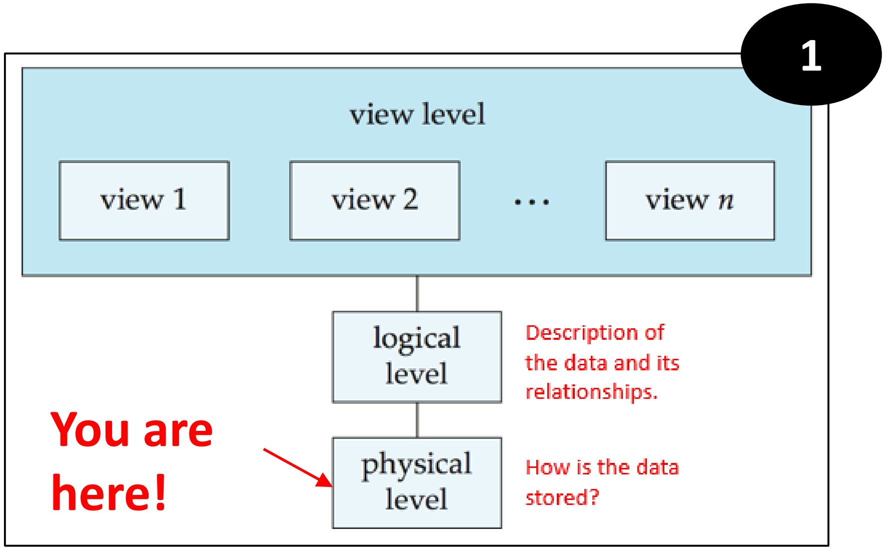
RDBMS and SQL
Physical View and Indexing

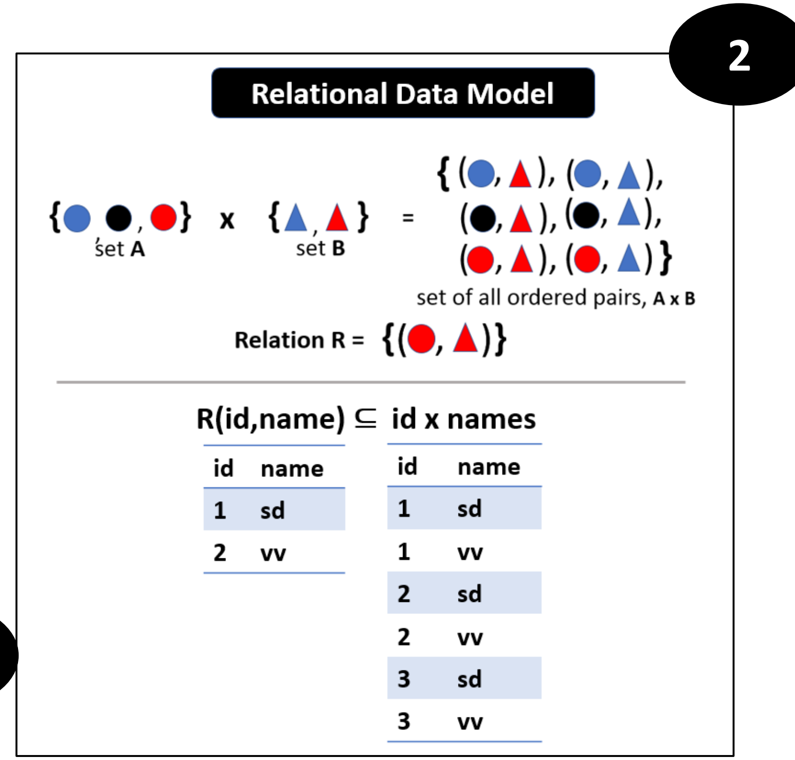
Venkatesh Vinayakarao
venkateshv@cmi.ac.in
http://vvtesh.co.in

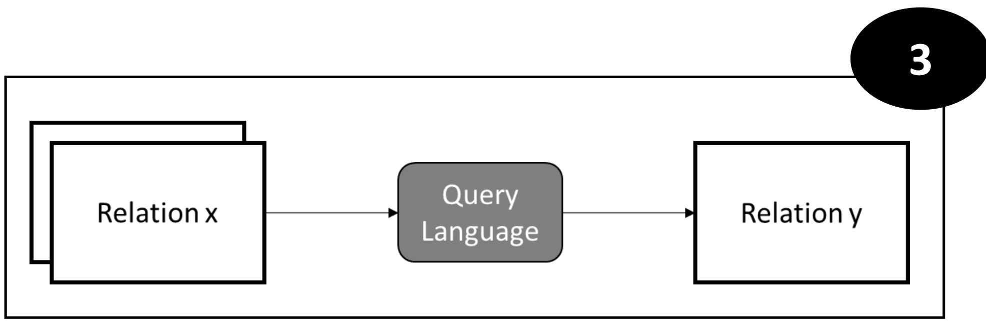
Chennai Mathematical Institute

Slide contents are borrowed from the course text. For the authors' original version of slides, visit: https://www.db-book.com/db6/slide-dir/index.html.

Story So Far…

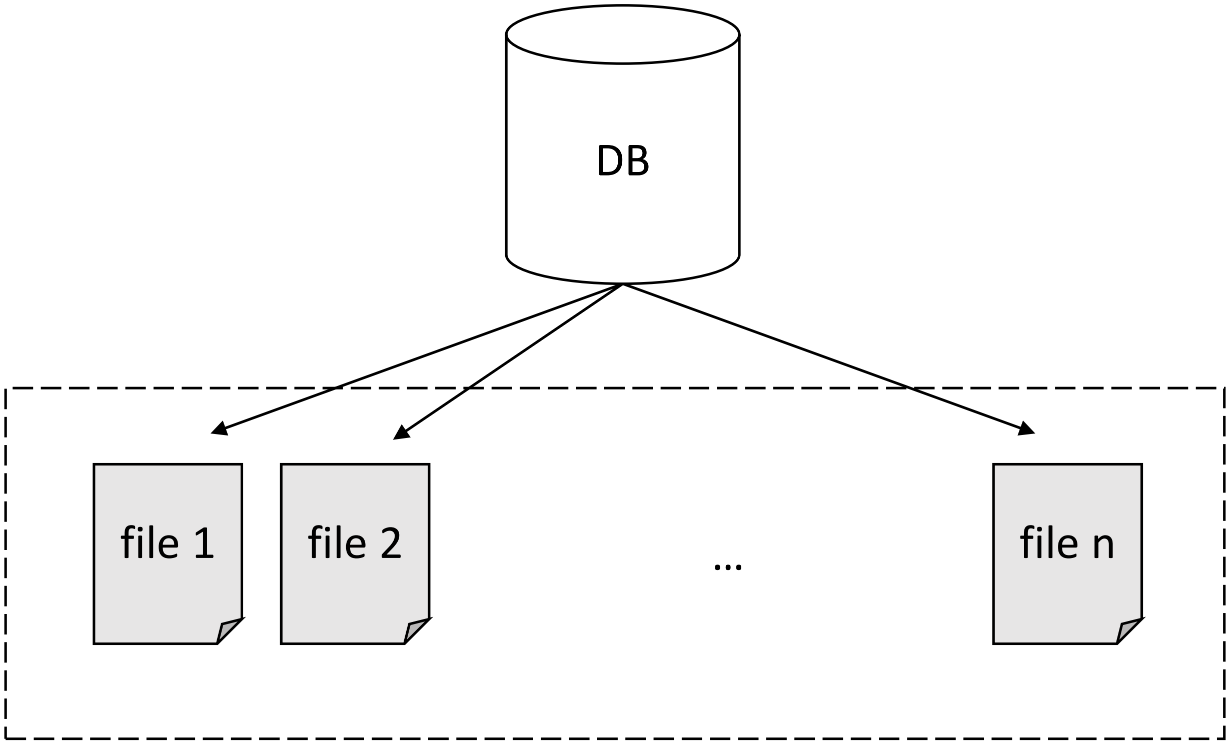






Relational Algebra
SQL

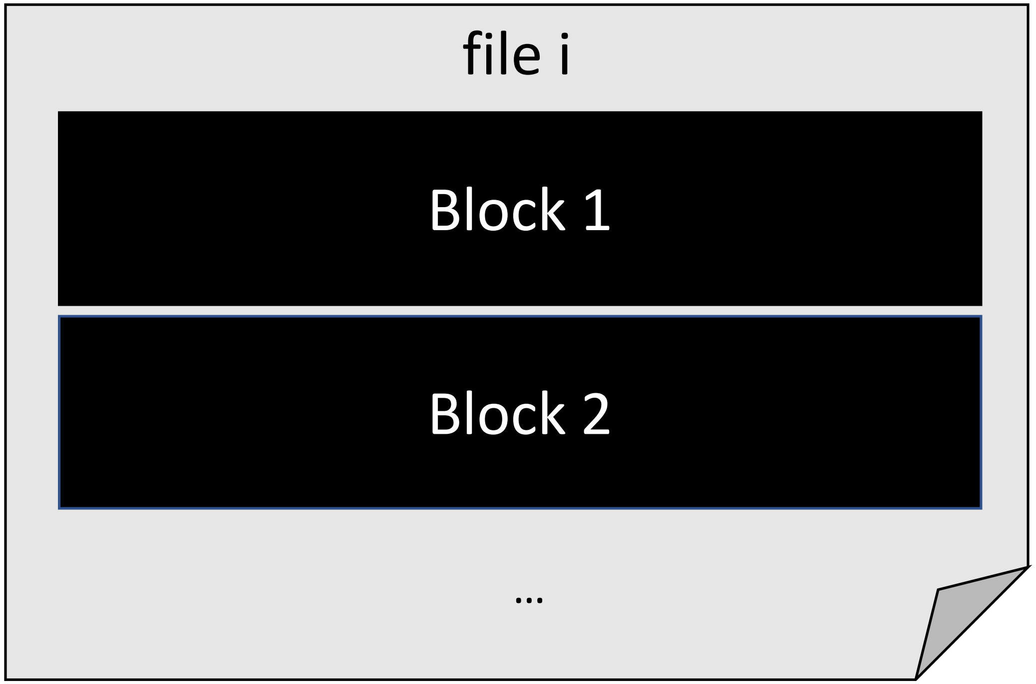
File Organization



Data stored as files.
Files are managed by the
underlying OS.

