Relatório 2º Trabalho de Estrutura de Dados

Professora: Patrícia Dockhorn Costa
Período: 2022/1

Decodificando e codificando com Huffman

Alunos: Rhuan Garcia de Assis Teixeira e Gabriel Braga Ladislau

Sumário

Introdução:

Para esse trabalho temos que implementar um compactador e um descompactador de arquivos.

Para isso teremos que mexer no baixo nível de bits, usando o TAD *bitmap*, com ele conseguimos fazer tudo necessário para trabalhar com bytes. Usamos o *doxygen* para documentar o código e conseguir criar um padrão de comentários. Fizemos comentários de todas as funções e estruturas presentes no nosso código. Usamos, também o serviço do *GitHub* para versionar nosso código e trabalhar com mais eficiência.

Implementação:

Para começar, fizemos uma implementação sem seguir nenhum padrão, reutilizamos as árvores que implementamos nas aulas para fazer a árvore de Huffman.

Usamos a biblioteca *assert.h* para verificar se os ponteiros usados nas funções do programa estão devidamente alocados e diferentes de *NULL*. Caso esteja *NULL* a função *assert()* para a execução do programa com erro (0 = erro).

Começamos primeiro pelo compactador e seus TADs para depois seguir para o Descompactador.

TAD VetChar:

Nesse TAD foi implementado o vetor de frequência de caracteres. Sua estrutura é basicamente composta por um vetor de 256 inteiros (uma posição para cada combinação possível em 8bits) em que armazenamos quantas vezes cada combinação apareceu durante o arquivo.

Data Fields

int **vetor** [**MAX_VET**]

Ele possui as seguintes funções:

Functions

VetChar *	**VetCharCria** (FILE *arqbase) Faz a criacao de um vetor frequencia dado um arquivo base. More...
void	**LiberaVetChar** (**VetChar** *alvo) Libera o vetor frequencia. More...
int	**VetGetPos** (**VetChar** *vet, int i) Retorna quantas vezes uma dada combinacao de btis apareceu no arquivo. More...

Dentre essas, vemos uma que vale ser melhor explicada neste relatório dada a maior complexidade, sendo ela:

VetCharCria

```
VetChar *VetCharCria(FILE *arqbase)
{

    VetChar *saida = (VetChar *)malloc(sizeof(VetChar));
    for (int i = 0; i < MAX_VET; i++)
    {
        saida->vetor[i] = 0;
    }

    PreencheVetChar(saida, arqbase);
    rewind(arqbase);
    return saida;
}
```

Nessa função alocamos a memória necessária para o vetor de frequência, inicializamos o mesmo com 0's para todos valores e em seguida chamamos uma função (presente abaixo) para que o vetor seja preenchido com as devidas informações

```
static void PreencheVetChar(VetChar *vetorDfreq, FILE *arq)
{
    unsigned char aux;

    while (!feof(arq))
    {
        if (fread(&aux, 1, 1, arq))
        {
            if (!aux)
                vetorDfreq->vetor[0]++;

            else
                vetorDfreq->vetor[aux]++;
        }
        else
            break;
    }
}
```

Nessa função, o arquivo é lido byte a byte, a cada leitura esse byte é convertido em um número usado como índice para nosso vetor de frequência. E então o contador dessa combinação de bits é incrementada.

TAD ListaGen

É uma simples implementação de lista genérica, podemos ver a estrutura de sua célula e sentinela abaixo

Célula:

Data Fields

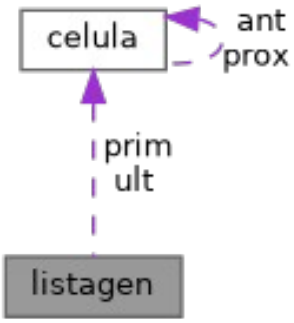
void *	item
Celula *	prox
Celula *	ant

Sentinela:

Data Fields

Celula *	prim
Celula *	ult

Organização:



Ele possui as seguintes funções:

Functions

Listagen *	**IniciaListaGen** () Inicia a lista vazia. More...
void	**InsereItemGen** (**Listagen** *lista, void *item) Insere um item no inicio da lista. More...
void *	**RetiraDaListaGen** (**Listagen** *lista, void *chave, int(*Comparador)(void *, void *)) Retira um item da lista comparando com uma função de callback do tipo de item. More...
void	**ImprimeListaGen** (**Listagen** *lista, void(*Imprime)(void *)) Imprime a Lista Generica, dada a funcao para a impressao de um elemento. More...
void	**LiberaListaGen** (**Listagen** *lista, void(*Destroi)(void *)) Libera a lista dando free também no item dela caso a função de destruir do tipo seja passada como argumento. More...
Listagen *	**ReorganizaLista** (**Listagen** *lista, int(*MenorQ)(void *, void *)) Função para organizar a lista dependendo da função passada como argumento, (Cria uma nova lista incluindo itens da antiga na nova, de acordo com a ordem estabelecida pela função de callback). More...
int	**VaziaLista** (**Listagen** *lista) Retorna 1 se a lista esta vazia, 0 caso contrário. More...
int	**PercorreLista** (**Listagen** *lista, int(*cb)(void *)) Percorre a lista executando a funcao passada como parametro. More...
void *	**RetornaPrimeiro** (**Listagen** *lista) Retorna item presente no primeiro elemento da lista. More...
void *	**RetornaUlt** (**Listagen** *lista) Retorna item presente no ultimo elemento da lista. More...
void *	**RetiraPrimeiro** (**Listagen** *lista) Retira o primeiro item da lista, retornando o objeto ligado a ele. More...
void	**InsereUltItemGen** (**Listagen** *lista, void *item) Insere um item no final da lista. More...
void *	**BuscaLista** (**Listagen** *lista, int(*Compara)(void *, void *), void *chave) Busca o elemento na lista dada a chave de busca, caso ache o elemento eh retornado caso contrario, retorna-se null. More...

