Week 9

Database Management Systems

Concepts of Indexing

- 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 (rec #)
- 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

- Indexing mechanisms used to speed up access to desired data.
  - For example:
    - ▷ Name in a faculty table
    - ▷ author catalog in 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

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*

Index Evaluation Metrics

- 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 **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 **non-clustering index**
- **Index-sequential file**: ordered sequential file with a primary index

Dense Indexing

- **Dense index** — Index record appears for every search-key value in the file.
- For example, 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	

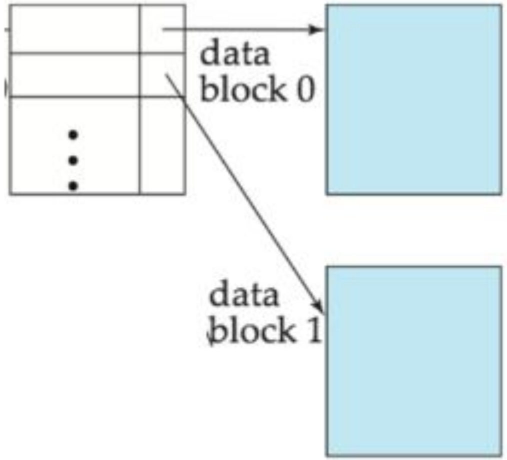
Sparse Indexing

- **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

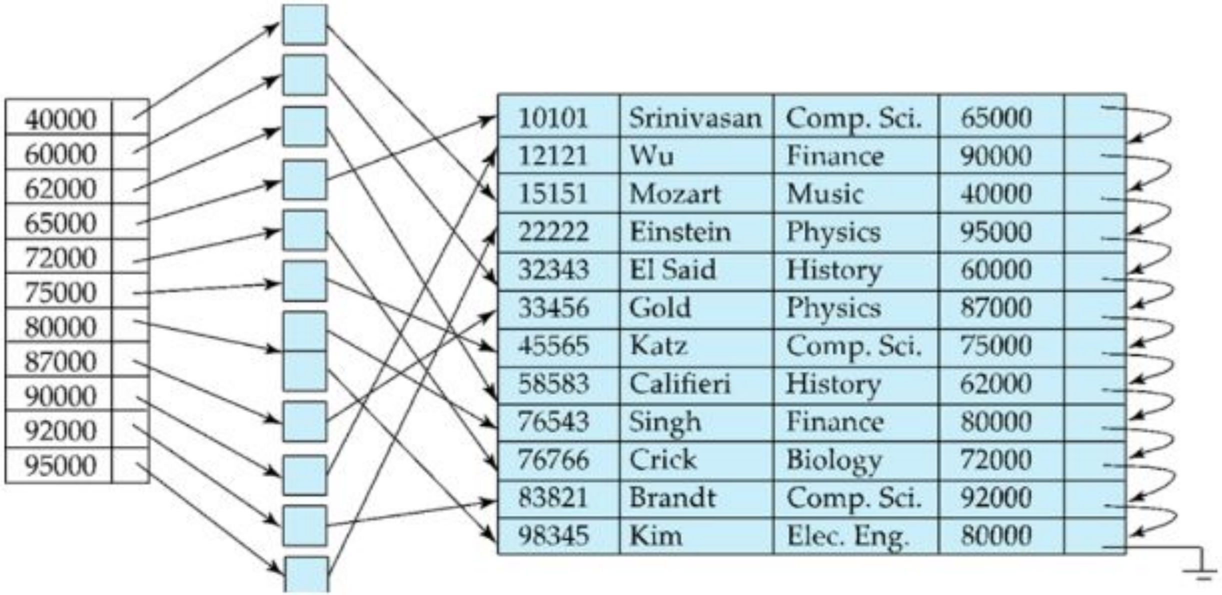
10101	
32343	
76766	

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

- 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



Secondary index on *salary* field of *instructor*

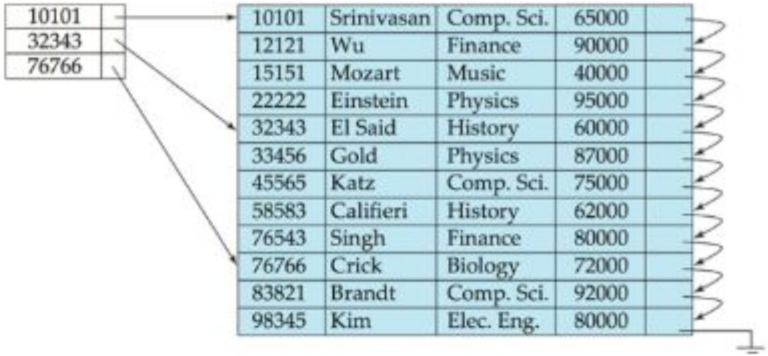
- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value.
- Secondary indices have to be dense

Multilevel Indexing

- 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 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.



10101	
32343	
76766	

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

- **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

Index Update: Insertion

- **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

Balanced Binary Search Trees:

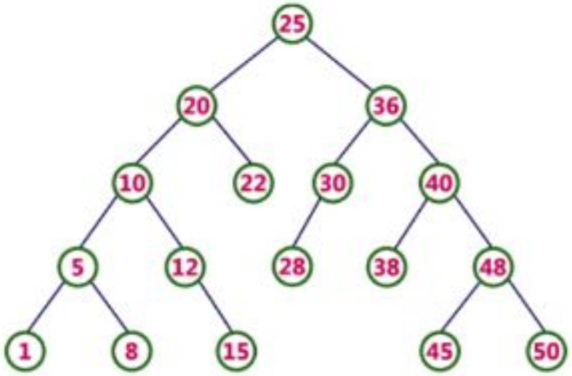
- How to search a key in a list of n data items?
  - Linear Search: $O(n)$: Find 28 $\Rightarrow$ 16 comparisons
    - ▷ Unordered items in an array – search sequentially
    - ▷ Unordered / Ordered items in a list – search sequentially

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

  - Binary Search: $O(\lg n)$: Find 28 $\Rightarrow$ 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)	

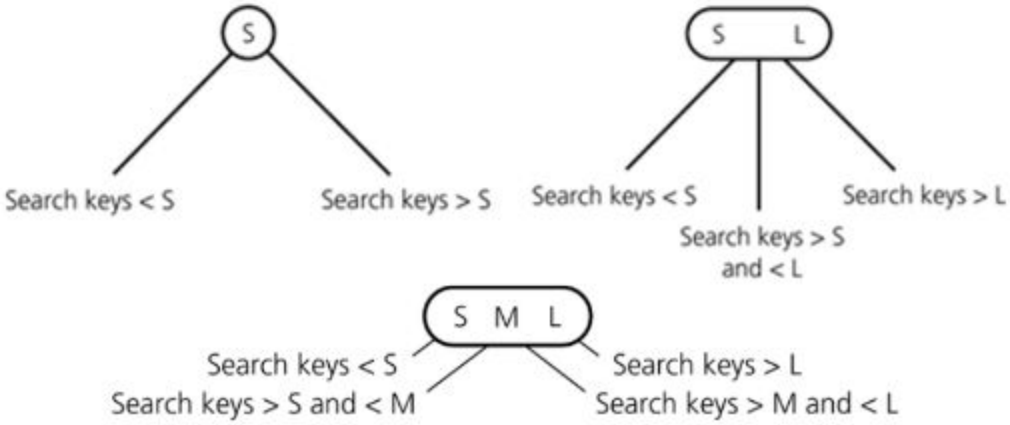
- Between an array and a list, there is a trade-off between search and insert/delete complexity
- For a BST of n nodes, $\lg n \leq h < n$, where h is the height of the tree
- A BST is balanced if $h \sim O(\lg n)$: this what we desire