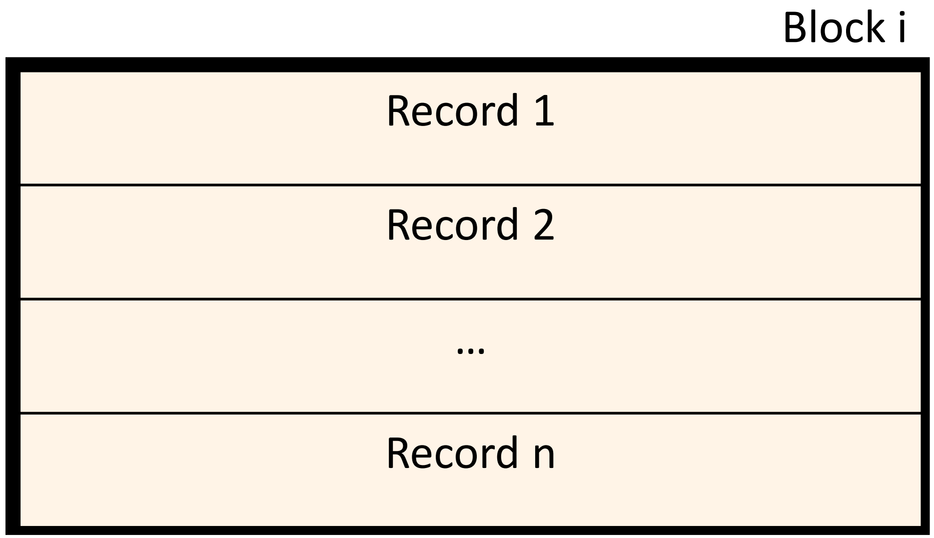
Files

- A **file** is a sequence of **blocks**.
- **Blocks** are fixed-length units of both storage allocation and data transfer.

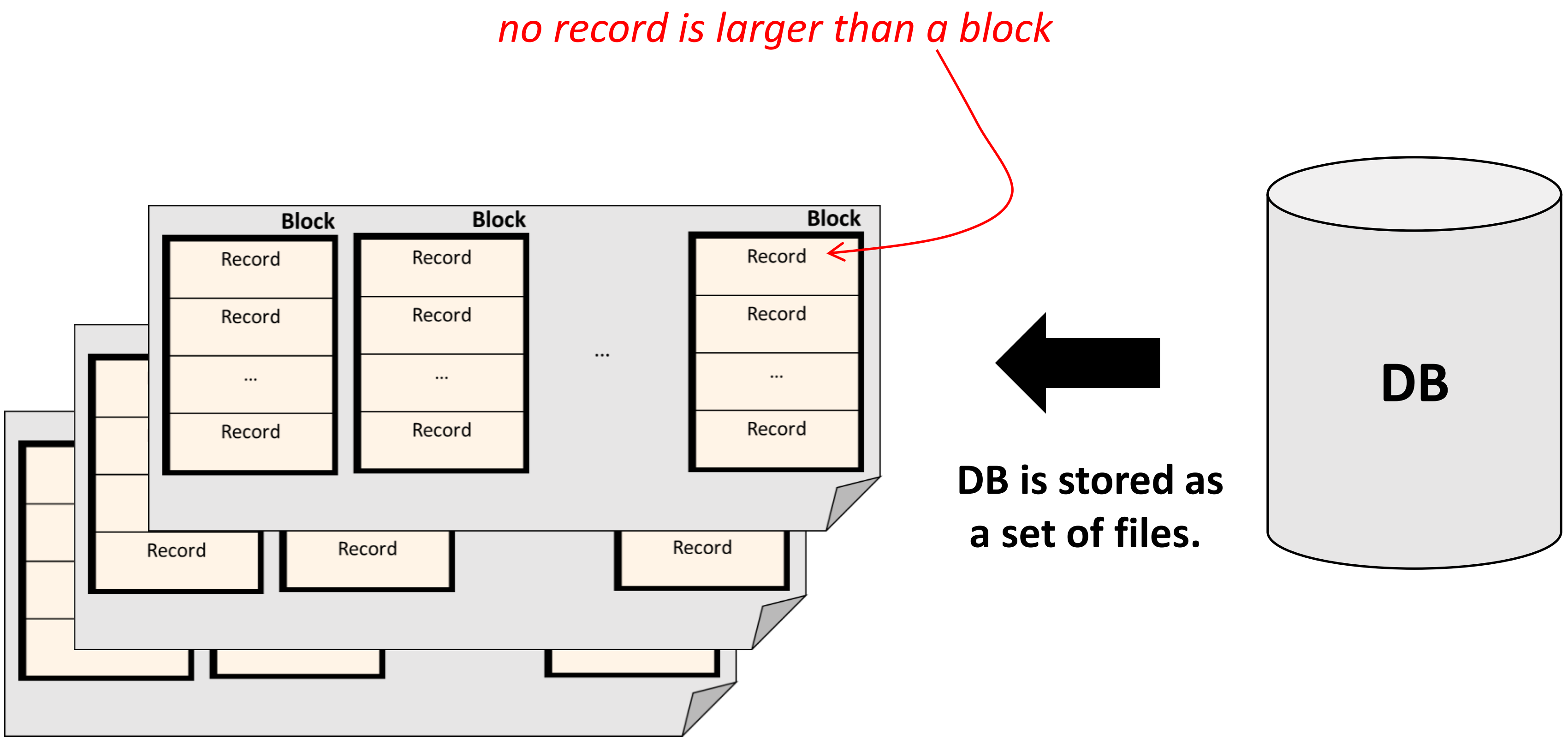


Records

- A **block** may contain several **records.**
- Each **record** is entirely contained in a single **block**.



File Organization



Approch1: Fixed-Length Records

type *instructor* = **record**
ID **varchar** (5);
name **varchar**(20);
dept_name **varchar** (20);
salary **numeric** (8,2);
end

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Quiz

- Assume each char takes 1 byte and numeric(8,2) type take 8 bytes of physical storage. Say, block size in our file system is 1 KB. If there are 20 records in our relation, how many block accesses will we need to retrieve all of them?

```
type instructor = record
        ID varchar (5);
        name varchar(20);
        dept_name varchar (20);
        salary numeric (8,2);
    end
```

