BKM_DATS: Databázové systémy
10. Indexing and Hashing

Vlastislav Dohnal

Indexing and Hashing

- Basic Concepts
- Ordered Indices
  - B^+-Tree Index
- Static Hashing
  - Dynamic Hashing
- Comparison of Ordered Indexing and Hashing
- Index Definition in SQL

Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library
- **Search Key** – an attribute or a set of attributes used to look up records in a file.
- An **index file** consists of records (called **index entries**) of the form

search-key	pointer

  - Index files are typically much smaller than the original file
- Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in sorted order
  - **Hash indices:** search keys are distributed uniformly across “buckets” using a “hash function”.

Ordered Indices

- In an **ordered index,** index entries are stored sorted on the search key value.
  - E.g., author catalog in library.
- **Primary index:** assume a sequential file, the index whose search key specifies the sequential order of records in the file.
  - Also called **clustering index**
  - The search key of a primary index is usually but not necessarily the primary key.
- **Secondary index**: an index whose search key specifies an order different from the sequential order of records in the file.
  - Also called **non-clustering index**

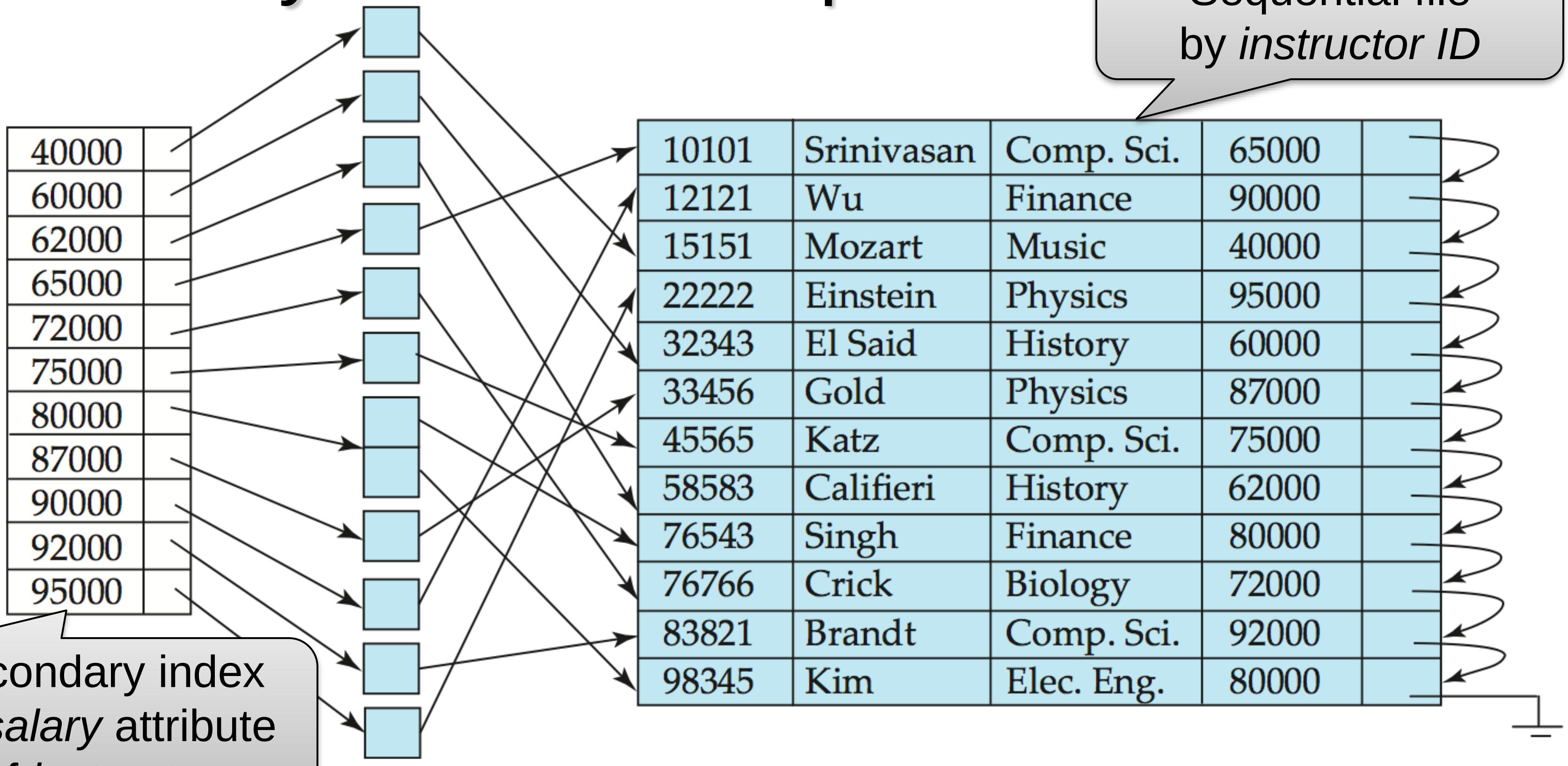
- **Index-sequential file:** sequential file with a primary index.

Secondary Indices

- Frequently, one wants to find all the records whose values in a certain attribute (which is not the search-key of the primary index) satisfy some condition.
  - Example 1:
    - The *instructor* relation stored sequentially by ID
    - We may want to find all instructors in a particular department
  - Example 2:
    - As above
    - We want to find all instructors with a specified salary or with salary in a specified range of values
- We can have a <u>secondary</u> index
  - where an index record exists <u>for each search-key value</u>

Secondary Indices Example

Sequential file
by *instructor ID*



40000	
60000	
62000	
65000	
72000	
75000	
80000	
87000	
90000	
92000	
95000	

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

Secondary index
on *salary* attribute
of *instructor*

- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value.
- Secondary indices <u>have to be dense.</u>

Primary and Secondary Indices

- Indices offer substantial benefits when searching for records.
- BUT: Updating indices imposes overhead on database modification
  - When a file is modified, every index on the file must be updated.
- Sequential scan using primary index is efficient.
- But a sequential scan using a secondary index is expensive.
  - Each record access may fetch a new block from disk
  - Block fetch requires about 5 to 10 milliseconds, versus about 100 nanoseconds for memory access

