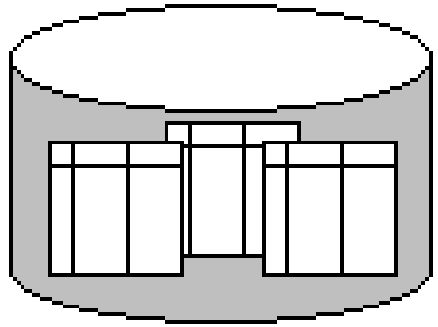
ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ

Συστήματα Διαχείρισης Βάσεων Δεδομένων

Διάλεξη 2η: Access methods, File organization, B+Tree

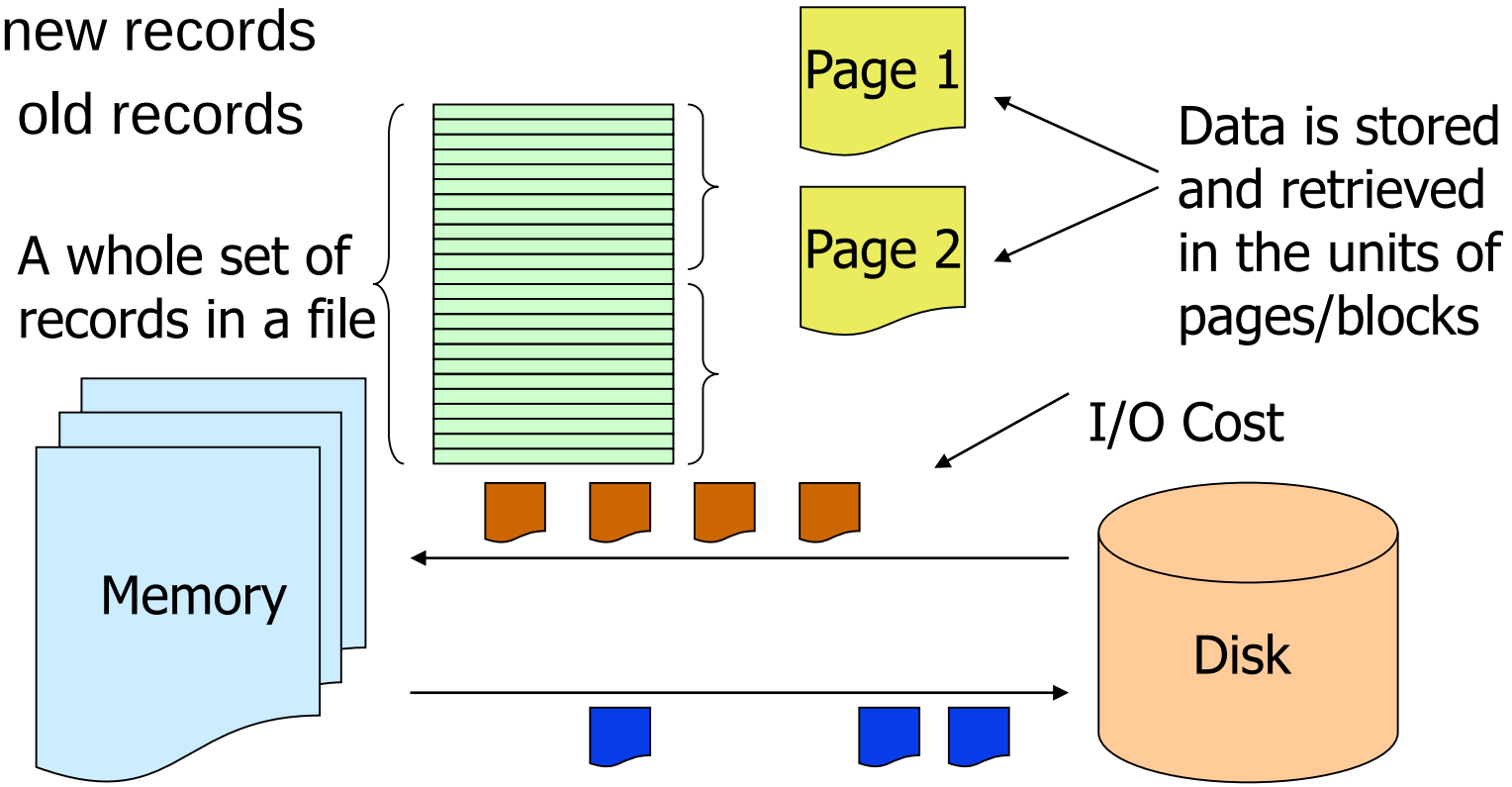
Δημήτρης Πλεξουσάκης
Τμήμα Επιστήμης Υπολογιστών

ACCESS METHODS: FILE ORGANIZATIONS, B+TREE



File Storage

- How to keep blocks of records on disk files
  - but must support operations:
    - scan all records
    - search for a record id (“RID”)
    - insert new records
    - delete old records



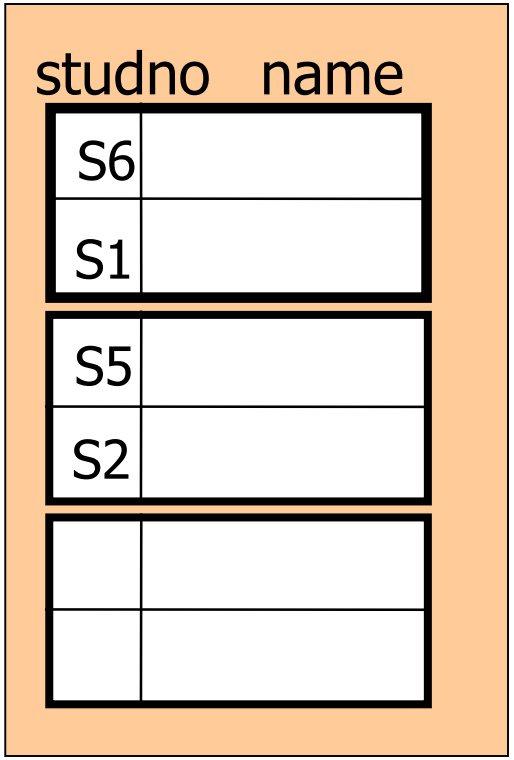
Alternative File Organizations

- Many alternatives exist, each ideal for some situation, and not so good in others:
  - Heap Files: Suitable when typical access is a file scan retrieving all records
  - Sorted Files: Best if records must be retrieved in some order, or only a `range' of records is needed
  - Hashed Files: Good for equality selections
    - File is a collection of buckets
      - Bucket = primary page plus zero or more overflow pages
    - Hashing function **h**:
      - **h(r)** = bucket in which record r belongs
      - **h** looks at only some of the fields of r, called the search fields

File Organization: Unordered Records

- Heap (or Pile) Files: Place records in the order they are inserted
  - New records inserted at the end of the file

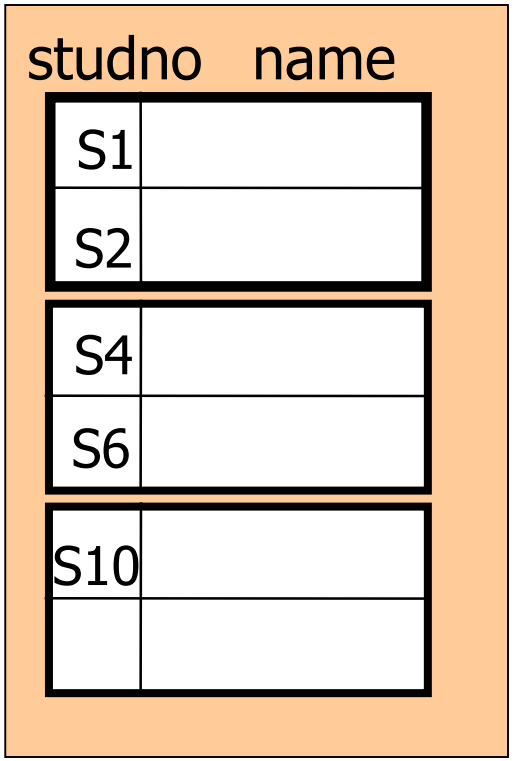
Insertion	efficient	
Deleting	expensive	reorganisation & fragmentation
Searching	expensive	linear n/2
Retrieval in order	expensive	sort



File Organization: Ordered Records

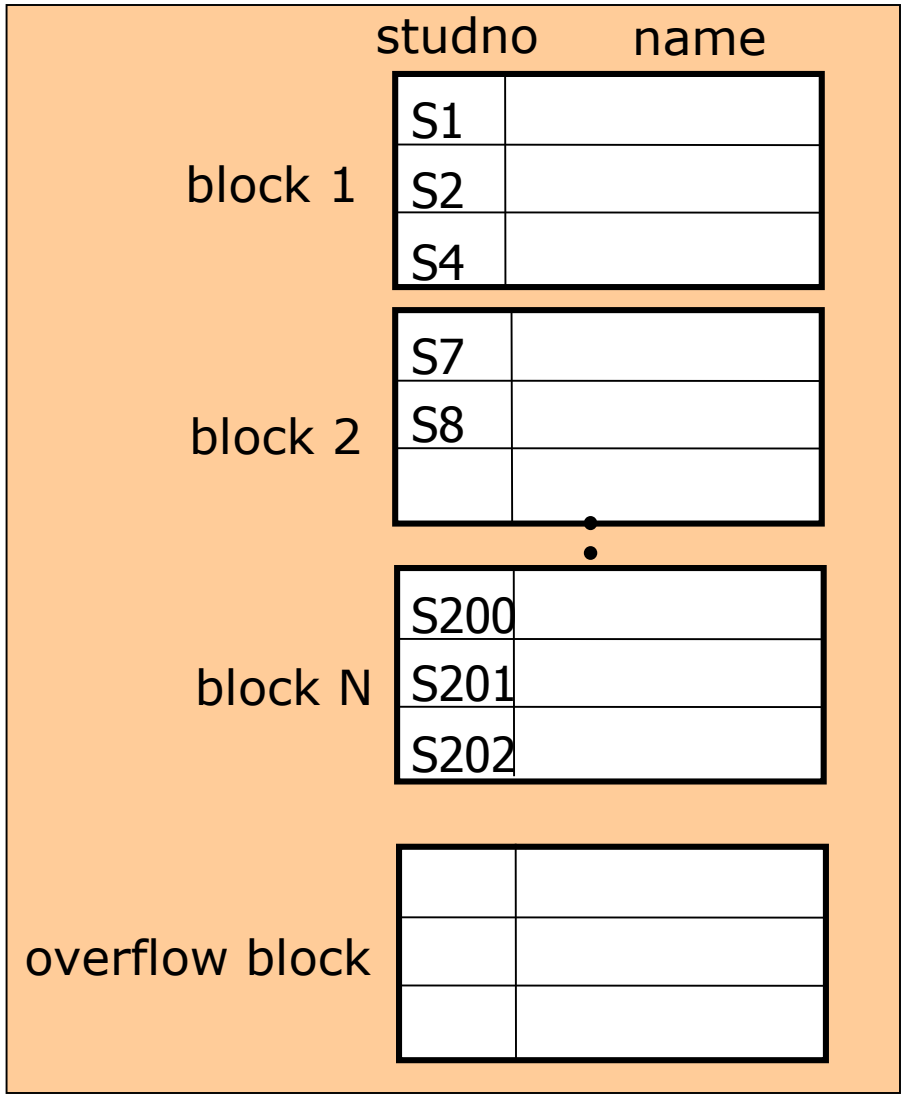
- Sorted (or Sequential) Files: Physically order the records of a file on disk based on values of one of the fields — ordering field / ordering key

Insertion	expensive	reposition
Deleting	expensive	reorganisation & fragmentation
Searching on ordering key	more efficient	binary $\log_2(n)$
Searching on non-ordering key	expensive	linear n/2
Retrieval in order of ordering key	efficient	no sort



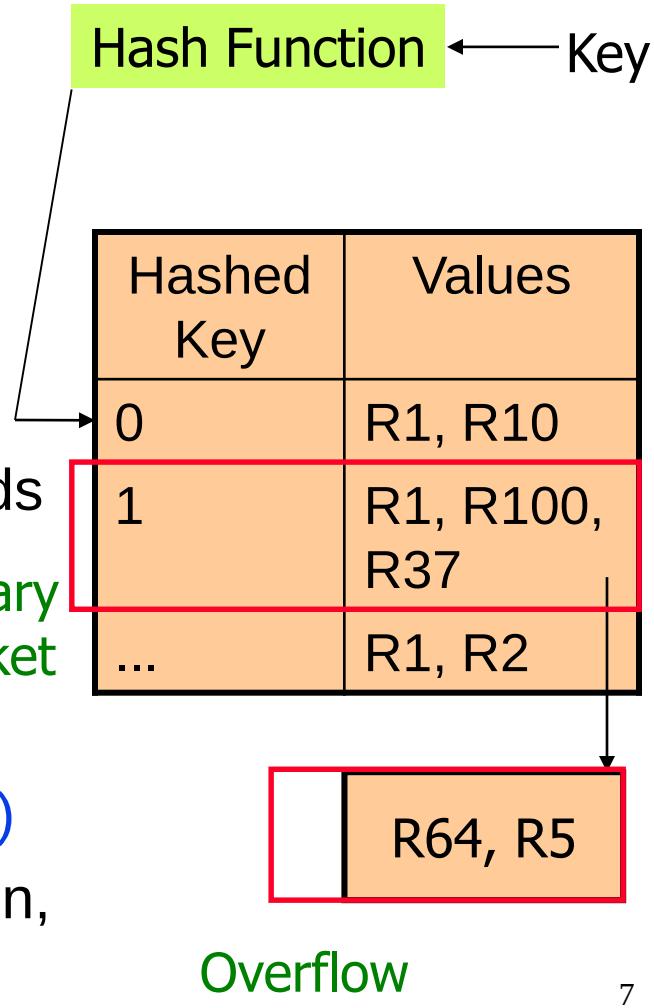
File Organization: Ordered Records with Overflow Blocks

- Insertion is expensive: records must be inserted in the correct order
  - must locate the position in the file where the record is to be inserted
    - if there is free space insert there
    - if no free space, insert the record in an overflow block
- It is common to keep a separate unordered overflow (or transaction) block for new records to improve insertion efficiency
  - this is periodically merged with the main ordered file



File Organization: Hashed Records

- Hashed Files: The record with hash key value K is stored in bucket i, where `i=h(K)`, and `h` is the hashing function
  - Blocks containing data can be viewed as buckets
- Collisions occur when a new record hashes to a bucket that is already full
  - An overflow file is kept for storing such records
  - Overflow records that hash to each bucket can be linked together
- Overflow handling in dynamic databases:
  - open addressing or chaining (closed hashing)
- Hashing can be used not only for file organization, but also for index-structure creation (more latter)



Cost Model for Analysis

- We ignore CPU costs, for simplicity:
  - **B**: The number of data blocks (pages) in a table
  - **R**: Number of records per block in a table
  - **D**: (Average) time to read or write disk block
  - Measuring number of block I/O's ignores gains of pre-fetching and sequential access;
    - thus, even I/O cost is only loosely approximated
  - Average-case analysis;
    - based on several simplistic assumptions
- Good enough to show the overall trends!

Assumptions in Analysis

- Single record insert and delete
- Heap Files:
  - Equality selection on key; exactly one match (scan ½ file on average)
  - Range search requires scan entire file since unordered
  - Insert always at end of file (load last page, add, write out)
  - Delete anywhere in the file (find page, delete record, write out)
- Sorted Files:
  - Selections on sort field(s), no overflow blocks for sorted files
  - Range search requires find first, then sequentially retrieve
  - Insert and delete in middle of file
  - Files compacted after deletions or reordered after insertions (read in remaining 0.5B pages, adjust, then write out)
- Hashed Files:
  - No overflow buckets, 80% page occupancy

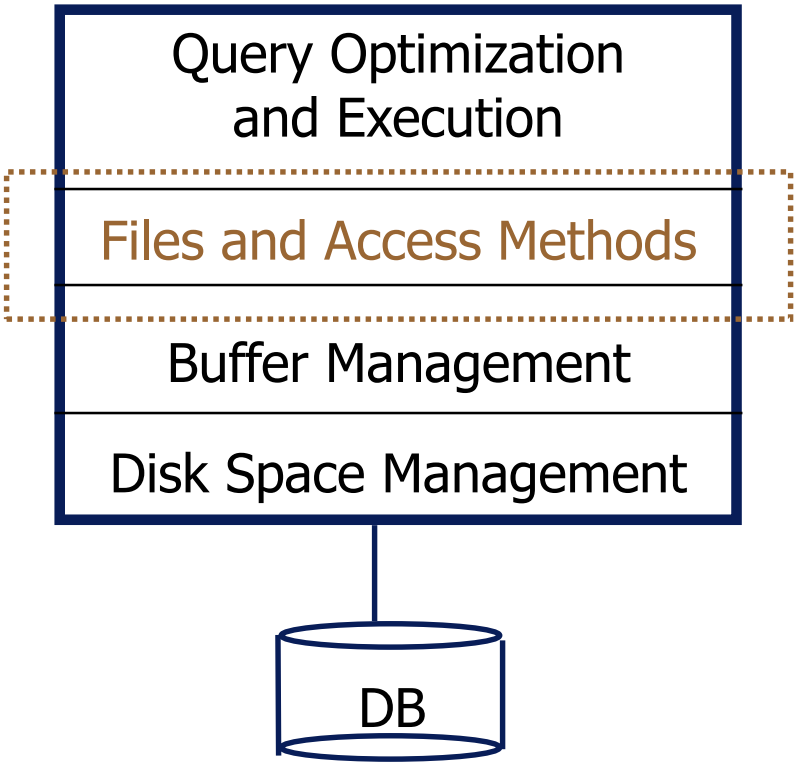
Cost of Operations

	Heap File	Sorted File	Hashed File
Scan all records	BD	BD	1.2BD
Equality Search	0.5BD	D ($\log_2$ B)	D
Range Search	BD	D ($\log_2$ B + # of pages with matches)	1.2BD
Insert	2D	Search + 2 (0.5 BD)	2D
Delete	Search + 2D	Search + BD	2D

Single Record and Range Search

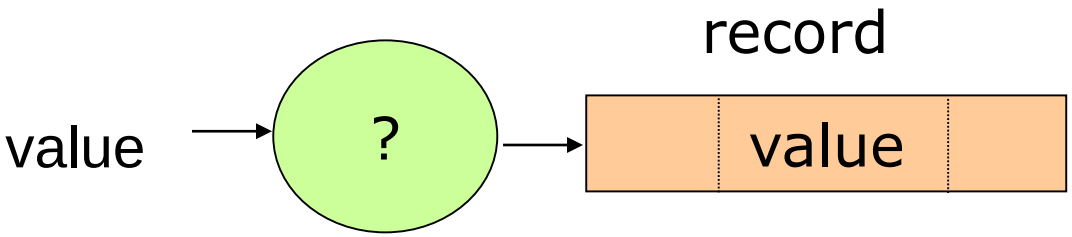
- Single record retrieval
  - "Find student name whose #AM = 1487"
- Range queries
  - "Find all students with grade < 3.0"
- Sequentially scanning of file is costly!
- If data is in sorted file
  - Binary search to find first such student
  - Scan to find others
  - Cost of binary search can still be quite high

The BIG picture…



What is an Index?

