EDA2001 – Comparativo de eficiência de diferentes tipos de árvores

Ayran de Abreu Silva Duarte Kauan Welter Bruggmann Luiz Gustavo Cordeiro
Mônica Caye

21 de junho de 2024

1 Introdução

Neste trabalho, comparamos o custo de inserção de valores aleatorizados em árvores dos diferentes tipos vistos em aula.

Os tipos de árvores sob consideração (Árvores AVL, árvores Rubro-Negras (RB) e árvores B) foram implementadas utilizando-se como referência os códigos providos em aula e pelas referências no final deste documento.

Como estimador de *custo* de um processo, utilizamos a quantidade de comparações que é realizada em sua execução.

2 Implementação da contagem do custo

Para armazenar o custo de uma inserção ou remoção de um nó de uma árvore, uma variável inteira `contadorOperacoes` é passada por referência em todas as funções que operam sobre os nós das árvores consideradas. O valor armazenado na memória relativa a essa variável é incrementado sempre que uma comparação é realizada.

Por exemplo, na implementação típica da remoção em uma árvore binária, substitui-se cada nó pelo seu antecessor imediato, que é calculado pela seguinte função:

```
No* minimo_arvore(Arvore* arvore, No* no){
    while(no->esquerda != arvore->NIL){
        no = no->esquerda;
    }
    return no;
}
```

A seguinte modificação faz com que, na execução desta função, o número de comparação realizadas incremente a variável `contadorOperacoes`:

```
No* minimo_arvore(Arvore* arvore, No* no , int * contadorOperacoes){
    while((*contadorOperacoes)++, no->esquerda != arvore->NIL){
        no = no->esquerda;
    }
    return no;
}
```

Modificações semelhantes são realizadas em todas as condicionais, operadores ternários e laços.

Este modo de realizar a contagem de custo é apropriado tanto para ser implementado em funções aninhadas quanto em funções recursivas, pois a mesma referência de um contador global pode ser passada para subprocessos, e não altera de nenhum modo significativo os argumentos e retornos das funções utilizadas.

Por exemplo, uma implementação comum da remoção em uma árvore é feita por meio de uma função com a seguinte forma:

```
void delecao(ArvoreRB* arvore, NoRB *no){
    ...
    y = minimo_arvoreRB(arvore, no->direita);
    ...
}
```

A versão alterada, que conta com o contador de operações, toma a forma

```
void delecao(ArvoreRB* arvore, NoRB *no , int * contadorOperacoes){
    ...
    y = minimo_arvoreRB(arvore, no->direita, contadorOperacoes);
    ...
}
```

3 Discussão teórica

Todos os processos de inserção e de remoção em árvores AVL e RB e B são realizados por buscas em profundidade com rotações seguidas de balanceamentos a partir das folhas para as raízes.

As rotações tomam tempo constante, logo a inserção e remoção devem ter um custo proporcional à altura da árvore.

Considere uma árvore AVL com n nós e altura h_{AVL}. Da teoria geral[1,2], pode-se deduzir que

$$h_{AVL} \leq \frac{\log_2 n}{\log_2(\varphi)} \approx 1.44 \log_2(n),$$

em que $\varphi = \left(\frac{1+\sqrt{5}}{2}\right)$, e essa desigualdade é assintóticamente ótima. Deste modo, no pior caso (i.e., uma árvore com a maior altura para um dado número de nós), temos que

$$h_{AVL} \approx 1.44 \log_2(n).$$

Similarmente, considerando uma árvore RB com n nós e altura h_{RB}, em [1, Lemma 13.1] se deduz

$$h_{RB} \leq 2 \log_2(n+1) \approx 2 \log_2(n)$$

[1] https://www.wolframalpha.com/input?i=x%280%29%3D1%2C+x%281%29%3D2%2C+x%28n%29+%3D+1%2Bx%28n-1%29%2Bx%28n-2%29

[2] https://stackoverflow.com/questions/30769383/finding-the-minimum-and-maximum-height-in-a-avl-tree-given-a-number-of-nodes

Deste modo, nos piores casos respectivos, a altura de uma árvore AVL com n nós será aproximadamente $1.44/2 \approx .72$ vezes a altura de uma árvore RB com a mesma quantidade de nós.

Isso nos dá uma perspectiva de que árvores RB são "menos balanceadas" (por serem mais altas) do que árvores AVL. Assim, espera-se que a busca em árvores AVL seja mais rápida do que em árvores RB. Por outro lado, deverão ser necessárias menos operações de balanceamento em árvores RB do que em árvores AVL, resultando em processos de inserção e remoção mais baratos.

