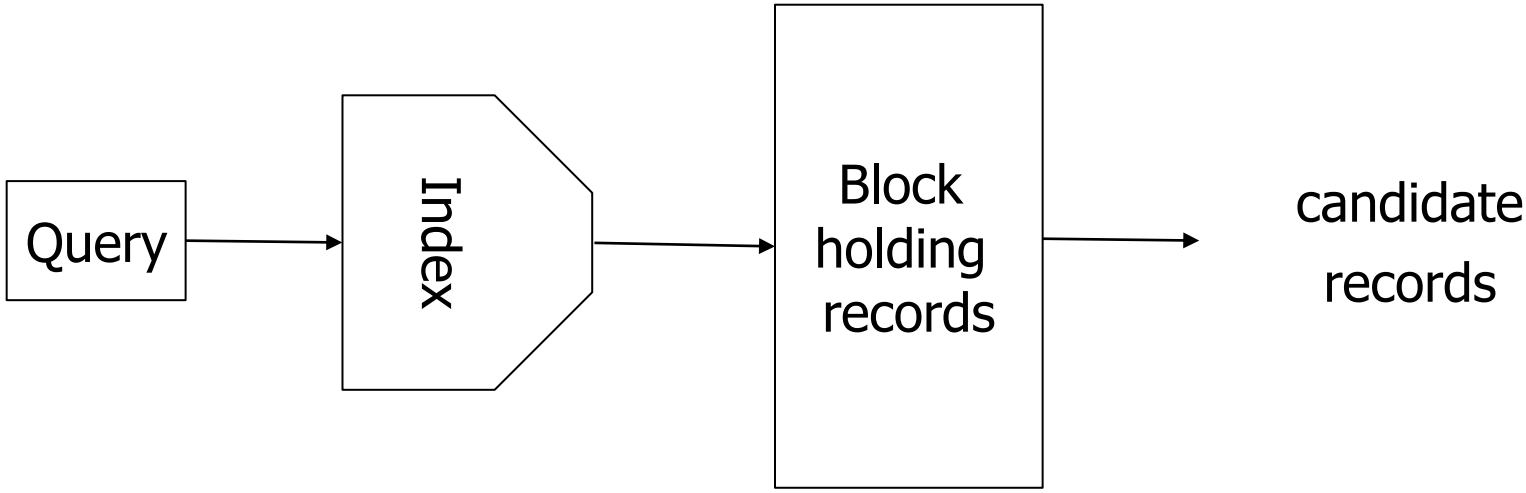
# Indexing

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

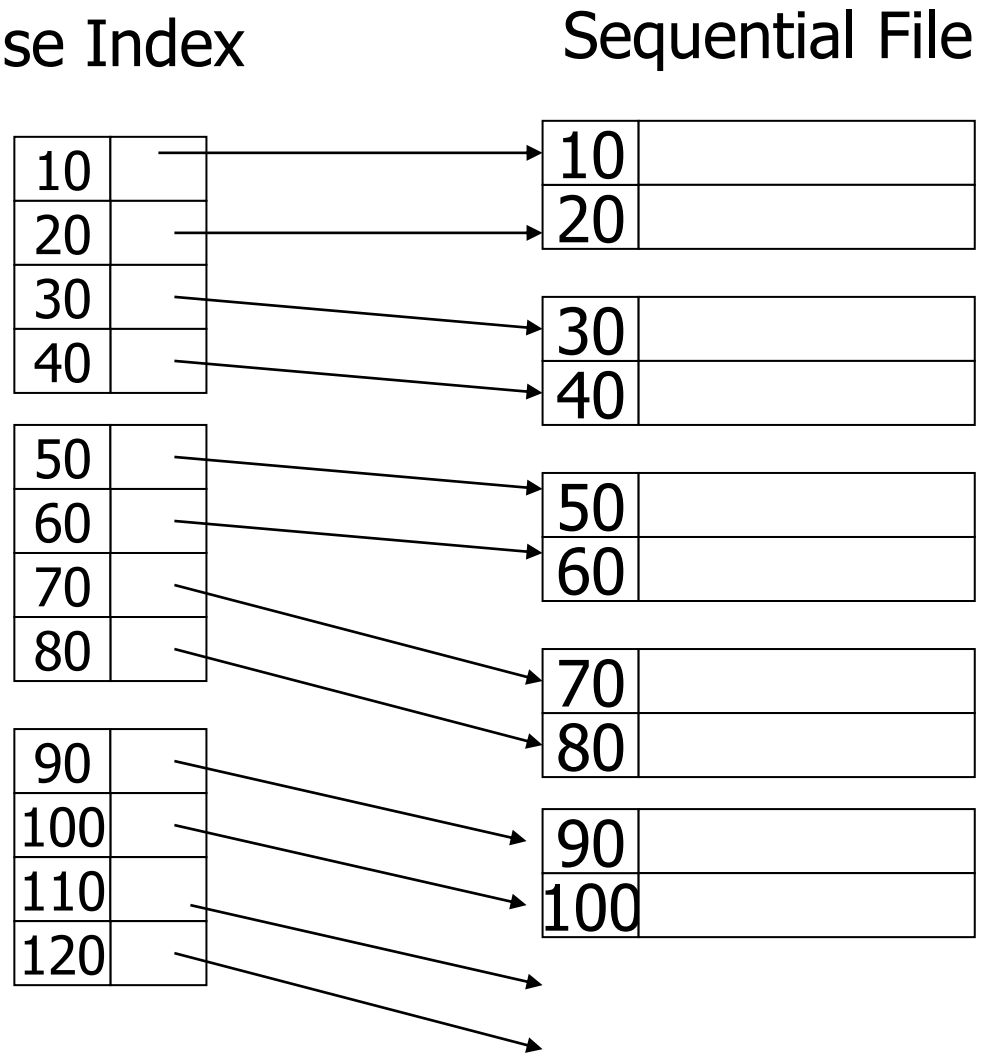
How to search for a key= 30 ?

How to search for a key= 25 ?

go to 20 then go to 25

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

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




page = 4kB

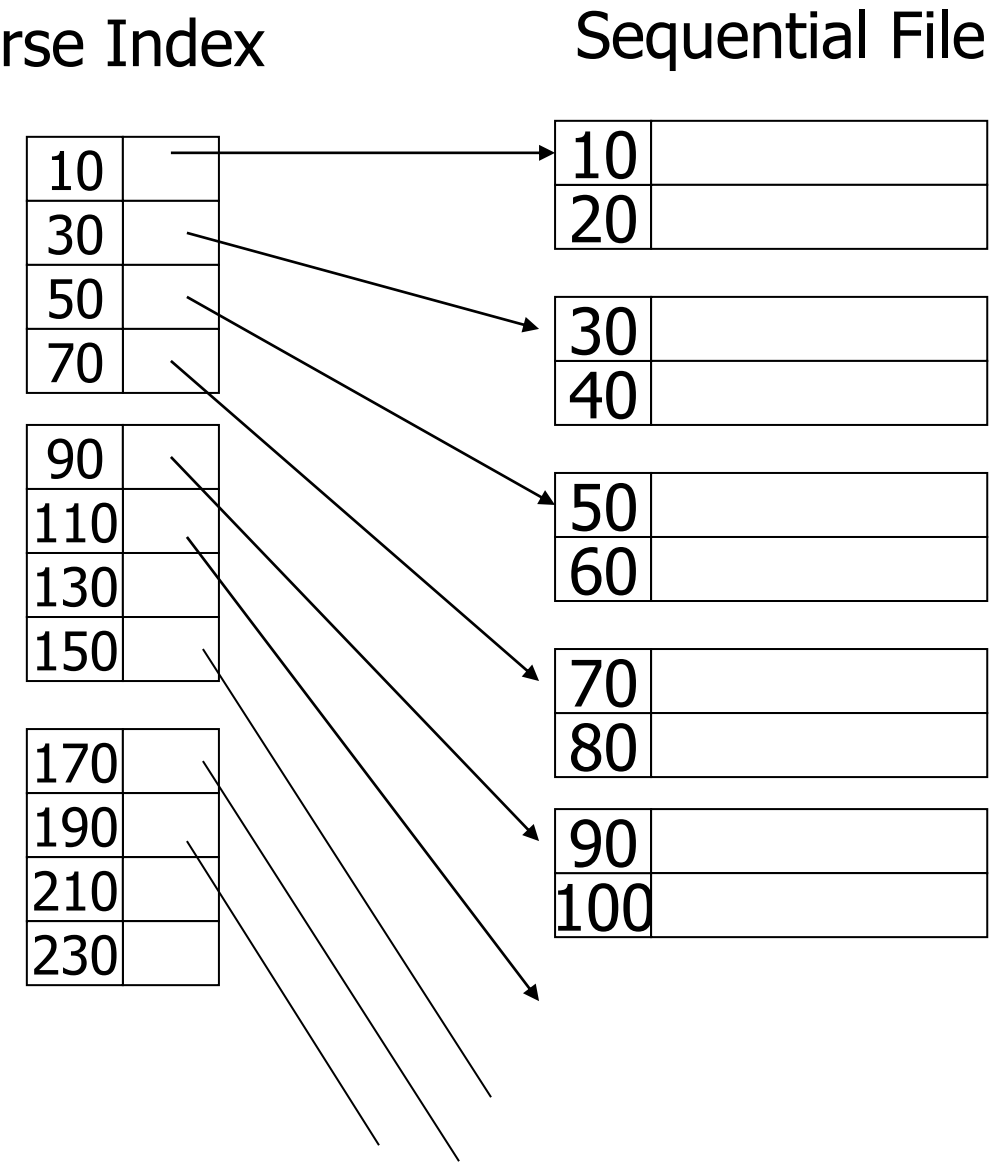
indexing par group

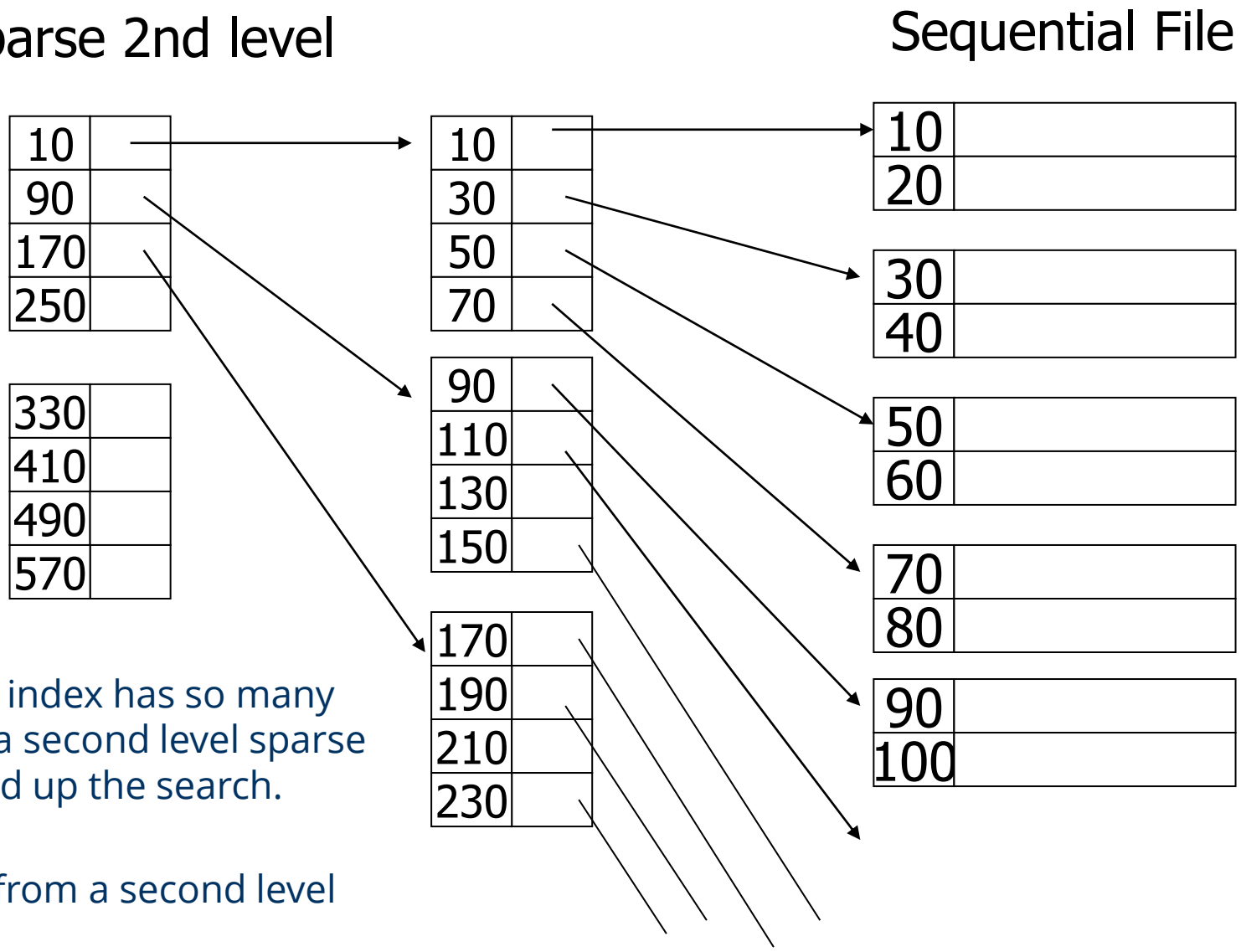
Sparse index = a pointer per block

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 ?





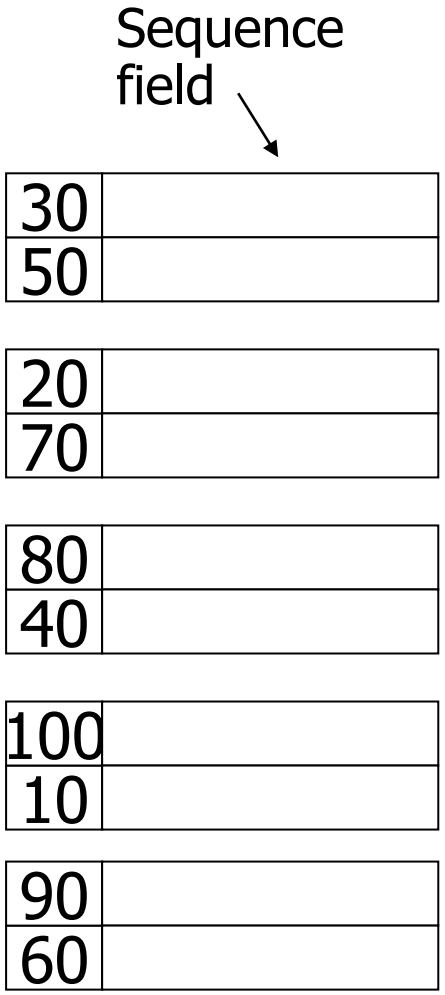

If the first level index has so many pages, adding a second level sparse index can speed up the search.

Do we benefit from a second level dense index ?

# Sparse vs. Dense Tradeoff

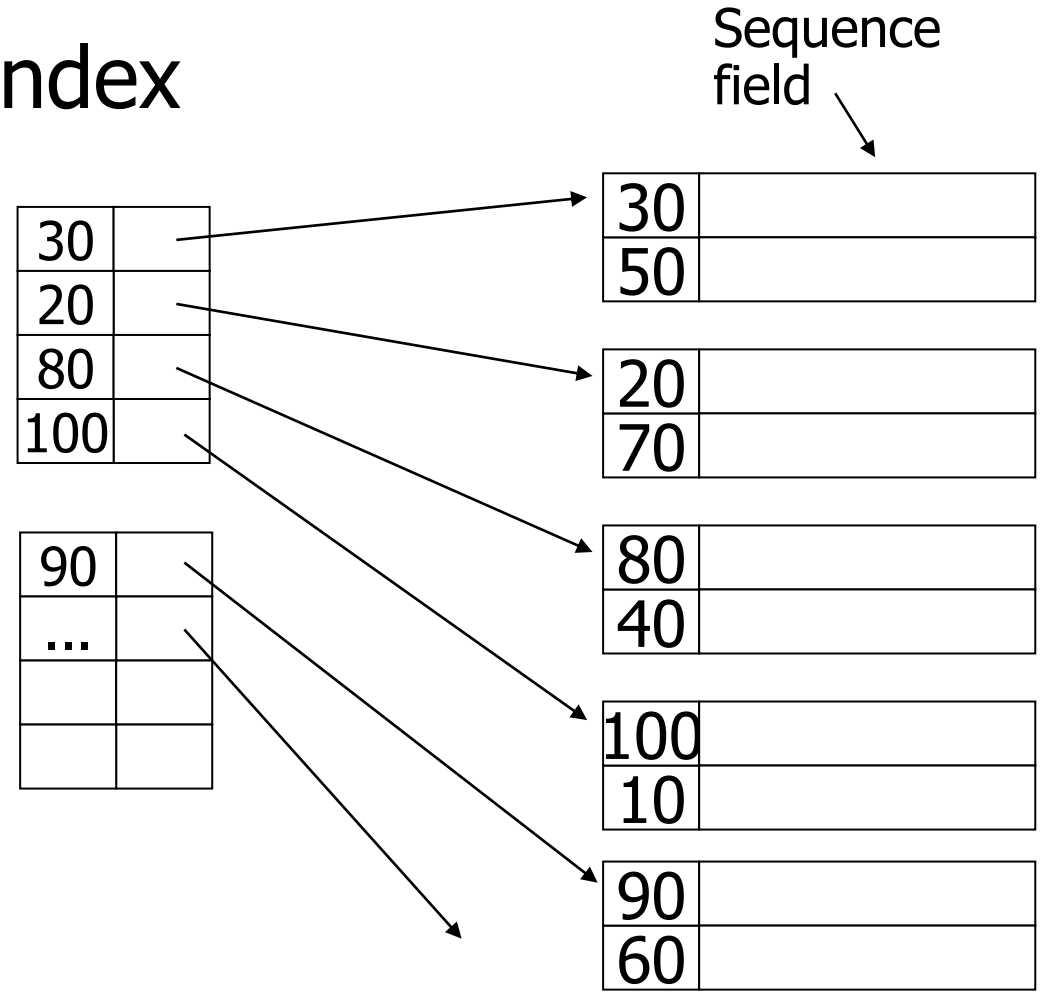
- Sparse: Less index space per record
  can keep more of index in memory
- Dense: Can tell if any record exists
  without accessing file

