Aplicação de Árvores: Código de Huffman

SCC0202 - Algoritmos e Estruturas de Dados I

Prof. Fernando V. Paulovich
http://www.icmc.usp.br/~paulovic
paulovic@icmc.usp.br

Instituto de Ciências Matemáticas e de Computação (ICMC)
Universidade de São Paulo (USP)

4 de novembro de 2010

Sumário

Sumário

Introdução

- Com o crescimento da quantidade de dados gerados e transmitidos, compactação desses se torna cada dia mais essencial
  - Armazenamento de dados (imagens médicas)
    - $5000 \times 3000 \times 2 = 30 Mbytes$
  - Transmissão de dados (Internet)
- Um método de compactação bem conhecido, o código de *Huffman*, se baseia em árvores binárias

Introdução

- Em um texto não compactado, um caractere é representado por um byte (ASCII), de forma que todo caractere é representado pelo mesmo número de bits

Caractere	Decimal	Binário
A	65	01000000
B	66	01000001
C	67	01000010
...		
X	88	01011000
Y	89	01011001
Z	90	01011010

Introdução

- Existem diversas formas de se compactar dados, a mais comum é buscar reduzir o número de bits que representam os caracteres mais frequentes

- Seja **E** o caractere mais frequente (em inglês isso é verdade), supondo que ele seja codificado com dois bits, **01**
  - Não é possível codificar todo alfabeto com dois bits: **00**, **01**, **10** e **11**
  - Podemos usar essas quatro combinações para codificar os quatro caracteres mais frequentes?

Introdução

- Nenhum caractere pode ser representado pela mesma combinação de bits que aparece no início de um código mais longo
  - Se **E** é **01** e **X** é **01011000**, não é possível diferenciar um do outro

Introdução

- Nenhum caractere pode ser representado pela mesma combinação de bits que aparece no início de um código mais longo
  - Se **E** é **01** e **X** é **01011000**, não é possível diferenciar um do outro

Regra

- Nenhum código pode ser o prefixo de qualquer outro código

Introdução

- Quando a frequência dos caracteres é conhecida a priori, e o documento segue essa frequência, essa abordagem funciona
- Porém, nem sempre isso é verdade
  - Artigo de jornal X código fonte Java
- Então é preciso fazer uma contagem

Introdução

- Suponha a mensagem "SUSIE SAYS IT IS EASY"

Caractere	Contagem
A	2
E	2
I	3
S	6
T	1
U	1
Y	2
Espaço	4
Avanço de linha	1

Introdução

- Definindo que os caracteres mais frequentes devem ser codificados com um número pequeno de bits, a seguinte decodificação pode ser usada

Caractere	Contagem	Código
A	2	010
E	2	1111
I	3	110
S	6	10
T	1	0110
U	1	01111
Y	2	1110
Espaço	4	00
Avanço de linha	1	01110

Introdução

- Usando essa codificação, “SUSIE SAYS IT IS EASY” seria transformada em

10 01111 10 110 1111 00 10 010 1110 10 00 110 0110 0110 00 110 10 00 1111 010 10 1110 01110

Introdução

- Usando essa codificação, "SUSIE SAYS IT IS EASY" seria transformada em

10 01111 10 110 1111 00 10 010 1110 10 00 110 0110 0110 00 110 10 00 1111 010 10 1110 01110

