

**Lab Report**

. **5. Decode the text encoded using Huffman.**

Course title: Algorithms Lab

Course code: CSE 222

Summitted to:

Faisal Bin Abul Kasem

Lecturer DIU

Department of CSE

Submitted by:

A

1. Mainur Rahman Murad (171-15-8758)

2. Siyamul Islam Siyam (171-15-8865)

3. Eusuf Uddin (71-15-8926)

4. Bipasha Babul Bristy (171-15-9208)

Group: 09

Section: “O”

Department of CSE

Daffodil International University

**Date of Submission:** 26/07/2018

**Topic: Decode the text encoded using Huffman.**

Team No:09

Team Name: **Red DraGonZ**

Team Member Name: ID:

1. Minur Rahman Murad (171-15-8758)

2. Siyamul Islam Siyam (171-15-8865)

3. Eusuf Uddin (71-15-8926)

4. Bipasha Babul Bristy (171-15-9208)

**Intro:**

In [computer science](https://en.wikipedia.org/wiki/Computer_science) and [information theory](https://en.wikipedia.org/wiki/Information_theory), a **Huffman code** is a particular type of optimal [prefix code](https://en.wikipedia.org/wiki/Prefix_code) that is commonly used for [lossless data compression](https://en.wikipedia.org/wiki/Lossless_data_compression). The process of finding and/or using such a code proceeds by means of **Huffman coding**, an algorithm developed by [David A. Huffman](https://en.wikipedia.org/wiki/David_A._Huffman) while he was a [Sc.D.](https://en.wikipedia.org/wiki/Doctor_of_Science) student at [MIT](https://en.wikipedia.org/wiki/Massachusetts_Institute_of_Technology), and published in the 1952 paper "A Method for the Construction of Minimum-Redundancy Codes".

Huffman codes are a widely used and very effective technique for compressing data; savings of 20% to 90% are typical, depending on the characteristics of the data being compressed. We consider the data to be a sequence of characters. Huffman’s greedy algorithm uses a table of the frequencies of occurrence of the characters to build up an optimal way of representing each character as a binary string.

**Approach:**

A divide-and-conquer approach might have us asking which characters should appear in the left and right subtrees and trying to build the tree from the top down. As with the optimal binary search tree, this will lead to to an exponential time algorithm.

A greedy approach places our **n** characters in **n** sub-trees and starts by combining the two least weight nodes into a tree which is assigned the sum of the two leaf node weights as the weight for its root node.

The decoding procedure is deceptively simple. Starting with the first bit in the stream, one then uses successive bits from the stream to determine whether to go left or right in the decoding tree. When we reach a leaf of the tree, we've decoded a character, so we place that character onto the (uncompressed) output stream. The next bit in the input stream is the first bit of the next character.

The time complexity of the Huffman algorithm is **O(nlogn)**. Using a heap to store the weight of each tree, each iteration requires **O(logn)** time to determine the cheapest weight and insert the new weight. There are **O(n)** iterations, one for each item.

***#Code in C***

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdbool.h>

struct tree

{

struct tree\* left;

struct tree\* right;

int count;

char value;

};

struct bitstring\_builder

{

char str[256];

int next\_index;

};

struct codebook

{

char\* codes[256];

};

struct heap

{

struct tree\*\* data;

int length;

int capacity;

};

bool is\_leaf(const struct tree\* t)

{

return !t->left && !t->right;

}

void swap(struct tree\*\* lhs, struct tree\*\* rhs)

{

struct tree\* tmp = \*lhs;

\*lhs = \*rhs;

\*rhs = tmp;

}

/\* The two concat functions are horribly inefficient \*/

void concat(char\*\* dst, const char\* src)

{

int dst\_len = strlen(\*dst);

int src\_len = strlen(src);

\*dst = realloc(\*dst, src\_len + dst\_len + 1);

strcat(\*dst, src);

}

void concat\_char(char\*\* dst, char c)

{

int len = strlen(\*dst);

\*dst = realloc(\*dst, len + 2);

