UNIVERSIDADE DE ÉVORA

Engenharia Informática
Estrutura de Dados e Algoritmos II

Trabalho Prático - 2ªFase

Professor: Vasco Pedro

Sarah Simon Luz 38116
Ana Ferro 39872

12 Maio, Ano Letivo 2019/2020

Índice

1 Estruturas de Dados

1.1 HashTables

A estrutura de dados principal escolhida para a resolução deste trabalho foi uma hash table. Uma hash table guarda os dados num formato de um array, onde cada valor tem o seu índice único, chamado de hash ou hash code, que foi criado através de uma hash function.

Vai ser criada uma hash tables, uma para guardar informação sobre país e uma estrutura semelhante, com uma *hash function* na memória principal que calcula um *"hash code"* que irá representar uma posição de memória no ficheiro, para guardar nós com a informação dos alunos mas também dos países.

Escolhemos esta estrutura de dados devido à facilidade de implementação e também pelas suas primitivas, em média, terem uma complexidade de O(1).

1.1.1 HashTable Alunos

Um aluno quando introduzido no sistema terá de ter: um id único de identificação, que consiste numa sequência de 6 caracteres, de entre letras maiúsculas e algarismos decimais; um código que identifica um país, numa sequência de duas letras maiúsculas; se está activo; se não estiver activo é necessário saber se já terminou ou se abandonou o ensino.

A informação dos alunos irá ser encapsulada numa struct com os respectivos parâmetros: *idaluno*, um array com 7 posições incluindo o carácter nulo; *idpais*, um array com 3 posições incluindo o carácter nulo; quatro booleanos, *ativo*, *concluido*, *abandonou* e *usado* o ultimo foi adicionado posteriormente.

Para saber o espaço máximo que cada struct aluno irá ocupar é feita a seguinte conta:

$$7 + 3 + 1 + 1 + 1 + 1 = 14bytes$$

Visto que a struct é constituída de chars e bools não será feito um alinhamento da memória. Visto que pode haver até 10 000 000 (10 milhões) de alunos introduzidos no sistema, ficará num total de 140 000 000 bytes que em MB serão 140MB.

No entanto, devido às restrições da memória principal apenas são permitidos até 64 MB, não podendo ser possível guardar todas as structs de aluno na memória principal.

Para resolver o problema da falta de espaço na memória principal, decidimos guardar a informação dos alunos num ficheiro em memória secundária que irá funcionar como um array. A posição das struct aluno em memória será atribuída através de um método semelhante à estrutura de dados, hash table. Produzindo um *"hash code"* com o identificador do aluno usando uma *"hash function"*. Sempre que for necessário aceder à informação de um aluno iremos calcular de novo o seu hash code.

Visto que iremos tratar os dados em memória como uma *hash table*, é necessário saber o comprimento da mesma. O comprimento de uma *hash table* tem de ser 1.3 vezes o número máximo de elementos que realmente estarão na tabela e ao mesmo tempo tem de ser um número primo.

Como o número máximo de alunos é 10 000 000 e

$$10000000 * 1.3 = 13000000$$

Com o número primo mais próximo, 13 000 001, o comprimento será 13 000 001. No entanto como tentativa de diminuir o tempo de execução este número foi aumentado para 20000003, diminuindo o número de possíveis colisões.

1.1.2 HashTable Países

Continua a ser necessário guardar a informação de cada país, nomeadamente: identificação do país, número total de alunos, número de alunos ativos, número de alunos que completaram os estudos, número de alunos que abandonaram os estudos e o índice da hashtable em que esse país se encontra. Foi incluído o campo do índice na hashtable pois é fundamental na reconstrução da hashtable.

Esta informação é encapsulada numa struct e cada ocupa:

$$3 + 4 + 4 + 4 + 4 + 4 = 23 + 1(\textit{alinhamento na memória}) = 24\ \textit{bytes}$$

Ao contrário do que acreditávamos e do que foi dito na fase de concepção, não podíamos limitar o número máximo de países a 195. Sendo assim, como não há total certeza do número de países decidimos calcular o número de combinações possíveis do código de identificação de um país e usá-lo como tal.

$$26 * 26 = 676$$

Relativamente ao dimensionamento da hashtable, partindo do princípio que no máximo existe 676 países, optamos por duplicar esse valor, e encontrando o número primo mais próximo ficámos com o tamanho de:

$$676 * 2 = 1352 \rightarrow 1361$$

Com estes valores chegamos à conclusão que, na memória principal, a hashtable países ocupa:

$$1361 * 24 = 32\ 664\ \textit{bytes} < 1MB$$

Atribuímos uma dimensão tão grande, porque não só havia espaço na memória principal mas permite-nos diminuir o número de colisões, diminuindo, consequentemente, o tempo de execução.

