Empilhando Trabalhos

Repositório dos trabalhos para a matéria de estrutura de dados.

View the Project on GitHub matncb/empilhando-trabalhos

Sistema de Recomendação Musical Baseado em Similaridade

1. Introdução e Objetivos

1.1 O que o Sistema Faz

Este projeto implementa um **Sistema de Recomendação Musical** que utiliza uma rede de conexões entre pessoas baseada na similaridade de suas playlists musicais. O sistema permite:

- **Cadastrar pessoas** com suas respectivas playlists musicais
- **Calcular similaridades** entre pessoas baseado em músicas compartilhadas
- **Gerar recomendações musicais** personalizadas para cada usuário
- **Visualizar a rede de conexões** através de uma matriz de adjacência
- **Exportar o grafo** para arquivo CSV para análise externa
- **Organizar pessoas** por tamanho de playlist usando uma árvore AVL
- **Gerenciar playlists** de forma dinâmica (adicionar/remover músicas)
- **Redimensionar o grafo automaticamente** quando novas pessoas são adicionadas

1.2 Objetivos do Projeto

O sistema foi desenvolvido com os seguintes objetivos:

1. **Modelar uma rede social de recomendações musicais** usando grafos para representar relações de similaridade
2. **Implementar algoritmos de recomendação** baseados em análise de similaridade
3. **Fornecer uma interface interativa** que permite ao usuário explorar e manipular a rede
4. **Validar decisões de implementação** através de justificativas técnicas para cada escolha estrutural

1.3 Contexto Escolhido

O contexto escolhido foi uma **rede social de recomendações musicais**, onde:

- **Vértices do grafo**: Representam pessoas cadastradas no sistema
- **Arestas do grafo**: Representam relações de similaridade entre pessoas, ponderadas pelo número de músicas compartilhadas
- **Funcionalidades principais**: Busca de pessoas similares, geração de recomendações personalizadas, exportação da rede

---

2. Arquitetura Geral do Sistema

2.1 Fluxo de Execução

O sistema opera em três fases principais:

1. **Inicialização**: Carrega dados de arquivos CSV (nomes e músicas), cria 100 pessoas com playlists aleatórias, calcula similaridades e constrói o grafo
2. **Processamento**: Atualiza estruturas de dados conforme o usuário interage (adiciona/remove pessoas ou músicas)
3. **Interface Interativa**: Permite ao usuário executar comandos para explorar a rede e obter recomendações

2.2 Estrutura de Dados Principal

O sistema utiliza uma arquitetura onde:

- **PopList**: Armazena todas as pessoas cadastradas (lista principal)
- **Graph**: Representa as similaridades entre pessoas (matriz de adjacência)
- **Tree (AVL)**: Organiza pessoas por tamanho de playlist para buscas eficientes
- **PlayList**: Cada pessoa possui uma playlist (lista de músicas)
- **Queue**: Armazena temporariamente recomendações geradas

3. Detalhamento dos TADs Utilizados

3.1 TAD Grafo (Obrigatório)

3.1.1 Estrutura e Implementação

O TAD Grafo é implementado usando uma **matriz de adjacência** bidimensional:

```
typedef struct Graph {
    int vertex_qnt;      // Número de vértices
    int **adjMatrix;     // Matriz de adjacência
} Graph;
```

A matriz `adjMatrix` é uma matriz quadrada de tamanho `vertex_qnt × vertex_qnt`, onde:

- `adjMatrix[i][j]` armazena o peso da aresta entre os vértices `i` e `j`
- O peso representa o número de músicas compartilhadas entre as pessoas `i` e `j`
- O grafo é **não-direcionado** (simétrico): `adjMatrix[i][j] = adjMatrix[j][i]`
- O grafo é **ponderado**: os pesos variam de 0 (sem similaridade) até o número máximo de músicas compartilhadas

3.1.2 Justificativa: Matriz de Adjacência vs Lista de Adjacência

A escolha pela **matriz de adjacência** foi fundamentada nas seguintes razões:

1. Grafo Denso Esperado

- Em um sistema de recomendação musical, é esperado que muitas pessoas compartilhem pelo menos algumas músicas
- Com 100 pessoas, o grafo tende a ser relativamente denso (muitas arestas)
- Para grafos densos, a matriz de adjacência é mais eficiente em termos de:
  - **Acesso direto**: O(1) para verificar/atualizar uma aresta
  - **Espaço**: $O(V^2)$ é aceitável quando $E \approx V^2$ (muitas arestas)

