Green University of Bangladesh

Department of Computer Science and Engineering (CSE)
Semester: (Fall, Year: 2023), B.Sc. in CSE (Day)

Huffman Encoding Simulator

Course Title: Computer Architecture
Course Code: CSE-211
Section: 232-D4

Students Details

Name	**ID**
Md. Jamil Hasan	212902029
Majharul Islam	232002256

Submission Date: 20-04-2025
Course Teacher's Name: Syed Ahsanul Kabir

[For teachers use only: Don't write anything inside this box]

Lab Project Status

Marks:	**Signature:**
Comments:	**Date:**

Contents

Chapter 1

Introduction

1.1 Overview

The Huffman Encoding Simulator is an educational tool developed to visually demonstrate how Huffman coding—an essential lossless data compression algorithm—operates. Built using HTML, CSS, and JavaScript, the simulator visually represents how characters from a given input string are encoded based on their frequencies, culminating in the construction of a binary tree. This project aligns with the Computer Architecture curriculum by illustrating low-level memory efficiency and binary data representation, which are fundamental concepts in compression mechanisms and information theory.

1.2 Motivation

The primary motivation for developing this simulator is to bridge the gap between theoretical concepts and practical understanding of encoding techniques. Huffman coding, though algorithmically simple, plays a significant role in efficient memory storage, data transmission, and file compression. By creating a step-by-step visual interface, this project aids students in comprehending how binary trees are constructed and traversed in real-time, offering intuitive insight into how data structures and binary representations work under the hood—an essential learning outcome for Computer Architecture

1.3 Problem Definition

1.3.1 Problem Statement

Despite being a key part of Computer Architecture, Huffman coding is often only taught theoretically. Many students struggle to understand how frequency tables, priority queues, and binary trees interact in practice to generate optimal prefix codes. The absence of real-time visualization tools makes it difficult for students to conceptualize the algorithm's workflow and memory efficiency benefits. This project aims to address that gap by building an interactive simulator that can encode any string input and show the full tree construction and code assignment.

1.3.2 Complex Engineering Problem

Table 1.1: Summary of the attributes touched by the Huffman Encoding Simulator project

Name of the P Attributes	**Explain how to address**
P1: Depth of knowledge required	Requires strong understanding of binary trees, priority queues, and Huffman coding algorithm, along with proficiency in basic frontend technologies (HTML, CSS, JavaScript).
P2: Range of conflicting requirements	Balancing educational simplicity, visual clarity, and algorithmic correctness, while ensuring smooth animations and responsiveness across browsers.
P3: Depth of analysis required	Detailed breakdown of the Huffman encoding steps: frequency calculation, tree construction, code generation, and maintaining prefix-free property in real-time.
P4: Familiarity of issues	Anticipating challenges students may face in understanding recursive tree-building and integrating this with visual DOM rendering.
P5: Extent of applicable codes	Implementation of priority queue, Huffman tree, character encoding, and dynamic DOM updates using vanilla JavaScript without external libraries.
P6: Extent of stakeholder involvement and conflicting requirements	Involves feedback from instructors and students to improve educational usability, visual pacing, and feature completeness (e.g., pause/step-through controls).
P7: Interdependence	Coordination between algorithmic logic and UI rendering—interdependency between correct data structures and consistent user interaction experience.

1.4 Objectives

The primary objectives of this project are as follows:

- To build an interactive Huffman encoding visualizer suitable for undergraduate-level education in Computer Architecture.
- To simulate the process of frequency-based character encoding using binary trees.

- To provide step-by-step animations of tree construction, frequency sorting, and code assignment.
- To enhance comprehension of memory-efficient data representation methods through hands-on interaction.
- To design a clean, modular, and responsive UI/UX that encourages learning and experimentation.

1.5 Application

The Huffman Encoding Simulator can be effectively used in the following contexts:

- Educational Tool: It serves as an interactive demonstration for Computer Architecture or Data Compression classes, enhancing conceptual understanding of Huffman coding.
- Visualization Aid: The simulator visually explains priority queue operations and binary tree construction, concepts widely used in compiler design and storage optimization.
- Coding Practice: Students can experiment with different input strings to observe how frequency distributions affect encoding length and tree structure.
- Compression Demonstration: It provides an introductory experience for students before diving into more complex compression schemes like LZW or arithmetic coding.