A informação vai ser guardada num ficheiro separado da informação dos alunos para poder ficar mais organizada.

2 Ficheiros de dados

Para garantir que a informação é preservada esta deve ser guardada de modo persistente, daí o uso de ficheiros de dados.

Ao contrário do que referimos na fase de concepção, decidimos dividir os dados em dois ficheiros, um que guarda todos os dados relativos aos países e outro que guarda os dados relativos aos alunos.

O ficheiro *"pais.bin"* contém uma lista de nós *país* e o ficheiro *"alunos.bin"* contém uma lista de nós *aluno*.

Na primeira execução do programa é criado dois ficheiros binários vazios. Os nós *país* são escritos de 24 em 24 bytes e os nós *aluno* são escritos de 14 em 14 bytes, cada um no seu respetivo ficheiro.

Figura ilustrativa do conteúdo do ficheiro *"pais.bin"*

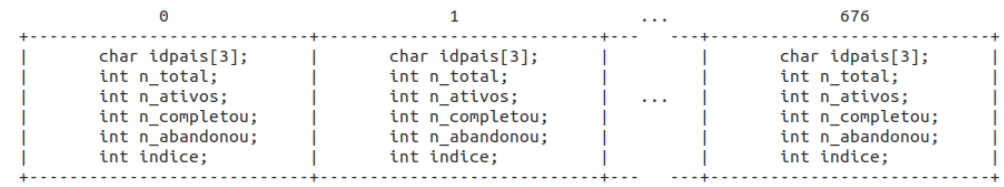
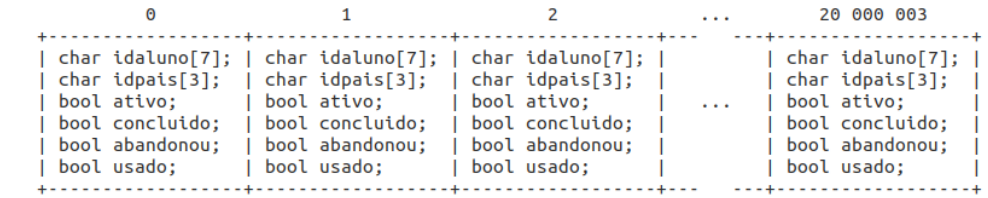


Figura ilustrativa do conteúdo do ficheiro *"alunos.bin"*



Neste caso, o tamanho dos ficheiros é constante visto que retém dados de duas lista estáticas, logo conseguimos calcular o máximo de espaço que vai ser utilizado.

No entanto a ordem pela qual os nós se encontram nos ficheiros diverge entre ambos. No ficheiro *"pais.bin"*, os nós são inseridos de acordo com os índices ocupados, não existindo espaços vazios entre eles. Ao contrário do ficheiro *"alunos.bin"*, que contabiliza todos os nós, preenchidos ou não

2.1 Nós país

Cada nó é constituído por:

- *idpais* - Código do país (tipo char);

- *n_total* - Número total de estudantes que frequentaram ou frequentam o ensino superior nesse país (tipo int) ;
- *n_ativo* - Número de estudantes ativos (tipo int);
- *n_completou* - Número de estudantes que completaram o curso (tipo int);
- *n_abandonou* - Número de estudantes que abandonaram o ensino (tipo int).
- *indice* - Posição em que o nó é guardado na hashtable (tipo int).

Feitas as contas cada nó *país* ocupa

$$4 + 4 + 4 + 4 + 3 + 4 = 23\ bytes$$

aos quais lhe acrescentamos 1 byte por questões de alinhamento, perfazendo um total de 24 bytes. Multiplicando pelo máximo número de elementos (especificado na secção 1.1.2) obtemos

$$676 * 24 = 16\ 224\ bytes < 1MB$$

2.2 Nós aluno

Cada nó é constituído por:

- *idaluno* - Identificador de aluno (tipo char);
- *idpais* - Código do país (tipo char);
- *ativo* - Indicador de aluno ativo ou desativo (tipo bool);
- *concluido* - Indicador que o aluno acabou o curso (tipo bool);
- *abandonou* - Indicador que o aluno desistiu do curso (tipo bool).
- *usado* - Indicador que a posição está ou já foi usada (tipo bool)

Feitas as contas cada nó *aluno* ocupa

$$7 + 3 + 1 + 1 + 1 + 1 = 14\ bytes$$

Multiplicando pelo máximo número de elementos obtemos

$$20\ 000\ 003 * 14 = 208\ 000\ 042\ bytes$$

que equivale a 208MB, 2/5 do espaço do espaço total disponibilizado para a memória secundária(514MB).

