EECS 2510: Nonlinear Data Structures & C++ Programming
Spring 2016

Programming Lab #4 – Due Tuesday, April 19, 2016 @ 11:59 PM

This lab will move us from memory-based trees, in which every node resides in RAM, to disk-based trees, where each resides as a fixed-size record on disk. B-Trees were designed to trade CPU time for avoiding comparatively "time-expensive" disk operations. This lab will explore that tradeoff.

1) Modify your previous AVL Tree code to work with disk-based records, rather than memory-resident nodes. This means that all child pointers, the root pointer, parent pointers (if you implemented them, but I don't recomment it) will now be of type `int`, rather than `node *`. You will only be keeping up to three nodes in memory at any one time – during the insertion, you will need nodes P and Q to find the appropriate leaf, of which to make the new node a child. During the re-balancing process, you should only need three nodes in memory at once – A, B, and C, as shown in the AVL schematics presented in the lecture slides. You will need to be able to modify F (A's parent), and the nodes between A and the insertion point, but you never need more than three nodes in memory at once.

   Note: you should not have code like `p->key` anymore; you should have `p.key`. Since you are using record (node) NUMBERS rather than addresses (pointers), the first node you create will be node #1 (and will be the first thing you write to the file). There will be no node #0, so use zero as a "null pointer".

2) Code the B-tree. You do not have to implement all seven dynamic set operations. All you *must* have is the Insert and (perhaps) Search routines (and an in-order traversal, but that should be trivial at this point). The value of `t` for your tree (i.e., non-root nodes in your tree will always contain between `t-1` and `2t-1` keys) should be a clearly-labeled constant. Your code should work with any (reasonable) arbitrary value of `t`. I will compile your code with a small value of `t` (perhaps 2 or 3) and run it, and then re-compile it with a larger value (perhaps 10 – 12), and re-run it. Your code will have to adjust the disk record size based on `t`. Your B-Tree code should never have more than three nodes in memory at once. The Cormen pseudocode already conforms to this restriction (note that the code on pp. 492-496 explicitly reads/writes nodes as needed).

For both algorithms, maintain a count of the number of times you read or write a record (i.e., get or save a node). Keep these two counts separate. That way, you can report the number of disk reads, writes, and total disk accesses your algorithm requires. You may find that your code is "read-heavy", "write-heavy", or pretty well balanced between the two.

Bear in mind that the O/S will provide some degree of disk caching for you automatically, which may partially obscure your results. I don't know of any way to get Windows to disable READ caching, but you CAN disable write caching (http://www.sevenforums.com/tutorials/10392-write-caching-enable-disable.html) if you would like to see how much the O/S is helping you out (just remember to re-enable it when you're finished). Also, make sure your tests are run on a physical hard drive, rather than a USB flash drive or an SSD. You may also want to experiment with C++'s `fflush()` function to force flushing of write operations. Check the documentation.

For both trees, report the final height of the tree, as well as the total number of nodes. For the B-Tree, report the loading factor – the number of keys used / the total number of keys available. Also report the total file sizes of the two trees' files.

The longest word in Shakespeare's plays is "**Honorificabilitudinitatibus**", which is 27 letters long. You will, therefore, need to make your `char[]` array for each key AT LEAST 28 bytes long. I recommend either going with something a bit longer (perhaps 32?), or explicitly checking the length of each string to confirm that it fits before you add it to the tree.

You will need to support "re-inserting" an item that's ALREADY in the tree. In order to accomplish this, simply add an integer field called `count` to each node, like you did for the BST program. When you attempt to insert a string into the tree, if that string is already found, then increment the `count`; if the item is not found, then insert the item as usual, setting the new `count` to 1.

You will need to modify your `InOrder` traversal code for each tree such that it computes the total number of items in the tree, both unique, and with duplicates – i.e., you will produce a count of the number of *distinct* items in the tree, and also the number of items counting duplicates (this will involve the `count` field). For example, there may be 100,000 *total* words, drawn from a total vocabulary (nodes in the tree) of 10,000. THOSE counts should match across both trees.

Finally, you will read the `Shakespeare.txt` file two more times (one for AVL, and then again for B-Tree), and place the individual WORDS into the tree. This can be done with the code from the AVL_RBT_BST assignment. Use the `clock()` function to report the total run time of each method (how long AVL took, and how long B-Tree took, so run one all the way through, and then the other; don't make one pass through the Shakespeare file and feed both trees simultaneously).

Write up a one-page report (more if you like, but 1 page minimum) that details your findings, comparing the results of the performance between AVL and B-Tree (using various values of `t` for the B-Tree) on the Shakespeare file. List what you expected, what you found, any surprises, any particular observations, etc. A graph that shows some performance metric as a function of `t` might be helpful, but don't let putting a graph on the page be a substitute for a full page's text to explain what you found.

Submit to Blackboard a 7-zip (preferred) archive of your entire Visual Studio workspace, along with your report (.doc/docx or pdf).

If you have questions or run into problems, contact me. Don't use code from ANY source other than the lecture slides or the Cormen text, even as a reference. Don't use any STL components – you should be able to write all of this with structs and the built-in language elements (like arrays).