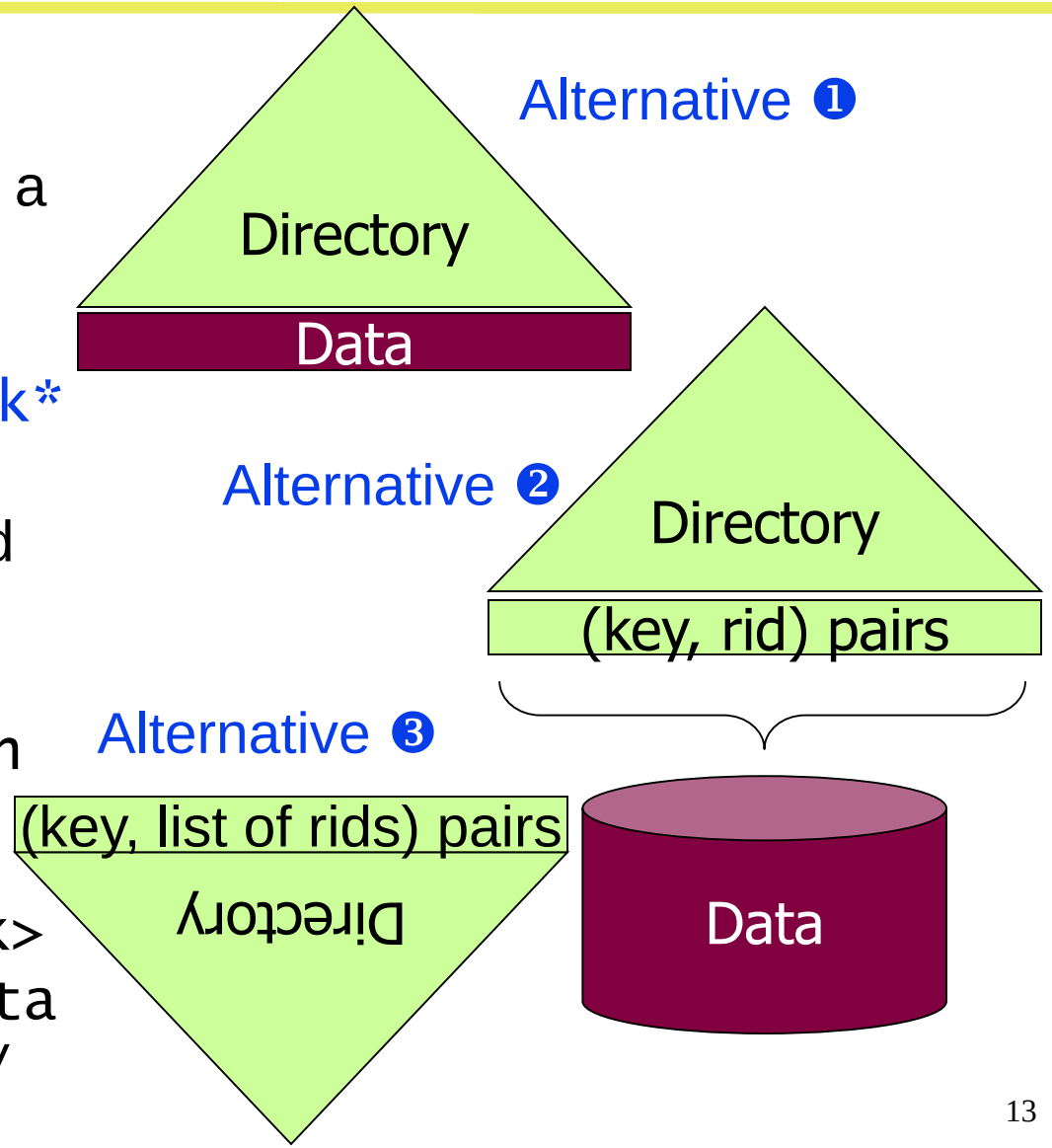
- Databases tend to be very large: millions or billions of records
- Answers to ultimate queries tend to be small
  - How to locate required records efficiently?
- Index: Data structure for locating efficiently records with given search key



  - Also facilitates a full scan of a relation if records not stored on physically consecutive blocks
- An index on a file speeds up selections on the search key fields for the index
  - Any subset of the fields of a relation can be the search key for an index on the relation
  - Search key is not the same as key (minimal set of fields that uniquely identify a record in a relation)

Indexes

- An index is a collection of data entries that supports efficient retrieval of all data entries k* with a given key value k
- Two issues:
  - What is stored as a data entry k* in an index ?
  - How are data entries organized (index structure)?
- Three main alternatives:
  - ❶ Actual data record with search key value k
  - ❷ <k, rid of data record with search key value k>
  - ❸ <k, list of rids of data records with search key value k>



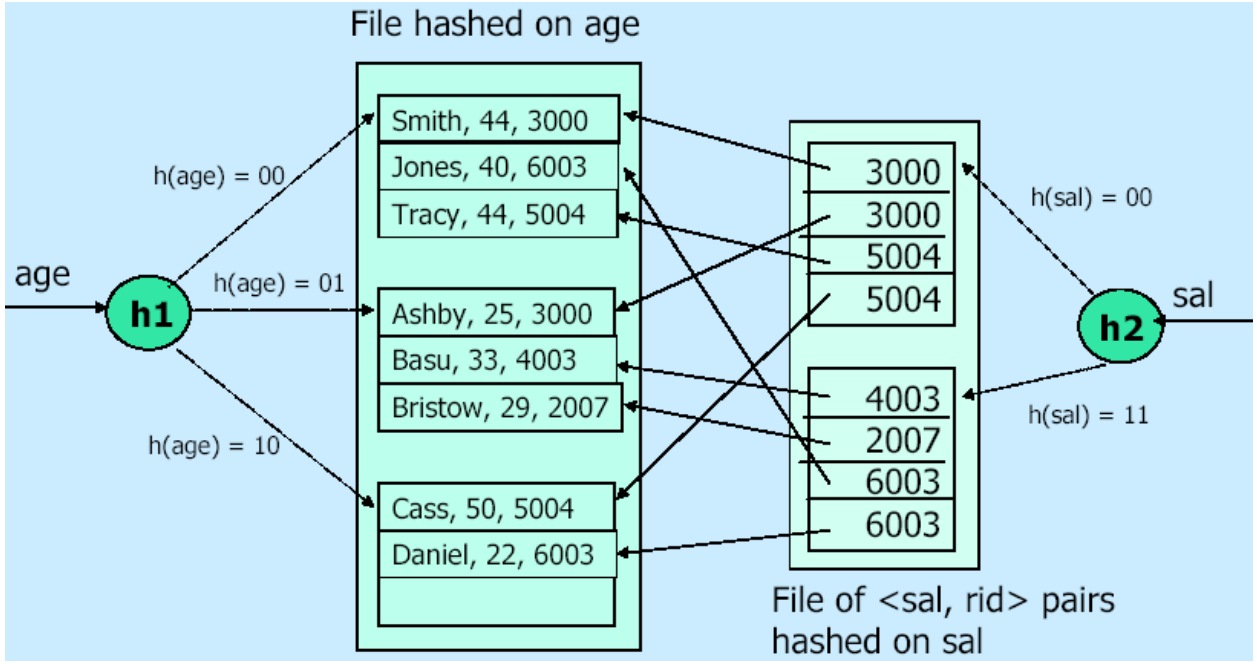
Index Data Entry k*

- Alternative ❶:
  - Index and data records stored together
  - A special file organization for data records
    - Heap or sorted or hash file organization
  - This alternative saves pointer lookups but can be expensive to maintain with insertions and deletions
  - At most one index on a given collection of data records can use Alt ❶
  - Otherwise, duplicate data records, leading to redundant storage and potential inconsistency
    - Rest of the index use Alternative ❷ or ❸
- Alternatives ❷ and ❸:
  - Index data entries point to data records
    - Typically much smaller than data records (easier to maintain)
  - Alternative ❸ better space utilization than Alternative ❷
    - But leads to variable sized data entries depending on the number of data records with a given search key value
    - Even worse, for large rid lists the data entry would have to span multiple blocks!

Example

- On Left : Data stored in file hashed on `age`
  - Alternative ❶: Data entries are actual data records
  - Hash function identifies page on which record lies.; no index used
- On Right: Heap file with Unclustered Hash Index on `sal`
  - Alternative ❷: Data entries are `<sal,rid>` pairs (with rid shown as arrow in figure)
  - Hash function identifies appropriate bucket

No ordering



No ordering

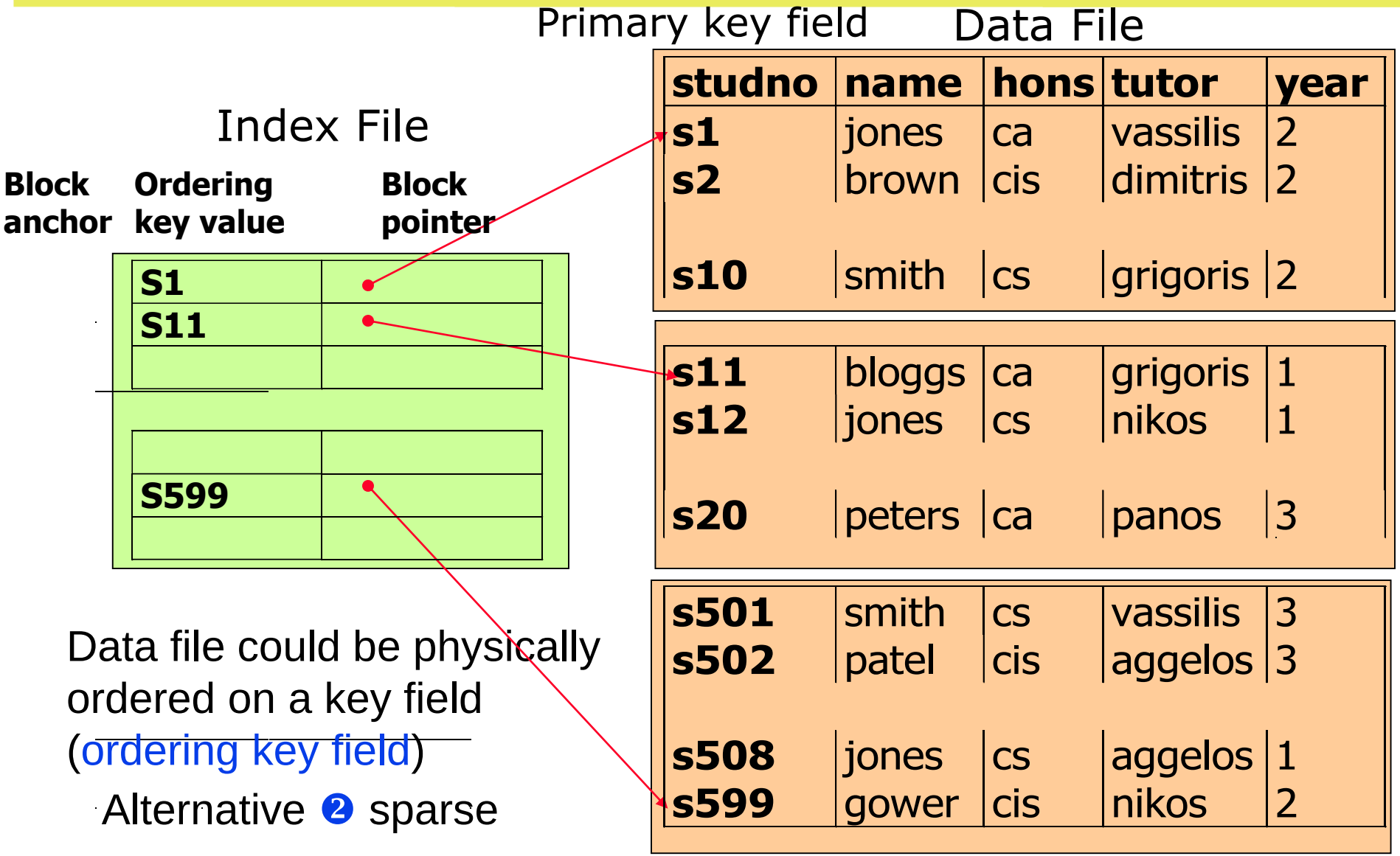
Index Data Entry k*

- Choice of alternative for data entries is orthogonal to the index structure used to speed up searches for data entries with a given search key value
- Index structures:
  - Tree-structured indexing
  - Hash-based indexing
- What kinds of selections do they support?
  - Selections of form field <op> constant
    - Equality selections (op is =)
    - Range selections (op is one of <, >, <=, >=, BETWEEN)
  - More exotic selections:
    - 2-dimensional (or n-dimensional) ranges (“east of Chania and west of Heraklion and North of Plakias and South of Spili”)
    - 2-dimensional (or n-dimensional) distances (“within 2 miles of Heraklion”)
    - Ranking queries (“10 restaurants closest to Knossos”)
    - Regular expression matches, genome string matches, etc.
    - One common n-dimensional index: R-tree

Index Classification

- Primary vs. Secondary indexes:
  - An index is called primary if the search key is a set of fields that includes the primary key
    - Usually implemented using Alternative ❶
  - An index is called unique if the search key contains a candidate key
  - No duplicates in the data entries
    - In general, secondary index usually contains duplicates
- Secondary indexes facilitate the answering of queries involving conditions on fields other than the primary key
  - Secondary indexes do not determine the location of records (i.e., blocks) since these are organized according to primary indexes
  - Alternatives ❶ and ❷ are not suitable for secondary indexes since they do not (explicitly, at least) allow for duplicate entries
- Typically, a relation has a primary index on its key attributes

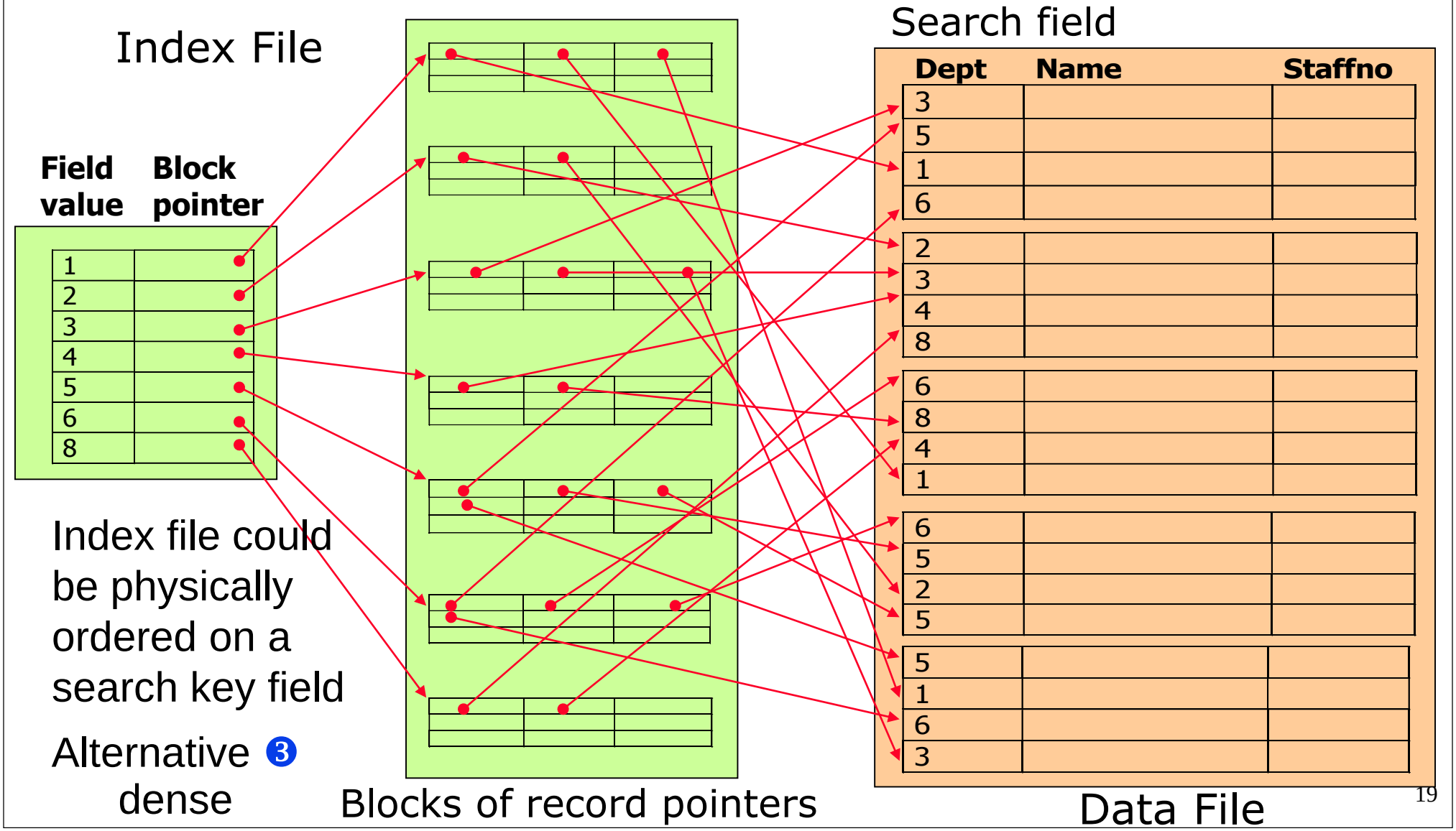
Primary Index on a Key Field



Data file could be physically ordered on a key field (ordering key field)

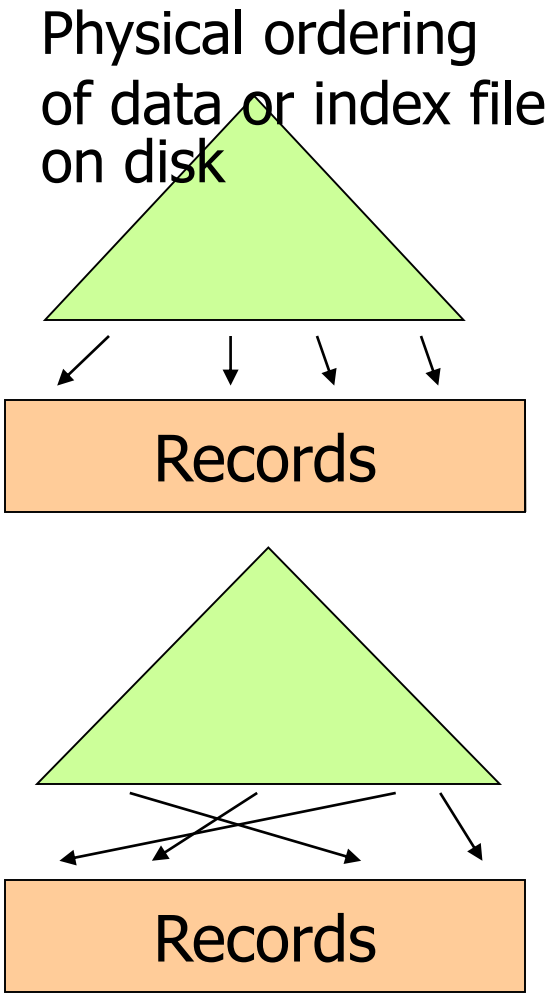
- Alternative ❷ sparse

Secondary Index on a Non-Key Field



Index Classification

- Clustered vs. Unclustered indexes:
  - An index is clustered if the order of data records in a file is the same as or `close to' the order of data entries in the index
  - An index is clustered if
    - It uses Alternative ❶ (by definition)
    - It uses Alternative ❷ or ❸ and the data records are sorted on the search key
- Clustered: file can be clustered on at most one search key (usually primary index)
  - Cost of retrieving data records through index varies greatly based on whether index is clustered or not!
- Unclustered: file does not constrain table organization (usually secondary index)
  - There might be several unclustered indexes per table