3 Operações

Antes de verificar as operações é necessário falar das flags e dos seus significados. Inicialmente apenas existiam três flags, *ativo*, *concluido* e *abandonou*. mas foi necessário inserir mais uma flag *usado*.

A flag *ativo* diz se o estudante está activo ou não. A flag *concluido* diz se o estudante já terminou ou não. A flag *abandonou* diz se o estudante abandonou ou não. A flag *usado* diz se aquela posição está a ser ou já foi usada. Esta flag foi adiciona no caso das colisões.

Se a flag *ativo* estiver marcada como *false*, pode significar várias coisas. Para saber qual o seu significado é necessário verificar as restantes flags.

1. Está desativado, mas a flag *concluido* está *true*. Significa que concluiu os estudos;

2. Está desativado, mas a flag *abandonou* está *true*. Significa que abandonou os estudos;

3. Está desativado e as outras duas flags também, significa que o aluno foi removido do sistema. E o seu identificador poderá ser reutilizado.

No entanto a flag ativo não tem nenhum "efeito"no significado da flag *usado* tal como nas outras flags. Quando uma struct é inserida em memória esta flag é marcada como true e quando o aluno é removido do sistema a flag mantem-se como true.

Isto é necessário porque inicialmente para identificar que um aluno foi removido era necessário verificar se todas as flags estavam marcadas como false podendo ser inserido dois identificadores iguais. Isto acontece quando ao calcular o hashcode de um identificador (idA) colide com outro identificador (idB). É feito o double hashing no idA e colocado noutra posição de memoria, se estiver vazia. Mas, quando removíamos o idB e tentavamos inserir ou procurar pelo idA, ou inseria de novo ou não encontrava o identificador. Para resolver este "bug", adicionámos a flag usado que indica se há uma struct na posição calculada ou já esteve, caso true ou se está vazio caso false. Esta flag facilita a procura porque diz onde é seguro parar a procura do indice do identificador em questão.

3.1 Introduzir um novo estudante

É usada a função `insere_aluno()`. Dados o identificador de um estudante e o código de um país, a inserção de um novo estudante no ficheiro é feita executando os seguintes passos:

1. Calcular o *"hash code"* do identificador do aluno e inicializar uma struct aluno.

2. Procurar no ficheiro a posição calculada pela *"hash function"* colocando a informação dentro da struct aluno inicializada, verificando se a flag *usado* está definida como true:

   2.1. Em caso afirmativo, pode haver três opções possíveis:

   2.1.1. Os identificadores são iguais mas é possível reutilizar, é então encontrada uma posição vaga;

   2.1.2. Os identificadores são iguais mas não é possível reutilizar. Neste caso, não é possível adicionar o aluno e é devolvida a seguinte mensagem:

   $$+ \textit{ estudante } < \textit{identificador} > \textit{existe}$$

   2.1.3. O identificador é diferente. Neste caso, irá ser feito um *"double hashing"* do identificador do aluno e os passos serão repetidas até chegar a uma passo terminal.

   2.2. Em caso negativo, significa que a posição calculada está vaga.

3. Quando encontrada uma posição vaga, a struct aluno criada previamente é preenchida com os dados do aluno e adicionado no ficheiro na posição calculada, também serão alterados os dados do país correspondente, mas apenas na memória principal, sendo criada uma struct para esse mesmo país caso ainda não tenha sido feito e colocado na memória principal com o *hash code* calculado, ou *double hashing* até encontrar uma posição, se necessário. Não é devolvida nenhuma mensagem.

Apesar de ser possível reutilizar identificadores de alunos que foram removidos, a posição onde o aluno que foi removido não pode ser reutilizado por outros identificadores que não sejam iguais ao removido. Estando essa posição de memória "reservada"para um identificador idêntico .

3.2 Remover um identificador ou assinar que um estudante Terminou ou que Abandonou

Estas três operações são todas efetuadas da mesma forma, portanto decidimos juntá-las numa só função com o auxílio de outras para ser mais eficiente. São usadas as funções `operacao_aluno()`, `fazer_operacao()` e `procurar_mem()`.

