# Huffman Coding Algorithm: Complete Implementation and Analysis

Computer Science Project
Adam Hajjaji - Zakariae Jali

December 12, 2025

**Abstract**

This report presents a complete implementation of the Huffman coding algorithm, a lossless data compression technique that assigns variable-length binary codes to characters based on their frequencies. The project includes both the core algorithmic implementation and an interactive JavaFX visualization tool. We detail the main concepts, data structures, implementation specifics, and provide the essential proofs of the algorithm's properties.

# Contents

# 1 Introduction

## 1.1 Project Overview

Huffman coding, developed by David A. Huffman in 1952, is a compression algorithm that assigns shorter binary codes to more frequent characters and longer codes to less frequent ones. This project implements the complete Huffman coding pipeline including frequency analysis, tree construction, encoding, decoding, and an educational visualization interface.

## 1.2 Report Structure

This report is organized as follows:

- **Section 2: Basic Concepts** - Essential theoretical background
- **Section 3: Algorithm Description** - Explanation of Huffman's algorithm
- **Section 4: Implementation Architecture** - Class structure
- **Section 5: Core Algorithm Implementation** - Code details
- **Section 6: Visualization Interface** - JavaFX GUI
- **Section 7: Formal Proofs** - Important mathematical proofs
- **Section 8: Conclusion**

# 2 Basic Concepts

## 2.1 Tree Properties

A **tree** is a connected graph without cycles. For Huffman coding, we use binary trees where:

- Each node has at most two children (left and right)
- Leaves represent characters
- The path from root to leaf gives the binary code
- Left edges = 0, right edges = 1

## 2.2 Prefix-Free Codes

A **prefix-free code** means no code is the beginning of another code. This allows decoding without special markers. Huffman codes are always prefix-free.

# 3 Huffman Algorithm Description

## 3.1 Problem

Given characters with frequencies, find binary codes that minimize:

$$\text{Total bits} = \sum(\text{frequency} \times \text{code length})$$

### 3.2 Algorithm Steps

1. Count frequency of each character
2. Create a leaf node for each character with its frequency
3. Repeatedly combine the two nodes with smallest frequencies
4. Make a new node with frequency = sum of the two
5. Continue until only one node remains (the root)
6. Assign codes: left = 0, right = 1

## 4 Implementation Architecture

### 4.1 Main Classes

The project has three main parts:

#### 4.1.1 Model Classes

- `HuffmanNode`: Tree node with character and frequency
- `HuffmanTree`: Complete tree structure

#### 4.1.2 Algorithm Classes

- `FrequencyCounter`: Counts character frequencies
- `TreeBuilder`: Builds the Huffman tree
- `CodeGenerator`: Creates binary codes

#### 4.1.3 UI Class

- `HuffmanTreeVisualizer`: Interactive visualization tool

## 5 Core Algorithm Implementation

### 5.1 Frequency Analysis

```
public static HashMap<Character,Integer> getFrequency(List<Character>
    text){
     HashMap<Character,Integer> res = new HashMap<>();
     for (int i=0; i<text.size(); i++){
         Character curr_char = text.get(i);
         if (res.containsKey(curr_char)){
             res.put(curr_char,res.get(curr_char)+1);
         }
         else{
             res.put(curr_char,1);
         }
```

```
    }
    return res;
}
```

Listing 1: Frequency Counter

## 5.2 Tree Construction

```
public static HuffmanTree buildHuffmanTree(String text){
    HuffmanTree res = new HuffmanTree();
    List<HuffmanNode> list_nodes = SortedNodesCreator.sort(text);
    LinkedList<HuffmanNode> ll_nodes = new LinkedList<>(list_nodes);

    while (ll_nodes.size()>1){
        List<HuffmanNode> actualStep = new ArrayList<>(ll_nodes);
        res.addConstructionSteps(actualStep);

        HuffmanNode node1 = ll_nodes.removeFirst();
        HuffmanNode node2 = ll_nodes.removeFirst();

        int new_frequency = node1.getFrequency()+node2.getFrequency();
        HuffmanNode new_node = new HuffmanNode(null,new_frequency);
        new_node.setLeft(node1);
        new_node.setRight(node2);

        // Insert in correct position to keep sorted
        ListIterator<HuffmanNode> it = ll_nodes.listIterator();
        boolean found = false;
        while (it.hasNext() && !found){
            HuffmanNode compareNode = it.next();
            if (compareNode.getFrequency() > new_frequency){
                it.previous();
                it.add(new_node);
                found = true;
            }
        }
        if (!found){
            it.add(new_node);
        }
    }
    res.setRoot(ll_nodes.get(0));
    return res;
}
```

