

Database Systems (資料庫系統)

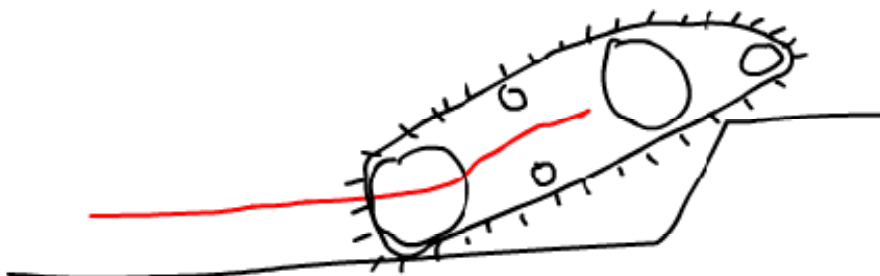
November 18, 2009

Lecture #8

1

K-Sketch (UW)

"Sketch" an animation for novice users



Announcement

- How was the midterm exam?
- Assignment #2 is due today.
- Assignment #3 (buffer manager) is out on the course webpage and due in 2 weeks.

3

Tree-Structured Indexing

Chapter 10

4

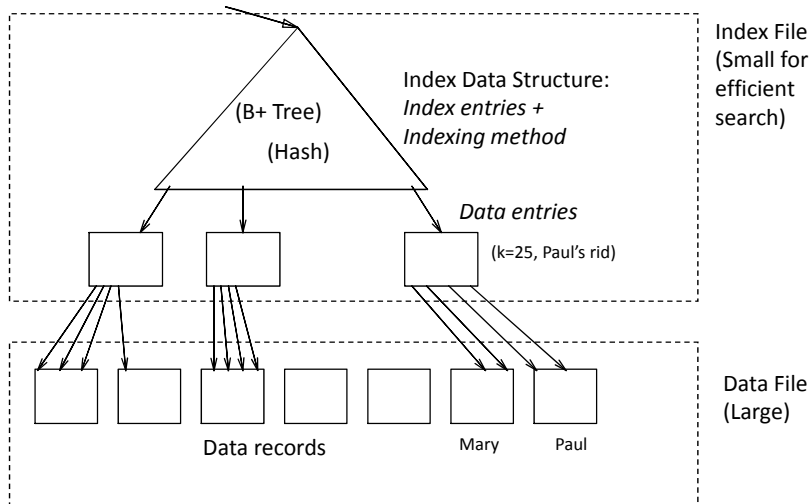
Outline

- Motivation for tree-structured indexes
- ISAM index
- B+ tree index
- Key compression
- B+ tree bulk-loading
- Clustered index

5

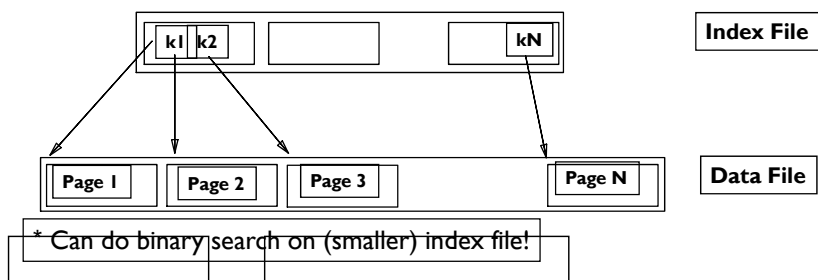
Review: Indexing Example

Search key value: find employees with age = 25



The Origin of [Tree] index

- Range search: *Find all students with gpa > 3.0*
 - Sorted data file: binary search to find first such student, then scan to find others.
 - Cost?
- Simple solution: create a smaller index file.
 - Cost of binary search over index file is reduced.



7

Why tree index?

- But, the index file can still be large.
 - The cost of binary search over the index file can still be large.
 - Can we further reduce search cost?
- Apply the simple solution again: create multiple levels of indexes.
 - Each index level is much smaller than the lower index level. This index structure is a tree.
- Say if a tree node is an index page holding, e.g., 100 indexes.
 - A tree with a depth of 4 (from the root index page to the leaf index page) can hold over [] records.
 - The cost of search is [] page access.

