Universidade De Évora

Departamento de Informática

Estruturas de Dados e Algoritmos II

Ano Letivo 2019-2020

Estudantes no mundo

Alunos:

Marcelo Feliz nº38073

Denis Lapuste nº42616

Docente:

Vasco Pedro

9 de junho, 2020

Índice

1. Introdução

No âmbito da unidade curricular de Estruturas de Dados e Algoritmos II pretende-se criar um programa que seja capaz de guardar a informação sobre estudantes (um identificador que, como o nome indica, vai identificar o estudante, logo vai ser único para cada um e o estado da sua matrícula) pelo mundo todo, quer seja de um país existente, quer seja de um país que possa vir a existir (desde que o código do país não seja já existente, onde o código corresponde a uma combinação de dois letras maiúsculas).

Para tal é necessário o programa ler comandos do seu *standard input* e executar as operações correspondentes, onde cada linha irá corresponder a um só comando a executar. Como a quantidade de alunos pode ir até aos 10000000 (dez milhões) de alunos guardados, a forma mais eficaz de os conseguir guardar e lhes aceder livremente, encontrámos ser através de uma *Hash Function*, e como a informação deve manter-se em caso de reinício do programa decidimos guardar toda a informação num ficheiro binário onde a posição correspondente era baseada na *Hash Function* mencionada anteriormente.

Como entre os comandos de *input* existe a opção de "Obter os dados de um país" ter a informação de cada país pronta caso necessário faz parte das funcionalidades e ler todo o ficheiro somando todas as ocorrências desse país não é eficiente, decidimos criar mais um ficheiro com uma matriz, em que guardamos a informação de cada país tais como: número de estudantes correntes, estudantes diplomados e estudantes que abandonaram (decidimos fazer a soma destes três sempre que for pedido a informação do país para dar o Total de estudantes, porque achamos ser mais eficaz em questão de memória ocupada no ficheiro binário e quando estivéssemos a ler do ficheiro em vez de quatro *Integers* onde cada um ocupa quatro bytes, ler só três, o mesmo acontece quando for escrito de volta). Este último ficheiro ao contrário do que guarda os alunos é totalmente carregado em memória no início do programa e guardado em disco sempre que terminamos a sua execução.

2. Desenvolvimento

2.1. Estruturas

2.1.1. Hashtable

Como já foi referido anteriormente, para se alcançar o objetivo do trabalho foi usada uma função *Hash*, de forma a guardar a informação no ficheiro com a complexidade mais baixa possível, e por isso iremos referir-nos ao ficheiro como uma *Hashtable*.

Decidimos guardar um estudante por posição na *Hashtable*. A informação a guardar de cada estudante é:

-Uma letra guardada em forma de *Char* para determinar o estado de um estudante podendo tomar um dos seguintes valores ("I" para indicar um estudante ativo ou corrente, "T" para indicar um estudante que terminou o curso, ou seja, diplomado, "A" que indica que o estudante abandonou e finalmente "R" que indica que a informação sobre o aluno não estará mais disponível na dita "Base de dados");

-Uma combinação de duas letras (*String*) guardada num *Char*[3], que corresponde ao país em que o estudante está a estudar (este conjunto de duas letras vai de "AA" a "ZZ" com todas as combinações possíveis);

-Uma combinação de 6 letras e/ou números (*String*) guardada num *Char*[7], que por sua vez corresponde ao identificador (ID) do estudante. Este identificador é único para cada estudante sendo só possível ser reutilizado caso a informação do mesmo seja "removida" da base de dados, ou seja, o estado for "R" caso contrário não será possível uma inserção de um estudante cujo identificador já esteja em uso.

Sempre que for para inserir ou aceder ao ficheiro é preciso multiplicar por *sizeof(Student)* já que a função de *hash* usada não o faz. Para evitar sobreposições no ficheiro.

Assim sendo podemos assumir que a posição correspondente ao aluno cujo identificador é "AAAAAA", no ficheiro binário está ocupada, e a informação lá escrita está da seguinte forma: "IPT AAAAA " (considerando que o país do estudante é "PT"). Para ter a certeza de que a informação guardada no ficheiro binário estava a ser guardada de forma correta de acordo com a ideia do grupo foi usado o comando "hexdump -C yourfile.bin" no terminal do *Linux* onde o output estava de acordo com o pretendido e a única diferença era que as posições vazias eram representadas por pontos.

