Data Storage & Indexes

Key Operations on an Index

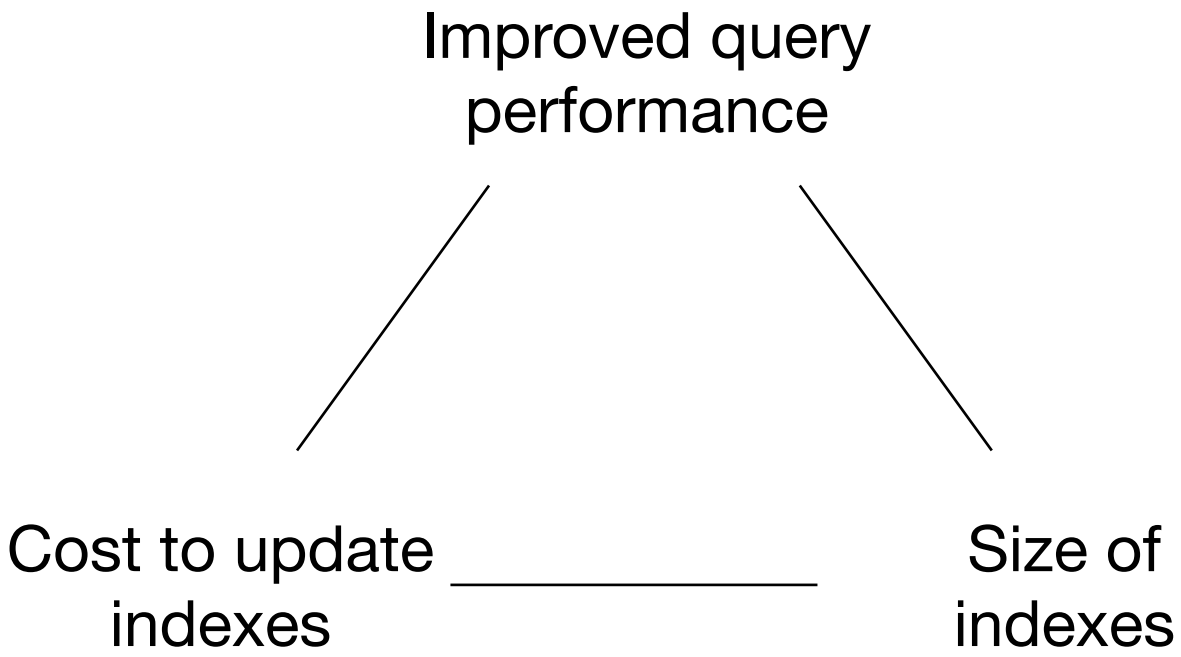
Find all records with a given **value** for a key

- » Key can be one field or a tuple of fields (e.g. `country="US" AND state="CA"`)
- » In some cases, only one matching record

Find all records with key in a given **range**

Find **nearest neighbor** to a data point?

Tradeoffs in Indexing



Some Types of Indexes

Conventional indexes

B-trees

Hash indexes

Multi-key indexing

Many standard data structures, but adapted to work well on disk

Sequential File

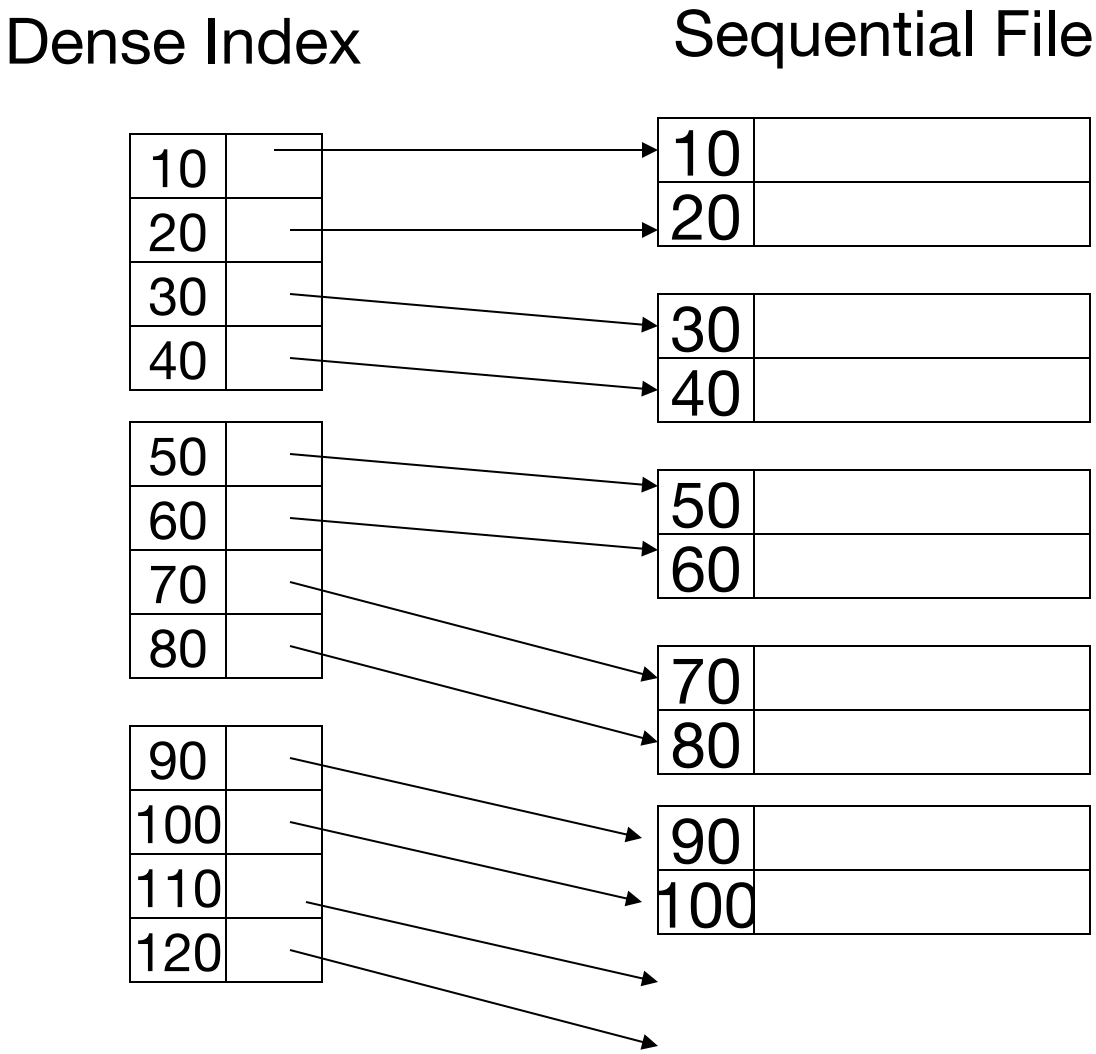
10	
20	

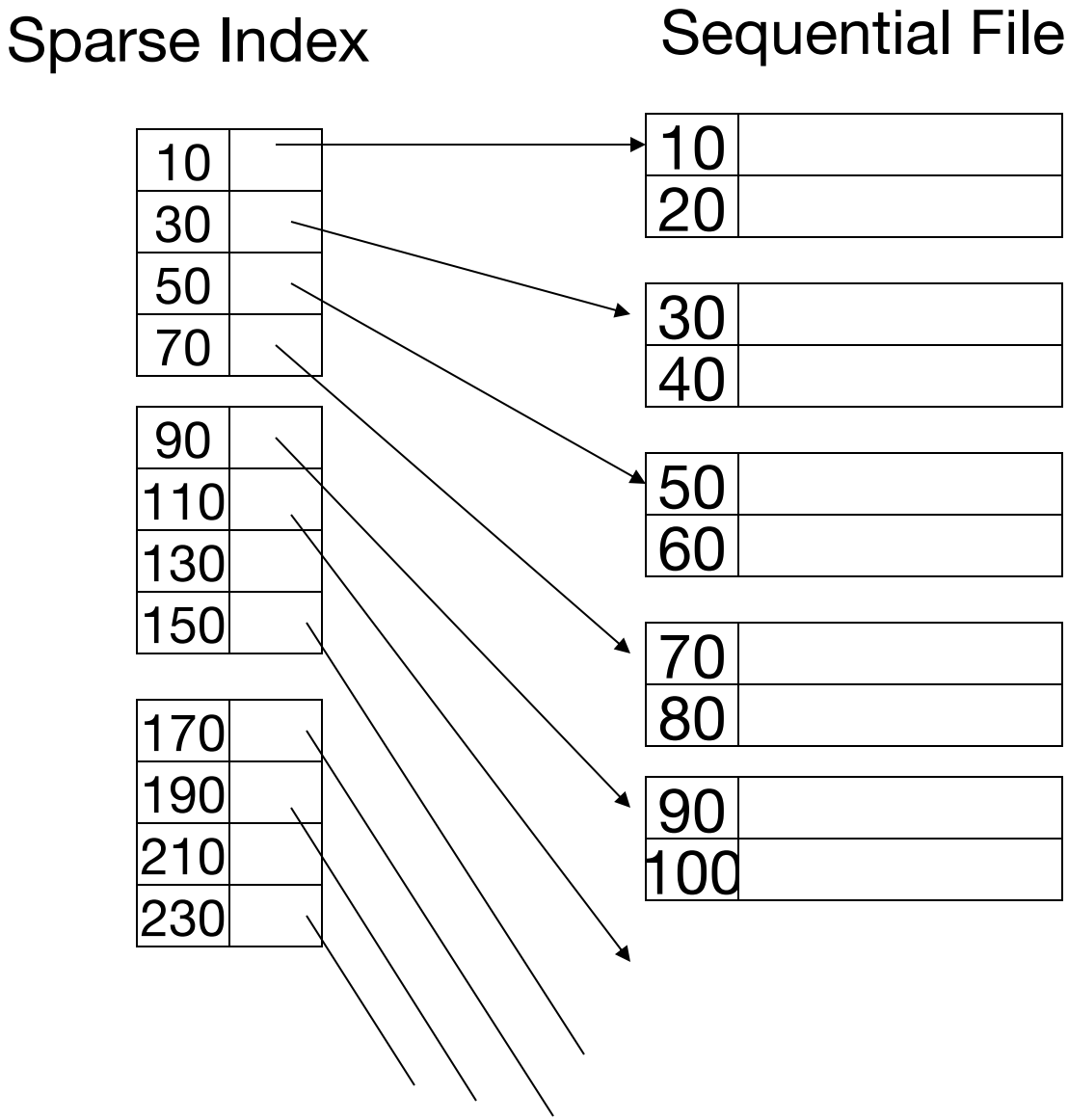
30	
40	

50	
60	

70	
80	

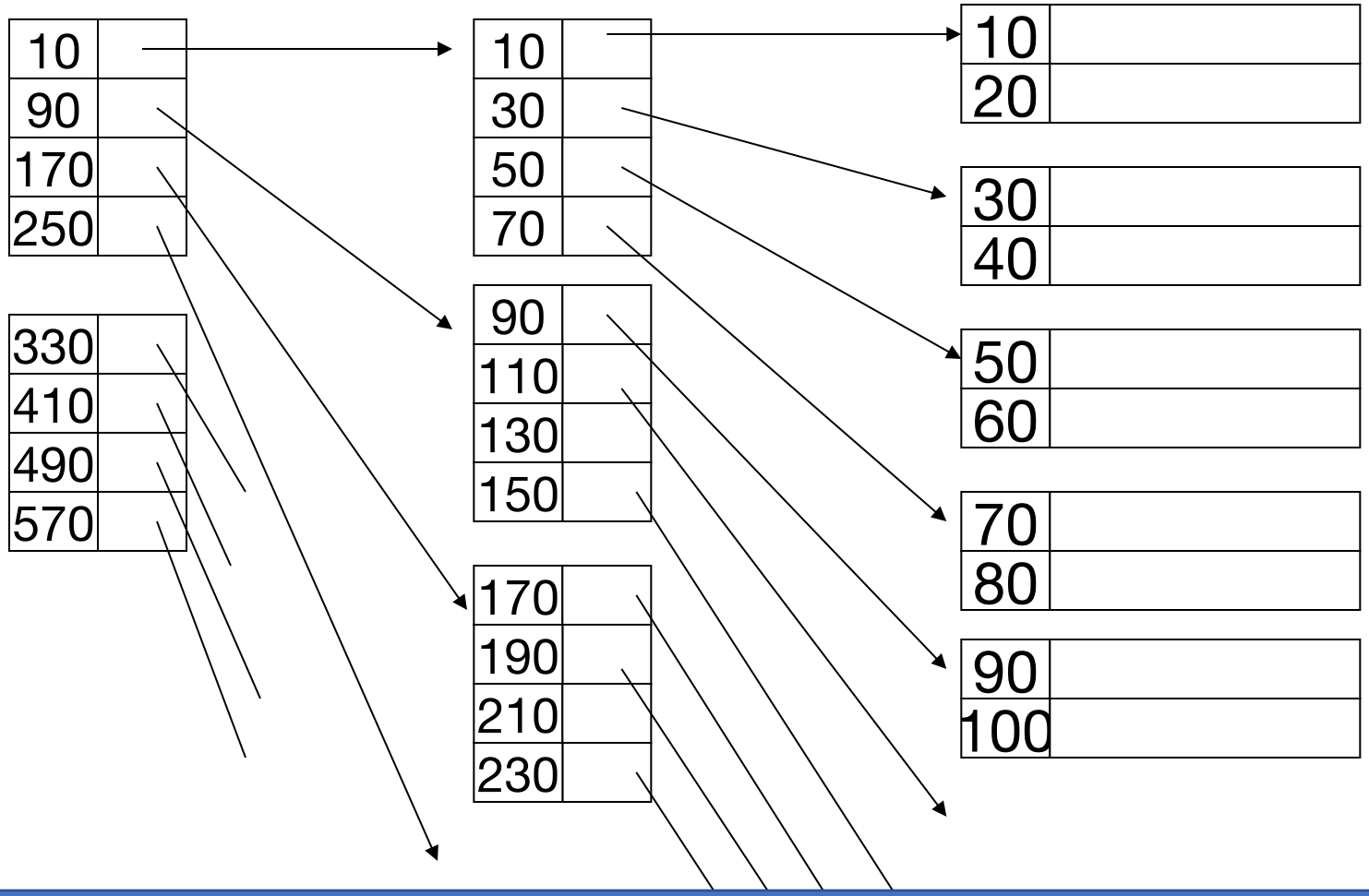
90	
100	


Dense Index
Sequential File
10
20
30
40
50
60
70
80
90
100
110
120
10
20
30
40
50
60
70
80
90
100


Sparse Index
Sequential File
10
30
50
70
90
110
130
150
170
190
210
230
10
20
30
40
50
60
70
80
90
100

2-level sparse index

Sequential File



File and 2nd level index blocks need not be contiguous on disk

Sparse vs Dense Tradeoff

Sparse: Less space usage, can keep more of index in memory

Dense: Can tell whether a key is present without accessing file

(Later: sparse better for insertions, dense needed for secondary indexes)

Terms

Search key of an index

Primary index (on primary key of ordered files)

Secondary index

Dense index (contains all search key values)

Sparse index

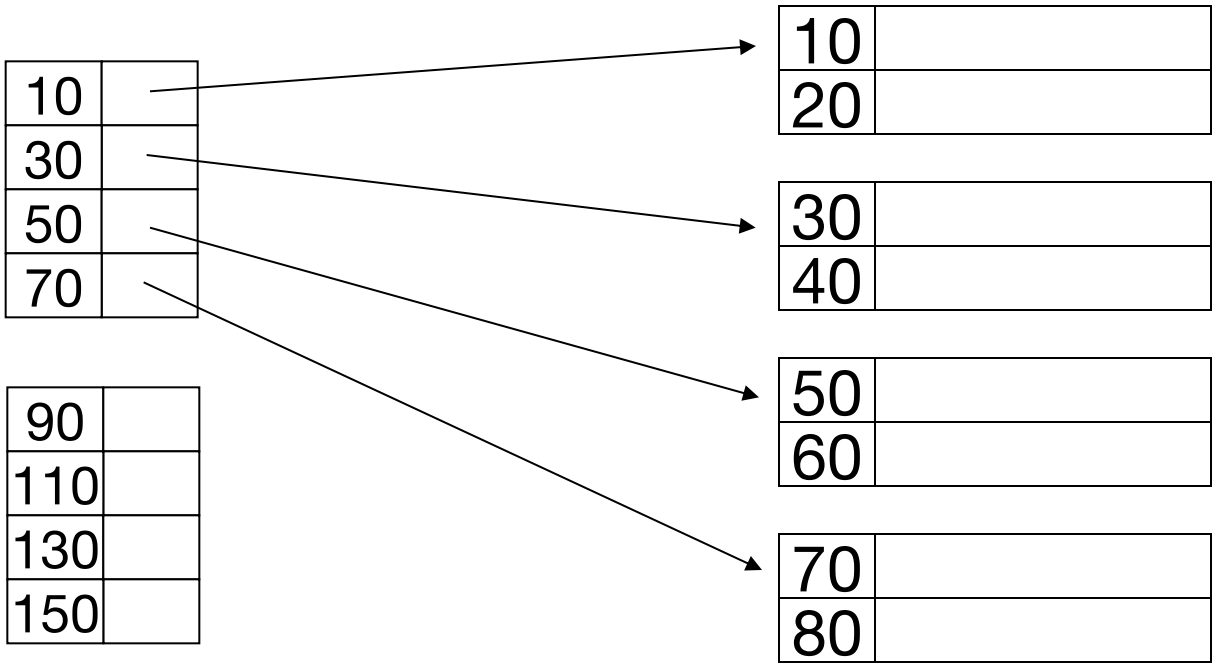
Multi-level index

Handling Duplicate Keys

For a primary index, can point to 1st instance of each item (assuming blocks are linked)

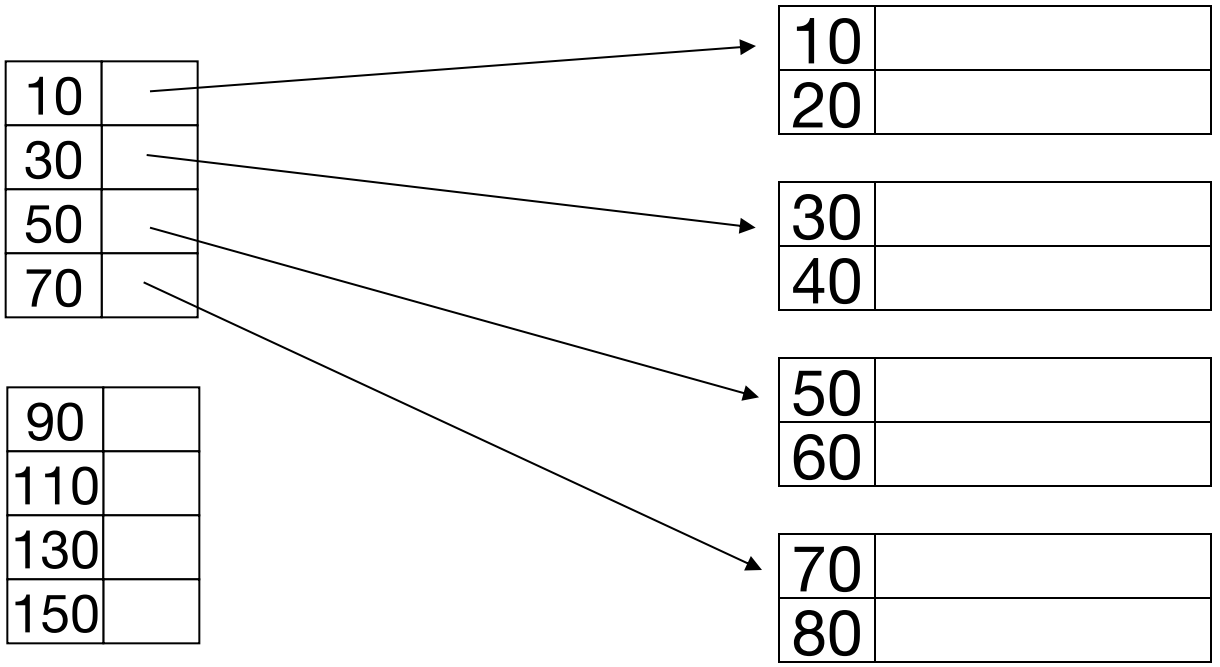
For a secondary index, need to point to a list of records since they can be anywhere

