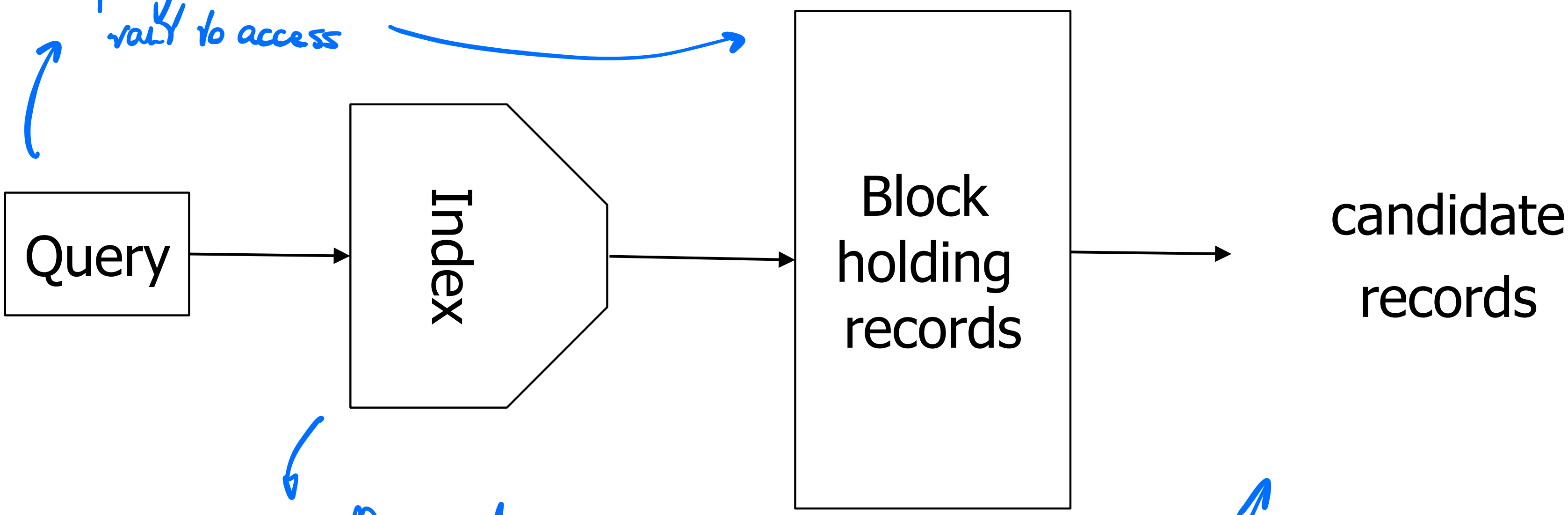
Indexing

Hector Garcia-Molina
Mahmoud Sakr

Indexing

contains data

we have a user query that want to access



an other way to access the block

it's organizing the data of the block and quickly searching for the records

returning the candidate records of the query

Topics

Building blocks!

- Conventional indexes
- B-trees
- Hashing schemes

when accessing records,
we can't access subpages,
always take whole pages

the way our data is
organized in the storage

Sequential File

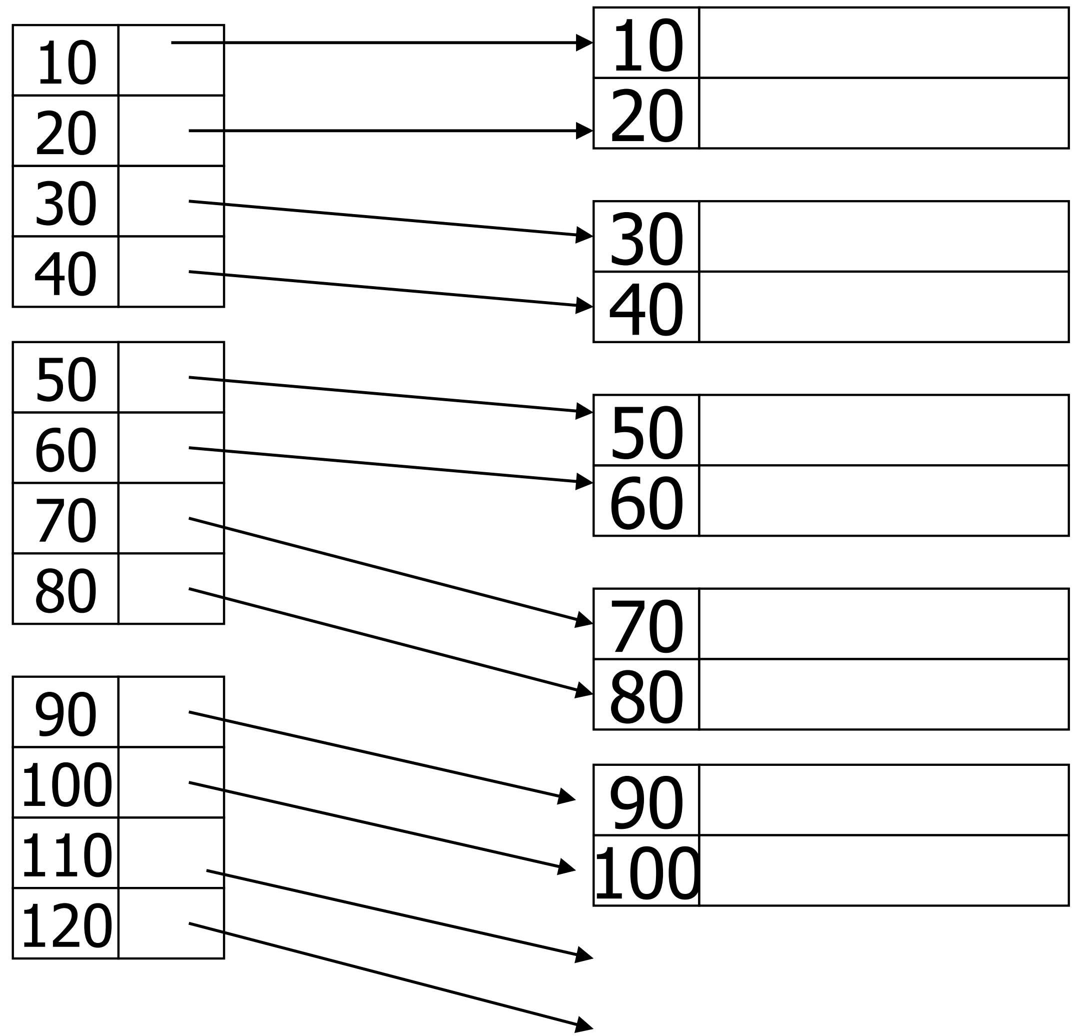
page ≈ 4 KB

10	
20	
30	
40	
50	
60	
70	
80	
90	
100	

sorted by the key

Dense Index

Sequential File



Dense Index = a pointer per key

sequential scan of the index

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

↳ we have a pointer that lead to the records

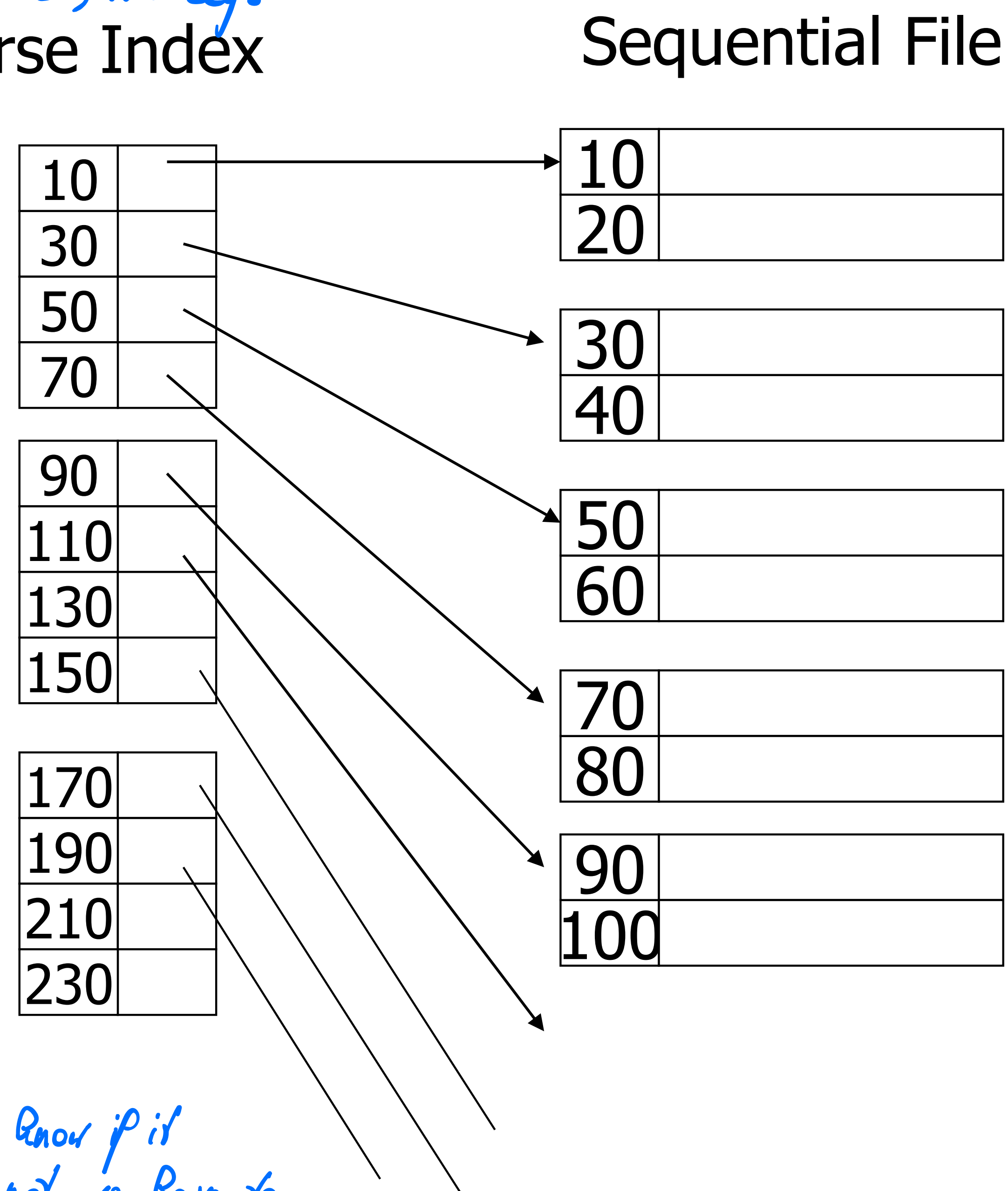
Why querying a dense index is more efficient than querying the sequential file ?

→ allow us to have a different order than the storage

⇒ we can sort the data in the index based on what we are looking for

↳ search through a sorted list is way faster → Binary search we are pointing blocks, not keys

Sparse Index



Sparse index = a pointer per block ↳ less index entries

How to search for a key= 30 ?

How to search for a key= 25 ?

not in the index records ⇒ we look for it in the previous page

↳ we do not know if it is there or not, we have to go through the records of the page sequentially

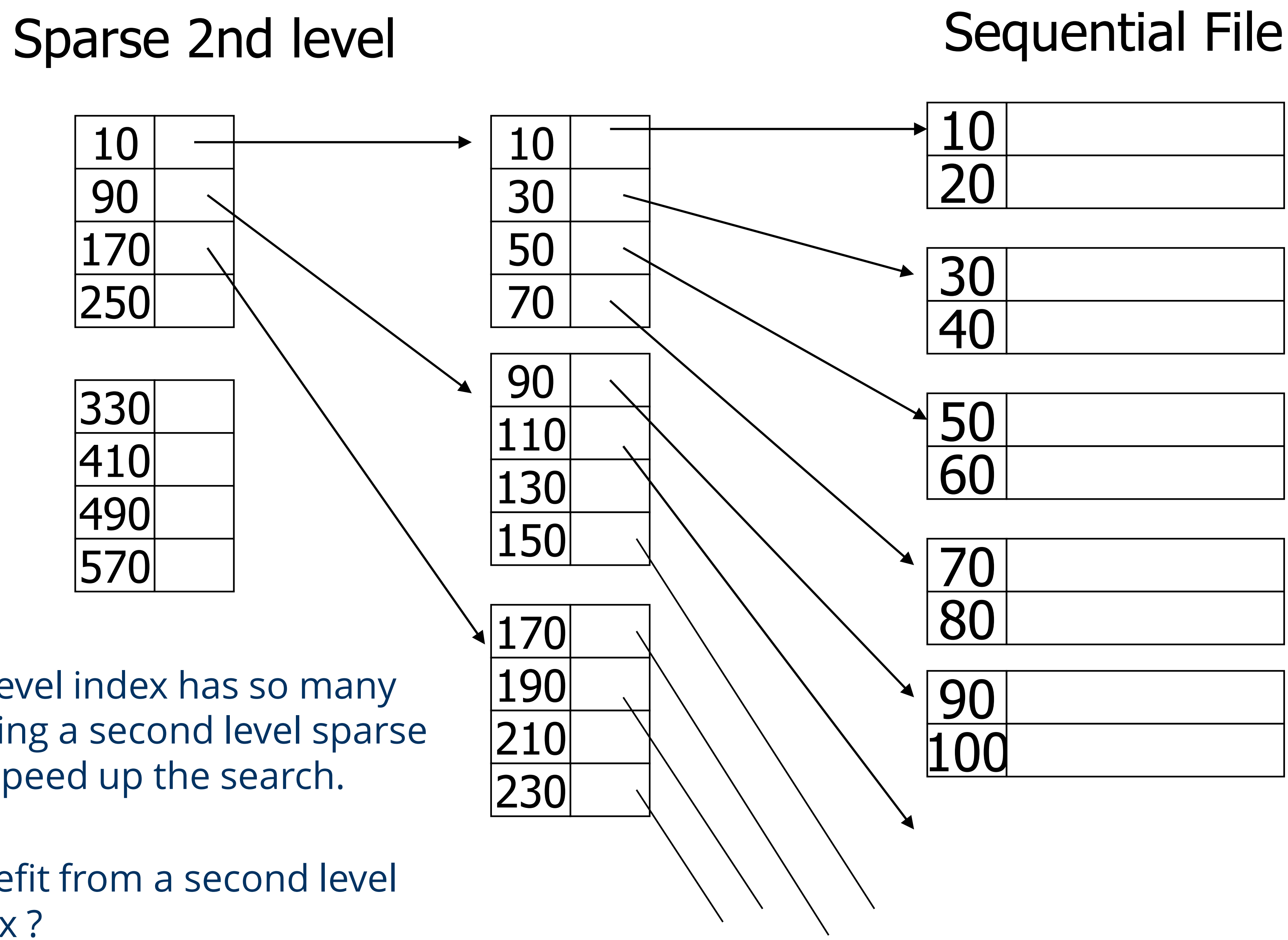
Can we use a sparse index on a non-sequential file ? No

↳ for the sparse index to work, we need to be sure that the pages are sorted

↓

if we are looking for 20 it must be between 10 and 30

⇒ sorted sequential file on the same key as the index



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, we keep the same amount of entries ⇒ useless

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

but bigger size, we could be unable
to store the whole index in the memory

reduce number
of access
(mean cost
in the database)

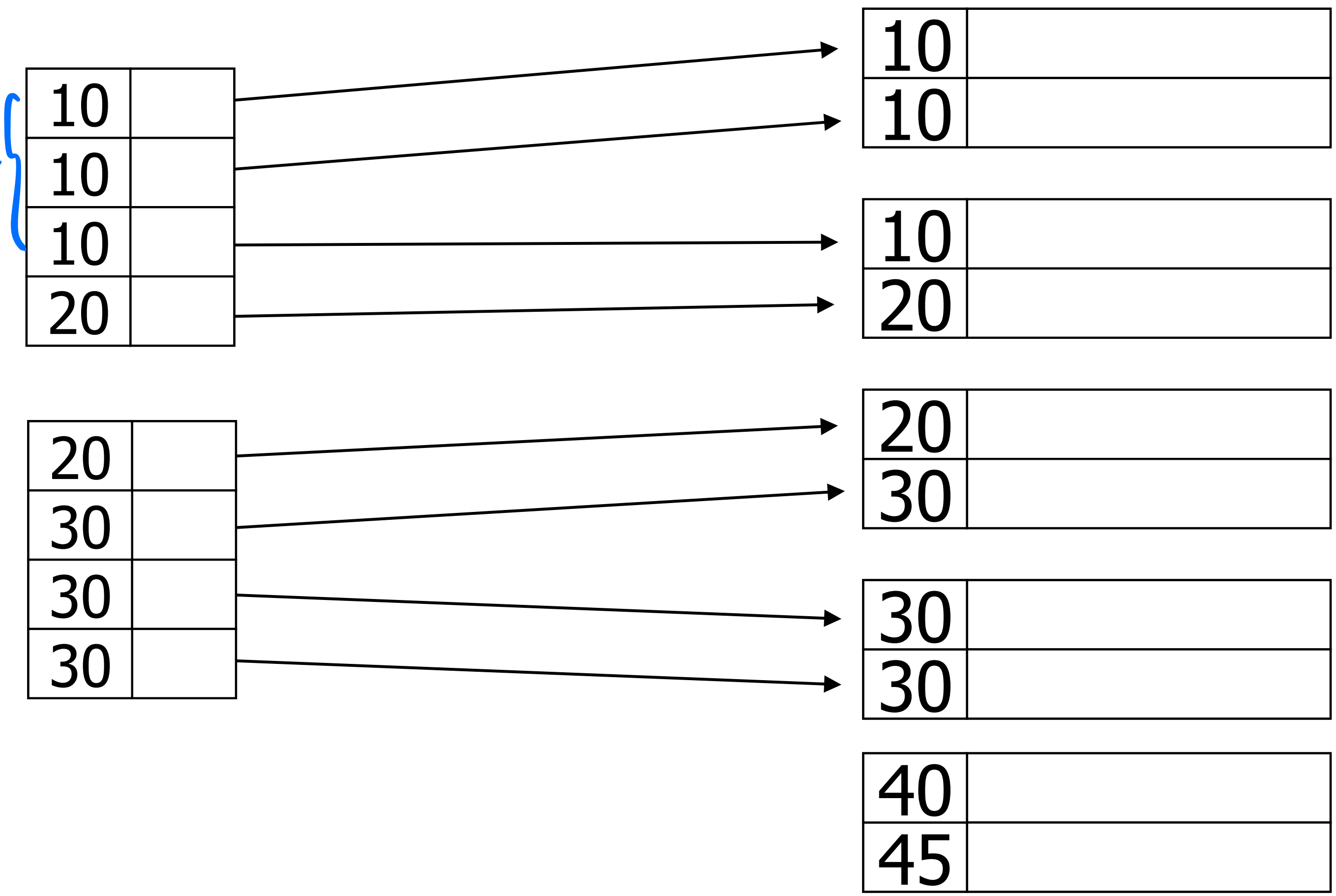
Duplicate keys

Dense index, one way to implement?

Dense index.

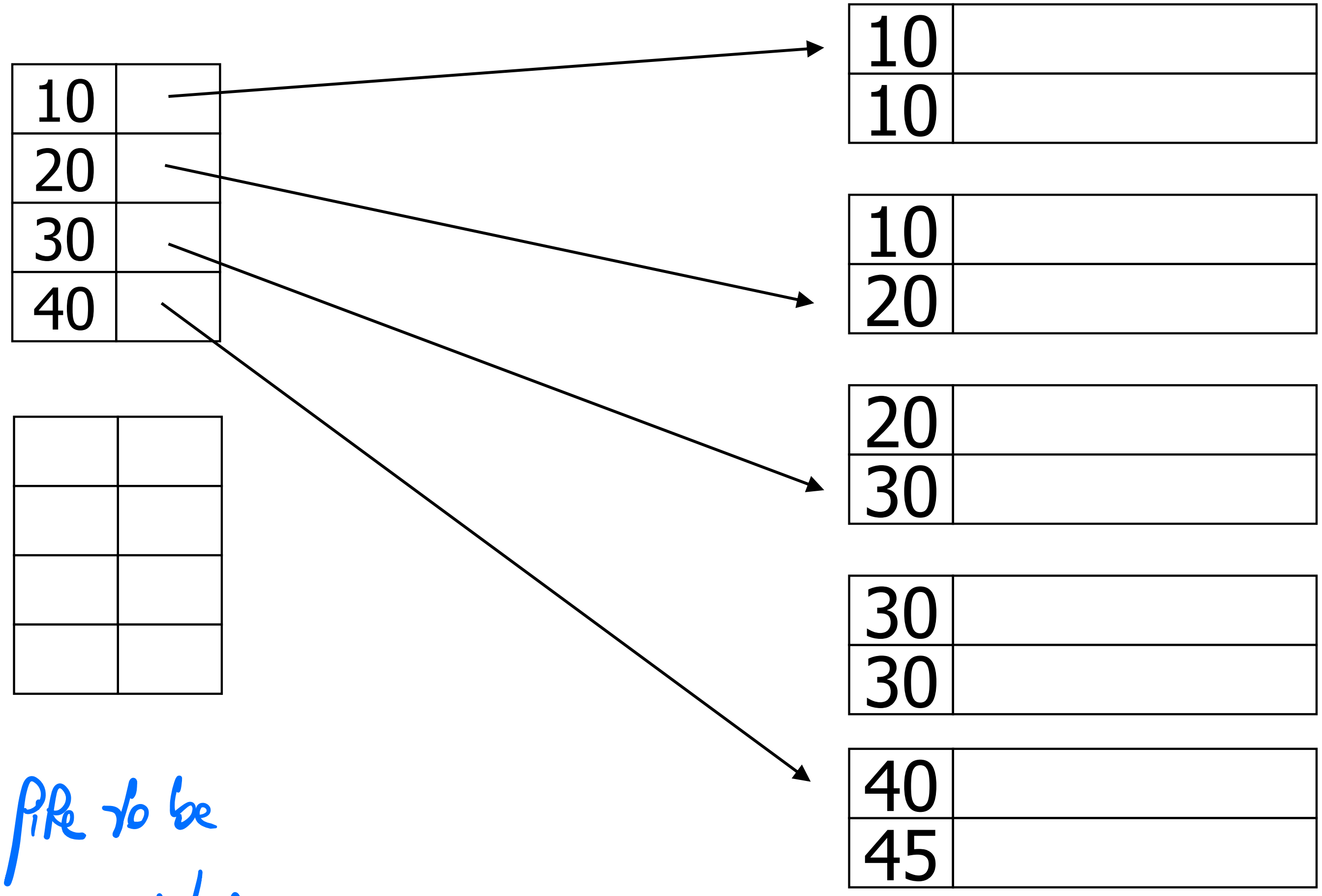
Can We do better ?

add to the side



Duplicate keys

Dense index, better way?



require the file to be sorted in the same order as the index

Duplicate keys

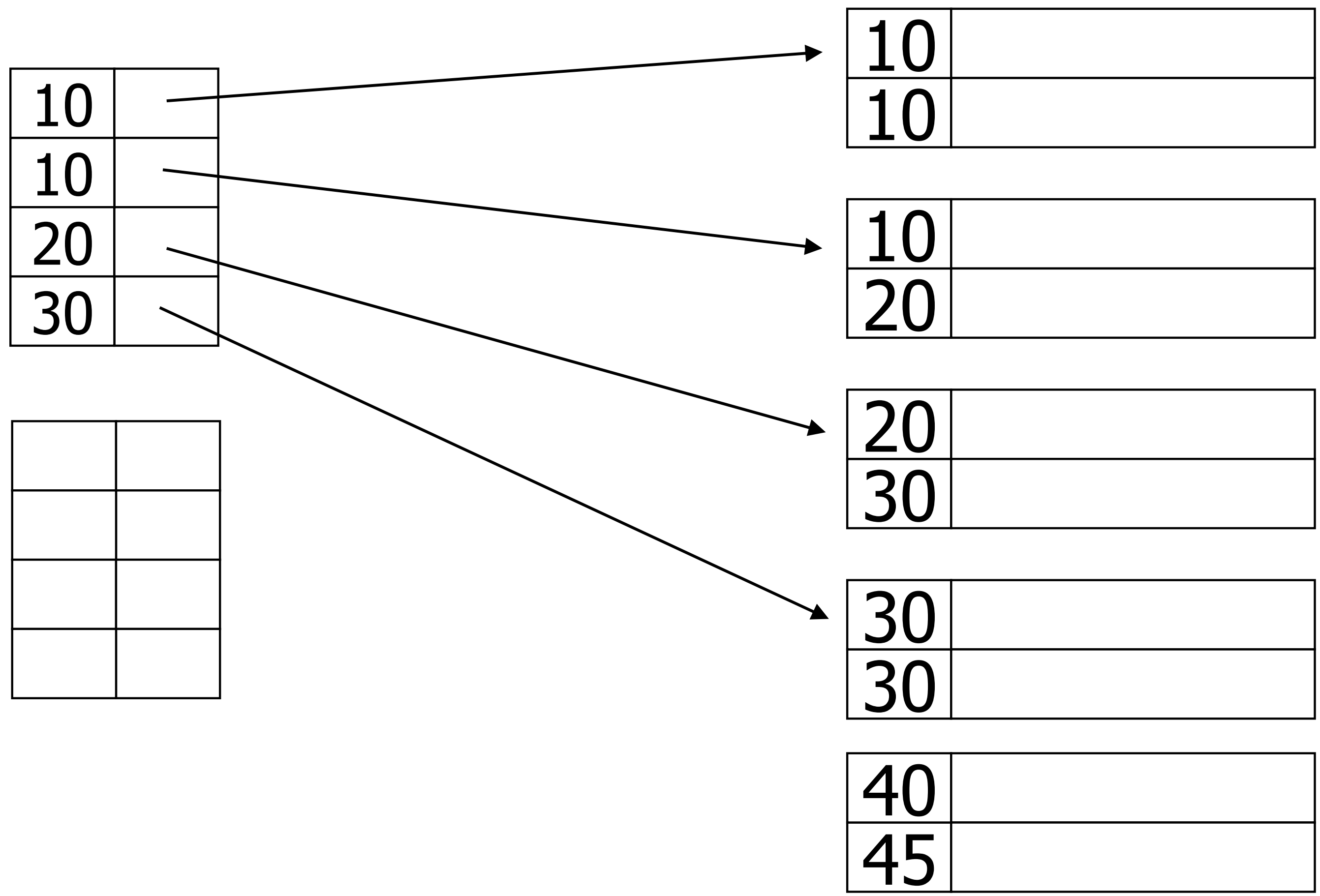
Sparse index, one way?

Sparse index - place key from block

How to search for 30 ?

1. binary search the index
2. find page of the record
3. scan backward and forward

since store by pages, can be same in the previous and the next page



Duplicate keys

Sparse index, another way?

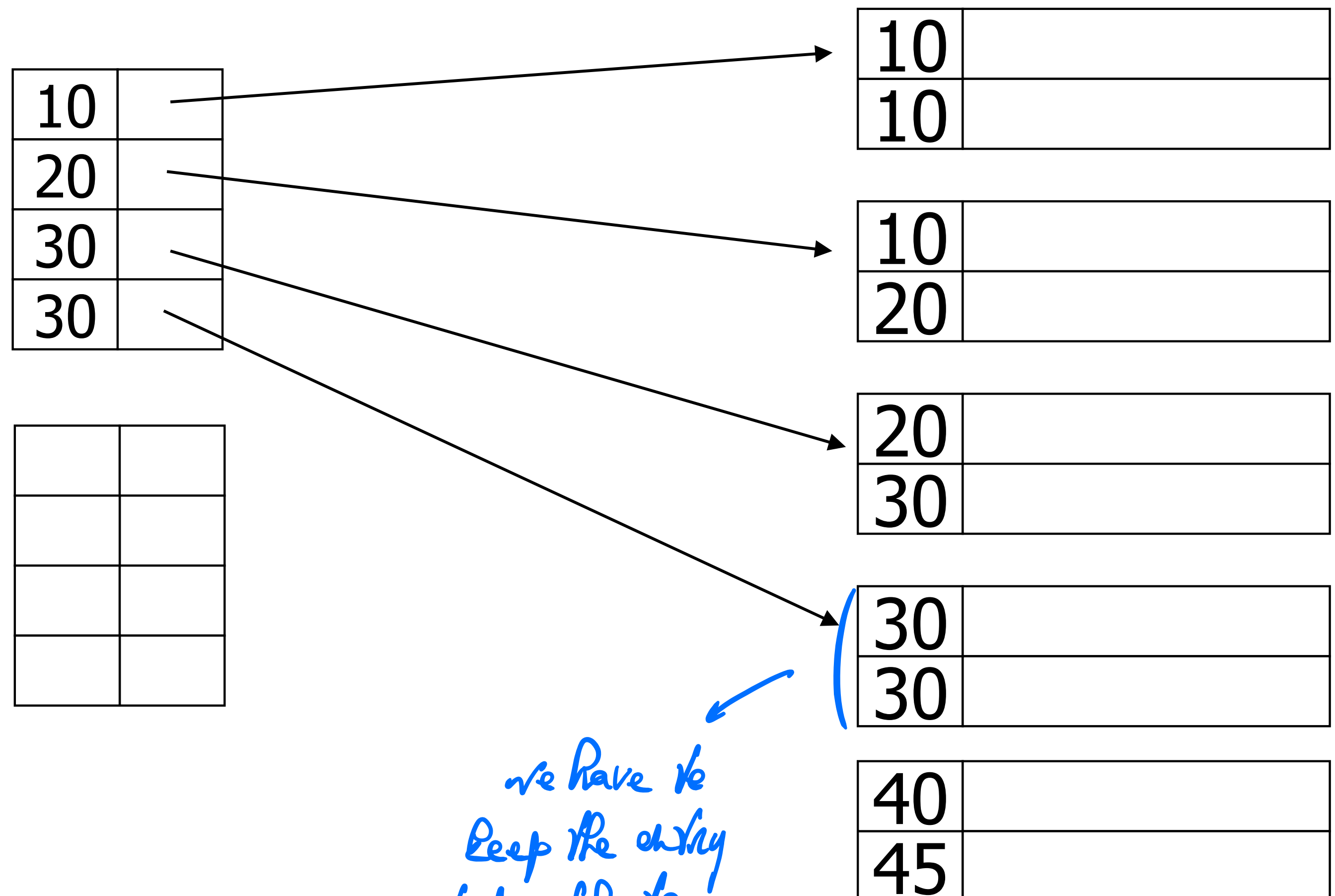
Sparse index - place first new key from block

How to search for 30 ?

