Universidade De Évora
Departamento de Informática
Estruturas de Dados e Algortimos II
Ano letivo 2018 - 2019

Fly me to the moon...

Alunos:
João da Conceição - 38052
Rúben Teimas - 39868

Docente:
Vasco Pedro

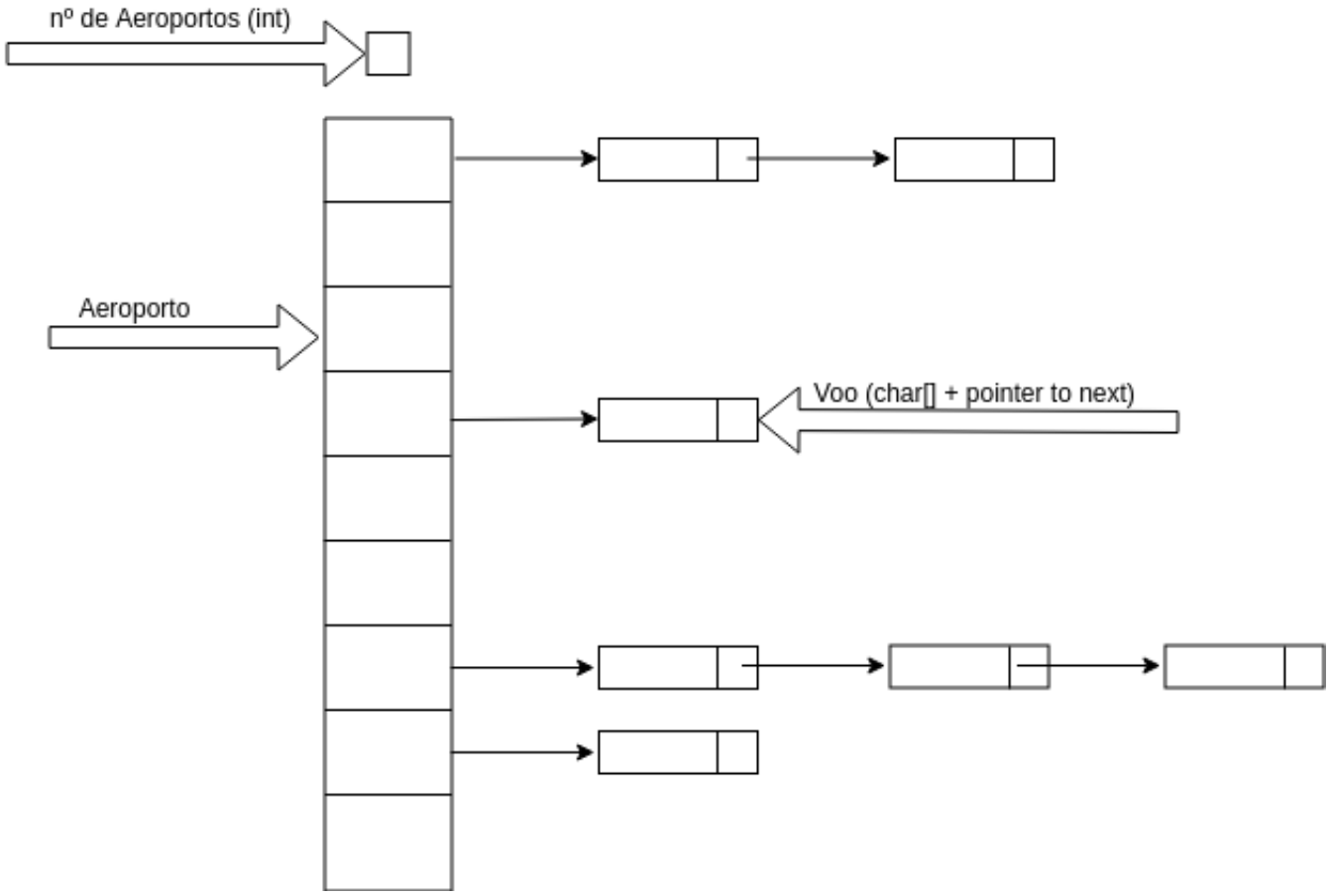
1 de Julho de 2019

1 Introdução

Este trabalho consiste na implementação das entidades Aeroportos e Voos, de modo a que fosse possivél encontrar o caminho mais curto(temporal) entre 2 Aeroportos, recorrendo aos Voos existentes.

As operações possiveis de realizar são adicionar Aeroportos(AI), adionar Voos(FI), remover Voos(FD) e procurar o caminho mais curto entre 2 Aeroportos(TR).

Para a realização do trabalho usamos uma variação de grafos na qual usamos uma hashtable em vez de um simples array.



Representação do grafo implementado

2 Estruturas Utilizadas

Para a realização deste trabalho foram usadas Hashtables, Linked-Lists e uma Priority Queue recorrendo a uma minHeap.

2.1 Hashtables

A escolha das Hashtables deve-se à sua baixa complexidade temporal, com um custo constante (*O(1)*) nas operações usadas, dado que tentamos ao máximo eliminar colisões recorrendo a uma função de hash com uma grande dispersão de dados (*djb2*).

Optámos por não usar função Re-hash por ser uma função muito cara em termos temporais. Assim, a complexidade espacial é constante (*O(1)*).

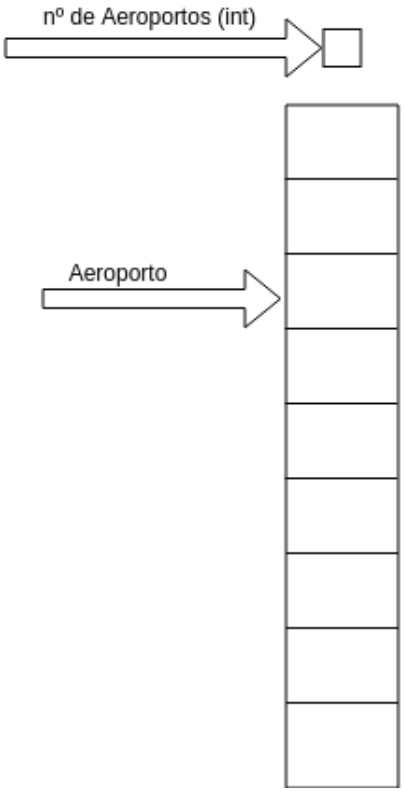
Foram definidas 2 Hashtables: a hashtable dos Voos e a hashtable dos Aeroportos. Diferem entre si no conteúdo das celulas e no número das mesmas, sendo que numa delas, as células são constituidos por Voos e na outra por Aeroportos.
De forma mais abrangente, as nossas estruturas de hashtables são constituídas por um inteiro que indica o número de células usadas e um array de células.

