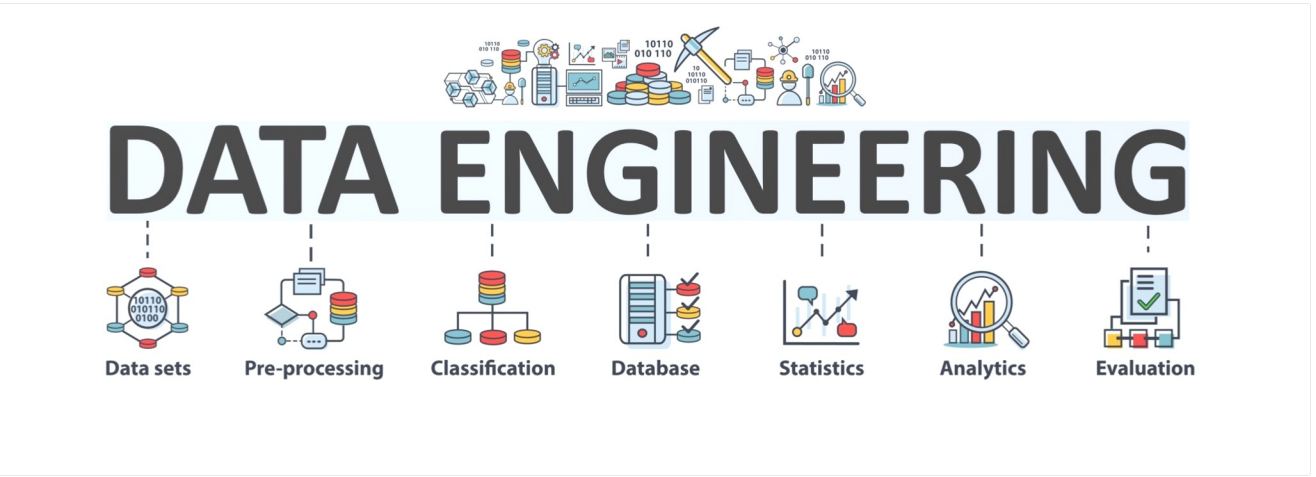


Lecture 5: Data Indexing

CS5481 Data Engineering

Instructor: Linqi Song

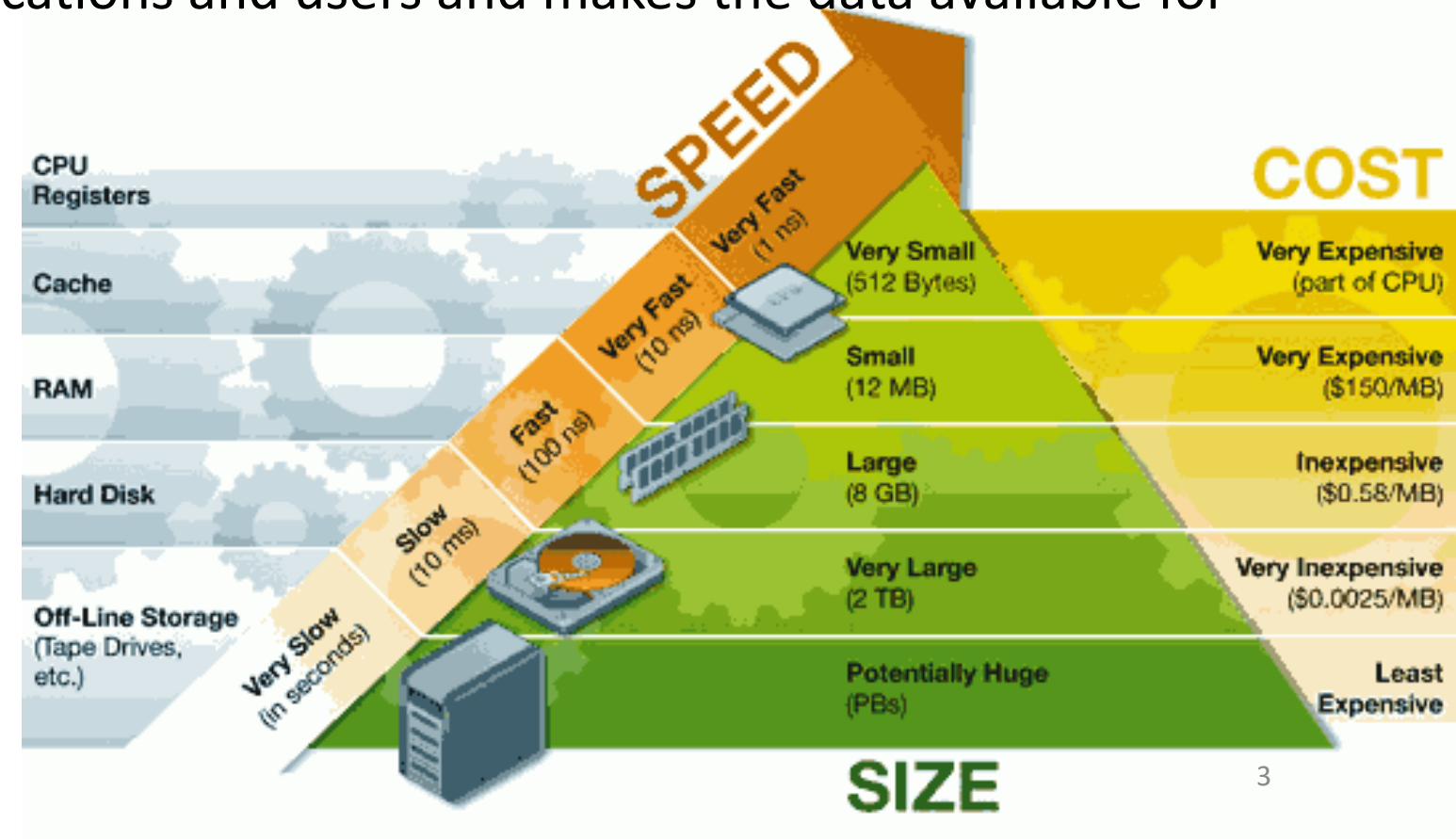
Outline

1. Data Storage
2. Records and Files
3. Hashed Files
4. Indexing Techniques
5. Tree Data Structure

Data storage

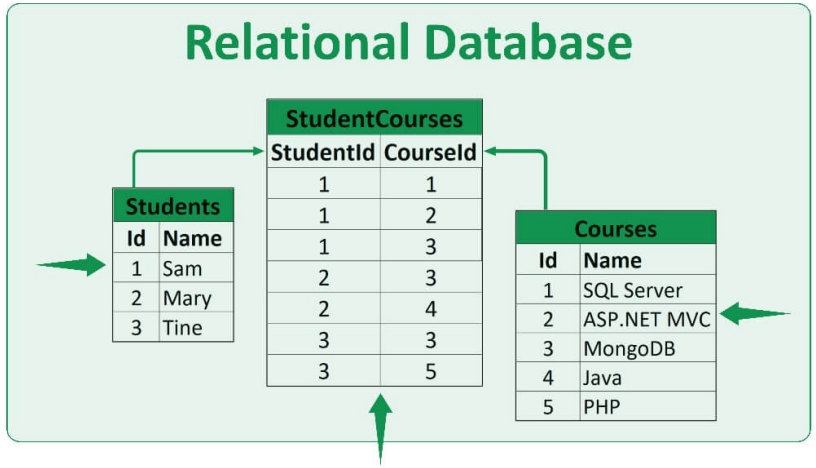
Memory Hierarchy

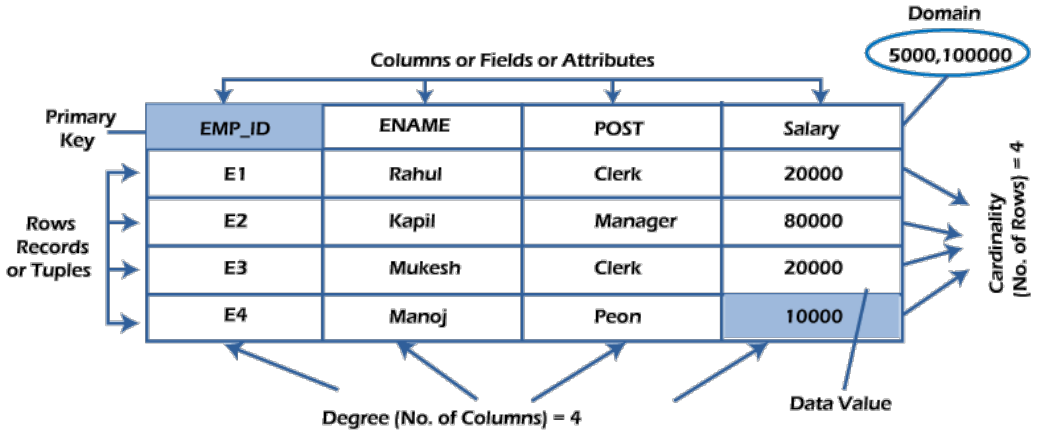
- **Storage medium:** In computers, a storage medium is a physical device that receives and retains electronic data for applications and users and makes the data available for retrieval.
- Slower in access delay but larger in memory size (less expensive).

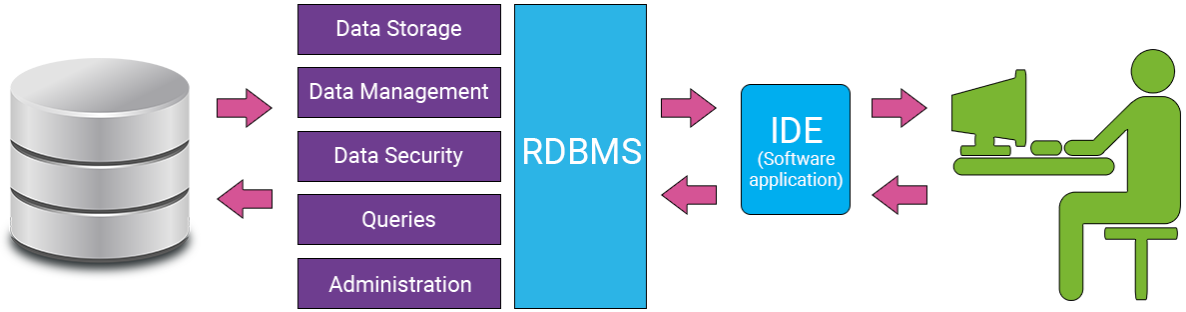


Relational databases

- A relational database is based on the relational model of data.
- A relational database management system (RDBMS) is a system to maintain relational databases.
- Many relational database systems are equipped with the option of using the SQL (Structured Query Language) for querying and maintaining the database.



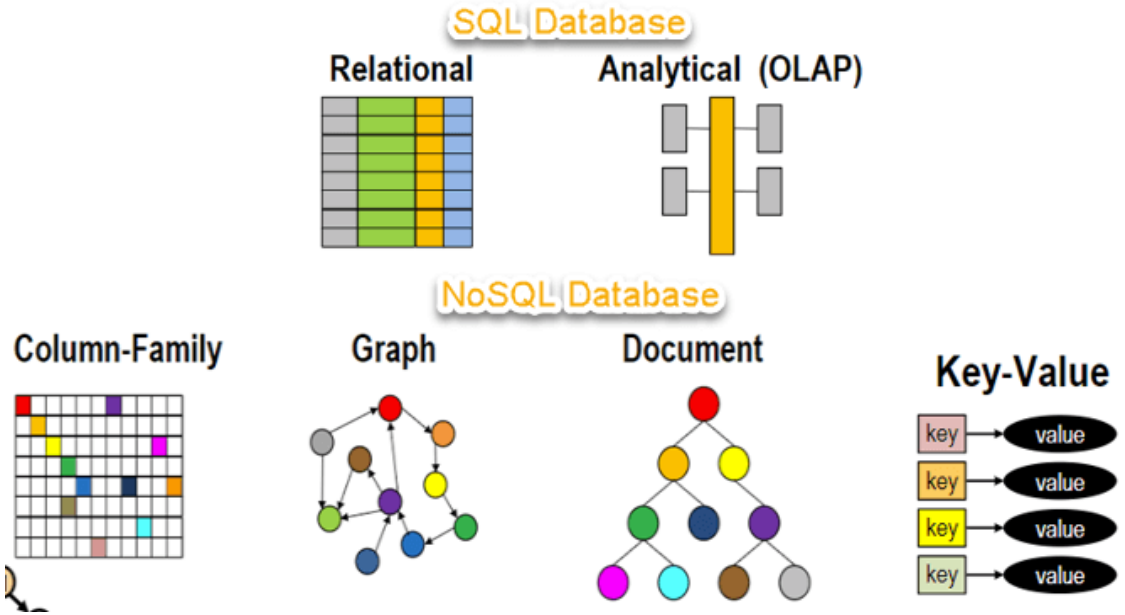




NoSQL databases

NoSQL databases are geared toward managing large sets of varied and frequently updated data, often in distributed systems or the cloud. They avoid the rigid schemas associated with relational databases. But the architectures themselves vary and are separated into four primary classifications, although types are blending over time.

- NoSQL stands for:
  - No Relational
  - No RDBMS
  - Not Only SQL



- NoSQL is an umbrella term for all databases and data stores that don't follow the RDBMS principles
  - A class of products
  - A collection of several (related) concepts about data storage and manipulation
  - Often related to large data sets

NoSQL database types (1)

Document databases

Store data elements in document-like structures that encode information in formats such as JSON.

+

Common uses include content management and monitoring web and mobile applications.

+

EXAMPLES

Couchbase Server, CouchDB, MarkLogic, MongoDB

Document data model

Collection of complex documents with arbitrary, nested data formats and varying "record" format.

Couchbase Data

Server stores metadata with each key/value pair (document)

Unique and Kept in RAM

```
meta
{
   "id": "u::tesla",
   "rev": "1-0002bce0000000000",
   "flags": 0,
   "expiration": 0,
   "type": "json"
}
```

Document Value

Most Recent In RAM And Persisted To Disk

```
document
{
   "sellerid": 123456,
   "type": "car",
   "style": "sedan",
   "year": 2013,
   "trim": "performance",
   "model": "s"
}
```

NoSQL database types (2)

Graph databases

Emphasize connections between data elements, storing related "nodes" in graphs to accelerate querying.

