# Tree-Structured Indexes



## Introduction

- *As for any index, 3 alternatives for data entries* **k***:
  - Data record with key value **k**
  - <**k**, rid of data record with search key value **k**>
  - <**k**, list of rids of data records with search key **k**>
- Choice is orthogonal to the *indexing technique* used to locate data entries **k***.
- Tree-structured indexing techniques support both *range searches* and *equality searches*.
- *ISAM*: static structure; *B+ tree*: dynamic, adjusts gracefully under inserts and deletes.



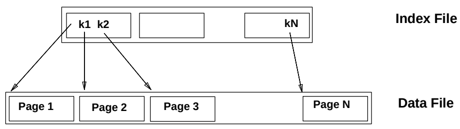
## Range Searches

- *``Find all students with gpa > 3.0''*
  - If data *entries* are sorted, do binary search to find first such student, then scan to find others.
- Problem?




## Range Searches

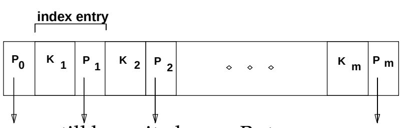
- Simple idea: Create an `index' file
  - What is search cost if each index page has F entries?



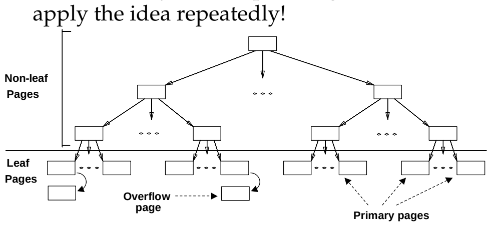

*Can do binary search on (smaller) index file!*



## ISAM




- Index file may still be quite large. But we can apply the idea repeatedly!

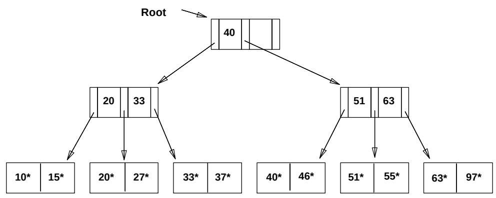



*Leaf pages contain data entries.*

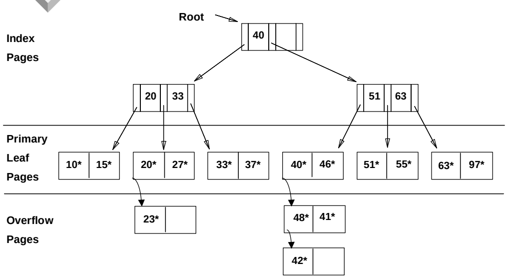


## Example ISAM Tree

- Each node can hold 2 entries
  - What is search cost if each leaf node can hold L entries and each index node can hold F entries?

