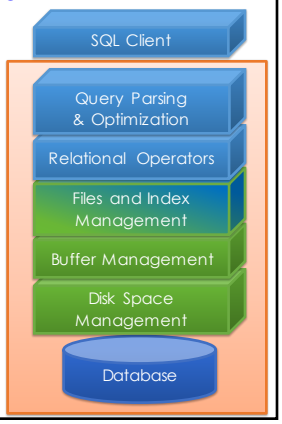
Tree-Structured Indexes

Berkeley cs186

Architecture of a DBMS

Last lecture:
- Cost of access patterns for different **file formats**
- Introduced **indexes** as a mechanism to reduce costs



Architecture of a DBMS

Last lecture:
- Cost of access patterns for different **file formats**
- Introduced **indexes** as a mechanism to reduce costs

Today we go deep on **tree indexes**: *B+-Trees*

Review: Files, Pages, Records

- Abstraction of stored data is "files" with "pages" of "records".
  - Records live on *pages*
  - Physical Record ID (RID) = <page#, slot#>
- *Variable length* data:
  - Fields in Records: offset array in header
  - Records on Pages: Slotted pages w/internal offsets & free space area
- Can be "lazy" about issues such as free space management, exact ordering, etc. – do it in batches
- Files can be unordered (heap), sorted, or kinda sorted (i.e., "clustered") on a *search key*.
  - Tradeoffs are update/maintenance cost vs. speed of accesses via the search key.
  - Files can be clustered (sorted) at most one way.
- Indexes can be used to speed up many kinds of accesses. (i.e., "access paths")

Tree-Structured Indexes: Introduction

- Selections of form field <op> constant
- Equality selections (op is =)
  - Either "tree" or "hash" indexes help here.
- Range selections (op is one of <, >, <=, >=, BETWEEN)
  - "Hash" indexes don't work for these.
- More complex selections (e.g. spatial containment)
  - There are fancier indexes that can do this…

- Tree-structured indexing techniques support both *range selections* and *equality selections*.
- *ISAM*: static structure; early index technology.
- *B+ tree*: dynamic, adjusts gracefully under inserts and deletes.
- ISAM = Indexed Sequential Access Method

Recall: Indexes

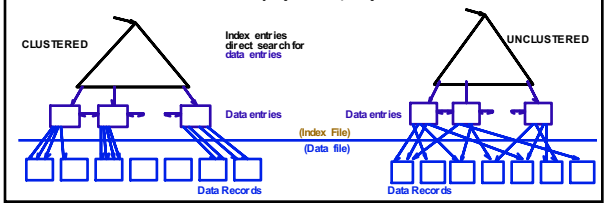
*An **index** is a data structure that enables fast lookup of data entries by search key*

- **Lookup:** may support many different operations
  - Equivalence, range, region, …
- **Search Key:** any *subset* of columns in the relation
  - Do not need to be unique
- **Data Entries:** items stored in the index, (**k**, {items})
  - Could be actual records or record ids (pointers)…
- **Many Types:** B-Tree, Hash, ISAM, R-Tree, GiST, …

- **Modification:** often support fast insert and delete
  - Lazily maintain ordering, clustering, **(today we address this)**

Recall: Data Storage Considerations

- How is data stored in the index?
  1. **By Value:** actual data record (with key value **k**)
  2. **By Reference:** <**k**, rid of matching data record>
  3. **By List of Refs.:** <**k**, list of rids of *all* matching data records>
- If stored externally (alt. 2, 3)

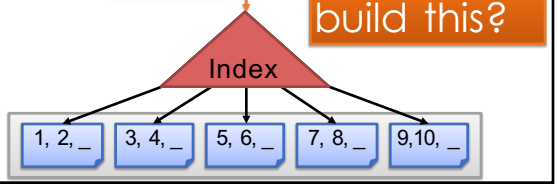


Recall: Example from Last Lecture

Assumptions:

- Store data by reference (Alternative 2)
- Clustered tree index with 2/3 full heap file page
  - **Sorted** heap file
  - **Fan-out** (F): relatively large.
  - Assume static index

How do we build this?



Simple Idea?

Input Heap File: 3, 4, 5 | 1, 2, 7 | 8, 6, 9 | 10,_,_

- Sort heap file & leave some space

1, 2, _ | 3, 4, _ | 5, 6, _ | 7, 8, _ | 9,10, _

  - Pages in logical order (seq. disk access)
  - Do we need next pointers?
    - Pages are in logical order (might still add them for simplicity)

Simple Idea?

Input Heap File: 3, 4, 5 | 1, 2, 7 | 8, 6, 9 | 10,_,_

- Sort heap file & leave some space

1, 2, _ | 3, 4, _ | 5, 6, _ | 7, 8, _ | 9,10, _

- **Cost?** (Assume > Sqrt(#Pages) memory)
  - How many passes?
    - Two passes

Simple Idea?

Input Heap File: 3, 4, 5 | 1, 2, 7 | 8, 6, 9 | 10,_,_

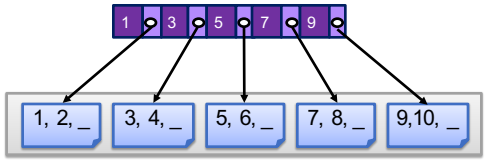
- Sort heap file & leave some space

1, 2, _ | 3, 4, _ | 5, 6, _ | 7, 8, _ | 9,10, _

- **Step 2:** *Build the index…*
  - Why not just binary search heap file?
    - Load record data just to read key
    - Small fan-out → deep tree → more IOs

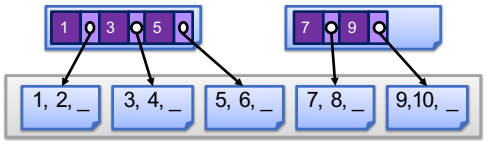
Build a high fan-out search tree

- Start simple: *Sorted (key, page id) file*
  - No record data
  - Binary search key file
  - **Forgot:** Need to break across pages!



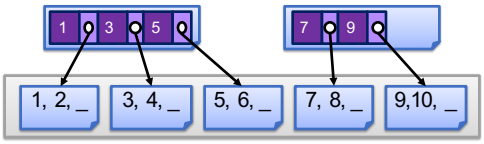
Build a high fan-out search tree

- Start simple: *Sorted (key, page id) file*
  - No record data
  - Binary search key file
  - **Forgot:** Need to break across pages!



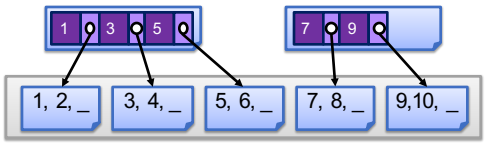
Build a high fan-out search tree

- Start simple: *Sorted (key, page id) file*
  - No record data
  - Binary search key file
    - **Complexity?**



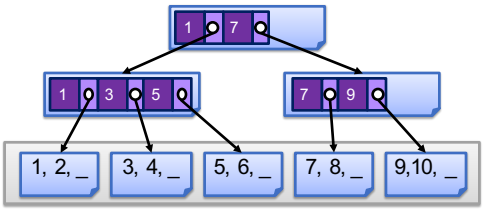
Build a high fan-out search tree

- Start simple: *Sorted (key, page id) file*
  - No record data
  - Binary search key file
    - **Complexity:** Still binary search of #Pages /(# Pair<key,ptr> that fit on page)



