Lecture 10. B-trees and Cache-Oblivious Data Structures

CpSc 8400: Algorithms and Data Structures

Brian C. Dean

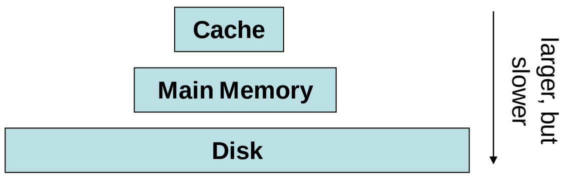
School of Computing
Clemson University
Spring, 2016

Motivation: Databases

- Databases are everywhere, many quite large (so large they can't even fit entirely in main memory).
- For example, when you access nearly any website, there is a database (e.g., MySql) on the "back end" feeding it information.
- How is information in a database actually stored "under the hood" so that we can quickly support:
  - Insertion of new records
  - Deletion of records
  - Fast queries for records or ranges of records based on an appropriate key

Block Memory Transfers

- Most memories are hierarchical in structure:



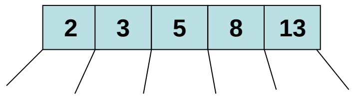
- Between levels, data is often transferred in large blocks (say, 1K at a time).
- Often the true performance of a data structure is determined by the number of blocks it accesses.
- How many disk accesses might be performed by a balanced binary search tree holding 1 billion records?

B-Trees

- The following balancing mechanisms give us O(log n) worst-case running times for all fundamental BST operations:
  - AVL trees (height-balanced)
  - Red-black trees
  - **B-trees**
- B-trees are somewhat similar to the preceding methods, except they aren't binary trees.
- Balancing a B-tree is fairly simple, and doesn't involve so many special cases.
- They are particularly well-suited for data stored on slow block-transfer media.

The B-Tree : Structure

- Each node in a binary tree stores 1 key and each non-leaf node has 1..2 children.
- In a B-tree, each node stores B – 1 … 2B – 1 keys and has B … 2B children:

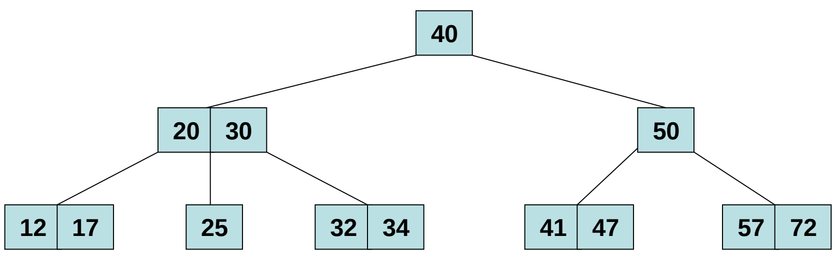


- The root is special, and has no lower limits. It could store a single key.
- B typically chosen so each node fits just right into a block transferred from disk (e.g., B = 1000).

5

The B-Tree : Structure

- All leaves have the same depth, so the total tree height is $O(\log_B n)$.

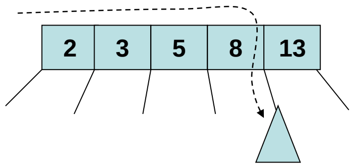


- All operations on a B-tree (e.g., *insert, delete, find, pred*, *succ*, *min*, *max, rank, select*) can be easily implemented in $O(B \log_B n)$ time.

6

Finding an Element

- Scan the root node sequentially to find the appropriate child pointer, then recursively search this child subtree.



- $O(B \log_B n)$ in the worst case.
  - If we assume $B = O(1)$, this is $O(\log n)$.
  - If we have a model where all we care about is block accesses, then only $O(\log_B n)$ time!

7

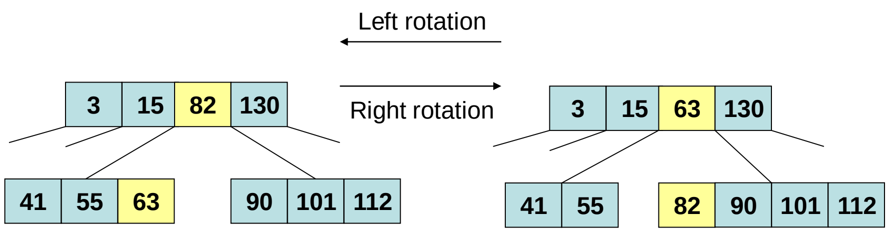
Insert and Delete

- Insertions and deletions always take place in leaf nodes.
- To ensure we delete from a leaf node, we may need to swap first with our predecessor or successor (just like with a BST when we delete a node with 2 children).
- The concern: insertion might make a node too large, or deletion might make a node too small.

8

Sharing With Your Siblings

- Rotations allow us to donate or steal elements from a sibling.