Com tudo isto podemos concluir que cada estudante irá ocupar 11 *bytes* no ficheiro binário.

Existe uma grande diferença entre esta dita *Hashtable* e uma *Hashtable* normal que se trata do tamanho total. O nosso ficheiro tem um tamanho máximo predefinido, que é de 498.8 Mb, este valor só é alcançado quando é introduzido um aluno na última posição (que só acontece para o identificador "ZZZZZZ" ou quando, pela correção de colisões linear, forem enchendo posições anteriores e consequentemente a ocupação desta *mais informações sobre como é calculada essa posição estará disponível mais abaixo). Porém numa *Hashtable* existe a função de *Re-Hash*, que aumenta o seu tamanho (é duplicado) caso esta alcance uma percentagem de ocupação maior que um certo limite. Esta característica não foi adaptada ao nosso programa pois o espaço disponível em disco era limitado.

Quanto à correção de colisões, inicialmente para lidar com colisões na *Hashtable*, a primeira ideia foi *double hashing,* mas como iria ser necessário a criação de uma nova função e não iria resolver totalmente o problema, decidimos fazer uma correção de colisões de forma linear (entre linear e quadrática foi somente uma questão de preferência), ou seja, caso a posição correspondente ao identificador do aluno a ser inserido já estiver ocupada a posição em que será inserido irá ser a seguinte (para calcular a posição seguinte acrescenta-se 11 *bytes* ou seja *sizeof(Student)*), e caso esta também esteja ocupada vai tentar inserir na seguinte, e assim sucessivamente. Se acontecer que a posição atual (onde estiver para ser inserido) for a última e estiver ocupada a seguinte posição a ser vista irá ser a primeira posição do ficheiro.

No exemplo a seguir é possível observar como o programa vai ler o input a seguir e agir de acordo com o pretendido:

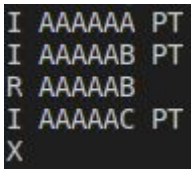

Fig.1 - Input

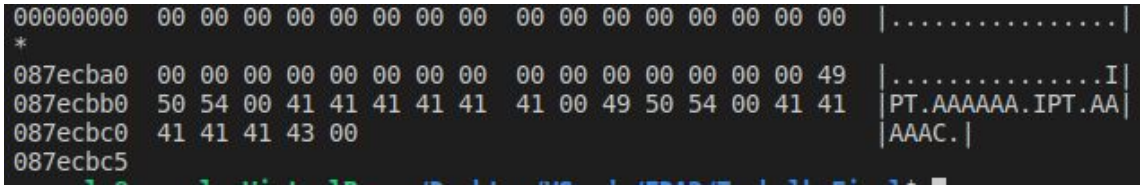

Fig.2 - Conteúdo do ficheiro

2.1.2. Matriz

Como já foi mencionado anteriormente, foi usada uma matriz (de *Country's* que se trata de um *struct* criado para guardar três valores inteiros) para guardar toda a informação dos países, onde cada posição da matriz correspondia a um país. Caso seja a primeira vez a correr o programa, ou seja não existe estudantes na "base de dados" nem nenhuma informação sobre os mesmos, esta matriz é criada na memória central logo no início do programa, com todos os seus valores iguais a zero, caso contrário, o que acontece é que o ficheiro (que lhe demos o nome de "country_Matrix.bin", este ficheiro tem um tamanho constante, de 16.2KB) é lido todo ordenadamente para uma matriz que é criada na memória central, mas desta vez vazia. No fim do programa, quer seja a primeira execução do programa quer não, a matriz em memória central é totalmente guardada no ficheiro, a maneira que é guardada no ficheiro é linear. Primeiro é guardado o *Country matrix*[0][0] que corresponde ao país cujo código é "AA", logo de seguida o país "AB", com todas as combinações de duas letras maiúsculas possíveis até "ZZ", fazendo assim um total de 676 *Country's* guardados sempre que o programa acaba. A complexidade da utilização deste ficheiro é constante, ou seja, ter muita informação ou pouca o tempo será sempre o mesmo. Foi mencionado que cada *struct Country* contém três *Integers*, a seguir encontra-se a descrição de cada valor de inteiro do *struct*:

-"active_students" guarda a quantidade de estudantes correntes;

-"end_students" guarda a quantidade de estudantes diplomados;

-"quit_students" guarda a quantidade de estudantes que abandonaram;

```
I AAAAAA PT
I AAAAAB PT
R AAAAAB
I AAAAAC PT
X
```

Fig.3 - Input

```
00000000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00002650  00 00 00 00 00 00 00 00  02 00 00 00 00 00 00 00  |................|
00002660  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00003f60
```

Fig.4 - Conteúdo do ficheiro

2.2. Função Hash

A função de *hash* usada calcula a posição do estudante a partir do identificador, que corresponde a uma combinação de letras e/ou números com o código de *ASCII* de cada caracter. Foi a função que nos pareceu mais viável quando se trata de obter um valor inteiro a partir de uma *String*.

