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

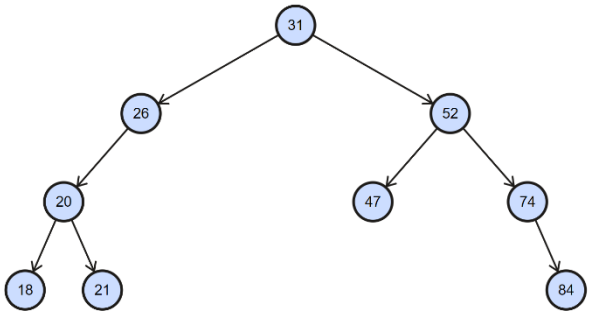
A binary search tree (BST hereon) is a data structure with a defined behaviour that stores comparable 'items' (a general catch-all term used to describe structures such as numbers or strings), commonly referred to as *nodes*. The reference to the word 'binary' suggests that the structure is comprised '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:

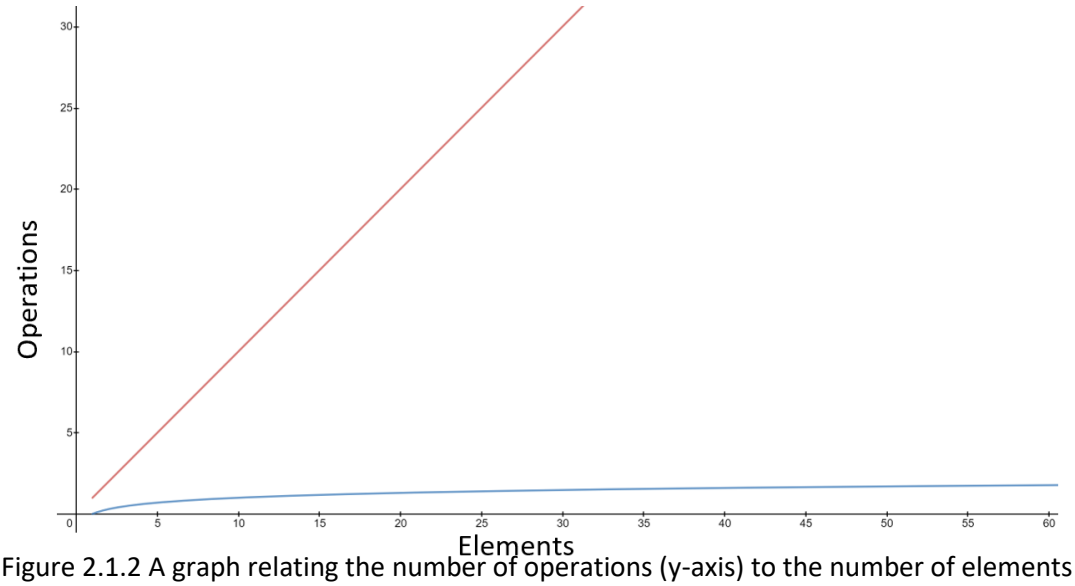


Figure 2.1.2 A graph relating the number of operations (y-axis) to the number of elements within that data structure (x-axis)

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 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

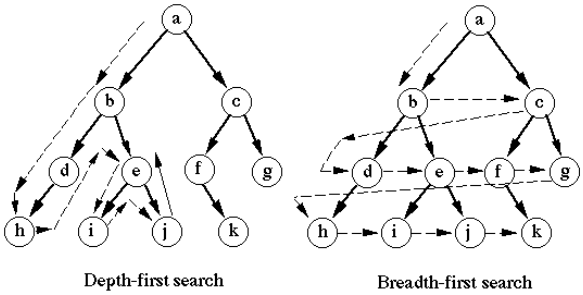


Figure 2.1.3 Depiction of BFS and DFS algorithms

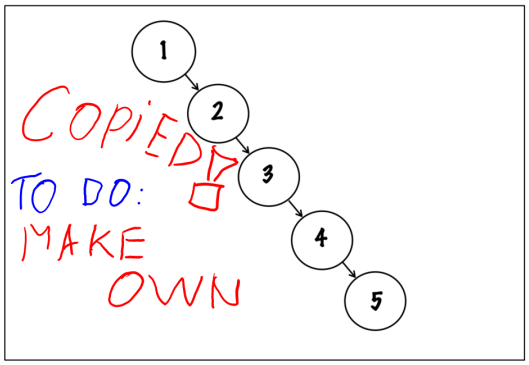


Figure 2.1.4 An unbalanced Binary Tree

$O(1{,}000{,}000)$. 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. 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 have the insert() method recurse 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. 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