Deletion: Sparse Index



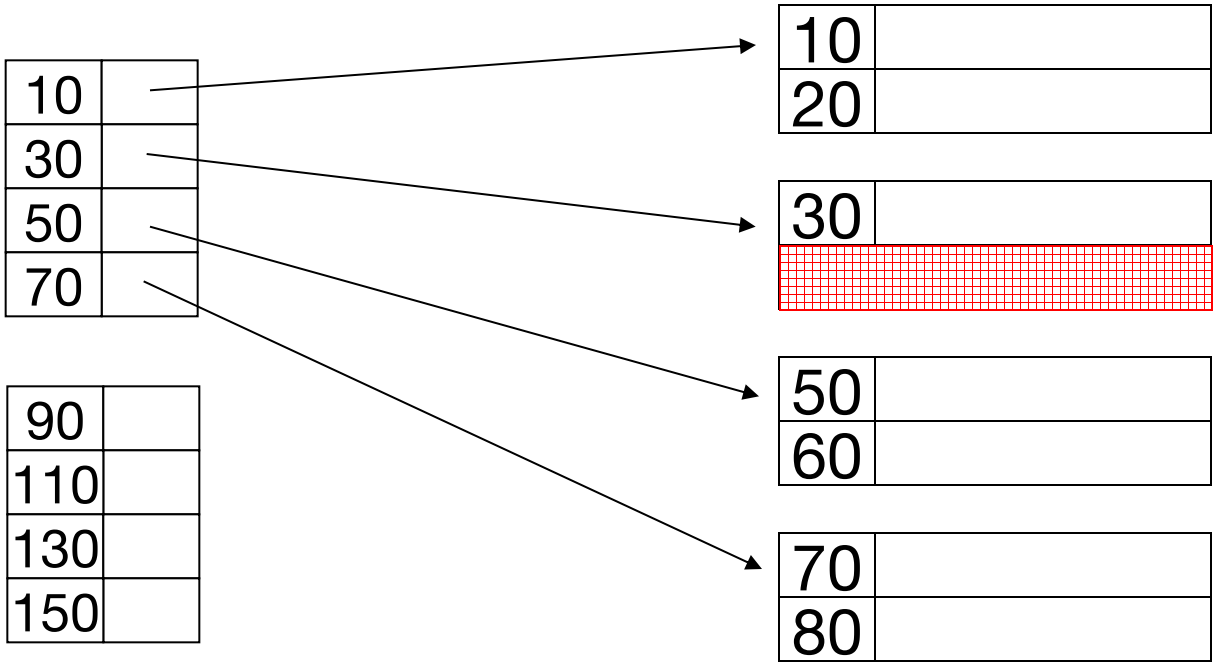
Deletion: Sparse Index

– delete record 40



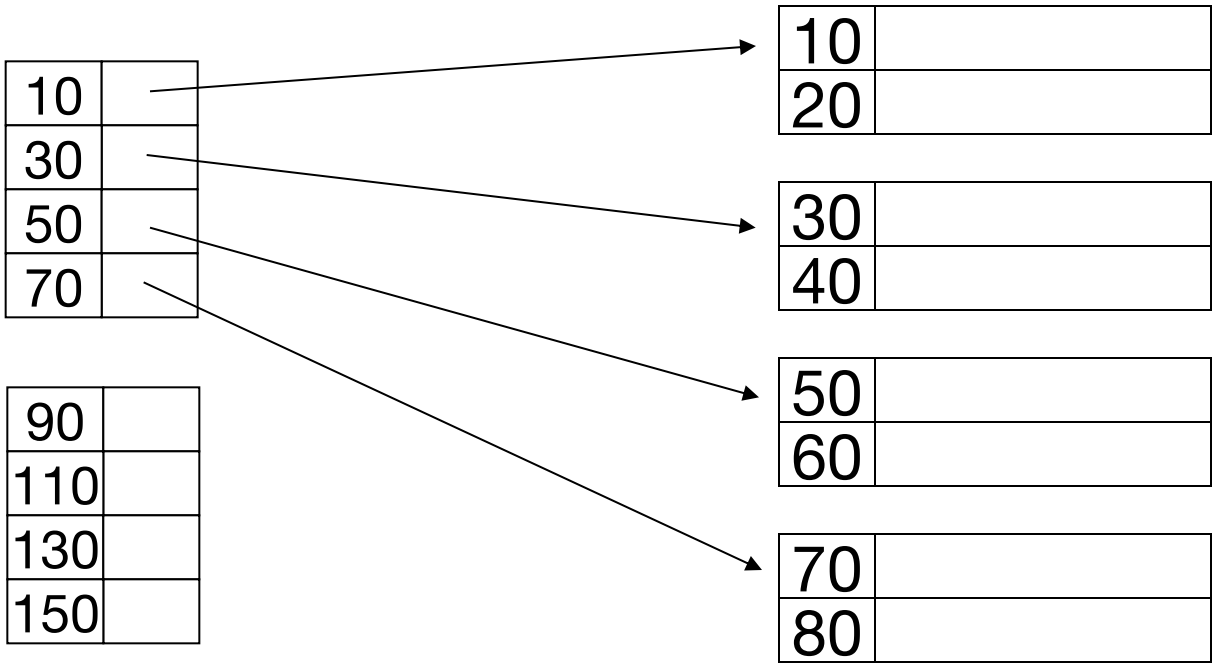
Deletion: Sparse Index

– delete record 40



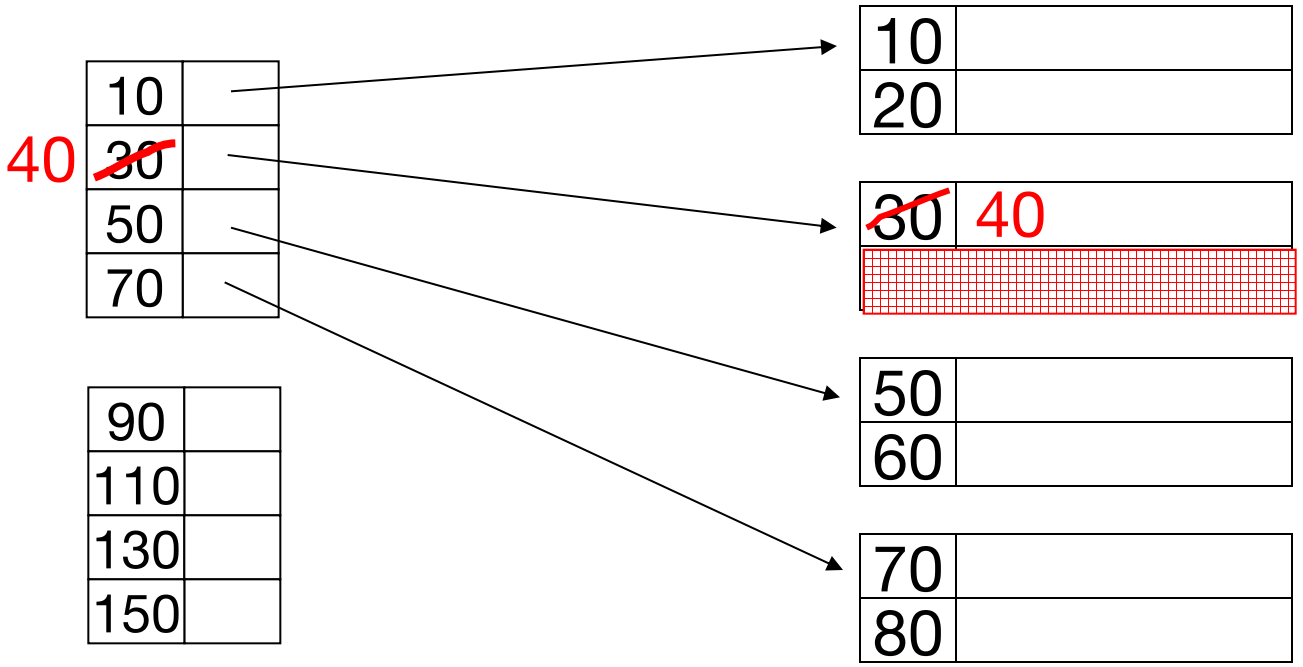
Deletion: Sparse Index

– delete record 30



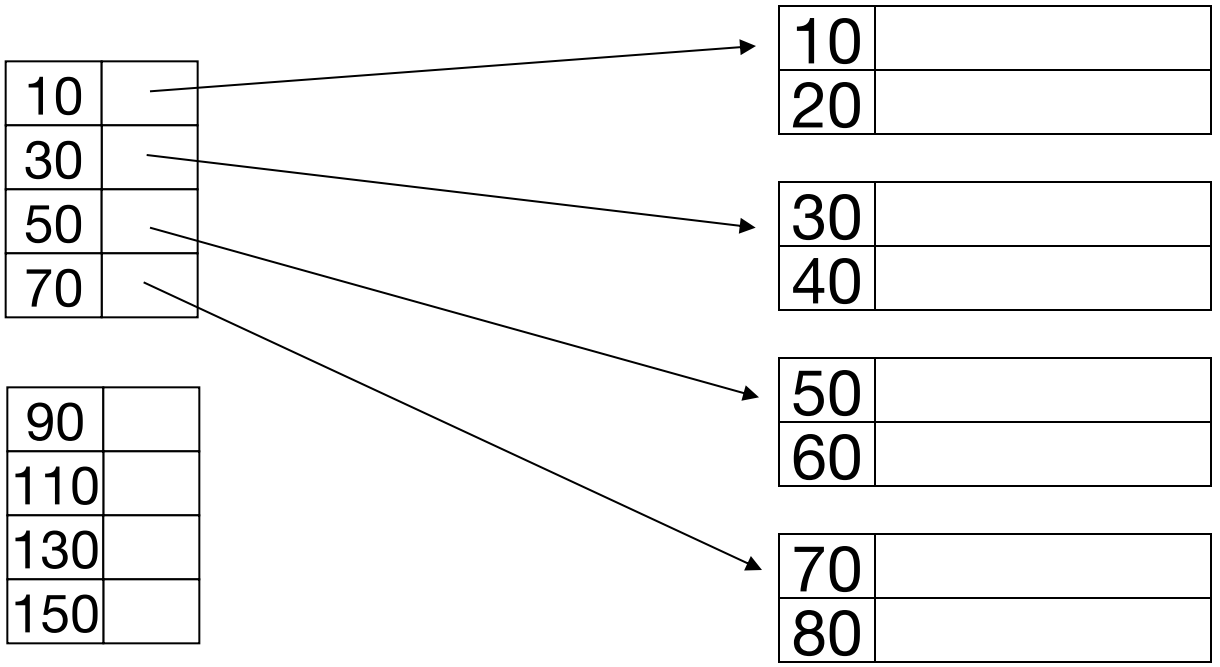
Deletion: Sparse Index

– delete record 30



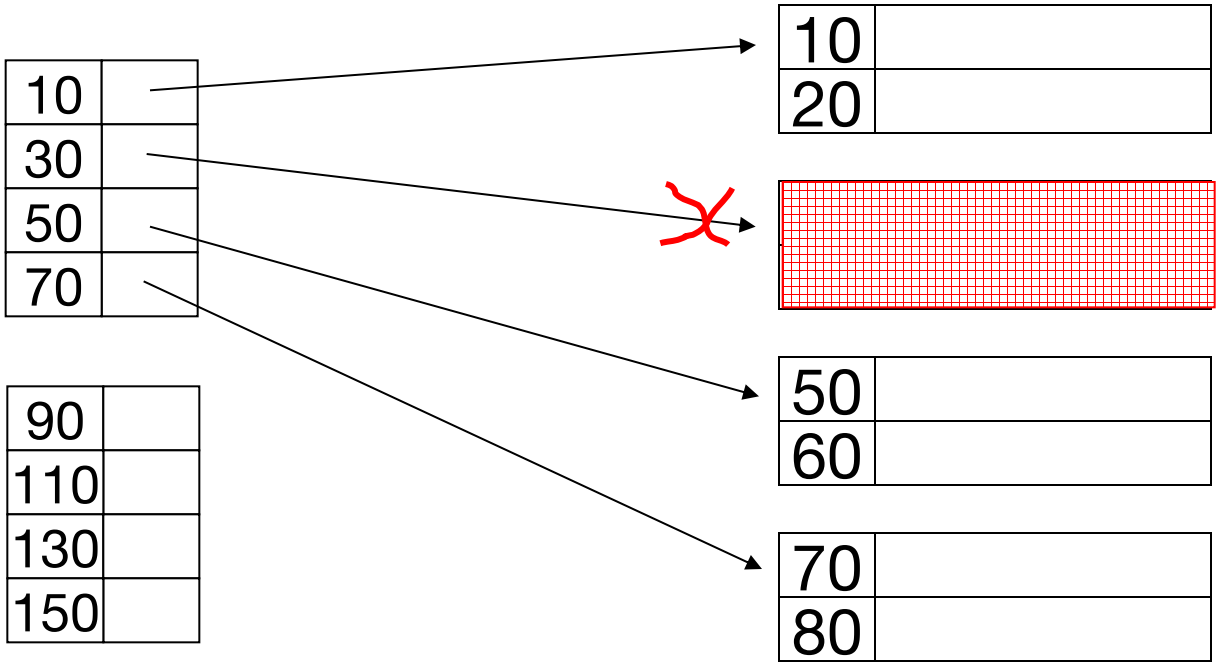
Deletion: Sparse Index

– delete records 30 & 40



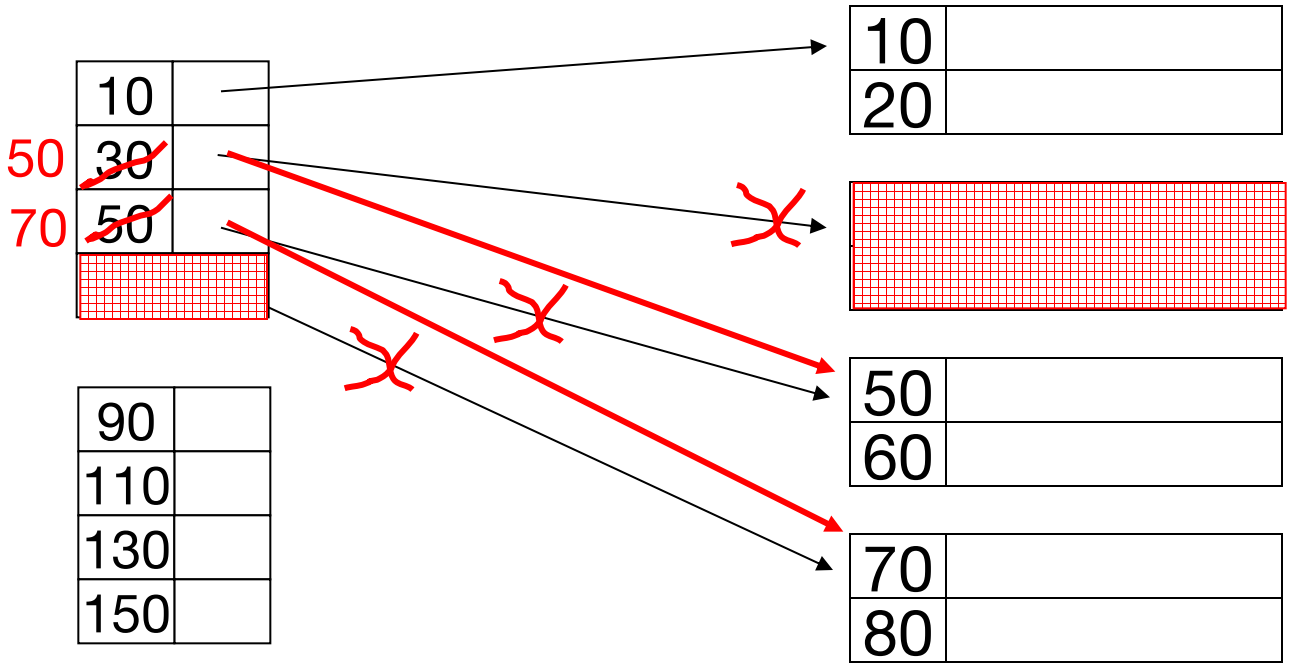
Deletion: Sparse Index

– delete records 30 & 40

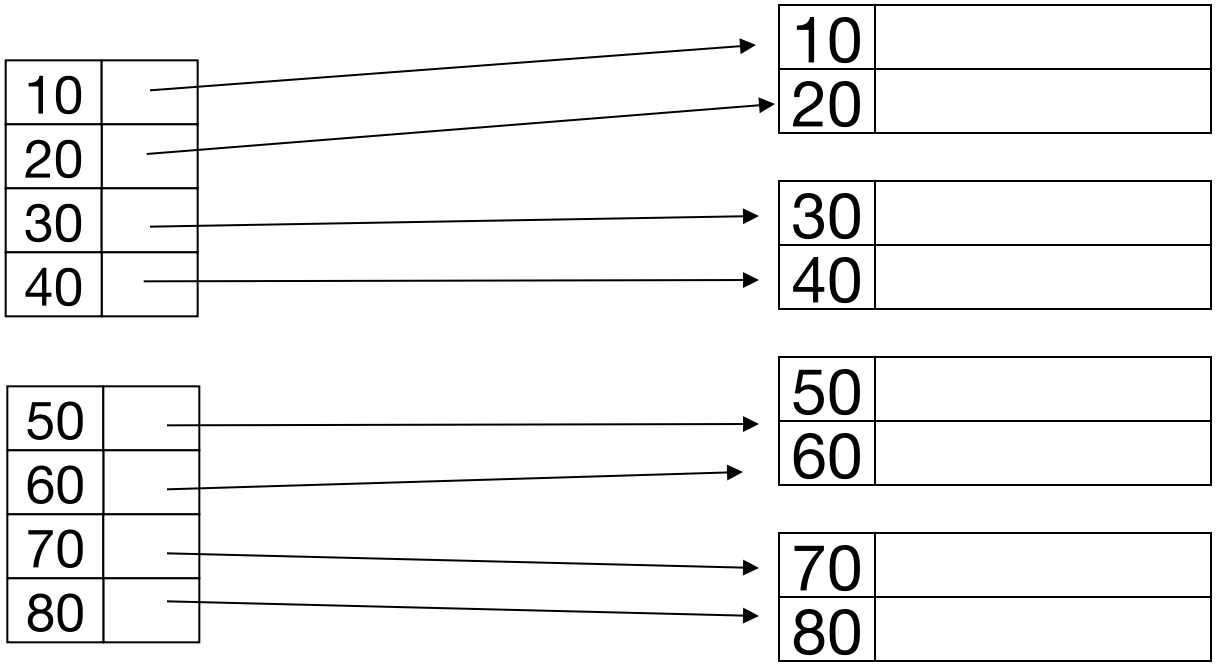


Deletion: Sparse Index

– delete records 30 & 40

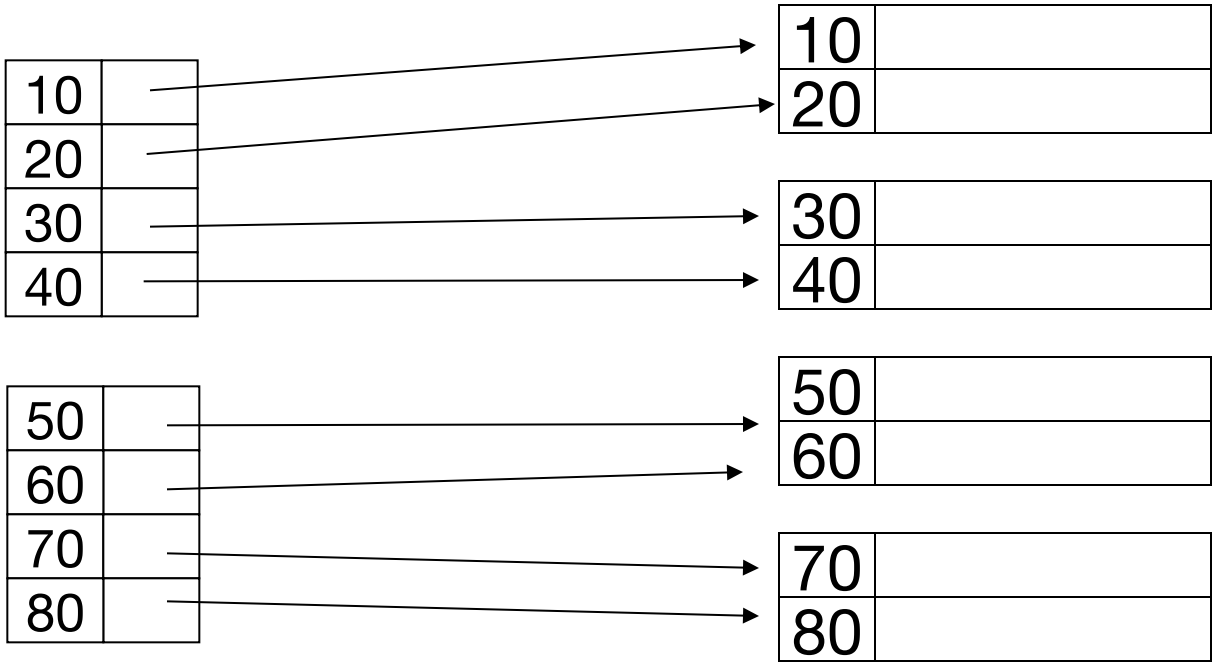


Deletion: Dense Index



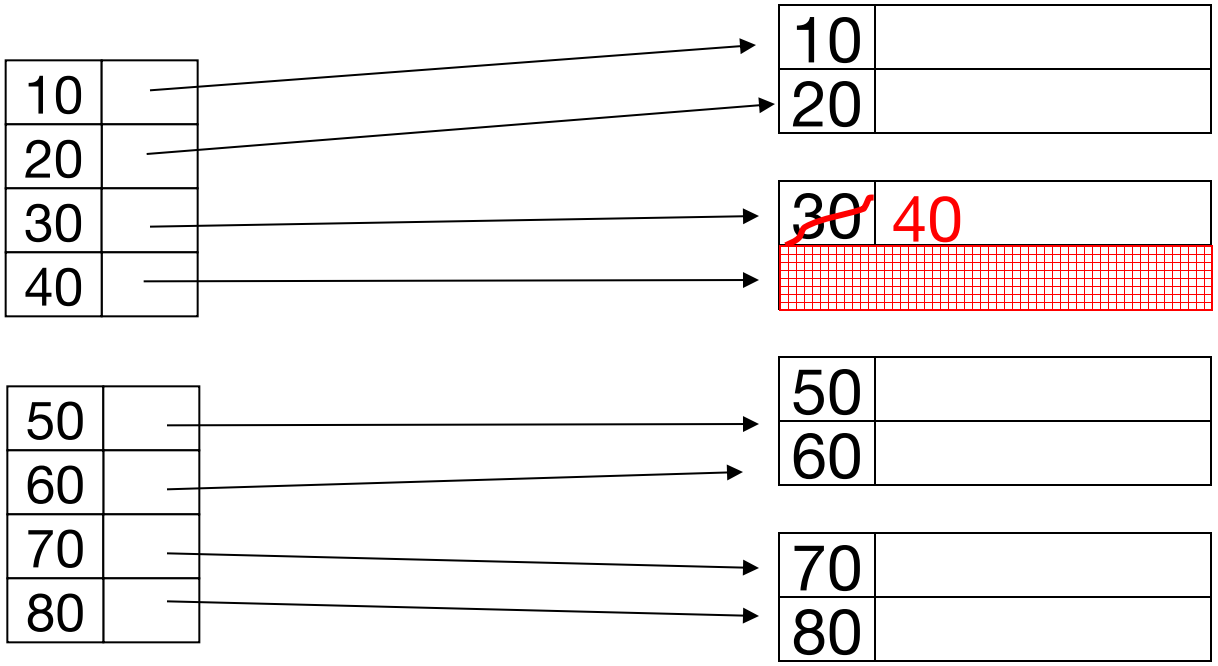
Deletion: Dense Index