Clustered Index

- To build a clustered index:
  - Sort the records in (heap) file on the index search key
  - Leave some free space in each page to absorb future insertions

Index File

Clustering field value	Block pointer
1	
2	
3	

Data File

year	name	hons	tutor	studno
1				
1				
1				
2				
2				
2				
3				
3				
3				
3				
3				
3				

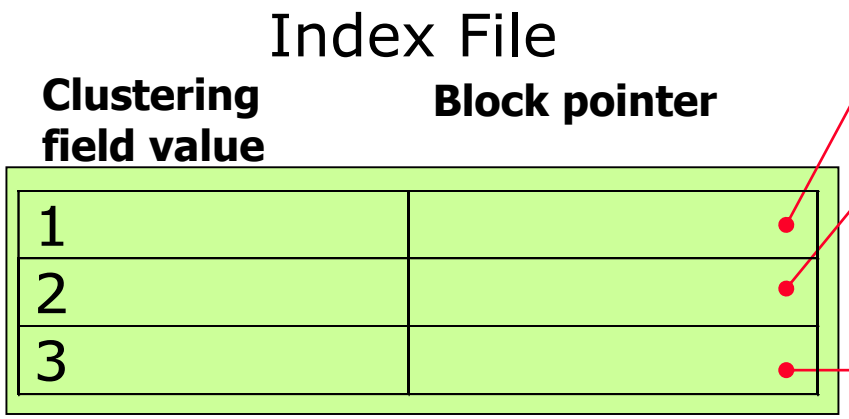
Data file is physically ordered on a non necessarily key field (clustering field)

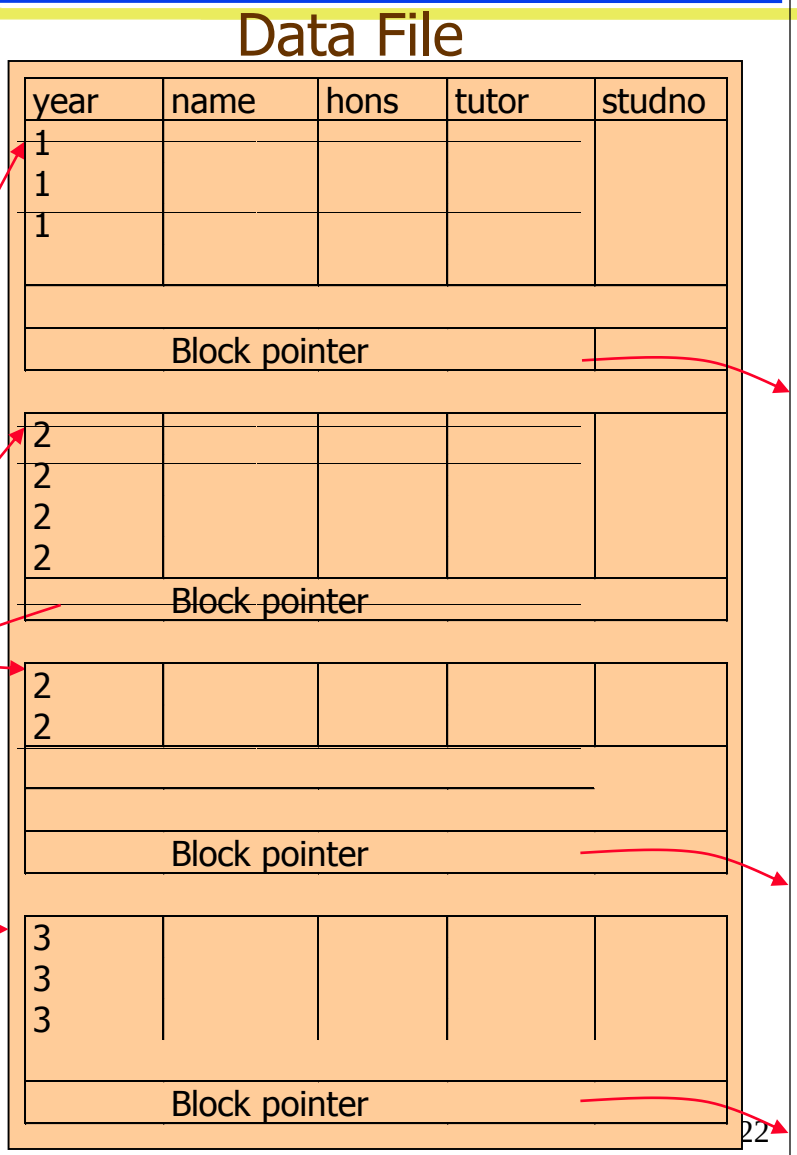
Alternative ❷ sparse

Clustered Index with Separate Blocks

- If free space is used up subsequently, further insertions to the page is handled using a linked list of overflow pages
- Need to reorganize file periodically to ensure good performance

- Clustered index is efficient for range searches but expensive to maintain





Separate blocks for each group of records with the same cluster field value

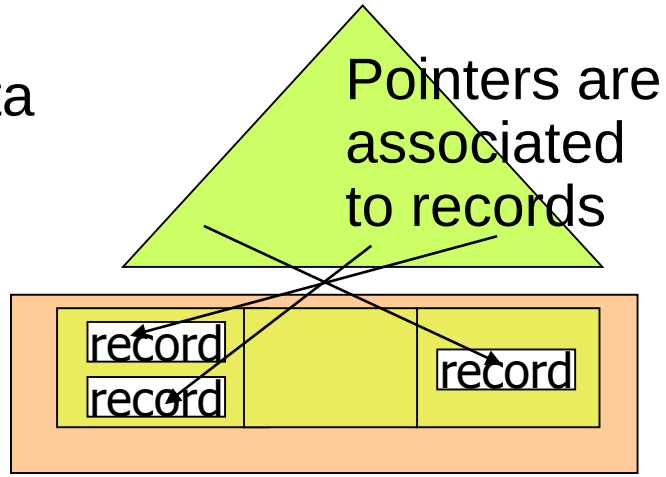
Alternative ❷ sparse

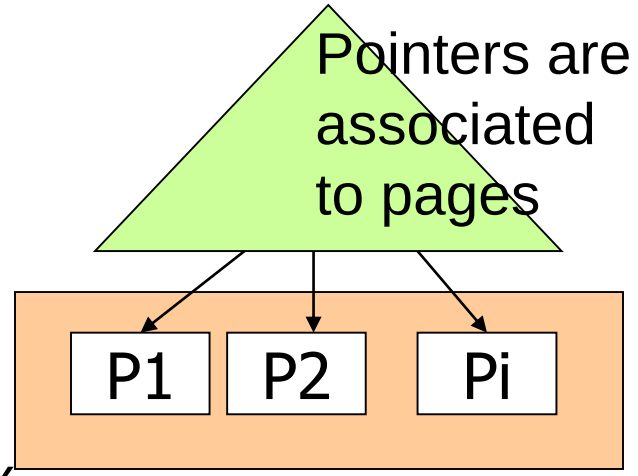
Example - Cost of Range Search

- Data file has 10.000 pages, 100 rows in range
- Page transfers for rows (assuming 20 rows/page):
  - Heap:
    - 10.000 (entire file must be scanned)
  - File sorted on search key:
    - $\log_2 10.000 + (5 \text{ or } 6) \approx 19$
  - Unclustered index:
    - ≤ 100
  - Clustered index:
    - 5 or 6
- Page transfers for index entries (assuming 200 entries/page)
  - Heap and sorted: 0
  - Unclustered secondary index: 1 or 2 (all entries for rows in the requested range must be read)
  - Clustered secondary index: 1 (only first entry must be read)

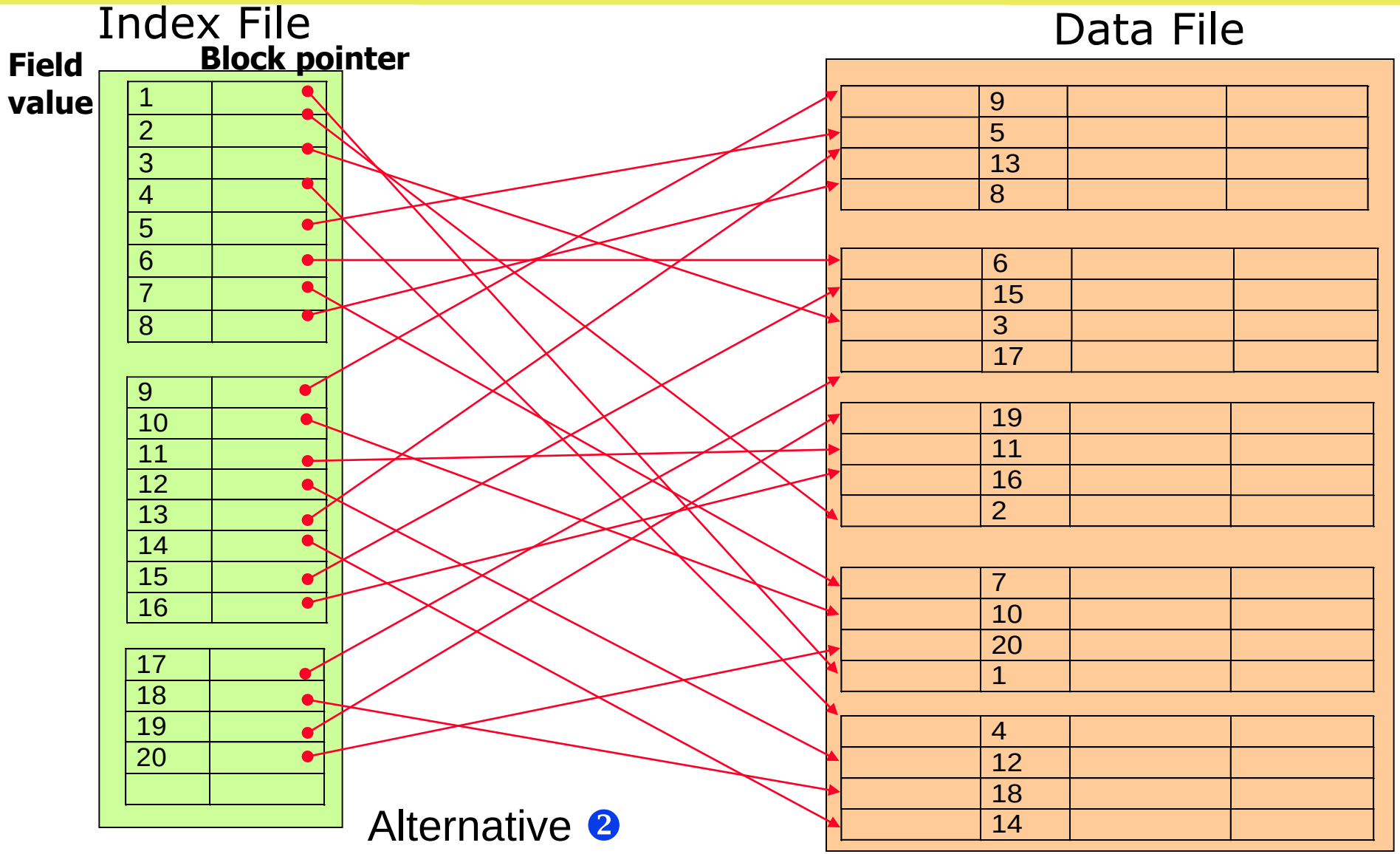
Index Classification

- Dense vs. Sparse indexes:
  - An index is dense if it contains at least one data entry for every search key value
    - Alternative ❶ always leads to build dense index
    - Alternative ❷ can be used to build a dense or sparse index
      - several data entries can have same search key value if there are duplicates
    - Alternative ❸ can be used to build a dense index
  - An index is sparse if it contains one data entry for each records' page in the data file
    - indicates the block of records, so
      - sparse indexes take less space
      - index scanning is faster
      - but...no existence test based on the index
- A file can have one sparse index and many dense indexes, because a sparse index relies on a unique physical ordering (clustering) of the data file on disk



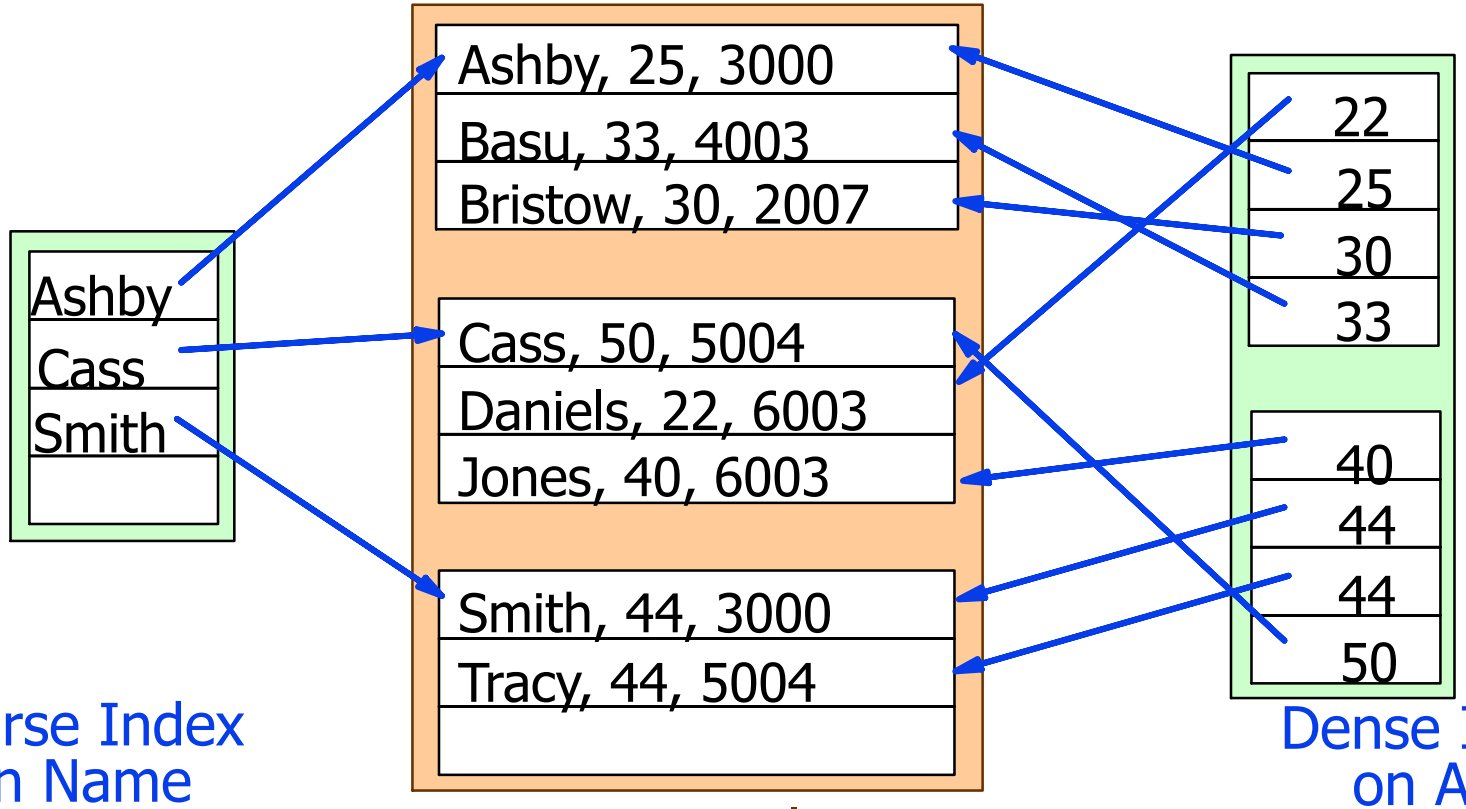


Dense Secondary Index on a non-ordering Key Field



Example

- On Left: same order of index data entries and records in the file
  - Primary, Clustered, Sparse index on Name
- On Right: different order of index data entries and records in the file
  - Secondary, Unclustered, Dense Index on Name

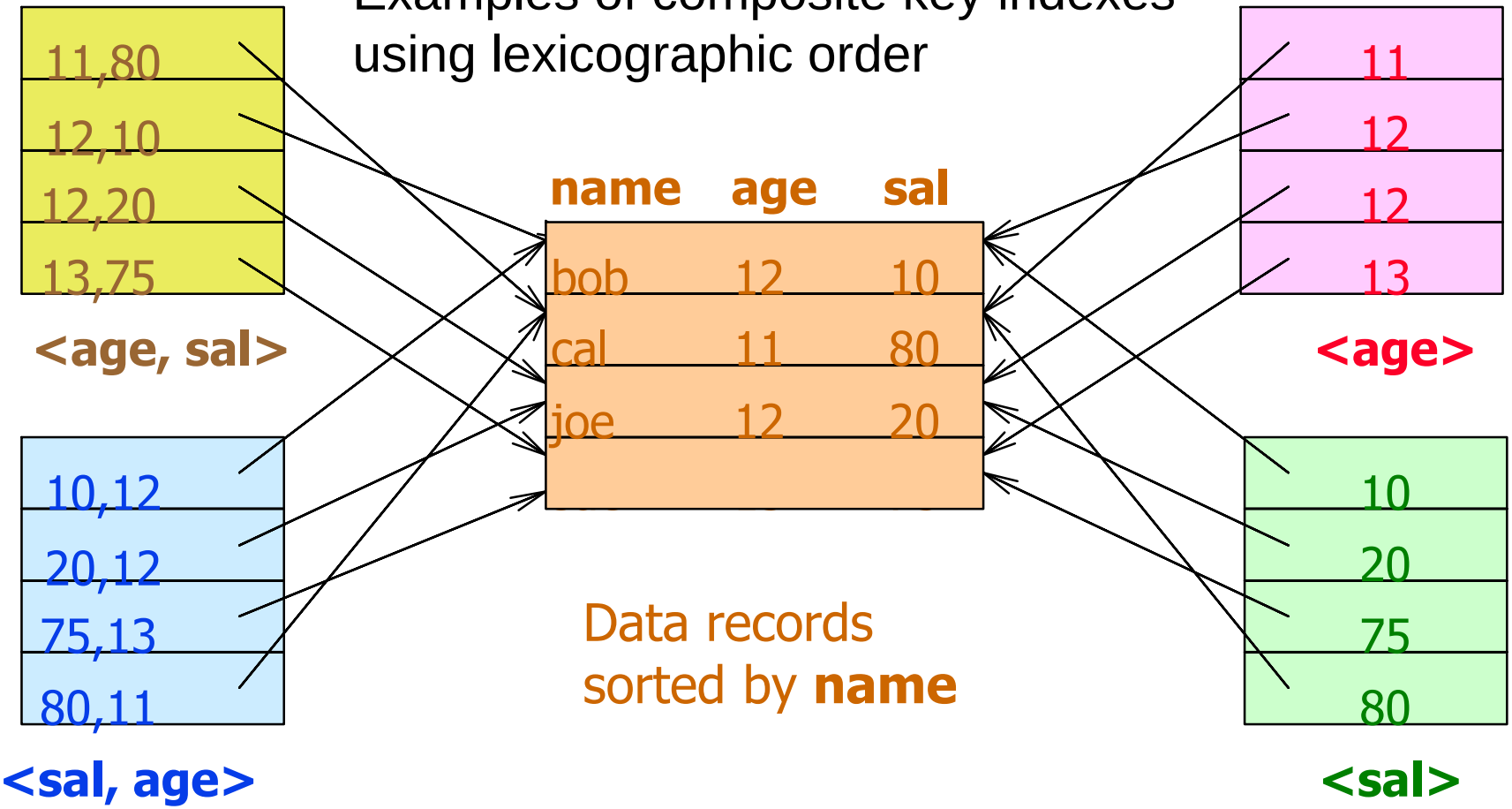


Index Classification

- Composite Index (also called a concatenated index): is an index that you create on multiple fields in a table
  - Fields in a composite index can appear in any order and need not be adjacent in the table
- Composite Search Keys: Search on a combination of fields
  - Equality query: Every field value is equal to a constant value e.g., Given a composite index with key <age,sal> :
    - age=20 and sal =75
  - Range query: Some field value is not a constant e.g., :
    - age > 20 (any value for sal)
    - age=20 and sal > 10
- Data entries in index sorted by search key to support range queries
  - Lexicographic order
  - Like the dictionary, but on fields, not letters!

