UNIVERSIDADE DE ÉVORA

Licenciatura em Engenharia Informática

Estruturas de Dados e Algoritmos II

Docente: Vasco Pedro

2018/2019

1 de Julho de 2019

Relatório:

Rede de Aeroportos

Autores:

Tiago Martinho 35735

Nuno Cerdeira 35843

Índice

1 Introdução

Este trabalho tem como objetivo desenvolver um **Sistema de Gestão de Voos e de Aeroportos** que irá permitir descobrir, a partir do conhecimento da rede de aeroportos e dos voos existentes, que voos deve um utilizador apanhar para chegar ao destino desejado. As entidades implementadas neste sistema são os **aeroportos** e os **voos**. A partir destas entidades, o sistema irá calcular as viagens entre dois aeroportos. Neste trabalho optámos por implementar a versão completa do enunciado apresentado onde um **aeroporto** irá ser identificado pelo seu código único, que será uma sequência de 3 ou 4 letras maiúsculas. A cada aeroporto está também associado o fuso horário da região em que este se localiza.

Um **fuso horário** que será indicado como a diferença entre a hora local e a hora no meridiano de Greenwich (GMT). Essa diferença será positiva nos fusos horários a Este de Greenwich (Europa, África, Ásia e Oceânia) e negativa naqueles a Oeste (América e parte das ilhas do Pacífico). O fuso horário não vai estar associado a um país mas sim a um aeroporto.

Um **voo** será identificado por um código único, que consiste numa sequência de 2 letras maiúsculas, seguida por uma sequência de 1 a 4 algarismos decimais. Além do **código único**, um voo é caracterizado pelos **aeroportos de partida e de destino**, pela sua **hora de partida**, dada no fuso horário do aeroporto de partida, e pela sua **duração** em minutos.

O programa vai ler os comandos a executar da entrada normal (standard input) e escrever as respostas na saída normal (standard output). Para além disto, o program vai guardar em **memória secundária** (em disco) toda a informação.

2 Estruturas de Dados

2.1 Descrição das Estruturas de Dados Utilizadas

Para a realização deste trabalho, optou-se por implementar **quatro tipos de estruturas de dados**: duas ***Hashtables*** de pesquisa linear, uma para os **Voos** outra para os **Aeroportos**; foi implementada também uma estrutura ***Heap*** (uma ***Queue*** com prioridade); implementou-se ainda uma ***single Linked List***.

2.1.1 *Hashtable* Voos (*fly_hash*)

A ***Hashtable* Voos**, trata-se de uma estrutura de dados que associa uma chave a um index num array. Foi escolhida uma implementação linear de endereçamento fechado visto que a função de hashing é única para cada chave, reduzindo assim as colisões. Escolheu-se esta estrutura pela sua **complexidade temporal *O(1)***, no **melhor caso**, e ***O(n)*** no **pior caso** relativamente à pesquisa de valores e inserção dos mesmos. Quanto às suas dimensões, de modo a minimizar ainda mais as colisões, optou-se por **duplicar o tamanho de voos máximos (750 000)**, e somou-se **3** resultando num valor final de **1 500 003** um número não primo mas com apenas 6 divisores.

Os valores guardados são apontadores para ***structs*** do tipo ***Flight*** que guardam toda a informação referente a cada voo. Ou seja, o seu **código**, o **aeroporto de partida**, o **aeroporto de destino**, a sua **hora de partida** e a **duração do voo**.

Esta ***hashtable*** encontra-se presente em memória central. Caso não seja a primeira vez que o programa é corrido, irá procurar em memória secundária pelo ficheiro referente à mesma e reconstrói (inserindo ***structs*** do tipo ***Flight***) em memória central. No fim da execução é novamente escrito o seu conteúdo em memória secundária.

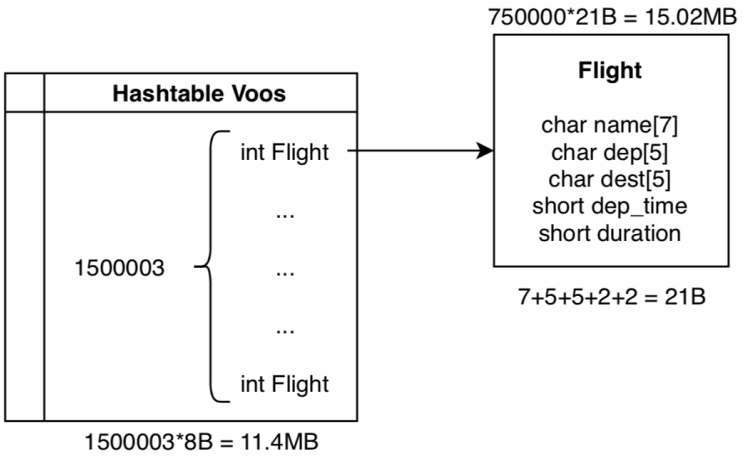

Fig.1 - *Hashtable* Voos

2.1.2 *Linked List* (*flight_list*)

Implementou-se uma ***single Linked List*** uma vez que o número de voos que partem de cada aeroporto é dinâmico podendo variar entre 0 e 150. Ao utilizar-se uma ***linked list*** só será alocado espaço caso este seja necessário. Outra vantagem é que a inserção de um novo elemento é constante ***O(1)*** pois trata-se de uma **inserção à cauda**, a **remoção** por outro lado tem como pior caso uma **complexidade de *O(n)***.

Como **existem no máximo 200 000 aeroportos** e de cada um **podem partir 150 voos** seria **200 000*150 = 30 000 000 voos possíveis**. Este valor será muito maior que os 750 000 referidos no enunciado. Ou seja, **existem no máximo 750 000 elementos de cada lista**.

Existe uma ***flight list*** para cada aeroporto com o intuito de haver **uma lista de voos que partem de cada aeroporto**, algo que será útil na aplicação do algoritmo de ***Dijkstra***.

Esta estrutura existe unicamente em memória central. Sendo construída quando é inserido um novo voo ou quando é lida a ***hashtable*** de voos. Cada lista contém um apontador para o primeiro elemento da lista e cada elemento por sua vez tem um apontador para o próximo elemento da lista e um apontador para uma ***struct*** do tipo ***Flight***.

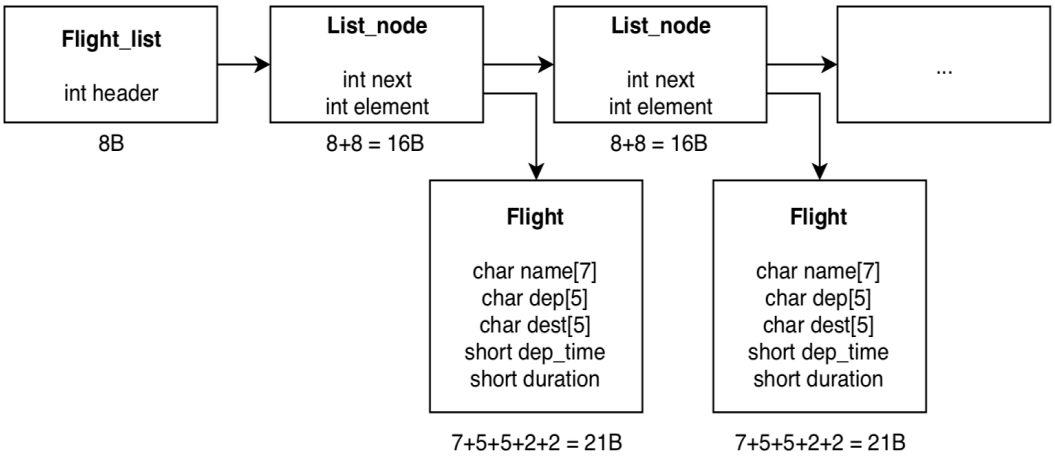

Fig.2 - *Linked List* Voos

2.1.3 *Hashtable* Aeroportos (*air_hash*)

Escolheu-se uma ***hashtable*** com uma implementação linear de endereçamento fechado pelas mesmas razões que se escolheu a ***hashtable* dos voos**, mas com **3 arrays em vez de apenas 1** aproveitando assim as vantagens da complexidade temporal na pesquisa presente nas ***hashtables***. Utilizou-se um ***hashcode* único** para cada aeroporto de modo a reduzir as colisões.