1. Uma struct aluno é inicializada e é feita a chamada da função `procurar_mem()`. Esta função irá receber o identificador do aluno que é pretendido efectuar as operações e a struct aluno que foi criada. No final devolve o índice onde o identificador está guardado e a informação do aluno correspondente ao identificador estará guardada na struct que foi passada como argumento. Ou então devolve -1 caso o identificador não tenha sido encontrado

2. Ao receber o valor devolvido pela função `procurar_mem()`, é verificado se o índice é maior que 0:
   2.1. Em caso negativo significa que não foi encontrado nenhum aluno com o identificador pedido e é devolvida a seguinte mensagem:

   $$+ \textit{estudante} < \textit{identificador} > \textit{inexistente}$$

   2.2. Em caso afirmativo, vai ser verificado se o aluno está ativo, mais especificamente se já foi marcado que terminou ou abandonou o curso. Se terminou é mandada a seguinte mensagem:

   $$+ \textit{estudante} < \textit{identificador} > \textit{terminou}$$

   Se abandonou é mandada a seguinte mensagem:

   $$+ \textit{estudante} < \textit{identificador} > \textit{abandonou}$$

   E a execução da função termina.

   Se ainda estiver ativo é verificado que tipo de operação é necessário ser efetuada. Isto é feito verificando que número foi passado como argumento para a função `operacao_aluno()` quando foi chamada. Existem três opções:

   2.2.1. 5: Remover.

   2.2.2. 6: Terminou.

   2.2.3. 7: Abandonou.

   Em todos estes casos é chamada a função `fazer_operacao()`. Que

desativa a flag ativo, mudando para *false* e vê que operação tem de ser efetuada. Se for para remover, marca todas as flags para *false* menos a *usado*. Se for para assinalar que terminou, marca a flag *concluido* como *true*. Se for para assinalar que abandonou, marca a flag *abandonou* como *true*. Actualiza a struct aluno que está guardada em disco. Faz as atualizações necessárias em memória principal ao país correspondente, criando uma nova entrada se esse país não existir com o auxilio da função `atualizar`$_i nfo_p aises()$.

3.3 Obter os dados de um país

Dado o código de um país, o sistema mostra o número total de estudantes que frequentaram ou frequentam o ensino superior nesse país, o número de estudantes ativos, o número de estudantes que completaram o curso e o número de estudantes que abandonaram o ensino.

Sempre que um aluno é introduzido ou alterações são feitas a um aluno, os dados do país onde esse aluno estuda também são alterados na memória principal. Para obter esses dados, são feitos os seguintes passos:

1. Calcular o *hash code* do país;
2. Aceder à *hash table* com o *hash code* e ver se a entrada corresponde ao país:
   2.1. Se o identificador corresponder ao país recebido, aceder aos dados desejados e devolve na consola;
   2.2. Se não corresponder, repetir o processo até encontrar o país desejado, depois aceder aos dados desejado, devolvendo-os na consola;
   2.3. Se a entrada estiver vazia. Significa que o país não existe em memória e devolve a mensagem:
   $$+ \textit{ sem dados sobre } < \textit{codigo} >$$

Nas situações 2.1 e 2.2 os dados de saída encontram-se no formato:

$$+ < \textit{codigo} > \textit{- correntes: } < C >, \textit{ diplomados: } < D >, \textit{ abandonaram: } < A >, \textit{ total: } < T >$$

Onde <C >é o número de estudantes ativos no país, <D >é o número de estudantes no país que completaram o curso, <A >é o número de estudantes no país que abandonaram o sistema de ensino e <T >é o número total de estudantes, no sistema, associados ao país dado.

3.4 Complexidade das operações

A complexidade destas operações é, em média, O(1), no entanto, este valor irá depender do número de colisões.

No pior dos casos, para a inserção de alunos, percorremos a tabela toda até encontrar um espaço vazio ou quando encontramos um espaço que já foi previamente usado pelo identificador que queremos inserir e que se encontra em condições de ser reutilizado. Para as restantes operações, percorremos a tabela até encontrar a posição correta, o que implica continuar a procurar mesmo quando se depara com uma posição que pode ser reutilizada. Para ambos os casos, vai corresponder a uma complexidade de 0(n).

Com o uso do algoritmo de hash *djb2*, e do tratamento de colisões (double hashing) conseguimos uma boa distribuição dos dados e, consequentemente, uma diminuição de colisões.

3.5 Acessos ao disco

Considerando que a leitura e escrita correspondem a um acesso ao disco cada uma, no melhor caso, isto é quando não existe colisões, cada operação à exceção da "Obter dados de um país"acede duas vezes ao disco, uma para verificar se a posição é a correta (leitura) e outra para realizar as alterações necessárias (escrita).

No pior caso, isto é, quando há colisões, cada operação acede $n+2$ vezes, sendo n o número de colisões e, consequente, leituras necessárias para verificar que se encontrou a posição correta, mais 2 acessos, já na posição certa, para verificação(leitura) e alteração de dados(escrita).

