#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_TREE\_HT 100

// A node of the heap (priority queue)

struct MinHeapNode {

char data; // One character

unsigned freq; // Frequency of the character

struct MinHeapNode \*left, \*right; // Left and right child nodes

};

// A min heap: A collection of nodes

struct MinHeap {

unsigned size; // Current size of min heap

unsigned capacity; // Capacity of min heap

struct MinHeapNode\*\* array; // Array of min heap node pointers

};

// A function to create a new min heap node

struct MinHeapNode\* newMinHeapNode(char data, unsigned freq) {

struct MinHeapNode\* temp = (struct MinHeapNode\*)malloc(sizeof(struct MinHeapNode));

temp->left = temp->right = NULL;

temp->data = data;

temp->freq = freq;

return temp;

}

// A function to create a min heap

struct MinHeap\* createMinHeap(unsigned capacity) {

struct MinHeap\* minHeap = (struct MinHeap\*)malloc(sizeof(struct MinHeap));

minHeap->size = 0;

minHeap->capacity = capacity;

minHeap->array = (struct MinHeapNode\*)malloc(minHeap->capacity \* sizeof(struct MinHeapNode));

return minHeap;

}

// A utility function to swap two min heap nodes

void swapMinHeapNode(struct MinHeapNode\*\* a, struct MinHeapNode\*\* b) {

struct MinHeapNode\* t = \*a;

\*a = \*b;

\*b = t;

}

// A standard function to min-heapify at a given index

void minHeapify(struct MinHeap\* minHeap, int idx) {

int smallest = idx;

int left = 2 \* idx + 1;

int right = 2 \* idx + 2;

if (left < minHeap->size && minHeap->array[left]->freq < minHeap->array[smallest]->freq)

smallest = left;

if (right < minHeap->size && minHeap->array[right]->freq < minHeap->array[smallest]->freq)

smallest = right;

if (smallest != idx) {

swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);

minHeapify(minHeap, smallest);

}

}

// A function to check if size of heap is 1 or not

int isSizeOne(struct MinHeap\* minHeap) {

return (minHeap->size == 1);

}

// A function to extract the minimum value node from heap

struct MinHeapNode\* extractMin(struct MinHeap\* minHeap) {

struct MinHeapNode\* temp = minHeap->array[0];

minHeap->array[0] = minHeap->array[minHeap->size - 1];

--minHeap->size;

minHeapify(minHeap, 0);

return temp;

}

// A function to insert a new node to the min heap

void insertMinHeap(struct MinHeap\* minHeap, struct MinHeapNode\* minHeapNode) {

++minHeap->size;

int i = minHeap->size - 1;

while (i && minHeapNode->freq < minHeap->array[(i - 1) / 2]->freq) {

minHeap->array[i] = minHeap->array[(i - 1) / 2];

i = (i - 1) / 2;

}

minHeap->array[i] = minHeapNode;

}

// A function to build a min heap

void buildMinHeap(struct MinHeap\* minHeap) {

int n = minHeap->size - 1;

for (int i = (n - 1) / 2; i >= 0; --i) {

minHeapify(minHeap, i);

}

}

// A function to print the huffman codes using the huffman tree

void printHuffmanCodes(struct MinHeapNode\* root, int arr[], int top) {

if (root->left) {

arr[top] = 0;

printHuffmanCodes(root->left, arr, top + 1);

}

if (root->right) {

arr[top] = 1;

printHuffmanCodes(root->right, arr, top + 1);

}

if (!root->left && !root->right) {

printf("%c: ", root->data);

for (int i = 0; i < top; ++i)

printf("%d", arr[i]);

printf("\n");

}

}

// A function to build the Huffman tree

struct MinHeapNode\* buildHuffmanTree(char data[], int freq[], int size) {

struct MinHeapNode \*left, \*right, \*top;

struct MinHeap\* minHeap = createMinHeap(size);

for (int i = 0; i < size; ++i)

minHeap->array[i] = newMinHeapNode(data[i], freq[i]);

minHeap->size = size;

buildMinHeap(minHeap);

while (!isSizeOne(minHeap)) {

left = extractMin(minHeap);

right = extractMin(minHeap);

top = newMinHeapNode('$', left->freq + right->freq);

top->left = left;

top->right = right;

insertMinHeap(minHeap, top);

}

return extractMin(minHeap);

}

// A function to calculate the size of the compressed data

int calculateCompressedSize(char data[], int freq[], int size, struct MinHeapNode\* root) {

int arr[MAX\_TREE\_HT], top = 0;

int compressedSize = 0;

printHuffmanCodes(root, arr, top);

// Count the number of bits for each character

for (int i = 0; i < size; i++) {

int codeLength = 0;

printHuffmanCodes(root, arr, top);

compressedSize += codeLength \* freq[i];

}

return compressedSize;

}

int main() {

char text[1000];

printf("Enter text to compress:\n");

fgets(text, sizeof(text), stdin);

text[strcspn(text, "\n")] = '\0'; // Remove the trailing newline character

int freq[256] = {0};

for (int i = 0; text[i] != '\0'; ++i) {

freq[text[i]]++;

}

int size = 0;

for (int i = 0; i < 256; ++i) {

if (freq[i] > 0) {

size++;

}

}

char data[size];

int f[size];

int index = 0;

for (int i = 0; i < 256; ++i) {

if (freq[i] > 0) {

data[index] = i;

f[index] = freq[i];

index++;

}

}

// Build the Huffman tree

struct MinHeapNode\* root = buildHuffmanTree(data, f, size);

// Print the size of text before compression

printf("Original size: %lu bytes\n", strlen(text));

// Calculate the compressed size using Huffman coding

int compressedSize = calculateCompressedSize(data, f, size, root);

// Print the size of text after compression

printf("Compressed size: %d bits\n", compressedSize);

    return 0;

}