2. Operações Frequentes

- O sistema precisa **consultar pesos de arestas constantemente** durante a geração de recomendações
- A matriz permite acesso O(1) a qualquer aresta, enquanto lista de adjacência requer O(V) no pior caso
- A operação `graph_get_edge_weight()` é chamada repetidamente no algoritmo de recomendação

3.1.3 Funcionalidades do Grafo

Criação (`graph_create`)

- Aloca memória para a estrutura e para a matriz
- Inicializa todos os pesos como 0 (sem arestas)
- Complexidade: $O(V^2)$ em tempo e espaço

Atualização de Arestas (`graph_update_edges`)

- Atualiza o peso de uma aresta entre dois vértices
- Mantém a simetria do grafo (não-direcionado)
- Complexidade: O(1)

Cálculo de Similaridade (`calculate_similarity`)

- Compara duas playlists e conta quantas músicas são iguais
- Utiliza comparação alfabética de nomes de músicas
- Complexidade: $O(M_1 \times M_2)$, onde M é o tamanho das playlists

Atualização do Grafo (`graph_update`)

- Recalcula todas as similaridades entre todos os pares de pessoas
- Utiliza dois loops aninhados para comparar cada par
- **Redimensiona o grafo automaticamente** se o número de pessoas exceder o tamanho atual
- Utiliza crescimento exponencial para evitar realocações frequentes

- Complexidade: O($V^2 \times M^2$), onde V é número de pessoas e M é tamanho médio das playlists

Redimensionamento Dinâmico (`graph_resize`)

- Redimensiona a matriz de adjacência quando necessário
- **Estratégia de crescimento exponencial**: Dobra o tamanho atual até atender à necessidade
- Evita realocações frequentes quando pessoas são adicionadas
- Preserva todos os dados existentes durante o redimensionamento
- Complexidade: O(N^2), onde N é o novo tamanho calculado
- **Exemplo**: Se o grafo tem 100 vértices e precisa de 101, cresce para 200 (não apenas 101)

Exportação (`export_graph`)

- Exporta a matriz de adjacência para um arquivo CSV (`graph.csv`)
- Permite visualização externa da rede de conexões
- Redimensiona o grafo automaticamente se necessário antes de exportar
- Exporta apenas até o número de pessoas cadastradas (não inclui espaços vazios)
- Complexidade: O(V^2)
- **Comando na interface**: `export` - Executa a exportação do grafo para CSV

3.1.4 Uso no Sistema

O grafo é utilizado em várias funcionalidades:

1. **Geração de Recomendações**: Consulta pesos de arestas para identificar pessoas similares
2. **Busca de Similares**: Lista todas as pessoas com similaridade > 0 para um usuário
3. **Visualização**: Exibe a matriz de adjacência para análise
4. **Atualização Dinâmica**: Recalcula similaridades quando playlists são modificadas

---

3.2 TAD PopList (Lista Duplamente Encadeada Ordenada)

3.2.1 Estrutura e Implementação

O TAD PopList é uma **lista duplamente encadeada ordenada alfabeticamente** por nome de pessoa:

```
typedef struct Element {
    struct Element *next;
    struct Element *prev;
    Person *person;
} Element;

typedef struct PopList {
    Element *start;
    Element *end;
    int elements;
} PopList;
```

3.2.2 Justificativa de Uso

Por que uma lista duplamente encadeada?

1. **Inserção Ordenada Eficiente**
   - Permite inserir elementos mantendo ordem alfabética
   - A busca para encontrar posição de inserção é O(n), mas a inserção em si é O(1)
   - A ordem alfabética facilita buscas e visualizações

2. **Remoção Eficiente**
   - Com ponteiros `prev` e `next`, a remoção é O(1) após encontrar o elemento
   - Não requer deslocamento de elementos como em arrays

3. **Flexibilidade**
   - Permite crescimento dinâmico sem realocação
   - Facilita adicionar/remover pessoas durante a execução

Por que não uma árvore de busca?

