Database 2

Lecture II

Alessandro Artale

Faculty of Computer Science – Free University of Bolzano

Room: 221

artale@inf.unibz.it

http://www.inf.unibz.it/∼artale/

2003/2004 – First Semester

Summary of Lecture II

- **I**ndexing.
  - Indexes on Sequential Files: Dense Vs. Sparse Indexes.
  - Primary Indexes with Duplicate Keys.
  - Secondary Indexes.
  - Document Indexing.
  - B-Tree Indexes.

Indexing

Indexing is the principal technique used to efficiently answering a given query.

- An Index for a DB is like an Index in a book:
  1. It is smaller that the book;
  2. The words are in sorted order;
  3. If we are looking for a particular topic we first search on the index, find the pages where it is discussed, go to the actual pages in the book.

Example.

`MovieStar(Name,Address,Gender,Birthdate)`

SELECT *
FROM MovieStar
WHERE Name = 'Jim Carrey';

All the blocks for the `MovieStar` relation should be inspected if there is no index on Name.

Index

An *Index* is a data structure that facilitates the query answering process by minimizing the number of disk accesses.

- An index structure is usually defined on a single Attribute of a Relation, called the **S**earch Key;
- An Index takes as input a Search Key value and returns the address of the record(s) (block physical address + offset of the record) holding that value.
- Index structure: Search Key-Pointer pairs

Search Key	Pointer to a data-file record

- The Searck Key values stored in the Index are *Sorted* and a binary search can be done on the Index.
- Only a small part of the records of a relation have to be inspected: Appropriate indexes can speed up query processing passing from minutes to seconds.

Index Structures

Different data structures give rise to different indexes:

1. **Indexes on Sequential Files (Primary Index);**
2. **Secondary Indexes on Unsorted Files;**
3. **B-Trees;**
4. **Hash Tables.**

Evaluating Different Index Structures

No one technique is the best. Each has to evaluated w.r.t. the following criteria:

- **Access Type.** Finding records either with a particular search key, or with the search key falling in a given range.
- **Access Time.** The time it takes to find item(s) using the index in question.
- **Insertion Time.** The time to insert an item in the data file, as well as the time to update the index.
- **Deletion Time.** The time to delete the item from the data file (which include the time to find the item), and the time to update the index.
- **Space Overhead.** Additional space for the index.

Summary

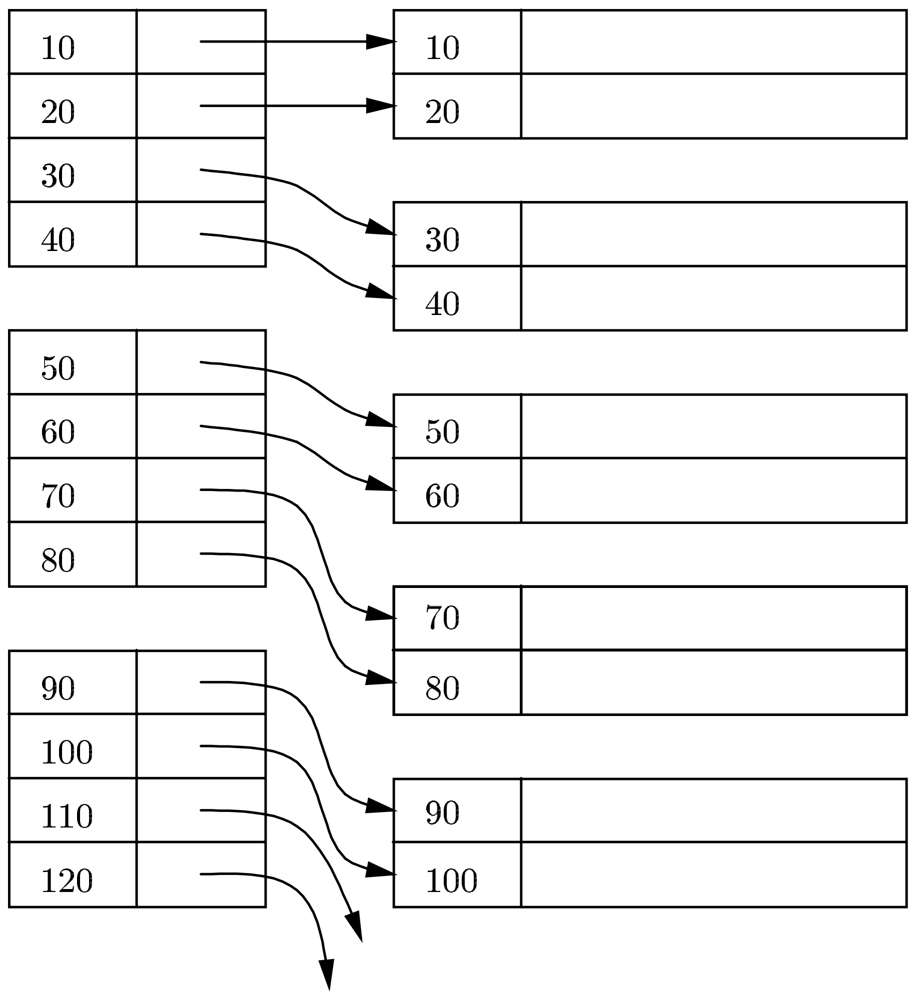
- Indexing
  - **I**ndexes on Sequential Files: Dense Vs. Sparse Indexes.
  - Primary Indexes with Duplicate Keys.
  - Secondary Indexes.
  - Document Indexing.
  - B-Tree Indexes.

Indexes on Sequential Files

- **I**ndex on Sequential File, also called **P**rimary Index, when the Index is associated to a *Data File* which is in turn *sorted with respect to the search key*.
  1. A Primary Index forces a sequential file organization on the Data File;
  2. Since a Data File can have just one order there can be just one Primary Index for Data File.
- Usually used when the search key is also the primary key of the relation.
- Usually, these indexes fit in main memory.
- Indexes on sequential files can be:
  1. **D**ense: One entry in the index file for every record in the data file;
  2. **S**parse: One entry in the index file for each block of the data file.

Dense Indexes

Every value of the search key has a representative in a **Dense Index**. The index maintains the keys in the same order as in the data file.



Database System Implementation, H. Garcia-Molina, J. Ullman, and J. Widom, Prentice-Hall, 2000.

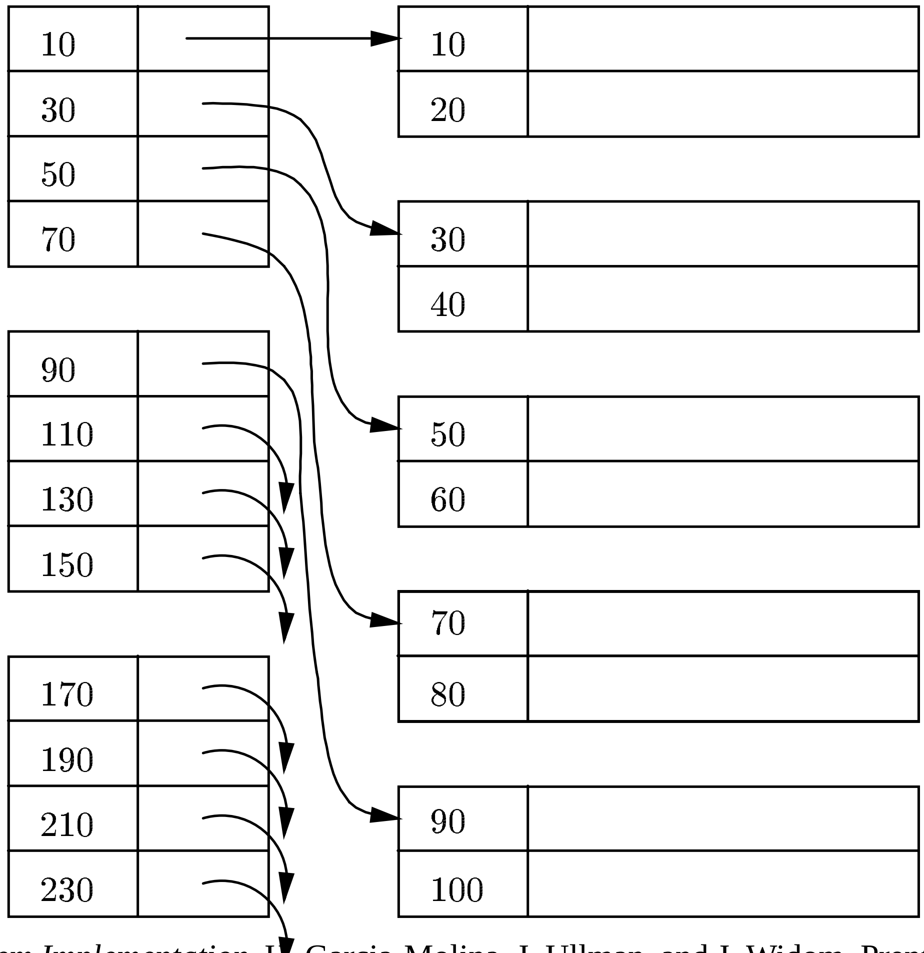
Queries with Dense Indexes

Algorithm for **Lookup**: Searching a data record with a given search key value.

- Given a search key K, the index is scanned and when K is found the associated pointer to the data file record is followed and the record (block containing it) is read in main memory.
- Dense indexes support also *range queries*: The minimum value is located first, if needed, consecutive blocks are loaded in main memory until a search key greater than the maximum value is found.
- Query-answering using dense indexes is *efficient*:
  1. Since the index is usually kept in main memory, just 1 disk I/O has to be performed during lookup;
  2. Since the index is sorted we can use binary search: If there are n search keys then at most $log_2 n$ steps are required to locate a given search key.

