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.

**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**

A Hashtable temporária em memória central irá variar em dimensão e consequentemente na quantidade de memória que ocupará. Esta nunca irá suportar ao mesmo tempo mais de cerca de 5 Milhões de estudantes, ocupando assim aproximadamente 60 MB. Quando a estrutura não estiver em uso estará vazia e toda a memória sem ser a da própria Hashtable será libertada.

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

Todos os ficheiros possuíram a mesma estrutura, apenas serão distinguidos pelo que país que representam.

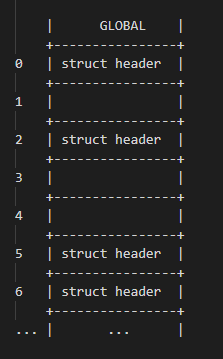
Cada ficheiro conterá uma HashTable com a informação desses alunos, em que cada posição da tabela representa um aluno. Daí que, poderemos ter um máximo de 195 ficheiros, isto é 193, um para cada país, 1 suplementar para o caso de um dos países possuir mais de 5 000 000 estudantes e 1 que contenham todos os estudantes.

O valor 5 000 000 é o total de alunos que conseguimos ter na memória central. Valor dado por:

64 000 000 / tamanho da Struct(12bytes) = 5 333333.

Por questão de organização e visto que o total de alunos possível é de 10 000 000, optamos por utilizar os 5 000 000 de forma a deixar alguma memória livre, para caso de necessidade.

A estrutura do ficheiro global (que contem todos os estudantes) é a seguinte:

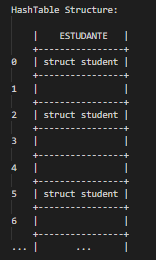


Cada posição da Tabela conterá uma struct do tipo header representando um aluno de forma a servir de ponte para a inserção ou remoção nos ficheiros de cada país. Struct essa que conterá os seguintes elementos:

🡪O **id** do aluno (char[7] id): **7 bytes**

🡪O **p\_id** do aluno (char[3] p\_id): **3 bytes**

A estrutura dos restantes ficheiros será a seguinte:



# 

# 

Cada posição da Tabela conterá uma struct que representará um aluno em específico. Struct essa que conterá os seguintes elementos:

🡪O **id** do aluno (char[7] id): **7 bytes**

🡪O **p\_id** do aluno (char[3] p\_id): **3 bytes**

🡪Um boleano para verificar se o aluno terminou o curso (bool **finished**): **1 byte**

🡪Um boleano para verificar se o aluno desistiu do curso (bool **quit**): **1 byte**

Tamanho da struct = (7 x 1 byte) + (3 x 1 byte) + (1 byte) + (1 byte) = **12 bytes**

O tamanho do ficheiro é variável, de acordo com o número de alunos existente, mas no pior caso, o ficheiro poderá ter um tamanho máximo de:

Tamanho máximo do ficheiro:

Tamanho do ficheiro = 10 000 000 x Tamanho da Struct =

10 000 000 x 12 bytes = 120 000 000 bytes = **120 MB**

# 

# 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, para fazer 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 com 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 (c1) e c2 colisões no ficheiro do respetivo país (c2) 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**

**Assinalar o abandono de um estudante**

**Obter dados de um país**

# Início e fim da execução

Em início de execução criaremos uma HashTable e uma Struct do tipo student, ambas em memória central. Caso esta tabela já exista, estará vazia.

No fim da execução, fechamos os ficheiros existentes, dado que as atualizações aos mesmos serão escritas em disco e libertamos toda a memória alocada ao longo de todo o programa.

# 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].