ECE30030/ITP30010 Database Systems

Indexes

Chapter 14

Charmgil Hong

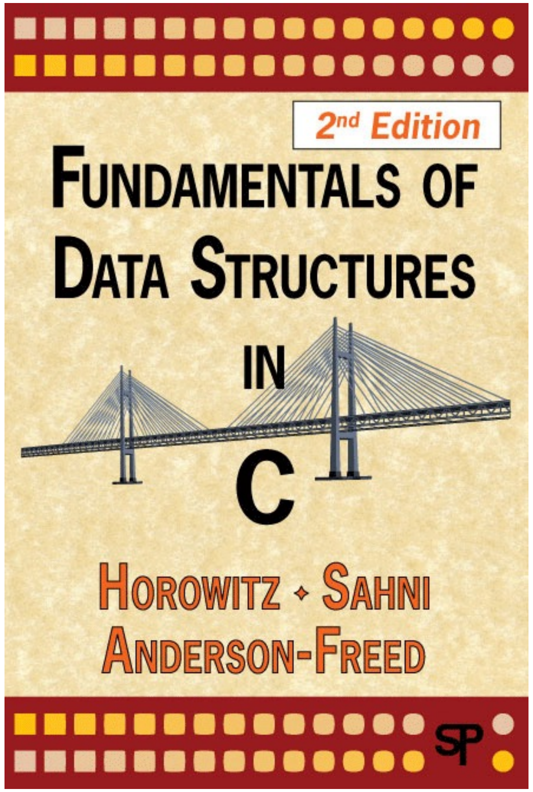
charmgil@handong.edu

Spring, 2025

Handong Global University

Data Structures

- Data structures

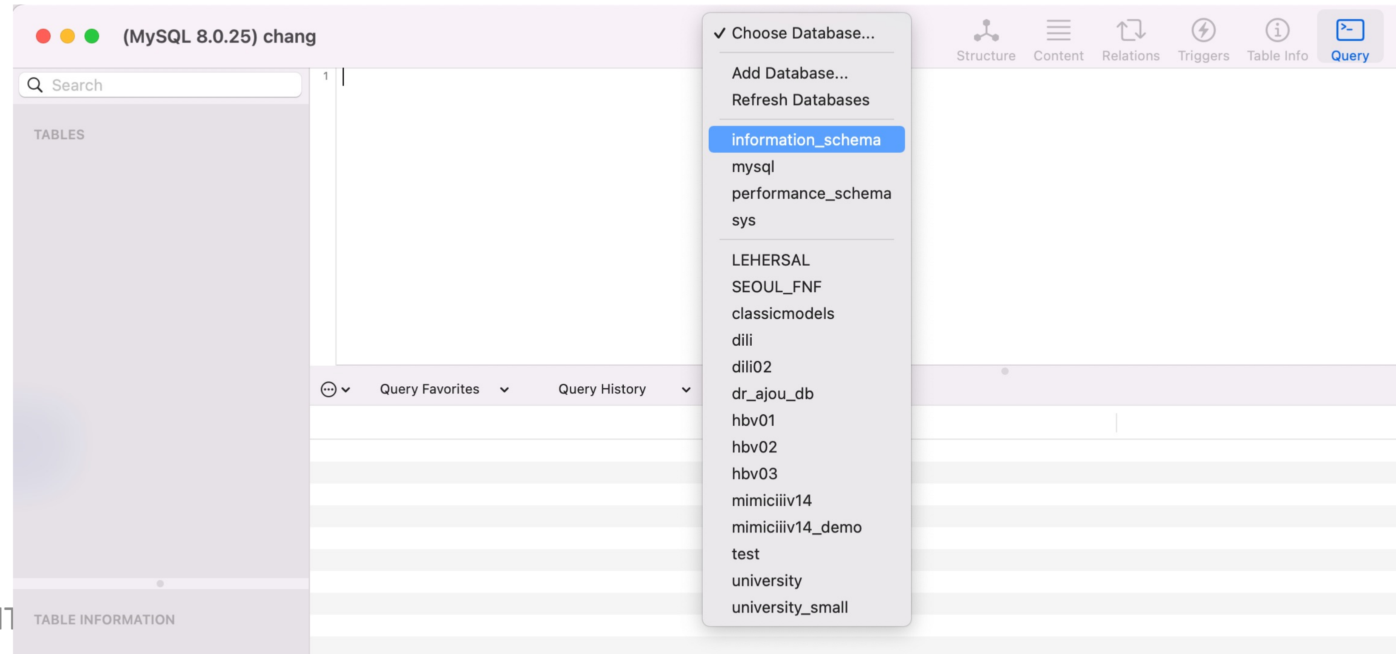


Data Structures in DB Internals

- Data structures are used all around DBMSs for many purposes
  - Internal meta-data
  - Core data storage
  - Temporary data structure
  - Table indexes

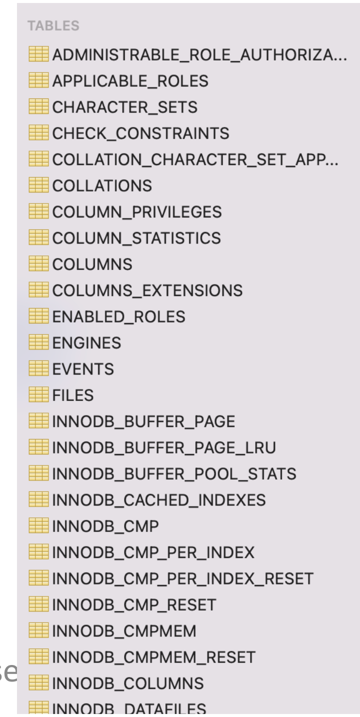
Data Structures in DB Internals

- Data structures are used all around DBMSs for many purposes
  - Internal meta-data
    - Page tables
    - Page directory
    - Page headers, Tuple headers
    - Various page mappings; *e.g.*, page_id to frame, page_id to some allocation on disk (hash tables)
    - …



Data Structures in DB Internals

- Data structures are used all around DBMSs for many purposes
  - Internal meta-data
    - Page tables
    - Page directory
    - Page headers, Tuple headers
    - Various page mappings; *e.g.*, page_id to frame, page_id to some allocation on disk (hash tables)
    - …



Data Structures in DB Internals

- Data structures are used all around DBMSs for many purposes
  - Internal meta-data
    - Page tables
    - Page directory
    - Page headers, Tuple headers
    - Various page mappings; *e.g.*, page_id to frame, page_id to some allocation on disk (hash tables)

- OPTIMIZER_TRACE
- PARAMETERS
- PARTITIONS
- PLUGINS
- PROCESSLIST
- PROFILING
- REFERENTIAL_CONSTRAINTS
- RESOURCE_GROUPS
- ROLE_COLUMN_GRANTS
- ROLE_ROUTINE_GRANTS
- ROLE_TABLE_GRANTS
- ROUTINES
- SCHEMA_PRIVILEGES
- SCHEMATA
- SCHEMATA_EXTENSIONS
- ST_GEOMETRY_COLUMNS
- ST_SPATIAL_REFERENCE_SYSTEMS
- ST_UNITS_OF_MEASURE
- STATISTICS
- TABLE_CONSTRAINTS
- TABLE_CONSTRAINTS_EXTENSIONS
- TABLE_PRIVILEGES
- TABLES
- TABLES_EXTENSIONS
- TABLESPACES
- TABLESPACES_EXTENSIONS
- TRIGGERS
- USER_ATTRIBUTES

TABLE INFORMATION

Filter

TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME	TABLE_TYPE	ENGINE	VERSION	ROW_FORMAT	TABLE_ROWS	AVG_ROW_I
def	university_small	classroom	BASE TABLE	InnoDB	10	Dynamic	5	
def	university_small	department	BASE TABLE	InnoDB	10	Dynamic	7	
def	university_small	course	BASE TABLE	InnoDB	10	Dynamic	13	
def	university_small	instructor	BASE TABLE	InnoDB	10	Dynamic	13	
def	university_small	section	BASE TABLE	InnoDB	10	Dynamic	15	
def	university_small	teaches	BASE TABLE	InnoDB	10	Dynamic	15	
def	university_small	student	BASE TABLE	InnoDB	10	Dynamic	13	
def	university_small	takes	BASE TABLE	InnoDB	10	Dynamic	22	
def	university_small	advisor	BASE TABLE	InnoDB	10	Dynamic	9	
def	university_small	time_slot	BASE TABLE	InnoDB	10	Dynamic	20	
def	university_small	prereq	BASE TABLE	InnoDB	10	Dynamic	7	
def	hbv02	song_sheet2_label_HBsAg_2...	BASE TABLE	InnoDB	10	Dynamic	4249	
def	SEOUL_FNF	scratch	BASE TABLE	InnoDB	10	Dynamic	6	
def	hbv02	song_sheet2_label_drug_2_2...	BASE TABLE	InnoDB	10	Dynamic	20804	
def	hbv02	song_sheet2_tmp_4_220404	BASE TABLE	InnoDB	10	Dynamic	341421	
def	hbv02	song_sheet2_label_HBeAg_2...	BASE TABLE	InnoDB	10	Dynamic	8568	
def	LEHERSAL	ADMISSIONS	BASE TABLE	InnoDB	10	Dynamic	129	
def	LEHERSAL	ICUSTAYS	BASE TABLE	InnoDB	10	Dynamic	136	
def	LEHERSAL	CHARTEVENTS	BASE TABLE	InnoDB	10	Dynamic	756434	
def	LEHERSAL	PATIENTS	BASE TABLE	InnoDB	10	Dynamic	100	
def	test	movies	BASE TABLE	InnoDB	10	Dynamic	0	
def	LEHERSAL	ICUSTAY_DETAIL	BASE TABLE	InnoDB	10	Dynamic	135	
def	hbv02	song_sheet2_dataset_HBV_D...	BASE TABLE	InnoDB	10	Dynamic	20916	
def	hbv02	song_sheet2_dataset_HBeAg...	BASE TABLE	InnoDB	10	Dynamic	8784	
def	hbv02	song_sheet2_dataset_HBsAg...	BASE TABLE	InnoDB	10	Dynamic	4335	
def	hbv02	song_sheet2_dataset_HBsAg...	BASE TABLE	InnoDB	10	Dynamic	1269	
def	dr_ajou_db	SW1_SHEET2_1Y_BODY_MEA...	BASE TABLE	InnoDB	10	Dynamic	45210	

Data Structures in DB Internals

- Data structures are used all around DBMSs for many purposes
  - Core data storage (the database itself)
    - Heaps of pages
    - Hash tables, b+tree or other types of trees
    - Memory cache (a huge hash table)
    - …

```
root@cheonmaji:/var/lib/mysql# ls
auto.cnf        ca-key.pem          ib_logfile0         sakila
binlog.000002   ca.pem              ib_logfile1         server-cert.pem
binlog.000003   classicmodels       ibtmp1              server-key.pem
binlog.000004   client-cert.pem     #innodb_temp        sys
binlog.000005   client-key.pem      mysql               temp
binlog.000006   employees           mysql.ibd           undo_001
binlog.000007   foodmart            mysql_upgrade_info  undo_002
binlog.000008   #ib_16384_0.dblwr   peace.pid           university
binlog.000009   #ib_16384_1.dblwr   performance_schema  university_small
binlog.000010   ib_buffer_pool      private_key.pem     world
binlog.index    ibdata1             public_key.pem      world_x
```

Data Structures in DB Internals

- Data structures are used all around DBMSs for many purposes
  - Temporary data structures
    - Created and used when executing queries
  - Table indexes
    - Building glossary of keys inside of tuples for quick look-ups

Data Structures in DB Internals

- Data structures are used all around DBMSs for many purposes
  - Internal meta-data
  - Core data storage
  - Temporary data structure
  - Table indexes

- *For the first three, hash tables may be enough*
  - Hash table: good for point query look-ups (single key look-ups)
    - Used in many places in DBMS; *e.g.*,
      - Meta-data
      - Storing *n*-ary tables
      - Temporary data structures
  - Hash tables cannot serve table indexes well

Agenda

- Database indexes
- Ordered indexes
  - Indexed-sequential files
  - B+tree index files
    - B+tree nodes
    - Queries on B+trees
    - B+tree operations
- Hash indexes (*will not cover*)
- Inverted index

