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

Rúben Peixoto (nº 37514)

Junho de 2020

Resumo

Neste relatório será falado do que realmente foi implementado na segunda fase. Nesta fase final, tal como na primeira fase, falarei das estruturas de dados que foram utilizadas assim como as suas operações e performance. Também será falado da interacção entre as estruturas de dados e os ficheiros que foram usados.

1 Estruturas de dados

Ao contrário do que foi escrito na primeira fase do relatório, nesta fase as estruturas de dados usadas foram uma BTree e uma HashTable.

A razão pela qual a BTree foi escolhida, deve-se ao facto de que esta foi desenhada especificamente para diminuir ao máximo o número de acessos ao disco. Relativamente à HashTable, considero que esta estrutura desempenhe um papel importante neste projecto porque, cada índice desta vai ter informação sobre um país (código do país, o número de alunos activos, o número de alunos que desistiram do curso e o número de alunos que terminaram o curso). Esta estrutura evitará assim, sempre que seja necessário saber os dados de um determinado país, de percorrer todos os nós da BTree, tornando o programa mais eficiente.

Tanto como a raiz da BTree como a HashTable e as configurações da BTree estarão em memória central e em memória secundária durante o programa todo, os restantes nós da BTree estarão guardados em memória secundária, porém haverá situações onde esses terão que ser criados e/ou manipulados em memória central.

HashTable

Relativamente à HashTable, neste programa foi denominada "HashCountry", e cada elemento desta tem a seguinte configuração:

```
typedef struct hash_node
{
    bool occupied;   // Indica se uma posição da HashTable está livre
    char country[3]; // Código do país
    int corr,        // Indica se o aluno continua a estudar
        diplo,       // Indica se o aluno terminou o curso
        aband;       // Indica se o aluno abandonou o curso
} HashNode;
```

Como se pode ver esta struct não contém o número total de alunos, aliás, esta informação está implícita na sua definição. O número total de alunos é calculado quando se pede a informação sobre um determinado país, através da soma do número de alunos activos com o número de alunos diplomados e o número de alunos que desistiram do curso.

A HashTable tem a seguinte estrutura:

```
typedef struct hash_country
{
    int ocupacao;     // Número de espaços preenchidos na HashTable
    HashNode h[751]; // Array com as informações dos países
} HashCountry;
```

Cada HashNode ocupa:

$$Tamanho = bool + char[3] + int + int + int$$

$$Tamanho = 1 + 1 * 3 + 4 + 4 + 4$$

$$Tamanho = 16\ bytes$$

Com isto o HashTable ocupará:

$$Tamanho = int + 751 * HashNode$$

$$Tamanho = 4 + 751 * 16$$

$$Tamanho = 12020\ bytes$$

O número de elementos da HashTable é de 751 porque embora o número de elementos necessários sejam apenas:

$$Número\ de\ elementos\ necessários = letras\ do\ alfabeto * letras\ do\ alfabeto$$

$$Número\ de\ elementos\ necessários = 26 * 26$$

$$Número\ de\ elementos\ necessários = 676\ elementos$$

achou-se melhor adicionar 75 posições e assim, ao aumentar a quantidade de espaço, tentar diminuir o número de colisões.

BTree

Para a BTree a definição desta e dos seus nós permanecem iguais assim como a struct dos estudantes.

Por isso a definição de Student é:

```
typedef struct student
{
    bool completed,    // Indica se o aluno terminou o curso
         dropped_out;  // Indica se o aluno desistiu
    char country[3],   // Código de um país
         global_id[7]; // Identificação de um aluno
} Student;
```

O espaço ocupado por um Student é:

$$Student = bool + bool + char[3] + char[7]$$

$$Student = 1 + 1 + 1 * 3 + 1 * 7$$

$$Student = 12\ bytes$$

Embora o código do país ocupe apenas 2 bytes e a identificação do país 6 bytes, é necessário colocar 1 bytes extra, para identificar o termino da palavra. Assim, o código do país terá 3 bytes e a identificação 7 bytes.