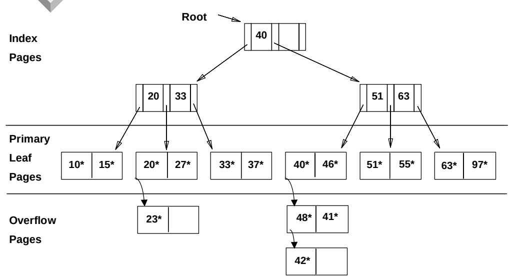


## After Inserting 23*, 48*, 41*, 42* ...



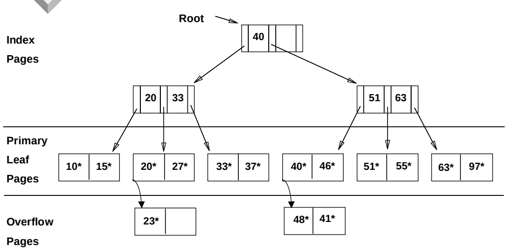



## … Then Deleting 42*



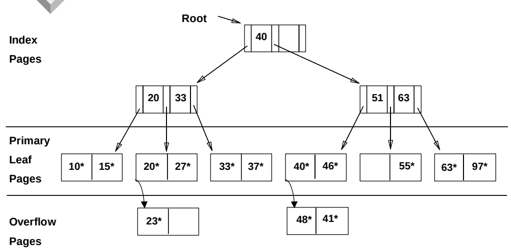



## … Then Deleting 51*






## After Deleting 41* and 51*

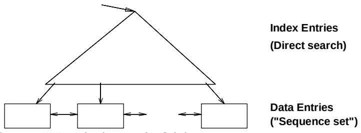



*Note 51 appears in Index Page but not in Leaf pages!*



## B+ Tree: The Most Widely Used Index

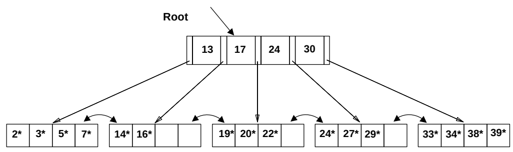
- Insert/delete at log $_F$ N cost; keep tree *height-balanced*. (F = fanout, N = # leaf pages)
- Minimum 50% occupancy (except for root). Each node contains **d** <= *m* <= 2**d** entries. The parameter **d** is called the *order* of the tree.
- Supports equality and range-searches efficiently.






## Example B+ Tree

- Search begins at root, 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 <u>know</u> it is not in the tree!*

## B+-tree Search Performance

- Assume leaf pages can hold L data entries
- Assume B+-tree has order d
- Assume the tree has to index N data entries

- What is the best-case search performance (measured in number of I/Os)?
- What is the worst-case search performance

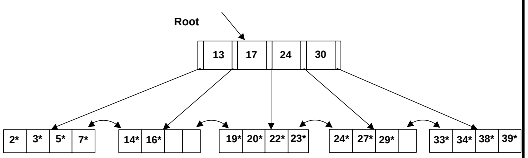


## B+ Trees in Practice

- Typical order: 100. Typical fill-factor: 67%.
  - 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

