# Tipos Especiais de Listas

## Aplicação de Árvores: Código de Huffman

# Sumário

- Conceitos Introdutórios
- Código de Huffman
- Implementação

# Introdução

- Com o crescimento da quantidade de dados gerados e transmitidos, compactação desses se torna cada dia mais essencial
  - Armazenamento de dados (imagens médicas
    - 5000x3000x2 = 30Mbytes
  - 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

- 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

- Taxa de Compactação

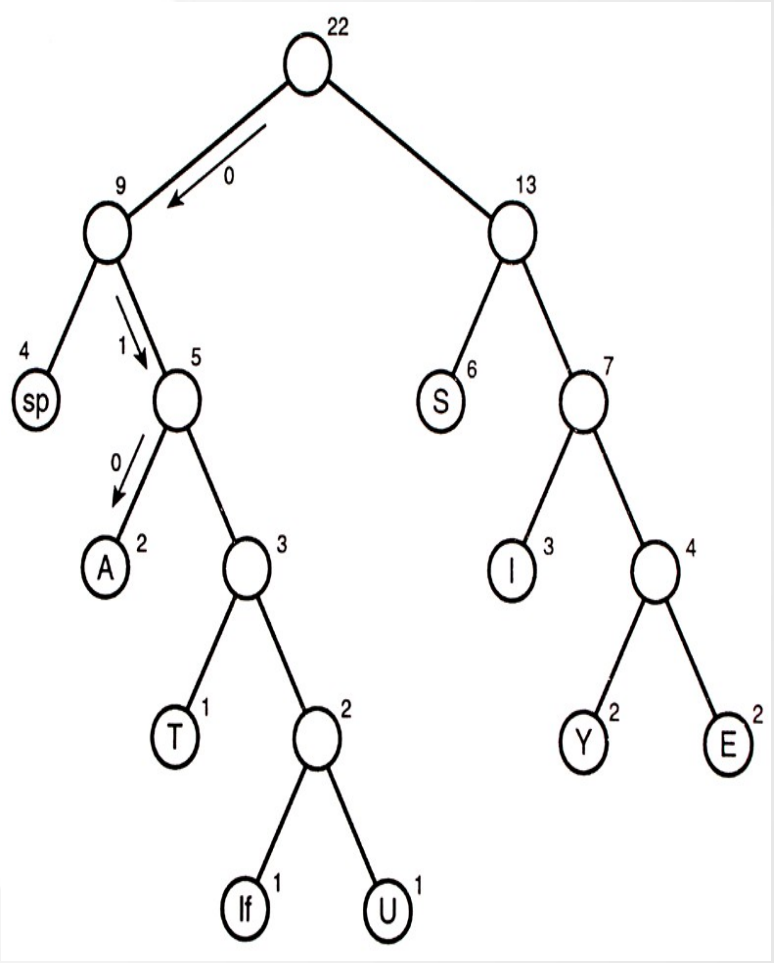
$$T_c = 1 - \frac{68}{22 \times 8} = 1 - \frac{68}{196} = 0,614$$

# Código de Huffman

# Decodificando com a Árvore de Huffman

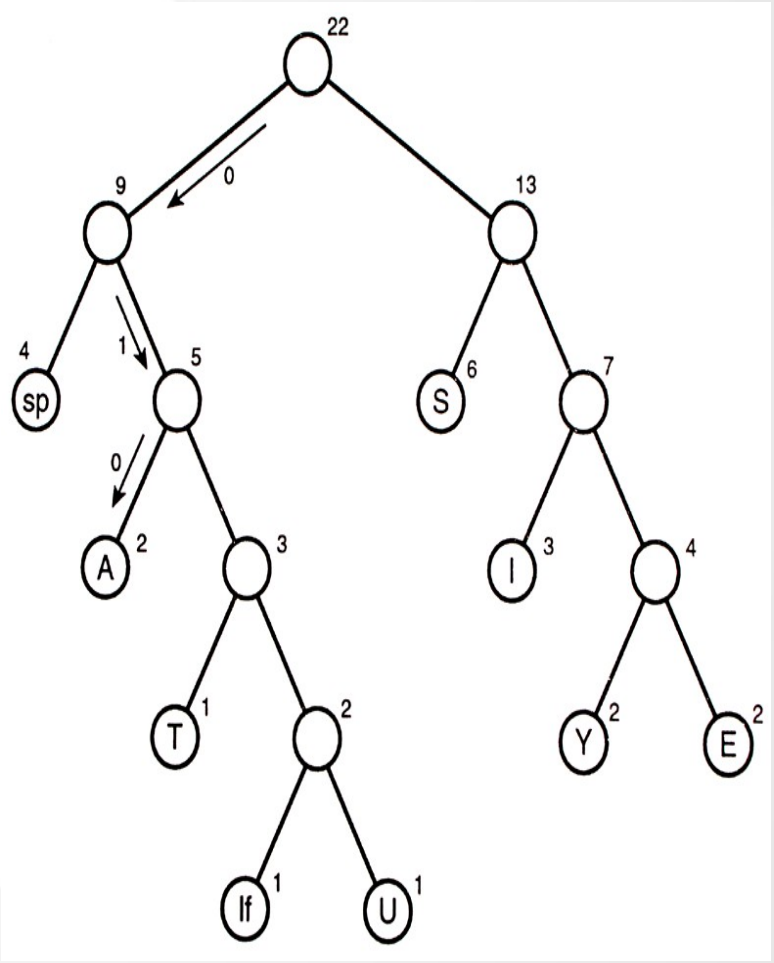
- Antes de vermos como codificar, vamos ver um processo mais fácil: a decodificação

- 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

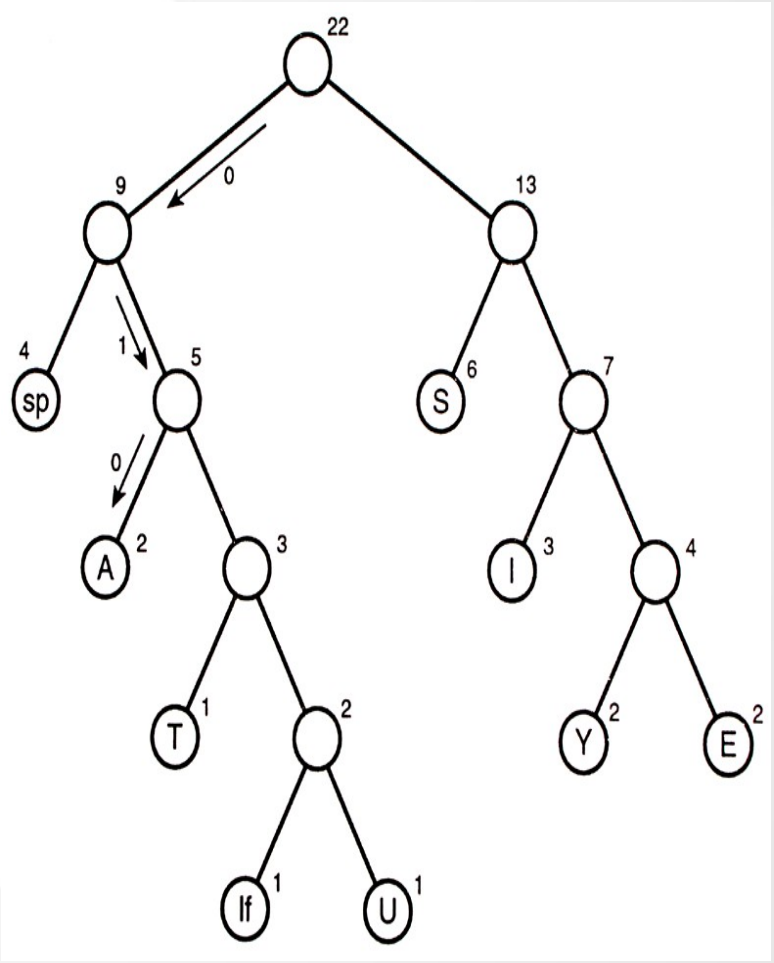
- Os caracteres das mensagem aparecem na árvore como folhas
- Quanto mais alta a frequência de um termo, mais alto ele aparecerá na árvore
- Números representam as frequências



