Estrutura de Dados
Relatório - Trabalho Prático 2

Nome: Náthaly do Amaral Verzas

1 Introdução

Este trabalho visa construir um compactador/descompactador de arquivos de texto e um relatório para mostrar como funciona. A ideia é que qualquer que seja a análise feita possa transmitir o conhecimento que aqui adquiri.

Compilação

Flags usadas para compilação. GNU G++ compilador Windows.
Para compilação do código:

g++ -Wall -g -pedantic Nathaly.cpp -o teste.exe
Para execução de Compactar:
teste c entrada.txt saida.txt
Para execução de descompactar:
teste d saida.txt original.txt

2 Bibliotecas usadas

```
#include <iostream>
using namespace std;

#include <fstream>
#include <string>
#include <vector>

//Numero maximo de simbolos do alfabeto
#define MAX_ASCII 256
```

3 Classe Node

Esta classe descreve e tem os atributos de uma arvore binaria.

```
class Node{
public:
static int count;

int caracter;
int freq;

Node *proximo;
Node *esquerda;
Node *direita;

Node(int freq){
count++;
this->freq = freq;
esquerda = direita = proximo = NULL;
}

Node(int caracter, int freq){
count++;
this->caracter = caracter;
esquerda = direita = proximo = NULL;
this->freq = freq;
}

~Node(){
count--;
}
};
int Node::count = 0;
```

Variáveis Globais e Funções

Usei o método de protótipo de funções e as seguintes variáves globais.

```
Node *node, *primeiro = NULL, *ultimo = NULL;
vector< vector<bool> > mapa(MAX_ASCII);
vector<bool> caminho;
string simbolos;


void Inserir_na_Lista(Node *node);
void Deletar_Arvore(Node *node);
```

```
void Atribuir_Codigo(Node *node, vector< bool > codigo);
void compactar(char *caminho_entrada, char* caminho_saida);
void descompactar(char *caminho_entrada, char* caminho_saida);
```

Função principal

Usei apenas dois if para testar condição de que o usuário quer realizar com arquivo de entrada.

C para compactar e D para descompactar.

Função Deleta Árvore

Esta função libera a memória alocada dinamicamente para para armazenar a arvore de Huffman. A função faz chamadas recursivas para deletar as sub-arvores dos filhos da direta e da esquerda.
node: Nó raiz da arvore a ser deletada.

```
void Deletar_Arvore(Node *node){
if(NULL != node->direita){
Deletar_Arvore(node->direita);
Deletar_Arvore(node->esquerda);
}
delete node;
}
```

Função que Atribui código para Nó

Esta função atribui o código para cada nó folha de uma arvore.
node: nó raiz da arvore.
código: código acumudado do caminho da arvore.

```
void Atribuir_Codigo(Node *node, vector< bool > codigo = vector<bool>()){
if(!codigo.empty()){
caminho.push_back(codigo.back());
}
if(NULL != node->direita){
codigo.push_back(false);
Atribuir_Codigo(node->esquerda, codigo);
codigo.back() = true;
Atribuir_Codigo(node->direita, codigo);
```

```
}else{
mapa[node->caracter] = codigo;
simbolos += node->caracter;
}
}
```

Insere na lista

Insere na lista ordenada de forma crescente de acordo com a frequência do simbolo. node: Nó a ser inserido.

```
   void Inserir_na_Lista(Node *node){

if(NULL != primeiro){
if(node->freq <= primeiro->freq){
node->proximo = primeiro;
primeiro = node;
}else if(node->freq >= ultimo->freq){
ultimo->proximo = node;
ultimo = node;
}else{
Node *move = primeiro->proximo, *anterior = primeiro;

while(node->freq > move->freq){
anterior = move;
move = move->proximo;
}

anterior->proximo = node;
node->proximo = move;
}
}else{
primeiro = ultimo = node;
}
}
```

Função Compactar

Na primeira etapa ela abre e cria arquivos. Seguida de contar a frequência dos símbolos lendo uma linha da entrada, para cada carácter da linha soma mais um ao tamanho do total da entrada e soma mais um a frequência de entrada, lê a próxima linha e só para o while quando chegar ao fim do arquivo d entrada.

