Relatório da 2ªFase do Trabalho Prático de Estruturas de Dados e Algoritmos II

Diogo Solipa(l43071) Leonardo Catarro(l43025)

Problema a ser avaliado no Mooshak

O problema a ser avaliado é o T: Total.

Alterações efetuadas na 2ª fase

As alterações efetuadas basearam se no facto de termos operações com custo linear mais complexas que o necessário, quando poderíamos fazer o mesmo com um custo constante.

A primeira grande mudança que fizemos foi no número de ficheiros, pois reduzimos para apenas 2 ficheiros, um para os estudantes e outro para guardar os dados dos países. A primeira ideia de 1 ficheiro por país devia-se a uma má interpretação onde pensamos que na operação obter dados de um país queríamos os nomes e todos os dados de todos os estudantes desse país e não apenas os números.

A segunda grande alteração foi às estruturas de dados. Passamos a ter zero estruturas de dados em memória central e duas hashtables em disco, 1 para cada ficheiro binário.

Depois as operações apenas adaptámos a estas alterações. No caso da inserção, caso esta fosse bem-sucedida e o país não se encontrar presente no ficheiro dos países então criávamos esse país e atualizávamos os seus dados. Caso ele já existisse bastava atualizar os dados desse país no ficheiro.

A remoção ficou muito parecida com a ideia inicial, a diferença foi que em vez de reescrevermos todo o ficheiro de novo para efetivamente apagar um estudante, usámos um marcador para saber se este se encontrava presente na tabela ou não.

Em relação ao abandono/terminar o curso, no ficheiro dos estudantes nada mudou, o que acontece de diferente é em vez de escrevermos o estudante no ficheiro do seu país, apenas atualizamos os dados no ficheiro dos países referente ao país do estudante.

A operação que mais alterações sofreu foi a obter dados de um país. Tal como descrito acima, uma má interpretação levou a uma solução muito mais complicada do que o realmente necessário. Para esta operação usávamos um array em memória central, mas passamos apenas a aceder a um ficheiro e retirarmos os dados de um país específico. Deste modo reduzimos significamente os acessos ao ficheiro e os recursos utilizados pelo programa.

Estruturas de Dados

Descrição

A estrutura que iremos utilizar no trabalho é a Hashtable, esta terá uma resolução de colisões quadrática, enquadrando-se no endereçamento aberto. Vamos ter 1 Hashtable para os países com os alunos ativos, os que abandonaram e os que acabaram.

Razões da escolha

Escolhemos a Hashtable como estrutura de dados, devido à sua eficiência e facilidade de uso. Esta vai facilitar muito comparativamente com uma lista ligada, graças à sua inserção, procura e remoção constante. Vão existir casos em que na pesquisa e inserção haverá colisões, mas a complexidade temporal continuará constante.

Dimensionamento

O dimensionamento da tabela para os estudantes é de aproximadamente 38 Milhões de posições e para a tabela dos países será de perto de 900 mil. De forma a utilizarmos todo o tamanho de disco disponível maximizamos as tabelas para termos o menor número de colisões possíveis em qualquer uma das tabelas, consequentemente reduzindo o número de acessos ao disco.

Ligação entre as estruturas

Não existe.

Localização

Em memória central ou primária, não vamos ter qualquer estrutura de dados. Em memória secundária irão ficar os dois ficheiros com as respetivas Hashtables de estudantes e países.

Descrição dos Elementos das estruturas

A Hashtable do ficheiro dos estudantes será composta por structs do tipo Student e, portanto, cada posição da Hashtable que esteja alocada terá um estudante associado em que a chave é a string id do estudante. Por outro lado, a Hashtable do ficheiro dos países terá structs do tipo Country, a chave desta tabela é o id do país.

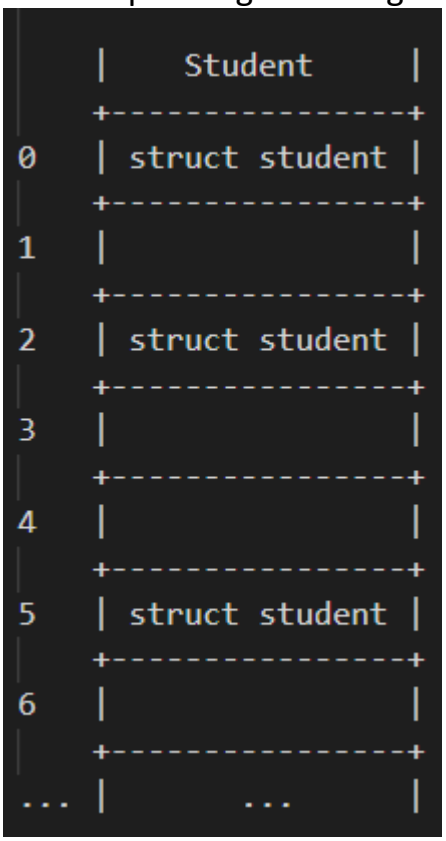
Memória ocupada pelas estruturas em memória central

Como não temos estruturas de dados em memória central, não vamos ter qualquer memória associada a estruturas de dados.

