UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA
ESTRUTURAS DE DADOS I

Gabrielly Barcelos Cariman
Lucas Manfioletti

PLAYED!

Vitória-ES

Agosto/2021

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

Este trabalho tem como objetivo aprender de forma prática fundamentos ensinados na matéria de Estrutura de Dados I, ministrada pela Professora Dr. Patrícia Dockhorn Costa.

Nele são utilizados conceitos como Tipos Abstratos de Dados (TADs), Lista Simplesmente Encadeada com Sentinela, leitura e escrita em arquivos, alocação dinâmica de memória e recursão, tudo utilizando a linguagem de programação C.

O problema apresentado para que fosse resolvido, é implementar de forma simplificada um aplicativos de streaming de música (PlayED!), que teria a funcionalidade de gerenciar uma rede de amigos que vão poder compartilhar músicas entre si em que cada pessoa contém as suas próprias músicas, playlists e amizades para definirem as musicas compartilhadas.

Para isso, foi utilizado principalmente TADs para organizar as informações de Música, Playlist e Pessoa. Além de que, esses TADs eram todos organizados em Listas Simplesmente Encadeadas com Sentinela, que também são TADs, uma para cada tipo diferente.

Cada um desses TADs possui dois arquivos, um onde ele é sinalizado que existe juntamente com suas funções específicas e documentação (extensão .h) e outro onde ele é de fato declarado e implementado (extensão .c), ou seja, onde está o código que possui a lógica de cada função.

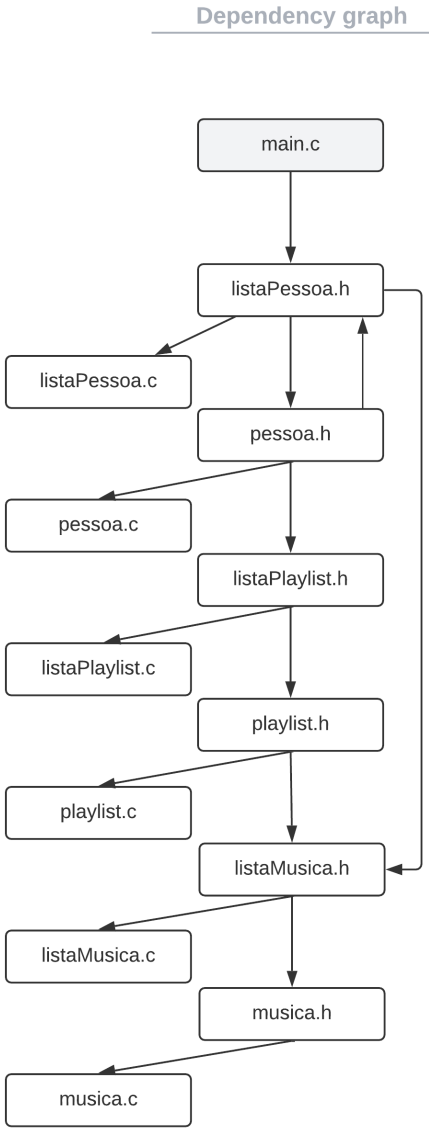
2 METODOLOGIA

2.1 Organização dos arquivos

Todos os diagramas apresentados neste trabalho foram feitos na plataforma Lucidchart ().

Os arquivos criados para a organização de todo o código do programa estão na Figura 1.

Figura 1 – Gráfico de Dependência



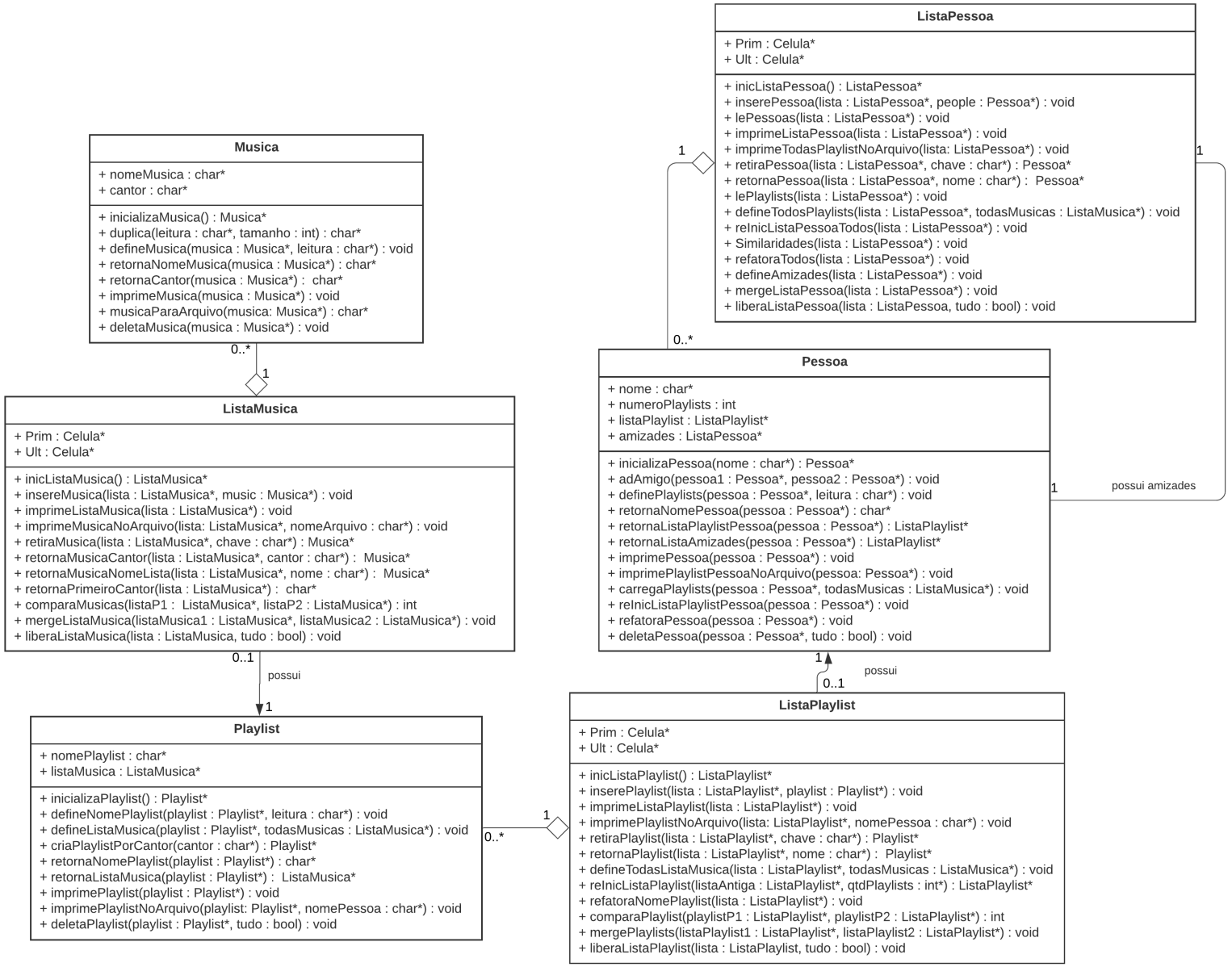
Fonte: Produção do próprio autor.

