Relatório da 1ªFase do Trabalho Prático

de Estruturas de Dados e Algoritmos II

Diogo Solipa(l43071) Leonardo Catarro(l43025)



# 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 por país com os alunos ativos e inativos desse país. Vamos então ter tantos ficheiros como o número de países das Nações Unidas e o número de Hastables máximo vai ser:

- 193 (nº países) + 1 (ficheiro repartido quando nºestudantes > 5 M) + 1 (tabela com todos os estudantes) = 195 tabelas

Uma outra estrutura que iremos utilizar é um array. Este array será de structs Student e vai servir para guardarmos os estudantes na memória central.

**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 a pesquisa será linear, que vai ser quando pesquisarmos por todos os alunos de um país. Tirando esse caso é a estrutura mais eficaz e eficiente para este tipo de dados e operações.

A escolha do array para a memória central foi graças à sua eficiência em termos de uso de memória. Isto porque poupamos usar referências para outros nós como numa AVL ou numa Lista ligada.

**Dimensionamento**

Sendo que não conseguimos saber de antemão quantos estudantes vai haver, o dimensionamento vai aumentando de acordo com a necessidade de a estrutura crescer para suportar mais estudantes, com a dimensão máxima de aproximadamente 20 Milhões de posições. Isto porque suportará no máximo 10 Milhões de estudantes e com um fator de reHash de 0.5 que se exceder esse fator (nº estudantes / capacidade) uma tabela com pelo menos o dobro da dimensão da anterior é criada e usamos sempre o primo mais próximo desse valor para uma melhor resolução de conflitos.

O array irá ter uma dimensão de 5 Milhões que é o número máximo de estudantes que pode haver em memória central. A sua ocupação será variável dependendo do número de estudantes que estarão ao mesmo tempo na estrutura. O número de estudantes que estão na estrutura a qualquer altura, é um contador que está guardado em RAM.

**Ligação entre as estruturas**

A única ligação existente entre estruturas é na operação de verificar todos os dados de um país onde os dados existentes no respetivo ficheiro serão transferidos para o array que está memória central (nunca excedendo o limite de memória).

**Localização**

Em memória central ou primária, irá ficar um array para suportar o uso de múltiplos estudantes no programa. Em memória secundária irão ficar todos os ficheiros com as respetivas Hashtables e os seus estudantes.

**Descrição dos Elementos das estruturas**

A Hashtable será composta por estruturas do tipo Student e, portanto, cada índex da Hashtable que esteja alocada terá um estudante associado em que a chave é a string id do estudante.

O array será composto por structs Student.

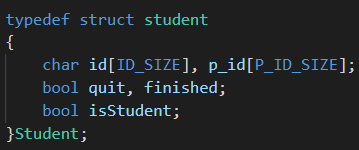
**Memória ocupada pelas estruturas em memória central**

O array em memória central irá variar na quantidade de memória que ocupará. Desde apenas 1 elemento quando queremos verificar algo para apenas 1 estudante ou até 5 Milhões de estudantes, ocupando assim aproximadamente 60 MB (5,000,000 X 12 bytes). Quando acabar uma verificação toda a memória das structs será libertada.

# 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. A struct student contém os seguintes elementos:



🡪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.

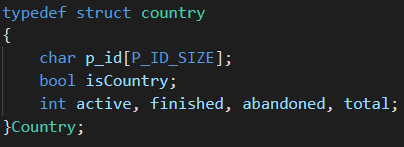
🡪bool isStudent: booleano para verificar se o estudante já existe ou não. Este booleano ocupa **1byte**

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 contem a informação dos países. Esta informação será guardada em Hash Tables do tipo country. A struct country tem os seguintes elemntos:



🡪char p\_id[P\_ID\_SIZE]: contém o id do país. Esta string ocupa **3bytes**.

🡪int active, finished, abandoned, total: inteiros que guardam o numero 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**

# Operações

**Introduzir um novo estudante**

O índex onde vamos inserir o estudante na Hashtable será calculado por uma função de hashing para Strings (provavelmente usaremos a função djb2 sendo que em primeira análise nos pareceu a melhor para strings).

1 – Acedemos ao ficheiro, do país do estudante que queremos introduzir, se este ainda não existir é criado. No caso do ficheiro já existir fazemos uma rápida pesquisa para sabermos se já existe um estudante com a chave do qual vamos inserir. Se o estudante já existir a operação é cancelada.