Ficheiros de Dados

Para a realização deste trabalho foram usados dois ficheiros:

→Um ficheiro que contem a informação dos estudantes. Este contém uma Hash Table do tipo student para guardar as informações dos estudantes, representada pelo seguinte diagrama:



A struct student contém os seguintes elementos:

```
typedef struct student
{
    char id[ID_SIZE], p_id[P_ID_SIZE];
    bool quit, finished;
    bool isStudent;
}Student;
```

→char id[ID_SIZE]: string com tamanho 7 para guardar o ID do estudante. Esta string ocupa **7 bytes.**

→char p_id[P_ID_SIZE]: string com tamanho 3 que contem o id do país do estudante. Esta string ocupa **3bytes.**

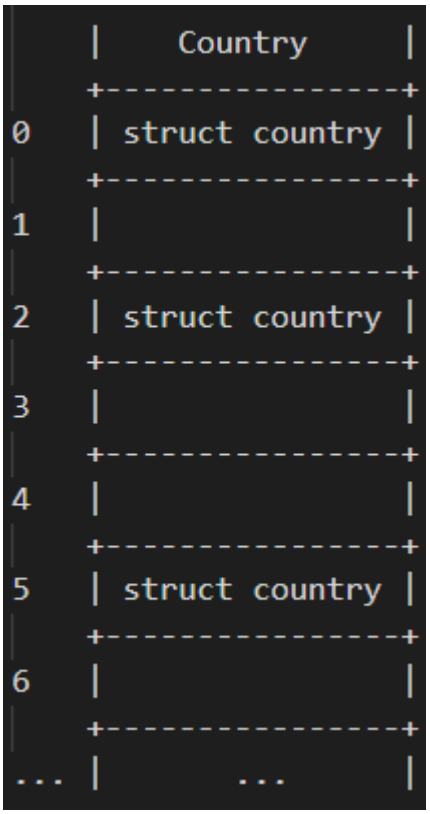
→bool quit, bool finished: booleano para marcar que o estudante saiu e finalizou o curso, respetivamente. Estes booleanos ocupam **1byte** cada.

→bool isStudent: booleano para verificar se o estudante já existe ou não. Este booleano ocupa **1byte**

Tamanho da struct student= 7bytes + 3bytes + 2bytes + 1byte = **13 bytes**

(NOTA: a struct ocupa 13bytes, porém depois com o alinhamento feito pelo C, esta passará a ocupar **16bytes**)

→O outro ficheiro contém a informação dos países. Esta informação será guardada numa Hash Table do tipo country. A HashTable terá a seguinte estrutura:



A struct country tem os seguintes elementos:

```
typedef struct country
{
    char p_id[P_ID_SIZE];
    bool isCountry;
    int active, finished, abandoned, total;
}Country;
```

→char p_id[P_ID_SIZE]: contém o id do país. Esta string ocupa **3bytes**.

→int active, finished, abandoned, total: inteiros que guardam o

número de alunos ativos, que terminaram o curso, que abandoram o curso e o total de alunos nesse país, respetivamente. Estes inteiros ocupam **4bytes** cada.

→bool isCountry: booleano para verificar se esse país já consta da hastable ou não. Este booleano ocupa **1byte.**

Tamanho da struct country = 3bytes + 1byte + (4*4bytes)
= **20bytes**

Baseado nas dimensões das Hash Tables o tamanho máximo dos ficheiros são:

→494MB para o ficheiro dos estudantes;
→9.3MB para o ficheiro dos países;

Operações

Introduzir um novo estudante

O índex onde vamos inserir o estudante na Hashtable será calculado por uma função de hashing para Strings mais especificamente a função djb2 que facilita muito gerar chaves únicas para cada chave evitando algumas colisões e consequentemente menos acessos ao disco.

1 – Pesquisamos o ficheiro dos estudantes e tentamos inserir o estudante, se este já existir na tabela a operação é cancelada. Caso contrário insere-se o estudante. No caso de a primeira posição acedida tiver um elemento removido da tabela ou um elemento ativo que seja diferente da chave do estudante que estamos a inserir, vão se dar colisões.

2 – No caso de a primeira posição acedida pertencer a um elemento removido, mas com um id igual ao do estudante a inserir essa será a posição de inserção. O outro caso é quando nessa mesma primeira posição calculada o id do estudante no ficheiro é igual ao id da chave e o estudante existe, nesse caso a operação é cancelada.