Sparse Indexes

- Used when dense indexes are too large: A Sparse Index uses less space at the expense of more time to find a record given a key.
- A sparse index holds **one key-pointer pair per data block**, usually the first record on the data block.



Database System Implementation, H. Garcia-Molina, J. Ullman, and J. Widom, Prentice-Hall, 2000.

Queries with Sparse Indexes

Algorithm for Lookup.

- Given a search key K:
  1. Search the sparse index for the greatest key $\leq$ to K using binary search;
  2. We retrieve the pointed block to main memory to look for the record with search key K (always using binary search).
- With respect to dense indexes we need to start two different binary searches: the first on the sparse index, and the second on the retrieved data block.
- Still 1 disk I/O for lookup.
- In conclusion, a Sparse Index is more efficient in space at the cost of a worst computing time in Main Memory.

Primary Dense Index: Example

Example of a **Primary Dense Index** with **Search Key**=`Account#`.

A-101	→
A-102	→
A-110	→
A-201	→
A-215	→
A-217	→
A-218	→
A-222	→
A-305	→

Account♯	Branch	Balance
A-101	Downtown	500
A-102	Perryridge	400
A-110	Downtown	600
A-201	Perryridge	900
A-215	Mianus	700
A-217	Brighton	750
A-218	Perryridge	700
A-222	Redwood	700
A-305	Round Hill	350

Primary Sparse Index: Example

Example of a **Primary Sparse Index** with **Search Key**=`Account#`.

A-101	
A-201	
A-218	

Account♯	Branch	Balance
A-101	Downtown	500
A-102	Perryridge	400
A-110	Downtown	600
A-201	Perryridge	900
A-215	Mianus	700
A-217	Brighton	750
A-218	Perryridge	700
A-222	Redwood	700
A-305	Round Hill	350

Summary

- Indexing
  - Indexes on Sequential Files: Dense Vs. Sparse Indexes.
  - **P**rimary Indexes with Duplicate Keys.
  - Secondary Indexes.
  - Document Indexing.
  - B-Tree Indexes.

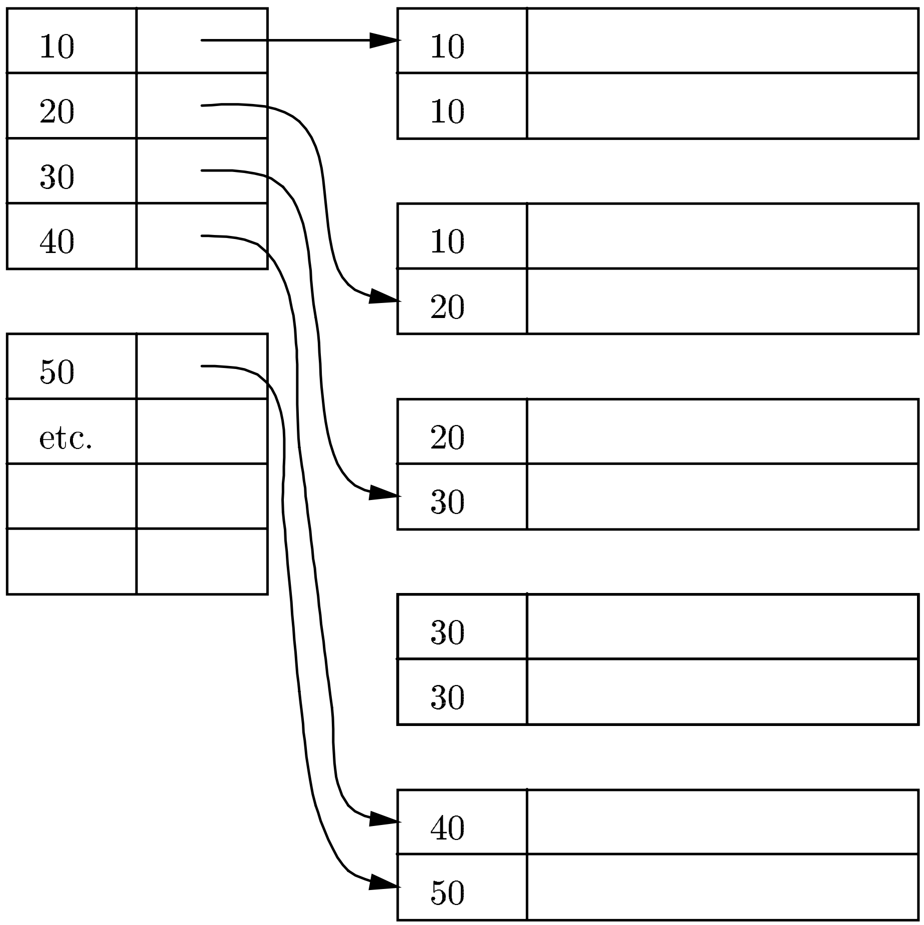
Primary Indexes with Duplicate Keys

- Indexes for **non key attributes**:
  More than one record with the same search key.
- As usual, the data file should be sorted w.r.t the search key to speak of primary indexes.
- Techniques for dense indexes:
  1. One entry for each record in the data file: Duplicate key-pointer pairs (not used);
  2. Just a single entry for each record in the data file with search key K – no duplicate key-pointer pairs: Pointer to the first record with search key K (more efficient).

Dense Index with Duplicate Search Keys

Single Entry Index

Lookup. Find the search key on the index, read the pointed disk block, possibly read successive blocks.



Database System Implementation, H. Garcia-Molina, J. Ullman, and J. Widom, Prentice-Hall, 2000.

Primary Dense Index with Duplicates: Example

Example of a **Primary Dense Index with Duplicates** with **Search Key**=`Branch`.

Brighton	
Downtown	
Mianus	
Perryridge	
Redwood	
Round Hill	

`Account♯`	`Branch`	`Balance`
A-217	Brighton	750
A-101	Downtown	500
A-110	Downtown	600
A-215	Mianus	700
A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700
A-222	Redwood	700
A-305	Round Hill	350

Analysis of Primary indexes

- **Advantages.**
  Efficient access of tuples with a given search key.
  - Very few blocks should be read (also in case of duplicate keys);
  - **R**ange Queries – looking for search key values in a certain range – are answered efficiently.

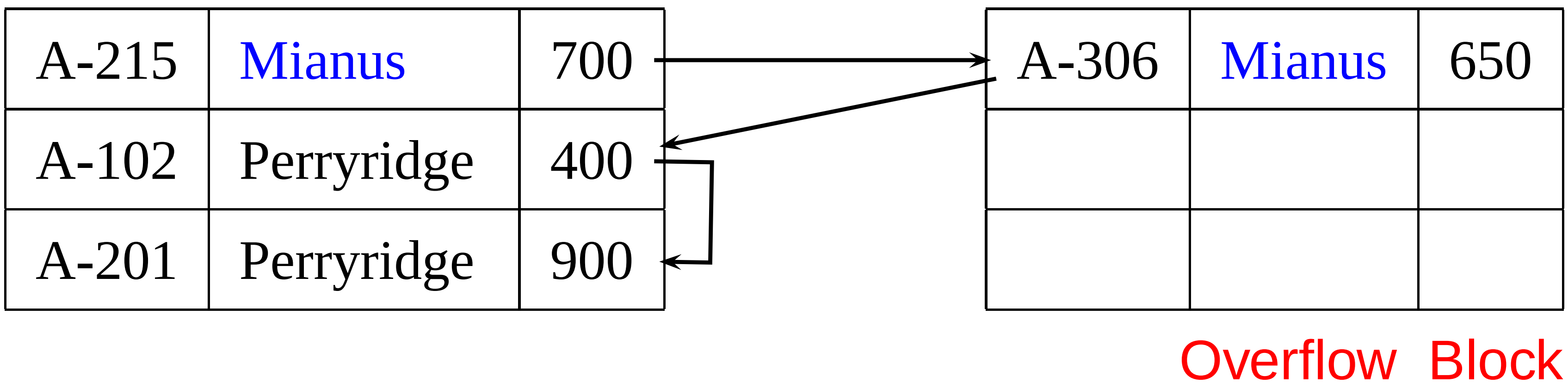
Analysis of Primary indexes (cont.)

- **Disadvantages.**

  Expensive maintenance of the physical records storage to maintain the sorted order.

  - Technique used for insertion based on **O**verflow Blocks.
    1. If there is space in the block insert the new record there in the right place;
    2. Otherwise, insert the new record in an *Overflow Blocks*. In order to maintain the order, records are linked by means of pointers: The pointer in each record points to the next record in search-key order.
  - In general, performance degrades as far as the relation grows. The file is *reorganized* when the system load is low.
  - An optimal solution is to implement primary indexes as *B-Tree* structures (presented soon).

Insertion in Sequential Files: Example



Summary

- Indexing
  - Indexes on Sequential Files: Dense Vs. Sparse Indexes.
  - Primary Indexes with Duplicate Keys.
  - Secondary Indexes.
  - Document Indexing.
  - B-Tree Indexes.

Secondary Indexes

- A *primary index* is an index on a file sorted w.r.t. the search key. Then a primary index *"controls"* the storage of records in the data file.
- Indexes on Sequential files and Hash Tables are examples of primary indexes.
- Since a file can have at most one physical order then it can have at most one primary index.
- Secondary Indexes facilitate query-answering on attributes other than primary keys – or, more generally, on *non-ordering* attributes.
- A file can have *several* secondary indexes.

Secondary indexes do not determine the placement of records in the data file.

