Week 9 Lecture 1

Class	BSCCS2001
Created	@November 2, 2021 2:40 PM
Materials	
Module #	41
Type	Lecture
Week #	9

Indexing and Hashing → Indexing (part 1)

Concepts of Indexing

Search Records

- Consider a table → Faculty (Name, Phone)

Index on "Name"			Table "Faculty"			Index on "Phone"	
Name	*Pointer*		*Rec #*	*Name*	*Phone*	*Pointer*	*Phone*
Anupam Basu	2		1	Partha Pratim Das	81998	6	81664
Pabitra Mitra	6		2	Anupam Basu	82404	1	81998
Partha Pratim Das	1		3	Ranjan Sen	84624	2	82404
Prabir Kumar Biswas	7		4	Sudeshna Sarkar	82432	4	82432
Rajib Mall	5		5	Rajib Mall	83668	5	83668
Ranjan Sen	3		6	Pabitra Mitra	81664	3	84624
Sudeshna Sarkar	4		7	Prabir Kumar Biswas	84772	7	84772

- How to search on Name?
  - Get the phone number for 'Pabitra Mitra'
  - Use "Name" Index — sorted on 'Name', search 'Pabitra Mitra' and navigate on pointer (red #)
- How to search on Phone?
  - Get the name of the faculty having phone number = 84772

    - Use "Phone" Index — sorted on 'Phone', search '84772' and navigate on pointer (rec #)
- We can keep the records sorted on 'Name' or on 'Phone' (called the primary index), but not on both

Basic Concepts

- Indexing mechanisms used to speed up access to desired data
    - For example →
        - Name in a faculty table
        - Author catalog in a library
- **Search Key** → Attribute to set of attributes used to look up records in a file
- An **index file** consists of records (called **index entries**) of the form

- 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***

Index Evaluation Metrics

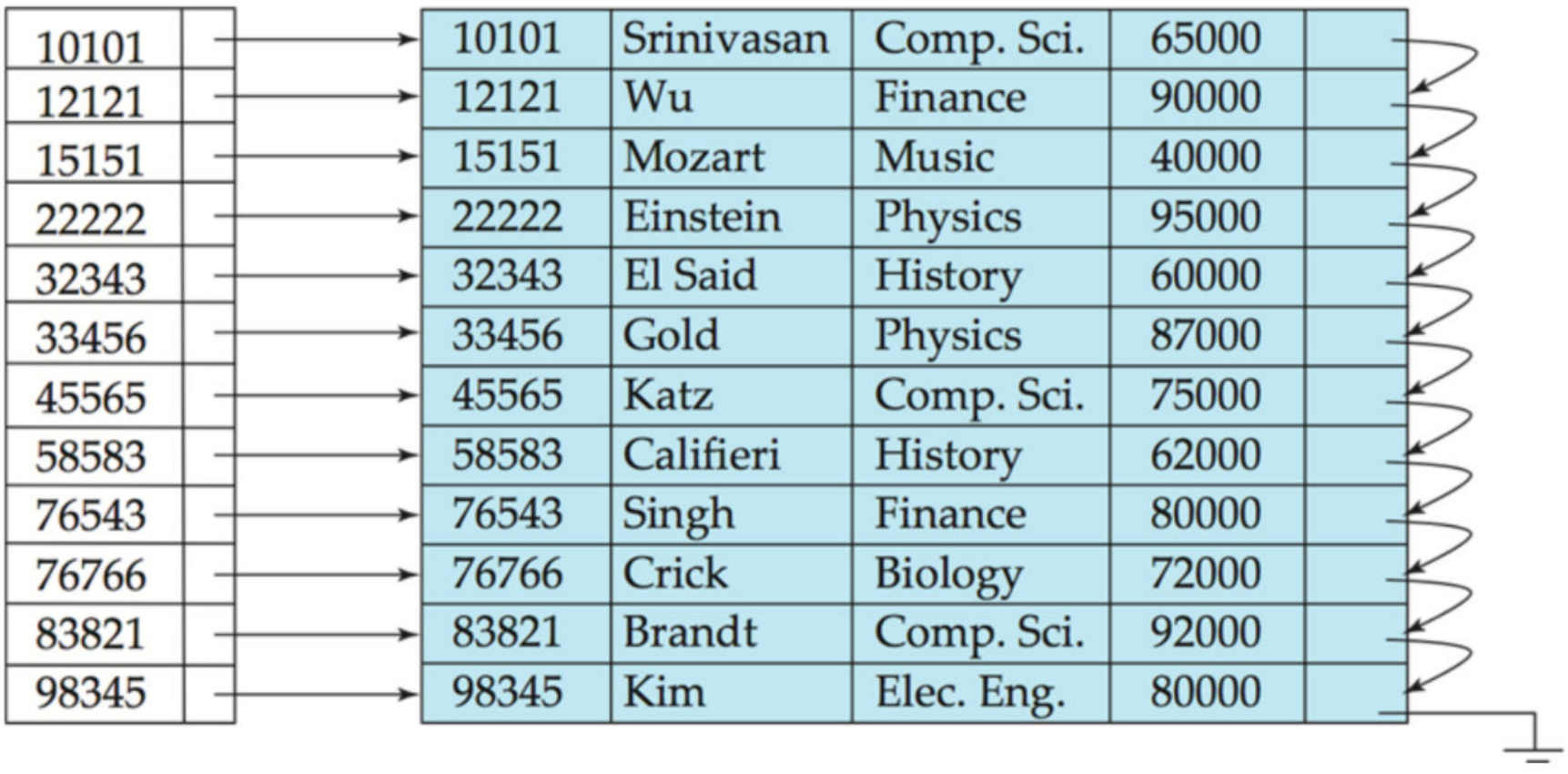
- Access types supported efficiently
    - For example →
        - records with a specified value in the attribute
        - records with an attribute value falling in a specified range of values
- Access time
- Insertion time
- Deletion time
- Space overhead

Ordered Indices

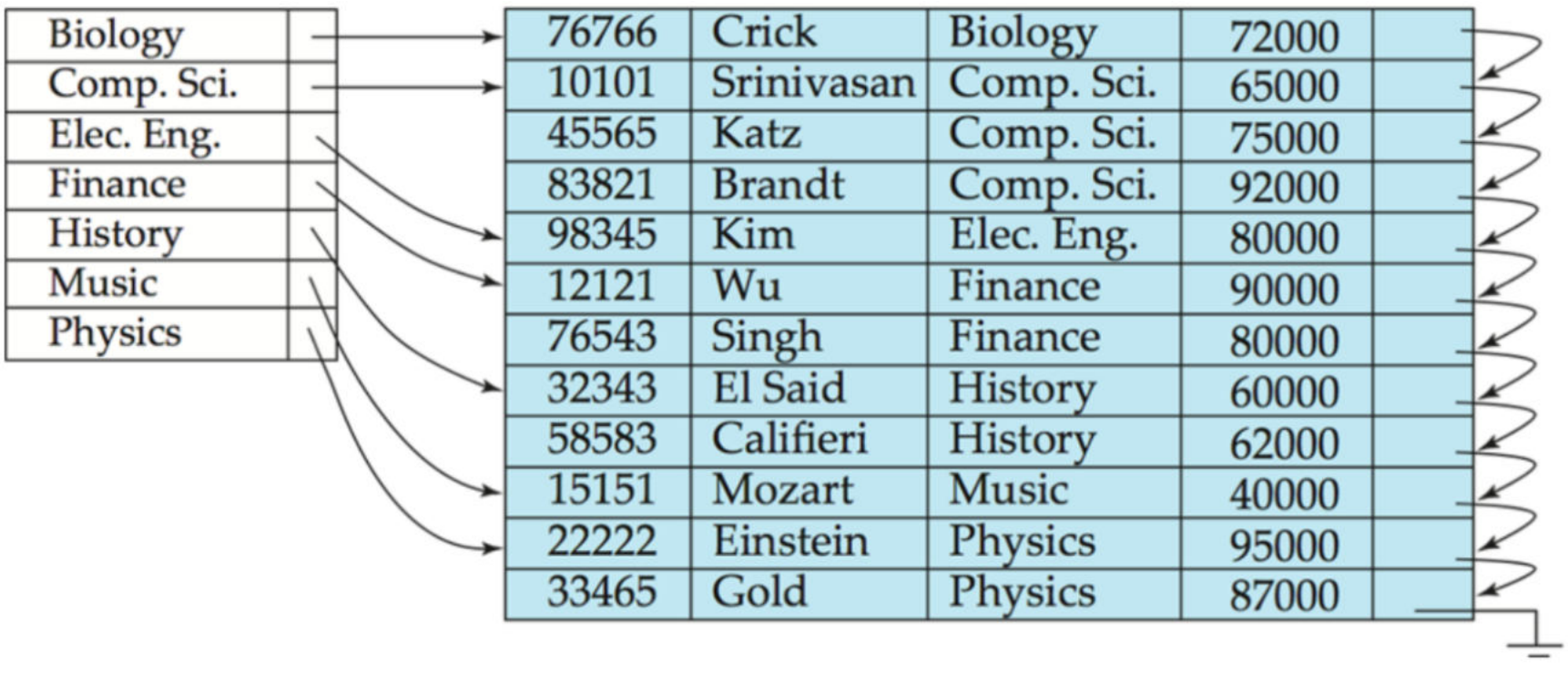
- In an **ordered index**, index entries are stored sorted on the search key value
    - For example → author catalog in library
- **Primary index** → In a sequentially ordered file, the index whose search key specifies the sequential order of the file
    - Also called the **clustering index**
    - 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 the file
    - Also called the **non-clustering index**
- **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
- For example → index on *ID* attribute of *instructor* relation

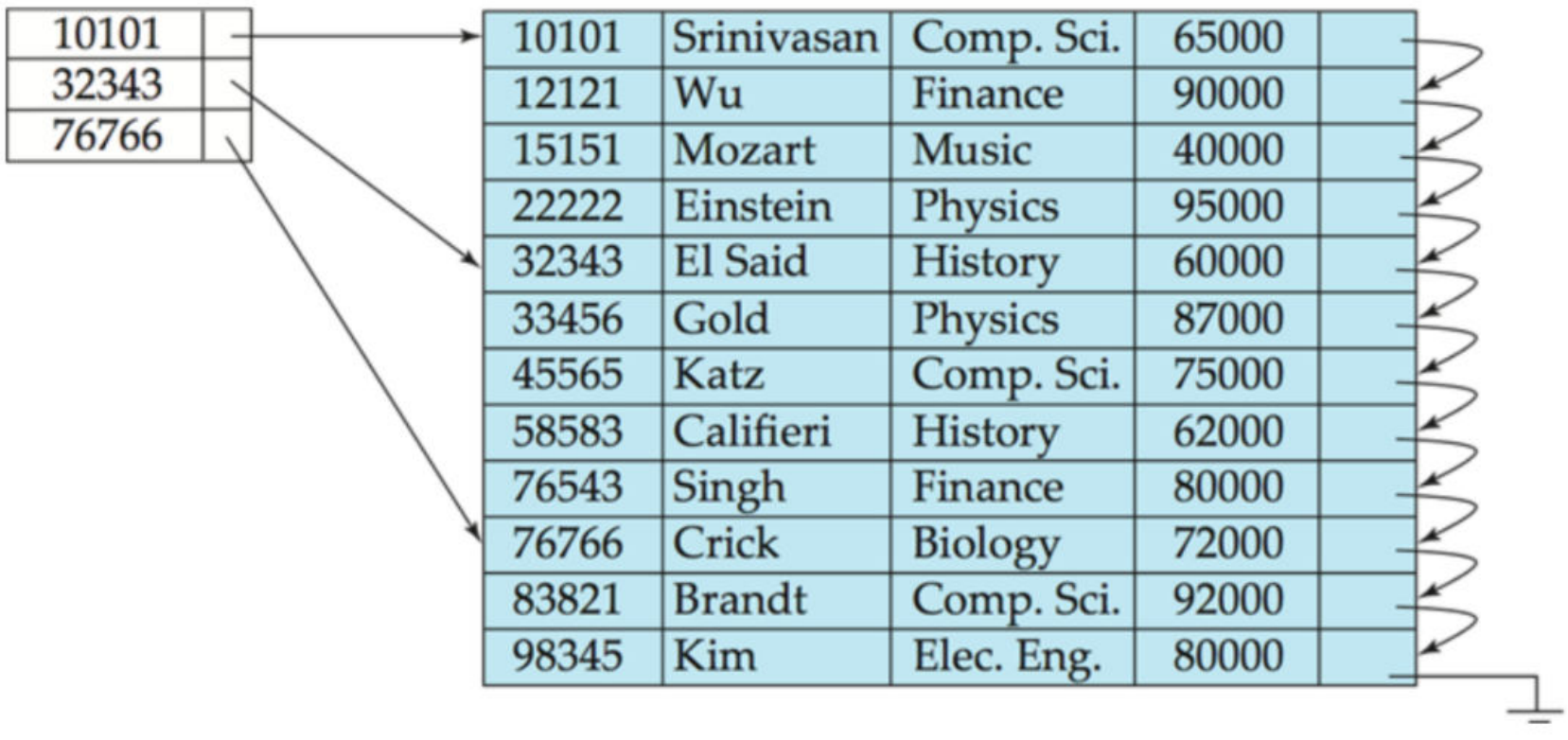


- Dense index on *dept_name*, with *instructor* file stored on *dept_name*

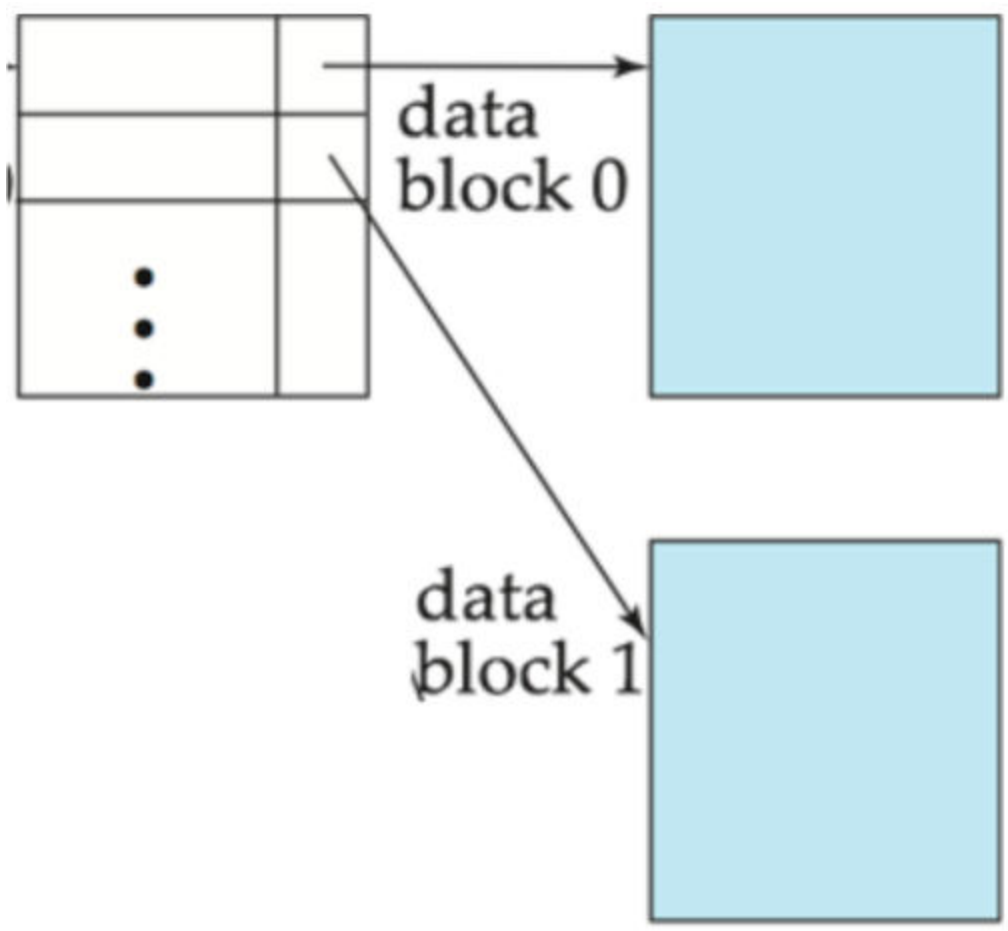


Sparse Index Files

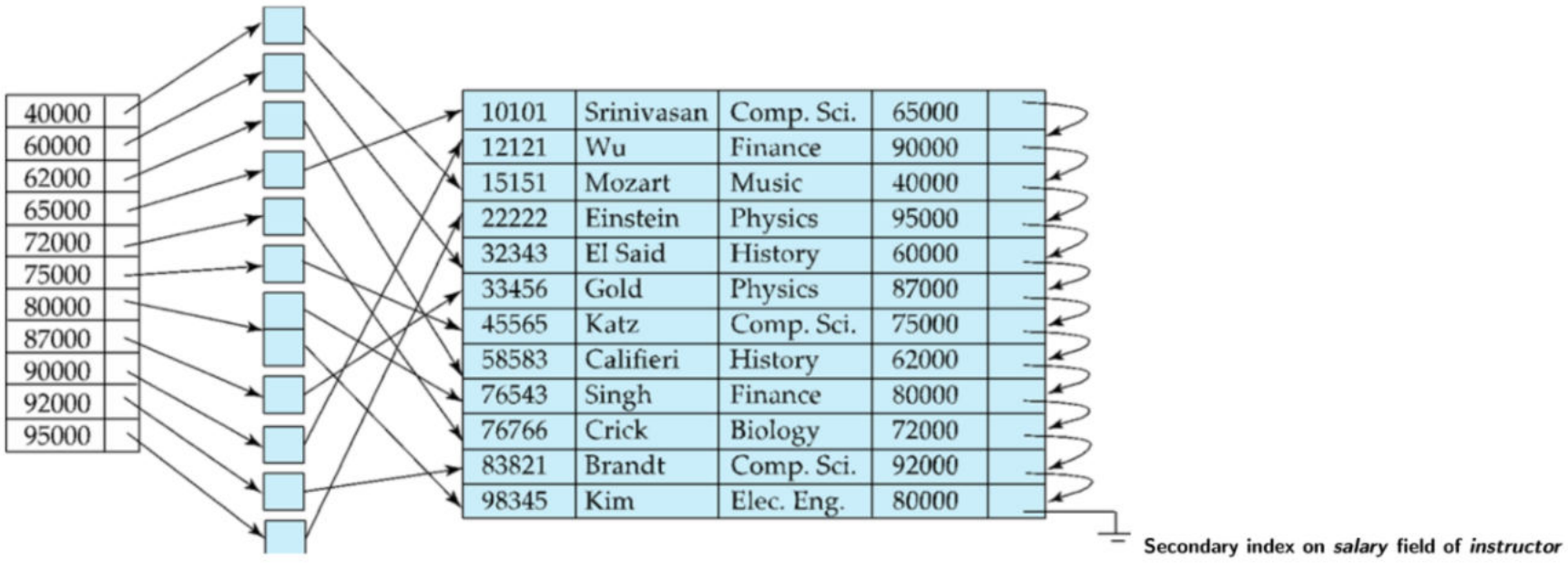
- **Sparse Index** → Contains index records for only some search-key values
  - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value K we →
  - Find index record with largest search-key value $< K$
  - Search file sequentially starting at the record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



- Compared to dense indices →
  - Less space and less maintenance overhead for insertions and deletions
  - Generally slower than dense index for locating records
- **Good tradeoff** → Sparse index with an index entry for every block in file, corresponding to least search-key value in the block



Secondary Indices Example



Secondary index on *salary* field of *instructor*

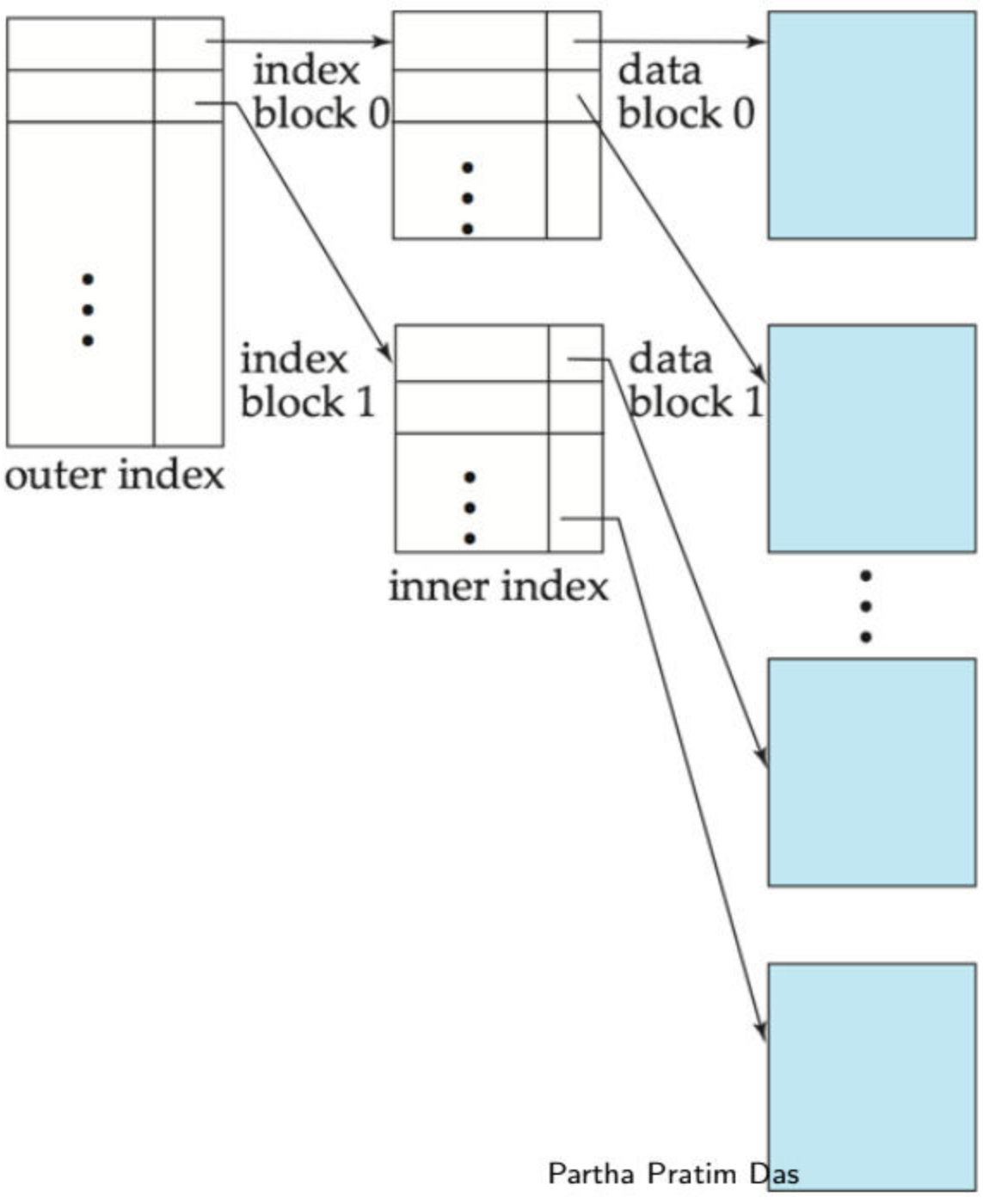
- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value
- Secondary indices have to be dense

