

DAA Project Report

File compression/decompression using Huffman algorithm.

Keshav Agarwal (RA2011003010357) [Lead]

Ali Hassan (RA2011003010370)

Problem Statement -

- The objective is to construct a text file compression / decompression program.
- File compression is a need in modern era because it will -

➤ **Save Storage Space**

Compressing folders can prevent problems before they occur, and also save more space for you to store other valuable content, and there will be no lag when using the computer.

➤ **Archive Documents**

Compression tools are also great for archiving old files. One can compress any number of files of the same type into a single file and add corresponding notes to them, which will simplify the file system.

➤ **Prevent Transmission Interruption**

File compression increases data transfer speed. The longer the file takes to send, the more likely the transfer will be interrupted. The time required to transfer a compressed word file is one tenth of the time required to transfer the same uncompressed file, which will greatly reduce the risk of unexpected interruptions, and guarantee the work efficiency.

➤ **Ensure Data Security**

File compression can also hide information. In addition, it is also a good choice to pack multiple files containing sensitive information with compression software and then add a password.

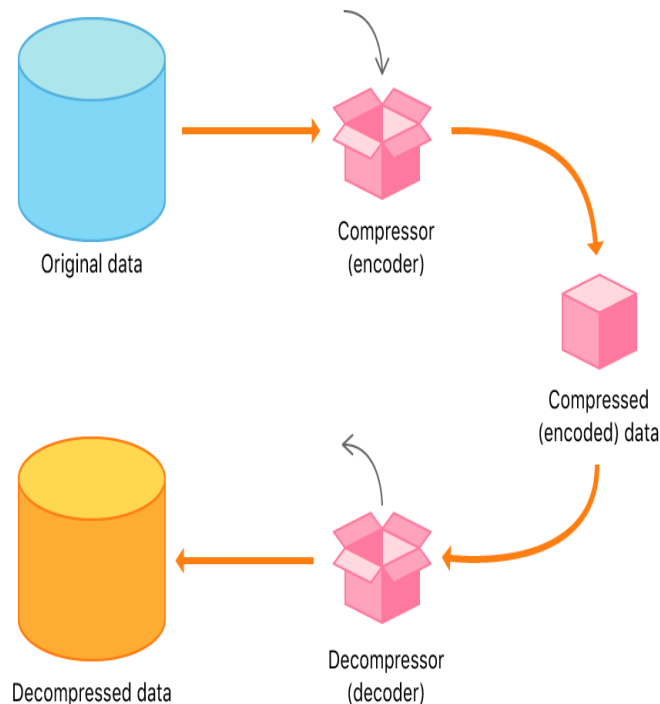
➤ **Meet Server Requirements**

On some Internet servers, file compression is mandatory. The server operator may not allow the transfer of uncompressed files. File compression is also useful when sending email attachments, which often have file size limitations.

BEST APPROACH -

- Greedy algorithm is the best approach for constructing the above-mentioned program because it greedily searches for an optimal solution.

- The Greedy method is the simplest and straightforward approach. It is not an algorithm, but it is a technique. The main function of this approach is that the decision is taken on the basis of the currently available information. Whatever the current information is present, the decision is made without worrying about the effect of the current decision in future.



HUFFMAN ALGORITHM -

- Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code.
- The variable-length codes assigned to input characters are Prefix codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bitstream.
- There are mainly two major parts in Huffman Coding
 - I. Build a Huffman Tree from input characters.
 - II. Traverse the Huffman Tree and assign codes to characters.
- Steps to build Huffman Tree
 1. Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.

2. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)
3. Extract two nodes with the minimum frequency from the min heap.
4. Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
5. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

Let us understand the algorithm with an example:

➤ Real-life applications of Huffman Encoding-

- Huffman encoding is widely used in compression formats like GZIP, PKZIP (WinZip) and BZIP2.
- Multimedia codecs like JPEG, PNG and MP3 uses Huffman encoding (to be more precised the prefix codes)
- Huffman encoding still dominates the compression industry since newer arithmetic and range coding schemes are avoided due to their patent issues.