↳ no need to scan back

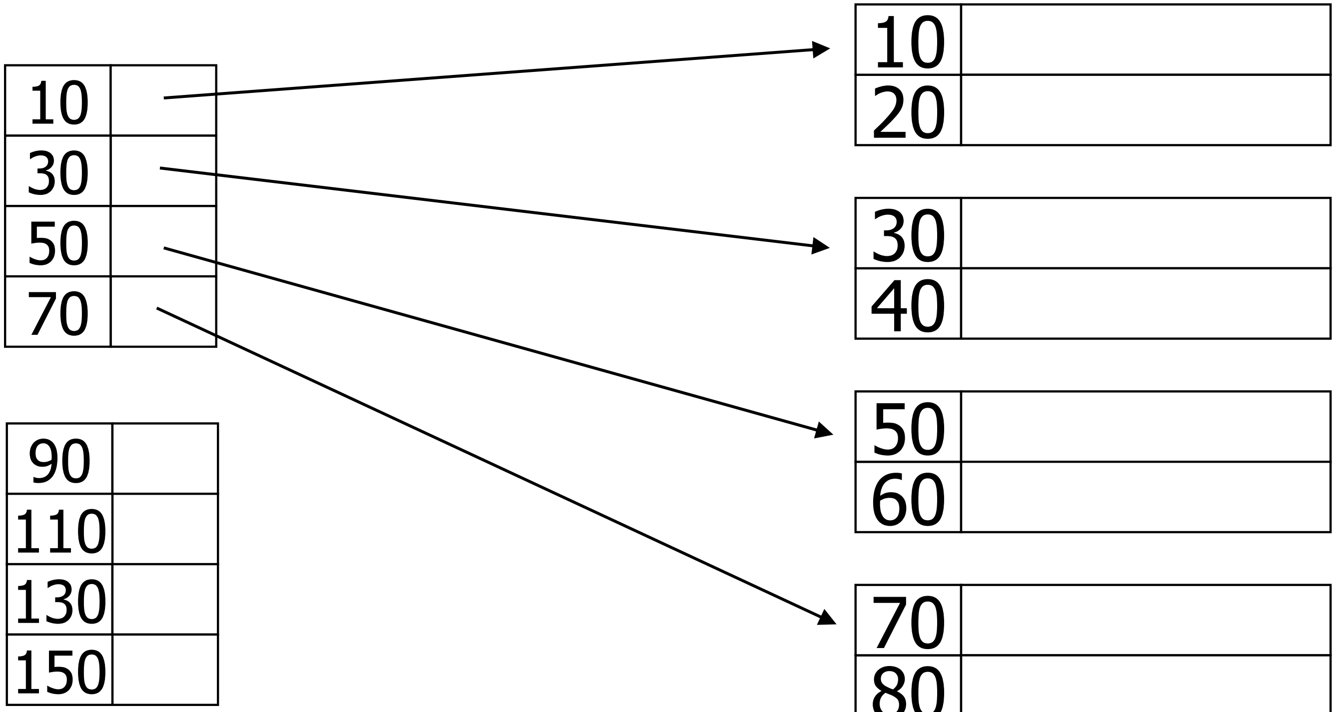
How to search for 35 ?

↳ go to the previous pointer and scan the page for the key



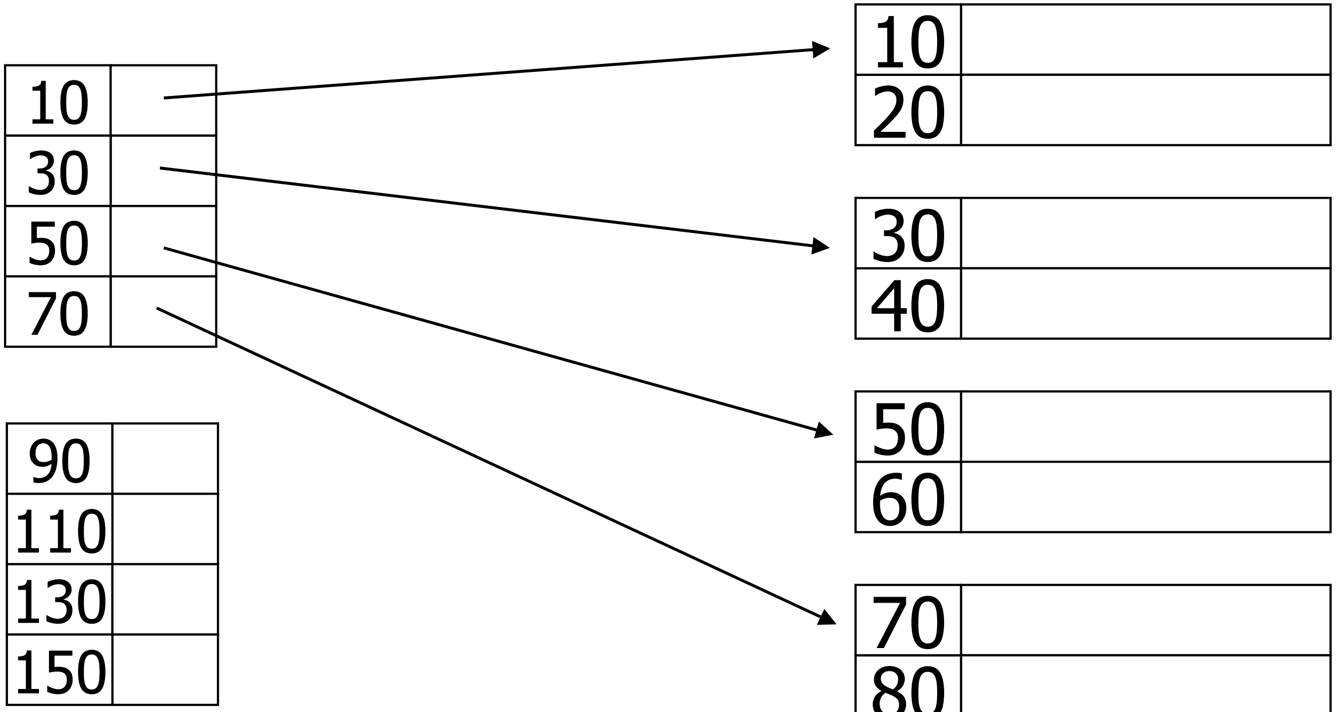
we have to keep the entry to be able to access it even though there is no new key

Deletion from sparse index



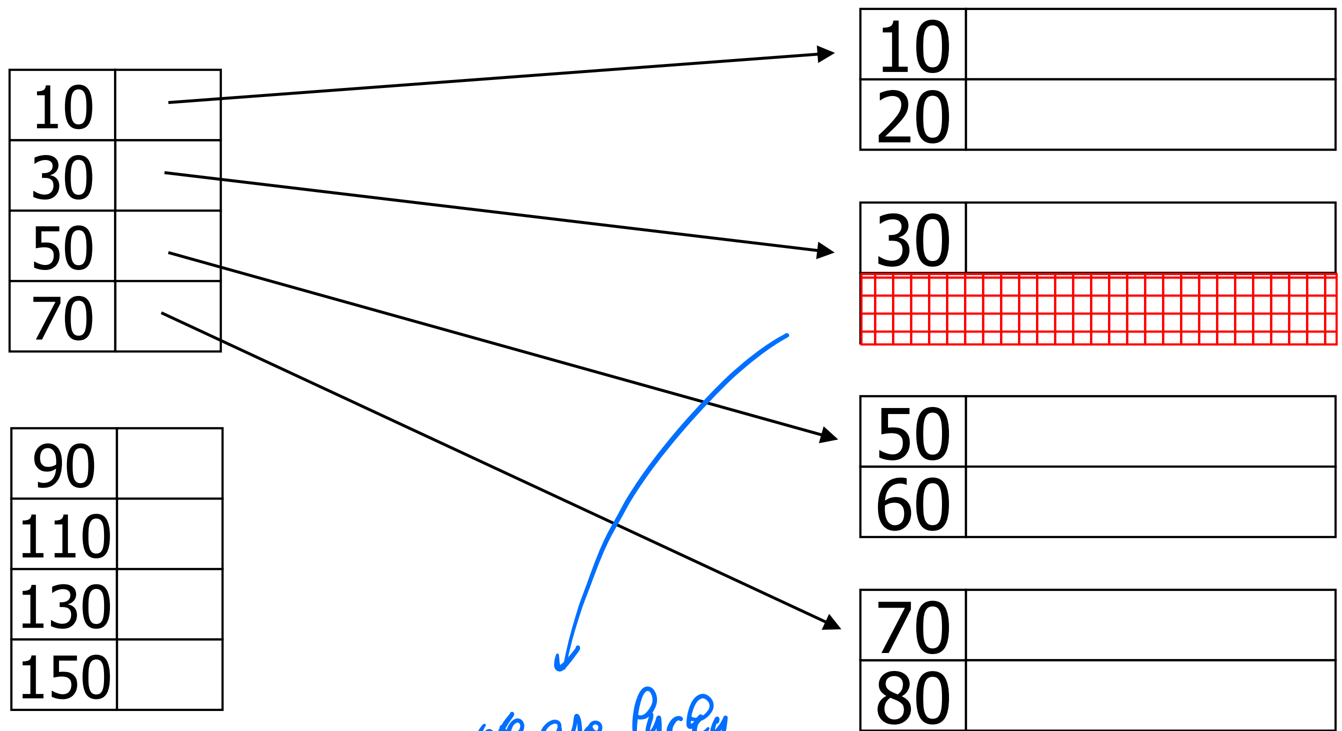
Deletion from sparse index

– delete record 40



Deletion from sparse index

– delete record 40

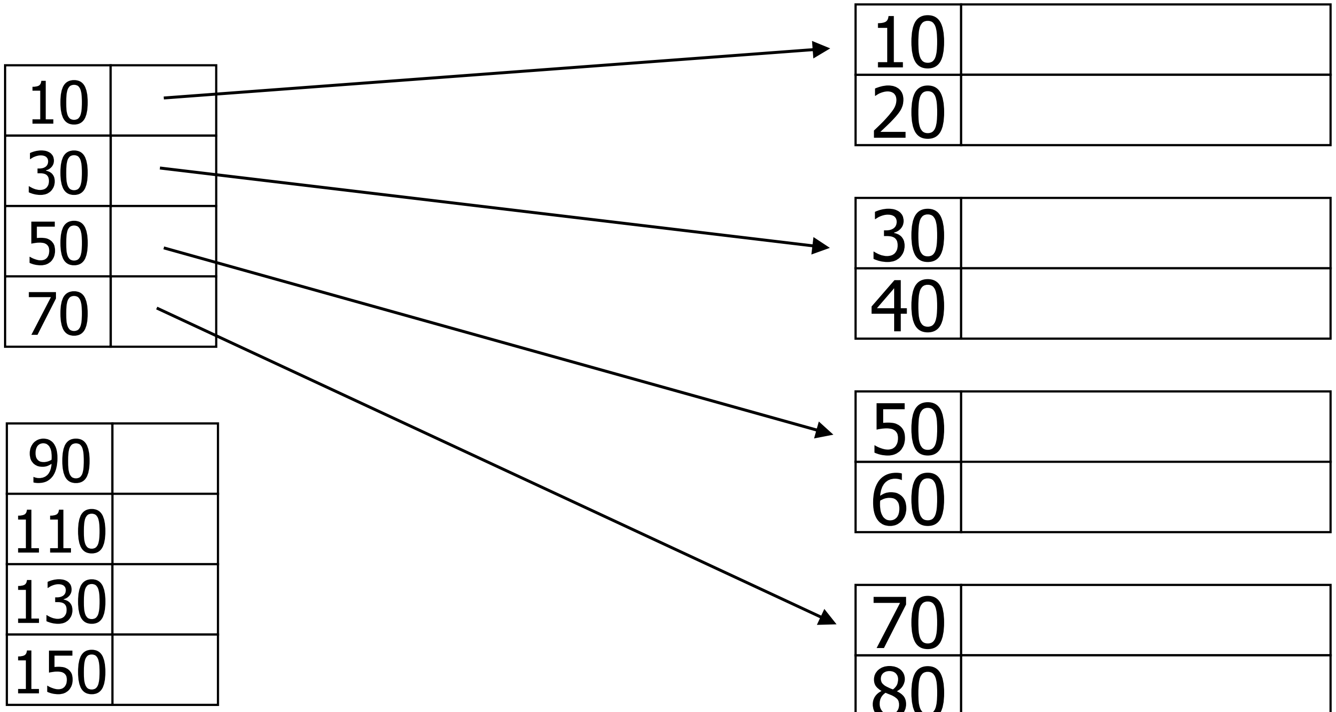


we are lucky
it was not a
key of the index
⇒ just suppress it

+ we could want to
keep the data sequential
in the file
(shifting up)

