Dicionários Avançados: Árvore AVL e Árvore Rubro-Negra (comparação com std::map)

Rafael Caveari Gomes

Universidade Federal Fluminense
Instituto de Computação
Programa de Pós-Graduação em Computação

Trabalho Final da Disciplina de Estrutura de Dados e Algoritmos
1º semestre de 2021
Professor Igor Machado Coelho

rafaelcaveari@id.uff.br

Resumo. *Esse trabalho visa apresentar dois tipos específicos de Árvores Binárias de Busca Balanceadas: árvore AVL e árvore Rubro-Negra. Além de realizar uma discussão teórica dos temas, foi realizada uma implementação em C++ de um dicionário utilizando as referidas estruturas de dados, sendo os códigos-fontes comentados disponibilizados em [Gomes 2021]. Também foi realizado uma comparação de desempenho entre as implementações propostas e a classe Map da Biblioteca Padrão do C++ [std::map 2021]. Em todos os testes realizados a utilização da biblioteca std::map foi mais eficiente do que as implementações de dicionários propostas, sendo no mínimo* 22% *mais eficiente na inserção de uma* $(chave, valor)$ *em comparação com o dicionário utilizando LLRB e até* 75% *mais rápido, na remoção de uma chave, também em comparação com a mesma estrutura de árvore Rubro-Negra.*

1. Introdução

O Dicionário (*Dictionary*), também conhecido como Mapa (*Map*) ou mapeamento, é um Tipo Abstrato de Dado (TAD) que visa oferecer operações de chave-valor. Servem para armazenar um conjunto de dados de certo tipo (estrutura homogênea), permitindo indexação da chave de busca por tipos arbitrários.

Um dicionário requer três operações básicas:

- Consultar uma chave;
- Adicionar uma $(chave, valor)$; e
- Remover uma chave.

Pode-se implementar um dicionário através de Árvores Binárias de Busca (ABB) ou por Tabelas de Dispersão/Hash [Coelho 2021].

Esse trabalho realizará a primeira abordagem, utilizando dois tipos especiais de ABB: AVL e Rubro-Negra.

1.1. Árvores Binárias de Busca

Podemos definir uma ABB como sendo uma estrutura T, tal que [Szwarcfiter and Markenzon 1994]:

- T possui n nós, com cada nó correspondendo a uma chave distinta $s_i \in S$ e possui como rótulo o valor $r(v) = s_i$;
- Sejam v, v_1 e v_2 nós distintos de T, sendo v_1 pertencente à subárvore esquerda de v, e v_2 à subárvore direita de v, tais que: $r(v_1) < r(v)$ e $r(v_2) > r(v)$.

Podemos utilizar a ABB para resolver o problema de busca, ou seja, determinar se um valor $x \in S$ ou não. Em caso positivo, localizar o índice i tal que $x = s_i$. Para isso, deve-se percorrer o caminho em T desde a sua raiz até s_i. Para determinar esse caminho, o passo inicial consiste em considerar a raiz de T. No passo geral, seja v o nó considerado. Se $x = r(v)$, a busca termina, pois, a chave desejada foi encontrada. Caso contrário, o novo nó a considerar será o filho esquerdo ou o direito de v, conforme respectivamente, $x < r(v)$ ou $x > r(v)$. Caso não exista o nó que deveria ser considerado, a busca termina. Nessa hipótese, S não contém a chave procurada x.

Para determinar a complexidade desse algoritmo, basta observar que, em cada passo, uma nova busca é iniciada recursivamente no filho à esquerda ou à direita. Sendo assim, a complexidade é igual ao número total de chamadas ocorridas no processo. Esse número também é igual ao número de nós existentes no caminho desde a raiz de T até o nó v onde o processo termina. Em um pior caso, v pode se encontrar a uma distância $O(n)$ da raiz de T. Assim sendo, este valor $O(n)$ constitui a complexidade da busca para uma árvore T genérica. O mesmo raciocínio pode ser aplicado as funções de adição e remoção de uma chave na ABB, sendo estes também de complexidade $O(n)$[Szwarcfiter and Markenzon 1994].

Da observação anterior, conclui-se que a complexidade da busca/remoção/adição de uma chave, para uma árvore T, é igual (no pior caso) à sua altura[1]. Portanto, é conveniente tentar uma construção da árvore T, de modo a obtê-la com altura mínima. A árvore que possui essa propriedade, para um conjunto de n chaves, estando dessa forma **perfeitamente balanceada**, é precisamente a completa[2]. Nesse caso, a complexidade dos algoritmos é igual a $O(\log n)$.