Implementation:

Project supports two functions:

- 1) Encode: Compresses input file passed.
- 2) Decode: Decompresses Huffman coded file passed back to its original file.

Encoding Source Code:

```
#include <iostream>
#include "huffman.hpp"
using namespace std;

int main(int argc, char* argv[]) {
    if (argc != 3) {
        cout << "Failed to detect Files";
        exit(1);
    }

    huffman f(argv[1], argv[2]);
    f.compress();
    cout << "Compressed successfully" << endl;

    return 0;
}
```

Decoding Source Code:

```
#include <iostream>
#include "huffman.hpp"
using namespace std;

int main(int argc, char* argv[]) {
    if (argc != 3) {
        cout << "Failed to detect Files";
        exit(1);
    }

    huffman f(argv[1], argv[2]);
    f.decompress();
    cout << "Decompressed successfully" << endl;

    return 0;
}
```

Huffman Code:

```

#include "huffman.hpp"

void huffman::createArr() {
    for (int i = 0; i < 128; i++) {
        arr.push_back(new Node());
        arr[i]->data = i;
        arr[i]->freq = 0;
    }
}

void huffman::traverse(Node* r, string str) {
    if (r->left == NULL && r->right == NULL) {
        r->code = str;
        return;
    }

    traverse(r->left, str + '0');
    traverse(r->right, str + '1');
}

int huffman::binToDec(string inStr) {
    int res = 0;
    for (auto c : inStr) {
        res = res * 2 + c - '0';
    }
    return res;
}

string huffman::decToBin(int inNum) {
    string temp = "", res = "";
    while (inNum > 0) {
        temp += (inNum % 2 + '0');
        inNum /= 2;
    }
    res.append(8 - temp.length(), '0');
    for (int i = temp.length() - 1; i >= 0; i--) {
        res += temp[i];
    }
    return res;
}

void huffman::buildTree(char a_code, string& path) {
    Node* curr = root;
    for (int i = 0; i < path.length(); i++) {
        if (path[i] == '0') {
            if (curr->left == NULL) {
                curr->left = new Node();
            }
        }
    }
}

```

```

    }
    curr = curr->left;
}
else if (path[i] == '1') {
    if (curr->right == NULL) {
        curr->right = new Node();
    }
    curr = curr->right;
}
}
curr->data = a_code;
}

void huffman::createMinHeap() {
    char id;
    inFile.open(inFileName, ios::in);
    inFile.get(id);
    //Incrementing frequency of characters that appear in the input file
    while (!inFile.eof()) {
        arr[id]->freq++;
        inFile.get(id);
    }
    inFile.close();
    //Pushing the Nodes which appear in the file into the priority queue (Min Heap)
    for (int i = 0; i < 128; i++) {
        if (arr[i]->freq > 0) {
            minHeap.push(arr[i]);
        }
    }
}

void huffman::createTree() {
    //Creating Huffman Tree with the Min Heap created earlier
    Node *left, *right;
    priority_queue <Node*, vector<Node*>, Compare> tempPQ(minHeap);
    while (tempPQ.size() != 1)
    {
        left = tempPQ.top();
        tempPQ.pop();

        right = tempPQ.top();
        tempPQ.pop();

        root = new Node();
        root->freq = left->freq + right->freq;

        root->left = left;
    }
}

```

```

        root->right = right;
        tempPQ.push(root);
    }
}

void huffman::createCodes() {
    //Traversing the Huffman Tree and assigning specific codes to each character
    traverse(root, "");
}

void huffman::saveEncodedFile() {
    //Saving encoded (.huf) file
    inFile.open(inFileName, ios::in);
    outFile.open(outFileName, ios::out | ios::binary);
    string in = "";
    string s = "";
    char id;

    //Saving the meta data (huffman tree)
    in += (char)minHeap.size();
    priority_queue<Node*, vector<Node*>, Compare> tempPQ(minHeap);
    while (!tempPQ.empty()) {
        Node* curr = tempPQ.top();
        in += curr->data;
        //Saving 16 decimal values representing code of curr->data
        s.assign(127 - curr->code.length(), '0');
        s += '1';
        s += curr->code;
        //Saving decimal values of every 8-bit binary code
        in += (char)binToDec(s.substr(0, 8));
        for (int i = 0; i < 15; i++) {
            s = s.substr(8);
            in += (char)binToDec(s.substr(0, 8));
        }
        tempPQ.pop();
    }
    s.clear();

    //Saving codes of every character appearing in the input file
    inFile.get(id);
    while (!inFile.eof()) {
        s += arr[id]->code;
        //Saving decimal values of every 8-bit binary code
        while (s.length() > 8) {
            in += (char)binToDec(s.substr(0, 8));
            s = s.substr(8);
        }
    }
}