Indexes in Books

- At the end of our textbook…
  - Complements the table of contents by enabling access to information by specific subject

Index

Contents

Indexes in Databases

- Indexing mechanisms are to speed up access to desired data
  - Improves the speed of data retrieval operations on a database table *at the cost of additional writes and storage space*
  - Replica of a subset of table attributes
    - An auxiliary data structure that enables more efficient traverse and search than sequential scans

Indexes in Databases

- A motivating example:
  SELECT * **FROM** *Instructor* **WHERE** *name* = 'Katz'
  - DBMS literally have to look at every single row of the *Instructor* table to see if the name matches 'Katz'
  - The purpose of having an index is to speed up search queries by cutting down the number of records/rows in a table that need to be examined

Instructor relation

ID	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	65000.00
12121	Wu	Finance	90000.00
15151	Mozart	Music	40000.00
22222	Einstein	Physics	95000.00
32343	El Said	History	60000.00
33456	Gold	Physics	87000.00
45565	Katz	Comp. Sci.	75000.00
58583	Califieri	History	62000.00
76543	Singh	Finance	80000.00
76766	Crick	Biology	72000.00
83821	Brandt	Comp. Sci.	92000.00
98345	Kim	Elec. Eng.	80000.00

Example

- Index in action
  - **SELECT** ICUSTAY_ID, DRUG, DOSE_VAL_RX, DOSE_UNIT_RX, ROUTE
    FROM PRESCRIPTIONS P
    WHERE P.DRUG **LIKE** 'amoxicillin%' **OR** P.DRUG **LIKE** 'cefazolin';

	ICUSTAY_ID	DRUG	DOSE_VAL_RX	DOSE_UNIT_RX	ROUTE
1	[redacted]	Amoxicillin-Clavulanic Acid	250	mg	PO
2	[redacted]	Amoxicillin	1000	mg	PO
3	<null>	Amoxicillin-Clavulanic Acid	500	mg	PO
4	<null>	CefazoLIN	2	g	IV
5	[redacted]	CefazoLIN	2	g	IV
6	[redacted]	Cefazolin	2	gm	IV
7	<null>	CefazoLIN	2	g	IV

```
[2021-05-22 22:43:43] 500 rows retrieved starting from 1 in 3 s 935 ms
(execution: 3 s 841 ms, fetching: 94 ms)
```

Example

- Index in action
  - **CREATE INDEX** PRESCRIPTIONS_idx04 **ON** PRESCRIPTIONS (DRUG);

  - **SELECT** ICUSTAY_ID, DRUG, DOSE_VAL_RX, DOSE_UNIT_RX, ROUTE
    FROM PRESCRIPTIONS P
    WHERE P.DRUG **LIKE** 'amoxicillin%' **OR** P.DRUG **LIKE** 'cefazolin';

	ICUSTAY_ID	DRUG	DOSE_VAL_RX	DOSE_UNIT_RX	ROUTE
1	[redacted]	Amoxicillin-Clavulanic Acid	250	mg	PO
2	[redacted]	Amoxicillin	1000	mg	PO
3	<null>	Amoxicillin-Clavulanic Acid	500	mg	PO
4	<null>	CefazoLIN	2	g	IV
5	[redacted]	CefazoLIN	2	g	IV
6	[redacted]	Cefazolin	2	gm	IV
7	<null>	CefazoLIN	2	g	IV

```
[2021-05-22 22:53:41] 500 rows retrieved starting from 1 in 410 ms
(execution: 371 ms, fetching: 39 ms)
```

Table Indexes

- Access types supported efficiently
  - Records with a specified value in the attribute
  - Records with an attribute value falling in a specified range of values

- Evaluation metrics
  - Access time
  - Insertion time
  - Deletion time
  - Space overhead

Table Indexes

- Basic concepts
  - Search-key: An attribute or set of attributes to look up records in a file
    - Here ”key” differs from that used in primary key, candidate key, superkey, …
    - Rather close to the “key” in the hash tables
  - An index file consists of records (index entries) that form:

search-key	pointer

    - Index files are typically much smaller than the original table

- Two basic kinds of indexes:
  - Ordered indexes: search-keys are stored in sorted order
  - Hash indexes: search-keys are distributed uniformly across “buckets” using a hash function(s)

Table Indexes

- A table may have multiple indexes
  - When the user execute queries, the DBMS figures out the best index(es) for each query
  - Trade-off: performance (query optimization) vs. the number of indexes to create per database
    - *Search overhead* vs. *Storage & maintenance overhead*

- The DBMS ensures that the contents of the table and the index are logically in sync
  - When an attribute changes, the change is also applied to the index
  - DBMSs are responsible for maintaining indexes and keep them synchronized with the underlying table

Agenda

- Database indexes
- **Ordered indexes**
  - Indexed-sequential files
  - B+tree index files
    - B+tree nodes
    - Queries on B+trees
    - B+tree operations
- Hash indexes (*will not cover*)
- Inverted index

Ordered Indexes

- Two types:
  - Indexed-Sequential Files
  - B+tree index files

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

Indexed-Sequential Files

- In an ordered index, index entries are stored sorted on the search-key values

- Few vocabs
  - Dense index: Index record appears for every search-key value in the file
  - Sparse index: Contains index records for only some search-key values

  - Clustering index: the index whose search-key specifies the sequential order of the file
    - = Primary index
    - Usually (but not always) the primary key
  - Non-clustering index: an index whose search-key specifies an order different from the sequential order of the file
    - = Secondary index

Indexed-Sequential Files

- Example: dense index on the *ID* attribute of the *instructor* relation

10101	
12121	
15151	
22222	
32343	
33456	
45565	
58583	
76543	
76766	
83821	
98345	

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	

Indexed-Sequential Files

- Example: dense index on the *dept_name* attribute of the *instructor* relation (sorted on *dept_name*)

Biology	
Comp. Sci.	
Elec. Eng.	
Finance	
History	
Music	
Physics	

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	

Indexed-Sequential Files

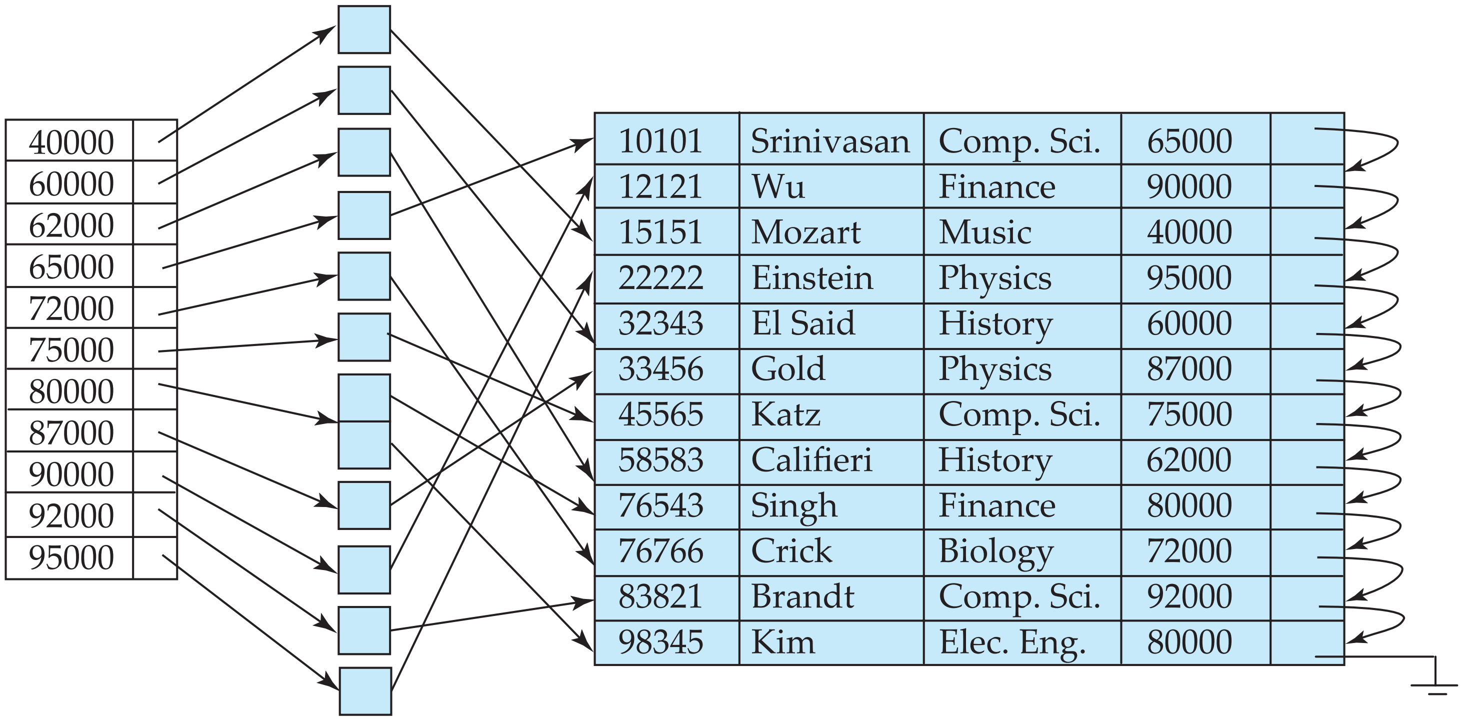
- Example: index on the *ID* attribute of the *instructor* relation
  - Sparse index: applicable when records are sequentially ordered on search-key
    - To locate a record with search-key value *K*:
      - Find index record with largest search-key value < *K*
      - Search file sequentially starting at the record to which the index record points

10101	
32343	
76766	

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	

Indexed-Sequential Files

- Example: Secondary index on the *salary* attribute of *instructor*
  - Secondary indices must be dense
  - Index record points to a bucket that contains pointers to all the actual records with that particular search-key value



Dense vs. Sparse Index

- Sparse index over dense index
  - Less space overhead
  - Less maintenance overhead for insertions and deletions
  - Generally, slower than dense index for locating records

Clustering vs. Non-Clustering Index

- Common rule of thumb: Indexes impose overhead on database modification
  - When a record is inserted or deleted, every index on the relation must be updated
  - When a record is updated, any index on an updated attribute must be updated

- Sequential scan using clustering index is efficient, but a sequential scan using a secondary (non-clustering) index is expensive on magnetic disk
  - Each record access may fetch a new block from disk
  - Each block fetch on magnetic disk requires about 5 to 10 milliseconds

Indexes on Multiple Keys

- Composite search-key
  - *E.g.*, index on the *instructor* relation on attributes (*name, ID*)
    - Values are sorted lexicographically
      - *E.g.*, (John, 12121) < (John, 13514)   and   (John, 13514) < (Peter, 11223)

    - One can query on just *name*, or on (*name, ID*)

Agenda

- Database indexes
- Ordered indexes
  - Indexed-sequential files
  - **B+tree index files**
    - B+tree nodes
    - Queries on B+trees
    - B+tree operations
- Hash indexes (*will not cover*)
- Inverted index

B+Tree Index

- Motivation: Disadvantage of indexed-sequential files
  - Performance degrades as file grows, since many overflow blocks get created
  - Periodic reorganization of entire file is required

