Instituto Politécnico de Viseu

Escola Superior de Tecnologia e Gestão de Viseu

Relatório de Estruturas de Dados

Projeto Prático

Realizado por:

Luís Ferro nº16790

Rafael Trabulo nº16823

Docentes: Francisco Morgado, Carlos Simões, Jorge Loureiro, Luís Soares e Ivan Pires

Viseu, 2020

Instituto Politécnico de Viseu

Escola Superior de Tecnologia e Gestão de Viseu

Licenciatura em Engenharia Informática

Estruturas de Dados

Projeto Prático

Ano Letivo 2019/2020

1º Ano, 2º Semestre

Realizado por:

Luís Ferro nº16790

Rafael Trabulo nº16823

Docentes: Francisco Morgado, Carlos Simões, Jorge Loureiro, Luís Soares e Ivan Pires

Viseu, 2020

Índice

Índice de Figuras

1. Introdução

No âmbito da unidade curricular de Estruturas de Dados, foi-nos proposto realizar um projeto prático com base numa biblioteca de uma escola superior, tendo como intuito a sua gestão.

A gestão desta biblioteca implica a gestão de livros, requisitantes, requisições, distritos, concelhos e distritos. Para os livros, foi utilizada uma estrutura de dados Hashing, onde a chave dessa estrutura são as áreas dos livros pertences à biblioteca, seguindo a exemplificação demonstrada no decorrer das aulas teóricas e práticas. No caso dos requisitantes, requisições, distritos, concelhos e distritos optámos por utilizar uma lista ligada genérica (recorrendo ao ponteiro para void).

A estrutura do nosso projeto consiste numa classe Hashing, onde a estrutura de dados Hashing e todas as funcionalidades relativas ao Hashing são definidas; uma classe Lista, onde a estrutura de dados da lista ligada e todas as funcionalidades relativas à lista são definidas; uma classe para Distritos, Concelhos, Freguesias, Livros, Requisições e Requisitantes, onde a estrutura de dados e funcionalidades relativas a cada um destes é definida; e por fim, uma classe Texto, onde todas as funcionalidades relativas à gestão dos ficheiros utilizados pelo programa são definidas.

2. Desenvolvimento

Além das funcionalidades a serem mencionadas neste capítulo relativas ao Hashing, Lista, Livros, Requisitantes, Requisições, Distritos, Concelhos e Distritos, temos também um sistema de gestão de sessões que oferece ao utilizador a opção de continuar a sessão anterior ou abrir uma nova sessão, onde a informação é a que foi dada por defeito pelos professores. Além disso, é guardado o progresso quando o utilizador sai do programa, ao carregar no (0) – Sair ou selecionando a opção existente no menu especifica para guardar o seu progresso até aquele momento.

2.1. Hashing

A estrutura Hashing foi utilizada na gestão dos livros, sendo a área dos livros a chave, que permite o mapeamento dessa chave com os livros pertencentes a essa área. Isto oferece a vantagem de aceder aos livros de uma determinada área de forma mais eficiente do que se fosse aplicada uma lista ligada.

```
#ifndef HASHING_H_INCLUDED
#define HASHING_H_INCLUDED

#include "Lista.h"
#include "Livro.h"

//---------------------------------

typedef struct no_has
{
    char CHAVE[20];
    struct no_has* Prox_Chave;
    LISTA* LLivros;
}NO_HAS;

typedef struct
{
    NO_HAS* Inicio;
    int NUM_CHAVES;
}Hashing;

#endif // HASHING_H_INCLUDED
```

Figura 1 - Estrutura Hashing

Em termos de estruturação e funções básicas para o funcionamento desta estrutura de Hashing, seguimos os exemplos e conceitos dados nas aulas teóricas e práticas. Além dessas funções, foram adicionadas outras funcionalidades que fazem uso do Hashing.

