



# BST AND HEAP

Huffman coding and decoding



# ESTRUTURA DO PROGRAMA

01

## Estruturas de Dados

- MinHeapNode
- MinHeap

02

## Funções Auxiliares

- newNode()
- createMinHeap()
- swapMinHeapNode()
- minHeapify()
- isLeaf()

03

## Algoritmos Principais

- buildMinHeap()
- extractMin()
- insertMinHeap()
- buildHuffmanTree()

```
8 // Estrutura para o nó da árvore de Huffman
9 struct MinHeapNode {
10     char data;
11     unsigned freq;
12     struct MinHeapNode *left, *right;
13 };
14
15 // Estrutura para a fila de prioridade (min-heap)
16 struct MinHeap {
17     unsigned size;
18     unsigned capacity;
19     struct MinHeapNode** array;
20 };
```

# ESTRUTURA DO PROGRAMA

01

## Estruturas de Dados

- MinHeapNode
- MinHeap

02

## Funções Auxiliares

- newNode()
- createMinHeap()
- swapMinHeapNode()
- minHeapify()
- isLeaf()

03

## Algoritmos Principais

- buildMinHeap()
- extractMin()
- insertMinHeap()
- buildHuffmanTree()

```
104 // Função para criar e construir um min-heap com os dados fornecidos
105 struct MinHeap* createAndBuildMinHeap(char data[], int freq[], int size) {
106     struct MinHeap* minHeap = createMinHeap(size);
107     for (int i = 0; i < size; ++i)
108         minHeap->array[i] = newNode(data[i], freq[i]);
109     minHeap->size = size;
110     buildMinHeap(minHeap);
111     return minHeap;
112 }
```



# ESTRUTURA DO PROGRAMA

01

## Estruturas de Dados

- MinHeapNode
- MinHeap

02

## Funções Auxiliares

- newNode()
- createMinHeap()
- swapMinHeapNode()
- minHeapify()
- isLeaf()

03

## Algoritmos Principais

- buildMinHeap()
- extractMin()
- insertMinHeap()
- buildHuffmanTree()

```
114 // Função para construir a árvore de Huffman
115 struct MinHeapNode* buildHuffmanTree(char data[], int freq[], int size) {
116     struct MinHeapNode *left, *right, *top;
117
118     // Criar uma fila de prioridade (min-heap)
119     struct MinHeap* minHeap = createAndBuildMinHeap(data, freq, size);
120
121     // Repetir até que o tamanho da fila seja 1
122     while (!isSizeOne(minHeap)) {
123         left = extractMin(minHeap);
124         right = extractMin(minHeap);
125
126         // Criar um novo nó interno com a soma das frequências
127         top = newNode('$', left->freq + right->freq);
128         top->left = left;
129         top->right = right;
130
131         insertMinHeap(minHeap, top);
132     }
133
134     return extractMin(minHeap); // A raiz da árvore
135 }
```

# MANIPULAÇÃO DE ARQUIVOS E VISUALIZAÇÃO

01

## HuffmanCodesFromHTML()

Lê um arquivo HTML, calcula a frequência de cada caractere e constrói a árvore de Huffman.

02

## writeDotFile()

Gera a árvore de Huffman no formato DOT, um formato de texto com estrutura e sintaxe definida.