# 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

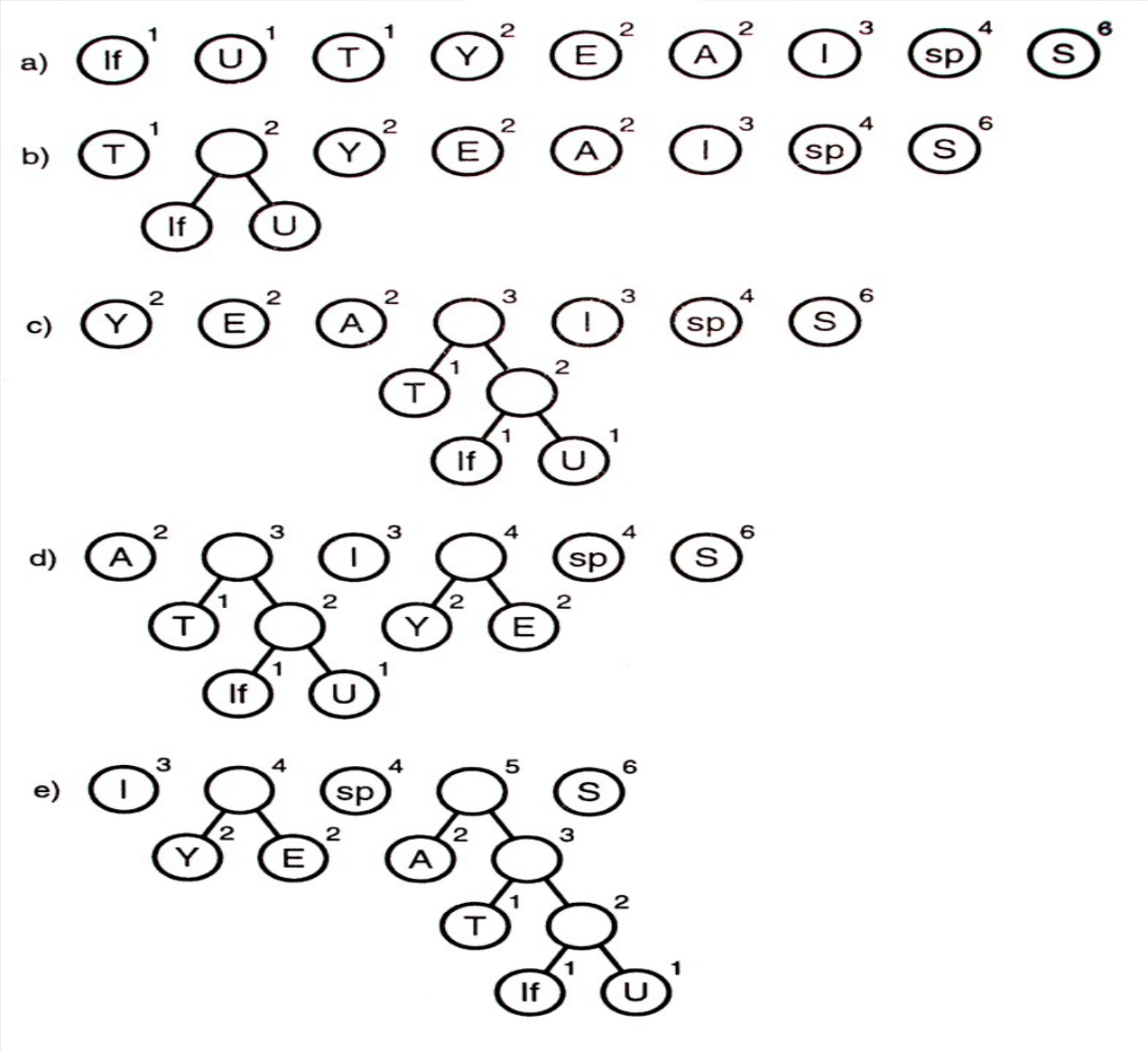
- Inicialização

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

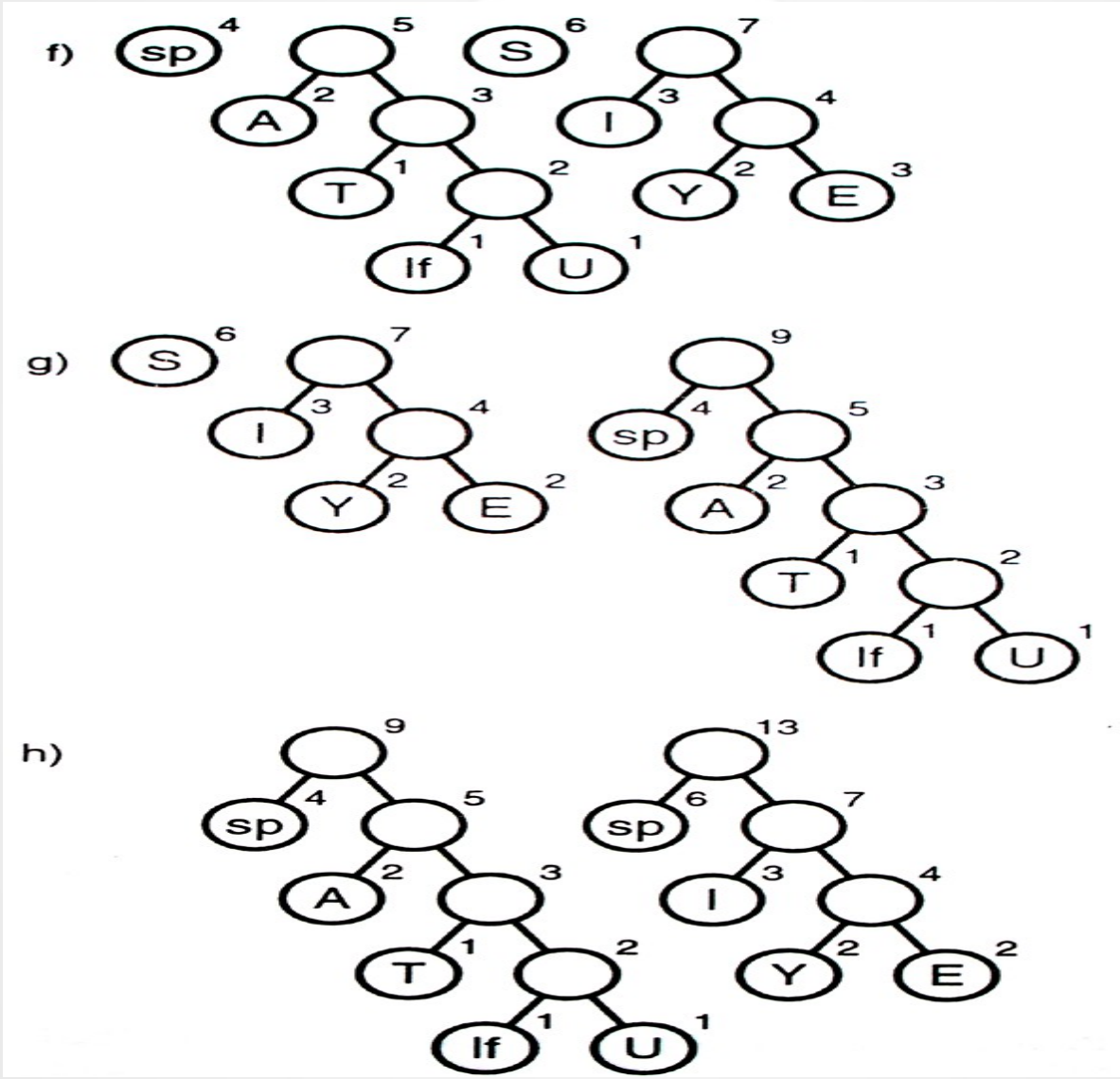
# Criando a Árvore de Huffman II

- Montagem
  - Remova da lista de nós, os dois nós menos frequentes
  - Crie um novo nó, cuja frequência seja a soma das frequências dos dois nós retirados
  - 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
  - Insira esse novo nó na lista ordenada de nós
  - Repita os passos 1 a 4 até restar apenas um nó na lista
  - Esse nó representa a árvore de Huffman

# Criando a Árvore de Huffman

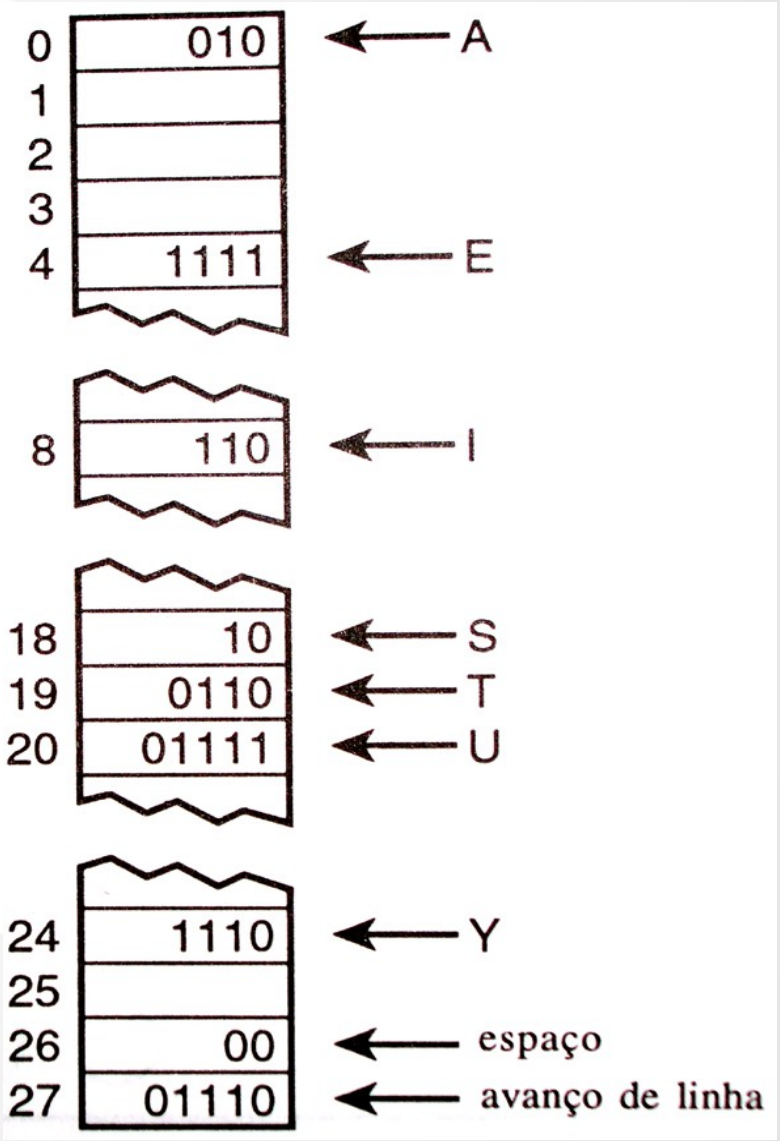


# Criando a Árvore de Huffman



# 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

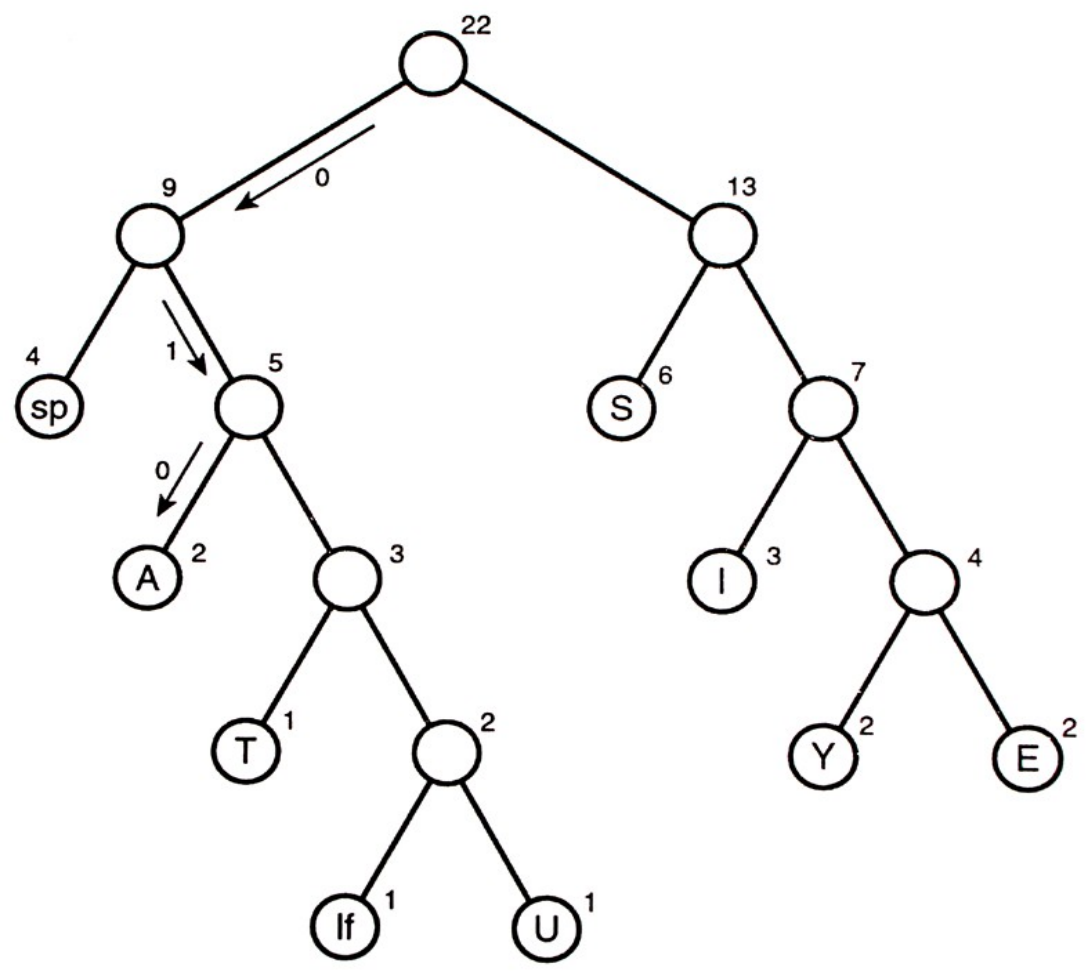


# Criando o Código de Huffman

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

- 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



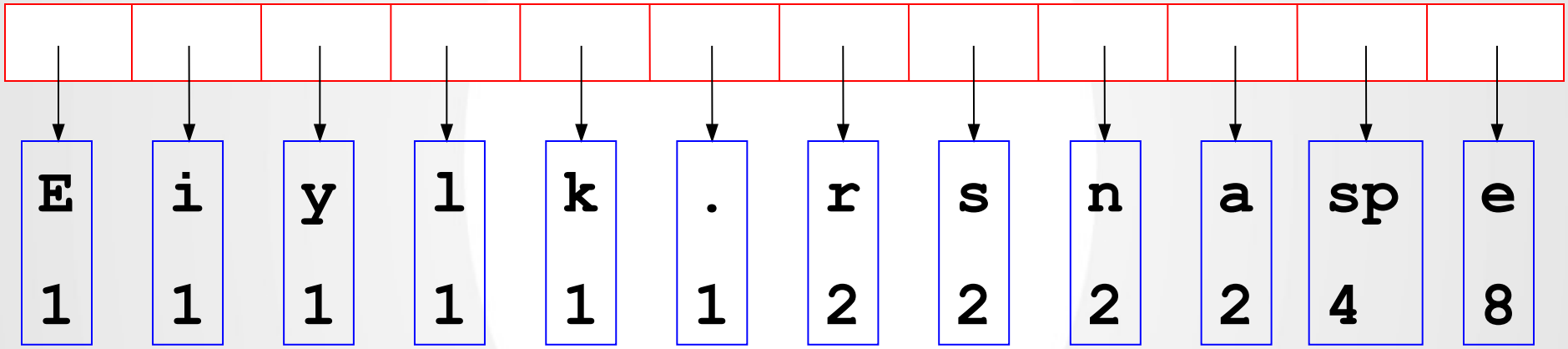
# Implementação

# Algoritmo básico

- Scanear o texto a ser comprimido e registar a ocorrência de todos os caracteres
- Ordenar os caracteres baseado no número de ocorrências no texto (prioridade)
- Construir a árvore de código de hufffman baseado na lista de prioridade
- Realizar um percurso na árvore para determinar todas as palavras de código
- Scanear o texto novamente e criar um novo arquivo usando  o código de huffman

# Criando a árvore (fila ordenada)

- A fila depois de inserir todos os nós

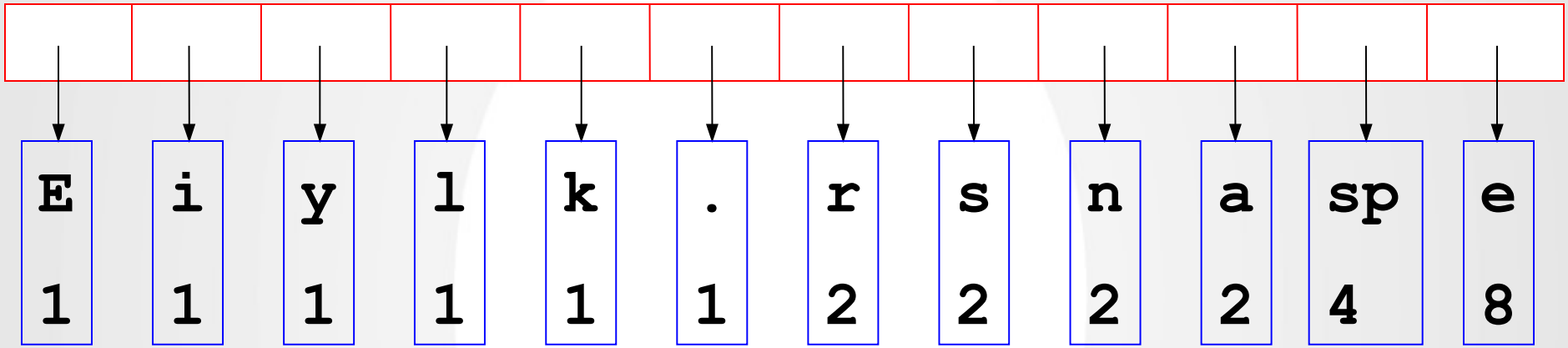



- Ponteiro Null não são mostrados

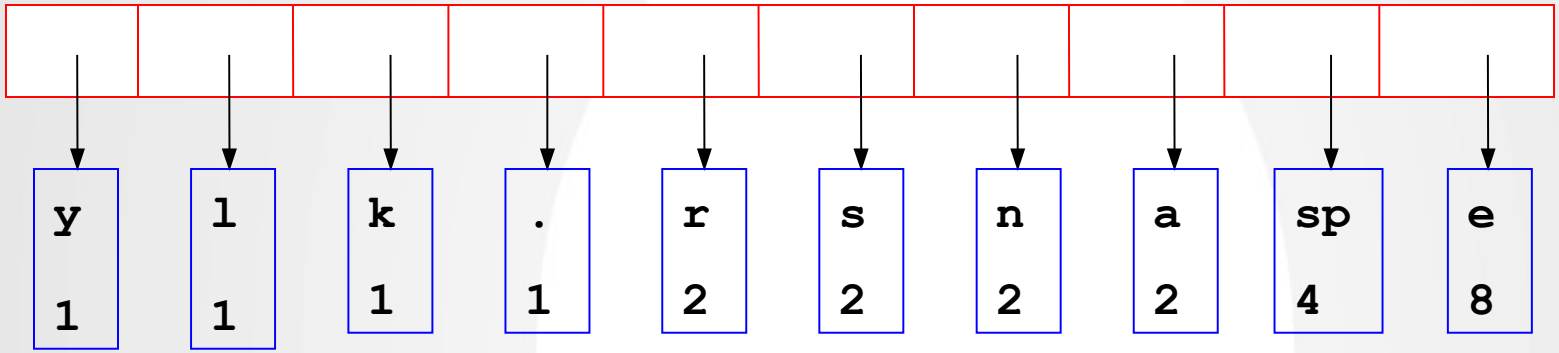
# Criando a Árvore

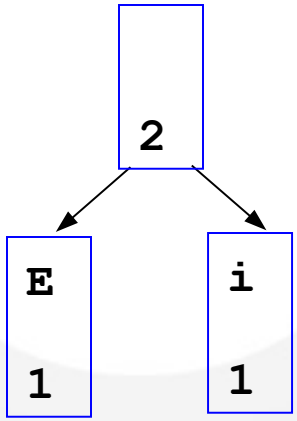
- Montagem
  - Remova da lista de nós, os dois nós menos frequentes
  - Crie um novo nó, cuja frequência seja a soma das frequências dos dois nós retirados
  - Defina como o filho da esquerda desse novo nó, o nó com a menor frequência dos retirados, e como filho da direita o mais frequente
  - Insira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