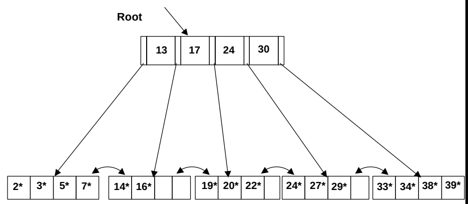


## Inserting 23*



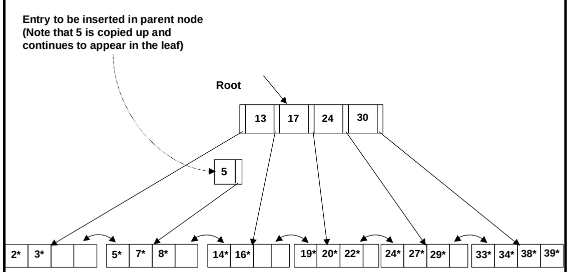



## Inserting 8* …



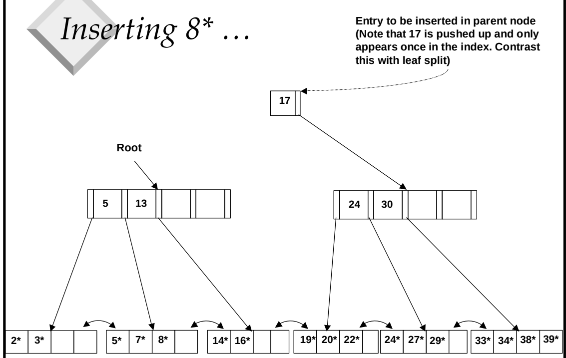



## Inserting 8* …

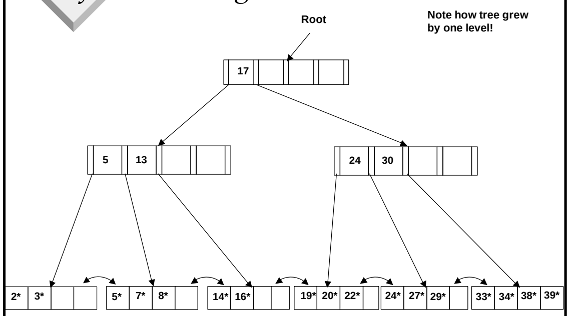





## Inserting 8* …

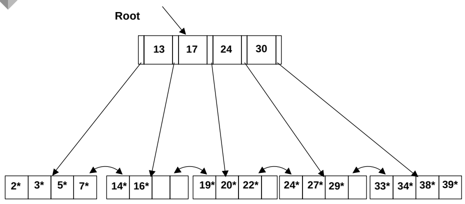


# After Inserting 8*





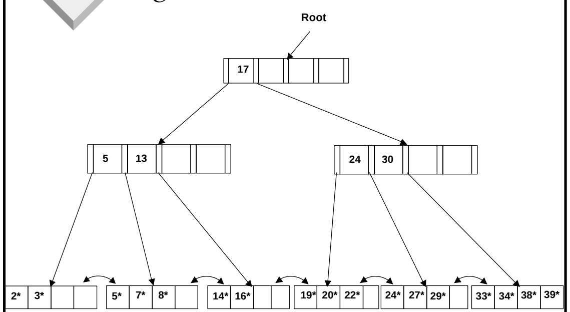

# Inserting 8* …




- In this example, could have “redistributed” to sibling instead of splitting
- Not usually done in practice (Why?)