Por fim, uma árvore B de grau M e altura $h_{B,M}$ com n nós satisfaz

$$h_{B,M} \leq \log_M \left(\frac{n+1}{2} \right).$$

Deste modo, a altura de uma árvore B pode chegar a ser aproximadamente $1.44 \log_2(M)$ vezes menor do que a de uma árvore AVL com o mesmo número de chaves.

Deste modo, espera-se que o número de nós visitados em uma busca em profundidade em uma árvore B diminua com o aumento de sua ordem. Por outro lado, ao visitar cada nó, realiza-se uma busca binária sobre suas chaves, o que tem custo $O(\log_2(M))$. Portanto, não é claro qual estrutura de dados é mais eficiente na busca simples em profundidade. Além disso, os processos de balanceamento em árvores B necessitam de reajustes nos vetores de chaves em cada nó, o que tem custo $O(M)$, o que dificulta a análise teórica comparação teórica dessas estruturas.

Por outro lado, o processo de balanceamento em árvores B só é necessário quando ocorre transbordo de um nó. Se a ordem M é suficientemente grande, esse processo se torna cada vez menos necessário, o que pode diminuir a quantidade de chamadas do processo de balanceamento e consequentemente diminuir o custo final dos processos de inserção e remoção.

4 Criação dos dados

As árvores estão implementadas nos arquivos `arvore_rb.c`, `arvore_avl.c` e `b_tree_clrs.c`. O processo de criação de datos está implementado no arquivo `data_generation.c`.

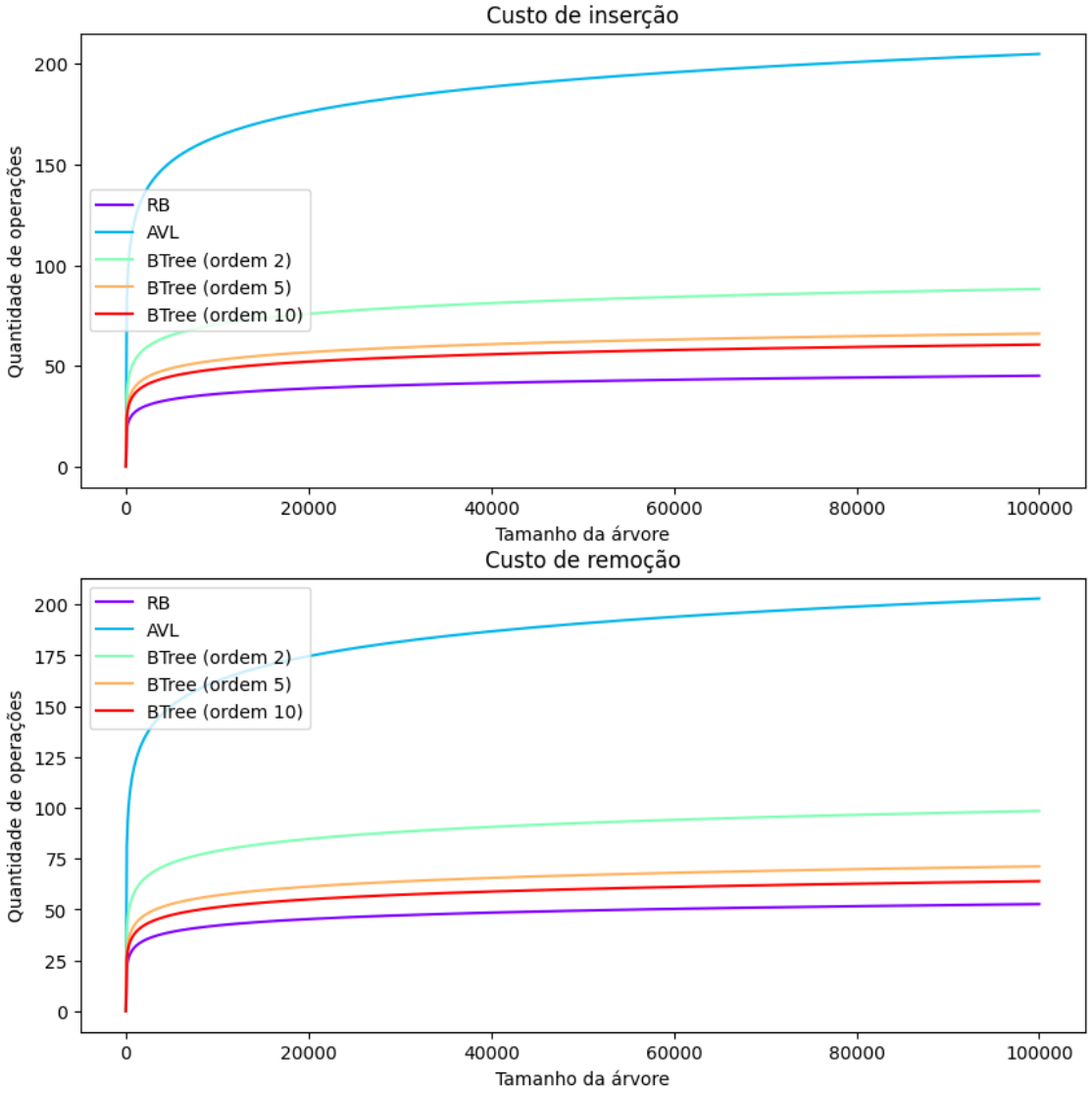
Para cada tipo de árvore sob consideração:

1. Cria-se um vetor `v` com 100.000 entradas aleatórias entre 0 e 99.999 (permitindo repetições)
2. Inserem-se, em sequência, as entradas de `v` como chaves em uma árvore.
3. Em cada inserção, grava-se o par dado pelo tamanho atual da árvore (número de chaves) e o número de operações realizadas nesta inserção.
4. O vetor `v` é aleatorizado pelo algoritmo de Fisher–Yates.
5. Removem-se, sequencialmente, as entradas (agora aleatorizadas) de `v` da árvore.
6. Em cada remoção, grava-se o par dado pelo tamanho atual da árvore e o número de operações realizadas nesta remoção.

Deste modo, obtemos vários pares de números (n, k), em que k é o número de operações realizadas para a inserção ou remoção de uma árvore com n chaves. Para obter um conjunto de dados mais representativo do caso médio, esse processo é repetido 50 vezes. Os dados são gravados em arquivos CSV para análise posterior.

4.1 Análise gráfica

Os dados criados formam nuvens de pontos próximas ao comportamento teórico esperado pelos algoritmos sob consideração. Sabemos que os processos de inserção e remoção em todas as árvores têm comportamento logarítmico. Para prover uma melhor visualização dos dados, os suavizamos com regressões logarítmicas (que nos dão curvas do tipo $\alpha \cdot \log_2(n)$ que melhor aproximam cada nuvem em termos do desvio quadrático).



Vemos, portanto, que, com respeito às inserções e remoções. - inserções e remoções em árvores AVL requerem mais acessos às suas chaves do que nos outros tipos de árvores. - Para graus pequenos, graus maiores determinam árvores B mais eficientes. - árvores RB têm o menor número de acesso às suas chaves (próximo ao das árvores B)

Observações sobre árvores B

Há pequenas variações nas definições de árvores B que podem ser encontradas na literatura. A seguinte tabela compara o número mínimo e máximo de chaves que cada nó de uma árvore B pode conter utilizando diferentes referências.

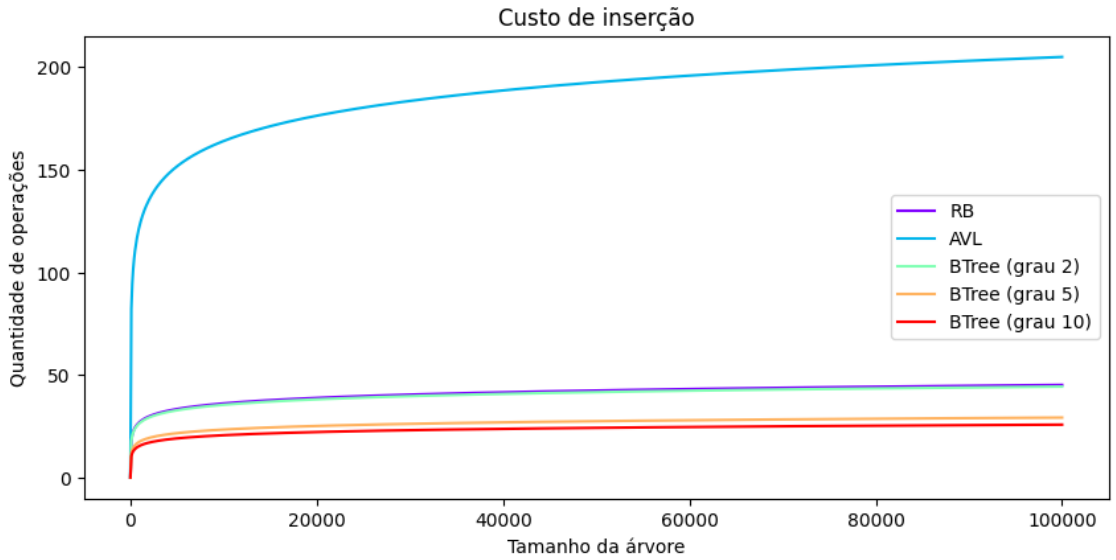
Referência	Mínimo de chaves	Máximo de chaves	Exemplos de (mínimo,máximo) de chaves	Nós folhas respeitam taxa mínima de ocupação?
Vista em aula	M	2M	(1,2), (2,4), (3,6), ...	SIM
[1]	t-1	2t-1	(1,3), (2,5), (3,7), ...	SIM
[2, p. 6.2.4]	$\lceil m/2 \rceil - 1$	m-1	(0,1), (1,2), (1,3), (2,4), (2,5), (3,6), (3,7), ...	NÃO

