Analysis Table

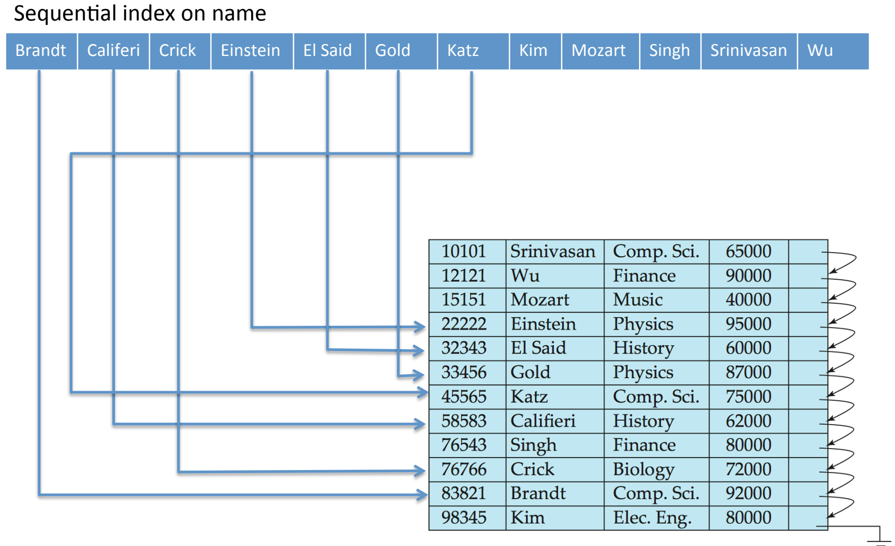
	heap	sequential	seq + primary index	secondary index
scan	$O(B)$	$O(B)$	$O(B_I)$ index scan	$O(B_I)$ index scan
= search	$O(B)$	$O(\log_2 B)$	$O(\log_2 B_I)$	$O(\log_2 B_I + m)$
<> search	$O(B)$	$O(\log_2 B + m)$	$O(\log_2 B_I + m)$	$O(\log_2 B_I + m)$
insert	$O(1)$	$O(B)$	$O(B_I)$	$O(B_I)$
delete	$O(B)$	$O(\log_2 B)$	$O(\log_2 B_I)$	$O(\log_2 B_I + m)$

Note:

- This table reflects the time it takes to resort a file or index after insertion:
  - $O(B)$ for a sequential file, $O(B_I)$ for an index
- Deletions do not necessarily require resorting

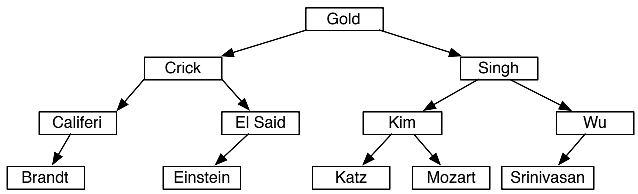
Binary Search

Sequential index on name



Binary Tree Search

- Binary search algorithm induces an ordered binary tree

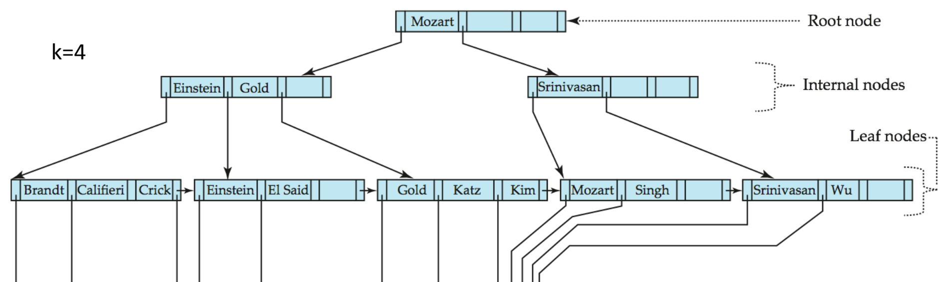


- What if we had a binary tree index instead of a sequential array index?

