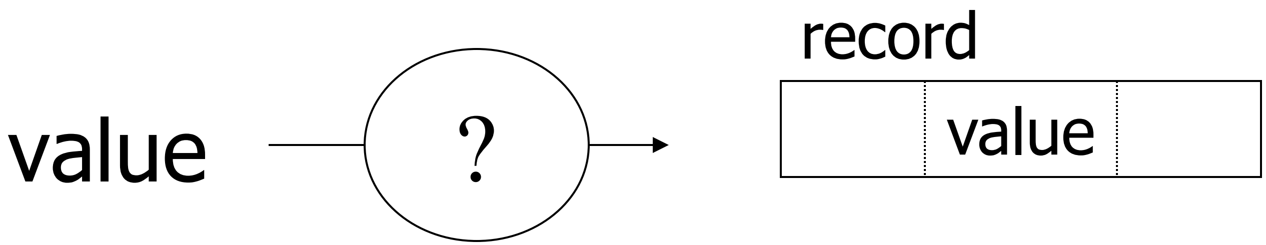
Ullman et al. :
Database System Principles

Notes 4: Indexing

Chapter 4

Indexing & Hashing



Topics

- Conventional indexes
- B-trees
- Hashing schemes

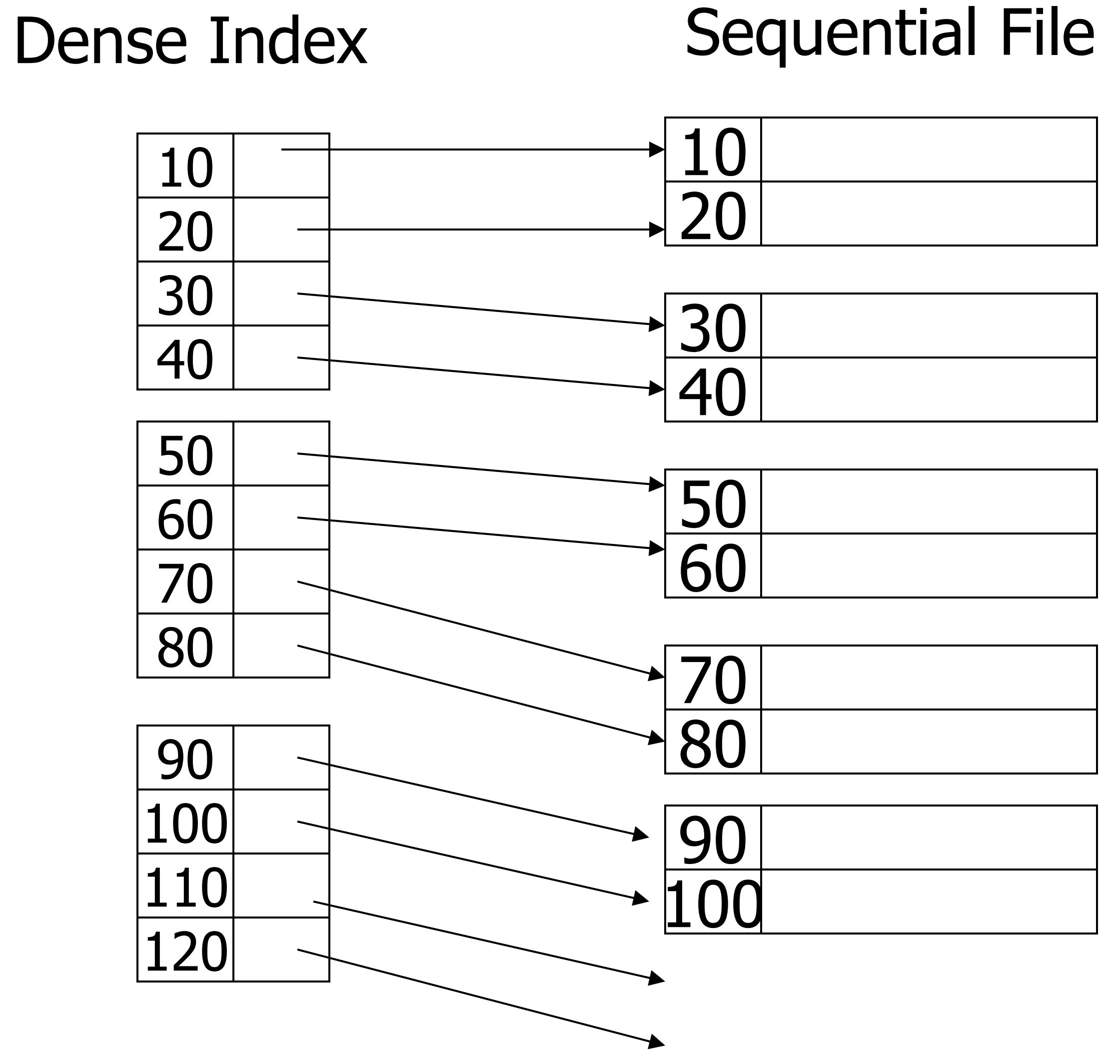
- A single-level index is an auxiliary file that makes it more efficient to search for a record in the data file.
- The index is usually specified on one field of the file (although it could be specified on several fields).
- One form of an index is a file of entries
  <field value, pointer to record>,
  which is ordered by field value.
- The index is called an access path on the field.

- The index file usually occupies considerably less disk blocks than the data file because its entries are much smaller
- A binary search on the index yields a pointer to the file record

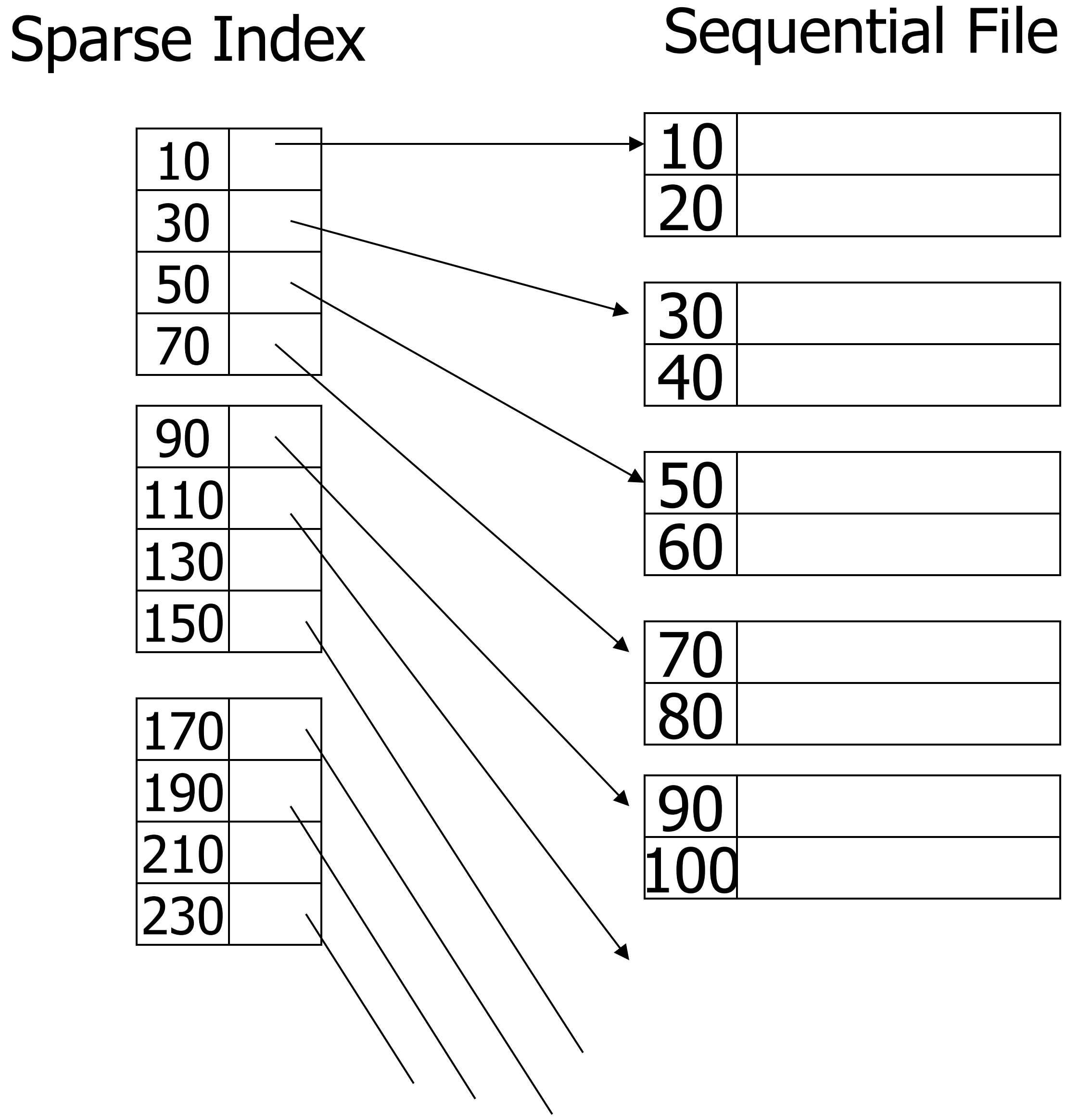
- Indexes can also be characterized as dense or sparse
  - A **dense index** has an index entry for every search key value (usually every record) in the data file.
  - A **sparse** **(or nondense)** **index**, on the other hand, has index entries for only some of the search values (typically one entry per data file block)

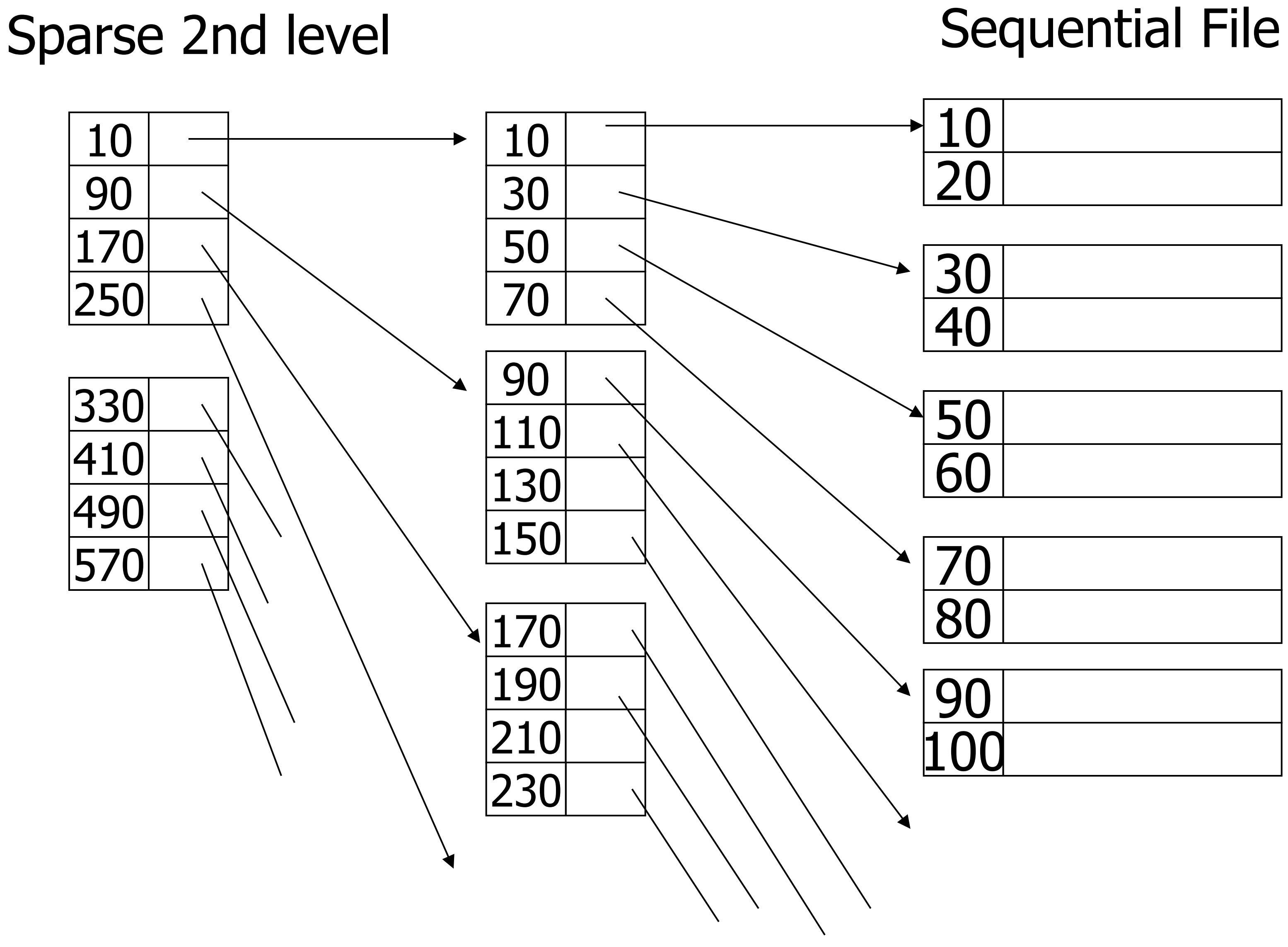
Sequential File

10	
20	
30	
40	
50	
60	
70	
80	
90	
100	



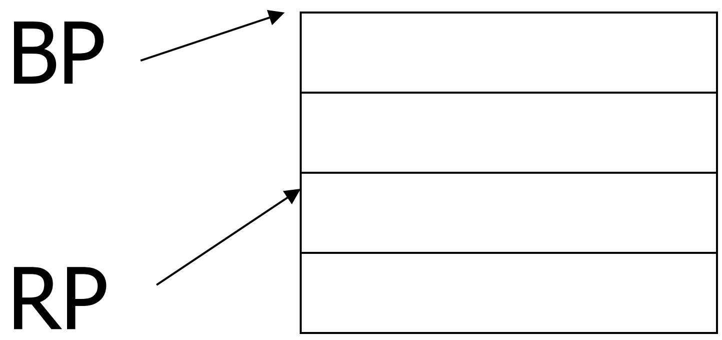
Dense Index
Sequential File
10
20
30
40
50
60
70
80
90
100
110
120
10
20
30
40
50
60
70
80
90
100


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


Sparse 2nd level
Sequential File
10
90
170
250
330
410
490
570
10
30
50
70
90
110
130
150
170
190
210
230
10
20
30
40
50
60
70
80
90
100

Notes on pointers:

(1) Block pointer (sparse index) can be smaller than record pointer



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

(Later:

- sparse better for insertions
- dense needed for secondary indexes)

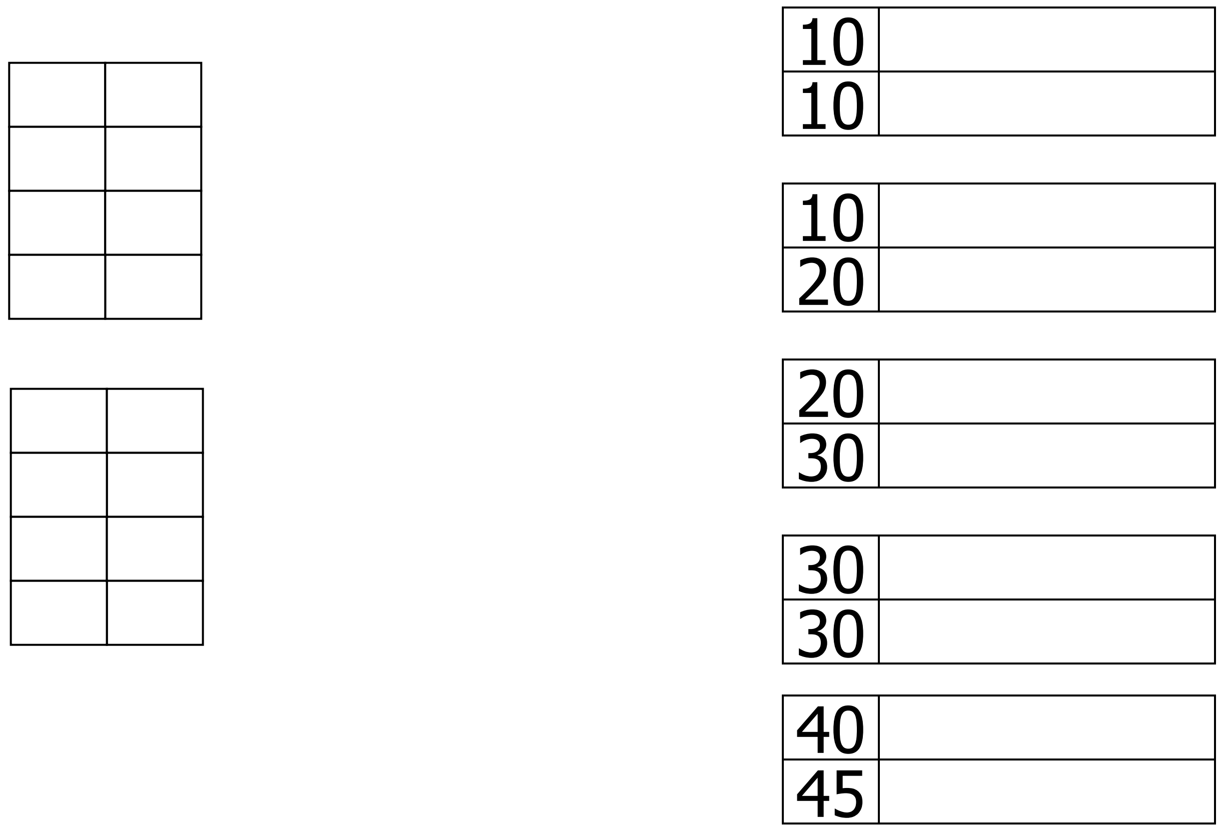
Terms

- Index sequential file
- Search key ($\neq$ primary key)
- Primary index (on ordering field)
- Secondary index (on non-ordering field)
- Dense index (all Search Key values in)
- Sparse index
- Multi-level index

Next:

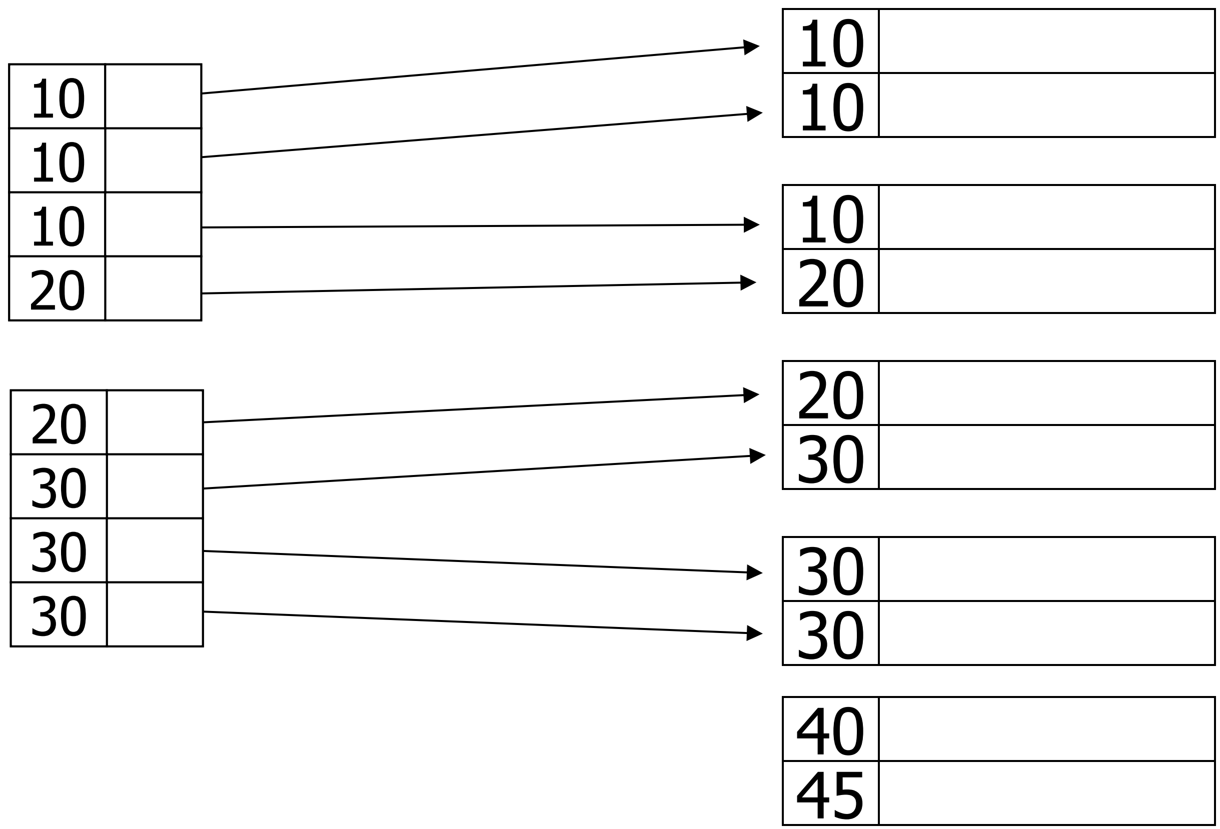
- Duplicate keys
- Deletion/Insertion
- Secondary indexes

Duplicate keys



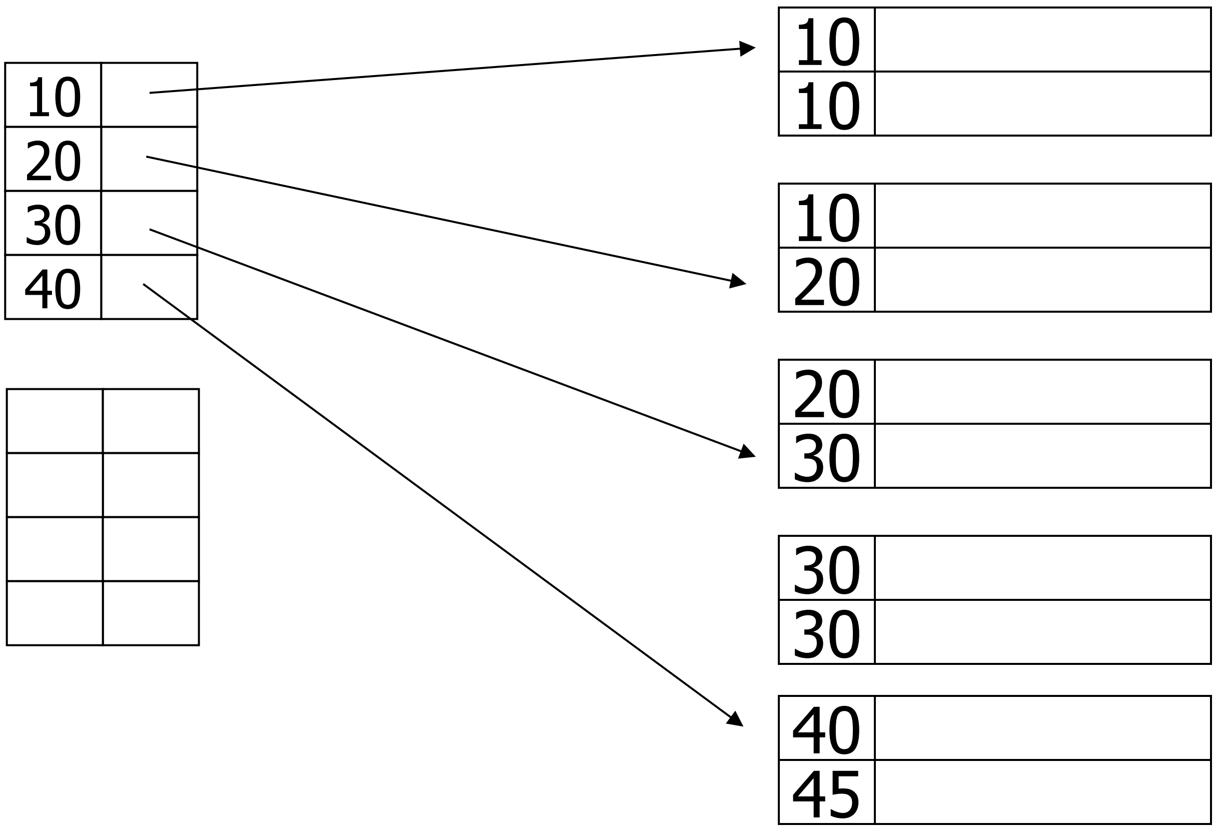
Duplicate keys

Dense index, one way to implement?



Duplicate keys

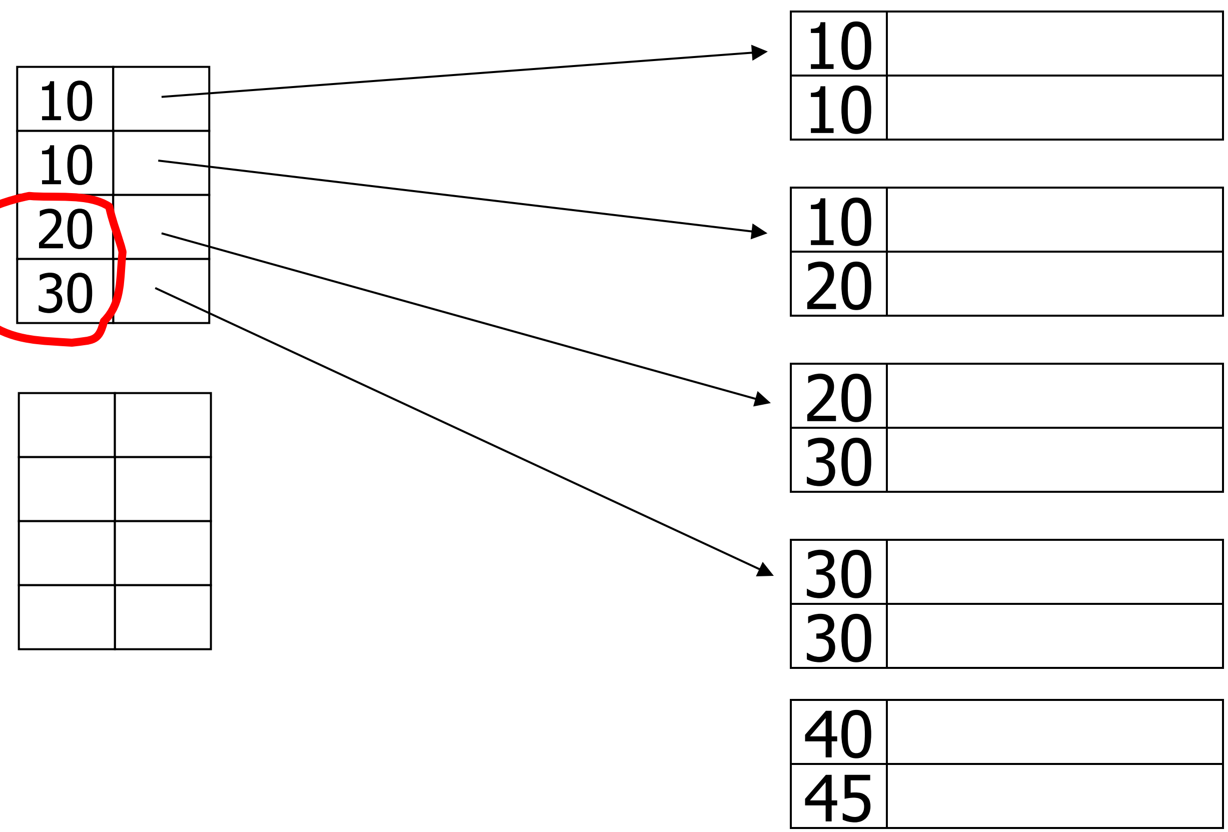
Dense index, better way?