```



```

        inFile.get(id);
    }

    //Finally if bits remaining are less than 8, append o's
    int count = 8 - s.length();
    if (s.length() < 8) {
        s.append(count, 'o');
    }
    in += (char)binToDec(s);
    //append count of appended o's
    in += (char)count;

    //write the in string to the output file
    outFile.write(in.c_str(), in.size());
    inFile.close();
    outFile.close();
}

void huffman::saveDecodedFile() {
    inFile.open(inFileName, ios::in | ios::binary);
    outFile.open(outFileName, ios::out);
    unsigned char size;
    inFile.read(reinterpret_cast<char*>(&size), 1);
    //Reading count at the end of the file which is number of bits appended to make final
    value 8-bit
    inFile.seekg(-1, ios::end);
    char counto;
    inFile.read(&counto, 1);
    //Ignoring the meta data (huffman tree) (1 + 17 * size) and reading remaining file
    inFile.seekg(1 + 17 * size, ios::beg);

    vector<unsigned char> text;
    unsigned char textseg;
    inFile.read(reinterpret_cast<char*>(&textseg), 1);
    while (!inFile.eof()) {
        text.push_back(textseg);
        inFile.read(reinterpret_cast<char*>(&textseg), 1);
    }

    Node *curr = root;
    string path;
    for (int i = 0; i < text.size() - 1; i++) {
        //Converting decimal number to its equivalent 8-bit binary code
        path = decToBin(text[i]);
        if (i == text.size() - 2) {
            path = path.substr(0, 8 - counto);
        }
    }
}