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 the disk
  - Block fetch requires about 5 to 10 milliseconds, versus about 100 nanoseconds for memory access

Multilevel Index

- If primary index does not fit in memory, access becomes expensive
- **Solution** → treat primary index kept on disk as a sequential file and construct a sparse index on it
  - outer index → a sparse index of primary index
  - inner index → the primary index file
- If even outer index is too large to fit in the main memory, yet another level of index can be created and so on
- Indices at all levels must be updated on insertion or deletion from the file



Index Update → Deletion

- If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also
- **Single-level index entry deletion** →
  - **Dense indices** → deletion of search-key is similar to file record deletion

    - **Sparse indices —**
        - If an entry for the search key exists in the index, it is deleted by replacing the entry in the index with the next search-key value in the file (in search-key order)
        - If the next search-key value already has an index entry, the entry is deleted instead of being replaced
- **Single-level index insertion** →
    - Perform a lookup using the search-key value appearing in the record to be inserted
    - **Dense indices** → if the search-key value does not appear in the index, insert it
    - **Sparse indices** → if index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created
        - If a new block is created, the first search-key value appearing in the new block is inserted into the index
- **Multilevel insertion and deletion** → algorithms are simple extensions of the single-level algorithms

Secondary Indices

- Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index) satisfy some condition
    - Example 1 → In the instructor relation stored sequentially by the ID, we may want to find all instructors in a particular department
    - Example 2 → as above, but where we want to find all instructions with a specified salary or with salary in a specific range of values
- We can have a secondary index with an index record for each search-key value

Week 9 Lecture 2

Class	BSCCS2001
Created	@November 2, 2021 4:46 PM
Materials	
Module #	42
Type	Lecture
Week #	9

Indexing and Hashing → Indexing (part 2)

Balanced Binary Search Trees

Search Data Structures

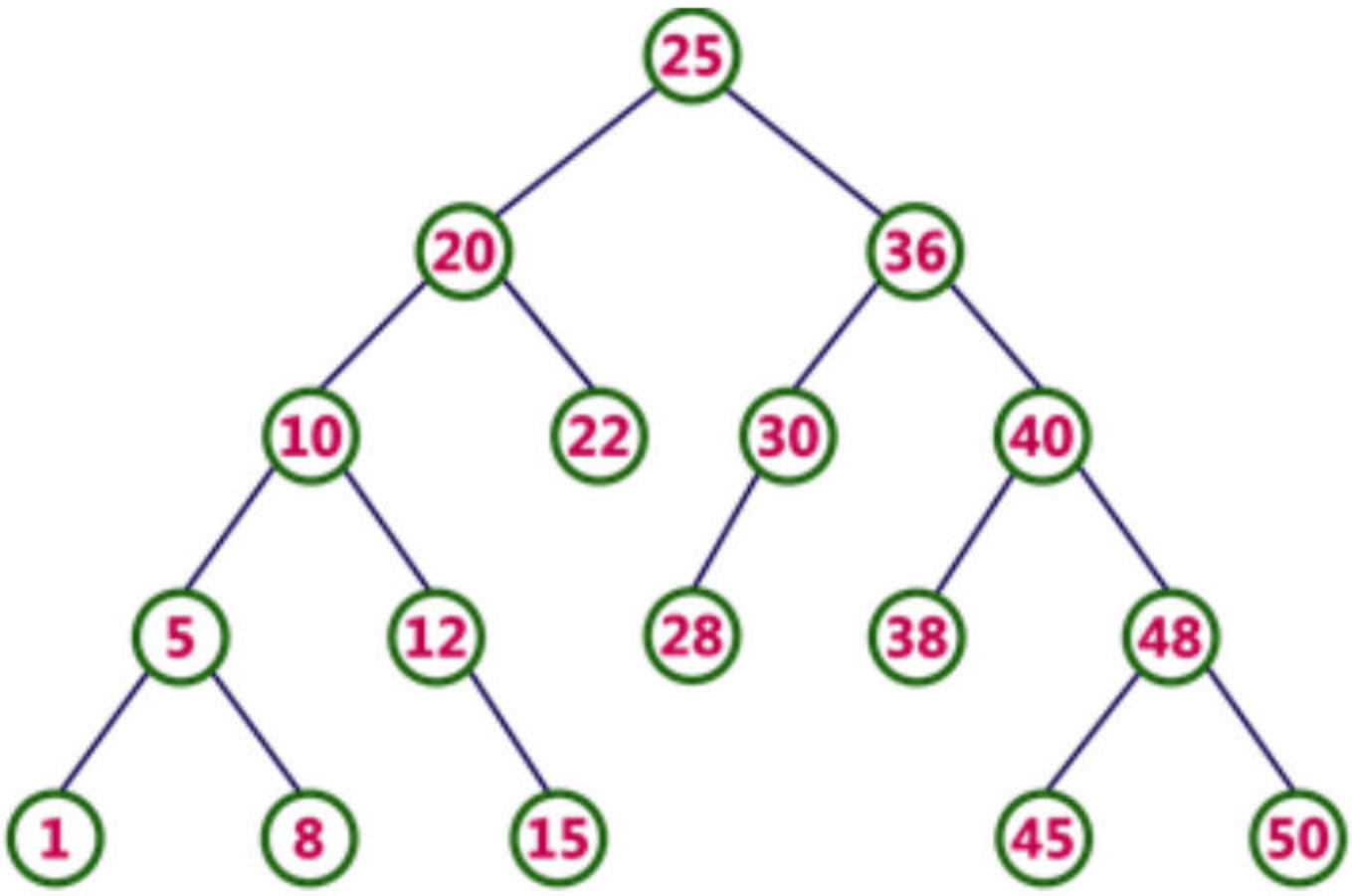
- How to search a key in a list of n data items?
  - Linear search: $O(n)$ → Find $28 \implies 16$ comparisons
    - Unordered items in an array — search sequentially
    - Unordered/Ordered items in a list — search sequentially

22	50	20	36	40	15	08	01	45	48	30	10	38	12	25	28	05	END

  - Binary search: $O(log\ n)$ → Find $28 \implies 4$ comparisons — 25, 36, 30, 28
    - Ordered items in an array — search by divide-and-conquer

01	05	08	10	12	15	20	22	25	28	30	36	38	40	45	48	50	END

    - Binary Search Tree — recursively on left/right



- Worst case time (n data items in the data structure)

Data Structure	Search	Insert	Delete	Remarks
Unordered Array	O(n)	O(1)	O(1)	The time to Insert / Delete an item is the time after the location of the item has been ascertained by Search.
Ordered Array	O(log n)	O(n)	O(n)	
Unordered List	O(n)	O(1)	O(1)	
Ordered List	O(n)	O(1)	O(1)	
Binary Search Tree	O(h)	O(1)	O(1)	

- Between an array and a list, there is a trade-off between search and insert/delete complexity
- For a BST of n nodes, $log\ n \leq h < n$, where h is the height of the tree
- A BST is balanced if $h \sim O(log\ n)$ → that is what we desire

Search Data Structures → BST

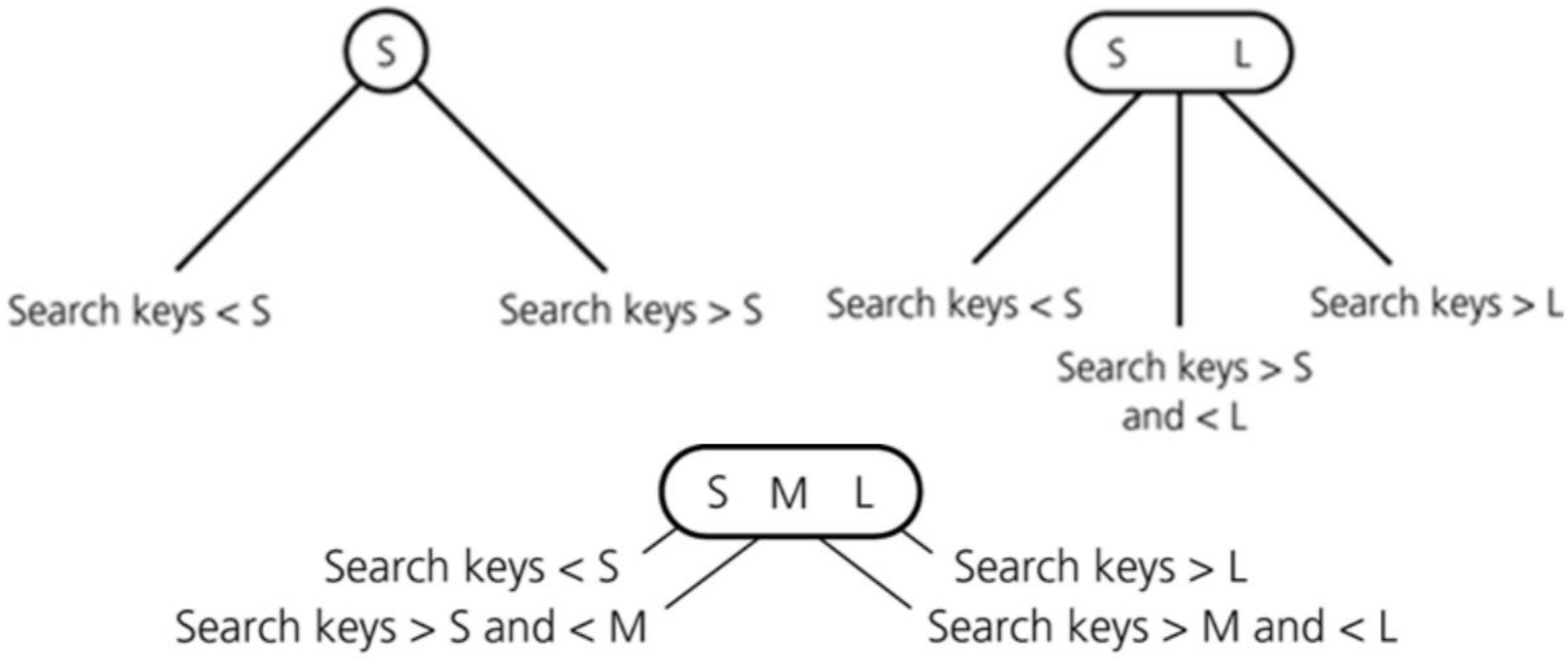
- In the worst case, searching a key in a BST is $O(h)$, where h is the height of the tree
- **Bad Tree** → $h \sim O(n)$
  - The BST is a skewed binary search tree (all the nodes except the leaf would have only one child)
  - This can happen if keys are inserted in sorted order
  - Height (h) of the BST having n elements becomes $n - 1$
  - Time complexity of search in BST becomes $O(n)$
- **Good Tree** → $h \sim O(log\ n)$
  - The BST is a balanced binary search tree
  - This is possible if
    - If keys are inserted in purely randomized order, Or
    - If the tree is explicitly balanced after every insertion
  - Height (h) of the binary search tree becomes $log\ n$
  - Time complexity of search in BST becomes $O(log\ n)$

Balanced Binary Search Trees

- A BST is balanced if $h \sim O(log\ n)$
- Balancing guarantees may be of various types →
  - Worst-case
    - AVL Tree: Self-balancing BST
      - Named after inventors Adelson-Velsky-Landis
      - Heights of the two child subtrees of any node differ by at most one: $|h_L - h_R| \leq 1$
      - If they differ by more than one, rebalancing is done by rotation
  - Randomized
    - Randomized BST
      - A BST of n keys is random if either it is empty $(n = 0)$ or the probability that a given key is at the root is $\frac{1}{n}$ and the left and the right subtrees are random
    - Skip List
      - A skip list is built (probabbilistically) in layers of ordered linked lists
  - Amortized
    - Splay
      - A BST where recently accessed elements are quick to access again
- These data structures have the optimal complexity for the required operations →
  - Search: $O(log\ n)$
  - Insert: Search + $O(1)$ → $O(log\ n)$
  - Delete: Search + $O(1)$ → $O(log\ n)$
- And they are →
  - Good for in-memory operations
  - Work well for small volume of data
  - Has complex rotation and/or similar operations
  - Do not scale for external data structures

2-3-4 Trees

- All leaves are at the same depth
  - Height, h, of all leaf nodes are the same
    - $h \sim O(log\ n)$
    - Complexity of search, insert and delete → $O(h) \sim O(log\ n)$
- All data is kept in sorted order
- Every node (leaf or internal) is a 2-node, 3-node or a 4-node (based on the number of links or children) and holds one, two or three data elements, respectively
- Generalizes easily to larger nodes
- Extends to external data structures
- Uses 3 kinds of nodes satisfying key relationships as shown below:
  - A 2-node must contain a single data item (S) and two links
  - A 3-node must contain two data items (S, L) and three links
  - A 4-node must contain three data items (S, M, L) and four links
  - A leaf may contain either one, two or three data items

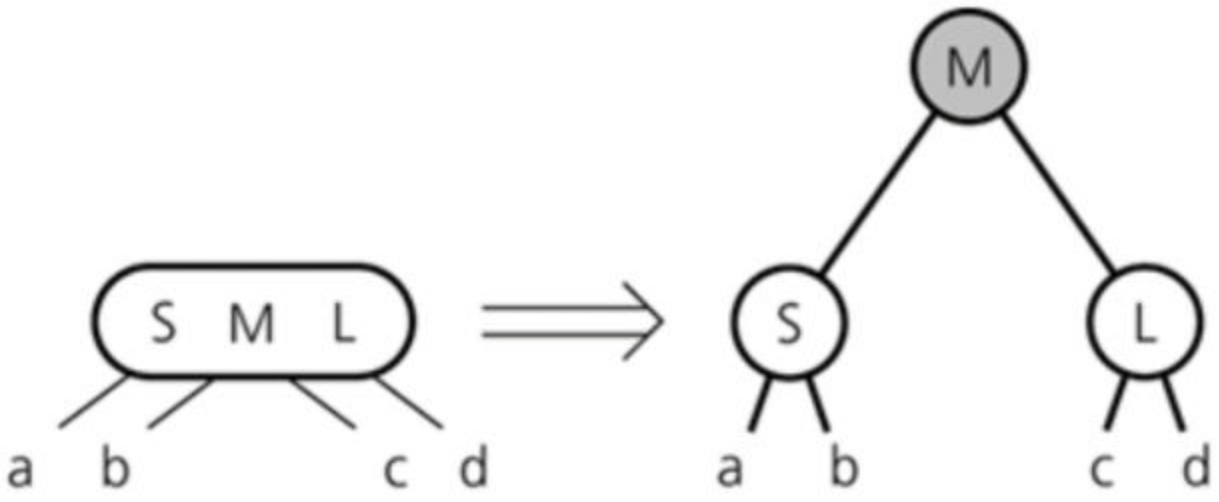


2-3-4 Trees → Search

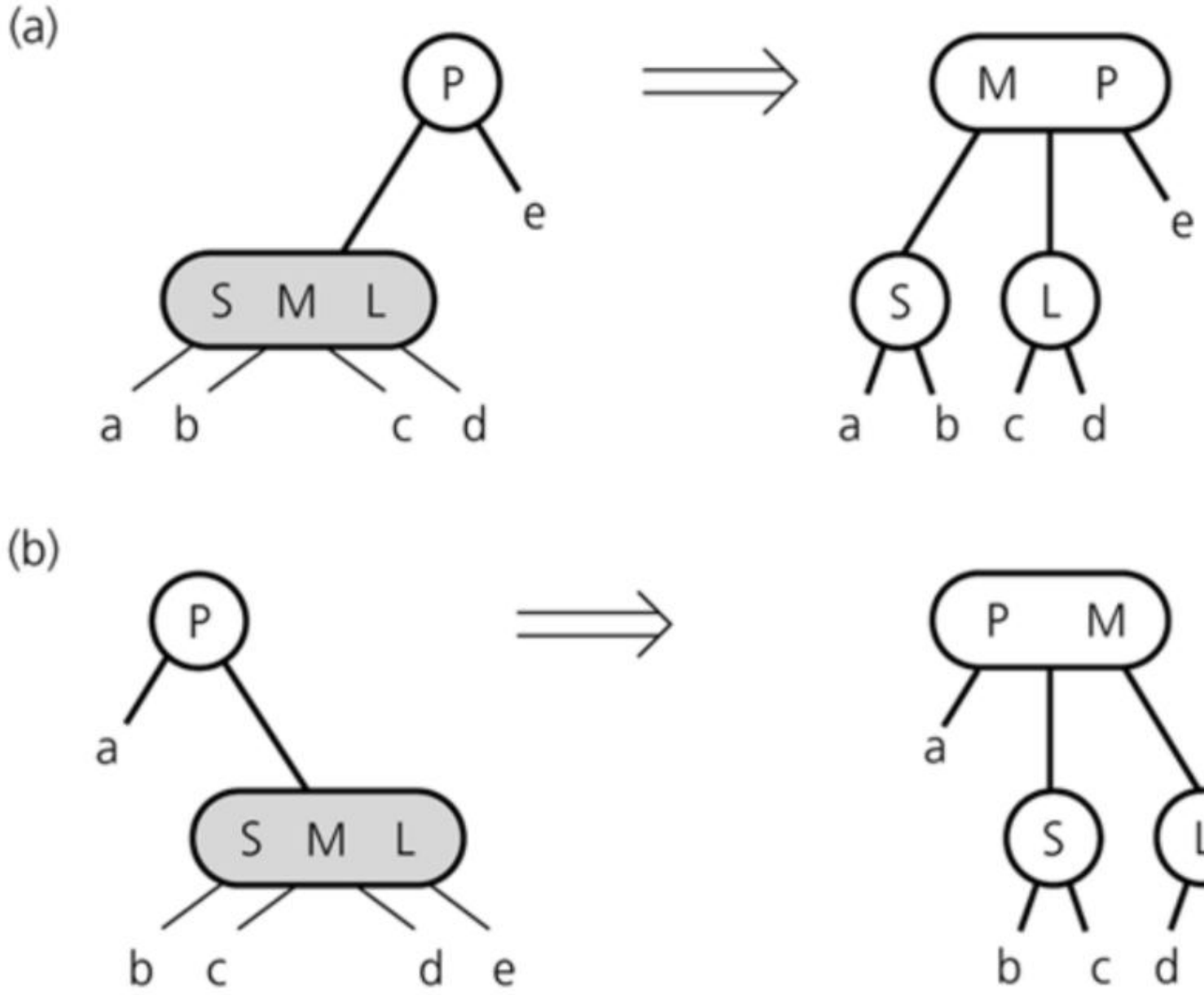
- Search
  - Simple and natural extension of search in BST

2-3-4 Trees → Insert

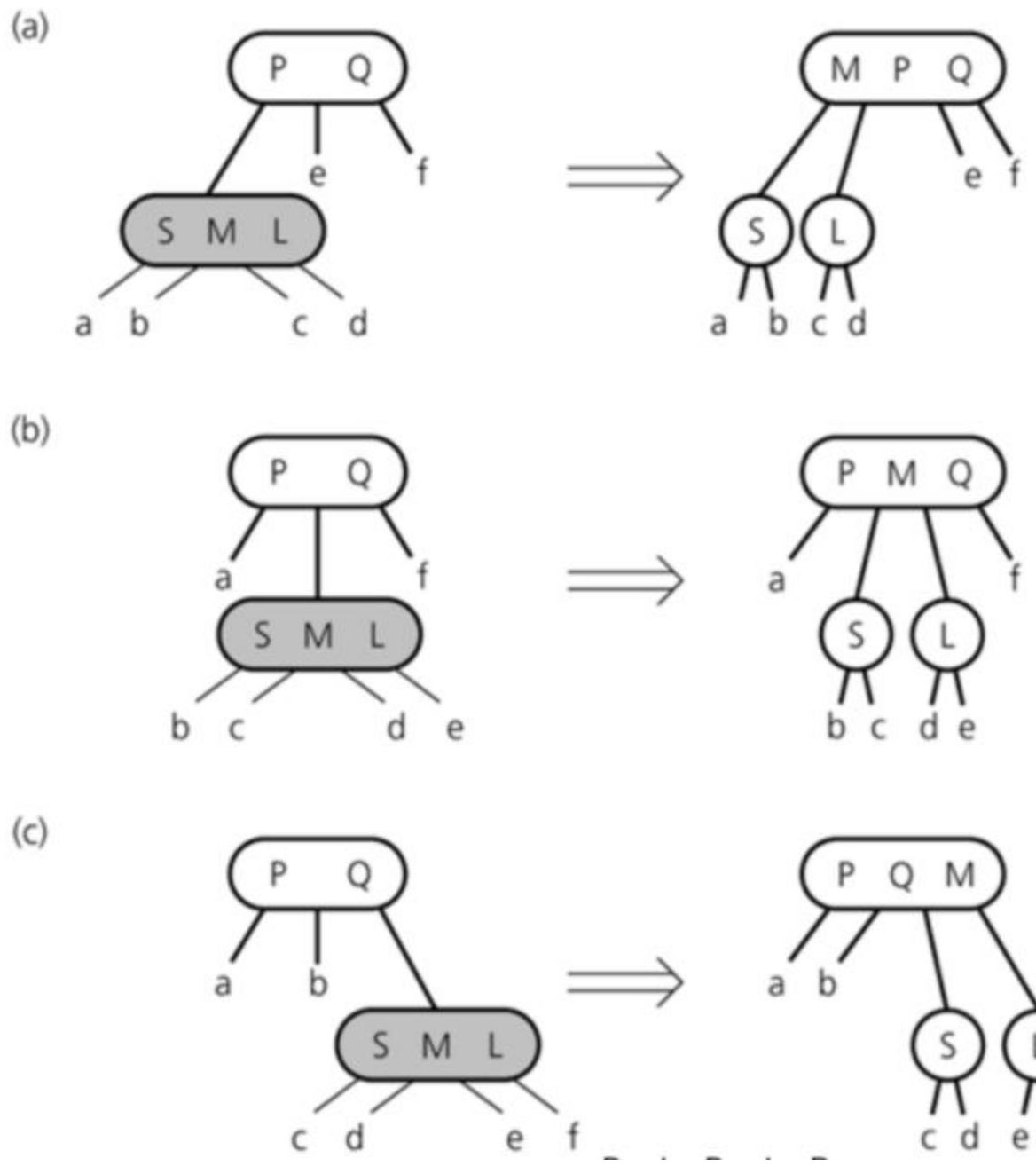
- Insert
  - Search to find the expected location
    - If it is a 2 node, change to 3 node and insert
    - If it is a 3 node, change to 4 node and insert
    - If it is a 4 node, split the node by moving the middle item to parent node and then insert
  - Node splitting
    - A 4-node is split as soon as it is encountered during a search from the root to a leaf
    - The 4-node that is split will
      - Be the root
      - Have a 2-node parent
      - Have a 3-node parent
- Splitting at Root



