THE UNIVERSITY *of* EDINBURGH

Advanced Database Systems

Spring 2024

Lecture #14:

Tree-Structured Indexing

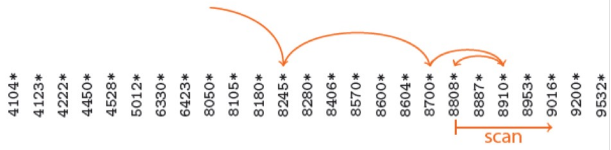
R&G: Chapter 10

Sorted Files and Binary Search

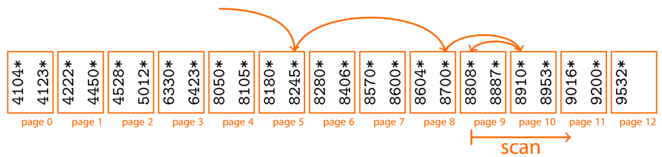
Efficient evaluation of range queries

```
SELECT * FROM Customer
 WHERE zipcode BETWEEN 8800 AND 8999
```

- Sort table on disk by zipcode
- Use binary search to find the first qualifying record
- Scan as long as zipcode ≤ 8999



Sorted Files and Binary Search



Sequential access during the scan phase

Need to read **$\log_2$(#records)** records during the search phase

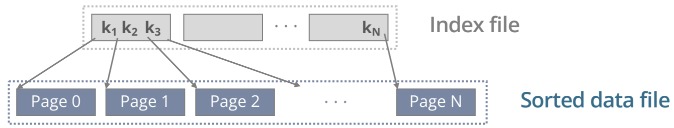
- Need to read about as many pages ☹
- Fan-out of 2 → deep tree → lots of I/Os
- Make far, unpredictable jumps ⇒ bad for page prefetching

Tree-Structured Indexing

Two index structures particularly shine with **range selections**

- **ISAM**: Static structure
- **B+ Tree**: Dynamic structure, adjust gracefully under inserts and deletes

Simple idea: Create an 'index' file

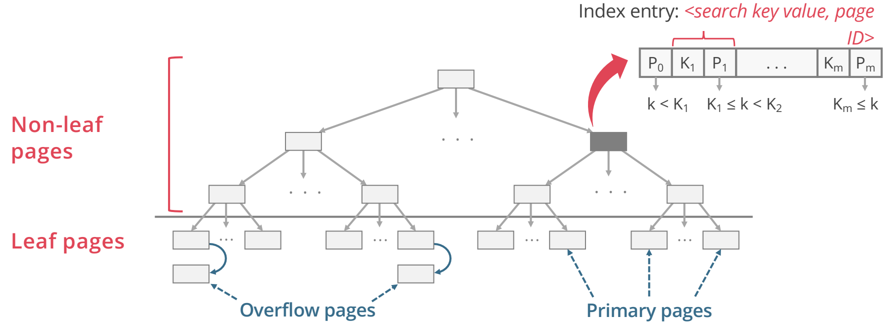


Can do binary search on (smaller) index file!

Tree-Structured Indexing

- Size of index is likely much smaller than size of data
  - Searching the index file is far **more efficient** than searching the data file
- Index file may still be quite large ⇒ **apply the idea repeatedly!**
  - Treat the topmost index level as data file
  - Add an index level on top of that
- Repeat until the topmost index level fits on one page
  - The topmost level is called the **root** page

ISAM: Indexed Sequential Access Method

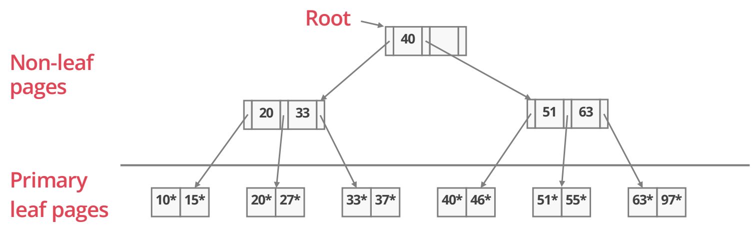


- Non-leaf pages only direct searches
- Leaf pages contain sorted index data entries **k*** (e.g., **<k, rid>**)