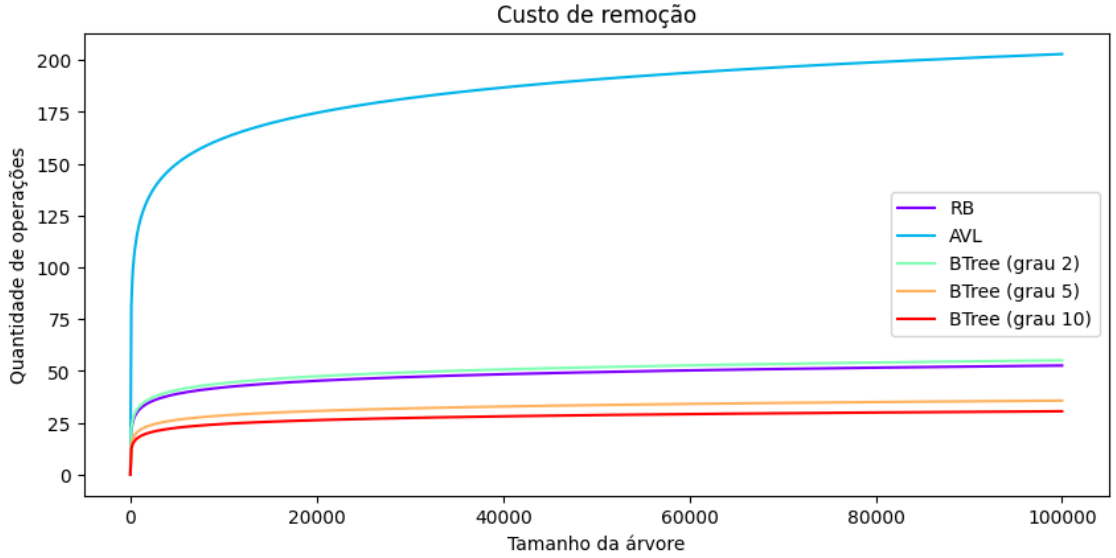
Vê-se, portanto, que os números de chaves permitidos nos nós de árvores B de [2] englobam as outras duas versões, que por sua vez são disjuntas. Essas diferenças fazem com que os algoritmos das duas primeiras referências não sejam imediatamente reutilizáveis nas outras. Porém, em [2], nós folhas podem ter menos nós do que a taxa mínima de ocupação determina, o que facilita a descrição da remoção (que é deixada como exercício), enquanto o caso 3c da remoção de [1] não pode ser adaptado para o tipo de árvore visto em aula.

Portanto, foi necessário decidir um tipo específico de implementação de Árvores B e do processo de remoção, e decidimos por implementar as Árvores B conforme [1], sendo esta a referência mais atual e amplamente utilizada.

Além disso, a literatura deixa claro que árvores B são implementadas comumente em sistemas de registros paginados, em que os acessos às páginas (nós da árvore) são muito mais lentos do que a realização de operações em dados carregados na memória principal.

Portanto, uma comparação mais justa do custo das operações de uma árvore B (com respeito às aplicações práticas) deve considerar somente os acessos aos nós, e não acessos às chaves guardadas em seus nós. Isto é feito com uma pequena modificação na implementação da contagem do custo em árvores B (removendo os incrementos nas buscas dentro dos nós). Nesse caso, obtemos os seguintes dados:





A situação então se inverte: Exceto para grau 2, árvores B requerem menos acessos a nós do que árvores RB (embora as quantidades finais continuem próximas), com melhoras cada vez menorescom o aumento do grau. Porém, nota-se que as diferença de árvores entre grau 2 e 5 é bastante mais significativa do que a diferença entre árvores de graus 5 e 10, indicando que aumentos excessivos de grau não geram aumentos significativos de eficiência.

Referências

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest e C. Stein, *Introduction to Algorithms, fourth edition*, English, Hardcover. The MIT Press, 5 de abr. de 2022, p. 1312, ISBN: 978-0262046305.

[2] D. Knuth, *Art of Computer Programming, The: Sorting and Searching, Volume 3*, English, Hardcover. Addison-Wesley Professional, 24 de abr. de 1998, p. 800, ISBN: 978-0201896855.