```

```

//Traversing huffman tree and appending resultant data to the file
for (int j = 0; j < path.size(); j++) {
    if (path[j] == '0') {
        curr = curr->left;
    }
    else {
        curr = curr->right;
    }

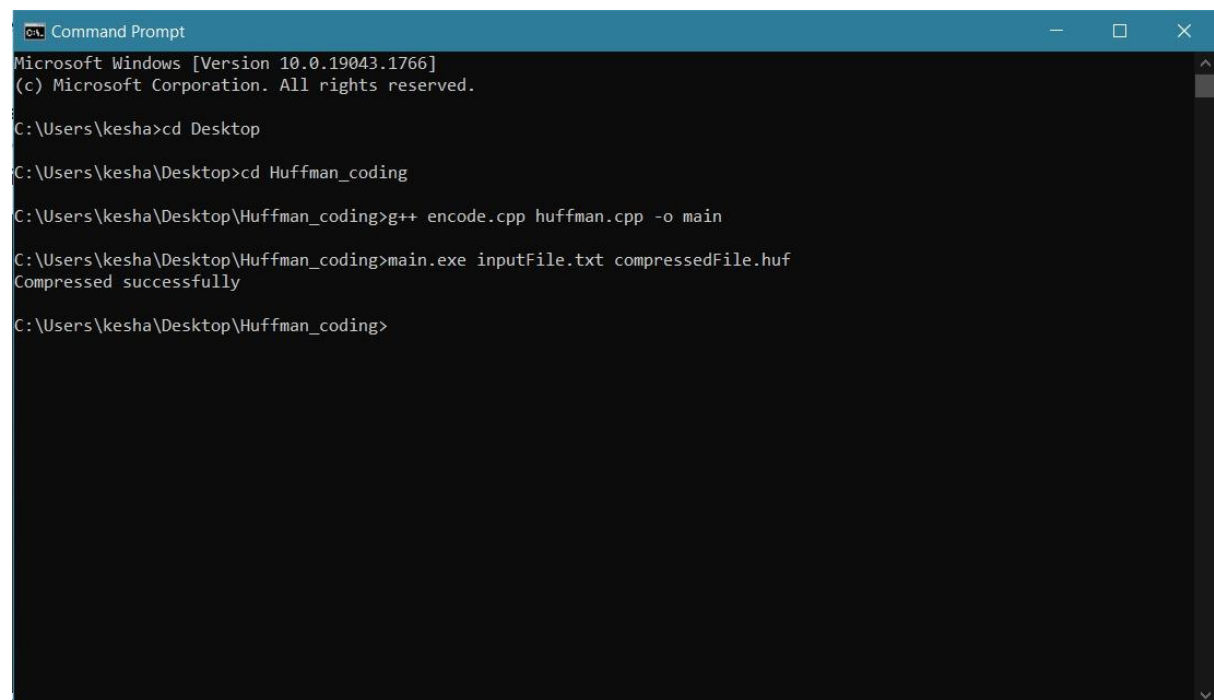
    if (curr->left == NULL && curr->right == NULL) {
        outFile.put(curr->data);
        curr = root;
    }
}
}
inFile.close();
outFile.close();
}

void huffman::getTree() {
    inFile.open(inFileName, ios::in | ios::binary);
    //Reading size of MinHeap
    unsigned char size;
    inFile.read(reinterpret_cast<char*>(&size), 1);
    root = new Node();
    //next size * (1 + 16) characters contain (char)data and (string)code[in decimal]
    for (int i = 0; i < size; i++) {
        char aCode;
        unsigned char hCodeC[16];
        inFile.read(&aCode, 1);
        inFile.read(reinterpret_cast<char*>(hCodeC), 16);
        //converting decimal characters into their binary equivalent to obtain code
        string hCodeStr = "";
        for (int i = 0; i < 16; i++) {
            hCodeStr += decToBin(hCodeC[i]);
        }
        //Removing padding by ignoring first (127 - curr->code.length()) '0's and next '1'
        character
        int j = 0;
        while (hCodeStr[j] == '0') {
            j++;
        }
        hCodeStr = hCodeStr.substr(j+1);
        //Adding node with aCode data and hCodeStr string to the huffman tree
        buildTree(aCode, hCodeStr);
    }
    inFile.close();
}

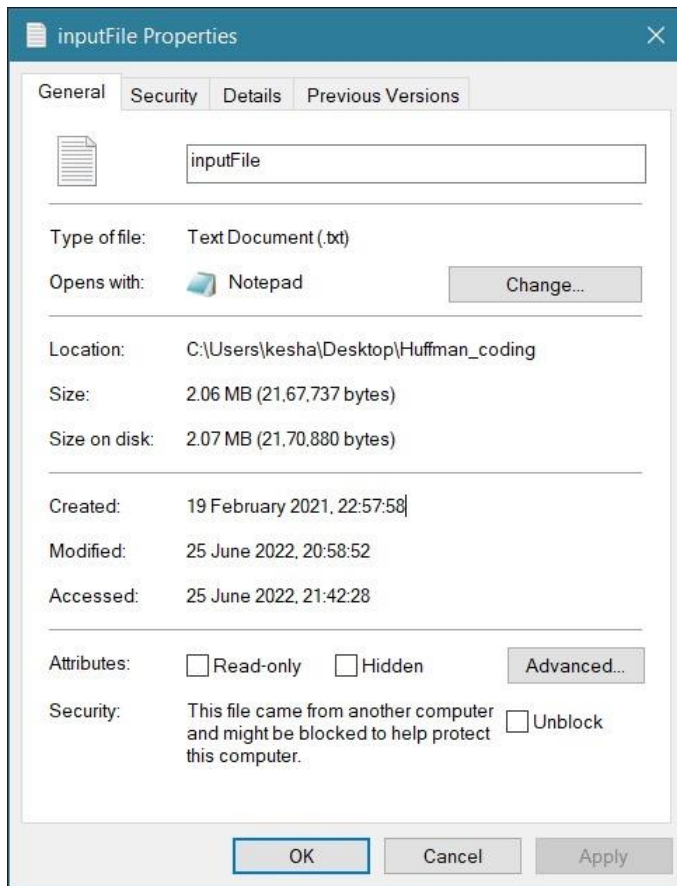
```

```
}  
  
void huffman::compress() {  
    createMinHeap();  
    createTree();  
    createCodes();  
    saveEncodedFile();  
}  
  
void huffman::decompress() {  
    getTree();  
    saveDecodedFile();  
}
```

