Licenciatura em Engenharia de Sistemas Informáticos – PL

Estrutura de Dados Avançadas

Ano 2022/2023

RELATÓRIO PRÁTICO

Discente:

Carlos Eduardo Bernardes de Sousa n.º 24880

Índice

Índice 2
Introdução 3
Objetivos 4
Estruturação 5
Código 7
Usabilidade 37
Conclusão 39

Informações adicionais:

Repositório GitHub: https://github.com/carlossousa1/EDA-24880

Introdução

Este relatório foi desenvolvido no âmbito do trabalho prático da unidade curricular de Estrutura de Dados Avançadas, do curso de licenciatura LESI-PL.

O objetivo deste relatório centra-se em comentar e documentar o código final desenvolvido na linguagem C, pondo em conta a temática posta no enunciado sobre a manipulação de estruturas de dados dinâmicas.

O corpo deste relatório encontra-se sectorizado em 4 componentes distintas, 1 - Objetivos, 2 - Estruturação, 3 - Código e 4 – Usabilidade, salientando-se também a secção da introdução e conclusão do projeto.

Dado isto, pretende-se com este trabalho, aprofundar os conhecimentos obtidos em aula e por conta própria sobre a temática das Estruturas de Dados Avançadas. Temática esta assimilada durante todo o decorrer do semestre letivo, assim como demonstrar a ligação ao todo da temática em um projeto independente e funcional.

Objetivos

Como é constado no enunciado, visa-se por meio deste projeto desenvolver uma solução em software para contornar um problema de dimensão média.

Pretende-se com esta solução, agilizar a gestão de meios de mobilidade urbana sob o contexto de uma "smart-city", recorrendo ao uso de estruturas dinâmicas, armazenamento em ficheiros, modularização e estruturação adequada do projeto.

Contemplando dois tipos distintos de utilizadores (Cliente e Gestor), considerou-se neste programa os seguintes objetivos:

- Agilizar o aluguer dos meios, disponibilizando uma estruturação dos meios disponíveis ligados e localizados a partir de um "geocódigo", além de disponibilizar aos clientes uma forma de pagamento através de um saldo recarregável;
- Permitir aos administradores (gestores) gerirem os diversos meios existentes a partir de funções de criação, edição, remoção e armazenamento de dados;
- Admitir somente clientes registados, considerando informações adicionais como o NIF, saldo, morada etc.;
- Permitir aos gestores a possibilidade de obter uma visão integrada a partir da consulta de históricos, estatísticas, validações etc...

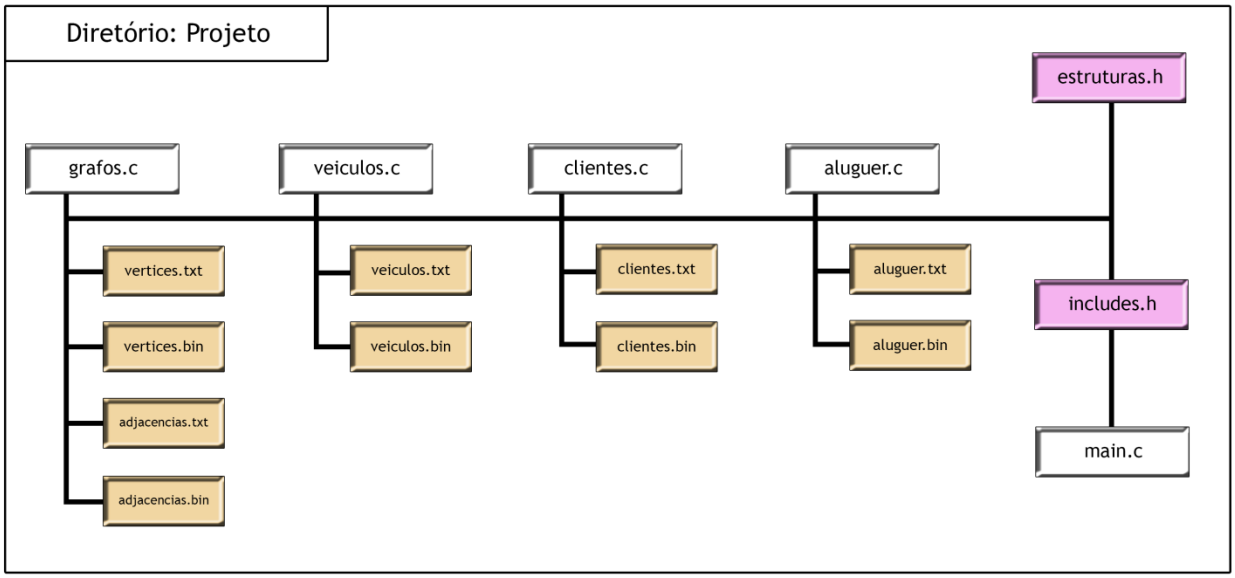
Posto isto, dividiu-se o desenvolvimento deste projeto em duas fases distintas, pondo em conta o conjunto de questões e período de defesa:

- Fase 1: Fase introdutória, foi voltada para o desenvolvimento das estruturas assim como o desenvolvimento das funções base que garantissem a gestão dos dados;
- Fase 2: Fase final, voltada para o desenvolvimento dos processos complexos. Recorreu-se nesta fase o desenvolvimento de funções estatísticas, pondo em mente conceitos avançados como a localização de cada dado.

Estruturação

Dado o enunciado, este projeto foi divido em diversos documentos diferenciados, cada um voltado para um tipo de dado e objetivo, distinguindo-se entre eles documentos em C (código), "header files" (ligação de dados) e ficheiros (ficheiros de texto e binários para leitura a armazenamento de dados):

Esquema (ficheiros principais):



Ficheiros:

- Ficheiro header "estrutura.h"

  Ficheiro do tipo "header" que contém todas estruturas predefinidas para garantir o funcionamento das listas e do grafo do projeto;
- Ficheiro header "includes.h"

  Ficheiro do tipo "header" responsável por interligar todos documentos ao "main.c". Contém a chamada de todos outros ficheiros C e a ligação com as estruturas predefinidas do "estruturas.h";
- Ficheiros C "veiculos.c", "clientes.c" e "aluguer.c":

  Conjunto de ficheiros com comandos de gestão semelhantes (inserção, remoção,

edição etc...), mas voltado para diferentes tipos de estruturas (veículos, clientes e alugueres). Acompanhado das funcionalidades de manuseio, estes ficheiros possuem funções de leitura e armazenamento de dados em ficheiros binários;

- Ficheiro C “grafos.c”:

  Ficheiro voltado exclusivamente para a segunda fase do projeto, este projeto contém as funções de manuseio de grafos, assim como as instruções mais completas envolvendo leitura dos dados em uma determinada localização;

- Ficheiros de texto (.txt) e binários (.bin):

  Documentos voltados para inserção e armazenamento de dados. Os ficheiros de texto são voltados para leituras de teste, sendo os mesmos exclusivos para os gestores testarem as funções durante a inicialização do programa. Já os ficheiros binários são voltados para o armazenamento de informação e leitura para clientes (e também gestores) durante a inicialização do programa;

- Ficheiro C “utilizador.c”:

  Contém as funções exclusivas dos clientes, geralmente voltadas para a gestão do saldo e da verificação de “login” (Se é gestor ou cliente).

