B^+ Tree and B Tree

Dr. Shrutilipi Bhattacharjee, Assistant Professor, Dept. of IT, NIT Karnataka, India

B^+ Tree Index Files

- B^+ tree indices are an alternative to indexed-sequential files

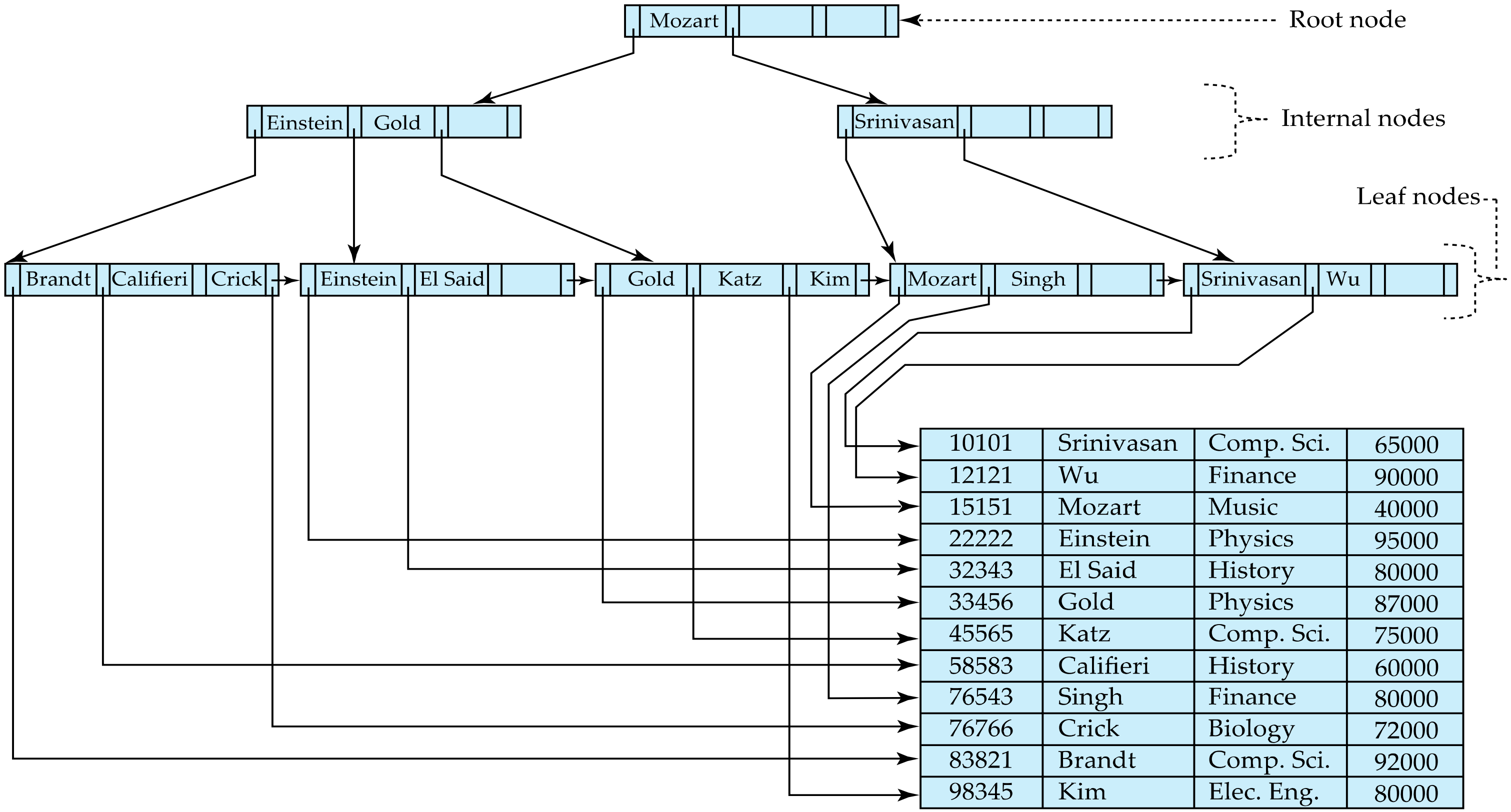
- Disadvantage of indexed-sequential files
  - Performance degrades as file grows, since many overflow blocks get created
  - Periodic reorganization of entire file is required

- Advantage of B^+ tree 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

- Advantages of B^+ trees outweigh disadvantages
  - B^+ trees are used extensively

Example of 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 Index Files

- 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 $n - 1$ key values
  - Special cases:
    - If the root is not a leaf, it has at least 2 children and maximum n children
    - If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and $(n - 1)$ key values

B^+ Tree Node Structure

- Typical node

P_1	K_1	P_2	…	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)