03

## GraphViz

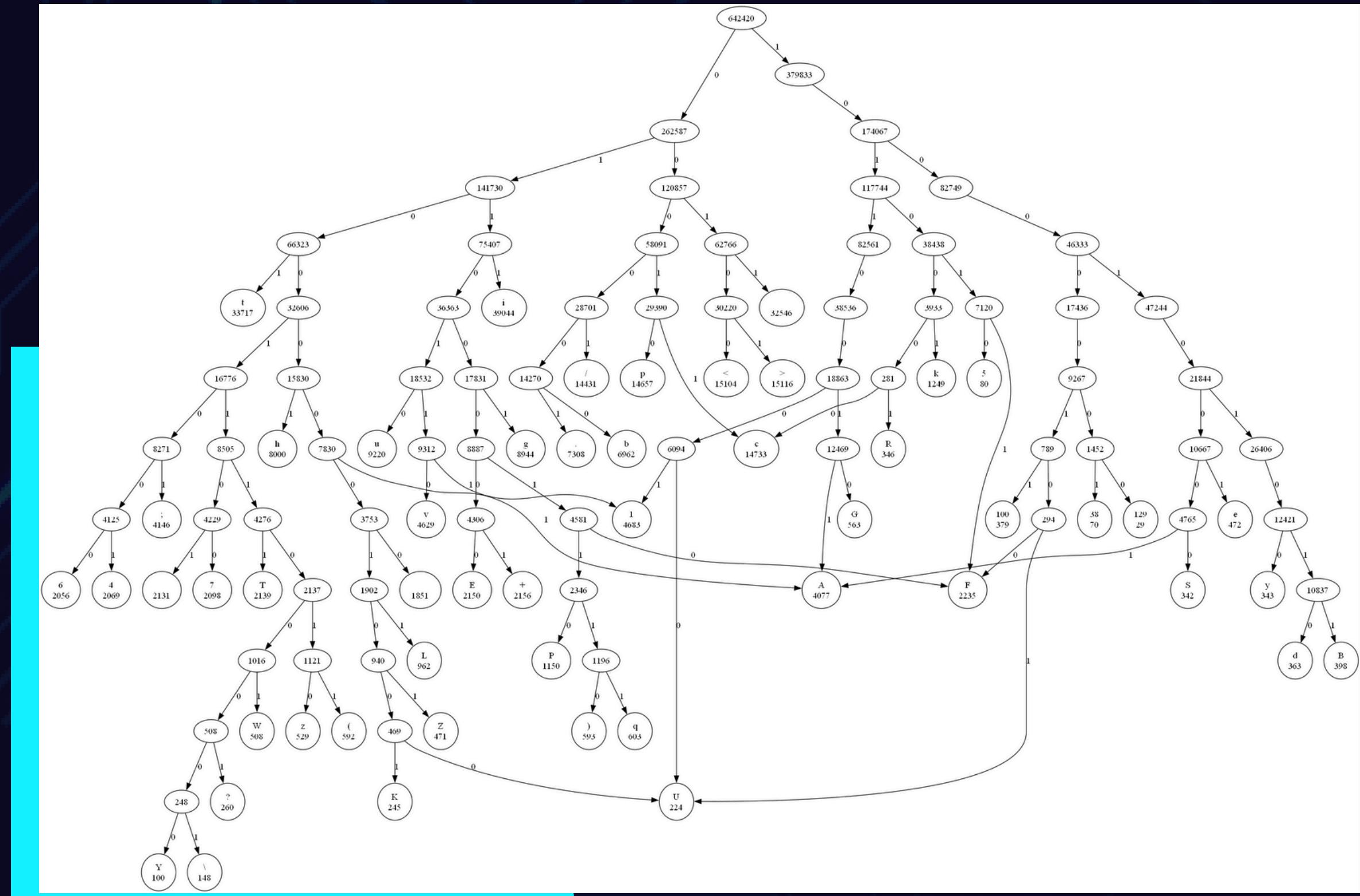
O arquivo DOT foi renderizado usando o Graphviz para produzir uma representação gráfica da árvore de Huffman.

```
186 // Função para gerar o arquivo DOT da árvore de Huffman
187 void generateDot(struct MinHeapNode* root, FILE* file) {
188     if (root == NULL)
189         return;
190
191     // Se for uma folha, mostre o caractere
192     if (isLeaf(root)) {
193         fprintf(file, "    \"%c\" [label=\"%c\\n%d\"];\n", root->data, root->data, root->freq);
194     } else {
195         fprintf(file, "    \"%d\" [label=\"%d\"];\n", root->freq, root->freq);
196     }
197
198     // Se o nó tiver filho esquerdo, conecte e gere para o filho esquerdo
199     if (root->left) {
200         if (isLeaf(root->left)) {
201             fprintf(file, "    \"%d\" -> \"%c\" [label=\"0\"];\n", root->freq, root->left->data);
202         } else {
203             fprintf(file, "    \"%d\" -> \"%d\" [label=\"0\"];\n", root->freq, root->left->freq);
204         }
205         generateDot(root->left, file);
206     }
207
208     // Se o nó tiver filho direito, conecte e gere para o filho direito
209     if (root->right) {
210         if (isLeaf(root->right)) {
211             fprintf(file, "    \"%d\" -> \"%c\" [label=\"1\"];\n", root->freq, root->right->data);
212         } else {
213             fprintf(file, "    \"%d\" -> \"%d\" [label=\"1\"];\n", root->freq, root->right->freq);
214         }
215         generateDot(root->right, file);
216     }
217 }
```