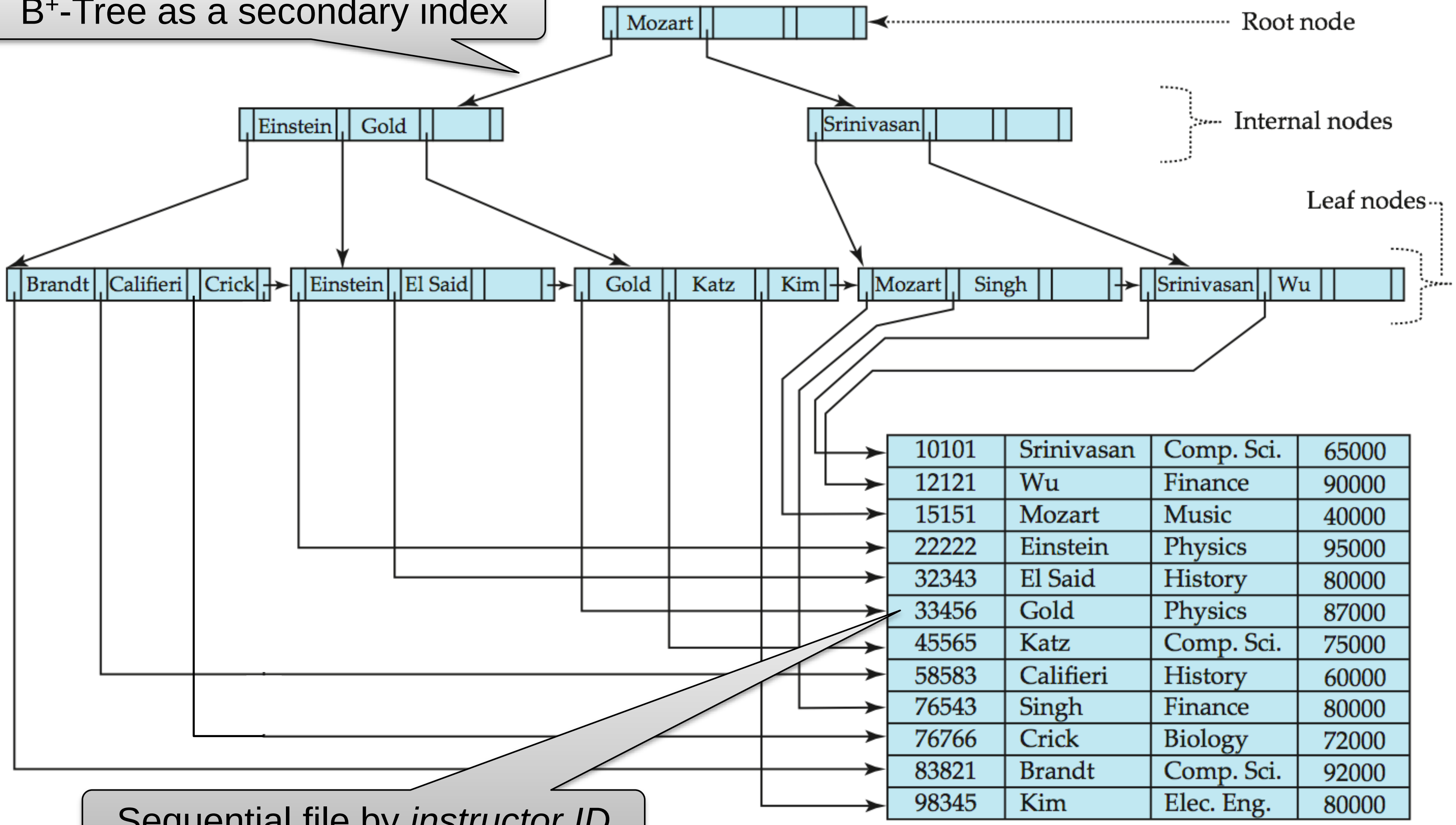
B^+-Tree Index Files

B^+-tree file organization is an alternative to indexed-sequential files.

- Disadvantage of indexed-sequential files
  - Performance degrades as file grows, since many overflow blocks get created.
  - Periodic reorganization of entire file is required.
- Advantage of B^+-tree files:
  - Automatically reorganizes itself with small, local, changes, in the face of insertions and deletions.
  - Reorganization of entire file is not required to maintain performance.
- (Minor) disadvantage of B^+-trees:
  - Extra insertion and deletion overhead, space overhead.
- Advantages of B^+-trees outweigh disadvantages
  - B^+-trees are used extensively

Example of B$^+$-Tree on instructor *name*

B$^+$-Tree as a secondary index



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

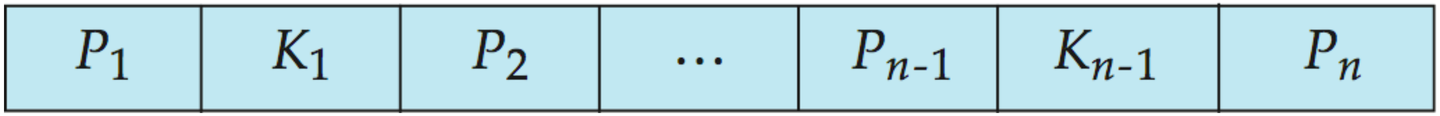
Sequential file by *instructor ID*

B^+-Tree Index

- A B^+-tree is a rooted tree satisfying the following properties:
  - All paths from root to leaf are of the same length
  - Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.
  - A leaf node has between $\lceil (n-1)/2 \rceil$ and $n-1$ values
  - Special cases:
    - If the root is not a leaf, it has at least 2 children.
    - If the root is a leaf (i.e., there are no other nodes in the tree), it can have between 0 and (n–1) values.