Para manter uma ABB perfeitamente balanceada, bastaria remover seus elementos com um percurso em ordem simétrica, inserindo-os em um vetor auxiliar (os elementos ficariam em ordem crescente). Depois reconstruir a árvore reinserindo o elemento central do vetor, e recursivamente reinserir os elementos centrais de cada metade. A complexidade desta operação, porém, é da ordem de $O(n)$ [Falcão 2020]. Então como manter as operações na ABB em $O(\log n)$? A resposta está na ABB balanceada, que resolve o problema de degeneração da árvore pelo controle de sua altura. Esse controle é obtido pelo cálculo de um fator de balanceamento (FB) (ou balanço) para cada nó, definido por [Coelho 2021]:

$$FB = h(esq) - h(dir) \tag{1}$$

onde $h(esq)$ e $h(dir)$ são as alturas dos filhos esquerdo e direito, respectivamente, do referido nó. Caso o filho não exista, então sua altura será zero.

[1] A altura de um nó x é o tamanho do maior caminho que conecta x a uma folha descendente. Denotamos a altura de x por $h(x)$.

[2] Uma árvore m-ária completa é aquela na qual todo nó com alguma subárvore vazia está no último ou penúltimo níveis, estando os nós do último nível completamente preenchidos da esquerda para a direita [Black 2016].

Dois tipos importantes de ABB balanceada são a AVL e a Árvore Rubro-Negra (ou Árvore Vermelho-Preto), objetos de discussão nesse trabalho.

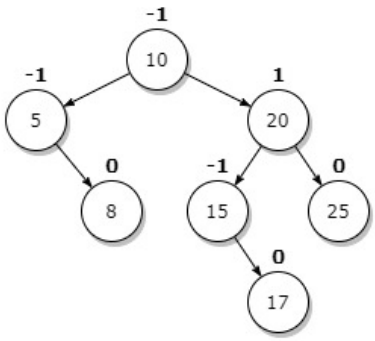
2. Árvore AVL

A ideia de balancear uma árvore de busca se deve inicialmente a [Adelson-Velskii and Landis 1962]. O referido trabalho propõe que, para qualquer nó em uma árvore AVL, as alturas de suas duas subárvores esquerda e direita diferem em módulo de até uma unidade. Dessa forma busca-se manter a AVL como uma árvore binária quase completa, tendo em vista que mantê-la perfeitamente balanceada exigiria uma maior complexidade nas operações.

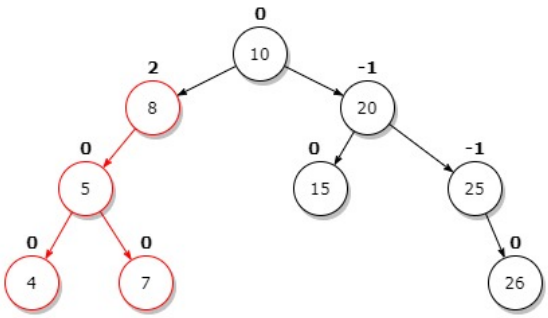
É possível provar que a altura máxima h de uma árvore AVL com n nós é $h = O(\log n)$ e que as operações de inserção, busca e remoção de nós possuem, no pior caso, complexidade $O(\log n)$ [Szwarcfiter and Markenzon 1994].

Um exemplo de uma árvore AVL em equilíbrio pode ser visto na Figura 1(a). Podemos observar que a referida árvore encontra-se balanceada, ou seja, todos os seus nós possuem $|FB| \leq 1$. A inserção de um nó de valor 16, por exemplo, ou a remoção de um nó de valor 8, fará com que a árvore perca o balanceamento (o valor acima de cada nó na Figura 1 é o seu respectivo FB). Nos algoritmos de inserção e remoção de elementos na AVL, os casos de desbalanceamento da árvore serão resolvidos executando-se localmente rotações simples ou duplas nos nós problemáticos, visto que o desequilíbrio é uma propriedade **do nó**. Na Figura 1(b) podemos ver uma árvore AVL contendo uma subárvore em desequilibro (em vermelho), onde o nó 8 possui $FB = 2$, ou seja, encontra-se desregulado.

Figura 1. Exemplos de árvores AVL.



(a) Árvore AVL em equilíbrio.



(b) Árvore AVL em desequilíbrio.

A seguir veremos resumidamente as operações de inserção e remoção de um nó em uma árvore AVL, além das rotações (necessárias para o rebalanceamento da árvore). As demais funções (criar, liberar e buscar) da árvore AVL são idênticas a de uma ABB. Os detalhes das implementações realizadas, com códigos-fontes comentados, podem ser vistos em [Gomes 2021].

2.1. Inserção

O algoritmo para a inserção de um novo nó na árvore AVL pode ser descrito resumidamente em três passos [Backes 2015]:

1. Se a raiz for vazia ou folha, inserir o nó;
2. Se o valor a ser inserido for menor que a raiz, vá para a subárvore esquerda; e
3. Se o valor a ser inserido for maior que a raiz, vá para a subárvore direita.

