Codificação de Huffman

Estruturas de Dados Básicas II Instituto Metrópole Digital

2019.2

1 Introdução

Hoje em dia, muitos arquivos são enviados através da internet (fotos, vídeos, áudio e texto). Por exemplo, **bilhões** de mensagens são trocadas todos os dias em mensageiros eletrônicos e mais de **65 anos** de duração são enviados todos os dias à plataformas especializadas em vídeo. Para transmitir e armazená-los de forma eficiente, geralmente comprimimos os arquivos antes de enviarmos ou armazenarmos e descomprimimos eles quando recebemos ou queremos acessar essas informações.

Esta compressão pode ocorrer de duas maneiras: com perdas ou sem perdas (lossless). Na compressão com perdas, a informação original não consegue ser recuperada por completo e o resultado da descompressão apresenta diferenças em relação ao conteúdo original. Isto é muito utilizado na compressão de áudio (MP3, por exemplo) e vídeo (MKV, por exemplo). Já na compressão sem perdas, a informação original é recuperada integralmente.

Para compressão de texto não é aceitável a perda de caracteres e geralmente esta compressão ocorre sem perdas. Neste trabalho iremos investigar o funcionamento de um compressor de texto onde a informação será armazenada, na maioria das vezes, com uma quantidade menor de bits do que o armazenamento de texto convencional.

2 Codificação de Huffman

Neste projeto você utilizará uma *heap* e uma árvore binária com implementação própria para criar um programa para comprimir e descomprimir arquivos de texto utilizando um algoritmo chamado de “Codificação de Huffman”. Este algoritmo foi desenvolvido em 1952 por David Huffman e, apesar de ser simples, variações de sua

implementação ainda são utilizadas hoje em dia em diversos dispositivos como TVs digitais, faxes e modens.

Normalmente informações textuais são armazenadas utilizando um padrão chamado *ASCII* (*American Stardand Code for Information Interchange*). Neste padrão cada caractere é representado utilizando 1 byte (8 bits) e cada valor possível (de 0 a 255) representa um caractere. A tabela abaixo apresenta alguns dos valores utilizados na tabela *ASCII*.

Caractere	ASCII	ASCII em Binário
a	97	01100001
b	98	01100010
c	99	01100011
e	101	01100101
z	122	01111010

A ideia do algoritmo é ao invés de utilizar a tabela *ASCII*, criar uma tabela própria onde os caracteres mais utilizados são representados por menos bits do que os caracteres mais utilizados. A ideia se baseia no fato de que se um texto não utiliza todos os 256 caracteres, não é necessário utilizar 8 bits para cada um dos caracteres da mensagem. Se, por exemplo, a letra "a" é mais utilizada em uma mensagem, eu posso atribuir um código que utiliza poucos bits para a letra "a". Já os caracteres menos utilizados recebem código com uma quantidade maior de bits, pois são utilizados com menor frequência.

Os passos necessários para implementar a compressão utilizando a codificação de Huffman são os seguintes:

1. **Contar a frequência de utilização dos caracteres.** Ler o conteúdo de uma mensagem e conte quantas vezes cada caractere aparece no texto.
2. **Criar uma árvore de codificação.** Construir uma árvore onde binária onde cada nó representa cada caractere. Uma fila de prioridade deve ser utilizada para ajudar na criação da árvore.
3. **Criar uma tabela de codificação.** Percorrer a árvore para saber qual o código de cada caractere.
4. **Codificar o texto.** Reexaminar o arquivo e, para cada caractere, substituir seu caractere por seu respectivo código binário no arquivo de saída do programa.

2.1 Contar a frequência de utilização dos caracteres

O primeiro passo é ler o conteúdo de um arquivo caractere por caractere e contar a frequência que cada um deles aparece no arquivo. Como exemplo, suponha que

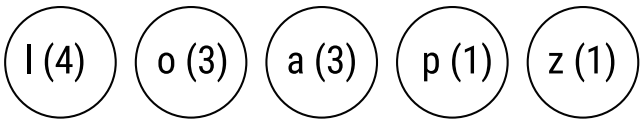
temos um arquivo chamado `teste1.txt` cujo conteúdo é: `lollapalooza`. Este arquivo ocupa 12 bytes (96 bits) de informação.