Código

Nesta secção do relatório, estará um breve resumo de cada funcionalidade inserida neste programa (organizado entre os ficheiros distintos):

- estruturas.h:

  Contém as estruturas predefinidas das listas e dos grafos, sendo as listas correspondentes a registo_clientes, registo_veiculos e registo_alugueis, enquanto o grafo corresponde a registo_node (vértices) e registo_adjacente (adjacência de cada vértice).

registo_clientes

```
typedef struct registo_clientes
{
    int id;
    char nome[TAM];
    float saldo;
    char morada[TAM];
    int gestor; // 1=gestor, resto=cliente

    struct registo_clientes* seguinte;
} Clientes;
```

registo_veiculos

```
typedef struct registo_veiculos
{
    int id;
    char tipo[TAM];
    float bateria;
    float autonomia;
    char geocode[TAM];

    struct registo_veiculos* seguinte;
} Veiculos;
```

registo_alugueis

```c
typedef struct registo_alugueis
{
    int id;
    int ini_dia; // (data inicio: dias)
    int ini_mes; // (data inicio: mes)
    int ini_ano; // (data inicio: ano)
    int fim_dia; // (data fim: dias)
    int fim_mes; // (data fim: mes)
    int fim_ano; // (data fim: ano)
    float valor; // (valor pago em euros)
    int id_veiculos; // (id do veiculo ligado ao aluguer)
    int id_utilizador; // (id do utilizador ligado ao aluguer)

    struct registo_alugueis* seguinte;
} Alugueis;
```

registo_node e registo_adjacente

```c
typedef struct registo_adjacente* Adjacente;
typedef struct registo_node* Node;

struct registo_adjacente
{
    Node PtDestino;
    int peso;

    Adjacente seguinte;
};

struct registo_node
{
    int ID;
    char* Cidade;
    float Latitude;
    float Longitude;

    Adjacente PtAdjacente;
    Node seguinte;
};
```

- **includes.h**

  Contém unicamente a chamada da primeira linha de cada função inserida nos outros documentos, para assim efetuar a ligação deste conjunto de comandos para o ficheiro principal denominado de “main.c”.

- **main.c**

  Ficheiro principal onde são chamadas todas outras funções para o utilizador. Contém menus interativos para os Clientes e Gestores. Acompanhado disto, faz-se durante a inicialização do programa a leitura de todos ficheiros de texto para testes, e no

caso dos Clientes leitura dos ficheiros binários para uso.

- **veiculos.c, clientes.c e aluguer.c**

  Dada a maioria das funções semelhantes em cada documento, será listado um único exemplo proveniente do ficheiro "clientes.c", recorrendo-se aos outros apenas em caso de necessidade (funcionalidades exclusivas).

  Este conjunto de comandos está voltado para o manuseio de listas a partir das estruturas "registo_clientes", "registo_veiculos" e "registo_alugueis":

ExisteCliente (ExisteVeiculo e ExisteAluguer)

```
/*
    Verificar se um cliente existe pelo seu id
*/
int ExisteCliente(Clientes* Inicio, int id)
{
    while (Inicio != NULL) {
        if (Inicio->id == id) return(1);
        Inicio = Inicio->seguinte;
    }
    return(0);
}
```

Pede pela variável inteira "id", e a partir dela, percorre a lista encadeada correspondente ao seu tipo, retornando 1 caso exista e 0 no caso de não conter um id com este dado.

InserirCliente (InserirVeiculo e InserirAluguer)

```c
/*
    Inserir clientes
*/
Clientes* InserirCliente(Clientes * Inicio, int id, char nome[], float saldo, char morada[], int gestor)
{
    if (ExisteCliente(Inicio, id) == 0)
    {
        Clientes* novo = malloc(sizeof(Clientes));
        if (novo != NULL)
        {
            novo->id = id;
            strcpy(novo->nome, nome);
            novo->saldo = saldo;
            strcpy(novo->morada, morada);
            novo->gestor = gestor;

            novo->seguinte = Inicio;
            return(novo);
        }
    }
    else return(Inicio);
}
```

Função utilizada para inserir um novo dado na lista, não o fazendo caso o ”id” correspondente ao dado conflitua com o novo pedido (já exista um dado com este valor).

Esta função aloca um novo espaço de memória utilizando o comando ”malloc”, obtendo assim novos dados pedidos pela função e por fim atualizando a lista a partir do ”novo->seguinte”, apontando os novos dados para o início da lista.

ListarCliente (ListarVeiculo e ListarAluguer)

```c
/*
    Listar clientes
*/
void ListarClientes(Clientes* Inicio)
{
    while (Inicio != NULL)
    {
        printf("%d %s %.2f %s %d\n", Inicio->id, Inicio->nome, Inicio->saldo, Inicio->morada, Inicio->gestor);
        Inicio = Inicio->seguinte;
    }
}
```

Percorre todos dados de uma lista e mostra todos os valores para o utilizador.

RemoverCliente (RemoverVeiculo e RemoverAluguer)

```c
/*
    Remover um cliente
*/
Clientes* RemoverCliente(Clientes* Inicio, int id)
{
    Clientes* anterior = Inicio, * atual = Inicio, * aux;
    if (atual == NULL) return(NULL);
    else if (atual->id == id) //remoção do 1º registo
    {
        aux = atual->seguinte;
        free(atual);
        return(aux);
    }
    else {
        while ((atual != NULL) && (atual->id != id)) {
            anterior = atual;
            atual = atual->seguinte;
        }
        if (atual == NULL) return(Inicio);
        else {
            anterior->seguinte = atual->seguinte;
            free(atual);
            return(Inicio);
        }
    }
}
```

Remove um dado da lista encadeada a partir do ”id” fornecido pelo utilizador.

A partir da chamada dos três ponteiros ”anterior”, ”atual” e ”aux” começa-se por (no caso de não estiver vazio a lista), verificar se o primeiro registo corresponde ao ”id” pedido, removendo o 1º registo e encerrando a função.

Caso o conjunto de dados não corresponda ao primeiro registo, é feito uma busca dentro de um ciclo while. Durante cada ciclo do loop, o ponteiro ”anterior” é atualizado para corresponder aos dados atuais enquanto o ponteiro ”atual” é atualizado para apontar para o próximo registo do cliente.

Após o término do ciclo while, verifica-se se o ponteiro ”atual” é nulo, se for nulo indica que o ”id” fornecido não corresponde a um valor da lista, retornando assim os dados da forma que foram chamados e encerrando a função. Caso isto não ocorra o ponteiro ”seguinte” do apontador ”anterior” para pular o dado no apontador ”atual” a ser removido, liberando espaço da memória dos dados e por fim encerrando a função.