- In the worst case, searching a key in a BST is $O(h)$, where h is the height of the key
- **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:** $h \sim O(\lg n)$
  - 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 $\lg n$
  - Time complexity of search in BST becomes $O(\lg n)$

- These data structures have optimal complexity for the required operations:
  - Search: $O(\lg n)$
  - Insert: Search + $O(1)$: $O(\lg n)$
  - Delete: Search + $O(1)$: $O(\lg n)$
- 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

- All leaves are at the same depth (the bottom level).
  - Height, h, of all leaf nodes are same
    - $h \sim O(\lg n)$
    - Complexity of search, insert and delete: $O(h) \sim O(\lg n)$
- 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

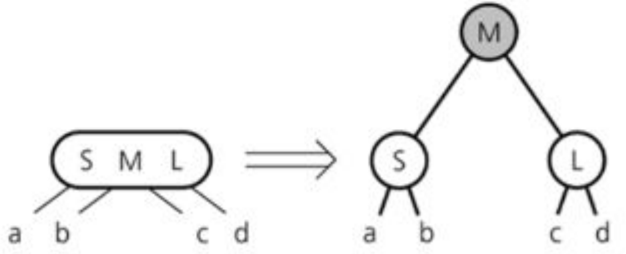
- Uses 3 kinds of nodes satisfying key relationships as shown below:
  - A 2-node must contain a single data item (S) and two links
  - A 3-node must contain two data items (S, L) and three links
  - A 4-node must contain three data items (S, M, L) and four links
  - A leaf may contain either one, two, or three data items



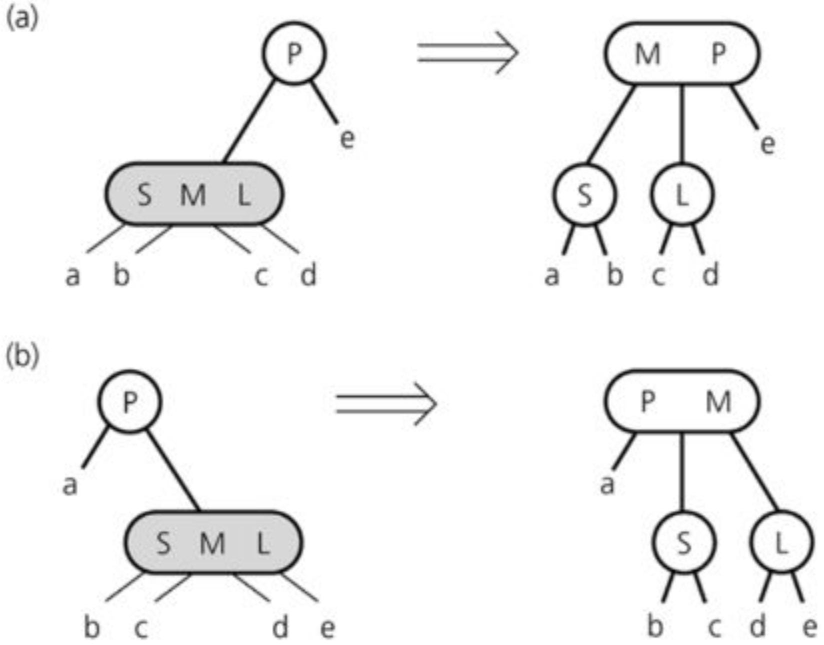
- Insert
  - Search to find expected location
    - ▷ If it is a 2 node, change to 3 node and insert
    - ▷ If it is a 3 node, change to 4 node and insert
    - ▷ If it is a 4 node, split the node by moving the middle item to parent node, then insert
  - Node Splitting
    - ▷ A 4-node is split as soon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

