Adaptive Huffman Coding Implementation Analysis

Mostafa Nashaat
ID: 202202075

April 10, 2025

Abstract

This report provides a detailed analysis of an adaptive Huffman coding implementation based on the FGK (Faller-Gallager-Knuth) algorithm. The report explains both the theoretical foundations and the practical implementation details. Three test cases are documented and analyzed to demonstrate the algorithm's performance in different scenarios. The report also includes a comprehensive breakdown of the code structure and the dynamics of the Huffman tree as it adapts to incoming symbols.

Contents

1 Introduction

1.1 Background

Huffman coding is a lossless data compression algorithm that assigns variable-length codes to input characters based on their frequencies. Traditional Huffman coding requires two passes over the data: one to calculate frequencies and build the tree, and another to perform the actual encoding.

Adaptive Huffman coding eliminates the need for this two-pass approach by dynamically updating the coding tree as data is processed. This makes it particularly useful for streaming applications and scenarios where the entire dataset is not available beforehand.

1.2 The FGK Algorithm

The implementation analyzed in this report follows the FGK (Faller-Gallager-Knuth) algorithm, which maintains a self-balancing binary tree with the following properties:

- Nodes are ordered by weight (frequency of occurrence)
- The sibling property is maintained: nodes with the same weight are grouped together
- A special NYT (Not Yet Transmitted) node represents symbols that haven't appeared yet
- When a new symbol appears, a subtree is created with the symbol and a new NYT node
- After processing a symbol, weights are updated and nodes may be swapped to maintain ordering

2 Implementation Details

2.1 Code Structure

The implementation consists of several Java classes:

- **AdaptiveHuffman**: Main driver class that coordinates encoding and decoding
- **Encoder**: Handles the encoding process
- **Decoder**: Handles the decoding process
- **HuffmanTree**: Manages the tree structure and node swapping logic

- **Node**: Represents individual nodes in the Huffman tree
- **HuffmanTreeVisualizer**: Provides visualization of the tree structure

