Instituto Politécnico de Viseu
Escola Superior de Tecnologia e Gestão de Viseu Departamento de Informática Unidade Curricular: Estruturas de dados

Projeto

Realizado por:

22587 – Miguel Rodrigues

22985 - Rafael Marques

22992 - Pedro Lopes

20888 - Francisco Marques

Instituto Politécnico de Viseu
Escola Superior de Tecnologia e Gestão de Viseu Departamento de Informática
Curso de Licenciatura em Engenharia Informática

Ano Letivo 2023/24

Viseu, 2024

Conteúdo

Introdução

Este projeto desenvolve um sistema de gestão de biblioteca em C, destinado a facilitar a administração de livros e requisições. O programa permite realizar diversas operações, como adicionar livros, listar livros, gerenciar requisições e requisitantes, além de oferecer funcionalidades para carregar e gravar dados de livros e requisições em ficheiros. Este relatório documenta a estrutura e as funcionalidades do programa, bem como as técnicas utilizadas para garantir sua correta operação.

Tópico I

1.1 Carregamento de Dados

O programa é capaz de carregar dados previamente guardados de livros, requisitantes e requisições. Utilizamos funções específicas para ler essas informações de ficheiros de texto e armazená-las nas estruturas adequadas do sistema.

1.2 Gravação de Dados

O programa permite gravar as informações atuais dos livros, requisitantes e requisições em ficheiros de texto. Essas operações podem ser solicitadas pelo utilizador ou ocorrer automaticamente antes do programa ser encerrado, garantindo que nenhuma informação seja perdida.

1.3 Gestão de Livros

O sistema oferece várias operações para a gestão dos livros na biblioteca. Essas operações permitem adicionar, listar e verificar a disponibilidade dos livros.

1.4 Gestão de Requisitantes

A gestão dos requisitantes é feita através de funções que permitem adicionar, listar e verificar informações dos requisitantes registrados na biblioteca.

1.5 Gestão de Requisições

As operações de gestão de requisições incluem a criação, listagem e devolução de livros requisitados pelos utilizadores.

1.6 Libertação de Memória

Para garantir que não haja perda de memória, o programa inclui funções que libertam a memória alocada dinamicamente para livros, requisitantes e requisições.

1.7 Paginação na Listagem de Requisitantes

Para facilitar a visualização dos requisitantes, o programa inclui uma funcionalidade de avanço página a página. Essa funcionalidade permite que o utilizador visualize os requisitantes em grupos, melhorando a navegação em listas extensas.

Tópico II

1. Carregamento de Dados

1.1 Carregar Livros: A função LoadLivros lê um ficheiro de texto que contém as informações dos livros e insere esses dados na biblioteca.

```c
void LoadLivros(BIBLIOTECA *B, const char *filename) {
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        fprintf(stderr, "Erro ao abrir o ficheiro %s\n", filename);
        return;
    }

    FILE *F_Logs = fopen(B->FICHEIRO_LOGS, "a");
    fprintf(F_Logs, "Carregando livros do ficheiro %s\n", filename);

    int id;
    char titulo[100];
    char autor[100];
    char area[50];
    int disponivel;

    while (fscanf(file, "%d\t%[^\t]\t%[^\t]\t%[^\t]\t%d\n", &id, titulo, autor, area, &disponivel) == 5) {
        LIVRO *livro = CriarLivro(id, titulo, autor, area);
        livro->disponivel = disponivel;
        AddLivroBiblioteca(B, livro);
        fprintf(F_Logs, "Livro lido: ID=%d, Título=%s, Autor=%s, Área=%s, Disponível=%d\n", id, titulo, autor, area, disponivel);
    }

    fclose(file);
    fclose(F_Logs);
}
```