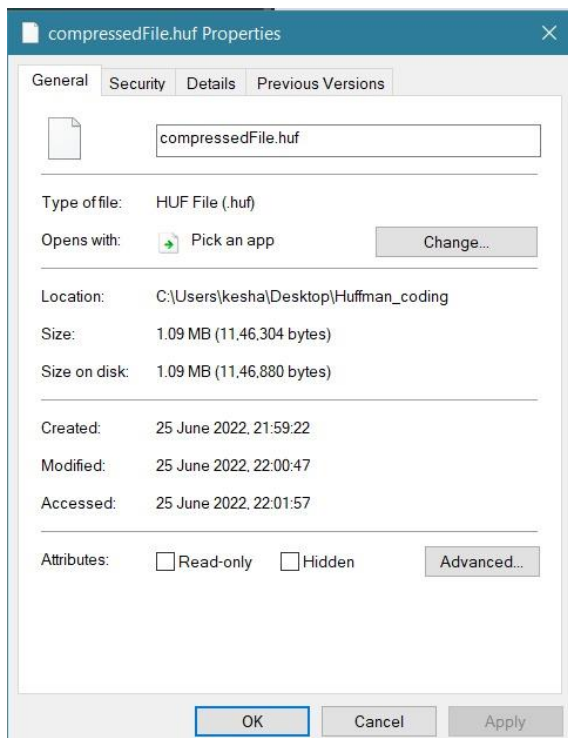
Compression Output:



```
Command Prompt  
Microsoft Windows [Version 10.0.19043.1766]  
(c) Microsoft Corporation. All rights reserved.  
  
C:\Users\kesha>cd Desktop  
  
C:\Users\kesha\Desktop>cd Huffman_coding  
  
C:\Users\kesha\Desktop\Huffman_coding>g++ encode.cpp huffman.cpp -o main  
  
C:\Users\kesha\Desktop\Huffman_coding>main.exe inputFile.txt compressedFile.huf  
Compressed successfully  
  
C:\Users\kesha\Desktop\Huffman_coding>
```