[Home](#)[Service](#)[About Us](#)[Contact](#)

# IMPLEMENTAÇÃO E TESTES

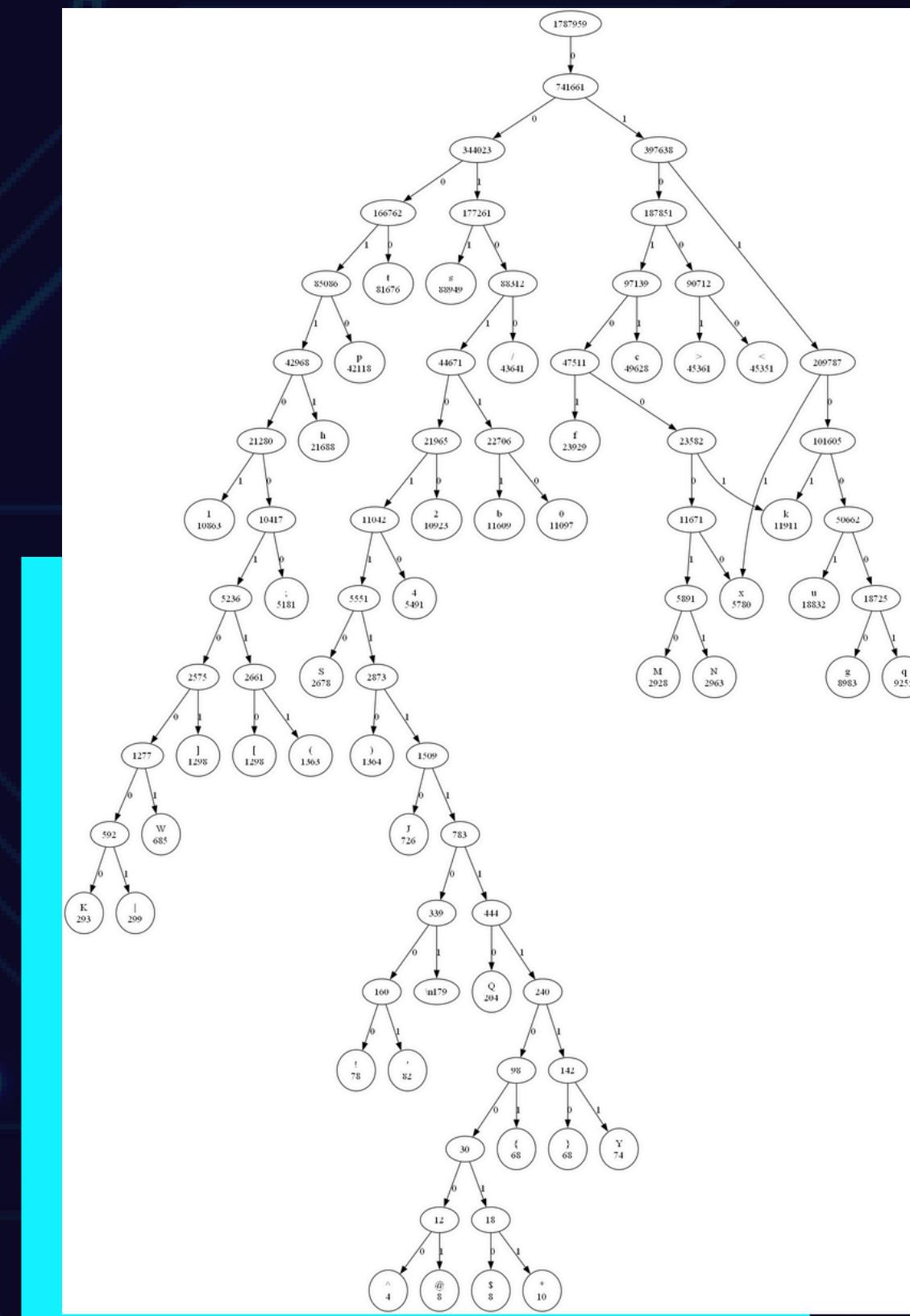
Uma das primeiras implementações do algoritmo de Huffman foi realizada utilizando a página HTML da Wikipédia sobre a Bíblia, disponível em ["https://pt.wikipedia.org/wiki/Bíblia"](https://pt.wikipedia.org/wiki/Bíblia).





