

Pontifícia Universidade Católica do Rio de Janeiro  
INF1010 - Estruturas de Dados Avançadas

Lucas Demarco Cambraia Lemos - 2110013  
Jayme Augusto Avelino de Paiva - 2210289

**Tarefa 4**  
**Compactação e descompactação de dados**

## 1. INTRODUÇÃO

O trabalho proposto trata-se da implementação do algoritmo de Huffman para compactação e descompactação de dados. O principal objetivo de utilizar esse tipo de algoritmo é ocupar menos espaço para armazenar dados, com base na frequência que os símbolos aparecem no arquivo que deseja ser compactado. Assim, este algoritmo permite armazenar símbolos que se repitam de uma forma mais otimizada.

Para codificar o algoritmo de Huffman, é necessário dividir o trabalho em etapas. Inicialmente, é necessário criar a árvore de Huffman, que consiste em uma árvore binária construída a partir das frequências de cada símbolo que aparece no arquivo. Em seguida, os códigos são atribuídos percorrendo a árvore do nó até a raiz, sendo para a esquerda 0 e para a direita 1. Desse modo, cada caractere terá um código binário próprio. Para a descompactação do arquivo, é necessário acessar a árvore criada no passo inicial, para identificar qual caractere está relacionado com o código passado.

## 2. EXPLICAÇÃO DA SOLUÇÃO

Depois de algumas orientações do professor, foi necessário fazer uma adaptação do código, para atendê-las. Fizemos outra função para escrever o arquivo *huffman\_tree.txt*, que contém as informações de todos os caracteres e as frequências. Esse arquivo é utilizado, posteriormente, para montar a árvore novamente e descompactar o arquivo que foi compactado.

A função inicial *leArqTexto* efetua a leitura do arquivo texto, pegando cada caractere e chamando a função *insereListaTemp* para inserir esse caractere lido em uma lista temporária. Se o caractere estiver sendo lido pela primeira vez, insere este caractere na lista com frequência 1; caso não for a primeira vez, soma 1 na frequência do caractere na lista já existente. A função *buscaListaTemp* serve para verificar se o caractere lido já conta na lista e, em caso positivo, retorna um ponteiro para a estrutura dele e, em caso negativo, retorna NULL, simbolizando que não encontrou o elemento. A função *leArqTexto* também adiciona um caractere novo '\*' para representar o fim do arquivo, auxiliando no processo de descompactação.

Após a lista temporária estar completa, é necessário ordená-la para conseguir montar a árvore de Huffman. Para isso, a função *ordenaLista* chama função *insereListaOrdenada*, para criar uma lista com a mesma estrutura da lista temporária, mas ordenada. Após a lista estar devidamente ordenada, com base nas frequências, a função

*criaArvore* é chamada para criar uma árvore binária apenas com a raiz, com a estrutura *NoArvore*, que possui apenas o caractere e a frequência. Após a árvore que somente possui a raiz estiver feita, a função *insereListaArvoresOrdenada* irá inseri-las na lista ordenada de árvores.

A lista de árvores permite realizar as operações para montar a árvore de Huffman. Para isso, chama-se a função *otimizaArvore* que, até o tamanho da lista ser 1, vai retirando os dois primeiros elementos da lista, ou seja, as duas menores frequências, e cria um pai para eles, cuja frequência é a soma dos seus filhos. Para isso, é chamada a função *retiraMenorNo* que realiza essa operação. O novo nó pai criado é inserido novamente na lista por meio da *insereListaArvoresOrdenada*. Para facilitar a criação da árvore, foi passado o caractere '~' para o nó pai, já que o professor orientou para ignorar esses símbolos no arquivo.

Com a árvore de Huffman pronta, a função *montaCodigos* recebe cada caractere do arquivo de texto e varre a árvore em busca dele. Nesse processo ela preenche uma matriz de chars também passada por parâmetro com o código de cada char. Essa matriz possui 256 linhas, de forma que existe uma linha para cada char possível, ela funciona como uma forma de *hashTable* onde a chave hash é o próprio valor do char. Além disso, a função *alturaArvore* calcula a altura da árvore de Huffman que é usada para saber quantas colunas a matriz precisa ter para poder comportar as strings formadas de '1' e '0' correspondentes ao código binário de cada char.