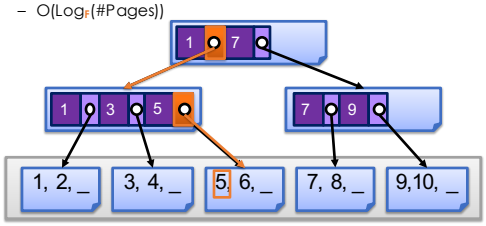
Build a high fan-out search tree

- Recursively "index" key file
- *Key* Invariant: ← Pun intended
  - Node[…, (K_L, P_L), (K_R, P_R),…] ➔ All tuples in range $K_L <= K < K_R$ are in tree P_L



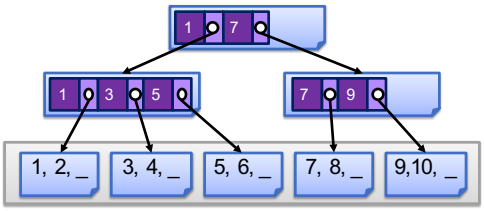
Search a high fan-out search tree

- Searching for 5?
  - Binary Search each node (page) starting at root
  - Follow pointers to next level of search tree
- **Complexity?**
  - O(Log_F(#Pages))



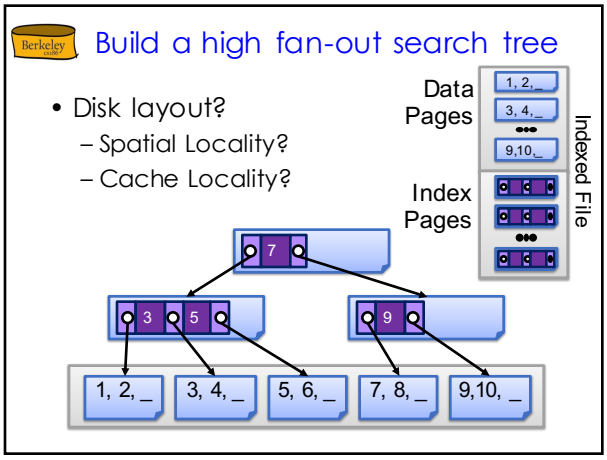
Left Key Optimization

- Optimization:
  - Do we need the left most key?



Build a high fan-out search tree

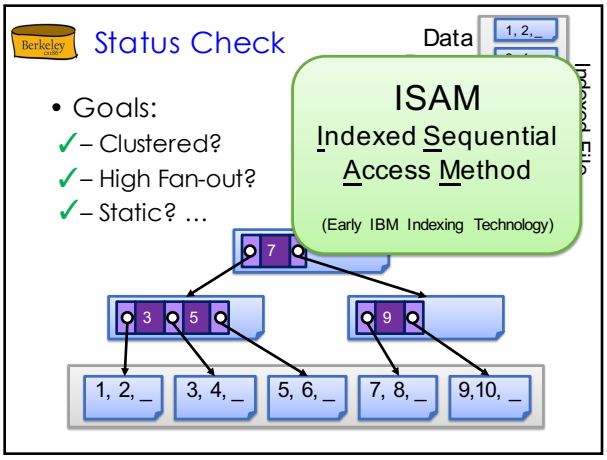
- Disk layout?
  - Spatial Locality?
  - Cache Locality?



Status Check

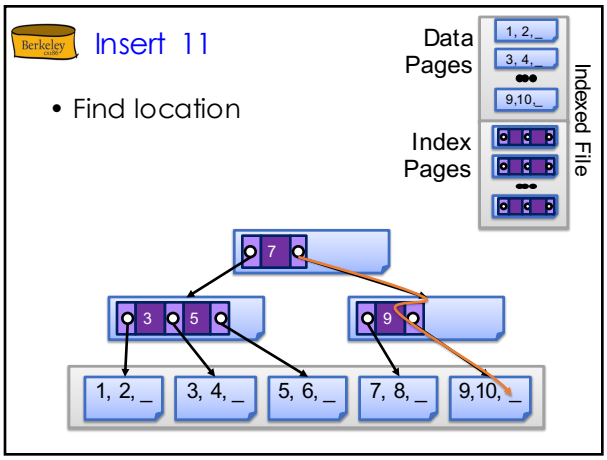
- Goals:
  - ✓ – Clustered?
  - ✓ – High Fan-out?
  - ✓ – Static? …

ISAM
Indexed Sequential Access Method
(Early IBM Indexing Technology)



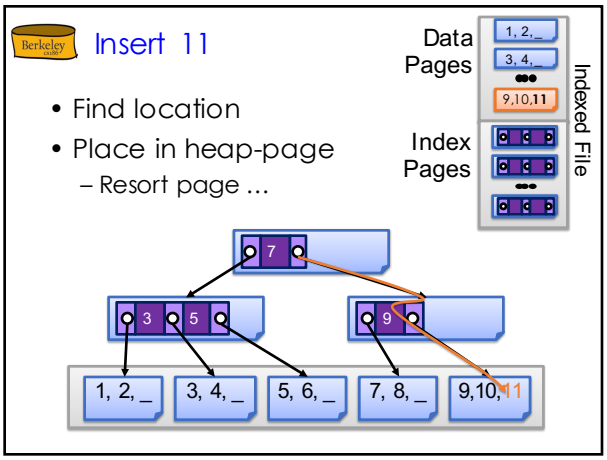
Insert 11

- Find location



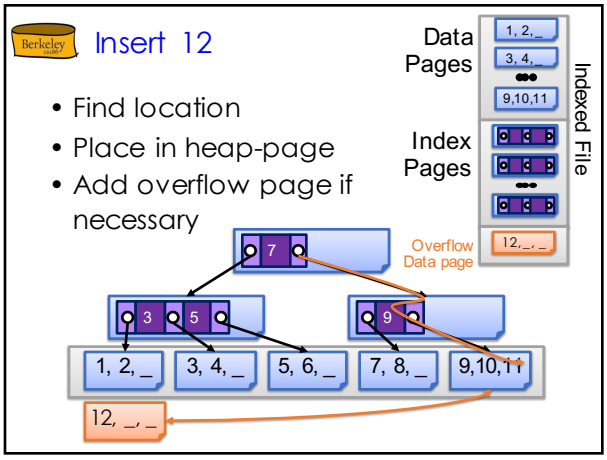
Insert 11

- Find location
- Place in heap-page
  - Resort page …



Insert 12

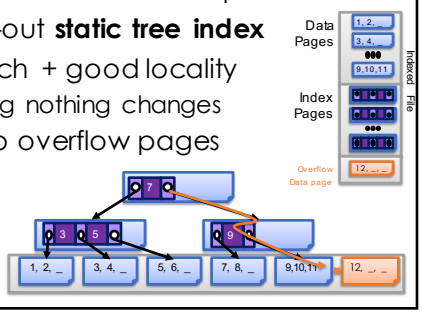
- Find location
- Place in heap-page
- Add overflow page if necessary



Recap

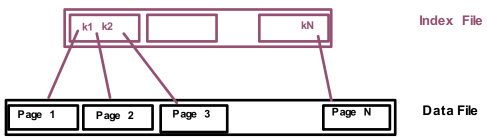
ISAM
Indexed Sequential Access Method

- Data entries in sorted heap file
- High fan-out **static tree index**
- Fast search + good locality
  - Assuming nothing changes
- Insert into overflow pages



Range Searches

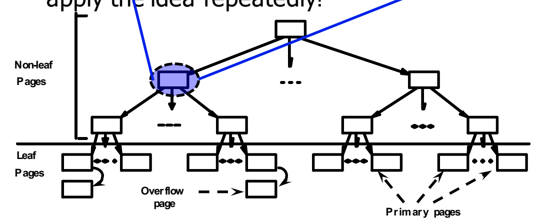
- *"Find all students with gpa > 3.0"*
  - Sorted file? Binary search to find first, scan to find others.
  - Cost of binary search in a database can be quite high. Q: Why?
- Simple idea: Create an 'index' file.