Insertion in the Tree

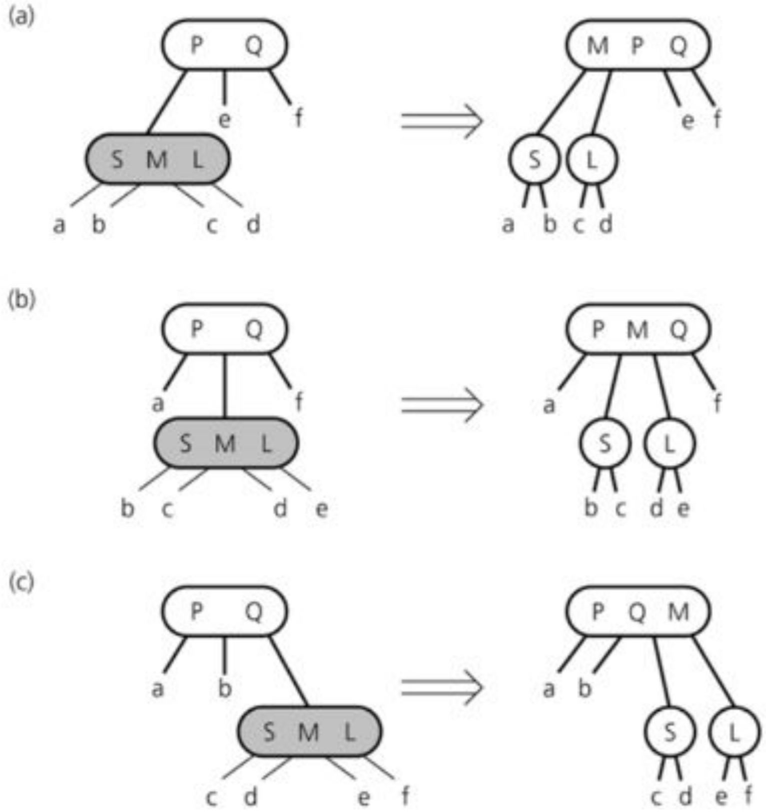
- Splitting at Root



- Splitting with 2 Node parent

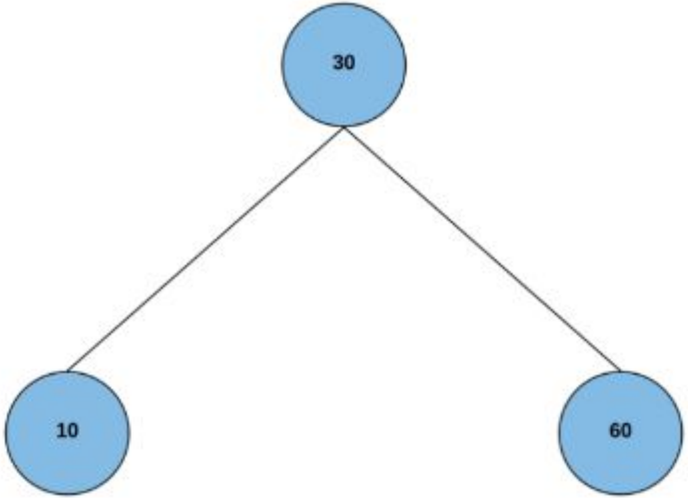


- Splitting with 3 Node parent

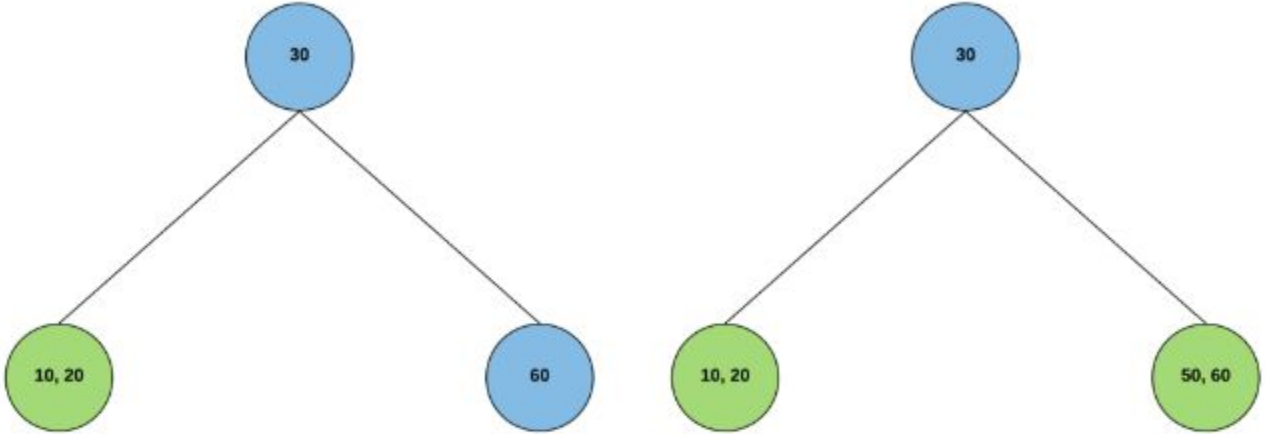


- Node Splitting: There are two strategies:
  - *Early*: Split a 4-node as soon as you cross on in traversal. It ensures that the tree does not have a path with multiple 4-nodes at any point
  - *Late*: Split a 4-node only when you need to insert an item in it. This might lead to cases where for one 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

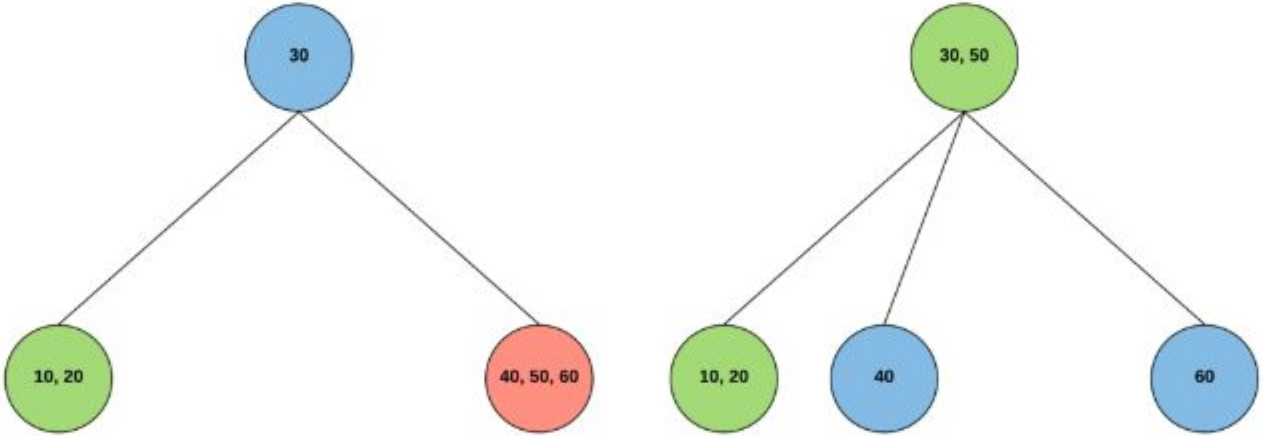
- Insert 10, 30, 60, 20, 50, 40, 70, 80, 15, 90, 100
- 10
- 10, 30
- 10, 30, 60
- Split for 20