Secondary Index: An Example

Let us consider the `MovieStar` relation:

`MovieStar(`<u>`Name`</u>`,Address,Gender,Birthdate)`

and a query involving the non-key `Birthdate` attribute:

SELECT Name, Address
FROM MovieStar
WHERE Birthdate = '1975-01-01';

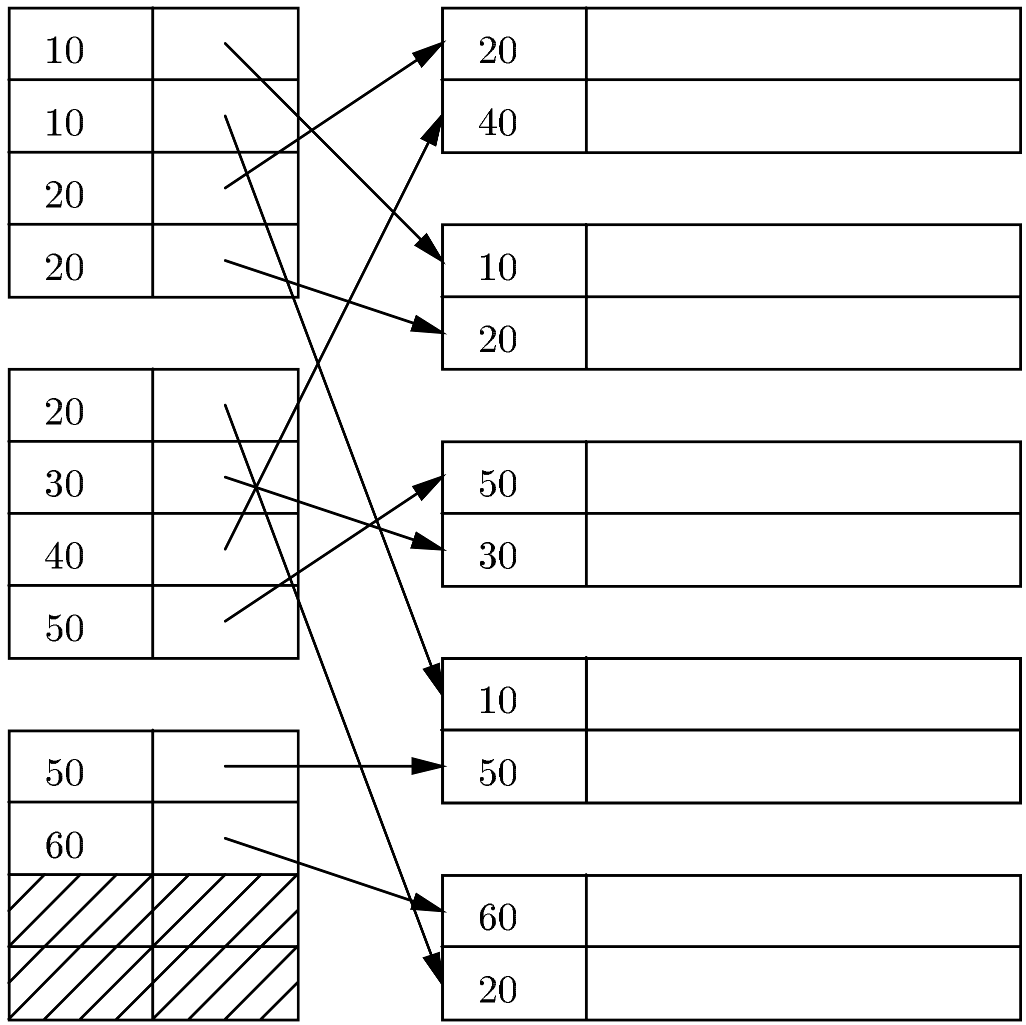
A secondary index on the `MovieStar` relation w.r.t. the `Birthdate` attribute would reduce the answering time.

Structure of Secondary Indexes

- Secondary Indexes are always Dense:
  Sparse secondary indexes make no sense!
- Secondary indexes are sorted w.r.t. the search key $\rightarrow$ Binary search.
- The Data File **IS NOT** sorted w.r.t. the Secondary Index Search Key!
- More than one data block may be needed for a given search key $\rightarrow$ in general more disk I/O to answer queries:
  - Secondary Indexes are less efficient than Primary Indexes.

Secondary Indexes: An Example

The example shows that 3 data blocks (i.e., 3 disk I/O) are needed to retrieve all the tuples with search key $K = 20$ using the Secondary Index.

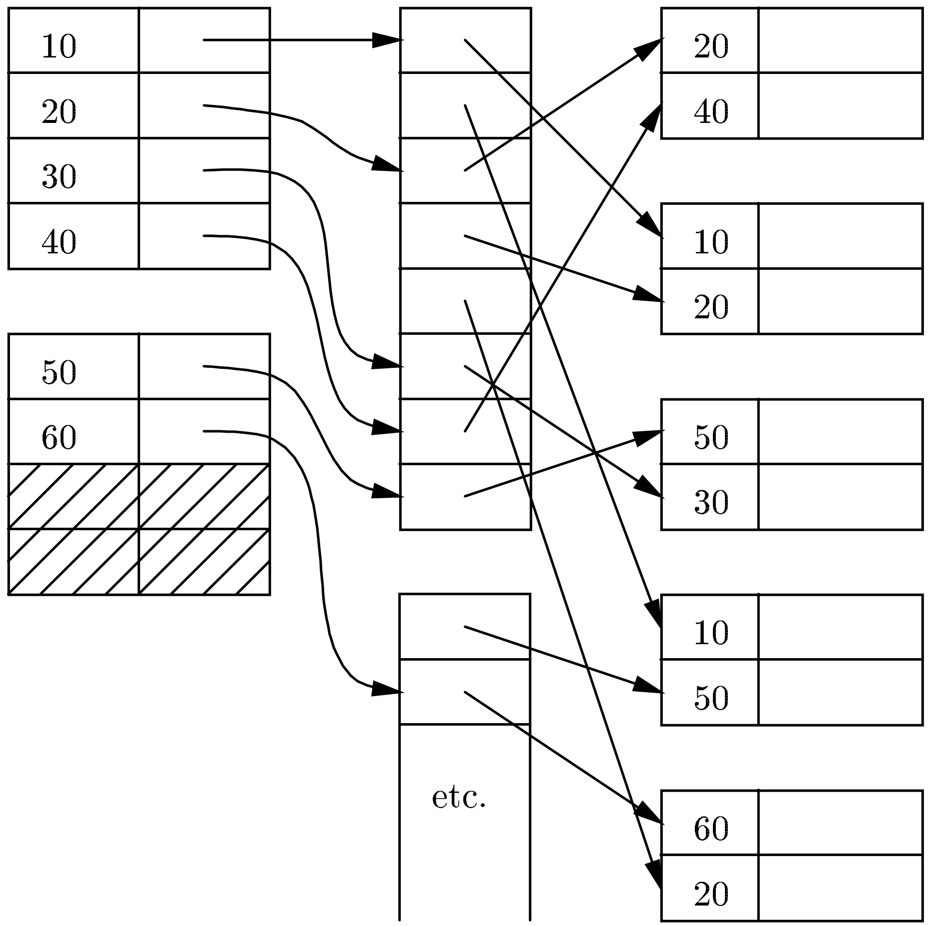


Database System Implementation, H. Garcia-Molina, J. Ullman, and J. Widom, Prentice-Hall, 2000.

Indirect Buckets

- To avoid repeating keys in secondary index, use a level of indirection, called **Buckets**.

- The index maintains only one key-pointer pair for each search key K: The pointer for K goes to a position in the bucket which contains pointers to records with search key K till the next position pointed by the index.



Database System Implementation, H. Garcia-Molina, J. Ullman, and J. Widom, Prentice-Hall, 2000.

Summary

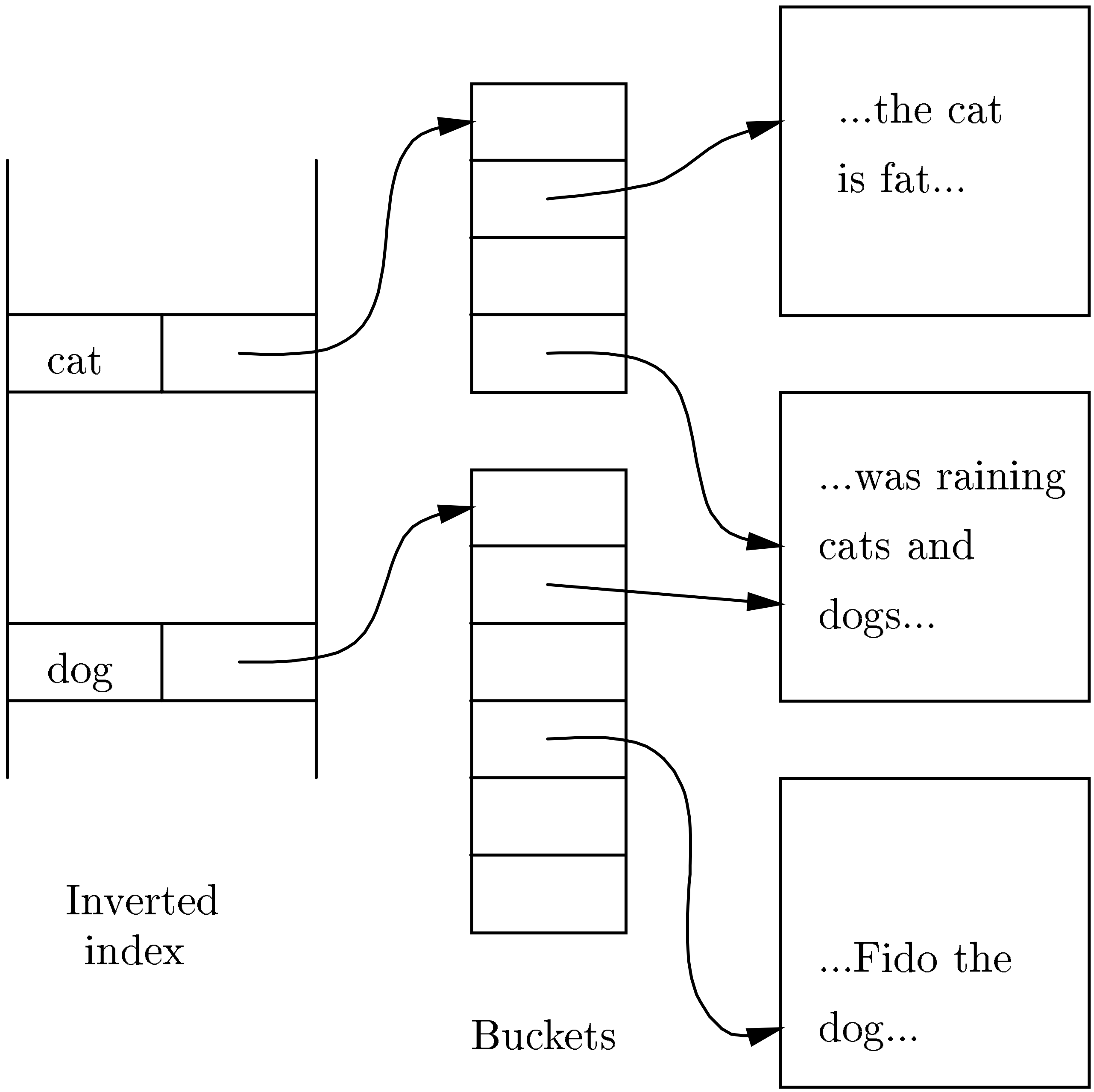
- Indexing
  - Indexes on Sequential Files: Dense Vs. Sparse Indexes.
  - Primary Indexes with Duplicate Keys.
  - Secondary Indexes.
  - **D**ocument Indexing.
  - B-Tree Indexes.