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

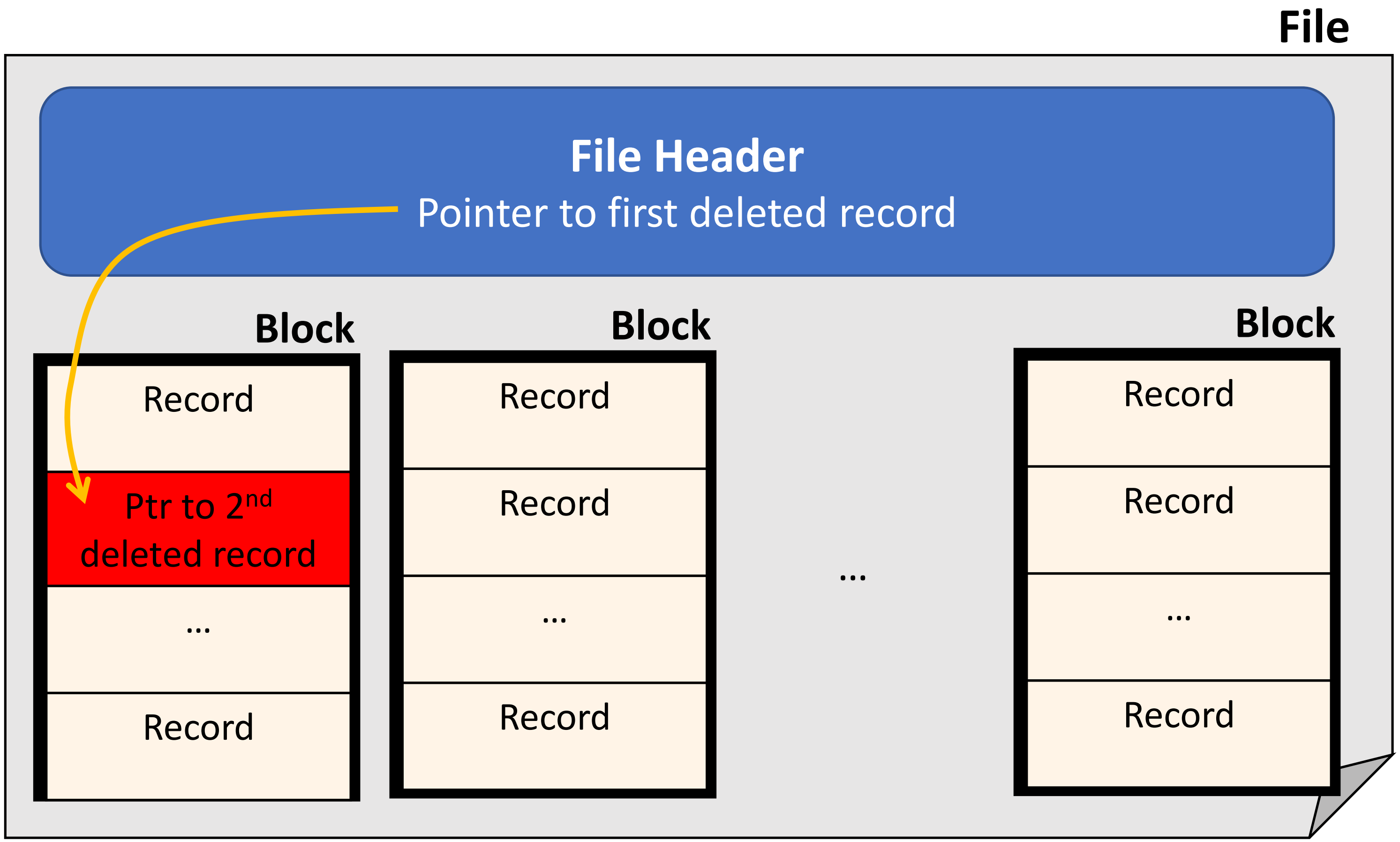
Quiz

- Assume each char takes 1 byte and numeric(8,2) type take 8 bytes of physical storage. Say, block size in our file system is 1 KB. If there are 20 records in our relation, how many block accesses will we need to retrieve all of them?
  - Record length = 53 bytes
  - Total no. of records = 20
  - Space required = 53 * 20 = 1060 bytes
  - Block size = 1024 bytes.
  - We need two block accesses to retrieve all records.

Issues

- Deletion
  - Causes gaps inside blocks.
- Space optimization
  - block size may not be a multiple of record length
    - space wasted in blocks.

Space Usage



Deleted records form a linked list called the "free list".

Free List

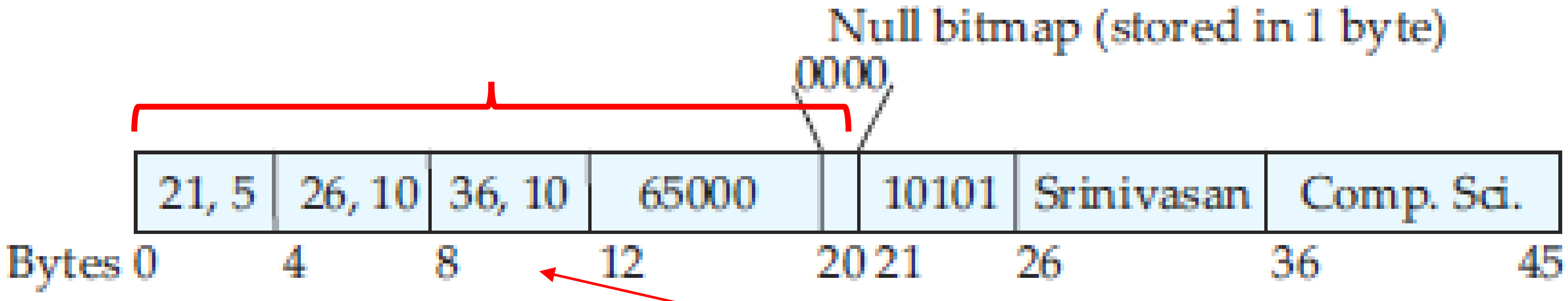
header				
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1				
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Free list

1 → 4 → 6

Variable Length Record

Metadata about the variable length data is stored (in fixed length part)



Read 10 bytes from 36th byte for this field

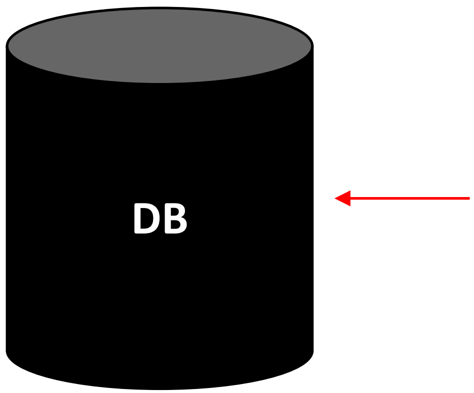
Storage Organization of Records

- **Heap file organization**
  - Place any record anywhere in the file.
  - Single file for each relation.
- **Sequential file organization**
  - Records are stored in sequential order (of key).
- **Hashing file organization**
  - Hash (some attribute of) records to blocks.

Indexing

Motivation

- We usually access only a small part of the DB.



Find the instructors in the physics department

Need additional structures to access data efficiently

Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library
- **Search Key** - 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:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
  - 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 the file.
- **Index-sequential file:** ordered sequential file with a primary index.

Dense Index Files

- **Dense index** — Index record appears for every search-key value in the file.
- E.g. index on *ID* attribute of *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	

Dense Index Files (Cont.)

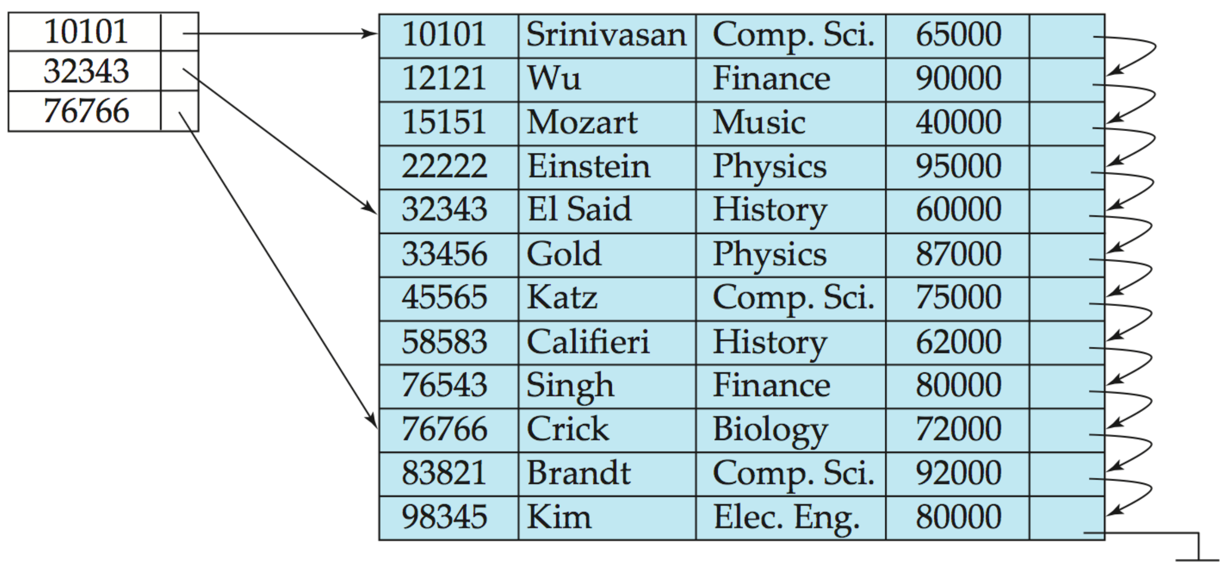
- Dense index on *dept_name*, with *instructor* file sorted on *dept_name*

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	

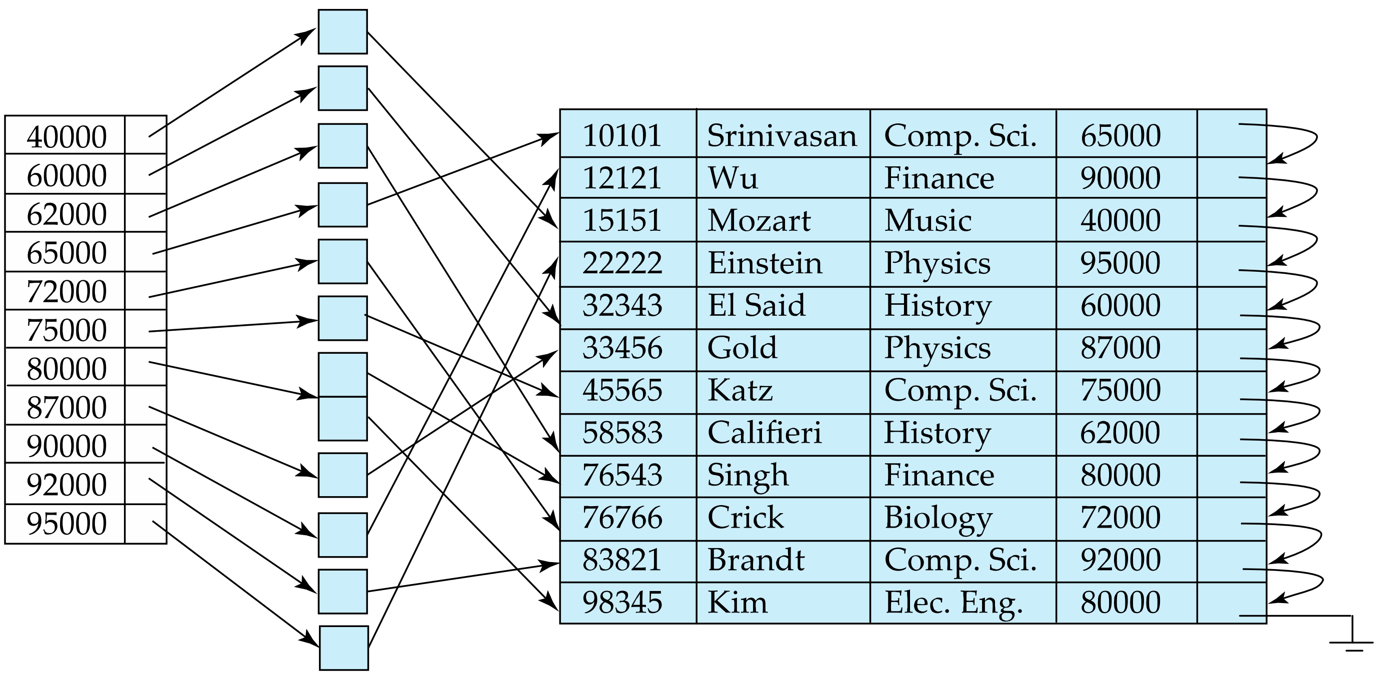
Sparse Index Files

- **Sparse Index**: contains index records for only some search-key values.
  - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value *K* we:
  - Find index record with largest search-key value < *K*
  - Search file sequentially starting at the record to which the index record points