Duplicate keys

Sparse index, one way?

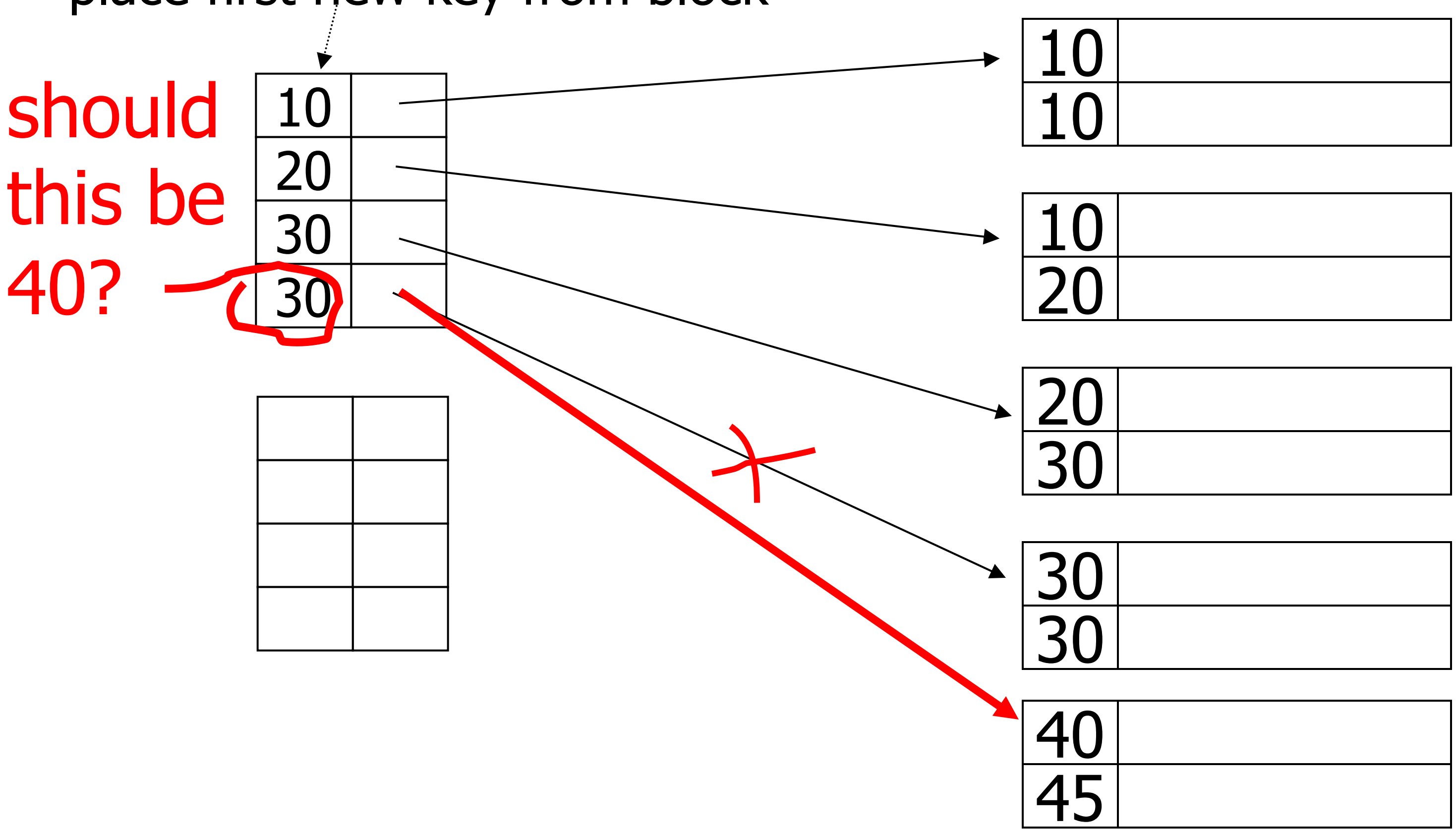
careful if looking for 20 or 30!



Duplicate keys

Sparse index, another way?

