Trabalho Prático 1:
Banco de Dados com Árvore B+

Algoritmos e Estrutura de Dados III

Dayman Novaes - daymannovaes@gmail.com

1 Introdução

Esse trabalho é a implementação de um banco de dados utilizando Árvore B+. Ao guardar registros em ordem, qualquer que seja a ordem e estrutura escolhida, quando o volume de dados é muito grande em relação à memória RAM do computador, devemos adotar estratégias e utilizar da memória externa do computador, isso é, o HD, por meio de arquivos.

Porém, precisamos aproveitar **ao máximo** a RAM do computador, ou memória primária, pois é a memória mais rápida a que temos acesso. Por isso, a cada leitura no HD, devemos trazer o máximo de dados possível para a RAM para, a partir daí, realizar a busca em memória primária que, em termos proporcionais, terá custo constante.

Porém, também devemos levar em conta que precisamos acessar o mínimo de vezes possível a memória externa. Por isso, uma árvore binária comum não seria eficiente, pois ela tem muitos níveis, e cada nível representa um acesso demorado ao HD.

Para resolver o problema, a estrutura mais aconselhada é a Árvore B (ou B+), que é uma espécia de árvore binária (na verdade é uma árvore n-ária) porém espalhada. É como fosse pego uma árvore binária comprida e a achatasse totalmente, aumentando sua largura porém diminuindo sua altura. Exemplos serão mostrados na próxima seção.

2 Modelagem do Problema

Analisando as figuras 1 e 2 claramente podemos perceber o potencial da árvore B. A árvore binário no exemplo tem quinze registros e já possui quatro níveis (que tornaria necessário quatro acessos à memória no pior caso). Já a árvore B em questão tem 34 registros, mais que o dobro, e possui três níveis, um a menos que a árvore binária.

É interessante notar que essa árvore B tem uma capacidade máxima de quatro registros por página, isso é, cada acesso á memória secundária carrega no máximo quatro registros à memória RAM. Esse número é chamado de **ordem da árvore**. Essa árvore tem ordem 4.