B$^+$-Tree Node Structure

- Node structure:



P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

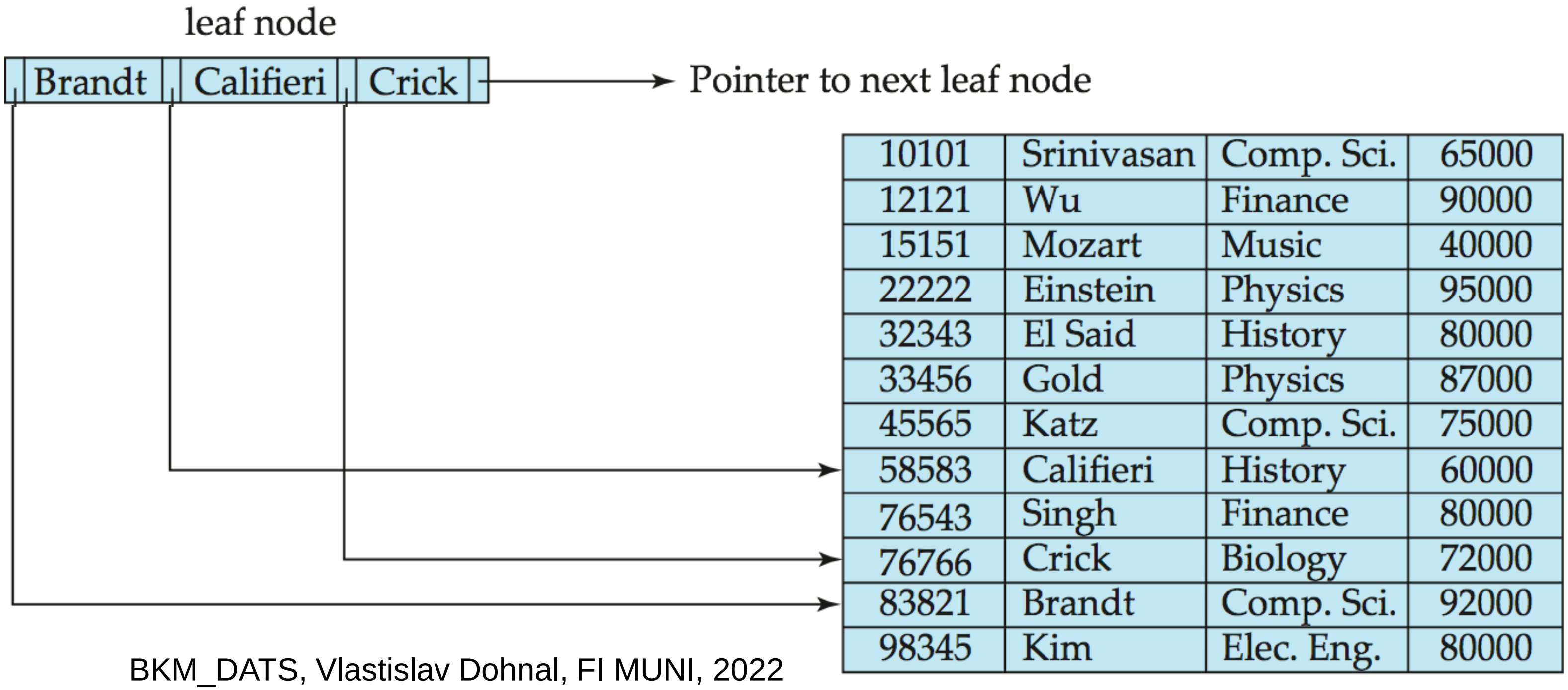
  - K_i are <u>values</u> of the search key
  - P_i are <u>pointers</u> to children (for non-leaf nodes)
    or pointers to records or buckets of records (for leaf nodes).
- The search-key values in a node are <u>ordered</u>

$$K_1 < K_2 < K_3 < . . . < K_{n-1}$$

(We assume no duplicate keys)

Leaf Nodes in B$^+$-Trees

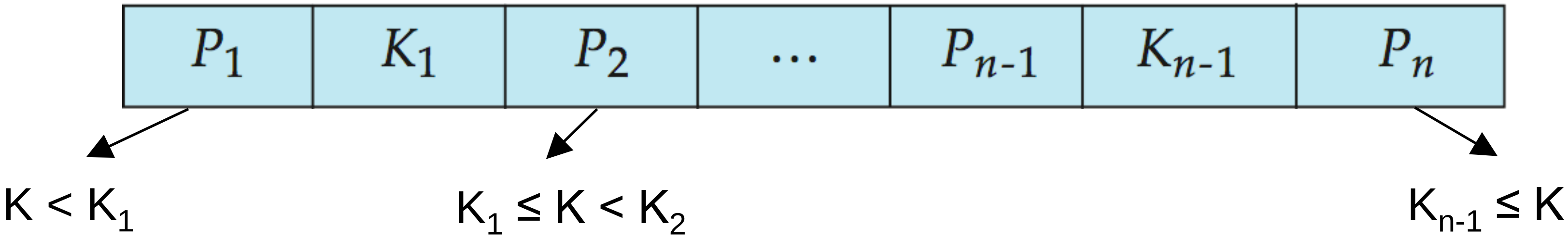
- Properties of a leaf node:
  - For i = 1, 2, . . ., n-1, pointer P_i points to a file record with search-key value K_i,
  - If L_i and L_j are leaf nodes and $i < j$, L_i's search-key values are less than L_j's search-key values
  - P_n points to the next leaf node in search-key order



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Non-Leaf Nodes in B$^+$-Trees

- Non-leaf nodes form a multi-level sparse index on the leaf nodes.
- For a non-leaf node with *n* pointers:
  - All the search-keys *K* in the sub-tree to which P_1 points are less than K_1 $(K < K_1)$
  - For $2 \leq i \leq n-1$, all the search-keys in the sub-tree to which P_i points have values *K* greater than or equal to K_{i-1} and less than K_i $(K_{i-1} \leq K < K_i)$
  - All the search-keys in the sub-tree to which P_n points have values *K* greater than or equal to K_{n-1} $(K_{n-1} \leq K)$