- Splitting with 2 Node parent



- Splitting with 3 Node parent

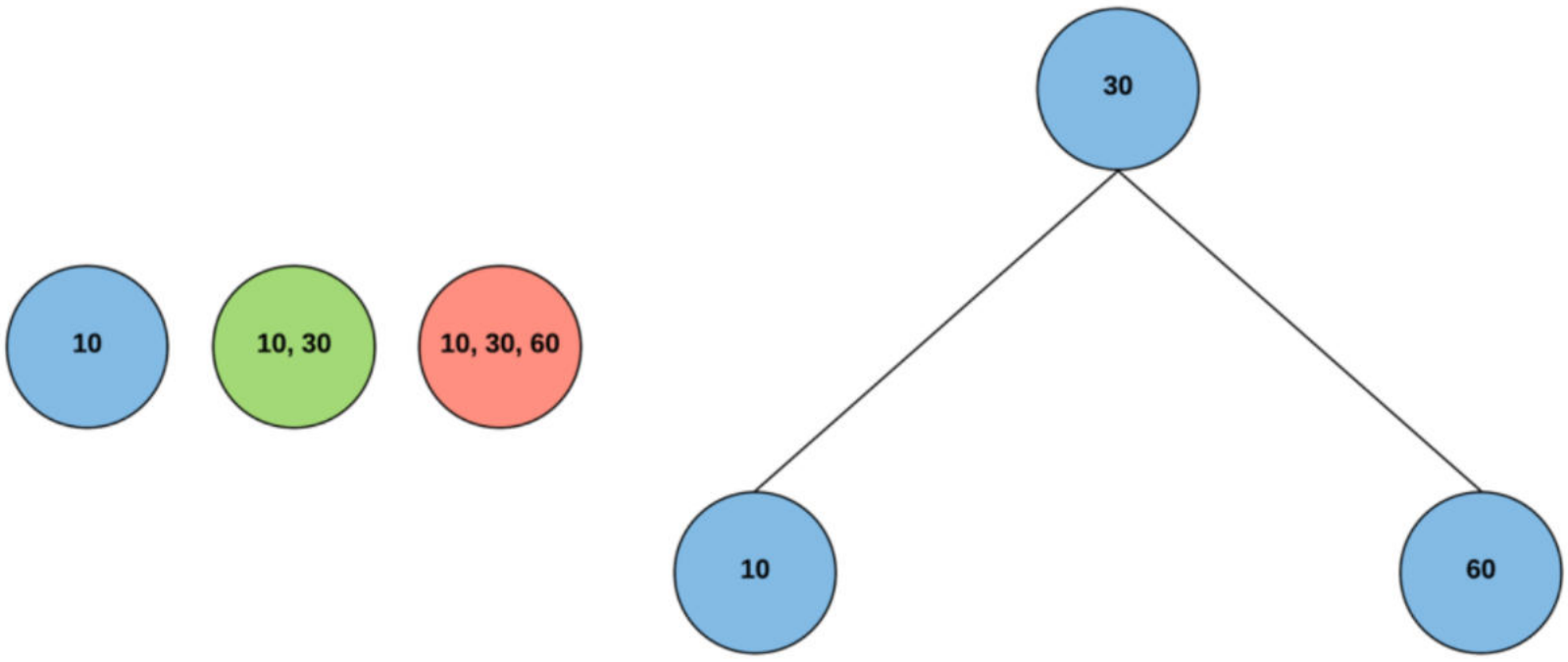


- Node Splitting: There are two strategies:
  - Early: Split a 4-node as soon as you cross on in traversal

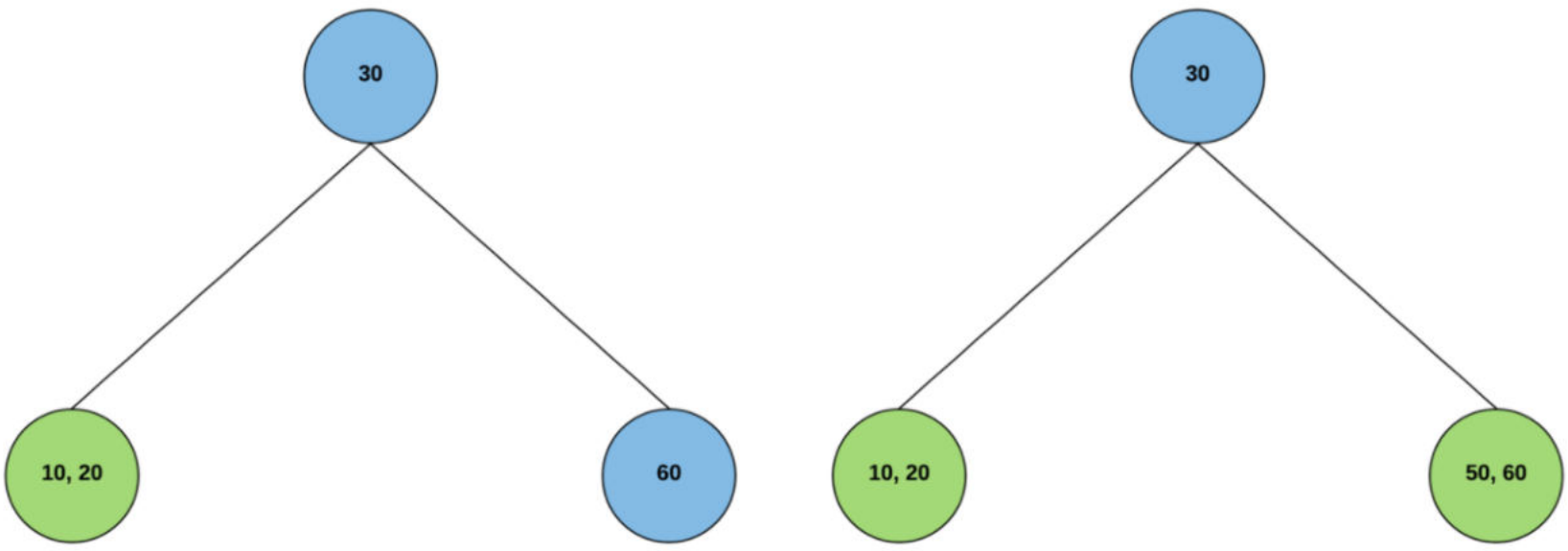
        - It ensures that the tree does not have a path with multiple 4-nodes at any point
    - Late: Split a 4-node only when you need to insert an item in it
        - This might lead to cases where for one insert we may need to perform $O(h)$ splits going till up to the root
- Both are valid and has the same complexity $O(h)$
    - However, they lead to different results
    - Different texts and sites follow different strategies
- Here, we are following early strategy

2-3-4 Trees: Insert → Example

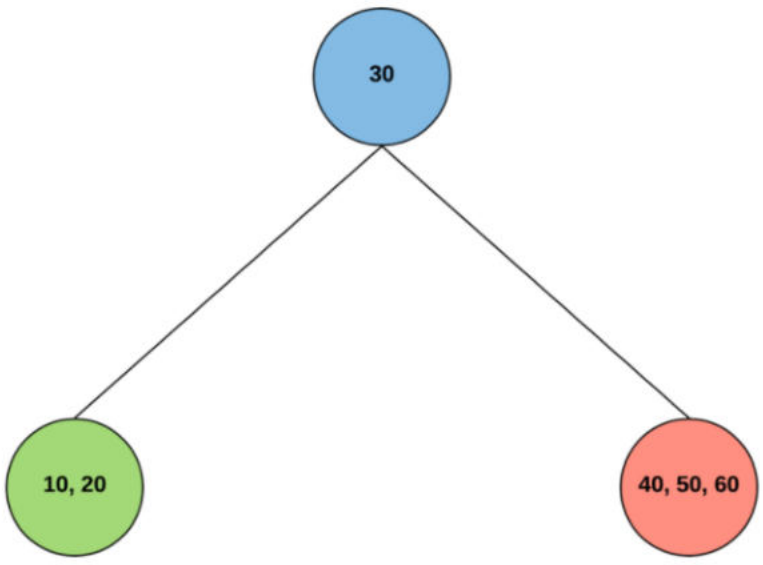
- Insert $10, 30, 60, 20, 50, 40, 70, 80, 15, 90, 100$
- 10
- $10, 30$
- $10, 30, 60$
- Split for 20

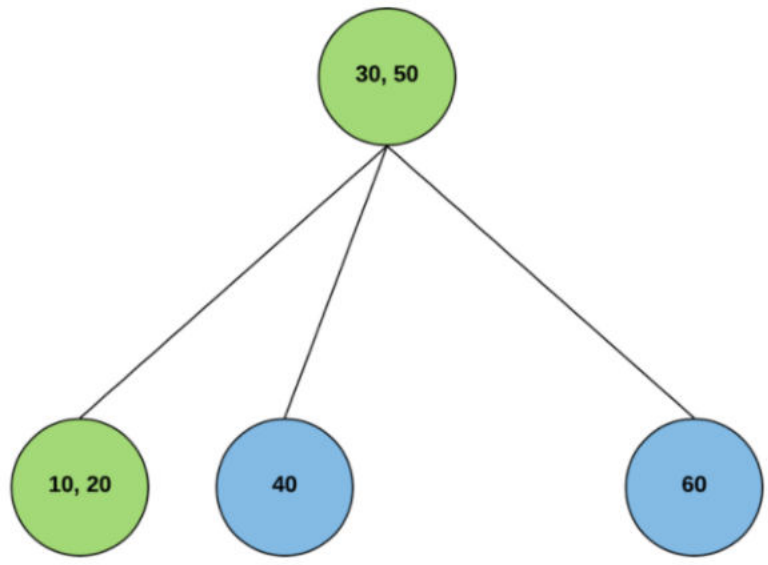


- $10, 30, 60, 20$
- $10, 30, 60, 20, 50$

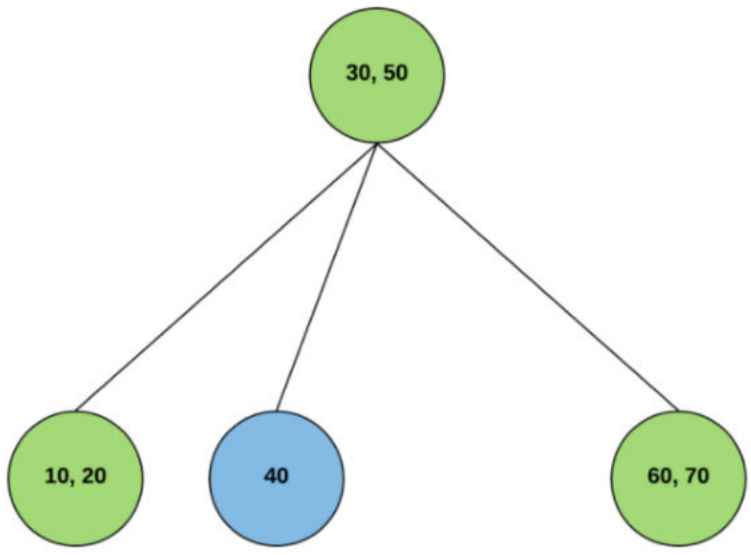


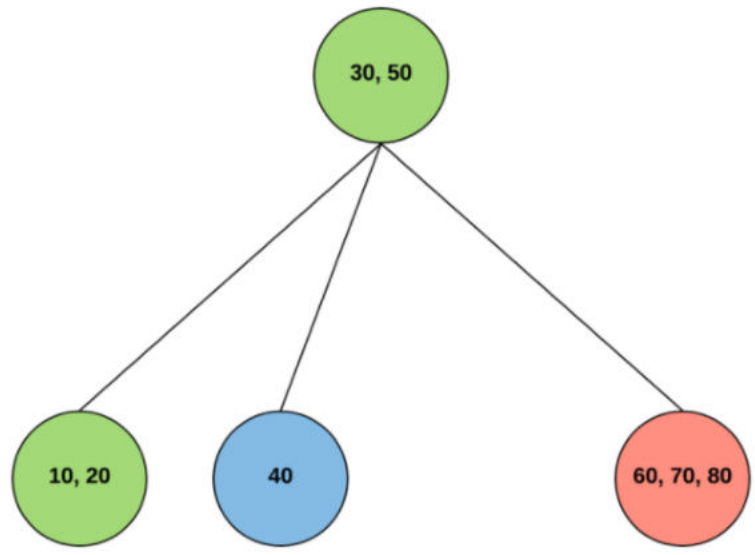
- $10, 30, 60, 20, 50, 40$
- Split for 70



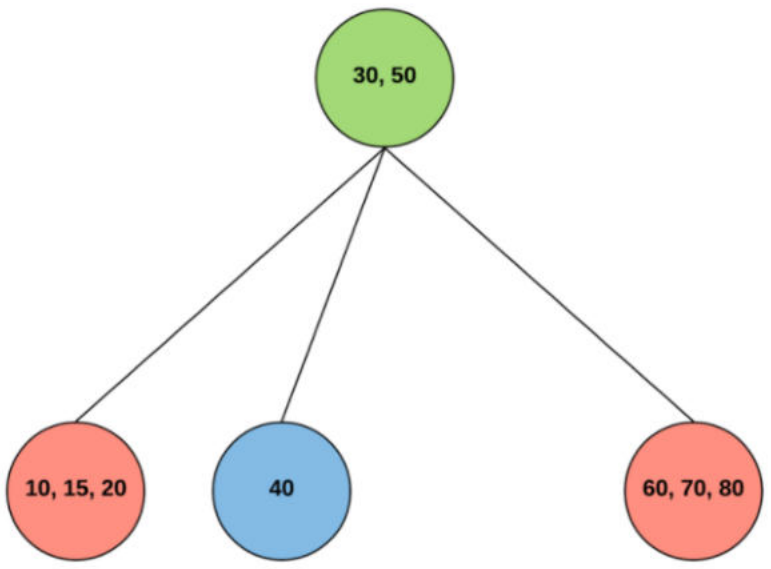


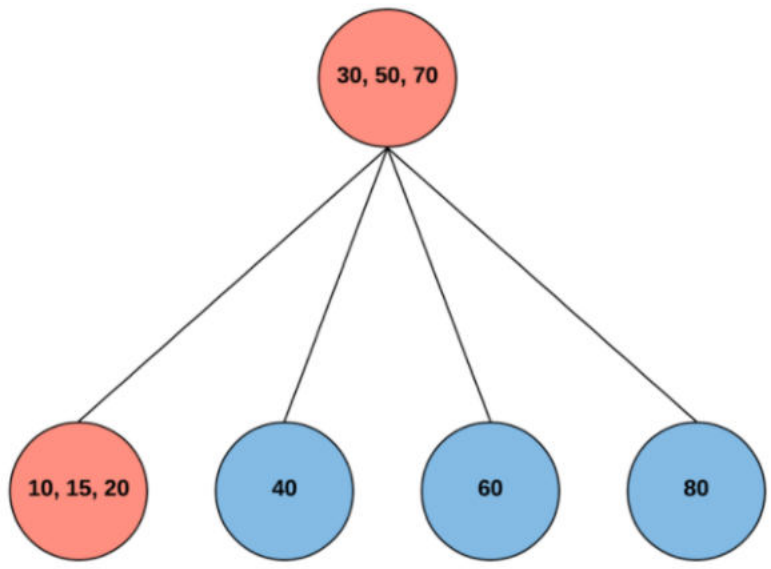
- $10, 30, 60, 20, 50, 40, 70$
- $10, 30, 60, 20, 50, 40, 70, 80$



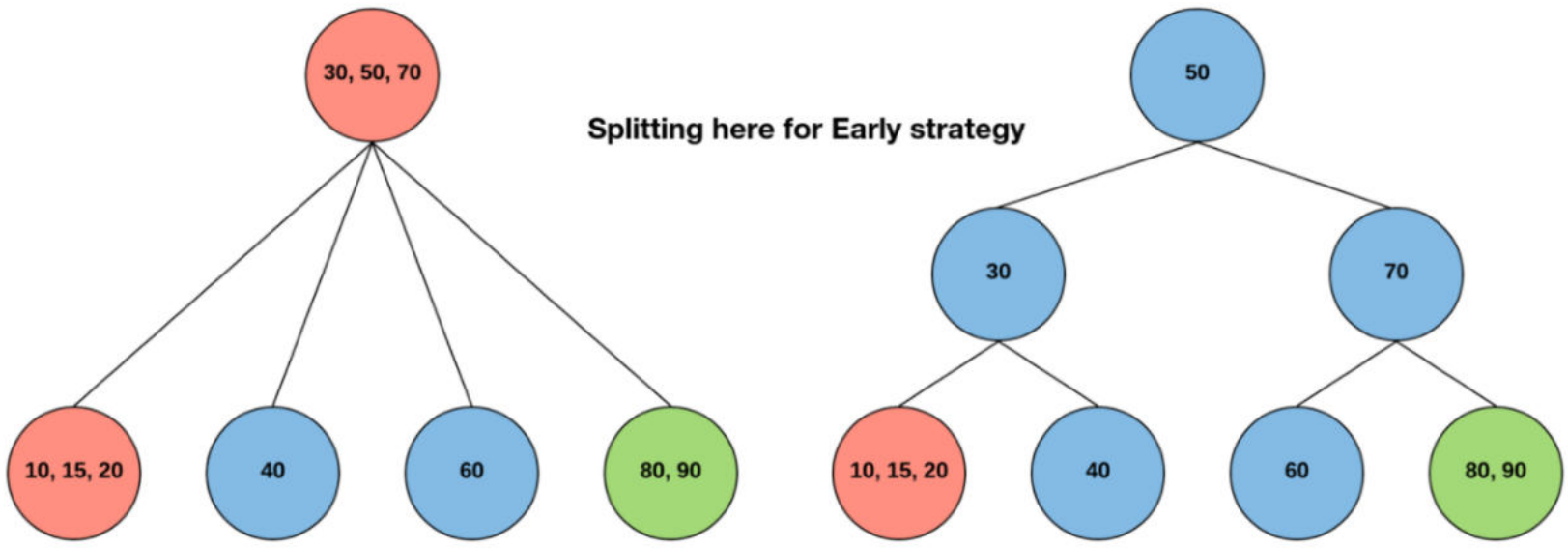


- $10, 30, 60, 20, 50, 40, 70, 80, 15$
- Split for 90

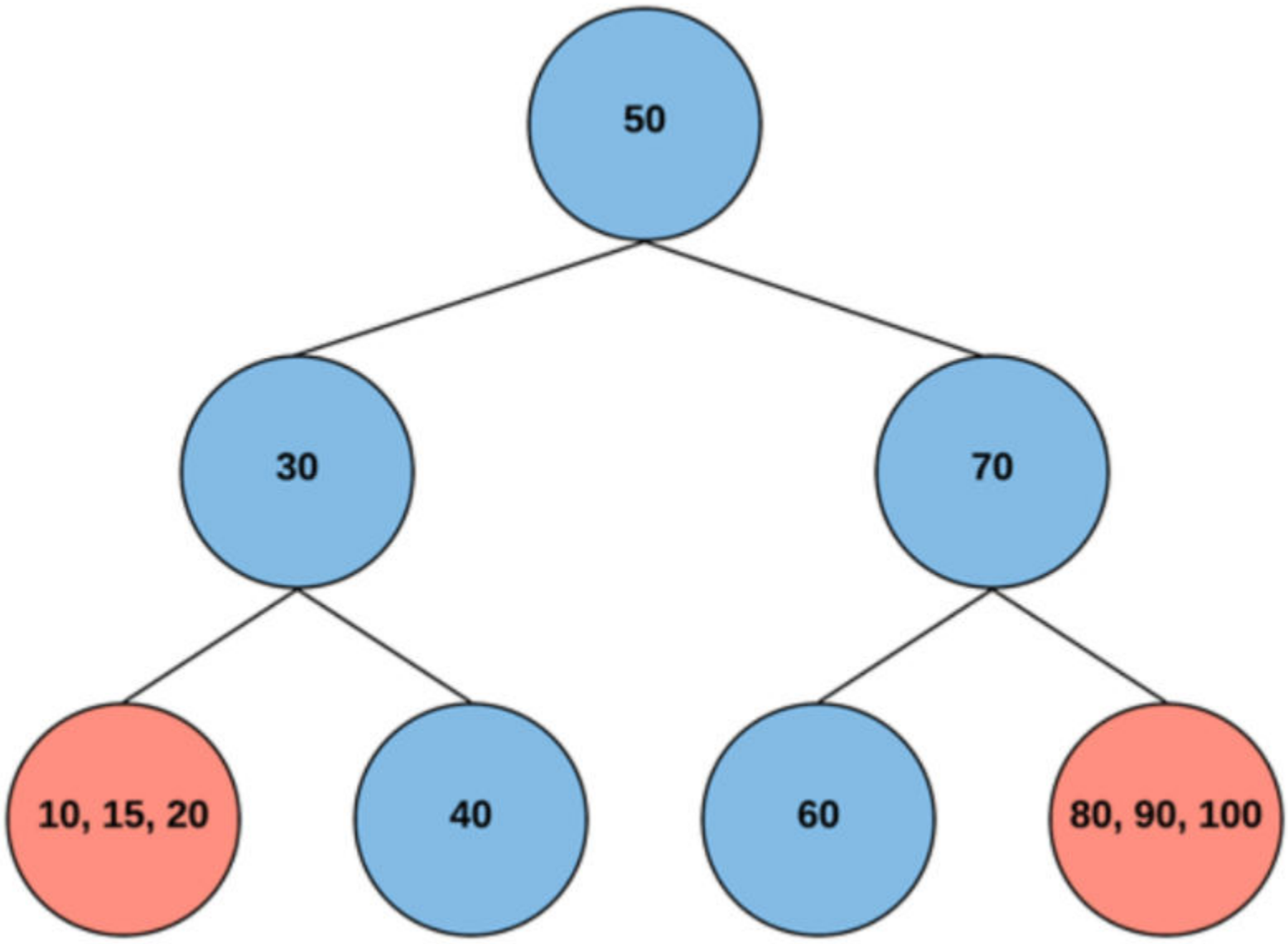




- $10, 30, 60, 20, 50, 40, 70, 80, 15, 90$
- Split for 100



- $10, 30, 60, 20, 50, 40, 70, 80, 15, 90, 100$



2-3-4 Trees: Delete

- Delete
  - Locate the node n that contains the item `theItem`
  - Find `theItem`'s inorder successor and swap it with `theItem` (deletion will always be at the leaf)
  - If that leaf is a 3-node or a 4-node, remove `theItem`
  - To ensure that `theItem` does not occur in a 2-node
    - Transform each 2-node encountered into a 3-node or a 4-node
    - Reverse different cases illustrated for splitting

2-3-4 Tree

- Advantages
  - All leaves are at the same depth: Height, $h \sim O(log\ n)$
  - Complexity of search, insert and delete: $O(h) \sim O(log\ n)$

    - All data is kept in sorted order
    - Generalizes easily to larger nodes
    - Extends to external data structures
- Disadvantages
    - Uses variety of node types — need to destruct and construct multiple nodes for converting a 2 Node to 3 Node, a 3 Node to a 4 Node, for splitting, etc

---

- Consider only one node type with space for 3 items and 4 links
    - Internal node (non-root) has 2 to 4 children (links)
    - Leaf node has 1 to 3 items
    - Wastes some space, but has several advantages for external data structures
- Generalizes easily to larger nodes
    - All paths from root to leaf are of the same length
    - Each node that is not a root or a leaf has between $\lceil \frac{n}{2} \rceil$ and n children
    - A leaf node has between $\lceil \frac{(n-1)}{2} \rceil$ 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, it can have between 0 and $(n - 1)$ values
- Extends to external data structures
    - B-Tree
    - 2-3-4 Tree is a B-Tree when $n = 4$

