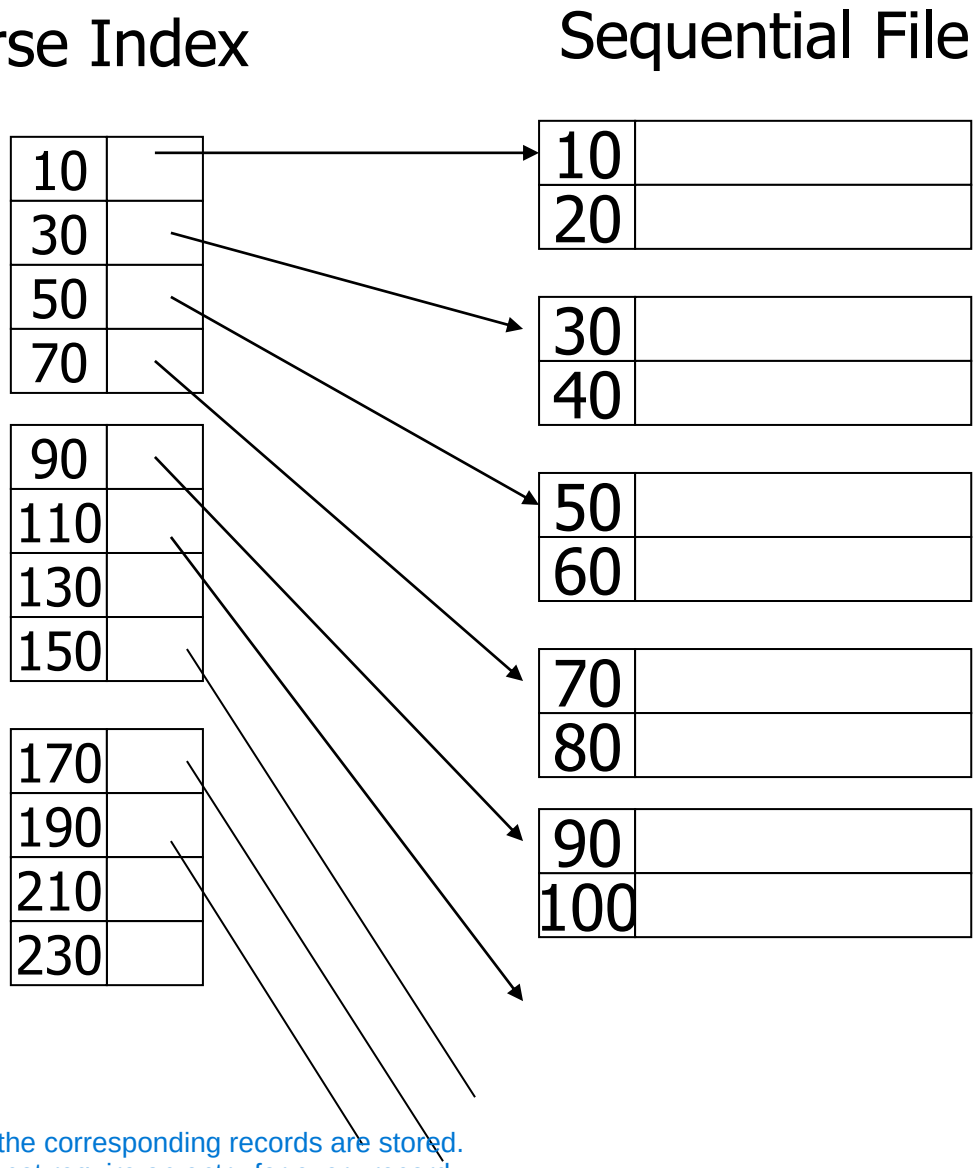
In a dense index, an entry exists in the index for every record in the table, and the index entries are sorted based on the indexed column(s).
helps the database system quickly navigate to the desired record when a search operation is performed based on the indexed column(s).
The dense index is particularly useful for range queries or queries involving comparison operators (e.g., greater than, less than) on the indexed column(s).
However, because it requires additional storage space for the index entries and may need maintenance during data modifications (inserts, updates, deletes)

Sparse index = a pointer per block

How to search for a key= 30 ?

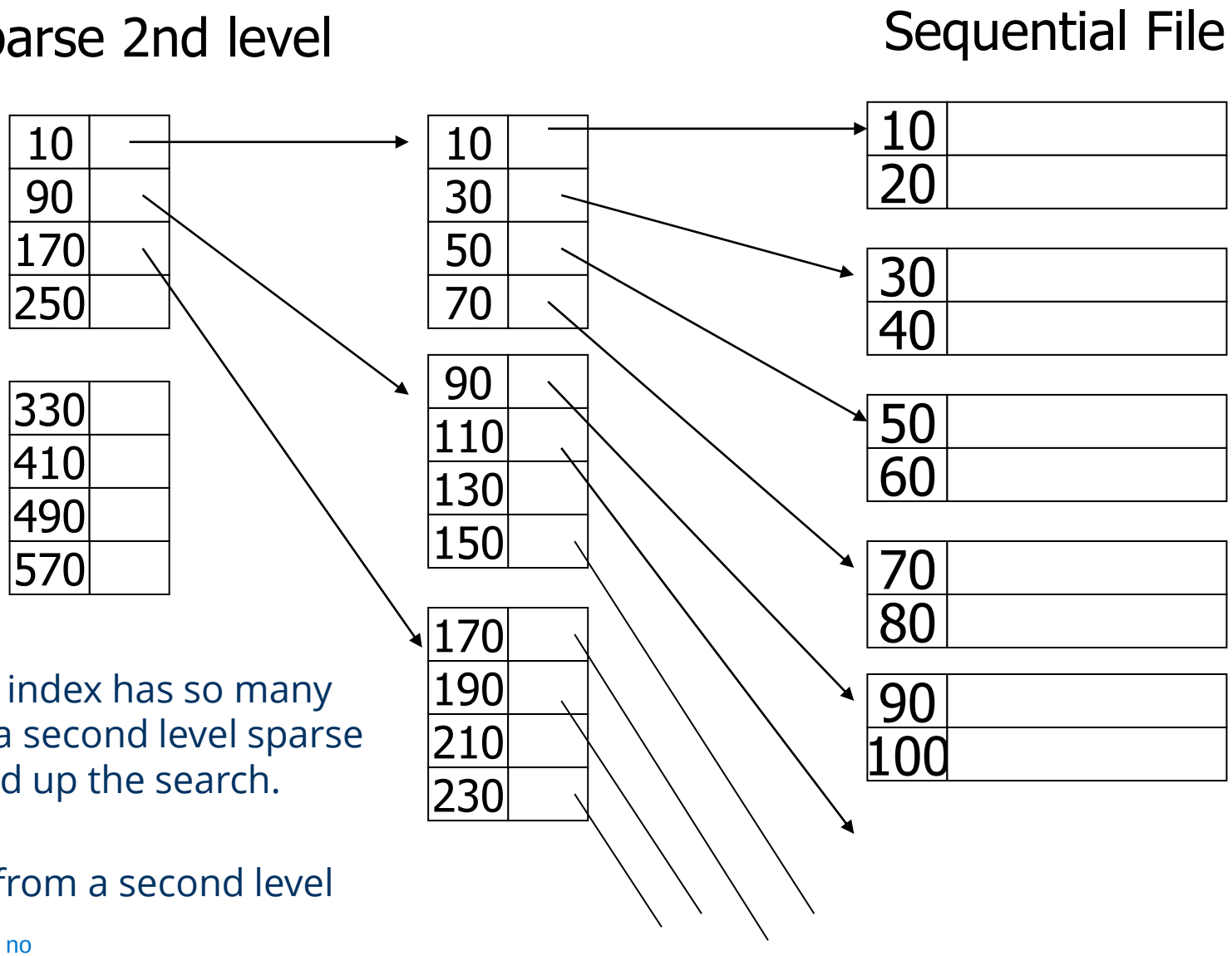
How to search for a key= 25 ?

Can we use a sparse index on a non-sequential file ? No



In a sparse index, the index entries point to specific locations in the table where the corresponding records are stored.
This makes sparse indexes more compact than dense indexes because they do not require an entry for every record.
However, it may take more time to locate a specific record if it does not have an entry in the index.

Sparse indexes are often used when the indexed column has a large number of distinct values, and creating an entry for every record would be impractical in terms of storage and maintenance.
They are efficient for certain types of queries but might not perform as well as dense indexes for range queries or queries involving comparison operators on the indexed column



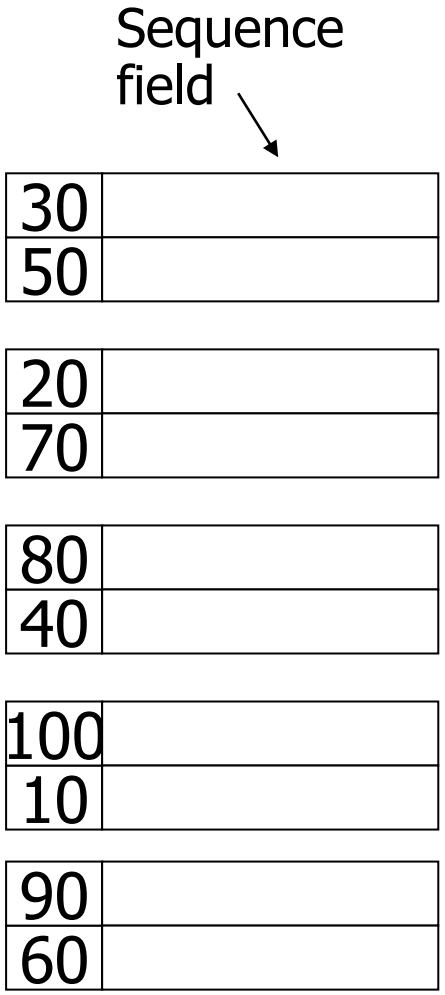
If the first level index has so many pages, adding a second level sparse index can speed up the search.

Do we benefit from a second level dense index ? no

Sparse vs. Dense Tradeoff

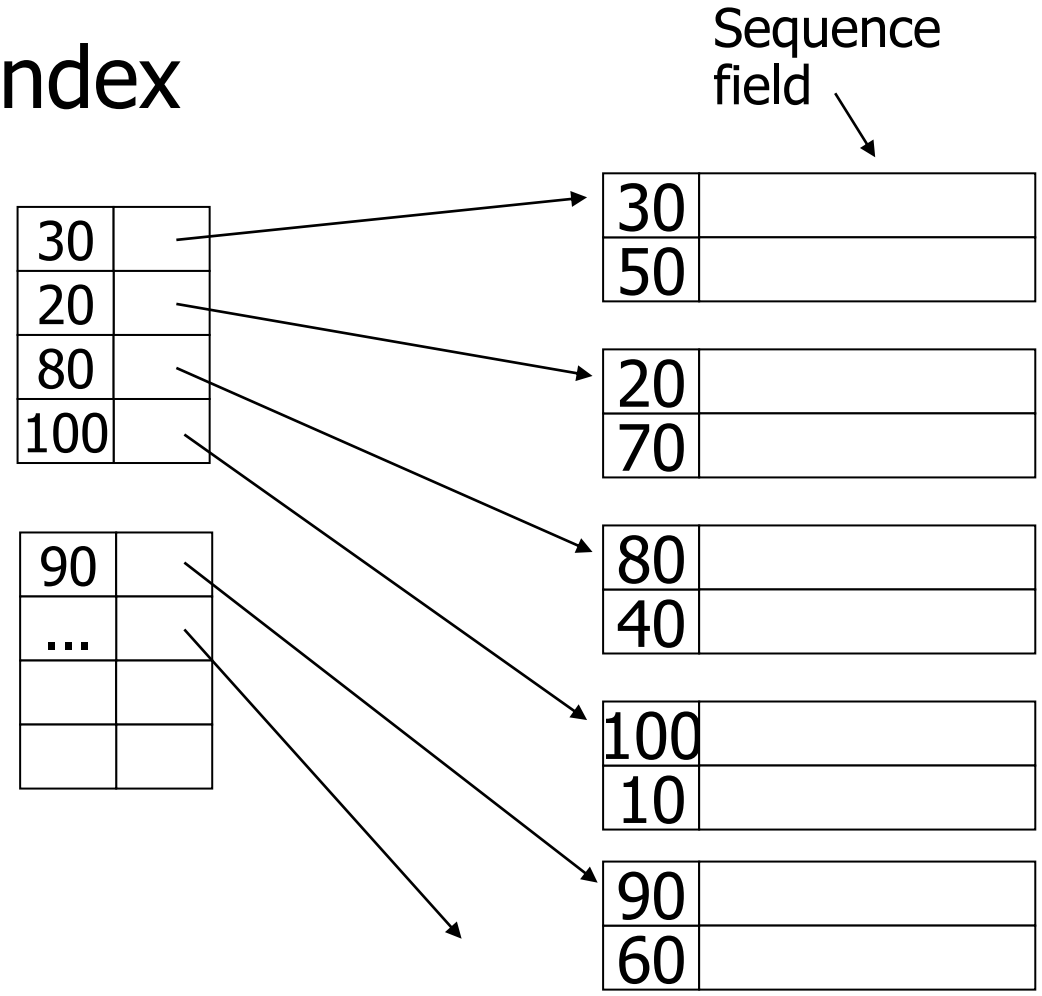
- Sparse: Less index space per record
  can keep more of index in memory
- Dense: Can tell if any record exists
  without accessing file