Temos a função "`MostrarHASHINGLivrosRequisitadosPorRequisitante`" que, como o nome indica, visa mostrar todos os livros requisitados por um dado requisitante. No programa principal, o utilizador passa o nome do requisitante que deseja verificar as requisições e, usando uma função auxiliar "`GetID`", obtemos o seu ID através do seu nome. Esse ID é por sua vez passado para a função "`MostrarHASHINGLivrosRequisitadosPorRequisitante`" juntamente com o Hashing e a Lista.

```
void MostrarHASHINGLivrosRequisitadosPorRequisitante(Hashing* H, LISTA* L, int parametro)
{
    if (!H && !L) return;

    NO* N = L->Inicio;
    NO_HAS* P = H->Inicio;

    int* array = (int*)malloc(L->NEL * sizeof(int)), i = 0, k = 0;
    //Inicialização do array
    for (int k = 0; k < L->NEL; k++) {
        array[k] = 0;
    }

    while (N)   //Percorrer lista de requisições
    {
        array[i++] = GetIDRequisitanteEmRequisicoes(N->INFO, parametro);    //Encher array com IDs dos livros relativos a um dado requisitante
        N = N->PROX;
    }

    while (P)
    {
        printf("[%s]\n", P->CHAVE);
        for (int k = 0; k < L->NEL; k++) {
            ProcurarLivroMaisRequisitadoRecente(P->LLivros, MostrarLivrosRequisitadosPorRequisitante, array[k]);    //Mostra livros requisitados por um dado requisitante
        }
        P = P->Prox_Chave;
    }

    free(array);
}
```

Figura 2 - MostrarHASHINGLivrosRequisitadosPorRequisitante

Nesta função, utilizamos um array para auxiliar a listagem dos livros por requisitante. Este array receberá todos os IDs dos livros associados ao requisitante, previamente introduzido pelo utilizador, da lista ligada Requisições. Com os IDs dos livros associados ao requisitante, basta percorrer o Hashing e listar os livros requisitados.

A função "`AreaComMaisLivros`" visa mostrar ao utilizador qual a área com o maior número de livros. Para tal, são usadas algumas variáveis auxiliares: tmp, que receberá o número de elementos associados a uma determinada chave do Hashing; quantidade, que receberá o valor mais alto e chave, que receberá a chave corresponde ao número de elementos mais alto.

A função "`PesquisarLivros`" é muito semelhante ao "`MostrarHASHING`", porém inclui a passagem de um parâmetro, que funcionará como um filtro.

A função "`Requisitar`" tem como único intuito alterar o estado do livro, caso não esteja requisitado, e retornar 0 ou 1, de forma a verificar se um determinado livro já se encontra requisitado ou não. No caso da função "`RequisitarIDLivro`", esta recebe o título do livro e retorna o seu ID. Já na função "`Devolver`", esta altera o estado do livro, caso este esteja requisitado.

Para listar os livros requisitados, foi utilizada a função "`MostrarLivrosRequisitados`" que segue a lógica da função "`MostrarHASHING`", porém apenas lista os livros com estado "REQUISITADO".

No caso da função "`MostrarLivrosMaisRecentes`", a lógica difere um pouco. Aqui, utilizamos dois nós Hashing. Um deles é percorrido com o intuito de recolher o ano mais recente retornado pela função "`GetListaLivrosMaisRecentes`". O ano mais recente é guardado na variável mais_recente e posteriormente utilizado quando percorremos o outro nó Hashing onde são listados os livros pertences ao ano contido na variável mais_recente.