Week 9 Lecture 3

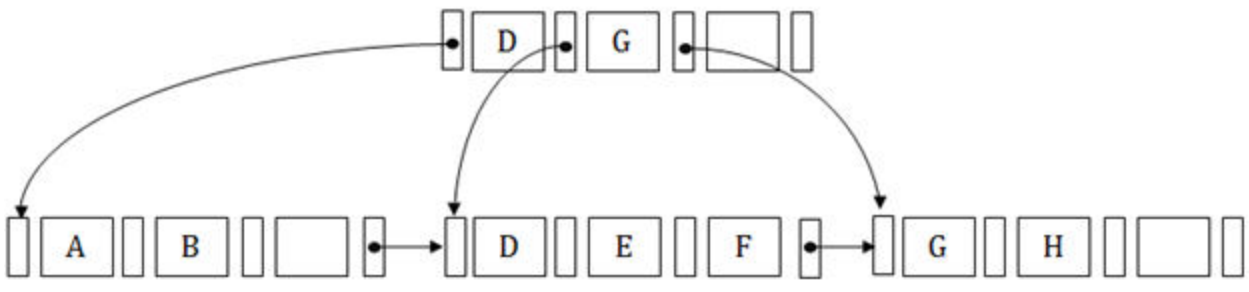
Class	BSCCS2001
Created	@November 2, 2021 5:49 PM
Materials	
Module #	43
Type	Lecture
Week #	9

Indexing and Hashing → Indexing (part 3)

B^+ Tree Index Files

B^+ Tree

- It is a ***balanced binary search tree***
  - It follows a multi-level index format like 2-3-4 Tree
- It has the leaf nodes denoting actual data pointers
- Ensures that all leaf nodes remain at the same height (like 2-3-4 Tree)
- It has the leaf nodes that are linked using a linked list
  - It can support random access as well as sequential access
- Example

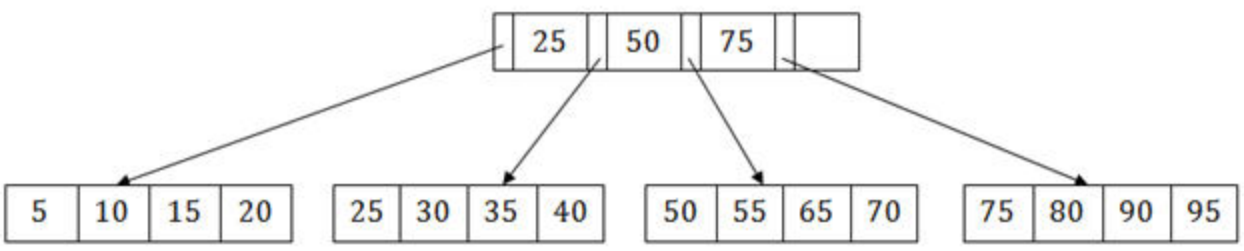


- Internal node contains
  - At least $\frac{n}{2}$ child pointers, except the root node

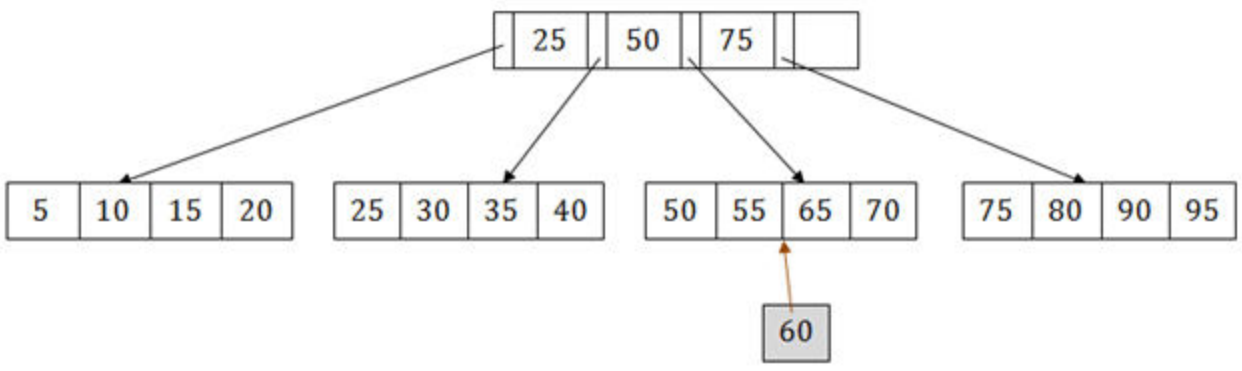
    - At most n pointers
- Leaf node contains
    - At least $\frac{n}{2}$ record pointers and $\frac{n}{2}$ key values
    - At most n record pointers and n key values
    - One block pointer P to point to next leaf node

B$^+$ Tree: Search

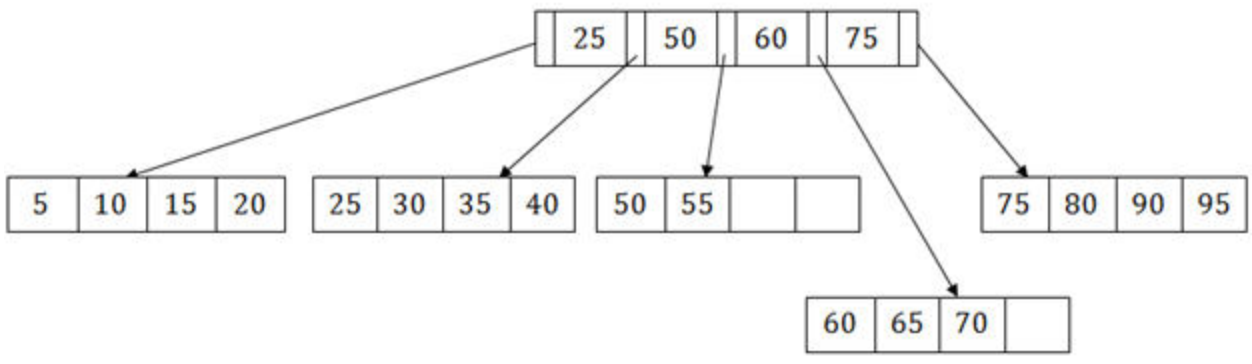
- Suppose we have to search 55 in the B$^+$ tree below
    - First, we will fetch for the intermediary node which will direct to the leaf node that can contain a record for 55
- So, in the intermediary node, we will find a branch between 50 and 75 nodes
    - Then at the end, we will be redirected to the third leaf node
    - Here DBMS will perform a sequential search to find 55



B$^+$ Tree: Insert

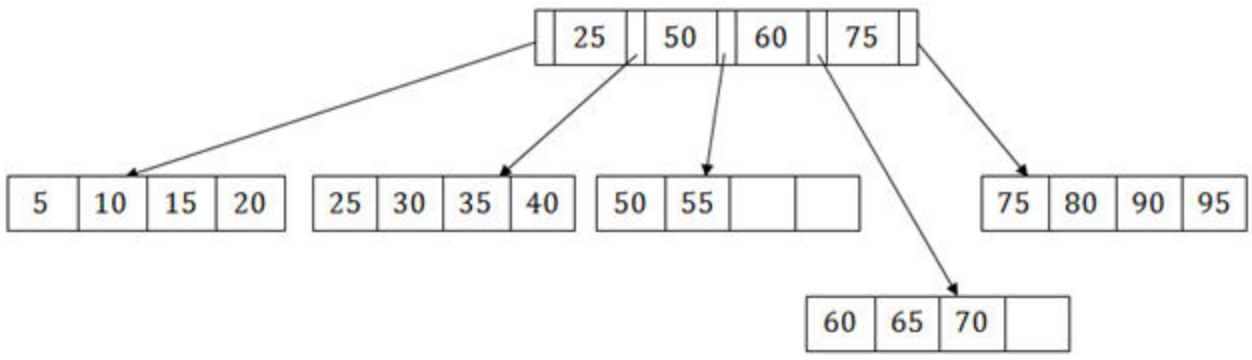


- Suppose we want to insert a record 60 that goes to the 3^{rd} leaf node after 55
- The leaf node of this tree is already full, so we cannot insert 60 there
- So we have to split the leaf node, so that it can be inserted into tree without affecting the fill factor, balance and order
- The 3^{rd} leaf node has the values $(50, 55, 60, 65, 70)$ and its current root node is 50
- We will split the leaf node of the tree in the middle so that its balance is not altered

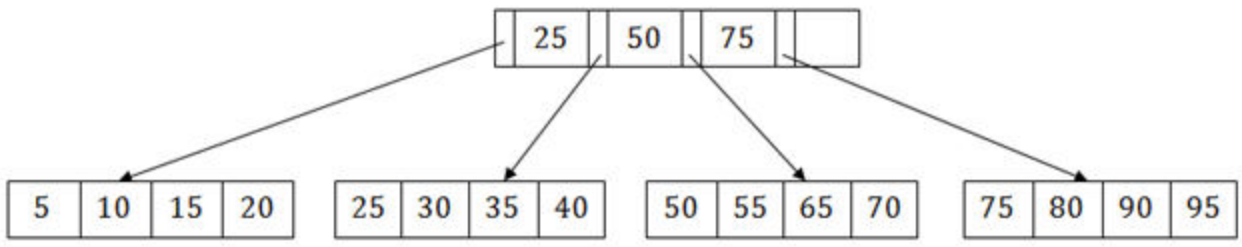


- So we can group $(50, 55)$ and $(60, 65, 70)$ into 2 leaf nodes
- If these two has to be leaf nodes, the intermediate node cannot branch from 50
- It should have 60 added to it, and then we can have pointers to a new leaf node
- This is how we can insert an entry when there is an overflow
    - In a normal scenario, it is very easy to find the node where it fits and then place it in that leaf node

B$^+$ Tree: Delete



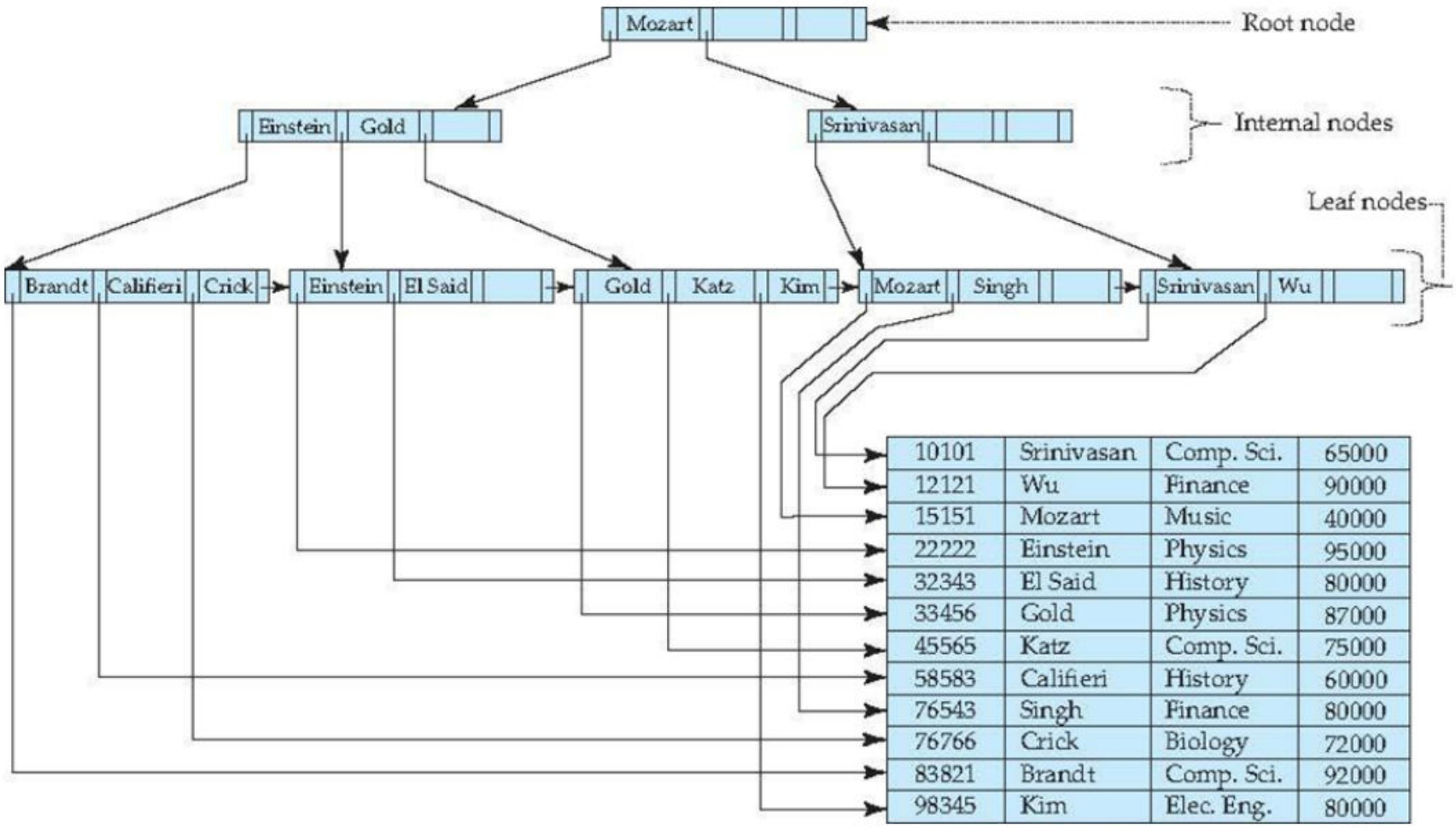
- To delete 60, we have to remove 60 from intermediate node as well as 4^{th} leaf node
- If we remove it from the intermediate node, then the tree will not remain a B^+ tree
- So, with deleting 60 we rearrange the nodes



B^+ Tree Index Files

- B^+ tree indices are an alternative to indexed-sequential files
- ***Disadvantages of ISAM files***
  - Performance degrades as file grows, since many overflow blocks get created
  - Periodic re-organization of entire file is required
- ***Advantages of B^+ tree index files***
  - Automatically re-organizes itself with small, local, changes in the face of insertions and deletions
  - Re-organization 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

B^+ Tree Index Files: Example



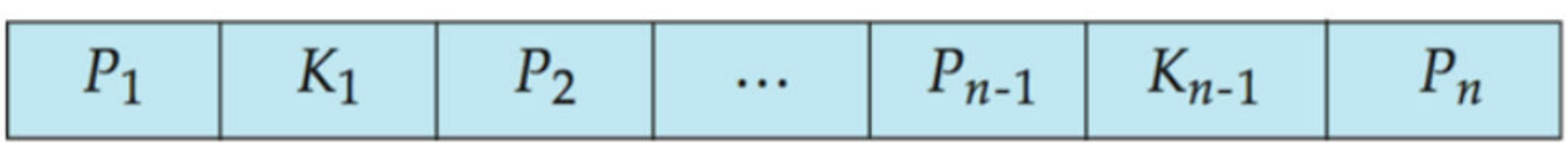
B^+ Tree Index Files: Structure

A B^+ tree is a rooted tree satisfying the following properties:

- All paths from the root to leaf are the same of the same length
- Each node that is not a root node or a leaf node has between $\lceil \frac{n}{2} \rceil$ and n children
- A leaf node has between $\lceil \frac{n-1}{2} \rceil$ 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 (that is, there are no other nodes in the tree) it can have between 0 and $(n-1)$ values

B^+ Tree Index Files: Node Structure

- Typical node



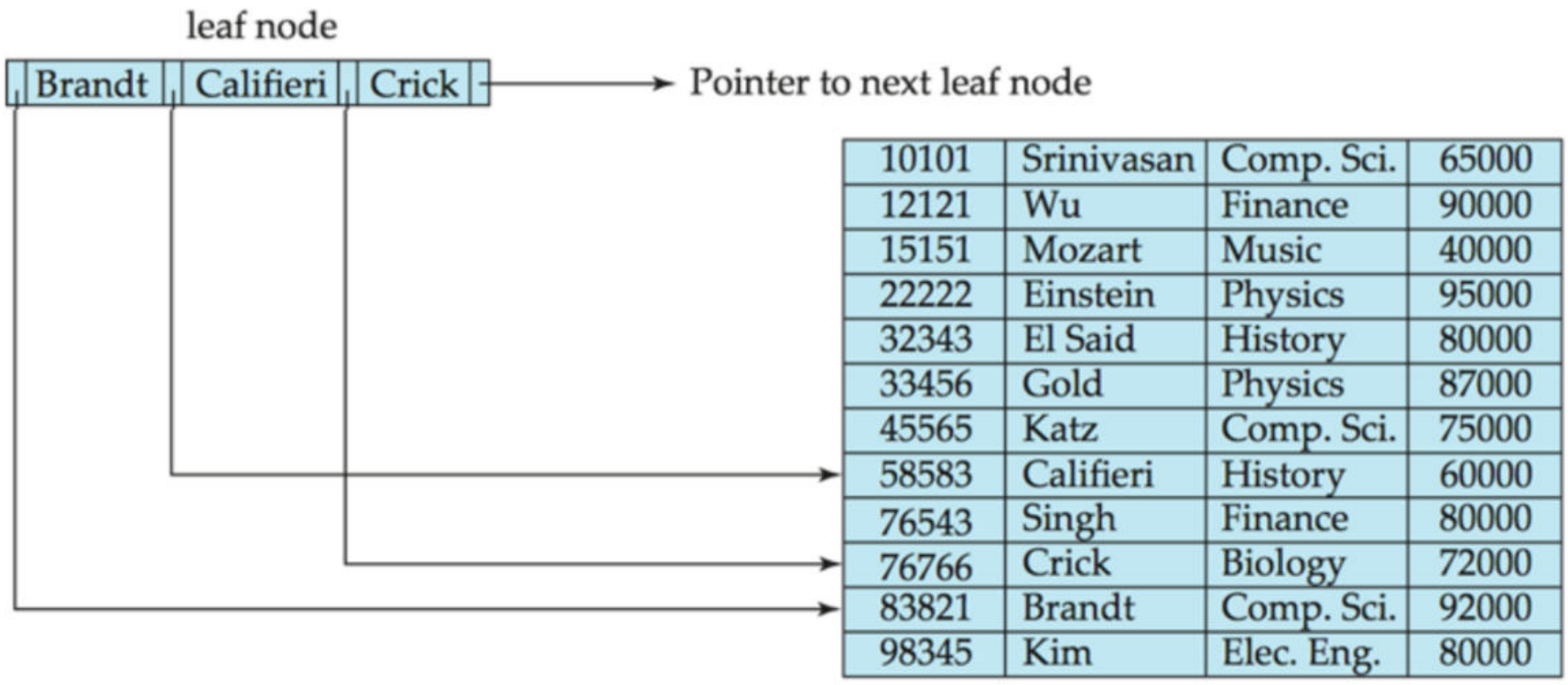
  - K_i are the search-key values
  - P_i are the pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes)
- The search-keys in a node are ordered

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

  (initially, assume no duplicate keys, address duplicates later)

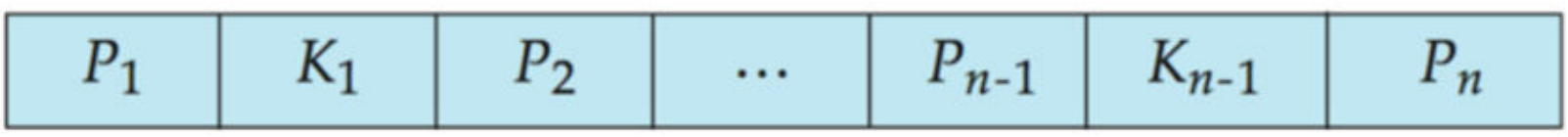
B^+ Tree Index Files: Leaf Nodes

- For $i = 1, 2, ..., n-1$, 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

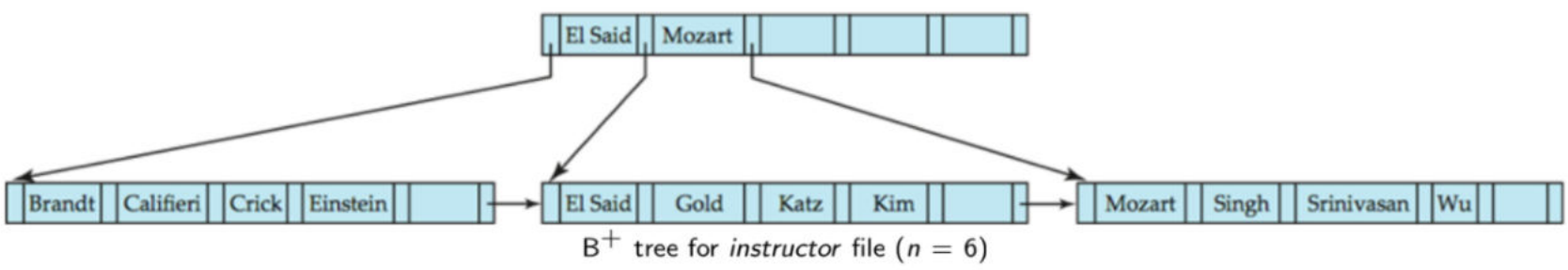


B$^+$ Tree Index Files: Non-Leaf Nodes

- 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 sub-tree to which P_1 points are less than K_1
  - For $2 \leq i \leq n-1$, all the search-keys in the sub-tree 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 sub-tree to which P_n points have values greater than or equal to K_{n-1}



B$^+$ Tree Index Files: Examples



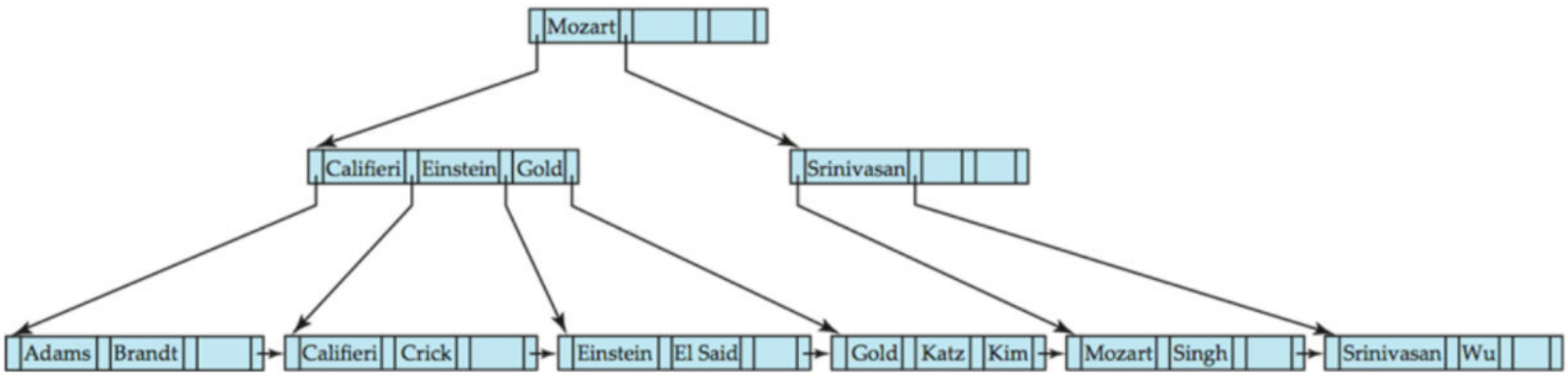
- Leaf nodes must have between 3 and 5 values: $\lceil \frac{n-1}{2} \rceil$ and $n-1$, with $n = 6$
- Non-leaf nodes other than root must have between 3 and 6 children: $\lceil \frac{n}{2} \rceil$ and n with $n = 6$
- Root must have at least 2 children

