Exercício-Programa 2 de
Algoritmos e Estrutura de Dados I

Fernanda Moraes Bernardo

Nº USP: 7971991

Turma: 94

Professor: Clodoaldo

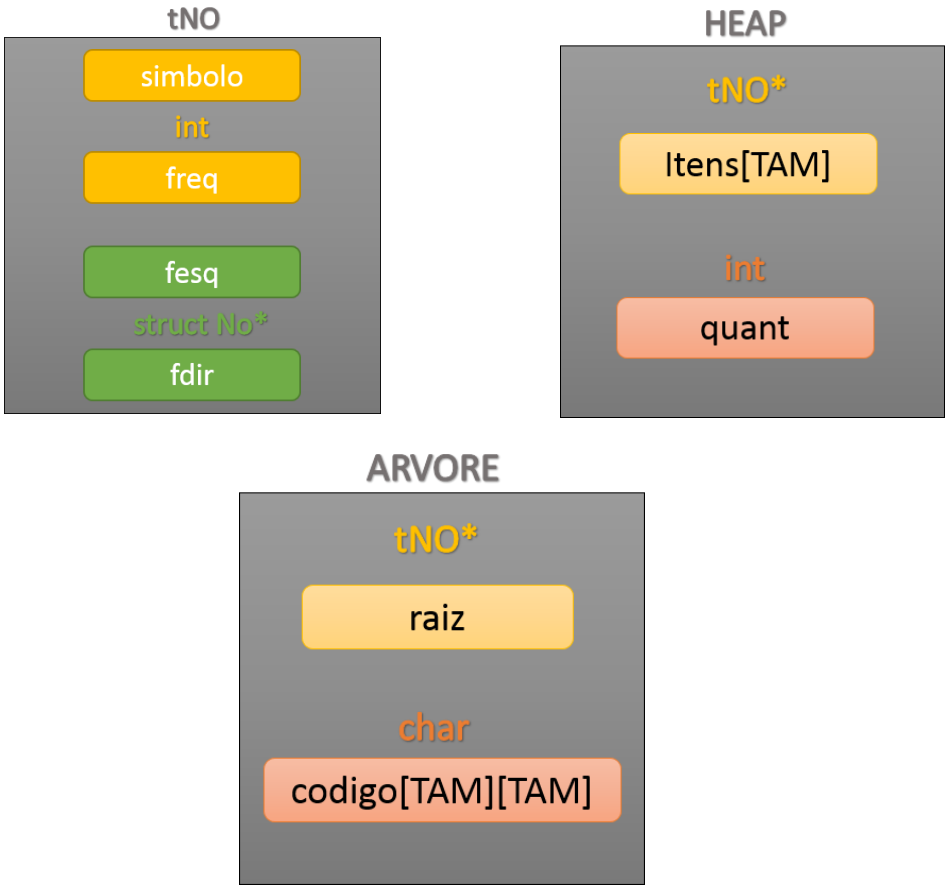
Introdução

Para resolver o problema de arquivos de texto que ocupam muito espaço na memória, é necessário a compactação desse arquivo. Com isso, esse EP exerce essa função, além da de descompactação. De maneira geral, esse programa é capaz de compactar e descompactar um arquivo (da extensão .txt para .bin e ao contrário), utilizando um algoritmo de Huffman. Esse algoritmo codifica um alfabeto de uma mensagem em função da frequência de cada caractere aparecido na mensagem.

Resumo

O código de codificação/decodificação funciona da seguinte forma: para a codificação, um arquivo de texto é lido, dele cada caractere é adicionado em uma estrutura de heap (arranjo de tNO). Em cada posição do arranjo tem um caractere e sua frequência no texto. Após essa etapa, o heap (arranjo) é ordenado. A partir disso, dois nós são retirados desse arranjo e unidos em apenas um tNO, suas frequências são somadas e os filhos desse novo tNO são os que foram retirados do heap. Esse novo tNO é novamente adicionado no heap de forma ordenada. Esse processo ocorre até que sobre apenas uma posição do arranjo. Essa posição contém uma árvore, e esse tNO será adicionado na raiz de uma árvore. O próximo passo é a codificação, para cada filho à esquerda um ‘0’ é adicionado e para cada filho à direita um ‘1’ é adicionado. Após feita a codificação, é o momento de voltar à mensagem lida do arquivo e trocar cada caractere por seu respectivo código. O último passo é gravar a árvore utilizada e o código gerado em um arquivo binário. O processo de decodificação é inverso à esse.