Document Retrieval and Inverted Indexes

Problem: Given a set of text documents we need to retrieve that ones where a particular word(s) occurs.

- Given the success of the Web this has become an urgent database problem.
- A document is thought as a tuple in the relation `Doc(ID,cat,dog,...)`, with one attribute for each possible word.
- Each attribute has a Boolean value, eg, the value of `cat` is `TRUE` if and only if the word *cat* appears in the document.
- An **I**nverted Index is a form of secondary index with indirect bucket containing – as search keys – all the attribute names of the `Doc` relation.
- Pointers are stored in a bucket file and consider only the `TRUE` occurrences of a search key.

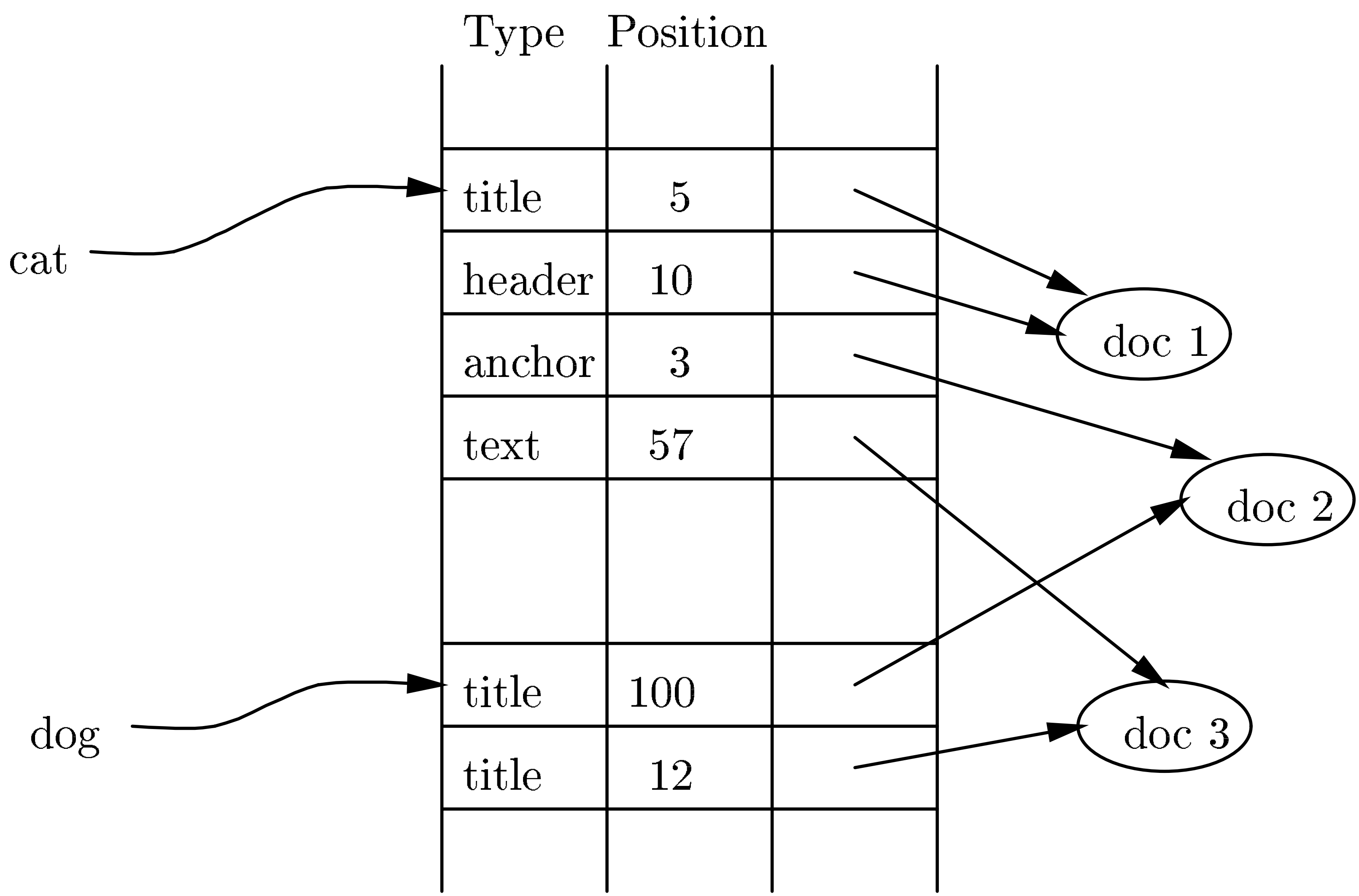
Inverted Indexes: An Example



Database System Implementation, H. Garcia-Molina, J. Ullman, and J. Widom, Prentice-Hall, 2000.

Additional Information in Buckets

Buckets can be extended to include “Type” (e.g., specify whether the word appears in the title, abstract or body), “Position” of word, etc.



Database System Implementation, H. Garcia-Molina, J. Ullman, and J. Widom, Prentice-Hall, 2000.

Summary

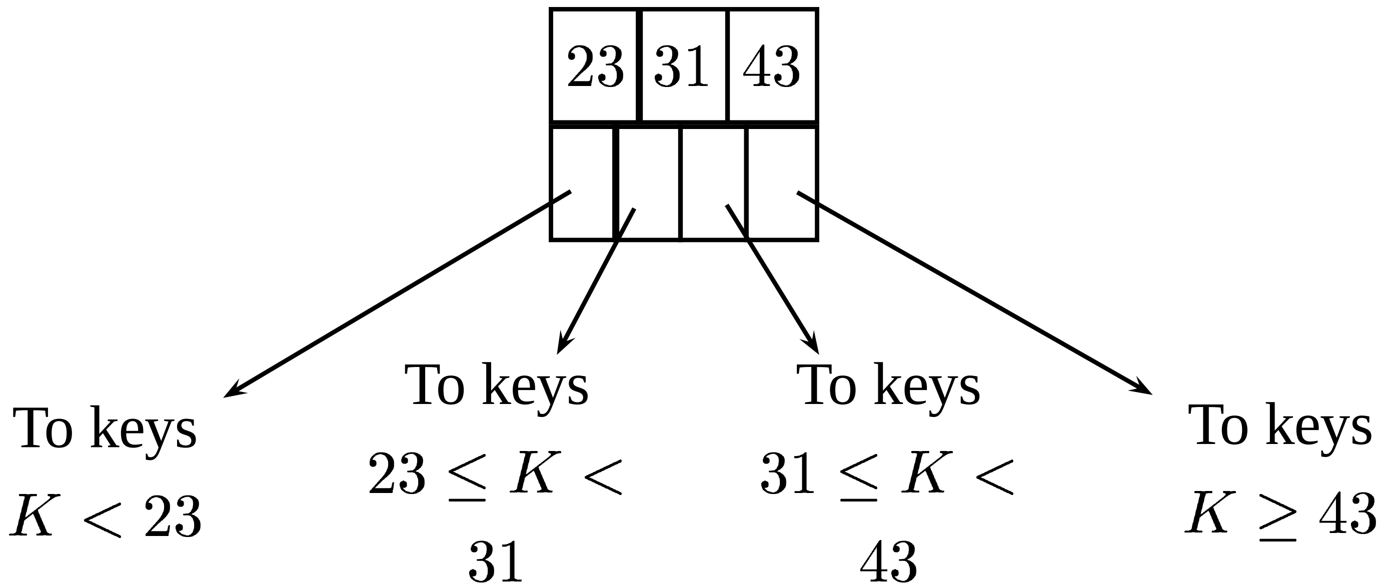
- Indexing
  - Indexes on Sequential Files: Dense Vs. Sparse Indexes.
  - Primary Indexes with Duplicate Keys.
  - Secondary Indexes.
  - Document Indexing.
  - **B-**Tree Indexes.

B-Trees

- A B-Tree is a multilevel index with a *Tree* structure;
- When used as primary index (i.e., on a sorted file) maintains efficiency against insertion and deletion of records avoiding file reorganization (the main disadvantage of index on sequential file);
- Also used to index very-large relations when single-level indexes don't fit in main memory;
- Commercial systems (DB2, ORACLE) implement indexes with B-Trees;
- In the following we will present the structure of so called B^+-*Tree* – the **B** stands for **Balanced Tree**.