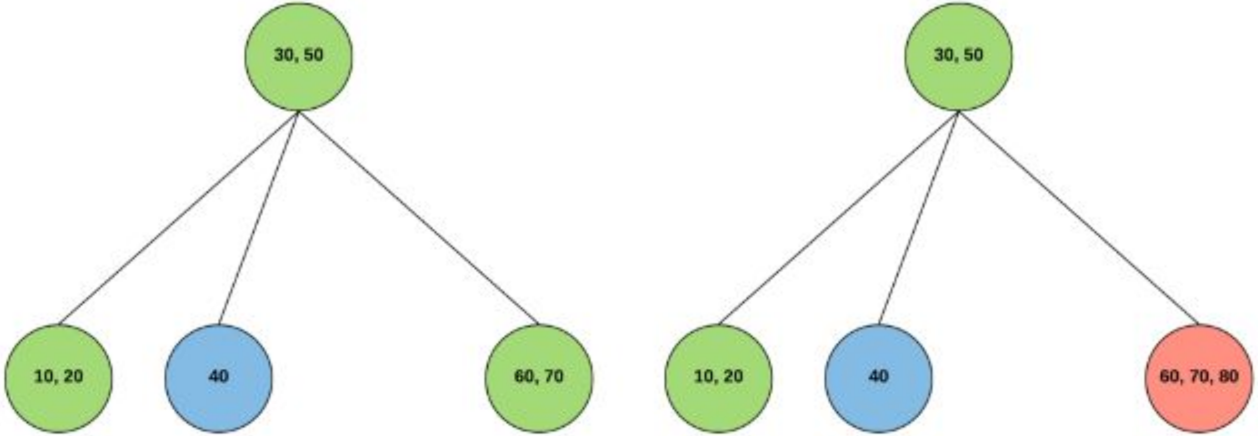
- 10, 30, 60, 20
- 10, 30, 60, 20, 50



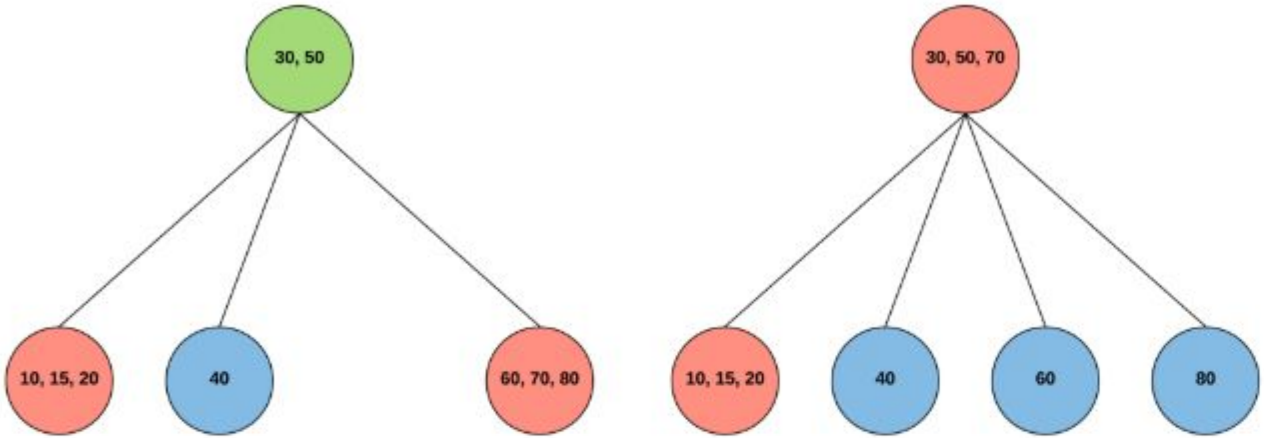
- 10, 30, 60, 20, 50, 40
- Split for 70



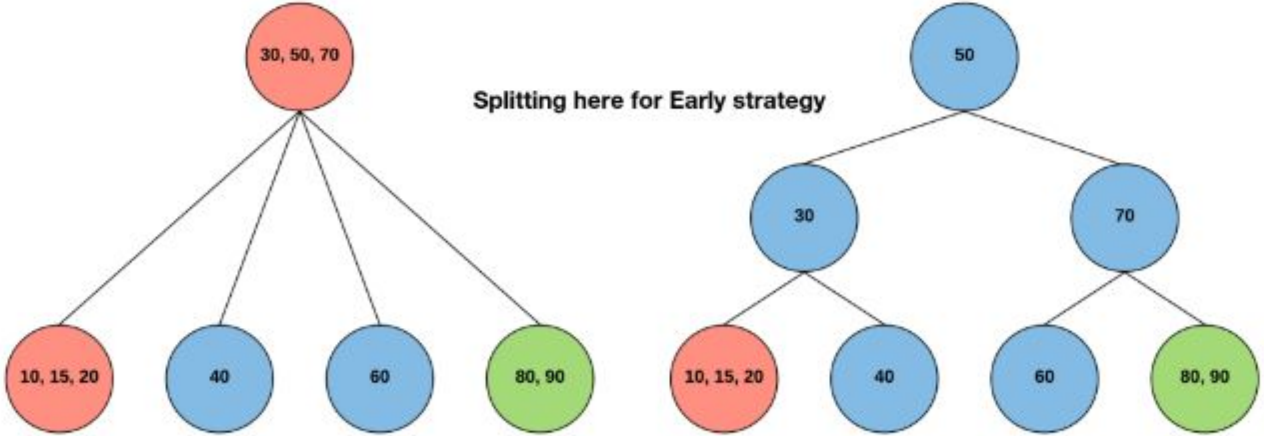
- 10, 30, 60, 20, 50, 40, 70
- 10, 30, 60, 20, 50, 40, 70, 80



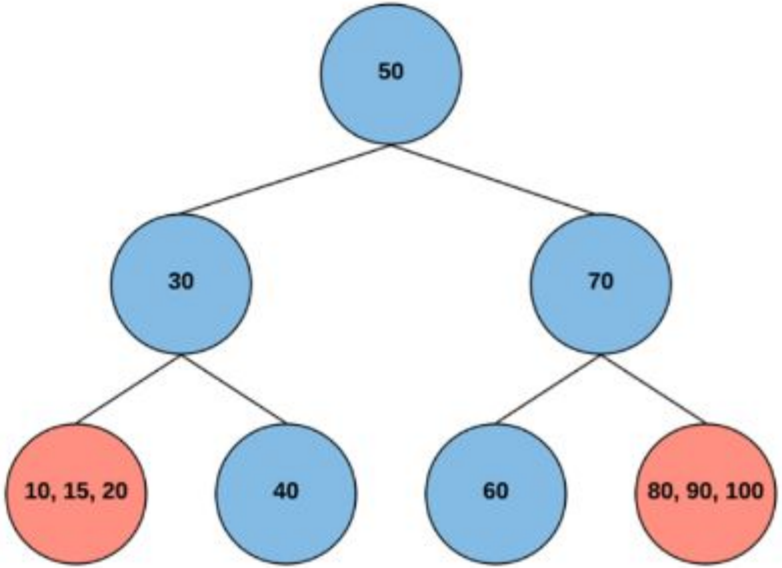
- 10, 30, 60, 20, 50, 40, 70, 80, 15
- Split for 90



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



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



- 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 a 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

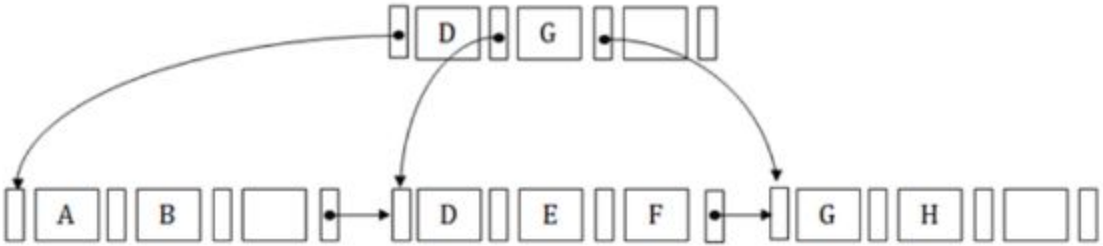
- Advantages
    - All leaves are at the same depth (the bottom level): Height, $h \sim O(\lg n)$
    - Complexity of search, insert and delete: $O(h) \sim O(\lg n)$
    - All data is kept in sorted order
    - Generalizes easily to larger nodes
    - Extends to external data structures
- Disadvantages
    - Uses variety of node types – need to destruct and construct multiple nodes for converting a 2 Node to 3 Node, a 3 Node to 4 Node, for splitting etc.

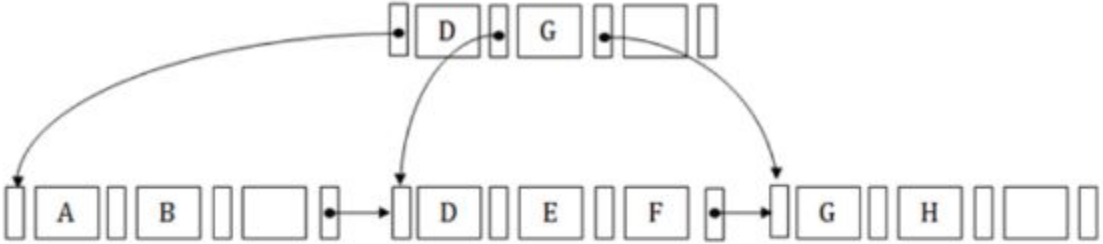
- Consider only one node type with space for 3 items and 4 links
  - Internal node (non-root) has 2 to 4 children (links)
  - Leaf node has 1 to 3 items
  - Wastes some space, but has several advantages for external data structure
- Generalizes easily to larger nodes
  - All paths from root to leaf are 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 $\left\lceil \frac{(n-1)}{2} \right\rceil$ 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 where $n = 4$

B+ Trees

The B^+ Tree

- Is a *balanced binary search tree*
    - Follows a *multi-level index* format like 2-3-4 Tree
- Has the *leaf nodes denoting actual data pointers*
- Ensures that all *leaf nodes remain at the same height* (like 2-3-4 Tree)
- Has the *leaf nodes are linked using a link list*
    - Can support *random access as well as sequential access*
- Example:

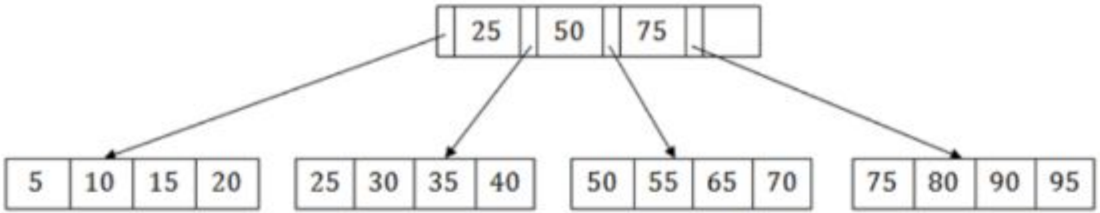




- Internal node contains
  - At least $\frac{n}{2}$ child pointers, except the root node
  - At most n pointers
- Leaf node contains
  - At least $\frac{n}{2}$ record pointers and $\frac{n}{2}$ key values
  - At most n record pointer and n key values
  - One block pointer P to point to next leaf node