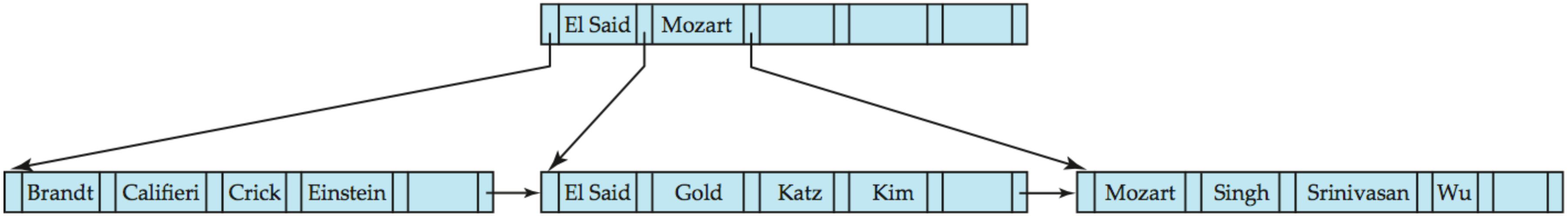


$K < K_1$

$K_1 \leq K < K_2$

$K_{n-1} \leq K$

Example of B^+-tree with n=6



B^+-tree on *name* for *instructor* file ($n = 6$)

- Leaf nodes must have
  - between 3 and 5 values ($\lceil (n-1)/2 \rceil$ and $n-1$, with $n = 6$).
- Non-leaf nodes other than root must have
  - between 3 and 6 children ($\lceil (n/2) \rceil$ and n with $n = 6$).
- Root must have at least 2 children.

Observations about B$^+$-trees

- Since the inter-node connections are done by pointers, “logically” close blocks need not be “physically” close.
- The non-leaf levels of the B$^+$-tree form a hierarchy of sparse indices.
- The B$^+$-tree contains a relatively small number of levels
    - Level below root has at least $2*\lceil n/2 \rceil$ values
    - Next level has at least $2*\lceil n/2 \rceil * \lceil n/2 \rceil$ values
    - etc.
  - If there are m search-key values in the file, the tree height is no more than $\lceil \log_{\lceil n/2 \rceil}(m) \rceil$
  - thus searches can be conducted efficiently.
- Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time (as we shall see).

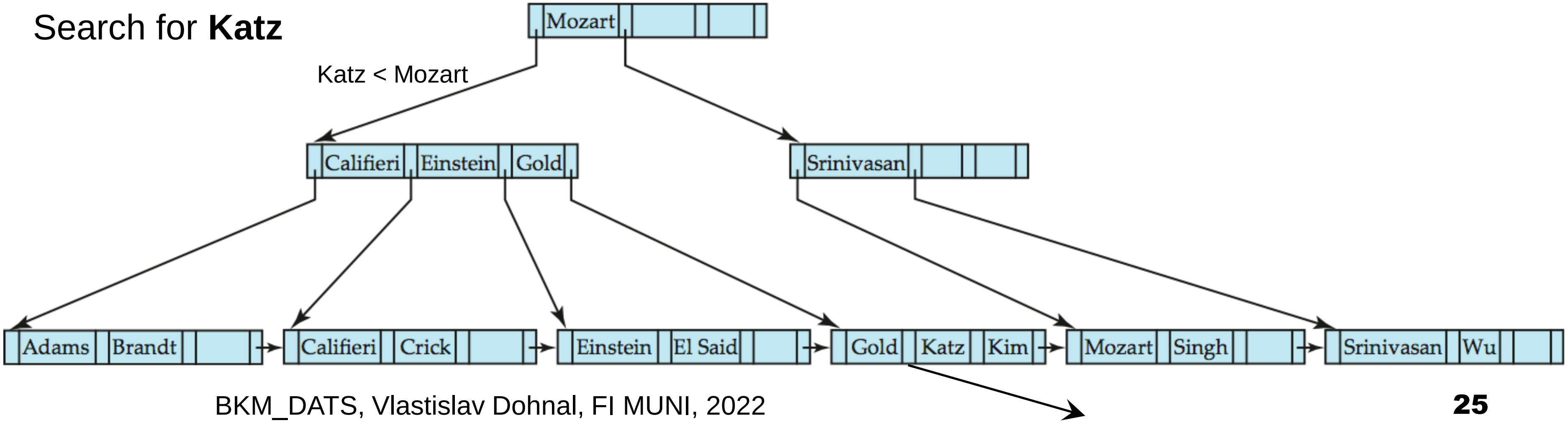
Queries on B^+-Trees

- Find record with a search-key value *V.*
  1. **Set** *C=root*
  2. **While** C is not a leaf node
     1. **Let** *i* be the least value such that $V < K_i$
     2. **If** no such exists, **Let** *i* be index of *last non-null pointer in C*
     3. **Set** *C* = node that $\boldsymbol{P_i}$ *points to*

  // now we are in a leaf node

  1. **Let** *i* be the value such that $K_i = V$
  2. **If** there is such a value *i,* follow pointer P_i to the desired record.
  3. **Else** no record with search-key value *k* exists.

Search for **Katz**



Katz < Mozart

Queries on B^+-Trees (Cont.)

Query evaluation efficiency:

- Tree height is no more than $\lceil \log_{\lceil n/2 \rceil}(m) \rceil$
  - where *m* is the number of search-key values in the file
  - *n* is B^+-tree arity (number of fan-outs)
- A node is generally the same size as a disk block, typically 8 KB
  - and *n* is typically around 200 (40 bytes per index entry).
- With 1 million search key values and $n = 200$
  - at most $log_{100}(1{,}000{,}000) = 3$ nodes are accessed in a lookup.
- Contrast this with a balanced binary tree with 1 million search key values – around 20 nodes are accessed in a lookup
  - above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds

Range (interval) queries:

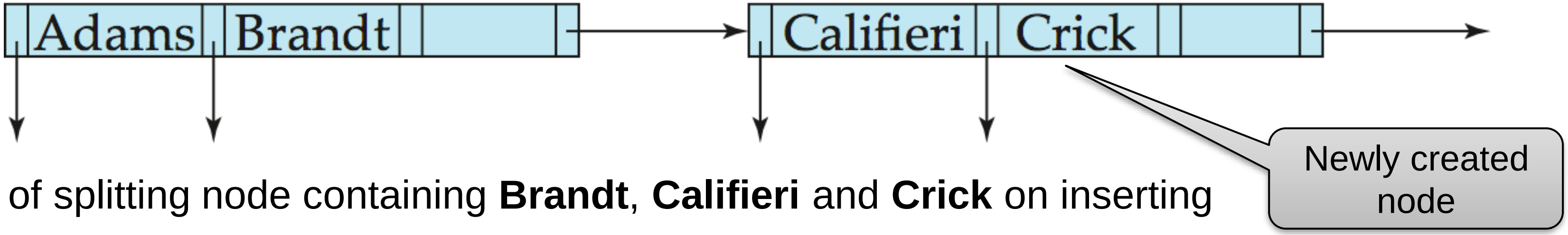
- Look for the lower boundary of the interval
- Use leaf-node chaining to inspect next siblings
  - Stop when a key value greater than the upper boundary is found

