Tree-Structured Indexes

Chapter 10

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.
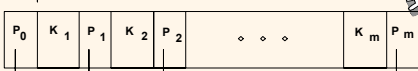
Range Searches

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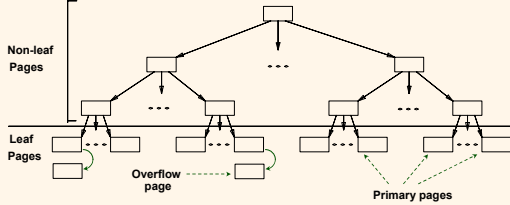- ❖ Simple idea: Create an `index' file.

Can do binary search on (smaller) index file!

ISAM

index entry

| P$_0$ | K$_1$ | P$_1$ | K$_2$ | P$_2$ | ◇ ◇ ◇ | K$_m$ | P$_m$ |

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

Non-leaf Pages

Leaf Pages

Overflow page

Primary pages

✉ *Leaf pages contain data entries.*

Comments on ISAM

Data Pages

Index Pages

Overflow pages

❖ *File creation*: Leaf (data) pages allocated sequentially, sorted by search key; then index pages allocated, then space for overflow pages.
❖ *Index entries*: <search key value, page id>; they `direct' search for *data entries*, which are in leaf pages.
❖ <u>*Search*</u>: Start at root; use key comparisons to go to leaf. Cost ∝ log $_F$ N ; F = # entries/index pg, N = # leaf pgs
❖ <u>*Insert*</u>: Find leaf data entry belongs to, and put it there.
❖ <u>*Delete*</u>: Find and remove from leaf; if empty overflow page, de-allocate.
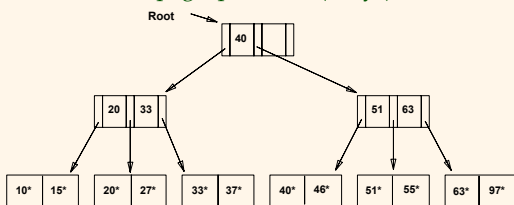
✉ **Static tree structure**: *inserts/deletes affect only leaf pages.*

Example ISAM Tree

❖ Each node can hold 2 entries; no need for `next-leaf-page' pointers. (Why?)

Root

| 40 | |

| 20 | 33 |

| 51 | 63 |

| 10* | 15* |

| 20* | 27* |

| 33* | 37* |

| 40* | 46* |

| 51* | 55* |

| 63* | 97* |

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

| 23* |

| 48* | 41* |

| 42* |

... Then Deleting 42*, 51*, 97*

Root

| 40 |

| 20 | 33 | | 51 | 63 |

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

| 23* |

| 48* | 41* |
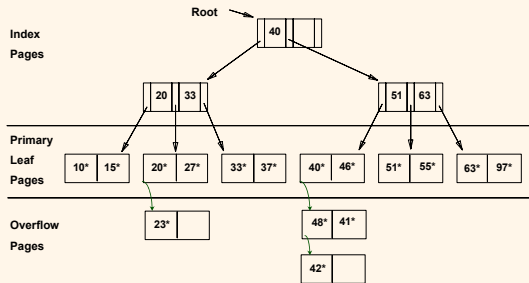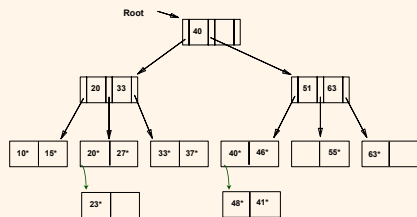
⊠ *Note that 51* appears in index levels, but not in leaf!*
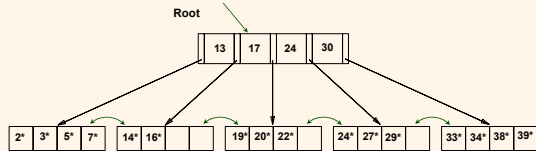

B+ Tree: Most Widely Used Index

❖ 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** <= $\underline{m}$ <= 2**d** entries. The parameter **d** is called the *order* of the tree.
❖ Supports equality and range-searches efficiently.

