UNIVERSIDADE ESTADUAL DA PARAÍBA

BACHARELADO EM CIÊNCIAS DA COMPUTAÇÃO

ERIC NATAN BATISTA TORRES

JASMINE MARIA DE ALMEIDA RIBEIRO

SONALLY CECÍLIA DANTAS SALES

THALES HENRIQUE DE SOUZA TELES

YASMIN PEREIRA TRAVASSOS RAMOS

RELATÓRIO PROJETO DE ESTRUTURA DE DADOS SOBRE ÁRVORES BINÁRIAS DE BUSCA

CAMPINA GRANDE

2024

SUMÁRIO

INTRODUÇÃO

As árvores são um tipo abstrato de dados que podem ser implementados em qualquer linguagem de programação. Nela, os elementos estão dispostos de forma hierárquica, e essa hierarquia varia de acordo com o tipo de árvore que está sendo implementada. Essa é a grande diferença de uma árvore para outras estruturas de dados comuns (como arranjos, listas, filas e pilhas) pois as árvores não são organizadas em uma sequência. Em uma árvore não podemos determinar o próximo elemento e o elemento anterior, porque as suas informações não se organizam de maneira enfileirada e sim hierárquica.

Uma árvore é formada por um conjunto de elementos que armazenam informações, chamados nodos ou nós. Toda a árvore possui o elemento chamado raiz, que possui ligações para outros elementos denominados ramos ou filhos. Estes ramos podem estar ligados a outros elementos que também podem possuir outros ramos. O elemento que não possui ramos é conhecido como nó folha, nó terminal ou nó externo. Além disso, toda árvore terá uma altura, que é o caminho mais longo da raiz até um nó folha.

A árvore também é considerada um tipo especial de grafo, pois nela há os nós (vértices) e arestas (conexões entre nós). Porém, a árvore possui algumas características especiais como **aciclicidade** (onde não é possível retornar a um nó após seguir uma sequência de arestas partindo de qualquer nó), e **conectividade** (todos os nós estão conectados de alguma forma, onde a partir de um nó inicial garantimos a conexão entre todos os nós).

Uma das operações mais importantes consiste em percorrer cada elemento da árvore uma única vez. Esse percurso, também chamado de travessia da árvore, pode ser feito em **pré-ordem** (onde os filhos de um nó são processados após o nó) ou em **pós-ordem** (onde os filhos são processados antes do nó). Em árvores binárias é possível ainda fazer uma travessia **em ordem**, em que se processa o filho à esquerda, o nó, e, finalmente, o filho à direita. Essas operações e a estrutura hierárquica tornam as árvores especialmente úteis em problemas de busca, ordenação, e armazenamento de dados em relações parentais complexas.

Nestas operações de busca, a altura está intrinsecamente ligada a eficiência das operações de busca, inserção e remoção de nós, pois determina o número de passos necessários (comparações) para percorrer a árvore. Uma árvore mais equilibrada tende a ter sua altura mais próxima a O(log n), enquanto em uma árvore desbalanceada essa altura estará mais próximo de O(n), ou seja, praticamente uma estrutura de dados linear.

METODOLOGIA

Esse relatório explora o comportamento de quatro tipos de árvores binárias - Árvore Binária de Busca (BST), Árvore Rubro-Negra, Árvore Splay e Árvore AVL - sob diferentes condições de organização de dados, com o objetivo de avaliar o desempenho em termos de tempo de execução, número de rotações e altura final.

As bases de dados utilizadas consistem em três configurações distintas para números entre 1 e 3 milhões de elementos:

- Dados completamente desordenados.
- Dados com os últimos 50% ordenados.
- Dados totalmente ordenados.