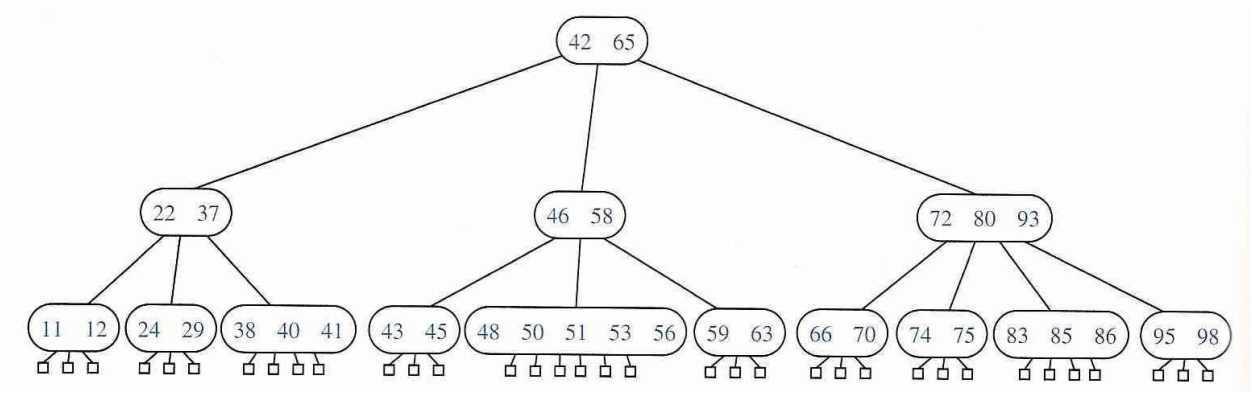

Figura 1: Árvore B de ordem 4

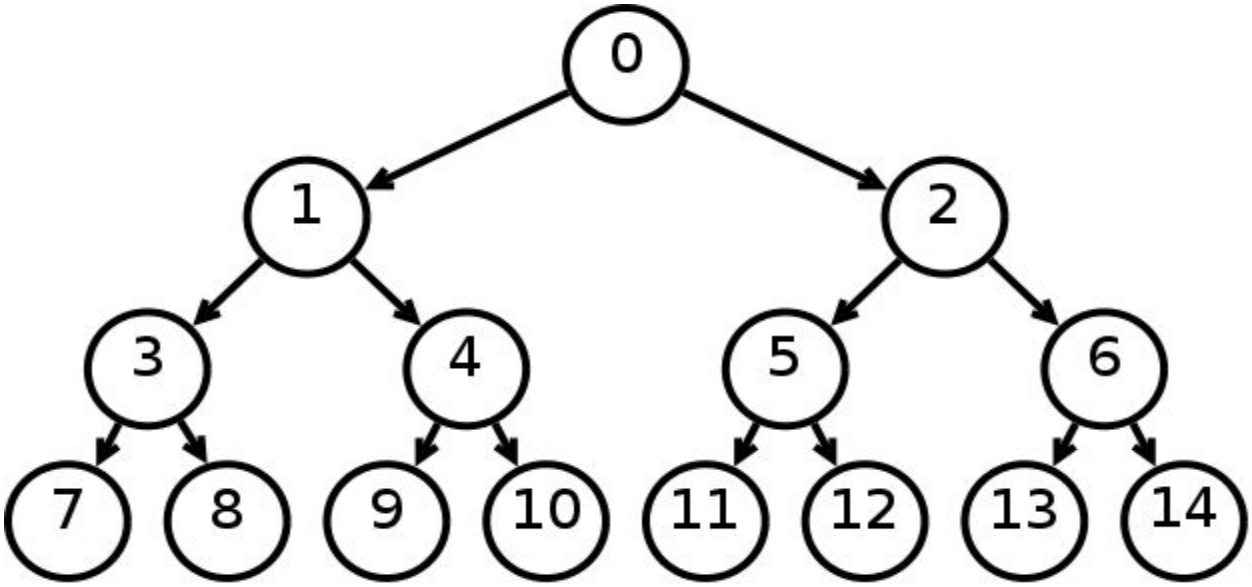

Figura 2: Árvore binária

Esse número deve ser escolhido para tornar mínimo o número de acessos ao HD, ou seja, para tornar a altura da árvore o menor possível, e para carregar o maior número de registros possíveis na RAM também.

No caso desse trabalho, a ordem da árvore B+ deve ser passado por parâmetro na execução do programa, juntamente com os arquivos de entrada, saída. Os dois últimos parâmetros são a quantidade de campos que cada registro do banco de dados terá, e qual campo corresponde ao identificador único do registro. Como no exemplo abaixo:

```
./tp1.exe saida.txt entrada.txt 100 4 0
```

2.1 Estrutura da árvore

A estrutura núcleo (representado em negrito no código 1) de cada página da árvore é relativamente simples, porém existem outras variáveis para auxuliar no funcionamento do programa, especialmente por se tratar de um armazenamento externo.

Código 1: estrutura da árvore B+

```
typedef struct Page {
        int isLeaf;                     // se é uma página interna ou não

        int count;                      // número de registros nessa página
        Pair *data;                     // vetor de registros(chave|valor)
        PagePointer *pointers;          // vetor de páginas filhas dessa página

        int *ids;                       // identificadores dos arquivos filhos
        int id;                         // identificador do arquivo dessa página

        PagePointer parent;             // aponta para a página pai
} Page;
```

Os registros da página, como pode-se observar, não são compostos apenas de um valor inteiro, que representaria a chave do registro inserido no banco de dados, porém de dois inteiros: **chave** e **valor**. O **valor** representa a posição do registro no arquivo de banco de dados, melhorando ainda mais a eficiência de nosso programa. Mais detalhes serão dados na seção 2.2.