Index Entries
(Direct search)

Data Entries
("Sequence set")
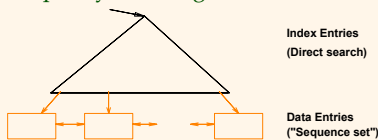
Example B+ Tree

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

Root

13	17	24	30

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

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

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
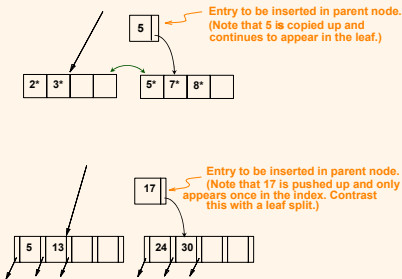 - Level 3 = 17,689 pages = 133 MBytes
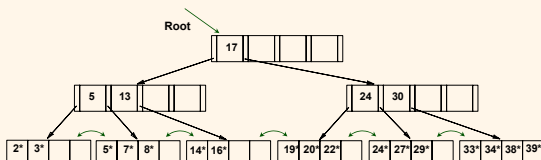

Inserting a Data Entry into a B+ Tree

- ❖ 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.*

Inserting 8* into Example B+ Tree

- ❖ 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.

Entry to be inserted in parent node.
(Note that 5 is copied up and continues to appear in the leaf.)

| 5 |

| 2* | 3* | | | | 5* | 7* | 8* | |

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.)

| 17 |

| 5 | 13 | | | | 24 | 30 | |

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* |

- ❖ Notice that root was split, leading to increase in height.
- ❖ In this example, we can avoid split by re-distributing entries; however, this is usually not done in practice.

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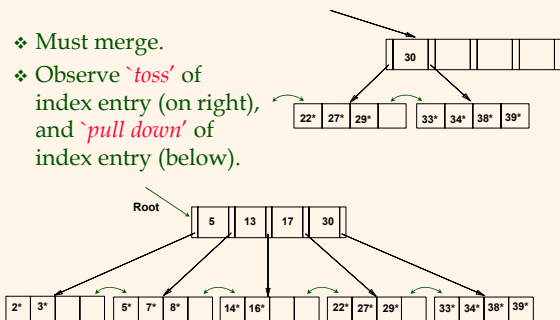- ❖ Merge could propagate to root, decreasing height.

Example Tree After (Inserting 8*, Then) Deleting 19* and 20* ...

Root

```
         17

   5   13              27   30

2* 3*   5* 7* 8*   14* 16*   22 24*   27* 29*   33* 34* 38* 39*
```
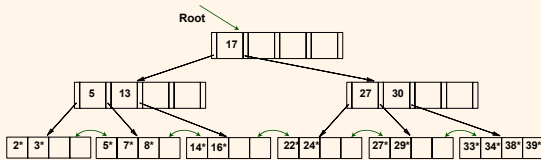
❖ Deleting 19* is easy.
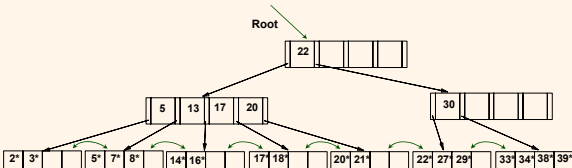❖ Deleting 20* is done with re-distribution. Notice how middle key is *copied up*.

... And Then Deleting 24*

❖ Must merge.
❖ Observe `*toss*' of index entry (on right), and `*pull down*' of index entry (below).

```
                    30

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

Root

```
      5   13   17   30

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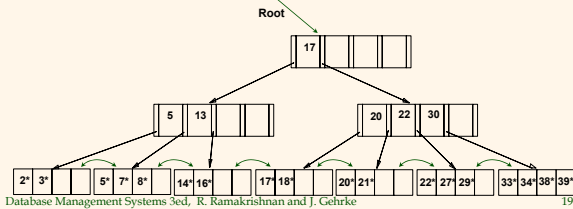

Example of Non-leaf Re-distribution

❖ Tree is shown below *during deletion* of 24*. (What could be a possible initial tree?)
❖ In contrast to previous example, can 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*
```

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*
```

Prefix Key Compression

- ❖ Important to increase fan-out. (Why?)
- ❖ Key values in index entries only `direct traffic'; can often compress them.
  - ▪ E.g., If we have 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*? (Can only compress *David Smith* to *Davi*)
    - • In general, while compressing, must leave each index entry greater than every key value (in any subtree) to its left.
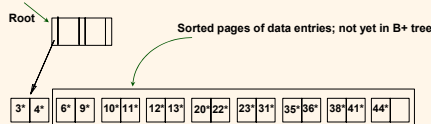- ❖ Insert/delete must be suitably modified.

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.
- ❖ *Bulk Loading* can be done much more efficiently.
- ❖ *Initialization*: Sort all 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 (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: _Bulk Loading_
  - 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.

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 differnt 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)).

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!
❖ 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.