Relatório da 1ª Fase do Trabalho Prático de Estruturas de Dados e Algoritmos II

Miguel Luís 37555 Nuno Andrade 42130

9 de Maio de 2020

1 Resumo

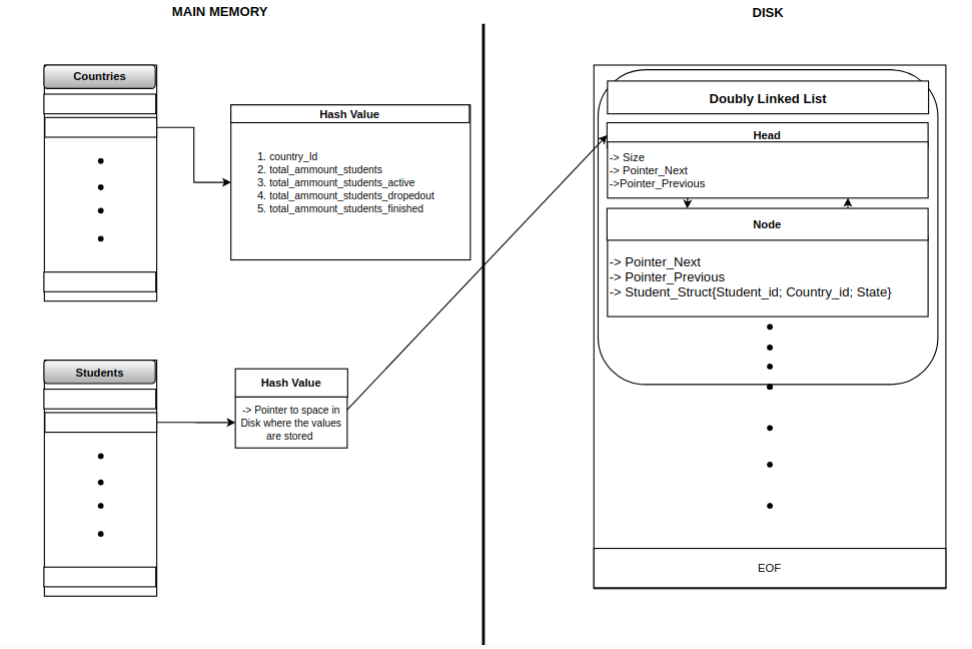

Figura 1: Data Structure Design

Neste trabalho requisitado pelo docente Vasco Pedro responsável pela cadeira de EDA2, encontra-se a descrição da implementação a realizar para uma aplicação, que permite registar estatísticas dos 193 estados membros da UNESCO. Na implementação pretendemos utilizar uma combinação de 2 Hash Tables em memória primária e Doubly Linked Lists em memória secundária.

2 Estruturas de dados

2.1 Estruturas de dados a utilizar e a sua descrição

Para este trabalho decidimos utilizar as seguintes estruturas de dados:

1- *Tabelas de dispersão*: Array associativo que mapeia chaves a valores, usando uma função de dispersão.

2- *Doubly Linked List duplamente ligadas*: Estrutura linear de tamanho variável em que os seus elementos não estão guardados de forma contígua, esta estrutura pode ser percorrida do início para o fim, ou do fim para o início ao contrário de uma Linked List.

Este programa irá ter duas folhas de dispersão(Hash Tables), uma para guardar os dados relativos a cada país, e outra para guardar a posição onde uma Doubly Linked List duplamente ligada(Doubly Linked List) que está ordenada de forma crescente, se encontra num determinado ficheiro. Doubly Linked List esta que contém dados de estudantes individuais em cada um dos seus nós.
Iremos daqui em diante referir-nos a estas estruturas de dados pelas suas nomenclaturas em Inglês

2.2 Razões da escolha

Escolhemos utilizar estas estruturas tanto pela sua facilidade de uso e implementação, como a velocidade de execução das operações que foram requisitadas.

Hash Table- Uma Hash Table permite-nos realizar operações de inserção, remoção e busca em tempo constante ou muito perto disso. A função de hash utilizada para obter as posições na hash table é a FNV-1(função escolhida após uma pesquisa sobre funções de hashing).

Hash_Table	Average	Worst Case
Search	O(1)	O(n)
Insert	O(1)	O(n)
Delete	O(1)	O(n)

Sendo **n** o tamanho da Hash Table.