# Criando a Árvore

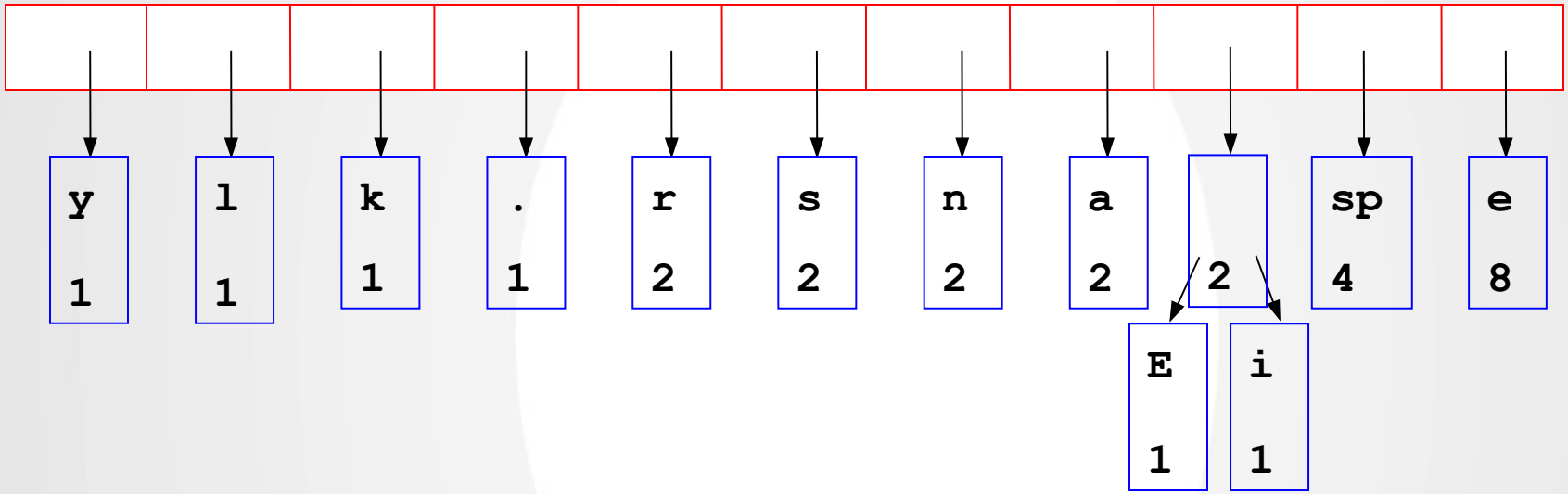


# Criando a Árvore

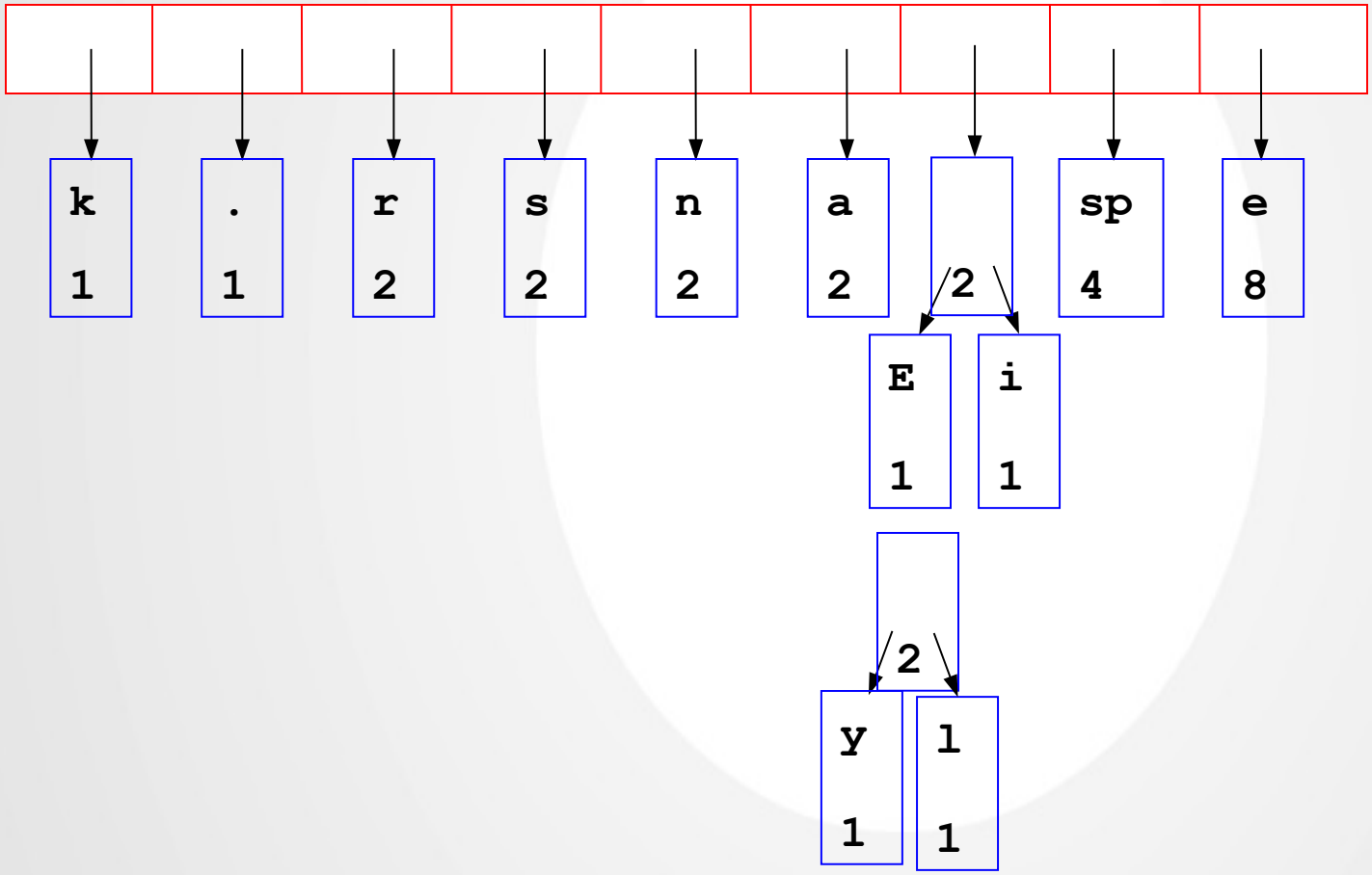


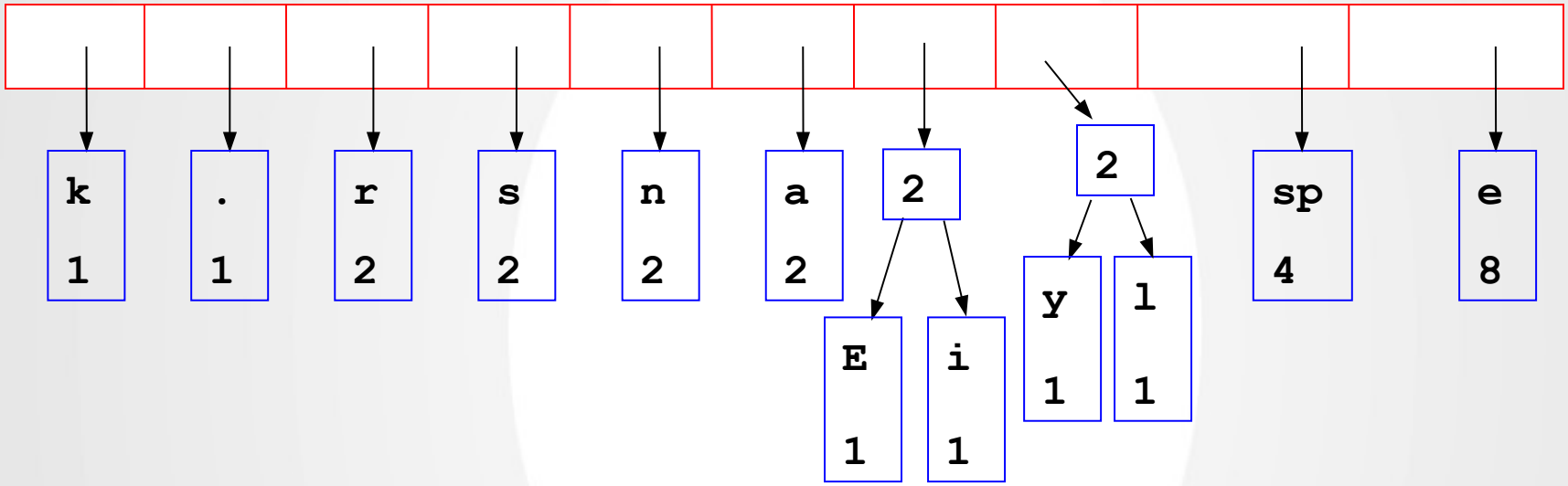
# Criando a Árvore



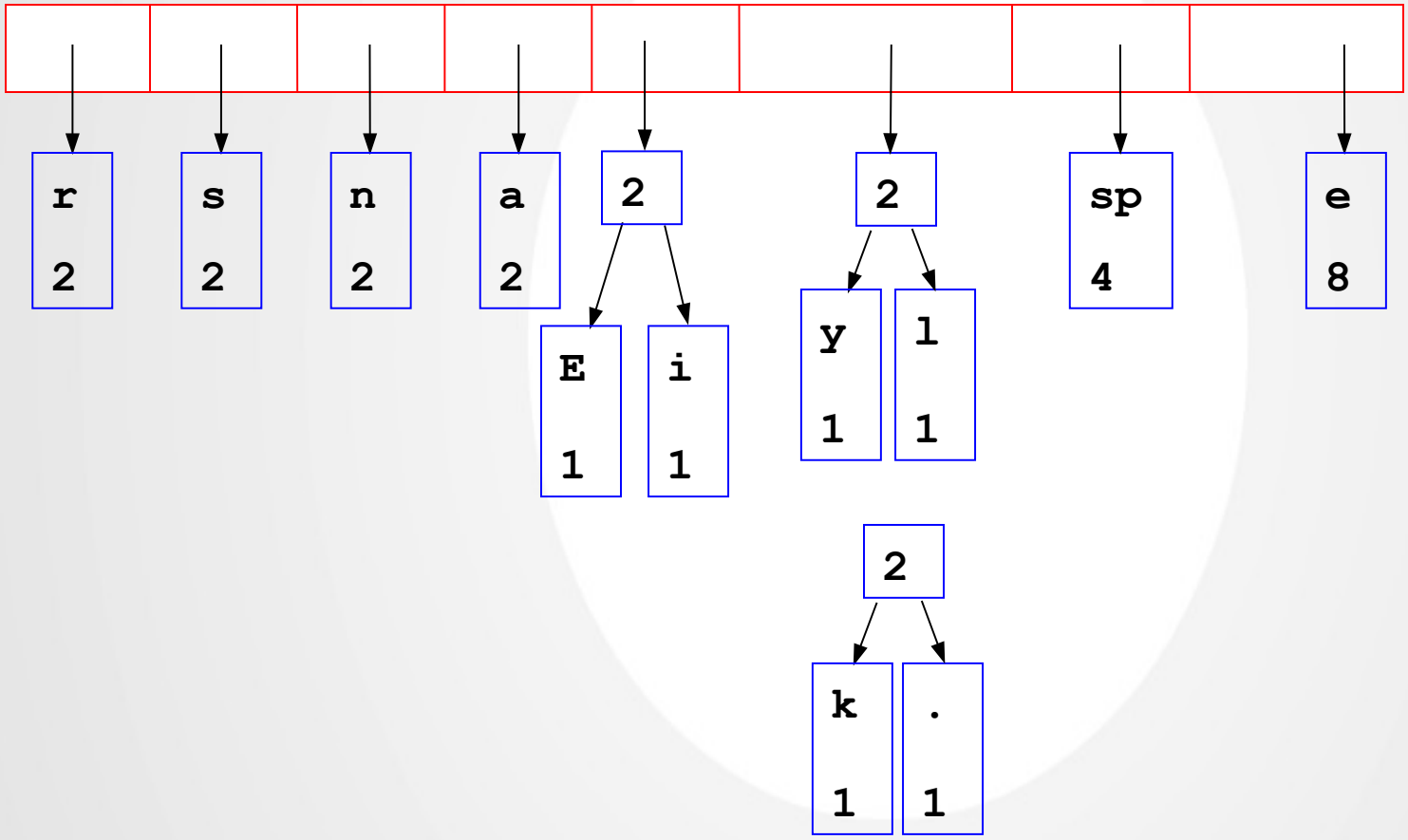
# Criando a Árvore



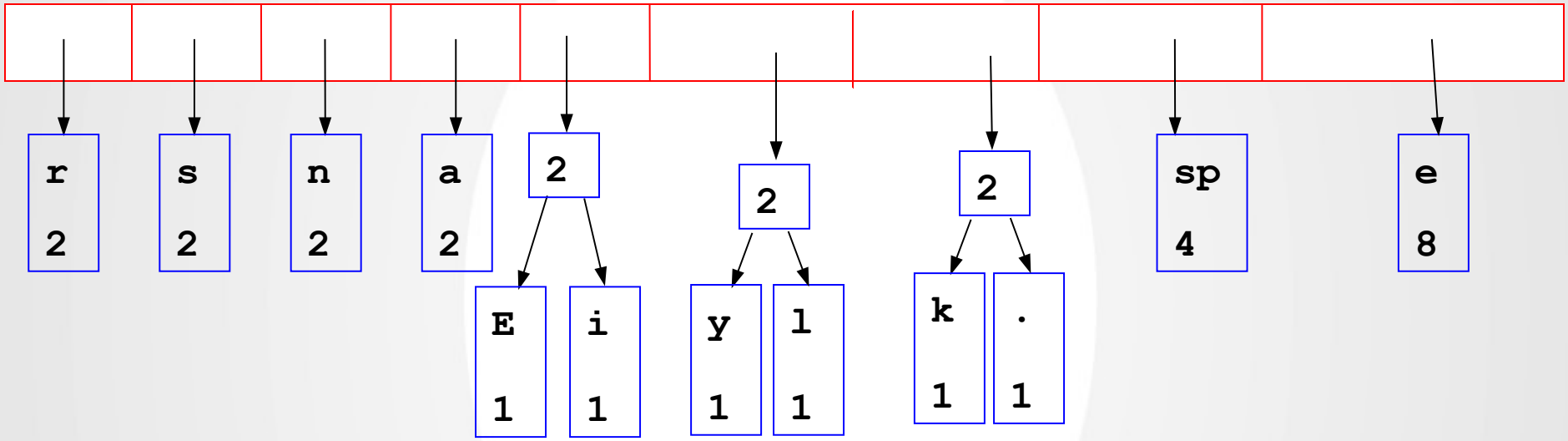
# Criando a Árvore



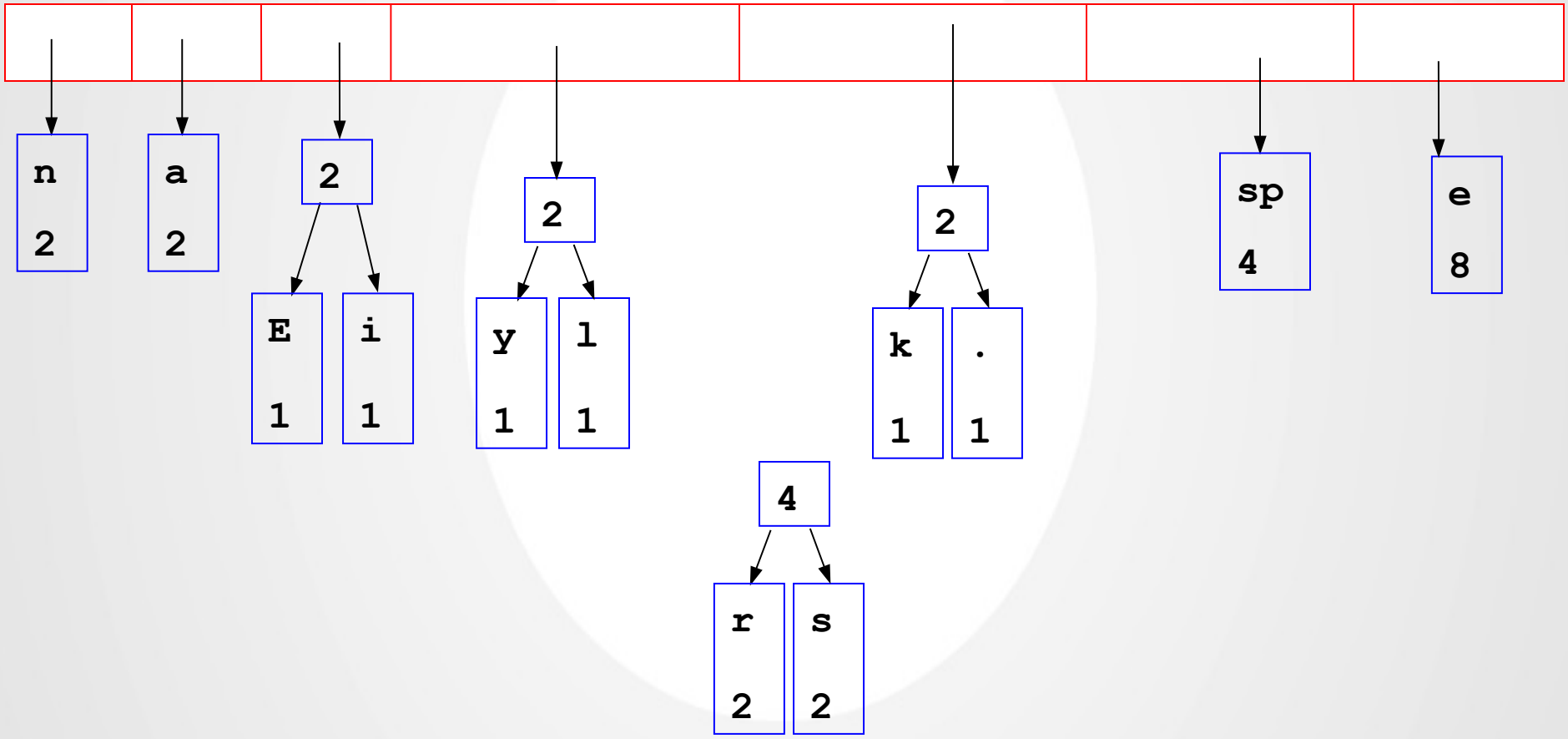