1.2 Carregar Requisitantes: A função LoadRequisitantes carrega as informações dos requisitantes a partir de um ficheiro de texto e faz a validação dos dados.

```c
void LoadRequisitantes(BIBLIOTECA *biblioteca, const char *filename) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        printf("Erro ao abrir o ficheiro %s\n", filename);
        return;
    }

    char linha[256];
    while (fgets(linha, sizeof(linha), file)) {
        char id[10], nome[100], data_nasc[11], id_freguesia[7];
        if (sscanf(linha, "%9s\t%99[^\t]\t%10s\t%6s", id, nome, data_nasc, id_freguesia) != 4) {
            registrar_erro(linha, "Formato inválido");
            continue;
        }

        if (!validar_id_requisitante(id)) {
            registrar_erro(linha, "ID Requisitante inválido");
            continue;
        }

        if (!validar_data_nasc(data_nasc)) {
            registrar_erro(linha, "Data de nascimento inválida");
            continue;
        }

        if (!validar_id_freguesia(id_freguesia, biblioteca->HFreguesias)) {
            registrar_erro(linha, "ID Freguesia inválido");
            continue;
        }

        // Criar e adicionar requisitante à biblioteca
        PESSOA *pessoa = (PESSOA *)malloc(sizeof(PESSOA));
        pessoa->ID = atoi(id);
        pessoa->NOME = strdup(nome);
        pessoa->DATA_NASC = strdup(data_nasc);
        pessoa->CATEGORIA = strdup(id_freguesia);

        AddHashing(biblioteca->HRequisitantes, pessoa);
    }

    fclose(file);
}
```

1.3 Carregar Requisições: A função carregarRequisicoes lê as requisições de um ficheiro de texto e insere-as na lista de requisições do sistema.

```c
void LoadRequisicoes(BIBLIOTECA *B, const char *filename) {
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        fprintf(stderr, "Erro ao abrir o ficheiro %s\n", filename);
        return;
    }

    FILE *F_Logs = fopen(B->FICHEIRO_LOGS, "a");
    fprintf(F_Logs, "Carregando requisicoes do ficheiro %s\n", filename);

    int id, reqId, livroId;
    char bufferReq[20];
    char bufferDev[20];
    char area[50];

    while (fscanf(file, "%d\t%d\t%d\t%s\t%[^\t]\t%[^\n]", &id, &reqId, &livroId, area, bufferReq, bufferDev) == 6) {
        PESSOA *requisitante = PesquisarRequisitante(B, reqId);
        LIVRO *livro = PesquisarLivro(B, livroId, area);

        if (requisitante && livro) {
            REQUISICAO *requisicao = CriarRequisicao(id, requisitante, livro);

            requisicao->data_requisicao = ConverterStringParaTime(bufferReq);

            if (strcmp(bufferDev, "Não devolvido") != 0) {
                requisicao->data_devolucao = ConverterStringParaTime(bufferDev);
            } else {
                requisicao->data_devolucao = 0;
            }

            AddInicio(B->LRequisicoes, requisicao);
            fprintf(F_Logs, "Requisicao lida: ID=%d, ReqID=%d, LivroID=%d, Area=%s, DataReq=%s, DataDev=%s\n", id, reqId, livroId, ar
        } else {
            fprintf(F_Logs, "Erro: Requisitante ou livro não encontrado para a requisicao ID=%d\n", id);
        }
    }

    fclose(file);
    fclose(F_Logs);
}
```

2. Gravação de Dados

2.1 Gravar Livros: A função gravarLivros guarda os dados dos livros em um ficheiro de texto.

```c
void SalvarLivros(BIBLIOTECA *B, const char *filename) {
    FILE *file = fopen(filename, "w");
    if (file == NULL) {
        perror("Erro ao abrir o ficheiro para salvar livros");
        return;
    }

    NO_CHAVE *chave = B->HLivros->LChaves->Inicio;
    while (chave) {
        NO *no = chave->DADOS->Inicio;
        while (no) {
            LIVRO *livro = (LIVRO *)no->Info;
            fprintf(file, "%d\t%s\t%s\t%s\t%d\n", livro->ID, livro->NOME, livro->AUTOR, livro->AREA, livro->disponivel);
            no = no->Prox;
        }
        chave = chave->Prox;
    }

    fclose(file);
}
```

