Documentação

Introdução:

Atualmente as pessoas utilizam bastante da compactação de arquivos para diversos usos no dia a dia, mas o principal é quando precisam enviar um arquivo muito grande e demoraria muito para enviar ou até mesmo as plataformas não suportam arquivos extremos, então aí entra a compactação, pois o processo reduz o tamanho do arquivo deixando-o mais conveniente para envio.

Existem diversos modos de compactação de arquivo e um deles é compactação por meio do algoritmo guloso de Huffman e o programa deste projeto utiliza desse meio para fazer a compactação. O processo consiste em criar uma tabela com a quantidade de valores que cabem em um único byte e a frequência em que eles se repetem no arquivo em questão e com esses dados é criado uma trie binária que em cada No folha da trie tem um byte especifico. Essa trie é estruturada de forma gulosa, pois ela pega os bytes que mais se repetem e colocam no inicio da trie e os que menos se repetem ficam posicionados ao fim da trie, e essa estratégia é usada porque é utilizado o caminho da raiz da trie até o determinado No folha que está o byte para construir um código em bits para o byte em questão e esse mesmo código em bits será escrito mais tarde no arquivo compactado. E como descobrir se o arquivo foi compactado de forma correta? Então foi criado uma função de descompactação com mesmo processo de compactação, porém de forma inversa.

Implementação:

A implementação foi a parte mais desafiadora, pois primeiro seria necessário estudar os conceitos e algoritmos de Huffman e só depois pensar em como implementar. Então a implementação foi desenvolvida em partes (mais importantes):

1° Tipos estruturados e TAD´s

O tipo estruturado No_trie_t foi criado com o intuito de ser um No para um arvore binaria, logo ele tem dois ponteiro do seu mesmo tipo no interior, além disso também tem os membros mais importantes para compactação que é um byte especifico e a frequência que esse byte aparece no arquivo a ser compactado.

```
11  // No para a arvore de Huffman
12  typedef struct TRIE
13  {
14      uint8_t symbol;// simbolo ou byte
15      uint32_t frequency;// frequência de symbols
16      struct TRIE *pLeft;
17      struct TRIE *pRight;
18  } No_trie_t;
```

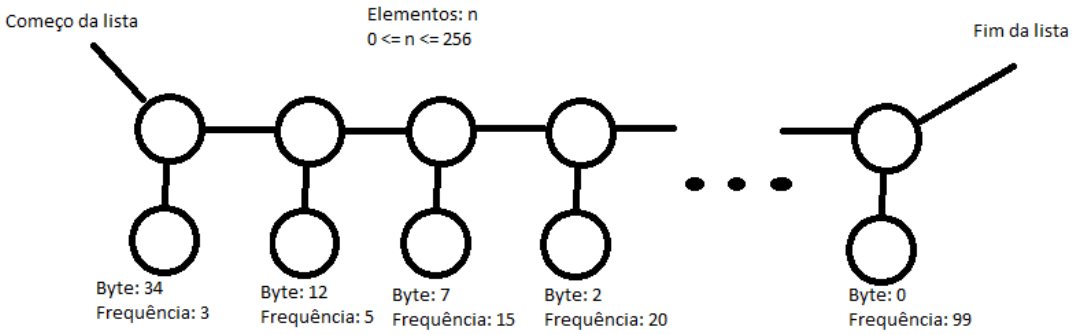
O tipo estruturado No_trie_list_t foir criado para ser um elementos de um lista de Nos, ou seja, em seu interior tem um ponteiro que aponta para o seu mesmo tipo e além disso também tem um ponteiro para um tipo de No_trie_t justamente para acessar a frequência dos bytes e ordenar uma lista que começa do menos frequente até o mais frequente.

```
20  // No para a lista de Nos da arvore de huffman
21  typedef struct NO_TRIE_LIST
22  {
23      No_trie_t *ptrie_member;
24      struct NO_TRIE_LIST *pNext_trie;
25  } No_trie_list_t;
```

O ultimo tipo estrutura importante é o Priority_List_t que tem como membros ponteiros para o inicio e o fim de uma lista ordenada de No_trie_list_t. Esse tipo estruturado também tem um membro para contar quantos Nos tem na lista.

```
27  // Tipo que conta os elementos de um lista e ponteiros para o começo e fim da lista
28  typedef struct PRIORITY_LIST
29  {
30      No_trie_list_t* pBegin_List;
31      No_trie_list_t* pEnd_List;
32      uint32_t Elements;
33  }Priority_List_t;
```

Como já mencionada foi utilizado uma lista para ordenar os Nos em frequências crescentes. A ilustração a seguir mostra um exemplo de como seria a lista:



O processo para a criação dessa lista utiliza de um conjunto de três funções, em que a primeira “newNotrie” cria um No do tipo No_trie_t com as informações do byte e da frequência dele no arquivo, a segundo “newNoList” cria um No do tipo No_trie_list_t e acopla ao No_trie_t criado anteriormente e por último a função que insere e ordenada cada no do tipo No_trie_list_t na lista “sortList”. Já como a implementação das três funções juntas é um pouco extensa omitimos o código, porém pode ser visualizado no arquivo “Huffman definitions.c”.

