Árvores B e armazenamento eficiente em memória secundária

Lucas Roberto Wagner Carneiro e Mariane Batista dos Santos

Instituto de Computação – Universidade Federal Fluminense (UFF)
Caixa Postal 24210-346 – Rio de Janeiro – RJ – Brasil

Departamento de Ciência da Computação
Universidade Federal Fluminense (UFF) – Rio de Janeiro, RJ – Brasil

lucasroberto@id.uff.br | mary.binha.mb@gmail.com

Abstract. *B-trees are included in the tree-based data structure group, which allow the indexing of many records in secondary memory, such as hard drives, HD and SSD, necessary in scenarios where the complete data structure does not fit in main memory. This is a fundamental part of information science that has become even more important as the volume of data grows exponentially. It aims to organize and store data efficiently, being used in numerous and diverse applications, such as databases and file systems in modern operating systems.*

Resumo. *As árvores B fazem parte de um grupo de estrutura de dados baseada em árvores, elas permitem a indexação de uma abundância de registros em memória secundária, como, por exemplo, em discos rígidos, HD e SSD, necessária em cenários onde a estrutura de dados completa não cabe na memória principal. Trata-se de uma parte fundamental da ciência da informação que vem tornando-se ainda mais importante à medida que o volume de dados cresce exponencialmente. Tem por objetivo organizar e armazenar dados de maneira eficiente, sendo utilizadas em inúmeras e variadas aplicações, tais como banco de dados e sistemas de arquivos em sistemas operacionais modernos.*

1. Introdução

Uma das principais estruturas de dados existentes é a baseada em árvores, sendo uma grande área de pesquisa da ciência da computação. A árvore B, foco deste trabalho, é uma vertente amplamente utilizada dessa estrutura, que se tornou uma alternativa potencialmente eficiente no armazenamento de dados. Sabemos que a memória principal tem alta velocidade, mas pouca capacidade, enquanto a memória secundária tem baixa velocidade, mas grande capacidade, dito isso as árvores B surgem para resolver o problema que ocorre quando a estrutura de dados não cabe inteiramente na memória principal. Caracterizam-se por serem perfeitamente balanceadas, onde todas as folhas apresentam-se em um mesmo nível, assim como por permitir a existência de múltiplos elementos por nó. Além desta introdução, apresentamos uma seção breve sobre árvores, seguida pelas propriedades gerais e implementação de uma árvore B,

discutimos os resultados atingidos, bem como fazemos uma análise de complexidade e, finalmente na última parte têm-se a conclusão e abertura para futuros trabalhos.

2. Árvores

Árvores são estruturas de dados caracterizadas por vários atributos, por exemplo, em relação a sua aridade, ou seja, o número máximo de filhos que cada nó pode ter, chamadas árvores M-ária. Outro aspecto importante está relacionado ao seu preenchimento, árvores cheias (ou perfeitas) e árvores completas. Entre as árvores M-árias, as binárias são as mais estudadas por serem mais simples e eficientes quando se trata de buscar e manipular informações em comparação com árvores de maior aridade, mais complexas e que exigem maior esforço computacional. Seu uso é incentivado ainda pelo fato de que árvores de qualquer aridade podem ser convertidas em binárias, o que é um grande benefício [4] em contrapartida, podem apresentar problemas com a falta de regras para inserção e exclusão de elementos, fazendo com que buscas posteriores sejam ineficientes. Para solucionar esses problemas, surgiram as árvores binárias de pesquisa, que adotam um processo interno de ordenação onde os nós à esquerda são menores que o nó raiz, e os à direita, maiores. Além de permitirem buscas mais eficientes, fazem com que o processo de inserção e exclusão de nós seja realizado mantendo essas propriedades [1,10]. Porém, em alguns cenários, como por exemplo a inserção de muitos elementos repetidos, as árvores podem ficar desbalanceadas, fazendo com que o processo perca desempenho, nesse contexto, surgem soluções alternativas, como as árvores rubro negras e árvores B.

3. Árvores B

Criadas por Rudolf Bayer e Edward M. McCreight e apresentadas no artigo intitulado *Organization and Maintenance of Large Ordered Indexes* em 1972 [8], as árvores B se diferenciam de outras árvores por terem múltiplos filhos, com mais de uma chave por nó. Esse aspecto além de permitir caminhos mais curtos que as árvores binárias de pesquisa, por exemplo, acelera o processo, já que se reduz o acesso à memória secundária e a necessidade de organização dos elementos nas operações de inserção e remoção. Útil em situações em que se trabalha com grandes volumes de dados que não cabem em sua totalidade na memória principal, possibilitando o carregamento de cada nó da árvore da memória secundária conforme seja necessário. As características descritas tornam as árvores B uma escolha apropriada para aplicações como bancos de dados, onde estas operações são mais frequentes.