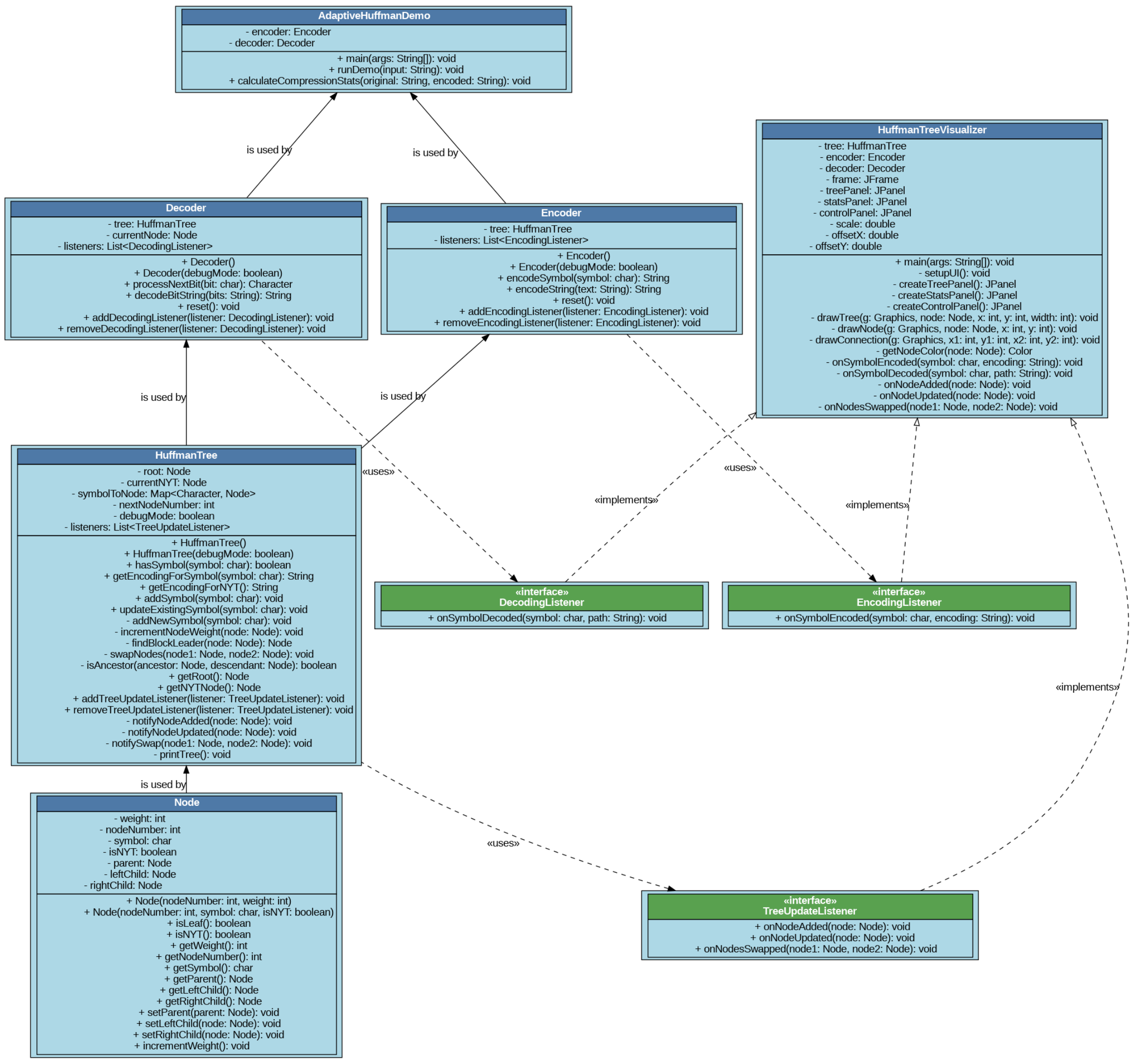

AdaptiveHuffmanDemo
- encoder: Encoder
- decoder: Decoder
+ main(args: String[]): void
+ runDemo(input: String): void
+ calculateCompressionStats(original: String, encoded: String): void
is used by
is used by
Decoder
- tree: HuffmanTree
- currentNode: Node
- listeners: List<DecodingListener>
+ Decoder()
+ Decoder(debugMode: boolean)
+ processNextBit(bit: char): Character
+ decodeBitString(bits: String): String
+ reset(): void
+ addDecodingListener(listener: DecodingListener): void
+ removeDecodingListener(listener: DecodingListener): void
Encoder
- tree: HuffmanTree
- listeners: List<EncodingListener>
+ Encoder()
+ Encoder(debugMode: boolean)
+ encodeSymbol(symbol: char): String
+ encodeString(text: String): String
+ reset(): void
+ addEncodingListener(listener: EncodingListener): void
+ removeEncodingListener(listener: EncodingListener): void
HuffmanTreeVisualizer
- tree: HuffmanTree
- encoder: Encoder
- decoder: Decoder
- frame: JFrame
- treePanel: JPanel
- statsPanel: JPanel
- controlPanel: JPanel
- scale: double
- offsetX: double
- offsetY: double
+ main(args: String[]): void
- setupUI(): void
- createTreePanel(): JPanel
- createStatsPanel(): JPanel
- createControlPanel(): JPanel
- drawTree(g: Graphics, node: Node, x: int, y: int, width: int): void
- drawNode(g: Graphics, node: Node, x: int, y: int): void
- drawConnection(g: Graphics, x1: int, y1: int, x2: int, y2: int): void
- getNodeColor(node: Node): Color
- onSymbolEncoded(symbol: char, encoding: String): void
- onSymbolDecoded(symbol: char, path: String): void
- onNodeAdded(node: Node): void
- onNodeUpdated(node: Node): void
- onNodesSwapped(node1: Node, node2: Node): void
is used by
is used by
«uses»
«uses»
«implements»
«implements»
HuffmanTree
- root: Node
- currentNYT: Node
- symbolToNode: Map<Character, Node>
- nextNodeNumber: int
- debugMode: boolean
- listeners: List<TreeUpdateListener>
+ HuffmanTree()
+ HuffmanTree(debugMode: boolean)
+ hasSymbol(symbol: char): boolean
+ getEncodingForSymbol(symbol: char): String
+ getEncodingForNYT(): String
+ addSymbol(symbol: char): void
+ updateExistingSymbol(symbol: char): void
- addNewSymbol(symbol: char): void
- incrementNodeWeight(node: Node): void
- findBlockLeader(node: Node): Node
- swapNodes(node1: Node, node2: Node): void
- isAncestor(ancestor: Node, descendant: Node): boolean
+ getRoot(): Node
+ getNYTNode(): Node
+ addTreeUpdateListener(listener: TreeUpdateListener): void
+ removeTreeUpdateListener(listener: TreeUpdateListener): void
- notifyNodeAdded(node: Node): void
- notifyNodeUpdated(node: Node): void
- notifySwap(node1: Node, node2: Node): void
- printTree(): void
«interface»
DecodingListener
+ onSymbolDecoded(symbol: char, path: String): void
«interface»
EncodingListener
+ onSymbolEncoded(symbol: char, encoding: String): void
«implements»
is used by
Node
- weight: int
- nodeNumber: int
- symbol: char
- isNYT: boolean
- parent: Node
- leftChild: Node
- rightChild: Node
+ Node(nodeNumber: int, weight: int)
+ Node(nodeNumber: int, symbol: char, isNYT: boolean)
+ isLeaf(): boolean
+ isNYT(): boolean
+ getWeight(): int
+ getNodeNumber(): int
+ getSymbol(): char
+ getParent(): Node
+ getLeftChild(): Node
+ getRightChild(): Node
+ setParent(parent: Node): void
+ setLeftChild(node: Node): void
+ setRightChild(node: Node): void
+ incrementWeight(): void
«uses»
«interface»
TreeUpdateListener
+ onNodeAdded(node: Node): void
+ onNodeUpdated(node: Node): void
+ onNodesSwapped(node1: Node, node2: Node): void

Figure 1: UML class diagram of the implementation

2.2 Key Components of the Implementation

2.2.1 Tree Structure

The Huffman tree is implemented as a binary tree where:

- Each internal node has exactly two children

- Leaf nodes represent symbols from the input alphabet
- The NYT node is a special leaf node for symbols not yet seen
- Each node has a weight representing its frequency

2.2.2 Node Numbering

The implementation uses a numbering scheme to identify nodes:

- Node numbers decrease from the root to the leaves
- The root starts with the highest number (511 in this implementation)
- When nodes are swapped, they retain their original numbers

Listing 1: Node Class Excerpt

```java
public class Node {
    private int weight;
    private int nodeNumber;
    private char symbol;
    private boolean isNYT;
    private Node parent;
    private Node leftChild;
    private Node rightChild;

    // Constructor for internal nodes
    public Node(int nodeNumber, int weight) {
        this.nodeNumber = nodeNumber;
        this.weight = weight;
        this.isNYT = false;
    }

    // Constructor for NYT and leaf nodes
    public Node(int nodeNumber, char symbol, boolean
       isNYT) {
        this.nodeNumber = nodeNumber;
        this.symbol = symbol;
        this.isNYT = isNYT;
        this.weight = isNYT ? 0 : 1;
    }

    // Methods to check node type
    public boolean isLeaf() {
        return leftChild == null && rightChild == null &&
            !isNYT;
    }

    public boolean isNYT() {
        return isNYT;
    }

    // Getters and setters...
}
```

