A HARDWARE ALGORITHM FOR SELF-BALANCING BINARY SEARCH TREES

by
Joseph Deiermann

A thesis submitted to the faculty of The University of Mississippi in partial fulfillment of the requirements of the Sally McDonnell Barksdale Honors College.

Oxford
December 2017

Approved by

____

Advisor: Assistant Professor Matthew Morrison

____

Reader: Associate Professor Elliot Hutchcraft

____

Reader: Professor Ramanarayanan Viswanathan

Acknowledgements

I first would like to thank the Honors' College and Department of Electrical Engineering for helping me prepare for a career in the field. I would like to thank Dr. Matthew Morrison for supporting me for two years on this project. I could not have finished this without him constantly making me focus on finishing my work. I would finally like to thank my parents for dealing with me all these years and teaching me the value of hard work.

Abstract

Binary search trees are binary trees with an ordering mechanism that makes the time to search for any given item within the tree greatly reduced compared to an unordered array of numbers. More specifically, a balanced binary search tree is faster at finding a specific item in the tree than an unbalanced tree. There are several algorithms that can automatically balance a binary search tree. Most of them do this through rotations directly in their respective insert functions. These algorithms are mostly implemented in software. This paper will present a hardware-based algorithm to balance binary search trees. This algorithm manipulates the ordering of a string representing a binary tree through swapping its elements in certain ways. It can then be used in software and hardware applications where sorting is used, such as in transducers, and where priority queues are needed, such as in bandwidth management on transmission lines.

TABLE OF CONTENTS

LIST OF FIGURES

Section 1: Introduction

Binary search trees are binary trees with the condition that the left children are smaller than the root node and the right children are greater than the root node [7]. With this ordering mechanism in binary search trees, the time to search for any given item within the tree is greatly reduced compared to an unordered array of numbers. When this binary search tree is balanced, the time to search for the given element is faster than an unbalanced tree. There are several algorithms that can automatically balance a binary search tree. Most of them do this through rotations directly in their respective insert functions [7]. These algorithms are mostly implemented in software. This paper will present a hardware-based algorithm to balance binary search trees. This algorithm manipulates the ordering of a string representing a binary tree through swapping numbers in certain ways.

There are a few reasons to do this algorithm in hardware. One of the main reasons is that hardware is a lot faster than software [3]. When software is used, it has to go through layers in the operating system to get the result to the user. It generally involves moving lots of things around in physical memory [13]. Then eventually this data is passed to the CPU. Every time the CPU calculates things, it goes through the same general procedures to get things done [13]. It then moves back through the layers to the end user. Instead of going through all these physical and virtual layers, one can use dedicated hardware to get the job done by bypassing all of those layers. This saves a lot of time and computing power. There is also a lot less overhead because of less workings.

This algorithm can be used in software and hardware applications where sorting is used, such as in transducers, and where priority queues are needed, such as in bandwidth management on transmission lines. For a transducer to work properly, one has to set up equations to find the gain and offset [14]. This is used to figure out the range of voltages we want the transducer to give us. Then we figure out how many steps are in between the maximum and minimum range. We then form a binary search tree of voltages [14]. Whenever a certain voltage comes in contact with a transducer, it maps that voltage with one inside the binary tree and sends that to wherever its needed.

A priority queue is a queue with each element having a priority [15]. When a network router sends out too much outgoing traffic for its bandwidth, the queue with the highest priority gets to pass through while others are halted [16]. This ensures that higher priority traffic gets through with the least delay and a low likelihood of the packets being rejected due to overflow [16]. A binary search tree can be used to construct and maintain a priority queue because of the ordering mechanism within the tree [15].

This paper will present a hardware algorithm to automatically balance binary search trees. In Section 2, the author presents what a balanced binary search tree is and one algorithm of automatically balancing one. Section 3 deals with previous readings related to this paper. In Section 4, the author presents the proposed hardware algorithm and example. In Section 5, the author presents the implementation of the proposed algorithm in VHDL and some examples of this implementation.

Section 2: BSTs and previous work

An n-tuple tree is a tree data structure that consists of a root and n children. A binary tree is a 2-tuple tree, which means it consists of a root and 2 children [7].

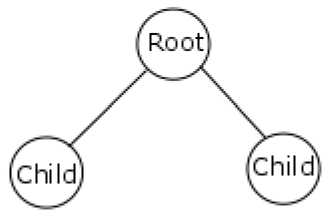


Figure 1

A binary search tree (BST) is a binary tree in which the child on the left (left child) is smaller than the root and the child on the right (right child) is greater than the root [7]. Relevant terms are: node, height, leaf and depth. A node is a generic term for a child or root. A leaf is a node with no children. The depth refers to the number of edges, or lines connecting each node, from the root to a selected node. The height is the depth of the deepest node [7]. Since one can put any number of ordered items in any order, there are different classifications of BSTs. These classifications describe the structure of the BST. Classifications relevant are: a degenerate BST, a balanced BST, an unbalanced BST, and a perfect BST [7]. A degenerate BST is a BST in which each node only has one child associated with it. A balanced BST is a BST in which has the minimum maximum depth. An unbalanced BST is any BST that is not balanced. A perfect BST is a BST in which all children not at the last level have two children and all leaf nodes have the same depth [7].

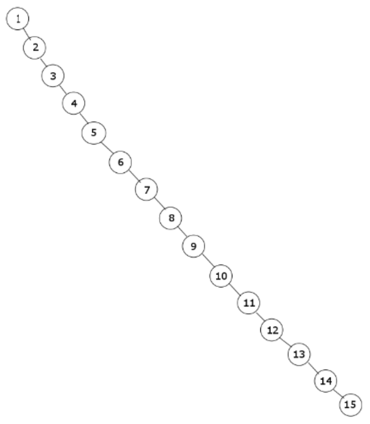


Figure 2. Degenerate Tree

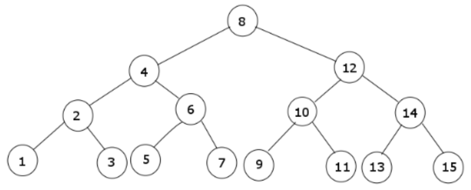


Figure 3. Balanced Tree

Figure 2 is an example of a degenerate BST, in which searching for an element in it would take a long time as the number of items approaches infinity. Let N be the total number of items. For some number n within N, it would take on the order of O(n^2) units of time to search for n [7]. This is the worst-case scenario. Figure 3 is an example of a balanced BST. For a structure in that form, the time it would take to find n in N elements would be on the order of O($\log_2 n$) [7]. This is the best-case scenario. Searching for an element in figure 3 would always be much faster and efficient than in figure 2. So, it is certainly advantageous to try and "balance" figure 2 to look like figure 3. A self-balancing BST is a BST that keeps its depth minimal when there are arbitrary insertions and deletions [7]. There are algorithms to automatically balance such a tree, such as making figure 2 to look like figure 3.