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

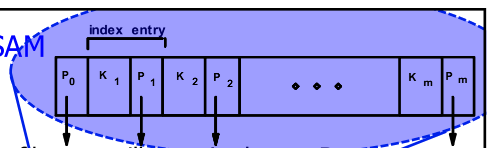
ISAM

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

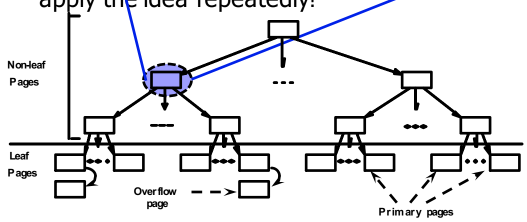


☛ *Leaf pages contain data entries.*

ISAM



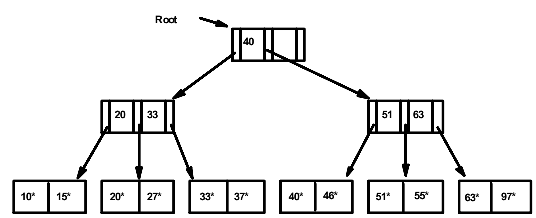
- Index file may still be quite large. But we can apply the idea repeatedly!



☛ *Leaf pages contain data entries.*

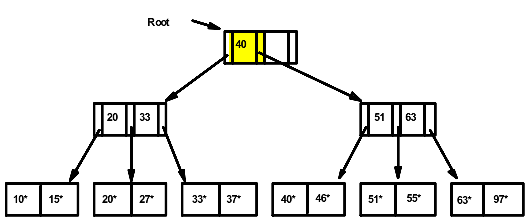
Example ISAM Tree

- ***Index entries*:<search key value, page id>** **they direct search for data entries *in leaves.***
- **Example where each node can hold 2 entries;**



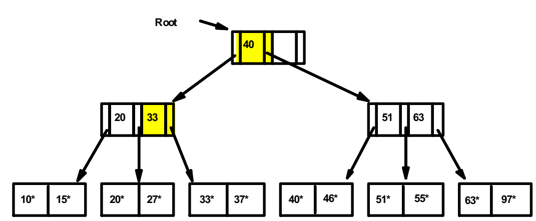
Example ISAM Tree

- ***Index entries*:<search key value, page id>** **they direct search for data entries *in leaves.***
- **Example where each node can hold 2 entries;**



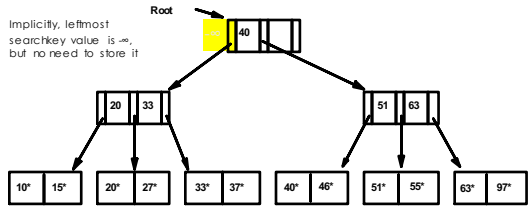
Example ISAM Tree

- ***Index entries*:<search key value, page id>** **they direct search for data entries *in leaves.***
- **Example where each node can hold 2 entries;**



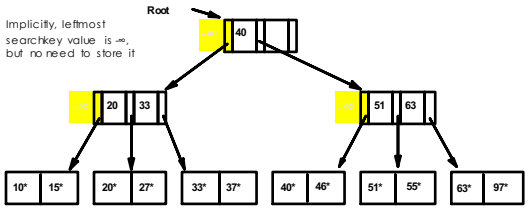
Example ISAM Tree

- ***Index entries*:<search key value, page id>**
  they direct search for data entries *in leaves.*
- **Example where each node can hold 2 entries;**



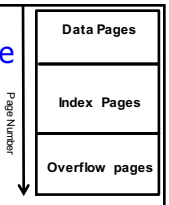
Example ISAM Tree

- ***Index entries*:<search key value, page id>**
  they direct search for data entries *in leaves.*
- **Example where each node can hold 2 entries;**



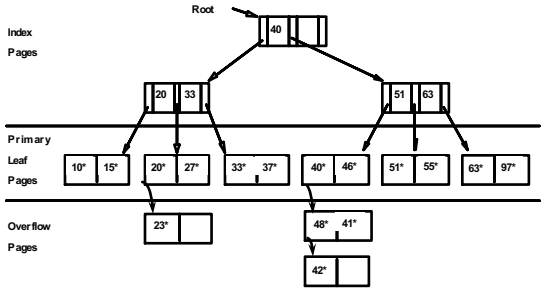
ISAM is a STATIC Structure

- *File creation*:
  - Leaf (data) pages stored in order by search key
  - then index pages
  - then overflow pgs.
- *Search*: Start at root; use key comparisons to go to leaf.
- Cost = $\log_F N$
  - F = # entries/pg (i.e., fanout)
  - N = # leaf pgs
  - no need for `next-leaf-page' pointers. (Why?)
- *Insert*: Find leaf that data entry belongs to, and put it there. **Overflow page** if necessary.
- *Delete*: Seek and destroy! If deleting a tuple empties an overflow page, de-allocate it and remove from linked-list.

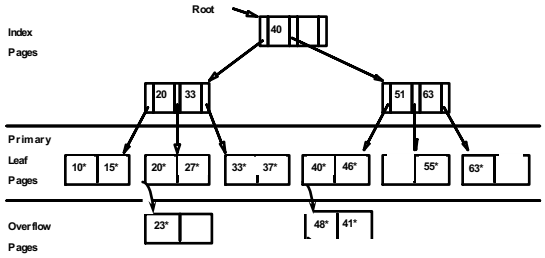


Static tree structure: *inserts/deletes affect only leaf pages.*

Example: Insert 23*, 48*, 41*, 42*



… then Deleting 42*, 51*, 97*



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

A Note of Caution

- ISAM is an old-fashioned idea
  - Introduced by IBM in 1960s
  - B+-trees are usually better, as we'll see
    - Though not *always (← we'll come back to this)*
- But, it's a good place to start
  - Simpler than B+-tree, many of the same ideas
- Upshot
  - **Don't** brag about ISAM on your resume
  - **Do** understand ISAM, and tradeoffs with B+-trees

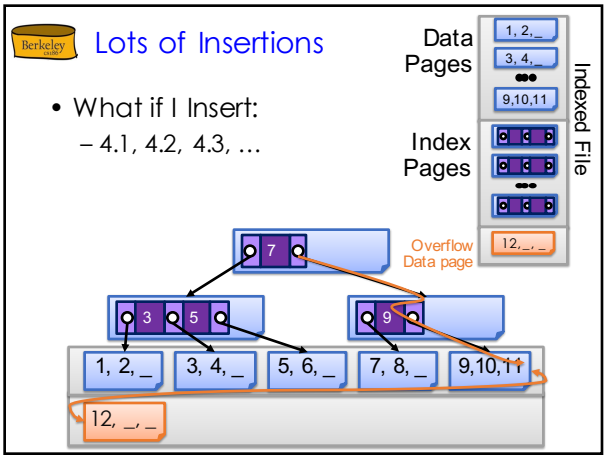
A Note of Caution

- ISAM is an old-fashioned idea
  - Introduced by IBM in 1960s
  - B+-trees are usually better, as we'll see
    - Though
- But, it's a
  - Simpler ideas
- Upshot
  - **Don't** brag about ISAM on your resume
  - **Do** understand ISAM, and tradeoffs with B+-trees

What's wrong with ISAM?