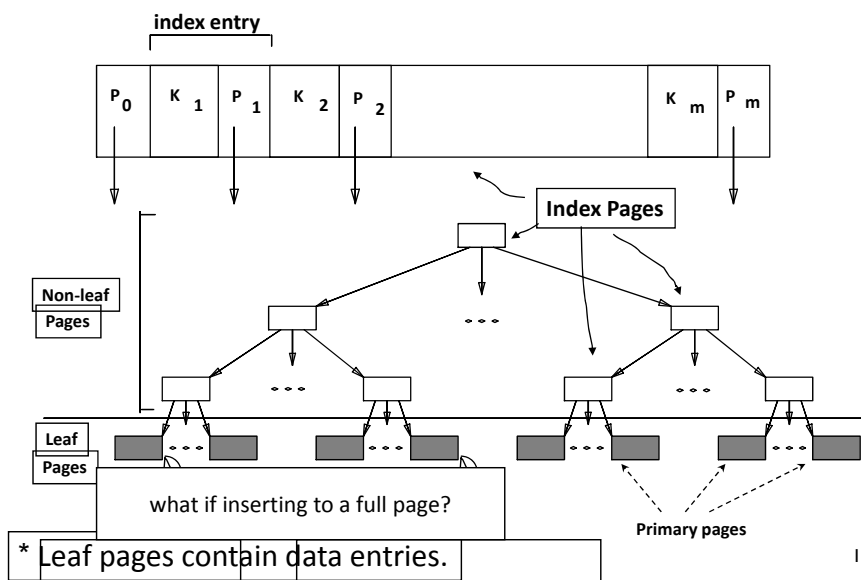
8

ISAM and B+ Tree

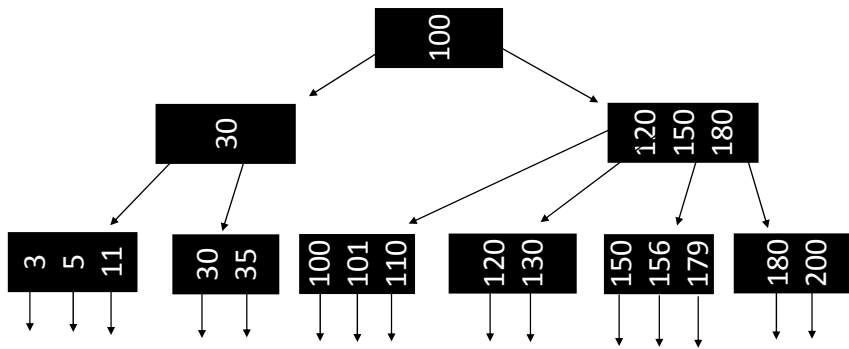
- Two tree-structured indexings:
 - ISAM (Indexed Sequential Access Method): static structure.
 - Assuming that the file does not grow or shrink too much.
 - B+ tree: dynamic structure
 - Tree structure adjusts gracefully under inserts and deletes.
- Analyze cost of the following operations:
 - Search
 - Insertion of data entries
 - Deletion of data entries
 - Concurrent access.

9

ISAM

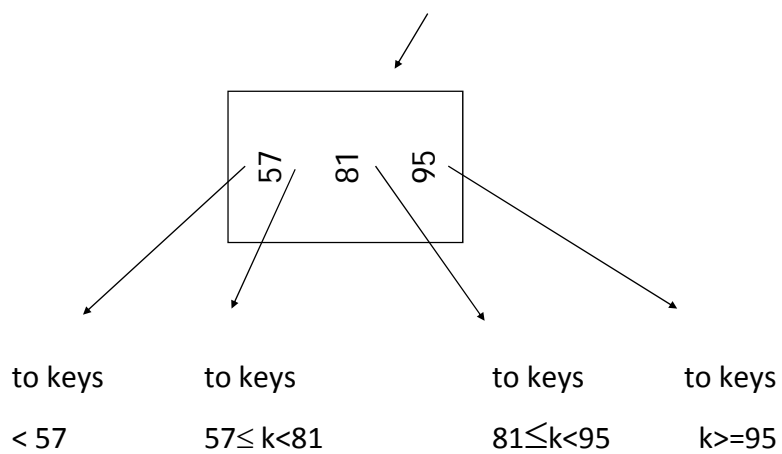


Example



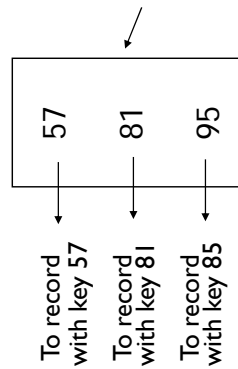
11

Non-leaf node



12

Leaf node



13

Comments on ISAM

- File creation:
 - Assume that data records are present and will not change much in the future.
 - Sort data records. Allocate data pages for the sorted data records.
 - Sort data entries based on the search keys. Allocate leaf index pages for sorted data entries sequentially.

14

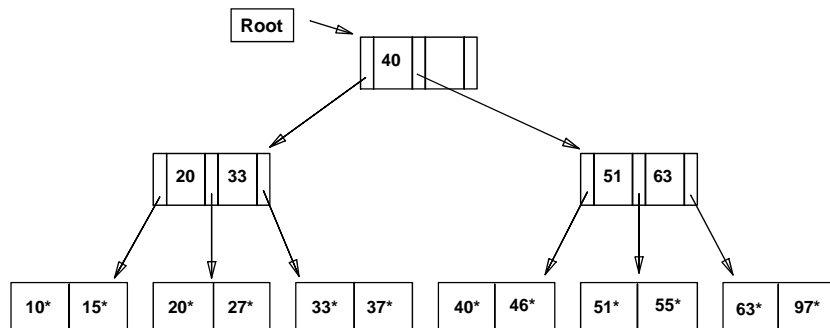
ISAM Operations

- Search: Start at root; use key comparisons to go to leaf.
 - Cost = $\log_F N$, where F = # entries/index page, N = # leaf pages
- Insert: Find the leaf page and put it there. If the leaf page is full, put it in the overflow page.
 - Cost = search cost + constant (assuming little or no overflow pages)
- Delete: Find and remove from the leaf page; if empty overflow page, de-allocate.
 - Cost = search cost + constant (assuming little or no overflow pages)