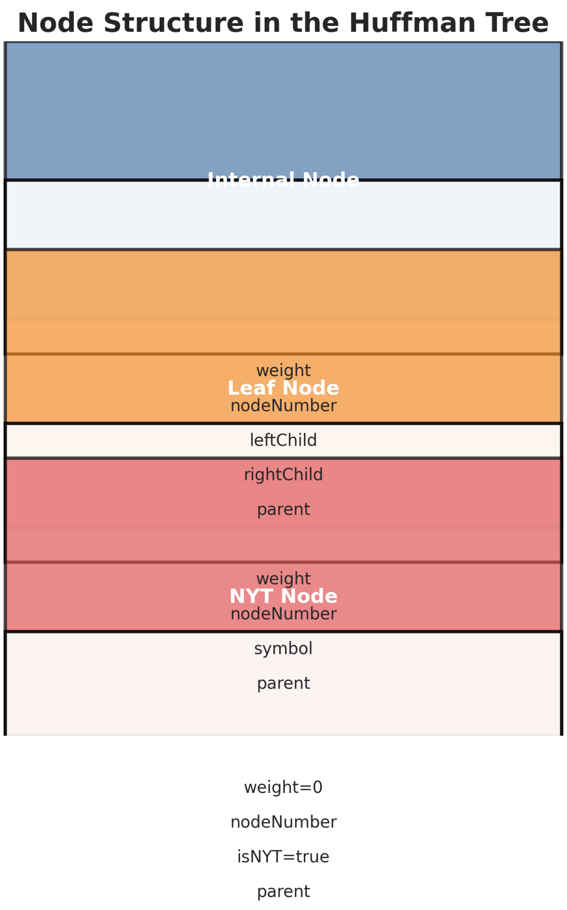


Figure 2: Node structure in the Huffman tree

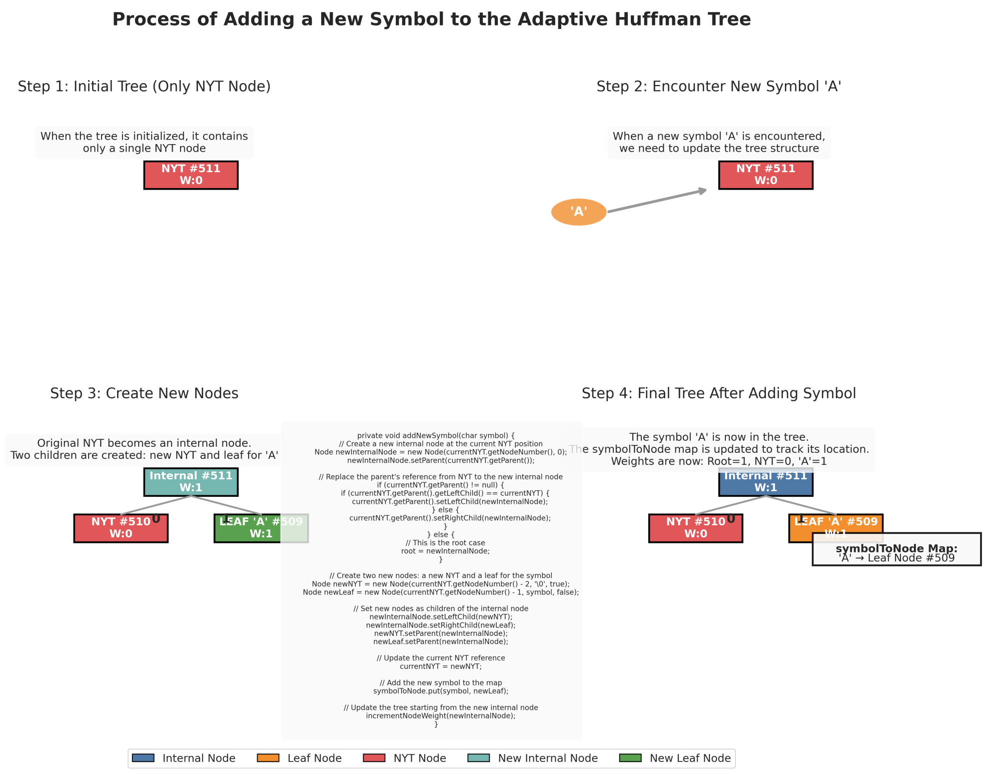
2.2.3 Symbol Addition and Tree Update

When a new symbol is encountered:

Listing 2: Adding New Symbol

```
private void addNewSymbol(char symbol) {
    // Create a new internal node at the current NYT
       position
    Node newInternalNode = new Node(currentNYT.
       getNodeNumber(), 0);
```

```java
    newInternalNode.setParent(currentNYT.getParent());

    // Replace the parent's reference from NYT to the new internal node
    if (currentNYT.getParent() != null) {
        if (currentNYT.getParent().getLeftChild() == currentNYT) {
            currentNYT.getParent().setLeftChild(newInternalNode);
        } else {
            currentNYT.getParent().setRightChild(newInternalNode);
        }
    } else {
        // This is the root case
        root = newInternalNode;
    }

    // Create two new nodes: a new NYT and a leaf for the symbol
    Node newNYT = new Node(currentNYT.getNodeNumber() - 2, '\0', true);
    Node newLeaf = new Node(currentNYT.getNodeNumber() - 1, symbol, false);

    // Set new nodes as children of the internal node
    newInternalNode.setLeftChild(newNYT);
    newInternalNode.setRightChild(newLeaf);
    newNYT.setParent(newInternalNode);
    newLeaf.setParent(newInternalNode);

    // Update the current NYT reference
    currentNYT = newNYT;

    // Add the new symbol to the map
    symbolToNode.put(symbol, newLeaf);

    // Update the tree starting from the new internal node
    incrementNodeWeight(newInternalNode);
}
```


Process of Adding a New Symbol to the Adaptive Huffman Tree
Step 1: Initial Tree (Only NYT Node)
When the tree is initialized, it contains only a single NYT node
NYT #511 W:0
Step 2: Encounter New Symbol 'A'
When a new symbol 'A' is encountered, we need to update the tree structure
NYT #511 W:0
'A'
Step 3: Create New Nodes
Original NYT becomes an internal node. Two children are created: new NYT and leaf for 'A'
Internal #511 W:1
NYT #510 W:0
0
LEAF 'A' #509 W:1
Step 4: Final Tree After Adding Symbol
The symbol 'A' is now in the tree. The symbolToNode map is updated to track its location. Weights are now: Root=1, NYT=0, 'A'=1
Internal #511 W:1
NYT #510 W:0
0
LEAF 'A' #509 W:1
symbolToNode Map:
'A' → Leaf Node #509
🟦 Internal Node
🟧 Leaf Node
🟥 NYT Node
🟦 New Internal Node
🟩 New Leaf Node

```
private void addNewSymbol(char symbol) {
// Create a new internal node at the current NYT position
Node newInternalNode = new Node(currentNYT.getNodeNumber(), 0);
newInternalNode.setParent(currentNYT.getParent());

// Replace the parent's reference from NYT to the new internal node
if (currentNYT.getParent() != null) {
if (currentNYT.getParent().getLeftChild() == currentNYT) {
currentNYT.getParent().setLeftChild(newInternalNode);
} else {
currentNYT.getParent().setRightChild(newInternalNode);
}
} else {
// This is the root case
root = newInternalNode;
}

// Create two new nodes: a new NYT and a leaf for the symbol
Node newNYT = new Node(currentNYT.getNodeNumber() - 2, '\0', true);
Node newLeaf = new Node(currentNYT.getNodeNumber() - 1, symbol, false);

// Set new nodes as children of the internal node
newInternalNode.setLeftChild(newNYT);
newInternalNode.setRightChild(newLeaf);
newNYT.setParent(newInternalNode);
newLeaf.setParent(newInternalNode);

// Update the current NYT reference
currentNYT = newNYT;

// Add the new symbol to the map
symbolToNode.put(symbol, newLeaf);

// Update the tree starting from the new internal node
incrementNodeWeight(newInternalNode);
}
```