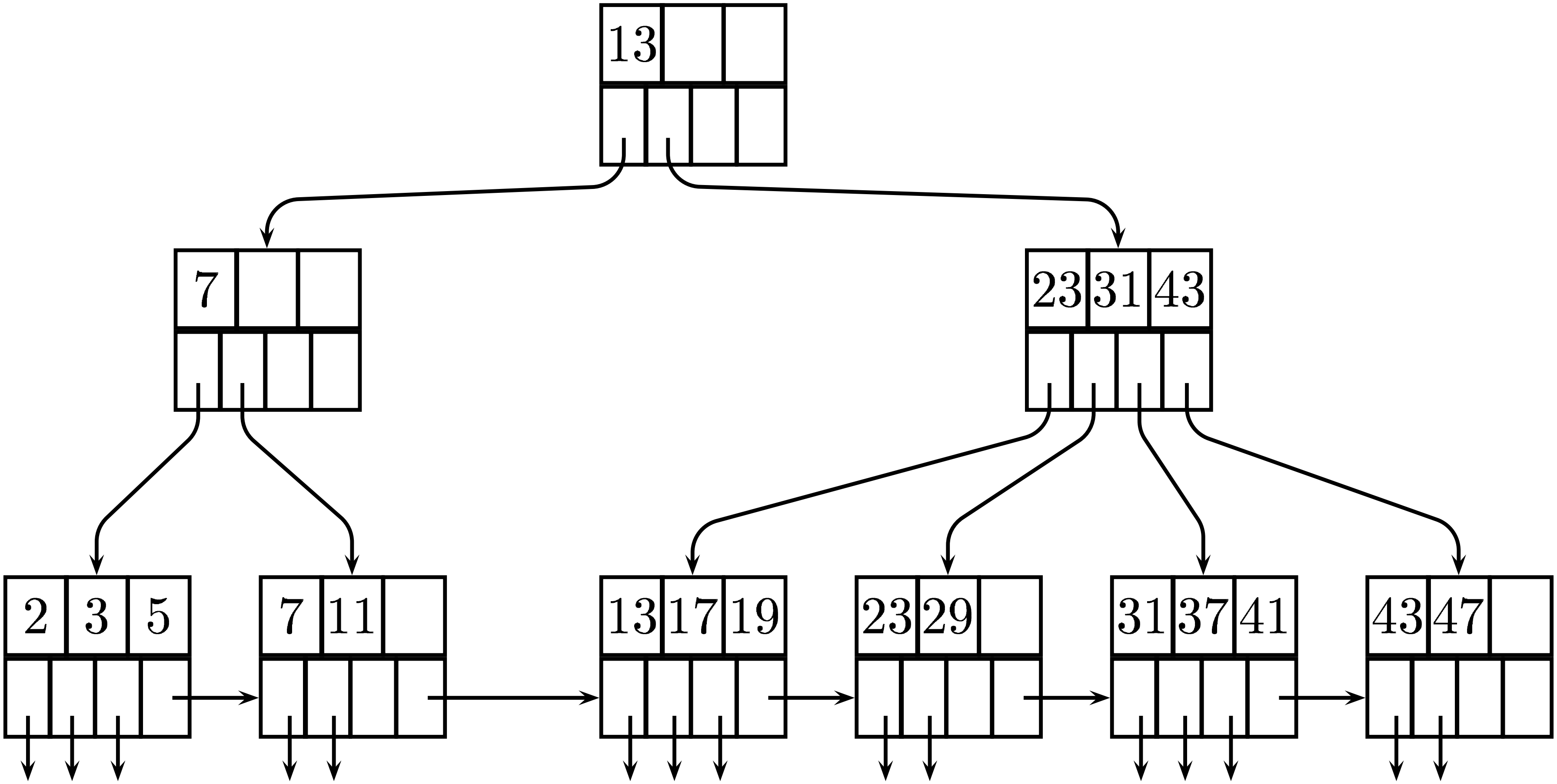
B-Trees (cont.)

- B-tree is usually a 3 levels tree: the root, an intermediate level, the leaves.
- All the leaves are at the same level $\rightarrow$ *Balanced Tree*.
- The size of each node of the B-tree is equal to a disk block. All nodes have the same format: **n keys** and $\mathbf{n}+1$ **pointers** $\rightarrow$ n key-pointer pairs plus 1 extra pointer.



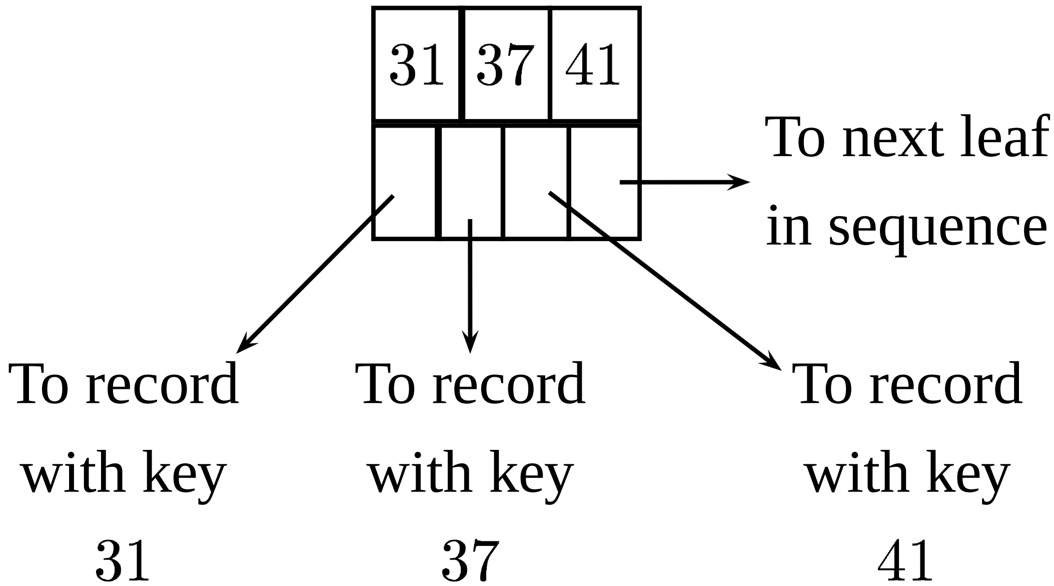
Example. Let a block be 4096 bytes, a search key be an integer of 4 bytes, and a pointer be 8 bytes. If there is no additional header in the block then n is the largest integer s.t. $4n + 8(n+1) \leq 4096 \;\; \rightarrow n = 340$.

B-Tree: An Example



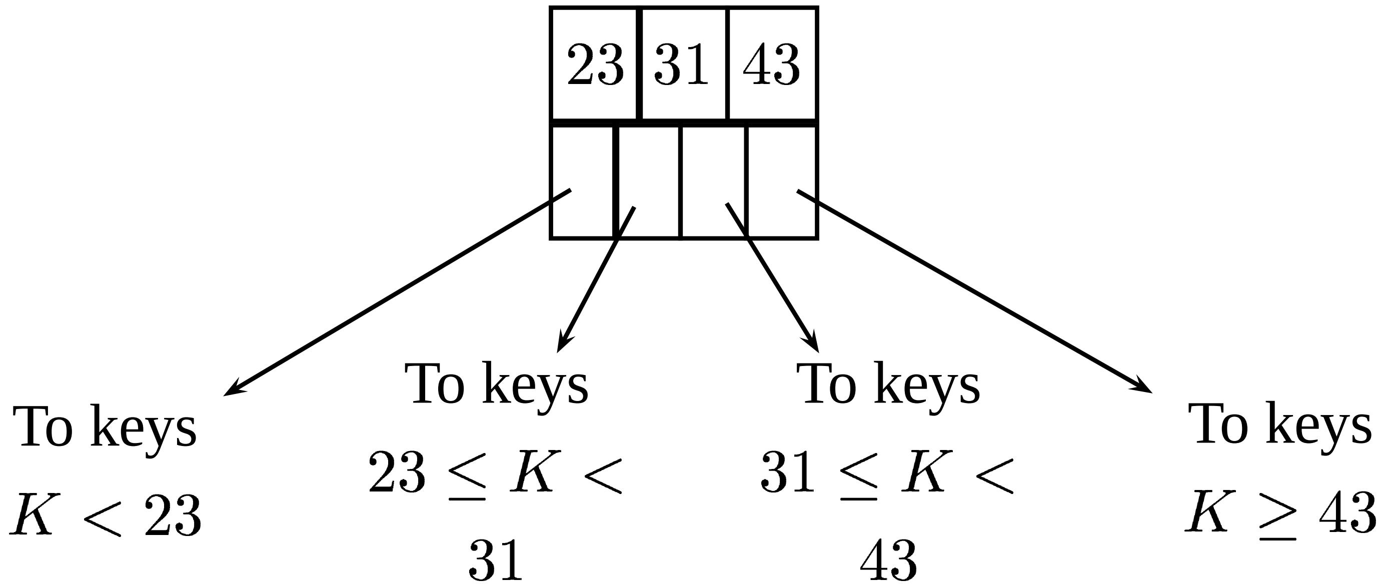
- Data file where search-keys are all the prime numbers from 2 to 47.
- All the keys appear once (in case of a dense index), and in sorted order at the leaves.
- A pointer points to either a file record (in case of a primary index structure) or to a bucket of pointers (in case of a secondary index structure).

Leaves



- One pointer to next leaf—used to chain leaf nodes in search-key order for efficient sequential processing;
- At least $\left\lfloor \frac{n+1}{2} \right\rfloor$ (round down) key-pointer pairs pointing to either records of the data file (as shown in the Figure) or to a bucket of pointers.