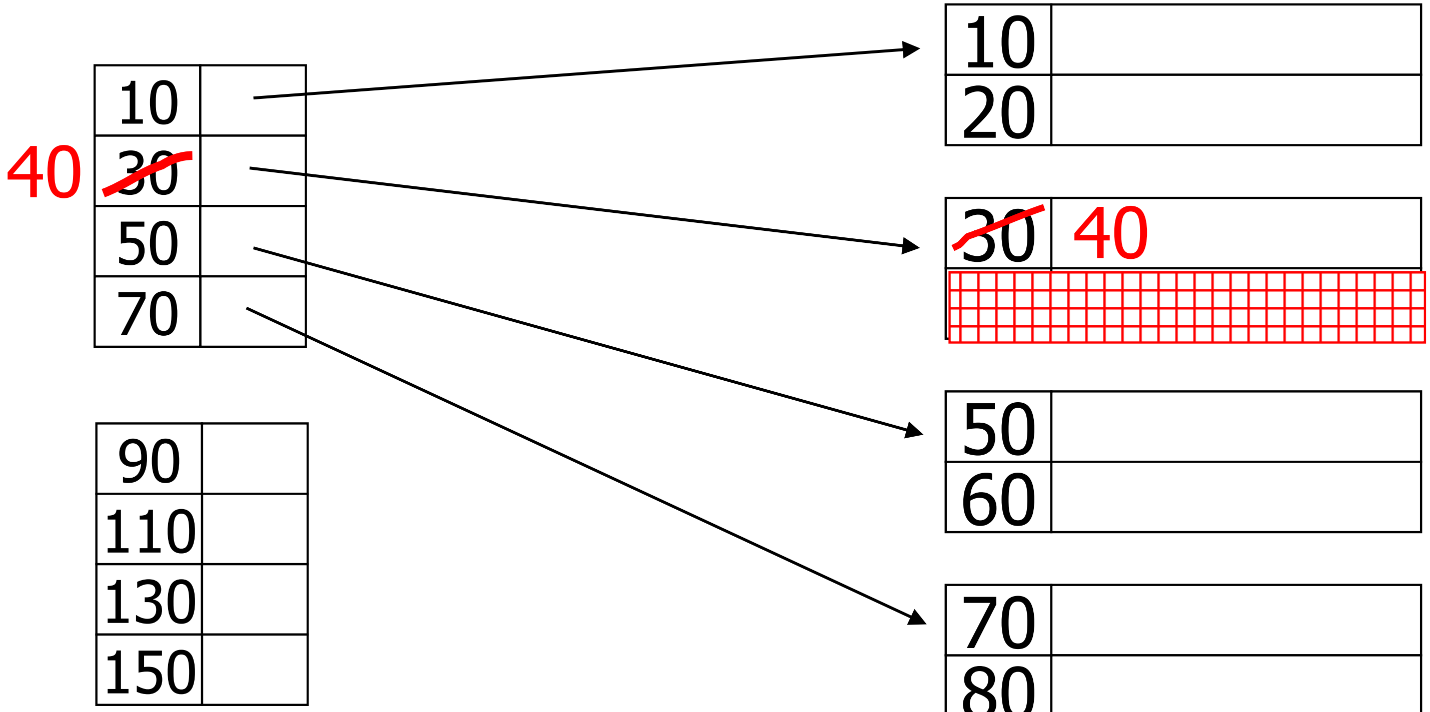
Deletion from sparse index

– delete record 30



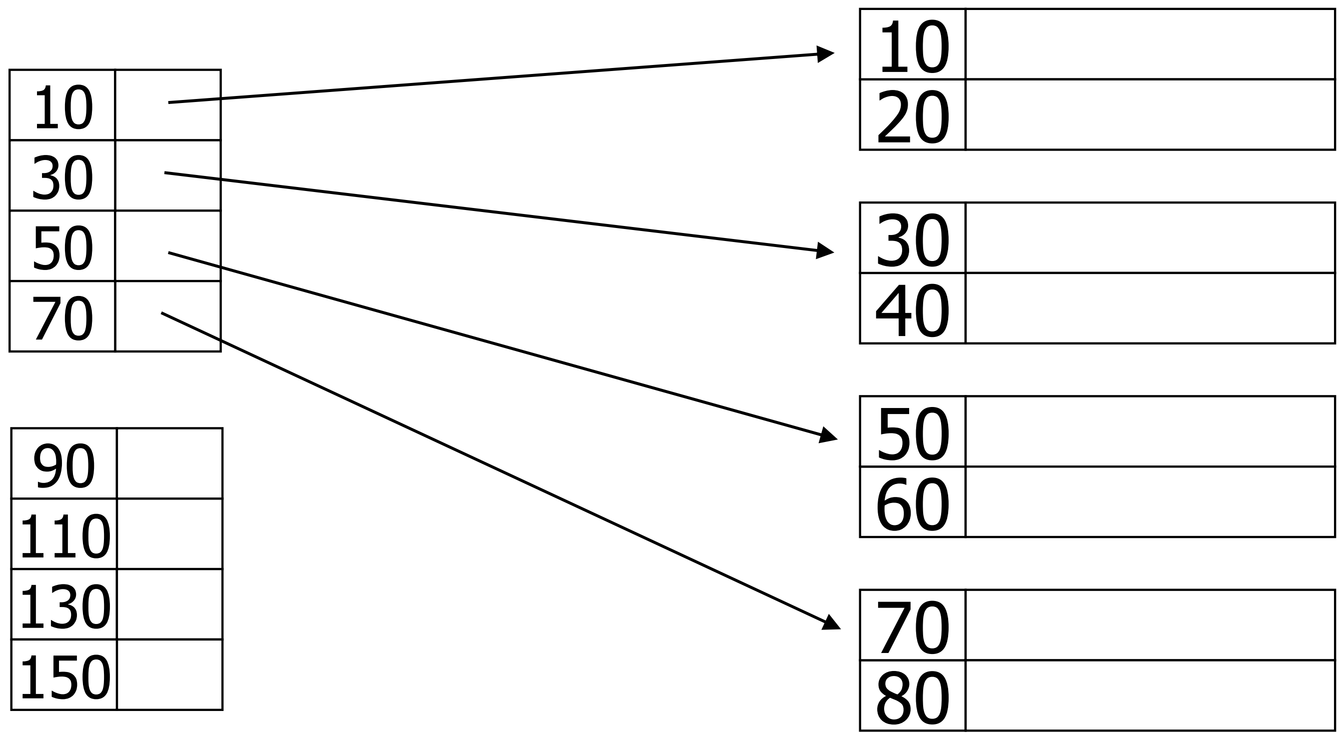
Deletion from sparse index

– delete record 30



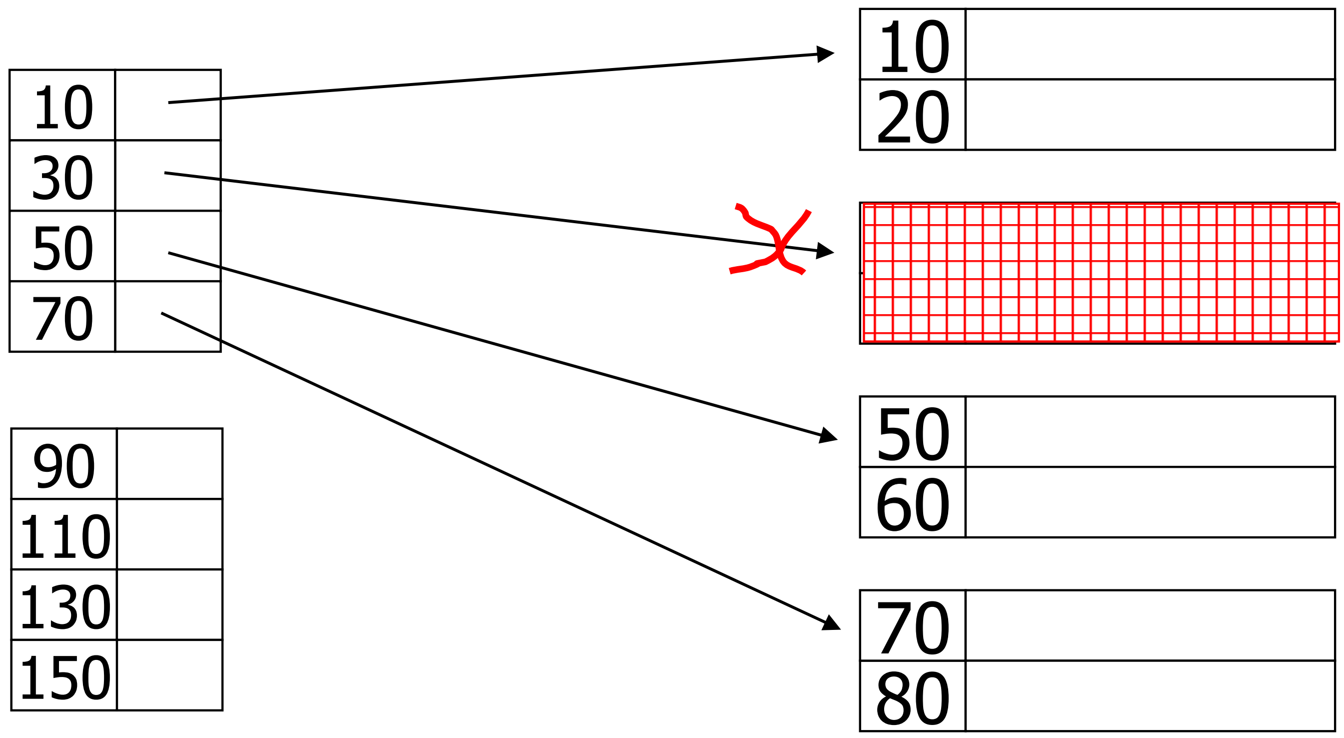
Deletion from sparse index

– delete records 30 & 40



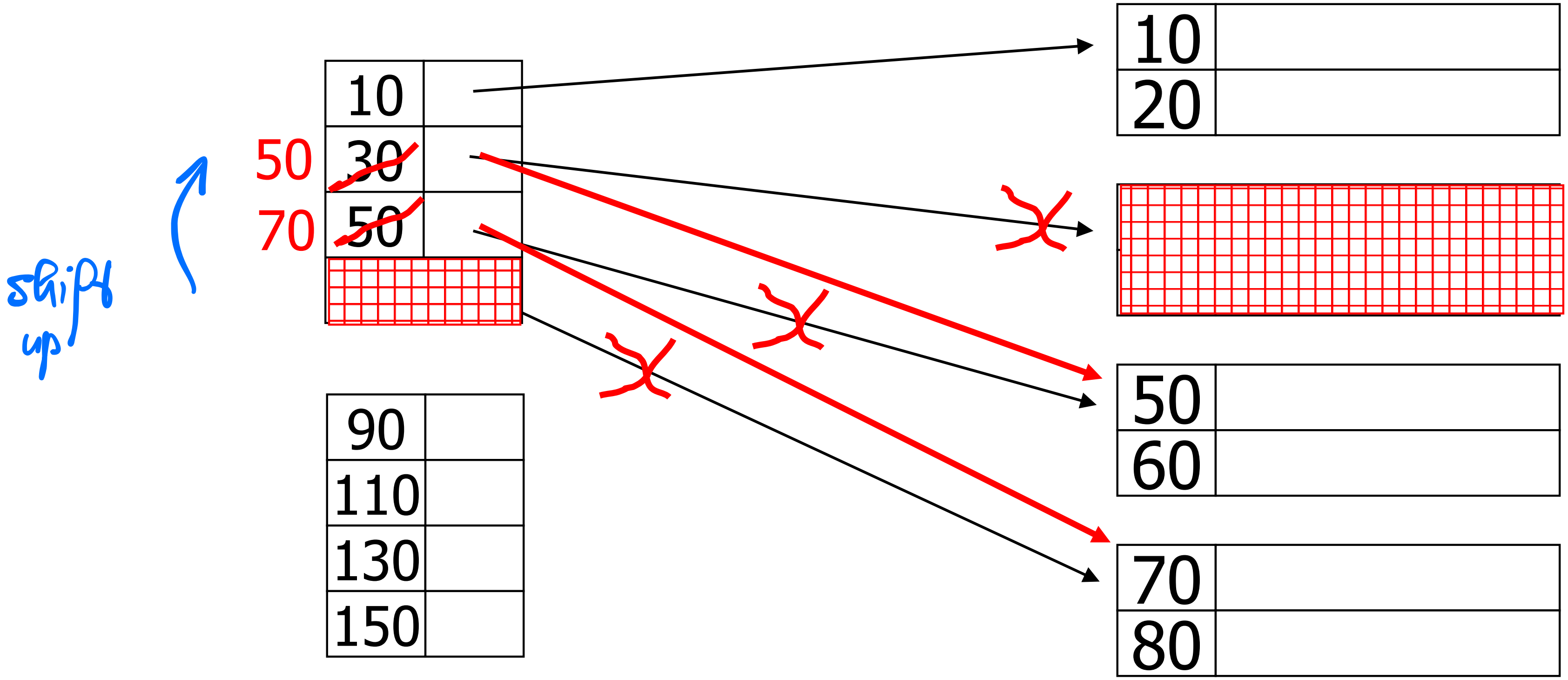
Deletion from sparse index

– delete records 30 & 40

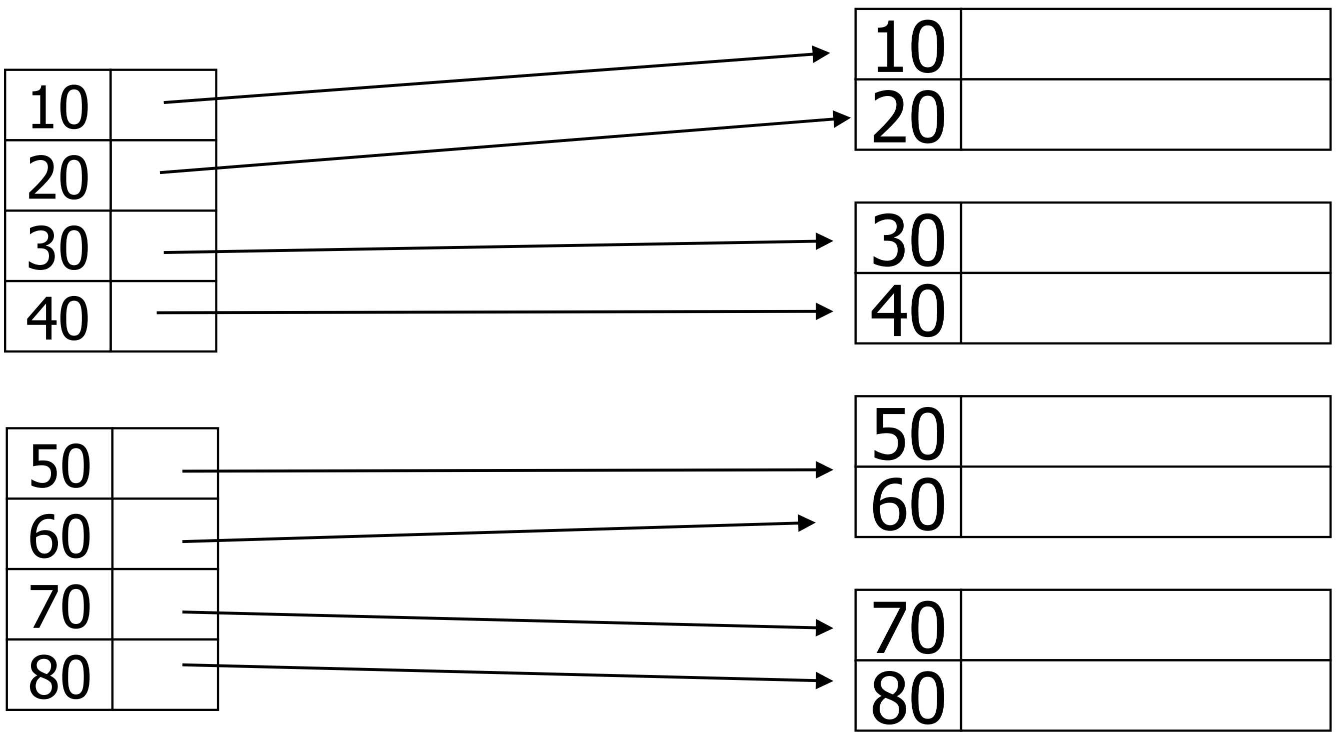


Deletion from sparse index

– delete records 30 & 40

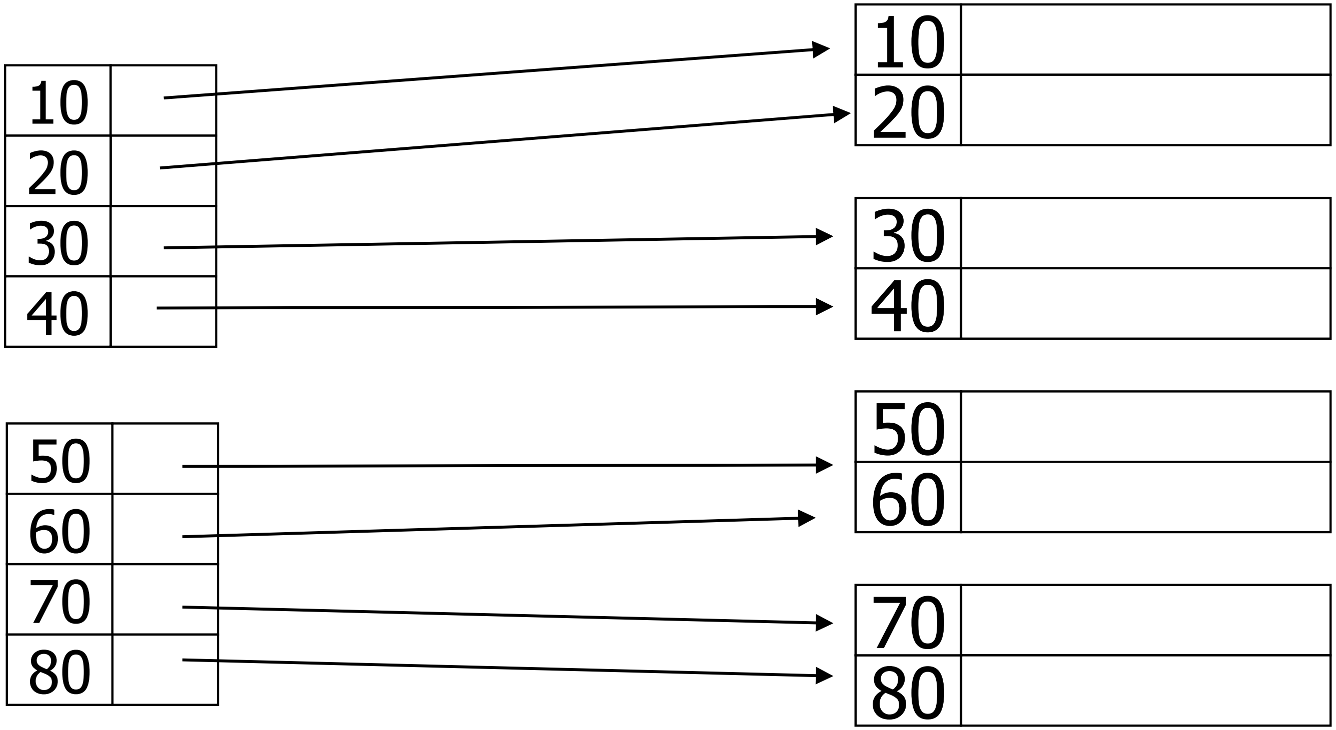


Deletion from dense index



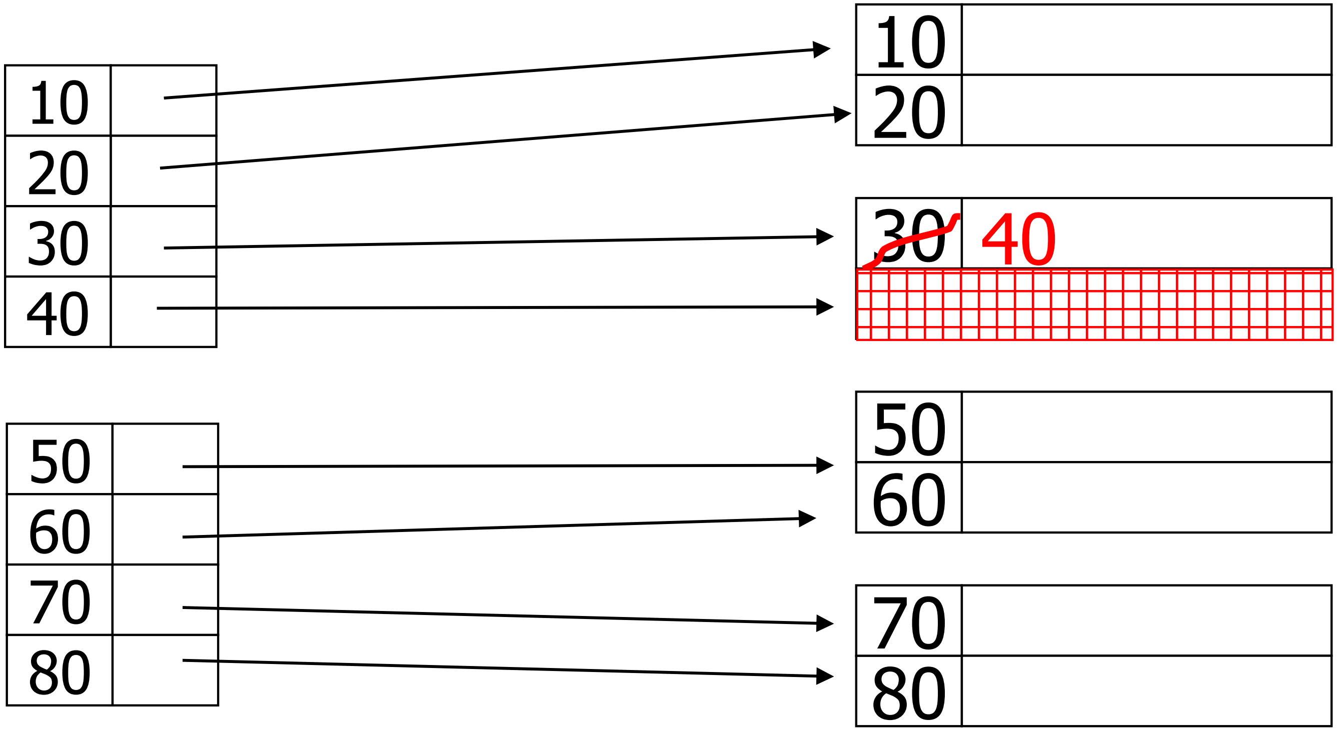
Deletion from dense index

– delete record 30



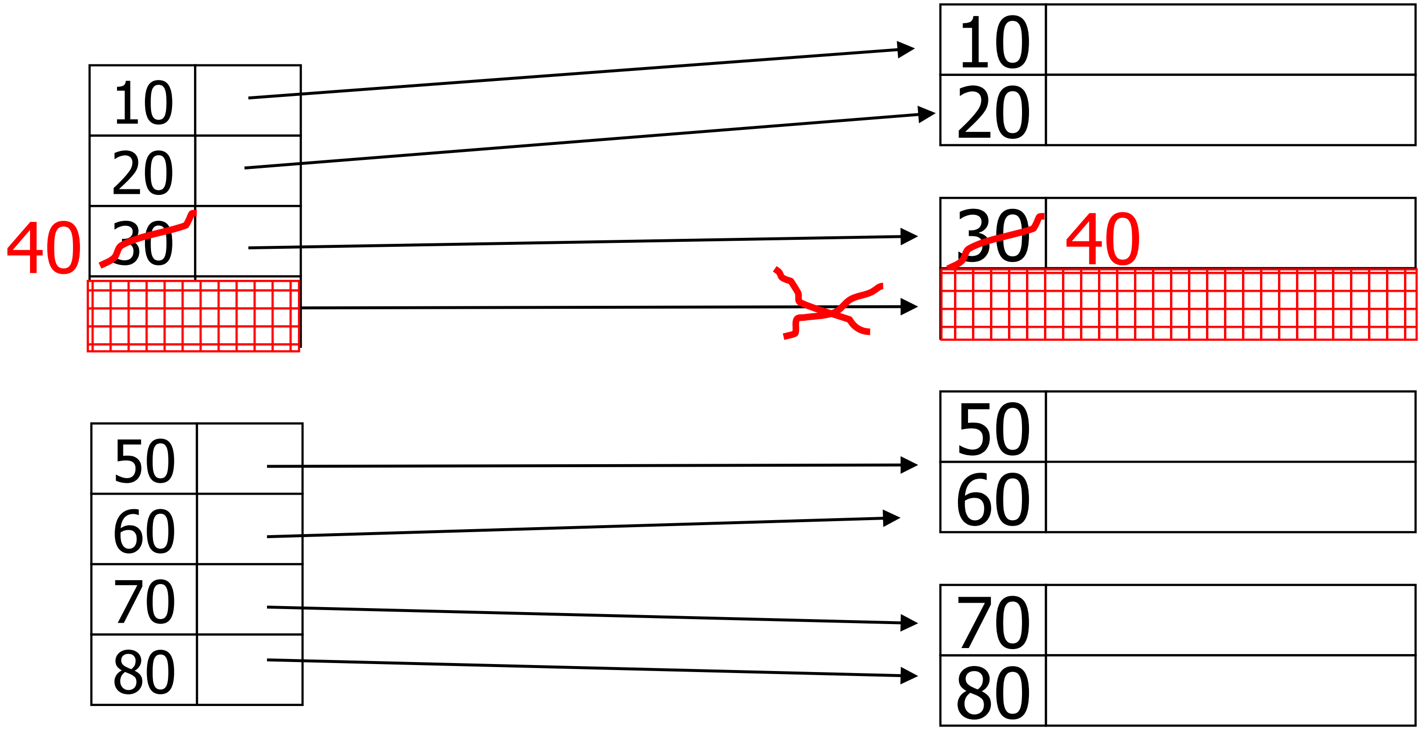
Deletion from dense index

– delete record 30

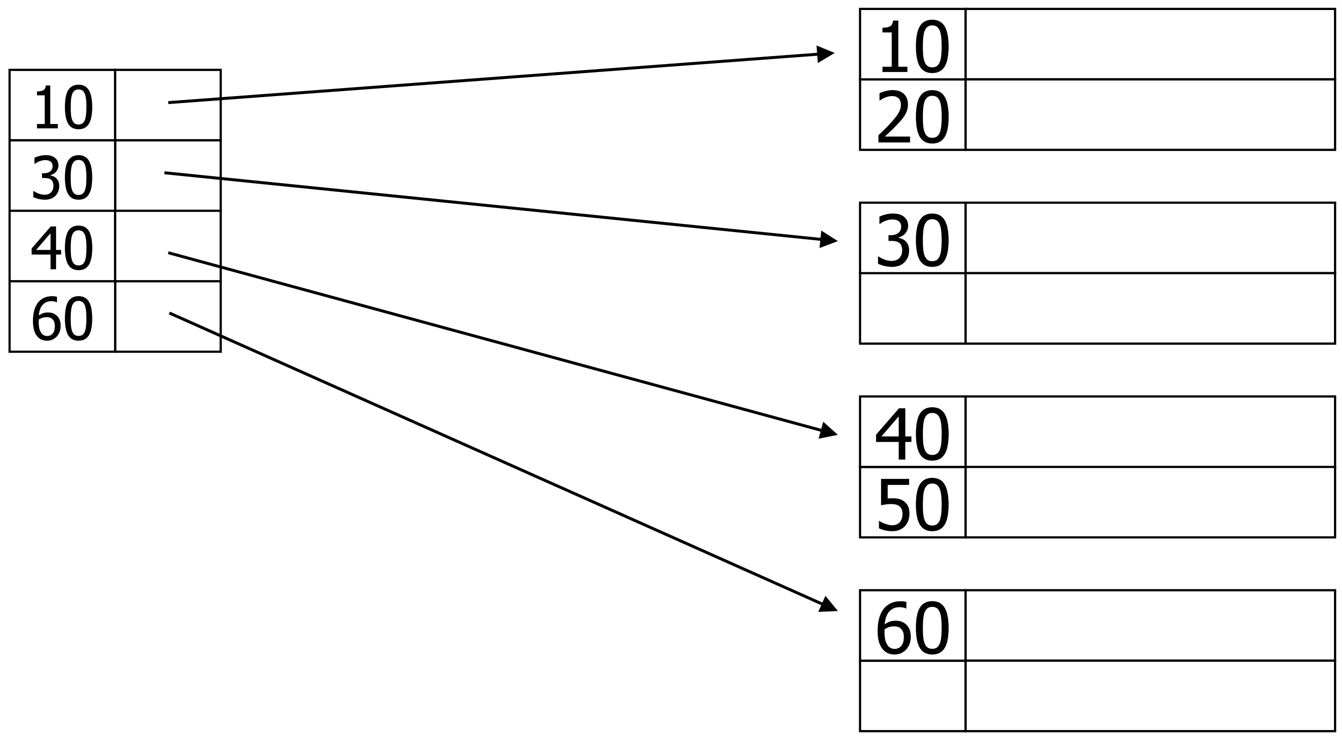


Deletion from dense index

– delete record 30

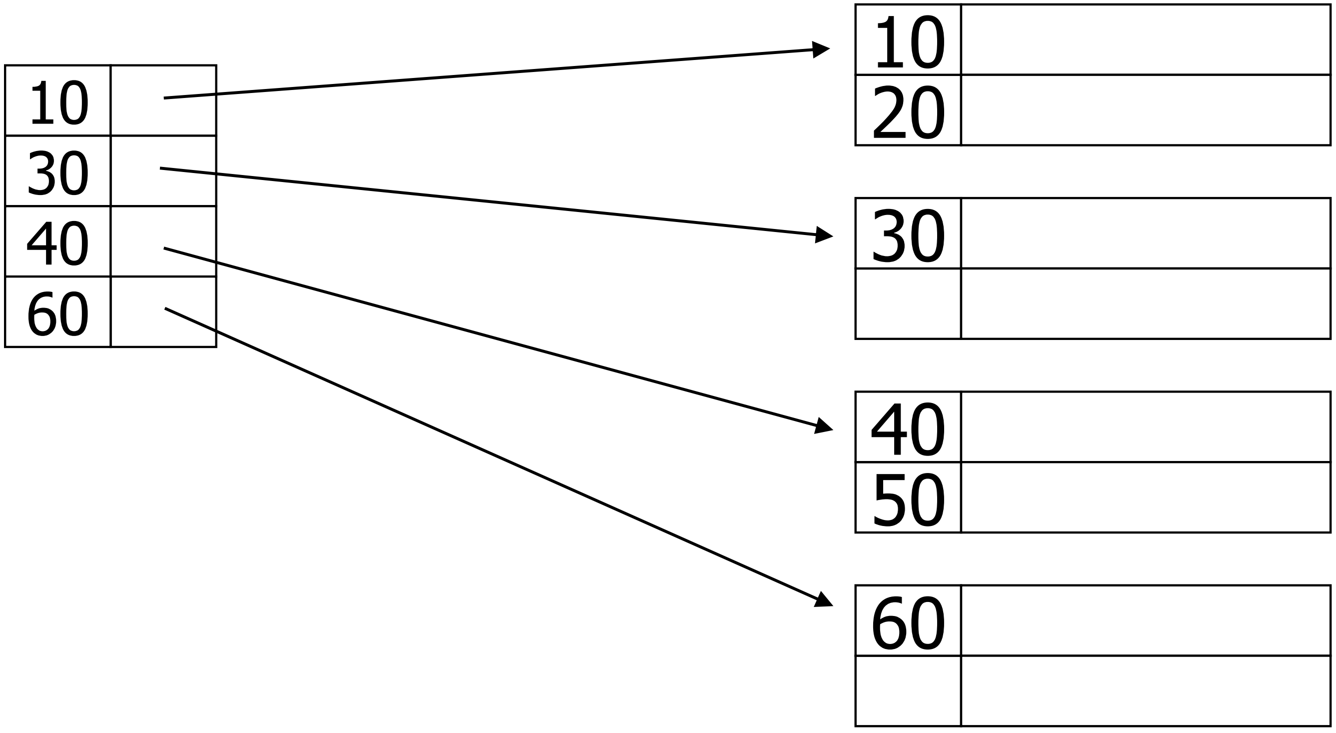


Insertion, sparse index case



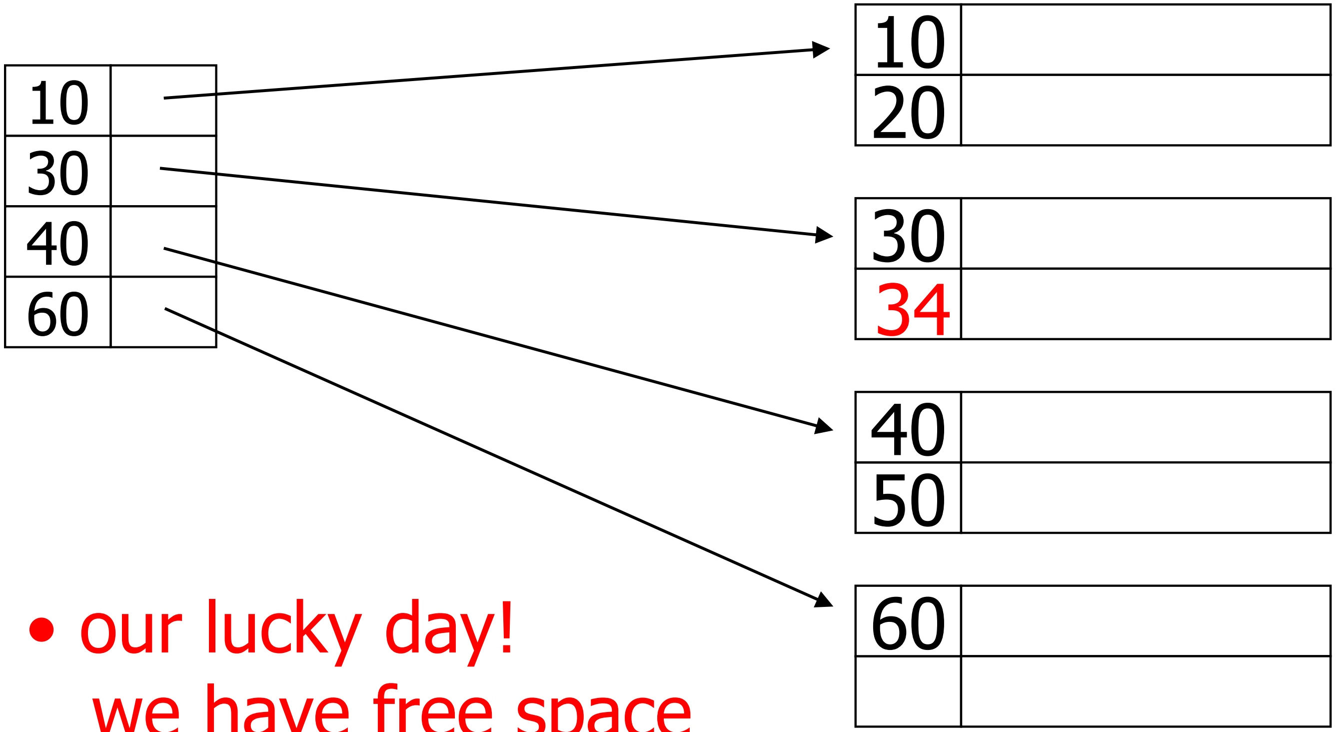
Insertion, sparse index case

– insert record 34



Insertion, sparse index case

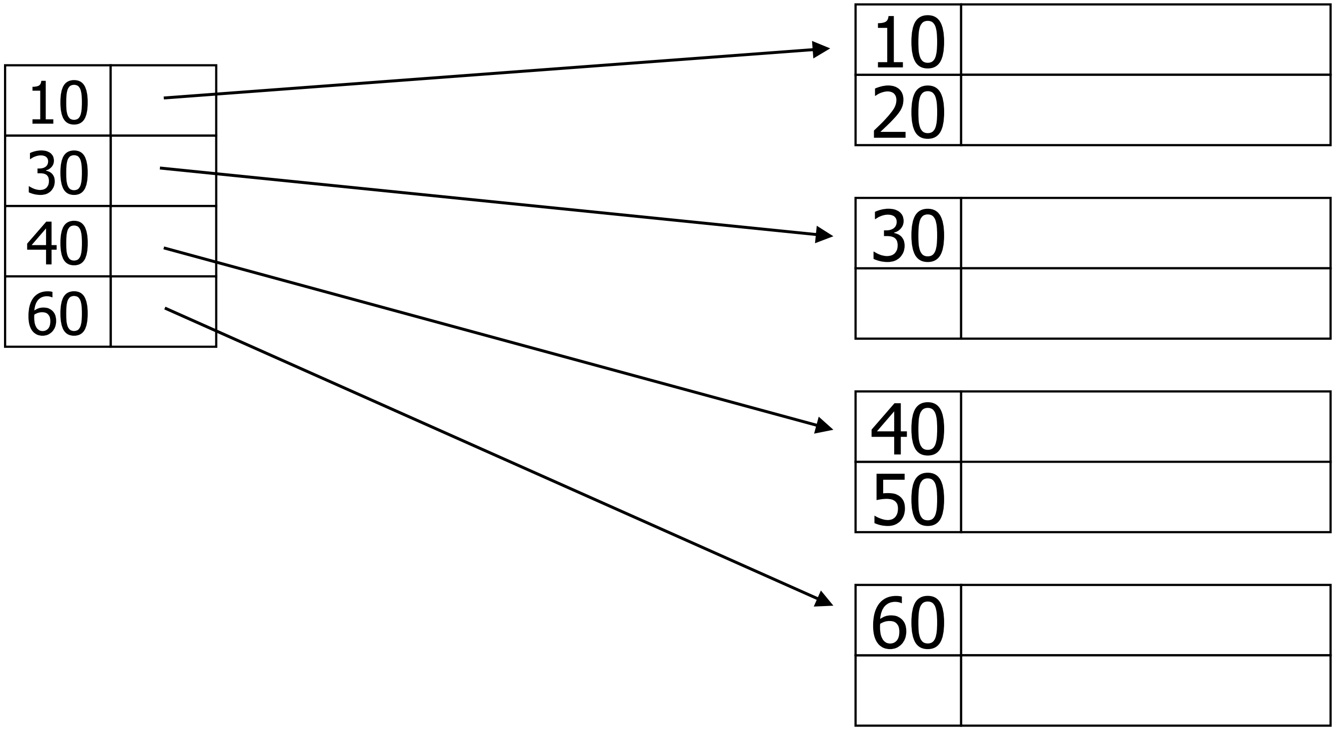
– insert record 34



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

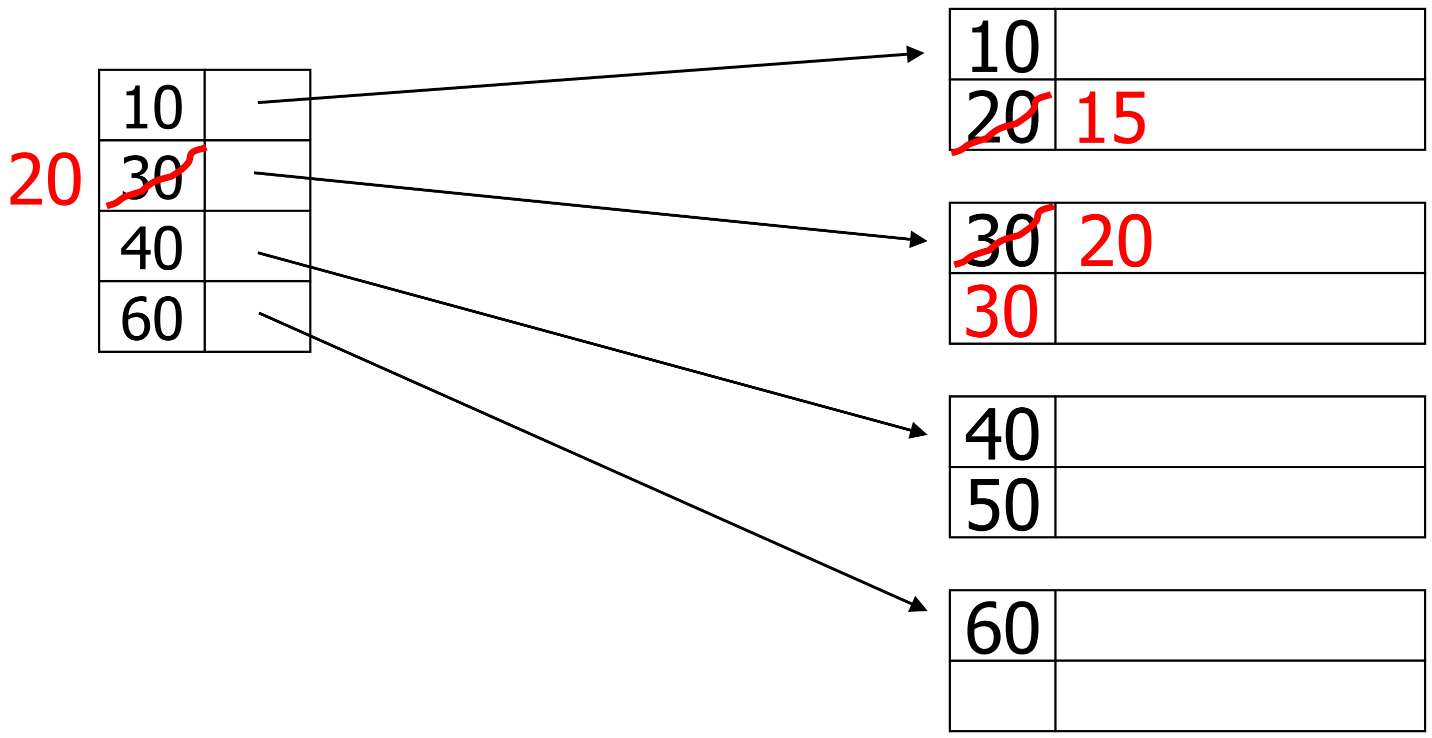
Insertion, sparse index case

– insert record 15



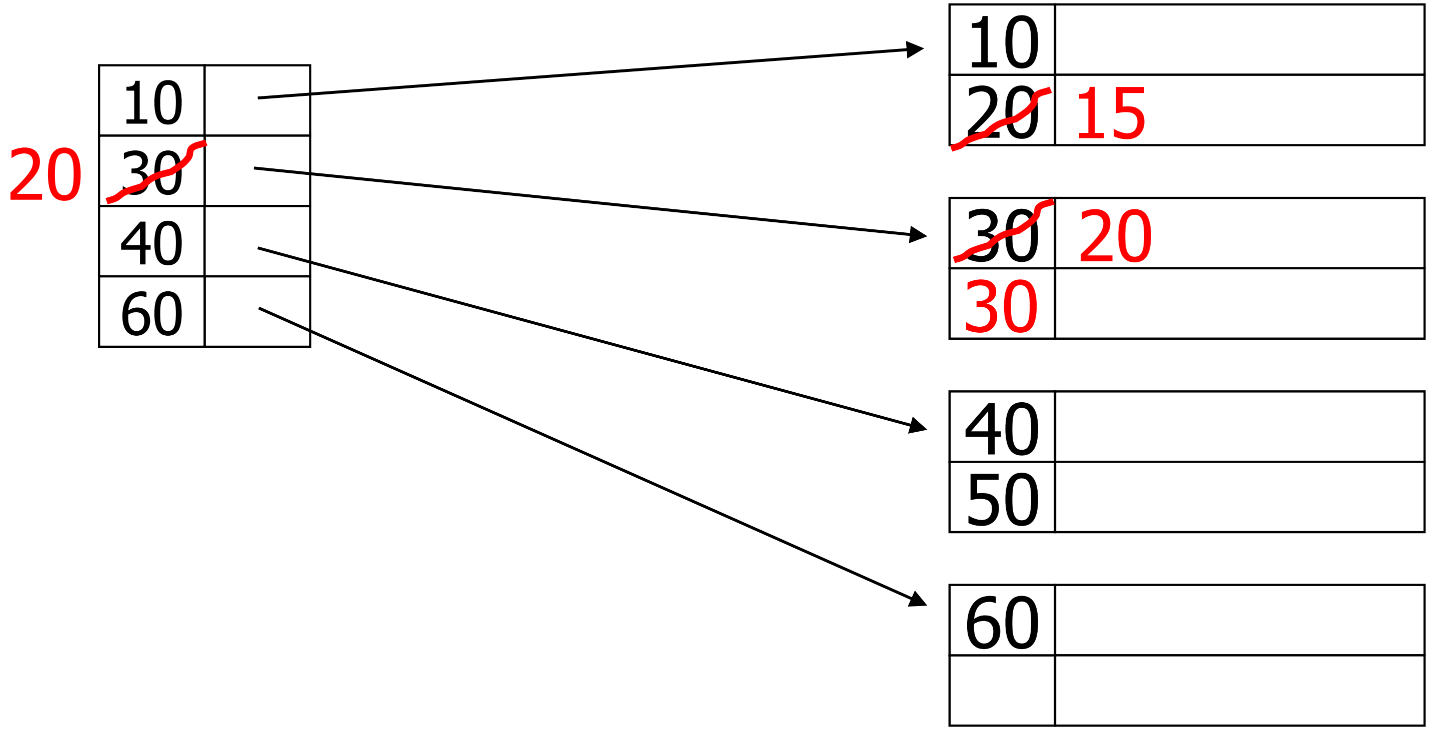
Insertion, sparse index case

- insert record 15



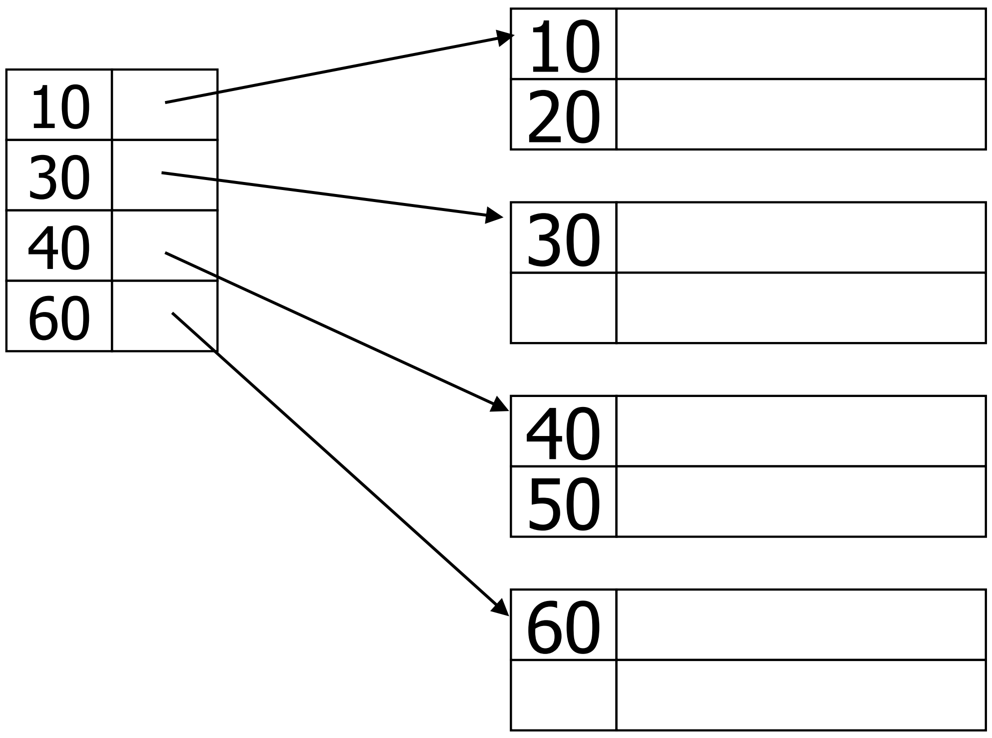