B+ Tree Index
(with a quick review of tree structures)

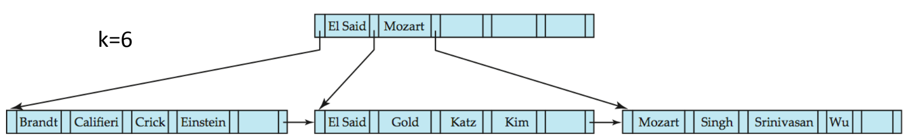
- An *ordered tree* is a tree in which children to the left of a parent have key values less than the parent; children to the right have key values greater
- A *balanced tree* is a tree in which, for any given parent node, the left subtree and right subtree are roughly the same height
  - Search, insert, delete in a balanced ordered tree is $O(\log_2 n)$
- A *B-tree* is a balanced, ordered tree in which each node can have up to k children and k-1 key values
  - A B-tree with k=2 is a balanced binary tree
  - Search, insert, delete in a B-tree is $O(\log_F n)$, where F is the "fanout"
- A *B+ tree* is a B tree with key duplication
  - every key value in the index is stored in a leaf node
    - which means key values may be duplicated in internal nodes
- A **B+ tree index** is a multi-layer index
  - leaf nodes contain full index in sorted order
  - internal nodes are "indexes into indexes"
  - leaf nodes also store pointers to location of full records on disk

Secondary B+ Tree Index on Name



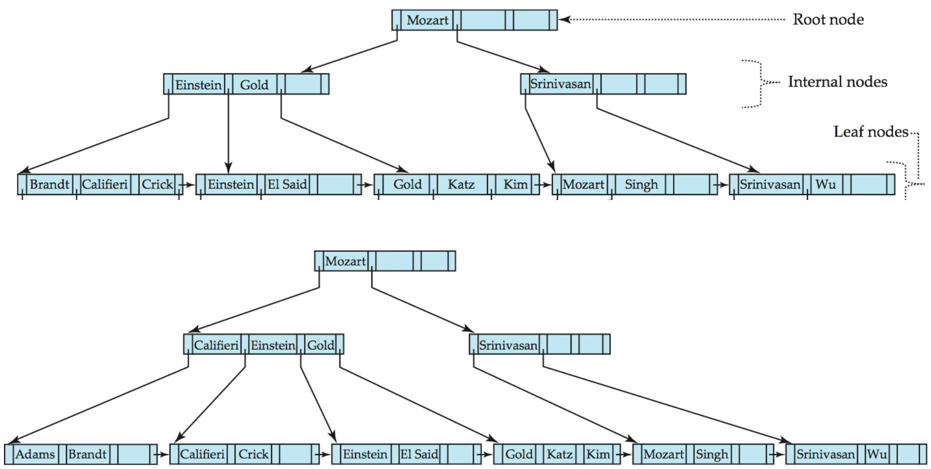
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000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

Secondary B+ Tree Index on Name



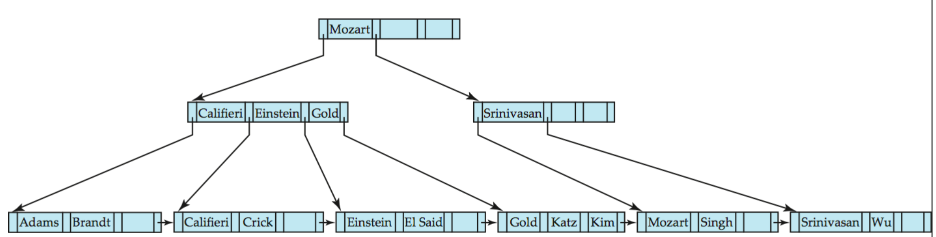
- Value of k influences height of tree
  - which in turn influences lookup/insert/delete time
  - … and space overhead
- Index height = O($\log_{k/2}$ n)
  - n = total number of keys
  - Total blocks for storage in worst case = 1 (root node) + k/2 (first layer) + $k^2/4$ (second layer) + … + B_l (leaf layer)
  - which works out to about O(B_l) extra blocks, or double the size of non-tree
- In real DBMS, k is very large (on the order of 10^3 or larger)
  - Results in a short, wide tree
  - Actual storage overhead much less than double size