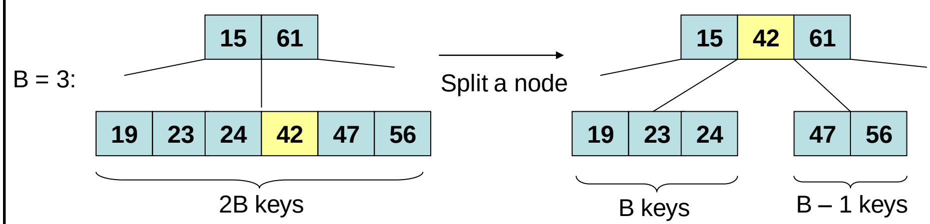


- A rotation can be implemented in O(1) time, although O(B) time is usually also ok.

9

Splitting a Node after Insertion

- Insertion of a new element into a leaf node might give the leaf node too many keys (2B of them).
- If so, we can split the leaf and donate its median element to the parent:

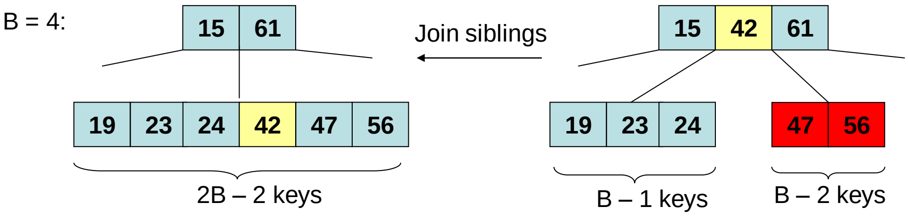


- This might give the parent too many elements, causing it to split, and then the grandparent, etc.
- We could have also donated to a sibling, if possible…

10

Joining Two Nodes After Deletion

- Deletion of an element might give a leaf node too few keys (B – 2 of them).
- Try to fix this by stealing from a sibling.
- If this fails, then we steal 1 element from our parent and join with an adjacent sibling.



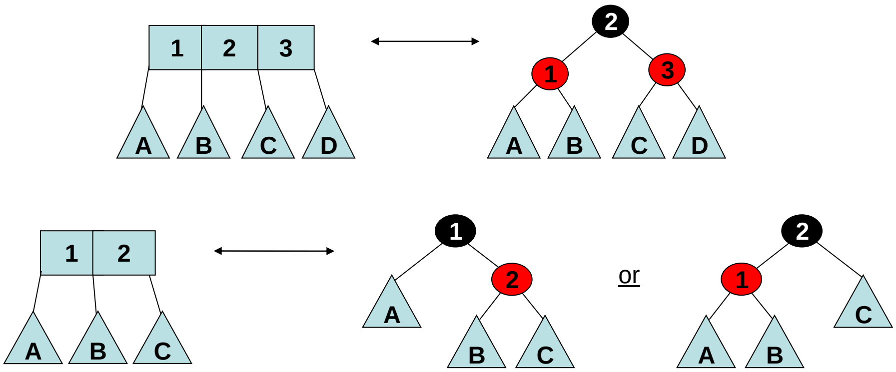
- This might give the parent too few elements, causing it to join with a sibling, and so on up the tree.

Insert and Delete : Summary

- *Insert* and *delete* both take O(B $\log_B$ n) total time.
  – Note that we can easily preserve simple augmented data like subtree heights / sizes during the process.
- *Insert* might result in a chain of splits that propagates up the tree.
  – If the root is split, this is the only case where a B-tree can increase in height.
- *Delete* might result in a chain of joins that propagates up the tree.
  – If the root is consumed by joining its two children, this is only case where a B-tree can decrease in height.

Equivalence with Red-Black Trees

- Interesting bit of trivia – a “2-3-4” tree (B=2) is essentially equivalent to a red-black tree!



13

Revisiting B-Tree Structure

- In a B-tree, each node has <u>B … 2B</u> children.
  (hence B-1 … 2B-1 keys)
- Why not B … 2B-1 children? (B-1 … 2B-2 keys)
  - This extra bit of slack gives us O(1) amortized memory writes per operation!

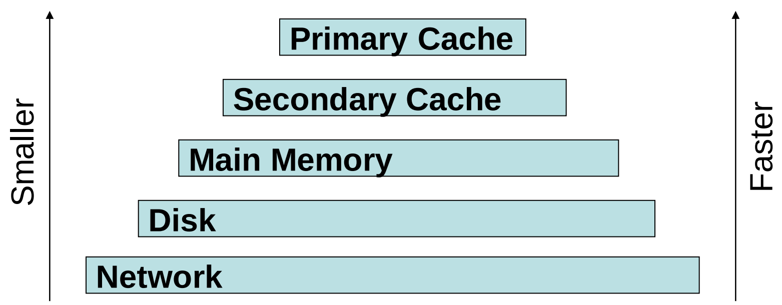
14

Revisiting B-Tree Performance

- Operations on a B-tree take $O(B \log_B n)$ time.
- How does this compare to $O(\log n)$ for other balanced BSTs? Can we make operations on a B-tree run in $O(\log n)$ time?

