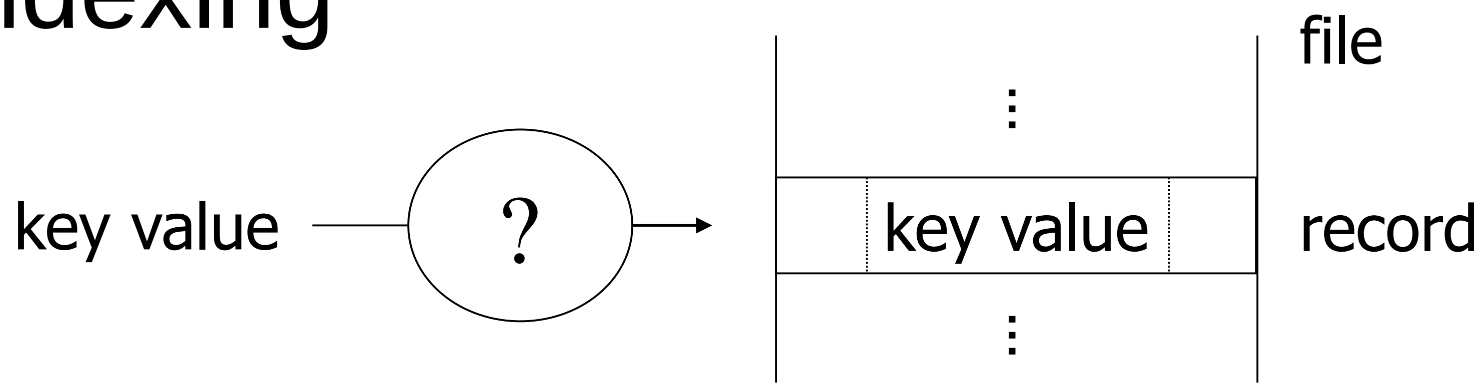
# PA152: Efficient Use of DB
# 4. Indexing

Vlastislav Dohnal

# Indexing




- Reason: faster access to records
  - than sequential (table) scan
- Variants:
  - Conventional indexes
  - B-tree
  - Hashing

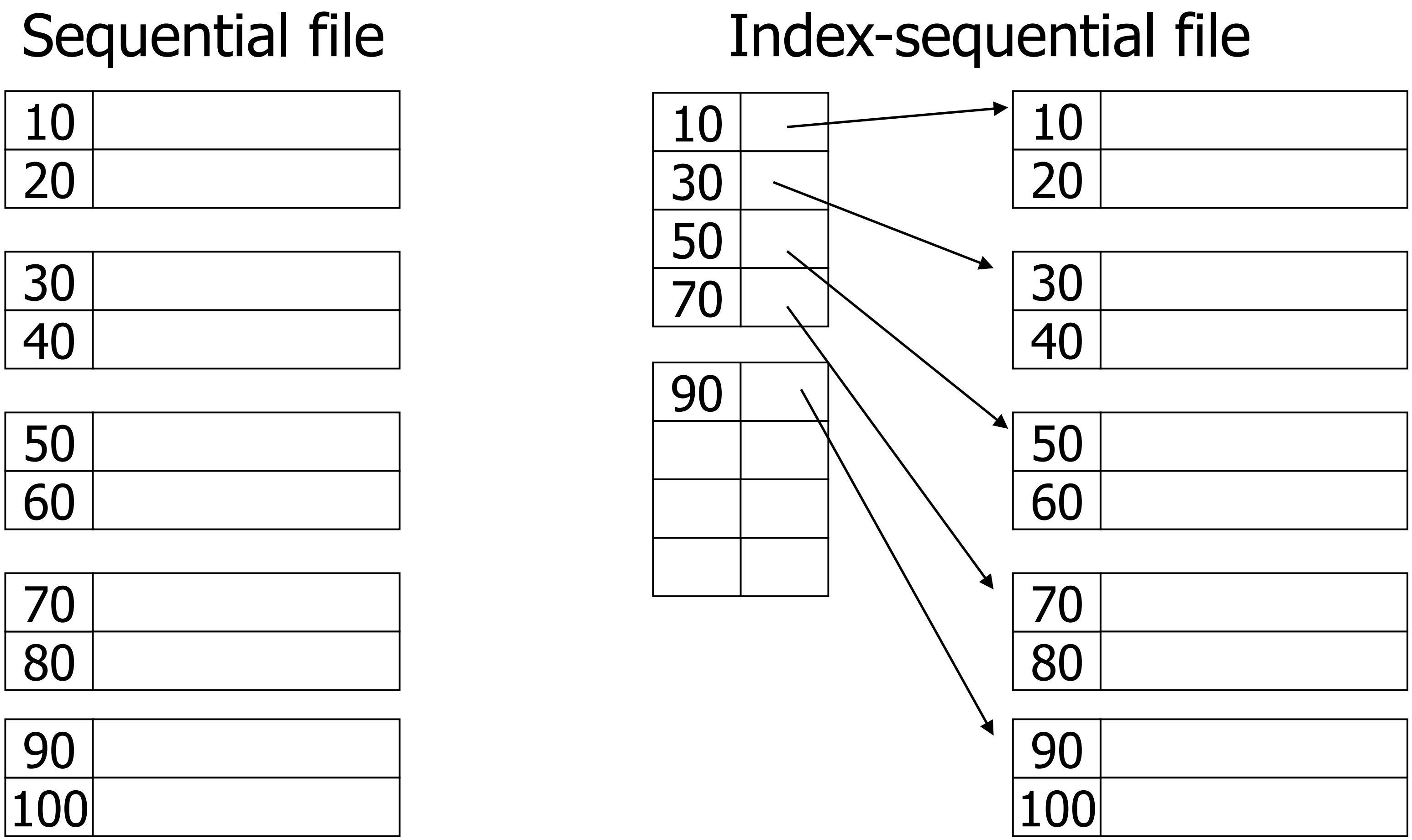
# Terminology

- Sequential file
  - Index-sequential file
- Search key
  - Primary key
- Index
  - Primary index
  - Secondary index

  - Dense index
  - Sparse index

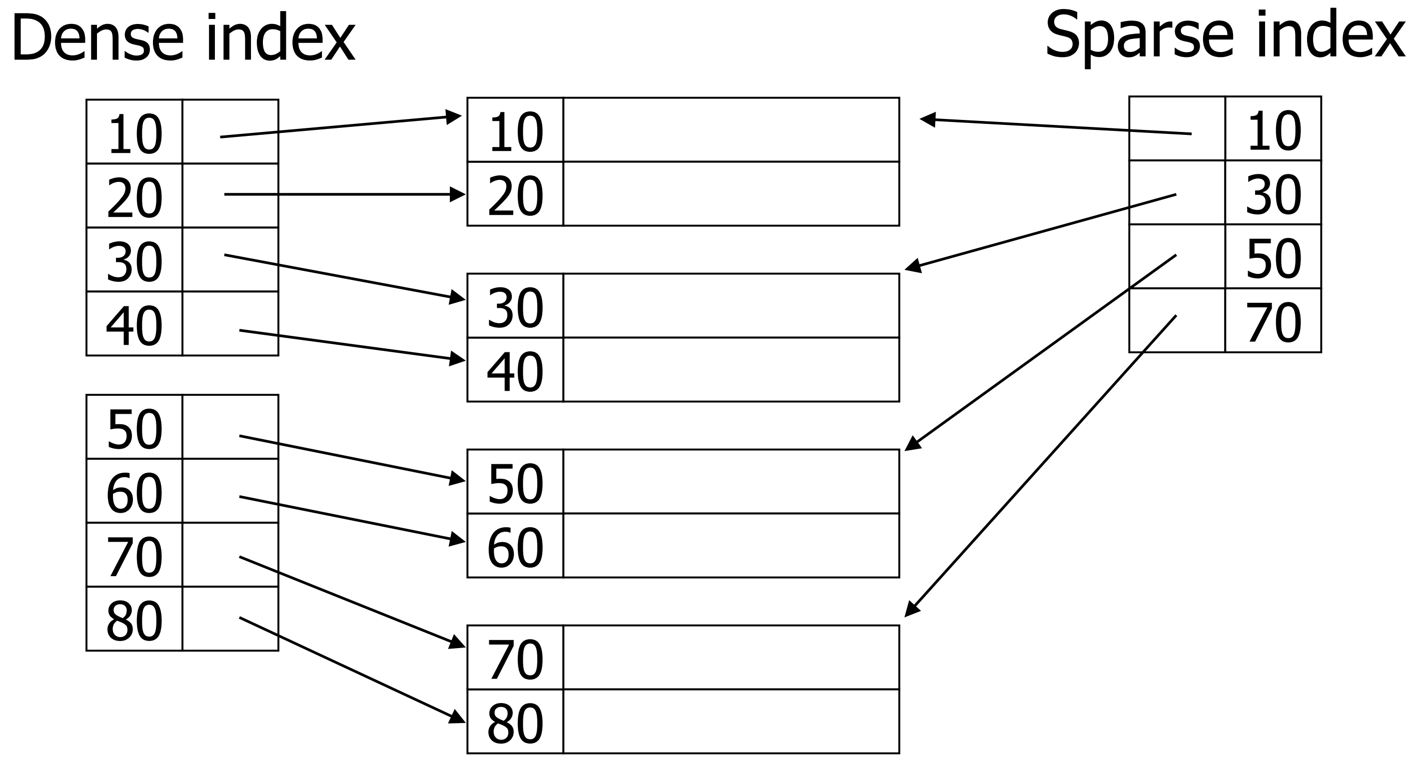
  - Multilevel index

# File


Sequential file
10
20
30
40
50
60
70
80
90
100
Index-sequential file
10
30
50
70
90
10
20
30
40
50
60
70
80
90
100

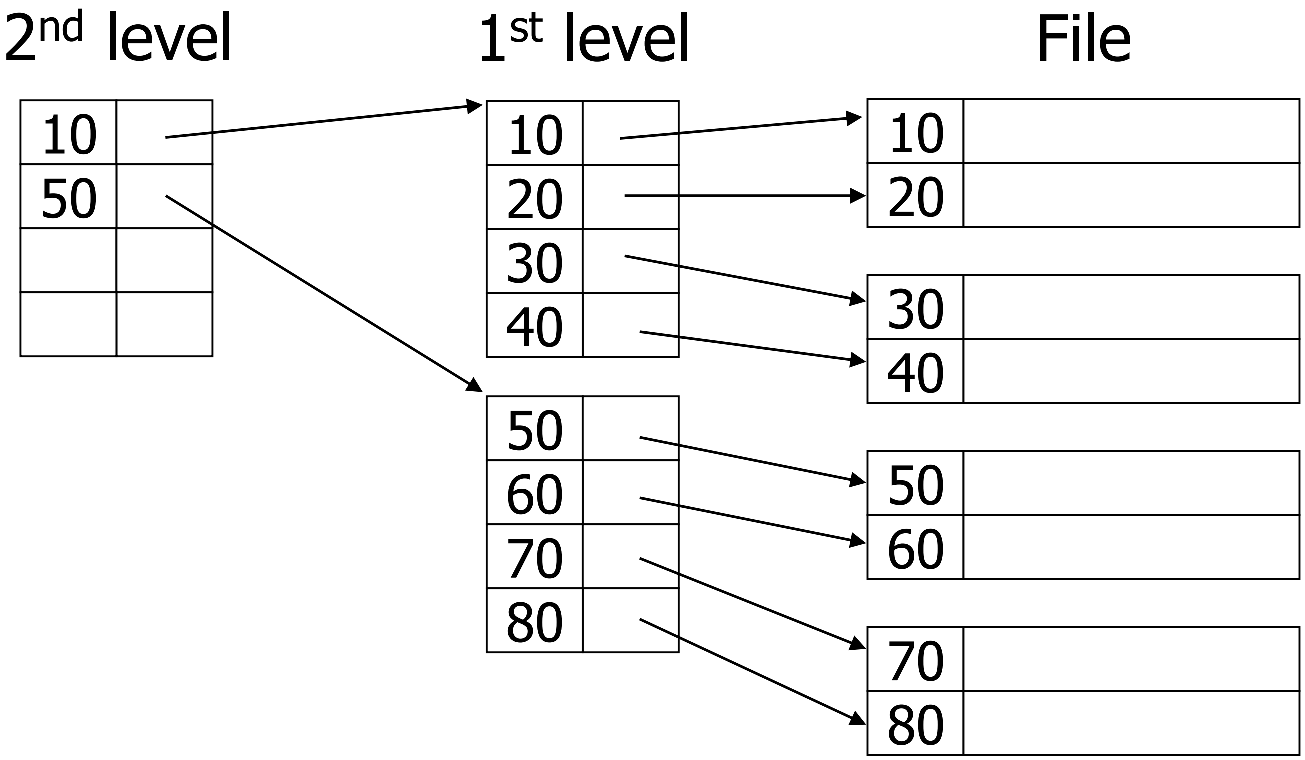
# Index

- Collection of items:
  - <key value, pointer to record/block>



# Index

- Multilevel index



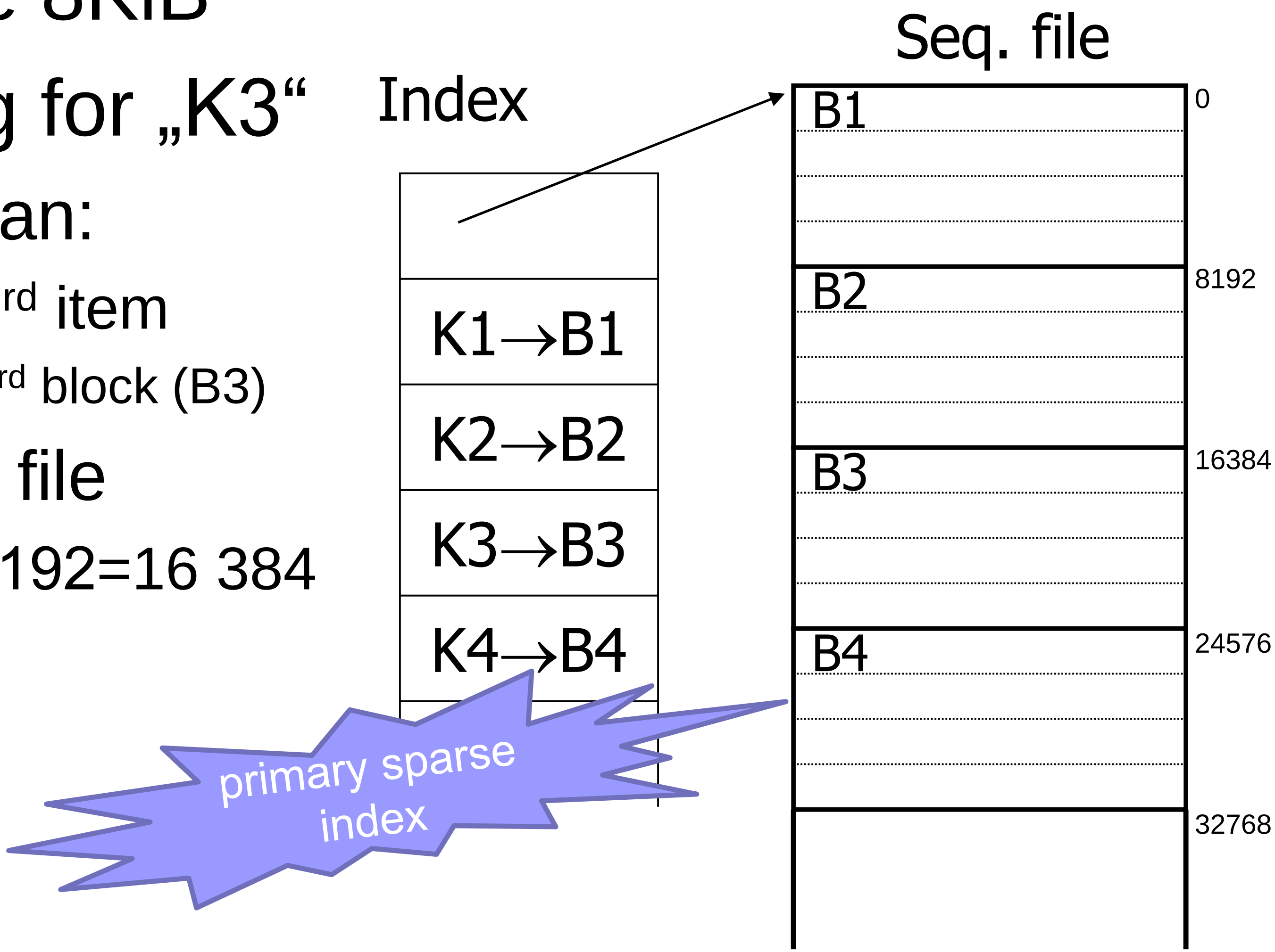

- Should indexes be dense in higher levels?

