Database Systems
(資料庫系統)

November 29/30, 2006

Lecture #8

1

Announcement

- Next week reading: Chapter 11 Hash-based Index
- Assignment #3 is due next week.
- Midterm exams are graded.

2

Tree-Structured Indexing

Chapter 10

3

Outline

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

4

Review: Three Alternatives for Data Entries

- As for any index, 3 alternatives for data entries **k***:

    (1) Clustered Index: Data record with key value **k**

    (2) Unclustered Index: <**k**, rid of data record with search key value **k**>

    (3) Unclustered Index: <**k**, list of rids of data records with search key **k**>, useful when search key is not unique (not a candidate key).

5

Review: Three Alternatives for Data Entries

- Choice of data entries is independent to the indexing technique used to locate data entries **k***.

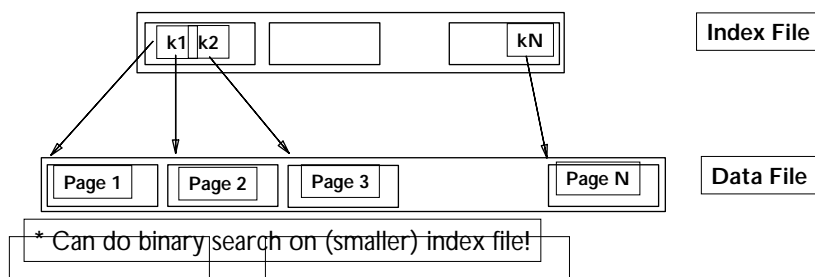   – Two general indexing techniques: hash-structured indexing or tree-structured indexing

6

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.

7

Why [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.

| k1 | k2 | | kN | | **Index File** |

| Page 1 | Page 2 | Page 3 | Page N | | **Data File** |

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

8

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.
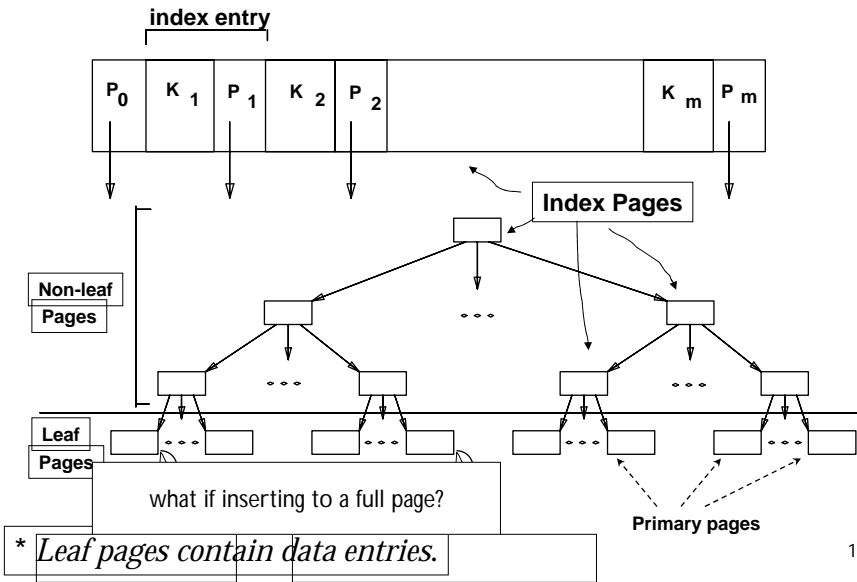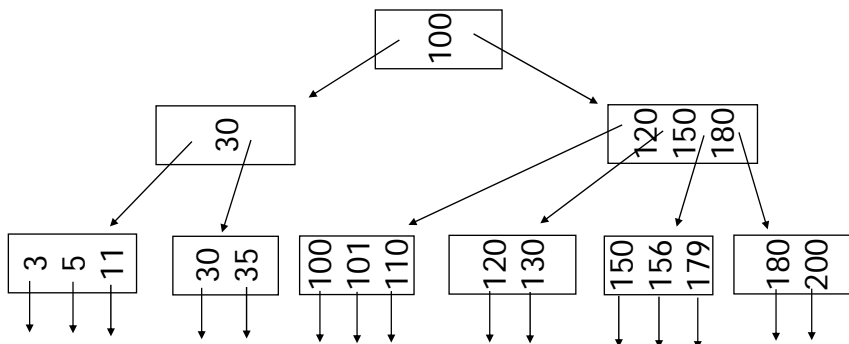 - The cost of search is [____] page access.