A função *montaCodigosCompleta* usa a função *montaCodigos* para preencher a matriz com os códigos de cada char do texto. Após isso, a função *montaStringCompactada* é responsável por pegar os códigos de cada char e fazer a string que será de fato o texto compactado. Isso é feito via operações *bitwise*, observando a string recebida do arquivo de texto original e criar um vetor de chars que é preenchido com bit a bit com os bits de cada char codificado. Isso é feito olhando a codificação do char e subtraindo 48 do valor de cada char nela (dessa forma '0' virá 0 e '1' vira 1) e colocando o primeiro bit do resultado dessa conta no vetor de chars. Aqui vale ressaltar que nem sempre a string compactada irá ocupar um número múltiplo de 8 bits, ou seja, não irá preencher totalmente o vetor da string compactada. Para solucionar isso, são colocado os bits de um char adicionado artificialmente (com frequência mínima) para preencher os bits restantes do vetor, dessa forma, a descompactação será capaz de saber quando parar, ao ler o char usado para indicar o fim do texto, sem incluir caracteres a mais na string do texto original durante a descompactação. Após a conclusão da

compactação, a função *writeBinFile* escreve um arquivo binário correspondente a string compactada.

Por fim, a função *descompacta* é capaz de ler o arquivo binário e ir preenchendo uma lista (que após a conclusão da leitura é transformada em um vetor de chars, string) com chars correspondentes à descompactação do arquivo binário. Isso é feito lendo bit a bit os bits da string compactada e navegando pela árvore de forma que, se o bit for 1, um ponteiro auxiliar da árvore é movido para direita, se for 0, para esquerda, até chegar numa folha, que irá guardar um char que será posto na lista de chars. Esse processo se repetirá até que a string acabe ou o char lido seja o char usado para determinar a parada de leitura, no caso o '\*'. Por fim a lista de chars é passada para uma string que é escrita em um novo arquivo texto, além de ser imprimida na tela.

Após todo o processo, as estruturas foram devidamente liberadas, com exceção da lista desordenada temporária e a lista ordenada, que foram liberadas assim que foi montada a lista de árvores.

### **3. CONCLUSÃO**

O desafio proposto nessa tarefa trouxe alguns obstáculos para o grupo, mas que conseguiram ser superados. Com os conteúdos apresentados no decorrer da disciplina, junto com pesquisas sobre o algoritmo na internet, o grupo conseguiu implementar a árvore de Huffman com facilidade. Para isso, decidimos mostrar no terminal a lista inicial e, em sequência, esta mesma lista, só que ordenada.

Na parte da compactação, encontramos mais dificuldade para pensar em um jeito de guardar os códigos, mas depois de várias tentativas, conseguimos fazer funcionar.

Considerando a frase teste: "As estruturas de dados são fundamentais para a organização e manipulação eficiente de informações", os códigos para cada caractere ficaram da seguinte forma:

```

-----CARACTERES E SEUS CÓDIGOS-----
* -> 110000
l -> 000000
c -> 11010
z -> 000001
g -> 000010
p -> 110001
i -> 0111
m -> 01100
n -> 0101
f -> 01101
o -> 1010
d -> 0100
a -> 111
u -> 11011
r -> 0001
t -> 11001
e -> 001
   -> 100
s -> 1011
A -> 000011