2.2 Outras estruturas

- Para fins semânticos, foi criado também um tipo de dados chamado Árvore (**Tree**), que é como uma página qualquer, tirando que essa deve ser usada apenas para representar a página raiz.

```
typedef struct Page Tree;
```

- A estrutura **PagePointer**, como o próprio nome já diz, é apenas um tipo ponteiro de página.

```
typedef struct Page *PagePointer;
```

- A estrutura **Pair** corresponde a uma estrutura de chave e valor. Essa abstração é importante não só para facilitar o desenvolvimento e o entendimento do código, mas também para diminuir a quantidade de parâmetros passados entre as funções.

```
typedef struct Pair {
        int key;                    // chave do registro
        int value;                  // número da linha do registro
} Pair;

// função para criar um Pair a partir de uma chave e valor
Pair *create_pair(int key, int value) {
        Pair *pair;
        pair = malloc(sizeof(Pair));

        pair->key = key;
        pair->value = value;

        return pair;
}
```

- Nesse trabalho, o parâmetro key representado na struct anterior, sempre corresponde também à uma chave de algum registro inserido anterior no banco de dados. Esse registro, que possui uma chave e vários campos, é representado pela struct **Record**.

```
typedef char Field[MAX_BUFF];     // field representa uma string
typedef struct Record {
        int key;
        int count;                // número de campos nesse registro
        Field *fields;            // cada registro tem vários campos
} Record;
```

2.3 Método de inserção e pesquisa

Como já explicado, utilizamos duas estruturas de arquivos para guardar os dados. Os registros em si são guardados sequencialmente em um arquivo próprio, em ordem de inserção, isto é, o segundo arquivo a ser inserido ocupa a segunda linha do arquivo, por exemplo.

E a outra estrutura de arquivos é para a árvore B+.

2.3.1 Inserção

A inserção segue uma lógica simples. Uma vez identificado o comando de adicionar (**add**), lê as próximas **n** entradas, correspondentes aos campos do registros e salva em uma nova linha do arquivo de registros, e como um novo registro na árvore B+. A chave do novo registro será a chave do registro inserido, e o valor da chave será a linha, isso, o número de registros já inseridos + 1.

Código 2: adiciona registro

```
seja registro um Registro

para i igual a 0, até i igual a n, faça
        seja campo um novo campo lido do arquivo
        se i igual ao indice da chave
                chave do registro = campo
        senão
                insere campo no registro

insere registro no arquivo de registros
Insere registro na árvore B+
```

Já a inserção na árvore segue a lógica de inserção em Árvores B+, porém, ao escolher qual página inserir, devemos primeiramente carregá-la da memória, inserir o registro, e depois escrevê-la na memória.

2.3.2 Pesquisa

A pesquisa em árvore B+ também é tão simples quanto uma pesquisa em árvore binária. Mas ao invés de apenas decidir se devemos ir para a esquerda ou direita, devemos escolher qual dos ***n*** filhos devemos entrar. Porém, como já explicado brevemente na seção anterior, antes de entrar na próxima página, devemos carregá-la da memória, para depois realizar a busca sequencial dentro da página.

Código 3: pesquisa registro

```
seja page a página raiz
seja key a chave do registro

enquanto page não é uma folha
        seja index o índice do filho correto de page (por busca sequencial em page)
        carrega da memória o filho de índice index de page
        seja page o filho de índice index de page

seja index o índice do filho correto de page (por busca sequencial em page)

seja row o value de índice index de page
seja record o registro na linha row no arquivo de registros

imprime record
```

3 Análise de Custo

Nessa seção será apresentado a análise de complexidade e custo teóricos tanto de tempo quanto para espaço. O custo do programa, por trabalhar com memória externa, reside basicamente nas duas estruturas que trabalham com essa memória externa, que é a Árvore B+ e o arquivo de registros sequencial.

3.1 Análise de Custo de Tempo

A primeira análise pode ser feita referente à estrutura de árvore B+.

- Sendo ***n*** o número de registros e ***m*** a ordem da árvore, sabemos que o caminho mais longo possível a ser percorrido dentro da árvore é:

  $$log_{m+1} n$$

  Que é igual ao número de acessos à memória secundária no pior caso.