Dentre essas, deve ser melhor explicada a função **ReorganizaLista**, presente abaixo.

```c
Listagen *ReorganizaLista(Listagen *lista, int (*MenorQ)(void *, void *))
{
    assert(MenorQ);

    if (lista)
    {
        Listagen *listaNova = IniciaListaGen();
        while (!VaziaLista(lista))
        {
            Celula *p = lista->prim;
            Celula *aux = p;
            for (; p && p->prox; p = p->prox)
            {
                aux = p->prox;
                if (MenorQ(aux, p))
                {
                    aux = p;
                }
            }
            InsereItemGen(listaNova, aux->item);
            RetiraDaListaGenPorCel(lista, aux);
        }
        free(lista);
        return listaNova;
    }
    return lista;
}
```

Nela é reorganizada a lista dada a função de comparação entre elementos (itens), um exemplo de uso é a reorganização de uma lista de arvores com base no peso das mesmas

TAD Arvore:

Para que seja possível implementar a árvore de codificação de Huffman foi necessário que a Arvore em si carregasse um inteiro representando o peso daquele caractere e um *char* que seria o caractere em questão, vemos a estrutura de arv abaixo:

Data Fields

unsigned char	**letra**
int	**peso**
Arv *	**esq**
Arv *	**dir**

Ele possui as funções abaixo:

Dentre elas, serão uteis maiores explicações sobre as seguintes:

Functions

Arv *	**ArvCriaVazia** () Cria arv Vazia. More...
Arv *	**ArvCria** (unsigned char letra, int peso, **Arv** *esq, **Arv** *dir) Função para criar Arvore, aqui podemos adicionar um caractere e seu peso correspodente na contagem do Algoritimo de Huffman. More...
Arv *	**ArvLibera** (**Arv** *a) Função para liberar todo o espaço ocupado pela árvore a, libera também suas raizes. More...
int	**ArvVazia** (**Arv** *a) Função que retorna 1 se a árvore está vazia. More...
void	**ArvImprime** (**Arv** *a) Função que imprime a arvore a, em pré-ordem. More...
Arv *	**ArvPai** (**Arv** *a, unsigned char c) Procura o pai do nó que contém o caractere c. More...
int	**QntdFolhas** (**Arv** *a) Retorna a quantidade de folhas de certa árvore. More...
unsigned char	**ArvChar** (**Arv** *a) Retorna o caractere da árvore caso seja diferente de NULL. More...
int	**ArvPeso** (**Arv** *a) Retorna o peso da árvore caso seja diferente de NULL. More...
int	**ArvAltura** (**Arv** *a) Retorna altura da arvore a. More...
bitmap *	**ExportaArvore** (**Arv** *a) Retorna o bitmap referente a arvore de codificacao seguindo a travessia de pre-ordem e bit de identificacao para folhas e nós. More...
int	**PosiscaoChar** (**Arv** *raiz, unsigned char c) Retorna se o no que possui o caractere a esta presente na esquerda ou direita da arvore passada. More...
int	**ExisteChar** (**Arv** *a, unsigned char c) Retorna se ha um no com o caractere c na arvore passada. More...
bitmap *	**CodificaChar** (**Arv** *raiz, unsigned char carac) Retorna o codigo referente ao caractere dado, na codificacao da arvore fornecida. More...
Arv *	**FazArvdeBitMap** (**bitmap** *bitmap) Dado um bitmap contendo a arvore serializada em pre-ordem, retorna a mesma desserializada para uso no projeto. More...
void	**PercorreArvorePorBitEEscreveSaida** (**BitIndex** *arquivo, **Arv** *arvore, unsigned long int tamTotalBits, FILE *saida) Decodifica o conteudo de um Bitmap Indexado com base na Arvore de Huffman passada, escrevendo no arquivo de saida o resultado da decodificacao. More...

ExportaArvore

Dada uma árvore binária, a função a serializa em pré-ordem colocando a saída em um bitmap. Nós são codificados como bit 0, folhas como bit 1, seguido do caractere presente na mesma.

```
bitmap *ExportaArvore(Arv *a)
{
    assert(a);

    unsigned int h = ArvAltura(a);
    unsigned int qntdfolhas = QntdFolhas(a);
    unsigned int tam = (1 + (h * 2) + (qntdfolhas * 8) * 2);
    while (tam % 8)
    {
        tam++;
    }

    bitmap *mapa = bitmapInit(tam);

    VarreduraArv(mapa, a);

    return mapa;
}
```

```
static void VarreduraArv(bitmap *mapa, Arv *a)
{
    if (a != NULL)
    {
        if (EhFolha(a))
        {
            bitmapAppendLeastSignificantBit(mapa, 1);
            EscreveChar(mapa, a->letra);
        }
        if (EhNo(a))
        {
            bitmapAppendLeastSignificantBit(mapa, 0);
            VarreduraArv(mapa, a->esq);
            VarreduraArv(mapa, a->dir);
        }
    }
}
```

