# 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 node searches within the tree compare in terms of time complexity with directly performing the node 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

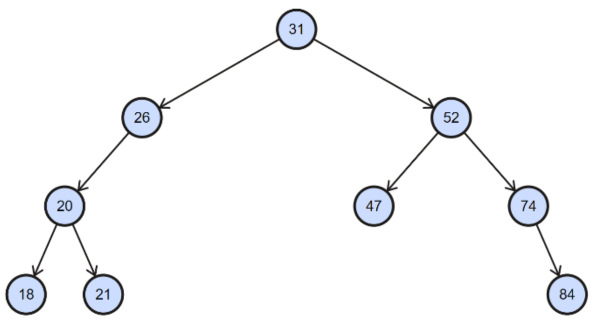
## 2.1 Binary Search Trees

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:

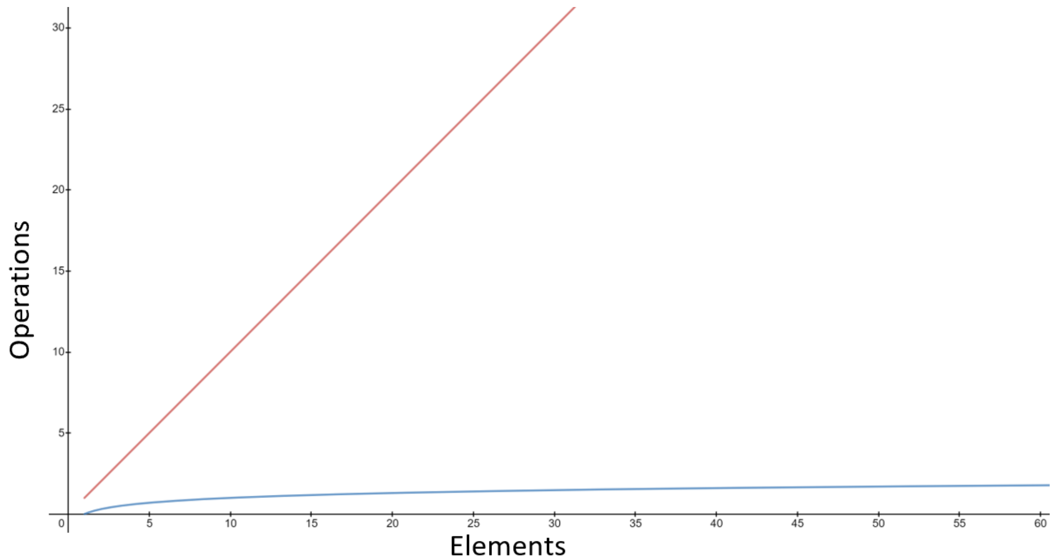



Figure 2.1.2 A graph relating the number of operations (y-axis) to the number of elements within that data structure (x-axis)

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.

Both the DFS and BFS methods will consistently yield their worst-case efficiency as they have a predefined behaviour for checking the BST as shown in Figure 2.1.3. BFS is a vertex-based technique that uses a queue data structure that follows a first-in-first-out methodology, meaning that the entirety of a vertex is visited and stored before moving on the adjacent vertex. This differs from DFS which is an edge-based technique that uses a stack data structure – first visited vertices are pushed into a stack continuously until when there are no more vertices left to visit at which point the stack is popped. For the purpose of this essay, the binary search method will be used to search the binary search tree because it uses a relatively low number of comparisons and a constant $O(1)$ space. While the efficiency in Figure 2.1.2 seems convincing, as previously mentioned it can degrade into $O(n)$ time complexity. Consider the insertion of the values: 1,2,3,4,5 in this order into a tree. This will produce the binary tree in Figure 2.1.4. The tree has a noticeable resemblance to a linked list because every node carries a