Most self-balancing algorithms have time proportional to the height of the tree. It is, thereby, desirable to keep the tree height small. Let n be the number of nodes and h the tree height. The most nodes a BST could have is $\sum_h 2^h$, which from the geometric series formula yields: 2^{h+1}-1 nodes for some given height h. We have the relation $n \leq 2^{h+1}-1$. This implies that h ≥ floor($\log_2 n$), where the floor function is used [7]. The floor function rounds values down to the nearest integer. This means that the minimum height of the tree is at least $\log_2 n$ rounded down. This height will play a critical role in self-balancing algorithms. They work by transforming the tree using tree rotations at insertion to keep height proportional to $\log_2 n$ [7]. Since maintaining the height at its minimum would cause great overhead, most self-balancing algorithms keep the height to within a constant of this bound. There are several different implementations of this.

The first such implementation is known as an Adelson-Velsky Landis (AVL) tree, named after its inventors [8]. This algorithm rotates a BST based on a balance factor b: b = height of right subtree – height of left subtree, the subtree referring to all of the children on the right and left side of the root [8]. The values for b can only be -1, 0, and 1. For example, in figure 5, b = 1 – 2 = -1, and in figure 4, b = 2 – 1 = 1. This balance factor determines whether there should be a rotation or not. When an element is inserted, it is initially inserted like a BST without any self-balancing algorithm. But since arbitrary insertions can make a degenerate tree, it is necessary to check the balance factor [8]. Depending on the situation, a number of different rotations between the local parents, root and children will vary. The balancing occurs during insertion. The goal is to get b to be -1, 0 or 1 [8]. This is illustrated in an example.

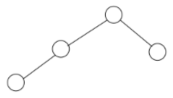


Figure 4

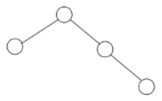


Figure 5

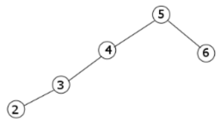


Figure 6

Consider figure 6. The balance factor b = 1 – 3 = -2. We want to change b to be -1, 0 or 1 while keeping the data structure a BST. Assume that 2 is just added. In order to do this, we have to do two rotations: one rotating 4 to be the right child of 3 and another rotating 3 to be the left child of 5, the root. This is illustrated in figure 7. Now we check b: b = 1 – 2 = -1. Therefore, the algorithm balanced the tree.

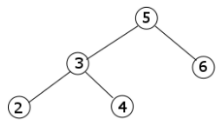


Figure 7

Since the algorithm presented in this paper is more based on this algorithm, we will limit the scope of detail to the AVL tree. Other algorithms include: AA tree, red-black tree, Splay tree, Scapegoat tree, and Treap. Each has advantages and disadvantages compared to AVL, but the AVL implementation is more stringently balanced than the rest [9].

Section 3: Previous readings

This section contains the readings that are related to my thesis.

CASM: A VLSI chip for approximate string matching:

This paper describes an alternate design of an algorithm for string matching. A common dynamic programming algorithm for string matching is using inserts, deletes, and substitutions. Insertions insert a character, deletions delete a character, and substitutions substitutes a character from another in the same position [1]. When comparing two strings, it is handy to put this in a matrix and calculate edit distance. Edit distance is the minimum cost associated with transforming a character or string to another [1]. This cost is a way of quantifying insertions, deletions, and substitutions. The edit distance is a way of quantifying how much to transform a given string to another. One way of calculating edit distance is finding the transformation costs, which is given, and calculate the total cost directly. This takes lots of space if the strings are large, which is the case for many applications [2]. This calls for a more efficient algorithm to help. This paper introduces a new algorithm that splits up the calculations and adds the calculations in an accumulator. The paper proves a way to calculate edit distance from the left and top elements with respect to a given matrix element. All we need is the first two elements for the left and top to calculate the rest. The authors have processing elements (PEs) representing one element in the matrix. They put these PEs in a matrix and accumulate the total cost from each of them [2]. This is proven against other algorithms to be much more efficient. They also show the execution time formula.

The PEs have two strings as inputs. These PEs perform computations along a -45 degree angle and are in a fixed position in a matrix [2]. This only works if the combined string length is N + 1, where N is the number of PEs. This also means that the number of PEs required is N – 1. The first computation is at D(1,1), which is the position in the top left corner of the matrix. This will be where our first elements will be compared [2]. After that, there is a chain reaction in which the subsequent comparisons depend on the first one only. When two strings enter from opposite sides of a PE, the comparison is performed. The next clock cycle the values move out and the next inputs enter. The PE modifies these values based on the comparison before. The PE actually computes the difference between the element and its top and left neighbors. Each of these results are then passed on the to their respective PE. A multiplexer is used to select the source. When strings are shifted out of the array, they take with them the last row and column of the edit distance matrix. From this, we can calculate the edit distance. This calculation is performed using an accumulator [2].

For an example, we'll let insertion = 1 = deletion and substitution = 2. We'll have oboe transform into bbo. We will remove the first o, sub in the second o for a b, and sub the e for an o. The edit distance is then 1 + 2 + 2 = 5.

Adaptive and Pipelined VLSI Designs for Tree – based Codes

This paper designs a hardware algorithm to implement data compression. It can be generalized to more data transformations, but the example in the paper is data compression. Data compression is compressing data by removing unnecessary data, which is lossy, or removing statistical redundancy, which is lossless [4]. The paper deals with lossless compression. Software compression is the most common method of compression. But this hardware design aims to be much more efficient by handling much more data for less time. This paper uses Huffman code as an example of implementation. Huffman code was designed for lossless compression [6]. We use a tree to illustrate how it works. We have to use a reverse tree in order to generate the bits in the correct order. The actual architecture design assigns a token at the start of a character. When the character is done being encoded, the token initiates a feedback token that starts from the beginning of the next character. This action occurs in pipeline. A memory management unit keeps track of the tokens and the decoded bits [3]. For adaptive Huffman coding, the length of code varies with time. The hardware architecture has the inputs of data put into a Weave sort stack and adds the next input with the current input and stores it in a buffer along with any special symbols in a register. The buffer puts the data in content addressable memory. This data is then put in shift registers to receive the next set of data [3].