9

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
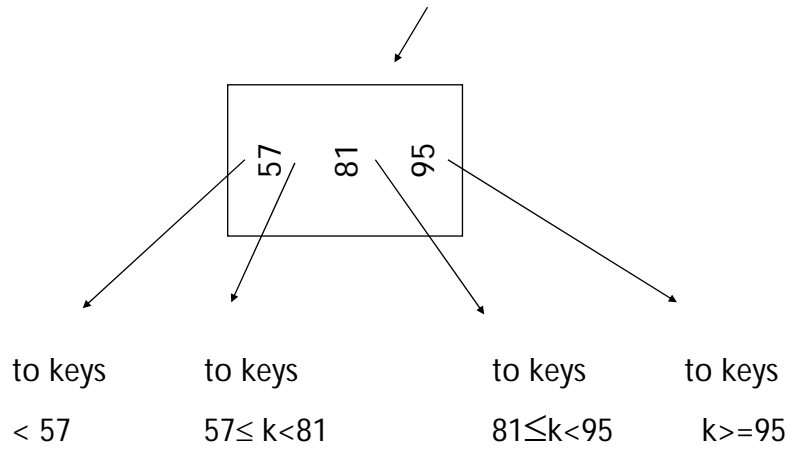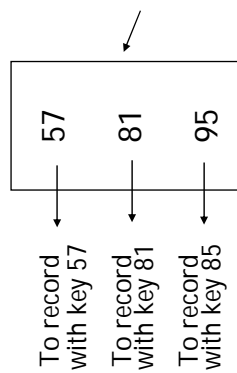 - Concurrent access.

10

ISAM

index entry

P_0	K_1	P_1	K_2	P_2			K_m	P_m

Index Pages

Non-leaf Pages

Leaf Pages

what if inserting to a full page?

Primary pages

* *Leaf pages contain data entries.*

11

Example

```
                        100

           30                        120 150 180

    3  5  11    30 35    100 101 110   120 130   150 156 179   180 200
```

12

Non-leaf node

57	81	95

to keys < 57

to keys 57≤ k<81

to keys 81≤k<95

to keys k>=95

13

Leaf node

57	81	95

To record with key 57

To record with key 81

To record with key 85

14

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.
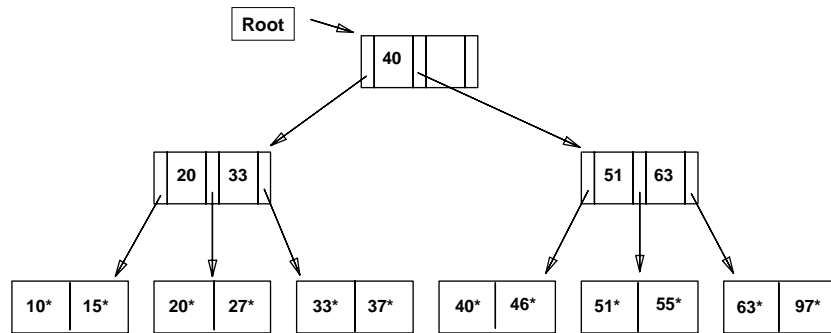
15

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)
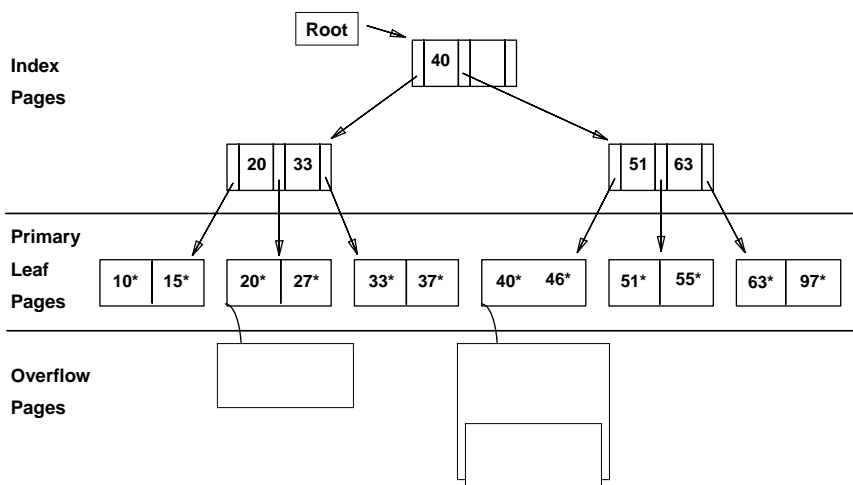
16

