Função de Custo e Complexidade

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_TREE_HT 100

// Estrutura para o nó da árvore de Huffman
struct MinHeapNode {
    char data;  // Caractere          O(1)
    unsigned freq;  // Frequência do caractere  O(1)
    struct MinHeapNode *left, *right;  // Filhos esquerdo e direito
O(1)
};

// Estrutura para a fila de prioridade (min-heap)
struct MinHeap {
    unsigned size; O(1)
    unsigned capacity; O(1)
    struct MinHeapNode** array; O(1)
};

// Função para criar um novo nó de min-heap
struct MinHeapNode* newNode(char data, unsigned freq) {
    struct MinHeapNode* temp = (struct
MinHeapNode*)malloc(sizeof(struct MinHeapNode));
    temp->data = data; O(1)
    temp->freq = freq; O(1)
    temp->left = temp->right = NULL;  O(2)
    return temp; O(1)
}

// Função para criar um min-heap
struct MinHeap* createMinHeap(unsigned capacity) {
    struct MinHeap* minHeap = (struct MinHeap*)malloc(sizeof(struct
MinHeap)); O(4)
    minHeap->size = 0; O(1)
    minHeap->capacity = capacity; O(1)
    minHeap->array = (struct MinHeapNode**)malloc(minHeap->capacity *
sizeof(struct MinHeapNode*)); O(d+4)
    return minHeap; O(1)
```

```
}

// Função para trocar dois nós do min-heap
void swapMinHeapNode(struct MinHeapNode** a, struct MinHeapNode** b) {
    struct MinHeapNode* t = *a; O(1)
    *a = *b; O(1)
    *b = t;  O(1)
}

// Função para ajustar o min-heap (heapify)
void minHeapify(struct MinHeap* minHeap, int idx) {
    int smallest = idx; O(1)
    int left = 2 * idx + 1; O(3)
    int right = 2 * idx + 2; O(3)

    if (left < minHeap->size && minHeap->array[left]->freq <
minHeap->array[smallest]->freq)    O(8)
        smallest = left;           O(1)

    if (right < minHeap->size && minHeap->array[right]->freq <
minHeap->array[smallest]->freq) O(8)
        smallest = right;       O(1)

    if (smallest != idx) { O(1)
   O(3) swapMinHeapNode(&minHeap->array[smallest],
&minHeap->array[idx]);
        minHeapify(minHeap, smallest); T(n)= T(n/2)+O(29) = O(log_2 n *
O(29))
    }
}

// Função para verificar se o tamanho do min-heap é 1
int isSizeOne(struct MinHeap* minHeap) {
    return (minHeap->size == 1); O(1)
}

// Função para remover o menor nó do min-heap
struct MinHeapNode* extractMin(struct MinHeap* minHeap) {
    struct MinHeapNode* temp = minHeap->array[0]; O(1)
    minHeap->array[0] = minHeap->array[minHeap->size - 1];O(1)
    --minHeap->size;       O(1)
    minHeapify(minHeap, 0); O(log_2 n * O(29))
```

```
    return temp;      O(1)
}

// Função para inserir um novo nó no min-heap
void insertMinHeap(struct MinHeap* minHeap, struct MinHeapNode*
minHeapNode) {
    ++minHeap->size;  O(1)
    int i = minHeap->size - 1;    O(2)

    while (i && minHeapNode->freq < minHeap->array[(i - 1) / 2]->freq)
{                                                    O(log_2 n)
        minHeap->array[i] = minHeap->array[(i - 1) / 2];  O(3)
        i = (i - 1) / 2;    O(3)
    }

    minHeap->array[i] = minHeapNode; O(1)
}

// Função para construir o min-heap
void buildMinHeap(struct MinHeap* minHeap) {
    int n = minHeap->size - 1; O(3)
                                                  (n−1)/2
    for (int i = (n - 1) / 2; i >= 0; --i)          Σ   O(log_2 n * O(29))
                                                  i=0
        minHeapify(minHeap, i);O(log_2 n * O(29))   O(29)* (n−1)/2 Σ O(log_2 n) =
                                                            i=0
}                                                   O(29)* (n+1)/2 *log(n)

