MC202GH - ESTRUTURAS DE DADOS

2º Semestre de 2017

Professor: Rafael C. S. Schouery

Monitores: Yulle Glebbyo Felipe Borges (PED)

Caio Vinícius Piologo Véras Fernandes (PAD)

Lab. 07 - Compressão de Textos

Peso 3

1. O Problema

Computadores modernos utilizam a codificação ASCII para representar caracteres e assim armazenar textos. Esta codificação utiliza 8 bits (1 Byte) para representar cada um dos 128 diferentes caracteres e símbolos disponíveis nos teclados padrão para a língua inglesa. É importante notar que ASCII não admite acentos ou outros símbolos presentes em outras linguagens (por exemplo: ç, ñ, e *kanjis*).

Um *tweet* de 140 caracteres, requer um espaço de 1120 bits (140 Bytes) para ser armazenado com esta codificação. Empresas como Twitter devem armazenar bilhões de tweets por anos, portanto, para eles, é importante otimizar a utilização de espaço em disco para armazenar estes dados. É possível utilizar o número de ocorrências dos caracteres em determinadas mensagens para criar uma codificação mais eficiente, que utiliza uma quantidade menor de bits para representar a mesma mensagem.

A **Codificação de Huffman** é um método de compressão de dados que utiliza uma cadeia de bits de tamanho variável para representar cada símbolo. Isto só é possível porque a codificação é garantidamente **livre de prefixo**. Isto quer dizer que a sequência de bits atribuída a um símbolo nunca será prefixo de uma sequência atribuída a outro símbolo qualquer, e a falha a garantir esta propriedade pode resultar em codificações ambíguas. Por exemplo, considere quatro caracteres `a`, `b`, `c` e `d` de codificações de tamanho variável `00`, `01`, `0` e `1` respectivamente. Esta codificação leva a ambiguidade, já que o código atribuído a `c` é um prefixo dos códigos atribuídos a `a` e `b`. Portanto se temos uma sequência de bits `0001` para ser decodificada, todos os seguintes resultados seriam corretos: `ab`, `acd`, `cccd` ou `ccb`.

Neste método, utilizamos o número de ocorrências de cada caractere em um conjunto de mensagens para criar uma árvore binária através da união de dois símbolos de menor número de ocorrências por um pai representando a soma de seus números de ocorrências. Isto é repetido até não haja mais símbolos a serem unidos, e ao fim do processo temos uma árvore binária onde cada símbolo é uma folha, e os símbolos de maior ocorrência estão nos nós internos da árvore. Podemos usar esta árvore para criar uma tabela com os codificação de cada símbolo presente nos textos através de um percurso da

raíz até a folha correspondente, anotando 0 sempre que tomar uma aresta levando a um filho à esquerda e 1 caso tome uma aresta que leve a um filho à direita.

1.1. Criando a Árvore

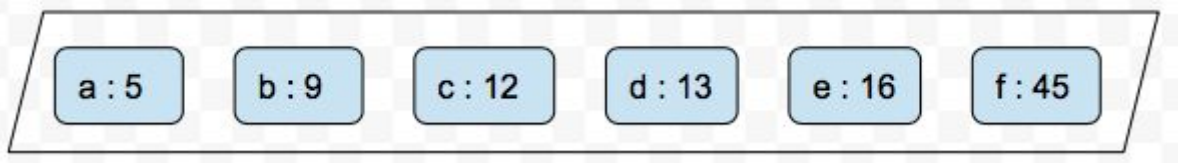
Suponhamos que lemos um texto com a seguinte tabela de ocorrências:

Símbolo	Número de Ocorrências
f	45
e	16
d	13
c	12
b	9
a	5

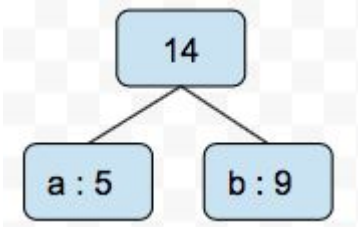
Construímos a árvore da seguinte maneira:

1. Cria-se um nó folha para cada símbolo e insere-os em uma fila de prioridades ordenada de maneira não-decrescente pela seus números de ocorrências;
2. Extrai-se dois nós de menor número de ocorrências da fila de prioridades;
3. Cria-se um nó interno com número de ocorrências igual a soma do número de ocorrência dos nós extraídos. Conecta-se o primeiro nó extraído da fila de prioridades como filho esquerdo do novo nó e o segundo como filho direito do novo nó;
4. Repita passos 2 e 3 até que a fila de prioridades tenha apenas um elemento. Este elemento é a raiz da árvore final.