Updates on B$^+$-Trees: Insertion

1. Find the leaf node in which the search-key value would appear
    1. (See query algorithm)
2. If the search-key value is already present in the leaf node
    1. Add the new record to the file
    2. If necessary, add a pointer to the bucket, which stores pointers to all records of the same search-key.
3. If the search-key value is not present, then
    1. Add the new record to the file
    2. If there is room in the leaf node,
insert (key-value, pointer) pair in the leaf node
    3. Otherwise, split the node
(along with the new (key-value, pointer) entry)
as discussed in the next slide.

Updates on B$^+$-Trees: Insertion (Cont.)

- Splitting a leaf node:
  - Take the *n* (search-key value, pointer) pairs (including the one being inserted) in sorted order.
    - Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
  - Let the new node be *p,* and let *k* be the least key value in *p.*
    - Insert (*k,p*) in the parent of the node being split.
  - If the parent is full, split it and **propagate** the split further up.
- Splitting of nodes proceeds upwards till a node that is not full, is found.
  - In the worst case, the root node may be split increasing the height of the tree by 1.

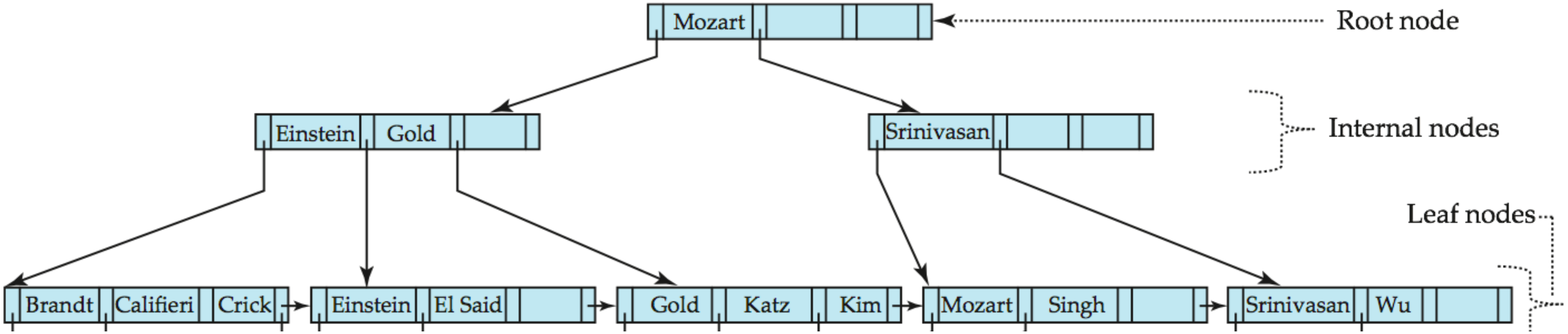


Result of splitting node containing **Brandt**, **Califieri** and **Crick** on inserting **Adams**

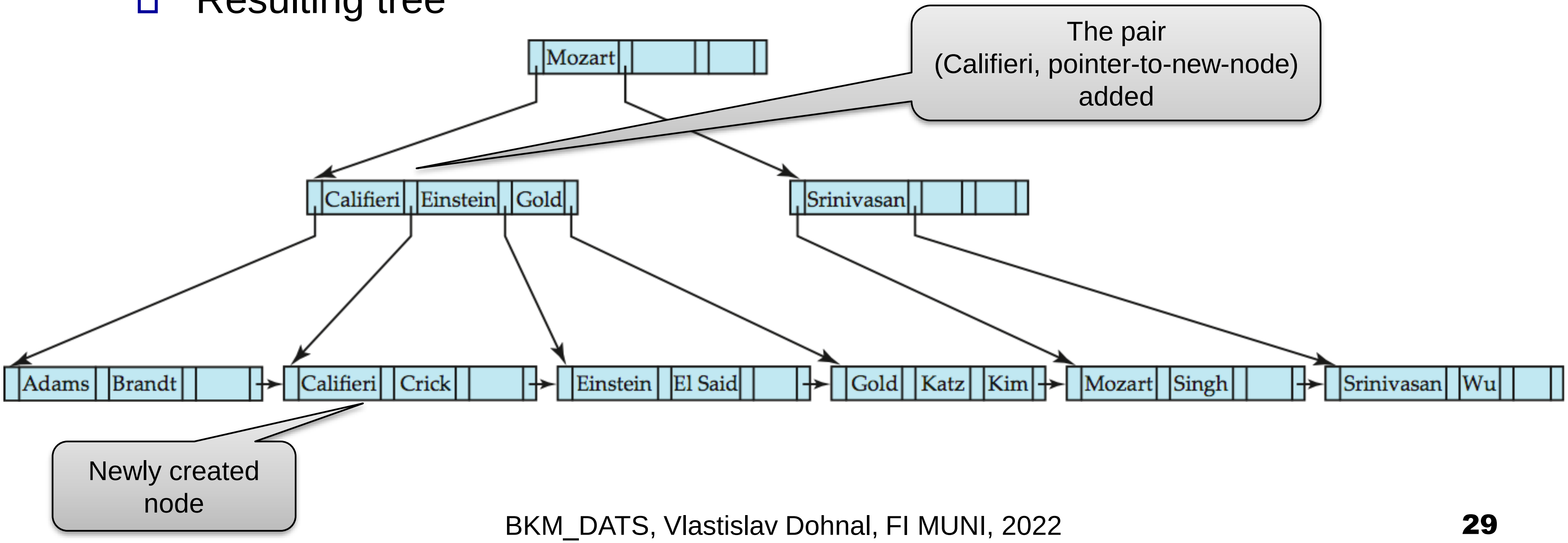
Next step: Insert entry with (Califieri,pointer-to-new-node) into parent

