Course Project

CS 1037
Computer Science Fundamentals II

Project Overview

Due Date: November 30, 2024, at 11:59 PM
Weight: 20% of the final grade
Submission: Via OWL Brightspace

Learning Objectives

- Implement complex data structures (binary trees, priority queues)
- Develop efficient algorithms for data compression
- Practice file I/O operations and bit manipulation
- Apply memory management principles
- Create well-documented, maintainable code

Background and Objectives

Project Description

This project implements Huffman coding, a fundamental text compression technique in computer science. The implementation will provide hands-on experience with binary trees, priority queues, and file operations while creating a practical compression utility.

Background on Huffman Coding

Text compression is an essential technique in computer science. It can reduce the size of files stored on disk, making better use of disk storage. It also makes it practical to send large amounts of data in compressed form across networks. There are many techniques for text compression; one such technique is called **Huffman encoding**, invented by D.A. Huffman at MIT in 1952. It is encoded based on the Unix compress utility and is part of the JPEG encoding process.

In character encodings we are most familiar with, the same number of bits represent each character: 8 bits for ASCII and 16 bits for Unicode. For example, the 8-bit ASCII code for the character A is 01000001, and for the character B is 01000010. There is a waste of space with the encoding of these characters. For instance, the ASCII code uses the same number of bits for each character, whether used frequently. One way to save space is by reducing the number of bits for some characters, specifically the most common ones, such as A and E, with a more extended bit pattern for the less common letters, such as Q and X.

Huffman coding compresses text by coding the component symbols of a file (letters of the alphabet, spaces, etc.) based on their occurrence frequency. The more often a symbol occurs, the shorter its code. One issue with variable-length codes is knowing where the code ends and where the next one starts. This is not a problem if no symbol's code is the start of the code for another symbol, and the Huffman code has this property.

The way that Huffman coding works for a set of symbols is that it builds a binary tree (called the **Huffman tree**) based on the frequencies of the symbols, and the codes can then be read off the tree, as **explained below**. We will first demonstrate how a Huffman tree is built with an example. Consider the following:

Set of characters and their frequency of use in some file, sorted into ascending order by frequency:

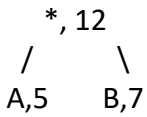
Symbol	Frequency
A	5
B	7
C	10
D	15
E	20
F	45

We will build a binary tree made up of nodes that store two data items: a symbol and the node's frequency. The symbols themselves will be stored only at the leaf nodes (the symbol field has no meaning in the interior nodes).

We first make a Huffman tree consisting of just a root containing a **Huffman pair** for every symbol/frequency pair in our set and add them to an ordered list of nodes such that the nodes are in ascending order by frequency. The ordered list will look like this:

A, 5 B, 7 C, 10 D, 15 E, 20 F, 45

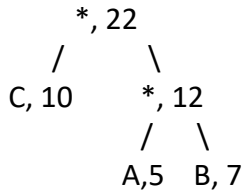
We start building the **Huffman tree** by removing the two nodes with the lowest frequencies from the ordered list and constructing a binary tree with these nodes as leaf nodes and a parent node with a frequency that is the sum of the two lower nodes' frequencies. (The symbol field in an interior node has no meaning and will be denoted by the character "*" in this example.) The resulting binary tree looks like this:



We now insert the new parent node, with frequency 12, into the ordered list at the proper location so that the list is still ordered by frequency. So now the list contains:

C, 10 *, 12 D, 15 E, 20 F, 45

Repeat the previous steps, combining the two lowest-frequency nodes into a binary tree with a new root node. This results in the binary tree:



and the list now contains:

D, 15 E, 20 *, 22 F, 45

Repeat until the list contains only one node, and we are done. This node is the root of the Huffman tree.
Note: The symbols are all in the tree's leaf nodes, and the symbols with the lowest frequencies are furthest from the tree's root.

To get the binary Huffman encoding for a symbol

Start with an empty encoding at the root of the tree. Traverse the tree to the node with the symbol, appending a 0 to the encoding every time you take a branch to the left and a 1 every time you take a branch to the right. For example, the encoding for the symbol A is 10, and the symbol B is 11. Because each symbol is at a leaf, the encoding for one symbol cannot be the start of the encoding for another symbol; therefore, the Huffman encoding is unambiguous.