3.1 Propriedades

Uma árvore B de ordem t, onde t é o número mínimo de filhos que um nó pode ter, satisfaz as seguintes propriedades:

i. A raiz é uma folha ou tem no mínimo 2 filhos;
ii. Cada caminho da raiz até qualquer folha tem o mesmo tamanho;
iii. Cada nó, exceto pela raiz e pelas folhas, possui no mínimo t e no máximo 2t filhos;
iv. Cada nó tem pelo menos t-1 e no máximo 2t-1 chaves.

Através da modificação do valor do t, é possível chegar em diferentes configurações de árvores B, variando o número de chaves e filhos por nó, personalizando a estrutura de acordo com o tipo de problema abordado.

3.2Operações

As árvores B, assim como outras árvores, possuem três operações principais, são elas: inserção, exclusão e busca.

Na inserção, busca-se manter as propriedades de funcionamento da árvore B, em especial que todas as folhas tenham a mesma profundidade e os nós tenham um número máximo de chaves (2t-1). Quando esse máximo é atingido, a chave que está na mediana do nó é movida para o nó pai, enquanto os demais são divididos em dois nós de tamanho mínimo (t-1). Essa operação pode ser repetida até a raiz caso seja necessário e, para a criação de uma nova raiz, define-se um critério, como por exemplo, a chave de maior valor. Assim, a altura da árvore aumenta uma unidade, ou seja, o processo de split é o único meio de uma árvore B crescer.

Na exclusão são necessários cuidados adicionais para garantir que o processo não gere uma árvore cuja estrutura viole as propriedades acima definidas, pois diferentemente da inserção que altera apenas os nós folha, no processo de remoção é possível eliminar uma chave de qualquer nó, afetando também a construção de nós intermediários. É preciso manter o número mínimo de chaves em um nó (t-1) e para isso são utilizados procedimentos de redistribuição de chaves entre nós, fusões ou movimentos de predecessores (maior valor à esquerda de um nó) e sucessores (menor valor à direita de um nó) para o nó pai. Apesar dessas possibilidades existirem, essencialmente a maioria dos valores estão concentrados nas folhas das árvores, então casos de exclusões em nós internos são geralmente pouco frequentes.

Por fim, ao realizar a operação de busca por um valor em uma árvore B, executa-se da forma mais otimizada possível. A partir da raiz, verifica-se da esquerda para à direita se cada um dos valores existentes no nó corrente é maior ou menor que o buscado, caso seja maior, a busca passa para o nó filho à esquerda, caso contrário, a busca segue para o próximo valor existente no nó corrente e, sendo maior que todos, a busca passa para o nó à direita. Com isso, reduz-se a quantidade de comparações necessárias para chegar ao resultado desejado [10].

3.3Aplicações práticas

Para dar visibilidade às aplicações práticas que utilizam esse tipo de estrutura de dados, destacamos duas principais, são elas: o sistema hierárquico de arquivos (HFS) desenvolvido pela Apple Inc. e sistemas de gerenciamento de bancos de dados, tendo um foco nos relacionais.

No HFS por exemplo, os registros de arquivo de catálogo, arquivo ativos (frequentemente usados) dentre outros, são organizados como um arquivo de árvore B.

Conforme em [9]:

Os nós que compõem a estrutura sempre possuem um tamanho fixo (expressado em bytes) de 512 bytes, são bem definidos e contém determinados tipos de informação. Existem 4 tipos de nós, são eles:

i. **Nó de cabeçalho.** Primeiro nó da árvore B, contém a informação necessária para encontrar qualquer outro nó na árvore;
ii. **Nós de mapa.** Contém registros de mapa contendo quaisquer dados de alocação (um bitmap que descreve os nós livres na árvore B) que ultrapassa o registro do mapa no nó de cabeçalho;

iii. Os **nós de índice** contêm registros de ponteiro que determinam a estrutura da árvore B;
iv. Os **nós folhas** mantêm registros de dados que contém os dados associados a uma determinada chave. A chave para cada registro de dados deve ser exclusiva.

Exemplo de processo aplicado a arquivos ativos:

i. Uma implementação qualquer altera a temperatura (importância relativa) de um arquivo ativo;
ii. O registro antigo do arquivo ativo deve ser removido;
iii. Um novo arquivo ativo com a nova temperatura deve ser inserido;
iv. Os dados do registro do thread devem ser alterados para conter a nova temperatura [9].

