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

Rio de Janeiro
2023.1

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

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.

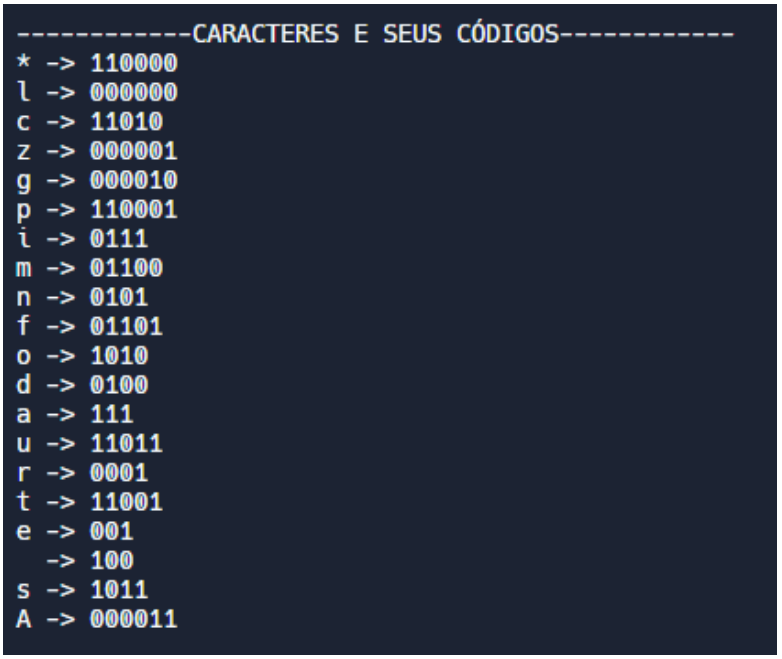
Após todo o processo, as estruturas foram devidamente liberadas, com exceção da lista desordenada temporária e a lista ordenada, que foram liberadas assim que foi montada a lista de árvores.

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 sao fundamentais para a organizacao e manipulacao eficiente de informacoes", os códigos para cada caractere ficaram da seguinte forma:



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
    no início da lista
106   if (auxiliar == NULL){
107     CaracLista* novoElem = (CaracLista*)malloc(sizeof(CaracLista));
108     if (novoElem == NULL){
109       printf("Erro de malloc");
110       return NULL;
111     }
112     novoElem->c = c;
113     novoElem->prox = lista;
114     novoElem->freq = 1;
115     return novoElem;
116   }
117   // Caso de encontrar o caractere na lista, irá somar a frequência do nó já existente
118   else{
119     auxiliar->freq++;
120     return lista;
121   }
122 }
```

```
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 }
```

```c
// Função chamada que percorre a lista original e chama a função ordena essa lista
ListaArvores* ordenaLista(CaracLista* lista){
  CaracLista* auxiliarInsercao;
  CaracLista* auxiliarListaArvores;
  CaracLista* listaOrdenada = NULL;
  // Percorre a lista original e cria uma nova lista ordenada de forma crescente com base na frequência
  for(auxiliarInsercao = lista; auxiliarInsercao != NULL; auxiliarInsercao = auxiliarInsercao->prox){
    listaOrdenada = insereListaOrdenada(listaOrdenada, auxiliarInsercao->c, auxiliarInsercao->freq);
  }
  // Cria uma lista de árvores, em que cada nó é uma árvore binária
  ListaArvores* listaArvores = (ListaArvores*)malloc(sizeof(ListaArvores));
  if (listaArvores == NULL){
    printf("Erro de malloc");
    return NULL;
  }
  listaArvores->primeiro = NULL;
  listaArvores->tamanho = 0;
  for(auxiliarListaArvores = listaOrdenada; auxiliarListaArvores != NULL; auxiliarListaArvores = auxiliarListaArvores->prox){
    // Chama função criaArvore, inicializando os ponteiros esq, dir e prox como NULL
    NoArvore* arvore = criaArvore(auxiliarListaArvores->freq, auxiliarListaArvores->c, NULL, NULL, NULL);
    // Essa árvore é passada como parâmetro para a função que insereas árvores ordenadas nas listas
    listaArvores = insereListaArvoresOrdenada(listaArvores, arvore);
  }
  // Libera a memória das listas que não serão mais utilizadas
  liberaLista(lista);
  liberaLista(listaOrdenada);
  return listaArvores;
}
```

```c
// Função que insere as árvores de somente um nó na lista de árvores
ListaArvores* insereListaArvoresOrdenada(ListaArvores* lista, NoArvore* arvore){
  NoArvore* auxiliar;
  if(lista->primeiro == NULL){
    lista->primeiro = arvore;
  }else if(arvore->freq < lista->primeiro->freq){
    arvore->prox = lista->primeiro;
    lista->primeiro = arvore;
  }else{
    auxiliar = lista->primeiro;
    while(auxiliar->prox != NULL && auxiliar->prox->freq <= arvore->freq){
      auxiliar = auxiliar->prox;
    }
    if(auxiliar->prox != NULL){
      arvore->prox = auxiliar->prox;
    }
    auxiliar->prox = arvore;
  }
  lista->tamanho++;
  return lista;
}
```