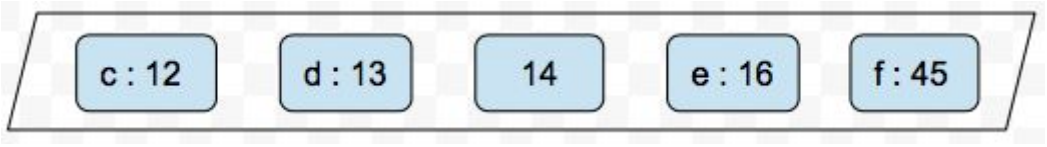
Inicialmente, teremos a fila de prioridades da seguinte forma (cada elemento é um nó):



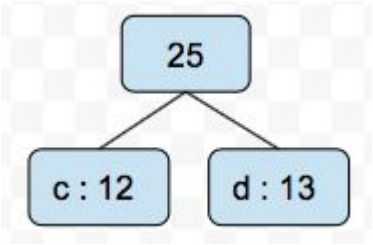
Então extraímos os elementos de menor número de ocorrências e formamos uma árvore a partir deles:



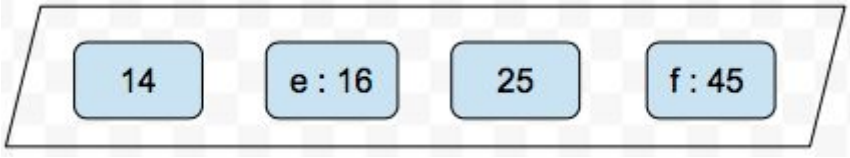
Após esta operação, teremos a fila de prioridades na seguinte forma:



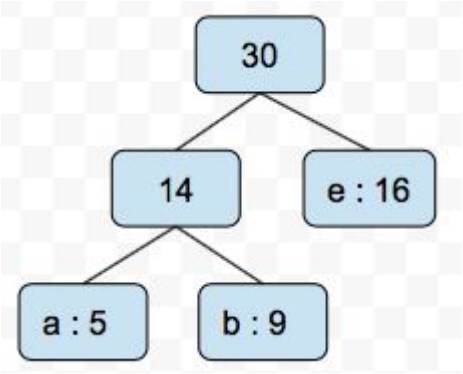
Desta vez formamos uma nova árvore a partir dos nós `c` e `d`:



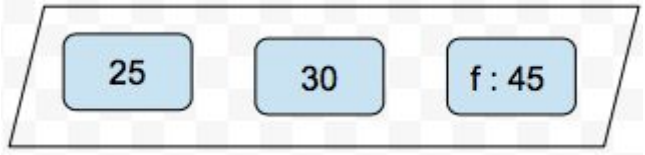
A fila de prioridade agora possui quatro nós:



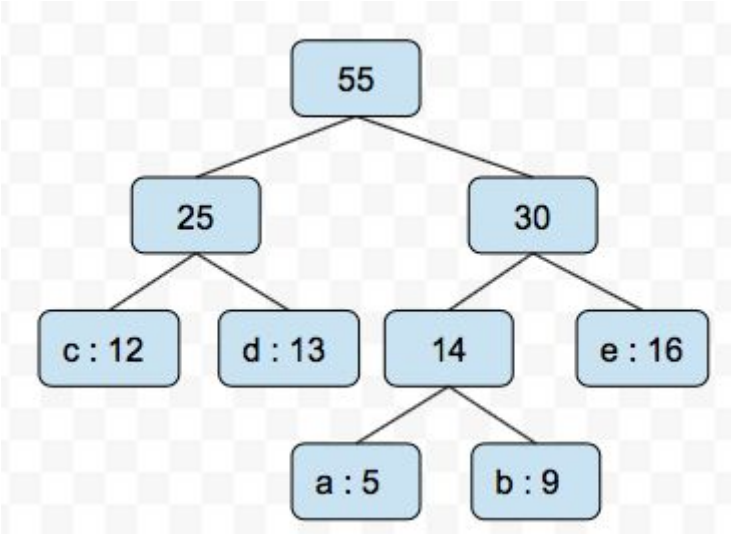
Ao unir os nós `14` e `e` temos a árvore:



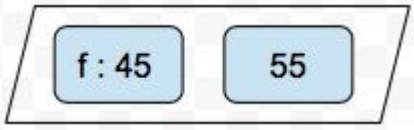
Restando a fila de prioridade na forma:



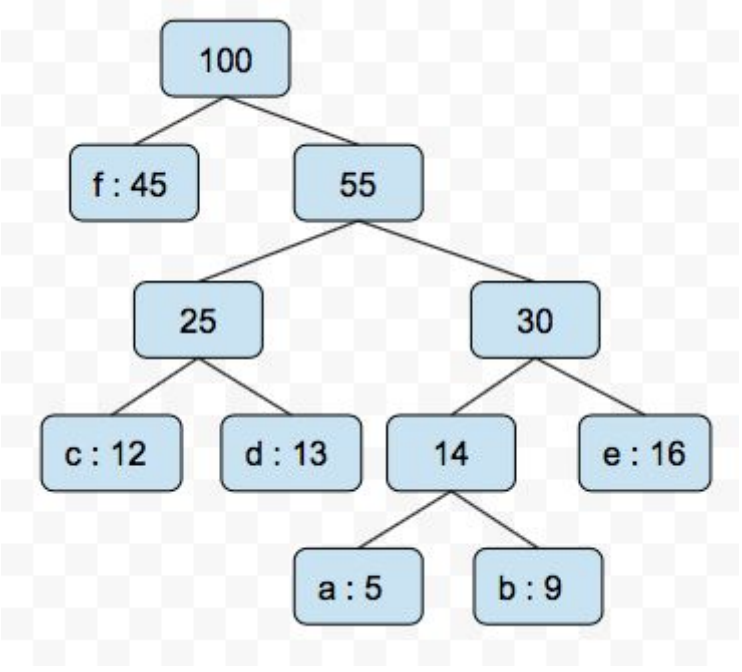
Ao unir as subárvores de `25` e `30` temos:



Por fim, a fila possui apenas dois elementos:



Ao unir os dois últimos elementos da fila, temos a seguinte árvore como resultado final:

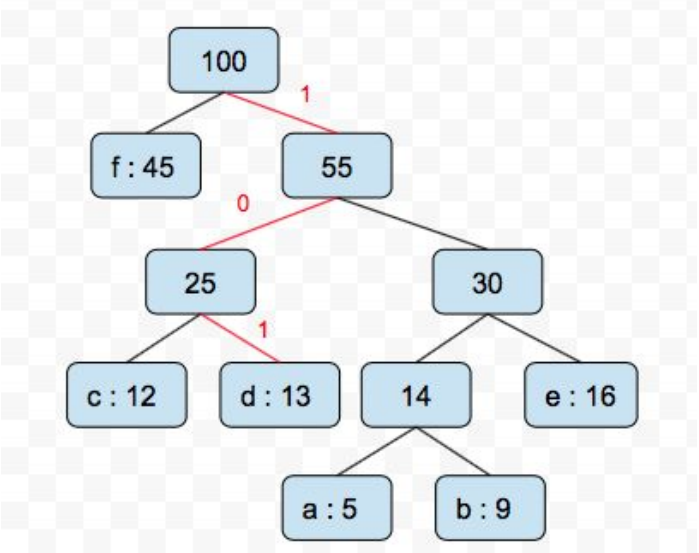


Para tornar o algoritmo mais determinístico e consequentemente mais fácil de avaliar, pedimos que **insiram os nós na fila de prioridade de acordo com a ordem que eles aparecem na tabela ASCII**. Assim, teremos que o primeiro caractere a ser inserido na fila de prioridade deve ser o espaço branco (ASCII 32) e o último o til (~, ASCII 126). Com isso podemos garantir que a fila de prioridade retornará sempre a mesma ordem de símbolos, mesmo em casos de empate no número de ocorrências.