For an example, we have a tree.

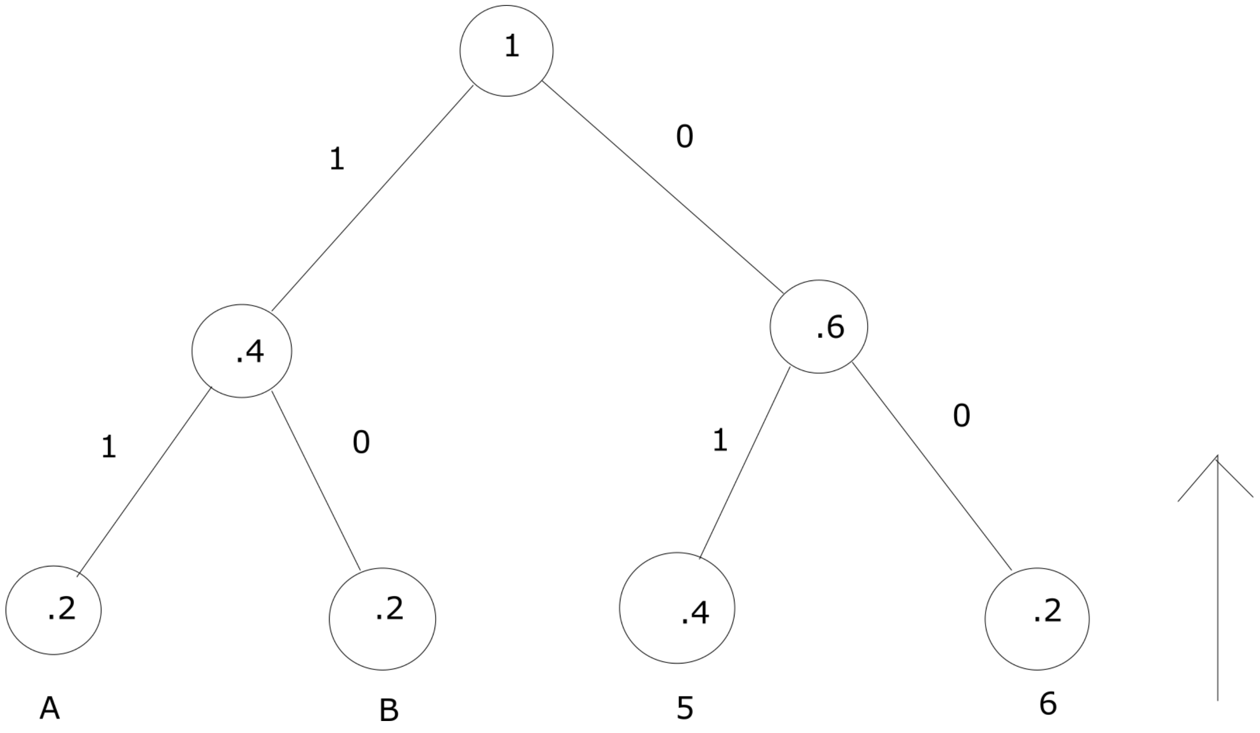


Figure 8

If we wanted the characters B5, then we go up the tree and reset to get 0110. The numbers in the circles are the probabilities.

On Software and Hardware Techniques of Data Engineering

This paper discusses ways of enhancing the speed of data compression in database management systems. In particular, it modifies the arithmetic coding algorithm to allow more characters be represented in an interval by widening the interval. This algorithm maps characters to real numbers within (0, 1). The unmodified algorithm assigns probability intervals between zero and one of probabilities that certain characters will be present. These are put in the interval (0, 1). These are subintervals of the greater interval (0, 1) [4]. In order to put an interval of when a message, or combination of characters, we take an interval of a character, then multiply the probability of the next character to the ending point of the interval to get a narrower interval. They also add an end of message character, which further narrows the interval [4]. When there are large groups of characters, the interval gets smaller and smaller. That means there are fewer characters that can be represented to be encoded. This paper modifies this such that the interval is larger and can represent more characters which improve efficiency. The authors add another character to signal when the locality changes, such as letters to numbers, and groups different localities together with higher probabilities including the end of message and change of locality symbols. The interval becomes orders of magnitude larger than before, making this algorithm more efficient and fast [5]. The authors then discuss the hardware implementation of such a circuit, and it is pipelined. They use Huffman coding [6]. They first assign characters to Huffman code. They then add dummy variables if necessary to make all of the variables the same length. An implementation of Huffman code involves a reverse binary tree [5]. Their hardware implementation has a token move up the reverse binary tree until it reaches the root. If the Huffman code is nothing but dummy variables, then the control signal is 0. Otherwise it is 1. When the token is done with one character it resets to go to the next and the Huffman code appears on a code buffer register and the code length is found. Every time that the control signal at the output is 1, it is counted and put in a memory management unit. This design can be generalized for multiple assignments [5].

For an example, we will have A have probabilities (0,.2), B – (.2, .6), 2 – (.6, .8), @ - (.8, .9), and EOM – (.9, 1). The ranges are .2, .4, .2, .1, and .1 respectively. To find B2, we first have (.2, .6). Then the @ goes in, (.02, .6). Then the 2, (.02, .12). Then the EOM, (.02, .012), which is the final probability.

Section 4: Proposed algorithm

In this section, the author presents another algorithm to balance BSTs. In the following paragraphs, each part is explained. The main algorithm is the proposed algorithm.

