Database Management Systems

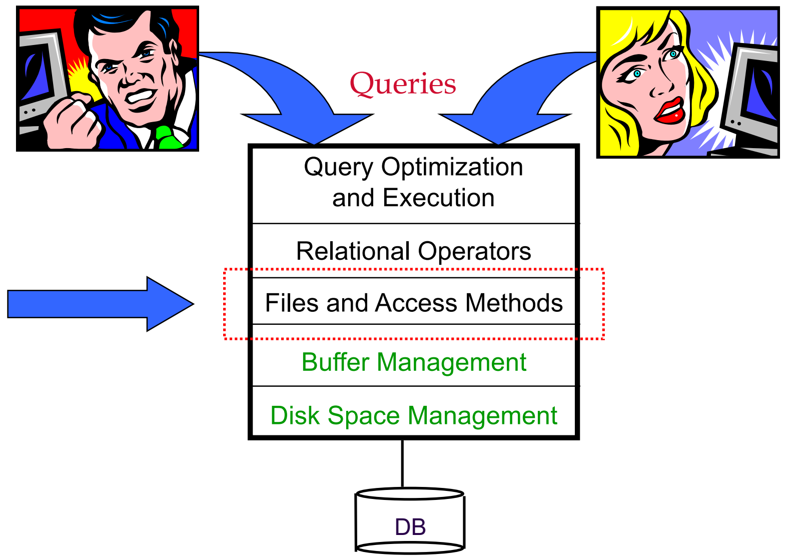
Fall 2014

Indexing

Chapters 8, 10, and 11

"If I had eight hours to chop down a tree,
I'd spend six sharpening my ax."
-- Abraham Lincoln

The BIG Picture



The Problem(s) with Sorted Files

1) Expensive to maintain

- Especially if you want to keep the records packed tightly.
- Q: What if you are willing to relax that constraint?

2) Can only sort according to a single search key

- File will effectively be a "heap" file for access via any other search key.
- e.g., how to search for a particular student id in a file sorted by major?

Indexes: Introduction

- **Sometimes, we want to retrieve records by specifying *values in one or more fields*, e.g.,**
  - Find all students in the "CS" department
  - Find all students with a gpa > 3.0
  - Find all students in CS with a gpa > 3.0
- **An *index* on a file is a disk-based data structure that speeds up selections on some *search key fields*.**
  - Any subset of the fields of a relation can be the search key for an index on the relation.
  - *Search key* is not the same as *key*
  - e.g., Search key doesn't have to be unique.

Indexes: Overview

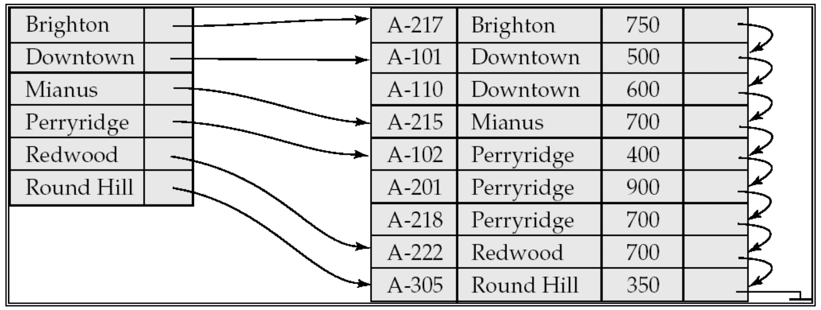
- **An index contains a collection of *data entries*, and supports efficient retrieval of all records with a given search key value** k**.**
  - Typically, index also contains auxiliary information that directs searches to the desired data entries
- **Many indexing techniques exist:**
  - B+ trees, hash-based structures, R trees, …
- **Can have multiple (different) indexes per file.**
  - E.g. file sorted by *age*, with a hash index on *salary* and a B+tree index on *name*.

Indexes: Overview

- **The index file usually occupies considerably less disk blocks than the data file because its entries are much smaller**
- **A binary search on the index yields a pointer to the file record**
- **Indexes can also be characterized as dense or sparse**

Dense Index Files

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

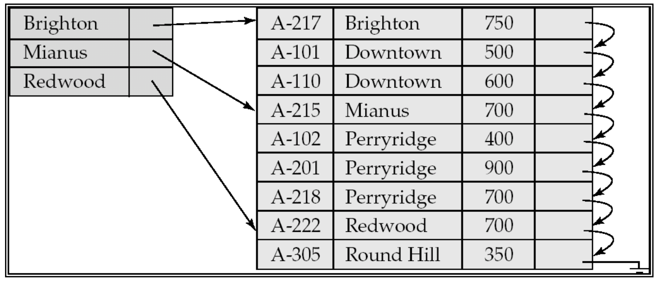


Brighton	
Downtown	
Mianus	
Perryridge	
Redwood	
Round Hill	

A-217	Brighton	750	
A-101	Downtown	500	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	

Sparse Index Files

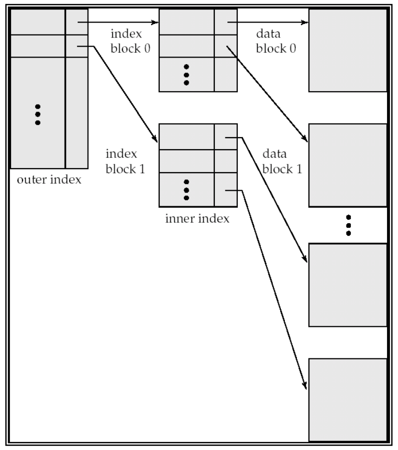
- **Sparse Index: contains index records for only some search-key values.**
  - Applicable when records are sequentially ordered on search-key
- **To locate a record with search-key value *K* we:**
  - Find index record with largest search-key value < *K*
  - Search file sequentially starting at the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



Brighton	
Mianus	
Redwood	

A-217	Brighton	750	
A-101	Downtown	500	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	

Multilevel Index (Cont.)



Basic Concepts

- **Indexing mechanisms used to speed up access to desired data.**
  - E.g., 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 Classification

1. **Selections (lookups) supported**
2. **Representation of data entries in index**
   - what kind of info is the index actually storing?
   - 3 alternatives here
3. **Clustered vs. Unclustered Indexes**
4. **Single Key vs. Composite Indexes**
5. **Tree-based, hash-based, other**

Indexes: Selections supported

field <*op*> constant

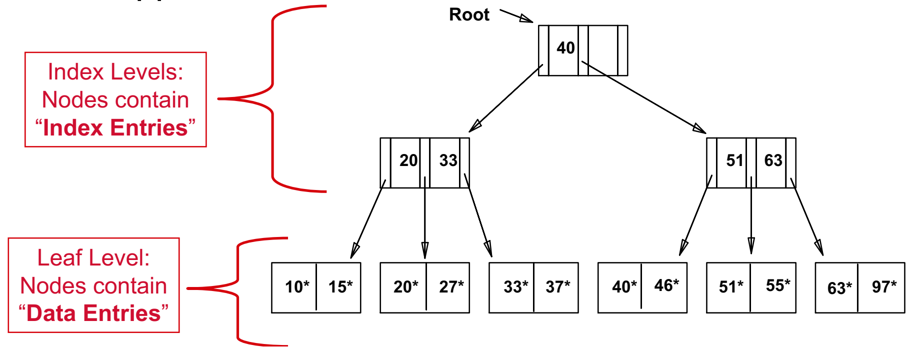
- **Equality selections (*op* is =)**
  - Either "tree" or "hash" indexes help here.
- **Range selections (*op* is one of <, >, <=, >=, BETWEEN)**
  - "Hash" indexes don't work for these.

