P.PORTO

ESCOLA
SUPERIOR
DE TECNOLOGIA
E GESTÃO

Projeto de Laboratório de Programação

Licenciatura em Engenharia Informática

2022/2023

Sónia Oliveira – 8220114

1. Introdução **2**

2. Funcionalidades Requeridas **2**

Estruturas: 2

Funções: 3

Funções gerais: 3

Funções de gestão de vendedores: 5

Funções de gestão de mercados: 6

Funções de atribuição de mercados a vendedores: 7

3. Funcionalidades propostas **7**

4. Estrutura analítica do projeto **8**

5. Funcionalidades implementadas **8**

Funcionalidades de vendedores: 8

Funcionalidades de Mercados: 12

Funcionalidades de gestão de mercados e vendedores: 14

Listagens: 16

Ficheiros: 18

6. Conclusão **20**

1. Introdução

O presente relatório refere-se ao desenvolvimento de um programa, no âmbito do projeto de Laboratório de Programação, que consiste numa aplicação de gestão de mercados, vendedores e comissões.

Para a elaboração deste trabalho recorreu-se à ferramenta NetBeans, um ambiente de desenvolvimento integrado gratuito e de código aberto para desenvolvedores de software, baseado em padrões e módulos escritos na linguagem C.

C é uma linguagem de programação compilada de propósito geral, estrutura, imperativa, processual, padronizada pela Organização Internacional para Padronização, criada em 1972 por Dennis Ritchie na empresa AT&T Bell Labs para desenvolvimento do sistema operacional Unix.

Como objetivo geral pretende-se através de toda a informação dada, ultrapassar as adversidades existentes, como a ocorrência de vários erros, redução de código repetido, entre outras, de forma a concluir o programa da melhor forma tendo em conta a utilização posterior por um utilizador.

Como objetivos principais têm-se a criação, atualização e remoção de mercados e de vendedores, e a associação entre mercados e vendedores, bem como a atribuição de mercados a vendedores.

2. Funcionalidades Requeridas

As funcionalidades requeridas são a gestão de vendedores, mercados e de comissões.

Para efeito de implementar estas funcionalidades, foi proposta a criação de estruturas de dados para um melhor funcionamento do programa. Estas estruturas são compostas por diversos tipos de dados, e são utilizadas para guardar informação ao longo do programa.

Todas as estruturas foram criadas e utilizadas com recurso à memória dinâmica, inicializadas com um espaço inicial, e, caso necessário, expandidas ou libertadas ao longo do programa, impedindo desperdícios ou fugas de memória.

Além disso, todas as funções necessárias ao funcionamento do programa foram implementadas repartidamente por vários ficheiros, devidamente identificados, de modo a manter a informação mais organizada.

Estruturas:

Estruturas no ficheiro geral.h, que são utilizadas em várias partes do programa:

```
typedef enum{
    ATIVO, INATIVO
}Estado; //estrutura que descreve o estado, de um vendedor ou mercado

typedef struct{
    int dia, mes, ano;
}Data;
```

Estruturas em mercados.h, referentes a informações sobre os mercados:

```c
typedef struct{
    char nomeMercado[MAX_NOME_MERC];
    char codigoMercado[MAX_COD_MERCADO];
    Estado estadoMercado;
    int contadorVendedores;
}Mercado; //informações referentes a cada mercado

typedef struct{
    int contador;
    int capacidade;
    Mercado *arrayMercados;
}Mercados; //informações referentes ao array de mercados
```

Estruturas em vendedor.h, referentes a dados sobre o vendedor e comissões de mercados aos quais está atribuído:

```c
typedef struct{
    char codigoMercado[MAX_COD_MERCADO];
    float comissoesVendedor;
    Data dataInicio;
    Data dataFinal;
}MercadoAtribuido; //informação relacionada com um dterminado mercadoa tribuido a um vendedor

typedef struct{
    char codVendedor[COD_VENDEDOR];
    char nomeVendedor[MAX_NOME_VENDEDOR];
    int long telefone;
    char email[MAX_NOME_VENDEDOR];
    MercadoAtribuido *arrayMercadosDeVendedor; //malloc para mercados de vendedor
    int contadorMercados;
    int capacidadeMercados;
    Estado estadoVendedor;
}Vendedor; //informações relacionadas com o vendedor

typedef struct{
    int contador;
    int capacidade;
    Vendedor *arrayVendedores;
}Vendedores; //estrutura para armazenar o array de vendedores
```

Funções:

Funções gerais:

Neste source file, estão implementadas as funções de leitura e apresentação de menus.

Foram desenvolvidas várias funções, uma para cada submenu, que são depois chamadas no menu principal, deste modo:

```
void apresentarMenus() {
    int op;
    do {
        printf("\n--------------------Bem Vindo---------------------\n");
        printf("\n1 - Gerir Vendedores\n");
        printf("\n2 - Gerir Mercados\n");
        printf("\n3 - Mercados e vendedores\n");
        printf("\n4 - Listagens\n");
        printf("\n5 - Guardar\n");
        printf("\n0 - Sair\n");
        printf("\n-------------------------------------------------\n");

        op = obterInt(0, 5, "Opção:\n");
        switch (op) {
            case 1:
                menuVendedor();
                break;
            case 2:
                menuMercados();
                break;
            case 3:
                menuMercadosEVendedores();
                break;
            case 4:
                listagens();
                break;
            case 5:
                guardarVendedores(&vendedores, VENDEDORES_F);
                guardarMercados(&mercados, MERCADOS_F);
                libertarMercados(&mercados);
                libertarVendedores(&vendedores);
                break;

            case 0:
                break;

        }

    } while (op != 0);

}
```

Por sua vez, cada submenu tem em si chamadas as funções referentes a cada funcionalidade.