A operação "Obter dados de um país"não faz acessos ao disco, mas está diretamente ligada à hashTable paises, que é carregada do disco para a memória principal cada vez que o programa é executado e carregada de volta ao disco no final da sua execução. Sabemos então que o número de acessos depende do número de nós *pais* preenchidos, fazendo com que o número de acessos seja n, limitado entre 1 e 676(máximo número de países).

4 Início e fim da execução

No início da execução do programa, são abertos dois ficheiro, caso não existem, são criados.

Se o ficheiro *"pais.bin"* existe, este é lido na sua totalidade. Esta leitura permite reconstruir a hash table países que passa a ficar carregada em memória principal. Ao guardar estes dados em memória principal permitenos diminuir o tempo de execução, evitando acessos desnecessários à memória secundário.

No final da execução do programa, é necessário atualizar o ficheiro *"pais.bin"*, escrevendo nele todas as posições preenchidas da hashtable paises. De seguida, os ficheiros são fechados, visto que todas as alterações relativamente aos nós *aluno* são feitas em disco ao longo do programa.

5 Bibliografia

- Hashfunctions -djb2
  http://www.cse.yorku.ca/ oz/hash.html
  13 junho, 15:42

- HashTable runtime complexity
  https://stackoverflow.com/questions/9214353/hash-table-runtime-complexity-insert-search-and-delete
  13 junho, 15:44

- Double Hashing
  https://www.geeksforgeeks.org/double-hashing/
  13 junho, 15:47

- Data Structure Alignment
  https://en.wikipedia.org/wiki/Data$_s$*tructure*$_a$*lignment*
  $13 junho, 20:02$

6 Anexos

6.1 Código do programa

6.1.1 Ficheiro main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "disco.h"

#define BDNOMEFICHEIROALUNO "alunos.bin"
#define BDNOMEFICHEIROPAIS "pais.bin"
#define MAX_ID_ALUNO 7
#define MAX_ID_PAIS 3

#define MAX_TAMANHO_PAISES 1361

int main(void)
{
    //codigo, codigo pais
    char operacao, identificador[MAX_ID_ALUNO], codigo[MAX_ID_PAIS];

    //Abre o ficheiro
    FILE *ficheiro_alunoBD = openFile(BDNOMEFICHEIROALUNO);
    FILE *ficheiro_paisBD = openFile(BDNOMEFICHEIROPAIS);

    struct paises *ht_paises = criar_hashTable();
    //obter a informacao dos paises que esta' guardada em disco e
        guarda na hashtable criada
    obter_info_paises(ficheiro_paisBD, ht_paises);

    while(scanf("%c", &operacao) != EOF)
    {
        switch(operacao)
        {
            case 'I':
                scanf("%s %s", identificador, codigo);
                insere_aluno(ficheiro_alunoBD, ht_paises, identificador,
                                                        codigo);
                break;
```

```
            case 'R':
                scanf("%s", identificador);
                operacao_aluno(ficheiro_alunoBD, ht_paises, identificador,5);
                break;

            case 'T':
                scanf("%s", identificador);
                operacao_aluno(ficheiro_alunoBD, ht_paises, identificador,6);
                break;

            case 'A':
                scanf("%s", identificador);
                operacao_aluno(ficheiro_alunoBD, ht_paises, identificador,7);
                break;

            case 'P':
                scanf("%s", codigo);
                obter_dados(ht_paises, codigo);
                break;

            case 'X':
                atualizar_BD(ficheiro_paisBD, ht_paises);
                fclose(ficheiro_paisBD);
                fclose(ficheiro_alunoBD);
                return 0;
                break;

            default:
            continue;
        }
    }
    return 0;
}
```

6.1.2 Ficheiro disco.h

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

#include "hashtable.h"

struct aluno
{
    char idaluno[7];
    char idpais[3];
    bool ativo;
    bool concluido;
    bool abandonou;
    bool usado;
};

struct aluno *criar_aluno();
struct aluno *adicionar_info(struct aluno *temp,char *identificador,
                                char *codigo);
FILE *openFile(const char *filename);
int ler_aluno(FILE *ficheiro,int indice, struct aluno *temp );
void escrever_aluno(FILE *ficheiro,int indice, struct aluno *temp);

int procurar_mem(FILE *ficheiro, struct aluno *temp, char *identificador);
void insere_aluno(FILE *ficheiro, struct paises *ht, char *identificador,
                                char *codigo);
void fazer_operacao(FILE *ficheiro, struct paises *ht, int indice,
                                struct aluno *temp, int modo);
void operacao_aluno(FILE *ficheiro, struct paises *ht, char *identificador,
                                int modo);

void obter_info_paises(FILE *ficheiro, struct paises *paises);
void atualizar_BD(FILE *ficheiro, struct paises *paises);
```