More exotic selections

- multi-dimensional ranges ("east of Berkeley and west of Truckee and North of Fresno and South of Eureka")
- multi-dimensional distances ("within 2 miles of Soda Hall")
- Ranking queries ("10 restaurants closest to Berkeley")
- Regular expression matches, genome string matches, etc.
- Keyword/Web search - includes "importance" of words in documents, link structure, …

Tree Index: Example

- *Index entries*:<search key value, page id> they direct search for data entries *in leaves.*
- In example: Fanout (F) = 3 (note: unrealistic!)
  - more typical: 16KB page, 67% full, 32Byte entries = approx 300


Root
40
20
33
51
63
10*
15*
20*
27*
33*
37*
40*
46*
51*
55*
63*
97*
Index Levels: Nodes contain "Index Entries"
Leaf Level: Nodes contain "Data Entries"

What's in a "Data Entry"?

- **Question: What is actually stored in the leaves of the index for key value "k"? (a data entry for key "k" is denoted "k*" in book and examples)**
- **Three alternatives:**
  1. Actual data record(s) with key value **k**
  2. {<**k**, rid of a matching data record>}
  3. <**k**, {rids of all matching data records}>
- **Choice is orthogonal to the indexing technique.**
  - e.g., B+ trees, hash-based structures, R trees, …

Alt 1= "Index-Organized File"

- **Actual data records are stored in leaves.**
  - If this is used, index structure becomes a file organization for data records (e.g., a sorted file).
  - At most one index on a given collection of data records can use Alternative 1.
  - This alternative saves pointer lookups but can be expensive to maintain with insertions and deletions.

Index-Organized File

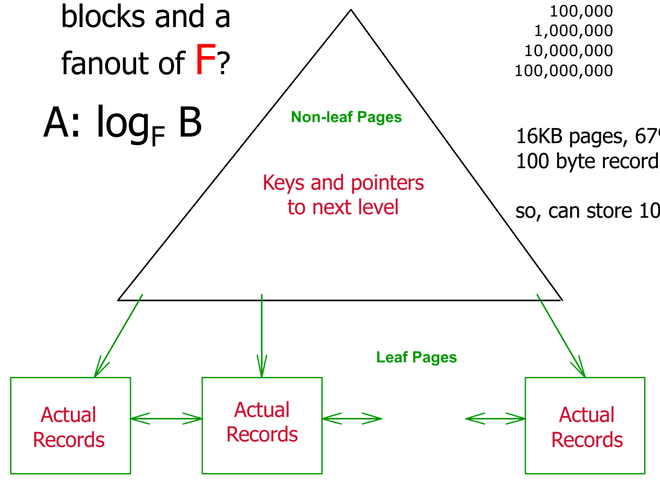
Q: How many levels if B leaf blocks and a fanout of F?

A: $\log_F B$

# Leaf Blocks	Fanout	Levels
1,000	300	3
10,000	300	3
100,000	300	4
1,000,000	300	4
10,000,000	300	4
100,000,000	300	5

16KB pages, 67%full and 100 byte records = approx 100 recs/page.

so, can store 10B rows with 5 levels.



Note: All pages at all levels are: "Slotted Pages"

Operation Cost

B: The size of the data (in pages)
R: Number of records per page
D: (Average) time to read or write disk page

	Heap File	**Sorted File (100% Occupancy)**	**Index-Organized File (67% Occupancy)**
Scan all records	**BD**	BD	**1.5 BD** (bcos 67% full)
Equality Search *unique key*	**0.5 BD**	$(\log_2 B) * D$	$(\log_F 1.5B) * D$
Range Search	**BD**	$[(\log_2 B) + \#match\ pg] * D$	$((\log_F 1.5B) + \#match\ pg) * D$
Insert	**2D**	$((\log_2 B) + B)D$	$((\log_F 1.5B) + 1)D$
Delete	$(0.5B+1)D$	$((\log_2 B) + B)D$ *(because rd,wrt 0.5 file)*	$((\log_F 1.5B) + 1)D$

Alternatives for Data Entries (Contd.)

Alternative 2

{<k, rid of a matching data record>}

and Alternative 3

<k, {rids of all matching data records}>

- Easier to maintain than Index-Organized.
  - On the other hand: Index-organized could be faster for reads.
- If more than one index is required on a given file, at most one index can use Alt 1; rest must use 2 or 3.
- Alt 3 more compact than Alt 2, but has *variable sized* data entries even with fixed-length search keys
- Even worse, for large rid lists the data entry would have to span multiple blocks!

Clustered vs. Unclustered Index

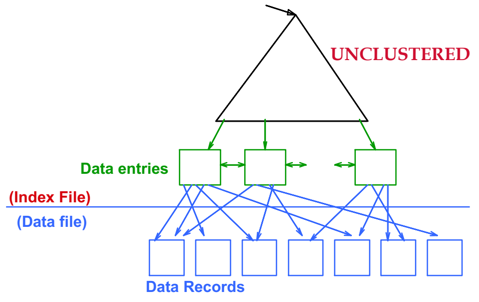
"Clustered" Index: the order of data records is the same as, or `close to', the order of index data entries.

- **A file can be clustered on at most one search key.**
- **Cost of retrieving data records through index varies *greatly* based on whether index is clustered or not!**
- **Index-organized implies clustered *but not vice-versa*.**
  - In other words, alt-1 is always clustered
  - alt 2 and alt 3 may or may not be clustered.

Example: Alt 2 index for a Heap File

For alts 2 or 3, typically two files – one for data records and one for the index.

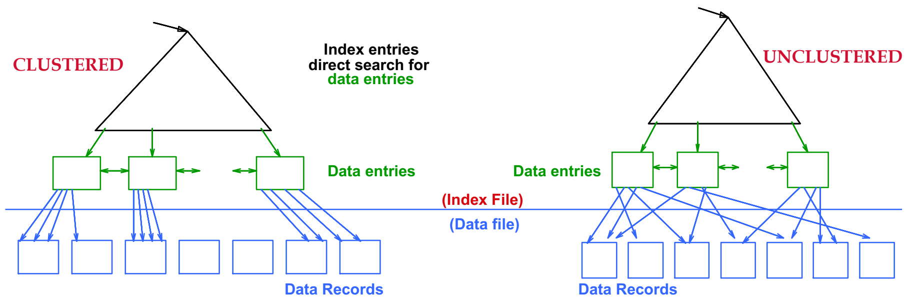
For an unclustered index, the order of data records in the data file is unrelated to the order of the data entries in the leaf level of the index.