# Indices and Pointers

- Pointers in indexes
  - Pointer to records
    - Block addr. + record offset (index)
  - Pointer to block
    - Block addr. =
      - file ID + block number
  - File is contiguous and sequential
    - May to store pointers to blocks
      - use “implicit” pointers, i.e., can be computed
      - e.g., block number derived from the order of items in index

# Implicit Block Pointers

- Block size 8KiB
- Searching for „K3“
  - Index scan:
    - K3 in 3rd item
      - → 3rd block (B3)
  - Offset in file
    - (3-1)·8192=16 384



# Duplicate Keys

- Index type
  - dense index?
  - sparse index?

File

| | |
|---|---|
| 10 | |
| 10 | |

| | |
|---|---|
| 10 | |
| 20 | |

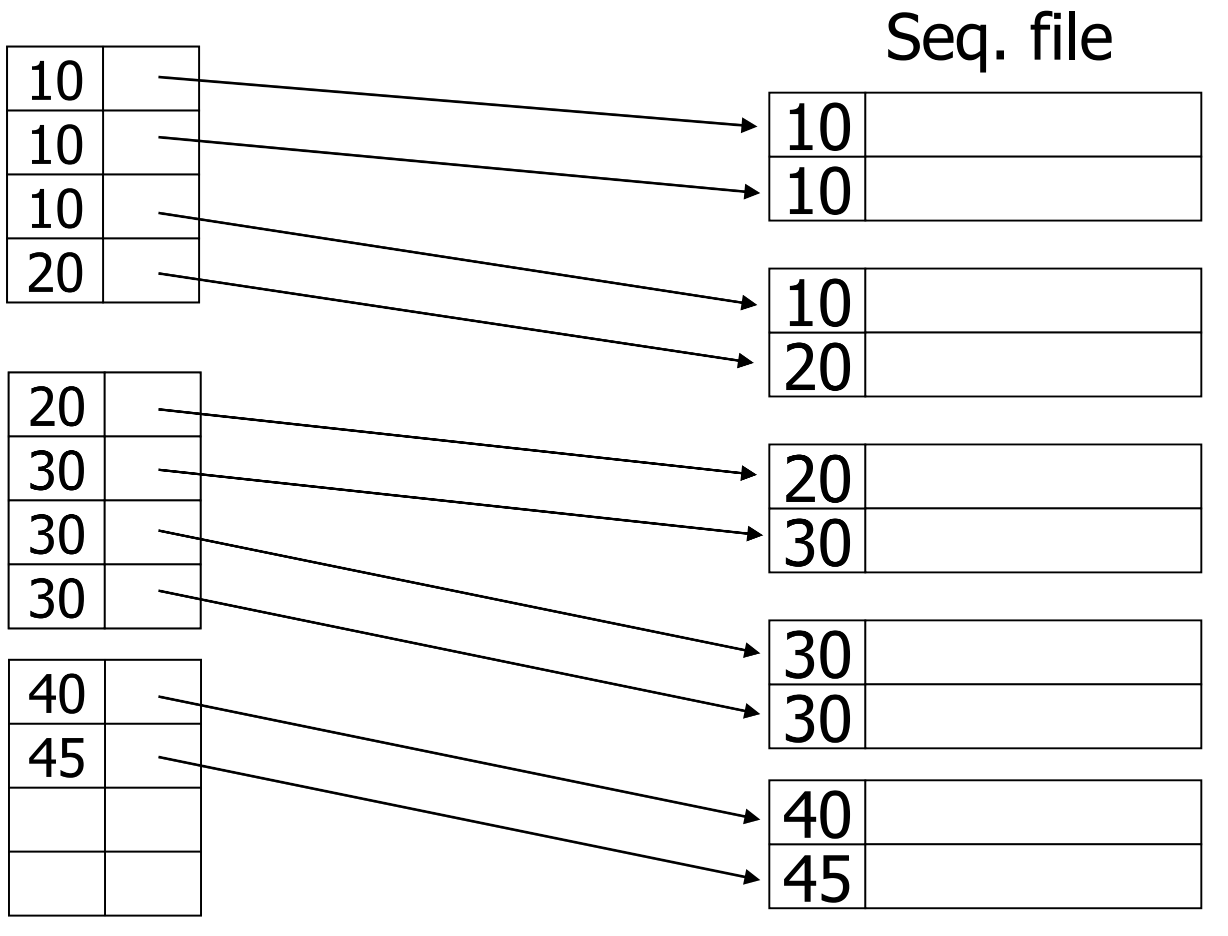
| | |
|---|---|
| 20 | |
| 30 | |

| | |
|---|---|
| 30 | |
| 30 | |

| | |
|---|---|
| 40 | |
| 45 | |

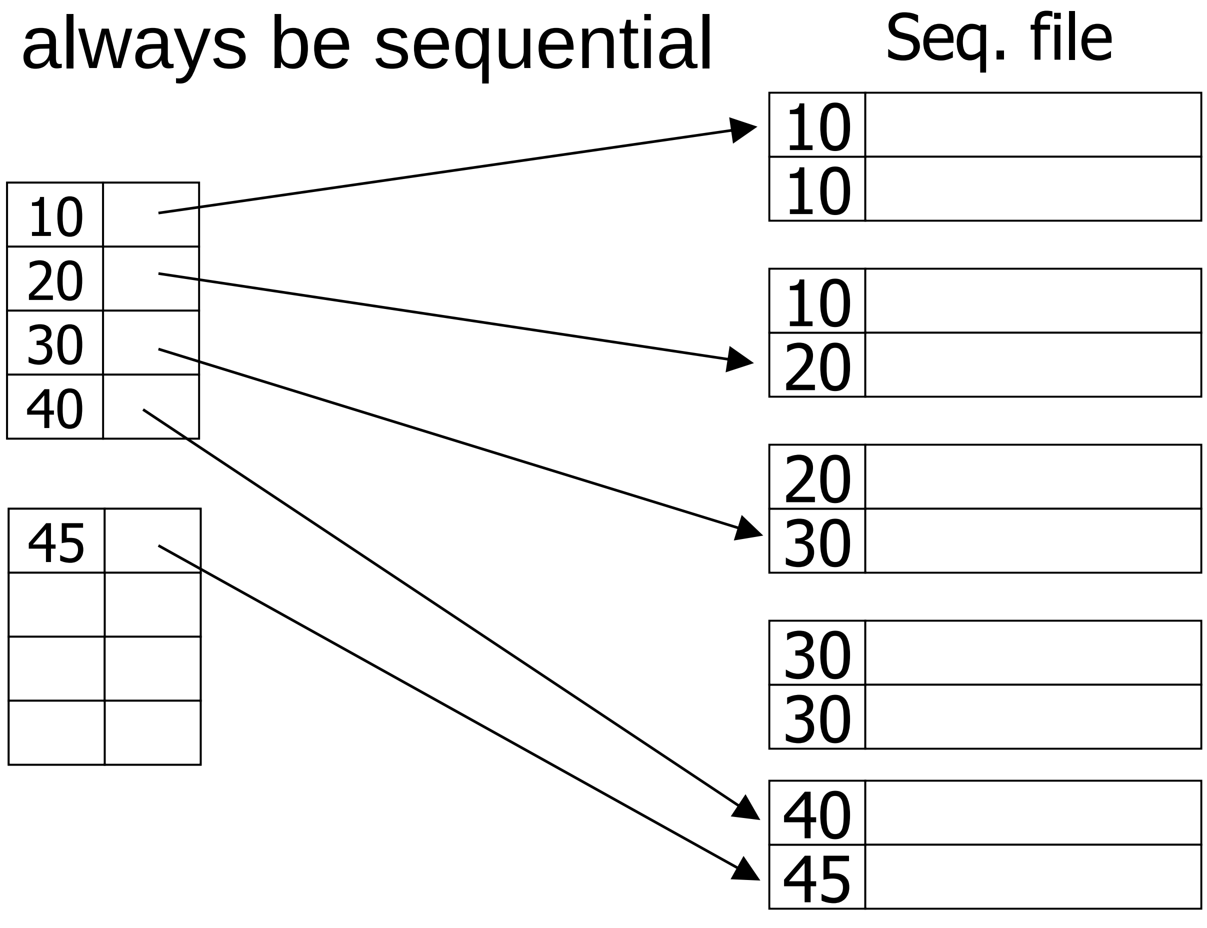
# Duplicate Keys: Dense Index

- Duplicate values in primary index



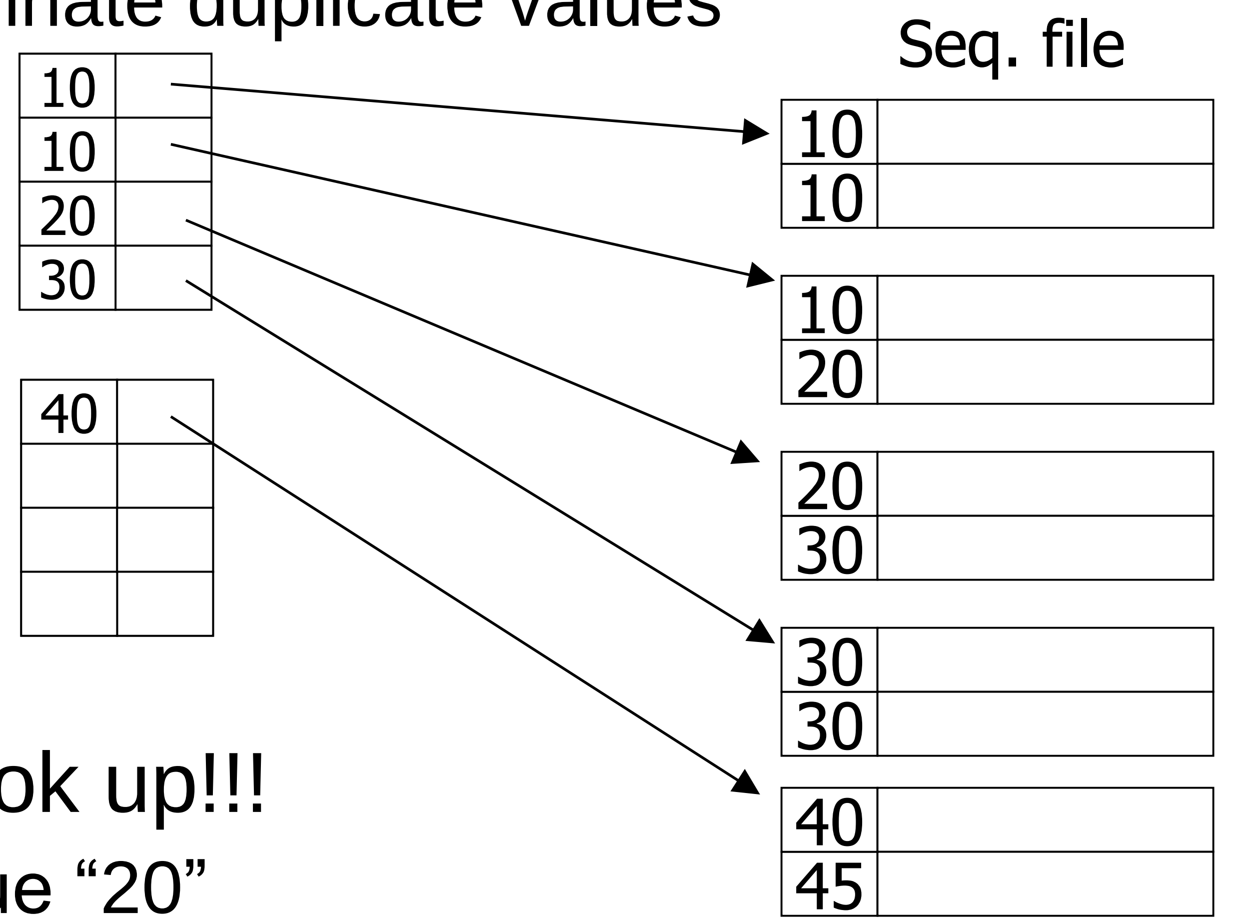
# Duplicate Keys: Dense Index

- Values in primary index are unique
  - File must always be sequential



# Duplicate Keys: Sparse Index

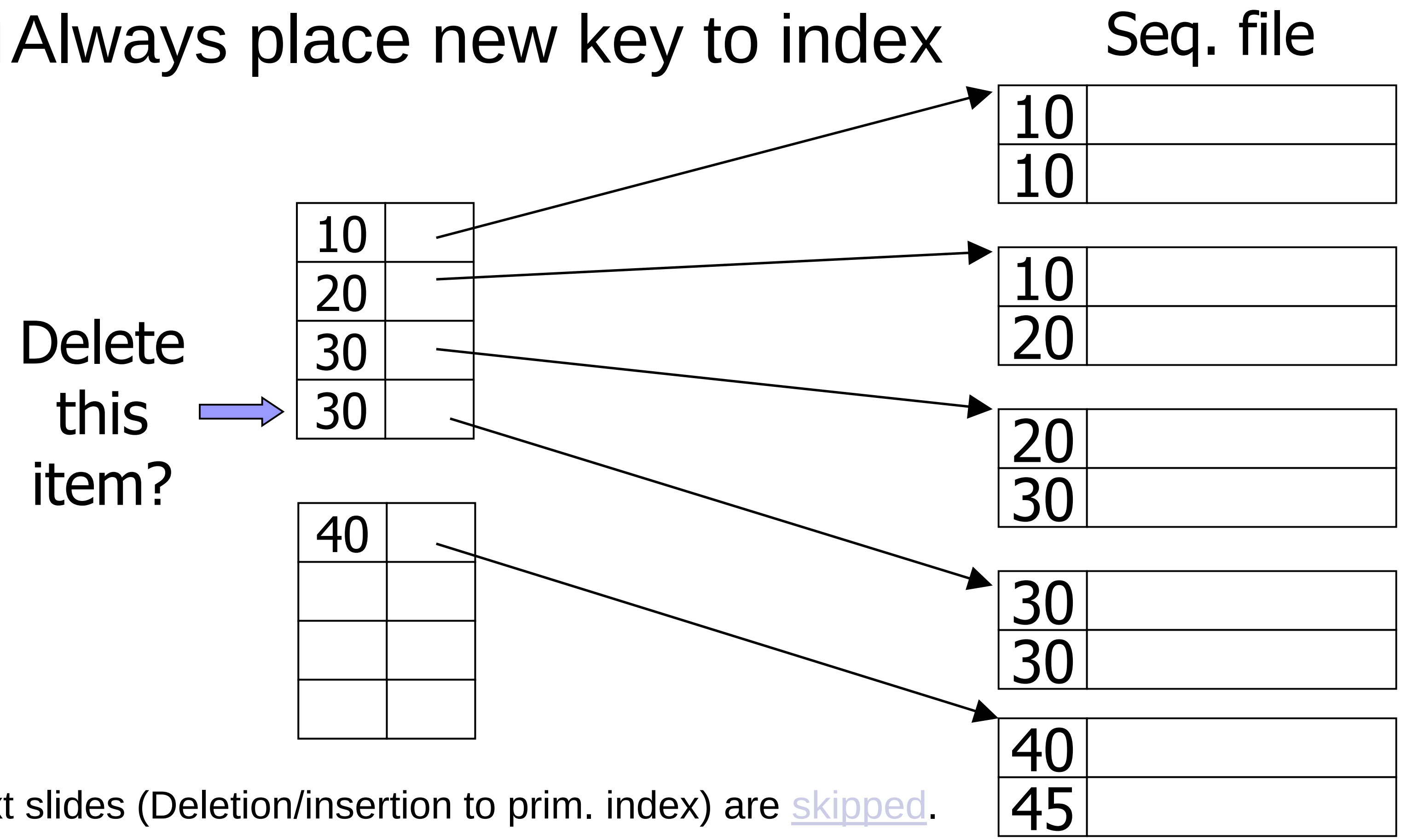
- Pointers with the first value in the block
  - Can eliminate duplicate values




- Record look up!!!
  - Find value “20”

# Duplicate Keys: Sparse Index

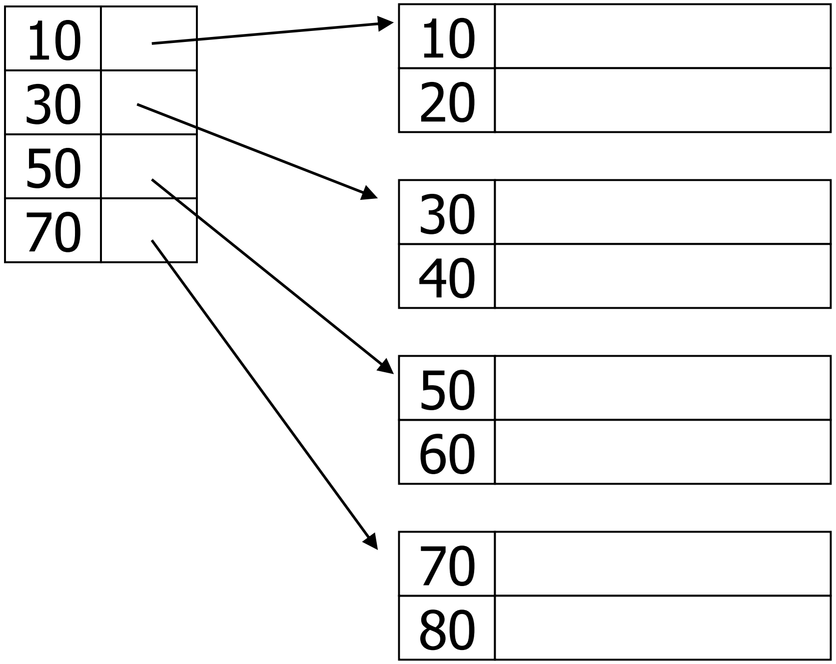
- Pointers with new value in block
  - Always place new key to index




- Next slides (Deletion/insertion to prim. index) are skipped.