A inserção nas Hash Tables são feitas por procurar em primeiro lugar a posição em que se deve inserir os Aeroportos/Voos. Esta procura é feita usando a função hash (*djb2*) que retorna a posição a que deve ser inserido. Em seguida verifica se essa posição na Hash se encontra livre **(NULL)**. Se essa posição estiver livre, aloca-se o espaço necessário e coloca se na hash. Se a posição estiver a ser ocupada significa que há colisões. Decidimos tratar as colisões com double hashing e linear. A segunda função hash pega no resultado da *djb2* e faz a divisão inteira por 3. Assim procuramos a próxima posição vazia.
Sempre que encontramos uma posição não vazia, vimos que está ocupada com o elemento que queremos adicionar, garantindo que não adicionamos duplicados.

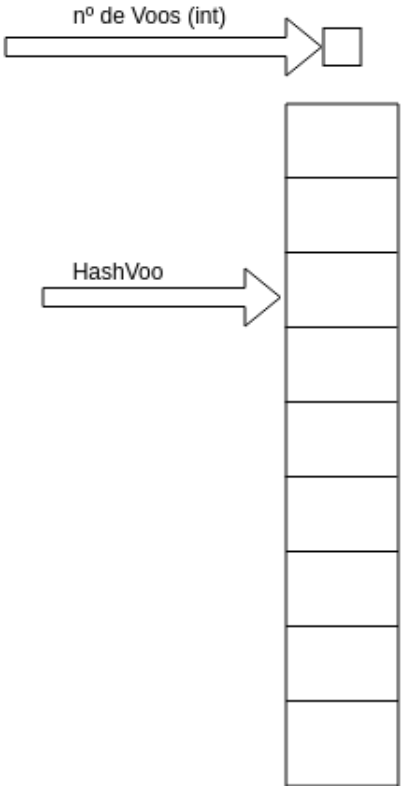
A remoção só nos é necessária nas hashTable dos Voos. Procuramos a sua posição e damos **free()** ao elemento.

Deste modo, a procura de um elemento, a sua adição e a sua remoçãp terão, normalmente, um custo temporal constante (*O(1)*), mas no pior dos casos terão complexinada temporal de *O(n)*, sendo n o número que elementos na hash.

A representação de uma hash table é feita numa **struct** que tem um **int** que representa o número de elementos e um **array** de elementos.
Nas posições em que deverão estar vazias estão a **NULL**.



HashTable Aeroportos



HashTable Voos

2.2 Linked-List

A escolha das listas teve como base o conceito de *lista de adjacências*.

A nossa lista foi implementada para ter nós Voo, sendo estes constituídos pelo código de um voo e o apontador para o próximo nó.

Quanto à complexidade temporal, a inserção na lista (***addVoo***) tem um custo de constante *O(1)* e a remoção (***delVoo***) um custo *O(n)*, sendo n o número de nós da lista. A nossa lista não tem uma função de pesquisa propriamente dita pois a procura dos Voos é efetuada na hashtable de Voos, dado que a complexidade temporal da pesquisa numa hashtable é menor do que numa linked-list. Só é realizada a procura no momento em que um nó é removido.

A inserção é feita atualizando a head da lita. Se a head estiver vazia, a head passa a ser esse novo voo. Se já houver mais voos, o novo voo aponta para a head e a head passa a ser esse novo voo. Assim, a inserção terá complexidade temporal de *O(1)*.

A remoção é feita percurrento a lista até encontrar o nó que pretendemos remover. Ao encontrar o no, o anterior passa a apontar o *next* do no que queremos remover e damos *free()* do nó removido. Assim. a remoção terá complexidade temporal no pior caso de *O(n)*, sendo **n** o número de nó existentes na lista.

Linked-List dos Voos

2.3 Priority-Queue

A Priority-Queue foi escolhida pois era necessária para a implementação do algortimo de Dijkstra.

Ao inserir, a complexidade temporal é de *O(1)*, contudo, a função que insere (***insert***) chama a função ***update***, que tem um complexidade temporal de *O(log(n))*, sendo n o número de nós na heap. Assim sendo, a função ***insert*** tem também uma complexidade *O(log(n))*.

Para remover o nó, precisamos de encontrar o que tem menor prioridade. Encontrar o nó a remover tem uma complexidade *O(1)*, pois o nó com menor peso estará sempre na front da queue. Após encontrar o nó a remover e coloca-lo num Aeroporto temporário irá chamar a função ***minHeapify*** que garante a propriedade de uma min Heap (que o peso da raíz é menor ou igual ao dos respetivos filhos).
A função ***minHeapify*** tem complexidade *O(log(n))*, sendo n o número de nós da priority-queue. Por essa razão, a função ***pop***, que remove o nó com menor peso, tem também complexidade *O(log(n))*.

3 Algoritmos

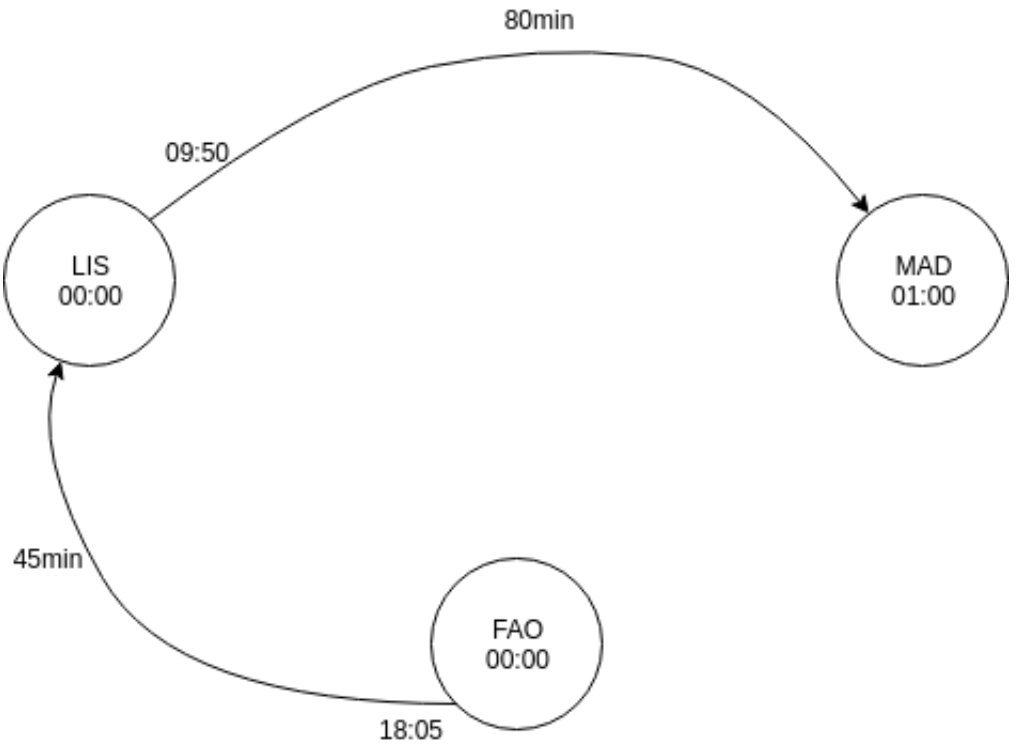
3.1 Dijkstra

Como a resolução de um dos problemas implica a prócura do caminhp mais rápido entre dois pontos, usámos um grafo orientado pesado (sem pesos negativos), achámos o algoritmos de dijkstra ser o melhor.