B$^+$-Tree Insertion

- Insert ``Adams''



- Resulting tree

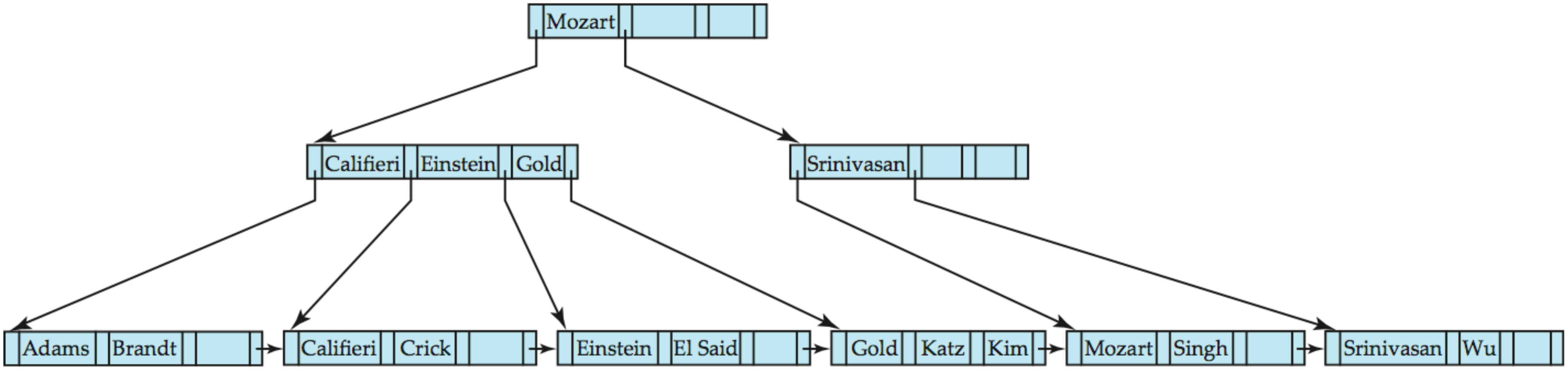


The pair
(Califieri, pointer-to-new-node)
added

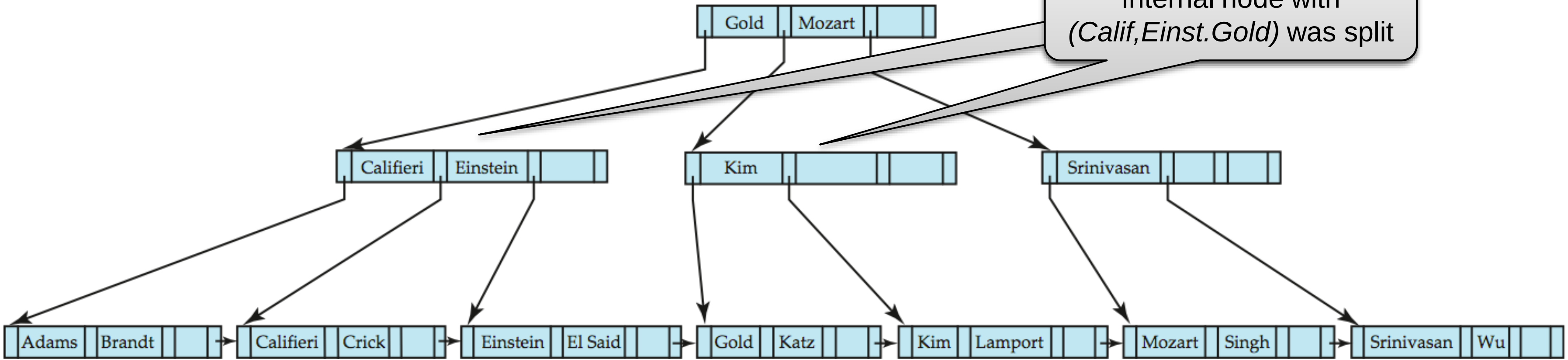
Newly created
node

B$^+$-Tree Insertion

- Insert `` Lamport”



- Resulting tree

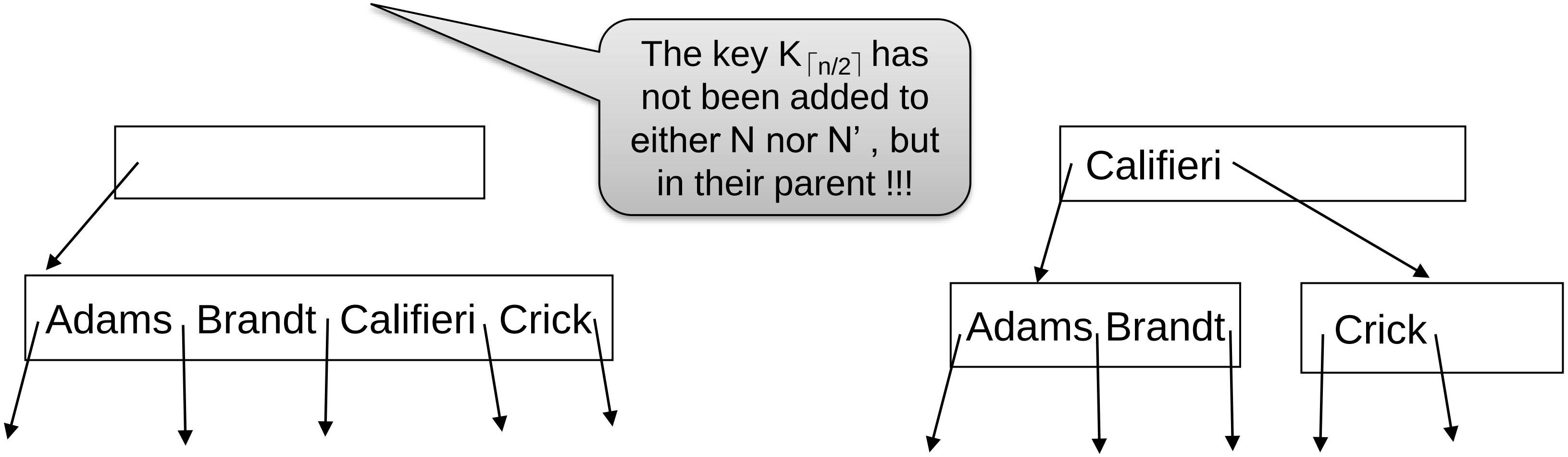


Internal node with *(Calif,Einst.Gold)* was split

Newly created node

Insertion in B^+-Trees (Cont.)