Lots of Insertions

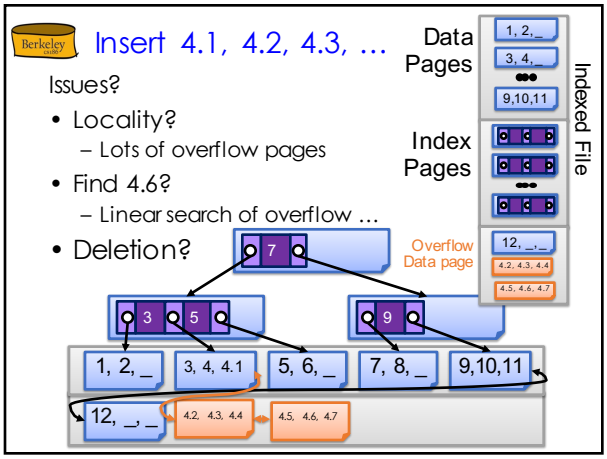
- What if I Insert:
  - 4.1, 4.2, 4.3, …



Insert 4.1, 4.2, 4.3, …

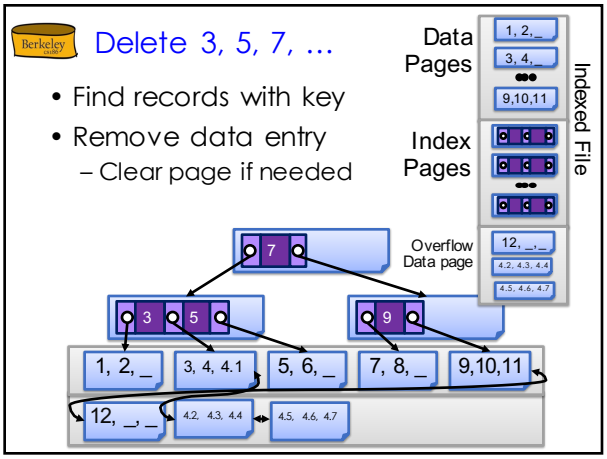
Issues?

- Locality?
  - Lots of overflow pages
- Find 4.6?
  - Linear search of overflow …
- Deletion?



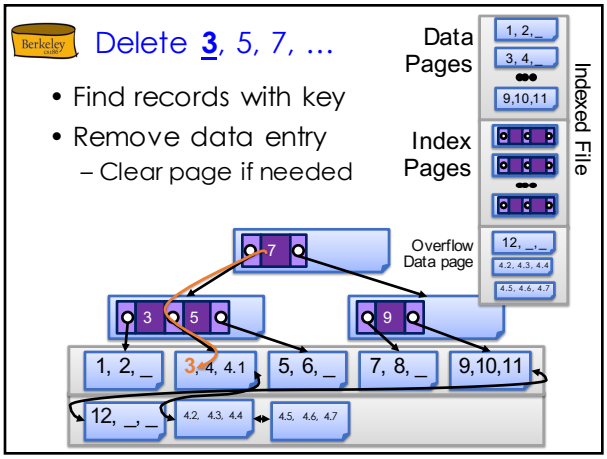
Delete 3, 5, 7, …

- Find records with key
- Remove data entry
  - Clear page if needed



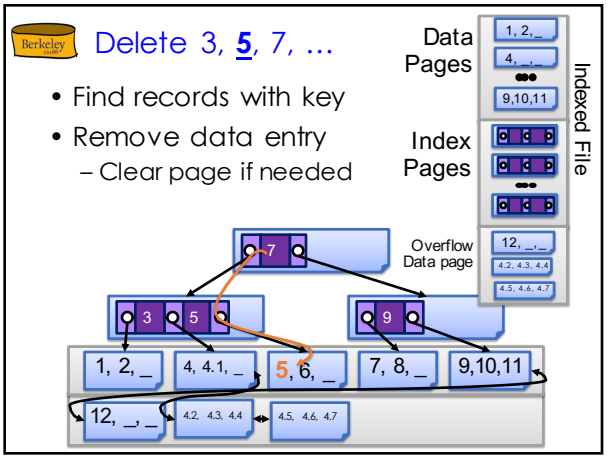
Delete **3**, 5, 7, …

- Find records with key
- Remove data entry
  - Clear page if needed



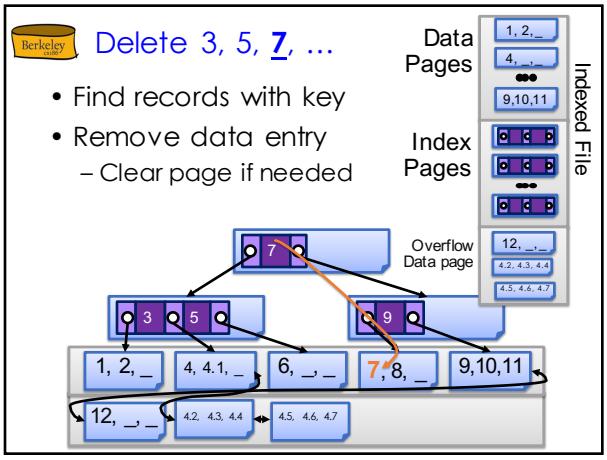
Delete 3, **5**, 7, …

- Find records with key
- Remove data entry
  - Clear page if needed



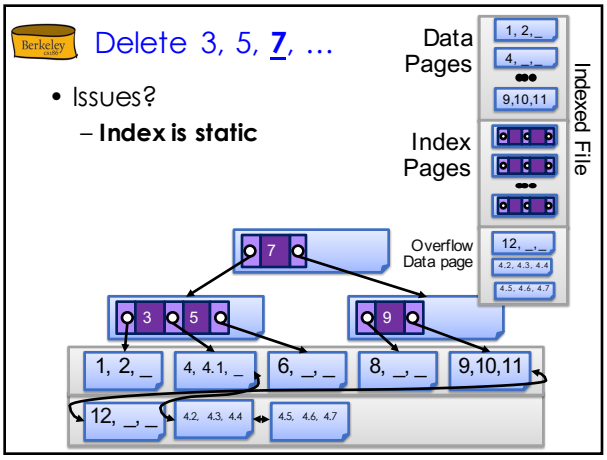
Delete 3, 5, 7, …

- Find records with key
- Remove data entry
  - Clear page if needed



Delete 3, 5, 7, …

- Issues?
  - **Index is static**



ISAM ---- Issues?

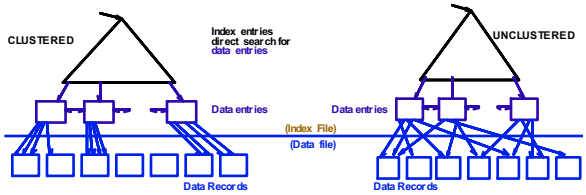
- Pros
  - Sequentially: Scan all records without touching index
  - Good for static data setting with big scans
    - When?
- Cons
  - Doesn't handle insertion or deletion well
  - Degrades to linear search

Indexes: B+-Trees

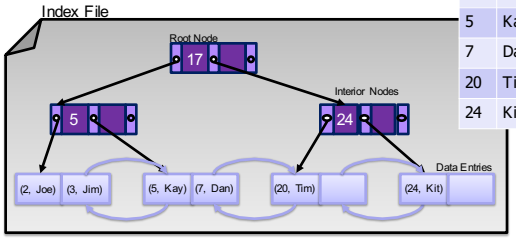
Berkeley cs186

Clarification: Data Storage Considerations

- How is data stored in the index?
  1. **By Value:** actual data record (with key value **k**)
  2. **By Reference:** <**k**, rid of matching data record>
  3. **By List of Refs.:** <**k**, list of rids of *all* matching data records>
- If stored externally (alt. 2, 3)



Alternative 1 Index (B+-Tree)

