Universidade Federal do Espírito Santo

Trabalho Prático

Descompactador de Huffman

Estrutura de Dados - 2025/1

Alunos: Artur Vítor Cintra Bernardes e Felipe Pedrini Oliveira

Objetivo

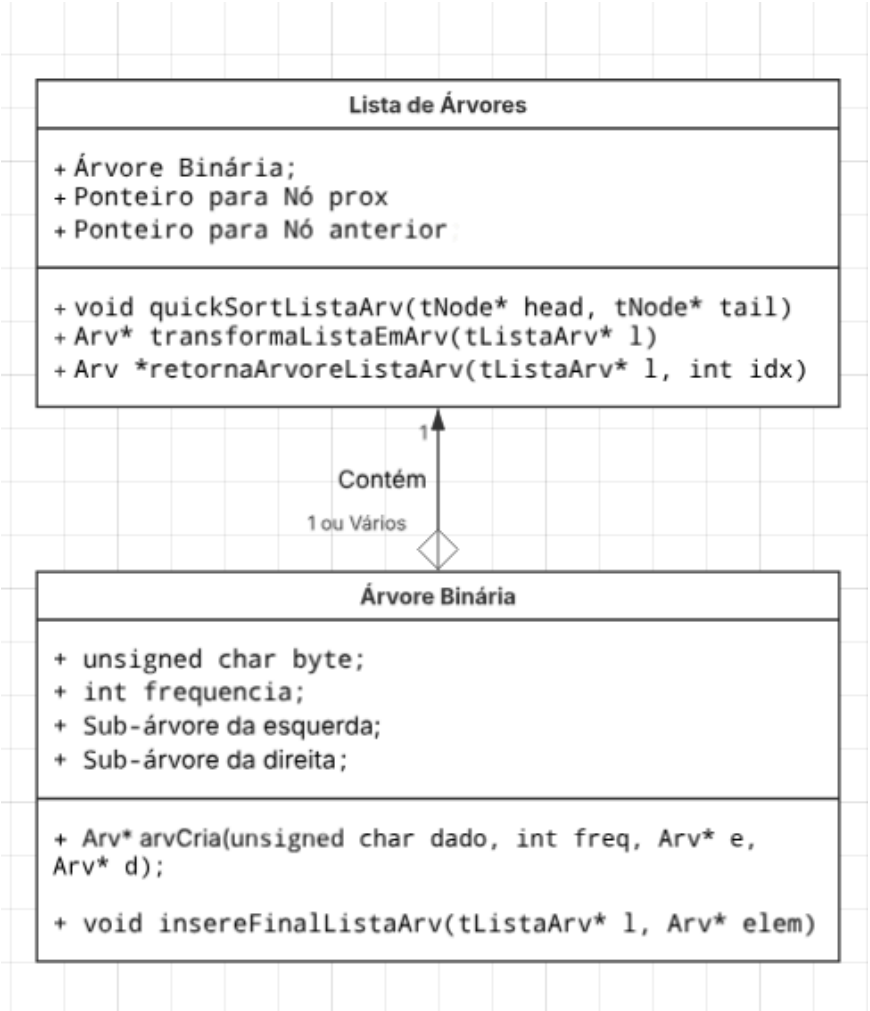
O objetivo deste trabalho é colocar em prática as habilidades de programação na linguagem C adquiridas ao longo do curso. O trabalho foi elaborado de forma a compactar e descompactar arquivos utilizando o algoritmo da árvore de Huffman, que é um método que usa as probabilidades de ocorrência dos símbolos no conjunto de dados a ser compactado para determinar códigos de tamanho variável para cada símbolo.

Introdução

Os compactadores foram muito utilizados no passado quando as mídias de armazenamento tinham preços elevados e era necessário economizar espaço para armazenamento.
No cotidiano, é comum precisarmos compactar e descompactar um arquivo, seja para enviar para alguma pessoa, diminuindo o tempo de transferência deles sobre uma rede, ou simplesmente para ele ocupar menos espaço no armazenamento. Tendo isso em mente, foi desenvolvido um compactador que segue a metodologia do processo de Codificação de Huffman, ao qual comprimi um arquivo sem perda de dados e pode também o descomprimir.

Implementação

O programa foi implementado seguindo o processo de Codificação da Árvore de Huffman que tem como ideia básica utilizar menos bits para representar caracteres que aparecem com uma maior quantidade de vezes. Para isso, precisamos utilizar uma lista encadeada que possui como elementos árvores binárias, que ao passar do tempo se fundem em uma única árvore e armazenam os caracteres nas suas folhas, fazendo com que o caminho até eles seja a sequência de bits que são usados para codificar os respectivos caracteres.



Principais Funções:

static int serializaArvoreBuffer(Arv *a, unsigned char *buffer, int pos): Essa função permite-nos serializar a nossa árvore em um buffer, que posteriormente será impresso no arquivo compactado, permitindo que posteriormente consigamos descompactar o arquivo seguindo o passo a passo que a árvore nos fornece.

void geraTabelaCompactador(Compactador *c): Gera a tabela que nos permite traduzir os caracteres para o caminho da árvore, essencial para a compactação.

static void preencheBitmapDescompactador(Descompactador *d, FILE *arquivo): Função que lê o arquivo e salva o bitmap dentro da estrutura do descompactador, permitindo uma maior facilidade na hora de descompactar o arquivo de forma definitiva.

Arv* transformaListaEmArv(tListaArv* l): Une os dois primeiros elementos da lista em uma nova árvore binária com raiz vazia e frequência igual à soma das frequências das duas subárvores, além disso esses dois primeiros elementos são excluídos da lista. O processo se repete, de forma recursiva, até restar apenas uma árvore na lista, esta árvore tem como frequência na raiz a soma da frequência de todas as outras sub-árvores.

void quickSortListaArv(tNode* head, tNode* tail): Ordena os nós da lista em ordem crescente de frequência, utilizando o algoritmo QuickSort adaptado para listas duplamente encadeadas.

void count_frequency(FILE* input, int* frequency_array, int n): Recebe um vetor de frequências, seu tamanho e recebe um arquivo binário. Essa função lê o arquivo e incrementa a frequência de cada caractere que aparece, sendo otimizado de forma que cada caractere ocupa o próprio índice na tabela ASCII.

Decisões:

Foi decidido utilizar TADs para o Compactador e Descompactador para facilitar a leitura e modularidade. No mais, foi seguido as especificações do trabalho.

Conclusão

O trabalho de Codificação de Huffman permitiu compreender de forma prática como algoritmos de compressão reduzem o tamanho de dados a partir da frequência de símbolos. A principal dificuldade encontrada foi fazer a depuração do código a partir do momento em que ele gerava um arquivo compactado, este que estava em binário e dificultava a leitura, sendo necessário ferramentas externas para facilitar a mesma. Outra complexidade foi a tomada de decisões para que o trabalho ficasse o mais simples, legível e otimizado possível, visto que trabalhar com binários exige um cuidado a mais com a legibilidade do código. Além disso, foi necessário atenção ao processo de codificação e decodificação para evitar inconsistências. No geral, a implementação trouxe uma visão clara da importância do balanceamento entre teoria e prática em algoritmos de compressão.

Bibliografia

- https://www.geeksforgeeks.org/dsa/quicksort-for-linked-list/
- CELES, Waldemar. **Introdução a Estruturas de Dados - Com Técnicas de Programação em C**.
- https://stackoverflow.com/questions/30007665/how-can-i-store-value-in-bit-in-c-language
- https://www.geeksforgeeks.org/c/extract-bits-in-c/