Example: Alt 2 index for a Heap File

For a clustered index:

- **Sort the heap file on the search key column(s)**
  - Leave some free space on pages for future inserts
- **Build the index**
- **Use overflow pages in data file if necessary**
  - Thus, clustering is only approximate – data records may not be exactly in sort order (can clean up later)



Unclustered vs. Clustered Indexes

- **What are the tradeoffs????**
- **Clustered Pros**
  - Efficient for range searches
  - May be able to do some types of compression
  - Possible locality benefits (related data?)
  - ???
- **Clustered Cons**
  - Maintenance cost (pay on the fly or be lazy with reorganization)
  - Can only cluster according to a single order

Operation Cost

B: The size of the data (in pages)

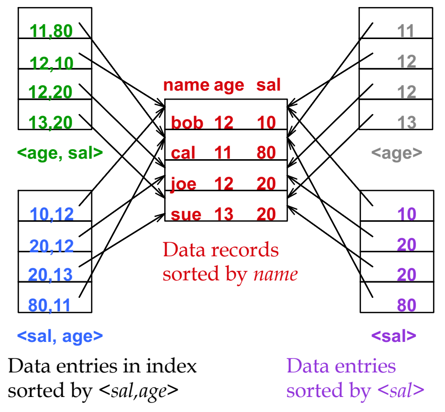
D: (Avg) time to read or write disk page

	Unclustered Alt-2 Tree Idx (Index file: 67% occupancy) (Data file: 100% occupancy)	**Clustered Alt-2 Tree Idx (Index and Data files: 67% occupancy)**
Scan all records	BD (ignore index)	1.5 BD (ignore index)
Equality Search *unique key*	$(1+ \log_F 0.5 B) * D$ assume an index entry is 1/3 the size of a record so index leaf level = .33 * 1.5B = 0.5B	$(1+ \log_F 0.5B) * D$
Range Search	$[(\log_F 0.5B) +$ #matching_leaf_pages – 1 + #match records$]*D$	$((\log_F 0.5B) +$ #match pages$)*D$
Insert	$((\log_F 0.5B)+3)D$	$((\log_F 0.5B)+3)D$
Delete	same as insert	same as insert

Composite Search Keys

- **Search on a combination of fields.**
  - Equality query: Every field value is equal to a constant value. E.g. wrt <age,sal> index:
    - age=20 and sal =75
  - Range query: Some field value is not a constant. E.g.:
    - age > 20; or age=20 and sal > 10
- **Data entries in index sorted by search key to support range queries.**
  - Lexicographic order
  - Like the dictionary, but on fields, not letters!

Examples of composite key indexes using lexicographic order.



Index Classification Revisited

1. Selections (lookups) supported

2. Representation of data entries in index

- what kind of info is the index actually storing?
- 3 alternatives here

3. Clustered vs. Unclustered Indexes

4. Single Key vs. Composite Indexes

5. Tree-based, hash-based, other

Tree-Structured Indexes

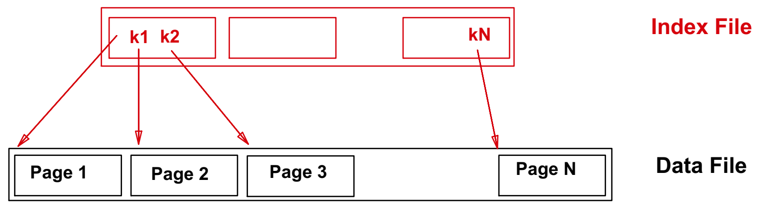
- **Tree-structured indexing techniques support both *range searches* and *equality searches*.**
- **Two examples:**
  - *ISAM*: static structure; early index technology.
  - *B+ tree*: dynamic, adjusts gracefully under inserts and deletes.

ISAM = Indexed Sequential Access Method

- **ISAM is an old-fashioned idea**
  - B+ trees are usually better, as we'll see
    - Though not *always*
- **But, it's a good place to start**
  - Simpler than B+ tree, but many of the same ideas
- **Upshot**
  - **Don't** brag about being an ISAM expert on your resume
  - **Do** understand how they work, and tradeoffs with B+ trees

Range Searches

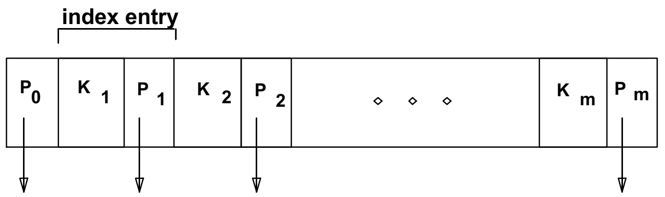
- ***``Find all students with gpa > 3.0''***
  - If data is in sorted file, do binary search to find first such student, then scan to find others.
  - Cost of binary search on disk is still quite high.
- **Simple idea: Create an `index' file.**



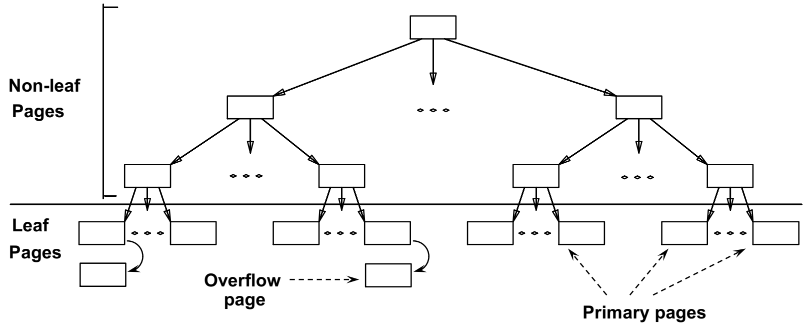
☛ *Can do binary search on (smaller) index file!*

☛*But what if index doesn't fit easily in memory?*

ISAM

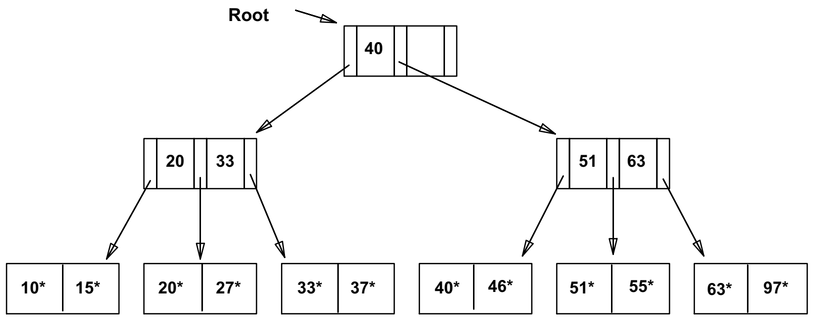


- **We can apply the idea repeatedly!**



Example ISAM Tree

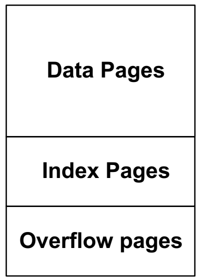
- *Index entries*:<search key value, page id> they direct search for data entries *in leaves.*
- Example where each node can hold 2 entries;



ISAM has a STATIC Index Structure

File creation:

1. Allocate leaf (data) pages sequentially
2. Sort records by search key
3. Allocate index pages
4. Allocate overflow pages



ISAM File Layout

Static tree structure: *inserts/deletes affect only leaf pages.*

ISAM (continued)

Data Pages
Index Pages
Overflow pages

Search: Start at root; use key comparisons to navigate to leaf.