15

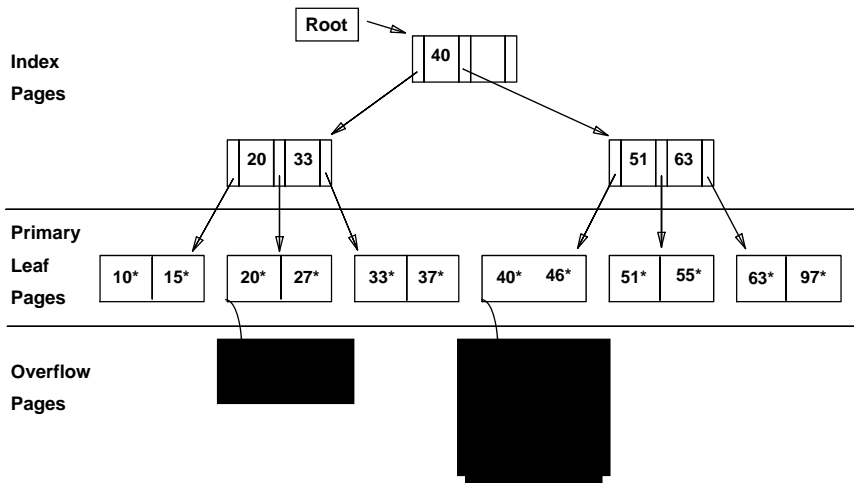
Example ISAM Tree

- Each node can hold 2 entries; no need for 'next-leaf-page' pointers in primary pages. Why not?
 - Primary pages are allocated sequentially at file creation time.



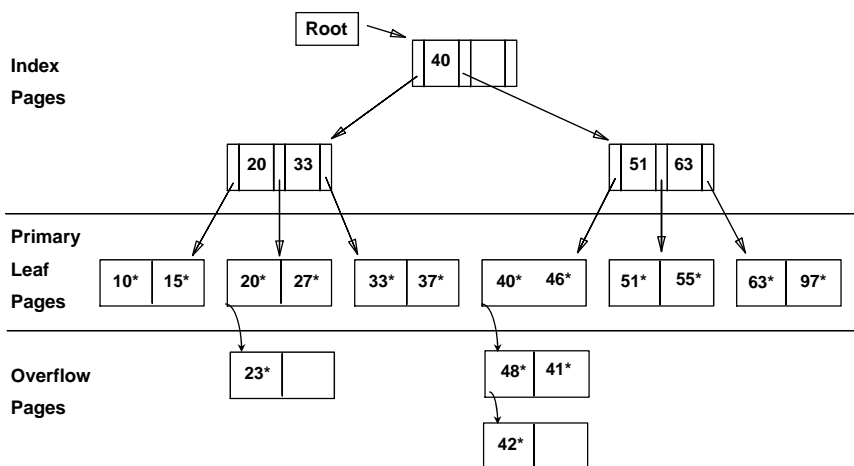
16

After Inserting 23*, 48*, 41*, 42* ...

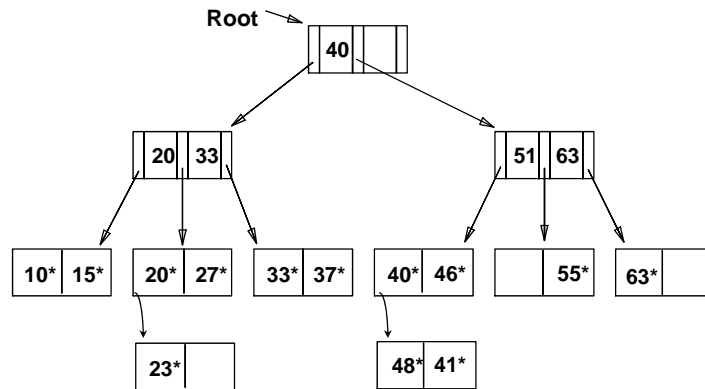


17

... Now Deleting 42*, 51*, 97*



18



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

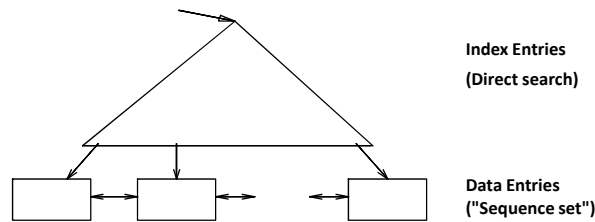
19

Properties of ISAM Tree

- Insertions and deletions affect only the leaf pages, not the non-leaf pages
 - index in the tree is static.
- Static index tree has both advantages & disadvantages.
 - Advantage: [redacted] access.
 - Disadvantage: [redacted] leading to poor [redacted]
- ISAM tree is good when data does not change much.
 - To accommodate some insertions, can leave the primary pages 20% empty.
- How to solve the disadvantage in ISAM tree?
 - B+ tree can support file growth & shrink efficiently, but at the cost of locking overhead.

