Fatma Kausar

CSC382

Prof Yumei Huo

Final Project – Huffman Coding

**PART I**

#include <iostream>

#include <string>

#include <queue>

#include <unordered\_map>

#include <iomanip>

using namespace std;

#define EMPTY\_STRING ""

// A Tree node

struct Node

{

char ch;

int freq;

Node \*left, \*right;

};

// Function to allocate a new tree node

Node\* getNode(char ch, int freq, Node\* left, Node\* right)

{

Node\* node = new Node();

node->ch = ch;

node->freq = freq;

node->left = left;

node->right = right;

return node;

}

// Comparison object to be used to order the heap

struct comp

{

bool operator()(Node\* l, Node\* r)

{

// highest priority item has lowest frequency

return l->freq > r->freq;

}

};

// Utility function to check if Huffman Tree contains only a single node

bool isLeaf(Node\* root) {

return root->left == nullptr && root->right == nullptr;

}

// Traverse the Huffman Tree and store Huffman Codes in a map.

void encode(Node\* root, string str, unordered\_map<char, string> &huffmanCode)

{

if (root == nullptr) {

return;

}

// found a leaf node

if (isLeaf(root)) {

huffmanCode[root->ch] = (str != EMPTY\_STRING) ? str : "1";

}

encode(root->left, str + "0", huffmanCode);

encode(root->right, str + "1", huffmanCode);

}

// Traverse the Huffman Tree and decode the encoded string

void decode(Node\* root, int &index, string str)

{

if (root == nullptr) {

return;

}

// found a leaf node

if (isLeaf(root)) {

cout << root->ch;

return;

}

index++;

if (str[index] == '0') {

decode(root->left, index, str);

}

else {

decode(root->right, index, str);

}

}

// Builds Huffman Tree and decode given input text

void buildHuffmanTree(string text)

{

// base case: empty string

if (text == EMPTY\_STRING) {

return;

}

// count frequency of appearance of each character

// and store it in a map

unordered\_map<char, int> freq;

for (char ch : text) {

freq[ch]++;

}

// Create a priority queue to store live nodes of

// Huffman tree;

priority\_queue<Node\*, vector<Node\*>, comp> pq;

// Create a leaf node for each character and add it

// to the priority queue.

for (auto pair : freq) {

pq.push(getNode(pair.first, pair.second, nullptr, nullptr));

}

// do till there is more than one node in the queue

while (pq.size() != 1)

{

// Remove the two nodes of highest priority

// (lowest frequency) from the queue

Node \*left = pq.top(); pq.pop();

Node \*right = pq.top(); pq.pop();

// Create a new internal node with these two nodes

// as children and with frequency equal to the sum

// of the two nodes' frequencies. Add the new node

// to the priority queue.

int sum = left->freq + right->freq;

pq.push(getNode('\0', sum, left, right));

}

// root stores pointer to root of Huffman Tree

Node\* root = pq.top();

// Traverse the Huffman Tree and store Huffman Codes

// in a map. Also print them

unordered\_map<char, string> huffmanCode;

encode(root, EMPTY\_STRING, huffmanCode);

cout << "Huffman Codes are:\n" << '\n';

cout << left << setw(15) << "Character" << setw(10) << "Code" << setw(10) << "Frequency" << endl;

cout << setfill('-') << setw(40) << "-" << endl;

cout << setfill(' ');

for (auto pair : huffmanCode) {

cout << setw(15) << pair.first << setw(10) << pair.second << setw(10) << freq[pair.first] << '\n';

}

cout << "\nOriginal string is:\n" << text << '\n';

// Print encoded string

string str;

for (char ch : text) {

str += huffmanCode[ch];

}

cout << "\nEncoded string is:\n" << str << '\n';

cout << "\nDecoded string is:\n";

if (isLeaf(root))

{

// Special case: For input like a, aa, aaa, etc

while (root->freq--) {

cout << root->ch;

}

}

else

{

// Traverse the Huffman Tree again and this time

// decode the encoded string

int index = -1;

while (index < (int)str.size() - 1) {

decode(root, index, str);

}

}

}

int main()