```

#### 4. IMAGENS DO CÓDIGO FONTE

```

86 // Função para ler o arquivo texto e chama função de inserir os caracteres na lista dos caracteres
87 CaracLista* leArqTexto(char* nomeArquivo, CaracLista* lista){
88     char c;
89     FILE *f = fopen(nomeArquivo, "r");
90     if (f == NULL){
91         printf("Erro ao abrir arquivo");
92         return NULL;
93     }
94     while((c = fgetc(f)) != EOF){
95         lista = insereListaTemp(lista, c);
96     }
97     lista = insereListaTemp(lista, '*');
98     return lista;
99 }
100
101 // Função para inserir os caracteres na lista
102 CaracLista* insereListaTemp(CaracLista* lista, char c){
103     // Verifica na lista existente se esse caractere já está presente lá
104     CaracLista* auxiliar = buscaListaTemp(lista, c);
105     // Caso de percorrer a lista inteira e não encontrar o caractere, então adiciona o novo caractere
106     // no início da lista
107     if (auxiliar == NULL){
108         CaracLista* novoElem = (CaracLista*)malloc(sizeof(CaracLista));
109         if (novoElem == NULL){
110             printf("Erro de malloc");
111             return NULL;
112         }
113         novoElem->c = c;
114         novoElem->prox = lista;
115         novoElem->freq = 1;
116         return novoElem;
117     }
118     // Caso de encontrar o caractere na lista, irá somar a frequência do nó já existente
119     else{
120         auxiliar->freq++;
121         return lista;
122     }
123 }

```

```

124 // Função para buscar um caractere na lista de caracteres
125 CaracLista* buscaListaTemp(CaracLista* lista, char c){
126     CaracLista* auxiliar;
127     for (auxiliar = lista; auxiliar != NULL; auxiliar = auxiliar->prox){
128         // Caso de encontrar o caractere, retorna um ponteiro para o nó que contém o caractere procurado
129         if (auxiliar->c == c){
130             return auxiliar;
131         }
132     }
133     // Caso percorra a lista inteira, retorna NULL, representando que não encontrou o elemento
134     return NULL;
135 }

```

```

137 // Função chamada que percorre a lista original e chama a função ordena essa lista
138 ListaArvores* ordenaLista(CaracLista* lista){
139     CaracLista* auxiliarInsercao;
140     CaracLista* auxiliarListaArvores;
141     CaracLista* listaOrdenada = NULL;
142     // Percorre a lista original e cria uma nova lista ordenada de forma crescente com base na
    frequência
143     for(auxiliarInsercao = lista; auxiliarInsercao != NULL; auxiliarInsercao = auxiliarInsercao->prox){
144         listaOrdenada = insereListaOrdenada(listaOrdenada, auxiliarInsercao->c, auxiliarInsercao->freq);
145     }
146     // Cria uma lista de árvores, em que cada nó é uma árvore binária
147     ListaArvores* listaArvores = (ListaArvores*)malloc(sizeof(ListaArvores));
148     if (listaArvores == NULL){
149         printf("Erro de malloc");
150         return NULL;
151     }
152     listaArvores->primeiro = NULL;
153     listaArvores->tamanho = 0;
154     for(auxiliarListaArvores = listaOrdenada; auxiliarListaArvores != NULL; auxiliarListaArvores =
    auxiliarListaArvores->prox){
155         // Chama função criaArvore, inicializando os ponteiros esq, dir e prox como NULL
156         NoArvore* arvore = criaArvore(auxiliarListaArvores->freq, auxiliarListaArvores->c, NULL, NULL,
    NULL);
157         // Essa árvore é passada como parâmetro para a função que insere as árvores ordenadas nas listas
158         listaArvores = insereListaArvoresOrdenada(listaArvores, arvore);
159     }
160     // Libera a memória das listas que não serão mais utilizadas
161     liberaLista(lista);
162     liberaLista(listaOrdenada);
163     return listaArvores;
164 }