A análise foi dividida em duas etapas principais:

1. **Inserção de Elementos:** Medimos o tempo de execução necessário para inserir os elementos nas diferentes árvores. A quantidade de rotações realizadas, especialmente em árvores balanceadas como a Rubro-Negra e a AVL, foi registrada para entender os mecanismos de auto-ajuste de cada estrutura. A altura final de cada árvore também foi calculada, pois impacta diretamente a eficiência da busca.
2. **Busca de Elementos:** Realizamos a busca de 1 milhão de elementos em uma árvore contendo 3 milhões de elementos, medindo o tempo de execução para verificar o desempenho de cada tipo de árvore em uma estrutura de grande escala.

Os testes foram realizados em um notebook Samsung com as seguintes especificações:

- **Processador**: 11th Gen Intel(R) Core(TM) i3-1115G4 @ 3.00GHz
- **Memória**: 12 GB DDR4 (8 GB + 4 GB) a 2666 MHz
- **Armazenamento**: SSD de 256 GB
- **Gráficos**: Intel(R) UHD Graphics com DirectX 12

- **Sistema Operacional**: Windows 11, 64 bits

O repositório do código utilizado para as análises se encontra na parte de referências, no final do relatório. Para o perfilamento do algoritmo, utilizamos o IntelliJ IDE, que fornece ferramentas precisas de análise de desempenho de funções, facilitando a identificação de gargalos e a avaliação detalhada de cada operação nas árvores.

ÁRVORE BINÁRIA DE BUSCA

- **Conceito**

Uma Árvore Binária de busca possui a seguinte estrutura: cada nó, a partir da raiz, vai possuir no máximo dois nós filhos, um filho à esquerda (FE) e um filho à direita (FD), além das outras propriedade de árvores (grafo acíclico, conexo e não dirigido). Além disso, para garantir a eficiência de uma busca nessa árvore, ela é definida de maneira recursiva. Para cada nó da árvore, todos os nós da subárvore à esquerda serão menores que o próprio nó, analogamente, os nós da subárvore à direita serão maiores do que o nó . Dessa forma, se a inserções forem feitas de maneira aleatória, teremos uma árvore balanceada (todos os nós folha estão em níveis semelhantes). Essa estrutura é muito importante para as ciências da computação, pois auxilia na realização de buscas, inserções, ordenações e operações de busca em intervalo (onde podemos encontrar todos os elementos cujo valores estão dentro de um intervalo específico) eficientes, devido à propriedade de ordenação dos nós.

Em uma árvore binária de busca (BST, do inglês *Binary Search Tree*), não é comum permitir valores repetidos, mas podem haver versões com pequenas adaptações, pois a estrutura é projetada para garantir que cada valor ou chave seja único. Isso facilita operações de busca, inserção e remoção, mantendo a ordem de forma eficiente e simplificando o percurso da árvore.

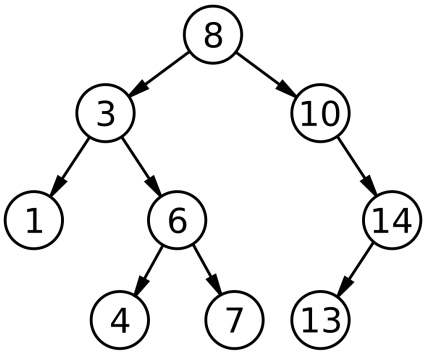


Figura 1 - Exemplo de uma Árvore Binária de Busca. Fonte (https://pt.wikipedia.org/wiki/%C3%81rvore_(estrutura_de_dados), 2024)

- **Estrutura do Nó**

Cada nó contém um valor e pode ter até dois nós filhos, ou seja, poderá ter dois ponteiros para outros dois nós que podem ser denominados filho esquerdo (representando o lado esquerdo do nó) e filho direito (representando o lado direito do nó).

O nó raiz é o ponto de partida da árvore, e todos os outros nós são acessíveis a partir dele.

```
public class No {
    private int valor;
    No esquerda, direita;
```

Figura 2 - Exemplo da estrutura de um nó implementado em Java. Fonte: código utilizado para este relatório, 2024.