- Remedy: B+tree index 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

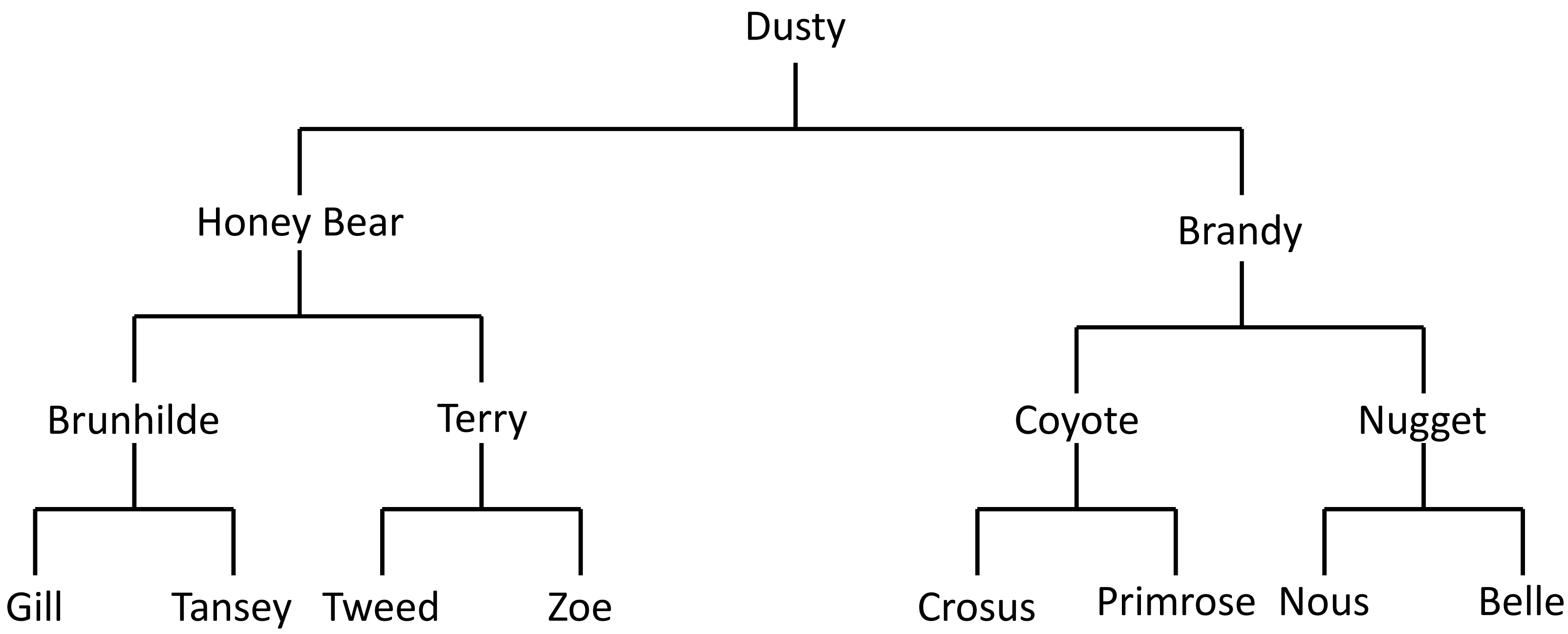
  - (Minor) disadvantage of B+trees:
    - Extra insertion and deletion overhead, space overhead

Trees

- Tree: A data structure representing hierarchical nature of a structure in a graphical form

Trees

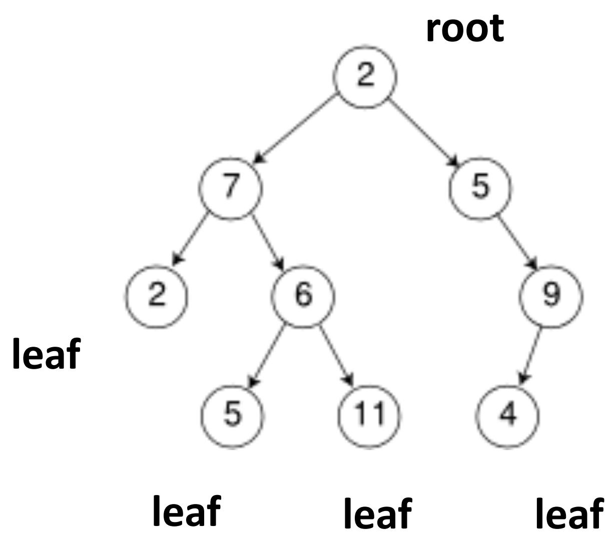
- Tree: A data structure representing hierarchical nature of a structure in a graphical form



< Pedigree >

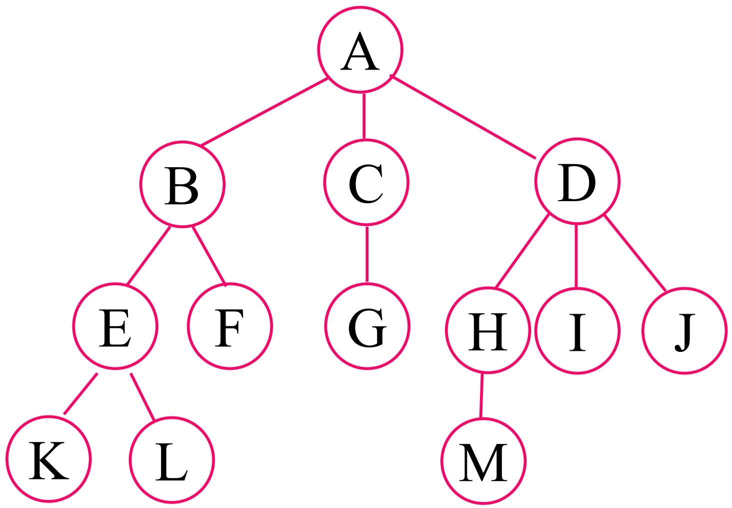
Trees

- Tree: a set of linked nodes that does not have a cycle
  - Each nodes has zero or more child nodes
  - A child has at most one parent
  - A node without a parent is called root
  - A node with no children is called leaf



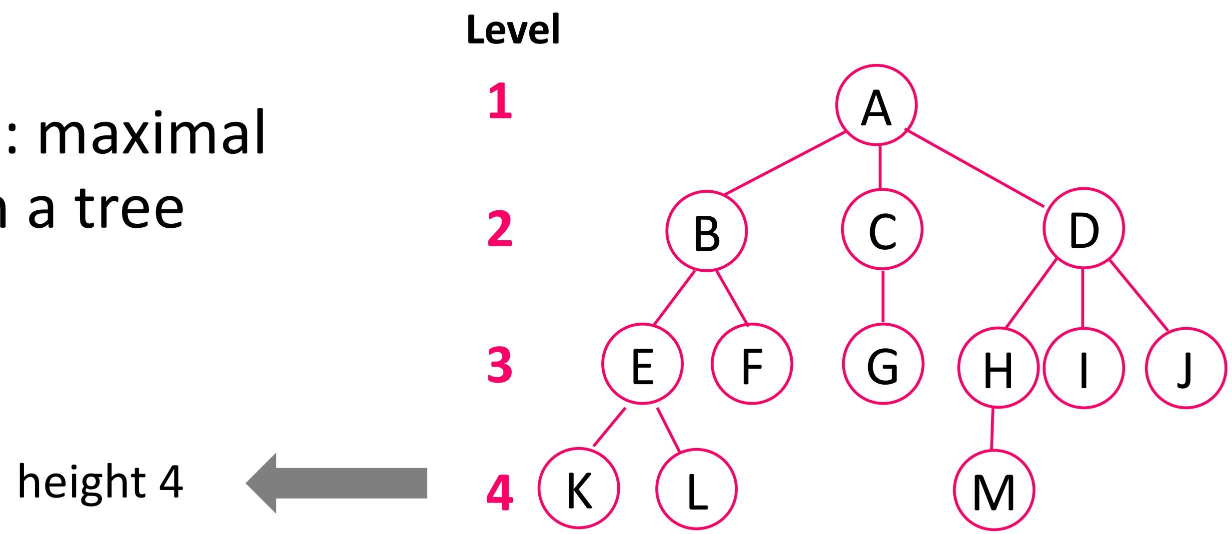
Terminologies

- Degree of a node: number of subtrees of a node
  - *E.g.*, degree of *A* = 3, *C* = 1, *F* = 0
- Degree of a tree: *maximum degree of the node* in a tree
  - *E.g.*, degree of the tree =3 (*A* and *D*)
- Leaf (of terminal node): node with 0 degree
  - *E.g., K, L, F, G, M, I, J*
- Sibling: children of the same parent
  - *E.g., H, I, J* are siblings



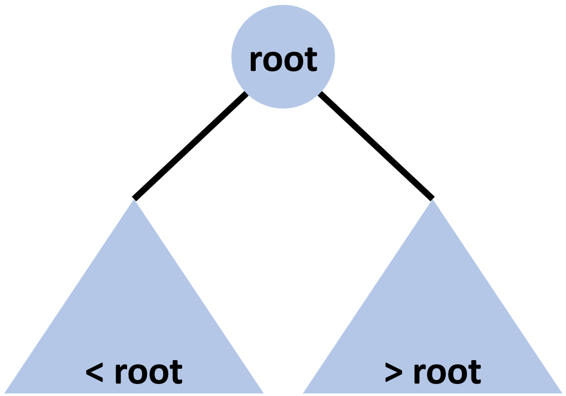
Terminologies

- Path: A sequence of nodes in which each node is adjacent to the next node
- Ancestors: All nodes along the path from root to the node
  - *E.g.*, ancestors of *M* are *A*, *D*, and *H*
- Descendants: All nodes in subtrees
  - *E.g.*, descendants of *B* are *E*, *F*, *K*, and *L*
- Level of node: Let the root node be at level one; if a node is at level *l*, then its children are at level *l*+1
  - *E.g.*, level of *M* = 4
- Height / depth of a tree: maximal level of any node in a tree



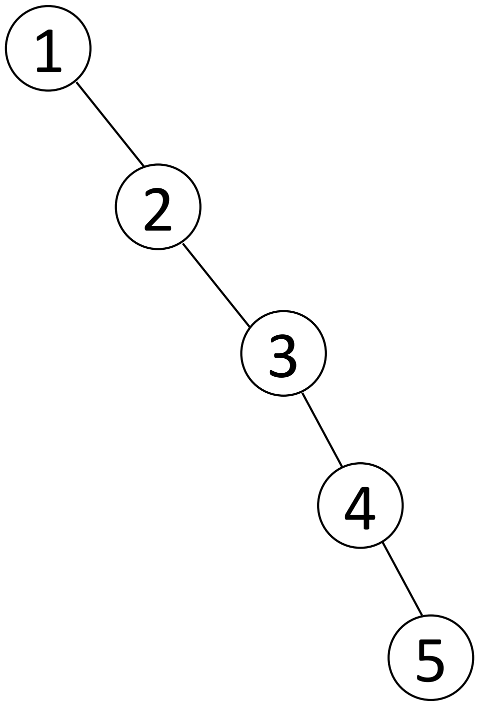
Binary Search Trees

- Motivation: oftentimes we would like to search an arbitrary element efficiently
- Binary Search Trees: a tree which may be empty, or satisfy the following conditions
  - Every element has a unique key (value)
  - Keys in left subtree must be smaller than that of root
  - Keys in right subtree must be larger than that of root
  - Left and right subtrees are also binary search tree
- Binary Search Trees may not be a complete tree

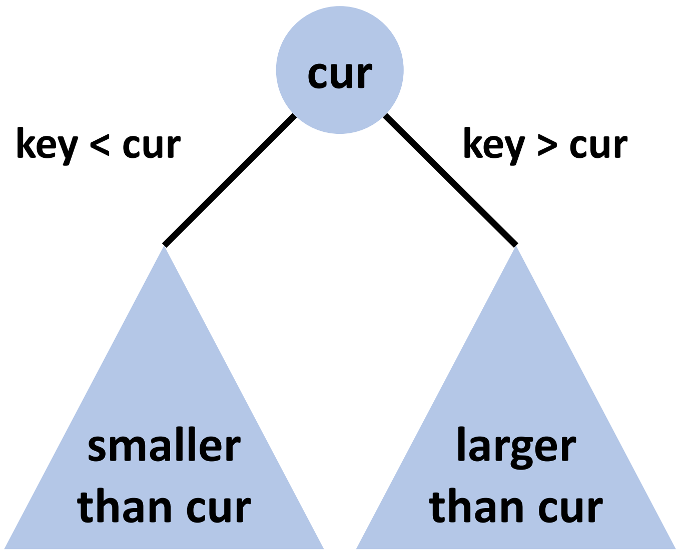


Binary Search Trees Issues

- Height of a binary search tree with *n* nodes
  - Worst case (sorted order): *n*
  - Average case (random): $O(log_2 n)$



Worst case (a skewed tree)



Random case

Remedy: Balanced Trees

- Self-balancing trees could alleviate the issue; *e.g.*,
  - AVL tree
  - Red-black tree
  - B+tree