Secondary indexes



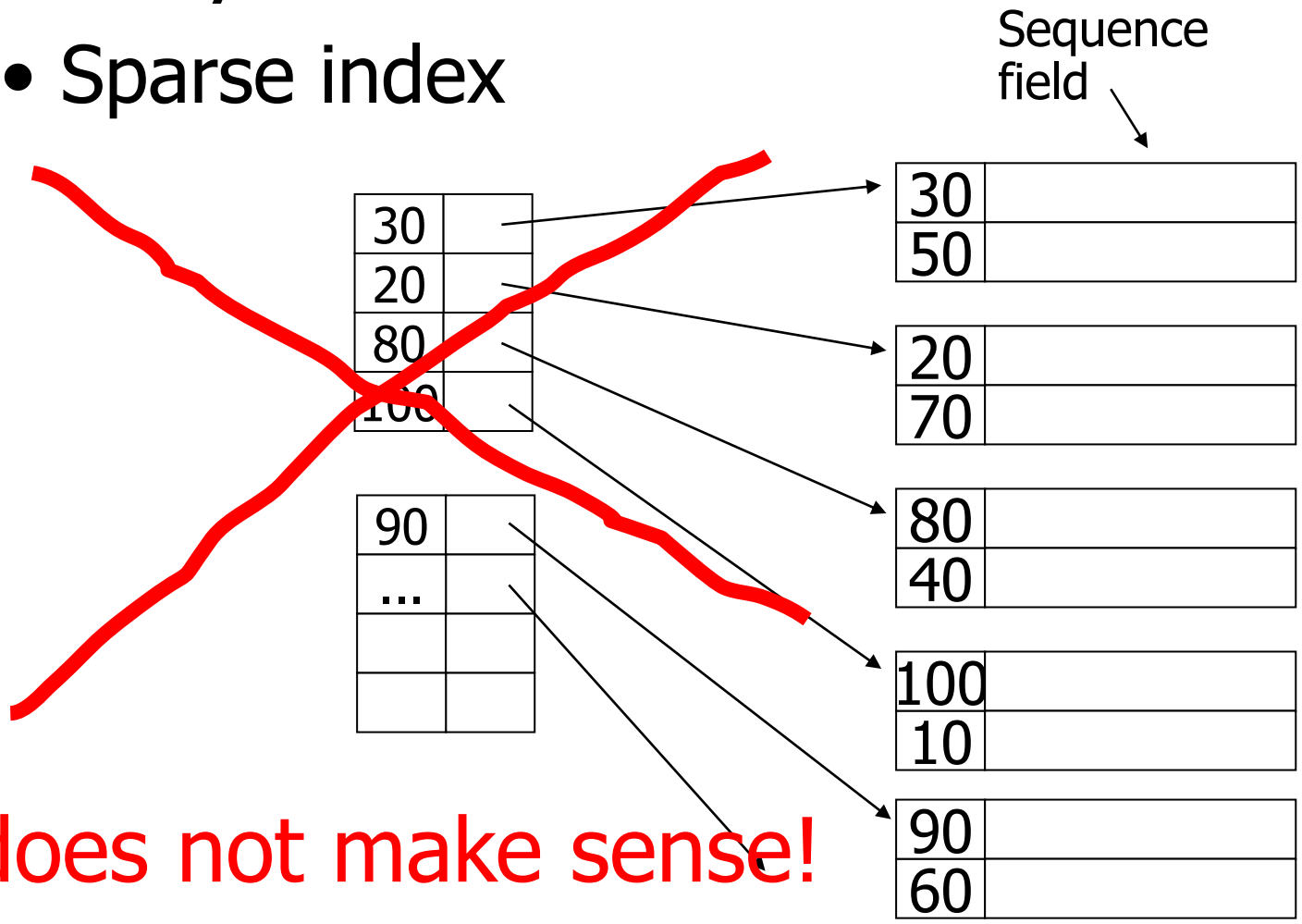
Secondary indexes

- Sparse index



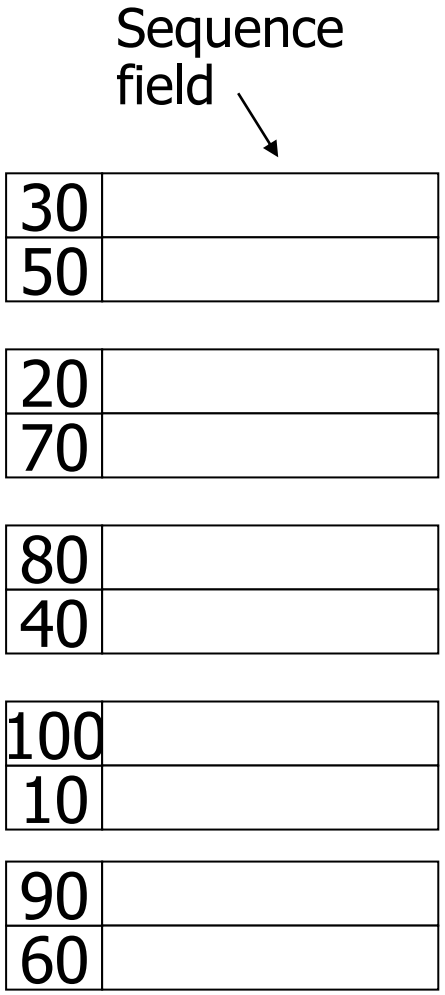
Secondary indexes

- Sparse index



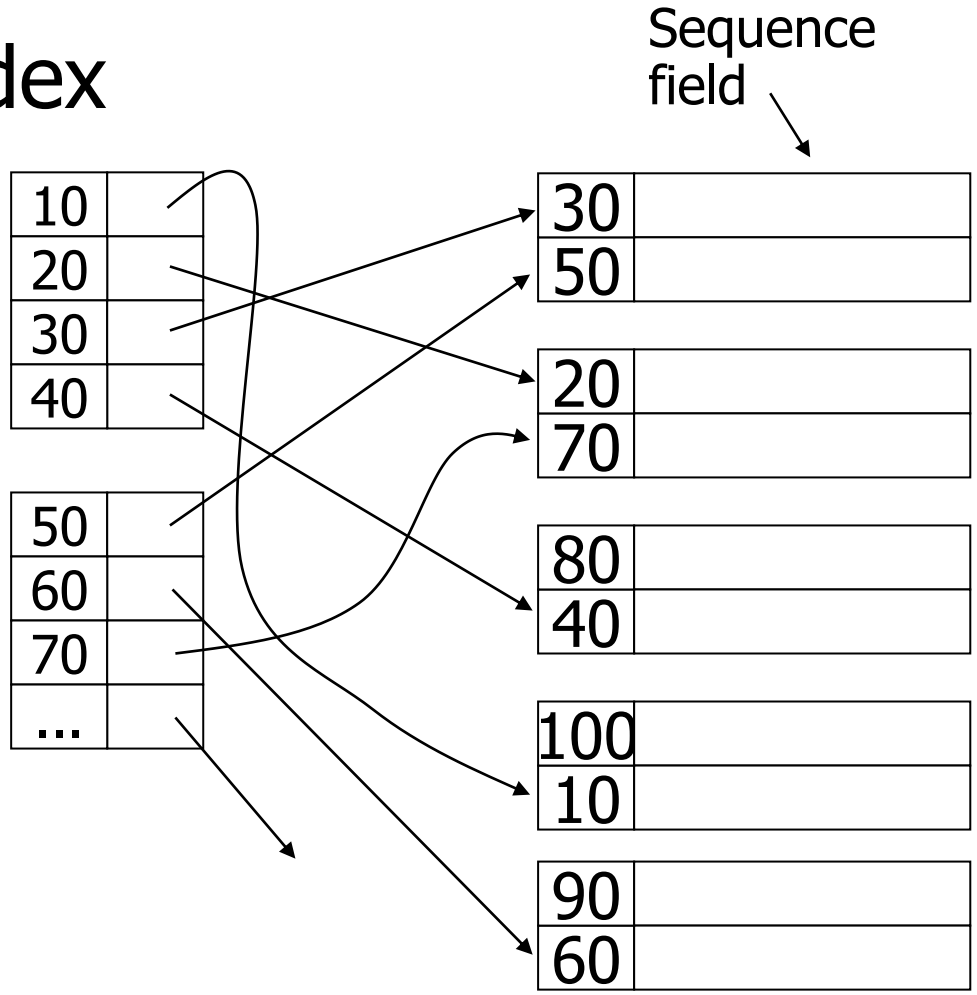
Secondary indexes

- Dense index



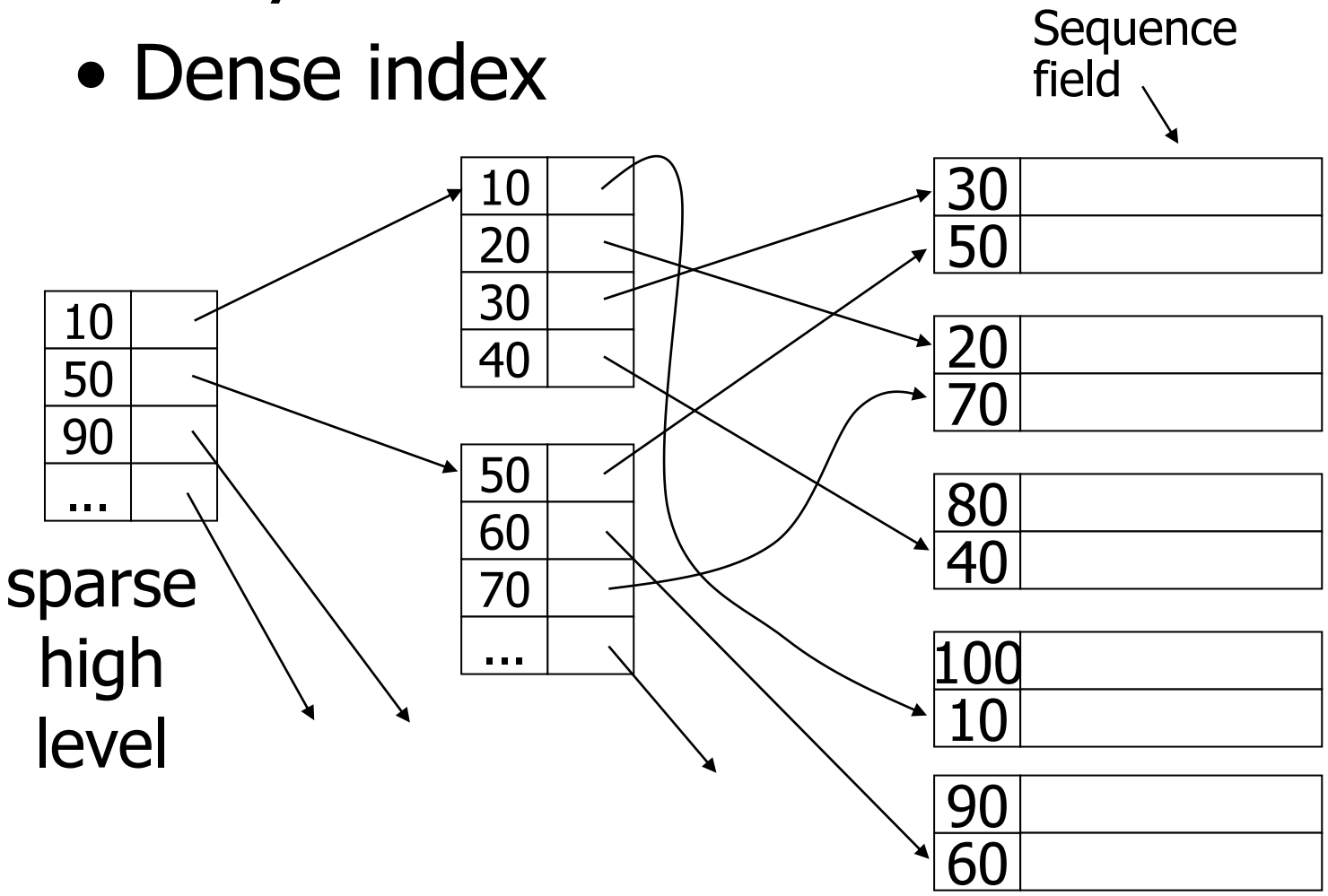
Secondary indexes

- Dense index



Secondary indexes

- Dense index



With secondary indexes:

- Lowest level is dense
- Other levels are sparse

Duplicate values & secondary indexes

20	
10	

20	
40	

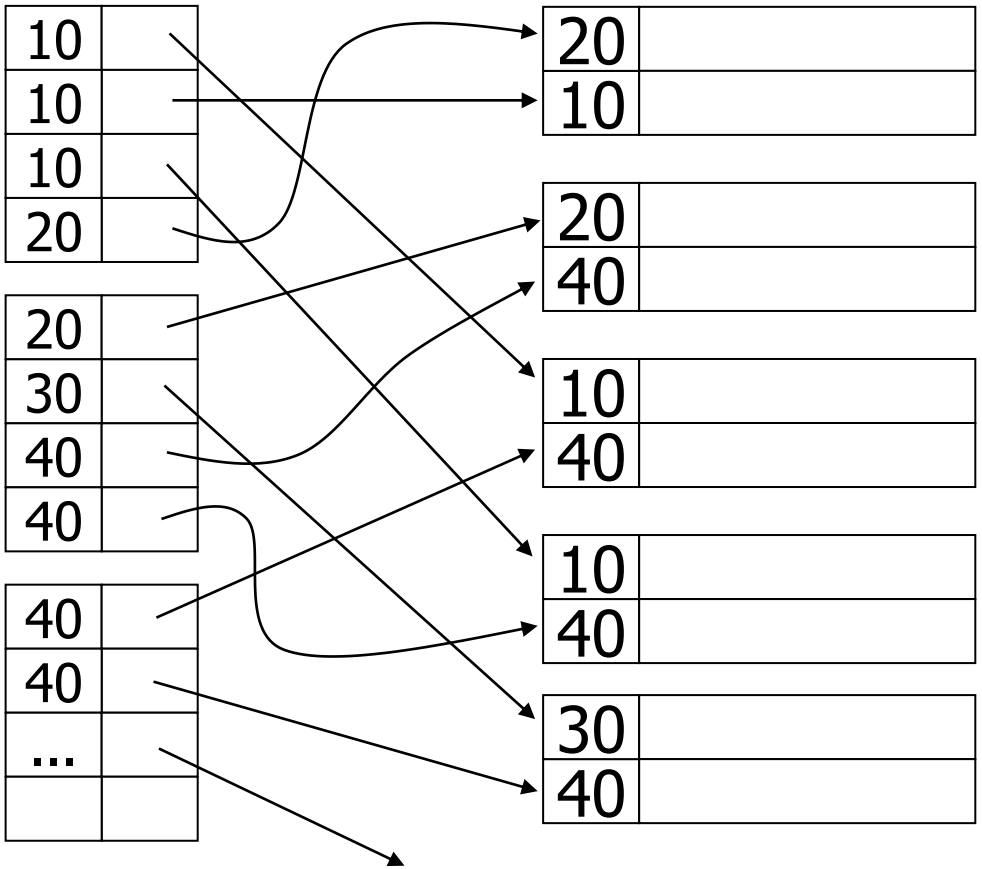
10	
40	

10	
40	

30	
40	

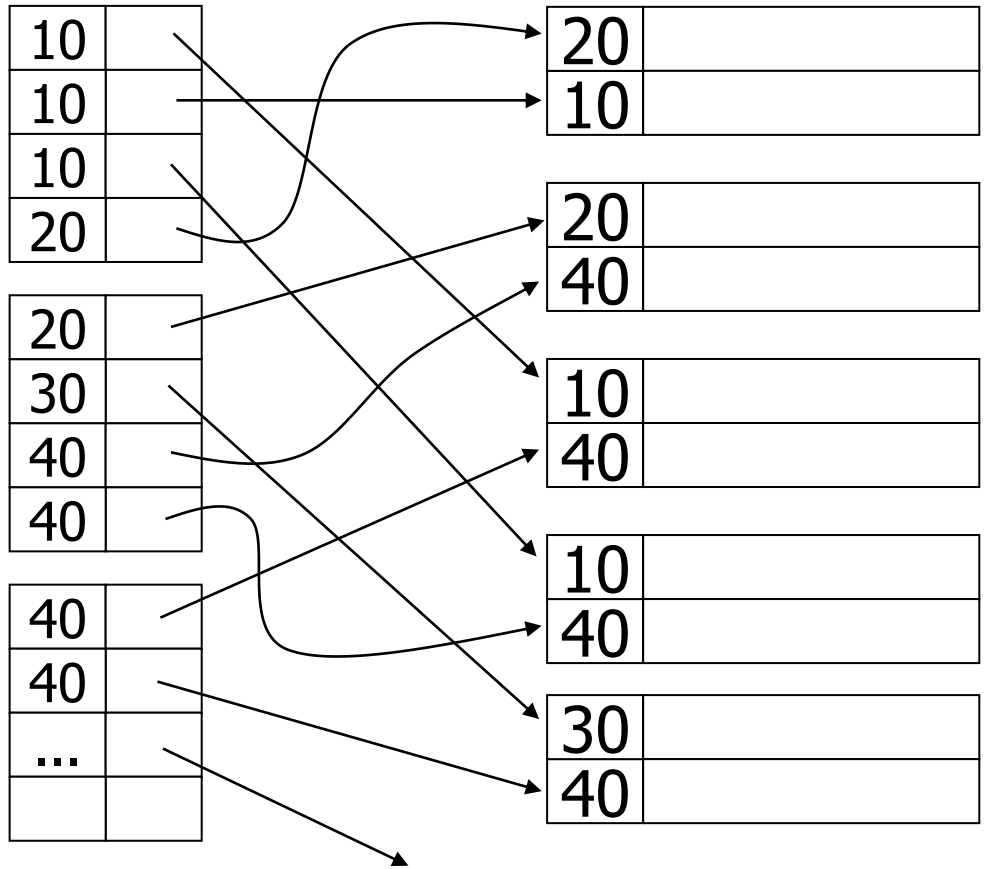
Duplicate values & secondary indexes

one option...



Duplicate values & secondary indexes

one option...



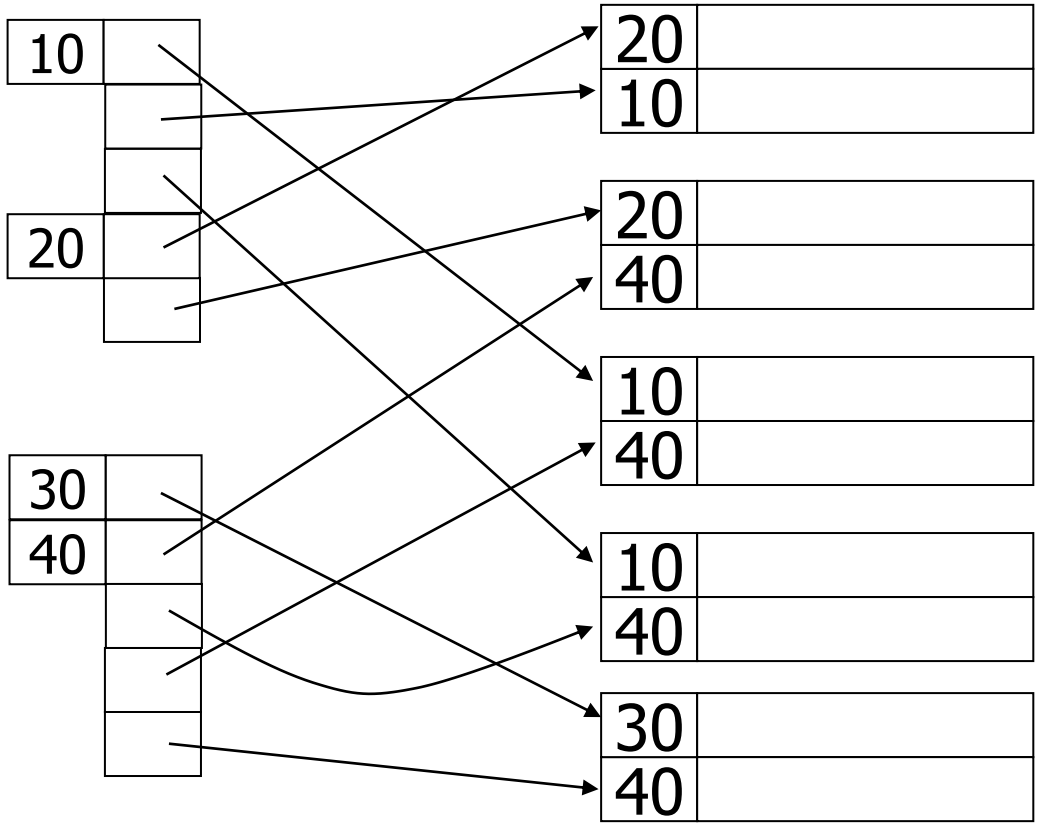
Problem:

excess overhead!