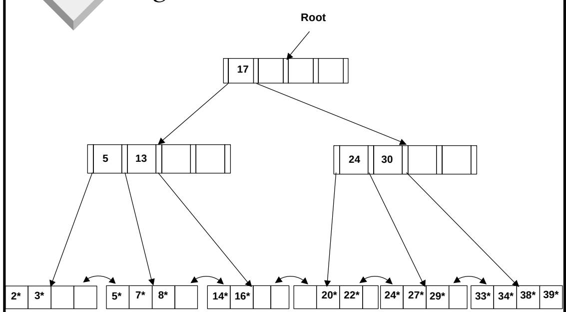


# Deleting 19* …



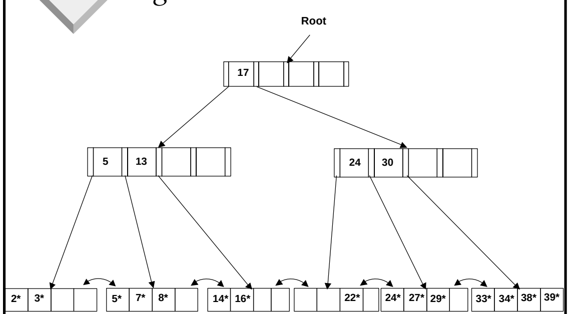



# Deleting 20* …



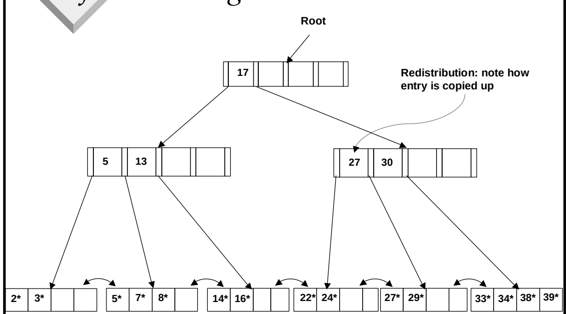



# Deleting 20* …

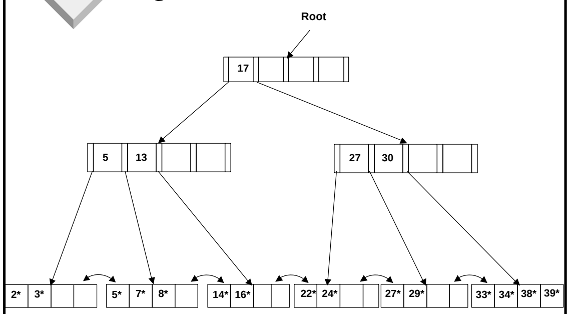





# After Deleting 20*

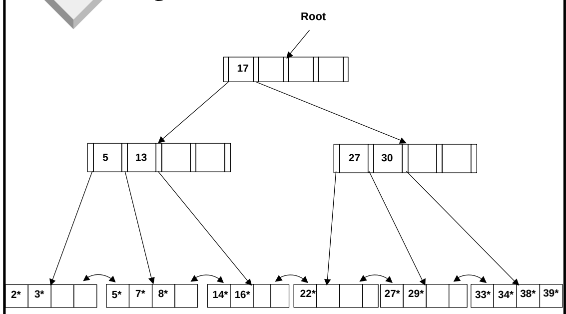


## Deleting 24* …



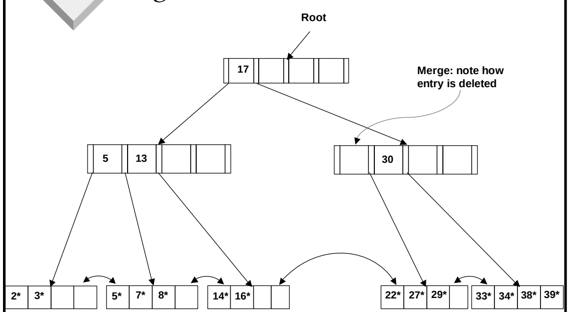



## Deleting 24* …



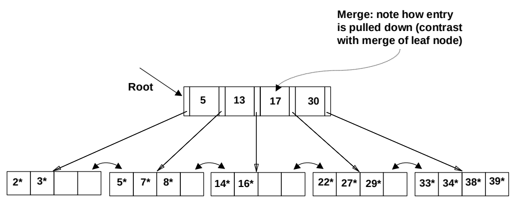



## Deleting 24* …





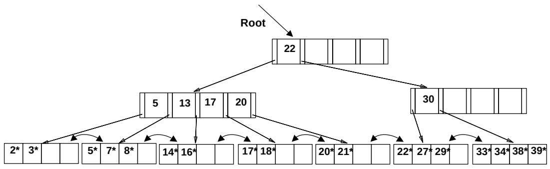

## Deleting 24* …






## Example of Non-leaf Re-distribution

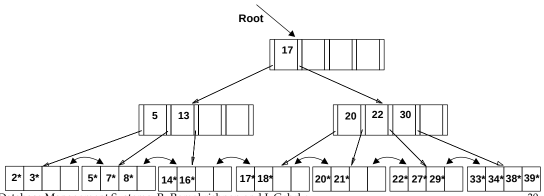
- *During* deletion of 24*
- In contrast to previous example, can re-distribute entry from left child of root to right child.






## After Re-distribution

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



## Composite Search Keys

- B+-tree index on (Age, Salary)
- Which queries can you answer efficiently using a B+-tree?
  - Age = 20, Salary = 100000
  - Age > 20, Salary = 100000
  - Age = 20, Salary > 100000
  - Age > 20, Salary > 100000



## Prefix Key Compression

- Important to increase fan-out (Why?)
- Key values in index entries only `direct traffic'; can often compress them
  - E.g., adjacent index entries with search key values *Dannon Yogurt*, *David Smith* and *Devarakonda Murthy*
  - We can abbreviate *David Smith* to *Dav*. (The other keys can be compressed too ...)
- Is this correct?
  - Not quite! What if there is a data entry *Davey Jones*?
  - Compressed key should be greater than every entry in left sub-tree
  - Insert/delete modified appropriately



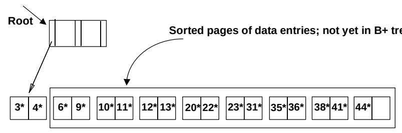
## A Note on `Order'

- *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)).



## Bulk Loading of a B+ Tree

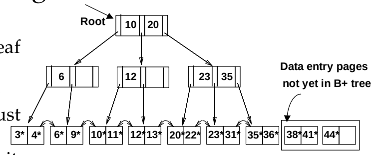
- If we have a large collection of records, and we want to create a B+ tree on some field, doing so by repeatedly inserting records is very slow.
- <u>Bulk Loading</u> can be done much more efficiently.
- *Initialization*: Sort all data entries, insert pointer to first (leaf) page in a new (root) page.

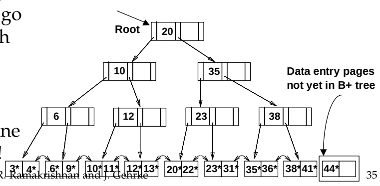





## Bulk Loading (Contd.)

- 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.)
- Much faster than repeated inserts, especially when one considers locking!






## Summary of Bulk Loading

- Option 1: multiple inserts.
  - Slow.
  - Does not give sequential storage of leaves.
- Option 2: <u>*Bulk Loading*</u>
  - Has advantages for concurrency control.
  - Fewer I/Os during build.
  - Leaves will be stored sequentially (and linked, of course).
  - Can control "fill factor" on pages.