+

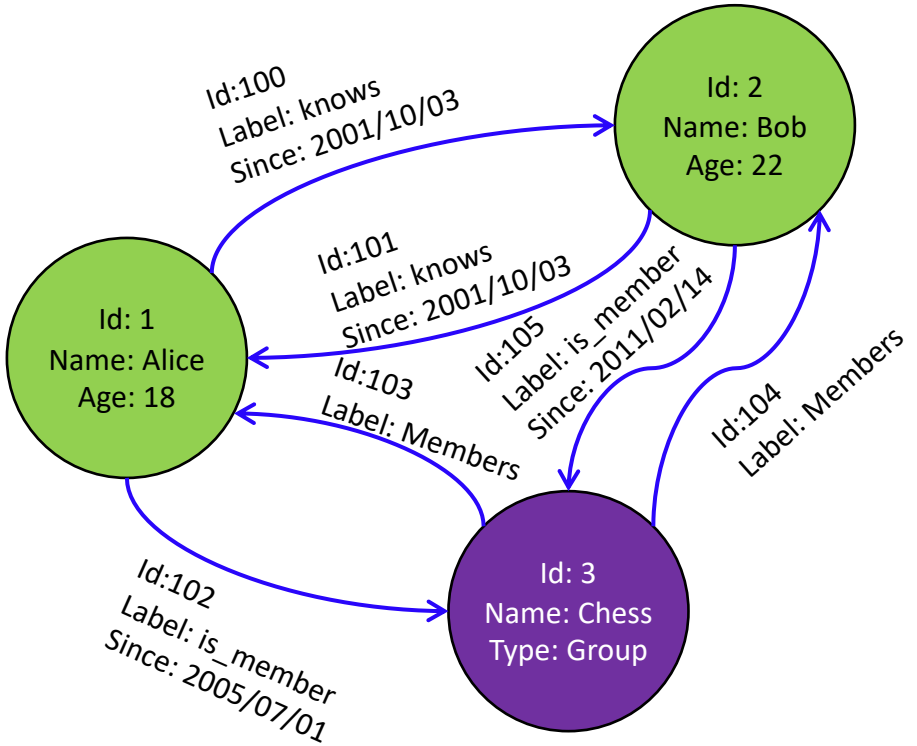
Common uses include recommendation engines and geospatial applications.

+

EXAMPLES

AllegroGraph, Amazon Neptune, ArangoDB, IBM Db2 Graph, Neo4j

- Data are represented as vertices and edges
- Graph databases are powerful for graph-like queries (e.g., find the shortest path between two elements)



NoSQL database types (3)

Key-value stores

Use a simple data model that pairs a unique key and its associated value in storing data elements.

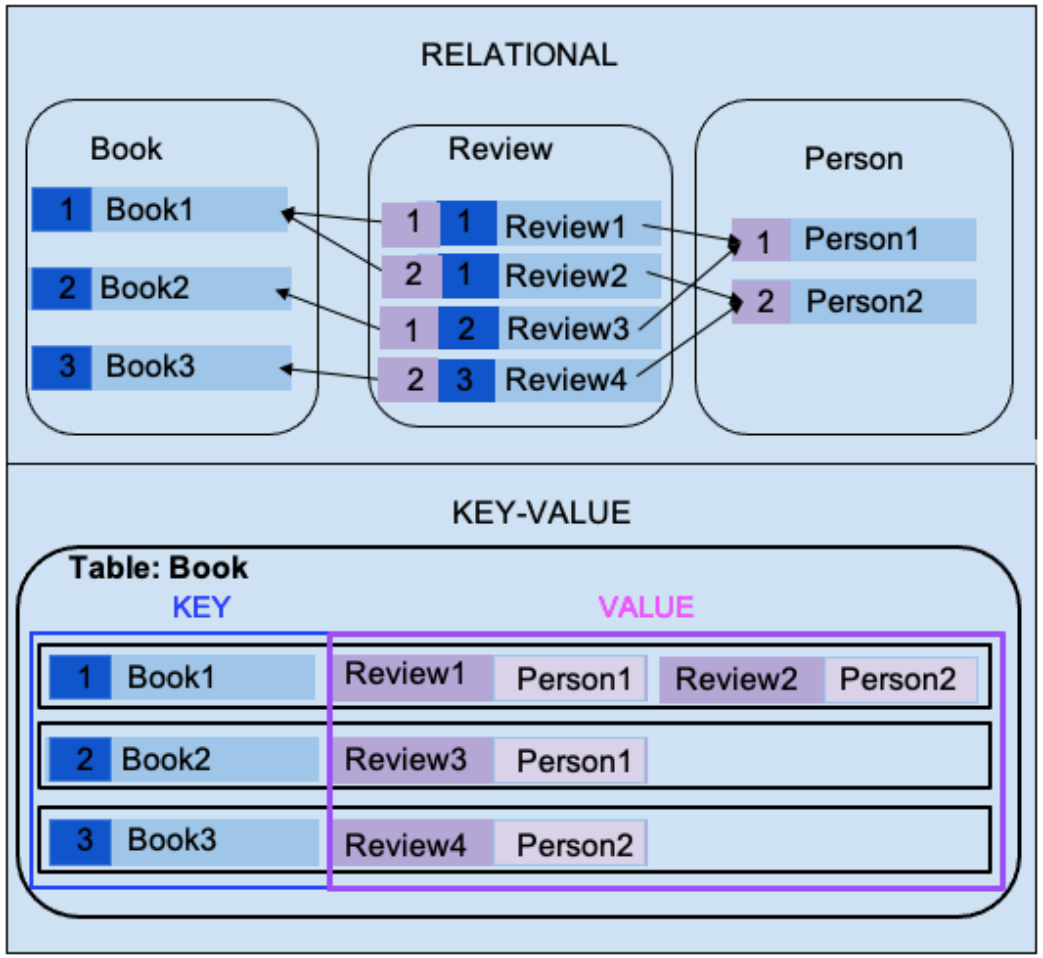
+

Common uses include storing clickstream data and application logs.

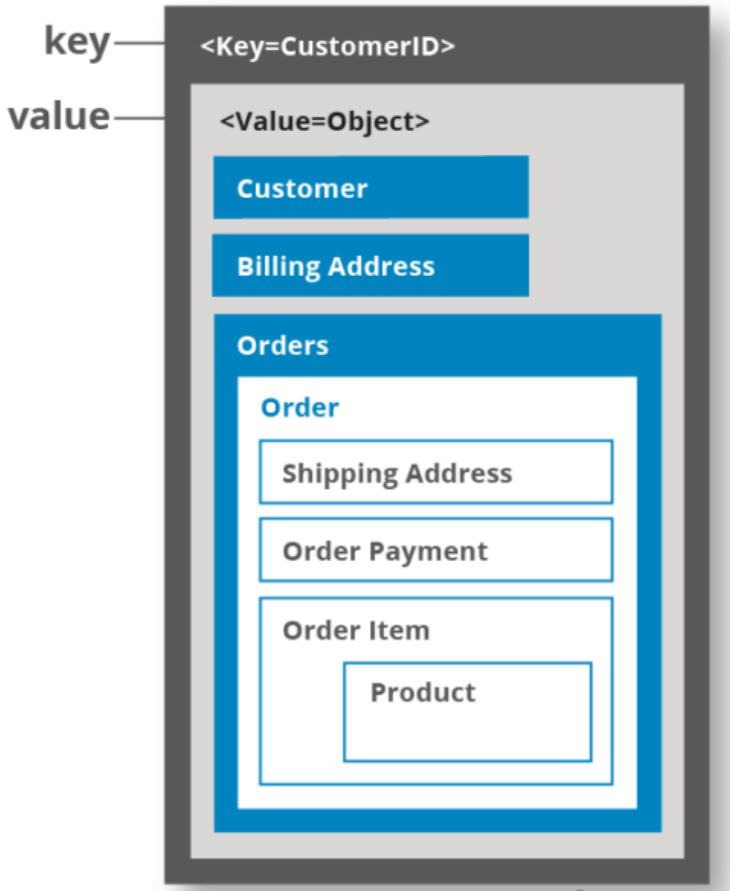
+

EXAMPLES

Aerospike, Amazon DynamoDB, Azure Table Storage, Redis, Riak



key	value
123	123 Main St.
126	(805) 477-3900



NoSQL database types (4)

Columnar Databases

Wide-column stores

Also called table-style databases, they store data across tables that can have very large numbers of columns.

+

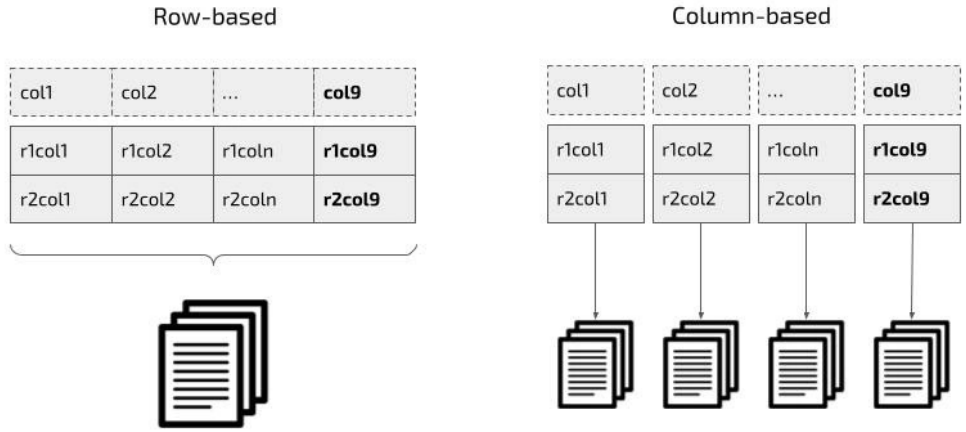
Common uses include internet search and other large-scale web applications.

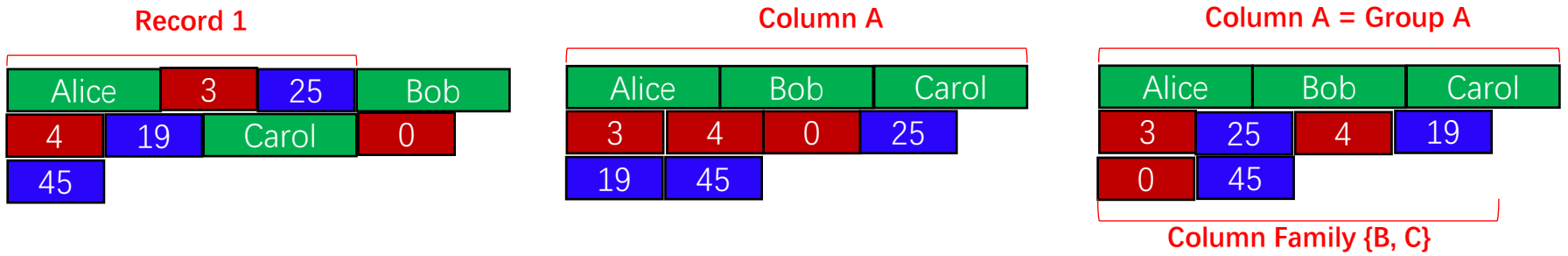
+

EXAMPLES

Accumulo, Cassandra, Google Cloud Bigtable, HBase, ScyllaDB

- Columnar databases are a hybrid of RDBMSs and Key-Value stores
  - Values are stored in groups of zero or more columns, but in Column-Order (as opposed to Row-Order)
  - Values are queried by matching keys
  - Fast access, e.g., average age

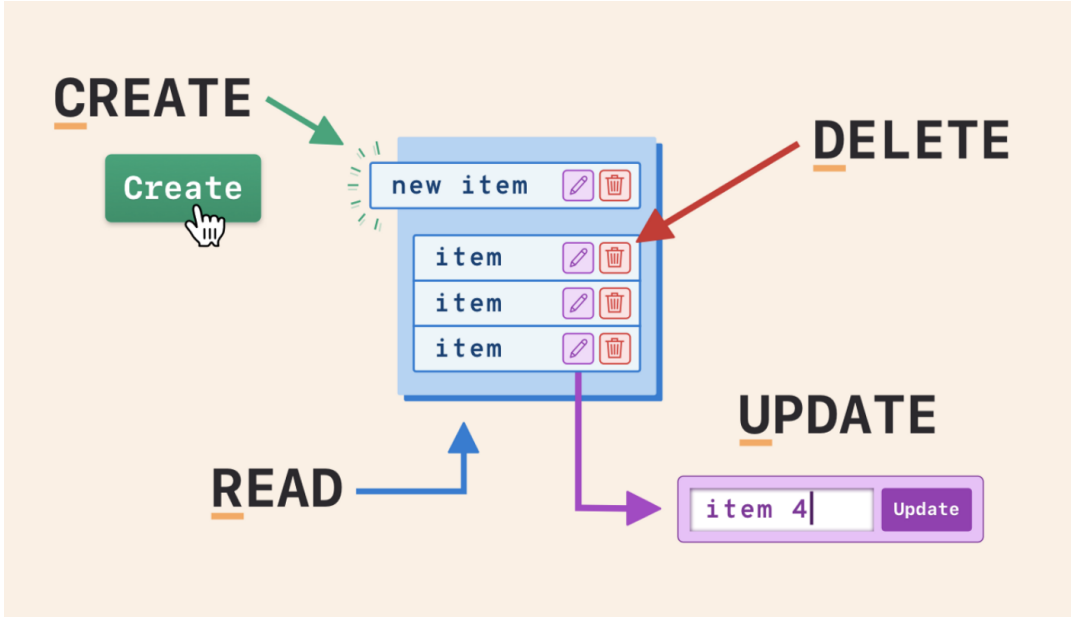


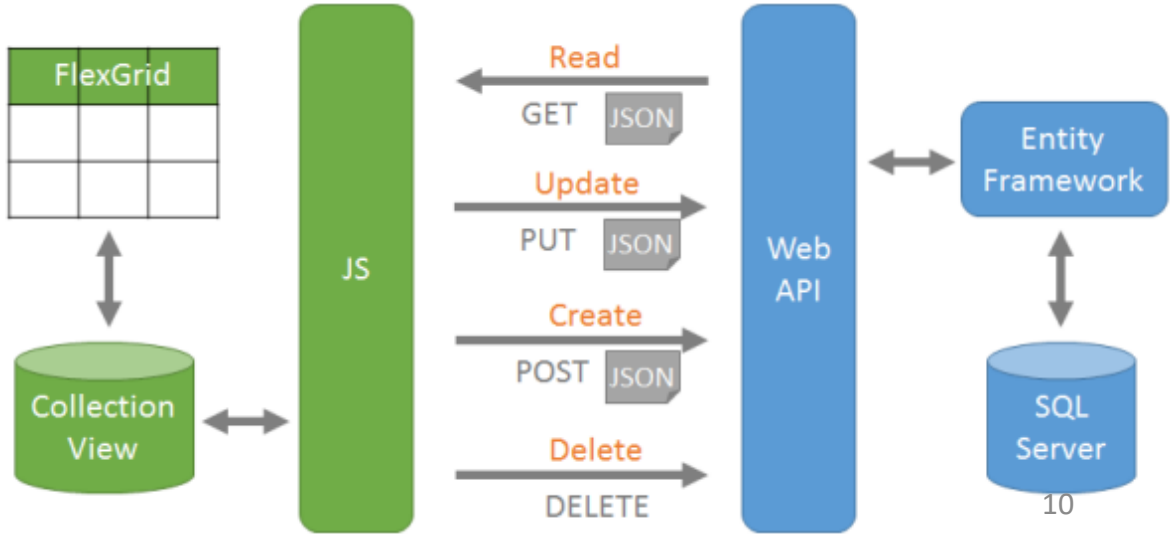


Row-Order *Columnar (or Column-Order)* *Columnar with Locality Groups*

CRUD-basic database operations

- *Create*, or add new entries
- *Read*, retrieve, search, or view existing entries
- *Update*, or edit existing entries
- *Delete*, deactivate, or remove existing entries