Podemos utilizar um mapa — também conhecido como dicionário — para armazenar os caracteres e a quantidade que eles aparecem no texto. É importante salientar que **todos** os caracteres devem ser considerados, incluindo espaços, pontuação e quebras de linha. A tabela abaixo mostra o dicionário com as frequências dos caracteres para o arquivo `teste1.txt`.

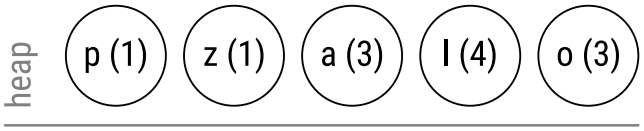
Caractere	Quantidade
l	4
o	3
a	3
p	1
z	1

2.2 Criar uma árvore de codificação

O segundo passo é criar uma árvore binária que será utilizada para computar o código de cada um dos caracteres do texto. Cada caractere com sua respectiva frequência será um nó.



Em seguida devemos colocar estes nós em uma fila de prioridade, onde os elementos com menor frequência possuem uma prioridade maior (*min heap*).



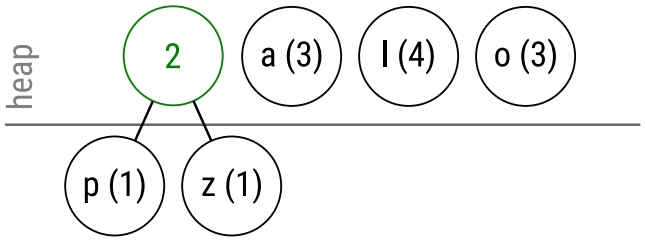
Após todos os caracteres estarem na fila, devemos construir uma árvore binária repetindo o seguinte procedimento até a fila não possuir mais elementos:

1. Remover os dois primeiros[1] elementos da fila e juntá-los em uma nova árvore cuja raiz possui como frequência a soma das frequências dos dois nós. Os dois elementos devem ser nós filhos desse novo nó. O primeiro elemento removido deve ser o filho da esquerda. O segundo elemento removido deve ser o filho da direita.

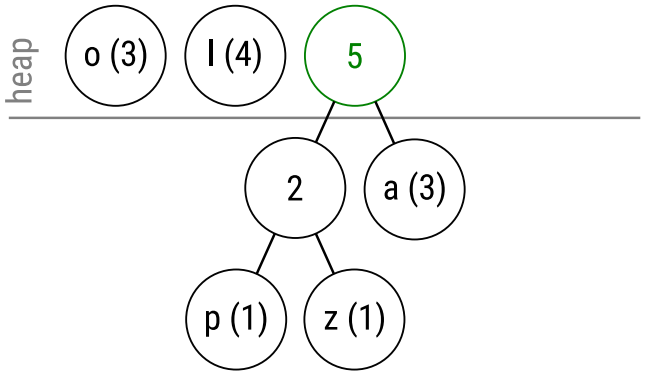
2. Adicionar esse novo nó à fila de prioridades na sua posição adequada

3. Este processo é repetido até não haver mais elementos na fila e todos os elementos pertencerem à uma única árvore binária cuja raiz contem todos os outros nós como filho. O nó raiz será considerado nossa árvore de codificação.