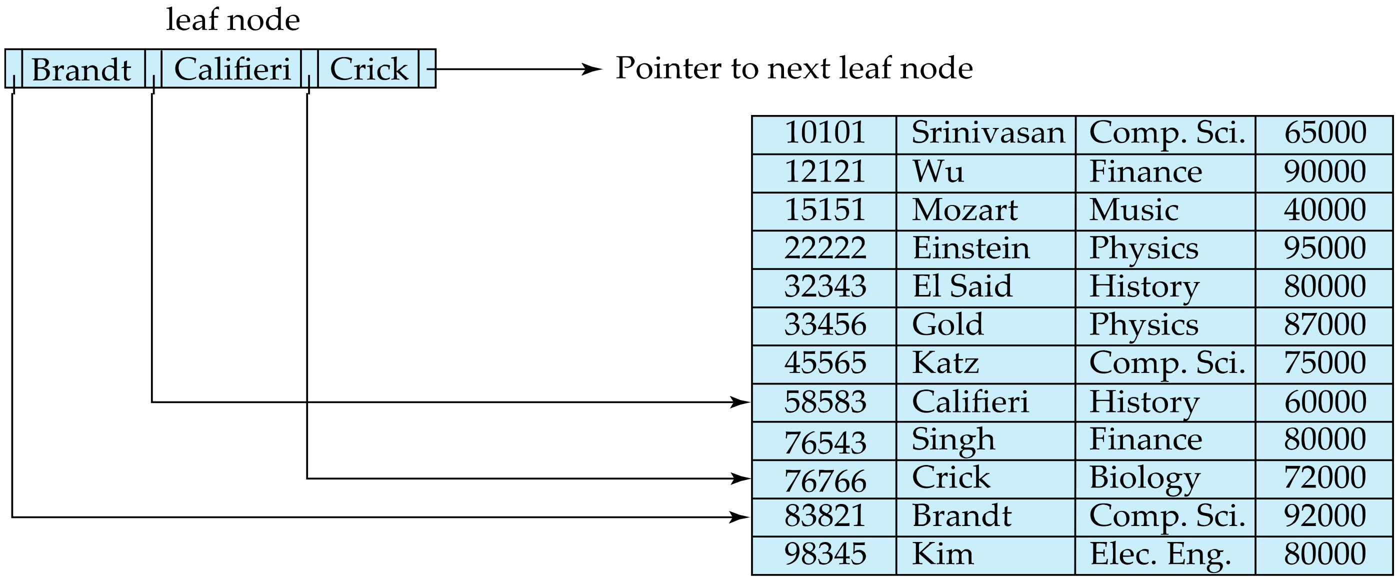
- The search-keys in a node are ordered

$$K_1 < K_2 < K_3 < . . . < K_{n–1}$$

(Initially assume no duplicate keys, address duplicates later)

Leaf Nodes in B+ Trees

- Properties of a leaf node:
  - 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

Non-Leaf Nodes in B^+ Trees

- Non leaf nodes form a multi-level sparse index on the leaf nodes

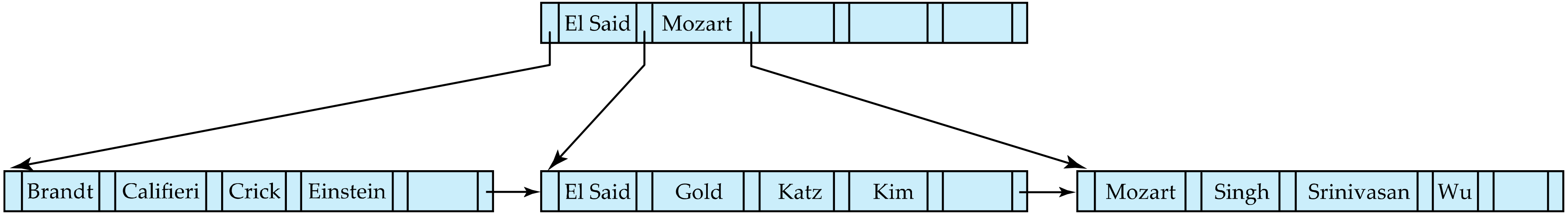
- For a non-leaf node with *m* pointers:
  - All the search-keys in the subtree to which P_1 points are less than K_1
  - For $2 \leq i \leq n - 1$, all the search-keys in the subtree to which P_i points have values greater than or equal to K_{i-1} and less than K_i
  - All the search-keys in the subtree to which P_n points have values greater than or equal to K_{n-1}

- General structure

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

Example of B+ Trees

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



  - Leaf nodes must have between 3 and 5 key 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

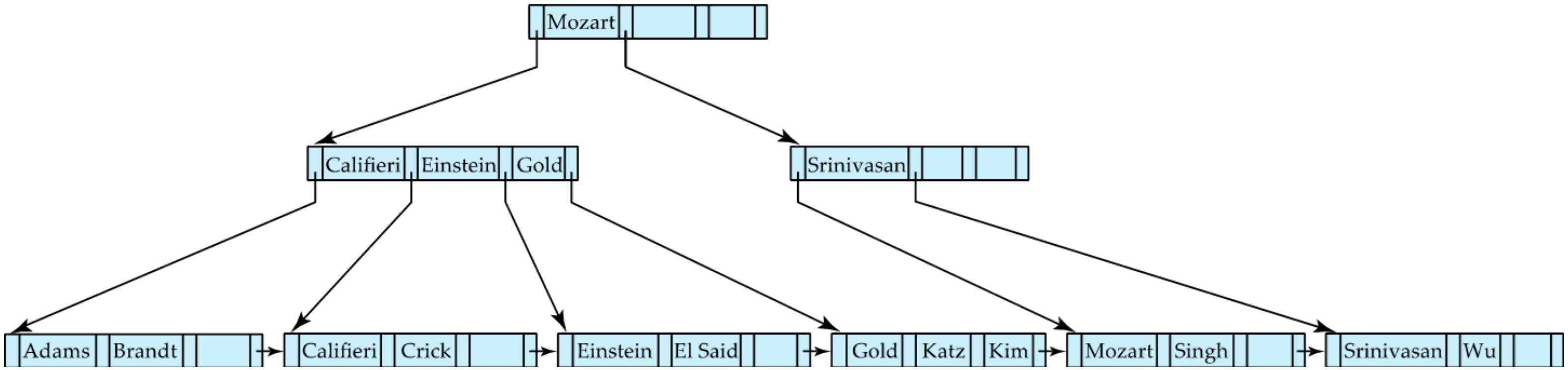
- The non-leaf levels of the B$^+$ tree form a hierarchy of sparse indices