Secondary Indices Example

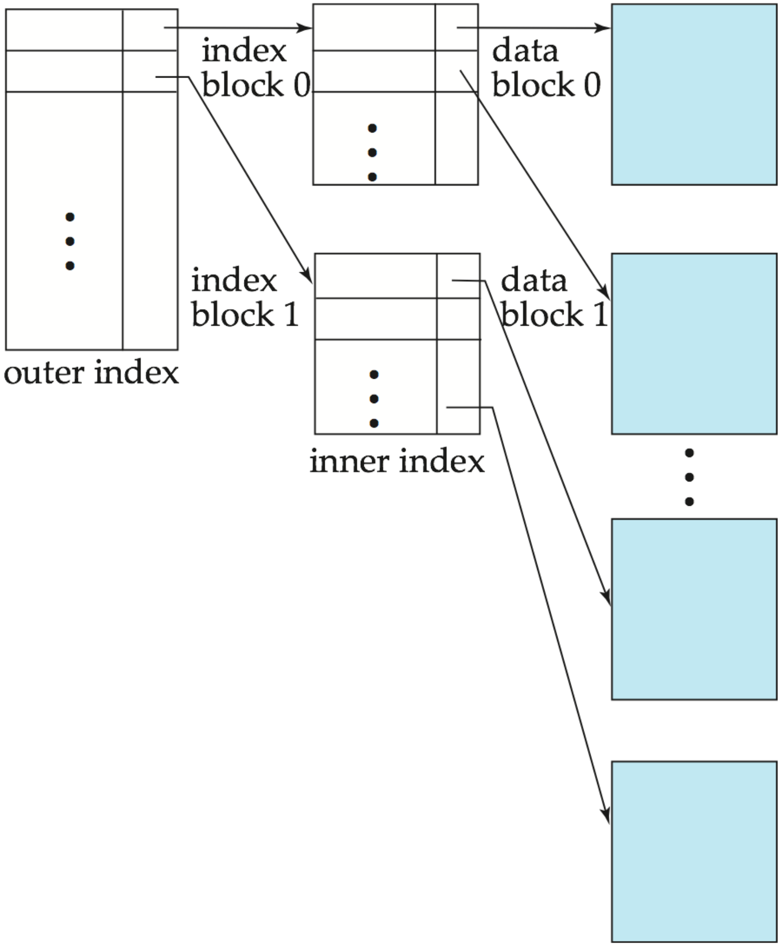
- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value.
- Secondary indices have to be dense



Secondary index on *salary* field of *instructor*

Multilevel Index

- If primary index does not fit in memory, access becomes expensive.

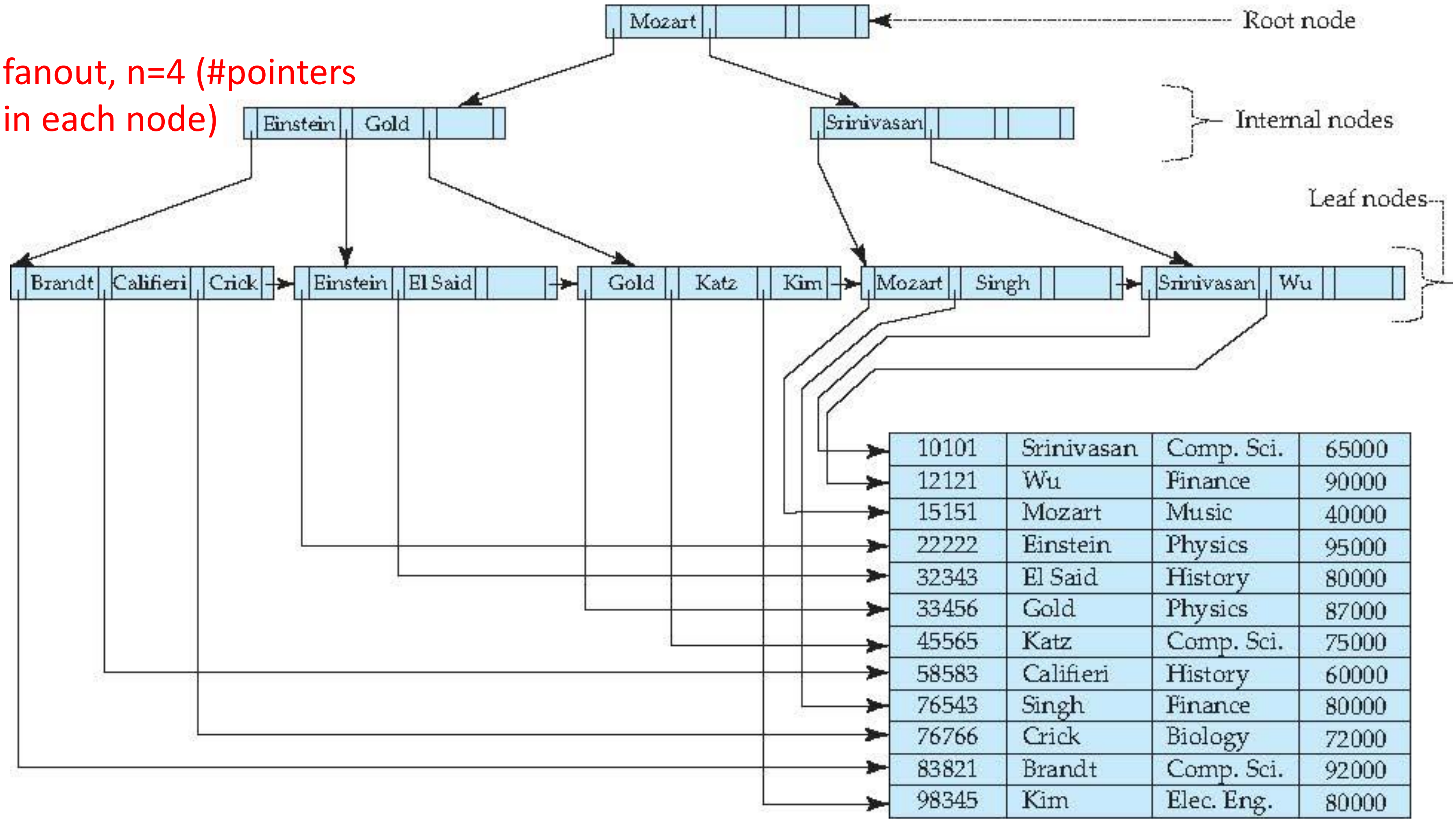


B^+-Tree Index Files

- B+-tree indices are an alternative to indexed-sequential files.
- Advantage of 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.

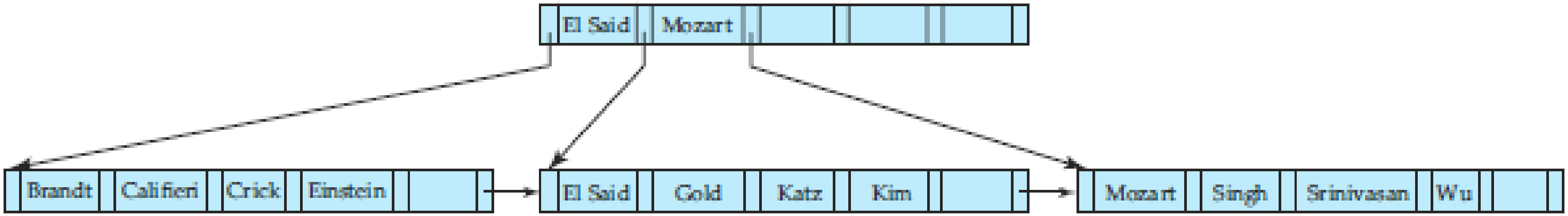
Example of B$^+$-Tree

fanout, n=4 (#pointers in each node)



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

n=6


El Said
Mozart
Brandt
Califieri
Crick
Einstein
El Said
Gold
Katz
Kim
Mozart
Singh
Srinivasan
Wu