Hierarchical Memory Layout

- In the RAM model, we assume every memory access takes $O(1)$ time.
- This isn't particularly realistic.
- Most memories have a hierarchical structure:



A Simple 2-Level External Memory Model

- Two layers of memory:
  – Fast cache of size M having M / B blocks of size B.
  – Slower main memory, also with block size B.
- If a memory access reads or writes an element not in the cache, we transfer the entire block (of size B) containing the element into the cache.
  – For simplicity, assume cache is **fully associative**, so each memory block can reside anywhere in cache.
- **Running time = total # of block transfers**.
  – Neglect running time of all other operations (e.g., addition, multiplication, comparison), since these happen in CPU registers and typically run much faster than memory transfers).

17

Block Replacement Policies

- Cache contains M / B blocks each of size B.
- When a new block transferred into the cache, we must evict some existing block.
- For simplicity, we assume an **optimal** (or **ideal**) page replacement policy:
  – Among existing blocks in cache, evict the one that will be used farthest ahead in the future.
  – This is highly unrealistic, but if an algorithm has running time $T(M, B) = O(T(M / 2, B))$ on an ideal cache (i.e., if its performance slows down only by a constant factor when the cache size is halved), then its running time will be $\Theta(T(M, B))$ using either:
    - LRU (least recently used), or
    - FIFO (first-in-first-out)

    page replacement (both common policies in practice).

18

Cache-Aware Algorithms

- If an algorithm or data structure knows M and B, it can optimize its performance accordingly.
- This is called a **cache-aware** algorithm.
- **Example:** Searching a sorted array.
  - Binary search runs in $O(\log n / B) = O(\log n - \log B)$ time. This is not optimal.
  - Using a B-tree (setting B = block size), we obtain $O(\log_B n)$, which is optimal in the comparison model.
- However, we often don't know M and B, and in a multi-level memory system these parameters may vary substantially from level to level…

Cache-Oblivious Algorithms

- An algorithm is said to be **cache-oblivious** if it doesn't know M or B, and yet its running time is always within a constant factor of an optimal cache-aware algorithm.
- **Example:** Reversing a length-n array.
  - Algorithm: scan and swap from both ends inward.
  - This uses n / B block transfers, which is optimal.
- Note: On a multi-level memory with O(1) levels, since the running time of a cache-oblivious algorithm is within O(1) of optimal between each successive pair of levels, it will still be within O(1) of optimal overall!

Cache-Oblivious Algorithms

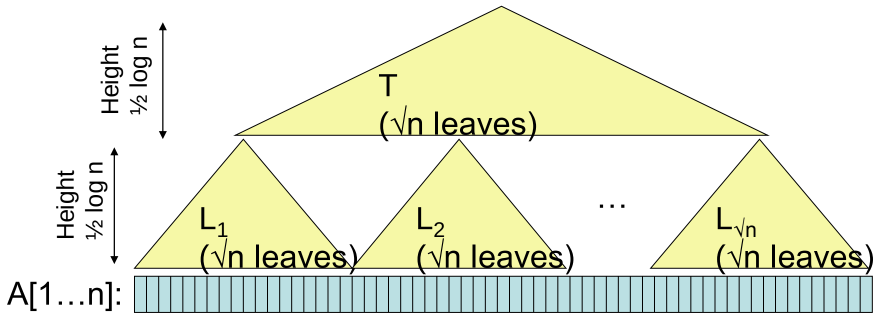
- For many problems, the best solution on a standard RAM is not cache-oblivious…
- Examples:
  - **Searching a sorted array with binary search**. Runs in O(log n – log B) time versus optimal $O(\log_B n)$.
  - **Sorting**. Quicksort / merge sort run in time O((n/B) log (n/B)) time, versus optimal $O((n/B) \log_{M/B} (n/B))$.
- For many algorithm and data structure problems, it is very interesting to ask whether or not we can develop cache-oblivious solutions!
  - E.g., priority queues, dynamic search trees, etc.

Cache-Oblivious Searching

- Consider the problem of searching a sorted array in the comparison model.
- Optimal cache-aware running time: $O(\log_B n)$ using a B-tree.
- Binary search runs in O(log (n/B)) = O(log n – log B) time, so it is not cache-oblivious!
- Can we store a sorted array in memory so that searching can be done in a cache-oblivious fashion, using only $O(\log_B n)$ block transfers? (remember, we don't know B…)

Cache-Oblivious Searching with the van Emde Boas (vEB) Layout

- Build a complete BST on top of our array A[1…n].
- Recursively decompose according to vEB layout:



- In memory, store T followed by L_1 … $L_{\sqrt{n}}$ (each recursively subdivided in the same fashion).

Cache-Oblivious Searching with the vEB Tree Layout

- Recursion effectively stops once we reach subtrees with ≤ B leaves, since each of these fits into O(1) blocks.
- Such subtrees have height Θ(log B).
- So a root-leaf path passes through only Θ(log n / log B) = $\Theta(\log_B n)$ of them.

  (therefore we hit $\Theta(\log_B n)$ blocks)