ISAM: Indexed Sequential Access Method

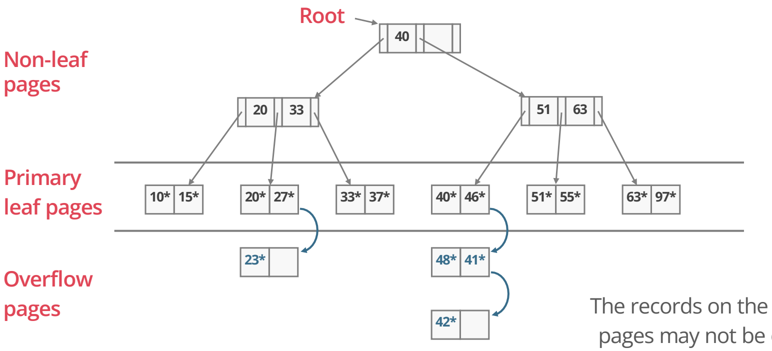
- Leaf (data) pages allocated **sequentially**, sorted by search key
  - No need to link leaf pages together
- **Search**: Start at root, use key comparisons to go to leaf
- **Insert**: Find the leaf where record belongs to
  - Insert record there if enough space
  - Otherwise, create an overflow page hanging off the primary leaf page
- **Delete**: Find and remove record from its leaf
  - If an overflow page becomes empty, deallocate it
- **Static tree structure: inserts/deletes affect only leaf pages**

Example ISAM Tree



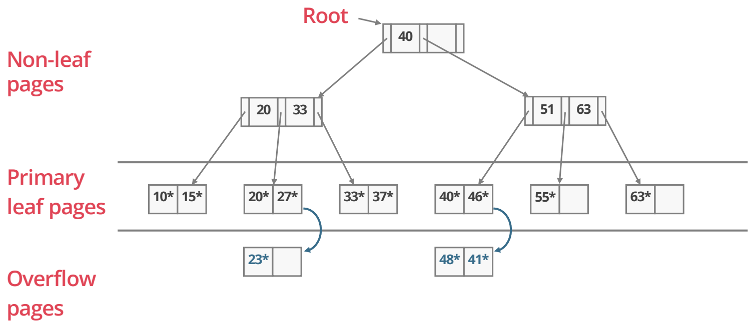
- Each node can hold two index entries plus one page pointer (the left-most one)

After Inserting 23*, 48*, 41*, 42*…



The records on the overflow pages may not be ordered

… Then Deleting 42*, 97*, 51*



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

Comments on ISAM

Non-leaf levels are not affected by inserts/deletes

- Need <u>not</u> be locked during concurrent index accesses
- Locking can be a bottleneck in dynamic tree indexes (particularly near the root)

ISAM may **lose balance** after heavy updating

- Creating long chains of (unsorted) overflow pages
- Search performance can degrade over time
- Leaving free space (~20%) during index creation partially reduces this problem

ISAM may be the index of choice for relatively static data

ISAM vs. Binary Search

N = number of pages in the data file (search space)

Fanout **F** = max #children / index node

- **F** = 3 in the previous example; **F** = 1000 typically

From the root page we are guided into an index subtree of size **N / F**

After **s** steps down the tree, the search space is reduced to $N \cdot (1/F)^s$

Assume we reach a leaf node after **s** steps

- $N \cdot (1/F)^s = 1$ hence $s = \log_F(N)$
- $F \gg 2$, hence $\log_F(N) \ll \log_2(N)$

ISAM is much more efficient than binary search!

B+ Tree: Most Widely Used Index

B+ tree is like ISAM but

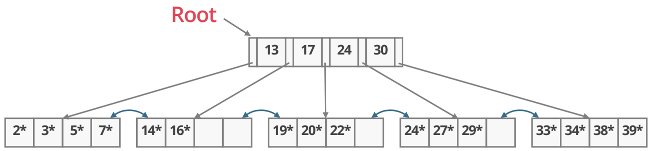
- **Has no overflow chains, it remains always balanced**
  - I.e., every leaf is at same depth
- Search performance **only dependent on the height**
  - Because of high fanout **F**, the height rarely exceeds 3
- Offers **efficient insert/delete procedures**
  - The data file can grow/shrink dynamically, non-leaf nodes are modified
- Each node (except the root) has a **minimum occupancy** of 50%
  - Each non-root node contains **d ≤ m ≤ 2d** entries, **d** is called the **order** of the tree

Original publication: R. Bayer and E.M. McCreight. Organisation and Maintenance of Large Ordered Indices. Acta Informatica, 1:3, 1972

Example B+ Tree

B+ tree of order **d** = 2

Note that leaf pages are doubly linked

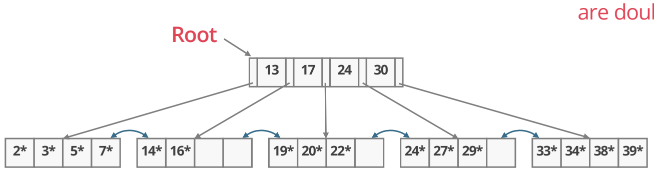


- Occupancy Invariant:
  - Each non-root node is at least partially full: **d** ≤ #entries ≤ **2d**
  - Max fan-out = **2d + 1**
- Data pages at bottom need not be stored in sequential order
  - Leaf pages allocated dynamically, linked via next and prev pointers

Example B+ Tree

B+ tree of order **d** = 2

Note that leaf pages are doubly linked



- 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 know it is not in the tree!