Deve-se aplicar o método recursivamente e, ao voltar na recursão, recalcular o valor da altura de cada subárvore e aplicar as rotações necessárias se $|FB| \geq 2$.

Conforme já comentado, a função de inserção de um nó em uma árvore AVL, no pior caso, possui complexidade $O(\log n)$ [Szwarcfiter and Markenzon 1994].

2.2. Remoção

Na remoção de um nó em uma árvore AVL podemos ter três situações distintas:

- Nó folha (sem filhos);
- Nó com um filho; e
- Nó com três filhos.

Devemos também tratar o balanceamento a cada remoção da mesma forma que na inserção. Remover um nó da subárvore da direita equivale a inserir um nó na subárvore da esquerda, ou seja, se o nó a ser removido for da subárvore da esquerda, devemos balancear a subárvore da direita, e vice-versa.

A função de remoção de um nó em uma árvore AVL, no pior caso, também possui complexidade $O(\log n)$ [Szwarcfiter and Markenzon 1994].

2.3. Rotações

Como dito anteriormente, o equilíbrio da árvore AVL é corrigido através das chamadas rotações. Existem quatro tipos básicos de rotações:

- Rotação à esquerda (*Right-Right* ou RR);
- Rotação à direita (*Left-Left* ou LL);
- Rotação dupla à esquerda (*Right-Left* ou RL); e
- Rotação dupla à direita (*Left-Right* ou LR).

Considere os exemplos das rotações simples realizadas sobre a raiz (nó em vermelho) das subárvores na Figura 2. Note que $|FB(raiz)| = 2$. Na rotação à esquerda, o nó B torna-se a nova raiz; o nó X vira filho da direita de A e o nó A vira filho da esquerda de B. A operação de rotação à direita é semelhante: o nó B torna-se a nova raiz; o nó X vira filho da esquerda do nó C e o nó A vira filho da esquerda de B. Em ambos os casos uma rotação simples foi suficiente para que o $|FB(raiz)| = 1$.

As rotações simples resolvem o desbalanceamento quando o nó desregulado e seu filho estão no mesmo sentido da inclinação. Isso não irá ocorrer quando o nó desbalanceado e seu filho estão inclinados no sentido inverso ao pai, ou seja, o nó foi inserido em *zigue-zague*. Nesses casos uma rotação dupla será necessária. Na Figura 3 as rotações duplas podem ser visualizadas. Na rotação dupla à esquerda é realizada uma rotação à direita na subárvore da direita e em seguida uma rotação à esquerda na árvore original. Na rotação dupla à direita o mesmo raciocínio pode ser aplicado. Observe que uma rotação dupla pode ser implementada como duas rotações simples.

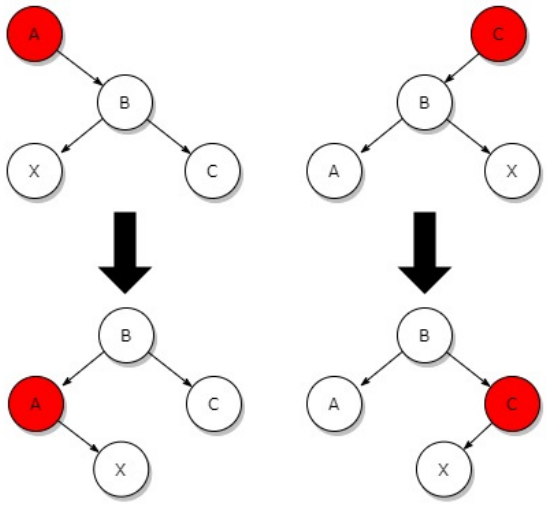


Figura 2. Operações de rotação à esquerda e rotação à direita da árvore AVL.

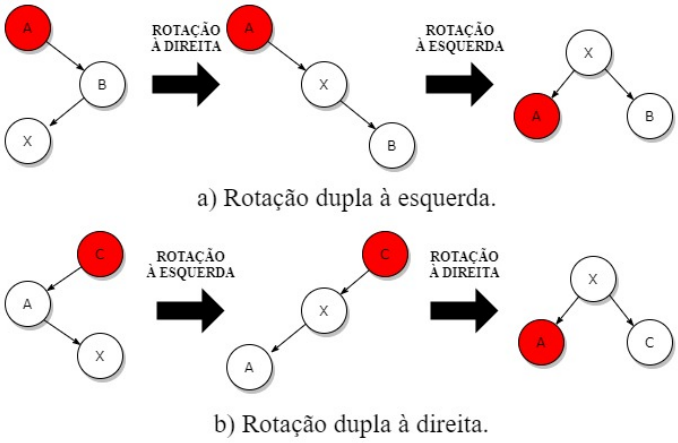


Figura 3. Operações de rotação dupla à esquerda e rotação dupla à direita da árvore AVL.