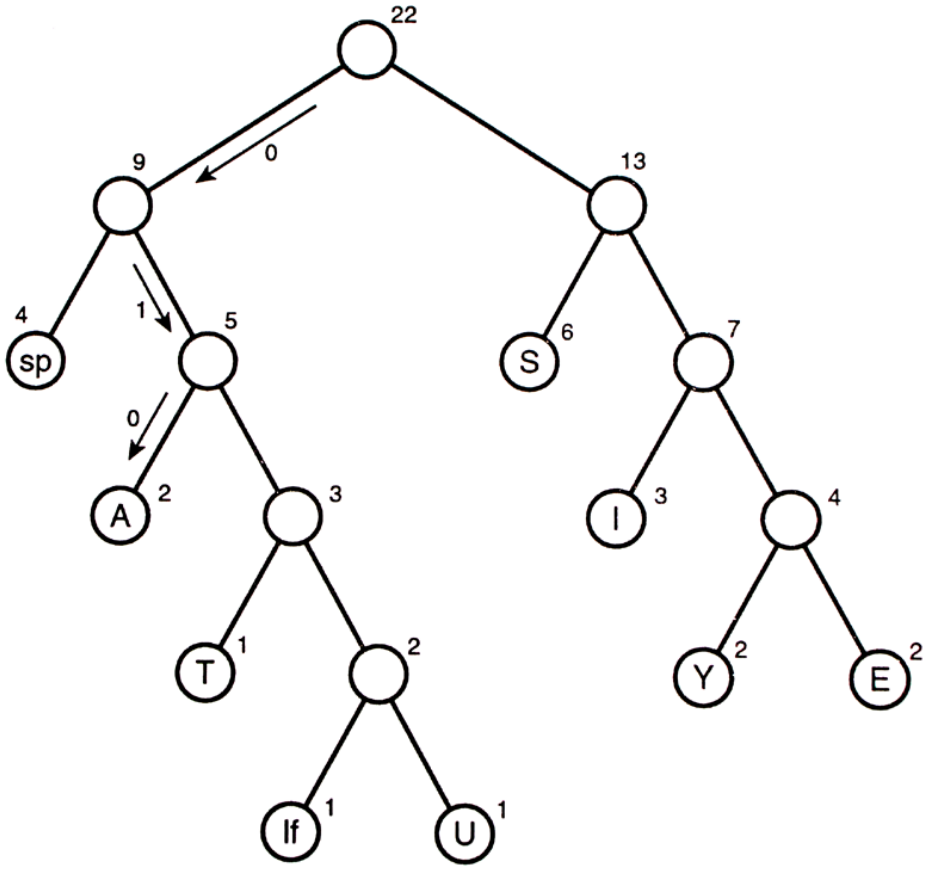
Taxa de Compactação

$$T_c = 1 - \frac{68}{22 \times 8} = 1 - \frac{68}{196} = 0,614$$

Sumário

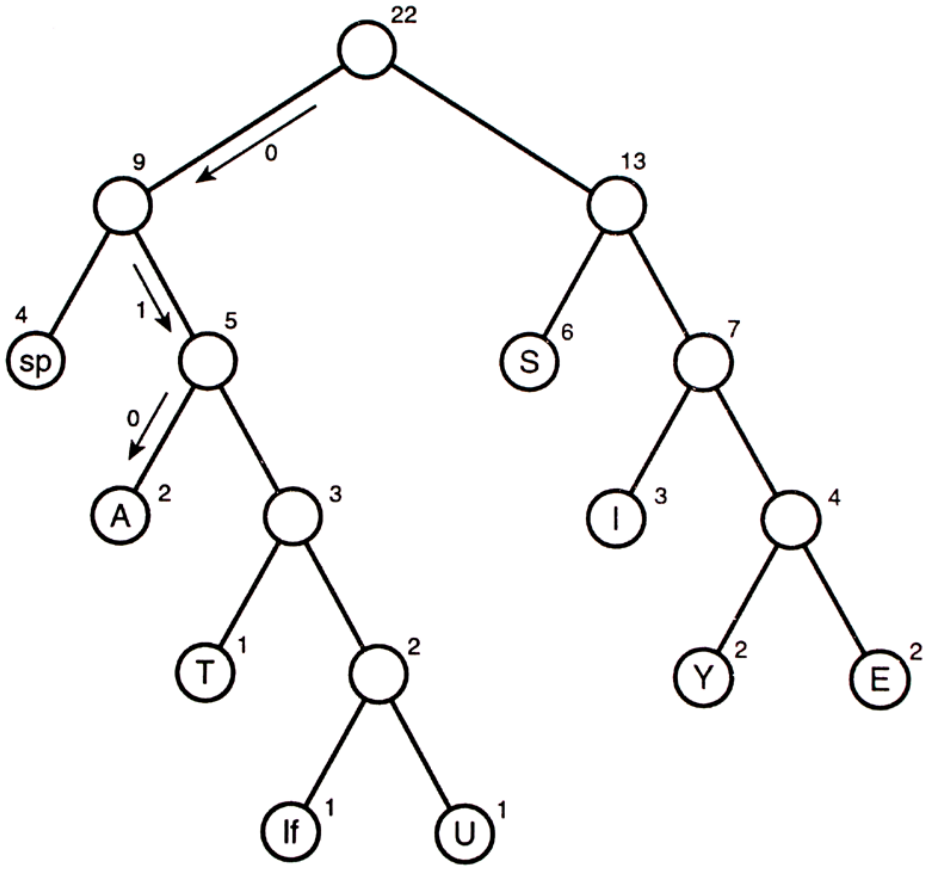
Decodificando com a Árvore de Huffman

- Antes de vermos como codificar, vamos ver um processo mais fácil: a **decodificação**
- Para se decodificar uma dada cadeia de bits e obtermos os caracteres originais usamos um tipo de árvore binária conhecida como **árvore de Huffman**



Decodificando com a Árvore de Huffman

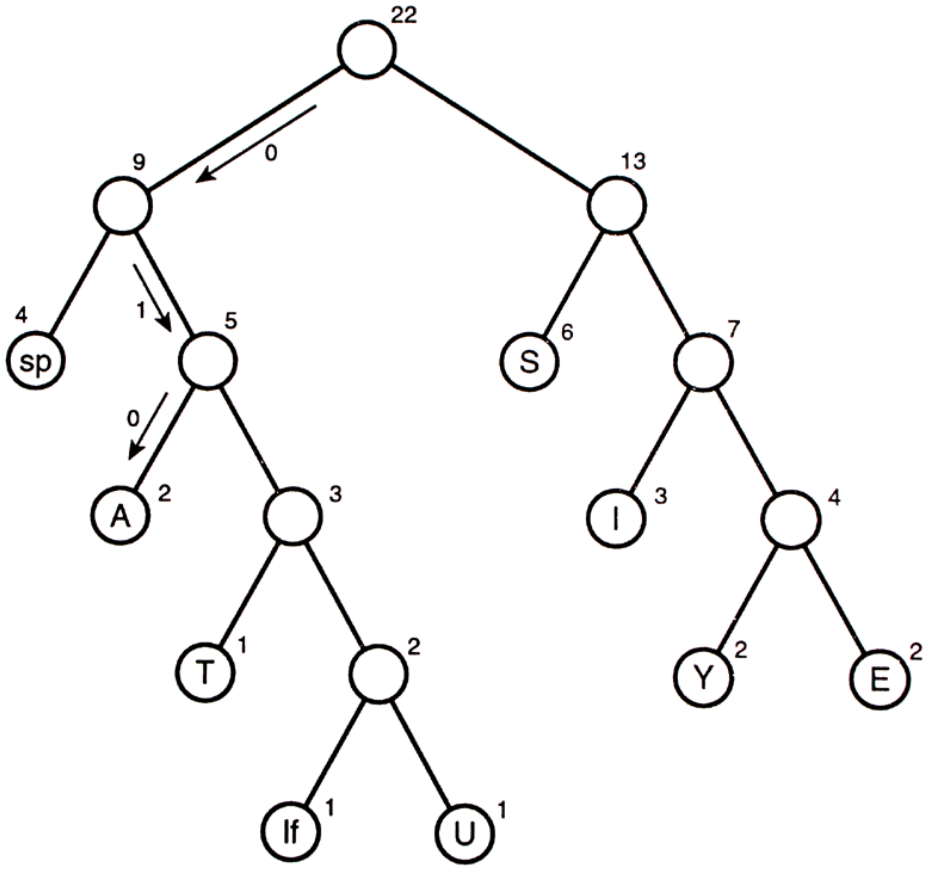
- Os **caracteres** das mensagem aparecem na árvore como **folhas**
- Quanto mais **alta a frequência** de um termo, mais **alto** ele aparecerá na **árvore**
- Números representam as frequências



Decodificando com a Árvore de Huffman

- Decodificando: para cada símbolo de entrada (bit)
  - Se aparecer um bit 0, desce para a esquerda
  - Se aparecer um bit 1, desce para a direita
- Atingiu uma folha, achou a codificação
- Repete o processo para o próximo símbolo de entrada

- A = 010



Criando a Árvore de Huffman I

- Existem diversas formas de se criar a árvore de Huffman, aqui vamos usar a abordagem mais comum