// Função para verificar se o nó é uma folha
int isLeaf(struct MinHeapNode* root) {
    return !(root->left) && !(root->right); O(3)
}

// Função para criar e construir um min-heap com os dados fornecidos
struct MinHeap* createAndBuildMinHeap(char data[], int freq[], int
size) {
    struct MinHeap* minHeap = createMinHeap(size); O(1) + O(d + 11)
    for (int i = 0; i < size; ++i)
                                                          n−1
        minHeap->array[i] = newNode(data[i], freq[i]); Σ (O(1)+O(5))=
                                                          i=0
    minHeap->size = size; O(1)          n−1          n−1
                                        Σ O(1)+ Σ O(5)= O(k)+O(5k)= O(6k)
                                        i=0    i=0
```

```
    buildMinHeap(minHeap); O(3)+O(29)*(n+1)/2 *O(log2 n)
    return minHeap;  O(1)
}

// Função para construir a árvore de Huffman
struct MinHeapNode* buildHuffmanTree(char data[], int freq[], int size)
{
    struct MinHeapNode *left, *right, *top;   O(1)

    // Criar uma fila de prioridade (min-heap)
    struct MinHeap* minHeap = createAndBuildMinHeap(data, freq, size);
                O(1)+ O(17)+O(29)*(n+1)/2 *O(log2 n)+O(6k)+O(d)

    // Repetir até que o tamanho da fila seja 1
    while (!isSizeOne(minHeap)) {  O(1)
        left = extractMin(minHeap);  O(log2 n) * O(29)+ O(4)
        right = extractMin(minHeap); O(log2 n) * O(29)+ O(4)

        // Criar um novo nó interno com a soma das frequências
        top = newNode('$', left->freq + right->freq);  O(1)
        top->left = left;           O(1)
        top->right = right;          O(1)

        insertMinHeap(minHeap, top); O(log2 n)+ O(10)
    }

    return extractMin(minHeap); //A raiz da árvore O(log2 n) * O(29)+
O(4)

}


// Função para contar a frequência dos caracteres em uma string
void calculateFrequencies(char* str, char* data, int* freq, int* size)
{
    int ascii[256] = {0}; O(1)
    int str_len = strlen(str); O(a)    número de caracteres no content
    for (int i = 0; i < str_len; i++) {
                                          n-1
        ascii[(int)str[i]]++;             Σ O(1) = 1(n-1+1-0) =
                                          i=0
O(a)
```

```
    }

    *size = 0;   O(1)
    for (int i = 0; i < 256; i++) {
        if (ascii[i] > 0) {
            data[*size] = (char)i;
            freq[*size] = ascii[i];
            (*size)++;
        }
    }
}

// Função para codificar um arquivo HTML e retornar a árvore
struct MinHeapNode* HuffmanCodesFromHTML(char* filename, char data[],
int freq[], int* size) {
    // Abrir arquivo HTML
    FILE *file = fopen(filename, "r"); O(1)
    if (!file) { O(1)
        printf("Erro ao abrir o arquivo.\n");  O(1)
        return NULL;   O(1)
    }

    // Ler o conteúdo do arquivo
    fseek(file, 0, SEEK_END); O(1)
    long fsize = ftell(file); O(1)      #tamanho do arquivo em bytes
    fseek(file, 0, SEEK_SET); O(1)

    char *content = (char*)malloc(fsize + 1);  O(m+1)
    fread(content, 1, fsize, file);   O(m)
    content[fsize] = '\0';  O(1)
    fclose(file);     O(1)

    printf("Conteúdo do arquivo HTML:\n%s\n", content); O(1)

    // Calcular frequências dos caracteres
    calculateFrequencies(content, data, freq, size); O(2a + 770)

    // Construir a árvore de Huffman
    struct MinHeapNode* root = buildHuffmanTree(data, freq, *size);
```

$$\sum_{i=0}^{255} (O(1)+O(1)+O(1)) = \sum_{i=0}^{255} O(1) + \sum_{i=0}^{255} O(1) + \sum_{i=0}^{255} O(1)$$

$$= 256 + 256 + 256 = O(768)$$