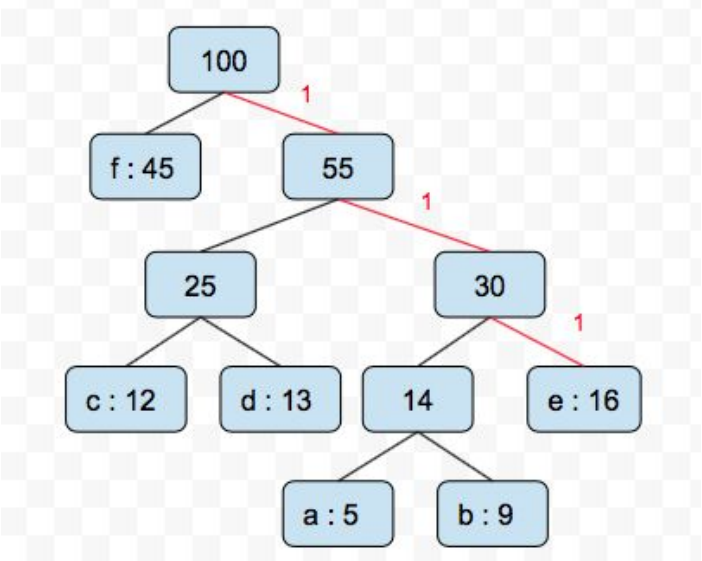
1.2. Codificando Uma Sequência

Para codificar uma mensagem é necessário percorrer a árvore de símbolos para encontrar a sequência de bits correspondente a cada caractere. A codificação de cada símbolo é dada através do percurso da raíz até a folha correspondente, onde cada aresta levando a um filho a esquerda representa um 0 na cadeia, e as arestas levando a um filho a direita corresponde a um 1.

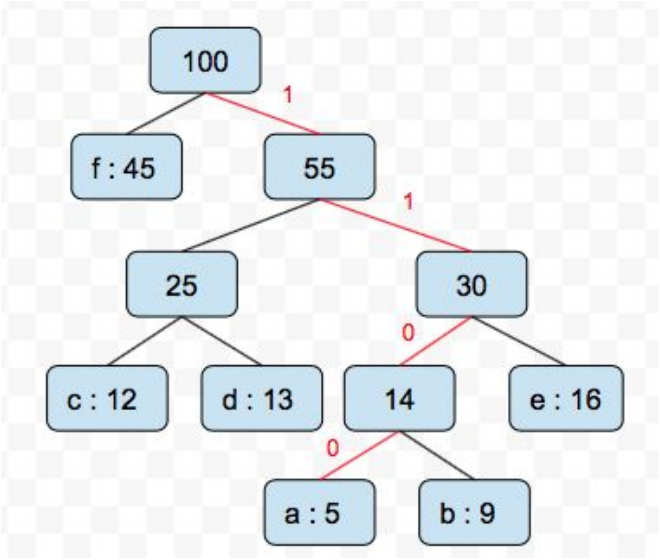
Para ilustrar, considere um exemplo com a entrada "`deadbeef`" para ser codificada:
Codificamos a letra `d` como a cadeia `101`



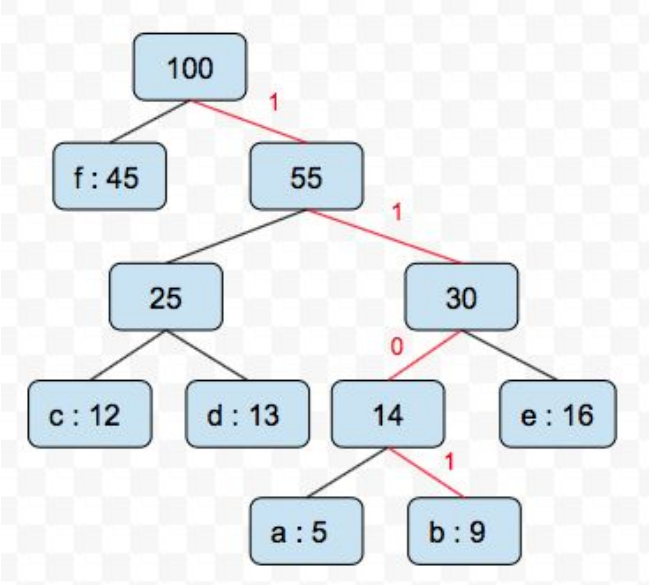
A letra `e` como a cadeia de bits `111`:



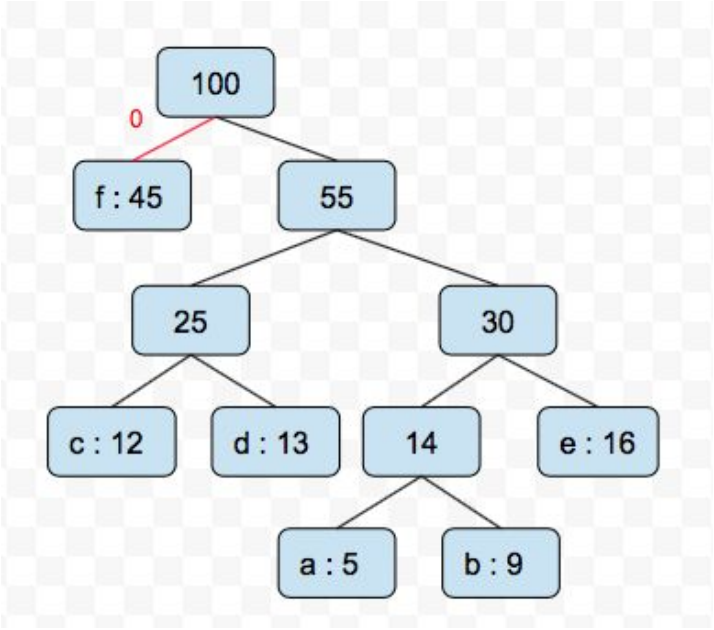
A letra `a` como a cadeia `1100`:



A letra `b` como a cadeia de bits `1101`:



E por fim, a letra `f` como o bit `1`:

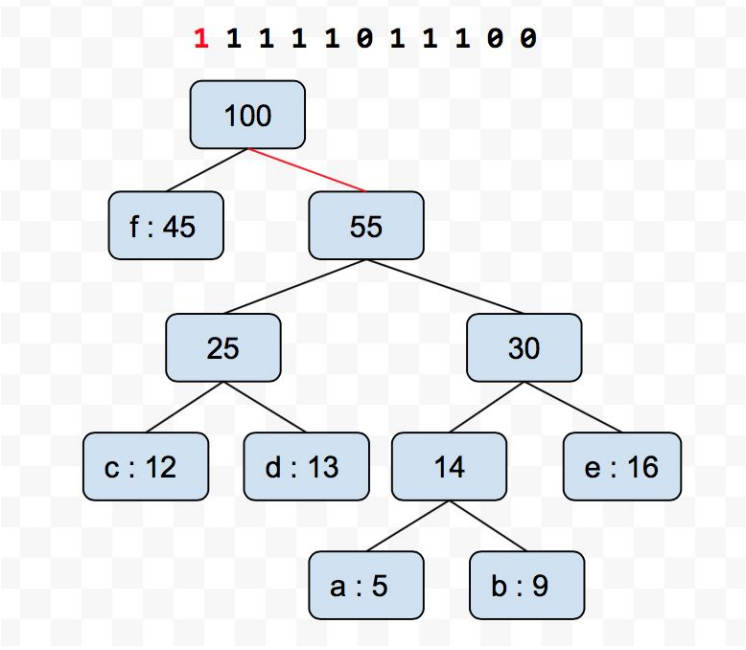


Assim, a mensagem "`deadbeef`" (64 bits, 8 Bytes) é codificada como `101111110010111011111110` (24 bits, 3 Bytes) alcançando uma taxa de compressão de aproximadamente 63%.

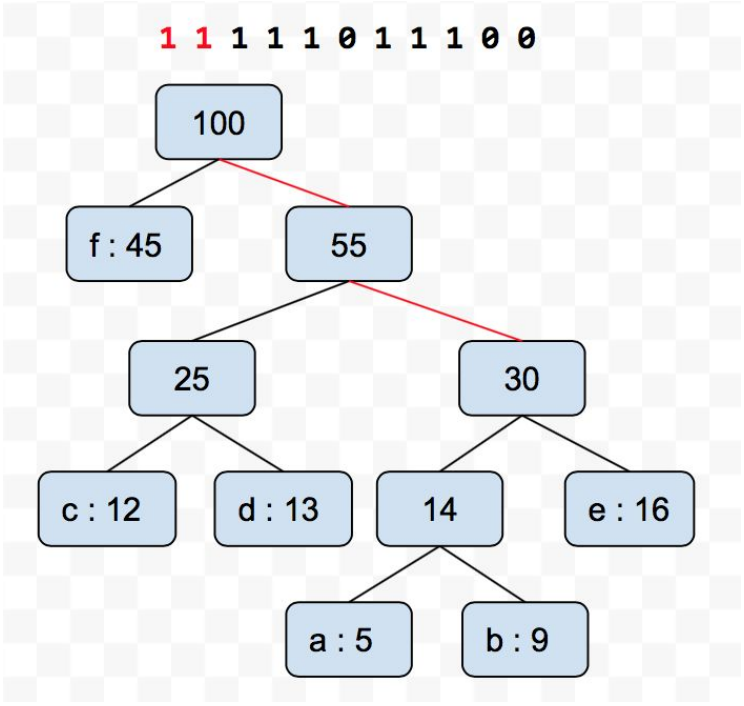
1.3. Decodificando Uma Sequência

Para decodificar uma sequência de bits é necessário realizar uma série de percursos na árvore de símbolos de acordo com os bits lidos. Um bit 0 representa um movimento para o filho à esquerda, enquanto um bit 1 representa um movimento para o filho à direita. Quando o percurso atinge uma folha, o processo é reiniciado a partir da raíz.