# Criando a Árvore



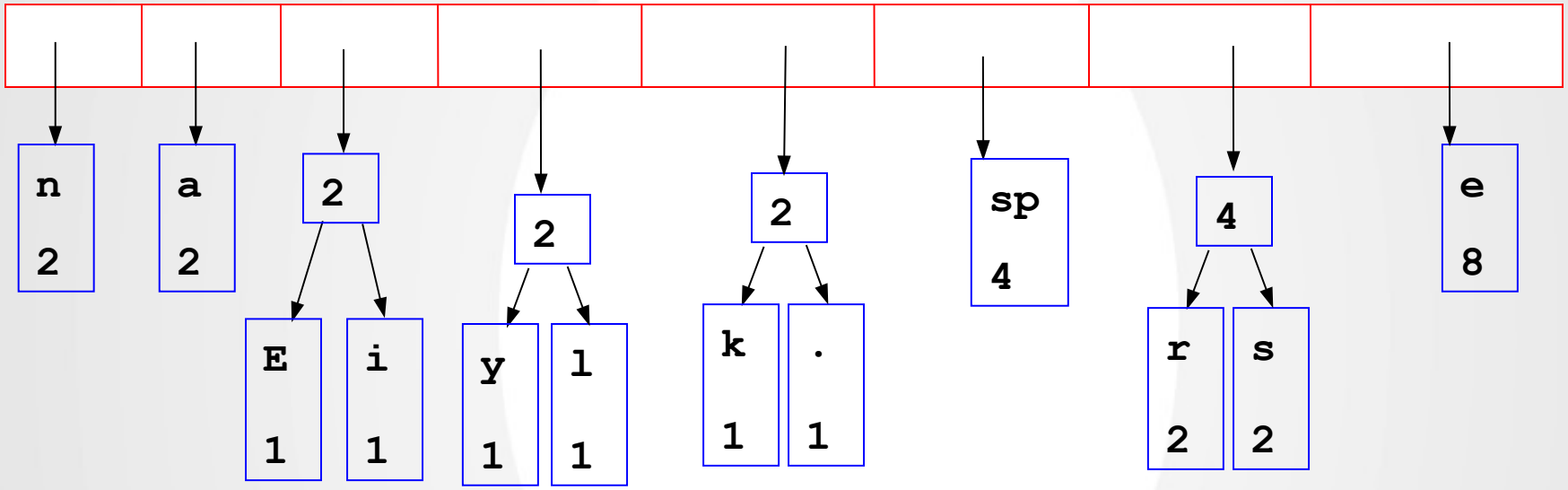
# Criando a Árvore



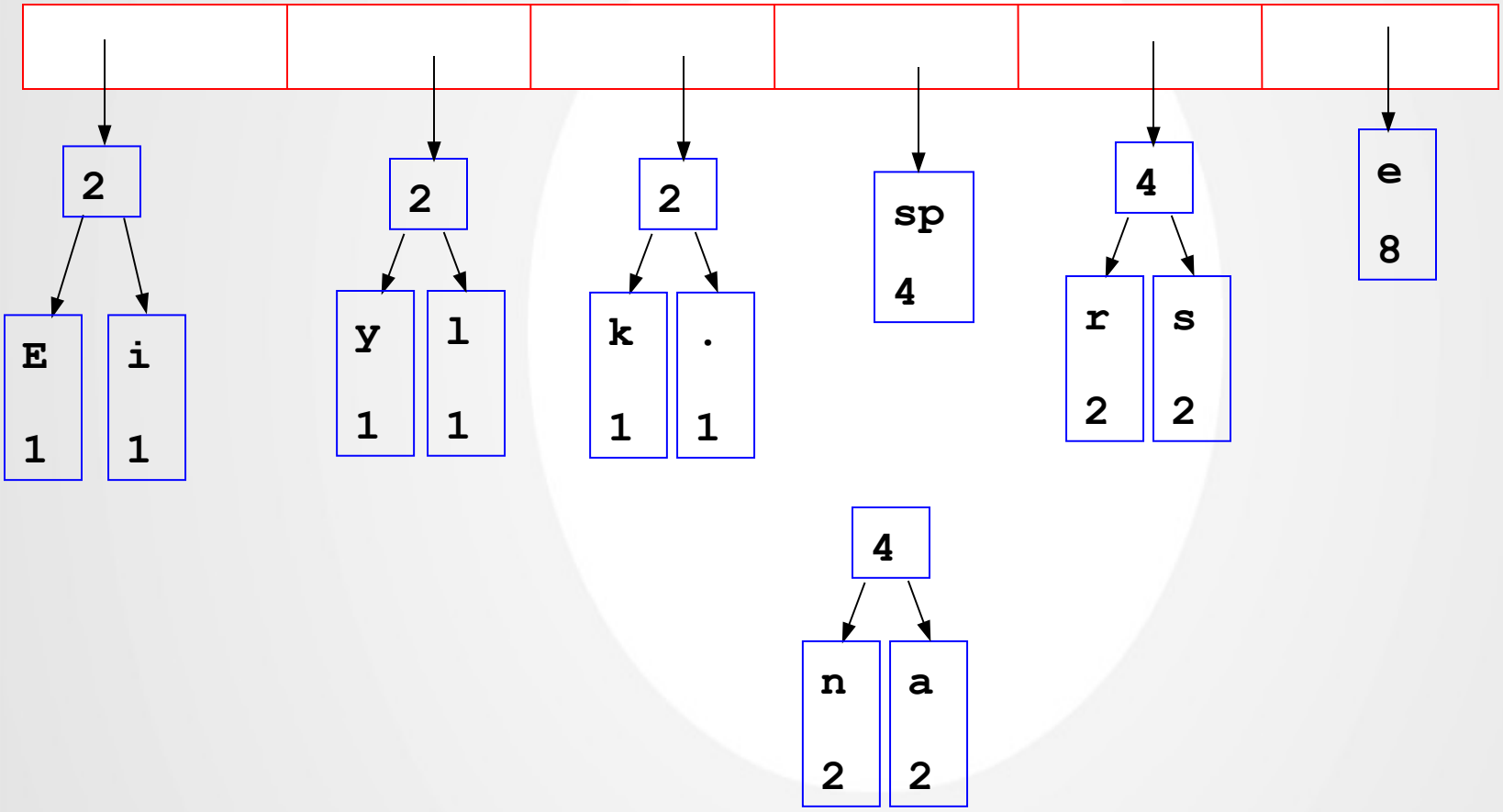
# Criando a Árvore



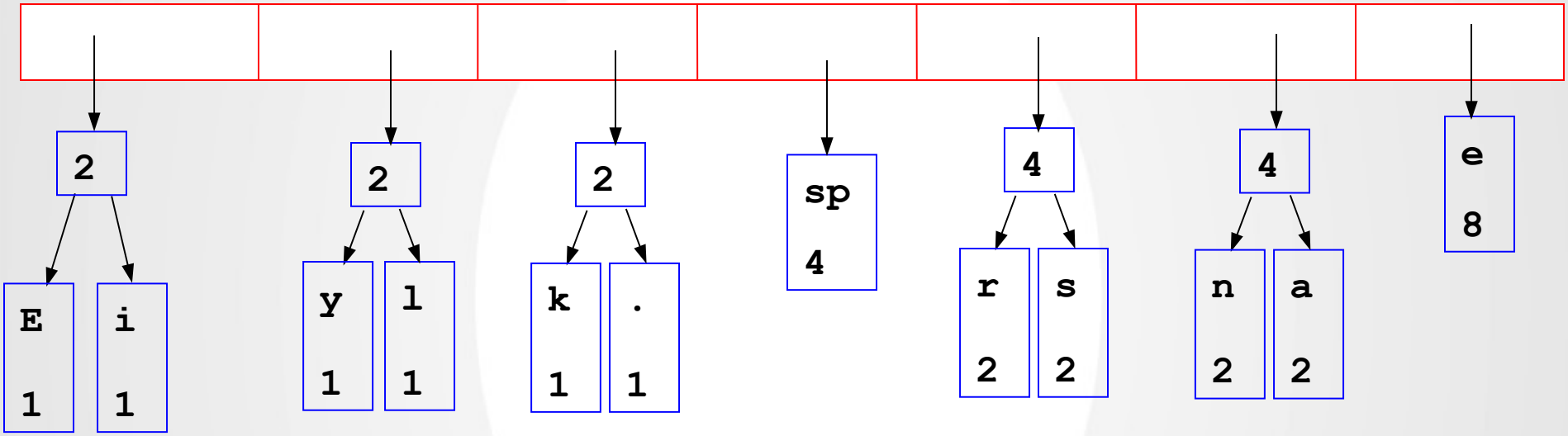
# Criando a Árvore



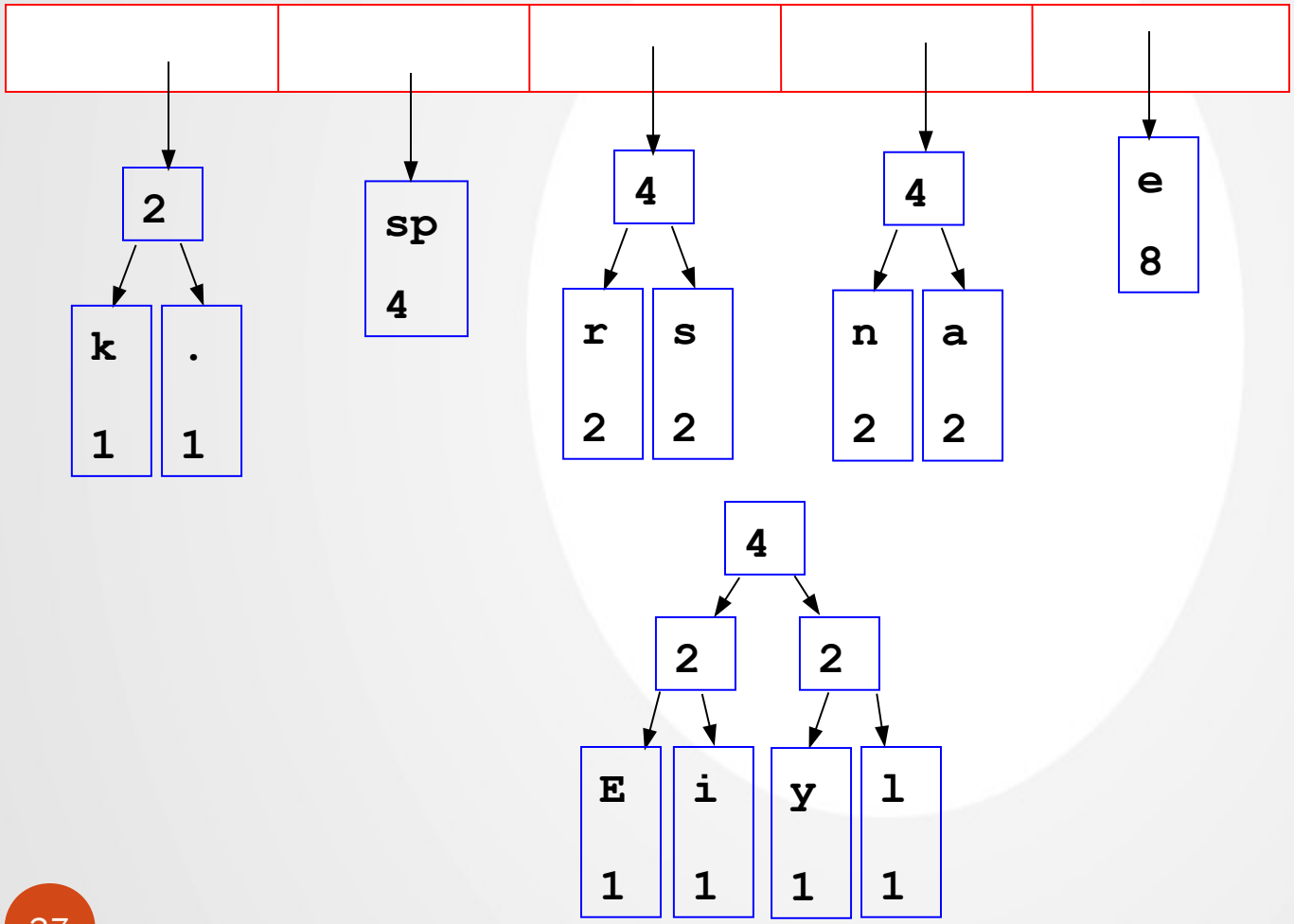
# Criando a Árvore



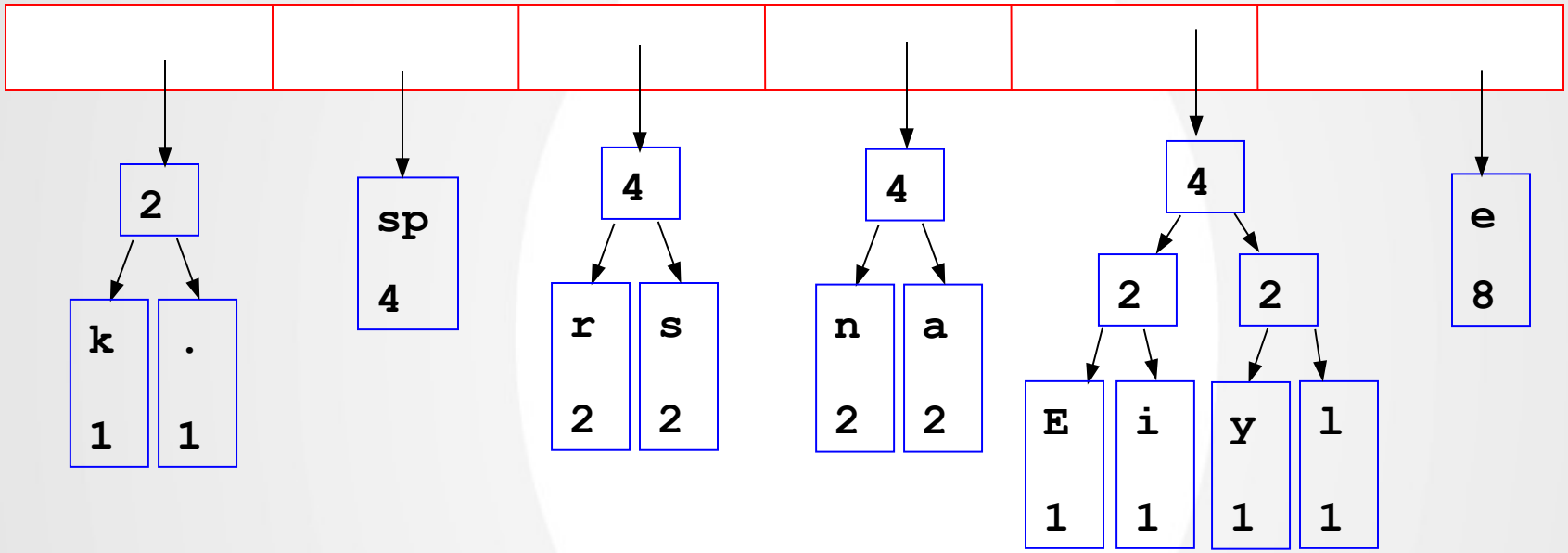