Interior Nodes



- All $n + 1$ pointers can be used to point to B-tree nodes of the inferior level;
- At least $\lceil \frac{n+1}{2} \rceil$ (round up) pointers must be used, and one more pointer than key is used.
  - **Exception:** the root may have only 2 children, then one key and two pointers, regardless of how large n is.

B-Trees as Indexes

B-Trees are useful for various types of indexes:

1. The search key is a primary or candidate key (i.e., no duplicates) of the data file and the B-Tree is a dense index with a key-pointer pair in a leaf for every record in the data file. The data file may or may not be sorted by primary key (primary or secondary index).

2. The search key is not a key (i.e., *duplicate values*) and the data file is *sorted* by this attribute. The B-Tree is a dense primary index with no duplicate key-pointer pairs: just a single entry for each record in the data file with search key K, and pointers to the first record with search key K.

3. The data file is *sorted* by search-key, and the B-Tree is a sparse primary index with a key-pointer pair for each data block of the data file.

4. The search key is not a key (i.e., *duplicate values*) and the data file is **NOT** *sorted*. The B-Tree is a secondary index with indirect bucket: No duplicate key-pointer pairs, just a single entry for each record in the data file with a given search key, and pointers to a bucket of pointers.

Lookup in B-trees

Problem: Given a B-tree (dense) index, find a record with search key K.

Recursive search, starting at the root and ending at a leaf:

1. If we are at a leaf then if K is among the keys of the leaf follow the associated pointer to the data file, else fail.

2. If we are at an interior node (included the root) with keys $K_1, K_2, \ldots, K_n$, then if $K < K_1$ then go to the first child, if $K_1 \leq K < K_2$ then go to the second child, and so on.

Note: B-Trees are useful for queries in which a range of values are asked for: *Range Queries*.

B-Tree Updates

Insertion and **D**eletion are more complicated that lookup. It may be necessary to either:

1. Split a node that becomes too large as the result of an insertion;
2. Merge nodes (i.e., combine nodes) if a node becomes too small as the result of a deletion.

B-Tree Insertion

Algorithm for inserting a new search key in a B-Tree.

1. Start a search for the key being inserted. If there is room for another key-pointer at the reached leaf, insert there;
2. If there is no room in the leaf, **split** the leaf in two and divide the keys between the two new nodes (each node is at least half full);
3. The splitting of nodes implies that a new key-pointer pair has to be inserted at the level above. If necessary the parent node will be split and we proceed recursively up the tree (including the root).

B-Tree Insertion: Splitting Leaves

Let N be a leaf whose capacity is n keys, and we need to insert the $(n+1)$ key-pointer pair.

1. Create a new sibling node M, to the right of N;
2. The first $\lceil \frac{n+1}{2} \rceil$ key-pointer pairs remain with N, while the other move to M.
3. The first key of the new node M is also inserted at the parent node.

Note: At least $\lfloor \frac{n+1}{2} \rfloor$ key-pointer pairs for both of the splitted nodes.

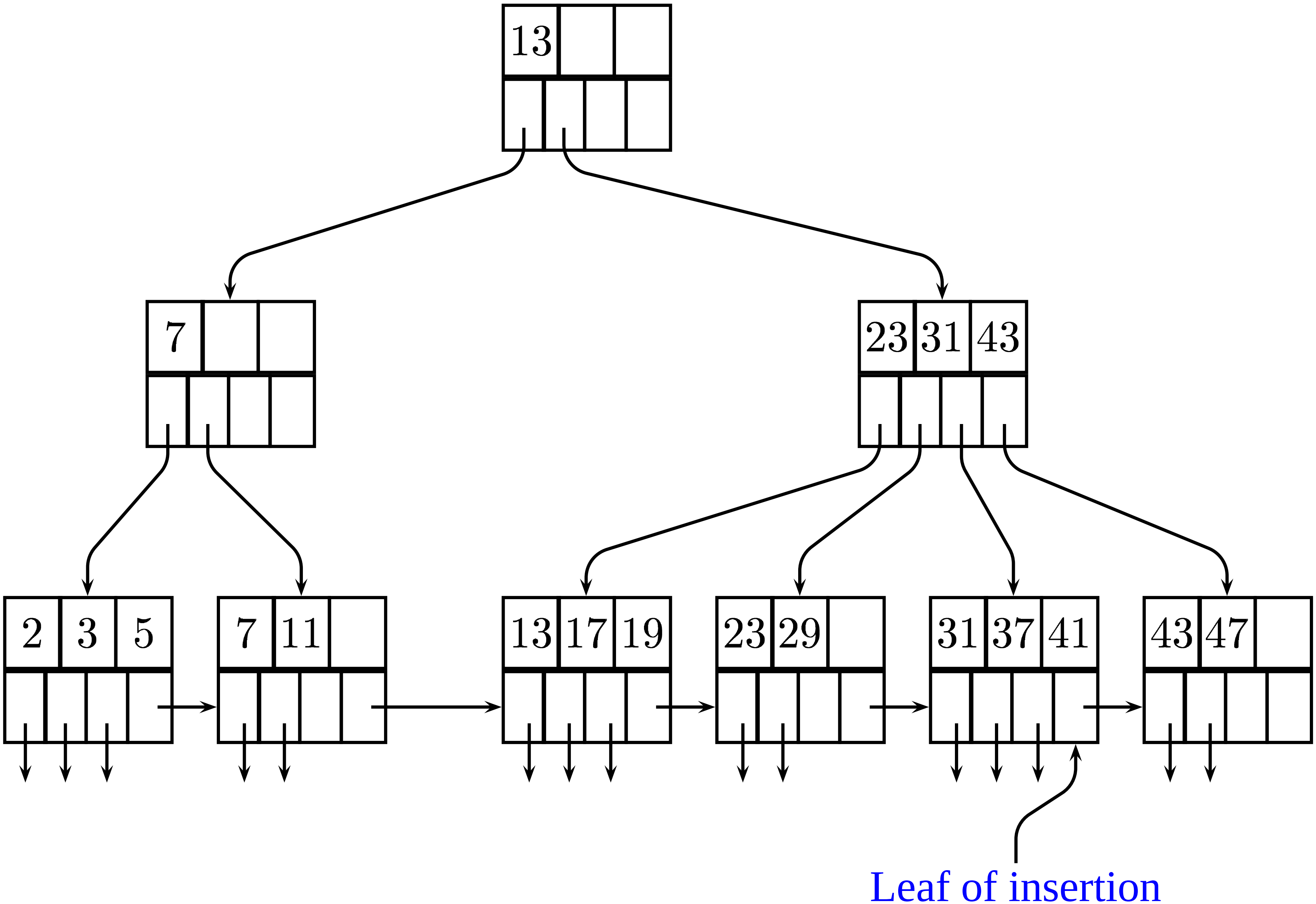
B-Tree Insertion: Splitting Interior Nodes

Let N be an interior node whose capacity is n keys and $(n+1)$ pointers, and N has been assigned the new pointer $(n+2)$ because of a node splitting at the inferior level.

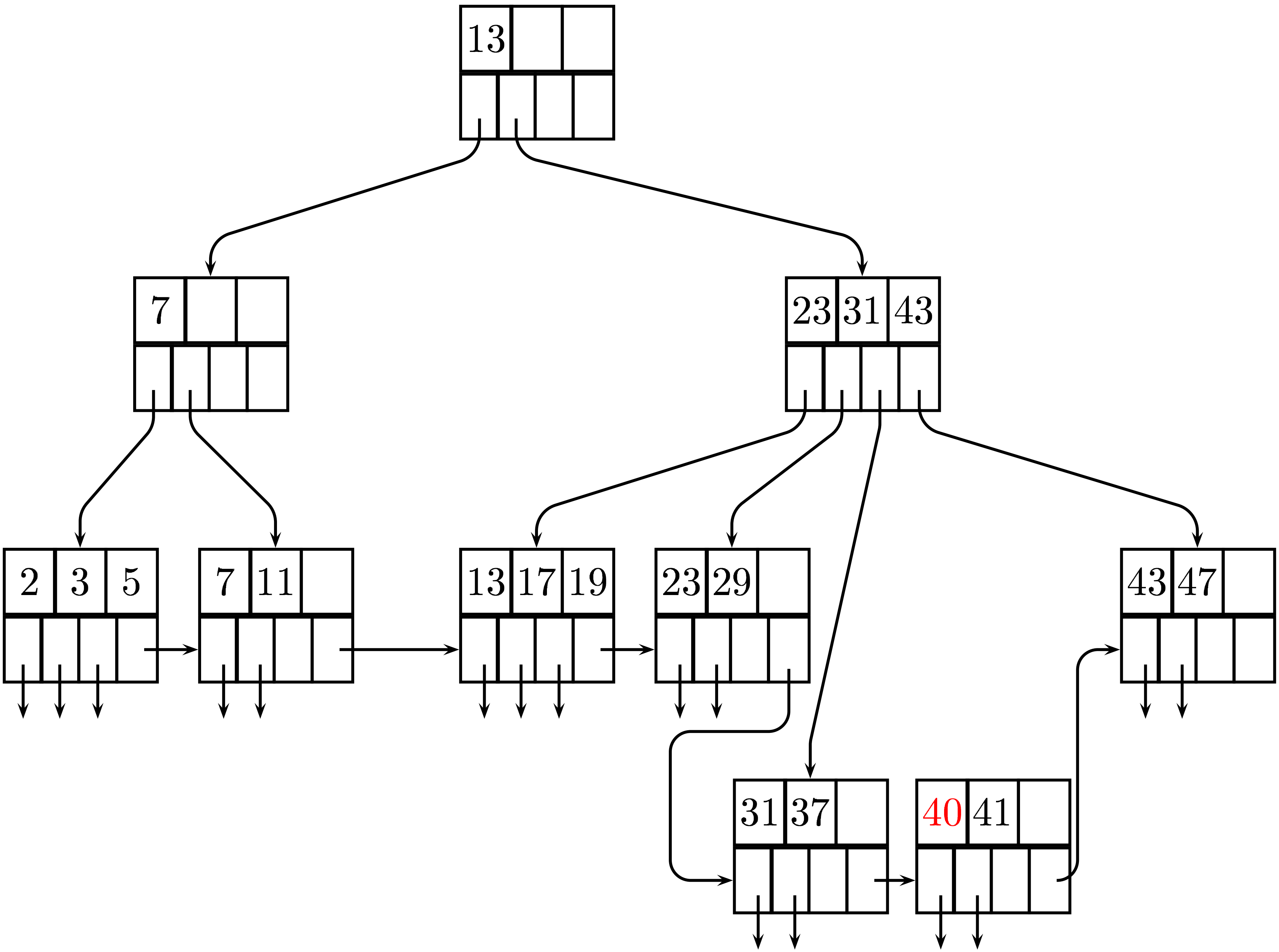
1. Create a new sibling node M, to the right of N;
2. Leave at N the first $\lceil \frac{n+2}{2} \rceil$ pointers, and move the other to M;
3. The first $\lceil \frac{n}{2} \rceil$ keys stay with N, while the last $\lfloor \frac{n}{2} \rfloor$ keys move to M. Since there are $(n+1)$ keys there is one key in the middle (say it K_l) that doesn't go with neither N nor M, but:
   - K_l is reachable via the first of M's children;
   - K_l is used by the common parent of N and M to distinguish the search between those two nodes.