6.1.3 Ficheiro disco.c

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

#include "disco.h"

#define MAX_TAMANHO_PAISES 1361
#define MAX_TAMANHO_ALUNOS 20000003
#define PRIMO_PAIS 7
#define PRIMO_ALUNO 3

//aloca o espaco necessario para uma struct aluno e devolve o pointer
        da memoria
struct aluno *criar_aluno()
{
    struct aluno *temp = malloc( sizeof(struct aluno));

    strcpy(temp->idaluno, "");
    strcpy(temp->idpais, "");
    temp->ativo = false;
    temp->concluido = false;
    temp->abandonou = false;
    temp->usado = false;
    return temp;
}

//dado um pointer de uma struct aluno, vai definir o identificador, codigo
        do pais e estabelecer as flags.
//devolve o pointer para a struct aluno
struct aluno *adicionar_info(struct aluno *temp, char *identificador,
        char *codigo)
{
    strcpy(temp->idaluno, identificador);
    strcpy(temp->idpais, codigo);
    temp->ativo = true;
    temp->concluido = false;
    temp->abandonou = false;
    temp->usado = true;
```

```
    return temp;
}

//abre o ficheiro, caso nao exista um, vai cria-lo
FILE *openFile(const char *filename)
{
    FILE *temp = fopen(filename, "rb+");

    if(temp == NULL)
    {
        temp = fopen(filename, "wb+");
    }

    return temp;
}

//le do ficheiro a informacao de uma struct aluno, a sua posicao e'calculada
        com o auxilio do indice.
//devolve 0 se nenhum aluno foi lido com sucesso, devolve 1 quando um aluno
        e'lido com sucesso
int ler_aluno(FILE *ficheiro,int indice, struct aluno *temp )
{
    int ret;
    fseek(ficheiro, indice*sizeof(struct aluno), SEEK_SET);
    ret = fread(temp, sizeof(struct aluno), 1, ficheiro);

    return ret;
}

//escreve no ficheiro a informacao de um aluno, a sua posicao e' calculada com
        o auxilio de um indice
void escrever_aluno(FILE *ficheiro,int indice, struct aluno *temp)
{
    fseek(ficheiro, indice*sizeof(struct aluno), SEEK_SET);
    fwrite(temp, sizeof(struct aluno), 1, ficheiro);
}



//vai procurar no ficheiro a posicao da struct aluno dado o seu identificador.
//se nao encontrar devolve -1, se encontrar devolve o indice
```

```
int procurar_mem(FILE *ficheiro, struct aluno *temp, char *identificador)
{
    //obtem o hashcode do identificador que está a tentar encontrar
    int indice = hashCode1(identificador, MAX_TAMANHO_ALUNOS);
    int contador = 1;

    ler_aluno(ficheiro, indice, temp);

    //procurar aluno
    while(temp->usado == true)
    {
        //compara o identificador encontrado com o identificador dado para ver
                se encontrou o aluno procurado
        if(strcmp(temp->idaluno, identificador)==0)
            return indice;

        //calcular o novo indice
        indice = double_hashing(contador, identificador,MAX_TAMANHO_ALUNOS,
                                PRIMO_ALUNO);
        contador++;

        ler_aluno(ficheiro, indice, temp);
    }

    //nao foi encontrado aluno
    return -1;
}

//inserir aluno novo, criando um indice a partir do seu identificador
void insere_aluno(FILE *ficheiro, struct paises *ht, char *identificador,
        char *codigo)
{
    struct aluno *temp = criar_aluno();
    int indice = hashCode1(identificador, MAX_TAMANHO_ALUNOS);
    int contador = 1;
    int ret;

    ret = ler_aluno(ficheiro, indice, temp);

    //encontrar uma vaga para introduzir o aluno, ou ver se ja' existe
    while(temp->usado == true && ret!=0)
```

```
    {
        //ja existiu um aluno nesta posicao, mas foi removido, ou seja,
                foi encontrada uma vaga

        if(temp->ativo==false && temp->concluido==false &&
            temp->abandonou==false && strcmp(temp->idaluno, identificador)== 0)
        {
            //inserir novo aluno
            atualizar_info_paises(ht, codigo, 0);
            escrever_aluno(ficheiro,indice,adicionar_info(temp,identificador,
                                                          codigo));
            free(temp);
            return;
        }