2.2 Gravar Requisitantes: A função SalvarRequisitantes grava as informações dos requisitantes em um arquivo de texto.

```c
void SalvarRequisitantes(BIBLIOTECA *B, const char *filename) {
    FILE *file = fopen(filename, "w");
    if (file == NULL) {
        perror("Erro ao abrir o ficheiro para salvar requisitantes");
        return;
    }

    NO_CHAVE *chave = B->HRequisitantes->LChaves->Inicio;
    while (chave) {
        NO *no = chave->DADOS->Inicio;
        while (no) {
            PESSOA *pessoa = no->Info;
            fprintf(file, "%d\t%s\t%s\t%s\n", pessoa->ID, pessoa->NOME, pessoa->DATA_NASC, pessoa->CATEGORIA);
            no = no->Prox;
        }
        chave = chave->Prox;
    }

    fclose(file);
}
```

2.3 Gravar Requisições: A função SalvarRequisicoes salva as requisições atuais em um arquivo de texto.

```c
void SalvarRequisicoes(BIBLIOTECA *B, const char *filename) {
    FILE *file = fopen(filename, "w");
    if (file == NULL) {
        perror("Erro ao abrir o ficheiro para salvar requisicoes");
        return;
    }

    NO *atual = B->LRequisicoes->Inicio;
    while (atual) {
        REQUISICAO *req = (REQUISICAO *)atual->Info;
        char bufferReq[20];
        char bufferDev[20];

        ConverterTimeParaString(req->data_requisicao, bufferReq, sizeof(bufferReq));
        if (req->data_devolucao != 0) {
            ConverterTimeParaString(req->data_devolucao, bufferDev, sizeof(bufferDev));
        } else {
            strcpy(bufferDev, "Não devolvido");
        }

        fprintf(file, "%d\t%d\t%d\t%s\t%s\t%s\n", req->ID, req->Ptr_Req->ID, req->Ptr_Livro->ID, req->Ptr_Livro->AREA, bufferReq, buf
        atual = atual->Prox;
    }

    fclose(file);
}
```

3. Gestão de Livros

3.1 Adicionar Livro: A função adicionarLivro permite inserir novos livros na biblioteca com todas as suas informações.

```c
int AddLivroBiblioteca(BIBLIOTECA *B, LIVRO *L)
{
    FILE *F_Logs = fopen(B->FICHEIRO_LOGS, "a");
    time_t now = time(NULL);
    fprintf(F_Logs, "Entrei em %s na data %s\n", __FUNCTION__, ctime(&now));

    if (!B || !L) return EXIT_FAILURE;

    // Verificar se já existe um livro com o mesmo ID na mesma área
    NO_CHAVE *chave = B->HLivros->LChaves->Inicio;
    while (chave != NULL) {
        if (strcmp(chave->KEY, L->AREA) == 0) {
            NO *atual = chave->DADOS->Inicio;
            while (atual != NULL) {
                LIVRO *livroExistente = (LIVRO *)atual->Info;
                if (livroExistente->ID == L->ID) {
                    // Livro com o mesmo ID já existe nesta área
                    fprintf(F_Logs, "Livro com ID %d já existe na área %s\n", L->ID, L->AREA);
                    fclose(F_Logs);
                    return EXIT_FAILURE;
                }
                atual = atual->Prox;
            }
        }
        chave = chave->Prox;
    }

    // Adicionar o livro se não existe um livro com o mesmo ID na mesma área
    NO_CHAVE *Key_colocar = FuncaoHashingLivros(B->HLivros, L);
    if (!Key_colocar)
    {
        Key_colocar = AddCHAVE(B->HLivros->LChaves, L->AREA);
    }
    AddInicio(Key_colocar->DADOS, L);

    fprintf(F_Logs, "Livro adicionado: ID=%d, Título=%s, Autor=%s, Área=%s\n", L->ID, L->NOME, L->AUTOR, L->AREA);

    fclose(F_Logs);
    return EXIT_SUCCESS;
}
```