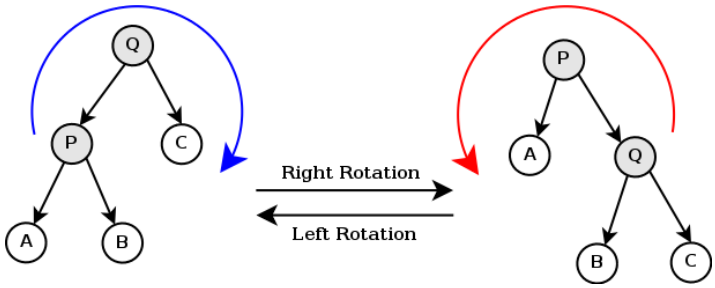


Figure 2.2.1 A depiction of a left and right rotation

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.

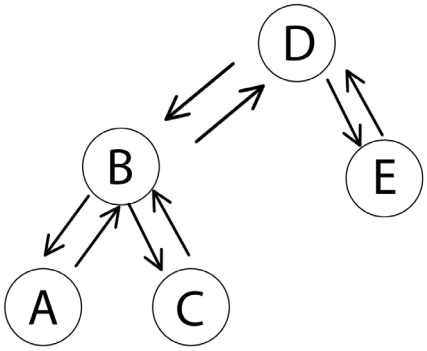


Figure 2.2.3 An initial example tree

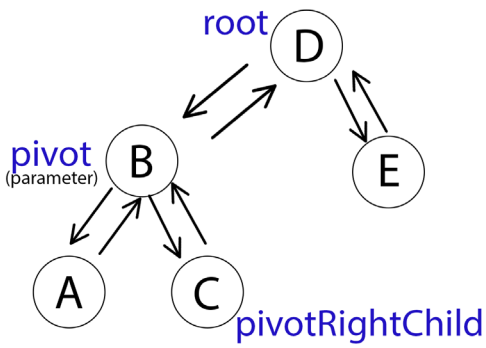


Figure 2.2.4 After executing lines 297-298 and assigning pointers

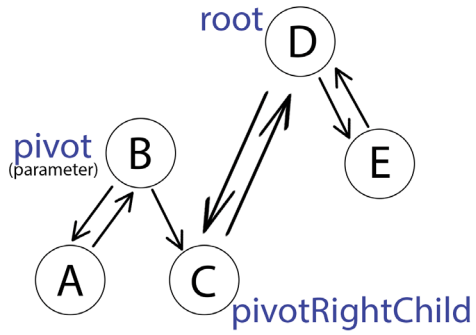


Figure 2.2.5 After executing lines 299-300 and reassigning the pivotRightChild. Note at this stage the pivotRightChild is the child of both the pivot and the root

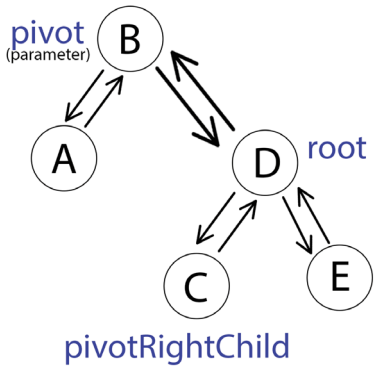


Figure 2.2.6 After executing lines 301-302 and making the root the right child of the pivot and its parent.

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.