Funções de gestão de vendedores:

```c
/*...5 lines */
void carregarVendedores(Vendedores *vendedores, char *ficheiro);

/*...5 lines */
void imprimirVendedor(Vendedor vendedor);

/*...6 lines */
int procurarVendedor(Vendedores vendedores, char* codVendedor);

/*...5 lines */
int criarVendedor(Vendedores *vendedores);

/*...5 lines */
void expandirVendedores(Vendedores *vendedores);

/*...5 lines */
void inserirVendedores(Vendedores *vendedores);

/*...5 lines */
void listarVendedores(Vendedores vendedores);

/*...5 lines */
void atualizarVendedor(Vendedores *vendedores);

/*...5 lines */
void libertarVendedores(Vendedores *vendedores);

/*...5 lines */
void removerVendedor(Vendedores *vendedores);

/*...5 lines */
void guardarVendedores(Vendedores *vendedores, char *ficheiro);

/*...6 lines */
int verificarVendedoresDeMercado(Vendedores vendedores, char *codMercado);
```

Estas são todas as funções de gestão de vendedores, sendo que algumas servem de auxílio a outras. No menu de gestão de vendedores, elas são chamadas do seguinte modo:

```c
void menuVendedor() {
    int op;
    do {
        printf("\n-----------------Menu Vendedor-----------------\n");
        printf("\n1 - Criar Vendedor\n");
        printf("\n2 - Atualizar Vendedor\n");
        printf("\n3 - Remover Vendedor\n");
        printf("\n4 - Listar Vendedores\n");
        printf("\n0 - Voltar\n");

        op = obterInt(0, 4, "Opção:\n");

        switch (op) {
            case 1:
                inserirVendedores(&vendedores);
                break;

            case 2:
                atualizarVendedor(&vendedores);
                break;

            case 3:
                removerVendedor(&vendedores);
                break;

            case 4:
                listarVendedores(vendedores);
                break;

            case 0:
                break;
        }
    } while (op != 0);
}
```

Funções de gestão de mercados:

```c
/*...7 lines */
void carregarMercados(Mercados *mercados, char *ficheiro);

/*...5 lines */
void criarMercado(Mercados *mercados);


/*...5 lines */
void atualizarMercados(Mercados *mercados);


/*...6 lines */
void removerMercado(Mercados *mercados, Vendedores *vendedores);


/*...6 lines */
int procurarMercado(Mercados *mercados, char* cod);

/*...5 lines */
void listarMercados(Mercados mercados);

/*...5 lines */
void imprimirMercado(Mercado mercado);

/*...5 lines */
void expandirMercados(Mercados *mercados);

/*...5 lines */
void inserirMercado(Mercados *mercados);

/*...5 lines */
void libertarMercados(Mercados *mercados);

/*...5 lines */
void guardarMercados(Mercados *mercados, char *ficheiro);
```

As funções de gestão de mercados seguem um modelo semelhante às de gestão de vendedores. No menu, são chamadas do seguinte modo:

```c
void menuMercados() {
    int op;

    do {
        printf("\n-----------------Menu Mercados-----------------\n");
        printf("\n1 - Criar Mercado\n");
        printf("\n2 - Atualizar Mercado\n");
        printf("\n3 - Remover Mercado\n");
        printf("\n4 - Listar Mercados\n");
        printf("\n0 - Voltar\n");

        op = obterInt(0, 4, "Opção:\n");
        switch (op) {
            case 1:
                inserirMercado(&mercados);
                break;

            case 2:
                atualizarMercados(&mercados);
                break;

            case 3:
                removerMercado(&mercados, &vendedores);
                break;

            case 4:
                listarMercados(mercados);

            case 0:
                break;
        }
    } while (op != 0);
}
```

Funções de atribuição de mercados a vendedores:

```
/*...8 lines */
int compararDatas(Data dataInicialNova, Data dataFinalNova, Data dataInicial, Data dataFinal);

/*...7 lines */
int procurarMercadoAtribuido(Vendedores vendedores, char *codMercado, char *codVendedor);

/*...5 lines */
void atribuirMercado(Mercados *mercados, Vendedores *vendedores);

/*...4 lines */
void removerMercadoAtribuido(Vendedores *vendedores);
```

A função de atribuição de mercado também se encarrega de pedir ao utilizador o valor percentual da comissão que o vendedor irá receber desse mercado ao longo do período de tempo que nele se encontra ativo.

```
void menuMercadosEVendedores() {
    int op;
    do {
        printf("\n----------------Mercados e Vendedores----------------\n");
        printf("\n1 - Associar mercado a um vendedor\n");
        printf("\n2 - Desassociar mercado a um vendedor\n");
        printf("\n0 - Voltar\n");

        op = obterInt(0, 2, "Opção:\n");
        switch (op) {
            case 1:
                atribuirMercado(&mercados, &vendedores);
                break;

            case 2:
                removerMercadoAtribuido(&vendedores);
                break;

            case 0:
                break;
        }
    } while (op != 0);
}
```

Este é o menu de mercados e vendedores.