CodificaChar

Dada uma árvore de Huffman e um char presente na mesma, essa função verifica de que lado o caractere informado se encontra na arvore, escreve o dado obtido no bitmap e em seguida faz uma recursão até que chegue a folha onde o caractere está presente.

```
static void Recursiva(bitmap *codificando, Arv *a, unsigned char c)
{
    // Tem que verificar se eh um no, pq caso seja uma folha nao precisa codificar
    if (EhNo(a))
    {
        if (!PosiscaoChar(a, c)) // Esta na esquerda
        {
            bitmapAppendLeastSignificantBit(codificando, 0);
            Recursiva(codificando, a->esq, c);
        }
        else // Esta na direita
        {
            bitmapAppendLeastSignificantBit(codificando, 1);
            Recursiva(codificando, a->dir, c);
        }
    }
}

bitmap *CodificaChar(Arv *raiz, unsigned char carac)
{
    bitmap *codigo = bitmapInit(ArvAltura(raiz) * 2);
    if (ExisteChar(raiz, carac))
    {
        Recursiva(codigo, raiz, carac);
    }
    return codigo;
}
```

FazArvdeBitMap

Faz a de-serialização de um arvore binária serializada em pré-ordem. Para isso, faz a indexação do bitmap de entrada, e faz a leitura do mesmo bit a bit, tratando-o como uma pilha. Ao encontrar uma folha (1) no bitmap de entrada, faz a leitura de 1 byte do bitmap, o colocando em uma folha devidamente localizada na arvore de saída.

```
Arv *RecursividadeCriadora(BitIndex *bitmap)
{
    Arv *saida;
    if (ProxBit(bitmap)) // Se for folha
    {
        saida = ArvCria(LeCaractere(bitmap), 1,
                        ArvCriaVazia(),
                        ArvCriaVazia());
        return saida;
    }
    else // Eh no
    {
        saida = ArvCria('\0', 0,
                        ArvCriaVazia(),
                        ArvCriaVazia());
        saida->esq = RecursividadeCriadora(bitmap);
        saida->dir = RecursividadeCriadora(bitmap);
        return saida;
    }
    return saida;
}

Arv *FazArvdeBitMap(bitmap *bitmap)
{
    BitIndex *bitindexado = IniciaBitIndex(bitmap);
    Arv *saida;
    saida = RecursividadeCriadora(bitindexado);
    LiberaBitIndx(bitindexado);
    return saida;
}
```

PercorreArvorePorBitEEscreveSaida

Dado um bitmap contendo um arquivo (sem cabeçalho e/ou arvore serializada) codificado usando uma arvore de Huffman e sua arvore de Huffman, essa função chama **RetornaCharRecursivamente** (que faz a leitura bit a bit do arquivo, até que se complete um caractere, retornando o mesmo) e escreve em disco o caractere recebido, até que o bitmap acabe.

```c
void PercorreArvorePorBitEEscreveSaida(BitIndex *arquivo, Arv *arvore,
                                       unsigned long int tamTotalBits, FILE *saida)
{
    unsigned long int contadorDebits[1];
    contadorDebits[0] = tamTotalBits;
    unsigned char aux;
    while (contadorDebits[0] != 0)
    {
        aux = RetornaCharRecursivamente(arquivo, arvore, contadorDebits);
        fwrite(&aux, 1, 1, saida);
    }
}

unsigned char RetornaCharRecursivamente(BitIndex *p, Arv *arvore,
                                        unsigned long int *contadorDebits)
{
    if (EhNo(arvore))
    {
        contadorDebits[0]--;
        if (ProxBit(p))
        {
            return RetornaCharRecursivamente(p, arvore->dir, contadorDebits);
        }
        else
        {
            return RetornaCharRecursivamente(p, arvore->esq, contadorDebits);
        }
    }
    if (EhFolha(arvore))
    {
        return arvore->letra;
    }
}
```

TAD ListaArv

Utilizando a lista genérica anteriormente explicada, foi construída a lista de árvores, que utilizamos para a montagem da árvore de Huffman.

Ele possui as seguintes funções:

Functions

Listagen *	**IniciaListaArv** () Inicializa lista de Arvores. More...
void	**InsereListaArv** (**Listagen** *lista, **Arv** *inserida) Insere arvore na lista passada. More...
void	**RetiraListaArvPeso** (**Listagen** *lista, int Peso) Retira a arvore da lista usando como chave de busca o peso. More...
void	**RetiraListaArvLetra** (**Listagen** *lista, unsigned char letra) Retira a arvore da lista usando como chave de busca a letra. More...
void	**ImprimeListaArv** (**Listagen** *lista) Imprime a lista de arvores. More...
Listagen *	**ReorganizaListaArv** (**Listagen** *lista) Reorganiza lista baseada em peso. More...
int	**PercorreListaArv** (**Listagen** *lista, int(*func)(void *)) Percorre a lista de arvores executando a funcao fornecida para cada elemento presente na lista. More...
int	**FundePrimeiros** (**Listagen** *lista) Retira as duas primeiras arvores da lista, cria um no raiz com o peso = soma dos pesos, insere o no no final da lista. More...
void	**PreencheLista** (**Listagen** *lista, **VetChar** *VetTemp) Preenche a lista de arvores com os dados presentes no arquivo passado como parametro. More...
void	**LiberaListaArv** (**Listagen** *lista) Faz a liberacao da lista de Arvores, liberando suas arvores. More...