- Splitting a non-leaf node: when inserting (k,p) into an already full internal node N
  - Copy N to an in-memory area M with space for n+1 pointers and n keys
  - Insert (k,p) into M (keep all items sorted!)
  - Copy $P_1, K_1, \ldots, K_{\lceil n/2 \rceil - 1}, P_{\lceil n/2 \rceil}$ from M back into node N
  - Copy $P_{\lceil n/2 \rceil + 1}, K_{\lceil n/2 \rceil + 1}, \ldots, K_n, P_{n+1}$ from M into newly allocated node N'
  - Insert $(K_{\lceil n/2 \rceil}, N')$ into the parent of N



The key $K_{\lceil n/2 \rceil}$ has not been added to either N nor N', but in their parent !!!

Hashing

- In a **hash file organization** we obtain the address of a record directly from its search-key value using a **hash function.**
  - Address is typically a **bucket** – a unit of storage containing one or more records
    - A bucket corresponds to a disk block.

- Hash function *h*
  - a function from the set of all search-key values *K* to the set of all bucket addresses *B.*
  - used to locate records for access, insertion as well as deletion.

- Records with different search-key values may be mapped to the same bucket
  - thus entire bucket has to be searched sequentially to locate a record.

Example of Hash File Organization

- Hash file organization of *instructor* file, using *dept_name* as the key.

bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

bucket 7

Example of Hash File Organization

- Hash function on *dept_name* can be defined as:
  - The binary representation of the i-th character in the alphabet is assumed to be the integer i.
    - E.g. A = 1, B = 2, …
  - The hash function returns the sum of the binary representations of all the characters modulo 8.
    - i.e., there are 8 buckets.
- E.g.
  - h(Music) = 1   (**M**=13, **u**=21, **s**=19, **i**=9, **c**=3 => 65 mod 8 => 1)
  - h(History) = 2
  - h(Physics) = 3
  - h(Elec. Eng.) = 3

Hash Functions

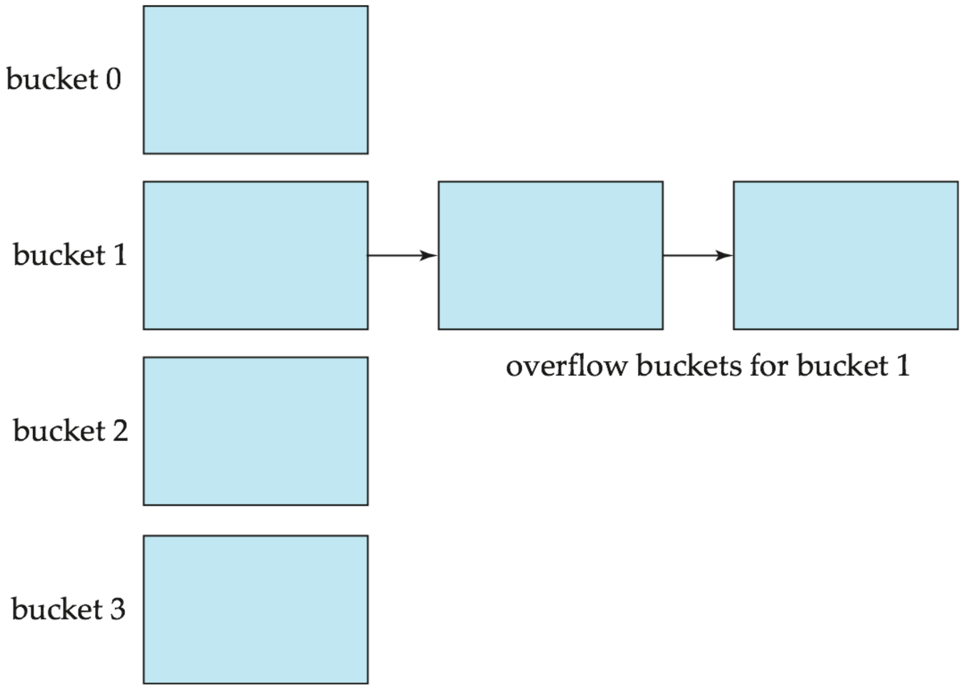
- Worst hash function maps all search-key values to the same bucket
  - this makes access time proportional to the number of search-key values in the file.
- An ideal hash function
  - **uniform** = each bucket is assigned the same number of search-key values from the set of *all* possible values.
  - **random** = each bucket will have the same number of records assigned to it irrespective of the *actual distribution* of search-key values in the file.
- Typical hash functions perform computation on the internal binary representation of the search-key.
  - Example for a numeric search-key:
    - a value V could be multiplied by a prime number and the result module the number of buckets could be returned.

Handling of Bucket Overflows

- **Collision** occurs when two *different search-key* values are hashed to the *same address* (bucket).
- Bucket overflow can occur because of
  - Insufficient bucket size
  - Skew in distribution of records. This can occur due to two reasons:
    - multiple records have the same search-key value
    - chosen hash function produces non-uniform distribution of key values
- Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using
  - **Overflow buckets**
  - **Collision function**

Handling of Bucket Overflows (Cont.)

- **Overflow chaining** – the overflow buckets of a given bucket are chained together in a linked list.
  - This scheme is called **closed hashing.**



- An alternative, called **open hashing**, which does not use overflow buckets, is not suitable for database applications.
  - A **collision function** is defined (it computes an alternative address for storing the record).