Relativamente à dimensão da ***hashtable*** foi escolhido o **dobro do valor total de aeroportos (200 000)** mais **3** resultando em **400 003** apesar de não se tratar de um número primo tem apenas 4 divisores de modo a minimizar as colisões.
Cada aeroporto tem um apontador para a sua lista de voos de partida e para o seu vértice.

Esta ***hashtable*** encontra-se presente em memória central. Caso não seja a primeira vez que o programa é corrido irá procurar em memória secundária pelo ficheiro referente à mesma e reconstrói (inserindo ***structs*** do tipo ***Airport***) em memória central. No fim da execução é novamente escrito o seu conteúdo em memória secundária. Os elementos que a englobam são ***structs*** do tipo ***Airport***, **Vertice** e ***Flight_list***. Os ***structs*** do tipo ***Airport*** mantêm a informação referente a cada aeroporto ou seja o seu **código**, o seu **GMT**, **um valor booleano para escrita no disco** e a **posição que o vértice mantém na *Heap***. Os ***structs*** do tipo **Vertice** guardam o **nome do aeroporto**, o **voo que apanhou para chegar lá (o *parent*)** e a **distância**. Tudo isto será informação necessária para o algoritmo de ***Djikstra***.

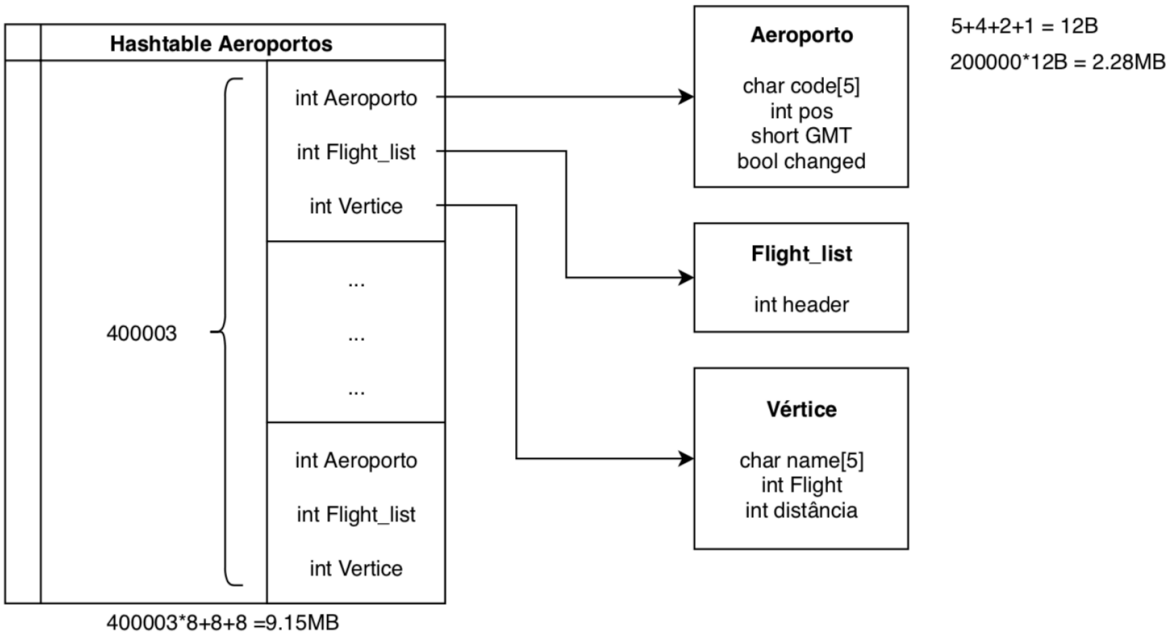

Fig.3 - *Hashtable* Aeroportos

2.1.4 *Min-Binary-Heap* (*heap*)

Esta ***binary heap*** serve como uma ***priority queue*** algo crucial para a implementação do algoritmo de ***Dijkstra***, pode-se resumir como sendo um array que segue as **propriedades de uma árvore binária completa**. Escolheu-se uma ***binary heap*** devido à sua complexidade temporal no respeito à **inserção e remoção de *O(log n)***. Esta ***heap*** tem uma **dimensão de 200 002 vértices** pois foi feita uma implementação que começa na segunda posição do ***array***.

Encontra-se unicamente em memória central. Os elementos do seu ***array*** são apontadores para ***structs*** do tipo **Vertice** e tem também **um valor inteiro que representa o tamanho da mesma**.

Fig.4 - *Heap*

3 Ficheiros de Dados

3.1 Aeroportos

Como o **número dos aeroportos presentes no sistema** pode variar entre **0 e 200 000** decidiu-se, numa tentativa de **reduzir o tempo de leitura**, **escrever todos os aeroportos** que foram **criados** (e **lidos**) nesta instância e **escrever no fim** um **aeroporto** cujo código tem um terminador nulo no primeiro caractere (algo que nunca deve acontecer). Deste modo **existe um delimitador** para quando for feita a leitura do ficheiro e para saber quando não existem mais elementos a ler.

Cada **aeroporto** tem 4 variáveis:

⇒ ***code*** - Código pelo qual cada aeroporto é identificado que pode variar entre 3 a 4 letras maiúsculas, contando com o caractere nulo fica 5 caracteres.

⇒ **pos** - Posição do vértice do respetivo aeroporto na ***heap***.

⇒ **GMT** - ***Greenwich Mean Time*** referente ao aeroporto. Trata-se apenas de um ***short*** pois o valor é guardado em minutos.

⇒ ***changed*** - Valor ***boolean*** que marca o aeroporto como criado nesta sessão.

Aeroporto 1	**Aeroporto 2**	**Aeroporto 3**	**Aeroporto ...**	**Aeroporto n+1**
char code[5] int pos short GMT bool changed	char code[5] int pos short GMT bool changed	char code[5] int pos short GMT bool changed	char code[5] int pos short GMT bool changed	char code[5] int pos short GMT bool changed
				Aeroporto com o code ='\0'

Fig.5 - *Hastable* Aeroportos 2

3.2 Voos

Como o **número dos voos presentes no sistema** também pode variar, entre **0 e 750 000** decidiu-se, numa tentativa de **reduzir o tempo de leitura, escrever todos os voos** que foram **criados** (e **lidos**) nesta instância e **escrever no fim** um **voo** cujo nome tem um terminador nulo no primeiro caractere (algo que nunca deve acontecer). Deste modo **existe um delimitador** para quando for feita a leitura do ficheiro e para saber quando não existem mais elementos a ler.

Cada **voo** tem 5 variáveis:

⇒ ***name*** - Nome pelo qual cada voo é identificado que pode variar entre 3 a 6 caracteres, sendo o último um terminador nulo fica 7.

⇒ **dep** - Código do aeroporto de partida que pode variar entre 3 a 4 letras maiúsculas, com um último terminador nulo fica 5.

⇒ **dest** - Código do aeroporto de destino que pode variar entre 3 a 4 letras maiúsculas, com um último terminador nulo fica 5.

⇒ ***dep_time*** - Tempo de partida do voo. Este valor é guardado em minutos.

⇒ ***duration*** - Tempo de duração do voo. Este valor é guardado em minutos e uma vez que nunca ultrapassa as 24h (1440 mins) pode ser guardado numa variável do tipo ***short***.