- 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 K search-key values in the file, the tree height is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$
  - 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

Queries on B$^+$ Trees

function *find*(*V*)
  C = root
  while (C is not a leaf node) {
    Let *i* be least number s.t. $V \leq K_i$
    If no such exists, set *C* = *last non-null pointer in C*
    Else { **if** $V = C.K_i$) Set C = P_{i+1} **else set** $C = C.P_i$ }
  }
  Let *i* be least value s.t., $K_i = V$
  If there is such a value *i*, follow pointer P_i to the desired record
  else no record with search-key value *k* exists



Handling Duplicates

- With duplicate search keys
  - In both leaf and internal nodes
    - We cannot guarantee that $K_1 < K_2 < K_3 <. . .< K_{N-1}$
    - But can guarantee $K_1 \leq K_2 \leq K_3 \leq. . .\leq K_{N-1}$
  - Search-keys in the subtree to which P_i points
    - Are $\leq K_i$, but not necessarily $< K_i$
    - To see why, suppose same search key value V is present in two leaf node L_i and L_{i+1}
    - Then in parent node K_i must be equal to V

Handling Duplicates

- We modify find procedure as follows
  - Traverse P_i even if $V = K_i$
  - As soon as we reach a leaf node C check if C has only search key values less than V
  - If so set C = right sibling of C before checking whether C contains V
- Procedure printAll
  - Uses modified find procedure to find first occurrence of V
  - Traverse through consecutive leaves to find all occurrences of V

Queries on B$^+$ Trees

- 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 kilobytes
  - And N is typically around 100 (40 bytes per index entry)
- With 1 million search key values and N = 100
  - At most log_{50}(1,000,000) = 4 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

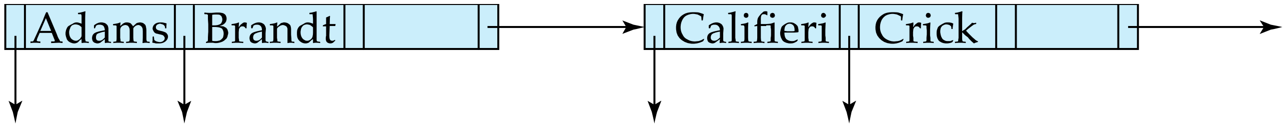
Updates on B^+ Trees: Insertion

- Find the leaf node in which the search-key value would appear
- If the search-key value is already present in the leaf node
  - Add record to the file
  - If necessary add a pointer to the bucket
- If the search-key value is not present, then
  - Add the record to the main file (and create a bucket if necessary)
  - If there is room in the leaf node, insert (key-value, pointer) pair in the leaf node
  - Otherwise, split the node (along with the new (key-value, pointer) entry) as discussed in the next slide

