Query Processing

Part 3: B+Trees

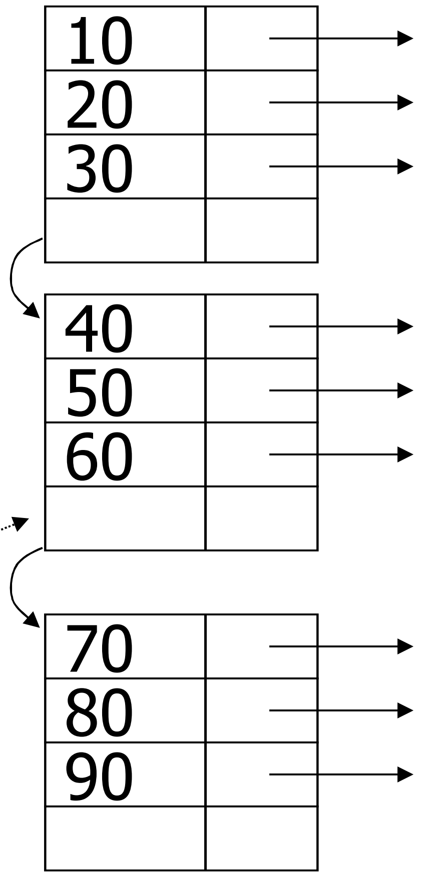
Dense and Sparse Indexes

- Advantages:
  - Simple
  - Index is sequential file, which is good for scans
  - Can take advantage of the disk structure
- Disadvantages:
  - Insertions expensive, and/or
  - Lose sequentiality & balance

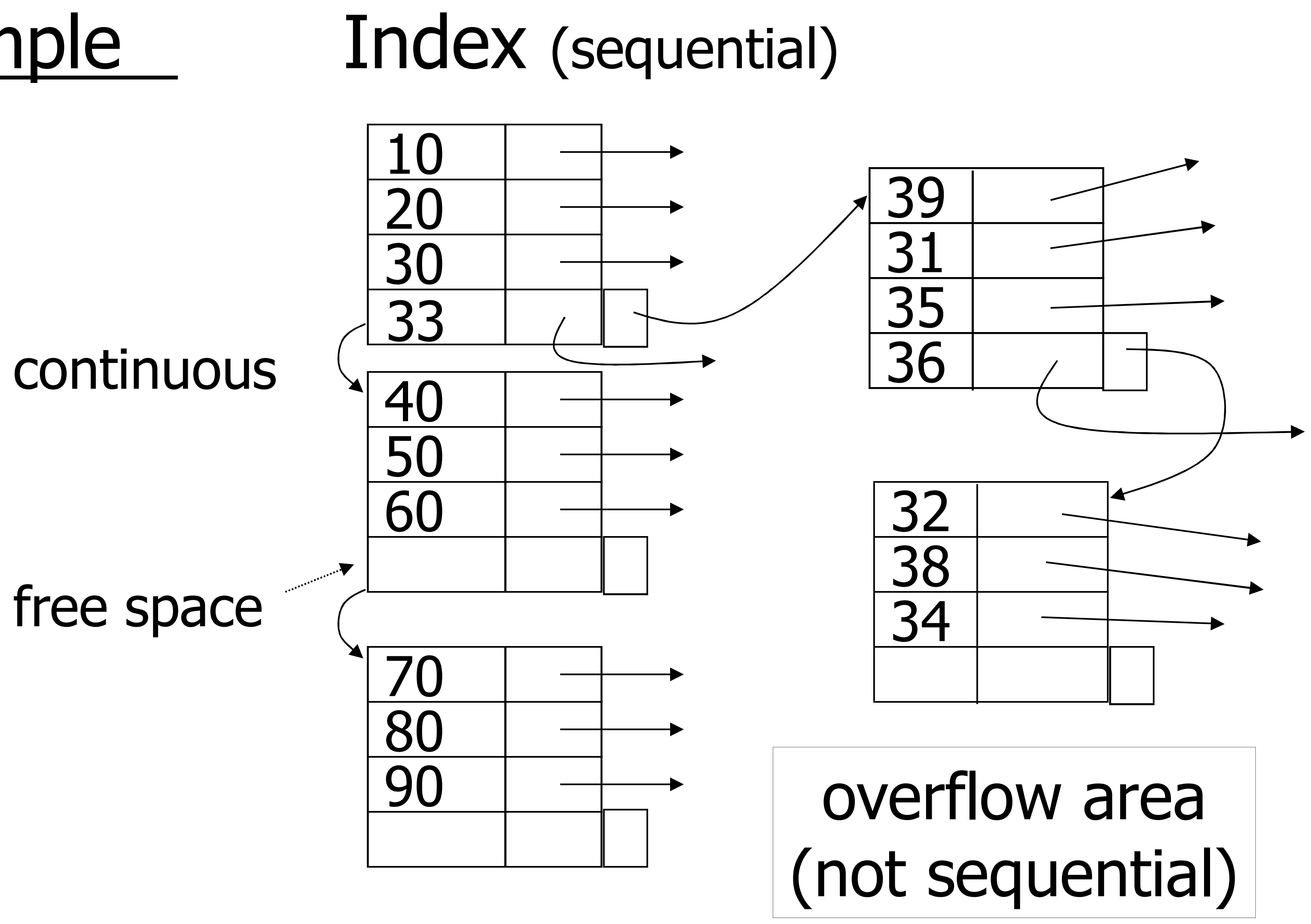
Example Index (sequential)

continuous

free space



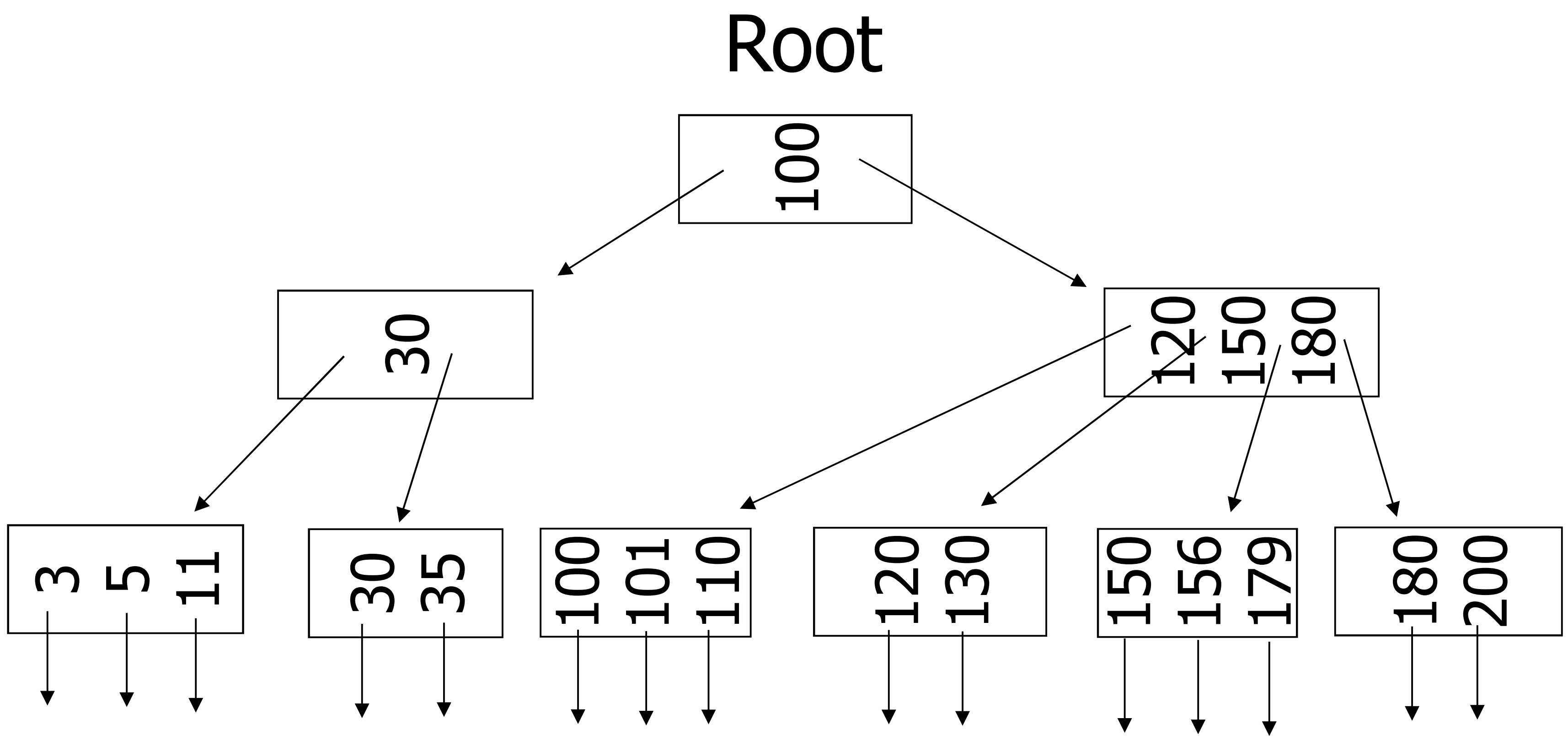
Example


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)

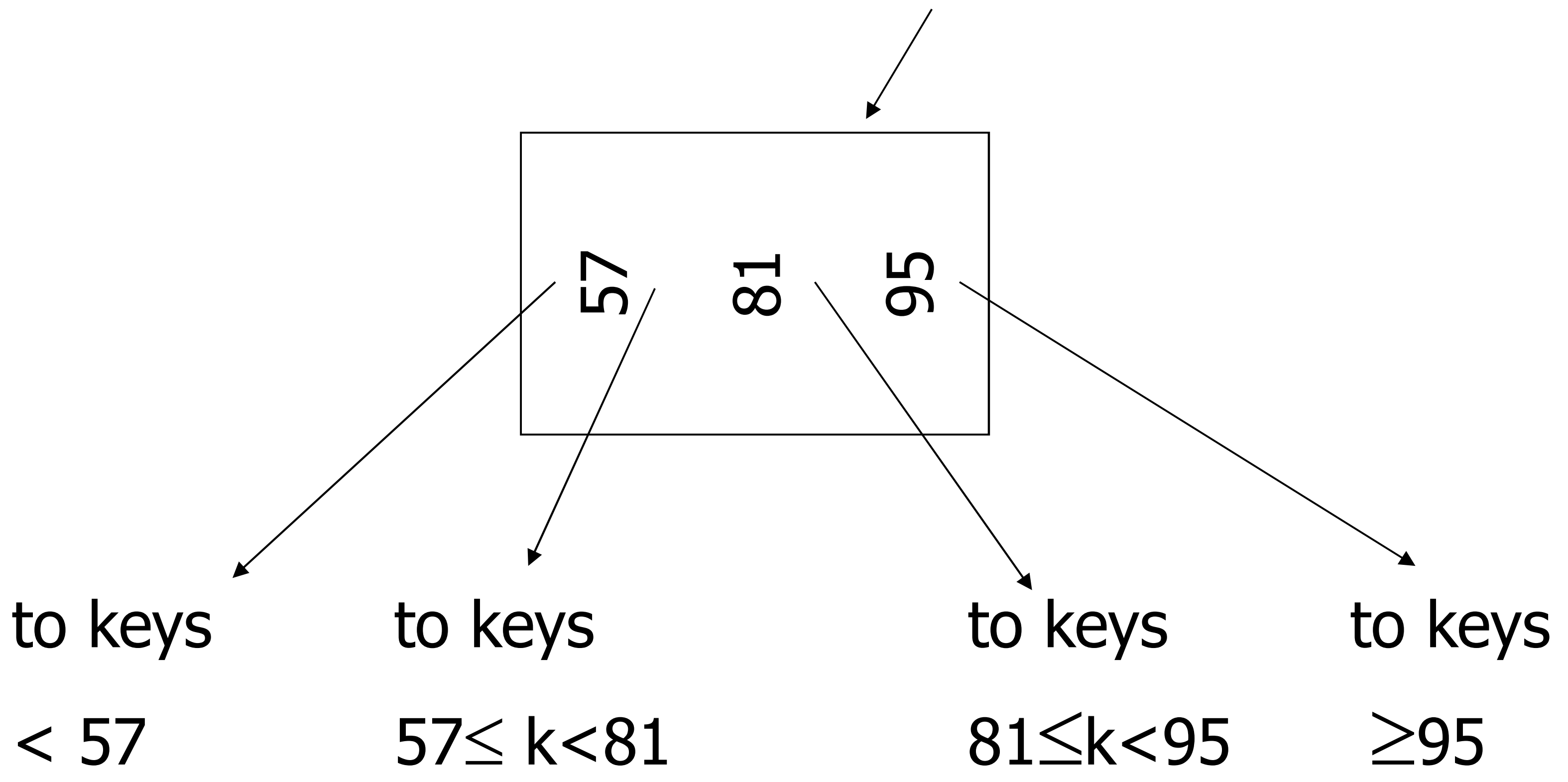
B+Trees: Main Idea

- Give up on sequentiality of index
- Try to get “balance”
- A node corresponds to a block
  - Because have to read (at least) one block
- Control the height by requiring nodes to be at least half full