Example ISAM Tree

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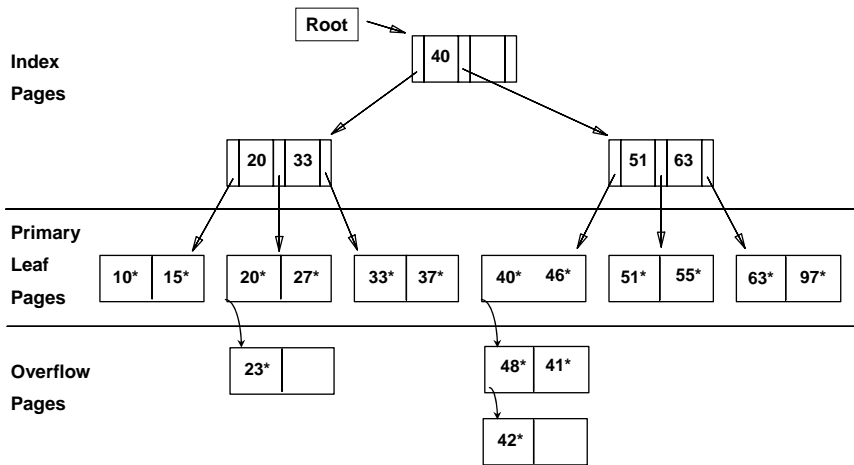 - Primary pages are allocated sequentially at file creation time.

Root

| 40 | |

| 20 | 33 |

| 51 | 63 |

| 10* | 15* | | 20* | 27* | | 33* | 37* | | 40* | 46* | | 51* | 55* | | 63* | 97* |

17

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

Root

Index Pages

| 40 | |

| 20 | 33 |

| 51 | 63 |

Primary Leaf Pages

| 10* | 15* | | 20* | 27* | | 33* | 37* | | 40* | 46* | | 51* | 55* | | 63* | 97* |

Overflow Pages

18

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

Root

Index Pages

```
40
```

```
20  33
```
```
51  63
```

Primary Leaf Pages

```
10*  15*    20*  27*    33*  37*    40*  46*    51*  55*    63*  97*
```

Overflow Pages

```
23*        48*  41*
```

```
42*
```

19

Root

```
40
```

```
20  33
```
```
51  63
```

```
10*  15*    20*  27*    33*  37*    40*  46*        55*    63*
```

```
23*        48*  41*
```
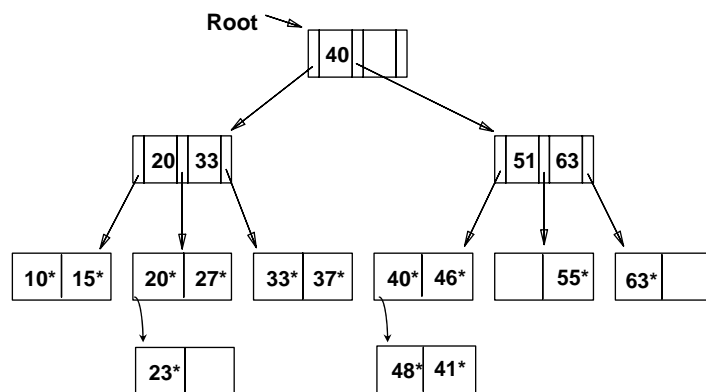
** Note that 51* appears in index levels, but not in leaf!*

20

10

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: [   ] concurrent access.
  - Disadvantage: when a file grows [   ] leading to poor performance.
- ISAM tree is good when data does not change much.
  - To accommodate some insertions, can leave the primarily pages 20% empty.
- B+ tree can support file growth & shrink efficiently, but at the cost of locking overhead.

21

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.