Na segunda etapa ordena os símbolos de acordo com sua frequência. Para então montar a árvore Huffman. Mapeia os códigos para cada símbolo do alfabeto chama função que atribui código.

Na terceira etapa por fim gera o arquivo compactado e volta para o inicio da entrada. Grava o cabeçalho:
1 - o numero de símbolos no alfabeto em little-endian.
2 - o tamanho do arquivo original em bytes em little-endian.
3 - os símbolos do alfabeto.

E insere a descrição da arvore no texto compactado.

Converte cada carácter do arquivo para o código correspondente e grava o texto compactado no arquivo de saída. Ressalto que para cada "bit"no arquivo junta oito casas no vetor para formar o byte. Grava então o carácter formado no arquivo de saída. Se for necessário, preenche o ultimo byte com bits excedentes.

Ultimo passo é liberar a memória alocada com função deleta arvore.

```
void compactar(char *caminho_entrada, char* caminho_saida){

//Abrir/Criar os arquivos

ifstream entrada(caminho_entrada);
if(!entrada){
cerr << "Erro: Nao foi possivel abrir o arquivo " << caminho_entrada << ".";
return;
}

ofstream saida("saida.txt");
if(!saida){
cerr << "Erro: Nao foi possivel abrir o arquivo " << caminho_saida << ".";
return;
}

short K = 0;
int T = 0;

string linha;
vector< int > frequencia(MAX_ASCII, 0);


//Contar a frequencia dos simbolos

getline(entrada, linha); //Le uma linha do entrada
do{
//Para cada caracter da linha
```

```
for(unsigned int aux = 0; aux < linha.size(); aux++){
T++; //Soma mais um ao tamanho total do entrada
(frequencia[linha[aux]])++; //soma mais um a frequencia do caracter
}
//Le a proxima linha
getline(entrada, linha);
}while(!(entrada.eof())); //Enquanto nao alcancar o fim do entrada


//Ordenar os simbolos de acordo com a frequancia

Node *node;

for(unsigned int aux = 0; aux < MAX_ASCII; aux++){
if(0 != frequencia[aux]){
K++;
node = new Node(aux, frequencia[aux]);
Inserir_na_Lista(node);
}
}


//Montar a arvore de Huffman

while(NULL != primeiro->proximo){
node = new Node(primeiro->freq + primeiro->proximo->freq);
node->esquerda = primeiro;
node->direita = primeiro->proximo;
primeiro = primeiro->proximo->proximo;
Inserir_na_Lista(node);
}


//Mapear os codigos para cada simbolo do alfabeto

Atribuir_Codigo(primeiro);


//Gravar o arquivo compactado

//Voltar para o inicio do entrada
entrada.clear();
entrada.seekg(ios::beg);
entrada.clear();
```

```
//Gravar o cabecalho

//Gravar o numero de simbolos no alfabeto em little-endian
saida << ((char) K) << ((char) (K >> 8));
//Gravar o tamanho do arquivo original em bytes em little-endian
saida << ((char) T) << ((char) (T >> 8)) << ((char) (T >> 16)) << ((char) (T >> 24)) ;
//Gravar os simbolos do alfabeto
saida << simbolos;
//Insere a descricao da arvore no texto compactado
vector< bool > texto(caminho);
//texto.insert(texto.begin(), caminho.begin(), caminho.end());


//Converte cada caracter do arquivo para o codigo correspondente.

getline(entrada, linha);
do{
for(unsigned int aux = 0; aux < linha.size(); aux++){
texto.insert(texto.end(), mapa[linha[aux]].begin(), mapa[linha[aux]].end());
}
getline(entrada, linha);
}while(!(entrada.eof()));


//Grava o texto compactado no arquivo de saida.

char t;
unsigned int aux;
//Para cada "bit" no arquivo
for(aux = 0; aux <= (texto.size() - 8);){
//Juntar oito casas do vetor para formar um byte
for(int bi = 0; bi < 8; bi++){
t = (t << 1) + (texto[aux++] ? 1 : 0);
}
//Gravar o caracter formado no arquivo de saida
saida << t;
}

//Se for necessario, preenche o ultimo byte com bits excedentes
if(aux != texto.size()){
saida << (t << (texto.size() - aux));
}
```

```
//Etapa Liberar a memoria utilizada.

Deletar_Arvore(primeiro);
}
```

Função de Descompactar

Esta função descompacta o arquivo indicado por caminho-entrada e grava no arquivo indicado por caminho-saída.