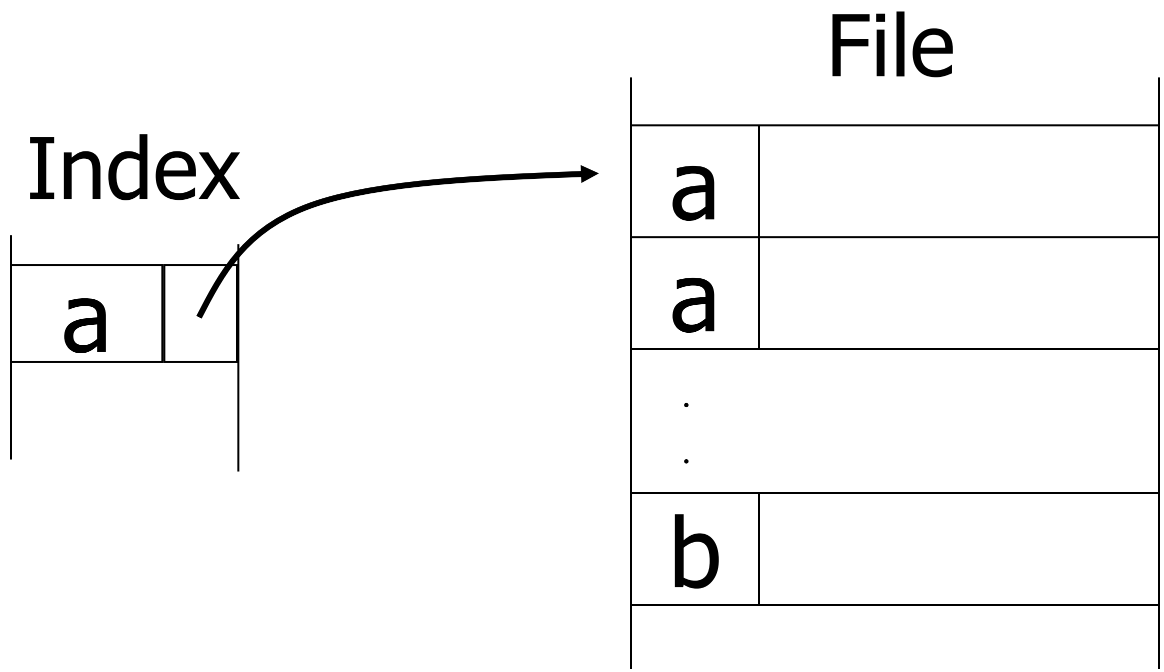
– place first new key from block



Summary

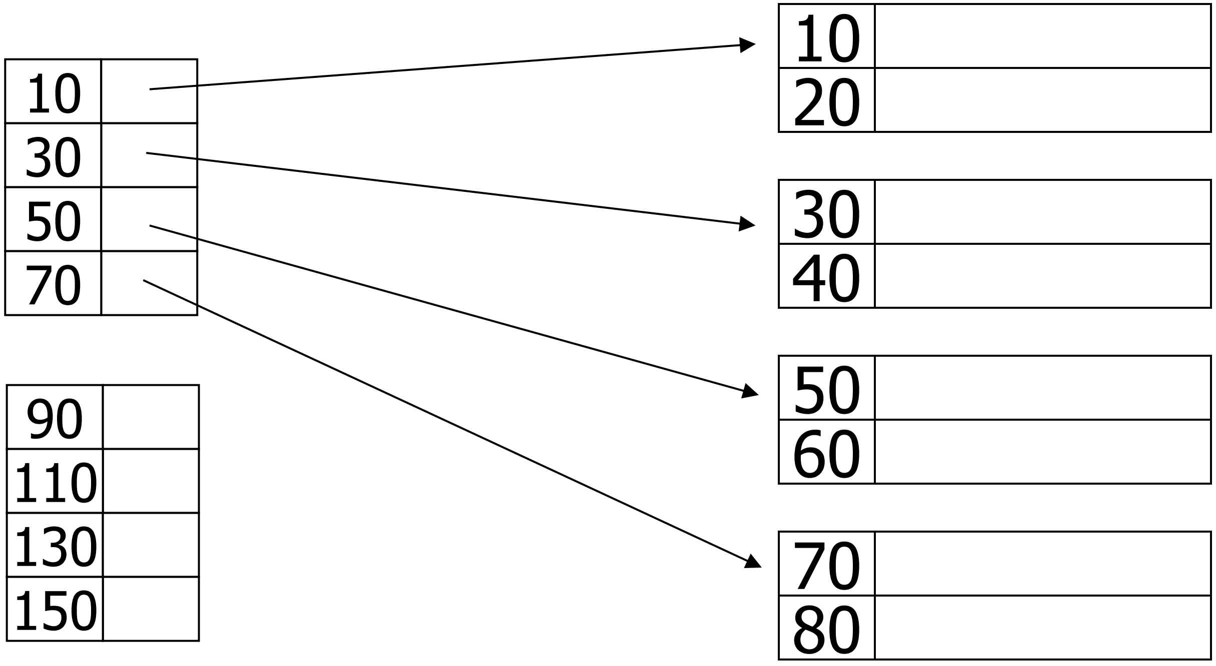
Duplicate values, primary index

- Index may point to <u>first</u> instance of each value only



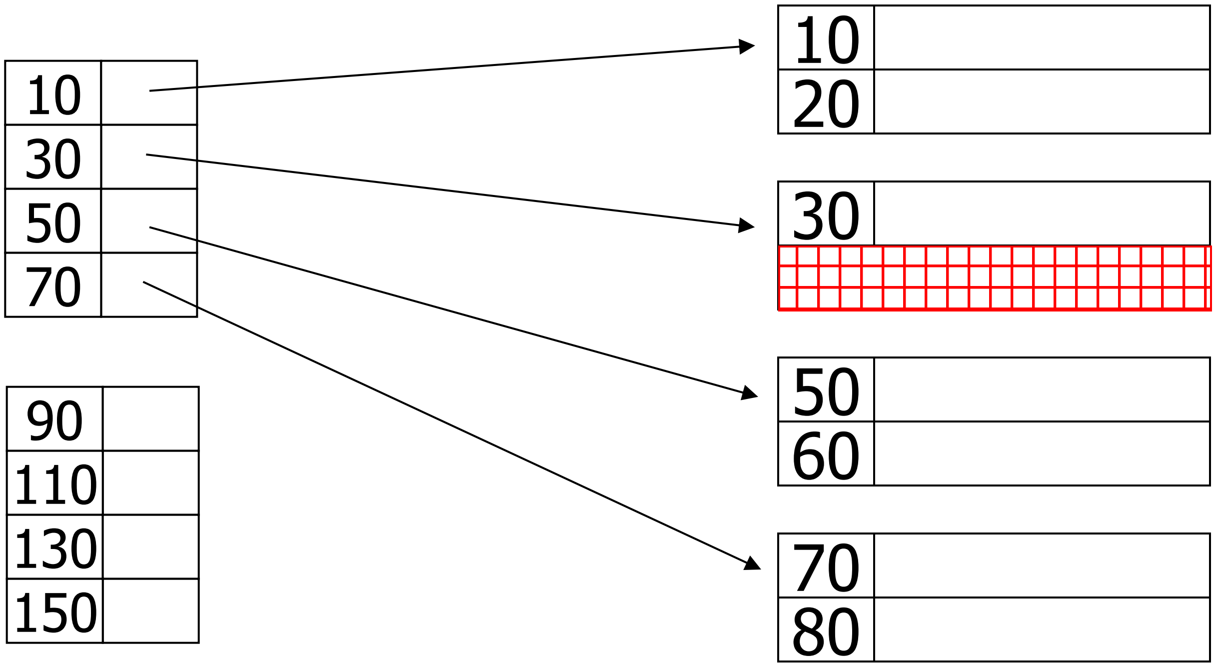
Deletion from sparse index

– delete record 40



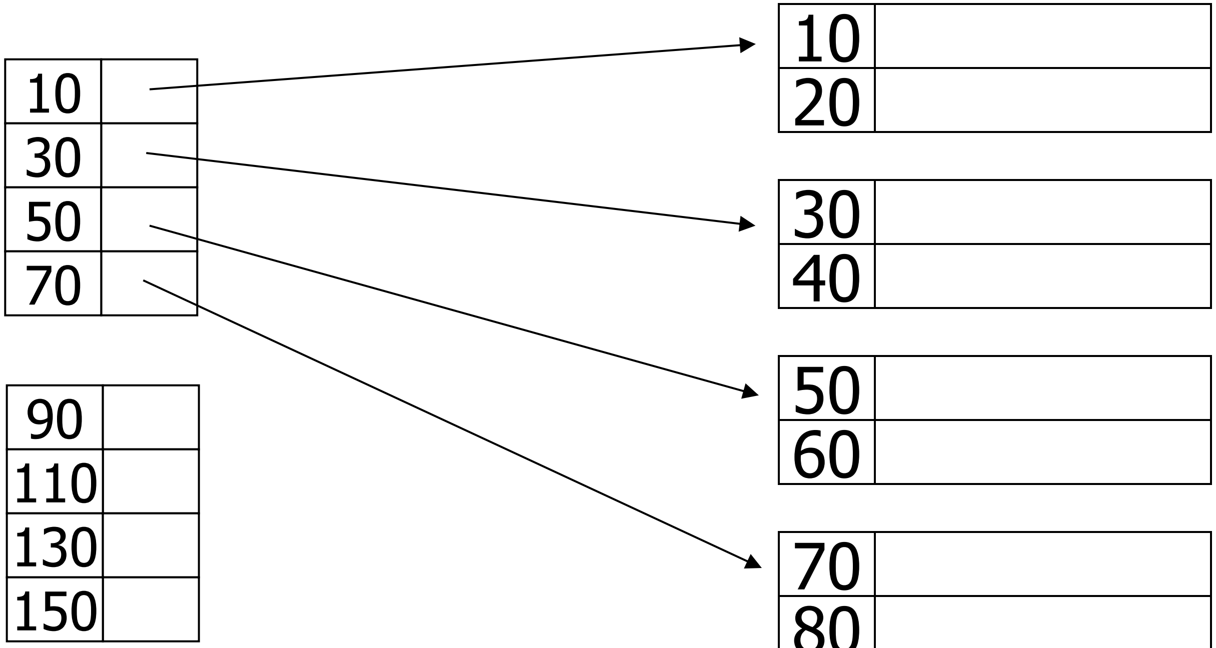
Deletion from sparse index

– delete record 40



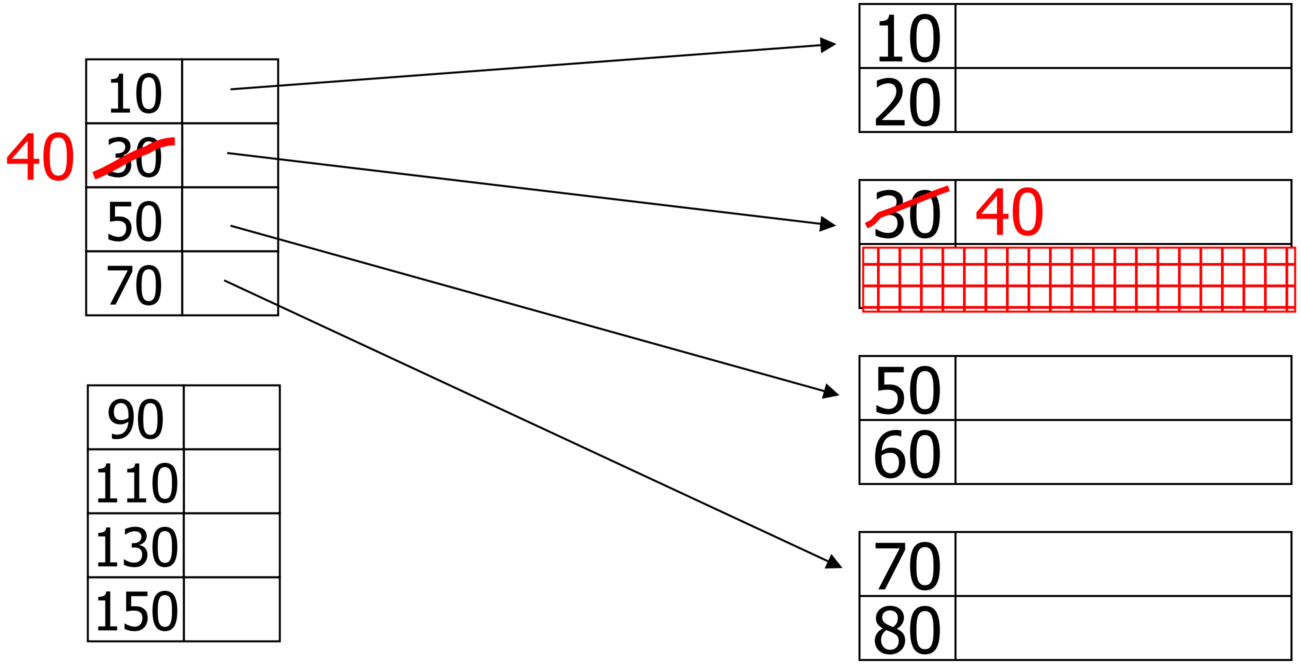
Deletion from sparse index

– delete record 30



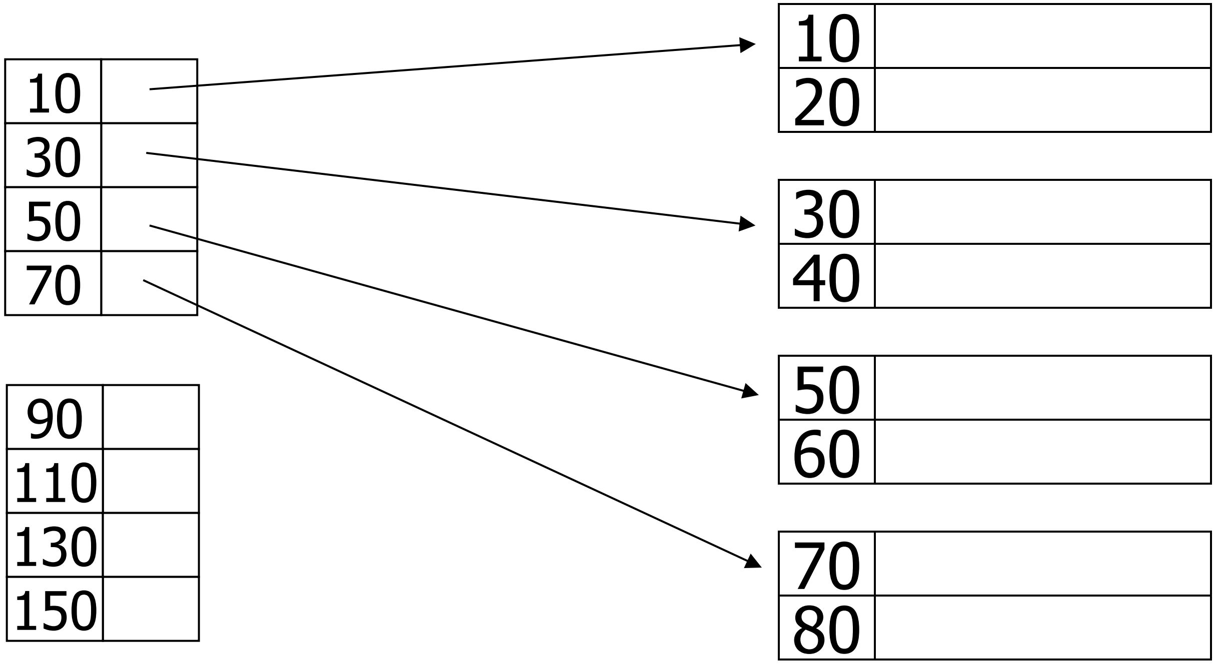
Deletion from sparse index

– delete record 30



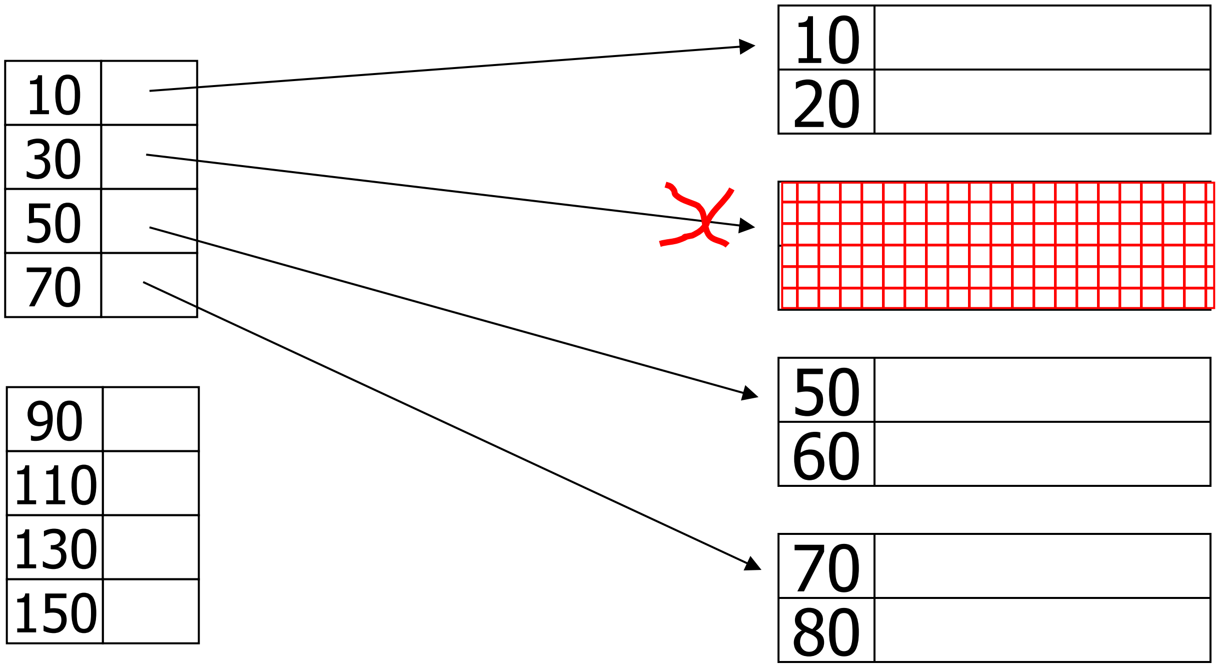
Deletion from sparse index

– delete records 30 & 40



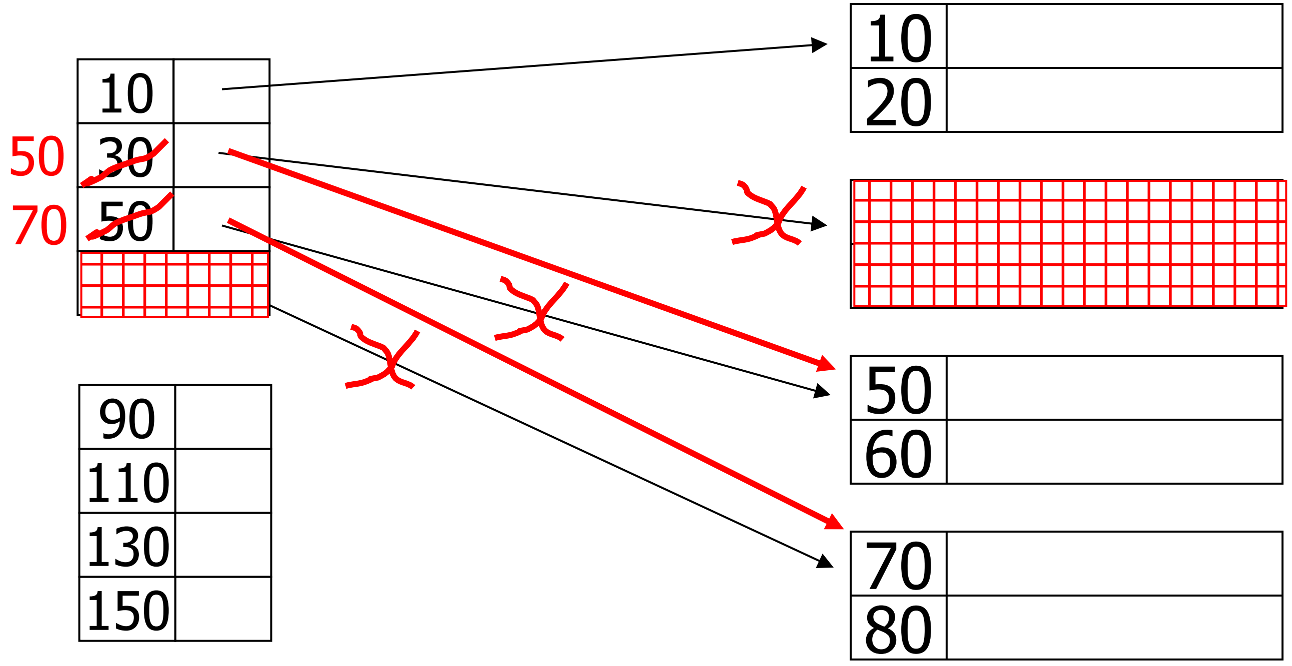
Deletion from sparse index

– delete records 30 & 40



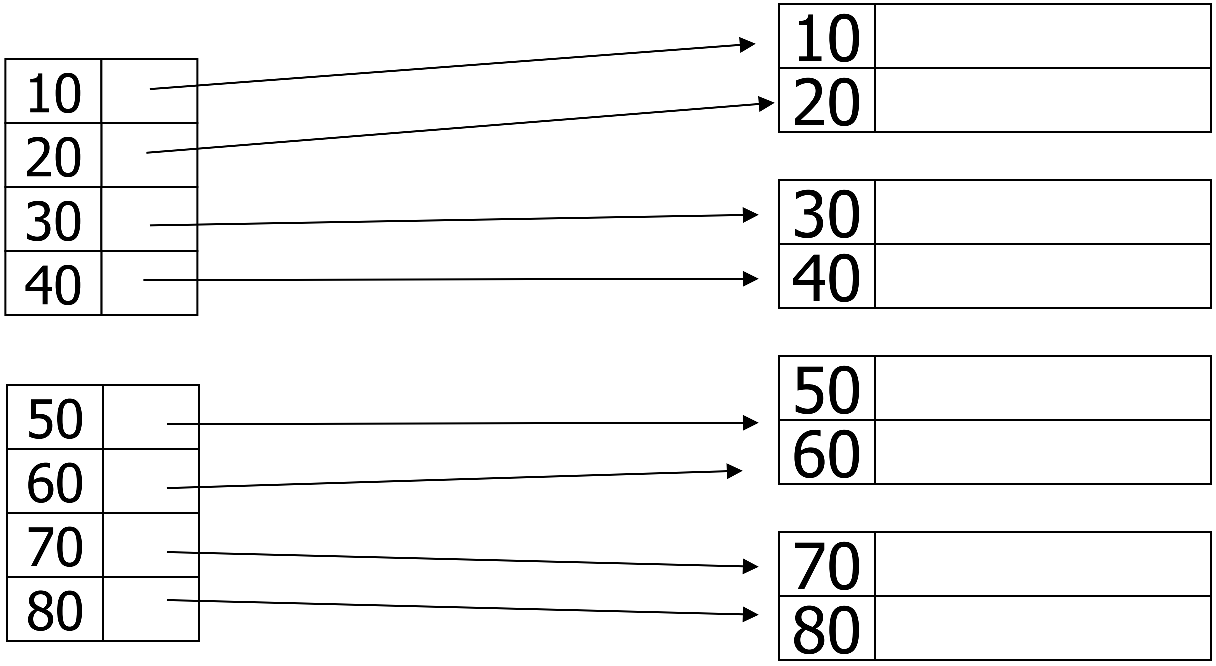
Deletion from sparse index

– delete records 30 & 40



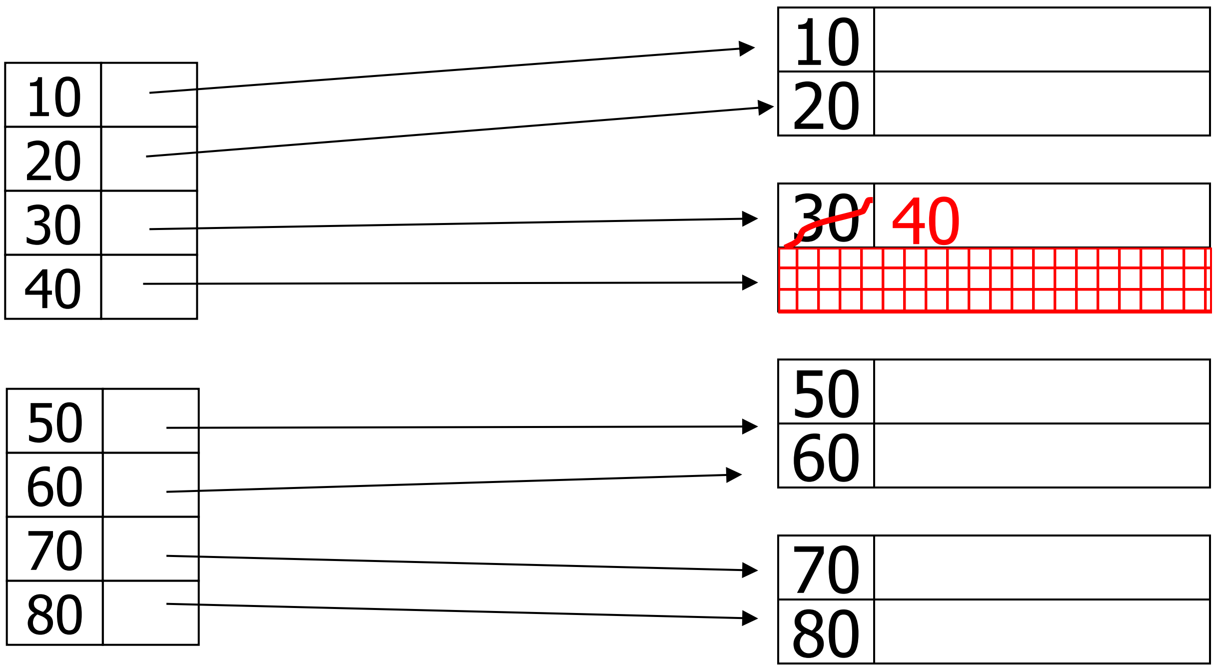
Deletion from dense index