B+ Trees in Practice

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

- Average fanout **F** = 2*100*0.67 = 133

Typical capacities

- Height 4: 133^4 = 312,900,721 records
- Height 3: 133^3 = 2,352,637 records

Can often hold top levels in buffer pools

- Level 1 = 1 page = 8KB
- Level 2 = 133 pages = 1MB
- Level 3 = 17,689 pages = 138MB

Inserting a Data Entry

Find correct leaf **L**

Put data entry into **L** in sorted order

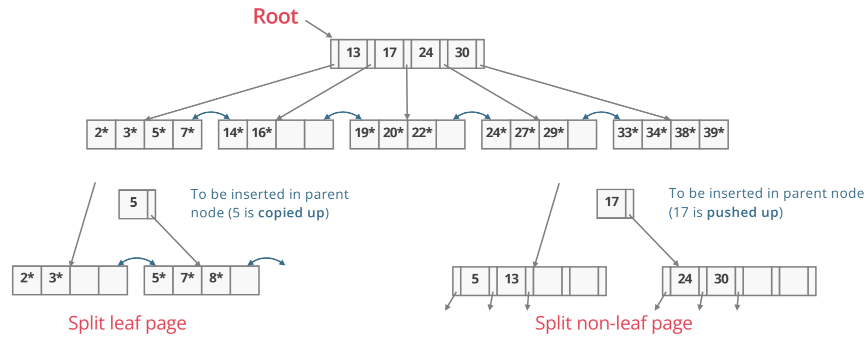
If **L** has enough space, done!

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**

To split inner node, redistribute entries evenly, but **push up** middle key

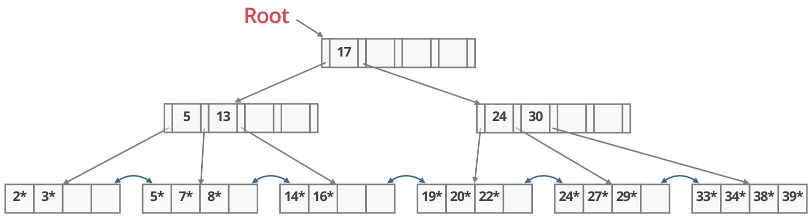
Inserting 8* into Example B+ Tree



Observe how minimum occupancy is guaranteed in both leaf and non-leaf page splits

Note difference between **copy-up** and **push-up**; be sure you understand the reasons for this

Example B+ Tree After Inserting 8*

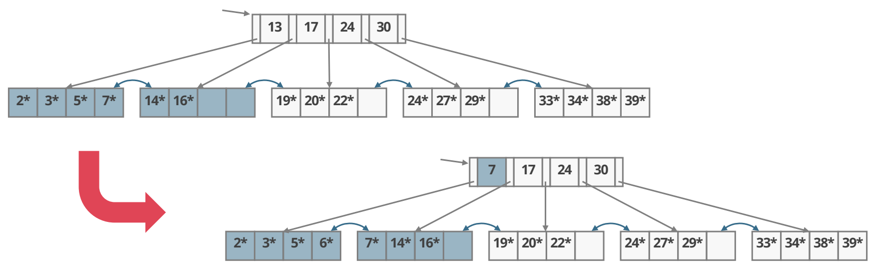


Notice that root was split, leading to increase in height

In this example, we can avoid split by **re-distributing** entries

Redistribution: Inserting 6*

Move keys to under-filled sibling pages and adjust separator



Adds additional I/O, but more efficient space use

In practice, redistribution is done only at leaf level (pointers provide direct access to siblings!)

21

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 **redistribute**, borrowing from sibling (adjacent node with same parent as **L**)
  - If redistribution 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

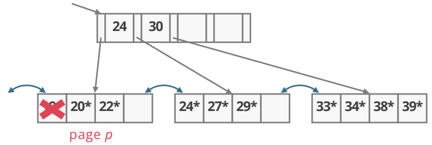
21

22

Deleting 19* and 20*...