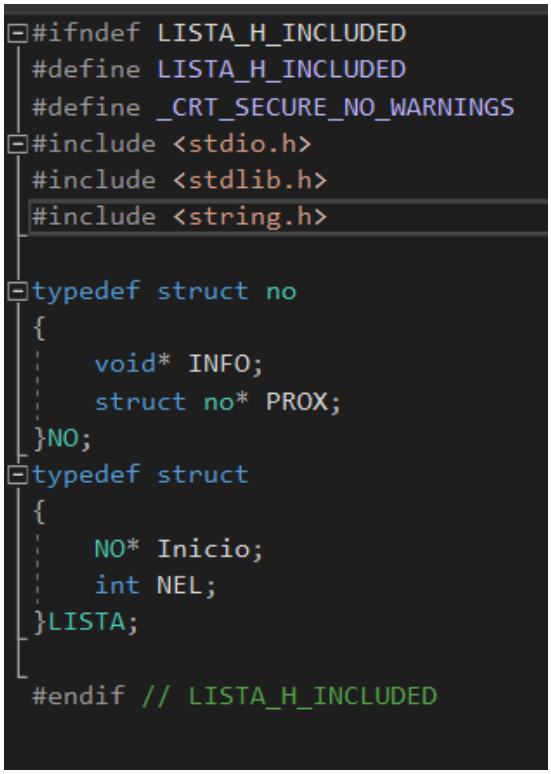
O funcionamento da função "`PesquisarLivroMaisRequisitado`" passa por receber o valor representante do número de vezes que um livro foi requisitado. O valor mais alto fica atribuido à variável maior_req_todos e posteriormente passado na função "`ProcurarLivroMaisRequisitadoRecente`" que tratará de listar o livro com o maior número de requisições.

As funções "`PesquisarAreaMaisRequisitada`" e "`PesquisarAreaMenosRequisitada`" funcionam de forma bastante similar, na medida em que, na primeira função a variável maior_req_todos é inicializada a 0 e posteriormente recebe o maior número de requisições de uma área enquanto que na segunda função, a variável maior_req_todos recebe imediatamente um número de requisições de uma área. Esta variável também passa a receber o menor número de requisições de uma área. Vale notar que a segunda função, "`PesquisarAreaMenosRequisitada`" é um extra que não estava incluso no enunciado.

Para finalizar a secção de Hashing, falta mencionar a função "`MemoriaHASHING`". Esta função trata de percorrer o Hashing somando à variável Mem o sizeof do ponteiro para o nó Hashing, sizeof do ponteiro para a chave do Hashing e sizeof do ponteiro para a nossa lista livros.

2.2. Lista

Para a gestão de requisitantes, requisições, distritos, concelhos e distritos optámos pelo uso de uma lista ligada genérica, utilizando ponteiro para void como mencionado no decorrer das aulas teóricas e práticas.



```
#ifndef LISTA_H_INCLUDED
#define LISTA_H_INCLUDED
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct no
{
    void* INFO;
    struct no* PROX;
}NO;
typedef struct
{
    NO* Inicio;
    int NEL;
}LISTA;

#endif // LISTA_H_INCLUDED
```

Figura 3 - Estrutura Lista Ligada Genérica

Em termos de estruturação e funções básicas para o funcionamento da lista ligada genérica, seguimos os exemplos e conceitos dados nas aulas teóricas e práticas. Além dessas funções, foram adicionadas outras funcionalidades que fazem uso da Lista.

A função "`GuardarLISTA`" é utilizada pelo sistema de sessões do nosso programa, bem como pela funcionalidade de Gravação dos dados no formato XML. É uma função simples que percorre uma dada lista e executa uma dada função.

Relativamente às funcionalidades ligadas às idades dos requisitantes, presentes no ficheiro Lista.cpp, temos a função "`MostrarIdadeMaxima`" que percorre uma dada lista, neste caso Requisitantes, e guarda a idade numa variável tmp. A idade mais alta entre os requisitantes é guardada numa variável idade_max e posteriormente mostrada ao utilizador.

Além desta, temos também a função "`MostrarIdadeMedia`" que percorre também uma dada lista, neste caso Requisitantes, e vai somando as idades recebidas na variável tmp. É verificado se o próximo elemento do ciclo While é nulo. Em caso afirmativo, a variável idade_med recebe o resultado da divisão do valor final de tmp pelo número de elementos existentes na lista Requisitantes e posteriormente mostrado ao utilizador.