        //ja' existe um aluno com este identificador
        if(strcmp(temp->idaluno, identificador)==0)
        {
            printf("+ estudante %s existe\n", identificador);
            free(temp);
            return;
        }

        //calcular o novo indice
        indice = double_hashing(contador, identificador,MAX_TAMANHO_ALUNOS,
                                    PRIMO_ALUNO);
        contador++;

        ret = ler_aluno(ficheiro, indice, temp);

    }

    //inserir novo aluno
    atualizar_info_paises(ht, codigo, 0);
    escrever_aluno(ficheiro,indice,adicionar_info(temp,identificador,codigo));

    free(temp);
}
```

```
//operacoes: 5, remove o aluno; 6, marca o aluno como terminou;
        7, marca o aluno como abandonou
void fazer_operacao(FILE *ficheiro, struct paises *ht, int indice,
        struct aluno *temp, int modo)
{
    temp->ativo = false;
    switch(modo)
    {
        //remover
        case 5:
            temp->concluido = false;
            temp->abandonou = false;
            break;

        //terminou
        case 6:
            temp->concluido = true;
            break;

        //abandonou
        case 7:
            temp->abandonou = true;
            break;

        default:
            break;
    }

    atualizar_info_paises(ht, temp->idpais, modo);
    escrever_aluno(ficheiro, indice, temp);
}

//ve que operacao foi pedida e exetua o pedido
void operacao_aluno(FILE *ficheiro, struct paises *ht, char *identificador,
        int modo)
{
    struct aluno* temp = criar_aluno();
    int indice = procurar_mem(ficheiro, temp, identificador);

    if(indice > 0)
    {
```

```
        //aluno esta' desativado
        if(temp->concluido == true)
        {
            printf("+ estudante %s terminou\n", identificador);
            free(temp);
            return;
        }

        //aluno esta' desativado
        if(temp->abandonou == true)
        {
            printf("+ estudante %s abandonou\n", identificador);
            free(temp);
            return;
        }

        //verifica que operacao ira' correr
        if(temp->ativo == true)
        {
            fazer_operacao(ficheiro, ht, indice, temp, modo);
            free(temp);
            return;
        }
    }

    printf("+ estudante %s inexistente\n", identificador);
    free(temp);
}

//vai "buscar" ao ficheiro a informacao dos paises
void obter_info_paises(FILE *ficheiro, struct paises *ht)
{
    struct pais *aux = malloc(sizeof(struct pais));

    while(fread(aux,sizeof(struct pais),1,ficheiro) != 0)
    {
        struct pais *temp = malloc(sizeof(struct pais));

        strcpy(temp->idpais, aux->idpais);
        temp->n_total = aux->n_total;
        temp->n_ativos = aux->n_ativos;
```

```
        temp->n_completou = aux->n_completou;
        temp->n_abandonou = aux->n_abandonou;
        temp->indice = aux->indice;

        ht->tabela[temp->indice] = temp;
    }

    free(aux);
}

//vai atualizar a informacao que esta' guardada em disco com as alteracoes
        feitas durante a execucao
void atualizar_BD(FILE *ficheiro, struct paises *ht)
{
    int contador = 0;

    for(int i = 0; i < MAX_TAMANHO_PAISES; i++)
    {
        if(ht->tabela[i] != NULL)
        {
            fseek(ficheiro, sizeof(struct pais) * contador, SEEK_SET);
            fwrite(ht->tabela[i], sizeof(struct pais), 1, ficheiro);
            contador++;
        }
    }

    free_hashtable(ht);
}
```

6.1.4 Ficheiro hashtable.h

```
#include <stdbool.h>

#define MAX_TAMANHO_PAISES 1361

struct pais
{
    char idpais[3];
    int n_total;
    int n_ativos;
    int n_completou;
    int n_abandonou;
    int indice;

};

struct paises{
    struct pais *tabela[MAX_TAMANHO_PAISES];
};