Entre elas, são válidas de trazer a esse relatório as seguintes funções:

FundePrimeiros

Nessa função, retiramos da lista os dois primeiros elementos e criamos um nó, cujo peso é igual a soma dos pesos e seus filhos são as árvores retiradas da lista. Após esse processo, inserimos esse nó criado no fim da lista.

```
int FundePrimeiros(Listagen *lista)
{
    if (MenosdeDoisElementos(lista) == 0)
    {
        Arv *esquerda = (Arv *)RetiraPrimeiro(lista);
        Arv *direita = (Arv *)RetiraPrimeiro(lista);

        Arv *inserido = ArvCria('\0', ArvPeso(esquerda) + ArvPeso(direita),
                                esquerda, direita);

        InsereUltItemGen(lista, inserido);
        return 1;
    }
    else
        return 0;
}
```

PreencheLista

Essa função possui a funcionalidade de, dado um vetor de frequência, preencher uma lista de árvores com a informação presente. Para isso, varremos o vetor e caso obtenha uma frequência maior que 0, cria uma folha cujo caractere é a combinação de bits (byte//caractere) relacionada a frequência.

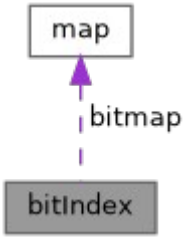
```
// Dentro dessa funcao, Usar VetCHAR para preencher a lista gen com as arvores respectivas.
// Com ela, preenche a lista de arvores inserindo todos com peso diff de 0
// Apos, chama organiza listadeArv, libera vetchar e retorna
void PreencheLista(Listagen *lista, VetChar *VetTemp)
{
    for (int i = 0; i < MAX_VET; i++)
    {
        if (VetGetPos(VetTemp, i))
        {
            Arv *a = ArvCria((unsigned char)i, VetGetPos(VetTemp, i),
                             ArvCriaVazia(), ArvCriaVazia());
            InsereListaArv(lista, a);
        }
    }
}
```

TAD BitIndex

É um simples tipo para adicionarmos algumas funcionalidades ao bitmap, tendo um index, podemos saber onde está a leitura do arquivo e, por exemplo, o tratar como uma pilha de bits.

Data Fields

bitmap *	**bitmap**
int	**index**



Nele Temos as seguintes funções:

Functions

BitIndex *	**IniciaBitIndex** (**bitmap** *bitm) Inicia um Bitmap indexado, dado um bitmap comum. More...
unsigned char	**ProxBit** (**BitIndex** ***bitmap**) Retorna o proximo bit do bitmap, caso omesmo fosse tratado como uma pilha. More...
unsigned char	**LeCaractere** (**BitIndex** *base) Retorna os proximos 8 bits do bitmap, concatenados em um char. More...
void	**LiberaBitIndx** (**BitIndex** ***bitmap**) Libera a estrutura de BitIndexado, sem liberar o bitmap em que foi baseado. More...

Dada a simplicidade de todas, não se faz necessária maiores explicações sobre o funcionamento de nenhuma.

TAD Codificador:

O nome é codificador porém este TAD é responsável pela compactação do arquivo. Nele encontramos a estrutura tabela de codificação que carrega para cada *char* em um arquivo, seu correspondente em *bitmap* tendo já a arvore de huffman. Segue abaixo:

```
struct TabelaDeCod
{
    unsigned char *carac;
    bitmap **codigo;
};
```

Para essa tabela fizemos funções de montar de acordo com a arvore de *Huffman* e o vetor de freq. e Libera para liberar memória.

```
Tabela *MontandoTabela(Arv *Huffman, VetChar *vetor, int tam)
{
    // Montando tabela de codificacao

    Tabela *tab = (Tabela *)malloc(sizeof(Tabela));

    int count = 0;

    tab->carac = (unsigned char *)malloc(tam * sizeof(unsigned char));
    tab->codigo = (bitmap **)malloc(tam * sizeof(bitmap *));

    // Preenchendo a tabela
    for (int i = 0; i < MAX_VET; i++)
    {
        if (VetGetPos(vetor, i) != 0)
        {
            // para cada caracter existente no arquivo de saida um código é criado aqui
            // o i representa o caracter no vetor de frequencia
            tab->carac[count] = (unsigned char)i;
            tab->codigo[count] = CodificaChar(Huffman, tab->carac[count]);
            count++;
        }
    }
    return tab;
}

void LiberaTabela(Tabela *tab, int qntd)
{
    for (int i = 0; i < qntd; i++)
    {
        bitmapLibera(tab->codigo[i]);
    }
    free(tab->codigo);
    free(tab->carac);
    free(tab);
}
```

Como podemos ver para cada um dos caracteres a função CodificaChar pega e gera um código em bitmap (de acordo com a arvore de *Huffman*) para o caractere i caso seu peso seja diferente de 0, temos assim uma tabela que para cada caractere no vetor de *char* seu respectivo código corresponde *(mesmo index)* no vetor de ponteiros para bitmap.