Insertion, sparse index case

– insert record 15



Insertion, sparse index case

– insert record 25



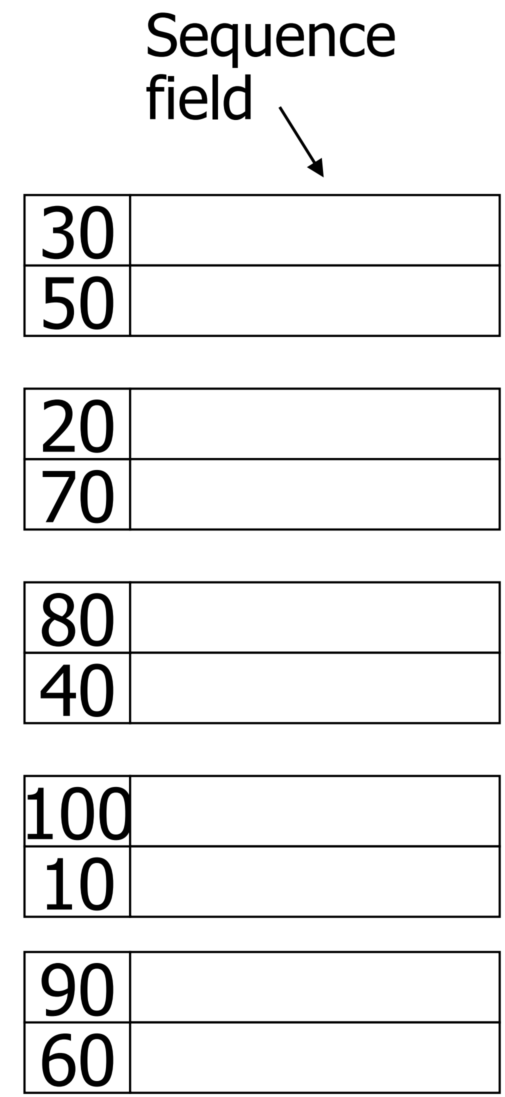
1. follow the index
   ↳ have to be after 10
2. we go to the page
3. we scan, we need to go to the next page
4. insert 25 and shift 30
5. replace the index key by 25

Insertion, dense index case

- Similar
- Often more expensive . . .

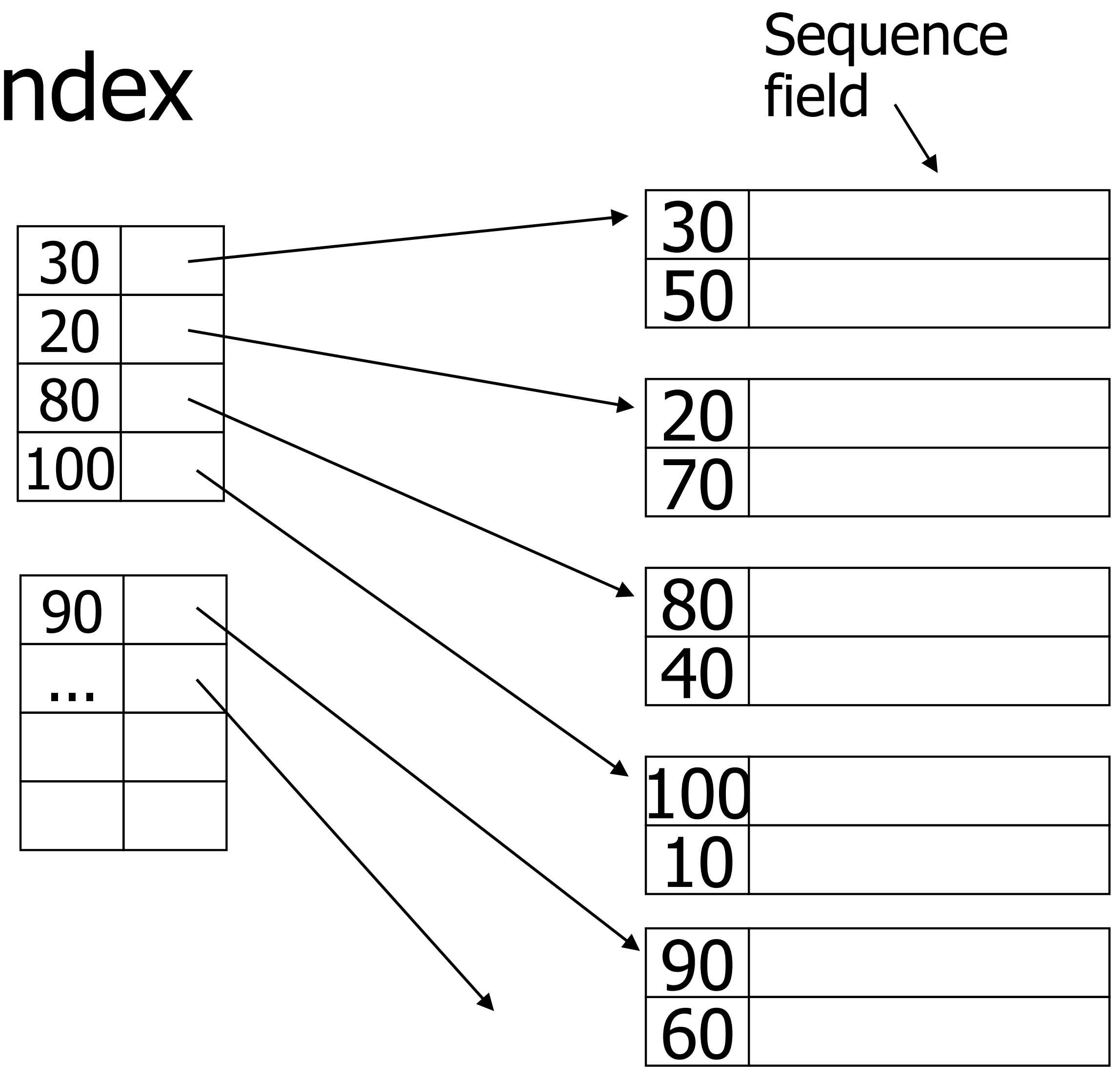
Secondary indexes

data is not stored sorted by the key



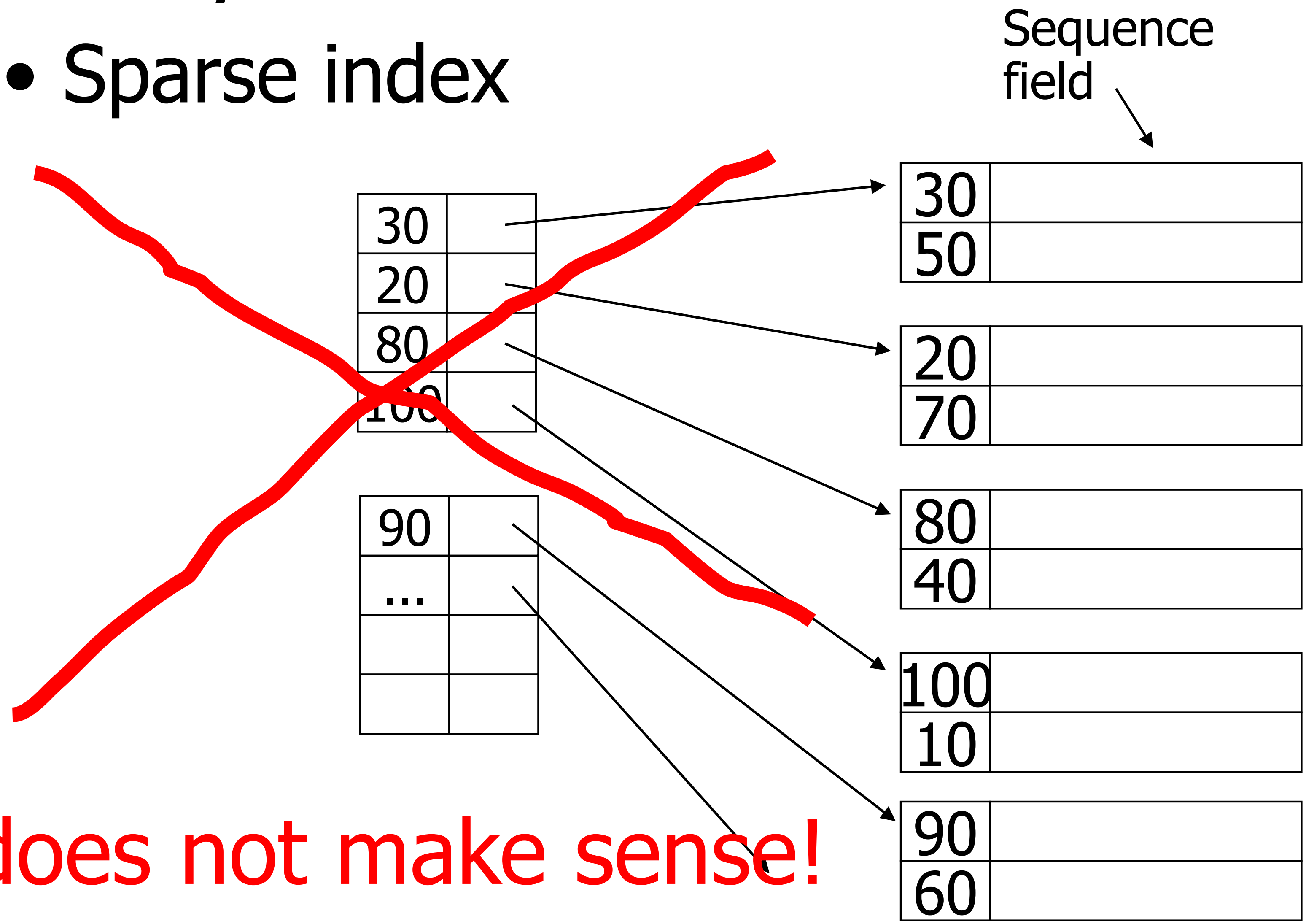
Secondary indexes

- Sparse index



Secondary indexes

- Sparse index

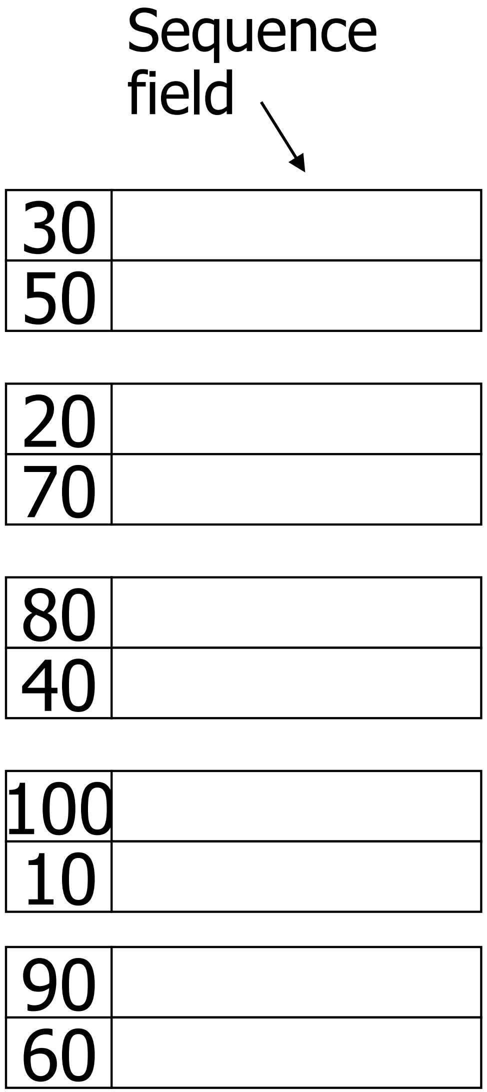


does not make sense!

Won't work because file is not sorted

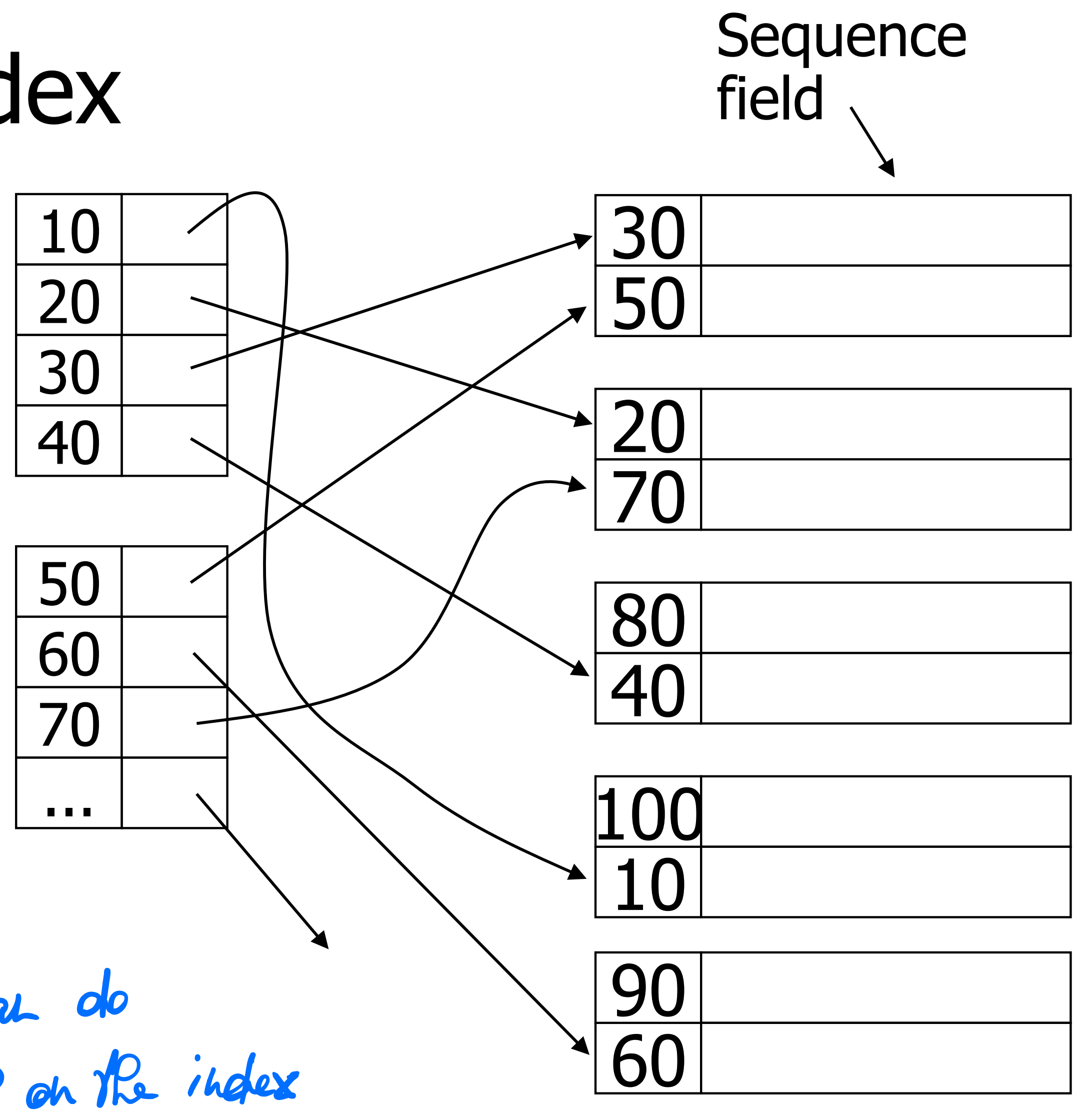
Secondary indexes

- Dense index



Secondary indexes

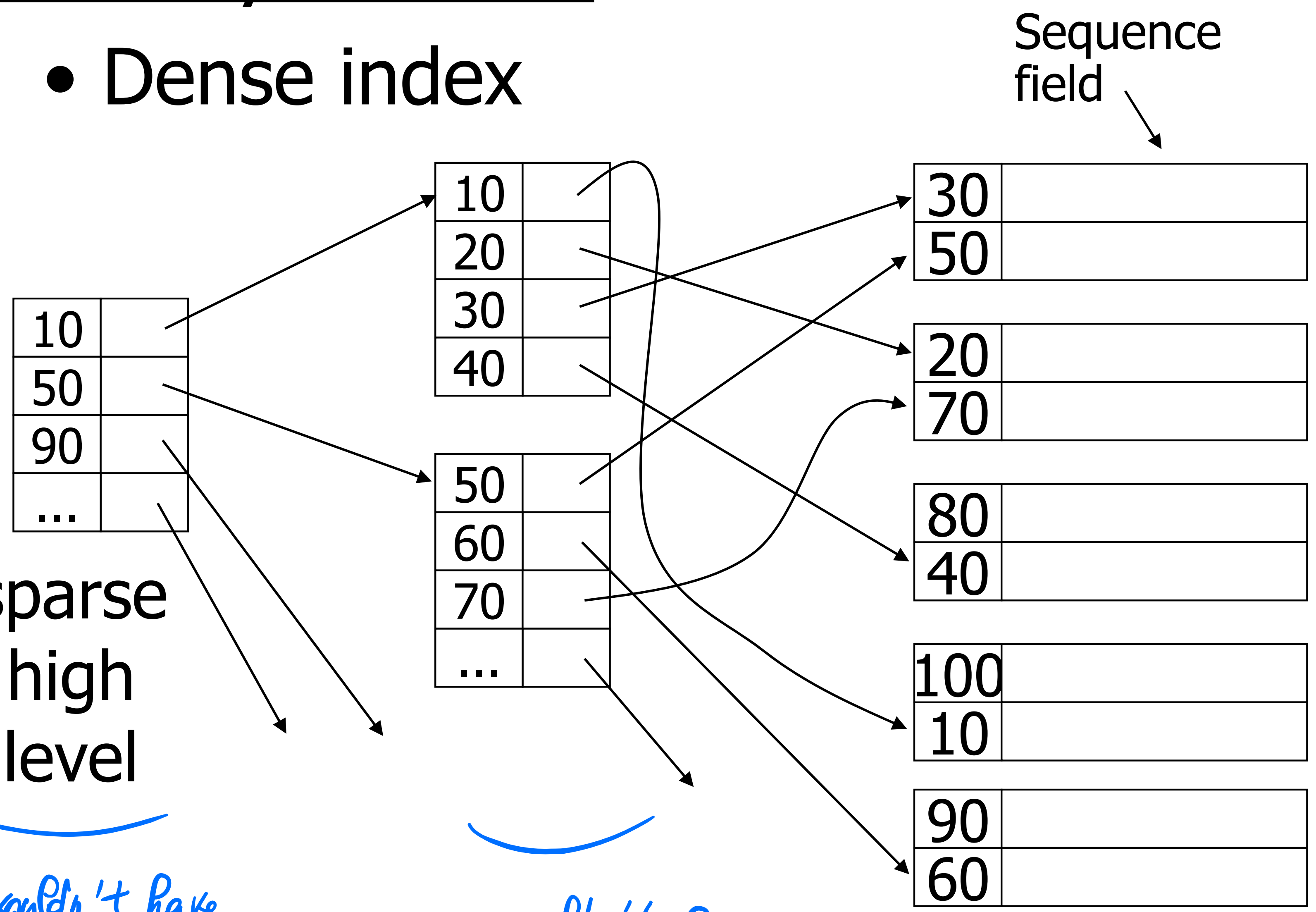
- Dense index



now we can do
binary search on the index
while we couldn't on the sequence
field

Secondary indexes

- Dense index



sparse high level

wouldn't have sense to be dense index

wouldn't have sense to be sparse index

With secondary indexes:

- Lowest level is dense
- Other levels are sparse

Duplicate values & secondary indexes

20	
10	

20	
40	

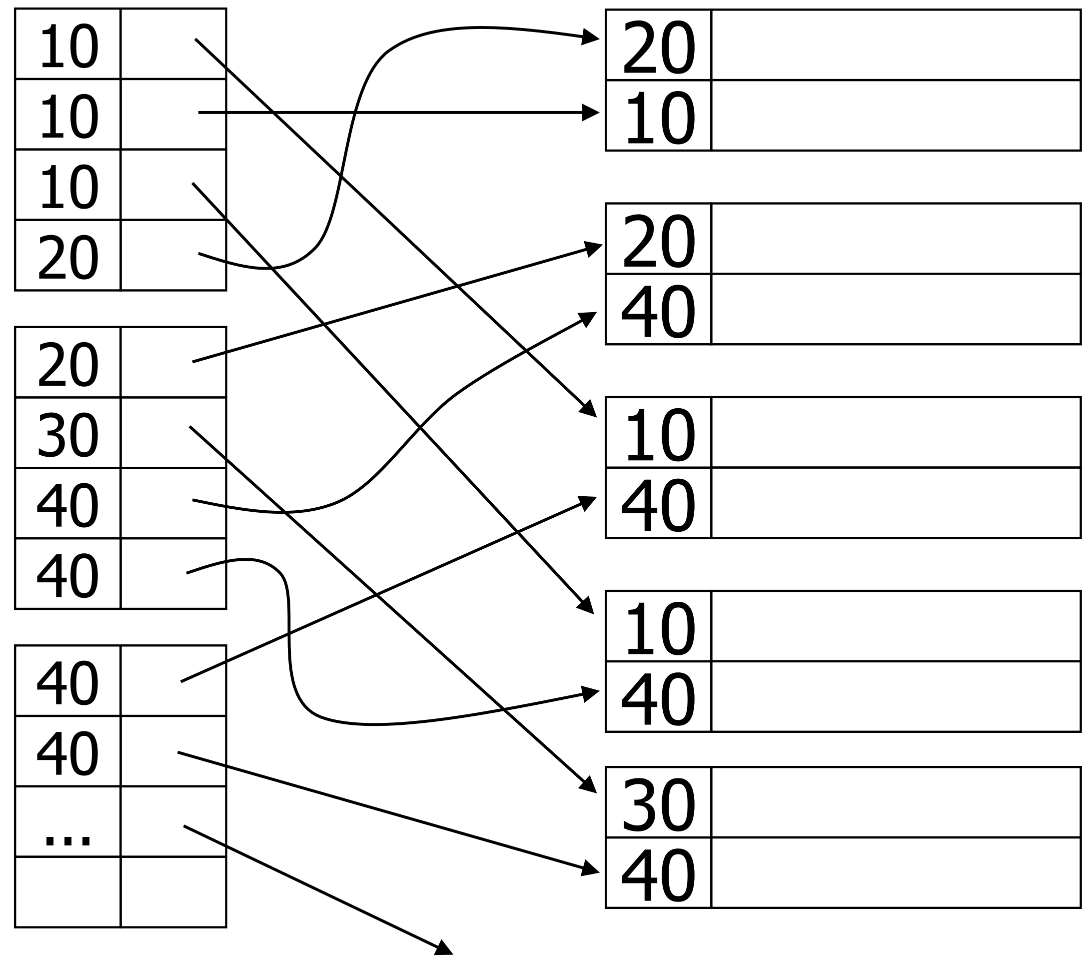
10	
40	

10	
40	

30	
40	

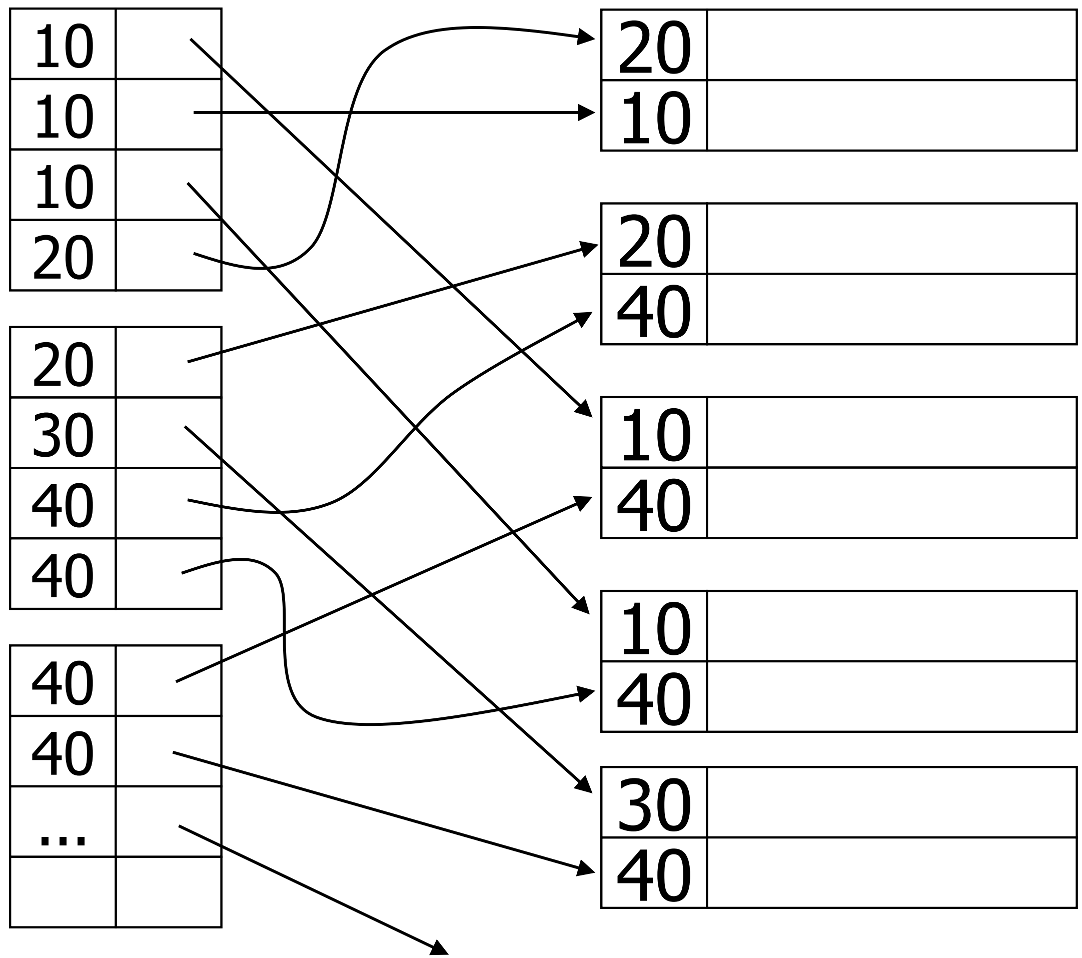
Duplicate values & secondary indexes

one option...