```

```

211 // Função que insere as árvores de somente um nó na lista de árvores
212 ▾ ListaArvores* insereListaArvoresOrdenada(ListaArvores* lista, NoArvore* arvore){
213     NoArvore* auxiliar;
214 ▾ if(lista->primeiro == NULL){
215     lista->primeiro = arvore;
216 ▾ }else if(arvore->freq < lista->primeiro->freq){
217     arvore->prox = lista->primeiro;
218     lista->primeiro = arvore;
219 ▾ }else{
220     auxiliar = lista->primeiro;
221 ▾ while(auxiliar->prox != NULL && auxiliar->prox->freq <= arvore->freq){
222     auxiliar = auxiliar->prox;
223     }
224 ▾ if(auxiliar->prox != NULL){
225     arvore->prox = auxiliar->prox;
226     }
227     auxiliar->prox = arvore;
228     }
229     lista->tamanho++;
230     return lista;
231 }

```

```

188 // Função que retorna uma lista ordenada, com base na lista original
189 ~ CaracLista* insereListaOrdenada(CaracLista* lst, char c, int freq){
190     CaracLista* novo;
191     CaracLista* ant = NULL;
192     CaracLista* p = lst;
193 ~ while(p != NULL && p->freq < freq){
194         ant = p;
195         p = p->prox;
196     }
197     novo = (CaracLista*)malloc(sizeof(CaracLista));
198 ~ if (novo == NULL){
199         printf("Erro de malloc");
200         return NULL;
201     }
202     novo->freq = freq;
203     novo->c = c;
204     // Caso de inserir no início da lista
205 ~ if (ant == NULL){
206         novo->prox = lst;
207         lst = novo;
208     }
209     // Caso de inserir no meio da lista, procura posição correta para inserir
210 ~ else{
211         novo->prox = ant->prox;
212         ant->prox = novo;
213     }
214     return lst;
215 }
216 // Função que cria as árvores com um nó para inserir na lista de árvores
217 ~ NoArvore* criaArvore(int freq, char c, NoArvore* prox, NoArvore* esq, NoArvore* dir){
218     NoArvore* novaArvore = (NoArvore*)malloc(sizeof(NoArvore));
219 ~ if (novaArvore == NULL){
220         printf("Erro de malloc");
221         return NULL;
222     }
223     novaArvore->c = c;
224     novaArvore->freq = freq;
225     novaArvore->cod = 0;
226     novaArvore->dir = dir;
227     novaArvore->esq = esq;
228     novaArvore->prox = prox;
229     return novaArvore;
230 }

```



```

233 // Função para retirar o menor nó de uma lista de árvores
234 NoArvore* retiraMenorNo(ListaArvores* lista){
235     NoArvore* auxiliar;
236     if(lista->primeiro != NULL){
237         auxiliar = lista->primeiro;
238         lista->primeiro = auxiliar->prox;
239         auxiliar->prox = NULL;
240         lista->tamanho--;
241     }
242     return auxiliar;
243 }
244
245 // Função para otimizar a árvore (juntar as árvores, até que sobre apenas uma)
246 NoArvore* otimizaArvore(ListaArvores* lista){
247     while(lista->tamanho > 1){
248         NoArvore* menorElem1 = retiraMenorNo(lista);
249         NoArvore* menorElem2 = retiraMenorNo(lista);
250         NoArvore* novo = (NoArvore*)malloc(sizeof(NoArvore));
251         if(novo == NULL){
252             printf("Erro de malloc\n");
253             return NULL;
254         }else{
255             // If para definir qual menorElem ficará na esquerda --> a menor frequência fica na esquerda
256             if(menorElem1->freq < menorElem2->freq){
257                 novo = criaArvore(menorElem1->freq + menorElem2->freq, '~', NULL, menorElem1, menorElem2);
258             }else{
259                 novo = criaArvore(menorElem1->freq + menorElem2->freq, '~', NULL, menorElem2, menorElem1);
260             }
261             lista = insereListaArvoresOrdenada(lista, novo);
262         }
263     }
264     return lista->primeiro;
265 }