2 – É calculado o índex de acordo com a chave do estudante (id) por uma função de hash. Acedemos ao ficheiro e no caso de já haver um estudante nessa posição, dá-se uma colisão. Se não houver colisão o estudante é inserido naquela posição.

3 – Inserimos também o estudante na Hashtable global que contém todos os alunos de todos os países.

4 – No caso de haver colisão temos de usar uma função auxiliar para re-calcular o índex e a nossa função terá uma resolução de colisões quadrática.

5 – O fator de carga é calculado por (nº estudantes / capacidade da tabela) e se este valor ultrapassar 0.5 após uma inserção, ou seja, mais de metade da tabela estiver em uso, será feito reHash. Esta operação irá re-calcular todos os indexes para as diferentes chaves e criará uma nova tabela em que a dimensão será o maior primo mais próximo do dobro da capacidade da anterior tabela.

A operação requer 3 acessos no caso em que a pesquisa apenas acede 1 vez ao ficheiro, ou seja, o índex obtido não continha nada e apenas voltamos a aceder para inserir o estudante. No caso de o estudante já estar na tabela vamos ter n acessos onde n é o número de posições que procuramos até encontrar o estudante. Por fim temos o caso em que tentamos inserir um estudante, mas existem colisões, logo c1 + c2 + 2, pois teremos c1 colisões no ficheiro global e c2 colisões no ficheiro do respetivo país, mais 2 acessos que representam a escrita nos ficheiros (tanto c1 como c2 podem ser 0).

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. O pior caso por outro lado é *O*(n), isto porque no caso de haver várias diferentes chaves com valores de hash igual procurar por essa chave pode ser linear. Com uma boa função de hash e um fator de carga baixo (0.5) não devemos ter esse problema.

**Remover um identificador**

Calcula-se o valor de hash do respetivo identificador e iremos aceder a essa posição da tabela global para remover o estudante.

1 – Calculamos o valor de hash do respetivo identificador. Com essa posição acede-se ao ficheiro global e carregamos para memória central a struct header que terá o país respetivo a esse identificador. Por fim apagamos toda a informação referente a esse identificador do ficheiro global. No caso de o identificador não existir a operação é cancelada.

2 – Acedemos ao ficheiro do respetivo país e procuramos pelo identificador, assim que o encontrarmos removemos toda a sua informação.

Em termos de acessos vamos ter 2 casos. O primeiro é quando não existe o identificador que estamos a tentar remover, portanto, vamos ter c acessos em que c é o número de colisões (pode ser 0). O outro caso é quando encontramos o identificador, carregamos a struct header para a RAM, apagamos toda a informação do ficheiro global e da acedemos ao ficheiro do respetivo país e removemos toda a informação desse identificador. Os acessos poderão ser calculados por c1 + c2 + 2, colisões no ficheiro global (c1) mais as colisões do ficheiro do país e por fim 1 acesso por ficheiro para remover a informação.

A complexidade temporal será em média também *O*(1), mas tal como em todas as operações o pior caso é *O*(n), mas que com uma boa função de hash reduzimos muito a probabilidade de isto ocorrer.

**Assinalar que um estudante terminou o curso**

Calculamos o valor de hash do id do estudante, acedemos ao ficheiro global para obtermos o país do estudante e procedemos a aceder a esse ficheiro para assinalar o término.

1 – Acede-se ao ficheiro global para obter o país do id que é dado no input, se este id não estiver na tabela global, o id é considerado inválido e a operação é parada.

2 – A struct header é carregada para memória central e acedemos ao país que estiver identificado no p\_id.

3 – Usamos o p\_id passado para descobrir o ficheiro no qual queremos procurar o estudante e o id como chave para encontrar o estudante em questão.

4 – Acedemos ao ficheiro do país do estudante e procuramos pelo id do mesmo e alteramos o valor do bool finished de FALSE para TRUE.

5 – Guarda-se as alterações ao ficheiro, fechando-o. Toda a memória alocada em RAM é libertada.

A operação requer c acessos, onde c é o número de colisões até chegar à conclusão de que o id não estava no sistema e cancelar a operação. No caso em que o id existe, teremos c1 + c2 + 2, pois teremos c1 colisões no ficheiro global e c2 colisões no ficheiro do respetivo país, mais 2 acessos que representam o carregar do primeiro ficheiro e o escrever do segundo (tanto c1 como c2 podem ser 0).

A operação é *O*(1) de complexidade temporal pois cada ação acaba por ser em tempo constante. O pior caso é mais uma vez *O*(n).