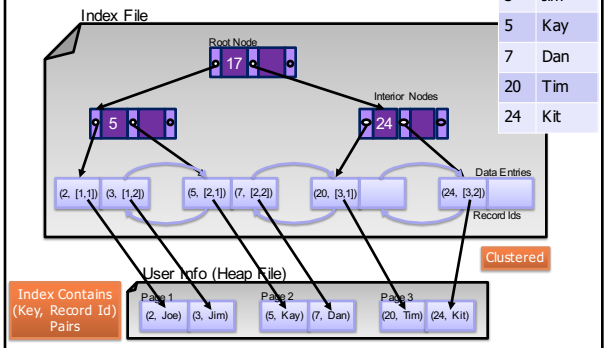


uid	name
2	Joe
3	Jim
5	Kay
7	Dan
20	Tim
24	Kit

- Record contents are stored directly in the index file.
  - No need to follow pointers.
- Need to be careful with moving data entries

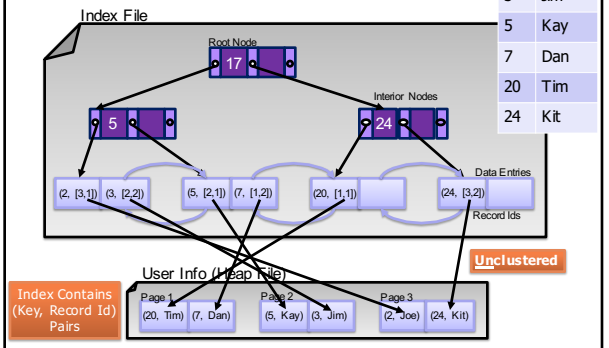
Alternative 2 Index (B+-Tree)

uid	name
2	Joe
3	Jim
5	Kay
7	Dan
20	Tim
24	Kit



Alternative 2 Index (B+-Tree)

uid	name
2	Joe
3	Jim
5	Kay
7	Dan
20	Tim
24	Kit



Enter the B+-Tree

- Similar to ISAM
  - Same interior node structure:
    - <Key, Page Ptr> Pairs with same guarantees
  - Same search routine as before
- **Dynamic Tree Index**
  - Always Balanced
  - Support efficient insertion deletion
    - Grows at root not leaves!
- "+"? B-tree that stores data in leaves only
  - Recall: Data = Records, Ref, or List of Ref

What does B stand for?

- Introduced in 1970 by Rudolf Bayer and Edward M. McCreight
- https://vimeo.com/73481096
  - Scrub to 16 minutes

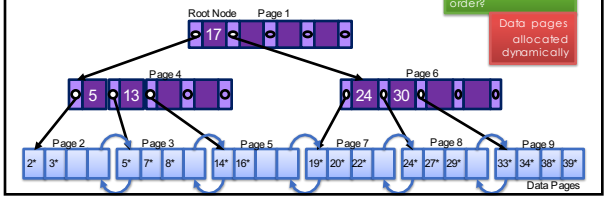
Example of a B+-Tree

- Structure is similar to ISAM
- **Occupancy Invariant:**
  Each interior node is at least partly full:
  d <= # entries <= 2d
  - d: ***order of the tree*** (max fan-out = 2d + 1)
- Data pages at bottom may not be in logical
  - Next and last pointers required

What is the value of d? 2

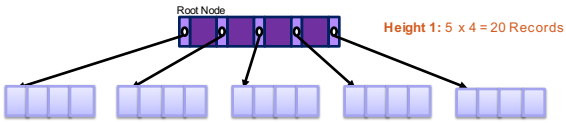
What about root? Root is special

Why not in sequential order? Data pages allocated dynamically



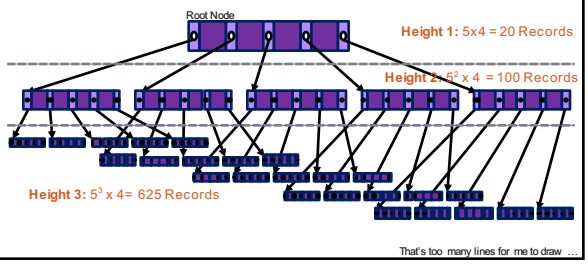
B+ Trees and Scale

- How big is a height 2 B+-Tree
  - d = 2 ➔ Fan-out?
  - Fan-out = 2d + 1 = 5



B+ Trees and Scale

- How big is a height 2 B+-Tree
  - d = 2 ➔ Fan-out?
  - Fan-out = 2d + 1 = 5



B+ Trees in Practice

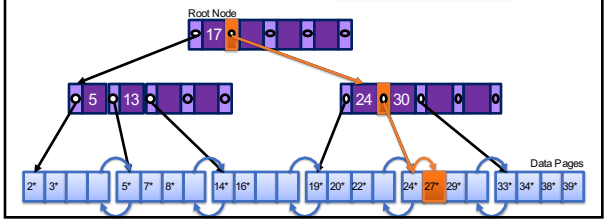
- Typical order: 1600. Typical fill-factor: 67%.
  - average fan-out = 2144
  - (assuming 128 Kbytes pages at 40Byte per record)
- At typical capacities:
  - Height 1: 2144^2 = **4,596,736 records**
  - Height 2: 2144^3 = **9,855,401,984 records**
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 128 Kbytes
  - Level 2 = 2144 pages = 274.4 Mbyte
    - Addresses 588.38 Gigabytes of leaf (data entry) pages.

Searching the B+-Tree

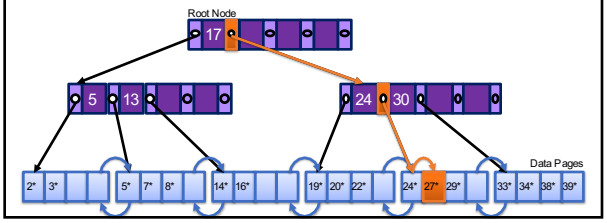
- Same as ISAM
- Find key = 27
  - Find split on each node
  - Follow pointer to next node

Use binary search on each page

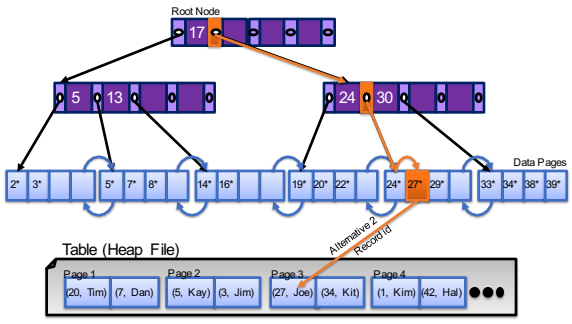
Am I done?
(Alternative 1 vs 2,3)



Searching the B+-Tree



Searching the B+-Tree

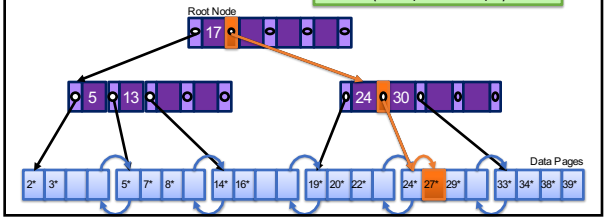


Searching the B+-Tree

- Same as ISAM
- Find key = 27
  - Find split on each node
  - Follow pointer to next node

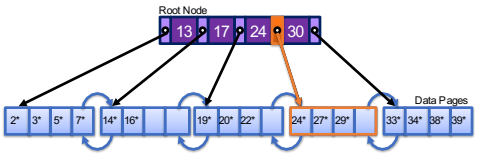
Use binary search on each page

How about insert?
(are you ready?)