# Secondary indexes



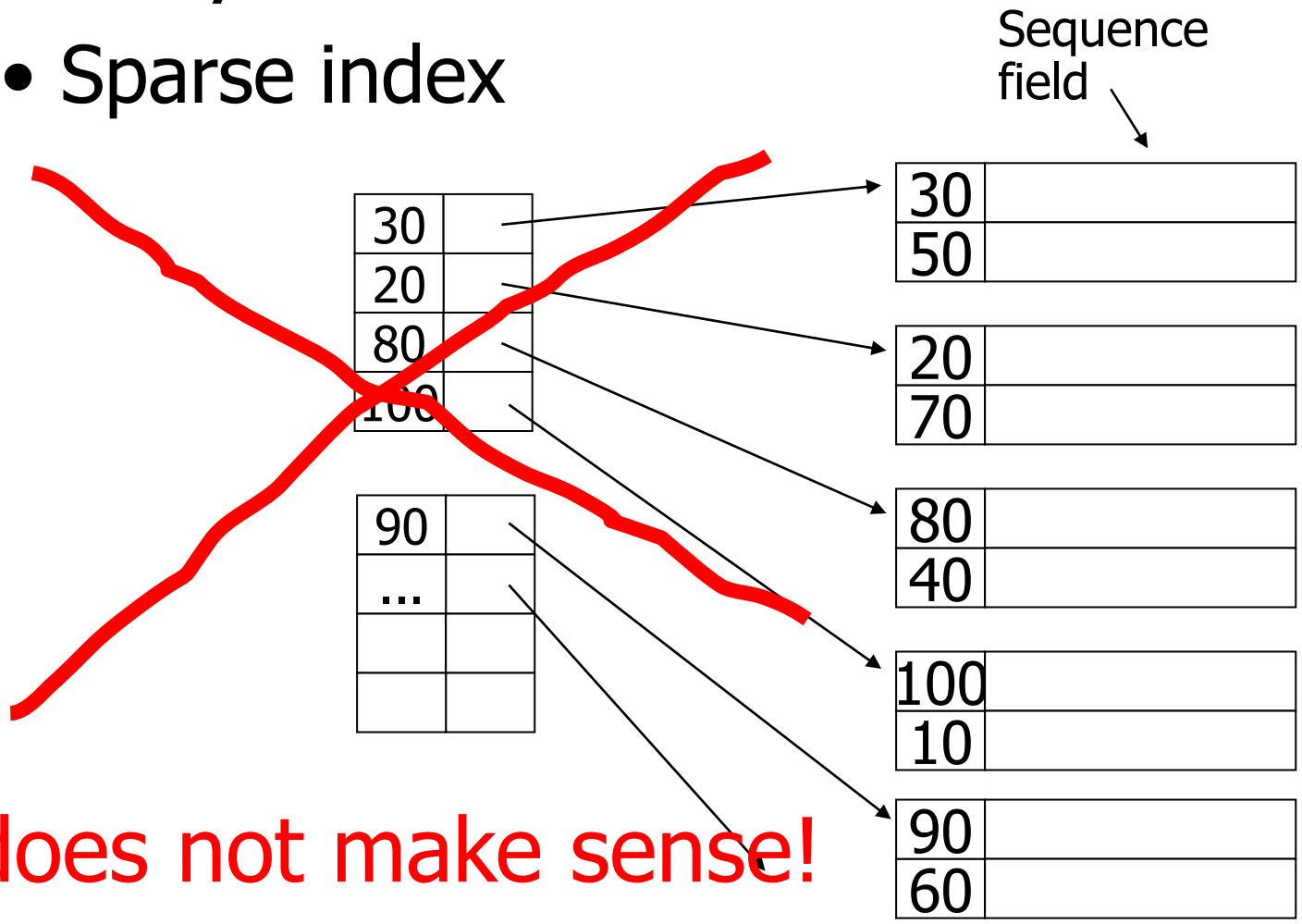
# Secondary indexes

- Sparse index



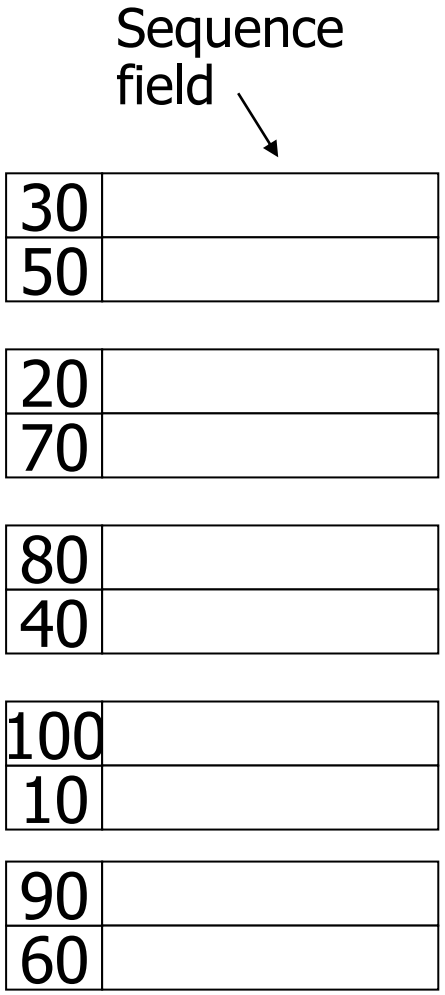
# Secondary indexes

- Sparse index



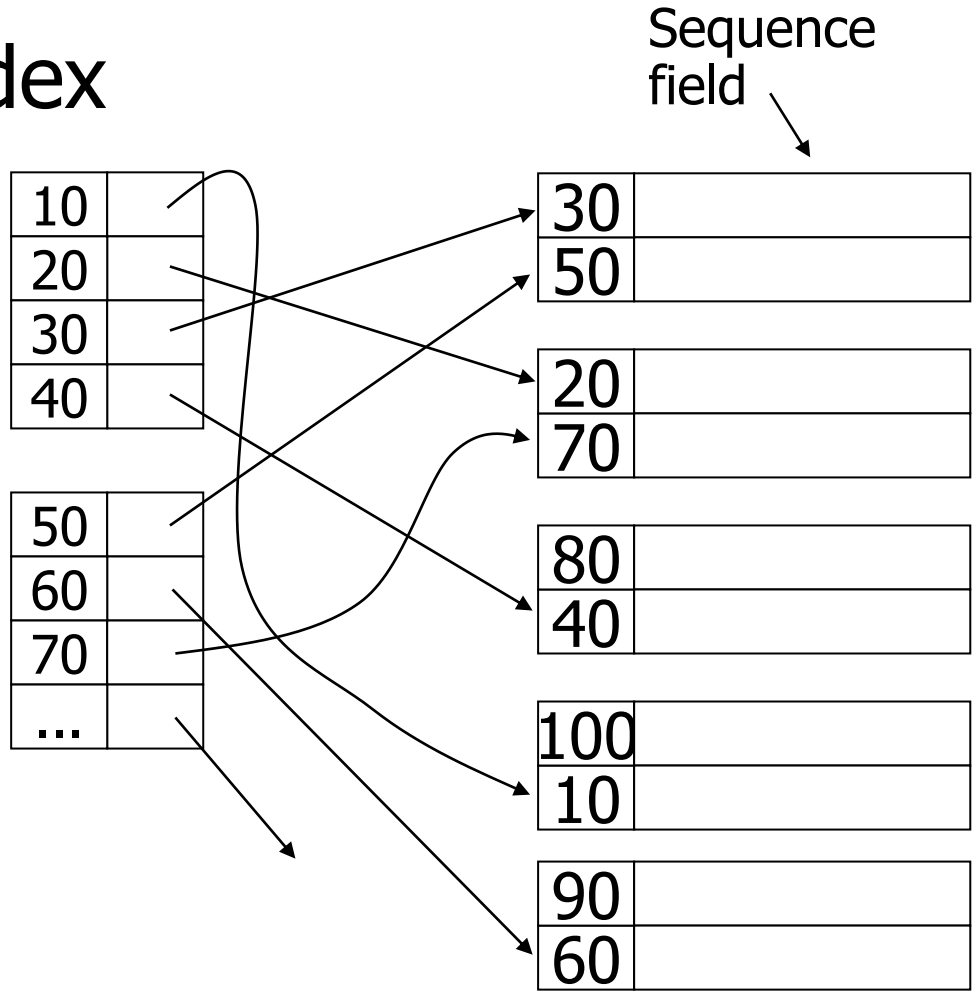
# Secondary indexes

- Dense index



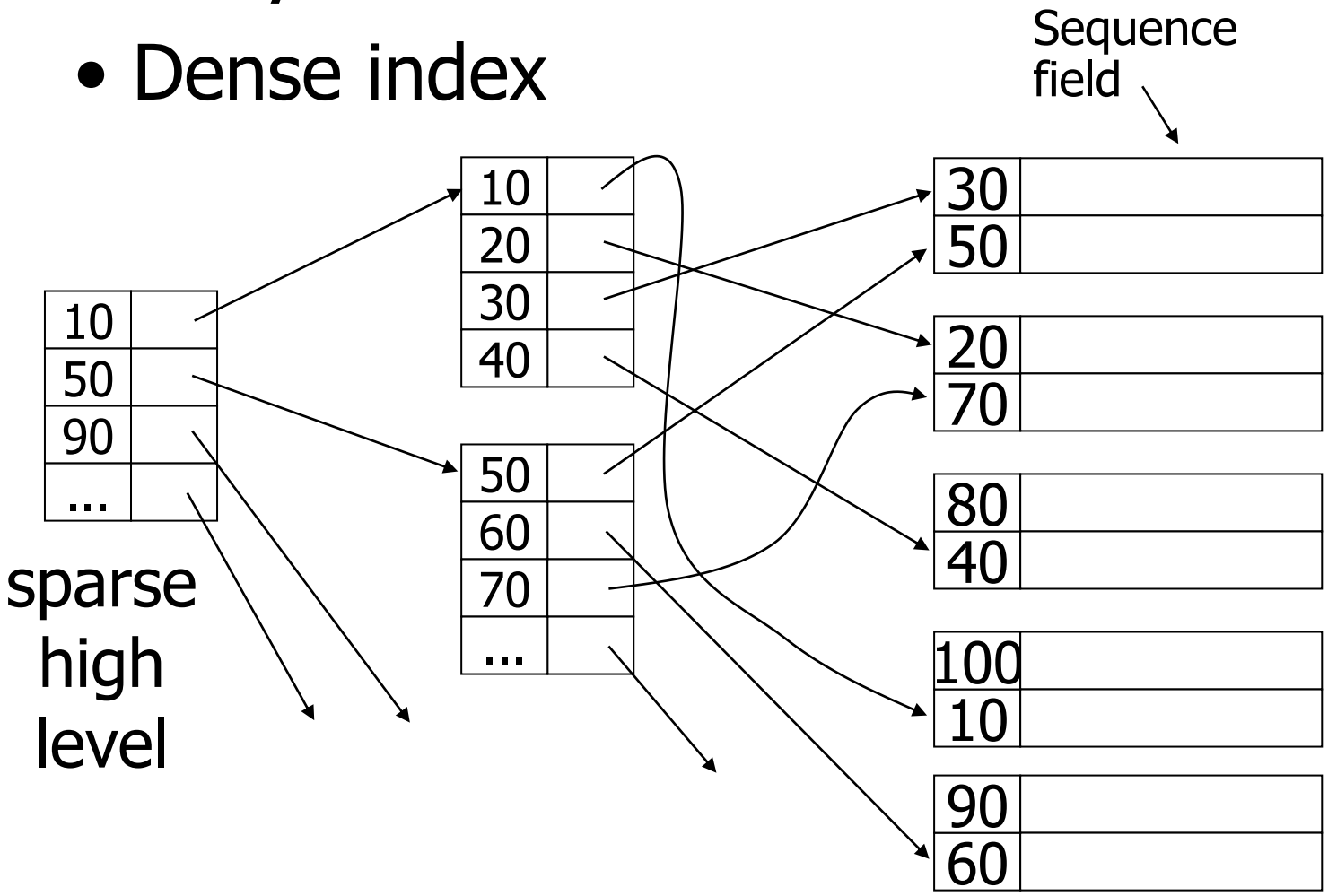
# Secondary indexes

- 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 | |

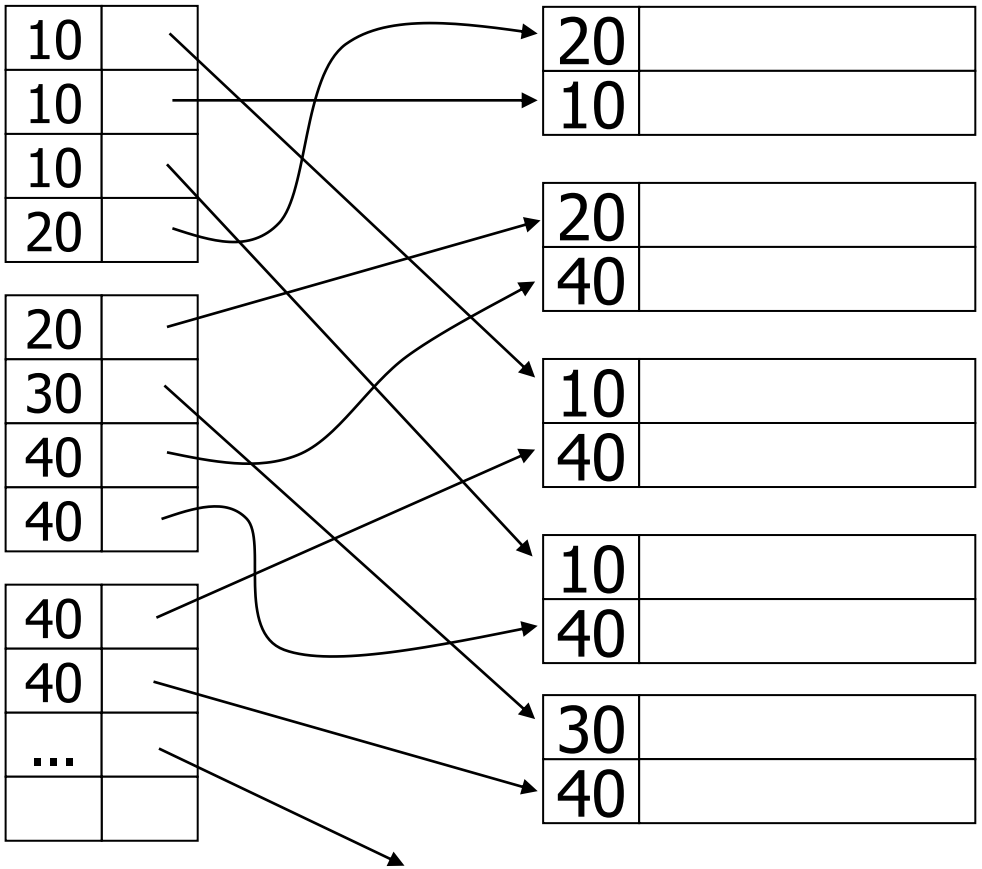
| | |
|---|---|
| 10 | |
| 40 | |

| | |
|---|---|
| 10 | |
| 40 | |

| | |
|---|---|
| 30 | |
| 40 | |

# Duplicate values & secondary indexes

one option...



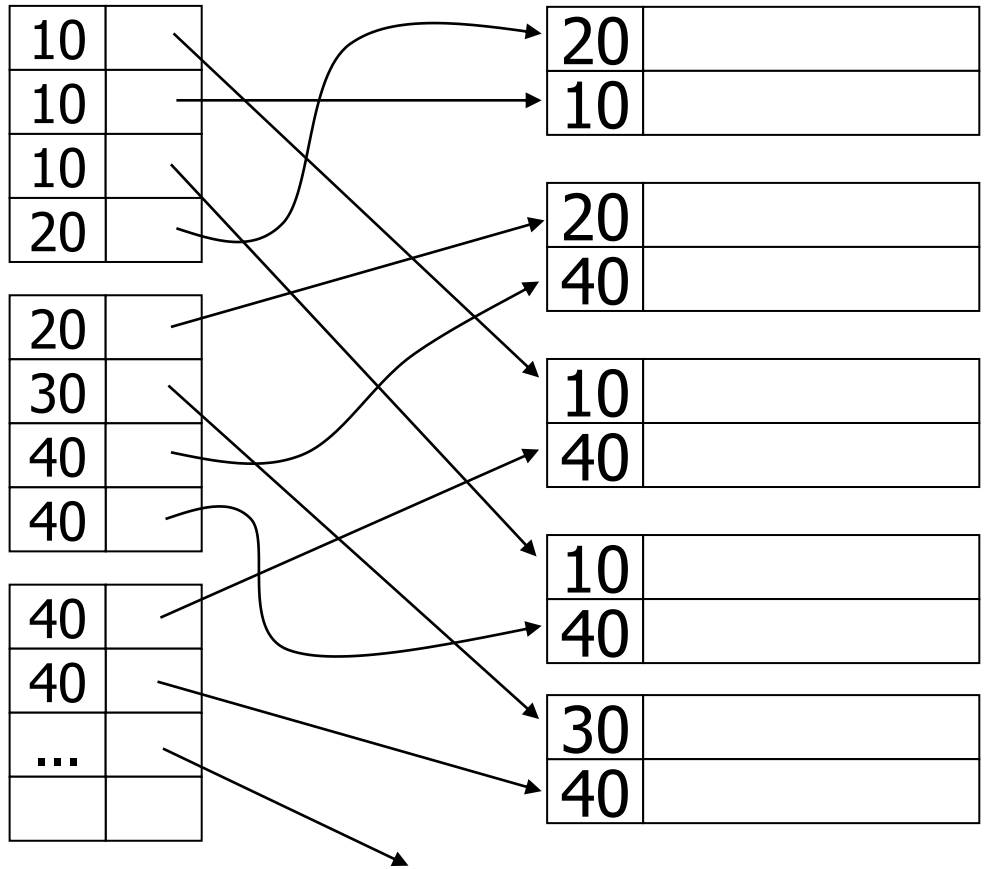
# Duplicate values & secondary indexes

one option...