# Criando a Árvore



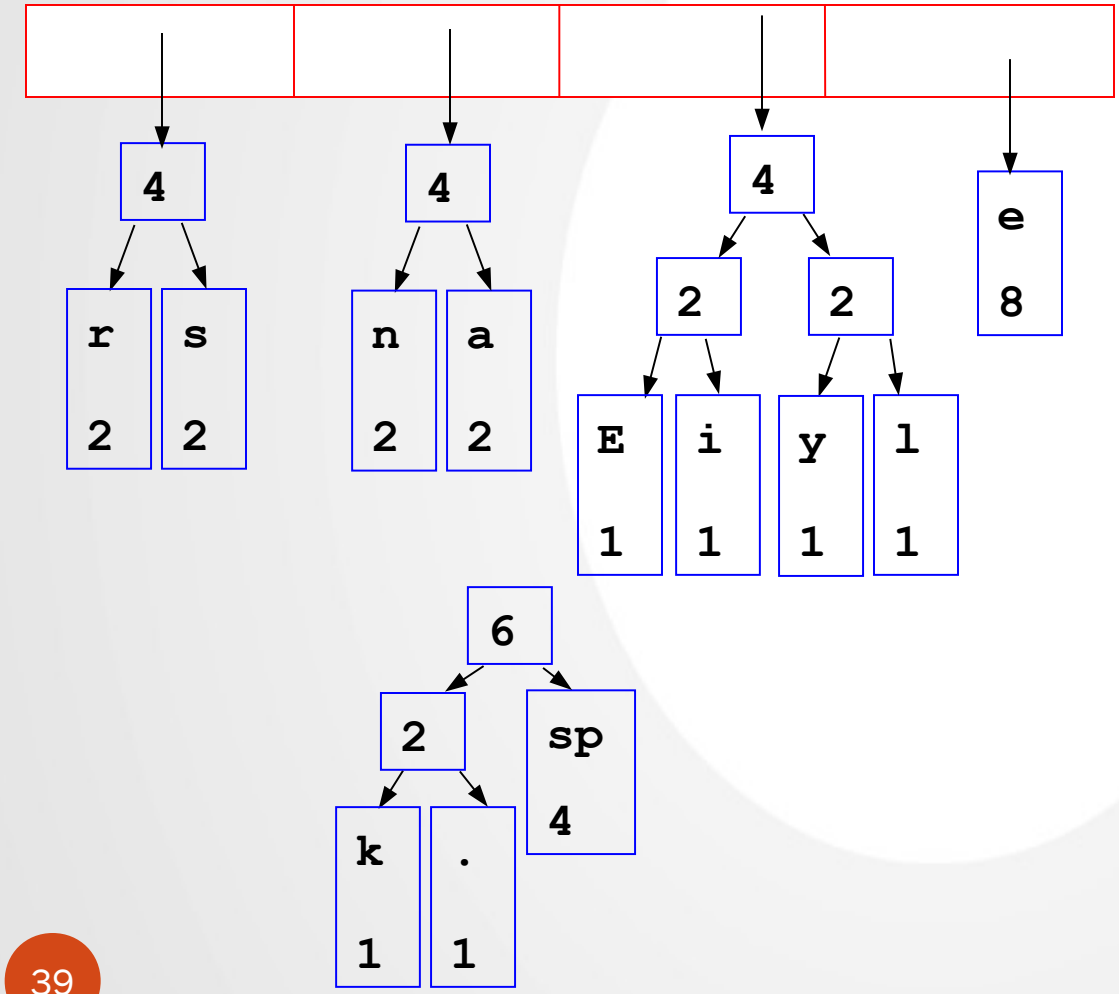
# Criando a Árvore



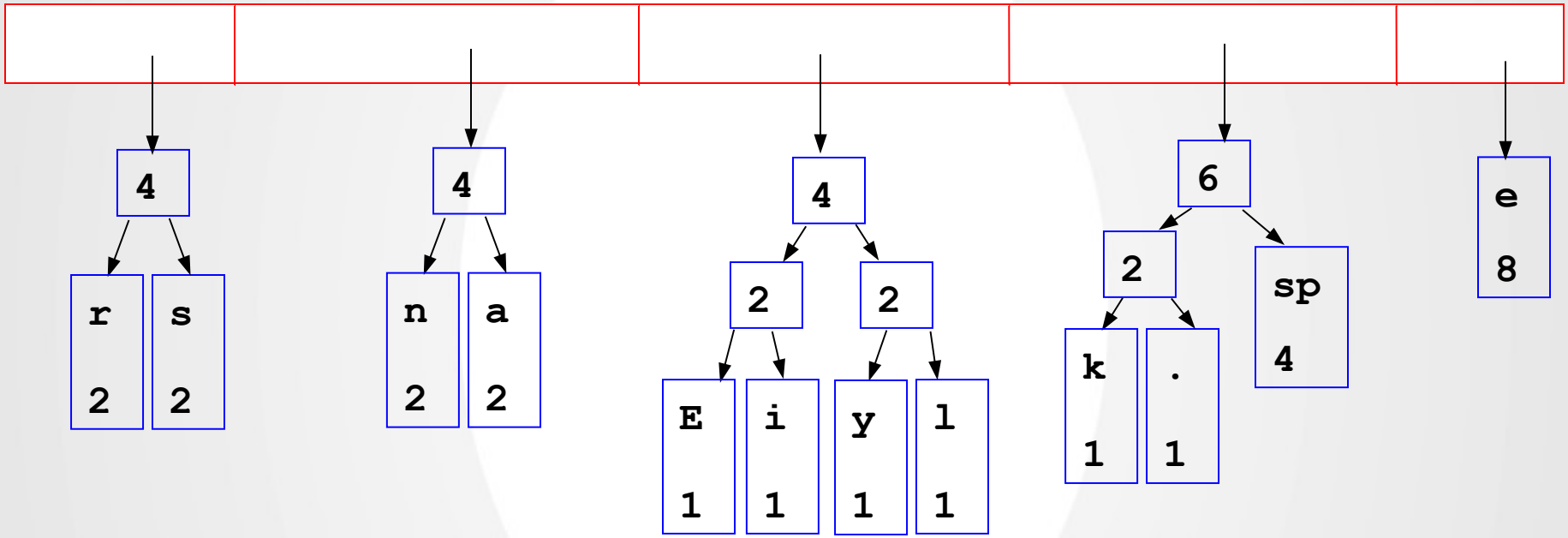
# Criando a Árvore



# Criando a Árvore

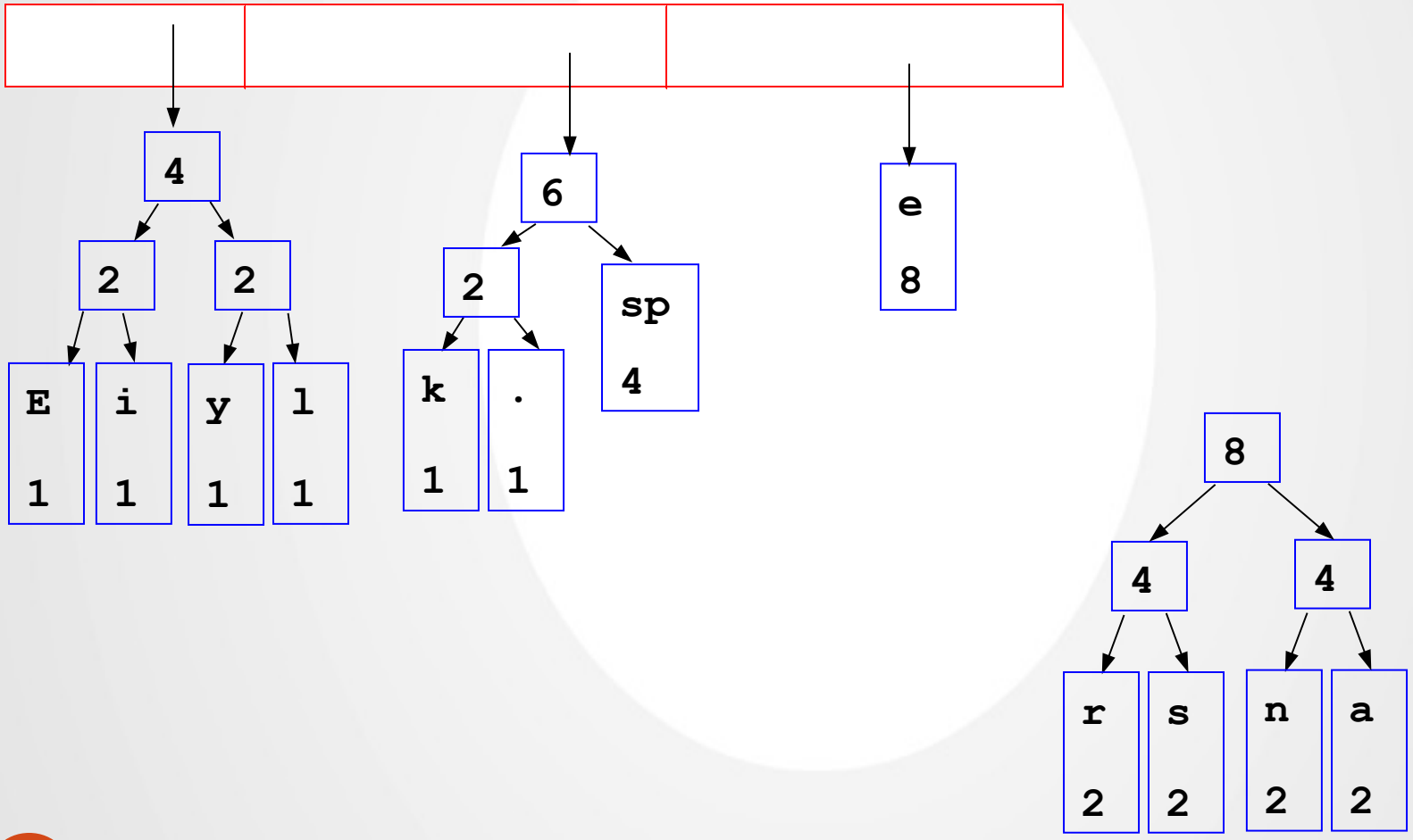


# Criando a Árvore

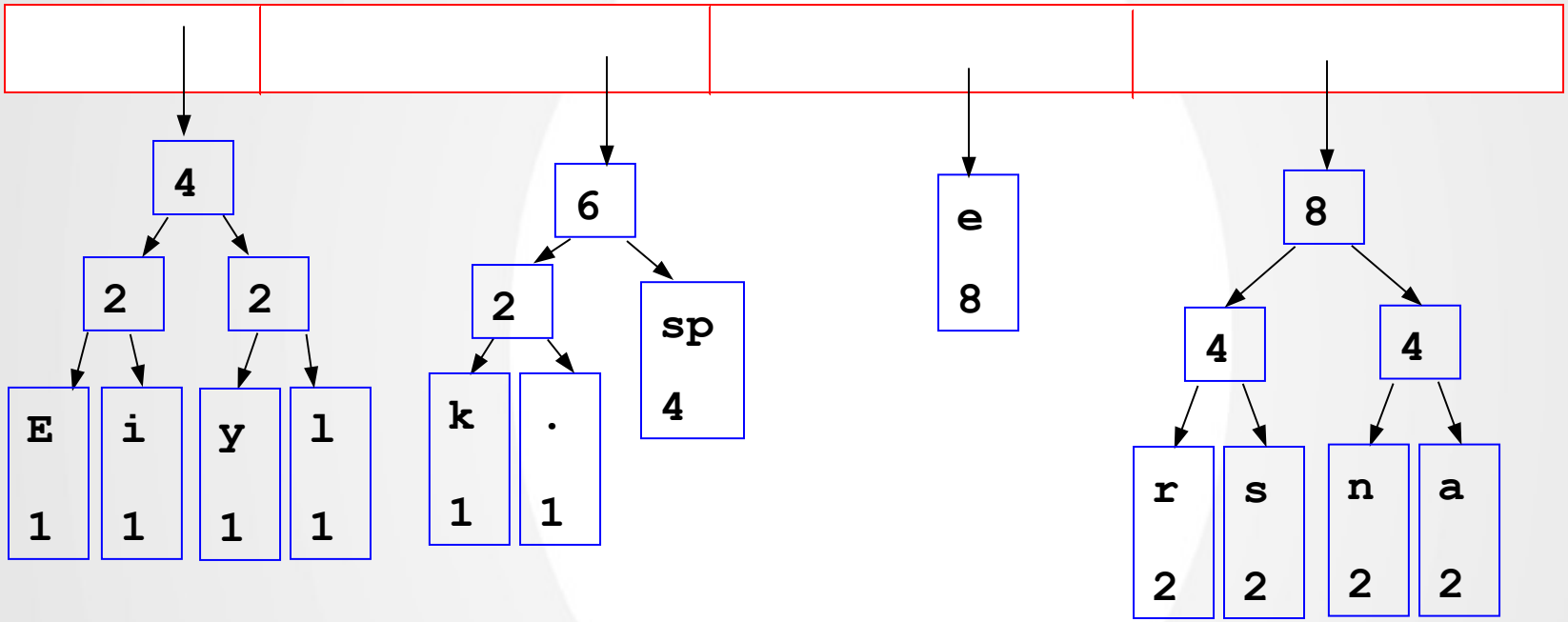



O que está acontecendo para os caracteres com um número baixo de ocorrências?