3. Funcionalidades propostas

Além das funcionalidades de gestão do programa, foram pedidas 3 listagens. Para o meu projeto, projetei as seguintes:

Mostrar os mercados mais populados - É de interesse para a empresa, e demonstra os mercados mais ativos.

Mostrar os vendedores com mais mercados - É útil pois permite saber quais os vendedores mais ativos, e quais podem necessitar de mais atenção por falta de trabalho.

Listar os vendedores com as comissões mais altas - Procura de entre os vendedores o que tem a comissão mais alta e lista os vendedores que têm comissões iguais.

Também me pareceu pertinente existirem funções de listagem tanto de mercados como de vendedores, e como era algo simples, também foi implementado.

4. Estrutura analítica do projeto

Foi feito um planeamento muito geral do projeto antes de o começar, a partir da interpretação do enunciado. A partir daí, todos os dias até ao dia de entrega foram-se abordando partes do projeto, desenvolvendo um objetivo para cada dia.

10/2	11/2	12/2	13/2	14/2
Leitura do enunciado e planeamento	Declaração das funções necessárias aos mercados	Declaração das funções necessárias aos vendedores	Resolução de problemas e despiste de bugs	Declaração das funções necessárias às comissões
Organização básica dos header e source files a utilizar, criação de menus	Implementação dessas funções mais qualquer função de auxílio que fosse necessária	Implementação dessas funções mais qualquer função de auxílio que fosse necessária	Retificação de problemas relacionados com memória dinâmica	Implementação dessas funções mais qualquer função de auxílio que fosse necessária

15/2	16/2	17/2	18/2	19/2
Adaptação das estruturas para acomodar os mercados atribuídos e comissões da forma mais simples possível	Resolução de erros de alocação de memória e de declaração de funções e estruturas nos header files	Documentação e comentários	Elaboração do relatório	Últimos testes, correção dos problemas em ficheiros
Adaptação das funções de vendedores de acordo com as funcionalidades das comissões	Despiste de bugs e implementação das verificações solicitadas (não sobreposição de períodos, etc..). Criação de listagens	Tentativa de resolver problemas persistentes quanto à escrita e leitura em ficheiros	Revisão do código e nova tentativa de resolução dos problemas de leitura e escrita em ficheiro	Entrega

5. Funcionalidades implementadas

Funcionalidades de vendedores:

➔ **Inserir Vendedores:**

```
void inserirVendedores(Vendedores *vendedores) {
    if (vendedores->contador == vendedores->capacidade) {
        expandirVendedores(vendedores);
    }

    if (vendedores->contador < vendedores->capacidade) {
        if (criarVendedor(vendedores) == -1) {
            puts("Vendedor já existe.\n");
        }
    } else {
        puts("Não é possível registar um novo vendedor.\n");
    }
}
```

Com esta função, é possível criar um novo vendedor e inseri-lo no array de vendedores.

Ela utiliza como auxílio as funções **expandirVendedores** e **criarVendedor**.

Deste modo, verifica se existe espaço no array para introduzir outro vendedor, aumentando o seu tamanho se necessário, e depois chama a função **criarVendedor** para pedir ao utilizador todas as informações do vendedor e inseri-lo em array. Caso não consiga, apresenta uma mensagem de erro.

expandirVendedores:

```c
void expandirVendedores(Vendedores *vendedores) {
    int tam = (vendedores->capacidade) == 0 ? VENDEDORES_INICIAL : vendedores->capacidade * 2;
    Vendedor *temp = (Vendedor *) realloc(vendedores->arrayVendedores, sizeof (Vendedor) * tam);
    if (temp != NULL) {
        vendedores->capacidade = tam;
        vendedores->arrayVendedores = temp;
    }
}
```

criarVendedor:

```c
int criarVendedor(Vendedores *vendedores) {
    long int telefone;
    lerString(vendedores->arrayVendedores[vendedores->contador].nomeVendedor, MAX_NOME_VENDEDOR,
            "Insira o nome do vendedor:\n");
    lerString(vendedores->arrayVendedores[vendedores->contador].codVendedor, COD_VENDEDOR,
            "Insira o codigo do vendedor (4 dígitos):\n");
    lerString(vendedores->arrayVendedores[vendedores->contador].email, MAX_NOME_VENDEDOR,
            "Insira o email do vendedor:\n");
    telefone = obterInt(900000000, 999999999, "Número de Telefone:");
    vendedores->arrayVendedores[vendedores->contador].telefone = telefone;
    cleanInputBuffer();

    if (procurarVendedor(*vendedores, vendedores->arrayVendedores[vendedores->contador].codVendedor) == -1) {
        vendedores->arrayVendedores[vendedores->contador].estadoVendedor = INATIVO;

        vendedores->arrayVendedores[vendedores->contador].contadorMercados = 0;
        vendedores->arrayVendedores[vendedores->contador].capacidadeMercados = TAM_INICIAL_MERCADOS;
        vendedores->arrayVendedores[vendedores->contador].arrayMercadosDeVendedor = (MercadoAtribuido *) malloc(
                TAM_INICIAL_MERCADOS * sizeof (MercadoAtribuido));
        return vendedores->contador++;
    } else {
        return -1;
    }
}
```