B+ Tree

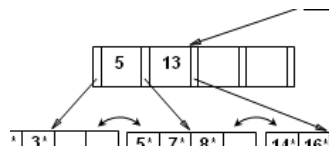
- It is similar to ISAM tree-structure, except:
 - It has no overflow chains (this is the cause of poor performance in ISAM).
 - When an insertion goes to a leaf page becomes full, a new leaf page is created.
 - Leaf pages are not allocated sequentially. Leaf pages are sorted and organized into doubly-linked list.
 - Index pages can grow and shrink with size of data file.



21

Properties of B+ Tree

- Keep tree height-balanced.
 - Balance means that distance from root to all leaf nodes are the same .
- Minimum 50% occupancy (except for root)
 - Each index page node must contain $d \leq m \leq 2d$ entries.
 - The parameter m is the number of occupied entries.
 - The parameter d is called the order of the tree (or $\frac{1}{2}$ node capacity)



22

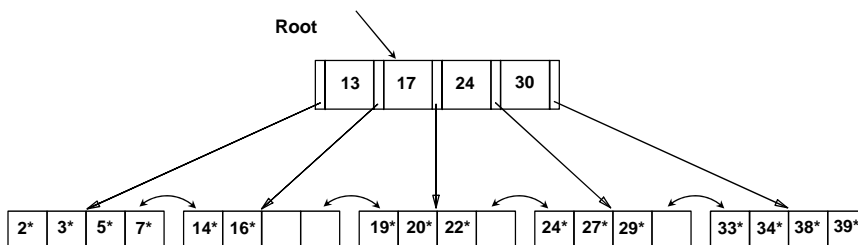
More Properties of B+ Tree

- Cost of search, insert, and delete (disk page I/Os):
 - $\Theta(\text{height of the tree}) = \Theta(\log_{m+1} N)$, where $N = \# \text{ leaf pages}$
- Supports equality and range-searches efficiently.
- B+ tree is the most widely used index.

23

Example B+ Tree

- Search begins at root, and key comparisons direct it to a leaf (same as in ISAM).
- Search for 5^* , 15^* , all data entries $\geq 24^*$...



24

B+ Trees in Practice

- Typical order: 100. Typical fill-factor: 66%.
 - average fanout = 133
- Typical capacities:
 - Height 4: $133^4 = 312,900,700$ records
 - Height 3: $133^3 = 2,352,637$ records
- 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

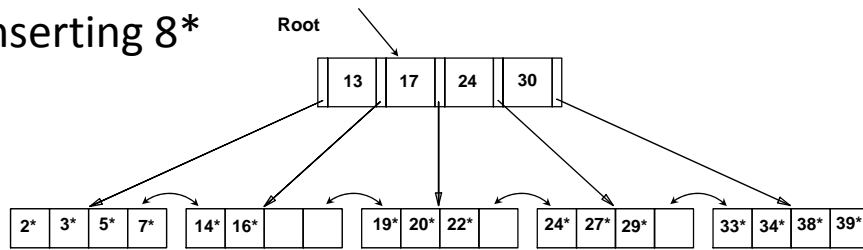
25

Inserting a Data Entry into a B+ Tree

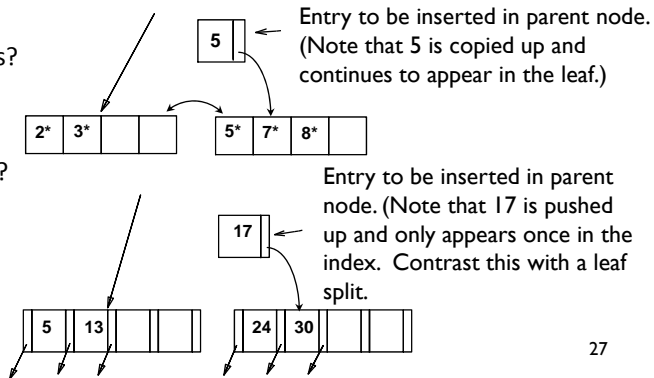
- Find correct leaf L .
- Put data entry onto L .
 - If L has enough space, *done!*
 - Else, must *split* L (into L and a new node $L2$)
 - Redistribute entries evenly, **copy up** middle key.
 - Insert index entry pointing to $L2$ into parent of L .
- This can happen recursively
 - To split index node, 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*.

26

Inserting 8*

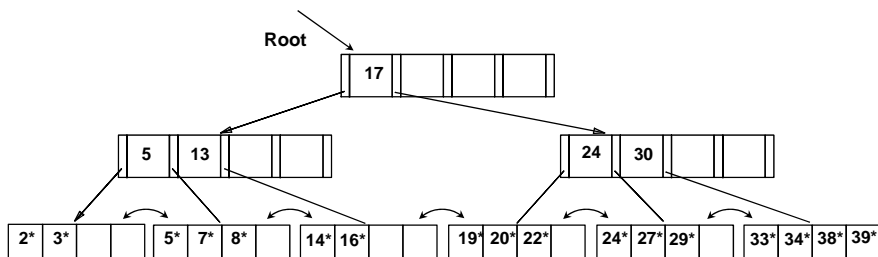


- Why is minimum occupancy always guaranteed in both leaf and index pg splits?
- What is the difference between *copy-up* and *push-up*?
- How to avoid splitting?