Note que para cada TAD existe um arquivo '.h' e outro '.c', e como foi abordado na introdução, o primeiro contém a declaração dos seus métodos e sua documentação, e o segundo a sua definição com a implementação da lógica das funções.

2.2 Modelagem

A modelagem dos TADs, o que cada um possui de atributos, métodos e os seus relacionamentos, pode ser observado no diagrama de classes UML na Figura 2.

Figura 2 – Diagrama de Classes



Fonte: Produção do próprio autor.

Analisando a Figura 2, percebe-se que os TADs possuem a seguinte estrutura (note que Celula não possui uma estrutura própria, isso será explicado mais adiante):

1. Musica - TAD que faz a organização de como uma Musica é armazenada na memória.
    - nomeMusica : char* - armazena o ponteiro para a string do nome da Musica;
    - cantor : char* - armazena o ponteiro para a sitring do nome do cantor/banda da Musica.
2. ListaMusica - TAD que faz a organização de como uma lista de Musicas é armazenada na memória.
    - Prim : Celula* - armazena o ponteiro para a primeira Celula da lista;
    - Ult : Celula* - armazena o ponteiro para a última Celula da lista.
3. Playlist - TAD que faz a organização de como uma Playlist é armazenada na memória.
    - nomePlaylist : char* - armazena o ponteiro para a string do nome da Playlist;
    - listaMusica : ListaMusica* - armazena o ponteiro para uma ListaMusica que possui todas as Musicas da Playlist.
4. ListaPlaylist - TAD que faz a organização de como uma lista de Playlists é armazenada na memória.
    - Prim : Celula* - armazena o ponteiro para a primeira Celula da lista;
    - Ult : Celula* - armazena o ponteiro para a última Celula da lista.
5. Pessoa - TAD que faz a organização de como uma Pessoa é armazenada na memória.
    - nome : char* - armazena o ponteiro para a string do nome da Pessoa;
    - numeroPlaylists : int - armazena o inteiro com o número de Playlists da Pessoa;
    - listaPlaylist : ListaPlaylist* - armazena o ponteiro para uma ListaPlaylist que possui todas as Playlist da Pessoa.
6. ListaPessoa - TAD que faz a organização de como uma lista de Pessoas é armazenada na memória.
    - Prim : Celula* - armazena o ponteiro para a primeira Celula da lista;
    - Ult : Celula* - armazena o ponteiro para a última Celula da lista.

Todas as listas implementadas no PlayED são listas com sentinelas. Isso porque, além desse tipo de estrutura apresenta maior segurança ao ser manipulado, por manter sempre o mesmo enderenço de memoria do primeiro elemento, a sentinela, uma restrição do trabalho era fazer todas as inserções ao final da lista. Dessa forma, é mais econômico fazer essas tipo de inserção com uma estrutura que já aponta para o final, a sentinela. Ademais, foi escolhido também que elas fossem simplesmente encadeadas em vez de duplamente, isso por que não há a necessidade de percorrer as listas no sentido contrário na maioria das funções.

As listas com sentinelas possuem a TAD Celula com a seguinte estrutura:

1. Celula
    - dado: TipoDado*
    - prox: Celula*

Cada lista possui sua própria implementação do TAD Celula, mas todos seguem a mesma lógica. Por exemplo: Na lista de Musica, ao invés de Celula possuir o atributo 'dado' do tipo 'TipoDado*', ela possui o atributo 'musica' (o nome do atributo não faz diferença, apenas é mais intuitivo colocar 'musica' nesse caso) do tipo 'Musica*'. E como dito, esse padrão se repete para as outras listas existentes.

2.3 Funções Importantes

Como foi dito anteriormente, cada TAD possui uma gama de funções diferentes, que realizam diversas operações que são essenciais para o funcionamento do programa como um todo. Além disso, existe também a main que utiliza as funções dos TADs.

Simplificando, cada TAD tem suas respectivas funções de inicialização, retornar algum valor, liberar espaço na memória, e no caso das listas, inserção, remoção e retorno depois de uma busca.