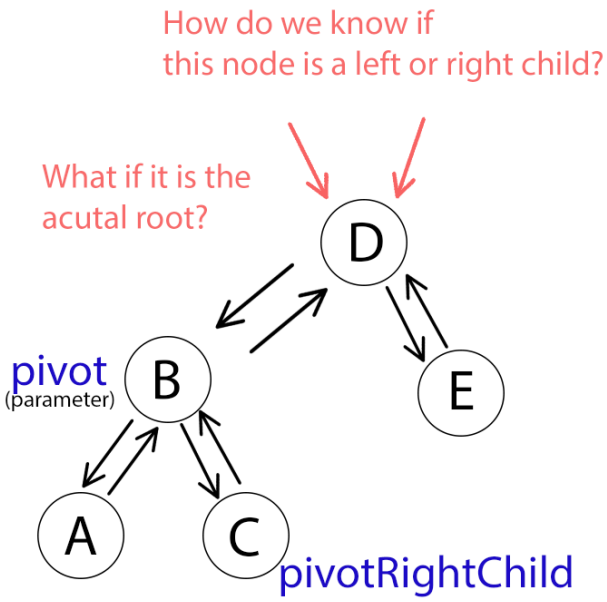


Figure 2.2.7 Problems with the code rotation code presented

In order to resolve such issues, a more complicated rotation algorithm must be developed, as demonstrated in Figure 2.2.8.

```
144    public void rightRotate(BinaryNode pivot) {
145        BinaryNode parent = pivot.parent;
146        BinaryNode pivotRightChild = pivot.right;
147        BinaryNode grandParent = null;
148        if(parent.parent != null) {
149            grandParent = parent.parent;
150        }
151
152        pivot.parent = parent.parent;
153        parent.parent  = pivot;
154        parent.left = null;
155        parent.left = pivotRightChild;//will be null if pivotRightChild is null
156        if(pivotRightChild != null)//only will be entered in there is a pivotRightChild to be assigned
157            pivotRightChild.parent = parent;
158        pivot.right = parent;
159
160        if(grandParent != null) {
161            if(grandParent.left != null) {
162                if (grandParent.left.item.toString().equals(parent.item.toString()))
163                    grandParent.left = pivot;
164            }
165            else if(grandParent.right != null) {
166                grandParent.right = pivot;
167            }
168        }
169        else if(parent.item.toString().equals(root.item.toString())) {
170            changedRoot = true;
171            root.parent = pivot;
172            root = pivot;
173        }
174    }
175
```

Figure 2.2.8 A more complicated rotation algorithm

The code performs the same algorithm as Figure 2.2.2 with additional focus on the parent assignment of the pivot and the movement of its right child. Line 148 is achieved only if the pivot has a parent, meaning that the pivot is not the root of the tree. Consideration is taken in the *if*-statement in line 148 whether or not the grandparent (double parent of pivot, illustrated in red in Figure 2.2.7) exists to assign it to the variable *grandParent* to be later used in line 160-173. Special consideration also needs to be taken for *pivotRightChild* when assigning its parent in the case that the pivot's right child is null, therefore avoiding a nullPointerException. If the pivot's right child parental assignment in line 157 was not surrounded by the if-statement in the line previous checking if the node is not null, we would be assigning a pointer to a null object, resulting in an exception thrown as shown in Figure 2.2.9.