Example

Examples of composite key indexes using lexicographic order



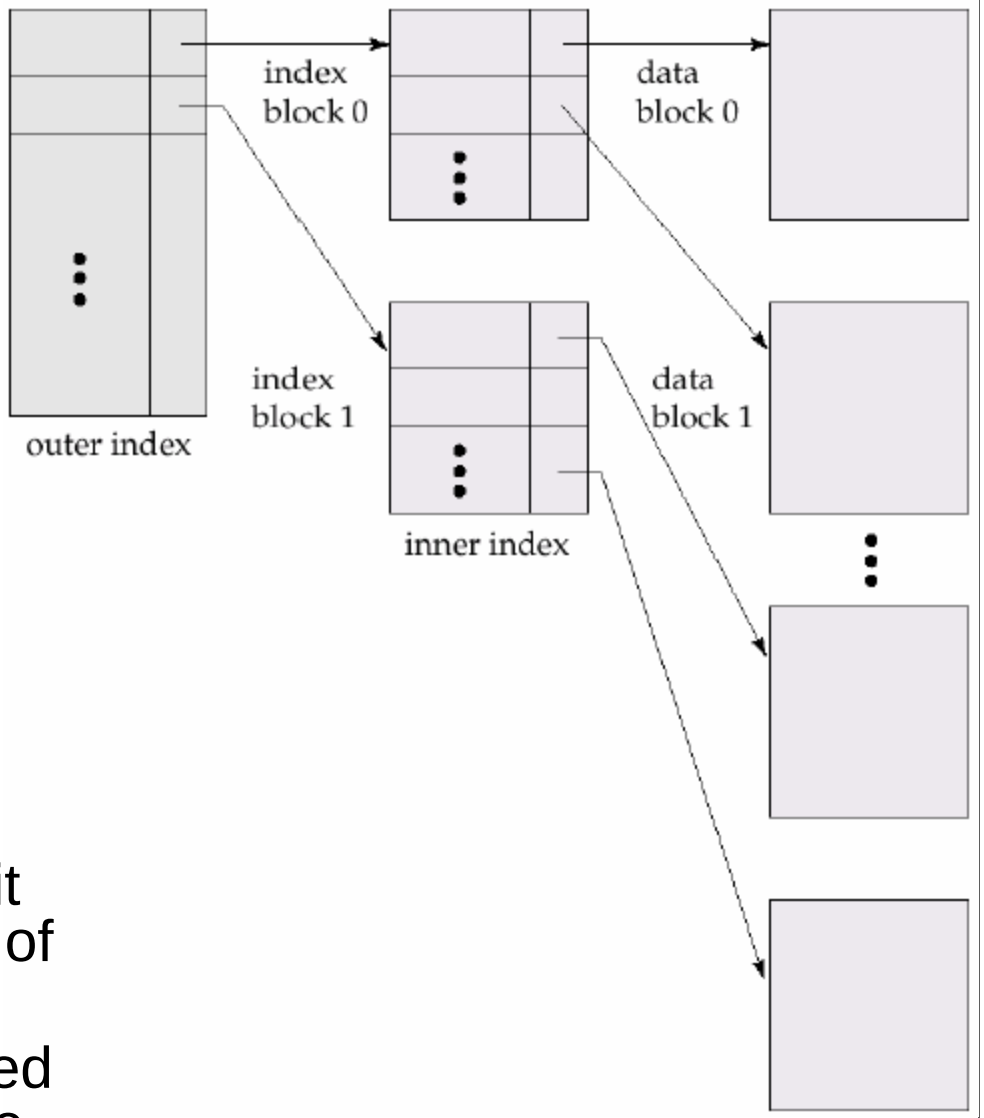
Data entries in index sorted by **<sal,age>**

Alternative ❷

Data entries in index sorted by **<sal>**

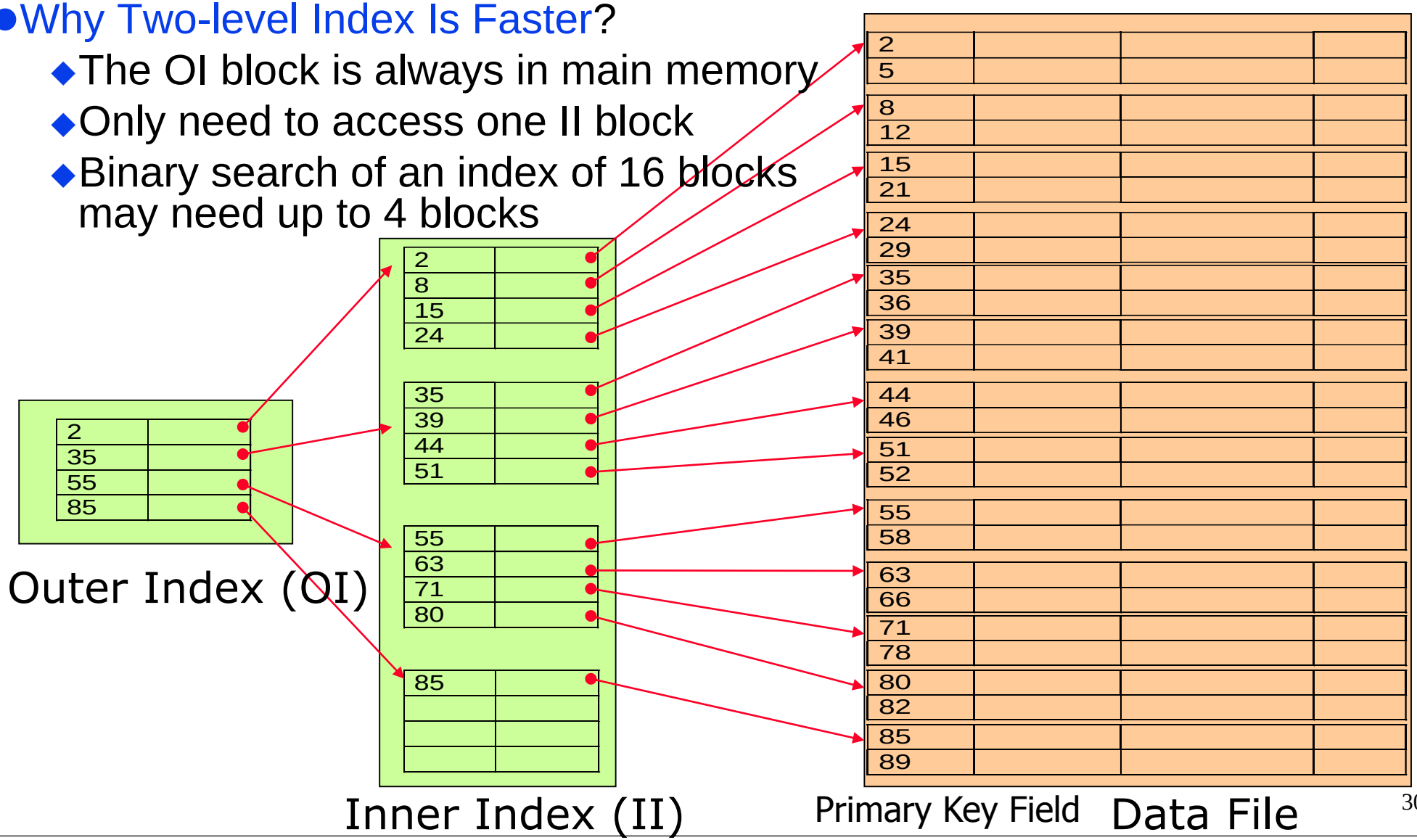
Multilevel Indices

- Even with a sparse index, index size may still grow too large to fit in main memory
  - Search a sparse index not completely in main memory is expensive
- Multilevel index: treat primary index kept on disk as a sequential file and construct a sparse index on it
  - Reduce number of disk accesses to index records
  - outer index – a sparse index of primary index
  - inner index – the primary index file
  - If even outer index is too large to fit in main memory, yet another level of index can be created, and so on
  - Indices at all levels must be updated on insertion or deletion from the file



Multi-leveled Indexes: an Index for an Index

- Why Two-level Index Is Faster?
  - The OI block is always in main memory
  - Only need to access one II block
  - Binary search of an index of 16 blocks may need up to 4 blocks



Types of Search Keys

- Unordered data files
  - lots of secondary indexes
  - sometimes for unclustered primary indexes*
- Specify ordering field for file
  - primary / clustered index
  - field used most often for joins

	Ordering Field	Non-ordering Field
Key field	Primary Index	Secondary Index (key)
Non-key Field	Clustered Index	Secondary Index (non-key)

*In the bibliography is considered that primary index are clustered

- Unclustered primary indexes are called secondary key indexes

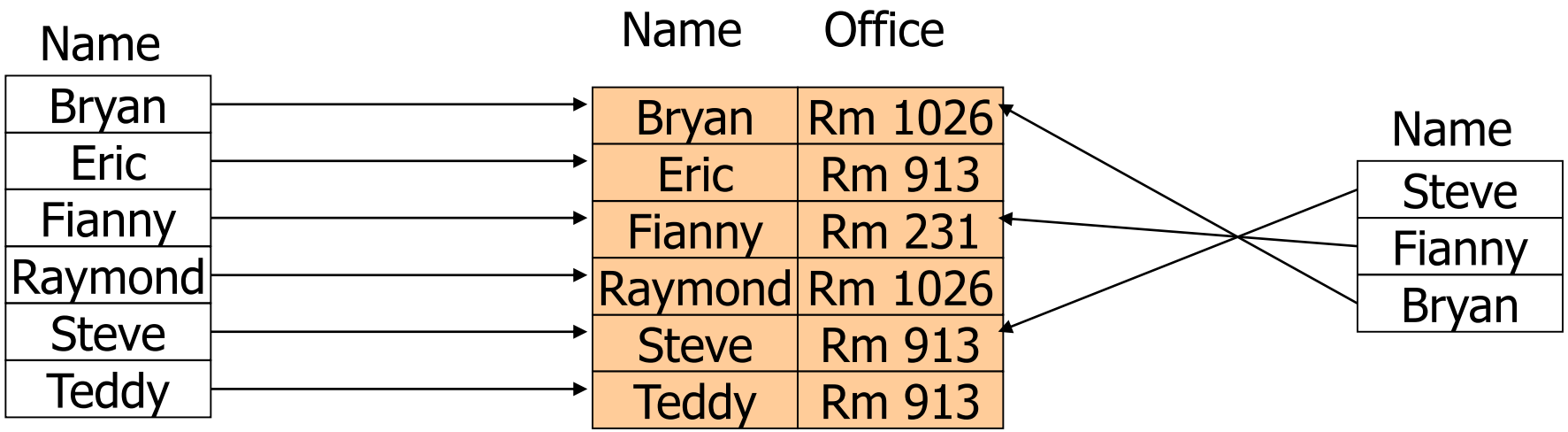
Types and Properties of Indexes

Properties of Index Types

Type of Index	Number of (first level) Index Entries	Dense or Sparse	Block Anchoring on the Data File
Primary	Number of blocks in data file	Sparse / Dense*	Yes / No*
Clustered	Number of distinct index field values	Sparse	Yes Alt ❶ / No Alt ❷&❸
Secondary (key*)	Number of records in data file	Sparse	No
Secondary (non-key) Alt ❶	Number of records in data file	Dense	No
Secondary (non-key) Alt ❷ & ❸	Number of distinct index field values	Sparse	No

Example

- Assume Name is the primary key and indexes implemented with Alt ❷

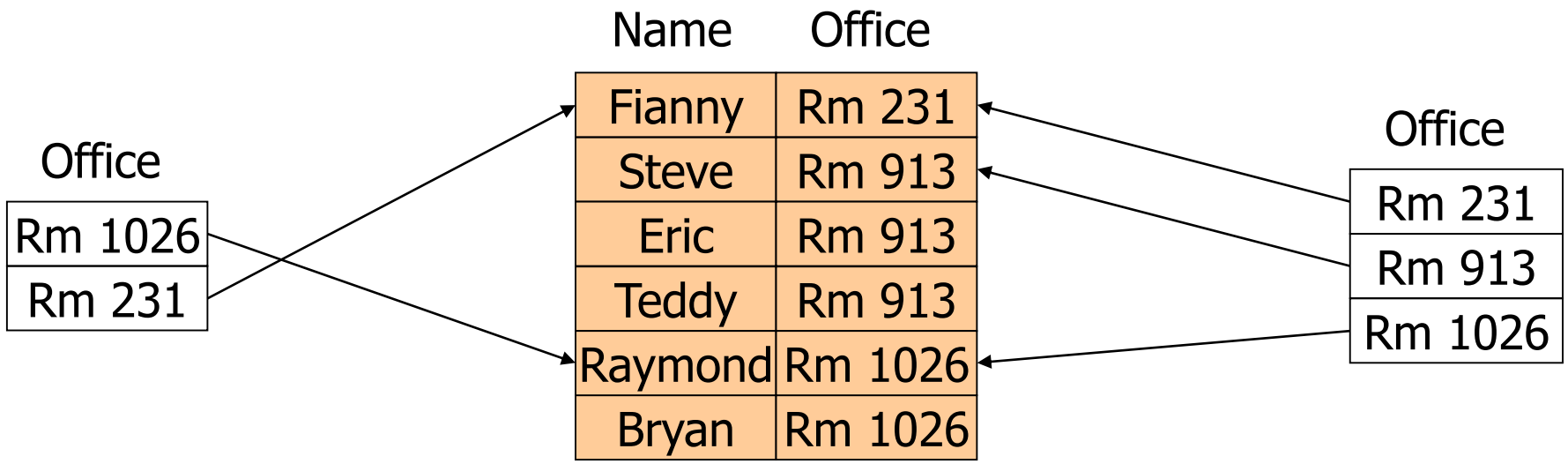


1. Primary
2. Clustered
3. Dense

1. Primary
2. Unclustered
3. Sparse

Example

- Assume Name is the primary key and Indexes implemented with Alt ❷



1. Secondary
2. Unclustered
3. Sparse

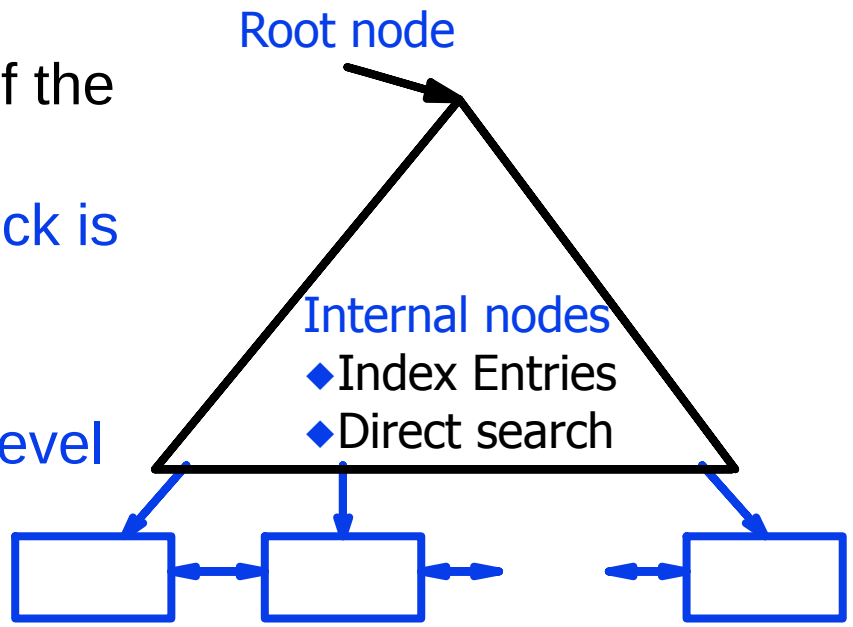
1. Secondary
2. Clustered
3. Dense

Tree-Structured Indexes

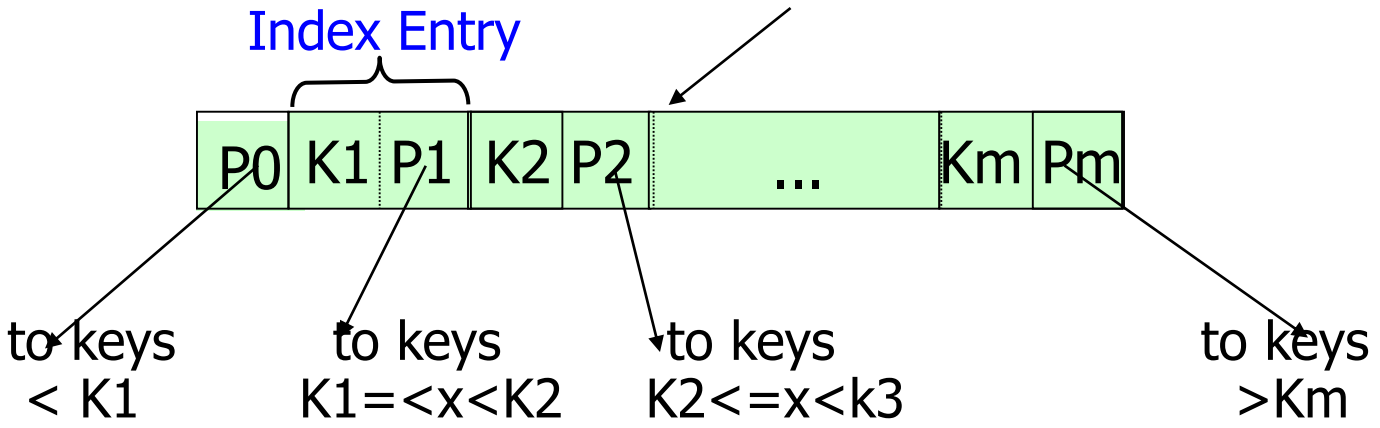
- Recall: 3 alternatives for data entries k*:
  - `Actual data record with search key value k`
  - `<k, rid of data record with search key value k>`
  - `<k, list of rids of data records with search key k>`
- Choice is orthogonal to the indexing technique used to locate data entries k*; index quality is measured in terms of
  - Access time
  - Insertion/deletion time
  - Search condition types
  - Disk and RAM Space requirements
- Tree-Structured Indexes: Efficient support for range search, insertion and deletion
  - Indexed Sequential Access Method (ISAM)
    - Static index structure where insertions and deletions affect only leaf pages
  - B+ tree
    - Dynamic index structure that adjusts gracefully to insertions and deletions