Já em relação aos bancos de dados, em especial os relacionais, árvores B são usadas para indexação. Isso implica na criação de uma estrutura auxiliar para apontar para os registros em uma tabela, que pode ser entendido como uma extensão da coluna índice. A busca será realizada de forma otimizada, por registros que contém o valor, não sendo necessário a leitura linha a linha individualmente. Esses índices são normalmente criados em colunas mais utilizadas em consultas, cabendo ao administrador da rede decidir por sua criação. Outra vantagem diz respeito a diminuição da sobrecarga no sistema quando uma consulta é realizada, generalizado pelo fato que índices podem ser criados para múltiplas colunas [2,3,7].

4. Implementação

A implementação das 3 principais operações foi realizada em C++, tendo como base o notebook disponibilizado pelo professor Marcos Didonet Del Fabro da UFPR que pode ser acessado através deste link. O código original foi alterado para se adequar às individualidades do trabalho proposto, bem como acrescido de comentários para gerar maior clareza a respeito de seu funcionamento. O código modificado pode ser acessado no github dos integrantes marianebsantos/Arvore-B e lucasroberto/Arvore-b.

4.1Análise de complexidade / Tempo de execução

Operações em árvores levam um tempo proporcional à sua altura, nesse caso, por sempre manter o balanceamento e possuir múltiplas chaves por nó, as árvores B possuem uma altura que em média é menor que outras estruturas, organizada a fim de diminuir o caminho da busca, reduzindo por consequência os acessos à memória secundária [10].
O processo de busca acessa $\theta(h) = \theta(\log_t n)$ nós, onde h é a altura e n é o número de chaves, isso porque no pior e no melhor caso ele acessa $O(\log_t n)$,sendo que além disso ele demora o tempo $O(t)$ dentro de cada nó, ou seja, o tempo total gasto é na verdade $O(t.h) = O(\mathrm{t}\,.\log_t n)$, mas como t é um número constante, a complexidade vai acabar sendo proporcional a $O(\log_t n)$ [10]. Essa complexidade difere das árvores binárias de pesquisa citadas anteriormente, que apresentam complexidade de ordem $O(n)$ no pior caso, por exemplo em árvores degeneradas. Note ainda que a montagem da árvore B tem complexidade $O(\mathrm{n}\,.\log_t n)$, isso porque são necessários n processos de inserção.
As operações de inserção e remoção precisam localizar o nó para inserir e remover corretamente a chave, portanto sua complexidade no melhor caso é $O(\log_t n)$, já no pior

caso, onde por exemplo é necessário realizar processo de divisão em todos os nós até a raiz (inserção) ou realizar processos de fusão e redistribuição (remoção), a complexidade será proporcional a $O(\log_t n)$.

4.2 Análise do tempo de execução da implementação

Os testes foram realizados na plataforma binder, que permitiu a utilização de um notebook JupyterLab versão 3.6.6 com 4GB de memória RAM. Para a utilização do kernel C++17, foi importada a estrutura conda-forge e a biblioteca xeus-cling 0.15.1. Para contabilizar o tempo de execução das diferentes operações da árvore B, utilizou-se a biblioteca chrono, com documentação disponível em cplusplus.com.

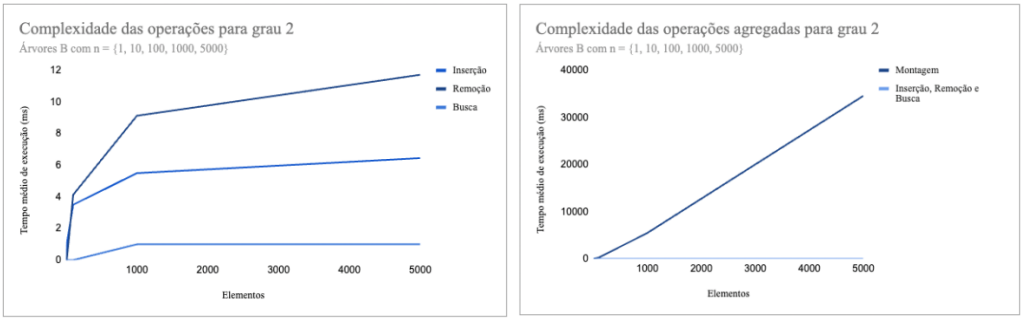
A avaliação dos resultados considerou o tempo médio e desvio padrão entre 100 execuções, variando-se tanto a ordem da árvore B, quanto o número de chaves.