– delete record 30



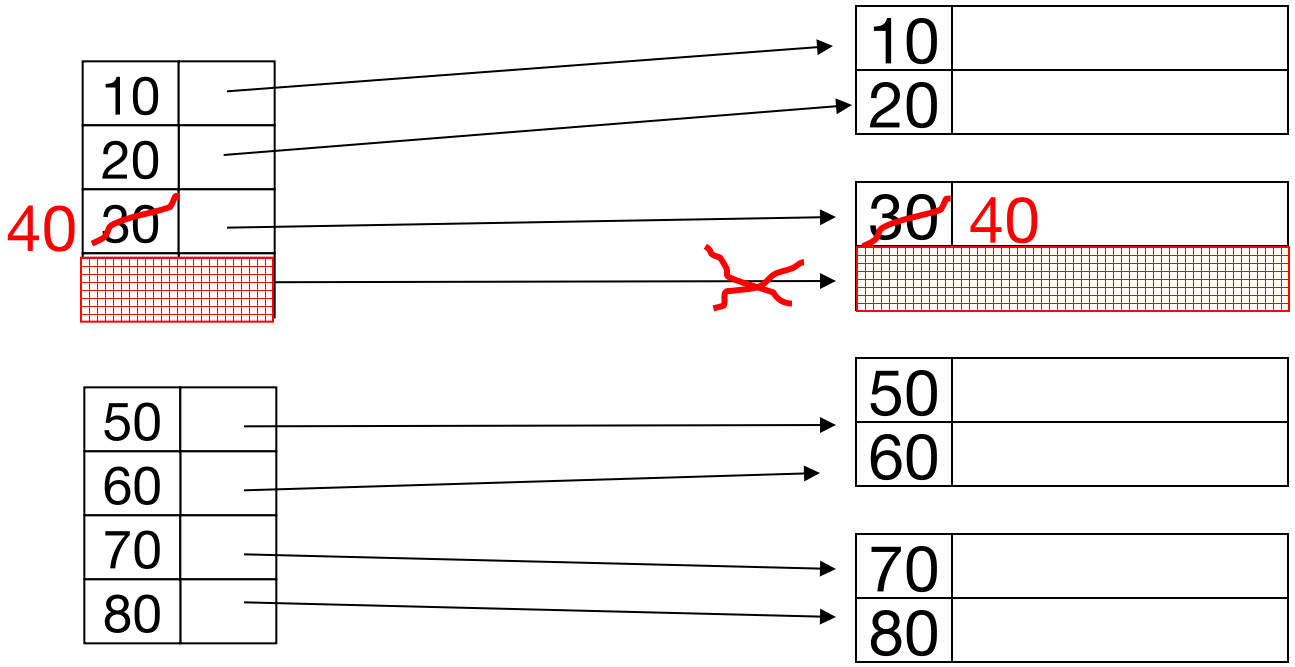
Deletion: Dense Index

– delete record 30



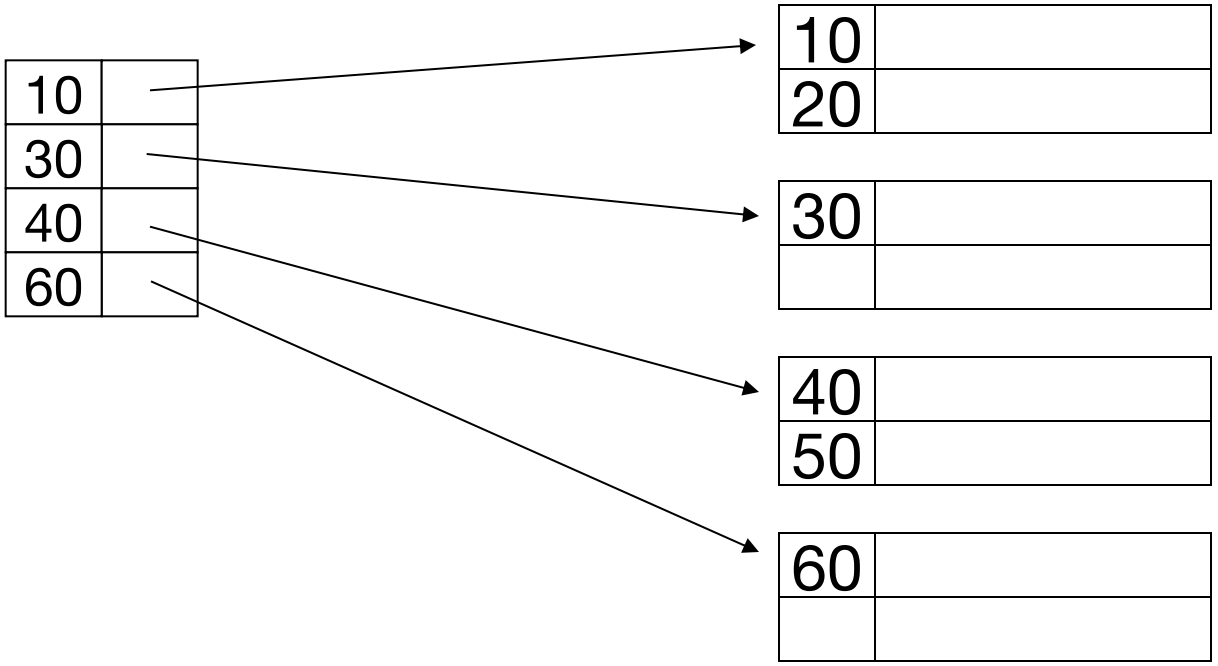
Deletion: Dense Index

– delete record 30



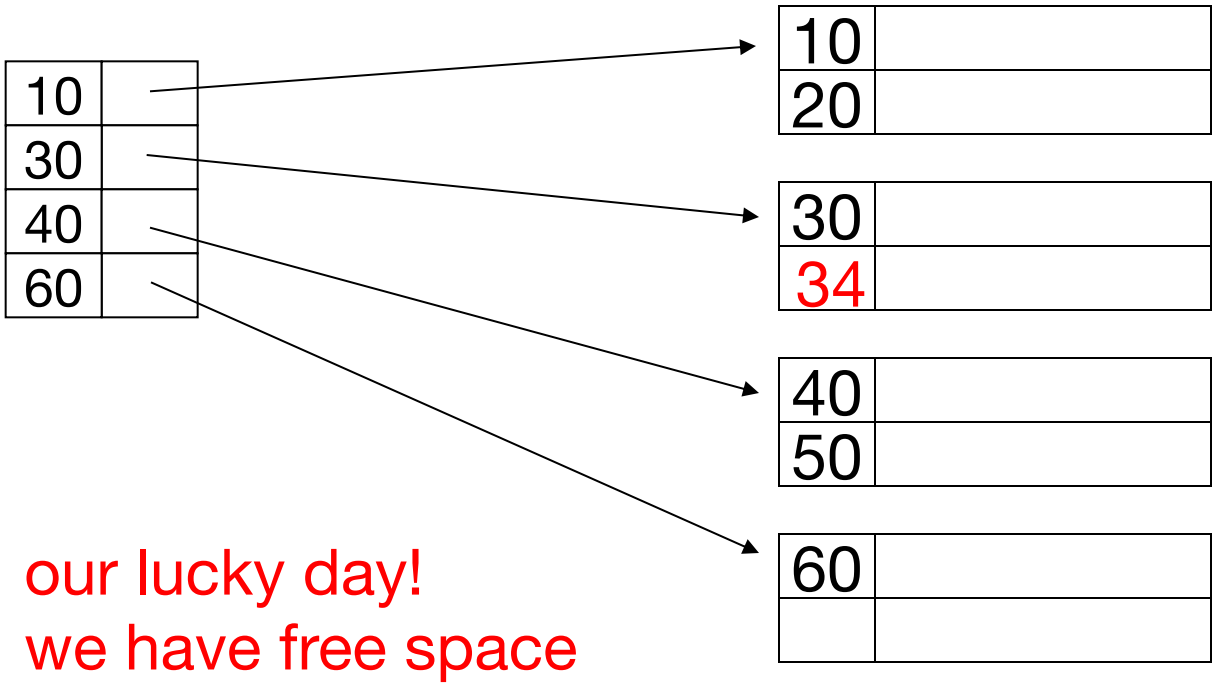
Insertion: Sparse Index

– insert record 34



Insertion: Sparse Index

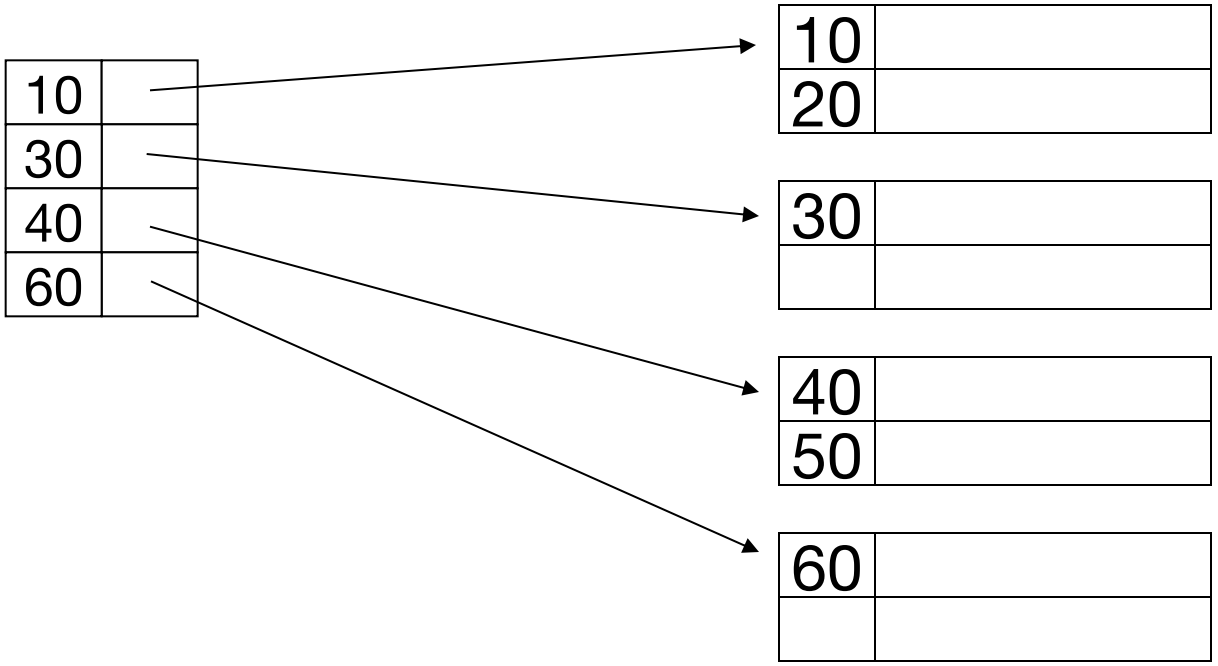
– insert record 34



our lucky day!
we have free space
where we need it!

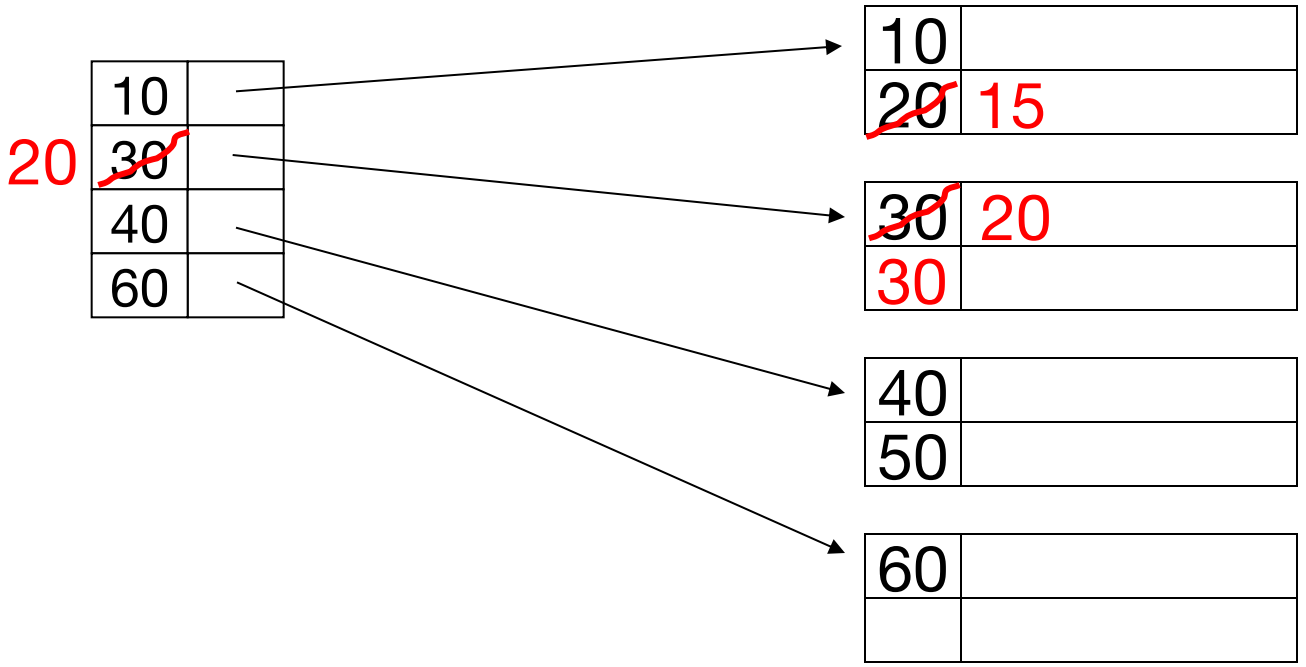
Insertion: Sparse Index

– insert record 15



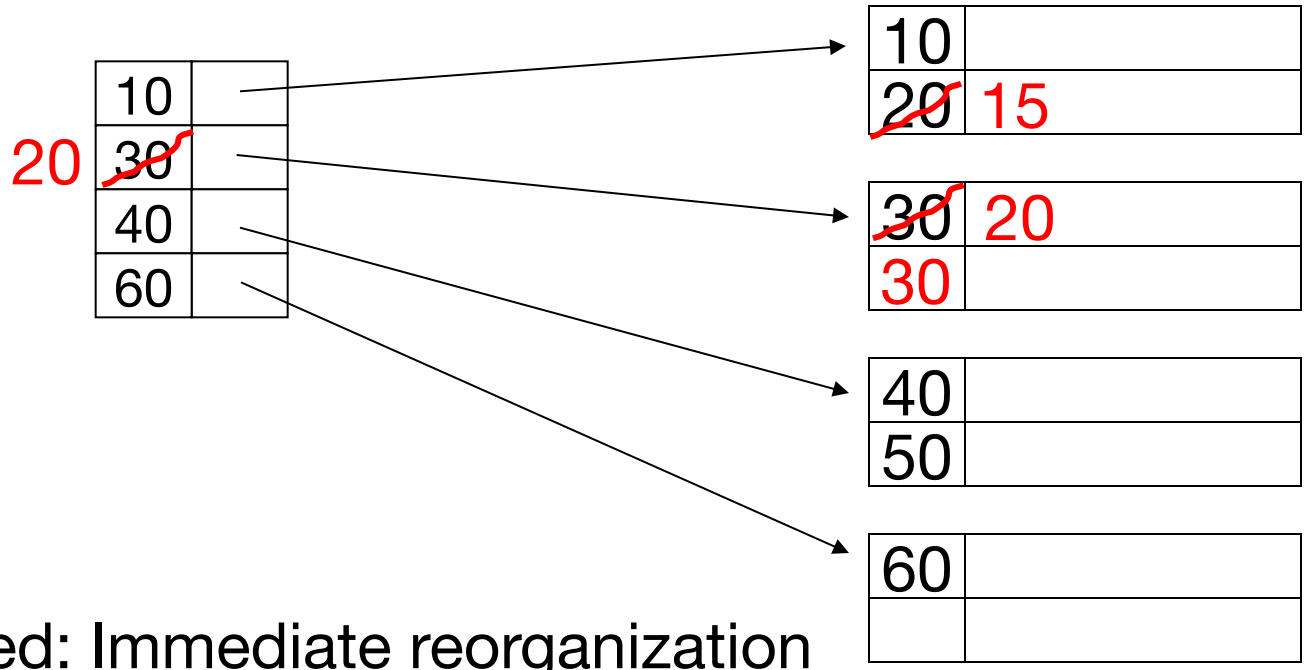
Insertion: Sparse Index

– insert record 15



Insertion: Sparse Index

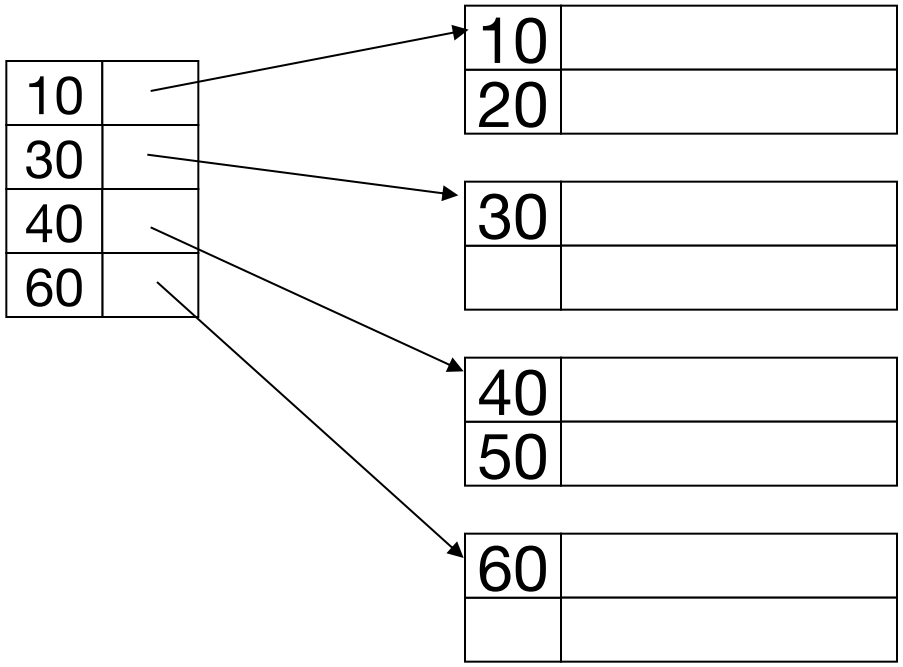
– insert record 15



- Illustrated: Immediate reorganization
- Variation:
  - insert new block (chained file)
  - update index

