Trabalho Prático de AEDS 3 – Banco de Dados com Árvores B+

Nome: Ronald Davi Rodrigues Pereira
Turma: TD1
Matrícula: 2015004437

Introdução

Atualmente, há um busca muito grande por algoritmos que sejam mais eficientes frente à outros, pelo simples fato de se otimizar o tempo de execução de certa operação. Alguns algoritmos de operações básicas em bancos de dados (inserção, remoção, busca e impressão) já são implementados à muitos anos atrás. Porém, muitos deles não são algoritmos ótimos em relação às novas implementações que foram surgindo ao longo do tempo. Desse modo, a implementação de uma árvore B+ inclui a questão de ser um algoritmo relativamente ótimo em relação a outras implementações já existentes.

Em bancos de dados atuais, muitos deles utilizam esse formato de implementação como esse trabalho prático: banco de dados em mémória secundária para realizar todas as operações. Logo, essa implementação de uma árvore B+ simulando um banco de dados de chaves de tipo número inteiro, é uma boa implementação para se realizar.

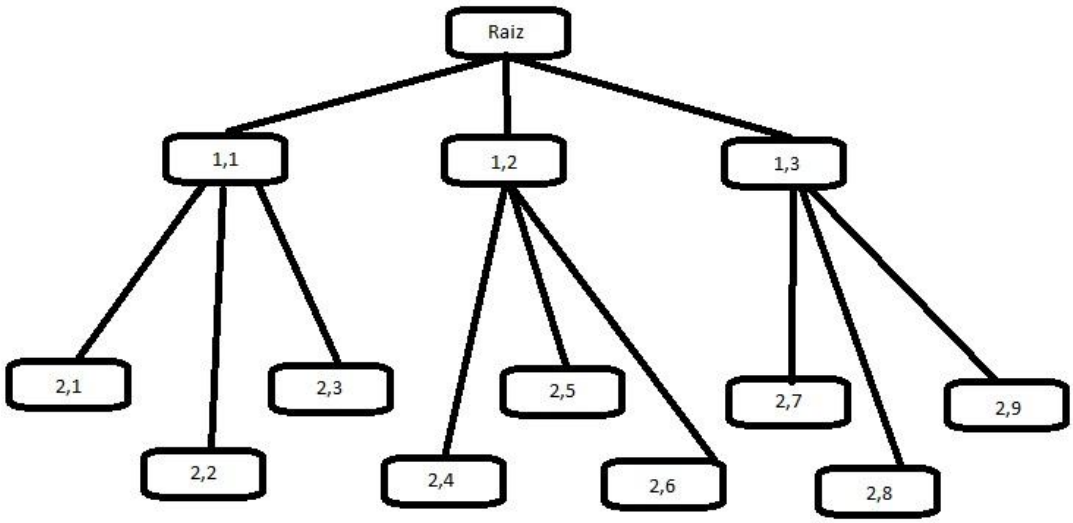
Para tal implementação, foram necessários muitos conhecimentos em relação a funcionamento de árvores, mais especificamente de uma árvore B+, Tipos Abstratos de Dados e funcionamento de arquivos (Files) na linguagem C.

A intuição da solução se baseia basicamente em encontrar um tipo de identificador para que a montagem da árvore seja efetiva ao longo da execução do programa com base em armazenamento em memória secundária.

Solução do Problema:

O algoritmo que resoluciona esse trabalho prático se baseia basicamente na identificação dos nodos internos da árvore B+ no arquivo temporário. As operações de busca, inserção e impressão são baseadas nessa identificação para que o caminhamento na árvore aconteça.

Abaixo segue a imagem ilustrando uma exemplificação da utilização desses indicadores no arquivo de texto gerado pelo programa em sua execução.



Nesse esquema, é utilizada uma árvore de ordem = 3. Logo, a raiz deverá ser mantida em memória primária em toda execução do programa (conforme foi pedido pela especificação do Trabalho Prático). Porém, ao longo que o número de chaves inseridas na árvore cresce, o número de filhos consequentes em níveis inferiores da árvore também aumentam.

Portanto, o algoritmo para se encontrar o caminhamento de acordo com os filhos é:

Identificador: (Nível da árvore,Índice de contagem) (por exemplo, 1,1 está no nível 1 da árvore e é a primeira célula no nó).

Exemplo da estruturação do arquivo:
1,1 – 8,15,
1,2 – 26,89,
1,3 – 101,150,
2,1 – 1,3,
2,2 – 8,12,
2,3 – 15,17,
2,4 – 23,24,
2,5 – 26,81,
2,6 – 89,91,
2,7 – 92,101,
2,8 – 148,149,
2,9 – 150,282,

E a raiz na memória principal conteria os números 17 e 91, por exemplo.