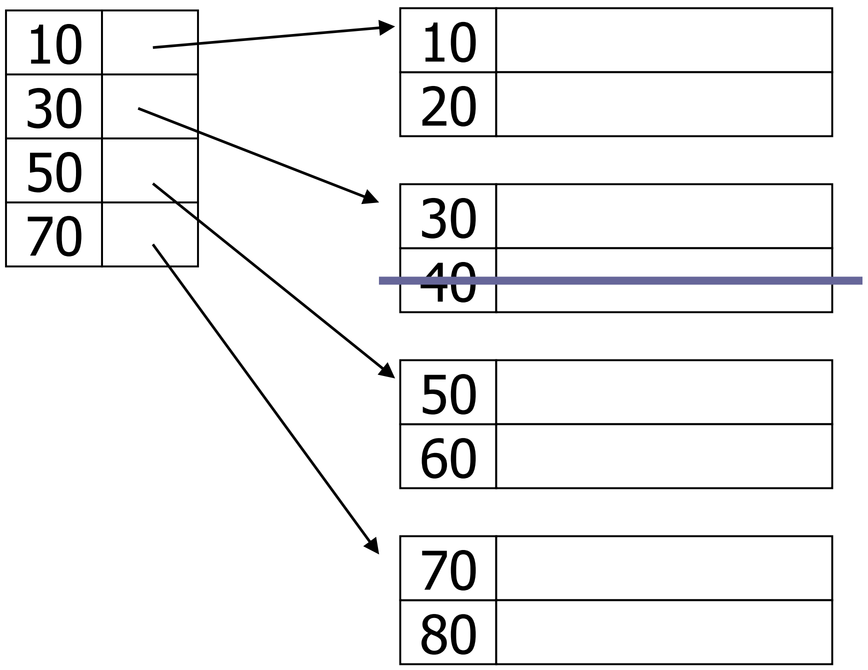
# Deletion from Index

- Sparse index
  - Delete record with key 40



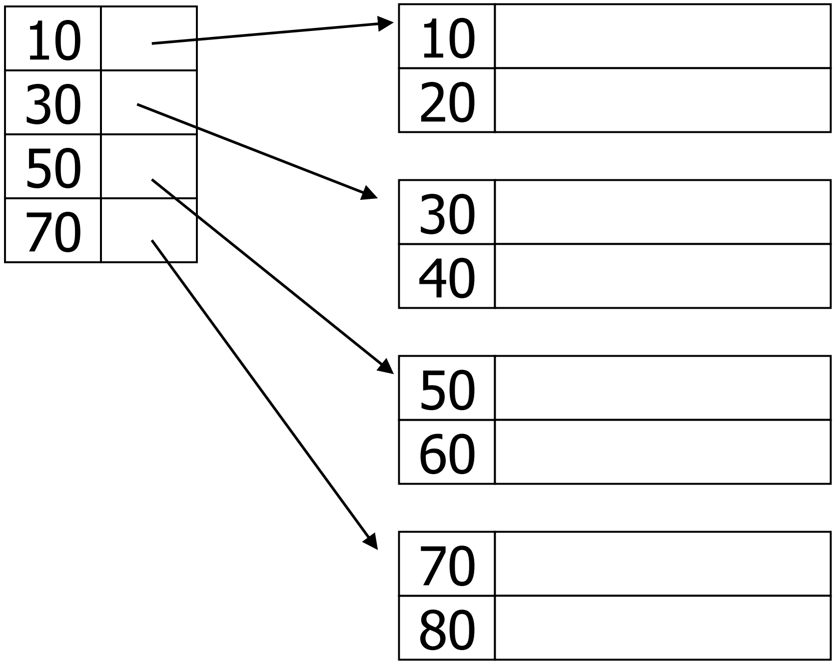
# Deletion from Index: Result

- Sparse index
  - After deletion of 40



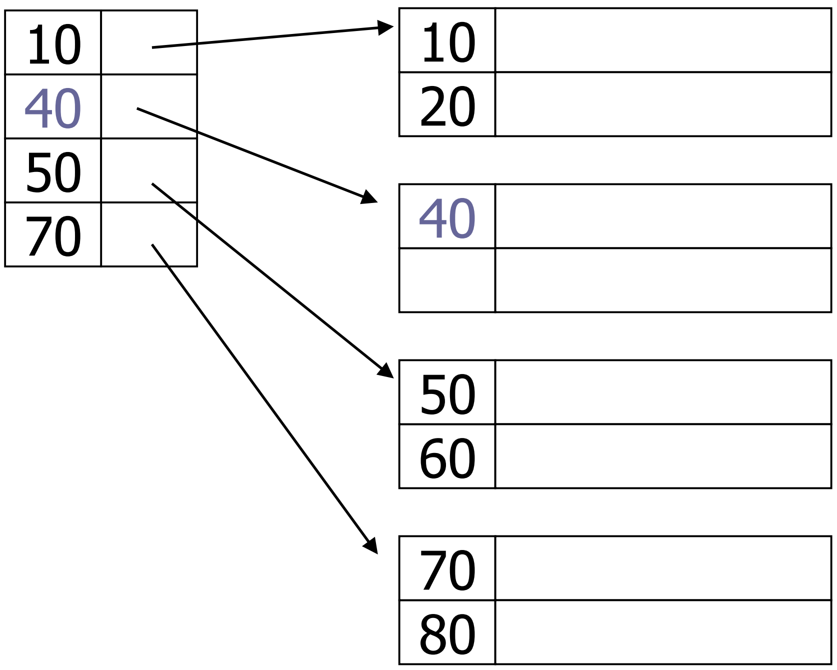
# Deletion from Index

- Sparse index
  - Delete record with key 30



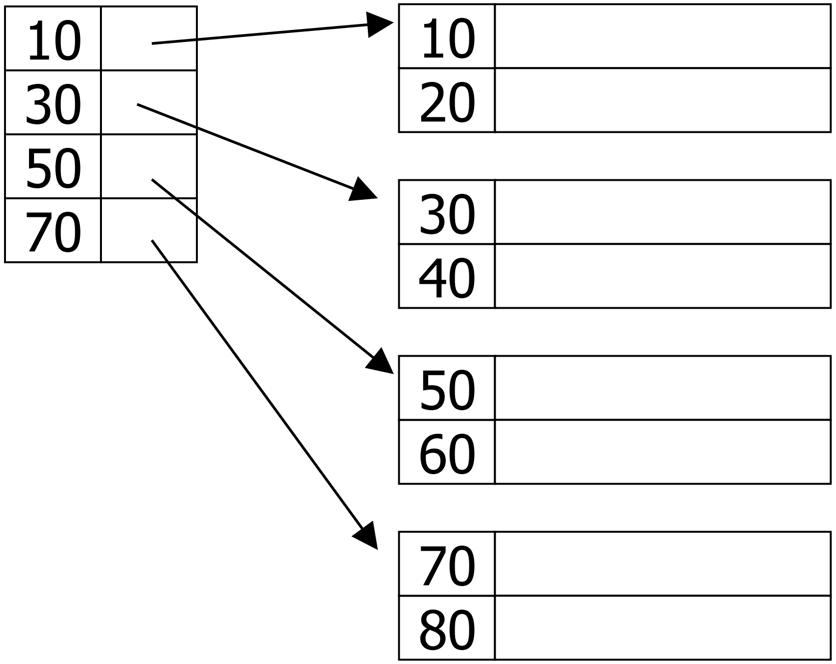
# Deletion from Index: Result

- Sparse index
  - After deletion of record30
    - New value in block changed, so update index



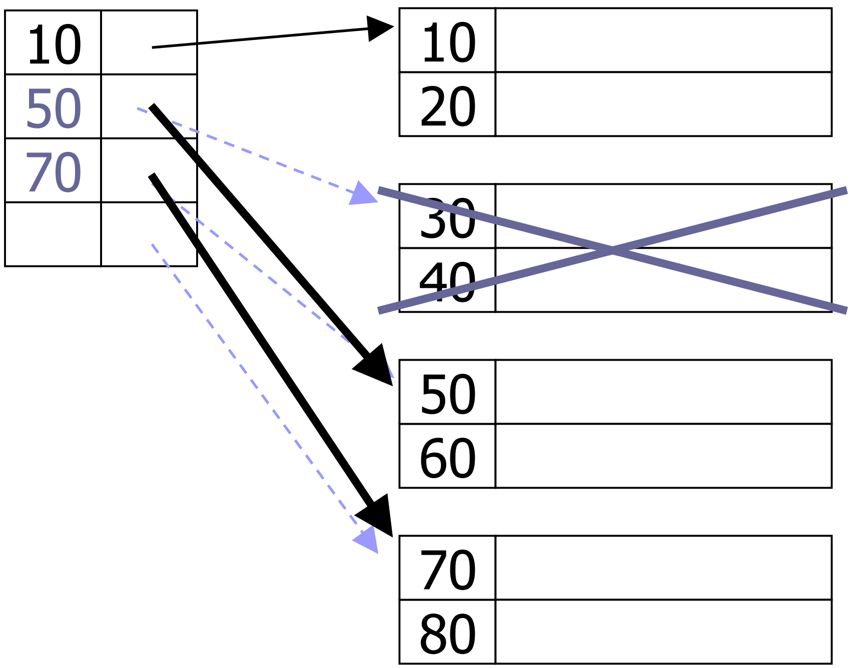
# Deletion from Index

- Sparse index
  - Delete records 30 and 40



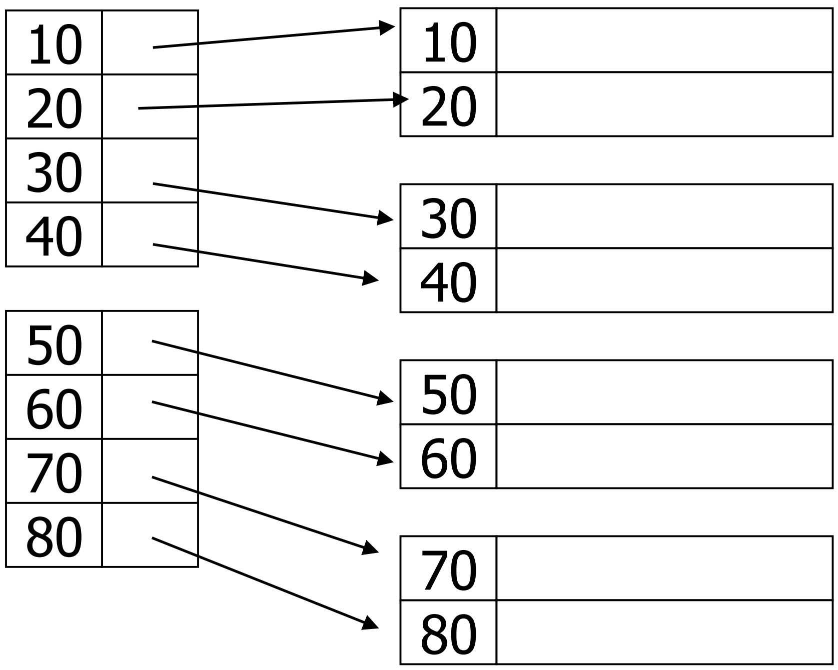
# Deletion from Index: Result

- Sparse index
  - After deletion of records 30 and 40
    - Block reclaimed, so update index