– delete record 30



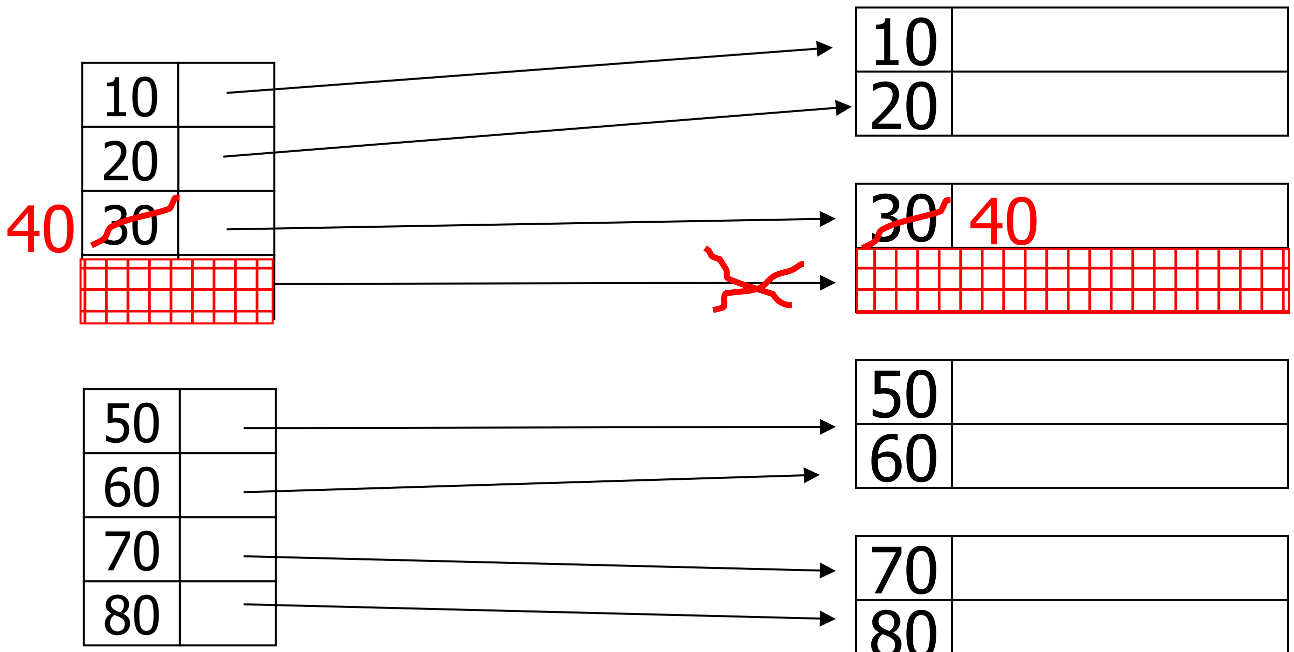
Deletion from dense index

– delete record 30



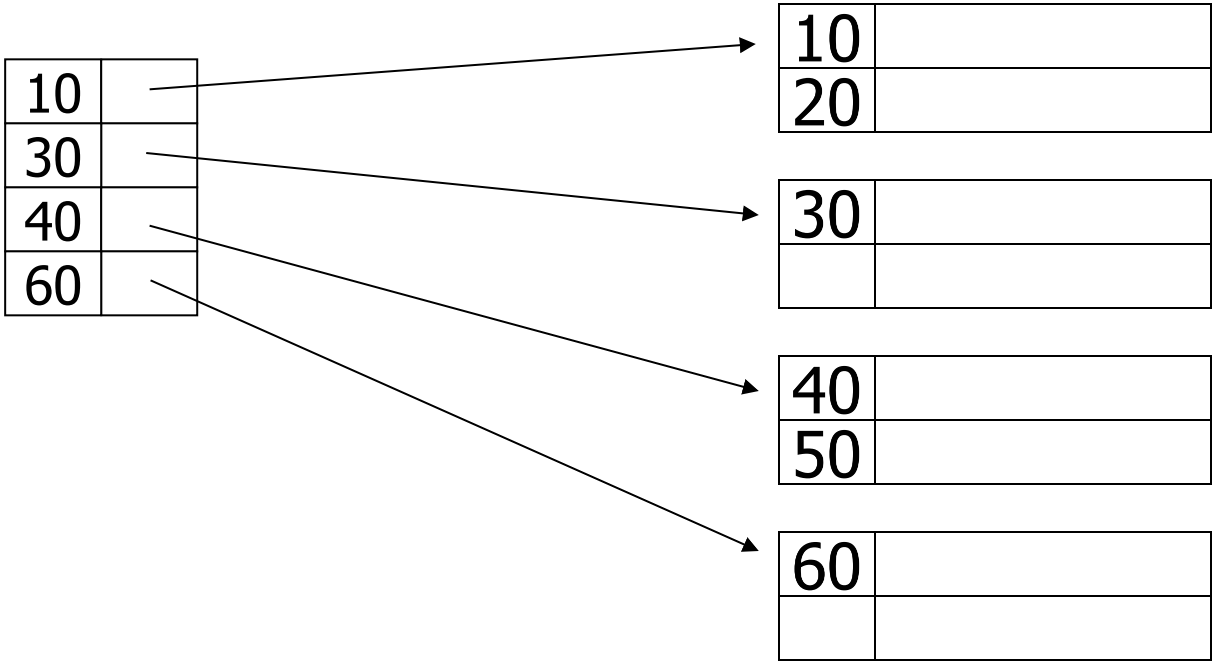
Deletion from dense index

– delete record 30



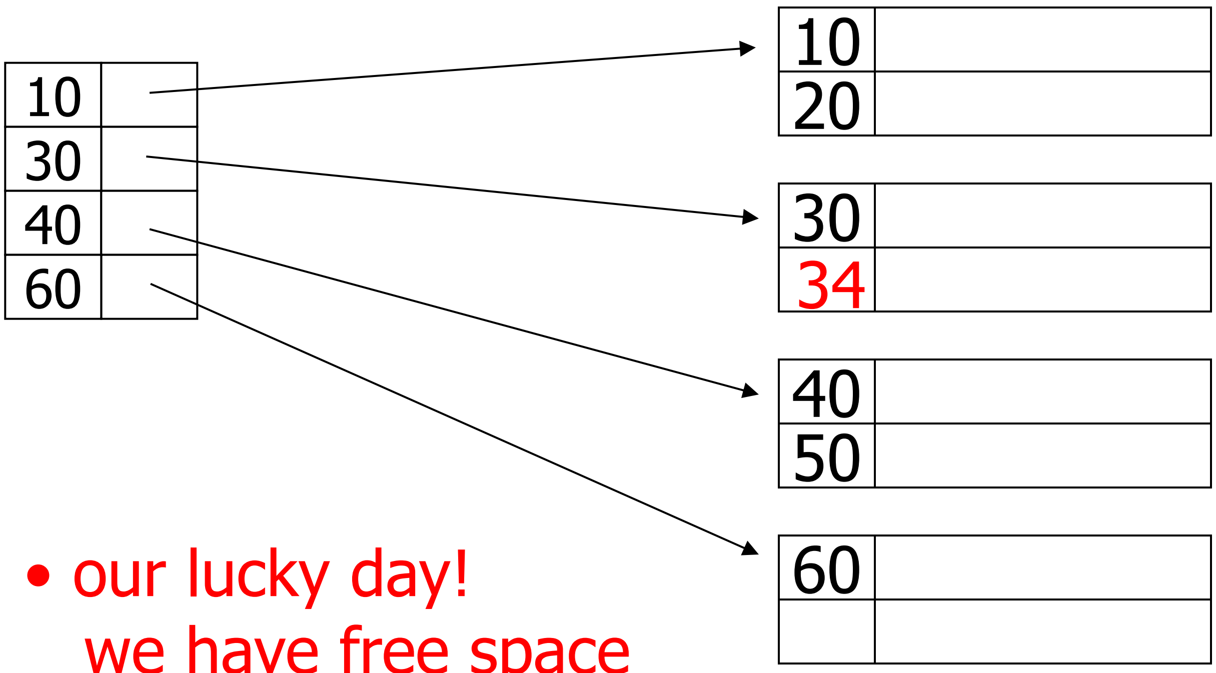
Insertion, sparse index case

– insert record 34



Insertion, sparse index case

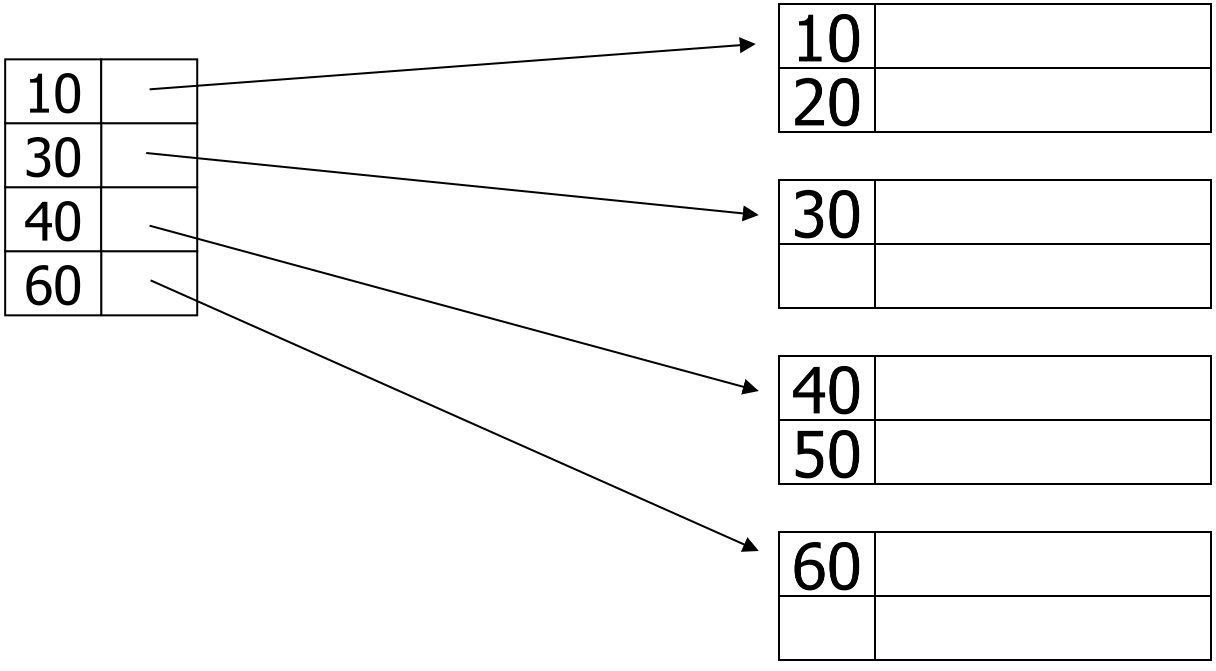
- insert record 34



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

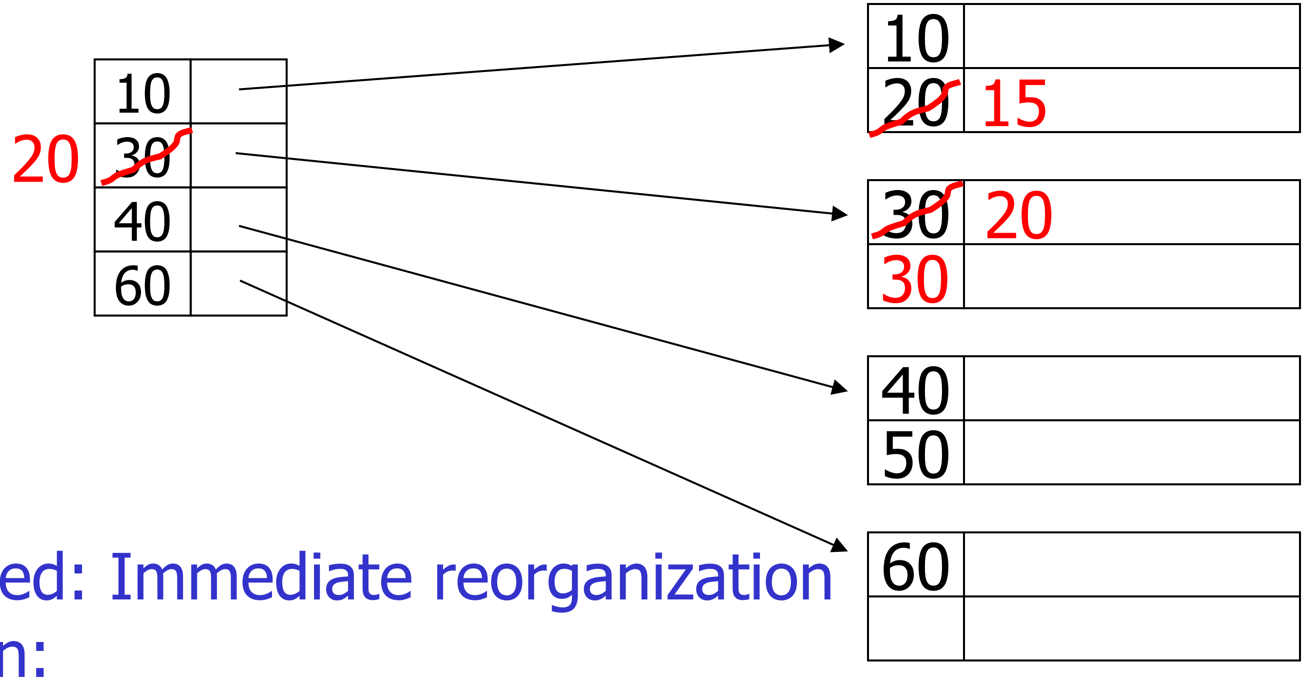
Insertion, sparse index case

– insert record 15



Insertion, sparse index case

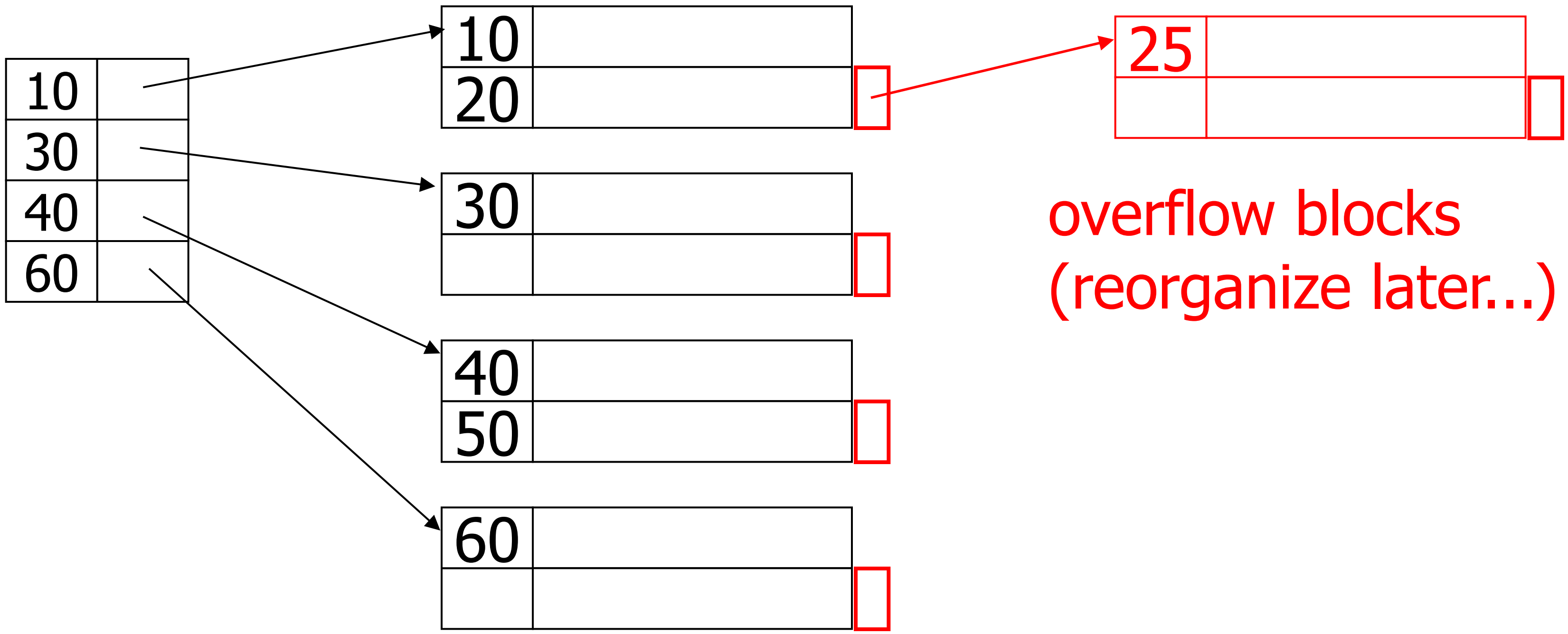
– insert record 15



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

Insertion, sparse index case

– insert record 25

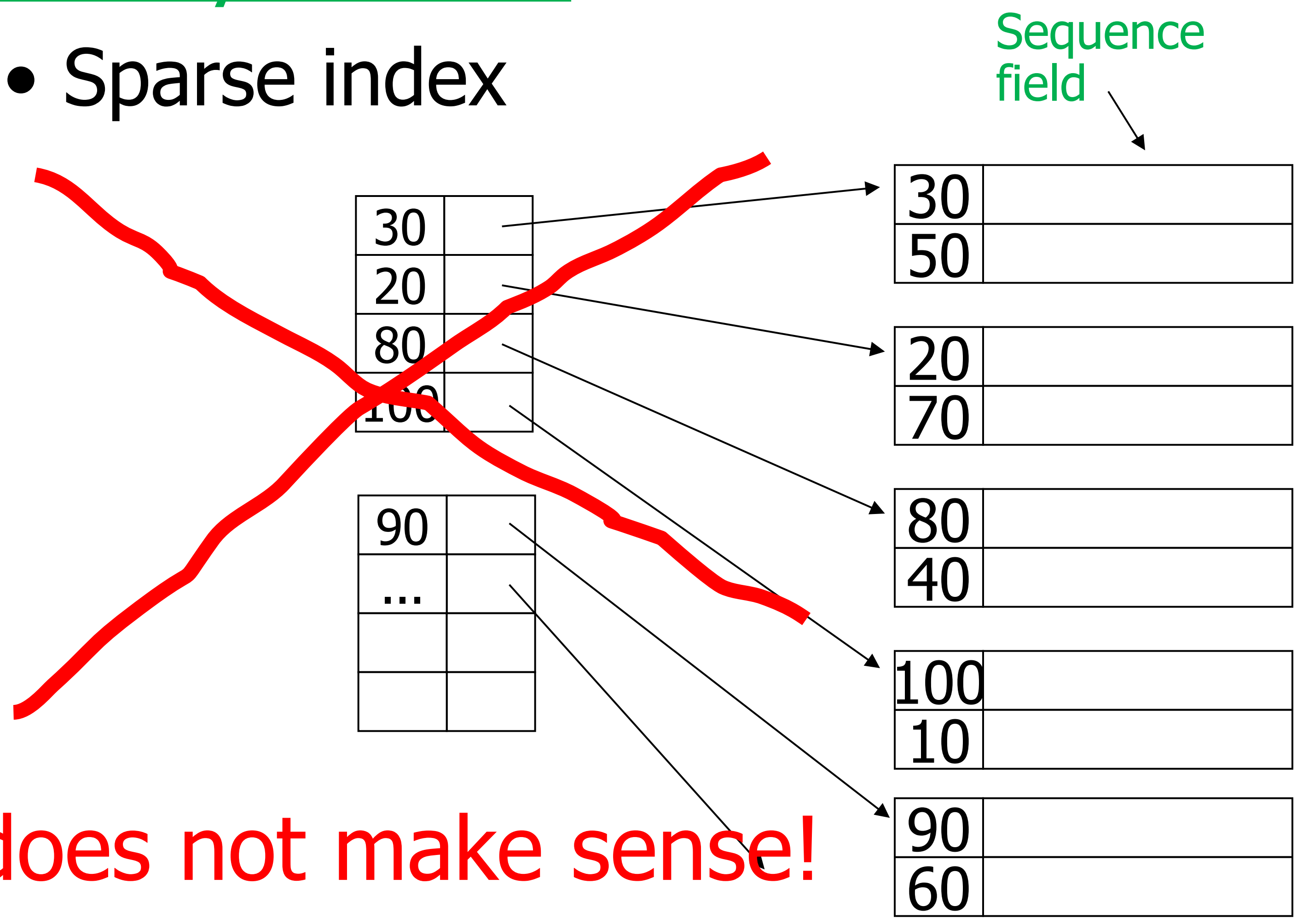


Insertion, dense index case

- Similar
- Often more expensive . . .

Secondary indexes

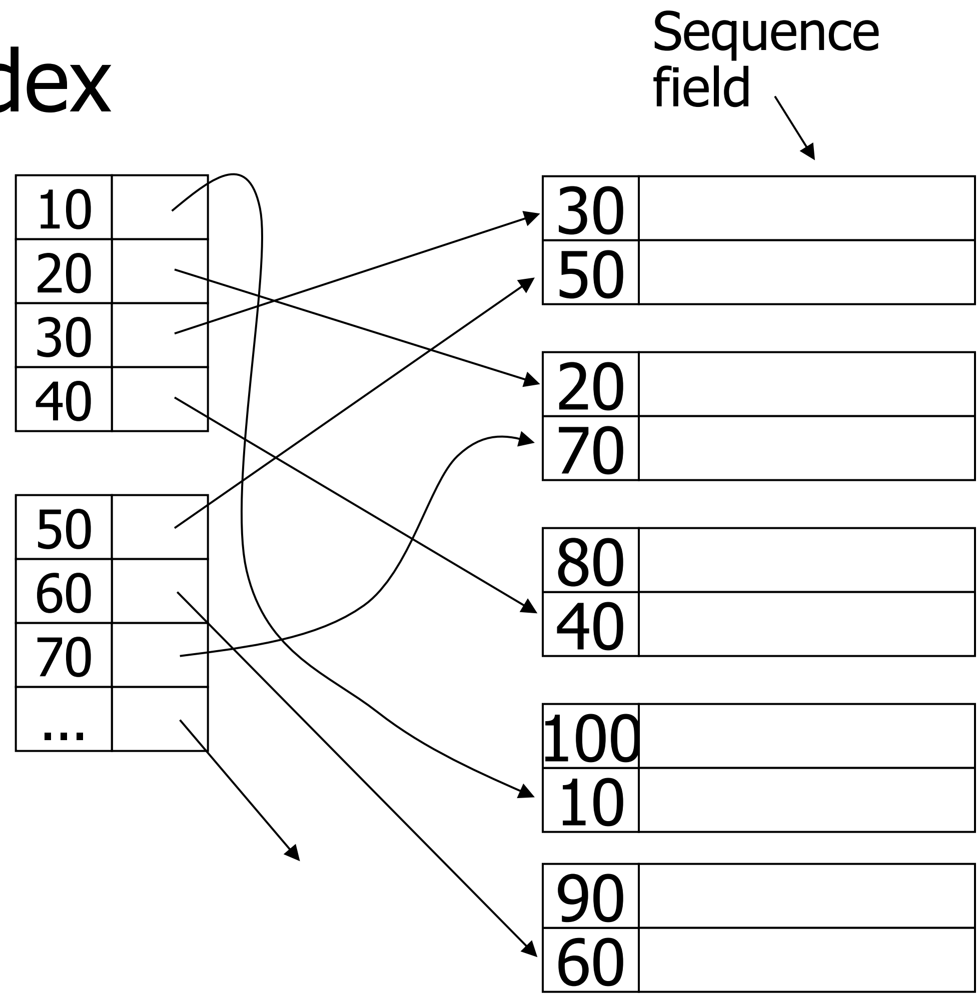
- Sparse index



does not make sense!