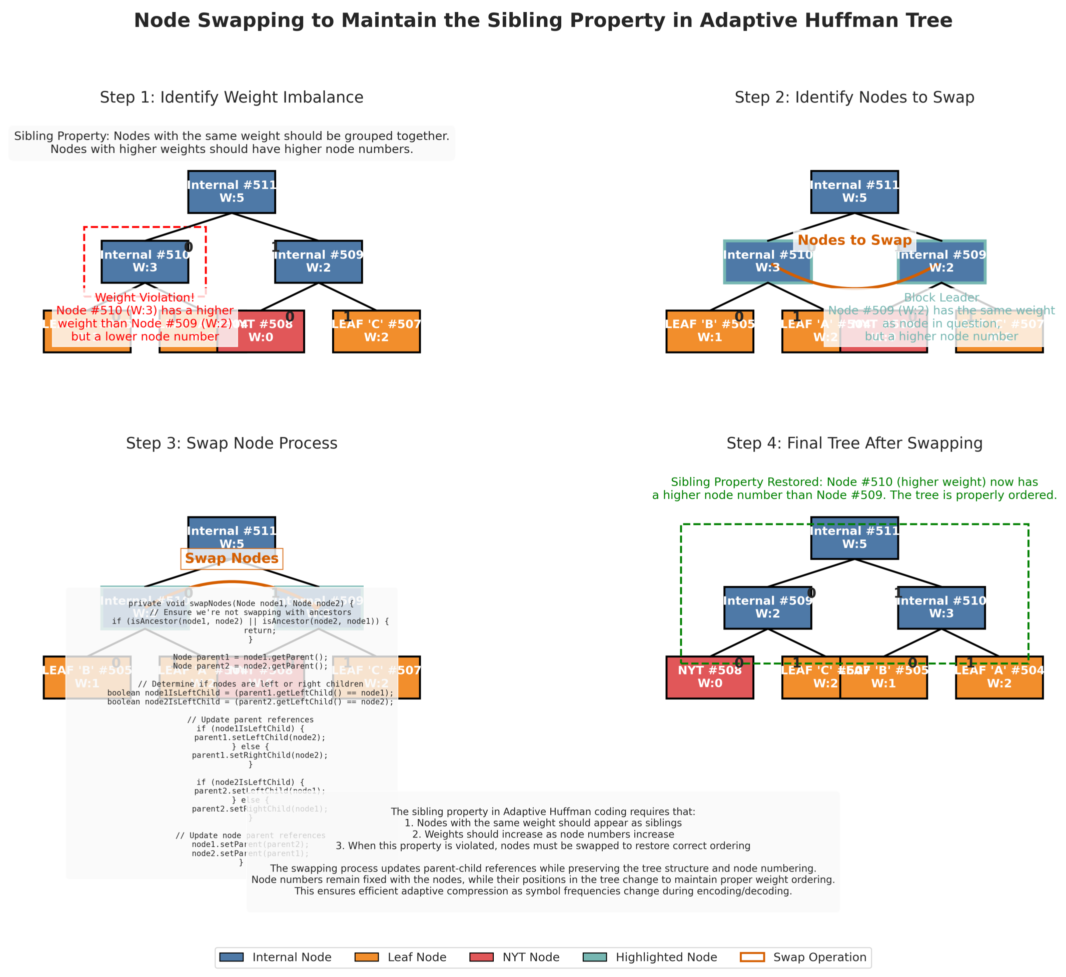
Figure 3: Process of adding a new symbol to the tree

2.2.4 Node Swapping

A crucial part of the adaptive algorithm is swapping nodes to maintain the sibling property:

Listing 3: Node Swapping

```java
private void swapNodes(Node node1, Node node2) {
    // Ensure we're not swapping with ancestors
    if (isAncestor(node1, node2) || isAncestor(node2,
       node1)) {
         return;
    }

    Node parent1 = node1.getParent();
    Node parent2 = node2.getParent();

    // Safety check
    if (parent1 == null || parent2 == null) {
         return;
    }

    // Determine if nodes are left or right children
    boolean node1IsLeftChild = (parent1.getLeftChild() ==
        node1);
    boolean node2IsLeftChild = (parent2.getLeftChild() ==
        node2);

    // Update parent references
    if (node1IsLeftChild) {
        parent1.setLeftChild(node2);
    } else {
        parent1.setRightChild(node2);
    }

    if (node2IsLeftChild) {
        parent2.setLeftChild(node1);
    } else {
        parent2.setRightChild(node1);
    }

    // Update node parent references
    node1.setParent(parent2);
    node2.setParent(parent1);

    // Notify listeners
    notifySwap(node1, node2);
}
```


Node Swapping to Maintain the Sibling Property in Adaptive Huffman Tree
Step 1: Identify Weight Imbalance
Sibling Property: Nodes with the same weight should be grouped together.
Nodes with higher weights should have higher node numbers.
Internal #511 W:5
Internal #510 W:3
Internal #509 W:2
Weight Violation!
Node #510 (W:3) has a higher weight than Node #509 (W:2) but a lower node number
NYT #508 W:0
LEAF 'C' #507 W:2
Step 2: Identify Nodes to Swap
Nodes to Swap
Block Leader
Node #509 (W:2) has the same weight as node in question, but a higher node number
LEAF 'B' #505 W:1
LEAF 'A' #506 W:2
Step 3: Swap Node Process
Swap Nodes
Step 4: Final Tree After Swapping
Sibling Property Restored: Node #510 (higher weight) now has a higher node number than Node #509. The tree is properly ordered.
LEAF 'A' #504 W:2
0
1
🟦 Internal Node
🟧 Leaf Node
🟥 NYT Node
🟩 Highlighted Node
□ Swap Operation

```
private void swapNodes(Node node1, Node node2) {
// Ensure we're not swapping with ancestors
if (isAncestor(node1, node2) || isAncestor(node2, node1)) {
return;
}

Node parent1 = node1.getParent();
Node parent2 = node2.getParent();

// Determine if nodes are left or right children
boolean node1IsLeftChild = (parent1.getLeftChild() == node1);
boolean node2IsLeftChild = (parent2.getLeftChild() == node2);

// Update parent references
if (node1IsLeftChild) {
parent1.setLeftChild(node2);
} else {
parent1.setRightChild(node2);
}

if (node2IsLeftChild) {
parent2.setLeftChild(node1);
} else {
parent2.setRightChild(node1);
}

// Update node parent references
node1.setParent(parent2);
node2.setParent(parent1);
}
```

The sibling property in Adaptive Huffman coding requires that:
1. Nodes with the same weight should appear as siblings
2. Weights should increase as node numbers increase
3. When this property is violated, nodes must be swapped to restore correct ordering

The swapping process updates parent-child references while preserving the tree structure and node numbering.
Node numbers remain fixed with the nodes, while their positions in the tree change to maintain proper weight ordering.
This ensures efficient adaptive compression as symbol frequencies change during encoding/decoding.