Voo 1	Voo 2	Voo 3	Voo ...	Voo n+1
char name[7] char dep[5] char dest[5] short dep_time short duration	char name[7] char dep[5] char dest[5] short dep_time short duration	char name[7] char dep[5] char dest[5] short dep_time short duration	char name[7] char dep[5] char dest[5] short dep_time short duration	char name[7] char dep[5] char dest[5] short dep_time short duration

Voo com o name ='\0'

Fig.6 - *Hastable* Voos 2

4 Operações

4.1 Primeira Abertura

Os ficheiros referentes às ***hashtables*** de voos e aeroportos são abertos. Caso o programa não consiga abrir o ficheiro em modo de leitura não irá executar as funções ***air_read*** e ***fly_read***.

4.2 Seguintes Aberturas

Abre-se os ficheiros referentes às ***hashtables***, como as estruturas são sempre escritas no fim de execução do programa então faz sentido ler estes ficheiros. Logo são executadas as funções ***air_read*** e ***fly_read***.

4.3 Introdução de um Aeroporto

1 - É alocado um novo espaço para a ***struct*** do tipo ***airport***. Guarda-se o nome e o seu GMT.

2 - Procura-se na ***hashtable*** de aeroportos (***air_hash***).

3 - Caso exista um aeroporto com o mesmo nome é imprimida no ***standard output*** uma mensagem **"+ aeroporto <aeroporto> existe"** e é libertado o ***struct*** alocado no primeiro ponto.

4 - Caso não exista, é inserido o novo aeroporto na ***hashtable***.

5 - É atribuída uma nova chave recorrendo à função ***hashcode*** e procura-se linearmente um lugar vazio.

6 - Insere-se no lugar livre e quando é inserido cria-se também uma nova ***flight_list*** e um novo vértice na mesma chave.

7 - Imprime-se no ***standard output*** a mensagem **"+ novo aeroporto <aeroporto>"**.

A complexidade temporal vai ser de ***O(1)***, no melhor caso, uma vez que a inserção na ***hashtable*** é de custo linear. E complexidade temporal de ***O(n)*** no pior caso pois trata-se de endereçamento fechado.

4.4 Introdução de um Voo

1 - É alocado um novo espaço para a ***struct*** do tipo ***flight***. Guarda-se o seu nome, aeroporto de partida, de destino, o tempo de partida (em minutos) e a duração do mesmo (em minutos).

2 - Procura-se na ***hashtable*** de voos (***fly_hash***).

3 - Caso exista um voo com o mesmo nome é imprimida no ***standard output*** uma mensagem **"+ voo <código voo> existe"** e é libertado o ***struct*** alocado no primeiro ponto.

4 - Caso não exista procura-se então se os aeroportos de partida e chegada existem na ***hashtable*** dos aeroportos.

5 - Caso não exista o primeiro aeroporto imprime-se no ***standard output*** a mensagem **"+ aeroporto <aeroporto de partida> desconhecido"**. Caso não exista o segundo imprime-se a seguinte mensagem **"+ aeroporto <aeroporto de destino> desconhecido"**. Para ambos os casos é libertado o ***struct*** criado no primeiro ponto da memória central.

6 - Adiciona o novo voo à ***flight_list*** do aeroporto de partida e adiciona o novo voo à ***hashtable*** de voos.

7 - Para adicionar à ***flight_list*** é guardado o próximo elemento como o elemento ***header*** da lista e é atualizado o novo ***header*** como o novo elemento adicionado.

8 - Imprime-se no ***standard output*** a mensagem **"+ novo voo <código voo>"**.

Complexidade temporal vai ser de ***O(1)***, no melhor caso, visto que a inserção na ***hashtable*** de voos é linear e a inserção na ***flight_list*** é também de ***O(1)***. No pior dos casos a complexidade temporal é de ***O(n)*** devido à inserção na ***hashtable*** uma vez que se usa um endereçamento fechado.

4.5 Remoção de um Voo

1 - Procura-se o voo na ***hashtable*** de voos pelo nome do voo.

2 - Caso não exista imprime-se no ***standard output*** a seguinte mensagem **"+ voo <código voo> inexistente"**.

3 - Caso exista procura-se o aeroporto de partida do voo na ***hashtable*** de aeroportos e remove-se o voo da lista do mesmo.

4 - Ao remover da ***flight_list*** percorre-se a lista até se encontrar o voo e remove-se o ***list_node*** mantendo as ligações existentes entre o elemento anterior e o próximo.

5 - Imprime-se no ***standard output*** a mensagem **"+ voo <código voo> removido"**.

A complexidade temporal vai ser de ***O(1)***, no melhor caso, isto é, a procura em ambas as ***hashtables*** é de custo constante e o voo é o primeiro na lista de voos de partida. Caso qualquer umas das condições anteriores não se confirmem a complexidade temporal é de ***O(n)***.

4.6 TR

1 - Procura-se o aeroporto de partida na ***hashtable*** de aeroportos e caso não se encontre imprime-se no ***standard output*** a mensagem **"+ aeroporto <aeroporto partida> desconhecido"**.

2 - Procura-se o aeroporto de destino na ***hashtable*** de aeroportos e caso não se encontre imprime-se no ***standard output*** a mensagem **"+ aeroporto <aeroporto destino> desconhecido"**.

3 - Percorre-se a ***hashtable*** de aeroportos e caso exista um vértice o pai deste fica a **NIL** e a distância a **INF** (maior valor de uma variável do tipo ***int***) adiciona-se à ***heap***.

4 - Encontra-se o vértice do aeroporto de início e guarda-se a sua distância com o valor de 0.

5 - Usando a função ***change_value*** uma implementação baseada na função ***decrease key*** que pode ser encontrada no livro de título: **"Introduction to Algorithms"** de Cormen T, Leiserson C, Rivest R, Stein C (2009). Mantém-se a propriedade base de uma ***min-binary-heap*** i.e. o pai é sempre maior que os filhos.

6 - O seguinte ciclo (7-14) é percorrido até a heap estar vazia.

7 - Remove-se o vértice com menor distância **(U)**.

8 - Calcula-se a hora de chegada ao aeroporto representado pelo vértice atual.

9 - Percorre-se cada um dos voos do vértice **(U)**.

10 - Encontra-se o vértice que representa o aeroporto de chegada do voo **(V)**.

11 - Caso o vértice **(U)** seja o vértice inicial então não existe tempo de ligação. Caso não seja o tempo de ligação será de 30 mins.

12 - Se o voo que está a ser tratado tem uma hora de partida antes da hora de chegada ao aeroporto mais o tempo de ligação, então terá que se esperar pelo próximo dia.

13 - Calcula-se o tempo de espera e soma-se o tempo de duração do voo.

14 - Caso a soma da distância do vértice **(U)** mais o tempo calculado no ponto 13 for menor que a distância do vértice **(V)** então a distância do vértice **(V)** passa a ser esse novo valor. É guardado o voo como ***parent*** do vértice **(V)** e atualiza-se o vértice na ***heap***.

15 - Se o ciclo terminar sem que o último vértice seja o vértice final então escreve a seguinte mensagem no ***sandard output*** **"+ sem voos de <aeroporto chegada> para <aeroporto destino>"**.

16 - Caso exista, é então construído o caminho através dos vértices e o voo ***parent***. Esta lista funciona como uma ***stack*** ou seja **LIFO** (***last in first out***) e depois de completa é percorrida e é então impresso no ***sandard output*** o caminho.

A complexidade temporal será de $O(|E| + |V| \log(|V|))$ graças à utilização de uma ***binary heap*** como ***priority queue***.

5 Expansão

Para se tratar de um volume da dados maiores seria necessário implementar ***b-trees*** no lugar de ***hashtables***. Devido à densidade de dados e ao peso acrescido que existiria em memória.

Poderia também ser utilizada uma ***hashtable*** em memória secundária com um ***hashing*** perfeito i.e. cada chave tem um e só um índice. Também usando uma metodologia ***as needed*** de modo a apenas ler da memória secundária o que fosse necessário reduzindo o tempo de acesso.

