UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
CURSO CIÊNCIA DA COMPUTAÇÃO

ELAINE DIAS PIRES
FILIPE GOMES ARANTE DE SOUZA

RELATÓRIO 1° TRABALHO PRÁTICO DE ESTRUTURAS DE DADOS: PLAYED

Vitória
2021

ELAINE DIAS PIRES
FILIPE GOMES ARANTE DE SOUZA

RELATÓRIO 1° TRABALHO PRÁTICO DE ESTRUTURAS DE DADOS: PLAYED

Trabalho apresentado à disciplina de Estruturas de Dados da Universidade Federal do Espírito Santo - UFES, como requisito parcial para avaliação semestral.

Professor: Patrícia Dockhorn Costa
Turma: 2020/1

VITÓRIA
2021

SUMÁRIO

1 INTRODUÇÃO

Neste trabalho temos uma plataforma de músicas fictícia, a qual pessoas podem se cadastrar, adicionar amigos e separar suas músicas favoritas em playlists.

Inicialmente, as playlists de cada usuário do PlayED estão organizadas conforme o gênero musical. Por exemplo, uma pessoa X poderia ter playlists de **rock**, **samba** e **pagode**, enquanto uma pessoa Y poderia ter playlists de **pop**, **sertanejo** e **eletrônica**.

Uma das tarefas a ser feita era reorganizar as playlists de cada pessoa de acordo com cantor ou banda.

Por exemplo, as playlists da pessoa X, depois de refatoradas, poderiam ser:

- Iron Maiden, Scorpions e Queens (artistas de **rock**);
- Zeca Pagodinho e Paulinho da Viola (artistas de **samba**);
- Alexandre Pires e Thiaguinho (artistas de **pagode**);

Já as da pessoa Y, depois de refatoradas, poderiam ser:

- Nick Minaj, Rihanna e Shakira (artistas de **pop**);
- Michel Teló, Marília Mendonça e Gustavo Lima (artistas de **sertanejo**);
- Akon, Jennifer Lopez e David Guetta (artistas de **eletrônica**);

Após a reorganização das listas de cada pessoa cadastrada na plataforma, é pedido que seja calculada a similaridade entre cada par de amigos, isto é, calcular quantas músicas eles têm em comum.

Para resolver este problema, utilizamos nosso conhecimento sobre tipos abstratos de dados (TADs), listas encadeadas e manipulação de arquivos que aprendemos ao longo do curso.

2 IMPLEMENTAÇÃO

2.1 TADS

Para modularizar o trabalho, criamos as seguintes TADs:

- TAD lista_pessoa
- TAD pessoa
- TAD lista_playlist
- TAD playlist
- TAD musica

O trabalho é sobre um “mini spotify”, então o TAD música é essencial, pois ele possui os campos chave para resolver os problemas propostos. Como cada pessoa possui várias músicas de diversos gêneros, criamos o TAD playlist para agrupar músicas por gêneros. Também percebemos que cada pessoa poderia ter mais de uma playlist, então criamos o TAD lista playlist para agrupar essas playlists. O TAD pessoa é primordial e criamos o TAD lista pessoa para fazer um tratamento de todas as pessoas do played.

Todas as listas citadas são simplesmente encadeadas com sentinelas para a primeira célula, a última célula e o tamanho (quantidade de elementos).

Segue abaixo definição das structs para entendimento da organização e hierarquização dos TADS:

```
struct musica{
    char* nome;
    char* artista;
};

struct playlist{
    CelMusica* first;
    CelMusica* last;
    char* nome;
    int tam;
};

struct lista_playlist{
    CelPlaylist* first;
    CelPlaylist* last;
    int tam;
};

struct pessoa{
    char* nome;
    Lista_playlist* songs;
    Lista_pessoa* amigos;
};

struct lista_pessoa{
    CelPessoa* first;
    CelPessoa* last;
    int tam;
};
```

Figura 1: Definição das structs.