Doubly Linked List- A inserção e numa Doubly Linked List pode ser consideravelmente mais rápido do que em uma Doubly Linked List simplesmente ligada. Apesar de esta estrutura ocupar mais espaço, não ultrapassa o limite máximo de memória indicado pelo docente.

Doubly_Linked_ List	Average	Worst Case
Search	O(n)	O(n)
Insert	O(n)	O(n)
Delete	O(n)	O(n)

Devido à Doubly Linked List estar ordenada e duplamente ligada, tanto o pior como o pior caso tem complexidades temporais de **n**, sendo que neste caso, **n** será metade do tamanho da Doubly Linked List.

2.3 Dimensões

Para se poder especificar a memória máxima que poderá vir a ser utilizada neste trabalho, necessitamos de apresentar as structs que planeamos implementar. Sendo estas as seguintes:

```
struct student {
    char s_id[6];
    char c_id[2];
    char  status;
}


struct country {
    char c_id[2];
    int total_students;
    int total_students_dropped_out;
    int total_students_active;


}
```

- **Student**

  Para representar um **aluno**, implementaremos a *struct students*, com os campos:

  s_id que caracteriza o ID global de cada estudante e ocupa *6 bytes*;

  c_id que representa o ID do país no qual o aluno pertence, de tamanho *2 bytes*;

  status que identifica o estado actual do aluno, ocupando *1 bytes*

- **Country**

  Para representar um **país**, implementaremos a *struct country*, com os campos:

  c_id que caracteriza o ID de cada país e ocupa *2 bytes*;

  total_students que representa o número total de estudantes do país;

  total_students_dropped_out que identifica o número de alunos que abandonaram os estudos;

  total_students_active que representa o número total de alunos que ainda frequentam os estudos;

total_students_finished que contabiliza o total dos alunos que terminaram os estudos.

- **Countries_Hash_Table**

  Considerando as características associadas à *struct coutry*, em relação ao espaço ocupado e as operações que se irão realizar sobre os dados armazenados, pretende-se agrupa-los numa **Hash Table** na memória principal durante a execução do programa. Aquando do término da aplicação, se já existir um ficheiro com o objectivo de guardar os dados desta estrutura, o ficheiro será actualizado, se não o ficheiro irá ser criado e os dados serão inseridos. .

  Esta estrutura irá utilizar como chave para realizar comparações, nas operações requisitadas, o elemento ***c_id***. Será inicializada com 397 posições, que é o número primo mais seguinte ao dobro dos 193 estados membros que pertencem à UNESCO.

  Cada elemento(struct country) desta estrutura, excepto o ID, é um *inteiro de 4 bytes*. Desta forma, temos uma soma de 18 bytes por país e um total de 0.076621 *MegaBytes*.

- **Students_Hash_Table**

  Devido ao grande número de alunos que poderá estar nos sistema, 10 milhões, não encontramos outra forma de guardar a informação para todos estes alunos, sem utilizar memória secundária. Por isto, guardamos as structs student em Sorted Doubly Linked Lists, as quais estão relacionadas com posições dentro da Hash Table Students.

  Portanto. Cada posição desta Hash Table, irá apenas conter um inteiro que nos indica a posição no ficheiro, onde estará a Doubly Linked List em que a struct a encontra se localiza. Ao ser inicializada esta estrutura, cada elemento irá ser inicializado com um char, e assim que for atribuído um valor na memória será convertido para inteiro.

  Tendo em conta o que cada elemento desta Hash Table irá conter, e com um tamanho de 15000017(número primo mas próximo de 1.5 vezes o número possível de alunos no sistema) que nos pareceu apropriado dado à nossa forma de lidar com colisões ao inserir alunos no sistema.

  Se se encher o sistema com 10 milhões de alunos, obtemos $15^6 \times 4$ bytes ou seja 60 MegaBytes. Se apenas 5 milhões de alunos forem inseridos na aplicação, então obtemos $7.4^6 \times 1$ byte $+7.4^6 \times 4$ bytes ou seja 37.5 *megabytes*

- **Sorted_Doubly_Linked_List**

  Esta estrutura irá conter na sua head, posição no ficheiro para a qual o elemento da estrutura **Students_Hash_Table** aponta. O tamanho da Doubly Linked List, um pointer para a sua tail, e um pointer para o nó seguinte. Os restantes nós da Doubly Linked List contém um nó que contém:

- Pointer para o nó seguinte(4 bytes);

- Pointer para o nó anterior(4 bytes);

- Struct student(9 bytes);

Devido ao algoritmo de hash escolhido(*FNV-1*) não esperamos ter muitas colisões, portanto para esta estrutura iremos apresentar cálculos para as seguintes possibilidades: **1.** Cada Doubly Linked List só contém 35×10^3 alunos; **2.** Uma Doubly Linked List para cada aluno.