Duplicate values & secondary indexes

one option...

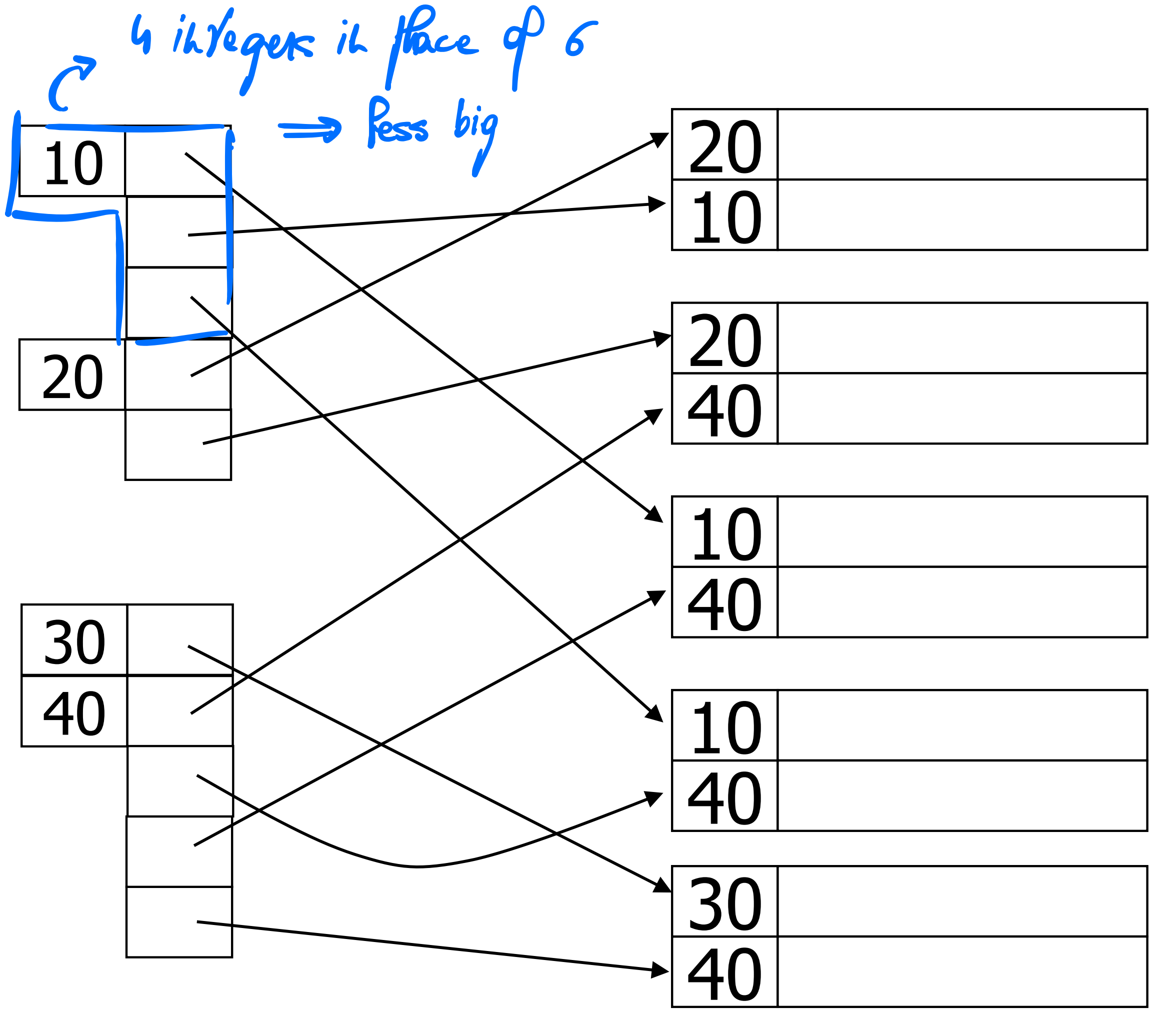


Problem:
excess overhead!

- disk space
- search time

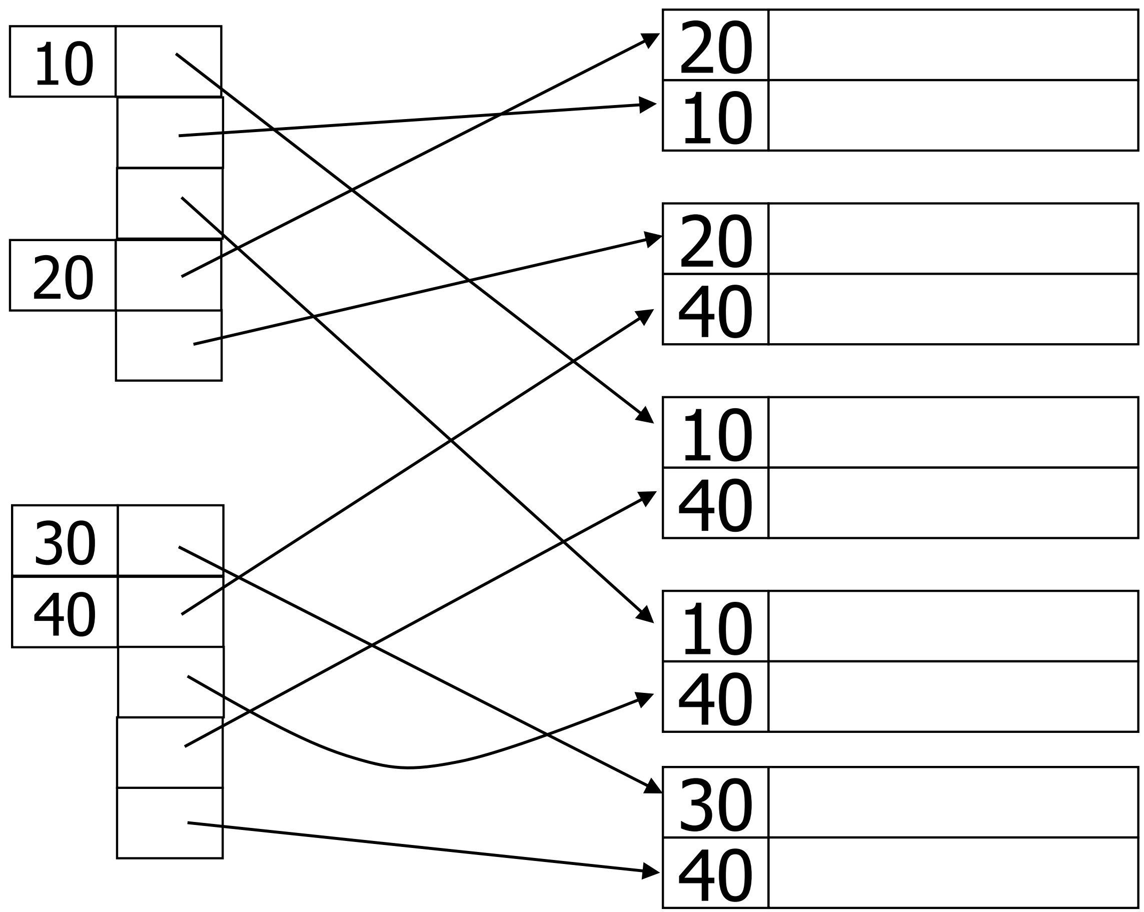
Duplicate values & secondary indexes

another option...



Duplicate values & secondary indexes

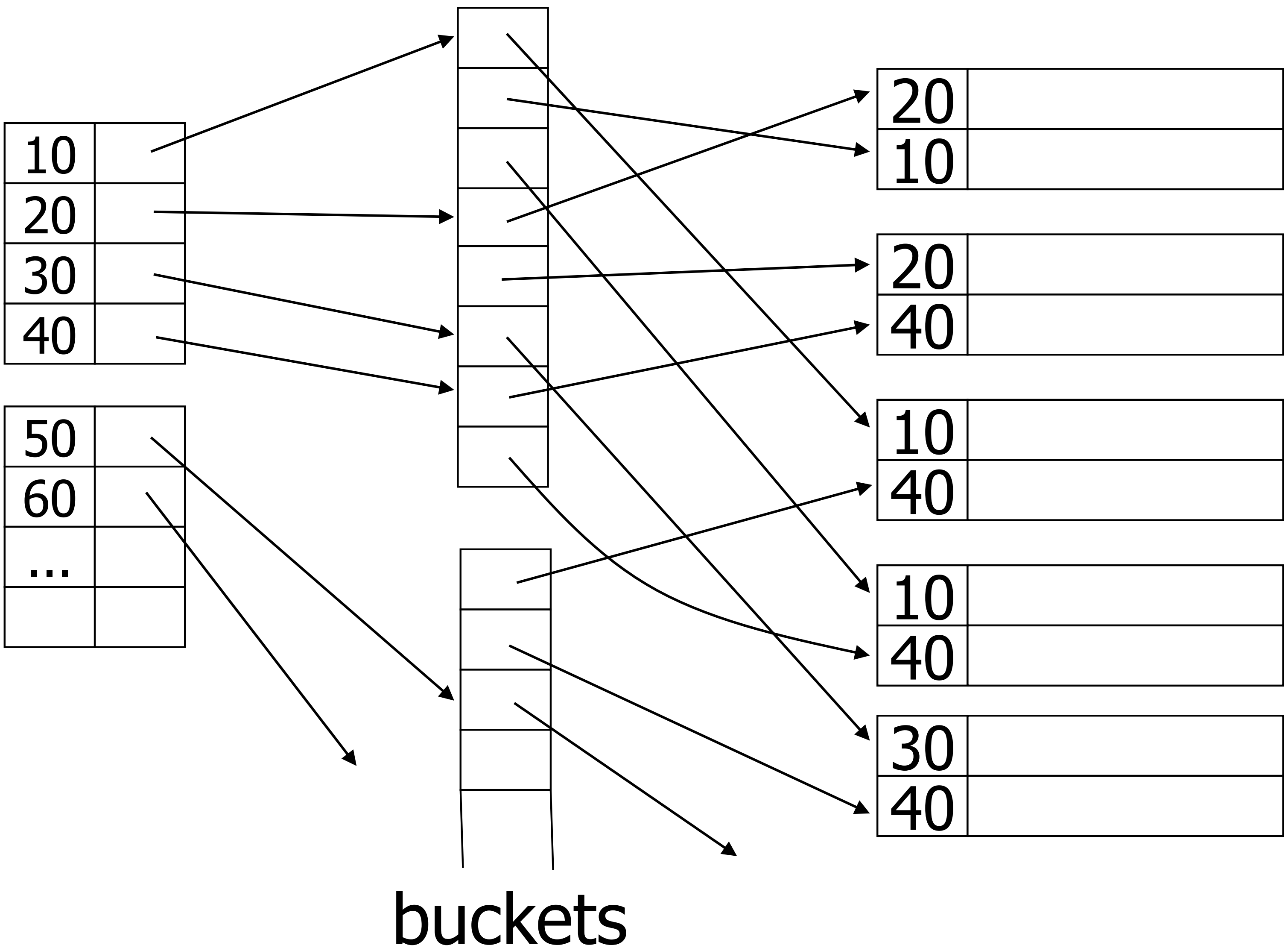
another option...



Problem: variable size records in index!

↳ we need fixed size records to do binary search

Duplicate values & secondary indexes

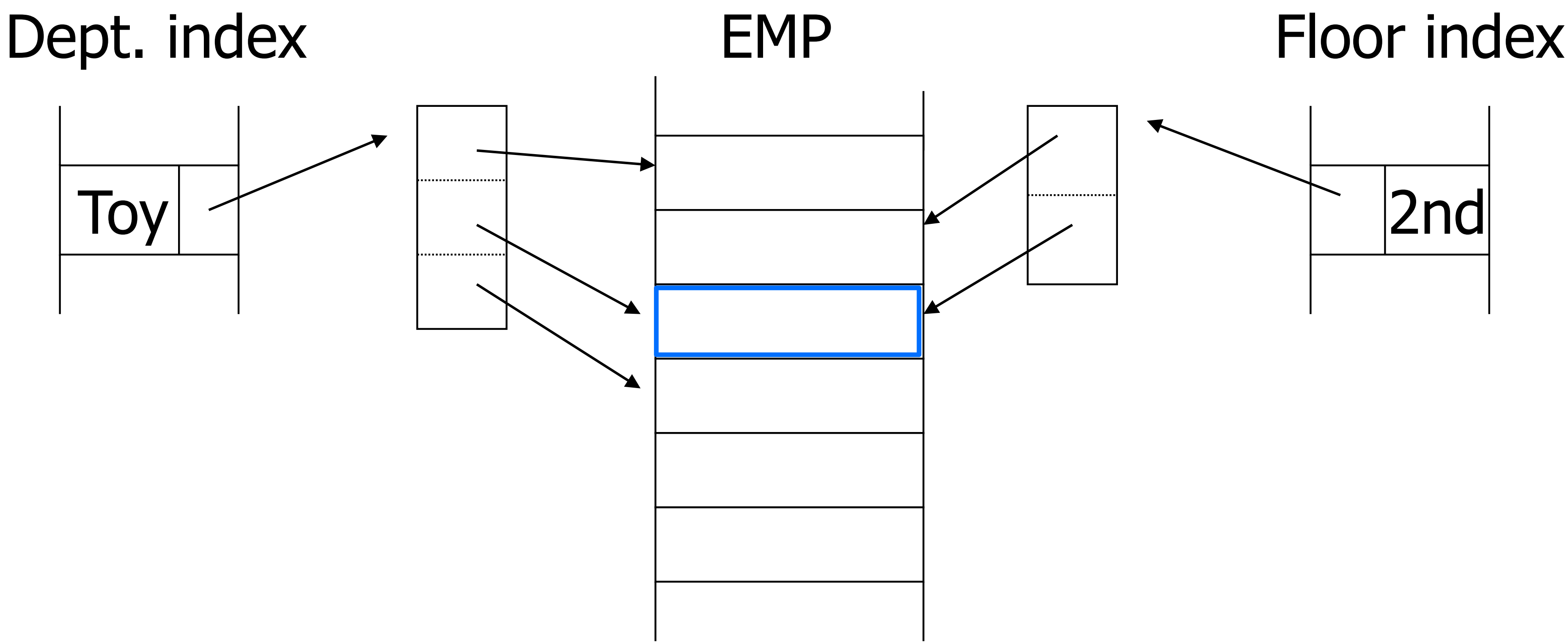


↳ still divided in pages

Why "bucket" idea is useful

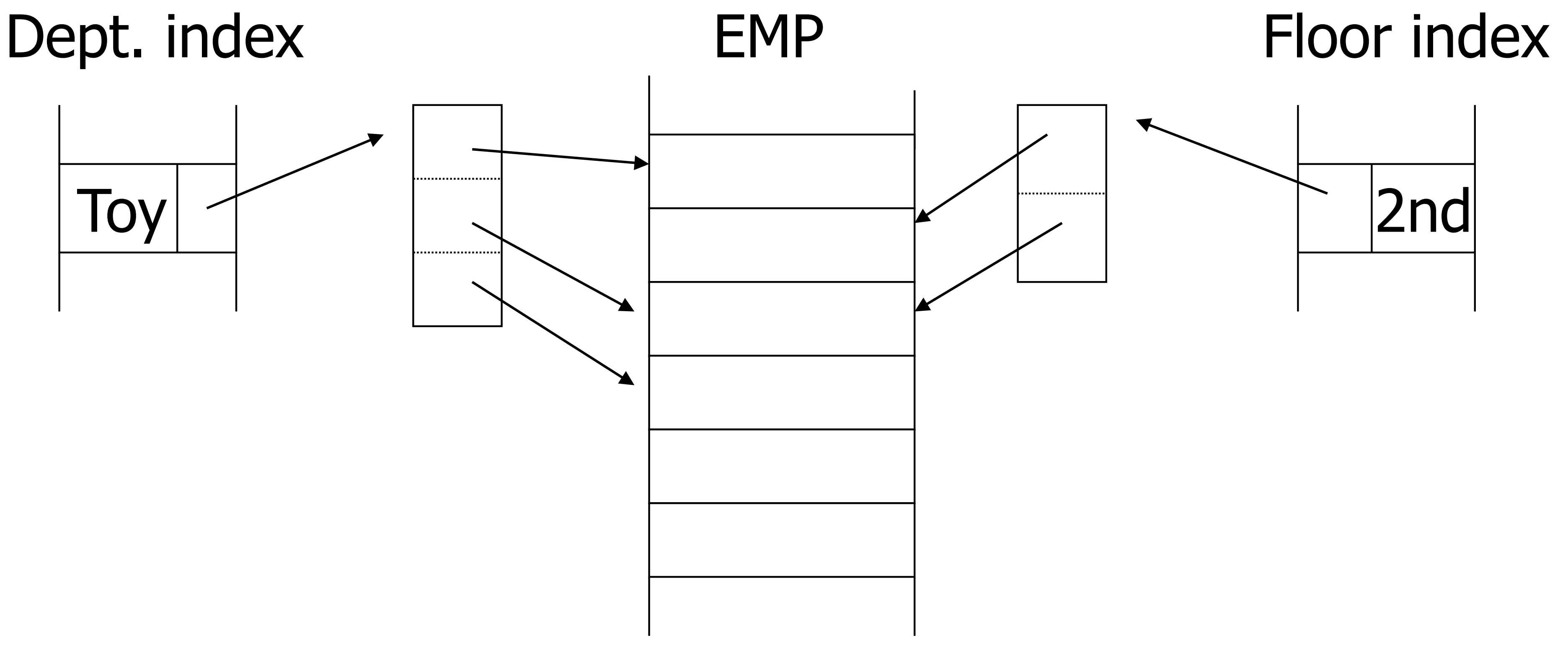
Indexes	Records
Name: primary	EMP (name,dept,floor,...)
Dept: secondary	
Floor: secondary	

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



because we have the buckets
we can compare the pointers
$\longrightarrow$ do not need to access the records

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

Note: those are building blocks
not used in practice

Conventional indexes

Advantage:

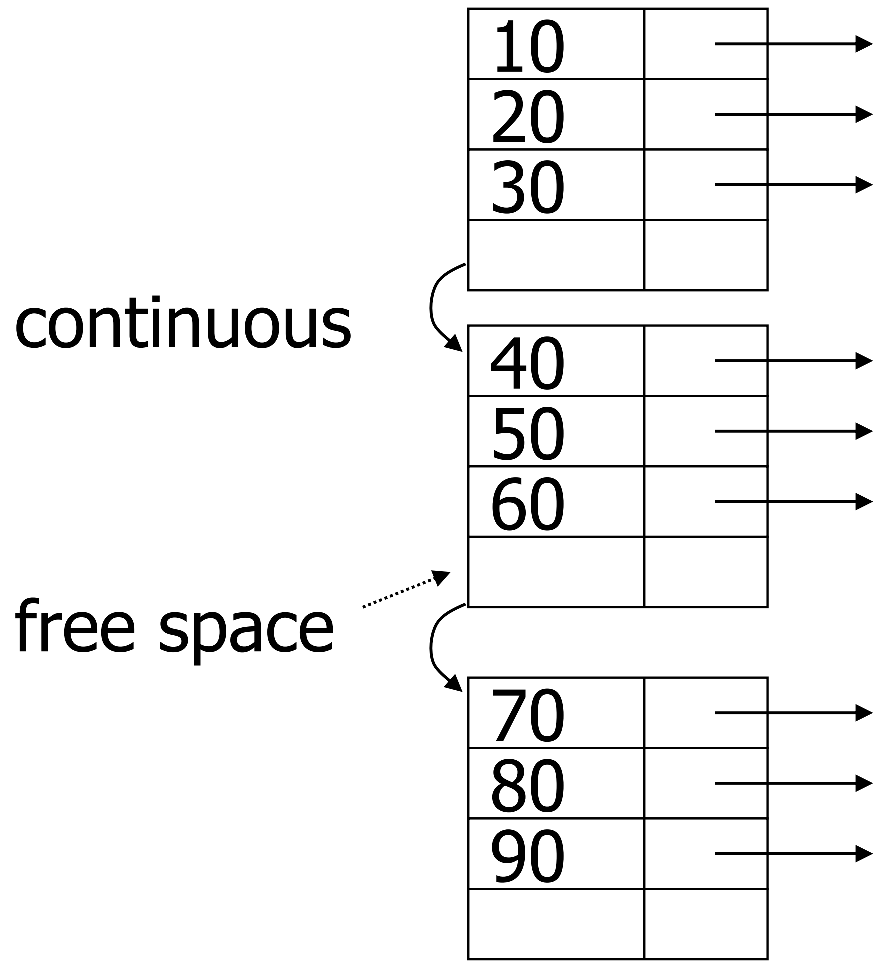
- Simple
- Index is sequential file
  good for scans

Disadvantage:

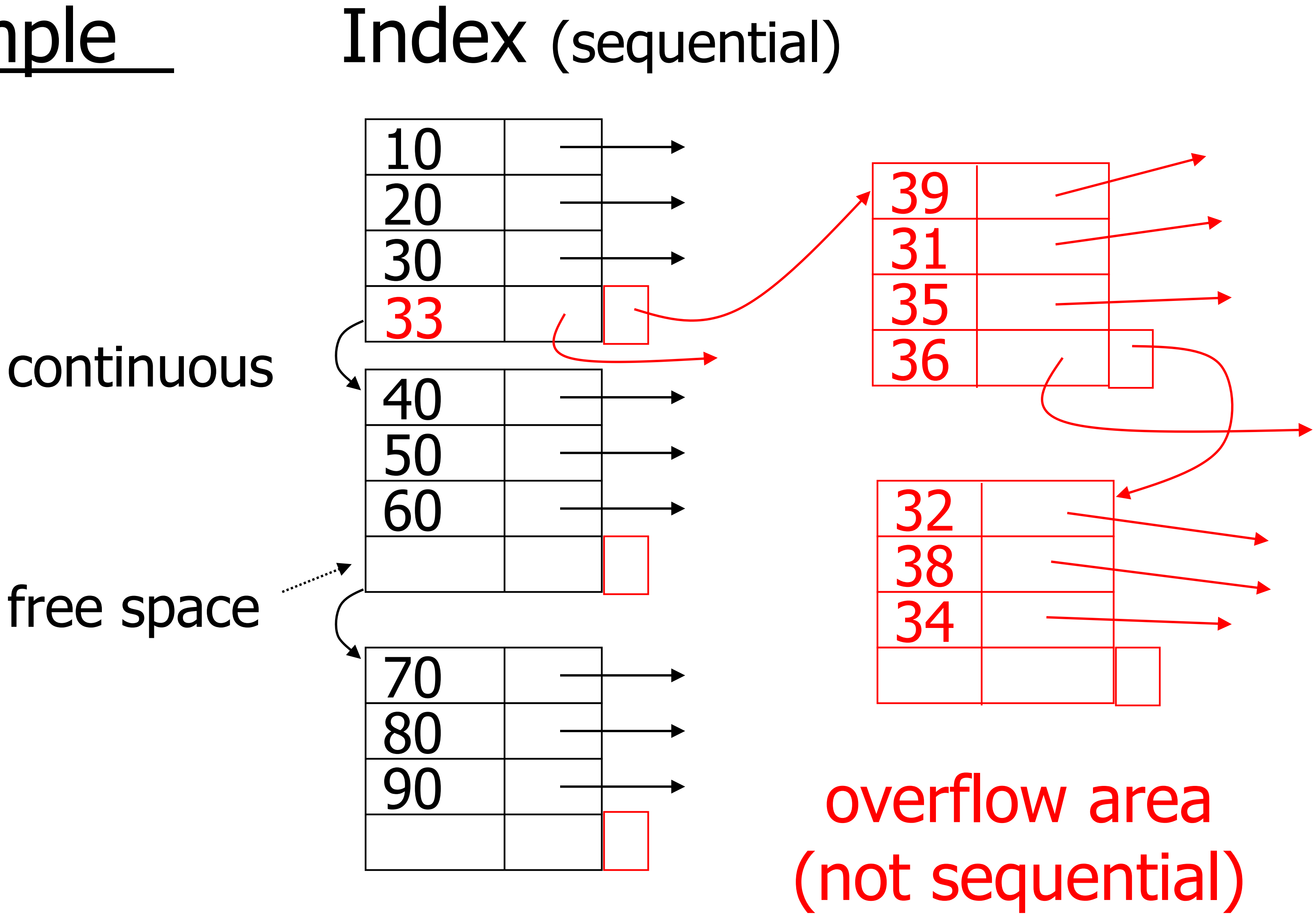
- Inserts expensive, or
- Lose sequentiality & balance

↳ the idea is to have indexes that are sorted
conventional indexes were sorted lists of key
⟹ maybe using other data structure would be better
↳ binary trees log(n) insertion and search

Example Index (sequential)