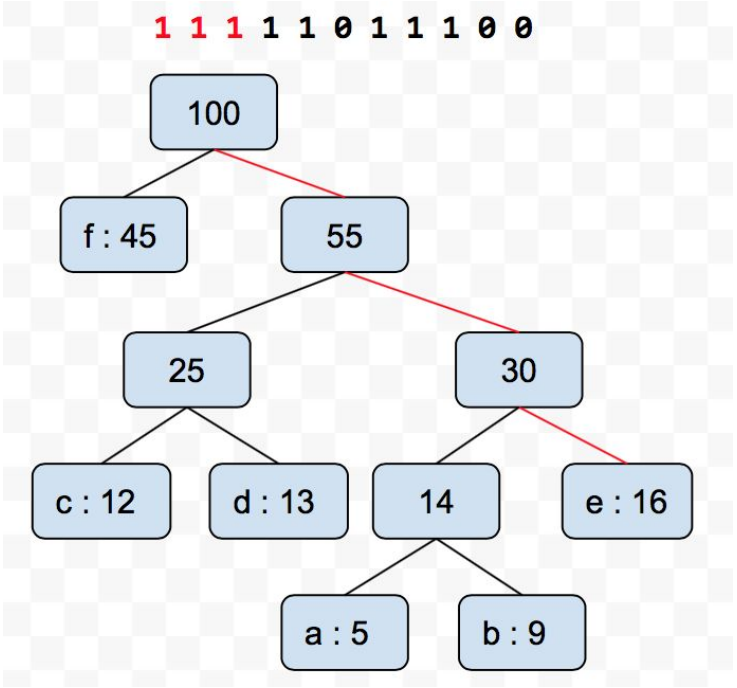
Para ilustrar, considere a sequência `11111011100` a ser decodificada
O primeiro bit, indica para acessarmos o filho direito da raiz:



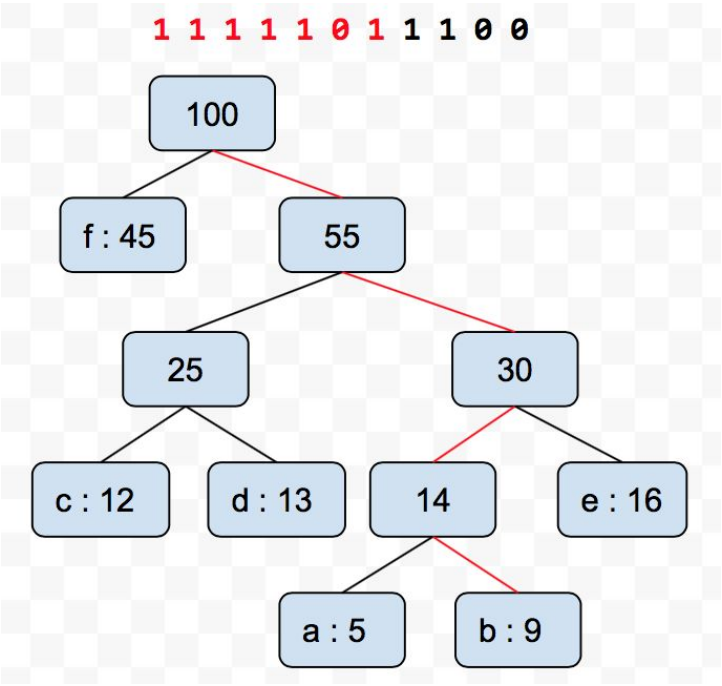
A partir deste nó, lemos outro bit 1, indicando para irmos para o filho da direita novamente:



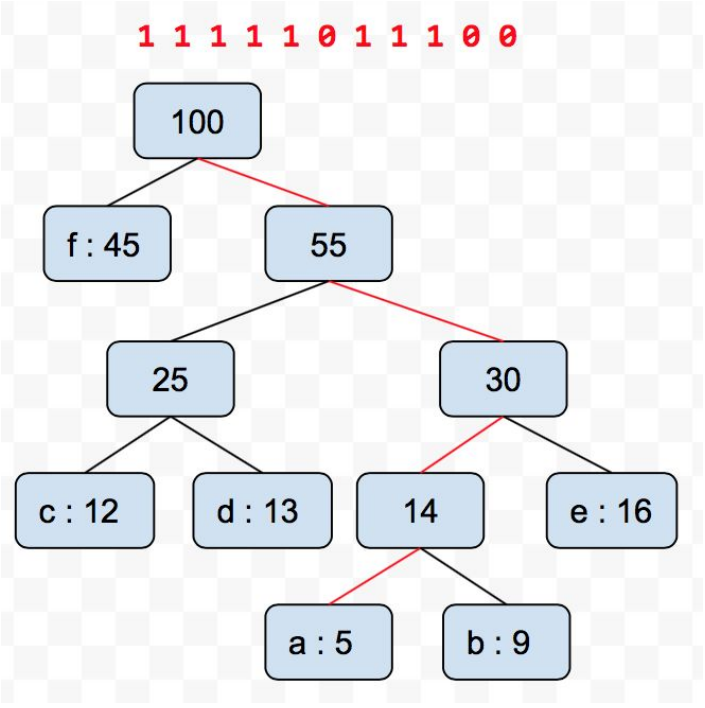
Do próximo nó, seguimos outro 1 para chegar no nó folha correspondente ao caractere e, que deve ser adicionado a saída para que o processo possa continuar a partir da raíz.



Começando novamente da raíz, é possível notar que os quatro próximos bits nos levarão ao nó folha correspondente ao caractere b.



Como um nó folha foi atingido, o processo recomeça a partir da raíz. De maneira semelhante, os quatro próximos bits nos levam ao nó folha que corresponde ao caractere a.



Neste ponto, temos que um nó folha foi atingido e todos os bits da sequência de entrada já foram processados, portanto o algoritmo termina. Assim, temos como resultado da decodificação de acordo com a árvore obtida a mensagem eba.

1.4. Objetivo

Nesta tarefa, você deverá criar um programa que implementa a Codificação de Huffman para criar uma árvore binária de símbolos a partir de um conjunto de tweets dados

como entrada, e então utilizar esta árvore para codificar outros tweets e decodificar sequências de bit transformando-os em tweets inteligíveis.

Para a implementação do algoritmo de criação da árvore, é necessário o uso de uma fila de prioridades, que será fornecida como material suplementar desta tarefa (ver a descrição desta estrutura no arquivo fp.h). Você **obrigatoriamente deve utilizar a fila de prioridades fornecida**. Esta será uma fila de nós de uma árvore binária, portanto é necessário que o nó da árvore binária a ser implementada obedeça a definição feita em fp.h, e que fp.h seja incluído corretamente em sua implementação da árvore binária:

```
typedef struct No {
  char simbolo;
  int freq;
  struct No *esq, *dir;
} No;
typedef No * p_no;
```

Além disso, para que a decodificação funcione para qualquer mensagem que não esteja no conjunto inicial de tweets, assumimos que todos os símbolos imprimíveis da tabela ASCII (ver Wikipedia) estão na árvore de símbolos, mesmo que seus números de ocorrência sejam 0. Desta forma, independente da entrada, **as árvores de símbolos devem conter todos os 94 caracteres imprimíveis da tabela ASCII, do espaço em branco (ASCII 32) até o til (ASCII 126)**.

2. Entrada

Ao iniciar o programa, deve-se criar a árvore de símbolos, portanto lê-se um inteiro **d** representando o número de mensagens utilizadas para criar a árvore. Em seguida são dados como entrada **d** mensagens de até 140 caracteres (sem contar o `\n` e `\0`), cada uma em uma linha (em outras palavras, separadas por um `\n`). Este processo não resulta em uma saída, ou seja, nada deve ser impresso na tela.

Uma vez que a árvore for criada, o programa deve entrar em um loop de comandos para codificar e decodificar mensagens. Estes comandos e suas entradas são descritos na tabela abaixo.

Para ler as mensagens, utilize o comando `fgets(msg,142,stdin)` onde `msg` é a variável onde a mensagem será salva, `142` delimita o tamanho máximo da mensagem (incluindo um possível `\n` e `\0`) e `stdin` indica que a leitura será feita da entrada padrão (teclado ou arquivo caso haja redirecionamento). É importante notar que diferente do comando `scanf`, o `fgets` inclui o caractere `\n` ao fim das mensagens lidas, portanto vocês devem tratar a entrada para ignorar este caractere ao processar uma mensagem.