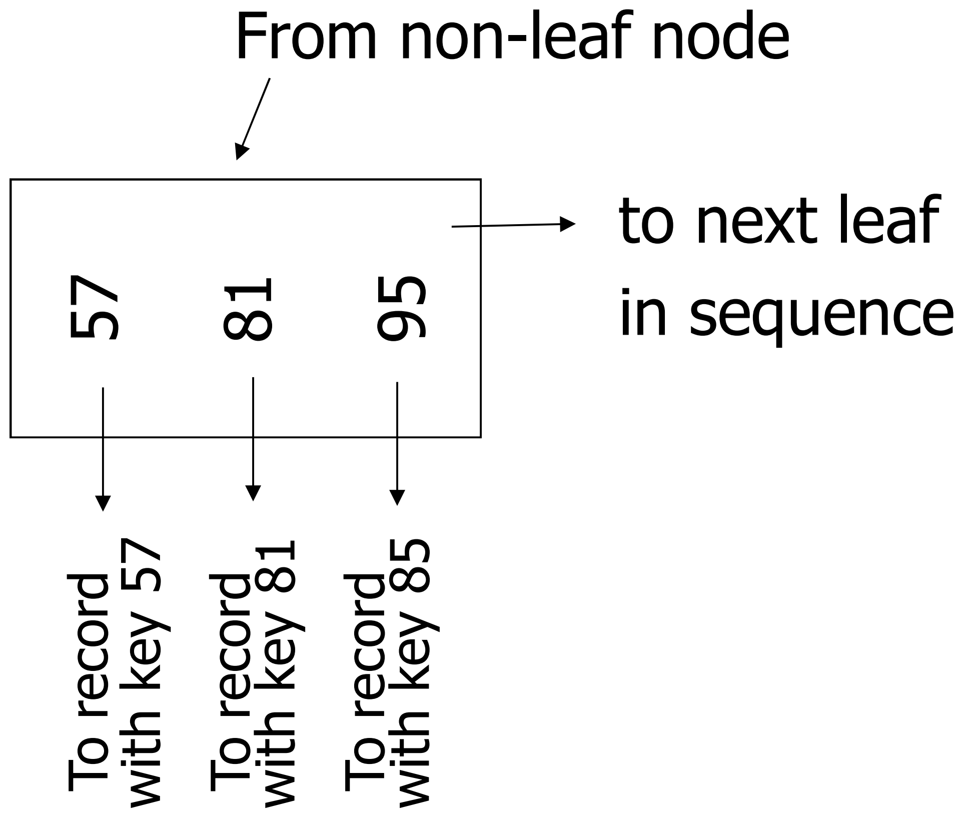
B+Tree Example n=3



Sample non-leaf

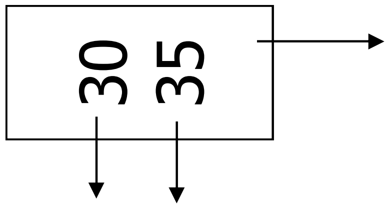


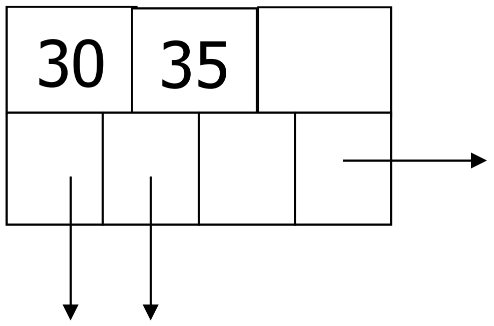
Sample leaf node



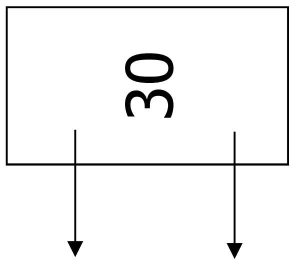
In textbook's notation n=3

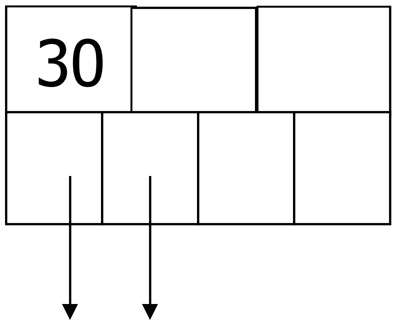
Leaf:





Non-leaf:





Maximal Sizes of Nodes

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

- In leaf nodes, a pointer is associated with each key, and there is one additional pointer to the next leaf in sequential order
- In non-leaf nodes, the leftmost pointer has no associated value

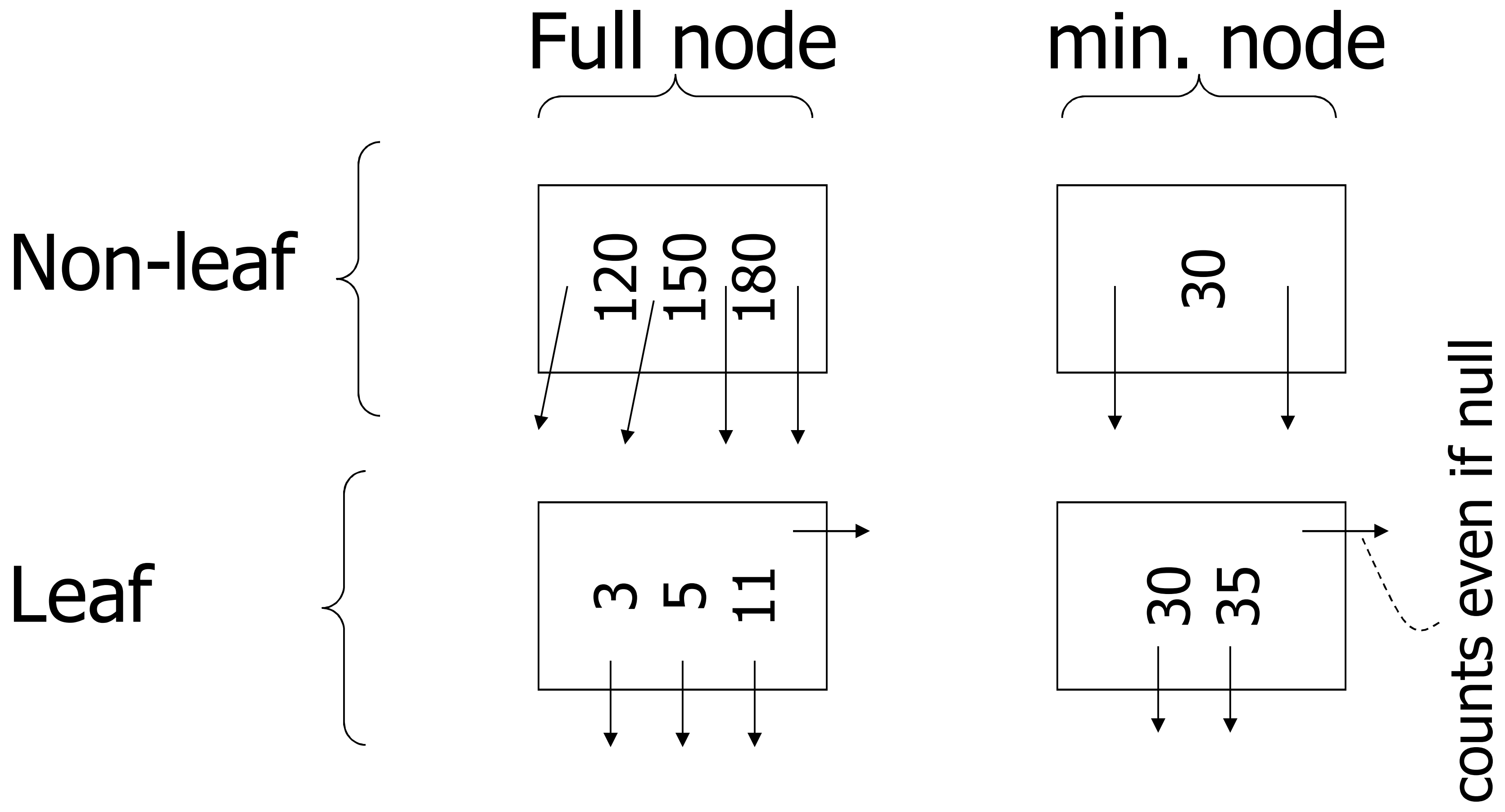
Don’t want nodes to be too empty

- Use at least

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

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

$\boxed{n=3}$


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"

Third rule on next slide

(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	n	$\lfloor (n+1)/2 \rfloor$	$\lfloor (n+1)/2 \rfloor$
Root	n+1	n	2	1

The sequence pointer at the bottom level is not counted in the above table

It is assumed that there are at least two nodes at the bottom level (or else, there is only a root, which may have no pointers at all if the file is empty)