```

```

267 // Função que monta os códigos de um char e coloca ele numa matriz onde o índice da linha equivale
    ao char e o conteúdo da linha é o código
268 int montaCodigos(NoArvore* arv, int pos, char** matrix, char c){
269     if (arv->c == c){
270         (*(matrix+(int)c)+pos) = '\0';
271         return 1;
272     }else if (!(arv->dir) && !(arv->esq))
273         return 0;
274     if (montaCodigos(arv->esq, pos+1, matrix, c)){
275         (*(matrix+(int)c)+pos) = '0';
276         return 1;
277     }else if (montaCodigos(arv->dir, pos+1, matrix, c)){
278         (*(matrix+(int)c)+pos) = '1';
279         return 1;
280     }
281     return 0;
282 }
283
284 //função auxiliar para saber a altura da árvore
285 int alturaArvore(NoArvore* raiz){
286     if (!raiz)
287         return 0;
288     int altEsq = 1+alturaArvore(raiz->esq);
289     int altDir = 1+alturaArvore(raiz->dir);
290     if (altDir > altEsq)
291         return altDir;
292     return altEsq;
293 }

```

```

295 //função para montar o código de todos os chars que aparecem no texto chamando a montaCodigos
296 char** montaCodigosCompleta(NoArvore* arv, char* nomeArquivo){
297     int altura = alturaArvore(arv);
298
299     CaracLista* listaAux = NULL; listaAux = leArqTexto(nomeArquivo, listaAux);
300     int tam = 0;
301     CaracLista* current = listaAux;
302     while (current){
303         tam++;
304         current = current->prox;
305     }
306     current = listaAux;
307     char letras[tam+1];
308     int pos = 0;
309     while (current){
310         *(letras+pos) = current->c;
311         pos++;
312         current = current->prox;
313     }
314
315     char** codigos = (char**)malloc(255*sizeof(char*));
316     if (!codigos){
317         printf("ERROR: Falta de memória\n");
318         exit(1);
319     }
320     for (int i = 0; i < 255; i++){
321         *(codigos+i) = (char*)malloc((altura+1)*sizeof(char));
322         if (!(*(codigos+i))){
323             printf("ERROR: Falta de memória\n");
324             exit(1);
325         }
326     }
327     char* aux = malloc(sizeof(char));
328     for (int i = 0; i<tam; i++){
329         montaCodigos(arv, 0, codigos, letras[i]);
330         printf("%c -> ", letras[i]);
331         char* bit = *(codigos+(int)letras[i]);
332         while (*bit != '\0'){
333             printf("%c", *bit);
334             bit++;
335         }
336         printf("\n");
337     }
338     liberaLista(listaAux);
339     return codigos;
340 }

```

```

342 //função que monta a string equivalente ao texto compactado seguindo os códigos da árvore
343 char* montaStringCompactada(char** codigo, char* nomeArquivo){
344     char c;
345     FILE *f = fopen(nomeArquivo, "r");
346     LCC* firstByte = (LCC*)malloc(sizeof(LCC));
347     if (!firstByte){
348         printf("ERROR: Falta de memória\n");
349         exit(1);
350     }
351     firstByte->byte = 0;
352     firstByte->ocuByte = 0;
353     firstByte->tamTotal = 0;
354     firstByte->prox = NULL;
355     LCC* current = firstByte;
356     LCC* ant;
357     while((c = fgetc(f)) != EOF){
358         char* cod = *(codigo+(int)c);
359         int tam = strlen(cod);
360         int posBit = 0;
361         while (posBit < tam){
362             char bit = *(cod+posBit)-48;
363             bit = bit<<(7-current->ocuByte);
364             current->byte |= bit;
365             current->ocuByte++;
366             posBit++;
367             if (current->ocuByte == 8){
368                 LCC* prox = (LCC*)malloc(sizeof(LCC));
369                 if (!prox){
370                     printf("ERROR: Falta de memória\n");
371                     exit(1);
372                 }
373                 prox->byte = 0;
374                 prox->ocuByte = 0;
375                 prox->tamTotal = 0;
376                 prox->prox = NULL;
377                 ant = current;
378                 current->prox = prox;
379                 current = current->prox;
380             }
381         }
382     }
383     if (current->ocuByte == 0){
384         LCC* auxiliar = current;
385         current = firstByte;
386         free(auxiliar);
387         ant->prox = NULL;
388     }
389     //adicionando um char para servir de indicação que acabou o conteúdo a ser descompactado no caso dele possuir um número de bits
    que não é múltiplo de 8