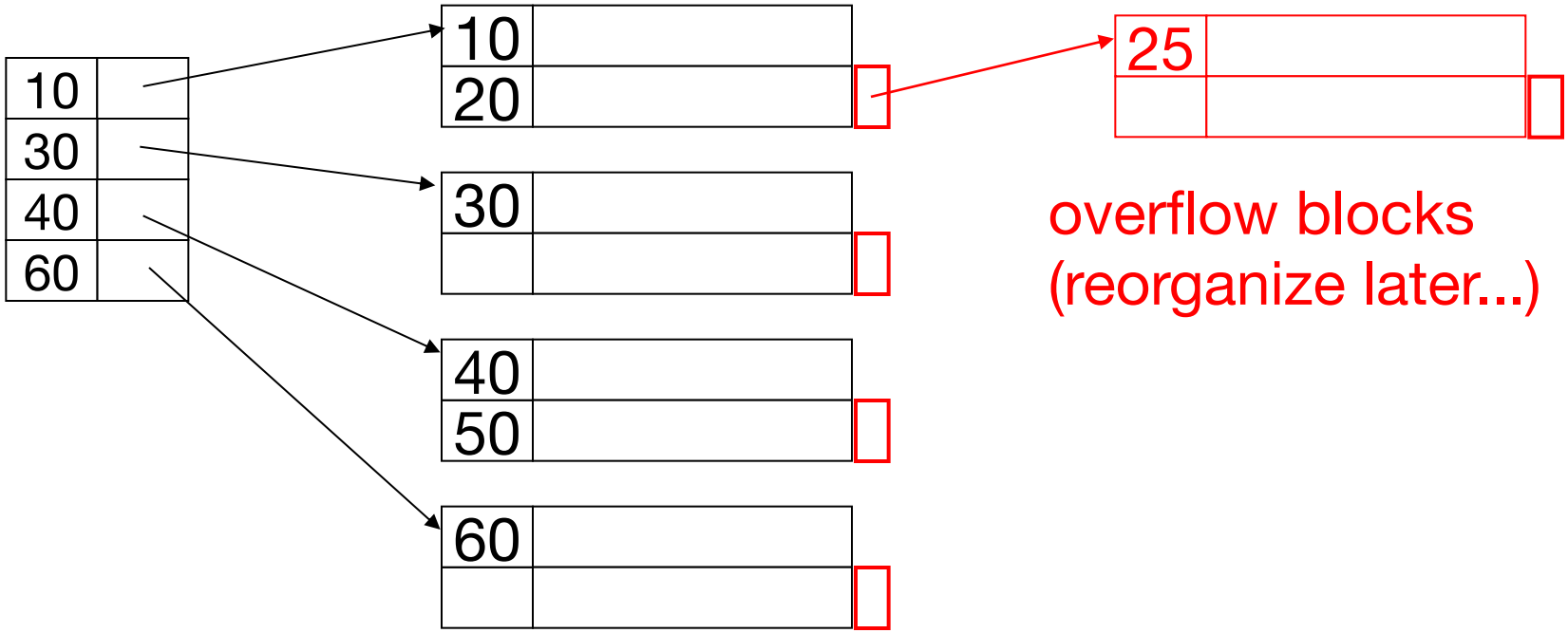
Insertion: Sparse Index

– insert record 25

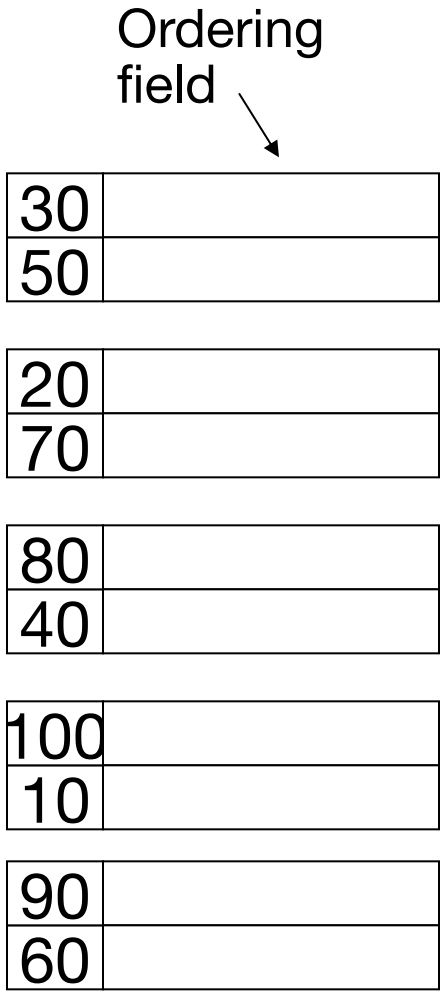


Insertion: Sparse Index

– insert record 25

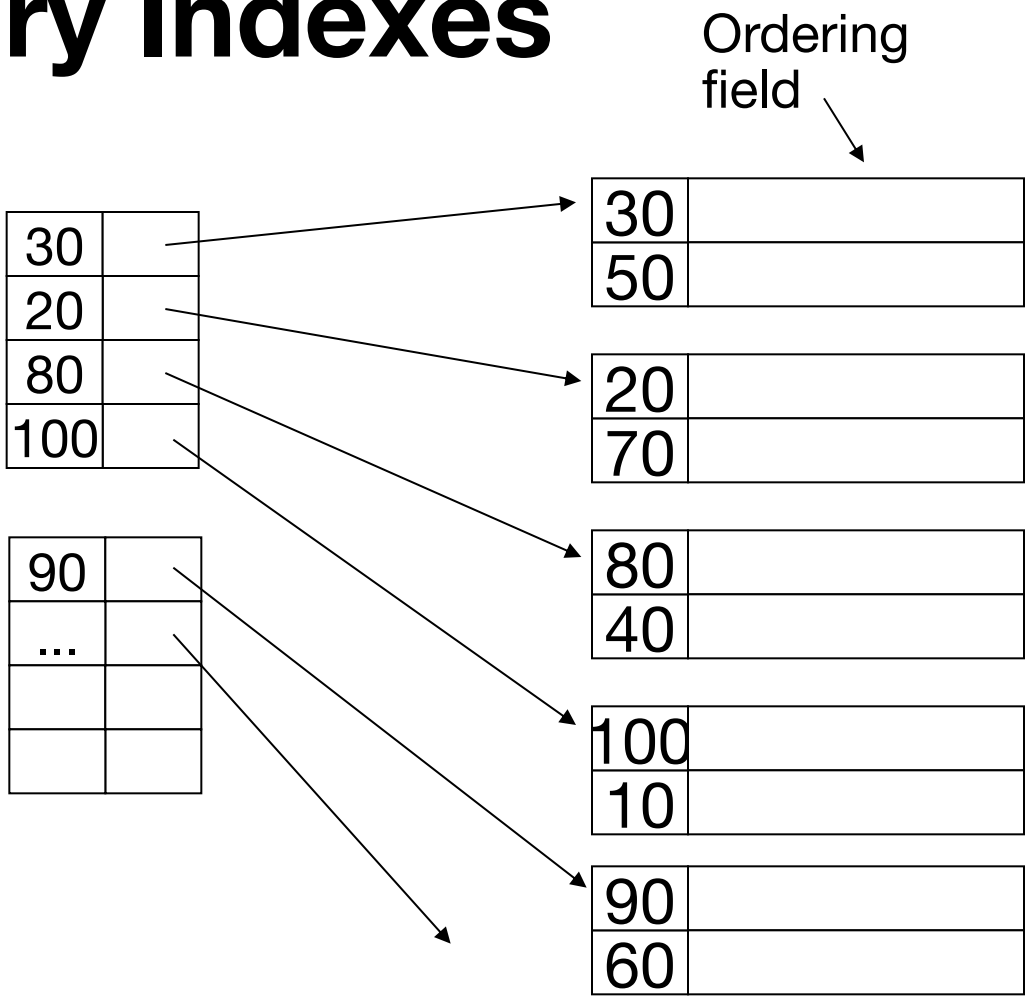


Secondary Indexes



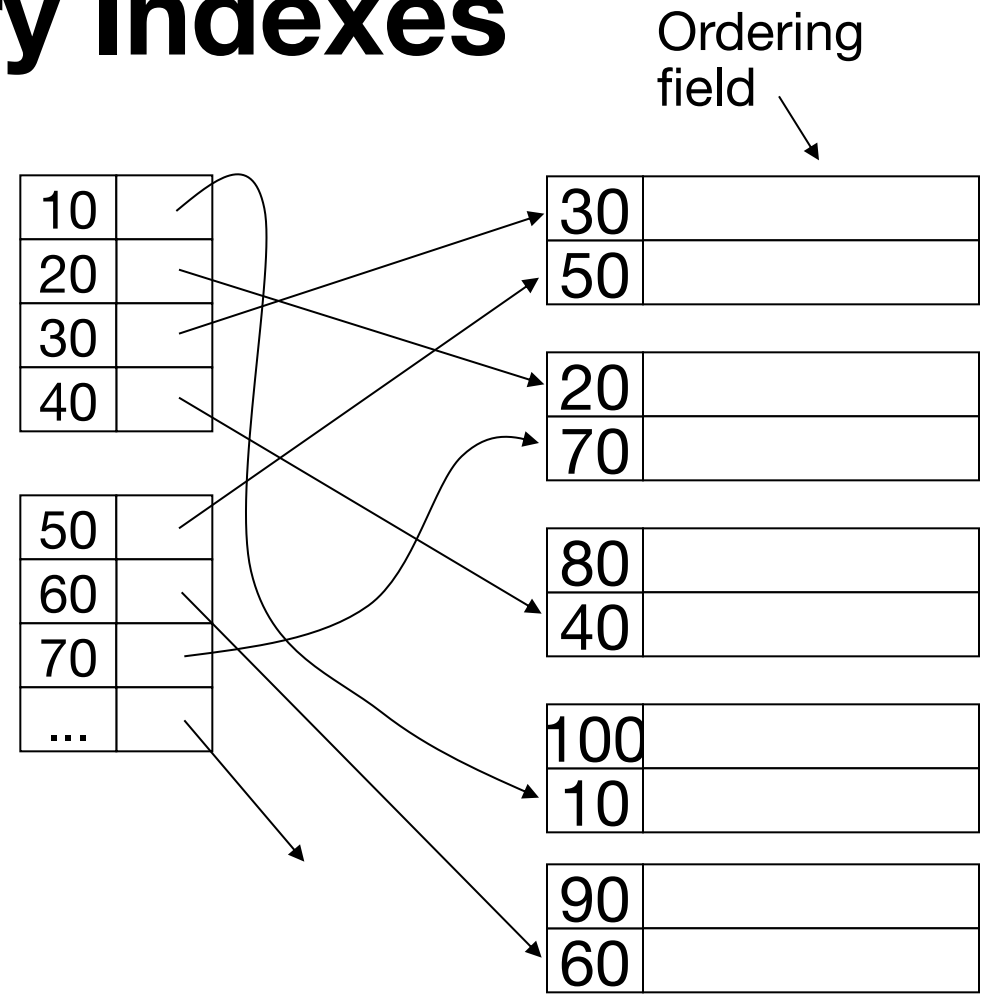
Secondary Indexes

Sparse index:

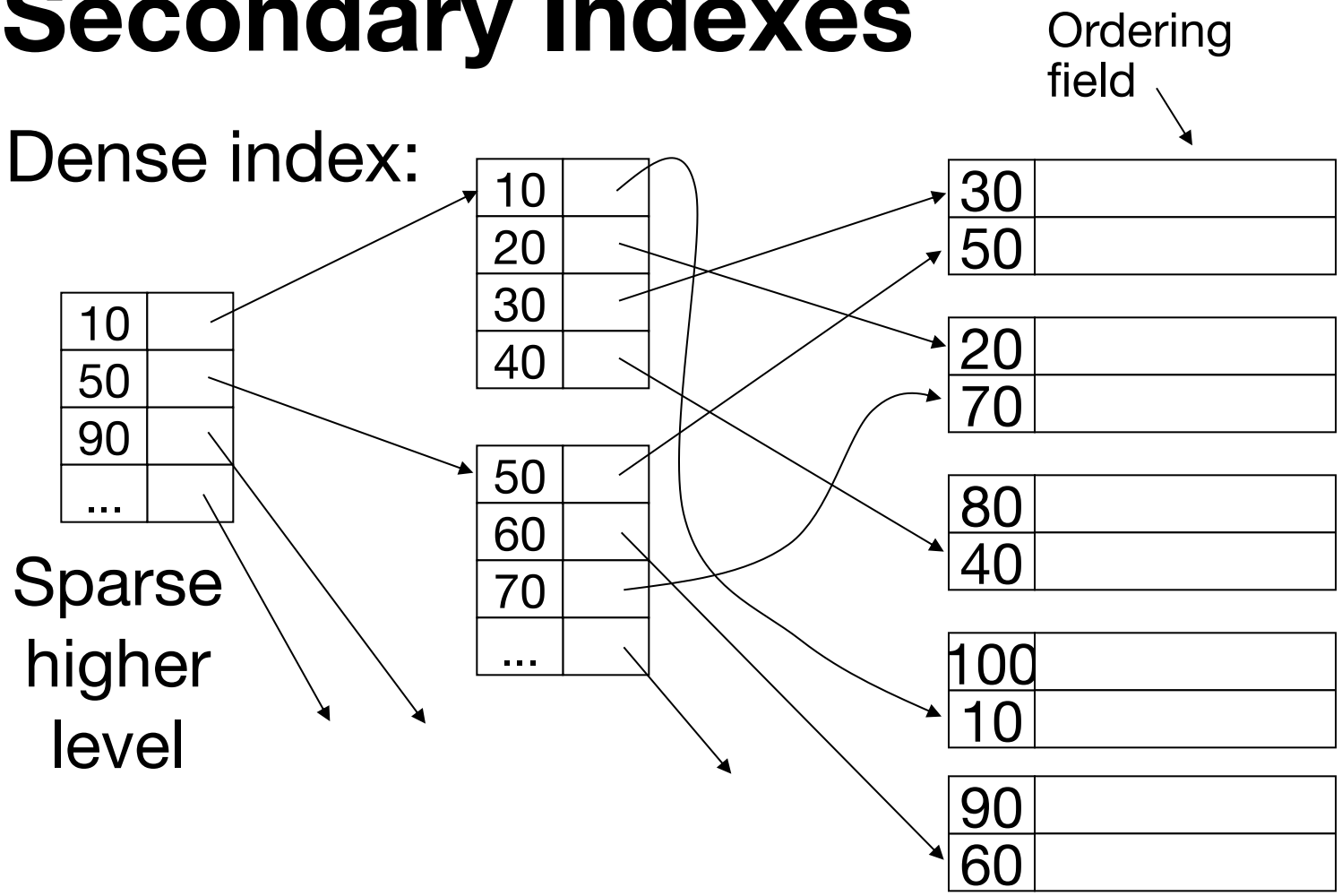


Secondary Indexes

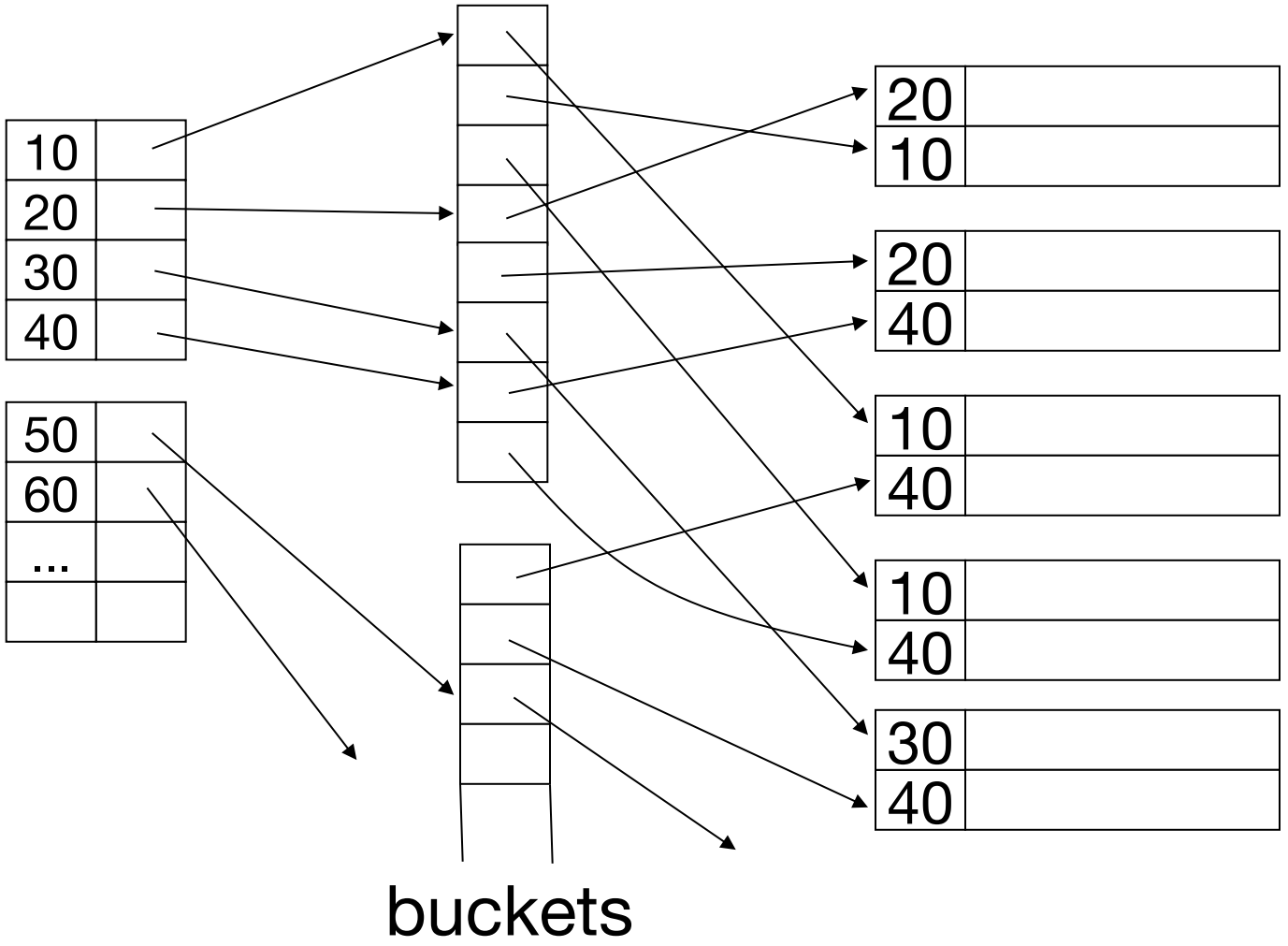
Dense index:



Secondary Indexes



Duplicate Values in Secondary Indexes



Conventional Indexes

Pros:

- Simple
- Index is sequential file (good for scans)

Cons:

- Inserts expensive, and/or
- Lose sequentiality & balance

Some Types of Indexes

Conventional indexes

B-trees

Hash indexes

Multi-key indexing

B-Trees

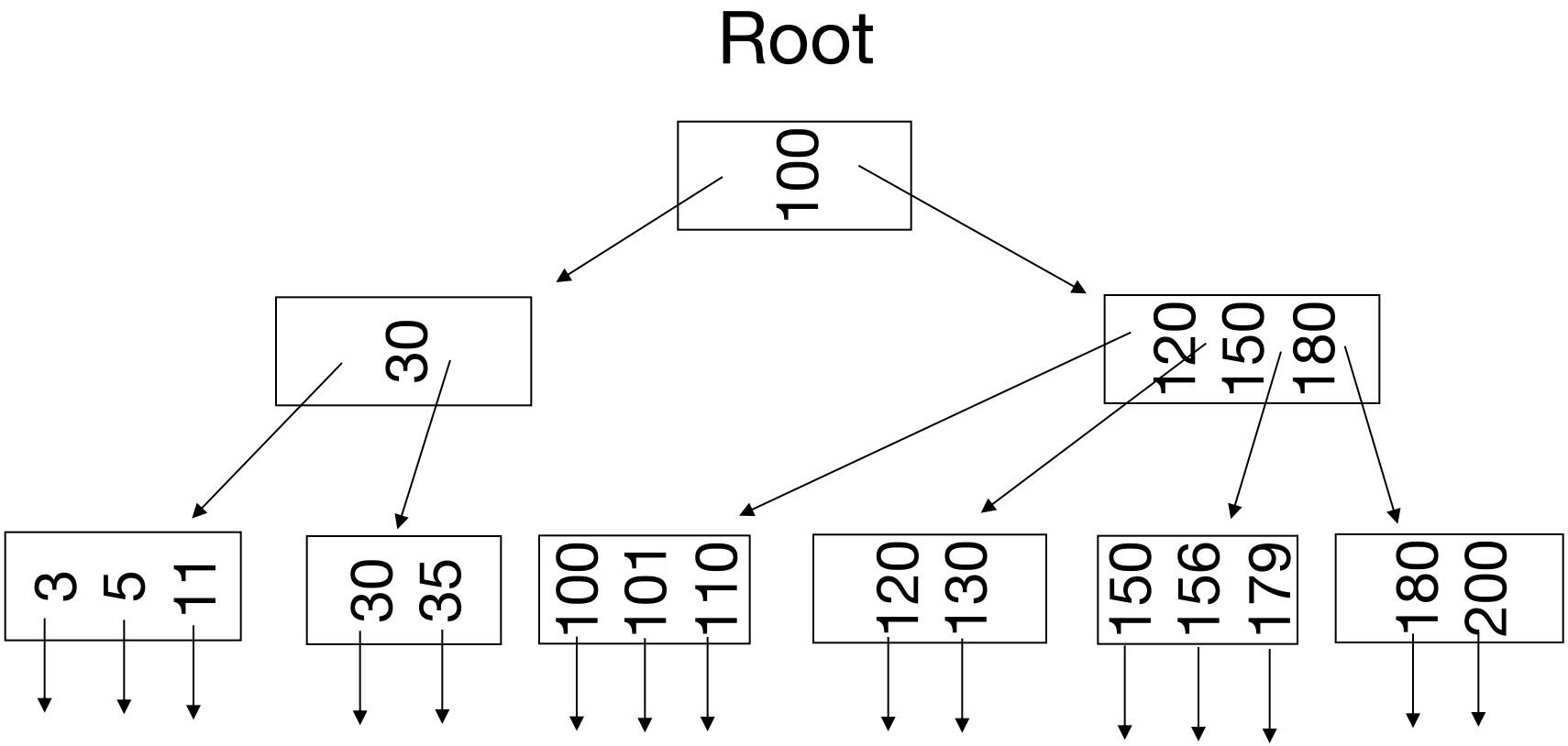
Another type of index

» Give up on sequentiality of index

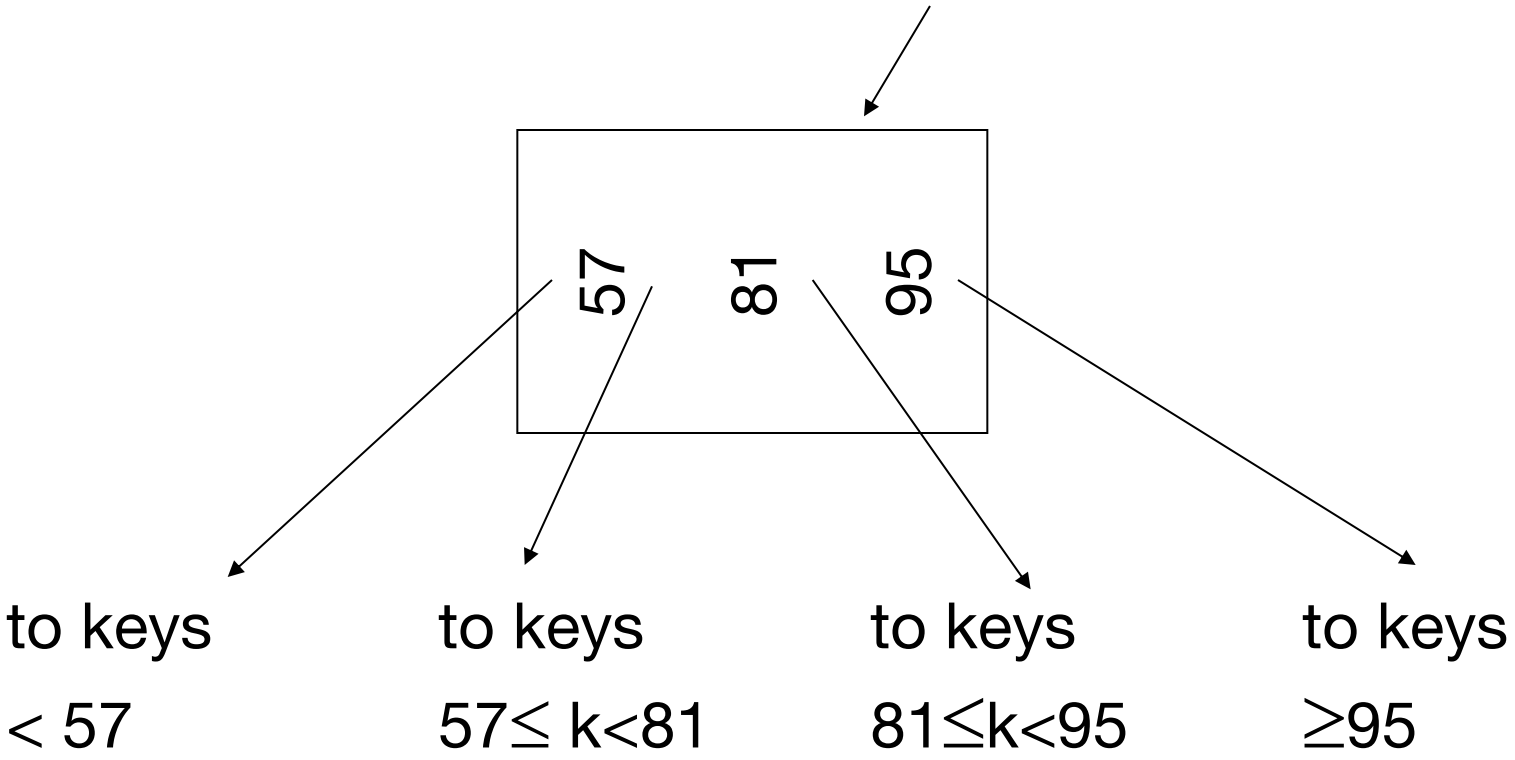
» Try to get “balance”

Note: the exact data structure we’ll look at is a **B+ tree**, but plain old “B-trees” are similar

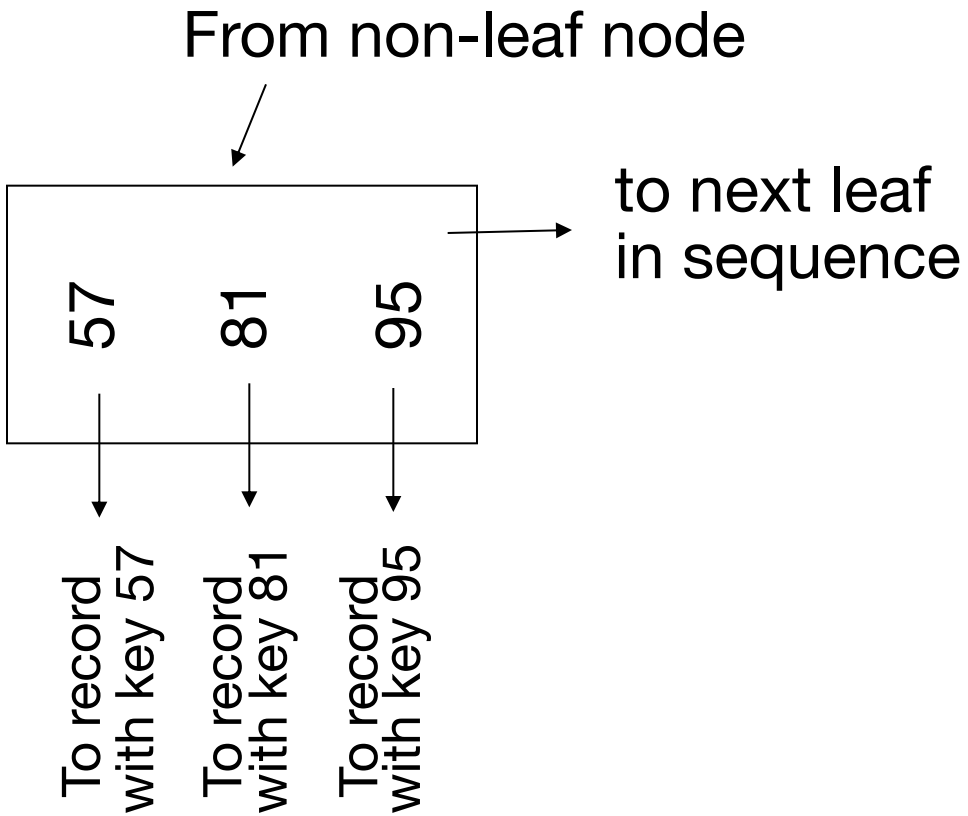
B+ Tree Example (n = 3)



Sample Non-Leaf



Sample Leaf Node



Size of Nodes on Disk

- n + 1 pointers
- n keys

(Fixed size nodes)

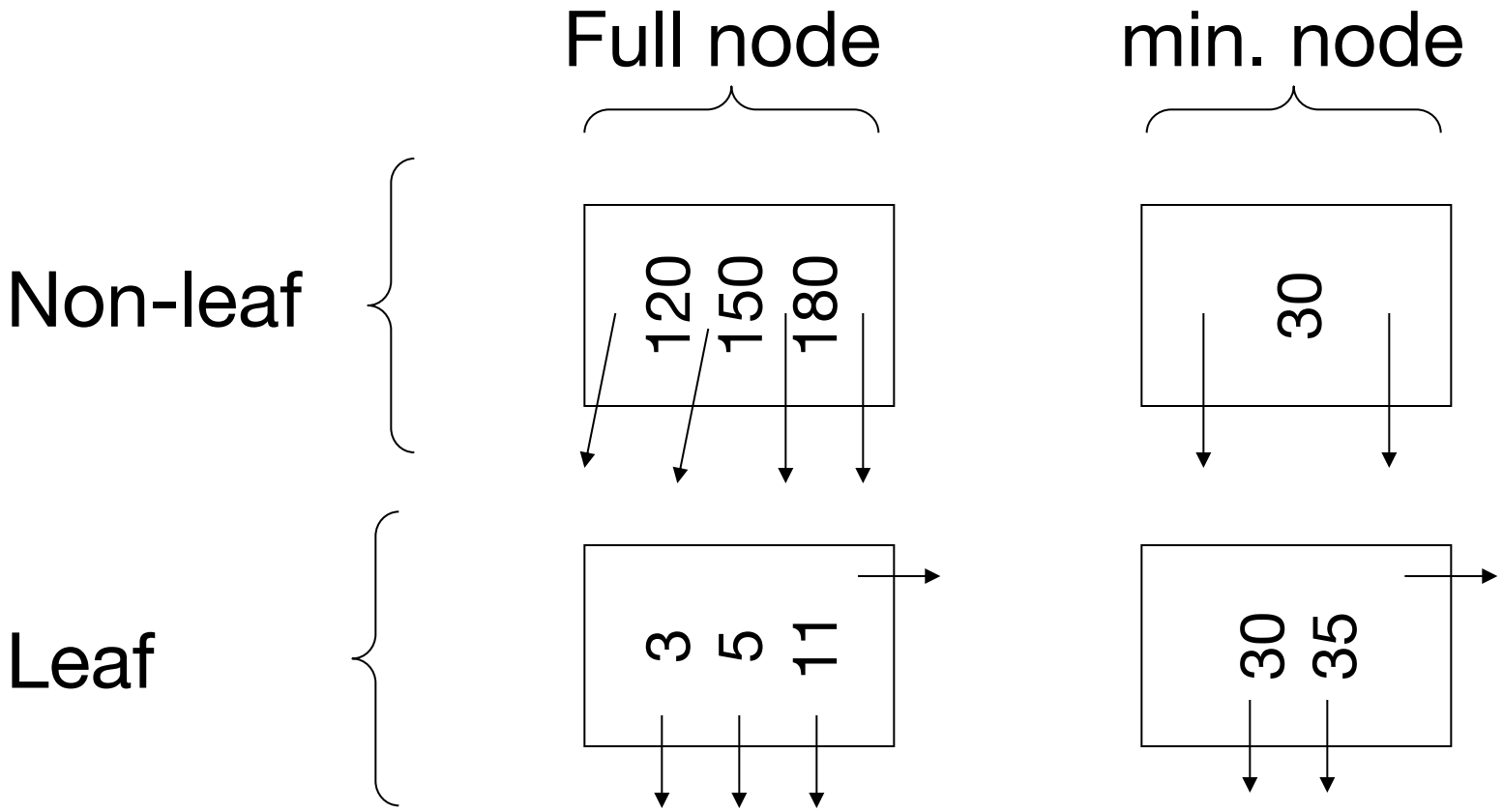
Don’t Want Nodes to be Too Empty

Use at least

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

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

Example: n = 3



B+ Tree Rules (tree of order n)

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

2. Pointers in leaves point to records, except for “sequence pointer”

B+ Tree Rules (tree of order n)

(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	2*	1

* When there is only one record in the B+ tree, min pointers in the root is 1 (the other pointers are null)