(\*dst)[len] = c;

(\*dst)[len + 1] = '\0';

}

char\* duplicate(const char\* src)

{

int length = strlen(src);

char\* dst = malloc(length + 1);

memcpy(dst, src, length + 1);

return dst;

}

void heap\_push(struct heap\* heap, struct tree\* value)

{

if (heap->capacity == heap->length)

{

heap->capacity = heap->capacity == 0 ? 4 : heap->capacity \* 2;

heap->data = realloc(heap->data, heap->capacity \* sizeof(struct tree\*));

}

heap->data[heap->length++] = value;

int index = heap->length - 1;

while (index)

{

int parent\_index = (index - 1) / 2;

if (heap->data[parent\_index]->count <= heap->data[index]->count)

{

break;

}

swap(&heap->data[parent\_index], &heap->data[index]);

index = parent\_index;

}

}

struct tree\* heap\_pop(struct heap\* heap)

{

if (!heap->length)

{

return NULL;

}

struct tree\* result = heap->data[0];

swap(&heap->data[0], &heap->data[--heap->length]);

int index = 0;

for (;;)

{

int target = index;

int left = 2 \* index + 1;

int right = left + 1;

if (left < heap->length &&

heap->data[left]->count < heap->data[target]->count)

{

target = left;

}

if (right < heap->length &&

heap->data[right]->count < heap->data[target]->count)

{

target = right;

}

if (target == index)

{

break;

}

swap(&heap->data[index], &heap->data[target]);

index = target;

}

return result;

}

void heap\_free(struct heap\* heap)

{

free(heap->data);

}

struct tree\* generate\_tree(const char\* str)

{

int counts[256] = { 0 };

for (; \*str != '\0'; ++str)

{

counts[(unsigned char)\*str] += 1;

}

struct heap heap = { 0 };

int i=0;

for (i; i < sizeof(counts) / sizeof(int); ++i)

{

if (counts[i])

{

struct tree\* tree = calloc(1, sizeof(struct tree));

tree->value = (char)i;

tree->count = counts[i];

heap\_push(&heap, tree);

}

}

while (heap.length > 1)

{

struct tree\* left = heap\_pop(&heap);

struct tree\* right = heap\_pop(&heap);

struct tree\* branch = calloc(1, sizeof(struct tree));

branch->count = left->count + right->count;

branch->left = left;

branch->right = right;

heap\_push(&heap, branch);

}

struct tree\* root = heap\_pop(&heap);

heap\_free(&heap);

return root;

}

void tree\_free(struct tree\* tree)

{

if (!tree) return;

tree\_free(tree->left);

tree\_free(tree->right);

free(tree);

}

void codebook\_recurse(const struct tree\* tree,

struct bitstring\_builder\* builder,

struct codebook\* codebook)

{

if (!tree)

{

return;

}

if (is\_leaf(tree))

{

builder->str[builder->next\_index] = '\0';

codebook->codes[(unsigned char)tree->value] = duplicate(builder->str);

return;

}

builder->str[builder->next\_index++] = '0';

codebook\_recurse(tree->left, builder, codebook);

builder->next\_index -= 1;

builder->str[builder->next\_index++] = '1';

codebook\_recurse(tree->right, builder, codebook);

builder->next\_index -= 1;

}

struct codebook generate\_codebook(const struct tree\* tree)

{

struct codebook codebook = { .codes = { 0 } };

struct bitstring\_builder builder = { .str = { 0 }, .next\_index = 0 };

codebook\_recurse(tree, &builder, &codebook);

return codebook;

}

void codebook\_free(struct codebook\* codebook)

{

int size = sizeof(codebook->codes) / sizeof(codebook->codes[0]);

int i=0;

for (i; i < size; ++i)

{

free(codebook->codes[i]);

}

}

const char\* get\_code(const struct codebook\* codebook, char c)

{

return codebook->codes[(unsigned char)c];

}

char\* encode(const char\* input, struct tree\*\* huffman\_tree,

struct codebook\* codebook)

