# Comparação de implementação de dicionários com árvores vermelho-preto e avl com std::map da linguagem C/C++

**Ítalo L. F. Portinho**[1]

[1]Instituto de Computação – Universidade Federal Fluminense (UFF)
Av. Gal. Milton Tavares de Souza, s/nº – 24.210-346 – Niterói – RJ – Brazil

italoleite@id.uff.br

***Abstract.*** *This paper presents implementations for an abstract data type dictionary based on binary search trees, more specifically the self balancing red-black tree and the AVL tree. We describe insertion, deletion and search operations in both trees and the actions taken to keep them balanced, and in sequence two tests are presented: the first one is insertion/exclusion intensive, and the second is lookup intensive. The conclusion is that the AVL tree has a lower height so is more recommended to lookup intensive applications, and the red-black tree is more efficient for insertions and deletions.*

***Resumo.*** *Este artigo apresenta implementações para um tipo de dados abstrato dicionário baseado em árvores binárias de busca, mais especificamente as árvores auto-balanceáveis vermelho-preto e AVL. São descritas as operações de inserção, exclusão e busca em ambas as árvores e as ações realizadas para mantê-las balanceadas. São apresentados dois tipos de testes, o primeiro intensivo em inserções e exclusões e o segundo intensivo em buscas e concluímos que as árvores AVL possuem altura menor e são mais eficientes para buscas e as árvores vermellho-preto são mais eficientes para inserções/exclusões.*

## 1. Árvores Binárias de Busca

Dicionários são estruturas de dados que suportam operações do tipo chave-valor, como inserção, exclusão e busca. Existem diversas formas de implementá-los e nesse estudo vamos apresentar uma forma baseada em árvores binárias de busca. Esse tipo de árvore possuí a propriedade de que dado um nó r, todos elementos da sua subárvore direita x são menores do que ele e, todos os elementos da sua subárvore esquerda y são maiores do que ele (Figura 1). Uma consequência dessa propriedade é que o percurso em ordem na árvore apresenta os elementos de forma ordenada. Neste artigo e na implementação vamos nos concentrar em árvores que armazenam números inteiros.

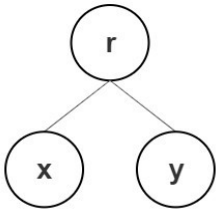


**Figura 1. Árvore binária de busca: subárvore x possuí elementos menores do que r, subárvore y possuí elementos maiores do que r.**

A operação de inserção de um elemento em uma árvore binária de busca começa comparando ele com a raíz da árvore e seguindo o caminho descendente repetindo as comparações até encontrar um ponteiro nulo, que será a posição do novo elemento (Figura 2). Esse também é o procedimento adotado numa operação de busca de um elemento, sendo que se o elemento não for nem menor nem maior que o valor do nó, significa que encontramos o elemento procurado, e se encontrarmos um ponteiro nulo, o elemento não está presente na árvore.

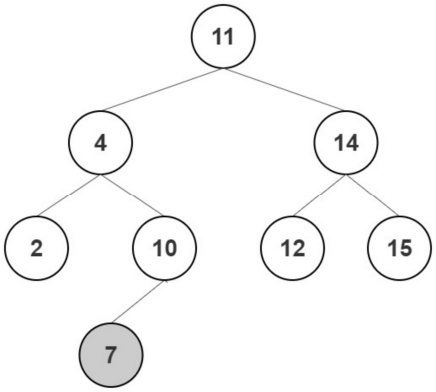


**Figura 2. Inserção do elemento 7 em uma árvore binária de busca com raíz em 11.**

A operação de exclusão começa como uma busca pelo elemento. Se o nó for uma folha basta removê-lo. Se o nó possuir apenas um filho, o nó do elemento é removido e atualizamos os ponteiros para o filho assumir seu lugar. Se o nó possuir dois filhos, ele é substituído pelo seu sucessor na árvore ou seja o menor elemento da sua subárvore direita (Figura 3). As operações de inserção, exclusão e busca possuem complexidade O(h), sendo h a altura da árvore, pois requerem no máximo um percurso

descendente na árvore.