Insertion into a B+Tree

(a) simple case
- space available in leaf

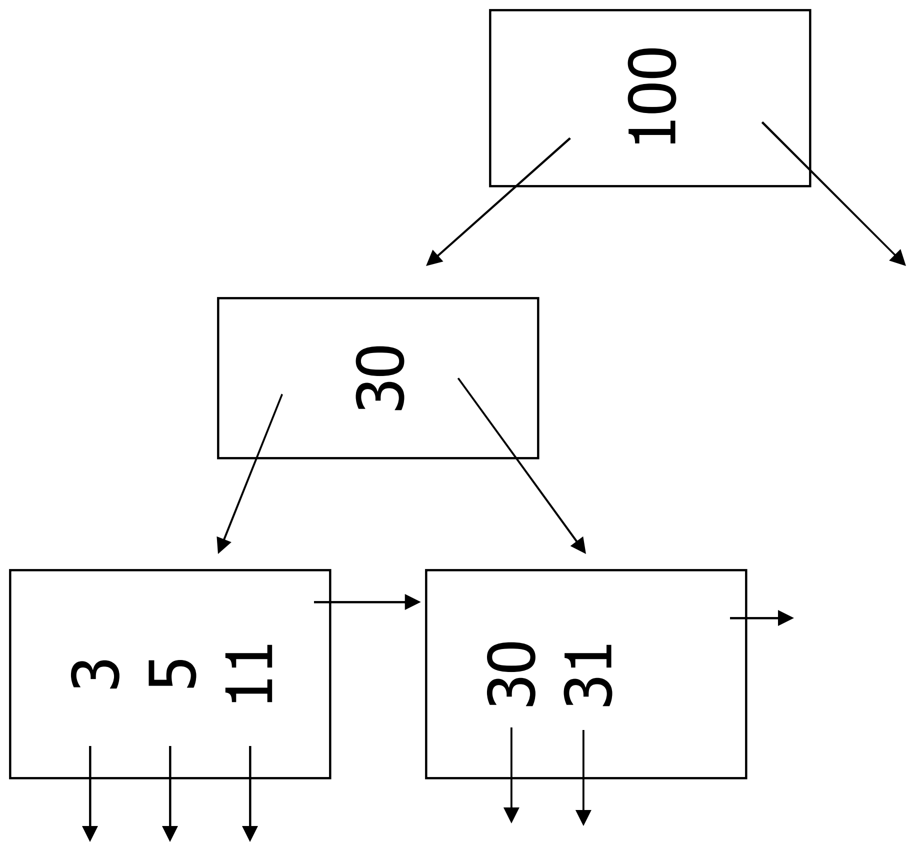
(b) leaf overflow

(c) non-leaf overflow

(d) new root

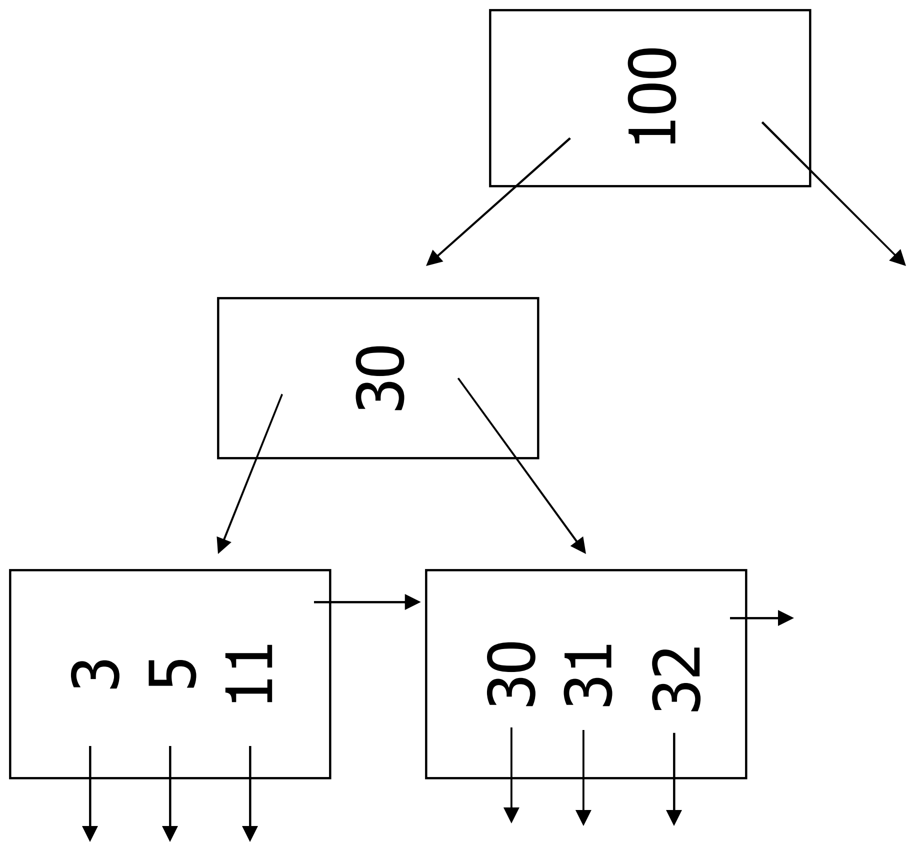
(a) Insert key = 32

n=3



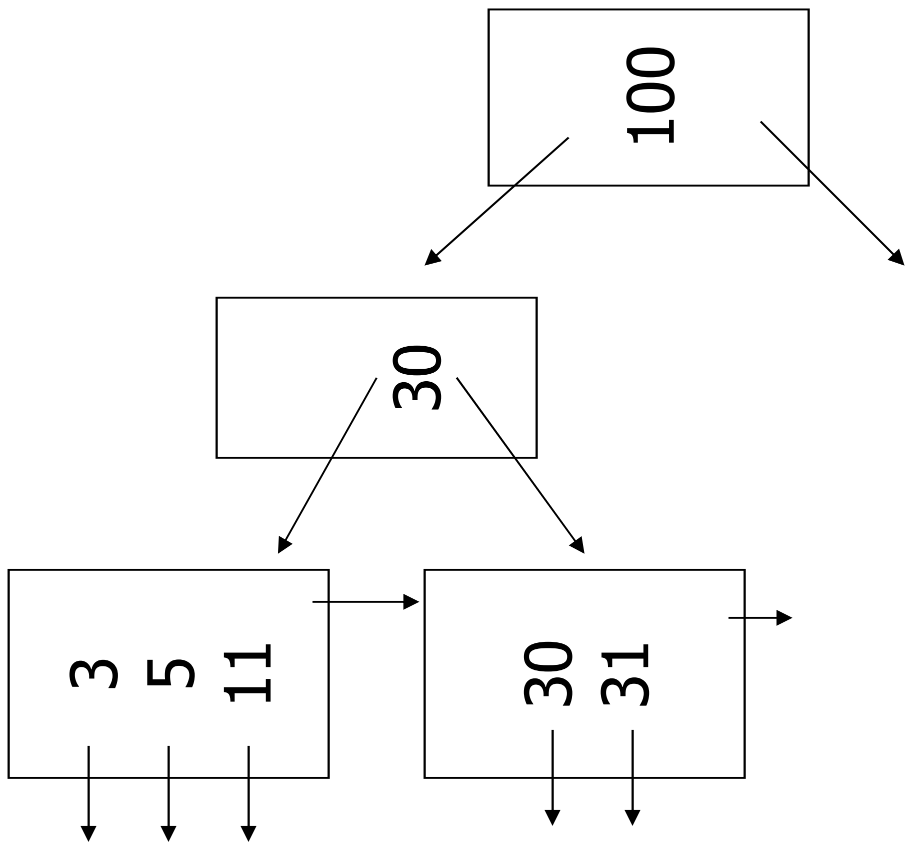
(a) Insert key = 32

n=3



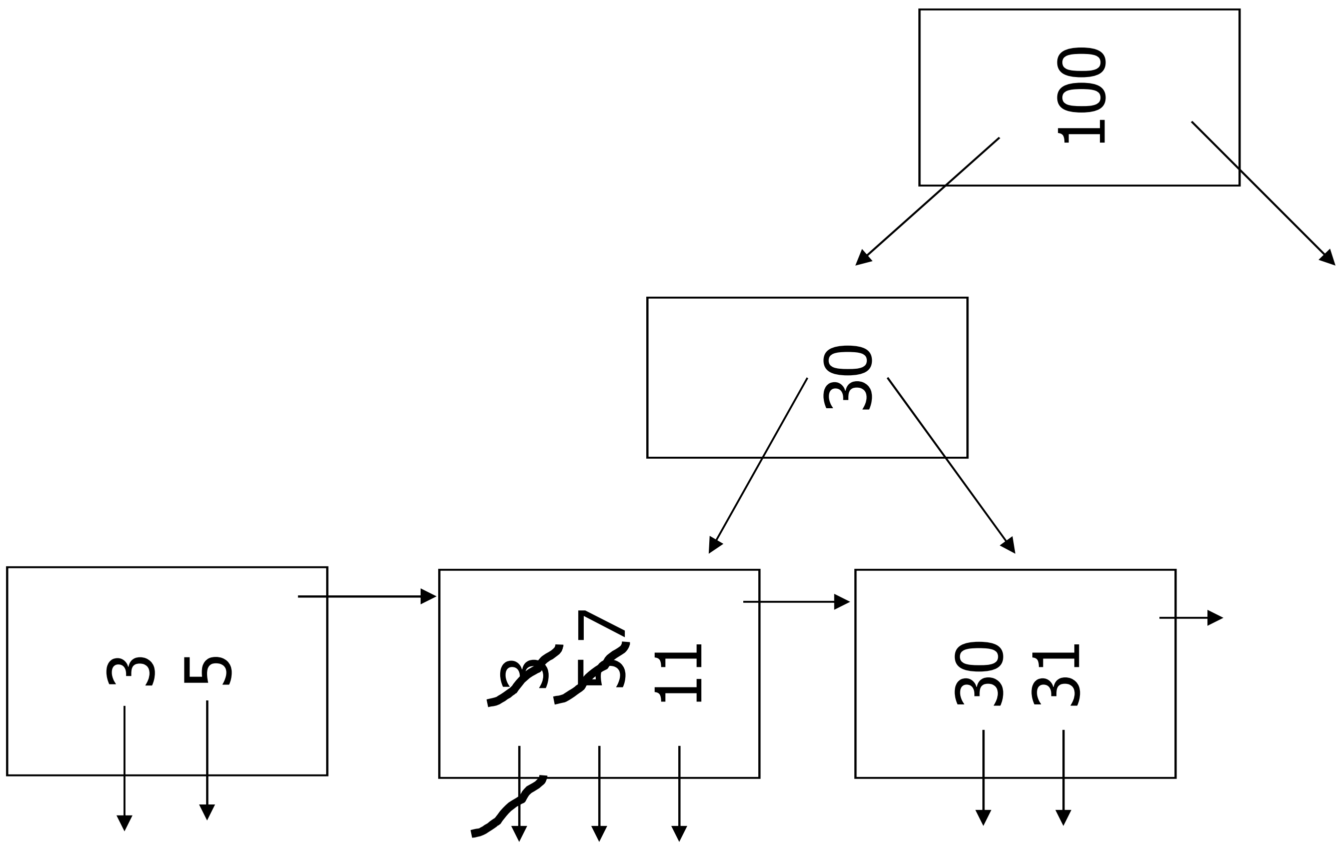
(a) Insert key = 7

n=3



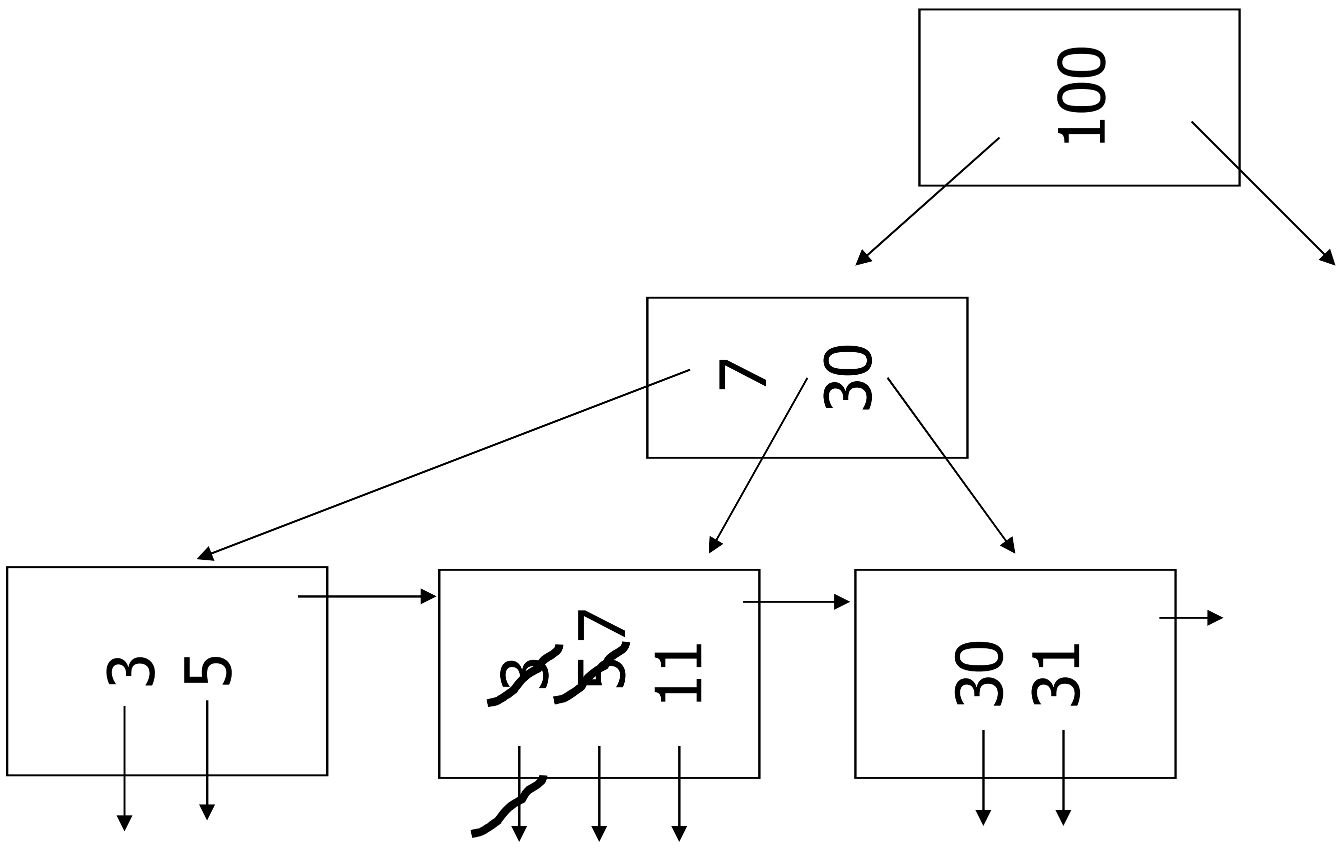
(a) Insert key = 7

n=3



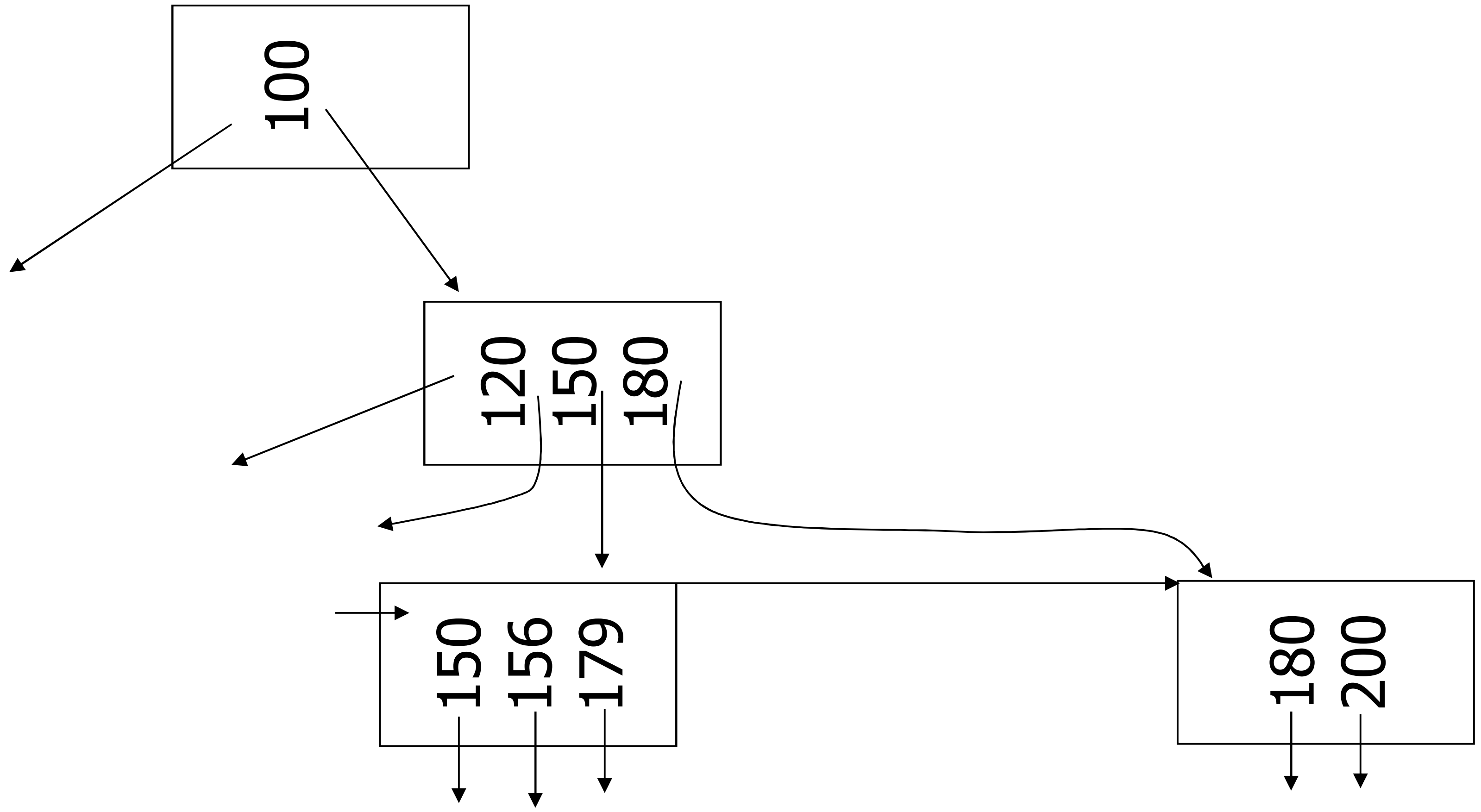
(a) Insert key = 7

$\boxed{n=3}$



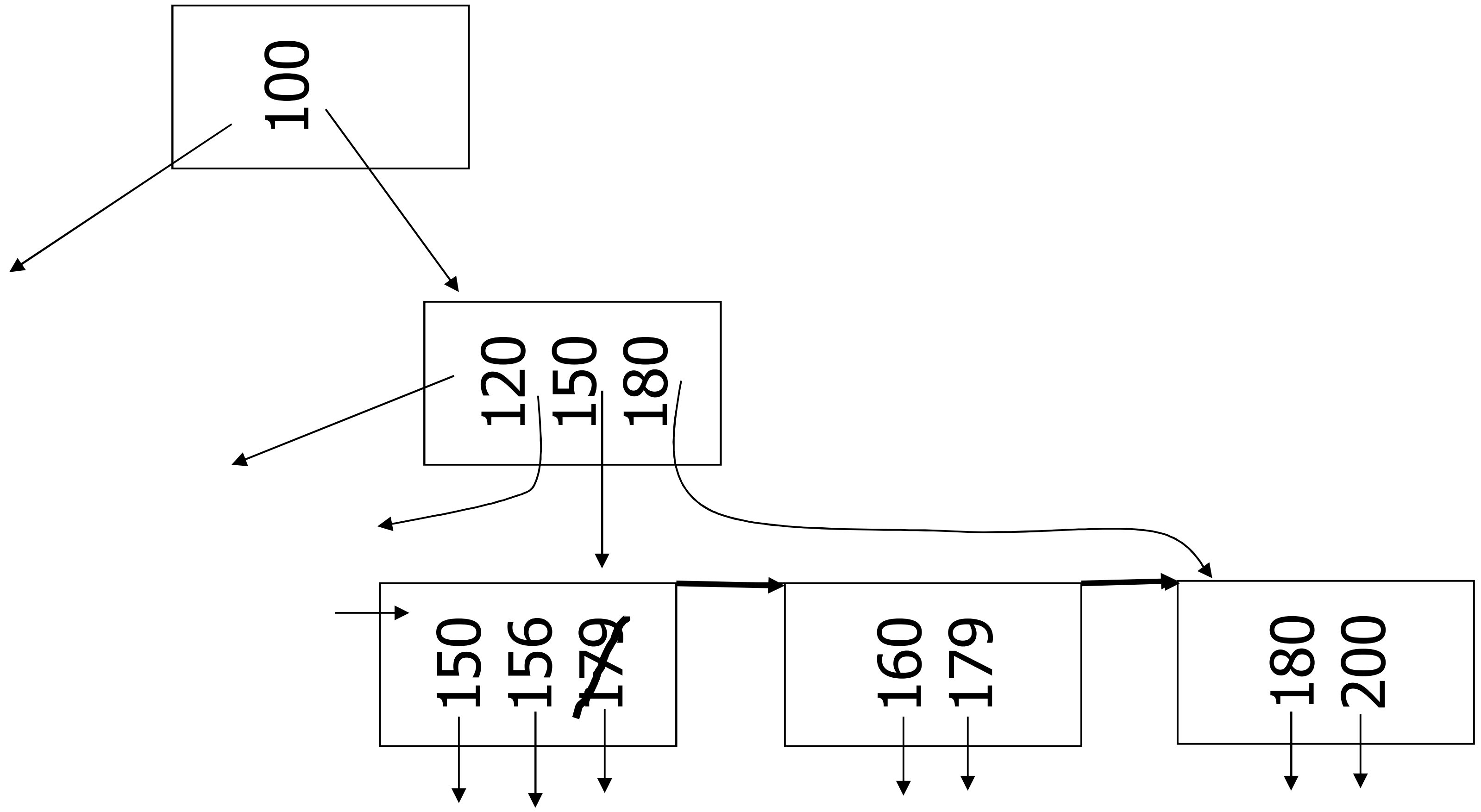
(c) Insert key = 160

n=3



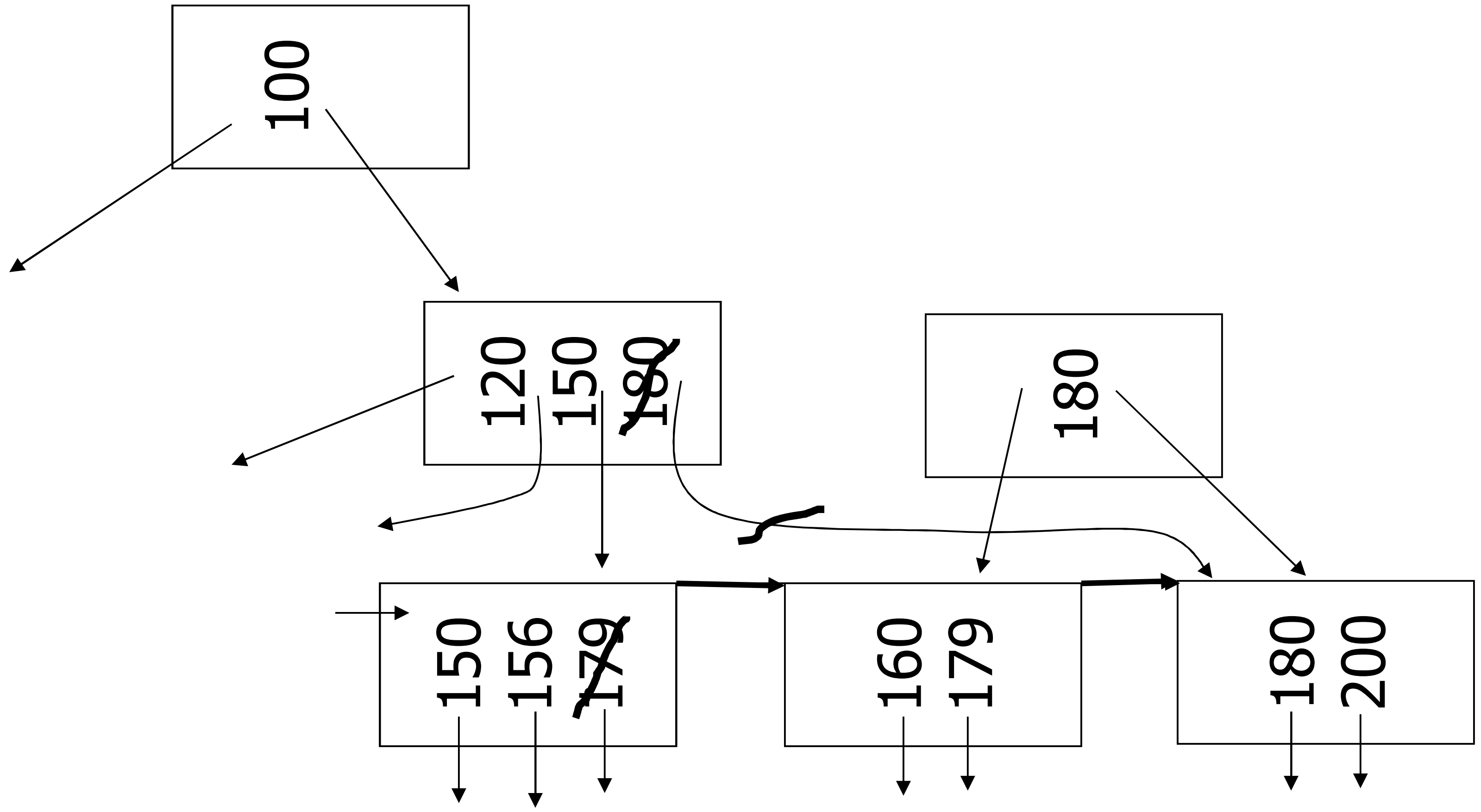
(c) Insert key = 160

n=3