{

\*huffman\_tree = generate\_tree(input);

\*codebook = generate\_codebook(\*huffman\_tree);

char\* result = duplicate(get\_code(codebook, \*input));

int result\_length = strlen(result);

int result\_capacity = result\_length;

input += 1;

for (; \*input; ++input)

{

concat(&result, get\_code(codebook, \*input));

}

return result;

}

const char\* decode\_recurse(const char\* input, const struct tree\* tree,

char\*\* result)

{

if (!tree)

{

return input;

}

if (is\_leaf(tree))

{

concat\_char(result, tree->value);

return input;

}

if (\*input == '0')

{

return decode\_recurse(input + 1, tree->left, result);

}

else

{

return decode\_recurse(input + 1, tree->right, result);

}

}

char\* decode(const char\* input, const struct tree\* tree)

{

char\* result = calloc(1, 1);

do

{

input = decode\_recurse(input, tree, &result);

}

while (\*input);

return result;

}

int main()

{

struct tree\* tree;

struct codebook codebook;

char text[256];

printf("\n\* \* \* \*\tWelcome to our project(Huffman Encoding & Decoding).....\n\n");

printf("Enter your text:");

scanf("%[^\n]",&text);

char\* encoded = encode(text, &tree, &codebook);

char\* decoded = decode(encoded, tree);

printf("\nFrequency:\n");

int i=0;

for (i; i <256; ++i)

{

if (codebook.codes[i])

{

printf("%c %s\n", (char)i, codebook.codes[i]);

}

}

printf("\nEncoded: %s\n\n", encoded);

printf("Decoded: %s\n", decoded);

tree\_free(tree);

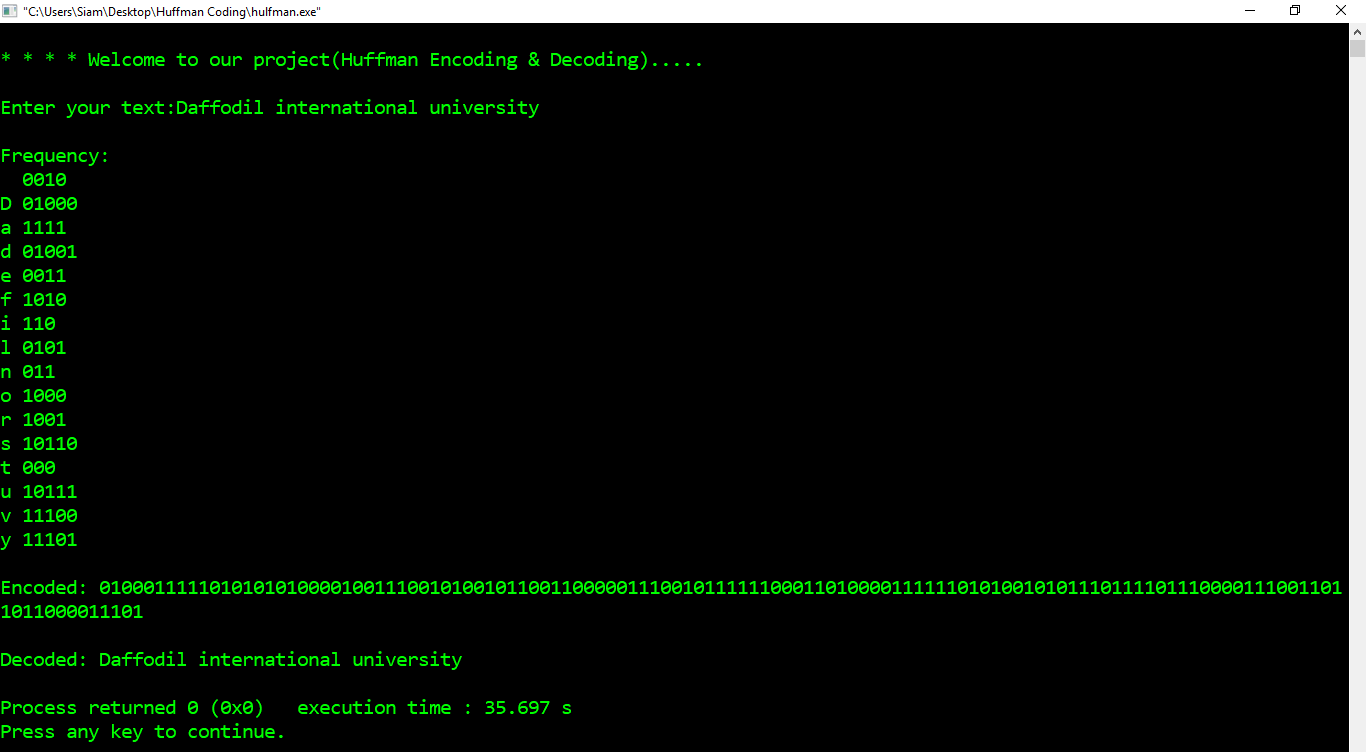
codebook\_free(&codebook);