- A PopList é a estrutura principal que armazena todas as pessoas
- Precisa manter ordem de inserção relativa e permitir acesso sequencial
- A árvore AVL é usada para uma organização diferente (por tamanho de playlist)

3.2.3 Funcionalidades Principais

Inserção (`poplist_add`)

- Insere pessoa mantendo ordem alfabética
- Verifica duplicatas (mesmo nome)
- Complexidade: O(n) para encontrar posição + O(1) para inserir

Busca por Nome (`poplist_search_by_name`)

- Percorre a lista até encontrar pessoa com nome correspondente
- Complexidade: O(n) no pior caso

Remoção (`poplist_remove_by_name`)

- Remove pessoa da lista mantendo ordem
- Complexidade: O(n) para encontrar + O(1) para remover

Listagem (`poplist_people`)

- Retorna array com todas as pessoas (para uso em loops)
- Complexidade: O(n)

3.2.4 Uso no Sistema

- Armazena todas as pessoas cadastradas
- Fornece acesso sequencial para cálculos de similaridade
- Permite busca por nome para operações do usuário
- Serve como referência principal para índices do grafo

3.3 TAD Tree (Árvore AVL)

3.3.1 Estrutura e Implementação

O TAD Tree é uma **árvore AVL (Adelson-Velsky e Landis)** que organiza pessoas por tamanho de playlist:

```
typedef struct Element {
    struct Element *left;
    struct Element *right;
    Person *person;
    int balance_factor;
} Element;

typedef struct Tree {
    Element *root;
    int elements;
} Tree;
```

A árvore é balanceada automaticamente para garantir altura O(log n).

3.3.2 Justificativa de Uso

Por que uma árvore AVL?

1. **Busca Eficiente por Tamanho**
    - Busca por tamanho de playlist em O(log n)
    - Útil para encontrar pessoas com playlists de tamanho específico
    - Permite range queries eficientes

2. **Ordenação Automática**
    - Mantém elementos ordenados por tamanho de playlist
    - Permite listagem ordenada em O(n)
    - Facilita visualização de pessoas por tamanho de playlist

3. **Balanceamento Automático**
    - AVL garante altura máxima de $\log_2(n)$
    - Evita degeneração em lista (que ocorreria em BST simples)
    - Operações sempre O(log n) garantido

Por que não usar apenas a PopList?

- PopList ordena por nome, não por tamanho de playlist
- Busca por tamanho seria O(n) na PopList
- A árvore oferece organização alternativa útil para diferentes consultas

3.3.3 Funcionalidades Principais

Inserção (`tree_add`)

- Insere pessoa ordenada por tamanho de playlist
- Balanceia automaticamente após inserção
- Complexidade: O(log n)

Remoção (`tree_remove`)

- Remove pessoa com tamanho de playlist específico
- Balanceia automaticamente após remoção
- Complexidade: O(log n)

Busca (`tree_search_by_playlist_size`)

- Encontra pessoa com tamanho de playlist exato
- Complexidade: O(log n)

Listagem (`tree_list`)

- Retorna array ordenado (inorder, preorder ou postorder)
- Complexidade: O(n)

Rotações de Balanceamento

- `tree_rotate_left`: Rotação simples à esquerda
- `tree_rotate_right`: Rotação simples à direita
- `tree_balance_node`: Aplica rotações necessárias baseado no fator de balanceamento

3.3.4 Lógica de Balanceamento AVL

O balanceamento funciona da seguinte forma:

1. **Cálculo do Fator de Balanceamento**: `balance_factor = altura(direita) - altura(esquerda)`
2. **Verificação de Desbalanceamento**: Se `|balance_factor| > 1`, a árvore está desbalanceada
3. **Aplicação de Rotações**:
    - **Rotação Simples à Direita**: Quando subárvore esquerda está mais alta
    - **Rotação Simples à Esquerda**: Quando subárvore direita está mais alta
    - **Rotação Dupla**: Quando o desbalanceamento está em subárvore interna

3.3.5 Uso no Sistema

- Organiza pessoas por tamanho de playlist para visualização alternativa
- Permite busca eficiente por pessoas com playlists de tamanho específico
- Atualizada dinamicamente quando playlists são modificadas
- Fornece visualização ordenada diferente da PopList

Observação importante: Apesar de ser levemente diferente do que é feito no resto das aplicações, para um sistema integrado completo, é muito útil poder ordenar os resultados por tamanho de playlist. Esse passo constitui um primeiro

degrau para uma ampliação do projeto.