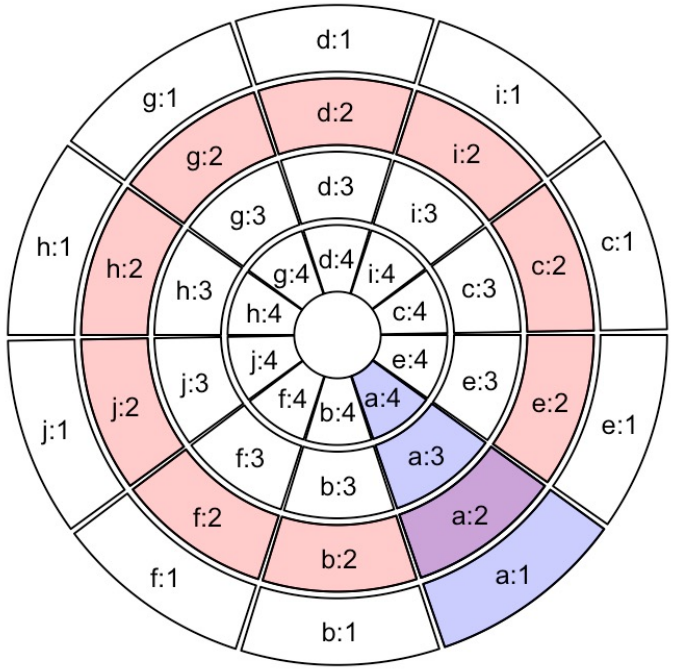


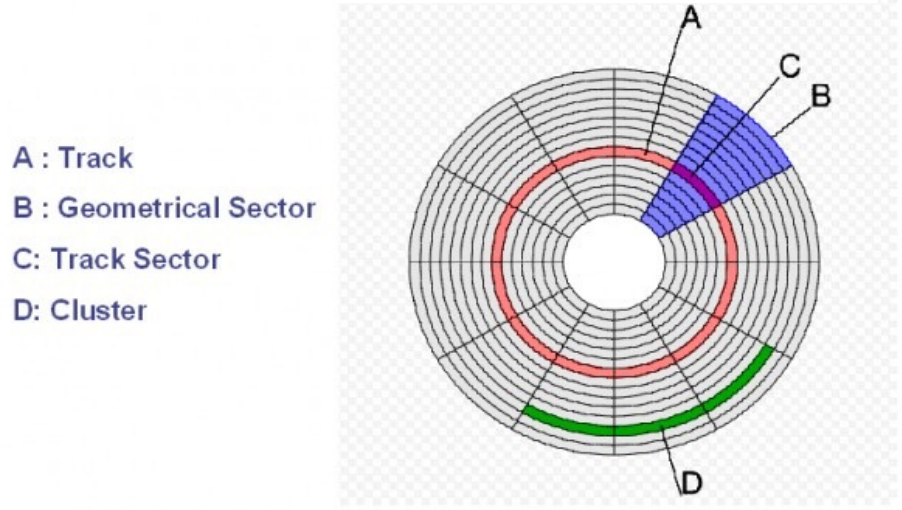


Records and files

Files of records are stored in a disc

- A record is a collection of data values (fields).
- A file (e.g., table) is a sequence of records (e.g., tuples).
- A file can have fixed-length or variable length records.
  - ❑**Fixed length records**: every record in the file has exactly the same size (in bytes).
  - ❑**Variable length records**: different records in the file have different sizes.





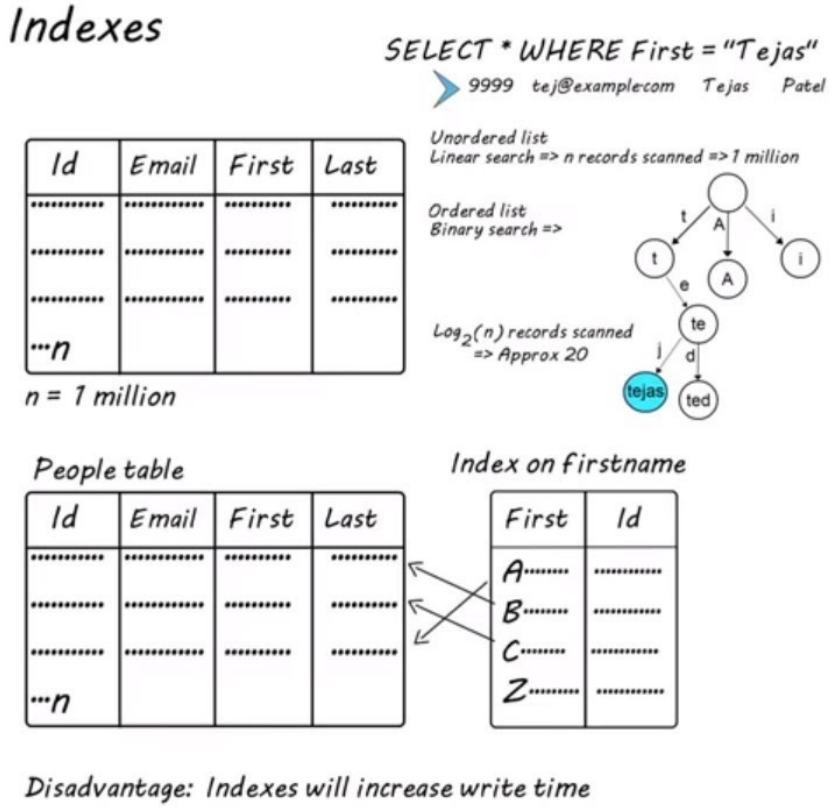
Record blocking

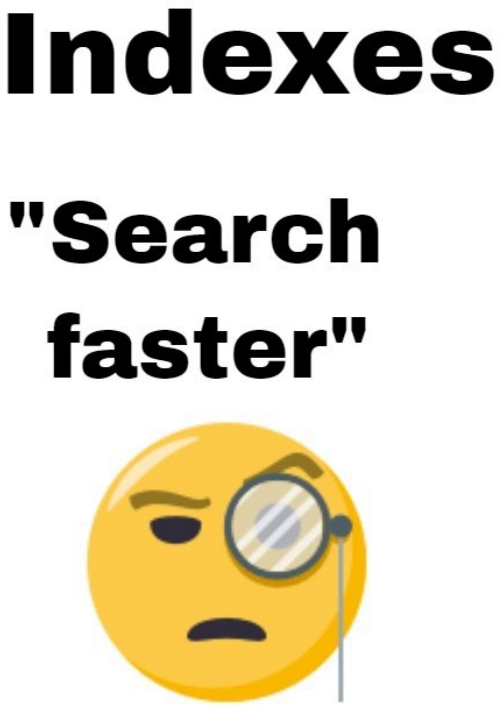
A record block is usually a track sector

- The records of a file must be allocated to disk blocks because a block is the unit of data transfer between disk and memory.
- When block size > record size, each block will contain numerous records.
  - **Blocking**: storing a number of records into one block in the disk.

Data indexing

A **database index** is a data structure that improves the speed of data retrieval operations on a database table at the cost of additional writes and storage space to maintain the index data structure.





Methods for organizing records on disks

- Unordered records (Heap files)
- Ordered records (Sorted files)
- Hashed records

Unordered files

Also called a heap file (records are unorded).

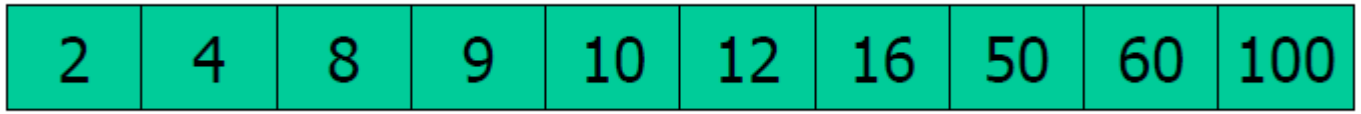
- Insertion: New records are inserted at the end of the file
  - ❑Arranged in their insertion sequence.

Record insertion is efficient (add to the end).

- Searching: A linear search through the file records is necessary to search for a record.
  - ❑This requires reading and searching half the file blocks on average, and is hence quite expensive (n/2)
  - ❑Worst case, all records (n)

Ordered files

- Also called a sequential file (records are ordered)
- File records are kept sorted by the values of an ordering field
- Insertion is expensive: records must be inserted in the correct order
  - ❑It is common to keep a separate unordered overflow file for new records to improve insertion efficiency; this is periodically merged with the main ordered file
- Reading the records in order of the ordering field is quite efficient



Ordered file example

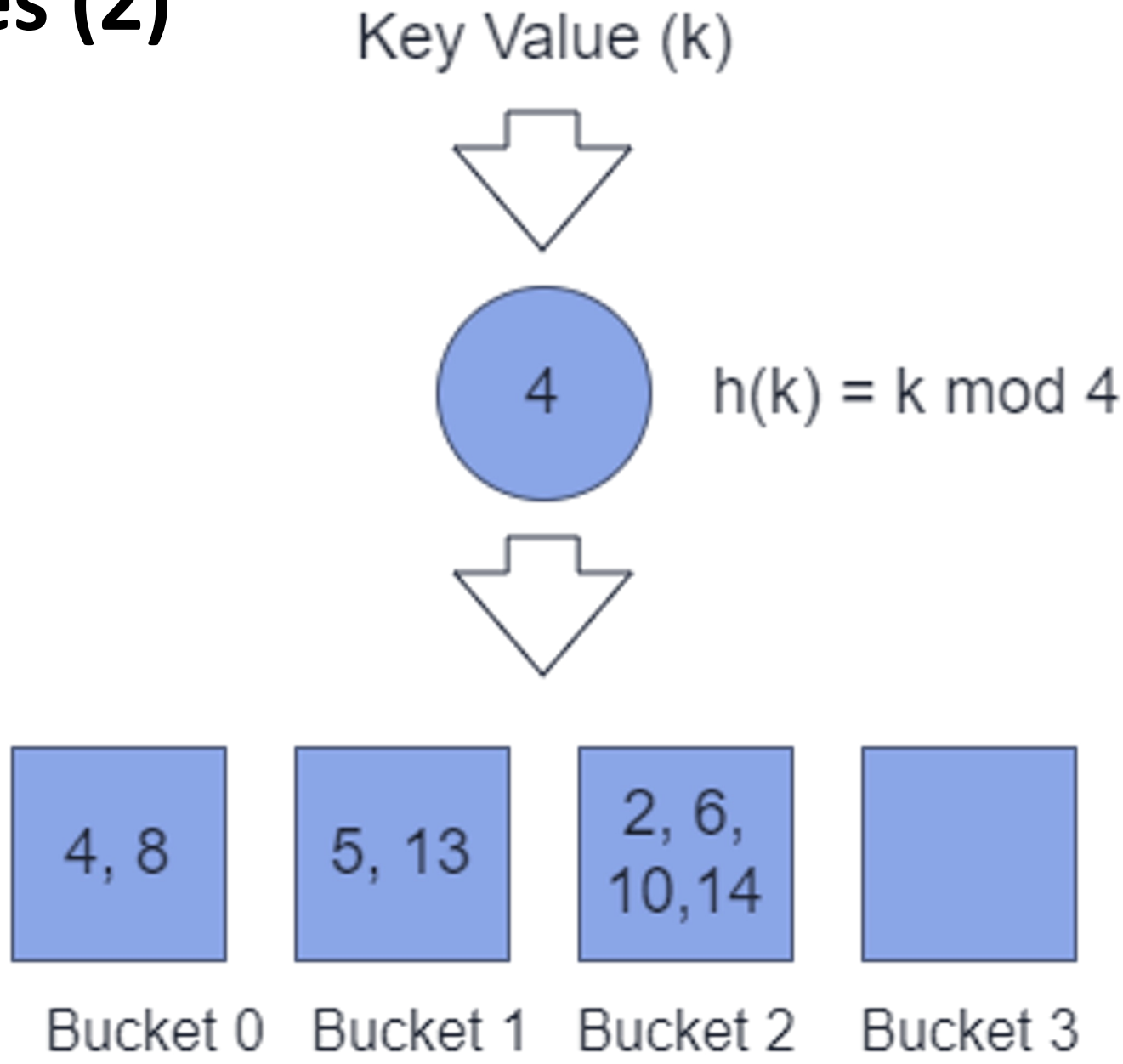
Some blocks of an ordered file of EMPLOYEE records with Name as the **ordering key field**.

	Name	Ssn	Birth_date	Job	Salary	Sex
Block 1	Aaron, Ed					
	Abbott, Diane					
	⋮					
	Acosta, Marc					
Block 2	Adams, John					
	Adams, Robin					
	⋮					
	Akers, Jan					
Block 3	Alexander, Ed					
	Alfred, Bob					
	⋮					
	Allen, Sam					
Block 4	Allen, Troy					
	Anders, Keith					
	⋮					
	Anderson, Rob					

Hashed files: a simple indexing idea (1)

- Hashing for disk files is called External Hashing (files on disk)
- The file blocks are divided into M equal-sized buckets, numbered bucket 0, bucket 1, ..., bucket M-1
- One of the file fields is designated to be the hash key of the file
- If a record has search key K, we store the record by linking it to the bucket list for the bucket numbered h(K)
  - ❑h(K) is a hash function that transforms the hash field value into an integer between 0 and M-1.

Hashed files (2)



Hashed files (3)