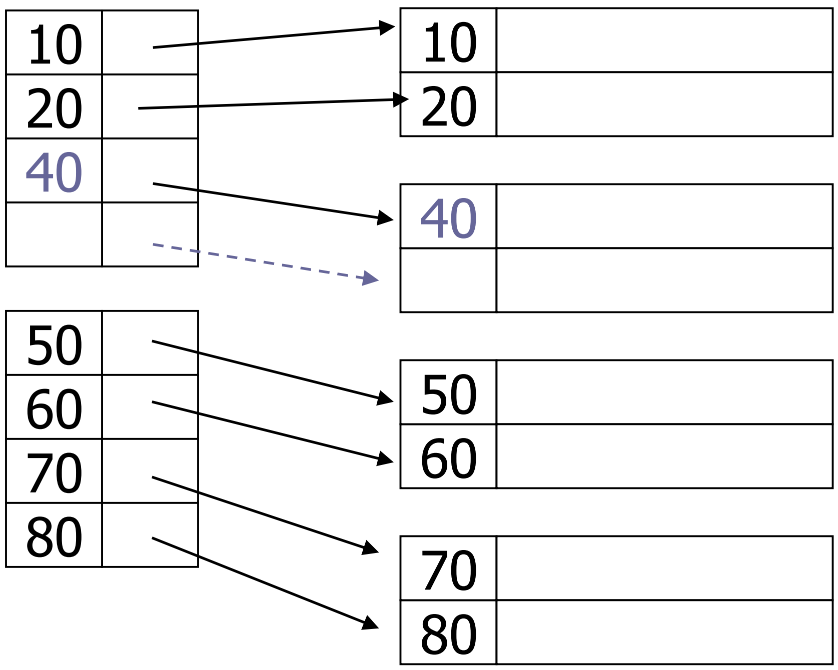
# Deletion from Index

- Dense index – always update index
  - Delete record with key 30



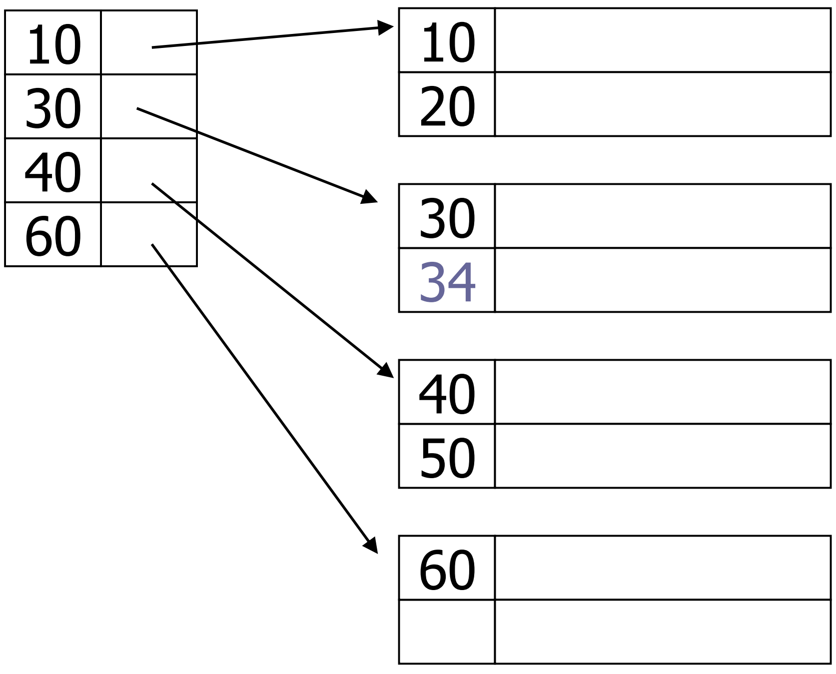
# Deletion from Index: Result

- Dense index
  - After deletion of record 30



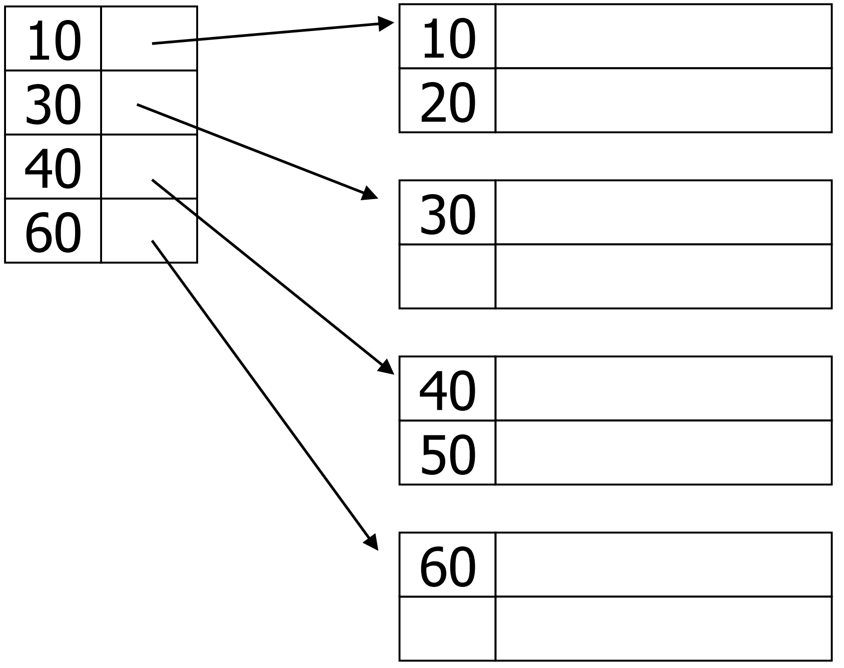
# Insertion to Index

- Sparse index
  - Insert record 34
    - Free space → no reorganization



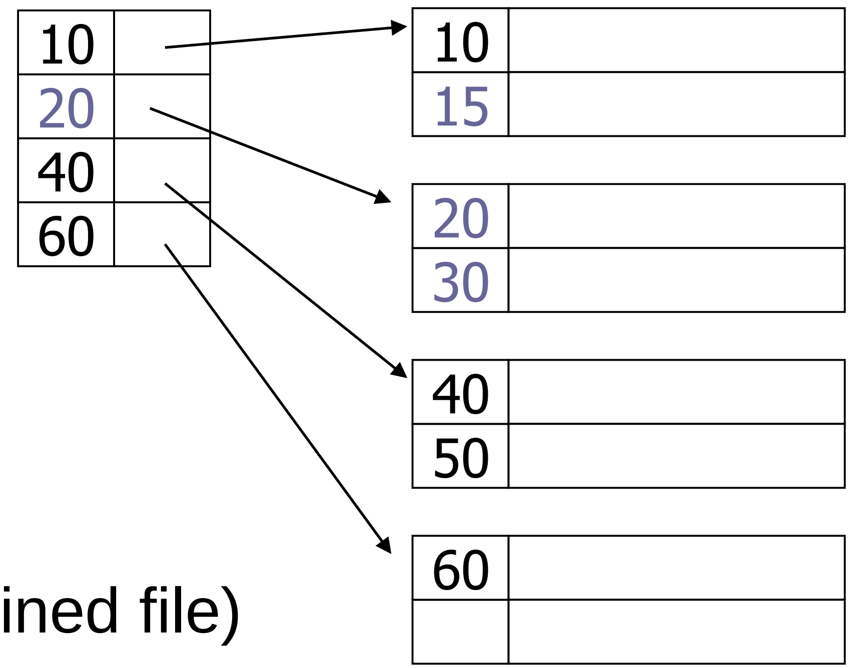
# Insertion to Index

- Sparse index
  - Insert record with key 15
    - No free space → reorganize immediately



# Insertion to Index

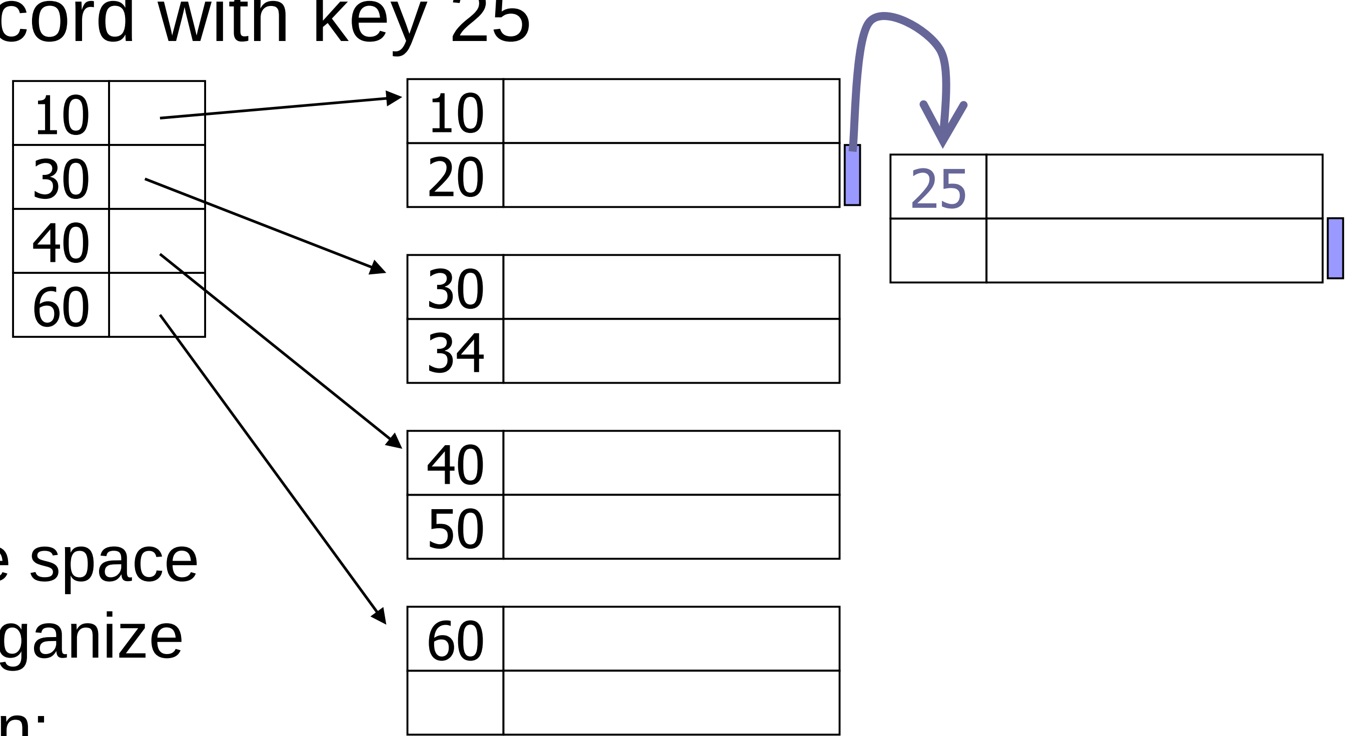
- Sparse index
  - Insert record with key 15
    - No free space → reorganize immediately
    - Solution: move some records to next block




  - Variation:
    - insert new block (chained file)
    - may corrupt implicit pointers

# Insertion to Index

- Sparse index
  - Insert record with key 25




- No free space
  → reorganize
- Solution:
  - allocate overflow block
  - Reorganize record into main file later

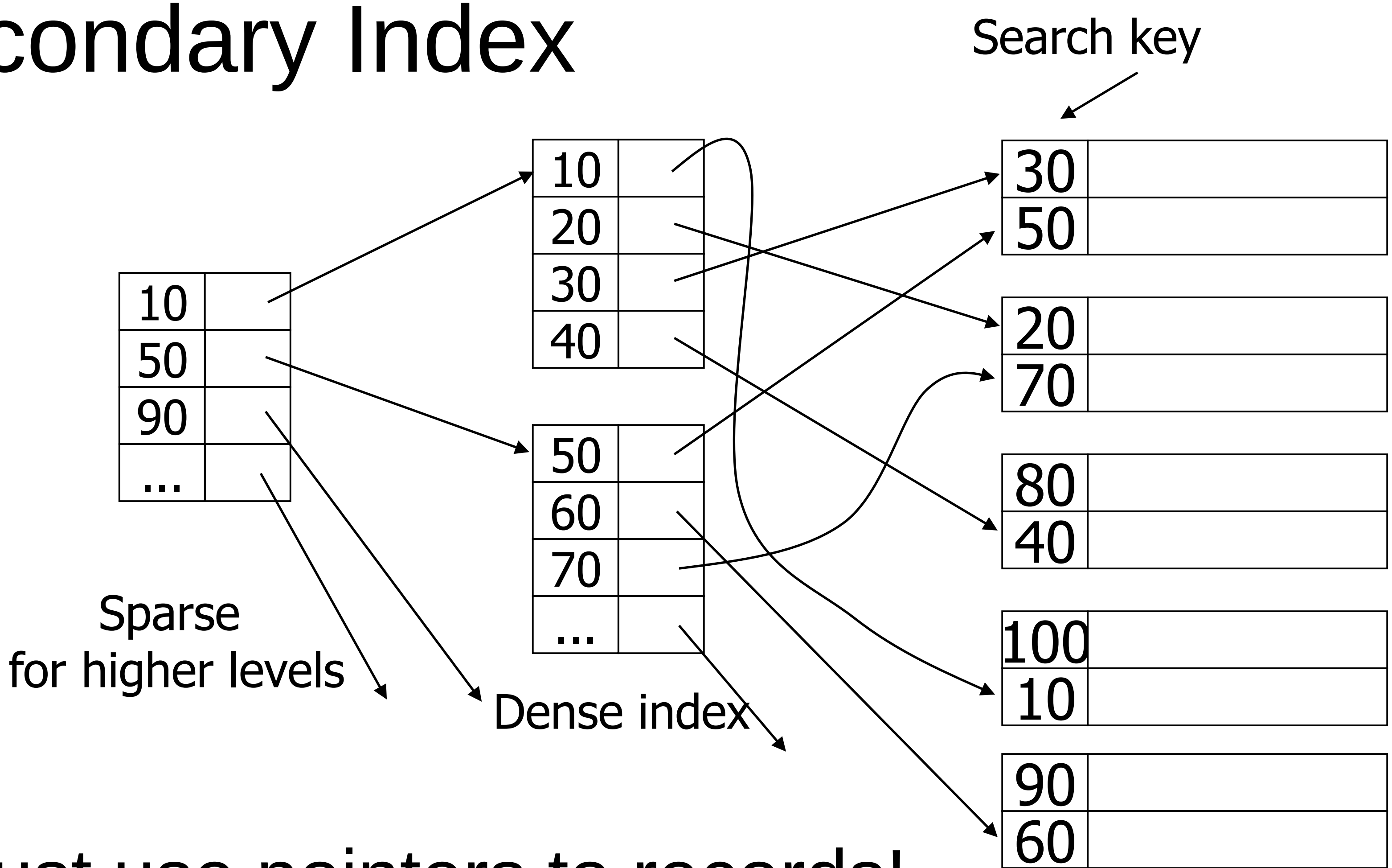
# Insertion to Index

- Dense index
  - Insert record
    - Update index – insert new item
    - Update file – by analogy to file update in sparse index case

# Secondary Index

- File ordered by another key
  - i.e., index created for different key than the primary file
  - Or the file is not ordered at all
- Which type:
  - Dense or sparse?

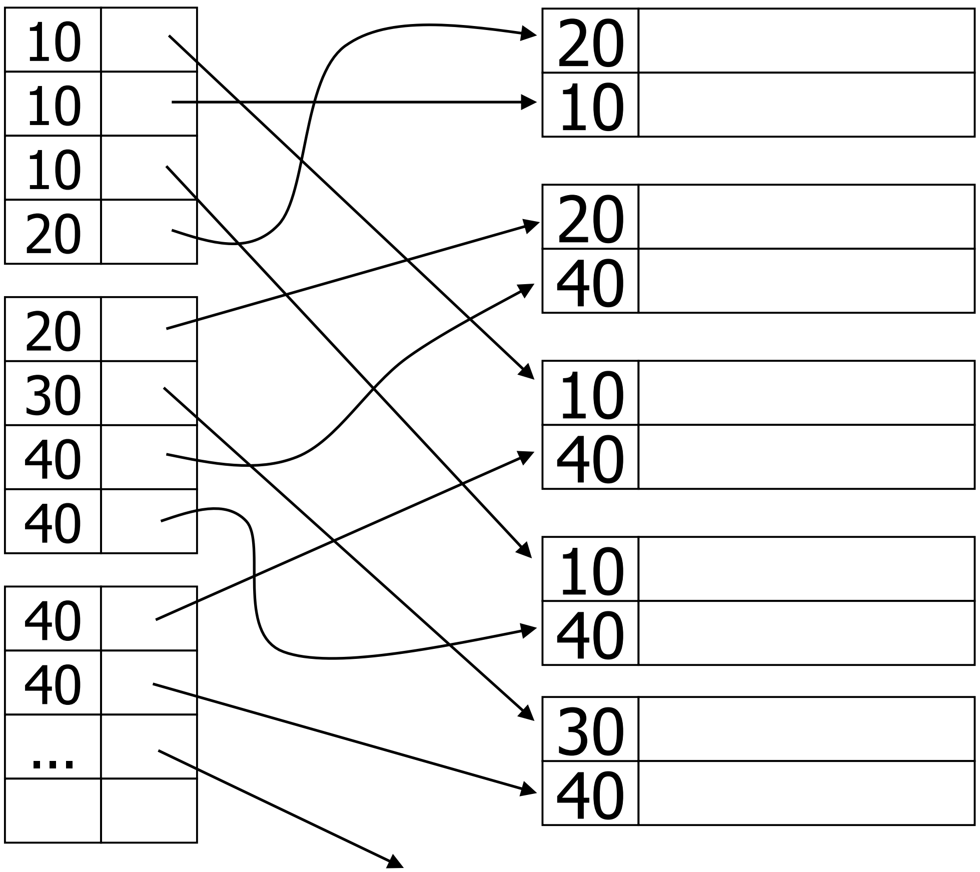
# Secondary Index




- Must use pointers to records!