Figure 4: Node swapping to maintain the sibling property

2.2.5 Weight Incrementation and Tree Update

After processing a symbol, node weights are updated and the tree is rebalanced:

Listing 4: Weight Incrementation

```
private void incrementNodeWeight(Node node) {
    while (node != null) {
        // Find the highest-numbered node of the same
           weight
        Node blockLeader = findBlockLeader(node);

        // If we need to swap
        if (blockLeader != null && node != blockLeader
            && node.getParent() != blockLeader
            && blockLeader.getParent() != node) {

            swapNodes(node, blockLeader);

            // After swap, continue updating with the
               node in its new position
            node = blockLeader;
        }

        // Increment the weight
        node.incrementWeight();

        // Move up to parent
        node = node.getParent();
    }
}
```

2.3 Encoding Process

The encoding process follows these steps:

1. Start with a tree containing only the NYT node
2. For each symbol in the input:
    (a) If the symbol has been seen before, output the path to its leaf node
    (b) If the symbol is new, output the path to the NYT node followed by the symbol's ASCII code
    (c) Update the tree (add new nodes or increment weights)
    (d) Rebalance the tree if necessary

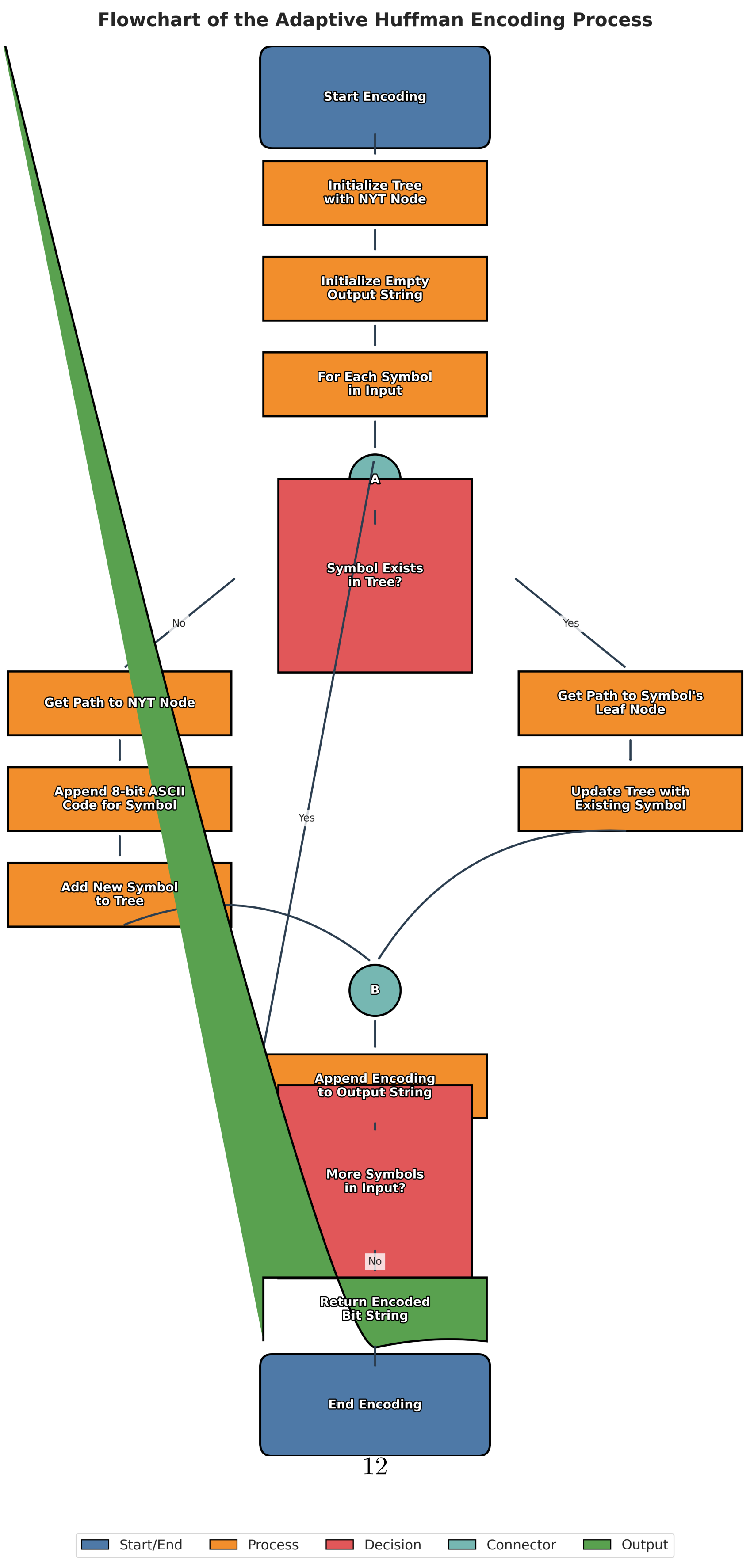

Flowchart of the Adaptive Huffman Encoding Process
Start Encoding
Initialize Tree with NYT Node
Initialize Empty Output String
For Each Symbol in Input
A
Symbol Exists in Tree?
No
Yes
Get Path to NYT Node
Append 8-bit ASCII Code for Symbol
Add New Symbol to Tree
Get Path to Symbol's Leaf Node
Update Tree with Existing Symbol
Yes
B
Append Encoding to Output String
More Symbols in Input?
No
Return Encoded Bit String
End Encoding
Start/End
Process
Decision
Connector
Output

Figure 5: Flowchart of the encoding process

2.4 Decoding Process

The decoding process is the reverse:

1. Start with the same initial tree as the encoder
2. Read bits from the input:
   (a) Follow the path in the tree according to the bits (0 = left, 1 = right)
   (b) If a leaf node is reached, output its symbol
   (c) If the NYT node is reached, read the next 8 bits to get the symbol
   (d) Update the tree identical to the encoder

