Huffman Coding & Adaptive Huffman Coding Investigation

Mark Goldwater, Sherrie Shen, and Hyegi Bang

October 21, 2019

1 What is Huffman Coding

1.1 Introduction

Huffman Coding is a method of data compression and is used in communication systems. The algorithm itself takes in as inputs the frequencies of each of the symbols in a given message desired to be transmitted and outputs a prefix code (described in the next subsection) which encodes the using the fewest possible bits among all possible binary prefix codes for the given symbols [3]. This investigation will describe the Huffman Coding algorithm, show an example of it being applied as well as our Java implementation of the algorithm. Moreover, we also prove the optimality of the Huffman's algorithm and adaptive Huffman Coding for a situation in which the message is being encoded as it is slowly being learned what the message is. Enjoy!

1.2 Prefix Codes

In encoding a message, one way to be sure that no string corresponds to more than one sequence of letters is to create a prefix code. In a prefix code, letters are encoded such that the bit string for a letter never occurs at the beginning of the code for another letter's encoding [3].

1.3 The Algorithm

The pseduocode for the software implementation of Huffman Coding can be summarized as below:

```
1. Compute the histogram of letters in a given message.
2. Initialize a node for each letter with the weight equal to its frequency and label to be the
   respective character.
3. Initialize a min priority queue (where relative size is determined by comparing frequency)
   and add the newly created nodes to the queue.
4. while size of queue is greater than 1:
   begin
     (a) Pop two elements with the lowest weights from the queue.
     (b) Create a new node with the weight equal to the sum of the weights of the two popped
         nodes. Assign the popped node with larger weights to be the left child of the newly
         created node and assign the popped node with smaller weights to be the right child.
     (c) Put the newly created node in the queue.
```

5. When there is only one node left in the queue, this node is the root of the Huffman prefix code tree.

6. The Huffman coding for the letter i is the concatenation of the labels of the edges in the unique path from the root to the node with label letter i. The edge connecting a parent to its left child is labeled as 0 and the edge connecting a parent to its right child is labeled as 1.

7. Given the root node of the Huffman tree, traverse the tree with depth first search algorithm to find the encoding for each letter.

8. Construct the encoded message using the encodings for each letter.

1.4 Example

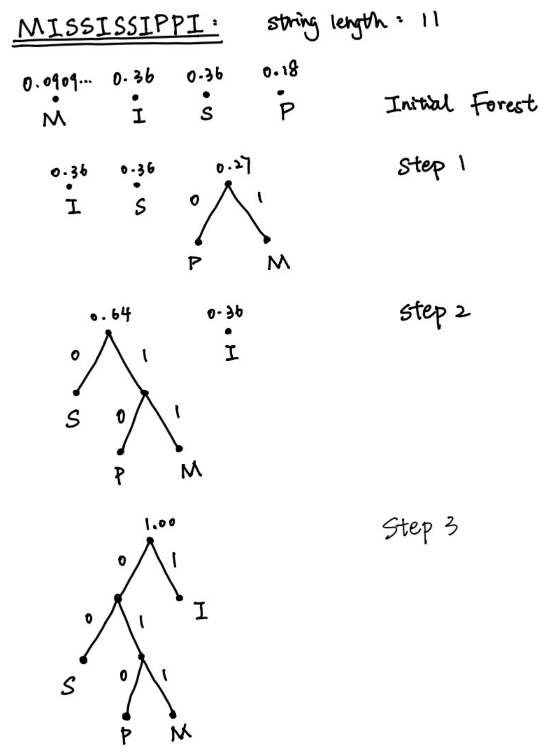
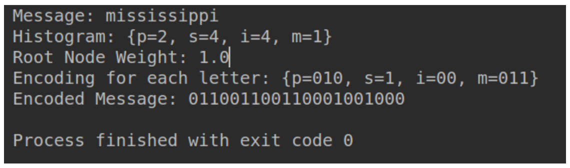


Figure 1: Huffman Enconding of "mississippi" using trees

The Figure 1 shows Huffman encoding of "mississippi" by building a tree based on the frequency of each letters. By reading the final Huffman tree, M is encoded as 1001, I is encoded as 11, S is encoded as 0 and P is encoded as 101, resulting "mississippi" to be encoded as 100111001100111011011 1.