Para um melhor uso do algoritmo para esta situação, partimos da implementação dada nas aulas mas com alterações.

O algoritmo começa por criar uma fila prioritária para para a usar.
Em seguida coloca o aeroporto de partida na heap.
Apartir daqui entra num ciclo que e que vai fazendo pop do Aeroporto com maior prioridade e percorre os voos todos que esse aeroporto tem.
o ciclo acaba sempre que achas o Aeroporto ao qual queremos chegar ou se a heap ficar vazia.
A cada voo encontrada, verifoca se o Aeroporto de destino do Aeroporto ja foi colocado na heap (com a flag), se não tiver sido metido ainda, coloca na heap e altera a flag.

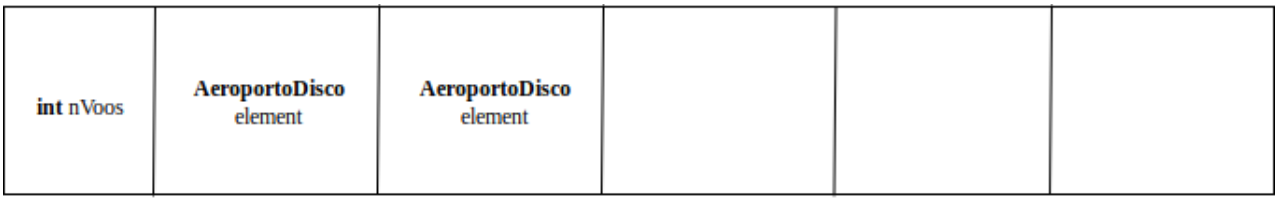
O algoritmo percebe também se o Aeroporto é intermédio ou não, pois se for, o tempo de espera dos voos tem que ser maior ou igual a 30 (os que forem menor será somado 1440min ao tempo de espera).
Assim o algoritmo usa a função de relax.



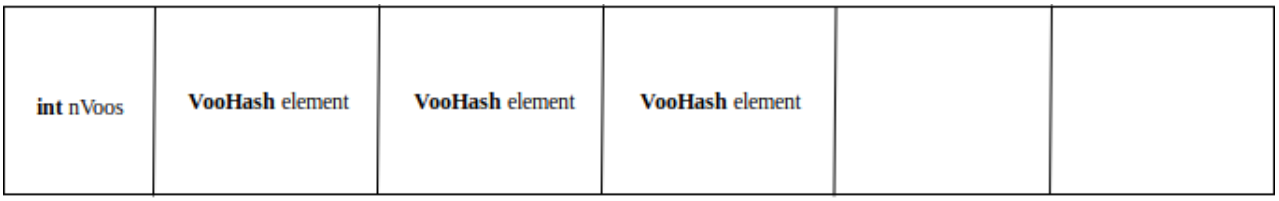
Exemplo de um grafo

4 Ficheiros de Dados

Para o trabalho usamos 2 ficheiros de disco: um para os Aeroportos, outro para os Voos. Os nossos ficheiros são ambos tratados como arrays. Na primeira posição do array fica um inteiro com o número de posições ocupadas e nas posições seguintes os elementos.



Ficheiro de Aeroportos



Ficheiro de Voos

De forma a poupar espaço na memória secundária, optamos por definir uma *strcut* ***AeroportoDisco*** que tem como parametros o código de Voo e o GMT.

Os acessos à memória secundária no nosso programa são feitos sempre que é introduzido um Aeroporto. Este vai ser escrito no ficheiro utilizando a função ***escreverAeroportoDisco***.

Por sua vez, os Voos só vão ser escritos em disco no final do programa recorrendo à função ***escreverVoosDisco***.

A leitura de ambos os ficheiros é efetuada no inicio do programa, recorrendo as funções ***lerAeroportosDisco*** e ***lerVoosDisco***, colocando os elementos nas respetivas hashtables e adicionando os Voos nos respetivos Aeroportos de partida.

5 Dimensões do Programa

5.1 Estruturas

5.1.1 Aux

A struct auxiliar ***Aux*** é constituída por um array de tamanho 200000 (número máximo de Aeroportos) e um inteiro que indica a ocupação.
Assim sendo esta struct ocupa 4 + (200000 * 8) = 1,6 MB.

5.1.2 Aeroporto

A struct tem 4 shorts, 1 inteiro, 1 array de chars de tamanho 5, 1 array de chars de tamanho 7, um boleano e 2 apontadores.
A struct ocupa (2 * 4) + 4 + (5 * 1) + (7 * 1) + 1 + (2 * 8) = 41 B.

5.1.3 HashAeroportos

A struct tem um array de tamanho 300003 (para garantir que o fator de cargar se mantem abaixo dos 0,7%) e um inteiro.
Portanto, a struct ocupa 4 + (300003 * 8) = 2,4 MB.

5.1.4 VooHash

A struct tem 2 arrays de chars de tamanho 5, 1 array de chars de tamanho 7 e 3 shorts.
(2 * (5 * 1)) + (7 * 1) + (2 * 3) = 23 B.

5.1.5 HTVoos

A struct tem 1 inteiro e um array de tamanho 900003. Assim sendo, 4 + (900003 * 8) = 7,2 MB.

5.1.6 Voo

Esta struct é o nó das listas e tem simplesmente um array de chars de tamanho 7 e um apontador para o próximo Voo. Logo, (7 * 1) + 8 = 15 B.

5.1.7 listVoos

Esta struct só tem um apontador para a cabeça da lista, logo a sua ocupação é de 8 B.

5.1.8 minHeap

Esta struct é constituída por um inteiro de ocupação e um array de tamanho 200001. Portanto, a struct ocupa 4 + (200001 * 8) = 1,6 MB.

5.2 Ocupação Máxima do Programa

Segundo as nossas contas, assumindo a ocupação máxima do programa (isto é 200000 Aeroportos e 750000 Voos), teriamos uma ocupação máxima de 1.6 M + 41 B * 200000 + 2.4 MB + 23 B * 750000 + 7.2 MB + 15 B * 750000 + 8 B + 1.6 MB = 49.4 MB.

Dado estes valores e dado que na entrega do Mooshak tinhamos o erro *Memory Limit Exceed* concluímos portanto que não consideramos toda a memória alocada nas nossas contas e que deveriamos ter feito um melhor planeamento da gestão de memória.

6 Operações

Para a realização deste trabalho tivemos que fazer alguns cálculos sobre horários.
Para tal usamos vários métodos.