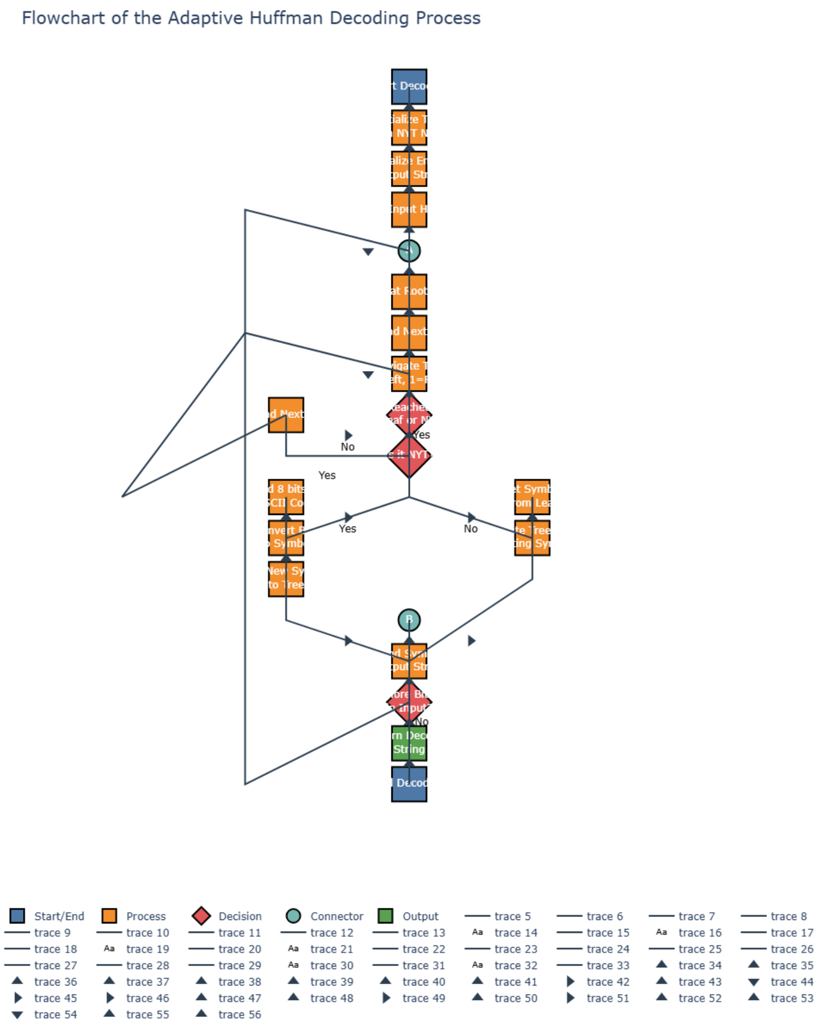


Figure 6: Flowchart of the decoding process

3 Test Case Analysis

3.1 Test Case 1: Simple Repetitive String

3.1.1 Input

`AABCBAACB`

3.1.2 Encoding Process

Symbol	**Binary Encoding**	**Explanation**	**Total Bits**
A	01000001	NYT + ASCII(A)	8
A	1	Path to A's leaf node	1
B	001000010	NYT + ASCII(B)	9
C	0001000011	NYT + ASCII(C)	10
B	10	Path to B's leaf node	2
A	0	Path to A's leaf node	1
A	0	Path to A's leaf node	1
C	111	Path to C's leaf node	3
B	10	Path to B's leaf node	2
Total			37

Table 1: Encoding of "AABCBAACB"

3.1.3 Results

- Original length: 72 bits (9 characters $\times$ 8 bits)
- Encoded length: 37 bits
- Compression ratio: 1.946
- Compression percentage: 48.61%

3.1.4 Analysis

This test case demonstrates good compression on a short string with repetitive characters. The adaptive algorithm quickly adjusts to the repeated symbols, assigning them shorter codes. The first occurrence of each character is expensive (8+ bits), but subsequent occurrences are encoded with only 1-3 bits.

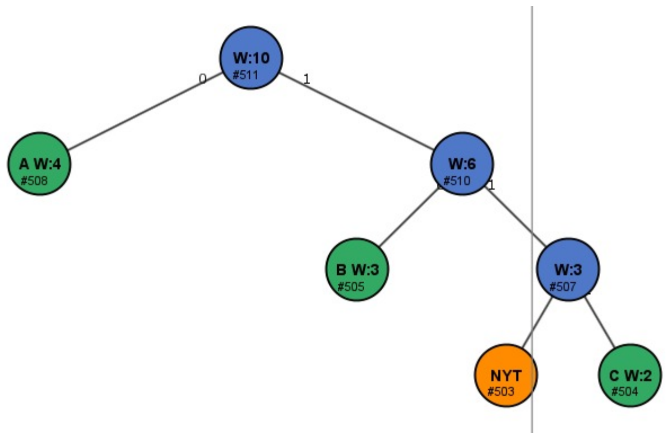


Figure 7: Final tree state after encoding "AABCBAACB"

3.2 Test Case 2: "Hello World"

3.2.1 Input

```
Hello World
```

3.2.2 Encoding Process

Symbol	Binary Encoding	Explanation	Total Bits
H	01001000	NYT + ASCII(H)	8
e	01100101	NYT + ASCII(e)	8
l	01101100	NYT + ASCII(l)	8
l	01	Path to l's leaf node	2
o	0001101111	NYT + ASCII(o)	10
space	10000100000	NYT + ASCII(' ')	11
W	01001010111	NYT + ASCII(W)	11
o	101	Path to o's leaf node	3
r	00001110010	NYT + ASCII(r)	11
l	10	Path to l's leaf node	2
d	110001100100	NYT + ASCII(d)	15
Total			89

Table 2: Encoding of "Hello World"

3.2.3 Results

- Original length: 88 bits (11 characters × 8 bits)
- Encoded length: 89 bits
- Compression ratio: 0.989

- Compression percentage: -1.14%

3.2.4 Analysis

This test case shows that for short texts with little repetition, adaptive Huffman coding may actually increase the size. With only 11 characters and 7 unique symbols, the overhead of encoding the first occurrence of each character outweighs the benefits of compression for the repeated characters. This test case illustrates an important limitation of adaptive Huffman coding for short inputs.

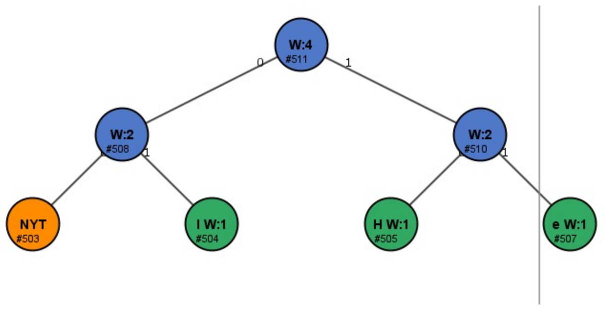


Figure 8: Tree evolution during "Hello World" encoding

3.3 Test Case 3: Longer Text with Mixed Content

3.3.1 Input

```
This is a longer test case for Adaptive Huffman coding.  It includes
various characters, punctuation, and repeated patterns to demonstrate
the effectiveness of the algorithm.
```

3.3.2 Results

- Original length: 1384 bits (173 characters $\times$ 8 bits)
- Encoded length: 944 bits
- Compression ratio: 1.466
- Compression percentage: 31.79%

3.3.3 Analysis

This longer test case demonstrates that adaptive Huffman coding becomes more effective as the input size increases. With a reasonable amount of repetition (common letters like 'e', 't', 'a', and spaces), the algorithm achieves significant compression of about 32%. The tree dynamically adapts to the frequency patterns in the text, assigning shorter codes to the most common characters.

