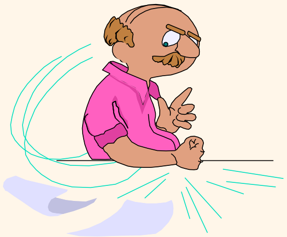
Introduction to Data Management

Lecture 21
(Indexing Wrap-up)

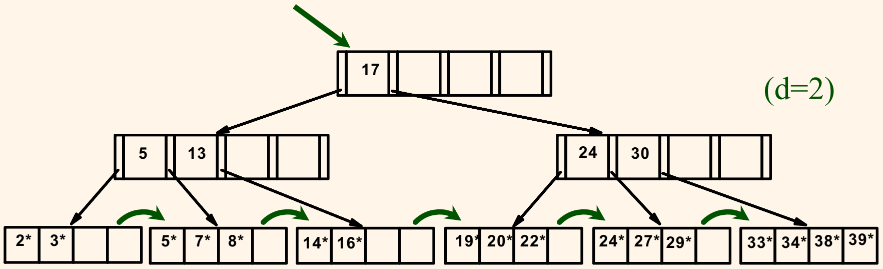


Instructor: Mike Carey
mjcarey@ics.uci.edu

Announcements

- Midterm #2 is **Wednesday (5/22) at 5 PM (!)**
  - Relational languages (see syllabus!)
  - Sample exam from last year is available
  - Assigned seating, similar to last time
- HW #6 is due **today at 7 PM**
  - One late "day" (***22 hours***) will be available
  - Solution coming tomorrow after **5 PM** (*really*)
- Today's lecture (assuming no surprises… ☺)
  - Finish our segment on database indexes
  - (Not on Midterm #2, of course)

Last Lecture We Went "Live"…



(d=2)

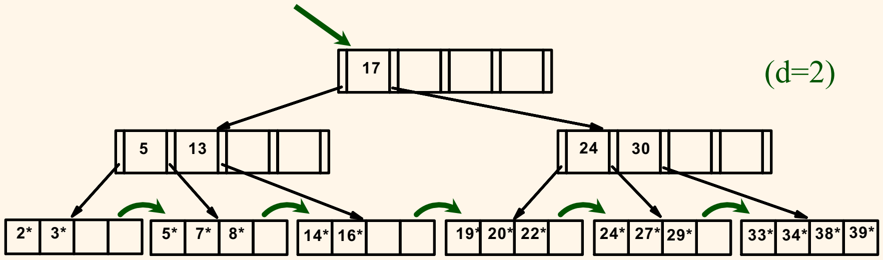
Reminder: Very cool online B+ tree viz tool available (☺)

- https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html
- Slight differences (internal key diffs: 13 → 14, 17 → 19)
- Their "*Max. Degree*" is our 2d+1 (limit of 5 ptrs/node above)

B+ Tree Deletion (Review)

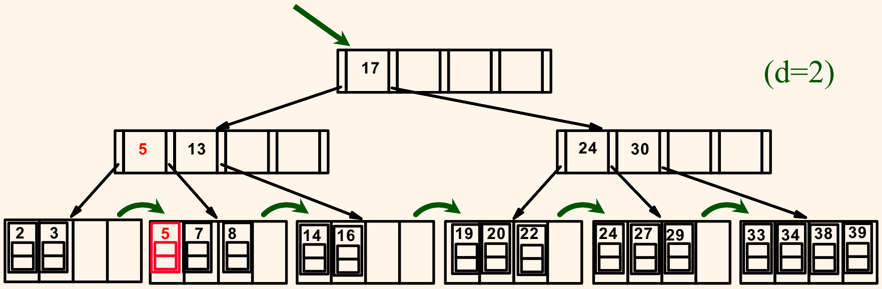
- Start at root, find leaf *L* where entry belongs.
- Remove the entry.
  - If L is still at least half-full, *done!*
  - If L has only **d-1** entries,
    - Try to redistribute, borrowing from *sibling (adjacent node with same parent as L).*
    - If re-distribution fails, ***merge*** *L* and sibling.
- If merge occurred, must delete search-guiding entry (pointing to *L* or sibling) from parent of *L*.
- Merge could propagate to root, decreasing height.