É importante destacar que as funções de liberar espaço na memória e remoção acontecem da seguinte forma: Quando o cliente utiliza a função de liberar uma lista de playlists, por exemplo, além de liberar o espaço da lista em si, também é liberado cada uma das playlists. E ao utilizar a função de remover uma playlist da lista de playlists, por exemplo, a playlist removida não tem seu espaço de memória liberado, apenas a Celula tem. Entretanto, vale

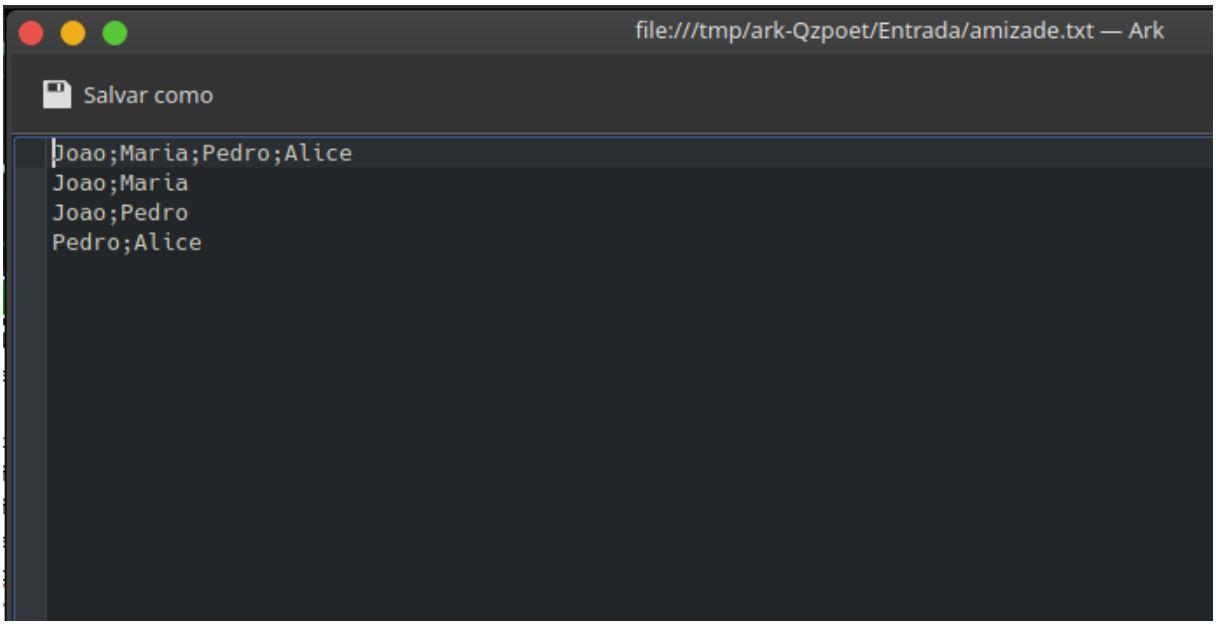
notar que as funções de liberar das listas tem como parâmetro, além da lista, um booleano 'tudo'. Caso o booleano 'tudo' passado seja 1, a função fará exatamente o que foi descrito acima. Caso seja 0, a função terá o mesmo comportamento descrito acima, menos para as músicas, apenas as músicas serão mantidas na memória. Outra observação importante é que na função de libera de ListaPessoa, ao definir o booleano 'tudo' como 0, apenas as Celulas e a sentinela são liberada. Caso ele seja 1, o funcionamento já foi descrito.

As funções mais complexas e interessantes serão explicadas a seguir:

1. Função para ler a primeira linha do arquivo amizades.txt e criar as pessoas da lista de pessoas:

    - **TAD ListaPessoa: void lePessoas(ListaPessoa *lista)** - o arquivo 'amizades.txt' é aberto no modo de leitura. Enquanto estiver tudo certo e a a leitura ainda estiver na primeira linha, os nomes separados por ';' serão lidos. A linha é lida usando o 'fgets' e o final do arquivo é verificado com o 'feof', essas função foram utilizadas após pesquisas em Backes (2019). Outra variável recebe os caracteres da string lida até o ';', ou seja, ela recebe o nome da pessoa que está disposta na primeira linha, fazendo assim a leitura de cada caractere até que seja encontrado o ';' ou quebra de linha, como exemplo do arquivo disposto na Figura 3.

Figura 3 – Arquivo de amizades



Fonte: Produção do próprio autor.

A função 'duplica' é chamada passando a leitura realizada (nome da pessoa) e a quantidade de caracteres para a leitura, retornando uma string com o nome da

pessoa. Esse nome é utilizado para chamar a função 'inicializaPessoa', a qual cria uma pessoa com o nome da string passado, e por fim, é chamada a função 'inserePessoa', a qual é passada a pessoa criada e a lista passada pela função principal 'lePessoa', fazendo então a inserção dessa pessoa na lista de pessoas passada. Por fim, o arquivo é fechado com a função 'fclose', baseada em Backes (2019).