3.4 TAD PlayList (Lista Duplamente Encadeada Ordenada)

3.4.1 Estrutura e Implementação

O TAD PlayList é idêntico em estrutura ao PopList, mas armazena músicas ao invés de pessoas:

```
typedef struct Element {
    struct Element *next;
    struct Element *prev;
    Music *music;
} Element;

typedef struct PlayList {
    Element *start;
    Element *end;
    int elements;
} PlayList;
```

3.4.2 Justificativa de Uso

Por que lista duplamente encadeada ordenada?

1. **Ordem Alfabética**
    - Mantém músicas ordenadas alfabeticamente por nome
    - Facilita busca e verificação de duplicatas
    - Melhora visualização para o usuário

2. **Prevenção de Duplicatas**
    - A inserção ordenada permite verificar duplicatas durante a inserção
    - Evita músicas repetidas na mesma playlist

3. **Remoção Eficiente**
    - Remoção O(1) após encontrar a música
    - Não requer deslocamento de elementos

Por que não um array?

- Arrays requerem realocação quando crescem
- Inserção ordenada em array é O(n) para deslocar elementos
- Lista encadeada permite inserção O(1) após encontrar posição

3.4.3 Funcionalidades Principais

Inserção (`playlist_add`)

- Insere música mantendo ordem alfabética
- Verifica e rejeita duplicatas
- Complexidade: O(m) onde m é tamanho da playlist

Remoção (`playlist_remove_by_name`)

- Remove música por nome
- Complexidade: O(m)

Busca (`playlist_search_by_name`)

- Encontra música por nome
- Complexidade: O(m)

Listagem (`playlist_songs`)

- Retorna array com todas as músicas

- Complexidade: O(m)

3.4.4 Uso no Sistema

- Cada pessoa possui uma PlayList
- Usada no cálculo de similaridade (comparação entre playlists)
- Permite adicionar/remover músicas dinamicamente
- Mantém ordem alfabética para organização

3.5 TAD Queue (Fila Circular)

3.5.1 Estrutura e Implementação

O TAD Queue é uma **fila circular** implementada com array:

```
typedef struct {
    int bottom, top, elements;
    Music *musics[QUEUE_SIZE];
} Queue;
```

3.5.2 Justificativa de Uso

Por que uma fila circular?

1. **Armazenamento Temporário de Recomendações**
    - As recomendações são geradas e armazenadas temporariamente
    - Fila permite adicionar no final e processar do início
    - Estrutura adequada para armazenar sequência de recomendações

2. **Eficiência com Array**
    - Array permite acesso O(1) por índice
    - Fila circular reutiliza espaço quando elementos são removidos
    - Mais eficiente que lista encadeada para tamanho fixo conhecido

3. **Tamanho Limitado**
    - Recomendações têm limite máximo (5 músicas)
    - Array de tamanho fixo é adequado

Por que não uma lista encadeada?

- Tamanho máximo conhecido (5 recomendações)
- Array é mais eficiente para tamanho fixo
- Fila circular evita realocações

3.5.3 Funcionalidades Principais

Inserção (`queue_add`)

- Adiciona música no final da fila
- Complexidade: O(1)

Remoção (`queue_remove`)

- Remove música do início da fila
- Complexidade: O(1)

Listagem (`queue_get_musics`)

- Retorna array com todas as músicas na ordem da fila
- Complexidade: O(n)

3.5.4 Uso no Sistema

- Armazena recomendações geradas temporariamente
- Permite exibir recomendações na ordem gerada

- Liberada após exibição

3.6 TADs Auxiliares

3.6.1 TAD Person

Estrutura que representa uma pessoa no sistema:

- Campos: nome, telefone, email, playlist
- Funções: criação, acesso, modificação, comparação

3.6.2 TAD Music

Estrutura que representa uma música:

- Campos: nome, artista, comentário
- Funções: criação, acesso, modificação, comparação

3.6.3 TAD DataLoader

Responsável por carregar dados de arquivos CSV:

- Lê nomes de pessoas de `names.csv`
- Lê músicas e artistas de `songs.csv`
- Fornece funções para obter dados aleatórios