B+ Tree Insert

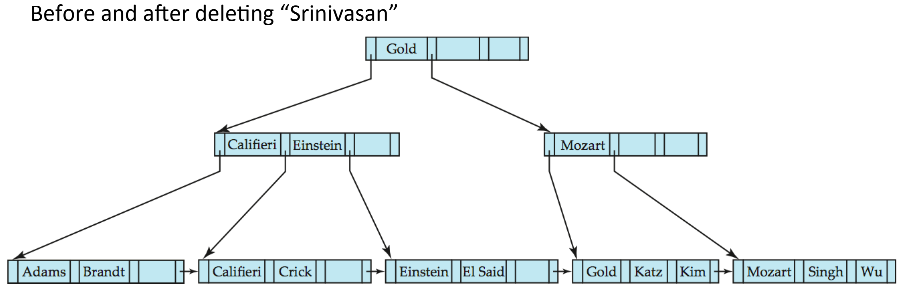


B$^+$-Tree before and after insertion of "Adams"

B+ Tree Delete

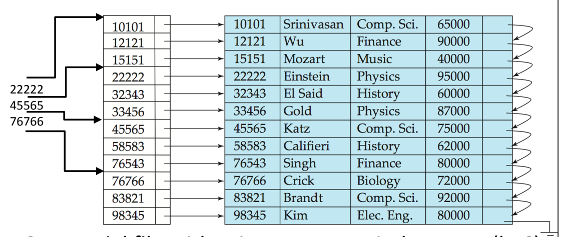


Before and after deleting "Srinivasan"



Sequential File With Primary B+ Tree Index

- Assume search key is primary key, so there is one index record per actual record
- Assume index records are 10% size of actual records
- Space costs:
  - Overhead?
- Time costs:
  - Scan?
  - Search with equality?
  - Search for range?
  - Insert?
  - Delete?



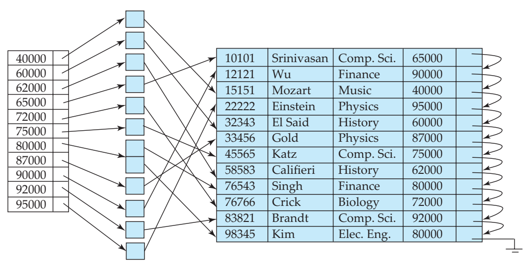
Sequential file with primary B+ tree index on ID (k=6)

Sequential File With Primary B+ Tree Index

- Assume search key is primary key, so there is one index record per actual record
- Assume index records are 10% size of actual records
- Space costs:
  - Overhead? $B_I + B_I/(k/2)$
    - B_I typically 0.15B, as blocks in B+ tree index are rarely 100% full
- Time costs:
  - Scan? B(D+RC) for full table scan, $B_I(D+R_IC)$ for index scan
  - Search with equality? $O(D \log_{k/2} B_I) + D$
  - Search for range? $O(D \log_{k/2} B_I) + mD$
  - Insert? $O(D \log_{k/2} B_I) + 4D$
  - Delete? $O(D \log_{k/2} B_I) + 4D$

Secondary B+ Tree Index

- File organization is a heap relative to the field being indexed
- Assume index records are 10% size of actual records
- Space costs:
  - Overhead?
- Time costs:
  - Scan?
  - Search with equality?
  - Search for range?
  - Insert?
  - Delete?



Imagine there is a B+ tree index on salary

Analysis Table

	heap	sequential	seq + primary index	secondary index	primary B+-tree index	secondary B+-tree index
scan	O(B)	O(B)	$O(B_I)$ index scan	$O(B_I)$ index scan	$O(B_I)$ index scan	$O(B_I)$ index scan
= search	O(B)	$O(\log_2 B)$	$O(\log_2 B_I)$	$O(\log_2 B_I + m)$	$O(\log_F B_I)$	$O(\log_F B_I + m)$
<> search	O(B)	$O(\log_2 B + m)$	$O(\log_2 B_I + m)$	$O(\log_2 B_I + m)$	$O(\log_F B_I + m)$	$O(\log_F B_I + m)$
insert	O(1)	O(B)	$O(B_I)$	$O(B_I)$	$O(\log_F B_I)$	$O(\log_F B_I)$
delete	O(B)	$O(\log_2 B)$	$O(\log_2 B_I)$	$O(\log_2 B_I + m)$	$O(\log_F B_I)$	$O(\log_F B_I + m)$