Estruturas


tNO
simbolo
int
freq
fesq
struct No*
fdir
HEAP
tNO*
Itens[TAM]
int
quant
ARVORE
tNO*
raiz
char
codigo[TAM][TAM]

Métodos

- void inicializaHeap (HEAP *heap)

```
void inicializaHeap(HEAP *heap) {
    int i;
    for (i=0; i<TAM; i++) heap->itens[i] = (tNO*) malloc(sizeof(tNO));
    heap->quant = -1;
}
```

Esse método inicializa todos os componentes do HEAP. Aloca espaço para cada tNO do arranjo (itens) e coloca quant como -1, ou seja, não tem nenhum elemento no HEAP.

- void inicializaArvore (ARVORE *arv)

```
void inicializaArvore (ARVORE *arv) {
    arv->raiz = NULL;
    int i;
    for (i=0; i<TAM; i++) arv->codigo[i][0] = '\0';
}
```

Esse método inicializa todos os componentes da ARVORE. Coloca a raiz apontando para NULL (não tem nada na arvore) e coloca um caracter final em cada linha do arranjo código.

- void limpar_arvore(ARVORE *arv)

```
//limpa uma arvore
void limpar_arvore_aux(tNO *raiz) {
  if (raiz != NULL) {
    limpar_arvore_aux(raiz->fesq);
    limpar_arvore_aux(raiz->fdir);
    free(raiz);
    }
}

void limpar_arvore(ARVORE *arv) {
  limpar_arvore_aux(arv->raiz);
  arv->raiz = NULL;
}
```

Esse método usa recursão para limpar (free) todos os nós da árvore e apontar a raiz para NULL

- int tamArquivo (FILE *fp)

```c
int tamArquivo (FILE *fp) {
    if (fp != NULL) {
        int cont = 0;
        char c;
        while(!feof(fp)){
            fread(&c,sizeof(char),1,fp);
            cont++;
        }
        fseek (fp,0,SEEK_SET );
        return cont;
    }
}
```

A partir de um arquivo, esse método é capaz de contar quantos caracteres tem nele. Logo após a contagem, coloca o ponteiro novamente na posição 0 do arquivo.

- unsigned char* adicionaChar (FILE *arquivo, HEAP *heap, unsigned char *msg, int* fim)

```c
//le o arquivo char por char e acrescenta no heap
unsigned char* adicionaChar (FILE *arquivo, HEAP *heap, unsigned char *msg, int* fim) {
    if (arquivo != NULL){
        int tam = tamArquivo(arquivo);
        msg = (unsigned char*) malloc((tam+1)*sizeof(unsigned char));
        int fimAux = *fim;
        int aux = 1;
        unsigned char caracter;
        fscanf (arquivo, "%c", &caracter);
        do {
            if (fimAux>=100*aux-1) {
                aux++;
                msg=realloc(msg, 100*aux);
            }
            inserirHeap(heap, caracter);
            fimAux++;
            msg[fimAux] = caracter;
            fscanf (arquivo, "%c", &caracter);
        } while (!feof(arquivo));
        fimAux++;
        msg[fimAux] = '\0';
        *fim = fimAux;
        return msg;
    }
}
```

A partir de um arquivo, é calculado o número de caracteres dele (tamArquivo), que é usado para alocar espaço na memória para armazenar o texto desse arquivo. Após isso, lê caractere por caractere do arquivo e armazena em msg.