- disk space
- search time

Duplicate values & secondary indexes

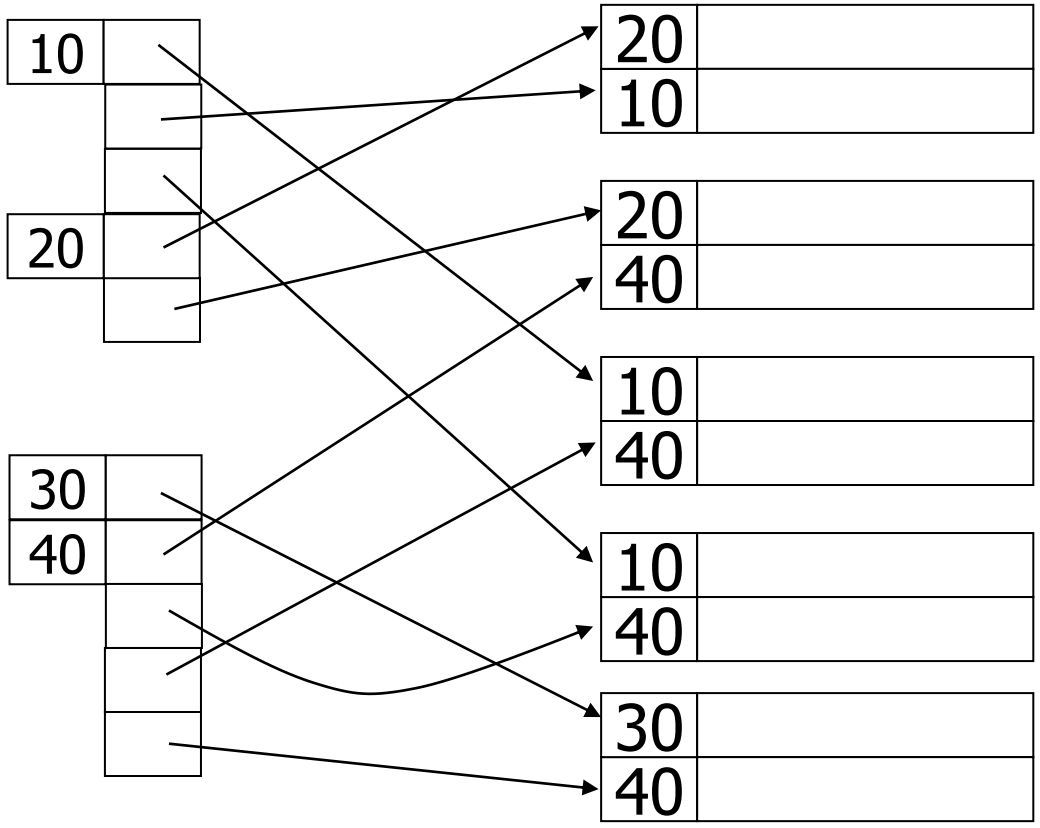
another option...



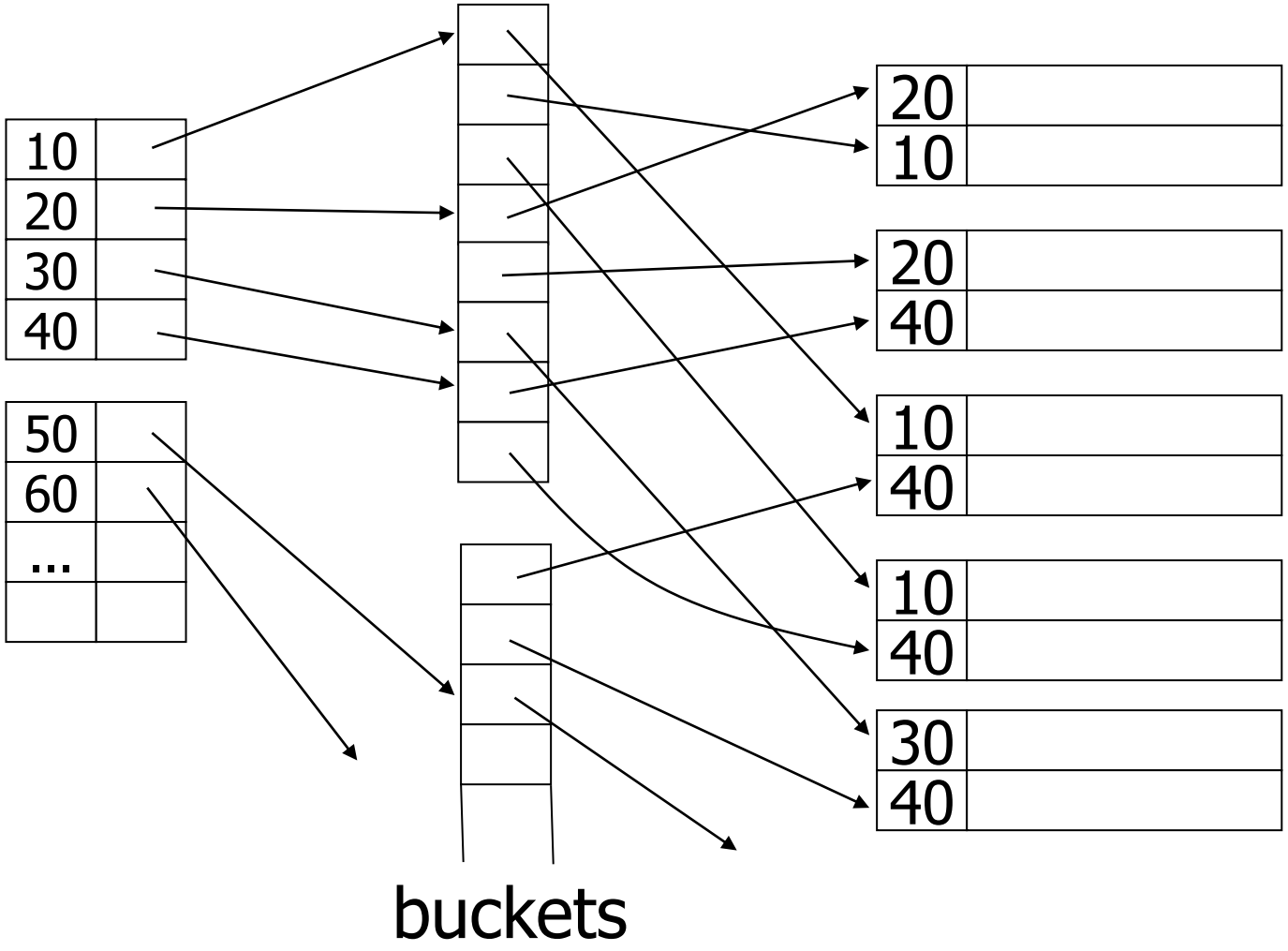
Duplicate values & secondary indexes

another option...

Problem:
variable size
records in
index!



Duplicate values & secondary indexes



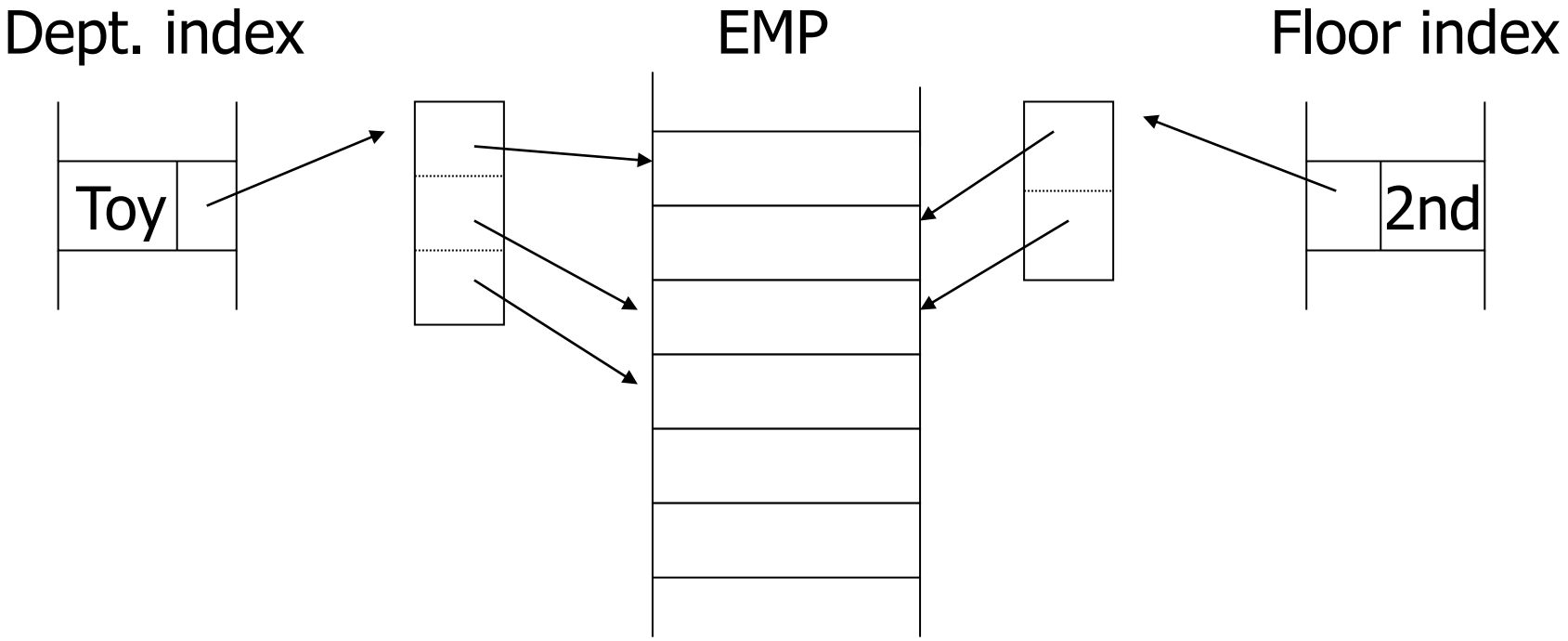
Index is fixed size

2+ number of repetition

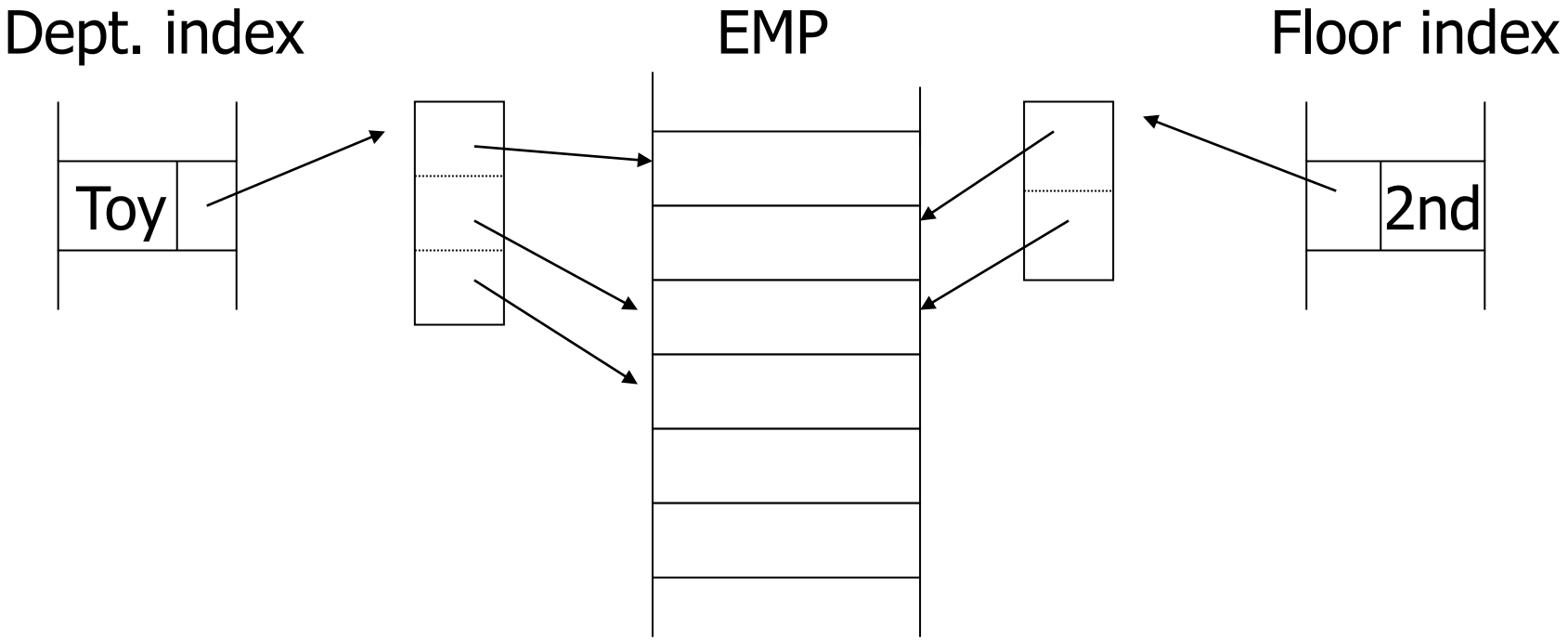
Why "bucket" idea is useful

Indexes	Records
Name: primary	EMP (name,dept,floor,...)
Dept: secondary	
Floor: secondary	

Query: Get employees in (Toy Dept) ∧ (2nd floor)



Query: Get employees in
(Toy Dept) $\wedge$ (2nd floor)



$\rightarrow$ Intersect toy bucket and 2nd Floor bucket to get set of matching EMP's

Conventional indexes

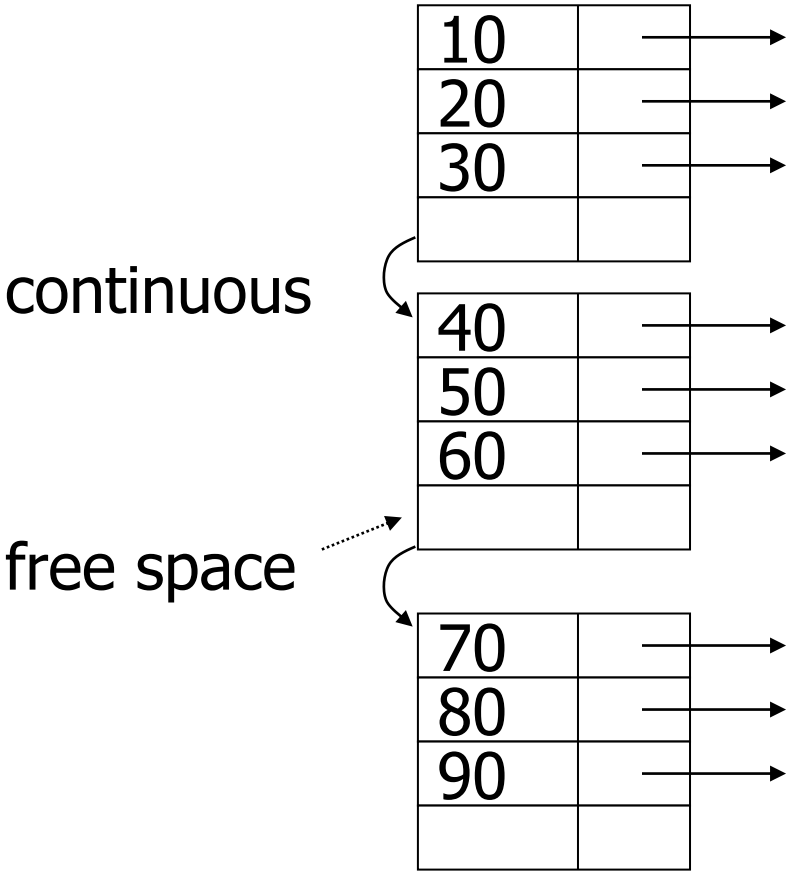
Advantage:

- Simple
- Index is sequential file
  good for scans

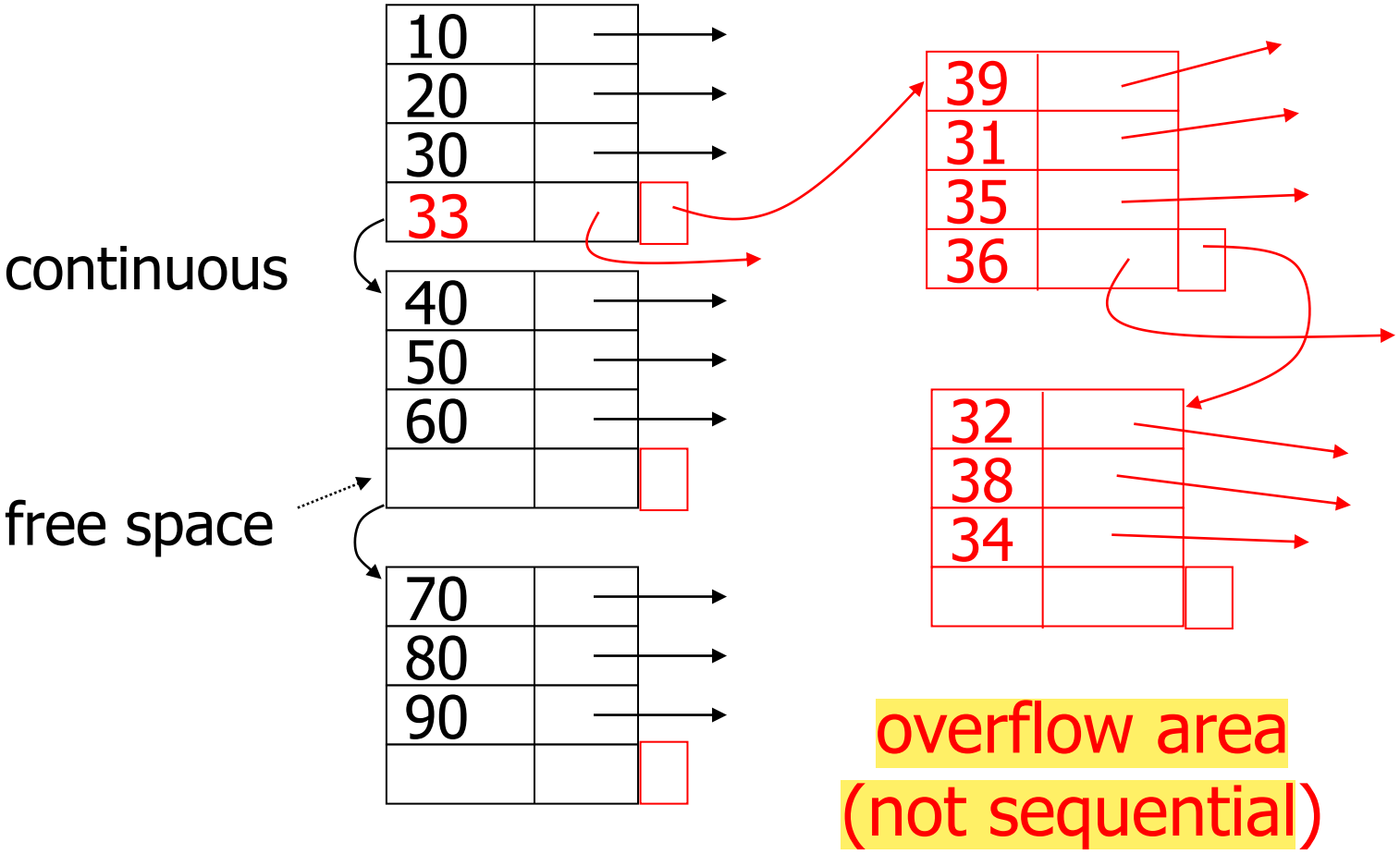
Disadvantage:

- Inserts expensive, or
- Lose sequentiality & balance

Example Index (sequential)



Example Index (sequential)

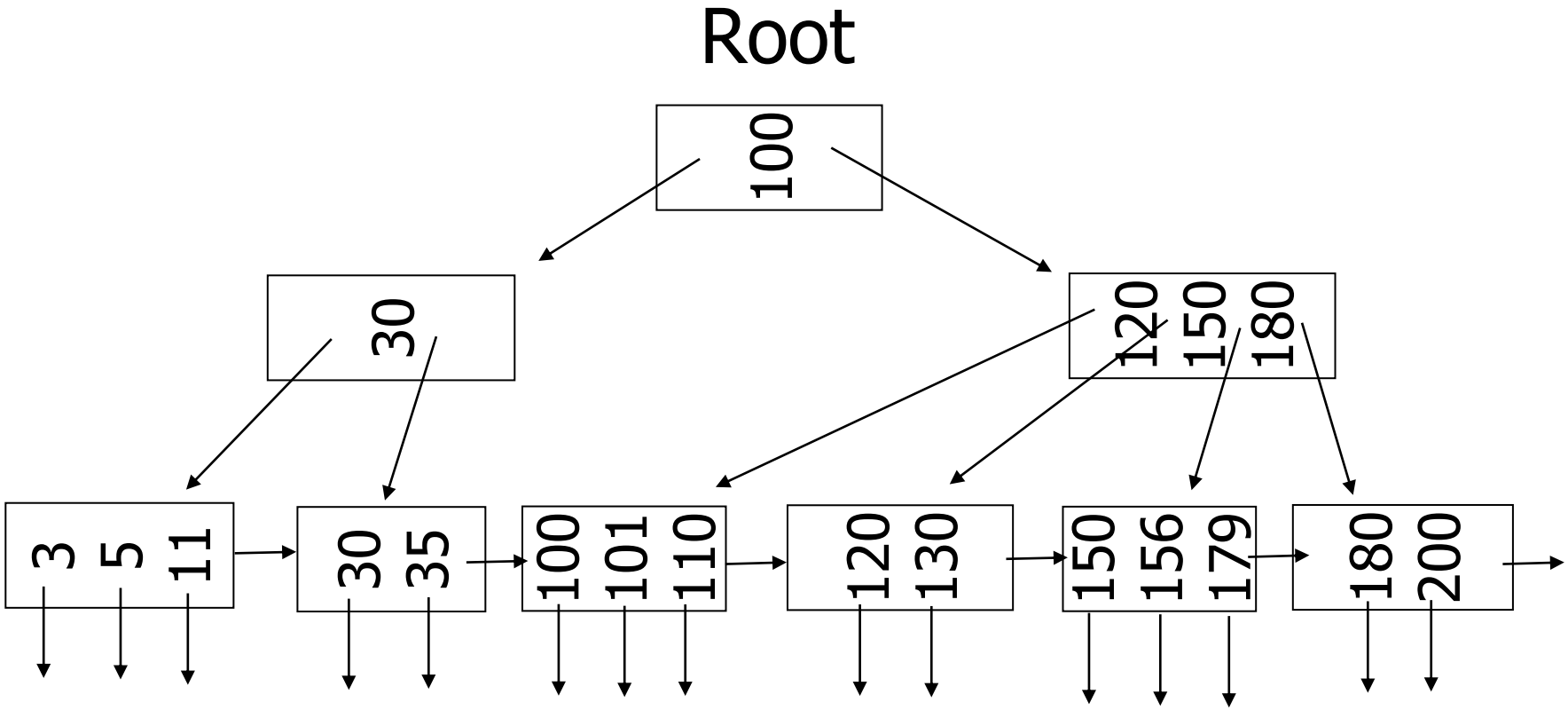


Outline:

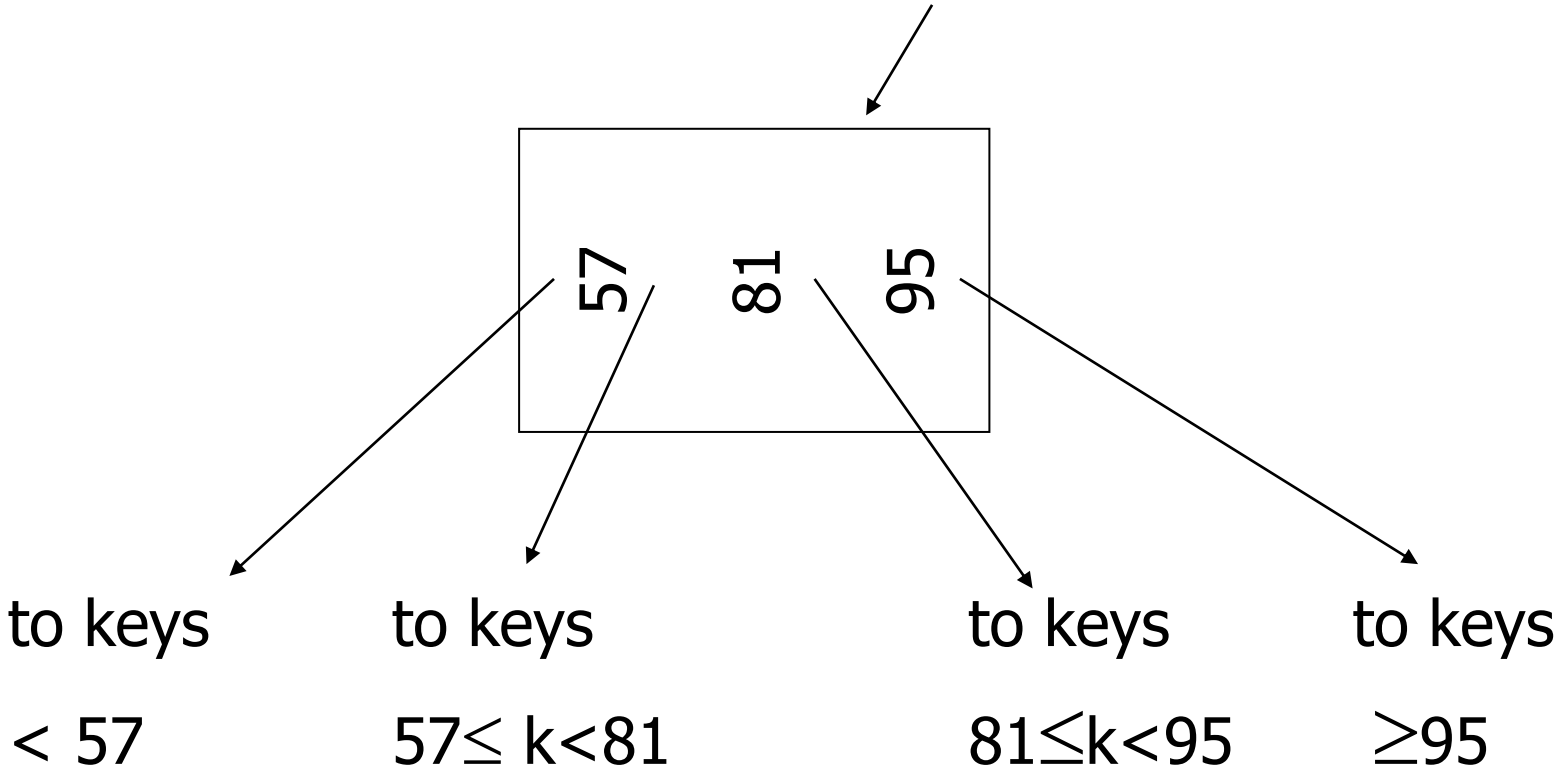
- Conventional indexes
- B-Trees $\Rightarrow$ NEXT
- Hashing schemes

- NEXT: Another type of index
  - Give up on sequentiality of index
  - Try to get “balance”

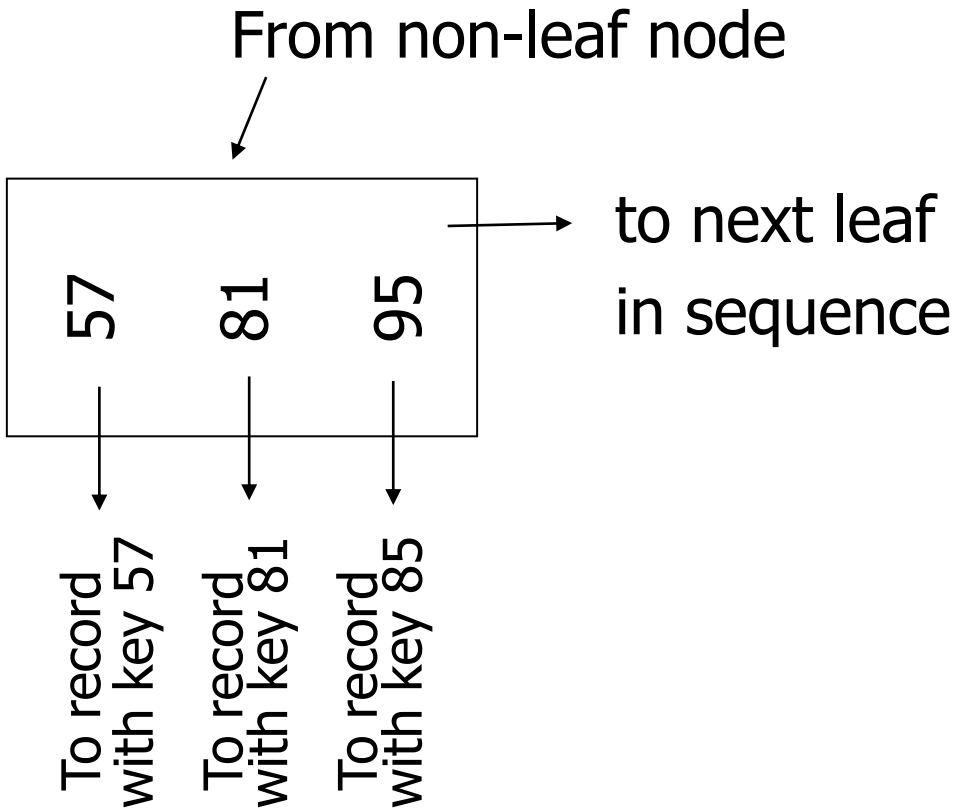
B+Tree Example n=3



Sample non-leaf



Sample leaf node:



Size of nodes: { n+1 pointers (fixed)
n keys (fixed)

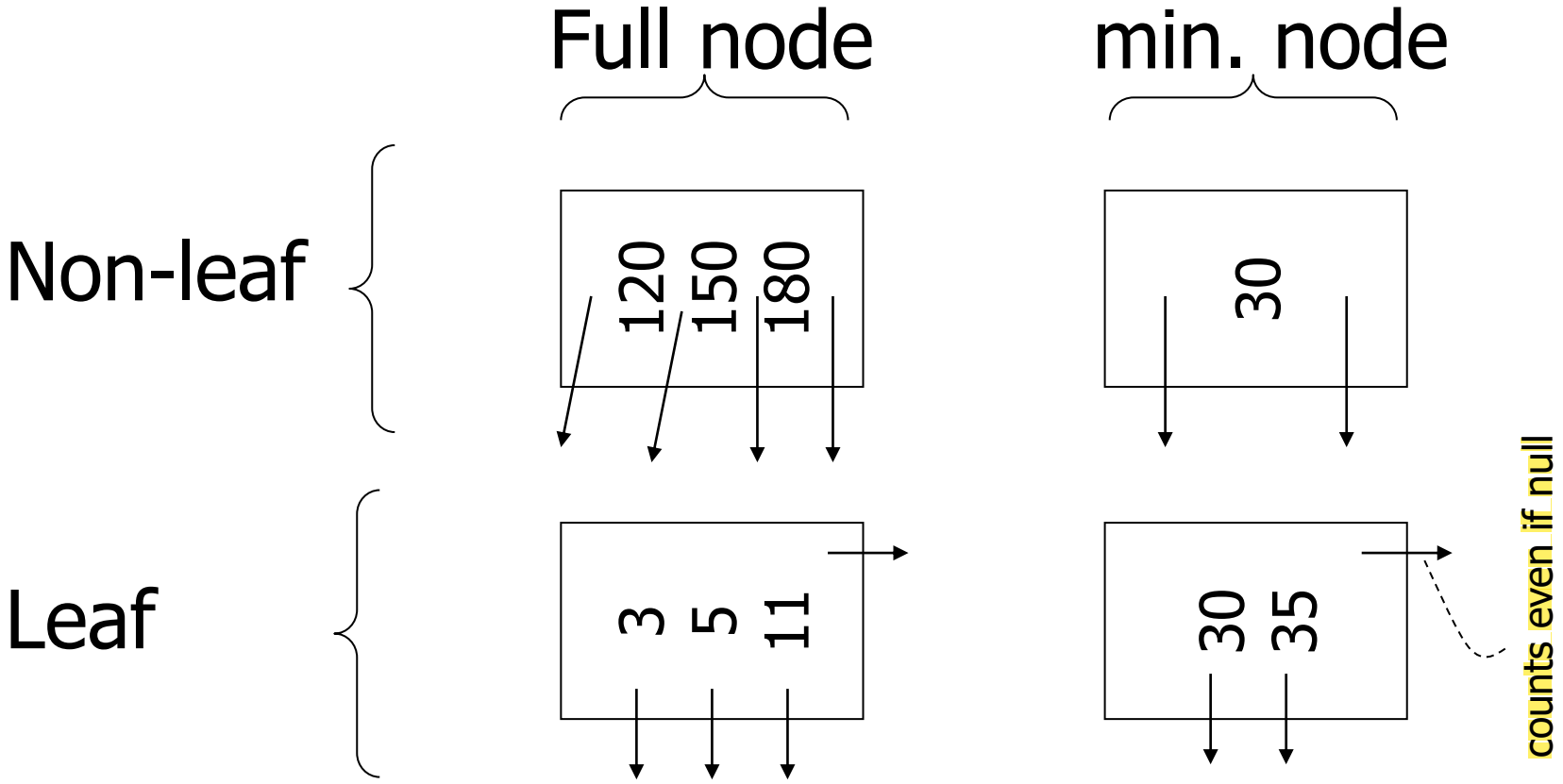
Don’t want nodes to be too empty

- Use at least

  Non-leaf: $\lceil (n+1)/2 \rceil$ pointers

  Leaf: $\lfloor (n+1)/2 \rfloor$ pointers to data

n=3


Full node
min. node
Non-leaf
Leaf
120
150
180
30
3
5
11
30
35
counts even if null

B+tree rules tree of order *n*

(1) All leaves at same lowest level
(balanced tree)

(2) Pointers in leaves point to records
except for "sequence pointer"

(3) Number of pointers/keys for B+tree

	Max ptrs	Max keys	Min ptrs→data	Min keys
Non-leaf (non-root)	n+1	n	$\lceil (n+1)/2 \rceil$	$\lceil (n+1)/2 \rceil - 1$
Leaf (non-root)	n+1	n	$\lfloor (n+1)/2 \rfloor$	$\lfloor (n+1)/2 \rfloor$
Root	n+1	n	1	1