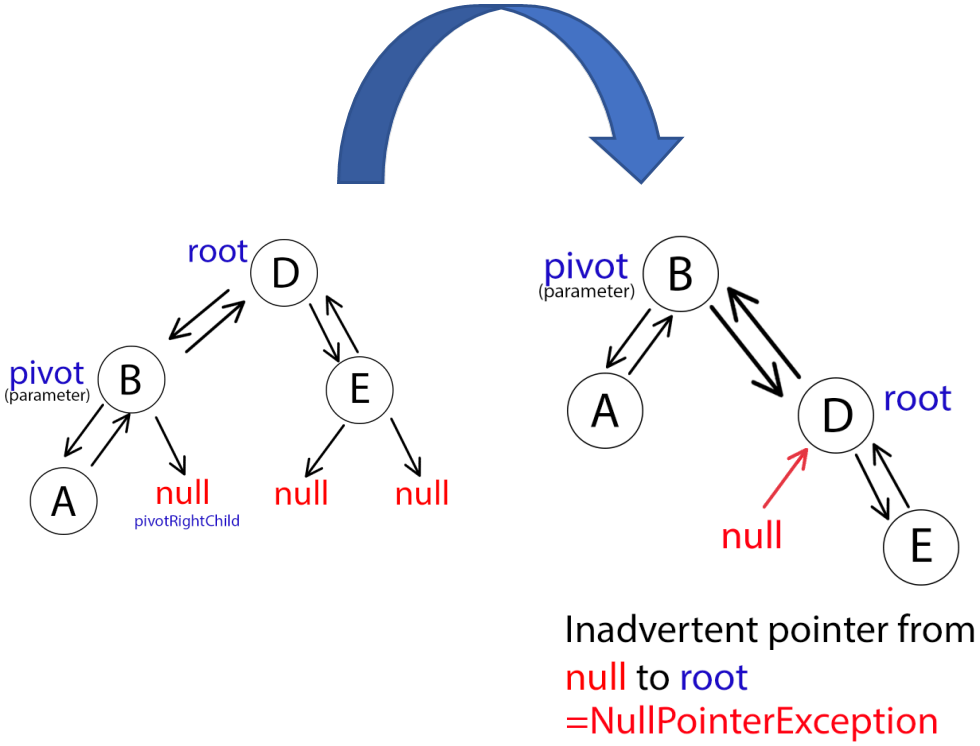


Figure 2.2.9 Why a NullPointerException could be thrown in line 157

The example tree in Figure 2.2.10 will be used to explain the entire rotation algorithm:

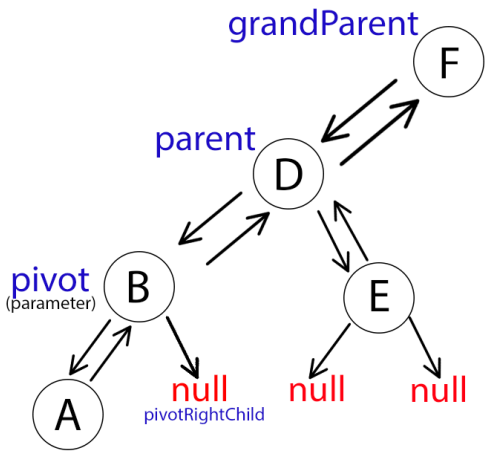


Figure 2.2.7 Initial status of tree

Following the execution of lines 145-159, as described in detail earlier with the simplistic example and the if-statement handling, the tree will look as that in Figure 2.2.12.

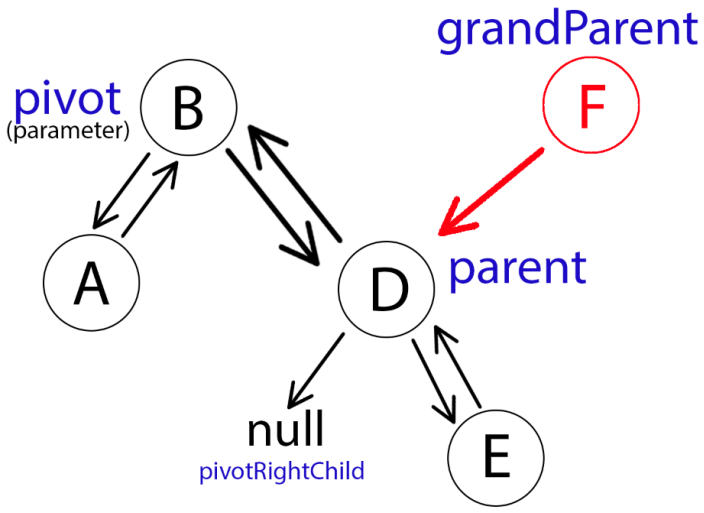


Figure 2.2.12 grandParent assignment incomplete

At this point the grandParent, F, still points to D which needs to be rectified. In order to do so, the relationship between the parent and grandParent needs to be determined, achieved in lines 160-168. The *if* at line 160 is only entered if the grandParent is not null, meaning the pointer needs to be

corrected. The relationship between the grandParent and parent (if the parent is grandParent's left or right child) is determined by the *if* statements in lines 161-162. By ensuring in line 161 that the left child of the grandParent is not null, we check if the left child's item is the *parent's* item (162). If true, we have determined the parent is its left child and can assign the pivot as it's left child. If this is not true, the *grandParent*'s right child has to be *parent*, reason why no additional verification is required to assign the right child as pivot. This produces the correct rotation in figure 2.2.12.