- int inserirHeap (HEAP *heap, unsigned char *caracter)

```
//insere um tNO com o caracter se ele não existir, caso contrário, aumenta a frequência
int inserirHeap (HEAP *heap, unsigned char *caracter) {
    int pos = (int) caracter;
    if (heap->quant <TAM) {
        if (heap->itens[pos]->freq == 0) {
            tNO *novo = (tNO*) malloc(sizeof(tNO));
            novo->simbolo = (int)caracter;
            novo->freq = 1;
            novo->fesq = NULL;
            novo->fdir = NULL;
            heap->itens[pos] = novo;
            heap->quant++;
        }
        else {
            heap->itens[pos]->freq++;
        }
        return 1;
    }
    return 0;
}
```

Esse método recebe um caractere para inserir no heap. Dessa forma verifica na posição do caractere (código binário do char em int) se ele já existe no heap. Se ele existir, adiciona um na frequência; caso contrário, cria um novo tNO e coloca ele no heap.

- void ordenaHeap (HEAP *heap)

```
//ordena heap de forma decrescente
void ordenaHeap (HEAP *heap) {
    int i, j;
    for (i=0; i<TAM-1; i++) {
        for (j=i+1; j<TAM; j++) {
            if (heap->itens[i]->freq < heap->itens[j]->freq) {
                tNO *aux = heap->itens[i];
                heap->itens[i] = heap->itens[j];
                heap->itens[j] = aux;
            }
        }
    }
}
```

A partir de um heap, ordena ele de forma decrescente pelas frequências dos caracteres.

- void inserirItemOrdenado (HEAP *heap, tNO *novo)

```
void inserirItemOrdenado (HEAP *heap, tNO *novo) {
    int i = heap->quant;
    while (i>=0 && novo->freq > heap->itens[i]->freq) {
        heap->itens[i+1] = heap->itens[i];
        i--;
    }
    heap->itens[i+1] = novo;
    heap->quant++;
}
```

Insere um novo tNO de forma ordenada no heap (decrescente).

- void removerDoisItens(HEAP *heap)

```
void removerDoisItens(HEAP *heap){
    int tam = heap->quant;
    tNO* novo = (tNO*) malloc (sizeof(tNO));
    novo->simbolo = '#';
    novo->freq = heap->itens[tam-1]->freq + heap->itens[tam]->freq;
    novo->fesq = heap->itens[tam]; //menor
    novo->fdir = heap->itens[tam-1]; //maior
    heap->quant -= 2;
    inserirItemOrdenado(heap, novo);
}
```

Aloca espaço para um novo tNO, com símbolo ‘#’. A frequência será a soma dos dois últimos tNO’s. O filho da direita (fdir) será o maior e o filho da esquerda (fesq) será o menor. Esses dois nós são removidos do heap e o novo nó será inserido de forma ordenada no heap

- void formarArvore (HEAP* heap, ARVORE *arv)

```
//junta dois nós e reinsere ele ordenado, até sobrar só um nó que é o nó raiz da árvore
void formarArvore (HEAP* heap, ARVORE *arv) {
    while (heap->quant > 0) {
        removerDoisItens(heap);
    }
    arv->raiz = heap->itens[0];
}
```

Faz um laço para remover os nós do heap e juntá-los em um, dois a dois, até sobrar apenas a primeira posição, que constitui a árvore. Portanto a raiz será a primeira posição do arranjo do heap.

- void criarCodigo (ARVORE *arv)

```
void criarCodigoAux (ARVORE *arv,  tNO *no, char *cod, int fim) {
    if (no != NULL) {
        if (no->fesq == NULL && no->fdir == NULL) { //nó folha
            int i;
            for (i = 0; i<=fim; i++) {
                arv->codigo[(int)no->simbolo][i] = cod[i];
            }
            arv->codigo[(int)no->simbolo][fim+1] = '\0';
        }
        else {
            if (no->fesq != NULL) {
                fim++;
                cod[fim] = '0';
                criarCodigoAux(arv, no->fesq, cod, fim);
                fim--;
            }
            if (no->fdir != NULL) {
                fim++;
                cod[fim] = '1';
                criarCodigoAux(arv, no->fdir, cod, fim);
                fim--;
            }
        }
    }
}

//a partir da árvore, gerar códigos para cada caracter
void criarCodigo (ARVORE *arv) {
    char cod[TAM];
    criarCodigoAux(arv, arv->raiz, cod, -1);
}
```

A partir de uma recursão, percorre toda a árvore, e cada vez que, na árvore, vai para a esquerda coloca ‘0’ no código e para a direita ‘1’. Dessa forma quando chega na folha, tem todo o caminho que fez para chegar até ela. Esse código é salvo na matriz da estrutura ARVORE.