(Our previous B+ Tree, including 8)*



(d=2)

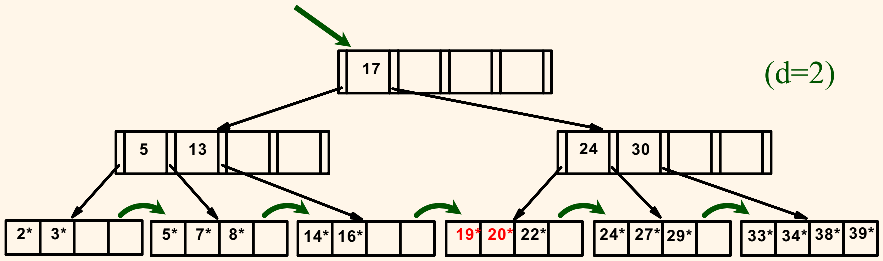
A Star Is (Un-)Born*



(d=2)

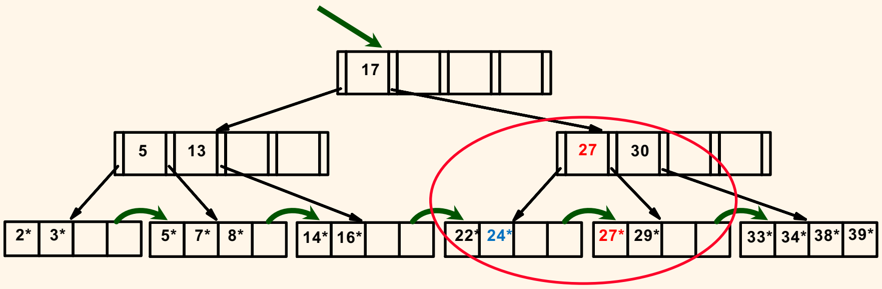
- Hopefully the picture above clarifies what's being vaguely denoted by the * notation…!
- Again: the leaves are where the ***data*** (like 5*, *a.k.a.* I(5)) actually lives!

Now, back to our previous B+ Tree...



(d=2)

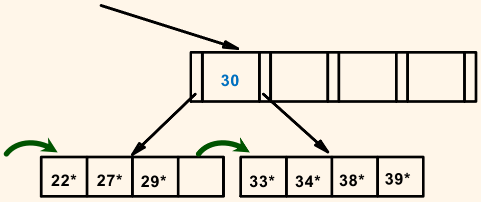
Example Tree After Deleting 19 and 20* ...*

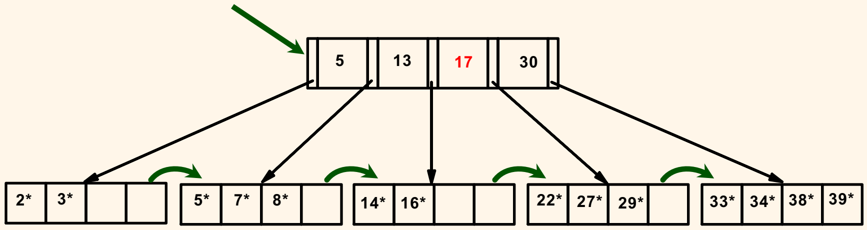


- Deleting 19* is easy.
- Deleting 20* is done with redistribution. Notice how middle key is *copied up*.