- **Funcionamento**

Por ser mais fácil de entender e garantir uma estrutura mais concisa, as árvores são implementadas de **forma recursiva**, fazendo comparações entre os nós já existentes. Porém, o uso dessa implementação apresenta o risco de gerar uma sobrecarga de chamadas (*Stackoverflow,* ou “estouro” de pilha), dependendo dos valores-chave dos nós que serão inseridos, a cada novo nó inserido, a quantidade de comparações será aumentada.

Na inserção da Árvore Binária, o valor que será acrescentado é comparado inicialmente com a raiz, se o valor for menor será feito uma comparação com a sub-árvore à esquerda, se o valor for maior será feita com uma comparação com a sub-árvore à direita, e assim sucessivamente até achar a posição correta do elemento, como podemos observar na figura 3.

```java
private No inserirRecursivo(No atual, int valor) {
    if (atual == null) {
        return new No(valor);
    }

    if (valor < atual.getValor()) {
        atual.esquerda = inserirRecursivo(atual.esquerda, valor);
    } else if (valor > atual.getValor()) { // Impede a inserção de duplicatas
        atual.direita = inserirRecursivo(atual.direita, valor);
    }
    return atual;
}
```

Figura 3 - Exemplo da estrutura de um nó. Fonte: código utilizado para este relatório, 2024.

O método de busca segue a mesma lógica, fazendo comparações recursivas entre os nós esquerda e direita. Já para a remoção, há uma implementação e lógica mais complexa, onde existem três cenários mais complexos, onde o nó a ser removido é uma folha (caso mais simples, pois não há nós esquerda e direta), quando o nó a ser removido tem um filho (terá que haver uma alteração na estrutura, onde o pai do nó a ser removido será conectado diretamente ao filho do nó que está sendo removido), e um terceiro caso mais complexo, onde o nó a ser removido tem dois filhos (nesse caso, encontra-se um substituto para esse nó, pois precisa haver um remanejamento e um lugar dos dois filhos)