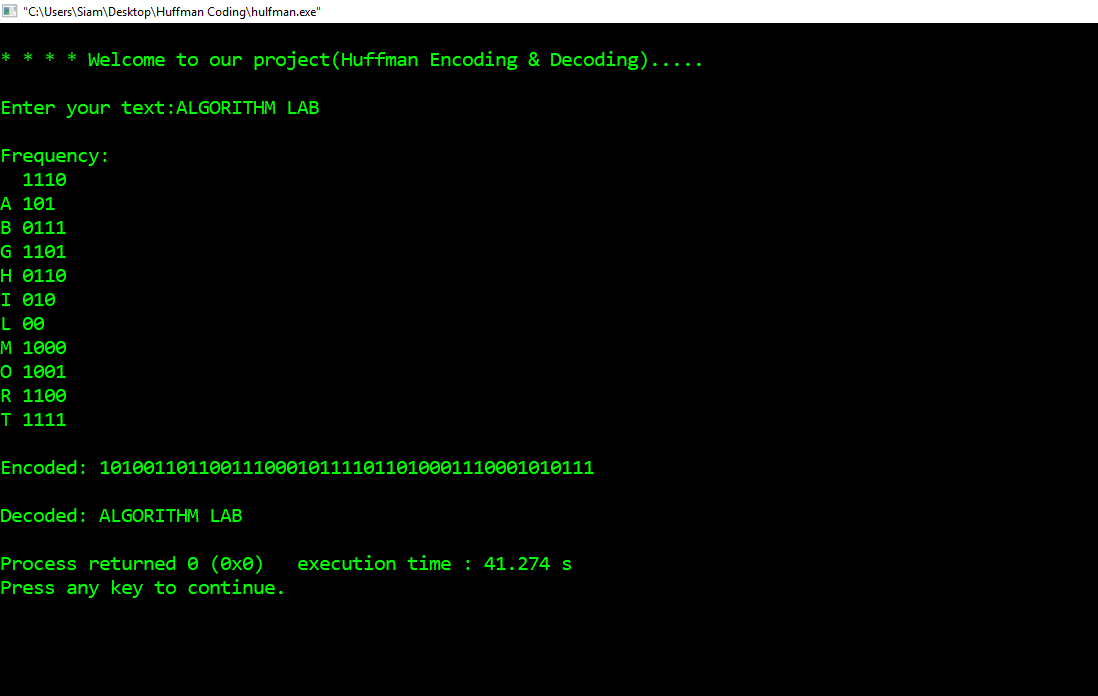
free(encoded);

free(decoded);

return 0;

***Output***





**Faced Problem:**

At first, we face how can we implement the problem and how we can design the algorithm.

Then we face to create some function, like how to concat the frequency for each character,

How to sort the heap nodes, how to set the condition for push pop in the stack. How pass frequency set in decode function. How can we print the final output. And also face some basic problem like,

Pointer

String Builder

Tree traversal