# Algoritmo de Huffman

**Allan de Lima da Silva**

[1]Universidade do Vale do Itajaí (UNIVALI)

allanlima@edu.univali.br

## 1. Enunciado

O objetivo do trabalho é implementar o Algoritmo de Huffman com suporte a "codificação"

(comprimir) e "decodificação" (expandir) de frases.

O programa a ser desenvolvido deve:

1. Codificar uma frase ASCII em Huffman (50%): Dado uma string de entrada (frase):
    a. Gerar a Tabela de Símbolos com a codificação de cada caractere;
    b. Gerar a Árvore de Huffman;
    c. Gerar a Codificação de Huffman;
2. Decodificar uma frase Huffman para ASCII (50%): Dada a tabela de símbolos com a codificação de caracteres e um texto codificado (no padrão de Huffman)
    a. Gerar a Árvore de Huffman; e
    b. Apresentar a frase original.

## 2. Desenvolvimento

O projeto foi desenvolvido utilizando a linguagem c++ dentro do software CodeBlocks onde para melhor organizar o projeto, foram usados 2 arquivos .h juntamente do arquivo principal main.cpp e para a implementação das listas necessárias para execução, foi utilizada a estrutura de lista encadeada simples implementada diversas vezes durante o percorrer da presente matéria.

A primeira execução do projeto, é identificar a frase passada pelo usuário por uma string, onde cada letra desta frase será inserida em uma lista encadeada, como é visto na figura 1.

```
string frase = "FRASE";
cout<<"FRASE: "<<frase<<endl;
char letra;

for (int i=0; i<frase.length(); ++i)
{
     letra = frase.at(i);
     insere_dado_inicio_huff(lista1,letra);
}
```

**Figura 1 Inserção de cada letra na lista encadeada**

Dentro da função “insere_dado_inicio_huff()” é inserido um novo dado dentro da lista, esse dado possuí as seguintes variáveis: Um char ‘dado’, que é utilizado para armazenar a letra, um int ‘freq’, para armazenar a frequência com que a letra é encontrada na frase, e referências para agir como uma lista encadeada simples (*próximo) e também como um no de árvore (*esquerda, *direita).

Caso essa letra já existir na lista, apenas é aumentada a frequência deste dado, sem que seja criado um novo.

Após inserida todas as letras dentro da lista, é impresso em console a lista duas vezes, onde a primeira é mostrada a lista original e depois a lista é ordenada de maneira crescente pela frequência, como pode ser visto na figura 2.

```
void ordernar_freq_Huff( TlistaHuff &l){
    bool ordenado = false;
    noHuffman *aux = new noHuffman;
    cout<<endl<<"\t||ORDENANDO LISTA DE MANEIRA\n\t  CRESCENTE PELA FREQUENCIA||"<<endl;
    while(ordenado!=true){
        ordenado = true;
        for (noHuffman*nav = l.inicio; nav!=NULL; nav = nav->prox){
            for(noHuffman*nav2 = nav->prox; nav2!=NULL;nav2 = nav2->prox){
                if(nav2->freq < nav->freq){
                        aux->dado = nav2->dado;
                        aux->freq = nav2->freq;

                        nav2->dado = nav->dado;
                        nav2->freq = nav->freq;

                        nav->dado = aux->dado;
                        nav->freq = aux->freq;
                        ordenado = false;
                }
            }
        }
    }
}
```

**Figura 2 Função de ordenação por frequência**

Com a lista ordenada, é iniciada a principal função do projeto ( arvore_Huff() ) onde é gerada a árvore. as principais operações desta função são:

A criação de dois nós e um navegador que servem de auxiliares para as próximas operações executadas.

São retirados os dois primeiros nós (dados) da lista e armazenados dentro dos auxiliares, é criado um nó, que será o “pai” destes dois nós retirados e terá como frequência a soma dos mesmos.

Usando o navegador da lista, é inserido esse novo nó “pai” sempre buscando manter a lista ainda sim ordenada de maneiras crescente, com a utilização do laço de repetição que pode ser visto na figura 3.

```
///INICIA NAVEGADOR PARA INSERIR O PAI
nav=l.inicio;
while(true){
    ///INSERE ELE DE ACORDO COM A FREQUENCIA

    ///ULTIMO ELEMENTO DA LISTA
    if(nav==nullptr){
        l.inicio = novo;
        novo->prox = nullptr;
        break;

    ///INSERÇÃO DO ELEMENTO COM MAIOR VALOR
    } else if (nav->prox == nullptr){
        novo->prox = nav->prox;
        nav->prox = novo;
        imprimir_lista_Huff(l);
        break;

    ///INSERÇÃO NA POSIÇÃO ANTERIOR AO VALOR MAIOR
    }else if(nav->prox->freq >= novo->freq){
        novo->prox = nav->prox;
        nav->prox = novo;
        imprimir_lista_Huff(l);
        break;
    }
    nav=nav->prox;
}
```

**Figura 3 Inserção do novo nó dentro da lista**

Essas operações são repetidas até que a lista possua apenas um nó, que será a raiz da árvore de Huffman.