Cost = $\log_F N$

F = # entries/pg (i.e., fanout)

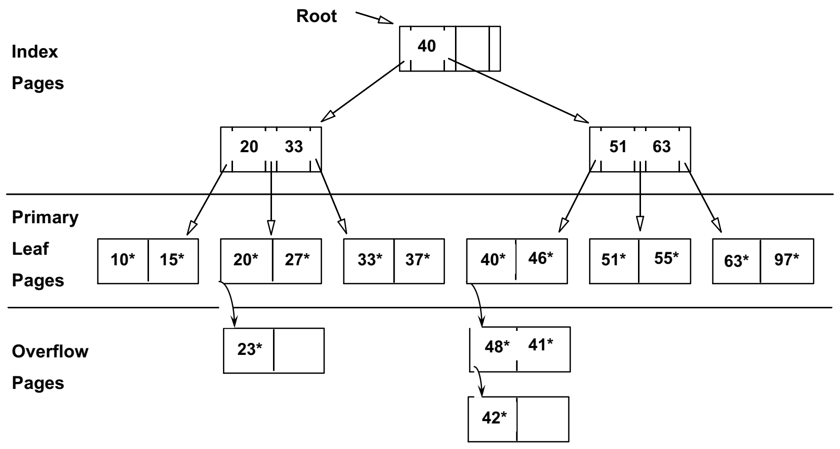
N = # leaf pgs

- no need for `next-leaf-page' pointers. (Why?)

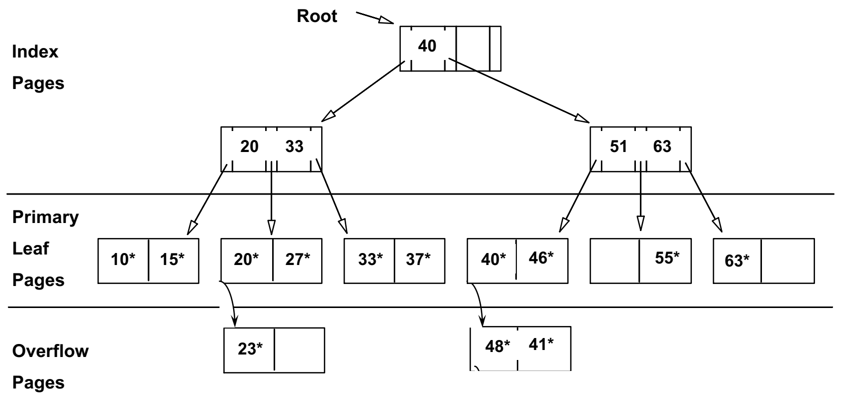
Insert: Find leaf that data entry belongs to, and put it there. Overflow page if necessary.

Delete: Find; remove from leaf; if empty de-allocate.

Example: Insert 23*, 48*, 41*, 42*



... then Deleting 42*, 51*, 97*



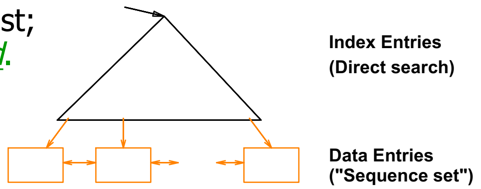
☛ *Note that 51* appears in index levels, but not in leaf!*

ISAM ---- Issues?

- **Pros**
  - ????
- **Cons**
  - ????

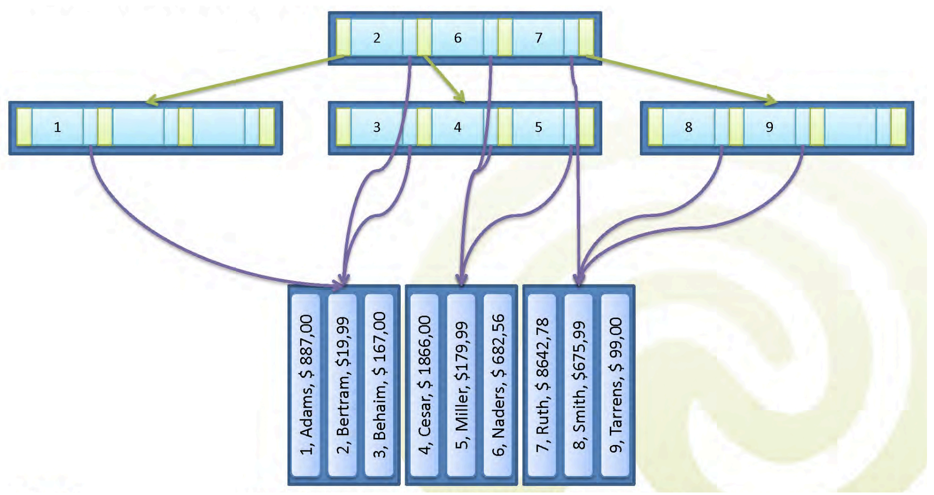
B+ Tree: The Most Widely Used Index

- Insert/delete at log $_F$ N cost; keep tree *height-balanced*.
  N = # leaf pages



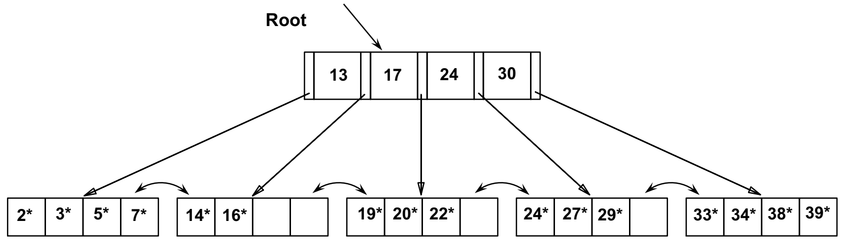
- Each node (except for root) contains *m entries:* d <= *m* <= 2d entries.
- "d" is called the *order* of the tree.
  (maintain 50% min occupancy)
- Supports equality and range-searches efficiently.
- As in ISAM, all searches go from root to leaves, but structure is dynamic.

B+ Tree: The Most Widely Used Index



Example B+ Tree

- **Search begins at root page, and key comparisons direct it to a leaf (as in ISAM).**
- **Search for 5*, 15*, all data entries >= 24* ...**



☛ *Based on the search for 15*, we know it is not in the tree!*

A Note on Terminology

- The "+" in B^+Tree indicates that it is a special kind of "B Tree" in which all the data entries reside in leaf pages.
  - In a vanilla "B Tree", data entries are sprinkled throughout the tree.
- B^+Trees are in many ways simpler to implement than B Trees.
  - And since we have a large fanout, the upper levels comprise only a tiny fraction of the total storage space in the tree.
- To confuse matters, most database people call B^+Trees "B Trees"!!!