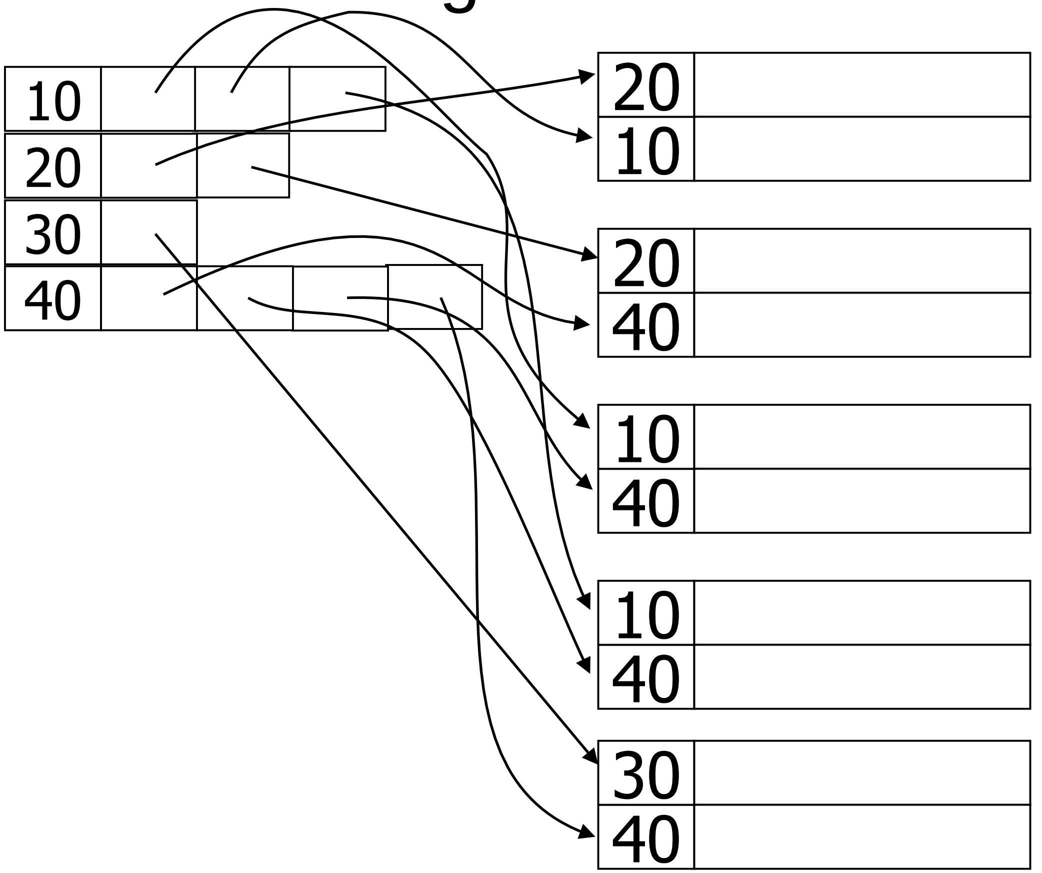
# Secondary Index: Duplicate Keys

- Replicated in index
  - Increases
    - space requirements
    - access time



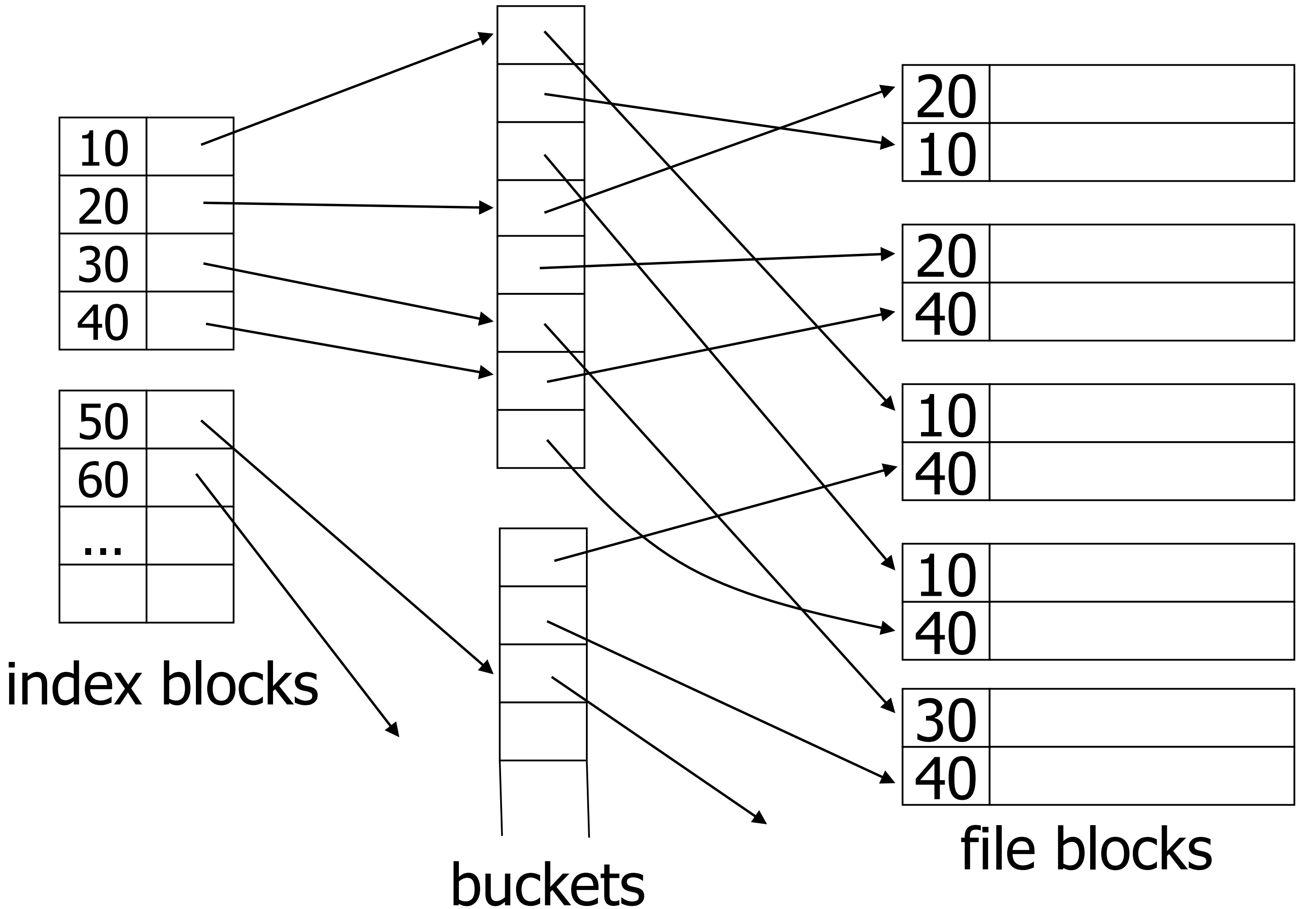
# Secondary Index: Duplicate Keys

- Index item contains list of pointers
  - But the item is of variable length



# Secondary Index: Duplicate Keys

- Shift the variable-length list to “buckets”



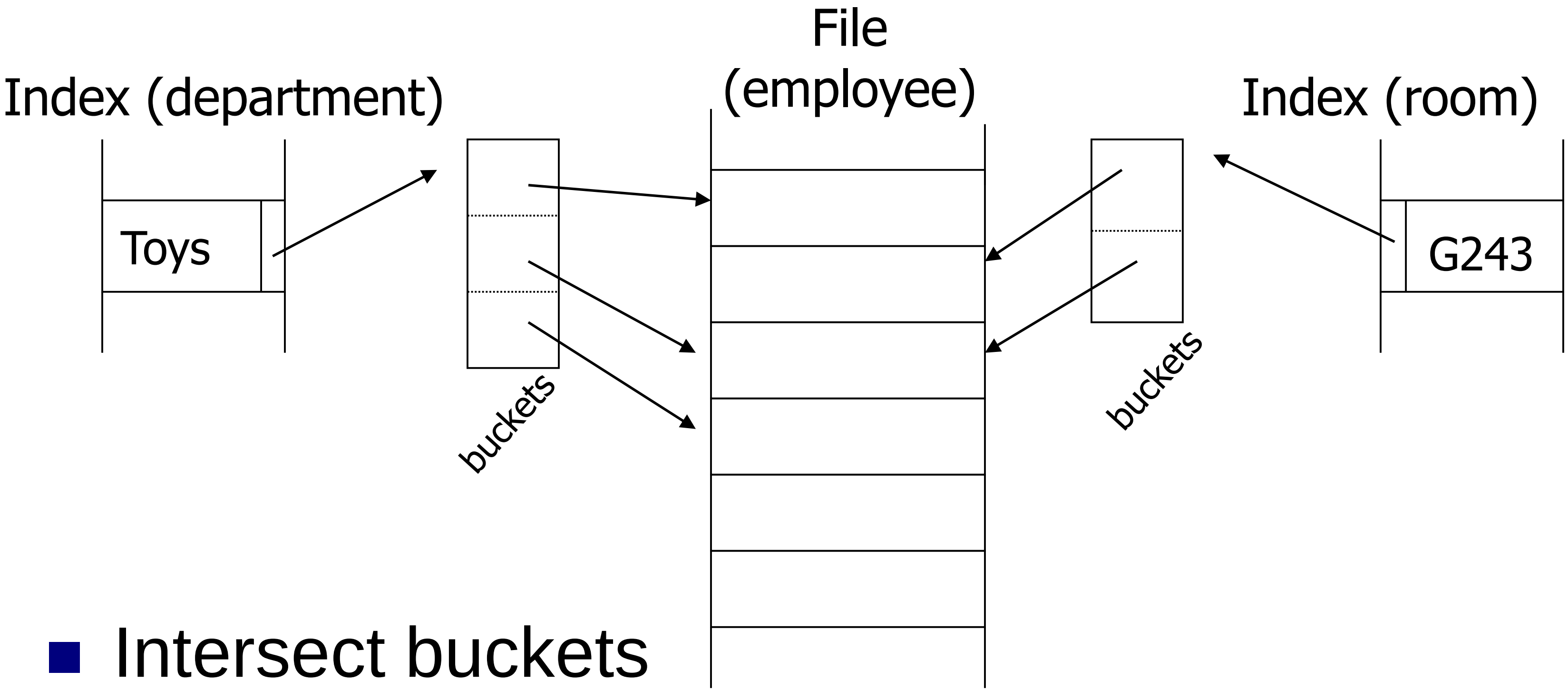
# Secondary Index: Duplicate Keys

- Advantage: a list of records for querying
  - Evaluate more selection constrains without accessing records

- Example:
  - Relation
    - employee(<u>name</u>, department, room)
  - Indexes:
    - name – primary index
    - department – secondary index
    - room – secondary index

# Secondary Index: Duplicate Keys

- Query: employee of Toys dept. in room G243




- Intersect buckets
  - To get pointers to matching employee records
  - Also used in *text information retrieval*

# Example: Text Information Retrieval

- "Full-text" index for documents
- Split documents into words

Document 1

*Caesar and Brutus are ambitious.*

1. Split to words
2. Stemming
3. Ignore words in stop-list

Word List for Doc1

- *Caesar*
- *Brutus*
- *ambitious*

- Build an inverted file
  - over all documents
  - i.e., a file of records <word; [docId, docId2, …]>

# Example: Text Information Retrieval

- Inverted file

| Term | docID |
|---|---|
| ambitious | 1 |
| brutus | 1 |
| brutus | 3 |
| capitol | 2 |
| caesar | 1 |
| caesar | 2 |

Relational view

Inverted file

| Term | Posting list of docIDs |
|---|---|
| ambitious | 1 |
| brutus | 1, 3 |
| capitol | 1 |
| caesar | 1, 2 |

- Retrieve docs containing *Brutus & Caesar*
  - Read *posting lists* for Brutus and Caesar
  - Intersect them

# Conventional Indexes: Summary

- Basic ideas
  - Sparse vs. dense; multilevel
- Insertion / deletion
  - Duplicate keys
    - in case of secondary indexes

- Advantages
  - Simple
  - Index is a sequential file too → good for „full scan“
- Disadvantages
  - Costly updates
  - Lost of physical “sequentiality”
    - due to overflow buckets

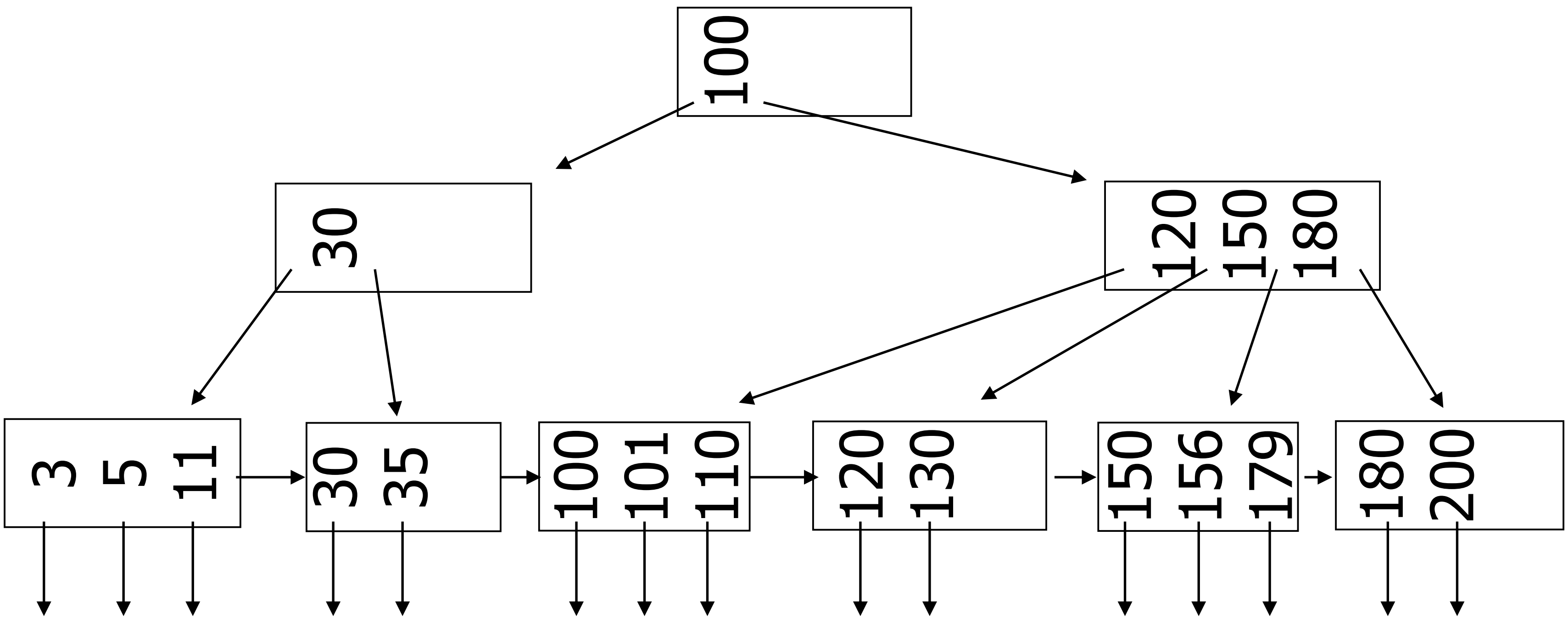
# B-trees

- Another index type
  - Sequential order not necessary
  - Balanced – max I/Os guarantee
- More variants
  - B-tree, $B^+$-tree, $B^*$-tree, …
    - Typically, by saying “*B-tree*” we mean “*$B^+$-tree*”!
- Origin
  - Rudolf Bayer and Ed McCreight invented the B-tree while working at Boeing Research Labs in 1971 (Bayer & McCreight 1972)
    - They did not explain what, if anything, the B stands for.
    - Douglas Comer explains:
      - The origin of "B-tree" has never been explained by the authors. As we shall see, "balanced," "broad," or "bushy" might apply. Others suggest that the "B" stands for Boeing. Because of his contributions, however, it seems appropriate to think of B-trees as "Bayer"-trees.