Notes:
- Insert time for sequential files and non-tree indexes reflects time to re-sort: O(B)
- For B+-trees, F is the fanout of the tree
  - F = ceiling((k+1)/2)

Hash File Organization and Hash Indexes

- A **hash table** is a data structure mapping keys to values
  - Consists of a **hash function** and one or more **buckets**
  - Each bucket stores values associated with one or more keys
  - The hash function h(k) takes a key as input and outputs a bucket address
- All values associated with a given key will be put in the same bucket
- A single bucket may contain values for more than one key

Hash File Organization

- Hash files organized so that all records with a given search key value are stored in the same block on disk
  - disk blocks are the buckets
  - h(search key) = block address
- Example: instructor relation organized as a hash file with dept_name as search key
  - 10 buckets
  - h(k) could be sum of integer values of characters mod 10
    - h(music) = 1, h(history) = 2, h(physics) = 3, h(elec. eng) = 3, etc

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

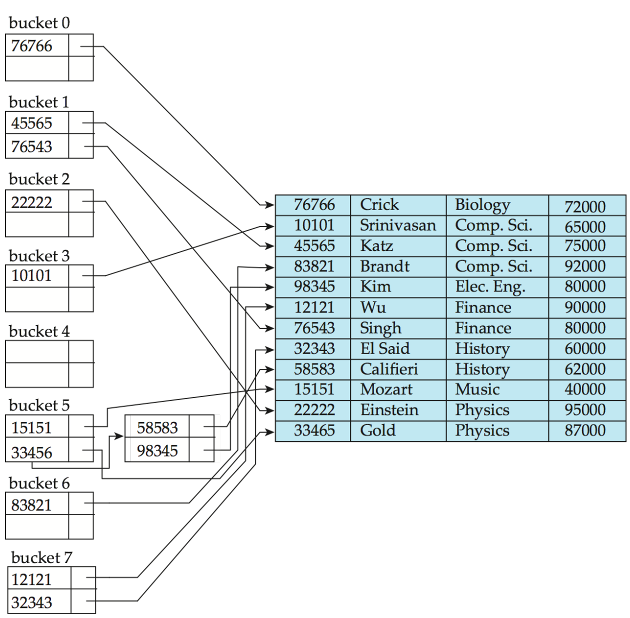
bucket 7

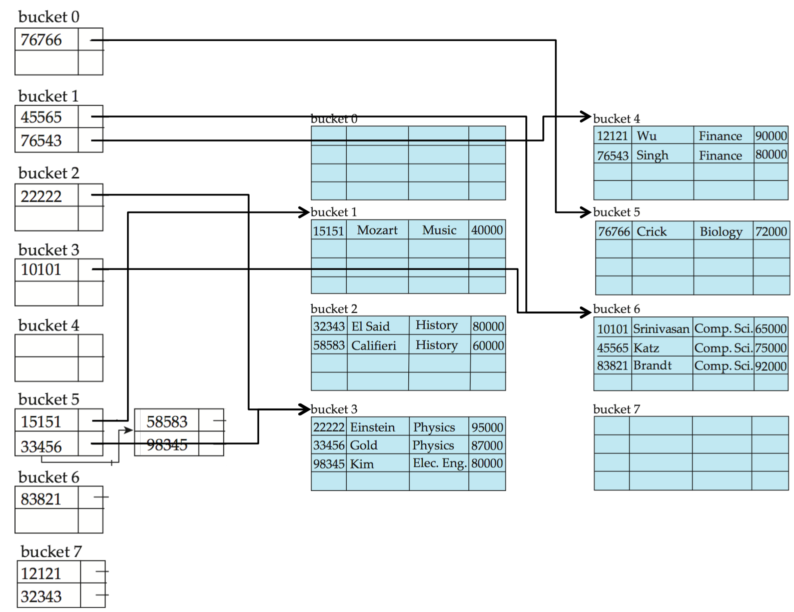
Overflow

- Impossible to guarantee that each block/bucket will have enough space for all records that hash there
- **Overflow** buckets store records that don't fit in the original bucket
  - **Closed hashing**: overflow buckets chained together in linked list
  - Each block contains pointer to its first overflow block
    - Each overflow block contains pointer to the next one