Problem:

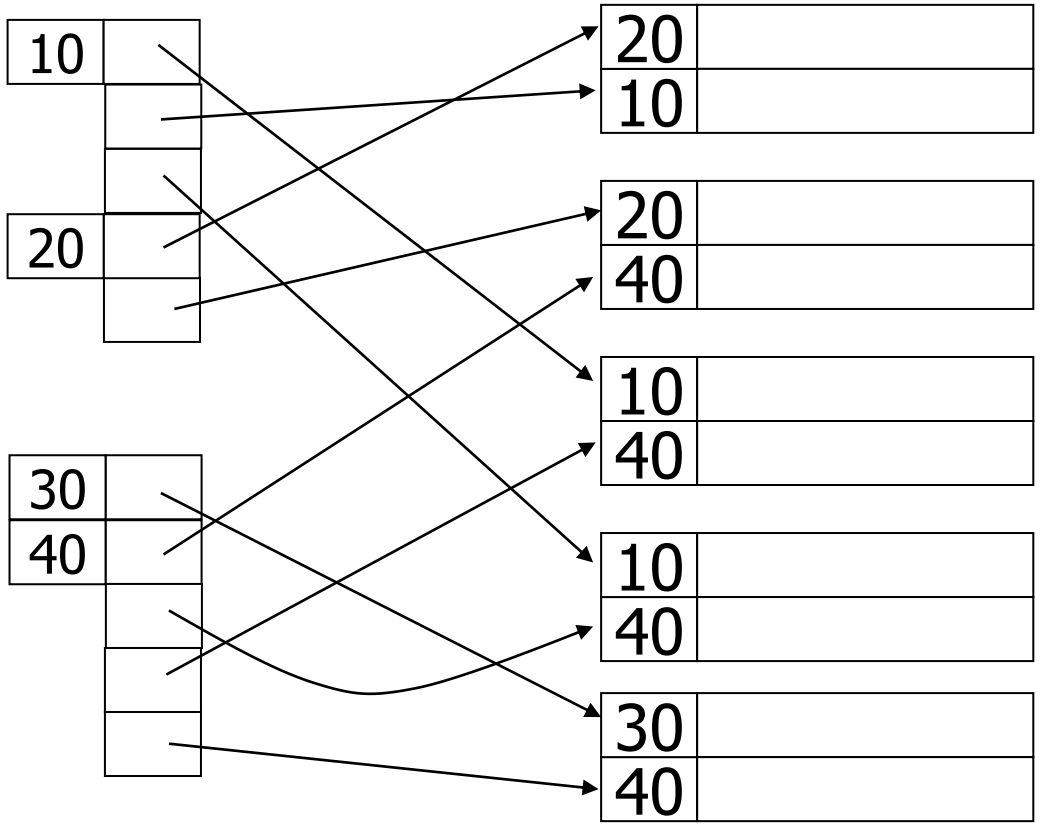
excess overhead!

- disk space
- search time



# Duplicate values & secondary indexes

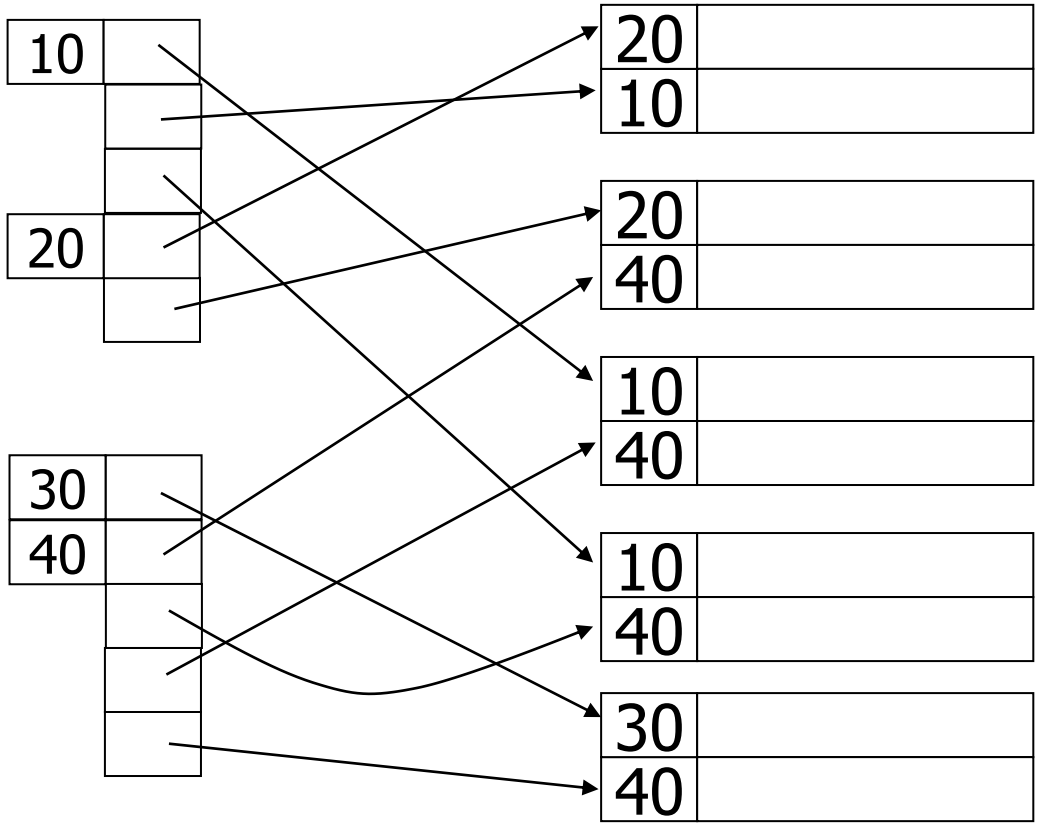
another option...



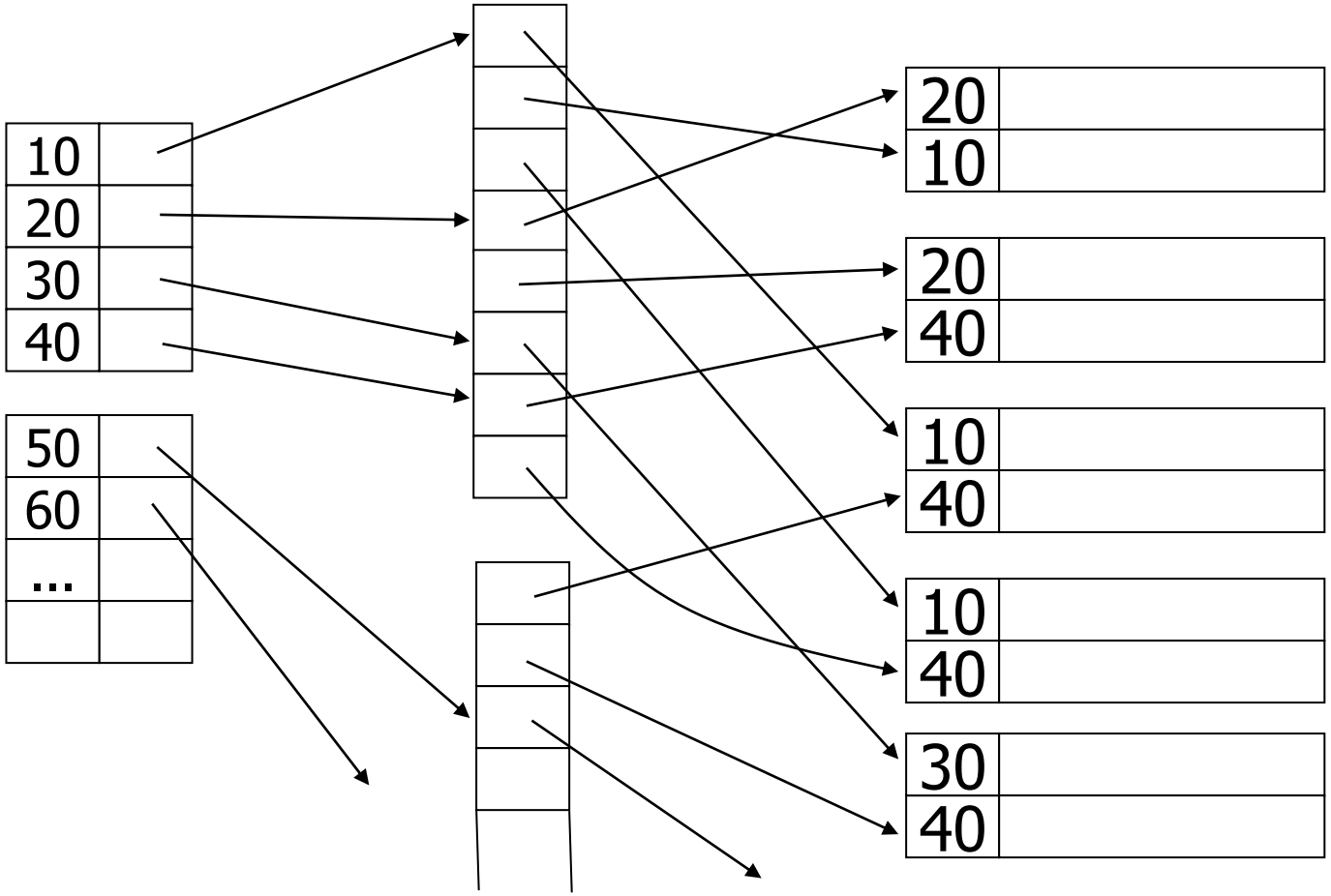
# Duplicate values & secondary indexes

another option...

Problem:
variable size
records in
index!



# Duplicate values & secondary indexes

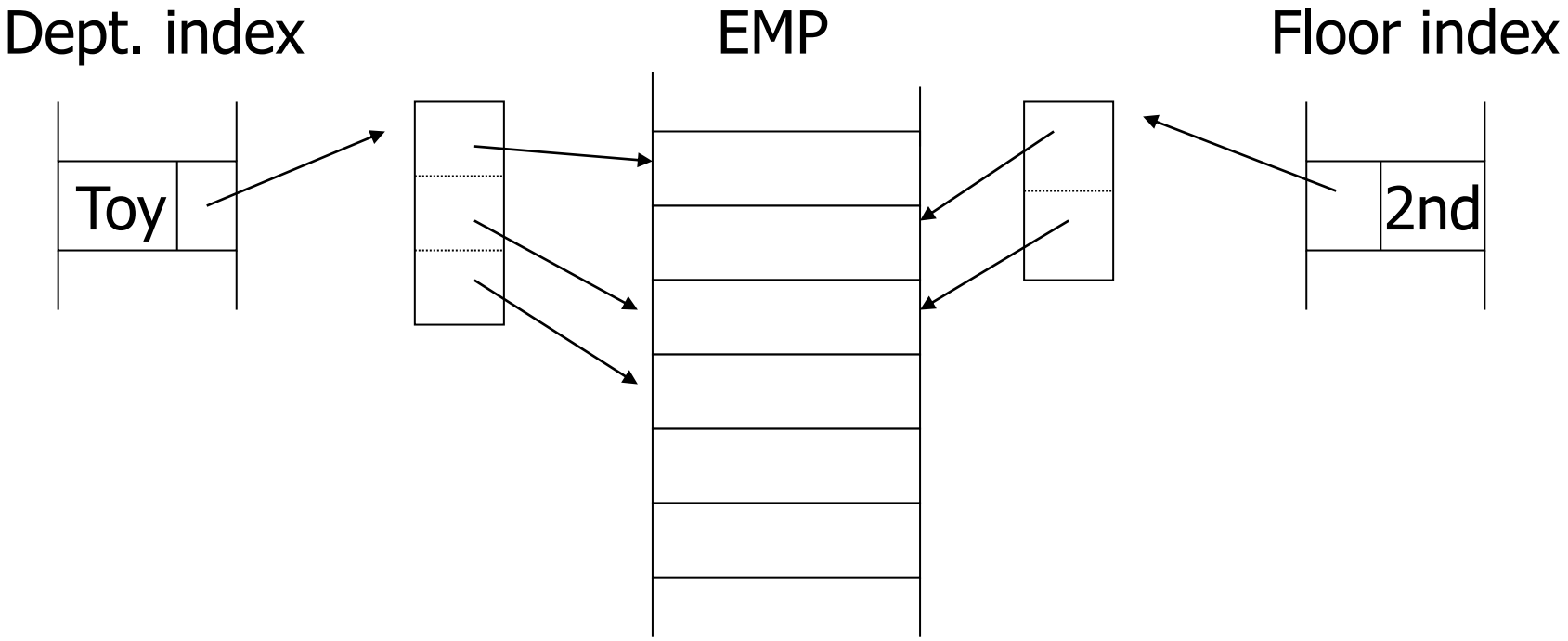


# Why "bucket" idea is useful

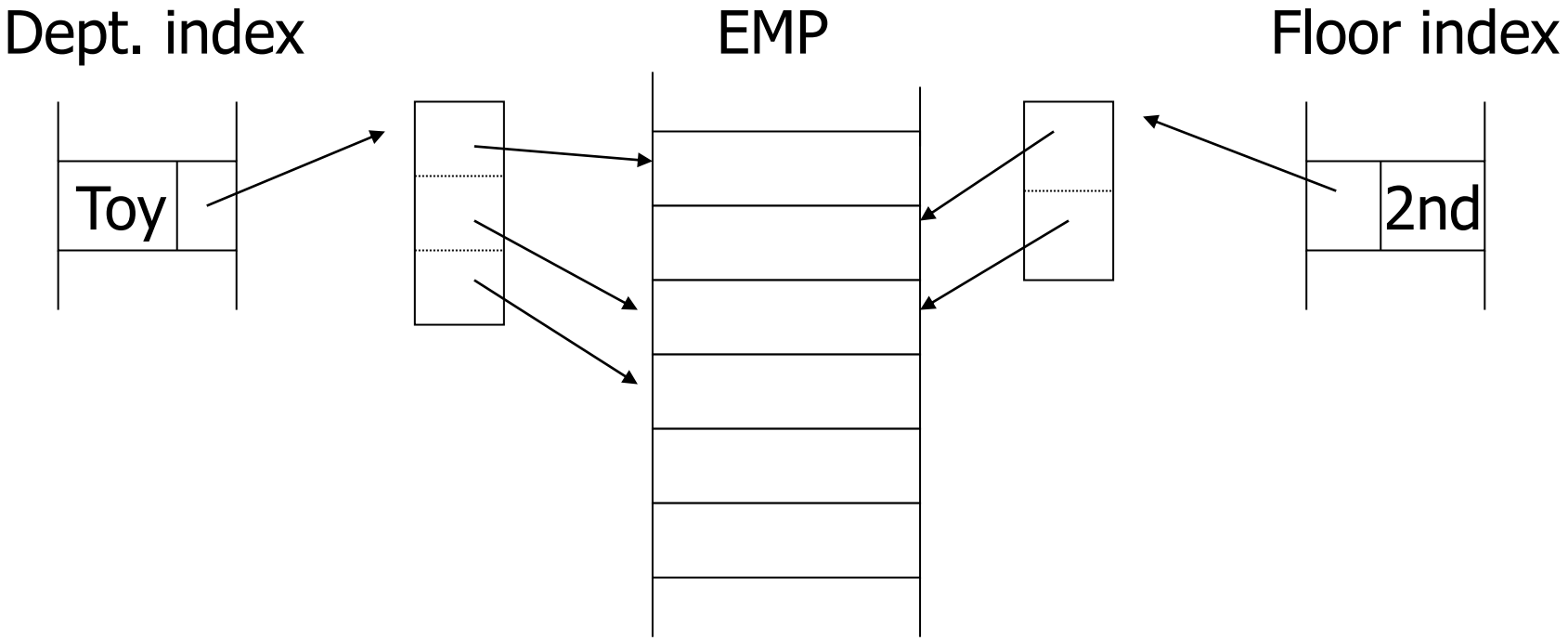
| Indexes | Records |
| --- | --- |
| Name: primary | EMP (name,dept,floor,...) |
| Dept: secondary | |
| Floor: secondary | |