Algorithm 1:

Input: Array of non-repeating integers of length n

Output: G(V,E) representing the Balanced Binary Search Tree

```
Structure {
    Node root, left, right, current = null;
    Int data; Int height = 0; Int depth;
}

Main Algorithm{
    Read array of integers and store in array = array;
    Int n = size(array); // n is size of array
    for(int I = 0; I < n; I ++)
        Insert(array[I]);
    While(n is not equal to $\sum_{m=0}^{getDepth} 2^m$ AND height is not equal to floor($\log_2$(n))){
        For(int I = getDepth; I > 0; I --){
            For(int j = 1; j < getElement(I + 1) + 2; j ++){
                If(array[getElement(I - 1) + 1] > array[getElement(I)])
                    Swap(&array[getElement(I - 1) + 1], &array[getElement(I)]);
                Else if(array[getElement(I) – 2j] > array[getElement(I) - j])
                    Swap(&array[getElement(I) -2j], &array[getElement(I) - j]);
            }
            For(int k = 0; k < getElement(I - 1) + 1; k ++){
```

```
                        Else if(array[getElement(I) – 2k] > array[getElement(I) -
k])
                            Swap(&array[getElement(I) -2k], &array[getElement(I)
- k]);
                    }
                    For(int x = 0; x < getElement(I - 1) + 1; x ++){
                        Else if(array[getElement(I - 1) - x] > array[getElement(I) -
x])
                            Swap(&array[getElement(I - 1) - x],
&array[getElement(I)- x]);
                        Else if(array[getElement(I - 1) - x] < array[getElement(I) –
1 - x])
                           Swap(&array[getElement(I - 1) - x],
&array[getElement(I) – 1 - x]);
                    }
                }
                Else
                    Print(error);
                deleteTree();
                For(int p = 0; p < n; p ++)
                    Insert(array[p]);
        }
}
getElement{
        int total = 0;
        for(int I = 0; I <= n; I ++)
                total += 2^n;
        return total;
}
// swap came from an assignment in a previous class
```

```
swap{
            int temp;
            temp = b*;
             b* = a*;
             a* = temp;
}
getDepthUtil{
    if (node is equal to null)
        return 0;
    if (node.data is equal to data)
        return depth;
    int lowerDepth = getDepthUtil(node.left, data, depth+1);

 if (lowerDepth does not equal 0)
        return lowerDepth;
lowerDepth = getDepthUtil(node.right, data, depth+1);
else
       return lowerDepth;
}

getDepth
    return getDepthUtil(node, data, 0);
Height{
            if (node = null)
              return 0;
            else
              left = height(node.left);
              right = height(node.right);
              return 1 + max(left, right);
}
```

```
Insert {
        if (root is equal to null)
            root = new node(data);
        else if (data < root.data)
            root.left = insert( data, root.left );

        else if (data > root.data)
            root.right = insert( data, root.right );

        else
          ;  //Do nothing
        root.height = max( height( root.left ), height( root.right ) ) + 1;
}

deleteTree {
   if (node equals null) return;

   deleteTree(node.left);
   deleteTree(node.right);

   free(node);
}
```

We first explain the helper functions. These helper functions come from various sources and are well known in the community.

The first helper function described is the deleteTree function. This is a recursive function that give each node a null value as it traverses down the tree. We need this to delete the original tree and insert the new tree in its place [10]. If the new element is greater than the root, it is linked to the right of it. If it is less than the root, it is linked to the left of it. It also raises the height by 1 to update our tree. The insert function inserts values in the BST without any balancing, unlike the AVL tree in which the balancing occurs in the insert function. Our balancing occurs in the main part of the algorithm. The height function gives us the height, which is necessary to see if the BST is balanced. This is recursive, adding 1 for each edge as it traverses down the tree [12]. The getDepth and get DepthUtil functions provide us the depth to use in our calculations. This is a recursive function that traverses down the tree to give us the depth [11]. The swap function swaps references to elements in memory. The result is the references will reference different numbers that will be reflected in the array. This is the way we rotate our nodes in our tree, as we will see in the main algorithm description. The getElement function returns the total number of elements in the tree. It is represented by $\sum_n 2^n$. We will need this to access particular elements in our algorithm [12].

We now turn our attention to our main algorithm. We first insert our initial elements to make our tree.

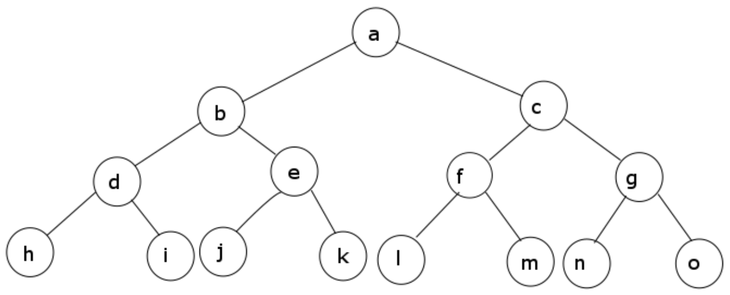


Figure 9

We will use our tree elements as a string. There should not be duplicate element values. For example, our figure 8 string would look like:

a-b c-d e f g-h I j k l m n o. The first element in the string is the root and the next two are its children. The relation is the same for the rest of the elements, each element has its two children on the next level, with each level represented by a dash. We use a while loop since the algorithm should continue until the tree is balanced. To determine if the tree is balanced, we have two conditions to continue in the loop: the size of the array is not equal to the calculated number of elements and that the height is not equal to floor($\log_2$(n)). Since we want the height to be minimized, we continue to iterate through the algorithm until the calculated height is equal to floor($\log_2$(n)). From the previous section, we know that the height has to be at least $\log_2$(n) rounded down. This establishes what the minimum height should be. If the height is not equal to the minimum, i.e. is greater than the minimum, the algorithm should continue. Likewise, if the number of elements is not equal to what should be the number of elements calculated from the depth, the algorithm will continue until those conditions are satisfied. For example, in figure 2, where we have a degenerate tree with 15 elements, the calculated minimum height would be floor($\log_2$(15)) = 3. But if we traverse the tree, we find the height to be 14. When the tree is balanced, like in figure 3, the height is 3, which equals the calculated minimum. Similarly, we know we have 15 elements in figures 2 and 3. But if we use the depth definition of total elements, in figure 2 we would have $\sum_{m=0}^{15} 2^m$ = 65535 elements, which is clearly wrong. For figure 3, we have $\sum_{m=0}^{3} 2^m$ = 15, which is the number of elements in this example. It can be noted that $\sum_{m=0}^{getDepth} 2^m$ is essentially the getElement function. I explicitly put in the formula for ease of reading. We now turn to what the algorithm actually does.

The way this algorithm works is a three-step process. The first for loop changes the grouping of elements. The second for loop changes the individual elements. The first step is swapping two elements if the second element in the particular grouping is less than the first one in that same grouping. It starts with the second to last element of the string and compares them. It then compares the previous two elements in the group, and continues throughout the string. The end elements of each grouping are also compared simultaneously. The first element in the string is left alone since it there is no element to compare it with. The second step is the same thing as the first except it starts at the last element. Again, the first element in the string is left alone and is done throughout the string. The last step is to balance the tree. We take three elements from the string, starting with the last element of the second to last grouping and pairing it with the last two elements of the whole string. The goal is to have the local root be the number whose value is between two other values. If the first element is less than the second, they are swapped. Then one has to check if the number swapped to the root is greater than the third number. If it is, they are swapped. The other conditions would be if the first element is greater than the last element, they are swapped. Then one has to compare the new first element is less than the second element. If it is, those elements are swapped. The algorithm keeps going on to the previous groupings until the numbers run out. We illustrate this with an example.