```

```

389     //adicionando um char para servir de indicação que acabou o conteúdo a ser descompactado no caso dele possuir um número de bits
    que não é múltiplo de 8
390     if (current->ocuByte < 8){
391         int bitsFaltando = 8 - current->ocuByte;
392         char padding = 0;
393         char* aux = *(codigo+(int)'*');
394         for (int cont = 0; cont<bitsFaltando; cont++){
395             char bit = *(aux+cont)-48;
396             bit = bit<<(7-current->ocuByte);
397             current->byte |= bit;
398             current->ocuByte++;
399         }
400     }
401     int tam = 0;
402     current = firstByte;
403     while(current){
404         tam++;
405         current = current->prox;
406     }
407     current = firstByte;
408     char* stringComp = (char*)malloc(tam*sizeof(char)+1);
409     for(int pos = 0; pos<tam; pos++){
410         *(stringComp+pos) = current->byte;
411         current = current->prox;
412     }
413     *(stringComp+tam) = '\0';
414     current = firstByte;
415     liberaLCC(firstByte);
416     return stringComp;
417 }

```

```

419 // Escrevendo arquivo binário
420 void writeBinFile(char* strCompact){
421     FILE* arquivo = fopen("teste.bin", "wb");
422     if (arquivo == NULL) {
423         printf("ERRO: Não foi possível abrir o arquivo de escrita.\n");
424         exit(1);
425     }
426     size_t tamanho = strlen(strCompact);
427     fwrite(strCompact, sizeof(char), tamanho, arquivo);
428     fclose(arquivo);
429 }
430
431 //descompactando arquivo binário
432 void descompacta(NoArvore* arv, char* nomeArquivoB){
433     FILE* arquivoB = fopen("teste.bin", "rb");
434     if (!arquivoB){
435         printf("ERROR: Não foi possível abrir arquivo compactado\n");
436         exit(1);
437     }
438
439     LCC* strDescomp = (LCC*)malloc(sizeof(LCC));
440     strDescomp->byte = 0;
441     strDescomp->prox = NULL;
442     char c;
443
444     LCC* current = strDescomp;
445     NoArvore* currentNode = arv;
446     while((c=fgetc(arquivoB))!=EOF){
447         int bitPos = 0;
448         while(bitPos < 8){
449             char aux = c;
450             aux = (aux<<bitPos)&(0x80);
451             bitPos++;
452             if (!aux){
453                 currentNode = currentNode->esq;
454             }else{
455                 currentNode = currentNode->dir;
456             }
457             if(!(currentNode->esq) && !(currentNode->dir)){
458                 if (currentNode->c == '*')
459                     break;
460                 current->byte = currentNode->c;
461                 current->prox = malloc(sizeof(LCC));
462                 current->prox->byte = 0;
463                 current->prox->prox = NULL;
464                 current = current->prox;
465                 currentNode = arv;
466             }
467         }
468     }

```

```
469     current = strDescomp;
470     int tam = 0;
471     while (current){
472         tam++;
473         current = current->prox;
474     }
475     current = strDescomp;
476     char strLida[tam+1];
477     for(int pos = 0; pos<tam; pos++){
478         *(strLida+pos) = current->byte;
479         current = current->prox;
480     }
481     *(strLida+tam) = '\0';
482
483     FILE* arquivo = fopen("testeDescompactado.txt", "w");
484     if (arquivo == NULL) {
485         printf("ERRO: Não foi possível abrir o arquivo de escrita.\n");
486         exit(1);
487     }
488     size_t tamanho = strlen(strLida);
489     fwrite(strLida, sizeof(char), tamanho, arquivo);
490     fclose(arquivo);
491     printf("String lida do arquivo comprimido:\n");
492     printf("%s\n", strLida);
493 }
```