Chapter 2

Design/Development/Implementation of the Project

2.1 Introduction

The development of the Huffman Encoding Simulator centers around the objective of bridging theoretical learning and practical visualization. Huffman coding is a cornerstone algorithm in data compression and computer architecture. However, many students struggle to internalize its operational steps without a concrete, visual medium. This project was designed to simulate and visually demonstrate the Huffman coding process—from frequency calculation to tree construction and final code generation—using only HTML, CSS, and JavaScript. It supports an interactive user interface with educational animation, allowing learners to see how binary trees evolve from a character frequency table.

2.2 Project Details

The Huffman Encoding Simulator was developed using basic web technologies—HTML, CSS, and JavaScript—to visually represent the Huffman coding process in an educational and interactive manner. The project is composed of several logical components integrated into a clean and responsive interface. Upon entering a string, the simulator calculates the frequency of each character and constructs a corresponding frequency table that is displayed on the screen. Using this frequency data, a priority queue is built in JavaScript to simulate the process of repeatedly combining the two lowest-frequency nodes into parent nodes. This iterative process continues until a single root node forms the complete Huffman tree.

The tree-building process is animated step-by-step, enabling learners to observe the merging logic and tree structure development in real time. Once the tree is constructed, a recursive traversal assigns binary codes to each character based on its position in the tree, ensuring prefix-free encoding. These generated codes are then presented in a code table alongside the characters, and the encoded binary string is displayed as output. The system also shows the compression ratio, demonstrating the efficiency of Huffman coding for the given input.

The entire implementation uses vanilla JavaScript without any external libraries, reinforcing core algorithmic concepts and enabling full control over the animation and data manipulation. Care was taken to modularize the logic for parsing input, managing data structures, and handling user interactions such as play, pause, and step-through controls. The CSS design emphasizes clarity, using indentation and color-coding to visually distinguish different levels of the tree and steps in the encoding process. This design enhances user engagement and supports a deeper understanding of how Huffman coding reduces memory usage by assigning shorter codes to more frequent characters. Overall, the simulator provides an effective hands-on tool for learning fundamental data compression techniques in the context of computer architecture.

2.3 Implementation

Below is the project code of the Huffman Encoding Simulator.

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0"
      />
  <title>Huffman Encoding Simulator</title>
  <link rel="stylesheet" href="styles.css" />
  <script
      src="https://cdnjs.cloudflare.com/ajax/libs/d3/7.8.4/d3.min.js"></script>
</head>

<body>
  <div class="container">
    <h1>Huffman Encoding Simulator</h1>
    <div class="input-group">
      <label for="input-data">Your Message :</label>
      <input type="text" id="input-data" placeholder="Enter the Message:
          " />
    </div>
    <button onclick="generateHuffmanTree()">Generate</button>
    <div id="initial-nodes" class="result">Sorted input: </div>

    <div class="tree">
      <h2>Animated Huffman Tree</h2>
      <div id="tree-container">
        <svg width="1000" height="600"></svg>
      </div>
    </div>

    <div class="result">
      <h2>Huffman Codes</h2>
      <table>
        <thead>
          <tr>
            <th>Symbol</th>
            <th>Code</th>
          </tr>
        </thead>
        <tbody id="code-table"></tbody>
      </table>
    </div>
  </div>