➔ **Atualizar Vendedores:**

```
void atualizarVendedor(Vendedores *vendedores) {

    char* codVendedor = malloc(COD_VENDEDOR * sizeof (char));
    char* nomeVendedor = malloc(MAX_NOME_VENDEDOR * sizeof (char));
    char* email = malloc(MAX_NOME_VENDEDOR * sizeof (char));
    long int telefone;
    Estado estado;
    lerString(codVendedor, COD_VENDEDOR, "Insira o codigo do vendedor:\n");
    int vendedor = procurarVendedor(*vendedores, codVendedor);

    if (vendedor != -1) {
        int op1;
        printf("Deseja alterar as informações pessoais do vendedor?\n1 - Sim\n2 - Nao");
        op1 = obterInt(1, 2, "Opção:\n");
        switch (op1) {
            case 1:
                lerString(nomeVendedor, MAX_NOME_VENDEDOR, "Insira o nome do vendedor:\n");
                lerString(email, MAX_NOME_VENDEDOR, "Insira o email do vendedor:\n");
                telefone = obterInt(900000000, 999999999, "Número de Telefone:");
                strcpy(vendedores->arrayVendedores[vendedor].nomeVendedor, nomeVendedor);
                strcpy(vendedores->arrayVendedores[vendedor].email, email);
                vendedores->arrayVendedores[vendedor].telefone = telefone;
                break;
            case 2:
                break;
        }
```

```
        int op2;

        printf("Deseja alterar o estado do vendedor?\n");
        printf("1 - Sim\n2 - Não\n");

        op2 = obterInt(1, 2, "Opção:\n");

        switch (op2) {
            case 1:
                if (vendedores->arrayVendedores[vendedor].estadoVendedor == ATIVO) {
                    vendedores->arrayVendedores[vendedor].estadoVendedor = INATIVO;
                } else {
                    vendedores->arrayVendedores[vendedor].estadoVendedor = ATIVO;
                }
                printf("Estado alterado com sucesso.\n");
                break;

            case 2:
                break;
        }
    }
    free(codVendedor);
    free(nomeVendedor);
    free(email);
```

Esta função procura, através do código do vendedor, o vendedor que o utilizador deseja alterar com o auxílio da função **procurarVendedor**. Depois, se este existir, pede ao utilizador as novas informações do vendedor e pergunta se o utilizador quer mudar o estado do vendedor.

procurarVendedor:

```c
int procurarVendedor(Vendedores vendedores, char *codVendedor) {
    int i;
    for (i = 0; i < vendedores.contador; i++) {
        if (strcmp(vendedores.arrayVendedores[i].codVendedor, codVendedor) == 0) {
            return i;
        }
    }
    return -1;
}
```

➔ **Remover Vendedores:**

```c
void removerVendedor(Vendedores *vendedores) {
    int i, vendedor;

    char codVendedor[COD_VENDEDOR];
    lerString(codVendedor, COD_VENDEDOR, "Insira o codigo do vendedor:\n");

    vendedor = procurarVendedor(*vendedores, codVendedor);

    if (vendedor != -1) {
        if (vendedores->arrayVendedores[vendedor].contadorMercados > 0) {
            printf("Não é possivel remover o vendedor, pois ele ainda está associado a mercado(s).\n");
            return;
        }

        free(vendedores->arrayVendedores[vendedor].arrayMercadosDeVendedor);
        for (i = vendedor; i < vendedores->contador - 1; i++) {
            vendedores->arrayVendedores[i] = vendedores->arrayVendedores[i + 1];
        }
        vendedores->contador--;
    } else {
        puts("Vendedor não existe.\n");
    }
}
```

Esta função procura o vendedor a eliminar através do seu código e verifica se não está associado a nenhum mercado. Se não estiver, apaga as informações desse vendedor e remove-o do array, preenchendo o espaço vazio por si deixado ao mover para a esquerda as posições do array que lhe seguiam.

➔ **Listar Vendedores:**

```c
void listarVendedores(Vendedores vendedores) {
    int i;

    if (vendedores.contador > 0) {
        for (i = 0; i < vendedores.contador; i++) {
            imprimirVendedor(vendedores.arrayVendedores[i]);
        }
    } else {
        puts("Lista vazia.\n");
    }
}
```

Esta função itera pelo array de vendedores e imprime a sua informação com o auxílio da função **imprimirVendedor.**

```c
void imprimirVendedor(Vendedor vendedor) {
    int i;
    printf("Código:%s\nNome:%s\nE-mail:%s\nNúmero de Telefone:%ld\n", vendedor.codVendedor, vendedor.nomeVendedor, vendedor.email, vendedor.telefone);
    if (vendedor.estadoVendedor == 0) {
        printf("Estado do Vendedor: ATIVO\n");
    } else {
        printf("Estado do Vendedor: INATIVO\n");
    };
    printf("Mercados:\n");
    for (i = 0; i < vendedor.contadorMercados; i++) {
        printf("%s (Comissão de %2.2f%%)\n\n", vendedor.arrayMercadosDeVendedor[i].codigoMercado, vendedor.arrayMercadosDeVendedor[i].comissoesVendedor);
    }
    puts("----------------------------");
}
```

Funcionalidades de Mercados:

As funcionalidades dos mercados seguem o mesmo modelo das dos vendedores.