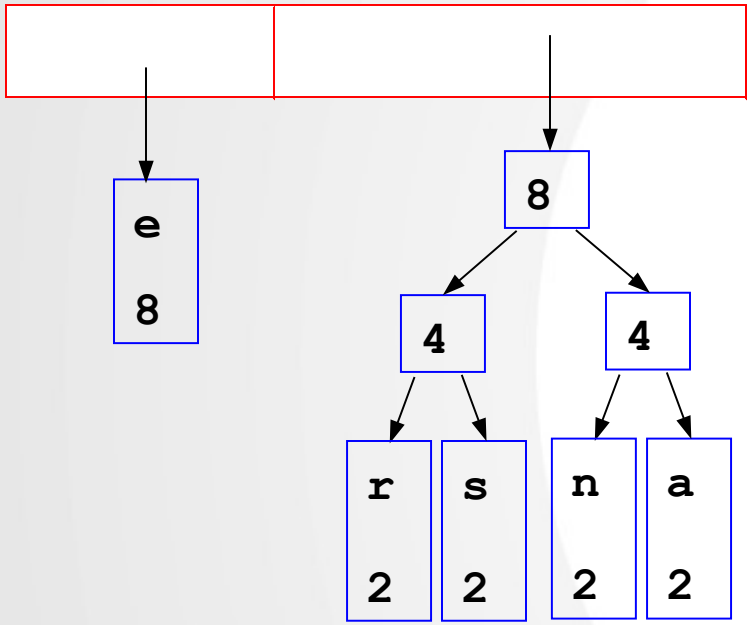
# Criando a Árvore

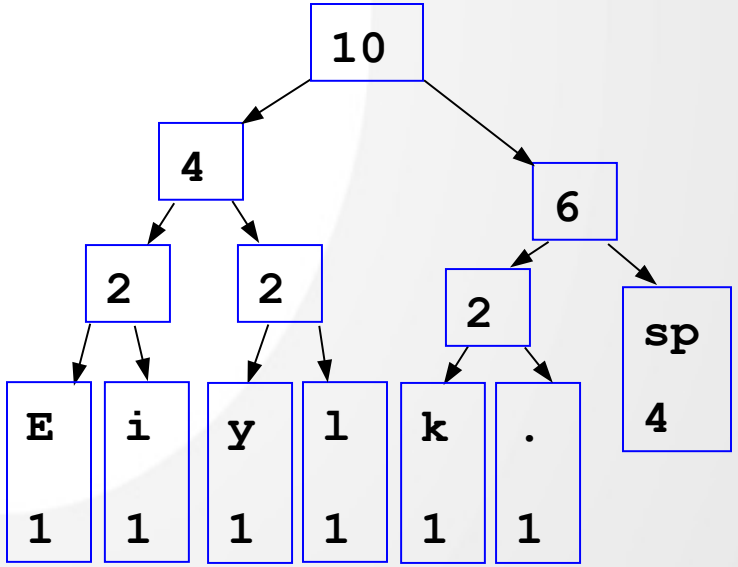


# Criando a Árvore

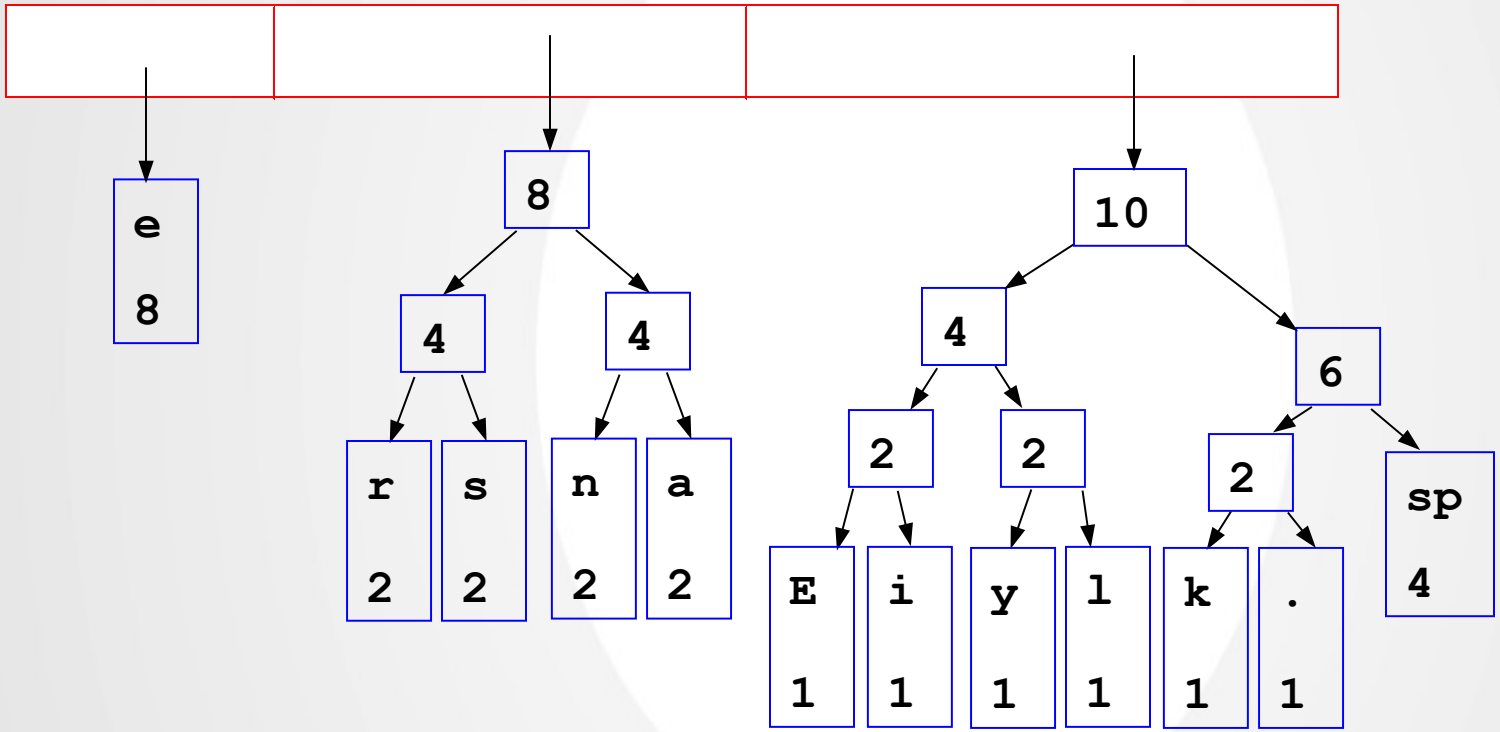


# Criando a Árvore

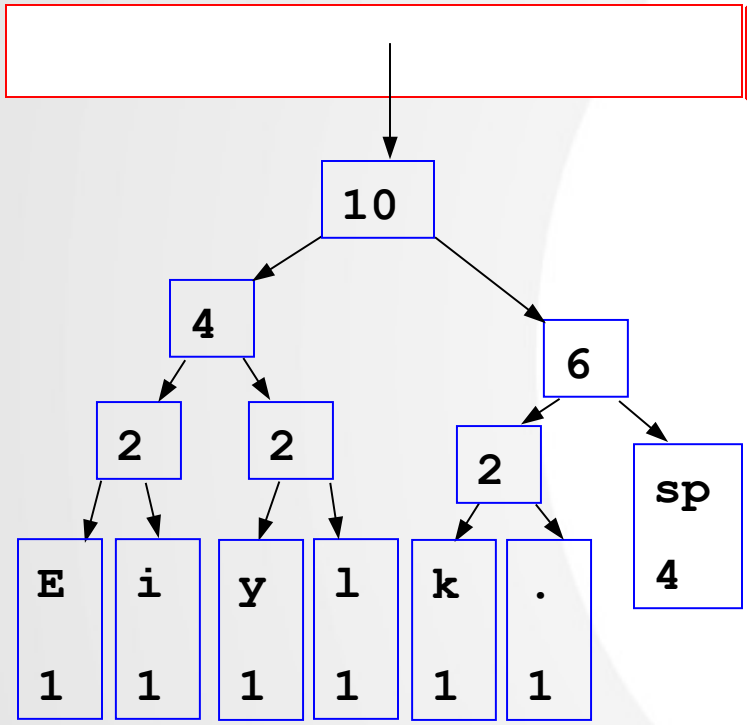


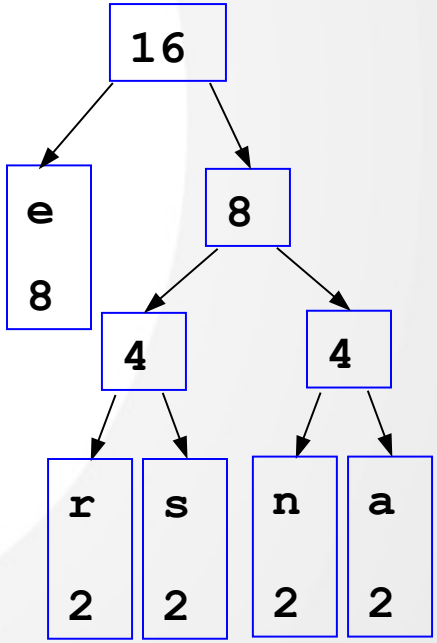



# Criando a Árvore

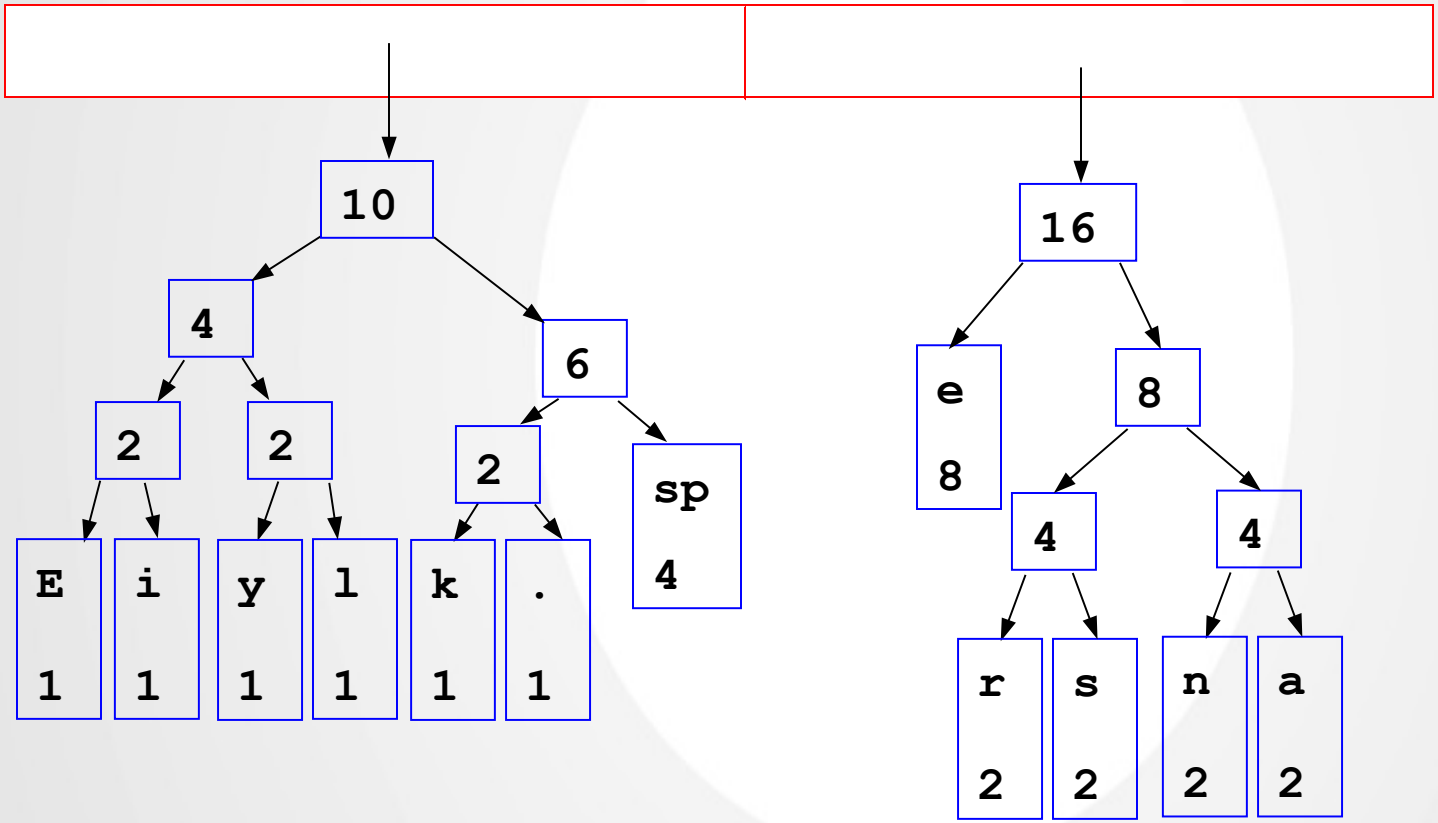