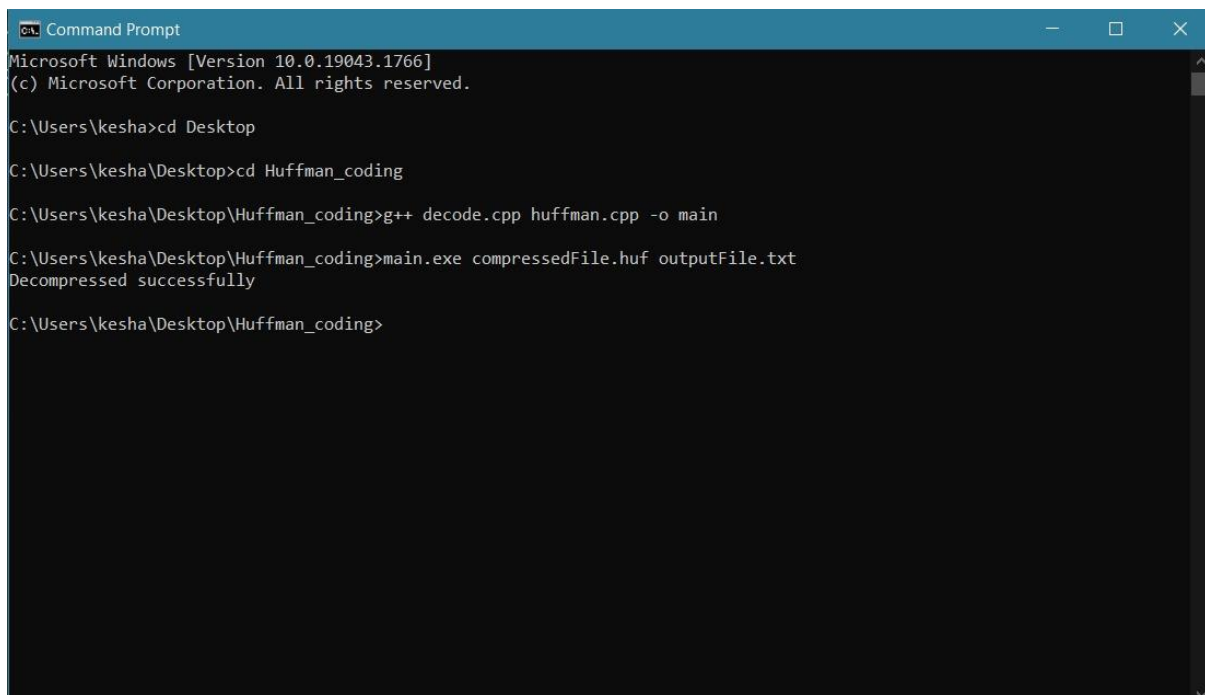


Normal File:



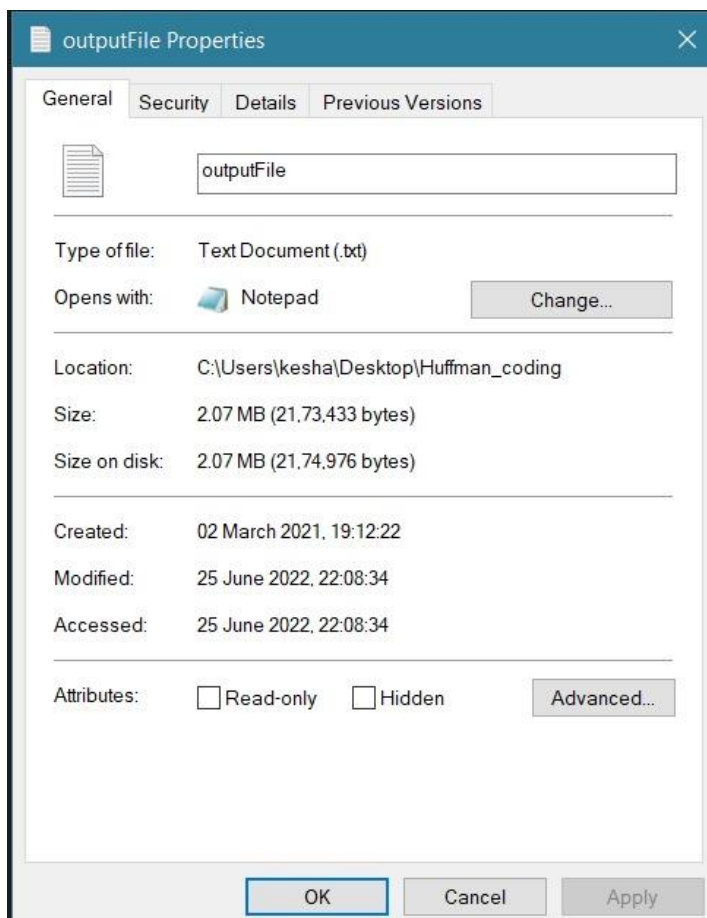
After Compression File:

Decompression Output:



```
Command Prompt
Microsoft Windows [Version 10.0.19043.1766]
(c) Microsoft Corporation. All rights reserved.

C:\Users\kesha>cd Desktop
C:\Users\kesha\Desktop>cd Huffman_coding
C:\Users\kesha\Desktop\Huffman_coding>g++ decode.cpp huffman.cpp -o main
C:\Users\kesha\Desktop\Huffman_coding>main.exe compressedFile.huf outputFile.txt
Decompressed successfully
C:\Users\kesha\Desktop\Huffman_coding>
```



Decompressed File:

Result:

This project is just an implementation of Huffman coding, it is not as efficient as the compression algorithm used currently to compress files.

Example: inputFile.txt (2.7MB) is compressed to compressedFile.huf (1.9MB) file and decompressed back to ouputFile.txt (2.7MB).