2. Funções para ler o arquivo playlists.txt e criar as playlists para as pessoas do arquivo:

   - **TAD ListaPessoa: void lePlaylists(ListaPessoa *lista)** - o arquivo 'playlists.txt' é aberto no modo de leitura. Enquanto estiver tudo certo e o final do arquivo não tiver chegado ao final, as linhas do arquivo vão ser lidas até a quebra de linha. As linhas são lidas usando o 'fgets' e o final do arquivo é verificado com o 'feof', essas função foram utilizadas após pesquisas em Backes (2019). Outra variável recebe os caracteres da string lida até o ';', ou seja, ela recebe o nome da pessoa que tem as playlists listadas. A função 'retornaPessoa' é chamada passando a lista do parâmetro e o nome da pessoa que acabou de ser lida, essa por sua vez, retorna a variável pessoa encontrada com o nome passado.Agora, a função 'definePlaylists' é chamada e é passada a pessoa encontrada na lista e o resto da linha lida que contém o número de playlists que a pessoa tem e o nome de cada playlist.
   - **TAD Pessoa: void definePlaylists(Pessoa *pessoa, char *leitura)** - primeiro, o primeiro caractere da string passada é lido e passado para modificar o atributo de numeroPlaylists da pessoa passada como parâmetro. A função 'inicListaPlaylist' é chamada e é inicializada uma Lista de Playlists, um for até o numeroPlaylists é iniciado e outra variável recebe os caracteres da string lida até os ';'. Dessa forma, essa variável recebe uma string no formato 'nomePlaylist.txt'. As funções 'inicializaPlaylist', 'defineNomePlaylist' e 'inserePlaylist' são chamadas e uma playlist é criado com nome lido e inserida na lista de Playlists criada anteriormente. A função 'defineNomePlaylist' recebe como parâmetro a playlist criada e a string lida no formato anteriormente descrito.
   - **TAD Playlist: void defineNomePlaylist(Playlist *playlist, char *leitura)** - uma variável recebe os caracteres da string passada até o ponto final. O campo nomePlaylist da playlist passada recebe o endereço dessa variável alocada dinamicamente com o nome da playlist, into é, ela recebe apenas o nome da playlist, sem o '.txt'.

3. Funções para ler e carregar as músicas de todos os arquivos de playlists:

   - **TAD ListaPessoa: void defineTodosPlaylists(ListaPessoa *lista, ListaMusica *todasMusicas)** - essa função anda por cada item da lista, chamando

a função 'carregaPlaylists', passando cada uma das pessoas da lista até o final dela.

- **TAD Pessoa: void carregaPlaylists(Pessoa *pessoa, ListaMusica *todasMusicas)** - essa função apenas chama a função 'defineTodasListaMusica' passando como parâmetro a lista de playlists da pessoa em questão.
- **TAD ListaPlaylist: void defineTodasListaMusica(ListaPlaylist *lista, ListaMusica *todasMusicas)** - essa função anda por cada item da lista, chamando a função 'defineListaMusica', passando cada uma das playlists da lista até o final dela.
- **TAD Playlist: void defineListaMusica(Playlist *playlist, ListaMusica *todasMusicas)** - uma lista de música é iniciada. Após pesquisas sobre a biblioteca 'string.h' em Casavella () e Costa (2016), utilizando o 'strcpy', é realizada a cópia do campo nomePlaylist e, com o 'strcat', a concatenação dessa variável com '.txt'. Isso irá gerar uma variável com o seguinte formato 'nomePlaylist.txt'. A partir disso, um arquivo é aberto com o nome informado anteriormente e é lida uma linha do arquivo até a quebra de linha, isso será feito até o final do arquivo. Uma Musica é inicializada e as funções 'defineMusica' e 'insereMusica' são chamadas para preencher os campos da Musica e inseri-la na lista de músicas que será usada para modificar o campo 'listaMusica' da playlist em questão. Além dos parâmetros já mencionados de todas as funções acimas, cada uma delas também tem mais um parâmetro o 'todasMusicas' que é um ponteiro para ListaMusica. Essa variável é uma Lista de Musicas que armazena todas as Musicas instanciadas no programa, ela é usada no final para liberar todas as Musicas alocadas. A função 'defineMusica' recebe como parâmetro a musica criada e a string lida no formato 'NomeCantor - NomeMusica'.
- **TAD Musica: void defineMusica(Musica *musica, char *leitura)** - primeiro, é verificada se a string leitura passada como parâmetro apresente como último caractere a quebra de linha. Caso ela não possua, é alocada uma nova string dinamicamente com a quebra de linha e uma flag é mudada de valor de 0 para 1, para que no final, a memória dessa nova variável seja liberada. A partir disso, duas strings estáticas são criadas. Agora, é feita a verificação de caractere por caractere da string passada como parâmetro até o '-', esses caracteres são armazenados em uma das variáveis estáticas criadas. Quando o '-' é encontrado, a função 'duplica' é chamada, passando a variável estática e o número de caracteres verificados até o '-', desconsiderando o caractere do espaço. Essa função será explicada posteriomente, mas ela retorna a string alocada dinamicamente que será utilizada para modificar o campo de 'cantor' da Musica. O mesmo processo é utilizado para modificar o campo 'nomeMusica',