B^+-Tree Node Structure

- Typical node

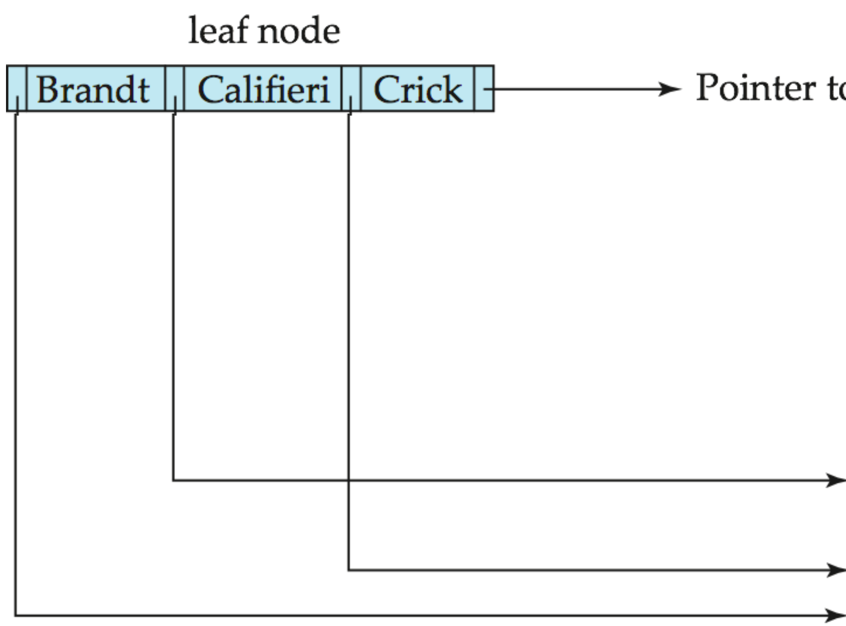
P_1	K_1	P_2	$\cdots$	P_{n-1}	K_{n-1}	P_n

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

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

Leaf Nodes in B$^+$-Trees

- For $i = 1, 2, \ldots, n-1$, pointer P_i points to a file record with search-key value K_i,
- 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_n points to next leaf node in search-key order

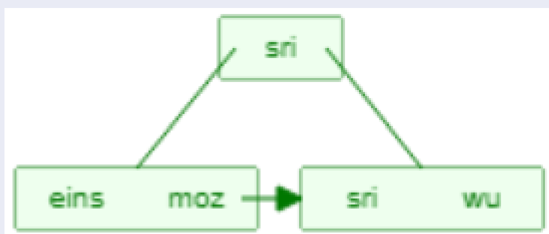
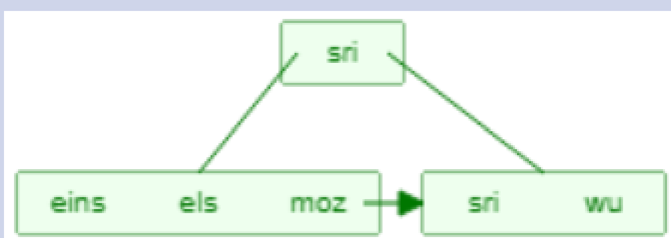


10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000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

Rules

- Root node
  - can hold fewer than $n/2$ pointers.
  - must hold at least two pointers, unless the tree consists of only one node.
- Internal nodes
  - all pointers are pointers to tree nodes.
  - and *must* hold at least $\lceil n/2 \rceil$ pointers and up to n pointers.
- Leaf nodes
  - Can contain from as few as $\lceil (n-1)/2 \rceil$ values, up to n-1 values

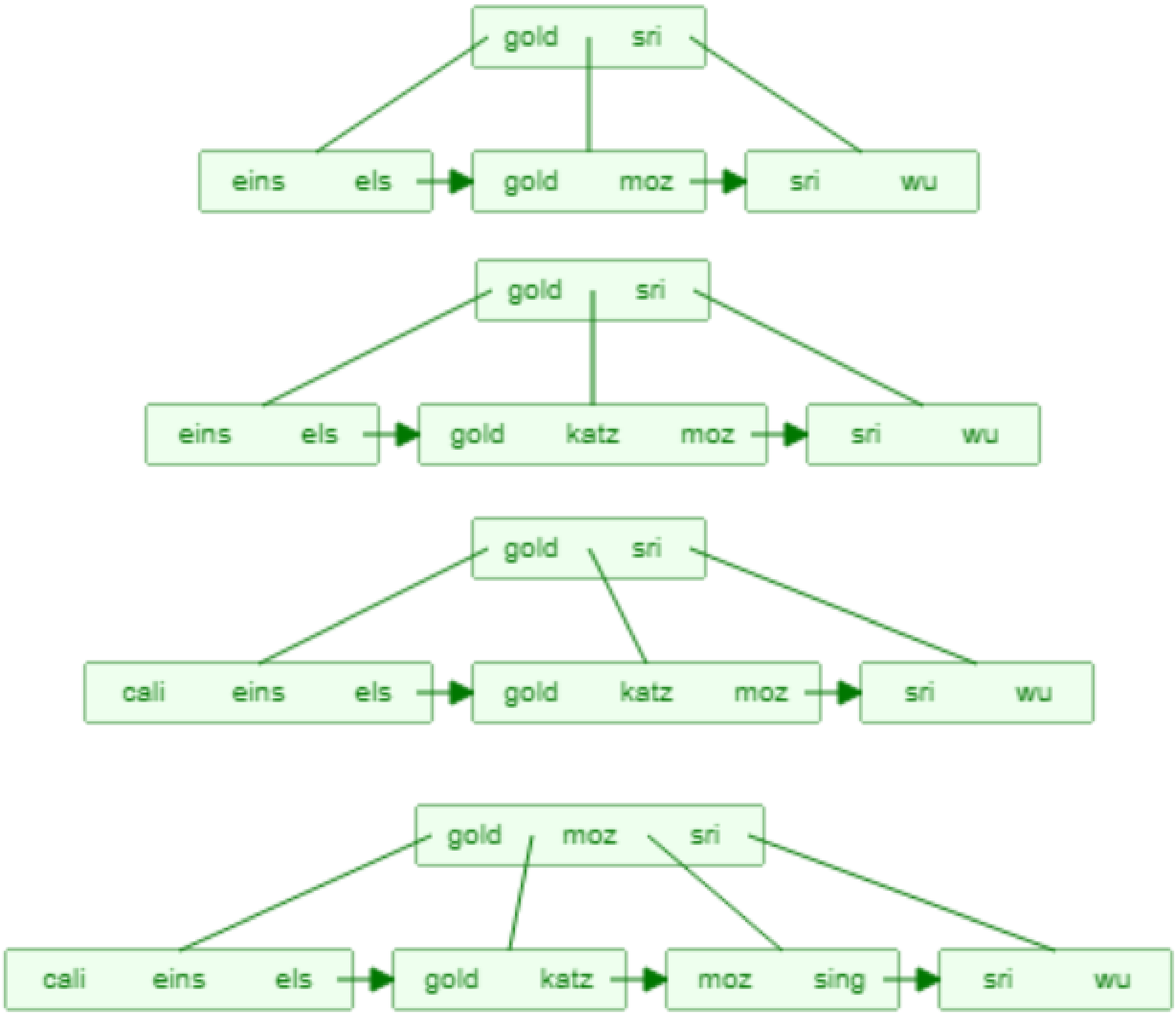
B+ Tree Construction

Insert	B+ Tree
sri	
wu	
moz	
ein	
els	

10101	Srinivasan
12121	Wu
15151	Mozart
22222	Einstein
32343	El Said
33456	Gold
45565	Katz
58583	Califieri
76543	Singh
76766	Crick
83821	Brandt
98345	Kim

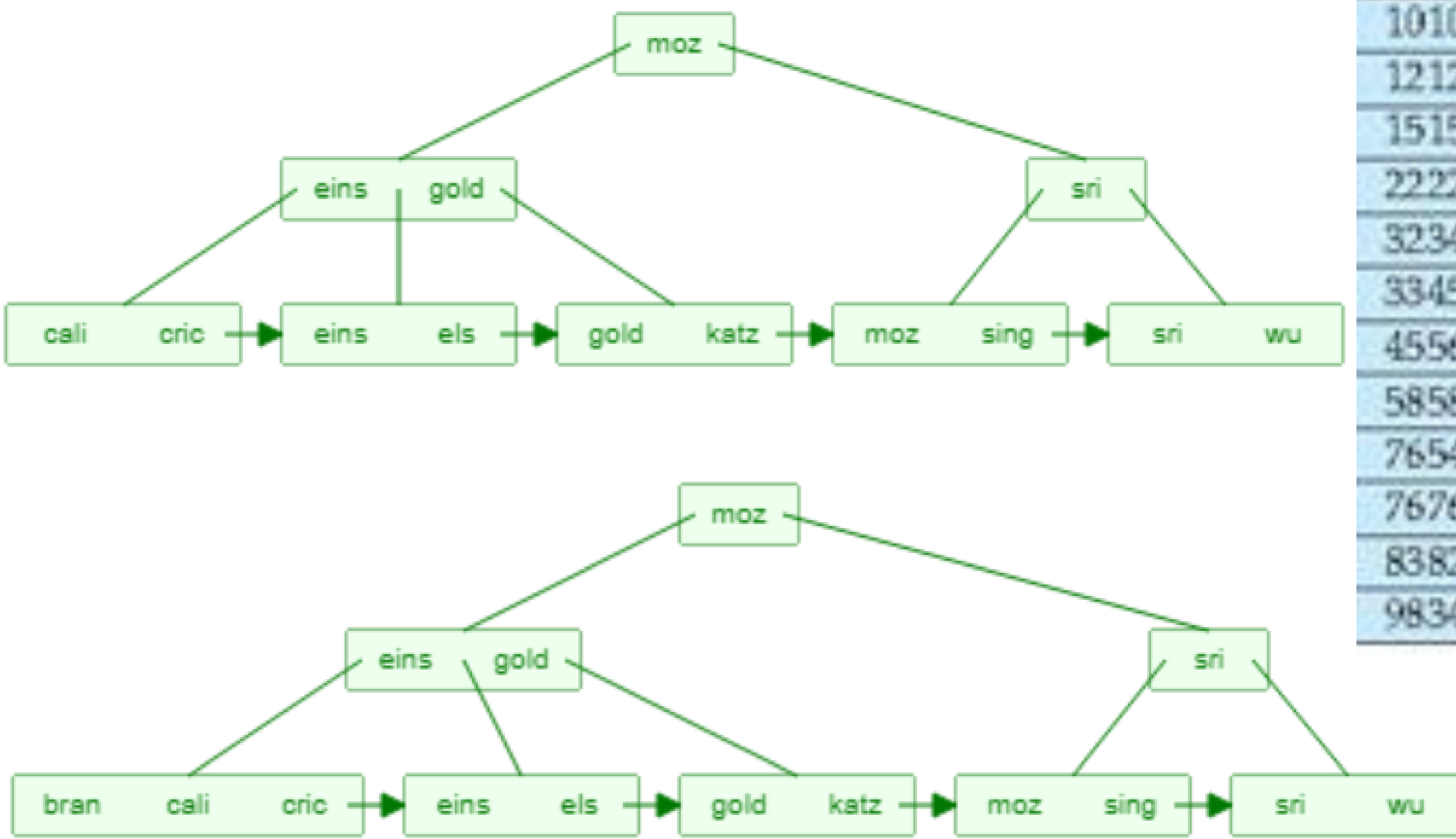
See
https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html