---

4. Algoritmos

4.1 Algoritmo de Cálculo de Similaridade

4.1.1 Descrição

O algoritmo `calculate_similarity` calcula quantas músicas duas pessoas compartilham:

```c
int calculate_similarity(Person *p1, Person *p2) {
    // Obtém playlists
    PlayList *pl1 = person_get_playlist(p1);
    PlayList *pl2 = person_get_playlist(p2);

    // Obtém arrays de músicas
    Music **songs1 = playlist_songs(pl1);
    Music **songs2 = playlist_songs(pl2);

    int count = 0;
    // Compara cada música de pl1 com cada música de pl2
    for (int i = 0; i < n1; i++) {
        for (int j = 0; j < n2; j++) {
            if (music_compare_order(songs1[i], songs2[j]) == MUSIC_EQUAL) {
                count++;
            }
        }
    }
    return count;
}
```

4.1.2 Complexidade

- **Tempo**: O($M_1 \times M_2$), onde M é o tamanho das playlists
- **Espaço**: O($M_1 + M_2$) para armazenar arrays temporários

4.2 Algoritmo de Geração de Recomendações

4.2.1 Descrição Detalhada

O algoritmo pertence à função `ui_generate_recommendations`. Ele gera recomendações musicais personalizadas para uma pessoa baseado nas playlists de pessoas similares.

Fase 1: Identificação de Candidatos

```
// Encontra todas as pessoas com similaridade > 0
for (int i = 0; i < n; i++) {
    if (i == person_idx) continue;
    int similarity = graph_get_edge_weight(graph, person_idx, i);
    if (similarity > 0) {
        candidates[candidates_count] = i;
        weights[candidates_count] = similarity;
        candidates_count++;
    }
}
```

Fase 2: Ordenação por Similaridade

```
// Ordena candidatos por similaridade (decrescente)
// Usa bubble sort simples
for (int i = 0; i < candidates_count - 1; i++) {
    for (int j = 0; j < candidates_count - i - 1; j++) {
        if (weights[j] < weights[j + 1]) {
            // Troca candidatos e pesos
        }
    }
}
```

Fase 3: Seleção dos Top 30%

```
// Seleciona os 30% mais similares
int top_percentage = 30;
int top_count = (candidates_count * top_percentage) / 100;
if (top_count < 1) top_count = 1;
```

Fase 4: Seleção Ponderada Aleatória

```
// Seleciona músicas usando seleção ponderada
// Pessoas mais similares têm maior probabilidade
int total_weight = 0;
for (int i = 0; i < top_count; i++) {
    total_weight += weights[i];
}

// Gera até 5 recomendações únicas
while (queue_get_elements(queue) < 5 && attempts < max_attempts) {
    // Seleção aleatória ponderada
    int random_val = rand() % total_weight;
    int cumulative = 0;
    for (int i = 0; i < top_count; i++) {
        cumulative += weights[i];
        if (random_val < cumulative) {
            selected_idx = candidates[i];
            break;
        }
    }

    // Seleciona música aleatória da playlist do candidato
    // Verifica duplicatas
```

```
    // Adiciona à fila de recomendações
}
```

4.2.2 Justificativa das Decisões

1. **Top 30%**: Foca nas pessoas mais similares, evitando recomendações de pessoas pouco relacionadas
2. **Seleção Ponderada**: Pessoas mais similares têm maior chance de contribuir com recomendações
3. **Verificação de Duplicatas**: Garante que músicas já na playlist do usuário não sejam recomendadas
4. **Limite de 5**: Quantidade razoável de recomendações para não sobrecarregar o usuário

4.2.3 Complexidade

- **Tempo**: O(C² + R × M), onde:
  - C = número de candidatos
  - R = número de recomendações geradas (máx 5)
  - M = tamanho médio das playlists

- **Espaço**: O(C + R)

4.3 Lógica de Atualização Dinâmica

4.3.1 Descrição

O sistema mantém sincronização entre todas as estruturas de dados quando pessoas ou músicas são adicionadas/removidas. Isso envolve:

1. **Atualizar a árvore AVL**: Remover pessoa com tamanho antigo, adicionar com tamanho novo
2. **Recalcular similaridades**: Atualizar todas as arestas do grafo relacionadas à pessoa
3. **Redimensionar o grafo**: Se necessário, expandir a matriz de adjacência
4. **Manter consistência**: Garantir que todas as estruturas estão sincronizadas