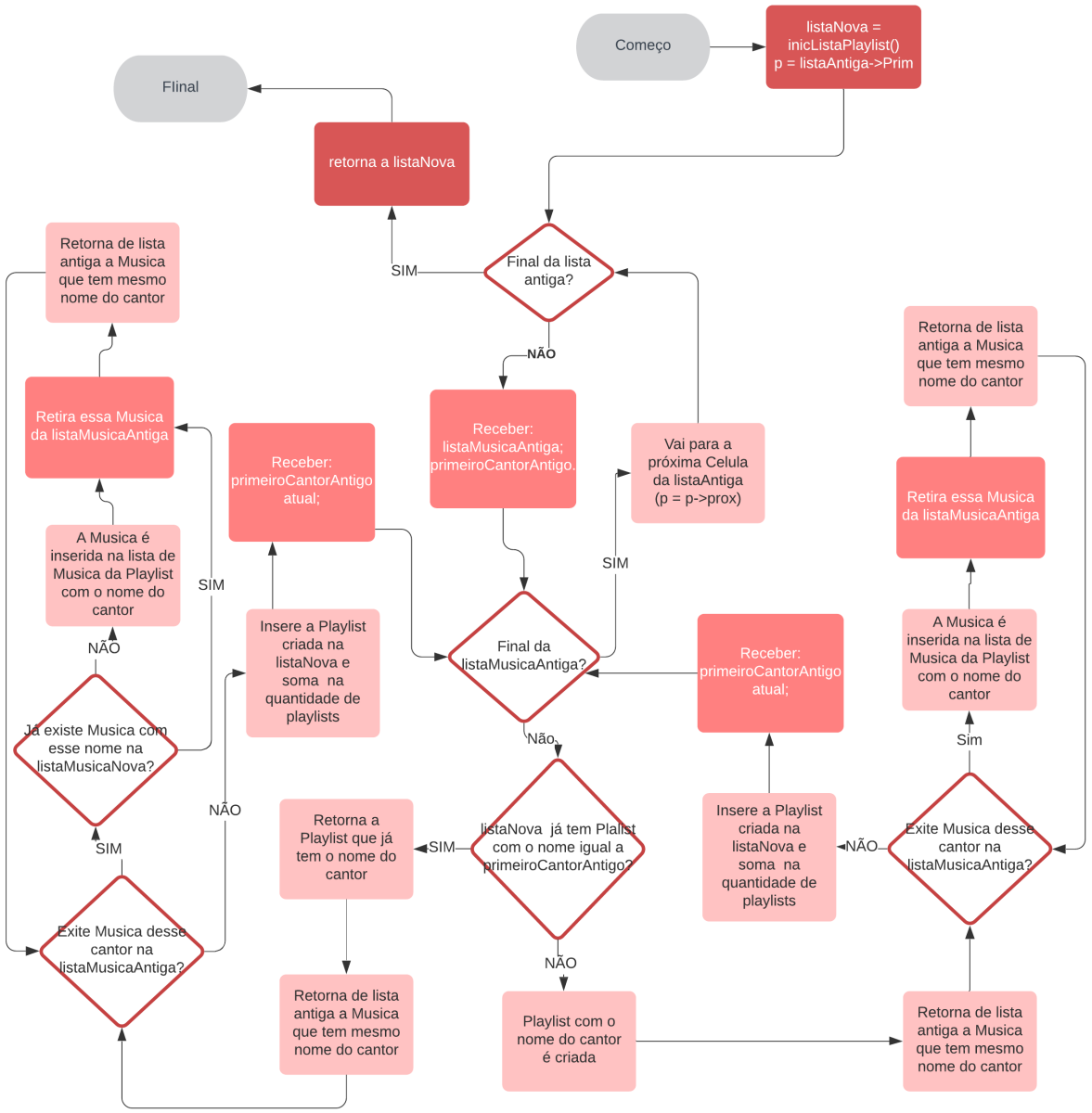
mas, primeiro são desconsiderados os caracteres '- ' e a verificação dos caracteres é feito até o a quebra de linha.

- **TAD Musica: char *duplica(char *leitura, int tamanho)** - A função 'duplica' está no TAD de Musica, mesmo não existindo ligação direta dela com Musica. Isso porque, é uma função que é utilizada muito no programa, inclusive em outros TADs, então faz sentido ela ser definida no TAD de mais baixo nível de 'include'. Essa função, recebe uma string estática e o tamanho que ela deve ter para ser alocada dinamicamente. Portanto, é feito um 'malloc' para armazenar o conteúdo da string passado e para o identificador de final de string. A partir disso, um for é utilizado para passar o caractere da string estática para a dinâmica até o tamanho final da string.

4. Funções para a partir das playlists iniciais, sejam criadas playlists por cantor ou banda para cada pessoa:

   - **TAD ListaPessoa: void reInicListaPlaylistTodos(ListaPessoa *lista)** - essa função anda por cada item da lista, chamando a função 'reInicListaPlaylistPessoa', passando cada uma das pessoas da lista até o final dela.
   - **TAD Pessoa: void reInicListaPlaylistPessoa(Pessoa *pessoa)** - nessa função uma nova lista de playlists, não instanciada, recebe a função 'reInicListaPlaylist'. Depois disso, a função 'liberaListaPlaylist' é chamada passando a listaPlaylist da pessoa e o valor booleano zero. Portanto, apenas as músicas que estão armazenadas serão mantidas na memória, como já explicado no início da Seção 2.3. A partir disso, campo 'listaPlaylist' e 'numeroPlaylists' da pessoa é modificado. A função 'reInicListaPlaylist' recebe como parâmetro a lista de playlists antiga da pessoa em questão e o endereço da variável inteira com a quantidade de playlists e retorna uma nova lista de playlists.
   - **TAD ListaPlaylist: ListaPlaylist *reInicListaPlaylist(ListaPlaylist *lista1, int *qtdPlaylists)** - para fazer todas as verificações de Musicas, nome de cantor, nome de Playlists foram usadas as funções: 'retornaListaMusica', 'retornaPrimeiroCantor', 'retornaPlaylist', 'retornaMusicaCantor', 'retornaMusicaNomeLista' e 'retornaNomeMusica'. Por ser uma das funções mais complexas do programa, será apresentado o Flowchart desse algoritmo na Figura 4.

Figura 4 – Flowchart de reInicListaPlaylist



Fonte: Produção do próprio autor.

Nessa função uma nova lista de playlist é inicializada. Essa função vai andar por cada item da lista de playlists passada como parâmetro até o seu fim. Quando entra em uma Celula nova, ela recebe a ListaMusica da Playlist da Celula e uma variável recebe o nome do cantor que está localizado na primeira Celula da ListaMusica da Playlist da Celula atual. Enquanto a lista de musicas não chegar ao fim, é feita uma verificação na nova lista de playlists. Caso não tenha uma playlist com o mesmo nome do cantor recebido, uma nova playlist é criada. A musica que tem como campo o nome do cantor usado para a verificação é retornada. É feita outra verificação, enquanto existirem musicas desse cantor nessa lista de musicas, ela é inserida na nova lista de músicas com musicas só de determinado cantor. Depois disso, essa determinada música é retirada da lista antiga de músicas dela. Uma nova verificação é feita para checar outras músicas com o mesmo cantor na lista de músicas antigas, até acabarem as músicas desse cantor nessa lista.