➔ **Inserir Mercados**

```
void inserirMercado(Mercados *mercados) {
    if (mercados->contador == mercados->capacidade) {
        expandirMercados(mercados);
    }

    if (mercados->contador < mercados->capacidade) {
        criarMercado(mercados);
    }

}
```

As funções de auxílio são idênticas às dos vendedores.

➔ **Atualizar Mercados**

```
void atualizarMercados(Mercados *mercados) {
    int i;
    char* codMercado = malloc(MAX_COD_MERCADO * sizeof(char));
    lerString(codMercado, MAX_COD_MERCADO, "Insira o código do mercado:\n");
    int mercado = procurarMercado(mercados, codMercado);
    int op;
    printf("Deseja alterar o estado do mercado?\n");
    printf("1 - Sim\n 2 - Não\n");

    op = obterInt(1, 2, "Opção:\n");

    switch (op) {
        case 1:
            if (mercados->arrayMercados[mercado].estadoMercado == ATIVO) {
                mercados->arrayMercados[mercado].estadoMercado = INATIVO;
            } else {
                mercados->arrayMercados[mercado].estadoMercado = ATIVO;
            }
            break;

        case 2:
            break;
    }

    free(codMercado);
}
```

Esta função apenas permite alterar o estado de um mercado de ativo para inativo, visto que não achei pertinente alterar outro tipo de informação.

➔ **Remover Mercados**

```
void removerMercado(Mercados *mercados, Vendedores *vendedores) {

    char codMercado[MAX_COD_MERCADO];
    lerString(codMercado, MAX_COD_MERCADO, "Insira o código do mercado:\n");

    int mercado = procurarMercado(mercados, codMercado);

    if (mercado != -1) {

        int numVendedores = verificarVendedoresDeMercado(*vendedores, codMercado);
        if (numVendedores > 0) {
            puts("Não é possível remover o mercado, pois existem vendedores associados a ele.");
            return;
        }

        for (int i = mercado; i < mercados->contador - 1; i++) {
            mercados->arrayMercados[i] = mercados->arrayMercados[i + 1];
        }
        mercados->contador--;
        puts("Mercado removido com sucesso.");
    } else {
        puts("Mercado não encontrado.");
    }
}
```

Esta função verifica se não existem vendedores associados a um certo mercado com o auxílio da função **verificarVendedoresDeMercado**, cujo retorno é o número de vendedores associados a um determinado mercado.

```
int verificarVendedoresDeMercado(Vendedores vendedores, char *codMercado) {
    int numVendedores = 0;
    for (int i = 0; i < vendedores.contador; i++) {
        for (int j = 0; j < vendedores.arrayVendedores[i].contadorMercados; j++) {
            if (strcmp(codMercado, vendedores.arrayVendedores[i].arrayMercadosDeVendedor[j].codigoMercado) == 0) {
                numVendedores++;
            }
        }
    }
    return numVendedores;
}
```

➔ **Listar Mercados**

```
void listarMercados(Mercados mercados) {
    int i;

    if (mercados.contador > 0) {
        for (i = 0; i < mercados.contador; i++) {
            imprimirMercado(mercados.arrayMercados[i]);
        }
    } else {
        puts("Não esistem mercados a listar.\n");
    }
}
```

Funcionalidades de gestão de associação entre mercados e vendedores:

➔ **Atribuição de mercado:**

```
void atribuirMercado(Mercados *mercados, Vendedores *vendedores) {

    char codMercado[MAX_COD_MERCADO];
    char codVendedor[COD_VENDEDOR];
    int mercado = 0, vendedor = 0, i;
    float comissao;
    Data dataInicialNova, dataFinalNova;
    int contador;

    if (vendedores->arrayVendedores->contadorMercados == vendedores->arrayVendedores->capacidadeMercados) {
        expandirArrayMercadosVendedor(vendedores);
    }

    lerString(codMercado, MAX_COD_MERCADO, "Insira o código do mercado:\n");
    mercado = procurarMercado(mercados, codMercado);
    if (mercado == -1) {
        puts("O mercado não existe.\n");
        return;
    }

    lerString(codVendedor, COD_VENDEDOR, "Insira o codigo do vendedor:\n");
    vendedor = procurarVendedor(*vendedores, codVendedor);
    if (vendedor == -1) {
        puts("O vendedor não existe.\n");
        return;
    }
    int contadorMercados = vendedores->arrayVendedores[vendedor].contadorMercados;

    strcpy(vendedores->arrayVendedores[vendedor].arrayMercadosDeVendedor[contadorMercados].codigoMercado, codMercado);

    comissao = obterFloat(10, 90, "Insira o valor da comissão (percentagem):\n");
    vendedores->arrayVendedores[vendedor].arrayMercadosDeVendedor[mercado].comissoesVendedor = comissao;

    dataInicialNova.dia = obterInt(1, 31, "Dia de início:\n");
    dataInicialNova.mes = obterInt(1, 12, "Mes de início:\n");
    dataInicialNova.ano = obterInt(2023, 2025, "Ano de início:\n");

    dataFinalNova.dia = obterInt(1, 31, "Dia de fim:\n");
    dataFinalNova.mes = obterInt(1, 12, "Mes de fim:\n");
    dataFinalNova.ano = obterInt(2023, 2025, "Ano de fim:\n");
```

```
    contador = compararDatas(dataInicialNova, dataFinalNova, vendedores->arrayVendedores[vendedor].arrayMercadosDeVendedor[mercado].dataInicio,
            vendedores->arrayVendedores[vendedor].arrayMercadosDeVendedor[mercado].dataFinal);
    if (contador != 0) {
        printf("O vendedor já está associado a este mercado neste período de tempo.\n");
        return;
    }
    vendedores->arrayVendedores[vendedor].arrayMercadosDeVendedor[contadorMercados].dataInicio = dataInicialNova;
    vendedores->arrayVendedores[vendedor].arrayMercadosDeVendedor[contadorMercados].dataFinal = dataFinalNova;
    vendedores->arrayVendedores[vendedor].contadorMercados++;
    mercados->arrayMercados[mercado].contadorVendedores++;
    puts("Mercado atribuido com sucesso.");
}
```