Com a árvore concluída, é iniciada uma lista que servirá para implementação da tabela, que possui as seguintes variáveis em cada um de seus dados: Um char ‘dado’, um int ‘frequencia’, uma string ‘codigo’ que servirá para armazenar a codificação huffman da letra e por fim uma referência para próxima posição da lista encadeada.

Para preencher a tabela, é utilizada uma função recursiva que percorrerá toda a árvore. Toda vez que a função encontrar um nó com conteúdo na árvore, ele é adicionado na tabela como um novo elemento. A string ‘codigo’ é gerada a partir da

navegação dentro da árvore, onde o código ira concatenar um ‘0’ ou ‘1’ dependendo da direção pela qual a seguir os nós, essa função pode ser vista na figura 4

```
void preenche_tabela(tabelaSimbolos &t,noHuffman *no,string codigo){

    ///NÓ POSSUI CONTEÚDO
    if(no->dado!= '-'){
        dadoTabela *novo = new dadoTabela;
        novo->dado = no->dado;
        novo->freq = no->freq;
        novo->cod = codigo;
        insere_inicio_tabela(t,novo);
    }

    if(no->esq != nullptr){
        preenche_tabela(t,no->esq,codigo+'0');
    }
    if(no->dir != nullptr){
        preenche_tabela(t,no->dir,codigo+'1');
    }

}
```

**Figura 4 Função para preencher tabela**

Com a tabela criada, novamente é passada todas as letras da frase por uma função para que elas sejam transcritas usando seu código Huffman, figura 5 e 6.

```
for (int i=0;i<frase.length();++i){
    letra = frase.at(i);
    cout<<codificacao_huff(tabela,letra)<<" ";
}
```

**Figura 5 Envio de cada letra para codificação**

```
string codificacao_huff(tabelaSimbolos t,char letra){
    if(t.inicio == nullptr){
        cout<<" ||Tabela vazia|| ";
    }else{
        int i = 0;
        for (dadoTabela*nav = t.inicio; nav!=NULL; nav = nav->prox){
            if(nav->dado == letra){
                return nav->cod;
            }
        }
    }
}
```

**Figura 6 Função de codificação, retornando o código de cada letra enviada**

## 3. Resultados.

Para demonstração da funcionalidade do projeto, foi utilizada a frase: “TRABALHO DE ESTRUTURAS”, na figura 7 pode-se ver também a inicialização da lista, que logo é apresentada de maneira ordenada na figura 8.

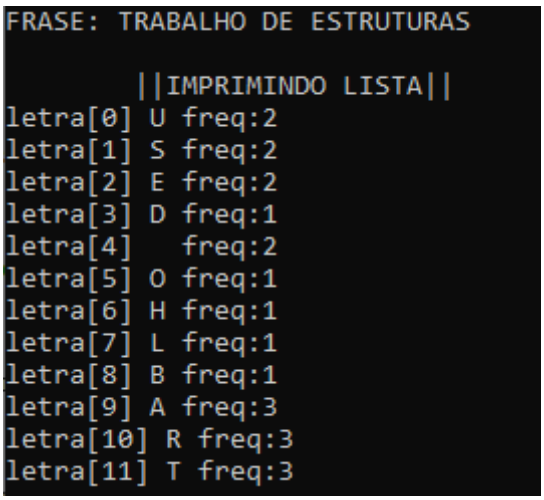


**Figura 7 Inicialiação da lista**

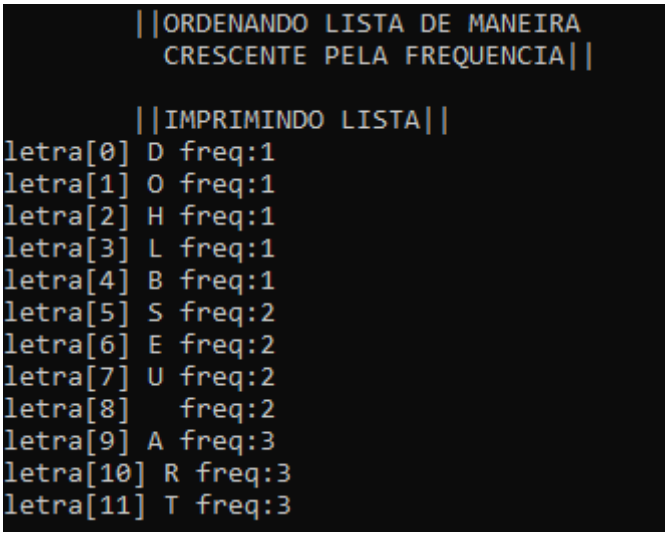


**Figura 8 Lista ordenada**

Com a lista ordenada, é iniciada a construção da árvore, que é apresentada passa a passo no console do usuário para melhor observarmos a alocação dos nós criados, como podemos ver na figura 9 e 10.