A struct "BTree" terá um apontador para o nó da raiz. O nó da BTree vai ter o seguinte formato:

```
typedef struct bnode
{
    bool folha;              // Indica se o nó é folha da BTree
    int ocupacao,            // Número de elementos (Estudante)
        indice_ficheiro,     // Índice deste nó no ficheiro
        filhos[2T];          // Índices dos nós filhos no ficheiro
    Student elementos[2T-1]; // Array de estudantes
} BNode;
```

Considerando que os nós da BTree vão ser inseridos em memória secundária, e que cada página dessa memória ocupa 4096 bytes então, cada nó terá que ter um tamanho muito próximo ao tamanho da página. Considerando que o processador é de 32 bits, e embora a folha seja um booleano e este ocupe 1 byte, devido ao alinhamento em memória e ao facto de que os restantes elementos encontram-se alinhados em pacotes de 4 bytes, vai ser necessário inserir 3 bytes vazios junto aquele (booleano).

Com isto, consegue-se calcular o grau de ramificação mínimo "T" que é:

$$folha + ocupacao + indice\_ficheiro + filhos * (2T) + elementos * (2T - 1) = 4096$$

$$4 + 4 + 4 + 4 * (2T) + 12 * (2T - 1) = 4096$$

$$12 + 8T + 24T - 12 = 4096$$

$$32T = 4096$$

$$T = 4096/32$$

$$T = 128$$

Como o grau de ramificação mínimo é 128 o nó ocupará 4096 bytes.

2 Ficheiros de dados

Tal como escrito no relatório da primeira fase, continua-se a usar dois ficheiros para o programa. Só que num ficheiro, chamado "index_file", para além de conter as configurações da BTree, também contém a HashTable falada anteriormente, vide Figura 2. Na Figura 1, continua a ser um ficheiro, chamado "bnodes_file", com todos os nós da BTree.

Quanto aos acessos a disco, os momentos em que irá ser necessário aceder a disco são:

1. No início do programa, para buscar as informações que estão no cabeçalho (BtreeConf), a HashTable e o nó da raiz da BTree.

2. Durante a execução do programa, para guardar, aceder e remover conteúdos dos ficheiros que contêm os nós da BTree, as configurações da Btree e a HashTable.

3. No fim do programa para guardar/actualizar as informações do cabeçalho do ficheiro e a HashTable.

Todas as operações de acesso a disco terão custo constante.

Ficheiro com os nós da BTree

Este ficheiro conterá todos os nós da BTree. De início a raiz do nó estará na posição zero do ficheiro, no entanto, à medida que se vai removendo e adicionando nós, o índice da raiz pode mudar.