# Criando a Árvore

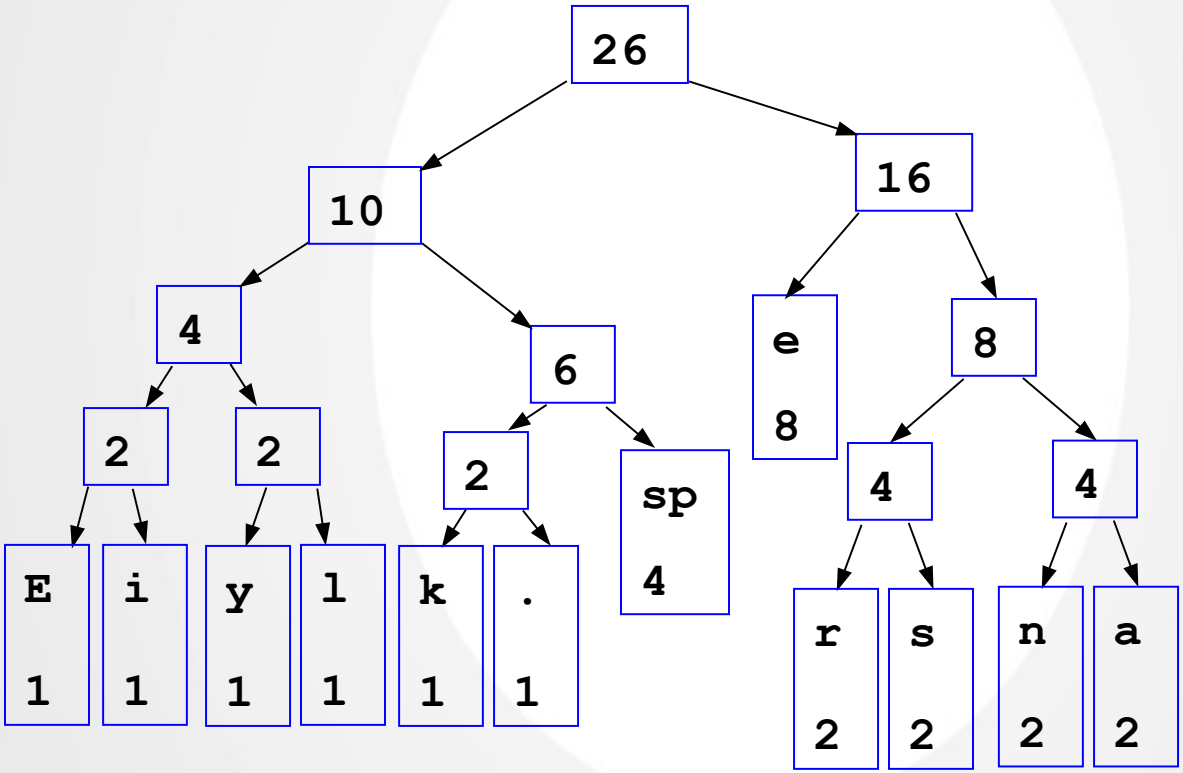


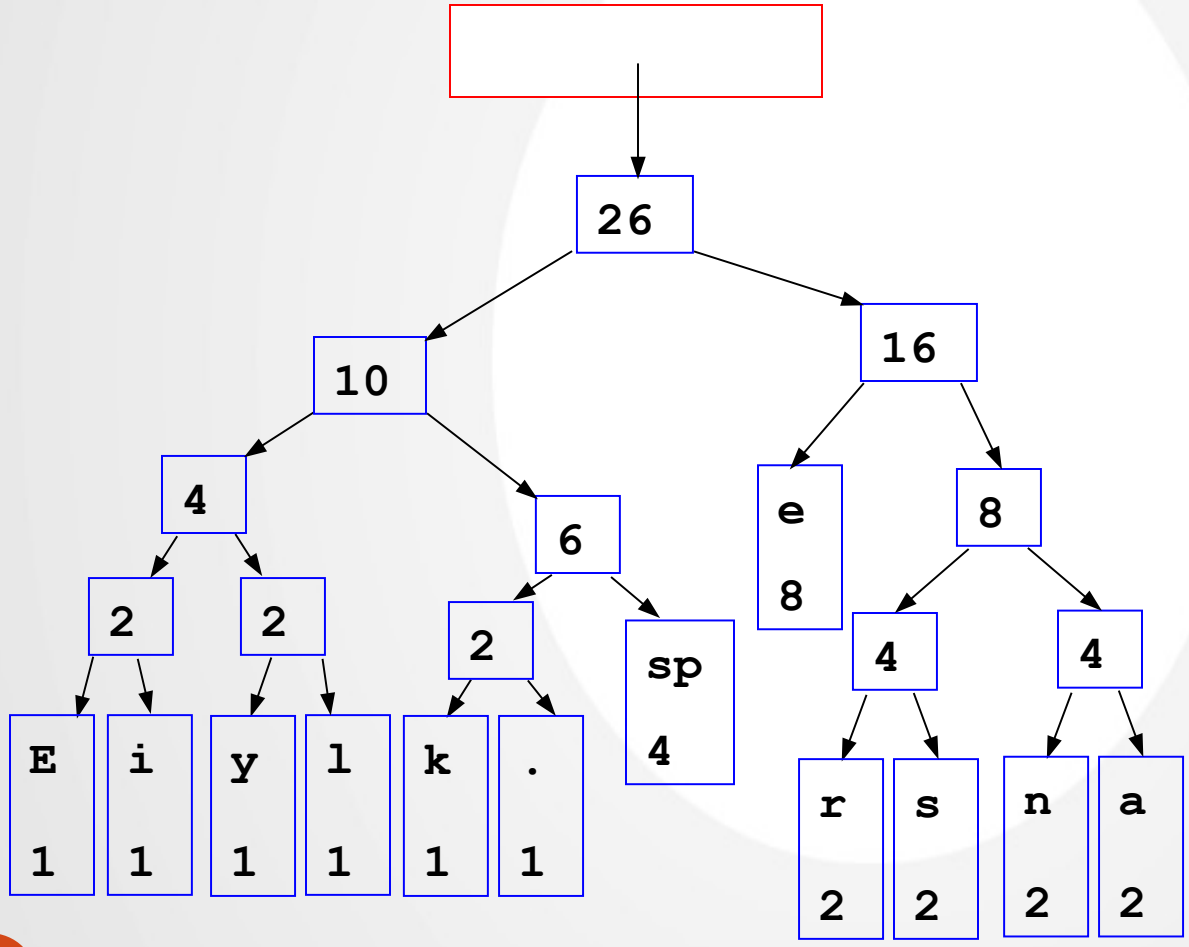
# Criando a Árvore



# Criando a Árvore



# Criando a Árvore



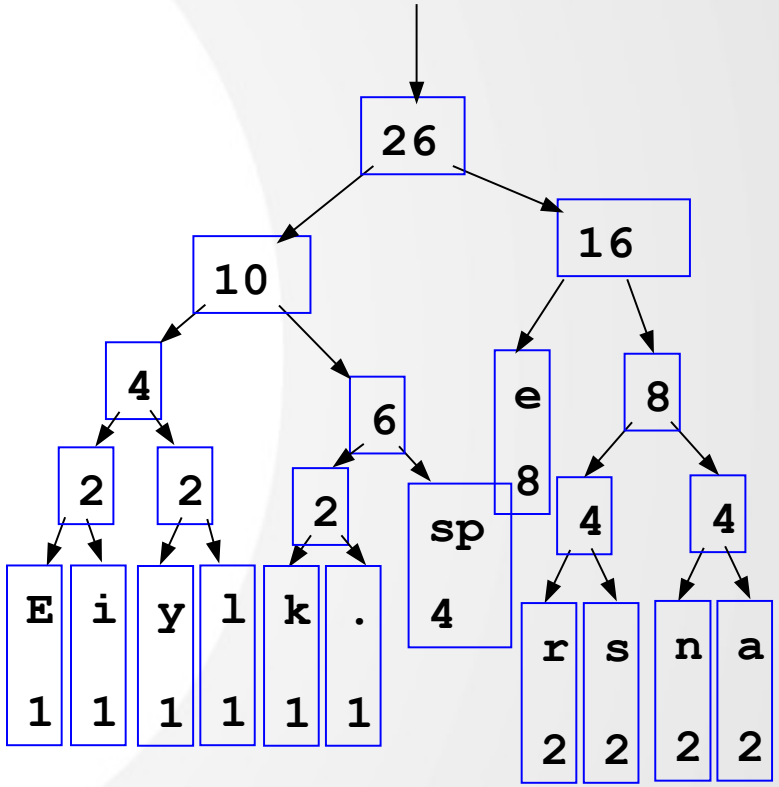

- Após desinfileirar este nó haverá somente um nó a esquerda na fila de prioridade