6 Conclusão

Após a realização deste trabalho foi possível perceber melhor o funcionamente de diversas **estruturas de dados**, algumas das quais implementadas neste projeto, e como estas interagem com a memória e o disco (memória secundária) de uma determinada máquina.

7 Bibliografia

Cormen T, Leiserson C, Rivest R, Stein C (2009) **"Introduction to Algorithms"** Terceira edição MIT Press.

https://en.wikipedia.org/wiki/Dijkstra+27s_algorithm

https://en.wikipedia.org/wiki/Binary_heap

https://en.wikipedia.org/wiki/Min-max_heap

8 Código

— flight_list.h —

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

//Retirado da implementac~ao de Tries das aulas praticas
#define ALPHABET_MIN 'A'
#define ALPHABET_MAX 'Z'
#define ALPHABET_SIZE (ALPHABET_MAX - ALPHABET_MIN + 1)
#define POS(c)  ((c) - ALPHABET_MIN)
#define NUM(c) ((c) - '0')

#define COMMAND_MAX 3
#define AIRPORT 5
#define MAX_AIRPORT 200000
#define MAX_AIR_HASH (MAX_AIRPORT + (MAX_AIRPORT) + 3)
#define AIR_FILENAME "air_hash.bin"

#define FLIGHT_CODE 7
#define MAX_FLIGHT 750000
#define MAX_FLIGHT_HASH (MAX_FLIGHT + (MAX_FLIGHT) + 3)
#define FLY_FILENAME "flight_hash.bin"

#define INF 2147483647
#define HEAP_SIZE (MAX_AIRPORT + 2)
#define PARENT(i) ((i) / 2)
#define LEFT(i) (2 * (i))
#define RIGHT(i) (2 * (i) + 1)

#define CONNECTION_TIME 30
#define MIN_IN_DAY (24 * 60)
#define MIN_IN_H 60
#define TIME(h, m) ((h) * MIN_IN_H + (m))
#define TO_GMT_H(m) ((m) / MIN_IN_H)
#define TO_GMT_M(m) ((m) % MIN_IN_H)


struct flight{
    char name[FLIGHT_CODE];
    char dep[AIRPORT];
    char dest[AIRPORT];
    short dep_time;
    short duration;
};
```

```
struct list_node{
    struct flight* element;
    struct list_node* next;
};


struct flight_list{
    struct list_node* header;
};


/*
------------------------------------------------------------------------
*/


struct list_node* list_node_new(struct flight* flight);
struct flight_list* flight_list_new();


/*------------------------------------------------------------------
*/


bool flight_list_empty(struct flight_list* list);
void flight_list_insert(struct flight_list* list, struct list_node*
new_node);
void flight_list_remove(struct flight_list* list, char name[
FLIGHT_CODE]);
struct list_node* flight_list_pop(struct flight_list* list);


/*
------------------------------------------------------------------------
*/


void flight_list_destroy(struct flight_list* list);
```

— flight_list.c —

```
#include "flight_list.h"


struct list_node* list_node_new(struct flight* flight){
    struct list_node* new = malloc(sizeof(struct list_node));

    if(new != NULL){
        new->element = flight;
```

```
        new->next = NULL;
    }
    return new;
}


struct flight_list* flight_list_new(){
    struct flight_list* new = malloc(sizeof(struct flight_list));

    if(new != NULL)
        new->header = NULL;

    return new;
}


/*
--------------------------------------------------------------------------
*/


bool flight_list_empty(struct flight_list* list){
    return list->header == NULL ? true : false;
}


void flight_list_insert(struct flight_list* list, struct list_node*
new_node){
    new_node->next = list->header;
    list->header = new_node;
}


void flight_list_remove(struct flight_list* list, char name[
FLIGHT_CODE]){
    if(flight_list_empty(list))
        return;

    struct list_node* current = list->header;
    struct list_node* aux;

    //apenas um elemento
    if(current->next == NULL && strcmp(current->element->name, name)
== 0){
        free(current);
        list->header = NULL;
        return;
    }

    //elemento 'e o primeiro da lista
```

```
    if(strcmp(name, current->element->name) == 0){
        list->header = current->next;
        free(current);
        return;
    }

    while(current->next != NULL){

        if(strcmp(current->next->element->name, name) == 0){
            aux = current->next;
            current->next = aux->next;
            free(aux);
            return;
        }

        current = current->next;
    }

    printf("Erro␣a␣remover␣da␣lista\n");
}


struct list_node* flight_list_pop(struct flight_list* list){
    struct list_node* temp = list->header;
    list->header = temp->next;
    return temp;
}


/*
--------------------------------------------------------------------------
*/


void flight_list_destroy(struct flight_list* list){
    if(list == NULL)
        return;

    struct list_node* current = list->header;
    struct list_node* aux = current;

    while(current != NULL){
        current = current->next;
        free(aux);
        aux = current;
    }

    free(list);
}
```

— air_hash.h —

```
#include "flight_list.h"

struct vertice{
    char name[AIRPORT];     //nome do vertice
    struct flight* parent;  //voo pai
    int distance;
};

struct airport{
    char code[AIRPORT];
    int pos;                    //posic~ao do vertice do aeroporto na
 heap
    short GMT;                   //GMT 'e guardado em minutos
    bool changed;
};

struct air_hash{
    struct airport* table[MAX_AIR_HASH];
    struct flight_list* flights[MAX_AIR_HASH];
    struct vertice* vertices[MAX_AIR_HASH];
};


/*
-------------------------------------------------------------------------
*/

struct vertice* vertice_new(char name[AIRPORT]);
struct airport* airport_new(char code[AIRPORT], short GMT);
struct air_hash* air_hash_new();


/*
-------------------------------------------------------------------------
*/


long int air_hashcode(char code[AIRPORT]);
void air_insert(struct air_hash* hashtable, struct airport* air_new);
struct airport* air_search(struct air_hash* hashtable, char code[
AIRPORT]);
long int air_search_index(struct air_hash* hashtable, char code[
AIRPORT]);


/*
-------------------------------------------------------------------------
*/
```

```
void air_read(FILE* air_fp, struct air_hash* air_hash);
void air_write(FILE* air_fp, struct air_hash* air_hash);
```

— air_hash.c —

```
#include "air_hash.h"


struct vertice* vertice_new(char name[AIRPORT]){
    struct vertice* new = malloc(sizeof(struct vertice));

    if(new != NULL){
        memset(new->name, '\0', AIRPORT);
        strcpy(new->name, name);
        new->distance = INF;
        new->parent = NULL;
    }
    return new;
}


struct airport* airport_new(char code[AIRPORT], short GMT){
    struct airport* new = malloc(sizeof(struct airport));

    if(new != NULL){
        memset(new->code, '\0', AIRPORT);
        strcpy(new->code, code);
        new->changed = true;
        new->GMT = GMT;
        new->pos = -1;
    }
    return new;
}


struct air_hash* air_hash_new(){
    struct air_hash* new = malloc(sizeof(struct air_hash));

    if(new != NULL){
        for(int i = 0; i < MAX_AIR_HASH; i++){
            new->table[i] = NULL;
            new->flights[i] = NULL;
            new->vertices[i] = NULL;
        }
    }
    return new;
}
```

```
/*
--------------------------------------------------------------------------
*/


long int air_hashcode(char code[AIRPORT]){
    long int key;

    //caso so' tenha 3 letras
    if(code[3] == '\0'){
        key = POS(code[0]) * ALPHABET_SIZE * ALPHABET_SIZE;
        key += POS(code[1]) * ALPHABET_SIZE;
        key += POS(code[2]);
    }
    else{
        key = POS(code[0] + 1) * ALPHABET_SIZE * ALPHABET_SIZE *
ALPHABET_SIZE;
        key += POS(code[1]) * ALPHABET_SIZE * ALPHABET_SIZE;
        key += POS(code[2]) * ALPHABET_SIZE;
        key += POS(code[3]);
    }

    return key % MAX_AIR_HASH;
}


void air_insert(struct air_hash* hashtable, struct airport* air_new){
    long int key = air_hashcode(air_new->code);

    if(key < 0){
        printf("ERRO␣NO␣HASHING␣DE␣AEROPORTOS␣INSERT\n");
        exit(1);
    }

    key %= MAX_AIR_HASH;

    while(hashtable->table[key] != NULL){
        key++;
        key %= MAX_AIR_HASH;
    }