Insert Into B+ Tree

(a) simple case
  » space available in leaf

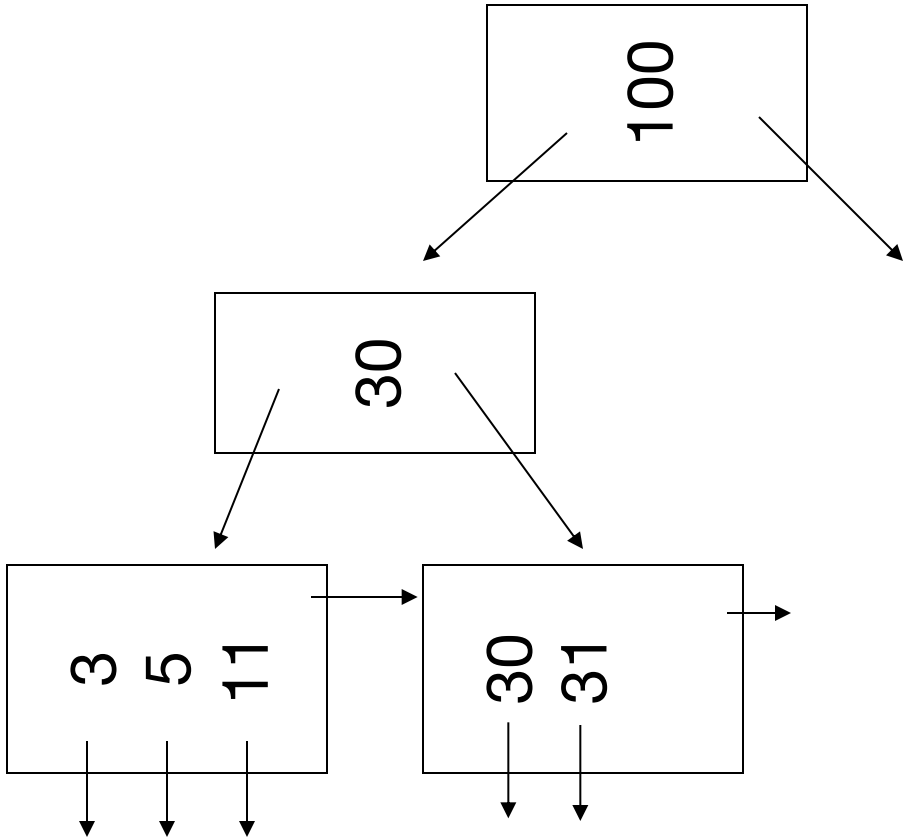
(b) leaf overflow

(c) non-leaf overflow

(d) new root

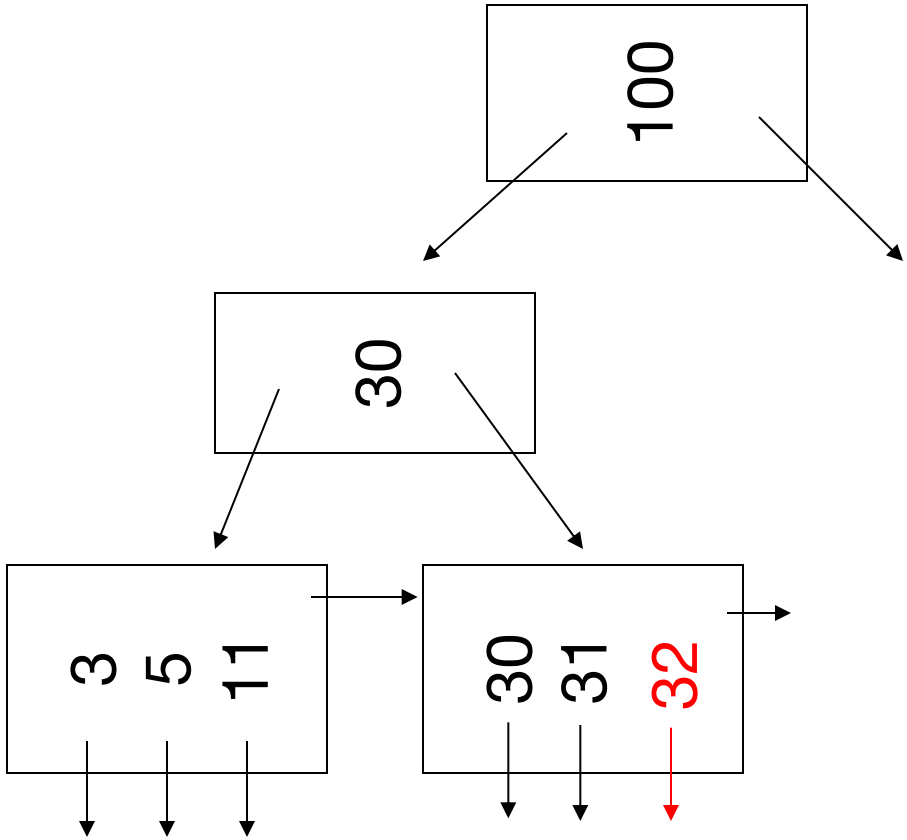
(a) Insert key = 32

n=3



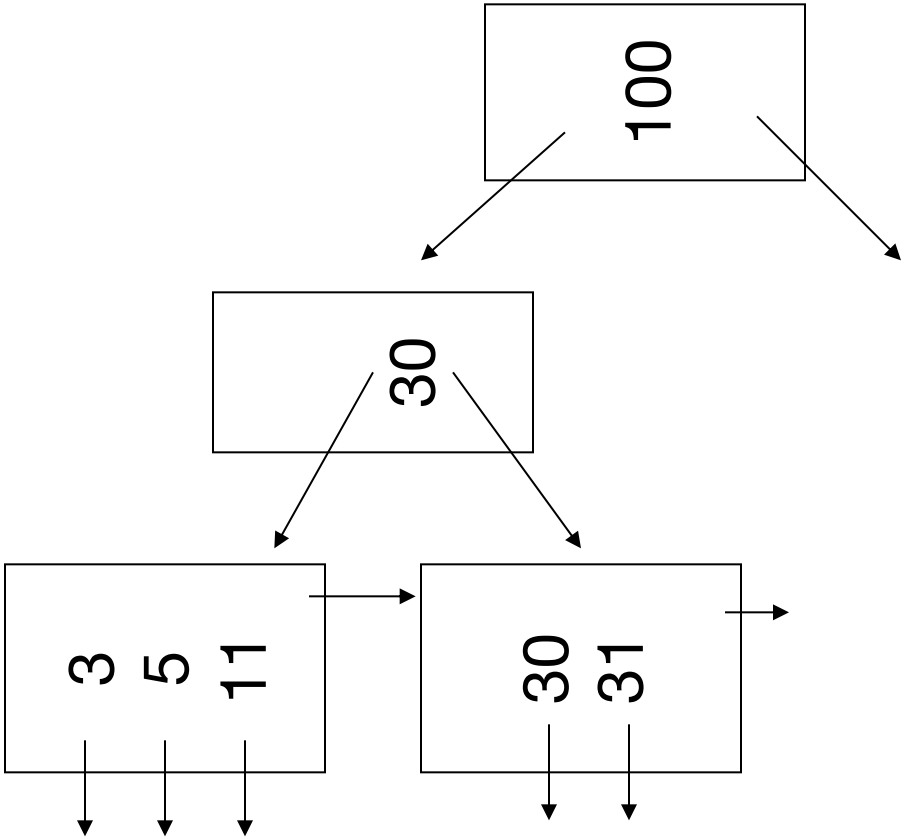
(a) Insert key = 32

n=3



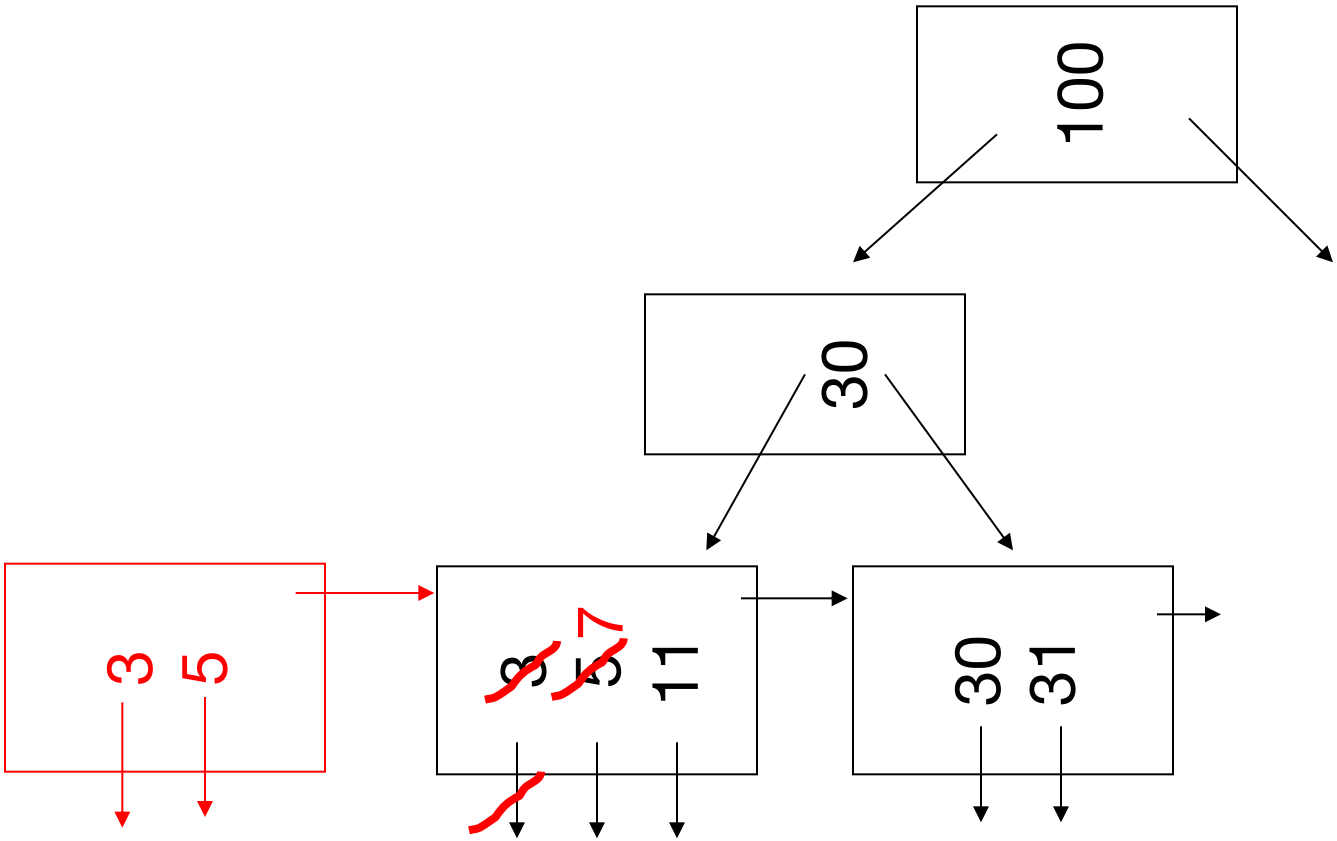
(a) Insert key = 7

n=3



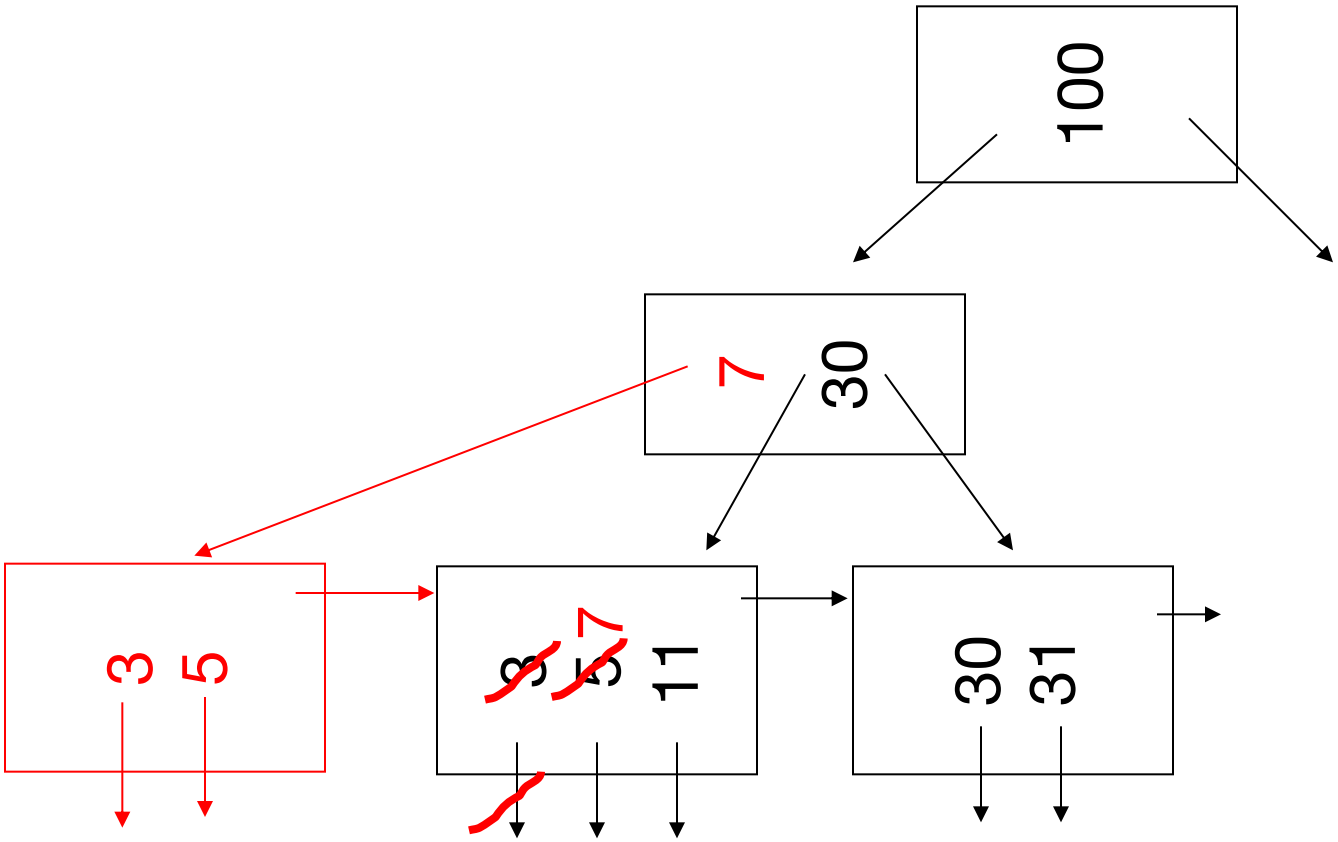
(a) Insert key = 7

n=3



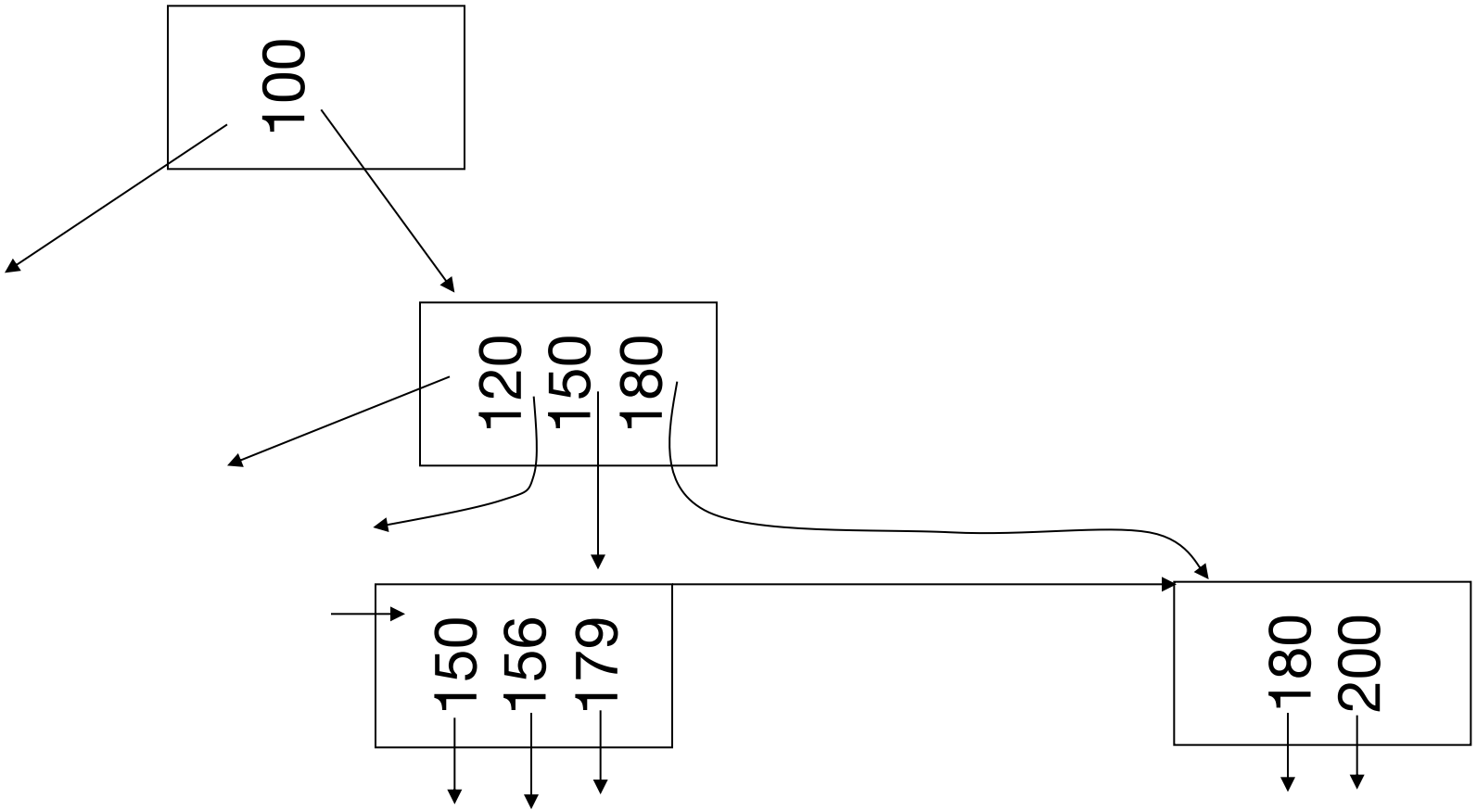
(a) Insert key = 7

n=3



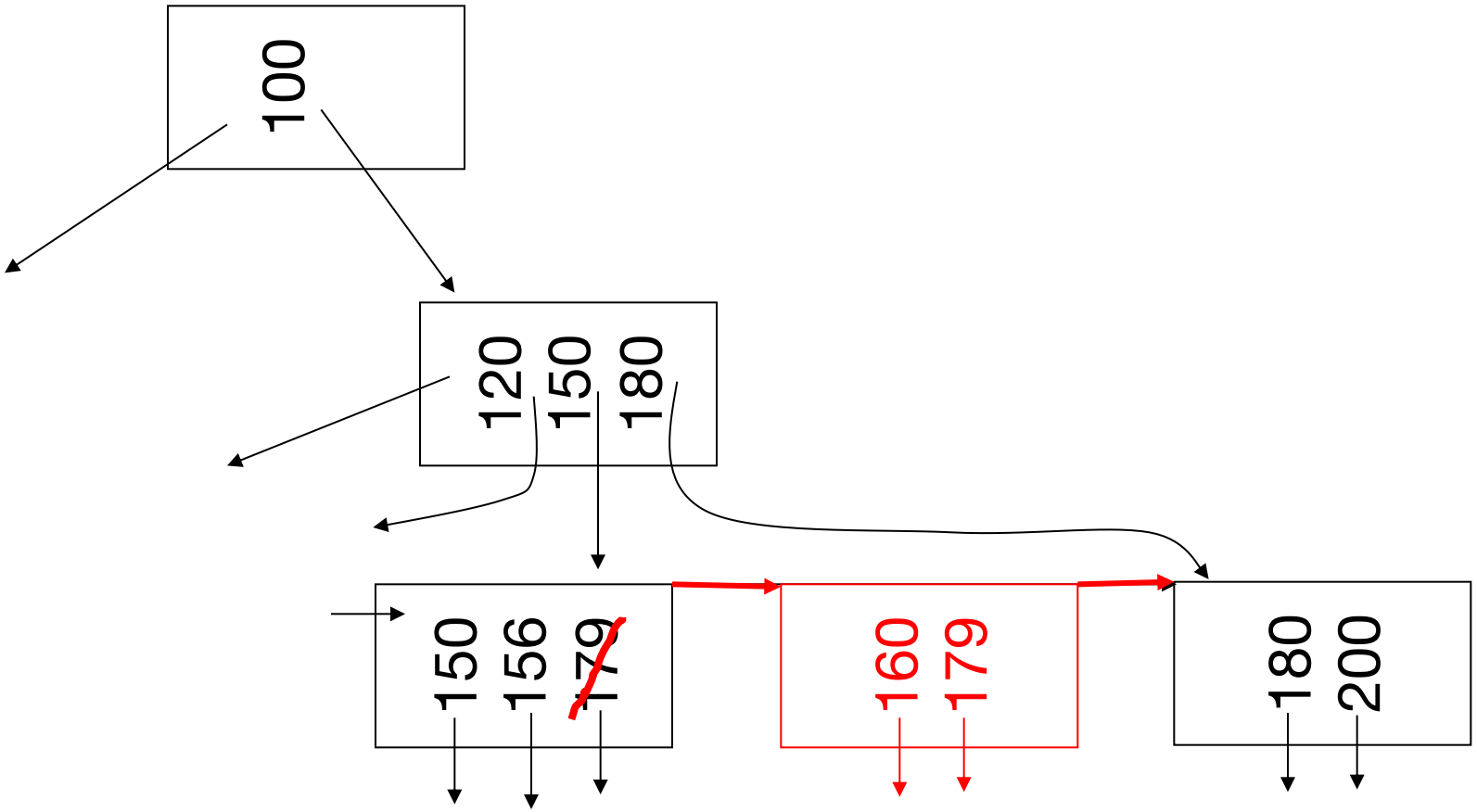
(c) Insert key = 160