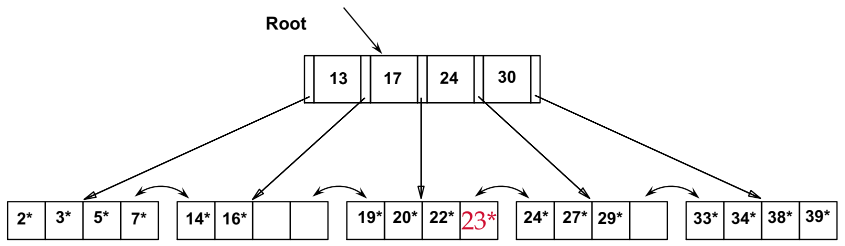
B+ Trees in Practice

- **Remember = Index nodes are disk pages**
  - e.g., fixed length unit of communication with disk
- **Typical order: 100. Typical fill-factor: 67%.**
  - average fanout = 133
- **Typical capacities:**
  - Height 3: 133^3 = 2,352,637 entries
  - Height 4: 133^4 = 312,900,700 entries
- **Can often hold top levels in buffer pool:**
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

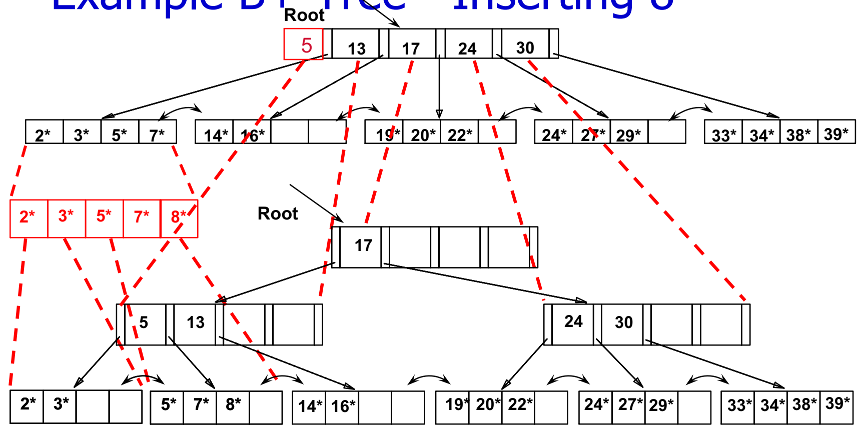
Inserting a Data Entry into a B+ Tree

- **Find correct leaf *L.***
- **Put data entry onto *L*.**
  - If *L* has enough space, *done*!
  - Else, must *split* *L (into L and a new node L2)*
    - Redistribute entries evenly, **copy up** middle key.
    - Insert index entry pointing to *L2* into parent of *L*.
- **This can happen recursively**
  - To split index node, redistribute entries evenly, but **push up** middle key. (Contrast with leaf splits.)
- **Splits "grow" tree; root split increases height.**
  - Tree growth: gets *wider* or *one level taller at top.*

Example B+ Tree – Inserting 23*



Example B+ Tree - Inserting 8*

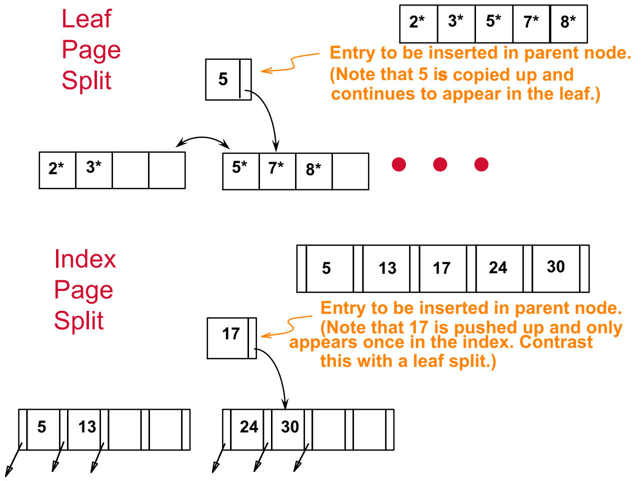


❖ Notice that root was split, leading to increase in height.

❖ In this example, we could avoid split by re-distributing entries; however, this is not done in practice.

Leaf vs. Index Page Split
(from previous example of inserting "8")

- **Observe how minimum occupancy is guaranteed in both leaf and index pg splits.**
- **Note difference between *copy-up* and *push-up*; be sure you understand the reasons for this.**

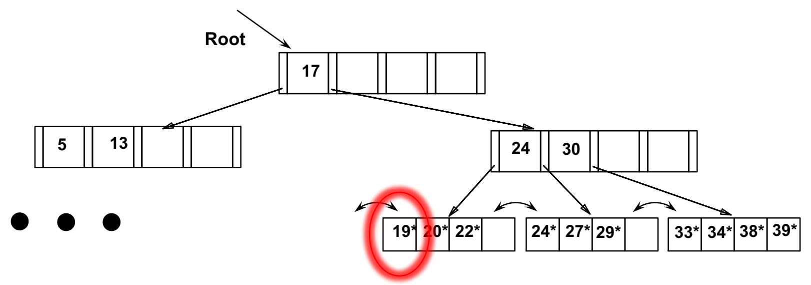


Leaf Page Split
2* 3* 5* 7* 8*
5
Entry to be inserted in parent node.
(Note that 5 is copied up and continues to appear in the leaf.)
2* 3*
5* 7* 8*
Index Page Split
5 13 17 24 30
17
Entry to be inserted in parent node.
(Note that 17 is pushed up and only appears once in the index. Contrast this with a leaf split.)
5 13
24 30

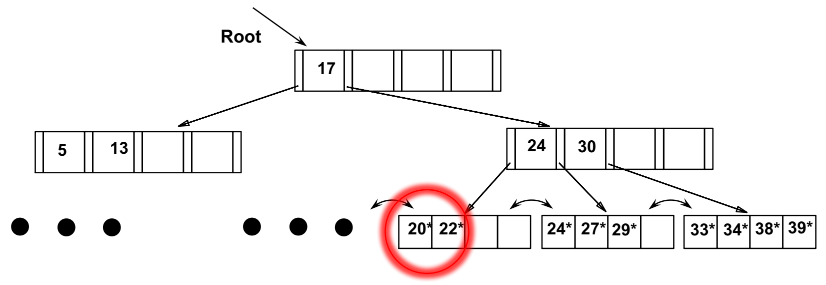
Deleting a Data Entry from a B+ Tree

- **Start at root, find leaf *L* where entry belongs.**
- **Remove the entry.**
  - If L is at least half-full, *done!*
  - If L has only **d-1** entries,
    - Try to re-distribute, borrowing from *sibling* *(adjacent node with same parent as L)*.
    - If re-distribution fails, *merge* *L* and sibling.
- **If merge occurred, must delete entry (pointing to *L* or sibling) from parent of *L*.**
- **Merge could propagate to root, decreasing height.**