    //cria logo uma lista de voos de partida e vertice
    air_new->changed = true;
    hashtable->flights[key] = flight_list_new();
    hashtable->vertices[key] = vertice_new(air_new->code);
    hashtable->table[key] = air_new;
}
```

```
struct airport* air_search(struct air_hash* hashtable, char code[
AIRPORT]){
    long int key = -1;
    key = air_hashcode(code);

    if(key < 0){
        printf("ERRO␣NO␣HASHING␣DE␣AEROPORTOS␣SEARCH\n");
        exit(1);
    }

    key %= MAX_AIR_HASH;

    while(hashtable->table[key] != NULL){
        if(strcmp(hashtable->table[key]->code, code) == 0)
            return hashtable->table[key];

        key++;
        key %= MAX_AIR_HASH;
    }
    return NULL;
}


long int air_search_index(struct air_hash* hashtable, char code[
AIRPORT]){
    long int key = air_hashcode(code);

    if(key < 0){
        printf("ERRO␣NO␣HASHING␣DE␣AEROPORTOS␣SEARCH␣INDEX\n");
        exit(1);
    }

    key %= MAX_AIR_HASH;

    while(hashtable->table[key] != NULL){
        if(strcmp(hashtable->table[key]->code, code) == 0)
            return key;

        key++;
        key %= MAX_AIR_HASH;
    }
    return -1;
}


/*
--------------------------------------------------------------------------------
*/
```

```
void air_read(FILE* air_fp, struct air_hash* air_hash){
    if(air_fp == NULL){
        printf("ERRO␣POINTER␣AIR_FP␣NULL.␣AIR_READ\n");
        exit(1);
    }

    struct airport* new_airport;

    fseek(air_fp, 0, SEEK_SET);

    for(int i = 0; i < MAX_AIRPORT; i++){
        new_airport = malloc(sizeof(struct airport));

        if(new_airport == NULL){
            printf("ERRO␣A␣CRIAR␣AEROPORTO.␣AIR_READ\n");
            exit(1);
        }

        memset(new_airport->code, '\0', AIRPORT);
        new_airport->changed = false;
        new_airport->pos = -1;
        new_airport->GMT = 0;

        fread(new_airport, sizeof(*new_airport), 1, air_fp);

        //ultimo aeroporto guardado tem sempre o code[0] = '\0'
        if(new_airport->code[0] == '\0'){
            free(new_airport);
            return;
        }

        air_insert(air_hash, new_airport);
    }
}


void air_write(FILE* air_fp, struct air_hash* air_hash){
    if(air_fp == NULL){
        printf("ERRO␣POINTER␣AIR_FP␣NULL.␣AIR_WRITE\n");
        exit(1);
    }

    fseek(air_fp, 0, SEEK_SET);

    struct airport* current;
    struct airport* blank = malloc(sizeof(struct airport));

    if(blank == NULL){
        printf("ERRO␣A␣CRIAR␣AEROPORTO.␣AIR_WRITE\n");
        exit(1);
```

```
    }

    //guarda todos os aeroportos presentes na hashtable
    for(int i = 0; i < MAX_AIR_HASH; i++){
        current = air_hash->table[i];

        if(current != NULL){
            if(current->changed)
                fwrite(current, sizeof(*current), 1, air_fp);

            free(current);
        }

        if(air_hash->flights[i] != NULL)
            flight_list_destroy(air_hash->flights[i]);

        if(air_hash->vertices[i] != NULL)
            free(air_hash->vertices[i]);
    }

    memset(blank->code, '\0', AIRPORT);
    blank->changed = false;
    blank->GMT = 0;
    blank->pos = -1;

    //guarda o ultimo aeroporto vazio, funciona como um delimitador
    fwrite(blank, sizeof(*blank), 1, air_fp);
    free(blank);
    free(air_hash);
}
```

— flight_hash.h —

```
#include "air_hash.h"

struct fly_hash{
    struct flight* table[MAX_FLIGHT_HASH];
};


/*
--------------------------------------------------------------------------
*/


struct flight* flight_new(char name[FLIGHT_CODE], char dep[AIRPORT],
                          char dest[AIRPORT], short dep_time, short
duration);
struct fly_hash* fly_hash_new();
```

```
/*
--------------------------------------------------------------------------
*/


long int fly_hashcode(char name[FLIGHT_CODE]);
void fly_insert(struct fly_hash* hashtable, struct flight* new_flight
);
struct flight* fly_search(struct fly_hash* hashtable, char name[
FLIGHT_CODE]);
long int fly_search_index(struct fly_hash* hashtable, char name[
FLIGHT_CODE]);



/*
--------------------------------------------------------------------------
*/


void fly_read(FILE* fly_fp, struct fly_hash* hashtable, struct
air_hash* air_hash);
void fly_write(FILE* fly_fp, struct fly_hash* hashtable);
```

— flight_hash.c —

```
#include "flight_hash.h"


struct flight* flight_new(char name[FLIGHT_CODE], char dep[AIRPORT],
                          char dest[AIRPORT], short dep_time, short
duration){

    struct flight* new = malloc(sizeof(struct flight));

    if(new != NULL){
        memset(new->name, '\0', FLIGHT_CODE);
        memset(new->dep, '\0', AIRPORT);
        memset(new->dest, '\0', AIRPORT);

        strcpy(new->name, name);
        strcpy(new->dep, dep);
        strcpy(new->dest, dest);
        new->dep_time = dep_time;
        new->duration = duration;
    }
    return new;
```

```
}


struct fly_hash* fly_hash_new(){
    struct fly_hash* new_hashtable = malloc(sizeof(struct fly_hash));

    if(new_hashtable != NULL){
        for(int i = 0; i < MAX_FLIGHT_HASH; i++)
            new_hashtable->table[i] = NULL;
    }

    return new_hashtable;
}


/*
------------------------------------------------------------------------
*/


long int fly_hashcode(char name[FLIGHT_CODE]){
    long int key;

    key = POS(name[0]) * ALPHABET_SIZE * 9999;
    key += POS(name[1]) * 9999;

    if(name[3] == '\0'){
        key += NUM(name[2]);
        return key;
    }

    if(name[4] == '\0'){
        key += NUM(name[2]) * 10;
        key += NUM(name[3]);
        return key;
    }

    if(name[5] == '\0'){
        key += NUM(name[2]) * 100;
        key += NUM(name[3]) * 10;
        key += NUM(name[4]);
        return key;
    }

    key += NUM(name[2]) * 1000;
    key += NUM(name[3]) * 100;
    key += NUM(name[4]) * 10;
    key += NUM(name[5]);
    return key % MAX_FLIGHT_HASH;
}
```