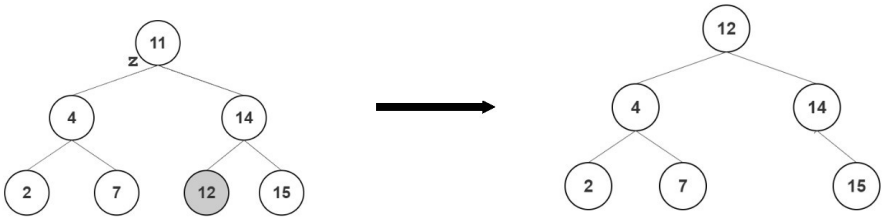


**Figura 3. Exclusão de um elemento com dois filhos em uma ABB.**

Podemos observar que dependendo da ordem que as operações de inserção e exclusão forem realizadas, elas podem levar a um desbalanceamento da árvore. Por exemplo a inserção de uma sequência ordenada de números tornaria a árvore uma lista (Figura 4), e sua altura seria o número de elementos (n) degradando suas operações de inserção, exclusão e busca para a complexidade O(n) enquanto que se a árvore se mantiver balanceada as operações serão realizadas com complexidade O(lg_n) [Cormen 2002]. Nas seções a seguir serão apresentadas duas árvores binárias de busca auto-balanceáveis, árvore vermelho-preto e árvore AVL, que garantem manter a altura da árvore proporcional à lg_n utilizando rotações, coloração dos nós e sua altura, e um teste que compara o desempenho dessas duas árvores.

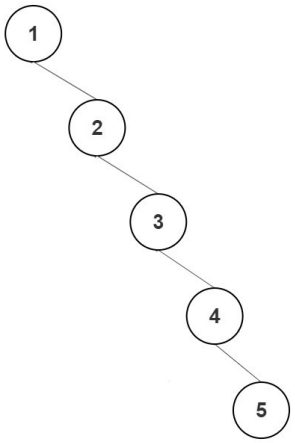


**Figura 4. Uma ABB totalmente desbalanceada, tendendo a uma lista.**

Rotações são operações de deslocamento de nós para esquerda ou direita e reatribuição de seus ponteiros filhos esquerdo e direito, com complexidade O(1) que serão utilizadas tanto nas árvores vermelho-preto quanto nas árvores AVL. As rotações serão guiadas pela coloração dos nós nas árvores vermelho-preto e pela sua altura nas árvores AVL.

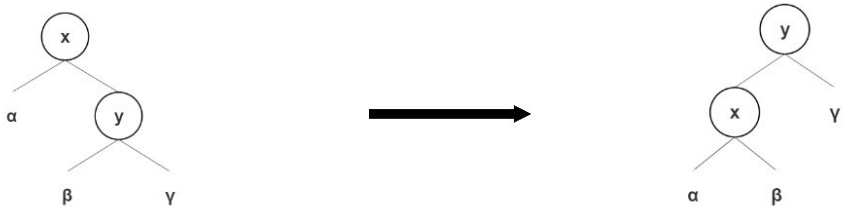


**Figura 5. Rotação a esquerda no nó x. Percurso em ordem permanece inalterado.**

## 2. Árvores vermelho-preto

Árvores vermelho-preto são árvores binárias de busca em que os nós são coloridos com vermelho ou preto e possuem as propriedades listadas a seguir:

- A raíz da árvore é preta;
- Os filhos nulos esquerdo e direito de uma folha são considerados pretos;
- Não podem haver dois nós vermelhos em sequência;
- Todo nó tem o mesmo número de nós pretos em seu caminho até uma folha.

As operações de inserção, busca e exclusão ocorrem como em uma árvore binária de busca comum, porém após inserções ou exclusões são executadas ações para checar se alguma propriedade foi violada e então restaurá-las com rotações e recoloração dos nós.

Em uma inserção o nó é colorido de vermelho e seguimos o caminho descendente a partir da raíz para achar sua posição na árvore. Se ele for a raíz, basta pinta-lo de preto. Se o pai for vermelho existem três casos:

- Caso 1: se o tio também for vermelho, recolorimos o tio o pai e o avô;
- Caso 2: nó, pai e avô não estão na mesma direção (formam um triângulo incompleto). Rotacionamos o pai do nó na direção contrária à ele.
- Caso 3: nó, pai e avô estão na mesma direção (formam uma linha). Colorimos o pai de preto, o avô de vermelho e, rotacionamos o avô na direção contrária ao nó (Figura 6).