As imagens abaixo detalham o processo passo a passo para o arquivo de exemplo. Os nós “p (1)” e “z (1)” são removidos da fila e são adicionados como filhos de um novo nó que possui prioridade 2. Este nó é adicionado na fila:

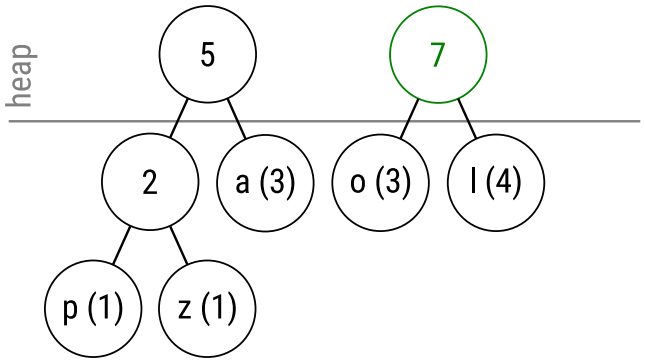


Os nós “2” e “a (3)” são removidos da fila e são adicionados como filhos de um novo nó que possui prioridade 5. Este nó é adicionado na fila:

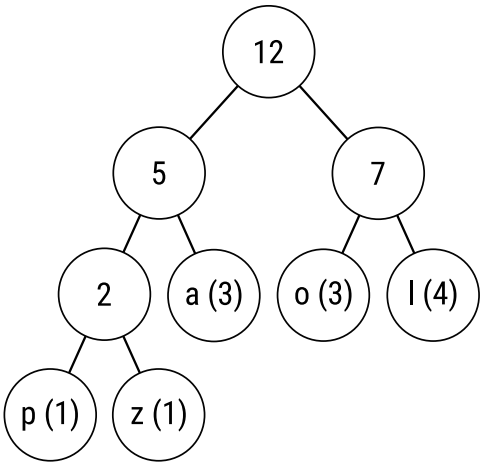


[1] A fila pode possuir apenas um elemento. Seu código deve checar esta condição.

Os nós “o (3)” e “l (4)” são removidos da fila e são adicionados como filhos de um novo nó que possui prioridade 7. Este nó é adicionado na fila:



Por último, os dois nós restantes, “5” e “7”, são removidos da fila e são adicionados como filhos de um novo nó que possui prioridade 12. Este nó é o nó raiz da nossa árvore de codificação:

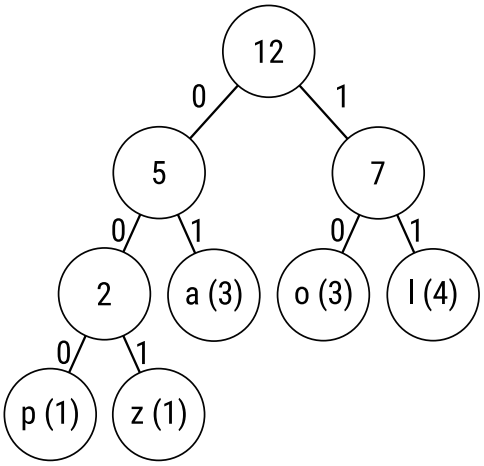


Podemos representar cada nó como sendo um objeto de uma classe `Node` como a apresentada abaixo.

```
class Node {
  private int letter;
  private int count;
  private Node left;
  private Node right;
  //...
  //restante das propriedades e métodos
}
```

2.3 Criar uma tabela de codificação

Cada um dos caracteres é uma folha na nossa árvore. Devemos atribuir o valor 0 às arestas dos filhos à esquerda e o valor 1 às arestas dos filhos à direita. O código de um caractere é a concatenação dos valores das arestas da raiz até o nó folha correspondente. A imagem abaixo ilustra este procedimento.



É necessário criar uma tabela de codificação relacionando um código à seu respectivo caractere. Novamente devemos utilizar um dicionário para esta etapa. A tabela abaixo mostra cada caractere da árvore acima com seu respectivo código binário.

Caractere	Código
l	11
o	10
a	01
p	000
z	001

2.4 Codificar o texto

Com a tabela de codificação gerada, precisamos revisitar o arquivo e, para cada caractere existente, adicionar seus bits correspondentes em um arquivo de saída.

l	o	l	l	a	p	a	l	o	o	z	a
11	10	11	11	01	000	01	11	10	10	001	01

Nesta etapa, você deve gerar um novo arquivo utilizando a cadeia de bits gerada para o arquivo. Note que **não** é pra gerar um arquivo de texto contendo zeros e uns e sim, um arquivo binário cujo os bits possuem tal valores. Este arquivo deve possuir a extensão **.edz**.