Inicialização

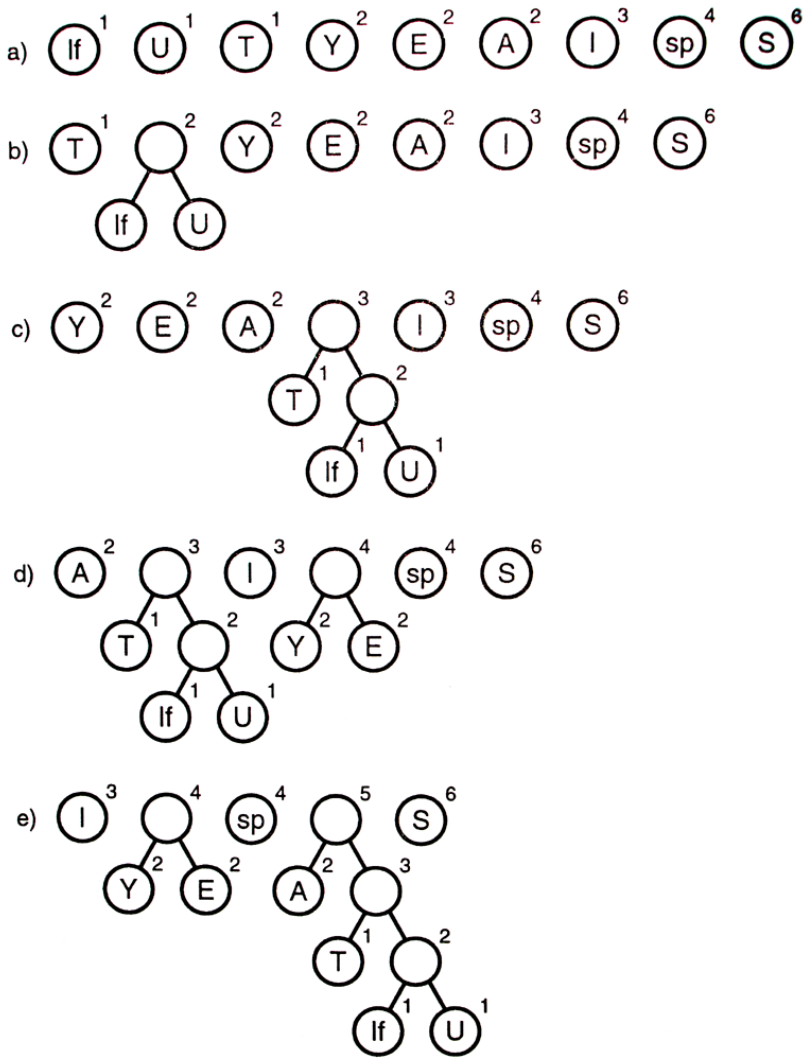
1. Crie uma nó da árvore para cada caractere distinto da mensagem
2. Crie uma lista de nós ordenada de acordo com a frequência de ocorrência dos caracteres