- void codificar (ARVORE *arv, unsigned char *msg, char *binario)

```c
//percorre a mensagem do arquivo char por char e substitui pelo código binário
void codificar (ARVORE *arv, unsigned char *msg, char *binario) {
    int fim = -1; //onde parou na codificação
    int i=0;
    while (msg[i] != '\0') {
        char *cod = arv->codigo[(int)msg[i]]; //código do caracter que está sendo lido
        int j=0;
        while (cod[j] != '\0') {
            fim++;
            binario[fim] = cod[j];
            j++;
        }
        i++;
    }
    fim++;
    binario[fim] = '\0';
    imprimirBinario(binario);
}
```

Esse método recebe a mensagem lida do arquivo, e percorre char por char e vai colocando em uma nova “string” o código em 0’s e 1’s correspondente a cada letra.

- unsigned char* decodificar(ARVORE *arv, unsigned char *cod, unsigned char *msg)

```c
unsigned char* decodificar(ARVORE *arv, unsigned char *cod, unsigned char *msg) {
    int i;
    int aux = 1;
    int fim = -1; //aponta para aa última posição da decodificação
    tNO *pno = arv->raiz;
    for (i=0; cod[i] != '\0'; i++) {
        if (fim>=100*aux-1) {
            aux++;
            msg=realloc(msg, 100*aux);
        }
        if (cod[i] == '0') {
            pno = pno->fesq;
        } else if (cod[i] == '1') {
            pno = pno->fdir;
        } else {
            printf("Simbolo codificado errado!\n");
        }
        if (pno->fesq == NULL && pno->fdir == NULL) {
            fim++;
            msg[fim] = pno->simbolo;
            pno = arv->raiz;
        }
    }
    msg[fim+1] = '\0';
    return msg;
}
```

Esse código, a partir da árvore e da mensagem codificada (0’s e 1’s) substitui cada código binário na letra correspondente, formando a mensagem novamente.

- void salva_cod (FILE *fp, char *binario)

```
void salva_cod (FILE *fp, char *binario) {//salva o codigo no arquivo
    int i;
    int tam = tam_cod(binario);//calcula o tamanho do codigo de bits
    int cont = 0;
    int cod_temp = -1;
    unsigned char a =(int)0; //8 bits todos com valores contendo zeros
    unsigned char b = (int) 1; //8 bits somente o ultimo com bit 1
    unsigned char *cod_grav = malloc((tam+1)*sizeof(char)); //codigo convertido para caracter
    for (i=0; binario[i] != '\0'; i++){
        cont++; //incrementa o contador
        if (binario[i]=='0') {//bit 0
            a = a<<1; //realiza um deslocamento a esquerda
        }
        else if (binario[i] == '1'){//bit 1
            a = a<<1; //realiza um deslocamento a esquerda
            a = a|b; //adiciona um bit a primeira posicao
        }
        else {
            printf("%s","Codificação incorreta");
        }
        if (cont==8) {/*completou 8 bits*/
            cod_temp++;
            cod_grav[cod_temp]=a; //armazena o 8 bits
            cont=0;//reseta contador
            a = (int) 0; //zera todos os bits
        }
    }
```

Esse método salva o código gerado em 0’s e 1’s em um arquivo binário (.bin)

- void gravaBinario(ARVORE *arv, FILE *fp, char *binario)

```c
void gravaBinario(ARVORE *arv, FILE *fp, char *binario){
    char marca ='@'; //marca o final da arvore
    if(fp == NULL) {
        printf("\nNao consigo abrir o arquivo ! ");
        exit(1);
    }
    else {
        printf("\nArquivo Criado com Sucesso");
    }
    printf("\nSalvando a arvore no arquivo");
    salva_arvore(arv->raiz,fp);

    printf("\nGravando uma marca de fim de arquivo\n");
    fwrite(&marca,sizeof(char),1,fp);

    printf("\nSalvando codigo no arquivo\n");
    salva_cod(fp,binario);

    printf("\nApagando a estrutura da arvore");
    limpar_arvore(arv);

    printf("\nFechando Arquivo\n");
    //fclose (fp);
}
```

Esse método chama os métodos para gravar a árvore e o código em um arquivo binário.