```c
// Função que retorna uma lista ordenada, com base na lista original
CaracLista* insereListaOrdenada(CaracLista* lst, char c, int freq){
  CaracLista* novo;
  CaracLista* ant = NULL;
  CaracLista* p = lst;
  while(p != NULL && p->freq < freq){
    ant = p;
    p = p->prox;
  }
  novo = (CaracLista*)malloc(sizeof(CaracLista));
  if (novo == NULL){
    printf("Erro de malloc");
    return NULL;
  }
  novo->freq = freq;
  novo->c = c;
  // Caso de inserir no início da lista
  if(ant == NULL){
    novo->prox = lst;
    lst = novo;
  }
  // Caso de inserir no meio da lista, procura posição correta para inserir
  else{
    novo->prox = ant->prox;
    ant->prox = novo;
  }
  return lst;
}
// Função que cria as árvores com um nó para inserir na lista de árvores
NoArvore* criaArvore(int freq, char c, NoArvore* prox, NoArvore* esq, NoArvore* dir){
  NoArvore* novaArvore = (NoArvore*)malloc(sizeof(NoArvore));
  if (novaArvore == NULL){
    printf("Erro de malloc");
    return NULL;
  }
  novaArvore->c = c;
  novaArvore->freq = freq;
  novaArvore->cod = 0;
  novaArvore->dir = dir;
  novaArvore->esq = esq;
  novaArvore->prox = prox;
  return novaArvore;
}
```

```c
// Função para retirar o menor nó de uma lista de árvores
NoArvore* retiraMenorNo(ListaArvores* lista){
  NoArvore* auxiliar;
  if(lista->primeiro != NULL){
    auxiliar = lista->primeiro;
    lista->primeiro = auxiliar->prox;
    auxiliar->prox = NULL;
    lista->tamanho--;
  }
  return auxiliar;
}

// Função para otimizar a árvore (juntar as árvores, até que sobre apenas uma)
NoArvore* otimizaArvore(ListaArvores* lista){
  while(lista->tamanho > 1){
    NoArvore* menorElem1 = retiraMenorNo(lista);
    NoArvore* menorElem2 = retiraMenorNo(lista);
    NoArvore* novo = (NoArvore*)malloc(sizeof(NoArvore));
    if(novo == NULL){
      printf("Erro de malloc\n");
      return NULL;
    }else{
      // If para definir qual menorElem ficará na esquerda --> a menor frequência fica na esquerda
      if(menorElem1->freq < menorElem2->freq){
        novo = criaArvore(menorElem1->freq + menorElem2->freq, '~', NULL, menorElem1, menorElem2);
      }else{
        novo = criaArvore(menorElem1->freq + menorElem2->freq, '~', NULL, menorElem2, menorElem1);
      }
      lista = insereListaArvoresOrdenada(lista, novo);
    }
  }
  return lista->primeiro;
}
```

```c
// Função que monta os códigos de um char e coloca ele numa matriz onde o indice da linha equivale ao char e o conteudo da linha é o código
int montaCodigos(NoArvore* arv, int pos, char** matrix, char c){
  if (arv->c == c){
    *(*(matrix+(int)c)+pos) = '\0';
    return 1;
  }else if (!(arv->dir) && !(arv->esq))
    return 0;
  if (montaCodigos(arv->esq, pos+1, matrix, c)){
    *(*(matrix+(int)c)+pos) = '0';
    return 1;
  }else if (montaCodigos(arv->dir, pos+1, matrix, c)){
    *(*(matrix+(int)c)+pos) = '1';
    return 1;
  }
  return 0;
}

//função auxiliar para saber a altura da arvore
int alturaArvore(NoArvore* raiz){
  if (!raiz)
    return 0;
  int altEsq = 1+alturaArvore(raiz->esq);
  int altDir = 1+alturaArvore(raiz->dir);
  if (altDir > altEsq)
    return altDir;
  return altEsq;
}
```

```c
//função para montar o código de todos os chars que aparecem no texto chamando a montaCodigos
char** montaCodigosCompleta(NoArvore* arv, char* nomeArquivo){
  int altura = alturaArvore(arv);