Esta é a função mais extensa de todo o programa. Ela começa por verificar se existe espaço no array de mercados do vendedor, caso contrário, expande-o, utilizando a função **expandirArrayMercadosVendedor.**

Depois, pede os códigos de mercado e de utilizador, e caso estes existam, procede com o funcionamento da função, pedindo ao utilizador as datas de início e de fim de atividade, bem como o valor percentual da comissão.

Para garantir que um vendedor não é associado ao mesmo mercado no mesmo período de tempo, o programa usa a função **compararDatas,** que funciona à base de um contador. Caso o contador seja maior que 1, quer dizer que a data nova inserida pelo utilizador coincide com a data de comissão já registada para aquele vendedor naquele mercado.

Depois disso, preenche a variável com a nova informação dada pelo utilizador, e incrementa o contador de mercados no vendedor, bem como o contador de vendedores no mercado.

expandirArrayMercadosVendedor:

```c
void expandirArrayMercadosVendedor(Vendedores *vendedores) {
    int tam = (vendedores->arrayVendedores->capacidadeMercados) == 0 ? MERCADOS_ATRIBUIDOS_INICIAL : vendedores->arrayVendedores->capacidadeMercados * 2;
    MercadoAtribuido *temp = (MercadoAtribuido *) realloc(vendedores->arrayVendedores->arrayMercadosDeVendedor, sizeof (MercadoAtribuido) * tam);
    if (temp != NULL) {
        vendedores->arrayVendedores->capacidadeMercados = tam;
        vendedores->arrayVendedores->arrayMercadosDeVendedor = temp;
    }
}
```

compararDatas:

```c
int compararDatas(Data dataInicialNova, Data dataFinalNova, Data dataInicial, Data dataFinal) {
    if ((dataInicialNova.ano < dataFinal.ano ||
            (dataInicialNova.ano == dataFinal.ano && dataInicialNova.mes < dataFinal.mes) ||
            (dataInicialNova.ano == dataFinal.ano && dataInicialNova.mes == dataFinal.mes && dataInicialNova.dia <= dataFinal.dia))
        && (dataInicial.ano < dataFinalNova.ano ||
            (dataInicial.ano == dataFinalNova.ano && dataInicial.mes < dataFinalNova.mes) ||
            (dataInicial.ano == dataFinalNova.ano && dataInicial.mes == dataFinalNova.mes && dataInicial.dia <= dataFinalNova.dia))) {
        return 1;
    } else {
        return 0;
    }
}
```

➔ **Desassociar mercados e vendedores**

```c
void removerMercadoAtribuido(Mercados *mercados, Vendedores *vendedores) {

    char codMercado[MAX_COD_MERCADO];
    char codVendedor[COD_VENDEDOR];

    lerString(codMercado, MAX_COD_MERCADO, "Insira o código do mercado:\n");

    lerString(codVendedor, COD_VENDEDOR, "Insira o código do vendedor:\n");

    int vendedor = procurarVendedor(*vendedores, codVendedor);
    int mercadoAssociado = procurarMercadoAtribuido(*vendedores, codMercado, codVendedor);
    int mercado = procurarMercado(mercados, codMercado);
    if (mercadoAssociado != -1) {
        for (int i = mercadoAssociado; i < vendedores->arrayVendedores[vendedor].contadorMercados - 1; i++) {
            for (int j = i; j < vendedores->arrayVendedores[vendedor].contadorMercados - 1; j++) {
                vendedores->arrayVendedores[vendedor].arrayMercadosDeVendedor[j] = vendedores->arrayVendedores[vendedor].arrayMercadosDeVendedor[j + 1];
            }
        }
        vendedores->arrayVendedores[vendedor].contadorMercados--;
        mercados->arrayMercados[mercado].contadorVendedores--;
        puts("Mercado removido com sucesso.\n");
    } else {
        puts("Não foi encontrado um mercado associado a este vendedor.\n");
    }
```

Esta função verifica a existência tanto do mercado como do vendedor, bem como se o mercado está associado ao vendedor. Caso afirmativo, remove o mercado da lista de mercados atribuídos do vendedor, e decrementa o contador de mercados no vendedor, bem como o contador de vendedores no mercado.