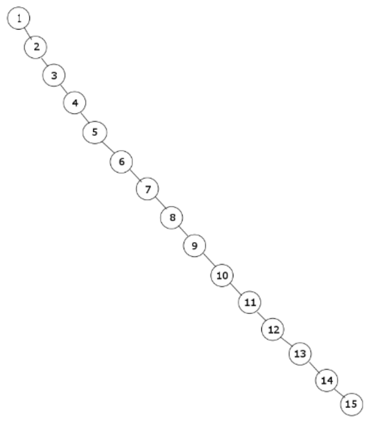


Figure 2

Using figure 2 as an example, repeated for convenience, the string would look like 1--2 3--4 5 6 7--8 9 10 11 12 13 14 15. Each level is divided by dashes. The first step is swapping two elements if the second element in the particular grouping is less than the first one in that same grouping. For our previous string, we have 1--2 3--4 5 6 7--8 9 10 11 12 13 14 15. Since every element doesn't satisfy the condition, nothing is swapped. The second step is the same thing as the first except it starts at the last element. Again, nothing changes since every element doesn't satisfy the condition. We have 1--2 3--4 5 6 7--8 9 10 11 12 13 14 15. The last step is to balance the tree. We take three elements from the string, starting with the last element of the second to last grouping and pairing it with the last two elements of the whole string. For our example, the first grouping to undergo this is highlighted and bolded; 1--2 3--4 5 6 **7**--8 9 10 11 12 13 **14 15**. We compare 7—14 15. Since 7 is less than 14, we swap them; 14—7 15. Then we check is 14 is greater than 15, which is false. If it were true, they would be swapped, but since it is false, our result is the grouping 14—7 15. The next grouping is 6—12 13. We do the same thing and the new grouping turns to 12—6 13. The algorithm keeps going on to the previous groupings until the numbers run out.

This three-step process repeats until the desired string is found. The following tables illustrate what happens in our example. Each table is one iteration. I calculated there to be 3 iterations of the algorithm. Since this is the worst-case scenario, any other combination of those numbers has to have a less than or equal number of iterations.

1st step (swap)	1--2 3--4 5 6 7--8 9 10 11 12 13 14 15

2nd step (swap)	1--2 3--4 5 6 7--8 9 10 11 12 13 14 15
3rd step (balance)	1--2 3--8 10 12 14--4 9 5 11 6 13 7 15 1--8 12--2 10 3 14--4 9 5 11 6 13 7 15 8--1 12--2 10 3 14--4 9 5 11 6 13 7 15

Iteration 1

1st step	8--1 12--2 3 10 14--4 5 9 6 11 7 13 15
2nd step	8--1 12--2 3 10 14--4 5 6 9 7 11 13 15
3rd step	8--1 12--4 6 10 14--2 5 3 9 7 11 13 15 8--4 12--1 6 10 14--2 5 3 9 7 11 13 15

Iteration 2

1st step	8--4 12--1 6 10 14--2 3 5 7 9 11 13 15
2nd step	8--4 12--1 6 10 14--2 3 5 7 9 11 13 15
3rd step	8--4 12--2 6 10 14--1 3 5 7 9 11 13 15

Iteration 3

So, our final string is 8—4 12—2 6 10 14—1 3 5 7 9 11 13 15. Our final tree is illustrated in figure 3, repeated here for convenience.

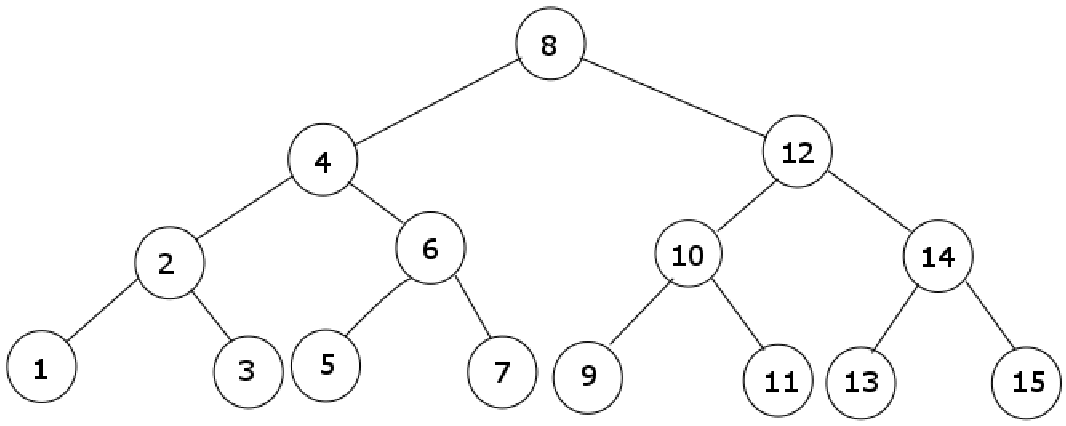
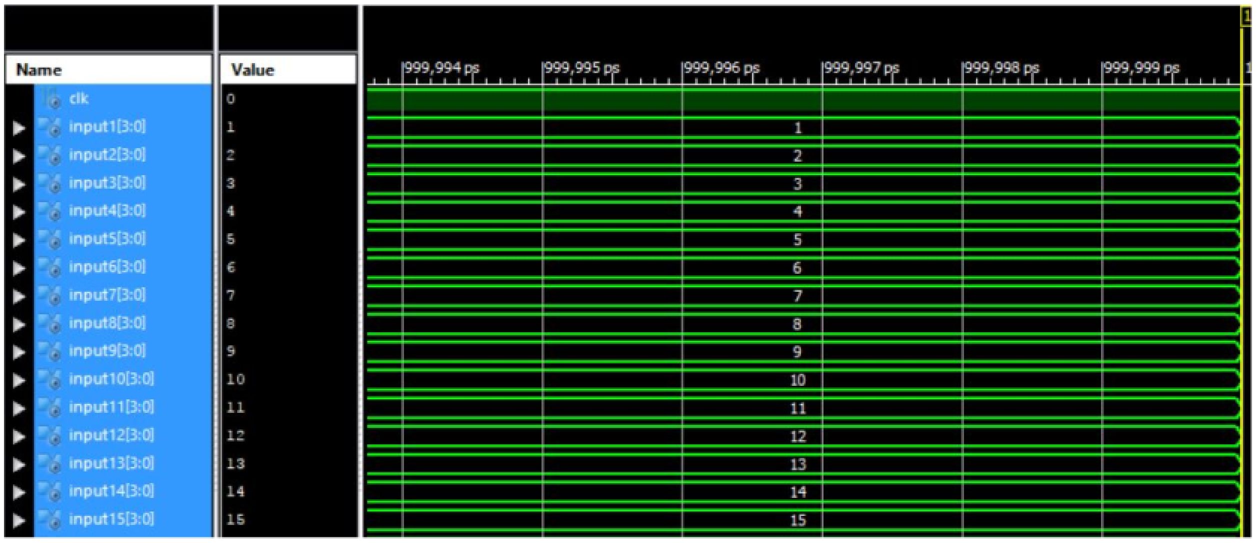