```
void fly_insert(struct fly_hash* hashtable, struct flight* new_flight
){
    long int key = fly_hashcode(new_flight->name);

    if(key < 0){
        printf("ERRO␣NO␣HASHING␣DE␣VOOS\n");
        exit(1);
    }

    key %= MAX_FLIGHT_HASH;

    while(hashtable->table[key] != NULL){
        key++;
        key %= MAX_FLIGHT_HASH;
    }

    hashtable->table[key] = new_flight;
}


struct flight* fly_search(struct fly_hash* hashtable, char name[
FLIGHT_CODE]){
    long int key = fly_hashcode(name);

    if(key < 0){
        printf("ERRO␣NO␣HASHING␣DE␣VOOS\n");
        exit(1);
    }

    key %= MAX_FLIGHT_HASH;

    while(hashtable->table[key] != NULL){
        if(strcmp(hashtable->table[key]->name, name) == 0)
            return hashtable->table[key];

        key++;
        key %= MAX_FLIGHT_HASH;
    }

    return NULL;
}

long int fly_search_index(struct fly_hash* hashtable, char name[
FLIGHT_CODE]){
    long int key = fly_hashcode(name);

    if(key < 0){
        printf("ERRO␣NO␣HASHING␣DE␣VOOS\n");
```

```
        exit(1);
    }

    key %= MAX_FLIGHT_HASH;

    while(hashtable->table[key] != NULL){
        if(strcmp(hashtable->table[key]->name, name) == 0)
            return key;

        key++;
        key %= MAX_FLIGHT_HASH;
    }

    return -1;
}

/*
------------------------------------------------------------------------
*/


void fly_read(FILE* fly_fp, struct fly_hash* hashtable, struct
air_hash* air_hash){

    if(fly_fp == NULL){
        printf("ERRO A LER FLY_HASH\n");
        exit(1);
    }

    struct list_node* node;
    struct flight* new_flight;
    long int key;
    fseek(fly_fp, 0, SEEK_SET);

    for(int i = 0; i < MAX_FLIGHT; ++i){
        new_flight = malloc(sizeof(struct flight));

        if(new_flight == NULL){
            printf("ERRO A CRIAR NOVO FLIGHT\n");
            exit(1);
        }

        memset(new_flight->name, '\0', FLIGHT_CODE);
        memset(new_flight->dep, '\0', AIRPORT);
        memset(new_flight->dest, '\0', AIRPORT);


        fread(new_flight, sizeof(*new_flight), 1, fly_fp);

        //ultimo voo guardado tem sempre o name[0] = '\0'
```

```
        if(new_flight->name[0] == '\0'){
            free(new_flight);
            return;
        }

        key = air_search_index(air_hash, new_flight->dep);

        if(key < 0){
            printf("ERRO NO HASHING DE AEROPORTOS\n");
            exit(1);
        }

        //insere o voo na lista de partida de voos do aeroporto
        node = list_node_new(new_flight);
        flight_list_insert(air_hash->flights[key], node);
        fly_insert(hashtable, new_flight);
    }
}


void fly_write(FILE* fly_fp, struct fly_hash* hashtable){
    if(fly_fp == NULL){
        printf("ERRO A ESCREVER FLY_HASH\n");
        exit(1);
    }

    struct flight* current;
    struct flight* blank = malloc(sizeof(struct flight));

    if(blank == NULL){
        printf("ERRO A CRIAR VOO\n");
        exit(1);
    }

    memset(blank->name, '\0', FLIGHT_CODE);
    memset(blank->dep, '\0', AIRPORT);
    memset(blank->dest, '\0', AIRPORT);
    blank->duration = 0;
    blank->dep_time = 0;


    fseek(fly_fp, 0, SEEK_SET);

    //guarda todos os voos presentes na hashtable
    for(int i = 0; i < MAX_FLIGHT_HASH; ++i){
        current = hashtable->table[i];

        if(current != NULL){
            fwrite(current, sizeof(*current), 1, fly_fp);
            free(current);
```

```
        }
    }

    //guarda o ultimo voo vazio, funciona como um delimitador
    fwrite(blank, sizeof(*blank), 1, fly_fp);

    free(blank);
    free(hashtable);
    fclose(fly_fp);
}
```

— min_heap.h —

```
#include "flight_hash.h"

/*
*************************************************************************+

+ Implementac~ao de min_binary_heap baseada no livro
                          +
+ escrito por: Cormen T, Leiserson C, Rivest R, Stein C   (2009)
             +
+ "Introduction to Algorithms" página 151.
                               +
*************************************************************************
*/
struct heap{
    struct vertice* array[HEAP_SIZE];
    int size;
};


struct heap* heap_new();


/*-------------------------------------------------------------------
*/



void min_heapify(struct air_hash* hashtable, struct heap* heap, int i
);
void build_min_heap(struct air_hash* hashtable, struct heap* heap);
struct vertice* heap_extract_min(struct air_hash* hashtable,
  struct heap* heap);
void heap_decrease_key(struct air_hash* hashtable, struct heap* heap,
  int i, struct vertice* key);
void min_heap_insert(struct air_hash* hashtable, struct heap* heap,
  struct vertice* key);
```

```
void change_value(struct air_hash* hashtable, struct heap* heap, int
i);
bool heap_empty(struct heap* heap);


/*-------------------------------------------------------------------
*/


void min_heap_destroy(struct heap* heap);

void print_heap(struct heap* heap);
```

— min_heap.c —

```
#include "min_heap.h"


struct heap* heap_new(){
    struct heap* new = malloc(sizeof(struct heap));

    if(new != NULL){
        for(int i = 0; i < HEAP_SIZE; ++i){
            new->array[i] = NULL;
        }
        new->size = 0;
    }

    return new;
}


/*-------------------------------------------------------------------
*/


void swap(struct air_hash* hashtable, struct heap* heap, int i, int j
){
    struct vertice* temp;

    struct airport* airport1 = air_search(hashtable, heap->array[i]->
name);
    struct airport* airport2 = air_search(hashtable, heap->array[j]->
name);
    airport1->pos = j;
    airport2->pos = i;

    temp = heap->array[j];
    heap->array[j] = heap->array[i];
```

```
    heap->array[i] = temp;
}


void min_heapify(struct air_hash* hashtable, struct heap* heap, int i
){
    int l = LEFT(i);
    int r = RIGHT(i);
    int smalest;

    if(l <= heap->size &&
       heap->array[l]->distance < heap->array[i]->distance)
        smalest = l;

    else
        smalest = i;

    if(r <= heap->size &&
       heap->array[r]->distance < heap->array[smalest]->distance)
        smalest = r;

    if(smalest != i){
        swap(hashtable, heap, i, smalest);
        min_heapify(hashtable, heap, smalest);
    }
}


struct vertice* heap_extract_min(struct air_hash* hashtable, struct
heap* heap){
    if(heap->size < 1)
        return NULL;

    struct vertice* min = heap->array[1];
    struct vertice* temp = heap->array[heap->size];

    heap->array[1] = temp;
    heap->array[heap->size] = NULL;

    air_search(hashtable, temp->name)->pos = 1;
    air_search(hashtable, min->name)->pos = -1;

    heap->size--;
    min_heapify(hashtable, heap, 1);
    return min;
}


void heap_decrease_key(struct air_hash* hashtable, struct heap* heap,
 int i, struct vertice* key){
```

```
    heap->array[i] = key;
    air_search(hashtable, key->name)->pos = i;

    while(i > 1 &&
          heap->array[PARENT(i)]->distance > heap->array[i]->distance
){

        swap(hashtable, heap, i, PARENT(i));
        i = PARENT(i);
    }
}


void min_heap_insert(struct air_hash* hashtable, struct heap* heap,
struct vertice* key){
    if(heap->size == 0){
        heap->size++;
        heap->array[heap->size] = key;
        air_search(hashtable, key->name)->pos = heap->size;
        return;
    }
    heap->size++;
    air_search(hashtable, key->name)->pos = heap->size;
    heap_decrease_key(hashtable, heap, heap->size, key);
}


bool heap_empty(struct heap* heap){
    return heap->size == 0 ? true : false;
}


void change_value(struct air_hash* hashtable, struct heap* heap, int
i){
    while(i > 1 &&
          heap->array[PARENT(i)]->distance > heap->array[i]->distance
){

        swap(hashtable, heap, i, PARENT(i));
        i = PARENT(i);
    }
}


void min_heap_destroy(struct heap* heap){
    for(int i = 0; i < HEAP_SIZE; i++){
        heap->array[i] = NULL;
    }
    free(heap);
```

```
}
```

— main.c —

```
#include "min_heap.h"

/*
*****************************************************************************+

+ Abre o ficheiro que guarda a informac~ao da Hashtable relativa aos
        +
+ aeroportos.
                                                                          +
+

+
+ Caso n~ao exista cria e muda o valor do air_first_time
representando        +
+ que foi criado.
                                                                  +
+*****************************************************************************
*/
FILE* open_air_file(FILE* air_fp, bool* air_first_time){
    air_fp = fopen(AIR_FILENAME, "r+");

    if(air_fp == NULL){ //caso n~ao exista
        air_fp = fopen(AIR_FILENAME, "w+");

        if(air_fp == NULL){
            printf("ERRO␣A␣CRIAR␣FICHEIRO␣ESCRITA.␣AIR_HASH.\n");
            exit(1);
        }

        *air_first_time = true;
    }
    return air_fp;
}


/*
*****************************************************************************+

+ Abre o ficheiro que guarda a informac~ao da Hashtable relativa aos
        +
+ voos.