B$^+$ Tree Index Files: 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 indices
- The B$^+$ tree contains a relatively small number of levels
  - Level below root has at least $2 * \lceil \frac{n}{2} \rceil$
  - Next level has at least $2 * \lceil \frac{n}{2} \rceil * \lceil \frac{n}{2} \rceil$ values
  - ... etc.
  - If there are K search-key values in the file, the tree height is no more than $\lceil log_{\lceil \frac{n}{2} \rceil}(K) \rceil$
  - 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

B$^+$ Tree Index Files: Queries

- Find record with search-key value V
  - C = root
  - While C is not a leaf node
    - Let i be least value such that $V \leq K_i$
    - If no such exists, set C = last non-null pointer in C
    - Else $\{$if $(V = K_i)$ Set $C = P_{i+1}$ else set $C = P_i\}$
  - 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



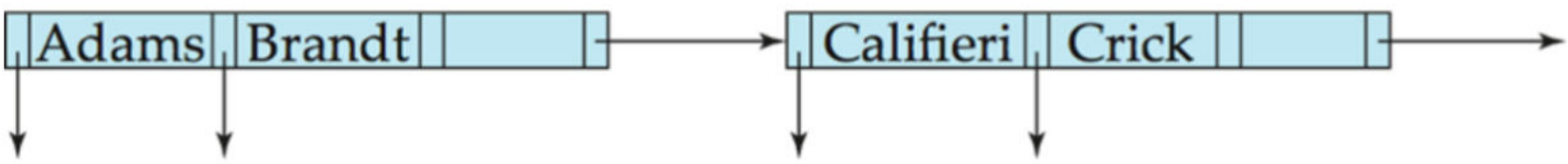
- If there are K search-key values in the file, the height of the tree is no more than $\lceil log_{\lceil \frac{n}{2} \rceil}(K) \rceil$
- A node is generally the same size as a disk block, typically 4 kilobytes
  - and n is typically around 100 (40 bytes per index entry)
- With 1 million search key 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 with a lookup
  - above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds

B$^+$ Tree Index Files: 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 sub-tree 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
- We modify the procedures 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 the first occurrence of V
  - traverse through consecutive leaves to find all occurrences of V

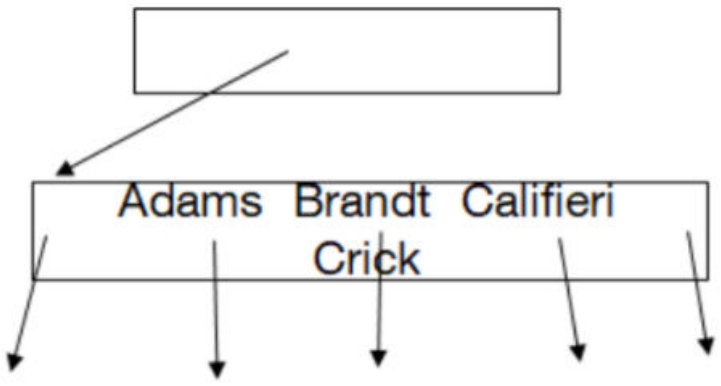
Updates on B$^+$ Trees: Insertion

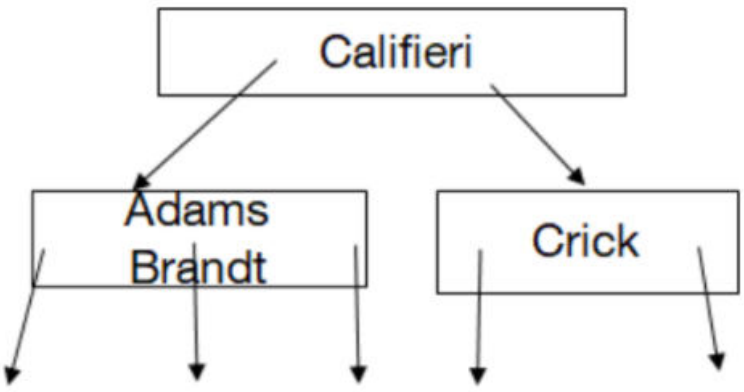
- Find the leaf node in which the search-key value would appear
- If the search-key value is already present in the leaf node
  - Add record to the file
  - If necessary, add a pointer to the bucket
- If the search-key value is not present, then
  - Add the record to the main file (and create a bucket if necessary)
  - If there is room in leaf node, insert (key-value, pointer) pair in the leaf node
  - Otherwise, split the node (along with the new (key-value, pointer) entry)
- Splitting a leaf node:
  - take the n (search-key value, pointer) pairs (including the one being inserted) in sorted order
    - Place the first $\lceil \frac{n}{2} \rceil$ 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



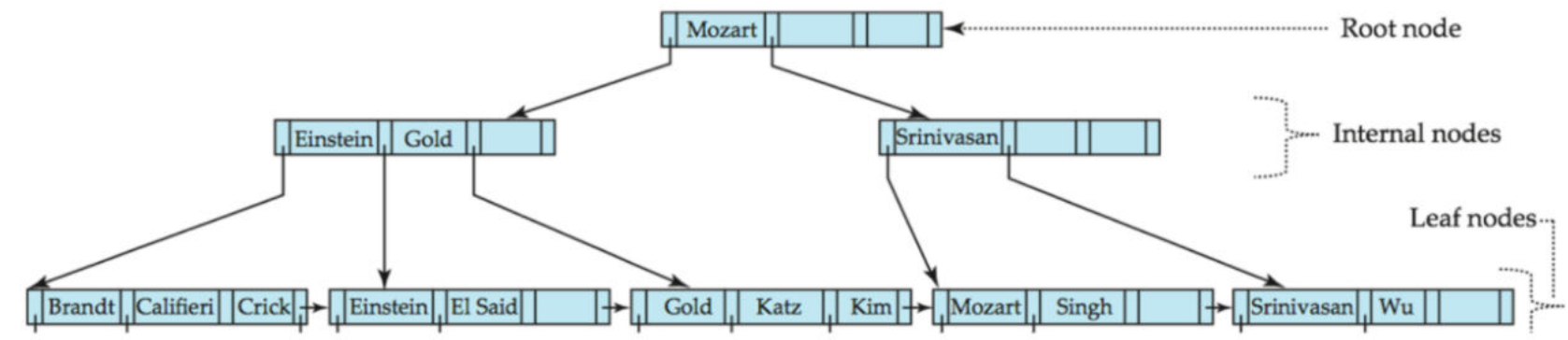
Result of splitting node containing Brandt, Califieri and Crick on inserting Adams
Next step: insert entry with (Califieri,pointer-to-new-node) into parent

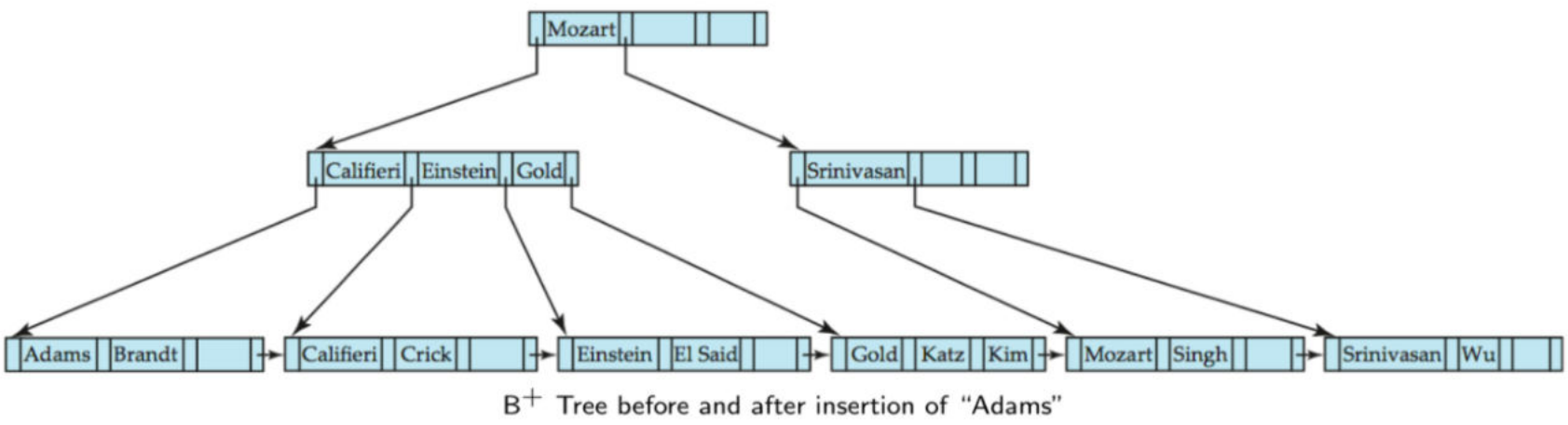
- Splitting a non-leaf node → when inserting (k, p) into an already full internal node N
  - Copy N to an in-memory area M with space for $n + 1$ pointers and n keys
  - Insert (k, p) into M
  - Copy $P_1, K_1, ..., K_{\lceil \frac{n}{2} \rceil - 1}, ..., P_{\lceil \frac{n}{2} \rceil}$ form M back into node N
  - Copy $P_{\lceil \frac{n}{2} \rceil + 1}, K_{\lceil \frac{n}{2} \rceil + 1}, ..., K_n, P_{n+1}$ from M into newly allocated node N'
  - Insert $(K_{\lceil \frac{n}{2} \rceil}, N')$ into parent N



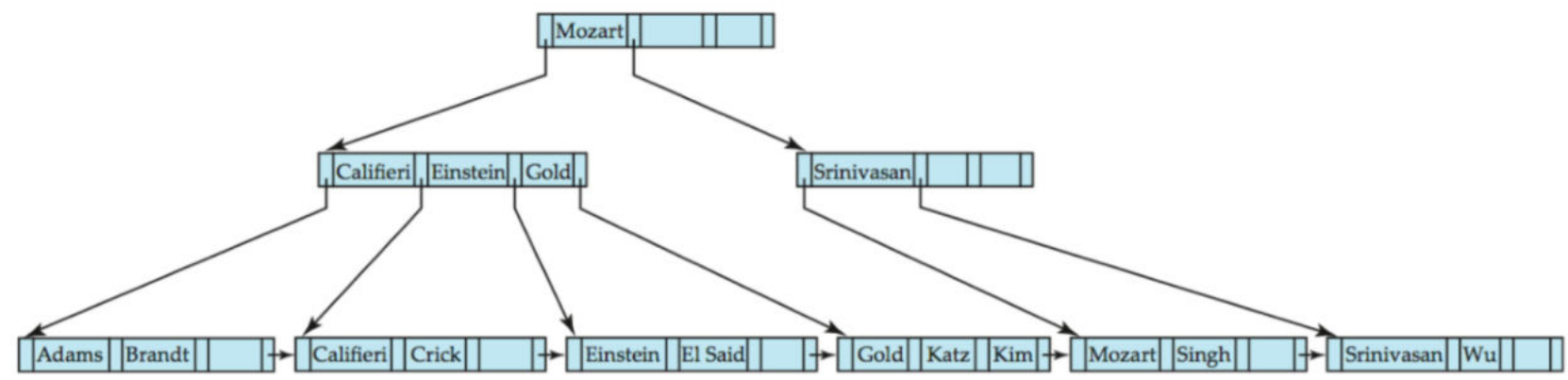


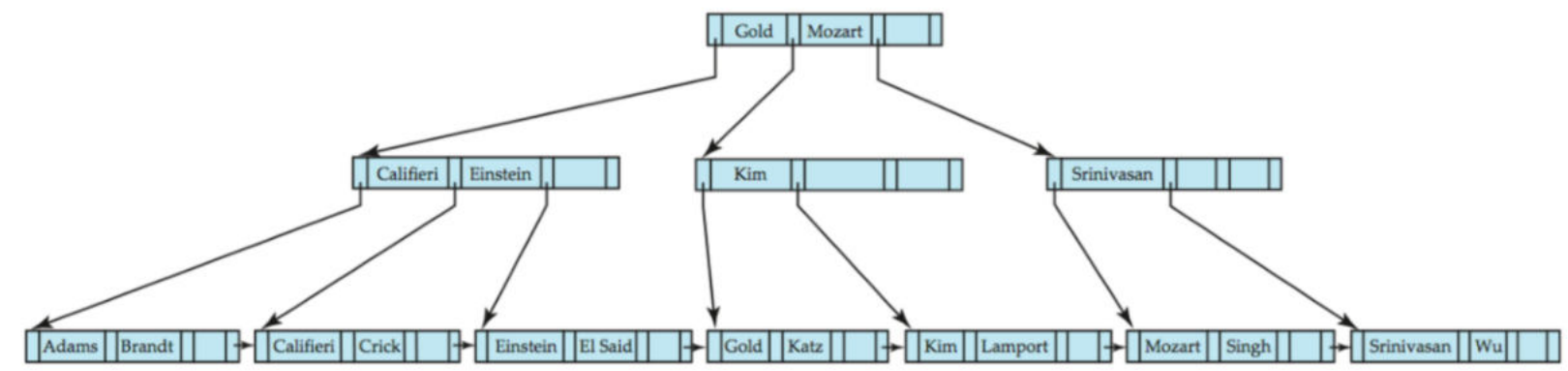
Updates on B$^+$ Trees: Insertion Example





B$^+$ Tree before and after insertion of "Adams"





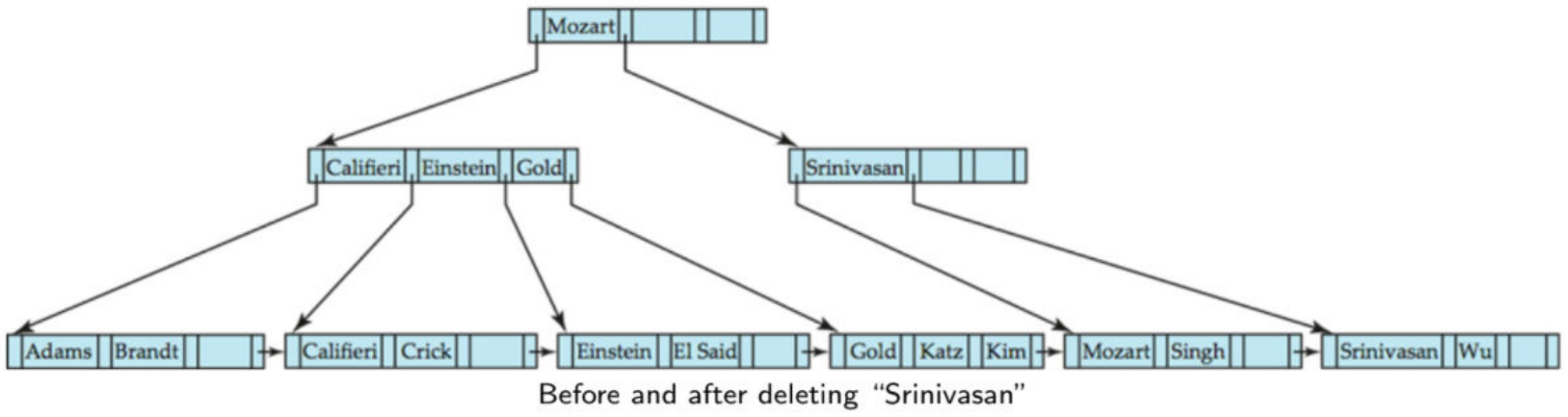
B$^+$ Tree before and after insertion of "Lamport"

Updates on B$^+$ Trees: Deletion

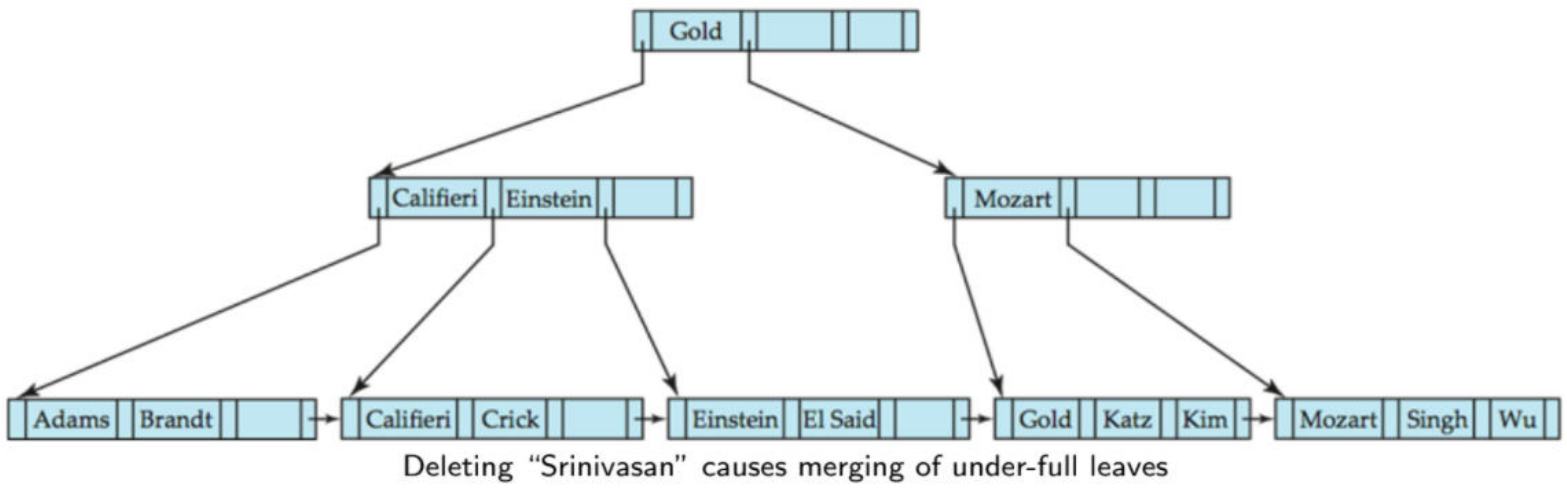
- Find the record to be deleted, and remove it from the main file and from the bucket (if present)
- Remove (search-key value, pointer) from the leaf node if there is no bucket or if the bucket has become empty
- If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then merge siblings:
  - Insert all the search-key values in the two nodes into a single node (the one of 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
- 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 $\lceil \frac{n}{2} \rceil$ 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

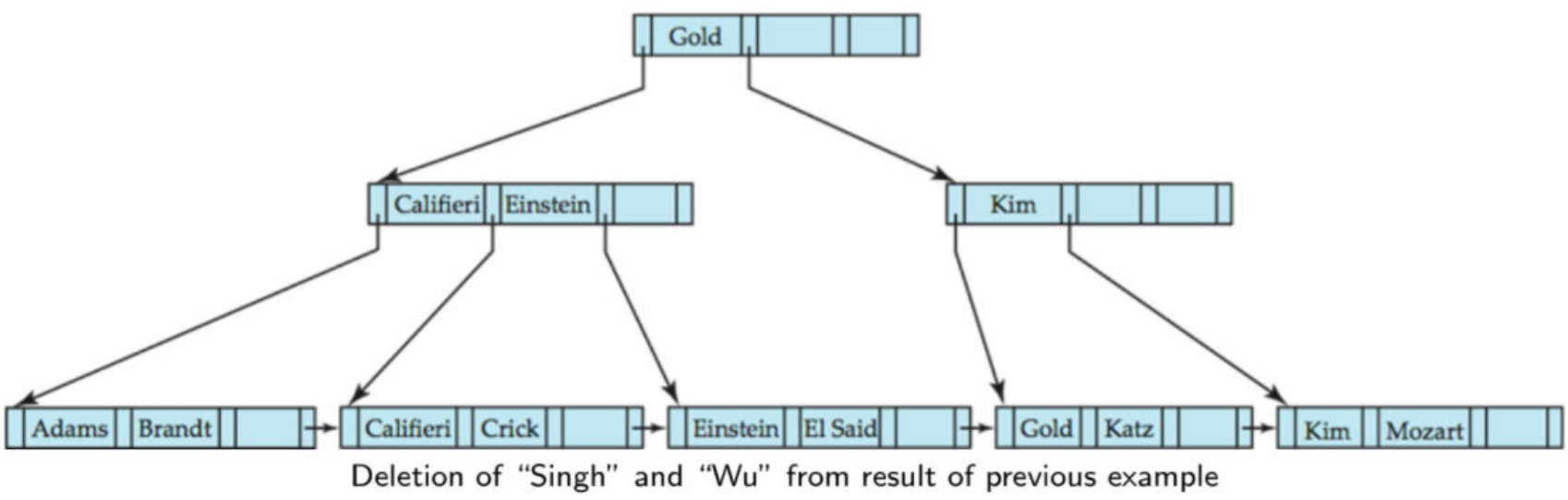
Updates on B$^+$ Trees: Deletion Example



Before and after deleting "Srinivasan"



Deleting "Srinivasan" causes merging of under-full leaves



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

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

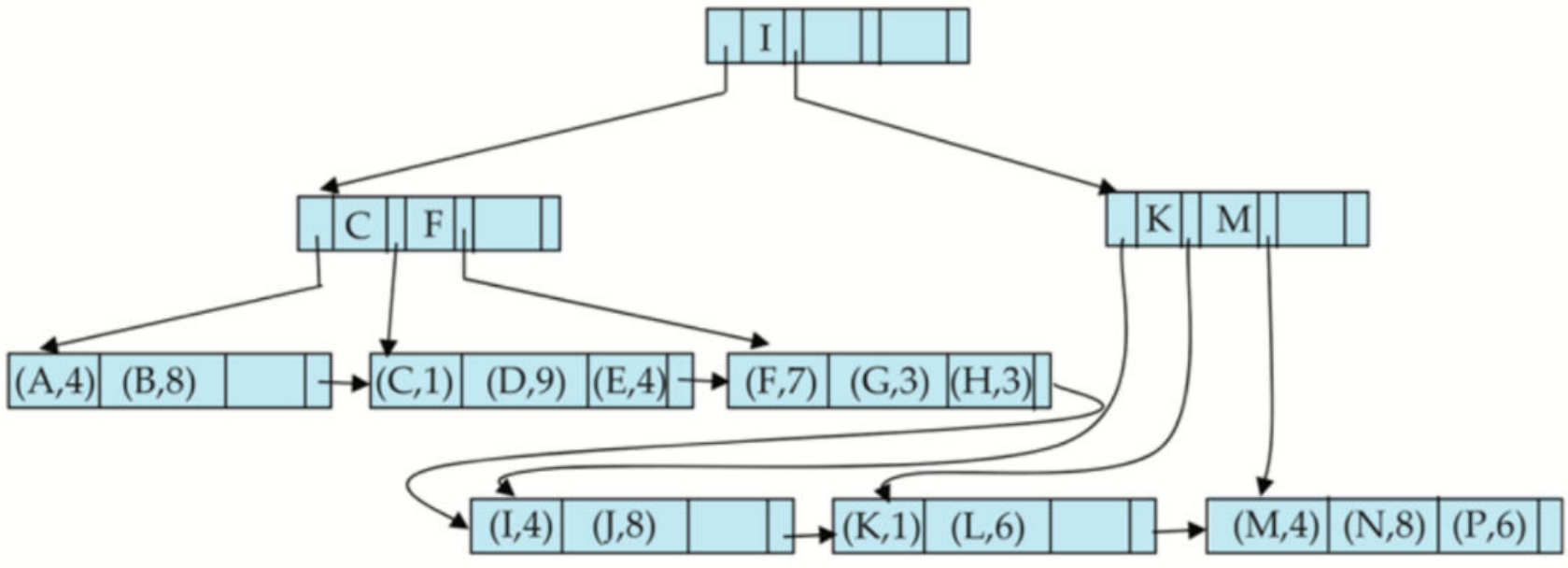
Before and after deletion of "Gold" from earlier example