# IMPLEMENTAÇÃO E TESTES

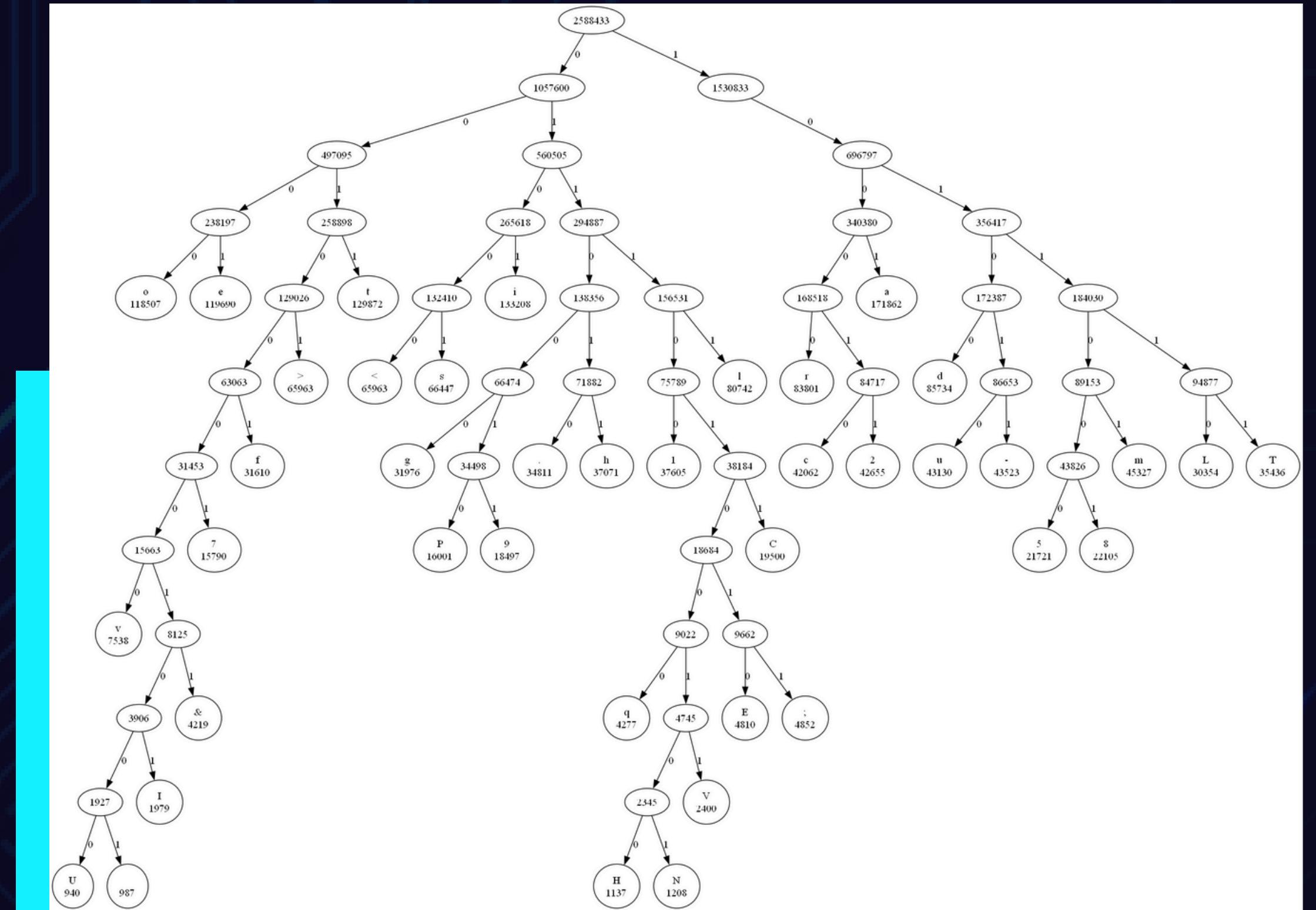
A terceira implementação do algoritmo de Huffman foi realizada com a página da Wikipédia sobre a História da Humanidade, disponível em [https://pt.wikipedia.org/wiki/História\\_da\\_humanidade](https://pt.wikipedia.org/wiki/História_da_humanidade). Uma página, com tamanho de 612 645 bytes.