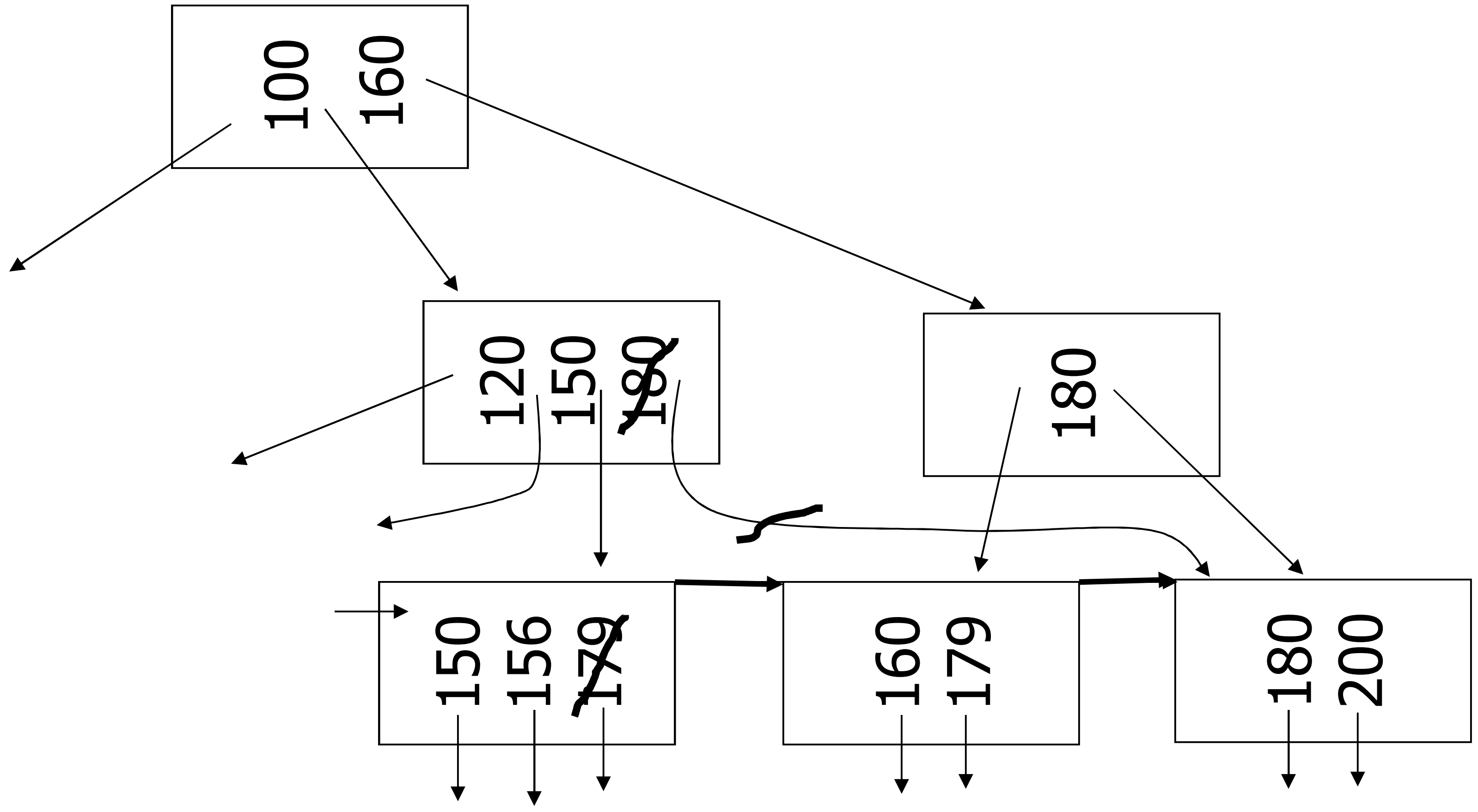
(c) Insert key = 160

n=3



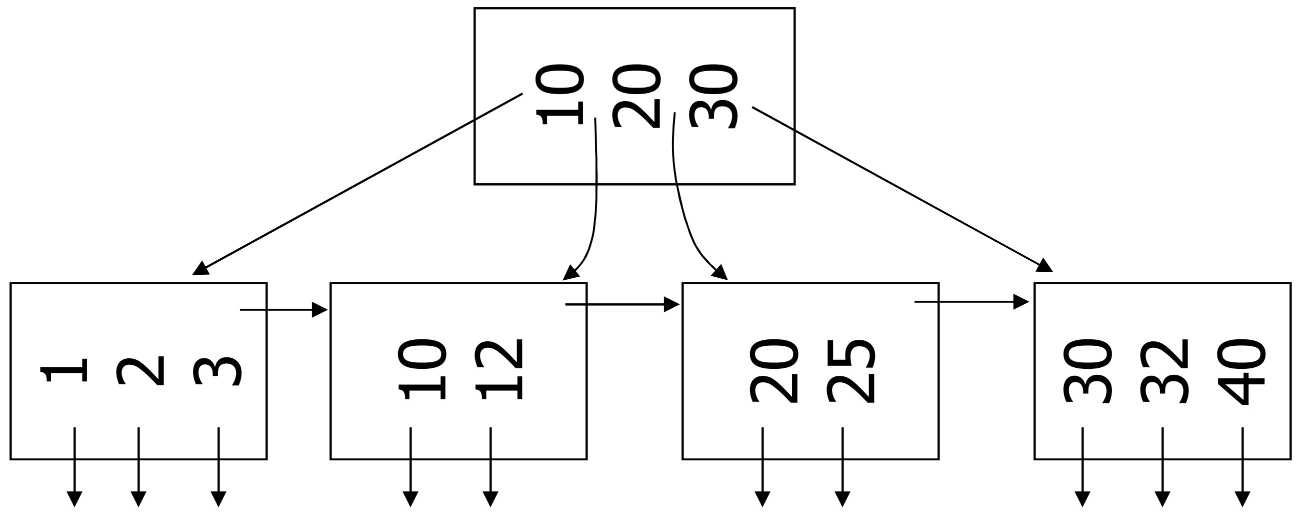
(c) Insert key = 160

n=3



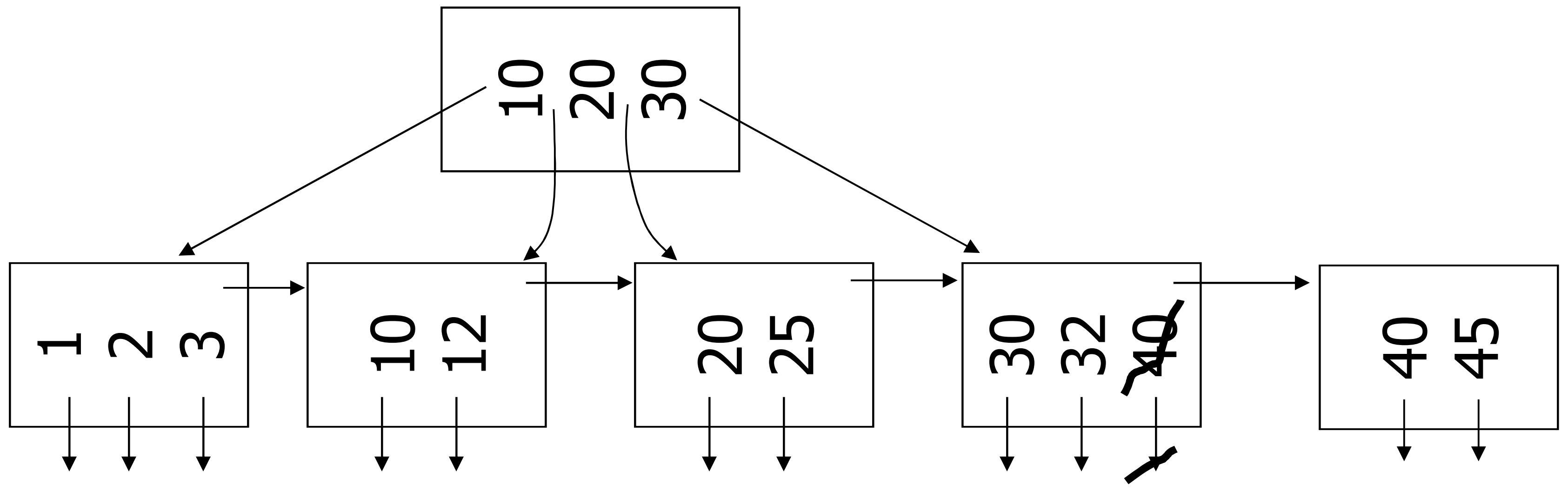
(d) New root, insert 45

n=3



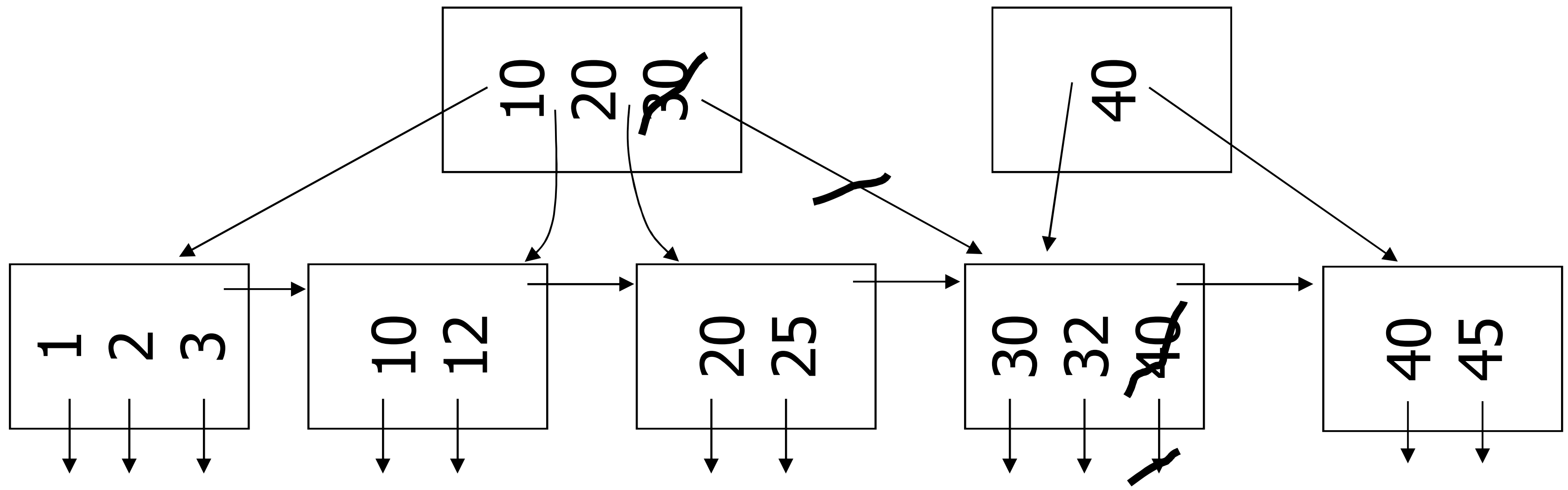
(d) New root, insert 45

n=3



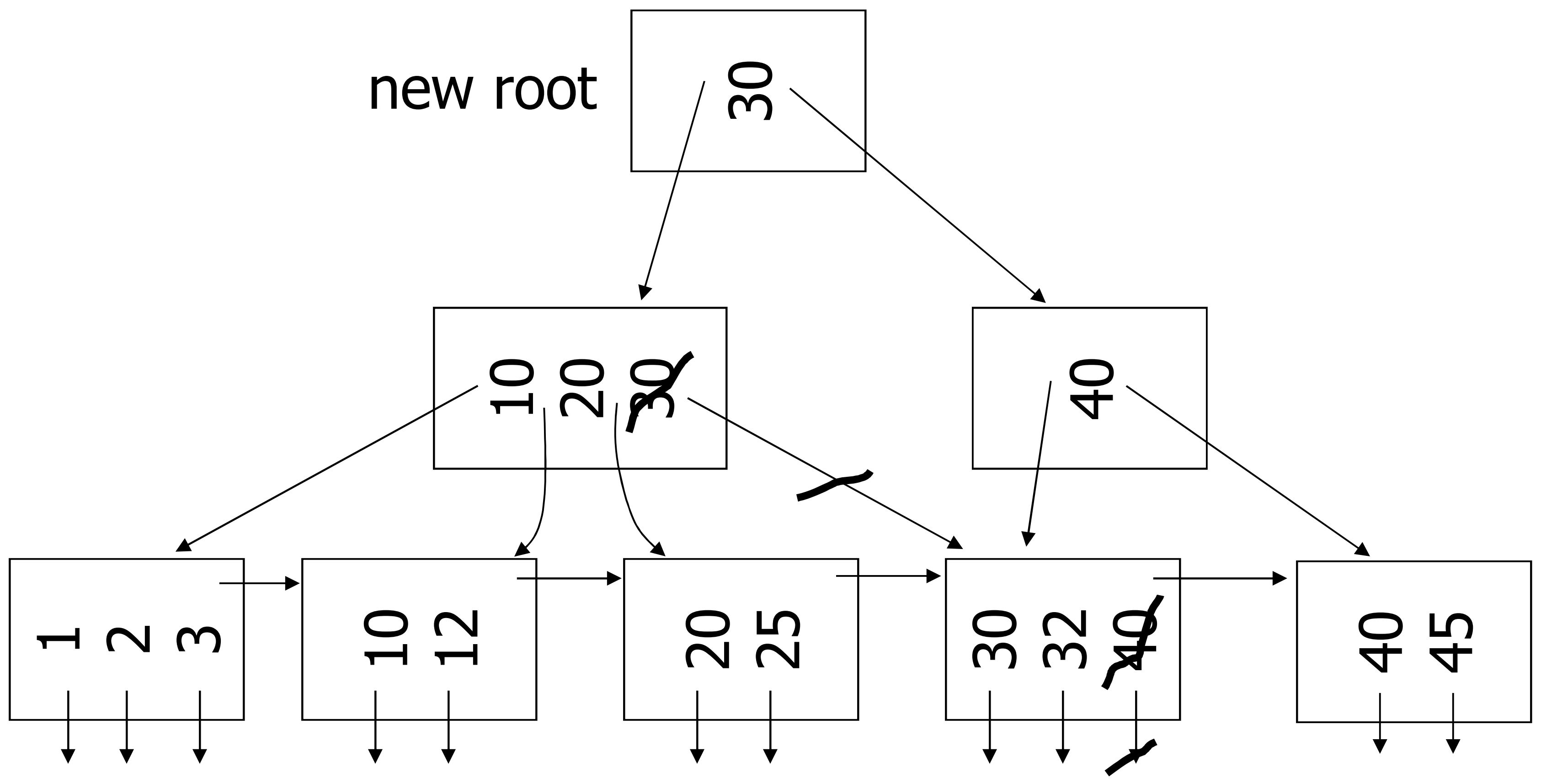
(d) New root, insert 45

n=3



(d) New root, insert 45

n=3



Insertion Made Simple

- View a node as a sorted list of pairs (*k*,*p*)
  - In a leaf, there is a pair (*k*,*p*) for each key and pointer associated with the same record
    - Ignore the sequence pointer to the next node
