COPYRIGHT © 2022 Jason Kotowski, All rights reserved.

**Huffman Trees**

Jason Kotowski (jk234@uakron.edu)

**Introduction**

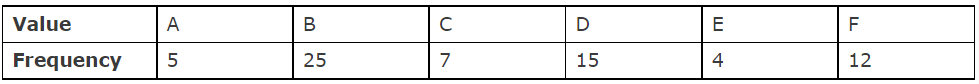
According to world-renowned computer scientist Donald Knuth, “Huffman coding is one of the fundamental ideas that people in computer science and data communications are using all the time.”[[1]](#footnote-0) A Huffman coding tree is a binary tree data structure in which each leaf node has a set character or number.[[2]](#footnote-1) Huffman trees are excellent data structures for data compression. However, they come at the cost of lower time efficiency. The details of Huffman trees will be discussed later.

The purpose of our research is to determine the efficiency of a huffman tree to encode a string of characters compared to other data structures. Our goal is to determine if Huffman trees can be used to efficiently store and display data compared to TREAPs. Another sub-goal is to compare the differences between a Huffman tree where the frequencies are all the same and one where the frequencies are all different.

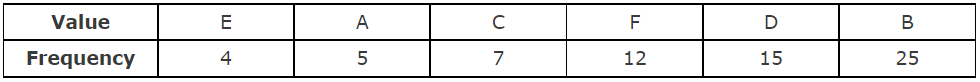
**Discussion**

Hypothesis: Huffman Trees will have increased time complexity compared to TREAP data structures. In addition, Huffman trees with constant frequencies will have better time efficiency than those with random frequencies. Huffman trees are often used to encode and decode strings of text. They do so by assigning each character a unique frequency. Huffman trees where all the frequencies are the same are referred to as “fixed-length codes.” They are much more space-efficient than random frequencies, but they are not used as often. The goal of a Huffman tree is to construct a tree with the minimum external path weight, in such a way that it is the path with the lowest sum of weighted path lengths for the given structure.[[3]](#footnote-2)

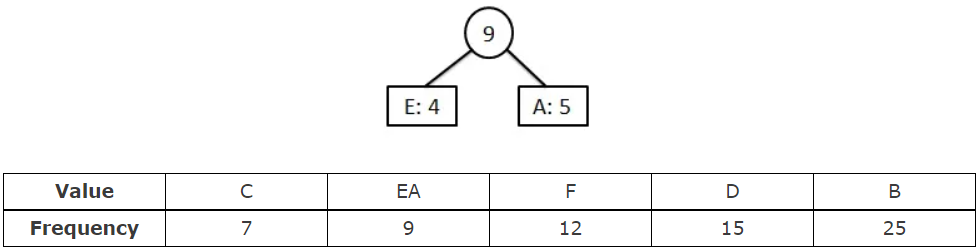
Huffman trees are constructed as such: suppose you have a table of letters, each having their own assigned frequency.



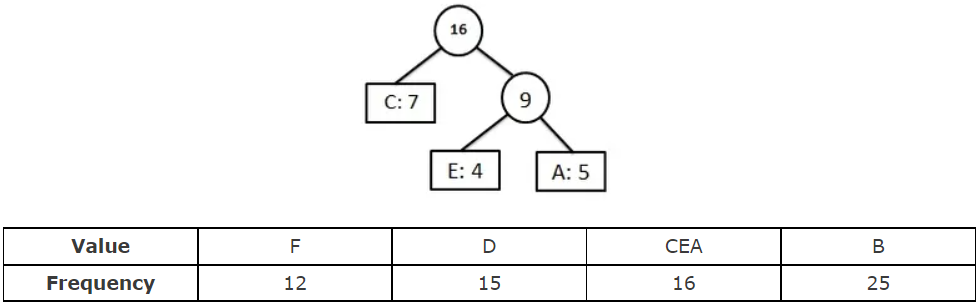
The first step is to order the values in the range from smallest to largest frequencies.