# IMPLEMENTAÇÃO E TESTES

O último teste realizado consistiu na aplicação do algoritmo de Huffman à página da Wikipédia sobre a Lista de Bens Tombados na Cidade de São Paulo, disponível em [https://pt.wikipedia.org/wiki/Lista\\_de\\_bens\\_tombados\\_na\\_cidade\\_de\\_S%C3%A3o\\_Paulo](https://pt.wikipedia.org/wiki/Lista_de_bens_tombados_na_cidade_de_S%C3%A3o_Paulo). Esta página, identificada como a maior do site da Wikipédia em português para o Brasil, com tamanho de 681 874 bytes.





# FUNÇÃO DE CUSTO E COMPLEXIDADE

**BST (Binary Search Tree) e heaps são estruturas de dados essenciais em algoritmos como a codificação de Huffman. O BST permite operações de busca, inserção e exclusão em  $O(\log n)$  em média, enquanto o heap oferece inserção e remoção em  $O(\log n)$ .**

A construção da árvore de Huffman utiliza um min-heap, e o custo dessa fase é  $O(n \log n)$ , onde  $n$  é o número de caracteres únicos. A complexidade total do algoritmo é  $O(n \log n)$ , sendo eficiente para compressão de dados.

```
t("Conteúdo do arquivo HTML:\n%s\n", content);
    calculateFrequencies(content, data, freq, size);
    buildHeap(freq);
    HuffmanNode* root = buildHuffmanTree(freq);
    HuffmanNode* current = root;
    while (current != NULL) {
        cout << current->character;
        current = current->left;
    }
    cout << endl;
    FILE *file = fopen("huffman.dot", "w");
    if (file == NULL) {
        cout << "Error opening file" << endl;
        exit(1);
    }
    printTree(file, root);
    fclose(file);
    cout << "File huffman.dot created successfully" << endl;
```



# FUNÇÃO DE CUSTO APROXIMADA

Função de Custo aproximada:

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

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



# TESTE DA FUNÇÃO DE CUSTO

n: 93

m: 25172

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

$$+ 37*93 + 4*25172 + 826$$

$$T(93, 25172) = 1.348,5 * \log_2 93 + 14,5 * \log_2 93 + 88 * \log_2 93$$

$$+ 3.441 + 100.688 + 826$$

# TESTE DA FUNÇÃO DE CUSTO

[Home](#)[Service](#)[About Us](#)[Contact](#)

$\log_2 93$  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  $\mu$ s

114.386 instruções - x

$$x = 333 * 114386 / 1000 \approx 38.090 \mu\text{s}$$

convertendo para segundos : 0,03809 s



# TESTE DA FUNÇÃO DE CUSTO

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

PORTS

```
$('.carousel').carousel({  
    interval: 5000 //changes the speed  
})  
</script>
```

```
</body>  
</html>
```

Tempo para construir o heap e a árvore de Huffman: 0.039000 segundos  
Arquivo DOT gerado com sucesso!

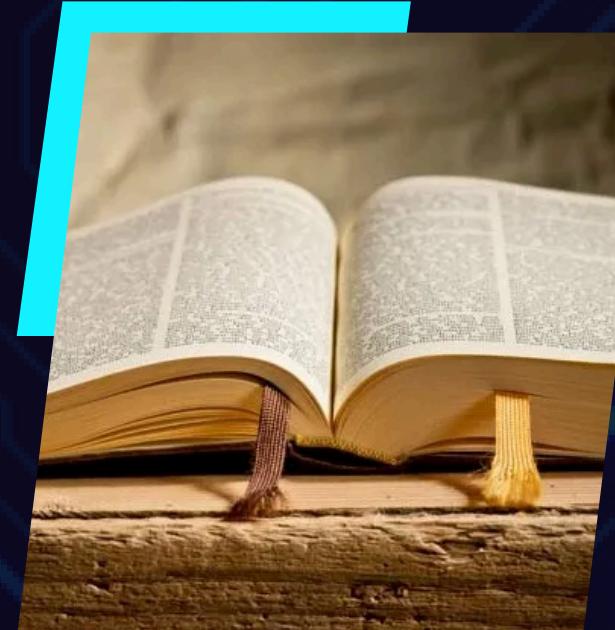
PS C:\Users\felip\Documents\Faculdade\8º Semestre\Análise de Algoritmos\ProjetoFinal> █