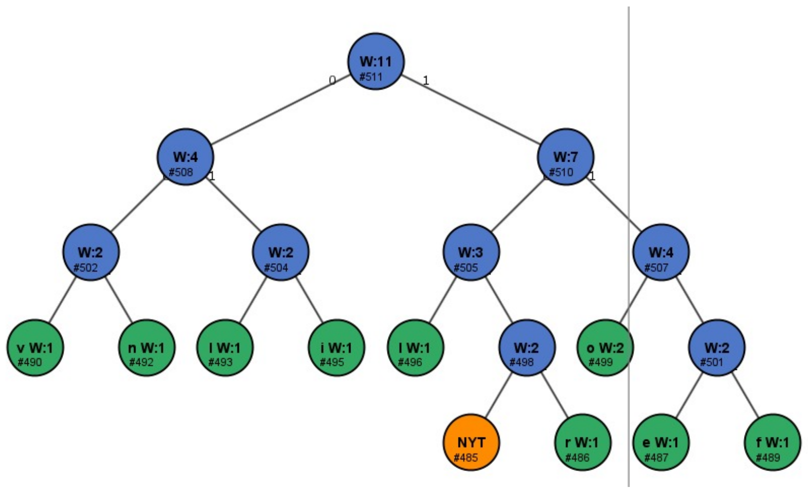


Figure 9: Partial tree state during encoding of the longer text

4 Performance Analysis

4.1 Compression Efficiency

The implementation shows varying compression efficiency:

- Short strings with high repetition: Good compression (Test Case 1: 48.61%)
- Short strings with low repetition: Slight expansion (Test Case 2: -1.14%)
- Longer texts with natural language patterns: Moderate compression (Test Case 3: 31.79%)

These results align with theoretical expectations for adaptive Huffman coding.

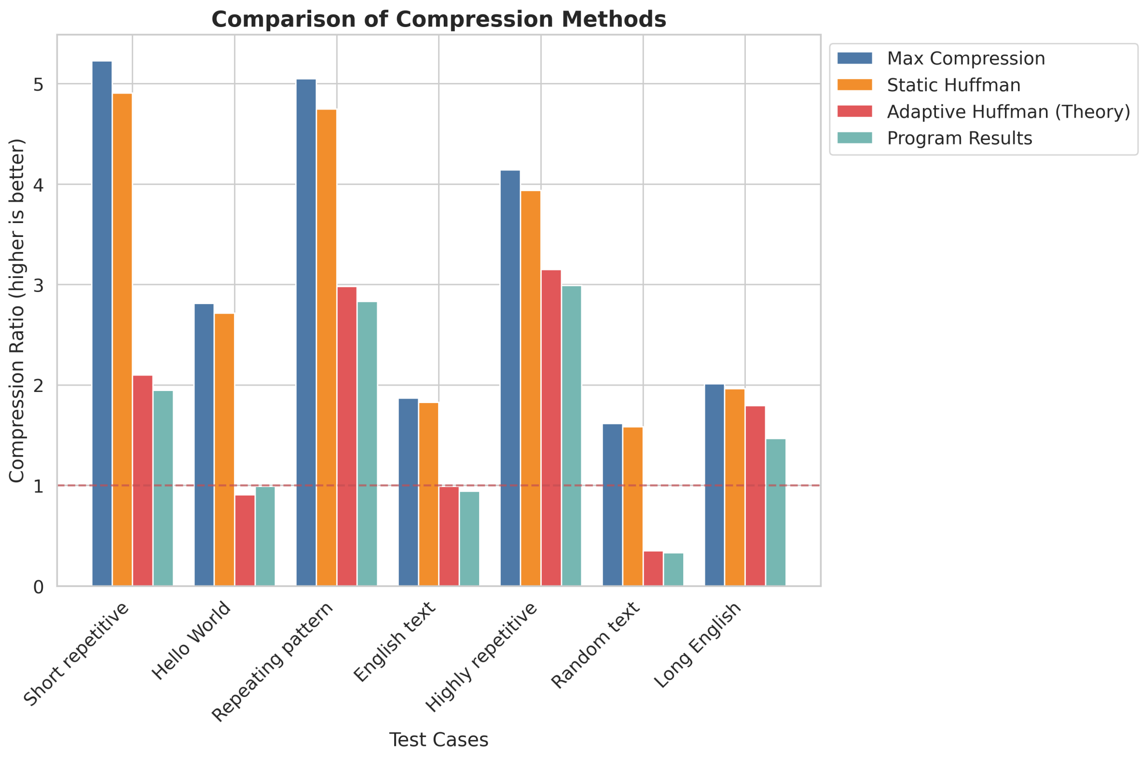


Figure 10: Compression efficiency comparison across test cases

4.2 Time Complexity

The implementation has the following time complexity characteristics:

- Encoding/decoding a single symbol: O(log n) in the average case, where n is the number of unique symbols
- Finding the block leader: O(n) in the worst case
- Tree update operations: O(log n) on average
- Overall encoding/decoding: O(m log n), where m is the input length

4.3 Space Complexity

- Tree structure: O(n) space, where n is the number of unique symbols
- Symbol-to-node mapping: O(n) space
- Overall: O(n) space complexity

5 Implementation Highlights

5.1 Debug Mode

The implementation includes a debug mode that outputs detailed information about tree operations:

- Tree state before and after processing each symbol
- Node swapping operations
- Weight updates
- Paths generated during encoding