**Index Entries
(Direct search)**

**Data Entries
("Sequence set")**
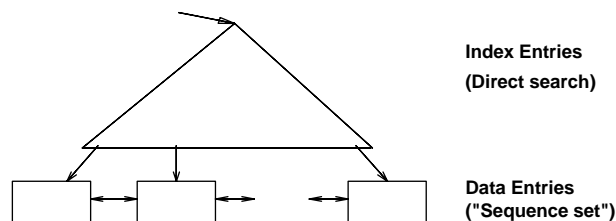
22

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 <= m <= 2d entries.
  - The parameter **m** is the number of occupied entries.
  - The parameter **d** is called the order of the tree (or ½ node capacity)
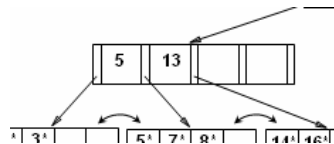
23

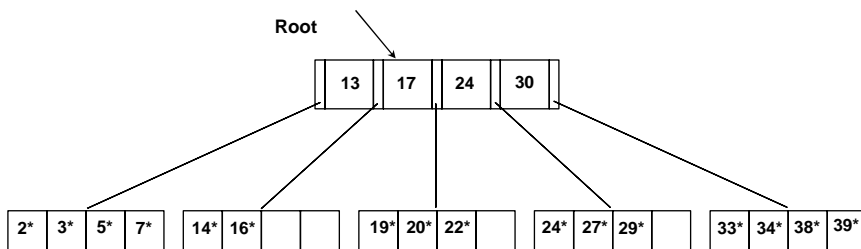# More Properties of B+ Tree

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

24

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 >= 24* ...

Root

13	17	24	30

| 2* | 3* | 5* | 7* | | 14* | 16* | | | | 19* | 20* | 22* | | | 24* | 27* | 29* | | | 33* | 34* | 38* | 39* |

25

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
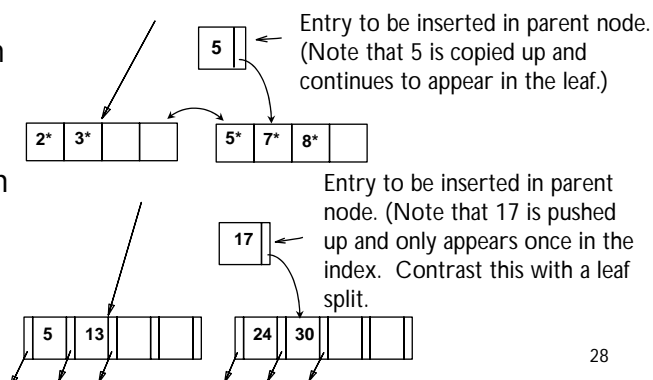
26

13

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.*

27

Inserting 8*

Root

| 13 | 17 | 24 | 30 |

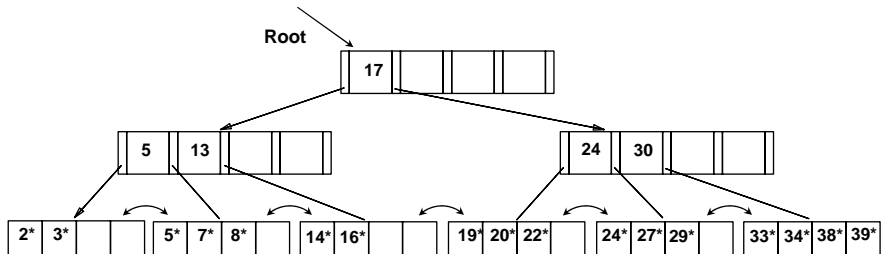| 2* | 3* | 5* | 7* | 14* | 16* | | | 19* | 20* | 22* | | 24* | 27* | 29* | | 33* | 34* | 38* | 39* |
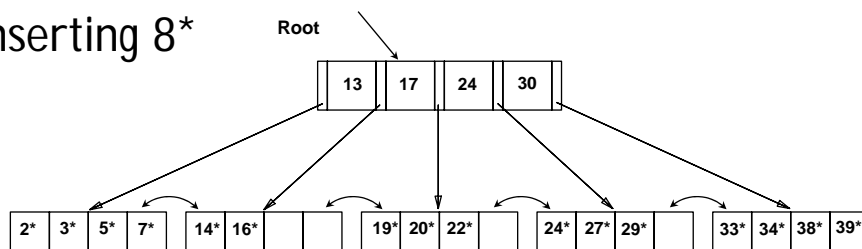
- Why is minimum occupancy always guaranteed in both leaf and index pg splits?
- What is the difference between *copy-up* and *push-up?*
- How to avoid splitting?