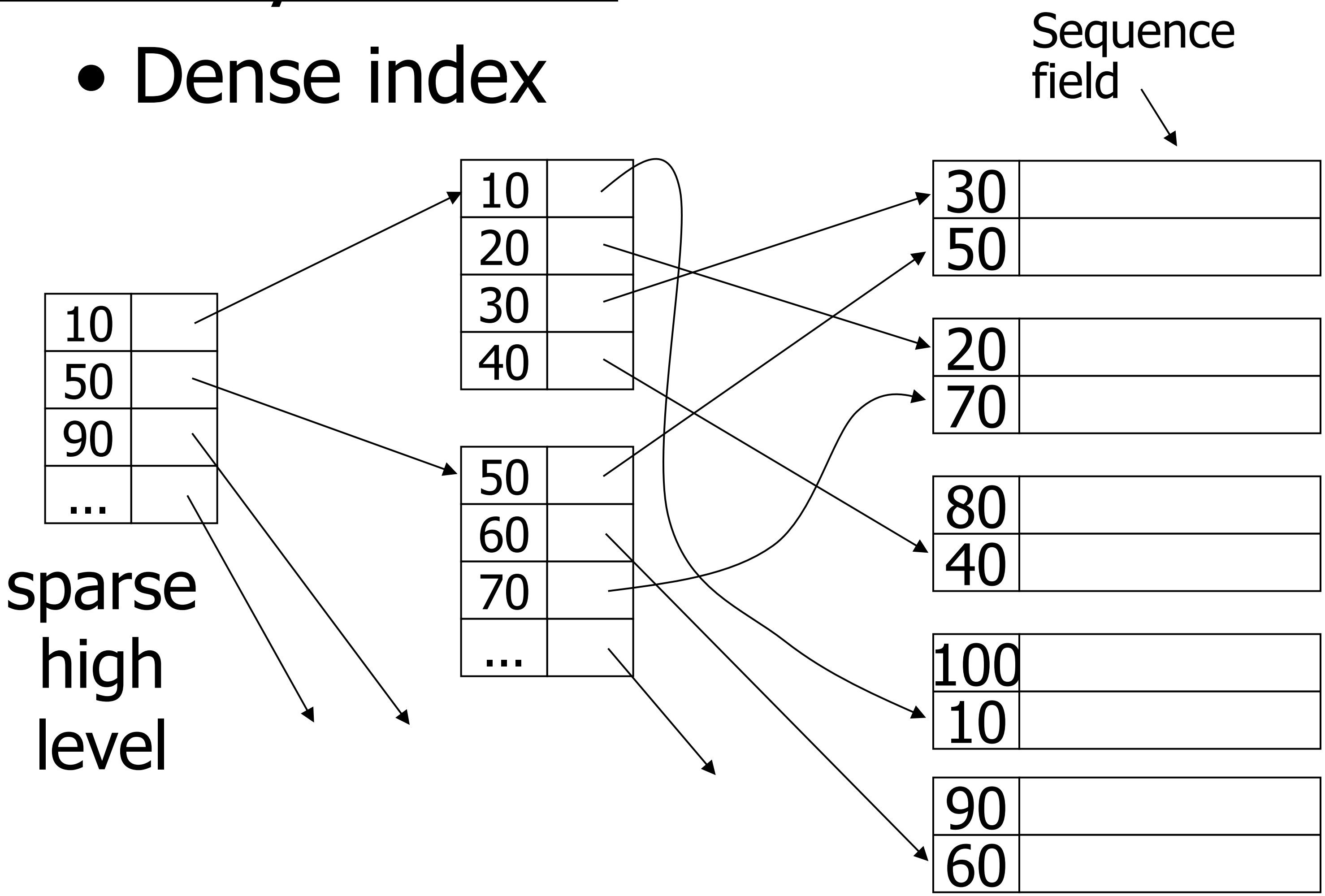
Secondary indexes

- Dense index



Secondary indexes

- Dense index



With secondary indexes:

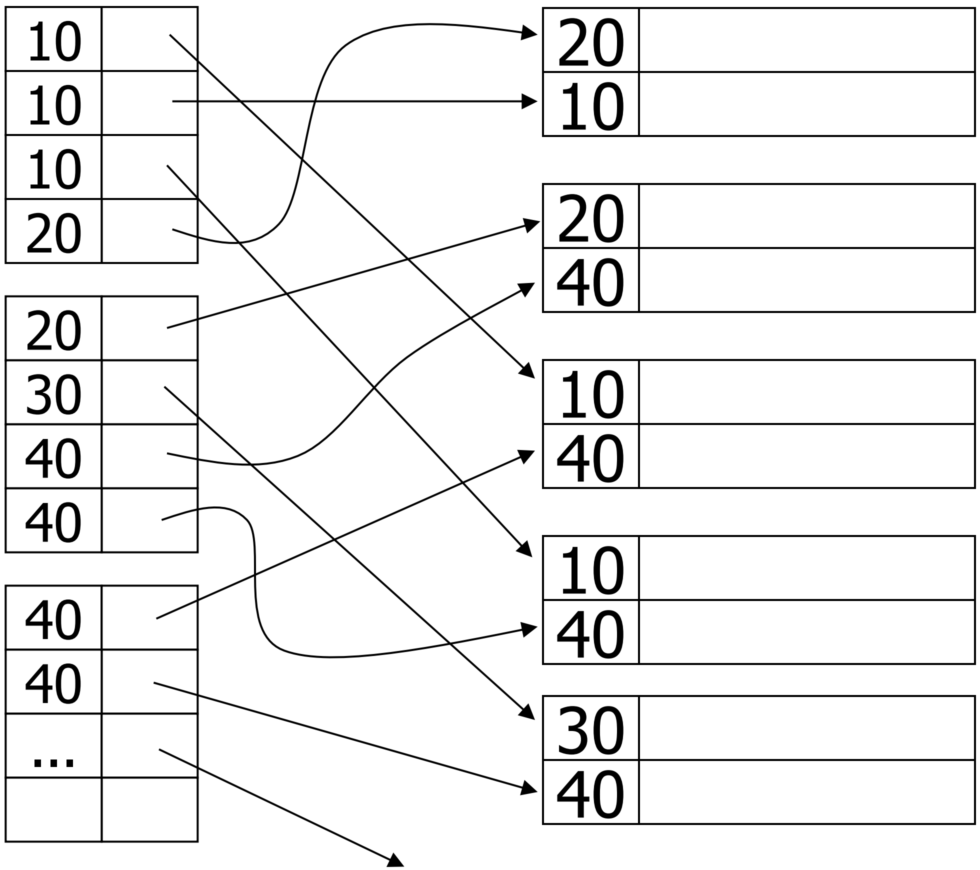
- Lowest level is dense
- Other levels are sparse

Also: Pointers are record pointers

(not block pointers)

Duplicate values & secondary indexes

one option...



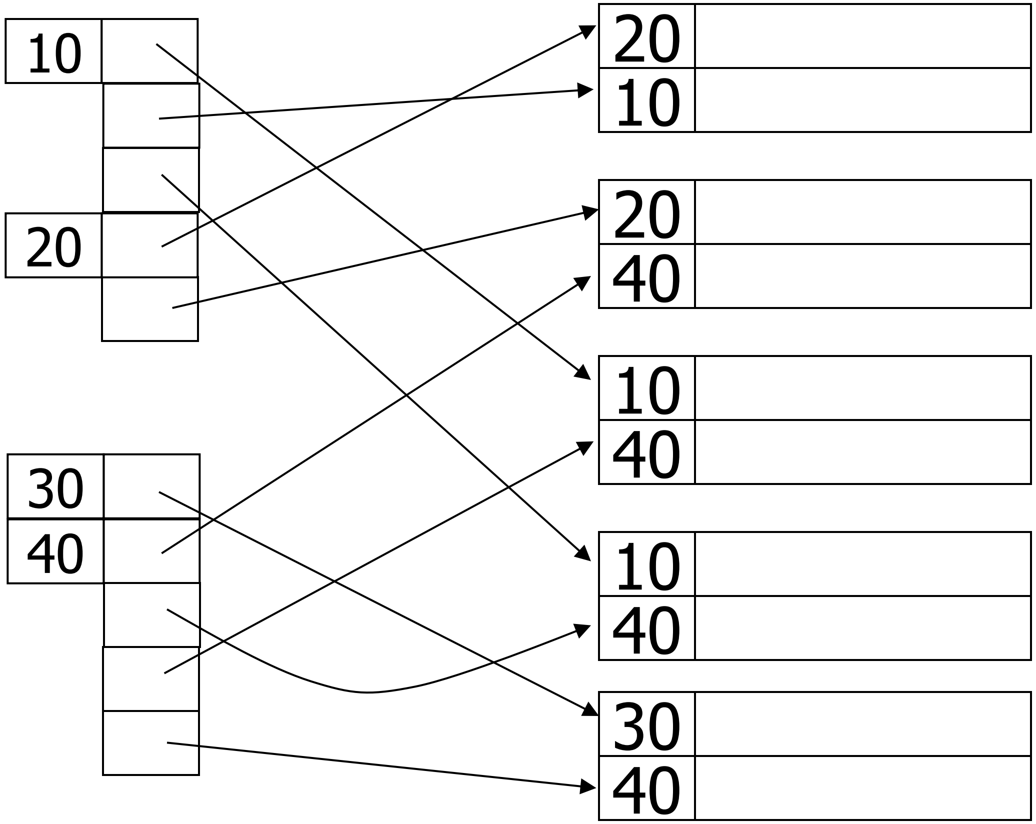
Problem:
excess overhead!

- disk space
- search time

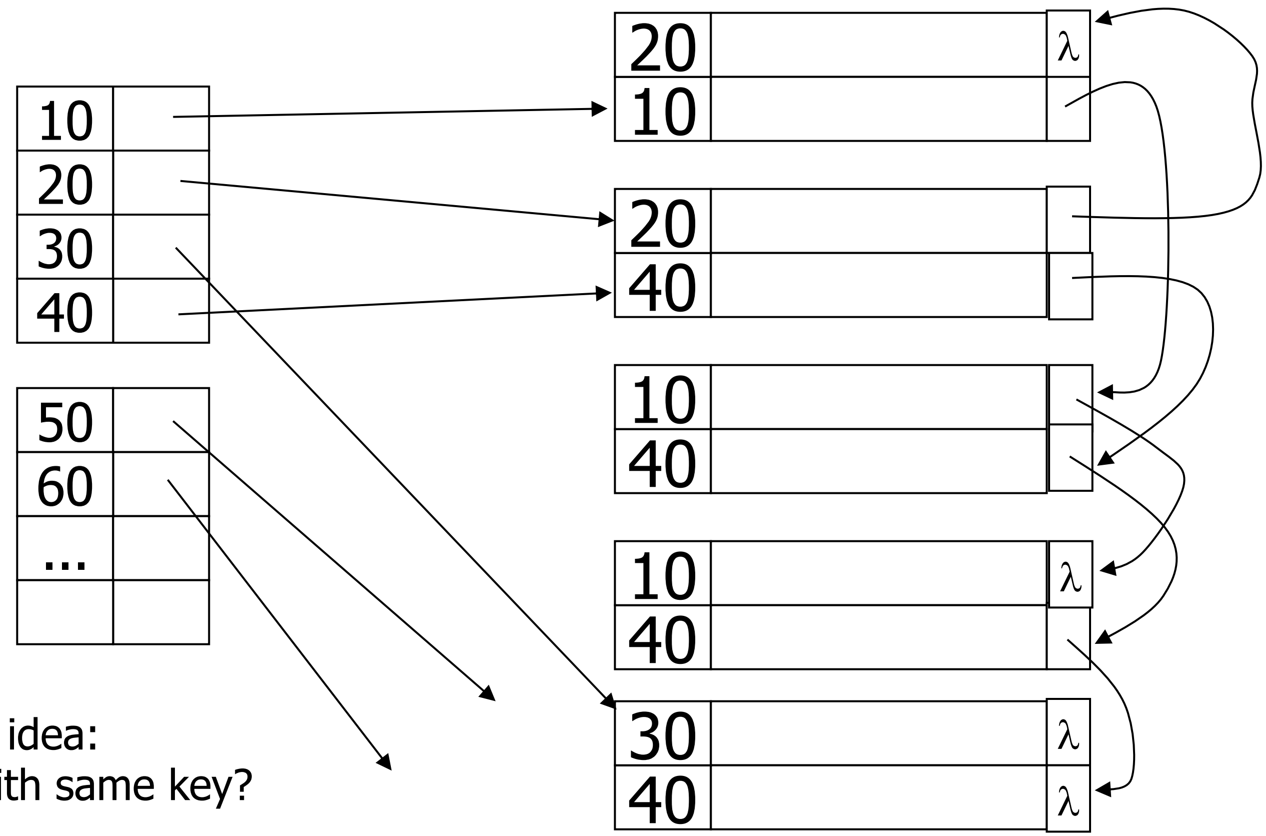
Duplicate values & secondary indexes

another option...

Problem: variable size records in index!



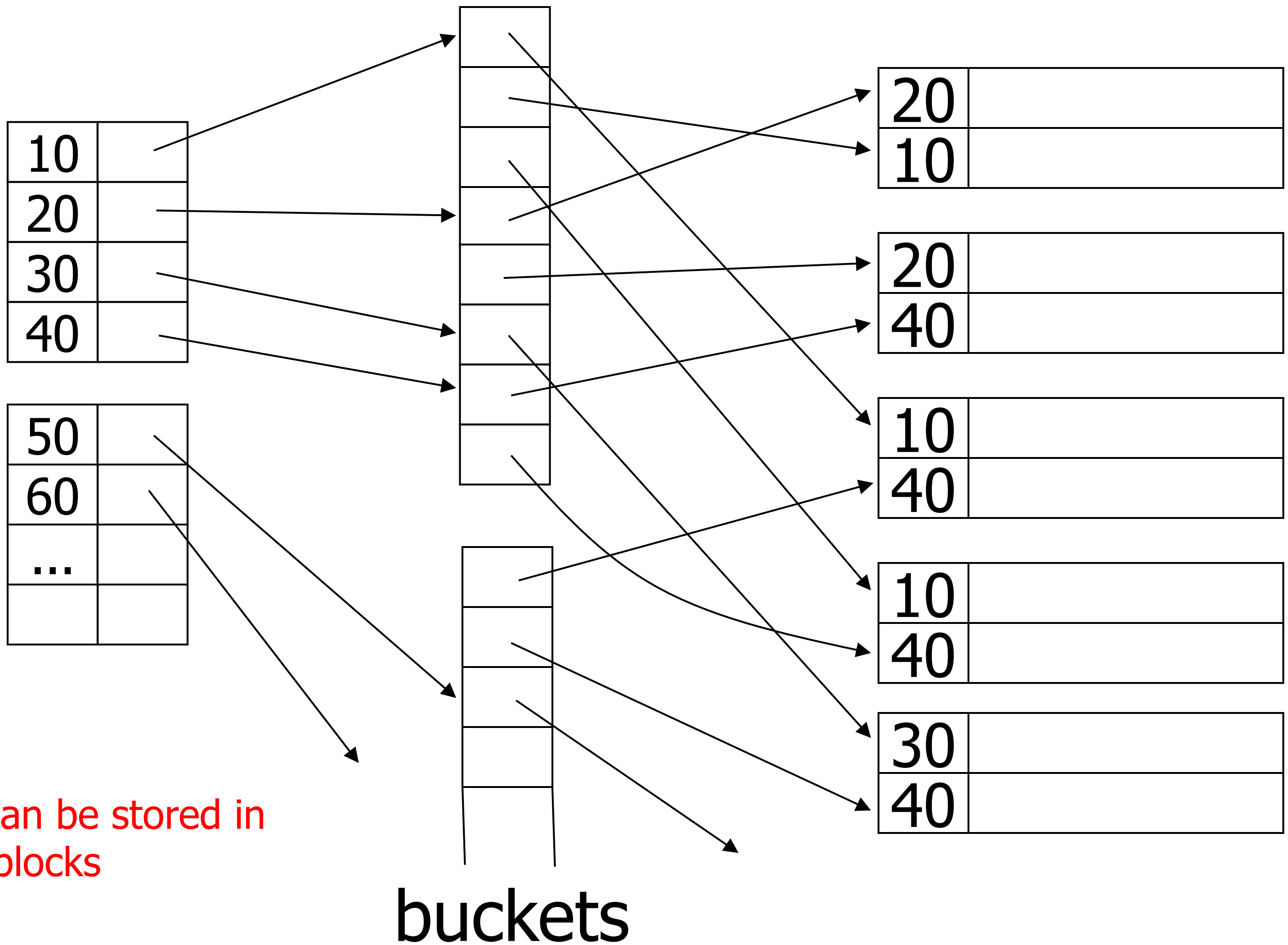
Duplicate values & secondary indexes



Another idea:
Chain records with same key?

Problems:
- Need to add fields to records
- Need to follow chain to know records

Duplicate values & secondary indexes



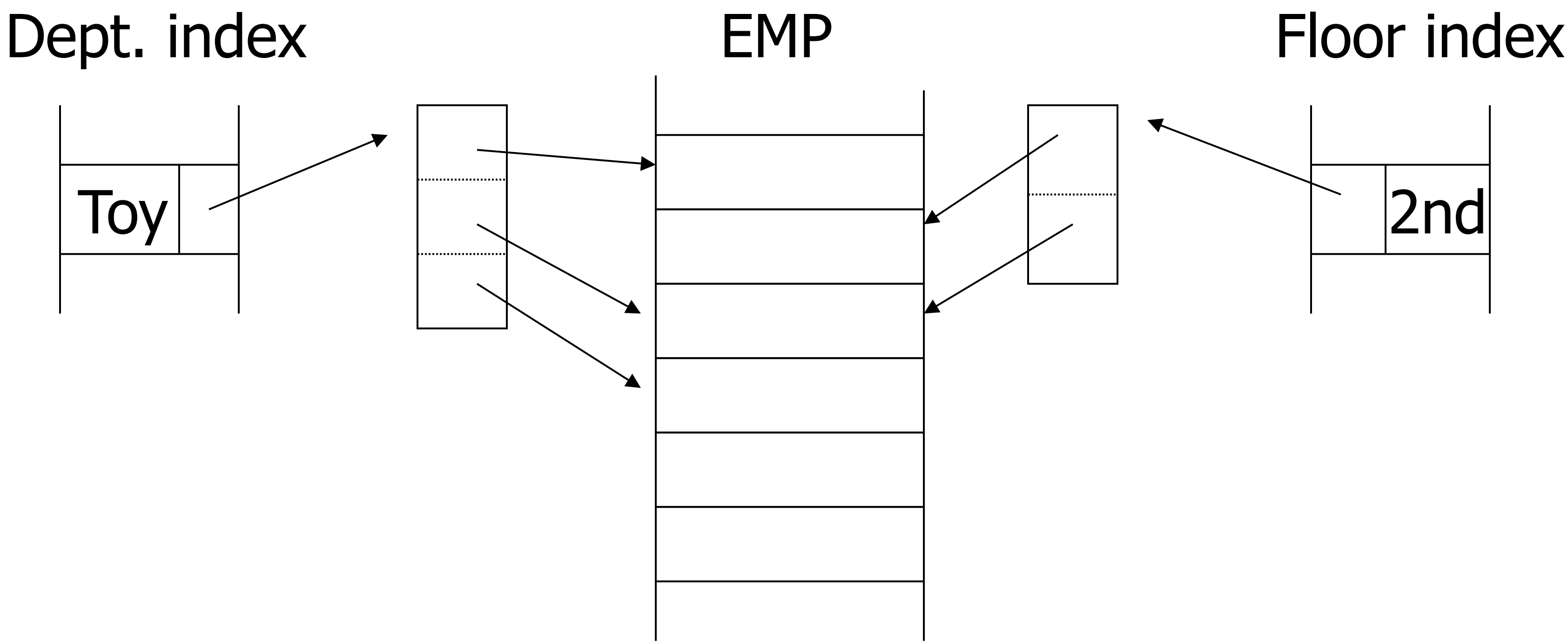
Pointers can be stored in separate blocks

Why "bucket" idea is useful

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

See the following query

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



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

Summary so far

- Conventional index
  - Basic Ideas: sparse, dense, multi-level…
  - Duplicate Keys
  - Deletion/Insertion
  - Secondary indexes

Conventional indexes

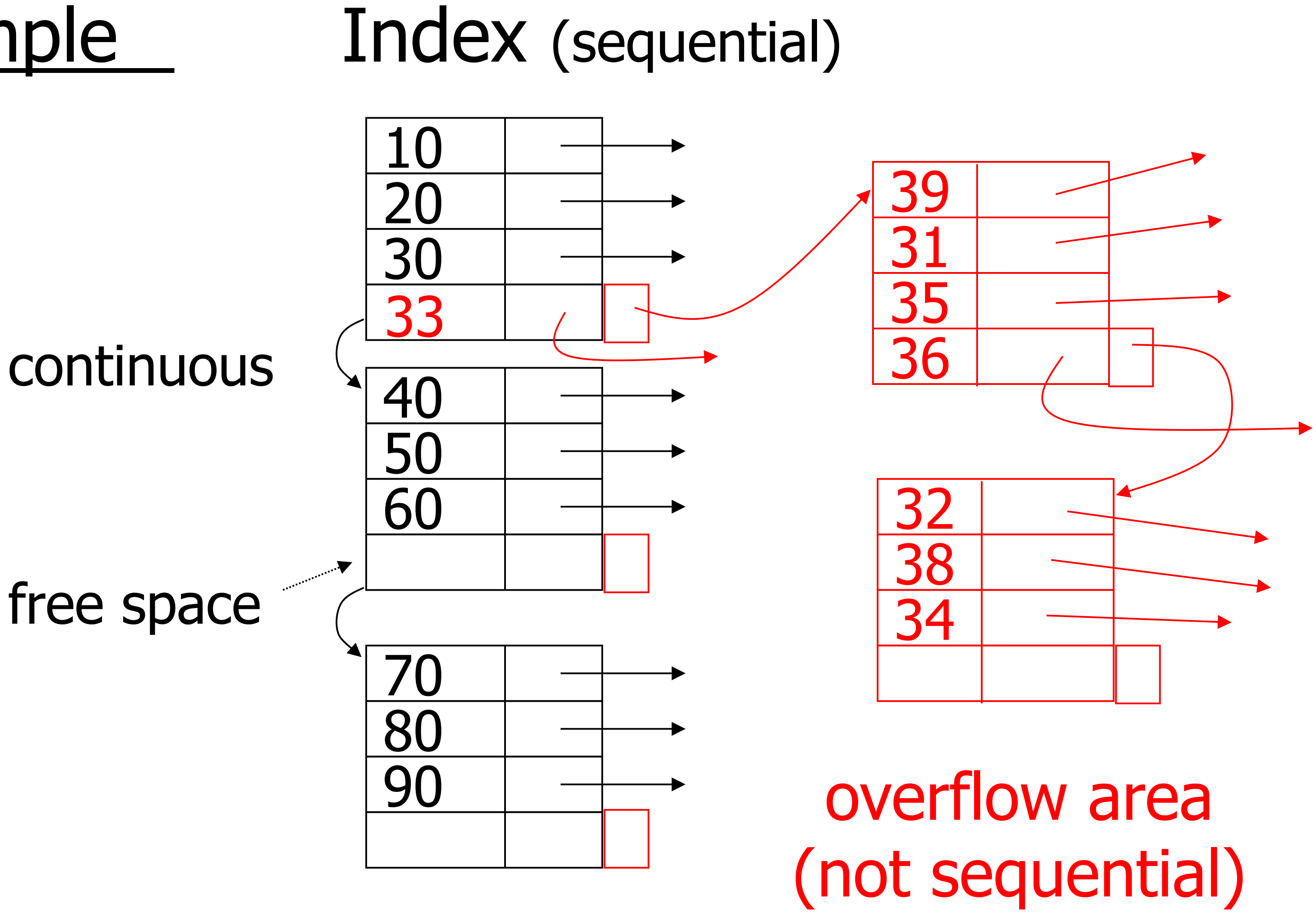
Advantage:

- Simple
- Index is sequential file
  good for scans

Disadvantage:

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

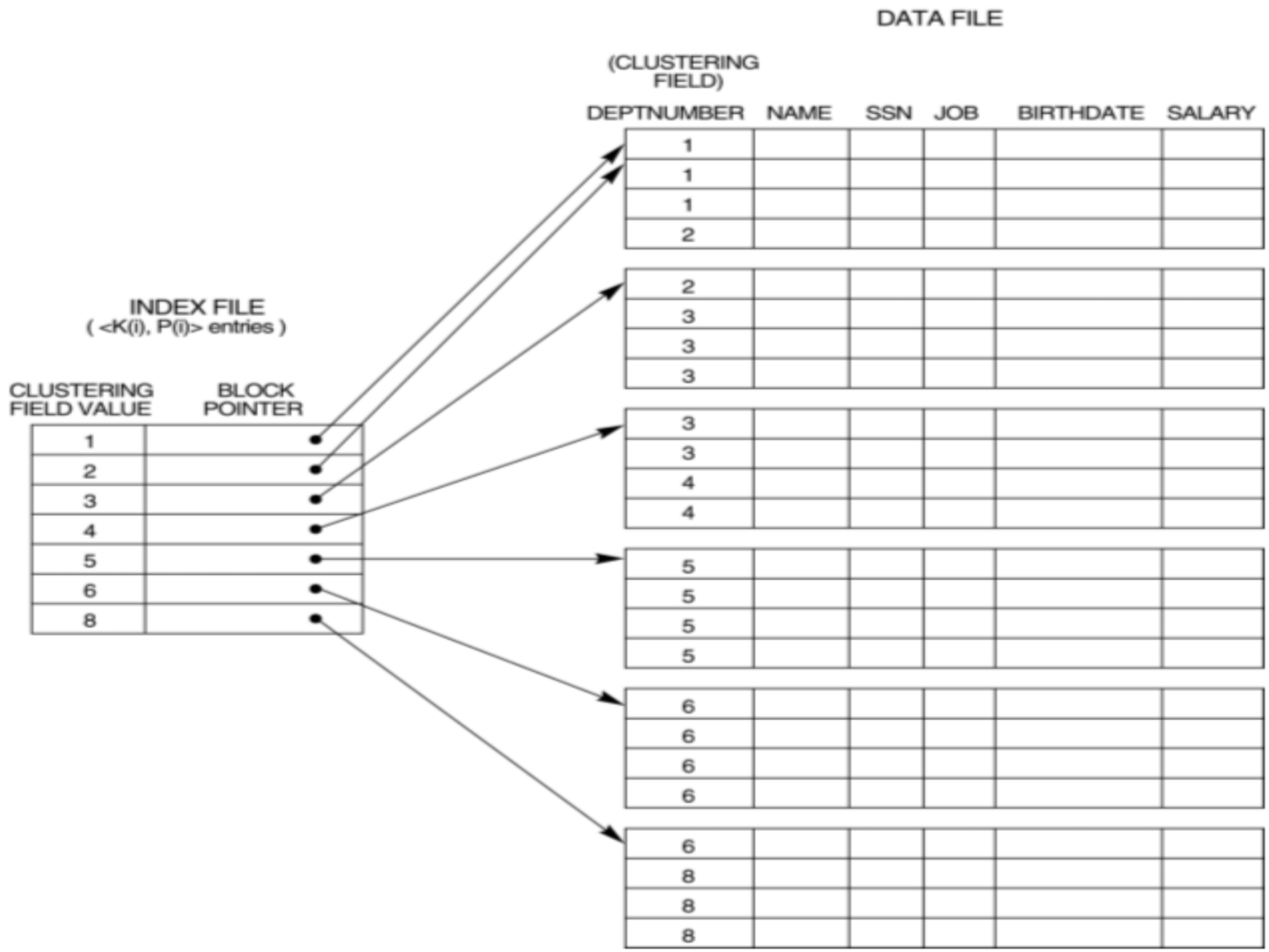
Example



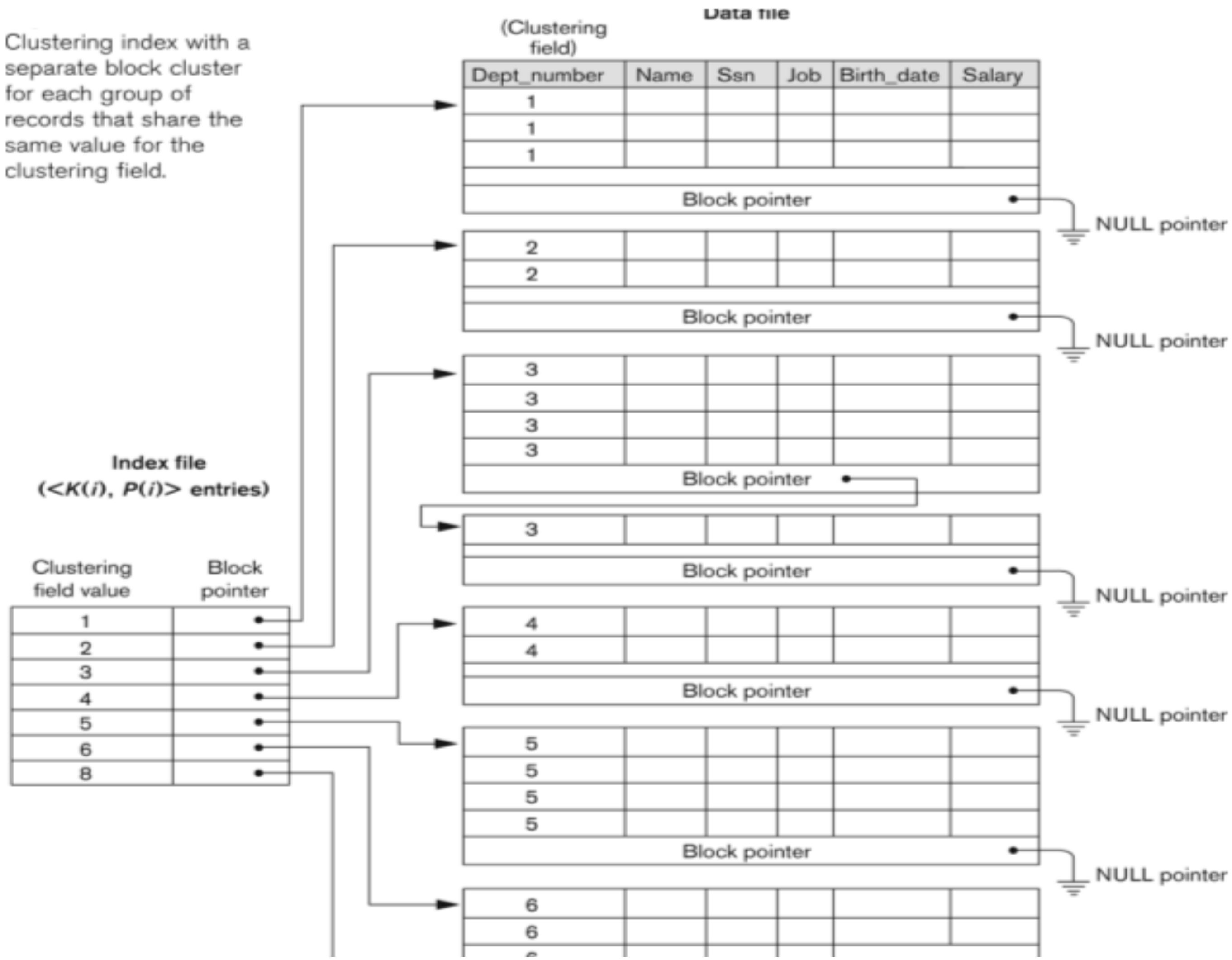
Index (sequential)
10
20
30
33
40
50
60
70
80
90
continuous
free space
39
31
35
36
32
38
34
overflow area
(not sequential)

- Clustering Index
  - Defined on an ordered data file
  - The data file is ordered on a *non-key field*.
  - Includes one index entry *for each distinct value* of the field; the index entry points to the first data block that contains records with that field value.
  - It is an example of *non-dense* index where Insertion and Deletion is relatively straightforward with a clustering index.

- Clustering Index



- Clustering Index version 2



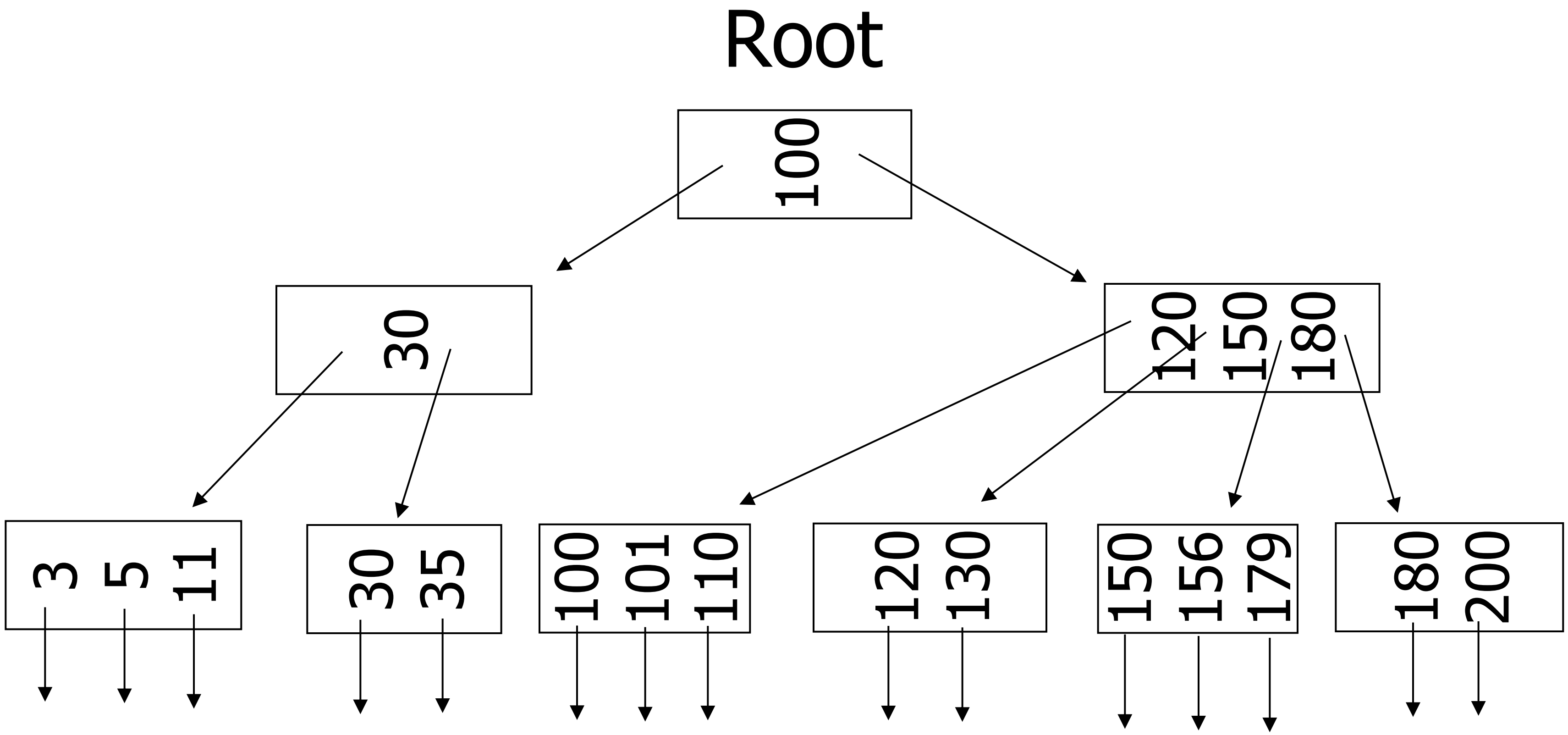
Outline:

- 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

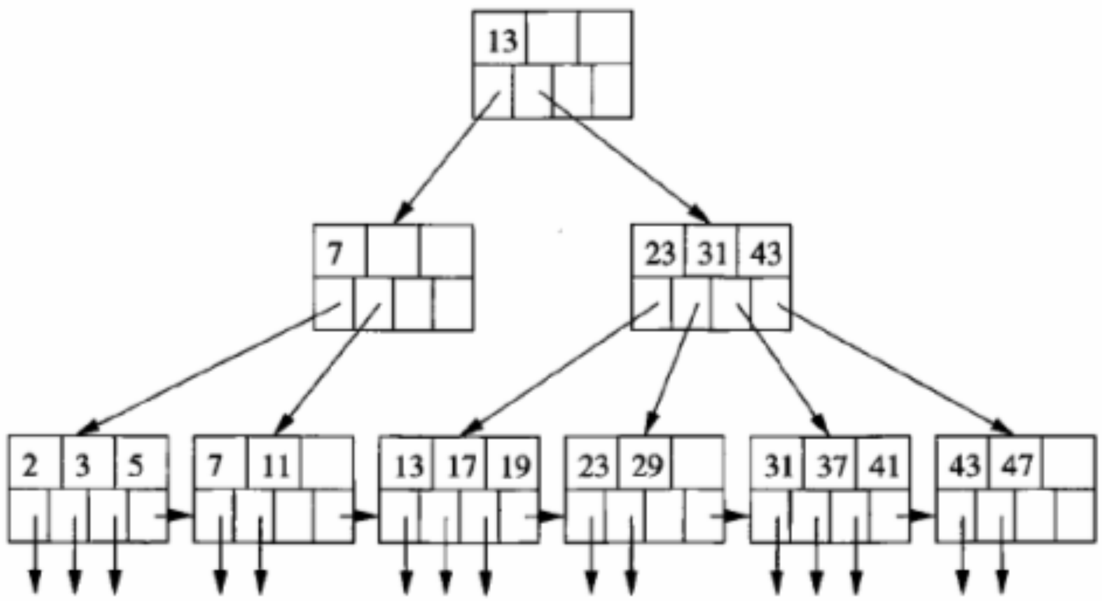
n=3



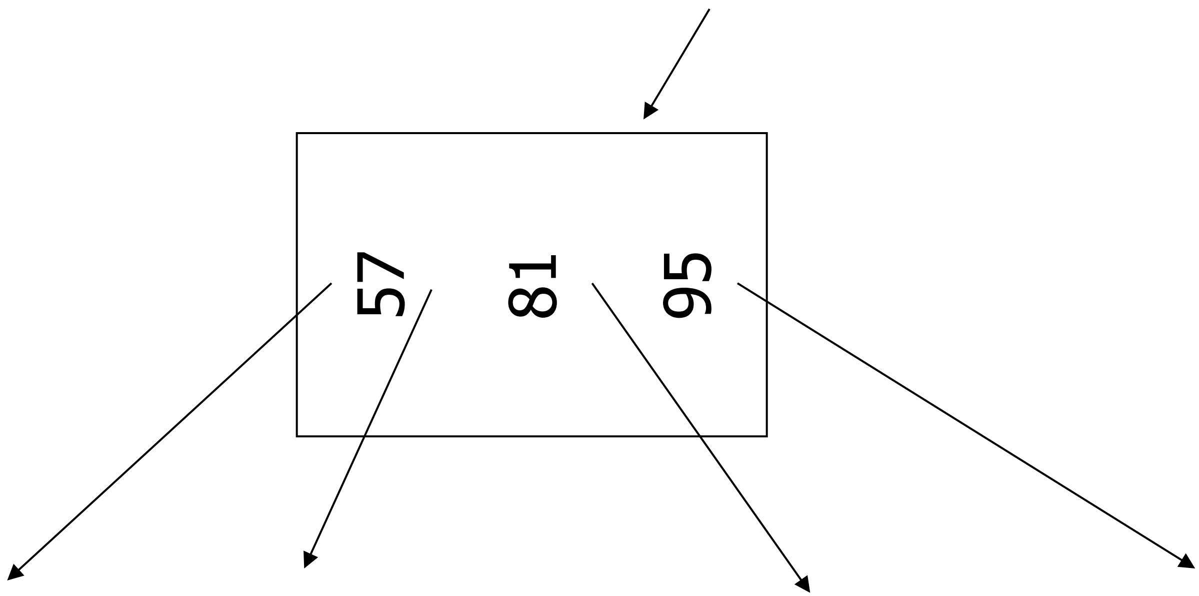
Lookup in a B+ tree

Useful for range queries too

SELECT * FROM R WHERE R.o >= a AND R.o <= b;

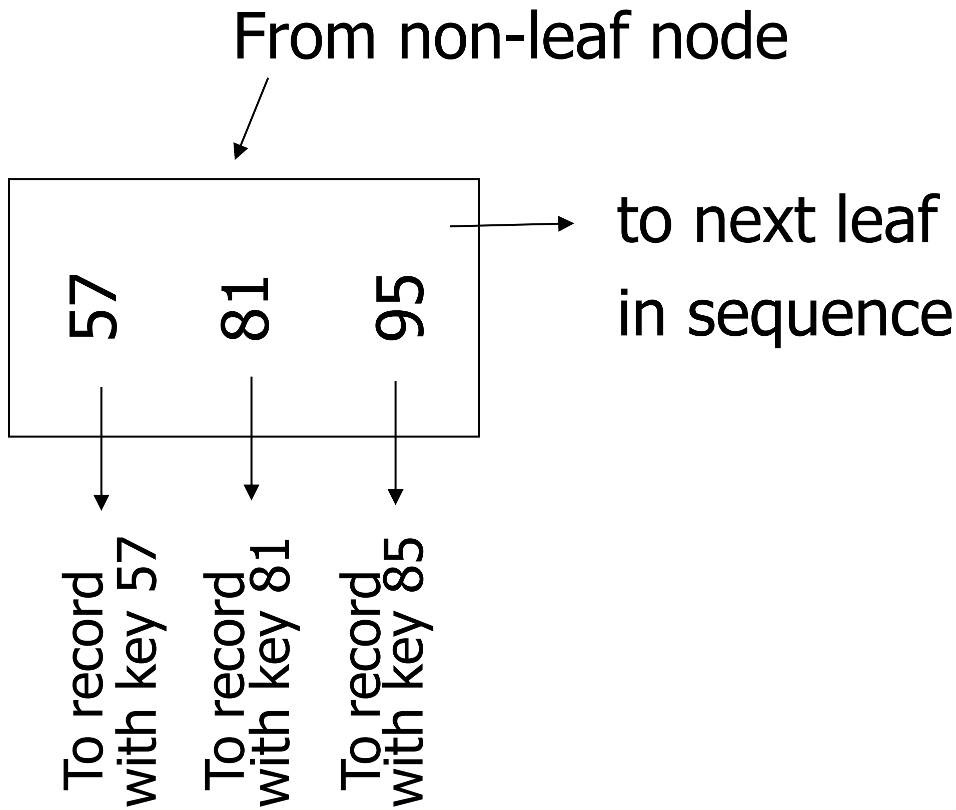


Sample non-leaf



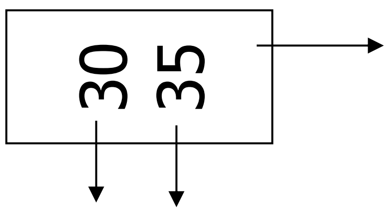
to keys	to keys	to keys	to keys
< 57	$57 \leq k < 81$	$81 \leq k < 95$	≥ 95

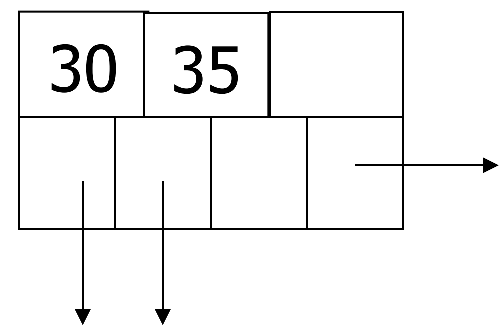
Sample leaf node:



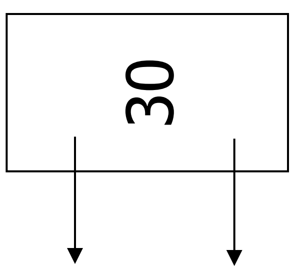
In textbook's notation   n=3

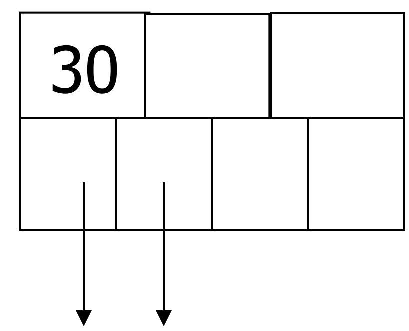
Leaf:





Non-leaf:





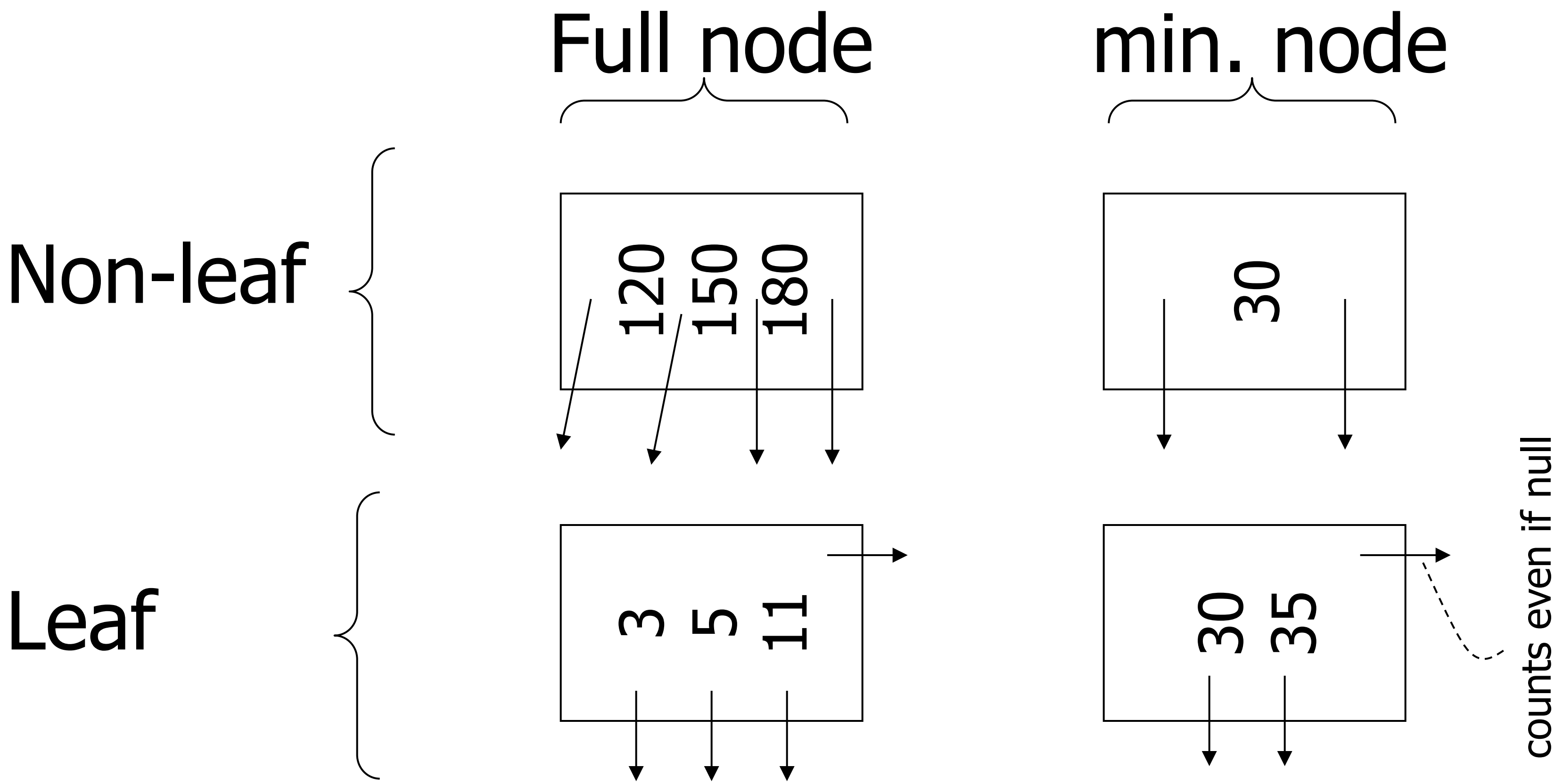
Size of nodes: n+1 pointers, n keys (fixed)

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"
(to next leaf)