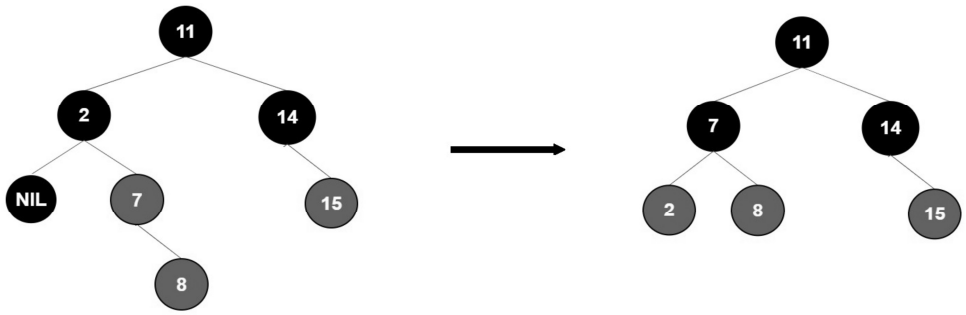


**Figura 6. Reestabelecendo propriedades de uma árvore vermelho-preto após inserção, caso 3 (Rotação a esquerda aplicada no nó 2).**

Na exclusão começamos como na árvore binária de busca, e gravamos uma variável temporária y que será igual ao nó excluído z, ou seu sucessor na árvore no caso de z possuir dois filhos, e também um ponteiro x que será esquerda ou direita de y. O procedimento de correção da árvore após a exclusão possuí 4 casos, que na verdade são 8 pois depende do ponteiro x ser filho da direita ou esquerda de seu pai, e fazem um teste no irmão de x, w:

- Caso 1: irmão vermelho.
- Caso 2: irmão possúi ambos os filhos pretos;
- Caso 3 irmão possui um filho preto.
- Caso 4 irmão tem filho vermelho.

Abaixo são apresentados pseudo-códigos das ações tomadas em cada caso, supondo o ponteiro x ser filho da esquerda [Cormen, 2002].

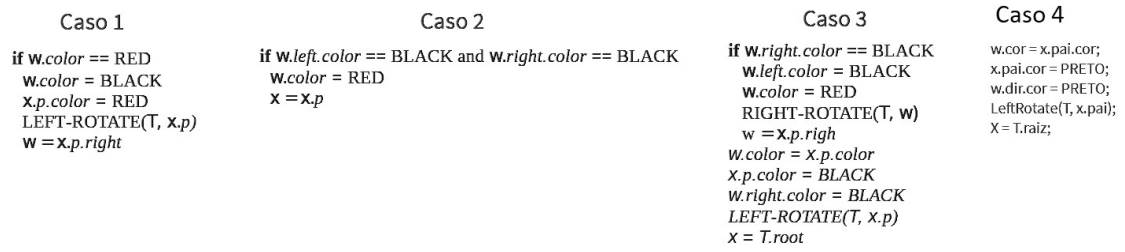


**Figura 7. Pseudo-código de correção pós delete numa árvore vermelho-preto.**

## 3. Árvores AVL

Nas árvores AVL, cada nó vai carregar a sua altura e a altura das subárvores direita e esquerda não pode variar por mais do que 1. Portanto o fator de balanceamento de um nó, definido como a diferença de altura entre as subárvores de um nó, somente pode assumir os valores [-1, 0, +1]. Inserções e exclusões acontecem como em uma árvore binária de busca comum, e é feito um percurso ascendente na árvore atualizando a altura dos nós e checando o fator de balanceamento[Demaine, 2011].

No caso de uma inserção, a verificação consiste em checar qual a subarvore mais pesada e se a chave foi inserida nessa subarvore. Se a chave foi inserida na mesma subárvore será realizada uma rotação simples no sentido contrário da subárvore mais pesada. Caso contrário serão realizadas duas rotações. A primeira será na raíz da subárvore mais pesada no mesmo sentido dessa subárvore e, a segunda será realizada no nó desbalanceado no sentido contrário dessa subárvore (Figura 8).