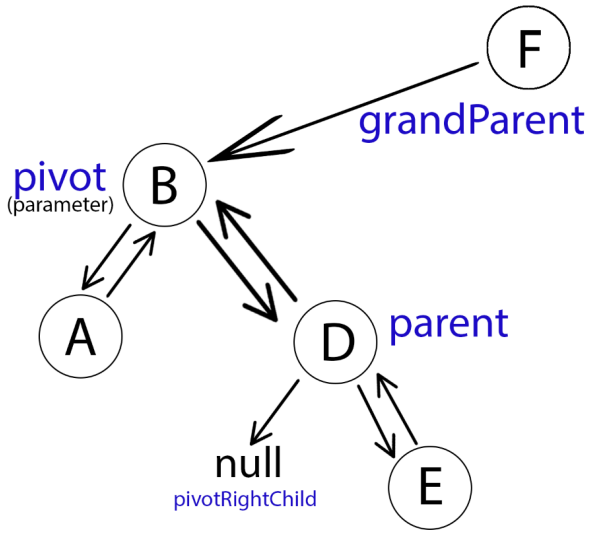


Figure 2.2.12 Completed Right Rotation

A final *else-if* statement exists to reassign the node in the case that *parent* is the root of the tree. This assigns the current *pivot* to *root* as following the rotation *pivot* will become the root of the tree.

At this point, tree rotations have been explained comprehensively and further detail will be placed upon the DSW algorithm.

2.3 DSW: Tree-to-Vine

Figure 2.3.1 demonstrates the algorithm used to turn the tree into a vine, the first part of the DSW algorithm.

```
322     void treeToVine (BinaryNode curent) {
323         BinaryNode currentNode = curent;
324         while (currentNode != null) {
325
326             while(currentNode.left != null) {
327                 rightRotate(currentNode.left);
328                 currentNode = currentNode.parent;
329             }
330
331             currentNode = currentNode.right;
332             nodeCount++;
333         }
334     }
335
336
```

Figure 2.3.1 treeToVine algorithm

What the algorithm essentially does is always attempt to iterate to the left and whenever possible to perform a right rotation, changing the balance of the tree towards the right. This will ultimately achieve a vine (linked-list) to the right. Figure 2.3.2 demonstrates an example tree that will have the **treeToVine()** function applied to it.

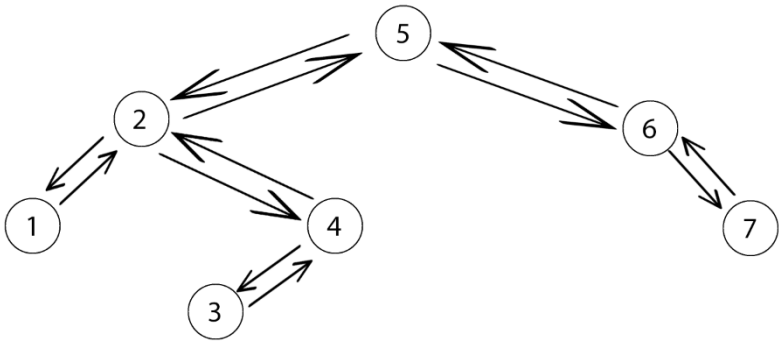


Figure 2.3.2 Example tree

Due to the *while* statements in lines 324, the algorithm will originate at the root and progress leftwards towards node “1” as shown in Figure 2.3.3a, producing the tree in Figure 2.3.3b.