B+ -Tree Indexes

- The most commonly used variation of B-trees
  - "B" stands for "balanced": all paths from the root to a leaf have the same length
  - Automatically maintain the number of indexing levels required for the size of the file being indexed (self-organized)
  - Manage block space so that each block is at least half-full
  - No overflow blocks needed
- Can be seen as a general form of multi-level indexes
- Measures
  - Order: the (maximum) number of indexing field values at each node
  - Height: the number of indexing levels from root to any leaf



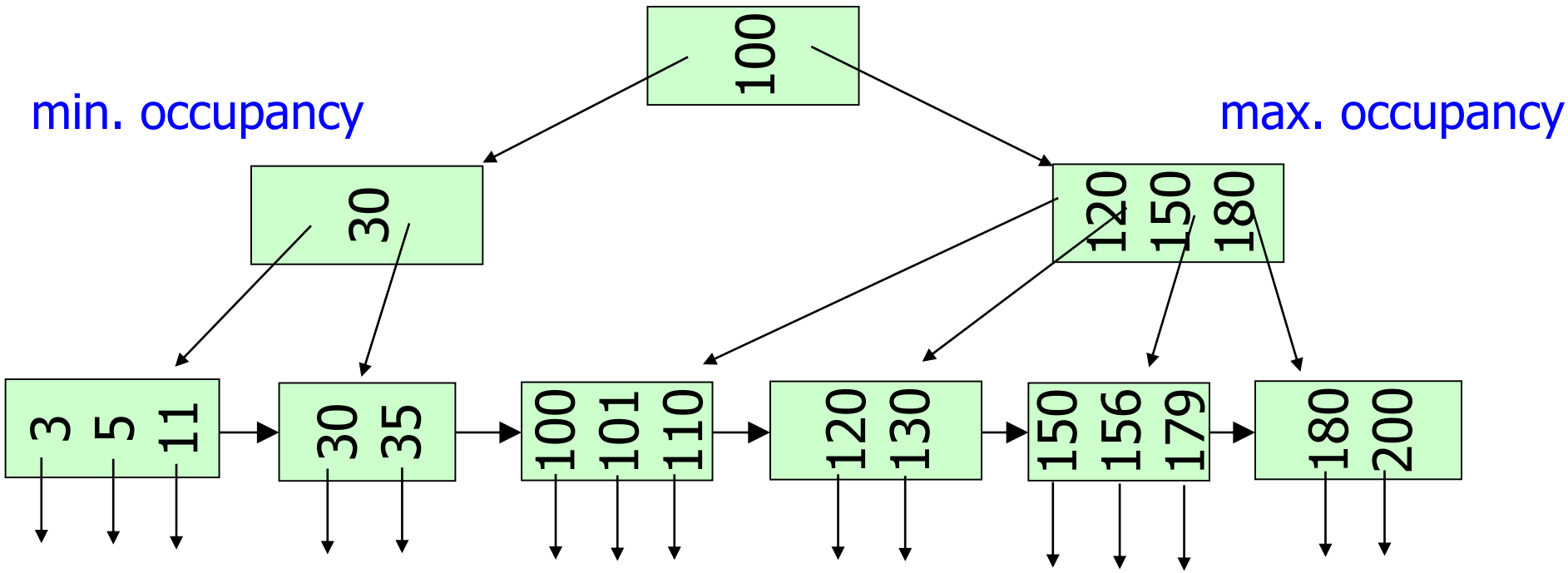
B+-Tree Node



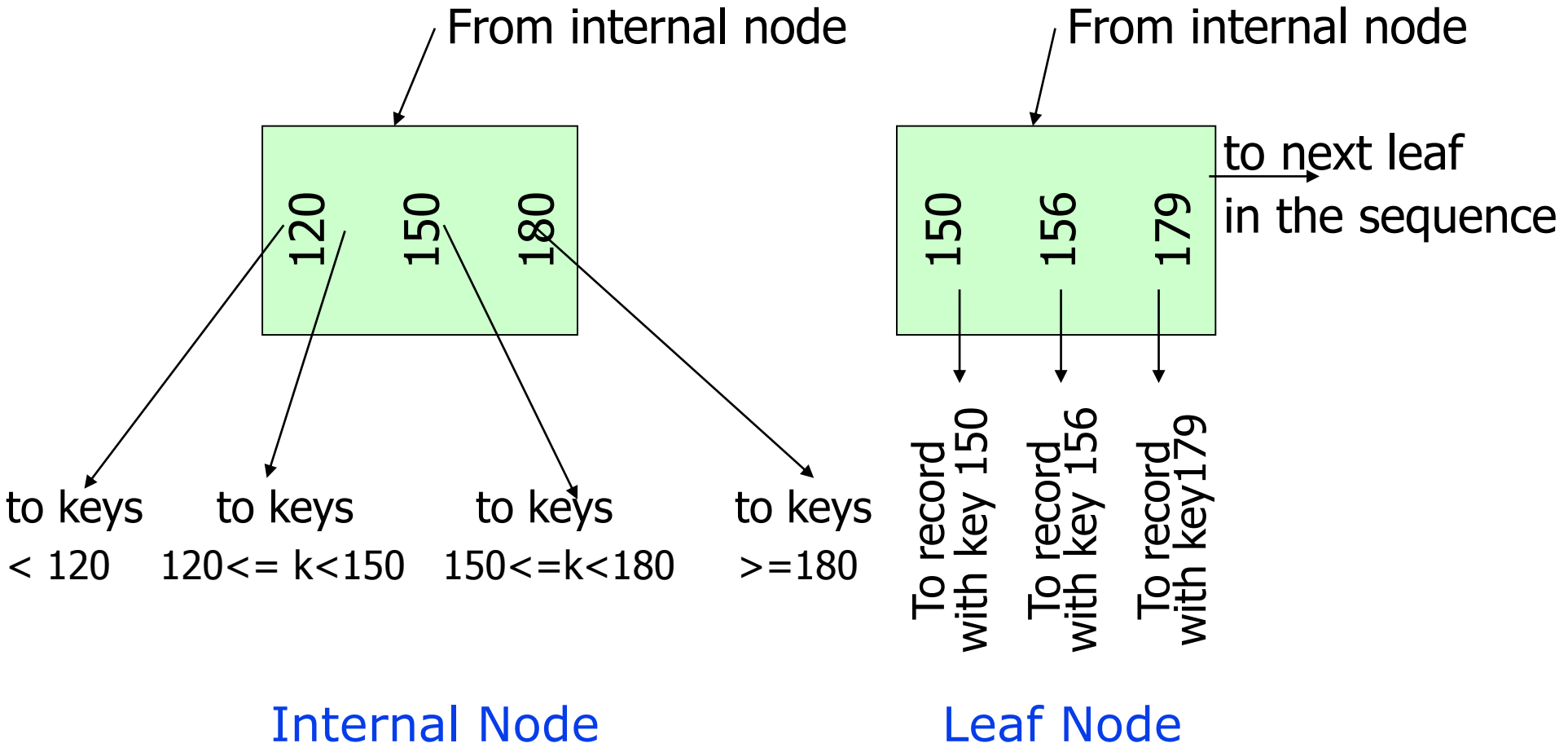
- A B+tree node contains m entries (occupancy) where
  - n/2 <= m <= n (half-full), n = order of tree (fanout)
  - m + 1 pointers to children
    - P_0 points to tree where K < K_1
    - P_i points to subtree where key values K_i<= K < K_{i+1}
    - P_n points to tree where K >= K_m
- A B-tree has record pointers also in non-leaf nodes
  - In practice, B+trees are preferred, and widely used

B+-Tree Example

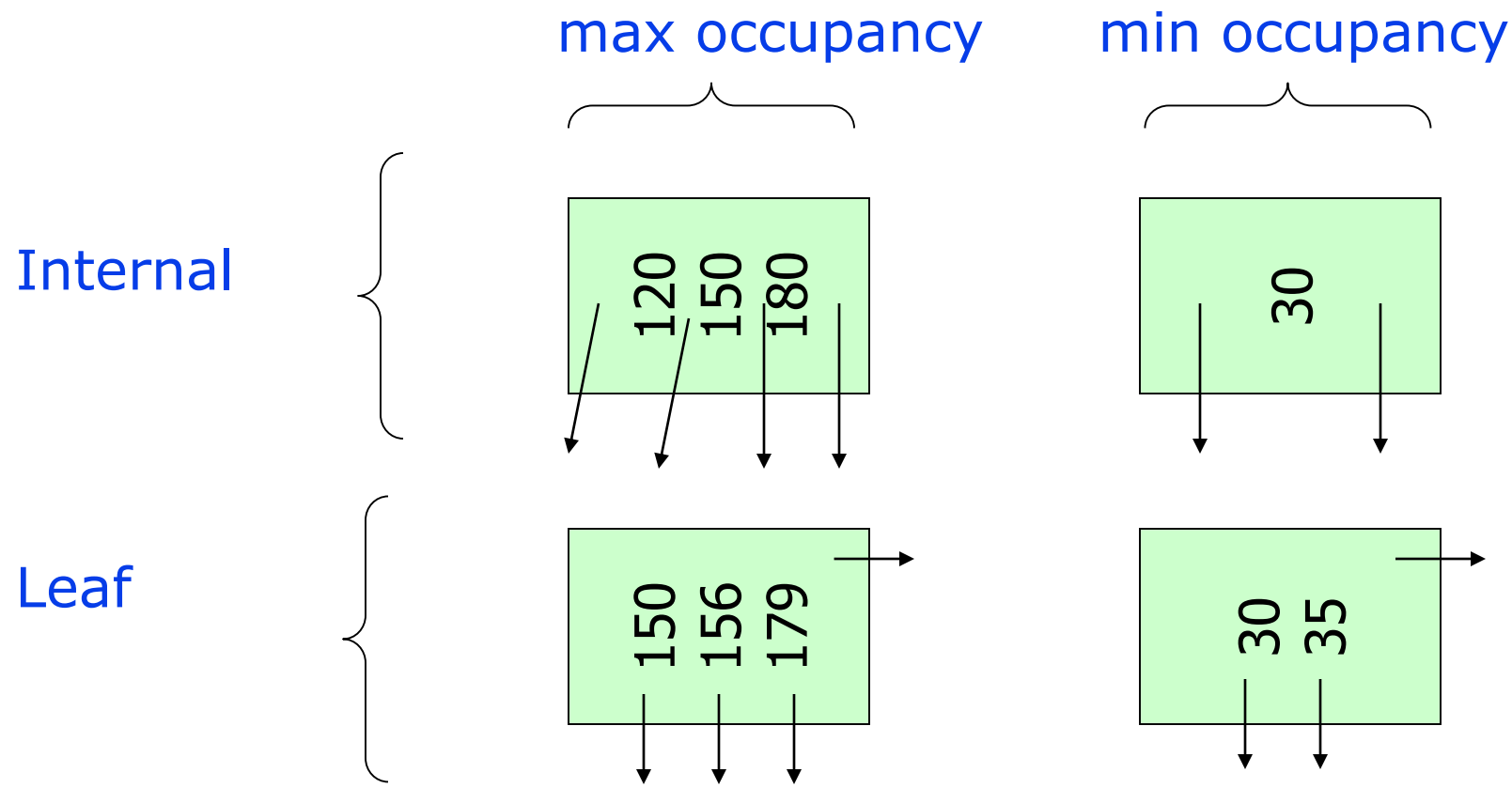
- Order n=3, integer key values



Sample non-Leaf (Internal) & Leaf Nodes



B+ -Tree Node Capacity



The Shape of a B+ -Tree

- Invariant rules on nodes
  - The root has at least two pointers (balanced tree)
  - All leaves are at the same lowest level (balanced tree)
  - Pointers in leaves point to records except for "sequence pointer"
  - Minimum 50% space occupancy for each node except the root; thus use at least (ceiling)
    - Internal: $\lceil (n+1)/2 \rceil$ pointers to nodes at the lower levels
    - Leaf: $\lfloor (n+1)/2 \rfloor$ pointers to records; `i-th` pointer points to a record with the `i-th` key

	Max ptrs	Max keys	Min ptrs→data	Min keys
Internal (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	2(*)	1

(*) 1, if only one record in the file

B+ -Tree Node Size

- Parameter "n" affects:
  - The height of the tree (Why?): $\lceil \log_n(N) \rceil$, where N = # of leaf nodes
    - The time of binary search within a node is very small compared to the disk I/O time
  - Thus, we want to have (on average) a large n to reduce the tree height
- If a node corresponds to a block, we choose a largest n to fill up the block space
  - n search key values and n+1 pointers should fit in a block
  - The value of n can be determined by the block size and the (key, pointer) types and sizes !!!
- Example: B=4096, K: integer (4 bytes), pointer: 8 bytes. How many keys and pointers fit in a node (= index block)?
  - $4n+8(n+1) <= 4096$, $n = \lfloor 4096 / 12 \rfloor = 340$
  - 340 keys and 341 pointers could fit in a node
  - 171 … 341 pointers in an internal node

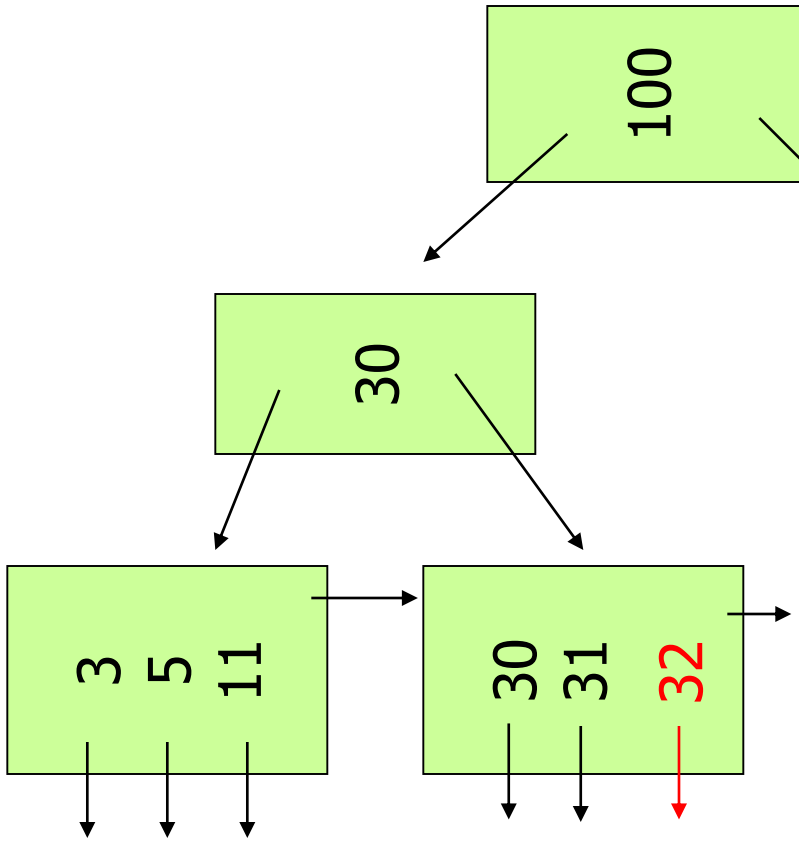
B+ -Tree Node Size in Practice

- Order (fanout) concept replaced by physical space criterion in practice (`at least half-full')
  - Index (internal) blocks can typically hold many more entries than leaf blocks (if we use Alternative ❶)
  - Variable sized records and search keys mean different nodes will contain different numbers of entries
  - Even with fixed length fields, multiple records with the same search key value (duplicates) can lead to variable-sized data entries (if we use Alternative ❸)
- Many real systems are even sloppier than this --- only reclaim space when a page is completely empty

Record Insertion

- When a record is inserted in the data file, the B+-tree must be changed accordingly:
  1. simple case
     - leaf not full: just insert (`key, pointer-to-record`)
  2. leaf overflow
  3. Internal node overflow
  4. new root
- Algorithm:
  - find the leaf node where the new key should be placed, and insert it in the block if there is space
  - if not, split the leaf in two and divide the keys between the two nodes so that each of them is at least half-full
  - if a new (`key, pointer`) needs to be added to the level above, apply the insertion strategy at the parent node
  - exception: if insertion propagates up to the root node and the root node has no room, split the root in two and create a new root node one level up with the nodes resulting from the split as its children