6.1 diferencaHoras(Aeroporto *Apartida, VooHash *voo)

A função **diferencaHoras(Aeroporto *Apartida, VooHash *voo)** tem como paramtros o Aeroporto do qual apanhamos um voo e o próprio voo. Este método retorna o tempo que estamos à espera no aeroporto até apanha o voo.
Este cálculo é feito subtraíndo as horas a que o voo parte ás horas a que chegamos ao aeroporto. Se esse valo for negativo signivica que temos que esperar até ao dia seguinte, o que em termos matemáticos temos que sumar 1440 que são 24h * 60min.
Por exemplo: Chegamos ao aeroporto de LIS ás 10:10, se quisermos apanhar o voo ás 10:30 o cálculo será o seguinte: 10*60 + 30 - (10*60 + 10) = 20, ou seja temos que esperar 20min.
Se tivermos que apanhar o voo das 09:00 os cálculos serão: 9*60 - (10*60 + 10) = -70, como é um valor negatimo somamos os 1440 que dá 1370min que temos que esperar pelo voo.

6.2 assertGMT(Aeroporto *aDestino, Aeroporto *aPartida, VooHash *voo)

Outra função que usámos para o cálculo de horários foi **assertGMT(Aeroporto *aDestino, Aeroporto *aPartida, VooHash *voo)**, que tem como paramtros o Aeroporto de destino do voo, o aeroporto de partida do voo e o próprio voo. Este método atualiza no aeroporto de destino o horário a que chega (no seu gmt) ao apanhar o voo.
Começámos por calcular a diferença nos GMTs entre os aeroportos. Em seguida atualizamos os minutos de chegada ao Aeroporto de destino, somando os minutos a que o voo começa com o resto da divisão inteira entre o a duração do voo e 60min e sumamos também a parte dos minutos do GMT.
Em seguida avaliamos esse para valor para fazer acertos. Se o valor der mais que 59 subtraimos 60 e sumamos 1hora ao gmt (por questões práticas), mas se o valor for negativo somamos 60 e por sua vez subtraímos 1h ao GMT.
Por fim, atualizamos as horas de chegada ao aeroporto de destino somando as horas a que o voo parte, com a divisão inteira entre o tempo do voo por 60 e sumamos a parte das horas do GMT. Fazemos então o acerto das horas. Se estas forem 24 ou mais subtraímos 24, mas se forem negativas adicionamos 24.

6.3 relax(Aeroporto *aPartida, Aeroporto *aDestino, VooHash *voo, int diff)

Esta Função tem como objetivo atualizar as "distancias"dos Aeroportos à origem.
Dado um Aeroporto de partida, um Aeroporto de destido, um voo e o tempo de espera para apanhar o voo, a função compara se a soma da distancia que o Aeroporto de partida tem com o tempo de voo e com o tempo de espera é menor que a distândia que o Aeroporto de destino. Se for menor, a distância do Aeroporto de destino passa a ser a distância calculada e o Aeroporto

de partida passa a ser o predecessor do Aeroporto de Destino.

7 Expansão

Para podermos proceder à expansão deste trabalho teriamos de fazer algumas alterações de forma a corrigir a gestão de memória. Para isso teríamos de "sacrificar"tempo de execução mas pensamos que não seria um problema muito grave, dado que o nosso trabalho aparentava ter um bom desempenho em termos temporais.

8 Bibliografia

https://gist.github.com/MohamedTaha98/ccdf734f13299efb73ff0b12f7ce429f

https://www.geeksforgeeks.org/min-heap-in-java

Para a implementação a Heap vimos também as aulas teóricas dadas em EDAI.

9 Código

9.1 Lista dos ficheiros

fly.c - Main to drabalho.
djikstra.h - Declaração das primitivas do algoritmo.
djikstra.c - Algoritmo aplicado nos grafos para a procura dos caminhos mais rápidos.
hashTableAeroportos.h - Declaração das primitivas da HashTable dos Aeroportos-
hashTableAeroportos.c - HashTable de Aeroportos.
hashTableVoos.h - Declaração das primitivas da HashTable dos Voos.
hashTableVoos.c - HashTable dos Voos
list.h - Declaração das primitivas da lista.
listVoo.c - Lista ligada de voos.
pQ.h - Declaração das primitivas da heap.
pQ.c - Heap binária de Aeroportos em que a prioridade é a distancia.