Figure 3

Section 5: Implementation

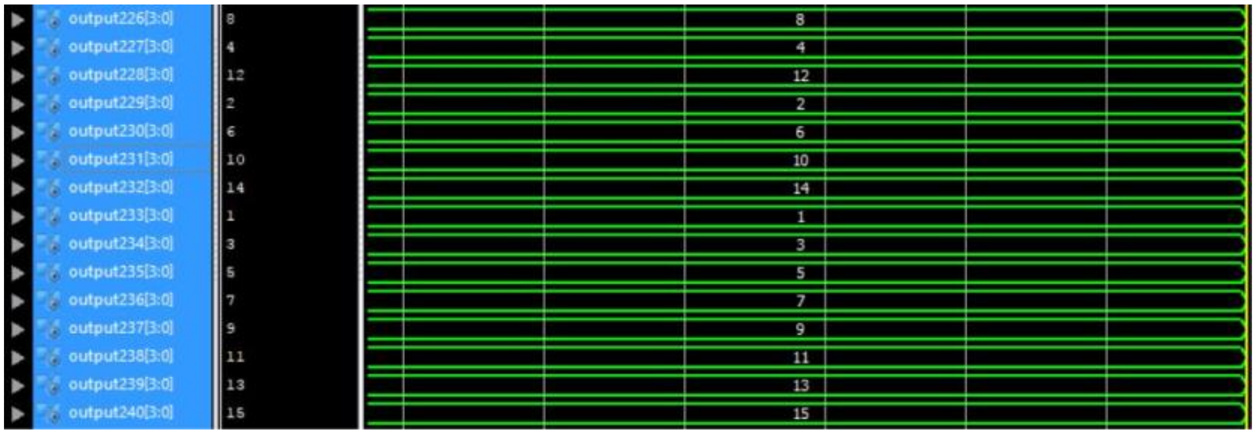
The author simulated Algorithm 1 in VHDL using Xilinx ISE 14.7 with 16 inputs. Since hardware description languages (HDLs) have a difficult time with while loops, the author decided to iterate the algorithm three times plus one more for safety. A loop in an HDL is interpreted as adding more physical hardware. For my while loop, and generally most while loops, the condition to stop is not known right away. As a result, ISE doesn't know how much hardware to make. The author, thereby, had to use a known number of iterations and calculated by hand that for the worst-case scenario, three iterations are the maximum it took to balance the tree. Therefore, anything better than the worst-case scenario will have to take less than or equal to four iterations to balance. The author shows three examples of the algorithm waveforms. There is a bug in ISE on my computer in which on input2, if there is a 1 in the hundreds' place on the number, it replaces the number with zero and messes up the tree. The author doesn't know why it does that. It is most likely a problem with the compiler.

The first example is the example from Section 3, using figures 2 and 3. The waveform for the inputs is in figure 9. Our inputs are 1-2 3- 4 5 6 7- 8 9 10 11 12 13 14 15.



Name	Value
clk	0
input1[3:0]	1
input2[3:0]	2
input3[3:0]	3
input4[3:0]	4
input5[3:0]	5
input6[3:0]	6
input7[3:0]	7
input8[3:0]	8
input9[3:0]	9
input10[3:0]	10
input11[3:0]	11
input12[3:0]	12
input13[3:0]	13
input14[3:0]	14
input15[3:0]	15

Figure 9

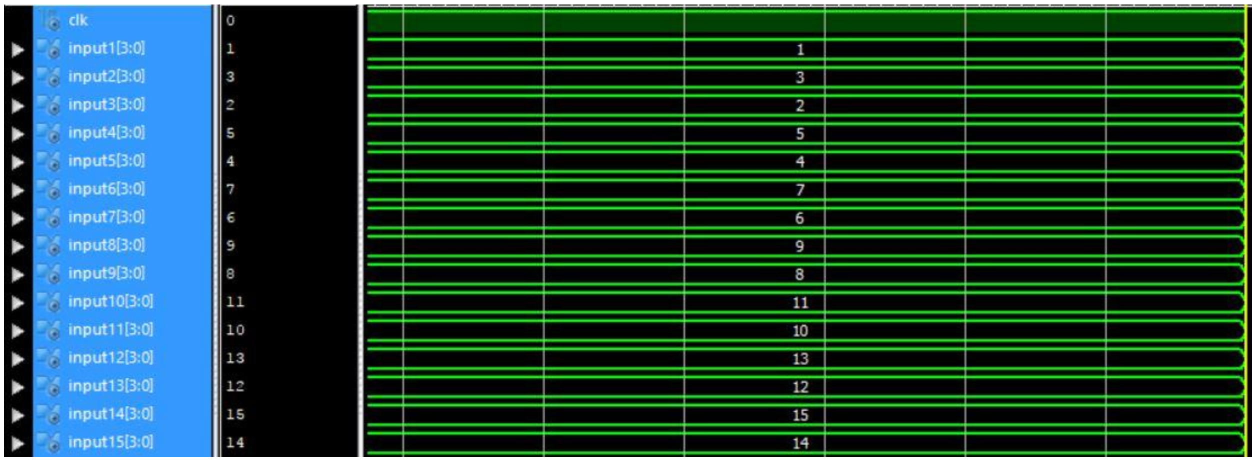


output226[3:0]	8
output227[3:0]	4
output228[3:0]	12
output229[3:0]	2
output230[3:0]	6
output231[3:0]	10
output232[3:0]	14
output233[3:0]	1
output234[3:0]	3
output235[3:0]	5
output236[3:0]	7
output237[3:0]	9
output238[3:0]	11
output239[3:0]	13
output240[3:0]	15