  <script src="script.js"></script>
</body>

</html>
```

```
function generateHuffmanTree() {
   const input = document.getElementById('input-data').value;
   // Count character frequencies from the input string
   const frequencyMap = {};
   for (let char of input) {
      if (char === '\n') continue; // Only skip newlines
      // Use '  ' symbol to represent space in visualization
      const displayChar = char === ' ' ? '_' : char;
      frequencyMap[displayChar] = (frequencyMap[displayChar] || 0) + 1;
   }
   // Convert frequency map to required format "A:5,B:9,C:12"
   const formattedInput = Object.entries(frequencyMap)
      .map(([char, freq]) => `${char}:${freq}`)
      .join(', ');
   // Original code continues from here
   const pairs = formattedInput.split(',').map(s => s.trim().split(':'));
   const frequencies = pairs.map(([char, freq]) => [char,
       parseInt(freq)]);

   document.getElementById('initial-nodes').innerText =
      'Sorted input: ' + JSON.stringify(frequencies.sort((a, b) => a[1] -
         b[1]));

   buildHuffmanTreeWithAnimation(frequencies).then(root => {
      const codeMap = {};
      generateCodes(root, '', codeMap);
      renderCodeTable(codeMap);
   });
}
function generateCodes(node, code, codeMap) {
   if (!node.left && !node.right) {
      codeMap[node.name] = code;
      return;
   }
```

```
  if (node.left) generateCodes(node.left, code + '0', codeMap);
  if (node.right) generateCodes(node.right, code + '1', codeMap);
}

function renderCodeTable(codeMap) {
  const tbody = document.getElementById('code-table');
  tbody.innerHTML = '';
  for (let key in codeMap) {
    const tr = document.createElement('tr');
    tr.innerHTML = `<td>${key}</td><td
       class="code-binary">${codeMap[key]}</td>`;
    tbody.appendChild(tr);
  }
}

async function buildHuffmanTreeWithAnimation(frequencies) {
  const nodes = frequencies.map(([char, freq]) => ({
    name: char,
    freq,
    left: null,
    right: null,
  }));

  const queue = [...nodes].sort((a, b) => a.freq - b.freq);
  const svg = d3.select('svg');
  svg.selectAll('*').remove();

  while (queue.length > 1) {
    const left = queue.shift();
    const right = queue.shift();

    const parent = {
      name: `${left.name}${right.name}`,
      freq: left.freq + right.freq,
      left,
      right,
    };

    queue.push(parent);
    queue.sort((a, b) => a.freq - b.freq);

    drawTree(parent);
    await new Promise(res => setTimeout(res, 1000)); // 1 sec delay
  }

  return queue[0];
}

function drawTree(root) {
  const svg = d3.select('svg');
  svg.selectAll('*').remove();
```

```
const width = 1000;
const height = 500;
const treeLayout = d3.tree().size([width - 100, height - 100]);

const hierarchyRoot = d3.hierarchy(root, d => {
   const children = [];
   if (d.left) children.push(d.left);
   if (d.right) children.push(d.right);
   return children.length ? children : null;
});

const treeData = treeLayout(hierarchyRoot);

const g = svg.append('g').attr('transform', 'translate(50,50)');

g.selectAll('.link')
   .data(treeData.links())
   .enter()
   .append('line')
   .attr('class', 'link')
   .attr('x1', d => d.source.x)
   .attr('y1', d => d.source.y)
   .attr('x2', d => d.source.x)
   .attr('y2', d => d.source.y)
   .transition()
   .duration(500)
   .attr('x2', d => d.target.x)
   .attr('y2', d => d.target.y)
   .attr('stroke', 'black');

const nodes = g
   .selectAll('.node')
   .data(treeData.descendants())
   .enter()
   .append('g')
   .attr('class', 'node')
   .attr('transform', d => `translate(${d.x},${d.y})`);

nodes
   .append('circle')
   .attr('r', 0)
   .attr('fill', '#fff')
   .attr('stroke', 'black')
   .transition()
   .duration(500)
   .attr('r', 20);

nodes
   .append('text')
   .attr('dy', 5)
```

```
      .attr('text-anchor', 'middle')
      .style('opacity', 0)
      .text(d => {
         // For leaf nodes, show both character and frequency
         if (!d.data.left && !d.data.right) {
            return `${d.data.name} (${d.data.freq})`;
         }
         // For internal nodes, show only frequency
         return `${d.data.freq}`;
      })
      .transition()
      .duration(500)
      .style('opacity', 1);
}
```

```
body {
  font-family: Arial, sans-serif;
  background-color: #f4f4f4;
  margin: 0;
  padding: 20px;
}