9.2 Conteúdo

— fly.c —

```
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
#include<stdbool.h>
#include "djikstra.h"

#define FILEAEROPORTOS "dbAeroportos.bin"
#define FILEVOOS "dbVoos.bin"

//Estrurura auxuiliar para escrever os aeroportos em disco
typedef struct AeroportoDisco
{
    char aeroporto[CODEMAX];
    short gmtH;
    short gmtM;
}AeroportoDisco;


//Fun   o que abre o ficheiro
FILE *openFile(char *nome)
{
     //Abre a file
    FILE *file = fopen(nome, "r+b");

    if(file != NULL)
        return file;

    //caso nao exista, cria
    file = fopen(nome, "w+b");
    int z = 0;
```

```
    fwrite(&z, sizeof(int), 1, file);
    return file;
}


//Fun  o que fecha um ficheiro
void close_file(FILE *file)
{
    fclose(file);
}


//Fun  o que escreve um aeropordo em mem ria secund ria
void escreverAeroportoDisco(FILE *file, char aeroporto[CODEMAX], short
gmtH, short gmtM, int nAeroportos)
{
    fseek(file, 0, SEEK_SET);
    fwrite(&nAeroportos, sizeof(int), 1, file);
    AeroportoDisco *temp = malloc(sizeof(AeroportoDisco));
    strcpy(temp->aeroporto, aeroporto);
    temp->gmtH = gmtH;
    temp->gmtM = gmtM;

    fseek(file, sizeof(int) + (nAeroportos-1)*sizeof(AeroportoDisco),
SEEK_SET);
    fwrite(temp, sizeof(AeroportoDisco), 1, file);

    free(temp);
}


//Fun  o que l  todos os Aeroportos que existem em mem ria
secund ria e os coloca em mem ria principal
void lerAeroportosDisco(FILE *file, HashAeroportos *hA)
{
    fseek(file, 0, SEEK_SET);
    int nA;
    fread(&nA, sizeof(int), 1, file);

    AeroportoDisco *in = malloc(sizeof(AeroportoDisco));
    for (int i = 0; i < nA; i++)
    {
        fread(in, sizeof(AeroportoDisco), 1, file);
        insertAeroporto(hA, in->aeroporto, in->gmtH, in->gmtM);
    }
    free(in);
}


//Fun  o que mete todos os voos que existem no final do programa e os
 coloca em mem ria secund ria
```

```
void escreverVoosDisco(FILE *file, HTVoos *hV)
{
    fseek(file, 0, SEEK_SET);
    fwrite(&hV->numeroVoo, sizeof(int), 1, file);
    int nV = hV->numeroVoo;

    for (int i = 0; i < MAXVOOS; i++)
    {
        if (nV == 0)
        {
            return;
        }

        if (hV->array[i] != NULL)
        {
            fwrite(hV->array[i], sizeof(VooHash), 1, file);
            nV --;
        }
    }
}


//Fun  o que l  todos os voos que existem em mem ria secund ria e
mete em mem ria principal
void lerVoosDisco(FILE *file, HashAeroportos *hA, HTVoos *hV)
{
    fseek(file, 0, SEEK_SET);
    int nV;
    fread(&nV, sizeof(int), 1, file);
    VooHash *temp = malloc(sizeof(VooHash));
    for (int i = 0; i < nV; i++)
    {
        fread(temp, sizeof(VooHash), 1, file);
        insertVoo(hV, temp->codigo, temp->partida, temp->destino, temp
->horas, temp->minutos, temp->duracao);
        addVoo(findAeroporto(hA, temp->partida)->lista, temp->codigo);
    }
    free(temp);

}

int main(void)
{

    Aux *aux = malloc(sizeof(Aux));
    aux->ocupacao = 0;

    HashAeroportos *hA = criarHashAeroportos();
    HTVoos *hV = criarHTVoos();

    FILE *fileA = openFile(FILEAEROPORTOS);
```

```
    FILE *fileV = openFile(FILEVOOS);

    lerAeroportosDisco(fileA, hA);
    lerVoosDisco(fileV, hA, hV);


    char operacao[2];
    while(scanf("%s", operacao) != EOF)
    {
        //Add Aeroporto
        if(strcmp(operacao, "AI") == 0)
        {
            char aeroporto[CODEMAX];
            short gmtH, gmtM;
            scanf("%s␣%hd:%hd", aeroporto, &gmtH, &gmtM);
            if(findAeroporto(hA, aeroporto) == NULL)
            {
                insertAeroporto(hA, aeroporto, gmtH, gmtM);
                printf("+␣novo␣aeroporto␣%s\n", aeroporto);
                escreverAeroportoDisco(fileA, aeroporto, gmtH, gmtM, hA
->numeroAeroportos);
            }
            else
            {
                printf("+␣aeroporto␣%s␣existe\n", aeroporto);
            }

        }
        //Add Voo
        else if(strcmp(operacao, "FI") == 0)
        {
            char codigoVoo[CODEVOO];
            char aeroportoPartida[CODEMAX];
            char aeroportoDestino[CODEMAX];
            short horas;
            short minutos;
            short duracao;
            scanf("%s␣%s␣%s␣%hd:%hd␣%hd", codigoVoo, aeroportoPartida,
aeroportoDestino, &horas, &minutos, &duracao);
            if (findVoo(hV, codigoVoo) == NULL)
            {
                if (findAeroporto(hA, aeroportoPartida)!= NULL)
                {
                    if (findAeroporto(hA, aeroportoDestino)!= NULL)
                    {
                        insertVoo(hV, codigoVoo, aeroportoPartida,
aeroportoDestino, horas, minutos, duracao);
                        addVoo(findAeroporto(hA, aeroportoPartida)->
lista, codigoVoo);
                        printf("+␣novo␣voo␣%s\n", codigoVoo);
                    }
```

```
                    else
                    {
                        printf("+ aeroporto %s desconhecido\n",
aeroportoDestino);
                    }
                }
                else
                {
                    printf("+ aeroporto %s desconhecido\n",
aeroportoPartida);
                }

            }
            else
            {
                printf("+ voo %s existe\n", codigoVoo);
            }
        }
        //Remo   o de voo
        else if(strcmp(operacao, "FD") == 0)
        {
            char codigoVoo[CODEVOO];
            scanf("%s", codigoVoo);
            if (findVoo(hV, codigoVoo) != NULL)
            {
                VooHash *rmAeroporto = findVoo(hV, codigoVoo);
                listaVoos * lista = findAeroporto(hA, rmAeroporto->
partida)->lista;
                delVoo(lista, codigoVoo);
                removeVoo(hV, codigoVoo);
                printf("+ voo %s removido\n", codigoVoo);
            }
            else
            {
                printf("+ voo %s inexistente\n", codigoVoo);
            }
        }

        //Procura caminho
        else if(strcmp(operacao, "TR") == 0)
        {
            char AeroportoPartida[CODEMAX];
            char AeroportoDestino[CODEMAX];
            short horas;
            short minutos;
            scanf("%s %s %hd:%hd", AeroportoPartida, AeroportoDestino,
&horas, &minutos);
            Aeroporto *partida = findAeroporto(hA, AeroportoPartida);
            Aeroporto *destino = findAeroporto(hA, AeroportoDestino);
            if (partida != NULL)
            {
```

```
                if (findAeroporto(hA, AeroportoDestino) != NULL)
                {
                    aux->array[aux->ocupacao] = partida;
                    aux->ocupacao ++;
                    dijkstra(hV, hA, AeroportoPartida, AeroportoDestino
, horas, minutos, aux);

                    if (destino->predecessor == NULL)
                    {
                        printf("+ sem voos de %s para %s\n",
AeroportoPartida, AeroportoDestino);
                    }
                    else
                    {
                        printf("Voo    De    Para Parte Chega\n");
                        printf("====== ==== ==== ===== =====\n");
                        printVoos(findAeroporto(hA, AeroportoDestino),
hV);
                        printf("Tempo de viagem: %d minutos\n", destino
->distance);
                    }
                    resetAux(aux);
                }
                else
                {
                    printf("+ aeroporto %s desconhecido\n",
AeroportoDestino);
                }
            }
            else
            {
                printf("+ aeroporto %s desconhecido\n",
AeroportoPartida);
            }
        }
        else if(strcmp(operacao, "X") == 0)
        {
            escreverVoosDisco(fileV, hV);
            destroyHashtable(hA);
            destroyHashtableVoo(hV);
            break;
        }
    }
    free(aux);

    close_file(fileA);
    close_file(fileV);
    return 0;
}
```

— djikstra.h —

```
#include "pQ.h"
#include "hashTableVoos.h"


#define INFINITY 9999999
#define MIN24H 1440
#define MIN1H 60
#define H24 24
#define AEROPORTOS200000 200000

//Estrutura auxiliar utilizada para que o algoritmo seja mais r pido,
pois n o ir  precisar de percurrer a hash toda
typedef struct Aux
{
    int ocupacao;
    Aeroporto *array[AEROPORTOS200000];
}Aux;

void resetAux(Aux *aux);
void ininitializeSingleSourcePartida(Aeroporto *partida, short
horasChegada, short minutosChegada);
void ininitializeSingleSourceA(Aeroporto *a);
short diferencaHoras(Aeroporto *Apartida, VooHash *voo);
void assertGMT(Aeroporto *aDestino, Aeroporto *aPartida, VooHash *voo);
bool relax(Aeroporto *aPartida, Aeroporto *aDestino, VooHash *voo, int
diff);
void dijkstra(HTVoos *hV, HashAeroportos *hA, char *aeroportoPartida,
char *aeroportoDestino, short horasChegada, short minutosChegada, Aux *
aux);
void printVoos(Aeroporto *chegada, HTVoos *hV);
```