Insert into B+tree

(a) simple case
  - space available in leaf

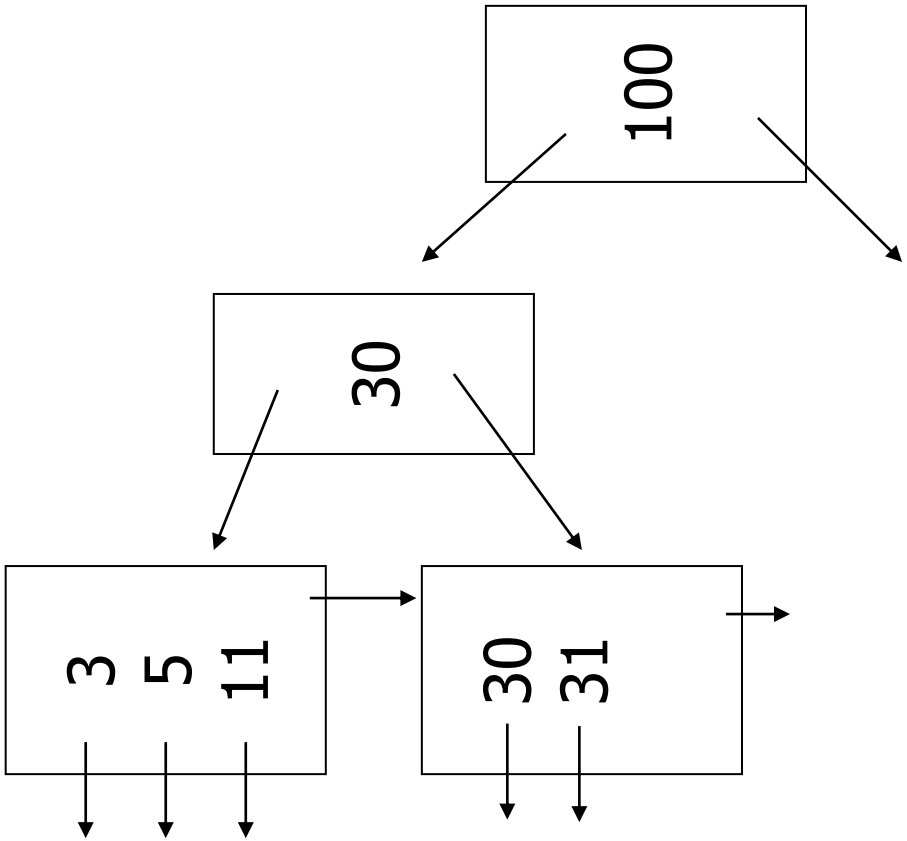
(b) leaf overflow

(c) non-leaf overflow

(d) new root

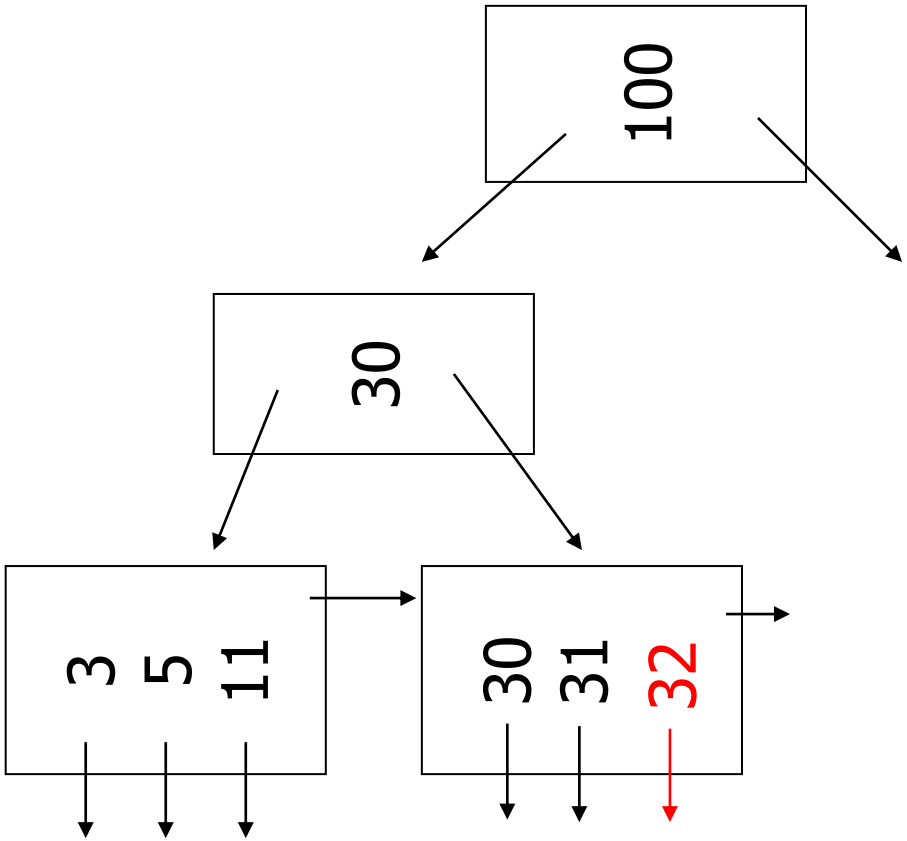
(a) Insert key = 32

n=3



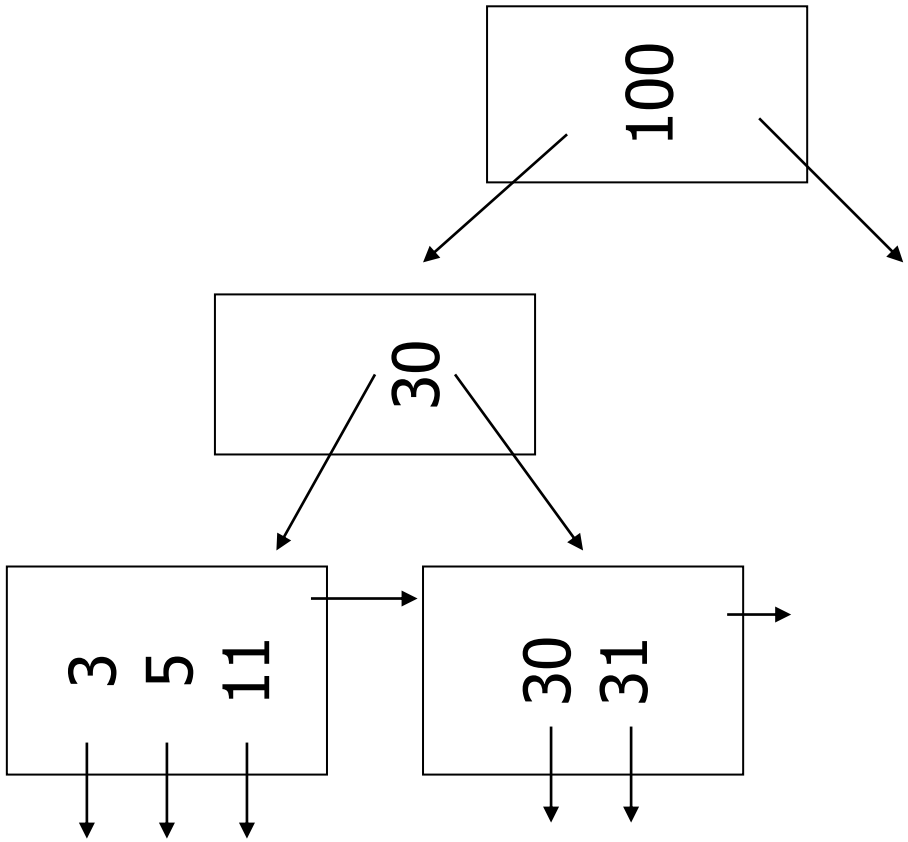
(a) Insert key = 32

n=3



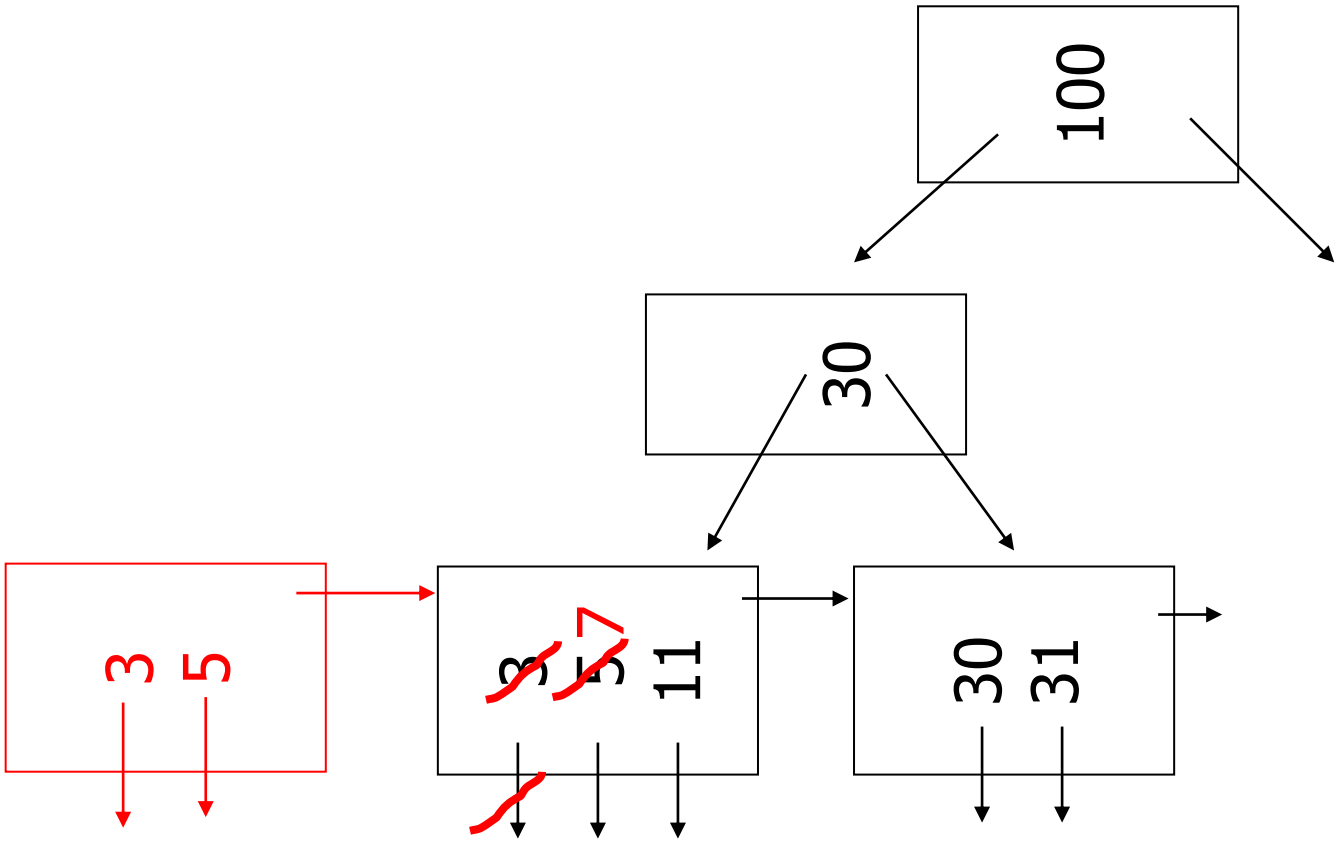
(a) Insert key = 7

n=3



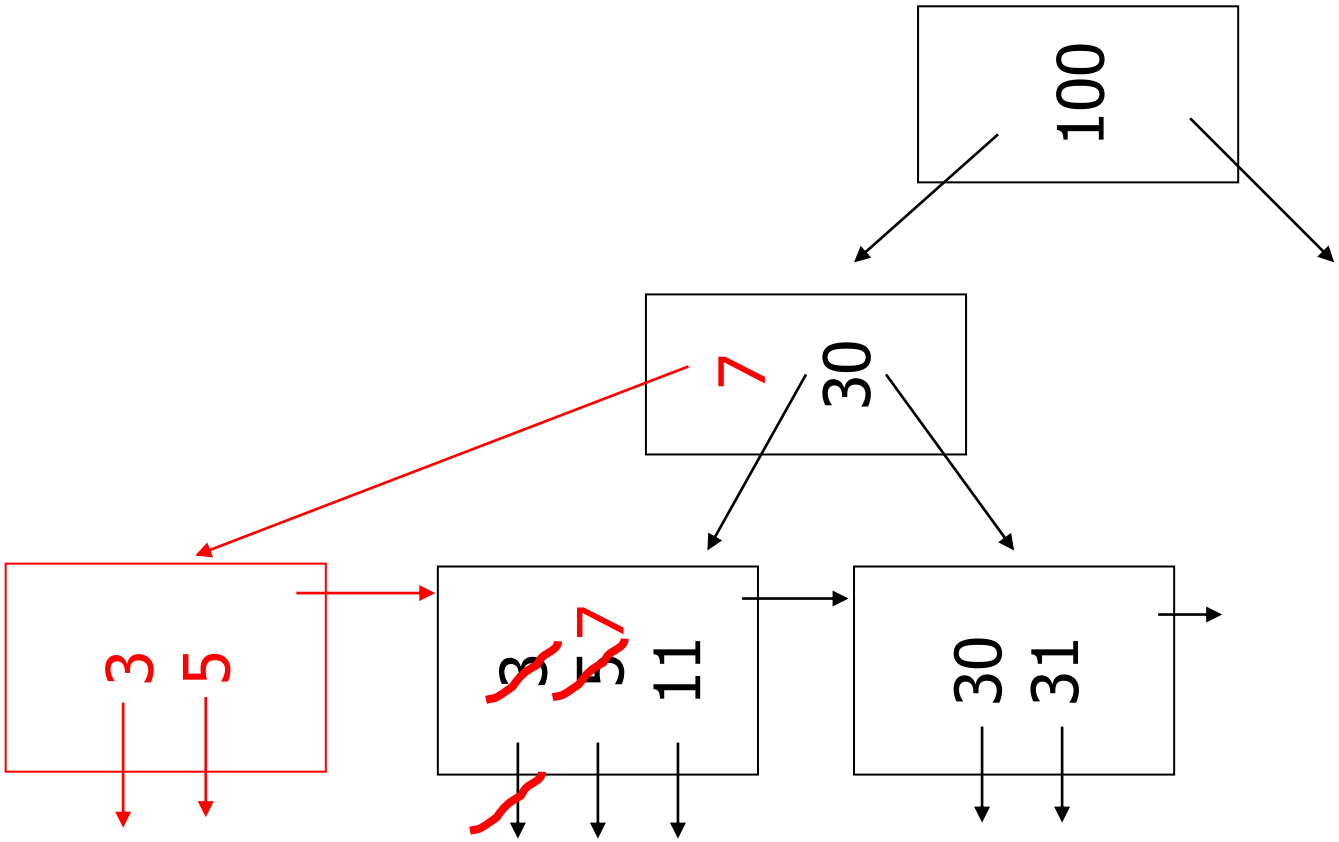
(a) Insert key = 7

n=3



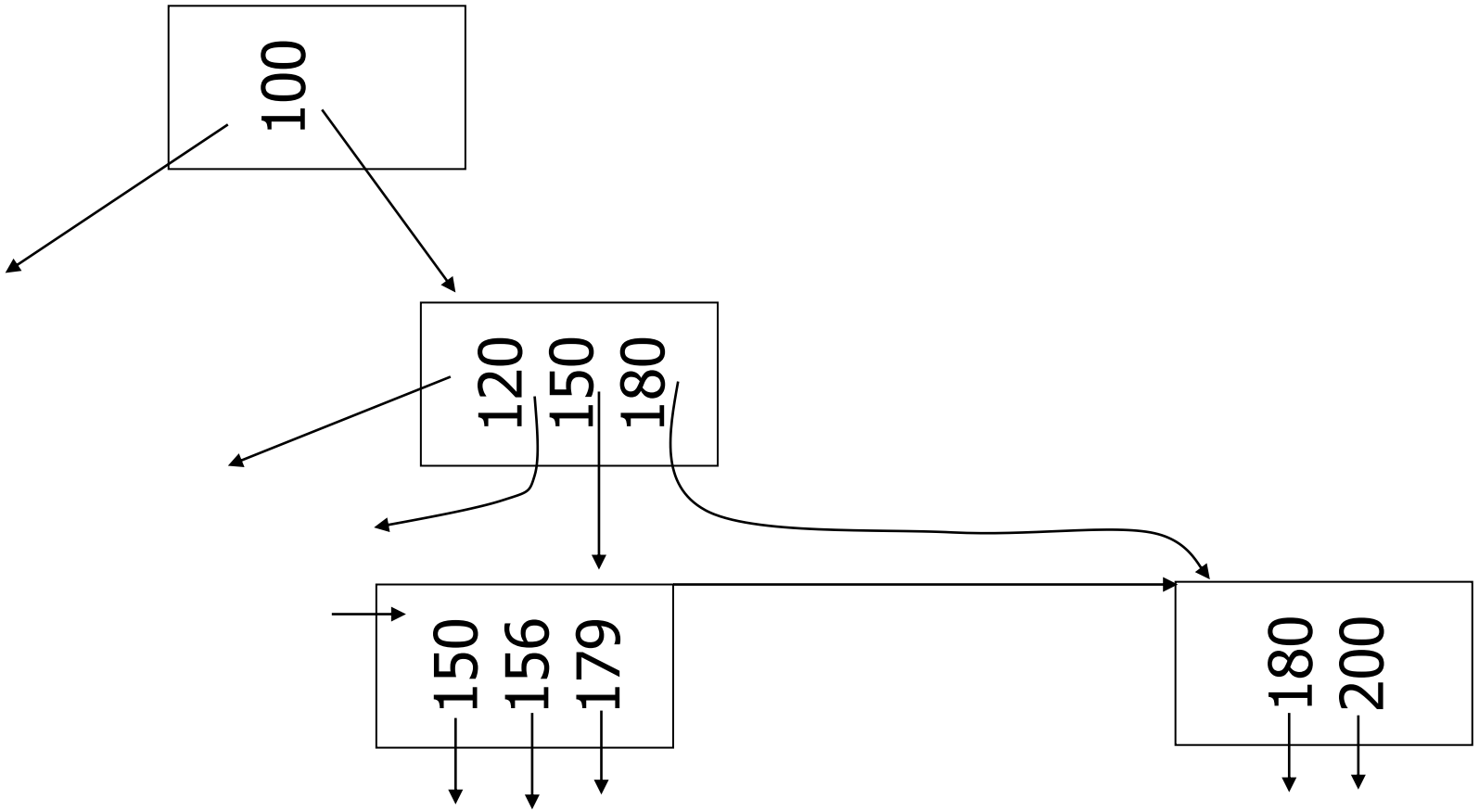
(a) Insert key = 7

n=3



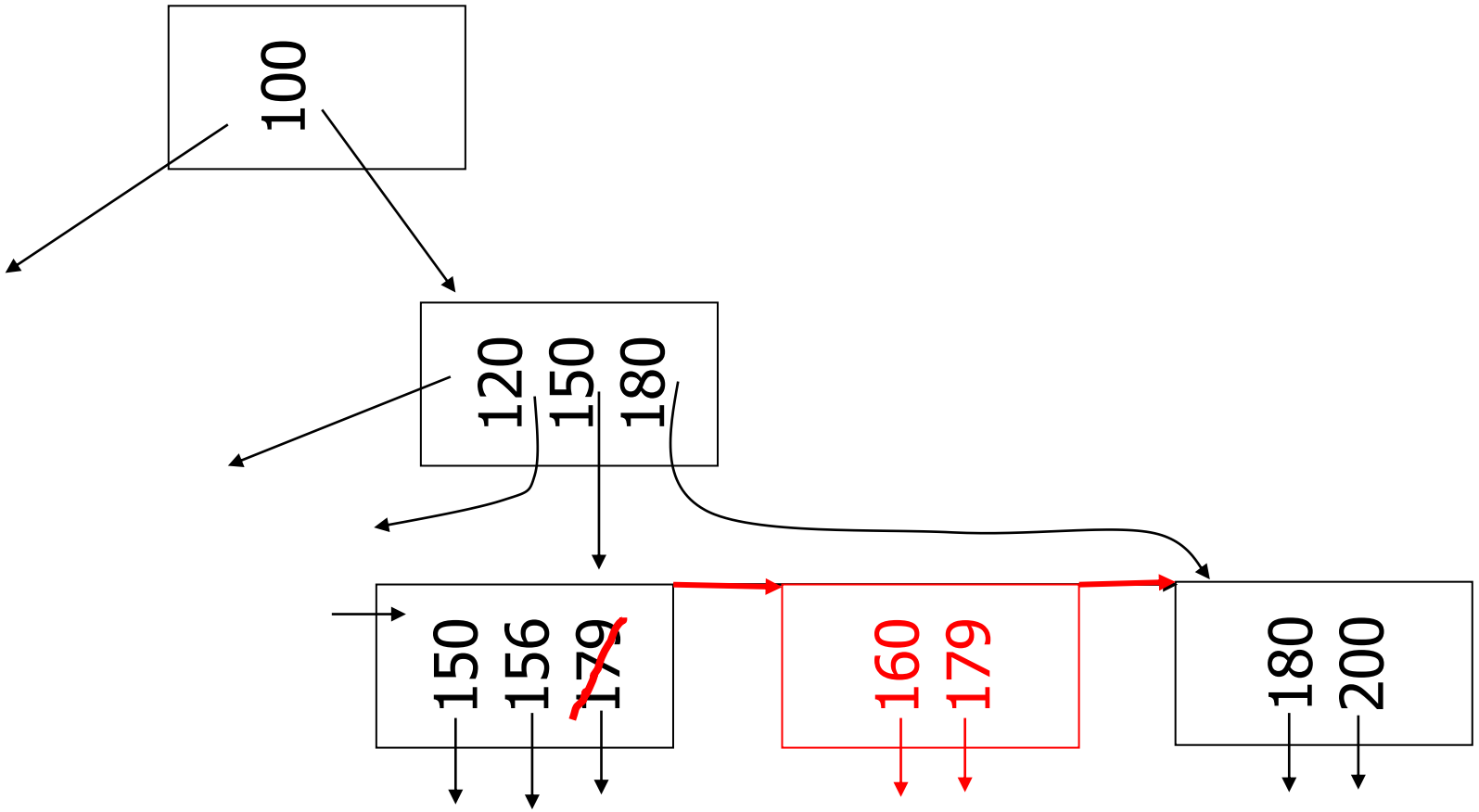
(c) Insert key = 160