Ainda ligado às idades dos utilizadores, temos a função "`MostrarIdadeSuperiorX`". Esta percorre também uma dada lista, no caso Requisitantes, e vai recebendo as idades numa variável tmp. Se o valor presente em tmp for superior ao parâmetro de entrada inserido pelo utilizador, é incrementada uma variável contagem. Uma vez terminado o ciclo, é mostrado ao utilizador quantos utilizadores têm idade superior ao parâmetro de entrada.

Para finalizar no que toca a idades na Lista.cpp, temos a função "`MostrarIdadeMaisComum`". Esta é um pouco mais complexa que as outras.

```
void MostrarIdadeMaisComum(LISTA* L, int (*func)(void*))
{
    if (!L) return;
    //Criação de dois arrays. Um vai conter a informação pertinente (neste caso, idades) e outro será auxiliar para permitir contar a frequência com que x idade aparece
    int *array = (int*)malloc(L->NEL * sizeof(int)), *array_freq = (int*)malloc(L->NEL * sizeof(int)), contagem = 0, idade_freq = 0, ocorrencias = 0;

    //Inicialização dos arrays. Os elementos do array_freq são inicializados a -1, uma vez que o 0 será uma flag utilizada mais para a frente
    for (int k = 0; k < L->NEL; k++) {
        array[k] = 0;
        array_freq[k] = -1;
    }

    int i = 0;
    NO* P = L->Inicio;

    while (P)
    {
        array[i++] = (*func)(P->INFO);  //Enche o array com as idades dos requisitantes
        P = P->PROX;
    }

    //Ciclo for que percorre os elementos todos dos requisitantes
    for (int k = 0; k < L->NEL; k++) {
        contagem = 1;
        //Ciclo for que percorre os elementos todos dos requisitantes, estando uma casa na frente em relação ao ciclo anterior
        for (int j = k + 1; j < L->NEL; j++) {
            //Se forem iguais
            if (array[k] == array[j]) {
                contagem++; //Incrementa a contagem
                array_freq[j] = 0;  //e é acionada a flag como foi encontrado. No caso, é metido a 0
            }
        }

        if (array_freq[k] != 0) {
            array_freq[k] = contagem;   //array_freq[k] fica com valor presente na contagem
            //Compara o valor do array_freq[k] com o número de ocorrencias previamente inicializado a 0
            if (array_freq[k] > ocorrencias) {
                ocorrencias = array_freq[k];    //Se o valor do array_freq[k] for superior, a variavel ocorrencias fica com esse valor
                idade_freq = array[k];  //e idade_freq, previamente inicializada a 0, fica com o valor do array[k]
            }
        }
    }

    printf("%d anos é a idade com mais requisitantes, totalizando %d requisitantes\n", idade_freq, ocorrencias);

    free(array);
    free(array_freq);
}
```

Figura 4 - MostrarIdadeMaisComum

A ideia nesta função passa pelo uso de dois arrays e algumas variáveis auxiliares. Uma dada lista é percorrida, enchendo um dos arrays com as idades dos utilizadores. Em seguida, são utilizados dois ciclos For. O primeiro trata o número de elementos da lista e o segundo trata de fazer o mesmo, porém está um "passo" na frente do primeiro ciclo For.

No segundo ciclo For, é verificado se o elemento do primeiro ciclo For é igual ao elemento do segundo ciclo For. Em caso afirmativo, é incrementada a variável contagem e acionada uma flag no segundo array, sendo metido o elemento do segundo ciclo For a 0.

No primeiro ciclo For, é verificado se aquele elemento é diferente de 0. Em caso afirmativo, esse elemento recebe o valor existente na variável contagem. Posteriormente é comparado o valor do segundo array com o número de ocorrências. Se o valor do segundo array for superior, a variável ocorrências fica com esse valor e a variável idade_freq fica com o valor do primeiro array. Em seguida, é mostrado ao utilizador a idade com o maior número de ocorrências.