# RESULTADOS DOS TESTES



**Tartarugas Ninjas**

0.223 segundos



**Bíblia**

0.456 segundos



**História da Humanidade**

1.234 segundos



**Elenco 2017 do Flamengo**

1.876 segundos



**Bens Tombados da Cidade de São Paulo**

2.472 segundos

# APLICAÇÃO PRÁTICA



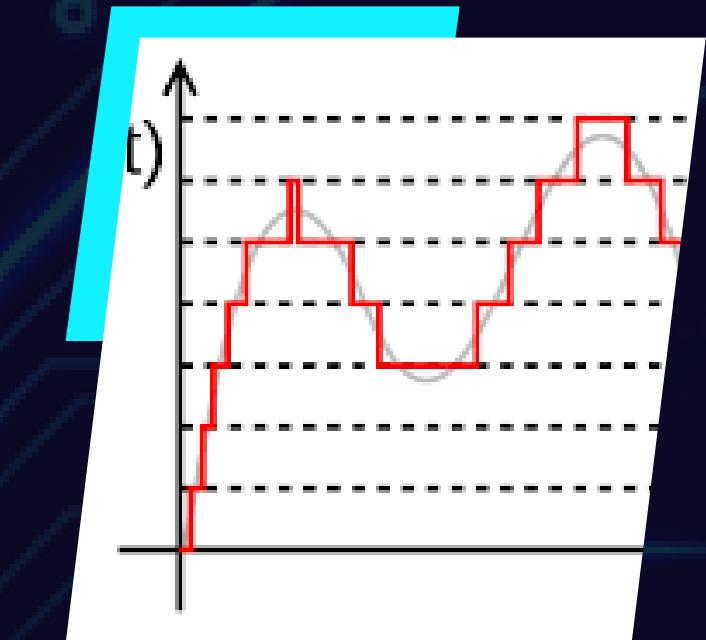
## Conversão da Imagens

Uma imagem é convertida em diferentes blocos, onde cada bloco representa as cores ou luminosidade.



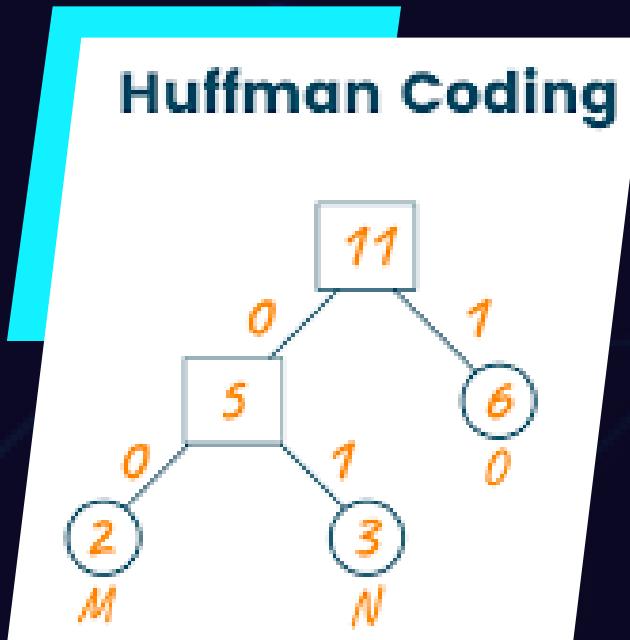
## Transformada Discreta do Cosseno (DCT)

O JPEG usa a DCT para transformar os dados de espaço de cores em coeficientes de frequência.



## Quantização

O processo de quantização reduz a precisão dos coeficientes, descartando as informações de alta frequência (menos perceptíveis ao olho humano).



## Codificação Huffman

Finalmente, os coeficientes quantizados são codificados usando o algoritmo de Huffman.



# OBRIGADO