RELATÓRIO TÉCNICO: SISTEMA DE GERENCIAMENTO DE BIBLIOTECA - V2.0 (DINÂMICO)

1 INTRODUÇÃO

Este relatório técnico mostra a segunda etapa do projeto "Sistema de Gerenciamento de Biblioteca", que é uma evolução do que foi feito na Unidade 1. O principal objetivo agora foi deixar de usar vetores de tamanho fixo (estáticos) e refazer o projeto usando alocação dinâmica de memória, ponteiros e matrizes, conforme pedido nos requisitos da disciplina.

Além dessa mudança interna, o sistema agora tem novas funções, como emprestar e devolver livros, uma busca por texto e um relatório que mostra quais livros são mais populares, tudo funcionando através do menu principal.

2 METODOLOGIA

Para manter a consistência, usei as mesmas ferramentas da U1: o editor **Visual Studio Code (VS Code)** e o compilador **GCC**. O programa foi rodado no **WSL (Ubuntu)** no Windows 10.

A ideia foi evoluir o projeto da U1. O código foi bastante reorganizado, principalmente com a criação de duas novas funções: `inicializarBiblioteca()` e `liberarBiblioteca()`. Ter essas funções separadas ajuda muito na manutenção, pois toda a preparação da memória fica em um lugar e toda a limpeza fica em outro. Essa foi a principal estratégia para evitar vazamentos de memória.

3 ANÁLISE DO CÓDIGO

Esta seção explica como os novos conceitos da Unidade 2 foram usados na prática dentro do projeto.

3.1 Strings e Manipulação de Texto

Na U1, os títulos dos livros gastavam um espaço fixo, mesmo que o título fosse curto. Agora, as strings são usadas de um jeito bem mais inteligente. Cada título usa `malloc` para alocar *apenas* o espaço exato de que precisa. Para fazer isso, usei várias funções da biblioteca `string.h`:

- **`strlen()` e `strcpy()`:** A primeira descobre o tamanho do título que o usuário digitou, e a segunda copia esse título para o espaço de memória que acabei de alocar.
- **`strstr()`:** É a função principal da nova busca (`buscarLivros()`). Ela consegue encontrar um pedaço de texto dentro de outro, então o usuário pode digitar só "Harry" e o sistema encontra "Harry Potter".
- **`strcspn()`:** Uma função muito importante para corrigir um bug. O `fgets` (que usei para ler os títulos) deixa um caractere de "Enter" (`\n`) no final da string. Usei `strcspn` para encontrar e remover esse `\n`.

3.2 Estruturas de Repetição Aninhadas

Usei laços aninhados em duas partes principais do projeto, assim como foi praticado nas listas de exercícios:

1. **Para criar a matriz de dados:** Na função `inicializarBiblioteca()`, um laço `for` passa pelas "linhas" da matriz, e outro `for` (dentro dele) aloca a memória para as "colunas" de cada linha.
2. **Para mostrar o relatório de popularidade:** Na função `imprimirRelatorio()`, um laço `for` passa por cada livro. Dentro dele, um segundo laço `for` é executado, e ele imprime um asterisco `*` para cada empréstimo que o livro teve, criando um gráfico de barras simples.

3.3 Matrizes

Para guardar mais informações dos livros (e não só o título), criei uma matriz dinâmica chamada `dadosLivros`. Ela funciona em paralelo com a lista de títulos. Cada "linha" `i` dessa matriz corresponde ao livro `biblioteca[i]`:

- `dadosLivros[i][0]` (coluna 0): Guarda o **Status** do livro (0 se está disponível, 1 se está emprestado).
- `dadosLivros[i][1]` (coluna 1): Guarda a **Contagem de Empréstimos**, para sabermos quais livros são mais populares.

As novas funções `emprestarLivros()` e `devolverLivros()` funcionam

basicamente mudando os números dessa matriz.

3.4 Ponteiros e Alocação Dinâmica

Esses foram os conceitos principais de toda a U2. O sistema foi refeito usando "ponteiros para ponteiros" (`char **biblioteca` e `int **dadosLivros`). Isso trouxe duas vantagens claras:

1. **Economia de Memória:** Como falei, agora só alocamos o espaço de memória exato que cada título precisa.
2. **Flexibilidade:** O tamanho da biblioteca não é mais fixo no código (como era '5' na U1). Agora, a capacidade é definida em `CAPACIDADE_INICIAL` no `main.c` e usada na inicialização.

O uso de ponteiros também deixou a função `removerLivros()` muito mais rápida. Na U1, eu tinha que copiar a string inteira de um lugar para o outro com um laço `for`. Agora, a lógica é muito mais eficiente: eu só preciso mover o *ponteiro* (o "endereço") do último livro para o lugar do livro que foi apagado. É uma operação quase instantânea.

Para garantir que o programa não tenha vazamentos de memória, criei uma regra clara: todo `malloc` precisa ter um `free`.

1. Na função `removerLivros()`, eu uso `free()` no título que está sendo apagado *antes* de mover o outro ponteiro para o lugar dele.
2. No final do programa, quando o usuário digita '0' para sair, a função `liberarBiblioteca()` é chamada para limpar toda a memória que foi alocada, livro por livro, e também a memória da matriz e dos vetores principais.

4. DIFICULDADES E SOLUÇÕES

O maior desafio foi entender a ideia de "ponteiro para ponteiro" (`**`). Foi difícil imaginar como um "endereço" podia apontar para *outro* "endereço", que só então apontava para o dado (o título do livro).

A outra grande dificuldade foi o gerenciamento da memória. No começo, o programa tinha vazamentos de memória porque na função `removerLivros()` eu esquecia de dar `free()` no título antigo antes de mover o ponteiro novo por cima. A solução foi achar o erro e corrigir a ordem: primeiro liberar a memória, depois atualizar o ponteiro, como está no código final.

5. CONCLUSÃO

Mudar o projeto da U1 para a U2 foi um salto enorme. Deu para entender por que C é uma linguagem tão poderosa, mas também por que ela dá mais trabalho. Ter que controlar a memória manualmente foi o maior aprendizado.

O projeto funciona bem, mas ainda tem um problema de organização: eu estou usando dois vetores separados (`biblioteca` para os títulos e `dadosLivros` para o status/popularidade). Se eu esquecer de atualizar um deles, os dados ficam errados. A melhoria mais óbvia para o futuro é usar `structs`, criando uma `struct Livro` que junte o título, o status e a popularidade em uma coisa só. Isso deixaria o código bem mais limpo e seguro.