(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

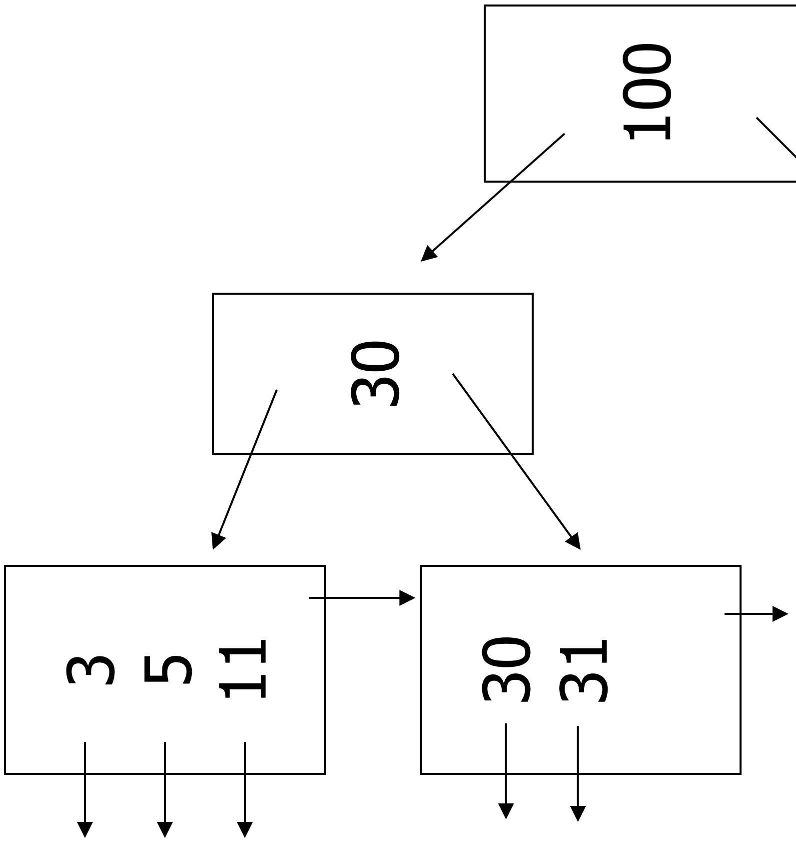
(b) leaf overflow

(c) non-leaf overflow

(d) new root

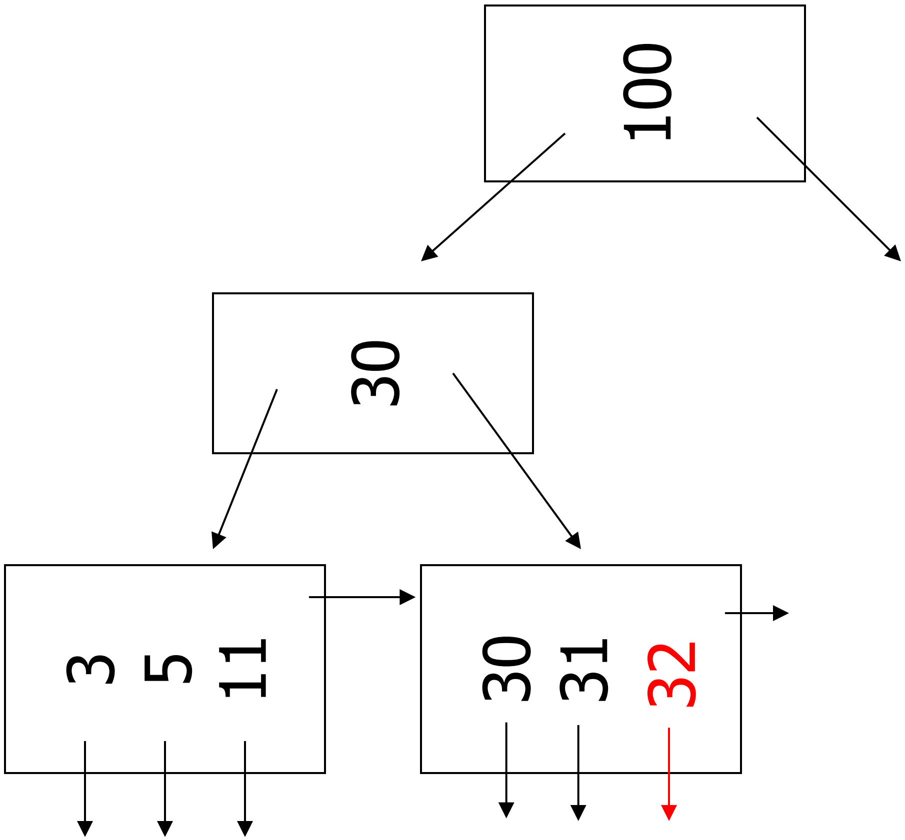
(a) Insert key = 32

n=3



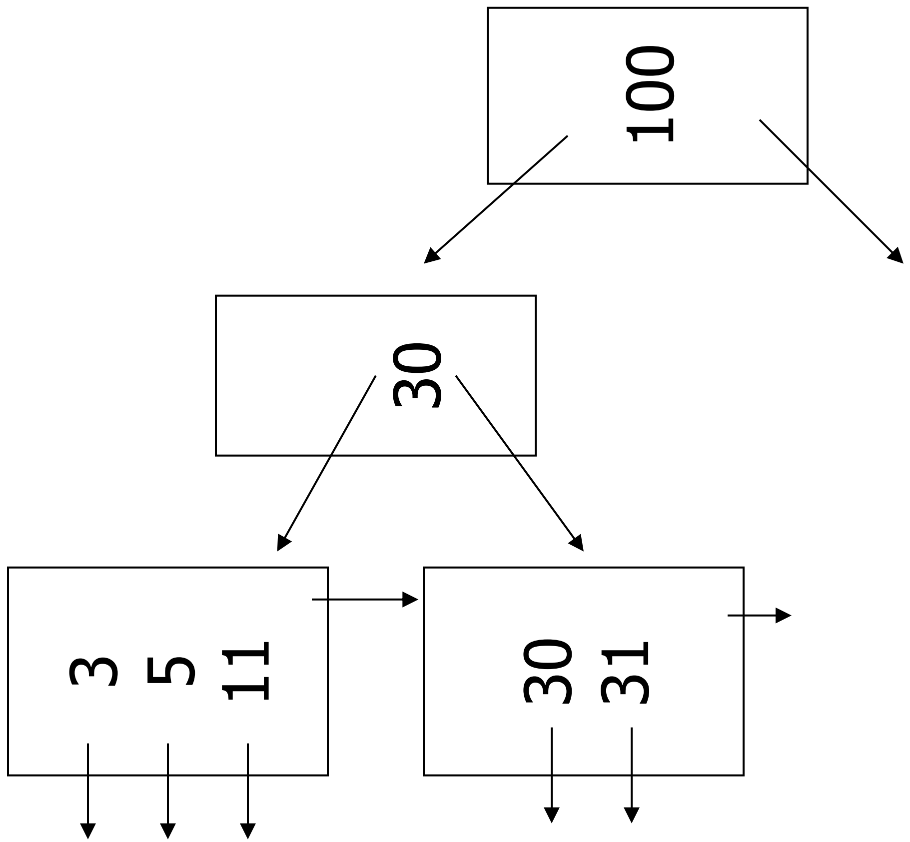
(a) Insert key = 32

n=3



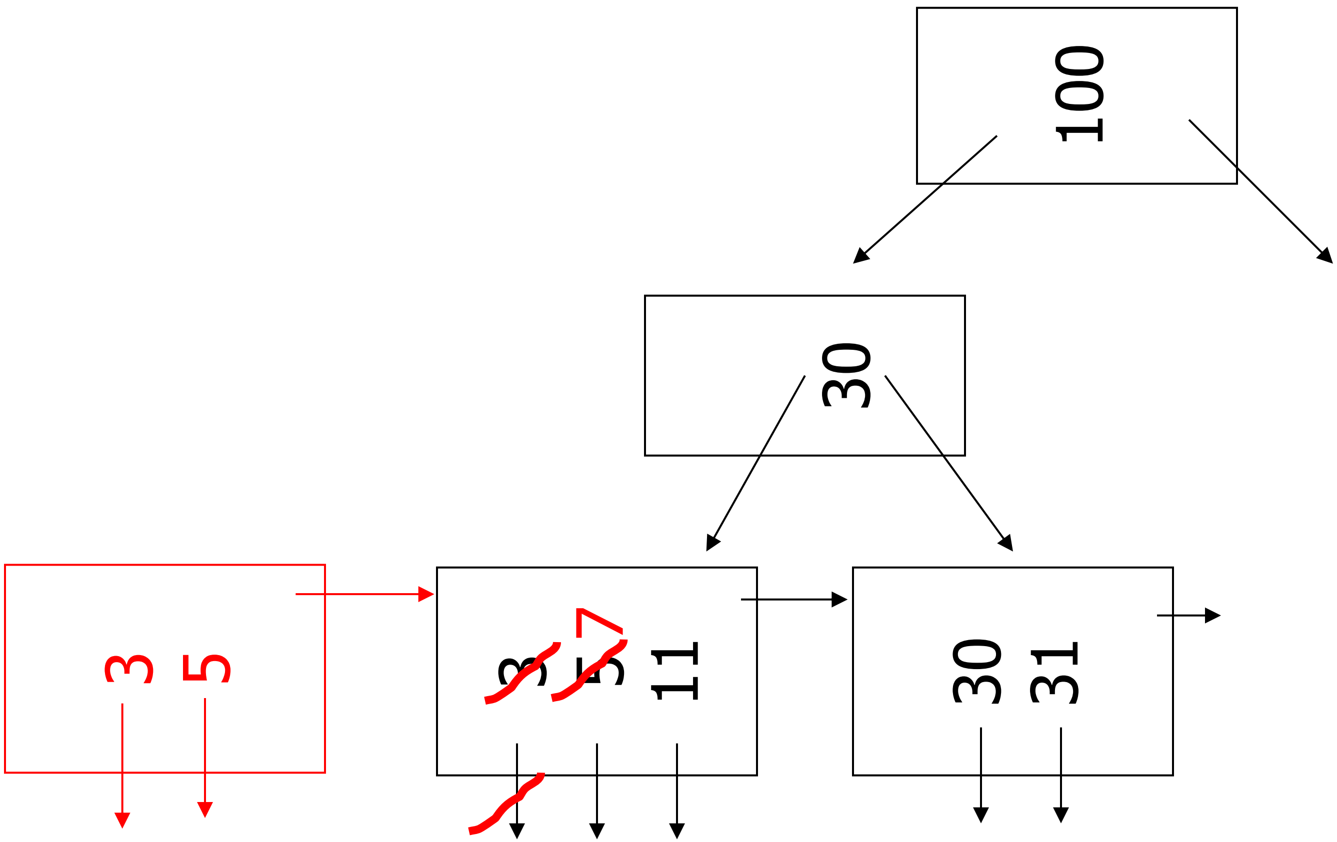
(a) Insert key = 7

n=3



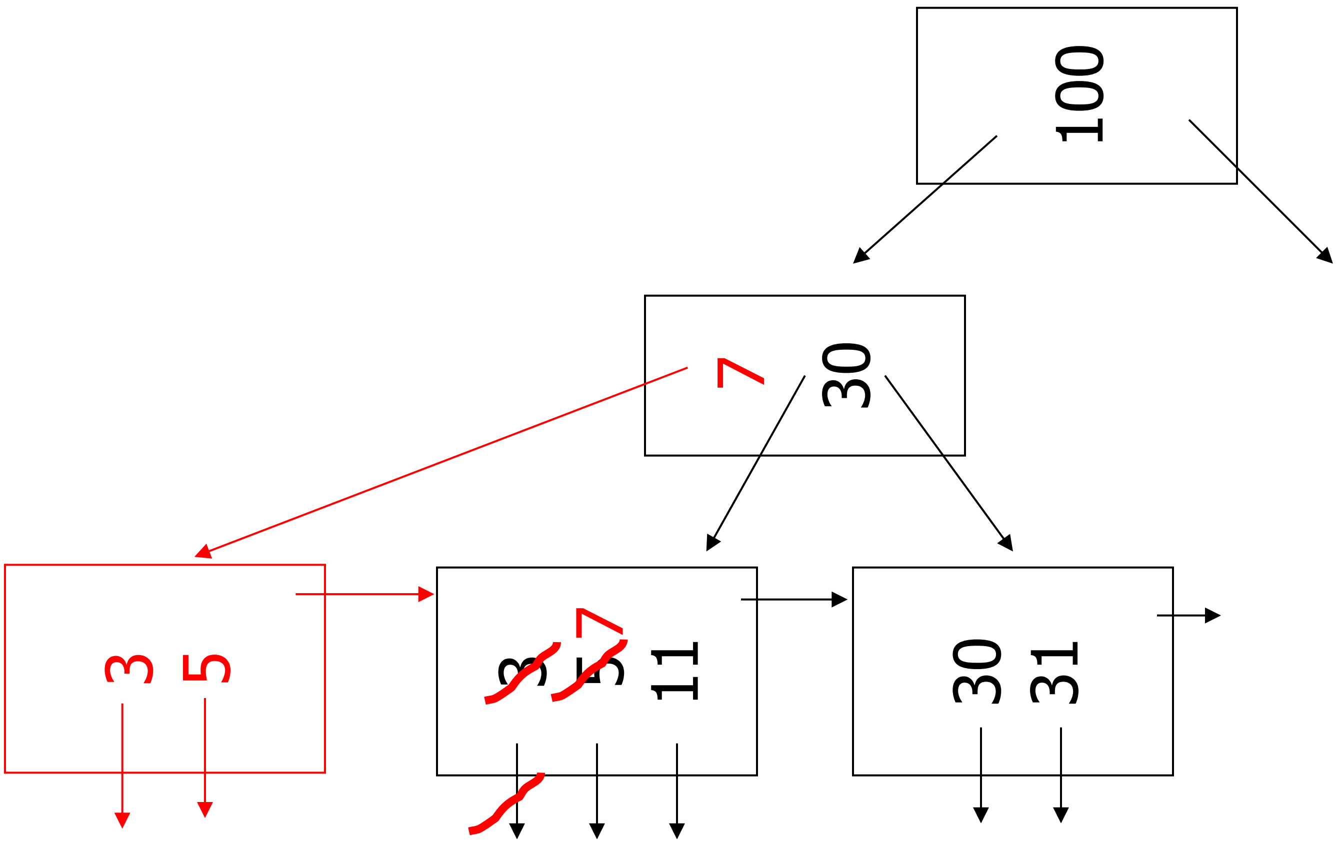
(a) Insert key = 7

n=3



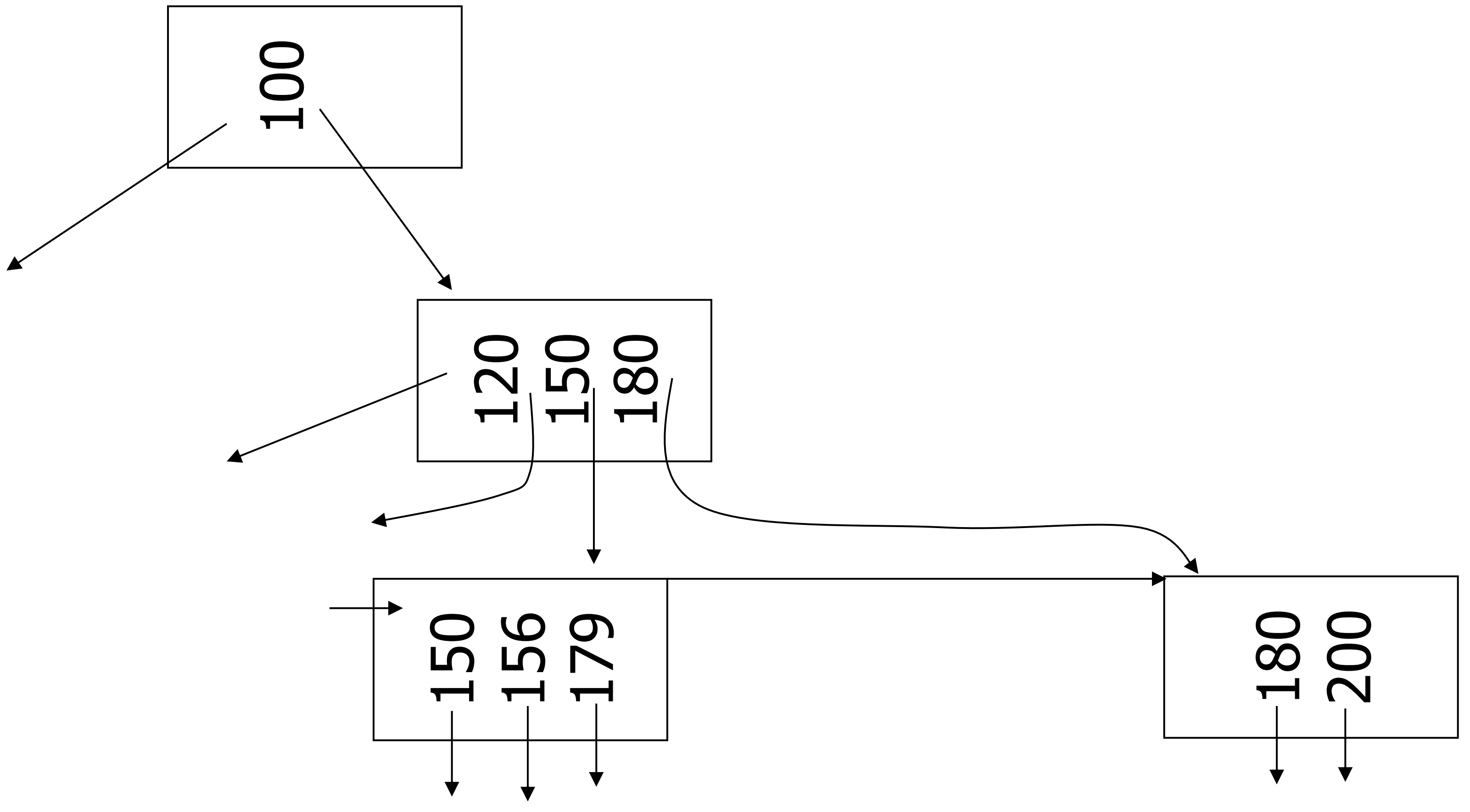
(a) Insert key = 7

n=3



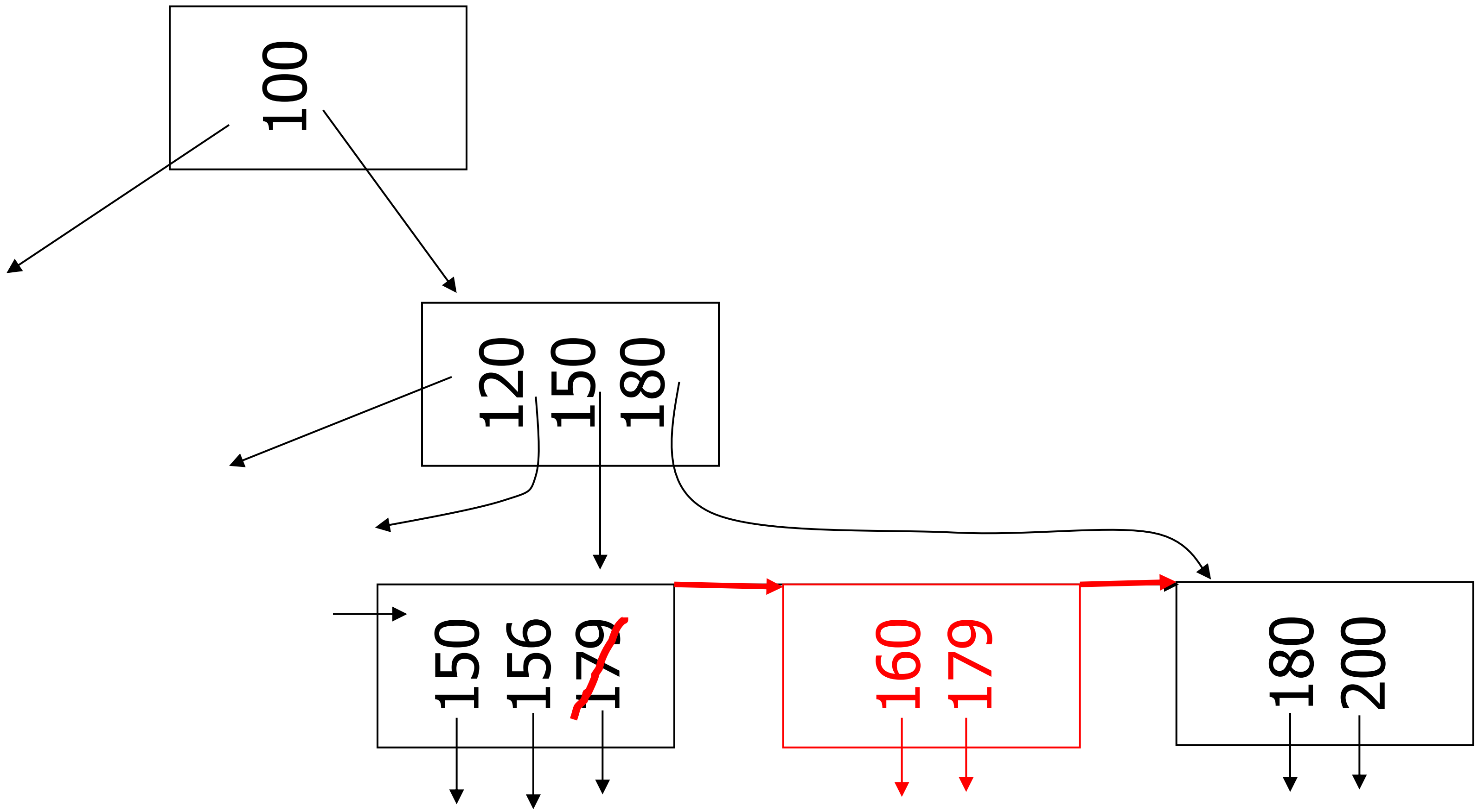
(c) Insert key = 160

n=3



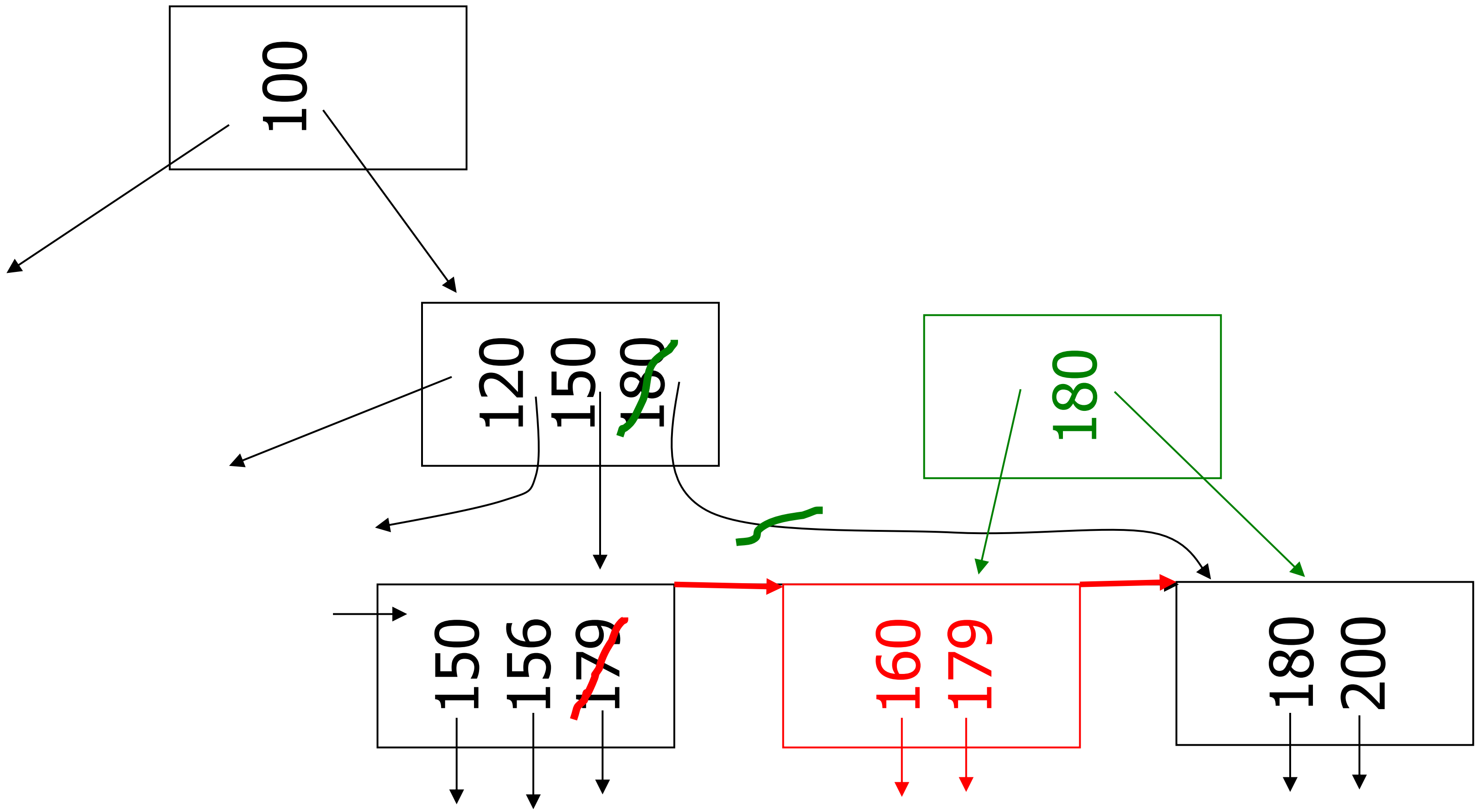
(c) Insert key = 160

n=3



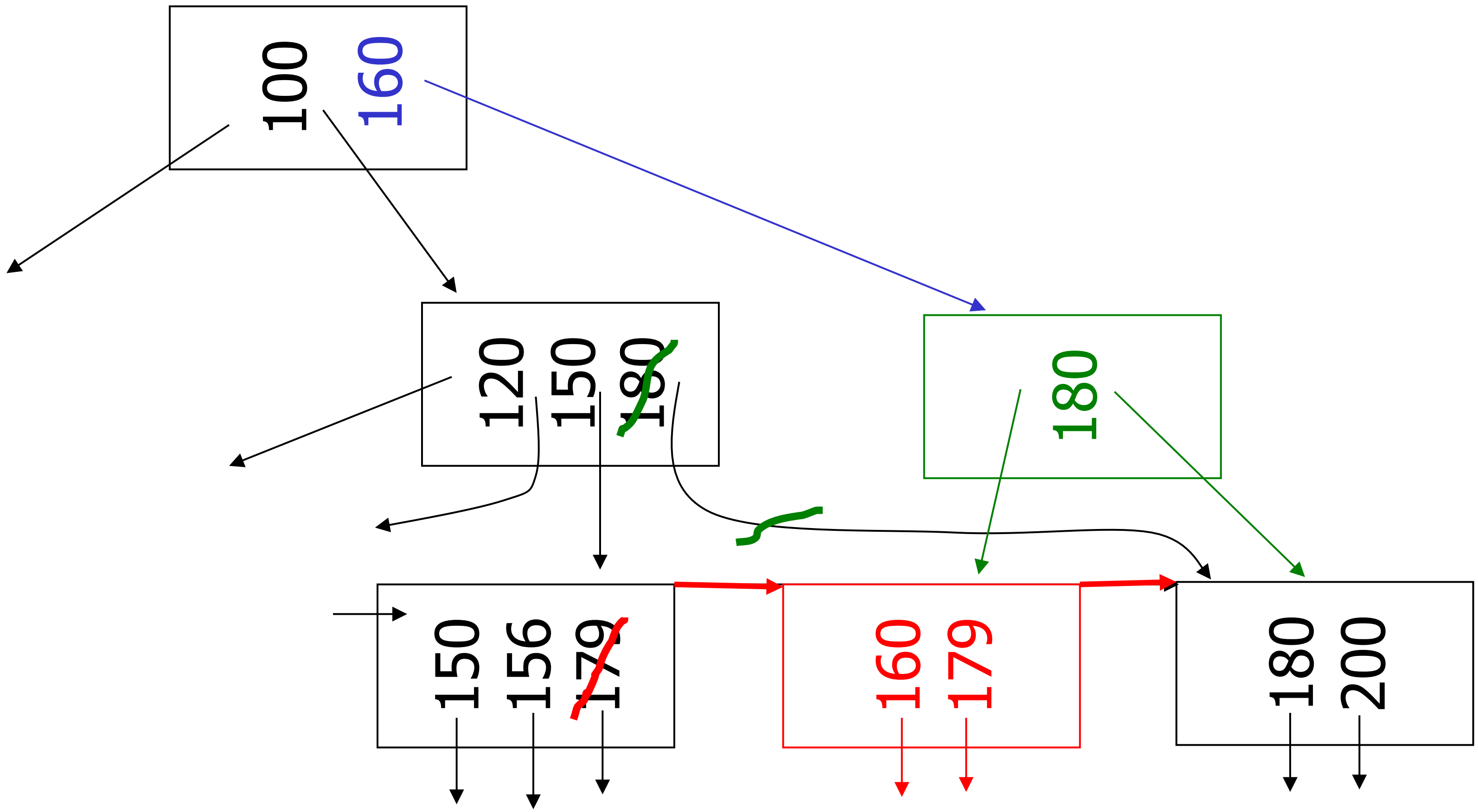
(c) Insert key = 160

n=3



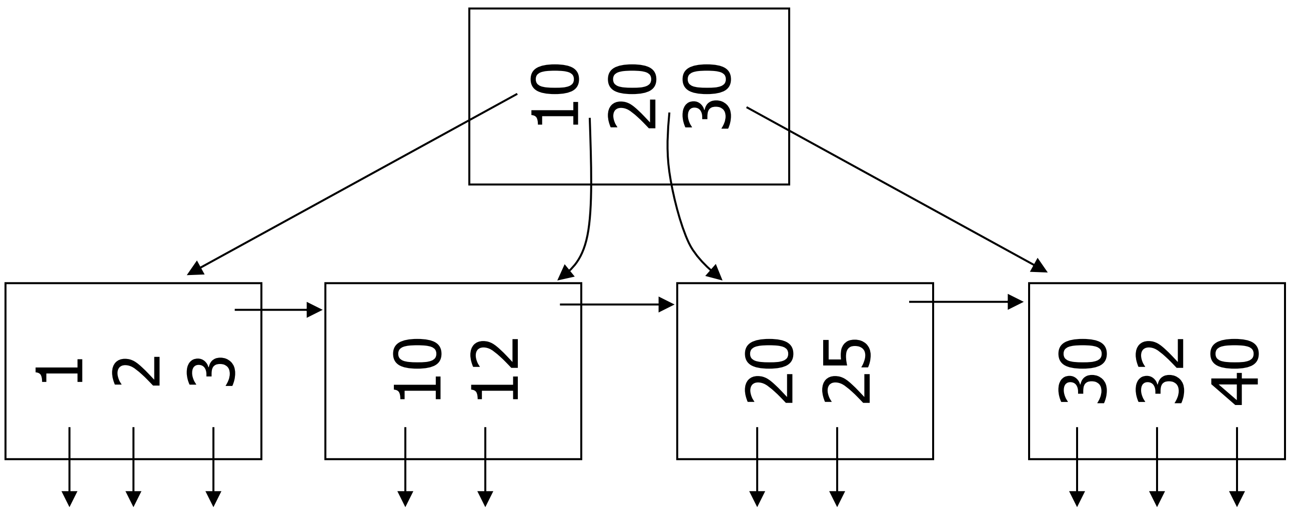
(c) Insert key = 160

n=3



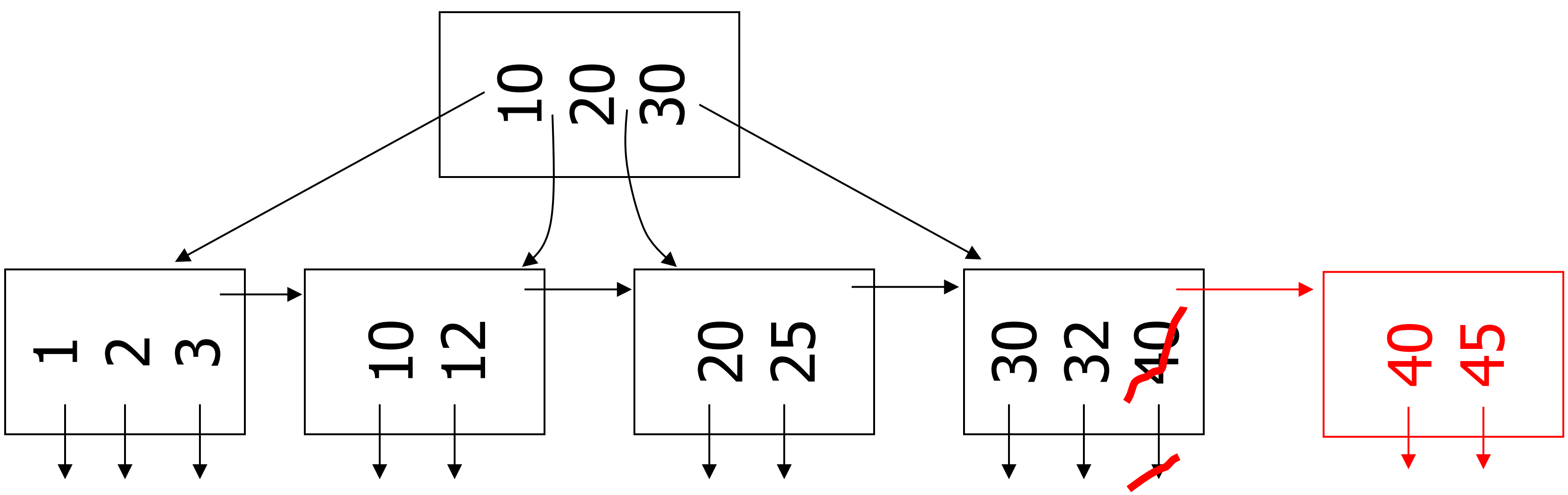
(d) New root, insert 45

n=3



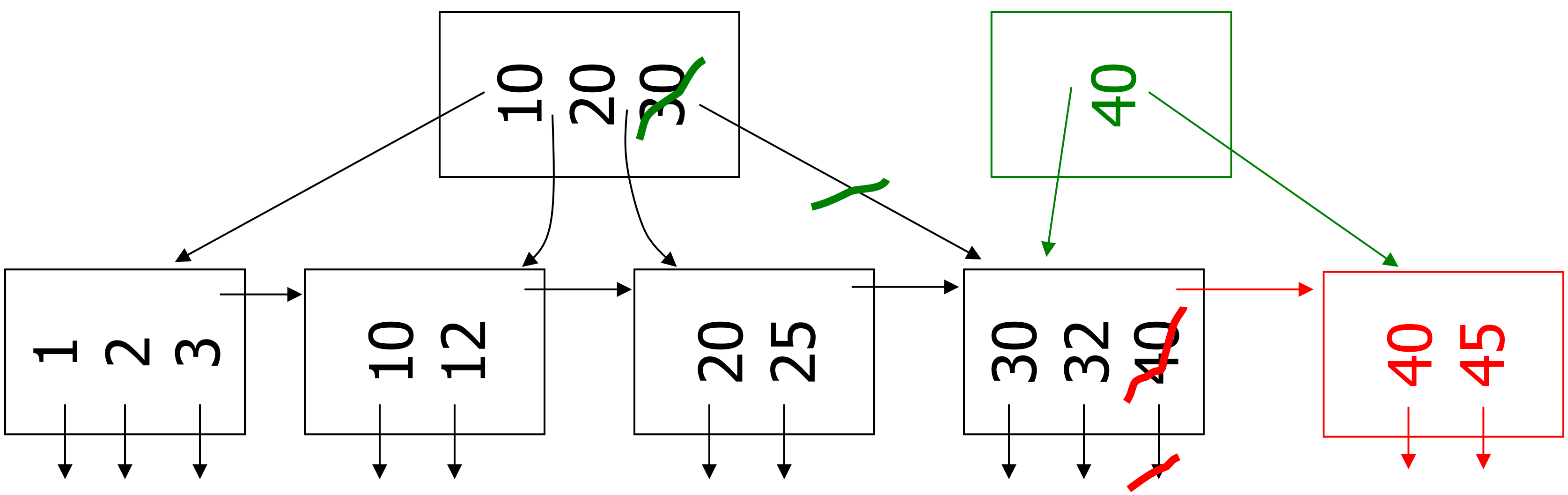
(d) New root, insert 45

n=3



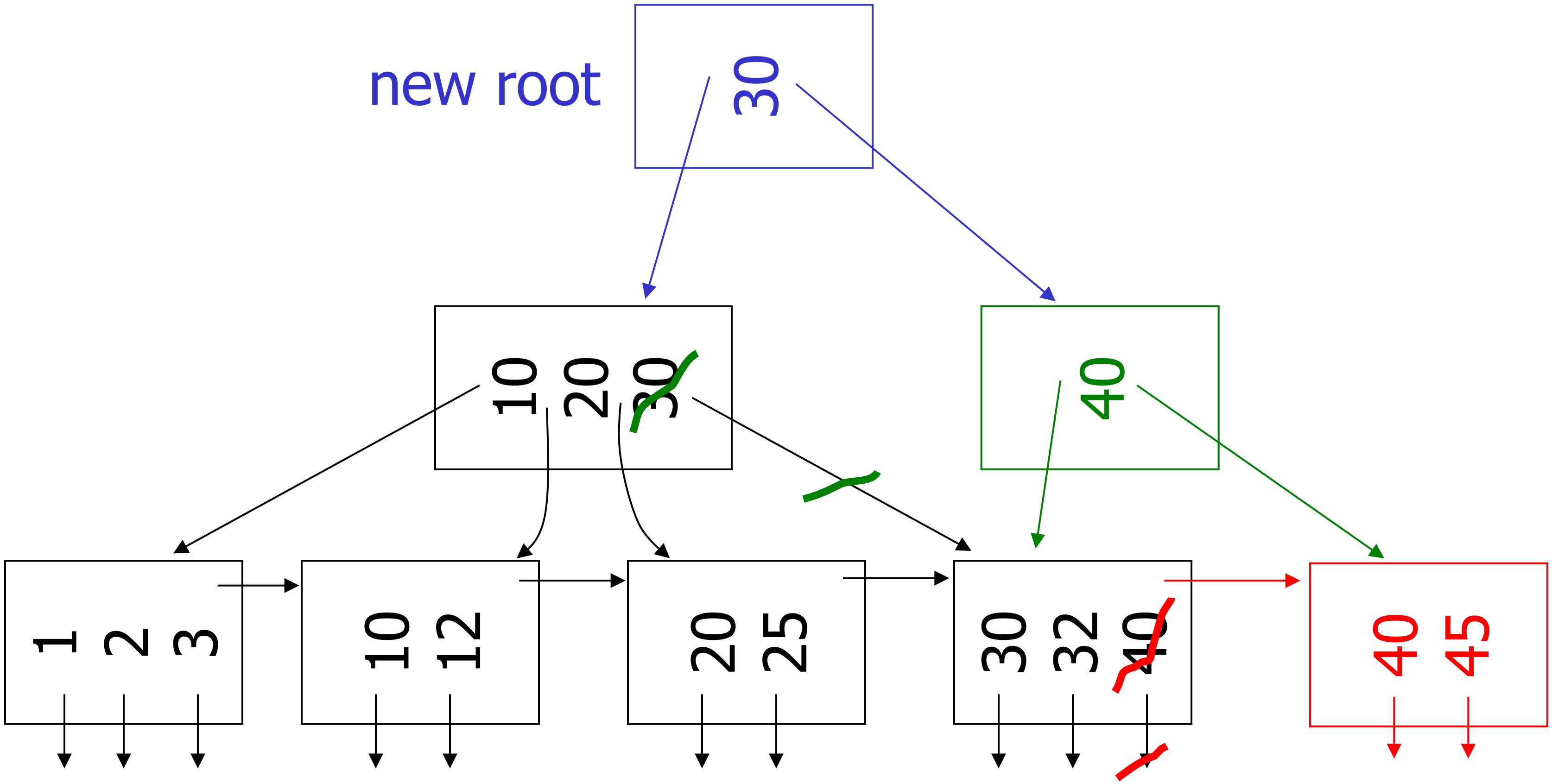
(d) New root, insert 45

n=3



(d) New root, insert 45

n=3



Deletion from B+tree

(a) Simple case - no example

(b) Coalesce with neighbor (sibling)