Hash Indexes

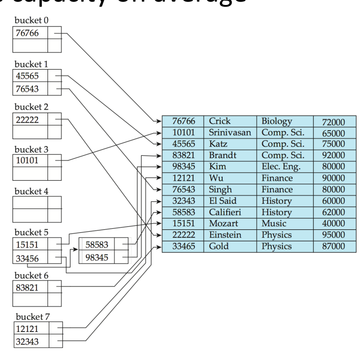
- A **hash index** maps search keys to blocks where index records are stored
  - Difference between hash index and hash file is that the index contains **only** search key values
- Note: a file organized using a hash function is already effectively indexed by a search key
  - Thus doesn't really make sense to talk about "primary indexes" on hash files
  - **All** hash indexes are secondary
  - But we will use "hash index" to refer to both a hash file org as well as a hash index





Heap File With Hash Index

- Assume search key is primary key, so there is one index record per actual record
- Assume index records are 10% size of actual records
- Assume index blocks are at 80% capacity on average
- Space costs:
  - Overhead?
- Time costs:
  - Scan?
  - Search with equality?
  - Search for range?
  - Insert?
  - Delete?



Pretend this is a heap file

Hash Indexes in Practice

- MySQL does not support hash indexes for tables stored on disk
  - It will let you execute `CREATE INDEX .. HASH`, but it will create a B+-tree index (without telling you)
  - Many DBMSs do not support them
- Why?
  - Difficult to predict how much space to reserve for index without knowing how many keys there will be
    - Which means it's difficult to decide on a good hash function
  - Efficiency of equality search with hash index rarely better than B+-tree with high k
  - Completely useless for range search
  - Lose the nice side-effect of a B+-tree that key values are already sorted
    - Which often means more post-processing time to sort
  - All in all: time spent coding hash index generally not worth the small gains that can be achieved given the losses that might be incurred

Analysis Table

	heap	sequential	seq + primary index	primary B+-tree index	secondary B+-tree index	hash
scan	O(B)	O(B)	$O(B_I)$ index scan	$O(B_I)$ index scan	$O(B_I)$ index scan	O(B) or $O(B_I)$
= search	O(B)	$O(\log_2 B)$	$O(\log_2 B_I)$	$O(\log_F B_I)$	$O(\log_F B_I + m)$	$O(B_m)$
<> search	O(B)	$O(\log_2 B + m)$	$O(\log_2 B_I + m)$	$O(\log_F B_I + m)$	$O(\log_F B_I + m)$	O(B)
insert	O(1)	O(B)	$O(B_I)$	$O(\log_F B_I)$	$O(\log_F B_I)$	$O(B_m)$
delete	O(B)	$O(\log_2 B)$	$O(\log_2 B_I)$	$O(\log_F B_I)$	$O(\log_F B_I + m)$	$O(B_m)$

Notes:

- Insert time for sequential files and non-tree indexes reflects time to re-sort: O(B)
- For B+-trees, F is the fanout of the tree
  - F = ceiling((k+1)/2)
- For hash storage, B_m is the number of blocks in the relevant bucket
  - $B_m < B_I$
  - But difficult to quantify relationship between B_m and $\log_F B_I$

Indexes and Performance Tuning

- A **workload** is a mix of queries and update operations
- We'd like to create indexes that will support the expected workload efficiently
- For each query in the workload:
  – What relations does it access?
  – What attributes are retrieved?
  – Which attributes are involved in select/join clauses?
  – How selective are those conditions?
- For each update in the workload:
  – What type of update (INSERT/UPDATE/DELETE)?
  – What attributes are affected?

Choosing Indexes

- What indexes should we create?
  – Which relations need indexes?
  – What fields should be used as search keys?
  – Do we need more than one index for a relation?
- What type of index?
  – Clustered? Hash? B+-tree?
  – No choice in MySQL: all files stored as sequential files with B+-tree indexes on primary key, and all secondary indexes are B+-tree indexes