- 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 deleted

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 is 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



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 $\lceil \frac{2n}{3} \rceil$ entries

Non-Unique Search Keys

- Alternatives to scheme described earlier
  - Buckets on separate block (bad idea)
  - List of tuple pointers with each key
    - Extra code to handle long lists
    - Deletion of a tuple can be expensive if there are many duplicates on search key
    - Low space overhead, no extra cost for queries
  - Make search key unique by adding a record-identifier
    - Extra storage overhead for keys
    - Simpler code for insertion/deletion
    - Widely used

Record Relocation and Secondary Indices

- If a record moves, all secondary indices that store record pointers have to be updated

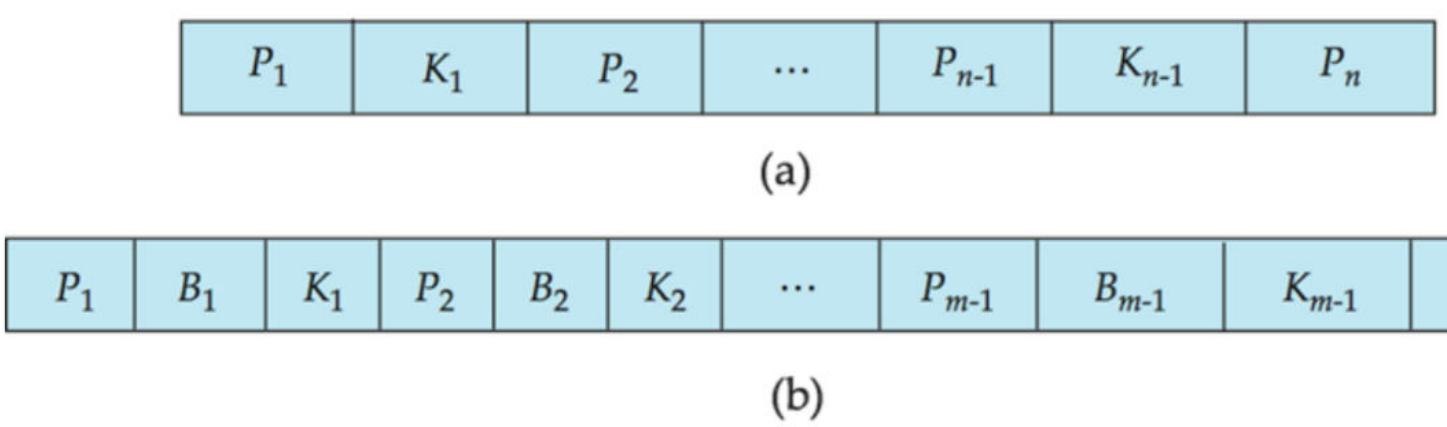
- Node splits in B^+ tree file organizations become very expensive
- ***Solution*** → Use primary-index search key instead of record pointer in secondary index
  - Extra traversal of primary index to locate record
    - Higher cost for queries, but node splits are cheap
  - Add record-id if primary-index search key is non-unique

Indexing Strings

- Variable length strings as keys
  - Variable fanout
  - Use space utilization as criterion for splitting, not numbers 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
      - For example → "Silas" and "Silberschatz" can be separated by "Silb
  - Keys in leaf nodes can be compressed by sharing common prefixes

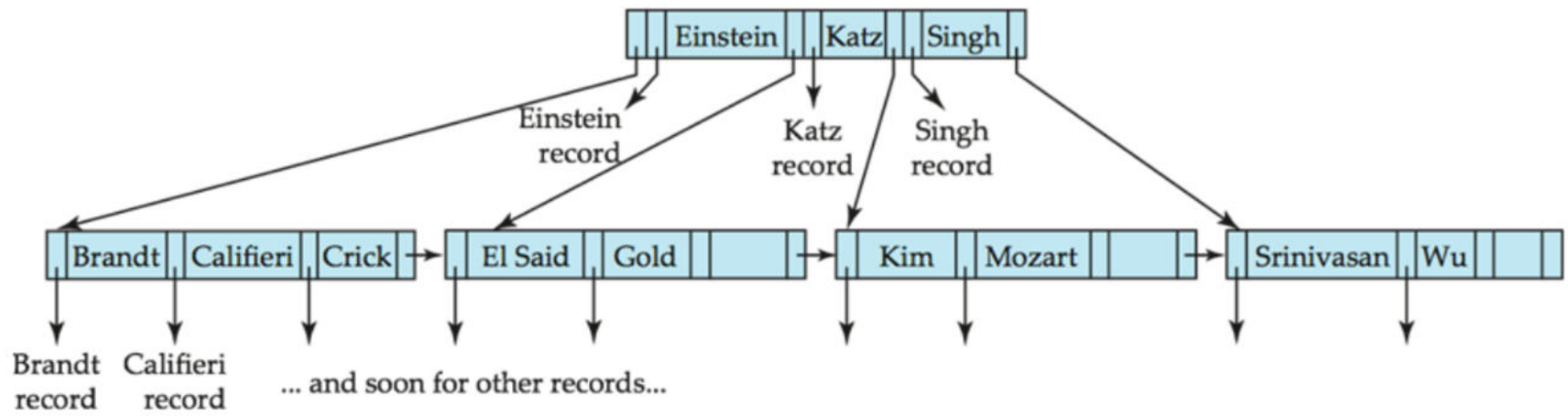
B-Tree Index Files

- 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

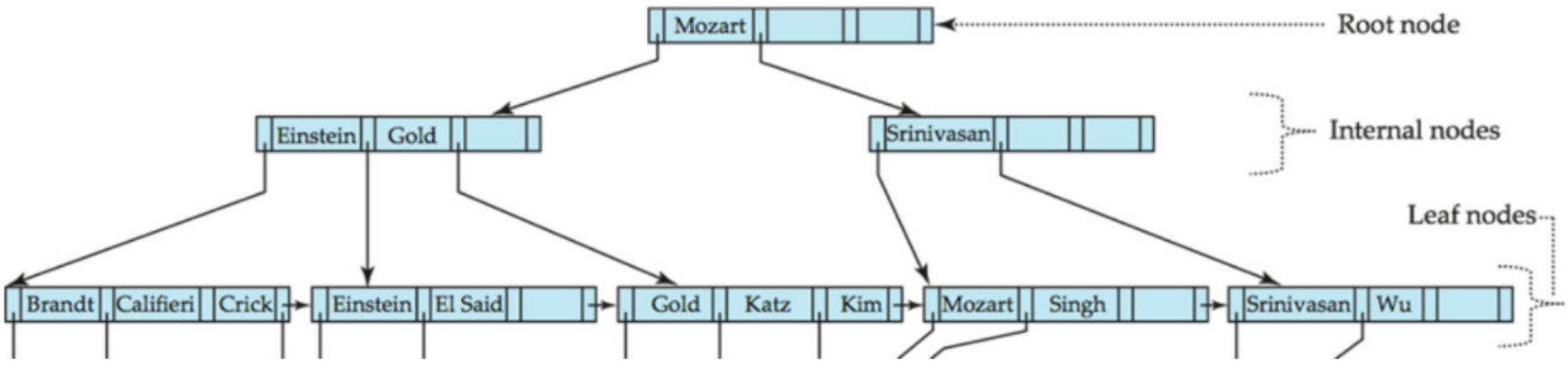


- Non-leaf node → pointers Bi are the bucket or file record pointers

B-Tree Index Files: Example



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



Comparison of B-Tree and B^+ Tree Index Files

- Advantages of B-Tree indices:
  - May use less tree nodes than a corresponding B^+ Tree
  - Sometimes possible to find search-key value before reaching the 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^+ Trees
  - Implementation is harder than B^+ Trees
- Typically, advantages of B-Trees do not outweigh the disadvantages

Week 9 Lecture 4

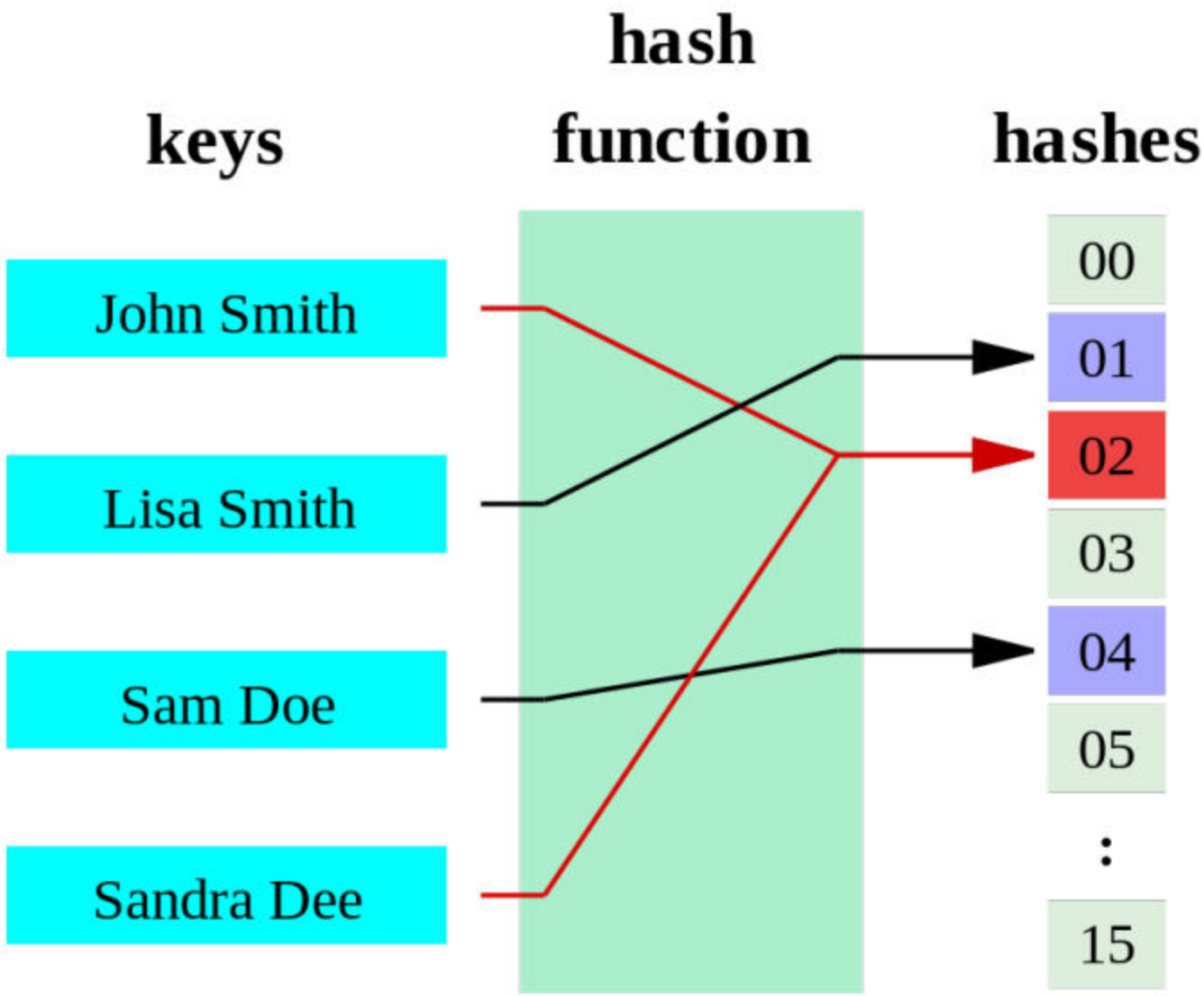
Class	BSCCS2001
Created	@November 3, 2021 5:55 PM
Materials	
Module #	44
Type	Lecture
Week #	9

Indexing and Hashing → Hashing

Static Hashing

Hash function

- A hash function h maps data of arbitrary size (from domain D) to fixed-size values (say, integers from 0 to $N > 0$ $h : D \to [0..N]$)
- Given key k, $h(k)$ is called hash values, hash codes, digests or simply hashes
- If for two keys $k_1 \neq k_2$, we have $h(k_1) = h(k_2)$, we say a collision has occurred
- A hash function should be Collision free and Fast



Static hashing

- A ***bucket*** is a unit of storage containing one or more records (a bucket is typically a disk block)
- In a ***hash file organization*** we obtain the bucket of a record directly from its search-key value using a ***hash function***
- Hash function h is a function from the set of all search-key values K to the set of all bucket addresses B
- Hash function is 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 key

- There are 10 buckets
- The binary representation of the i^{th} character is assumed to be integer i
- The hash function returns the sum of the binary representations of the characters modulo 10
  - For example

    $h(Music) = 1 \qquad h(History) = 2$

    $h(Physics) = 3 \quad h(Elec.\ Eng.) = 3$

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

Hash file organization of instructor file, using dept_name as key

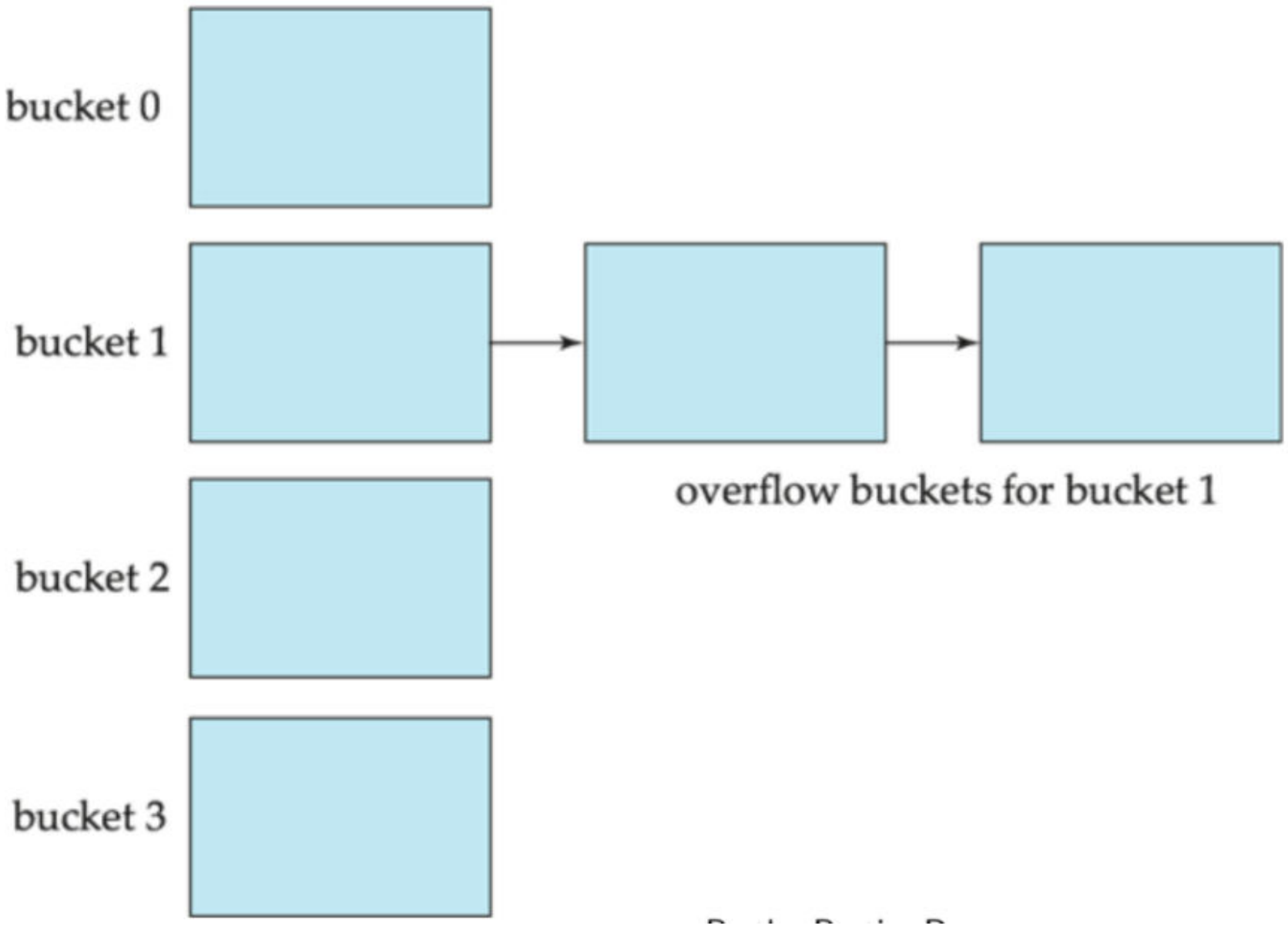
Hash functions

- 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 is uniform ie. each bucket is assigned the same number of search-key values from the set of all possible values
- Ideal hash functions is random, so 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
    - For example, for a string search-key, the binary representations of all the characters in the string could be added and the sum modulo the number of buckets could be returned

Handling of bucket overflows

- Bucket overflow can occur because of
    - Insufficient buckets
    - Skew in distribution of records
        - This can occur due to two reasons:
            - Multiple records have the same search-key value
            - Chose 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
- **Overflow chaining** → the overflow buckets of a given bucket are chained together in a linked list
- Above scheme is called ***closed hashing***

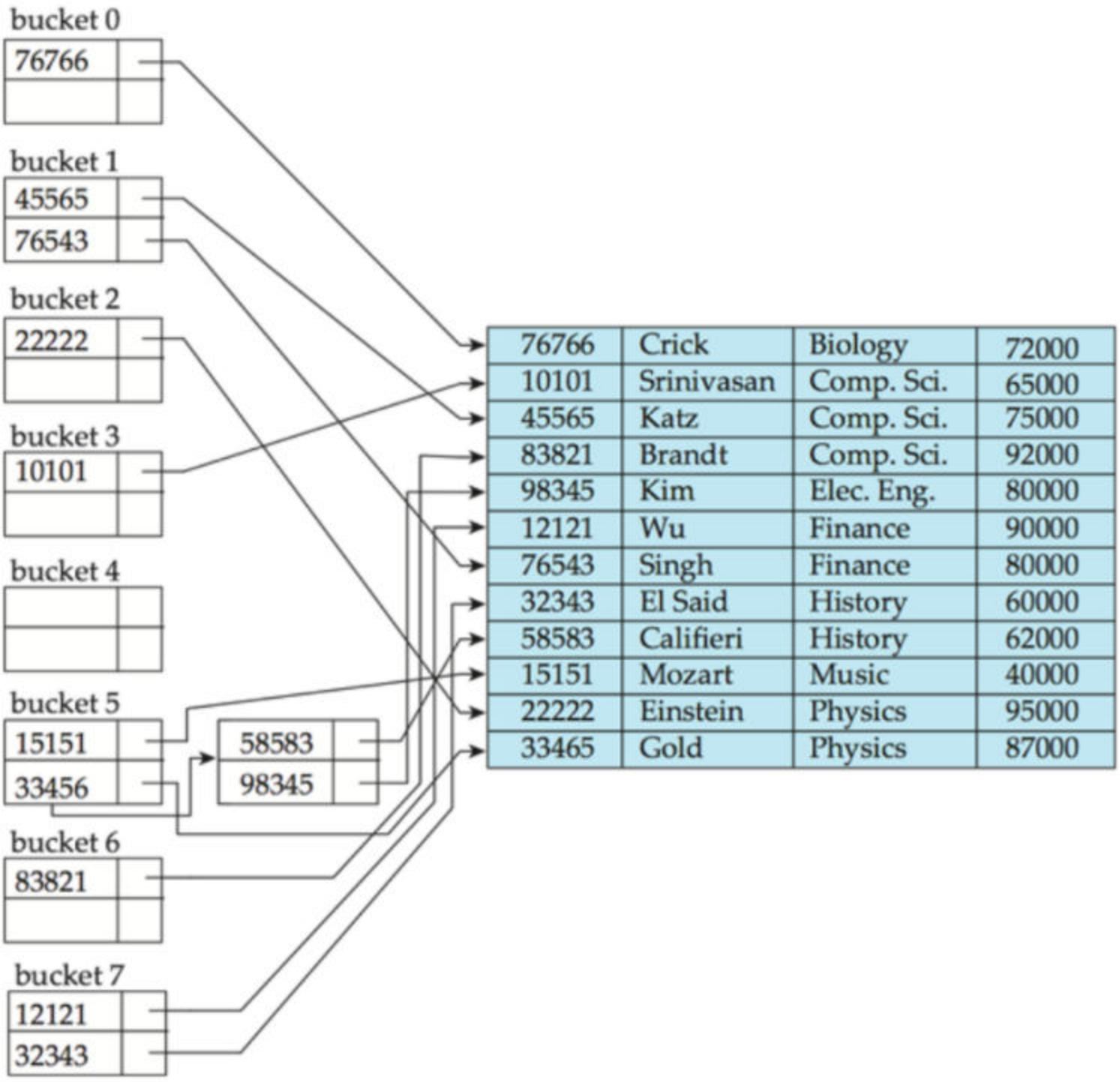
- An alternative, called ***open hashing***, which does not use overflow buckets, is not suitable for database applications



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 secondary indices
  - If the file itself is organized using hashing, a separate primary hash index on it using the same search-key is unnecessary
  - However, we use the term hash index to refer to both secondary index structures and hash organized files

Example of Hash Index



- Hash index on instructor, on attribute ID
- Computed by adding the digits modulo 8