3.2 Essas funções estão interligadas para mostrar informações detalhadas sobre a biblioteca, seus livros e suas áreas de interesse. ShowBiblioteca coordena a exibição geral, enquanto ShowHashingLivro organiza e imprime os livros por área usando uma estrutura de hashing. ShowListaLivro percorre e exibe todos os livros em uma lista específica, e MostrarLivro imprime os detalhes individuais de cada livro.

```c
void ShowBiblioteca(BIBLIOTECA *B)
{
    FILE *F_Logs = fopen(B->FICHEIRO_LOGS, "a");
    time_t now = time(NULL) ;
    fprintf(F_Logs, "Entrei em %s na data %s\n", __FUNCTION__, ctime(&now));
    printf("NOME BIBLIOTECA = [%s]\n", B->NOME);
    // Vosso Codigo.....
    ShowHashingLivro(B->HLivros);

    fclose(F_Logs);
}
```

```c
void ShowHashingLivro(HASHING *H)
{
    if (!H) return;
    if (!H->LChaves) return;
    NO_CHAVE *P = H->LChaves->Inicio;
    while (P)
    {
        printf("\n\n");
        printf("                    Área       [%s]                                    \n", P->KEY);
        //printf("\n");
        ShowListaLivro(P->DADOS);
        P = P->Prox;

    }
    printf("\n");
}
```

```c
//
void ShowListaLivro(LISTA *L)
{
    if (!L) return;
    //printf("NEL = %d\n", L->NEL);
    NO *P = L->Inicio;
    while (P != NULL)
    {

        MostrarLivro(P->Info);
        P = P->Prox;
    }
    printf("\n");
}
```

```c
void MostrarLivro(LIVRO *P)
{
    printf("\Livro: ID: %d [%s] [%s] [%s] [%s]\n",
           P->ID, P->NOME,P->AUTOR, P->AREA,P->disponivel? "Disponivel" : "Requisitado");
}
```

3.3 Verificar Livro por ISBN: A função PesquisarLivro realiza a pesquisa na estrutura de dados da biblioteca e retorna o ponteiro para o livro encontrado ou NULL caso o livro não seja encontrado.

```c
case 3: {
    char area[20];
    int id = LerInteiro("Indique o ID do livro:");
    printf("\nIndique a area do livro:");
    scanf(" %[^\n]", area);
    LIVRO *found = PesquisarLivro(Bib, id, area);
    if (found) {
        MostrarLivro(found);
    } else {
        printf("Livro nao encontrado\n");
    }
    break;
}
```

```c
LIVRO *PesquisarLivro(BIBLIOTECA *B, int id, const char *area) {
    //printf("%s\n", __FUNCTION__);
    NO_CHAVE *chave = B->HLivros->LChaves->Inicio;

    while (chave != NULL) {
        if (strcmp(chave->KEY, area) == 0) {
            NO *atual = chave->DADOS->Inicio;
            while (atual != NULL) {
                LIVRO *livro = (LIVRO *)atual->Info;
                if (livro->ID == id) {
                    return livro;
                }
                atual = atual->Prox;
            }
        }
        chave = chave->Prox;
    }
    printf("Livro com ID %d não encontrado na área %s\n", id, area);
    return NULL;
}
```

4. Gestão de Requisitantes

4.1 Adicionar Novo Requisitante.

Funções Utilizadas:

GerarIdRequisitante(BIBLIOTECA *Bib): Gera um ID único para um novo requisitante com base nos requisitantes já existentes na biblioteca.

validar_id_freguesia(const char *freguesia, HASHING *HFreguesias): Valida se a freguesia fornecida existe na estrutura de hashing HFreguesias.

validar_data_nasc(const char *data_nasc): Valida se a data de nascimento fornecida está no formato correto.

CriarPessoa(int id, const char *nome, const char *freguesia, const char *data_nasc): Cria e retorna um novo objeto PESSOA com os dados fornecidos.