- Já o custo de leitura no arquivo de registros possui um problema que foi tardiamente percebido e que é explicado em detalhes na seção **4 - dificuldades**.

  Apesar da árvore B+ possuir o **número da linha** na qual o registro está inserido, devido à uma limitação de sistema operacional e de implementação, não podemos simplesmente "pular" ***k*** linhas no arquivo, a menos que soubessemos o tamanho exato de cada linha,o

que não ocorre, uma vez que podemos ter uma chave com dois, três ou quatro digitos em diante. Devido a esse fato, o acesso no pior caso é ***n*** para ***n*** registros.

- Outro ponto interessante, é que a operação mais cara referente à árvore B+ é a divisão de uma página e a inserção do registro no pai, pois é uma operação em que ocorre alguns rearranjos na árvore. Para melhorar esse problema, há a técnica de **overflow**, que é explicada com detalhes também na seção **4 - dificuldades**.

3.2 Análise de Custo de Espaço

- Analitcamente é observado que o espaço ocupado por uma árvore B+ após ***n*** registros inseridos, é aproximadamente 69% do espaço antes reservado à ela.
- O arquivo de registros ocupa o espaço de ***n*** vezes o tamanho de um registro. Sendo ***n*** o tamanho do registro, representando pelas strings de seus campos.

4 Dificuldades

Esse trabalho apresentou uma série de obstáculos durante o seu desenvolvimento, sendo um dos trabalhos mais desafiadores já feitos por mim até o momento. A implementação da árvore B+ externa foi uma das maiores dificuldades, pois ela apresenta uma série de nuances e múltiplos casos especiais que devem ser tratados com atenção. E o fato da inserção ter momentos de recursividade, em vários momentos ocorreu de um ajuste para consertar um erro, desfazer um outro previamente consertado.

4.1 Perca de eficiência com arquivo sequencial

Ao escolher a estratégia de usar um arquivo sequencial para salvar os registros, fez com que o programa perdesse a grande eficiência logarítmica da árvore B+, para uma eficiência linear.

Esse fato pode ser facilmente contornado com duas possíveis implementações, mas que não puderam ser postas em práticas por falta de tempo:

- Não usar um arquivo diferente para salvar os registros, e salvar os registros diretamente na árvore. Bastaria incluir a struct de *record* dentro da struct de *page*. Dessa forma, ao encontrar a página correta de pesquisa, já teríamos em mãos o registro requerido.
- Não incluir a *chave* no arquivo de registros, dessa forma todas as linhas teriam **o mesmo** tamanho, possibilitando uma busca direta no arquivo. Apesar de ser mais uma pesquisa em arquivo, essa pesquisa seria realizada em tempo constante, mantendo a eficiência logarítmica da árvore.

4.2 Serialização e desserialização

Esse é um processo fácil de implementado, até porque já é explicado uma ótima forma de implementação na especificação do TP. Porém esse processo teve que ser deixado de lado pelo mesmo motivo da dificuldade acima.

4.3 Técnica de overflow

Uma outra **feature** que desejava implementar na árvore, mas que não foi possível também, era a técnica de **overflow**. Pois, como mencionado na seção **3.1**, a operação mais cara na árvore B+ é a divisão de uma página.

Essa técnica consiste em, ao observar a necessidade de dividir uma página, olhar as páginas vizinhas, para saber se há algum espaço para inserir o novo registro apenas com um *rearranjo* de registros, que é uma operação mais eficiente do que separar uma página.

5 Conclusão

Como já explicado na seção anterior, esse trabalho foi muito desafiador e difícil de concluir, porém foi de grande valia para o meu aprendizado, tanto quanto acadêmico, quanto disciplinar. É interessante (e também importante para o crescimento profissional) aprender novas estruturas de dados que resolvem problemas do mundo real de forma **eficiente**, que é o caso da árvore B+, que é uma adaptação da árvore binária e se encaixa **exatamente** nos desafios existentes e mostrados na introdução desse trabalho.