External hashed files

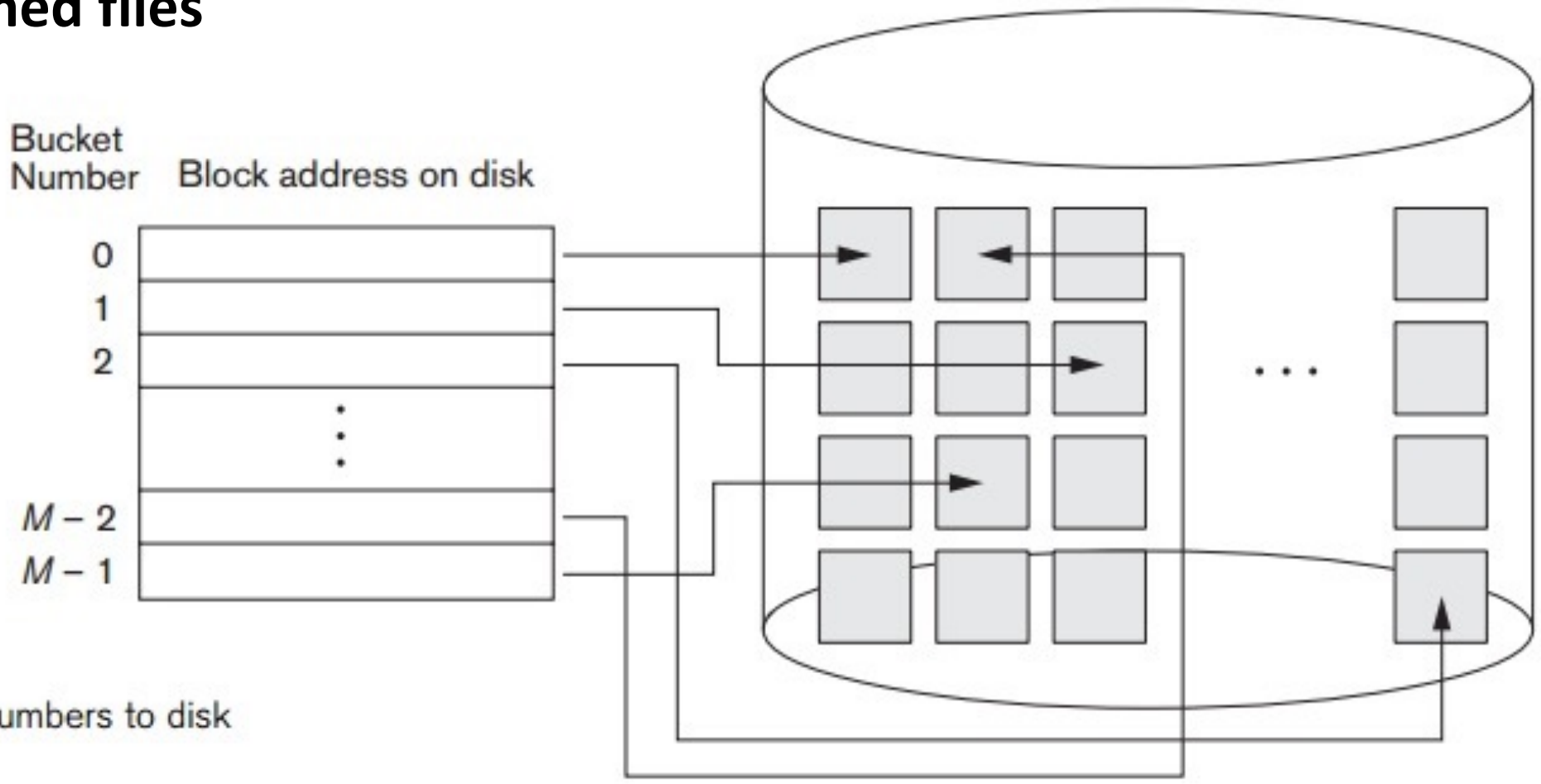
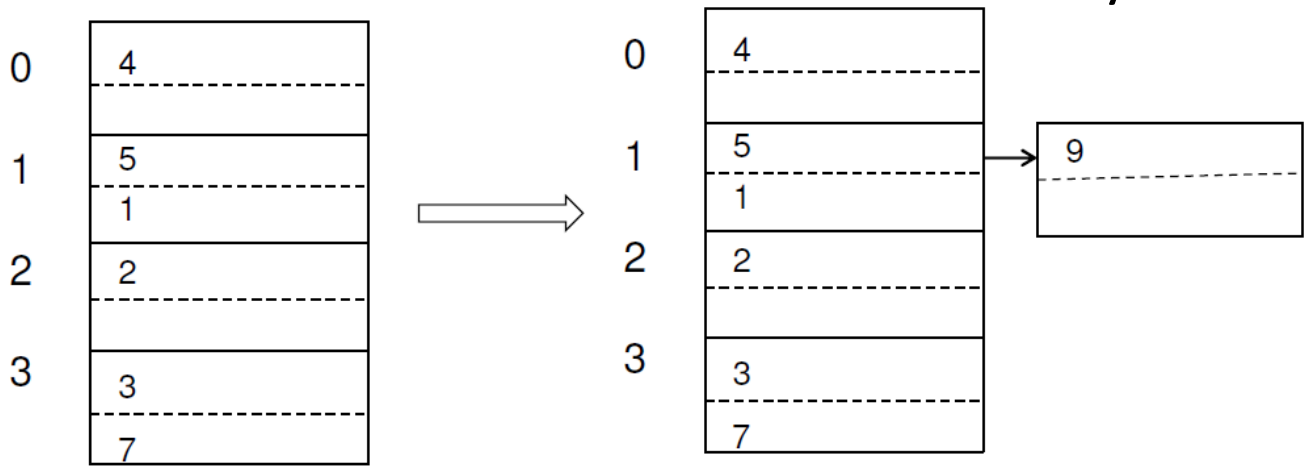


Figure 17.9
Matching bucket numbers to disk block addresses.

Hashed files (4)

Hashed tables example

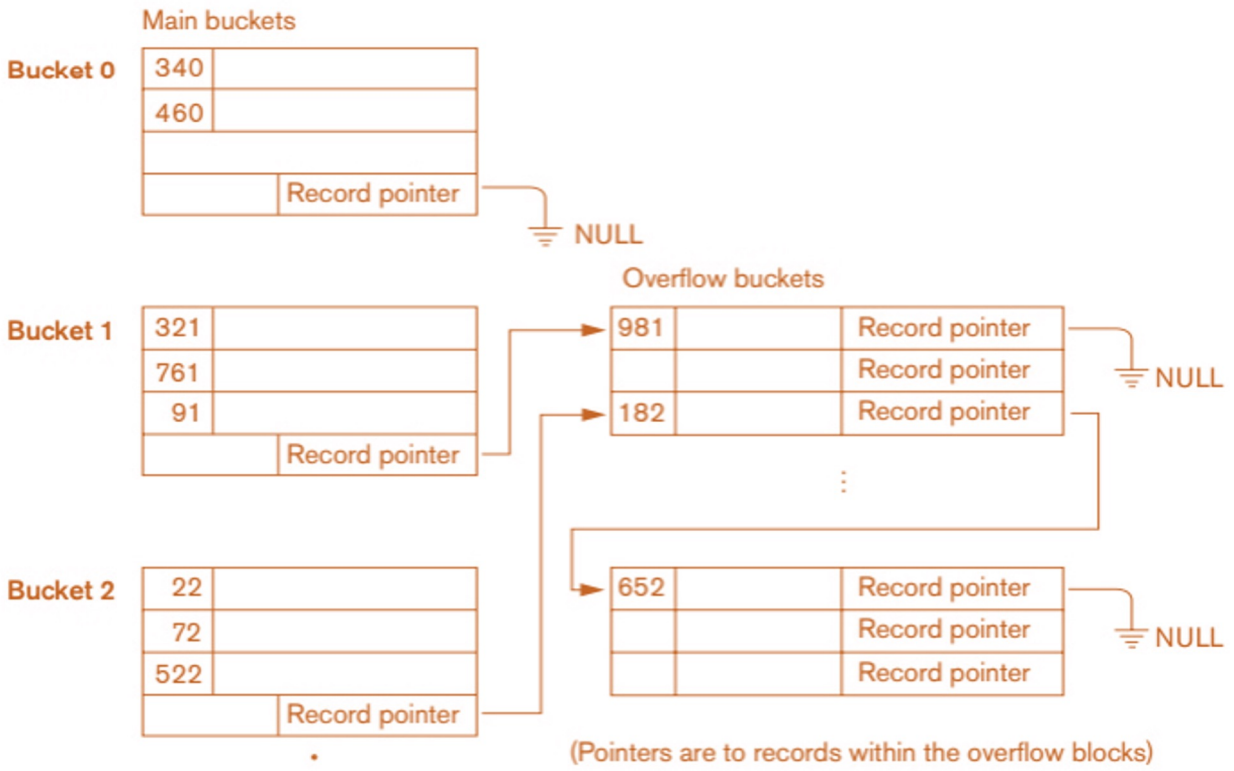
- We assume a block can hold two records and M = 4, i.e., the hash function h returns values from 0 to 3.
- Example: We add to the hash table a record with key g and h(g) = 1. Then we add the new record to the bucket numbered 1.
- **Collisions** occur when a new record hashes to a bucket that is already **full -> Collision resolution** is needed.



Hashed files (5)

Collision resolution by chaining

- For this method, various overflow locations are kept, usually by extending the array with a number of overflow positions
- In addition, a pointer field is added to each bucket
- A collision is resolved by:
  - ❑placing the new record in an unused overflow bucket and
  - ❑setting the pointer of the occupied hash address bucket to the address of that overflow bucket



Hashed files (6)

- To reduce the number of overflow, a hash file is typically kept 70-80% full.
- The **hash function h** should distribute the records **uniformly** among the buckets.
  - Otherwise, search time will be increased because many overflow records will exist.
  - Searching overflow records are more expensive.
- Main disadvantages of **static external hashing**:
  - **Fixed number of buckets M** -> Difficult to expand and shrink the file dynamically.
  - -> Need hashing techniques that allow **dynamic file expansion.**

Hashed files (7)

Extendible and dynamic hashing: Hashing techniques are extended to allow **dynamic growth and shrinking** of the number of file records .

- Both dynamic and extendible hashing use the **binary representation** (e.g., 1100…) of the hash value h(K) in order to access a directory.
  - In **Extendible Hashing**, the directory is an **array** of size 2^d, where d is called the global depth.
  - In **Dynamic Hashing**, the directory is a **binary tree**.
- The directories can be stored on disk, and they expand or shrink dynamically.
  - Directory entries point to the disk blocks that contain the stored records.
- An insertion in a disk block that is full causes the block to split into two blocks and the records are redistributed among the two blocks.
  - The directory is updated appropriately.

Hashed files (8)

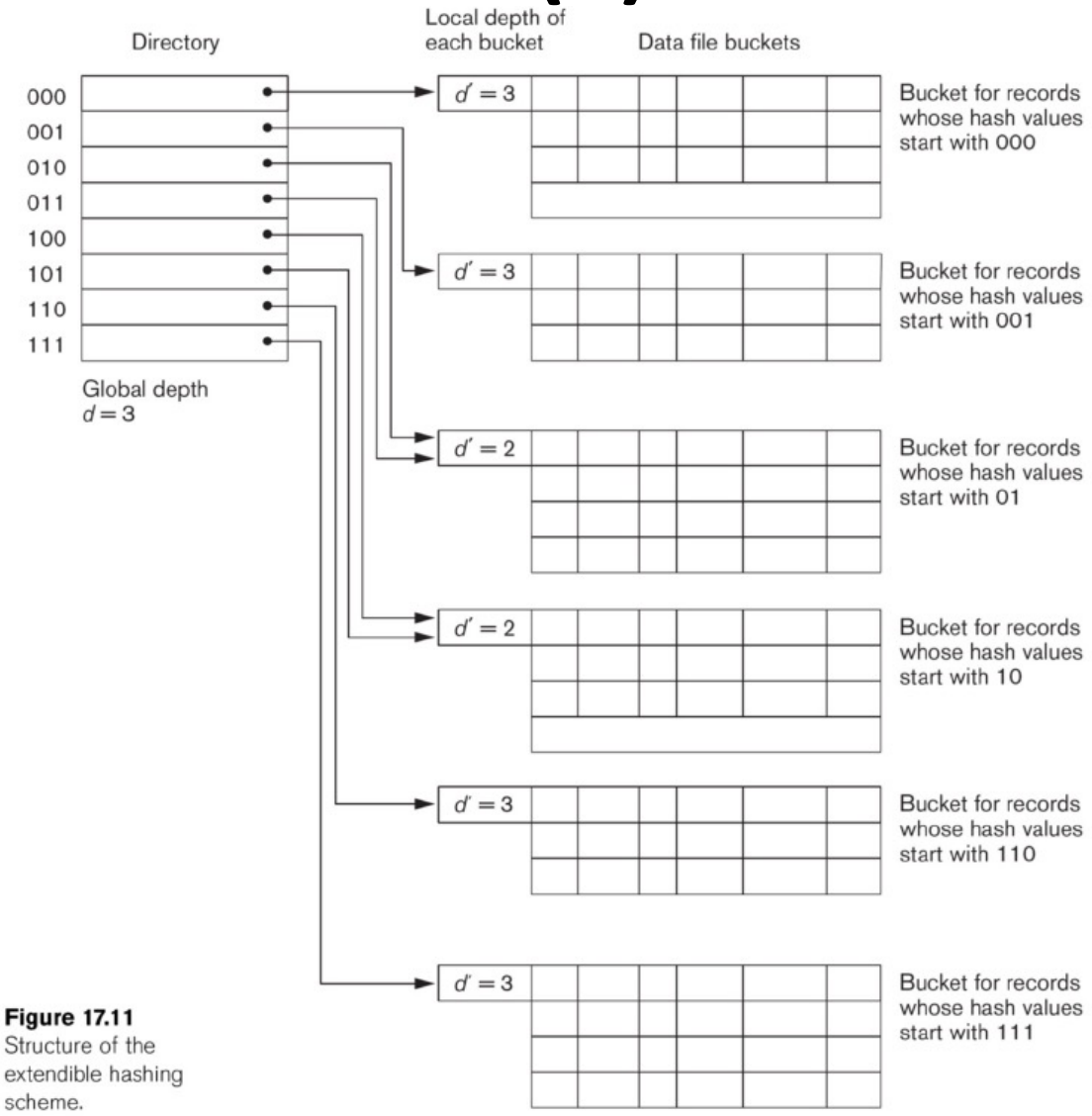


Figure 17.11
Structure of the extendible hashing scheme.

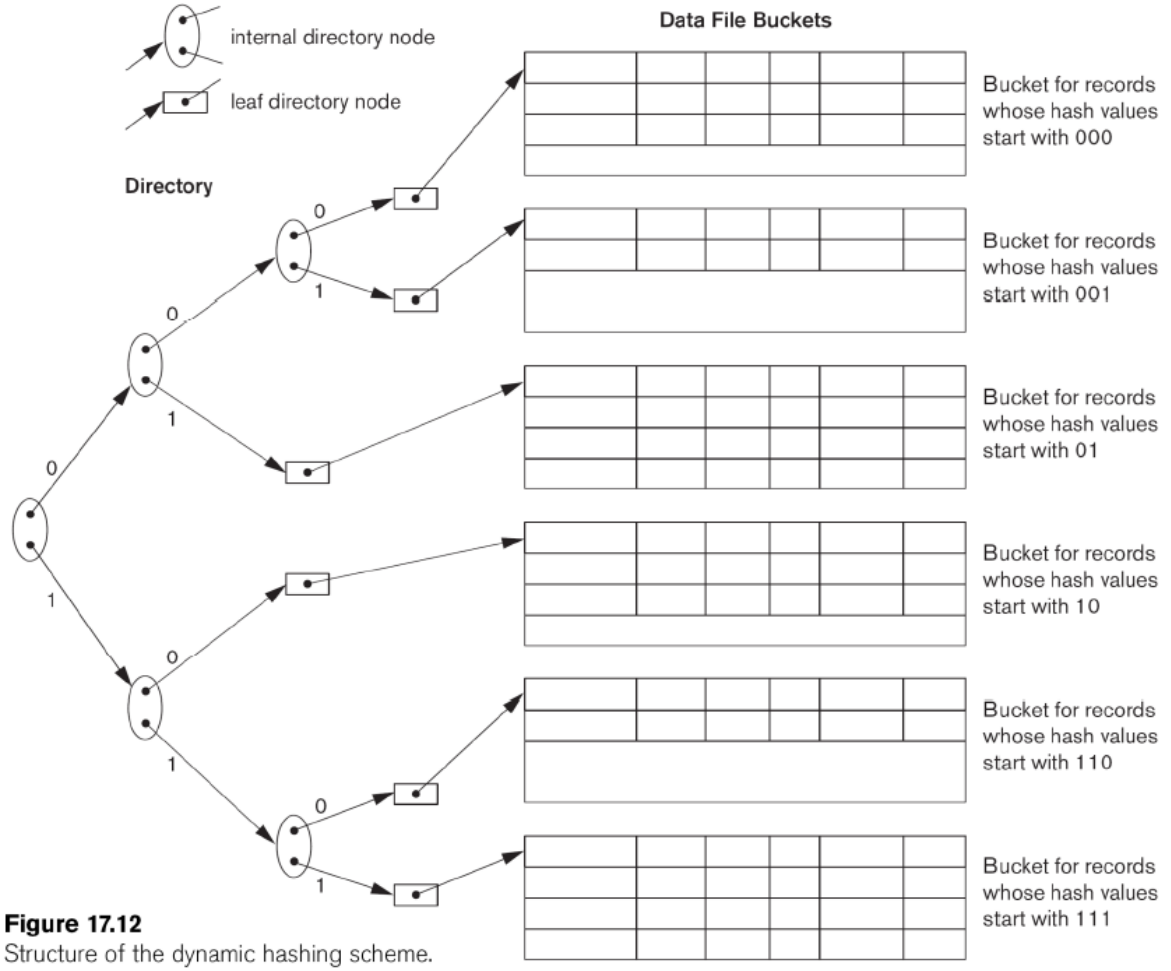


Figure 17.12
Structure of the dynamic hashing scheme.

