# Findings about the behaviour of Binary Search Trees

Binary search trees organise data in a way such that data retrieval could be done in O(log(n)) time. Each node in a binary tree contain data and two child pointers. A BST is constructed such that left child is smaller than the parent, and the right child is bigger. In my implementation of BST, each node has a key used to compare nodes, and a linked list of values. A linked list is used to store values that has the same keys.

In a BST, all nodes in the left branch of a node is smaller, and all nodes in the right branch is bigger. And in a balanced BST, the number of nodes in left branch would be the same as the number nodes in the right branch for any node. In a balanced BST, each time the search algorithm compares a node with its search key and find that it is different, it would be able to reduce its search space by half. Because the target node could only either be in the left branch or right branch.

However, if the BST is not balanced, in the worst case each node only has one child, the search algorithm can only be guaranteed to reduce the search space by one for each comparison.

So, for a completely balanced BST, the number of comparisons needed to find the linked list of values corresponding to the search key is Θ(log(K)) or O(log(n))| K = #unique keys, n = #values.

For an unbalanced BST, the complexity is O(n), although its exact performance depends on how unbalanced the BST is.

In theory, if nodes were inserted in order, if random data were inputted, the complexity should be Θ(log(K)). Because the BST should be balanced for random data. Where for sorted data, the complexity would be Θ (k) or O(n), with #comparisons ~= n/2. Because all nodes would only have one child that is either larger or smaller. (Depending on how the data was sorted). And a random value would take n/2 comparisons to be found.

If the search term is the value rather than the key, each comparison doesn’t give the search algorithm any information about where the target value is. The algorithm needs to traverse to every node in the BST and loop though the linked list for each node in order to find all values that matches. So, in theory the complexity should be O(n), with #comparisons = n.

## Stage 1:

**Number of comparisons needed**

|  |  |  |  |
| --- | --- | --- | --- |
| Data size (n) | Random Data | Sorted Data | Random Sorted Data (sort -R | Data is grouped by keys, but the keys are shuffled) |
| 200 | 10.2 | 94.1 | 9.16 |
| 400 | 11.43 | 186.867 | 9.73 |
| 800 | 12.1 | 394.433 | 10.87 |
| 1600 | 12.86 | 668.167 | 12.33 |
| 3200 | 15.13 | 1461.67 | 14.07 |

Number of comparisons is calculated by taking the average number of comparisons needed to search through 30 random keys.

## Stage 2:

**Number of comparisons needed**

|  |  |  |  |
| --- | --- | --- | --- |
| Data size (n) | Random Data | Sorted Data | Random Sorted Data (sort -R | Data is grouped by keys, but the keys are shuffled) |
| 200 | 200 | 200 | 200 |
| 400 | 400 | 400 | 400 |
| 800 | 800 | 800 | 800 |
| 1600 | 1600 | 1600 | 1600 |
| 3200 | 3200 | 3200 | 3200 |

## Discussion

Experimental results for stage 2 exactly matched theory, #comparisons = n.

Results for stage 1 closely matched theory. For a random data set, #comparisons grew at a logarithmic rate, just always being a bit higher than the ideal case of a perfectly balanced BST - O(log2(n)). This makes sense because a random dataset would not construct a perfectly balanced BST.

#comparisons grew at a linear rate, O(n), for sorted data.

#comparisons grew at a logarithmic rate for randomly sorted data. This makes sense because the keys are still random, despite the fact that all values with the same type are grouped together. #comparisons would still closely match the case for random data.

Unfortunately, real life cab data are sorted. To improve the O(n) search time, the data could be scrambled before it’s inserted. This would increase the initial loading time, but the search time would improve to O(log(n)). This trade-off would be favourable. However, every time new data comes in, the total data set needs to be shuffled and reinserted into a new BST. This leads to an insertion time that grows with n. A solution would be to rebalance the BST every time a new node is inserted. This involves swapping sub-branches of the BST but would ensure a perfectly balanced BST and better insertion time.

## Graphs