Uma vez entendido como as rotações funcionam, o próximo passo é saber qual utilizar em cada caso de desbalanceamento do nó. Considerando que o nó C foi inserido como filho do nó B e que B é filho do nó A. Se [Backes 2015]:

- $FB(A) = +2$ e $FB(B) = +1$: aplicar rotação LL em A;
- $FB(A) = -2$ e $FB(B) = -1$: aplicar rotação RR em A;
- $FB(A) = +2$ e $FB(B) = -1$: aplicar rotação LR em A; e
- $FB(A) = -2$ e $FB(B) = +1$: aplicar rotação RL em A.

As rotações são implementadas como uma simples substituição de ponteiros (não dependem da quantidade de nós da árvore), logo possuem complexidade $O(1)$.

3. Árvore Rubro-Negra

Originalmente o conceito de árvore Rubro-Negra foi criado por [Bayer 1972] e chamada de "Árvore Binária Simétrica". Posteriormente, em um trabalho de

[Guibas and Sedgewick 1978], adquiriu o seu nome atual. Uma variação da Árvore Rubro-Negra foi proposta por [Sedgewick 2008], de implementação mais simples e será a abordagem utilizada nesse trabalho.

Na árvore Rubro-Negra é utilizado um esquema de coloração dos nós para manter o balanceamento da árvore. As propriedades da árvore Rubro-Negra são [Backes 2015]:

- Todo nó da árvore é vermelho ou preto;
- A raiz é sempre preta;
- Se um nó é vermelho, então seus filhos são pretos, ou seja, não existem nós vermelhos consecutivos;
- Para cada nó, todos os caminhos desse nó para os nós folhas descendentes contém o mesmo número de nós pretos; e
- Todo nó folha (NULL) é preto[3].

Assim como a árvore AVL, a Rubro-Negra permite o rebalanceamento local, ou seja, apenas a parte afetada pela operação de inserção ou remoção de um nó é balanceada. Para isso, são utilizadas rotações e ajuste de cores na etapa de rebalanceamento. Essas operações buscam corrigir as propriedades da árvore Rubro-Negra que foram violadas. Um exemplo de árvore Rubro-Negra pode ser vista na Figura 4(a).

É possível provar que as operações de inserção, busca e remoção de nós em uma árvore Rubro-Negra possuem, no pior caso, complexidade $O(\log n)$ [Szwarcfiter and Markenzon 1994].

Conforme mencionado anteriormente, nesse trabalho será implementada uma árvore Rubro-Negra Caída Para a Esquerda (*Left-Leaning Red–Black* ou LLRB), que é uma variante da árvore Rubro-Negra proposta por [Sedgewick 2008] que garante a mesma complexidade de operações, mas possui uma implementação mais simples na inserção e remoção de nós. Um exemplo de árvore LLRB pode ser visto em Figura 4(b).

A LLRB atenderá a todas as propriedades da árvore Rubro-Negra convencional, adicionando-se mais uma: se um nó é vermelho, então ele será filho esquerdo do seu pai. Dessa forma caímos na implementação de uma árvore 2-3. Na árvore 2-3, cada nó pode armazenar um ou dois valores e, dependendo da quantidade de valores armazenados, ter dois (um valor) ou três (dois valores) filhos. Dessa forma possui um funcionamento semelhante à árvore binária, agora possuindo também um subárvore do meio, onde estarão os elementos maiores do que o primeiro, mas menores que o segundo valor do nó pai. A representação da árvore LLRB como uma árvore 2-3 pode ser vista na Figura 4(c).

A implementação da LLRB corresponde a implementação de uma árvore 2-3 se considerarmos que o nó vermelho será sempre o valor menor de um nó contendo dois valores e três subárvores. Assim, balancear a árvore rubro-negra equivale a manipular uma árvore 2-3, uma tarefa muito mais simples do que manipular uma árvore AVL ou uma Rubro-Negra convencional.

A seguir veremos resumidamente as operações de rotação e outras funções para o rebalanceamento da LLRB, utilizadas pelas funções de inserção e remoção de um nó. As funções criar, liberar e buscar da LLRB são idênticas a de uma ABB. Os detalhes

[3]Como toda folha possui dois ponteiros NULL, em algumas representações de uma árvore Rubro-Negra essa informação é, para fins didáticos, ignorada.