* Source: Wikipedia

# B$^{+}$-tree

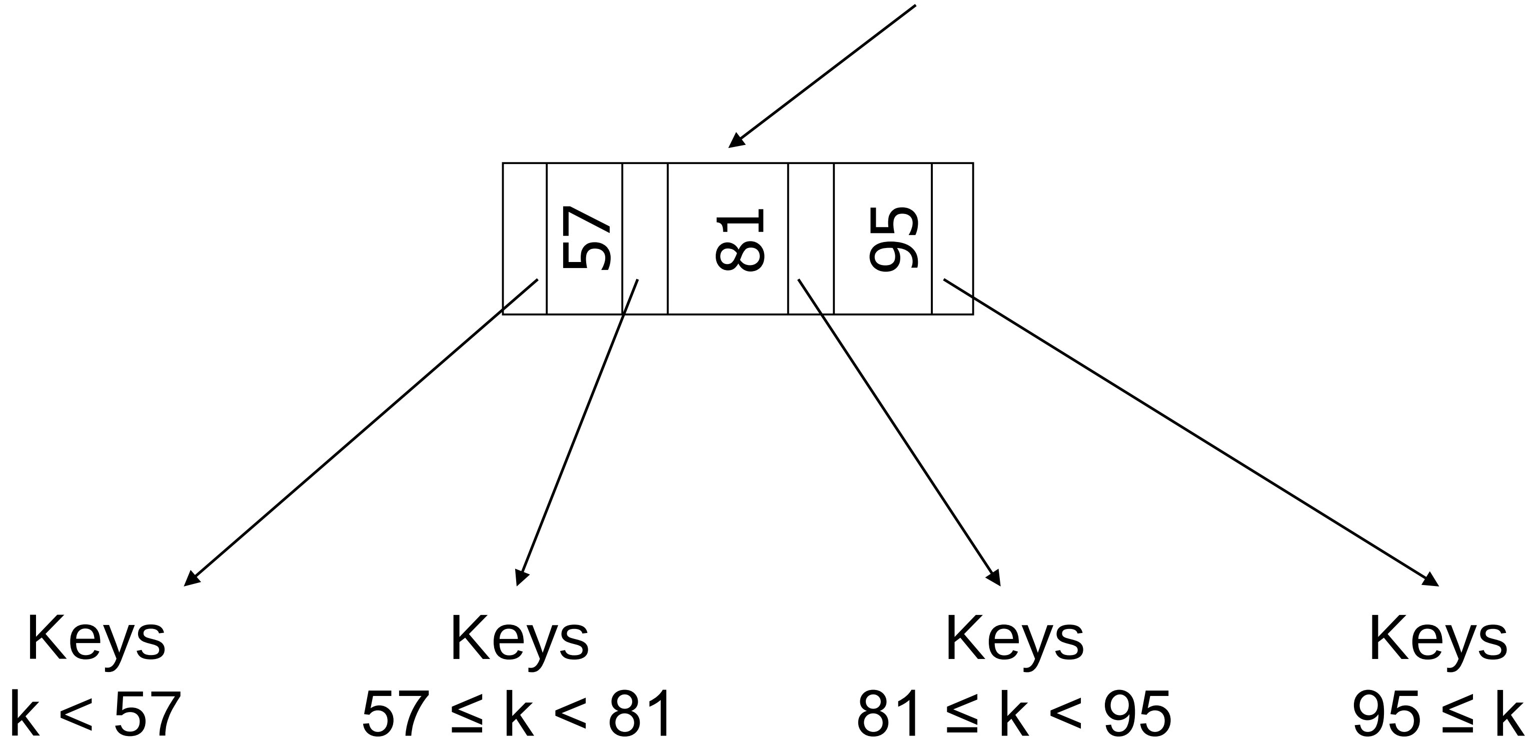
- Example *n=4*




… pointers to record in file …

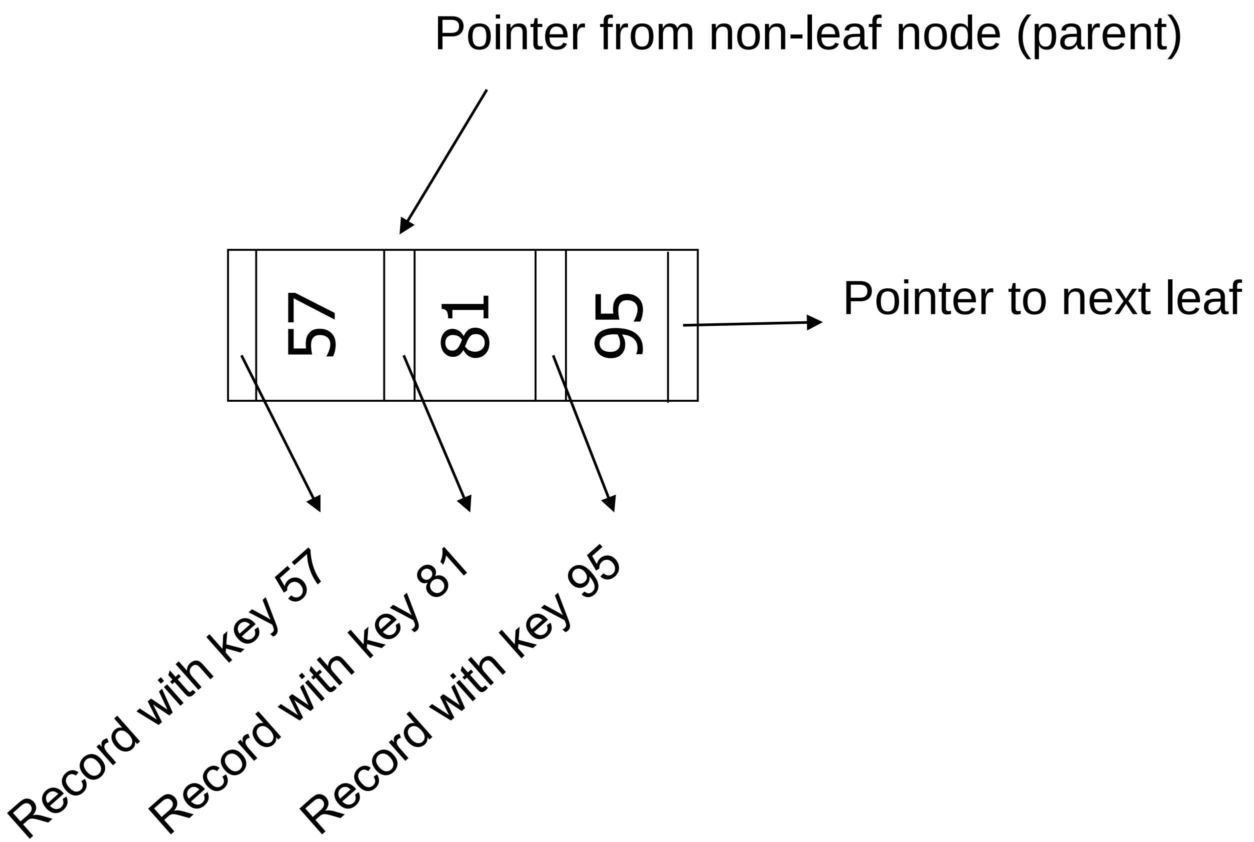
# $B^+$-tree

- Non-leaf node, *n=4*



# $B^+$-tree

- Leaf node, *n=4*




Record pointers:

Revision follows, so skip it.

# $B^+$-tree

- Parameter *n* (tree arity) influences:
  - Node format:

| $p_1$ | $K_1$ | $p_2$ | $K_2$ | $p_3$ | … | $p_{n-1}$ | $K_{n-1}$ | $p_n$ |
|---|---|---|---|---|---|---|---|---|

  - Minimal occupation
  - Leaf node
    - All leaves at same lowest level
    - $p_i$ points to record with key $K_i$ (data)
    - $p_n$ points to next leaf (chained leaves)
  - Non-leaf node
    - $p_i$ points to node organizing keys K: $K_{i-1} \leq K < K_i$

# B⁺-tree

- Occupation constraints

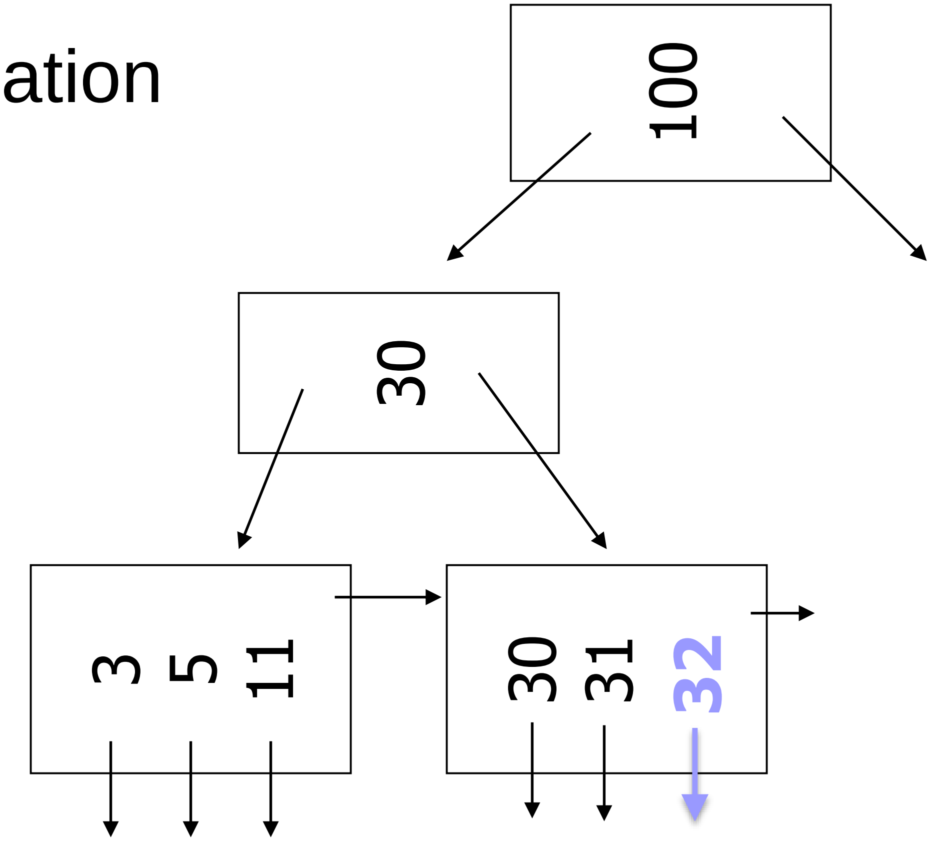
| | Max pointers | Min pointers | Max keys | Min keys |
|---|---|---|---|---|
| Non-leaf (not root) | n (children) | $\lceil n/2 \rceil$ (children) | n-1 | $\lceil n/2 \rceil$ -1 |
| Non-leaf (root) | n (children) | 2 (children) | n-1 | 1 |
| Leaf (not root) | n-1 (records) | $\lceil (n-1)/2 \rceil$ (records) | n-1 (records) | $\lceil (n-1)/2 \rceil$ (records) |
| Leaf (root) | n-1 (records) | 0 (records) | n-1 (records) | 0 (records) |

# $B^+$-tree: Insertion

- Principle: Grows from leaves to root
- Procedure: Find leaf node and insert new key
  - Including pointer to the new record
  - Update parent if necessary
- Insert cases:
  a) No reorganization
     - Free capacity in leaf
  b) Split leaf
  c) Split non-leaf
  d) Split root

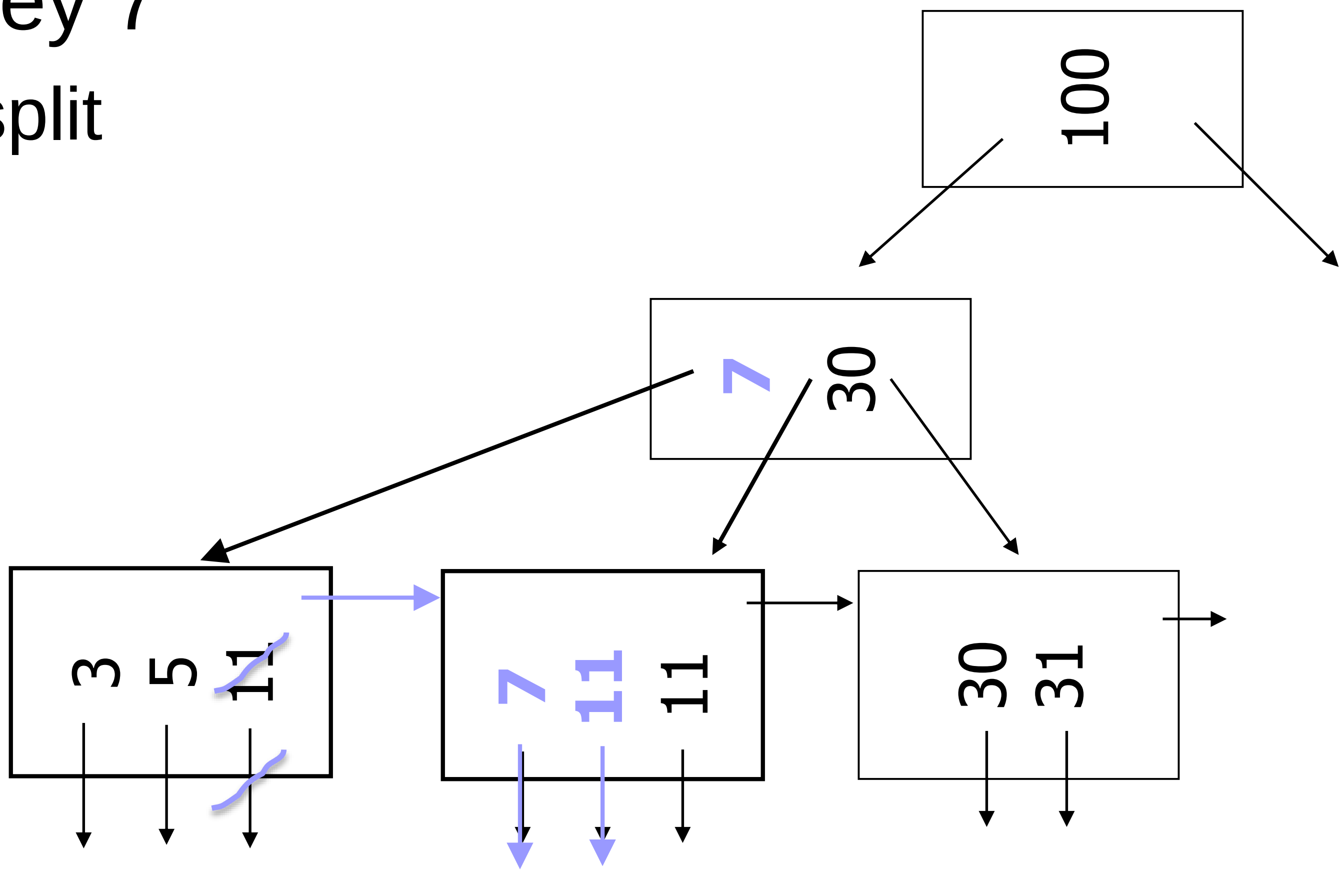
# B$^+$-tree: *n=4*

- Insert key 32
  - No reorganization



# B⁺-tree: *n=4*

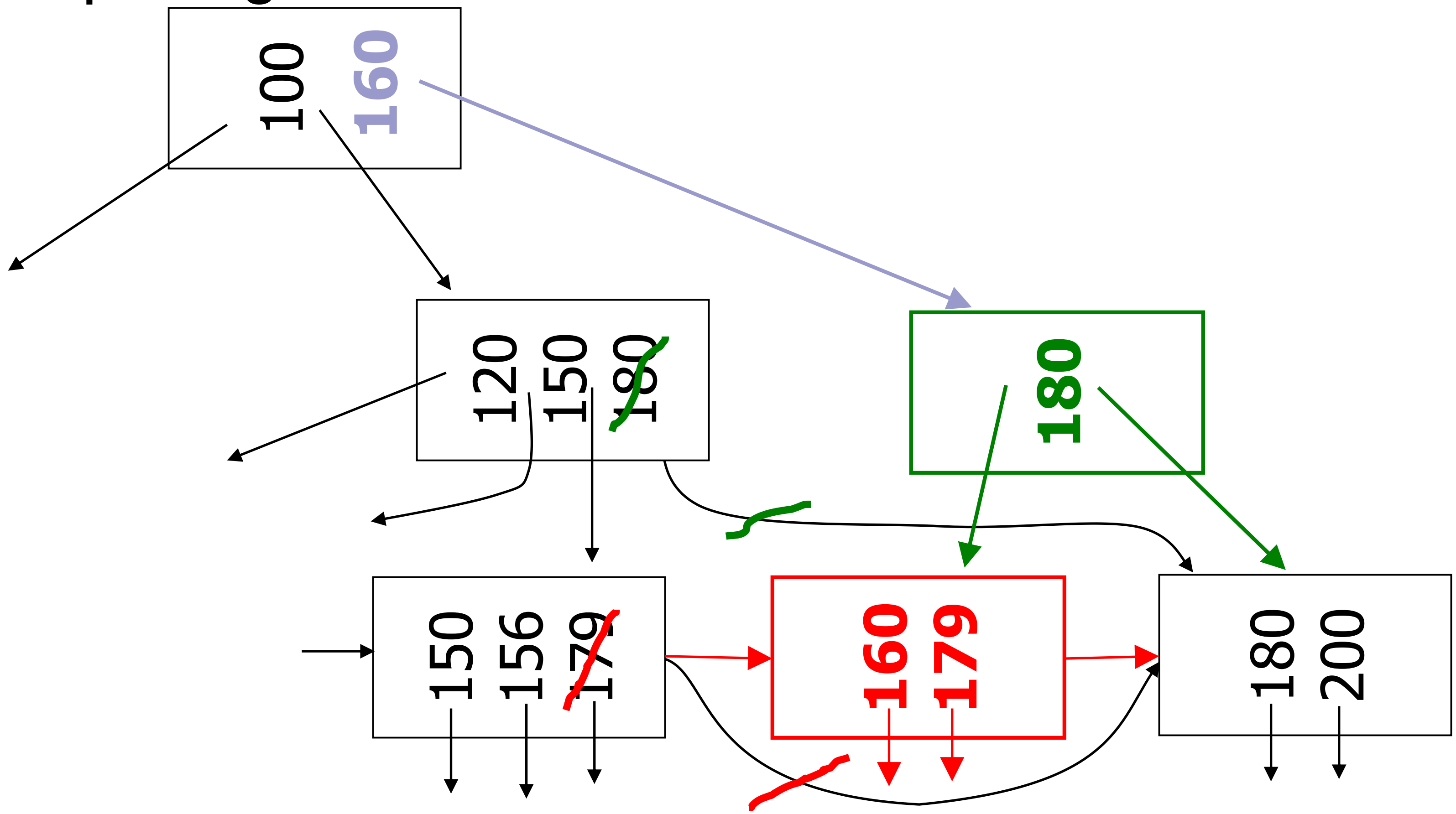
- Insert key 7
  - Leaf split




Deletion variants follow, which is revision, so skip it.