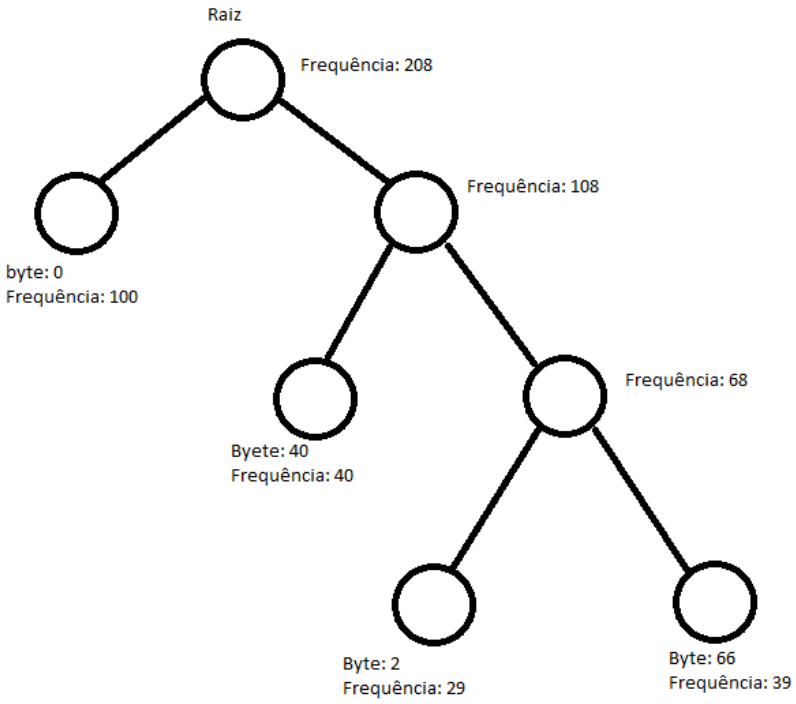
A forma de organizar essas estruturas de dados em uma lista começando com os bytes menos frequentes até os mais frequente é a verdadeira estratégia para a construção da arvore de Huffman. O algoritmo consiste em tirar os dois Nos menos frequentes da lista e adicionar um novo No_trie_list_t que esteja acoplado a um novo No_trie_t que seja pai dos dois Nos retirados da lista e que tenha uma frequência que seja a soma das frequências dos Nos filhos. O seu

membro símbolo/byte será insignificante, pois ele só vai servir de passagem até os Nos folhas onde realmente está localizado os bytes significantes. Esse processo continua até restar apenas um No_trie_list_t na lista e que no caso esse mesmo No está acoplado ao No raiz da arvore de Huffman, então só restaria retornar o No raiz para obter a arvore completa. O algoritmo a seguir faz todo o processo comentado:

```
99    Priority_List_t List_Prt = {.pBegin_List = NULL,
100                               .pEnd_List = NULL,
101                               .Elements = 0U      };
102   bool verific = 0;
103
104   for(uint16_t Count = 0U; Count < TABLE_SIZE_ASCII; Count++)
105   {
106       // Para cada byte que tenha pelo menos uma repeticação é colocado na lista
107       if (ListFrequenc[Count] > 0)
108       {
109           verific = sortList
110           (
111               &List_Prt,
112               newNoList(
113                   newNotrie((uint8_t)Count, ListFrequenc[Count], 0, 0)
114               )
115           );
116
117           if(!verific)
118               return NULL;
119       }
120   }
```

```
122   No_trie_t  *left, *right;
123
124   while(List_Prt.Elements > 1){
125       left = removeFirst(&List_Prt);// Remove o primeiro No atual da lista
126       right = removeFirst(&List_Prt);// Remove o primeiro No atual da lista
127
128       // Add um novo No na lista
129       // Seu membro Trie é pai do left e right
130       // Sua frequencia é a soma da frequencia de left e right
131       verific = sortList
132           (
133               &List_Prt,
134               newNoList(
135                   newNotrie(0, left->frequency + right->frequency, left, right)
136               )
137           );
138       if (!verific)
139           return NULL;
140   }
141   // Retorna a raiz da arvore de Huffman
142   return removeFirst(&List_Prt);
```

A representação gráfica da arvore de Huffman, exemplo:



2° Compactação

Para a compactação resolvi pegar todo o arquivo de entrada, ou seja, o arquivo a ser compactado e colocar em buffer na memoria com intuito de não precisar ficar pegando a informação do disco. O funcionamento do processo de compactação é percorrer cada byte do buffer (arquivo a ser compactado) e buscar o caminho deste byte na arvore de Huffman. O caminho será armazenado em um vetor de char, mas como o caminho é em bits, então comprimimos esses bits em um byte, quando esse byte completar 8 bits compridos será simplesmente escrito no arquivo de saída.