Note: At least $\lceil \frac{n+1}{2} \rceil$ pointers for both of the splitted nodes.

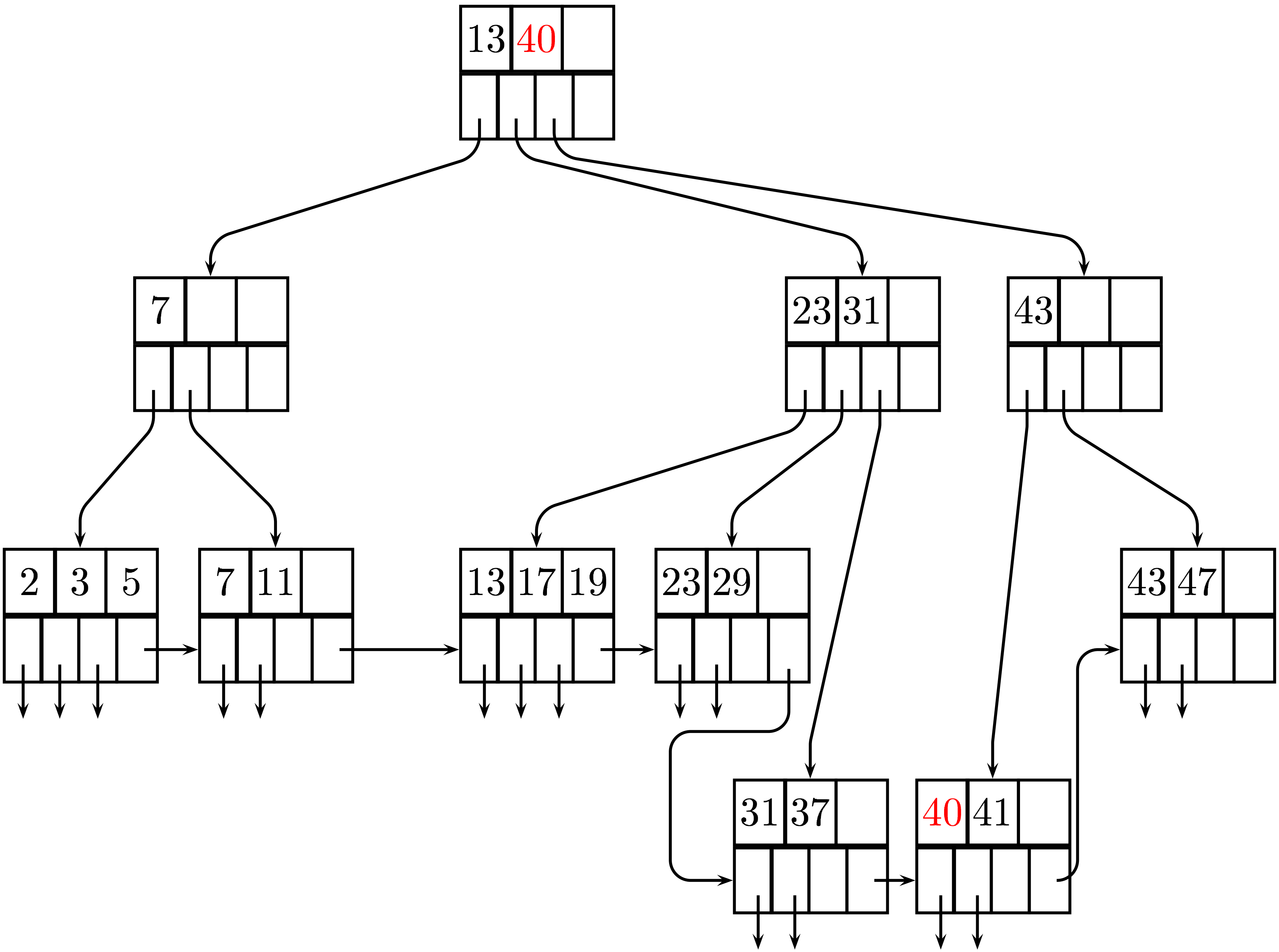
Example: B-Tree Insertion of the key 40



Example: Splitting the Leaf for Inserting Key 40



Example: Splitting Interior Nodes

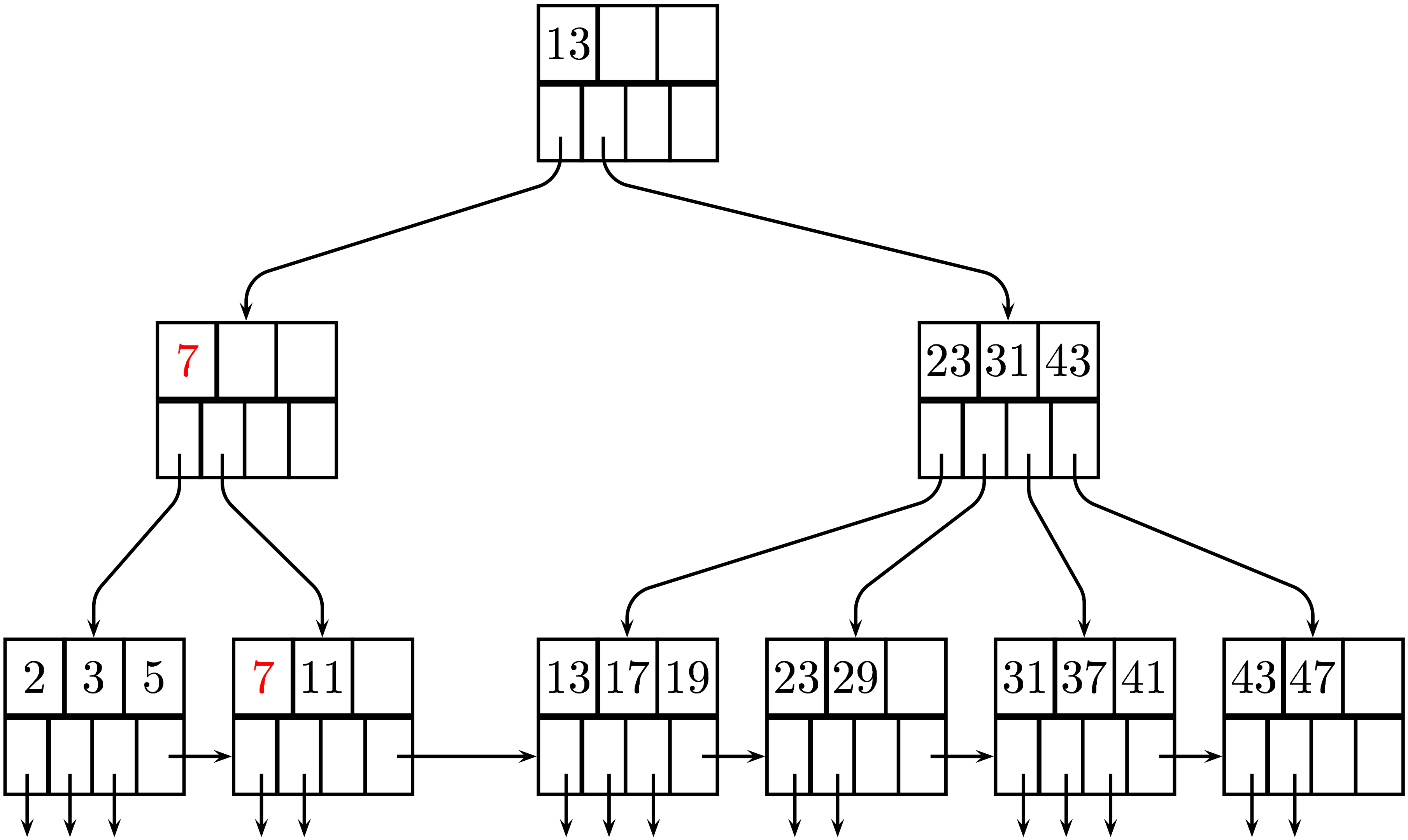


B-Tree Deletion

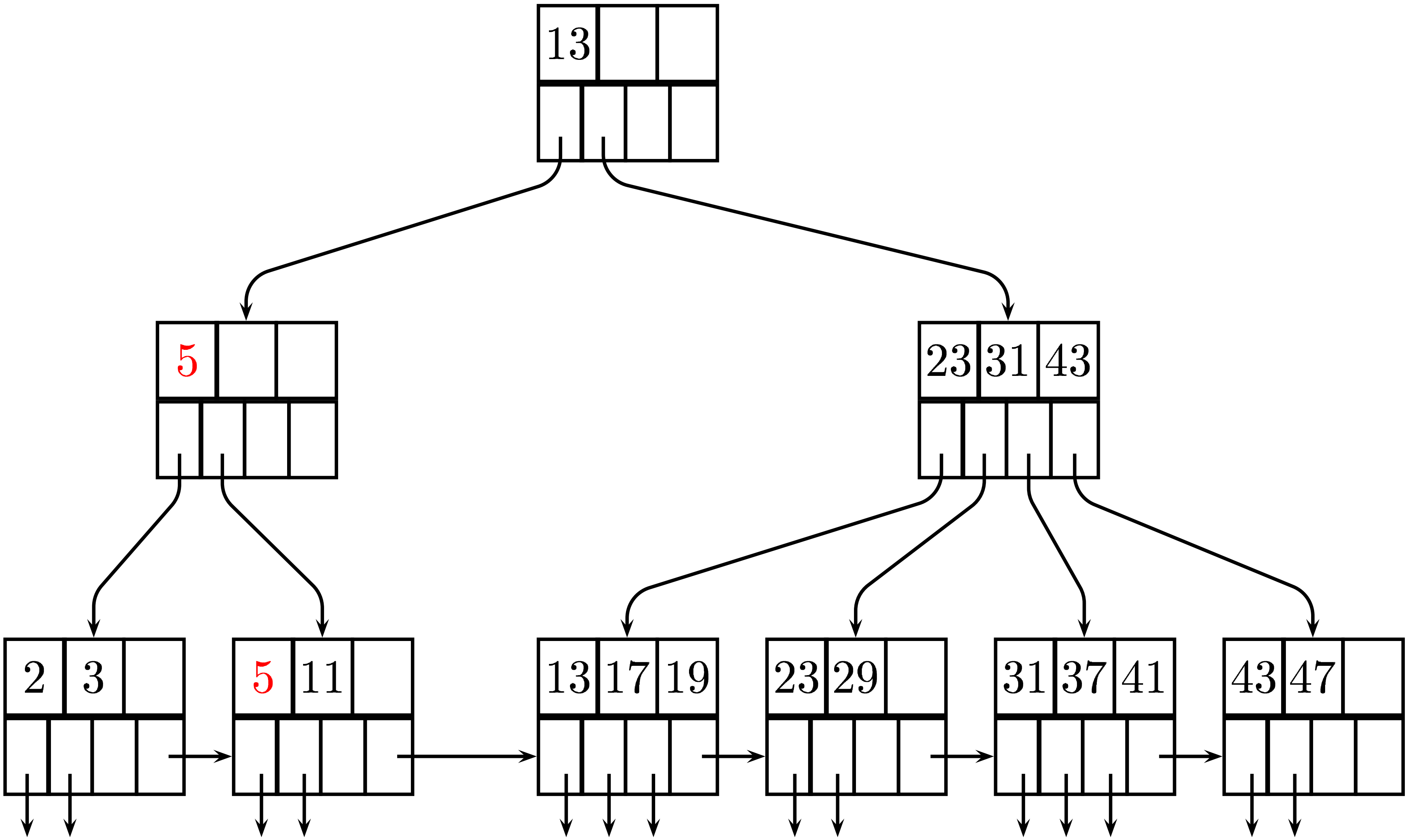
Algorithm for deleting a search key in a B-Tree.

1. Start a search for the key being deleted;
2. Delete the record from the data file and the key-pointer pair from the leaf of the B-tree;
3. If the lower limit of keys and pointers on a leaf is violated then two cases are possible:
   (a) Look for an adjacent sibling that is above lower limit and "steal" a key-pointer pair from that leaf, keeping the order of keys intact. Make sure keys for the parent are adjusted to reflect the new situation.
   (b) Hard case: no adjacent sibling can provide an extra key. Then there must be two adjacent siblings leaves, one at minimum, one below minimum capacity. Just enough to **merge** nodes deleting one of them. Keys at the parent should be adjusted, and then delete a key and a pointer. If the parent is below the minimum capacity then we recursively apply the deletion algorithm at the parent.

Example: B-Tree Deletion of the Key 7



Example: B-Tree Deletion of the Key 7 (cont.)



Efficiency of B-Trees