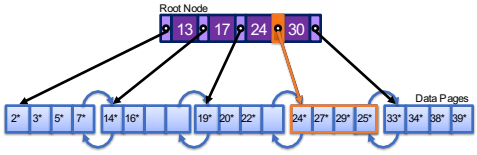
Inserting 25* into a B+-Tree

- Find the correct leaf



Inserting 25* into a B+-Tree

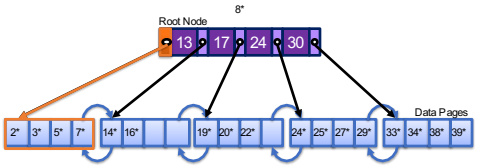
- Find the correct leaf
- If there is room in the leaf just add the entry
  - Sort the page leaf page by key



That was easy!
(it gets harder)

Inserting 8* into a B+-Tree

- Find the correct leaf



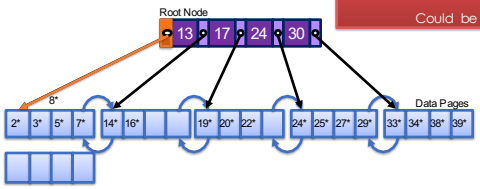
Inserting 8* into a B+-Tree

- Find the correct leaf
  - Split leaf if there is not enough room
  - Redistribute entries evenly

What happens to record ids if this is an Alternative 1 index (store data by value)?

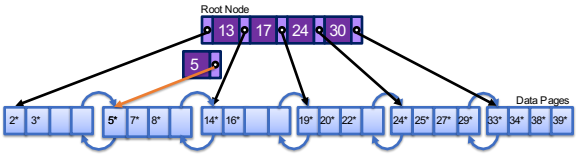
Careful: Moving records to new pages changes records id

Could be pricey!!!



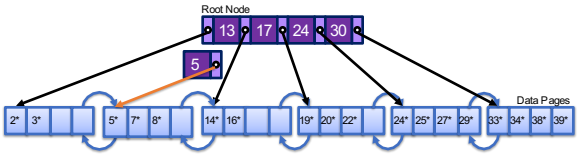
Inserting 8* into a B+-Tree

- Find the correct leaf
  - Split leaf if there is not enough room
  - Redistribute entries evenly
  - **Copy** up middle key



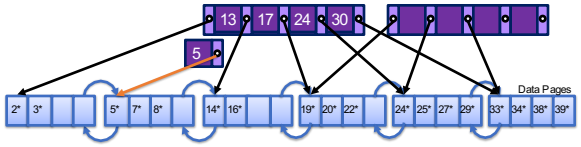
Inserting 8* into a B+-Tree

- Find the correct leaf
  - Split leaf if there is not enough room
  - Redistribute entries evenly
  - **Copy** up middle key



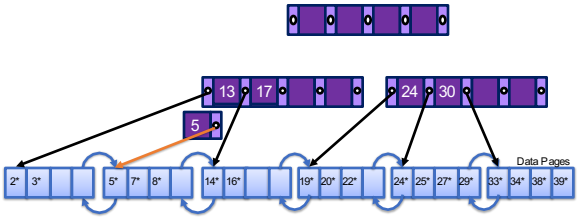
Inserting 8* into a B+-Tree

- Find the correct leaf
  - Split leaf if there is not enough room
  - Redistribute entries evenly
  - **Copy** up middle key
- Recursively split index nodes
  - Redistribute right d keys



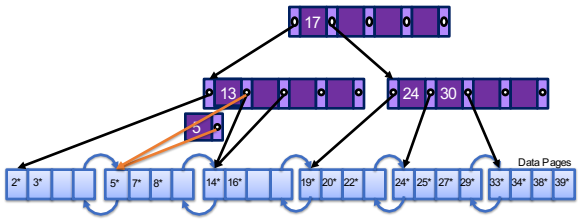
Inserting 8* into a B+-Tree

- Find the correct leaf
  - Split leaf if there is not enough room
  - Redistribute entries evenly
  - **Copy** up middle key
- Recursively split index nodes
  - Redistribute right d keys
  - **Push** up middle key



Inserting 8* into a B+-Tree

- Find the correct leaf
  - Split leaf if there is not enough room
  - Redistribute entries evenly
  - **Copy** up middle key
- Recursively split index nodes
  - Redistribute right d keys
  - **Push** up middle key



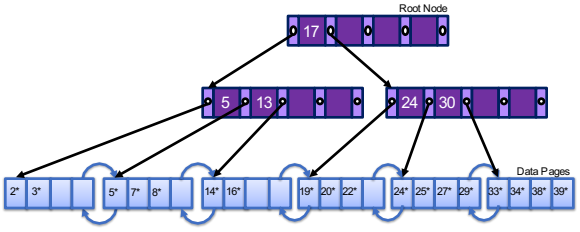
Inserting 8* into a B+-Tree

- Find the correct leaf
  - Split leaf if there is not enough room
  - Redistribute entries evenly
  - **Copy** up middle key
- Recursively split index nodes
  - Redistribute right d keys
  - **Push** up middle key

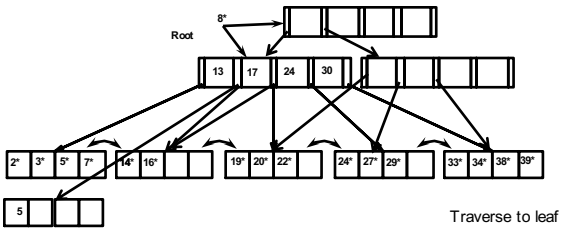
Check Invariants:

Key Invariant: Node [..., (Ki, Pi), ...] → Ki <= K for all K in Pi Sub-tree

Occupancy Invariant: d <= # entries <= 2d



Old Animation: Insert 8*



Traverse to leaf

Split Leaf

Copy up middle key

Split Parent

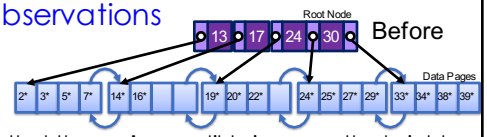
New Root

Push up middle key

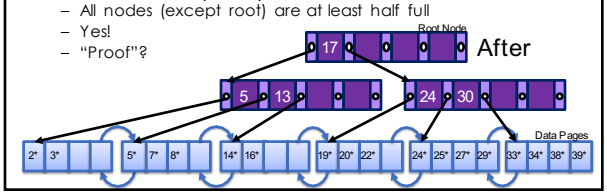
Inserting a Data Entry into a B+ Tree

- 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.*

Before and After Observations



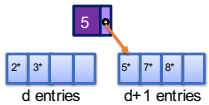
- Notice that the **root** was split to increase the height
  - *Grow from the root not the leaves*
  - ➔ All paths from root to leaves are equal lengths
- Does the occupancy invariant hold?
  - All nodes (except root) are at least half full
  - Yes!
  - "Proof"?



Splitting a Leaf

- Start with full leaf (2d) entries: d = 2
  - Add a 2d +1 entry (8*)

- Split into leaves with (d, d+1) entries
  - **Copy** key up to parent



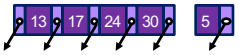
Why copy key and not push key up to parent?

Key has value attached (5*)

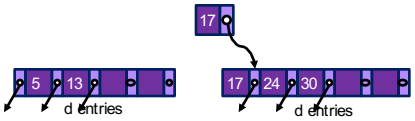
Occupancy invariant holds after split.

Splitting an interior node

- Start with full interior node (2d) entries: d = 2
  - Add a 2d +1 entry

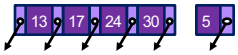


- Split into nodes with (d, d) entries
  - **Push** key up to parent



Splitting an interior node

- Start with full interior node (2d) entries: d = 2
  - Add a 2d +1 entry

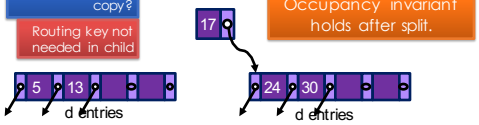


- Split into nodes with (d, d) entries
  - **Push** key up to parent

Why push and not copy?