Como o código *ASCII* dos números (entre 0 e 9) varia entre 48 e 57, não nos pareceu muito útil manter estes valores, já que do 58 ao 64 existem símbolos que não seriam usados em qualquer circunstância pois os identificadores só contêm números e letras, então foi decidido aumentar o código *ASCII* dos números por sete unidades. Assim o código de "0" a "Z" corresponde a 55 e 90 respetivamente, onde cada valor entre estes dois designa ou uma letra ou um número entre zero e nove.

Para calcular a posição do estudante, por cada caracter do identificador vai se multiplicar o valor de *ASCII* correspondente (de acordo com o que foi definido anteriormente) por 36^n (onde n é o número de caracteres que existem à direita do caracter em questão) e no final faz-se a soma desses valores todos. Após o cálculo feito irá ser-lhe subtraído o "BASICHASHFICH" que corresponde ao *hashValue* (inteiro calculado a partir desta fórmula) do "000000", assim o identificador "000000" vai corresponder à primeira posição do ficheiro, já que a partir desta fórmula não é possível obter um valor mais baixo que este. Após a subtração feita multiplicou-se o *hashValue* de "ZZZZZZ" por 11 (número de bytes dos *structs* a se guardar no ficheiro) para obter o tamanho máximo que o ficheiro alcançava, o valor obtido era de 23944605168 (como se tratam de *int* ou *long* toda a parte não inteira dos cálculos é descartada), ou seja o espaço do ficheiro seria maior que 22000 Mb o que é extremamente superior ao limite dado. Esta seria uma das únicas maneiras de evitar colisões, já que a função é crescente e diferente para qualquer que seja a combinação de letras e números possível para o identificador, mas devido ao limite de memória usável houve uma necessidade de diminuir o tamanho do ficheiro consideravelmente então por consecutivas tentativas de divisão por valores inteiros chegou-se à conclusão que dividir por 48 seria a melhor opção, ficando assim o *hashValue* de "ZZZZZZ" igual a 498845941, o que é aproximadamente 498.8 Mb.

Desta forma, ao dividir de novo pelo número de bytes, foi possível obter o *hashValue* do "ZZZZZZ" que corresponde a 4534963, e foi nos possível concluir que desta forma temos +45 milhões de posições na *hashTable*.

Exemplo para o identificador "045AB4" :

((55*36^5 + 59*36^4 + 60*36^3 + 65*36^2 + 66*36^1 + 59*36^0) - 3420657955)/48

que depois de multiplicado por *sizeof(Student)* vai dar a posição exata no ficheiro.

2.3. Complexidade

Começamos por ler ou iniciar a matriz dos países, vai ter uma complexidade constante porque o seu tempo de execução é sempre o mesmo.

O(1)

Inputs(exceto P), na melhor das hipóteses vai ser um acesso direto sem colisão

O(1)

Na pior das hipóteses poderá ter acesso com colisões, e nesse caso a complexidade vai piorar (linear)

O(n)

Para inputs com "P", acesso a matriz todos os acessos vão ser diretos, logo temos

O(1)

No fim da execução é guarda a matriz no ficheiro, isto é uma complexidade constante também, porque são sempre percorridas as 676 posições da matriz

O(1)

Portanto podemos concluir que:

Caso a melhor hipótese ocorra a complexidade da execução vai ser O(1)

Caso a pior hipótese ocorra a complexidade da execução vai ser O(n)

Realisticamente podemos afirmar que a complexidade do programa será entre O(1) e O(n) dependendo dos ID's dados no input e consequentes colisões.

2.3.1. Descrição das 5 Operações

Obs: qualquer ID guardado no ficheiro com estado "R" significa que foi removido e portanto vai ser ignorado pelas operações exceto se for uma posição útil para a inserção de um novo estudante.