4.3 Resultados

A partir dos resultados observados, nota-se que as curvas do tempo de execução das operações de busca, inserção e remoção possuem comportamentos muito similares no geral, variando pouco a escala dos dados. Quando adicionamos a operação de montagem, a escala dos dados é rapidamente alterada, evidenciando a grande diferença de complexidade entre elas. A escolha do grau para essa visão foi arbitrária, já que os comportamentos eram similares para outros graus.

Análise de Complexidade (Microsegundos)						
Nº Chaves	**Operação**	**Grau 2**	**Grau 3**	**Grau 5**	**Grau 10**	**Grau 100**
n = 1	**Inserção**	0	0	0	0	0
	Remoção	0	0	0	0	0
	Busca	0	0	0	0	0
	Montagem	13.39 ± 2.26	14.66 ± 3.97	14.73 ± 3.39	14.38 ± 4.63	26.77 ± 5.94
n = 10	**Inserção**	1	0.02 ± 0.14	0	0	0
	Remoção	0	0	0	0	0
	Busca	0	0	0	0	0
	Montagem	20.2 ± 9.34	27.7 ± 3.89	19.11 ± 7.74	23.36 ± 4.23	17.66 ± 4.72
n = 100	**Inserção**	3.5 ± 0.87	3.98 ± 1.16	0.11 ± 0.31	0	0
	Remoção	4.13 ± 1.17	4.9 ± 1.9	2.03 ± 1.27	0.76 ± 0.45	2.7 ± 0.46
	Busca	0	0	0	0	0
	Montagem	250.4 ± 12.33	193.08 ± 12.16	98.94 ± 7.02	114 ± 29.98	164.78 ± 115.46
n = 1000	**Inserção**	5.5 ± 0.62	3.31 ± 0.82	2.24 ± 0.90	2.01 ± 0.82	0.02 ± 0.14
	Remoção	9.13 ± 1.74	8.84 ± 3.39	5.84 ± 2.25	4.21 ± 3.33	6.18 ± 2.17
	Busca	1	2	2,38 ± 1,74	0,17 ± 0,40	3,21 ± 1,62
	Montagem	5486.01 ± 2094.7	3421.42 ± 1317.78	2645.53 ± 2513.33	1510.8 ± 229.487	3035.98 ± 3378.64
n = 5000	**Inserção**	6.45 ± 1.39	3.14 ± 0.59	3.16 ± 0.49	2.35 ± 1.70	0.18 ± 0.49
	Remoção	11.72 ± 2.37	7.27 ± 0.85	7.05 ± 2.62	5.44 ± 2.13	7.48 ± 1.69
	Busca	1.01 ± 0.1	2,14 ± 0,59	3,23 ± 1,61	0,79 ± 0,43	2,03 ± 0,17
	Montagem	34568 ± 4937.47	20999.8 ± 5334.26	15593.7 ± 5362.87	11605.7 ± 2230.27	18810.9 ± 5814.73