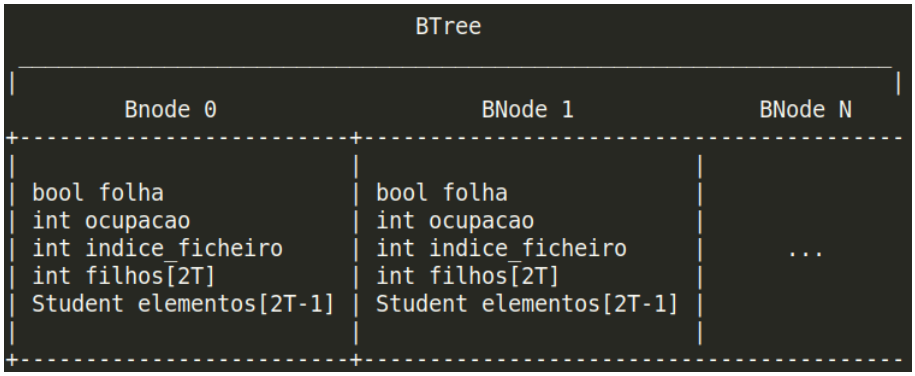

Figura 1: Ficheiro com os nós de uma BTree

Ficheiro com as configurações da BTree e HashTable

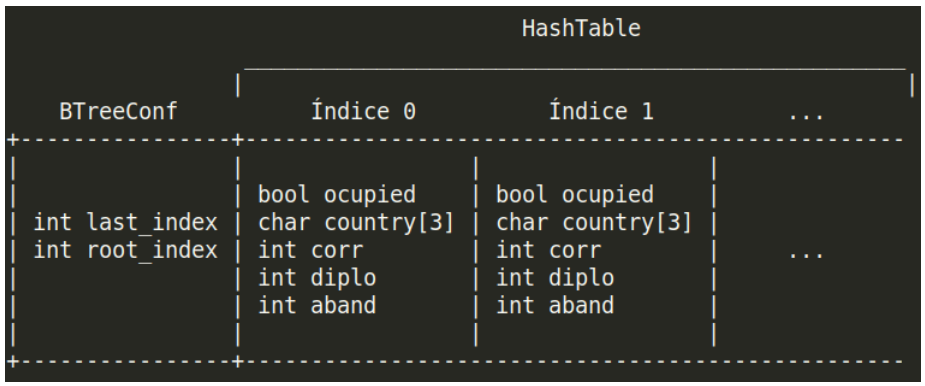

Figura 2: Ficheiro com as configurações da BTree mais a HashTable

Neste ficheiro temos 8 bytes dedicados às configurações da BTree. Estas configurações consistem em identificar a posição do nó da raiz, "root_index", para conseguirmos entrá-lo sempre que iniciarmos o programa, e um valor inteiro, "last_index" que mostra qual foi o último índice atribuído a um nó. Este valor é importante para conseguir identificar os futuros nós.

A seguir aos 8 bytes temos 12020 bytes, tal como calculado anteriormente, dedicados para a HashTable.

3 Operações

Excepto a operação "Obter dados relativos a um país", que usa apenas a HashTable, as restantes, usam as operações da BTree como operações da HashTable.

Inserção de um novo estudante

Para inserir um estudante, o programa de receber dois inputs: a identificação global de um estudante e o código de um país. Posto isto o programa cria um Student e vai guardá-lo na BTree usando a função correspondente para o efeito. Caso o estudante exista, tenha terminado o curso, ou tenha abandonado o curso, o programa imprime uma mensagem de acordo com uma das situações e este adicionado à BTree. Caso o estudante não exista, o programa adiciona o estudante na BTree. Depois de adicionado o estudante, o programa vai ver se existem informações relativas ao código do país do estudante na HashTable. Se existir, o programa adiciona 1 ao número de alunos activos,

caso contrário, o programa cria um HashNode apenas com 1 estudante activo e o código do país e, de seguida, adiciona esta informação à HashTable.

A complexidade para adicionar um estudante na BTree é:

$$\theta(T * log_T(n))$$

em que "T" = grau de ramificação mínimo e "n" = quantidade de estudantes.

No pior dos casos, a complexidade para adicionar ou pesquisar a informação de um país na HashTable é:

$$\theta(H)$$

em que "H" = tamanho da HashTable.

No pior dos casos o programa tem que adicionar o estudante na BTree, pesquisar se a HashTable tem as informações de um determinado país e, por fim, caso não exista, adicionar a informação do novo país à HashTable. Por isso a complexidade do programa para adicionar um estudante é:

$$Adicionar\ BTree + Pesquisar\ HashTable + Adicionar\ HashTable$$

$$\theta(T * log_T(n)) + \theta(H) + \theta(H)$$

$$\theta(T * log_T(n))$$

Remover um identificador

Para remover um estudante o programa recebe a identificação global de um estudante através do input. De seguida, através das funções da BTree, vai se tentar remover um estudante da BTree. Caso a identificação não exista, o aluno associado a essa identificação tenha terminado o curso, ou o aluno tenha abandonado o curso, o programa vai imprimir um output de acordo com as três situações apresentadas. Caso o estudante exista e este esteja activo, será possível remover as informações associadas a essa identificação. Caso o estudante seja removido, o programa vai verificar se existe informações do país do estudante na HashTable. Se não existir não faz nada, caso contrário o programa retira 1 ao número total de estudante activos.

A complexidade para remover um estudante na BTree é:

$$\theta(T * log_T(n))$$

em que "T" = grau de ramificação mínimo e "n" = quantidade de estudantes.

No pior dos casos, a complexidade para pesquisar a informação de um país na HashTable é:

$$\theta(H)$$

em que "H" = tamanho da HashTable.

