Christian Ropchock

EECS 2510 Non-Linear Data Structures

03-23-18

AVL/BTree Comparison Report

**Introduction**

There are two different ways that exist in C++ that are used to store pointer references in Search Tree nodes. The first way is to use a memory allocated pointer reference in some array that can be used to reference nodes, which in turn uses up memory from RAM during the execution of the code. The second method involves Disk File I/O which stores the nodes of the tree to a Disk File, and uses this Disk File for read/write operations. This lab compares the Disk I/O operations that two different Search Trees have, and compares these results to check which tree operates better with Disk I/O operations.

**Expected Outcomes**

Every time that a child pointer is taken down any tree, it constitutes a disk access. Disk accesses take more time to perform then main memory accesses. This is why B-Trees are implemented, because they guarantee a smaller height then any of the other Search Tree data structures. With the smaller height in the B-Tree data structure, this represents the number of Disk I/O operations which are used to update nodes when they are accessed in the execution of the code. The B-Tree uses something known as a Branching Factor that determines the number of keys that each node is allowed to hold at one given time. Traversing an array of keys is similar to performing memory access, in which Rotational Latency will play a big role due to different elements of the array existing in different locations on the disk.

For the B-Tree, there is going to be a huge trade-off on Disk Space needed to allocate for larger nodes due to larger values of t, and the height of the tree. Depending on the total amount of memory allocated for the B-Tree on Disk, it may or may not be a suitable data structure to use. Since Disk I/O is slower then main memory access by a factor of 100,000; it is optimal to try and reduce disk accesses as much as possible.

**Results from Testing**

In the practical sense of this experiment, there is huge overhead reading nodes to and from disk files; this experiment is more for looking at what goes on behind the scenes of these operations. For different values of t, it is expected that the height of the B-Tree should go down, while the file size used to store the tree increases, which is demonstrated in the table below.

|  |  |  |  |
| --- | --- | --- | --- |
| **Value of t** | **Height of Tree** | **Disk File Size** | **Loading Factor** |
| 2 | 13 | 2821359 B | 73.62% |
| 5 | 7 | 2115846 B | 87.65% |
| 12 | 5 | 1932320 B | 94.53% |
|  |  |  |  |
|  |  |  |  |

**Figure 1. BTree Results on Shakespeare.txt (4.819 MB) with different loading factors**

As one can see from the results above, the opposite of my expectations for the Disk File size have occurred. This resulted because I didn’t account for the node allocation, with higher values of t, there are less nodes that are created for the Disk File. Each node has to allocate memory for the array of keys, array of child pointers, array of correlated counts, leaf flag, and n. So, the overhead for creating more nodes triumphs the overhead needed to store larger nodes with more keys. Another aspect of the BTree is the loading factor. It is noted that as the branching factor goes up, the loading factor goes up as well. This is due to the fact that each individual node created creates 2\*t-1 keys for the node, but when it splits, each individual child node it splits off into contains t – 1 keys, but still has the occupancy for 2\*t-1 keys.

It is also known that with larger values of t, the number of disk reads should be less than that of smaller values of t. This is because, as one traverses down a child pointer in the tree, this counts as a Disk File I/O operation, which requires reading in a key to be able to access the next pointer. A different file was chosen for testing this claim, and the results are shown as below in the form of a graph that shows growth in disk reads as t becomes larger.

**Figure 2. Disk Writes and Disk Reads for increasing Branching Factor for BTree**

As one can see from the data above, the disk writes and disk reads decrease for larger values of t. The reason behind this, is that for larger branching factors, the height of the tree becomes smaller, thus there are less Disk I/O operations needed to traverse through the tree. The reason why Disk Reads decrease at a sharper rate is that Disk Reads need to read in each individual node for Disk I/O, in which disk write only needs to traverse the pointer paths.

In terms of comparing performances between the AVL Tree and BTree for Disk File I/O, one can look at the heights of the tree as well as the Disk Read/Writes that are performed on each tree. These two data structures are compared against each for performing their Insert methods on three differently sized files. The results are as shown below…

|  |  |  |
| --- | --- | --- |
|  | AVL Tree | Btree |
| File 1 (1.79 MB, t = 3)) |  |  |
| Disk Writes | 416768 | 350092 |
| Disk Reads | 6732168 | 2058303 |
| File Size | 837.07 KB | 1.30 MB |
| Height | 17 | 9 |
| File 2 (4.82 MB, t = 2) |  |  |
| Disk Writes | 1053927 | 975466 |
| Disk Reads | 18528134 | 7830411 |
| File Size | 1.517 MB | 8.81 MB |
| Height | 18 | 13 |
| File 3 (30.1MB, t = 3) |  |  |
| Disk Writes | 6344637 | 6027401 |
| Disk Reads | 134732020 | 42842939 |
| File Size | 4.47 MB | 6.91 MB |
| Height (31.1 MB) | 20 | 10 |

**Figure 3. AVL vs. BTree Comparison Results**

As one can see, B-Tree outperforms AVL in many of the categories. For instance, a B-Tree requires less Disk Writes as well as Disk Reads then AVL, and B-Tree was also still set on a relatively low branching factor for all three file size tests. The reason for this is that a B-Tree will always have more than a 2-way branching factor, in which the individual nodes can store more child pointers and keys. Since there aren’t as many new levels needed to be created for the B-Tree, it also triumphs the AVL tree in height every time. If one were to take away the overhead time from reading and writing each individual node to the file in the code, then the B-Tree would seem to outperform the AVL tree according to this data, because it does more memory searches, rather than Disk I/O searches which requires more time. The only trade-off is that the B-Tree seems to require more storage capacity to maintain it on a Disk, so, if the correct amount of storage is available, then the B-Tree would have the better performance overall.