Example


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

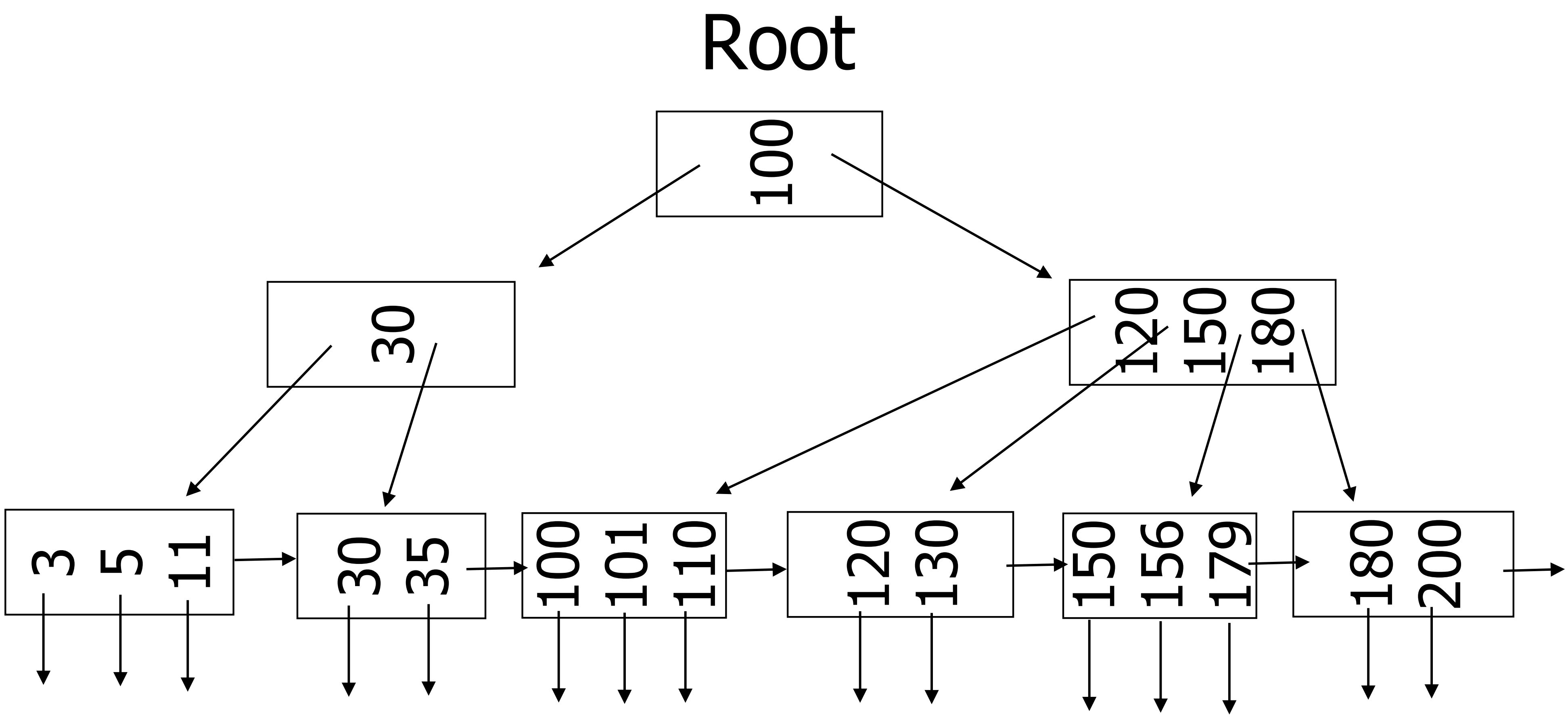
Outline:

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

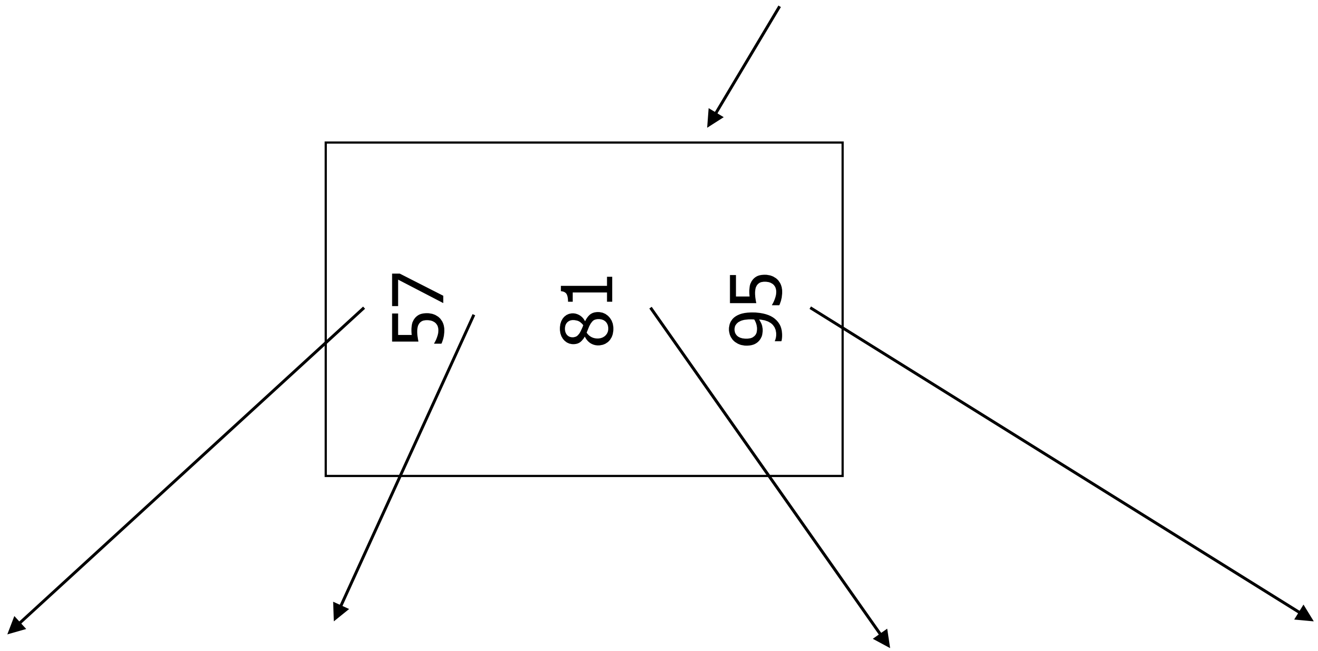
- NEXT: Another type of index
  - Give up on sequentiality of index
  - Try to get "balance"

unbalanced trees
are far from the
log(n) insertion
and search

B+Tree Example n=3

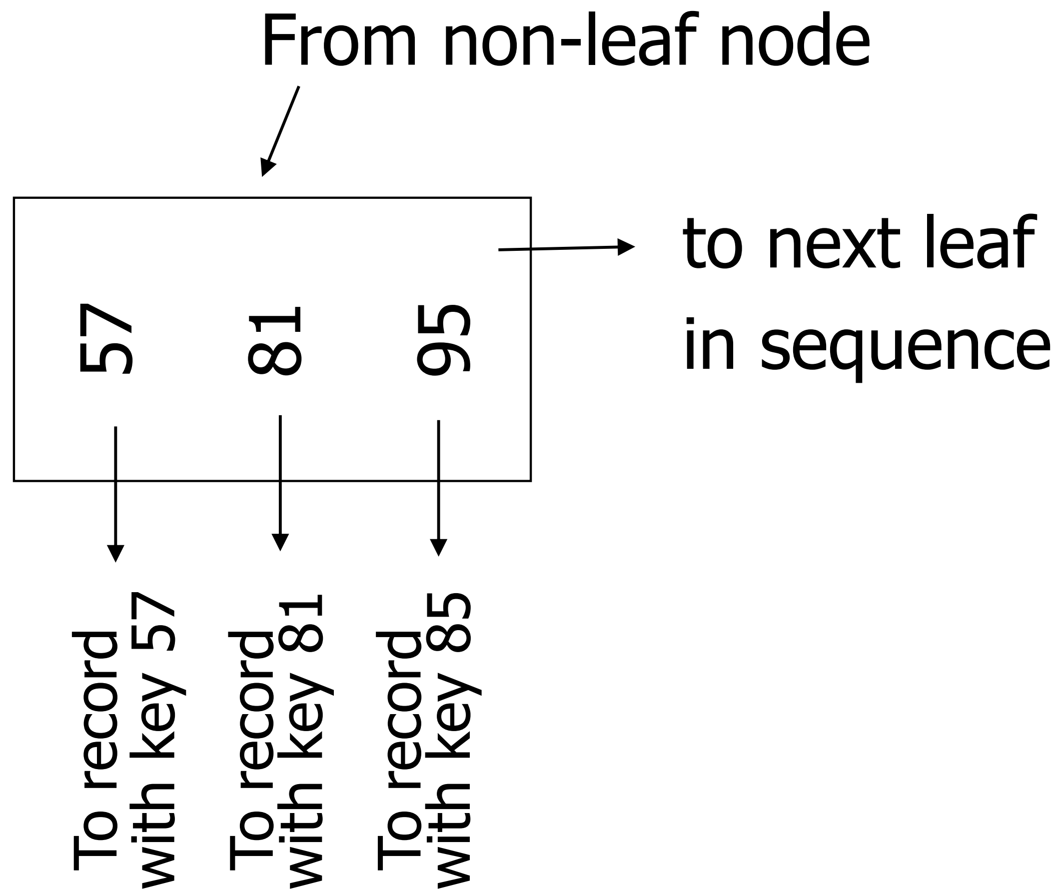


Sample non-leaf



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

Sample leaf node:



Size of nodes: $\left\{ \begin{array}{l} \text{n+1 pointers} \\ \text{n keys} \end{array} \right.$ <u>(fixed)</u>

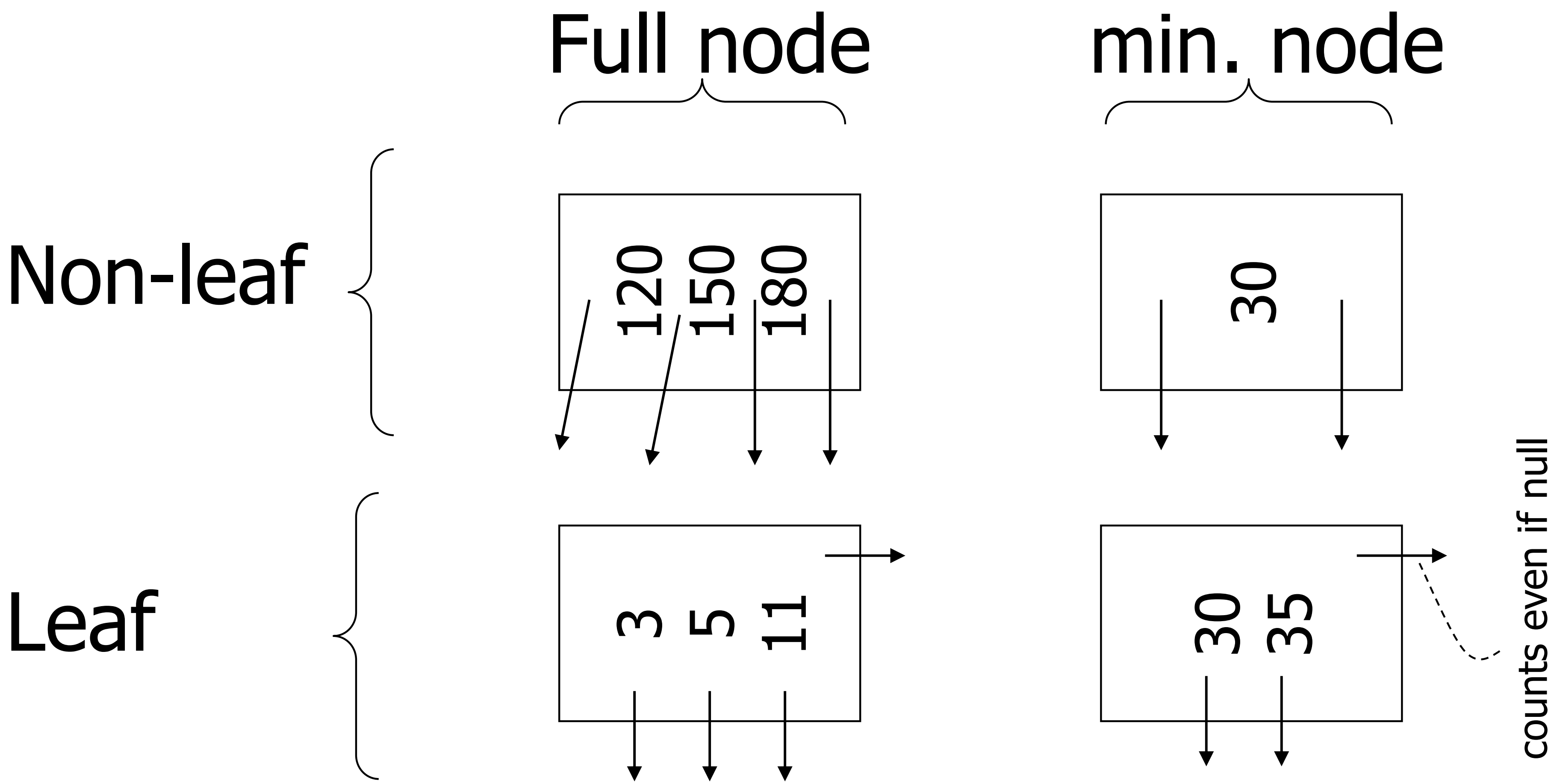
Don't want nodes to be too empty

- 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
120
150
180
30
Leaf
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"