Este processo se repete até que todos os bytes do buffer sejam percorridos e escritos em forma de código binário no arquivo de saída, além disso outras informações também são escritas no arquivo de saída (compactado) como o tamanho do arquivo original, o type file e a frequência dos bytes que se repetem, pois essas informações serão necessárias para a descompactação.

Algoritmo de compactação:

```
267     while (pTrack != BuffFileIn + SizeFile)
268     {
269         bool test = codHuff(source, *pTrack, pathCaract, 0U);
270         if (!test)
271             return HUFF_ERRC_FALIED;
272 
273         uint8_t* pPath = pathCaract;
274 
275         while(*pPath)
276         {
277             if (*pPath == '1')
278             {
279                 byte <<= 1;
280 
281                 byte = byte | 1;
282             }
283 
284             if (*pPath == '0')
285             {
286                 byte <<= 1;
287             }
288 
289             NumberBits++;
290 
291             if (!(NumberBits % 8))
292             {
293                 NumberBits = 0;
294                 fwrite(&byte, sizeof(byte), 1, Out_file);
295                 byte = 0U;
296             }
297 
298             pPath++;
```

3° Descompactação

Uma vez com arquivo compactado para descompactar é bem mais fácil tendo as informações corretas sobre o arquivo original. O processo de descompactação funciona de forma inversa a compactação, ou seja, vai ler do arquivo compactado as informações do original e os códigos binários. Cada bit do código binário irá caminhar sobre a arvore binaria e ao encontrar um No folha o byte/símbolo deste No será escrito no arquivo de saída.

```
365     while (CountElement < Size_FileOri)
366     {
367         byte = *track;
368         for( CountBits = 0; CountBits < SIZE_BITS_OF_BYTE; CountBits++)
369         {
370             if(CountElement == Size_FileOri)
371                 break;
372 
373             if(readBoolean(&byte)){
374                 hold = hold->pRight;
375             }else{
376                 hold = hold->pLeft;
377             }
378 
379             byte <<= 1;
380 
381             if(!hold->pRight && !hold->pLeft)
382             {
383                 fwrite(&(hold->symbol), sizeof(uint8_t), 1, Out_file);
384                 CountElement++;
385                 hold = source;
386             }
387         }
388 
389         track++;
390     }
```

Análise dos testes:

Teste 1°

Arquivo original: letra faroeste caboclo.txt

Tamanho: 6.806 bytes

Arquivo compactado: faroest.hf

Tamanho: 4.948 bytes

Taxa de compressão: 27,2%

Teste 2°

Arquivo original: Imagem.bmp

Tamanho: 3.126 bytes

Arquivo compactado: img.hf

Tamanho: 1.571 bytes

Taxa de compressão: 49.7%

Teste 3°

Arquivo original: blackblue.bmp

Tamanho: 861.054 bytes

Arquivo compactado: blackblue.hf

Tamanho: 164.181 bytes

Taxa de compressão: 80%

Teste 4°

Arquivo original: baroes da pisadinha.mp3

Tamanho: 3.264.444 bytes

Arquivo compactado: baroes.hf

Tamanho: 3.183.764 bytes

Taxa de compressão: 2.7%

Teste 5°

Arquivo original: Thor.bmp

Tamanho: 24.883.254 bytes

Arquivo compactado: DeusdoTrovao.hf

Tamanho: 17.090.947 bytes

Taxa de compressão: 31.3%

Conclusão:

A conclusão final sobre trabalho foi que o algoritmo de Huffman funciona bem com arquivos .txt, imagens .bmp e arquivos de musica .mp3, porém falta eficiente com relação ao tempo, pois em arquivos extensos demora para compactar. Entretanto o objetivo de compactar de descompactar foi alcançado. Mas não foi simples devido um dos maiores empecilhos que foi a construção da arvore de Huffman de forma correta, além que precisei testar com várias entradas antes de implementar os algoritmos de compactação e descompactação. E para concluir foi uma das melhores experiência com trabalhos de programação justamente pelo fato de pesquisar, estudar, aprender e implementar por conta própria.

Bibliografia:

CORMEN, Thomas; LEISERSON, Charles; RIVEST, Ronald; STEIN, Clifford. Algoritmos: Teoria e prática. 3° edição. Rio de janeiro. Elsevier, 10/04/2012

BRASIL. Instituto de matemática e estatísticas universidade de São Paulo, São Paulo, SP, 2018, https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/huffman.html

BRUNO, Davidson. Compressão de dados pelo algoritmo de Huffman. Medium. 2019. Disponível em: < https://medium.com/@davidsonbrsilva/compress%C3%A3o-de-dados-pelo-algoritmo-de-huffman-5e04bc437d77 >. Acesso em: 29/03/2021.