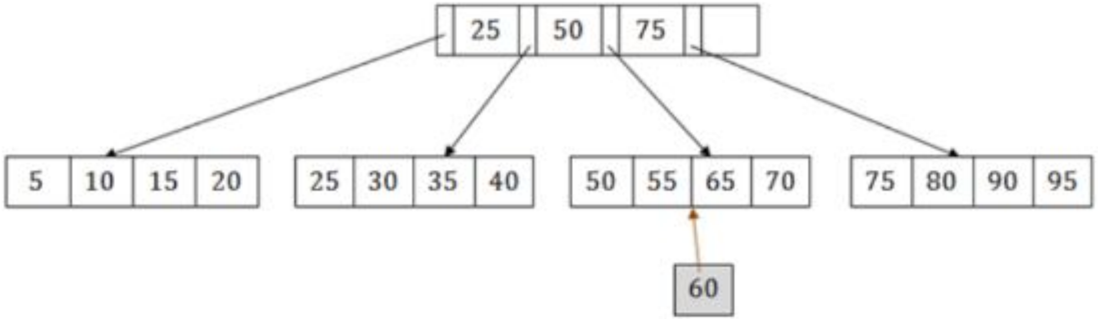
Note: These are approximate values, we will discuss more precise values later in this lecture.

- 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 tha 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

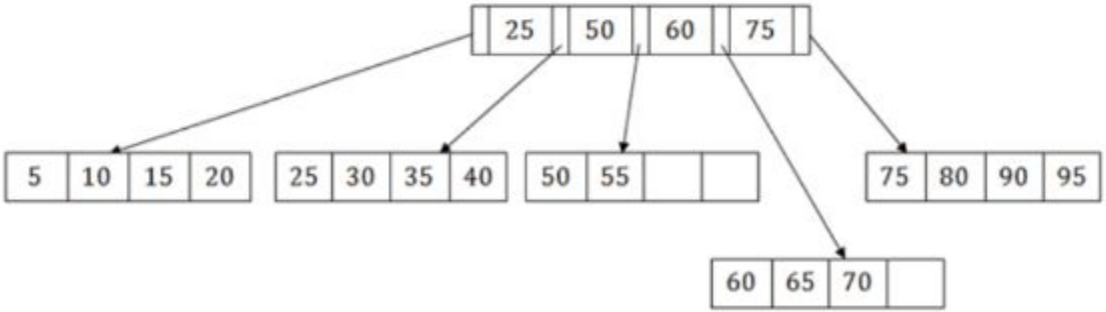


Source: B^+ *Tree*

B+ Tree Insertion

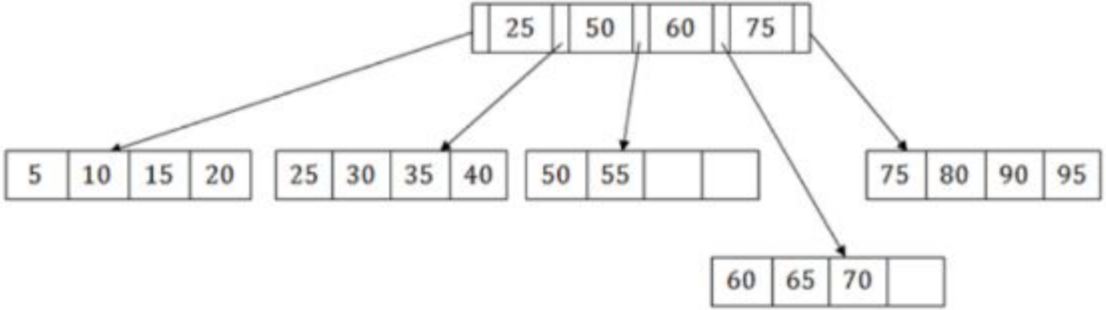


- Suppose we want to insert a record 60 that goes to 3^{rd} leaf node after 55
- The leaf node of this tree is already full, so we cannot insert 60 there
- So we have to split the leaf node, so that it can be inserted into tree without affecting the fill factor, balance and order
- The 3^{rd} leaf node 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

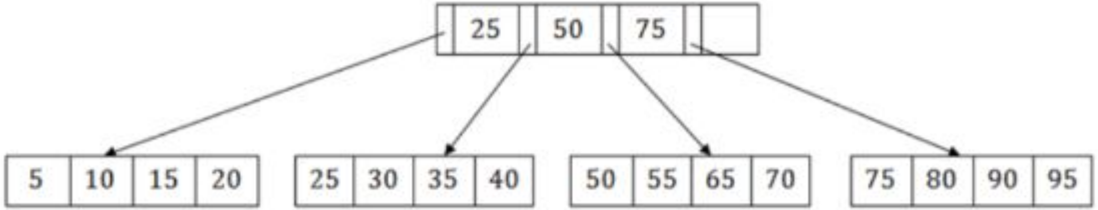


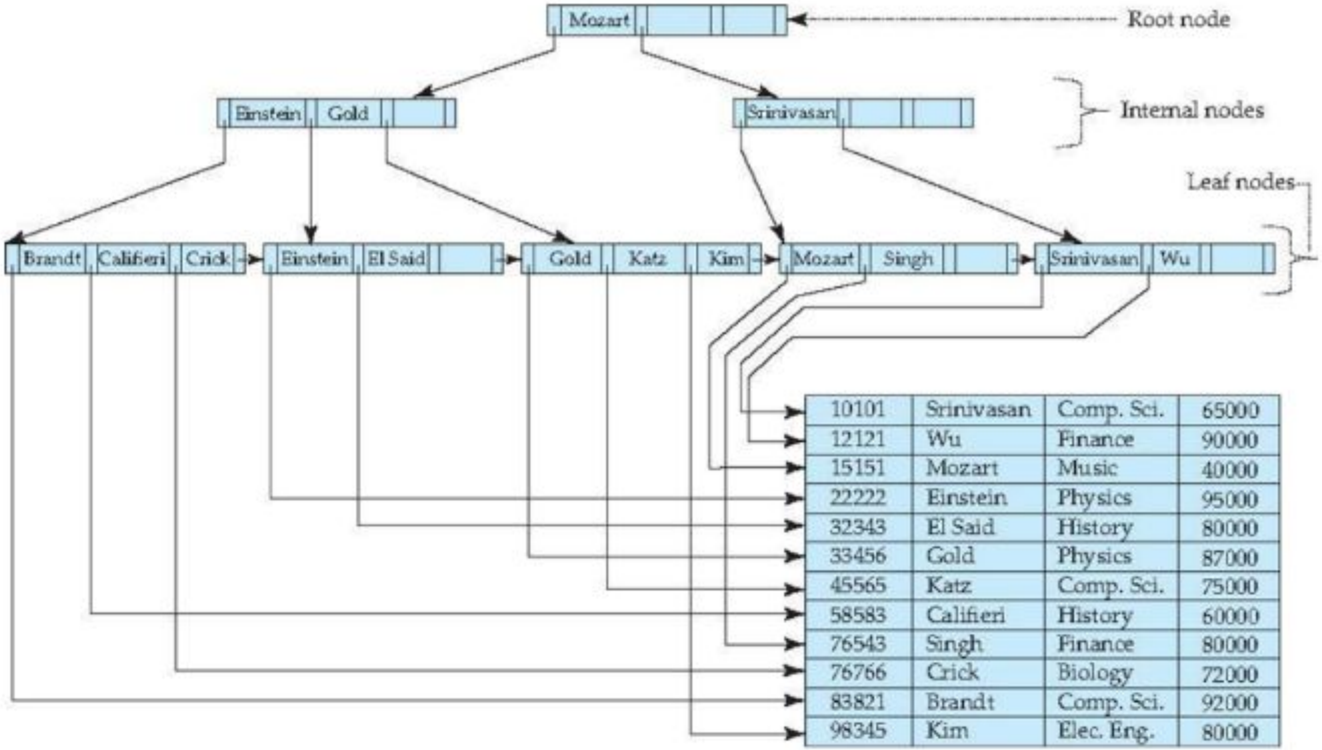
- So we can group (50, 55) and (60, 65, 70) into 2 leaf nodes
- If these two has to be leaf nodes, the intermediate node cannot branch from 50
- It should have 60 added to it, and then we can have pointers to a new leaf node