Extendible Hashing

Dynamic Hashing

Indexing techniques (1)

- **Index** are **auxiliary files** on disk that provide secondary access paths, which provide **fast access to records** **without affecting the physical placement of records** in the primary data file on disk.
- An index is **an ordered sequence of entries <indexing field value, pointer to record>**:
  - ❑ It is ordered by the **indexing field value**
  - ❑ The **pointer** provides the address of the required record.
- Instead of searching the data file sequentially, we search the index to get the address of the required records.
- The index file usually occupies **considerably less disk blocks** than the data file because its entries are much smaller

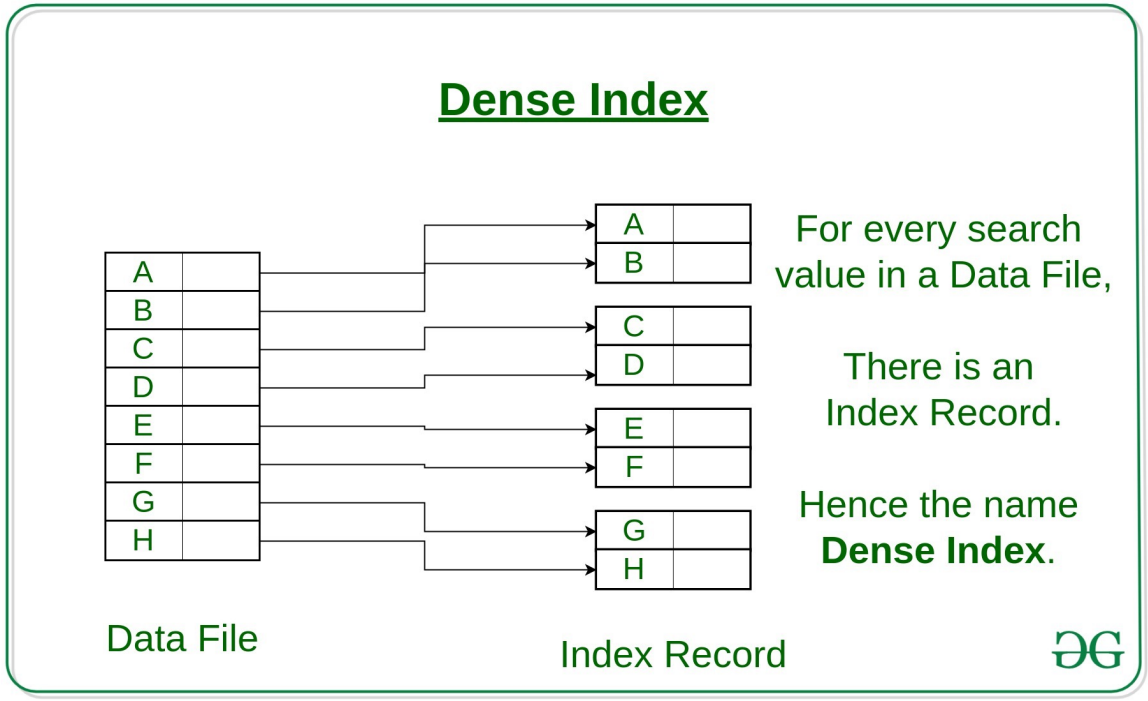
Indexing techniques (2)

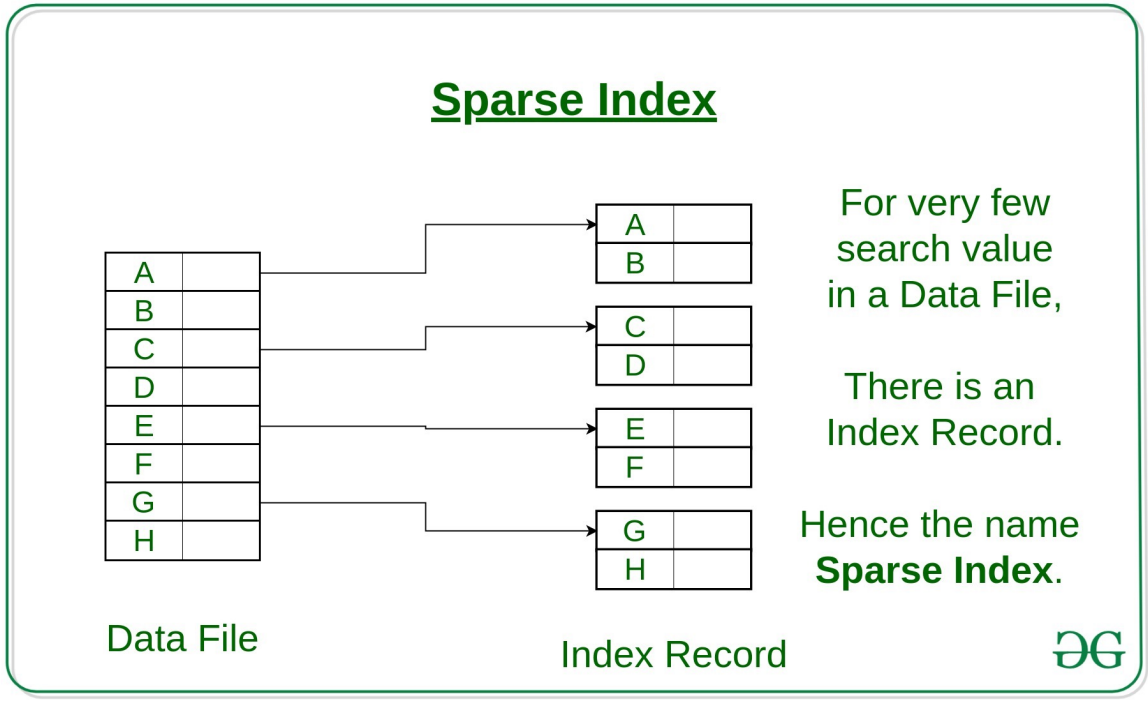
Dense index vs sparse index

- A **dense index** has an index entry for every search key value (and hence every record) in the data file.
  - ❑ Thus larger index size
- A **sparse index**, on the other hand, has index entries for only some of the search values.
  - ❑ Thus smaller index size

Indexing techniques (3)

Dense index vs. sparse index





Indexing techniques (4)

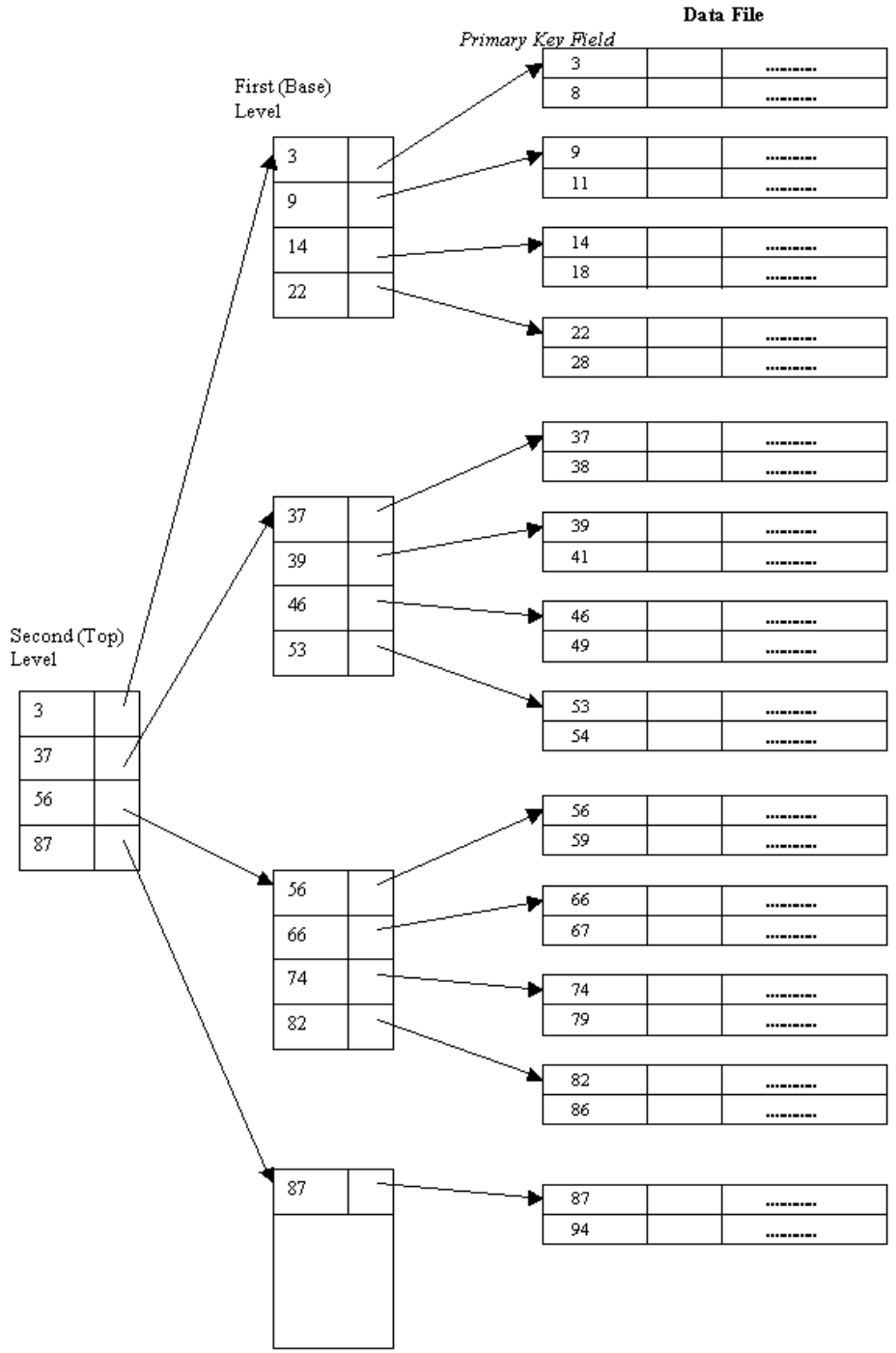
Multi-level indices

- Because a single-level index is an ordered file, we can create another **index to the index itself**.
  - ❑ In this case, the original index file is called the **first-level index** and the index to the index is called the **second-level index**.
- We can repeat the process, creating a third, fourth, ..., top level **until all entries of the top level fit in one disk block.**

Indexing techniques (5)

A 2-level index

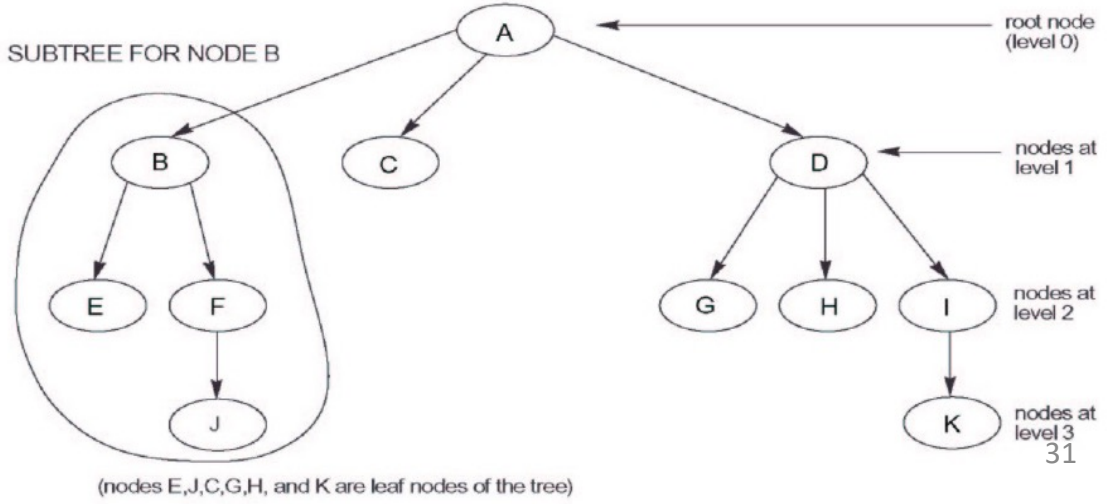
A **multi-level index** can be created for any type of first-level index as long as the first-level index consists of **more than** one disk block.



Tree data structure (1)

A tree is formed of multi-level nodes

- Except the root node, each node in the tree has one parent node and zero or more child node.
- A node that does not have any child nodes is called a **leaf node**.
- A non-leaf node is called **internal node.**
- A sub-tree of a node consists of the node and all its descendant.
- If the leaf nodes are at different levels, the tree is called **unbalanced.**



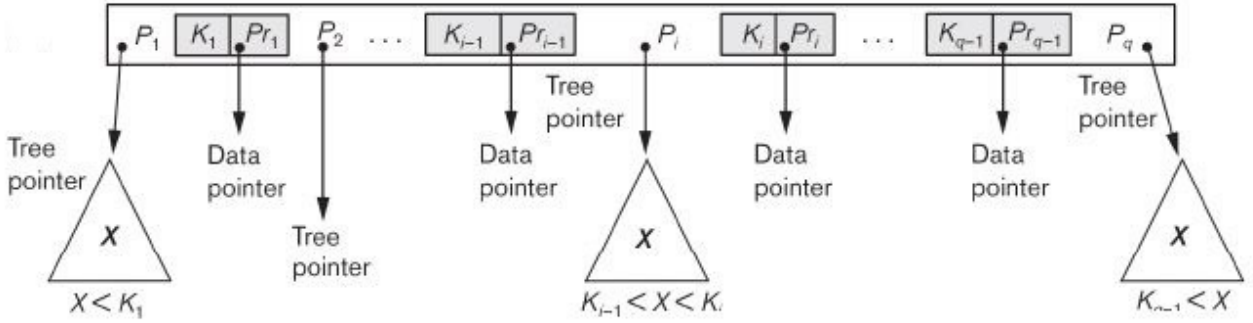
Tree data structure (2)

B-tree index

Each internal node in a B-tree is of the form

<P1, <K1, Pr1>, P2, <K2, Pr2>, …, <Kq-1, Prq-1>, Pq> where q ≤ p

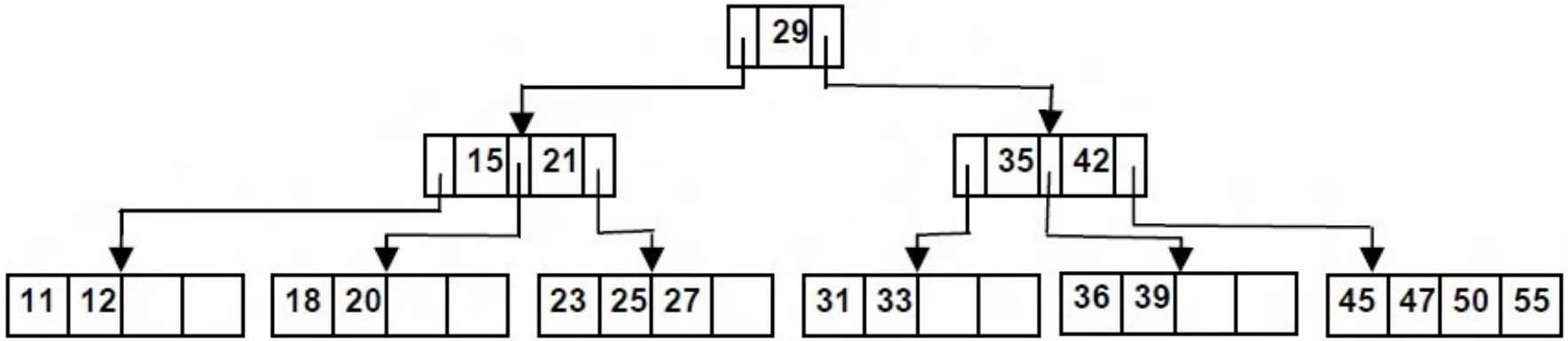
- Each Pi is a tree pointer to another node in the B-tree
- Each Pri is a data pointer points to the record whose search key field value = Ki
- Within each node, K1 < K2 < … < Kq-1
- For all search key field values X in the subtree pointed by Pi, we have Ki-1 < X < Ki for 1 < i < q; X < Ki for i = 1; and Ki-1 < X for i = q



Tree data structure (3)

B-tree index

- All **leaf nodes** are at the same level (**balanced tree**).
- All leaf nodes have the same structure as internal nodes except that all of their **tree pointers** of Pi are NULL.
- Each node has at **most p tree pointers (order of the tree).**
- Each node except the root and leaf nodes, has **at least ceil(p/2)** tree pointers.