EMP = values

# 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

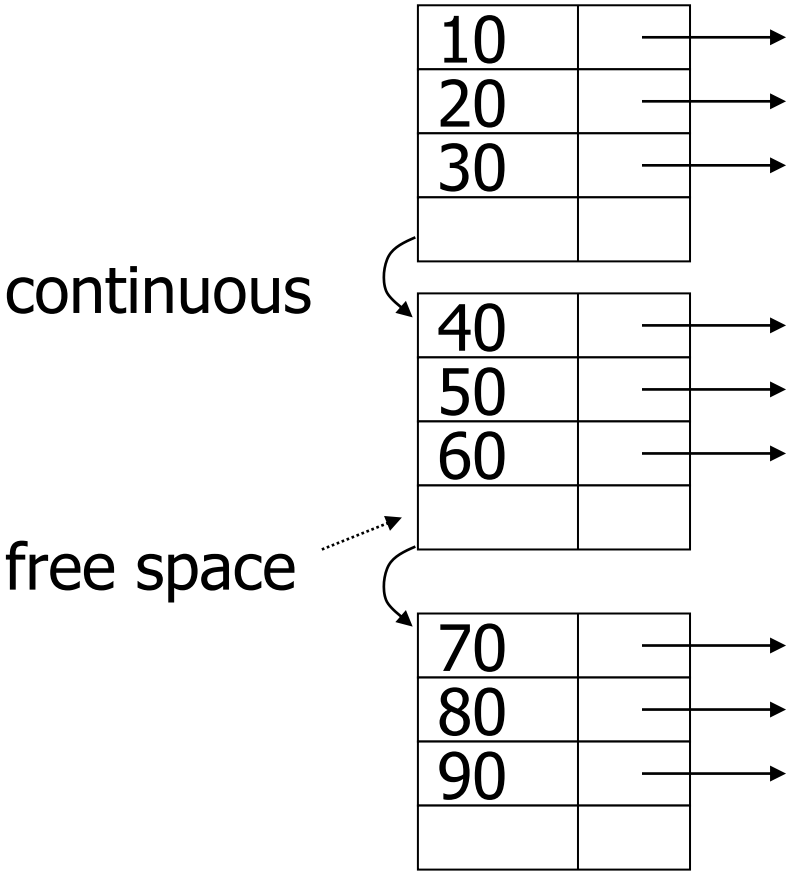
## Advantage:

- Simple
- Index is sequential file
  good for scans

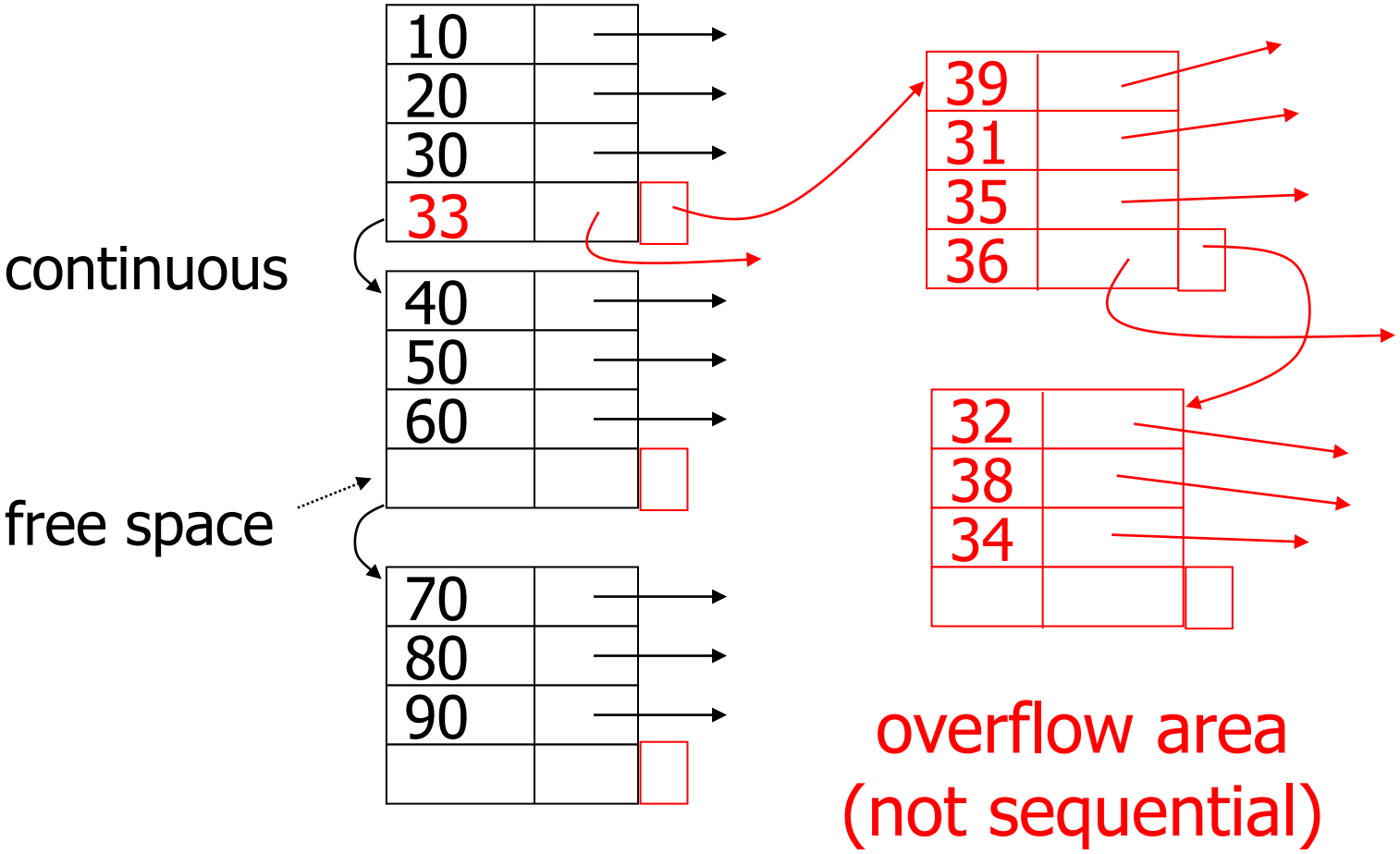
## Disadvantage:

- Inserts expensive, or
- Lose sequentiality & balance

# Example Index (sequential)



# Example Index (sequential)



# Outline:

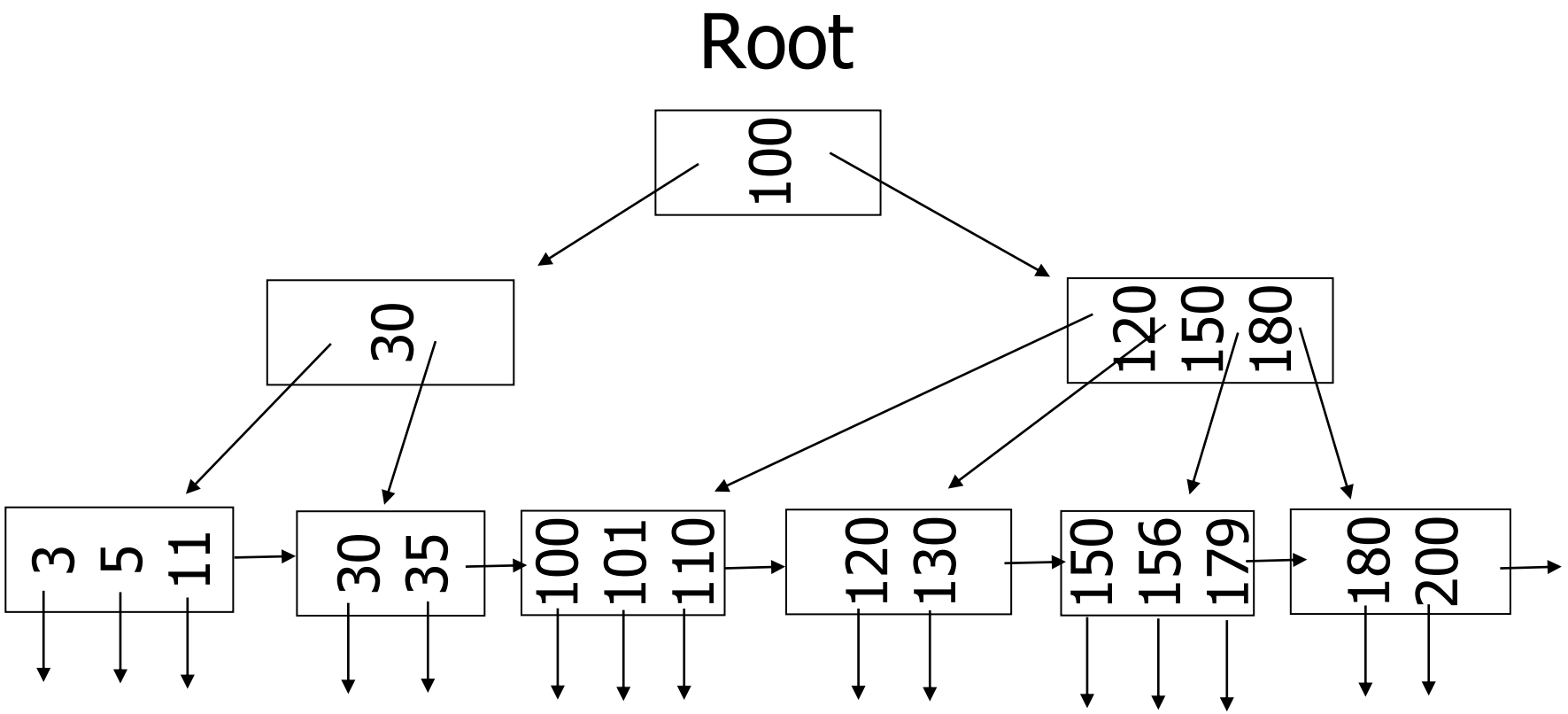
- Conventional indexes
- B-Trees $\Rightarrow$ NEXT
- Hashing schemes

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

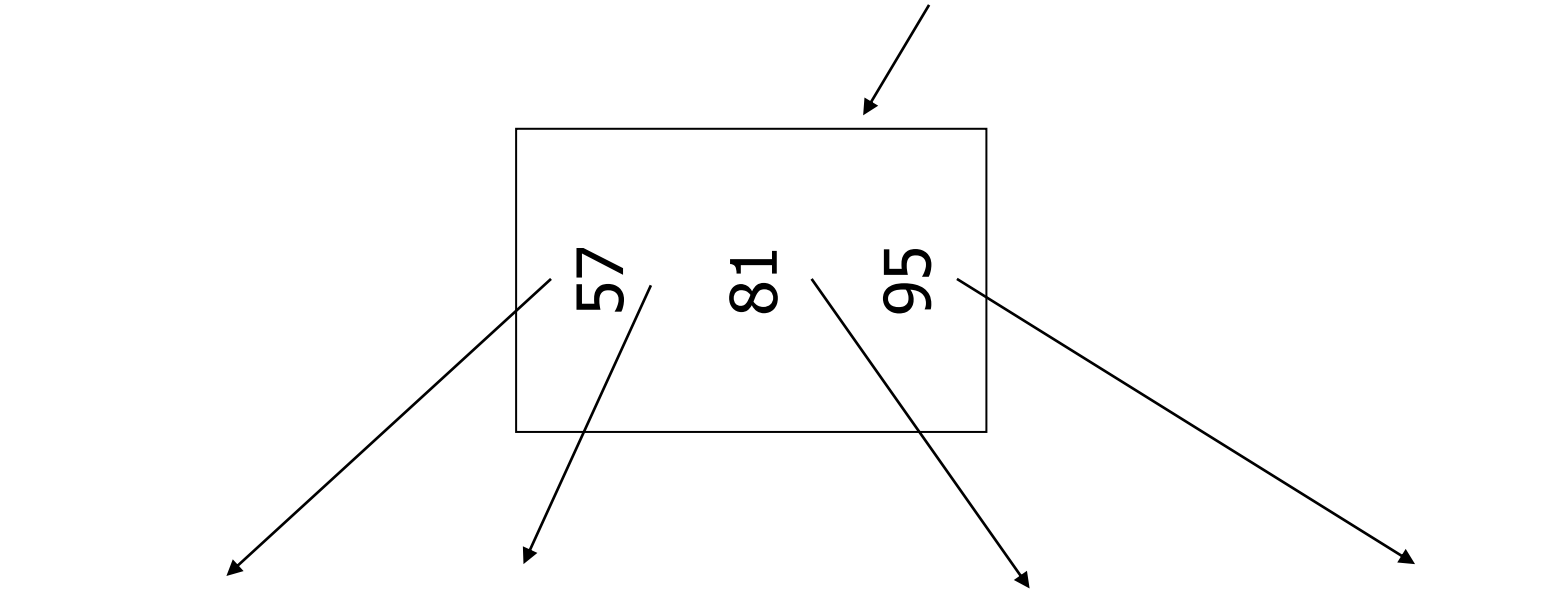
# B+Tree Example

n=3

a sort of linked tree



# Sample non-leaf

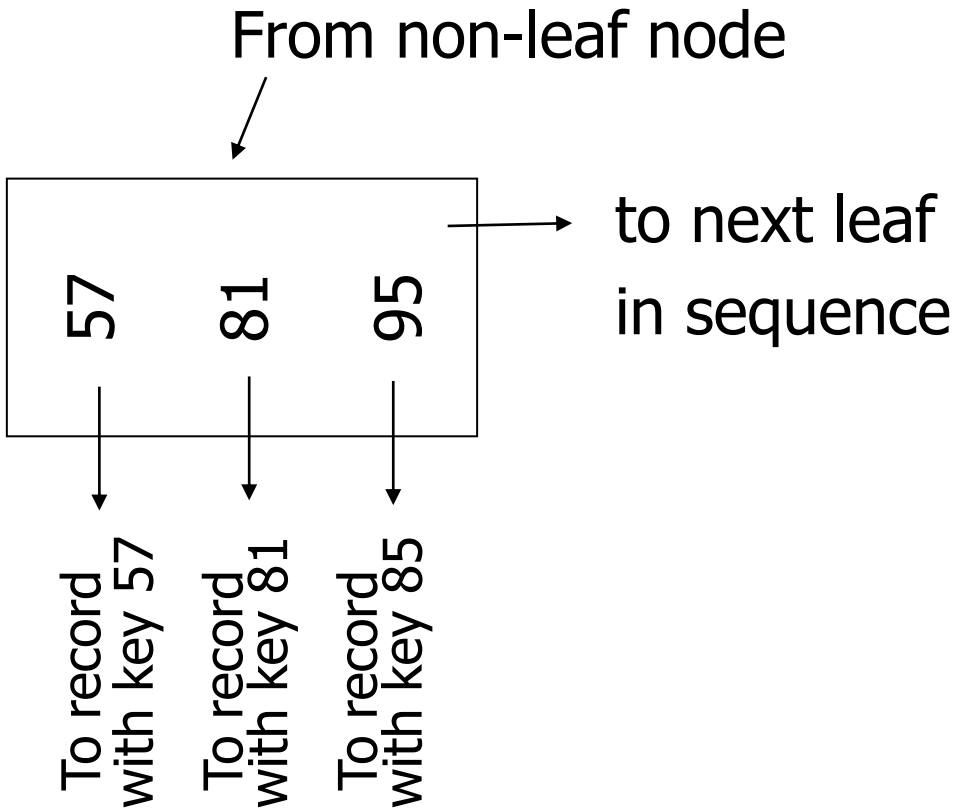



| to keys | to keys | to keys | to keys |
|---|---|---|---|
| < 57 | 57≤ k<81 | 81≤k<95 | ≥95 |

Pas # en bas

# Sample leaf node:

tout en bas de l'arbre



n is determined by the page size

Size of nodes:
- n+1 pointers
- n keys

(fixed)