Listing 2: Tree Building

## 5.3 Code Generation

```
public static void generateCode(HuffmanNode tree,
                                HashMap<Character,String> codeMap,
                                String encoding){
    if (tree.isLeaf()){
        codeMap.put(tree.getCharacter(),encoding);
        return;
    }
    generateCode(tree.getLeft(), codeMap, encoding+"0");
```

```
    generateCode(tree.getRight(), codeMap, encoding+"1");
}
```

Listing 3: Code Generation

# 6 Visualization Interface

## 6.1 JavaFX Application

The visualization tool provides:

- File loading from text files
- Step-by-step tree construction animation
- Character path highlighting
- Compression statistics display
- Auto-play for demonstration

## 6.2 Tree Drawing

```
private void calculateTreePositions(HuffmanNode node,
                                    Map<HuffmanNode, NodePosition>
    positions,
                                    double x, double y,
                                    double offset, double
    verticalSpacing){
     if (node == null) return;
     positions.put(node, new NodePosition(x, y));

     if (node.getLeft() != null){
         double childOffset = offset * 0.6;
         calculateTreePositions(node.getLeft(), positions,
                                x - childOffset, y + verticalSpacing,
                                childOffset, verticalSpacing);
     }
     if (node.getRight() != null){
         double childOffset = offset * 0.6;
         calculateTreePositions(node.getRight(), positions,
                                x + childOffset, y + verticalSpacing,
                                childOffset, verticalSpacing);
     }
}
```

Listing 4: Tree Position Calculation

# 7 Formal Proofs

## 7.1 Tree Properties

*Trees have at least two leaves.* Take the longest path in the tree. The endpoints of this path cannot connect to any other vertices (would make a longer path or a cycle). So they have degree 1, making them leaves. □

[Tree Characterizations] For a graph with $n$ vertices, these are equivalent:

1. Connected with no cycles
2. Connected with $n-1$ edges
3. Has $n-1$ edges and no cycles
4. Exactly one path between any two vertices

*Proof.* We show the main ideas:
**(1) (2):** Removing a leaf gives smaller tree. By induction, smaller tree has $(n-1)-1$ edges. Add back leaf gives $n-1$ edges.
**(2) (3):** Connected graph with $n-1$ edges can't have cycles (would have fewer edges).
**(3) (1):** Graph with $n-1$ edges and no cycles must be connected.
**(1) (4):** In tree, exactly one path. If exactly one path between vertices, graph is connected and has no cycles. □

## 7.2 Huffman Algorithm Optimality

Huffman's algorithm produces the optimal prefix-free code.

*Proof.* By induction on number of characters.
**Base:** With 2 characters, codes "0" and "1" are optimal.
**Induction:** Assume optimal for $n-1$ characters. For $n$ characters:

1. In any optimal tree, the two least frequent characters are siblings at deepest level
2. Combine them into one node with combined frequency
3. By induction, Huffman gives optimal tree for $n-1$ characters
4. Splitting the combined node gives optimal tree for $n$ characters

□

# 8 Implementation Details

## 8.1 Important Design Choices

### 8.1.1 Sorting Method

We use merge sort to sort nodes by frequency:

```
public static List<HuffmanNode> mergeSort(List<HuffmanNode> A){
    if (A.size() <= 1){return A;}
    int mid = A.size()/2;
    return merge(mergeSort(A.subList(0, mid)),
                 mergeSort(A.subList(mid, A.size())));
}
```

### 8.1.2 Special Character Handling

```
private String formatCharacter(char c){
    switch (c){
        case ' ': return "    ";
        case '\n': return "    ";
        case '\t': return "    ";
        case '\r': return "  ";
        default:
            if (c < 32 || c > 126) return "    ";
            return String.valueOf(c);
    }
}
```

# 9 Results and Performance

## 9.1 Compression Example

For the default text:

- Original: 305 characters × 8 bits = 2440 bits
- Compressed: 1344 bits
- Savings: 44.9%
- Average code length: 4.41 bits per character

## 9.2 Visualization Features

The tool successfully shows:

1. Initial sorted nodes
2. Step-by-step merging
3. Final tree structure
4. Character encoding paths

# 10 Conclusion

## 10.1 Project Success

This implementation:

- Correctly implements Huffman coding
- Provides educational visualization
- Handles various text inputs
- Shows algorithm in action