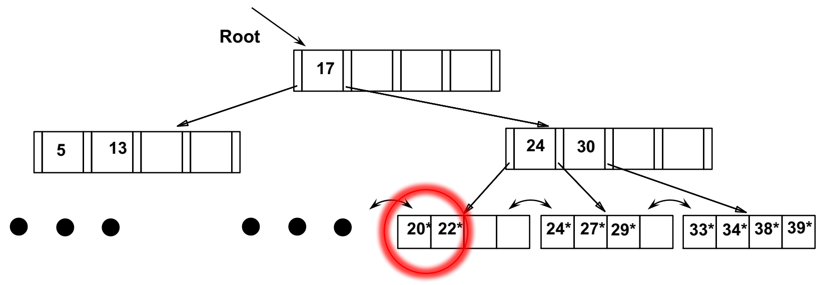
Example Tree - Delete 19*



Example Tree - Delete 19*

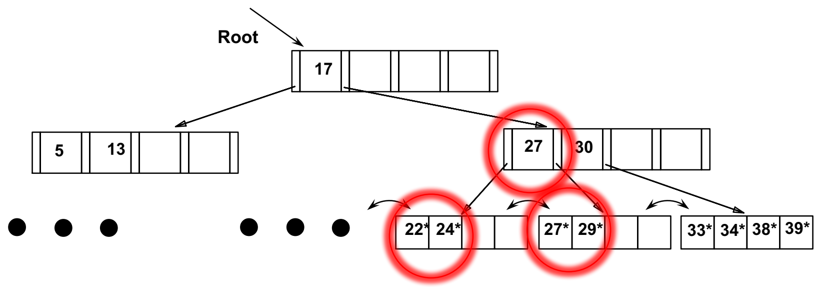


Example Tree – Now, Delete 20*

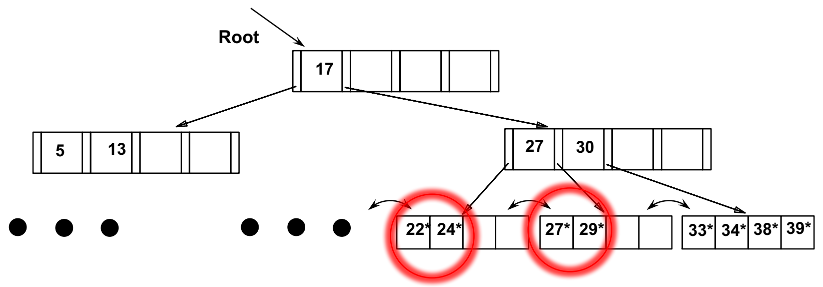


Redistribute

Example Tree – Then Delete 20*



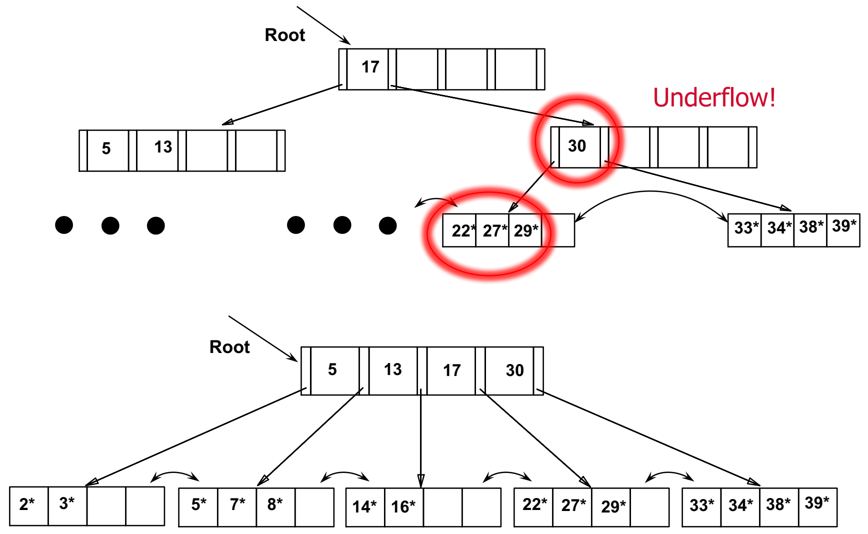
Example Tree – Then Delete 24*



Underflow!

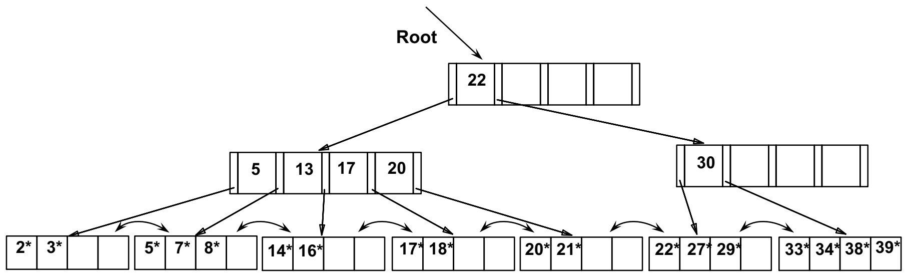
Can't redistribute, must Merge…

Example Tree – Then Delete 24*



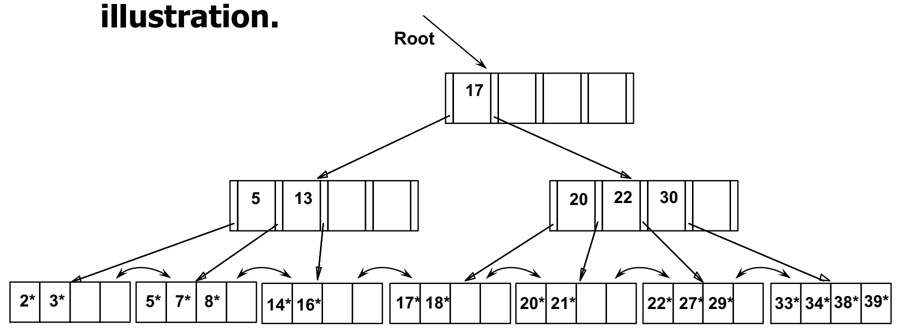
Example of Non-leaf Re-distribution

- **Tree is shown below *during deletion* of 24***
- **In contrast to previous example, can re-distribute entry from left child of root to right child.**



After Re-distribution

- **Intuitively, entries are re-distributed by *`pushing through'* the splitting entry in the parent node.**
- **It suffices to re-distribute index entry with key 20; we've re-distributed 17 as well for illustration.**



A Note on `Order'

- ***Order*** (d) **concept replaced by physical space criterion in practice (*`at least half-full'*).**
  - Index pages can typically hold many more entries than leaf pages.
  - Variable sized records and search keys mean different nodes will contain different numbers of entries.
  - Even with fixed length fields, multiple records with the same search key value (*duplicates*) can lead to variable-sized data entries (if we use Alternative (3)).
- **Many real systems are even sloppier than this -- only reclaim space when a page is *completely* empty.**

Introduction to Hash-based Indexes

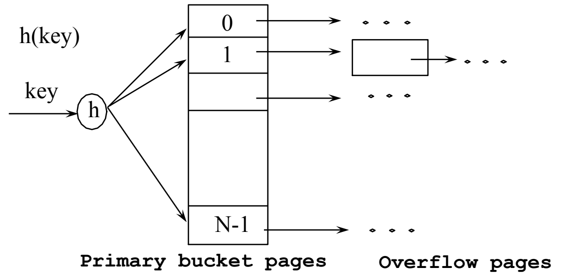
- ***Hash-based*** **indexes are best for *equality selections*.** *Cannot* **support range searches.**
- **Static and dynamic hashing techniques exist; trade-offs similar to ISAM vs. B+ trees.**

Static Hashing

- **# primary pages fixed, allocated sequentially, never de-allocated; overflow pages if needed.**
- A simple hash function (for N buckets):

  $h(k)$ = k MOD N

  is bucket # where data entry with key *k* belongs.