Figure 10

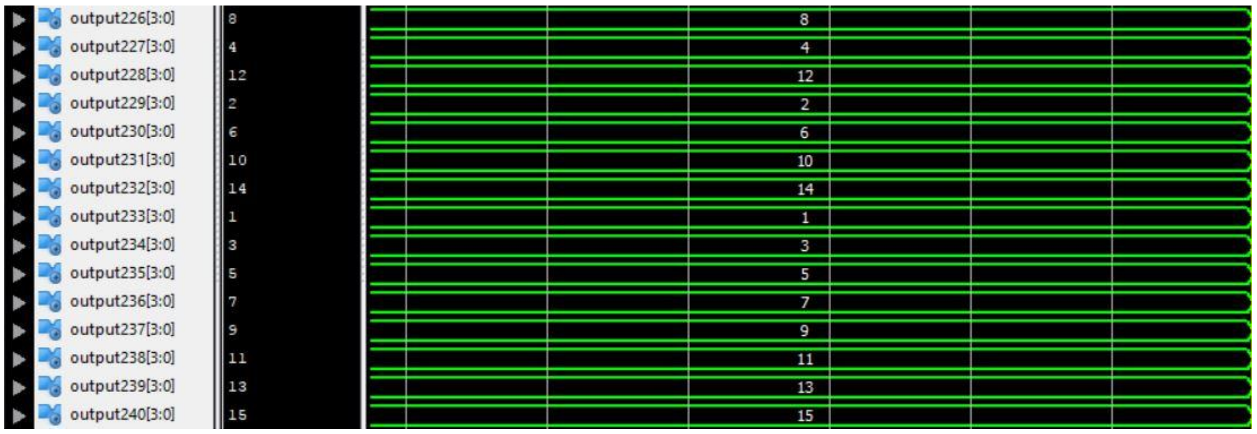
The output is in figure 10. The output string is 8-4 12- 2 6 10 14-1 3 5 7 9 11 13 15, which is what we get in our example. The next simulation will have the input string as 1-3 2- 5 4 7 6- 9 8 11 10 13 12 15 14, shown in figure 11.

The output, shown in figure 12 is 8-4 12-2 6 10 14-1 3 5 7 9 11 13 15.



```
clk            0
input1[3:0]    1     1
input2[3:0]    3     3
input3[3:0]    2     2
input4[3:0]    5     5
input5[3:0]    4     4
input6[3:0]    7     7
input7[3:0]    6     6
input8[3:0]    9     9
input9[3:0]    8     8
input10[3:0]   11    11
input11[3:0]   10    10
input12[3:0]   13    13
input13[3:0]   12    12
input14[3:0]   15    15
input15[3:0]   14    14
```

Figure 11

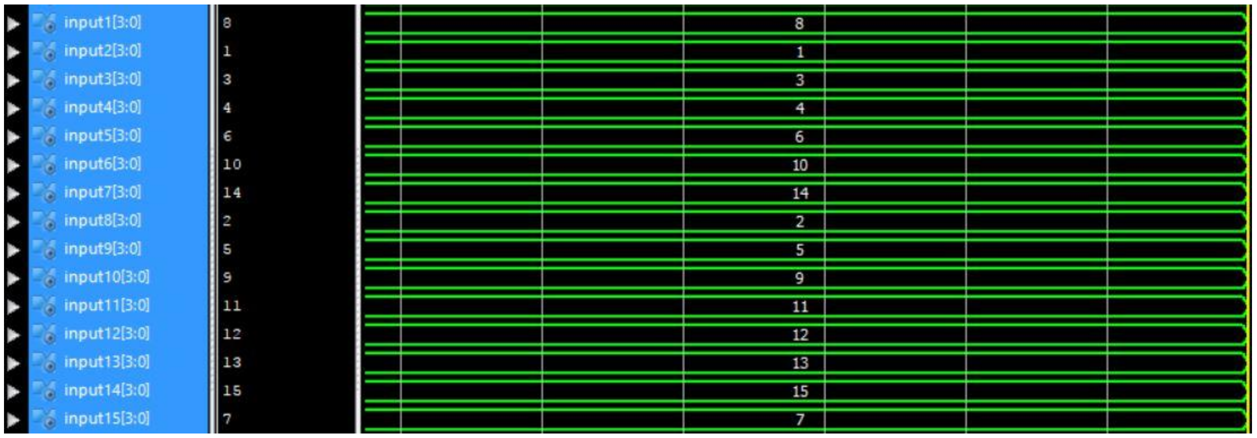


```
output226[3:0]   8     8
output227[3:0]   4     4
output228[3:0]   12    12
output229[3:0]   2     2
output230[3:0]   6     6
output231[3:0]   10    10
output232[3:0]   14    14
output233[3:0]   1     1
output234[3:0]   3     3
output235[3:0]   5     5
output236[3:0]   7     7
output237[3:0]   9     9
output238[3:0]   11    11
output239[3:0]   13    13
output240[3:0]   15    15
```

Figure 12

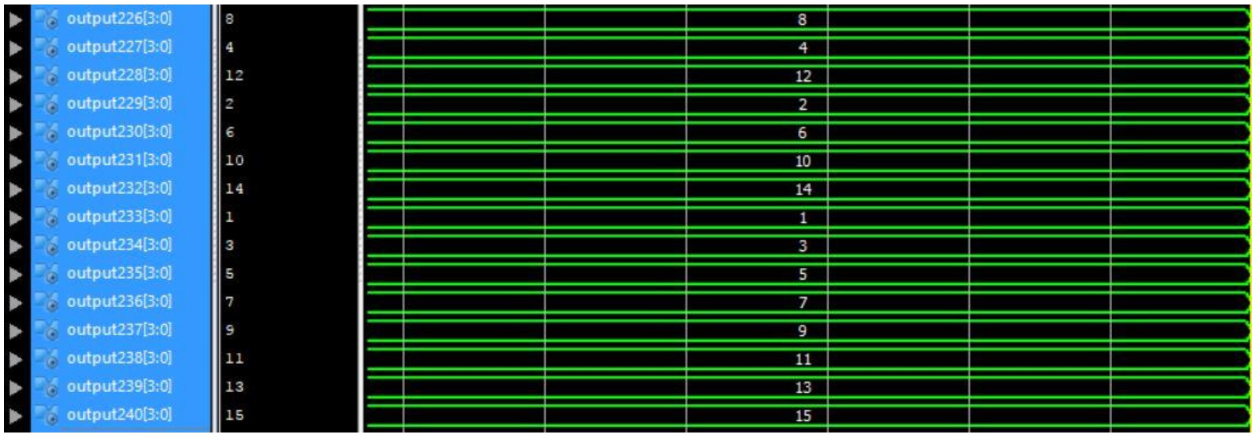
The next simulation will have the input string as 8-1 3- 4 6 10 14-2 5 9 11 12 13 15 7,