27

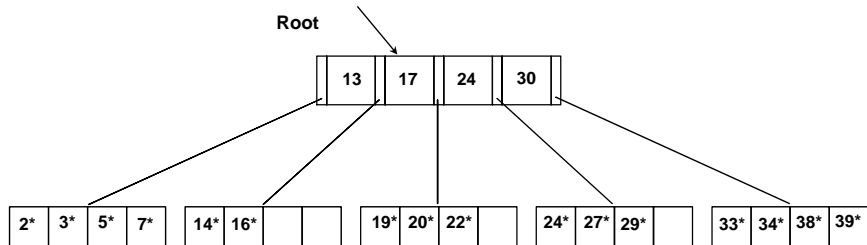
Example B+ Tree After Inserting 8*



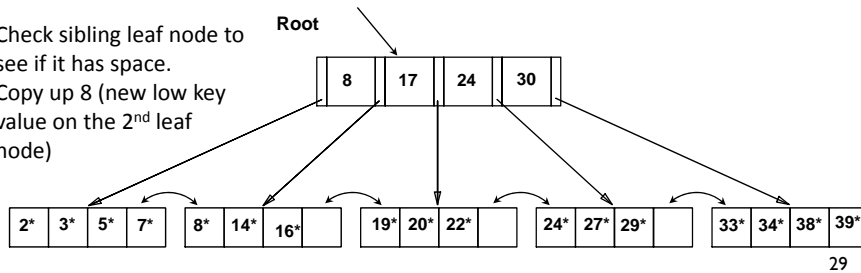
Root was split, leading to increase in height.
Avoid split by re-distributing entries.

28

Redistribution after Inserting 8*



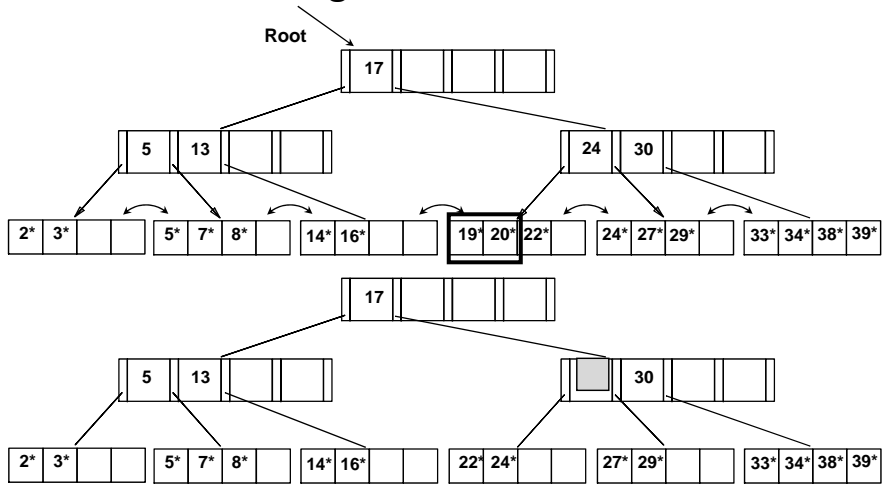
Check sibling leaf node to see if it has space.
Copy up 8 (new low key value on the 2nd leaf node)



Deleting a Data Entry from a B+ Tree

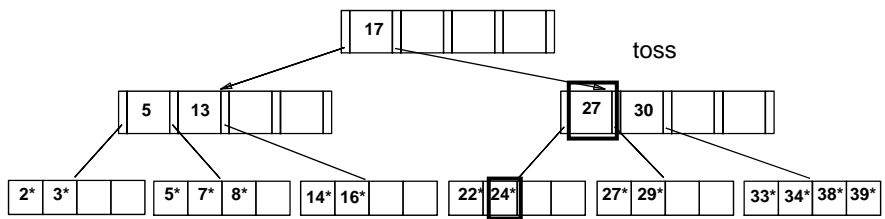
- Start at root, find leaf L where entry belongs.
- Remove the entry.
 - If L is at least half-full, *done!*
 - What if L is less than half-full?
 - Try to re-distribute, borrowing from *sibling (adjacent node with same parent as L)*.
 - What if re-distribution fails?
 - *merge L and sibling.*
- If merge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

Tree After Deleting 19* and 20* ...

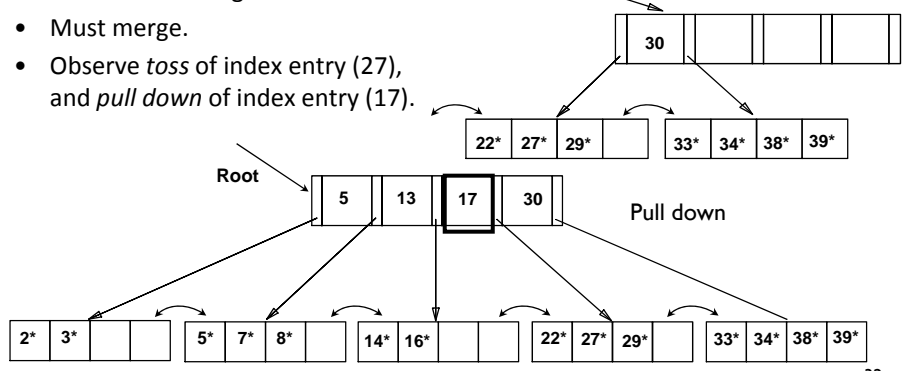


- Deleting 19* is easy.
- Deleting 20* is done with re-distribution. Notice how middle key is *copied up*.