Updates on B$^+$ Trees: Insertion

- 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

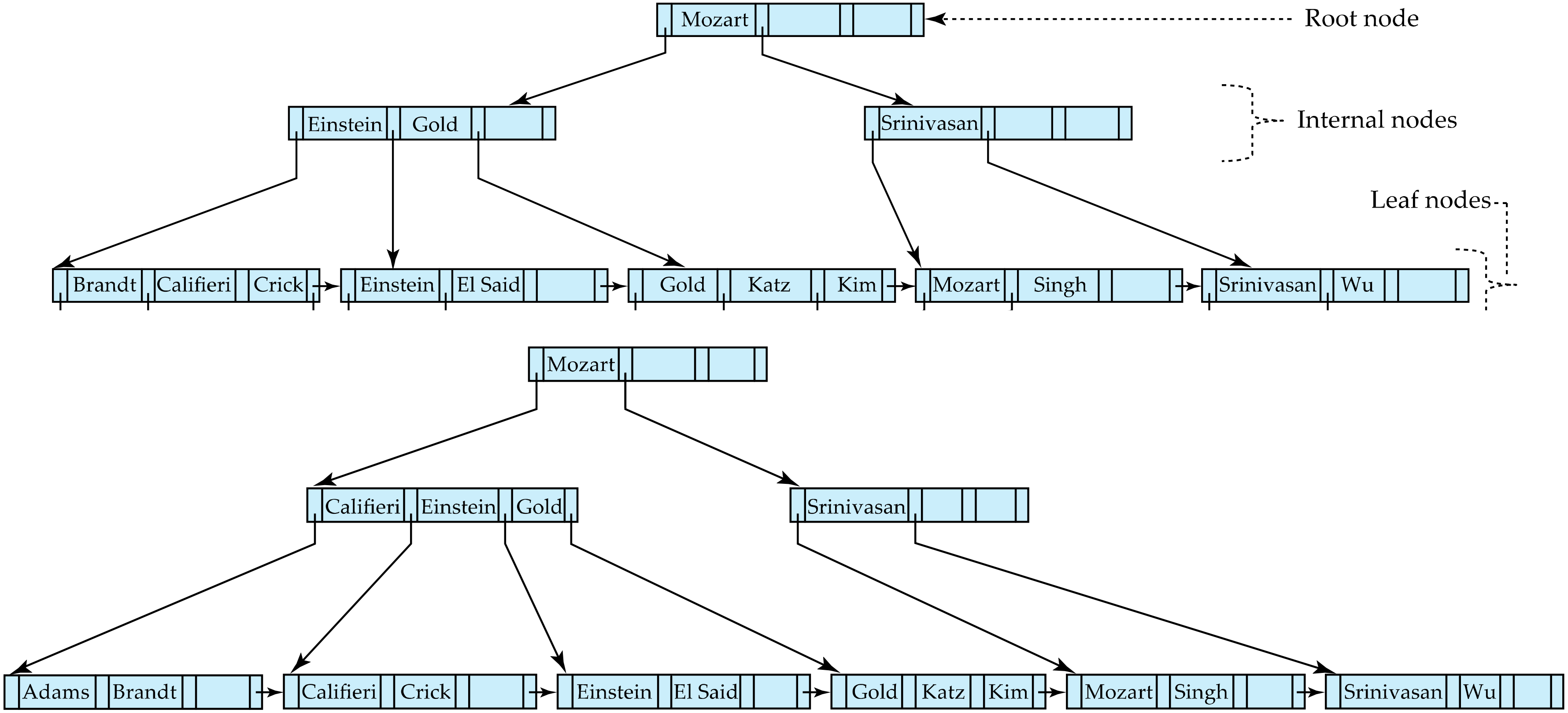
- Splitting of nodes proceeds upwards till a node that is not full is found
  - In the worst case the root node may be split increasing the height of the tree by 1



  - Result of splitting node containing *Brandt*, *Califieri* and *Crick* on inserting *Adams*
  - Next step: insert entry with (*Califieri*, pointer-to-new-node) into parent