EditarCliente (EditarVeiculo e EditarAluguer)

```c
/*
    Editar dados de um cliente
*/
Clientes* EditarCliente(Clientes* Inicio, int id, char nome[40], float saldo, char morada[40], int gestor)
{
    if (ExisteCliente(Inicio, id) == 1)
    {
        while (Inicio != NULL)
        {
            if (Inicio->id == id)
            {
                strcpy(Inicio->nome, nome);
                Inicio->saldo = saldo;
                strcpy(Inicio->morada, morada);
                Inicio->gestor = gestor;
            }
            Inicio = Inicio->seguinte;
        }
        return(Inicio);
    }
    else
    printf("Cliente nao encontrado!\n");
}
```

Função que permite editar um dos dados de um lista. Percorre-se a lista por um ciclo While e verifica-se o ”id” correspondente ao pedido pela função, alterando assim os dados do mesmo. A função antes da veririfcação faz uma chamada da função ”ExisteCliente” para verificar se há um dado correspondente ao ”id” pedido.

QuantClientes (QuantVeiculos e QuantAlugueis)

```c
/*
    Devolver número de clientes
*/
int QuantClientes(Clientes* Inicio)
{
    Clientes* aux = Inicio;
    int quant = 0;
    if (aux == NULL) return(0);
    else {
        while (aux != NULL) {
            quant++;
            aux = aux->seguinte;
        }
    }
    return(quant);
}
```

A partir de um ciclo while e um contador, percorre-se toda a lista e retorna no fim o

numéro de ciclos percorridos, correspondente ao número de dados na lista.

TrocaDados e DecrescenteVeiculos (Exclusivo de ”veiculos.c”)

TrocaDados

```c
/*
    Funcao complementar para trocar valores de duas "lista veiculos" diferentes
*/
void TrocaDados(Veiculos* a, Veiculos* b)
{
    int temp_id = a->id;
    char temp_tipo[50];
    strcpy(temp_tipo, a->tipo);
    float temp_bateria = a->bateria;
    float temp_autonomia = a->autonomia;
    char temp_geocode[50];
    strcpy(temp_geocode, a->geocode);

    a->id = b->id;
    strcpy(a->tipo, b->tipo);
    a->bateria = b->bateria;
    a->autonomia = b->autonomia;
    strcpy(a->geocode, b->geocode);

    b->id = temp_id;
    strcpy(b->tipo, temp_tipo);
    b->bateria = temp_bateria;
    b->autonomia = temp_autonomia;
    strcpy(b->geocode, temp_geocode);
}
```

Função complementar da ”DecrescenteVeiculos”, aponta dois valores diferentes e efetua no retorno uma troca dos valores de um para o outro.

DecrescenteVeiculos

```c
/*
    Funcao para veiculos por autonomia em ordem decrescente
*/
void DescrescenteVeiculos(Veiculos* Inicio)
{
    Veiculos* atual = Inicio;
    Veiculos* temp = NULL;
    float temp_autonomia = 0;

    // criar lista temporária dos veículos
    while (atual != NULL)
    {
        Veiculos* novo_veiculo = (Veiculos*)malloc(sizeof(Veiculos));
        novo_veiculo->id = atual->id;
        strcpy(novo_veiculo->tipo, atual->tipo);
        novo_veiculo->bateria = atual->bateria;
        novo_veiculo->autonomia = atual->autonomia;
        strcpy(novo_veiculo->geocode, atual->geocode);
        novo_veiculo->seguinte = temp;
        temp = novo_veiculo;

        atual = atual->seguinte;
    }

    // ordenar lista temporária por ordem decrescente de autonomia
    for (Veiculos* novo = temp; novo != NULL; novo = novo->seguinte)
    {
        for (Veiculos* j = novo->seguinte; j != NULL; j = j->seguinte)
        {
            if (j->autonomia > novo->autonomia)
            {
                TrocaDados(novo, j);
            }
        }
    }

    // imprimir na tela as informações de cada veículo
    for (Veiculos* novo = temp; novo != NULL; novo = novo->seguinte)
    {
        printf("%d %s %.2f %.2f %s\n", novo->id, novo->tipo, novo->bateria, novo->autonomia, novo->geocode);
    }

    // liberar memória alocada para a lista temporária
    while (temp != NULL)
    {
        Veiculos* proximo = temp->seguinte;
        free(temp);
        temp = proximo;
    }
}
```

Devolve para o utilizador uma nova lista ordenada de forma decrescente a partir da variável ”autonomia”.

A partir do ponteiro ”atual”, percorre-se toda a lista para a criação de uma lista temporária a partir do ponteiro ”novo”. Logo após isso, ordena-se o ponteiro ”novo” conforme o exigido no enunciado, trocando os dados da função internamente com a função auxiliar ”TrocaDados” no caso de necessidade.

Por fim mostra-se a nova lista ordenada e libera-se a memória da lista temporária.

GeocodigoVeiculos (Exclusivo de ”veiculos.c”)

```c
/*
    Listar veículos por geocode
*/
void GeocodigoVeiculos(Veiculos* Inicio, char geocode[])
{
    Veiculos* atual = Inicio;
    int quant = 0;

    while (atual != NULL) {
        if (strcmp(atual->geocode, geocode) == 0) {
            printf("%d %s %.2f %.2f %s\n", atual->id, atual->tipo, atual->bateria, atual->autonomia, atual->geocode);
            quant = 1;
        }

        atual = atual->seguinte;
    }

    if (!quant) {
        printf("Nenhum veiculo encontrado com o geocode %s!\n", geocode);
    }
}
```

Função responsável por listar os veículos com a variável ”geocode” correspondente ao inserido na função.

O seu funcionamento consiste em percorrer toda a lista dentro de um ciclo while, retornando apenas os dados com o geocódigo correspondente. Caso não se encontre nenhuma valor, o programa retornar erro por meio da variável ”quant”.

VeiculoAlugueis(Exclusivo de ”aluguer.c”)

```c
/*
    Registo do aluguer de um determinado veiculo
*/
int VeiculoAlugueis(Alugueis* Inicio, int id_veiculos)
{
    Alugueis* atual = Inicio;
    while(atual != NULL)
    {
        if(atual->id_veiculos == id_veiculos)
        {
            printf("%d %d %d %d %d %d %.2f %d %d\n", atual->id, atual->ini_dia,
        atual->ini_mes, atual->ini_ano, atual->fim_dia, atual->fim_mes,
        atual->fim_ano, atual->valor, atual->id_veiculos, atual->id_utilizador);
        }
        atual = atual->seguinte;
    }
}
```

Através de um ciclo repetitivo, retorna todos valores da lista ”Aluguer” cuja variável

”id_veiculos” corresponde ao inserido dentro da função.