... And Then Deleting *24**

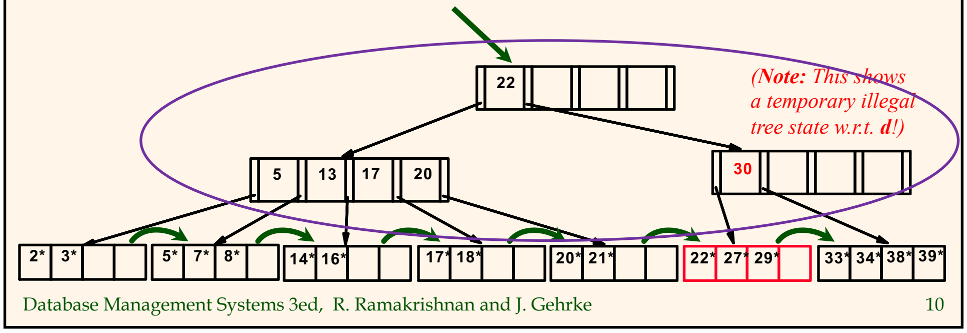
- Must merge.
- Observe "*toss*" of index entry (on right), and "*pull down*" of index entry (below).





Example of Non-leaf Redistribution

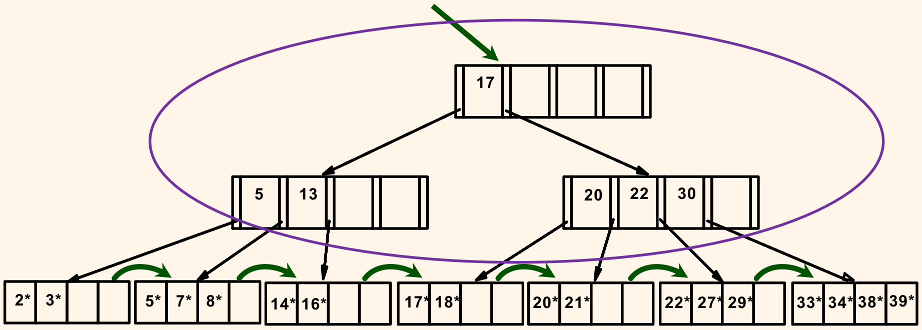
- New/different example B+ tree is shown below *during deletion* of an entry 24*
- In contrast to previous example, can redistribute entry from left child of root to right child.



*(**Note:** This shows a temporary illegal tree state w.r.t. **d**!)*

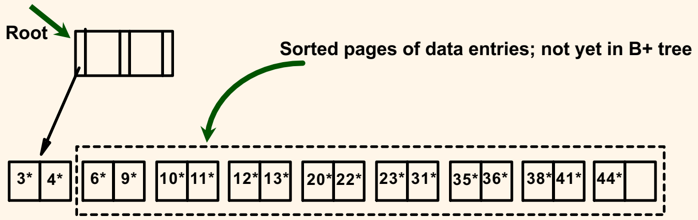
After Redistribution

- Intuitively, entries are redistributed by "*pushing through*" (or "rotating", if you prefer) the splitting entry in the parent node.



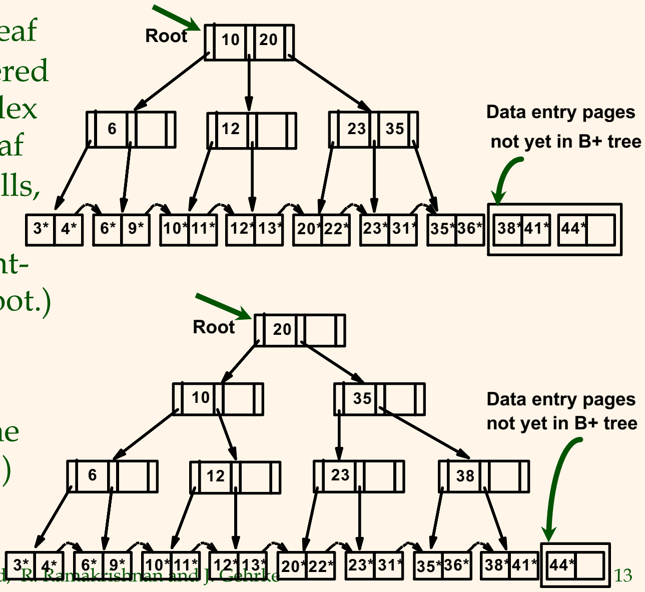
Bulk Loading of a B+ Tree

- 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.



Bulk Loading (cont'd.)