+
+
```

```
+
+ Caso n~ao exista cria e muda o valor do fly_first_time
representando        +
+ que foi criado.
                                                                    +
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*/
FILE* open_fly_file(FILE* fly_fp, bool* fly_first_time){
    fly_fp = fopen(FLY_FILENAME, "r+");

    if(fly_fp == NULL){ //caso n~ao exista
        fly_fp = fopen(FLY_FILENAME, "w+");

        if(fly_fp == NULL){
            printf("ERRO␣A␣CRIAR␣FICHEIRO␣ESCRITA.␣FLY_HASH.\n");
            exit(1);
        }

        *fly_first_time = true;
    }

    return fly_fp;
}


/*
**************************************************************************+

+ Procura se o aeroporto a ser adicionado ao sistema j'a existe. Se
existir   +
+ imprime no standard output que o aeroporto j'a existe.
                    +
+

+
+ Caso n~ao exista adiciona o aeroporto `a hashtable e imprime no
standard      +
+ output uma mensagem a dizer que foi criado.
                                +
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*/
void add_airport(struct air_hash* air_hash, char code[AIRPORT], short
 GMT){
    struct airport* new = airport_new(code, GMT);

    if(air_search(air_hash, code) != NULL){
        printf("+␣aeroporto␣%s␣existe\n", code);
        free(new);
        return;
    }
```

```
    air_insert(air_hash, new);
    printf("+ novo aeroporto %s\n", code);
}


/*
*************************************************************************+

+ Procura se o voo a ser adicionado ao sistema j'a existe. Se existir
        +
+ imprime no standard output que o voo j'a existe.
                            +
+

+
+ Se não existir procura se ambos o aeroporto de chegada e o de
partida       +
+ existem. Se n~ao existirem imprime uma mensagem a dizer q os
aeroportos     +
+ s~ao desconhecidos.
                                                          +
+

+
+ Caso n~ao exista adiciona o voo `a hashtable de voos,  adiciona `a
lista de +
+ voos de partida do aeroporto de partida e imprime no standard
             +
+ output uma mensagem a dizer que foi criado.
                                  +
+************************************************************************
*/
void add_flight(struct air_hash* air_hash, struct fly_hash* fly_hash,
    struct flight* new){

    if(fly_search(fly_hash, new->name) != NULL){
        printf("+ voo %s existe\n", new->name);
        free(new);
        return;
    }

    if(air_search(air_hash, new->dep) == NULL){
        printf("+ aeroporto %s desconhecido\n", new->dep);
        free(new);
        return;
    }

    if(air_search(air_hash, new->dest) == NULL){
        printf("+ aeroporto %s desconhecido\n", new->dest);
```

```
        free(new);
        return;
    }

    long int key = air_search_index(air_hash, new->dep);

    //adiciona `a lista de voos que partem do aeroporto
    struct list_node* node = list_node_new(new);
    flight_list_insert(air_hash->flights[key], node);

    //adiciona `a hashtable de voos
    fly_insert(fly_hash, new);
    printf("+ novo voo %s\n", new->name);
}

/*
*****************************************************************************+

+ Procura se o voo a ser removido existe na hashtable de voos.
               +
+

+
+ Se não existir imprime no standard output que o voo n~ao existe.
           +
+

+
+ Caso exista remove o voo da lista de voos que partem do aeroporto
de        +
+ partida e remove-o tamb'em da hashtable de voos.
                            +
+****************************************************************************
*/
void remove_flight(struct air_hash* air_hash, struct fly_hash*
fly_hash,
    char flight_name[FLIGHT_CODE]){

    struct flight* fligth = fly_search(fly_hash, flight_name);

    if(fligth == NULL){
        printf("+ voo %s inexistente\n", flight_name);
        return;
    }

    long int key = air_search_index(air_hash, fligth->dep);
    long int key2 = fly_search_index(fly_hash, flight_name);

    //remove o voo da lista de voos que partem do aeroporto de
partida
```

```
    flight_list_remove(air_hash->flights[key], flight_name);
    free(fligth); //liberta o voo da hashtable de voos
    fly_hash->table[key2] = NULL;
    printf("+ voo %s removido\n", flight_name);

}


/*
*****************************************************************************+

+ Inicializa todos os v'ertices com parent = NIL e distancia =
infinito.      +
+

+
+ Insere os v'ertices numa lista de prioridades (uma min-heap).
              +
+*****************************************************************************
*/
void initialize_single_source(struct air_hash* air_hash, struct heap*
 heap,
    char start[AIRPORT]){

    struct vertice* new_vertice;
    struct airport* airport;

    for(int i = 0; i < MAX_AIR_HASH; i++){
        new_vertice = air_hash->vertices[i];

        if(new_vertice != NULL){
            new_vertice->parent = NULL;
            new_vertice->distance = INF;

            min_heap_insert(air_hash, heap, new_vertice);
        }
    }

    airport = air_search(air_hash, start);
    new_vertice = heap->array[airport->pos];
    new_vertice->distance = 0;
   change_value(air_hash, heap, airport->pos);
}


/*
*****************************************************************************+

+ implementac~ao do algoritmo de Dijkstra baseada no livro
                  +
```

```
+ escrito por: Cormen T, Leiserson C, Rivest R, Stein C   (2009)
            +
+ "Introduction to Algorithms" página 151.
                                    +
**************************************************************************
*/
struct vertice* dijktra(struct air_hash* air_hash, struct heap* queue
,
                        short dep_time, char start[AIRPORT], char end
[AIRPORT]){

    struct list_node* current_flight;                   //ADJ
    struct vertice* vertice1;                           //U
    struct vertice* vertice2;                           //V

    struct airport* airport1;
    struct airport* airport2;
    struct flight* flight1;

    int GMT_dif;
    int arrival_time;
    int time;                                           //W
    long int key;

    initialize_single_source(air_hash, queue, start);

    airport1 = air_search(air_hash, start);
    short GMT_inicial = airport1->GMT;

    while(!heap_empty(queue)){

        //remove da queue por ordem
        vertice1 = heap_extract_min(air_hash, queue);


        //caso seja o aeroporto de partida a hora de chegada 'e a
dep_time
        if(strcmp(vertice1->name, start) == 0)
            arrival_time = dep_time;


            //c'alculo da hora de chegada ao ve'rtice U
        else{

            if(vertice1->parent == NULL)
                continue;

            airport1 = air_search(air_hash, vertice1->name);
            GMT_dif = airport1->GMT - GMT_inicial;
            arrival_time = (dep_time + vertice1->distance + (GMT_dif)
```

```
);
        }


        //acerto de hora
        if(arrival_time > MIN_IN_DAY)
            arrival_time = (short) (arrival_time % MIN_IN_DAY);

        if(arrival_time < 0)
            arrival_time = (short) (MIN_IN_DAY + arrival_time);