No pior dos casos, o programa tem que remover o estudante na BTree, pesquisar na HashTable para encontrar as informações de um determinado país e, por fim, caso exista essa informação, alterar a informação do novo país na HashTable. Por isso a complexidade do programa para remover um estudante é:

$$Remover\ BTree + Pesquisar\ HashTable + Alterar\ Informação$$

$$\theta(T * log_T(n)) + \theta(H) + \theta(1)$$

$$\theta(T * log_T(n))$$

Assinalar que um estudante terminou o curso

Para assinalar que um estudante terminou o curso, o programa recebe, como input, a identificação de um estudante. A seguir, através da pesquisa na BTree, vai buscar o nó da BTree que contém esse estudante e coloca-se a variável que indica que o aluno terminou o curso a "true" e volta-se a guardar a informação em disco. Com isto, vai se à HashTable buscar a informação relativa ao país do estudante e retira-se 1 ao número de estudantes activos e adiciona-se 1 ao número de estudantes que terminaram o curso. Para calcular a complexidade deste algoritmo temos que decompô-lo em 2 passos essenciais: Buscar o estudante à BTree e pesquisar na HashTable.

Logo, a complexidade de pesquisar na BTree é

$$\theta(T * log_T(n))$$

, em que "T" é o grau mínimo de ramificação e "n" é o número de estudantes, e para pesquisar na HashTable, no pior dos casos, é

$$\theta(H)$$

, em que "H" é o número de elementos da HashTable. Posto isto, a complexidade deste algoritmo é

$$\theta(T * log_T(n))$$

Assinalar que um estudante abandonou o curso

Para assinalar que um estudante abandonou o curso, o programa recebe, como input, a identificação de um estudante. A seguir, através da pesquisa na BTree, vai buscar o nó da BTree que contém esse estudante e coloca-se a variável que indica que o aluno abandonou o curso a "true" e volta-se a guardar a informação em disco. Com isto, vai se à HashTable buscar a informação relativa ao país do estudante e retira-se 1 ao número de estudantes activos e adiciona-se 1 ao número de estudantes que abandonaram o curso. Para calcular a complexidade deste algoritmo temos que decompô-lo em 2 passos essenciais: Buscar o estudante à BTree e pesquisar na HashTable.

Logo, a complexidade de pesquisar na BTree é

$$\theta(T * log_T(n))$$

, em que "T" é o grau mínimo de ramificação e "n" é o número de estudantes, e para pesquisar na HashTable, no pior dos casos, é

$$\theta(H)$$

, em que "H" é o número de elementos da HashTable. Posto isto, a complexidade deste algoritmo é

$$\theta(T * log_T(n))$$

Obter dados relativos a um país

Esta operação recebe como input o código de um país, e o objectivo desta função é aceder à HashTable e buscar as informações relativas a esse código. Se não tiver informação sobre esse país é imprimido "H", caso contrário é imprimido uma mensagem com o seguinte formato:

A complexidade deste algoritmo, equivale à complexidade de se pesquisar na HashTable, por isso, no melhor dos casos, que equivale a encontrar o elemento logo no primeiro valor do hashCode obtido, é $\theta(1)$, no pior dos casos é $\theta(H)$, em que "H" é o número de elementos da HashTable.

4 Início e fim da execução

No início da execução do programa, são abertos dois ficheiros, um com as definições da BTree (índice do nó da raiz no ficheiro "bnodes_file" e a ultima posição atribuída a um nó) mais a HashTable, e o outro ficheiro com os nós da BTree.

Depois dos ficheiros abertos, vai-se buscar as configurações da BTree e a HashTable ao ficheiro "index_file" e coloca-se em memória principal. De seguida, como já se tem as configurações iniciais da BTree, vai se buscar o nó da raiz da BTree ao ficheiro "bnodes_file" e coloca-se em memória principal. Caso o programa esteja a executar pela primeira vez, os ficheiros "index_file" e "bnodes_file" são criados no momento, a raiz da BTree vai estar no índice zero no ficheiro "bnodes_file" e a HashTable vai estar vazia.

Quando estes dados estiver prontos, o programa executa as operações requeridas pelo enunciado.

No fim da execução do programa, tanto as configurações da BTree como a HashTable serão guardadas em memória secundária, no ficheiro "index_file". Com isto libertam-se: a BTree, as configurações da BTree e a HashTable da memória principal. Por fim, fecham-se os ficheiros e o programa termina.

5 Bibliografia

1. Slides das aulas teóricas do professor Vasco Pedro.
2. Introduction to Algorithms, Third Edition.
3. HashCode, https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#hashCode--