- Index entries for leaf pages always entered into right-most index page just above leaf level. When one fills, it splits. (A split may go up the right-most path to the root.)
- Much faster than repeated inserts!
- Can also control the leaf" fill factor" (%)



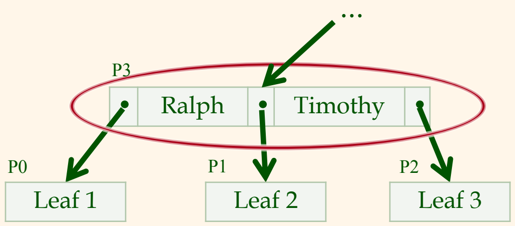
A Note on B+ Tree "Order"

- (Mythical!) *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 that 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 in the tree's leaf pages.

(Page Implementation Details)

Q: What if you were to "open up" a B+ Tree page?

- Control info (e.g., level, # children, free space offset)
- Search key array (with possible on-page indirection for variable-length data, using offsets), or key/data array – for non-leaf *vs.* leaf pages, respectively
- Child pointer array, where pointer = page id on disk!

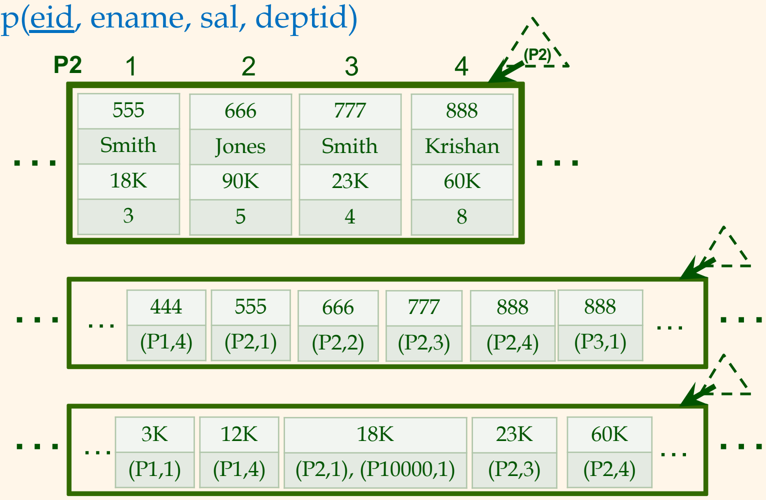


0	Level (1)
	NumChildren (3)
8	Free offset (40)
	Key 0 offset (32)
	Key 1 offset (37)
20	Child 1 page id (P0)
	Child 2 page id (P1)
	Child 3 page id (P2)
32	Key 0 ("Ralph")
37	Key 1 ("Timothy")
40	...

} *not to scale...*

(Leaf Page I(k) Alternatives Revisited)

Ex: Emp(eid, ename, sal, deptid)



Alternative 1: (records)

1	2	3	4
555	666	777	888
Smith	Jones	Smith	Krishan
18K	90K	23K	60K
3	5	4	8

Alternative 2: (RIDs)

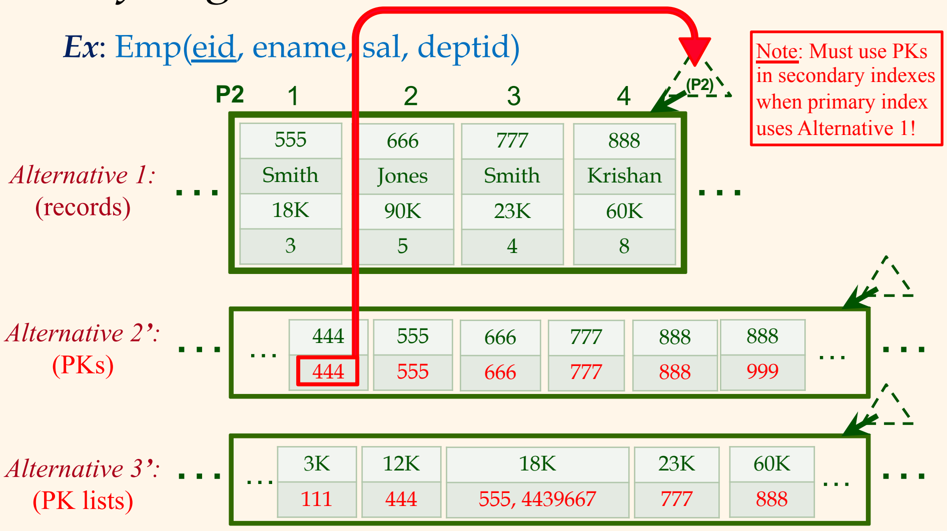
…							…
	444	555	666	777	888	888	
	(P1,4)	(P2,1)	(P2,2)	(P2,3)	(P2,4)	(P3,1)	