Segue também abaixo diagrama ilustrado da organização entre os TADS:

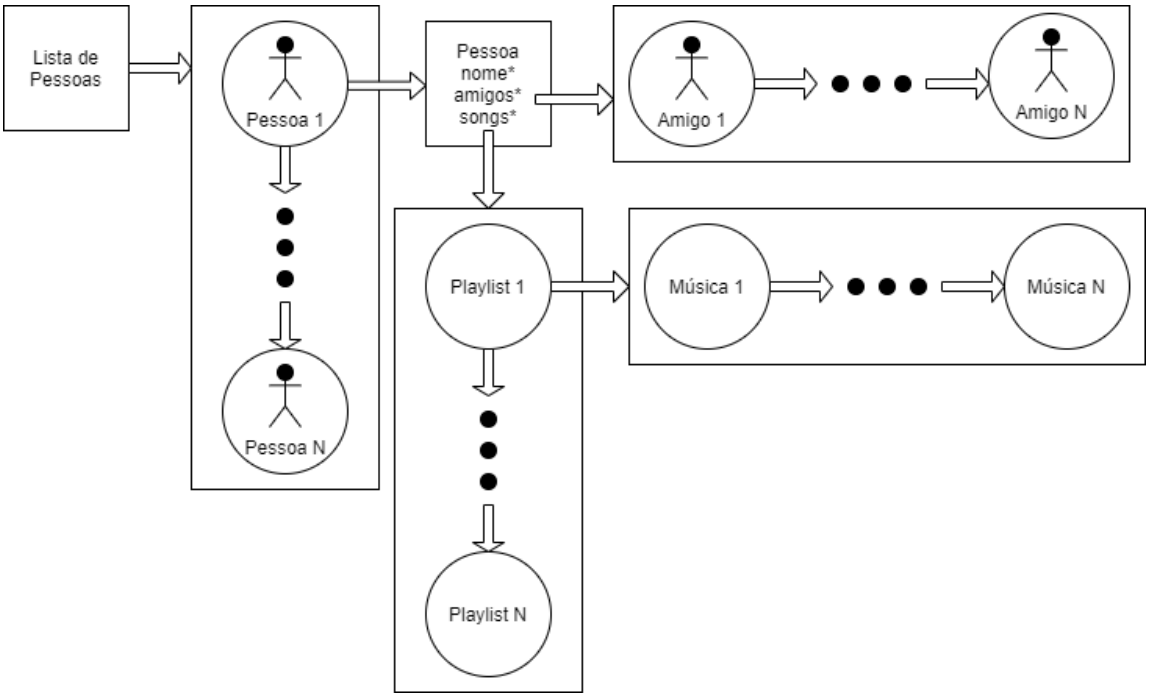

Figura 2: Ilustração da organização entre os TADS.

2.2 PRINCIPAIS FUNÇÕES

```
Lista_pessoa* inicializaUsuarios(char* fileNameAmizades, char* fileNamePlaylists);
```

A função inicializa usuários é muito importante pois é nela que fazemos a leitura dos arquivos amizade.txt e chamamos a função `inserePlaylistsNasPessoas`, que faz a leitura do arquivo playlists.txt . Na função `inicializaUsuarios`, nós criamos a lista de pessoas do played e inicializamos seus campos: nome, lista de amigos e lista de playlists.

```
void refatoraListaPlaylistDaListaPessoa(Lista_pessoa* listaPessoa);
```

A função `refatoraListaPlaylistDaListaPessoa` é a função mais externa de várias outras funções que refatoram. Primeiro, nós criamos uma função que refatora uma playlist de uma pessoa, depois criamos uma função que refatora uma lista de playlist de uma pessoa e por fim, criamos `refatoraListaPlaylistDaListaPessoa` que refatora a lista de playlist de todas as pessoas do PlayED.

```
void geraArquivosSaida(Lista_pessoa* listaPessoa);
```

Função responsável por gerar todos os outputs do trabalho. Ela chama as funções de impressão de arquivos, que são:

```
escrevePlaylistsRefatoradasArquivo(listaPessoa);
```

Escreve no arquivo played-refatorada.txt. É responsável por mostrar o resultado da refatoração das playlists de cada pessoa do PlayED.

```
imprimeListaPlaylistDaListaPessoaNoArquivo(listaPessoa);
```

Cria uma pasta para cada pessoa do PlayED e gera arquivos com as músicas que possuem cada playlist após a refatoração delas.

```
void similaridade(Lista_pessoa* listaPessoa);
```

A função **`similaridade`** calcula quantas músicas em comum possui cada par de amigos do PlayED e imprime os dados no arquivo similaridades.txt.