Tabela1. Tempo médio entre 100 execuções

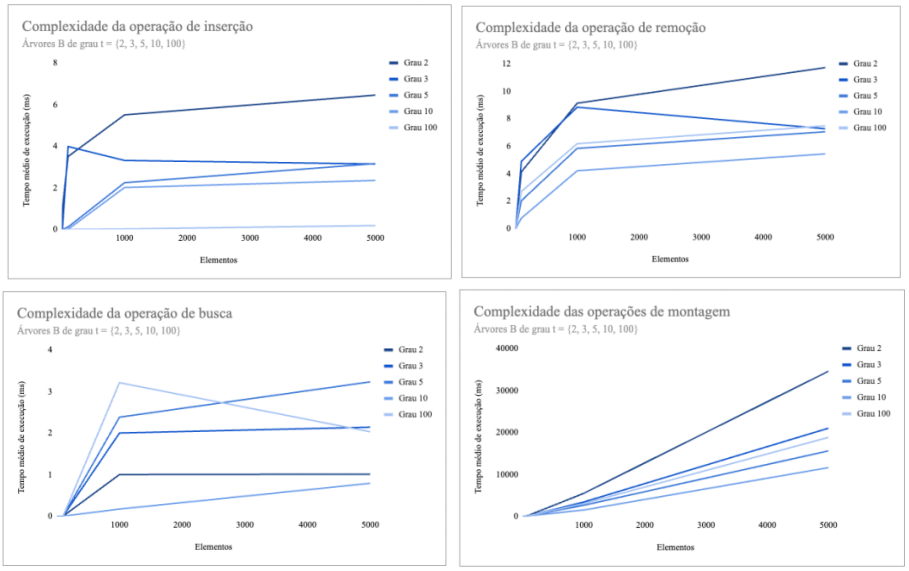


Gráficos [1-2]. Tempo de execução por tipo de operação

Quando recortamos por grau da árvore, exploramos resultados mais variados. Podemos ver que na operação de busca, quando há menos de 1000 elementos na árvore, o grau 100 é o mais lento, seguido pelos graus 5, 3, 2 e 10 nessa ordem, conforme o número de elementos na árvore aumenta, o tempo de execução do grau 100 diminui a ponto de torná-lo bem próximo do desempenho das árvores de grau 3. Na inserção quanto maior o grau da árvore, menor é o tempo de execução, ao passo que o número de chaves aumenta, o grau 3 começa a ter um desempenho melhor, chegando a ser praticamente igual ao tempo de execução do grau 5 para 5000 chaves. A remoção para menos de 1000 elementos na árvore, apresenta um tempo maior de execução para o grau 2, seguido pelos graus 3, 100, 5 e 10 nessa ordem, ao passo que o número de elementos aumenta, o grau 3 começa a ter um desempenho melhor, chegando a ser mais rápido que o grau 100 para 5000 chaves. A montagem por sua vez mostra um comportamento mais linear, mais rápida para graus maiores, com exceção do grau 100 que se mostra mediano.

Breve análise por grau da árvore considerando os diferentes números de elementos:

i. Grau 10 foi o mais eficiente em todas as operações, menos inserção. Logo, para aplicações em que busca e exclusão sejam mais frequentes, essa se mostra uma boa opção;
ii. Grau 2 foi o menos eficiente em todas as operações, não justificando sua utilização;
iii. Grau 100 apresentou resultado mediano de desempenho no geral, sendo mais eficiente na inserção. Pode ser uma boa opção para aplicações com foco nessa operação;
iv. Grau3 foi o segundo pior em desempenho no geral;
v. Grau5foi o que apresentou mais variações em relação ao seu desempenho, considerando as diferentes operações.



Gráficos [2-3-4-5]. Tempo de execução por grau da árvore B

4. Conclusão

Este trabalho apresentou e discutiu características da estrutura de árvores B, que tem por objetivo ser uma alternativa computacionalmente eficiente de armazenamento em memória secundária assim como explorou algumas aplicações práticas. Aplicações interessantes como o caso do sistema de arquivos HFS visto na seção 3.3, utilizam a estrutura de árvores B na prática, evidenciando seu espaço de atuação.
Os resultados sugerem que os tempos de execução das operações de busca, inserção e remoção no geral possuem comportamentos muito similares ao passo que aumentamos o número de elementos na árvore, tendo uma complexidade proporcional a $O(\log_t n)$. Além disso, pelas variações observadas quando consideramos diferentes graus da árvore, deve-se atentar ao definir o grau da árvore que irá se trabalhar, assim como reforça o artigo original. A estrutura de árvores B é uma proposta relevante para um problema já conhecido e que possui potencial para ser explorada em trabalhos futuros, sendo uma possibilidade de tema a evolução para as árvores B+.

Referências

[1] Algorithms in C, Robert Sedgewick, Addison-Wesley, 1992, seção de Árvores B.

[2] Behavior of B-Tree Operator Classes. Disponível em: <https://www.postgresql.org/docs/current/btree-behavior.html>

[3] Bitmap Index vs. B-tree Index: Which and When? Disponível em: <https://www.oracle.com/technical-resources/articles/sharma-indexes.html>.

[4] COELHO, I. M. Árvores. 2023. [Apresentação de Slides]. Universidade Federal Fluminense. Disciplina de Estruturas de Dados e Algoritmos. (link)

[5] FEOFILOFF, P. Estruturas de Dados: Árvores B. Disponível em: https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/B-trees.html

[6] MACHADO LIMA, A. Organização de arquivos Árvores B. Disponível em: <https://edisciplinas.usp.br/pluginfile.php/7739711/mod_resource/content/2/ACH2024-Aula17-ArvoresB-parte1.pdf>

[7] MySQL :: MySQL 8.0 Reference Manual :: 8.3.9 Comparison of B-Tree and Hash Indexes. Disponível em: <https://dev.mysql.com/doc/refman/8.0/en/index-btree-hash.html>.

[8] R. Bayer, E. McCreight: Organization and Maintenance of Large Ordered Indexes - Acta Informatica, Vol. 1, Fasc. 3, 1972, pp. 173-189 (link)

[9] Technical Note TN1150: HFS Plus Volume Format. Disponível em: <https://developer.apple.com/library/archive/technotes/tn/tn1150.html#BTrees>

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Algoritmos: Teoria e Prática. 3a edição. Elsevier, 2012. ISBN 9788535236996