31



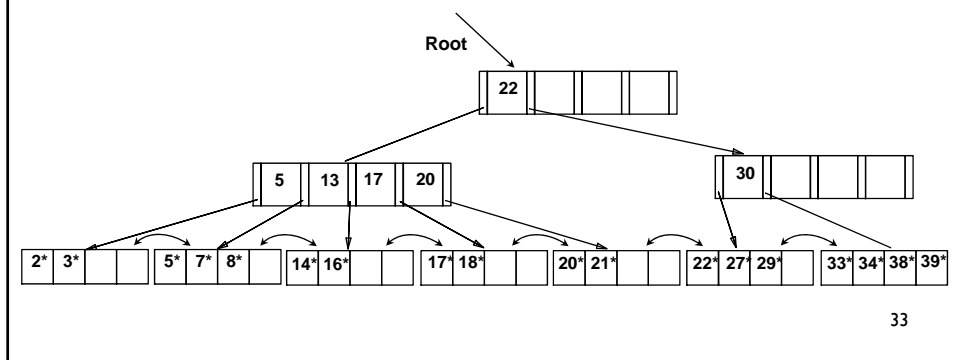
- And then deleting 24*
- Must merge.
- Observe *toss* of index entry (27), and *pull down* of index entry (17).



32

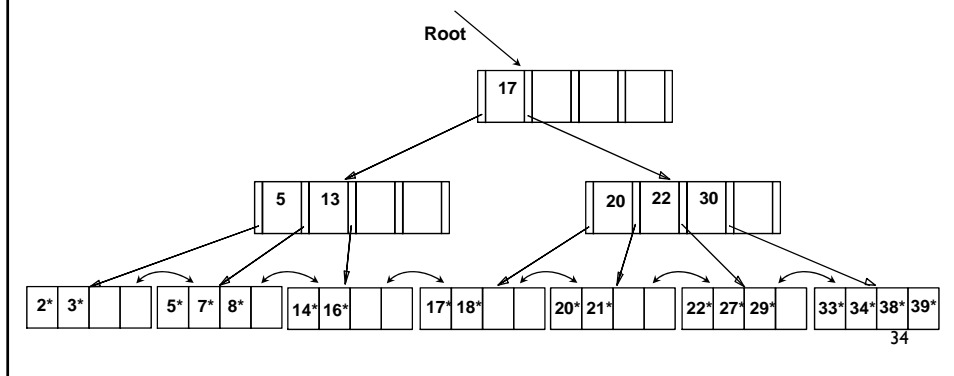
Example of Non-leaf Re-distribution

- Tree is shown below *during deletion* of 24*.
- May re-distribute entry from left child of root to right child.



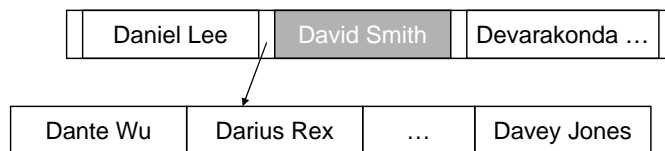
After Re-distribution

- 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.



Prefix Key Compression

- Important to increase fan-out. (Why?)
- Key values in index entries only `direct traffic`; can often compress them.
 - Compress “David Smith” to “Dav”? How about “Davi”?
 - In general, while compressing, must leave each index entry greater than every key value (in any subtree) to its left.



35

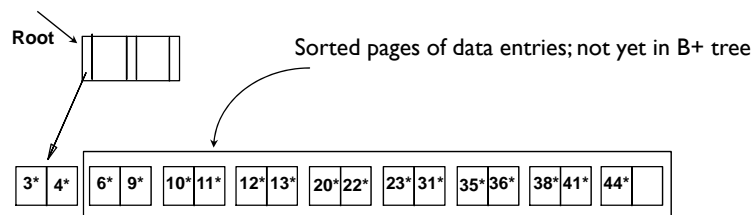
Bulk Loading of a B+ Tree

- Given a large collection of records
- We want to create a B+ tree on some field
- How to do it slowly?
 - Insert each record repeatedly. What is the cost of building index?
 - # entries * $\log_F(N)$, where F = fan-out, N = # index pages
- How to do it quickly?
 - Insert at the unit of a page.