— djikstra.c —

```
#include "djikstra.h"

//Fun  o que colada a flag a false de todos os aeroportos que "
participaram" no algoritmo de dijkstra
void resetAux(Aux *aux)
{
    for (int i = 0; i < aux->ocupacao; i++)
    {
        aux->array[i]->flag = false;
        aux->array[i]->predecessor = NULL;
    }

    aux->ocupacao = 0;
}


//Fun  o que inicializa o Aeroporto de partida com a flag a false
para o algoritmo de dijkstra
```

```
void ininitializeSingleSourcePartida(Aeroporto *partida, short
horasChegada, short minutosChegada)
{
    partida->distance = 0;
    partida->predecessor = NULL;
    partida->horasChegada = horasChegada;
    partida->minutosChegada = minutosChegada;
    partida->flag = true;
}

//Fun  o que inicializa um Aeroporto com a flag a false para o
algoritmo de dijkstra
void ininitializeSingleSourceA(Aeroporto *a)
{
    a->distance = INFINITY;
    a->predecessor = NULL;
    a->flag = true;
}


//Fun  a que da os minutos de espera num aeroporto para apanhar o voo
short diferencaHoras(Aeroporto *Apartida, VooHash *voo)
{
    int mins;
    mins = voo->horas*MIN1H + voo->minutos;
    mins = mins - (Apartida->horasChegada*MIN1H + Apartida->
minutosChegada);
    if (mins < 0)
    {
        mins += MIN24H;
    }
    return mins;
}


//Fun  o que calcula a hora de chegada ao aeropordo de destino no seu
 gmt a partir do gmt do aeroporto de partida, e do tempo do voo
void assertGMT(Aeroporto *aDestino, Aeroporto *aPartida, VooHash *voo)
{
    short gmtH = aDestino->gmtHoras - aPartida->gmtHoras;
    short gmtM = aDestino->gmtMinutos - aPartida->gmtMinutos;

    aDestino->minutosChegada = voo->minutos + (voo->duracao % MIN1H) +
gmtM;
    while (aDestino->minutosChegada >= MIN1H)
    {
        aDestino->minutosChegada -= MIN1H;
        gmtH ++;
    }

    //talvez fa a falta :3
```

```
    while (aDestino->minutosChegada < 0)
    {
        aDestino->minutosChegada += MIN1H;
        gmtH --;
    }

    //neste caso n o se adiciona a dura  o do voo pois    dado em
minutos e descontado no ciclo while em cima
    aDestino->horasChegada = voo->horas + (voo->duracao / MIN1H) + gmtH
;

    while (aDestino->horasChegada >= H24)
    {
        aDestino->horasChegada -= H24;
    }

    while (aDestino->horasChegada < 0)
    {
        aDestino->horasChegada += H24;
    }
}


//Fun  o que aplica o relax
bool relax(Aeroporto *aPartida, Aeroporto *aDestino, VooHash *voo, int
diff)
{
    if ((aPartida->distance + voo->duracao + diff) < aDestino->distance
)
    {
        aDestino->distance = aPartida->distance + voo->duracao + diff;
        aDestino->predecessor = aPartida;

        assertGMT(aDestino, aPartida, voo);
        strcpy(aDestino->codeVOO, voo->codigo);

        return true;
    }
    return false;

}


//Fun  o que aplica o alguritmo de Dijkstra
void dijkstra(HTVoos *hV, HashAeroportos *hA, char *aeroportoPartida,
char *aeroportoDestino, short horasChegada, short minutosChegada, Aux *
aux)
{
    min_heap *pQ = new_heap();

    Aeroporto *partida = findAeroporto(hA, aeroportoPartida);
```

```
    ininitializeSingleSourcePartida(partida, horasChegada,
minutosChegada);
    insert(pQ, partida);
    Voo *vTemp;
    VooHash *vHashTemp;
    Aeroporto *AeroportoTemp;

    while (!emptyHeap(pQ))
    {
        partida = pop(pQ);
        if (strcmp(partida->codigo, aeroportoDestino) == 0)
        {
            destroyHeap(pQ);
            return;
        }

        vTemp = partida->lista->head;
        while (vTemp != NULL)
        {
            vHashTemp = findVoo(hV, vTemp->codVoo);
            AeroportoTemp = findAeroporto(hA, vHashTemp->destino);

            if (!AeroportoTemp->flag)
            {
                ininitializeSingleSourceA(AeroportoTemp);
                insert(pQ, AeroportoTemp);
                aux->array[aux->ocupacao] = AeroportoTemp;
                aux->ocupacao ++;
            }

            if (diferencaHoras(partida, vHashTemp) >= 30 || partida->
distance == 0)
            {
                if(relax(partida, findAeroporto(hA,vHashTemp->destino),
 vHashTemp, diferencaHoras(partida, vHashTemp)))
                {
                    update(pQ, findAeroporto(hA,vHashTemp->destino));
                }
            }
            else if (diferencaHoras(partida, vHashTemp) < 30 && partida
->distance != 0)
            {
                int diff = diferencaHoras(partida, vHashTemp) + MIN24H;
                if(relax(partida, findAeroporto(hA,vHashTemp->destino),
 vHashTemp, diff))
                {
                    update(pQ, findAeroporto(hA,vHashTemp->destino));
                }
            }

            vTemp = vTemp->next;
```

```
        }
    }
    destroyHeap(pQ);
}


//Fun  o que faz print dos voos de forma recurseiva. A partir do
aeroporto de destido, chega ao aeroporto de partida e faz prin dos voos
 por ordem
//Tem como argumento a hash table onde est o os Voos e o Aeroporto de
destino
void printVoos(Aeroporto *chegada, HTVoos *hV)
{
    if (chegada->predecessor != NULL)
    {
        printVoos(chegada->predecessor, hV);

    }
    else
    {
        return;
    }

    printf("%-6s␣%-4s␣%-4s␣%02d:%02d␣%02d:%02d\n",chegada->codeVOO,
chegada->predecessor->codigo, chegada->codigo, findVoo(hV, chegada->
codeVOO)->horas, findVoo(hV, chegada->codeVOO)->minutos, chegada->
horasChegada, chegada->minutosChegada);
}
```