Record Insertion Example

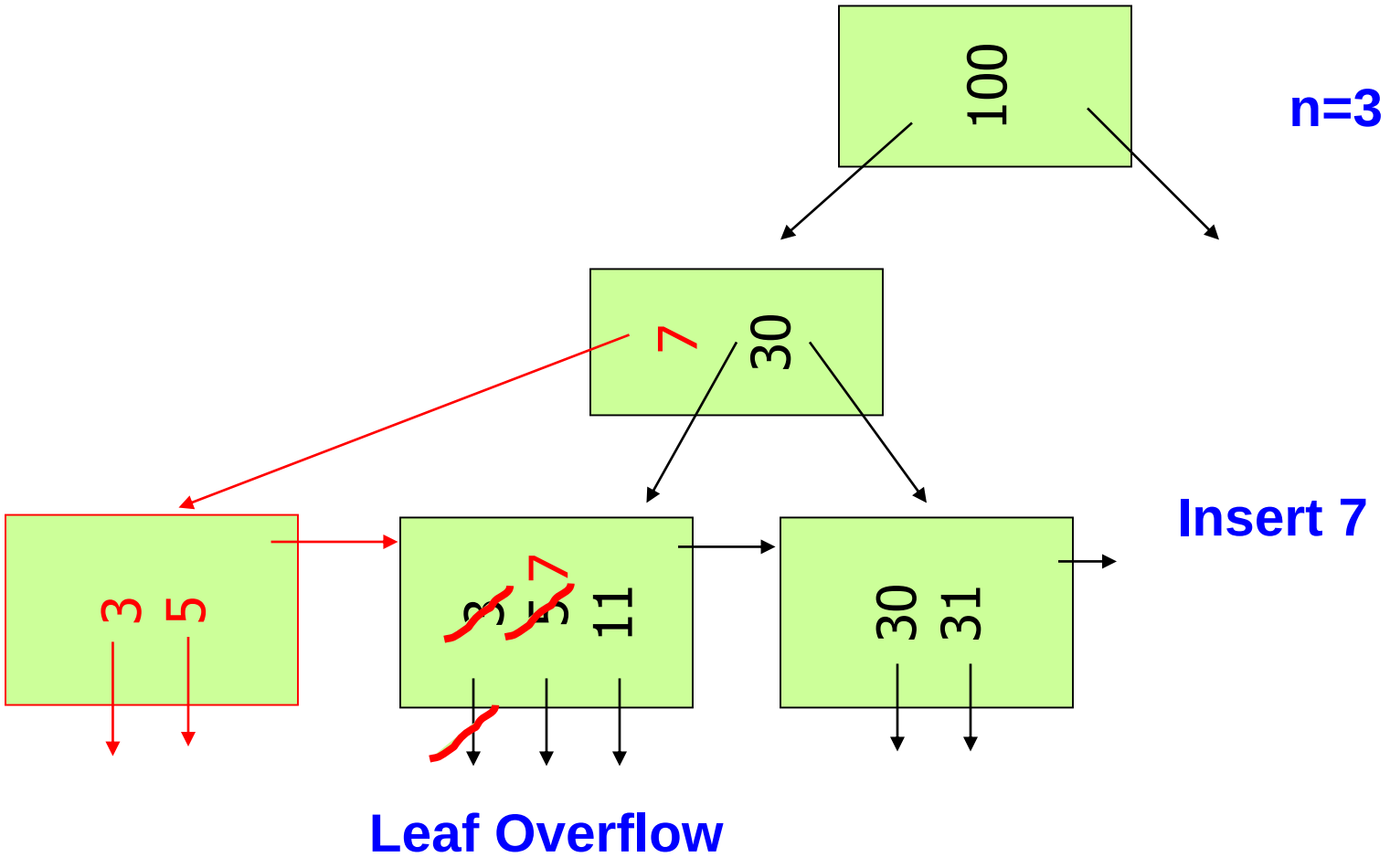


n=3

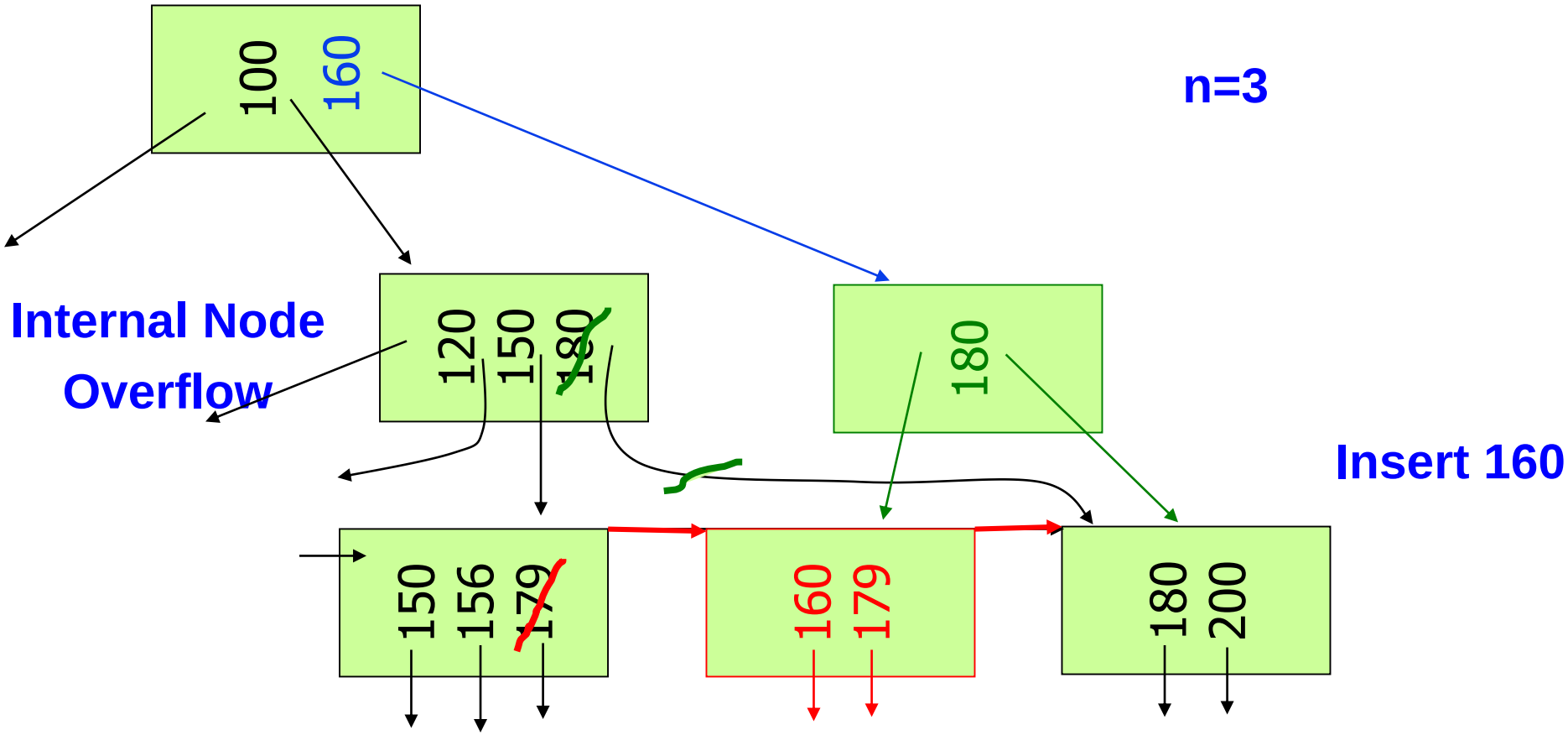
Insert 32

There is space available

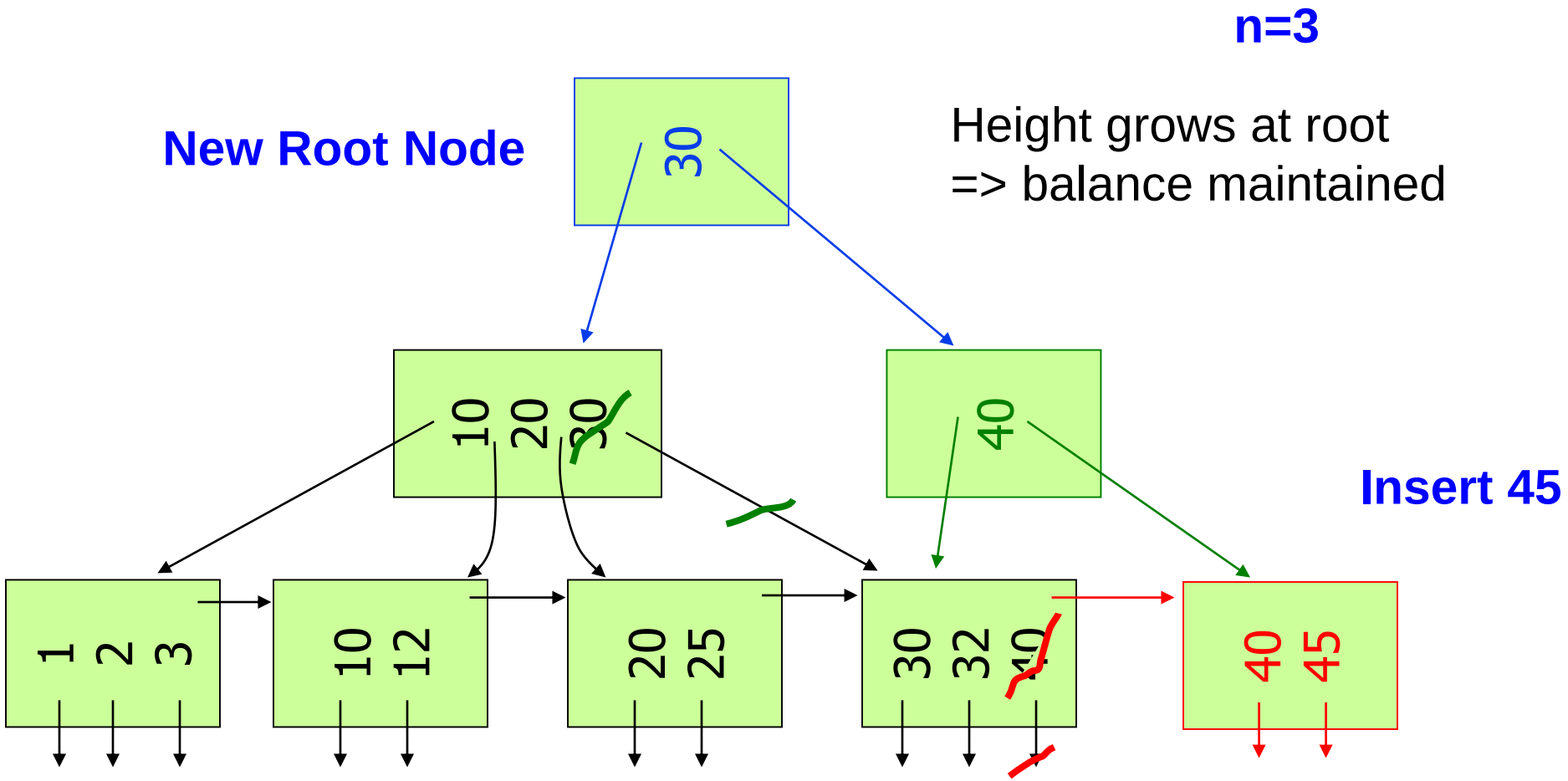
Record Insertion Example



Record Insertion Example



Record Insertion Example



Record Deletion

- When a record is deleted in the data file, the B+-tree must be changed accordingly:
  - ❶ Simple case: no underflow; Otherwise ...
  - ❷ Borrow keys from an adjacent sibling (if it doesn't become too empty); Else ...
  - ❸ Coalesce with a sibling node
  - ❹ Cases ❶, ❷ or ❸ at internal nodes

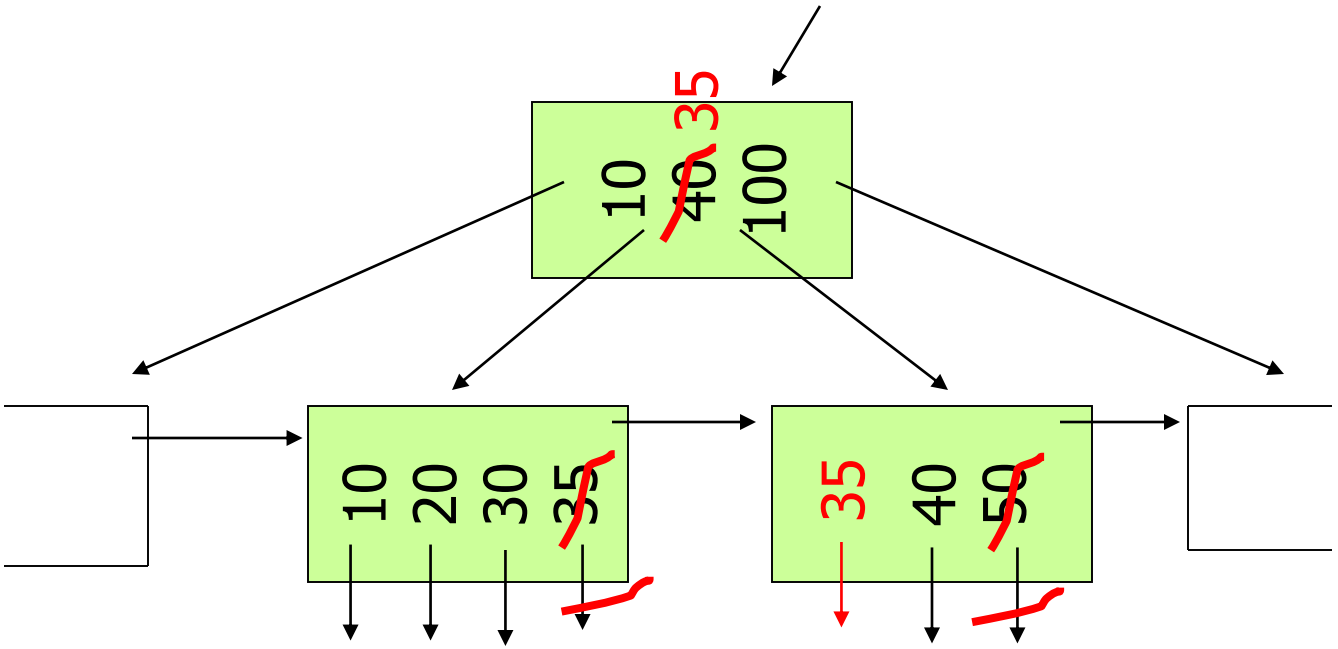
- Algorithm:
  - find the record `(key, pointer)` and delete it from the block where it resides;
    - If the block is still half full, nothing needs to be done

Record Deletion

- If deletion occurs at a record that is exactly half-full, tree needs to be adjusted. Let N be the node at which the deletion occurs:
  - if one of the adjacent siblings of N has more than the minimum (`key,pointer`), then one can be moved to N
    - The parent of N may need to be adjusted as a result of this change
  - if neither adjacent sibling can be used for providing a key, then node N and one of its siblings together have no more than the max number of records allowable
    - These nodes can be merged (one is deleted)
    - Keys at the parent need to be adjusted (by deleting a record)
      - If the parent remains half full, no more changes are needed
      - If not, the deletion is applied recursively at the parent node

Record Deletion Example

n=4



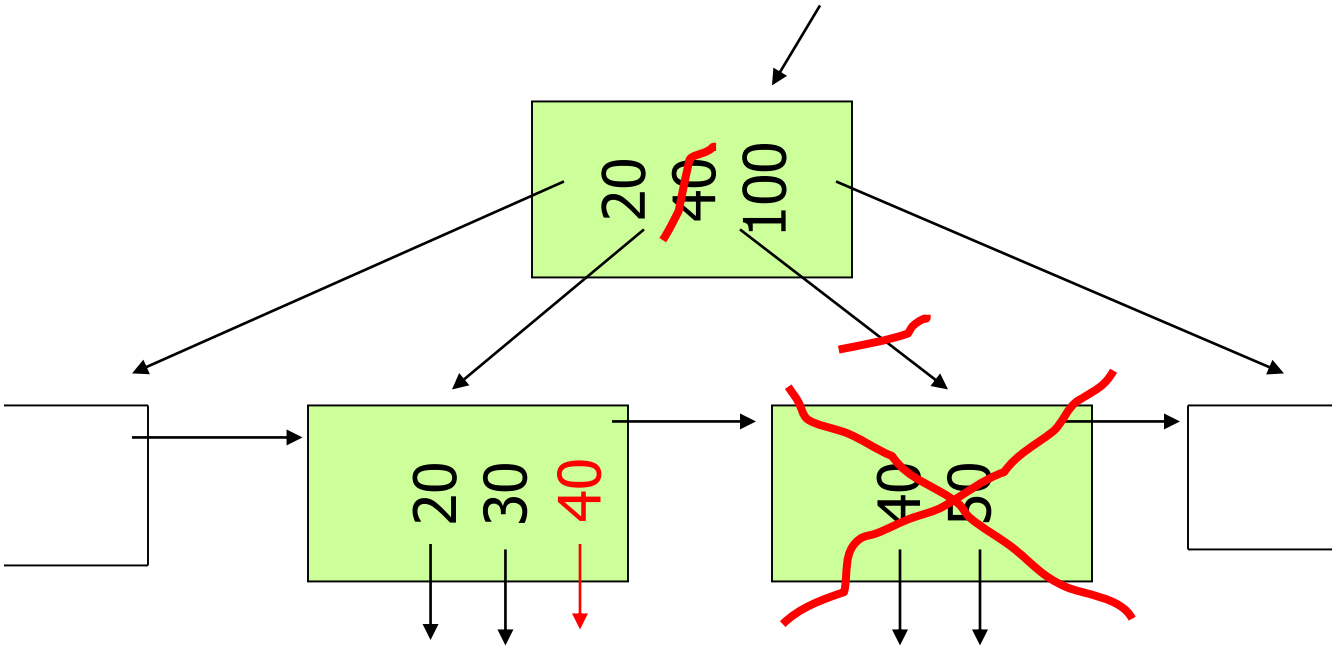
Delete 50

Borrow keys

=> min # of keys
in a leaf = $\lfloor 5/2 \rfloor = 2$

Record Deletion Example

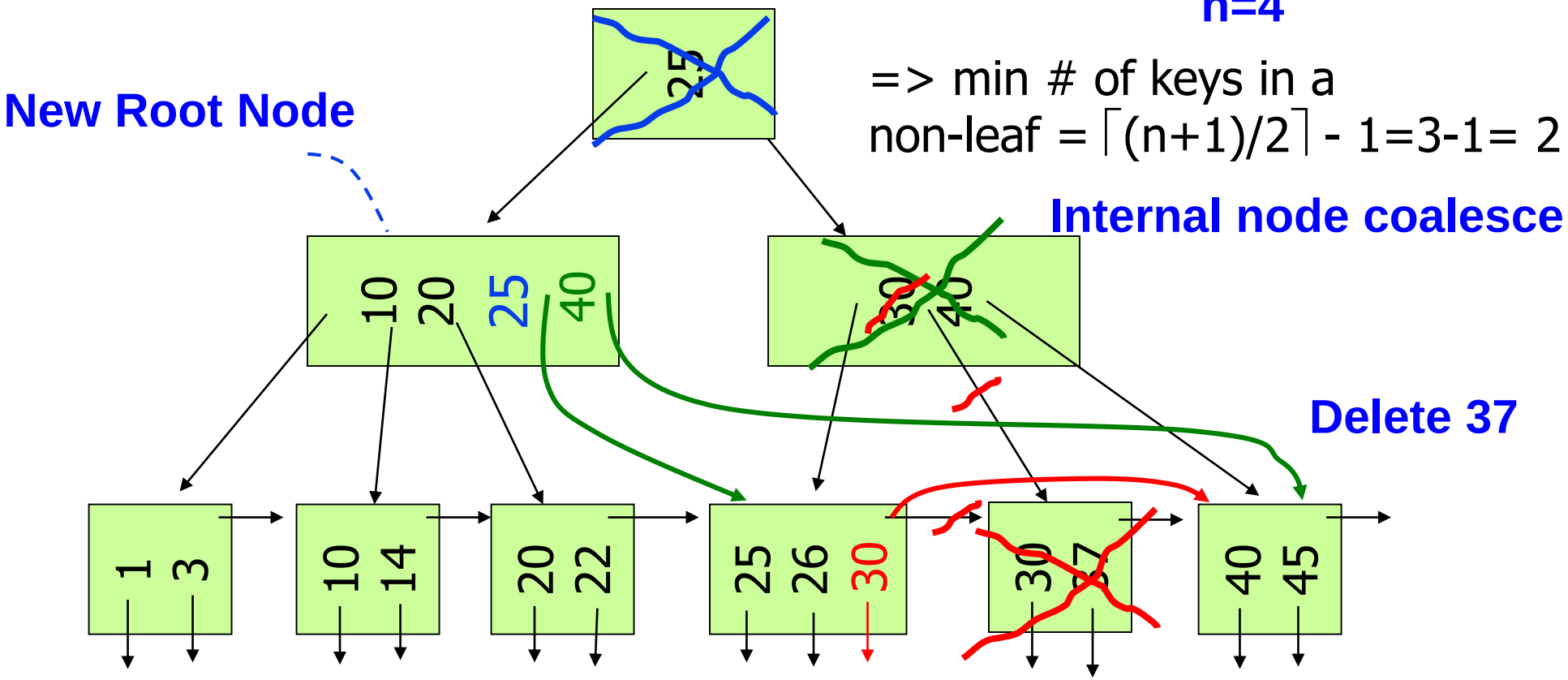
n=4



Delete 50

Coalesce with a leaf sibling

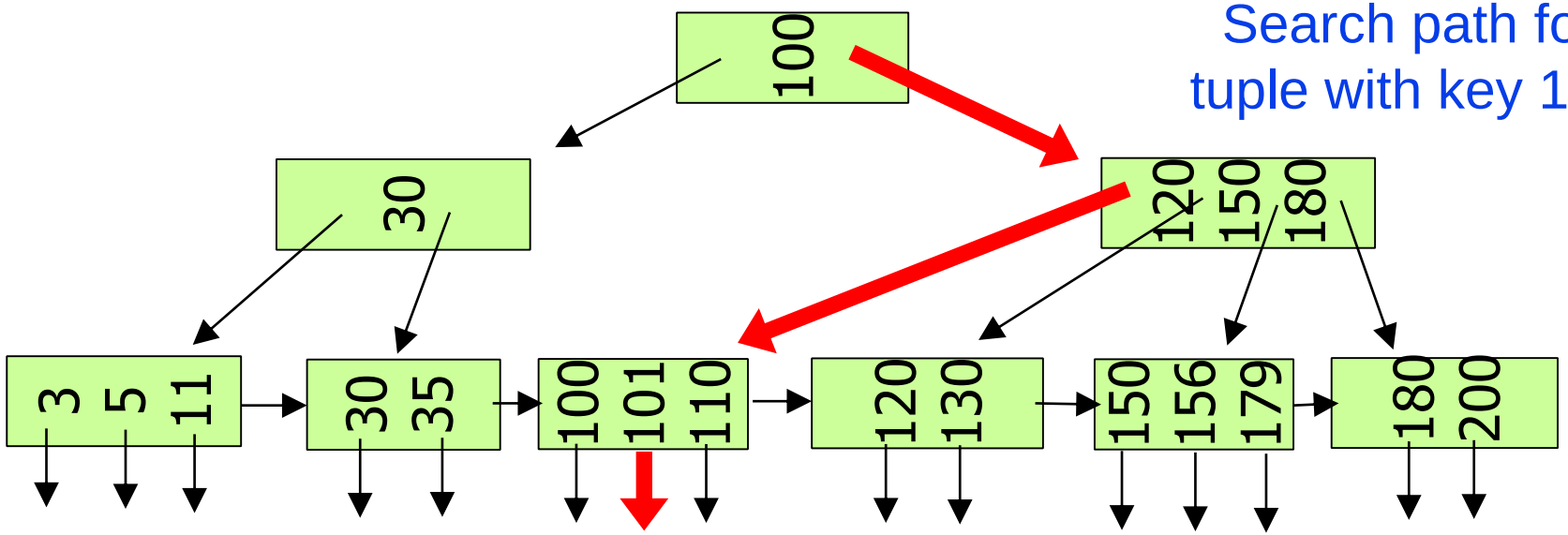
Record Deletion Example



B+ -Tree Deletions in Practice

- Often, coalescing is not implemented
  - Too hard and not worth it!
    - later insertions may return the node back to its required minimum size
  - Compromise: Try redistributing keys with a sibling;
    - If not possible, leave it there
  - if all accesses to the records go through the B+-tree, can place a "tombstone" for the deleted record at the leaf
    - Periodic global rebuilding may be used to remove tombstones when they start taking too much space