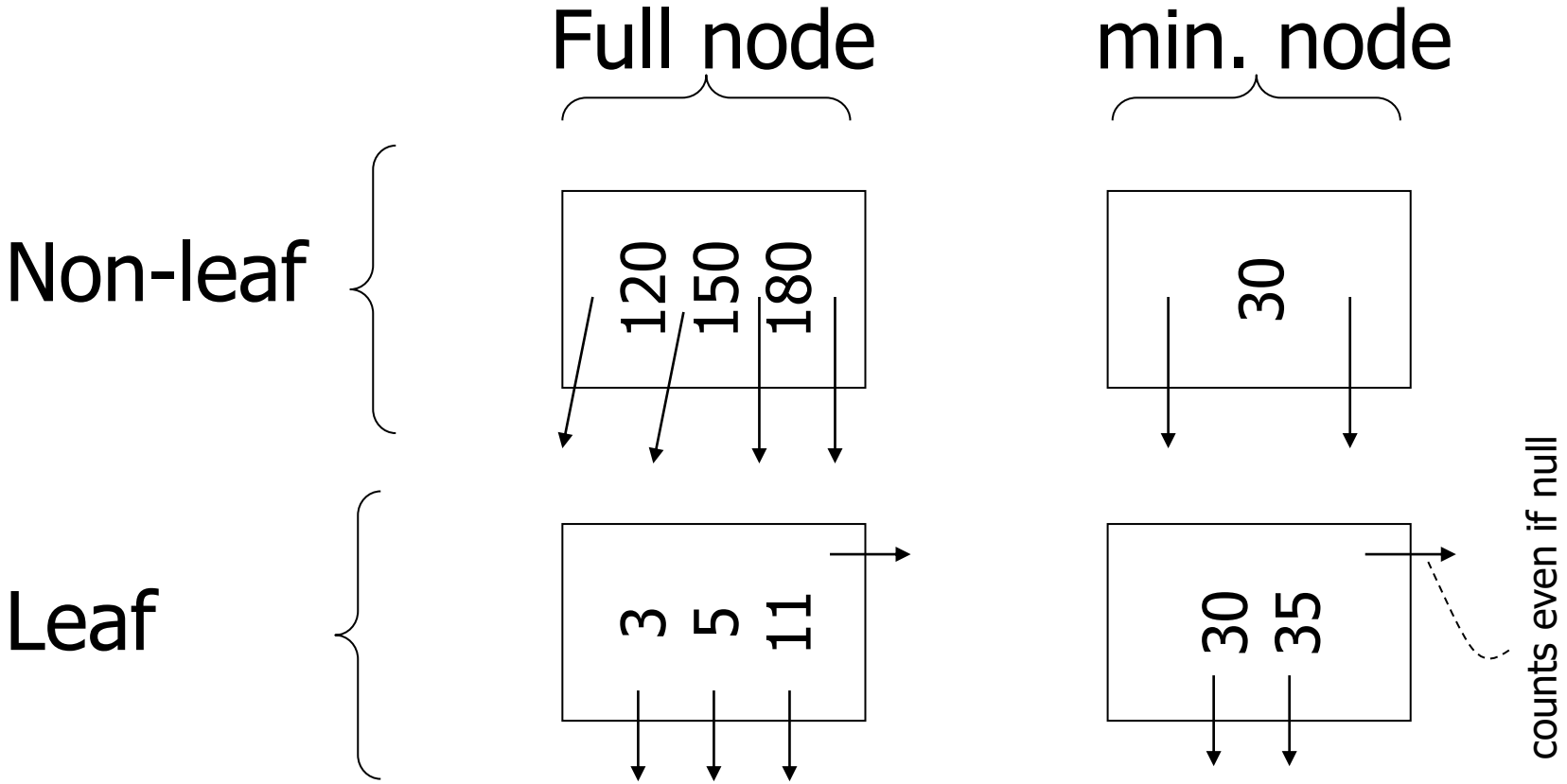
# Don’t want nodes to be too empty

- 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"