LerClientes (LerVeiculos e LerAlugueis)

```c
/*
    Ler ficheiro ".txt" dos clientes
*/
Clientes* LerClientes()
{
    FILE* fp;
    int id, gestor;
    char nome[40], morada[40];
    float saldo;

    Clientes* aux = NULL;
    fp = fopen("clientes.txt", "r");
    if (fp != NULL)
    {
        while (!feof(fp))
        {
            fscanf(fp, "%d;%[^;];%f;%[^;];%d\n", &id, nome, &saldo, morada, &gestor);
            aux = InserirCliente(aux, id, nome, saldo, morada, gestor);
        }
        fclose(fp);
    }
    return(aux);
}
```

Função responsável por ler os dados contidos no ficheiro de texto ”clientes.txt” (OU veiculos.txt ou aluguer.txt). Após uma verificação o programa percorre todas as linhas do ficheiro, utilizando logo em seguida a função ”InserirCliente” (OU InserirVeiculo ou InserirAluguer) para armazenar na lista os novos dados obtidos por meio do ponteiro ”aux”.

LerClientesBin (OU LerVeiculosBin ou LerAlugueisBin)

```c
/*
    Ler ficheiro ".bin" dos clientes
*/
Clientes* LerClientesBin()
{
    FILE* fp;
    int id, gestor;
    char nome[40], morada[40];
    float saldo;

    Clientes* aux=NULL;
    fp = fopen("clientes.bin","rb"); // "rb"  read binary
    if (fp!=NULL)
    {
        while (!feof(fp))
        {
            fread(&id, sizeof(int), 1, fp);
            fread(nome, sizeof(char), 20, fp);
            fread(&saldo, sizeof(float), 1, fp);
            fread(morada, sizeof(char), 20, fp);
            fread(&gestor, sizeof(int), 1, fp);
            aux = InserirCliente(aux, id, nome, saldo, morada, gestor);
        }
        fclose(fp);
    }
    return(aux);
}
```

Função responsável por ler os dados do ficheiro binário ”clientes.bin” (OU veiculos.bin ou aluguer.bin). Possui uma estrutura semelhante a função LerClientes, apenas com diferenciais na função ”fopen” e no modo que os dados são lidos a partir do ”fread” no lugar do ”fscanf”.

guardarClientesBin (OU guardarVeiculosBin ou guardarAlugueisBin)

```c
/*
    Guardar clientes em um ficheiro ".bin"
*/
int guardarClientesBin(Clientes* Inicio)
{
    FILE* fp;
    fp = fopen("clientes.bin","wb");  // "wb" write binary

    if (fp!=NULL)
    {
        Clientes* aux= Inicio; // aux referencia o início da lista ligada
        while (aux != NULL)
        {
            fwrite(&aux->id, sizeof(int), 1, fp);
            fwrite(aux->nome, sizeof(char), 20, fp);
            fwrite(&aux->saldo, sizeof(float), 1, fp);
            fwrite(&aux->morada, sizeof(char), 20, fp);
            fwrite(&aux->gestor, sizeof(int), 1, fp);
            aux = aux->seguinte;
        }
        fclose(fp);
        return(1);
    }
    else return(0);
}
```

Responsável por armazenar os dados da lista em um ficheiro binário. Para isto é chamado o caminho do ficheiro, e logo após isso escreve-se todos os dados linha por linha a partir do ponteiro ”aux”, armazenando assim cada linha após o término do ciclo.

- **grafos.c**

  Ficheiro que contém o código do manuseio do grafo, possui funções de leitura e manipulação destes dados envolvendo ficheiros, além da distinção desses dados a partir de constantes complexas como localização (a partir do peso atribuído a cada adjacência ou a latitude e longitude de cada vértice).

  Este conjunto de comandos interage diretamente com as estruturas “registo_node” e “registo_adjacente” correspondendo aos vértices e as suas respetivas adjacências.

  InserirNode (Inserir Vértice)

```c
/*
    Inserir nodes (Vértices)
*/
Node InserirNode(int ID, char* Cidade, float Latitude, float Longitude)
{
    Node novoNode = (Node)malloc(sizeof(struct registo_node));
    novoNode->ID = ID;
    novoNode->Cidade = strdup(Cidade);
    novoNode->Latitude = Latitude;
    novoNode->Longitude = Longitude;
    novoNode->PtAdjacente = NULL;
    novoNode->seguinte = NULL;
    return novoNode;
}
```

Responsável por inserir um novo dado a estrutura ”registo_node” correspondente aos vértices após a leitura dos dados inseridos pelo utilizador. Para isso é criado um novo dado a partir da função ”malloc” que aloca um novo espaço na memória, contido dentro ponteiro ”novoNode” para retornar no fim a lista atualizada.

CriarAdjacente

```c
/*
    Criar adjacencias
    (Funcao auxeiliar para "AdicinarAdjacente")
*/
Adjacente CriarAdjacente(Node PtDestino, int peso)
{
    Adjacente novoAdjacente = (Adjacente)malloc(sizeof(struct registo_adjacente));
    novoAdjacente->PtDestino = PtDestino;
    novoAdjacente->peso = peso;
    novoAdjacente->seguinte = NULL;
    return novoAdjacente;
}
```

Semelhante a função InserirNode, aloca um novo espaço na memória e a partir de um ponteiro atualiza a lista inteira.

InserirAdjacente

```c
/*
    Inserir adjacencias
*/
void InserirAdjacente(Node node, Node adjacente, int peso)
{
    Adjacente novoAdjacente = CriarAdjacente(adjacente, peso);

    if (node->PtAdjacente == NULL)
    {
        node->PtAdjacente = novoAdjacente;
    } else
    {
        Adjacente ultimoAdjacente = node->PtAdjacente;
        while (ultimoAdjacente->seguinte != NULL)
        {
            ultimoAdjacente = ultimoAdjacente->seguinte;
        }
        ultimoAdjacente->seguinte = novoAdjacente;
    }
}
```

Função responsável por estabelecer uma adjacência a um ”Node” (vértice) específico do grafo dado que cada vértice contém um lista de adjacências dentro de si.

Recorrendo a função ”CriarAdjacente” para criar a noa adjacência, armazenando os dados em ”novoAdjacente”. Após isso faz-se uma verificação na lista de adjacências do ”Node”, pondo diretamente o dado no começo caso o mesmo esteja vazio, ou percorrendo a lista inteira pelo ciclo while até se chegar ao último adjacente e com isso colocar o novo valor.

ListarGrafos

```c
/*
    Listar grafo
*/
void ListarGrafos(Node grafo)
{
    Node node = grafo;
    while (node != NULL)
    {
        printf("Cidade: %s (%.2f,%.2f)\n", node->Cidade, node->Latitude, node->Longitude);

        Adjacente adjacente = node->PtAdjacente;
        while (adjacente != NULL)
        {
            printf("  Adjacente: %s, Peso: %d\n", adjacente->PtDestino->Cidade, adjacente->peso);
            adjacente = adjacente->seguinte;
        }

        node = node->seguinte;
    }
}
```

Lista todos dados do grafo desde os ”Nodes” (vértices) até suas respetivas adjacências por meio de um ciclo while.