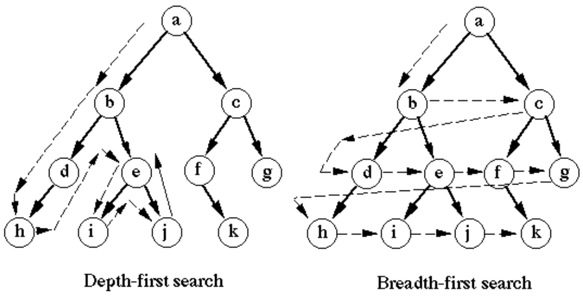



Figure 2.1.3 Depiction of BFS and DFS algorithms

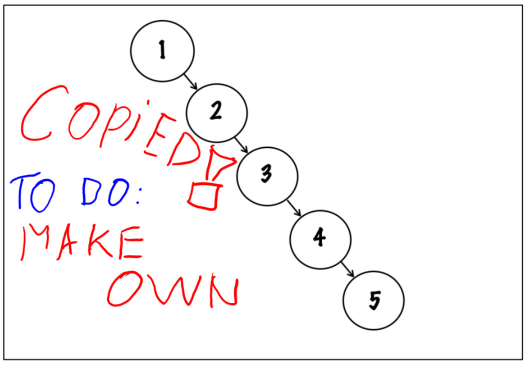



Figure 2.1.4 An unbalanced Binary Tree

pointer to only one child. If one was to perform a search for the node with the value ‘5’, the algorithm would be $O(5)$, the exact same as a linear search. This example uses a small unbalanced tree, but if a tree contained 1 million nodes, a similar structure to Figure 2.1.4, would require up to $O(1{,}000{,}000)$. This is significantly more when compared to the worst-case of $O(20)$ if perfectly balanced, thus demonstrating the need for balancing a tree in order to optimise search efficiency.

Now, the BST algorithm will be looked at more closely. The **insert()** and **insertRoot()** functions are shown below.

```
47    public void insertStartRandom(BinaryNode currentNode, Item newItem, int depth) {
48        depth++;
49        if(root == null) {
50            insertRoot(newItem);
51            return;
52        }
53        if(Integer.valueOf(currentNode.item.toString()) > Integer.valueOf(newItem.toString())) {
54            if (currentNode.left != null) {
55                insertStartRandom(currentNode.left, newItem, depth);
56            } else {
57                currentNode.left = new BinaryNode(newItem, currentNode, depth);
58                if (depth>maxDepth) {
59                    maxDepth = depth;
60                }
61                depth = 1;
62                return;
63            }
64        }
65        else if(Integer.valueOf(currentNode.item.toString()) < Integer.valueOf(newItem.toString())) {
66            if (currentNode.right != null) {
67                insertStartRandom(currentNode.right, newItem, depth);
68            } else {
69                currentNode.right = new BinaryNode(newItem, currentNode, depth);
70                if (depth>maxDepth) {
71                    maxDepth = depth;
72                }
73                depth = 1;
74                return;
75            }
76        }
77    }
78
79    public void insertRoot(Item newItem) {
80        root = new BinaryNode();
81        root.parent = null;
82        root.item = newItem;
83        root.height = 1;
84        maxDepth = 1;
85    }
86
```

Figure 2.1.5 BST insert() and insertRoot() function

As seen in Figure 2.1.5, the insert() function applies a recursive approach to inserting the nodes in their appropriate spot within the tree, with the node object being named BinaryNode. The function takes a parameter currentNode which refers to the current node's item. This is compared to the value of newItem, the item desired to be inserted within the new node. The depth parameter is used to assign to the BinaryNode object it's depth and increments by one each time the method is recursed. This method is always called using the NodeFactory's root class instance variable, which initially is null, prompting the if statement in line 49. This invokes the insertRoot() helper method which creates a null BinaryRoot object and assigns it it's item, height and updates class instance variable maxDepth to reflect a single root node. If the root exists, the code progresses and compares the value of the item to be inserted to the currentNode (which always is initially in the original call the root, which has been populated). If the comparison between the currentNode's item or the item to be inserted yields that the currentNode has a greater value, the code will progress to the left child in the if statement containing lines 53-64. Upon entering, if the left child is null, the insert() function calls itself with the left child as the currentNode. Similar manipulation applies if the initial condition in line 53 is not met meaning that the opposite condition – newItem's value is greater than currentNode's value – as seen in line 65's else-if. Either conditional will enter and have the insert() method recurse and increment the depth counter until the child where the comparison between currentNode's item and newItem (conditions in line 53 and 65) yields a child node that is null, triggering either the else in line 56 or 68 based on the comparison between the item values. At this point, a new BinaryNode object is created as the left/right child of the currentNode that contains the item of newItem and appropriate depth. The currentNode is also passed as a parameter when creating the object such that every node other than null also has a pointer to its parent. Using this insert method can yield an unbalanced binary tree of infinite size.

## 2.2 Day–Stout–Warren Algorithm

The Day–Stout–Warren (DSW) algorithm is designed to efficiently balance a BST, reducing its height – the number of nodes in the longest path from the root to a leaf (inclusive) – to $O(\log(n))$ nodes. In other words, it creates a perfectly balanced three with a height of $\log_2(n)$. It was designed by Quentin F. Stout and Bette Warren in their 1986 paper based on previous work by Colin Day in 1976. The algorithm runtime complexity is linear to the number of nodes in the tree and the space complexity is constant. The DSW algorithm consists of two phases: first an initial 'Tree-to-Vine' procedure reconfigures the original tree into a sorted vine, similar to a linked-list, where all roots are the right child of the parent with the nearest greater value. This completely unbalanced tree is then reconfigured using the 'Vine-to-Tree' procedure that utilizes the height of the to reconfigure the vine into a balanced tree. In order for either of the procedures to work, the tree must undergo 'rotations' within its structure. Now, we will look in depth into the rotation process.