B+ Tree Construction



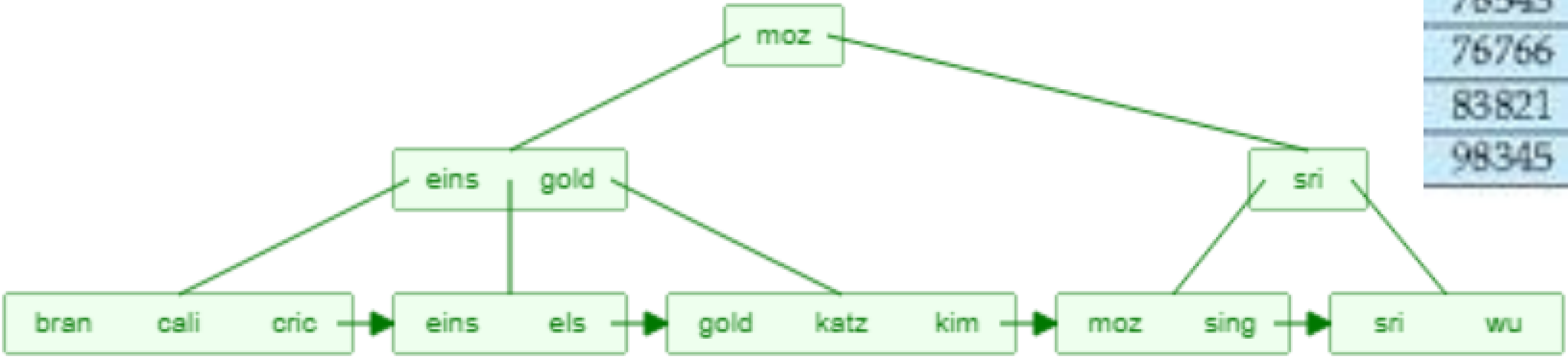
10101	Srinivasan
12121	Wu
15151	Mozart
22222	Einstein
32343	El Said
33456	Gold
45565	Katz
58583	Califieri
76543	Singh
76766	Crick
83821	Brandt
98345	Kim

B+ Tree Insertion



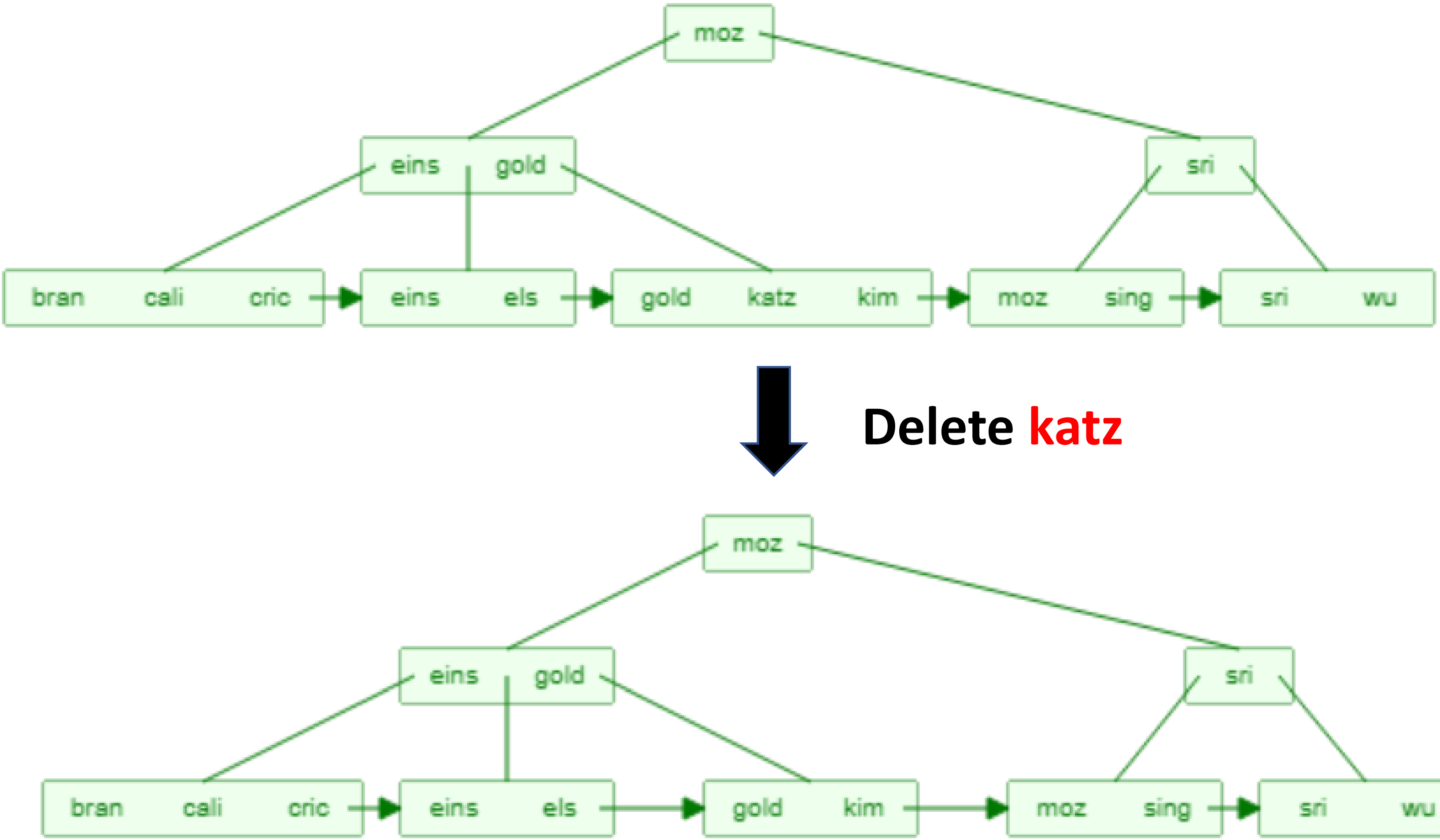
10101	Srinivasan
12121	Wu
15151	Mozart
22222	Einstein
32343	El Said
33456	Gold
45565	Katz
58583	Califieri
76543	Singh
76766	Crick
83821	Brandt
98345	Kim

B+ Tree Insertion

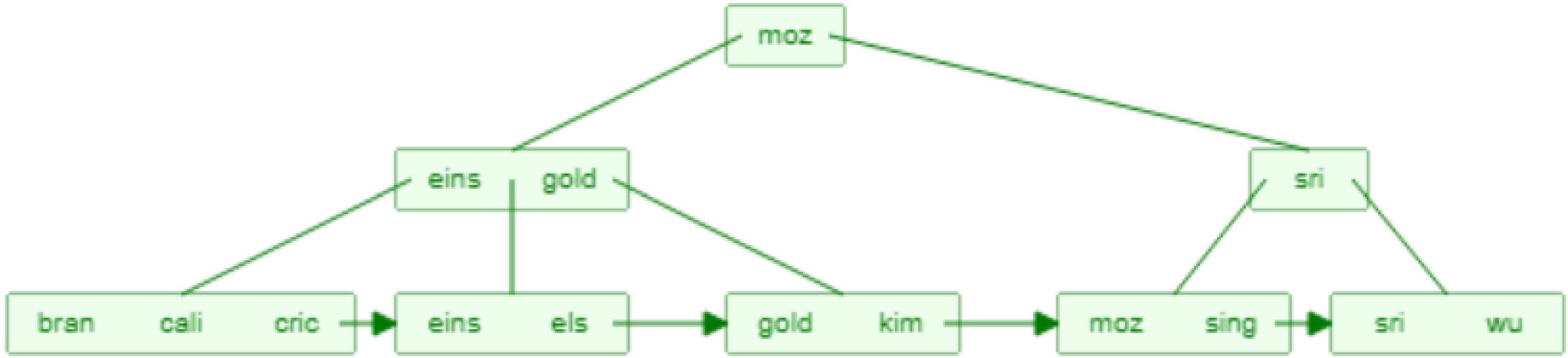


10101	Srinivasan
12121	Wu
15151	Mozart
22222	Einstein
32343	El Said
33456	Gold
45565	Katz
58583	Califieri
76543	Singh
76766	Crick
83821	Brandt
98345	Kim