Criando a Árvore de Huffman II

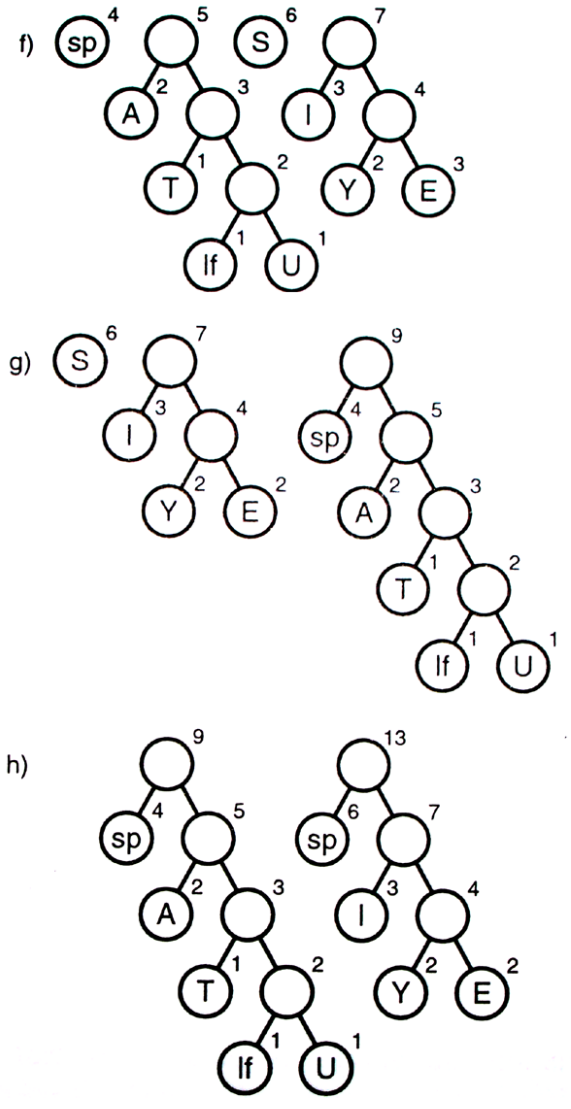
Montagem

1. Remova da lista de nós, os dois nós menos frequentes
2. Crie um novo nó, cuja frequência seja a soma das frequências dos dois nós retirados
3. Defina como o filho da esquerda desse novo nó, o nó com a menor frequência dos retirados, e como filho da direita o mais frequente
4. Insira esse novo nó na lista ordenada de nós
5. Repita os passos 1 a 4 até restar apenas um nó na lista
6. Esse nó representa a árvore de Huffman

Criando a Árvore de Huffman

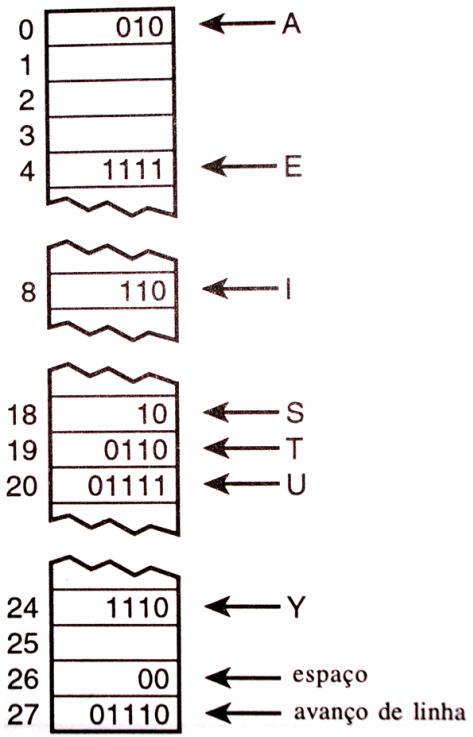


Criando a Árvore de Huffman



Codificando uma Mensagem

- Para se codificar uma mensagem, primeiro deve-se criar uma tabela que mapeie cada caractere de entrada em um código definido pela árvore
- Um jeito simples é criar um vetor onde os índices representam os códigos ASCII e que as células armazenem os bits da codificação
  - Por exemplo, o caractere **A** pode ficar no índice **0**, o **B** no índice **1**, e assim por diante
- Assim, para se codificar uma mensagem, para cada caractere de entrada, um valor da tabela é escolhido