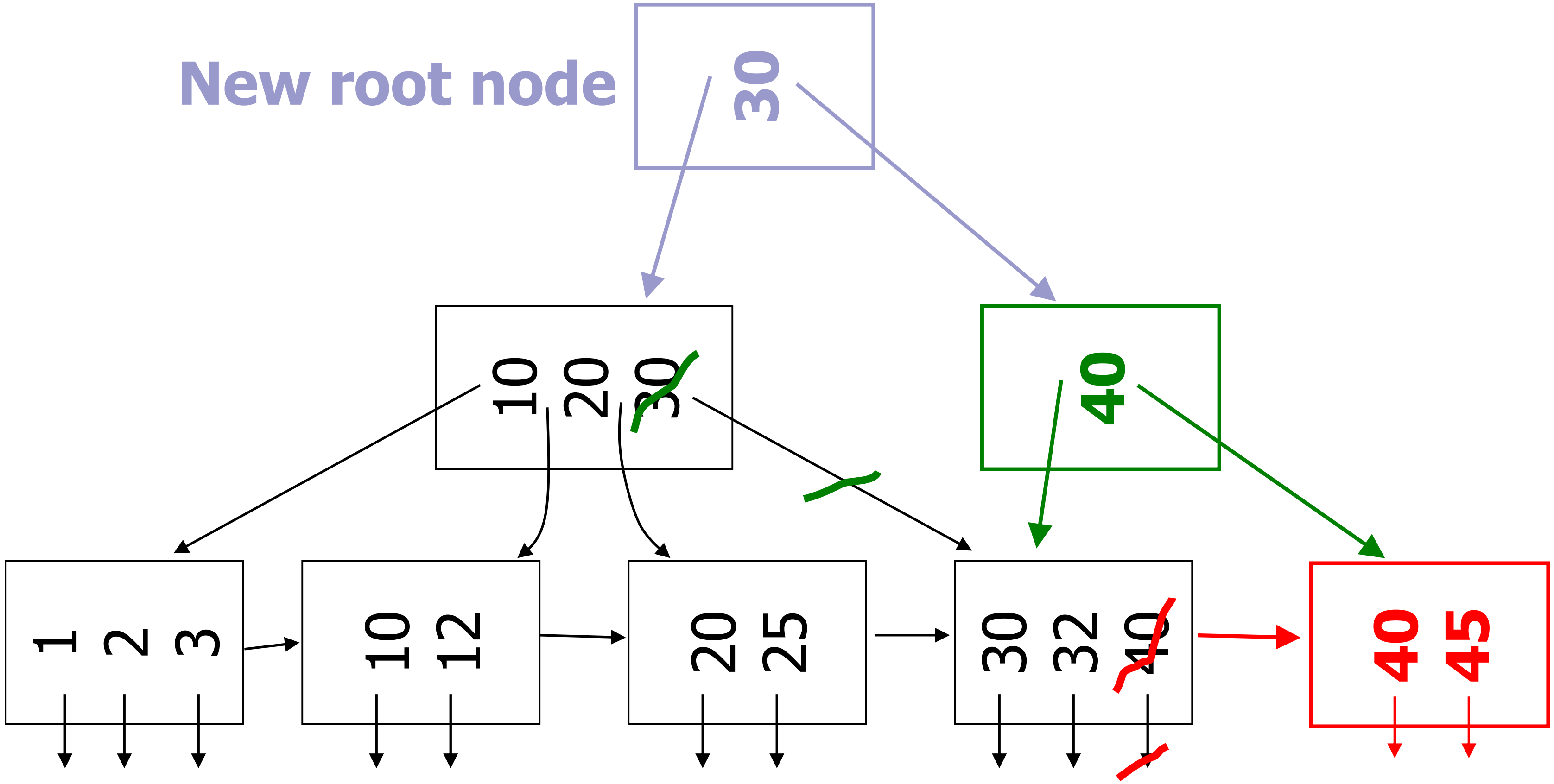
# B⁺-tree: *n=4*

- Insert key 160
  - Splitting non-leaf node



# B⁺-tree: *n=4*

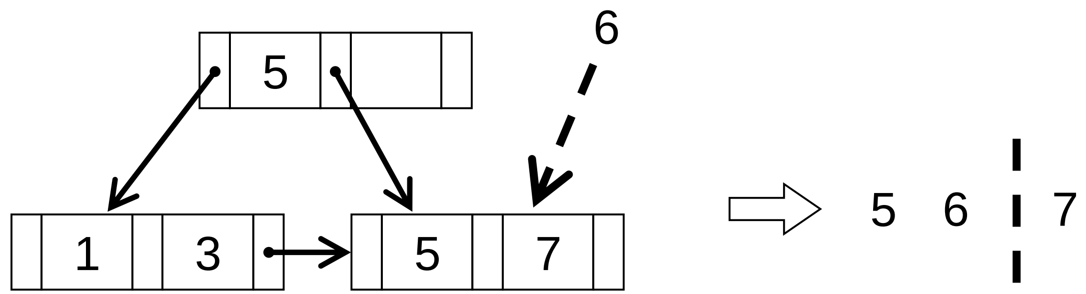
- Insert key 45
  - Split root



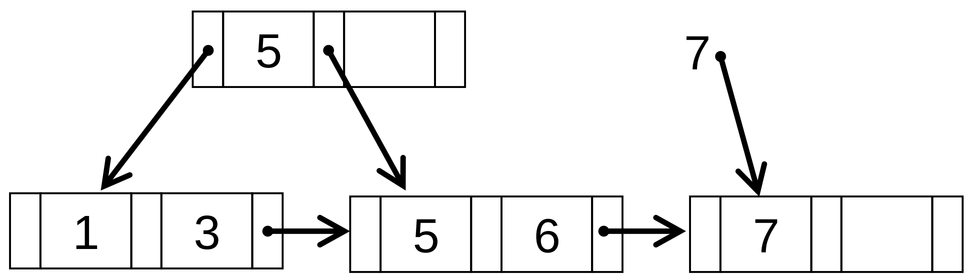
# B+-tree: Split Leaf

*n*=3, insert key 6

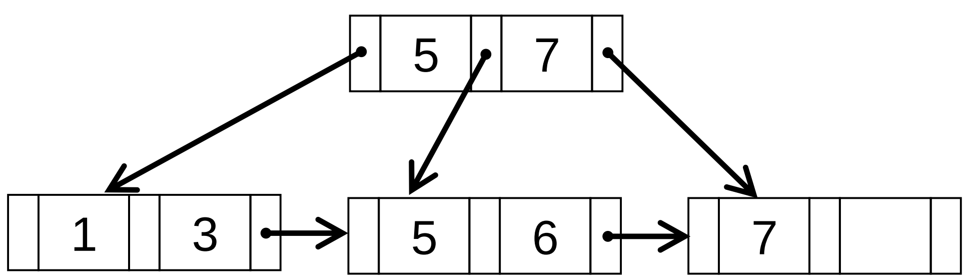
1.




2.




3.



# B$^{+}$-tree: Split non-leaf node

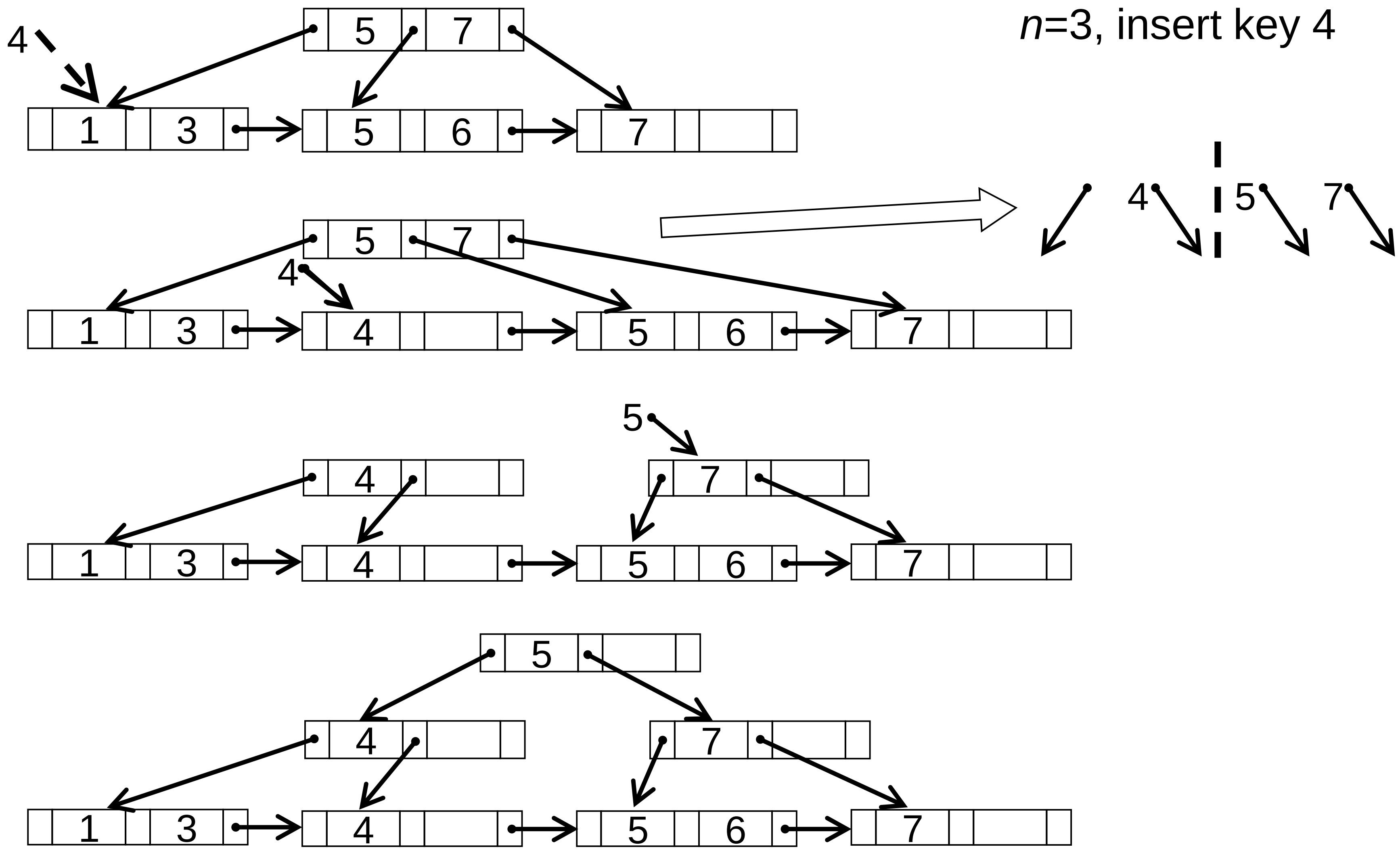
*n*=3, insert key 4

1.

2.

3.

4.

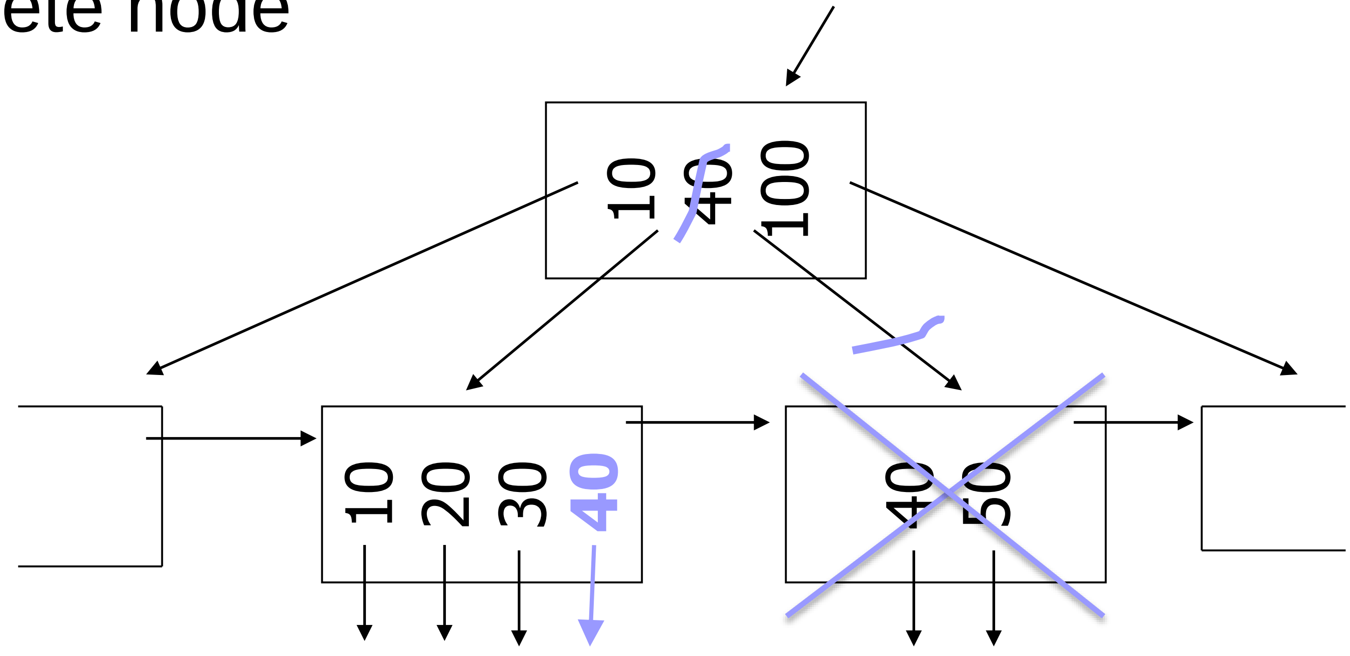


# $B^+$-tree: Deletion

- Find leaf node and delete key
  - Including the corresponding record
  - Delete node if empty, …
- Deletion cases:
  a) No reorganization (leaf is not “underfilled”)
  b) Coalesce with neighbor (sibling node) and delete node
  c) Redistribute keys between neighbors (without node deletion)
  d) Cases (b) and (c) for non-leaf nodes