1.5 Implementation of Huffman Coding

We implemented the Huffman Coding Algorithm in java with a min priority queue to construct a Huffman Tree and depth first search to fetch the encoding for each letter. The specific code implementation is attached in the Appendix section and can also be found on our GitHub repository at https://github.com/xieruishen/Huffman-Coding.



```
Message: mississippi
Histogram: {p=2, s=4, i=4, m=1}
Root Node Weight: 1.0
Encoding for each letter: {p=010, s=1, i=00, m=011}
Encoded Message: 0110011001100010010 00

Process finished with exit code 0
```

Figure 2: Huffman Enconding of "mississippi" from the java implementation.

2 Proving the Optimality of Huffman Coding [1]

In order to quantify the optimality of a given prefix tree T, which encodes a certain alphabet C, we can calculate the average number of bits needed to encode a single character. Let $p(x)$ be the probability of seeing the character x in the encoded text, and $d_T(x)$ denote the length of of the encoded letter (which can be measured by the depth of said letter in the prefix tree T). The average number of bits per symbol can be calculated with the following formula:

$$B(T) = \sum_{x \in C} p(x) d_T(x) \tag{1}$$

We can state the proof of optimality for Huffman code as the following: we want to show that given an alphabet C and the frequencies of occurrence $p(x)$ for each character $x \in C$, the Huffman Code algorithm computes a prefix code T that minimizes the expected length of the encoded bit-string i.e. $B(T)$.

The general approach to this proof is to show that any optimal tree which differs from that constructed from Huffman's algorithm can be converted to one equal to Huffman's tree without increasing its cost. This is accomplished by identifying places where the two solutions differ and modifying the non-greedy solution so that it can become the greedy solution, and show that this modification does not increase the cost of the tree.

The approach to this is based on some essential observations:

1. A Huffman tree is a **full binary tree** which means that every internal node has exactly two children. Note that having an internal node with one child is less efficient in cost than replacing this node with its child without increasing the tree's cost. As a result, we can safely limit our consideration to only full binary trees.

2. Any optimal code tree can be assumed to have the two characters with the lowest probabilities as siblings at the maximum depth of the tree (Proven below)

With these pieces of information we can merge these two characters into a single "meta-character" whose probability is the sum of the individual frequencies. Then, we will have one less character in the alphabet and be in a position to apply induction on the remaining $n - 1$ characters.

First, lets prove our second observation.

Lemma 1: Consider the two characters, node x and y with the smallest probabilities. Then there is an optimal code tree in which these two characters are siblings at the maximum depth in the tree.

Proof. Let T be any optimal prefix code tree, and let b and c be two node siblings at the maximum depth of the tree (there may be many such siblings, and if so, we can pick any such pair). If $(x, y) = (b, c)$ then we are done!

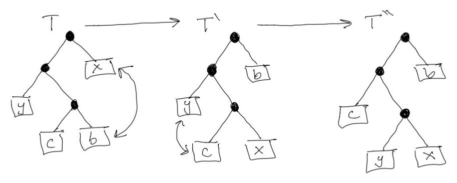


Figure 3: This figure demonstrates how switching of character nodes can produce a code tree with the two nodes of least frequency as siblings at the maximum depth of the tree

Let $p(n)$ be the weight of a particular node. For leaves node, $p(n)$ is equal to the frequency of the label occurring in the message. For internal nodes, $p(n)$ is equal to the sum of weights of its two children. If we cannot find a pair of (b, c) such that $(x, y) = (b, c)$, WLOG pick any pair of leaf nodes of this tree such that $p(b) \leq p(c)$ and $p(x) \leq p(y)$. Since x and y have the two smallest probabilities it follows that $p(x) \leq p(b)$ and $p(y) \leq p(c)$. Because b and c are at the deepest level of the tree we know that $d_T(b) \geq d_T(x)$ and $d_T(c) \geq d_T(y)$. Thus, we have $p(b) - p(x) \geq 0$ and $d_T(b) - d_T(x) \geq 0$, so their product is non-negative. Now let's suppose that we switch the positions of x and b in the tree T resulting in T' as shown above in Figure 3. Let $B(T)$ be the average cost of bits used to encode a character. Let's see how the cost changes as we go from T to T':