B+Tree

- B-Tree family
  - *B-Tree* generally refers to a class of balanced tree data structures (B-tree and its variants)
  - Properties
    - Storage-friendly: B-trees work well on any layer of the storage hierarchy
      - Work well on both disks and main memory
    - Good performance on random and sequential accesses
      - Seeks/cache misses once for each node
    - Universal applicability

  - When people are mentioning B-Tree as a data structure, most likely they are referring to B+Tree
    - B-Tree (1971)
    - **B+Tree (1973)**
    - B*Tree (1977?)
    - B^{link}-Tree (1981)

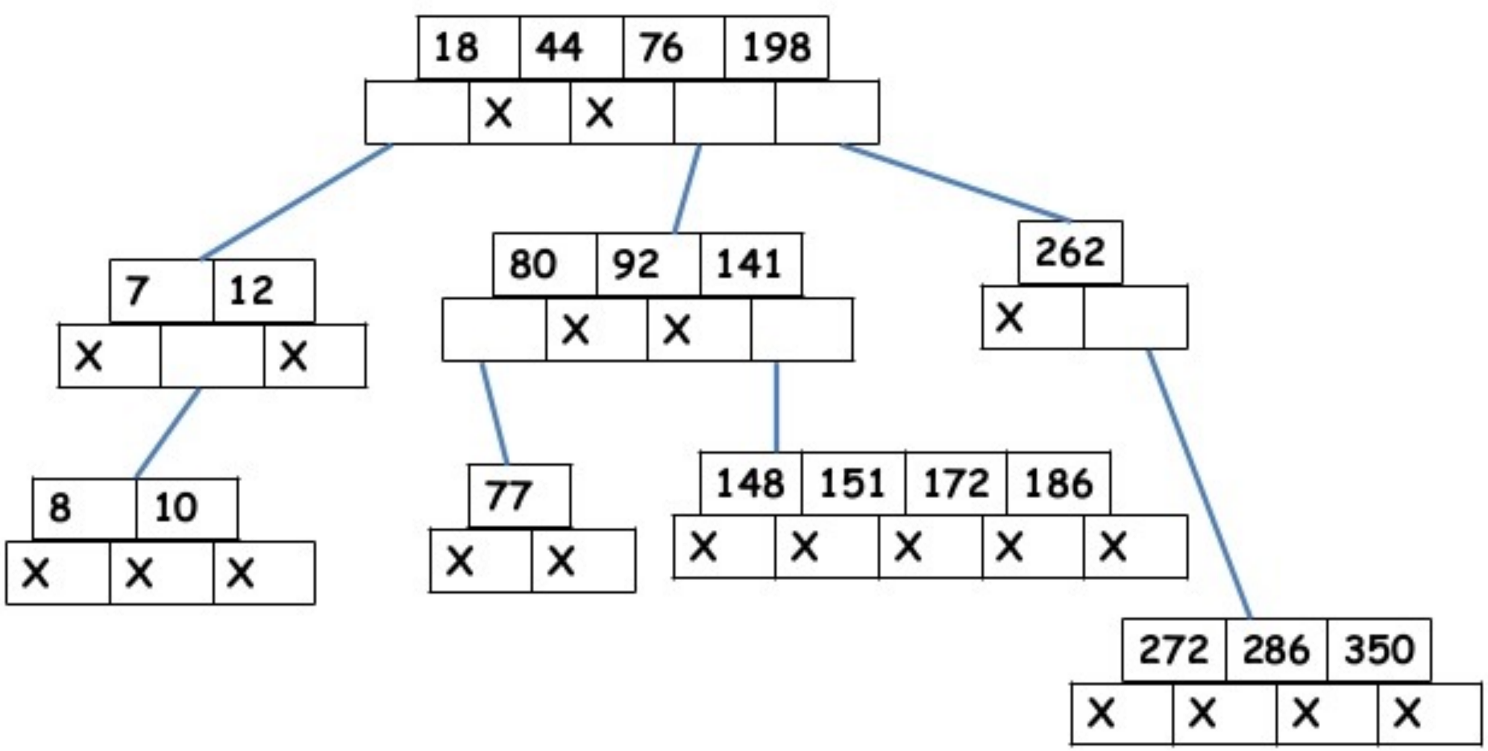
B+Tree

- B+Tree: a self-balancing tree that keeps data sorted and allows searches
  - B stands for "balanced”
  - It is a generalization of a binary search tree, in that a node can have more than two children
  - Optimized for systems that read and write large blocks of data

  - Sequential access, insertions, and deletions are done in O(log *n*)
    - For a tree with *n* nodes, the distance between the root and any leaf node is always log *n*

B+Tree

- B+tree is an *M*-way search tree with the following properties:
  - *M*-way search tree:
    - Multi-way tree (a generalized version of binary search trees)
    - Each node contains a maximum of *M*-1 elements and *M* children

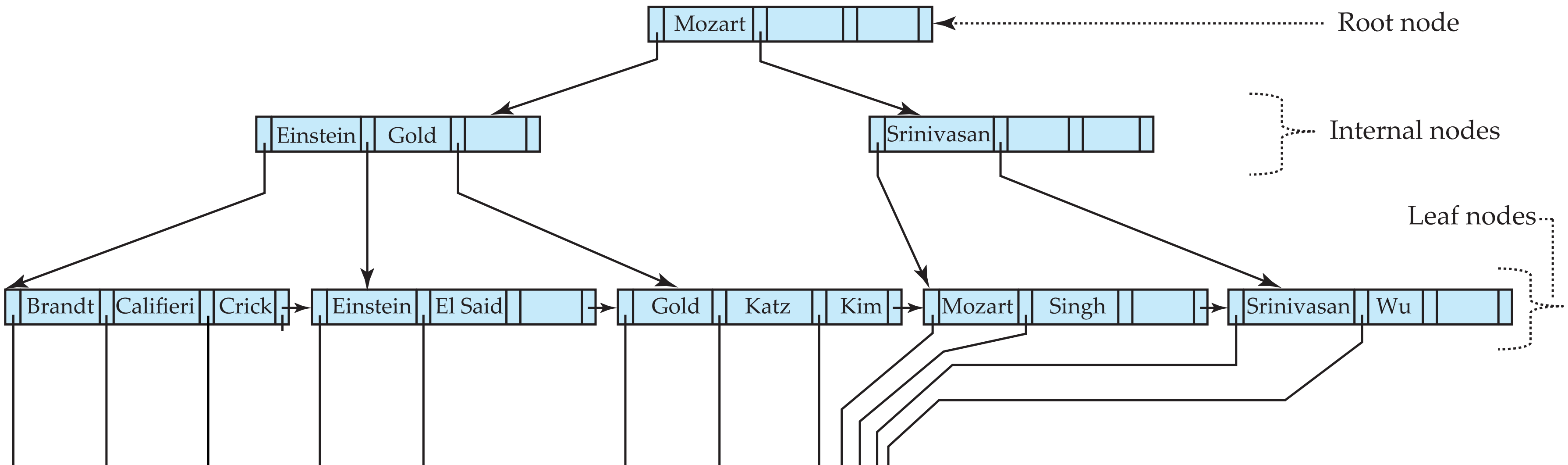


B+Tree

- B+tree is an *M*-way search tree with the following properties:
  - *M*-way search tree:
    - Multi-way tree (a generalized version of binary search trees)
    - Each node contains a maximum of *M*-1 elements and *M* children
  - Perfectly balanced; *i.e.*, every leaf node is at the same depth
    - Distance to any leaf node is always log *n*
  - Every inner node (other than the root) is at least half-full
    - $M/2 - 1 \leq$ #keys $\leq M-1$
    - Every node has at least $M/2 - 1$ children
  - 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 (*M*-1) values

B+Tree

- An example B+tree (*a 4-way B+tree*)



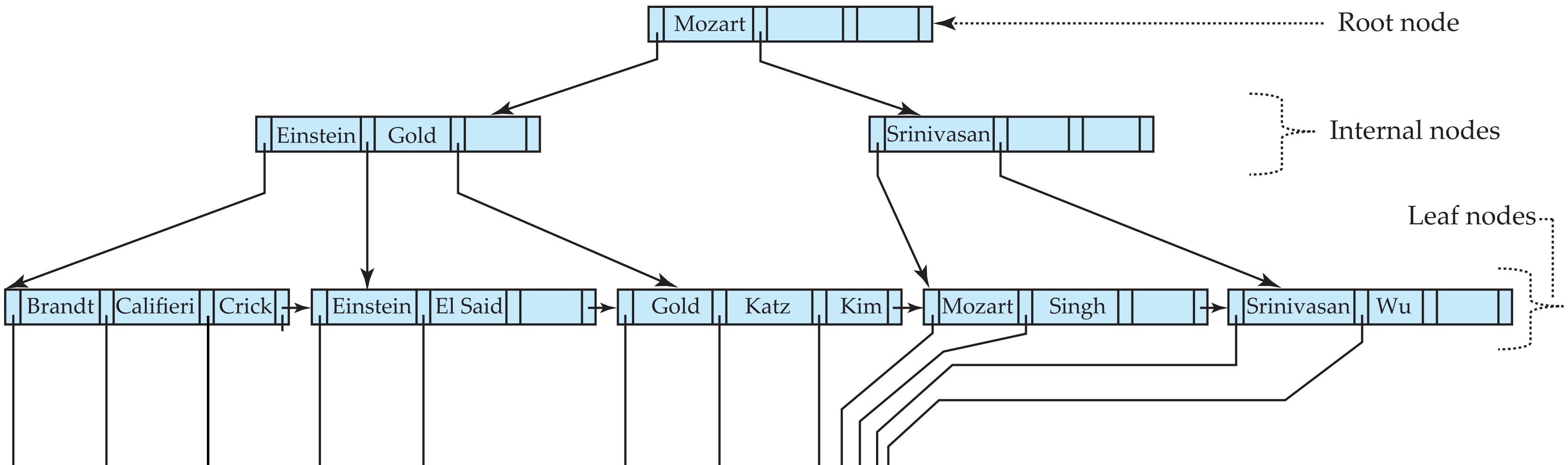
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000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

Agenda

- Database indexes
- Ordered indexes
  - Indexed-sequential files
  - B+tree index files
    - **B+tree nodes**
    - Queries on B+trees
    - B+tree operations
- Hash indexes (*will not cover*)
- Inverted index

B+Tree

- An example B+tree (*a 4-way B+tree*)



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

B+Tree Nodes

- Typical B+tree nodes

P_1	K_1	P_2	…	P_{M-1}	K_{M-1}	P_M

  - K_i are the search-key values (what are indexed)
  - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes)
  - The arrays are kept in sorted key order
    $K_1 < K_2 < K_3 < \ldots < K_{M-1}$

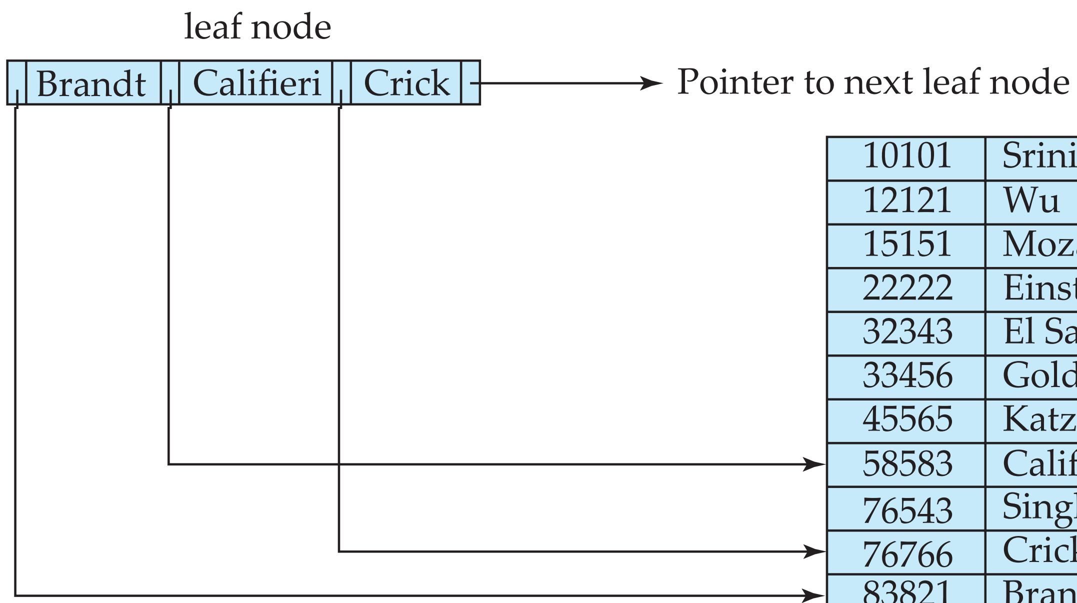
B+Tree Nodes

- Non-leaf nodes
  - All the search-keys in the subtree to which P_1 points are less than K_1
  - For $2 \leq i \leq M - 1$, all the search-keys in the subtree to which P_i points have values greater than or equal to K_{i-1} and less than K_i
  - All the search-keys in the subtree to which P_M points have values greater than or equal to K_{M-1}