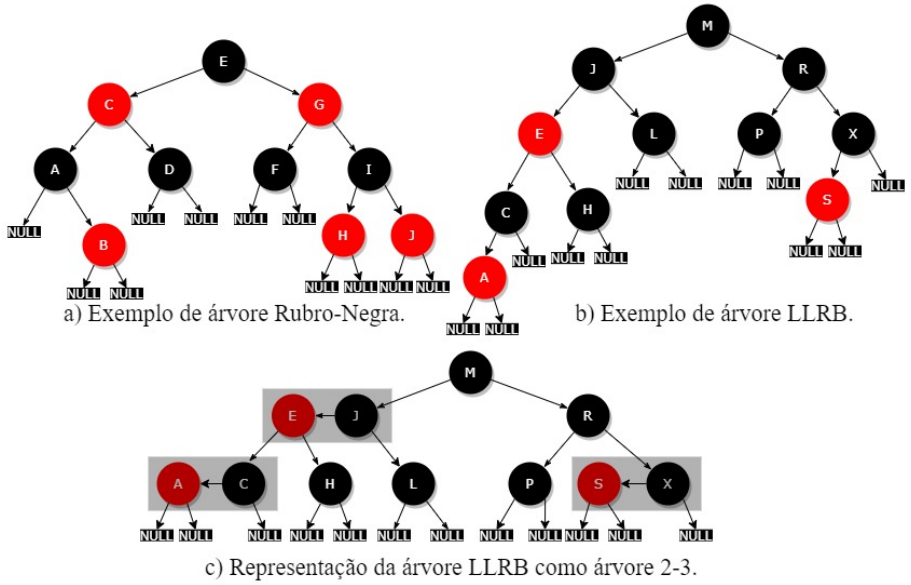


Figura 4. Exemplo de árvores Rubro-Negra, LLRB e 2-3.

das implementações realizadas, com códigos-fontes comentados, podem ser vistos em [Gomes 2021].

3.1. Rotações

Diferente da árvore AVL, temos apenas dois tipos de rotações na LLRB: rotação à esquerda e rotação à direita. Dado um conjunto de três nós, a rotação visa deslocar um nó vermelho que esteja à esquerda para à direita e vice-versa. Assim como na árvore AVL, as rotações são implementadas como uma simples substituição de ponteiros (não dependem da quantidade de nós da árvore), logo possuem complexidade constante.

Considerando um nó A tendo um nó B como filho direito, uma rotação à esquerda consiste em mover o nó B para o lugar de A, tornando A filho esquerdo de B. O nó B então recebe a cor de A, ficando o nó A vermelho (veja Figura 5(a)). A rotação à direita, que pode ser vista na Figura 5(b), é semelhante: considera-se um nó A tendo um nó B como filho esquerdo, movemos o nó B para o lugar de A, tornando A filho direito de B. O nó B então recebe a cor de A, ficando o nó A vermelho. Importante salientar que esses procedimentos apenas rotacionam o nó vermelho, não tratando nenhuma violação das características da LLRB, que devem ser corrigidas por outras operações, como uma mudança de cores, por exemplo.

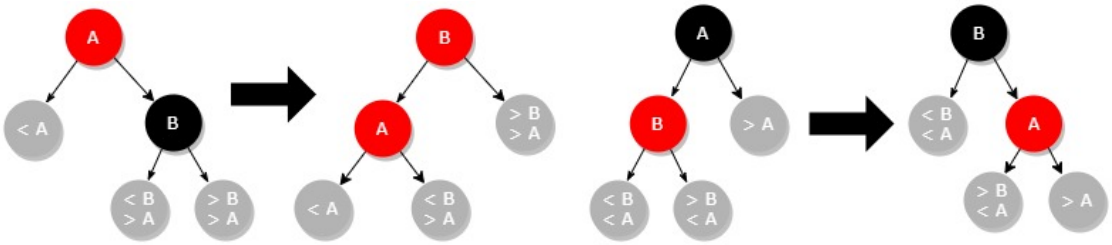
3.2. Outras Funções de Balanceamento

Temos necessidade de outras três funções[4], além das rotações, para restabelecer o balanceamento da LLRB e garantir que suas propriedades sejam respeitadas [Backes 2015]:

- Mover um nó vermelho para a esquerda;
- Mover um nó vermelho para a direita; e
- Arrumar o balanceamento.

[4]Todas as funções citadas realizam apenas troca de ponteiros e mudança de cores, não dependendo do número de nós da árvore, sendo assim, de complexidade $O(1)$.

Figura 5. Rotações na LLRB.



(a) Rotação à esquerda na LLRB.

(b) Rotação à direita na LLRB.

A função que move um nó vermelho para a esquerda recebe um nó A e troca a sua cor e a de seus filhos, conforme pode ser visto na Figura 6(a). Se o filho da esquerda do filho direito é vermelho, então aplica-se uma rotação à direita no filho direito e uma rotação à esquerda no pai e troca-se as cores do nó pai e de seus filhos.

A função que move um nó vermelho para a direita é muito semelhante: ela recebe um nó C e troca a sua cor e a de seus filhos, exemplificado na Figura 6(b). Se o filho da esquerda do filho esquerdo é vermelho, então aplica-se uma rotação à direita no pai e troca as cores do nó pai e de seus filhos.