```
   O(1)+ O(29)*(n+1)/2 *O(log2 n)+ 3*O(log2 n)*O(29)+O(log2
n)+O(6k)+O(d)+O(45)

    free(content); O(1)
    return root; O(1)
}

// Função para gerar o arquivo DOT da árvore de Huffman
void generateDot(struct MinHeapNode* root, FILE* file) {
    if (root == NULL) O(1)
        return; O(1)

    // Se for uma folha, mostra o caractere
    if (isLeaf(root)) { O(3)
        fprintf(file, "    \"%c\" [label=\"%c\\n%d\"];\n", root->data,
root->data, root->freq); O(1)
    } else {
        fprintf(file, "    \"%d\" [label=\"%d\"];\n", root->freq,
root->freq); O(1)
    }

    // Se o nó tiver filho esquerdo, conecta e gera para o filho
esquerdo
    if (root->left) { O(1)
        if (isLeaf(root->left)) { O(3)
            fprintf(file, "    \"%d\" -> \"%c\" [label=\"0\"];\n",
root->freq, root->left->data); O(1)
        } else {
            fprintf(file, "    \"%d\" -> \"%d\" [label=\"0\"];\n",
root->freq, root->left->freq); O(1)
        }
        generateDot(root->left, file); T(L)
    }

    // Se o nó tiver filho direito, conecta e gera para o filho direito
    if (root->right) { O(1)
        if (isLeaf(root->right)) { O(3)
            fprintf(file, "    \"%d\" -> \"%c\" [label=\"1\"];\n",
root->freq, root->right->data); O(1)
        } else {
            fprintf(file, "    \"%d\" -> \"%d\" [label=\"1\"];\n",
root->freq, root->right->freq); O(1)
        }
```

```
        generateDot(root->right, file); T(R)
    }
}

// Função para iniciar o processo de geração de arquivo DOT para a
árvore de Huffman
void writeDotFile(struct MinHeapNode* root) {
    FILE* file = fopen("huffman_tree.dot", "w");O(1)
    if (!file) {O(1)
        printf("Erro ao criar arquivo DOT.\n"); O(1)
        return;O(1)
    }

    fprintf(file, "digraph HuffmanTree {\n"); O(1)
    generateDot(root, file); 2*O(15)*n-O(15)
    fprintf(file, "}\n"); O(1)

    fclose(file);  O(1)
    printf("Arquivo DOT gerado com sucesso!\n"); O(1)
}

// Função principal
int main() {
    char data[256];  O(1)
    int freq[256];  O(1)
    int size;      O(1)

    struct MinHeapNode* root = HuffmanCodesFromHTML("pagina.html",
data, freq, &size);
O(1)+ O(29)*$\frac{n+1}{2}$ *O($log_2$ n)+ 3*O($log_2$ n)*O(29)+O($log_2$
n)+2*O(m)+O(6k)+O(d)+2*O(a)+O(829)