Static Hashing (Contd.)

- **Buckets contain *data entries*.**
- **Hash fn works on *search key* field of record *r*. Use MOD N to distribute values over range 0 ... N-1.**
  - **h**(*key*) = key MOD N works well for uniformly distributed data.
    - **h**(*key*) = (a * *key* + b).
    - a and b are constants; lots known about how to tune **h**.
  - various ways to tune **h** for non-uniform (checksums, crypto, etc.).
- **Buckets contain *data entries*.**
- **As with any static structure: Long overflow chains can develop and degrade performance.**
  - *Extendible* and *Linear Hashing*: Dynamic techniques to fix this problem.

Extendible Hashing

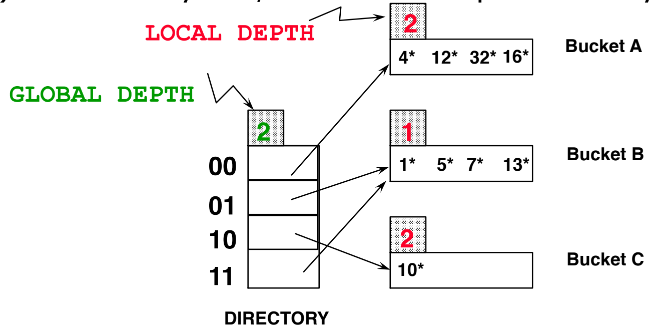
- **Situation: Bucket (primary page) becomes full.**
  - Want to avoid overflow pages
- **How about we add more buckets (i.e., increase "N")?**
  - Okay, but need a new hash function!
- ***Doubling* # of buckets makes this easier**
  - Say N values are powers of 2 – how to do "mod N"?
  - What happens to hash function when you double "N"?
- **Problems with Doubling**
  - Don't want to have to double the size of the file.
  - Don't want to have to move all the data.

Extendible Hashing (continued)

- ***Idea*: Add a level of indirection!**
- **Use *directory of pointers to buckets,***
- **Double # of buckets by *doubling the directory***
  - Directory much smaller than file, so doubling it is much cheaper.
- **Split only the bucket that just overflowed!**
  - *No overflow pages*!
  - Trick lies in how hash function is adjusted!

Extendible Hashing – How it Works

- **Directory is array of size 4, so 2 bits needed.**
- **Bucket for record *r* has entry with index = `*global depth*' least significant bits of h(*r*);**
  - If **h**(*r*) = 5 = binary 101, it is in bucket pointed to by 01.
  - If **h**(*r*) = 7 = binary 111, it is in bucket pointed to by 11.

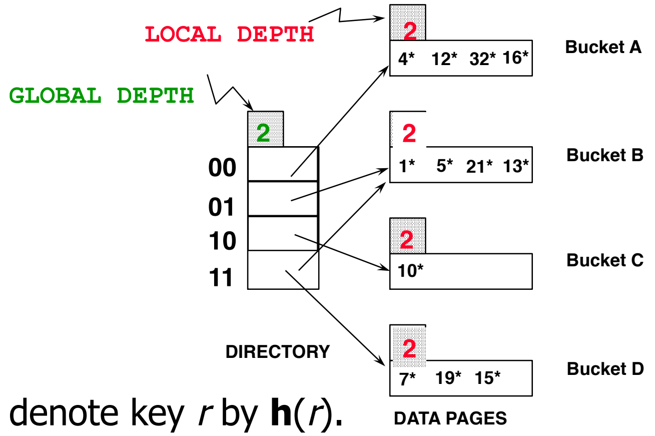


Handling Inserts

- Find bucket where record belongs.
- If there's room, put it there.
- Else, if bucket is full, *split* it:
  - increment local depth of original page
  - allocate new page with new local depth
  - re-distribute records from original page.
  - add entry for the new page to the directory

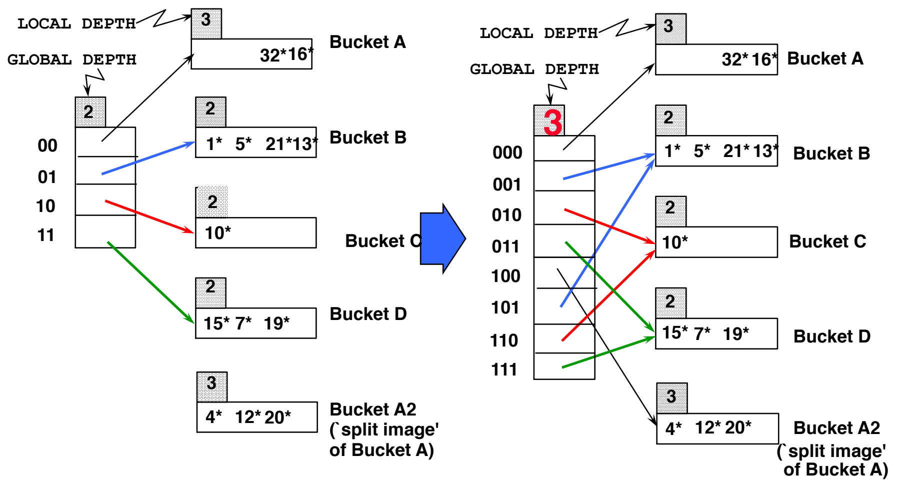
Example: Insert 21, then 19, 15

- **21 = 10101**
- **19 = 10011**
- **15 = 01111**



we denote key *r* by **h**(*r*).

Insert **h**(r)=20 (Causes Doubling)

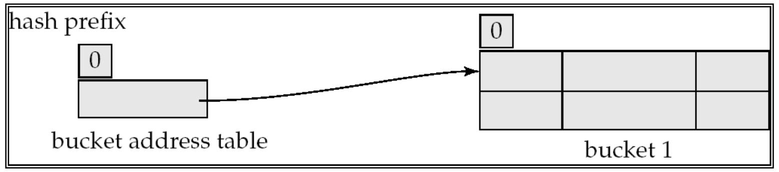


Points to Note

- **20 = binary 10100. Last** 2 **bits (00) tell us *r* belongs in either A or A2. Last** 3 **bits needed to tell which.**
  - *Global depth of directory*: Max # of bits needed to tell which bucket an entry belongs to.
  - *Local depth of a bucket*: # of bits used to determine if an entry belongs to this bucket.
- **When does bucket split cause directory doubling?**
  - Before insert, *local depth* of bucket = *global depth*. Insert causes *local depth* to become > *global depth*; directory is doubled by *copying it over* and `fixing' pointer to split image page.