shown in figure 13. The output, shown in figure 14 is 8-4 12-2 6 10 14-1 3 5 7 9 11 13 15.



```
input1[3:0]   8    8
input2[3:0]   1    1
input3[3:0]   3    3
input4[3:0]   4    4
input5[3:0]   6    6
input6[3:0]   10   10
input7[3:0]   14   14
input8[3:0]   2    2
input9[3:0]   5    5
input10[3:0]  9    9
input11[3:0]  11   11
input12[3:0]  12   12
input13[3:0]  13   13
input14[3:0]  15   15
input15[3:0]  7    7
```

Figure 13



```
output226[3:0]  8    8
output227[3:0]  4    4
output228[3:0]  12   12
output229[3:0]  2    2
output230[3:0]  6    6
output231[3:0]  10   10
output232[3:0]  14   14
output233[3:0]  1    1
output234[3:0]  3    3
output235[3:0]  5    5
output236[3:0]  7    7
output237[3:0]  9    9
output238[3:0]  11   11
output239[3:0]  13   13
output240[3:0]  15   15
```

Figure 14

Section 6: Conclusion

Binary search trees are binary trees with the condition that the left children are smaller than the root and the right children are greater than the root. With this ordering mechanism in binary search trees, the time to search for any given item within the tree is greatly reduced. More specifically, a balanced binary search tree is faster at finding a specific item in the tree than what is a degenerate binary tree. There are several algorithms that can automatically balance a binary search tree. Most of them do this through rotations directly in their respective insert functions. These algorithms are mostly implemented in software. This paper presented an algorithm to balance binary search trees based on hardware. The inputs of the tree were mapped to an array in level order. The algorithm then manipulated this array based on conditions presented in Section 4. The author then tested this algorithm on three strings and got the correct resultant tree.

References

[1] *Влади́мир И. Левенштейн (1965). Двоичные коды с исправлением выпадений, вставок и замещений символов [Binary codes capable of correcting deletions, insertions, and reversals].Доклады Академий Наук СССР (in Russian)* ***163*** *(4): 845–8.*Appeared in English as: *Levenshtein, Vladimir I. (February 1966). "Binary codes capable of correcting deletions, insertions, and reversals". Soviet Physics Doklady* ***10*** *(8): 707–710.*

[2] Sastry, R.; Ranganathan, N.; Remedios, K., "CASM: a VLSI chip for approximate string matching," in *Pattern Analysis and Machine Intelligence, IEEE Transactions on* , vol.17, no.8, pp.824-830, Aug 1995
doi: 10.1109/34.400575
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=400575&isnumber=9031

[3] Mukherjee, A.; Ranganathan, N.; Bassiouni, M.A., "Adaptive and pipelined VLSI designs for tree-based codes," in *Computer Design: VLSI in Computers and Processors, 1989. ICCD '89. Proceedings., 1989 IEEE International Conference on* , vol., no., pp.369-372, 2-4 Oct 1989
doi: 10.1109/ICCD.1989.63390
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=63390&isnumber=2316

[4] Langdon, G.G., Jr., "An Introduction to Arithmetic Coding," in *IBM Journal of Research and Development* , vol.28, no.2, pp.135-149, March 1984
doi: 10.1147/rd.282.0135
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5390377&isnumber=5390374

[5] Bassiouni, M.A.; Mukherjee, A.; Ranganathan, N., "On software and hardware techniques of data engineering," in *Data Engineering, 1989. Proceedings. Fifth International Conference on* , vol., no., pp.208-215, 6-10 Feb 1989
doi: 10.1109/ICDE.1989.47216
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=47216&isnumber=1789

[6] Huffman, D.A., "A Method for the Construction of Minimum-Redundancy Codes," in *Proceedings of the IRE* , vol.40, no.9, pp.1098-1101, Sept. 1952

doi: 10.1109/JRPROC.1952.273898

URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4051119&isnumber=4051100

[7] Donald Knuth. *The Art of Computer Programming,* Volume 3: *Sorting and Searching,* Second Edition. Addison-Wesley, 1998. ISBN 0-201-89685-0. Section 6.2.3: Balanced Trees, pp. 458-481.

[8] *Georgy Adelson-Velsky, G.; Evgenii Landis (1962). "An algorithm for the organization of information. Proceedings of the USSR Academy of Sciences (in Russion).* ***146****: 263-266.* English translation by Myron J. Ricci in *Soviet Math. Doklady,* 3:1259-1263, 1962.

[9] Pfaff, Ben. "Performance analysis of BSTs in system software". Proceedings of the joint international conference on Measurement and modeling of computer systems 32.1 (2004). 410-411. Print.

URL: https://web.stanford.edu/~blp/papers/libavl.pdf

[10] Write a program to Delete a tree. (n.d.). Retrieved from http://www.geeksforgeeks.org/write-a-c-program-to-delete-a-tree/

[11] Get Level of a node in a Binary Tree. (n.d.). Retrieved from http://www.geeksforgeeks.org/get-level-of-a-node-in-a-binary-tree/. Modified to find depth.

[12] Java Program to Implement Self Balancing Binary Search Tree. (n.d.). Retrieved from http://www.sanfoundry.com/java-program-implement-self-balancing-binary-search-tree/. Modified helper functions.

[13] Hansen, Per Brinch. *Classic Operating Systems: from Batch Processing to Distributed Systems*. Springer New York, 2011.

[14] Fraden, Jacob. "Handbook of Modern Sensors." *Physics, Designs, and Applications* | *Jacob Fraden* | *Springer*, Springer-Verlag New York, 2010, www.springer.com/us/book/9781493900404.

[15] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 6.5: Priority queues, pp. 138–142.

[16] Bandwidth Management Introduction. (n.d.). Retrieved from https://webhelp.radware.com/AppDirector/v214/214Classes%20and%20Bandwidth%20Management.06.02.htm