$$\begin{aligned}
B(T') &= B(T) - (\text{old cost for b and x}) + (\text{new cost for b and x}) \\
&= B(T) - (p(x)d_T(x) + p(b)d_T(b)) + (p(x)d_T(b) + p(b)d_T(x)) \\
&= B(T) + p(x)(d_T(b) - d_T(x)) - p(b)(d_T(b) - d_T(x)) \\
&= B(T) - (p(b) - p(x))(d_T(b) - d_T(x)) \\
&\leq B(T)
\end{aligned} \tag{2}$$

Note that the last step in the above algebra follows because we have shown that $(p(b) - p(x))(d_T(b) - d_T(x)) \geq 0$. From this cost calculation it is clear that the cost does not increase (and cannot decrease either because we assumed T was already optimal). By the same logic, we can again switch y and c to obtain the final tree T'' in Figure 3. This will give us $B(T'') \leq B(T')$ and we therefore proves Lemma 1. ■

In order to show that the entire Huffman tree is optimal, we need to extend this argument using induction. In order to reduce from n characters to $n-1$, we will do the same combination that Huffman's algorithm does: merge characters x and y into a new meta-character z, whose probability is the sum of x and y.

Lemma 2: Let T_n be any prefix-code tree that satisfies the property of Lemma 1 and T_{n-1} be the tree that results by replacing the two nodes of least frequency and their parent with a single leaf node such that $p(z) = p(x) + p(y)$. Then $B(T_n) = B(T_{n-1}) + p(z)$.

Proof. Let d denote the depths of x and y in T_n. Because of the way we are creating z it is at depth $d - 1$ in T_{n-1} as shown below in Figure 4.

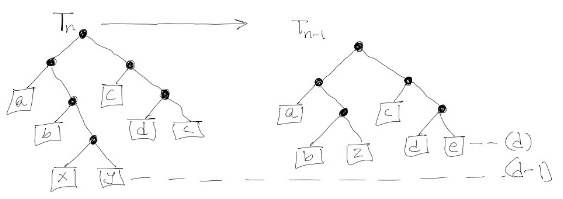


Figure 4: This figure shows the process shows the process of combing the nodes representing characters x and y into the meta-character z.

Because z replaces x and y the costs of the two trees can be expressed as the following equation:

$$\begin{aligned} B(T_n) &= B(T_{n-1}) - (\text{z's cost in } B(T_{n-1})) + (\text{x and y's costs in } B(T_n)) \\ &= B(T_{n-1}) - p(z)(d-1) + (p(x)d + p(y)d) \\ &= B(T_{n-1}) - p(z)(d-1) + p(z)d \\ &= B(T_{n-1}) + p(z) \end{aligned} \tag{3}$$

■

The cost of trees T_n and T_{n-1} differ only by the constant term $p(z)$ which does not depend on the trees structure unlike $B(T_{n-1})$. Therefore, minimizing the cost of T_n is equivalent to minimizing the cost of T_{n-1}. This fact allows us to prove our main result.

Finally, let's address the main proof: Prove Huffman's algorithm produces an optimal prefix code tree.

Proof. Let $P(n)$ be the statement that Huffman's algorithm produces an optimal prefix code tree for n characters.

Basis Step: $P(n)$ is true for $n = 1$ since there is only one tree possible.