Deficiencies of Static Hashing

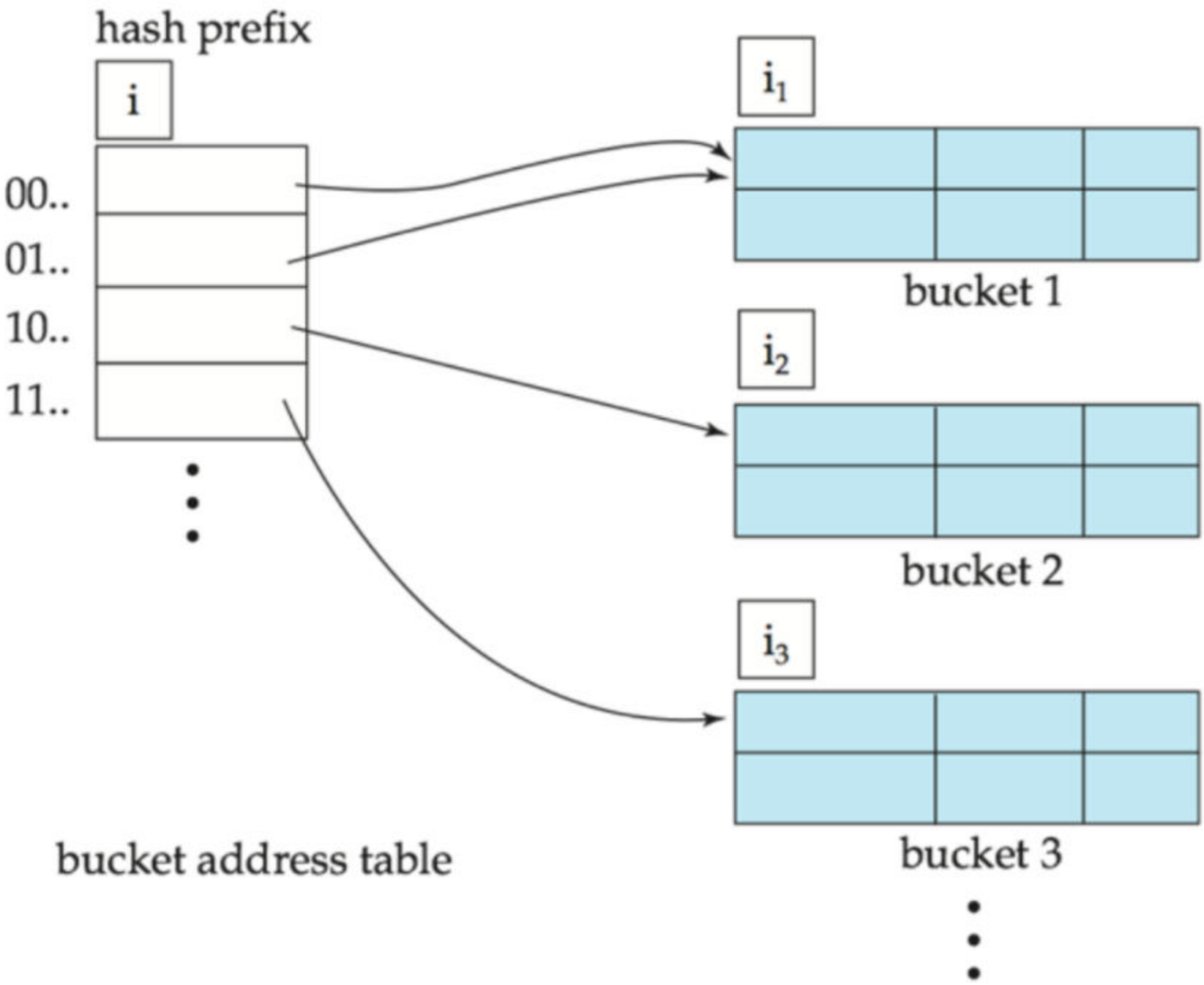
- In static hashing, function h maps search-key values to a fixed set of B of bucket addressed
  - Databases grow or shrink with time
    - If initial number of buckets is too small, and the file grows, performance will degrade due to too much overflows
    - If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and the buckets will be underfull)
    - 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
- **Extendable hashing** → one form of dynamic hashing
  - Hash function generates values over a large range — typically b-bit integers with b = 32
  - At any time, use only a prefix of the hash function to index into a table of bucket addresses
  - Let the length of the prefix be i bits, $0 \leq i \leq 32$

- Bucket address table size $= 2^i$
  - Initially, $i = 0$
  - Value of i grows and shrinks as the size of the database grows and shrinks
- Multiple entries in the bucket address table may point to a bucket (Why?)
- Thus, actual number of buckets is $< 2^i$
  - The number of buckets also changes dynamically due to coalescing and splitting of buckets

General Extendable Hash Structure



In this structure, $i_2 = i_3 = i$, whereas $i_1 = i - 1$

Decode i_j number of bits to find the record in bucket j

$i_j \leq i$

Use of Extendable Hash Structure

- Each bucket j stores a value i_j
  - All the entries that point to the same bucket have the same values on the first i_j bits
- To locate the bucket containing search-key K_j
  - Compute $h(K_j) = X$
  - Use the first i high order bits of X as a displacement into bucket address table and follow the pointer to appropriate bucket
- To insert a record with search-key value K_j
  - Follow same procedure as look-up and locate the bucket, say j
  - If there is room in the bucket j insert record in the bucket
  - Else the bucket must be split and insertion re-attempted

    - Overflow buckets used instead of some cases

Insertion in Extendable Hash Structure

To split a bucket j when inserting record with search-key value K_j

- If $i > i_j$ (more than one pointer to bucket j)
    - Allocate a new bucket z, and set $i_j = i_z = (i_j + 1)$
    - Update the second half of the bucket address table entries originally pointing to j, to point to z
    - Remove each record in the bucket j and re-insert (in j or z)
    - Re-compute new bucket for K_j and insert record in the bucket (further splitting is required if the bucket is still full)
- If $i = i_j$ (only one pointer to bucket j)
    - If i reach some limit b, or too many splits have happened in this insertion, create an overflow bucket
    - Else
        - Increment i and double the size of the bucket address table
        - Replace each entry in the table by 2 entries that point to the same bucket
        - Re-compute new bucket address table entry for K_j
            - Now, $i > i_j$ so use the first case above

Deletion in Extendable Hash Structure

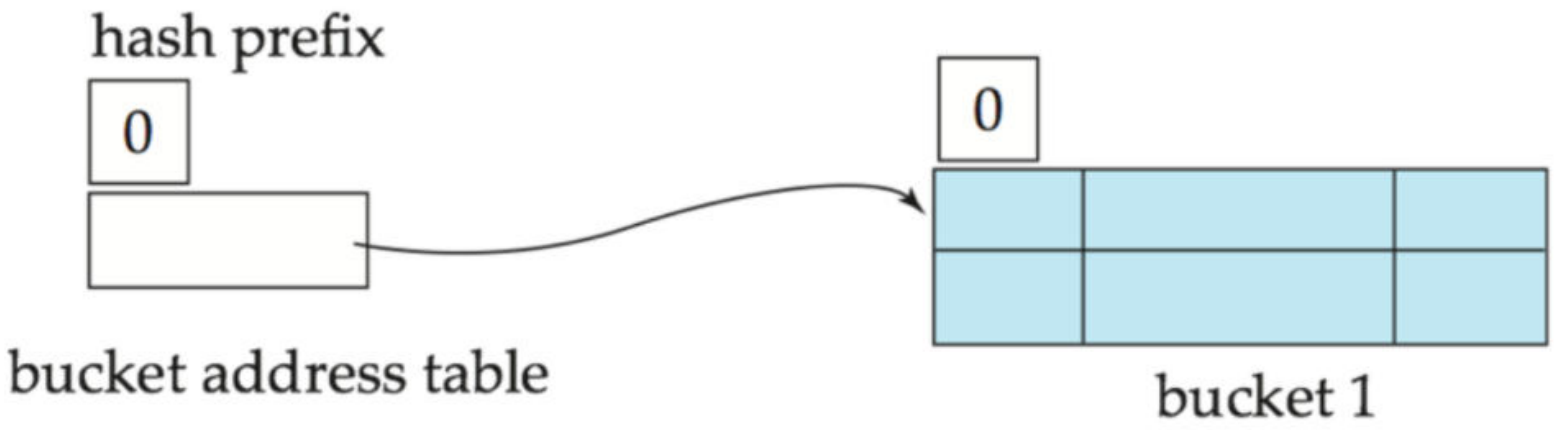
- To delete a key value
    - locate it in its bucket and remove it
    - The bucket itself can be removed if it becomes empty (with appropriate updates to the bucket address table)
    - Coalescing of buckets can be done (can coalesce only with a "buddy" bucket having same value of i_j and $i_j - 1$ prefix, if it is present)
    - Decreasing bucket address table size is also possible
        - Note → decreasing bucket address table size is an expensive operation and should be done only if number of buckets becomes much smaller than the size of the table

Use of Extendable Hash Structure: Example

dept_name	h(*dept_name*)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

Example

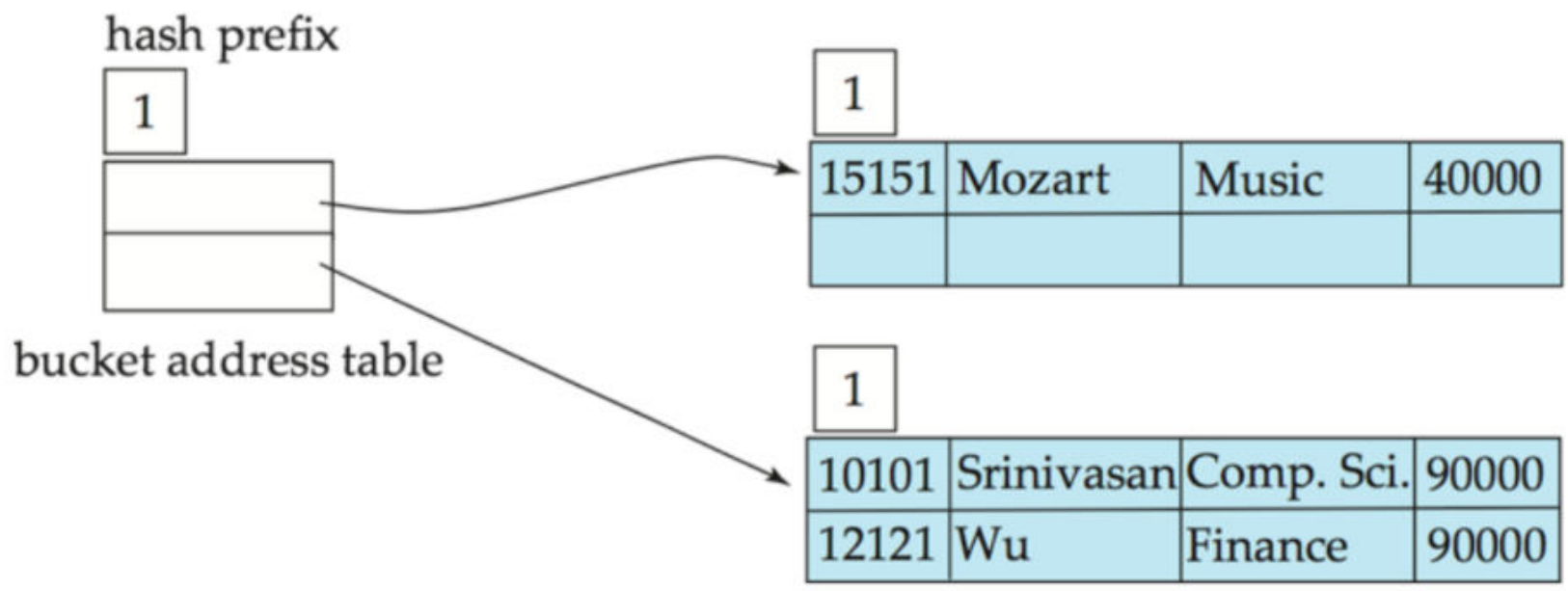
- Initial Hash structure; bucket size = 2



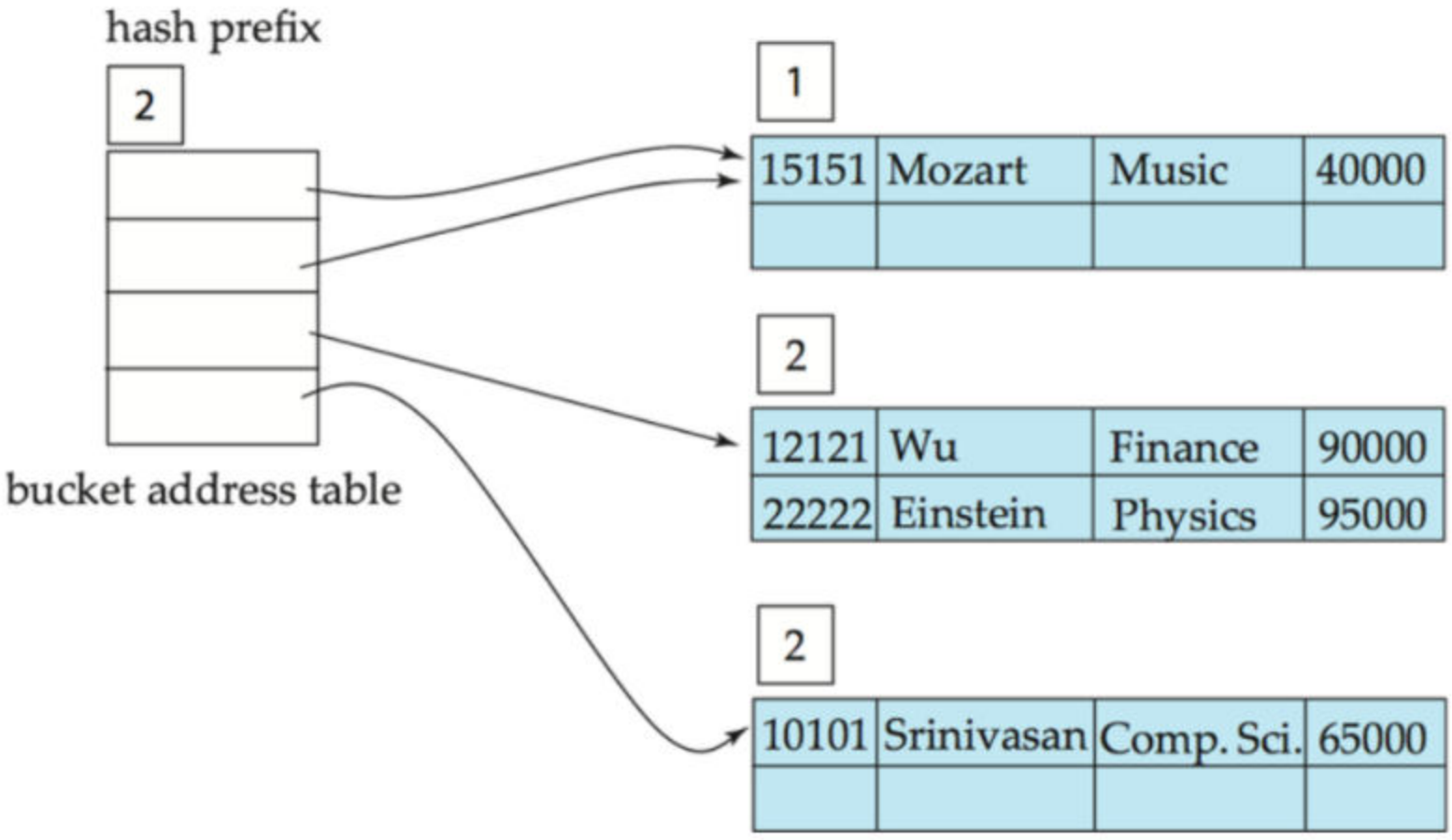
- Insert "Mozart", "Srinivasan" and "Wu" records

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

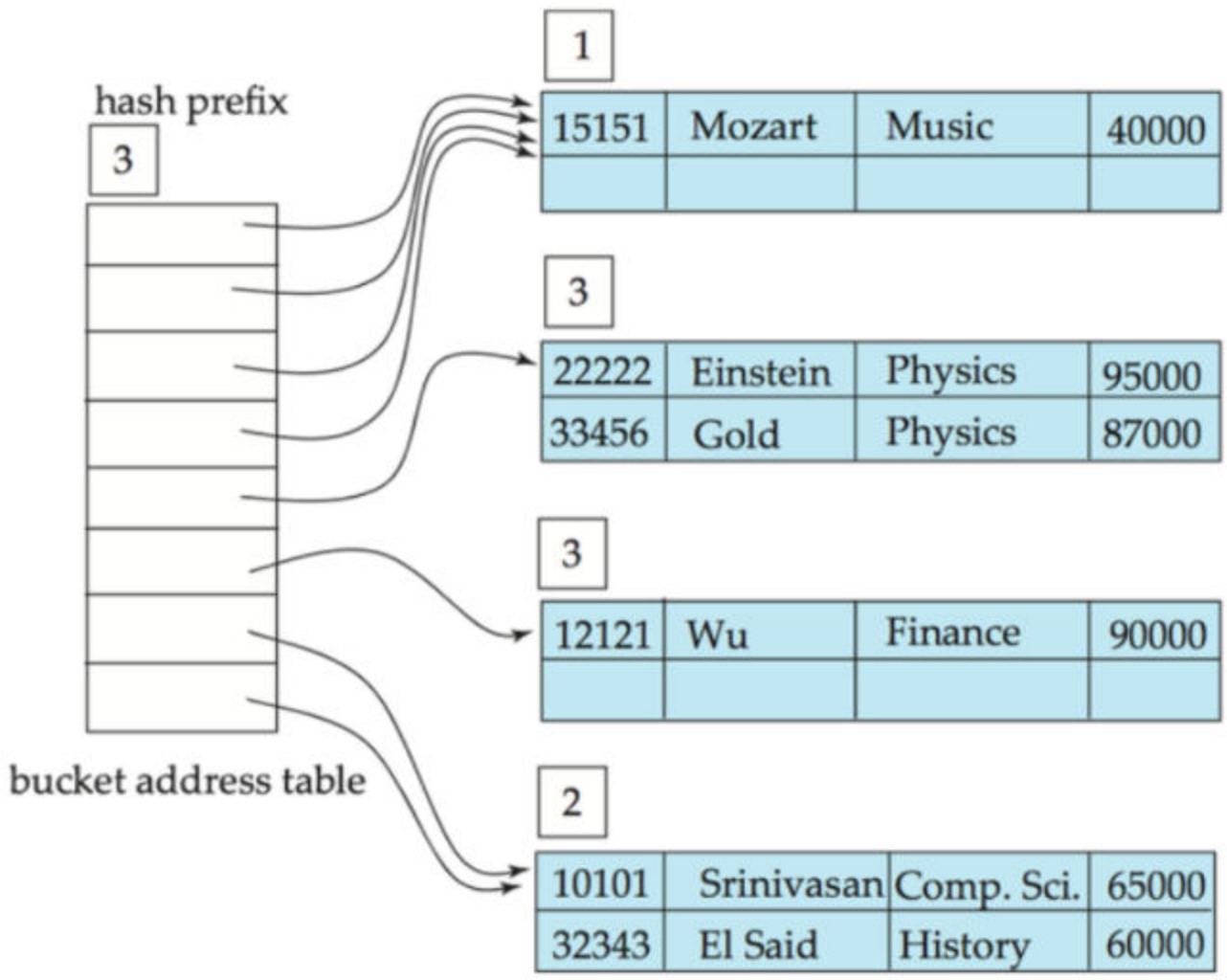
- Hash structure after insertion of "Mozart", "Srinivasan" and "Wu" records



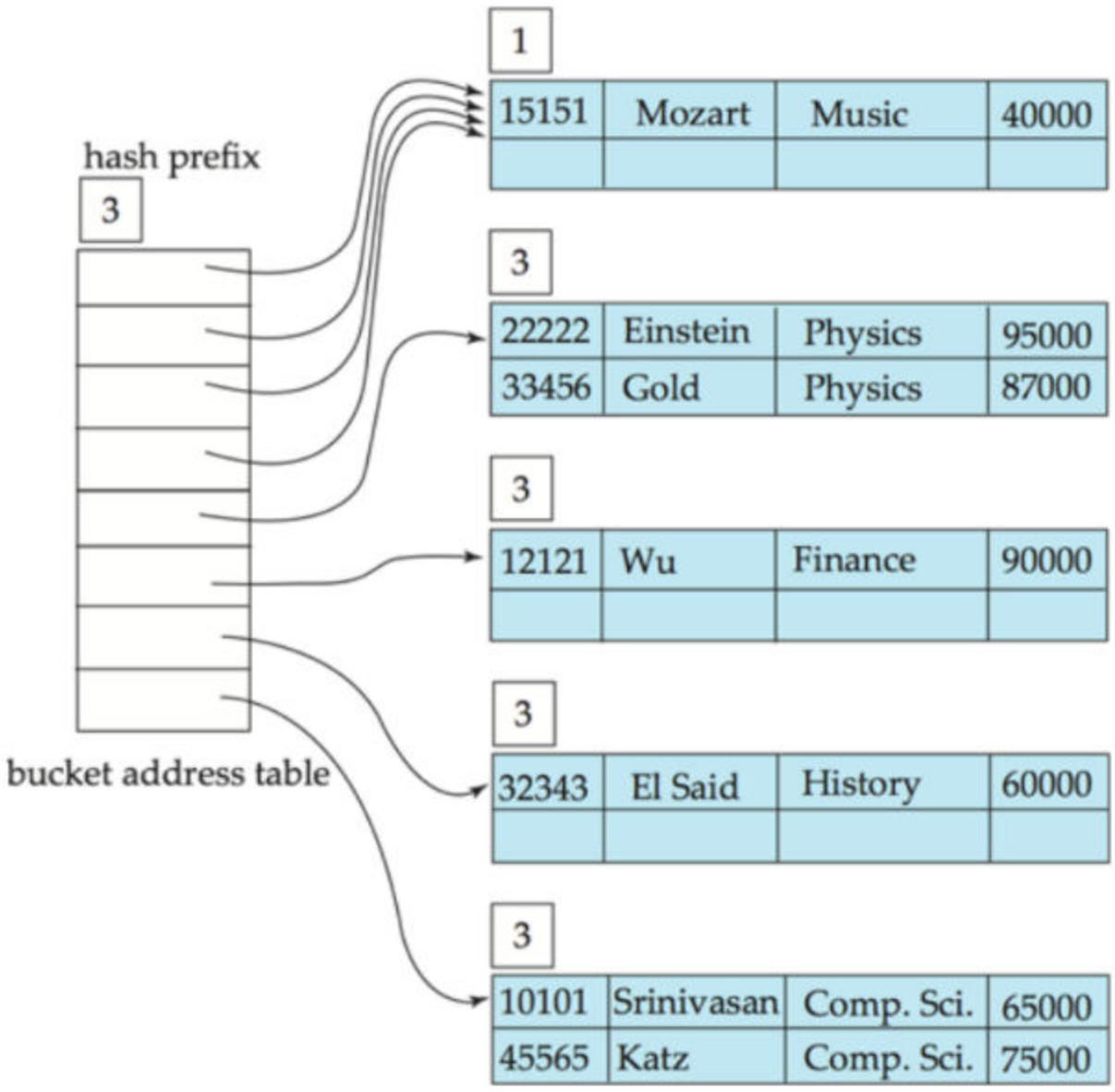
- Insert Einstein record
- Hash structure after insertion of "Einstein" record