Por fim, essa nova playlist criada é inserida na nova lista de playlist e é somado 1 ao conteúdo da variável com a quantidade de playlists. Se já existir uma playlist na nova lista com o mesmo nome do cantor da primeira Celula da lista de Musica que está sendo usada para a verificação, uma busca é feita na lista de playlists novas e é retornada a playlist com o mesmo nome do cantor. Agora, o mesmo processo é feito como descrito anteriormente. A musica com o nome do cantor é retornada, adicionado a parte de verificar se ela já existe na nova lista de música, se ela não existir, é inserida na lista. Depois, é retirada da lista antiga e esse processo ocorre, enquanto existirem musicas desse determinado cantor na lista de músicas antiga.

Após todo esse processo, o próximo primeiro cantor será chamado da lista de músicas antigas e tudo será repetido até que não exista um próximo primeiro canto na lista de músicas antigas. Quando esse momento chegar e a lista de músicas antigas estiver vazia, o processo será repetido para a próxima playlist até o final da lista de playlist.

5. Funções que geram o arquivo de saída 'played-refatorada.txt':

   - **TAD ListaPessoa: void refatoraTodos(ListaPessoa *lista)** - essa função exclui o arquivo 'played-refatorada.txt' com a função 'remove', vista em Backes (2019). Além disso, anda por cada item da lista, chamando a função 'refatoraPessoa', passando cada uma das pessoas da lista até o final dela.
   - **TAD Pessoa: void refatoraPessoa(Pessoa *pessoa)** - essa função abre o arquivo 'played-refatorada.txt' com a opção de adicionar no final do arquivo ou, se o arquivo não existir, criar o arquivo. Se o arquivo for aberto com sucesso, a função 'fprintf', vista em Backes (2019), é usada para imprimir

'nomePessoa;numeroPlaylists;' no arquivo. Se a gravação dessas informações tiver sucesso, a função 'refatoraNomePlaylist' é chamada.

- **TAD ListaPlaylist: void refatoraNomePlaylist(ListaPlaylist *lista)** - essa função abri o arquivo 'played-refatorada.txt' com a mesma opção descrita acima. Essa função anda por cada item da lista de playlists, imprimindo o nome de cada Playlist e ';' no arquivo. É feita uma verificação se a Celula atual é a última, se for, após o nome da Playlist não é colocado um ';', mas sim uma quebra de linha.

6. Funções para gerar os arquivos com as novas playlists, organizadas por pastas separadas pelo nome da pessoa:

   - **TAD ListaPessoa: imprimeTodasPlaylistsNoArquivo(ListaPessoa *lista)** - essa função anda por todos os item da lista, chamando a função 'imprimePlaylistPessoaNoArquivo', passando cada uma das pessoas da lista até o final dela.
   - **TAD Pessoa: void imprimePlaylistPessoaNoArquivo(Pessoa *pessoa)** - primeiro, a função 'static int criardiretorio(char *nome)', vista em Durub (2008), é chamada passando o nome da pessoa como parâmetro. Essa função verifica o sistema operacional e de acordo com ele, chama uma função que cria uma pasta com o nome passado como parâmetro. Depois disso, a função 'imprimeListaPlaylistNoArquivo' é chamada, passando a lista de Playlists e o nome da pessoa.
   - **TAD ListaPlaylist: void imprimeListaPlaylistNoArquivo(ListaPlaylist *lista, char *nomePessoa)** - essa função anda por todos os item da lista, chamando a função 'imprimePlaylistNoArquivo', passando cada uma das playlists da lista e o nome da Pessoa recebido como parâmetro.
   - **TAD Playlist: void imprimePlaylistNoArquivo(Playlist *playlist, char *nomePessoa)** - uma variável de ponteiro de char é alocada em um espaço suficiente para para conter os caracteres de './nomePessoa/nomePlaylist.txt', usando as funções de 'strcpy' e 'strcat' essa string vai receber os argumentos do nome da Pessoa e nomePlaylist. Um arquivo com esse mesmo nome é criado vazio para escrita. Isto é, um arquivo com o nome da Playlist é criado dentro da pasta que tem o nome da pessoa. Depois disso, a função 'imprimeMusicasNoArquivo' é chamada passando a listaMusica da Playlist e o nome do arquivo que já foi todo concatenado anteriormente.
   - **TAD ListaMusica: void imprimeMusicasNoArquivo(ListaMusica *lista, char *nomeArquivo)** - o arquivo com o nome passado é aberto com a opção de adicionar escrita no final. A função 'musicaParaArquivo' é chamada, passando uma Musica da lista de cada vez, essa retorna uma string. É feita uma

verificação se está na última Musica, se estiver o último da string retornada, anteriomente a quebra de linha, é trocado por um identificador de final de string. Por fim, a string recebida é colocada no arquivo com a função 'fputs', vista em Backes (2019).

- **TAD Musica: char *musicaParaArquivo(Musica *musica)** - uma variável de ponteiro de char é alocada em um espaço suficiente para para conter os caracteres de 'cantor - nomeMusica' com uma caracter de quebra de linha no final. Usando as funções de 'strcpy' e 'strcat' essa string vai receber os argumentos do nomeMusica e cantor da Musica e o caracter de quebra de linha. A função retorna a string concatenada no formato de 'cantor - nomeMusica', essa será impressa no arquivo como já foi descrito na função acima.

