Concept of Operations (ConOps)

# Table of Contents

Contents

[Table of Contents 1](#_Toc103529725)

[1.0 Introduction 1](#_Toc103529726)

[1.1 Project Description 1](#_Toc103529727)

[1.1.1 Background 1](#_Toc103529728)

[1.1.2 Assumptions and Constraints 1](#_Toc103529729)

[1.2 Overview of the Envisioned System 2](#_Toc103529730)

[1.2.1 Overview 2](#_Toc103529731)

[1.2.2 System Scope 2](#_Toc103529732)

[2.0 Description of Envisioned System 2](#_Toc103529733)

[2.1 Needs, Goals and Objectives of Envisioned System 2](#_Toc103529734)

[2.2 Overview of System and Key Elements 3](#_Toc103529735)

# 1.0 Introduction

## 1.1 Project Description

This project aims to implement an efficient and lossless memory compression code based on the Hoffman algorithm.

### 1.1.1 Background

Nowadays, people say those who have information that has power. In this condition, the fast transfer of data has crucial importance. That’s why there comes an issue with finding a way to save the information in the most efficient way, which will affect its fast transformation in the future. This project will interpret and develop a proven hight efficiency lossless compression algorithm. The algorithm also can be considered a low-level encoding algorithm.

### 1.1.2 Assumptions and Constraints

This project is assumed to have significant demand in all the spheres and fields that will use it. Optimization and good memory management are necessary as memory is a vulnerable resource for now. A constraint is a heap memory that may be used more than usual during the algorithm running process. However, the extra used memory will be freed and usable for other operations after implementation. Also, to accomplish the mission, the algorithm should be ready by 16.05.22.

## 1.2 Overview of the Envisioned System

### 1.2.1 Overview

The algorithm is mainly used to compress textual data. The algorithm take the character and the frequency that character repeats. Based on the low-to-high frequency, the algorithm creates a binary tree. The tree uses characters as leaves. From the beginning of the tree, moving the right algorithm saves the value 0, and moving the left saves 1. The same happens for each step until the algorithm riches the leaf with the needed. The created sequence of 0’s and 1’s then used to represent the character instead of the actual long binary code of the character.

### 1.2.2 System Scope

Based on the explorations, created flow charts and initial understanding of the algorithm the code is going to have a complexity ofO (n\*log(n)). The system’s external interfaces are hardware (main memory, heap memory), software (language compilator) of the device that will run the code which means that system will properly communicate with those components.

# 2.0 Description of Envisioned System

## 2.1 Needs, Goals and Objectives of Envisioned System

|  |  |  |
| --- | --- | --- |
| ***Needs*** | ***Goals*** | ***Objectives*** |
| Algorithmic thinking | Get experienced with using c language, | Create an algorithm to help save the same binary code using less space in memory |
| Skills in programming in C language | study working with binary code, | Understand coding techniques to optimize memory management |
| Skills in working with Linux terminal | understand how the complexity of the algorithm changes based on using different coding features |  |

## 2.2 Overview of System and Key Elements

*This section describes at a functional level the various elements that will make up the system, including the users and operators. These descriptions should be implementation free; that is, not specific to any implementation or design but rather a general description of what the system and its elements will be expected to do. Graphics, pictorials, videos, and models may be used to aid this description.*

The algorithm will get as an input two arrays. The first array will contain characters that exist in the text that should be compressed. The second array will contain the frequency of each character accordingly. An example is represented in table below.

|  |  |  |
| --- | --- | --- |
| Array 1 (characters) | Array 2 (frequency) | Number of bits used to save all repeating characters  (frequency\*8(bits)) |
| A | 10 | 80 |
| E | 15 | 120 |
| I | 12 | 96 |
| S | 3 | 24 |
| T | 4 | 32 |
| P | 13 | 104 |
| Newline | 1 | 8 |
| Total number of bits initially | | 464 |

The algorithm based on the frequency (from low to high) creates a binary tree. The tree is going to be created using nods. As we will create tree it will eventually end up in one central value from which using nodes the algorithm will move right or left to fin the value we are searching for. Based on its steps on each direction the algorithm will save the position of the value using (1) for step to right and (0) for step left. The tree based on our table will look like the one bellow.