- void gravaTxt (FILE *fp, unsigned char* msg)

```c
void gravaTxt (FILE *fp, unsigned char* msg) {
    if (fp != NULL) {
        int i = 0;
        while (msg[i] != '\0') {
            fwrite(&msg[i], sizeof(char), 1, fp);
            i++;
        }
    }
}
```

Esse método grava uma mensagem em um arquivo texto (.txt)

- unsigned char* ler_cod (FILE *fp, unsigned char* cod)

```
unsigned char* ler_cod (FILE *fp, unsigned char* cod) {
    int tam;
    int i;
    int val;
    int cont = 0;
    unsigned char a;
    unsigned char b;
    fread(&tam,sizeof(int),1,fp);//le o numero de codigos
    printf("\nForam encontrados %d codigos no arquivo\n", tam);
    cod = malloc((tam+1)*sizeof(char));//cria espaço para armazenar tam caracteres
    fread(&a,sizeof(char),1,fp);//le o proximo caracter
    while (!feof(fp)) {
        b=(int)128;
        for (i=0;i<8;i++){
            if (cont<tam){ //para quando atingiu o numero de bits
                val=(int)(a&b);
                if (val==0) {
                    cod[cont]='0';
                }
                else {
                    cod[cont]='1';
                }
                b=b>>1;
                cont++;
            }
        }
        fread(&a,sizeof(char),1,fp);
    }
    cod[cont+1]='\0';
    return cod;
}
```

Esse método a partir de um arquivo, lê todos os caracteres binários de um arquivo e salva-os em uma “string”

- void salva_arvore(tNO *p,FILE *fp)

```
void salva_arvore(tNO *p,FILE *fp) { //salva uma arvore em disco  -  ? usado para marcar um no sem filho
  if (p==NULL){
    char c='?';
    fwrite(&c,sizeof(char),1,fp);
  }else{
    //printf("%c",c);
    fwrite(&p->simbolo,sizeof(char),1,fp);
    salva_arvore(p->fesq,fp);
    salva_arvore(p->fdir,fp);
  }
}
```

Esse método salva a estrutura da árvore em um arquivo binário.

- void carrega_arvore(tNO **p,FILE *fp)

```
//esta funÃ§Ã£o realiza a leiura de uma arvore em disco
void carrega_arvore(tNO **p,FILE *fp){
  char c;
  fread(&c,sizeof(char),1,fp);

  if (feof(fp)) return;

  if (c=='@') return;

  if (c!='?'){
    //printf("%c",c);
    tNO *paux = (tNO *)malloc(sizeof(tNO));
    paux->fdir=NULL;
    paux->fesq=NULL;
    paux->simbolo = c;
    *p=paux;
    carrega_arvore(&((*p)->fesq),fp);
    carrega_arvore(&((*p)->fdir),fp);
  }
}
```

Esse método le uma árvore de um arquivo binário e armazena na estrutura correspondente

- int tam_cod(char *cod)

```
int tam_cod(char *cod){
    int i,tam=0;
    for (i=0;cod[i]!='\0';i++)  tam++;
    return tam;
}
```

Esse método a partir de uma “string”, calcula o tamanho dela.