P_1	K_1	P_2	…	P_{M-1}	K_{M-1}	P_M

B+Tree Nodes

- Leaf nodes
  - For i = 1, 2, . . ., M–1, pointer P_i points to a file record with search-key value K_i,
  - Search-keys are sorted: If L_i, L_j are leaf nodes and $i < j$, L_i's search-key values are less than or equal to L_j's search-key values
  - P_M points to next leaf node in search-key order



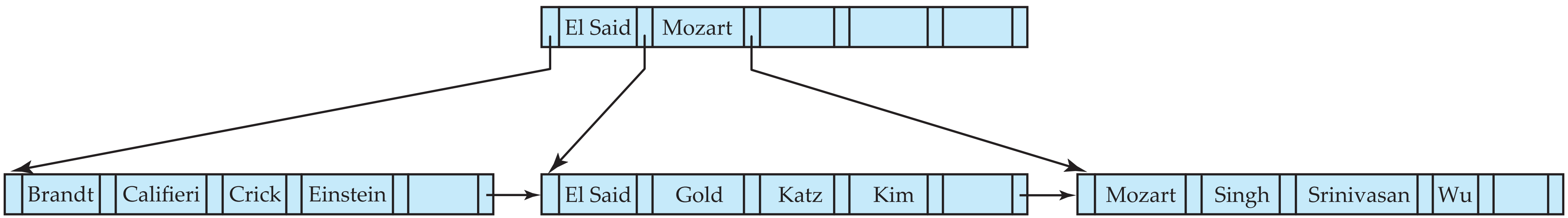
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000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

B+Tree Nodes

- In commercial products, leaf node values may vary
  - For $i = 1, 2, \ldots, M-1$, pointer P_i points to:
    - Use-case #1 (as discussed in the previous slide): Record IDs – a pointer to the location of the tuple that the index entry corresponds to
      - PostgreSQL, DB2, SQL Server, Oracle
    - Use-case #2: Tuple data – The actual contents of the tuple is stored in the leaf node
      - Secondary indexes have to store the record ID as their values
      - More complicated
      - MySQL, SQLite, SQL Server, Oracle

Another Example B+Tree

- B+tree for *instructor* (degree=6 or 6-way B+tree)



  - Leaf nodes must have between 3 and 5 values
    ∵ $\lceil (M-1)/2 \rceil$ and $M-1$, with $M = 6$

  - Non-leaf nodes other than root must have between 3 and 6 children
    ∵ $\lceil (M/2 \rceil$ and M with $M = 6$

  - Root must have at least 2 children

Observations

- 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 indexes

- The B+tree contains a relatively small number of levels
  - If there are K search-key values in the file, the tree height is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$ $\Rightarrow$ searches can be done efficiently
    - 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
    - *… and so forth*

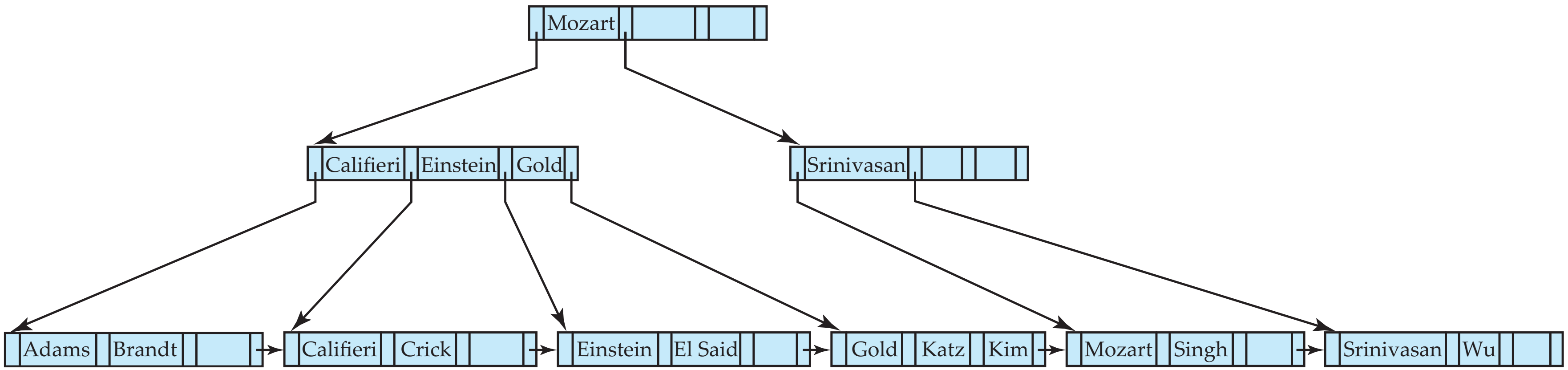
- Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time (we'll see)

Agenda

- Database indexes
- Ordered indexes
  - Indexed-sequential files
  - B+tree index files
    - B+tree nodes
    - **Queries on B+trees**
    - B+tree operations
- Hash indexes (*will not cover*)
- Inverted index

Queries on B+Trees

- **function** *find*(v)
  1. *C=root*
  2. **while** (*C* is not a leaf node)
     1. Let i be least number s.t. $V \leq K_i$
     2. **if** there is no such number *i then*
     3. Set C = *last non-null pointer in C*
     4. **else if** ($v = C.K_i$) Set $C = P_{i+1}$
     5. **else set** $C = C.P_i$
  3. **if** for some i, $K_i = V$ **then** return $C.P_i$
  4. **else** return null /* no record with search-key value *v* exists. */



Queries on B+Trees

- Range queries find all records with search-key values in a given range
  - **function** *findRange*(*lb, ub*) which returns set of all such records is available in the textbook
  - Real implementations usually provide an iterator interface to fetch matching records one at a time, using a *next*() function

Queries on B+Trees

- Time complexity: If there are K search-key values in the file, the height of the tree is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$

- A node is generally the same size as a disk block, typically 4 KB
  - and M is typically around 100 (40 bytes per index entry)

- With 1 million search-key values and M = 100
  - At most $log_{50}(1{,}000{,}000)$ = 4 nodes are accessed in a lookup traversal from root to leaf
  - *C.f.*, a *balanced binary tree* with 1 million search-key values — around 20 nodes are accessed in a lookup

Agenda

- Database indexes
- Ordered indexes
  - Indexed-sequential files
  - B+tree index files
    - B+tree nodes
    - Queries on B+trees
    - **B+tree operations**
- Hash indexes (*will not cover*)
- Inverted index

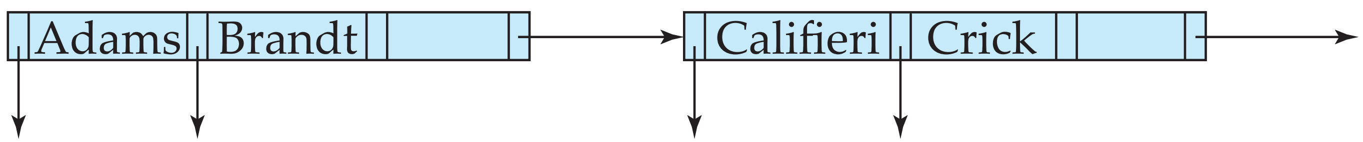
B+Tree Operations

- Insertion
  1. Find the proper leaf node *L* for the newly inserted key
  2. Put data entry into *L* in sorted order
  3. If *L* has enough space, done
     Otherwise, split *L*: . . . . a half of the keys stay in L
     another half goes to a new node L_2
     - Redistribute entries evenly
     - Insert index entry pointing to L_2 into the parent of *L*

- Demo: https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html
  (Algorithm Visualizations, David Galles)

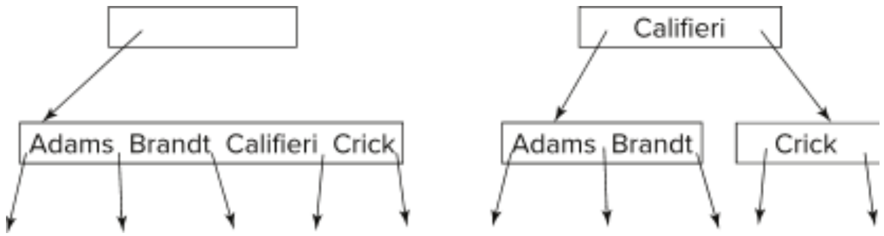
B+Tree Operations

- Insertion (cont'd)
  - Splitting a leaf node:
    - Take the M (search-key, pointer) pairs, including the one being inserted, in sorted order
    - Place the first $\lceil M/2 \rceil$ 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

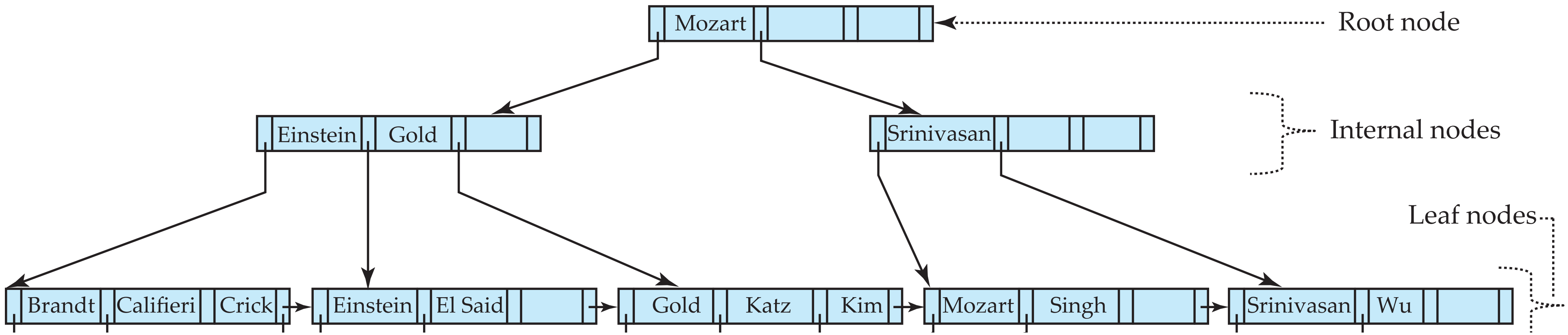


B+Tree Operations

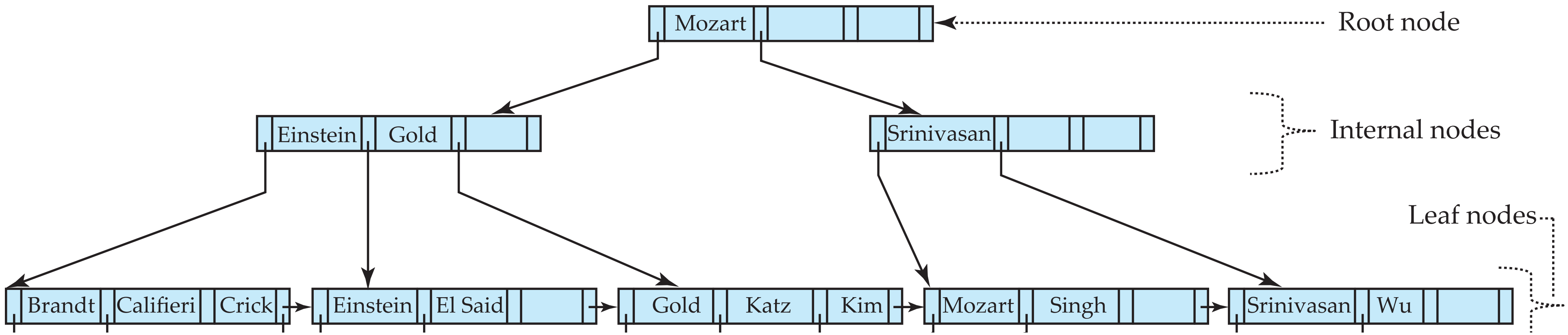
- Splitting a non-leaf node: when inserting (*k*,*p*) into an already full internal node *N*
  - Copy *N* to an in-memory area S with space for M+1 pointers and M keys
  - Insert (*k*,*p*) into S
  - Copy $P_1, K_1, \ldots, K_{\lceil M/2 \rceil - 1}, P_{\lceil M/2 \rceil}$ from S back into node N
  - Copy $P_{\lceil M/2 \rceil + 1}, K_{\lceil M/2 \rceil + 1}, \ldots, K_M, P_{M+1}$ from S into newly allocated node N’
  - Insert ($K_{\lceil M/2 \rceil}$,N') into parent N
  - Example