{

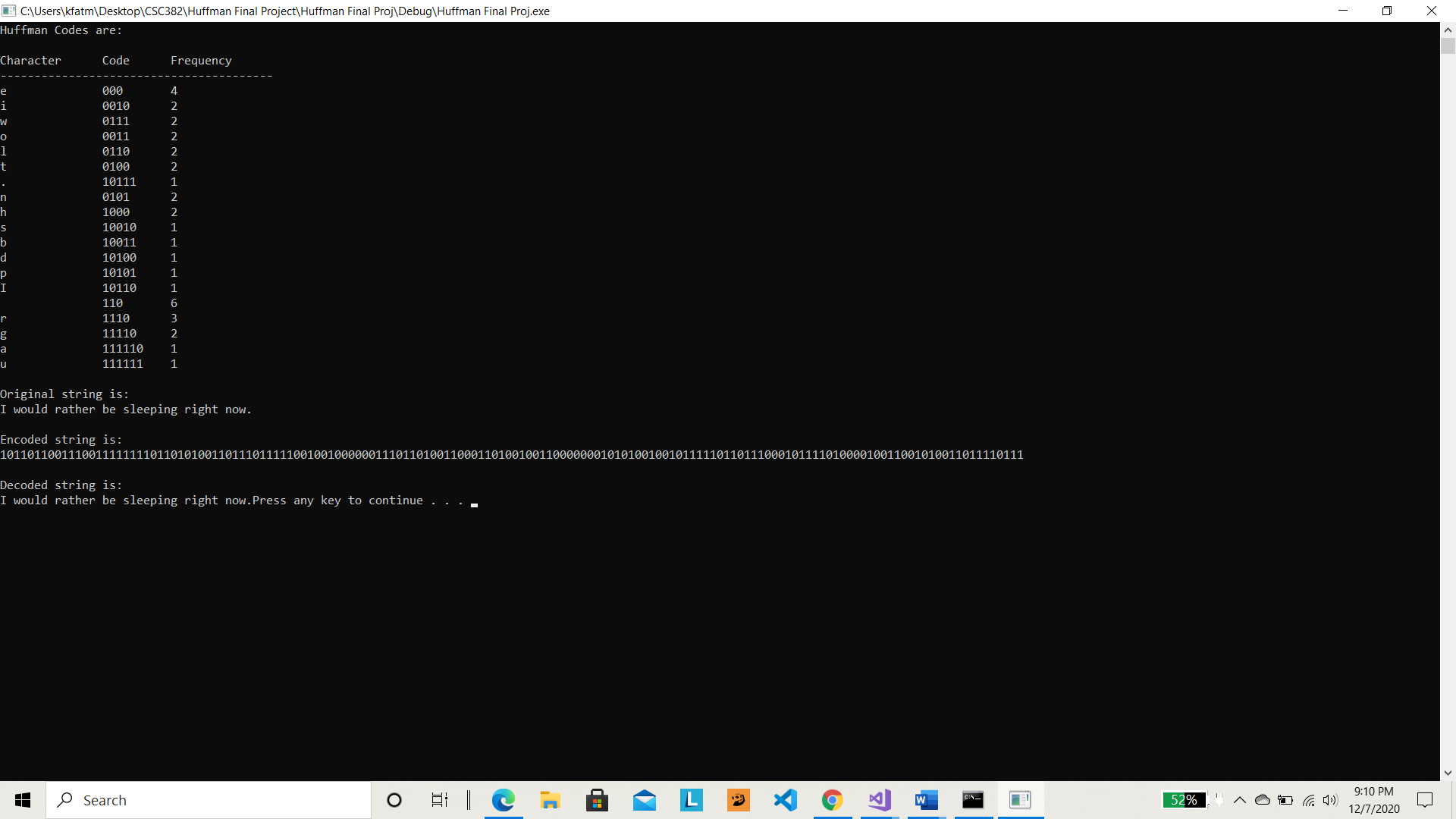
string text = "I would rather be sleeping right now.";

buildHuffmanTree(text);

system("PAUSE");

return 0;

}



**PART II**

#include <iostream>

#include <string>

#include <queue>

#include <unordered\_map>

#include <iomanip>

#include <fstream>

#include <bitset>

using namespace std;

#define EMPTY\_STRING ""

// TREE NODE

struct Node

{

char ch;

int freq;

Node \*left, \*right;

};

// ALLOCATE NEW TREE NODE

Node\* getNode(char ch, int freq, Node\* left, Node\* right)

{

Node\* node = new Node();

node->ch = ch;

node->freq = freq;

node->left = left;

node->right = right;

return node;

}

// COMPARE TO ORDER THE HEAP

struct comp

{

bool operator()(Node\* l, Node\* r)

{

// highest priority item has lowest frequency

return l->freq > r->freq;

}

};

// CHECK IF HUFFMAN TREE CONTAINS ONLY ONE NODE

bool isLeaf(Node\* root) {

return root->left == nullptr && root->right == nullptr;

}

// TRAVERSE HUFFMAN TREE AND STORE CODES IN A MAP

void encode(Node\* root, string str, unordered\_map<char, string> &huffmanCode)

{

if (root == nullptr) {

return;

}

// found a leaf node

if (isLeaf(root)) {

huffmanCode[root->ch] = (str != EMPTY\_STRING) ? str : "1";

}

encode(root->left, str + "0", huffmanCode);

encode(root->right, str + "1", huffmanCode);

}

// TRAVERSE HUFFMAN TREE AND DECODE THE ENCODED STRING

void decode(Node\* root, int &index, string str)

{

if (root == nullptr) {

return;

}

// found a leaf node

if (isLeaf(root)) {

cout << root->ch;

return;

}

index++;

if (str[index] == '0') {

decode(root->left, index, str);

}

else {

decode(root->right, index, str);

}

}

// BUILDS HUFFMAN TREE AND DECODES

string buildHuffmanTree(string text)

{

// base case: empty string

if (text == EMPTY\_STRING) {

return text;

}

// count frequency of appearance of each character

// and store it in a map

unordered\_map<char, int> freq;

for (char ch : text) {

freq[ch]++;