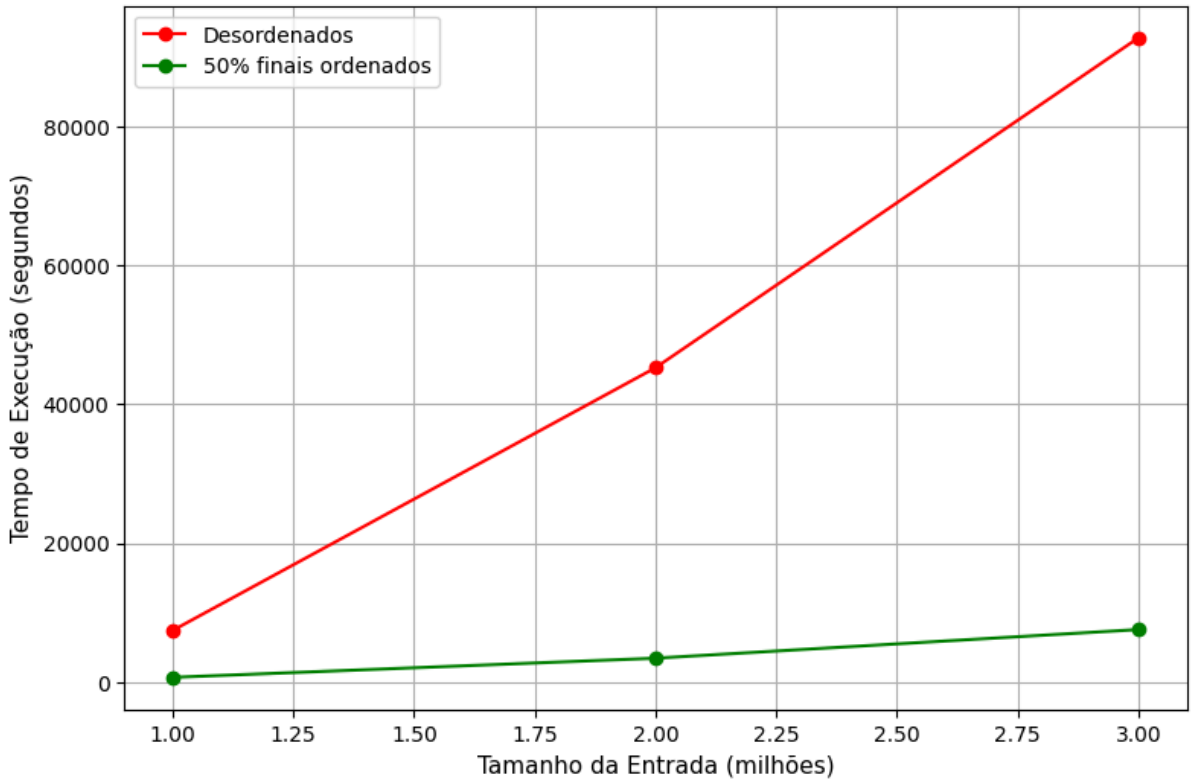
- **Análise**

Quadro 1 - Tempo de execução em milissegundos da inserção de elementos na Árvore Binária de Busca.

Tamanho da Entrada	Desordenados	Ordenados	50% finais ordenados
1 milhão	7415	stackoverflow	672
2 milhões	45274	stackoverflow	3432
3 milhões	92806	stackoverflow	7560

Fonte: Elaboração própria com os dados analisados. (2024)

Gráfico 1 - Tempo de execução em milissegundos da inserção de elementos na Árvore Binária.



Fonte: Elaboração própria com os dados analisados. (2024)

Pelo quadro 1, podemos verificar que a inserção dos dados ordenados na estrutura ocasionou um estouro da pilha de chamadas, pela característica recursiva da implementação do método de inserção. Quando o método de inserção é chamado, o valor a ser inserido é comparado ao valor do nó atual (na primeira chamada será a raiz da árvore), caso o valor seja menor, o método de inserção é chamado novamente para a subárvore à esquerda do nó atual. Caso o valor seja maior, o mesmo acontece para a subárvore à direita do nó atual, as chamadas continuam até que o nó atual seja NIL, e o nó atual será atribuído com o novo valor.

Portanto, quando inserimos os nós de modo crescente, todos os nós serão inseridos como filho direito do último nó da árvore (nó de nível mais baixo), ocasionando numa estrutura de altura n (quantidade de nós inseridos), e o custo de cada inserção passa a ser O(n). Como o tamanho máximo da pilha utilizada nos testes é de 22798, por volta da 22798ª inserção o limite é atingido(utilizamos a

expressão “por volta” porque não conseguimos garantir que todo o espaço da pilha de chamadas foi utilizado para as inserções).

Analisando o gráfico 1, nas bases com dados desordenados, percebemos que o tempo de execução cresce de maneira praticamente linear, o que foge do esperado numa árvores com inserções aleatórias. O tempo esperado para a inserção deve estar entre O(n) numa árvore totalmente desbalanceada e O(log n) numa árvore perfeitamente balanceada. Mesmo que todas as inserções tenham sido feitas sem nenhum erro(*Stackoverflow)* e seja esperado que o tempo de execução aumente à medida que o tamanho da entrada aumenta, o valor selecionado para ser a raiz da árvore provavelmente não foi o melhor possível, ocasionando um gráfico um gráfico mais próximo de O(n).