A função "`MostrarSobrenomeMaisComum`" segue uma lógica extremamente semelhante à função anterior, sendo a única diferença entre estas funções o tipo de variável com que trabalham.

A função "`ProcurarDistrito`" tem como objetivo retornar o ID de um distrito dado o seu nome.

A função "`ProcurarPessoasDistrito`" visa devolver o número de pessoas pertencentes a um dado um distrito e com um dado apelido.

A função "`ProcurarLISTA`" é extremamente semelhante à função "`MostrarLISTA`". A única diferença entre estas duas é que a primeira função permite a passagem de um parâmetro do tipo CHAR por parte de um utilizador, sendo assim útil para filtragem.

A função "`ProcurarRequisitar`" tem como único intuito retornar 0 ou 1, permitindo verificar se um determinado livro já está requisitado ou não.

A função "`ProcurarLivroMaisRequisitadoRecente`" é extremamente semelhante à função "`ProcurarLISTA`" porém a primeira função permite a passagem de um parâmetro do tipo INT.

As funções "`GetRequisicoesArea`", "`GetListaLivrosMaisRecentes`", "`GetRequisicoesLivroMaisRequisitado`", "`GetID`" e "`GetIDRequisitantesComIDLivro`" seguem todas uma lógica já explicada neste relatório, onde temos uma variável tmp que recebe um determinado valor, individualmente. O valor mais alto é guardado numa variável X e posteriormente retornado.

A função "SortLISTA" tem também alguma complexidade envolvida. Esta é utilizada para ordenar os requisitantes por nome, apelido e ID freguesia.

```
void SortLISTA(LISTA* L, char* (*func1)(void*), void (*func2)(void*, const char*))
{
    if (!L) return;

    char** array = (char**)malloc(L->NEL * sizeof(char*)), * tmp;
    int i = 0, j = 0, k = 0, l = 0, size = L->NEL;

    NO* P = L->Inicio;

    while (P)
    {
        array[i++] = (*func1)(P->INFO); //O array recebe a informação pela qual desejo organizar a lista como apelido, nome e id_freguesia
        P = P->PROX;
    }

    //Ordena o array
    for (j = 0; j < size - 1; ++j)
    {
        for (k = j + 1; k < size; ++k)
        {
            if (strcmp(array[j], array[k]) > 0)
            {
                //Procede a fazer a troca dos elementos
                tmp = array[j];
                array[j] = array[k];
                array[k] = tmp;
            }
        }
    }
    //Remove elementos duplicados do array
    for (j = 0; j < size; j++)
    {
        for (k = j + 1; k < size; k++)
        {
            //Verifica se são iguais
            if (strcmp(array[j], array[k]) == 0)
            {
                for (l = k; l < size; l++)
                {
                    //Atribui o valor de l+1 a l
                    array[l] = array[l + 1];
                }
                size--; //Decrementa o tamanho, uma vez que foi removido um duplicado
                k--;
            }
        }
        //Procura na lista os elementos do array já organizado e sem duplicados
        ProcurarLISTA(L, (*func2), array[j]);
    }

    free(array);
```

Figura 5 - SortLISTA

Para o seu funcionamento, temos um array. É percorrida uma dada lista, neste caso Requisitantes, e trata de encher o array com a informação pela qual o utilizador deseja organizar a lista, apelido nome ou id_freguesia.

Posteriormente é feita a ordenação do array. Para tal temos dois ciclos For. O primeiro percorre os elementos todos do array e o segundo também percorre os elementos todos do array, porém está um "passo" na frente. Se o elemento do primeiro ciclo For for maior que o elemento do segundo ciclo For, é feita uma troca.

Uma vez ordenado, procedemos para a remoção de elementos duplicados do array. A lógica é semelhante ao que foi feito anteriormente, porém agora queremos procurar por uma

igualdade. Encontrada uma igualdade, percorremos a partir do elemento encontrado até ao final atribuindo um valor do próximo elemento ao atual. Posteriormente, decrementamos o tamanho do array uma vez que também removemos um duplicado.