# Criando a Árvore

Desinfileirar o único nó a esquerda na fila

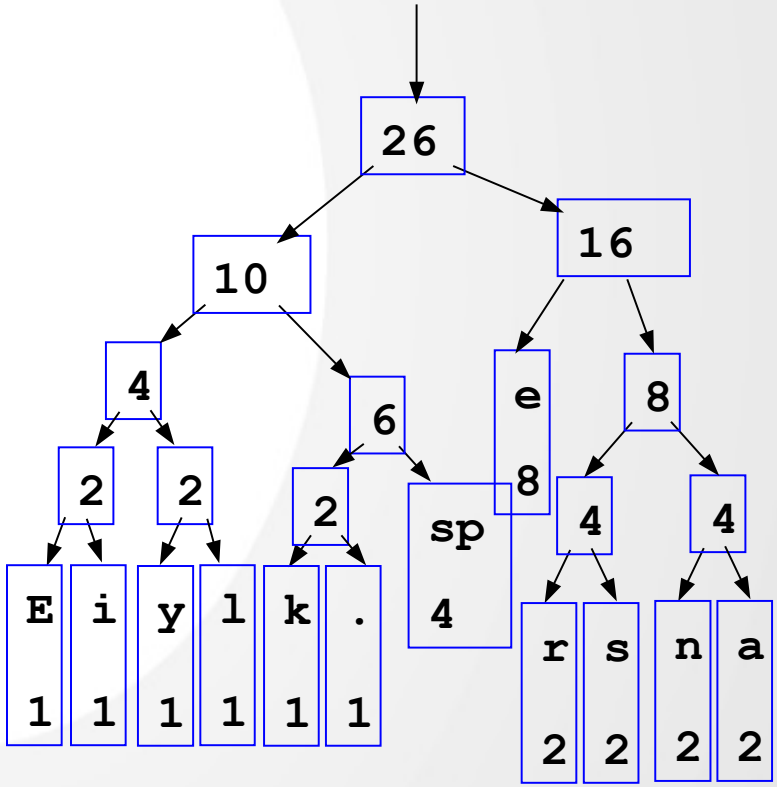
Esta árvore contém o novo codigo para cada caracter

Frequencia do nó raiz deverá ser igual a frequência dos caracteres no texto



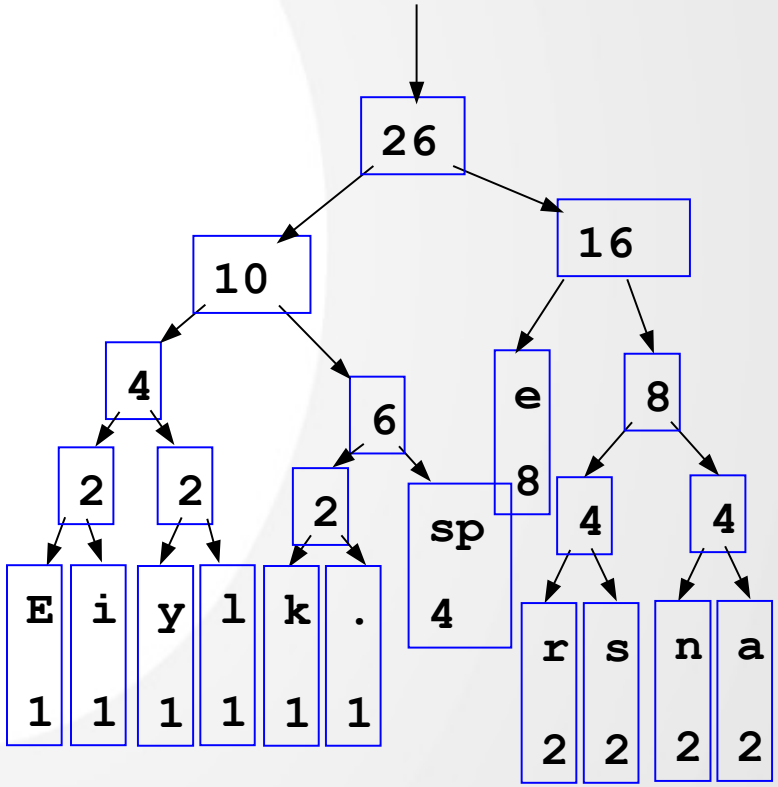
# Codificando o arquivo

- Realizar uma travessia da árvore para obter novas palavras de código
- Indo para a esquerda é 0 indo para a direita é a 1
- Palavra de código só é completada quando um nó folha é atingido



# Codificando o arquivo

| Char | Code |
|---|---|
| E | 0000 |
| i | 0001 |
| y | 0010 |
| l | 0011 |
| k | 0100 |
| . | 0101 |
| space | 011 |
| e | 10 |
| r | 1100 |
| s | 1101 |
| n | 1110 |
| a | 1111 |



# Codificando o arquivo

Escanei o texto e arquivo de código using novo código de palavras

Eerie eyes seen near lake.

```
000010110000110011
100010101101010011
111010111111000110 0
111111010010010 1
```

Por que não é necessário um separador de caracter?

| Char | Code |
|---|---|
| E | 0000 |
| i | 0001 |
| y | 0010 |
| l | 0011 |
| k | 0100 |
| . | 0101 |
| space | 011 |
| e | 10 |
| r | 1100 |
| s | 1101 |
| n | 1110 |
| a | 1111 |

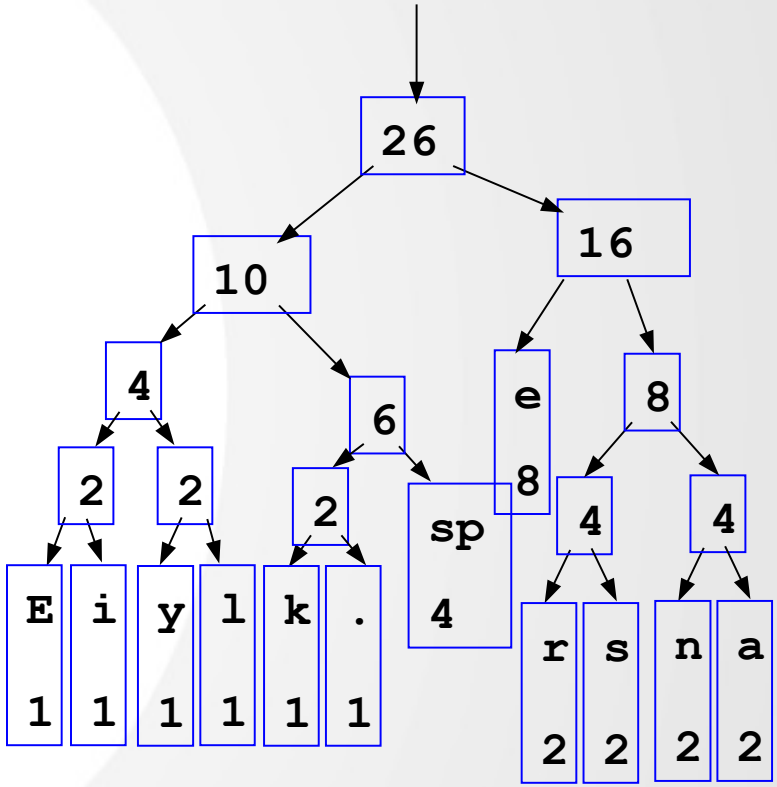
# Decodificando o arquivo

- Como o receptor sabe quais são os códigos?
- Árvore construída para cada arquivo de texto.
  - Considera frequência para cada arquivo
  - Grande sucesso na compressão, especialmente para arquivos menores
- Arvore predeterminada
  - Com base na análise estatística dos arquivos de texto ou tipos de arquivos
- A transmissão de dados baseado em bits contra baseado em byte

# Decodificando o arquivo

- Uma vez que o receptor tem a árvore, ele scanear o stream de bit recebidos
- 0 --> ir para esquerda
- 1 → ir para direira

```
10100011011110111
0111111000110101
```



# Nó da Árvore

- Nó da árvore de Huffman

```
typedef struct NO {
 int simbolo;
 int freq;
 struct NO *fesq;
 struct NO *fdir;
}tNO;
```

| simbolo | freq | fesq | fdir |
|---|---|---|---|

# Heap Mínimo

```
#define TAM 500
typedef struct NO {
 tNO *itens [TAM];
 int fim;
 }HEAP;
```

```
void criar_heap(HEAP *heap);
int vazia_heap(HEAP *heap);
int cheia_heap(HEAP *heap);
int tamanho_heap(HEAP *heap);
void subir(HEAP *heap, int indice);
int inserir_heap(HEAP *heap, NO *no);
void descer(HEAP *heap, int indice);
struct tNO *remover_heap(HEAP *heap);
void imprimir_heap(HEAP *heap);
```

# Árvore Binária (Huffman)

```
#define TAM 500
typedef struct {
  tNO *raiz;
  char codigo[TAM][TAM];
 } ARVORE;
```

```
void inicializar_arvore(ARVORE *arv);
void limpar_arvore_aux(NO *raiz);
void limpar_arvore(ARVORE *arv);
void preordem_aux(NO *raiz);
void preordem(ARVORE *arv);
void criar_arvore(ARVORE *arv, char *msg);
void criar_codigo_aux(ARVORE *arv, NO *no, char *cod,
int fim);
void criar_codigo(ARVORE *arv);
void imprimir_codigo(ARVORE *arv);
void codificar(ARVORE *arv, char *msg, char *cod);
void decodificar(ARVORE *arv, char *cod, char *msg);
```

# Árvore Binária (Huffman)

```
void inicializar_arvore(ARVORE *arv){
   int i;
   for (i=0; i < TAM; i++) {
     arv->codigo[i][0] = '\0';
   }
   arv->raiz = NULL;
}
```

# Árvore Binária (Huffman)

```
void limpar_arvore_aux(tNO *raiz) {
    if (raiz != NULL) {
    limpar_arvore_aux(raiz->fesq);
    limpar_arvore_aux(raiz->fdir);
    free(raiz);
    }
}
```

```
void limpar_arvore(ARVORE *arv) {
    limpar_arvore_aux(arv->raiz);
    arv->raiz = NULL;
}
```

```
void criar_arvore(ARVORE *arv, char *msg) {
  //contando a frequencia (ASCII)
  int i, freq[TAM];
  for (i=0; i < TAM; i++) freq[i] = 0;
  for (i=0; msg[i] != '\0'; i++) {
       freq[(int)msg[i]]++;
  }
  HEAP heap;
  criar_heap(&heap);
  for (i=0; i < TAM; i++) {
     if (freq[i] > 0) {
       NO *pno = (NO *)malloc(sizeof(NO));
       pno->simbolo = i;
       pno->freq = freq[i];
       pno->fesq = NULL;
       pno->fdir = NULL;
       inserir_heap(&heap, pno);
     }
   }
  ...
}
```

```
void criar_arvore(ARVORE *arv, char *msg) {
   ...
   while (tamanho_heap(&heap) > 1) {
     tNO *pfesq=remover_heap(&heap);
     tNO *pfdir=remover_heap(&heap);
     tNO *pnovo = (NO *)malloc(sizeof(NO));
     pnovo->simbolo = '#';
     pnovo->freq = pfesq->freq + pfdir->freq;
     pnovo->fesq = pfesq;
     pnovo->fdir = pfdir;

    inserir_heap(&heap, pnovo);
  }

 arv->raiz = remover_heap(&heap);
}
```

# Criando Código

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

```
void criar_codigo_aux(ARVORE *arv, tNO *no,char *cod, int
fim) {
      if (no != NULL) {
            if (no->fesq == NULL && no->fdir == NULL) {
              int i;
              for (i=0; i <= fim; i++) {
                 arv->codigo[(int)no->simbolo][i]= cod[i];}
              arv->codigo[(int)no->simbolo][fim+1] = '\0';
            } else {
                if (no->fesq != NULL) {
                   fim++;
                   cod[fim] = '0';
                   criar_codigo_aux(arv,no->fesq,cod, fim);
                   fim--;
                 }
                 if (no->fdir != NULL) {
                  fim++;
                  cod[fim] = '1';
                  criar_codigo_aux(arv, no->fdir,cod, fim);
                  fim--;
                }
          }//
      } //
}//
```

# Codificando uma Mensagem

```
void codificar(ARVORE *arv, char *msg, char *cod) {
  int i, j, cod_fim;
  cod_fim = -1; //aponta para a última posição da codificação
  for (i=0; msg[i] != '\0'; i++) {
      //recuperando o código do caractere
      char *pcod = arv->codigo[(int)msg[i]];
      //copiando o código na codificação
      for (j=0; pcod[j] != '\0'; j++) {
         cod_fim++;
         cod[cod_fim] = pcod[j];
      }
   }
   cod[cod_fim+1] = '\0';
}
```

# Decodificando uma Mensagem

```
void decodificar(ARVORE *arv, char *cod, char *msg) {
  int i, decod_fim;
  decod_fim = -1; //aponta para a última posição da decodificação
  NO *pno = arv->raiz;
  for (i=0; cod[i] != '\0'; i++) {
    if (cod[i] == '0') {
       pno = pno->fesq;
    } else if (cod[i] == '1') {
       pno = pno->fdir;
    } else {
      printf("Simbolo codificado errado!\n");
    }
    if (pno->fesq == NULL && pno->fdir == NULL) {
       decod_fim++;
       msg[decod_fim] = pno->simbolo;
       pno = arv->raiz;
    }
  }
  msg[decod_fim+1] = '\0';
}
```

# Exercícios

- Terminar a implementação da compactação/descompactação usando Huffman