struct paises *criar_hashTable();
struct paises *novo_pais();
void free_hashtable(struct paises *ht);
int procurar(struct paises *ht, char *codigo);
int inserir(struct paises *ht, char *codigo, int total, int ativos,
        int completou, int abandonou);
int hashCode(char *codigo);
unsigned int hashCode1(char *codigo, int tamanho);
int hashCode2(char *codigo, int tamanho, int primo);
int double_hashing(int contador, char *identificador, int tamanho, int primo);
void obter_dados(struct paises *ht, char *codigo);
void atualizar_info_paises(struct paises *ht, char *codigo, int modo);
```

6.1.5 ficheiro hashtable.c

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

#define MAX_TAMANHO_PAISES 1361
#define PRIMO_PAIS 7

//cria uma hashtable e devolve o pointer para a mesma hashtable
struct paises *criar_hashTable()
{
    struct paises *novo = malloc(sizeof(struct paises));
    if(novo == NULL)
        return NULL;

    for(int i=0; i < MAX_TAMANHO_PAISES; i++)
        novo->tabela[i] = NULL;

    return novo;
}

//recebe uma hashtable e depois da free
void free_hashtable(struct paises *ht)
{
    for(int i = 0; i< MAX_TAMANHO_PAISES; i++)
        free(ht->tabela[i]);
    free(ht);
}

//recebe um pointer de uma hashtable e o codigo do pais que esta' a
        procurar. se o pais for encontrado devolve o indice,
        caso contrario devolve -1
int procurar(struct paises *ht, char *codigo)
{
    int indice = hashCode1(codigo, MAX_TAMANHO_PAISES);
    int contador = 1;

    while(ht->tabela[indice]!= NULL)
```

```
    {
        if(strcmp(ht->tabela[indice]->idpais,codigo) == 0)
            return indice;


        indice = double_hashing(contador, codigo, MAX_TAMANHO_PAISES,
                                    PRIMO_PAIS);
        contador++;
    }

    return -1;
}

//recebe um pointer de uma hashtable, o codigo do pais que quer inserir
        e informacao que tem de adicionar. devolve o indice onde o pais
        foi adicionado
int inserir(struct paises *ht, char *codigo, int total, int ativos,
        int completou, int abandonou)
{
    int indice = hashCode1(codigo,MAX_TAMANHO_PAISES);
    int contador = 1;

    while(ht->tabela[indice] != NULL && strcmp(ht->tabela[indice]->idpais,
            codigo) != 0)
    {
        indice = double_hashing(contador, codigo, MAX_TAMANHO_PAISES,
                                    PRIMO_PAIS);
        contador++;
    }

    struct pais *temp = malloc(sizeof(struct pais));

    strcpy(temp->idpais, codigo);
    temp->n_total = total;
    temp->n_ativos = ativos;
    temp->n_completou = completou;
    temp->n_abandonou = abandonou;
    temp->indice = indice;

    ht->tabela[indice] = temp;
```

```
    return indice;
}

// Algoritmo de hash - djb2
//Retirado de http://www.cse.yorku.ca/~oz/hash.html
unsigned int hashCode1(char *codigo, int tamanho)
{
    unsigned int hash = 5381;
    int c;

    while ((c = *codigo++))
        hash = ((hash << 5) + hash) + c;

    return hash % tamanho;
}

int hashCode2(char *codigo, int tamanho, int primo)
{
    return primo - ((hashCode1(codigo, tamanho)) % primo);
}

//em caso de colisao, double hashing do identificador
int double_hashing(int contador, char *identificador, int tamanho, int primo)
{
    return (hashCode1(identificador, tamanho) + contador *
                hashCode2(identificador, tamanho, primo)) % tamanho;
}

//obter os dados de um dado pais com o codigo do mesmo
void obter_dados(struct paises *ht, char *codigo)
{
    int indice = procurar(ht, codigo);

    if (indice == -1 || ht->tabela[indice]->n_total == 0)
        printf("+ sem dados sobre %s\n", codigo);

    else
        printf("+ %s - correntes: %d, diplomados: %d, abandonaram: %d,
                total: %d\n", codigo, ht->tabela[indice]->n_ativos,
                ht->tabela[indice]->n_completou,
                ht->tabela[indice]->n_abandonou,
```

```
                ht->tabela[indice]->n_total);
}

//atualiza a informacao de um pais depois de terem sido feitas operacoes.
        caso o pais ainda nao tenha sido inserido, e' inserido
void atualizar_info_paises(struct paises *ht, char *codigo, int modo)
{
    int indice = procurar(ht, codigo);

    if(indice == -1)
        indice = inserir(ht, codigo,0,0,0,0);

    switch(modo)
    {
        case 0:
            ht->tabela[indice]->n_total += 1;
            ht->tabela[indice]->n_ativos += 1;

            break;
        //remover
        case 5:
            ht->tabela[indice]->n_total -= 1;
            ht->tabela[indice]->n_ativos -= 1;

            break;
        //completou
        case 6:
            ht->tabela[indice]->n_ativos -= 1;
            ht->tabela[indice]->n_completou += 1;
            break;
        //abandonou
        case 7:
            ht->tabela[indice]->n_ativos -= 1;
            ht->tabela[indice]->n_abandonou += 1;
            break;

        default:
            break;
    }
}
```