LerVertices

```c
/*
    Ler ficheiro ".txt" dos Nodes (Vértices)
*/
void LerVertices(Node* grafo)
{
    FILE* arquivo = fopen("vertices.txt", "r");
    if (arquivo == NULL) {
        printf("Erro ao abrir o arquivo de vertices.\n");
        return;
    }

    int ID;
    char Cidade[100];
    float Latitude, Longitude;
    Node novoNode, ultimoNode = NULL;

    while (fscanf(arquivo, "%d %s %f %f", &ID, Cidade, &Latitude, &Longitude) == 4) {
        novoNode = InserirNode(ID, Cidade, Latitude, Longitude);

        if (*grafo == NULL) {
            *grafo = novoNode;
        } else {
            ultimoNode->seguinte = novoNode;
        }
        ultimoNode = novoNode;
    }

    fclose(arquivo);
}
```

Função responsável por ler os vértices armazenados no ficheiro de texto ”vertices.txt”. Este processo é feito logo após a definição do caminho do ficheiro, lendo todos os dados a partir da função fscanf em um ciclo while que consequentemente retorna os dados com o término do seu percurso.

LerVerticesBin

```
    Ler ficheiro ".bin" dos nodes (Vértices)
*/
void LerVerticesBin(Node* grafo)
{
    FILE* arquivo = fopen("vertices.bin", "rb");
    if (arquivo == NULL) {
        printf("Erro ao abrir o arquivo de vertices.\n");
        return;
    }

    int ID;
    char Cidade[100];
    float Latitude, Longitude;
    Node novoNode, ultimoNode = NULL;

    while (fread(&ID, sizeof(int), 1, arquivo) == 1) {
        fread(Cidade, sizeof(char), 100, arquivo);
        fread(&Latitude, sizeof(float), 1, arquivo);
        fread(&Longitude, sizeof(float), 1, arquivo);

        novoNode = InserirNode(ID, Cidade, Latitude, Longitude);

        if (*grafo == NULL) {
            *grafo = novoNode;
        } else {
            ultimoNode->seguinte = novoNode;
        }
        ultimoNode = novoNode;
    }

    fclose(arquivo);
}
```

Similarmente a função ”LerVertices” lê-se um ficheiro para se obter os dados, so que nesse caso trata-se de um ficheiro binário. A estrturução assemelha-se também ao exemplo anterior com pequenas exceções na chamada do ”fopen” e na leitura dos dados com o ”fread".

LerAdjacencias

```c
    Ler ficheiro ".txt" das Adjacencias
*/
void LerAdjacencias(Node grafo)
{
    FILE* arquivo = fopen("adjacencias.txt", "r");
    if (arquivo == NULL) {
        printf("Erro ao abrir o arquivo de adjacencias.\n");
        return;
    }

    int IDOrigem, IDAdjacente, peso;

    while (fscanf(arquivo, "%d %d %d", &IDOrigem, &IDAdjacente, &peso) == 3) {
        Node node = grafo;
        Node adjacente = NULL;

        while (node != NULL) {
            if (node->ID == IDOrigem) {
                adjacente = node;
                break;
            }
            node = node->seguinte;
        }

        if (adjacente != NULL) {
            Node nodeAdjacente = grafo;

            while (nodeAdjacente != NULL) {
                if (nodeAdjacente->ID == IDAdjacente) {
                    InserirAdjacente(adjacente, nodeAdjacente, peso);
                    break;
                }
                nodeAdjacente = nodeAdjacente->seguinte;
            }

            if (nodeAdjacente == NULL) {
                printf("Cidade adjacente nao encontrada.\n");
            }
        } else {
            printf("Cidade de origem nao encontrada.\n");
        }
    }

    fclose(arquivo);
}
```

Função que lê o ficheiro ”adjacencias.txt”, retornando as adjacencias correspondentes a um ”Node” (Vértice).

Para fazer isto, a função após a uma verificação, percorre primeiramente todos ”Nodes” até encontrar o mesmo com o ”id” correspondente a adjacência. Com este node encontrado, percorre-se mais a frente um novo ciclo para atualizar a lista das adjacências acoplada a este vértice (Para isto, faz-se uso da função ”InserirAdjacente”).

LerAdjacenciasBin

```c
/*
    Ler ficheiro ".bin" das adjacencias
*/
void LerAdjacenciasBin(Node grafo)
{
    FILE* arquivo = fopen("adjacencias.bin", "rb");
    if (arquivo == NULL) {
        printf("Erro ao abrir o arquivo de adjacencias.\n");
        return;
    }

    int IDOrigem, IDAdjacente, peso;

    while (fread(&IDOrigem, sizeof(int), 1, arquivo) == 1) {
        fread(&IDAdjacente, sizeof(int), 1, arquivo);
        fread(&peso, sizeof(int), 1, arquivo);

        Node node = grafo;
        Node adjacente = NULL;

        while (node != NULL) {
            if (node->ID == IDOrigem) {
                adjacente = node;
                break;
            }
            node = node->seguinte;
        }

        if (adjacente != NULL) {
            Node nodeAdjacente = grafo;

            while (nodeAdjacente != NULL) {
                if (nodeAdjacente->ID == IDAdjacente) {
                    InserirAdjacente(adjacente, nodeAdjacente, peso);
                    break;
                }
                nodeAdjacente = nodeAdjacente->seguinte;
            }

            if (nodeAdjacente == NULL) {
                printf("Cidade adjacente nao encontrada.\n");
            }
        } else {
            printf("Cidade de origem nao encontrada.\n");
        }
    }
```

Função que lê o ficheiro binário ”adjacencias.bin”. Desempenha um papel semelhante ao ”LerAdjacencias” com exceção na chamada do caminho do ficheiro e na substituição do ”fprintf” pelo ”fread”.

GuardarVerticesBin

```c
/*
    Guardar nodes (Vertices) em um ficheiro ".bin"
*/
void guardarVerticesBin(Node grafo)
{
    FILE* arquivo = fopen("vertices.bin", "wb");
    if (arquivo == NULL) {
        printf("Erro ao abrir o arquivo de vertices.\n");
        return;
    }

    Node node = grafo;
    while (node != NULL) {
        fwrite(&node->ID, sizeof(int), 1, arquivo);
        fwrite(node->Cidade, sizeof(char), 100, arquivo);
        fwrite(&node->Latitude, sizeof(float), 1, arquivo);
        fwrite(&node->Longitude, sizeof(float), 1, arquivo);

        node = node->seguinte;
    }

    fclose(arquivo);
}
```

Função responsabilizada por guardar os ”Nodes” (Vértices) dentro de um ficheiro binário denominado de ”vertices.bin”. Para efetuar esta tarefa lê-se o caminho e tipo do ficheiro, apenas para depois dentro de um ciclo while percorrer toda lista até chegar ao local para armazenar o respetivo dado atualizado.