Inductive Step: If $n \geq 2$ then we know we have the two characters of least frequency as siblings at the maximum depth of the tree by Lemma 1. Huffman's algorithm replaces these nodes by a character z whose probability is the sum of their probabilities. Let's assume that $P(n-1)$ is true for some $n \geq 2$. Let's call the tree representing this Huffman code T_{n-1}. Replacing node z with nodes x and y (essentially we are going from T' to T in Figure 4) creates a tree T_n whose cost is higher by $p(z) = p(x) + p(y)$. Since T_{n-1} is optimal and the optimality of T_n only depends on this term being optimal because $p(z)$ does not depend on the tree's structure, T_n is also optimal completing the inductive step. ■

3 Adaptive Huffman Coding

3.1 Algorithm

For the final part of our investigation, we explored the FGK algorithm for Adaptive Huffman Coding. Adaptive Huffman Coding is necessary when there is no prior knowledge of the frequency of each symbol in the message. The tree dynamically adjusts the Huffman Tree as the data is being transmitted. The Huffman tree for adaptive Huffman coding has the following properties:

- Every node except for the root node has a sibling.

- For nodes of the same level (siblings), nodes on the right have higher order than nodes on the left. In other words, On each level, the node farthest to the right will have the highest order although there might be other nodes with equal weights.
- Nodes at a greater height have higher order than nodes at lower height.
- Internal nodes contain weights equal to the sum of their children's weights
- Node's with higher weights have higher orders.

The pseudocode for the Adaptive Huffman Code algorithm can be summarized as follows:

```
1. Initialize the tree with a “0-Node”.
2. For the incoming letter $i$,
    • If we have not seen this letter before, create a new node for this element and add
      it as the right child of the 0-Node. We initialize the weight for this node to be 1. Set
      current node to be the parent of the new node. Create a new 0-Node as the left child of
      the 0-Node. Move to the check step.
    • If we have seen this letter before, set current node to be the node for this element
      and move to check and update step.
3. Check and Update Step:
    (a) If there is any node with a higher order (larger height or further to the right) that
        has the same weight as current node, swap current node with the one with the highest
        order. When swapping two nodes, their corresponding subtree is also carried with the
        node.
    (b) Increment the weight for current node by 1.
    (c) Set current node to be the parent of current node.
   while current node is not Null:
   begin
    (a) If there is any node with a higher order (larger height or further to the right) has
        the same weight as current node, swap current node with the one with the highest order.
        When swapping two nodes, their corresponding subtree is also carried with the node.
    (b) Recalculate the weight by adding the weights of the child nodes.
    (c) Set current node to be the parent of current node.
   end
```

{The Adaptive Huffman coding for the letter i is the concatenation of the labels of the edges in the unique path from the root to the node with label letter i. The edge connecting a parent to its left child is labeled as 0 and the edge connecting a parent to its right child is labeled as 1.}