-1: $10^6/(35 \times 10^3)$ dá aproximadamente 286 Doubly Linked Lists. Sendo que o primeiro nó terá 2 inteiros e short, e cada nó após este irá conter, 2 inteiros mais a struct student que ocupa 9 bytes. Obtemos 10 bytes $\times 286 + 17$ bytes $\times 10^6 = 170.00286$ megabytes

-2: Sendo que o primeiro nó terá 2 inteiros e short, e cada nó após este irá conter, 2 inteiros mais a struct student que ocupa 9 bytes. Obtemos 10 bytes $\times 10^6 + 17$ bytes $\times 10^6 = 270$ megabytes

Ambos os casos cabem perfeitamente dentro dos limites da memória disponíveis.

Memória Total Utilizada

Memória principal: 60.076621 megabytes no pior caso e 37.576621 megabytes no pior caso
Memória secundária: 270 megabytes no pior caso e 170 megabytes no pior caso

3 Ficheiros de dados

O Sistema contará com 3 ficheiros de dados, um para armazenar os dados de cada país, outro para os dados dos dos alunos, e outro para armazenar a Hash Table relativa aos estudantes

4 Operações

Às operações requisitadas no enunciado, adicionamos uma nova operação que irá ser utilizada várias vezes durante o trabalho, de modo a simplificar este relatório e diminuir a sua extensão.

- **Retorna Doubly Linked List**

  Esta operação tem como objectivo retornar a Doubly Linked List guardada em disco, que esta relacionada a uma posição na Hash Table mantida em memória, onde um estudante seria inserido, removido ou ao qual se alteraria os dados.Os passos para esta operação são:

  1- O ID composto por 6 caracteres irá ser introduzido numa função de hash FNV-1, a qual irá retornar um hash value a utilizar para procurar na Hash Table student.

  2- Ao se ter a posição a aceder na memória secundária, a Doubly Linked List indicada na Hash Table é trazida para memória principal e retornada.

  Esta operação apenas realiza um acesso ao ficheiro, ao ir obter a Doubly Linked List guardada em disco.

  A complexidade temporal desta operação é constante(O(1)), visto que apenas calcula o hash value de acordo com o ID fornecido e tem acesso directo ao elemento que procura na Hash Table.

- **Inserção de um novo estudante**

  Esta operação vai calcular a posição na Hash Table Students onde está o valor da posição no ficheiro, da Doubly Linked List em que o aluno irá ser inserido. De forma a aumentar a velocidade do programa, o aluno será inserido de modo a mantar a Doubly Linked List ordenada

  1- Após a leitura do pedido de introdução de um novo estudante no sistema, será criada outra instância da struct student com os dados recebidos.

  2- A função **Retorna Doubly Linked List** é chamada e retorna a Doubly Linked List desejada.

  3- A Doubly Linked List irá ser percorrida e a struct student será inserida na posição adequada.

  4- A Doubly Linked List é actualizada no ficheiro

  IDs repetidos- Se durante este processo for encontrado algum ID igual ao do aluno que está a ser inserido no sistema, a função retorna que a

inserção não é possível pois já existe alguém com esse ID e pode ser um aluno activo ou não.

Esta operação irá sempre realizar 2 acessos ao ficheiro, pois irá necessitar de trazer a Doubly Linked List da memória secundária para fazer as alterações necessárias, e só depois voltar a guardar no ficheiro.

A complexidade temporal desta operação é constante(O(1)) para descobrir a posição na memória secundária da Doubly Linked List a utilizar, e linear(O(n)) para inserção, sendo n o tamanho da Doubly Linked List.

- **Remover um identificador**

  Esta operação funciona de forma semelhante à inserção de um novo estudante, sendo a única diferença que nesta operação a informação relativa ao estudante que tenha o ID fornecido é apagada do sistema.

  ID não encontrado- Caso não seja encontrado nenhum ID igual ao fornecido do Standard Input, então a função retorna que o aluno não pode ser removido do sistema pois não fazia parte do mesmo.

  Tanto o número de acessos ao ficheiro como a complexidade temporal é igual a operação de **inserção de um novo estudante no sistema**.