Voltando ao compactador em si temos essas funções privadas:

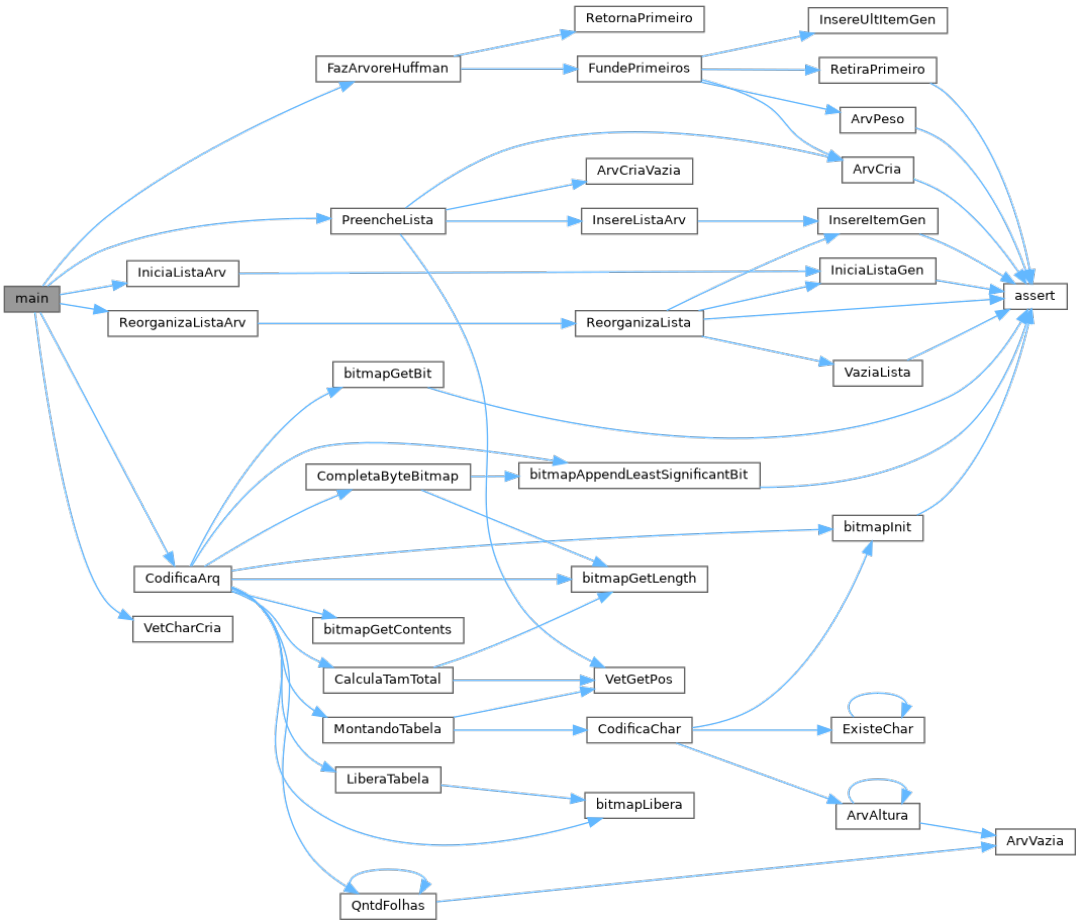
```c
//------------------------- Funções Privadas ----------------------------------------//
/**
 * @brief Abre o arquivo de saída com o nome do arquivo
 *        de entrada
 *
 * @param path - local do arquivo
 * @return FILE* - saida (<nome>.comp)
 */
static FILE *AbreSaida(char path[200]);

/**
 * @brief
 *
 * @param path - local e nome do arq. de entrada
 * @param saida  - arq de saida
 */
static void ColocaExtensaoNaSaida(char path[200], FILE *saida);

/**
 * @brief Monta o cabeçalho do arquivo comprimido
 *
 * @param arvorebase - arvore de huffman
 * @param saida - arquivo de saida
 */
static void ColocaArvoreNaSaida(Arv *arvorebase, FILE *saida);

/**
 * @brief Libera a memoria dinamicamente alocada
 *
 * @param ent - arquivo de entrada
 * @param saida - arquivo de saida
 * @param vetor - vetor de frenquencia de char
 * @param lista - lista de arvore que carrega a arvore de huffman
 */
static void LiberaCodificador(FILE *ent, FILE *saida, VetChar *vetor, Listagen *lista);
//------------------------- Funções Privadas ----------------------------------------//
```

A *main* segue esse fluxograma:



Olhando o código:

```c
int main(int argc, char const *argv[])
{

    if (argc < 2)
    {
        printf("USO: ./prog <nomedoarquivo>\n");
        exit(1);
    }

    char path[200];
    strcpy(path,argv[1]);


    // Abre arquivo de entrada
    FILE *arquivo = fopen(path, "r");
    // Abre arquivo de saida/////////
    FILE *saida = AbreSaida(path);

    ColocaExtensaoNaSaida(path, saida);

    // Cria vetor de frequencia de char
    VetChar *VetorFreq = VetCharCria(arquivo);

    // Faz lista de arvores
    Listagen *lista = IniciaListaArv();
    PreencheLista(lista, VetorFreq);

    // Prepara para Algoritmo de Huffman(menor para maior)
    lista = ReorganizaListaArv(lista);

    // Algoritmo
    Arv *arvorebase = FazArvoreHuffman(lista);

    // Faz o cabeçalho do arquivo de saida
    ColocaArvoreNaSaida(arvorebase, saida);

    // Codifica o arquivo de acordo com a arvore de huffman
    CodificaArq(arquivo, arvorebase, VetorFreq, saida);

    // Liberando memoria dinamica alocada
    LiberaCodificador(arquivo, saida, VetorFreq, lista);

    return 0;
}
```