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* | |

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 | | | |

28

14

Example B+ Tree After Inserting 8*

Root

17

5 13

24 30

2* 3* | 5* 7* 8* | 14* 16* | 19* 20* 22* | 24* 27* 29* | 33* 34* 38* 39*
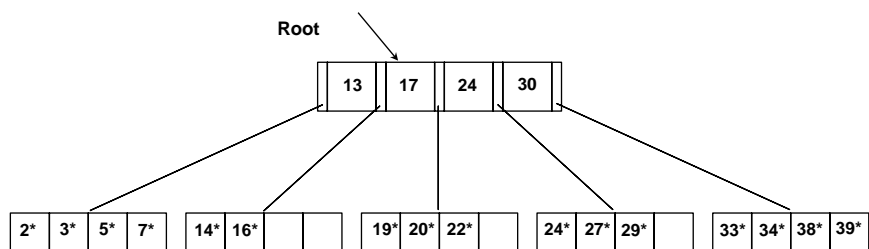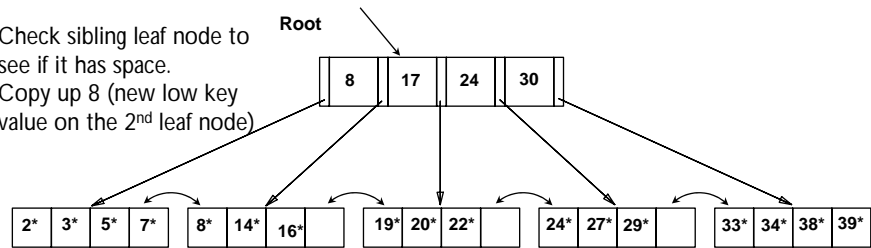
Root was split, leading to increase in height.
Avoid split by re-distributing entries.

29

Redistribution after Inserting 8*

Root

13 17 24 30

2* 3* 5* 7* | 14* 16* | 19* 20* 22* | 24* 27* 29* | 33* 34* 38* 39*

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

Root

8 17 24 30

2* 3* 5* 7* | 8* 14* 16* | 19* 20* 22* | 24* 27* 29* | 33* 34* 38* 39*

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!*
  - 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*.
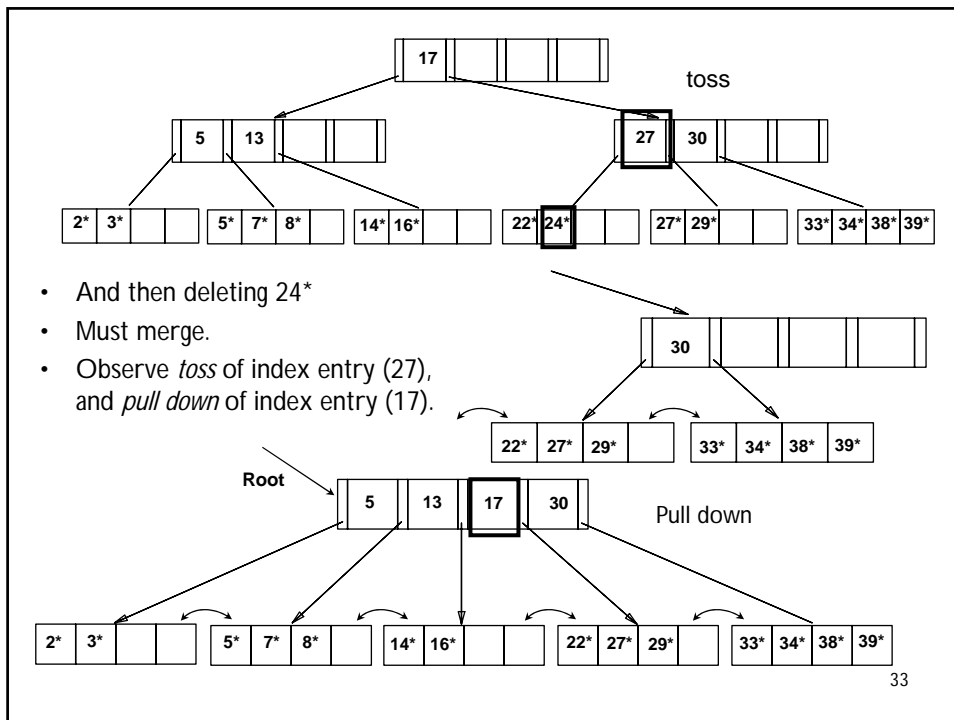- Merge could propagate to root, decreasing height.

