Universidade de São Paulo – USP
Escola de Engenharia de São Carlos – EESC
Departamento de Engenharia Elétrica e de Computação – SEL

Trabalho 02 – Árvore B

SCC0603: Algoritmos e Estruturas de Dados II

Profa. Elaine Parros Machado de Sousa

Engenharia de Computação

Alunos:	Estevam Fernandes Arantes	9763105
	Henrique Andrews Prado	9771463
	Osmar Bor Horng Chen	9288359
	Willian Gonzaga Leodegario	9771293

São Carlos
2017

SUMÁRIO

1. O Trabalho

O trabalho tem como objetivo aplicar os conceitos estudados na disciplina de Algoritmos e Estruturas de Dados II para a solução de problemas práticos.

Dessa forma, o grupo desenvolveu um programa em C++ capaz de implementar um TAD para Árvore-B de ordem 5 possuindo a funcionalidade de indexação de dados, com as operações de criação, inserção, busca e exibição da Árvore-B, correspondente ao ponto extra.

2. Estrutura do Arquivo de Índice

O arquivo de índice foi montado com a abstração da Árvore-B, utilizando em seu início uma struct Flag para salvar os dados referentes à árvore, sendo estes a raiz, o número de páginas, o ponteiro para o cálculo do número de uma página quando for escrita e uma flagOk, que será verdadeira quando o arquivo estiver atualizado e falsa caso contrário, utilizada para a checagem se houve alguma corrupção no arquivo, ou seja, alguma função o utilizou mas não foi terminada corretamente.

A função utilizada para a interpretação direta do arquivo de índice é rrnToOffset, que calcula o offset de uma página com rrn passado como parâmetro, assim usado para os fseeks no arquivo de índice nas funções de pagina_escrever e pagina_ler.

3. Estruturas de Dados Utilizadas

Para a implementação do programa, foram utilizados 2 arquivos que trabalham em conjunto, o indices.idx e dados.dat, especificados nos defines do arquivo bTree.hpp.

No arquivo de índices, onde será baseada a b-tree, foram escritos os ids e byte_offset em forma de página, utilizando dados auxiliares para o número de chaves e para os ponteiros para as próximas chaves. Com o RRN da página é possível obter, graças ao seu tamanho fixo, diretamente as informações dela, podendo assim recuperar o byte offset no arquivo de dados referente ao id desejado.

Assim, tendo o byte offset da informação desejada, basta ir diretamente ao arquivo de dados na posição desejada utilizando fseeks e recuperar primeiramente o tamanho da informação e logo em seguida utilizar um buffer que, através da função de parsing, recupera o id, título e gênero da música requisitada, assim a imprimindo.

4. Estratégias de Implementação

Para a implementação foi buscada a maior modularização do código.

Foram criadas as funções *pagina_ler* e *pagina_escrever* para que fosse feita a leitura de uma determinada página (RRN) passada por parâmetro, ou a escrita dessa, guardada no ponteiro para a struct de página.

Foi feita também a função de *página_check*, que ordena uma página determinada de acordo com as suas chaves, utilizada somente na função de split para evitar erros.

As funções de *pagina_insercao* e *pagina_split* serão chamadas pela função principal de inserção, para quando for encontrada a página correta para a inserção de uma chave.

No caso da *arvore_**iniciar***, que, caso o parâmetro build seja falso, somente irá ler a struct flag do arquivo e estabelecerá a base da árvore, isto é, a sua raiz e os seus atributos. Caso o parâmetro seja verdadeiro, chama a função build, que zera toda a árvore e insere, ao passar pelo arquivo de dados todo, elemento a elemento na árvore, utilizando função *arvore_inserir*.

A função de **inserção** toma como parâmetro os elementos que se deseja inserir, primeiramente realiza uma busca para checar se o ID existe na árvore, de modo a evitar duplicatas, e então chama a subfunção *arvore_inserir*. Tal função faz uso de uma simulação de recursão através de um while, começando pela sua raiz e seguindo pelo filho onde estaria a posição ideal do elemento. Quando encontra-se essa posição, avalia se é necessário realizar o split e, em caso positivo, chama a função *pagina_split*, a qual segue o processo de split padrão utilizando um vetor auxiliar para a quebra em nova página e a promoção do id mediano. Então, no final, sabendo o rrn da página correto, a escreve utilizando as funções de arquivo.

Na **busca**, foi utilizada, em conjunto com a função *busca*, a sub função *arvore_busca*, a qual segue um procedimento semelhante à busca do local ideal da inserção e, caso encontre o id retorna a página onde ele está e caso contrário retorna -1. Assim, através desse retorno é possível avaliar se a chave está contida na árvore (se for diferente de -1) e, através do RRN da página, lê-se a informação correspondente e já a imprime diretamente no log.

Na função (extra) *arvore_**imprimir***, é feita a impressão por níveis da Árvore-B. Para isso foi utilizada uma fila estática, programada em forma de vetor que, através dos ponteiros, que são os RRNs das páginas, lê-se diretamente do arquivo e imprime cada uma das páginas, no formato solicitado.

5. Complexidade de Tempo e Espaço

Como as operações em disco são extremamente mais custosas, as complexidades temporais giram em torno destas.

Assim, tanto no caso da busca como da inserção, teremos um máximo da ordem de ceil($\log_{M/2} N$) acessos à disco, sendo sendo M a ordem da árvore, que no caso é 5, e N o número de elementos na árvore como um todo, já que esses elementos, no pior caso, estarão divididos em páginas com M/2 IDs cada uma, assim caracterizando a complexidade de tais operações, visto que em ambos os casos são realizadas apenas operações em RAM dentro de cada uma das páginas, as quais tem complexidade linear porém não são levadas em conta na ordem de tempo geral devido ao maior tempo de acesso das informações de disco, conforme comentado.

No caso da construção da árvore, a comprexidade é O($N * \log_{M/2} N$), visto que será necessário passar pelo arquivo de dados carregando N buffers e inserindo cada um deles, a qual terá complexidade $\log_{M/2} N$, totalizando a complexidade dita anteriormente, visto que as demais operações serão realizadas em RAM dentro de cada uma das páginas.

6. Instruções de compilação e execução

O programa foi testado nos sistemas Linux e Windows, através da bash no segundo, porém a IDE CodeBlocks corrompe os arquivos e portanto não permite a leitura e escrita adequada destes no formato desejado.

Portanto, para contornar tal problema, foi feito um arquivo Makefile que acompanha o código no .zip.

Para compilar o programa de modo que a execução seja limpa e os prints apenas no arquivo de log, utilize: make all

No caso de não funcionar, provavelmente é devido à não instalação do c++0x no computador, então é aconselhado, para rodar, utilizar: make oldC

Caso queira ver os prints utilizados para debug, com diversos dados da árvore e do arquivo, utilize: make debug

Para rodar, basta utilizar: make run

Em caso alternativo, é possível rodar através do comando ./main