Para descobrir o valor de cada similaridade, varremos a lista de amigos de cada pessoa e varremos as listas de playlists de cada amigo comparando com uma pessoa fixada e contabilizando num contador quando encontrasse uma música igual.

Há um problema no momento de imprimir o arquivo: Se não houver tratamento, as similaridades serão impressas em dobro. Por exemplo, se Maria e João são amigos, no arquivo seria impressa a similaridade de João e Maria e depois de Maria e João. Para não ocorrer essa duplicação, criamos um campo “int similaridade” em cada célula de uma lista de pessoas, que são inicializadas com o valor -1. Após o cálculo de um par de amigos, o valor descoberto é setado nesse campo tanto na célula que contém maria na lista de amigos de joão e vice-versa. Assim, esse campo serve como um marcador de passagem, ou seja, se já imprimiu no arquivo ou não. Se chegássemos num par de amigos e o valor do campo similaridade fosse -1, significa que essa dupla de amigos ainda não foi escrita no arquivo. Se fosse diferente de -1, já teria sido escrita.

3 DIRETÓRIOS

Dividimos o trabalho nas seguintes pastas:

- **src:** pasta que contém os arquivos .c do trabalho.
- **include:** pasta que contém os arquivos .h do trabalho.
- **client:** pasta que contém o arquivo main do trabalho.
- **data:** pasta que contém as pastas de entrada e saída do trabalho. A pasta Entrada contém os arquivos amizades.txt, playlists.txt e todas as playlists, tais como acoustic.hits.txt . A pasta Saida contém os arquivos gerados pelo nosso programa.

4 MAKEFILE

Criamos um makefile para facilitar a compilação do trabalho.

Para compilar o programa basta utilizar o comando: **`$ make`**

Para executar, basta digitar: **`$ make run`**

Para apagar os arquivos objeto e o executável, digite: **`$ make clean`**

Para apagar as saídas de um teste da pasta data/Saida, digite: **`$ make reset`**

5 CONCLUSÃO

O trabalho foi bem interessante. Pela primeira vez trabalhamos com várias listas ao mesmo tempo, e devido a isso aprendemos bastante e amadureceu ainda mais os conceitos ensinados nas aulas síncronas e assíncronas do curso.

Sentimos dificuldade na parte de pensar sobre como criar o PlayED, pois se não planejássemos bem a infraestrutura do código, seria bem mais difícil resolver os problemas do trabalho, além de provavelmente nos fazer demorar mais tempo.

Um dos maiores desafios do começo do trabalho foi o fato de cada pessoa ter uma lista de pessoas como campo (uma lista de amigos). Como pegamos todos os usuários do played e os armazenamos numa lista de pessoas, ocorreu uma espécie de recursão e tivemos o problema de círculo de inclusão entre bibliotecas.

Ainda no início da criação do código, interpretamos mal a leitura dos arquivos na especificação e estávamos passando todos eles (incluindo os arquivos de músicas, como sertanejo.txt, eletronica.txt, etc) na linha de comando utilizando argc e argv. Após perceber nosso equívoco, tivemos de refazer algumas funções.

Além disso, quando estávamos começando a programar a refatoração das playlists, nós conseguimos pensar como executá-la, mas ao passar para código estava bastante complicado. Nesse ponto percebemos o quão importante é modularizar o código e suas funções, dividindo uma tarefa em várias outras menores, assim tornando o programa mais legível. Essa ideia foi perfeitamente aplicada na implementação do cálculo da similaridade e nos facilitou bastante a desenvolvê-la, além do restante do trabalho como um todo.

6 BIBLIOGRAFIA

Slides e vídeo aulas disponibilizados no classroom da turma.