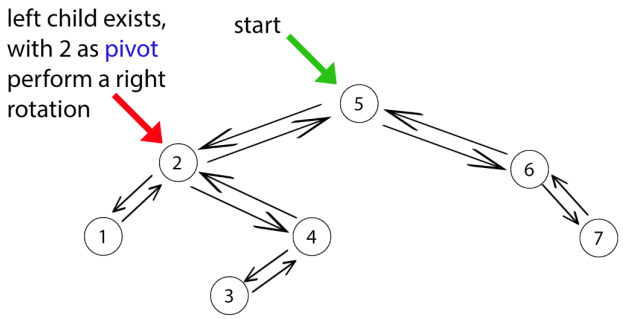


Figure 2.3.3a First iteration of the nested loop

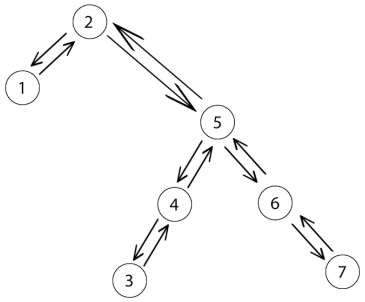


Figure 2.3.3b Tree produced as a result of the rotation

As a result of the change in the root of the tree, the tree rotation algorithm discussed earlier will cause *currentNode* to become the *root*, meaning that the next iteration of the outer while will begin with the *root*, as show in in 2.3.4.

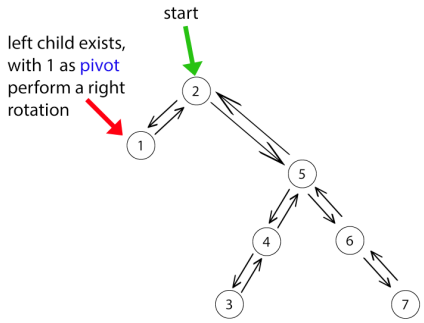


Figure 2.3.4a Second iteration of the nested loop

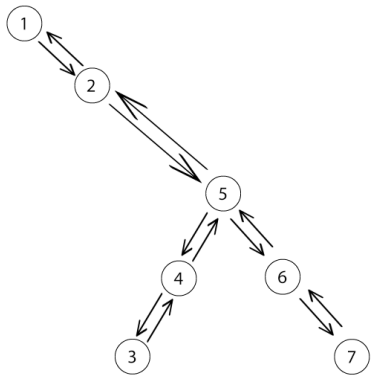


Figure 2.3.4b Tree produced as a result of the

Once more, the root has changed. However, this time the next iteration of the nested *while* will not have its condition fulfilled until when the node “5” is reached, reason for which node 2 will have no rotation. The resulting tree is shown in figure 2.3.5.

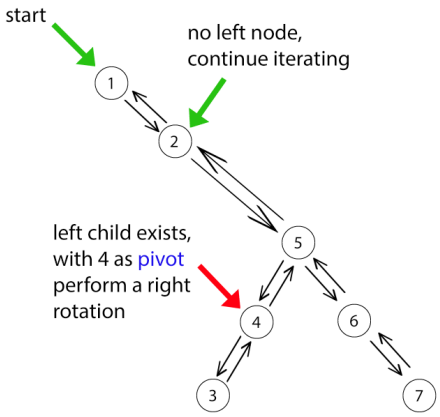


Figure 2.3.5a Third iteration of the nested loop

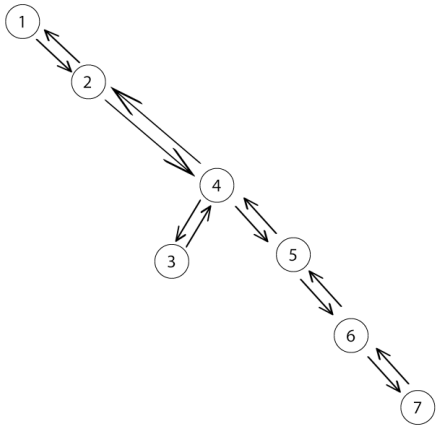


Figure 2.3.5b Tree produced as a result of the rotation

At this point, the tree is visibly nearing completion. The *currentNode* changes to its parent, "2", fulfilling the nested *while's* condition for the last time.