A exclusão ocorre de forma análoga, fazendo um percurso ascendente a partir da posição do nó excluído e atualizando a altura dos nós. O fator de balanceamento de um nó desbalanceado será comparado com o fato de balanceamento de sua subárvore mais pesada. Se ambos tiverem o mesmo sinal será feita uma rotação simples do nó no sentido contrário da subárvore pesada, caso contrário será feita uma rotação dupla. A primera na subárvore pesada no mesmo sentido dela, e a segunda no nó, em sentido contrário.

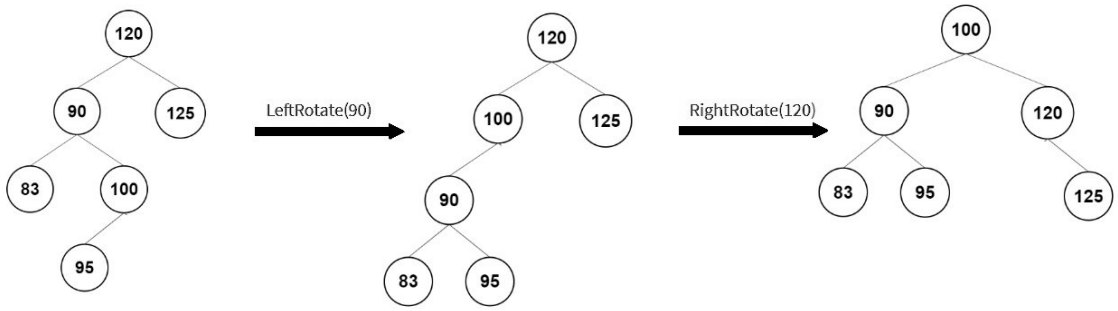


**Figura 8. Rotação dupla em uma árvore AVL – nó 120 desbalanceado.**

## 4. O Teste

O teste realizado consiste em dois conjuntos de operações. No primeiro são realizadas 100000 operações de inclusão de números pseudo-aleatórios, intercaladas por exclusões a cada 100 operações. No segundo conjunto são realizadas 100000 operações de busca

de números pseudo aleatórios. É coletado o número de ciclos de processador gasto por cada conjunto de teste e esse número é convertido para milissegundos usando a constante CLOCKS_PER_SEC da linguagem C/C++ que nos fornece o número de ciclos de processador por segundo. O teste foi aplicado à implementações de árvore vermelho-preto, árvore AVL e também ao map nativo da linguagem. Os resultados obtidos para o conjunto de teste 1 são mostrados na tabela 1. Podemos constatar que a árvore AVL possui altura menor, o que condiz com resultados anteriores que comprovam que o limite superior da altura para uma árvore AVL é 1,44*lg_n , enquanto que para uma árvore vermelhor-preto é 2*lg_n [PFAFF, 2004]. A árvore vermelho-preto obteve melhores resultados levando em conta somente o tempo de execução, o que a torna preferível para aplicações intensivas em inserções e exclusões. No conjunto de testes intensivo em buscas o resultado foi condizente com o fato da altura da árvore AVL ser menor pois seu tempo de execução foi quase 15% menor. Abaixo apresento a tabela com os resultados dos dois conjuntos de teste.

**Tabela 1. Resultados do conjunto de testes 1.**

| | altura | tempo(ms) |
|---|---|---|
| vermelho-preto | 20 | 26 |
| AVL | 18 | 45 |
| std::map | N/A | 53 |

**Tabela 2. Resultados do conjunto de testes 2.**

| | tempo(ms) |
|---|---|
| vermelho-preto | 24 |
| AVL | 21 |
| sdt::map | 49 |

## Referências


Cormen, T., Leiserson, C., Rivest, R., Stein, C. (2002) “Algoritmos, Teoria e Prática”, p.220-231. Elsevier

Demaine, Erik. "AVL Trees, AVL Sort", Recorded at MIT, Fall 2011. https://www.youtube.com/watch?v=FNeL18KsWPc

Pfaff, B., "Performance Analysis of BSTs in System Software," 2004 SIGMETRICS/Performance poster, June 2004