- void ler(ARVORE *arv, FILE *fp, unsigned char **binario, unsigned char **msg)

```
void ler(ARVORE *arv, FILE *fp, unsigned char **binario, unsigned char **msg){
    int cont=0;
    char c;
    if(fp ==NULL) {
        printf("\nNao consigo abrir o arquivo ! ");
        exit(1);
    }
    else {
        printf("\nArquivo aberto com Sucesso");
    }
    //procura @
    while(!feof(fp)){
        fread(&c,sizeof(char),1,fp);
        if (c=='@') break;
    }
    //ate o final do arquivo
    while(!feof(fp)){
        fread(&c,sizeof(char),1,fp);
        cont++;
    }
    fseek (fp,0,SEEK_SET ); //volta o ponteiro para inicio

    carrega_arvore(&arv->raiz,fp);//carrega a arvore

    while(!feof(fp)){
        fread(&c,sizeof(char),1,fp);
        if (c=='@') break;
    }
    *binario = ler_cod(fp,*binario);
    printf("Texto Binário:\n");
    imprimirBinario(*binario);
    *msg = decodificar(arv, *binario, *msg);
    //printf("Árvore:\n");
    //imprimirArvore(arv->raiz);
}
```

Esse método lê um arquivo binário, tanto a parte da árvore (localizada na primeira parte do arquivo) quanto a parte do código (localizada logo depois da árvore, separada por um caractere ‘@’). Armazenando cada parte em sua respectiva estrutura.

- int main(int argc, char const *argv[])

```c
int main(int argc, char const *argv[]) {
    if (!setlocale (LC_CTYPE, "")) {
        fprintf(stderr, "Não pode abrir a localização especificada");
        return 1;
    }

    HEAP *heap = (HEAP*) malloc(sizeof(HEAP));
    ARVORE *arv = (ARVORE*) malloc(sizeof(ARVORE));
    inicializaHeap(heap);
    FILE *arquivo;
    if (arquivo == NULL) {
        printf("Não consigo abrir o arquivo\n");
        return 1;
    }
    char const tipo = argv[1];
    char const *arquivoEntrada = argv[2];
    char const *arquivoSaida = argv[3];
if (tipo == 'c')  { //COMPACTAÇÃO
    arquivo = fopen (arquivoEntrada, "r");
    int *fim = (int*) malloc(sizeof(int));
    *fim = -1;
    unsigned char *msg = adicionaChar(arquivo, heap, msg, fim);
    fclose (arquivo);
    ordenaHeap (heap);
    formarArvore (heap, arv);
    criarCodigo (arv);
    int tam = tam_cod(msg);
    printf("Tamanho msg: %d\n", tam);
    char *binario = (char*) malloc((tam*TAM)*sizeof(char));
    codificar (arv, msg, binario);
    int tam2 = tam_cod(binario);
    printf("Tamanho bin: %d\n", tam2);
    //imprimirBinario(binario);

    FILE *fp = fopen (arquivoSaida,"wb");
    if (fp  == NULL) { //abre o arquivo para escrita em modo binario
      printf("\nNao consigo abrir o arquivo!");
      exit(1);
    }
    gravaBinario (arv, fp, binario); //salva o codigo em arquivo
    fclose (fp); //fecha o arquivo
    return 0;
}
```

```
    else if (tipo == 'd') { //DESCOMPACTAÇÃO
        arquivo = fopen (arquivoEntrada, "rb");
        FILE *fp = fopen (arquivoSaida, "w");
        if (arquivo == NULL || fp == NULL) {
            printf("\nNão consigo abrir o arquivo!");
            exit(1);
        }
        unsigned char *binario;
        unsigned char *msg = (char*) malloc(100*(sizeof(char)));
        ler (arv,arquivo, &binario, &msg);
        imprimirBinario(msg);
        gravaTxt(fp, msg);
        fclose(arquivo);
    }
    else {
        printf("Arquivo com extensão errada!\n");
        return 1;
    }
}
```

No main, todas as estruturas são alocadas e inicializadas (através dos métodos de inicialização já mostrados). O que será executado (codificação ou decodificação) e os arquivos de entrada e saída são passados pelo usuário na hora da chamada do programa.

Se for uma compactação, o arquivo será aperto para leitura (‘r’) e serão chamados os métodos para leitura dos caracteres e para adicioná-los no heap. Logo depois, será chamado o método para ordenar o heap e formar a árvore. Após isso, ele irá codificar a mensagem e gravar no arquivo binário.

Se for uma descompactação, será o processo contrário. Primeiro, abrirá o arquivo binário para leitura, lerá o texto, decodificará e gravará em um arquivo de texto (.txt.)