B+Tree Insertion

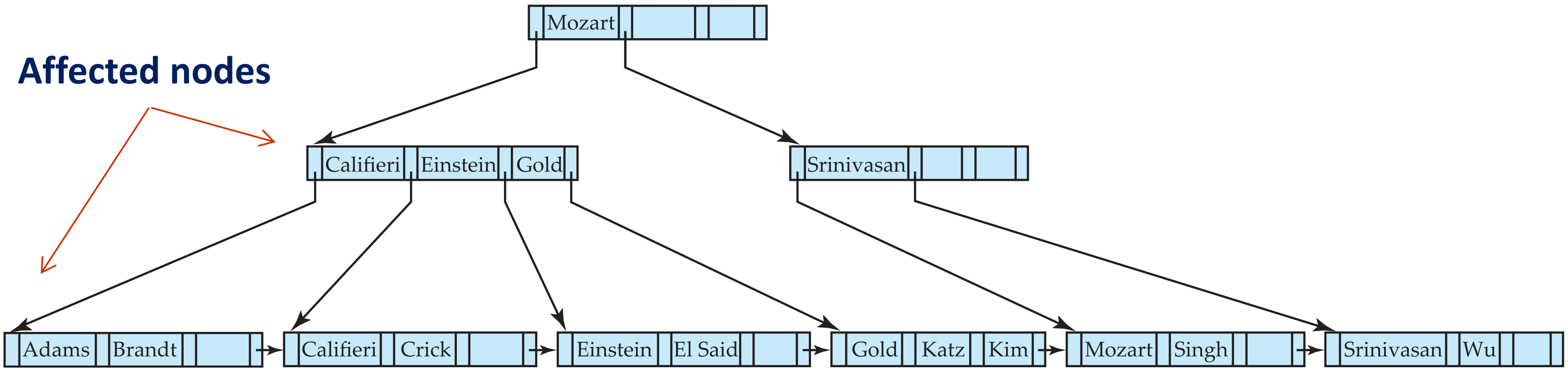


B+Tree Insertion

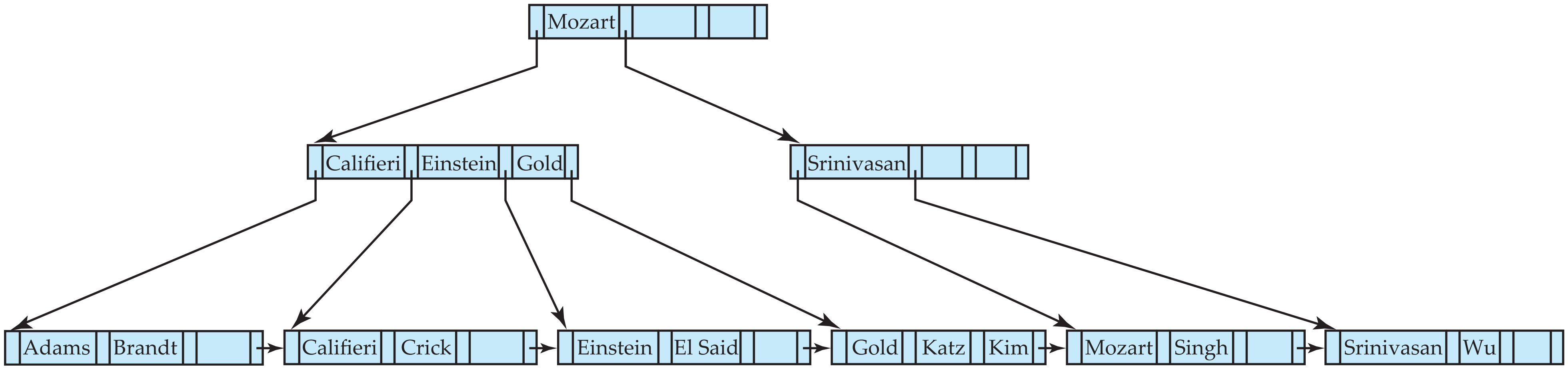


- Insert "Adams"

Affected nodes

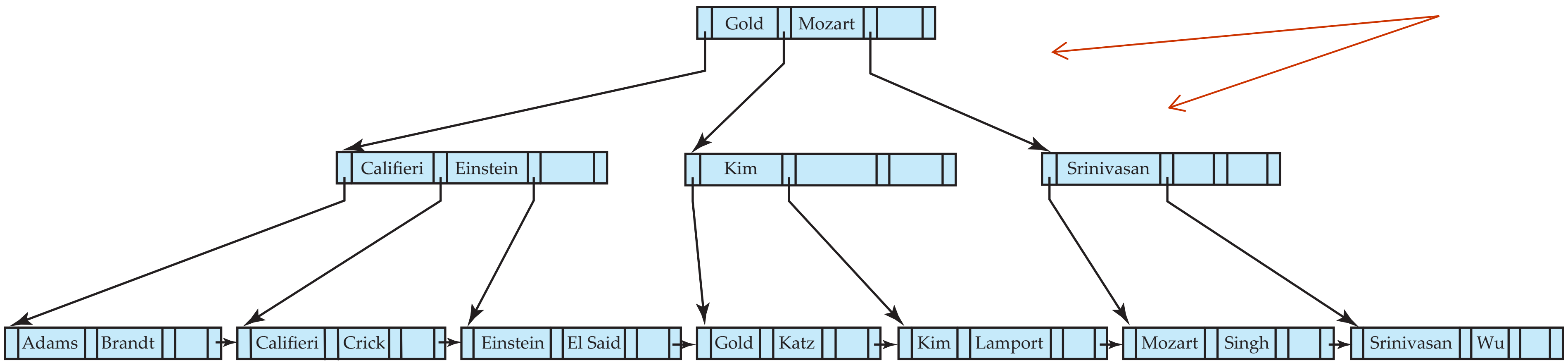


B+Tree Insertion



- Insert “Lamport”

Affected nodes



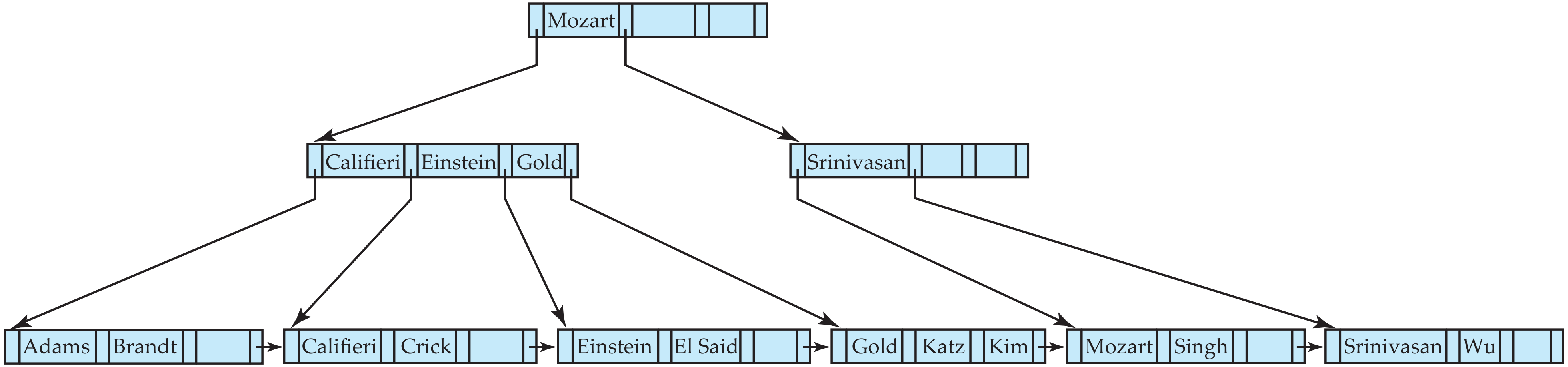
Affected nodes

B+Tree Operations

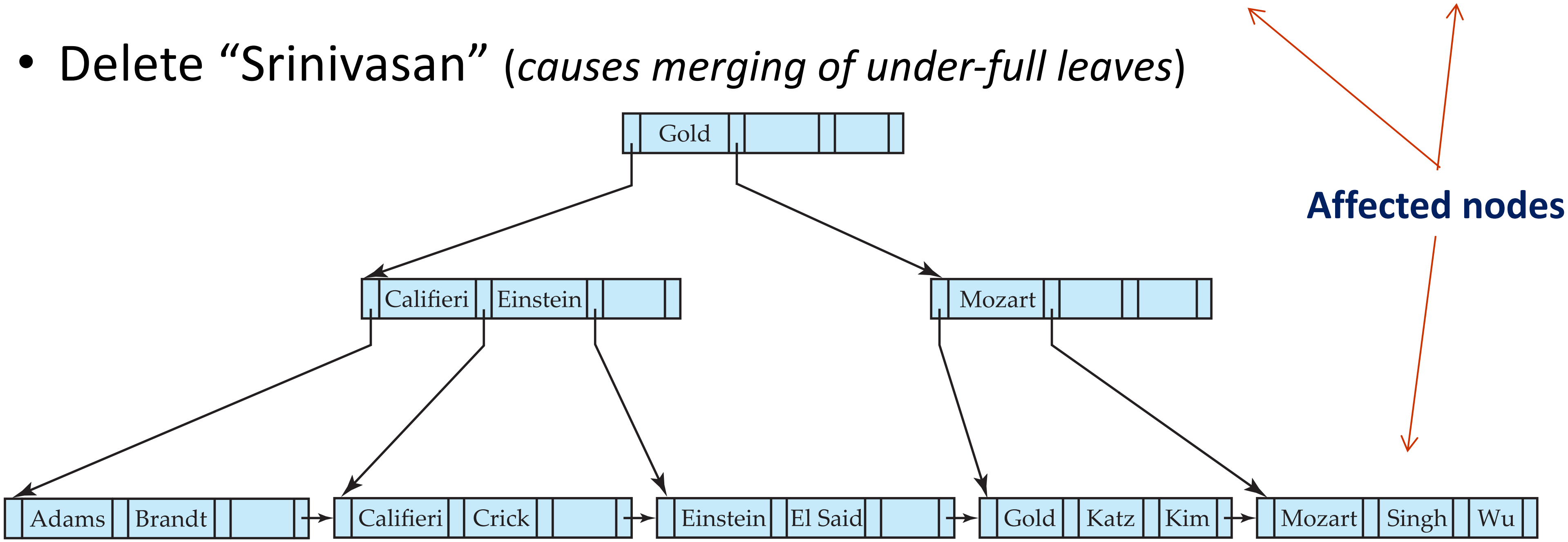
- Deletion
  1. Start at root, find leaf *L* where entry belongs
  2. Remove the entry
  3. If *L* is at least half-full, done
     If *L* has only *M*/2-1 entries
     - Try to rebalance, by borrowing a key from a sibling
  4. If a merge occurred, one must delete its entry from the parent of *L*
     - If redistribution fails, merge *L* and one of its sibling

- Demo: https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html
  (Algorithm Visualizations, David Galles)