Vemos que a *main* começa verificando caso o programa foi chamado corretamente, depois passamos o caminho para o arquivo salvando no path, abrimos o arquivo de entrada e a saida, depois usamos uma função para colocar a extensão do arquivo e seu tamanho em bytes (4(tamanho) .txt por exemplo) na saída, segue a função:

```c
static void ColocaExtensaoNaSaida(char path[200], FILE *saida)
{
    char ext[200];
    sscanf(path, "./%*[^.]%s", ext);
    fprintf(saida, "%ld%s", strlen(ext), ext);
}
```

Após isso criamos o vetor de frequência de caracteres para formar a arvore de codificação, mas antes colocamos os caracteres presentes em arvores e inserimos na lista, que logo após é organizada para usar o algorítimo de *Huffman.* Com isso pronto montamos a arvore e colocamos ela na saída, colocamos seu tamanho em bits antes para saber quando parar de ler a arvore, completamos o bitmap com 0 e exportamos:

```c
Arv *FazArvoreHuffman(Listagen *listabase)
{
    while (FundePrimeiros(listabase))
    {
    }
    return RetornaPrimeiro(listabase);
}
```

```c
static void ColocaArvoreNaSaida(Arv *arvorebase, FILE *saida)
{
    bitmap *SaidaArvore = ExportaArvore(arvorebase);
    unsigned int tamArv = bitmapGetLength(SaidaArvore);
    fprintf(saida, "%d", tamArv);
    CompletaByteBitmap(SaidaArvore);
    fwrite(bitmapGetContents(SaidaArvore), 1, bitmapGetLength(SaidaArvore) / 8, saida);
    bitmapLibera(SaidaArvore);
}
```

Depois começamos a realmente compactar o arquivo com a função *CodificaArq*.

```c
void CodificaArq(FILE *arq, Arv *Huffman, VetChar *Vetor, FILE *arqSaida)
{
    // 14MB
    bitmap *saida = bitmapInit(112000000);
    int tam = QntdFolhas(Huffman);

    Tabela *tab = MontandoTabela(Huffman, Vetor, tam);

    // Escreve o tamanho total do arquivo - Subcabecalho do arquivo
    unsigned long int TAM_TOTAL = CalculaTamTotal(Vetor, tab, tam);
    fprintf(arqSaida, "%ld", TAM_TOTAL);

    // Codificacao do arquivo de fato
    unsigned char aux;
    int index;
    bitmap *codificando;
```

Acima vemos que ela começa inicializando um *bitmap* de 14mb e pegamos o tamanho da tabela de codificação vendo quantas folhas a arvore de *Huffman* tem, montamos a tabela e calculamos quanto o arquivo vai ter de tamanho e escrevemos na saída. CalculaTamTotal:

```c
unsigned long int CalculaTamTotal(VetChar *Vetor, Tabela *tab, int tam)
{
    unsigned long int TAM_TOTAL = 0;
    for (int i = 0; i < MAX_VET; i++)
    {
        if (VetGetPos(Vetor, i) != 0)
        {
            for (int j = 0; j < tam; j++)
            {
                if (tab->carac[j] == (unsigned char)i)
                {
                    TAM_TOTAL += VetGetPos(Vetor, i) * bitmapGetLength(tab->codigo[j]);
                }
            }
        }
    }
    return TAM_TOTAL;
}
```

A função acima simplesmente calcula o tamanho do código gasto por cada caractere da tabela de codificação vezes seu peso para saber quantos bits no total serão usados.

```c
    while (!feof(arq))
    {
        // Ao ler um byte
        if (fread(&aux, 1, 1, arq))
        {
            // Procura o mesmo na tabela
            index = 0;
            while (index < tam && tab->carac[index] != aux)
            {
                index++;
            }
            codificando = tab->codigo[index];

            // Escreve no bitmap de saida o codigo do mesmo
            for (int i = 0; i < bitmapGetLength(codificando); i++)
            {
                bitmapAppendLeastSignificantBit(saida,
                                                bitmapGetBit(codificando, i));
            }
        }

        // Escreve em disco caso buffer esteja maior que 7MB (50%) e tenha bytes completos
        if (bitmapGetLength(saida) > 56000000 && (bitmapGetLength(saida) % 8) == 0)
        {
            fwrite(bitmapGetContents(saida), 1, bitmapGetLength(saida) / 8, arqSaida);
            bitmapLibera(saida);
            bitmap *saida = bitmapInit(112000000);
        }
    }

    // Garantia de que caso tenha algo a ser escrito, seja escrito
    if (bitmapGetLength(saida))
    {
        // completa o resto do byte com 0's
        CompletaByteBitmap(saida);
        fwrite(bitmapGetContents(saida), 1, bitmapGetLength(saida) / 8, arqSaida);
    }

    bitmapLibera(saida);
    LiberaTabela(tab, tam);
    rewind(arq);
```

O resto da função faz o seguinte, lê um byte, procura seu código na tabela e escreve no bitmap de saída, caso este mapa tenha ultrapassado 7mb nós escrevemos na saída e liberamos ele, assim evitamos de gastar muita memória usando o bitmap.