Desse modo, o caminhamento na árvore por meio da leitura desse arquivo seria de fácil efetividade uma vez que esse caminhamento segue a regra de Filho[min]=(índice*ordem)-(ordem-1) e Filho[max]=(índice*ordem).

A operação de busca (comando search) seria a efeticação desse carregamento dos nodos internos na memória principal e o seu consequente caminhamento.

A operação de inserção (comando add) utilizaria o comando Search para fazer o caminhamento na árvore e ao encontrar a folha aonde a chave deve ser inserida, ela é inserida nessa folha.

A operação de impressão (comando dump) só imprimiria o arquivo temporário, excluindo os níveis da árvore e os índices.

Análise de complexidade:

A análise de complexidade será feita primeiramente em função da variável *n*, que representa o número de operações relevantes dadas (atribuições).

TipoPagina *AlocaPagina(int ordem, int field): Essa função realiza somente a alocação dinâmica da página da árvore, definida no arvoreb+.h. Logo, essa é uma função que possui duas atribuições, sendo então O(2). Simplificando, essa função tem um limite assintótico superior igual a **O(1).**

TipoPagina *DesalocaPagina(TipoPagina *Pagina, int ordem, int field): Essa função realiza a desalocação do espaço alocado anteriormente. Logo, por fazer duas operações relevantes de free(), essa função é O(2). Simplificando, essa função tem um limite assintótico superior igual a **O(1).**

void BuscaRegistro(FILE *reg, FILE *saida, int ordem, int field, unsigned long long int chave): Essa função faz a busca do registro dado como parâmetro pela operação search e imprime no arquivo de saída a formatação exata dela com o registro, caso encontrado nas estruturas da árvore. Logo, pela busca do arquivo, essa função é O(n), sendo n o número de registros que foram inseridos no programa e que estão registrados no arquivo. Logo, essa função tem um limite assintótico superior igual à **O(n).**

int main(): O programa principal é regido pela função de maior complexidade, uma vez que a soma de todos as complexidades das funções que ele chama é a que rege a sua complexidade. Logo, temos que **O(n)** é a maior complexidade de suas funções, sendo esse então a complexidade da função principal do programa.

Avaliação experimental:

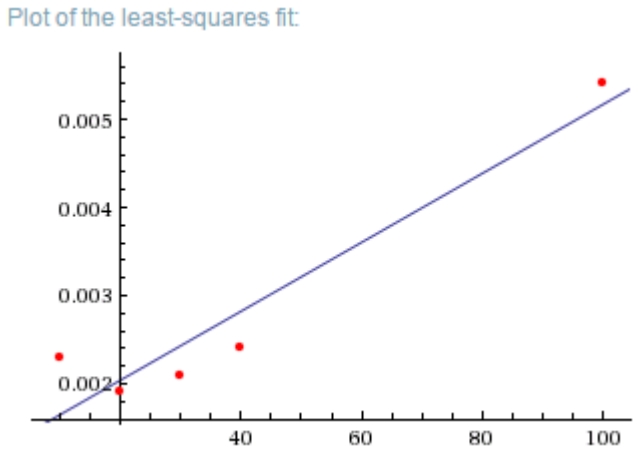
Vários testes foram realizados com o programa de forma a verificar o seu funcionamento. O teste que será utilizado abaixo foi realizado em um Intel Core i7, sshd de 128 Gb, com 16 Gb de memória RAM.

Foram analisados entradas de tamanhos de entrada iguais (n = linhas inserção e busca)(10 registros inseridos)), variando apenas os números de multiplicador das entradas (n, 2n, 3n, ..., 10n).

n → 0.0023 segundos
2n → 0.0019 segundos
3n → 0.0021 segundos
4n → 0.0024 segundos
[...]

10n → 0.0054 segundos

A partir desses pontos de coordenadas foi gerado o gráfico de regressão quadrática abaixo:



Função linear que melhor expressa a função de tempo de execução do programa:
F(t) = 0.0000392x + 0.001252

Com isso, pode-se concluir que o tempo vai aumentando quase de maneira linear, porém ainda assim linear, ao longo que o tamanho da entrada vai aumentando, o que é facilmente verificado pelo fato da complexidade da função ser O(n).

Conclusão:

A implementação desse trabalho prático foi bem complicada e difícil, tanto que não foi concluída em sua totalidade. Somente as funções de inserção dos registros e a busca (comandos add e search) estão funcionais.

A maior dificuldade foi que a implementação da árvore no arquivo de forma automática e que serve para todos os casos de teste e se adequa para todas as passagens de parâmetro de ordem possível.

A dificuldade maior foi conseguir estruturar o arquivo de forma q a bipartição da raiz não prejudicasse o atual offset, mas não consegui pensar em algum modo em tempo hábil. Logo, o programa está incompleto, mas consegue processar a saída quando se insere um registro e acontece a busca dele (add e search) somente.