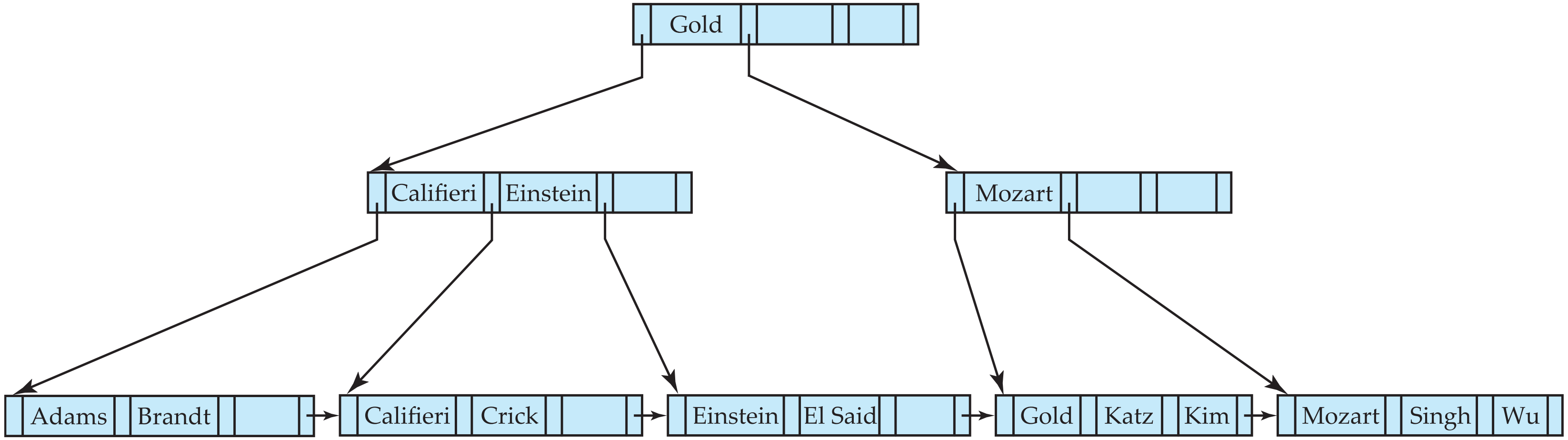
B+Tree Deletion



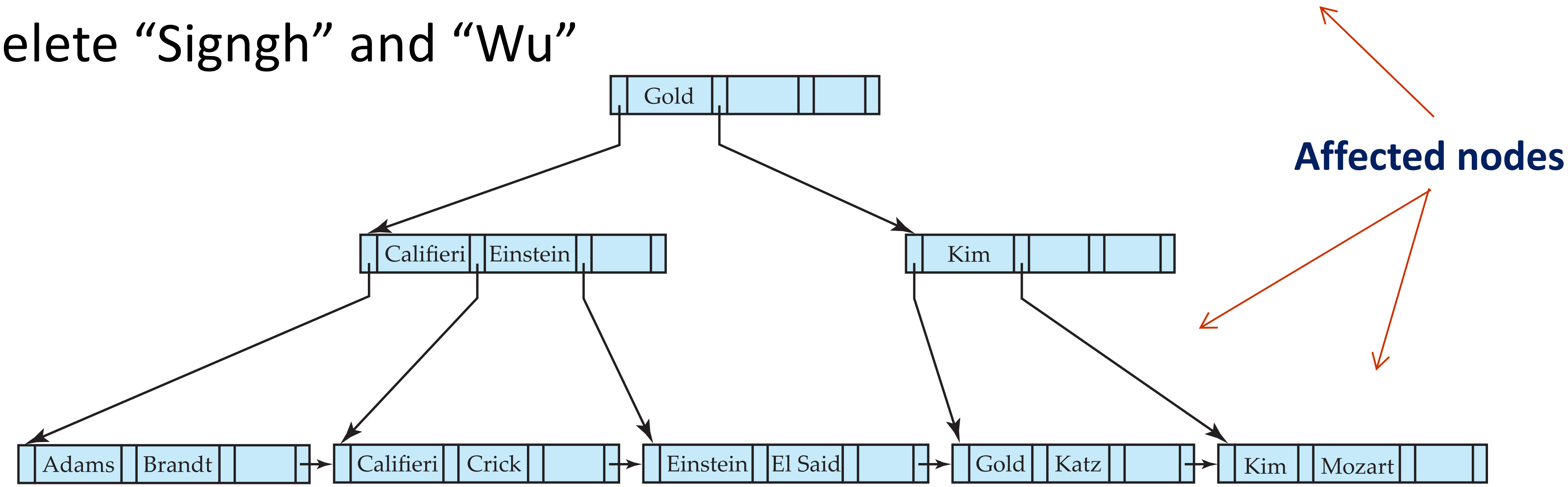
- Delete “Srinivasan” (*causes merging of under-full leaves*)



B+Tree Deletion

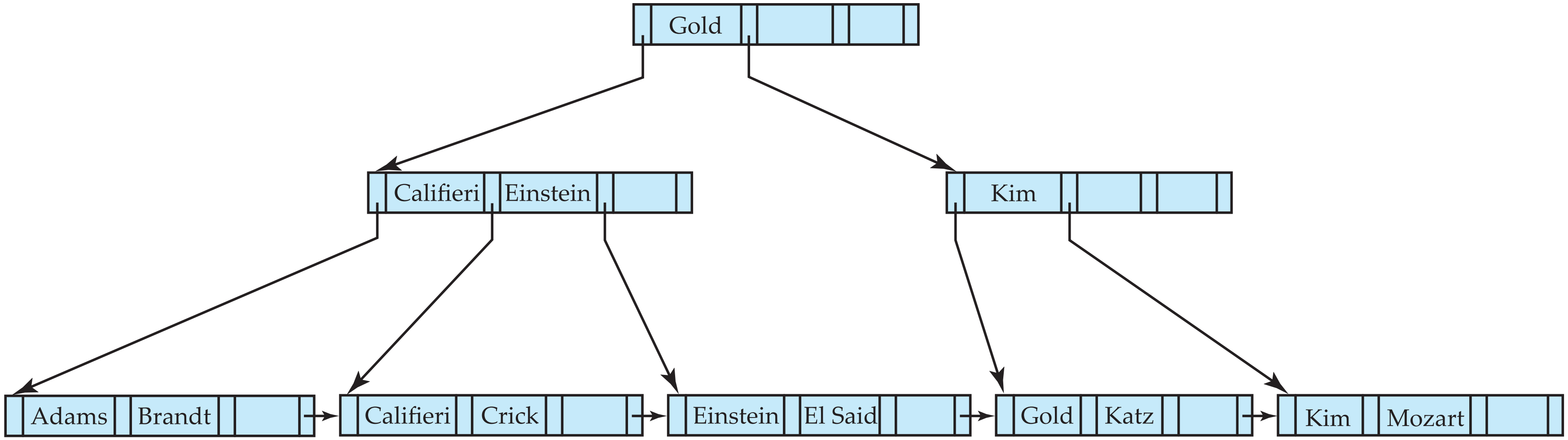


- Delete “Signgh” and “Wu”

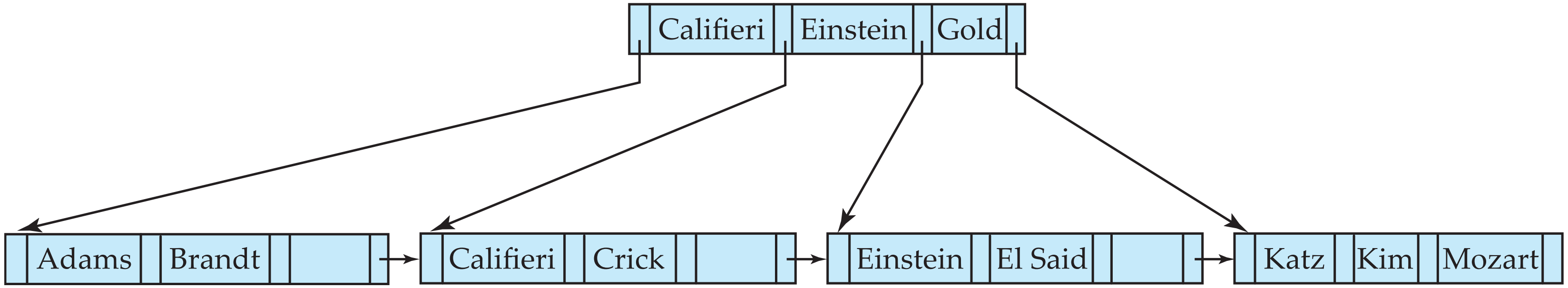


- Leaf containing Singh and Wu became under-full, and borrowed a value Kim from its left sibling
- Search-key value in the parent changes as a result

B+Tree Deletion



- Delete "Gold"



  - Node with Gold and Katz became under-full, and was merged with its sibling
  - Parent node becomes under-full, and is merged with its sibling
    - Value separating two nodes (at the parent) is pulled down when merging
  - Root node then has only one child, and is deleted

Remarks on B+Tree Deletion

- The node deletions may cascade upwards till a node which has $\lceil M/2 \rceil$ or more pointers is found
- If the root node has only one pointer after deletion, it is deleted and the sole child becomes the root

Indexes with B+Trees

- Key take-homes
  - No need to search whole table (time efficient)
  - No need to explicitly sort data
  - Insertions and deletions can be done in logarithmic time

  - Use extra space (replica of a subset of data)

Agenda

- Database indexes
- Ordered indexes
  - Indexed-sequential files
  - B+tree index files
    - B+tree nodes
    - Queries on B+trees
    - **B+tree operations**
- Hash indexes (*will not cover*)
- **Inverted index**

Inverted Index

- Observation
  - The tree indexes that we have discussed so far:
    - Useful for "point" and "range" queries
      - Find all customers in zip code = 15213
      - Find all orders between June 2018 and September 2018
  - They are NOT good at keyword searches:
    - Find all Wikipedia articles that contain the word "Soccer"

Inverted Index

- Inverted Index
  - An inverted index stores a mapping of words to records that contain those words in the target attribute
    - Sometimes called a full-text search index
    - Also called a concordance in old contexts
  - The major DBMSs support these natively
    - MySQL: https://dev.mysql.com/doc/refman/8.0/en/innodb-fulltext-index.html
      - One has to use InnoDB as one's DB engine, and special queries to look up
  - There are also specialized DBMSs
    - *Lucene, Solr, Elasticsearch, Sphinx, Xapian*