7. Funções para ler e criar as listas de amigos:

   - **TAD Pessoa: void adAmigo(Pessoa *pessoa1, Pessoa *pessoa2)** - A função recebe duas pessoas, e adiciona na lista de pessoas de amizade, de uma pessoa a outra, e vice-versa.
   - **TAD ListaPessoa: void defineAmizades(ListaPessoa *lista)** - essa função faz a abertura do arquivo amizades.txt, através de métodos de leitura obtidos em Backes (2019), e faz a leitura inicial da primeira linha, através da função 'fscanf' para que a mesma seja ignorada, já que ela somente apresenta o nome de todas as pessoas. Em seguida, é criado um laço de repetição para que o arquivo seja lido até o seu final ou seja, sua leitura seja diferente de 'EOF', assim, são obtidos os nomes das duas pessoas que são amigas, e com a função 'retornaPessoa' obtêm-se as duas pessoas amigas. Assim, uma é adiciona na lista de amizades da outra, e vice-versa com a função 'adAmigo' Após, o arquivo é fechado com a função 'fclose' pesquisada em Backes (2019), e a função encerrada.

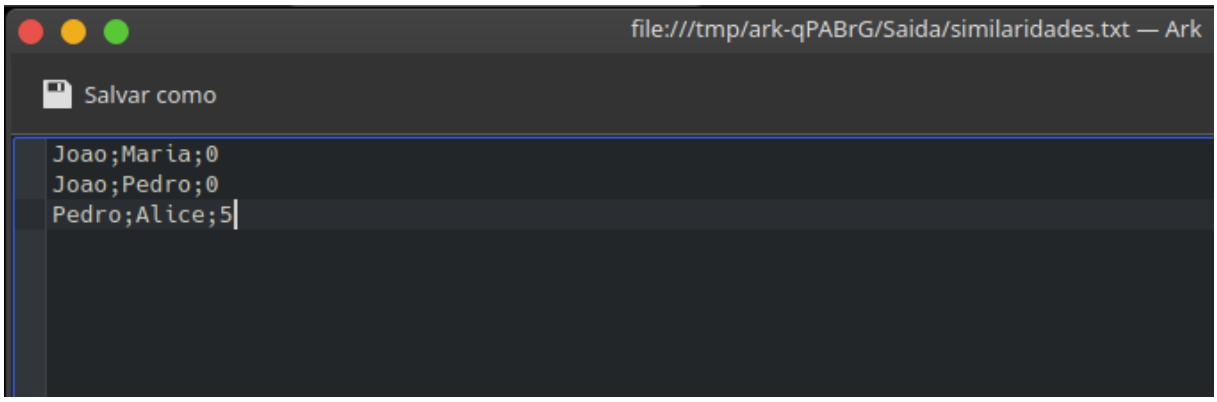
8. Funções para gerar o arquivo de similaridades entre as pessoas:

   - **TAD ListaMusica: int comparaMusicas(ListaMusica *listaP1, ListaMusica *listaP2)** - A função recebe duas listas de música e utiliza de dois laços de repetição para varrer as duas listas de músicas, comparando-as, e utilizando de um contador inteiro para as similaridades, retornando assim o valor de musicas iguais.
   - **TAD ListaPlaylist: int comparaPlaylist(ListaPlaylist *playlistP1, ListaPlaylist *playlistP2)** - A função recebe duas listas de playlists e navega comparando todas as playlists entre si, de forma a retornar o valor inteiro da quantidade de músicas em comum de cada lista, para isso utilizando de dois laços de repetição para navegação entre as playlists e chamando a função

'comparaMusicas', passando duas listas de músicas das duas playlists em questão no laço de repetição.

- **TAD ListaPessoa: void Similaridades(ListaPessoa *lista)** - essa função faz a abertura do arquivo amizades.txt, através de métodos de leitura obtidos em Backes (2019), e faz a leitura inicial da primeira linha, através da função 'fscanf' para que a mesma seja ignorada, já que ela somente apresenta o nome de todas as pessoas. Em seguida, é criado um laço de repetição para que o arquivo seja lido até o seu final ou seja, sua leitura seja diferente de 'EOF', assim, com o método 'fgets' é feita a leitura até o final da linha, e essa leitura é dividida por laços de repetição, de forma a separar duas strings, a primeira terminando no ";"e a segunda no caractere de quebra de linha, assim, essas duas strings com o nome de duas pessoas amigas, são atribuídas a duas variáveis. Esses nomes são passados a função 'retornaPessoa', que com o nome e a lista de pessoas, retorna a pessoa a qual o nome é passado. Essas duas pessoas então são passadas a função 'retornaListaPlaylistPessoa' que retorna a lista de playlists da pessoa passada. Assim, com duas listas de playlists das pessoas lidas naquela linha do arquivo, chama-se a função 'comparaPlaylist' passando as duas listas de playlists, retornando um valor inteiro atribuído a uma variável. Por fim, outro arquivo chamado similaridades.txt é criado e gravado com o nome das duas pessoas, e o valor de retorno da função 'comparaPlaylist' é gravado, como disposto na Figura 5.

Figura 5 – Arquivo de similaridades



Fonte: Produção do próprio autor.

Após, os dois arquivos abertos são fechados com a função 'fclose' pesquisada em Backes (2019), e a função encerrada.