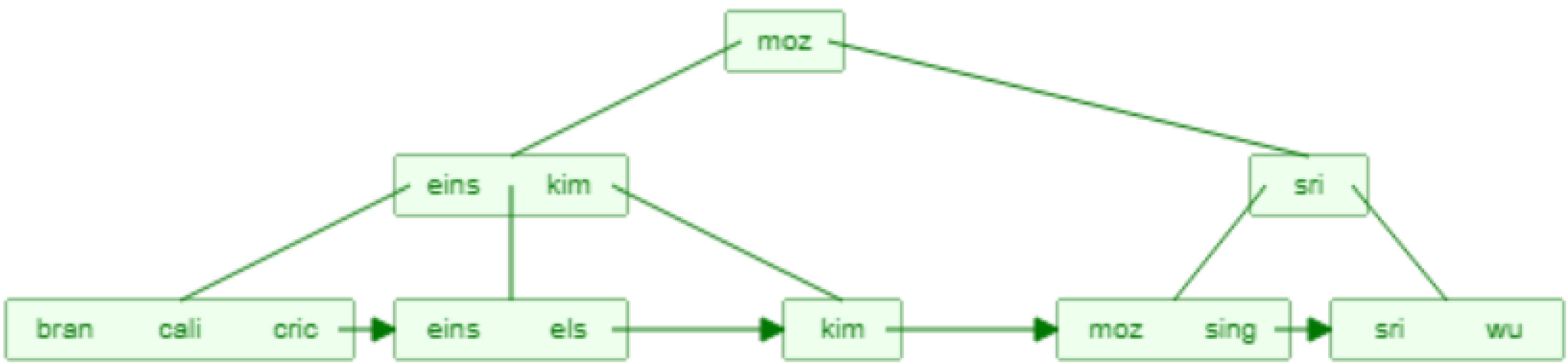
Deletion of a Key in a B+ Tree



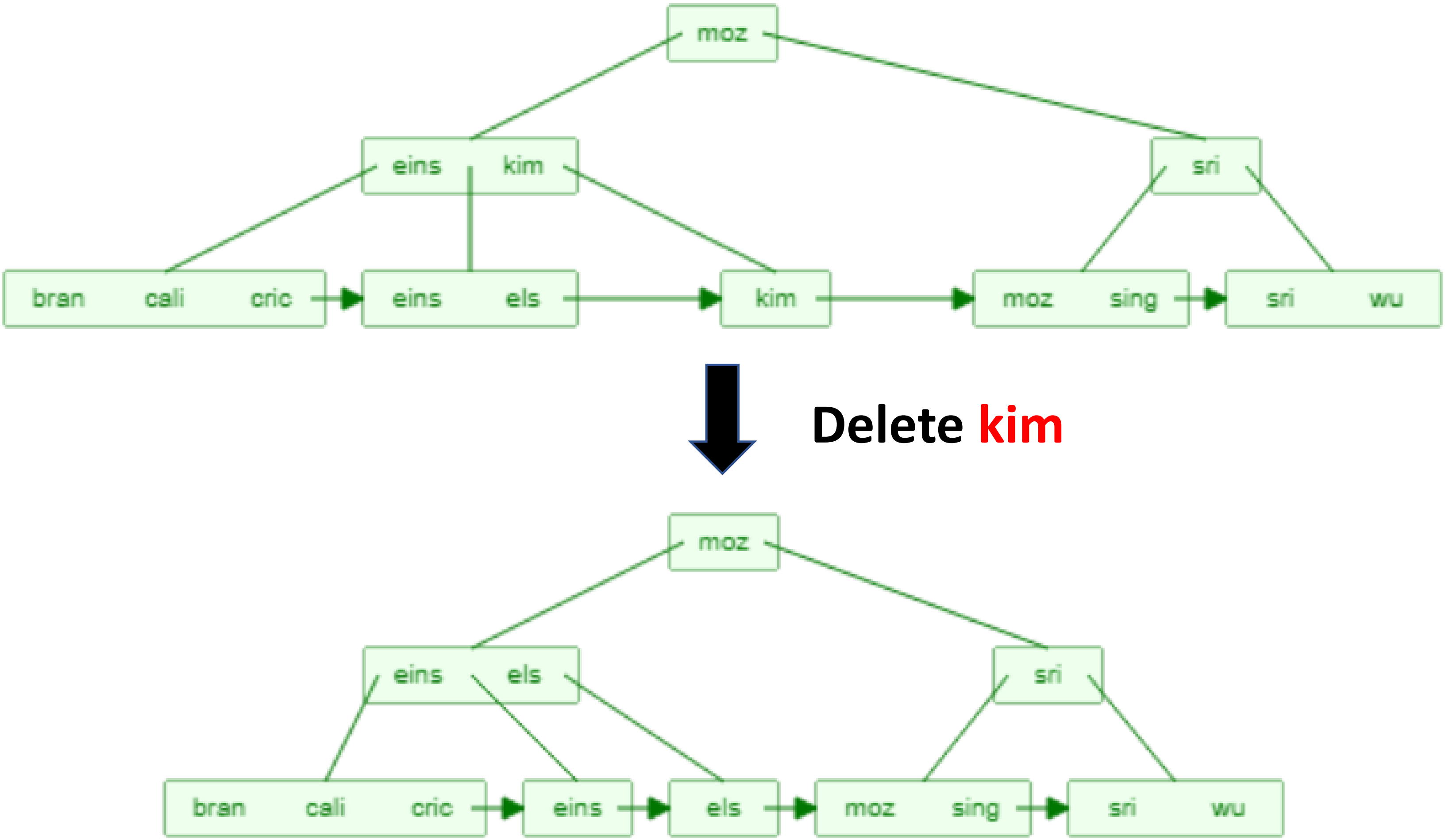
Deletion of a Key in a B+ Tree



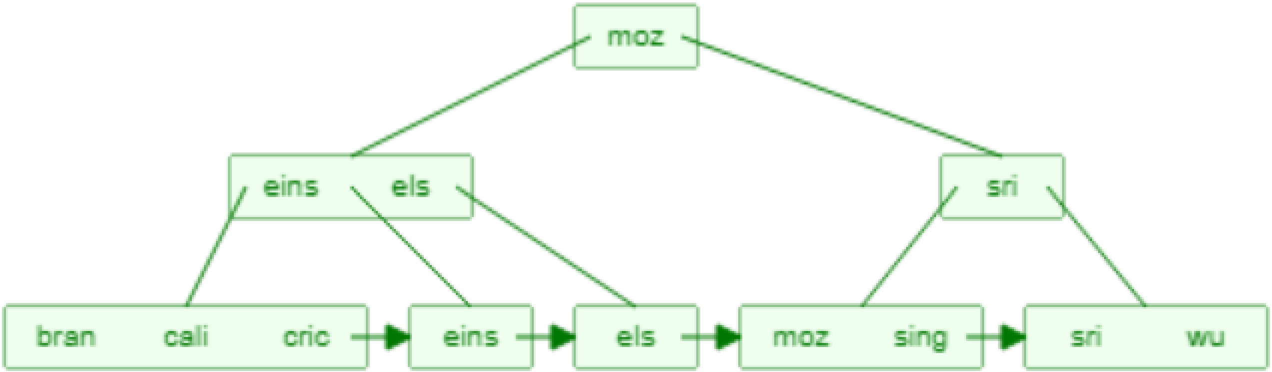
Delete gold



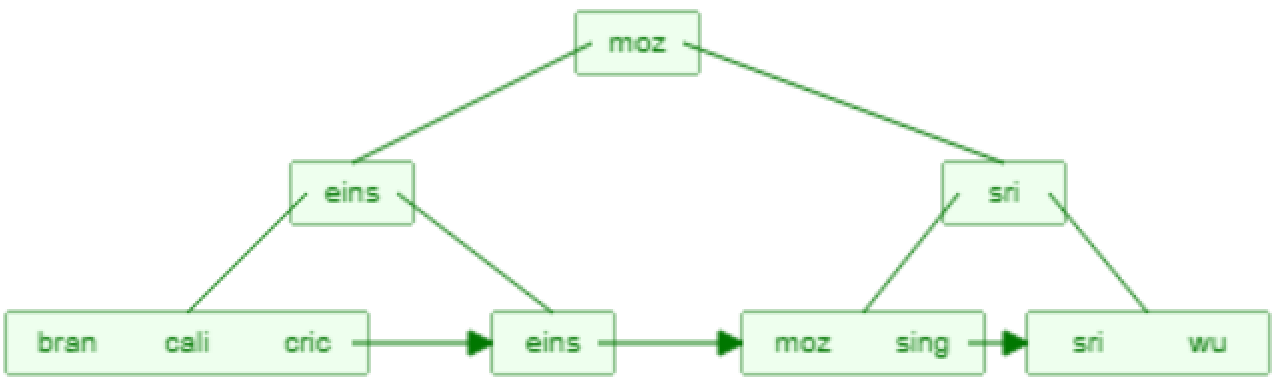
Deletion of a Key in a B+ Tree



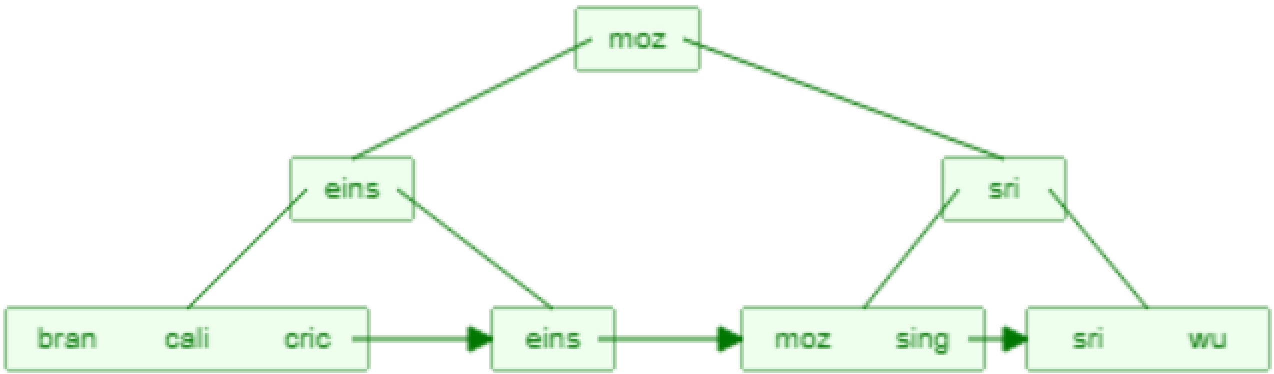
Deletion of a Key in a B+ Tree



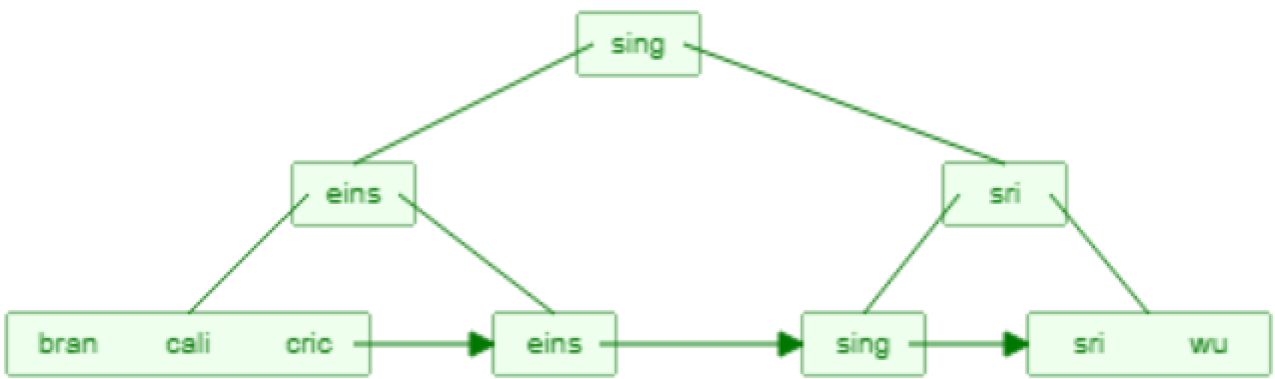
Delete els



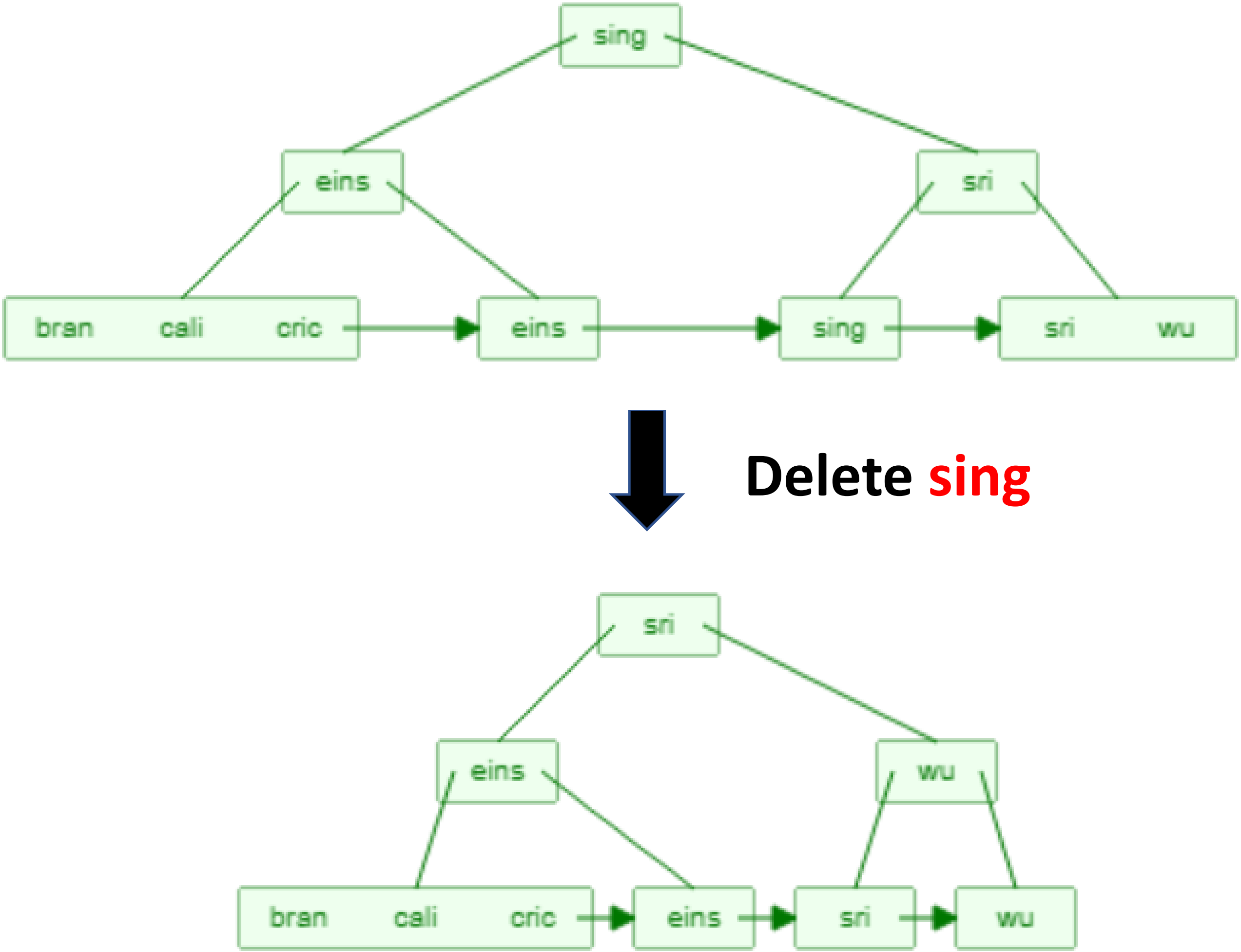
Deletion of a Key in a B+ Tree



Delete moz



Deletion of a Key in a B+ Tree



Readings

- Insert and Delete algorithms over B+Trees

```
procedure delete(value K, pointer P)
    find the leaf node L that contains (K, P)
    delete_entry(L, K, P)

procedure delete_entry(node N, value K, pointer P)
    delete (K, P) from N
    if (N is the root and N has only one remaining child)
    then make the child of N the new root of the tree and delete N
    else if (N has too few values/pointers) then begin
        Let N' be the previous or next child of parent(N)
        Let K' be the value between pointers N and N' in parent(N)
        if (entries in N and N' can fit in a single node)
            then begin /* Coalesce nodes */
                if (N is a predecessor of N') then swap_variables(N, N')
                if (N is not a leaf)
                    then append K' and all pointers and values in N to N'
                    else append all (K_i, P_i) pairs in N to N'; set N'.P_n = N.P_n
                delete_entry(parent(N), K', N); delete node N
            end
        else begin /* Redistribution: borrow an entry from N' */
            if (N' is a predecessor of N) then begin
                if (N is a nonleaf node) then begin
                    let m be such that N'.P_m is the last pointer in N'