Routing key not needed in child

Occupancy invariant holds after split.



Nice Animation On-Line

- Great online animation of B+-Trees

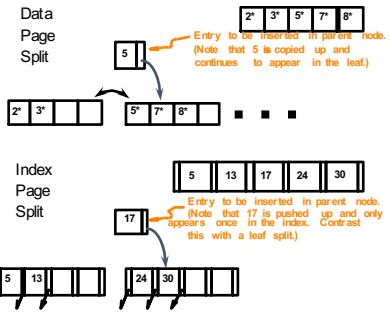
https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html

- One small difference to note:
  - Upon deletion of leftmost value in a node, it updates the parent index entry
  - Incurs unnecessary extra writes

Data vs. Index Page Split

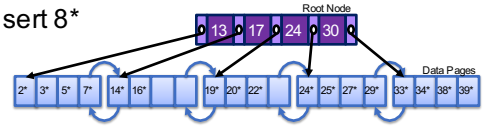
(from previous example of inserting "8*")

- 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.

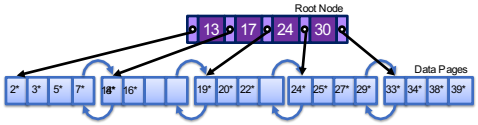


Did we have to split?

Insert 8*

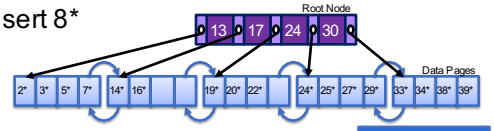


- What else could we do?
  - Redistribute keys?



Did we have to split?

Insert 8*

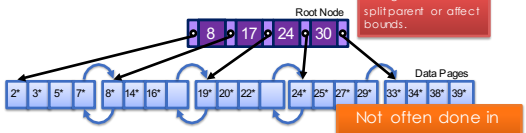


- What else could we do?
  - Redistribute keys?

Do we need to recurse?

No. Why?

Change does not split parent or affect bounds.



Not often done in practice

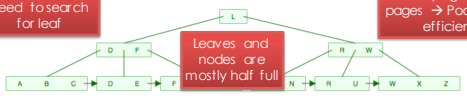
Bulk Loading of B+ Tree

Suppose we want to build an index on a large table

- Would it be efficient to just call insert repeatedly?
  - No … Why not?
  - Random Order: CLZARNDXEKFWIUB

Constantly need to search for leaf

Modifying random pages → Poor cache efficiency



Work through this animation https://www.cs.usfca.edu/~galles/visualization/BPlusTree.ht

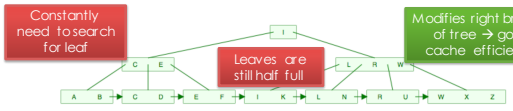
Bulk Loading of B+ Tree

Suppose we want to build an index on a large table

- Would it be efficient to just call insert repeatedly?
  - No … Why not?
  - **Sorterd** Order? ABCDEFIKLNRUWXZ

Constantly need to search for leaf

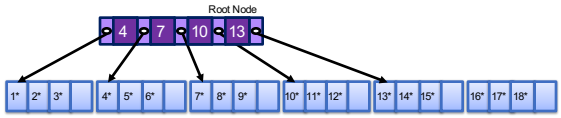
Modifies right branch of tree → good cache efficiency



Work through this animation https://www.cs.usfca.edu/~galles/visualization/BPlusTree.ht

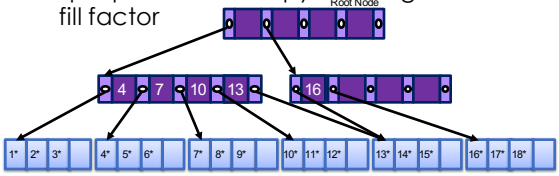
Smarter Bulk Loading a B+-Tree

- Sort the input records by key:
  - 1*, 2*, 3*, 4*, …
- Fill leaf pages to fill factor (e.g., ¾)
  - Updating parent until full



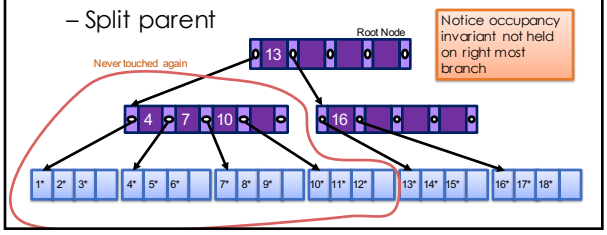
Smarter Bulk Loading a B+-Tree

- Sort the input records by key:
  - 1*, 2*, 3*, 4*, …
- Fill leaf pages to fill factor (e.g., ¾)
  - Updating parent until full
  - Split parent and copy to sibling to achieve fill factor



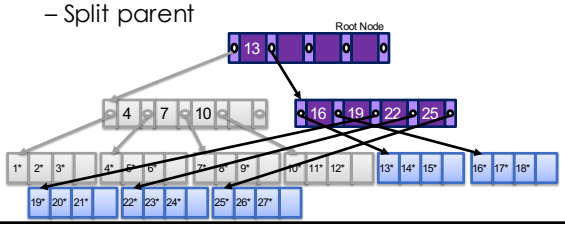
Smarter Bulk Loading a B+-Tree

- Sort the input records by key:
  - 1*, 2*, 3*, 4*, …
- Fill leaf pages to fill factor (e.g., ¾)
  - Updating parent until full
  - Split parent



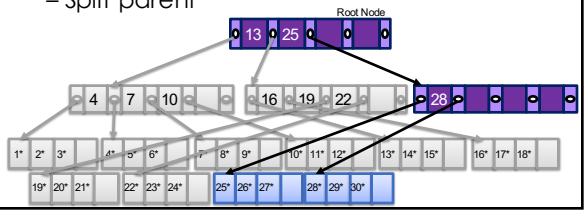
Smarter Bulk Loading a B+-Tree

- Sort the input records by key:
  - 1*, 2*, 3*, 4*, …
- Fill leaf pages to fill factor (e.g., ¾)
  - Updating parent until full
  - Split parent



Smarter Bulk Loading a B+-Tree

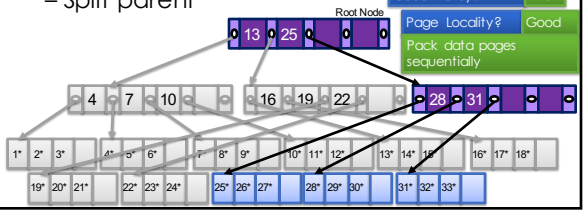
- Sort the input records by key:
  - 1*, 2*, 3*, 4*, …
- Fill leaf pages to fill factor (e.g., ¾)
  - Updating parent until full
  - Split parent



Smarter Bulk Loading a B+-Tree

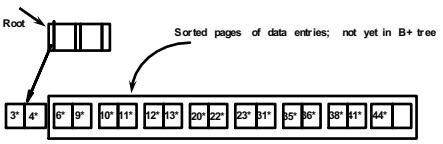
- Sort the input records by key:
  - 1*, 2*, 3*, 4*, …
- Fill leaf pages to fill factor (e.g., ¾)
  - Updating parent until full
  - Split parent

Cache efficient? Yes
Good Policy? LRU
Page Locality? Good
Pack data pages sequentially



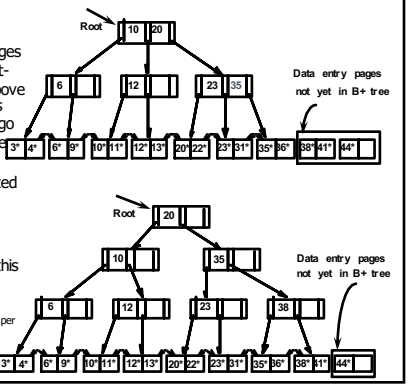
Bulk Loading of a B+ Tree

- Given: large collection of records
- Desire: B+ tree on some field
- Bad idea: repeatedly insert records
  - Slow
  - Also poor leaf space utilization. Why?
    - Play with the web animation: try to make up values that keep leaves full!
- *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 (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.
- Exercise: what kind of buffer pool hit rate will this give you for different policies?
  - Q1: how many references per page?
  - Q1: how often are they re-referenced?