4.3.2 Adição de Pessoa

Quando uma nova pessoa é adicionada ao sistema:

```
void ui_add_person(...) {
    // 1. Cria pessoa e playlist vazia
    Person *person = person_create(name, tel, email);
    PlayList *playlist = playlist_create();
    person_set_playlist(person, playlist);

    // 2. Adiciona à PopList
    poplist_add(poplist, person);

    // 3. Adiciona à árvore AVL (tamanho 0 inicialmente)
    tree_add(tree, person);

    // 4. Atualiza o grafo (recalcula similaridades e redimensiona se necessário)
    graph_update(graph, poplist);
}
```

Processo de atualização do grafo durante adição:

1. Verifica se `n_pessoas > vertex_qnt`
2. Se sim, redimensiona usando crescimento exponencial (dobra o tamanho)
3. Recalcula todas as similaridades entre todos os pares de pessoas
4. Atualiza a matriz de adjacência

Vantagens do crescimento exponencial:

- Evita realocações frequentes quando múltiplas pessoas são adicionadas
- Se o grafo tem 100 vértices e precisa de 101, cresce para 200
- Próximas adições até 200 não requerem redimensionamento
- Reduz custo amortizado de O(n) realocações para O(log n)

4.3.3 Remoção de Pessoa

Quando uma pessoa é removida:

```
void ui_remove_person(...) {
    // 1. Obtém tamanho da playlist antes de remover
    PlayList *pl = person_get_playlist(person);
    int playlist_size = playlist_get_elements(pl);

    // 2. Remove da PopList
    poplist_remove_by_name(poplist, name);

    // 3. Remove da árvore AVL
    tree_remove(tree, playlist_size);

    // 4. Atualiza o grafo (recalcula similaridades)
    graph_update(graph, poplist);
}
```

Observação importante: O grafo não é reduzido quando pessoas são removidas. A matriz mantém seu tamanho atual, mas apenas as pessoas existentes são consideradas nos cálculos. Isso evita realocações desnecessárias e mantém espaço para futuras adições.

4.3.4 Adição/Remoção de Música

Quando uma música é adicionada ou removida de uma playlist:

```
void ui_add_music(...) {
    // 1. Obtém tamanho antigo
    int old_size = playlist_get_elements(playlist);

    // 2. Adiciona música
    playlist_add(playlist, music);

    // 3. Obtém novo tamanho
    int new_size = playlist_get_elements(playlist);

    // 4. Se tamanho mudou, atualiza estruturas
    if (old_size != new_size) {
        // Atualiza árvore AVL
        tree_remove(tree, old_size);
        tree_add(tree, person);

        // Recalcula todas as similaridades
        graph_update(graph, poplist);
    }
}
```

Por que recalcular todas as similaridades?

- Quando uma pessoa adiciona/remove músicas, suas similaridades com TODAS as outras pessoas podem mudar
- A implementação atual recalcula tudo para garantir consistência total

4.3.5 Complexidade

Operação	Complexidade	Justificativa
Adicionar pessoa	$O(n + \log n + V^2 \times M^2)$	Inserção na PopList + inserção na árvore + atualização do grafo
Remover pessoa	$O(n + \log n + V^2 \times M^2)$	Remoção da PopList + remoção da árvore + atualização do grafo
Adicionar música	$O(m + \log n + V^2 \times M^2)$	Inserção na playlist + atualização da árvore + atualização do grafo
Redimensionar grafo	$O(N^2)$	Onde N é o novo tamanho (crescimento exponencial reduz frequência)

Nota sobre redimensionamento: O crescimento exponencial garante que o número de redimensionamentos seja O(log n) para n adições

---

5. Interação com o Usuário

1. ✅ **list**: Lista todas as pessoas cadastradas
2. ✅ **recommend <nome>**: Gera recomendações para uma pessoa
3. ✅ **similar <nome>**: Mostra pessoas similares
4. ✅ **graph**: Visualiza matriz de adjacência
5. ✅ **tree**: Visualiza árvore AVL ordenada
6. ✅ **export**: Exporta grafo para arquivo CSV (graph.csv)
7. ✅ **add_person**: Adiciona nova pessoa
8. ✅ **remove_person**: Remove pessoa
9. ✅ **add_music**: Adiciona música à playlist
10. ✅ **remove_music**: Remove música da playlist
11. ✅ **playlist <nome>**: Mostra playlist de uma pessoa
12. ✅ **help**: Mostra ajuda
13. ✅ **off**: Encerra programa