Por este mesmo motivo, caso você o `fgets` logo após uma chamada de `scanf`, a mensagem não será lida com sucesso pois o `fgets` irá capturar o `\n` que o `scanf` não capturou. Para resolver este problema, é necessário descartar o `\n` da entrada padrão após qualquer chamada de `scanf`, o que pode ser feito com a seguinte linha:

```
while ((c = getchar()) != '\n' && c != EOF);
```

ID	Nome	Descrição
1	Codificar	Dada uma mensagem na forma de um tweet (máximo 140 caracteres, sem contar `\n` e `\0`) contendo apenas caracteres da parte imprimível da tabela ASCII (ver Wikipedia), deve-se imprimir uma sequência de bits que corresponde a mensagem de entrada, de acordo com a árvore de símbolos criada ao início do programa. Considere que a codificação de um caractere nunca terá mais que 128 bits
2	Decodificar	Dada uma sequência de bits (sem espaços), imprime a mensagem representada pela sequência de entrada, de acordo com a árvore de símbolos criada ao início do programa
0	Finalização	Saia do sistema, certificando-se que toda a memória alocada foi liberada.

Exemplo de entrada:

```
1
fffffffffffffffffffffffffffffffffffffffffffffffeeeeeeeeeeeeeeeeddddd
ddddddddccccccccccccbbbbbbbbbaaaaa
1
deadbeef
2
11111011100
0
```

Obs: O resultado do exemplo acima pode diferir de sua implementação, uma vez que ele foi construído a mão para facilitar seu entendimento, logo a árvore deste exemplo não inclui caracteres que não aparecem na mensagem inicial. Para exemplos mais detalhados ver os casos de teste abertos no SuSy.

3. Saída

A tabela abaixo descreve o formato esperado da saída de cada comando definido na seção anterior.

ID	Saída	Comentário
1	SEQUENCIA_DE_BITS Ex: `11111011100`	Imprime-se apenas uma sequência de bits sem separação alguma entre eles e terminados com um `\n`

2	MENSAGEM Ex: `Hello, world!`	Imprime-se uma mensagem de no máximo 140 caracteres, todos no intervalo imprimível da tabela ASCII (ver Wikipedia), e terminada com um `\n`
0	Sistema encerrado.	Antes de encerrar o sistema, é necessário liberar toda a memória alocada

Exemplo de saída:

```
101111110010111011111110
eba
Sistema encerrado.
```

Obs: A saída acima pode diferir de sua implementação, uma vez que as entradas foram construídas a mão para facilitar seu entendimento, logo a árvore deste exemplo não inclui caracteres que não aparecem na mensagem inicial. Para exemplos mais detalhados ver os casos de teste abertos no SuSy.

4. Informações

- Este laboratório possui **peso 3**.
- **Não** há um número máximo de submissões.
- No início de cada arquivo a ser submetido, insira um comentário com seu nome, RA e uma breve descrição do conteúdo do arquivo.
- Apenas comentários no formato `/* comentário */` serão aceitos. Comentários com `//` serão acusados como erros pelo SuSy.
- Os cabeçalhos permitidos são:
  - `stdio.h`
  - `stdlib.h`
  - `string.h`
- Os arquivos **fp.c** e **fp.h** serão dados como material de apoio (em arquivos auxiliares no SuSy) para que você consiga testar o programa em sua máquina. Entretanto, estes arquivos não deverão ser submetidos no SuSy, pois eles já estarão no sistema.
- A submissão da sua solução deverá conter múltiplos arquivos:
  - **arvore_binaria.h**: interface da estrutura de árvore binária (apenas a declaração das funções)
  - **arvore_binaria.c**: implementação da estrutura de árvore binária

    - **lab07.c**: cliente que utiliza a estrutura
- Será disponibilizado um **Makefile** para este trabalho na página do laboratório:
    - Para compilar seu projeto, basta navegar até o diretório do projeto e utilizar o comando ‘make’ no terminal do Linux.
    - Veja mais instruções na página da disciplina.
- Toda a memória alocada deve ser liberada adequadamente ao fim do programa. Isso pode te ajudar a evitar a ocorrência de falhas relacionadas a memória.
- Indente corretamente todo o seu código. Escolha entre espaços ou tabs para indentação e seja coerente em todos os arquivos. Indentação é fundamental para a legibilidade do seu código, e ajuda no entendimento do fluxo de controle do seu programa.

5. Critérios de Avaliação

- Seu código deverá passar por todos os casos de teste do SuSy definidos para este laboratório. Caso positivo, utilizaremos os seguintes critérios adicionais:
    - A **não utilização/implementação da estrutura árvore binária** resultará em **nota zero**
    - A não utilização da estrutura de **fila de prioridades** fornecida como material de apoio resultará em **nota zero**
    - Falha na implementação do comportamento esperado para árvores binárias acarretará em uma penalização
    - Utilização de bibliotecas ou funções não permitidas pelo enunciado resultará em uma **penalização severa** na nota