Depois que o arquivo todo foi compactado nós verificamos se ainda tem algo na saída completamos com 0 e escrevemos na saída. Liberamos o mapa a tabela e voltamos o ponteiro do arquivo para o inicio com *rewind*.

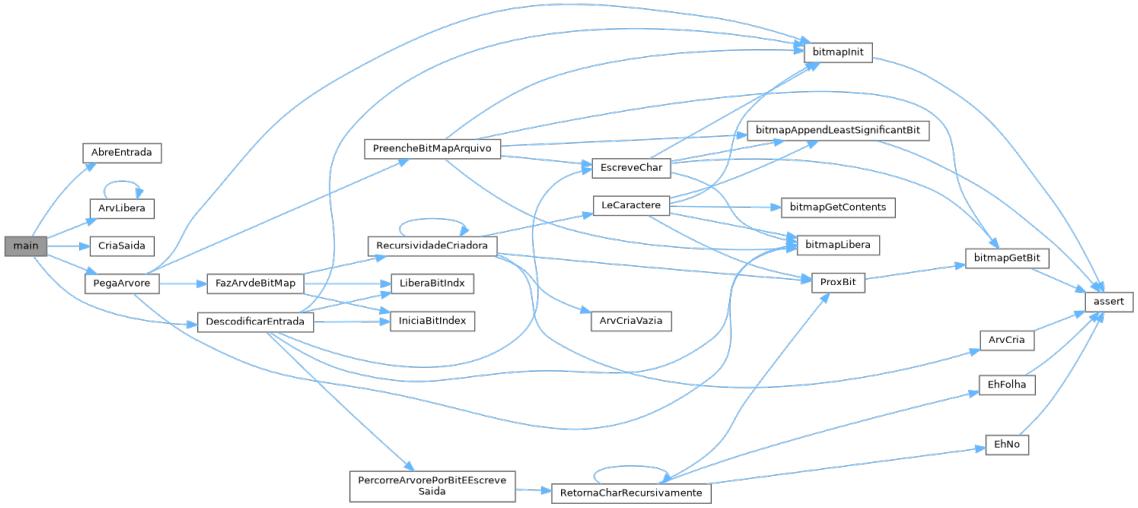
Voltando a *main* o próximo passo é liberar tudo:

```c
static void LiberaCodificador(FILE *ent, FILE *saida, VetChar *vetor, Listagen *lista)
{
    fclose(ent);
    fclose(saida);
    LiberaVetChar(vetor);
    LiberaListaArv(lista);
}
```

Obs.: Como a lista ainda tem a raiz da arvore de *Huffman* liberando a lista nós liberamos a Arvore e assim termina o Compactador.

TAD Decodificador:

O nome é decodificador, porém este TAD é responsável pela descompactação do arquivo. Começando pela *main* temos o seguinte fluxograma:



Como podemos ver o TAD BitIndex é utilizado aqui para que possamos fazer manipulação mais tranquila de bits.Segue as funções utilizadas e o que elas fazem em um curto resumo (a frente explicaremos melhor):

Functions

int	**main** (int argc, char const *argv[])
FILE *	**AbreEntrada** (const char *arg) Faz a abertura de um arquivo cujo caminho foi passado. More...
FILE *	**CriaSaida** (FILE *entrada, const char *path) Dada a informacao presente no cabecalho, eh criado o arquivo para a saida da decodificacao. More...
Arv *	**PegaArvore** (FILE *entrada) Monta uma arvore de Huffman com base no cabecalho presente no arquivo de entrada. More...
void	**PreencheBitMapArquivo** (**bitmap** *arv, FILE *arquivo, int qntBit) Preenche bitmap com a quantidade de bits informada do arquivo passado. More...
void	**DescodificarEntrada** (FILE *entrada, **Arv** *arvore, FILE *saida) Decodifica o arquivo de entrada com base na arvore de Huffman passada Escrevendo o resultado no arquivo de saida. More...

Vamos olhar o código da main:

```c
int main(int argc, char const *argv[])
{
    if (argc < 2)
    {
        printf("USO: ./prog <nomedoarquivo>\n");
        exit(1);
    }

    // abrindo entrada.
    FILE *entrada = AbreEntrada(argv[1]);
    // abrindo saida
    FILE *saida = CriaSaida(entrada, argv[1]);

    // Leitura de arvore
    Arv *arvore = PegaArvore(entrada);

    // DEBUG
    // ArvImprime(arvore);

    // ler arquivo e usar arvore para descodificar
    DescodificarEntrada(entrada, arvore, saida);

    // Liberacao de memoria
    fclose(entrada);
    fclose(saida);
    ArvLibera(arvore);

    return 0;
}
```

Podemos ver que o programa começa da mesma forma que o compactador, fazemos verificações e abrimos os arquivos de entrada e saída, para fazer isso lemos quantos bytes vai ser gasto para a extensão e logo depois lemos a extensão da entrada, com ela em mãos a saída é aberta.

```c
FILE *CriaSaida(FILE *entrada, const char *path)
{
    char aux[200];
    strcpy(aux, path);
    // pegando somente o nome sem a extensao .comp
    char aux2[200];
    sscanf(path, "./%[^.]", aux2);

    // pegando a extensao usada
    int ndebytes = 0;
    fscanf(entrada, "%d", &ndebytes);
    // +1 por causa do \0
    unsigned char ext[ndebytes + 1];
    // lendo extensao
    for (int i = 0; i < ndebytes; i++)
    {
        fscanf(entrada, "%c", &ext[i]);
    }
    ext[ndebytes] = '\0';
    // concatenando nome do arquivo + extensão
    unsigned char aux3[strlen(aux2) + strlen(ext) + 1];
    sprintf(aux3, "%s%s", aux2, ext);

    // abrindo
    return fopen(aux3, "w");
}
```