36

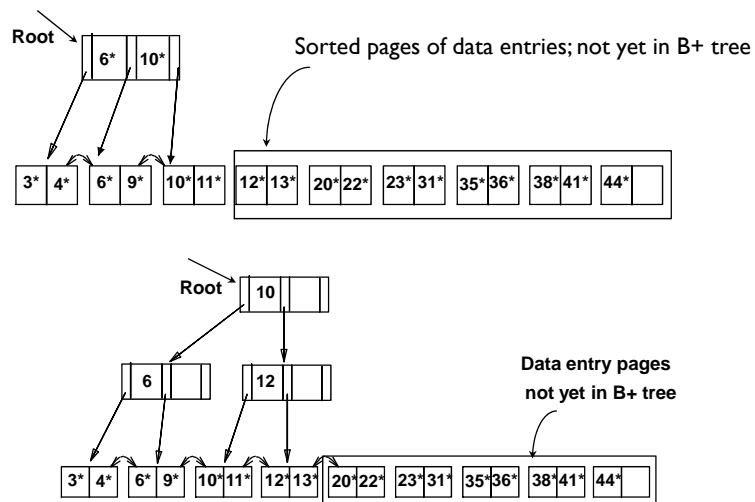
Bulk Loading of a B+ Tree

- Bulk Loading can be done much more efficiently.
 - Add one page at a time (rather than one record at a time).
 - Step 1: Sort data entries. Insert pointer to first (leaf) page in a new (root) page.



37

Bulk Loading(Conti)



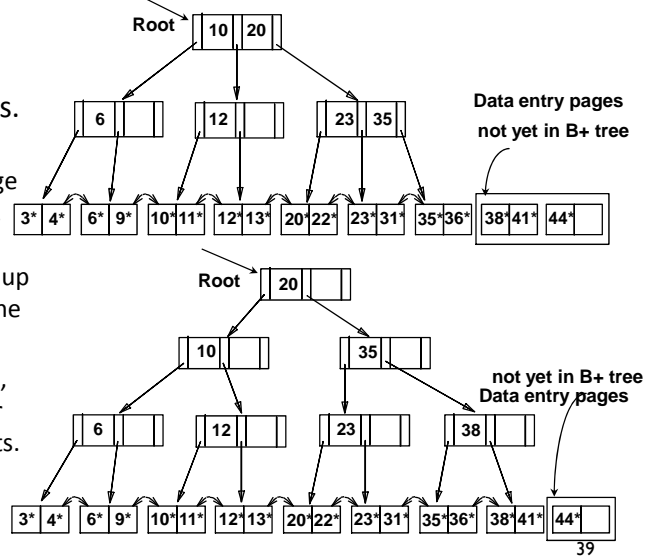
38

Bulk Loading (Contd.)

- Step 2: Build Index entries for leaf pages.

- Always entered into right-most index page just above leaf level. When this fills up, it splits. (Split may go up right-most path to the root.)

- Cost = # index pages, which is much faster than repeated inserts.



Summary of Bulk Loading

- Option 1: multiple inserts.
 - More I/Os during build.
 - Does not give sequential storage of leaves.
- Option 2: Bulk Loading
 - Fewer I/Os during build.
 - Leaves will be stored sequentially (and linked, of course).
 - Can control “fill factor” on pages.

A Note on 'Order'

- *Order* (the parameter **d**) concept denote minimum occupancy on the number of entries per index page.
 - Not practical in real implementation. Why not?
 - 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.
- Order is replaced by physical space criterion (*'at least half-full'*).

41

Review: Three Alternatives for Data Entries

- As for any index, 3 alternatives for data entries **k***:
 - (1) Clustered Index: data entry is data record (sorted by **k**)
 - (2) Unclustered Index: data entry separated from data record: $\langle \mathbf{k}, \text{rid of data record with search key value } \mathbf{k} \rangle$
 - (3) Unclustered Index: data entry separated from data record: $\langle \mathbf{k}, \text{list of rids of data records with search key } \mathbf{k} \rangle$, useful when search key is not unique (not a candidate key).
- What is the performance (cost of retrieving records) difference between clustered and un-clustered index?

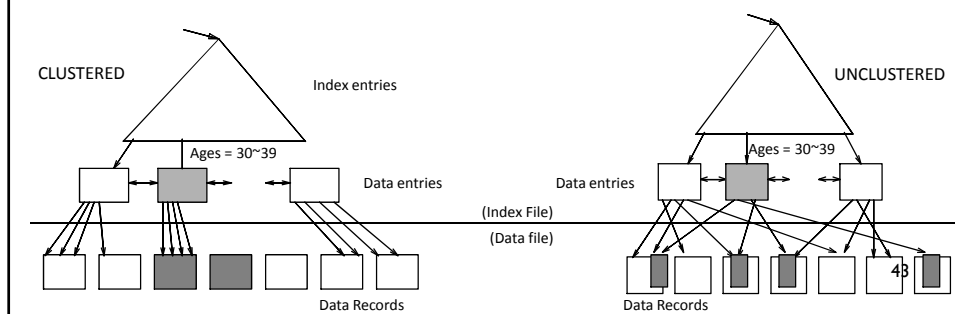
42

Review: Clustered vs. Unclustered Index

- Examples: retrieve all the employees of ages 30~39.
 - What is the worst-case cost (# disk page I/Os) of clustered index?
 - What is the worst-case cost of unclustered index?