  - In a non-leaf node, (*k*,*p*) is created for each pointer *p* and value *k* on its left side
    - The meaning of (*k*,*p*) is that *p* points to the subtree whose smallest key is *k*

What is the Pair for the Leftmost Pointer?

- Create a *virtual* pair (k,p) for the leftmost pointer in each non-leaf node d
  - On the leftmost branch, k is $-\infty$
  - In other non-leaf nodes, k is taken from the last proper (i.e., non-virtual) pair that was used to reach d
- The virtual pairs are created when traversing the path from the root to a leaf
  - For insertion, no need to create the virtual pairs

Insertion Process

- We start by inserting a new pair (*k*,*p*) into a leaf
- If a node *d* becomes overfull
  - Create a new node *d′* on the right side of *d*, and
  - Move half of the pairs to the new node
- If *d′* is non-leaf, the key of its leftmost pair becomes virtual (i.e., is deleted after the next step)
- Create a pair (*k′*,*p′*), where *k′* is the smallest key in *d′* and *p′* is a pointer to *d′*
- Insert (*k′*,*p′*) into the parent of *d*, on the right side of the pair that points to *d*
  - (*k′*,*p′*) is never inserted in the leftmost position
- If the parent of *d* becomes overfull, continue recursively (may have to create a new root eventually)

Deletion from a B+tree

(a) Simple case - no example

(b) Merge with neighbor (sibling)