Logo após já conseguimos pegar a arvore de codificação pelo cabeçalho com a função PegaArvore utilizando o a entrada. Segue a função abaixo:

```c
Arv *PegaArvore(FILE *entrada)
{
    // buscando quantidade de bits gasto pela codificacao da arv
    int qntDeBits;
    fscanf(entrada, "%d", &qntDeBits);
    // fread(qntDeBits,1,1,entrada);

    // iniciando bitmap com o tamanho, +7 so p debug e garantia
    bitmap *arvore = bitmapInit(qntDeBits + 7);

    PreencheBitMapArquivo(arvore, entrada, qntDeBits);
    Arv *arvoresaida = FazArvdeBitMap(arvore);
    bitmapLibera(arvore);

    return arvoresaida;
}
```

Conseguimos ver claramente os passos da função aqui acima, primeiro lemos a quantidade de bits gastos pela arvore, iniciamos um bitmap e preenchemos o bitmap com a arvore usando a função EscreveChar já explicada anteriormente. Segue a função:

```c
void PreencheBitMapArquivo(bitmap *arv, FILE *arquivo, int qntBit)
{
    unsigned char temp;

    int qntBy = qntBit / 8;
    int preenchimento = 8 - (qntBit % 8);
    if (qntBit % 8)
        qntBy++;

    int i = 0, j = 0;
    while (i < qntBy)
    {
        fread(&temp, 1, 1, arquivo);

        // Se nao eh o final da arvore
        if (j < qntBit - 7)
        {
            EscreveChar(arv, temp);
            j += 8;
        }
        else // Ultimo byte
        {
            unsigned int aux = (unsigned int)temp;
            unsigned int enviado = 0;
            bitmap *temporarioInvertido = bitmapInit(8);
            for (int i = 0; i < 8; i++)
            {
                bitmapAppendLeastSignificantBit(temporarioInvertido, aux % 2);
                aux = aux / 2;
            }
            for (int i = 0; i < (qntBit - j); i++)
            {
                bitmapAppendLeastSignificantBit(arv, bitmapGetBit(temporarioInvertido, 7 - i));
                enviado = enviado / 10;
            }
            bitmapLibera(temporarioInvertido);
        }
        i++;
    }
}
```

Depois usamos a função do TAD Arvore FazArvdeBitaMap para fazer a arvore em si usando seu *bitmap,* logo após o *bitmap* é liberado e a arvore montada é retornada. Voltando a *main* depois de ter a arvore em mãos o próximo passo é decodificar o arquivo, usando a função DecodificarEntrada abaixo:

```c
void DescodificarEntrada(FILE *entrada, Arv *arvore, FILE *saida)
{
    // pega tam do arq (tam sem lixo)
    unsigned long int tamTotalBits;
    fscanf(entrada, "%ld", &tamTotalBits);

    unsigned char aux;

    int tamTotalBitsGastos = tamTotalBits;

    // tam com lixo
    while (tamTotalBitsGastos % 8)
        tamTotalBitsGastos++;

    // inicializando bitmap do arquivo
    bitmap *btmapArq = bitmapInit(tamTotalBitsGastos);
    BitIndex *arquivo = IniciaBitIndex(btmapArq);

    int counter = 0;

    while (counter != tamTotalBitsGastos / 8)
    {
        if (fread(&aux, 1, 1, entrada) == 0)
            break;

        // escreve um byte em bitmap
        EscreveChar(btmapArq, aux);
        counter++;
    }

    PercorreArvorePorBitEEscreveSaida(arquivo, arvore, tamTotalBits, saida);

    bitmapLibera(btmapArq);
    LiberaBitIndx(arquivo);
}
```

Como podemos ver na função acima após ler o tamanho total de bits gastados pelo arquivo em si iniciamos um bitmap com esse tamanho e transformamos ele em BitIndex, depois lemos até o final do arquivo e gravamos no bitmap.

Logo após usamos a função PercorreArvorePorBitEEscreveSaida que roda a arvore de acordo com o bitmap (1 para direita e 0 para a esquerda), para encontrar os devidos caracteres de acordo com o que está na entrada. Depois escrever na saída o resultado final, que é no caso o arquivo descompactado.

Depois toda a memória é liberada.

Conclusão:

Um dos trabalhos que mexe com o mais baixo nível em C foi bem interessante, com certeza aprendemos muito fazendo ele. Uma das partes mais difícil foi fazer algoritmos que trabalhassem com o binário em si, mesmo já tendo a biblioteca do bitmap disponível ainda foi necessário criar a BitIndex para descompactar os arquivos. Uma experiência que com certeza iremos carregar durante nossas jornadas como estudantes de Ciência da Computação.

Grande parte do aprendizado que extraímos desse trabalho se deve a essa complexidade de 0’s e 1’s, que normalmente é abstraída.