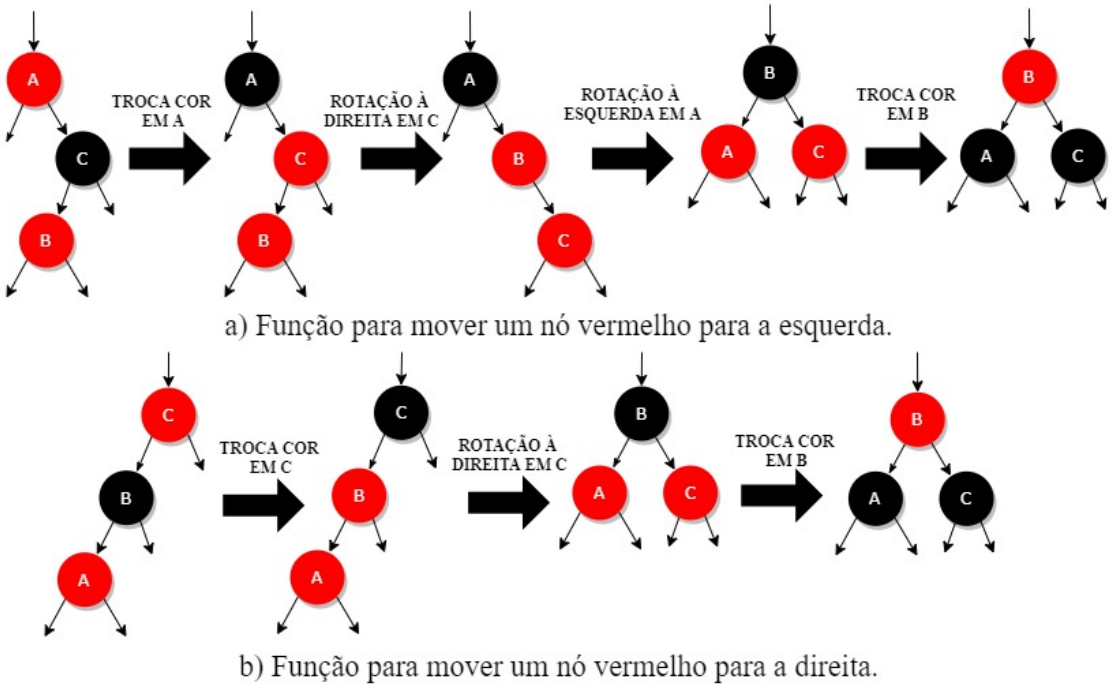


Figura 6. Funções para mover um nó vermelho para a direita e para a esquerda na LLBR.

A função auxiliar usada no rebalanceamento da árvore trata de três situações:

- Se o filho direito é vermelho: aplica uma rotação à esquerda;
- Se o filho esquerdo e o neto da esquerda são vermelhos: realiza uma rotação à direita; e

- Se ambos os filhos são vermelhos: troca a cor do pai e dos filhos.

Dessa forma finalizamos nossa breve discussão sobre a LLRB.

4. Aplicações

Diversos usos das árvores AVL e Rubro-Negra podem ser encontrados na literatura, nas mais diversas áreas de pesquisa. O uso dessas estruturas de dados é justificado quando se deseja armazenar dados com características hierárquicas e quando exista a necessidade de realizar diversas operações de busca, aproveitando-se da complexidade $O(\log N)$ dessa operação nessas estruturas.

Uma aplicação interessante de uma árvore AVL está descrita em [Alakus 2021], onde é proposto um novo método de mapeamento proteico para verificar as interações de proteínas entre a Covid-19 e humanos, utilizando *Deep Learning*. No referido estudo, os vinte aminoácidos existentes na natureza, sendo estes as estruturas básicas que formam as proteínas, são codificados e inseridos em ordem alfabética em uma árvore AVL. Essa árvore então é utilizada pelos diversos métodos de mapeamento e redes neurais descritos no estudo.

Em [Zhang et al. 2020] é proposto um algoritmo para classificação de pacotes IPv6 baseado em uma árvore AVL de *hash*. Já em [Huang 2020] é apresentada uma *blockchain*, classicamente uma estrutura de lista encadeada, a partir de uma árvore AVL. A nova arquitetura descrita, além de manter as características originais de uma *blockchain*, teria agora também buscas mais rápidas na proposta dos autores. Em [Rytter 2003] é utilizado uma árvore AVL como uma estrutura de dados em um algoritmo de compressão de dados.

Em [Liu et al. 2021] é proposto um esquema para verificar a integridade de múltiplas cópias de um arquivo em um serviço em Nuvem baseado em uma árvore Rubro-Negra. No estudo de [Xie et al. 2021], uma árvore Rubro-Negra é utilizada no desenvolvimento e implementação de um interpretador eficiente para robôs industriais, sendo usada para criar uma tabela de símbolos e registrar informações variáveis. Já no trabalho de [Lu and Sahni 2004] é proposta uma estrutura de tabela de roteamento dinâmica em que a busca, inserção ou remoção tem complexidade de tempo $O(\log N)$, utilizando uma árvore Rubro-Negra.