B+ Tree Deletion



- 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 re-arranging the nodes:





Root node
Internal nodes
Leaf nodes
Mozart
Einstein
Gold
Srinivasan
Brandt
Califieri
Crick
Einstein
El Said
Gold
Katz
Kim
Mozart
Singh
Srinivasan
Wu
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

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 $\lceil \frac{n}{2} \rceil$ and n children
- A leaf node has between an $\lceil \frac{n-1}{2} \rceil$ 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.

- 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 $2 * \lceil \frac{n}{2} \rceil$ values
  - 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 $\lceil log_{\lceil n/2 \rceil}(K) \rceil$
  - 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 Trees:

- 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

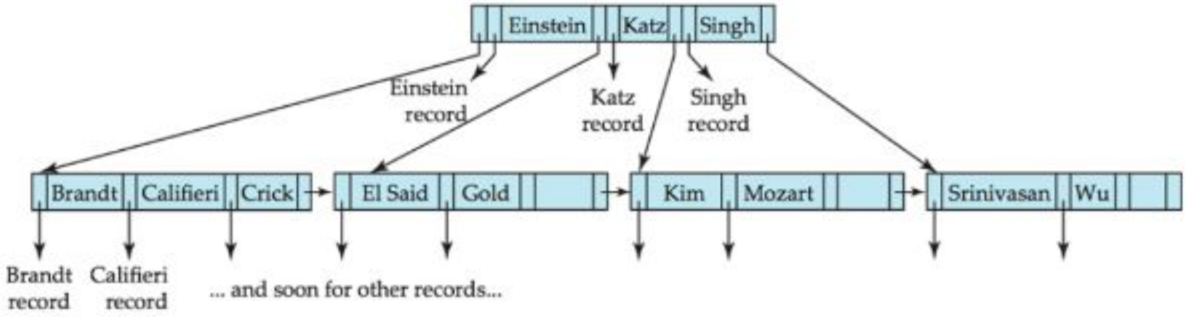
P_1	K_1	P_2	...	P_{n-1}	K_{n-1}	P_n

(a)

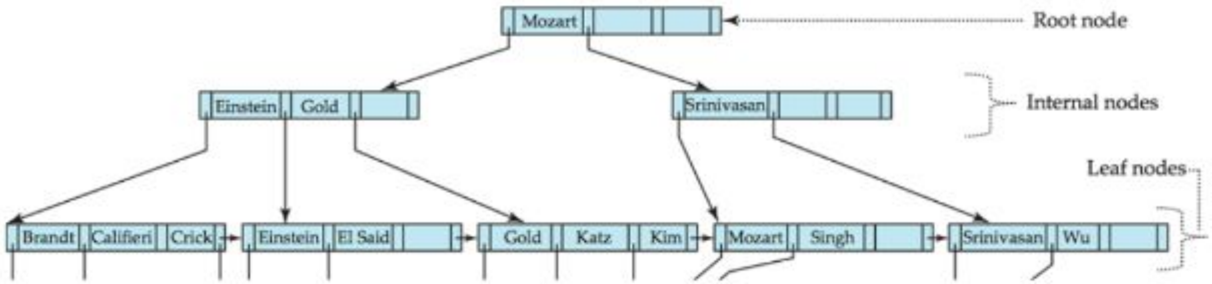
P_1	B_1	K_1	P_2	B_2	K_2	...	P_{m-1}	B_{m-1}	K_{m-1}	P_m

(b)

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



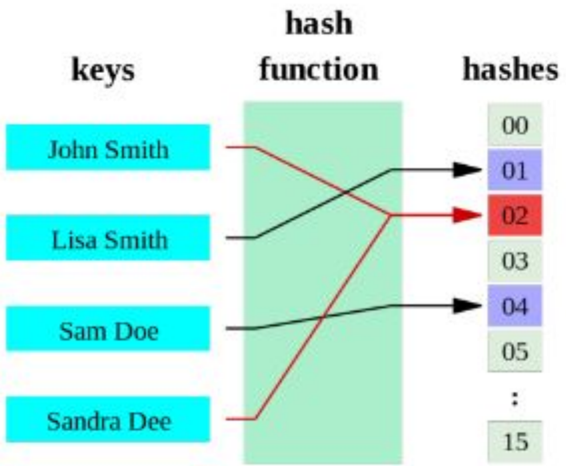
B-tree (above) and B^+ tree (below) on same data



Hashing:

Static Hashing

- A hash function h maps data of arbitrary size (from domain D) to fixed-size values (say, integers from 0 to $N > 0$ $h : D \rightarrow [0..N]$
- 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 occured
- A hash function should be *Collision Free* and *Fast*



- 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

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 the integer i
- The hash function returns the sum of the binary representations of the characters modulo 10
  - For example

$$\begin{array}{ll} h(\text{Music}) = 1 & h(\text{History}) = 2 \\ h(\text{Physics}) = 3 & h(\text{Elec. Eng.}) = 3 \end{array}$$