procurarMercadoAtribuido:

```c
int procurarMercadoAtribuido(Vendedores vendedores, char *codMercado, char *codVendedor) {

    int i;
    for (i = 0; i < vendedores.contador; i++) {
        if (strcmp(vendedores.arrayVendedores[i].codVendedor, codVendedor) == 0) {
            for (int j = 0; j < vendedores.arrayVendedores[i].contadorMercados; i++) {
                if (strcmp(vendedores.arrayVendedores[i].arrayMercadosDeVendedor[j].codigoMercado, codMercado) == 0) {
                    return j;
                }
            }
        }
        return -1;
    }
}
```

Listagens:

```c
void listagens() {
    int op;
    do {
        printf("\n-----------------Listagens-----------------\n");
        printf("\n1 - Vendedores ordenados por número de mercados\n");
        printf("\n2 - Mercados mais populados\n");
        printf("\n3 - Comissoes mais altas por vendedor\n");
        printf("\n0 - Voltar\n");

        op = obterInt(0, 3, "Opção:\n");
        switch (op) {
            case 1:
                listarVendedoresComMaisMercados(&vendedores);
                break;

            case 2:
                int n;
                printf("Quantos mercados quer listar?\n");
                scanf("%d", &n);
                imprimirNMercadosMaisPopulados(&mercados,n);
                break;

            case 3:
                listarVendedoresMaiorComissao(&vendedores);
                break;

            case 0:
                break;
        }
    } while (op != 0);
}
```

Estas foram as listagens implementadas.

➔ **Vendedores ordenados por número de mercados:**

```c
void listarVendedoresComMaisMercados(Vendedores *vendedores) {
    int i, j;
    Vendedor temp;

    for (i = 0; i < vendedores->contador; i++) {
        for (j = i+1; j < vendedores->contador; j++) {
            if (vendedores->arrayVendedores[i].contadorMercados < vendedores->arrayVendedores[j].contadorMercados) {
                temp = vendedores->arrayVendedores[i];
                vendedores->arrayVendedores[i] = vendedores->arrayVendedores[j];
                vendedores->arrayVendedores[j] = temp;
            }
        }
    }

    printf("Ranking dos vendedores por nº de mercados em que estão ativos:\n");
    for (i = 0; i < vendedores->contador; i++) {
        printf("%d. %s: %d mercados ativos\n", i+1, vendedores->arrayVendedores[i].nomeVendedor, vendedores->arrayVendedores[i].contadorMercados);
    }
}
```

➔ **Mercados mais populados:**

```c
void imprimirNMercadosMaisPopulados(Mercados *mercados, int n) {
    if (mercados->contador == 0) {
        printf("Não há mercados registados.\n");
        return;
    }

    Mercado *tempArray = (Mercado*) malloc(mercados->contador * sizeof(Mercado));
    memcpy(tempArray, mercados->arrayMercados, mercados->contador * sizeof(Mercado));

    qsort(tempArray, mercados->contador, sizeof(Mercado), compararMercados);

    printf("Os %d mercados com mais vendedores ativos:\n", n);
    for (int i = 0; i < n && i < mercados->contador; i++) {
        printf("%d. %s (%d vendedores)\n", i+1, tempArray[i].nomeMercado, tempArray[i].contadorVendedores);
    }

    free(tempArray);
}
```

Esta função cria um array de mercados temporário, com uma cópia dos mercados, e depois usa a função **qsort**, que é uma função em C que ordena arrays, e necessita de uma função de comparação como parâmetro, que, por sua vez, receba como parâmetro dois ponteiros do tipo **void**. Neste caso, tenho a função **compararMercados.**

```c
int compararMercados(const void *a, const void *b) {
    const Mercado *mercadoA = a;
    const Mercado *mercadoB = b;
    return mercadoB->contadorVendedores - mercadoA->contadorVendedores;
}
```

Depois disso, a função imprime os n mercados mais populados no sistema.

➔ **Listar os vendedores com a comissão mais alta**

```c
void listarVendedoresMaiorComissao(Vendedores *vendedores) {
    int i, j;
    float maxComissao = 0;
    Vendedor *vendedoresMaiorComissao;
    int numVendedoresMaiorComissao = 0;

    vendedoresMaiorComissao = (Vendedor*) malloc(vendedores->contador * sizeof(Vendedor));

    for (i = 0; i < vendedores->contador; i++) {
        for (j = 0; j < vendedores->arrayVendedores[i].contadorMercados; j++) {
            if (vendedores->arrayVendedores[i].arrayMercadosDeVendedor[j].comissoesVendedor > maxComissao) {
                maxComissao = vendedores->arrayVendedores[i].arrayMercadosDeVendedor[j].comissoesVendedor;
            }
        }
    }


    for (i = 0; i < vendedores->contador; i++) {
        for (j = 0; j < vendedores->arrayVendedores[i].contadorMercados; j++) {
            if (vendedores->arrayVendedores[i].arrayMercadosDeVendedor[j].comissoesVendedor == maxComissao) {
                vendedoresMaiorComissao[numVendedoresMaiorComissao++] = vendedores->arrayVendedores[i];
                break;
            }
        }
    }

    printf("Vendedores com maior comissão: ");
    for (i = 0; i < numVendedoresMaiorComissao; i++) {
        printf("%s ", vendedoresMaiorComissao[i].nomeVendedor);
    }
    printf("\n");

    free(vendedoresMaiorComissao);
}
```

A função procura dentre os vendedores o(s) que tem a comissão mais alta, guarda-os num array, e lista-o(s).