5. Análise de Resultados

Conforme já mencionado, a parte prática desse trabalho consiste em uma comparação das implementações de dicionários realizadas utilizando as estruturas de dados de uma árvore AVL e de uma árvore Rubro-Negra, vistas nas Seções 2 e 3, respectivamente, com a classe Map da Biblioteca Padrão do C++ [std::map 2021]. Foram comparadas as funções de inserção $(chave, valor)$, busca $(chave)$ e remoção $(chave)$.

Foi utilizada a biblioteca *std::chrono* do C++ [std::chrono 2021] para medir o tempo de execução de cada trecho de código de interesse da implementação do dicionário realizada e da *std::map*. Foram utilizados valores de N variando de 500.000 à $15.000.000$. Para diminuir as distorções, as medições foram realizadas 500.000 vezes e a média foi utilizada para plotar os gráficos da eficiência da *std::map* em relação ao dicionário implementado.

A planilha com os dados brutos obtidos e as imagens em alta resolução de todos os gráficos plotados com as comparações entre as estruturas de dados podem ser vistas em [Gomes 2021].

5.1. Inserção

Na função de inserção, os elementos foram inseridos sequencialmente, ou seja, foi inserido o $(0, 0)$, depois o $(1, 1)$, depois o $(2, 2)$... até $(N - 1, N - 1)$. O tempo de inserção de cada $(chave, valor)$ foi calculado até povoar a estrutura com N elementos.

Com os valores obtidos podemos observar que inserir uma $(chave, valor)$ utilizando a biblioteca *std::map* é, na média, 29% e 22% mais rápido do que utilizando a implementação de dicionários com árvore AVL e árvore Rubro-Negra, respectivamente.

5.2. Busca

Após povoar cada estrutura de dados conforme descrito na Seção 5.1, 500.000 elementos foram gerados sequencialmente de 0 até $500.000 - 1$ e buscados na estrutura. O tempo de busca de cada chave foi calculado até encontrar todos os 500.000 elementos na estrutura. Os resultados comparativos podem ser vistos nas Figuras 7 e 8.

Podemos observar que buscar uma chave utilizando a biblioteca *std::map* é, na média, 45% e 34% mais rápido do que utilizando a implementação de dicionários com árvore AVL e árvore Rubro-Negra, respectivamente.

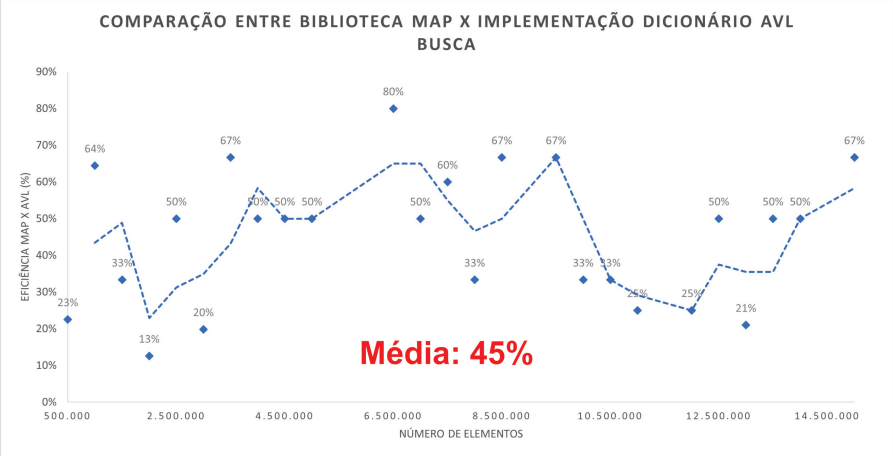


Figura 7. Comparação entre a biblioteca Map com implementação de Dicionário AVL – Busca $(chave)$.

5.3. Remoção

Após povoar cada estrutura de dados conforme descrito na Seção 5.1, 500.000 elementos foram gerados sequencialmente de 0 até $500.000 - 1$ e removidos na estrutura. O tempo de remoção de cada chave foi calculado até remover todos os 500.000 elementos do dicionário.

Analisando os dados obtidos, podemos observar que remover uma chave utilizando a biblioteca *std::map* é, na média, 34% e 75% mais rápido do que utilizando a implementação de dicionários com árvore AVL e árvore Rubro-Negra, respectivamente.