AddRequisitante(BIBLIOTECA *B, PESSOA *req): Adiciona um requisitante (req) à biblioteca (B).

```
char nome[50], freguesia[50], data_nasc[11];

int id;
limparEcra();

// Gerar automaticamente um ID único para o requisitante
id = GerarIdRequisitante(Bib);

printf("ID do requisitante gerado automaticamente: %d\n", id);

printf("Nome do requisitante: ");
scanf(" %[^\n]", nome);

printf("Freguesia do requisitante: ");
scanf(" %[^\n]", freguesia);

if (!validar_id_freguesia(freguesia, Bib->HFreguesias)) {
    printf("ID Freguesia inválido\n");
    sleep(2);
    break;
}
printf("Data de nascimento (dd-mm-aaaa): ");
scanf(" %[^\n]", data_nasc);
if (!validar_data_nasc(data_nasc)) {
    printf("Data de nascimento inválida\n");
    break;
}

PESSOA *newReq = CriarPessoa(id, nome, freguesia, data_nasc);

if (AddRequisitante(Bib, newReq) == EXIT_SUCCESS) {
    printf("Requisitante adicionado com sucesso\n");
    sleep(2); // Pausa por 2 segundos antes de continuar
} else {
    printf("Erro ao adicionar requisitante\n");
    sleep(2); // Pausa por 2 segundos antes de continuar
}
```

4.2 Listar Requisitantes: A função listarRequisitantesPorPagina mostra um número de requisitantes registrados na biblioteca de forma organizada separados por páginas.

4.3 Pesquisar Requisitante por Nome: Neste trecho de código, há um caso específico dentro de um switch-case que permite ao utilizador pesquisar requisitantes por ID ou por nome na biblioteca, fornecendo opções para visualizar os livros requisitados por requisitantes encontrados.

```
case 3: {
    int op;
    printf("1 - Pesquisar ID || 0 - Pesquisar Nome\n");
    scanf("%d", &op);
    if (op == 1) {
        int id = LerInteiro("ID do requisitante: ");
        PESSOA *found = PesquisarRequisitante(Bib, id);
        if (found) {
            MostrarPessoa(found);
            printf("1 - Ver requisicoes || 2 - Sair\n");
            scanf("%d", &op);
            if (op == 1) {
                MostrarLivrosRequisitadosPorPessoa(Bib, found);
            }
        } else {
            printf("Requisitante nao encontrado\n");
        }
    } else {
        char nome[50];
        printf("Insira o nome:");
        scanf(" %[^\n]", nome);

        PESSOA *foundName = PesquisarRequisitanteNome(Bib, nome);
        if (foundName) {
            MostrarPessoa(foundName);
            printf("1 - Ver requisicoes || 2 - Sair\n");
            scanf("%d", &op);
            if (op == 1) {
                MostrarLivrosRequisitadosPorPessoa(Bib, foundName);
            }
        } else {
            printf("Requisitante não encontrado\n");
        }
    }
    break;
}
```

5. Gestão de Requisições

5.1 Efetuar Requisição: A função CiarRequisicao permite que um livro seja requisitado por um requisitante, registrando a requisição e atualizando o estado do livro de disponível para requisitado.

```c
int CriarRequisicaoBiblioteca(BIBLIOTECA *B, int idReq, int idLivro, const char *area)
{
    PESSOA *req = PesquisarRequisitante(B, idReq);
    LIVRO *livro = PesquisarLivroPorArea(B, idLivro, area);

    if (req == NULL)
    {
        printf("Requisitante não encontrado\n");
        return 0;
    }
    if (livro == NULL)
    {
        printf("Livro não encontrado\n");
        return 0;
    }
    if (livro->disponivel == 0)
    {
        printf("Livro já está requisitado\n");
        return 0;
    }

    REQUISICAO *newReq = CriarRequisicao(rand(), req, livro);
    AdicionarRequisicao(B, newReq);
    livro->disponivel = 0; // Define o livro como indisponível

    return 1;
}
```

5.2 Devolver Livro: A função DevolverLivro possibilita a devolução de um livro requisitado, alterando seu estado de requisitado para disponível.

```c
int RegistarDevolucao(BIBLIOTECA *B, int idReq, int idLivro, const char *area)
{
    NO *atual = B->LRequisicoes->Inicio;
    int encontrouRequisicao = 0;

    while (atual != NULL) {
        REQUISICAO *requisicao = (REQUISICAO *)atual->Info;
        if (requisicao->Ptr_Req->ID == idReq && requisicao->Ptr_Livro->ID == idLivro && strcmp(requisicao->Ptr_Livro->AREA, area) == 0) {
            encontrouRequisicao = 1;
            if (requisicao->data_devolucao == 0) {
                time(&requisicao->data_devolucao);
                requisicao->Ptr_Livro->disponivel = 1; // o livro volta a estar disponivel
                printf("Devolução registrada com sucesso.\n");
                return 1;
            }
        }
        atual = atual->Prox;
    }

    if (encontrouRequisicao) {
        printf("O livro já foi devolvido anteriormente.\n");
    } else {
        printf("Requisição não encontrada.\n");
    }
    return 0;
}
```