Tree rotations are operations within a binary tree that change the structure of the tree without

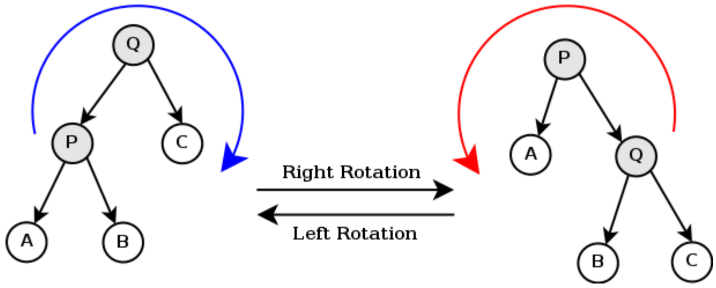



Figure 2.2.1 A depiction of a left and right rotation

interfering with the order of the elements.

Their primary function of rotations is to change the shape of the tree, more specifically used to decrease the tree's height by moving smaller subtrees down and larger subtrees up. Rotations require a 'pivot' node that moves around a 'root' node – in this case 'root' refers to the parent of the pivot node. Rotations can occur either towards the 'left' or 'right' direction with reference to the movement of the pivot in relation to the tree and root, as demonstrated in Figure 2.2.1. The most important detail when understanding tree rotations is the constraints presented. Most notably, the order of the leaves of the tree (nodes A, B and C in Figure 2.2.1) are kept the same after either rotation direction meaning that the same in-order traversal is achieved. Furthermore, tree structure is preserved as after performing any rotation, the left child is smaller than the parent and the right child is greater than the parent. Interestingly, the child of the pivot can become the child of a root without violating either constraint. The simplest code required for a right rotation is demonstrated in Figure 2.2.2

```
295     void rightRotateSimplified(BinaryNode pivot) {
296         try {
297                 BinaryNode root = pivot.parent;
298                 BinaryNode pivotRightChild = pivot.right;;
299                 pivotRightChild.parent = root;
300                 root.left = pivotRightChild;
301                 pivot.right = root;
302                 root.parent = pivot;
303             }
304         catch(NullPointerException e) {
305 
306         }
307     }
308 
```

Figure 2.2.2 Simplified code for a right rotation

A diagram illustrating the process is shown below:

Note the arrows illustrate both the parent and child relationships.

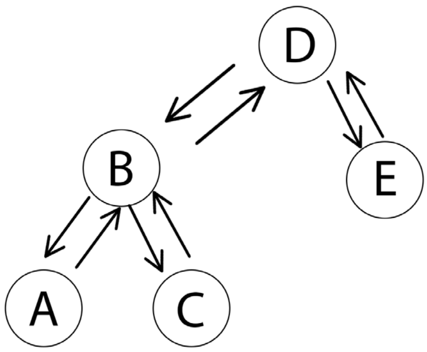



Figure 2.2.3 An initial example tree

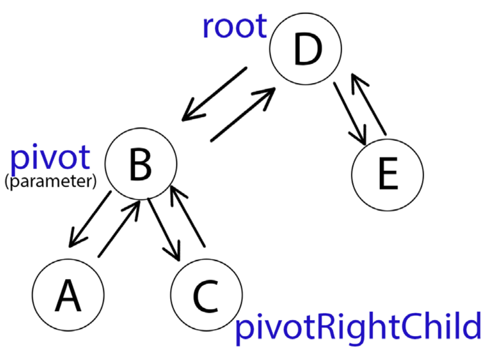



Figure 2.2.4 After executing lines 297-298 and assigning pointers

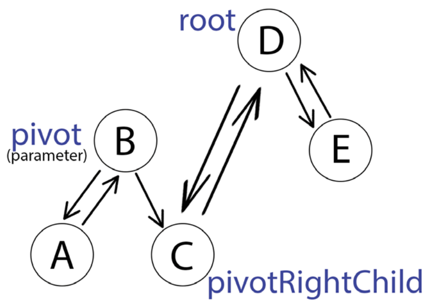



Figure 2.2.5 After executing lines 299-300 and reassigning the pivotRightChild. Note at this stage the pivotRightChild is the child of both the pivot and the root

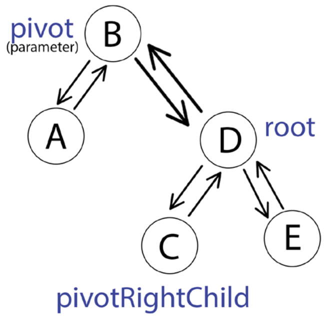



Figure 2.2.6 After executing lines 301-302 and making the root the right child of the pivot and its parent.