}

// Create a priority queue to store live nodes of

// Huffman tree;

priority\_queue<Node\*, vector<Node\*>, comp> pq;

// Create a leaf node for each character and add it

// to the priority queue.

for (auto pair : freq) {

pq.push(getNode(pair.first, pair.second, nullptr, nullptr));

}

// do till there is more than one node in the queue

while (pq.size() != 1)

{

// Remove the two nodes of highest priority

// (lowest frequency) from the queue

Node \*left = pq.top(); pq.pop();

Node \*right = pq.top(); pq.pop();

// Create a new internal node with these two nodes

// as children and with frequency equal to the sum

// of the two nodes' frequencies. Add the new node

// to the priority queue.

int sum = left->freq + right->freq;

pq.push(getNode('\0', sum, left, right));

}

// root stores pointer to root of Huffman Tree

Node\* root = pq.top();

// Traverse the Huffman Tree and store Huffman Codes

// in a map. Also print them

unordered\_map<char, string> huffmanCode;

encode(root, EMPTY\_STRING, huffmanCode);

cout << "Huffman Codes are:\n" << '\n';

cout << left << setw(15) << "Character" << setw(10) << "Code" << setw(10) << "Frequency" << endl;

cout << setfill('-') << setw(40) << "-" << endl;

cout << setfill(' ');

for (auto pair : huffmanCode) {

cout << setw(15) << pair.first << setw(10) << pair.second << setw(10) << freq[pair.first] << '\n';

}

cout << "\nOriginal string is:\n" << text << '\n';

// Print encoded string

string str;

for (char ch : text) {

str += huffmanCode[ch];

}

cout << "\nEncoded string is:\n" << str << '\n';

cout << "\nDecoded string is:\n";

if (isLeaf(root))

{

// Special case: For input like a, aa, aaa, etc

while (root->freq--) {

cout << root->ch;

}

}

else

{

// Decode the encoded string

int index = -1;

while (index < (int)str.size() - 1) {

decode(root, index, str);

}

}

return str;

}

// Convert to byte

char byteToChar(string byte) {

bitset<8> temp(byte);

return temp.to\_ulong();

}

// Compresses the encoded string

string compress(string str) {

string text = "";

for (int i = 0; i < str.length(); i += 8) {

if (i + 8 >= str.length())

break;

char ch = byteToChar(str.substr(i, i + 8));

text += ch;

}

return text;

}

// Reads file and stores encoded string in binary file

void encodeFile(string fileName) {

//Read a file

string text = "";

ifstream file(fileName);

if (file.is\_open()) {

string line = "";

while (getline(file, line)) {

text += line;

}

file.close();

}

//Encode the file into a new binary file

ofstream outputFile("encoded" + fileName, ios::out | ios::binary);

if (!outputFile) {

cout << "Cannot open file" << endl;

return;

}

string compressedStr = compress(buildHuffmanTree(text));

for (int i = 0; i < compressedStr.length(); i++) {

outputFile << compressedStr[i];

}

outputFile.close();

}

// Decompress encoded string

string charToByte(char ch) {

bitset<8> temp(ch);

return temp.to\_string();

}

// Reads encoded file and decompresses and decodes

void decodeFile(string fileName) {

// Read in the encoded file

ifstream encodedFile("encoded" + fileName, ios::in | ios::binary);

string text = "";

if (encodedFile.is\_open()) {

char c;

while (encodedFile.good()) {

encodedFile.get(c);

text += charToByte(c);

}

encodedFile.close();

}

// Decode the string using the Huffman Tree

// Output decoded string into the decoded file

ofstream outputFile("decoded" + fileName, ios::out);

if (!outputFile) {

cout << "Cannot open file" << endl;

return;

}

for (int i = 0; i < text.length(); i++) {

outputFile << text[i];

}

outputFile.close();

}

int main()

{

for (int i = 1; i <= 20; i++) {

string fileName = "input" + to\_string(i) + ".txt";

encodeFile(fileName);

}

system("PAUSE");

return 0;

}

|  |  |  |  |
| --- | --- | --- | --- |
| **Input File Number** | **Size of original file** | **Size of compressed file** | **Compression Ratio** |
| **1** | 246 | 130 | .52 |
| **2** | 168 | 90 | .53 |
| **3** | 588 | 332 | .56 |
| **4** | 1271 | 741 | .58 |
| **5** | 1271 | 741 | .58 |
| **6** | 677 | 363 | .53 |
| **7** | 1100 | 590 | .53 |
| **8** | 312 | 172 | .55 |
| **9** | 1135 | 679 | .59 |
| **10** | 1050 | 630 | .60 |
| **11** | 258 | 133 | .51 |
| **12** | 438 | 245 | .55 |
| **13** | 819 | 450 | .54 |
| **14** | 819 | 450 | .54 |
| **15** | 253 | 133 | .52 |
| **16** | 922 | 495 | .53 |
| **17** | 834 | 457 | .54 |
| **18** | 287 | 533 | .53 |
| **19** | 696 | 376 | .54 |
| **20** | 707 | 381 | .53 |