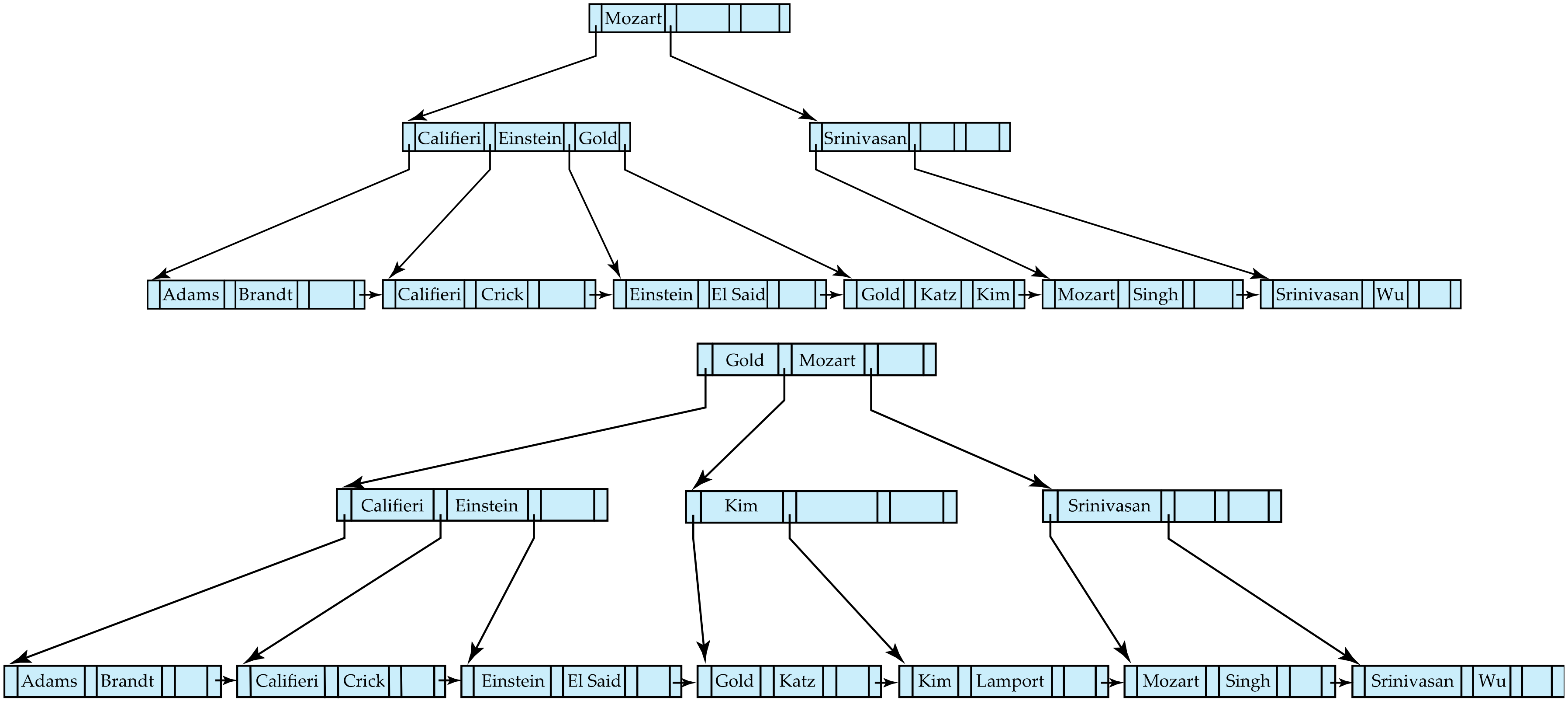
B$^+$ Trees Insertion

- B$^+$ tree before and after insertion of "Adams"



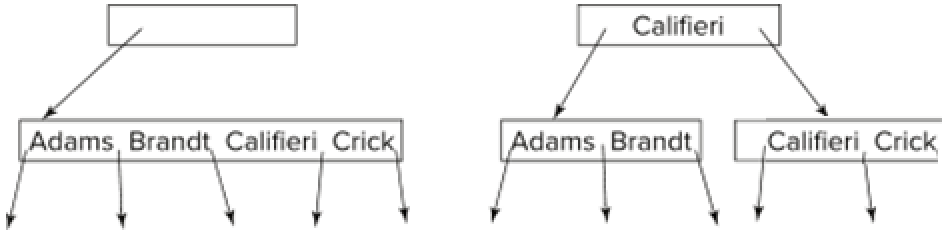
B+ Trees Insertion

- B+ tree before and after insertion of “Lamport”



Insertion in B^+ Trees

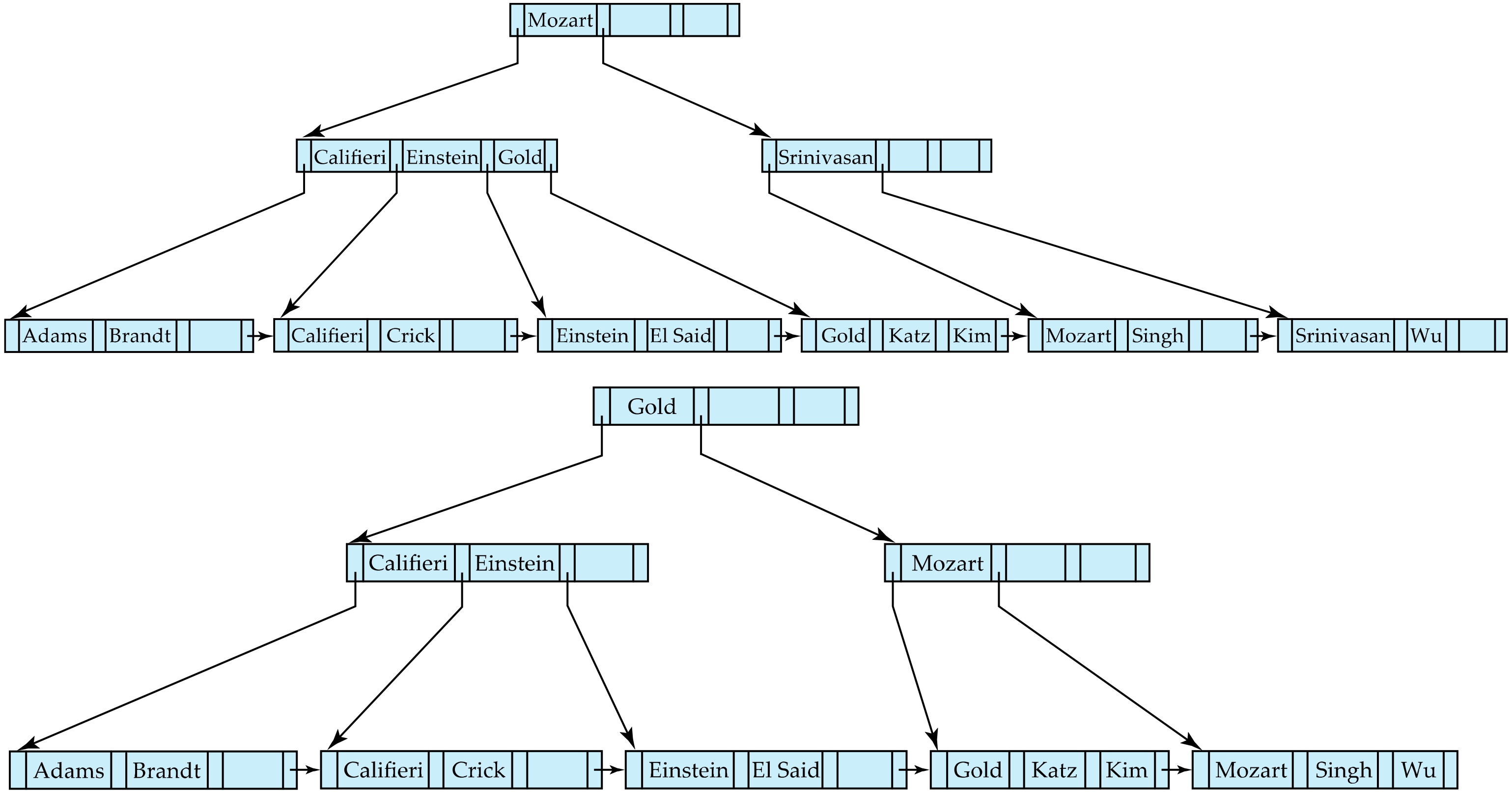
- Splitting a non-leaf node: When inserting (*k, p*) into an already full internal node *N*
  - Copy *N* to an in-memory area *M* with space for *n + 1* pointers and *n* keys
  - Insert (*k, p*) into M
  - Copy $P_1, K_1, \ldots, K_{\lceil n/2 \rceil - 1}, P_{\lceil n/2 \rceil}$ from M back into node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