n=3



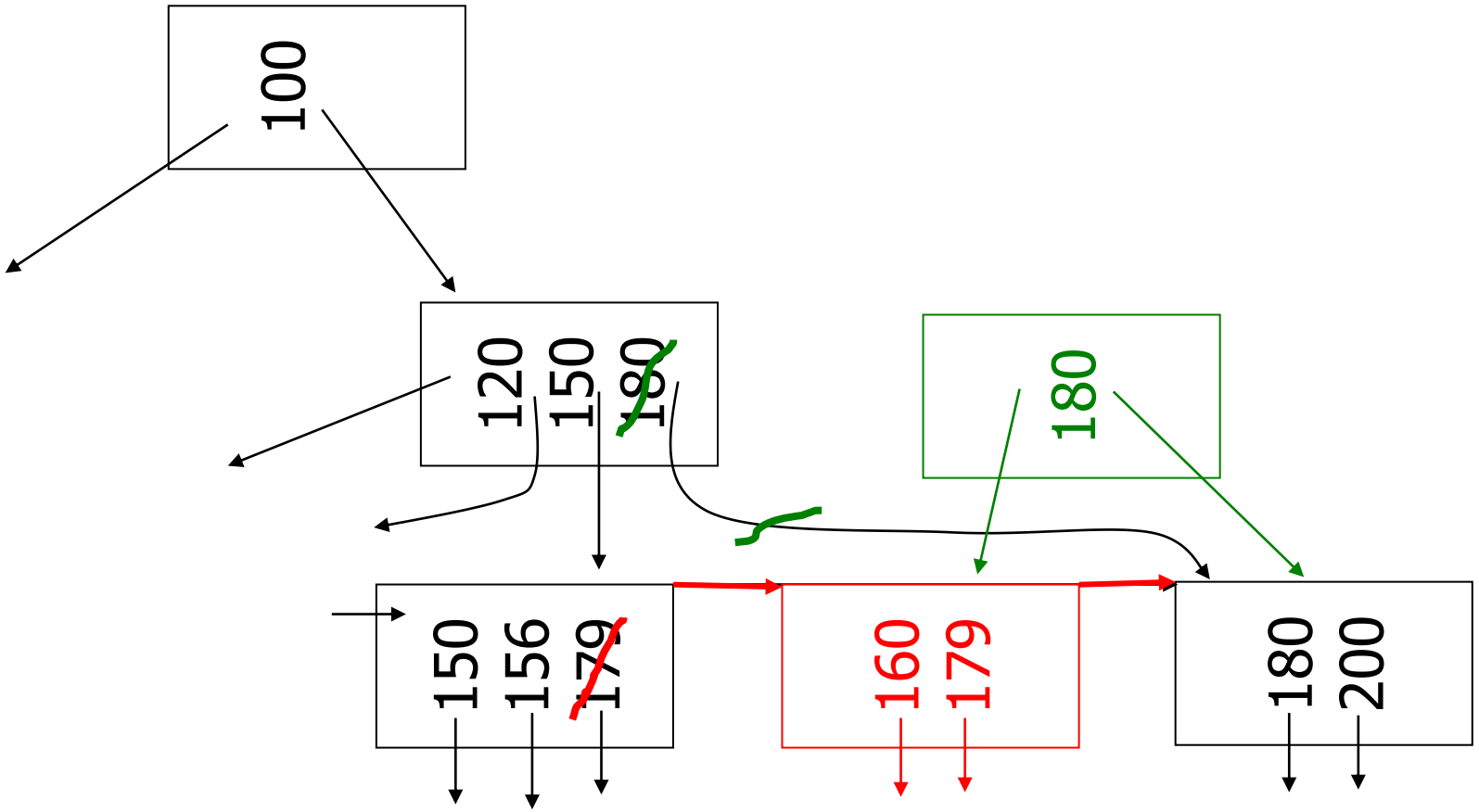
(c) Insert key = 160

n=3



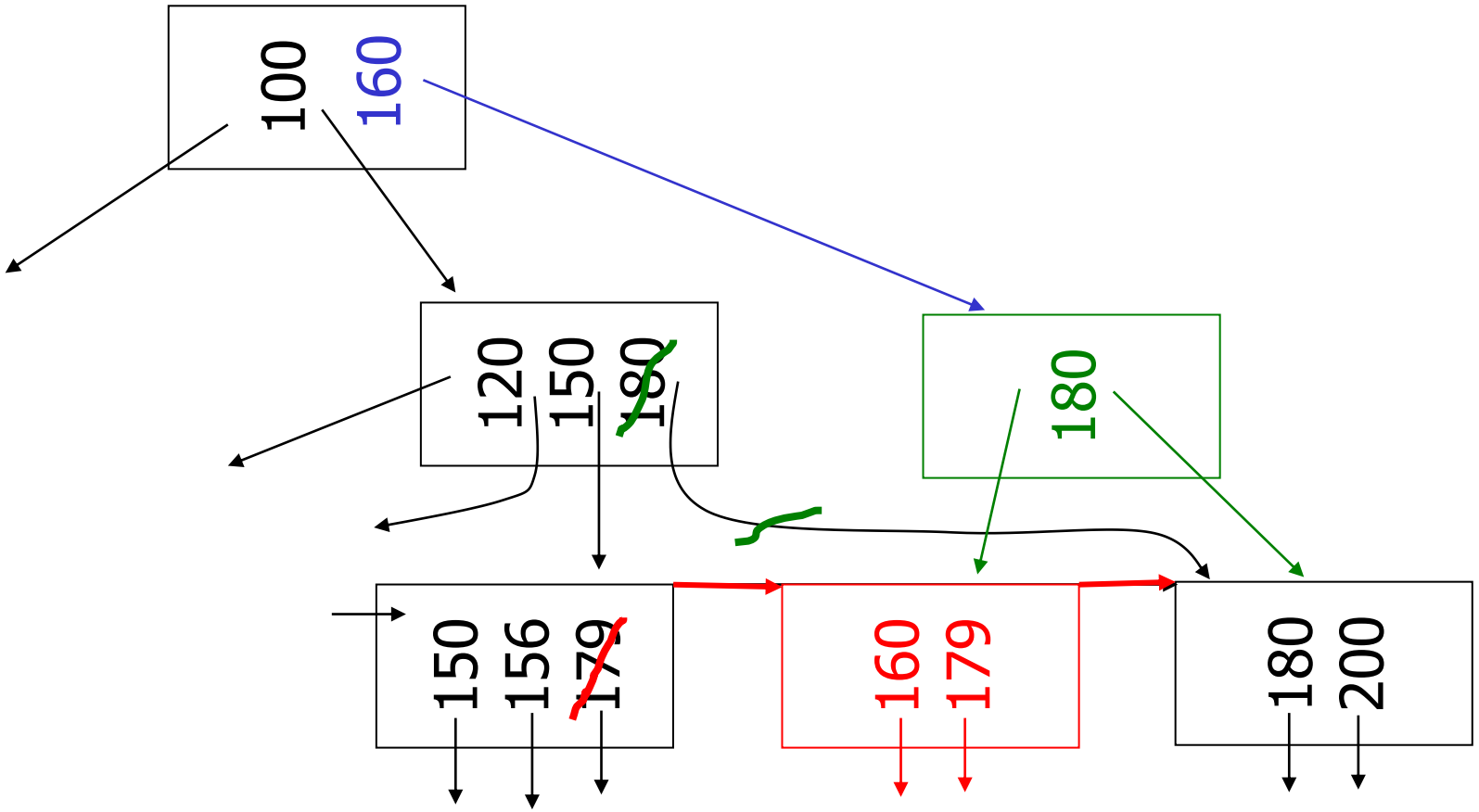
(c) Insert key = 160

n=3



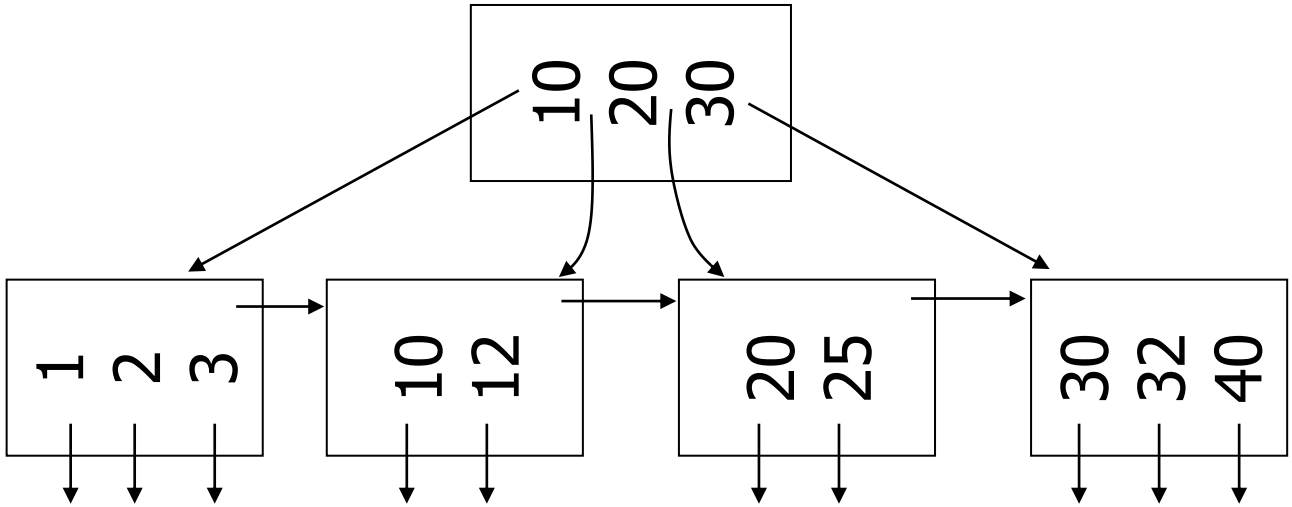
(c) Insert key = 160

n=3



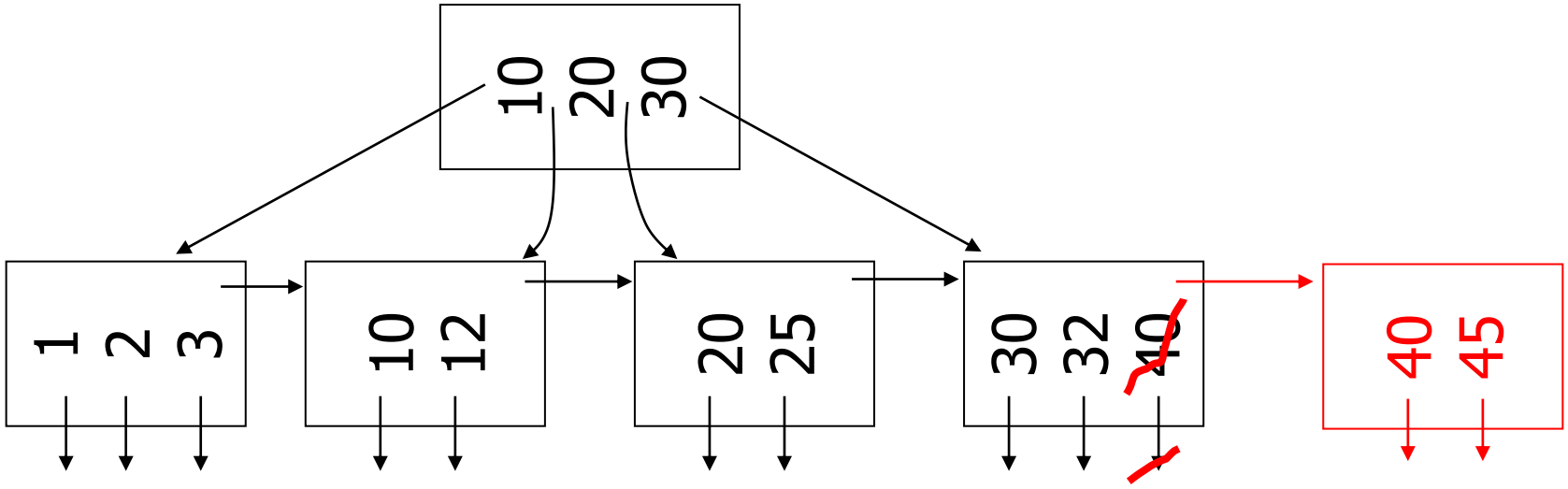
(d) New root, insert 45

n=3



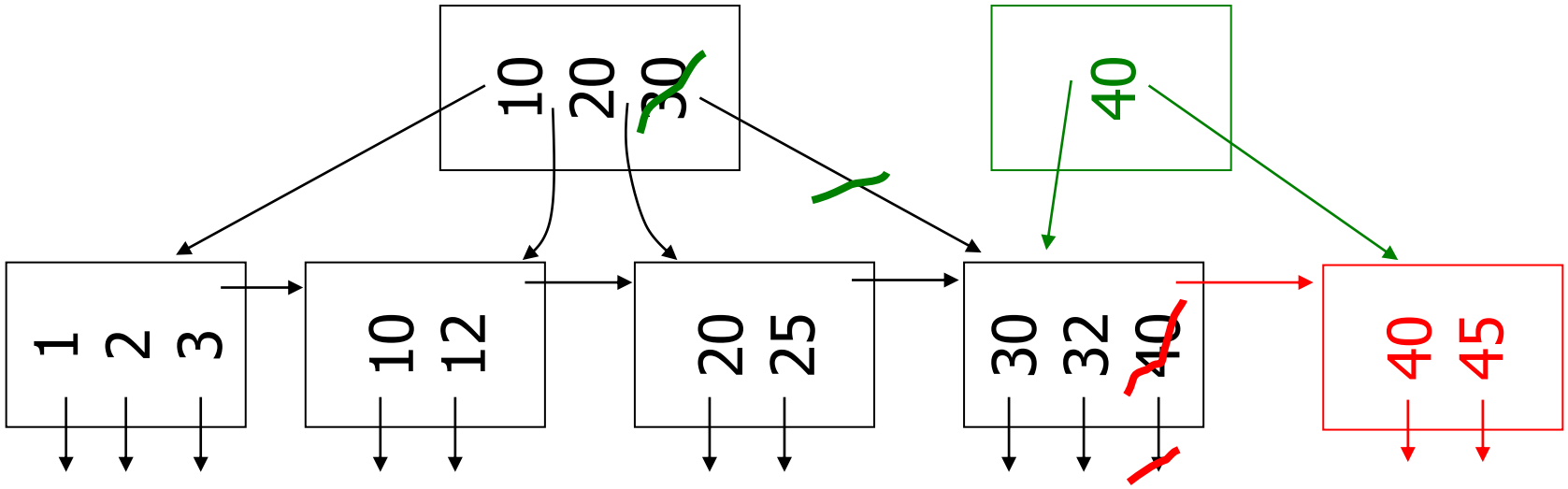
(d) New root, insert 45

n=3



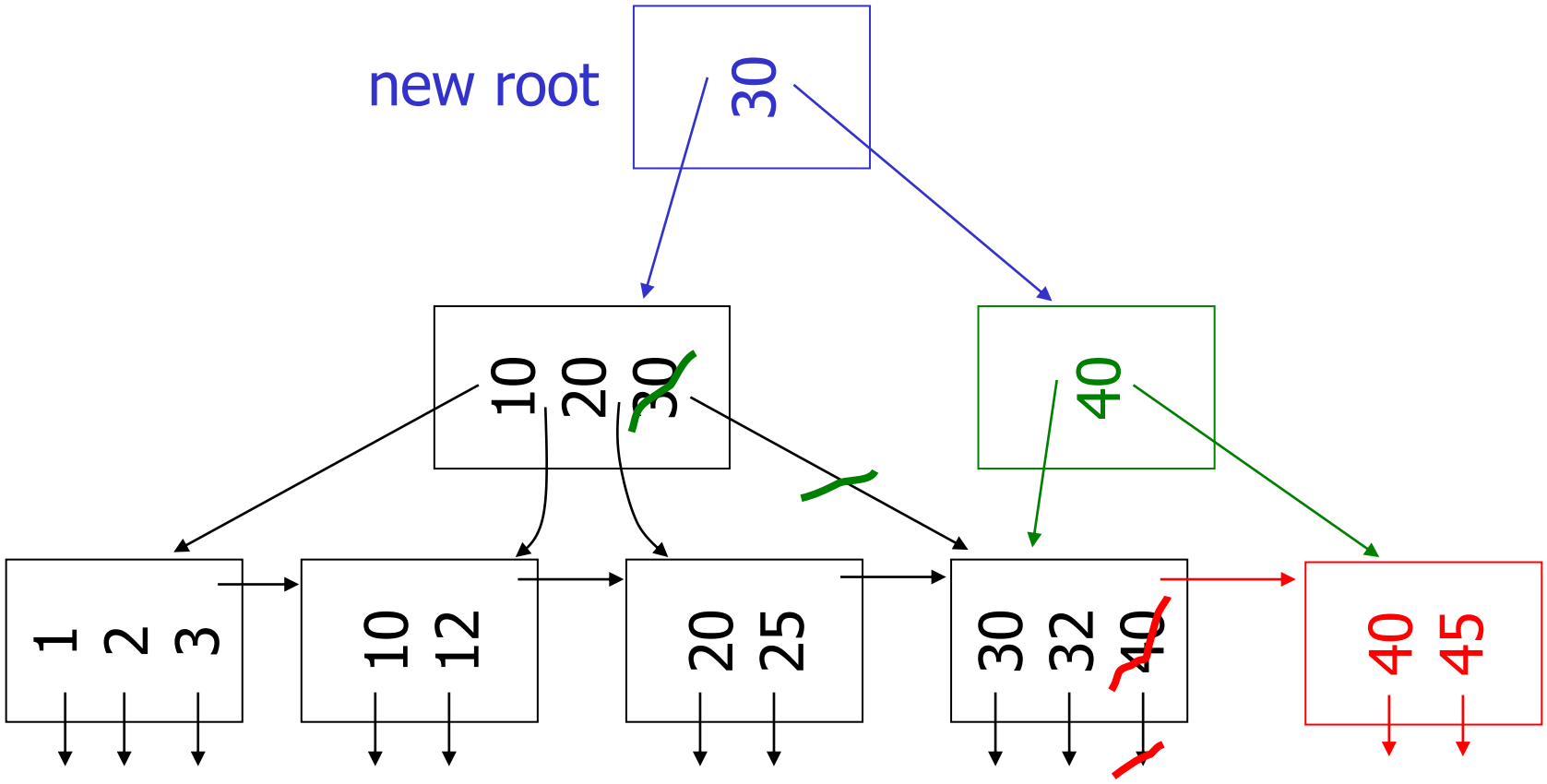
(d) New root, insert 45

n=3



(d) New root, insert 45

n=3



Deletion from B+tree

(a) Simple case - no example

(b) Coalesce with neighbor (sibling)