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

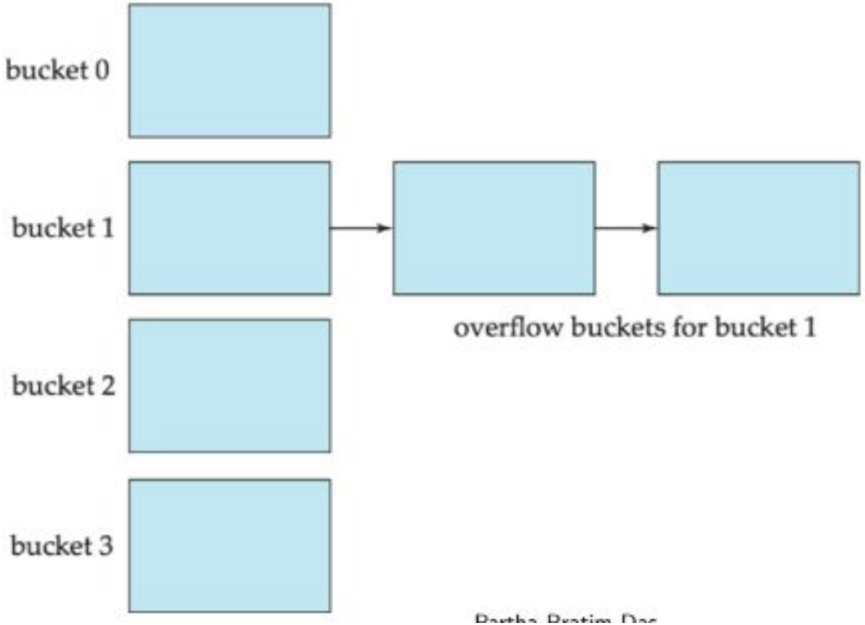
bucket 7

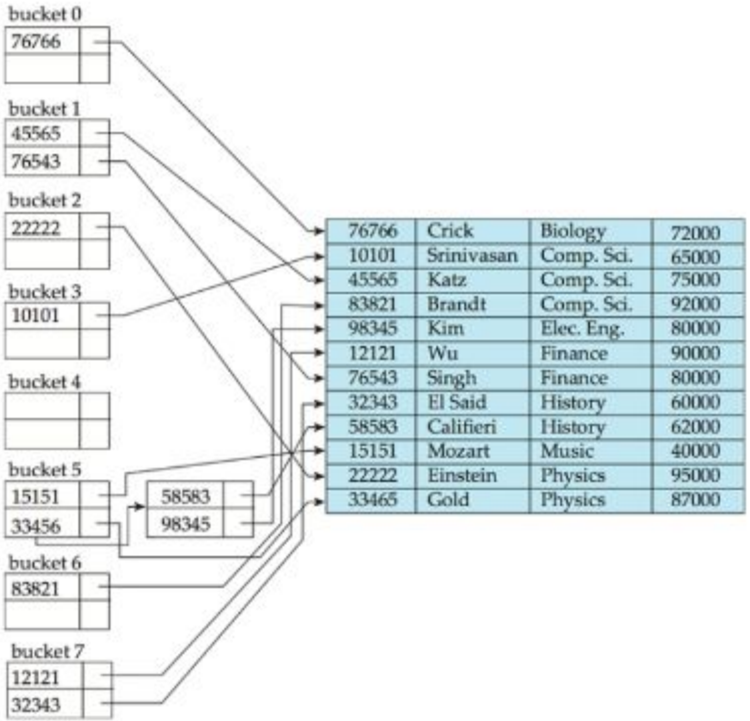
Hash file organization of instructor file, using *dept_name* as key

- 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**, i.e., each bucket is assigned the same number of search-key values from the set of *all* possible values
- Ideal hash function 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

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

- **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





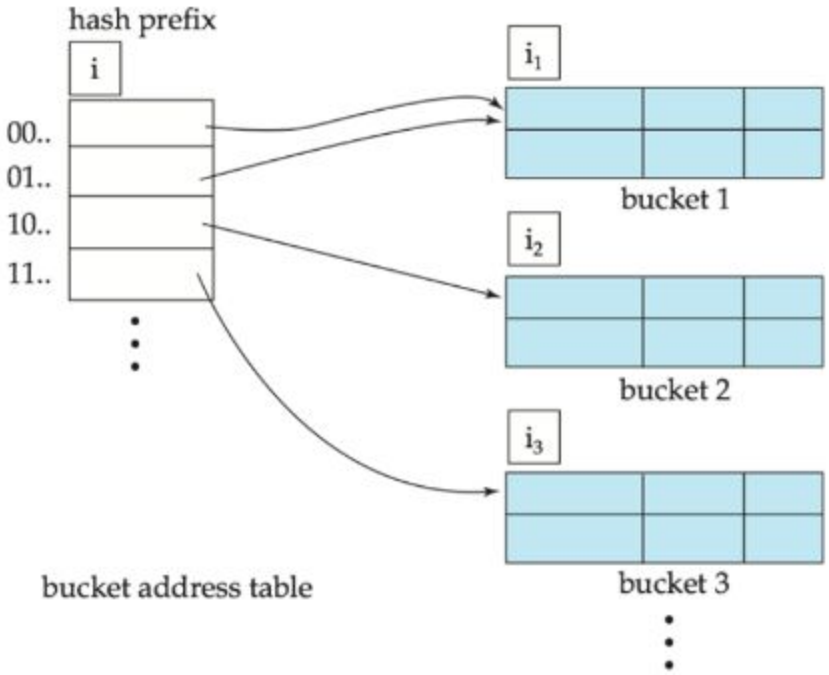
- Hash index on *instructor*, on attribute *ID*
- Computed by adding the digits modulo 8

Deficiencies of Static Hashing

- In static hashing, function *h* maps search-key values to a fixed set of *B* of bucket addresses. Databases grow or shrink with time
  - If initial number of buckets is too small, and file grows, performance will degrade due to too much overflows
  - If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be 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


hash prefix
i
00..
01..
10..
11..
bucket address table
i_1
bucket 1
i_2
bucket 2
i_3
bucket 3

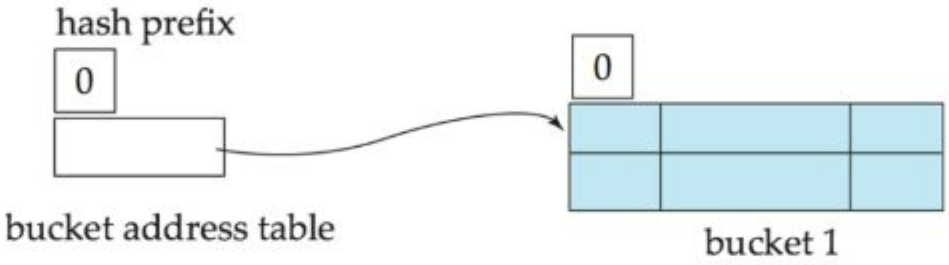
- Each bucket j stores a value i_j
  - All the entries that point to the same bucket have the same values on the first i_j bits
- To locate the bucket containing search-key K_j
  - Compute $h(K_j) = X$
  - Use the first i high order bits of X as a displacement into bucket address table, and follow the 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 (next slide)
    - Overflow buckets used instead in some cases (will see shortly)

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

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

- Initial Hash structure; bucket size = 2

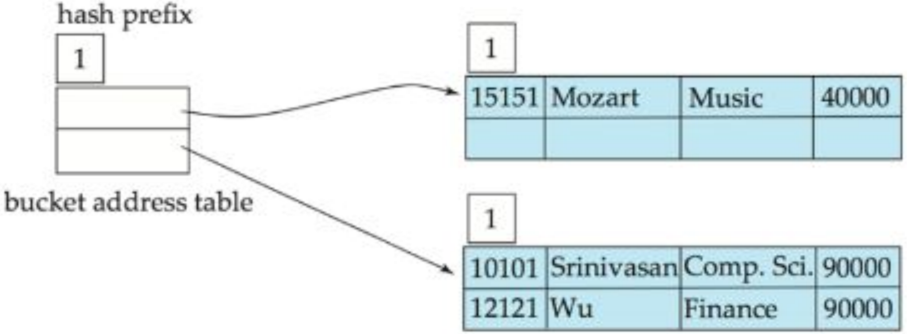


- Insert "Mozart", "Srinivasan", and "Wu" records

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

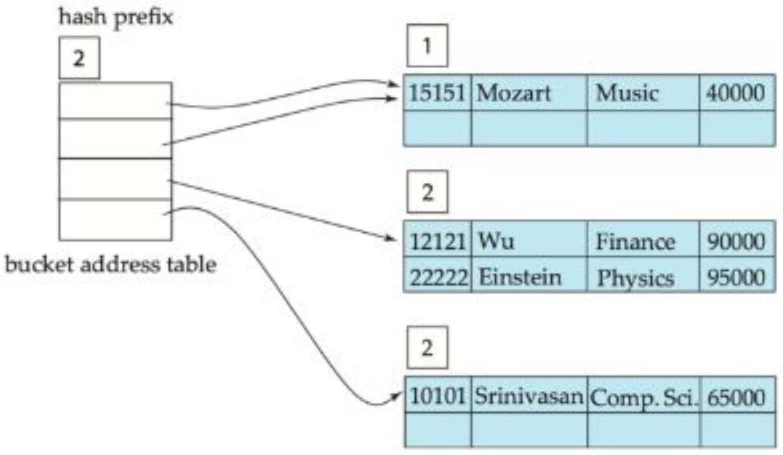
dept_name	h(*dept_name*)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100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