Finalmente, usamos a função "`ProcurarLISTA`" para mostrar a lista organizada por uma das três opções.

Para terminar esta secção, falta mencionar a função "`MemoriaLista`". Esta função trata de percorrer a Lista somando à variável Mem o sizeof do ponteiro para o nó Lista e sizeof da Info.

2.3. Texto

As funções relativas ao texto englobam a separação de linhas em campos, organização de livros, organizar requisitantes, carregar livros, carregar freguesias, carregar concelhos, carregar distritos, carregar requisitantes, gravar em XML, gravar sessão e carregar sessão.

A função "`SepararCampos`" pega numa linha lida de um ficheiro e com um número de campos, bem como um delimitador, separa os campos permitindo que estes sejam posteriormente inseridos nas devidas estruturas.

A função "`OrganizarLivros`", com recurso à função "`SepararCampos`" verifica os campos de forma a remover quaisquer campos que não respeitem os requisitos necessários. Esses campos considerados inválidos, são enviados para o ficheiro logs_livros.txt. O conteúdo desse ficheiro é sempre acrescentado, nunca sendo apagado.

A função "`OrganizarRequisitantes`", com recurso à função "`SepararCampos`" verifica os campos de forma a remover quaisquer campos que não respeitem os requisitos necessários. Esses campos considerados inválidos, são enviados para o ficheiro logs.txt. O conteúdo desse ficheiro é sempre acrescentado, nunca sendo apagado.

As funções "`CarregarLivros`", "`CarregarFreguesias`", "`CarregarConcelhos`", "`CarregarDistritos`", "`CarregarRequisitantes`" e "`CarregarSessao`" tratam de carregar os dados para as duas devidas estruturas de dados. As primeiras cinco funções são utilizadas quando o utilizador opta pela opção de criar sessão. A última função é utilizada quando um utilizador opta por continuar com a sessão anterior. Esta sessão anterior é resultante da gravação dos dados anteriormente, seja de forma manual ou na saída do programa.

A função "`GravarXML`" grava toda a informação em formato XML. O utilizador dá o nome que deseja para o ficheiro, sem a adição da extensão no nome, e este é gravado. Esta função faz uso da função "`GuardarLISTA`".

A função "`GravarSessao`" grava toda a informação referente à sessão do utilizador, permitindo guardar todo o seu progresso para que, caso deseje, possa utilizar essa informação para manter o seu progresso.

2.4. Livro

A estrutura utilizada para o armazenamento do Livro é a seguinte.

```c
#ifndef LIVRO_H_INCLUDED
#define LIVRO_H_INCLUDED

#include "Lista.h"
#include "Hashing.h"

typedef struct
{
    char* ISBN;
    char* TITULO;
    char* AUTOR;
    char* AREA;
    char* ANO;
    char ESTADO_REQ[15];
    int N_REQ;
}LIVRO;

#endif // LIVRO_H_INCLUDED
```

Figura 6 - Estrutura Livro

Além das funções mencionadas e explicadas no decorrer das aulas teóricas e práticas, temos funções como "`MostrarLivrosRequisitadosPorRequisitante`", "`GravarLivro`", "`GravarLivro_Sessao`", "`MostrarLivroMaisRequisitado`", "`MostrarLivroMaisRecente`", "`ProcurarLivro`", "`GetLivrosMaisRecentes`", "`GetRequisicoesLivros`", "`GetIDLivro`", "`RequisitarLivro`", "`DevolverLivro`" e "`MostrarRequisicoes`".

Estas funções são relativamente simples, não passando de comparações entre parâmetros, listagens, retorno de parâmetros, alterações de estados de livros, etc. São principalmente chamadas nas funções mencionadas na secção 2.2. Listas.