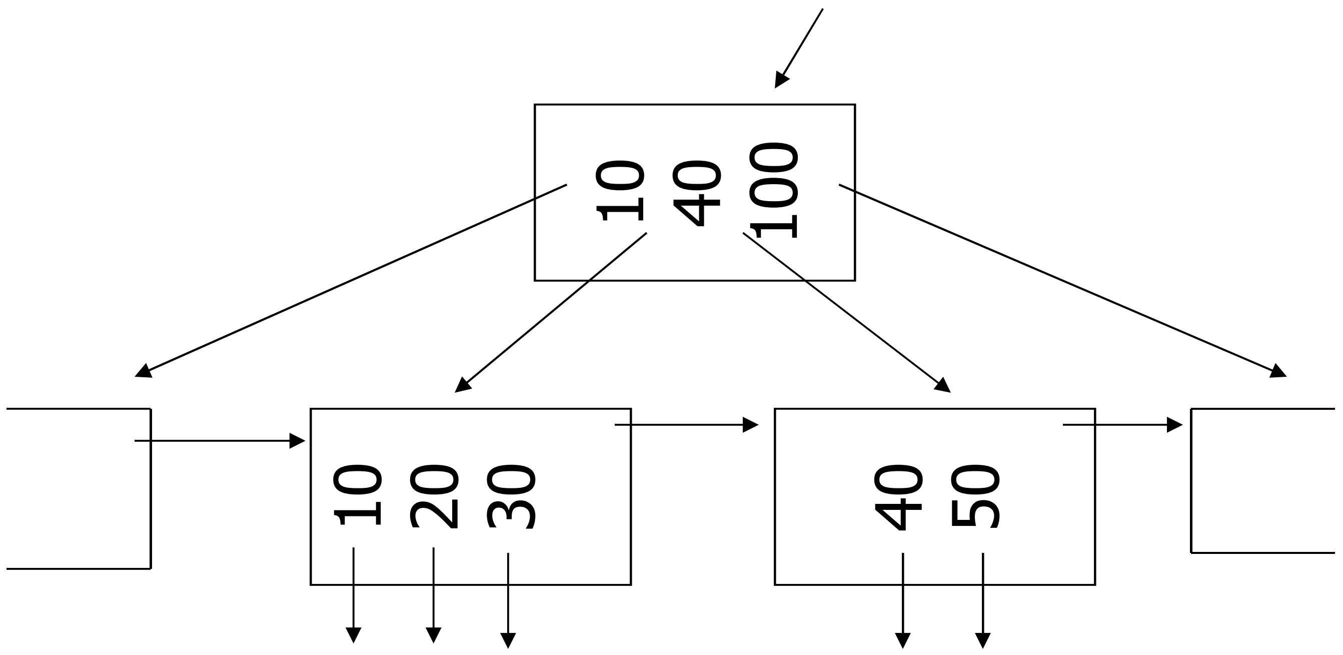
(c) Re-distribute keys

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

(b) Merging with sibling

– Delete 50

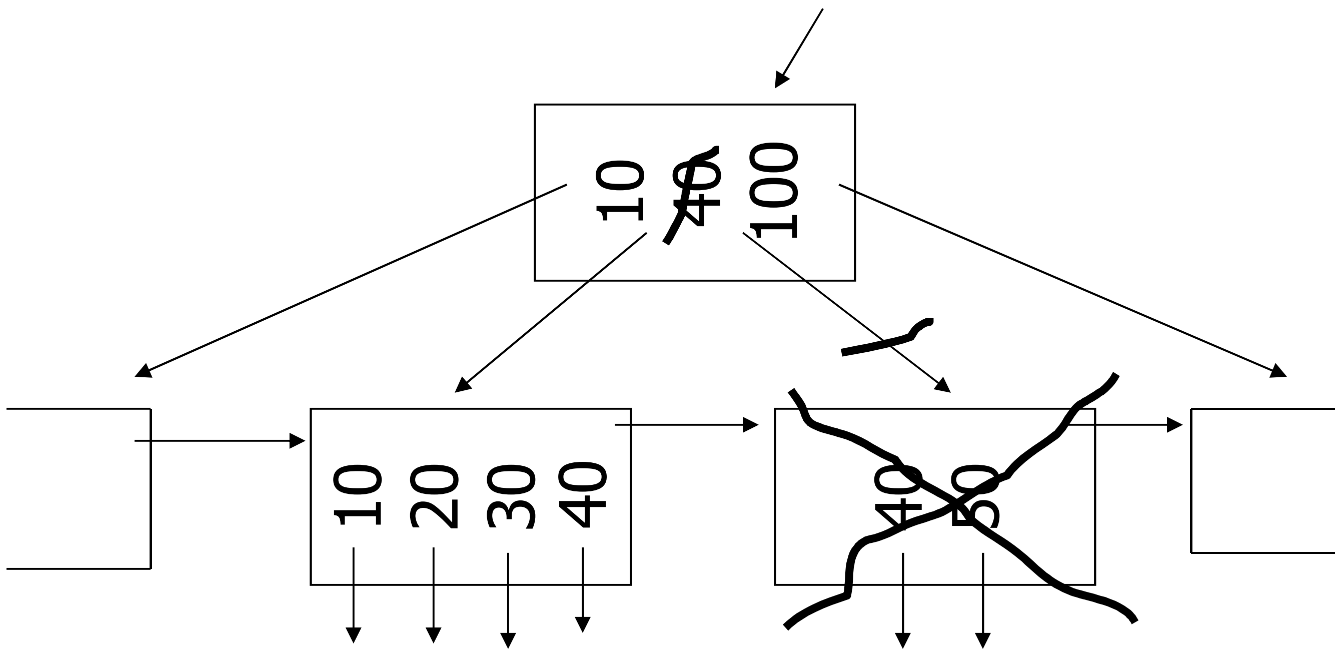
n=4



(b) Merging with sibling

– Delete 50

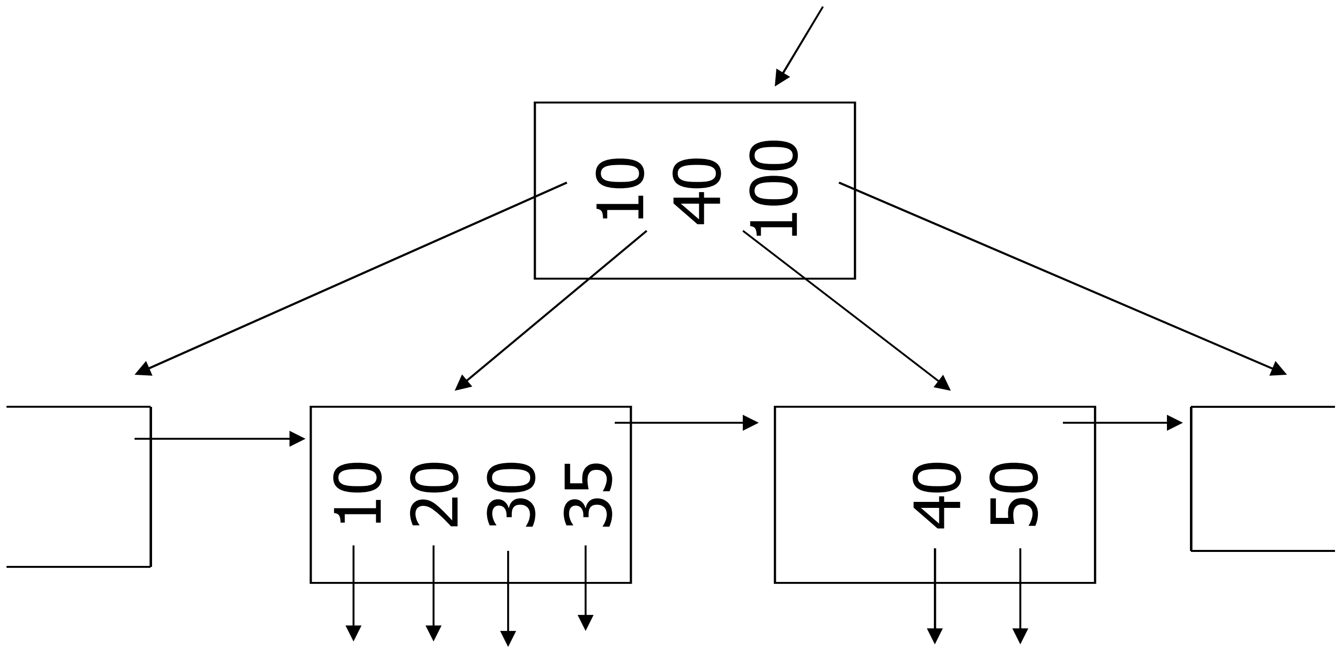
n=4



(c) Redistribute keys

– Delete 50

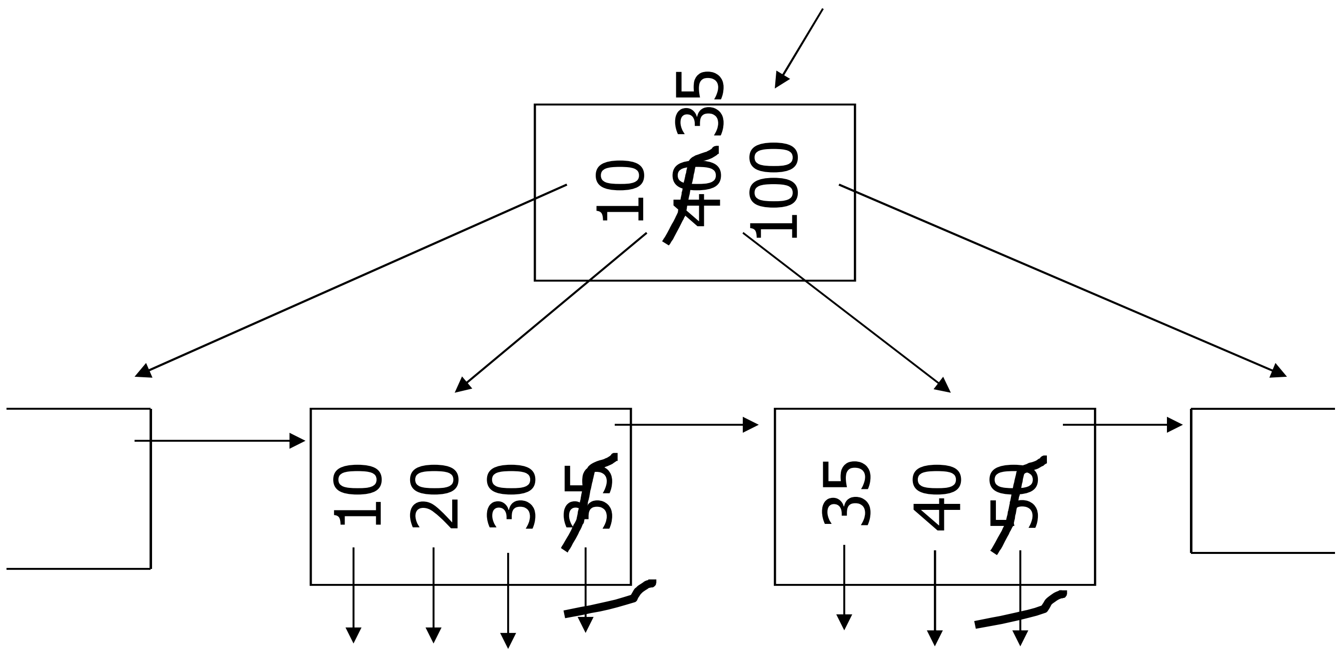
n=4



(c) Redistribute keys

– Delete 50

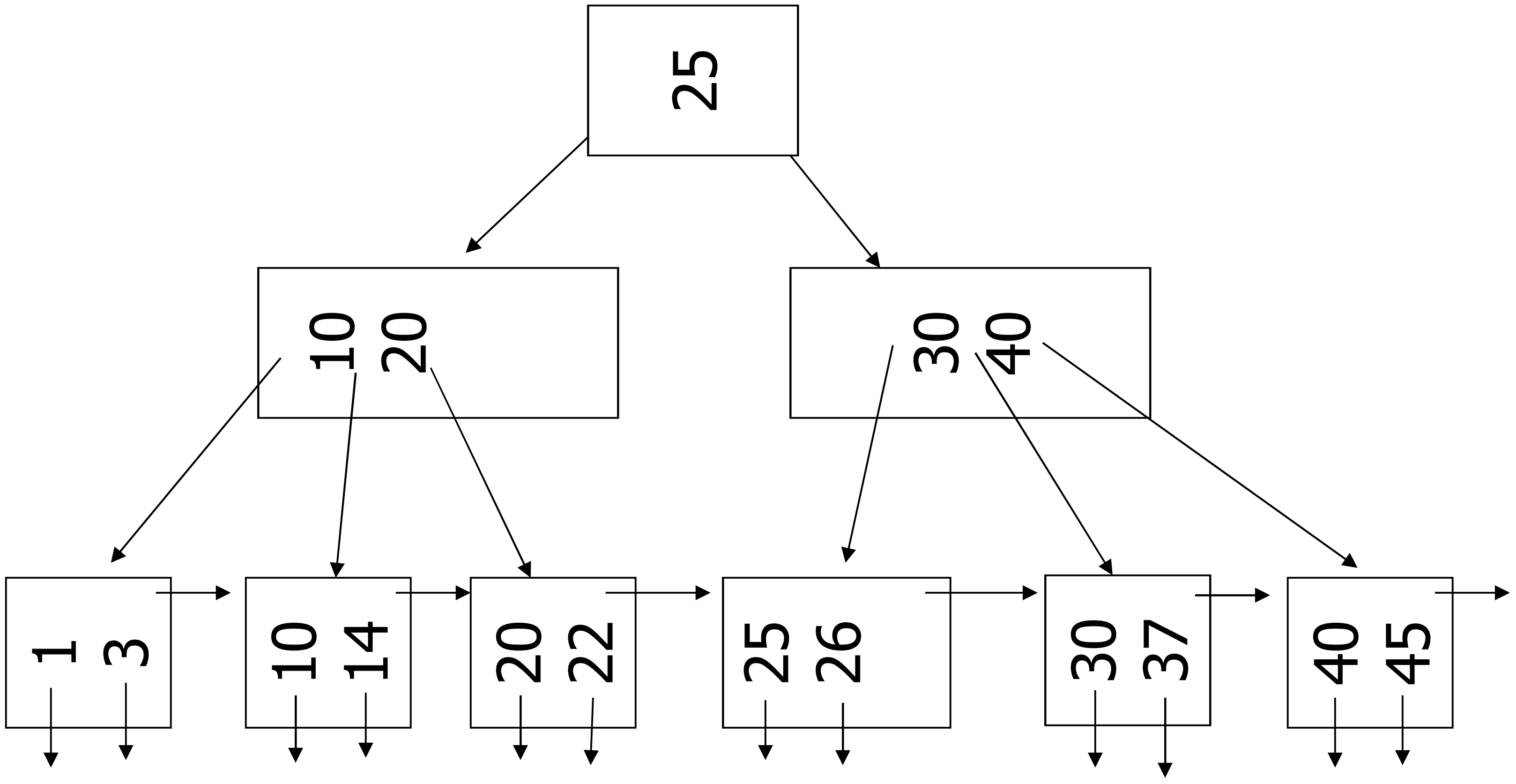
n=4



(d) Non-leaf merging

– Delete 37

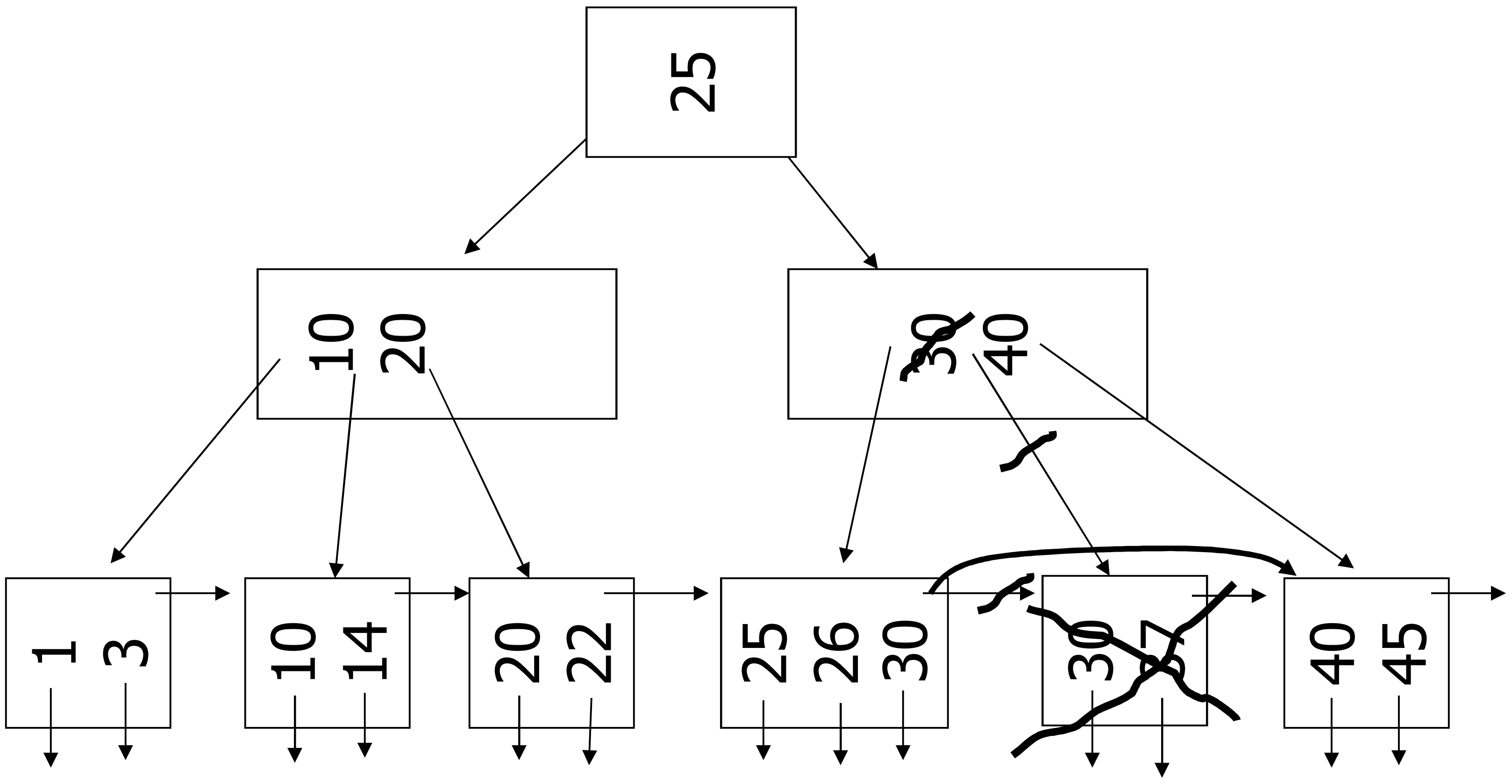
n=4



(d) Non-leaf merging

– Delete 37

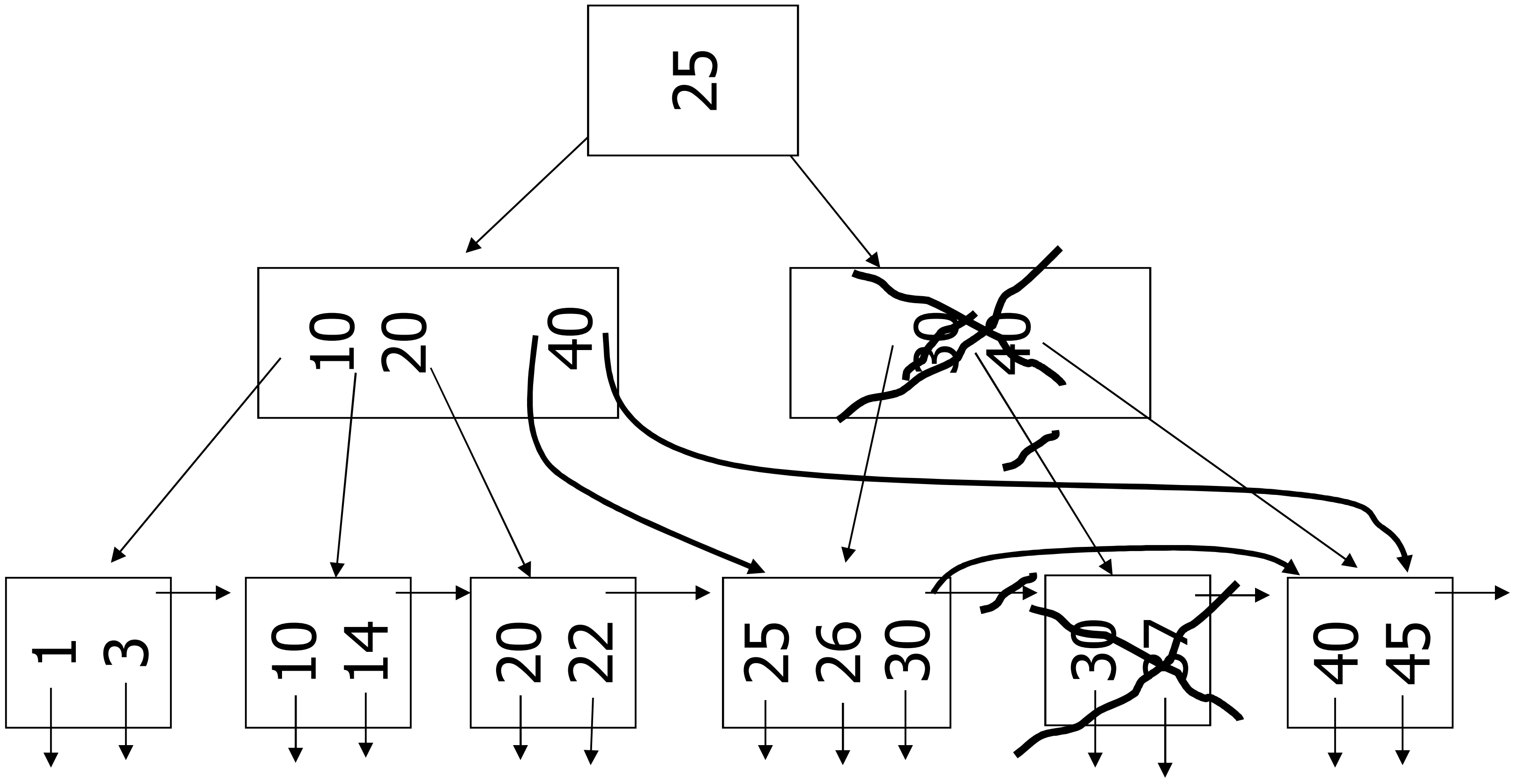
n=4



(d) Non-leaf merging

- Delete 37

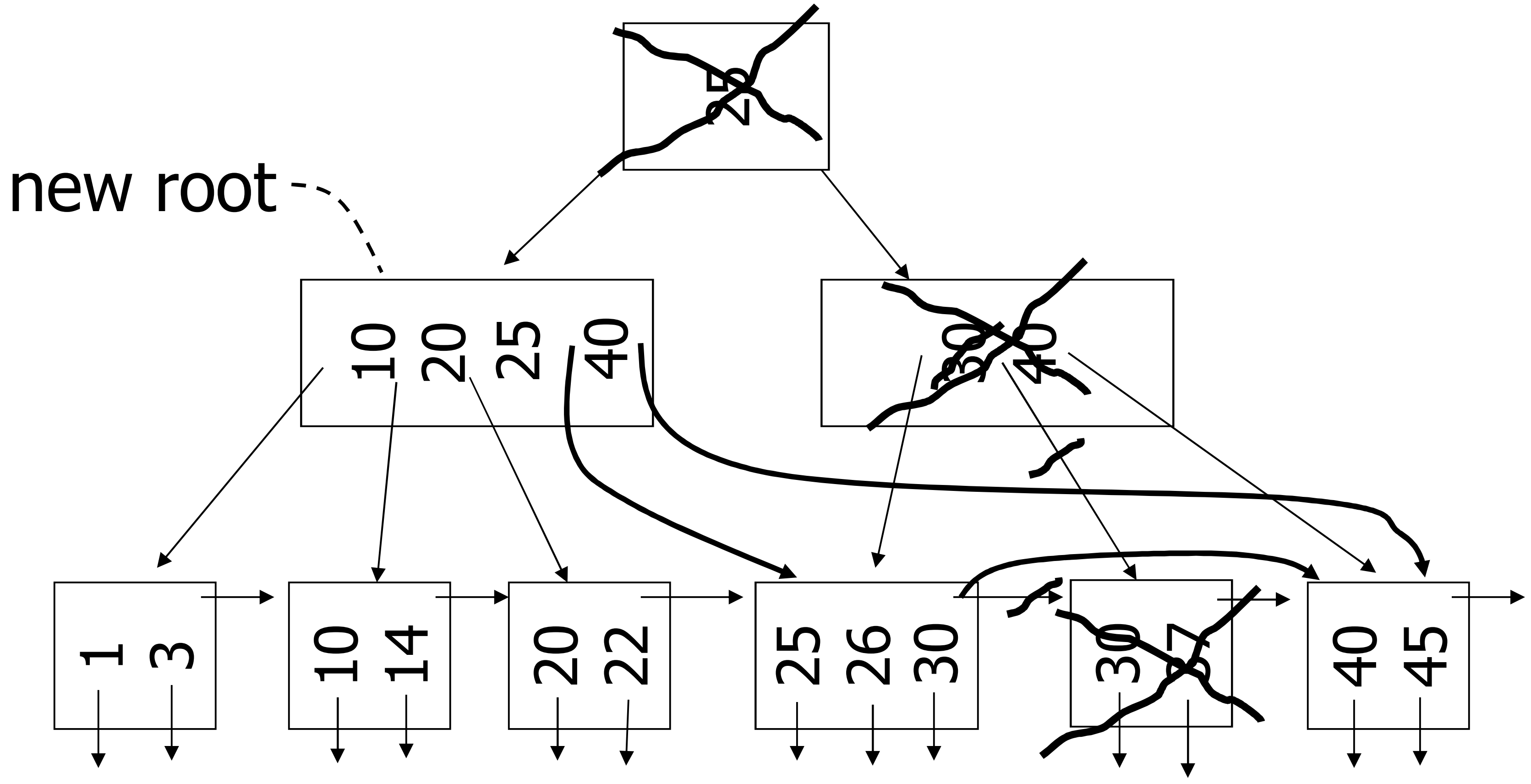
n=4



(d) Non-leaf merging

- Delete 37

n=4



B+Tree Deletions in Practice

- Often, merging is <u>not</u> implemented
  - Too hard and not worth it!
    - Files usually grow in size over time

B+Trees as Search Keys

- A B+Tree can implement either a primary or a secondary search key
- If it is a secondary search key, then the leaves store the records of a one-level dense index, namely,
  - Pairs of a key and a pointer
- If it is a primary search key, the leaves can store either a one-level index or the file itself
  - In the first option, which type of index is it?
  - In the second option, the records are longer than those stored in non-leaf nodes – is it a problem?

Answers on next slide

Suppose that a B+tree is a primary search key

- If the file is stored separately and is sorted on the primary key, then the bottom level of the B+tree is a one-level sparse index
- If the file is stored separately, but is not sorted (i.e., the file is a heap), then the bottom level must be a one-level dense index
- If the file is stored in the leaves of the B+tree, then the records in the leaves are longer than those in the non-leaf nodes