31

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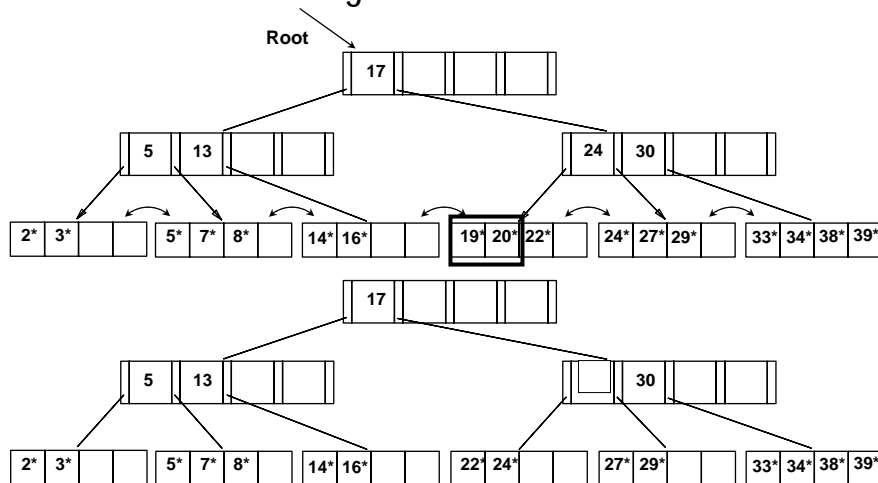
- Deleting 19* is easy.
- Deleting 20* is done with re-distribution. Notice how middle key is *copied up*.

32

17

toss

5 | **13**

27 | **30**

2* 3* | 5* 7* 8* | 14* 16* | 22* 24* | 27* 29* | 33* 34* 38* 39*

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

30

22* 27* 29* | 33* 34* 38* 39*

Root

5 | **13** | **17** | **30**   Pull down

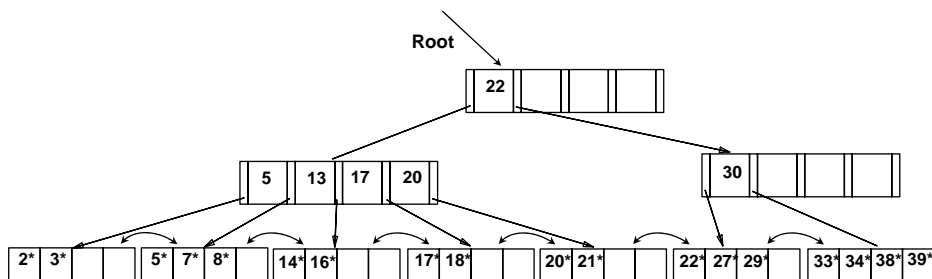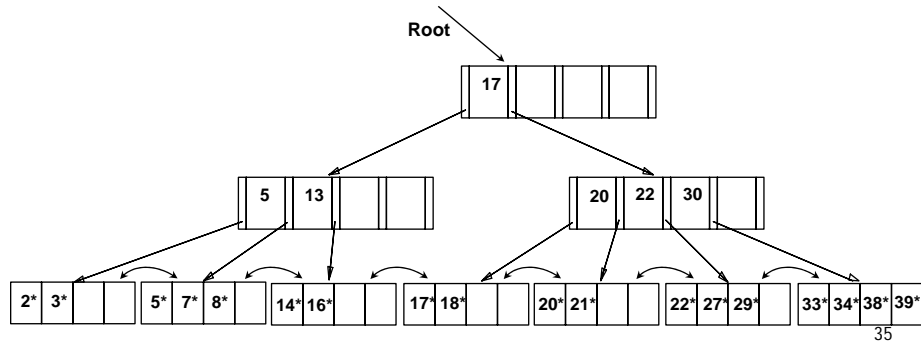2* 3* | 5* 7* 8* | 14* 16* | 22* 27* 29* | 33* 34* 38* 39*

33

Example of Non-leaf Re-distribution

- Tree is shown below *during deletion* of 24*. (What could be a possible initial tree?)
- May re-distribute entry from left child of root to right child.