KUBiC: Korea Unification Bigdata Center

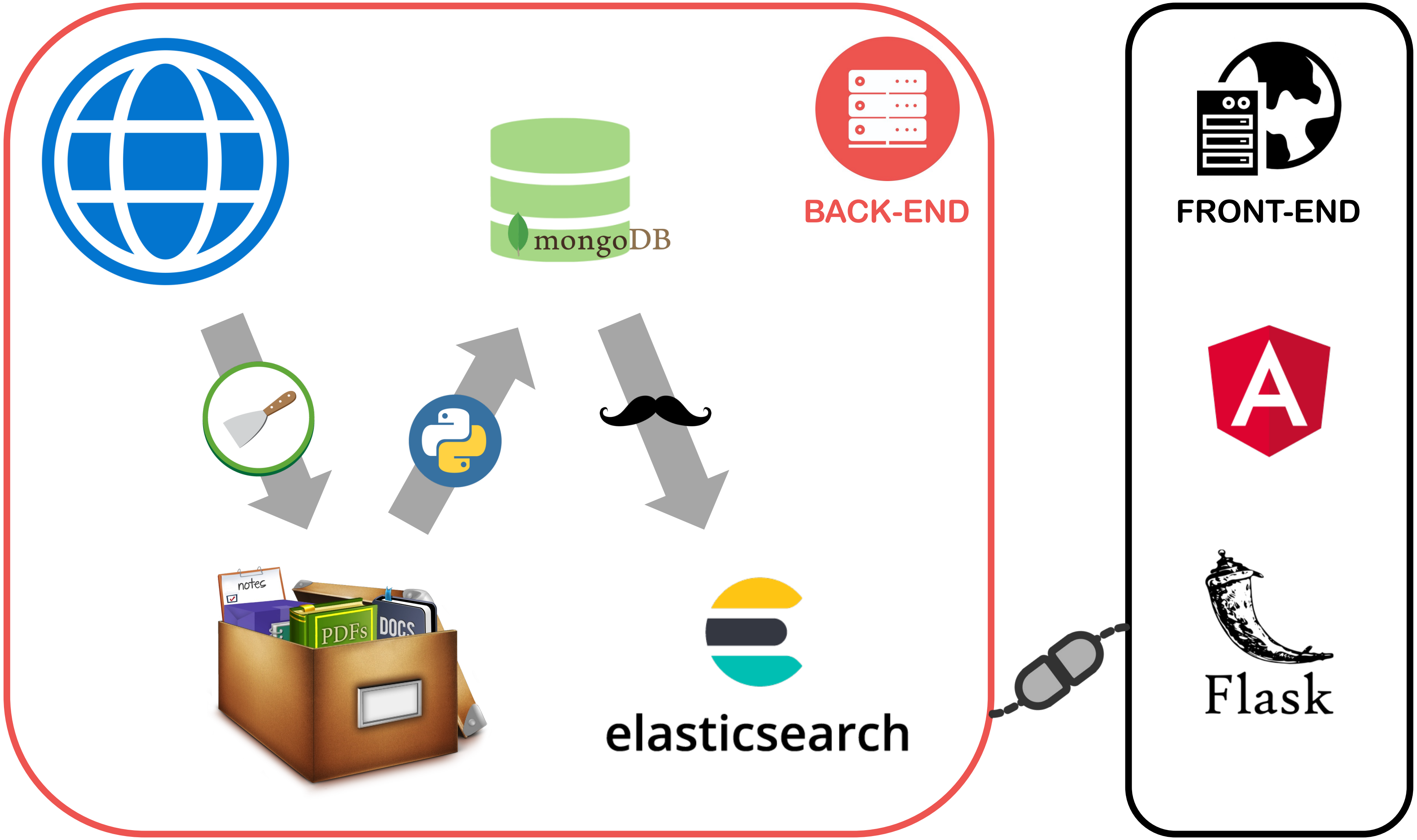
- A government-funded project on a data-center development focusing on the Korean unification
  - URL: https://kubic.handong.edu/
  - Data archive + search engine + web-based analysis tools, specialized on the Korean unification and North Korea research
  - Contains 20,000+ academic papers and government reports on the relevant topics

KUBiC
Korea Unification Bigdata Center

한동대학교
HANDONG GLOBAL UNIVERSITY

통일부
Ministry of Unification

Project Taskflow



Inverted Index

- Example

Doc 1
I did enact Julius Caesar: I was killed i' the Capitol; Brutus killed me.

Doc 2
So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious:

term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

$\Longrightarrow$

term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

$\Longrightarrow$

term	doc. freq.	→	postings lists
ambitious	1	→	2
be	1	→	2
brutus	2	→	1 → 2
capitol	1	→	1
caesar	2	→	1 → 2
did	1	→	1
enact	1	→	1
hath	1	→	2
I	1	→	1
i'	1	→	1
it	1	→	2
julius	1	→	1
killed	1	→	1
let	1	→	2
me	1	→	1
noble	1	→	2
so	1	→	2
the	2	→	1 → 2
told	1	→	2
you	1	→	2
was	2	→	1 → 2
with	1	→	2

► **Figure 1.3** Building an index by sorting and grouping. The sequence of terms in each document, tagged by their documentID (left) is sorted alphabetically (middle). Instances of the same term are then grouped by word and then by documentID. The terms and documentIDs are then separated out (right). The dictionary stores the terms, and has a pointer to the postings list for each term. It commonly also stores other summary information such as, here, the document frequency of each term. We use this information for improving query time efficiency and, later, for weighting in ranked retrieval models. Each postings list stores the list of documents in which a term occurs, and may store other information such as the term frequency (the frequency of each term in each document) or the position(s) of the term in each document.

Source: https://nlp.stanford.edu/IR-book/html/htmledition/a-first-take-at-building-an-inverted-index-1.html

Indexer steps: Token sequence

- Sequence of (Modified token, Document ID) pairs

Doc 1

I did enact Julius Caesar I was killed i' the Capitol; Brutus killed me.

Doc 2

So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious

Term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

Indexer steps: Sort

- Sort by terms
  - And then docID
  - This is the core indexing step

Term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

Term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

Indexer steps: Dictionary & Postings

- Multiple term entries in a single document are merged
- Split into Dictionary and Postings
- Doc. frequency information is added

Term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

→

term	doc. freq.	→	postings lists
ambitious	1	→	2
be	1	→	2
brutus	2	→	1 → 2
capitol	1	→	1
caesar	2	→	1 → 2
did	1	→	1
enact	1	→	1
hath	1	→	2
i	1	→	1
i'	1	→	1
it	1	→	2
julius	1	→	1
killed	1	→	1
let	1	→	2
me	1	→	1
noble	1	→	2
so	1	→	2
the	2	→	1 → 2
told	1	→	2
you	1	→	2
was	2	→	1 → 2
with	1	→	2

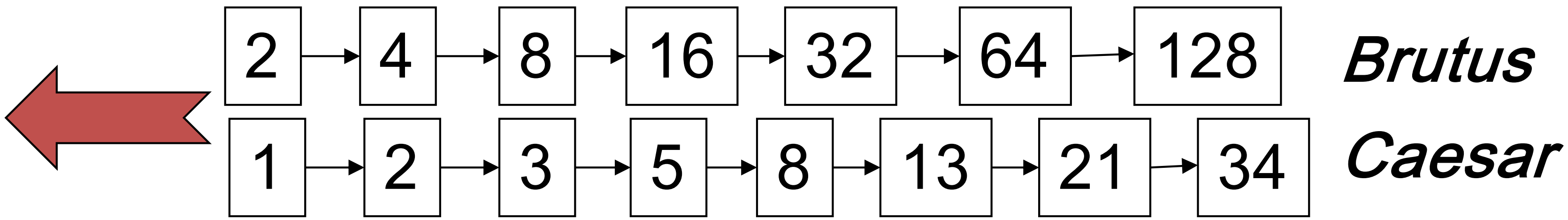
Inverted Index

- For each term *t*, we must store a list of all documents that contain *t*
  - Identify each doc by a **docID**, a document serial number
  - Can we use fixed-size arrays for this?
    - In memory, can use linked lists or variable length arrays

Brutus	1	2	4	11	31	45	173	174
Caesar	1	2	4	5	6	16	57	132
Calpurnia	2	31	54	101				

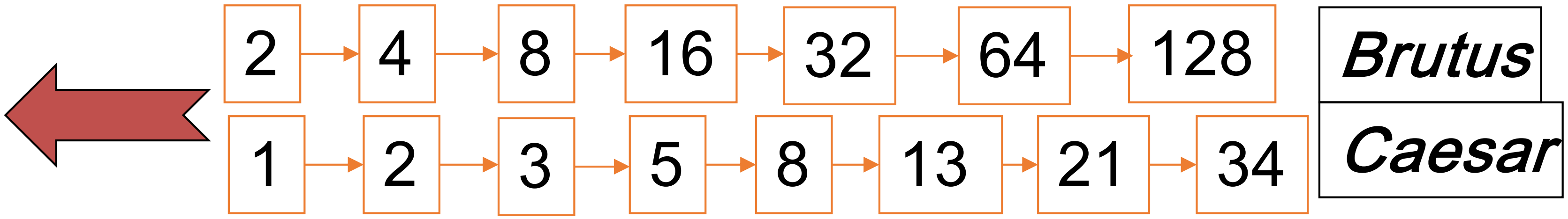
Query Processing

- How do we process a query, using the index we just built?
- Query processing: AND
  - *E.g.*, Consider processing the query:
    - ***Brutus*** *AND* ***Caesar***
      - Locate ***Brutus*** in the Dictionary;
        - Retrieve its postings
      - Locate ***Caesar*** in the Dictionary;
        - Retrieve its postings
      - "Merge" the two postings (intersect the document sets):



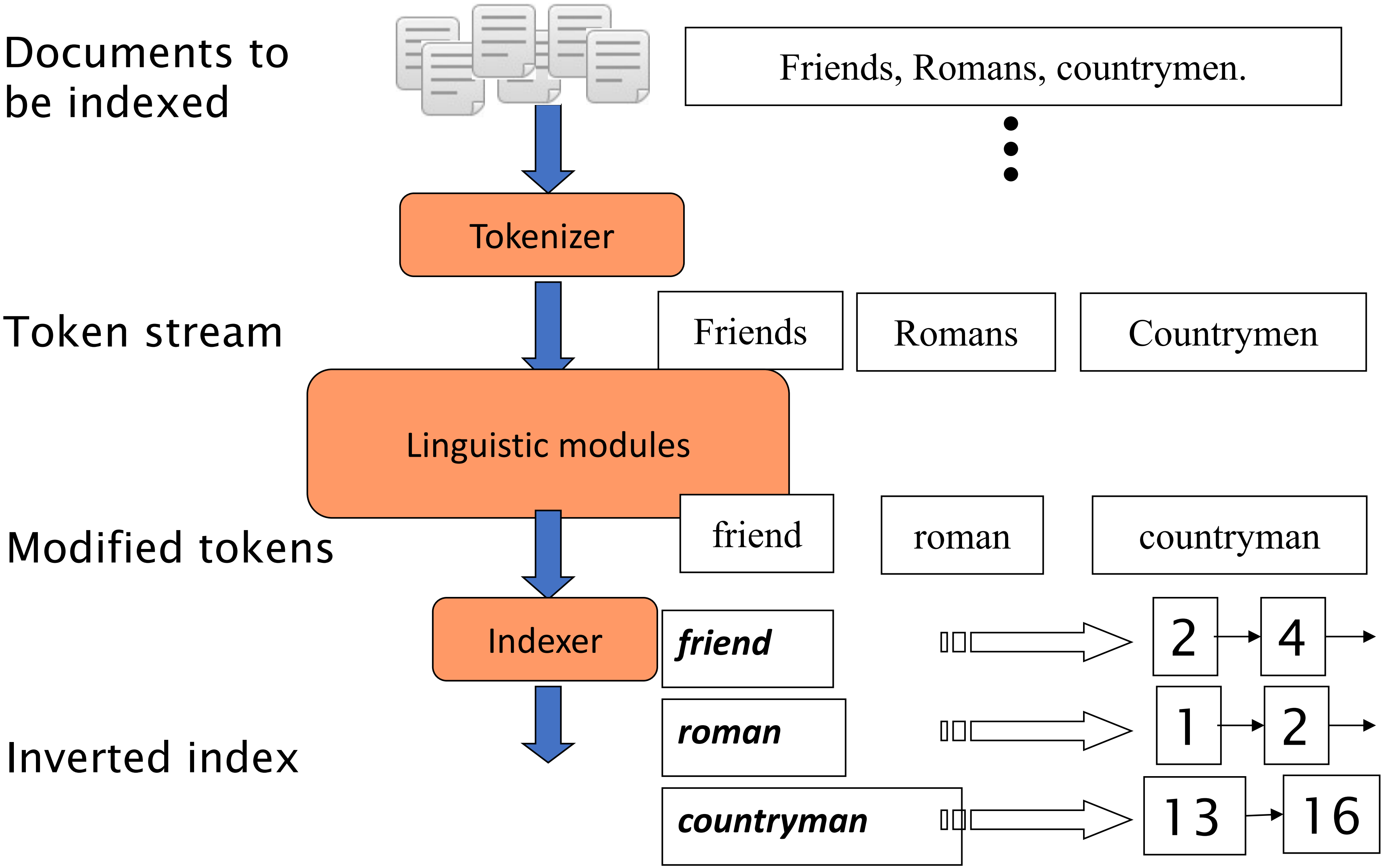
Query Processing

- *E.g.*, (cont'd)
  - Walk through the two postings simultaneously, in time linear in the total number of postings entries



- If the list lengths are x and y, the merge takes $O(x+y)$ operations
  - <u>Crucial assumption</u>: postings are sorted by docID

Inverted Index



Ranking Algorithms

- Inverted index simply offers the list of documents that contain the search-word
- Ranking/ordering of the search results lets the users access important documents earlier/easier

* 참고: https://opensourceconnections.com/blog/2015/10/16/bm25-the-next-generation-of-lucene-relevation/