# (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

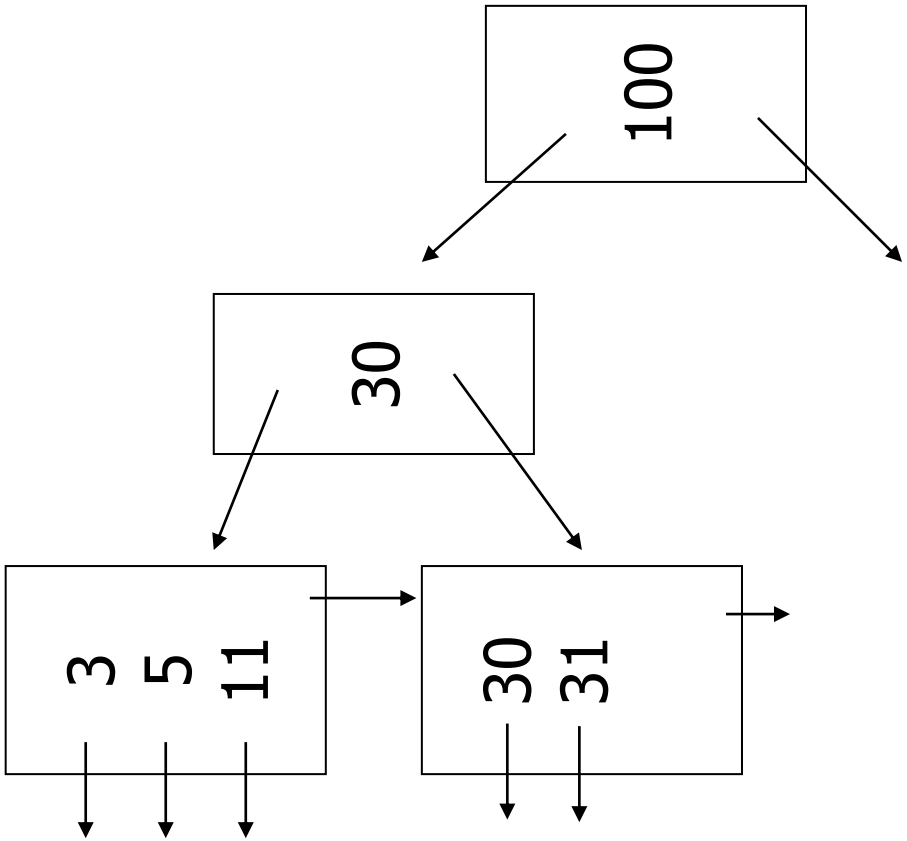
(b) leaf overflow

(c) non-leaf overflow

(d) new root

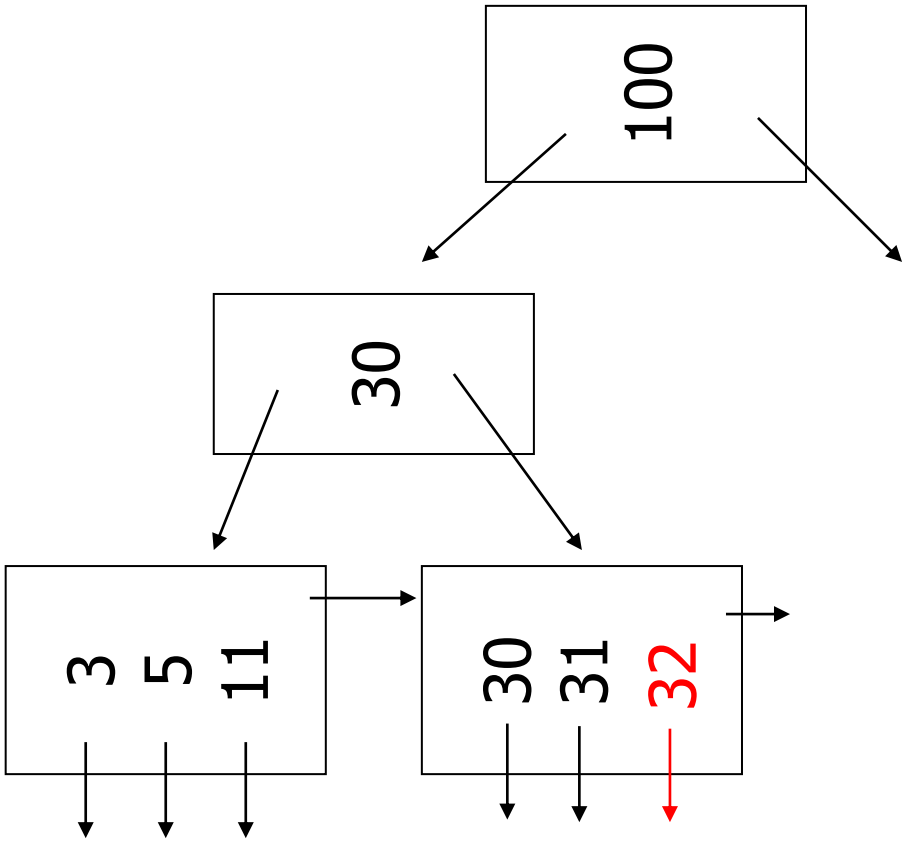
# (a) Insert key = 32

n=3



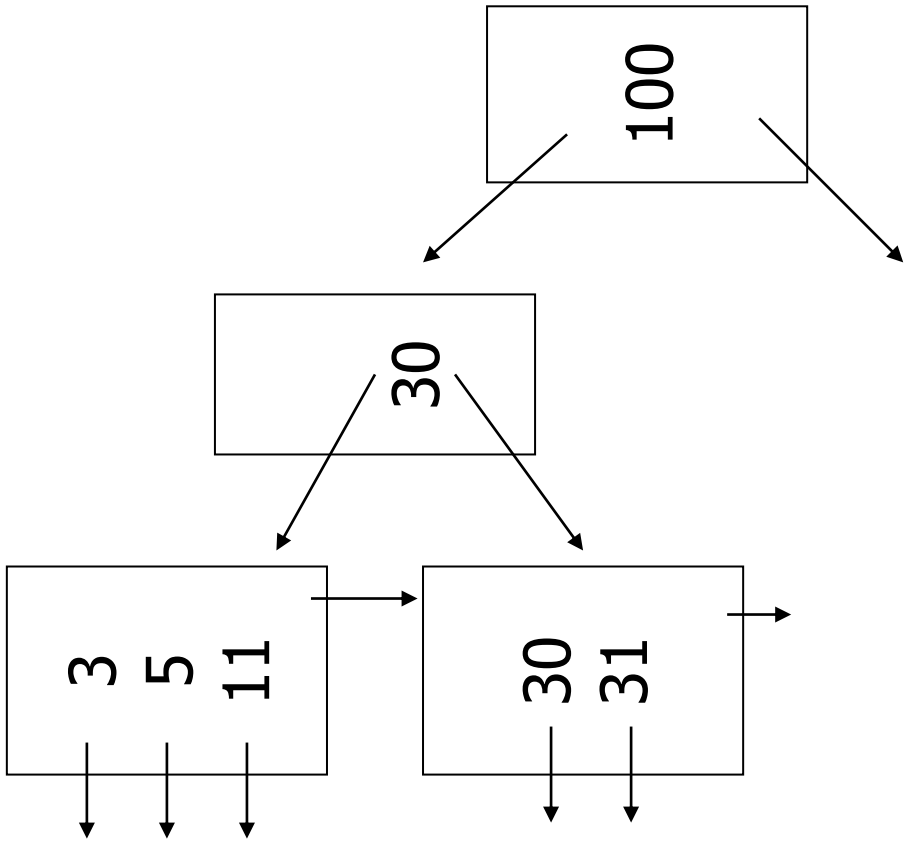
# (a) Insert key = 32

n=3



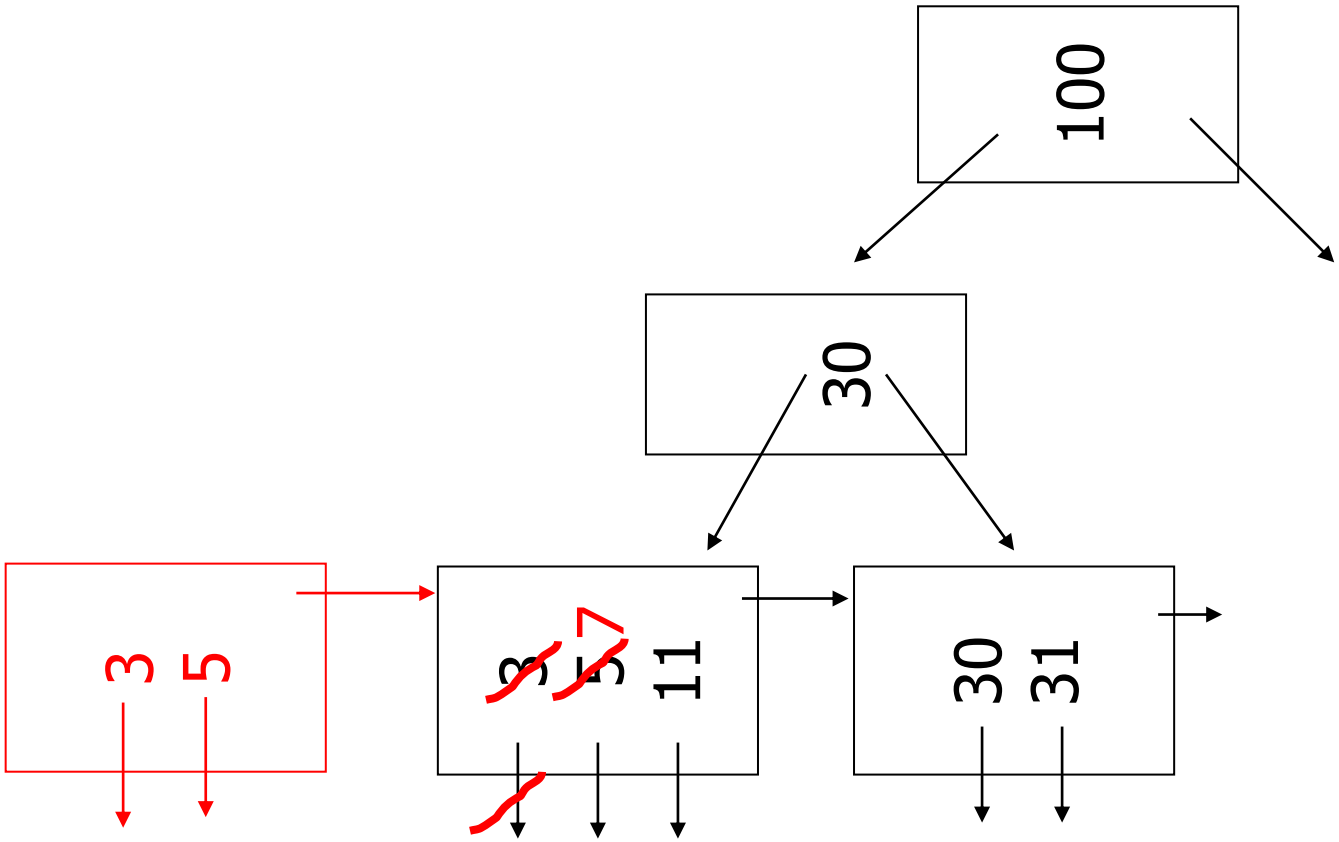
# (a) Insert key = 7

n=3



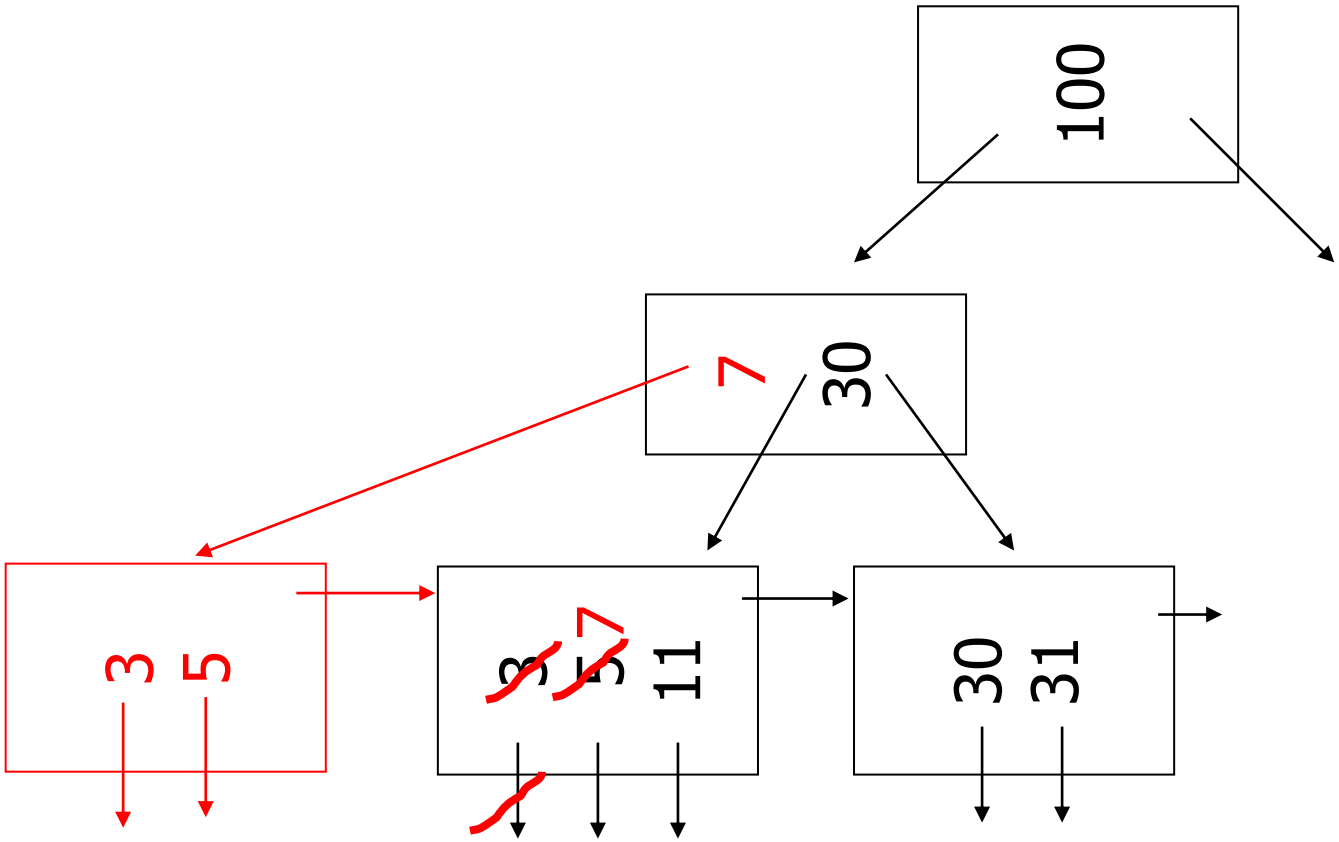
# (a) Insert key = 7

n=3



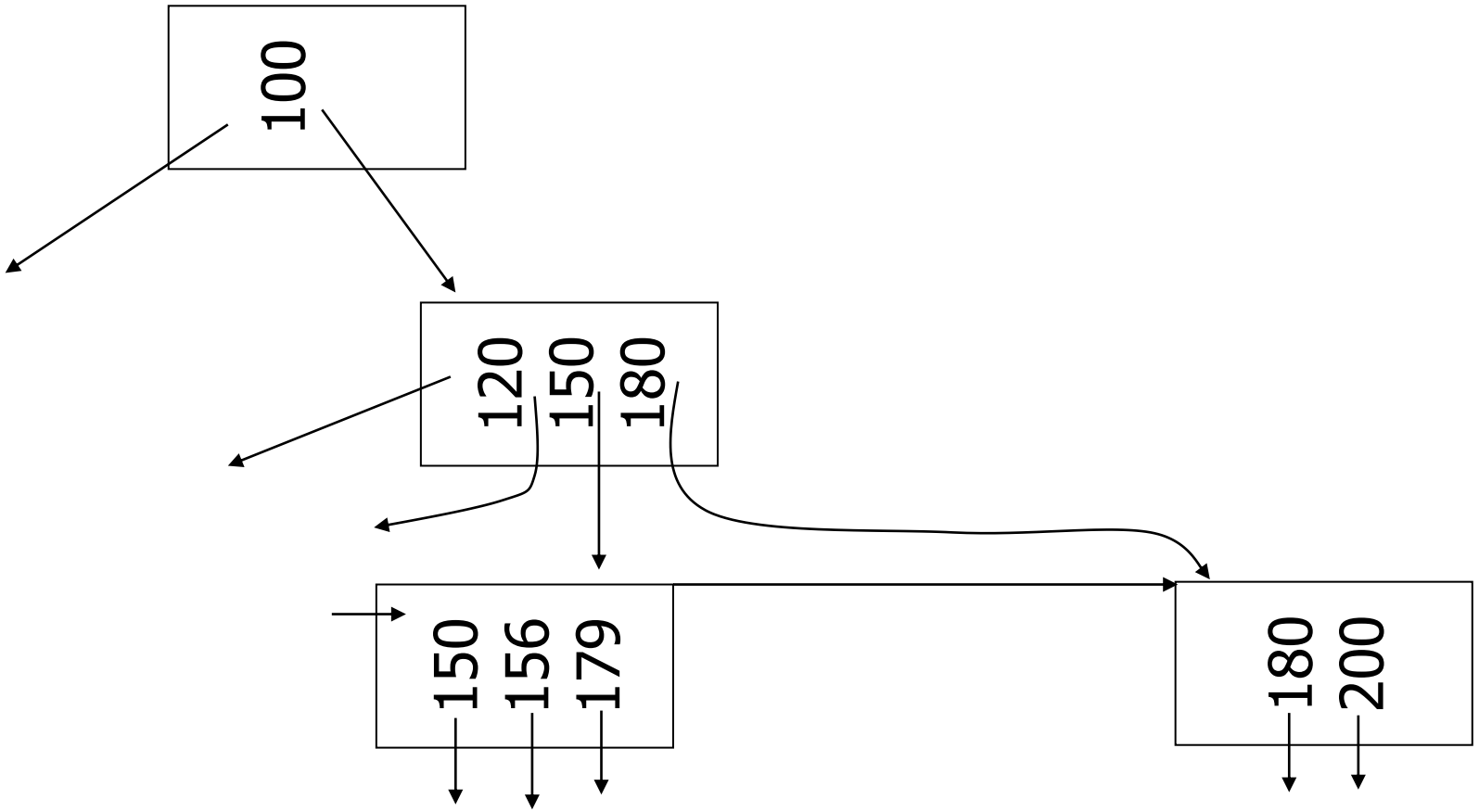
# (a) Insert key = 7

n=3



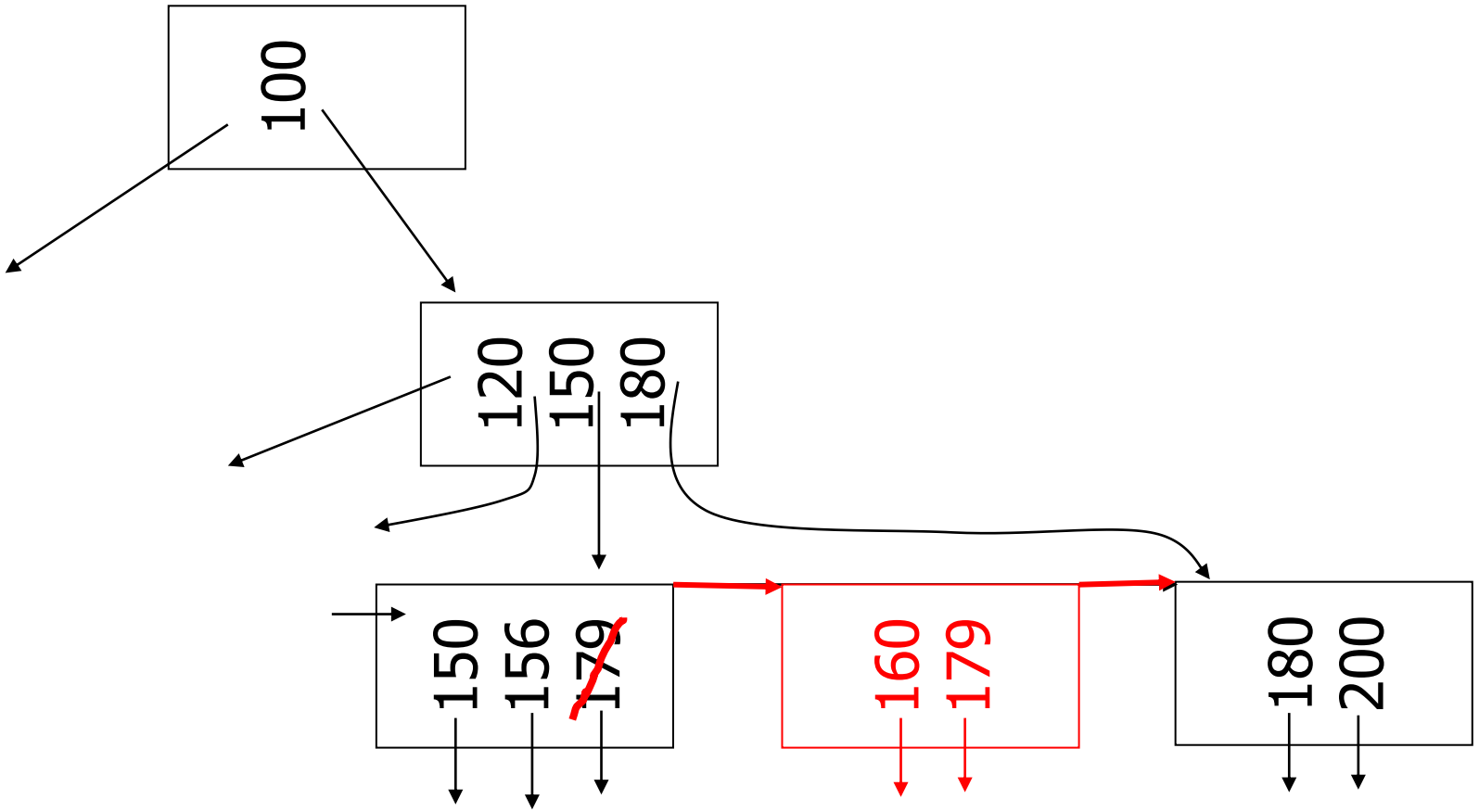
# (c) Insert key = 160

n=3