  CaracLista* listaAux = NULL; listaAux = leArqTexto(nomeArquivo, listaAux);
  int tam = 0;
  CaracLista* current = listaAux;
  while (current){
    tam++;
    current = current->prox;
  }
  current = listaAux;
  char letras[tam+1];
  int pos = 0;
  while (current){
    *(letras+pos) = current->c;
    pos++;
    current = current->prox;
  }

  char** codigos = (char**)malloc(255*sizeof(char*));
  if (!codigos){
    printf("ERROR: Falta de memória\n");
    exit(1);
  }
  for (int i = 0; i < 255; i++){
    *(codigos+i) = (char*)malloc((altura+1)*sizeof(char));
    if (!(*(codigos+i))){
      printf("ERROR: Falta de memória\n");
      exit(1);
    }
  }
  char* aux = malloc(sizeof(char));
  for (int i = 0; i<tam; i++){
    montaCodigos(arv, 0, codigos, letras[i]);
    printf("%c -> ", letras[i]);
    char* bit = *(codigos+(int)letras[i]);
    while (*bit != '\0'){
      printf("%c", *bit);
      bit++;
    }
    printf("\n");
  }
  liberaLista(listaAux);
  return codigos;
}
```

```c
//função que monta a string equivalente ao texto compactado seguindo os códigos da arvore
char* montaStringCompactada(char** codigo, char* nomeArquivo){
  char c;
  FILE *f = fopen(nomeArquivo, "r");
  LCC* firstByte = (LCC*)malloc(sizeof(LCC));
  if (!firstByte){
    printf("ERROR: Falta de memória\n");
    exit(1);
  }
  firstByte->byte = 0;
  firstByte->ocuByte = 0;
  firstByte->tamTotal = 0;
  firstByte->prox = NULL;
  LCC* current = firstByte;
  LCC* ant;
  while((c = fgetc(f)) != EOF){
    char* cod = *(codigo+(int)c);
    int tam = strlen(cod);
    int posBit = 0;
    while (posBit < tam){
      char bit = *(cod+posBit)-48;
      bit = bit<<(7-current->ocuByte);
      current->byte |= bit;
      current->ocuByte++;
      posBit++;
      if (current->ocuByte == 8){
        LCC* prox = (LCC*)malloc(sizeof(LCC));
        if (!prox){
          printf("ERROR: Falta de memória\n");
          exit(1);
        }
        prox->byte = 0;
        prox->ocuByte = 0;
        prox->tamTotal = 0;
        prox->prox = NULL;
        ant = current;
        current->prox = prox;
        current = current->prox;
      }
    }
  }
  if (current->ocuByte == 0){
    LCC* auxiliar = current;
    current = firstByte;
    free(auxiliar);
    ant->prox = NULL;
  }
  //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
```

```c
  //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
  if (current->ocuByte < 8){
    int bitsFaltando = 8 - current->ocuByte;
    char padding = 0;
    char* aux = *(codigo+(int)'*');
    for (int cont = 0; cont<bitsFaltando; cont++){
      char bit = *(aux+cont)-48;
      bit = bit<<(7-current->ocuByte);
      current->byte |= bit;
      current->ocuByte++;
    }
  }
  int tam = 0;
  current = firstByte;
  while(current){
    tam++;
    current = current->prox;
  }
  current = firstByte;
  char* stringComp = (char*)malloc(tam*sizeof(char)+1);
  for(int pos = 0; pos<tam; pos++){
    *(stringComp+pos) = current->byte;
    current = current->prox;
  }
  *(stringComp+tam) = '\0';
  current = firstByte;
  liberaLCC(firstByte);
  return stringComp;
}
```

```c
// Escrevendo arquivo binário
void writeBinFile(char* strCompact){
  FILE* arquivo = fopen("teste.bin", "wb");
    if (arquivo == NULL) {
        printf("ERRO: Não foi possível abrir o arquivo de escrita.\n");
        exit(1);
    }
    size_t tamanho = strlen(strCompact);
    fwrite(strCompact, sizeof(char), tamanho, arquivo);
    fclose(arquivo);
}

//descompactando arquivo binário
void descompacta(NoArvore* arv, char* nomeArquivoB){
  FILE* arquivoB = fopen("teste.bin", "rb");
  if (!arquivoB){
    printf("ERROR: Não foi possível abrir arquivo compactado\n");
    exit(1);
  }

  LCC* strDescomp = (LCC*)malloc(sizeof(LCC));
  strDescomp->byte = 0;
  strDescomp->prox = NULL;
  char c;

  LCC* current = strDescomp;
  NoArvore* currentNode = arv;
  while((c=fgetc(arquivoB))!=EOF){
    int bitPos = 0;
    while(bitPos < 8){
      char aux = c;
      aux = (aux<<bitPos)&(0x80);
      bitPos++;
      if (!aux){
        currentNode = currentNode->esq;
      }else{
        currentNode = currentNode->dir;
      }
      if(!(currentNode->esq) && !(currentNode->dir)){
        if (currentNode->c == '*')
          break;
        current->byte = currentNode->c;
        current->prox = malloc(sizeof(LCC));
        current->prox->byte = 0;
        current->prox->prox = NULL;
        current = current->prox;
        currentNode = arv;
      }
    }
  }
```

```
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   }
```