Read pseudocode in book!

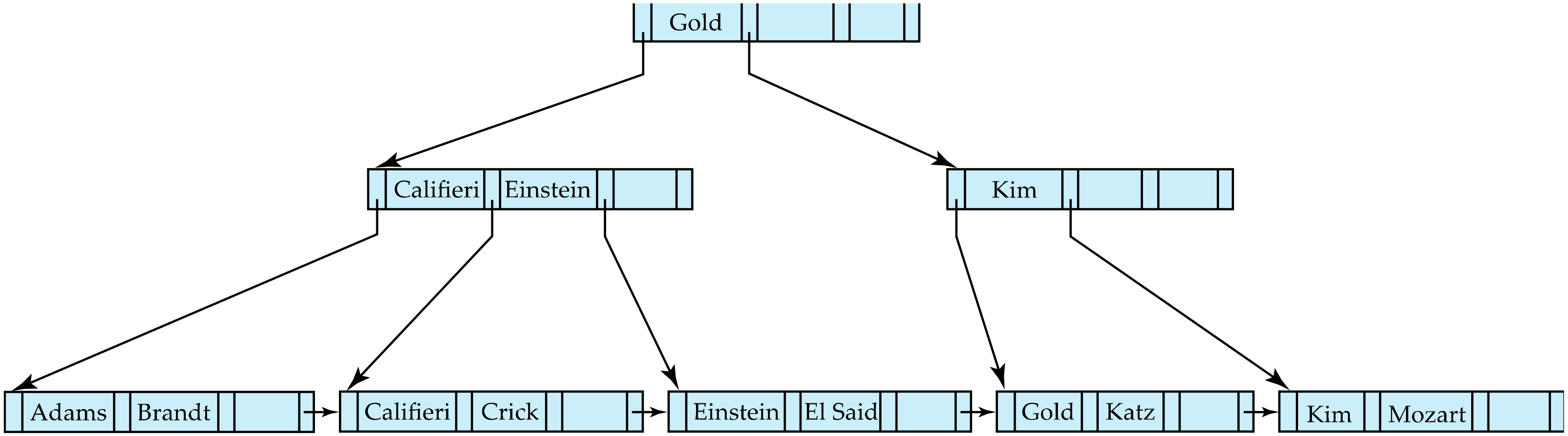
Examples of B+ Trees Deletion

- Deleting "*Srinivasan*" causes merging of under-full leaves



Examples of B+ Trees Deletion

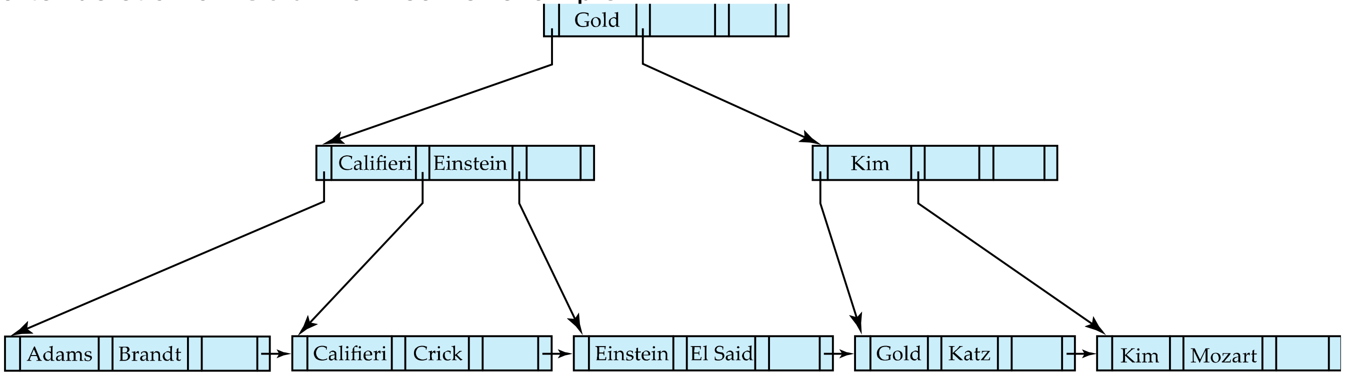
- Deletion of "*Singh*" and "*Wu*" from result of previous example