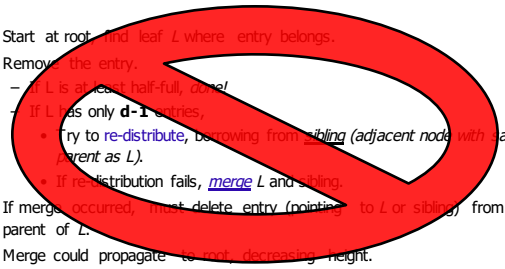
Summary of Bulk Loading

- Option 1: multiple inserts.
  - Slow.
  - Does not give sequential storage of leaves.
- Option 2: *Bulk Loading*
  - Fewer IOs during build. (Why?)
  - Leaves will be stored sequentially (and linked, of course).
  - Can control "fill factor" on pages.

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.

Deleting a Data Entry from a B+ Tree

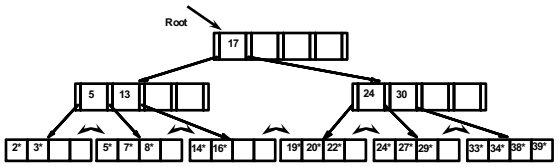


- Start at root, find leaf *L* where entry belongs.
- Remove the entry.
  - If L is at least half-full, *done!*
  - If L has only **d-1** entries,
    - Try to re-distribute, borrowing from *sibling (adjacent node with same parent as L)*.
    - If re-distribution fails, *merge* L and sibling.
- If merge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

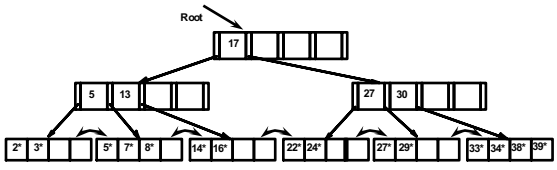
In practice, many systems do not worry about ensuring half-full pages. Just **let page slowly go empty**; if it's truly empty, just **delete from tree** and **leave unbalanced**.

You won't be tested on delete.

Example Tree (including 8*) Delete 19* and 20* …



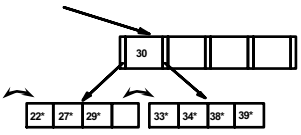
Example Tree (including 8*) Delete 19* and 20* …

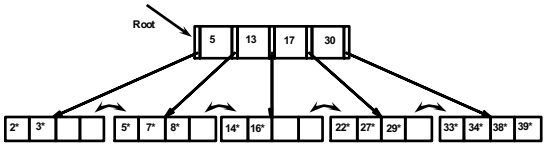


- Deleting 19* is easy.
- Deleting 20* is done with re-distribution. Notice how middle key is *copied up*.

… And Then Deleting 24*

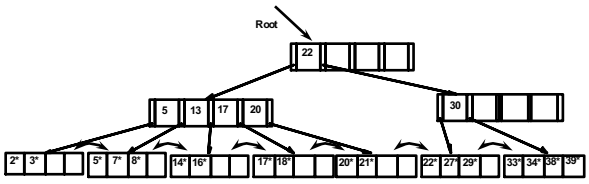
- Must merge.
- Observe `*toss*' of index entry (on right), and `*pull down*' of index entry (below).





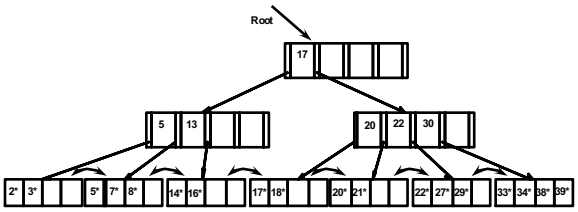
Example of Non-leaf Re-distribution

- 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.



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.

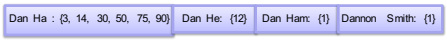


Variable Length Keys & Records

- 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:

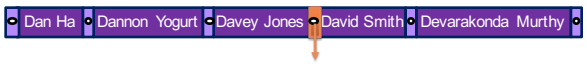


Redefine Occupancy Invariant

- *Order* (**d**) makes little sense with variable-length entries
  - **Variable sized** records and search keys:
    - different nodes have different numbers of entries.
  - **Example:** *single entry might fill entire page*
  - **Index pages** often hold many **more entries** than leaf pages.
  - Even with fixed length fields, Alternative (3) 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.
  - Why?
    - Reduce unnecessary IOs

Prefix Compress Keys?

- How can we get more keys on a page?



- What if we compress the keys:

– Are these the same?
  - David Jones?

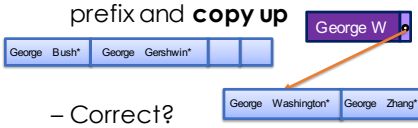
– Not the same! Can't just reduce to prefix.

– But maybe we can fix it?

Prefix Key Compression

- What if we compress starting at leaf:

– on split, determine minimum splitting prefix and **copy up**

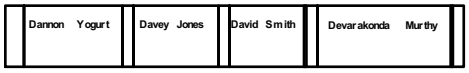


– Correct?

– Can we compress further?

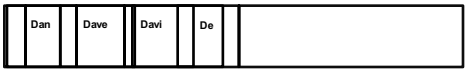
Prefix Key Compression

- Important to increase fan-out. (Why?)
- Key values in index entries just`direct traffic' ; can often compress them.



Prefix Key Compression

- Important to increase fan-out. (Why?)
- Key values in index entries just`direct traffic' ; can often compress them.

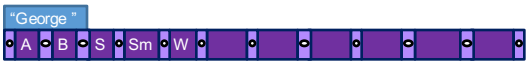


- Is this correct?
- Ensure that each index entry is greater than every key value (in any descendant leaf) to its left.
- In practice we compress upon "copy up" from leaf.

Suffix Key Compression

- All keys have large common prefix

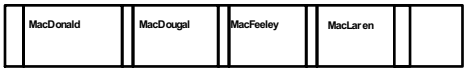
- Move common prefix to header



- Still use full prefix in comparisons
- When might this be especially useful?
  - Composite Keys. Example?
    - <Zip code, Last Name, First Name>

Suffix Key Compression

- If many index entries share a common prefix



- Particularly useful for composite keys
  - Why?

Suffix Key Compression

- If many index entries share a common prefix

- Particularly useful for composite keys
  - Why?

Summary

- 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.
  - Typically, 67% occupancy on average.
  - Usually preferable to ISAM; adjusts to growth gracefully.
  - Caution: if data entries are data records, splits can change rids!

Summary (Contd.)

- Bulk loading can be much faster than repeated inserts for creating a B+ tree on a large data set.
- Key compression increases fanout, reduces height.
- B+ tree widely used because of its versatility.
  - One of the most optimized components of a DBMS.

Architecture of a DBMS

This lecture:

- Finished **tree indexes**: ISAM and *B+-Trees*

Next Lecture:

- Relational Operators