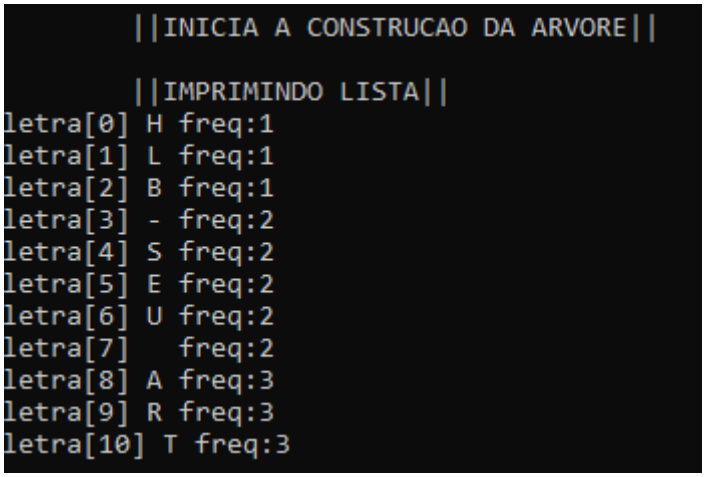


**Figura 9 Inicio da contrução da árvore**


**Figura 10 Execução da construção da árvore.**

Essa execução continua até que a lista possua apenas um elemento, figura 11.

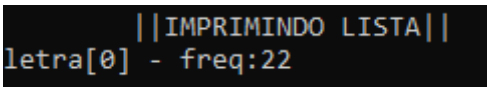


**Figura 11 Lista com apenas um elemento restante**

Com a árvore construída, é feita a impressão pré-fixa da arvore em console, figura 12.

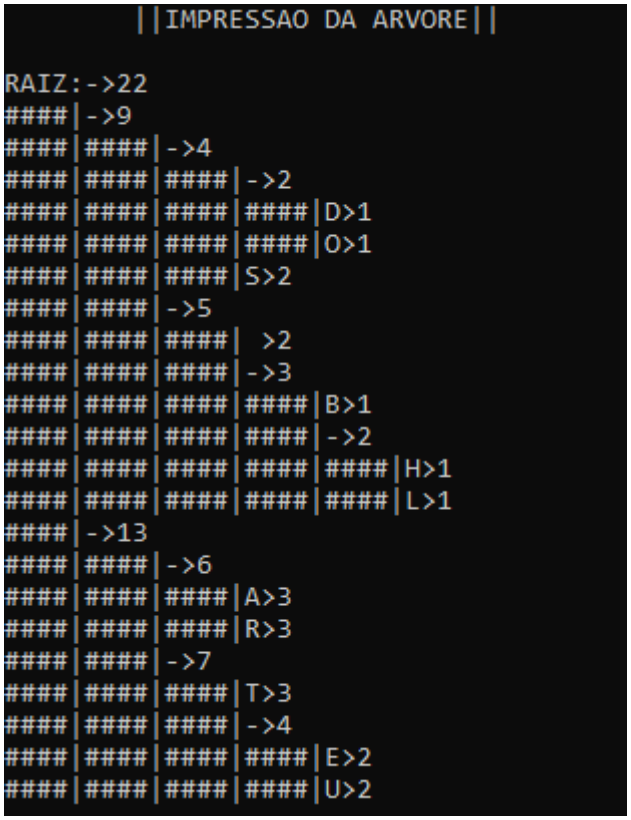


**Figura 12 Impressão da árvore em console**

É feita impressão da tabela construída e após isso, vemos a frase codificada utilizando a tabela, nas figuras 13 e 14.

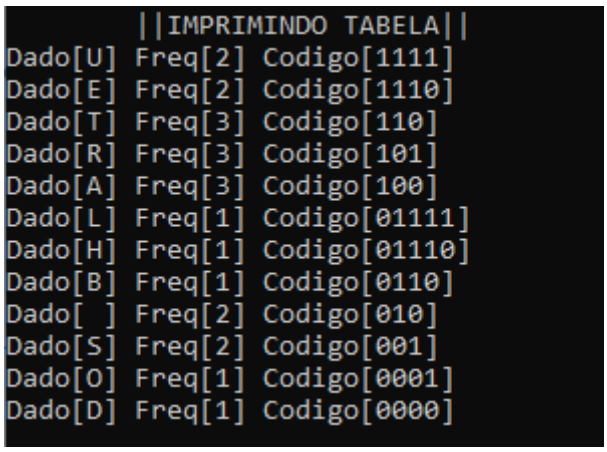


**Figura 13 Impressão da tabela**

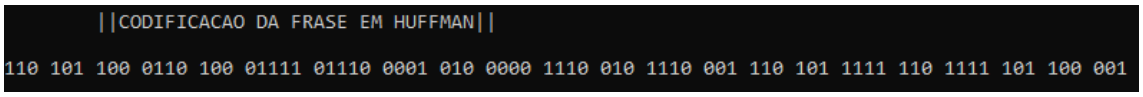


**Figura 14 Codificação da frase**