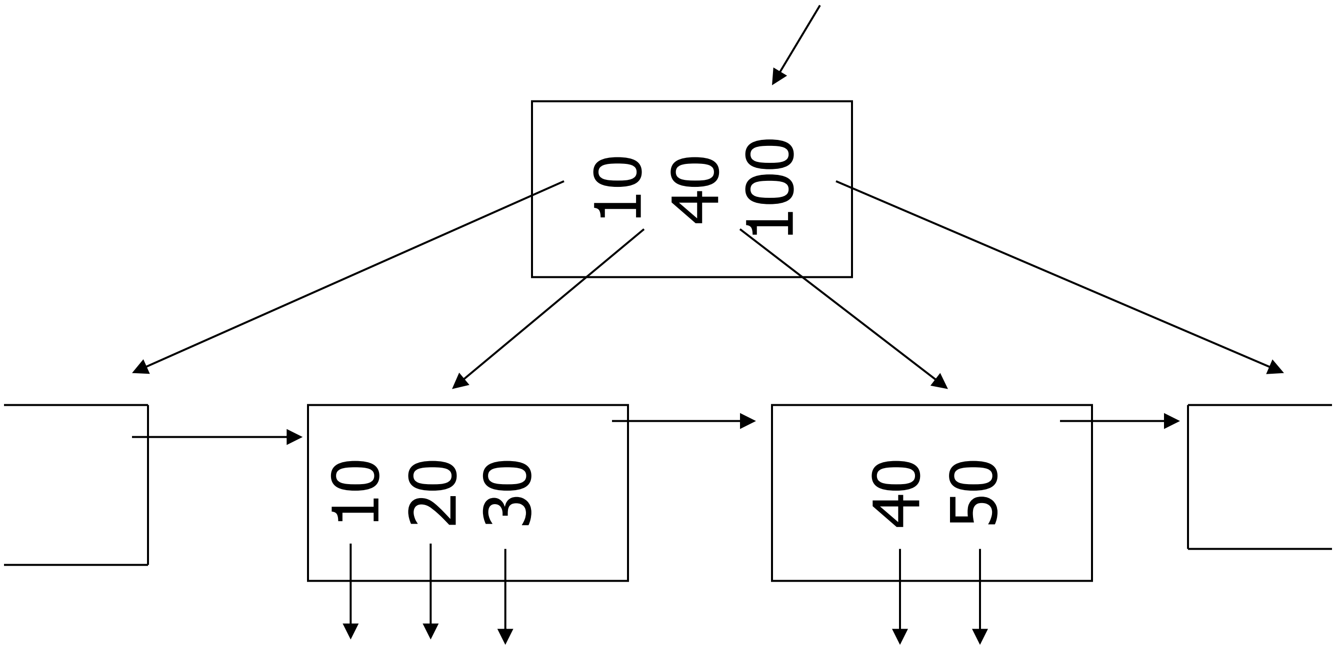
(c) Re-distribute keys

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

(b) Coalesce with sibling

– Delete 50

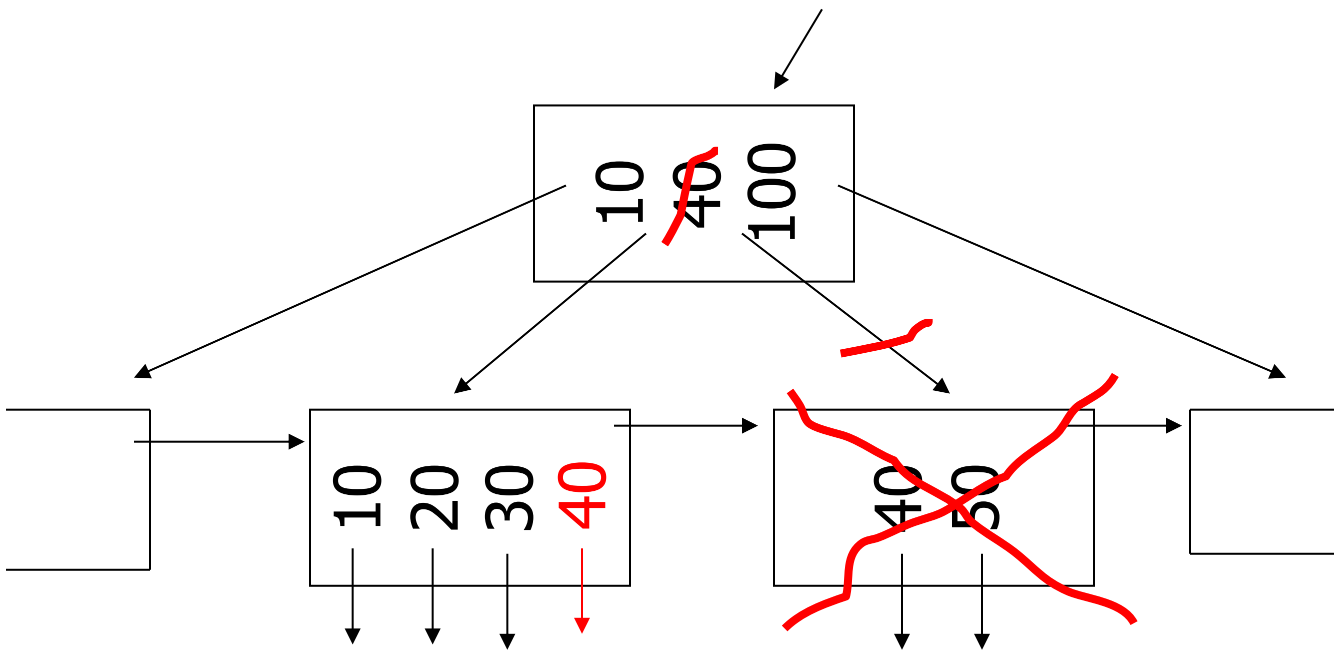
n=4



(b) Coalesce with sibling

– Delete 50

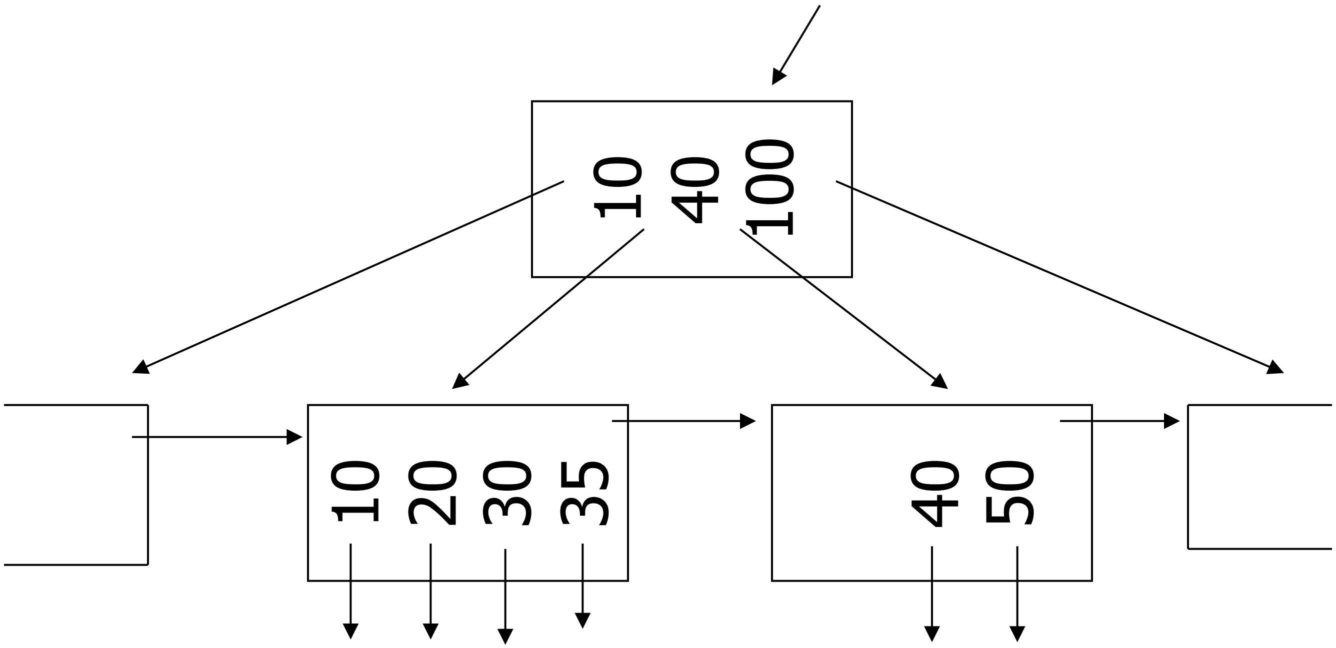
n=4



(c) Redistribute keys

– Delete 50

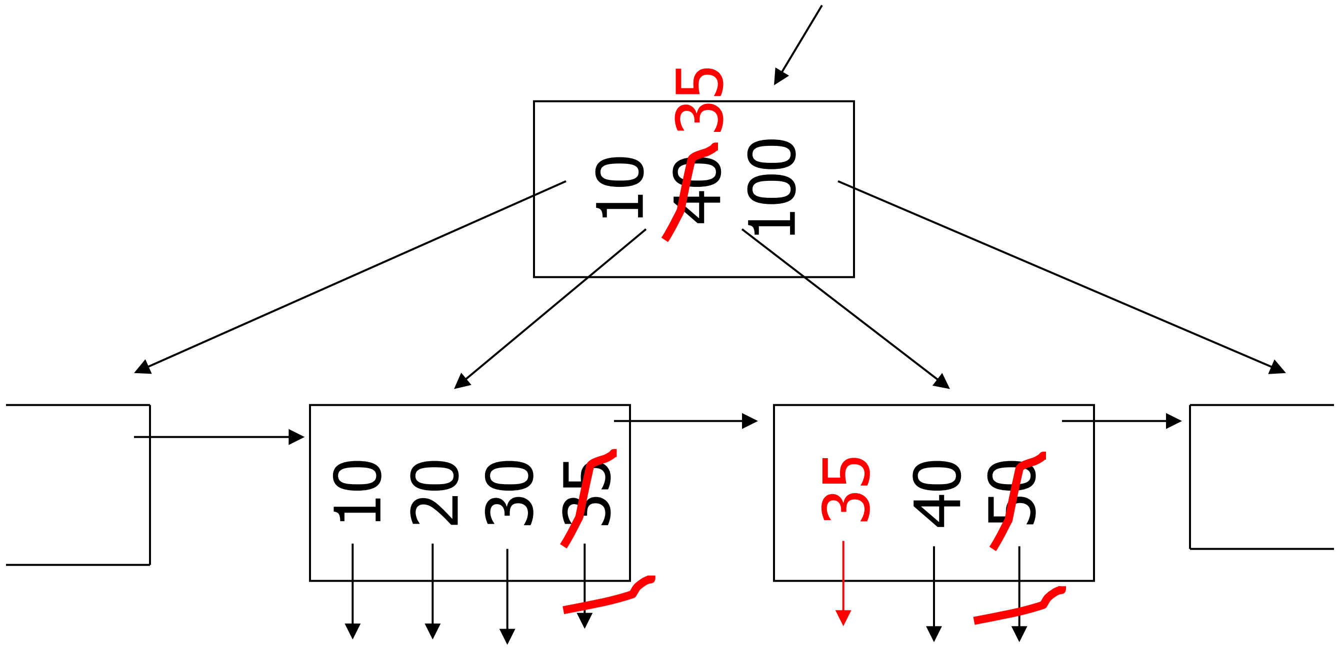
n=4



(c) Redistribute keys

– Delete 50

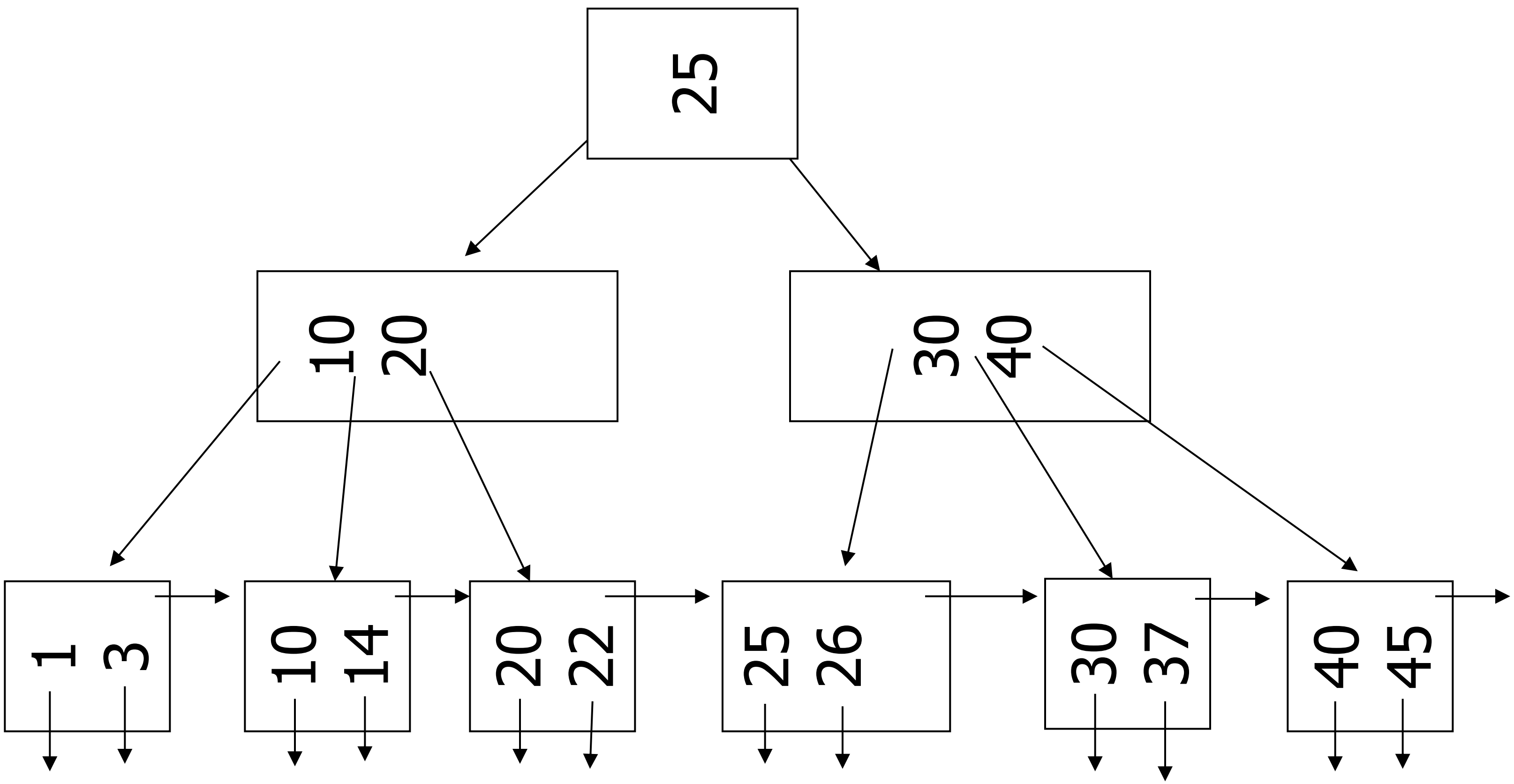
n=4



(d) Non-leaf coalese

– Delete 37

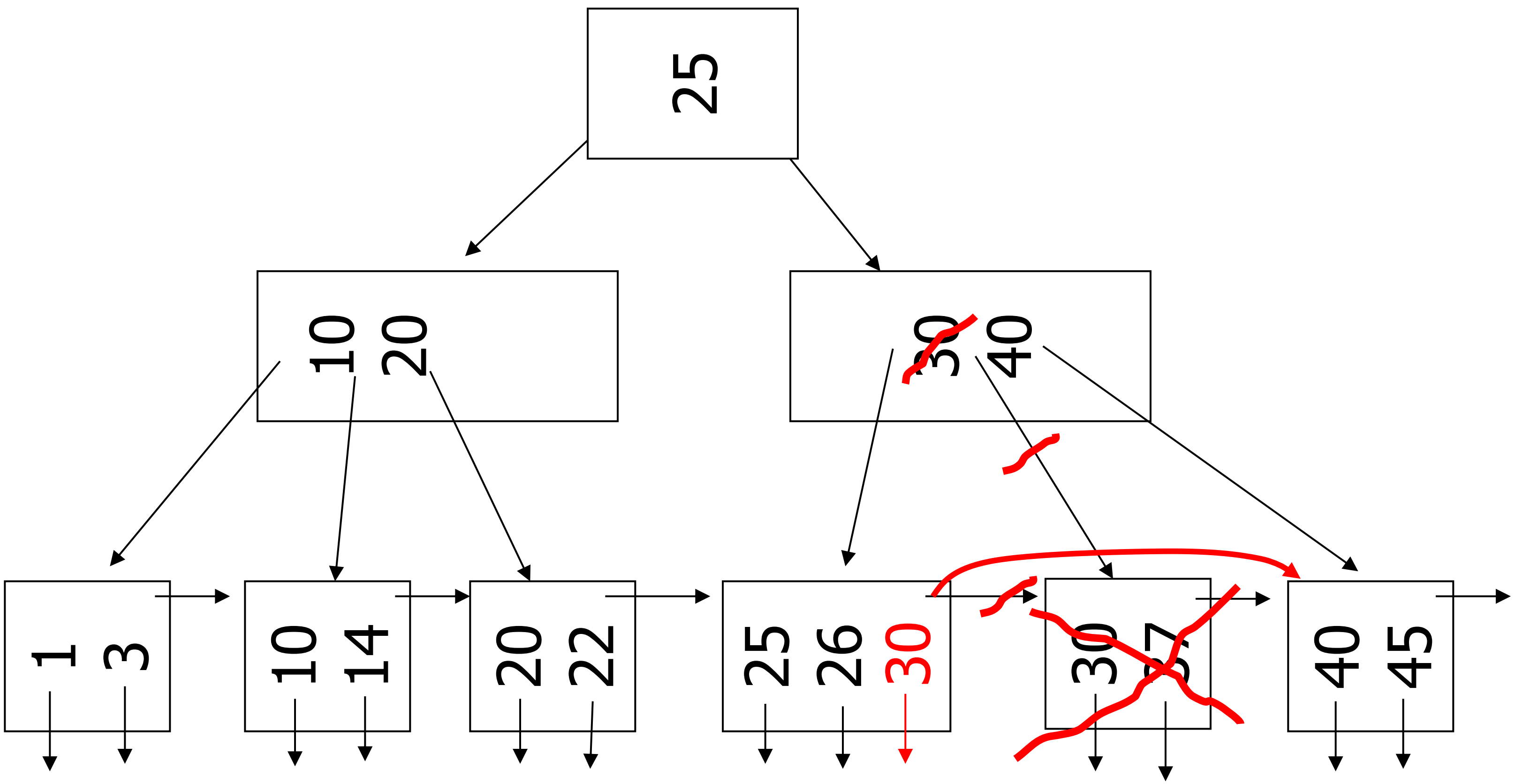
n=4



(d) Non-leaf coalese

n=4

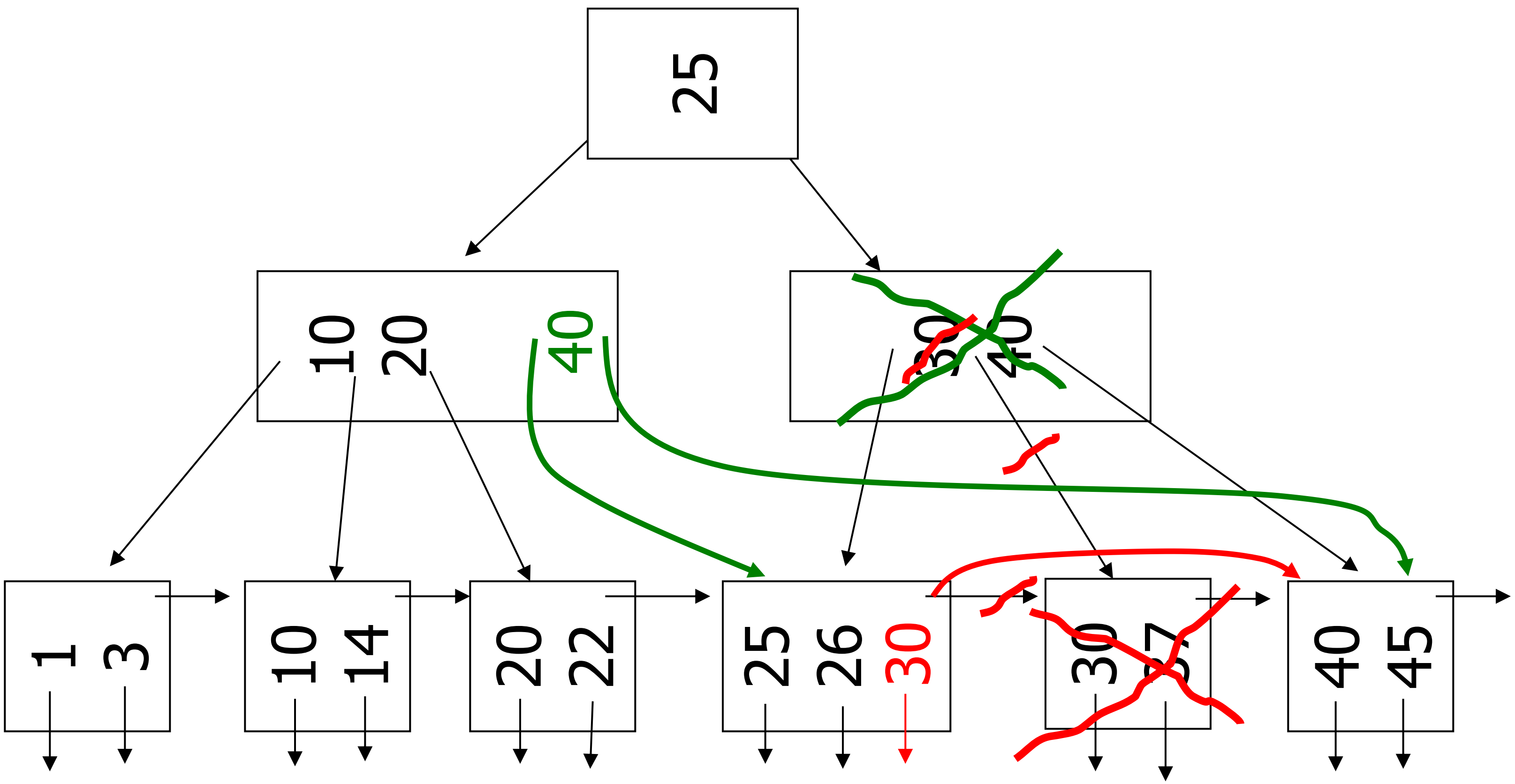
– Delete 37



(d) Non-leaf coalese

– Delete 37

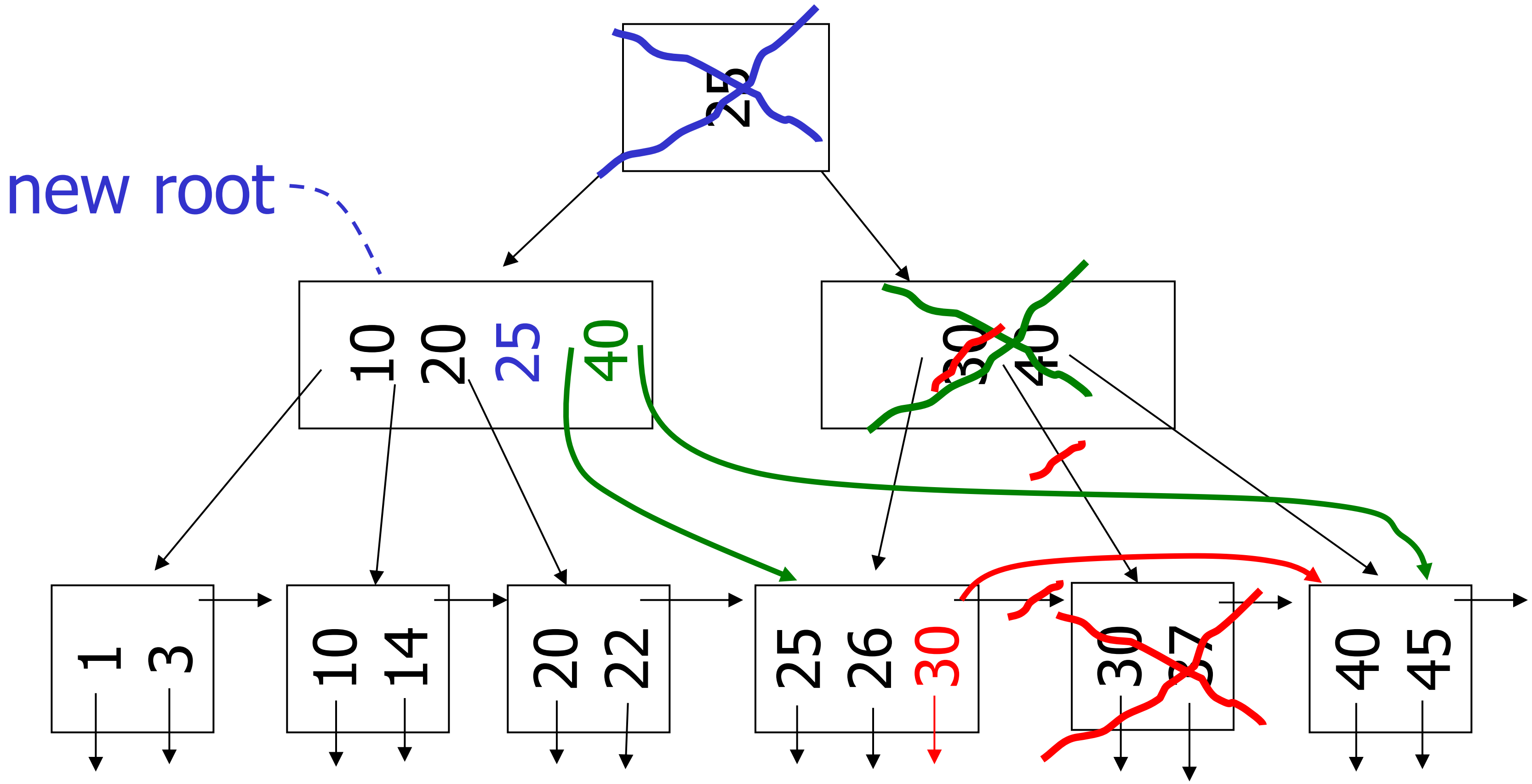
n=4



(d) Non-leaf coalese

– Delete 37

n=4



B+tree deletions in practice

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

- Speaking of buffering…

  Is LRU a good policy for B+tree buffers?

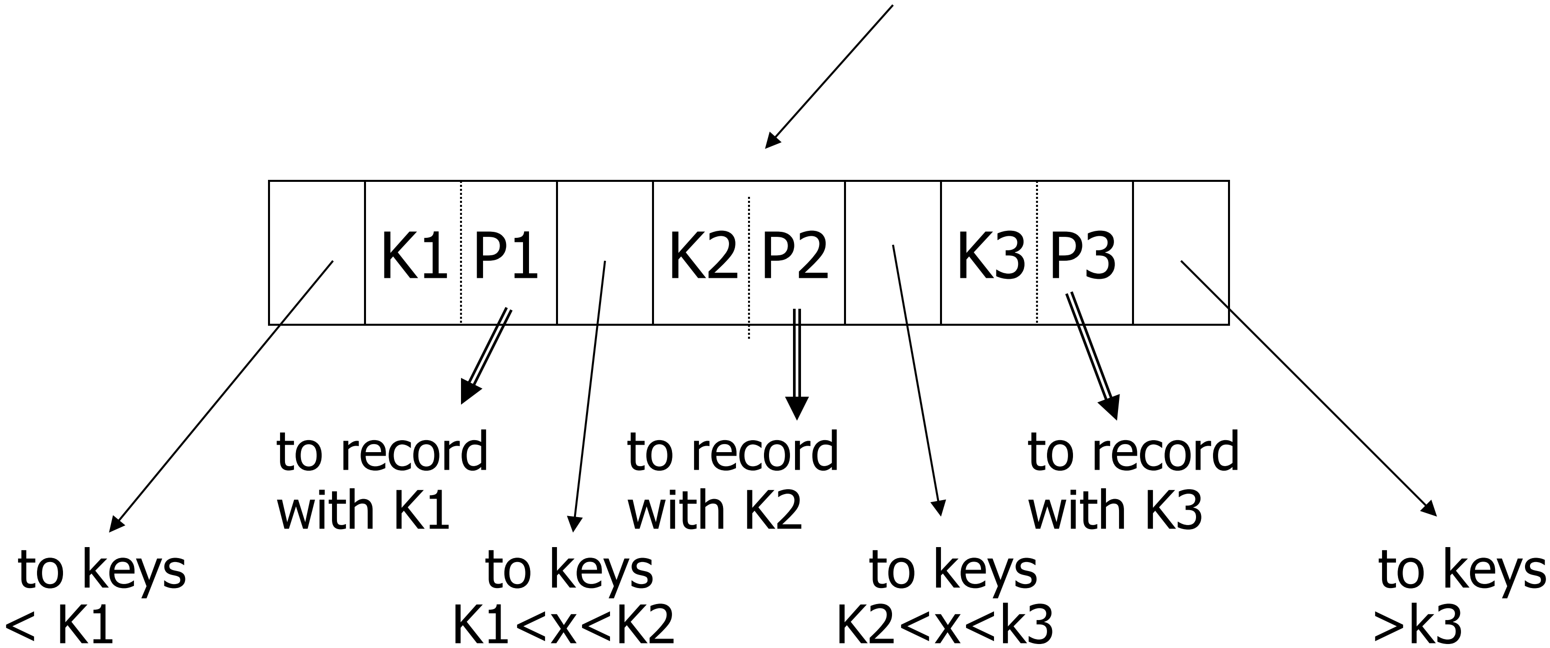
  → Of course not!

  → Should try to keep root in memory at all times

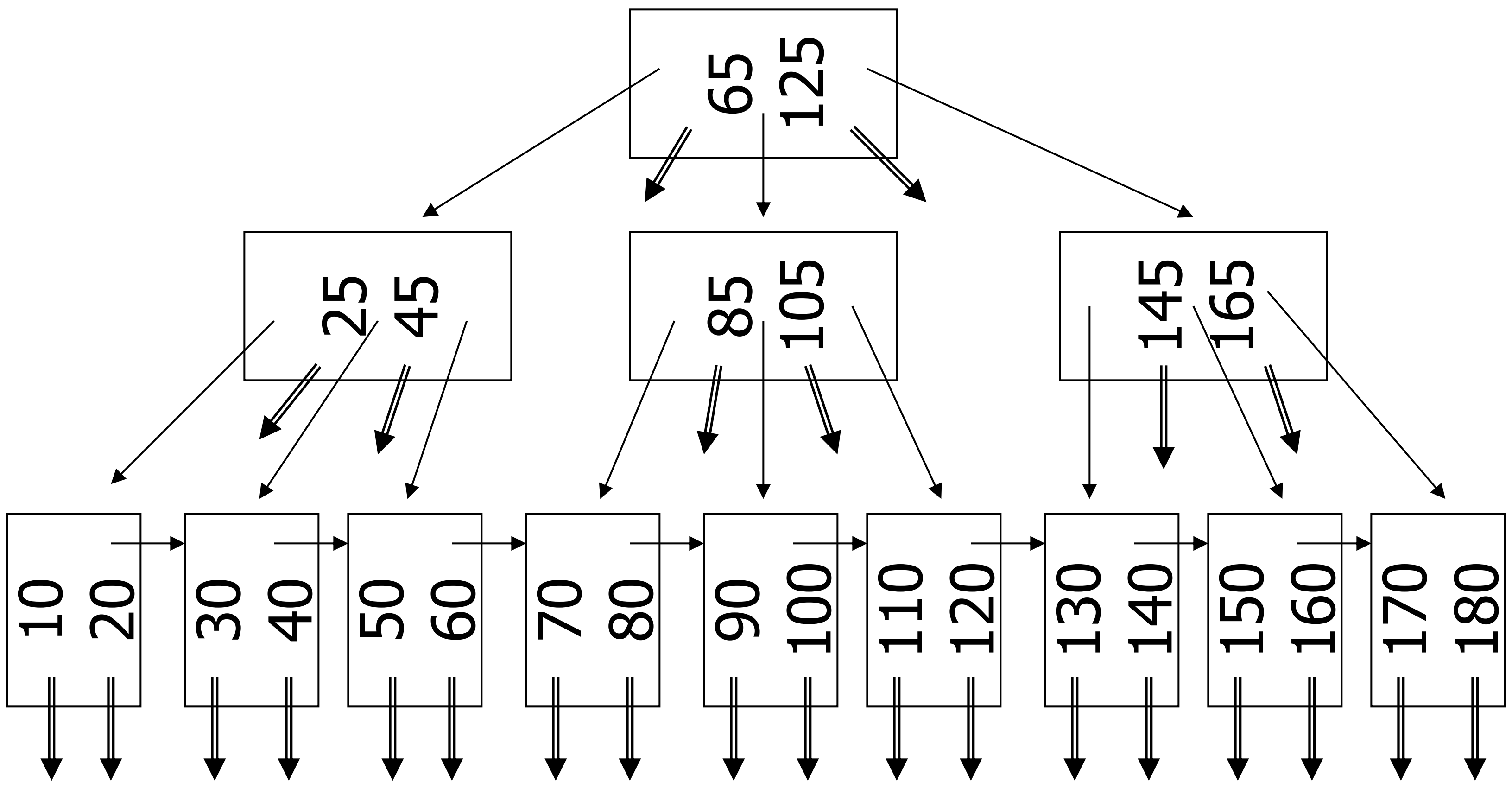
  (and perhaps some nodes from second level)

Variation on B+tree: B-tree (no +)

- Idea:
  - Avoid duplicate keys
  - Have record pointers in non-leaf nodes

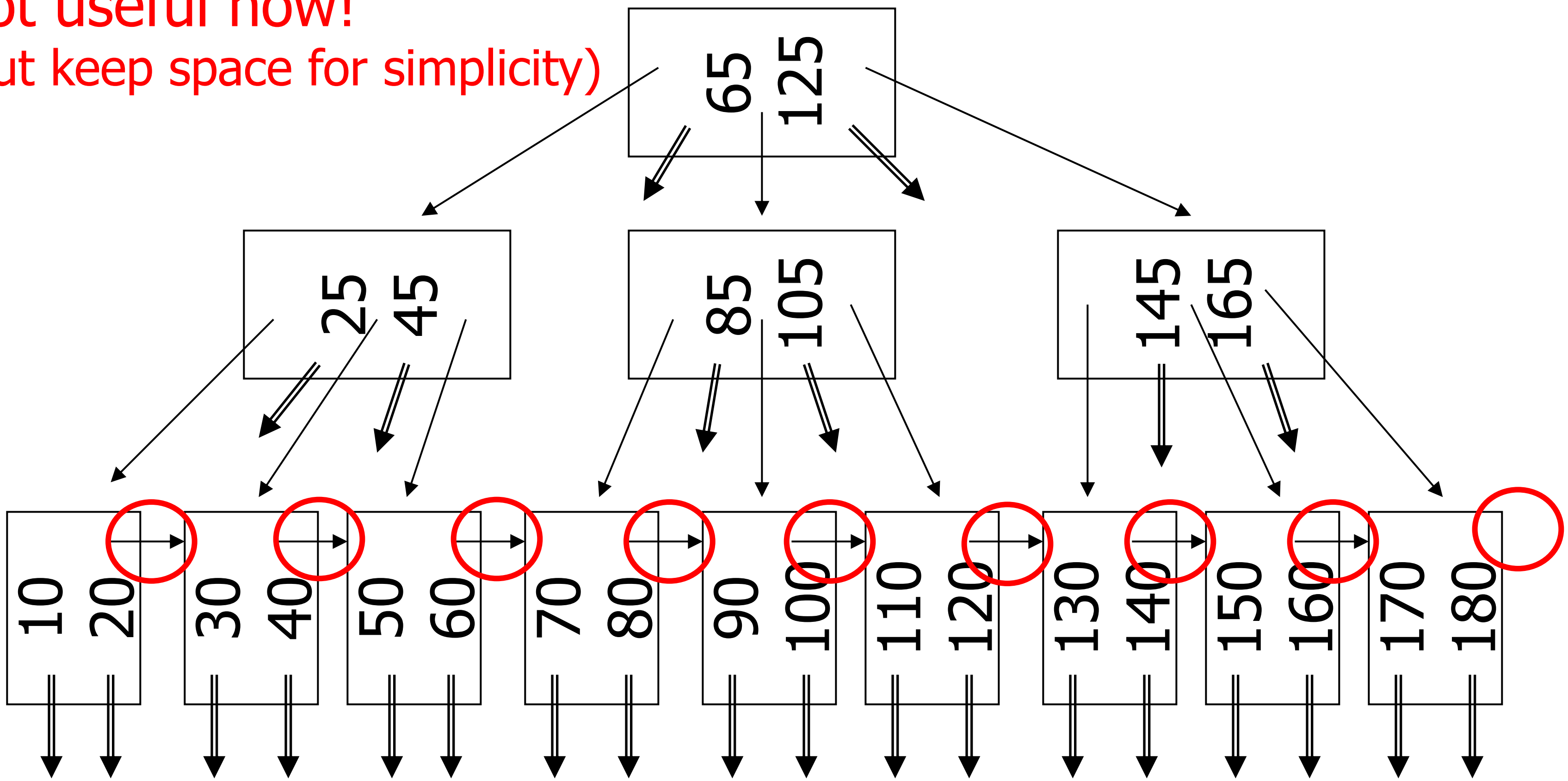

K1 P1
K2 P2
K3 P3
to record
with K1
to record
with K2
to record
with K3
to keys
< K1
to keys
K1<x<K2
to keys
K2<x<k3
to keys
>k3

B-tree example n=2



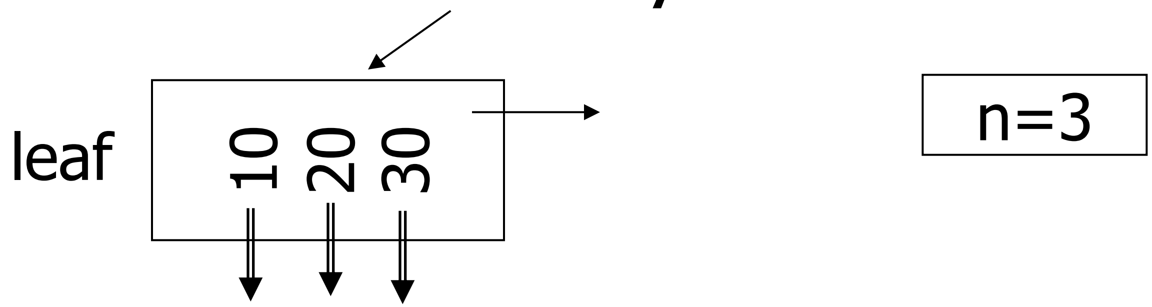
B-tree example n=2

- sequence pointers
  not useful now!
  (but keep space for simplicity)



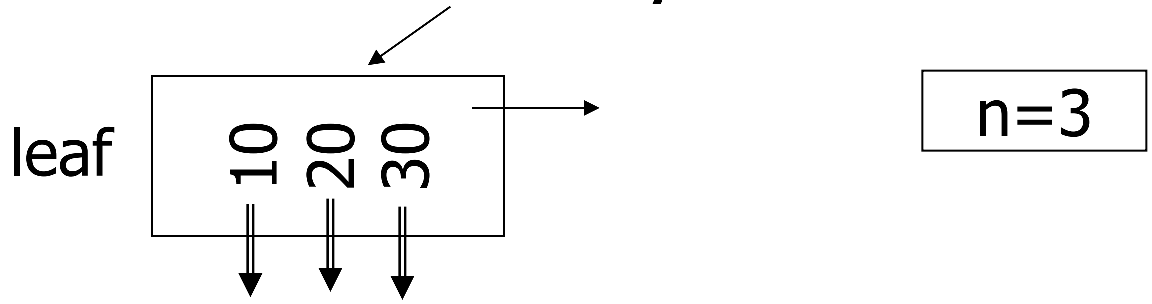
Note on inserts

- Say we insert record with key = 25

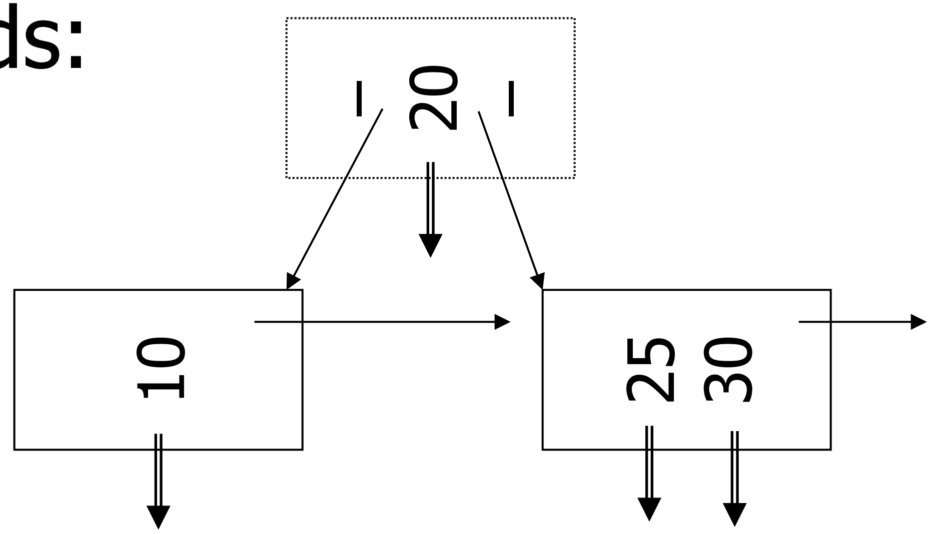


Note on inserts

- Say we insert record with key = 25



- Afterwards:



So, for B-trees:

	MAX			MIN		
	Tree Ptrs	Rec Ptrs	Keys	Tree Ptrs	Rec Ptrs	Keys
Non-leaf non-root	n+1	n	n	$\lceil (n+1)/2 \rceil$	$\lceil (n+1)/2 \rceil$-1	$\lceil (n+1)/2 \rceil$-1
Leaf non-root	1	n	n	1	$\lfloor n/2 \rfloor$	$\lfloor n/2 \rfloor$
Root non-leaf	n+1	n	n	2	1	1
Root Leaf	1	n	n	1	1	1

Tradeoffs:

☺ B-trees have faster lookup than B+trees

☹ in B-tree, non-leaf & leaf different sizes

☹ in B-tree, deletion more complicated

Tradeoffs:

☺ B-trees have faster lookup than B+trees

☹ in B-tree, non-leaf & leaf different sizes

☹ in B-tree, deletion more complicated

➔ B+trees preferred!

But note:

- If blocks are fixed size

  (due to disk and buffering restrictions)

  Then lookup for B+tree is
  actually better!!

Outline/summary

- Conventional Indexes
    - Sparse vs. dense
    - Primary vs. secondary
- B trees
    - B+trees vs. B-trees
    - B+trees vs. indexed sequential
- Hashing schemes --> Next