Também é necessário armazenar a tabela de codificação para podermos, futuramente, recriar o arquivo com seu conteúdo original. Esta arquivo deverá ser gerado utilizado um arquivo de texto comum, onde cada linha possui o caractere seguido pelo seu codigo. Este arquivo deve possuir a extensão **.edt**. A tabela de símbolos para o arquivo `teste1.txt` deve possuir o seguinte conteúdo:

```
l11
o10
a01
p000
z001
```

Em Java, podemos utilizar a classe `java.util.BitSet` para manipularmos bits. Consultem a documentação da classe para saber como utilizá-la.

Atenção! A menor unidade de armazenamento possível é um `byte` (8 bits). Não é possível armazenar uma quantidade de bits arbitrária. Podemos apenas armazenar uma quantidade de bits que são múltiplos de 8. No caso do `teste1.txt`, nossa codificação utilizou 26 bits. Então, o arquivo deverá possuir 32 bits, no mínimo. Seu programa não deve considerar os bits da posição 27 a posição 32, pois eles não fazem parte do conteúdo original. Uma forma fácil de resolver isso é adicionar na fila de prioridades um nó extra para representar o fim do arquivo (EOF) com a frequência 1 e utilizá-lo no final da codificação para indicar que o seu programa não deve ler mais bits. No nosso exemplo, ao invés de codificar apenas `lollapalooza` você codificaria `lollapaloozaEOF`, onde EOF pode ser qualquer valor que não tenha correspondência na tabela ASCII[2].

2.5 Recuperando o arquivo

Com o arquivo comprimido e com a tabela é possível recuperar a mensagem original. Primeiro devemos ler o arquivo da tabela, e gerarmos novamente a tabela de codificação no nosso programa. Em seguida, devemos ler o arquivo e transformá-lo em um `BitSet` novamente. Em seguida, devemos ler bit a bit e ir concatenando seu valor até encontrarmos um correspondente na nossa tabela de codificação. Devemos repetir este processo até encontrarmos o `EOF`.

[2]Por este motivo que na nossa classe `Node` utilizamos um inteiro ao invés de um `char` para a propriedade `letter`. Ao utilizar um inteiro, podemos criar um nó com um valor maior que 255, que é o máximo permitido para o `char` (e o máximo da tabela ASCII).

- Lemos o primeiro bit e não temos uma chave correspondente na nossa tabela.
  1110111101000011110100 0101
- Lemos o segundo bit e encontramos o valor "l" na nossa tabela. Adicionamos este caractere ao nosso *buffer* de saída.
  1110111101000011110100 0101
- Resetamos nossa chave e começamos novamente a partir do proximo bit.
  1110111101000011110100 0101
- Como não encontramos um caractere correspondente, concatenamos o caractere e continuamos a busca na tabela. Neste caso encontramos o "a" e devemos concatená-lo ao *buffer* de saída.
  1110111101000011110100 0101
- Repetimos este processo até encontrarmos o `EOF`.

3 Descrição

Neste trabalho você deve criar um programa para codificar e decodificar um arquivo de texto utilizando a codificação de Huffman. Seu programa deve receber os arquivos como argumentos de linha de comando. **Programas que colocarem o nome do arquivo dentro do código ou solicitarem o nome do arquivo através de um *prompt* (utilizando a classe `java.util.Scanner`, por exemplo) serão ignorados completamente.**

Seu programa deve funcionar com a seguintes interfaces de interação:

```
./programa.jar compress arquivo.txt arquivo.edz arquivo.edt
./programa.jar extract arquivo.edz arquivo.edt arquivo.txt
```

onde:

- `programa.jar` é o arquivo executável gerado pelo seu código.
- `compress` e `extract` são as operações possíveis.
- `arquivo.txt` é o caminho do arquivo de texto de entrada no `compress` e de saída no `extract`.
- `arquivo.edz` é o caminho do arquivo que contém (ou conterá) o arquivo comprimido.