GuardarAdjacenciasBin

```c
/*
    Guardar adjacencias em um ficheiro ".bin"
*/
void guardarAdjacenciasBin(Node grafo)
{
    FILE* arquivo = fopen("adjacencias.bin", "wb");
    if (arquivo == NULL) {
        printf("Erro ao abrir o arquivo de adjacencias.\n");
        return;
    }

    Node node = grafo;
    while (node != NULL) {
        Adjacente adjacente = node->PtAdjacente;
        while (adjacente != NULL) {
            fwrite(&node->ID, sizeof(int), 1, arquivo);
            fwrite(&adjacente->PtDestino->ID, sizeof(int), 1, arquivo);
            fwrite(&adjacente->peso, sizeof(int), 1, arquivo);
            adjacente = adjacente->seguinte;
        }
        node = node->seguinte;
    }

    fclose(arquivo);
}
```

Guarda os dados das adjacências dos grafos dentro do ficheiro ”adjacencias.bin”. Após definir o caminho e o tipo do ficheiro, o programa percorre dois ciclos while, um para encontrar cada node e outro para armazenar no fim de cada lista a adjacência correspondente ao Node (Vértice).

listaVeiculosRaio

```c
/*
   Distancia dos Veiculos em um raio
*/
void listarVeiculosRaio(Node grafo, Veiculos* Inicio, float clienteLat, float clienteLon, float raio, char tipoVeiculo[])
{
    // Inicialização dos arrays estáticos
    int visitado[MAX_VERTICES] = {0};
    float distancia[MAX_VERTICES];
    Node fila[MAX_VERTICES];
    int frente = 0, tras = 0;

    int geocode;
    while (Inicio != NULL)
    {
        // Localização do vértice inicial (cliente)
        Node verticeAtual = grafo;
        while (verticeAtual != NULL)
        {
            // Verifica se há veículo associado à cidade atual
            if (verticeAtual->PtAdjacente != NULL)
            {
                // Cálculo da distância euclidiana entre o cliente e o veículo
                float lat1 = clienteLat;
                float lon1 = clienteLon;
                float lat2 = verticeAtual->Latitude;
                float lon2 = verticeAtual->Longitude;
                float distanciaAtual = sqrt(pow(lat2 - lat1, 2) + pow(lon2 - lon1, 2));

                // Verifica se o veículo está dentro do raio de busca e pertence à cidade alvo
                geocode = atoi(Inicio->geocode);
                if (!visitado[verticeAtual->ID] && distanciaAtual <= raio && verticeAtual->ID == geocode && strcmp(Inicio->tipo, tipoVeiculo) == 0)
                {
                    printf("- %s / %s (Distância: %.2f)\n", verticeAtual->Cidade, Inicio->tipo, distanciaAtual);
                    visitado[verticeAtual->ID] = 1;
                    distancia[verticeAtual->ID] = distanciaAtual;
                    fila[tras++] = verticeAtual;
                }
            }

            verticeAtual = verticeAtual->seguinte;
        }

        // BFS adaptado para calcular a distância acumulada com base em latitude e longitude
        while (frente != tras)
        {
            Node noAtual = fila[frente++];
            Adjacente adjacente = noAtual->PtAdjacente;
            while (adjacente != NULL)
            {
                Node noAdjacente = adjacente->PtDestino;

                // Verifica se há veículo associado à cidade adjacente
                if (noAdjacente->PtAdjacente != NULL) {
                    // Cálculo da distância euclidiana entre os pontos geográficos
                    float lat1 = noAtual->Latitude;
                    float lon1 = noAtual->Longitude;
                    float lat2 = noAdjacente->Latitude;
                    float lon2 = noAdjacente->Longitude;
                    float distanciaAtual = sqrt(pow(lat2 - lat1, 2) + pow(lon2 - lon1, 2));

                    // Verifica se o veículo está dentro do raio de busca e pertence à cidade alvo
                    geocode = atoi(Inicio->geocode);
                    if (!visitado[noAdjacente->ID] && distanciaAtual <= raio && noAdjacente->ID == geocode && strcmp(Inicio->tipo, tipoVeiculo) == 0) {
                        printf("- %s / %s (Distância: %.2f)\n", noAdjacente->Cidade, Inicio->tipo, distancia[noAtual->ID] + distanciaAtual);
                        visitado[noAdjacente->ID] = 1;
                        distancia[noAdjacente->ID] = distancia[noAtual->ID] + distanciaAtual;
                        fila[tras++] = noAdjacente;
                    }
                }

                adjacente = adjacente->seguinte;
            }
        }

        Inicio = Inicio->seguinte;
    }
}
```

A seguinte função visa solucionar o problema 2 da segunda fase deste projeto, problema este que consiste em listar todos veículos a partir de um raio de distância, usando como referência a localização do cliente.

Para isto, faz-se uso das variáveis contidas no ”registo_node” denominadas de ”latitude” e ”longitude”, que após definida a posição do cliente e a distância máxima, calcula a coordenada do cliente para cada veículo em um ciclo a partir de uma fórmula de cálculo da distância euclidiana.

Utilizando arrays estáticos, inicia-se o ”visitado”, ”distancia” e ”fila” visando armazenar informações pertinentes aos vértices acumulados, distâncias acumuladas e vértices e pontos a serem visitados. Dado isto, lista-se os seguintes passos:

1 - Loop que percorre todos veículos armazenados;

2 - Itera-se cada ponto para verificar se à veículos associados a cada cidade;

3 - Para cada vértice com veículos associados, calcula-se a distância euclidiana entre a

localização do cliente e a atual;

4 - Caso o veículo esteja dentro do raio exigido, é impresso o resultado.

5 - Marca-se o vértice atual como visitado armazenando o vértice e a distância acumulada nos arrays ”distancia” e ”fila”;

6 - Inicia-se uma busca em largura adaptada com base na latitude e longitude e enquanto a fila não estiver vazia percorre-se as respetivas adjacências deste vértice, repetindo o mesmo processo que havia sido feito com os ”Nodes" (marcar como visitado, calcular distâncias etc...);

7 - Com o término de todos loops (”Nodes” e suas respetivas adjacências) a função termina.