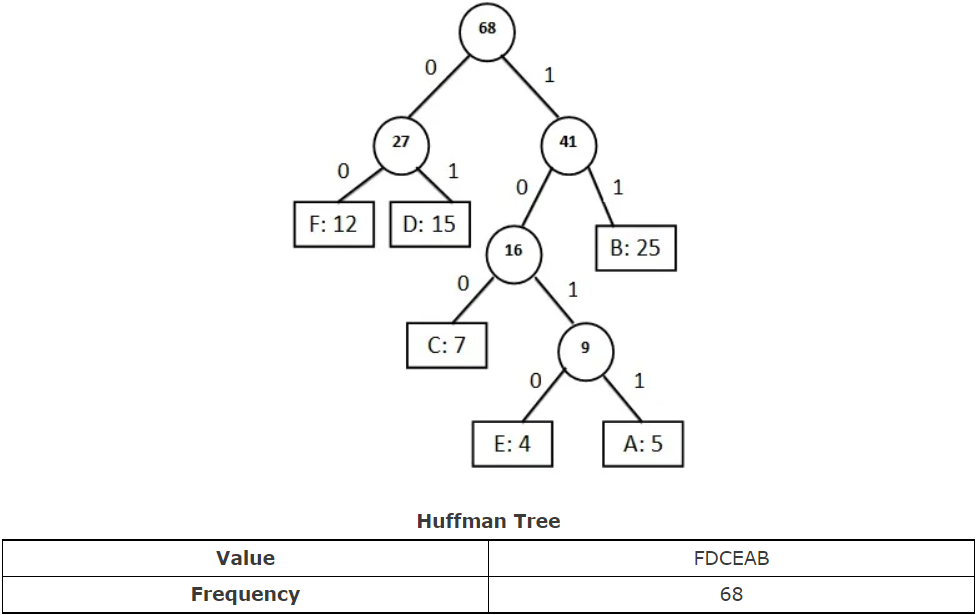
The second step is to find the two smallest values and connect them on the graph as such:



The third step is to take the next smallest number and connect it on the graph as such:



Keep repeating the above steps, and eventually, your tree will be fully built.

[[4]](#footnote-3)

As can be seen above, each path has an assigned binary character. If the path diverges left, it has a 0. If it diverges right, it has a 1. You can now see how a letter can be encoded in such a way. For example, if you wanted to represent the letter ‘A’, the encoded letter would translate to “1011.”

One might be able to see the usefulness of a Huffman tree to store character values. Disk file storage becomes much more efficient, in that the encoded characters are easier to store than the uncompressed characters. It is also worth noting that the information can be uncompressed to be used in other programs. The space complexity of a Huffman tree is represented by O(k) for the fully constructed tree and O(n) for the decoded text. Assume k is equal to the number of alphabet symbols and n is equal to the number of nodes. O(k) will always prove to use less memory and disk space than O(n). This still comes at the cost of increased time complexity and reduced time efficiency. It is merely a question of whether the programmer favors space complexity over time complexity. Despite this ultimatum, Huffman coding trees are very useful in coding projects and programs.

One of the data structures that the Huffman tree will be compared to is referred to as a TREAP. TREAPs are a type of randomized binary search trees that combine the properties of a tree and a heap, hence the name “TREAP.” The complexity performance is centered around the expression O(logn) and the height of the tree. The data usually never diverges from this calculation. It is also worth noting that the properties are randomly assigned, similar to heap-order.[[5]](#footnote-4) Compared to Huffman coding trees, TREAPs have better time efficiency, but are slightly worse in space efficiency.

**Specifications**

IDE: CodeBlocks 20.03

Device: Alienware 15 R3

Processor: Intel(R) Core(TM) i5-7300HQ CPU @ 2.50 GHz

System Type: 64-bit operating system, x64-based processor

**Methodology/Code Samples**

This research project involves three separate data structures. The first one, built by me, is a Huffman tree that has been reconfigured to accept integers instead of chars. The second one is another Huffman tree borrowed from another source. It has also been reconfigured to accept integers. The third one is a TREAP borrowed from another researcher. All data structures will follow the same process of accepting an integer, or an array of integers, and constructing a tree with those values as nodes. The Huffman trees will test for both constant and random frequencies. The TREAP will only test for random frequencies, as that is how it was constructed by the other researcher.

Time complexity is measured using a complexity timer and recorder class. It was provided by the University of Akron Department of Arts and Sciences. The intended use was for students to utilize the files to aid in the research of their data structures. The timer is an object created from a timer class, and it starts running at the moment of creation. It can be stopped and restarted using the functions below:

timer.stop();

timer.restart();

The timer class is integral to measuring the time complexity of this program. The times of all three structures will be recorded and displayed for the user. Each data structure and frequency type will be tested 10 times, and the average will be taken of that set.