Root

22

5 | **13** | **17** | **20**

30

2* 3* | 5* 7* 8* | 14* 16* | 17* 18* | 20* 21* | 22* 27* 29* | 33* 34* 38* 39*

34

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.

Root

| 17 | | | |

| 5 | 13 | | |

| 20 | 22 | 30 | |

| 2* | 3* | | | 5* | 7* | 8* | | 14* | 16* | | | 17* | 18* | | | 20* | 21* | | 22* | 27* | 29* | | 33* | 34* | 38* | 39* |

35

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.

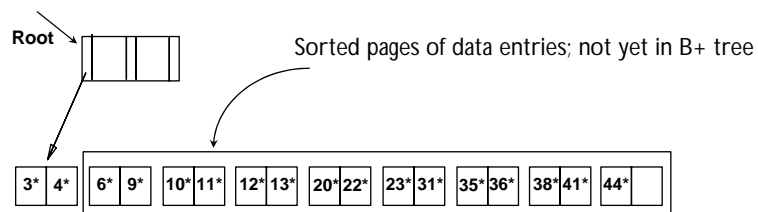| | Daniel Lee | | David Smith | | Devarakonda ... | |

| Dante Wu | Darius Rex | | ... | | Davey Jones |
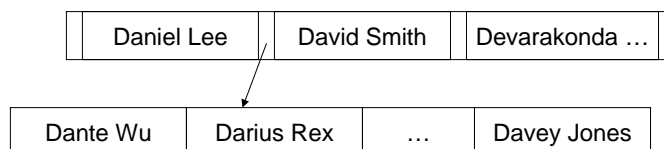
36

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?
  - Repeatedly insertions. What is the cost of building index?
    - # entries * $\log_F(N)$, where F = fan-out, N = # index pages
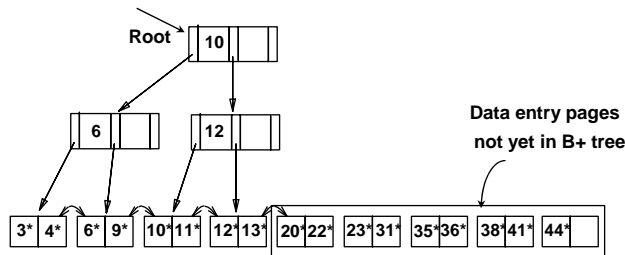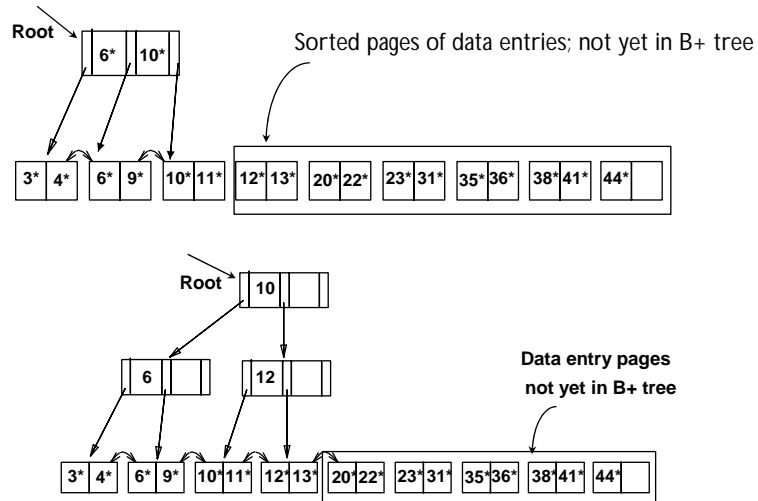- How to do it quickly?