menorCaminhoVeiculos

```c
/*
    Percorrer grafo para recolher veículos com menos de 50% da bateria
*/
void menorCaminhoVeiculos(Node grafo, Veiculos* listaVeiculos, char* tipoVeiculo, int origem, int destino)
{
    int dist[MAX_VERTICES];
    int anterior[MAX_VERTICES];
    int visitado[MAX_VERTICES];
    int fila[MAX_VERTICES];
    int frente = 0, tras = 0;
    int i, geocode;

    // Inicialização
    for (i = 0; i < MAX_VERTICES; i++)
    {
        dist[i] = INT_MAX;
        anterior[i] = -1;
        visitado[i] = 0;
    }

    dist[origem] = 0;
    visitado[origem] = 1;
    fila[tras++] = origem;

    while (frente != tras)
    {
        int verticeAtual = fila[frente++];
        Node noAtual = grafo;
        while (noAtual != NULL)
        {
            if (noAtual->ID == verticeAtual)
            {
                Adjacente adjacente = noAtual->PtAdjacente;
                while (adjacente != NULL)
                {
                    Node noAdjacente = adjacente->PtDestino;
                    int pesoAdjacente = adjacente->peso;

                    if (!visitado[noAdjacente->ID])
                    {
                        visitado[noAdjacente->ID] = 1;
                        dist[noAdjacente->ID] = dist[verticeAtual] + pesoAdjacente;
                        anterior[noAdjacente->ID] = verticeAtual;
                        fila[tras++] = noAdjacente->ID;
                    } else if (dist[verticeAtual] + pesoAdjacente < dist[noAdjacente->ID])
                    {
                        dist[noAdjacente->ID] = dist[verticeAtual] + pesoAdjacente;
                        anterior[noAdjacente->ID] = verticeAtual;
                    }

                    adjacente = adjacente->seguinte;
                }
                break;
            }
            noAtual = noAtual->seguinte;
        }
    }
```

```c
    printf("\n");
    // Imprime o menor caminho
    if (dist[destino] == INT_MAX)
    {
        printf("Não há caminho entre os nós %d e %d.\n", origem, destino);
    } else
    {
        printf("Menor caminho entre os nós %d e %d:\n", origem, destino);
        int caminho[MAX_VERTICES];
        int tamanhoCaminho = 0;
        int no = destino;
        while (no != -1)
        {
            caminho[tamanhoCaminho++] = no;
            no = anterior[no];
        }

        printf("Percurso com veículos do tipo '%s': ", tipoVeiculo);
        for (i = tamanhoCaminho - 1; i >= 0; i--)
        {
            Node noAtual = grafo;
            while (noAtual != NULL)
            {
                if (noAtual->ID == caminho[i])
                {
                    Veiculos* veiculoAtual = listaVeiculos;
                    while (veiculoAtual != NULL)
                    {
                        geocode = atoi(veiculoAtual->geocode);
                        // verifica e retorna se há veículo no vértice
                        if (geocode == noAtual->ID && strcmp(veiculoAtual->tipo, tipoVeiculo) == 0)
                        {
                            printf("%d ", caminho[i]);
                            if (veiculoAtual->bateria < 50)
                            {
                                printf("(%s) ", veiculoAtual->tipo);
                            }
                            break;
                        }
                        veiculoAtual = veiculoAtual->seguinte;
                    }
                    break;
                }
                noAtual = noAtual->seguinte;
            }
        }
        printf("\n");
        printf("Distância total: %d\n", dist[destino]);
    }
}
```

Função feita para desenvolver o problema 3 da segunda fase. Dado isto, encontra-se o menor caminho de dois pontos inseridos pelo utilizador, retornando através do percurso apenas os veículos de um determinado tipo e com bateria abaixo de 50%.

Este função recorre o algoritmo de Dijkstra para obter o menor caminho entre dois pontos

de um grafo e regista as distâncias acumuladas e dos vértices anteriores a medida que percorre o grafo. Apenas para no final imprimir os dados desejados.

Dado isto, a função percorre os seguintes passos para obter o pretendido:

1 - Inicializa os arrays estáticos para armazenar as distâncias, vértices anteriores, visitados e a fila dos mesmos a serem processados.

2 - Loop inicial para definir os valores iniciais dos arrays;

3 - Define a distância de origem e a marca como visitada, adicionando-a a fila logo em seguida;

4 - Enquanto a fila não estiver vazia, um vértice é retirado da fila e suas adjacências são exploradas;

5 - Para cada vértice adjacente ainda não visitado é atualizada a distância acumulada e o dado anterior a partir dos ”pesos” percorridos de cada adjacência;

6 - Se o vértice adjacente já foi visitado, mais uma distância menor foi encontrada, a distância é atualizada e o dado anterior também é atualizado;

7 - Percorre todo o processo repetidamente até que todos os vértices sejam visitados, imprimindo no fim o menor caminho ou devolvendo uma mensagem de erro caso não seja possível chegar do ponto de início ao fim;

8 - A impressão é feita baseado nas condições do enunciado, listando juntamente dos vértices os veículos de um tipo com bateria abaixo de 50%.

- **Ficheiros de texto (.txt) e binários (.bin)**

  Pertinente aos ficheiros de texto, ressalta-se a forma que os mesmos devem ser escritos para leitura dos dados:

  clientes.txt, veiculos.txt e aluguer.txt

```
1;1;1.000000;1.000000;1
2;2;2.000000;2.000000;2
4;4;4.000000;4.000000;4
```

  Dados divididos em ”;” onde o primeiro corresponde ao ”id” e o restante às variáveis, baseado na ordem que se é definida na própria estrutura desta lista. Cada dado é dividido em linhas diferentes.

vertices.txt

```
1 Cidade1 10 20
2 Cidade2 15 25
3 Cidade3 12 18
```

Dados divididos em espaços em branco, cada linha contém um dado diferente. Os ”Nodes” (Vétices) são distinguidos pelo primeiro valor, correspondente ao ”id”. O segundo campo corresponde ao nome e o terceiro e quarto a latitude e longitude.

adjacencias.txt

```
1 2 15
2 3 5
2 1 10
3 2 10
```

Dados divididos em espaços em branco onde cada linha contém um dado diferente. O primeiro dado corresponde ao ”id”, o segundo ao vértice que esta adjacência pertence e o campo final ao peso atribuído a esta adjacência.

- **utilizador.c**

Documento cujas funções são unicamente voltadas para a interface do cliente, ou versões das funções anteriores adaptadas à necessidade do cliente. Inclui funções de manuseio do saldo, dos alugueres etc...

VerificaUtilizador

```c
/*
    Verificar se o Utilizador é Cliente ou Gestor
*/
int VerificaUtilizador(Clientes* Inicio, int id)
{
    Clientes* atual = Inicio;

    while (atual != NULL) {
        if (atual->id == id && atual->gestor == 1)
        {
            printf("Logado como Gestor!\n\n");
            return -1;
        }
        else if (atual->id == id)
        {
            printf("Logado como Cliente!\n\n");
            return atual->id;
        }

        atual = atual->seguinte;
    }
}
```

Verifica se o id inserido corresponde ao de um utilizador ou de um gestor, retornando o valor do tipo em inteiro (-1 = gestor, resto=cliente). Percorre-se a lista inteira dos clientes e compara-se o id inserido e se a variável ”gestor” está ou não igualada a 1 (caso esteja, é gestor).

LerSaldo

```c
/*
    Ler saldo de um Cliente
*/
int LerSaldo(Clientes* Inicio, int id)
{
    Clientes* atual = Inicio;

    while (atual != NULL)
    {
        if (atual->id == id)
            return atual->saldo;
        atual = atual->seguinte;
    }
}
```

Função que retorna o saldo do utilizador ao percorrer a lista dos clientes e obter a linha com o ”id” correspondente.