↳ point to the 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

rule broken
only for the root

Insert into B+tree

whole idea of using a binary tree

↳ low cost insertion

(a) simple case

- space available in leaf

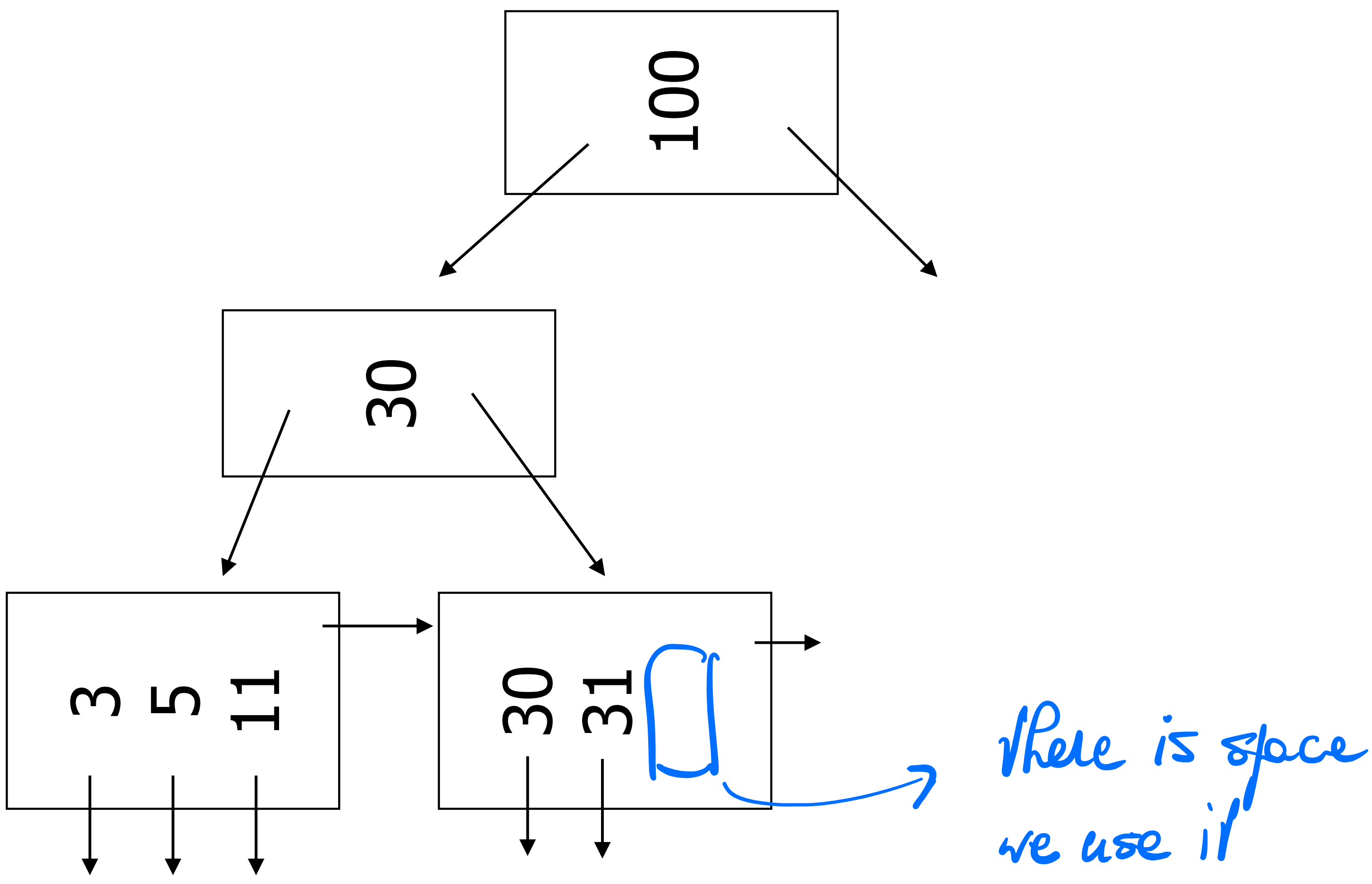
(b) leaf overflow

(c) non-leaf overflow

(d) new root

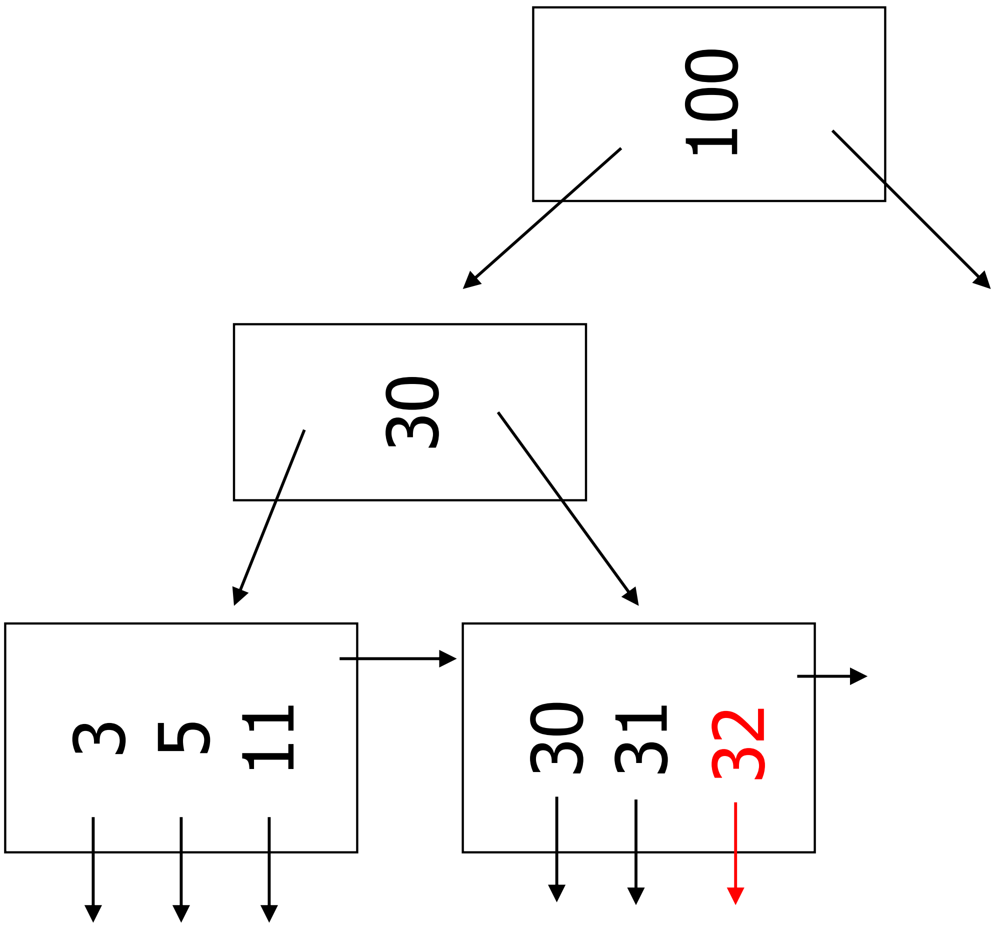
(a) Insert key = 32

$\boxed{n=3}$



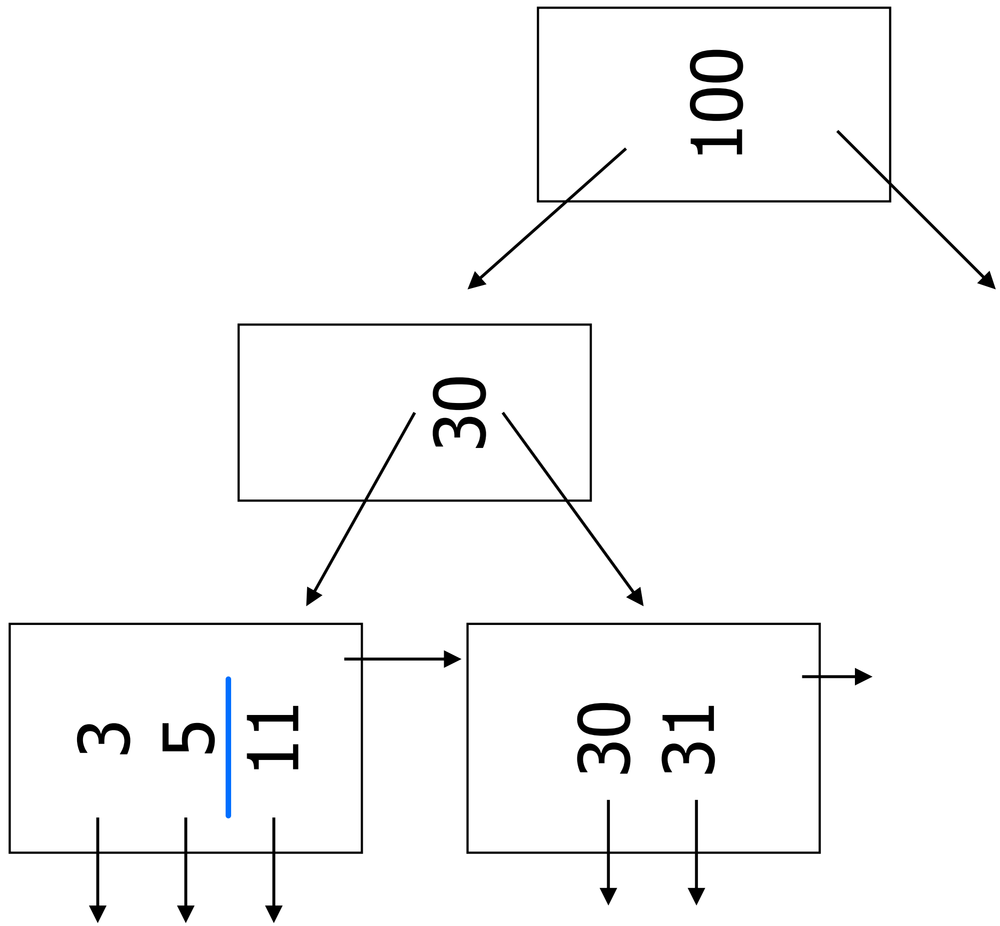
(a) Insert key = 32

n=3



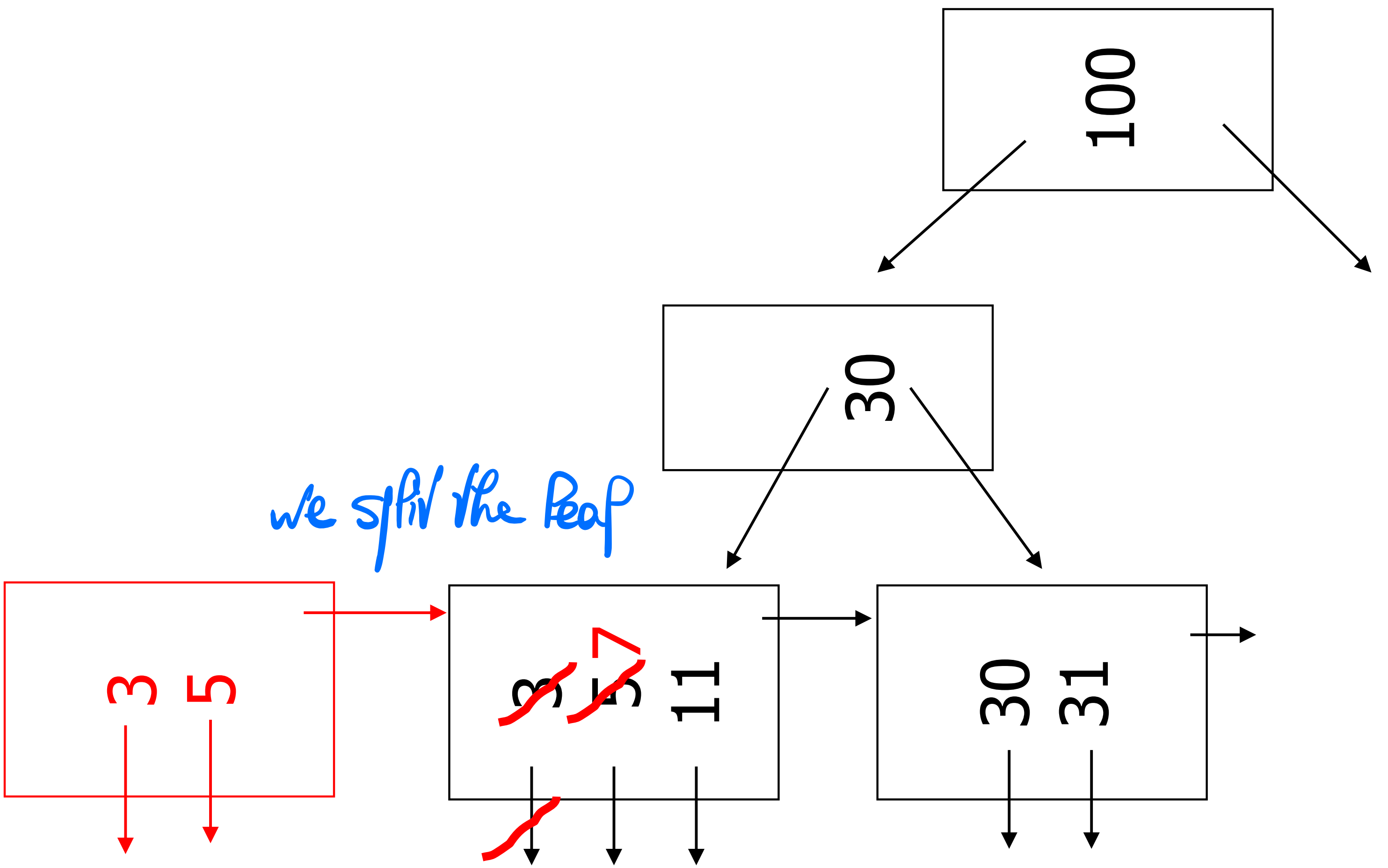
(a) Insert key = 7

n=3



(a) Insert key = 7

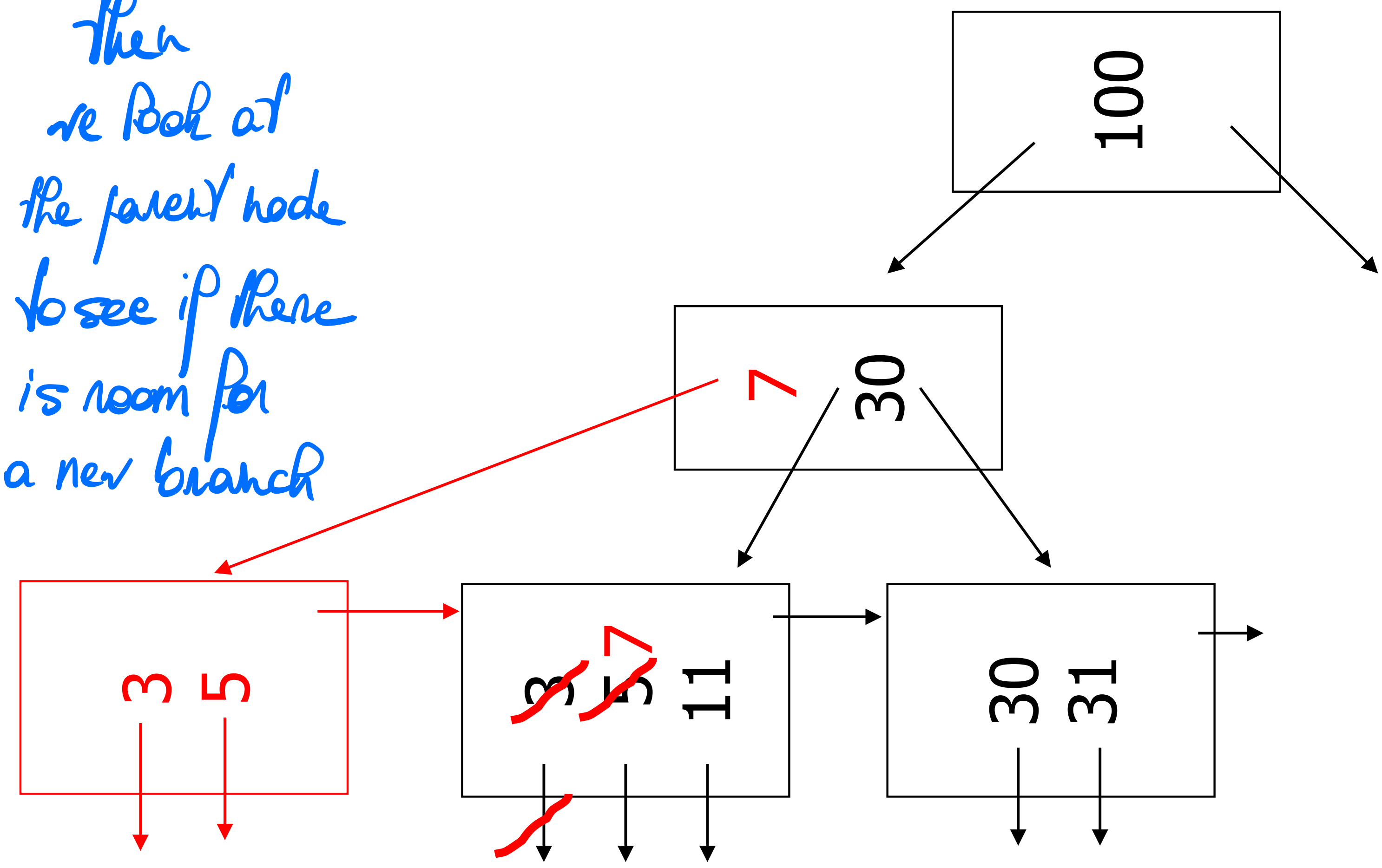
n=3



(a) Insert key = 7

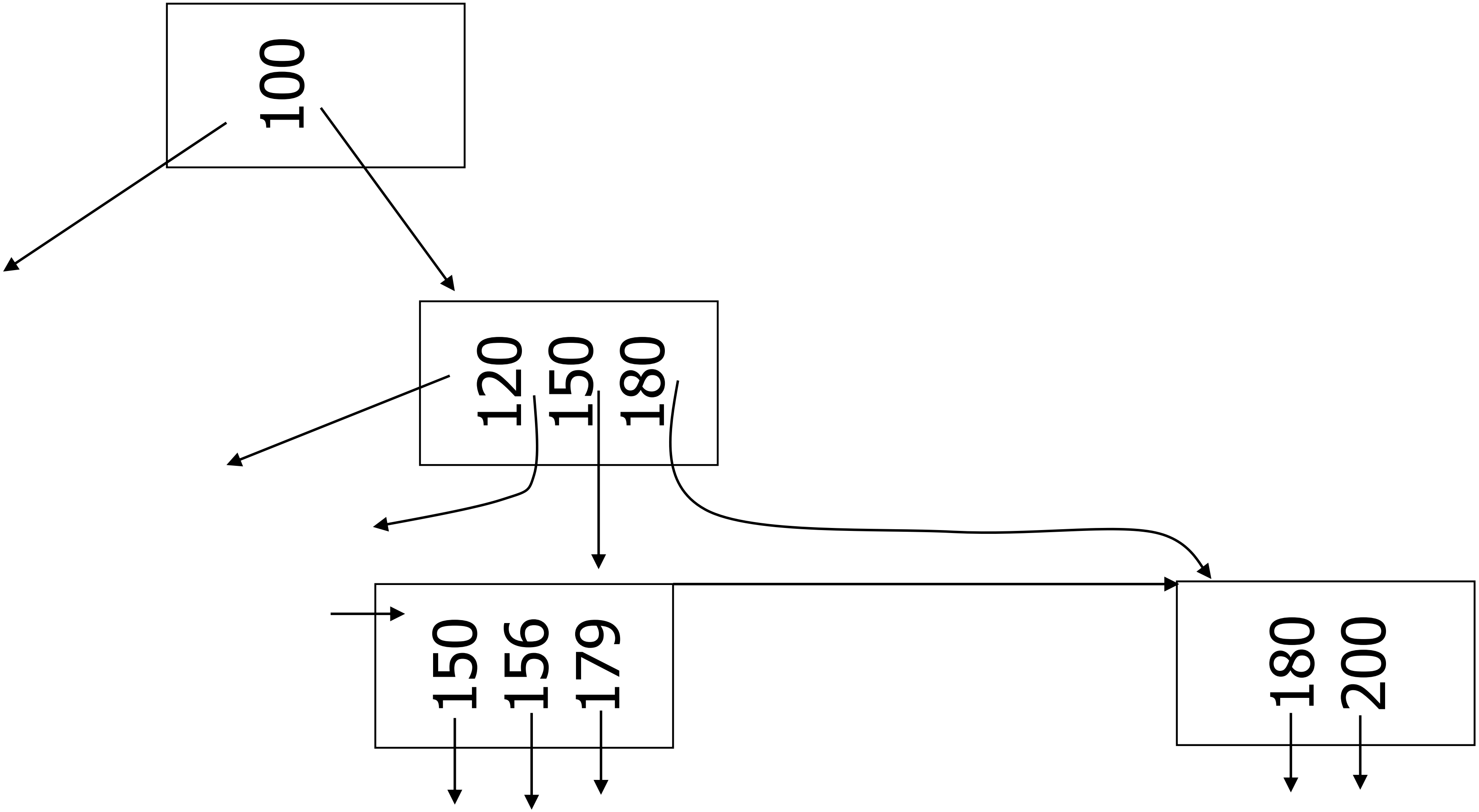
$\boxed{n=3}$

Then we look at the parent node to see if there is room for a new branch



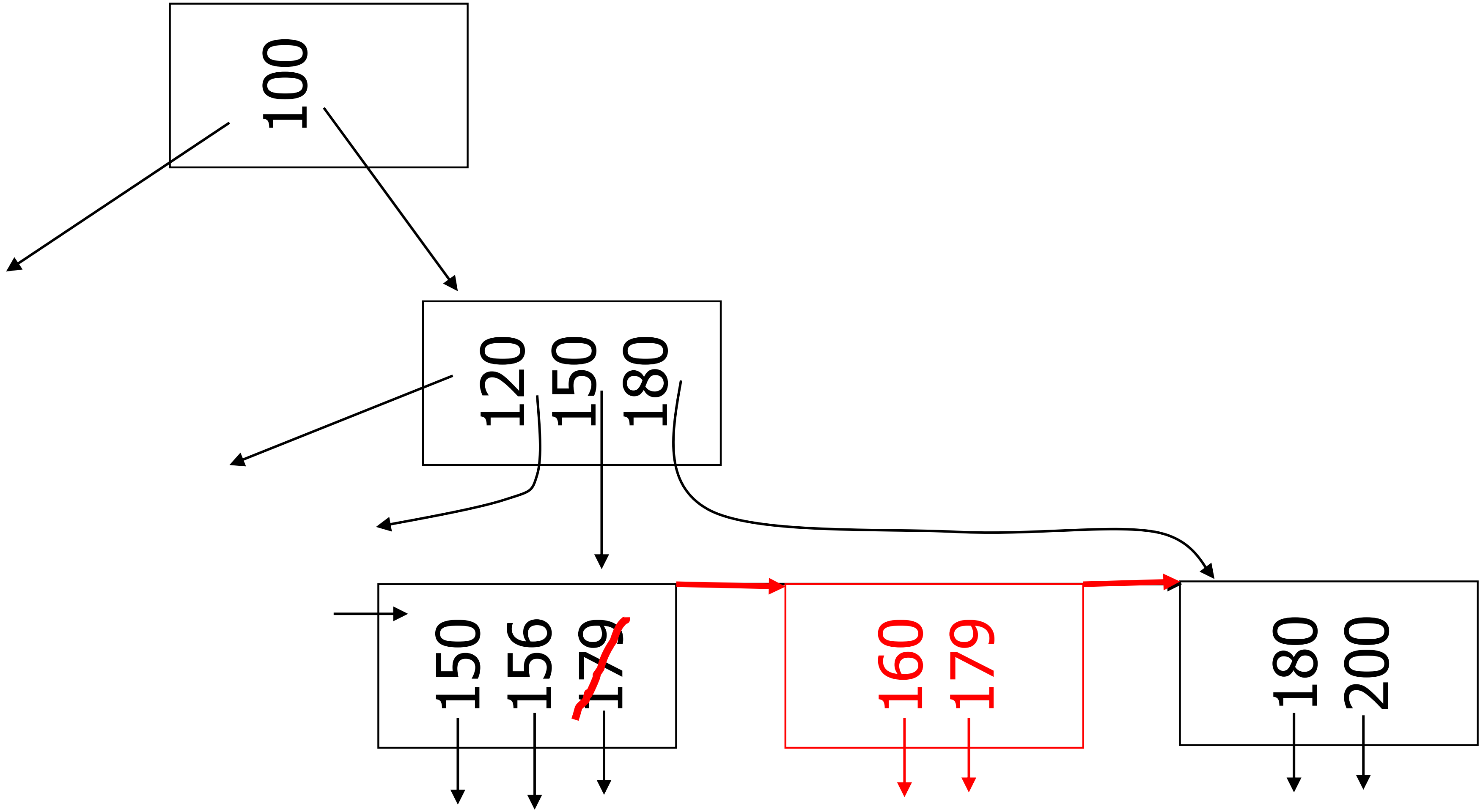
(c) Insert key = 160

n=3



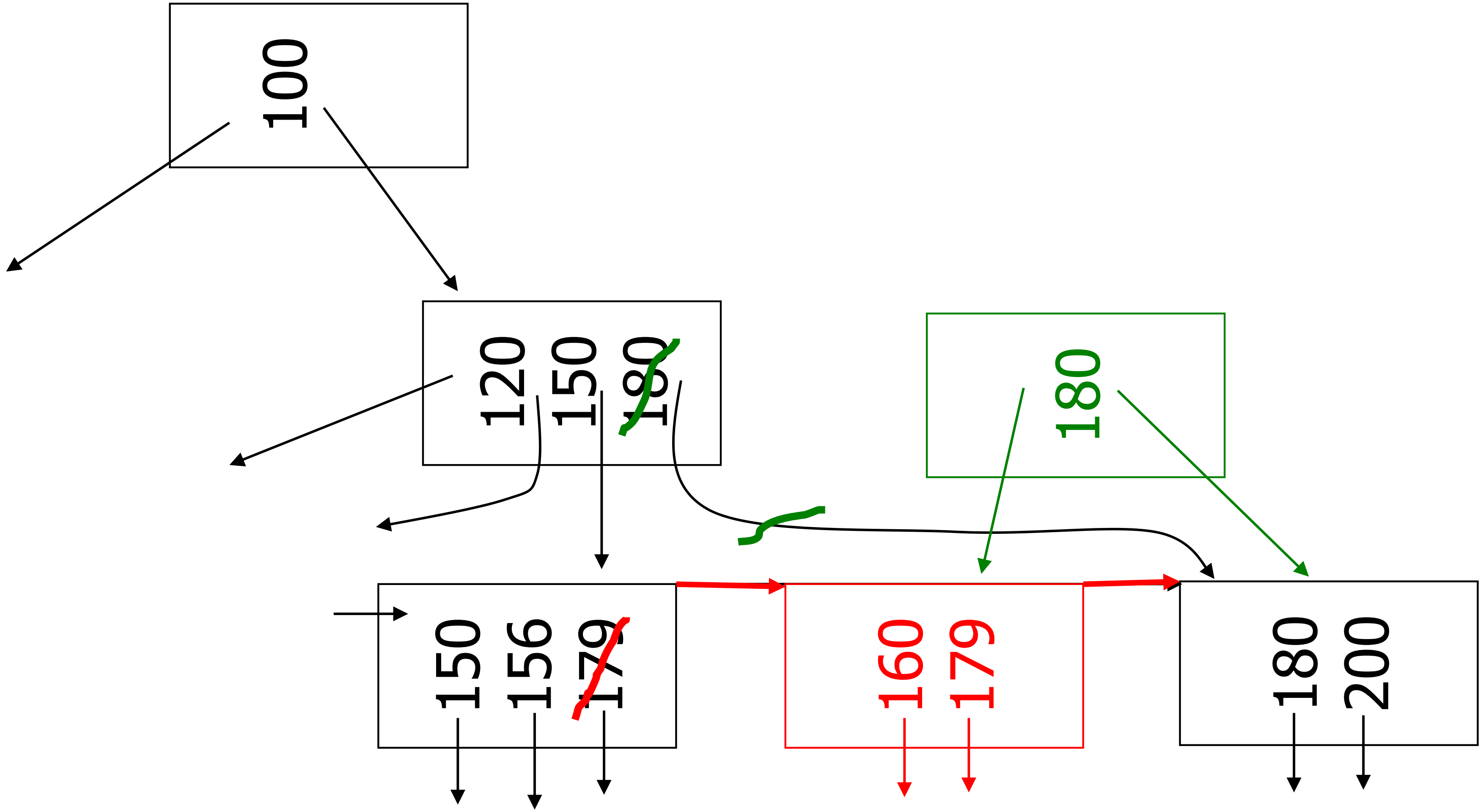
(c) Insert key = 160

n=3



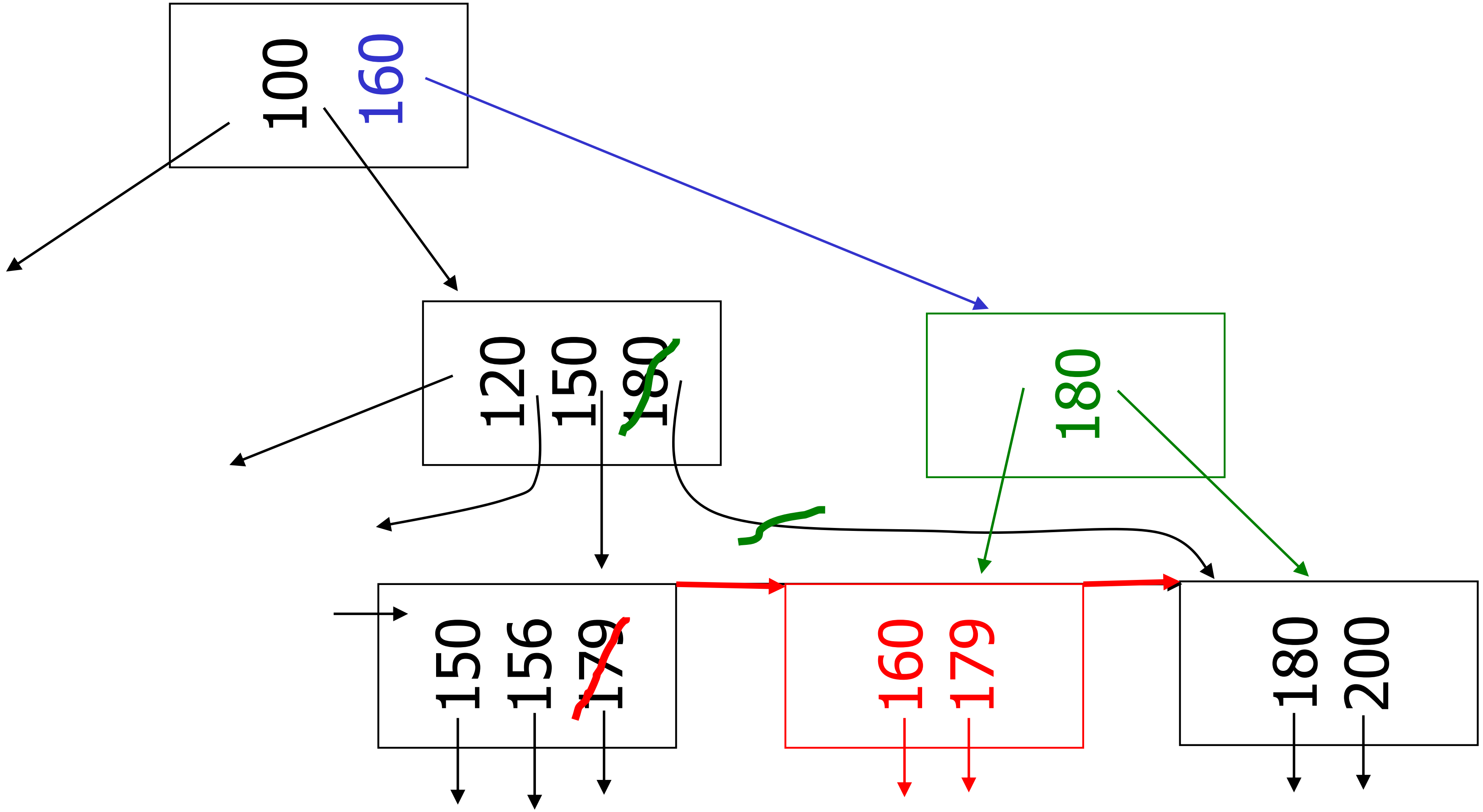
(c) Insert key = 160

n=3



(c) Insert key = 160

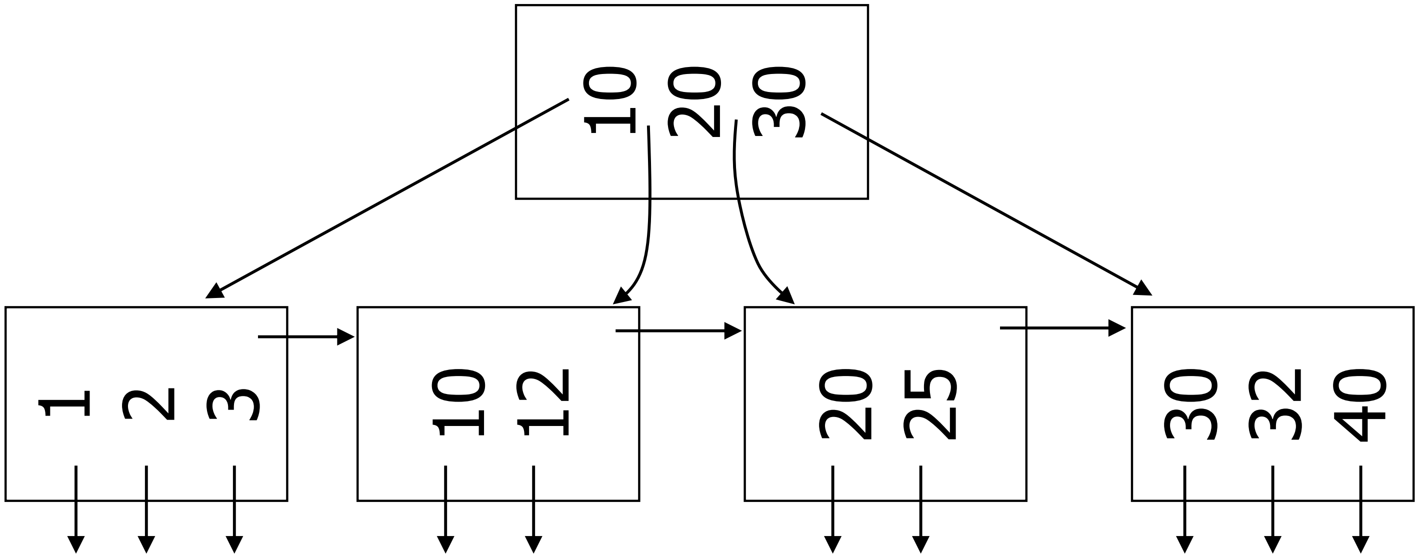
n=3



(d) New root, insert 45

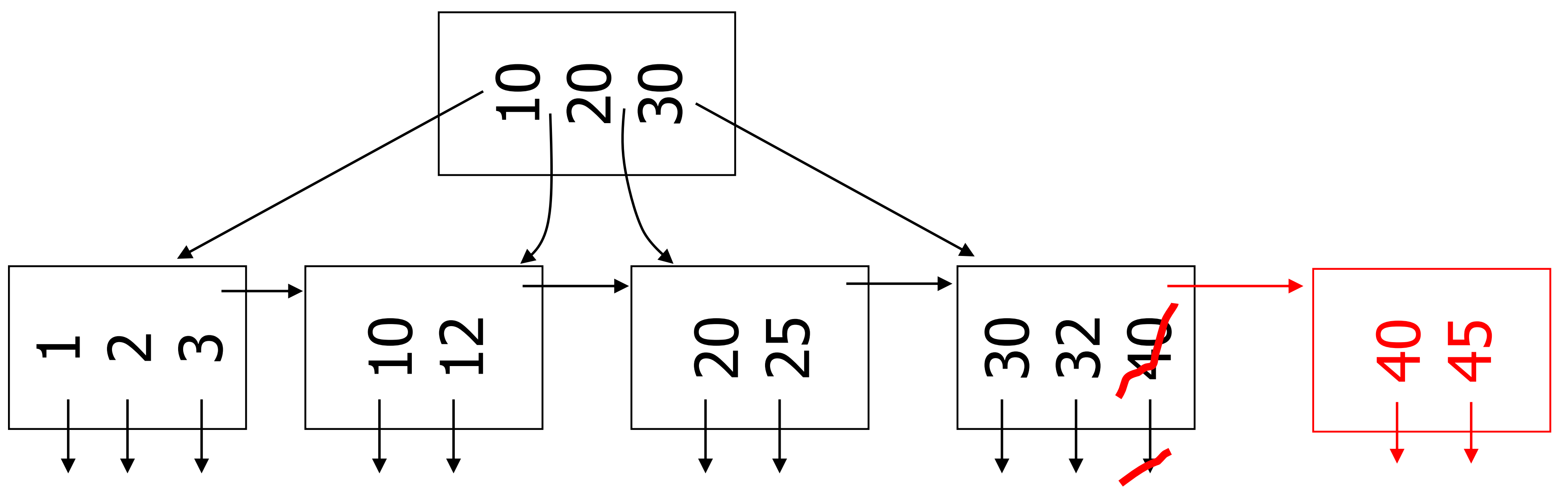
n=3

root is full



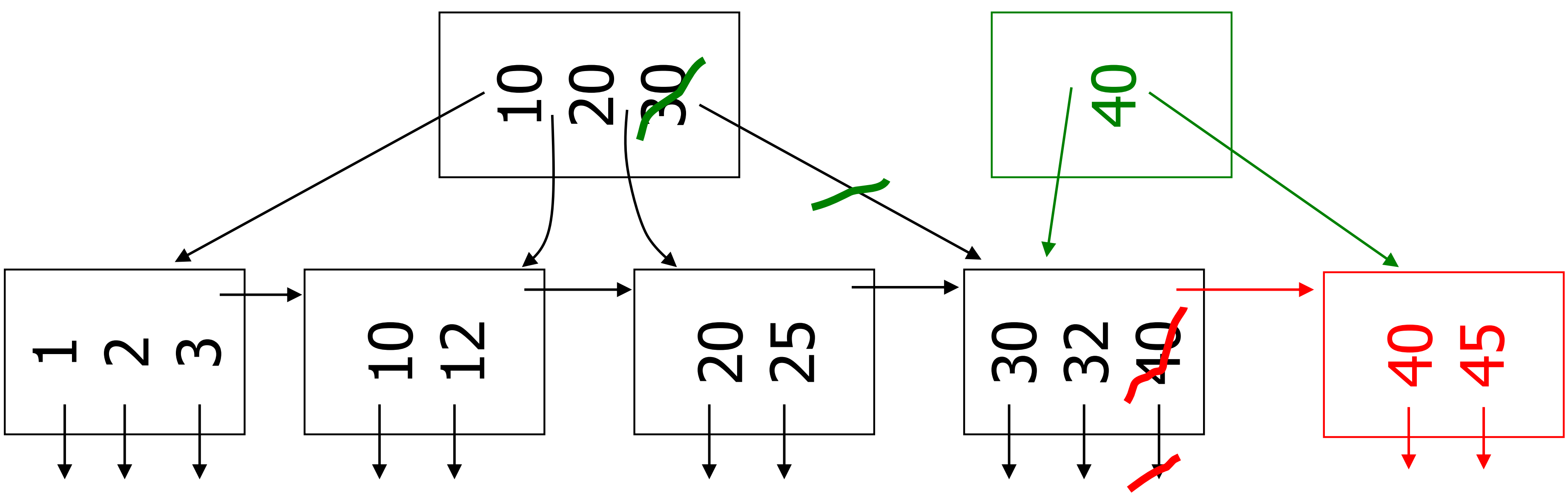
(d) New root, insert 45

n=3



(d) New root, insert 45

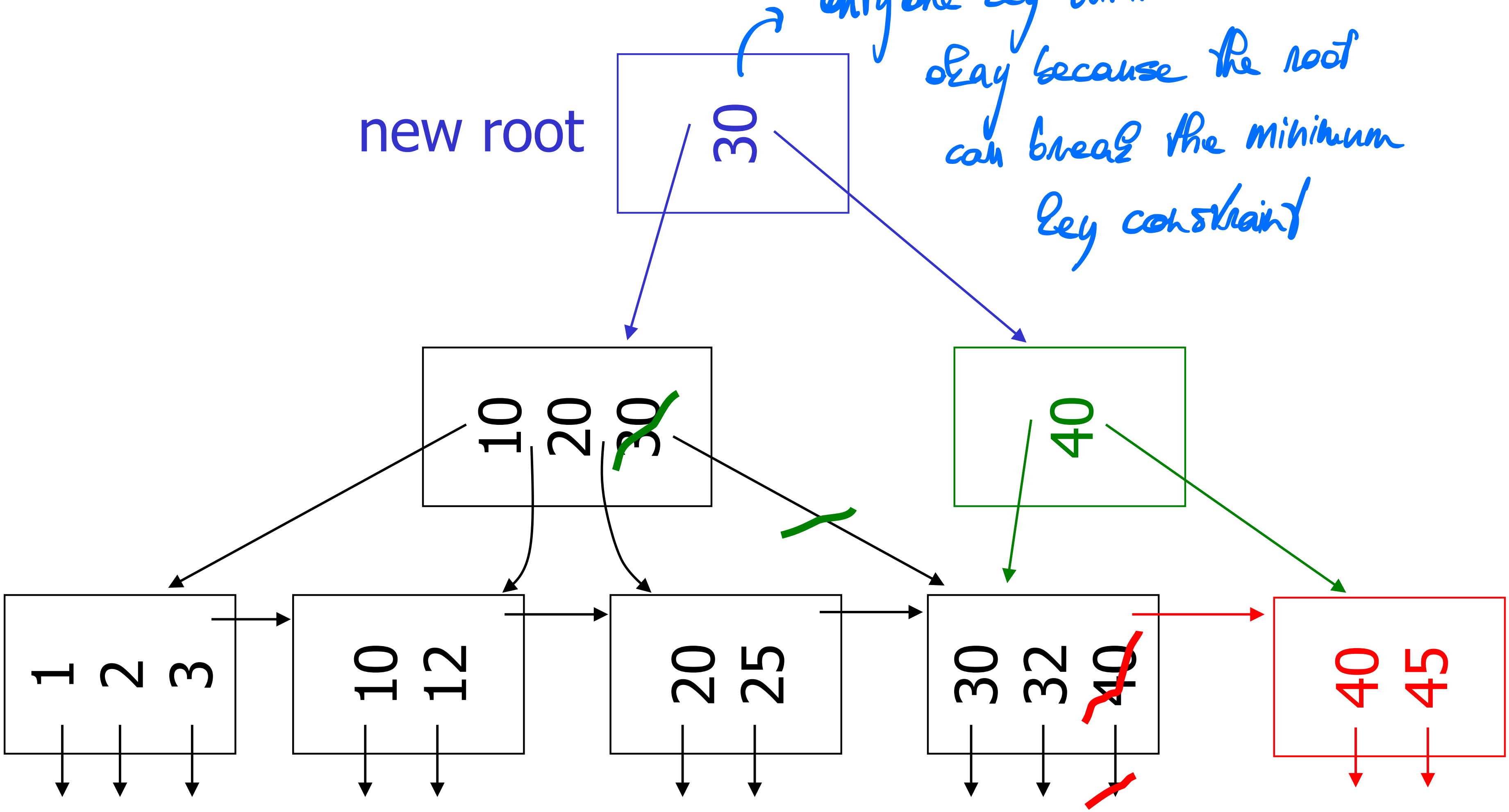
n=3



(d) New root, insert 45

n=3

only one key but it's okay because the root can break the minimum key constraint



Deletion from B+tree

go fast through it
↳ not implemented in reality because too hard and too expensive

(a) Simple case - no example

(b) Coalesce with neighbor (sibling)