Tree data structure (4)

B+ tree index

- Most implementations of a dynamic multilevel index use a variation of the B-tree data structure called a B+-tree.
- In a B+-tree, data pointers are stored only at the leaf nodes of the tree.
  - ❑ The pointers in internal nodes are tree pointers to blocks that are tree nodes.
  - ❑ The pointers in leaf nodes are data pointers to the data file records.
- Because entries in the internal nodes of a B+-tree does not include data pointers, more entries (tree pointers) can be packed into an internal node and thus fewer levels (higher capacity).
- The leaf nodes of a B+-tree are usually linked to provide ordered access on the search field to the records.

Tree data structure (5)

B+ tree index (internal nodes)

The structure of internal nodes of a B+-tree of order p

- Each internal node is of the form **<P1, K1, P2, K2, …,Pq-1, Kq-1,Pq>** where $q \leq p$
  - Pi : tree pointer
  - Ki : search key field value
- Within each node, $K1 < K2 < \ldots < Kq-1$
- For all search key field values X in the subtree pointed by Pi, we have $Ki-1 < X \leq Ki$ for $1 < i < q$; $X \leq Ki$ for $i = 1$; and $Ki-1 < X$ for $i = q$ or $Ki-1 \leq X < Ki$ for $1 < i < q$; $X < Ki$ for $i = 1$; and $Ki-1 \leq X$ for $i = q$.
- Each node has at **most p tree pointers**
- Each node except the root and leaf nodes has **at least ceil(p/2)** tree pointers

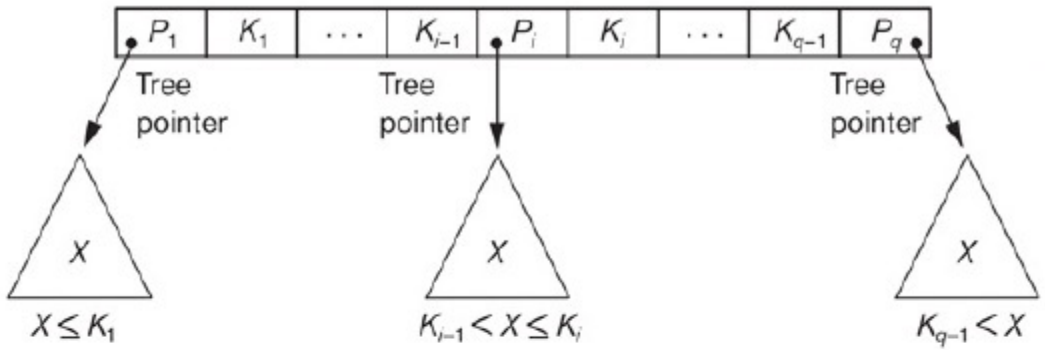
Tree data structure (6)

B+ tree index (leaf nodes)

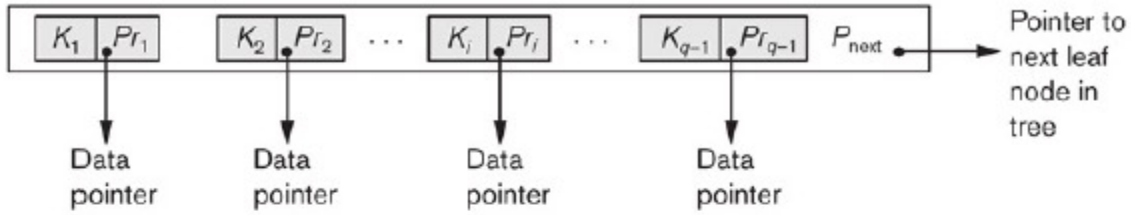
- Each leaf node is of the form **<<K1, Pr1> ,<K2, Pr2>, …,<Kq-1, Prq-1>, Pnext>**
  - ❑ Pri is a data pointer pointing to the record whose search field is Ki.
  - ❑ Pnext points to the next leaf node of the tree.
- Within each node, K1 < K2 < … < Kq-1.
- Each leaf node has at least ceil(p/2) values.
- All leaf nodes are at the same level.

Tree data structure (7)

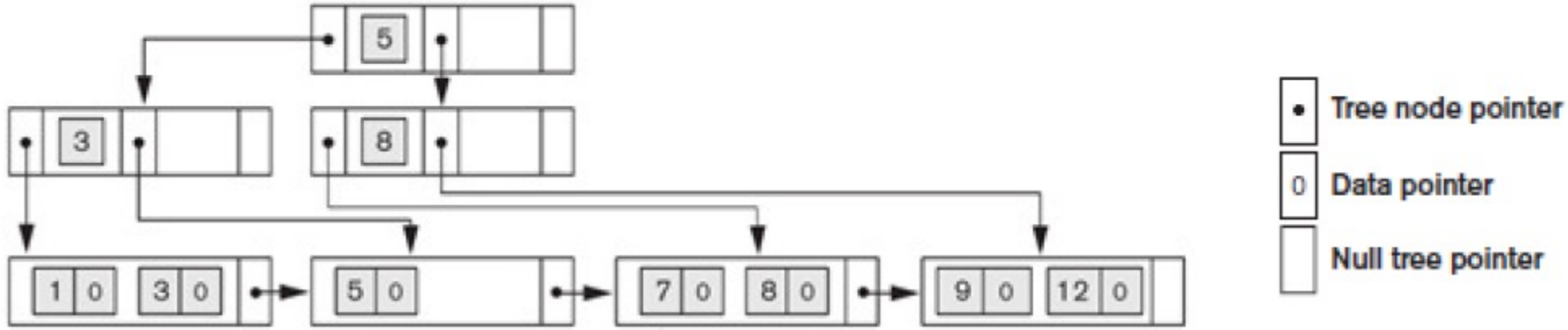
B+ tree index example



Internal node



Leaf node



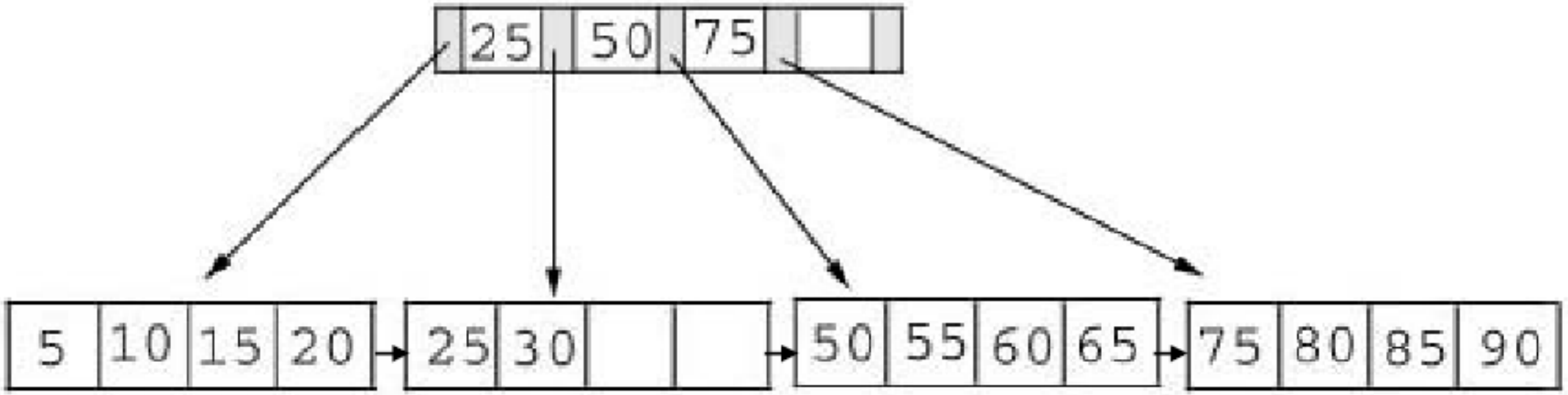
Tree data structure (8)

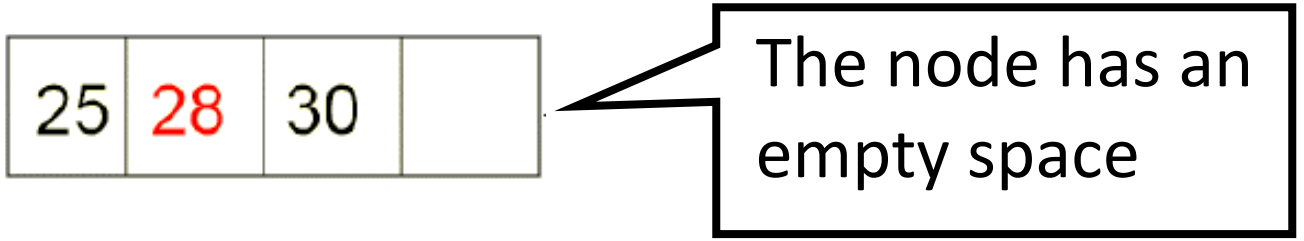
B+ tree index insertion

- **Step 1:** Descend to the leaf node where the key fits
- **Step 2:**
  - ❑ (Case 1): If the node has an empty space, insert the key into the node.
  - ❑ (Case 2) If the node is already full, split it into two nodes by the middle key value, distributing the keys evenly between the two nodes, so each node is half full.
    - (Case 2a) If the node is a leaf, take a copy of the middle key value, and repeat step 2 to insert it into the parent node.
    - (Case 2b) If the node is a non-leaf, exclude the middle key value during the split and repeat step 2 to insert this excluded value into the parent node.

Tree data structure (9)

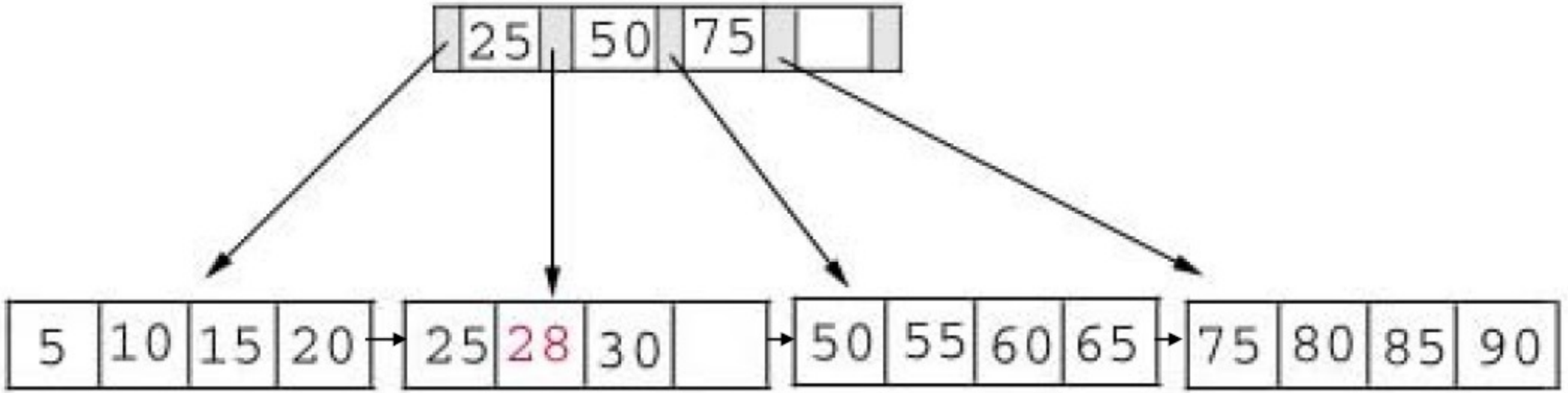
B+ tree index insertion – example 1: insert 28 into the tree below





Tree data structure (10)

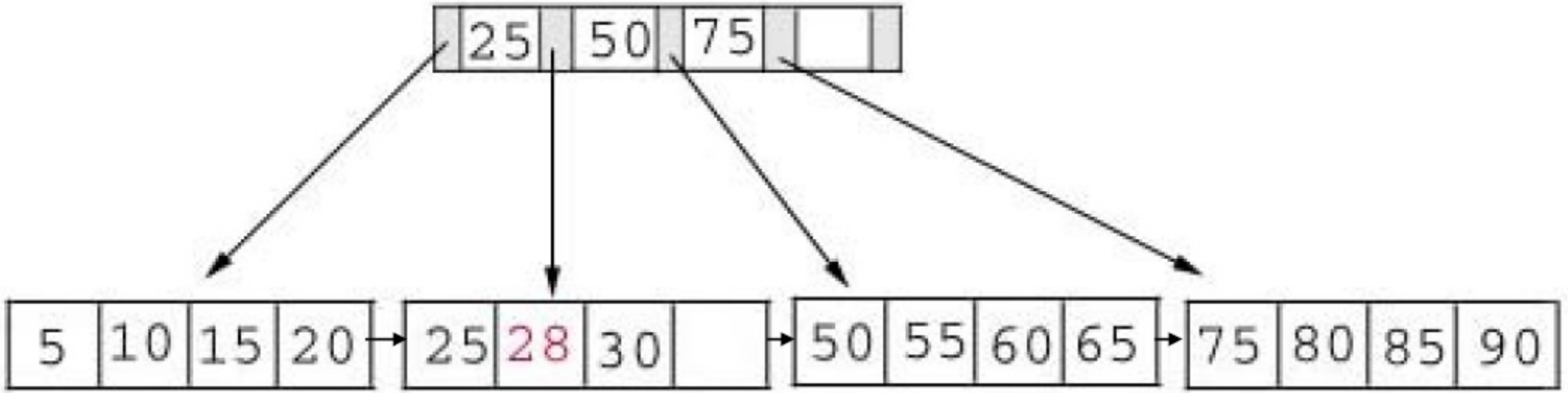
B+ tree index insertion – example 1: insert 28 into the tree below



Insert 28 into the appropriation leaf node.