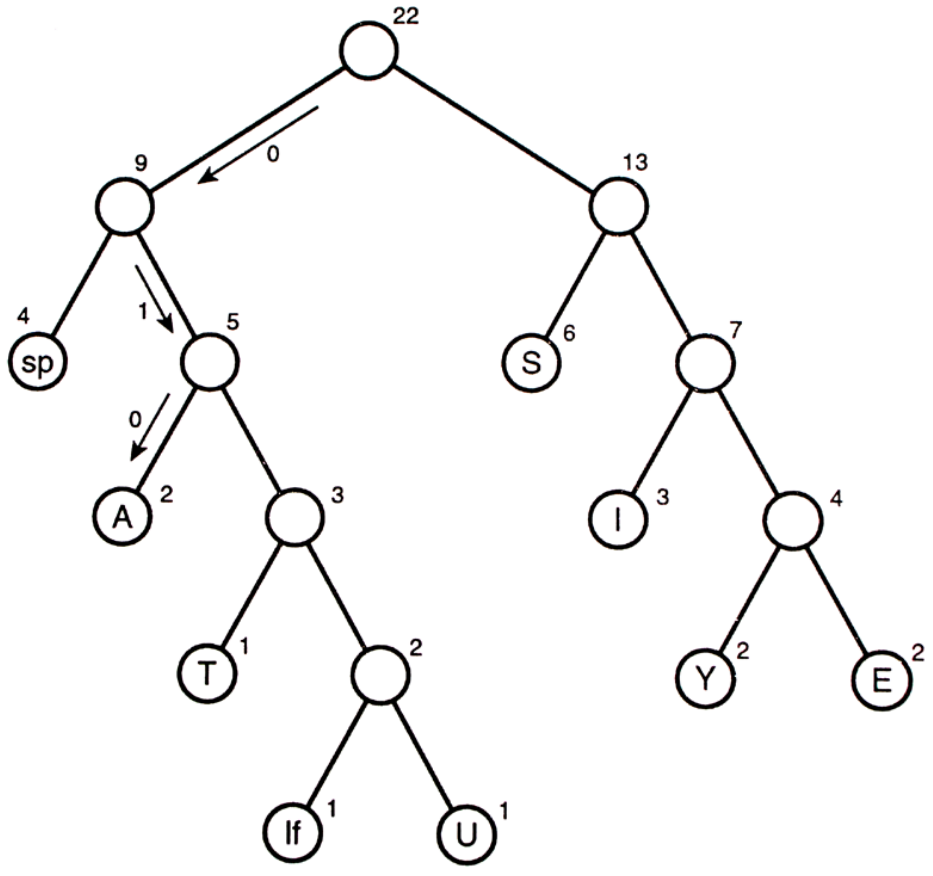


Criando o Código de Huffman

- O processo para se criar o código de Huffman para cada caractere distinto é similar a decodificação de uma mensagem

1. Dado um nó folha, parte-se da raiz até alcançá-lo
   - Se desceu pelo filho da esquerda, acrescenta 0 ao código
   - Se desceu pelo filho da direita, acrescenta 1 ao código

Criando o Código de Huffman



Sumário

Nó da Árvore

- Nó da árvore de Huffman

```
1  typedef struct NO {
2    int simbolo;
3    int freq;
4
5    struct NO *fesq;
6    struct NO *fdir;
7  } NO;
```

Heap Mínimo

```
1  #define TAM 500
2
3  typedef struct {
4    NO *itens[TAM];
5    int fim;
6  } HEAP;
7
8  void criar_heap(HEAP *heap);
9  int vazia_heap(HEAP *heap);
10 int cheia_heap(HEAP *heap);
11 int tamanho_heap(HEAP *heap);
12 void subir(HEAP *heap, int indice);
13 int inserir_heap(HEAP *heap, NO *no);
14 void descer(HEAP *heap, int indice);
15 struct NO *remover_heap(HEAP *heap);
16 void imprimir_heap(HEAP *heap);
```

Árvore Binária (Huffman)

```
1  #define TAM 500
2
3  typedef struct {
4    NO *raiz;
5    char codigo[TAM][TAM];
6  } ARVORE;
7
8  void inicializar_arvore(ARVORE *arv);
9  void limpar_arvore_aux(NO *raiz);
10 void limpar_arvore(ARVORE *arv);
11 void preordem_aux(NO *raiz);
12 void preordem(ARVORE *arv);
13
14 void criar_arvore(ARVORE *arv, char *msg);
15 void criar_codigo_aux(ARVORE *arv, NO *no, char *cod, int fim);
16 void criar_codigo(ARVORE *arv);
17 void imprimir_codigo(ARVORE *arv);
18
19 void codificar(ARVORE *arv, char *msg, char *cod);
20 void decodificar(ARVORE *arv, char *cod, char *msg);
```

Árvore Binária (Huffman)

```
1  void inicializar_arvore(ARVORE *arv) {
2    int i;
3
4    for (i=0; i < TAM; i++) {
5      arv->codigo[i][0] = '\0';
6    }
7
8    arv->raiz = NULL;
9  }
10
11 void limpar_arvore_aux(NO *raiz) {
12   if (raiz != NULL) {
13     limpar_arvore_aux(raiz->fesq);
14     limpar_arvore_aux(raiz->fdir);
15     free(raiz);
16   }
17 }
18
19 void limpar_arvore(ARVORE *arv) {
20   limpar_arvore_aux(arv->raiz);
21   arv->raiz = NULL;
22 }
```

