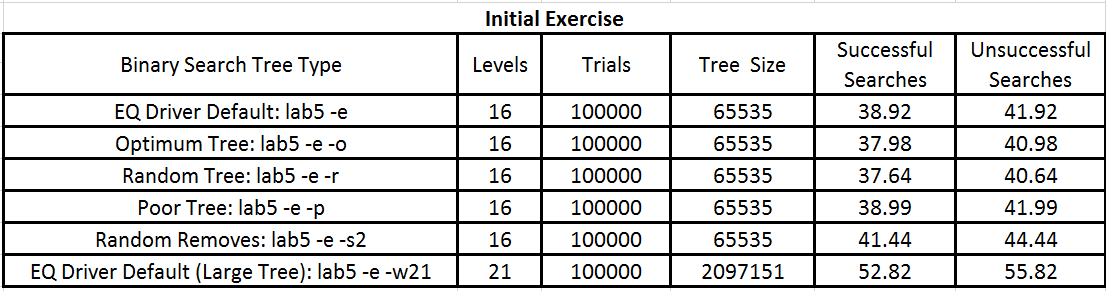
Austin Johnson

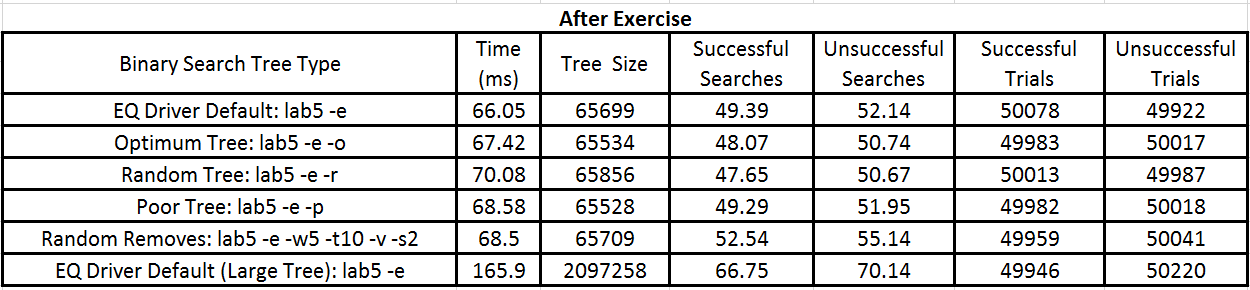
25 October 2016

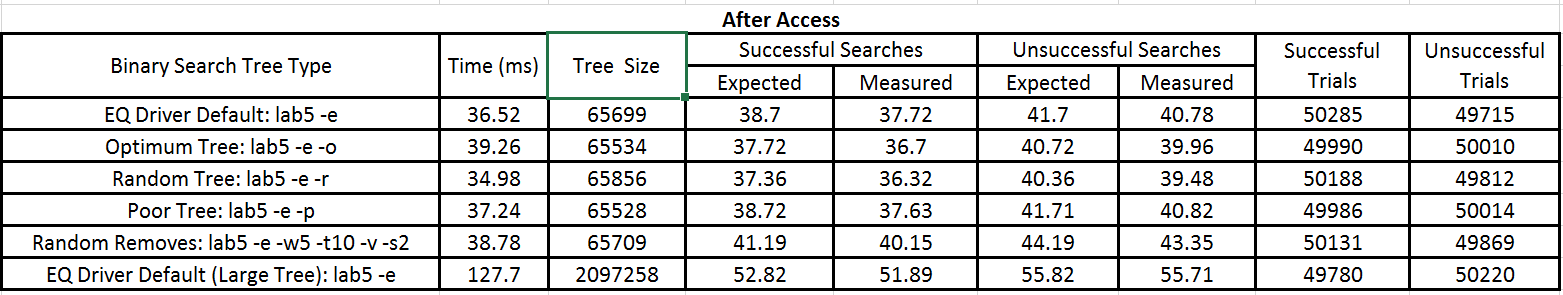
C03603497

MP5

Performance Evaluation







For the initial exercise data, the amount of successful searches versus unsuccessful searches appears to be pretty even. There were always more unsuccessful search comparisons due to having to check each key in the tree versus finding a match and stopping. For the data after the exercise, the random tree appeared to take the most time and produced the largest tree size. However, it took the random remove design the longest to search for a given key. I expect this is due to having a poor tree from all of the random removes it takes. When accessing a key after the exercise, the driver default and poor tree performed almost identical. The only notable difference was that the default produced more successful trials over the poor tree design. The poor tree design averaged a higher search number than the optimum and random tree, making it the worst design out of the three (which is expected). The optimum tree did not produce better results over the random tree. It actually took more searches on average than the random tree, making the random tree the best out of the three implementations. The random removes tree was the worst out of all of the data collected (disregarding the large tree). My implementation performed pretty closely to Standish’s implementation of comparing keys. His implementation appears to always expect around 1 more key comparison than I measured. That appears to happen on every tree implementation. My implementation follows the successful search time complexity of O(log n). I implemented an iterative access solution that goes to the left subtree if the key is less than the current and goes to the right subtree if the key is greater than the current. In the worst case of a successful search, it has to go all the way to the leaf. This is proportional to the length of the longest path from the root to the leaf. This is equivalent to the time complexity O(height of the tree) which is O(log n).