Bulk Loading of a B+ Tree

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

Root

Sorted pages of data entries; not yet in B+ tree

| 3* | 4* | | 6* | 9* | | 10* | 11* | | 12* | 13* | | 20* | 22* | | 23* | 31* | | 35* | 36* | | 38* | 41* | | 44* |

Bulk Loading(Conti)

Root → | 6* | 10* |

Sorted pages of data entries; not yet in B+ tree

| 3* | 4* | | 6* | 9* | | 10* | 11* | | 12* | 13* | | 20* | 22* | | 23* | 31* | | 35* | 36* | | 38* | 41* | | 44* | |

Root → | 10 | |

| 6 | | | | 12 | |

Data entry pages
not yet in B+ tree

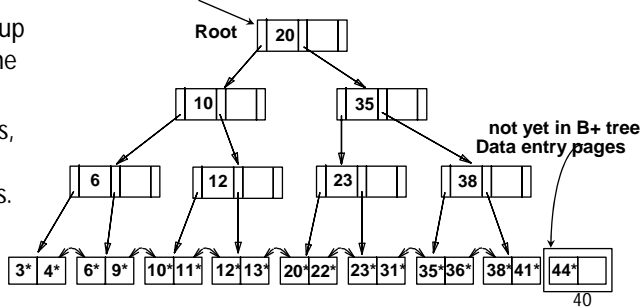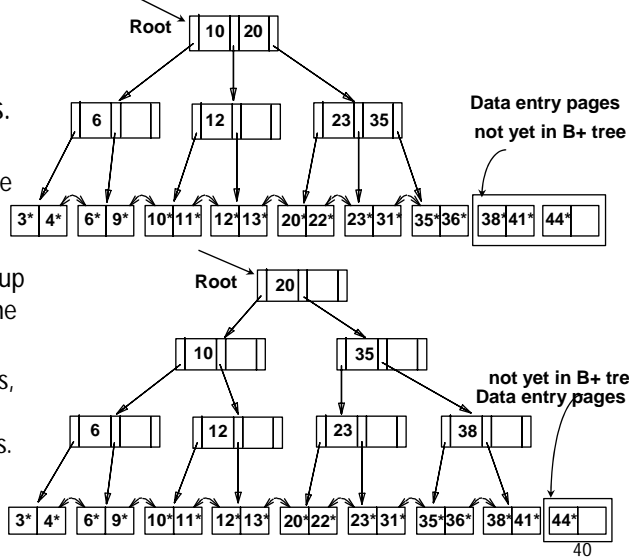| 3* | 4* | | 6* | 9* | | 10* | 11* | | 12* | 13* | | 20* | 22* | | 23* | 31* | | 35* | 36* | | 38* | 41* | | 44* |

39

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.

Root → | 10 | 20 |

| 6 | | | | 12 | | | | 23 | 35 |

Data entry pages
not yet in B+ tree

| 3* | 4* | | 6* | 9* | | 10* | 11* | | 12* | 13* | | 20* | 22* | | 23* | 31* | | 35* | 36* | | 38* | 41* | | 44* | |

Root → | 20 | |

| 10 | | | | 35 | |

| 6 | | | | 12 | | | | 23 | | | | 38 | |

not yet in B+ tree
Data entry pages

| 3* | 4* | | 6* | 9* | | 10* | 11* | | 12* | 13* | | 20* | 22* | | 23* | 31* | | 35* | 36* | | 38* | 41* | | 44* |

40

20

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.
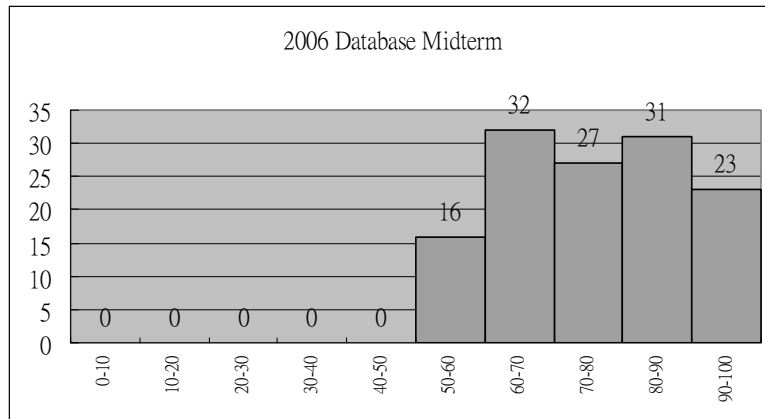
A Note on `Order'

- *Order* (the parameter **d**) concept denote minimum occupancy on the number of entries per index page.
  - But it is not practical in real implementation. Why?
    - 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').

Midterm Exam

Average = 76, Std = 13

2006 Database Midterm

43