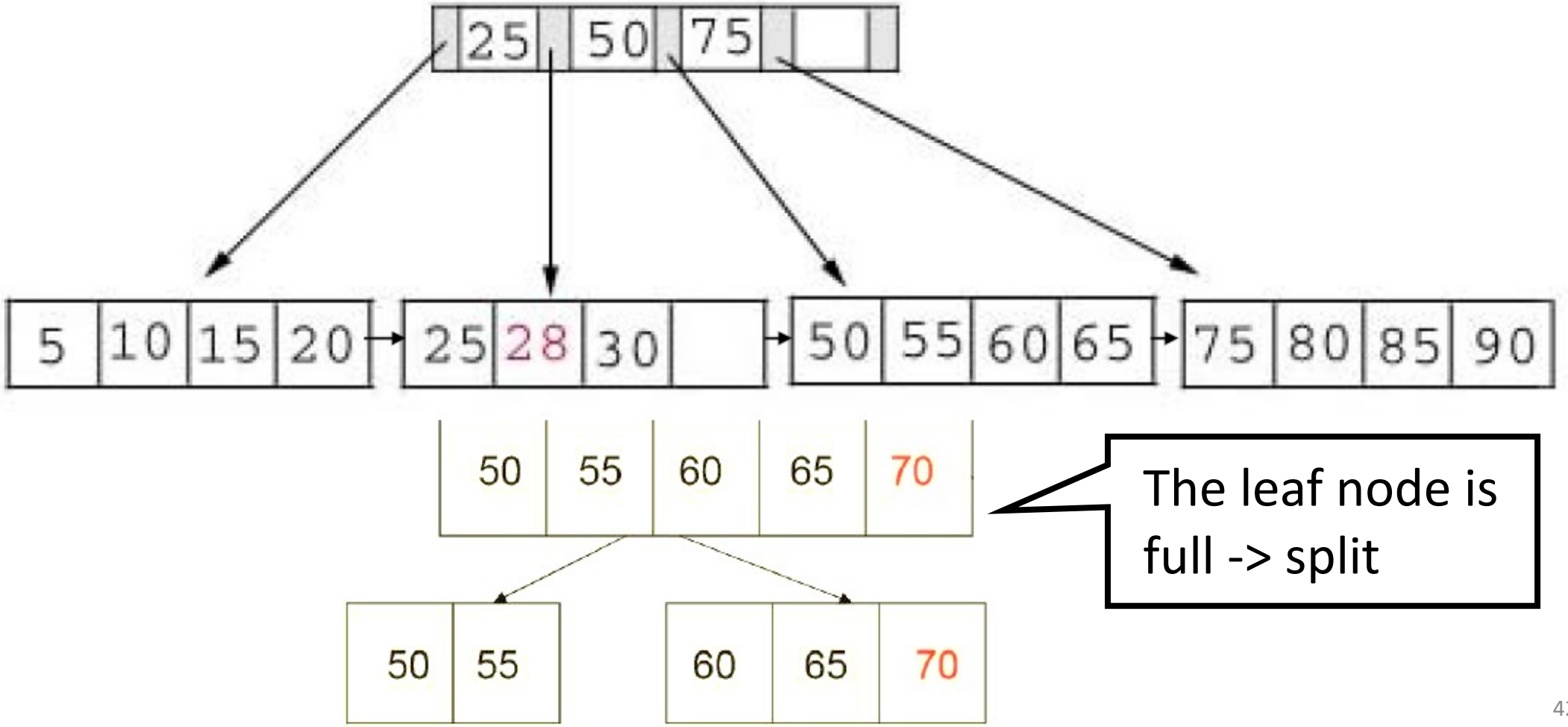
Tree data structure (11)

B+ tree index insertion – example 2: insert 70 into the tree below



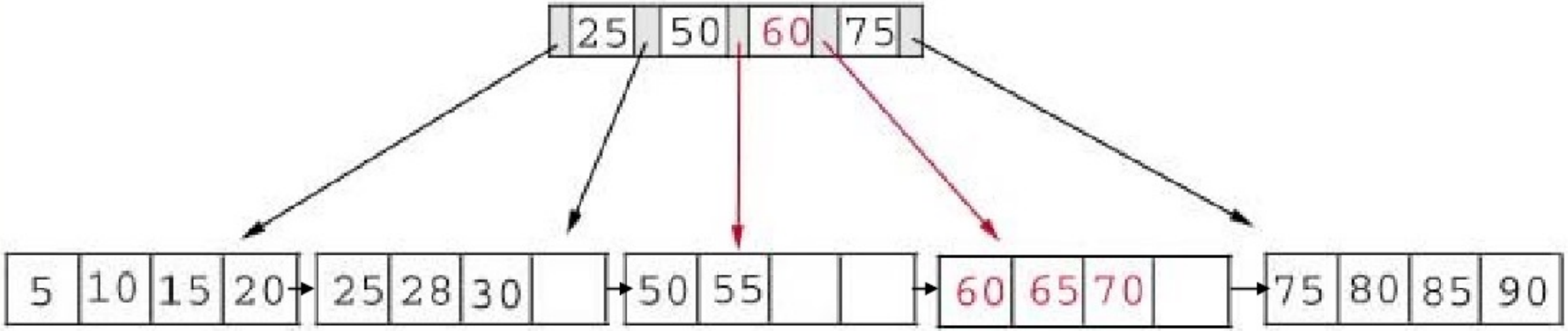
Tree data structure (13)

B+ tree index insertion – example 2: insert 70 into the tree below



Tree data structure (14)

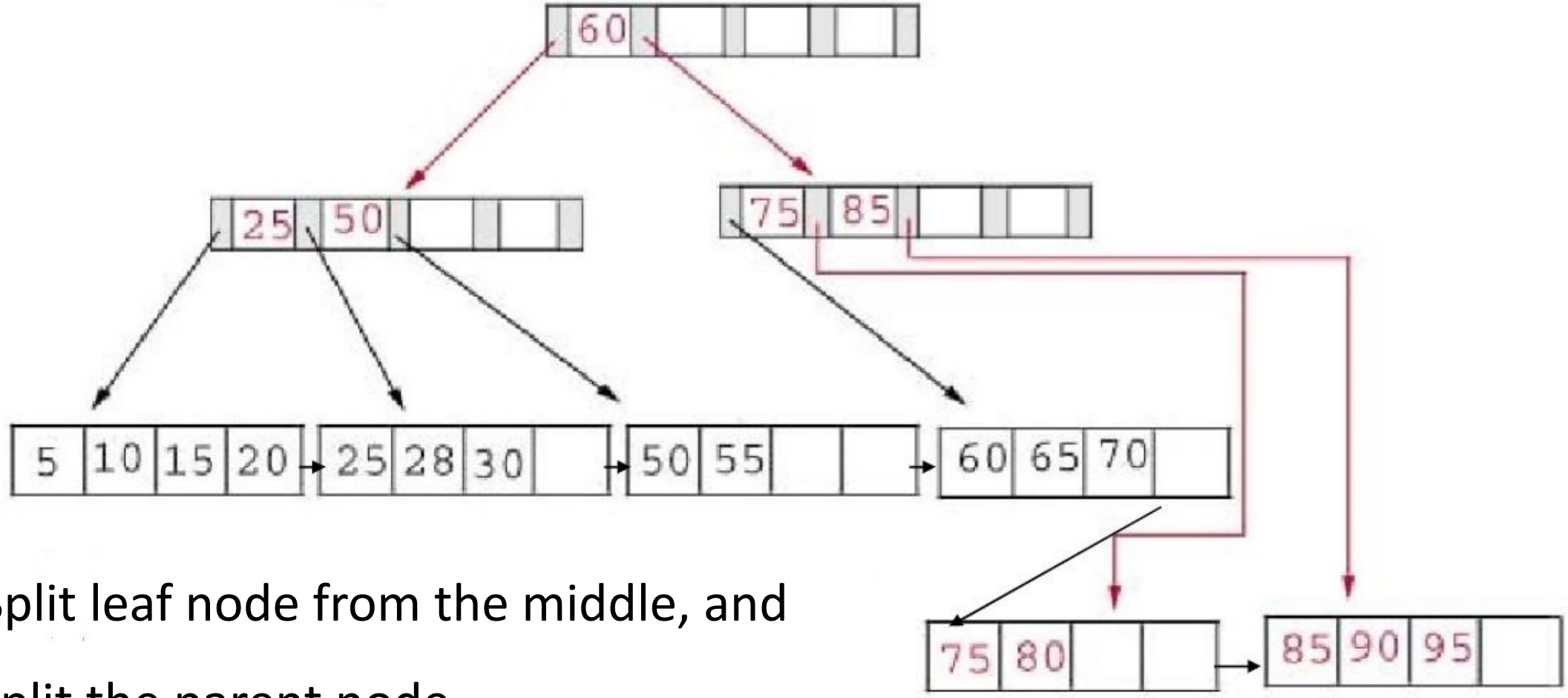
B+ tree index insertion – example 2: insert 70 into the tree below



Split leaf node from the middle, chose the middle key 60, and place it into the parent node.

Tree data structure (15)

B+ tree index insertion – example 3: insert 95 into the tree below



Split leaf node from the middle, and

split the parent node.

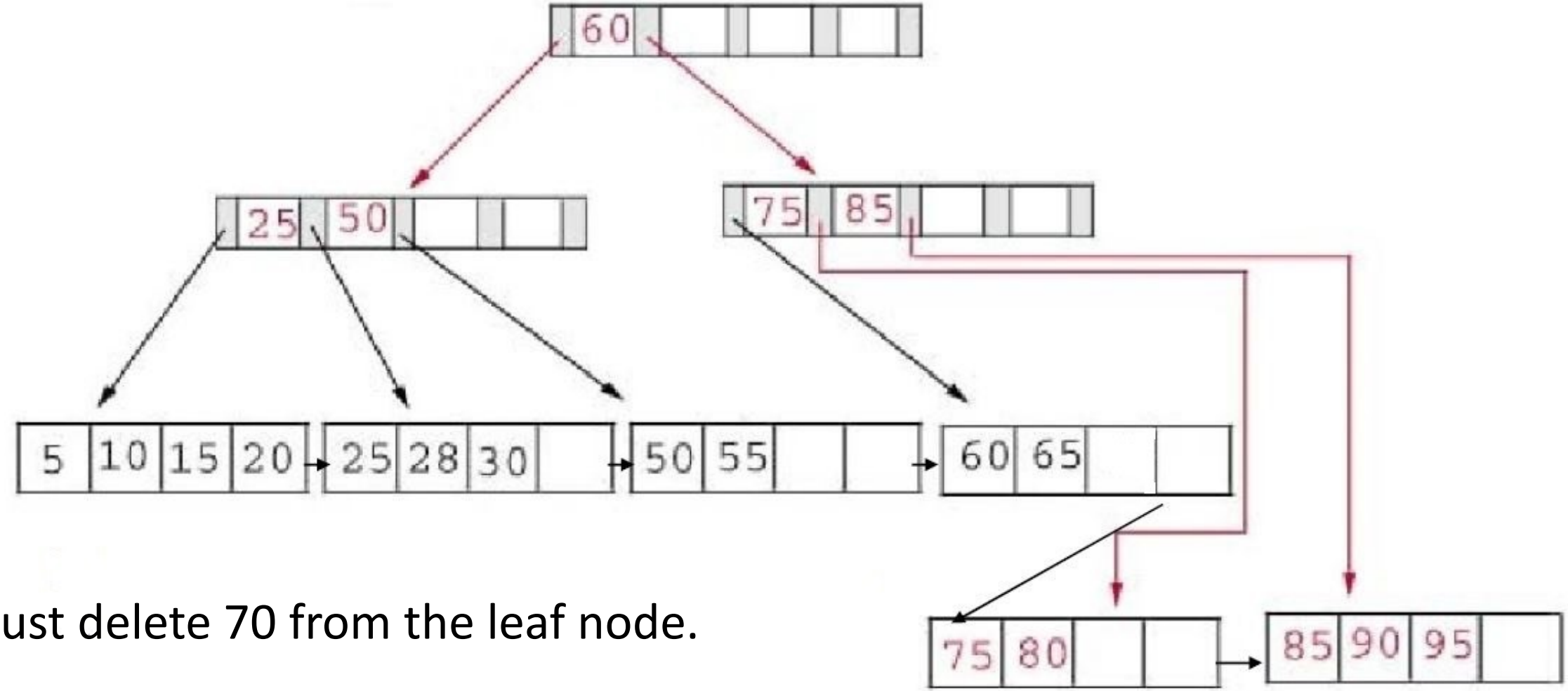
Tree data structure (16)

B+ tree index deletion

- Step1: Descend to the leaf node where the key fits
- Step 2: Remove the required key and associated reference from the node.
  - ❑ (Case 1): If the node still has half-full keys, repair the keys in parent node to reflect the change in child node if necessary and stop.
  - ❑ (Case 2): If the node is less than half-full, but left or right sibling node is more than half-full, redistribute the keys between this node and its sibling. Repair the keys in the level above to represent that these nodes now have a different “split point” between them.
  - ❑ (Case 3): If the node is less than half-full, and left and right sibling node are just half-full, merge the node with its sibling. Repeat step 2 to delete the unnecessary key in its parent.

Tree data structure (17)

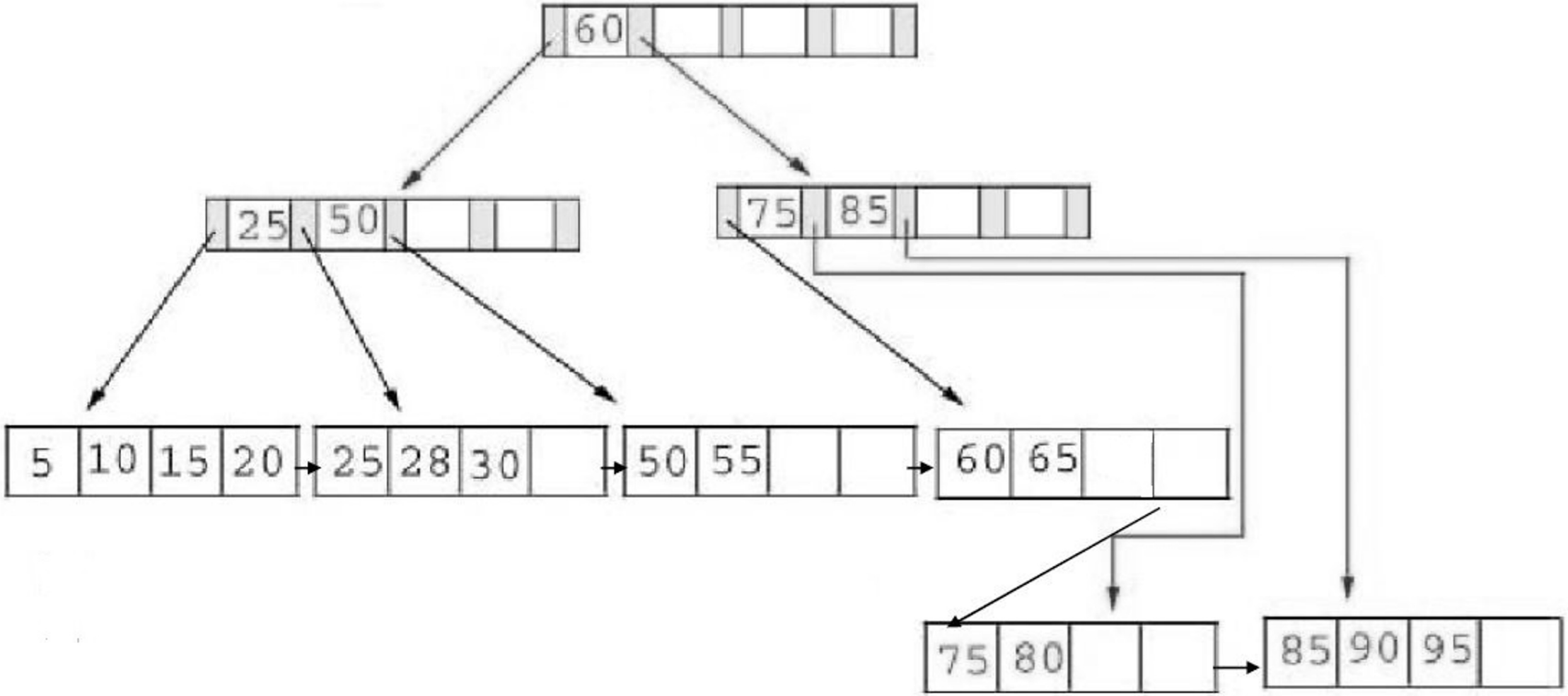
B+ tree index deletion – example 1: delete 70 from the tree below



Just delete 70 from the leaf node.

