CSE 373: B-Trees

Pete Morcos
University of Washington
4/14/00

http://www.cs.washington.edu/education/courses/cse373/00sp

A new style of tree

- B-trees are unlike the other trees we've seen
  - Data only stored in leaves; interior nodes just for searching
  - Each node has many children (often hundreds)
    - Thus, tree is extremely shallow
- Very important in database systems
- Designed for high performance when managing enormous amounts of data

Disks

- Many databases hold many gigabytes or terabytes (10^{12}) of info
  - Too much to fit in memory
- Disk access time is measured in ms, memory time in ns—about a million times slower
- When disk data is accessed, you read a whole *page* (512 bytes to a few K), not just one byte
  - We'll assume 1000 byte pages for simplicity
- The all-important goal is to **reduce the number of disk accesses!**

Why not just use binary trees?

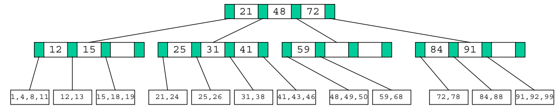
- Recall that in binary search trees, each node visited reduces search space by about one-half
  - Thus, $\log_2 N$ node accesses needed per search
- $\log_2$ 1,000,000,000,000 is about 40
  - 40 disk accesses for each piece of data is unacceptable
- Since we are going to get a whole page of data per disk read anyway, make nodes as big as possible
  - Each node has M children
  - Suppose search keys are strings up to 36 bytes
  - M = 1000 / (36 + 4 *[for a child pointer]*) = 250
  - $\log_{250}$ 1,000,000,000,000 = 5 (as opposed to 40)
- Often, top 2 levels fit in memory, so medium size B-trees only have to hit the disk once (for the actual data node)

Our simplifying assumptions

- Databases can't use pointers in B-trees since some nodes will be in memory, some on disk
- We'll assume everything fits in memory
- Some of our written problems will ask you about 1000 byte disk pages, as on previous slide
- But most examples and homework will use what I'll call a mini B-tree, with M = 4

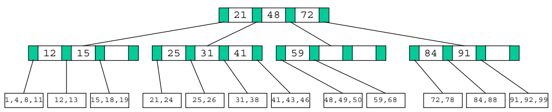
Example B-tree

- B-tree rules:
  - All leaves at same depth, and hold sorted array of data
  - Root has 2 to M children (labeled P_0 to P_{M-1})
  - Other non-leaf nodes have $\lceil M/2 \rceil$ to M children
- Each non-leaf has up to M-1 values (k_1 to k_{M-1}); child P_i holds values $\geq k_i$; P0 holds stuff $\leq k_1$
- Note that values in non-leaves are not actual data!



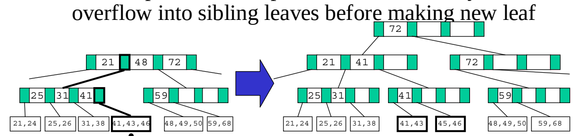
Find

- Similar to binary search trees, but now have M possible choices at each node—O(M) work to pick one, or O(log M) if we binary search the node
- Overall search time is O(log M * $\log_M$ N)



Insert

- Do a Find to pick the right leaf, then add to leaf
- If leaf overflows, split it into two and add new child to parent (e.g. inserting 45 in example)
  - This might overflow parent, so repeat recursively to root
  - Splitting root is only way that tree gets taller
- More sophisticated implementation would try to overflow into sibling leaves before making new leaf



Remove

- Remove item from leaf
- If leaf becomes empty, remove it
  - optionally try stealing data from sibling leaf first
- This might cause parent to have < $\lceil M/2 \rceil$ children
  - Try stealing a value from sibling if possible
  - Else, merge with a sibling—might cause next node up to be too small, so do this recursively
  - If root drops below 2 children, delete it (tree gets shorter)
- Deletion might cause interior nodes to contain values that are no longer in the database (e.g. deleting 41 in example)
  - Not a problem since interior nodes still valid for navigation

Analysis of Insert/Remove

- O($\log_M$ N) steps taken (height of tree)
- Unlike Find, each step might require a rearrangement of a node, which is O(M) work
- Total time complexity is, then, O(M $\log_M$ N)
  - Can rewrite as O([M/log M] log N)

- For large M, worse than binary search trees if everything is in memory, but far better if lower nodes would require disk accesses

Next Week

- Sorting
- An important topic, so we'll spend some time on it
- Read 7-7.3 and 7.5-7.6 (either book)