Searching B+ -Trees: Equality Queries



- Assuming no duplicates are contained, we need to find records with a given search-key value K
  1. If at a leaf, search the key values found there
     - If the i-th leaf entry is K, then the i-th pointer points to the record
  2. If at an internal node with keys K_1, K_2, …, K_n then
     - if $K < K_1$, go to the 1st child; if $K_{i-1} <= K < K_i$, go to the i-th child
  3. Recursively apply this process at the child node

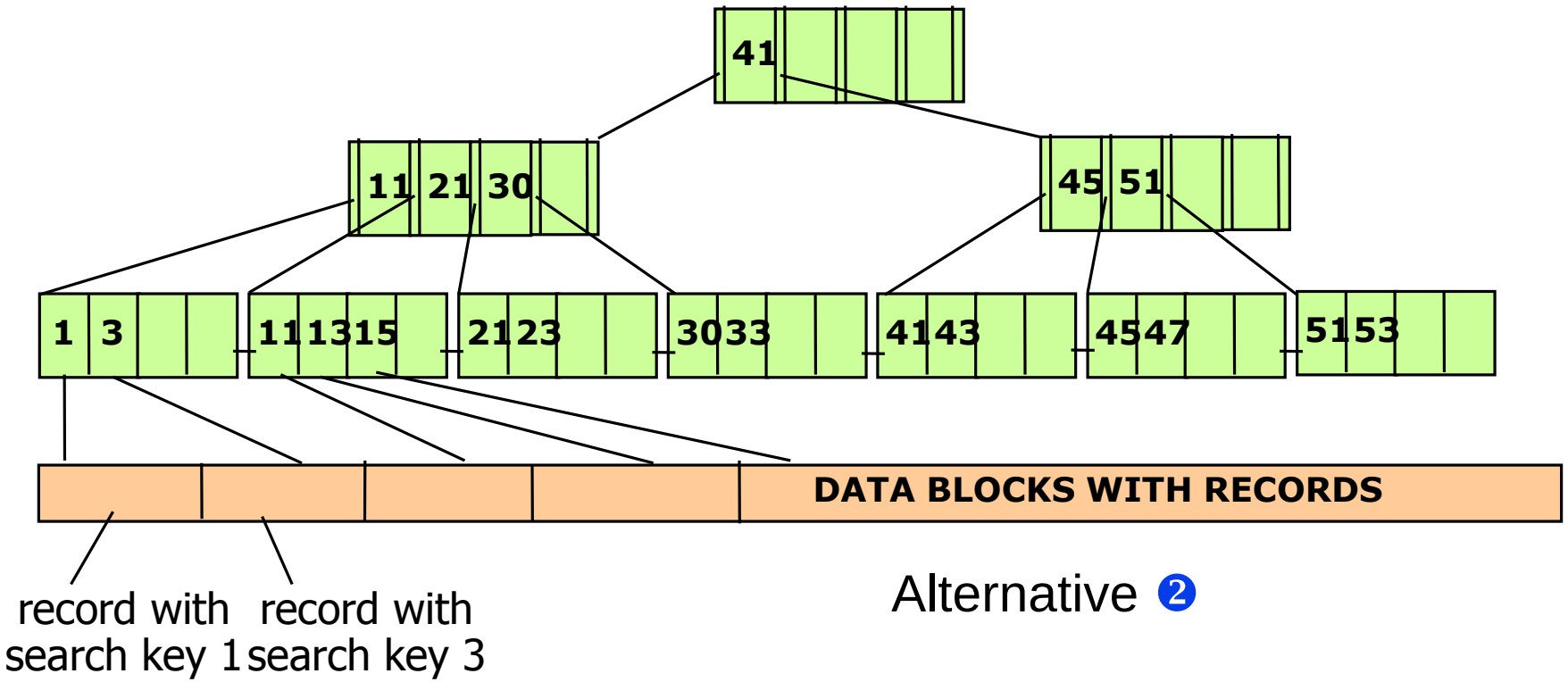
Searching B+ -Trees: Range Queries

- B+-trees are also useful for answering queries in which a range of values are sought (range queries), i.e. use a comparison operator other than = or <>
  - To find all records with key values in the range [a,b]:
    1. Search for key value a using the previous procedure; at the leaf node where a could be, search for keys >=a
    2. As long as the key values obtained are < b follow the next-block pointers to the subsequent blocks and keep examining the key values
    3. Repeat step 2 until a value >=b is found or the chain of blocks ends
  - If b=∞, all the blocks in the leaf nodes are examined
  - If a=−∞, the search starts at the leftmost leaf node

Applications of B+ -Trees

- Can serve as a dense index: there is a `(key, pointer)` in leaf nodes for every record in a data file
  - search key in B+ -Tree is the primary key of the data file
  - data file may or may not be sorted according to its primary key
- Can serve as a sparse index: there is a `(key, pointer)` in leaf nodes for every block of a data file that is sorted according to its primary key
- Can serve as a secondary index: if the file is sorted by a non-key attribute, there is a `(key, pointer)` in leaf nodes pointing to the first of records having this sort-key value
- For alternative ❶ the B+ tree represents the index as well as the data file itself (i.e., a leaf node contains the actual data records)
- For alternatives ❷ or ❸, the B+ tree lives in a file distinct from the actual data file; the p_i are (one or more) rid(s) pointing into the data file

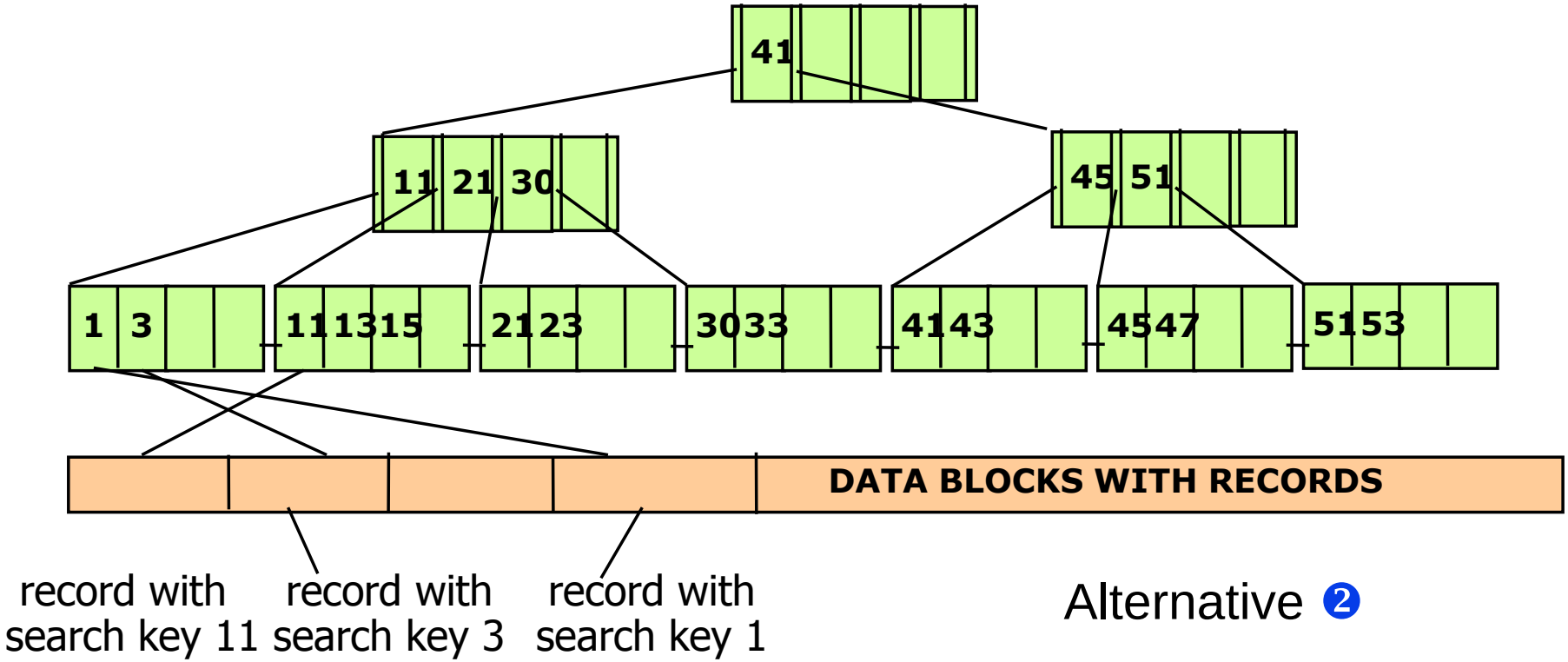
Clustered (primary) B+ -Tree (on candidate key)



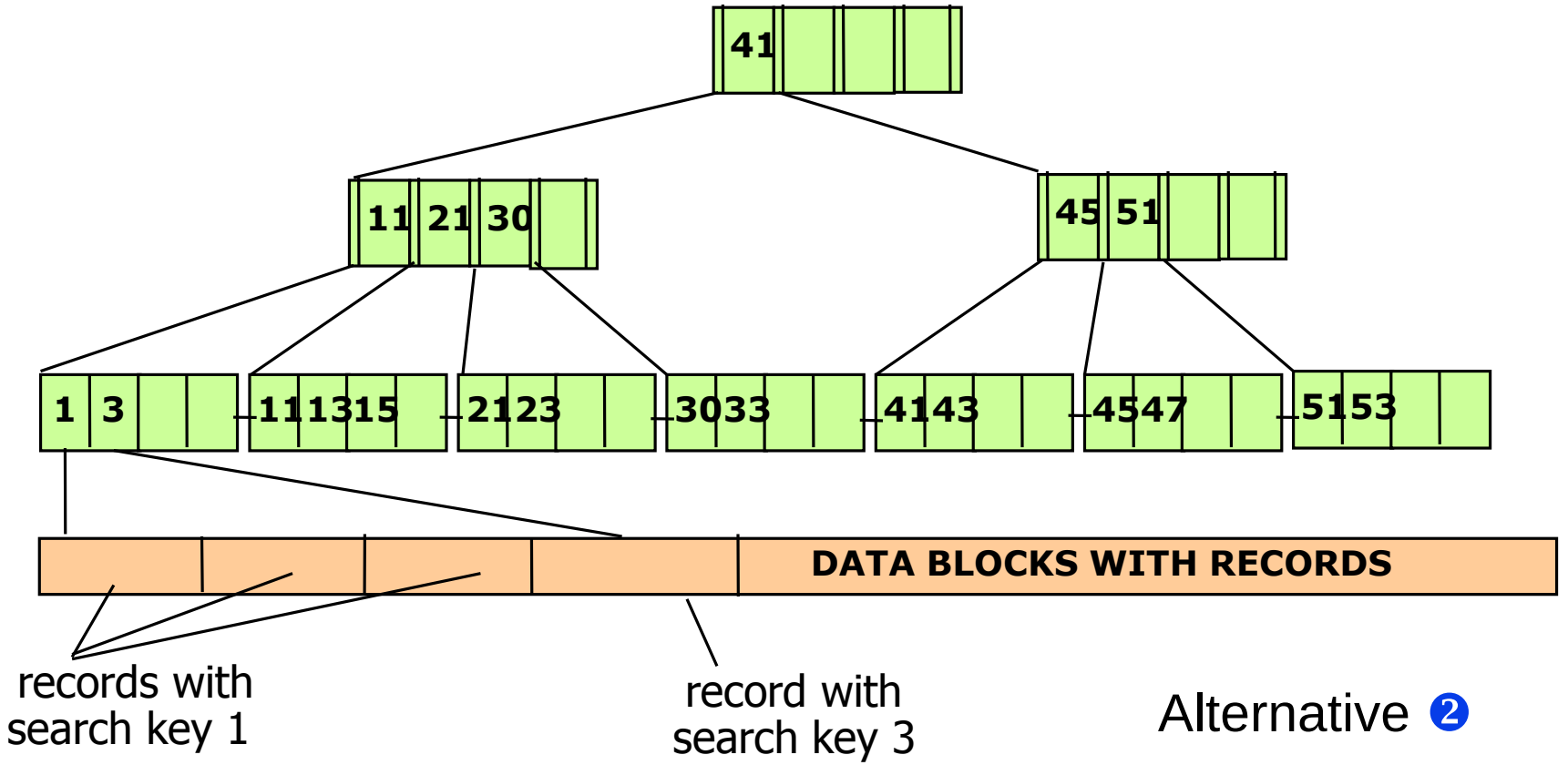
Alternative ❷

This example corresponds to dense B+-tree index:
Every search key value appears in a leaf node

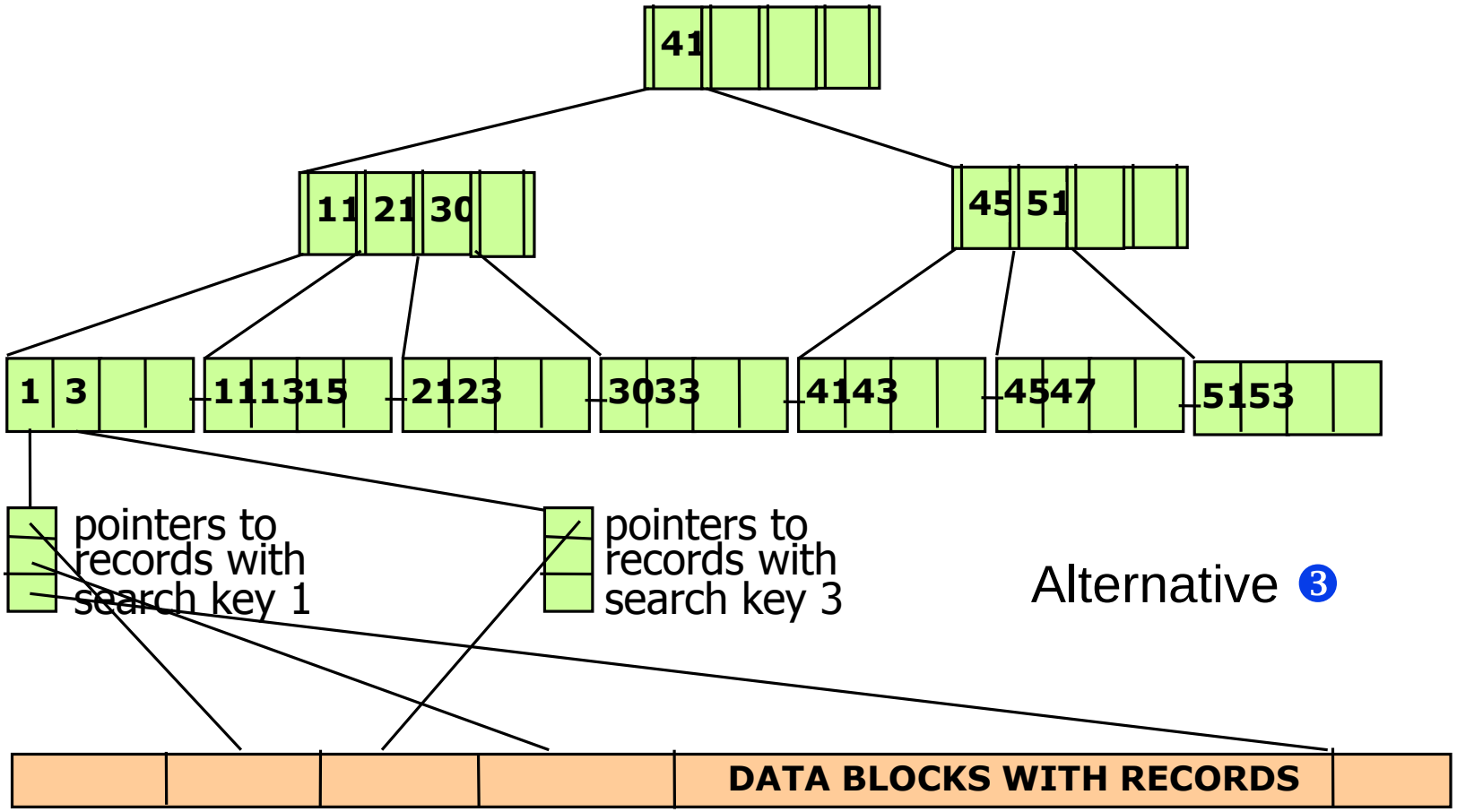
Unclustered (primary) B+ -Tree (on candidate key)



Clustered (secondary) B+ -Tree (on non-candidate key)