**Assinalar o abandono de um estudante**

Calculamos o valor de hash do id do estudante, acedemos ao ficheiro global para obtermos o país do estudante e procedemos a aceder a esse ficheiro para assinalar o término.

1 – Acede-se ao ficheiro global para obter o país do id que é dado no input, se este id não estiver na tabela global, o id é considerado inválido e a operação é parada.

2 – A struct header é carregada para memória central e acedemos ao país que estiver identificado no p\_id.

3 – Usamos o p\_id passado para descobrir o ficheiro no qual queremos procurar o estudante e o id como chave para encontrar o estudante em questão.

4 – Acedemos ao ficheiro do país do estudante e procuramos pelo id do mesmo e alteramos o valor do bool quit de FALSE para TRUE.

5 – Guarda-se as alterações ao ficheiro, fechando-o. Toda a memória alocada em RAM é libertada.

A operação requer c acessos, onde c é o número de colisões até chegar à conclusão de que o id não estava no sistema e cancelar a operação. No caso em que o id existe, teremos c1 + c2 + 2, pois teremos c1 colisões no ficheiro global e c2 colisões no ficheiro do respetivo país, mais 2 acessos que representam o carregar do primeiro ficheiro e o escrever do segundo (tanto c1 como c2 podem ser 0).

A operação é *O*(1) de complexidade temporal pois cada ação acaba por ser em tempo constante. O pior caso é mais uma vez *O*(n).

**Obter dados de um país**

Acede-se ao ficheiro referente ao país do qual queremos obter os dados, verificamos o tamanho do ficheiro e carregamos os dados que forem possíveis carregar para RAM.

1– Acedemos ao ficheiro referente ao país do qual queremos obter dados, se este não existir a operação é cancelada e significada que não existem estudantes no sistema desse país.

2 – Se o ficheiro existir, vamos ver o tamanho do ficheiro e se for menor que 60 MB podemos carregá-lo diretamente para RAM, struct após struct, se for maior vamos ter de repartir o ficheiro em metade e carregamos uma metade para processar, libertamos essas structs da memória central e carregamos a outra metade.

5 – No fim, fecharemos o ficheiro selecionado e damos free a toda a memoria alocada no fim do processo.

Com 10 milhões de estudantes no sistema nunca vamos ter de partir o ficheiro em mais que 2 vezes.

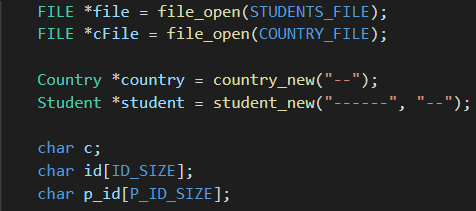
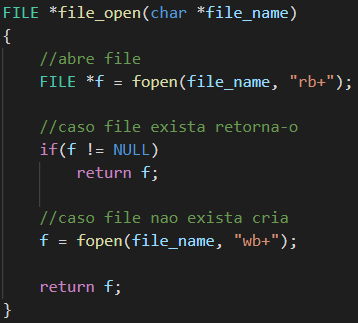
Em termos de acessos ao disco vão ser tantos como o número de structs presentes nesse ficheiro e no caso de o ficheiro não existir teremos 0 acessos.

A complexidade temporal desta ação é *O*(n), em que n é o número de estudantes no ficheiro. Esta é a operação que mais recursos vai usar e mais lenta sendo que é uma procura linear, que percorre a tabela inteira.

# Início e fim da execução

No inicio da execução, serão abertos os ficheiros binários para leitura e escrita, apartir da função . Caso estes ficheiros não existam, serão criados. De seguida, são cridas as structs do tipo country e do tipo student. Por ultimo, são declarados um char, uma string id e uma string p\_id. Estes servirão para posterior leitura do input do programa.

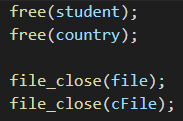
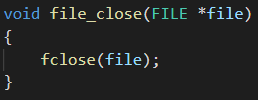
Esta operações traduzem-se apartir desta porção do código:

Código da main (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

O que se traduz pelo seguinte código:

Código da main (2)função file\_close

# Bibliografia e Webgrafia

🡪[Thomas H. Cormen](http://www.cs.dartmouth.edu/~thc/), [Charles E. Leiserson](http://supertech.csail.mit.edu/~cel/) e [Ronald L. Rivest](http://theory.csail.mit.edu/~rivest/), [Clifford Stein](http://www.columbia.edu/~cs2035/) , *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>