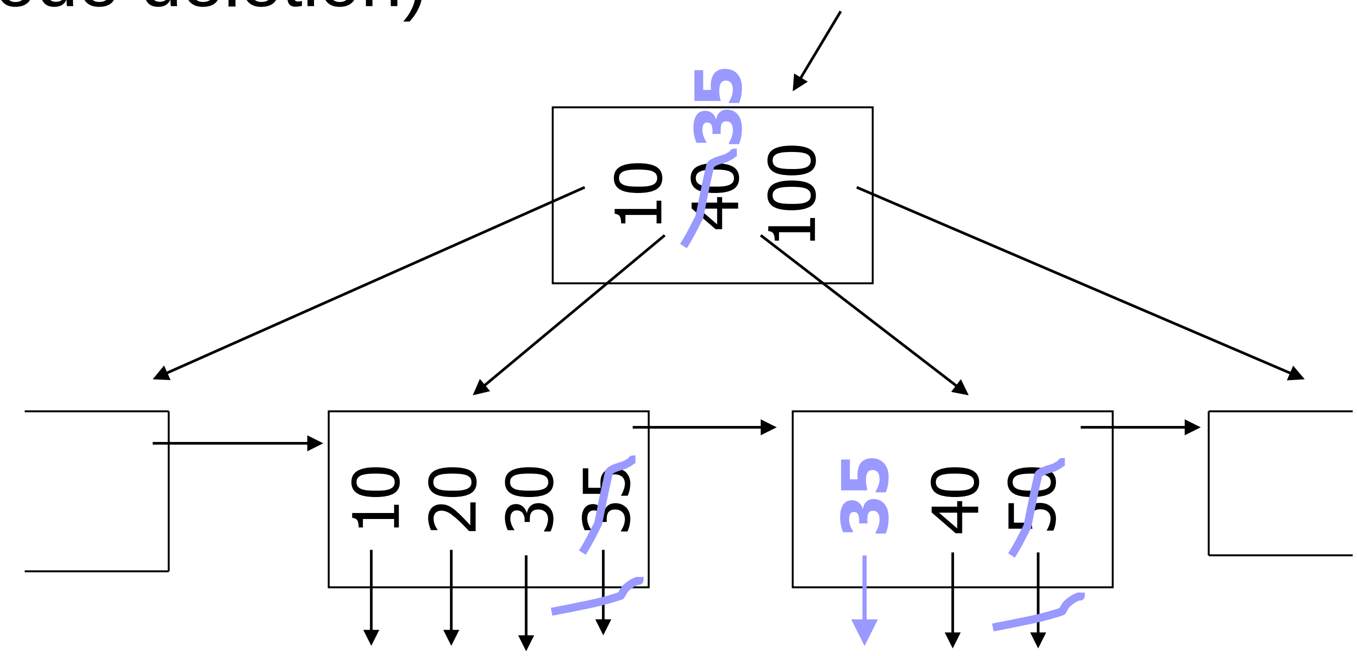
# B+-tree: *n=5*

- Delete key 50
  - Coalesce (merge) keys into a neighbor and delete node



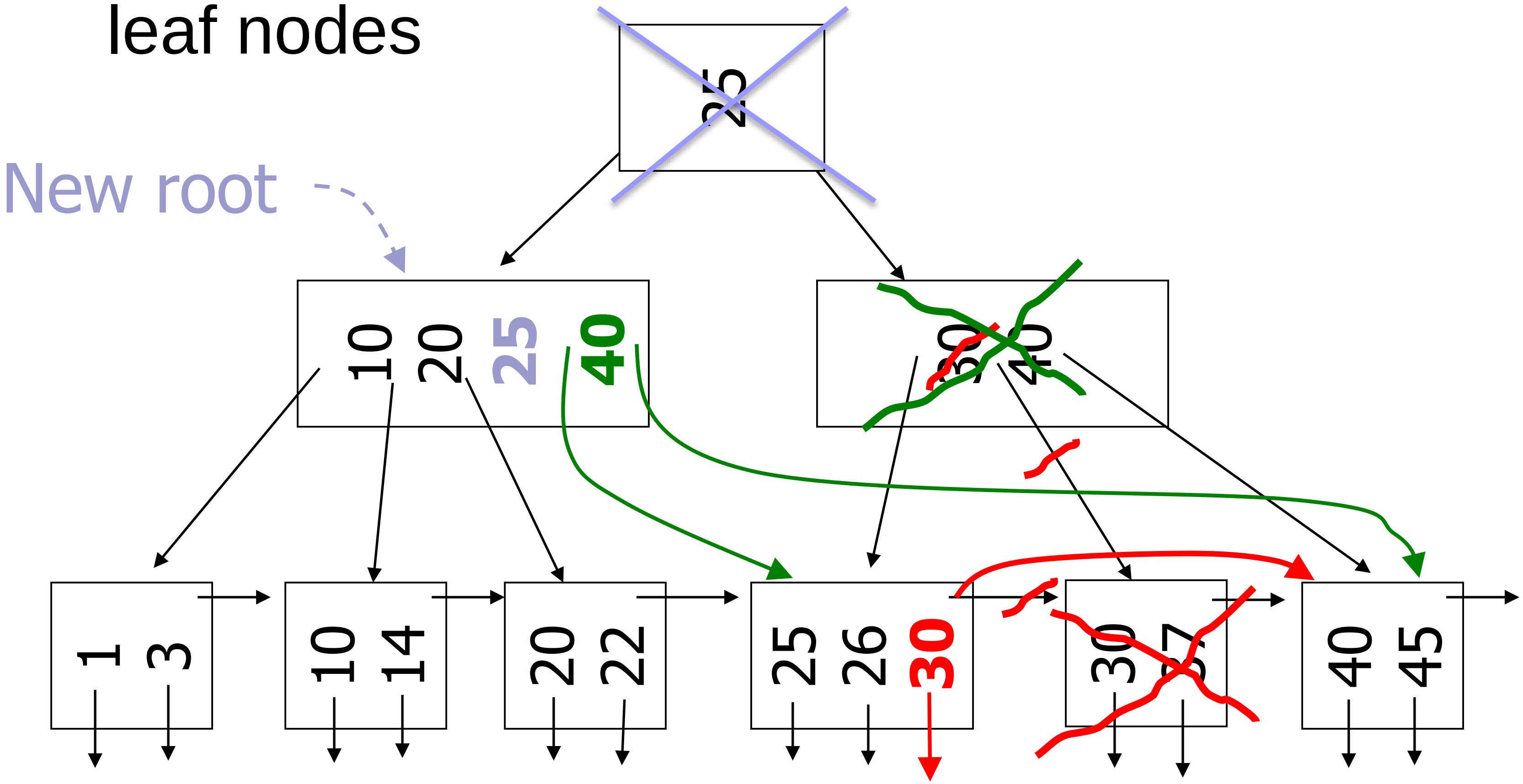
# B+-tree: *n=5*

- Delete key 50
  - Redistribute keys between neighbors (avoid node deletion)



# B+-tree: *n=5*

- Delete key 37
  - Redistribute keys between neighboring non-leaf nodes



# B$^+$-tree: Deletion

- Practice:
  - Coalescing often not implemented
    - More inserts than deletes (both random) leads to utilization of 65-69% even if nodes not merged
  - Too complex and low impact

# $B^+$-tree vs. Conventional index

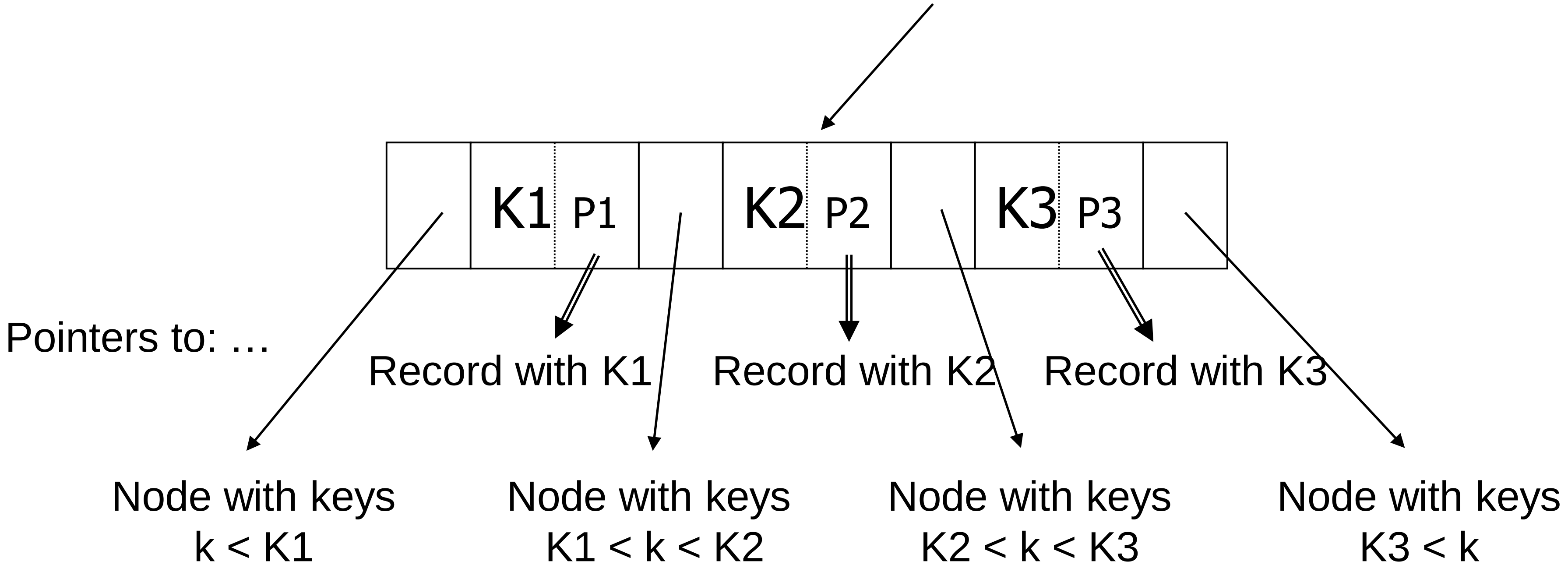
- Block size 4 KiB
  - Key = 4B, pointer to block/rec = 4B
  - Multilevel *secondary* index
    - sparse: 512 keys and pointers to a block
    - dense: 512 keys and pointers to records
  - $B^+$-tree
    - non-leaf node: 512 pointers to other nodes
    - leaf: 511 pointers to records
- Comparison in records in a relation:
  - Full 2-level indexes: (1st level == 1 block)
    - Sec. index: up to 262 144 records ($512^h$)
      - *up to 1 048 576 records if implicit indexes are used*
    - $B^+$-tree: up to 261 632 records ($512^{h-1} \cdot 511$)
    - Prim. index (all sparse levels): up to $512^{h+1}$ records

# $B^+$-tree vs. Conventional index

- Conclusion:
  - ☹ $B^+$-tree has larger space overhead
    - ☺ Is dynamic, but may not be physically sequential
    - ☹ $B^+$-tree – more complex locking
  - ☹ Conventional index must be reorganized as whole
    - DBMS does not know when to reorganize
      - ☺ $B^+$-tree makes small local reorganizations
      - ☹ Conventional index needs large reorganizations
  - Buffer manager
    - ☺ $B^+$-tree – fixed buffer requirements (log depth)
    - ☹ Static index – must use overflow blocks to be efficient
      - Linear complexity due to overflow areas
    - LRU is no good for $B^+$-trees!
- **$B^+$-tree is a better organization.**

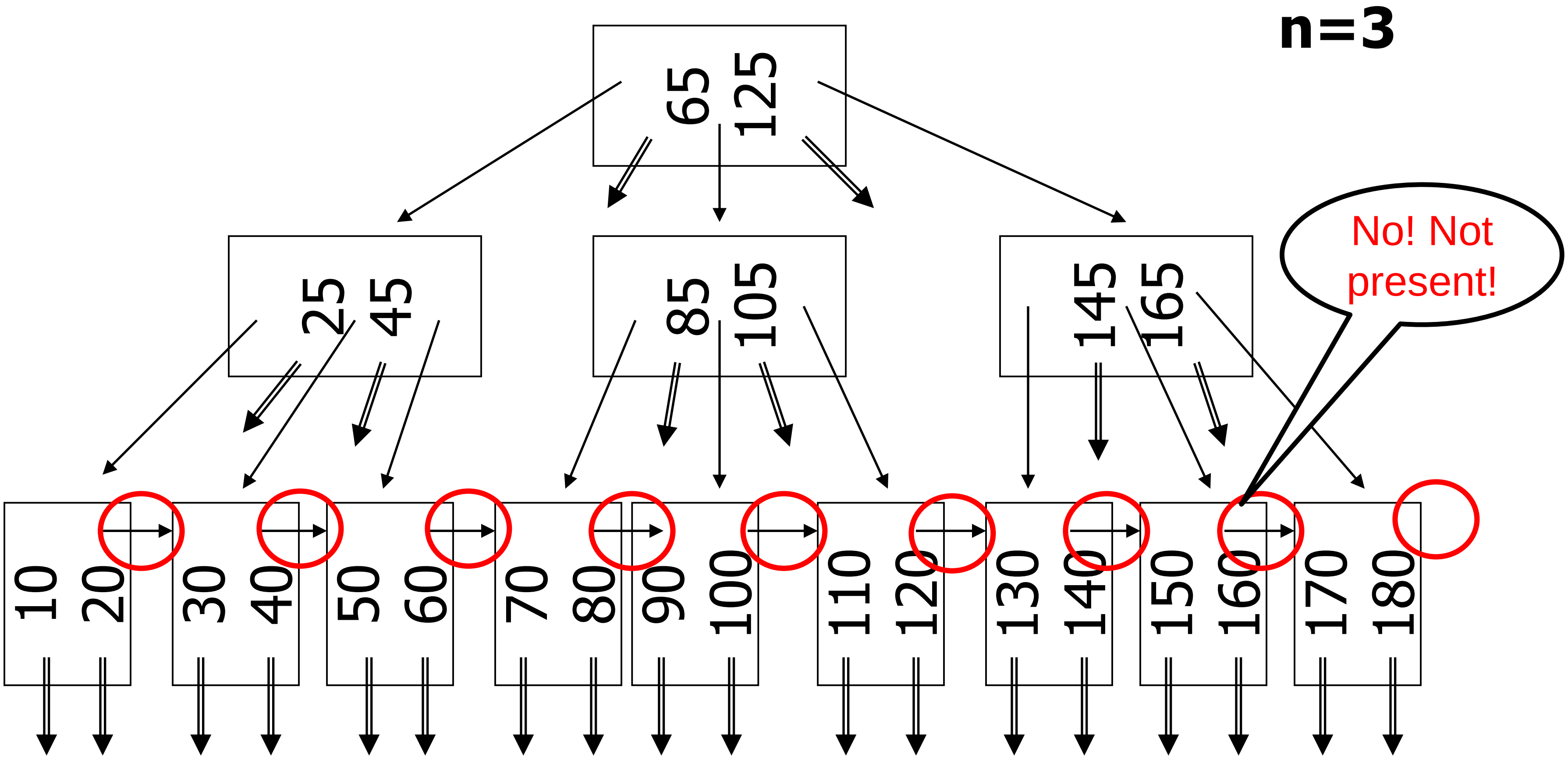
# B-tree (without $^{+}$)

- Idea: no key replication
  - → record pointer also in non-leaf nodes
  - Different constraints on key values in subtrees



# B-tree: Example

- Leaf chaining cannot be used



# B-tree

- Occupation constrains

| | Max pointers | Min pointers | Max keys | Min keys |
|---|---|---|---|---|
| Non-leaf (non-root) | n (children) | ⌈n/2⌉ (children) | n-1 (keys and pointers) | ⌈n/2⌉ -1 (keys and pointers) |
| Non-leaf (root) | n (children) | 2 (children) | n-1 (keys and pointers) | 1 (keys and pointers) |
| Leaf (non-root) | n-1 (records) | ⌈(n-1)/2⌉ (records) | n-1 (record pointers) | ⌈(n-1)/2⌉ (record pointers) |
| Leaf (root) | n-1 (records) | 0 (records) | n-1 (record pointers) | 0 (record pointers) |

# Comparison: B-tree and B$^+$-tree

- Sizes
  - Block = 4KiB
  - Pointer = 4 bytes
  - Key = 4 bytes
- Assume a *full 2-level* tree
  - 1 root and leaves
  - Each node in one block

# Comparison: B-tree

- Root:
  - 341 keys + 341 record pointers
  - 342 pointers to child nodes (blocks)
    - 341·(4+4) + 342·4 = 4096 bytes
- Leaf:
  - 512 keys + 512 record pointers
    - 512 · (4+4) = 4096 bytes
- Total records:
  - 341 + 342 · 512 = 175 445 recs

# Comparison: $B^+$-tree

- Root:
  - 511 keys, 512 block pointers
    - 511·4 + 512·4 = 4092 bytes
- Leaf:
  - 511 keys + 511 record pointers
    - 511 · (4+4) + 4 = 4092 bytes
- Total records:
  - 512 · 511 = 261 632 recs

# Comparison: Result

- Read I/Os:
  - B-tree
    - $P_{1\ read} = 341 / 175\ 445 = 0{,}2\%$
    - $P_{2\ reads} = 1 - P_{1\ read} = 99{,}8\%$
  - $B^+$-tree
    - $P_{2\ reads} = 100\%$

# Comparison: Result

- B-trees
  - 👍 Faster lookup
    - Not always, can be deeper (see prev. slide)
  - 👎 Different formats of non-leaf & leaf nodes
  - 👎 Deletion more complicated

➔ $B^+$-trees preferred!

# $B^+$-tree

- $B^+$-tree as file
  - Leaves store the records themselves.
- Duplicate keys
  - Pointers in leaves = pointers to buckets
    - i.e., blocks with a list of record pointers with the same key value
- Variable-length key values (e.g., strings)
  - Store completely → low arity, varying arity, …
  - Use prefixes (prefix compression)

# Lecture's Takeaways

- Efficiency of B+ trees
- Handling duplicate keys
  - i.e. multiple records with the same key value.
- Revision of terminology
  - Dense / sparse index
  - Primary / secondary index
    - Clustered / non-clustered index
  - Covering index