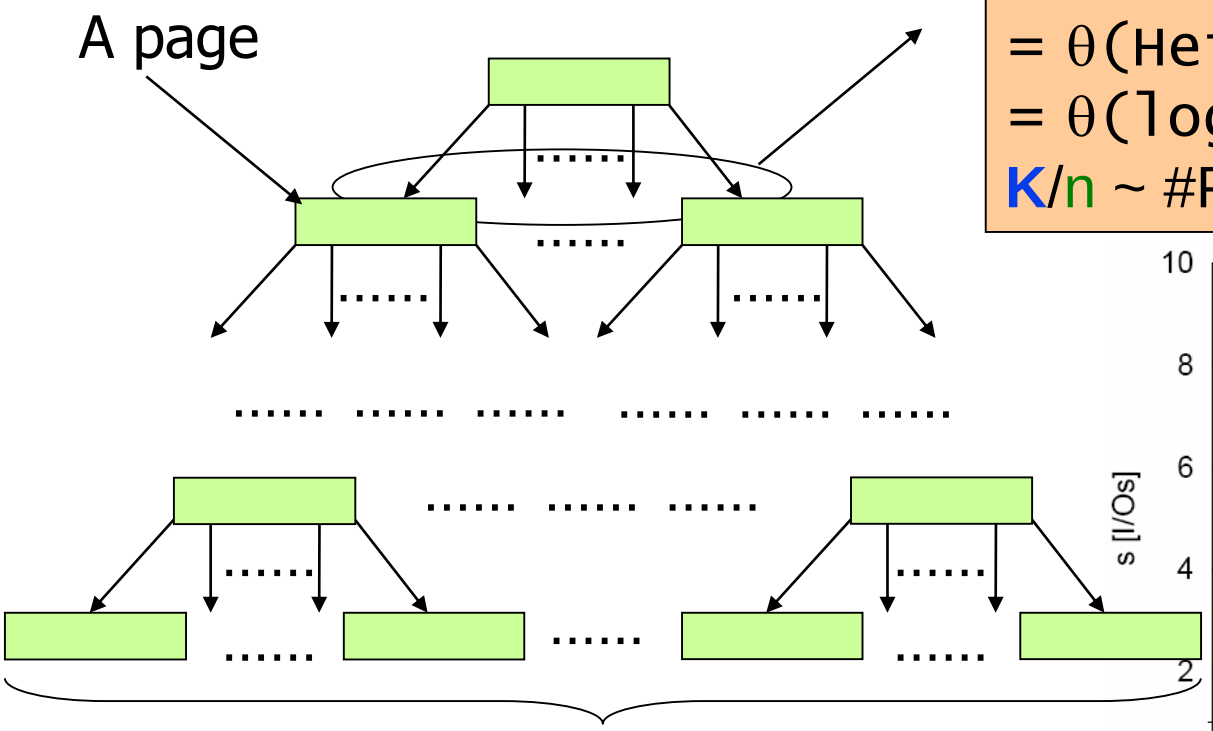


Unclustered (secondary) B+ -Tree (on non-candidate key)

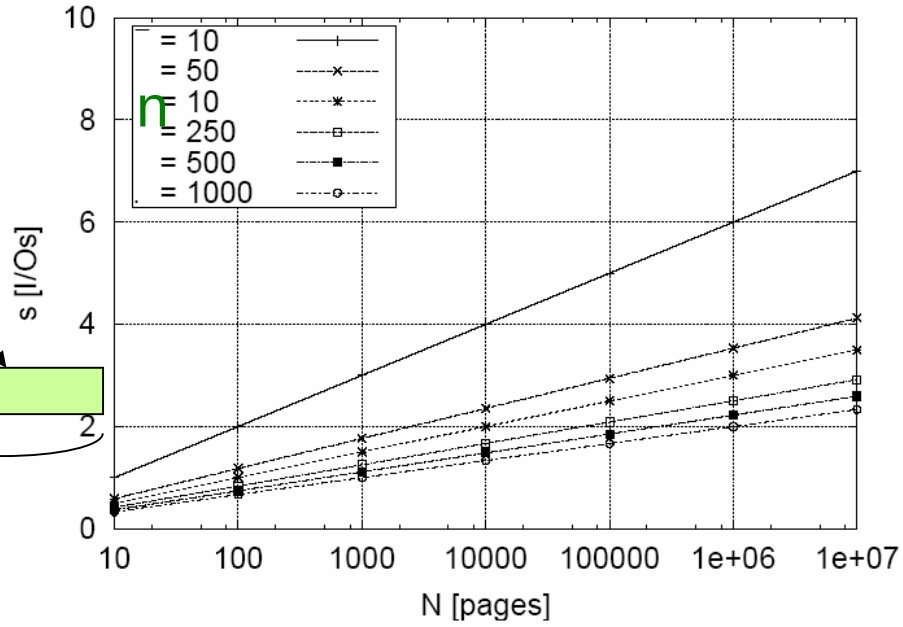


Analysis of B+ -Trees

Assume that the average number of keys per node is n and there are K search key values in the file

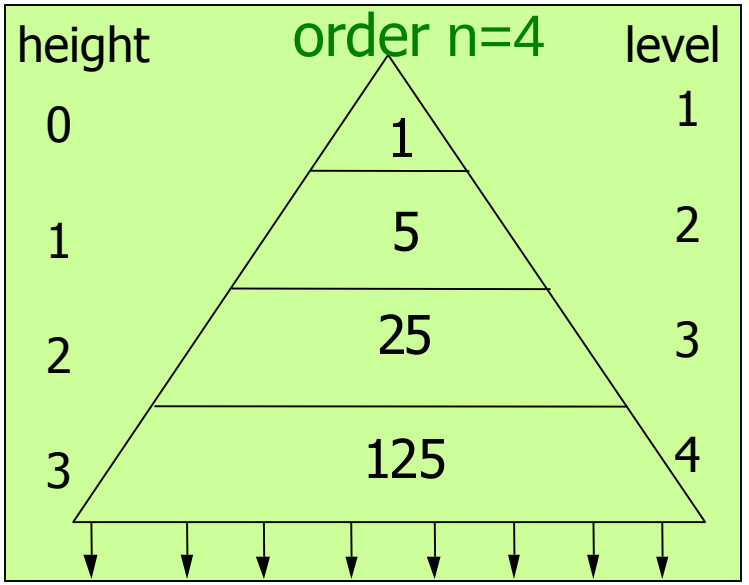


The time of searching
$= \theta(\text{Height of tree})$
$= \theta(\log_n(K/n))$
K/n ~ #Pages N



Analysis of B+ -Trees

- Each node has to perform at most n (i.e., #key) comparisons
- The number of nodes visited is limited by the level of the tree
- A tree with l levels (height+1) has
  - at most $(n+1)^{(l-1)}$ leaves
  - at least $(n+1/2)^{(l-1)}$ leaves
- Each leaf has at least
  - $n+1/2$ pointers to records if dense or
  - 1 block of $n+1$ if sparse
- If there are K search key values in the file the tree is no taller than $\lfloor \log_{\lceil n+1 / 2 \rceil} K/n+1 \rfloor$
  - Let n = 100, K = 1 million, each query will search up to only 4 nodes!
  - Note that for a balanced binary tree, it may require about 20 block access on average per query



Efficiency of B+ -Trees

- The main advantage of B+-trees is that search, insertion and deletion can be performed with few block accesses
  - number of block accesses required is the number of levels of the tree + 1 (lookup) or 2 (insert/delete)
  - in most cases 3 levels are sufficient
- Example: Assume that on average a node has 255 #keys
  - a three-level B+-tree has at most 255^2 = 65025 leaves with total of 255^3 or about 16,6 million pointers to records
  - if root block kept in main memory, each record can be accessed with 2+1 disk I/Os;
  - If all 256 internal nodes are in main memory, record access requires 1+1 disk I/Os (256 x 4 KB = 1 MB; quite feasible!)
- Is LRU a good policy for B+tree buffers? No!
  - Should try to keep root block in memory at all times; and
  - perhaps some internal node blocks from second level

B+ -Trees in Practice

- Typical order: 200
  - Typical space utilization (occupancy or fill-factor): 67%
  - Average fanout = 132 (i.e., #keys) +1 =133 pointers
- Typical capacities:
  - Level 4: 133^4= 312.900.700 pointers to records
  - Level 3: 133^3= 2.352.637 pointers to leaves
- Example: Suppose there are 1.000.000.000 data entries
  - #Levels = $\log_{133}(1.000.000.000/132) < 4$ (dense case)
  - The cost is 5 pages read
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17689 pages = 133 MBytes

B+ -Tree Optimizations

- To improve performance, we want to reduce the height
- Two strategies for decreasing the number of leaf pages
  - shorten the data stored in leaf pages using compression techniques
  - increase index (internal)node fanout by compressing key representation
- These compression techniques:
  - reduce I/O costs
  - but typically complicate insertion, deletion, and search algorithms
- Observation: Key values in inner index nodes are used only to direct traffic to the appropriate leaf page
  - During the search procedure, we need to find the smallest index i in each visited node such that $k_i <= k < k_i+1$ and then we follow link p_i
  - We do not need the key values k_i in their entirety to use them as guides
  - Rather, we could arbitrarily chose any suitable value k'_i such that k'_i separates the values to its left from those to its right. In particular, we can chose as short a value as possible
- Deciding the policy for maintaining B+tree is part of physical database design

Example: Prefix Compression

- Use prefix compression at the lower nodes
- Consider a page containing keys:

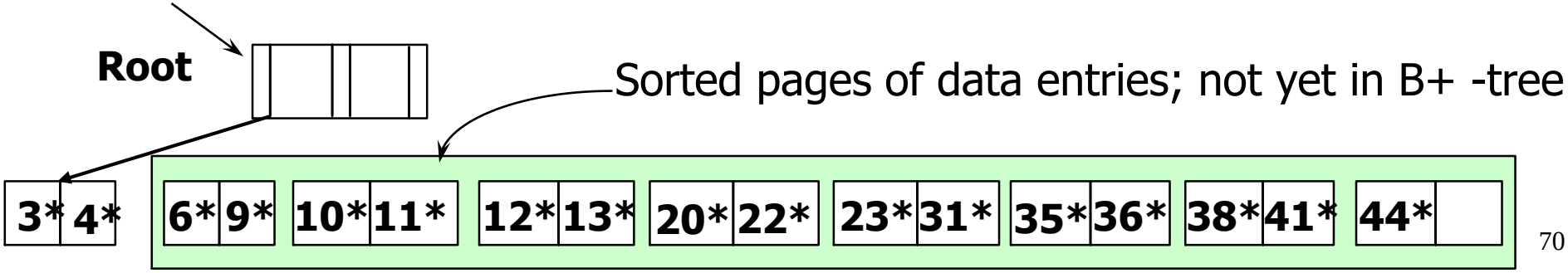
  `Manino, Manna, Mannari, Mannarino, Mannella, Mannelli`
- "Man" is a common prefix, thus we can store keys as:

  `(3 ino)  (3 na) (5 ri)  (7 no)  (4 ella)  (7 i)`
- Construct the strings:

  `(3, ino): "Man" + ino → Manino`
  `(3 na): "Man" + na → Manna`
  `(5 ri): "Manna" + ri → Mannari`
  `→ e.g., pick first 5 characters of previous string`
  `(7 no): "Mannari" + no → Mannarino`
  `(4 ella): "Mann" + ella → Mannella`
  `(7 i): "Mannell" + I → Mannelli`
- In general, while compressing, must leave each index entry greater than every key value (in any subtree) to its left

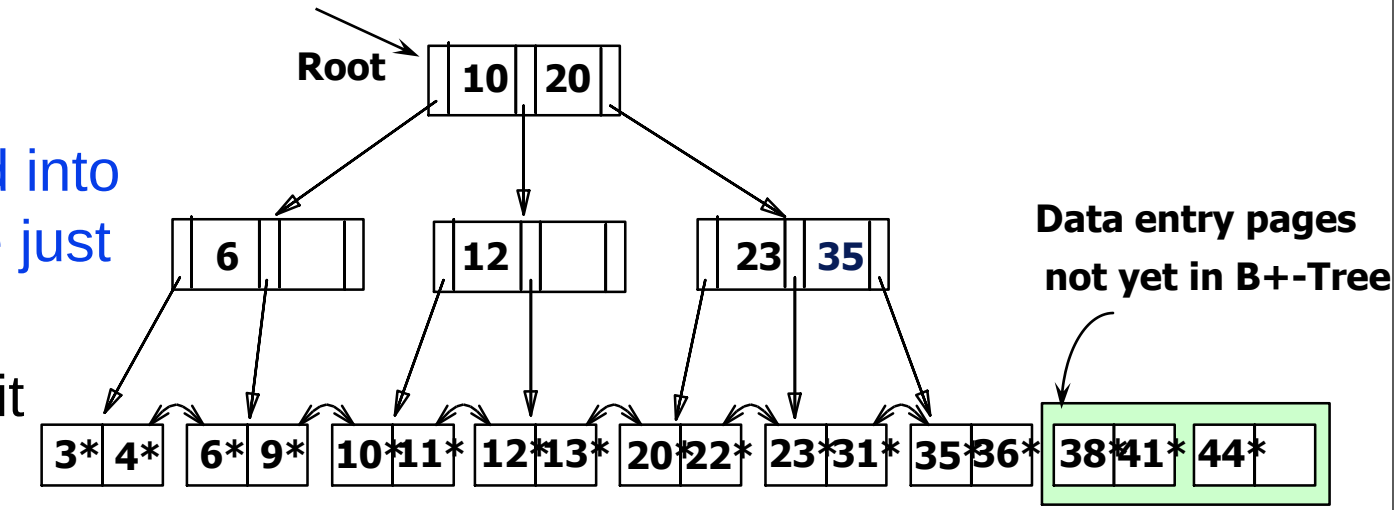
Bulk Loading of a B+ -Tree

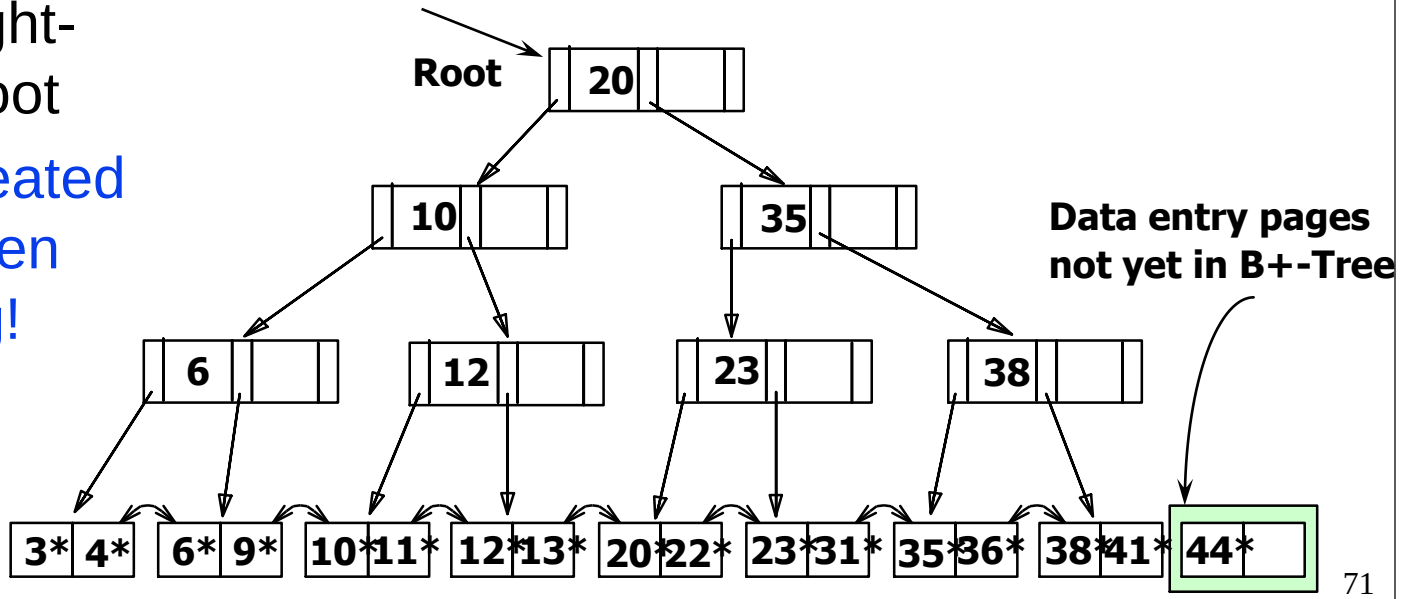
- If we have a large collection of records, and we want to create a B+ tree on some field, doing so by repeatedly inserting records
  - is very slow
  - leads to minimal leaf utilization
  - does not give sequential storage of leaves (aka clustering)
- Bulk Loading can be done much more efficiently: Sort all data entries, insert pointer to first (leaf) page in a new (root) page
  - Fewer I/Os for building (i.e., buffer pool is utilized more effectively)
  - Fewer locks for concurrency control (tree traversals are saved)
  - Leaves will be stored sequentially (and linked, of course)
  - Better control of "fill factor" on pages (leaf nodes filled up completely)



Bulk Loading

- Index entries for leaf pages always entered into right-most index page just above leaf level
  - When this fills up, it splits
  - Split may go up right-most path to the root
- Much faster than repeated inserts, especially when one considers locking!
  - Time and Space efficient !





Τέλος Ενότητας

Χρηματοδότηση

•Το παρόν εκπαιδευτικό υλικό έχει αναπτυχθεί στα πλαίσια του εκπαιδευτικού έργου του διδάσκοντα.

•Το έργο «**Ανοικτά Ακαδημαϊκά Μαθήματα στο Πανεπιστήμιο Κρήτης**» έχει χρηματοδοτήσει μόνο τη αναδιαμόρφωση του εκπαιδευτικού υλικού.

•Το έργο υλοποιείται στο πλαίσιο του Επιχειρησιακού Προγράμματος «Εκπαίδευση και Δια Βίου Μάθηση» και συγχρηματοδοτείται από την Ευρωπαϊκή Ένωση (Ευρωπαϊκό Κοινωνικό Ταμείο) και από εθνικούς πόρους.

Ευρωπαϊκή Ένωση
Ευρωπαϊκό Κοινωνικό Ταμείο

ΕΠΙΧΕΙΡΗΣΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΕΚΠΑΙΔΕΥΣΗ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗ
επένδυση στην κοινωνία της γνώσης
ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ ΚΑΙ ΘΡΗΣΚΕΥΜΑΤΩΝ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ

ΕΣΠΑ 2007-2013
πρόγραμμα για την ανάπτυξη
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ

Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης

Σημειώματα

Σημείωμα αδειοδότησης

•Το παρόν υλικό διατίθεται με τους όρους της άδειας χρήσης Creative Commons Αναφορά, Μη Εμπορική Χρήση, Όχι Παράγωγο Έργο 4.0 [1] ή μεταγενέστερη, Διεθνής Έκδοση. Εξαιρούνται τα αυτοτελή έργα τρίτων π.χ. φωτογραφίες, διαγράμματα κ.λ.π., τα οποία εμπεριέχονται σε αυτό και τα οποία αναφέρονται μαζί με τους όρους χρήσης τους στο «Σημείωμα Χρήσης Έργων Τρίτων».

[1] http://creativecommons.org/licenses/by-nc-nd/4.0/

•Ως **Μη Εμπορική** ορίζεται η χρήση:

–που δεν περιλαμβάνει άμεσο ή έμμεσο οικονομικό όφελος από την χρήση του έργου, για το διανομέα του έργου και αδειοδόχο

–που δεν περιλαμβάνει οικονομική συναλλαγή ως προϋπόθεση για τη χρήση ή πρόσβαση στο έργο

–που δεν προσπορίζει στο διανομέα του έργου και αδειοδόχο έμμεσο οικονομικό όφελος (π.χ. διαφημίσεις) από την προβολή του έργου σε διαδικτυακό τόπο

•Ο δικαιούχος μπορεί να παρέχει στον αδειοδόχο ξεχωριστή άδεια να χρησιμοποιεί το έργο για εμπορική χρήση, εφόσον αυτό του ζητηθεί.

.

Σημείωμα Αναφοράς