# 1. Introduction

This essay will primarily focus on the structure of *Binary Search Trees (BST)*, a relatively complex data structure which has applications in database implementations. Although the utilization of BSTs is varied, this essay will specifically look into *optimizing* such trees for *better long-term performance*. One possible way to optimise a BST would be to perform *node rotations* – the process of changing the structure of the tree without interfering with the order of the elements – in order to achieve a *balanced tree* – a tree where the number of edges from the root to the deepest leaves varies by a maximum of one. Given a populated original tree, the *time complexity* required to balance the tree and then perform searches within the tree will be investigated in comparison with performing searches within the tree directly. Time complexity is a universal term used to characterise the amount of time taken for an algorithm to run given a set of input values of a certain size. Hence, the question emerges: How does re-balancing a Binary Search Tree using the Day–Stout–Warren algorithm followed performing leaf searches within the tree compare in terms of time complexity with directly performing the leaf searches within the original tree. Although I do not take IB Computer Science, this RQ links to Topic 5 of the Higher-Level syllabus.

# 2. Relevant Theory

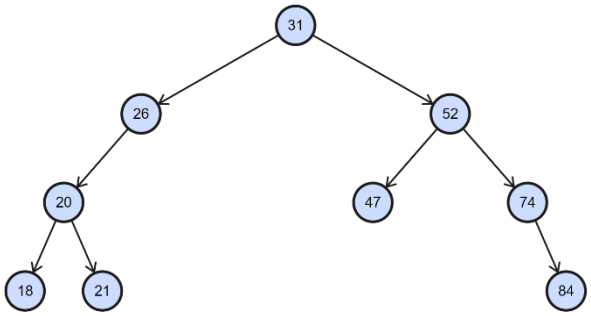
## 2.1 Binary Search Trees

A binary search tree (BST hereon) is a data structure with a defined behaviour that stores comparable "items" (a general catch-all term used to describe structures such as numbers or strings), commonly referred to as *nodes*. The reference to the word "binary" suggests that the structure is comprised "of two things". In the case of BSTs, it refers to its limitation by definition that each node can point to maximum of two other nodes within the tree, commonly referred to as children. A node's children can be distinguished through the fact that one is referred to as being a left child while the other is referred to as a right child. Even if a node has only one child, it is still referred to as a left or a right child, with the opposite child pointer being null. Furthermore, a node can also be considered to have no children if both pointers are null.

Because the two child nodes are comparable, the left child of the node will always have a value 'less than' the parent node which means the right child must have a value that is 'greater than' the parent node. To insert a node into its correct position in the tree:

1) If the tree contains no root node – a node that represents the top value of the tree – set the node to be inserted to be root
2) Create a pointer to the root node
3) If the root node is not null, compare the item of the node to be inserted to the value of the pointer. If the value of the node is smaller than the pointer's value, change the pointer to the left child and vice versa if the value is greater
4) Repeat step three until the pointer points to null. Insert the node at the current position and set the correct relationship with its parent node

An example of a correct Binary Tree is shown in the Figure 2.1.1 below:




The implementation of “search” within the structure name of “Binary Search Tree”, illustrates the Binary Search Tree’s main purpose: searching for a specific node. Inherently, due to the organization of the nodes within the tree, after each search operation the number of remaining possible nodes ideally halves, making the data structure is very efficient in comparison to other types of structures.

| | Average | Worst Case |
|---|---|---|
| BST | $O(\log(n))$ | $O(n)$* |
| Array | $O(n)$ | $O(n)$ |
| Stack | $O(n)$ | $O(n)$ |
| Que | $O(n)$ | $O(n)$ |
| Doubly-Linked-List | $O(n)$ | $O(n)$ |

Table 2.1.1 Time complexities of searching structures containing *n* elements

*: BST degraded into a Linked-List resulting in a sequential search where the number of remaining possible nodes deceases by 1

Graphing average time complexities within these structures to avoid BST degradation, as shown in Figure 2.1.2, we can see a massive difference in time efficiency:

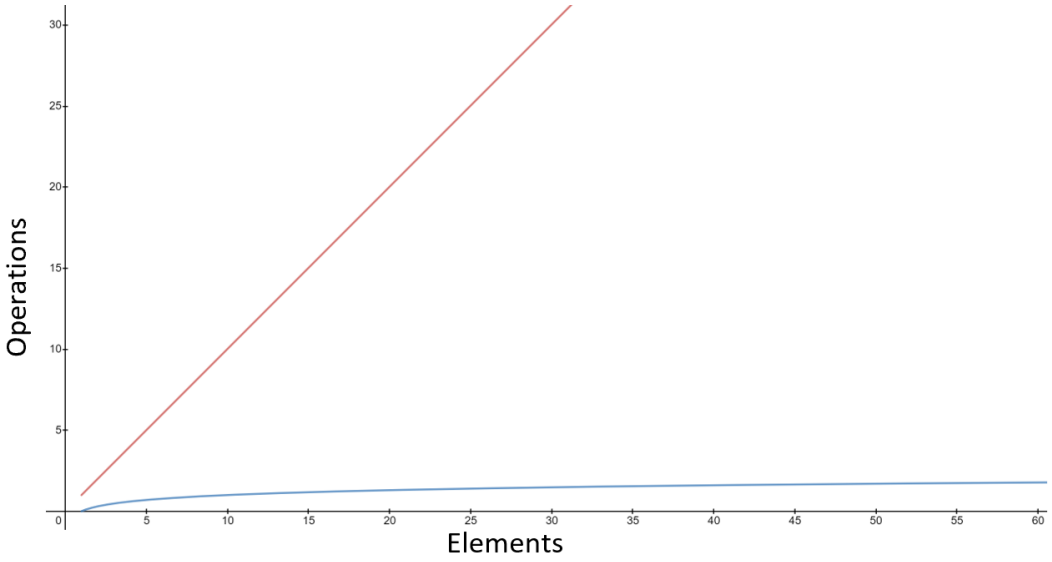



Figure 2.1.2 A graph relating the number of operations (y-axis) to the number of elements within that data structure (x-axis)

Searching a binary tree presents multiple possible alternate methods such as depth-first-search (DFS) and breath-first-search (DFS). Table 2.1.2 compares the best-case, average and worst-case scenarios of such algorithms.

| | Best Case | Average | Worst Case |
|---|---|---|---|
| Binary Search | $O(1)$ | $O(\log(n))$ | $O(\log(n))$ |
| DFS | $O(1)$ | $N/A$ | $O(\|V\| + \|E\|) = O(b^d)$ |
| BFS | $O(1)$ | $N/A$ | $O(\|V\| + \|E\|) = O(b^d)$ |

Table 2.1.2 Time complexities of search algorithms

Note: $|V|$ is the number of vertices and $|E|$ is the number of edges in the graph. This can be rewritten as $O(b^d)$ where *b* is the maximum path length and *m* is the maximum path length.