---

6. Resumo da Complexidade Geral do Sistema

6.1 Operações Principais

Operação	Complexidade	Justificativa
Adicionar pessoa	$O(n + \log n + V^2 \times M^2)$	Inserção na PopList + inserção na árvore + atualização do grafo
Remover pessoa	$O(n + \log n + V^2 \times M^2)$	Remoção da PopList + remoção da árvore + atualização do grafo
Adicionar música	$O(m + \log n + V^2 \times M^2)$	Inserção na playlist + atualização da árvore + atualização do grafo
Redimensionar grafo	$O(N^2)$ amortizado	Onde N é o novo tamanho; crescimento exponencial reduz frequência
Gerar recomendações	$O(C^2 + R \times M)$	Ordenação de candidatos + seleção de músicas
Buscar similares	$O(V)$	Percorre todas as pessoas
Visualizar grafo	$O(V^2)$	Percorre matriz de adjacência
Exportar grafo	$O(V^2)$	Escreve matriz completa no arquivo CSV

Onde:

- n = número de pessoas
- m = tamanho da playlist
- V = número de vértices (pessoas)
- M = tamanho médio das playlists
- C = número de candidatos similares
- R = número de recomendações geradas

6.2 Espaço

- **Grafo**: $O(V^2)$ - matriz de adjacência
- **PopList**: O(n) - lista encadeada
- **Árvore AVL**: O(n) - árvore binária
- **PlayLists**: $O(n \times m)$ - uma playlist por pessoa
- **Total**: $O(V^2 + n \times m)$

7. Conclusão, Resumo e Observações Técnicas

Este projeto demonstra a aplicação prática de múltiplos TADs trabalhando em conjunto para resolver um problema real de recomendação musical. As decisões de implementação foram fundamentadas em:

1. **Eficiência**: Escolha de estruturas adequadas para cada operação
2. **Funcionalidade**: Sistema completo e interativo
3. **Consistência**: Sincronização entre todas as estruturas

7.1 Estruturas de Dados Utilizadas

- **Grafo**: Representação por matriz de adjacência
- **Lista Duplamente Encadeada**: PopList e PlayList
- **Árvore AVL**: Tree para organização por tamanho
- **Fila Circular**: Queue para recomendações temporárias

7.2 Algoritmos Implementados

- Cálculo de similaridade (comparação de playlists)
- Geração de recomendações (seleção ponderada)
- Balanceamento AVL (rotações simples e duplas)
- Atualização dinâmica de estruturas

7.3 Tratamento de Erros e Segurança

7.3.1 Validações Implementadas

- **Verificação de ponteiros nulos**: Todas as funções verificam ponteiros antes de uso
- **Validação de parâmetros**: Entradas são validadas antes de processamento
- **Tratamento de memória insuficiente**: Verificações de `malloc` e `realloc`
- **Mensagens de erro informativas**: Usuário recebe feedback claro sobre erros

7.3.2 Proteções contra Buffer Overflow

- **`string_split`**: Usa `strncpy` com limite para evitar overflow
- **Validação de tamanho de strings**: Strings são truncadas se excederem `MAX_CMD_LENGTH`
- **Verificação de limites de arrays**: Arrays fixos têm verificações de limites

7.3.3 Proteções de Memória

- **Verificação de `strdup` com NULL**: Evita comportamento indefinido
- **Validação antes de acesso a listas**: Verifica se `start` é NULL antes de percorrer
- **Limites em loops**: Loops têm verificações para evitar acesso fora dos limites
- **Divisão por zero**: Verificações antes de operações que podem causar divisão por zero

7.3.4 Redimensionamento Seguro

- **Verificação de overflow**: Crescimento exponencial verifica overflow de multiplicação
- **Liberação de memória**: Em caso de falha, toda memória alocada é liberada
- **Preservação de dados**: Durante redimensionamento, dados existentes são preservados

---