3.2 Example

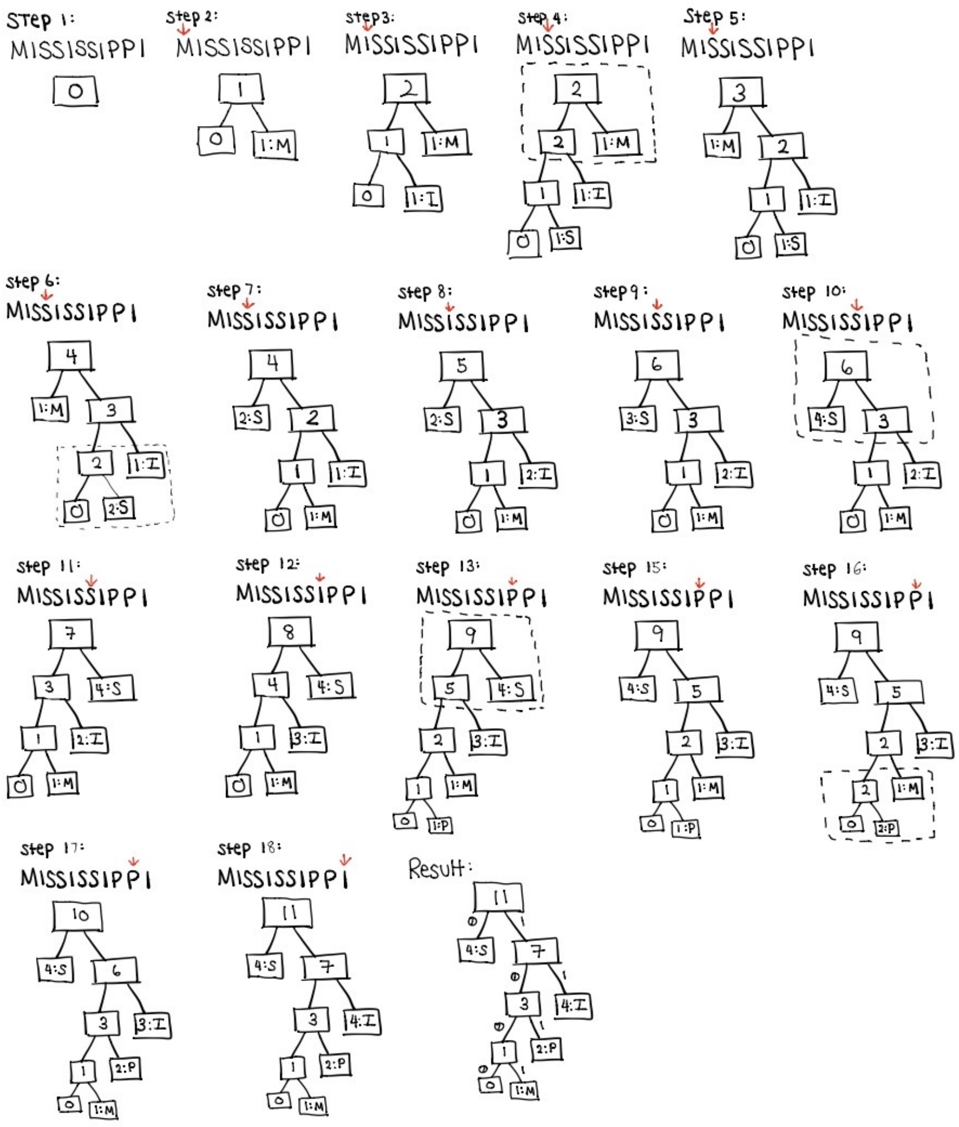


Figure 5: Adaptive Huffman Enconding of ”mississippi”.

Figure 5 presents the steps taken to build Adaptive Hoffman tree for the word ”Mississippi”. In order to implement Adaptive Hoffman Encoding algorithm to the message ”Mississippi” which are explained below. The red arrow in the Figure 4 indicates the node data that is being inserted during the corresponding step.

- Step 1: FGK Huffman tree begins with a 0-node, which is used to identify a newly inserted character. A new node is always inserted as a right child of a 0-node.
- Step 2: Since it is the first occurrence of the character, M, a node with value of 1 and labelled

M is inserted as a right child of the 0-node, root node, and another 0-node as its sibling. The value of parent node is the sum of its children: being 0 + 1 = 1.