- B-Trees allow lookup, insertion and deletion of records of very large relations using few disk I/O.

- When the capacity n of B-Tree nodes is reasonably large ($n > 10$) splitting and merging of nodes is rare.

- 3 levels are typical: let a block be 4096 bytes, a search key be an integer of 4 bytes, and a pointer be 8 bytes $\rightarrow n = 340$. Suppose that a node is occupied midway between the minimum (170) and the maximum, then each node has 255 pointers $\rightarrow$ Root $+255$ children $+255^2 = 65025$ leaves $\rightarrow$ $65025 * 255 = 255^3 = 16.6$million pointers to data file records.

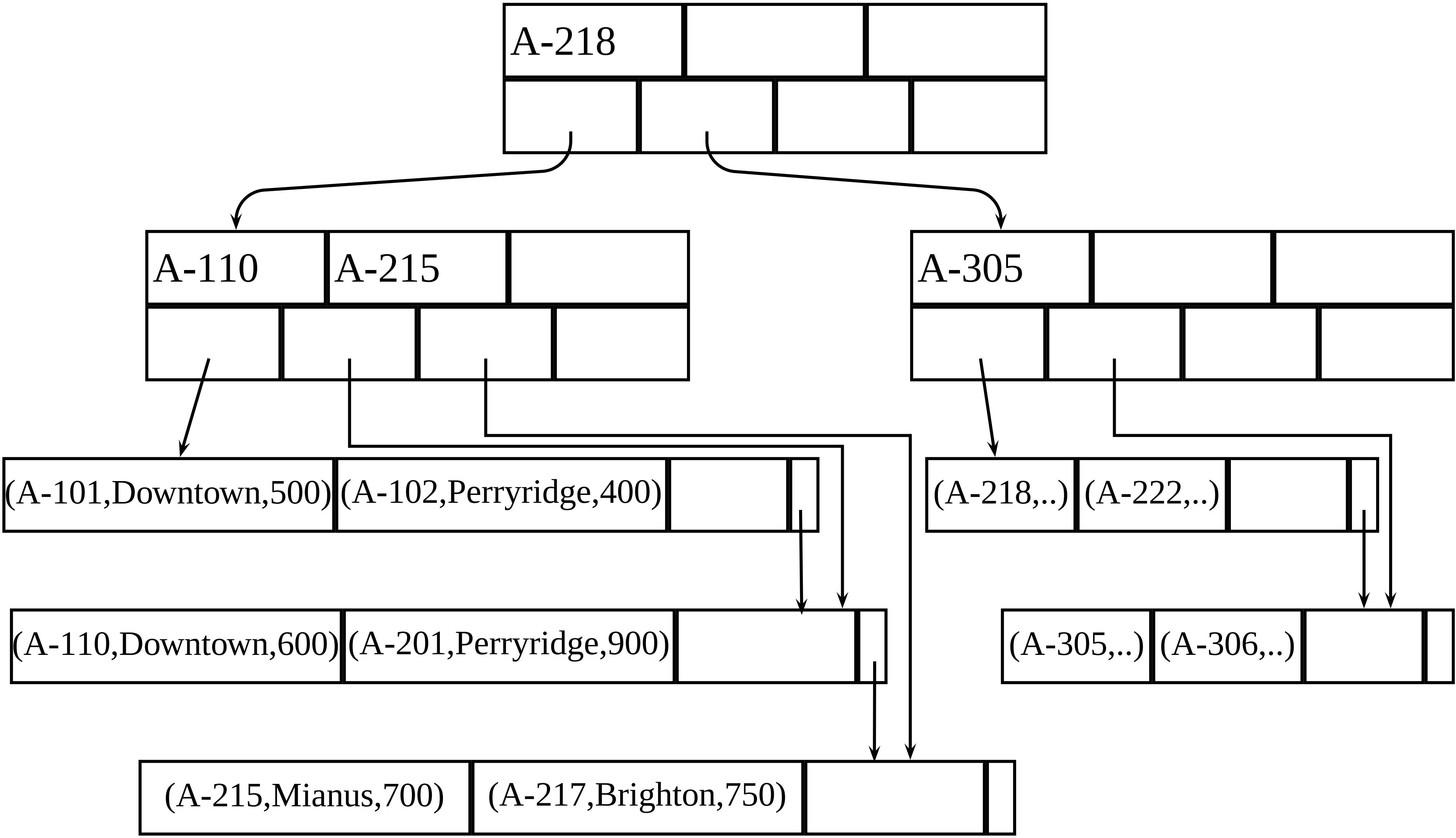
- If the root is kept in main memory lookup requires 2 disk I/O for traversing the tree and 1 disk I/O for accessing the record, if also second level in main memory a single disk I/O is sufficient for traversing the tree.

Efficiency of B-Trees (cont.)

B-Tree maintains its efficiency against relation updates.

- A relation is physically stored into a B-Tree.
- The actual records are stored in the leaf level of the B-Tree.
- Insertion and deletion can cause either node splitting or merging – i.e., no need for overflow blocks.

Efficiency of B-Trees: Relation Storage Example



Summary of Lecture II

- Indexing
  - Indexes on Sequential Files: Dense Vs. Sparse Indexes.
  - Primary Indexes with Duplicate Keys.
  - Secondary Indexes.
  - Document Indexing.
  - B-Tree Indexes.