- **Assinalar que um estudante terminou o curso**

  Ao receber o pedido de alteração dos dados de um estudante para assinalar que terminou o curso. Esta função irá realizar os seguintes passos:

  1- Após a leitura do pedido de alteração dos dados do estudante. A função **Retorna Doubly Linked List** é chamada e retorna a Doubly Linked List associada ao estudante indicado.

  2- A Doubly Linked List é percorrida até encontrar o aluno correcto, e o char *char status* da struct do estudante é alterada para indicar que o estudante terminou o curso.

  3- A Doubly Linked List associada ao estudante é actualizada na memória secundária.

  ID não encontrado- Caso não seja encontrado nenhum ID igual ao fornecido do Standard Input, então a função retorna que o aluno não pode ser assinalado como alguém que terminou o curso, pois não pertence ao sistema.

  Esta função realiza 2 acessos ao ficheiro, um ao obter a Doubly Linked List e outro ao actualizar a Doubly Linked List no ficheiro.

  A complexidade temporal desta operação é constante(O(1)) ao encontrar a posição na memória onde a Doubly Linked List que contém os dados do aluno se encontra, e no pior caso, linear(O(n)) a percorrer a Doubly Linked List para encontrar o aluno.

- **Assinalar o abandono de um estudante**

  Esta função funciona exactamente da mesma forma que a função **Assina que um estudante terminou o curso**, excepto que ao alterar a struct student, altera para representar que o estudante abandonou o curso.

  Esta função realiza 2 acessos ao ficheiro, um ao obter a Doubly Linked List e outro ao actualizar a Doubly Linked List no ficheiro.

  A complexidade temporal desta operação é constante(O(1)) ao encontrar a posição na memória onde a Doubly Linked List que contém os dados do aluno se encontra, e no pior caso, linear (O(n)) a percorrer a Doubly Linked List para encontrar o aluno.

- **Obter os dados de um país**

  Esta função retorna os dados de um país após os mesmos terem sido requisitados. Os dados de cada país são actualizados cada vez que há alguma alteração aos alunos desse mesmo país. Os passos desta operação são:

  1- Cálculo do hash value do ID do país sobre o qual se quer obter os dados. A função de hash é FNV-1, a mesma função aplicada à Hash Table relativa aos estudantes.

  2- Após o calculo do hash value, o ID presente na posição da Hash Table indicada pela função de hash, irá ser comparado com o do país desejado, e se for igual, a função retorna os mesmos, se não for, irá fazer rehash do valor até encontrar o país desejado.

  Esta função irá realizar 0 acessos a ficheiros.

  A complexidade temporal desta função é constante(O(1)) uma vez que obtém a posição na Hash Table logo após ter realizado a função de hash.

5 Início e fim da execução

5.1 Início

No inicio da execução do programa, será feita uma verificação, de modo a averiguar se os ficheiros onde os dados deste sistema são guardados, já se encontram presentes no computador onde o programa foi inicializado. **Se** os ficheiros se encontrarem no computador, a aplicação irá ler e guardar os dados necessários em memória principal. **Se não** a aplicação irá inicializar as estruturas e ficheiros necessários.

Os ficheiros que esta aplicação procura no início da sua execução são:
1: "Students.data"que contém as Doubly Linked Lists onde as structs students são guardadas.
2: "Countries.data"que contém a Hash Table com toda a informação relativa aos países pertencentes ao sistema.
3: "Students_Hash_Table.data"que contém a posição das Doubly Linked Lists no ficheiro "Students.data".

Os dados dos nodes da Hash Table countries irão todos ser inicializados a 0, tanto as que forem *int*, ou as de *char* tendo estas o valor char indicado na tabela de ASCII para a posição 0.
O mesmo procedimento acontecerá com a Hash Table students.

Depois de ter realizado estes passos, a aplicação fica a aguardar pedidos do Standard Input, aos quais irá responder de acordo com o que foi previamente especificado.

5.2 FIM

Ao terminar a execução, a aplicação irá actualizar todos os ficheiros com os dados existentes na memória principal e limpar a memória principal.

6 Observações

A forma como decidimos implementar a aplicação pode ser um pouco mais lenta do que o esperado pelo docente, mas devido as restrições de memória, foi a forma que nos pareceu mais adequada realizar a implementação.

Devido ao tamanho das estruturas, poderá ser necessário algum alinhamento para memória secundária.

7 Bibliografia

Hash functions analysis 1
Hash functions analysis 2
Data structres and algoritghms time and space complexity
Hash Table data structre 1
Hash Table data strucure 2
Linked List data strucutre