        //ADJ (U)
        key = air_search_index(air_hash, vertice1->name);
        current_flight = air_hash->flights[key]->header;

        while(current_flight != NULL){ //FOR ADJ U
            flight1 = current_flight->element;

            airport2 = air_search(air_hash, flight1->dest);


            if(airport2->pos == -1){ //foi visitado
                current_flight = current_flight->next;
                continue;
            }

            vertice2 = queue->array[airport2->pos]; //V



            //se U for o v'ertice inicial n~ao existe tempo de ligac~
ao
            if(strcmp(vertice1->name, start) == 0)
                time = 0;

                //tempo de ligac~ao
            else
                time = CONNECTION_TIME;


            //o voo 'e depois da hora de chegada
            if(arrival_time + time <= flight1->dep_time){
                time = flight1->dep_time - arrival_time;    //tempo
de espera
                time += flight1->duration;                  //durac~
ao do voo
            }
```

```
                //espera um dia
            else{
                time = (short) (MIN_IN_DAY - arrival_time + flight1->
dep_time);
                time += flight1->duration;
            }


            //RELAX
            if(vertice1->distance + time < vertice2->distance){
                vertice2->distance = vertice1->distance + time;
                vertice2->parent = flight1;
                change_value(air_hash, queue, airport2->pos);
            }

            current_flight = current_flight->next;
        }


        if(strcmp(vertice1->name, end) == 0)
            return vertice1;
    }

    return NULL;
}


/*
*************************************************************************+

+ Constr'oi uma lista ligada com os voos que fazem o percurso
resultante do   +
+ algoritmo de Dijkstra e imprime-o.
                                          +
+************************************************************************
*/
void build_path(struct air_hash* air_hash, struct vertice*
end_vertice){

    struct flight_list* path = flight_list_new();
    struct list_node* node;
    struct vertice* current = end_vertice;
    struct airport* airport1;
    struct airport* airport2;


    //constr'oi o caminho desde o 'ultimo v'ertice at'e ao inicial
por ordem
    //de in'icio para o fim
    while(current->parent != NULL){
```

```
    node = list_node_new(current->parent);
    flight_list_insert(path, node);

    current = air_hash->vertices[air_search_index(air_hash,
        current->parent->dep)];
}


short GMT_dif;
short time, time1_h, time1_m, time2_h, time2_m;


while(!flight_list_empty(path)){
    node = flight_list_pop(path);

    airport1 = air_search(air_hash, node->element->dep);
    airport2 = air_search(air_hash, node->element->dest);

    GMT_dif = airport2->GMT - airport1->GMT;

    time = node->element->dep_time; //hora de partida

    //acerto de hora
    if(time > MIN_IN_DAY)
        time = (short) (time % MIN_IN_DAY);


    time1_h = (short) TO_GMT_H(time);
    time1_m = (short) TO_GMT_M(time);

    if(time1_h < 0)
        time1_m = (short) ((time * -1) % 60);

    if(GMT_dif > 0 && GMT_dif < MIN_IN_DAY)
        GMT_dif %= MIN_IN_DAY;

    //hora de chegada
    time += node->element->duration + (GMT_dif);

    //acerto de hora
    if(time > MIN_IN_DAY)
        time = (short) (time % MIN_IN_DAY);

    if(time < 0)
        time = (short) ((MIN_IN_DAY + time) % MIN_IN_DAY);

    time2_h = (short)TO_GMT_H(time);
    time2_m = (short)TO_GMT_M(time);
```

```
        printf("%-6s␣%-4s␣%-4s␣%02hd:%02hd␣%02hd:%02hd\n", node->
element->name,
            node->element->dep, node->element->dest, time1_h, time1_m
, time2_h, time2_m);
        free(node);
    }

    //liberta a lista da mem'oria
    flight_list_destroy(path);
}


/*
*************************************************************************+

+ C'alcula a viagem com menos tempo de durac~ao entre dois aeroportos
.
+
+ Caso n~ao encontre os aeroportos imprime no standard output
+ que s~ao desconhecidos.
+
+ Se não aplica Dijkstra e imprime o caminho.
+************************************************************************
*/
void TR(struct air_hash* air_hash, struct heap* heap, short dep_time,
    char start[AIRPORT], char end[AIRPORT]){

    if(air_search(air_hash, start) == NULL){
        printf("+␣aeroporto␣%s␣desconhecido\n", start);
        return;
    }

    if(air_search(air_hash, end) == NULL){
        printf("+␣aeroporto␣%s␣desconhecido\n", end);
        return;
    }


    struct vertice* end_vertice;


    //recebe o v'ertice final depois de aplicado dijkstra
    end_vertice = dijktra(air_hash, heap, dep_time, start, end);


    if(end_vertice == NULL){
        printf("+␣sem␣voos␣de␣%s␣para␣%s\n", start, end);
        heap->size = 0;
        return;
    }
```

```
    //caso o vertice final n~ao seja o suposto
    if(strcmp(end_vertice->name, end) != 0) {
        printf("+ sem voos de %s para %s\n", start, end);
        heap->size = 0;
        return;
    }

    //imprime o caminho
    printf("Voo    De    Para Parte Chega\n");
    printf("====== ==== ==== ===== =====\n");
    build_path(air_hash, end_vertice);
    printf("Tempo de viagem: %d minutos\n", end_vertice->distance);

    heap->size = 0;
}

int main(){

    FILE* air_fp = NULL;
    FILE* fly_fp = NULL;
    bool air_first_time = false;
    bool fly_first_time = false;

    air_fp = open_air_file(air_fp, &air_first_time);
    fly_fp = open_fly_file(fly_fp, &fly_first_time);


    char command[COMMAND_MAX];
    char airport_name1[AIRPORT];
    char airport_name2[AIRPORT];
    char flight_name[FLIGHT_CODE];
    short GMT_h, GMT_m, GMT, duration;
    bool flag = true;


    struct air_hash* air_hash = air_hash_new();
    struct fly_hash* fly_hash = fly_hash_new();

    struct heap* heap = heap_new();

    struct flight* flight1;


    if(!air_first_time)
        air_read(air_fp, air_hash);

    if(!fly_first_time)
        fly_read(fly_fp, fly_hash, air_hash);
```

```
    while(flag){
        memset(command, '\0', COMMAND_MAX);
        memset(airport_name1, '\0', AIRPORT);
        memset(airport_name2, '\0', AIRPORT);
        memset(flight_name, '\0', FLIGHT_CODE);
        scanf("%s", command);

        if(strcmp(command, "AI") == 0){
            scanf("%s␣%hd:%hd", airport_name1, &GMT_h, &GMT_m);

            //acerto do GMT para GMT's negativos
            if(GMT_h < 0)
                GMT_m = (short) (GMT_m * -1);

            GMT = (short) (TIME(GMT_h, GMT_m));
            add_airport(air_hash, airport_name1, GMT);
        }


        else if(strcmp(command, "FI") == 0){
            scanf("%s␣%s␣%s␣%hd:%hd␣%hd", flight_name, airport_name1,
 airport_name2, &GMT_h, &GMT_m, &duration);

            //acerto do GMT para GMT's negativos
            if(GMT_h < 0)
                GMT_m = (short) (GMT_m * -1);

            GMT = (short) (TIME(GMT_h, GMT_m));

            flight1 = flight_new(flight_name, airport_name1,
airport_name2, GMT, duration);
            add_flight(air_hash, fly_hash, flight1);
        }

        else if(strcmp(command, "FD") == 0){
            scanf("%s", flight_name);
            remove_flight(air_hash, fly_hash, flight_name);
        }

        else if(strcmp(command, "TR") == 0){
            scanf("%s␣%s␣%hd:%hd", airport_name1, airport_name2, &
GMT_h, &GMT_m);
            GMT = (short) TIME(GMT_h, GMT_m);

            TR(air_hash, heap, GMT, airport_name1, airport_name2);
        }

        else
            flag = false;
```

```
    }

    air_write(air_fp, air_hash);
    fly_write(fly_fp, fly_hash);
    free(heap);
    return 0;
}
```