Ficheiros:

➔ Carregar dados de vendedores

```
void carregarVendedores(Vendedores *vendedores, char *ficheiro) {

    int i, sucesso = 0;

    FILE *apontadorFicheiro = fopen(ficheiro, "rb");

    if (apontadorFicheiro != NULL) {

        fread(&vendedores->contador, sizeof (int), 1, apontadorFicheiro);

        if (vendedores->contador > 0) {
            vendedores->capacidade = vendedores->contador;
            vendedores->arrayVendedores = (Vendedor *) malloc(vendedores->capacidade * sizeof (Vendedor));


            for (i = 0; i < vendedores->contador; i++) {
                fread(&vendedores->arrayVendedores[i], sizeof (Vendedor), 1, apontadorFicheiro);
                vendedores->arrayVendedores[i].arrayMercadosDeVendedor =
                        malloc(vendedores->arrayVendedores[i].capacidadeMercados * sizeof (MercadoAtribuido));
                fread(vendedores->arrayVendedores[i].arrayMercadosDeVendedor,
                        sizeof (MercadoAtribuido), vendedores->arrayVendedores[i].contadorMercados, apontadorFicheiro);
            }
            sucesso = 1;
        }


    }
    if (sucesso != 1) {
        apontadorFicheiro = fopen(ficheiro, "wb");
        if (apontadorFicheiro != NULL) {
            vendedores->arrayVendedores = (Vendedor *) malloc(VENDEDORES_INICIAL * sizeof (Vendedor));
            vendedores->contador = 0;
            vendedores->capacidade = VENDEDORES_INICIAL;

        }
    }
     fclose(apontadorFicheiro);
}
```

➔ Guardar dados de vendedores

```
void guardarVendedores(Vendedores *vendedores, char *ficheiro) {

    int i;

    FILE *apontadorFicheiro = fopen(ficheiro, "wb");

    if (apontadorFicheiro != NULL) {
        fwrite(&vendedores->contador, sizeof (int), 1, apontadorFicheiro);

        if (vendedores->contador > 0) {
            for (i = 0; i < vendedores->contador; i++) {
                fwrite(&vendedores->arrayVendedores[i], sizeof (Vendedor), 1, apontadorFicheiro);
                fwrite(vendedores->arrayVendedores[i].arrayMercadosDeVendedor, sizeof (MercadoAtribuido),
                        vendedores->arrayVendedores[i].contadorMercados, apontadorFicheiro);
            }
        }
        fclose(apontadorFicheiro);
    } else {
        puts("Erro ao guardar vendedores.");
    }
    puts("Guardado com sucesso.\n");
}
```

➔ **Carregar dados de mercados**

```c
void carregarMercados(Mercados *mercados, char *ficheiro) {
    int i, sucesso = 0;

    FILE *apontadorFicheiro = fopen(ficheiro, "rb");
    if (apontadorFicheiro != NULL) {

        fread(&mercados->contador, sizeof (int), 1, apontadorFicheiro);

        if (mercados->contador > 0) {

            mercados->capacidade = mercados->contador;
            mercados->arrayMercados = (Mercado*) malloc(mercados->contador * sizeof (Mercado));

            for (i = 0; i < mercados->contador; i++) {
                fread(&mercados->arrayMercados[i], sizeof (Mercado), 1, apontadorFicheiro);

            }
            sucesso = 1;
        }
        fclose(apontadorFicheiro);
    }

    if (sucesso != 1) {
        apontadorFicheiro = fopen(ficheiro, "wb");

        if (apontadorFicheiro != NULL) {
            mercados->arrayMercados = (Mercado*) malloc(TAM_INICIAL_MERCADOS * sizeof (Mercado));
            mercados->contador = 0;
            mercados->capacidade = TAM_INICIAL_MERCADOS;

            fclose(apontadorFicheiro);
        }
    }
}
```

➔ **Guardar dados de mercados**

```c
void guardarMercados(Mercados *mercados, char *ficheiro) {
    int i;

    FILE *apontadorFicheiro = fopen(ficheiro, "wb");

    if (apontadorFicheiro == NULL) {
        printf("Não foi possível guardar.\n");
        exit(EXIT_FAILURE);
    }

    fwrite(&mercados->contador, sizeof (int), 1, apontadorFicheiro);

    for (i = 0; i < mercados->contador; i++) {
        fwrite(&mercados->arrayMercados[i], sizeof (Mercado), 1, apontadorFicheiro);
    }

    fclose(apontadorFicheiro);
}
```

6. Conclusão

Apesar de certos constrangimentos e dificuldades ao longo da execução do projeto, foi possível realizá-lo e consolidar os conhecimentos obtidos na UC, no que toca em programação em C, que não é das mais simples de desenvolver.

Sinto que expandi os meus conhecimentos, mesmo que ainda tenha imenso a aprender, e haja imensa coisa que possa melhorar no meu projeto.