2.3.1.1. Introduzir um novo estudante

Quando é lido pelo input que a operação é do tipo "I" (inserção):

1. É calculado o Hash do ID; O(1)
2. Acedemos ao ficheiro e verificamos se este ID já se encontra lá, neste caso e dependendo de colisões a complexidade pode ser O(1)caso não haja colisões ou na pior das hipóteses O(n), mas podemos afirmar que vamos sempre ter pelo menos um acesso neste ponto.
    a) Caso o ID já exista terminamos a operação e enviamos para o output "+ estudante <ID> existe". O(1)
    b) Caso o ID não exista, é então introduzido no ficheiro, e modificada a informação na matriz(Incrementado um estudante ativo no país correspondente), depois terminada a operação. O(1)

2.3.1.2. Remover um identificador

Quando é lido pelo input que a operação é do tipo "R" (remoção):

1. É calculado o Hash do ID; O(1)
2. Acedemos ao ficheiro e verificamos se este ID existe, neste caso e dependendo de colisões a complexidade pode ser O(1)caso não haja colisões ou na pior das hipóteses O(n), mas podemos afirmar que vamos sempre ter pelo menos um acesso neste ponto.
    a) Caso o ID tenha sido encontrado e o seu estado for "I" alteramos o seu estado para "R" e modificamos a informação dos países de acordo(decrementar um estudante ativo no país correspondente). O(1)
    b) Se o ID não for encontrado, enviamos para o output "+ estudante <ID> inexistente".O(1)
    c) Se o estado do ID for "T", enviamos para o output "+ estudante <ID> terminou".O(1)
    d) Se o estado do ID for "A", enviamos para o output "+ estudante <ID> abandonou".O(1)

2.3.1.3. Assinalar que um estudante terminou o curso

Quando é lido pelo input que a operação é do tipo "T" (terminou):

1. É calculado o Hash do ID; O(1)
2. Acedemos ao ficheiro e verificamos se este ID existe, neste caso e dependendo de colisões a complexidade pode ser O(1)caso não haja colisões ou na pior das hipóteses O(n), mas podemos afirmar que vamos sempre ter pelo menos um acesso neste ponto.
    a) Caso o ID tenha sido encontrado e o seu estado for "I" alteramos o seu estado para "T" e modificamos a informação dos países de acordo(decrementar um estudante ativo no país correspondente e incrementamos um estudante nos que terminaram do mesmo país). O(1)
    b) Se o ID não for encontrado, enviamos para o output "+ estudante <ID> inexistente".O(1)
    c) Se o estado do ID for "T", enviamos para o output "+ estudante <ID> terminou".O(1)
    d) Se o estado do ID for "A", enviamos para o output "+ estudante <ID> abandonou".O(1)

2.3.1.4. Assinalar o abandono de um estudante

Quando é lido pelo input que a operação é do tipo "A" (Abandonou):

1. É calculado o Hash do ID; O(1)
2. Acedemos ao ficheiro e verificamos se este ID existe, neste caso e dependendo de colisões a complexidade pode ser O(1)caso não haja colisões ou na pior das hipóteses O(n), mas podemos afirmar que vamos sempre ter pelo menos um acesso neste ponto.
    a) Caso o ID tenha sido encontrado e o seu estado for "I" alteramos o seu estado para "A" e modificamos a informação dos países de acordo(decrementar um estudante ativo no país correspondente e incrementamos um estudante nos que abandonaram do mesmo país). O(1)
    b) Se o ID não for encontrado, enviamos para o output "+ estudante <ID> inexistente".O(1)
    c) Se o estado do ID for "T", enviamos para o output "+ estudante <ID> terminou".O(1)
    d) Se o estado do ID for "A", enviamos para o output "+ estudante <ID> abandonou".O(1)

2.3.1.5. Obter dados de um país

Quando é lido pelo input que a operação é do tipo “P”:

1. Calculamos a posição da matriz que contém a informação desse país com as 2 letras fornecidas(EX. ‘P AB’ conseguimos saber que a informação deste país estará guardada na matrix[0][1] porque “A” - “A” = 0 e “B” - “A” = 1); O(1)
2. Depois de sabermos onde está informação pedida temos de a analisar:
    a) Se não existirem estudantes correspondentes ao país então é enviado para o output “+ sem dados sobre <País>”; O(1)
    b) Caso exista informação então somamos a quantidade de alunos que terminaram, abandonaram e existem para criar o total de alunos do país(soma), podendo assim enviar para o output “+ <País> - concorrentes: <existem>, diplomados: <terminaram>, abandonaram: <abandonaram>, total: <soma>”; O(1)

2.3.1.6. Terminar a execução do programa

Quando é lido pelo input que a operação é do tipo “X”:

Quando é lido a operação ”X” o programa não vai aceitar mais inputs, se existirem não serão lidos.