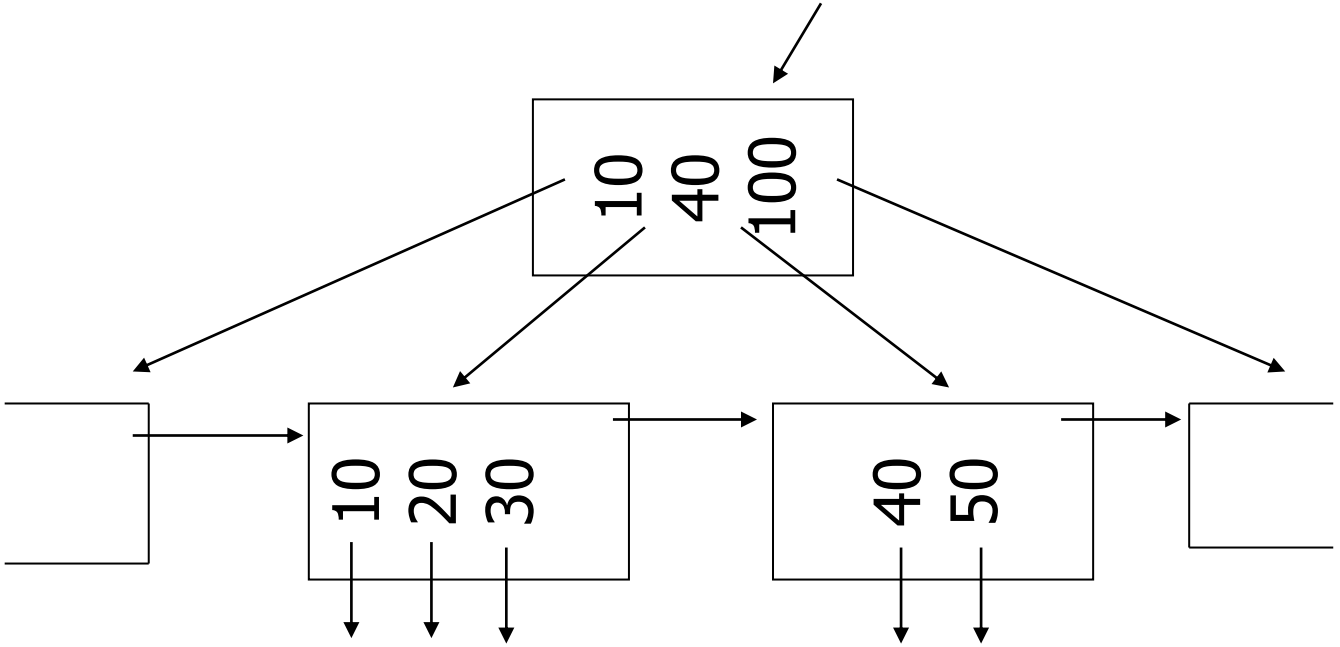
(c) Re-distribute keys

(d) Cases (b) or (c) at non-leaf

(b) Coalesce with sibling

– Delete 50

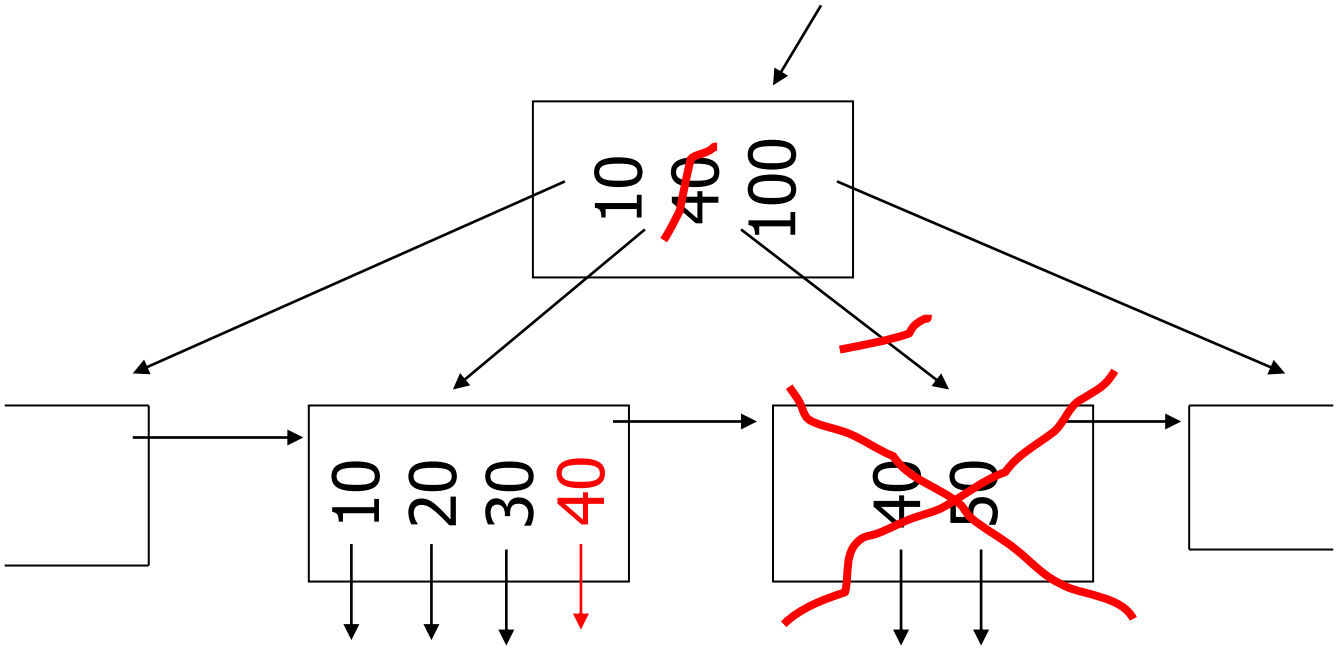
n=4



(b) Coalesce with sibling

– Delete 50

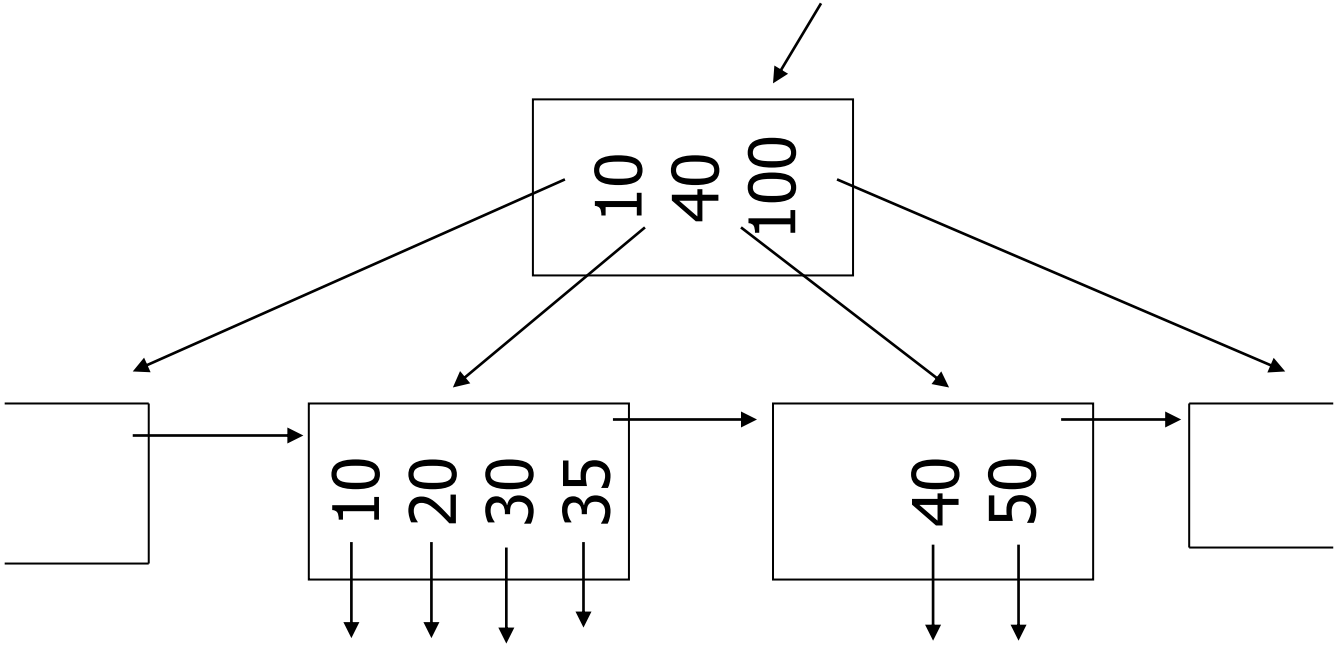
n=4



(c) Redistribute keys

– Delete 50

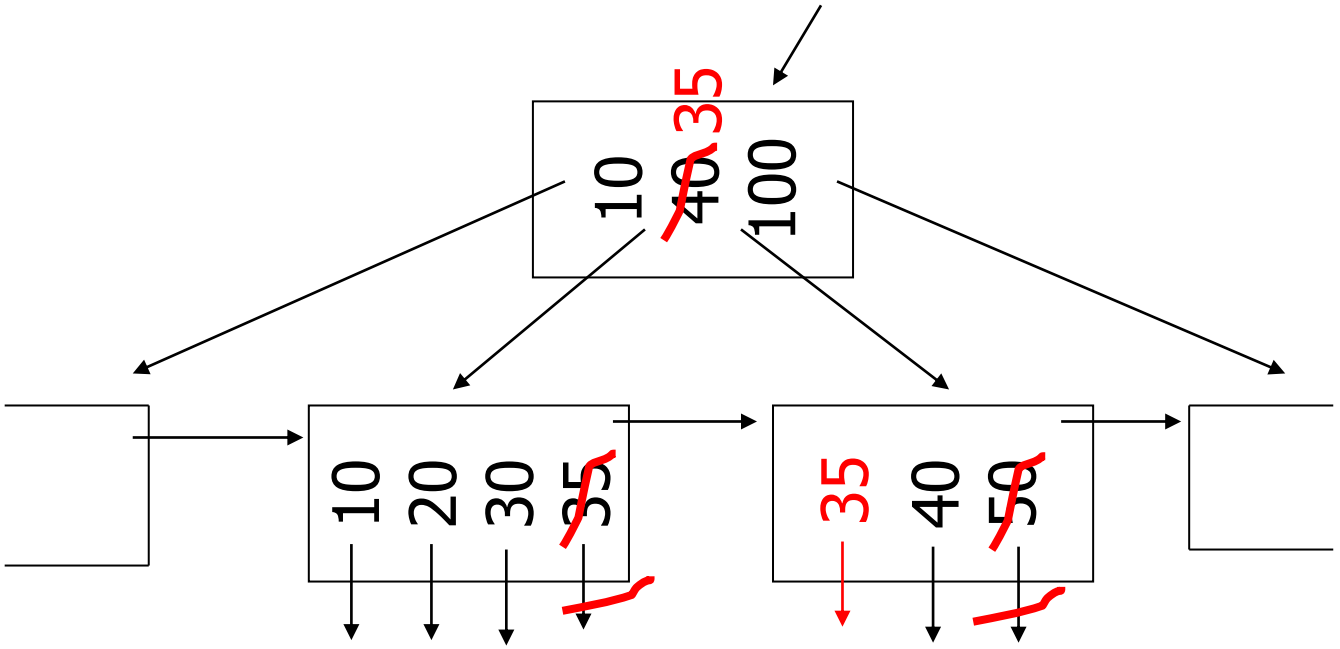
n=4



(c) Redistribute keys

– Delete 50

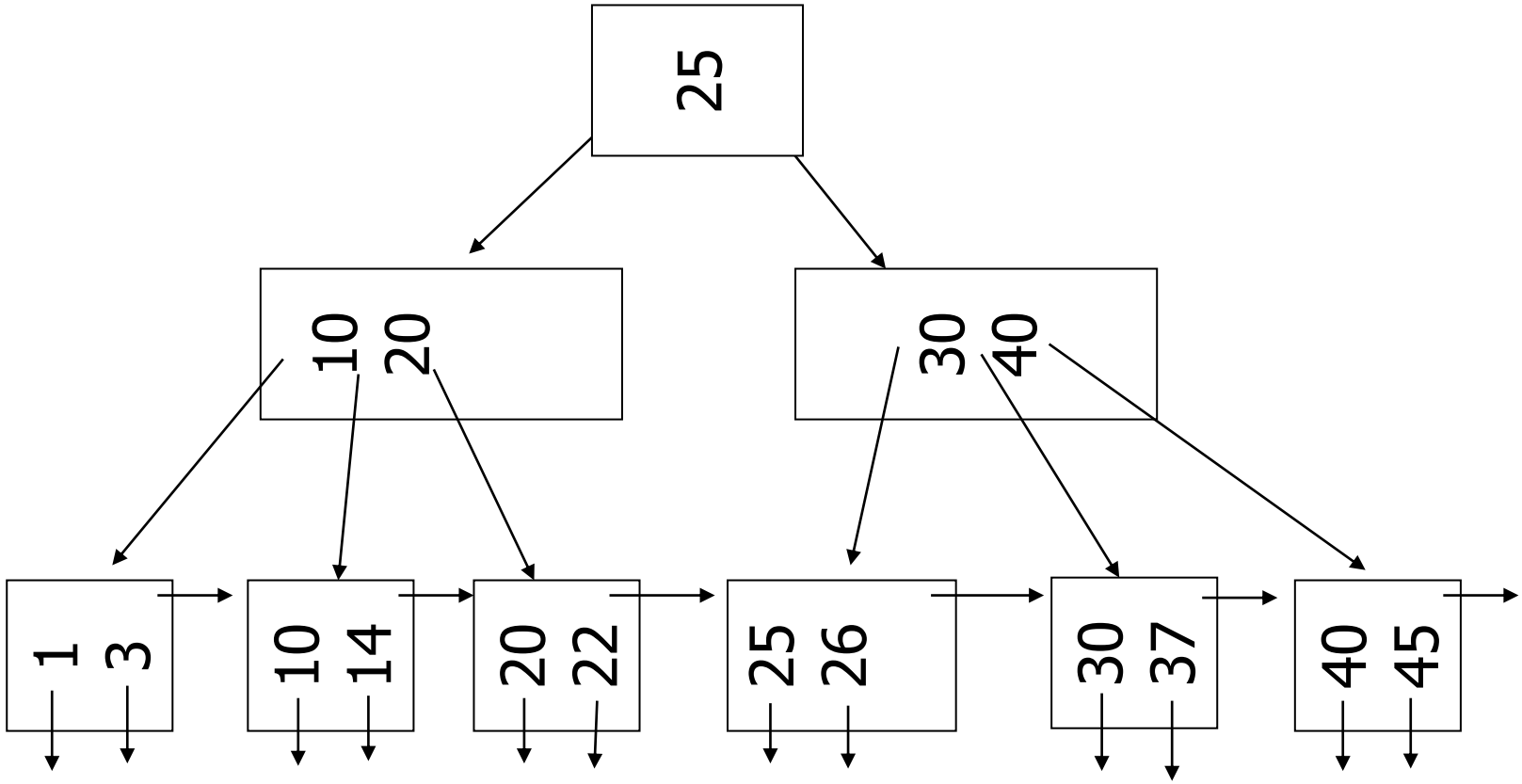
n=4



(d) Non-leaf coalesce

– Delete 37

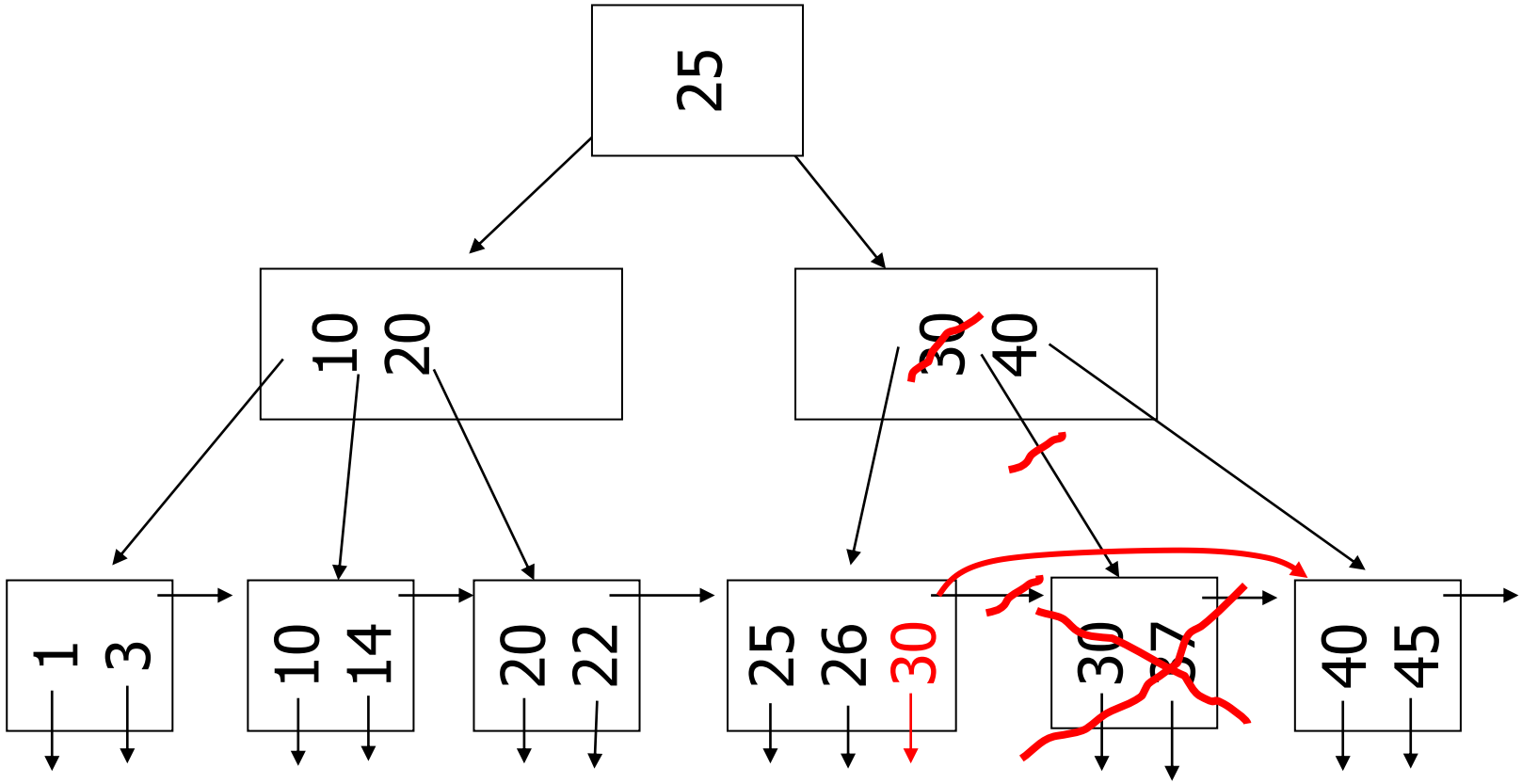
n=4



(d) Non-leaf coalesce

– Delete 37

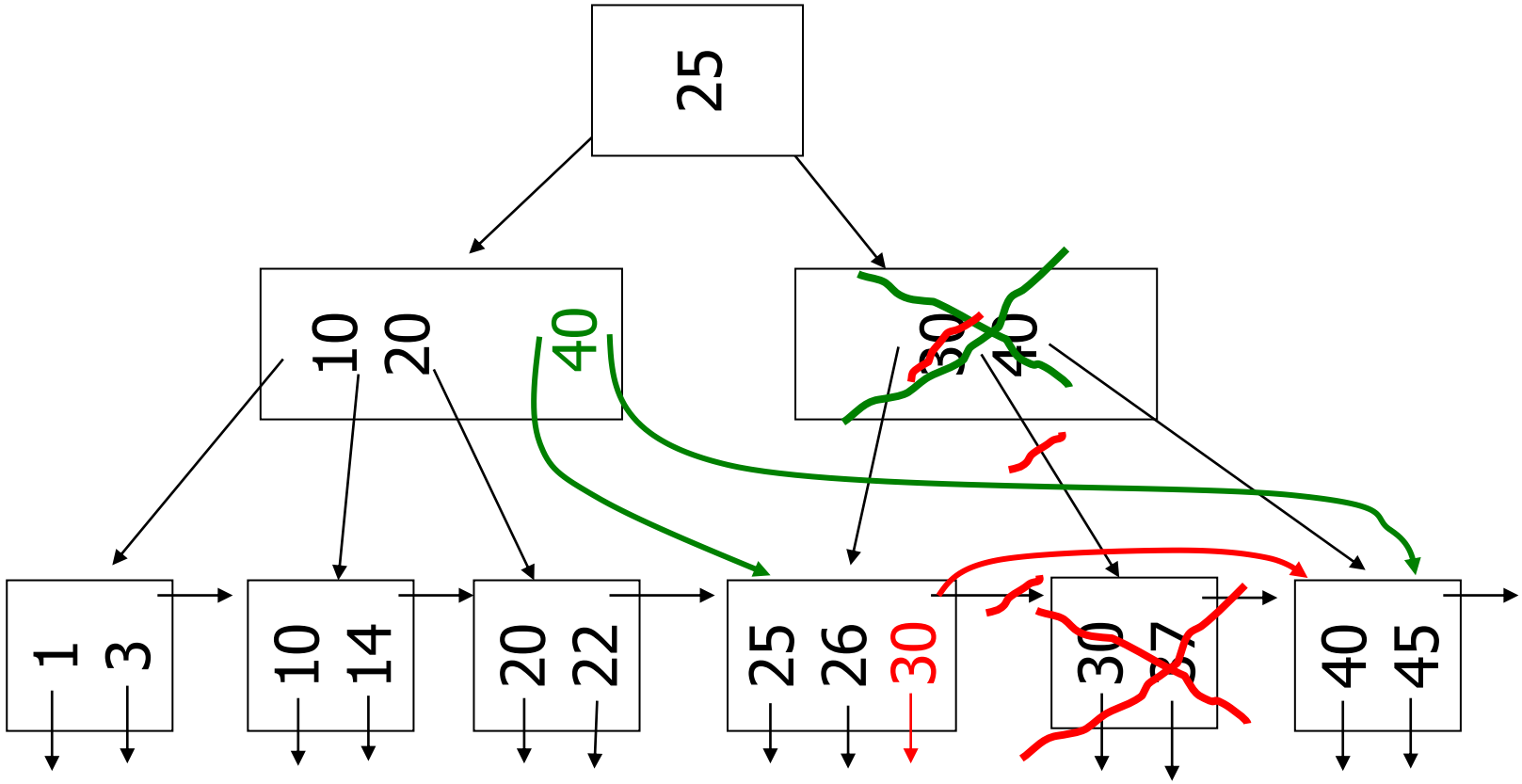
n=4



(d) Non-leaf coalesce