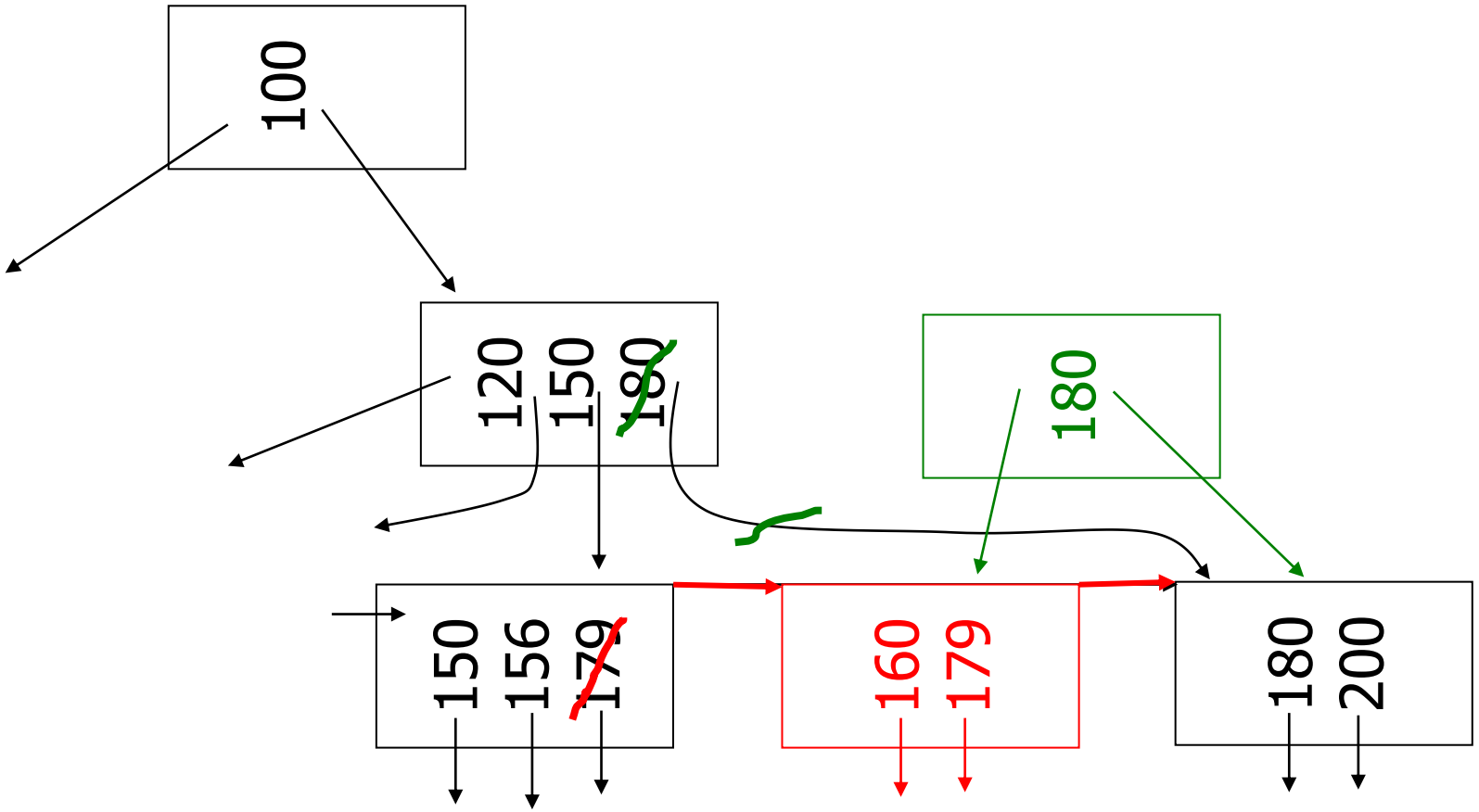
# (c) Insert key = 160

n=3



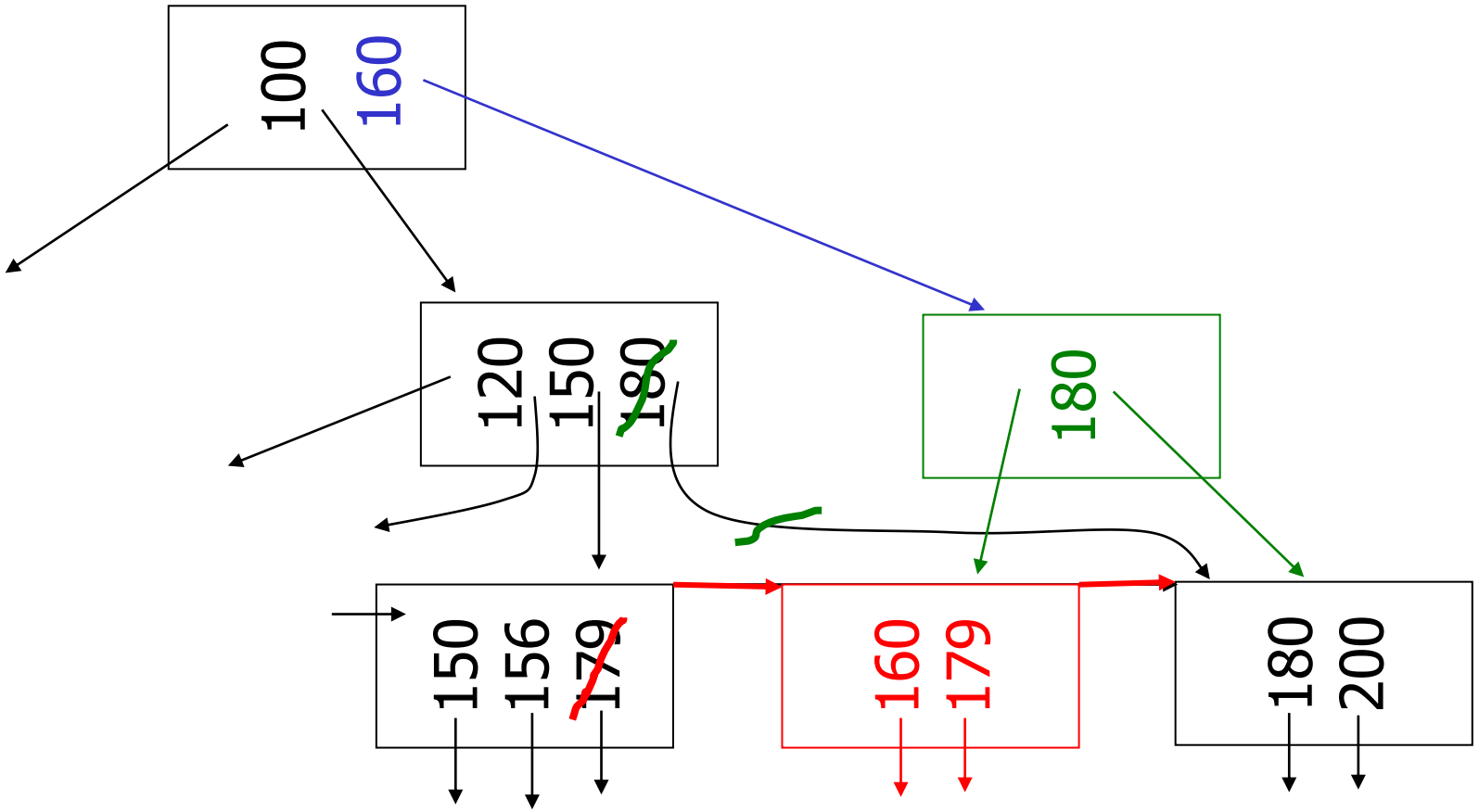
# (c) Insert key = 160

n=3



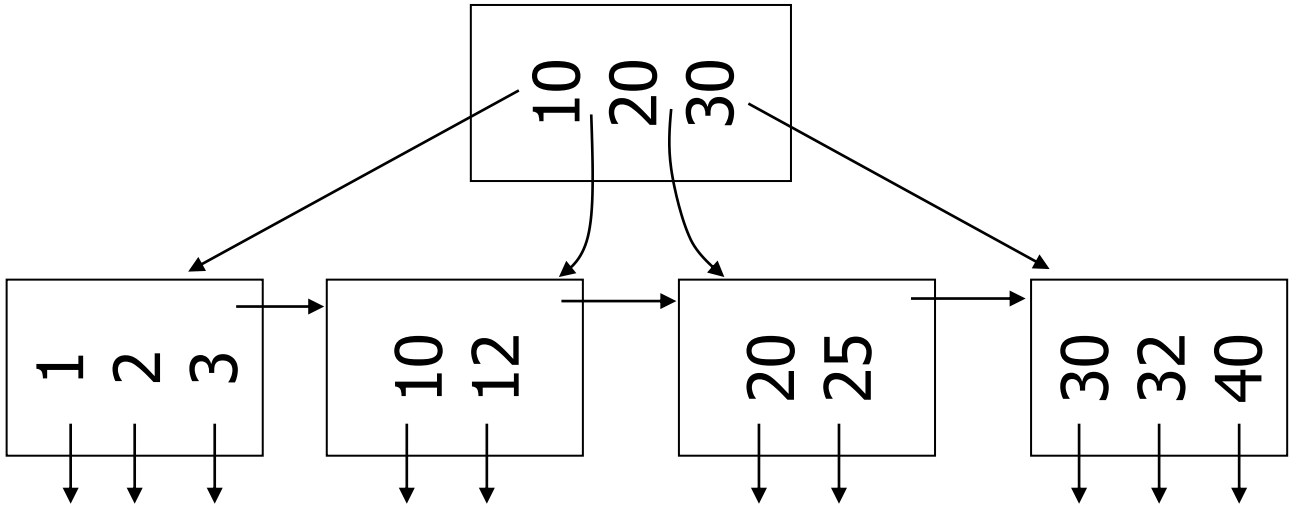
# (c) Insert key = 160

n=3



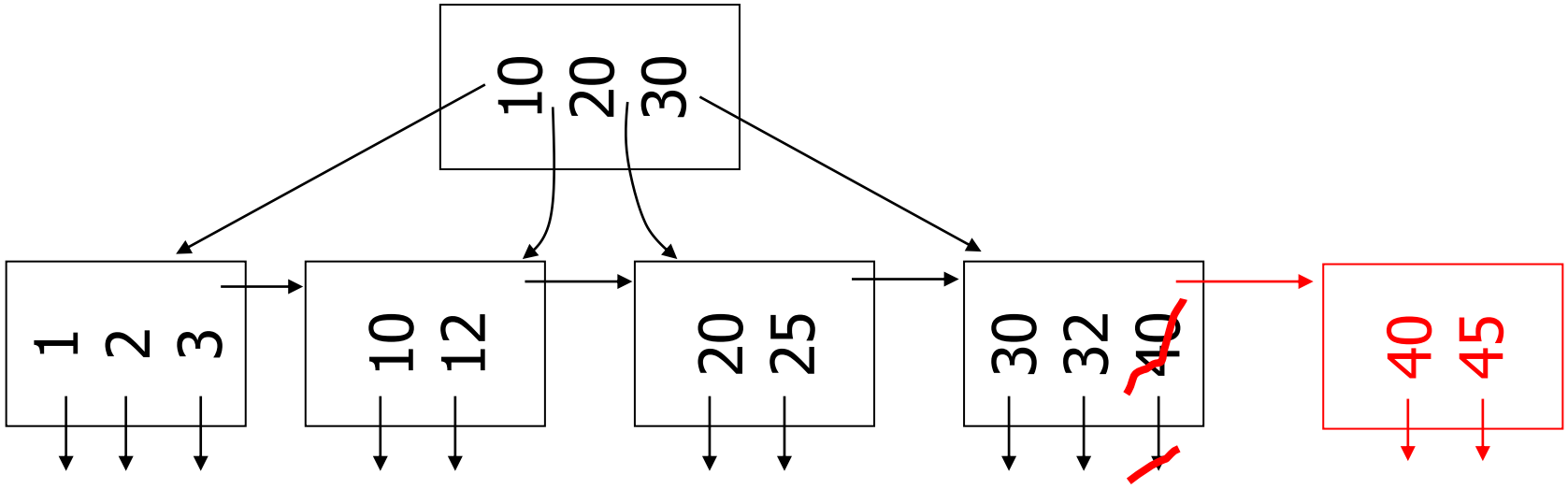
# (d) New root, insert 45

n=3



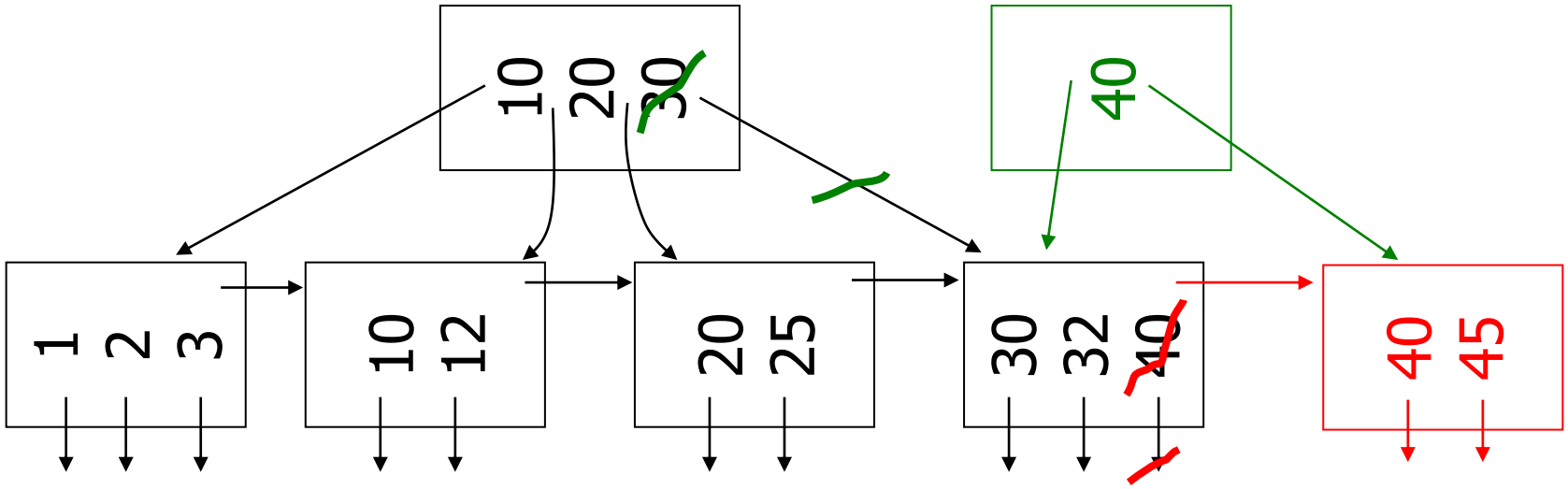
# (d) New root, insert 45

n=3



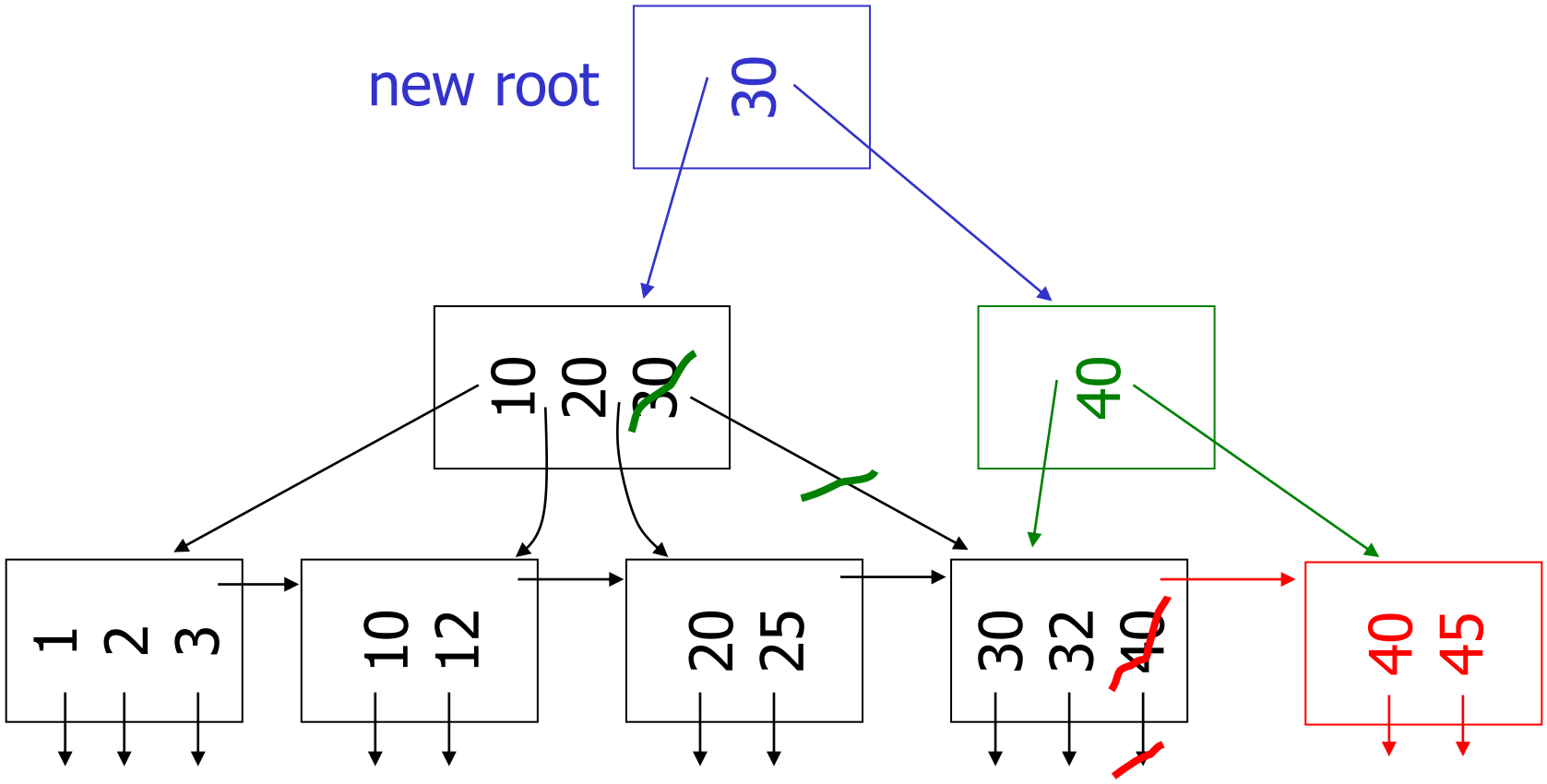
# (d) New root, insert 45

n=3



# (d) New root, insert 45

n=3



# Deletion from B+tree

(a) Simple case - no example

(b) Coalesce with neighbor (sibling)