5.3 ListarRequisiçoes: A função ListarRequisiçoes mostra todos os livros que já foram requisitados, junto com informações sobre o requisitante, a data de requisição, se já foi devolvido (se sim retorna a data de devolução) e informa se o livro está disponível ou requisitado por alguém.

```
void ListarRequisicoes(BIBLIOTECA *B) {
    NO *atual = B->LRequisicoes->Inicio;
    while (atual != NULL) {
        REQUISICAO *requisicao = (REQUISICAO *)atual->Info;
        char *areaAtual = requisicao->Ptr_Livro->AREA;
        int encontrada = 0;

        // Verificar se a área já foi listada
        NO *tmp = B->LRequisicoes->Inicio;
        while (tmp != atual) {
            REQUISICAO *reqTmp = (REQUISICAO *)tmp->Info;
            if (strcmp(reqTmp->Ptr_Livro->AREA, areaAtual) == 0) {
                encontrada = 1;
                break;
            }
            tmp = tmp->Prox;
        }

        // Se a área não foi listada, listar as requisições dessa área
        if (!encontrada) {
            //printf("||-------------------------||-------------------------||\n");
            printf("||--------Área--------||--------%s--------||-------------------------||\n", areaAtual);
            NO *innerAtual = B->LRequisicoes->Inicio;
            while (innerAtual != NULL) {
                REQUISICAO *innerRequisicao = (REQUISICAO *)innerAtual->Info;
                if (strcmp(innerRequisicao->Ptr_Livro->AREA, areaAtual) == 0) {
                    MostrarRequisicao(innerRequisicao);
                }
                innerAtual = innerAtual->Prox;
            }
        }
        atual = atual->Prox;
    }
}
```

6. Liberação de Memória

6.1 Destruir Biblioteca: A função DestruirBiblioteca liberta a memória alocada para os livros, requisitantes e as requisições.

```c
void DestruirBiblioteca(BIBLIOTECA *B) {
    FILE *F_Logs = fopen(B->FICHEIRO_LOGS, "a");
    if (F_Logs == NULL) {
        perror("Erro ao abrir o arquivo de logs");
        return;
    }

    time_t now = time(NULL);
    fprintf(F_Logs, "Entrei em %s na data %s\n", __FUNCTION__, ctime(&now));

    // Liberar memória para nome e ficheiro de logs
    if (B->NOME) free(B->NOME);
    if (B->FICHEIRO_LOGS) free(B->FICHEIRO_LOGS);

    // Destruir hashing de livros
    DestruirHashing(B->HLivros, DestruirLivro);
    // Destruir hashing de requisitantes
    DestruirHashing(B->HRequisitantes, DestruirPessoa);
    // Destruir hashing de distritos
    DestruirHashing(B->HDistritos, DestruirDistrito);
    // Destruir hashing de concelhos
    DestruirHashing(B->HConcelhos, DestruirConcelho);
    // Destruir hashing de freguesias
    DestruirHashing(B->HFreguesias, DestruirFreguesia);
    // Destruir todas as requisições
    NO *current = B->LRequisicoes->Inicio;
    while (current != NULL) {
        NO *next = current->Prox;
        DestruirRequisicao((REQUISICAO *)current->Info);
        free(current);
        current = next;
    }
    // Destruir a lista de requisições em si
    free(B->LRequisicoes);
    // Liberar a estrutura da biblioteca
    free(B);
    printf("Biblioteca destruída com sucesso\n");
    fprintf(F_Logs, "Biblioteca destruída com sucesso em %s\n", ctime(&now));
    fclose(F_Logs);
}
```