The main file initializes two different types of arrays: one that holds the integer values, and one that holds the frequency values. Then, it populates the array with integers using a for loop.

int huffSize1 = 5; // size of array

int huffArr1[huffSize1]; // array for huffman tree

int huffFreq1[] = {1, 1, 1, 1, 1}; // add frequencies for each int

for (int i = 0; i <= huffSize1; ++i) { // populate huffman array with ints

huffArr1[i] = i;

}

This block of code is repeated for each iteration of the Huffman tree. With each new iteration, either the value array, the frequency array, or both are modified. These arrays are then used in the encode() function.

std::cout << "Huffman Tree 1" << std::endl;

timer timer1; // create timer

encode(huffArr1, huffFreq1, huffSize1);

timer1.stop();

std::cout << "Time to complete: " << timer1.time() << std::endl;

The timer object is also created in this instance. In later similar calls, the timer is merely restarted. After the encode() function, the timer stops, and the resulting time value is outputted to the console.

The implementation and include files contain several structs that build and compare leaf nodes.

// Huffman tree node

struct MinHeapNode {

int data;

int freq;

MinHeapNode \*left, \*right;

MinHeapNode(int data, int freq) {

left = right = NULL;

this->data = data;

this->freq = freq;

}

};

// Comparison of two nodes.

struct compare {

bool operator()(MinHeapNode \*l, MinHeapNode \*r) {

return ((l->freq) > (r->freq));

}

};

These structs are called in the encode() function.

// Build Huffman Tree

void encode(int data[], int freq[], int size) {

struct MinHeapNode \*left, \*right, \*top;

// Create a min heap.

std::priority\_queue<MinHeapNode\*, std::vector<MinHeapNode\*>, compare> minheap;

// For each character create a leaf node and insert each leaf node in the heap.

for(int i = 0; i < size; i++)

minheap.push(new MinHeapNode (data[i], freq[i]));

// Iterate while size of min heap doesn't become 1

while(minheap.size() != 1) {

// Extract two nodes from the heap.

left = minheap.top();

minheap.pop();

right = minheap.top();

minheap.pop();

// Create internal node with added frequencies

// Add node to new heap

top = new MinHeapNode('$', (left->freq) + (right->freq));

top->left = left;

top->right = right;

minheap.push(top);

}

// Call function to print the codes.

printCodes(minheap.top(), " ");

}

It creates a min heap and inserts each value into that heap. It extracts two nodes from the heap to create the leaf nodes. It keeps doing this until it reaches the end. After that, it finally calls the printCodes() function.

//Print Huffman Codes

void printCodes(struct MinHeapNode\* root, std::string huffString){

// If it reaches end

if(!root)

return;

// Check if data exists

if(root->data != '$')

std::cout << root->data << ": " << huffString << std::endl;

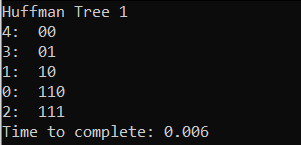
printCodes(root->left, (huffString + "0"));

printCodes(root->right, (huffString + "1"));

}

This function uses several recursive calls. This recursion may have an effect on the time complexity on the code. For this program, recursion was used for design efficiency.

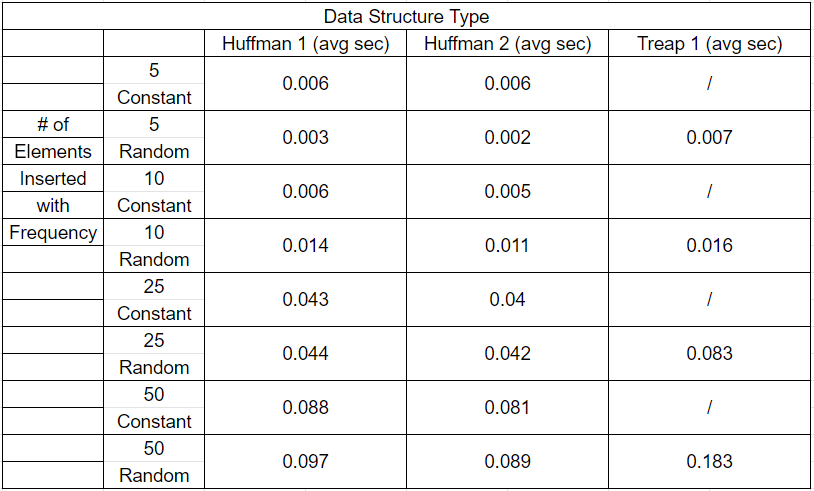
The resulting program will produce this in the console:



The final code is aligned next to its corresponding value. The tree can be graphed using the data presented here. As previously stated, this process is repeated 10 times for each unique data structure and frequency type. The average is taken of these results and recorded onto a spreadsheet.