Use of Extendible Hash Structure: Example

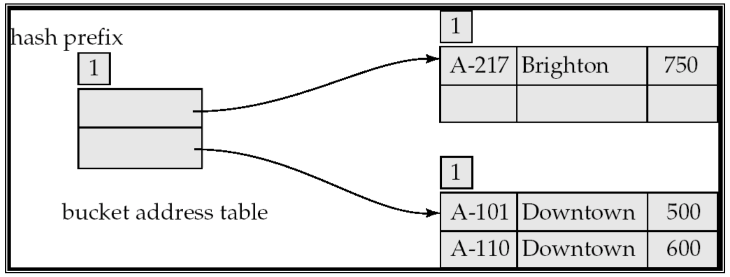
branch_name	h(*branch_name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111 1001 1100 0000 0001



Initial Hash structure, bucket size = 2

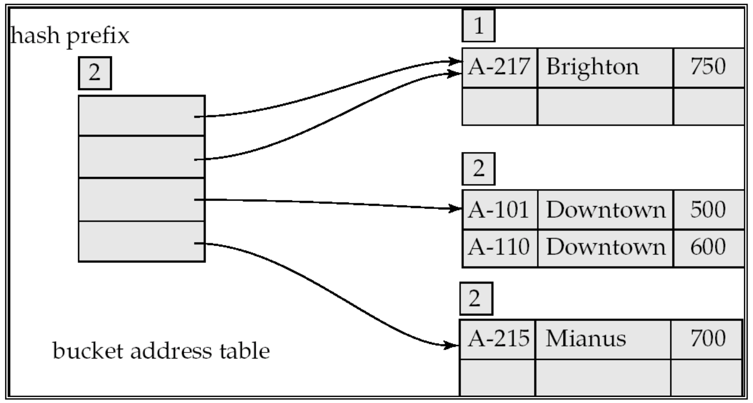
Example (Cont.)

- **Hash structure after insertion of one Brighton and two Downtown records**

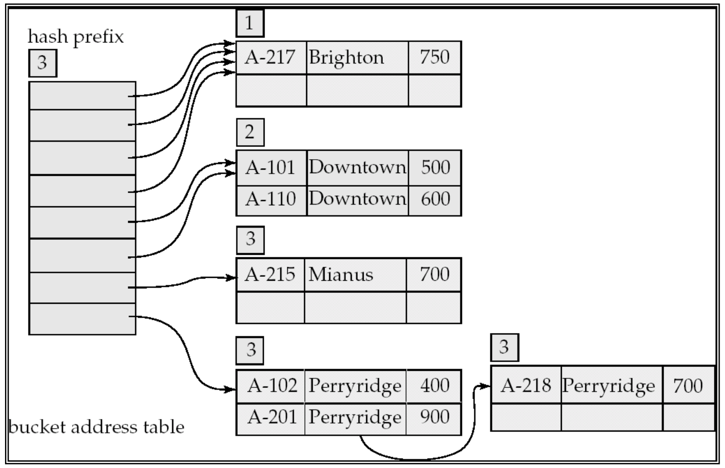


Example (Cont.)

Hash structure after insertion of Mianus record



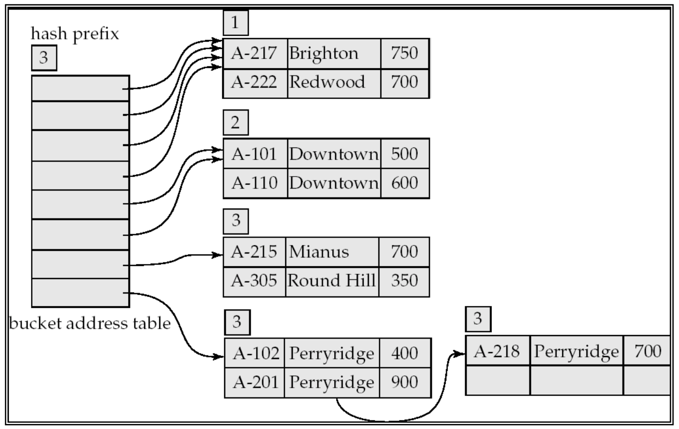
Example (Cont.)



Hash structure after insertion of three Perryridge records

Example (Cont.)

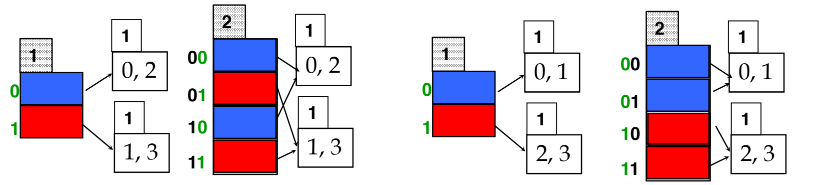
- **Hash structure after insertion of Redwood and Round Hill records**



Directory Doubling

Why use least significant bits in directory (instead of the *most* significant ones)?

Allows for doubling by copying the directory and appending the new copy to the original.



Least Significant vs. Most Significant

Q: Any other reasons?

Comments on Extendible Hashing

- **If directory fits in memory, equality search answered with one disk access; else two.**
  - 100MB file, 100 bytes/rec, 4K pages contains 1,000,000 records (as data entries) and 25,000 directory elements; chances are high that directory will fit in memory.
  - Directory grows in spurts, and, if the distribution *of hash values* is skewed, directory can grow large.
  - Multiple entries with same hash value cause problems!

- Delete: **If removal of data entry makes bucket empty, can be merged with `split image'. If each directory element points to same bucket as its split image, can halve directory.**

Summary

- **Tree-structured indexes are ideal for range-searches, also good for equality searches.**
- **ISAM is a static structure.**
  - Only leaf pages modified; overflow pages needed.
  - Overflow chains can degrade performance unless size of data set and data distribution stay constant.
- **B+ tree is a dynamic structure.**
  - Inserts/deletes leave tree height-balanced; log $_F$ N cost.
  - High fanout (**F**) means depth rarely more than 3 or 4.
  - Almost always better than maintaining a sorted file.

Summary (Contd.)

- Typically, 67% occupancy on average.
- Usually preferable to ISAM, modulo *locking* considerations; adjusts to growth gracefully.
- If data entries are data records, splits can change rids!

- **Other topics:**
  - Key compression increases fanout, reduces height.
  - Bulk loading can be much faster than repeated inserts for creating a B+ tree on a large data set.
- **Most widely used index in database management systems because of its versatility. One of the most optimized components of a DBMS.**

Summary – Hash Indexes

- **Hash-based indexes: best for equality searches, cannot support range searches.**
- **Static Hashing can lead to long overflow chains.**
- **Extendible Hashing avoids overflow pages by splitting a full bucket when a new data entry is to be added to it. (*Duplicates may require overflow pages.*)**
  - Directory to keep track of buckets, doubles periodically.
  - Can get large with skewed data; additional I/O if this does not fit in main memory.
- **"Linear hashing" solves some problems of Extendible hashing – not covered in this course, but check out book section 11.3 – it's very cool!**

Summary

- **Index Definition in SQL**
- **Create an index**

  -
    ```
    create index <index-name> on <relation-name>
                     (<attribute-list>)
    ```
  - E.g.: 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 is a candidate key.**
  - Not really required if SQL unique integrity constraint is supported
- **To drop an index**
  - drop index <index-name>
- **Most database systems allow specification of type of index, and clustering.**