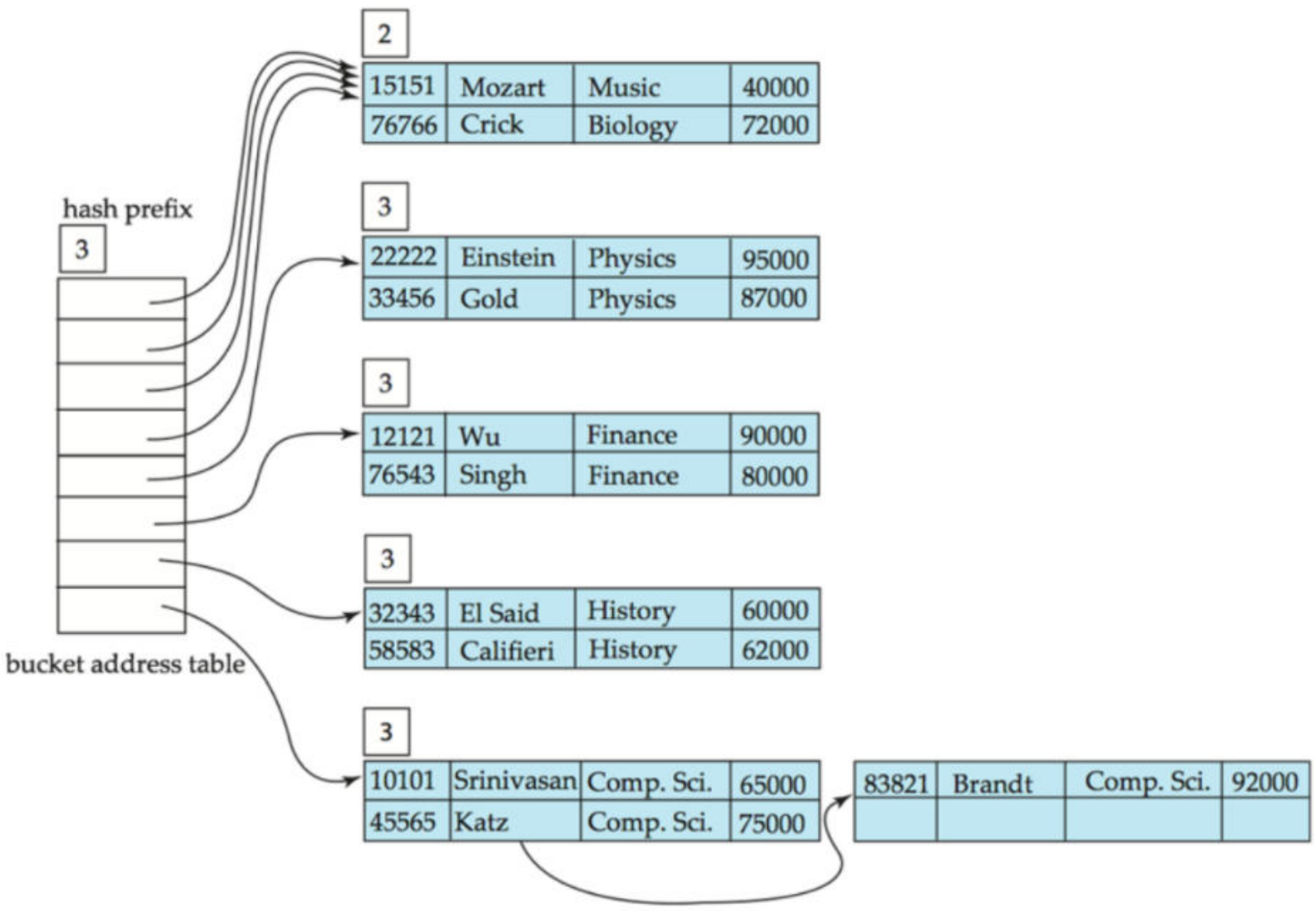
- Insert "Gold" and "El Said" records
- Hash structure after insertion of "Gold" and "El Said" records



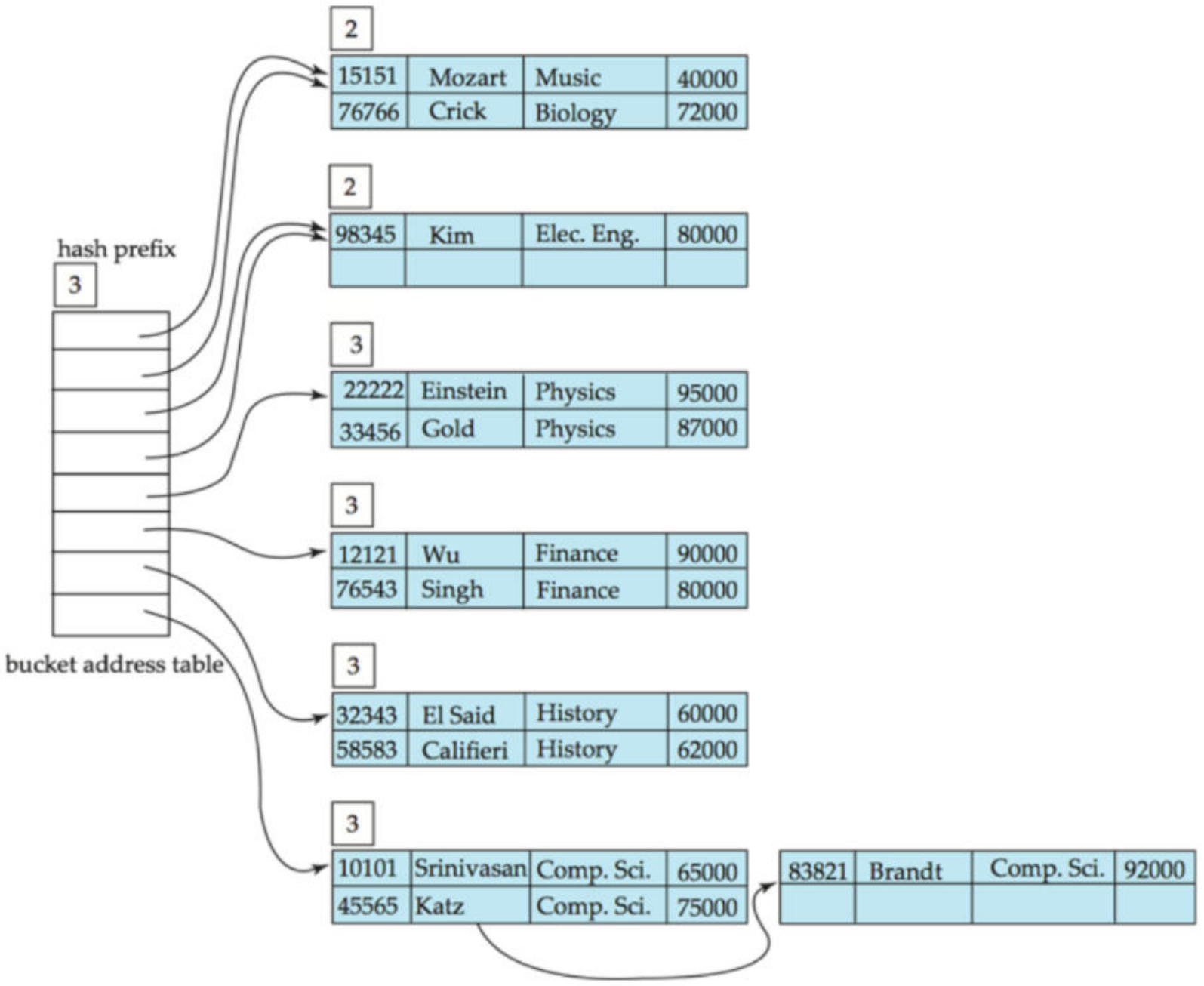
- Insert Katz record
- Hash structure after insertion of "Katz" record



- Insert "Singh", "Califieri", "Crick", "Brandt" records
- Hash structure after insertion of "Singh", "Califieri", "Crick", "Brandt" records



- Insert Kim record
- Hash structure after insertion of "Kim" record



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

Comparison Schemes

Extendable Hashing vs Other Schemes

- Benefits of extendable hashing
  - Hash performances does not degrade with growth of file
  - Minimal space overhead
- Disadvantages of extendable hashing
  - Extra level of indirection to find the desired record
  - Bucket address table may itself become very big (larger than memory)
    - Cannot allocate very large contiguous areas on the disk either
    - Solution $\rightarrow$ B^+-tree structure to locate desired record in bucket address table
  - Changing size of bucket address table is an expensive operation

- Linear hashing is an alternative mechanism
  - Allows incremental growth of its directory (equivalent to bucket address table)
  - At the cost of more bucket overflows

Comparison of Ordered Indexing and Hashing

- Cost of periodic re-organization
- Relative frequency of insertions and deletion
- Is it desirable to optimize average access time at the expense of worst-case access time?
- Expected type of queries:
  - Hashing is generally better at retrieving records having a specified value of the key
  - If range queries is common, ordered indices are to be preferred
- **In practice:**
  - PostgreSQL supports hash indices, but discourages use due to poor performance
  - Oracle supports static has organization, but not has indices
  - SQLServer supports only B^+ Trees

Bitmap Indices

- Bitmap indices are a special type of index designed for efficient querying on multiple keys
- Records in a relation are assumed to be numbered sequentially from, say 0
  - Given a number n it must be easy to retrieve record n
    - Particularly easy if records are of fixed size
- Applicable on attributes that take on a relatively small number of distinct values
  - For example, gender, country, state, ...
  - For example, income-level (income broken up into small number of levels such as 0-9999, 10000-19999, 20000-50000, 50000-infinity)
- A bitmap is simply an array of bits
- In its simplest form a bitmap index on an attribute has a bitmap for each value of the attribute
  - Bitmap has as many bits as records
  - In a bitmap for value v, the bit for a record is 1 if the record has the value v for the attribute and is 0 otherwise

record number	*ID*	*gender*	*income_level*
0	76766	m	L1
1	22222	f	L2
2	12121	f	L1
3	15151	m	L4
4	58583	f	L3

Bitmaps for *gender*

m	10010
f	01101

Bitmaps for *income_level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

- Bitmap indices are useful for queries on multiple attributes
  - not particularly useful for single attribute queries
- Queries are answered using bitmap operations

    - Intersection (AND)
    - Union (OR)
    - Complementation (NOT)
- Each operation takes two bitmaps of the same size and applies the operation on corresponding bits to get the result bitmap
    - For example, 100110 AND 110011 = 100010
        - 100110 OR 110011 = 110111
        - NOT 100110 = 011001
    - Males with income level L1: 10010 AND 10100 = 10000
        - Can then retrieve required tuples
        - Counting number of matching tuples is even faster
- Bitmap indices generally very small compared with relation size
    - For example, if record is 100 bytes, space for a single bitmap is 1/800 of space and by relation
        - If number of distinct attribute values is 8, bitmap is only 1% of relation size
- Deletion needs to be handled properly
    - Existence bitmap to note if there is a valid record at a record location
    - Needed for complementation
        - not(A=v): (NOT bitmap-A-v) AND ExistenceBitmap
- Should keep bitmaps for all values, even null value
    - To correctly handle SQL null semantics for NOT(A=v):
        - intersect above result with (NOT bitmap-A-Null)

Bitmap Indices: Efficient Bitmap Operations

- Bitmaps are packed into words; a single word and (a basic CPU instruction) computes and of 32 or 64 bits at once
    - For example, 1-million-bit maps can be and-ed with just 31,250 instructions
- Counting number of 1s can be done fast by a trick:
    - Use each byte to index into a pre-computed array of 256 elements each storing the count of 1s in the binary representation
        - Can use pairs of bytes to speed up further at a higher memory cost
    - Add up the retrieved counts
- Bitmaps can be used instead of Tuple-ID lists at leaf levels of B^+-trees for values that have a large number of matching records
    - Worthwhile, if $> 1/64$ of the records have that value, assuming a tuple-id is 64 bits
    - Above technique merges benefits of bitmap and B^+-tree indices

Week 9 Lecture 5

Class	BSCCS2001
Created	@November 4, 2021 2:11 PM
Materials	
Module #	45
Type	Lecture
Week #	9

Indexing and Hashing → Index Design

Index Definition in SQL

Index in SQL

- Create an index

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

  For example: **create index** *b-index* **on** *branch* (*branch_name*)
- Use **create unique index** to indirectly specify and enforce the condition that the search key is a candidate key
  - Not really required if SQL **unique** integrity constraint is supported — it is preferred
- To drop an index

  `drop index <index-name>`
- Most database systems allow specification of type of index and clustering
  - You can also create an index for a cluster
  - You can create a composite index on multiple columns up to a maximum of 32 columns
    - A composite index key cannot exceed roughly one-half (minus some overhead) of the available space in the data block

Index in SQL: Examples

- Create an index for a single column, to speed up queries that test that column:
  - CREATE INDEX emp_ename ON emp_tab(ename);

- Specify several storage settings explicitly for the index

```
CREATE INDEX emp_ename ON emp_tab(ename)
  TABLESPACE users // Allocation of space in the Database to contain schema objects
  STORAGE ( // Specify how Database should store a database object
    INITIAL 20K // Specify the size of the 1st extent of the object
    NEXT 20K // Specify in bytes the size of the 2nd extent to be allocated to the object
    PCTINCREASE 75) // Specify the percent by which later extents grow over
    PCTFREE 0 // 0% of each data block this table's data segment be free for updates
    COMPUTE STATISTICS;
```

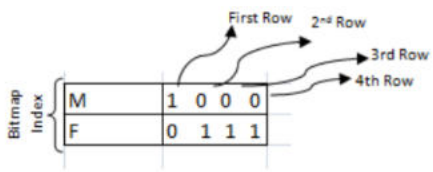
- Create index on two columns, to speed up queries that test either the first column or both columns
  - CREATE INDEX emp_ename ON emp_tab(ename, empno) COMPUTE STATISTICS;
- If a query is going to sort on the function UPPER(ENAME), an index on the ENAME column itself would not speed up this operation and it might be slow to call the function for each result row
  - A function-based index pre-computes the result of the function for each column value, speeding up queries that use the function for searching or sorting:

    CREATE INDEX emp_upper_ename ON emp_tab(UPPER(ename)) COMPUTE STATISTICS;

Index in SQL: Bitmap

- **create bitmap index** <index-name> **on** <relation-name> (<attribute-list>)
- Example:
  - Student (Student_ID, Name, Address, Age, Gender, Semester)
  - CREATE BITMAP INDEX Idx_Gender ON Student (Gender);
  - CREATE BITMAP INDEX Idx_Semester ON Student (Semester);

STUDENT					
STUDENT_ID	STUDENT_NAME	ADDRESS	AGE	GENDER	SEMESTER
100	Joseph	Alaiedon Township	20	M	1
101	Allen	Fraser Township	21	F	1
102	Chris	Clinton Township	20	F	2
103	Patty	Troy	22	F	4



SEMESTER	
1	1 1 0 0
2	0 0 1 0
3	0 0 0 0
4	0 0 0 1

- SELECT * FROM Student WHERE Gender = 'F' AND Semester = 4;
  - AND 0 1 1 1 with 0 0 0 1 to get the result

Multiple-Key Access

- Use multiple indices for certain types of queries
- Example:

  select *ID*

  from *instructor*

  where *dept_name* = "Finance" **and** *salary* = 80000
- Possible strategies for processing query using indices on single attributes:
  - Use index on *dept_name* to find instructors with department name Finance; test *salary = 80000*
  - Use index on salary to find instructors with a salary of 80000; test *dept_name = "Finance"*
  - Use *dept_name* index to find pointers to all records pertaining to the "Finance" department
    - Similarly use index on *salary*
      - Take intersection of both sets of pointers obtained

Multiple-Key Access: Indices

- **Composite Search Keys** are search keys containing more than one attribute
  - For example, (*dept_name, salary*)
- Lexicographic ordering: $(a_1, a_2) < (b_1, b_2)$ if either

    - $a_1 < b_1$ or
    - $a_1 = b_1 \; and \; a_2 < b_2$
- Hence, the order is important

Multiple-Key Access: Indices on Multiple Attributes

Suppose we have an index on combined search-key:

(dept_name, salary)

- With the **where** clause

  where *dept_name* = "Finance" **and** *salary* = 80000

  the index on (*dept_name, salary*) can be used to fetch only records that satisfy both conditions

  - Using separate indices in less efficient — we may fetch many records (or pointers) that satisfy only one of the conditions
  - Can also efficiently handle

    where *dept_name* = "Finance" **and** *salary* < 80000
  - But cannot efficiently handle

    where *dept_name* $<$ "Finance" **and** *balance* = 80000
    - May fetch many records that satisfy the first but not the second condition

Privileges Required to Create an Index

- When using indexes in an application, you might need to request that the DBA grant privileges or make changes to initialization parameters
- To create a new index
  - You must own, or have the INDEX object privilege for the corresponding table
  - The schema that contains the index must also have a quota for the tablespace intended to contain the index, or the UNLIMITED TABLESPACE system privilege
  - To create an index in another user's schema, you must have the CREATE ANY INDEX system privilege
- Function-based indexes also require the QUERY_REWRITE privilege and that the QUERY_REWRITE_ENABLED initialization parameter to be set to TRUE

Guidelines for Indexing

- Previously we had studied various issues for a proper design of a relational database system
- This focused on:
  - ***Normalization of Tables*** leading to:
    - Reduction of Redundancy to minimize possibilities of Anomaly
    - Easier adherence to constraints (various dependencies)
    - Efficiency of access and update — a better normalized design often gives better performance
- The performance of a database system, however, is also significantly impacted by the way the data is physically organized and managed
  - These are done through:
    - Indexing and Hashing
- While normalization and design are startup time activities that are usually performed once at the beginning (and rarely changed later), the performance behavior continues to evolve as the database is used over time
  - Hence we need to continually
    - ***Collect Statistics*** about data (of various tables) to learn of the patterns
    - ***Adjust the Indexes*** on the tables to optimize performance
- There is no sound theory that determines optimal performance

    - Rather, we take a quick look into a few common guidelines that can help you keep your database agile in its behaviour

Guidelines for Indexing: Ground Rules

- Some guidelines — heuristic and common sense, but time-tested are - are summarized here as a set of Ground Rules for Indexing
    - **Rule 0** → *Indexes lead to Access — Update Tradeoff*
    - **Rule 1** → *Index the Correct Tables*
    - **Rule 2** → *Index the Correct Columns*
    - **Rule 3** → *Limit the Number of Indexes for Each Table*
    - **Rule 4** → *Choose the Order of Columns in Composite Indexes*
    - **Rule 5** → *Gather Statistics to Make Index Usage More Accurate*
    - **Rule 6** → *Drop Indexes That Are No Longer Required*

Guidelines for Indexing: Rule 0

- **Rule 0** → *Indexes lead to access — Update tradeoff*
    - Every query (access) results in a 'search' on the underlying physical data structures
        - Having specific index on search field can significantly improve performance
    - Every update (insert/delete/values update) results in update of the index files — an overhead or penalty for quicker access
        - Having unnecessary indexes can cause significant degradation or performance of various operations
        - Index files may also occupy significant space on your disk and/or
        - Cause slow behavior due to memory limitations during index computations
    - Use informed judgment to index

Guidelines for Indexing: Rule 1

- **Rule 1** → *Index the correct tables*
    - Create an index if you frequently want to **retrieve less than 15%** of the rows in a large table
        - The percentage varies greatly according to the relative speed of a table scan and how clustered the row data is about the index key
            - The faster the table scan, the lower the percentage
            - More clustered the row data, the higher the percentage
- Index columns used for joins to improve performance on **joins of multiple tables**
- Primary and unique keys automatically have indexes, but you might want to create an **index on a foreign key**
- **Small tables** do not require indexes
    - If a query is taking too long, then the table might have grown from small to large

Guidelines for Indexing: Rule 2

- **Rule 2** → *Index the correct columns*
    - Columns with the following characteristics are candidates for indexing:
        - Values are relatively unique in the column
        - There is a wide range of values (good for regular indexes)
        - There is a small range of values (good for bitmap indexes)
        - The column contains many nulls, but queries often select all rows having a value
            - In this case, a comparison that matches all the non-null values, such as:
                - WHERE COL_X > -9.99 *power(10, 125) is preferable to WHERE COL_X IS NOT NULL
                - This is because the first uses an index on COL_X (if COL_X is a numeric column)

- Columns with the following characteristics are less suitable for indexing:
  - There are many nulls in the column and you do not search on the non-null values
  - LONG and LONG RAW columns cannot be indexed
- The size of single index entry cannot exceed roughly one-half (minus some overhead) of the available space in the data block

Guidelines for Indexing: Rule 3

- **Rule 3** → *Limit the number of indexes for each table*
  - The more indexes, the more overhead is incurred as the table is altered
    - When rows are inserted or deleted, all indexes on the table must be updated
    - When a column is updated, all indexes on the column must be updated
  - You must weigh the performance benefit of indexes for queries against the performance overhead of the updates
    - If a table is primarily read-only, you might use more indexes, but, if a table is heavily updated, you might use fewer indexes

Guidelines for Indexing: Rule 4

- **Rule 4** → *Choose the order of columns in composite indexes*
  - The order of columns in the CREATE INDEX statement can affect performance
    - Put the column used most often first in the index
    - You can create a composite index (using several columns) and the same index can be used for queries that reference all of these columns, or just some of them

Table VENDOR_PARTS

VEND ID	PART NO	UNIT COST
1012	10–440	.25
1012	10–441	.39
1012	457	4.95
1010	10–440	.27
1010	457	5.10
1220	08–300	1.33
1012	08–300	1.19
1292	457	5.28

  - For the VENDOR_PARTS table, assume that there are 5 vendors and each vendor has about 1,000 parts
    - Suppose, VENDOR_PARTS is commonly queries as:
      - `SELECT * FROM vendor_parts WHERE part_no = 457 AND vendor_id = 1012;`
      - Create a composite index with the most selective (with most values) column first
        - `CREATE INDEX ind_vendor_id ON vendor_parts (part_no, vendor_id);`
  - Composite indexes speed up queries that use the leading portion of the index:
    - So, queries with WHERE clauses using only PART_NO column also runs faster
    - With only 5 distinct values, a separate index on VENDOR_ID does not help

Guidelines for Indexing: Rule 5

- **Rule 5** → *Gather statistics to make index usage more accurate*
  - The database can use indexes more effectively when it has statistical information about the tables involved in the queries

        - Gather statistics when the indexes are created by including the keywords COMPUTE STATISTICS in the CREATE INDEX statement
        - As data is updated and the distribution of values changes, periodically refresh the statistic by calling procedures like (in Oracle):
            - `DBMS_STATS.GATHER_TABLE_STATISTICS`
            - `DBMS_STATS.GATHER_SCHEMA_STATISTICS`

Guidelines for Indexing: Rule 6

- **Rule 6** → *Drop indexes that are no longer required*
    - You might drop an index if:
        - It does not speed up queries
            - The table might be very small, or there might be many rows in the table but very few index entries
        - The queries in your applications do not use the index
        - The index must be dropped before being rebuilt
    - When you drop an index, all extents of the index's segment are returned to the containing tablespace and become available for other objects in the tablespace
    - Use the SQL command DROP INDEX to drop an index
        - For example, the following statement drops a specific named index:
            - `DROP INDEX Emp_ename;`
    - If you drop a table, then all associated indexes are dropped too
    - To drop an index, the index must be contained in your schema or you must have the `DROP ANY INDEX` system privilege