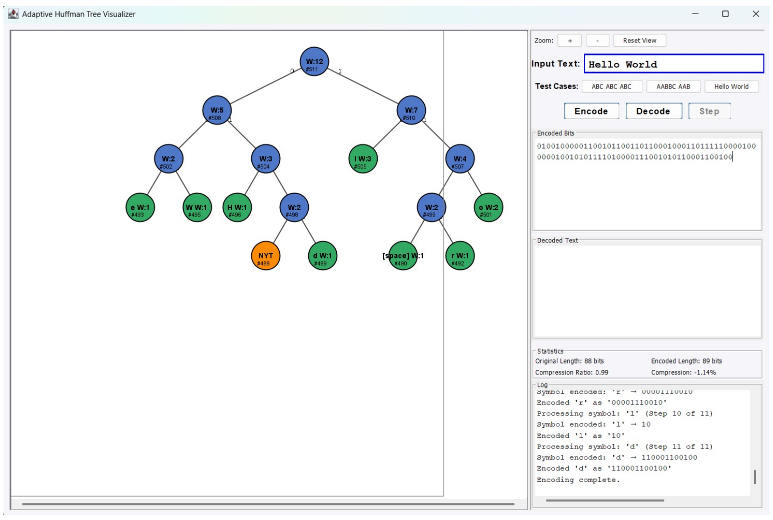
Listing 5: Debug Output Example

```
DEBUG [Tree] >>> Processing Symbol: 'A' <<< (ASCII: 65)
Current Tree State BEFORE processing:
-------------------- Tree Structure --------------------
+- [INTERNAL #512 W:0] [ROOT] [NYT]
--------------------------------------------------------
DEBUG [Tree] Symbol 'A' is NEW. Adding to tree.
DEBUG [Tree] Created new nodes for symbol 'A'
DEBUG [Tree] <<< Finished Processing Symbol: 'A' >>>
Final Tree State:
-------------------- Tree Structure --------------------
+- [INTERNAL #511 W:1] [ROOT]
   +- [NYT #510 W:0] [Current NYT]
   +- [LEAF:'A' #509 W:1]
--------------------------------------------------------
```

5.2 Visualization Component

The implementation includes a visualization component that renders the Huffman tree graphically:

- Interactive display of the tree structure
- Color-coded nodes (NYT, leaf, internal)
- Step-by-step animation of the encoding/decoding process
- Zoom and pan functionality for navigating large trees



Input Text: Hello World

Encoded Bits

```
0100100000110010110011011000100011011111100000100
00001001010111110100001110010101100011001 00
```

Statistics

```
Original Length: 88 bits          Encoded Length: 89 bits
Compression Ratio: 0.99           Compression: -1.14%
```

Log

```
Symbol encoded: 'r' → 00001110010
Encoded 'r' as '00001110010'
Processing symbol: 'l' (Step 10 of 11)
Symbol encoded: 'l' → 10
Encoded 'l' as '10'
Processing symbol: 'd' (Step 11 of 11)
Symbol encoded: 'd' → 110001100100
Encoded 'd' as '110001100100'
Encoding complete.
```

Figure 11: Huffman Tree Visualizer interface

6 Conclusion

6.1 Summary of Findings

The adaptive Huffman coding implementation based on the FGK algorithm demonstrates the expected behavior of this compression technique:

- Effective compression for repetitive content
- Better performance on longer texts
- Single-pass operation without requiring prior knowledge of symbol frequencies

The test cases illustrate both the strengths and limitations of the algorithm, showing that it's most suitable for compressing longer texts with repeating patterns.

6.2 Advantages of This Implementation

- Clean, modular code structure
- Comprehensive debugging and visualization tools
- Strict adherence to the FGK algorithm
- Well-documented code with detailed comments

6.3 Potential Improvements

- Optimization of the block leader search to improve performance
- Implementation of bit-level I/O for practical file compression
- Additional heuristics to avoid expansion on short inputs
- Parallel processing for large inputs

7 References

1. Knuth, D. E. (1985). "Dynamic Huffman coding". Journal of Algorithms, 6(2), 163–180.

2. Vitter, J. S. (1987). "Design and analysis of dynamic Huffman codes". Journal of the ACM, 34(4), 825–845.

3. Gallager, R. G. (1978). "Variations on a theme by Huffman". IEEE Transactions on Information Theory, 24(6), 668–674.

4. Faller, N. (1973). "An adaptive system for data compression". Record of the 7th Asilomar Conference on Circuits, Systems, and Computers, 593–597.

8 Appendix: Key Code Snippets

8.1 Symbol Processing

```java
public String encodeSymbol(char symbol) {
    StringBuilder encoded = new StringBuilder();

    // Check if the symbol already exists in the tree
    if (tree.hasSymbol(symbol)) {
        // Get the encoding path for the existing symbol
        encoded.append(tree.getEncodingForSymbol(symbol))
            ;

        // Update the tree with the existing symbol
        tree.updateExistingSymbol(symbol);
    } else {
        // Get the path to the NYT node for new symbols
        encoded.append(tree.getEncodingForNYT());

        // Add the ASCII code for the new symbol
        String asciiCode = toBinaryString(symbol, 8);
        encoded.append(asciiCode);

```

```java
            // Add the symbol to the tree
            tree.addSymbol(symbol);
        }

        // Notify listeners
        for (EncodingListener listener : listeners) {
            listener.onSymbolEncoded(symbol, encoded.toString
                ());
        }

        return encoded.toString();
    }
```

8.2 Block Leader Search

```java
private Node findBlockLeader(Node node) {
    if (node == null) return null;

    int targetWeight = node.getWeight();
    Node leader = null;

    // Use a node queue for breadth-first search
    Queue<Node> nodeQueue = new LinkedList<>();
    nodeQueue.add(root);

    while (!nodeQueue.isEmpty()) {
        Node current = nodeQueue.poll();

        // If we find a node of the target weight
        if (current.getWeight() == targetWeight) {
            // Check if this is a candidate for leader
            if (!isDescendant(current, node) &&
                (leader == null || current.getNodeNumber
                    () > leader.getNodeNumber())) {
                leader = current;
            }
        }

        // Add children to the queue
        if (current.getLeftChild() != null) {
            nodeQueue.add(current.getLeftChild());
        }
        if (current.getRightChild() != null) {
            nodeQueue.add(current.getRightChild());
        }
    }

    return leader;
}
```

8.3 Tree State Visualization

```java
private void printTree() {
    if (!debugMode) return;

    System.out.println("-------------------- Tree Structure --------------------");
    printNode(root, "", true);
    System.out.println("--------------------------------------------------------");
}

private void printNode(Node node, String prefix, boolean isTail) {
    if (node == null) return;

    System.out.print(prefix);
    System.out.print(isTail ? "+- " : "|  ");

    // Print node details
    System.out.print("[");
    if (node.isLeaf()) {
        System.out.print("LEAF:'" + displayableChar(node.getSymbol()) + "'");
    } else if (node.isNYT()) {
        System.out.print("NYT");
    } else {
        System.out.print("INTERNAL");
    }

    System.out.print(" #" + node.getNodeNumber() + " W:" + node.getWeight() + "]");

    if (node == root) System.out.print(" [ROOT]");
    if (node.isNYT() && node == currentNYT) System.out.print(" [Current NYT]");

    System.out.println();

    // Print children
    if (node.getRightChild() != null || node.getLeftChild() != null) {
        printNode(node.getLeftChild(), prefix + (isTail ? "   " : "|  "),
                  node.getRightChild() == null);
        printNode(node.getRightChild(), prefix + (isTail
            ? "    " : "|   "), true);
    }
}
```