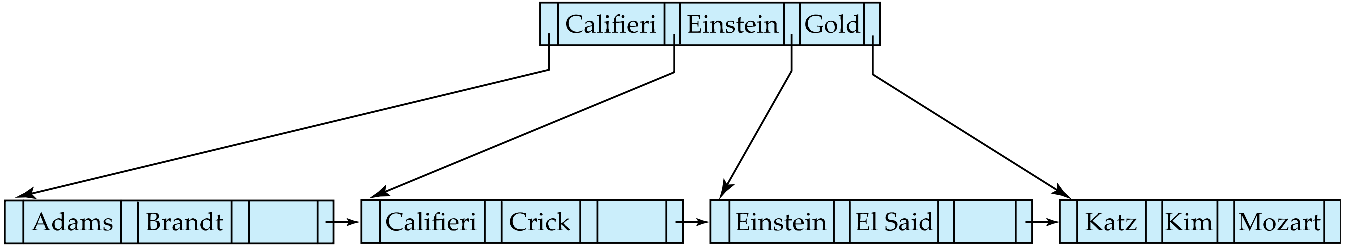


  - Leaf containing *Singh* and *Wu* became underfull, and borrowed a value *Kim* from its left sibling
  - Search-key value in the parent changes as a result

Examples of B$^+$ Trees Deletion

- Before and after deletion of "*Gold*" from earlier example





  - Node with *Gold* and *Katz* became underfull, and was merged with its sibling
  - Parent node becomes underfull, 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 delete

Updates on B$^+$ Trees: Deletion

- Find the record to be deleted, and remove it from the main file and from the bucket (if present)

- Remove (search-key value, pointer) from the leaf node if there is no bucket or if the bucket has become empty

- If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then ***merge siblings:***
  - Insert all the search-key values in the two nodes into a single node (the one on the left), and delete the other node
  - Delete the pair (K_{i-1}, P_i), where P_i is the pointer to the deleted node, from its parent, recursively using the above procedure

Updates on B$^+$ Trees: Deletion

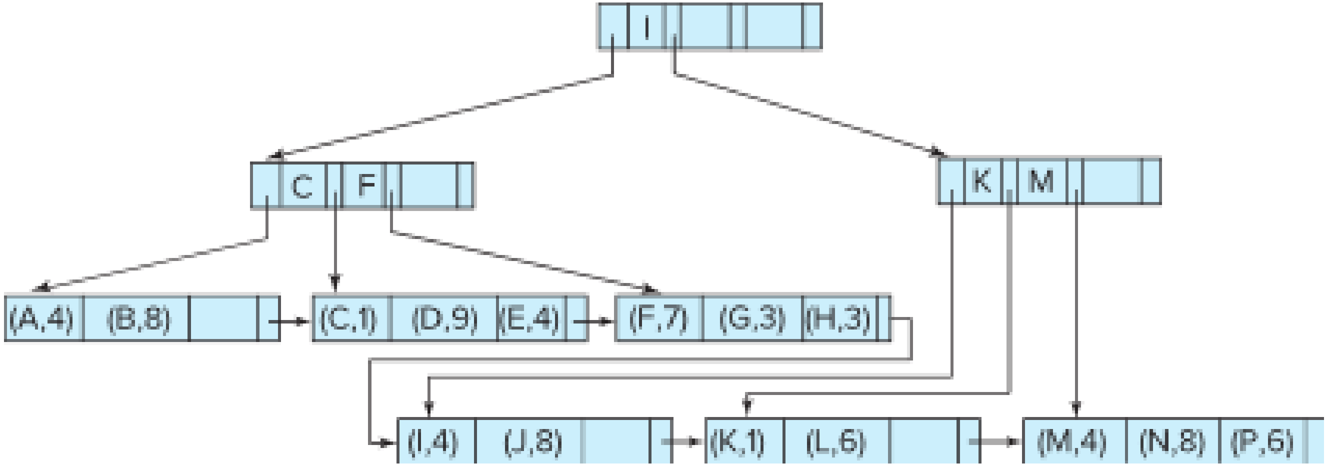
- Otherwise, if the node has too few entries due to the removal, but the entries in the node and a sibling do not fit into a single node, then ***redistribute pointers***:
  - Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries
  - Update the corresponding search-key value in the parent of the node
- The node deletions may cascade upwards till a node which has $\lceil n/2 \rceil$ 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

B^+ Tree File Organization

- Index file degradation problem is solved by using B^+ tree indices
- Data file degradation problem is solved by using B^+ tree file organization
- The leaf nodes in a B^+ tree file organization store records, instead of pointers
- Leaf nodes are still required to be half full
  - Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a non-leaf node
- Insertion and deletion are handled in the same way as insertion and deletion of entries in a B^+ tree index

B+ Tree File Organization

- Example of B+ tree file organization



- Good space utilization important since records use more space than pointers
- To improve space utilization, involve more sibling nodes in redistribution during splits and merges
  – Involving 2 siblings in redistribution (to avoid split / merge where possible) results in each node having at least entries

Other Issues in Indexing

Record relocation and secondary indices

- If a record moves, all secondary indices that store record pointers have to be updated
- Node splits in B^+ tree file organizations become very expensive
- *Solution*: Use search key of B^+ tree file organization instead of record pointer in secondary index
  - Extra traversal of primary index to locate record
    - Higher cost for queries, but node splits are cheap
  - Add record-id if primary-index search key is non-unique