9. Funções para fazer o 'merge' das playlists dos mesmos cantores/bandas mas, somente entre amigos:

- **TAD ListaPessoa: void mergeListaPessoa(ListaPessoa *lista)** - essa função percorre toda a lista de Pessoas passada e para cada pessoa da lista, ela retorna a lista de amigos da pessoa e para cada amigo chama a função 'mergePlaylists', passando a própria lista de playlist e também a lista do amigo atual. Quando ela termina de percorrer toda a lista passada, a função estática 'mergeListaPessoaEstatica' é chamada passando o primeiro elemento da lista de pessoas.
- **TAD ListaPlaylist: void mergePlaylists(ListaPlaylist *playlists1, ListaPlaylist *playlists2)** - essa função percorre toda a primeira lista de Playlists passada. Para cada elemento dessa lista, o seu nome é retornado e, usando a função 'retornaPlaylist', caso tenha uma playlist com o mesmo nome na segunda lista, ela é retornada. Se essa playlist de fato existir, a função 'mergeListaMusica' é chamada passando as listas de Musicas das duas Playlists que têm o mesmo nome.
- **TAD ListaMusica: void mergeListaMusica(ListaMusica *listaMusica1, ListaMusica *listaMusica2)** - primeiro, cada Musica da primeira lista de Musicas é percorrida e para cada uma é retornado o seu nome e verificado se essa Musica já está na segunda lista de Musicas. Se ela ainda não estiver, ela será inserida. Após isso, todo o processo será repetido. Porém, dessa vez, a lista percorrida será a segunda lista de Musicas, com a repetição do processo, as Musicas que estiverem na segunda lista, mas não na primeira, serão inseridas na primeira lista.
- **TAD ListaPessoa: static void mergeListaPessoaEstatica(Celula *p)** - depois de todo o processo descrito ser concluído e a lista de pessoas for varrida do início para o fim, essa função estática do TAD é chamada passando a Celula do primeiro elemento da lista de Pessoas. A função é usada de forma recursiva, visto em Costa (2015), a primeira verificação é o caso base da Celula enviada como parâmetro não ser nula. Isto é, se a Celula do parâmetro não for nula, a função 'mergeListaPessoaEstatica' é chamada novamente passando 'p->prox' para reduzir a recursão. Esse processo irá acontecer até o final da lista, porque o 'p->prox' do último elemento é nulo. Quando a lista chegar ao final e retornar para o escopo da última chamada da função, o mesmo processo de retornar as amizades e para cada amigo chamar a função 'mergePlaylists' passando as listas de Playlists da pessoa e do amigo será repetido. Contudo, dessa vez, a lista de pessoas será varrida do último elemento para o primeiro. O fato da lista ser varrida duas vez, uma do início para o final e depois do último elemento para o primeiro faz com que o 'merge' das Playlists ocorra para todos os amigos da lista, não importando a ordem ou localização em que eles se encontrem dentro da lista.

3 CONCLUSÃO

O trabalho com certeza foi de grande importância para fixar todos os conceitos abordados até a etapa atual que a disciplina se encontra. Além de servir como fixação, foi preciso uma pesquisa por conteúdo adicional, como a parte de leitura e escrita em arquivos e como trabalhar com eles.

Os maiores desafios presente no trabalho foram conseguir pensar em como implementar a função 'reInicListaPlaylist' e a 'mergeListaPessoaEstatica', assim como quais passos seguir para isso. Além disso, a manipulação de arquivos, a formatação de strings, assim como o armazenamento dessas estruturas na Heap, e a manipulação de diferentes sentinelas entre as listas, de forma a manipular funções entre listas diferentes, que levaram algum tempo para serem pesquisados, implementados e testados até que estivessem finalizados.

Trabalhos assim são de grande relevância durante a graduação, pois aproxima a teoria aos problemas encontrados no mundo e gera um sentimento de que a implementação poderia ser utilizado para resolver um problema real.

REFERÊNCIAS

BACKES, A. LINGUAGEM C: ARQUIVOS. 2019. Disponível em: <http://www.facom.ufu.br/~backes/gsi011/Aula00-Arquivos.pdf>. Acesso em: 15 ago. 2021. Citado 5 vezes nas páginas 9, 10, 14, 16 e 17.

CASAVELLA, E. A biblioteca string.h. Disponível em: <http://linguagemc.com.br/a-biblioteca-string-h/>. Acesso em: 15 ago. 2021. Citado na página 11.

COSTA, P. D. Aula 14: Recursão. 2015. Disponível em: <https://drive.google.com/file/d/1YKF-8swUrF5lqwQS8xGE5KRFQYKjd0uW/view>. Acesso em: 24 ago. 2021. Citado na página 18.

COSTA, P. D. Aula 6: Cadeias de Caracteres. 2016. Disponível em: <https://drive.google.com/file/d/1P--FrqR4hKY_QDHftTkO1LhKZOxGFgSC/view>. Acesso em: 15 ago. 2021. Citado na página 11.

DURUB. Criar Pasta. 2008. Disponível em: <https://forum.scriptbrasil.com.br/topic/127580-resolvido20criar-pasta/>. Acesso em: 15 ago. 2021. Citado na página 15.

LUCIDCHART. Disponível em: <https://www.lucidchart.com/pages/pt/landing?utm_source=google&utm_medium=cpc&utm_campaign=_chart_pt_allcountries_mixed_search_brand_exact_&km_CPC_CampaignId=1500131167&km_CPC_AdGroupID=59412157138&km_CPC_Keyword=lucidchart&km_CPC_MatchType=e&km_CPC_ExtensionID=&km_CPC_Network=g&km_CPC_AdPosition=&km_CPC_Creative=294337318298&km_CPC_TargetID=kwd-33511936169&km_CPC_Country=1031797&km_CPC_Device=c&km_CPC_placement=&km_CPC_target=&mkwid=seFG43w3S_pcrid_294337318298_pkw_lucidchart_pmt_e_pdv_c_slid___pgrid_59412157138_ptaid_kwd-33511936169_&gclid=CjwKCAjw9uKIBhA8EiwAYPUS3Ea5dWA8cWLzUHVOTkbkMxw5RMQwUEH2ItcKpVxdm18U7hTBwE>. Acesso em: 15 ago. 2021. Citado na página 4.