    if (root != NULL) { O(1)
        writeDotFile(root);  2*O(15)*n-O(9)
    }

    return 0;O(1)
}
```

Resultado extraído do código:

O(29)*$O(\frac{n+1}{2})$ *O(log_2 n)+ 3*O(log_2 n)*O(29)+O(log_2 n)+O(30)*O(n)+2*O(m)+O(6k)+O(d)+2*O(a)+O(826)

`T(n) =` **29*$\frac{n+1}{2}$ *log_2 n+ 3*log_2 n*29+log_2 n +2*15*n+2*m+6k+d+2*a+826**

`T(n) =` **29*$\frac{n+1}{2}$ *log_2 n+ 87log_2 n+log_2 n +30n+2*m+6k+d+2*a+826**

`T(n) =` **29*$\frac{n+1}{2}$ *log_2 n+ 88log_2 n +30n+2*m+6k+d+2*a+826**

`T(n) =` $\frac{29n+29}{2}$ ***log_2 n+ 88log_2 n +30n+2*m+6k+d+2*a+826**

Função de Custo aproximada:

`T(n) =` $\frac{29n}{2}log_2$ **n+**$\frac{29}{2}log_2$ **n+ 88**log_2 **n +30n+2*m+6k+d+2*a+826**

variaveis:
n: mainHeap->size - número de elementos no min-heap
m: fsize - tamanho total do arquivo HTML em bytes
k: size - número de caracteres ÚNICOS.
d:minHeap->capacity - define quantos elementos na fila de prioridade o minHeap será capaz de armazenar
a: str_len - número de caracteres percorre todo o content onde possui o conteúdo do file.

n: mainHeap->size - 93
m: fsize - 25172
k: size - 93
d:minHeap->capacity - 93
a: str_len - 25172

observando as variáveis iguais vamos simplificar a função de custo, reduzindo o número de variáveis:

Função de Custo aproximada:

$T(n,m) = \frac{29n}{2} log_2 n + \frac{29}{2} log_2 n + 88 log_2 n + 30n + 2*m + 6n + n + 2*m + 826$

$T(n,m) = \frac{29n}{2} log_2 n + \frac{29}{2} log_2 n + 88 log_2 n + 37n + 4*m + 826$

agora irei fazer um teste:
pagina_testada.html

n: 93
m: 25172

$T(93, 25172) = \frac{29*93}{2} log_2 93 + \frac{29}{2} log_2 93 + 88 log_2 93 + 37*93 + 4*25172 + 826$

$T(93, 25172) = 1.348,5 * log_2 93 + 14,5 * log_2 93 + 88 * log_2 93 + 3.441 + 100.688 + 826$

$log_2 93$ **é aproximadamente 6.5**

$T(93, 25172) = 1.348,5 * 6,5 + 14,5 * 6,5 + 88*6,5 + 3.441 + 100.688 + 826$

$T(93, 25172) = 8.765,25 + 94,25 + 572 + 3.441 + 100.688 + 826$

$T(93, 25172)$ **é aproximadamente 114.386 instruções**

1000 instruções - 333 µs
114.386 instruções - x

x=333*114386/1000≈38.090 µs

convertendo para segundos : 0,03809 s

tempo de execução computado pelo algoritmo (computador 1**): 0,039000 s**

computador 2: 0.009000s

benstombados.html:

n: 96

m: 2637040

$T(96,2637040) = \frac{29*96}{2} log_2 96 + \frac{29}{2} log_2 96 + 88 log_2 96 + 37*96 + 4*2637040 + 826$

$T(96,2637040) = 1.392 log_2 96 + 14,5 log_2 96 + 88 log_2 96 + 37*96 + 4*2637040 + 826$

$log_2 96$ **é aproximadamente** 6,58

$1.392 * 6,58 + 14,5 * 6,58 + 88 * 6,58 + 37*96 + 4*2637040 + 826$

$T(96,2637040)$ **é aproximadamente** 10.562.371 **instruções**

1000 instruções - 333 µs

10.562.371 **instruções - x**

x=333*10.562.371**/1000≈3.517.269µs**

convertendo para segundos : 3,517269 s

tempo de execução computado pelo algoritmo (computador1)≈ 2,4300 s

tempo computador2 = 0.426000s

biblia.html:

n= 95

m: 658893

`T(95,`658893`) =` $\frac{29*95}{2}log_2$ **95+**$\frac{29}{2}log_2$ **95+ 88**log_2 **95+ 37*95+4***658893**+826**

`T(95,`658893`) =` $1.377,5log_2$ **96+**$14,5log_2$ **96+ 88**log_2 **96+ 37*95+4***658893**+826**

log_2 **95 é aproximadamente** 6,56

$1.377,5$ *6,56**+**$14,5$ *6,56**+ 88***6,56**+ 37*95+4***658893**+826**

`T(95,`658893`) é aproximadamente` 2.649.621 i**nstruções**

1000 instruções - 333 µs

2.649.621 **instruções - x**

x=333*10.562.371**/1000≈882.323µs**

convertendo para segundos = 0,882323 s

tempo no computador1 ≈0.40000s

tempo no computador2≈0.050000s

`T(n,m) =` $\frac{29n}{2}log_2$ **n+**$\frac{29}{2}log_2$ **n+ 88**log_2 **n +37n+4*m+826**

Complexidade: O(n*log n)