  - The parameter n (which determines the max and min number of records in the nodes) is different in the leaves (i.e., it is smaller than in the non-leaf nodes)

B+Trees vs. Dense & Sparse Indexes

- D & S indexes could be more efficient if
  - When creating or reorganizing, we leave enough space for future insertions
    - But we should not leave too much space or else we'll hurt performance
  - We reorganize before performance deteriorates too much
- But it is hard to know how much free space to leave and when to reorganize
  - Hence, B+trees are more efficient in practice

Comment

- When constructing a B+tree, we can
  - Make each level sequential
  - Leave some free space in each node so that future insertions will not destroy sequentiality immediately
- When performance deteriorates, we should reconstruct the B+tree
- Sequentiality is not such a great advantage of S & D indexes compared with B+trees

Buffering Requirements

- Since B+trees have no overflow blocks, they need to look at only one block from each level
- D & S indexes require larger and variable-size buffer space
- Also, only after reading a block, can we determine whether an overflow block also has to be read (and that overflow block is unlikely to be contiguous with the first block)
  - Render read-ahead buffering ineffective
- BTW, is LRU good for B+trees?

B Tree: A Variation on a B+Tree

- In a B Tree, some of the records (of the index or the file) are stored in non-leaf nodes
  - Why is it a good idea?
- Disadvantage of B trees
  - Two sizes of records in non-leaf nodes
    - For fixed-sized blocks, fanout of non-leaf nodes is smaller; hence, tree is deeper and lookup takes longer
  - Deletion is more complicated
  - Hard to chain the records (of the index or the file)