– Delete 37

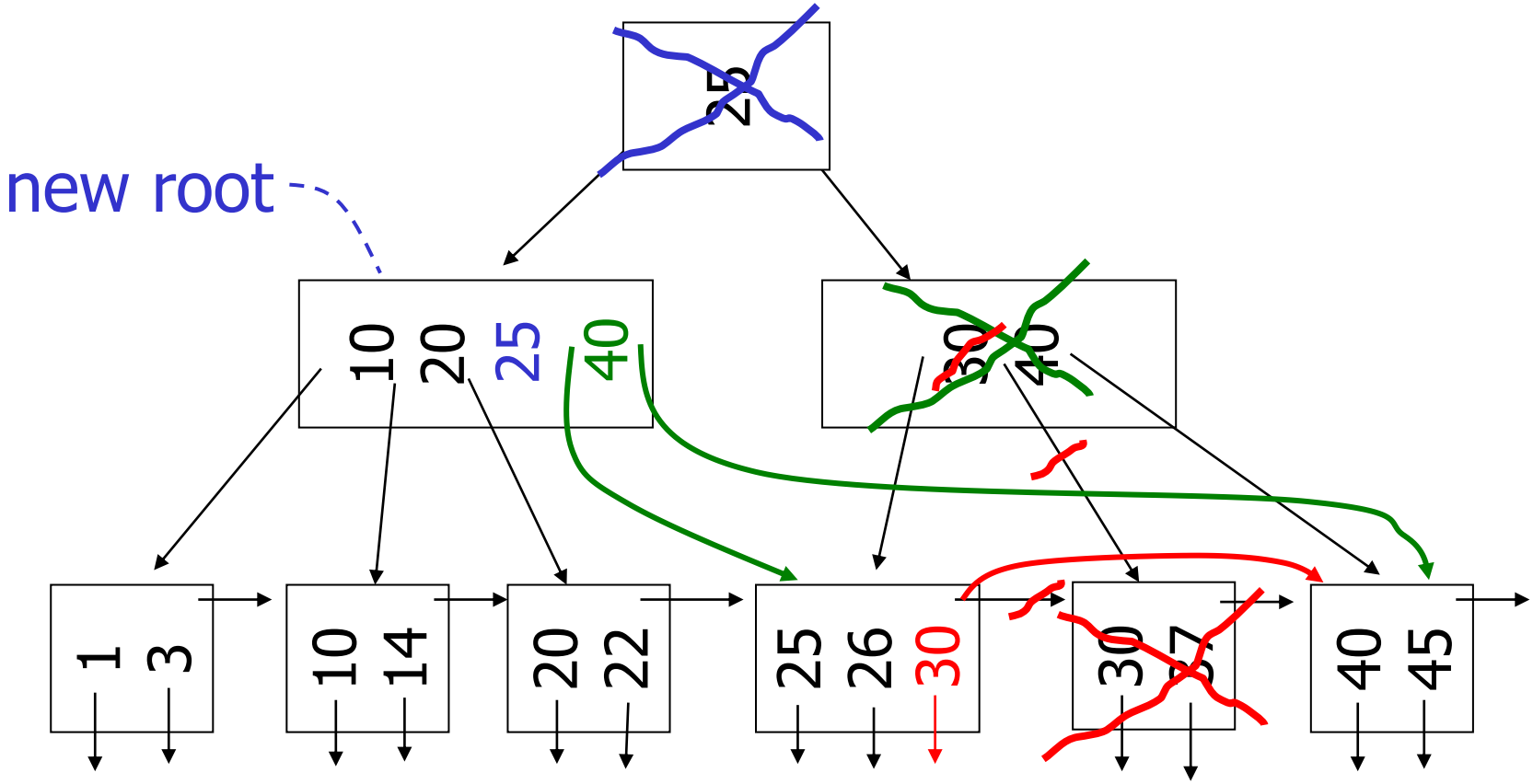
n=4



(d) Non-leaf coalesce

– Delete 37

n=4



B+tree deletions in practice

- Often, coalescing is not implemented
  - Too hard and not worth it!

http://mlwiki.org/index.php/B-Tree

Outline/summary

- Conventional Indexes
  - Sparse vs. dense
  - Primary vs. secondary
- B trees
- Hashing schemes (recommended reading, not mandatory)

The slides in this lecture are taken from:

- Hector Garcia-Molina, CS 245: Database System Principles, Notes 4: Indexing.

Reading

- Héctor García-Molina, Jeffrey Ullman, and Jennifer Widom. Database Systems: The Complete Book