- Insert Einstein 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

- Hash structure after insertion of "Einstein" record

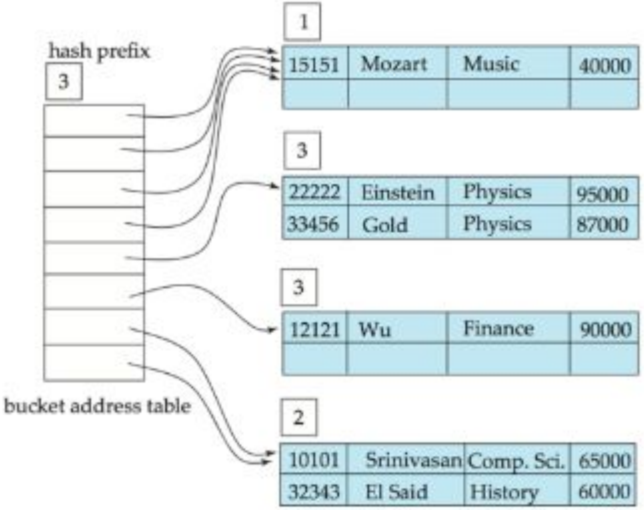


dept_name	h(*dept_name*)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100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

- Insert "Gold" and "El Said" 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 "Gold" and "El Said" records

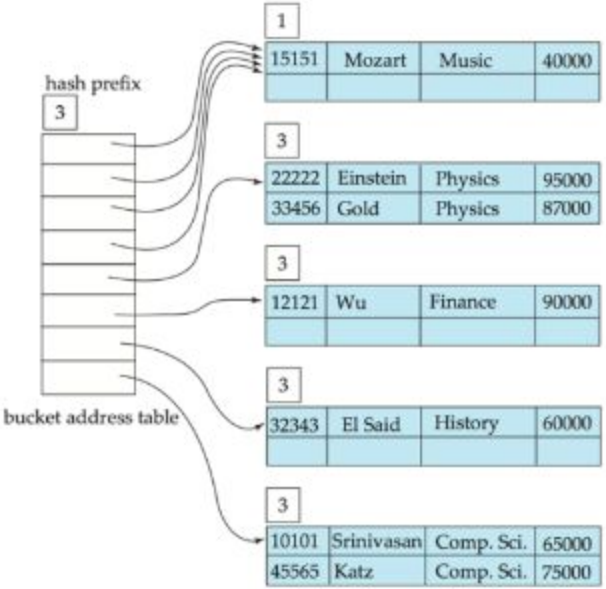


- Insert Katz record

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

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 "Katz" record

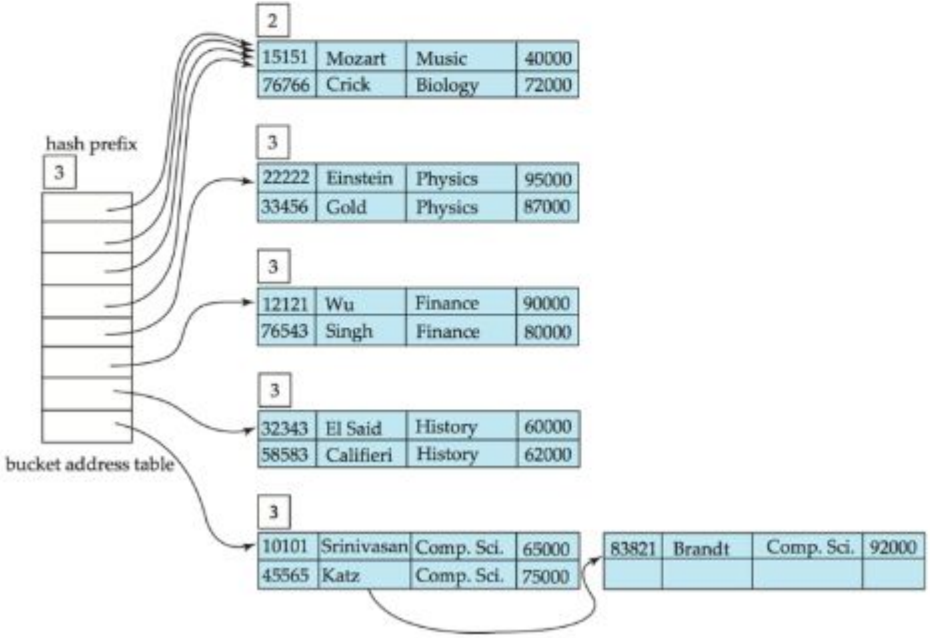


dept_name	h(*dept_name*)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100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

- Insert "Singh", "Califieri", "Crick", "Brandt" 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 "Singh", "Califieri", "Crick", "Brandt" records

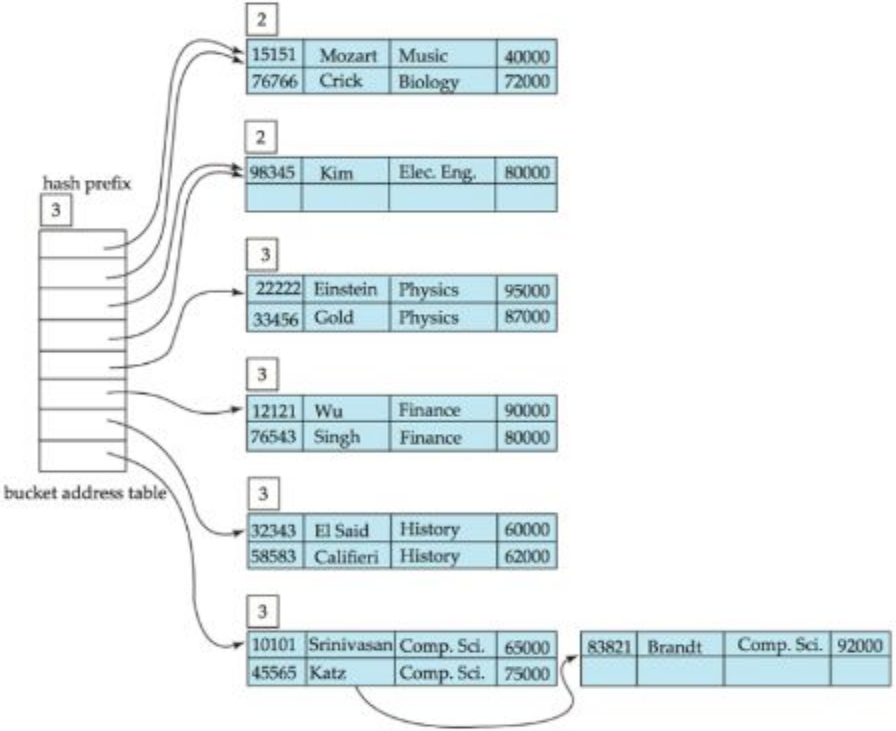


dept_name	h(*dept_name*)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100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

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

- Insert Kim record

- Hash structure after insertion of “Kim” record



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

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

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 a 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