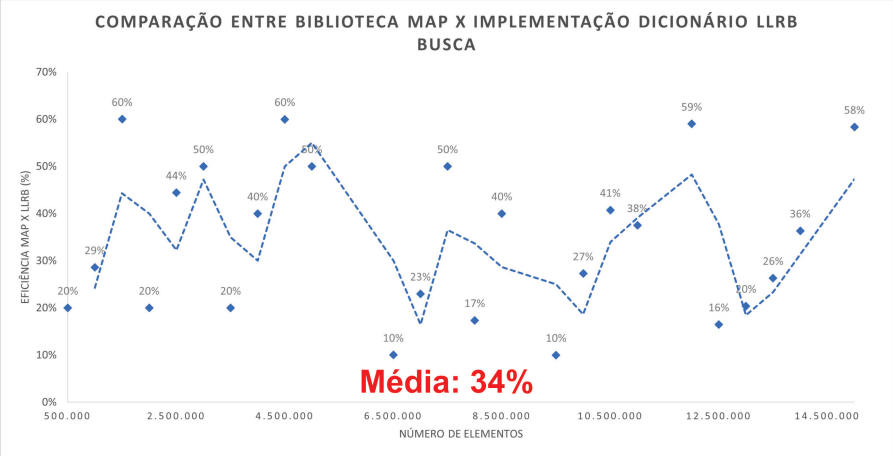


Figura 8. Comparação entre a biblioteca Map com implementação de Dicionário LLRB – Busca $(chave)$.

6. Conclusões

Em todos os testes realizados a utilização da biblioteca *std::map* foi mais eficiente do que as implementações de dicionários propostas, sendo no mínimo 22% mais eficiente na inserção de uma $(chave, valor)$ em comparação com o dicionário utilizando LLRB e até 75% mais rápido, na remoção de uma chave, também em comparação com a mesma estrutura de árvore Rubro-Negra.

Apesar da *std::map* ser implementada utilizando os conceitos de uma árvore Rubro-Negra, conforme descrito na documentação, fica evidente que a utilização de estruturas de dados nativas da linguagem de programação utilizada, além de reduzir possíveis problemas de lógica de programação inseridas pelo usuário, são consideravelmente mais eficientes, pois também utilizam técnicas de otimização[5] próprias.

Referências

Adelson-Velskii, G. and Landis, E. (1962). An algorithm for organization of information. *Proceedings of the USSR Academy of Sciences*, 146:263–266.

Alakus, T.B.; Turkoglu, I. (2021). A novel protein mapping method for predicting the protein interactions in covid-19 disease by deep learning. *Interdiscip Sci Comput Life Sci*, 13:44–60.

Backes, A. R. (2015). Programação descomplicada. `www.facom.ufu.br/~backes/`.

Bayer, R. (1972). Symmetric binary b-trees: data structure and maintenance algorithms.

Black, P. E. (2016). Dictionary of algorithms and data structures - complete binary tree. `https://xlinux.nist.gov/dads/HTML/completeBinaryTree.html`.

Coelho, I. M. (2021). Curso de estrutura de dados e algoritmos. `https://igormcoelho.github.io/curso-estruturas-de-dados-i/`.

[5]Ao compilar os códigos desse trabalho foi utilizado a *flag -O3* no compilador [GCC 2021].

Falcão, A. X. (2020). Curso de estrutura de dados. https://www.ic.unicamp.br/~afalcao/mc202/index.html.

GCC (2021). Gcc - options that control optimization. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

Gomes, R. C. (2021). Dicionários avançados: Avl e Árvore rubro-negra (comparação com std::map). https://github.com/rafaelcaveari/dicionarios-avancados-arvore-AVL-arvore-Rubro-Negra.

Guibas, L. and Sedgewick, R. (1978). A dichromatic framework for balanced trees.

Huang, T.; Huang, J. (2020). An efficient data structure for distributed ledger in blockchain systems. In *2020 International Computer Symposium (ICS)*, pages 175–178.

Liu, Z., Liu, Y., Yang, X., and Li, X. (2021). Integrity auditing for multi-copy in cloud storage based on red-black tree. *IEEE Access*, 9:75117–75131.

Lu, H. and Sahni, S. (2004). O(log n) dynamic router-tables for prefixes and ranges. *IEEE Transactions on Computers*, 53(10):1217–1230.

Rytter, W. (2003). Application of lempel–ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1):211–222.

Sedgewick, R. (2008). Left-leaning red-black trees. https://www.cs.princeton.edu/~rs/talks/LLRB/LLRB.pdf.

std::chrono (2021). C++ reference - std::chrono. https://en.cppreference.com/w/cpp/chrono.

std::map (2021). C++ reference - std::map. https://en.cppreference.com/w/cpp/container/map.

Szwarcfiter, J. and Markenzon, L. (1994). *Estruturas de Dados e Seus Algoritmos*. Livros Técnicos e Científicos.

Xie, M., Li, D., Cheng, M., Li, S., and Luo, Y. (2021). Design and implementation of an efficient program interpreter for industrial robot. *Journal of Physics: Conference Series*, 1884(1):012018.

Zhang, Y.-Y., Chen, X.-X., and Zhang, X. (2020). Pcha: A fast packet classification algorithm for ipv6 based on hash and avl tree. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, pages 397–404.