n=3



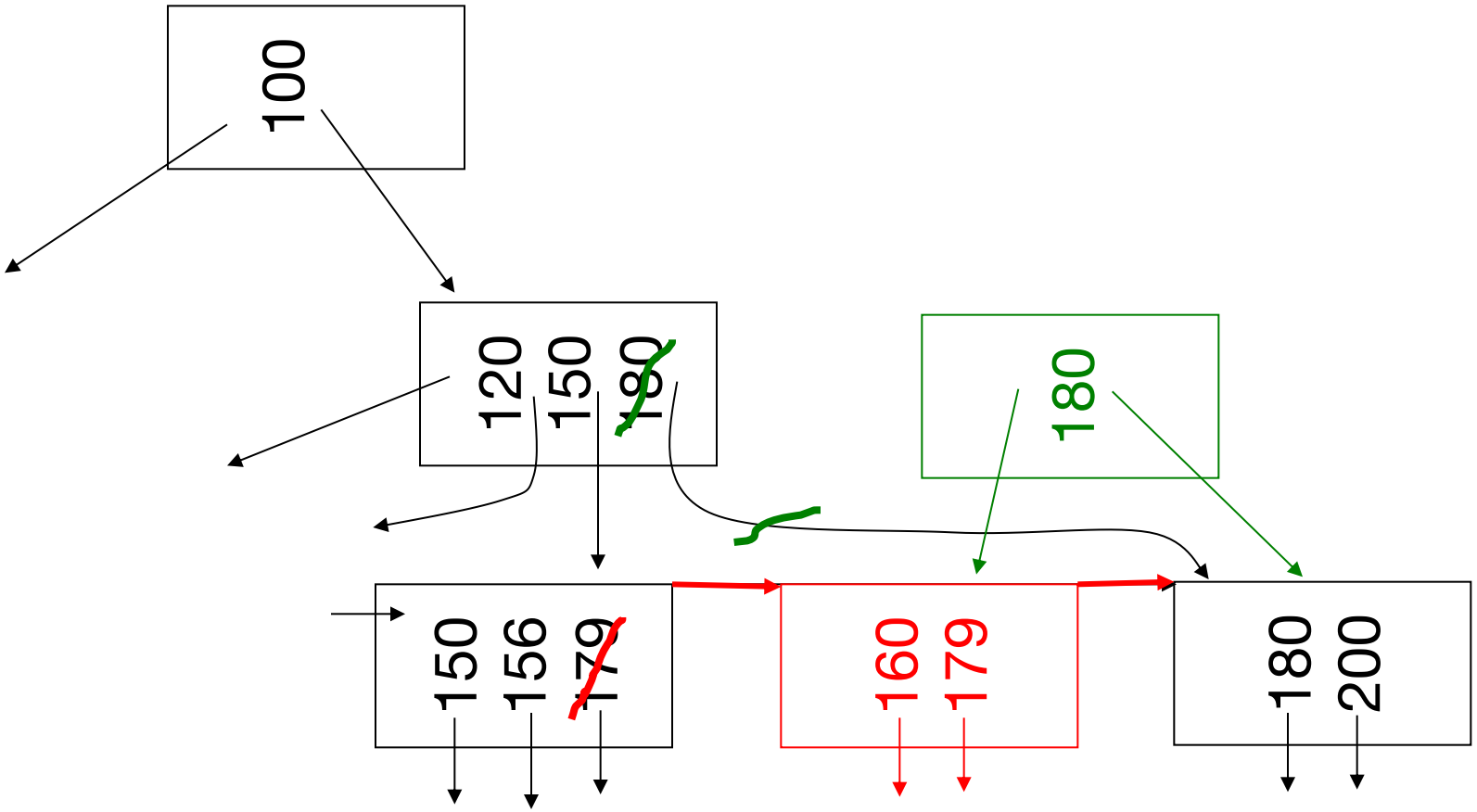
(c) Insert key = 160

n=3



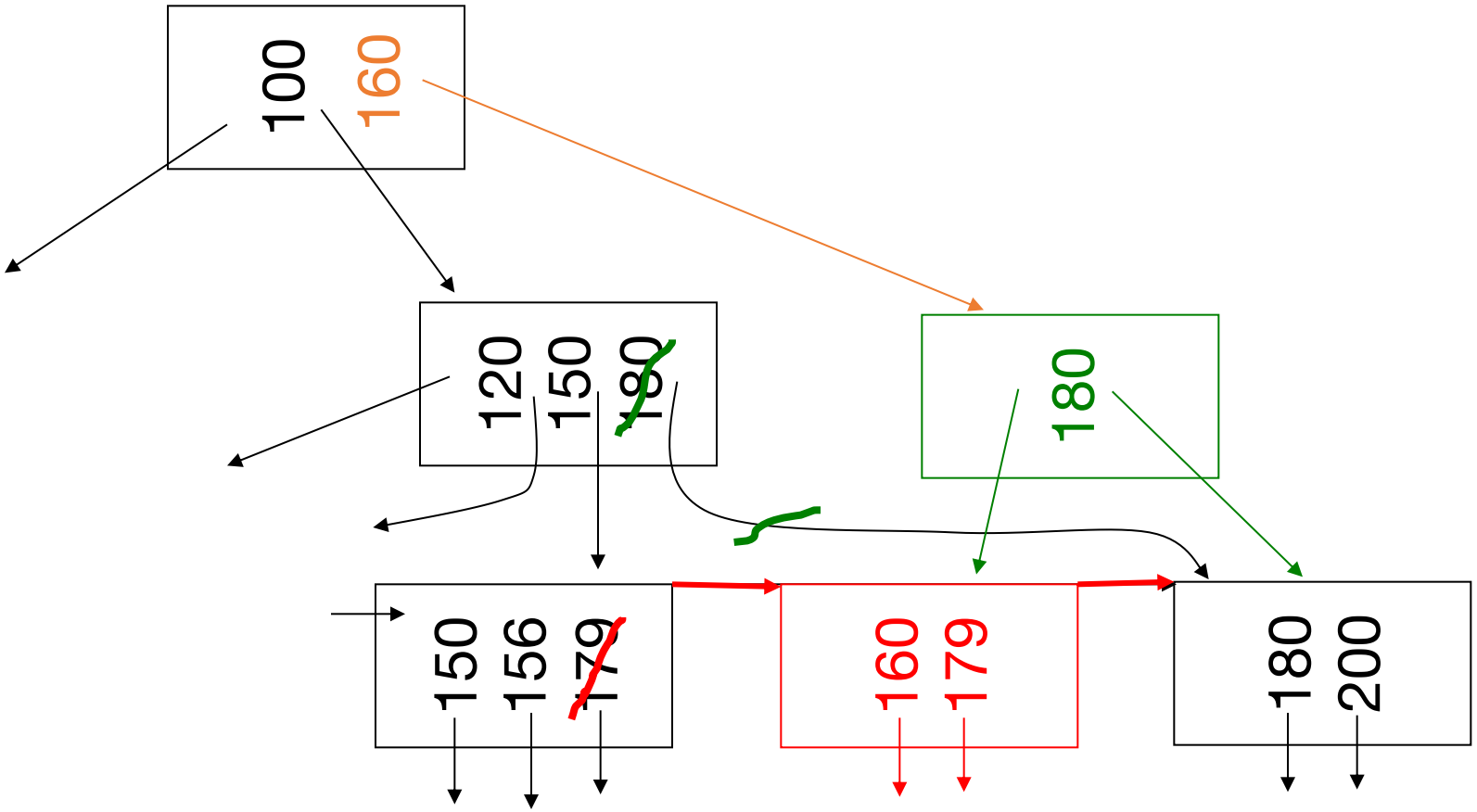
(c) Insert key = 160

n=3



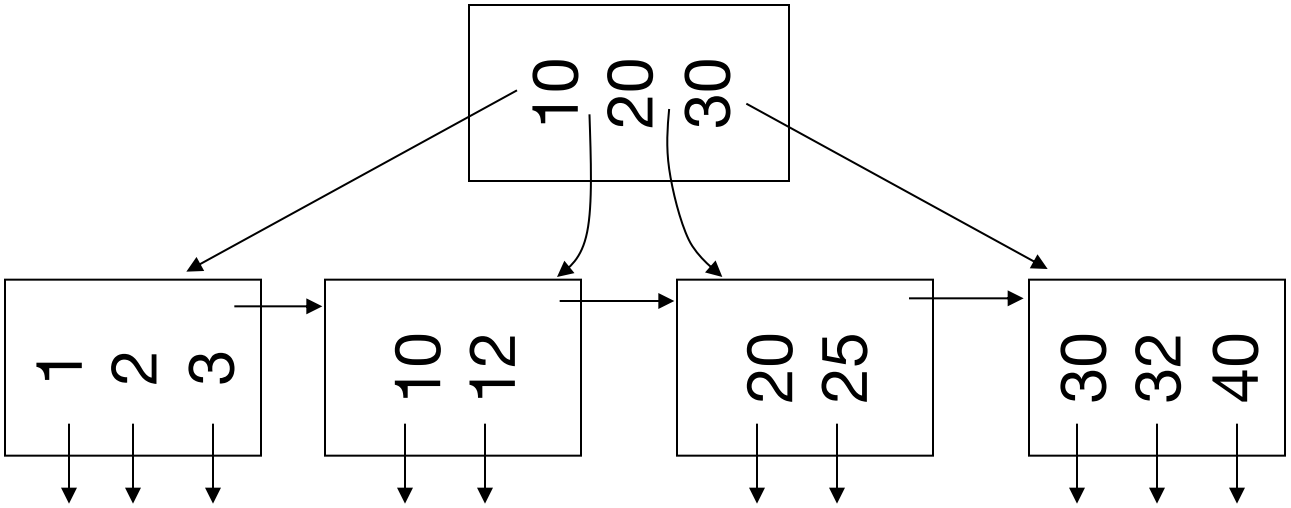
(c) Insert key = 160

n=3



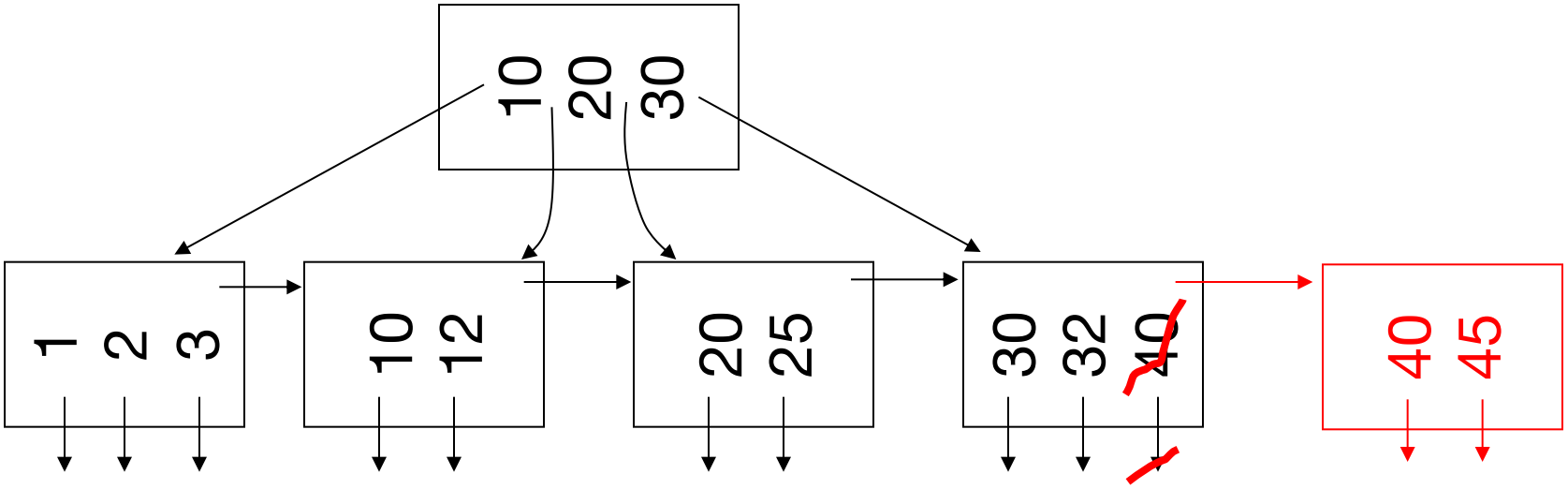
(d) New root, insert 45

n=3



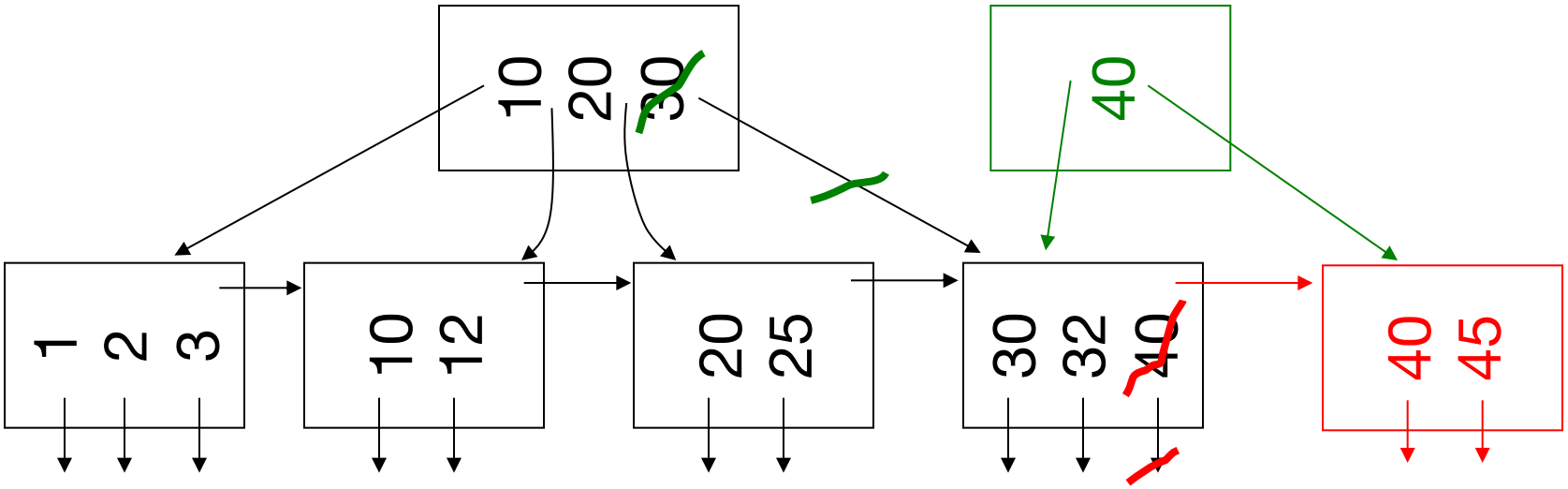
(d) New root, insert 45

n=3



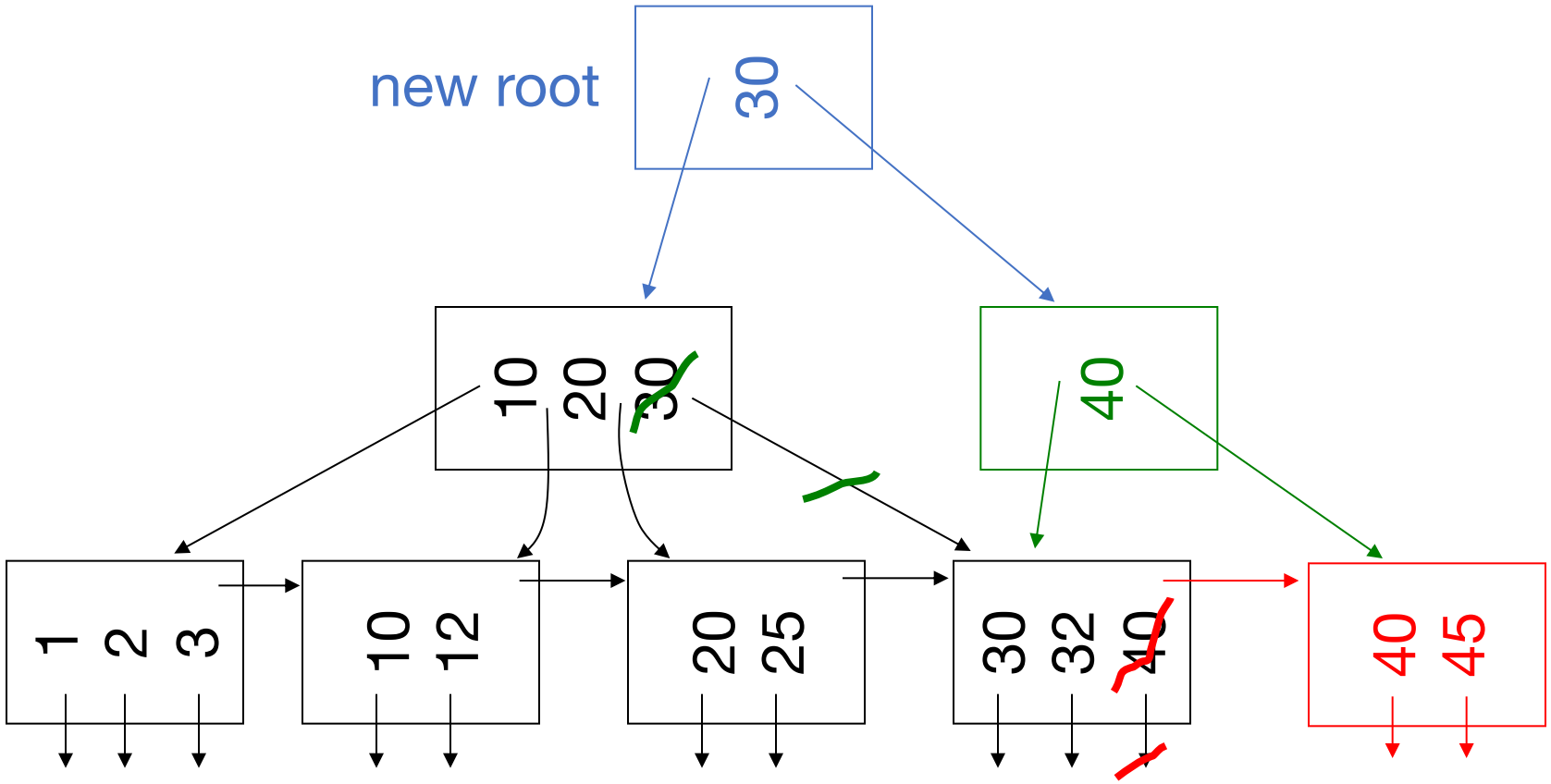
(d) New root, insert 45

n=3



(d) New root, insert 45

n=3



Deletion from B+tree

(a) Simple case: no example

(b) Coalesce with neighbor (sibling)