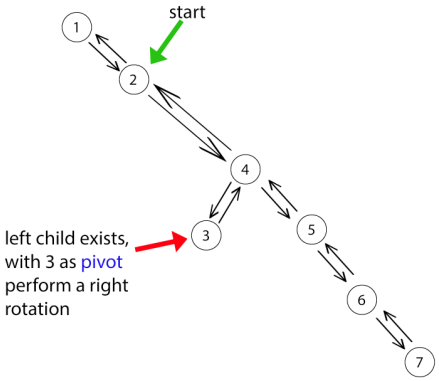


Figure 2.3.6a Third iteration of the nested loop

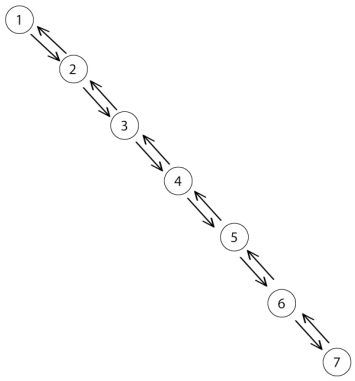


Figure 2.3.6b Tree produced as a result of the rotation

Clearly, the vine has successfully been created. This will be confirmed once the outer while reaches the "7" node that has no right child, as shown in figure 2.3.7. For evert complete outer *while* loop operation, the *nodeCount* value is incremented by one to provide the number of nodes in the tree for the Vine-to-Tree part of the DSW algorithm.

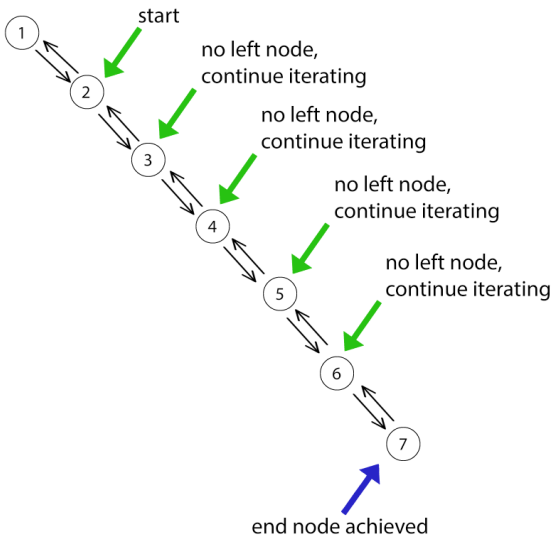


Figure 2.3.7 Vine successfully produced

2.4 DSW: Vine-to-Tree

This part of the balancing algorithm is arguably the more beautiful one. Mathematics is used to determine the number of left rotations required to balance the tree. First, using the counter from the **treeToVine()** method, the number of nodes in the bottom-most layer of the tree is determined using the formula: $nodeCount - 2^{(int)log_2(nodeCount+1)} - 1$. For example, in the case of 7 nodes, the top layer will be filled with one, second layer with two, third layer with four which leaves no nodes for the bottom-most layer. According to the equation, $7 - 2^{log_2(7+1)} - 1 = 7 - 2^3 - 1 = 0$

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.

4. Methodology

The relevant theory to the question was previously described. The specifics of the experiment run in Java will now be approached.

4.1 Independent Variables

The independent variable in this experiment is the size of the tree.

I believe that by incrementing n by____

4.2 Control Variables

Integrated Development Environment Used:

All of the code will be run on Eclipse IDE for Developers version 2019-03 build id: 20190314-1200. The accompanying Java Runtime Environment is version 8 update 201 for 64-bit Windows operating systems.

Computer and Operating System:

Microsoft Surface 6 (i7-8650U CPU @1.9GHz overclocked 2.11GHz, 16GB RAM 1866MHz DD3) running Windows 10 Home version 1809.

Same data type:

All variables used to compare the values of the nodes will be integers.

Same searches:

To ensure that the efficiency is measured correctly between the unbalanced and balanced tree, the searches performed in the first tree will be the same ones performed in the second tree. This will be achieved by creating an ArrayList before searching the unbalanced tree that will be iterated. The same ArrayList will be iterated when searching the balanced tree. This ensures that efficiently is being measured correctly.

4.3 Methodology

1) Create an unbalanced BST that contains n nodes.
2) Create an ArrayList that contains p values within the tree
3) Iterate the array list searching for the nodes within the tree containing that value and print in the console the time required in nanoseconds
4) Run the balancing function and iterate the ArrayList once more and perform searches for the nodes containing the item in the tree. Print in the console the time required for balancing the tree and performing the searches in nanoseconds.
5) Perform the previous steps on a new code run for a total of five trials for each increment.
6) Calculate the average time required for unbalanced searches and DSW followed by searches and find the averages.