Antes do término do programa a matriz guardada em memória central é guardada em ficheiro. Depois de ter sido guardada então o programa termina a sua execução; O(1)

3. Conclusão

Em conclusão achamos que conseguimos com sucesso cumprir todos os objectivos deste trabalho, assim a nossa submissão final no *Mooshak* que deverá ser avaliada é referente ao problema "T" (Total).

A realização deste trabalho foi nos útil para aprendermos mais sobre como funcionam ficheiros binários e como manipular a informação neles de forma eficiente a partir da linguagem de programação "C".

No início tentámos utilizar o máximo de memória central para tentar aproveitar as suas vantagens, já que é relativamente mais rápido. Mas depois de a implementação feita chegámos à conclusão de que só iria ser mais eficiente para quando o número de alunos fosse relativamente pequeno, porque ao serem inseridos vários alunos com a memória central cheia ia ser necessário fazer as verificações sempre e acabava por perder demasiado tempo. Para o caso de ser preciso alterar informação sobre um aluno seria necessário verificar se este estivesse em memória central e caso não estivesse seria necessário copiar do ficheiro, fazendo assim o programa mais lento ainda para um número elevado de comandos no *input*. Finalmente o problema que consideramos no final foi o tempo que este programa ia demorar a guardar todos os alunos da memória central para o ficheiro no final da execução.

Outra alteração feita foi no ficheiro principal (segundo ficheiro, o ficheiro que guarda todos os alunos com a respetiva informação) foi que decidiu-se não usar um *array* de *LinkedLists* e sim uma "*Hashtable*", esta alteração foi feita porque chegou-se à conclusão de que seria mais eficiente em questão de procura (um factor muito importante para este trabalho). Também seria difícil controlar o ficheiro e a sua informação já que dependendo do tamanho da lista seria necessário um espaço diferente e isso poderia vir a influenciar a posição da próxima, sendo assim extremamente mais complicada a implementação do programa.

Relativamente ao primeiro ficheiro onde ia ser guardada a informação dos países, inicialmente tínhamos em mente guardar também um *array* de inteiros para controlar o tamanho de cada *LinkedList* no segundo ficheiro, mas como esta parte foi descartada como mencionado anteriormente decidiu-se descartar também este *array*.

O problema que nos foi mais difícil de entender foi um erro causado pela falta de verificação se uma posição existia no ficheiro antes de a tentar ler. Quando o programa era corrido nos nossos computadores, este funcionava corretamente (e sem grande demora) para qualquer que fossem os ficheiros *input* usados, mas no *Mooshak* o problema fazia com que o programa estivesse em espera durante um longo período de tempo que o conteúdo da *String* se tornasse em uma *String* vazia, assim excedendo o tempo limite sem guardar nada nos ficheiros. Após entendida e corrigida a falha, o trabalho funcionou na plataforma de submissões.

Numa parte mais final do trabalho e a razão pela qual o nosso trabalho não estava a passar na fase "T" foi uma pequena falha numa optimização que acabou por fazer com que a matriz não fosse corretamente guardada no ficheiro, a razão é simples, depois de uma execução em que já tivéssemos dados no ficheiro so estavamos a carregar em memória central os países à medida que precisávamos deles, no entanto no fim guardávamos todos os valores da nossa matriz em ficheiro, substituindo alguns valores importantes mas que não tinham sido utilizados naquela execução por zero (pois no início do programa uma matriz com todos os valores a zero era criada em memória central), o que nos causou algumas perdas de informação. Ao detectar este problema decidiu-se voltar à ideia inicial que era carregar toda a matriz para memória central no início do programa, assim, o programa passou então na fase "T".

4. Webgrafia:

25/06/2020

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

https://dev.to/sharkdp/what-is-a-binary-file-2cf5

https://stackoverflow.com/questions/1765311/how-to-view-files-in-binary-from-bash

30/06/2020

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

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

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

5. Bibliografia:

Foram usados materiais que tínhamos de outras cadeiras, o mais importante referir é matéria sobre Hashtables de EDA1.

Obs: Foram utilizados muitos outros sites/recursos, no entanto ou não foram relevantes ou acabámos por ir num caminho diferente em que não foram úteis.