7. Outras Funcionalidades

7.1 AreaMaisComumBiblioteca: Determina qual área da biblioteca possui o maior número de livros e exibe essa informação.

```
void AreaMaisComumBiblioteca(BIBLIOTECA *B) {
    FILE *F_Logs = fopen(B->FICHEIRO_LOGS, "a");
    time_t now = time(NULL);
    fprintf(F_Logs, "Entrei em %s na data %s\n", __FUNCTION__, ctime(&now));

    NO_CHAVE *P = B->HLivros->LChaves->Inicio;
    NO_CHAVE *areaMaisComum = NULL;
    int maxLivros = 0;

    // Contar o número de livros em cada área
    while (P) {
        int contagemLivros = 0;
        NO *livroAtual = P->DADOS->Inicio;
        while (livroAtual) {
            contagemLivros++;
            livroAtual = livroAtual->Prox;
        }

        // Verificar se esta área tem mais livros do que a atual máxima
        if (contagemLivros > maxLivros) {
            maxLivros = contagemLivros;
            areaMaisComum = P;
        }
        P = P->Prox;
    }

    // Verificar outras áreas que têm o mesmo número de livros que o máximo
    P = B->HLivros->LChaves->Inicio;
    fprintf(F_Logs, "Áreas com o maior número de livros (%d):\n", maxLivros);
    printf("Áreas com o maior número de livros (%d):\n", maxLivros);
    while (P) {
        int contagemLivros = 0;
        NO *livroAtual = P->DADOS->Inicio;
        while (livroAtual) {
            contagemLivros++;
            livroAtual = livroAtual->Prox;
        }

        // Se esta área tem o mesmo número de livros que o máximo, registrar
        if (contagemLivros == maxLivros) {
            fprintf(F_Logs, "Área: %s, Número de livros: %d\n", P->KEY, contagemLivros);
            printf("Área: [%s], Número de livros: [%d]\n", P->KEY, contagemLivros);
        }
        P = P->Prox;
    }

    fclose(F_Logs);
```

7.2 Outros: O MenuOutros oferece 3 opções, guardar os dados da biblioteca em ficheiros com formato .txt e .csv, calcular e apresentar a quantidade total de memória ocupado por todas as estruturas de dados da biblioteca e também permite ao utilizador gravar informações da biblioteca num ficheiro XML.

```
case 4:
    switch (MenuOutros()) {
        case 1:
            SalvarBiblioteca(Bib);
            break;
        case 2: {
            size_t memoriaTotal = CalcularMemoriaBiblioteca(Bib);
            printf("Memória ocupada por toda a estrutura de dados: %zu bytes\n", memoriaTotal);
            break;
        }
        case 3: {
            char nomeArquivo[256];
            printf("Digite o nome do arquivo XML para salvar as informações: ");
            scanf("%255s", nomeArquivo);
            GravarInformacaoXML(Bib, nomeArquivo);
            printf("Informações da biblioteca gravadas em arquivo XML: %s\n", nomeArquivo);
            break;
        }
        default:
            printf("Opcao nao implementada\n");
            break;
    }
    break;
default:
```

Conclusão

O projeto envolve o desenvolvimento de um programa em linguagem C para otimizar a gestão de uma biblioteca. O sistema facilita a administração de livros, requisitantes e requisições, garantindo eficiência e organização.

Com as funcionalidades implementadas, o programa possui a capacidade de carregar e gravar dados de livros, requisitantes e requisições, permitindo uma administração contínua e segura da biblioteca. Além disso, o sistema oferece operações detalhadas para a gestão de livros, incluindo a adição, listagem e verificação de disponibilidade.

A gestão de requisitantes é igualmente eficiente, permitindo adicionar, listar e procurar requisitantes pelo nome. A gestão de requisições é abrangente, possibilitando efetuar, listar e devolver livros requisitados, garantindo um controle preciso sobre o estado dos livros na biblioteca.

A listagem paginada de requisitantes melhora a navegação em listas extensas.

O programa oferece uma solução robusta para a gestão de bibliotecas, preservando a integridade dos dados e auxiliando na administração diária e na tomada de decisões estratégicas.