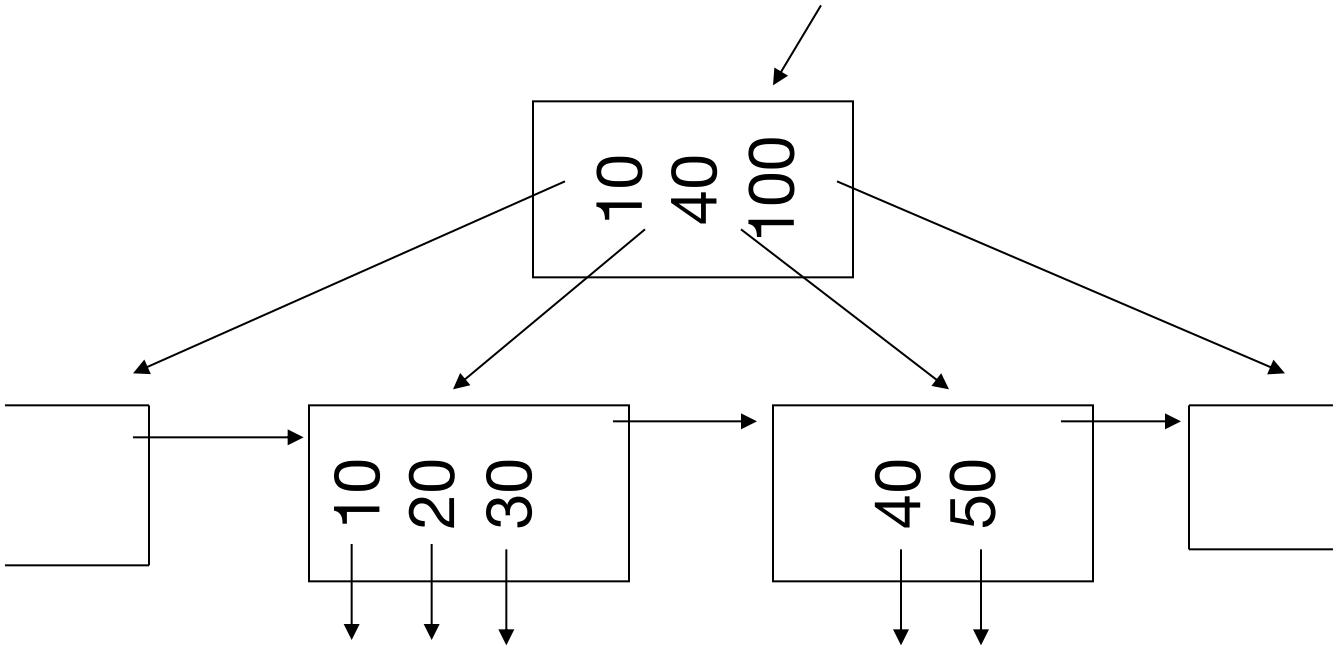
(c) Re-distribute keys

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

(b) Coalesce with sibling

» Delete 50

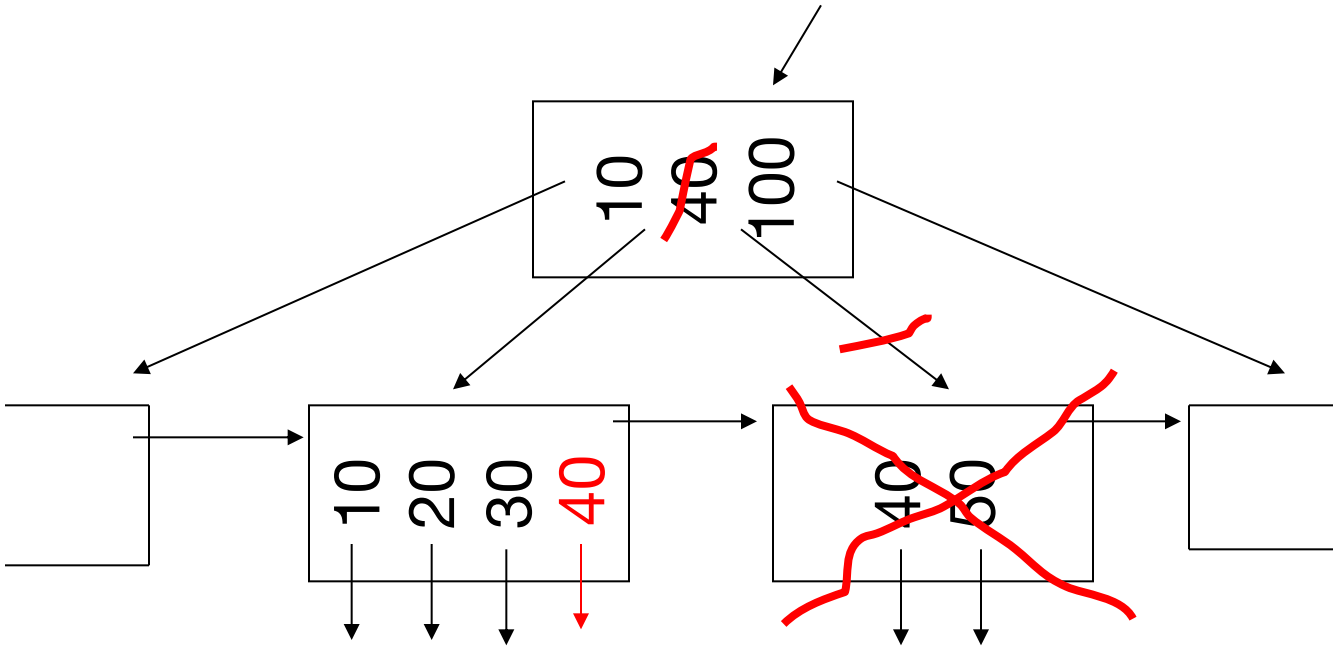
n=4



(b) Coalesce with sibling

» Delete 50

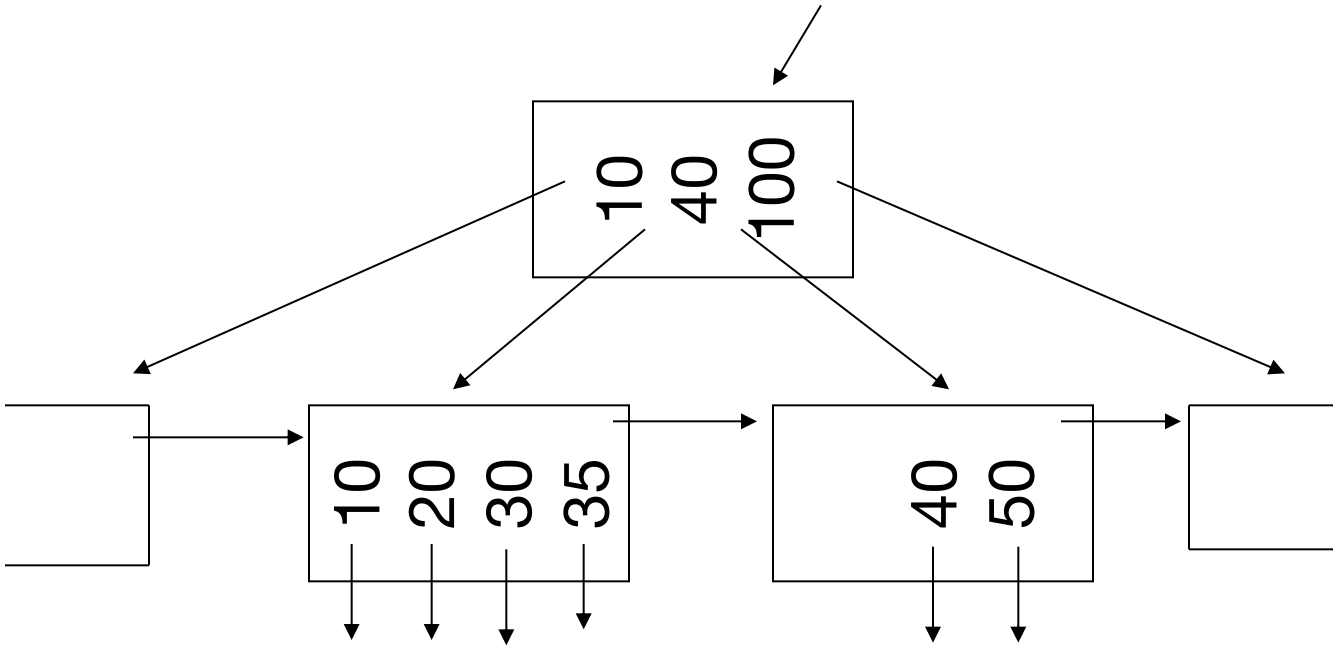
n=4



(c) Redistribute keys

» Delete 50

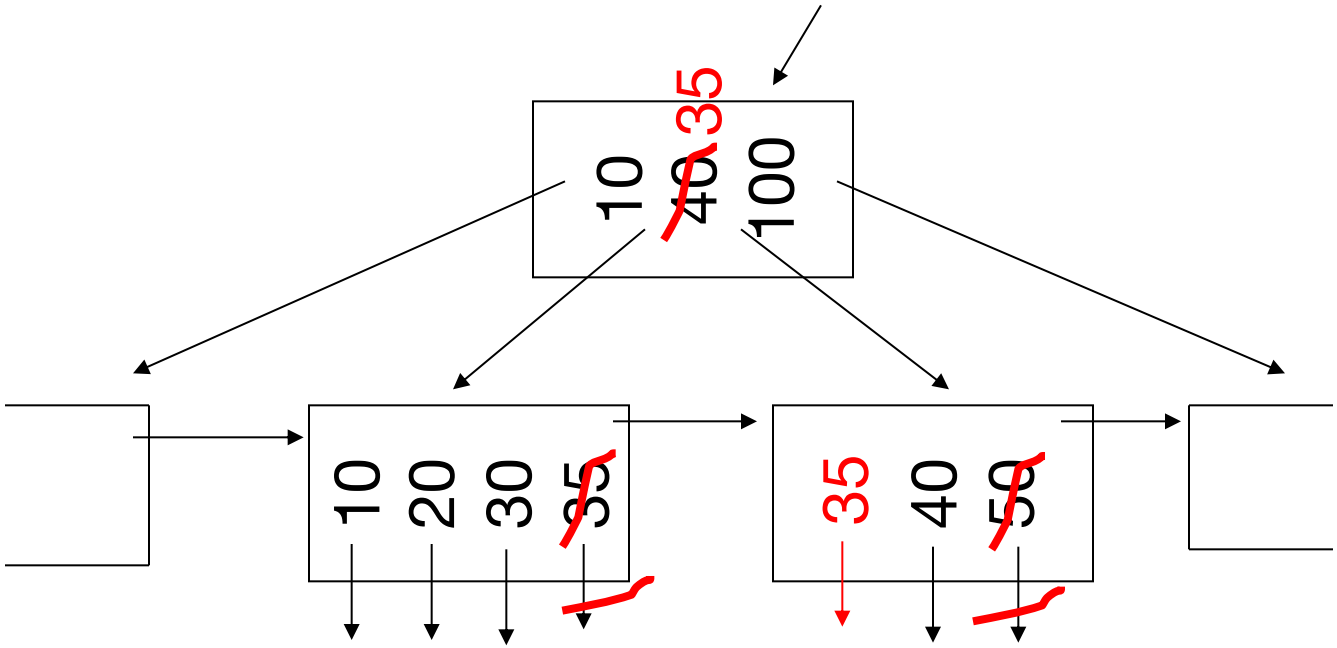
n=4



(c) Redistribute keys

» Delete 50

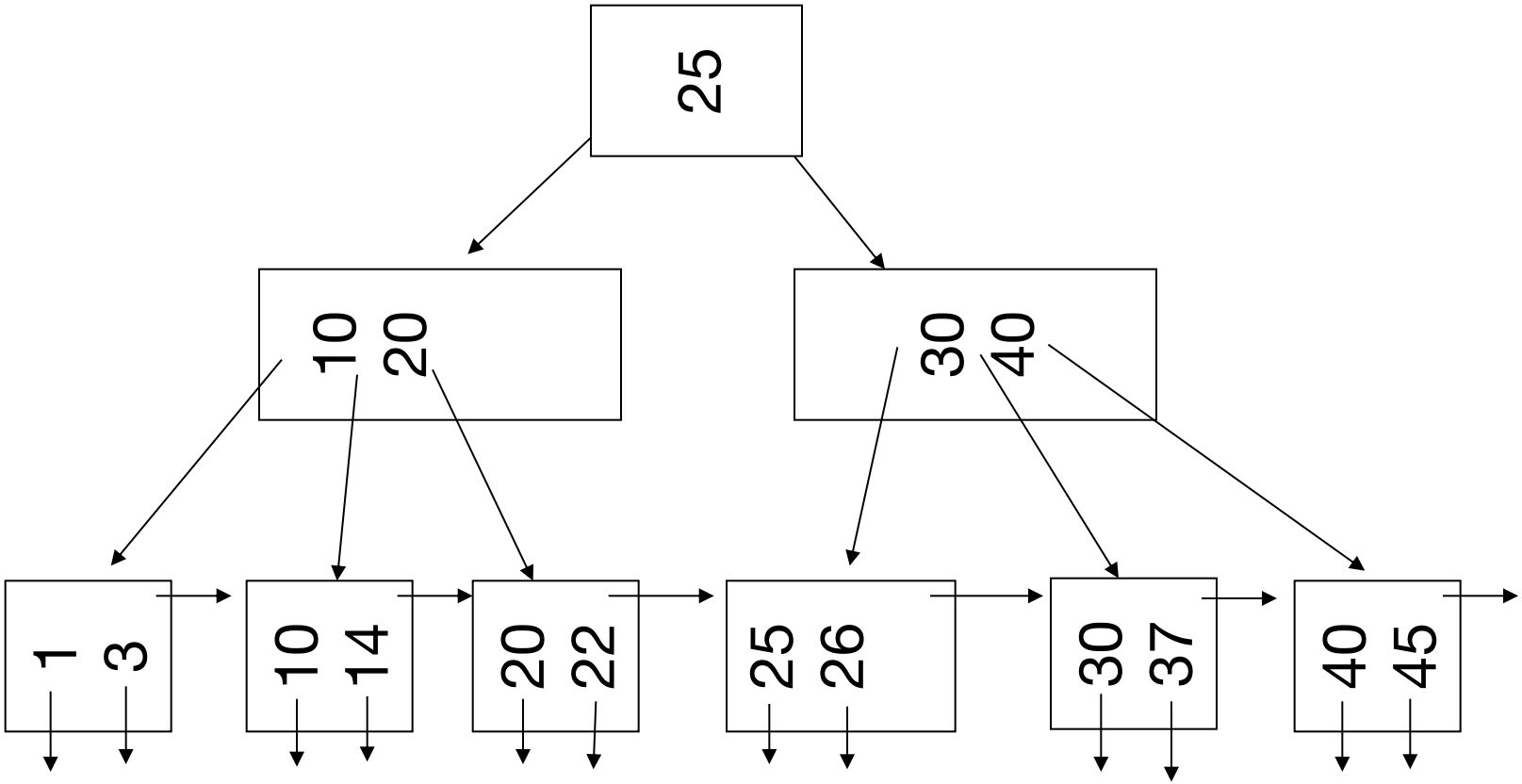
n=4



(d) Non-leaf coalesce

– Delete 37

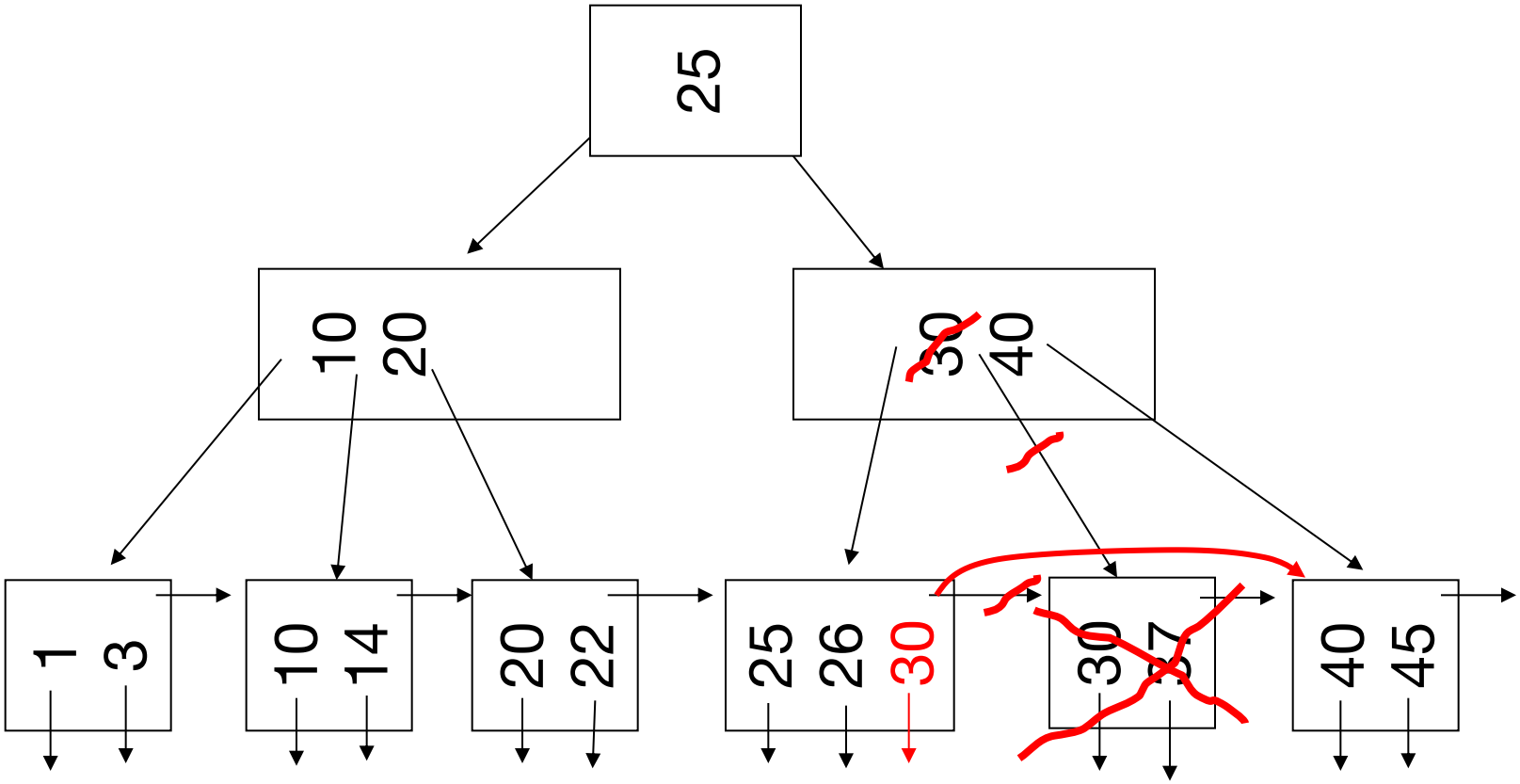
n=4



(d) Non-leaf coalesce

– Delete 37

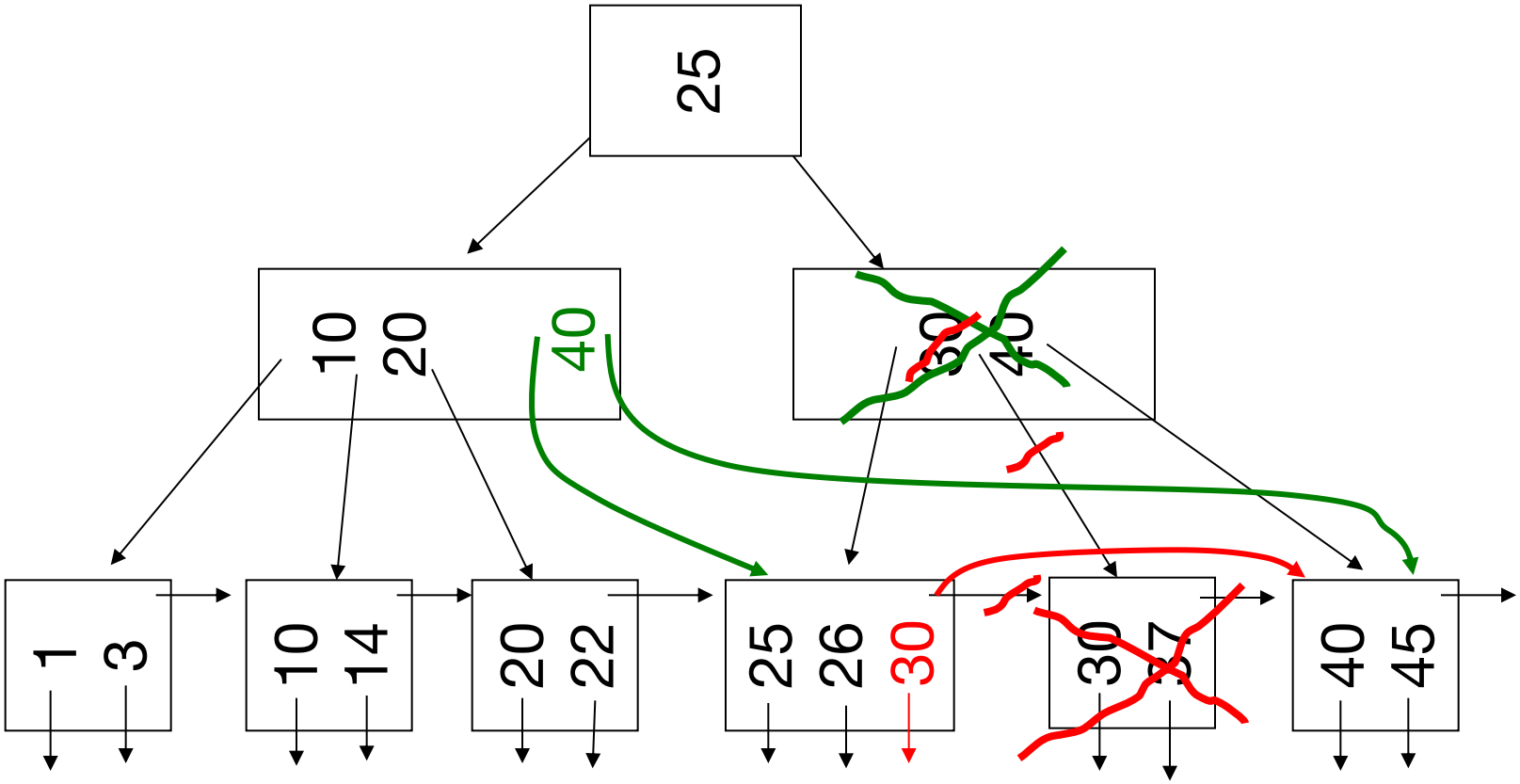
n=4



(d) Non-leaf coalesce

– Delete 37

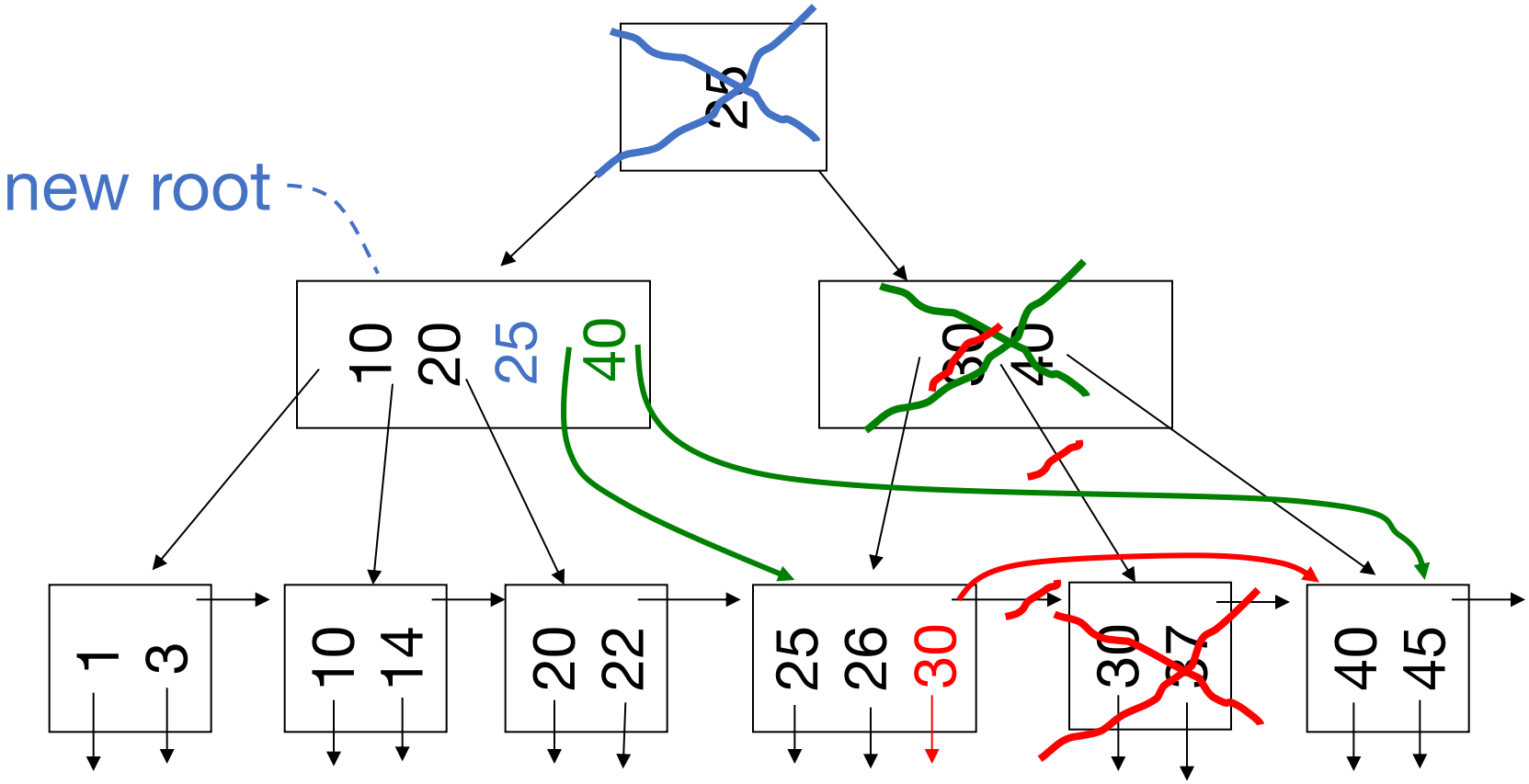
n=4



(d) Non-leaf coalesce

– Delete 37

n=4



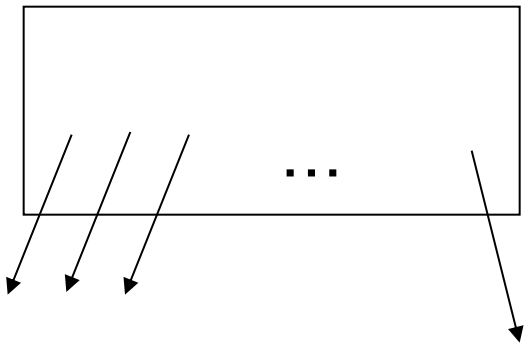
B+ Tree Deletion in Practice

Often, coalescing is not implemented

» Too hard and not worth it! Most datasets only tend to grow in size over time.

Interesting Problem:

For B+ tree, how large should n be?



n is number of keys / node

With modern hardware, get n = 1000 or more

Summary

Wide range of indexes for different data types and queries (e.g. range vs exact)

Key concerns: query time, cost to update, and size of index

Next: given all these storage data structures, how do we run our queries?