- `arquivo.edt` é o caminho do arquivo que contém (ou conterá) a tabela de codificação.

Além deste requisito de interface, também será necessário utilizar, no mínimo, três classes: `Node`, `Compressor` e `Extractor`. A classe `Compressor` deve ser utilizada para comprimir o texto e a classe `Extractor` para converter o arquivo comprimido de volta no arquivo original. Outras classes adicionais podem ser utilizadas. Não há restrições na organização interna dessas classes.

A última restrição diz respeito às classes disponíveis no Java. É permitido utilizar as classes `java.util.Map` (ou `java.util.HashMap`) como dicionário, mas não é permitido utilizar a classe `java.util.PriorityQueue`. Em outras palavras, a única estrutura de dados necessária para o trabalho que você não precisa fazer é o dicionário.

Arquivos de entrada estão disponíveis junto com este documento no SIGAA para testes.

4 Pontuação

Este trabalho possui uma pontuação de 70 pontos e pode ser feito **individualmente** ou **em dupla**. A pontuação está distribuída da seguinte maneira:

- Gerar a árvore de codificação — 20 pontos
- Codificar e gerar os arquivos de saída (**.edz** e **.edt**) — 25 pontos
- Recuperar a mensagem original a partir dos arquivos **.edz** e **.edt** — 25 pontos

4.1 Bonificações e Penalidades

Além da pontuação normal, este trabalho pode ter seu total alterado nas seguintes condições:

- Criação dos testes unitários.
  — até 20 pontos
- Unir a tabela de codificação e o arquivo comprimido em um arquivo só com extensão **.edc**.
  — até 20 pontos
- Exibir na saída a taxa de compressão (Exemplo: O arquivo foi possui 37,7% do seu tamanho original).
  — até 5 pontos

- Gerar um novo programa que utiliza uma tabela embutida feita a partir de um dicionário de palavras em português. Este programa funciona igual ao anterior, só que não necessita do gerar nem ler a tabela de um arquivo. O programa original não deve ser substituído por este. Eles devem ser enviados juntos.
  — até 15 pontos

- Não possuir as três classes indicadas na seção anterior.
  — até -15 pontos

- Não possuir a interface de interação indicada na seção anterior[3].
  — -100% dos pontos

5 Observações

- Há mais em um arquivo do que se pode ver.

- Em ambientes Unix, você pode utilizar o comando `xxd` para exibir o conteúdo de arquivos binários (Exemplo: `xxd -b teste1.edz`).

- Em ambientes Unix, você pode utilizar o comando `diff` para comparar o conteúdo de dois arquivos (Exemplo: `diff teste1.txt teste1-output.txt`).

- Para ler os arquivos, você pode utilizar as classes `java.io.File` e `java.io.FileInputStream`.

- Considere os testes unitários seus aliados ao invés de pontos extras.

- Divida o trabalho que deve ser feito em passos. Quanto mais específico cada um desses passos for, melhor. Crie uma lista de tarefas com estes passos e finalize cada um deles individualmente antes de ir para o próximo. Você vai perceber que seu trabalho vai se tornar mais fácil.

6 Colaboração e Plágio

Ajudar o colega de classe é legal, mas copiar seu trabalho, não. Por isso, cópias de trabalhos **não** serão toleradas, resultando em nota **zero** para **todos** os envolvidos. O mesmo vale para códigos copiados da internet.

[3] As unicas exceções são para os casos que vão utilizar apenas um arquivo só contendo a tabela e o conteúdo ao invés de dois e no programa que possui uma tabela de codificação embutida.

7 Entrega

O trabalho deve ser entregue em um único arquivo compactado contendo:

- O código-fonte do seu programa zipado ou um link para um repositório git.
- Os arquivos **.edz** e **.edt** (e, possivelmente, **.edc**) para os arquivos de testes fornecidos que podem ser comprimidos.
- Um arquivo contendo o nome e a matrícula dos participantes chamado `grupo.txt`.

O arquivo deve ser enviado **apenas** pelo *SIGAA* através da opção *Tarefas* até a data divulgada no sistema.