B-tree Insertion: The Empirical Analysis

Christopher Roadcap, Nahesha Paulection, Ben Lehman

Introduction

B-trees are self-balancing structures that retain sorted data to be searched, inserted into, and deleted from. With a time complexity of $O(\log n)$, the purpose of this empirical study is to implement a version of B-tree insertion and analyze our implementation run-time efficiency.

Methods

For this experiment, we designed a B-tree using Java and assumed only positive unique non-zero integers could be inserted. For this implementation our order was 5. To implement the B-tree we used an object called pair that held a max key and a node that held the children. The basic premise was that each pair had a key called maxKey and its corresponding children are keys lesser than that. New keys that are greater than any available maxKeys are inserted into the right-most.

A general rule of B trees is that new keys must always be inserted at the leaf node. A node is a leaf node if and only if each key associated with that node does not have any children. Root is indicated when a pair does not have a parent. To find the correct leaf node for a new key to be inserted in, starting from the root, the new key is compared to its maxKeys. Traversal of levels occurs when the new key is less than a maxKey. Once traversal of a level happens, the checking of whether the node is a leaf occurs. If the node is not a leaf, traverse down the appropriate path. If this node is a leaf and not full, then we simply insert the new pair. If the node is a leaf and is full, we find and promote the middle pair to its parent. Split the current node so that keys less than the promoted key is within its children, while the keys greater are stored in the

right most. Repeat the conditions until the tree is balanced. When root becomes full, a new node takes the middle pair splits the old node as described before and the new node becomes root. The children affected by the change of root is corrected and balanced as well.

The design type of the B-tree is space-time tradeoffs. It is space-time tradeoff because the algorithm consumes more memory by storing a pointer to all of the relevant data for that given node; thus making data access shorter.

Results

For the empirical analysis of our implementation of this algorithm, we generated 10 sets of varied unique integers and timed our b-tree's ability to balance itself. Each varied set ran 50 trials. The average of each set was plotted in figure 2.

Figure 1. Tabular form of Average Runtime

Number of Inserted	Average Runtime
10000	20.14
20000	34.72
30000	52.40
40000	70.64
50000	90.7
75000	150.28
100000	243.02
250000	751.16
500000	1493.22
1000000	2127.92

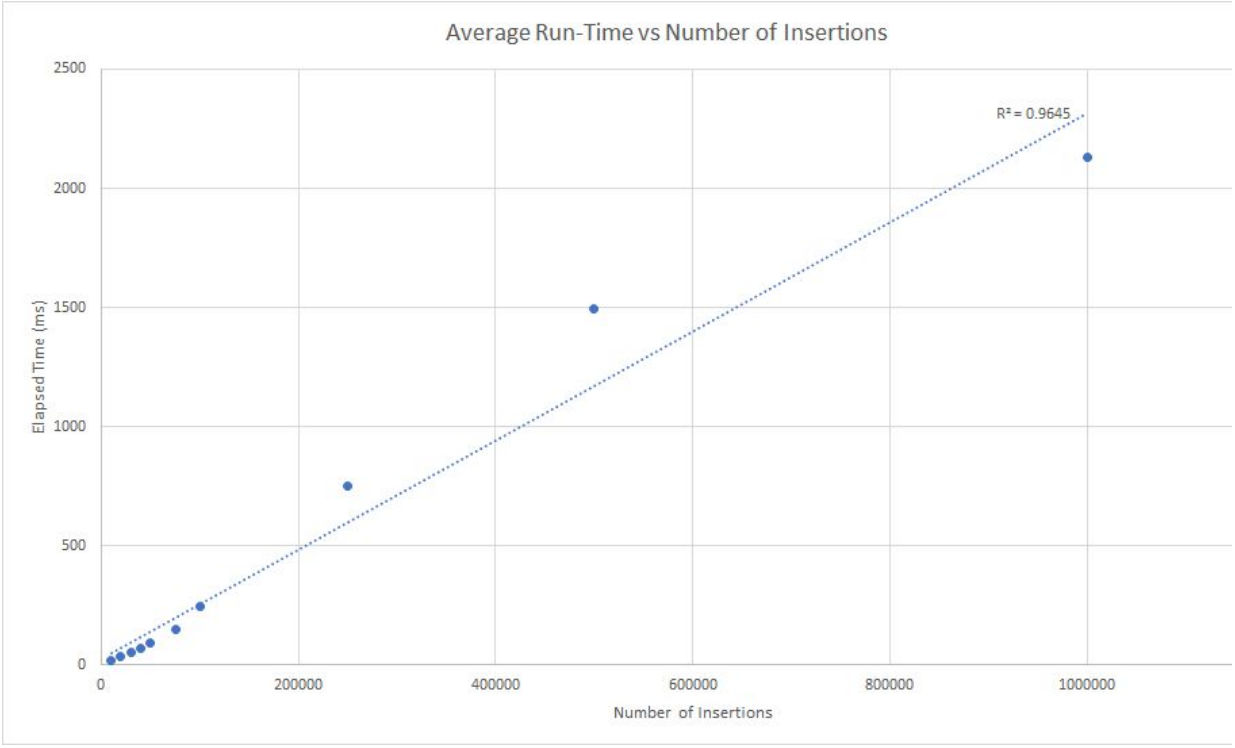

Figure 2. Graphical display of Number of insertions vs Average Runtime(ms)(R^2 = .9645)

Discussion

After examining the tabular data, the order of growth for our implementation of B-tree insertion is *O*(n). The insertion of 10,000 numbers average time was .002014 ms while the insertion of 1,000,000 numbers average runtime for one number was .002127 ms. This indicates that as the amount of inserted numbers increased the runtime stayed nearly constant; and because the R^2 value is extremely close to one, we can say with confidence that the average data obtained for each insertion size is linear.

We should note that the average runtime for 1,000,000 was lower than 500,000 insertions. This could be an indicator that our order of growth is actually *O*(logn); However, a major problem during our experiment occurred when we were collecting data for bigger insertion sizes. Our machines began to run out of usable memory when insertion sizes started to approach 1,000,000 data points. We believe the slowdown we experienced with these larger sizes is associated with the Java garbage collector. This would be something to explore in the future.