— hashTableAeroportos.h —

```
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
#include<stdbool.h>
#include "list.h"

#define MAXAEROPORTOS 300003 // num. primo
#define CODEMAX 5

typedef struct Aeroporto
{
    char codigo[CODEMAX];
    short gmtHoras;
    short gmtMinutos;

    struct listVoos *lista;

    int distance;
    struct Aeroporto *predecessor;

    short horasChegada;
```

```
    short minutosChegada;

    bool flag;

    char codeVOO[CODEMAXVOO];
} Aeroporto;

typedef struct HashAeroportos
{
    int numeroAeroportos;
    Aeroporto *array[MAXAEROPORTOS];
} HashAeroportos;

HashAeroportos *criarHashAeroportos();
unsigned long hashCodeAeroporto(char *str);
int findPos(HashAeroportos *hashA, char *code);
Aeroporto *findAeroporto(HashAeroportos *hashA, char *code);
bool isEmpty(HashAeroportos *hashA);
bool insertAeroporto(HashAeroportos *hashA, char *code, short gmtHoras,
 short gmtMinutos);
void destroyHashtable(HashAeroportos *ht);
```

— hashTableAeroportos.c —

```
#include "pQ.h"


HashAeroportos *criarHashAeroportos()
{
    HashAeroportos *hash = (HashAeroportos *)malloc(sizeof(
HashAeroportos));
    for(int i = 0; i < MAXAEROPORTOS; i++)
    {
        hash->array[i] = NULL;
    }
    hash->numeroAeroportos = 0;
    return hash;
}

void destroyHashtable(HashAeroportos *ht)
{
    for(int i = 0; i < MAXAEROPORTOS; i++)
    {
        if (ht->array[i] != NULL)
        {
            listaVoosDestroy(ht->array[i]->lista);
            free(ht->array[i]);
        }
    }
    free(ht);
}
```

```
//https://gist.github.com/MohamedTaha98/
ccdf734f13299efb73ff0b12f7ce429f
unsigned long hashCodeAeroporto(char *str)
{

    unsigned long hash = 5381;
    int c;
    while ((c = *str++))
        hash = ((hash << 5) + hash) + c;
    return hash % MAXAEROPORTOS;

}

//Fun   o hash para o double hashing
int hashCodeAeroporto2(char *code)
{
    int codeR = hashCodeAeroporto(code);
    codeR = codeR % 3;
    return codeR;
}

//Fun o que procura a posi   o do aeroporto ou a primeira posi   o a
 null, usando fun   es de hash
int findPos(HashAeroportos *hashA, char *code)
{
    int hashcode = hashCodeAeroporto(code);
    int p = 1;
    while (hashA->array[hashcode] != NULL)
    {
        if (strcmp(hashA->array[hashcode]->codigo, code) == 0)
        {
            return hashcode;
        }
        hashcode = (hashcode + hashCodeAeroporto2(code) + p) %
MAXAEROPORTOS;
        p ++;
    }
    return hashcode;
}

//Fun   o que dado um c digo de aeroporto, retorna o aeroporto
Aeroporto *findAeroporto(HashAeroportos *hashA, char *code)
{
    if (hashA->array[findPos(hashA, code)] == NULL)
    {
        return NULL;
    }

    Aeroporto *temp = hashA->array[findPos(hashA, code)];
    return temp;
```

```
}

//Fun   o q ue diz que a hash esta vazia ou n o
bool isEmpty(HashAeroportos *hashA)
{
    if (hashA->numeroAeroportos == 0)
    {
        return true;
    }
    return false;
}


//Fun   o que insere um aeropordo na hash table
bool insertAeroporto(HashAeroportos *hashA, char *code, short gmtHoras,
 short gmtMinutos)
{
    int pos = findPos(hashA, code);

    hashA->array[pos] = (Aeroporto *)malloc(sizeof(Aeroporto));
    strcpy(hashA->array[pos]->codigo, code);
    hashA->array[pos]->lista = new_listaVoos();
    hashA->array[pos]->gmtHoras = gmtHoras;
    if (gmtHoras < 0)
    {
        hashA->array[pos]->gmtMinutos = 0 - gmtMinutos;
    }
    else
    {
        hashA->array[pos]->gmtMinutos = gmtMinutos;
    }
    hashA->array[pos]->flag = false;
    hashA->numeroAeroportos ++;
    return true;
}
```

— hashTableVoos.h —

```
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
#include<stdbool.h>

#define MAXVOOS 900003 // num. primo
#define CODEVOO 7
#define CODEMAX 5


typedef struct VooHash
{
    char codigo[CODEVOO];
    char partida[CODEMAX];
```

```
    char destino[CODEMAX];
    short horas;
    short minutos;
    short duracao;
} VooHash;

typedef struct HTVoos
{
    int numeroVoo;
    VooHash *array[MAXVOOS];
} HTVoos;

HTVoos *criarHTVoos();
void destroyHashtableVoo(HTVoos *ht);
unsigned long hashCodeVoo(char *str);
int hashCodeVoo2(char *code);
int findPosVoos(HTVoos *hashA, char *code);
VooHash *findVoo(HTVoos *hashV, char *code);
bool isEmptyVoo(HTVoos *hashA);
bool insertVoo(HTVoos *hashA, char *code, char *partida, char *destino,
 short horas, short minutos, short duracao);
void removeVoo(HTVoos *hashA, char *code);
```

— hashTableVoos.c —

```
#include "hashTableVoos.h"

HTVoos *criarHTVoos()
{
    HTVoos *hash = (HTVoos *)malloc(sizeof(HTVoos));
    for(int i = 0; i < MAXVOOS; i++)
    {
        hash->array[i] = NULL;
    }
    hash->numeroVoo = 0;
    return hash;
}

void destroyHashtableVoo(HTVoos *ht)
{
    for(int i = 0; i < MAXVOOS; i++)
    {
        if (ht->array[i] != NULL)
        {
            free(ht->array[i]);
        }
    }
    free(ht);
}
```

//https://gist.github.com/MohamedTaha98/

```
ccdf734f13299efb73ff0b12f7ce429f
unsigned long hashCodeVoo(char *str) {

    unsigned long hash = 5381;
    int c;
    while ((c = *str++))
        hash = ((hash << 5) + hash) + c; /* hash * 33 + c */
    return hash % MAXVOOS;
}

int hashCodeVoo2(char *code)
{
    int codeR = hashCodeVoo(code);
    codeR = codeR % 3;
    return codeR;
}

int findPosVoos(HTVoos *hashA, char *code)
{
    int hashcode = hashCodeVoo(code);
    int p = 1;
    while (hashA->array[hashcode] != NULL)
    {
        if (strcmp(hashA->array[hashcode]->codigo, code) == 0)
        {
            return hashcode;
        }
        hashcode = (hashcode + hashCodeVoo2(code) + p) % MAXVOOS;
        p ++;
    }
    return hashcode;
}

VooHash *findVoo(HTVoos *hashV, char *code)
{
    if (hashV->array[findPosVoos(hashV, code)] == NULL)
    {
        return NULL;
    }

    VooHash *temp = hashV->array[findPosVoos(hashV, code)];
    return temp;
}

bool isEmptyVoo(HTVoos *hashA)
{
    if (hashA->numeroVoo == 0)
    {
        return true;
    }
    return false;
```

```
}

bool insertVoo(HTVoos *hashA, char *code, char *partida, char *destino,
 short horas, short minutos, short duracao)
{
    int pos = findPosVoos(hashA, code);

    hashA->array[pos] = malloc(sizeof(VooHash));
    strcpy(hashA->array[pos]->codigo, code);
    strcpy(hashA->array[pos]->partida, partida);
    strcpy(hashA->array[pos]->destino, destino);
    hashA->array[pos]->horas = horas;
    hashA->array[pos]->minutos = minutos;
    hashA->array[pos]->duracao = duracao;

    hashA->numeroVoo ++;
    return true;
}

void removeVoo(HTVoos *hashV, char *code)
{
    VooHash *temp = hashV->array[findPosVoos(hashV, code)];
    hashV->array[findPosVoos(hashV, code)] = 0;
    free(temp);
    hashV->numeroVoo --;
}
```