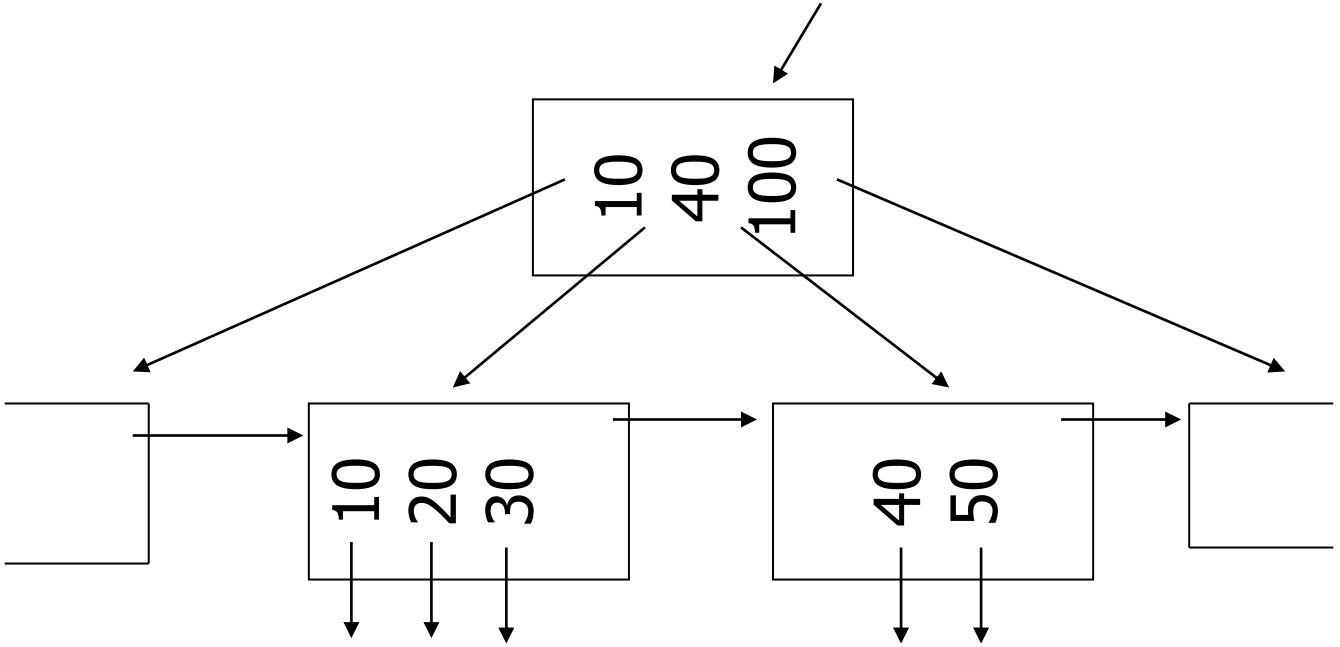
(c) Re-distribute keys

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

# (b) Coalesce with sibling

– Delete 50

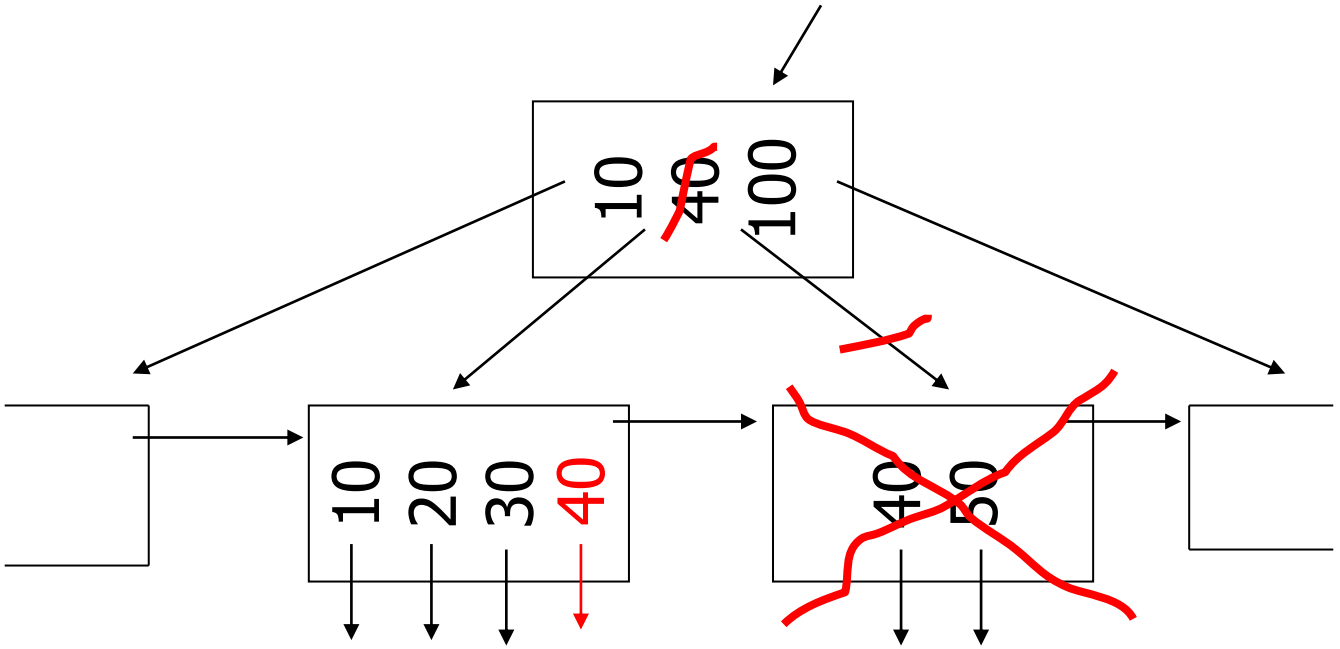
n=4



# (b) Coalesce with sibling

– Delete 50

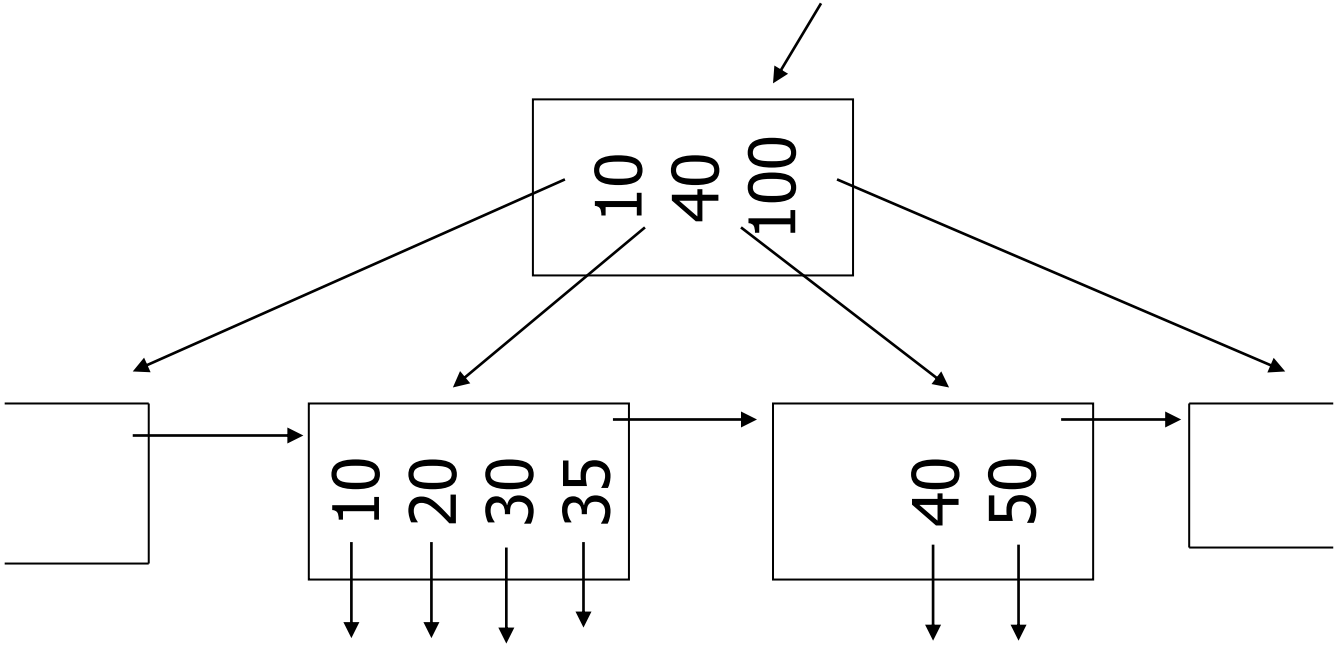
n=4



# (c) Redistribute keys

– Delete 50

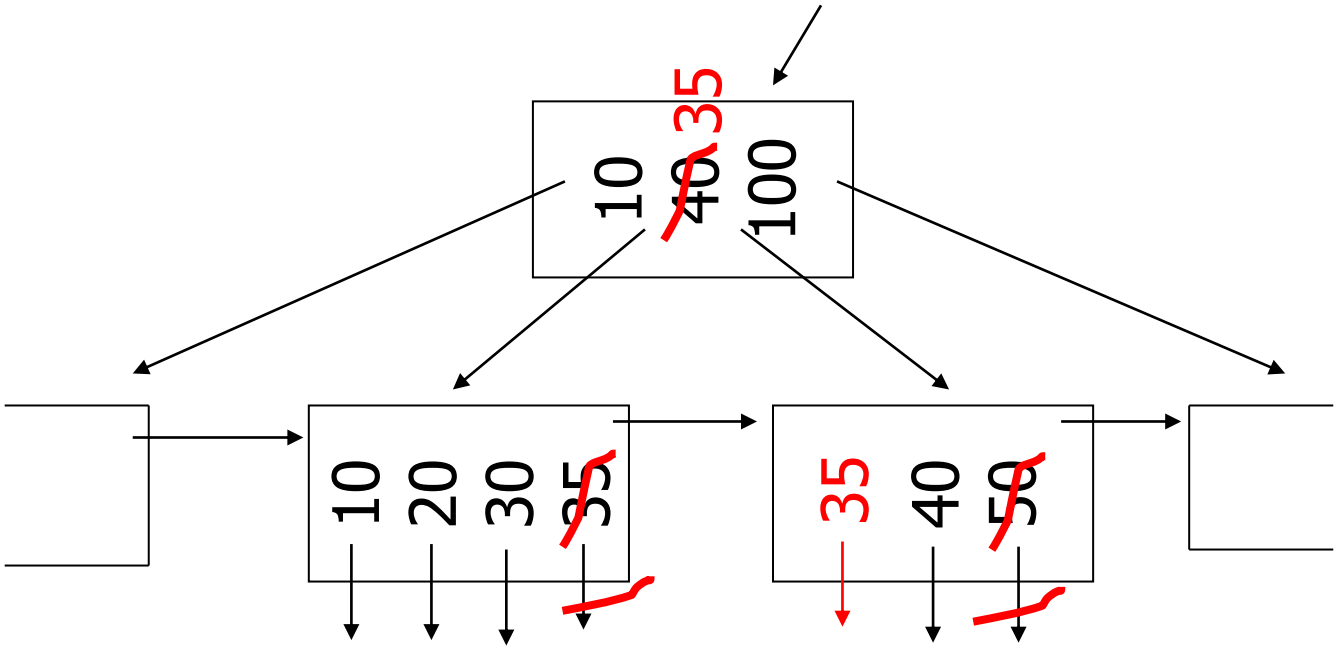
n=4



# (c) Redistribute keys

– Delete 50

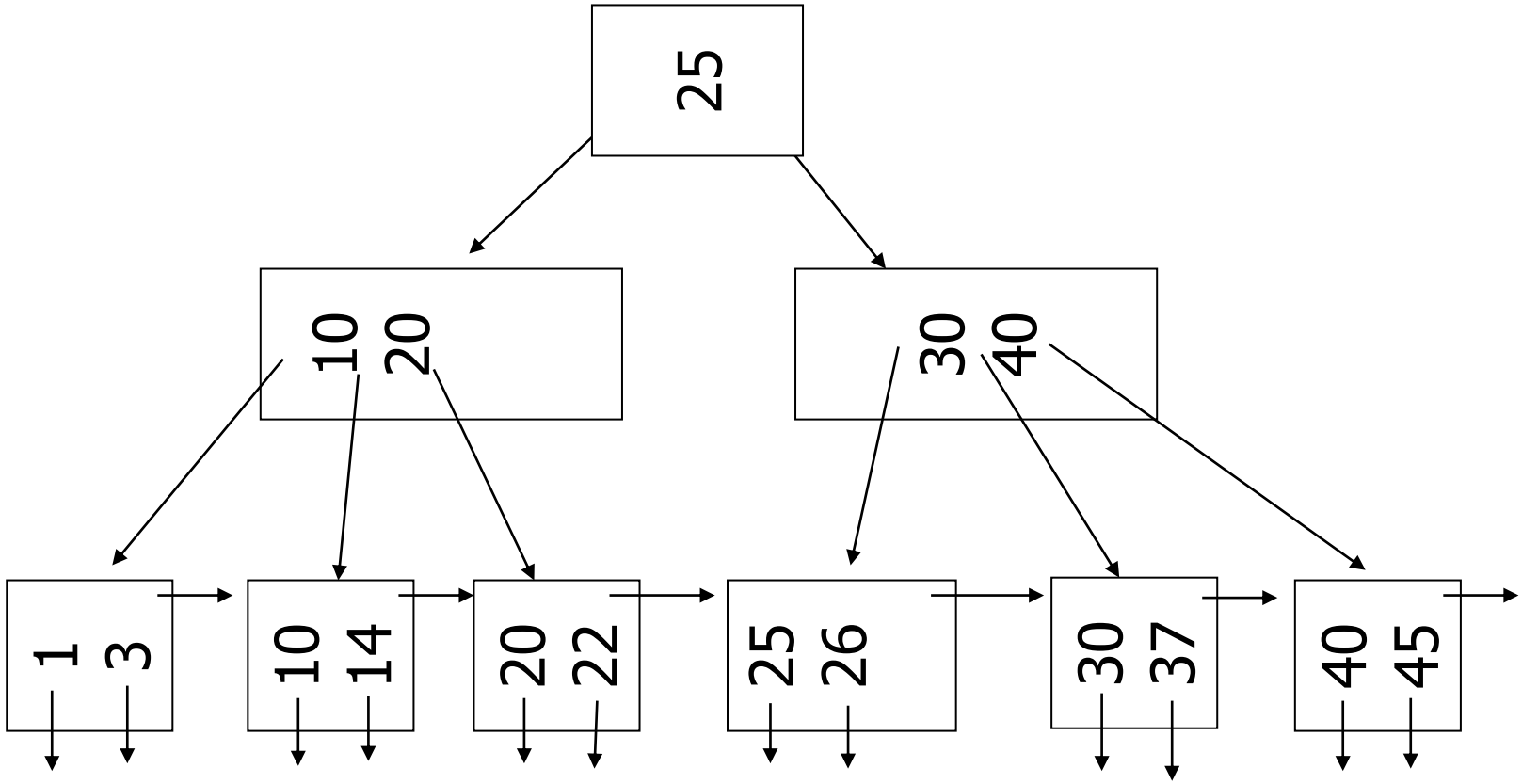
n=4



# (d) Non-leaf coalesce

– Delete 37

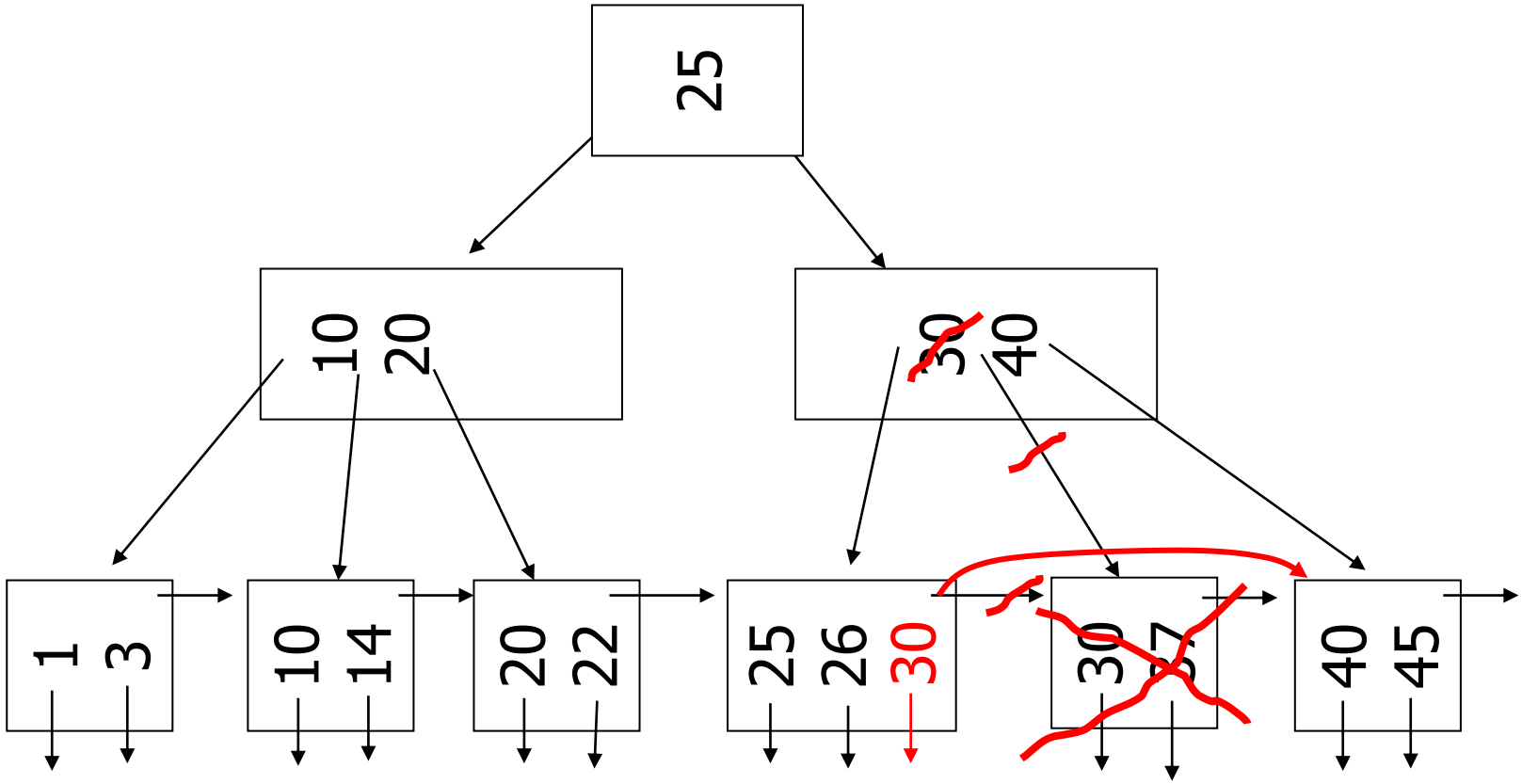
n=4



# (d) Non-leaf coalesce

– Delete 37

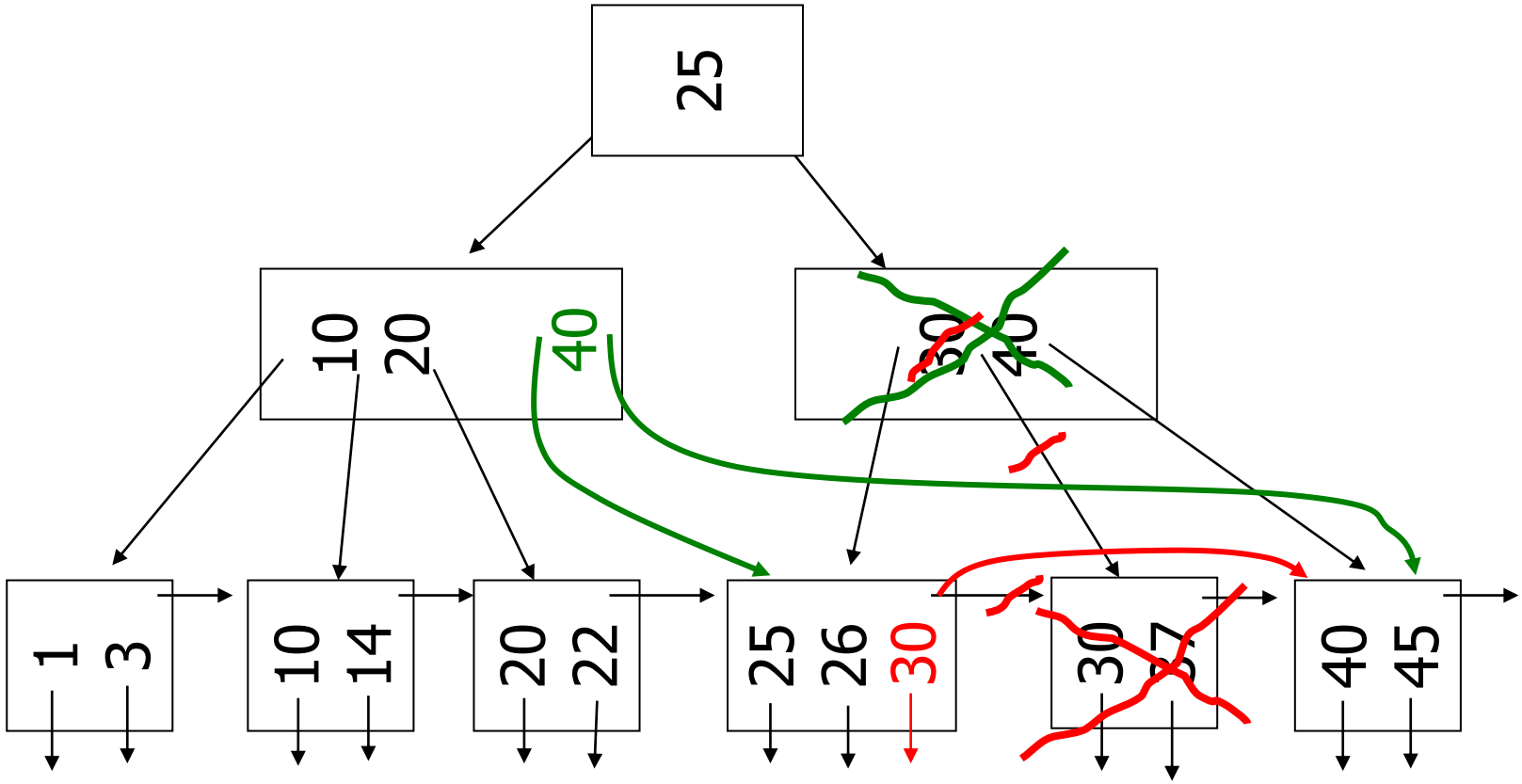
n=4



# (d) Non-leaf coalesce

– Delete 37

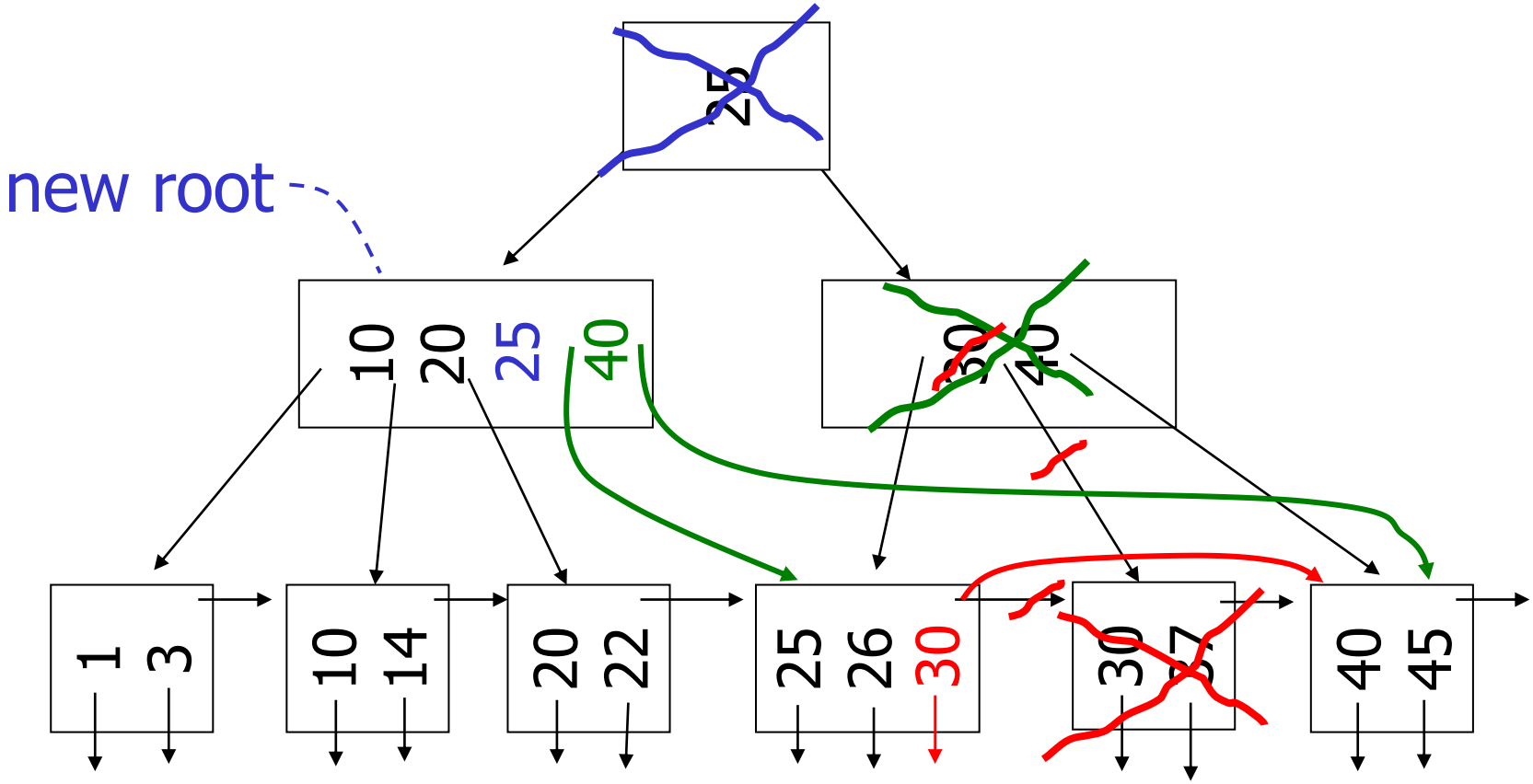
n=4



# (d) Non-leaf coalesce

– Delete 37

n=4



# B+tree deletions in practice

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

# 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