Deleting 19* is easy

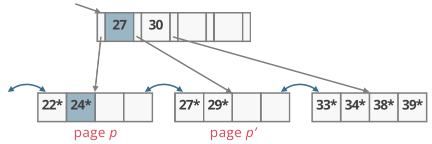
no underflow since *p* remains with **d** = 2 entries



Deleting 20*

p underflow and *p′* has > **d** entries
⇒ **(leaf node) redistribution**

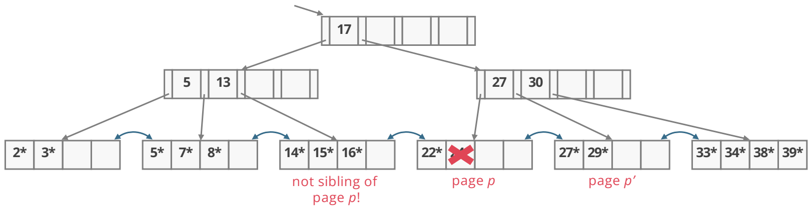
Notice how middle key is **copied up**



22

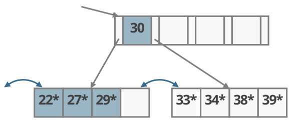
23

... And Then Deleting 24*



p underflow and *p′* has **d** = 2 entries
⇒ **(leaf node) merge**

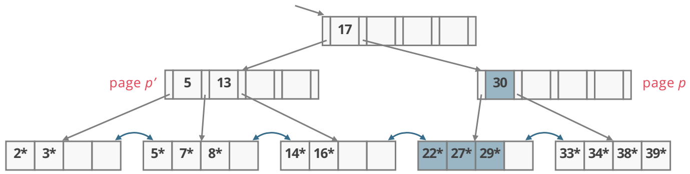
Delete separator between *p* and *p′* (**27**) recursively



23

24

... Still Deleting 24* (Merge)

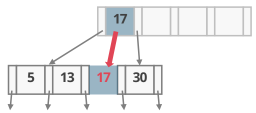


Delete **27** separator

p underflow and *p′* has **d** = 2 entries
⇒ **(non-leaf node) merge**

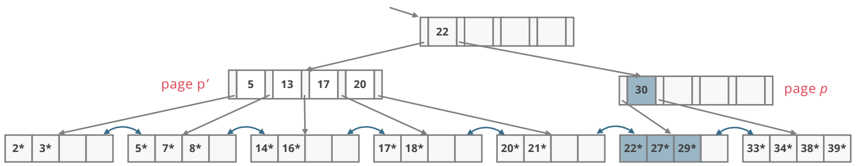
Merge *p* and *p′* by “pulling down” the separator

Since root is empty, delete it



24

… Still Deleting 24* (Redistribution)

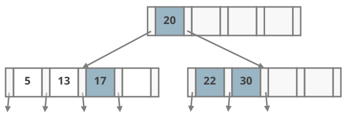


Assume a different left subtree

p underflow and *p′* has > **d** = 2 entries

⇒ (non-leaf node) redistribution

Redistribute entries by ’pushing through’ the splitting entry in the parent



B+ Tree: Deletions

In practice, occupancy invariant often not enforced

Just delete leaf entries and leave space

If new inserts come, great

- This is common

If page becomes completely empty, can delete

- Parent may become underflow
- That’s OK too

Guarantees still attractive: $\log_F$(max size of tree)

Variable Length Keys & Records

So far we have been using integer keys

What would happen to our occupancy invariant with variable length keys?

What about data in leaf pages stored using Variant C?

Redefine Occupancy Invariant

Order (**d**) makes little sense with variable-length entries

- Different nodes have different numbers of entries
- Non-leaf index pages often hold many **more entries** than leaf pages
- Even with fixed length fields, Variant **C** gives variable length data entries

Use a physical criterion in practice: **at-least half-full**

- Measured in **bytes**

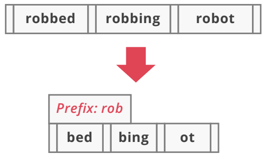
Many real systems are even sloppier than this

- Only reclaim space when a page is completely empty
- Basically the deletion policy we described above…

Optimizations

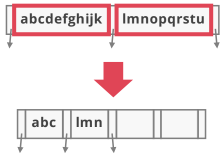
Prefix compression

- Sorted keys in the same leaf node are likely to have the same prefix
- Instead of storing entire keys, extract common prefix and store only unique suffix for each key



Suffix truncation

- The keys in the inner nodes are only used to “direct traffic”. We do not need the entire key
- Store a minimum prefix needed to correctly route probes into the index



Summary

ISAM and B+ tree support both **range searches** and **equality searches**

ISAM suitable for mostly static data

B+ tree is always a good choice

Great B+ tree visualisation:
https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html