- Step 3: Since it is the first occurrence of the character, I, a node with value of 1 and labelled I is be inserted as a right child of a 0-node. Another 0-node is added as a sibling child of I, updating its parent node with a value of 1. Therefore, the value of root node is updated to 2. The nodes are updated as a sum of its child nodes until the root node is updated.
- Step 4: Since it is the first occurrence of the character S, a node with value of 1 is ind labelled S implemented as a right child of a 0-node, the sibling child of node I. Another 0-node is added as a sibling child of node S, updating its parent node with a value of 1. As we update the tree, we encounter the problem where the nodes of the tree are not in the order of non-decreasing frequency: [0, 1, 1, 1, 2, 1] as shown inside the dashed box.
- Step 5: To order the frequency in non-decreasing order, re interchange the children node of the root node so that the frequency is in the order of [0,1,1,1,1,2,3].
- Step 6: Since a node with a value of frequency S is present in the tree, we need to increment the frequency 1, resulting a value of 2. This breaks the non-decreasing order of the tree: [0,2,2,1].
- Step 7: Before updating the node, we need to re-order the tree. We swap the node that needs to be incremented with highest order of the node with the same weight. In this case, we will swap the node with labelled S with node with labelled M since node labelled M is the node in highest order with a value of 1. After the swapping is completed, we then increment the node of S by 1, resulting a value of 2. The parent node, root node, is also updated by the sum of its child node.
- Step 8: Since a node labelled I is present in the tree, we need to increment the frequency by 1, resulting a value of 2. The parent node is updated to 3 and the root node is updated to 5. The tree has a non-decreasing order of weights: [0,1,1,2,3,3,5].
- Step 9: Since a node with a value of frequency S is present in the tree, we need to increment the frequency 1, resulting a value of 3. We compute the each node by summing the weights of its children. We do the same for the parent and repeat until we reach the root node.
- Step 10: Since a node with a value of frequency S is present in the tree, we need to increment the frequency 1, resulting a value of 4. The dashed box violates the sibling rule as right node should always be greater or equal than the left node.
- Step 11: To encounter the problem, we interchange the nodes that violates the sibling rule, which in this case are the child nodes of the root node. Therefore, the tree has a non-decreasing order of frequency: [0,1,1,3,4,4,8].
- Step 12: Since a node labelled I is present in the tree, we need to increment the frequency by 1, resulting a value of 3. The parent node is updated to 4 and the root node is updated to 8.
- Step 13: Since it is the first occurrence of the character, P, a node with value of 1 and labelled P is be implemented as a right child of a 0-node, the left child of the root node. Another 0-node is added as a sibling child of P, updating its parent node with a value of 1. The parent nodes where its child node are updated are also updated. During this process, the nodes in the dashed box violates the sibling property as the node with smaller order has a greater value than the node with higher order.

- Step 14: We interchange the nodes that violates the sibling property, in this case are the child nodes of root node, resulting a frequency order of [0,1,1,1,2,3,4,5,9].
- Step 15: Since a node with a value of frequency P is present in the tree, we need to increment the frequency by 1, resulting a value of 2. When the parent node of this node is updated, we encounter a problem where the left node, parent node of node labelled P is greater than its sibling node, node labelled M.
- Step 16: Before we update the node labelled P, we need to reorder the tree by swapping with the node that has the same weights and in the highest order, which in this case is node M. After the swap, we update the node with label P by incrementing it, resulting a value of 2. We compute each node by summing the weights of its children. We do the same for the parent and repeat until we reach the root node.
- Step 17: Since a node with a value of frequency I is present in the tree, we need to increment the frequency 1, resulting a value of 4. We update all the node where any of its child node are updated until it reaches the root node. No other modifications has to be done since the updated tree has a non-decreasing order of frequency.

  We are able to encode each letters using the Adaptive Hoffman Tree of the word "Mississippi" built above. M is encoded as 1001, I is encoded as 11, S is encoded as 0 and P is encoded as 101, resulting "Mississippi" to be encoded as 1001110011001110110111.