The catch statement is used to attempt to not cause the code to exit given certain situations such as nullPointers, when reference is made to a null BinaryNode. Although simple, this code fails to address certain limitations such as the case where the "root" node actually is the root of the tree and the instance variable needs to be reassigned. Furthermore, the code also fails to assign the pivot BinaryNode shown in Figure 2.2.6 its parent. Complications arise as shown in Figure 2.2.7.

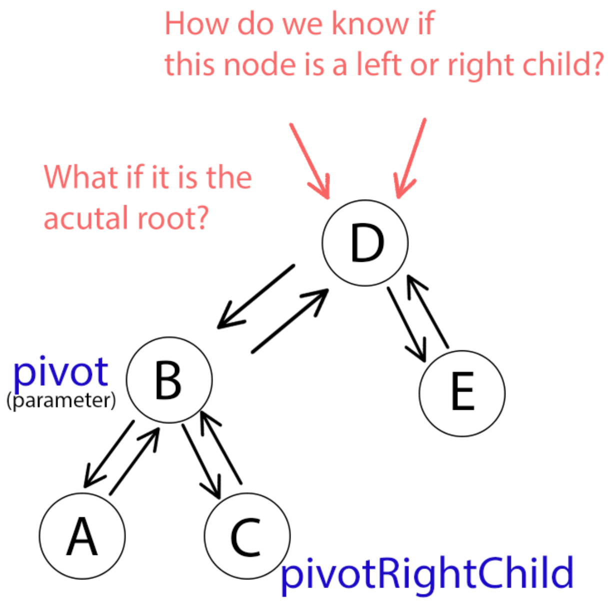



Figure 2.2.6 After executing lines 301-302 and making the root the right child of the pivot and its parent.

How are such limitations considered?

```
    public void rightRotate(BinaryNode pivot) {
        BinaryNode doubleParent = null;
        BinaryNode parent = pivot.parent;
        BinaryNode pivotRightChild = pivot.right;

        if(pivot.parent != null) {
            pivot.parent = parent.parent;
            if(parent.parent != null) {
                doubleParent = parent.parent;
            }
        }
        parent.parent  = pivot;
        parent.left = null;
        parent.left = pivotRightChild;//will be null if pivotRightChild is null
        if(pivotRightChild != null)//only will be entered in there is a pivotRightChild to be assigned
            pivotRightChild.parent = parent;
        pivot.right = parent;

        if (parent.item.toString().equals(root.item.toString())) {
            System.out.println("changedRoot");
            changedRoot = true;
            root.parent = pivot;
            root = pivot;
        }
        else if(doubleParent != null) {
            if(doubleParent.left != null) {
                if (doubleParent.left.item.toString().equals(parent.item.toString()))
                    doubleParent.left = pivot;
            }
            else if(doubleParent.right != null) {
                doubleParent.right = pivot;
            }
        }
    }
```

TO DO: finish explaining rotation limitation solution

Explain T2V

Explain V2T

## 3. Hypothesis and Applied Theory

At this point the relevant theory has been described in good detail and one can appreciate the algorithm – the most effective approach available. Now, it is important to consider if balancing a tree is efficient, and if so, at what amount of search operations and number of nodes. An experiment will be carried out compare the time complexity of searching a random unbalanced tree to that of performing a DSW balancing algorithm followed by the same node searches. Although a binary search (average Big O $O(\log(n))$) will be used for both approaches, the average time will defer from experimental data as trees might require more searches based on balancing.

The experiment will measure the relationship between the size of the tree being searched, x, and the time required to perform 50,000 searches within that tree, y. Data collected will be plotted on a graph and a trend line will be created and used to determine a relationship between the two variables and even to predict at which point one approach will become more efficient, if applicable.

I hypothesize that there will be a linear relationship between the two variables described and that eventually, the balanced tree will become more efficient than the unbalanced tree. This hypothesis is derived from the theoretical approach to the number of searches required reach a desired node: a balanced tree will almost always require less operations to reach a specific node when compared to an unbalanced tree. These z fewer operations per search will eventually make the method more efficient and therefore justify the time complexity required to balance the tree.