— list.h —

```
#include <stdbool.h>

#define CODEMAXVOO 7
#define MAXAEROPORTOS 300003

typedef struct Voo
{
    char codVoo[CODEMAXVOO];
    struct Voo *next;
} Voo;

typedef struct listVoos
{
    Voo *head;
} listaVoos;

typedef struct Grafo
{
    int nAeroportos;
    listaVoos *lista[MAXAEROPORTOS];
} Grafo;

listaVoos *new_listaVoos();
```

```
Voo *new_voo(char cod[CODEMAXVOO]);
void listaVoosDestroy(listaVoos *lista);
void addVoo(listaVoos *list, char codeVoo[CODEMAXVOO]);
void delVoo(listaVoos *list, char codeVoo[CODEMAXVOO]);
```

— listVoo.c —

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "list.h"

Voo *new_voo(char cod[CODEMAXVOO])
{
    Voo *node = malloc(sizeof(Voo));

    strcpy(node->codVoo, cod);

    node->next = NULL;

    return node;
}

listaVoos *new_listaVoos()
{
    listaVoos *lista = malloc(sizeof(listaVoos));

    lista->head = NULL;

    return lista;
}

void addVoo(listaVoos *lista, char cod[CODEMAXVOO])
{
    Voo *novoVoo = new_voo(cod);

    if(lista->head == NULL)
    {
        lista->head = novoVoo;
        lista->head->next = NULL;
    }
    else
    {
        novoVoo->next = lista->head;
        lista->head = novoVoo;
    }
}

void delVoo(listaVoos *lista, char cod[CODEMAXVOO])
{
    Voo *anterior = NULL;
    Voo *atual = lista->head;
```

```
    //Caso o n  esta na cabe a
    if(atual != NULL && strcmp(atual->codVoo, cod) == 0)
    {
        lista->head = atual->next;
        free(atual);
        return;
    }

    while(atual != NULL && strcmp(atual->codVoo, cod) != 0)
    {
        anterior = atual;
        atual = atual->next;
    }

    anterior->next = atual->next;
    free(atual);
}

void listaVoosDestroy(listaVoos *lista)
{
    Voo *atual = lista->head;
    Voo *temp = NULL;

    while(atual != NULL)
    {
        temp = atual->next;
        free(atual);
        atual = temp;
    }
    free(lista);
}
```

— pQ.h —

```
#include "hashTableAeroportos.h"

#define MAXSIZE 200001
#define FRONT 1

typedef struct min_heap
{
    int occupied;
    Aeroporto *array[MAXSIZE];
} min_heap;

min_heap *new_heap();
void destroyHeap(min_heap *h);
int parent(int pos);
int leftChild(int pos);
int rightChild(int pos);
bool isLeaf(min_heap *h, int pos);
```

```
void swap(min_heap *h, int fpos, int spos);
void minHeapify(min_heap *h, int pos);
void insert(min_heap *h, Aeroporto *element);
void minHeap(min_heap *h);
Aeroporto *pop(min_heap *h);
bool emptyHeap(min_heap *h);
void update(min_heap *h, Aeroporto *element);
```

— pQ.c —

```
#include <stdio.h>
#include <string.h>
#include "pQ.h"

//https://www.geeksforgeeks.org/min-heap-in-java/

min_heap *new_heap()
{
    min_heap *Queue = malloc(sizeof(min_heap));

    Queue->occupied = 0;

    return Queue;
}

void destroyHeap(min_heap *h)
{
    free(h);
}

int parent(int pos)
{
    return (pos / 2);
}

int leftChild(int pos)
{
    return (2 * pos);
}

int rightChild(int pos)
{
    return ((2*pos) + 1);
}

void swap(min_heap *h, int fpos, int spos)
{
    Aeroporto *temp = h->array[fpos];
    h->array[fpos] = h->array[spos];
    h->array[spos] = temp;
}
```

```
bool emptyHeap(min_heap *h)
{
    return h->occupied == 0;
}

void minHeapify(min_heap *h, int pos)
{
    if((leftChild(pos) <= h->occupied && (h->array[pos]->distance > h->
array[leftChild(pos)]->distance)) || (rightChild(pos) <= pos  && (h->
array[pos]->distance > h->array[rightChild(pos)]->distance)))
    {
        if((h->array[leftChild(pos)]->distance) < (h->array[rightChild(
pos)]->distance))
        {
            swap(h, pos, leftChild(pos));
            minHeapify(h, leftChild(pos));
        }

        else
        {
            swap(h, pos, rightChild(pos));
            minHeapify(h, rightChild(pos));
        }

    }
}

void update(min_heap *h, Aeroporto *element)
{
    int current = h->occupied;
    while(current != 1 && h->array[current]->distance < h->array[parent
(current)]->distance)
    {
        swap(h, current, parent(current));
        current = parent(current);
    }
}

void insert(min_heap *h, Aeroporto *element)
{
    if(h->occupied >= MAXSIZE)
    {
        return;
    }

    if(emptyHeap(h))
    {
        h->occupied++;
        h->array[h->occupied] = element;
        return;
    }
```

```
    h->array[++h->occupied] = element;


    update(h, element);

    /*

    int current = h->occupied;

    while(current != 1 && h->array[current]->distance < h->array[parent
(current)]->distance)
    {
        swap(h, current, parent(current));
        current = parent(current);
    }

    */
}

void minHeap(min_heap *h)
{
    for(int pos = (h->occupied/2); pos >= 1; pos--)
    {
        minHeapify(h, pos);
    }
}

Aeroporto *pop(min_heap *h)
{
    Aeroporto *popped = h->array[FRONT];
    if (popped == NULL)
    {
        return NULL;
    }

    h->array[FRONT] = h->array[h->occupied--];
    minHeapify(h, FRONT);
    return popped;
}
```