3 – O último caso é quando existem colisões como explicado no 1 ponto. O processo de inserção só irá parar quando encontrar uma posição para inserir, ou seja, uma posição fora dos atuais limites do ficheiro, mas dentro dos limites da tabela ou quando encontrar uma posição com uma struct

Student vazia, em que o ID é apenas o '\0'. O processo é cancelado quando o estudante já existe e, portanto, não será possível encontrar uma posição para inserir o estudante.

4 – Acedemos ao ficheiro, dos países e procuramos pelo país do estudante que introduzimos, se este ainda não existir é criado. No caso de o país já existir apenas atualizamos os números do respetivo país. As colisões são tratadas exatamente como são tratas as colisões no ficheiro dos estudantes. A única diferença é a chave, o tamanho da tabela e o facto de não haver remoções na tabela dos países logo um dos casos das colisões deixa de existir.

Se a chave do estudante a inserir for igual à chave do valor de hash significa que se o estudante não estiver removido, este existe e encontra-se na tabela, a operação terá apenas 2 acessos ao disco. Por outro lado, se o estado do estudante for removido, podemos inserir nessa mesma posição inicial e temos também apenas 2 acessos ao disco.
A operação requer 4 acessos no caso em que o estudante ainda não existe e, a inserção é na 1 posição encontrada, e o seu país também não (1 acesso para procurar, outro para criar o país e outro para atualizar os seus valores).
Por fim temos o caso em que tentamos inserir um estudante, mas a primeira posição de hash estava ocupada. Se não conseguimos garantir que o estudante não se encontra na tabela vamos ter colisões até encontrarmos uma posição válida para inserir como descrito acima ou encontrarmos o estudante já presente na tabela.

A complexidade temporal deste algoritmo será geralmente $O(1)$, ou seja, constante. Isto porque para qualquer ação, seja a procura inicial, ou a inserção todas as operações são feitas em tempo constante.

Remover um identificador

À semelhança da operação da inserção ambas param nos mesmo casos, ou seja posição vazia, ID igual e posição fora do tamanho do ficheiro. A diferença é o que fazem nessas posições.

1 – Acedemos ao ficheiro dos estudantes e procuramos pela chave do estudante a remover.

2 – Se a posição for uma struct Student com ID vazio ou se for uma posição fora do tamanho do ficheiro então a operação é cancelada pois o estudante não se encontra na tabela.

3 – Quando encontramos um estudante com um ID igual ao ID da chave

existem 2 opções. A primeira é quando o estado do estudante é de estar presente na tabela, ou seja, encontramos o estudante a remover. O outro caso é quando este já foi removido e não se encontra presente na tabela, nesse caso a operação é cancelada.

4 – Se encontramos o estudante e este não estiver marcado por ter acabado ou abandonado o curso então a operação é bem-sucedida, caso contrário a operação é cancelada.

5 – Caso a operação foi bem-sucedida, resta aceder ao ficheiro dos países e atualizar o respetivo país.

Em termos de acessos vamos ter 2 casos no ficheiro dos estudantes. O primeiro é quando temos apenas 2 acessos ao ficheiro, caso seja porque encontramos o estudante imediatamente ou porque chegamos à conclusão de que este não está na tabela. O segundo caso é quando ocorrem colisões, nesse caso vamos ter tantos acessos como o número de colisões até encontrarmos o estudante que procuramos ou numa posição em que a operação seja cancelada. Para o ficheiro dos países se não houver colisões teremos 2 acessos ao ficheiro ou o número de colisões até encontrar o país será traduzido em acessos.

A complexidade temporal será também $O(1)$, dado que cada acesso ao ficheiro é em tempo constante e o número de colisões mesmo com 10 Milhões de estudantes é bastante reduzido.

Assinalar que um estudante terminou o curso

Calculamos o valor de hash do id do estudante, acedemos ao ficheiro para procurar o estudante em causa. O processo de procura é exatamente igual ao processo de procura na remoção.

1 – Acede-se ao ficheiro dos estudantes e procuramos pela chave do estudante que queremos marcar.
2 – Se o estudante estiver na tabela e não estiver marcado como abandonado ou acabado o curso, então vamos alterar o bool quit para TRUE. No caso de o estudante se encontrar presente na tabela, mas um dos marcadores sobre o curso (acabar e abandonar) estiver a TRUE então a operação é cancelada.
3 – Se a operação foi bem-sucedida vamos aceder ao ficheiro dos países e atualizamos os seus valores.

A operação requer 4 acessos (ler e escrever em ambos os casos), quando a operação é bem-sucedida e encontramos o estudante e o país na primeira posição que procuramos. No caso de termos colisões vamos ter tantos

acessos como o número de colisões. A diferença é que se a operação for bem-sucedida também vamos ter a possibilidade de colisões que se vão traduzir em acessos no ficheiro dos países.