Cost = number of pages retrieved

mr = # matched records; mp = # pages containing matched records

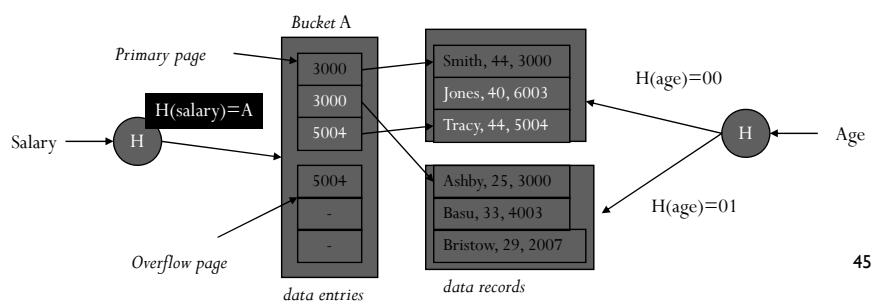


Review: Two General indexing techniques

- Hash-structured indexing or tree-structured indexing
- Briefly describe them and ask you to compare them on cost of
 - Equality search: find all employees of age 30.
 - Range search: find all the employees of ages 30~39.

Hash-Based Indexes

- Data entries (key, rid) are grouped into buckets.
- Bucket = *primary* page plus zero or more *overflow* pages.
- *Hashing function h*: $h(r)$ = bucket in which record r belongs. h looks at the *search key* fields of r .
- If Alternative (1) is used, the buckets contain the data records.



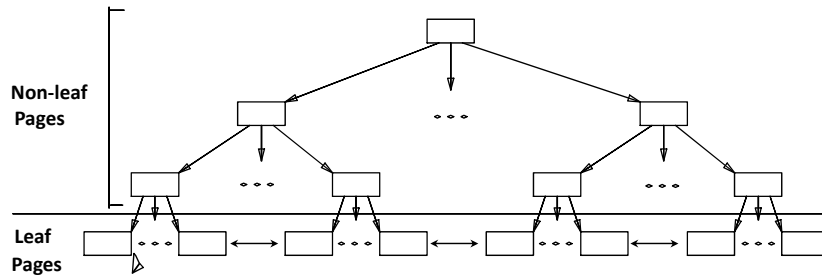
45

Hash-based Indexes (Cont)

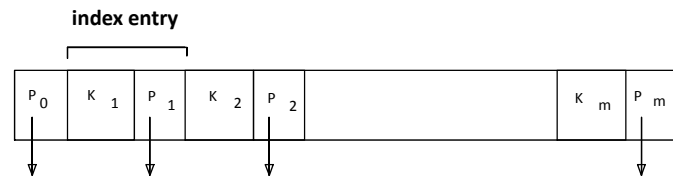
- Search on key value:
 - Apply key value to the hash function \rightarrow bucket number
 - Retrieve the primary page of the bucket. Search records in the primary page. If not found, search the overflow pages.
 - Cost of locating rids: # pages in bucket (small)
- Insert a record:
 - Apply key value to the hash function \rightarrow bucket number
 - If all (primary & overflow) pages in that bucket are full, allocate a new overflow page.
 - Cost: similar to search.
- Delete a record
 - Cost: similar to search.

46

B+ Tree Indexes

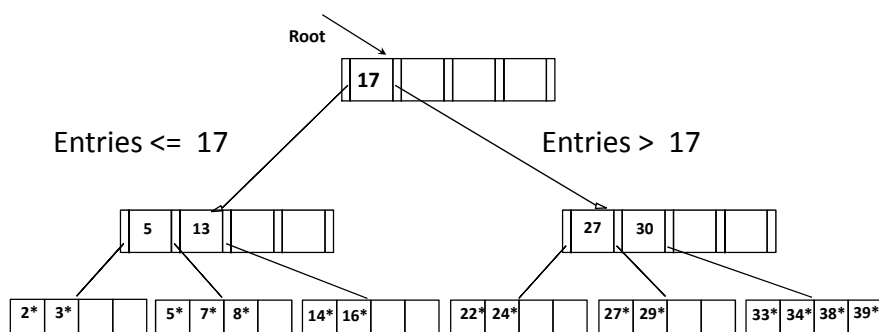


Leaf pages contain data entries, and are chained (prev & next)
 Non-leaf pages contain index entries and direct searches:



47

Example B+ Tree



- Find 7*, 29*? 15* < age < 30*
- Insert/delete: Find data entry in leaf, then change it. Need to adjust parent sometimes.
 - And change sometimes bubbles up the tree (keep the tree balance)
- More details about tree-based index in Chapter 10.

48

Tree vs. Hash-Structured Indexing

- Tree index supports both *range searches* and *equality searches* efficiently.
 - Why efficient range searches?
 - Data entries (on the leaf nodes of the tree) are sorted.
 - Perform equality search on the first qualifying data entry + scan to find the rests.
 - Data records also need to be sorted by search key in case that the range searches access record fields other than the search key.
- Hash index supports equality search efficiently, but not range search.
 - Why inefficient range searches?
 - Data entries are hashed (using a hash function), not sorted.

49