                    remove (N'.K_{m-1}, N'.P_m) from N'
                    insert (N'.P_m, K') as the first pointer and value in N,
                        by shifting other pointers and values right
                    replace K' in parent(N) by N'.K_{m-1}
                end
                else begin
                    let m be such that (N'.P_m, N'.K_m) is the last pointer/value
```

```
procedure insert(value K, pointer P)
    if (tree is empty) create an empty leaf node L, which is also the r
    else Find the leaf node L that should contain key value K
    if (L has less than n - 1 key values)
        then insert_in_leaf (L, K, P)
        else begin /* L has n - 1 key values already, split it */
            Create node L'
            Copy L.P_1 ... L.K_{n-1} to a block of memory T that can
                hold n (pointer, key-value) pairs
            insert_in_leaf (T, K, P)
            Set L'.P_n = L.P_n; Set L.P_n = L'
            Erase L.P_1 through L.K_{n-1} from L
            Copy T.P_1 through T.K_{⌈n/2⌉} from T into L starting at L.
            Copy T.P_{⌈n/2⌉+1} through T.K_n from T into L' starting at
            Let K' be the smallest key-value in L'
            insert_in_parent(L, K', L')
        end

ocedure insert_in_leaf (node L, value K, pointer P)
    if (K < L.K_1)
        then insert P, K into L just before L.P_1
        else begin
            Let K_i be the highest value in L that is less than K
            Insert P, K into L just after T.K_i
        end

ocedure insert_in_parent(node N, value K', node N')
    if (N is the root of the tree)
        then begin
            Create a new node R containing N, K', N'   /* N and N
            Make R the root of the tree
```

Thank You

B+ Tree simulation available at
https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html