DepositarQuantia

```
/*
    Depositar quantia no saldo de um Cliente
*/
Clientes* DepositarQuantia(Clientes* Inicio, int id, float saldo)
{
    if (ExisteCliente(Inicio, id) == 1)
    {
        while (Inicio != NULL)
        {
            if (Inicio->id == id)
            {
                Inicio->saldo = saldo+Inicio->saldo;
            }
            Inicio = Inicio->seguinte;
        }
        return(Inicio);
    }
    else
    printf("Erro ao depositar quantia!\n");
}
```

A verificar se o “id” do cliente existe, e percorrer a lista dos clientes, altera-se apenas o saldo desta linha somando o valor atual com o novo.

RemoverQuantia

```c
/*
    Depositar quantia no saldo de um Cliente
*/
Clientes* RemoverQuantia(Clientes* Inicio, int id, float saldo)
{
    if (ExisteCliente(Inicio, id) == 1)
    {
        while (Inicio != NULL)
        {
            if (Inicio->id == id)
            {
                Inicio->saldo = Inicio->saldo-saldo;
            }
            Inicio = Inicio->seguinte;
        }
        printf("Aluguer removido!\n");
        return(Inicio);
    }
    else
    printf("Erro ao remover quantia!\n");
}
```

Função que remove determinada quantia do saldo de um utilizador. Semelhante a função ”DepositarQuantia”, valida-se o ”id” do cliente e depois percorre toda lista dos Clientes até encontrar ao correspondente, subtraindo o valor inserido com o atual.

ListarAluguerCliente

```c
/*
    Listar aluguer de um Cliente
*/
int ListarAlugueisCliente(Alugueis* Inicio, int id)
{
    Alugueis* atual = Inicio;
    int cont=0;

    while (atual != NULL)
    {
        if (atual->id_utilizador == id)
        {
            printf("%d %d %d %d %d %d %d %.2f %d %d\n", atual->id, atual->ini_dia, atual->ini_mes, atual->ini_ano, atual->fim_dia, atual
        }
        cont++;
        atual = atual->seguinte;
    }
    if (cont == 0)
    {
        return 0;
    }
    else return 1;
}
```

Função que lista os dados dos Alugueres, mas que retorna apenas as linhas onde o ”id_utilizador” corresponde ao do Cliente.

removerAluguerCliente

```c
/*
    Remover Aluguer de um Cliente
*/
Alugueis* removerAluguerCliente(Alugueis* Inicio, int id, int id_utilizador)
{
    Alugueis* anterior = Inicio, * atual = Inicio, * aux;
    if (atual == NULL) return(NULL);
    else if (atual->id == id && atual->id_utilizador == id_utilizador) //remoção do 1º registo
    {
        aux = atual->seguinte;
        free(atual);
        return(aux);
    }
    else
    {
        while ((atual != NULL) && (atual->id != id))
        {
            anterior = atual;
            atual = atual->seguinte;
        }
        if (atual == NULL && atual->id_utilizador == id_utilizador) return(Inicio);
        else
        {
            anterior->seguinte = atual->seguinte;
            if (atual->id_utilizador == id_utilizador)
                free(atual);
            return(Inicio);
        }
    }
}
```

Função semelhante ao ”removerAluguer”, com exceção do ”id” e do ”id_utilizador” que apenas permitem a remoção de alugueres pertencentes ao Cliente.

Usabilidade

Compilação:

Dado que este projeto foi desenvolvido em ambiente Linux, será esclarecido os passos e requisitos necessários para a execução do projeto neste ambiente.

- GCC (Compilador)
- Terminal Linux

Com todos requisitos basta executar o seguinte comando dentro do diretório do projeto:

```
gcc *.c -o programa -lm
```

E depois executar o seguinte comando para executar a aplicação:

```
./programa
```

Utilização:

Logo na inicialização do programa, será pedido ao utilizador um valor correspondente ao "id" que deseja aceder. "Id" este que a partir das funções dadas irá distinguir se o sistema fará login como gestor (administrador) ou cliente (O programa insistirá que se insira um "id" válido, para sair neste caso basta pressionar CTRL+C):

O programa é divido em diferentes "interfaces" sendo uma correspondente ao gestor e outra ao cliente:

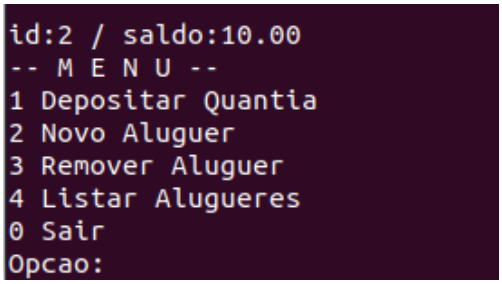
- **Interface Gestor**

A interface dos gestores contém grande parte das funções desenvolvidas neste

projeto e engloba o uso de todos ficheiros do tipo C em seus comandos, acompanhado disto, durante a inicialização do programa é lido todos dados dos ficheiros de texto.

Além da inserção dos números correspondentes às opções, as mesmas podem variar em 4 ”subinterfaces” distintas, gestão dos veículos, clientes, alugueres e dos grafos. Para alternar entre cada interface, é sempre selecionado a opção 9, ao qual disponibilizará ao gestor a opção de troca de interfaces.

- **Interface Cliente**



A interface do cliente é mais simples, não possui ”subinterfaces” e engloba uma gama limitada de comandos, unicamente voltados a gestão da quantia e gestão de alugueres. Esta versão do programa sempre mostra o saldo do cliente assim como seu ”id“, atualizando-o a cada comando inserido.

Conclusão

Este trabalho permitiu o discente alargar e aprofundar os conhecimentos obtidos, no decorrer do semestre, relativo ao desenvolvimento de soluções mais avançadas em C, assim como aplicá-las num contexto prático, desenvolvendo situações úteis em problemas de contexto real, como a própria solução desenvolvida neste projeto, relativa a gestão de dispositivos elétricos sob o contexto de um "smat-city".

Durante o desenvolvimento da primeira fase do projeto, foi possível obter e acentuar ainda mais os conceitos base em programação em C. Conhecimento estes, voltados ao manuseio da memória e a sua alocação para o desenvolvimento do programa a partir do uso de apontadores, permitindo por meio disso, desenvolver uma alternativa aos arrays para desenvolver uma alternativa às práticas de manipulação de dados e ficheiros por meio de listas existentes, com vantagens visíveis durante todo o desenvolvimento deste programa.

A versatilidade do manuseio das listas me permitiu com a segunda fase do projeto, desenvolver grafos provenientes destes mesmos dados, permitindo realizar comparações mais avançadas que envolvem constantes complexas como a própria localização de cada veículo espalhado por um ambiente. Com o uso de listas alternadas foi possível também obter e reascender conceitos estabelecidos no primeiro semestre voltado a outras unidades curriculares como a matemática discreta, fazendo uso de conceitos como o algoritmo de Dijkstra para chegar ao objetivo pretendido e fórmulas de cálculos para obter a distância não euclidiana ao pôr em causa constantes como latitude e longitude.

Concluo com este trabalho, a importância desse processo de desenvolvimento tanto para a área de programa quanto para outras áreas provenientes do curso, que juntas põe em causa a importância de seu uso em um contexto prático, como neste exemplo, o desenvolvimento de um programa para resolver problemas distintos.