4 Appendix

4.1 Code

```
import java.util.*;

// Class for a tree node
class HuffmanNode {

    HuffmanNode left; // Pointer to left child
    HuffmanNode right; // Pointer to right child
    double weight; // Frequency
    String label; // Name of symbol
}

// Class with comparitor function to compare mannitudes of node frequencies
class HuffmanNodeComparator implements Comparator<HuffmanNode> {
    public int compare(HuffmanNode x, HuffmanNode y)
    {
        if (x.weight - y.weight > 0){
            return 1;
        }
        else if(x.weight - y.weight == 0){
            return 0;
        }
        else {
            return -1;
        }
    }
```

```java
}


public class HuffmanCoding {

   // Function to recursively construct the Huffman Tree
    private static HuffmanNode constructHuffmanTreeHelper(PriorityQueue<HuffmanNode>
        huffmanNodes){

      // If there is one node, simply return it
        if(huffmanNodes.size() == 1){
            return huffmanNodes.remove();
        }

        // Pop off two nodes with smalles frequencies
        HuffmanNode smaller = huffmanNodes.remove();
        HuffmanNode larger = huffmanNodes.remove();

        // Create a new node which has these two as children (larger weight on left side)
        HuffmanNode combined = new HuffmanNode();
        combined.left = larger;
        combined.right = smaller;
        combined.weight = larger.weight + smaller.weight;

        // Add this new combined node back to the priority queue
        huffmanNodes.add(combined);

        // Recursivly construct the tree!
        return constructHuffmanTreeHelper(huffmanNodes);
    }

    // Function to set up and kick off recursive creation of the Huffman Tree
    public static HuffmanNode constructHuffmanTree(HashMap<String, Integer> histogram,
        int numAlphabet, int messageLen){

      // Instantiate priority queue to be filled with nodes
        PriorityQueue<HuffmanNode> huffmanNodes = new PriorityQueue<>(numAlphabet, new
            HuffmanNodeComparator());

        // For each element in message histogram, create node with label, and weight
            (frequency) and set
        // children to null and add the node to the priority queue
        for (HashMap.Entry mapElement : histogram.entrySet()){
            HuffmanNode node = new HuffmanNode();
            node.label = (String)mapElement.getKey();
            node.weight = (Integer)mapElement.getValue() / (double) messageLen;
            node.left = null;
            node.right = null;
            huffmanNodes.add(node);
        }

        // Kick off recursive construction of tree
        return constructHuffmanTreeHelper(huffmanNodes);
    }
```

```
// Depth first search the tree keeping track of left/right child traversals and then
    store final label in a hash map
private static void getEncodings(HashMap<String, String> encodings, HuffmanNode
    current, String encoding){
    if (current.label != null){
        encodings.put(current.label, encoding);
        return;
    }
    if (current.left != null){
        getEncodings(encodings, current.left, encoding+"0");
    }
    if (current.right != null){
        getEncodings(encodings, current.right, encoding+"1");
    }
}

// Function to create hashmap where key is symbol and value is number if times
    appeared in the message
private static HashMap<String, Integer> getHistogram(String message){
    HashMap<String, Integer> histogram = new HashMap<>();
    for(int i = 0; i < message.length(); i++){
        String alphabet = Character.toString(message.charAt(i));
        if(histogram.containsKey(alphabet)){
            histogram.put(alphabet,histogram.get(alphabet)+1);
        }
        else{
            histogram.put(alphabet,1);
        }
    }
    return histogram;
}

// Function to encode a message with Huffman code with a hashmap that has letter to
    encoding
private static String encodeMessage(String message, HashMap<String, String>
    encodingMap){
    StringBuilder encodedMesage = new StringBuilder("");
    for(int i = 0; i < message.length(); i++){
        String letter = Character.toString(message.charAt(i));
        encodedMesage.append(encodingMap.get(letter));
    }
    return encodedMesage.toString();

}

public static void main(String[] args) {
  // Message to encode
    String message = "mississippi";
    System.out.println("Message: " + message);

    // Create and print message histogram to the terminal
    HashMap<String, Integer> histogram = getHistogram(message);
    System.out.println("Histogram: " + histogram);
```

```
        // Construct the Huffman Tree and return the root node and print its weight to
            the terminal
        HuffmanNode root = constructHuffmanTree(histogram, histogram.size(),
            message.length());
        System.out.println("Root Node Weight: " + root.weight);

        // DFS the Huffman Tree and store the codes in a hashmap and print to the terminal
        HashMap<String, String> symbolCodes = new HashMap<>();
        getEncodings(symbolCodes, root, "");
        System.out.println("Encoding for each letter: " + symbolCodes);

        // Encode the message
        System.out.println("Encoded Message: " + encodeMessage(message, symbolCodes));
    }
}
```

References

[1] Mount, Dave "CMSC 451: Lecture 6 Greedy Algorithms: Huffman Coding", 2017, *PDF* file.

[2] Low, Jonathan. Adaptive Huffman Coding, www2.cs.duke.edu/csed/curious/compression/adaptivehuff.html?fbclid=IwAR1xXD-cAhVUWRkdUbt6NhMo99E60MT_pzdKxd_mjxjMX0GClwcPD_vO_eQ.

[3] Rosen, Kenneth H. Discrete Mathematics and Its Applications., 2003. Print.