Alternative 3: (RID lists)

…						…
	3K	12K	18K	23K	60K	
	(P1,1)	(P1,4)	(P2,1), (P10000,1)	(P2,3)	(P2,4)	

(Leaf Page I(k) Alternatives, *cont.*)

Ex: Emp(eid, ename, sal, deptid)



Note: Must use PKs in secondary indexes when primary index uses Alternative 1!

Alternative 1: (records)

1	2	3	4
555	666	777	888
Smith	Jones	Smith	Krishan
18K	90K	23K	60K
3	5	4	8

Alternative 2': (PKs)

444	555	666	777	888	888
444	555	666	777	888	999

Alternative 3': (PK lists)

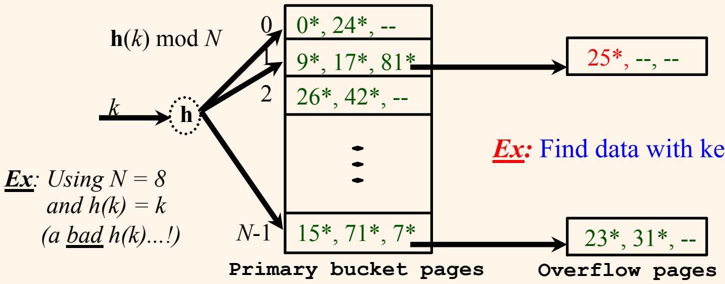
3K	12K	18K	23K	60K
111	444	555, 4439667	777	888

A Brief Aside: Hash-Based Indexes

- *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!*
- *Hash-based* indexes are fast for *equality selections*. ***Cannot*** support range searches.
- Static and dynamic hashing techniques exist; trade-offs similar to ISAM vs. B+ trees.

Static Hashed Indexes

- # primary pages fixed, allocated sequentially, never de-allocated; overflow pages if needed.
- **h**(*k*) mod *N* = bucket (page) to which data entry with key *k* belongs. (*N* = # of buckets)



Ex: *Using N = 8 and h(k) = k (a bad h(k)...!)*

Ex: Find data with key=25 → 25*

Static Hashed Indexes (*Cont'd.*)

- Buckets contain *data entries* (like for ISAM or B+ trees) – very similar to what we just looked at.
- Hash function works on *search key* field of record *r*. Must distribute values over range 0 ... M-1.
  - **h**(*key*) = (a * *key* + b) *mod M* works fairly well.
  - a and b are constants; lots known about how to tune **h**.
- Long overflow chains can develop and degrade performance. (Analogous to ISAM.)
  - *Extendible Hashing* and *Linear Hashing*: More dynamic approaches that address this problem. (Take CS122c!)

Indexing Summary

- Tree-structured indexes are ideal for range-searches, also good for equality searches.
- ISAM is a static structure. (Prehistoric B+ Tree!)
  - 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** → tree depth rarely more than 3-4.
  - https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html

Indexing Summary (Cont'd.)

- Bulk loading can be much faster than repeated inserts for creating a B+ tree on a large data set.
- Most widely used index in DBMS land, and also outside of DBMSs, because of its versatility. Also the most optimized (e.g., for bulk loads, locking, crash recovery, and so on).
- Other database indexes to be aware of:
  - Hash-based (for *exact-match* queries).
  - R-tree (for *spatial* indexing and queries).
  - Inverted keyword (for *text* indexing and queries).