2.5. Requisitante

A estrutura utilizada para o armazenamento do Requisitante é a seguinte.

```c
#ifndef REQUISITANTE_H_INCLUDED
#define REQUISITANTE_H_INCLUDED

#include <time.h>
#include "Lista.h"
#include "Freguesia.h"

typedef struct
{
    char* ID_REQUISITANTE;
    char* REQUISITANTE;
    char* DATA_NASC;
    char ID_FREGUESIA[7];
    int JA_REQUISITOU;
    int TEM_REQUISICAO;
}REQUISITANTE;

#endif // REQUISITANTE_H_INCLUDED
```

Figura 7 - Estrutura Requisitante

Além das funções mencionadas e explicadas no decorrer das aulas teóricas e práticas, temos funções como "`MostrarRequisitanteOrdernadoFreg`", "`MostrarRequisitanteOrdernadoApelido`", "`GravarRequisitante`", "`GravarRequisitante_Sessao`", "`ProcurarRequisitante`", "`AlterarEstadoRequisitar`", "`AlterarEstadoDevolver`", "`GetTemRequisicoes`", "`GetIdadeRequisitantes`", "`GetNomeRequisitantes`", "`GetSobrenomeRequisitantes`", "`GetNumPessoasFreguesiaApelido`", "`GetIDRequisitante`", "`GetIDFreguesia`" e "`GetNuncaRequisitou`".

Estas funções são também relativamente simples, não passando de comparações entre parâmetros, listagens, retorno de parâmetros, alterações de estados de requisitantes, etc. São principalmente chamadas nas funções mencionadas na secção 2.2. Listas.

2.6. Requisição

A estrutura utilizada para o armazenamento do Requisição é a seguinte.

```
#ifndef REQUISICAO_H_INCLUDED
#define REQUISICAO_H_INCLUDED

#include "Lista.h"

typedef struct
{
    int ID_REQUISITANTE;
    int ID_LIVRO;
}REQUISICAO;

#endif // REQUISICAO_H_INCLUDED
```

Figura 8 - Estrutura Requisição

Além das funções mencionadas e explicadas no decorrer das aulas teóricas e práticas, temos funções como "`MostrarRequisicoesID`", "`GravarRequisicoes`", "`GravarRequisicoes_Sessao`", "`GetIDRequisitante`", "`GetIDRequisitanteEmRequisicoes`" e "`RemoverRequisicao`".

Estas funções são relativamente simples, não passando de comparações entre parâmetros, listagens e retorno de parâmetros. São principalmente chamadas nas funções mencionadas na secção 2.2. Listas.

2.7. Freguesia

A estrutura utilizada para o armazenamento da Freguesia é a seguinte.

```c
#ifndef FREGUESIA_H_INCLUDED
#define FREGUESIA_H_INCLUDED

#include "Lista.h"
#include "Concelho.h"

typedef struct
{
    char* ID_FREGUESIA;
    char* NOME_FREGUESIA;
    char ID_CONCELHO[5];
}FREGUESIA;

#endif // FREGUESIA_H_INCLUDED
```

Figura 9 - Estrutura Freguesia

Além das funções mencionadas e explicadas no decorrer das aulas teóricas e práticas, temos funções como "`GravarFreguesia`" e "`GravarFreguesia_Sessao`".

Estas funções são relativamente simples, não passando de listagens para ficheiros. São principalmente chamadas nas funções mencionadas na secção 2.2. Listas.

Vale notar que na função "`CriarFreguesia`", o ID concelho é conseguido "partindo" o ID freguesia.

```c
int tmp = atoi(id_freguesia);
char zero[3];
strcpy(zero, "0");
char* id_concelho = (char*)malloc((strlen(id_freguesia) + 1) * sizeof(char));
//A partir do id_freguesia, conseguimos "partir" o id conseguido o id_concelho
tmp = tmp / 100;
itoa(tmp, id_concelho, 10);
//Se o id_concelho for < 10, concateno um 0 ao seu valor
if (tmp < 10) id_concelho = strcat(zero, id_concelho);

strcpy(X->ID_CONCELHO, id_concelho);
```