(c) Re-distribute keys

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

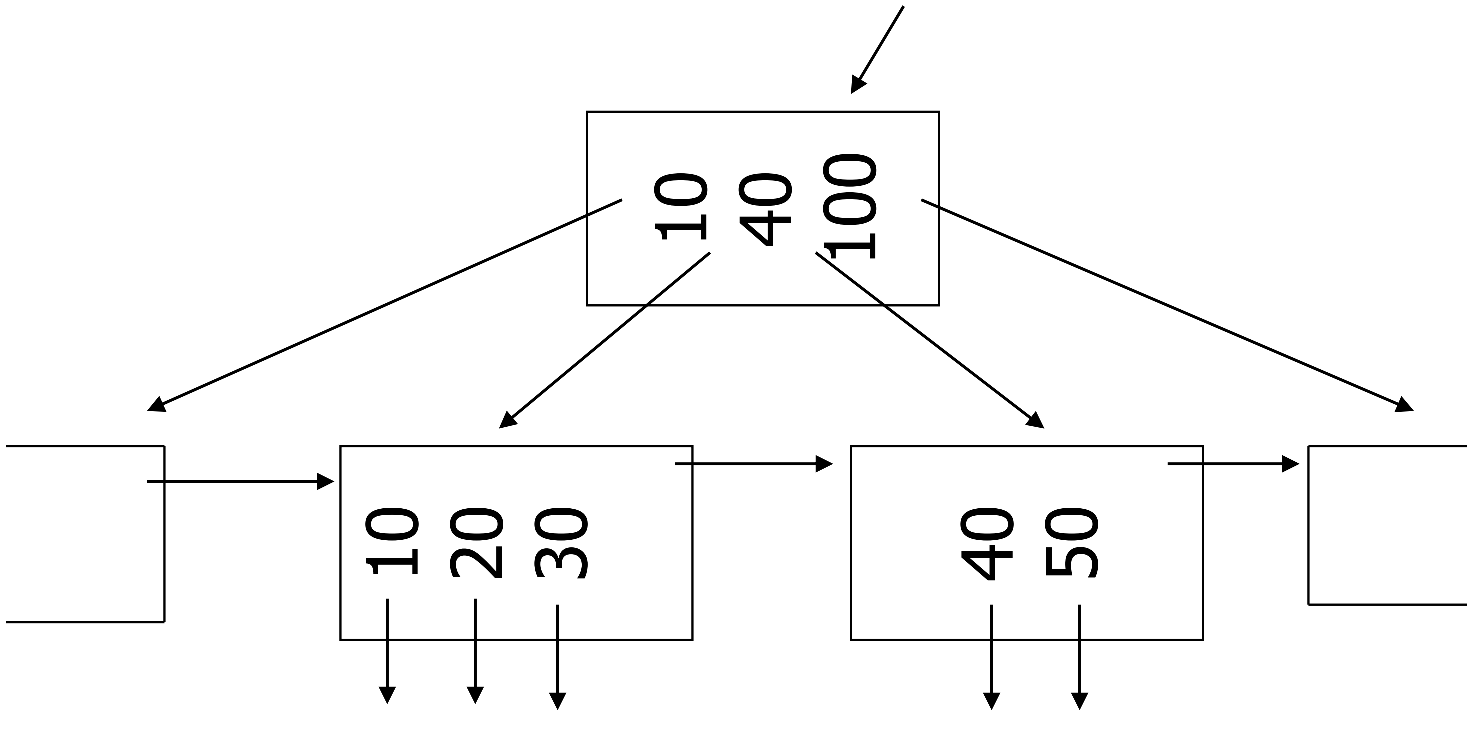
we only mark the values as deleted and go next

when there are to many fragments we relabel the tree

(b) Coalesce with sibling

– Delete 50

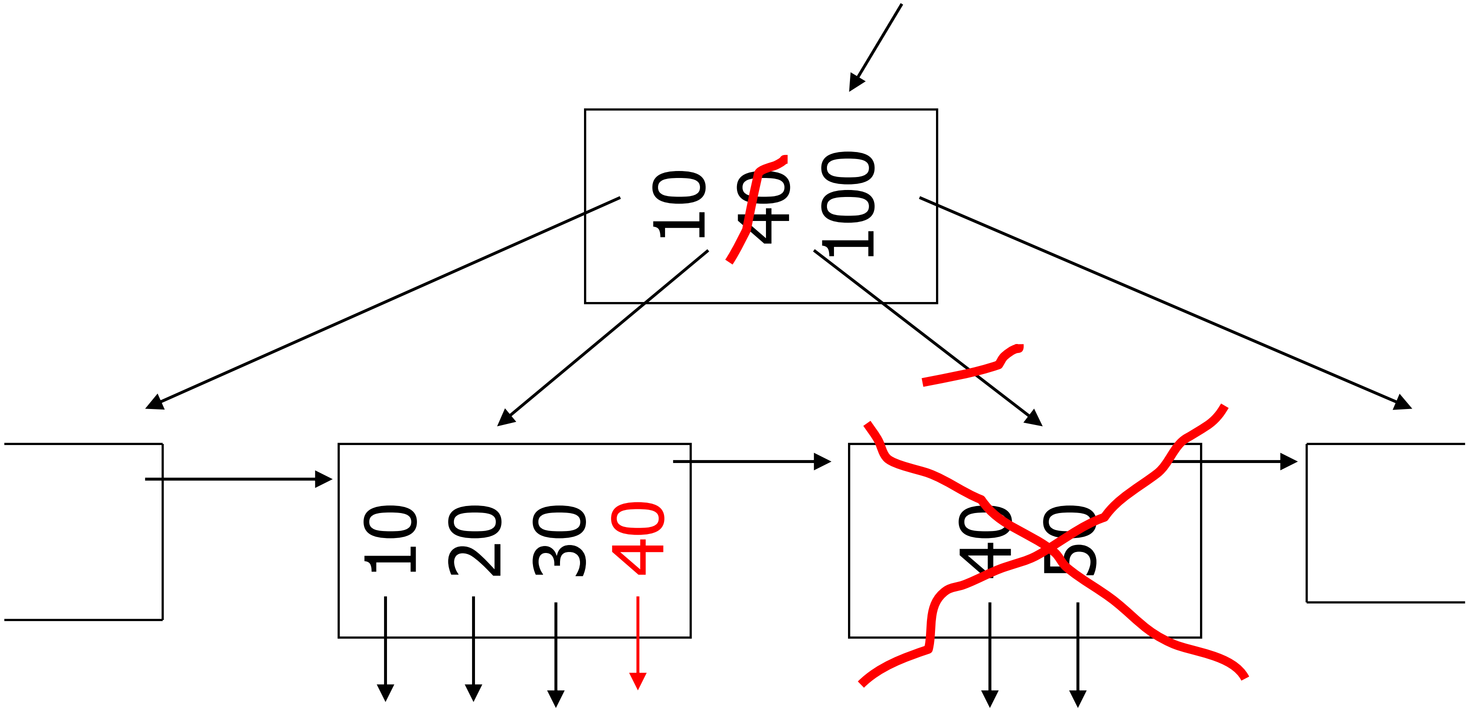
n=4



(b) Coalesce with sibling

- Delete 50

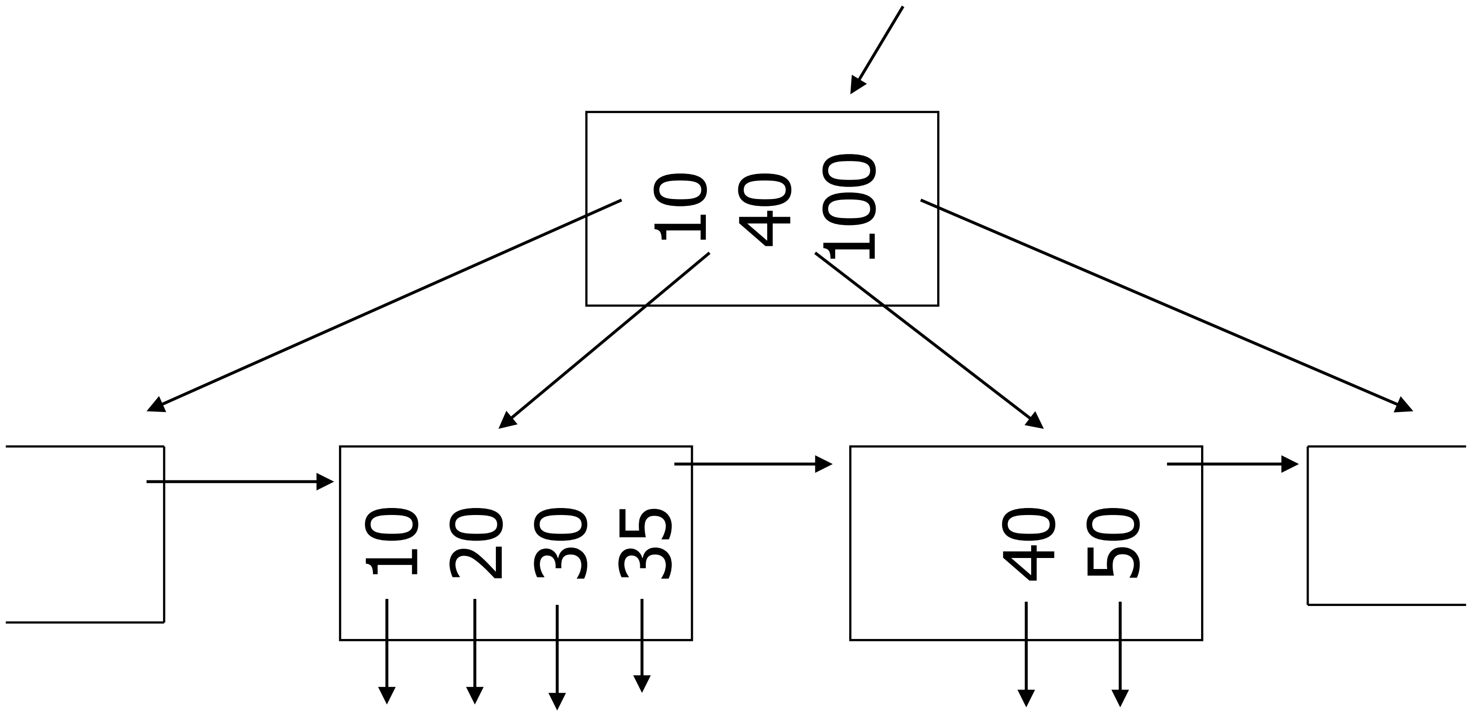
n=4



(c) Redistribute keys

– Delete 50

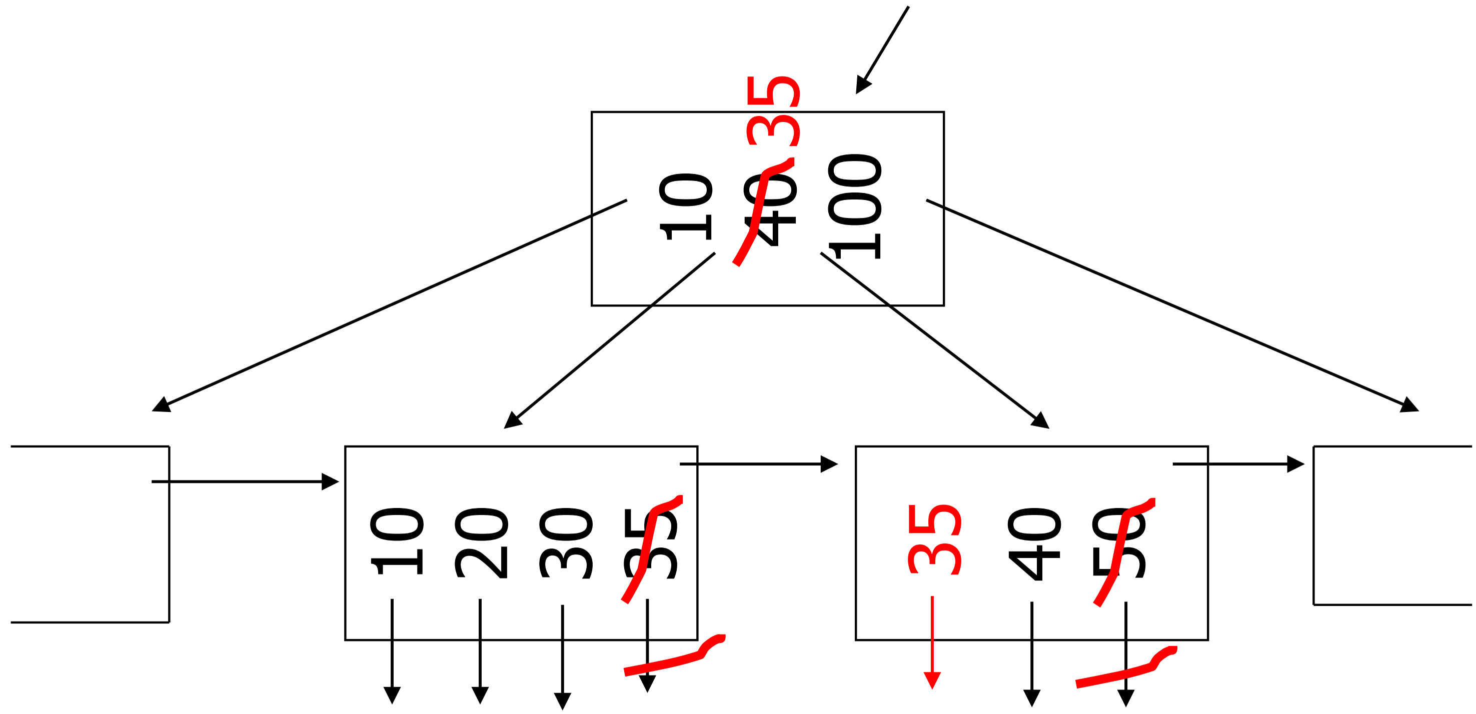
n=4



(c) Redistribute keys

– Delete 50

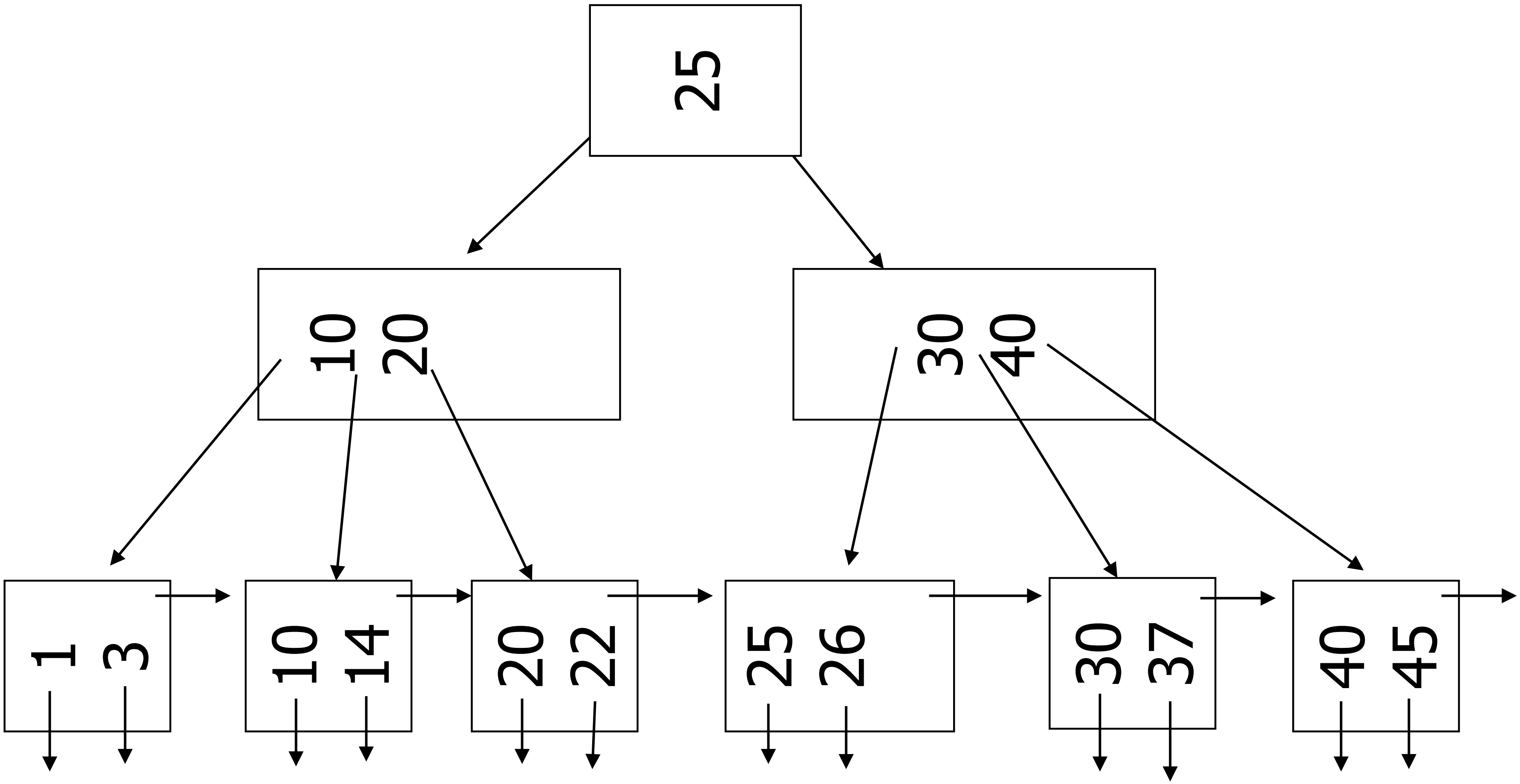
n=4



(d) Non-leaf coalesce

– Delete 37

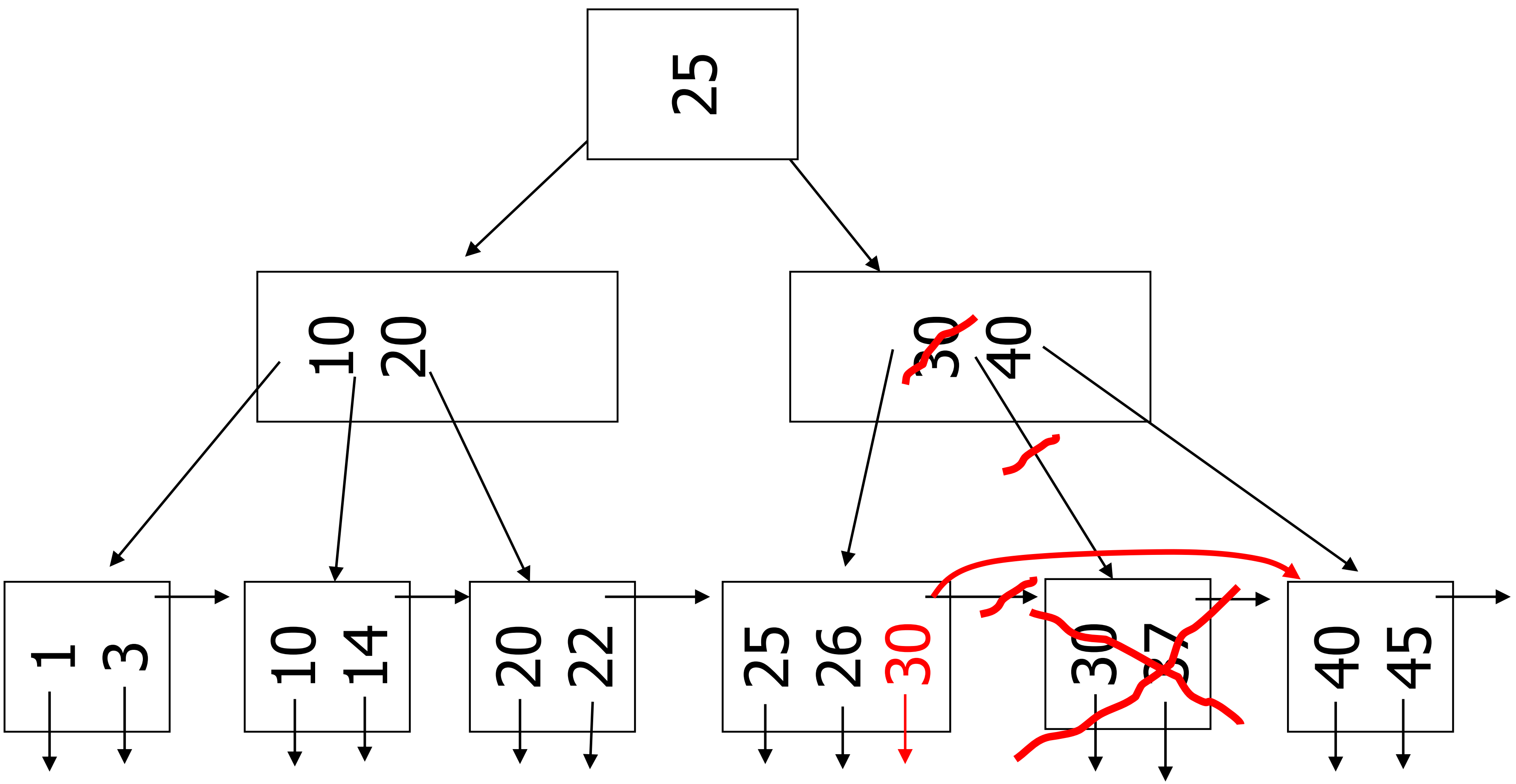
n=4



(d) Non-leaf coalesce

- Delete 37

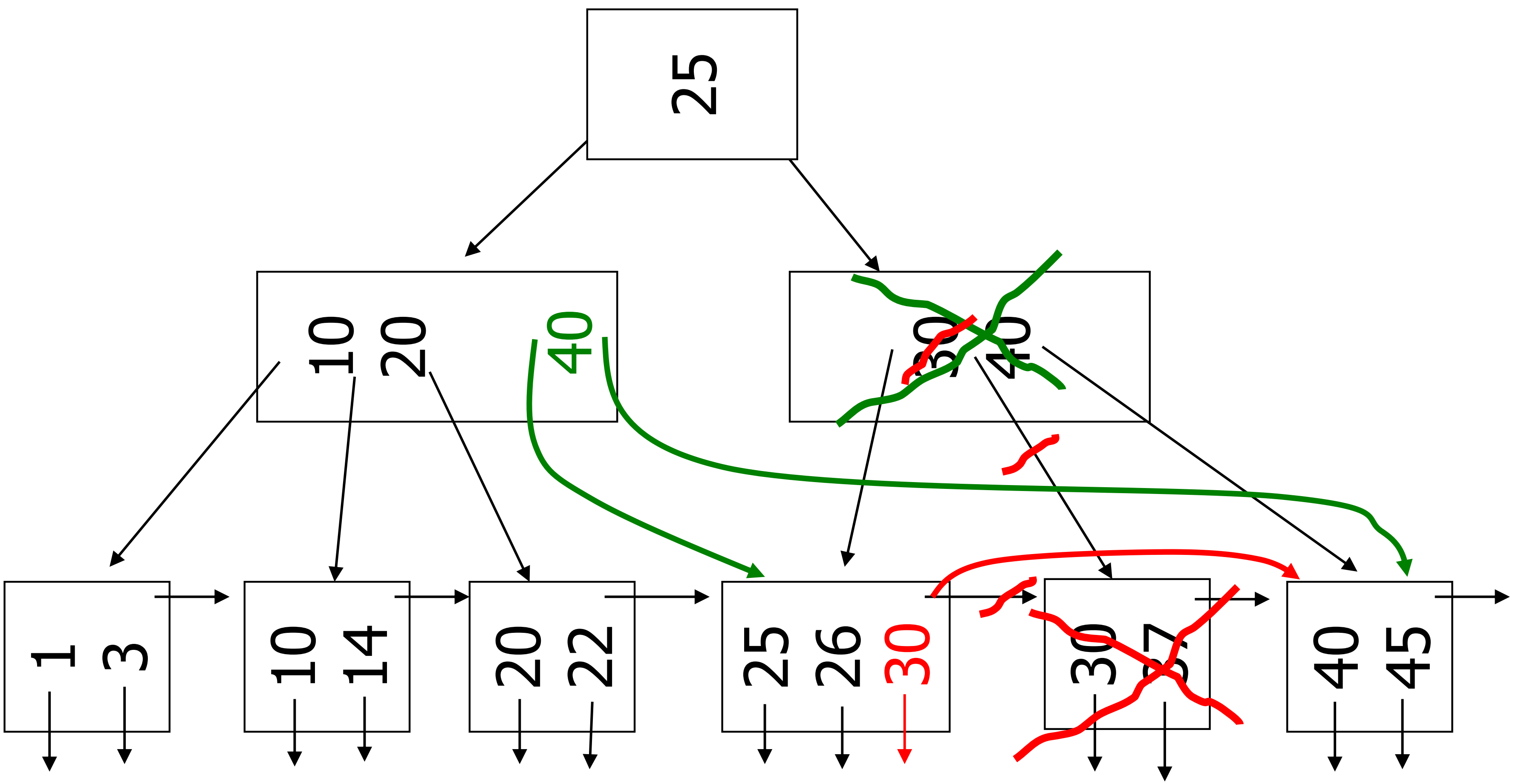
n=4



(d) Non-leaf coalesce

– Delete 37

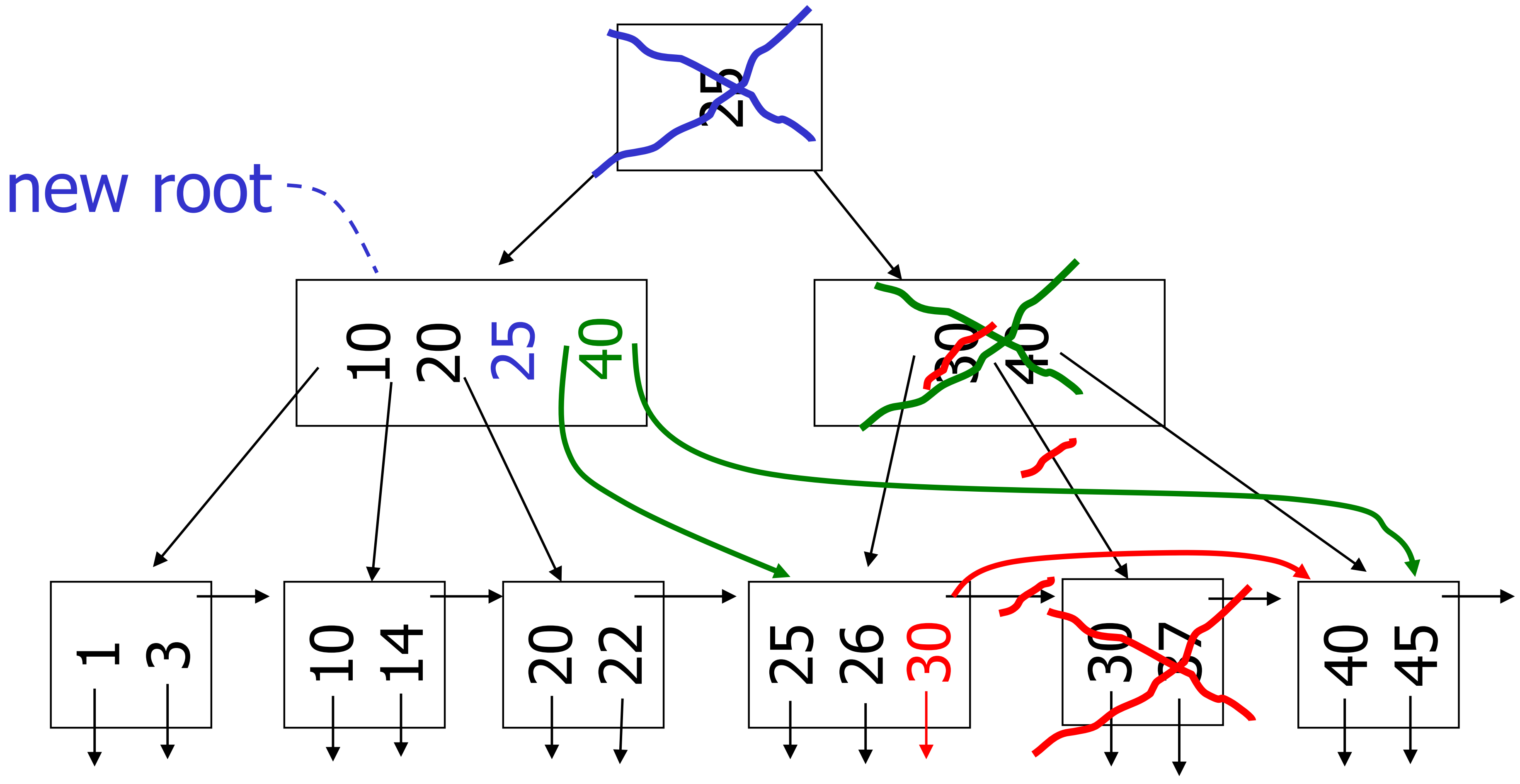
n=4



(d) Non-leaf coalesce

– Delete 37

n=4



B+tree deletions in practice

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

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