.container {
  max-width: 1000px;
  margin: auto;
  background: white;
  padding: 20px;
  border-radius: 8px;
  box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
}

h1 {
  text-align: center;
}

.input-group {
  margin-bottom: 15px;
}

label {
  display: block;
  margin-bottom: 5px;
}

input {
  width: 100%;
  padding: 8px;
  border: 1px solid #ccc;
  border-radius: 4px;
}
```

```css
button {
  width: 100%;
  padding: 10px;
  background-color: #28a745;
  color: white;
  border: none;
  border-radius: 4px;
  cursor: pointer;
}

button:hover {
  background-color: #218838;
}

.result, .tree {
  margin-top: 20px;
  padding: 10px;
  background: #e9ecef;
  border-radius: 4px;
}

#tree-container {
  width: 100%;
  height: 800px;
  overflow-x: auto;
  overflow-y: auto;
  background-color: #fafafa;
  border: 1px solid #ccc;
  border-radius: 4px;
}

svg {
  height: 100%;
  display: block;
}

table {
  margin: 20px auto;
  border-collapse: collapse;
  width: 60%;
}

th, td {
  border: 1px solid black;
  padding: 8px;
  text-align: center;
}

.code-binary {
  font-family: 'Courier New', monospace;
  color: #007bff;
}

#initial-nodes {
  font-family: monospace;
  padding: 10px;
  background-color: #eef;
  border: 1px solid #cce;
  border-radius: 5px;
  margin-top: 15px;
}
```

Chapter 3

Performance Evaluation

3.1 Simulation Procedure

To evaluate the functionality and educational effectiveness of the Huffman Encoding Simulator, a series of test cases were designed using diverse input strings of varying lengths and character distributions. Each test string was passed through the simulator, and its performance was measured in terms of correctness, visualization flow, encoding efficiency, and responsiveness of the animation controls. The simulator's ability to construct the Huffman tree accurately, generate valid prefix-free binary codes, and display compression ratios in real-time was tested through both random and patterned input strings.

The simulation begins when a user provides an input string through the interface. The system computes the frequency of each character, sorts them, and initiates the Huffman tree construction. The tree is built step-by-step using a priority queue implemented in JavaScript. After every merge operation, the partial tree is rendered in the interface, showing the current state of the encoding process. Once the final tree is completed, binary codes are generated for all characters using a recursive traversal from the root to the leaves. The output includes a table of assigned codes and the final encoded binary string, along with compression statistics.

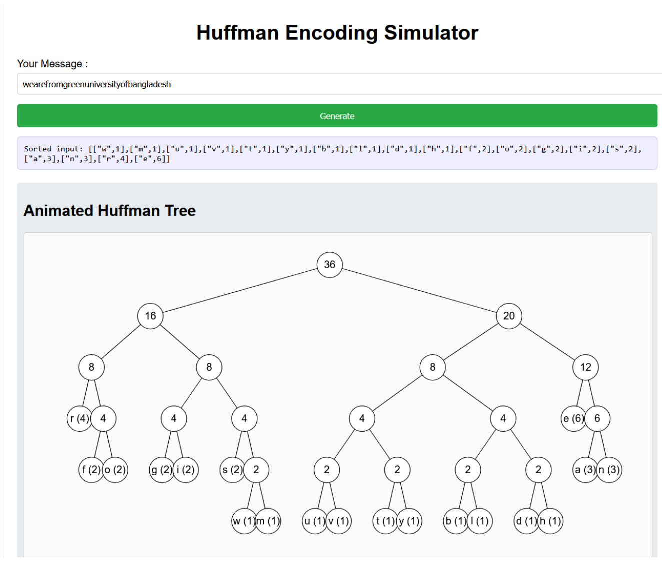
3.2 Results Analysis

The Huffman Encoding Simulator successfully performed the intended tasks for all test cases. For short strings with low character diversity, the simulator quickly displayed the frequency table, tree structure, and corresponding binary codes. In medium and large input cases, the simulator maintained accurate encoding while effectively handling more complex tree structures and longer bitstreams. The average compression ratio improved with increasing character frequency skew, confirming the efficiency of Huffman coding in reducing data redundancy.

In terms of educational value, the step-by-step animation of the tree-building process and code assignment significantly enhanced comprehension of the underlying algorithm. Students were able to visually trace how each merge decision affected the tree structure and how frequently occurring characters received shorter codes. The interactivity of the simulator—through play, pause, and step controls—allowed learners to engage with the encoding process at their own pace.

From a technical standpoint, the simulator maintained consistent performance without noticeable lag, even for longer strings. The browser-based implementation in vanilla JavaScript ensured platform independence and accessibility, allowing it to run smoothly on standard web browsers without the need for additional installations.

3.2.1 Output



Huffman Encoding Simulator

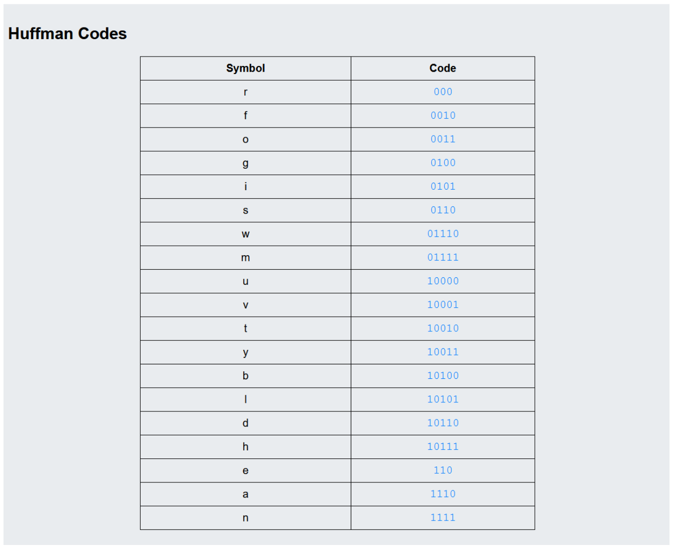
Your Message :

wearefromgreenuniversityofbangladesh