Criar Árvore

```
1  void criar_arvore(ARVORE *arv, char *msg) {
2    //contando a frequencia (ASCII)
3    int i, freq[TAM];
4    for (i=0; i < TAM; i++) freq[i] = 0;
5    for (i=0; msg[i] != '\0'; i++) {
6      freq[(int)msg[i]]++;
7    }
8
9    HEAP heap;
10   criar_heap(&heap);
11
12   for (i=0; i < TAM; i++) {
13     if (freq[i] > 0) {
14       NO *pno = (NO *)malloc(sizeof(NO));
15       pno->simbolo = i;
16       pno->freq = freq[i];
17       pno->fesq = NULL;
18       pno->fdir = NULL;
19
20       inserir_heap(&heap, pno);
21     }
22   }
23   ...
24 }
```

Criar Árvore

```
1  void criar_arvore(ARVORE *arv, char *msg) {
2    ...
3
4    while (tamanho_heap(&heap) > 1) {
5      NO *pfesq=remover_heap(&heap);
6      NO *pfdir=remover_heap(&heap);
7
8      NO *pnovo = (NO *)malloc(sizeof(NO));
9      pnovo->simbolo = '#';
10     pnovo->freq = pfesq->freq + pfdir->freq;
11     pnovo->fesq = pfesq;
12     pnovo->fdir = pfdir;
13
14     inserir_heap(&heap, pnovo);
15   }
16
17   arv->raiz = remover_heap(&heap);
18 }
```

Criando o Código

```
1  void criar_codigo(ARVORE *arv) {
2    char codigo[TAM];
3    criar_codigo_aux(arv, arv->raiz, codigo, -1);
4  }
```

Criando o Código

```
1  void criar_codigo_aux(ARVORE *arv, NO *no, char *cod, int fim) {
2    if (no != NULL) {
3      if (no->fesq == NULL && no->fdir == NULL) {
4        int i;
5        for (i=0; i <= fim; i++) {
6          arv->codigo[(int)no->simbolo][i] = cod[i];
7        }
8        arv->codigo[(int)no->simbolo][fim+1] = '\0';
9
10     } else {
11       if (no->fesq != NULL) {
12         fim++;
13         cod[fim] = '0';
14         criar_codigo_aux(arv, no->fesq, cod, fim);
15         fim--;
16       }
17
18       if (no->fdir != NULL) {
19         fim++;
20         cod[fim] = '1';
21         criar_codigo_aux(arv, no->fdir, cod, fim);
22         fim--;
23       }
24     }
25   }
26 }
```

Codificando uma Mensagem

```
1  void codificar(ARVORE *arv, char *msg, char *cod) {
2    int i, j, cod_fim;
3
4    cod_fim = -1; //aponta para a última posição da codificação
5
6    for (i=0; msg[i] != '\0'; i++) {
7      //recuperando o código do caractere
8      char *pcod = arv->codigo[(int)msg[i]];
9
10     //copiando o código na codificação
11     for (j=0; pcod[j] != '\0'; j++) {
12       cod_fim++;
13       cod[cod_fim] = pcod[j];
14     }
15   }
16
17   cod[cod_fim+1] = '\0';
18 }
```

Decodificando um Código

```
1  void decodificar(ARVORE *arv, char *cod, char *msg) {
2    int i, decod_fim;
3
4    decod_fim = -1; //aponta para a última posição da decodificação
5
6    NO *pno = arv->raiz;
7
8    for (i=0; cod[i] != '\0'; i++) {
9      if (cod[i] == '0') {
10       pno = pno->fesq;
11     } else if (cod[i] == '1') {
12       pno = pno->fdir;
13     } else {
14       printf("Simbolo codificado errado!\n");
15     }
16
17     if (pno->fesq == NULL && pno->fdir == NULL) {
18       decod_fim++;
19       msg[decod_fim] = pno->simbolo;
20       pno = arv->raiz;
21     }
22   }
23
24   msg[decod_fim+1] = '\0';
25 }
```

Exercício

- Terminar a implementação da compactação/descompactação usando Huffman