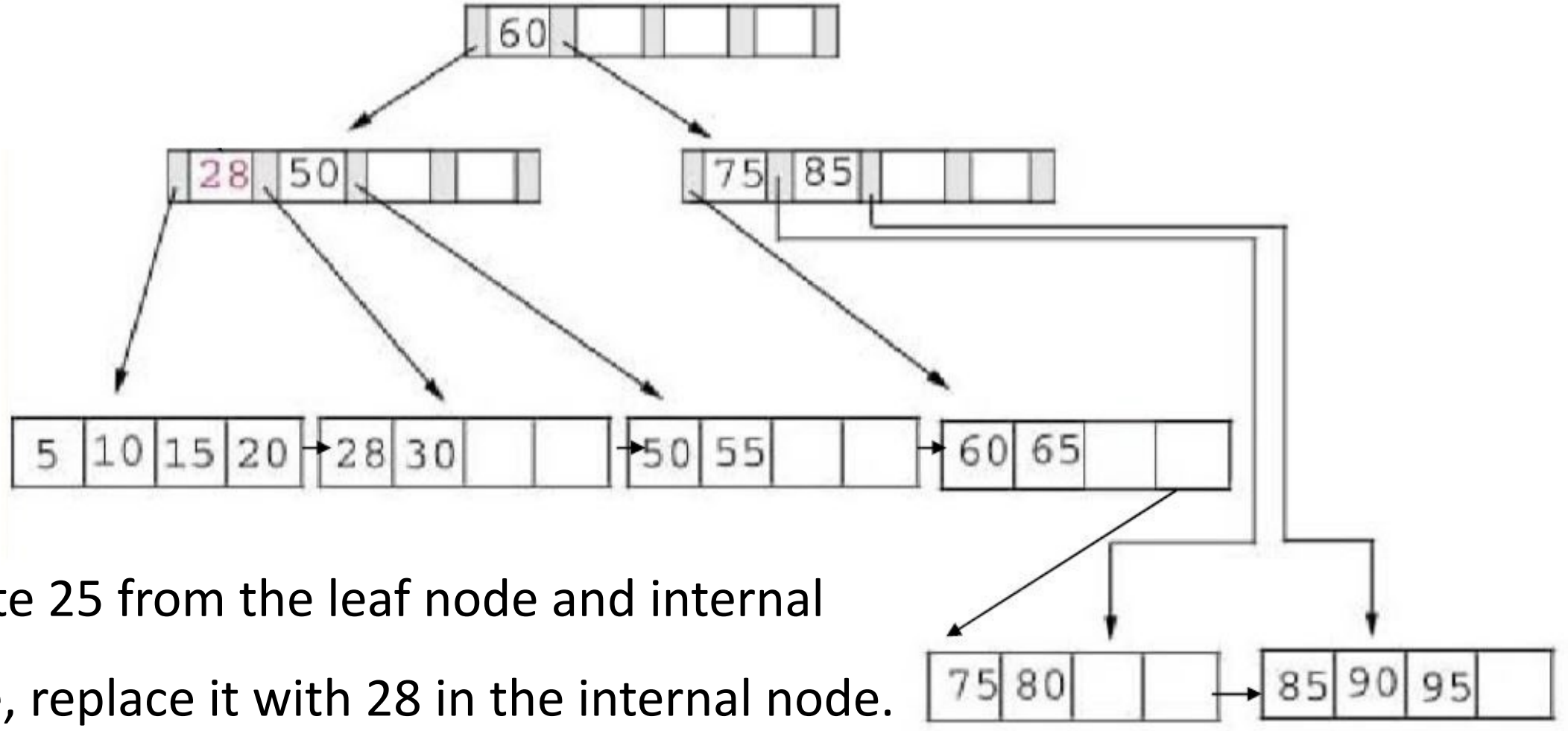
Tree data structure (18)

B+ tree index deletion – example 2: delete 25 from the tree below



Tree data structure (19)

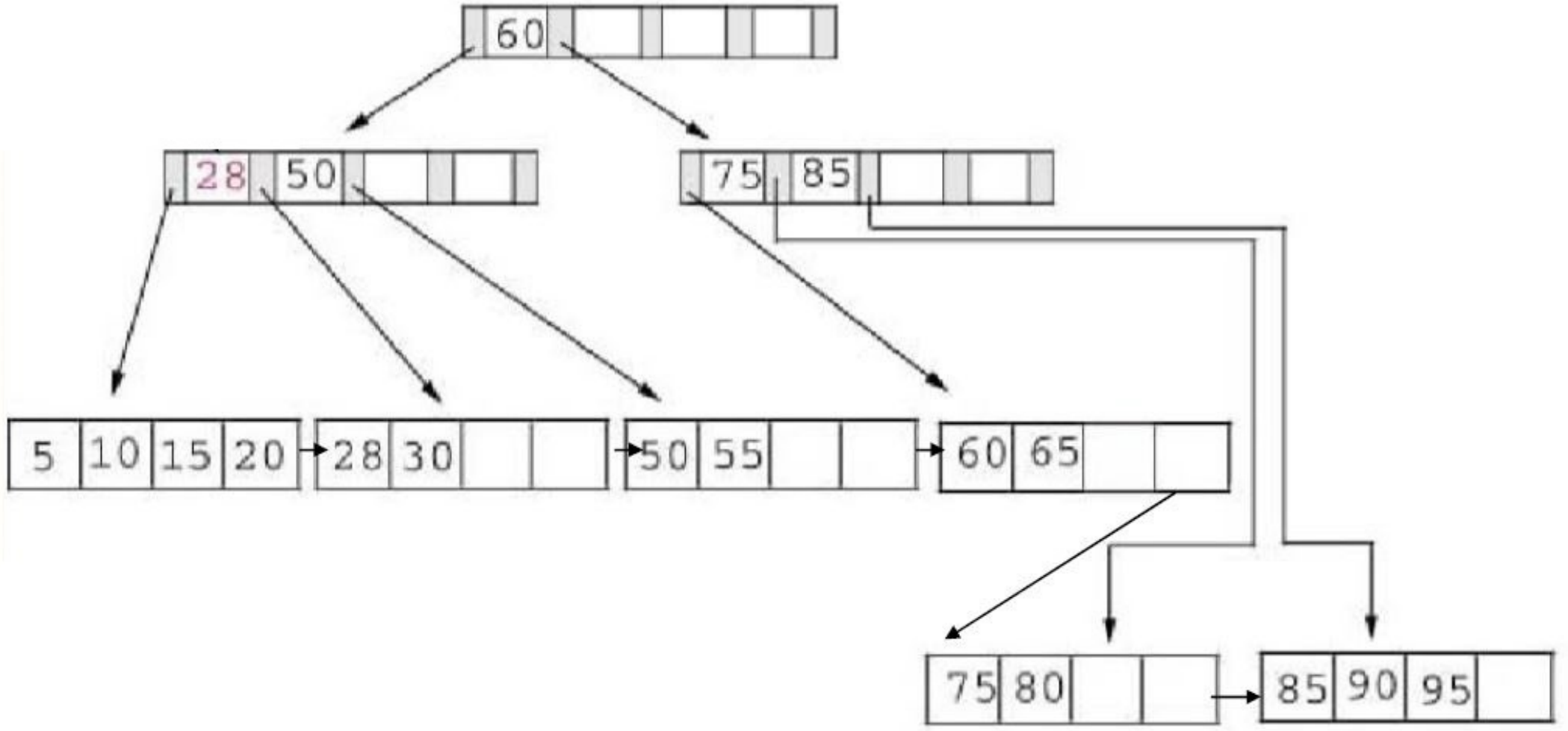
B+ tree index deletion – example 2: delete 25 from the tree below



Delete 25 from the leaf node and internal node, replace it with 28 in the internal node.

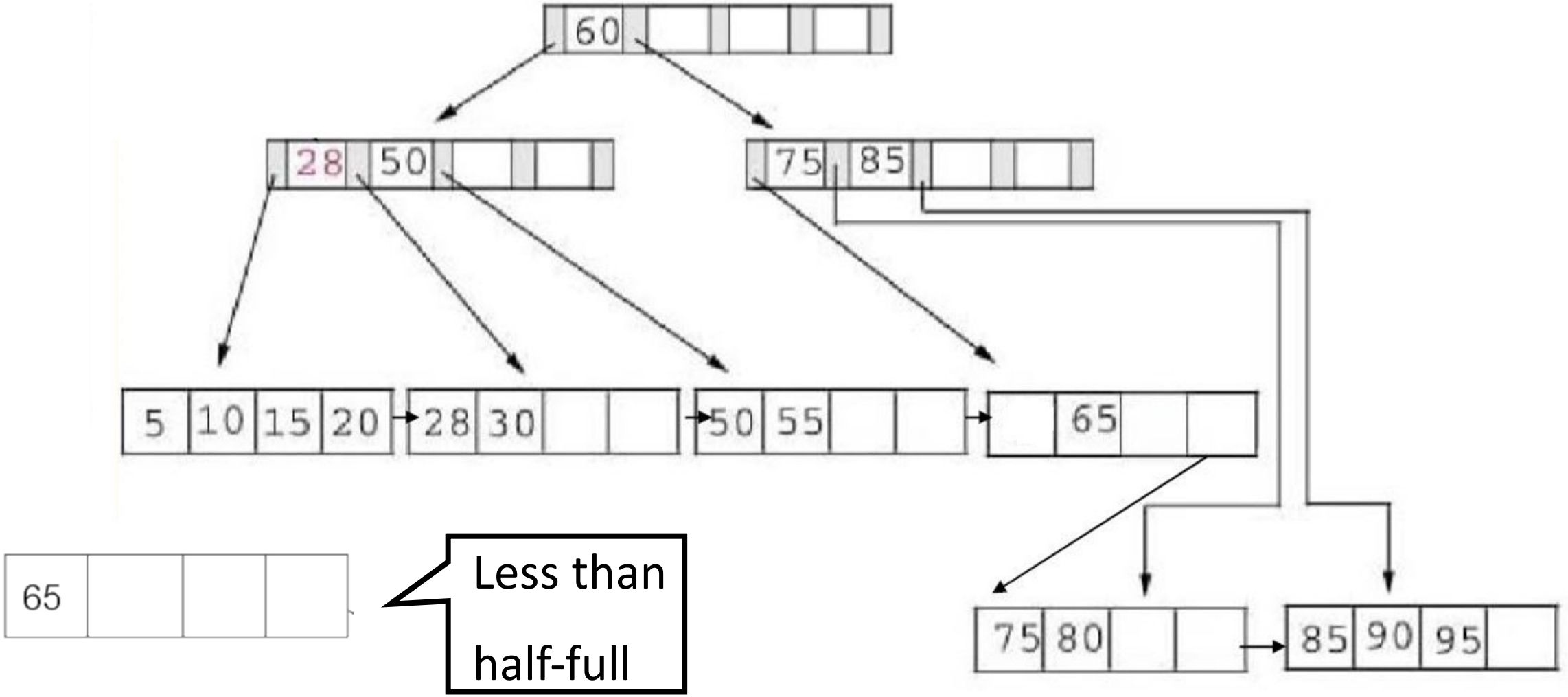
Tree data structure (20)

B+ tree index deletion – example 3: delete 60 from the tree below



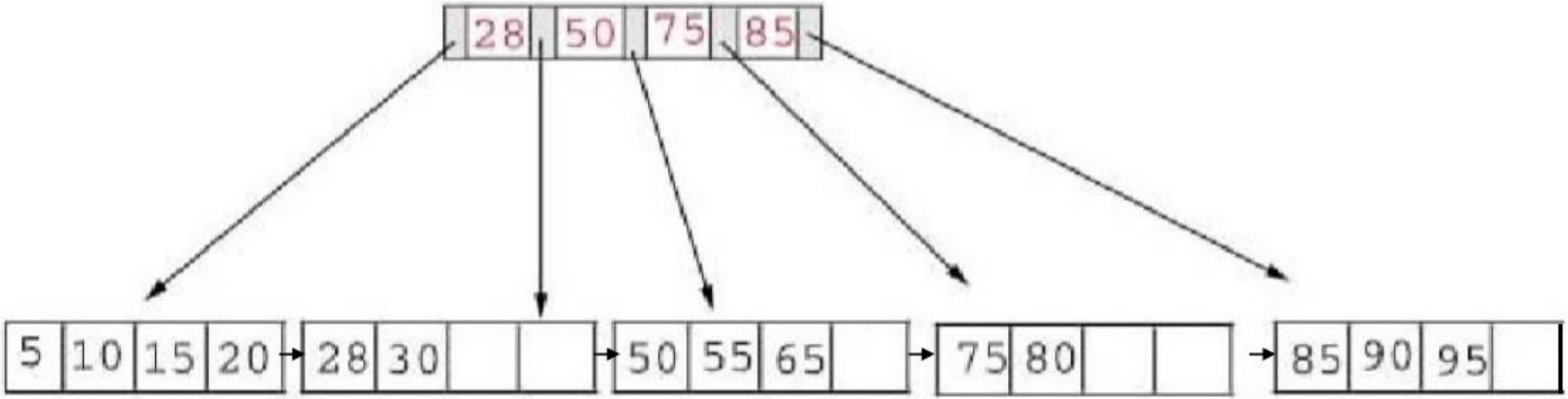
Tree data structure (21)

B+ tree index deletion – example 3: delete 60 from the tree below



Tree data structure (22)

B+ tree index deletion – example 3: delete 60 from the tree below



Delete 60 from the leaf node, combine leaf nodes and then internal nodes.

Thanks for your attention!

Appendix

1. https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html