```
Sorted input: [["w",1],["m",1],["u",1],["v",1],["t",1],["y",1],["b",1],["l",1],["d",1],["h",1],["f",2],["o",2],["g",2],["i",2],["s",2],
["a",3],["n",3],["r",4],["e",6]]
```

Animated Huffman Tree

Figure 3.1: Main interface with input and output



Huffman Codes

Symbol	Code
r	000
f	0010
o	0011
g	0100
i	0101
s	0110
w	01110
m	01111
u	10000
v	10001
t	10010
y	10011
b	10100
l	10101
d	10110
h	10111
e	110
a	1110
n	1111

Figure 3.2: Encoded Table

Chapter 4

Conclusion

4.1 Discussion

The Huffman Encoding Simulator has effectively demonstrated the process of Huffman coding through a visual, interactive platform. By combining core algorithmic logic with frontend development, the project bridges the gap between theoretical understanding and practical learning. Students are able to observe how character frequencies influence the construction of the binary tree and how shorter binary codes are assigned to more frequent characters. The visual step-by-step animation enhances comprehension and makes abstract concepts in data compression more tangible. Overall, the simulator serves as an engaging educational tool for Computer Architecture coursework.

4.2 Limitations

While the simulator achieves its core functionality, it has a few limitations. The current implementation is browser-based and may not scale well for extremely long input strings due to rendering limitations. The tree visualization is basic and may become cluttered when handling inputs with many unique characters. Additionally, the simulator does not yet support decoding or error handling for non-alphabetic characters and whitespace in complex input strings. All animations are client-side, and performance may vary slightly depending on the device and browser used.

4.3 Scope of Future Work

Several enhancements can be made to extend the capabilities of the simulator. Future versions could include support for decoding the encoded string, real-time bit comparison with fixed-length encoding, and visual indicators for compression efficiency. The tree visualization can be improved with scalable vector graphics (SVG) or canvas-based rendering to handle larger datasets. Additionally, integrating explanatory tooltips or step annotations could make the learning experience even more intuitive. Expanding the project to support other compression algorithms.