A picture containing scale, accessory, device

Description automatically generated

The points that connect two characters take the value equals to the sum of the frequencyes of those two values. For example, for values new line and S the connection node have the value 4. And now to read the character A the algorithm saves only two bits 10. So the table changes in the following way

|  |  |  |  |
| --- | --- | --- | --- |
| Array 1 (characters) | Number of bits to save the character | Array 2 (frequency) | Number of bits used to save all repeating characters  (frequency\*(Number of bits)) |
| A | 110 | 10 | 30 |
| E | 10 | 15 | 30 |
| I | 00 | 12 | 24 |
| S | 11111 | 3 | 15 |
| T | 1110 | 4 | 16 |
| P | 01 | 13 | 26 |
| Newline | 11110 | 1 | 5 |
| Total number of bits initially | | | 146 |

*The part lists key terms used in the ConOps and provides a description of their meaning.*

|  |
| --- |
| #include <stdlib.h>    #include <minheap.h>    **struct** huffman\_node {  **char** data;      unsigned **int** frequency;  **struct** huffman\_node \*left;  **struct** huffman\_node \*right;  };  **typedef** **struct** huffman\_node huffman\_node;    huffman\_node \*huffman\_node\_create(**char** data, unsigned **int** frequency)  {      huffman\_node \*node = **malloc**(**sizeof**(huffman\_node));  **if** (node) {          node->data = data;          node->frequency = frequency;          node->left = NULL;          node->right = NULL;      }  **return** node;  }    **void** huffman\_node\_delete(huffman\_node \*node)  {  **if** (node) {          huffman\_node\_delete(node->left);          huffman\_node\_delete(node->right);  **free**(node);      }  }    unsigned **int** max(unsigned **int** a, unsigned **int** b)  {  **return** a > b ? a : b;  }    unsigned **int** huffman\_node\_height(**const** huffman\_node \*node)  {      unsigned **int** height = 0;  **if** (node->left || node->right) {          height = max(node->left ? huffman\_node\_height(node->left) : 0,                  node->right ? huffman\_node\_height(node->right) : 0) + 1;      }  **return** height;  }    **void** huffman\_node\_print(**const** huffman\_node \*node, unsigned **int** indent)  {      unsigned **int** i;  **for** (i = 0; i < indent; i++) {  **printf**("  ");      }  **printf**("%c %u\n", node->data, node->frequency);  **if** (node->left != NULL) {          huffman\_node\_print(node->left, indent + 1);      }  **if** (node->right != NULL) {          huffman\_node\_print(node->right, indent + 1);      }  }    **typedef** **void** huffmanfn(**char**, **const** unsigned **int** \*, **size\_t**);    **void** huffman\_node\_encodings(**const** huffman\_node \*node, unsigned **int** \*arr,          unsigned **int** pos, huffmanfn fun)  {  **if** (node->left) {          arr[pos] = 0;          huffman\_node\_encodings(node->left, arr, pos + 1, fun);      }  **if** (node->right) {          arr[pos] = 1;          huffman\_node\_encodings(node->right, arr, pos + 1, fun);      }  **if** (!(node->left || node->right)) {          fun(node->data, arr, pos);      }  }    **void** huffman(**const** **char** \*letters, **const** **int** \*frequencies, **size\_t** size, huffmanfn fun)  {      minheap \*heap = minheap\_create();      unsigned **int** i;      huffman\_node \*top;      unsigned **int** \*arr;      /\* Populate the heap \*/  **for** (i = 0; i < size; i++) {          minheap\_add(heap, huffman\_node\_create(letters[i], frequencies[i]), frequencies[i]);      }      /\* Build the tree \*/  **while** (minheap\_get\_count(heap) != 1) {          huffman\_node \*left = minheap\_remove\_min(heap);          huffman\_node \*right = minheap\_remove\_min(heap);          top = huffman\_node\_create('$', left->frequency + right->frequency);          top->left = left;          top->right = right;          minheap\_add(heap, top, top->frequency);      }      top = minheap\_remove\_min(heap);      /\* Generate the encodings \*/      arr = **malloc**(huffman\_node\_height(top) \* **sizeof**(unsigned **int**));      huffman\_node\_encodings(top, arr, 0, fun);      /\* Clean up \*/      huffman\_node\_delete(top);  **free**(arr);      minheap\_delete(heap);  } |

Example program:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | **void** print(**char** letter, **const** unsigned **int** \*arr, **size\_t** len)  {      unsigned **int** i;  **printf**("%c: ", letter);  **for** (i = 0; i < len; i++) {  **printf**("%u", arr[i]);      }  **putchar**('\n');  }    **int** main(**void**)  {  **char** letters[] = {'a', 'b', 'c', 'd', 'e', 'f'};  **int** frequencies[] = {45, 13, 12, 16, 9, 5};  **const** **size\_t** size = **sizeof**(letters) / **sizeof**(letters[0]);      huffman(letters, frequencies, size, print);  **return** 0;  } |