Figura 10 - Obtenção ID concelho

2.8. Concelho

A estrutura utilizada para o armazenamento do Concelho é a seguinte.

```c
#ifndef CONCELHO_H_INCLUDED
#define CONCELHO_H_INCLUDED

#include "Lista.h"
#include "Distrito.h"

typedef struct
{
    char* ID_CONCELHO;
    char* NOME_CONCELHO;
    char ID_DISTRITO[3];
}CONCELHO;

#endif // CONCELHO_H_INCLUDED
```

Figura 11 - Estrutura Concelho

Além das funções mencionadas e explicadas no decorrer das aulas teóricas e práticas, temos funções como "`GravarConcelho`" e "`GravarConcelho_Sessao`".

Estas funções são relativamente simples, não passando de listagens para ficheiros. São principalmente chamadas nas funções mencionadas na secção 2.2. Listas.

Vale notar que na função "`CriarConcelho`", o ID distrito é conseguido "partindo" o ID concelho.

```c
int tmp = atoi(id_concelho);
char zero[3];
strcpy(zero, "0");
char* id_distrito = (char*)malloc((strlen(id_concelho) + 1) * sizeof(char));
//A partir do id_concelho, conseguimos "partir" o id conseguido o id_distrito
tmp = tmp / 100;
itoa(tmp, id_distrito, 10);
//Se o id_distrito for < 10, concateno um 0 ao seu valor
if(tmp < 10) id_distrito = strcat(zero, id_distrito);

strcpy(X->ID_DISTRITO, id_distrito);
```

Figura 12 - Obtenção ID distrito

2.9. Distrito

A estrutura utilizada para o armazenamento do Distrito é a seguinte.

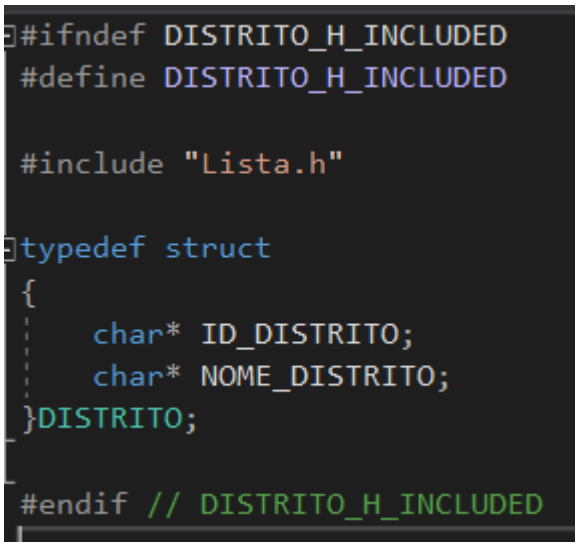


Figura 13 - Estrutura Distrito

Além das funções mencionadas e explicadas no decorrer das aulas teóricas e práticas, temos funções como "`GravarDistrito`", "`GravarDistrito_Sessao`" e "`GetDistritos`".

Estas funções são relativamente simples, não passando de listagens para ficheiros. São principalmente chamadas nas funções mencionadas na secção 2.2. Listas.

3. Conclusão

Concluído o trabalho, podemos dizer que foi um projeto bem-sucedido, tendo todas as funcionalidades pedidas no enunciado com exceção das funcionalidades extra e da paginação nas listagens. Apesar disto, acreditamos também que haja uma boa margem para melhorar no futuro, seja com uma gestão de memória mais eficiente, seja com a adição de requisições de livros utilizando o ID do livro ao invés do nome, caso o utilizador assim deseje.

A realização deste projeto prático permitiu também meter em prática os conhecimentos dados no decorrer das aulas teóricas e práticas deste semestre, bem como fomentar o nosso à vontade no solucionamento de eventuais problemas que aparecem no processo de desenvolvimento, como foi o caso com este projeto.

Em suma, podemos dizer que este projeto prático foi muito benéfico.