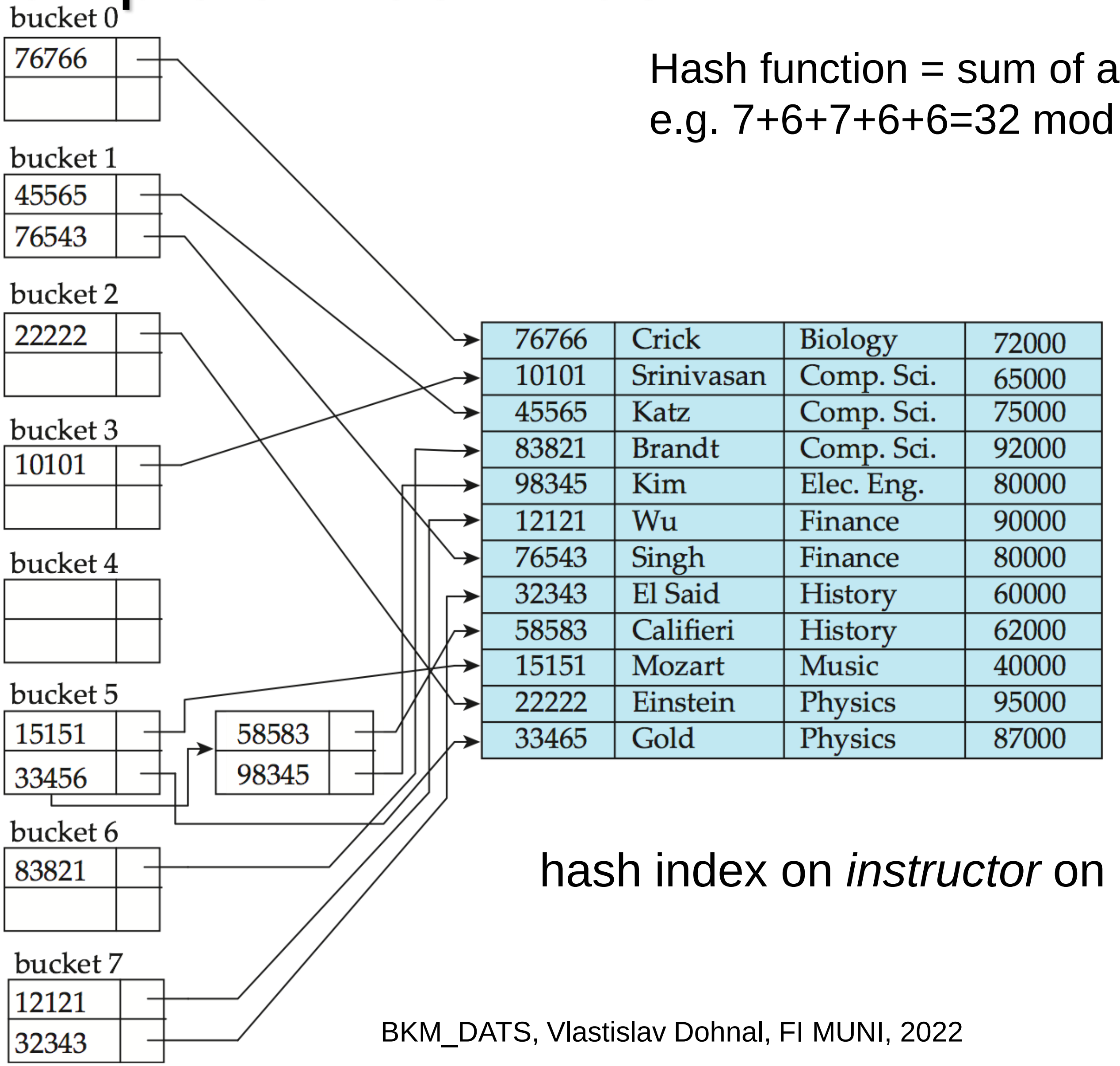
Hash Indices

- Hashing can be used not only for file organization, but also for index-structure creation.
- A **hash index** organizes the search keys, with their associated record pointers, into a hash file structure.
- Strictly speaking, hash indices are always dense indices
  - If the file itself is organized using hashing, a separate primary hash index on it using the same search-key is unnecessary.

- Hash indices are typically used as secondary indices
  - However, we use the term hash index to refer to both secondary index structures and hash organized files.

Example of Hash Index

Hash function = sum of all digits modulo 8
e.g. 7+6+7+6+6=32 mod 8



76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
33465	Gold	Physics	87000

hash index on *instructor* on attribute *ID*

Deficiencies of Hashing

- **Static hashing:** the function *h* maps search-key values to a fixed set of bucket addresses. Databases grow or shrink with time.
  - If initial number of buckets is too small, and file grows, performance will degrade due to too much overflows.
  - If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be under-filled).
  - If database shrinks, again space will be wasted.
- One solution
  - Periodic re-organization of the file with a new hash function
  - Expensive, disrupts normal operations
- Better solution
  - Allow the number of buckets to be modified dynamically.

Dynamic Hashing

- Good for database that grows and shrinks in size
- Allows the hash function to be modified dynamically
- **Dynamic hashing** (subject of course PV062 File Organizations)
  - Allows incremental growth / shrinkage of address space
  - **Extensible hashing**
    - Directory of bucket pointers
  - **Linear hashing**
    - Bucket address space is linearly increased

Comparison of Ordered Indexing and Hashing

- Hashing
  - constant query time
    - constant time to compute address
    - linear time when overflow buckets are present or a collision function defined
      - usually inevitable
  - type of query
    - exact match (records having a specified search-key value)
    - range search – almost impossible
- Indexing
  - logarithmic query time
  - type of query
    - exact match
    - range search (in B^+ trees, very good efficiency)

Comparison of Ordered Indexing and Hashing

- Cost of periodic re-organization
- Relative frequency of insertions and deletions
- Is it desirable to optimize average access time at the expense of worst-case access time?
- In practice:
  - PostgreSQL supports hash indices but only single-column indexes.
    - Values are not stored in the index, but rather their 4-byte hash codes only.
  - Oracle supports a static hash organization but not hash indices.
  - SQLServer supports B$^+$-trees only.

Index Definition in SQL

- Create an index

  create index <index-name> **on** <relation-name>
  (<attribute-list>)

  E.g.: **create index** *branch_index* **on** *branch(branch_name)*

- Drop an index

  drop index <index-name>

- Most database systems allow specification of type of index, and clustering.