Indexing Strings

- Variable length strings as keys
  - Variable fan-out
  - Use space utilization as criterion for splitting, not number of pointers

- **Prefix compression**
  - Key values at internal nodes can be prefixes of full key
    - Keep enough characters to distinguish entries in the subtrees separated by the key value
      - E.g., “Silas” and “Silberschatz” can be separated by “Silb”
  - Keys in leaf node can be compressed by sharing common prefixes

B-Tree Index Files

- Similar to B+ tree, but B-tree allows search-key values to appear only once; eliminates redundant storage of search keys

- Search keys in non-leaf nodes appear nowhere else in the B-tree; an additional pointer field for each search key in a non-leaf node must be included

- Generalized B-tree leaf node

P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

(a)

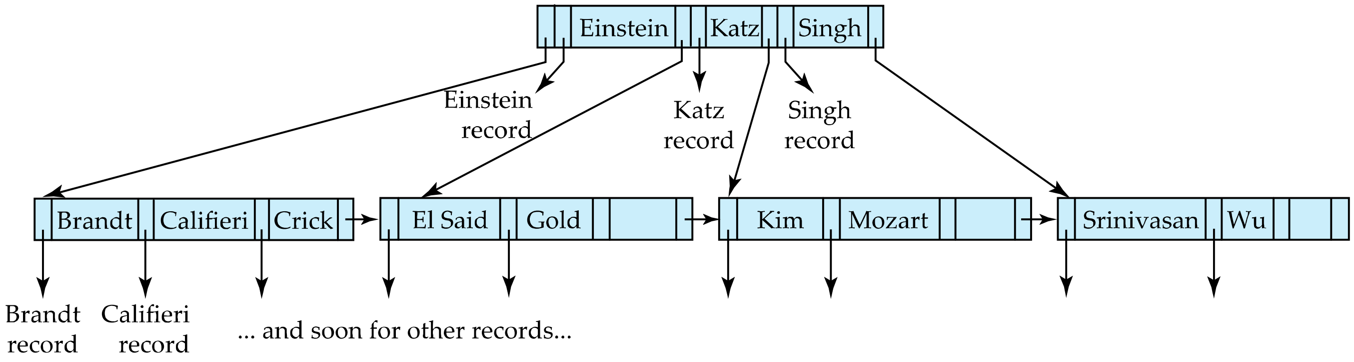
P_1	B_1	K_1	P_2	B_2	K_2	…	P_{m-1}	B_{m-1}	K_{m-1}	P_m

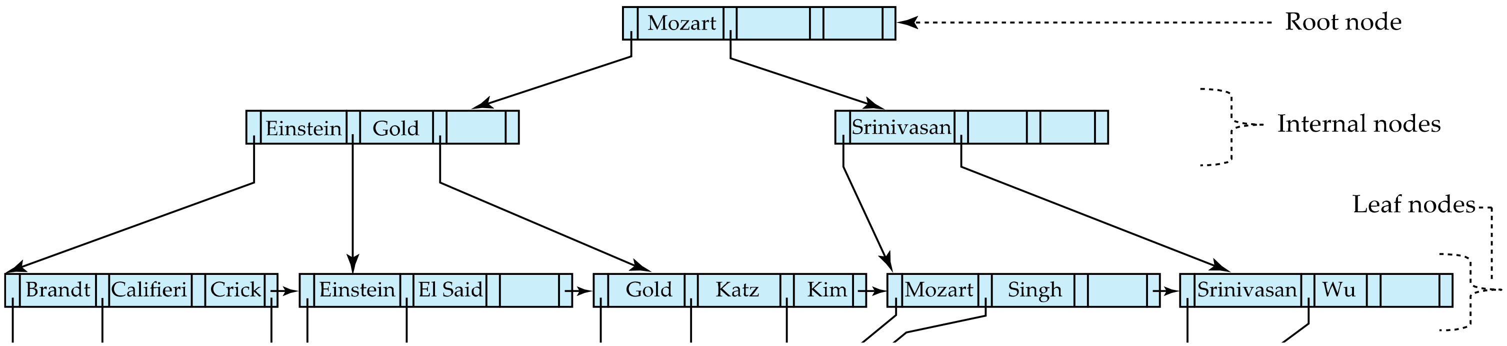
(b)

- Non-leaf node pointers B_i are the bucket or file record pointers

B-Tree Index File Example

- B-tree (above) and B$^+$ tree (below) on same data





B-Tree Index File Example

- Advantages of B-tree indices:
  - May use less tree nodes than a corresponding B^+ tree
  - Sometimes possible to find search-key value before reaching leaf node

- Disadvantages of B-tree indices:
  - Only small fraction of all search-key values are found early
  - Non-leaf nodes are larger, so fan-out is reduced
  - Thus, B-trees typically have greater depth than corresponding B^+ tree
  - Insertion and deletion more complicated than in B^+ tree
  - Implementation is harder than B^+ tree

- Typically, advantages of B-trees do not outweigh disadvantages

Next Lecture

Hashing

Thank you for your attention...

Any question?

Contact:
Department of Information Technology, NITK Surathkal, India
6th Floor, Room: 13
Phone: +91-9477678768
E-mail: shrutilipi@nitk.edu.in