Na primeira etapa abre e cria os arquivos. Em seguida lê o cabeçalho os números de símbolos no alfabeto, números de caracteres no alfabeto e a lista de símbolos do alfabeto.
1 - Lê o arquivo e armazena em vetor de "bits"(booleano).
2 - Montar a árvore.
3 - Descompactar texto.

Enquanto não achar todos os caracteres do texto original não para.

```
   void Descompactar(char *caminho_entrada, char* caminho_saida){

//Abrir/Criar os arquivos

ifstream entrada(caminho_entrada, ios::binary);
if(!entrada){
cerr << "Erro: Nao foi possivel abrir o arquivo " << caminho_entrada << ".";
return;
}

saida.open(caminho_saida);
if(!saida){
cerr << "Erro: Nao foi possivel abrir o arquivo " << caminho_saida << ".";
return;
}


//Ler o cabecalho

char byte_lido[6];
   entrada.read(byte_lido, 6);
//Numero de simbolos no alfabeto
K = (((short)byte_lido[1]) << 8) + byte_lido[0];
//Numero de caracteres do arquivo original
int T = (((int)byte_lido[5]) << 24) + (((int)byte_lido[4]) << 16) + (((int)byte_lido[3]) <<
```

```
//Ler lista de simbolos do alfabeto
char *simbolo = new char[K];
entrada.read(simbolo, K);


//Ler arquivo e armazenar num vetor de "bits"(booleano)

char caracter;
while(entrada.get(caracter)){
for(int aux = 7; aux >= 0; aux--){
texto.push_back(((caracter >> aux) & 1) == 1);
}
}


//Montar a arvore

count_bit = count_simbolo = 0;
primeiro = new Node();
Montar_Arvore(primeiro, simbolo);
delete[] simbolo;


//Descompactar texto.

count_simbolo = 0;
//Enquanto nao achar todos os caracteres do texto original
while(count_simbolo < T){
Andar_Pela_Arvore(primeiro);
}
}
```

Montar Árvore Huffman

Esta função monta a árvore de Huffman a partir das informações extraídas do arquivo compactado.
node: Nó raiz da arvore.

```
void Montar_Arvore(Node *node, char simbolo[]){
//Se falta apenas um simbolo a ser atribuido, ou o bit lido  "1"
if(K == (count_simbolo + 1) || (texto[count_bit++])){
//Atribui o proximo simbolo da lista
```

```
node->caracter = simbolo[count_simbolo++];
}else{
//Acresenta um nó a esquerda
node->esquerda = new Node();
Montar_Arvore(node->esquerda, simbolo);
//Acresenta um nó a direita
node->direita = new Node();
Montar_Arvore(node->direita, simbolo);
}
}
```

Caminho da Árvore

Esta função percorre a árvore segundo os bits lidos do arquivo até encontrar um nó folha. Quando encontra insere o carácter correspondente no arquivo de saída. Percurso Pré-Ordem node: Nó raiz da arvore.

```
void Andar_Pela_Arvore(Node *node){
//Se o nó não tiver filho, então é uma folha da arvore, portanto, um caracter
if(NULL == node->direita){
//Insere o caracter no arquivo
saida << (char)(node->caracter);
count_simbolo++;
}else{
//Se for um bit "1"
if(texto[count_bit++]){
//Vai para o filho da direita
Andar_Pela_Arvore(node->direita);
}else{//Se for um bit "0"
//Vai para o filho da esquerda
Andar_Pela_Arvore(node->esquerda);
}
}
}
```