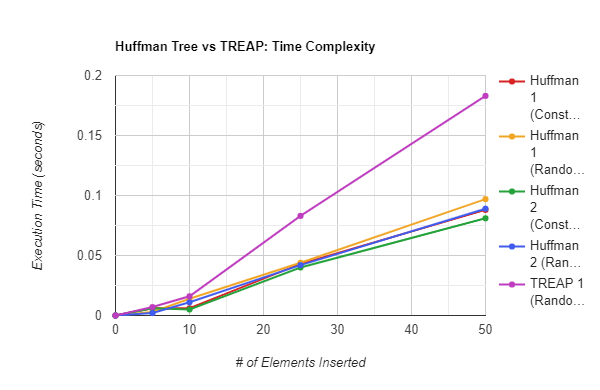
**Results**

The results of the program are listed below:



On average, the Huffman trees with constant frequencies performed better than the Huffman trees with random frequencies. This is in line with the definition of fixed-length codes. The second Huffman has a faster execution time, signaling better optimization. The only abnormality with the Huffman trees is with the lowest number of elements. In both Huffman trees, the random frequencies performed better than the constant frequencies with 5 elements. Unexpectedly, the TREAP data structure performed worse than the Huffman trees. This could be related to the optimization of the code.

The data has also been revisualized into a line graph shown below:



The difference between Huffman trees and TREAPs can be visibly seen with the line graph. Under 10 values, the various time complexity differences are minute. However, as more values are added, the TREAP line increases more in execution time. This contradicts what is known about Huffman coding trees and their weaknesses.

Interestingly, I ran into an odd issue during this research project. When testing the TREAP data structure, the complexity timer was producing odd, and often broken, number values. After half an hour of tinkering with the complexity timer class and the TREAP, I figured out what the issue was. When I was testing my other Huffman data structures earlier on my laptop, it was not charging. The laptop was charging, however, when I was testing the TREAP. Somehow, the laptop being plugged in affected the complexity timer’s values. This is a little warning to other researchers: hardware can still have a great effect on your project code. Be vigilant of every possible variable in your research.

**Conclusions**

The efficiency of these data structures is affected by multiple factors. They can range from a simple optimization fix to a hardware issue. Previously, I contacted the researcher about the time differences in their TREAP, and they sent back a fixed version of the program with a destructor. The fixed version is the version used in the testing above. In the older version, the objects were constructed and left there in the program. This caused significant time efficiency issues. Despite the fix, the current TREAP still lags behind the Huffman trees. The code could still be optimized to better compete with the Huffman trees.

One other worry I had about the data structures is the very subtle variables that might affect research output. As I mentioned above, the difference between my laptop being plugged in or not made a huge difference in the functionality of my code. This is why I listed my hardware and software specifications. If the program ran on a more advanced and faster computer, the time efficiency would be greatly improved.

The data shows that despite what we know about the strengths and weaknesses of Huffman trees, a poorly optimized data structure can nullify those strengths. The opposite is also true, in that a well-optimized data structure can be a powerhouse of efficiency. More testing and optimization can be done in the future to further improve the results of the Huffman tree. Knowing the intricacies of your data structure is important, but if you cannot properly design your program, those intricacies will be wasted knowledge.

**Sources**

Chakraborty, A. (2020, January 16). *Huffman Trees in Data Structure*. Tutorialspoint. Retrieved March 7, 2022, from https://www.tutorialspoint.com/huffman-trees-in-data-structure

“Huffman Coding Trees”. OpenDSA. Data Structures. College of Arts and Sciences, The University of Akron.

“Huffman Trees”. U1 ds05. Data Structures. College of Arts and Sciences, The University of Akron. slide 3.

“Treaps”. U1 ds06. Data Structures. College of Arts and Sciences, The University of Akron. Slides 4-24.

1. “Huffman Trees”. U1 ds05. Data Structures. College of Arts and Sciences, The University of Akron. slide 3. [↑](#footnote-ref-0)
2. Chakraborty, A. (2020, January 16). *Huffman Trees in Data Structure*. Tutorialspoint. Retrieved March 7, 2022, from https://www.tutorialspoint.com/huffman-trees-in-data-structure [↑](#footnote-ref-1)
3. “Huffman Coding Trees”. OpenDSA. Data Structures. College of Arts and Sciences, The University of Akron. [↑](#footnote-ref-2)
4. Mishra, N. (2016, July 18). *Huffman coding algorithm with example*. The Crazy Programmer. Retrieved March 7, 2022, from https://www.thecrazyprogrammer.com/2014/09/huffman-coding-algorithm-with-example.html [↑](#footnote-ref-3)
5. “Treaps”. U1 ds06. Data Structures. College of Arts and Sciences, The University of Akron. Slides 4-24. [↑](#footnote-ref-4)