A operação é *O*(1) de complexidade temporal pois cada ação acaba por ser em tempo constante.

Assinalar o abandono de um estudante

Esta operação é idêntica ao término de um estudante, tem apenas uma diferença que é o bool que alteramos.

1 – Acede-se ao ficheiro dos estudantes e procuramos pela chave do estudante que queremos marcar.
2 – Se o estudante estiver na tabela e não estiver marcado como abandonado ou acabado o curso, então vamos alterar o bool abandoned para TRUE. No caso de o estudante se encontrar presente na tabela, mas um dos marcadores sobre o curso (acabar e abandonar) estiver a TRUE então a operação é cancelada.
3 – Se a operação foi bem-sucedida vamos aceder ao ficheiro dos países e atualizamos os seus valores.

A operação requer 4 acessos (ler e escrever em ambos os casos), quando a operação é bem-sucedida e encontramos o estudante e o país na primeira posição que procuramos. No caso de termos colisões vamos ter tantos acessos como o número de colisões. A diferença é que se a operação for bem-sucedida também vamos ter a possibilidade de colisões que se vão traduzir em acessos no ficheiro dos países.

A operação é *O*(1) de complexidade temporal pois cada ação acaba por ser em tempo constante.

Obter dados de um país

Acede-se ao ficheiro dos países e fazemos uma procura semelhante à que acontece nas anteriores operações (Remoção, abandono e terminar o curso)

1– Acedemos ao dos países e procuramos pelo ID do país do qual queremos obter dados. Neste ficheiro como não existem remoções não vamos ter de nos preocupar com os casos de países que foram apagados da tabela.
2 – Se o país não existir na tabela a operação é cancelada.
3 – No caso em que encontramos o país em causa temos 2 opções, quando o total de alunos é zero ou quando é diferente de zero. O total ser zero implica que a operação foi bem-sucedida e que o país existe, mas não temos

dados sobre o mesmo. Por outro lado, se for diferente de zero então a operação está completa e processa-se o print desejado.

Em termos de acessos ao disco vão apenas depende do número de colisões, se for zero, ou seja, país encontrado na 1 posição fornecida pela função de hash temos 1 acesso. Caso contrário vamos ter tantos acessos como o número de colisões.

A complexidade temporal desta ação é tal como todas as outras $O(1)$, dependendo apenas no número de colisões, sendo este sempre um número reduzido.

Início e fim da execução

No início da execução, serão abertos os ficheiros binários para leitura e escrita, apartir da função $file\_open()_{(1)}$. Caso estes ficheiros não existam, serão criados. De seguida, são cridas as structs do tipo country e do tipo student. Por último, são declarados um char, uma string id e uma string p_id. Estes servirão para posterior leitura do input do programa.

Esta operações traduzem-se apartir desta porção do código:

```
FILE *file = file_open(STUDENTS_FILE);
FILE *cFile = file_open(COUNTRY_FILE);

Country *country = country_new("--");
Student *student = student_new("------", "--");

char c;
char id[ID_SIZE];
char p_id[P_ID_SIZE];
```

Código da main

```
FILE *file_open(char *file_name)
{
    //abre file
    FILE *f = fopen(file_name, "rb+");

    //caso file exista retorna-o
    if(f != NULL)
        return f;

    //caso file nao exista cria
    f = fopen(file_name, "wb+");

    return f;
}
```

(1)função file_open

No fim da execução do programa, libertamos a memória alocada para as structs student e country e fechamos ambos os ficheiros binários através da função $file\_close()_{(2)}$

O que se traduz pelo seguinte código:

```
free(student);
free(country);

file_close(file);
file_close(cFile);
```

Código da main

```
void file_close(FILE *file)
{
    fclose(file);
}
```

(2)função file_close

Bibliografia e Webgrafia

→Thomas H. Cormen, Charles E. Leiserson e Ronald L. Rivest, Clifford Stein , *Introduction to Algorithms* , 3ªEdição.

→Prof.Vasco Pedro, Estruturas de Dados e Algoritmos 2, plataforma moodle: https://www.moodle.uevora.pt/1920/course/view.php?id=1390 [2 de abril de 2020].

→Geeks for Geeks, Secondary Memory Acess: https://www.geeksforgeeks.org/introduction-of-secondary-memory/

→Mazeika, Prof.Dr.Dalius, Data Structures and Algorithms: http://dma.vgtu.lt/DS/DS12.pdf [1 abril de 2020].

→UNESCO, Wikipedia : https://pt.wikipedia.org/wiki/Organiza%C3%A7%C3%A3o_das_Na%C3%A7%C3%B5es_Unidas_para_a_Educa%C3%A7%C3%A3o,_a_Ci%C3%AAncia_e_a_Cultura

→Dan Bernstein, djb2 algorithm: http://www.cse.yorku.ca/~oz/hash.html