To decode a Huffman encoding

You can use a Huffman tree to decode previously encoded text with its binary patterns. The decoding algorithm reads each bit from the file, one at a time, and uses this bit to traverse the Huffman tree. If the bit is a 0, you move left in the tree. If the bit is 1, you move right. You do this until you hit a leaf node. Leaf nodes represent characters, so you output that character once you reach a leaf.

For example, suppose we are given the same encoding tree above, and we are asked to decode a second file containing the following bits: 1110010001001010011

Using the Huffman tree, we walk from the root until we find characters, then output them and return to the root. Review Figure 1 below.

Development Tips

- Work step-by-step. Get each part of the encoding program working before starting on the next one. You need to test each function individually before combining everything.
- Start with small test files (two characters, ten characters, one sentence) to practice before you try compressing large text.

- Your implementation should be robust enough to compress any size of text files.
- Your program only has to decompress valid files compressed by your program. You do not need to take special precautions to protect against user error, such as decompressing a file that isn't in the proper compressed format.
- Don't be concerned about the slow reading/writing phase for huge files.
- Implement features incrementally
- Test thoroughly at each step
- Use debugging tools effectively
- Plan memory management early
- Review the textbook from page 290 to page 294 to learn more about the Huffman code.

We read a 1 (right), then a 1 (right). We reach **'b'** and output **b**. Back to root.	11100100010010010011
We read a 1 (right), then a 0 (left). We reach **'a'** and output **a**. (Back to root.)	11100100010010010011
We read a 0 (left), then a 1 (right), then a 0 (left). We reach **'c'** and output **c**.	11100100010010010011
We read a 0 (left), then a 0 (left). We reach **' '** and output a space.	11100100010010010011
We read a 1 (right), then a 0 (left). We reach **'a'** and output **a**.	11100100010010010011
We read a 0 (left), then a 1 (right), then a 0 (left). We reach **'c'** and output **c**.	11100100010010010011
We read a 1 (right), then a 0 (left). We reach **'a'** and output **a**.	11100100010010010011
We read a 0, 1, 1. This is our EOF encoding pattern, so we stop. The overall decoded text is **bac aca**.	

Figure 1: Decoding a Binary Huffman Tree Traversal Sequence

Be aware of the following common issues

- Memory leaks
- File handling errors
- Performance optimization

Technical Requirements

Functional Specifications

Write a program according to the specifications found below:

Core Features

1. **Text File Compression**:
   - Read input text files of any size
   - Generate frequency table of characters
   - Build Huffman tree
   - Create a compressed output file

2. **File Decompression**:
   - Read compressed files

- Reconstruct Huffman tree
- Generate original text file
- Verify data integrity

Input/Output Specifications

1. **Input Files**
   - Format: ASCII text files
   - Size: Up to 100MB
   - Character Set: Standard ASCII (0-127)

2. **Output Files**
   - Compression: `.huf` extension
   - Decompression: `.txt` extension
   - Must maintain original file integrity

Note: The starter kit for this project is available as a ZIP archive on OWL Brightspace.

Non-functional Specifications

- The project is to be done in a team.
- Include README with build instructions.
- Include comments in your code. Add comments at the beginning of your program, indicating who the code's author is and briefly describing the code. Add comments to methods and instance variables.
- Add comments to explain the meaning of potentially confusing and/or significant parts of your code.
- Use C coding conventions and good programming techniques. Read the textbook about comments, coding conventions, and good programming techniques.
- DO NOT put the code inline in the textbox. Do not submit your object or executable files. If you do this and do not attach your source & header files, you will receive a Zero mark!
- To submit, make a single ZIP file with all your source codes & header files (if any) and submit it via OWL by Monday, Nov 30 @ 11:59 PM.
- Submit your programs with proper comments.
- Late submissions without prior arrangement or a valid explanation will result in reduced marks.
- The late policy will be applied to either late submissions or submissions through email.

Academic Integrity

- All work must be original and cite any references used
- **Collaboration policy**: discuss concepts but code individually
- Plagiarism will result in zero grade

What Your Submission Will Be Graded On

Marking Scheme

Component	Weight
Functionality	40%
Code Quality	25%
Documentation	15%
Performance & Efficiency	15%
Test Coverage	5%

Evaluation Criteria

1. Functionality (40%)
   - Correct compression/decompression
   - Handling of all test cases
   - Error handling
   - Feature completeness

2. Code Quality (25%)
   - Code organization
   - Memory management
   - Commenting
   - Style guidelines

3. Documentation (15%)
   - Code documentation
   - README completeness
   - Test documentation
   - Design decisions

4. Performance (15%)
   - Meeting time complexity requirements
   - Meeting space complexity requirements
   - Achieving compression ratios
   - Handling large files

5. Testing (5%)
   - Test coverage
   - Edge case handling
   - Performance testing

Appendix A

Huffman Coding Project - Technical Documentation

Code Documentation Templates

File Header Template

```
/*****************************************************************************
**
 * Filename: <filename>
 * Author: <student name>
 * Student ID: <student id>
 * Version: <version number>
 * Date Created: <date>
 * Last Modified: <date>
 *
 * Description:
 * <Detailed description of what this file does>
 *
 * Dependencies:
 * - List any required libraries or other files
 *
 * Compilation:
 * <Include compilation command>
 *****************************************************************************/
```

Function Documentation Template

```
/*****************************************************************************
**
 * Function: function_name
 *
 * Purpose:
 *    Detailed description of what the function does
 *
 * Parameters:
 *    param1 (type) - description of parameter 1
 *    param2 (type) - description of parameter 2
 *
 * Returns:
 *    type - description of return value
 *
```

```
 * Errors:
 *    List possible error conditions and how they're handled
 *
 * Notes:
 *    Any additional information about usage or limitations
 ******************************************************************************/
```

Struct Documentation Template

```
/******************************************************************************
**
 * Structure: struct_name
 *
 * Purpose:
 *    Description of what this structure represents
 *
 * Fields:
 *    field1 (type) - description
 *    field2 (type) - description
 *
 * Notes:
 *    Any special considerations about memory allocation or usage
*******************************************************************************/
```

Appendix B – Test Cases

Basic Test Cases

1. Single Character File
Input: "aaaaa"
Expected Output
- **Frequency Table**: {'a': 5}
- **Compressed Size**: 1 byte
- **Verification**: Must decompress back to "aaaaa"

2. Two Character File
Input: "aabbaa"
Expected Output
- **Frequency Table**: {'a': 4, 'b': 2}
- **Huffman Codes**: {'a': '0', 'b': '1'}
- **Compressed Bit Stream**: "001100"

3. Simple Sentence
Input: "hello world"
Expected Output
- **Frequency Table**:
'h': 1, 'e': 1, 'l': 3, 'o': 2, ' ': 1, 'w': 1, 'r': 1, 'd': 1
- Example compressed output size should be less than the input size

Edge Testing Cases

1. Empty File
Input: ""
Required: The program should handle with appropriate error message

2. Single Character Repeated
Input: "zzzzzzzzzzzzzzz" (15 'z' characters)
Expected: Should compress efficiently

Performance Test Cases
1. Large File Test
Input: 1MB text file
Requirements:
- **Compression Time**: < 5 seconds on standard hardware
- **Compression Ratio**: At least 40% reduction

2. Varied Content Test
Input: Mix of letters, numbers, and special characters
Requirements:
- Must handle all ASCII characters
- Must maintain data integrity

Appendix C

Memory Management Guidelines

Allocation Guidelines – Node Creation

```
// Correct way to allocate a new node
HuffmanNode* createNode(char data, int frequency) {
    HuffmanNode* node = (HuffmanNode*)malloc(sizeof(HuffmanNode));
    if (node == NULL) {
        // Handle allocation failure
        return NULL;
    }
    node->data = data;
    node->frequency = frequency;
    node->left = NULL;
    node->right = NULL;
    return node;
}
```

Tree Management

```
// Guidelines for tree memory management
- Allocate nodes only when necessary
- Keep track of all allocated nodes
- Implement proper tree deletion
- Use helper functions for common operations
```

Deallocation Requirements – Tree Deletion

```
void deleteTree(HuffmanNode* root) {
    if (root == NULL) return;

    // Recursive deletion
    deleteTree(root->left);
    deleteTree(root->right);

    // Free the node
    free(root);
}
```

Memory Cleanup Checklist

```
- Free all dynamically allocated nodes, Free frequency table, Free encoding table
- Close all file handles
- Handle partial cleanup in case of errors
```