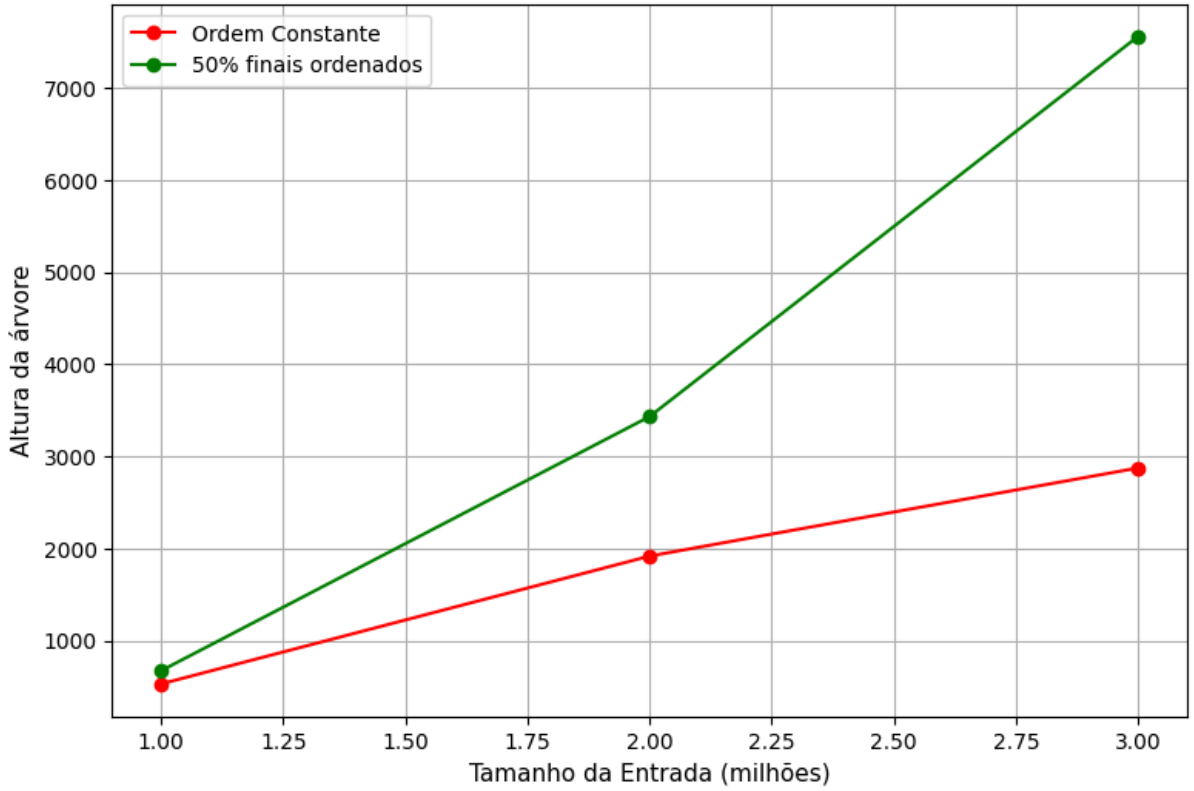
Por outro lado, a base com metade dos dados ordenados apresentou o melhor desempenho. Como temos a primeira metade dos nós inseridos balanceados, o nó escolhido para minimizar a altura da árvore foi eficiente, de modo que as inserções ordenadas não desbalancearam a árvore o suficiente para que aumentasse tanto o tempo de execução, por mais que o tempo de execução também aumenta proporcionalmente a entrada.

Quadro 2 - Altura da Árvore Binária de acordo com a entrada.

Tamanho da Entrada	Desordenados	Ordenados	50% finais ordenados
1 milhão	529	stackoverflow	328
2 milhões	1917	stackoverflow	1100
3 milhões	2874	stackoverflow	1626

Fonte: Elaboração própria com os dados analisados. (2024)

Gráfico 2 - Altura da Árvore Binária de Busca de acordo com a entrada.



Fonte: Elaboração própria com os dados analisados. (2024)

Pela tabela 2, na base com os dados ordenados, a altura antes do estouro da pilha acontecer, foi algo por volta 22798 níveis, por conta do tamanho máximo de chamadas ser 22798. Esse valor é esperado no pior caso, com sua altura sendo igual ao número de nós inseridos, resultado na base que obteve o pior resultado.

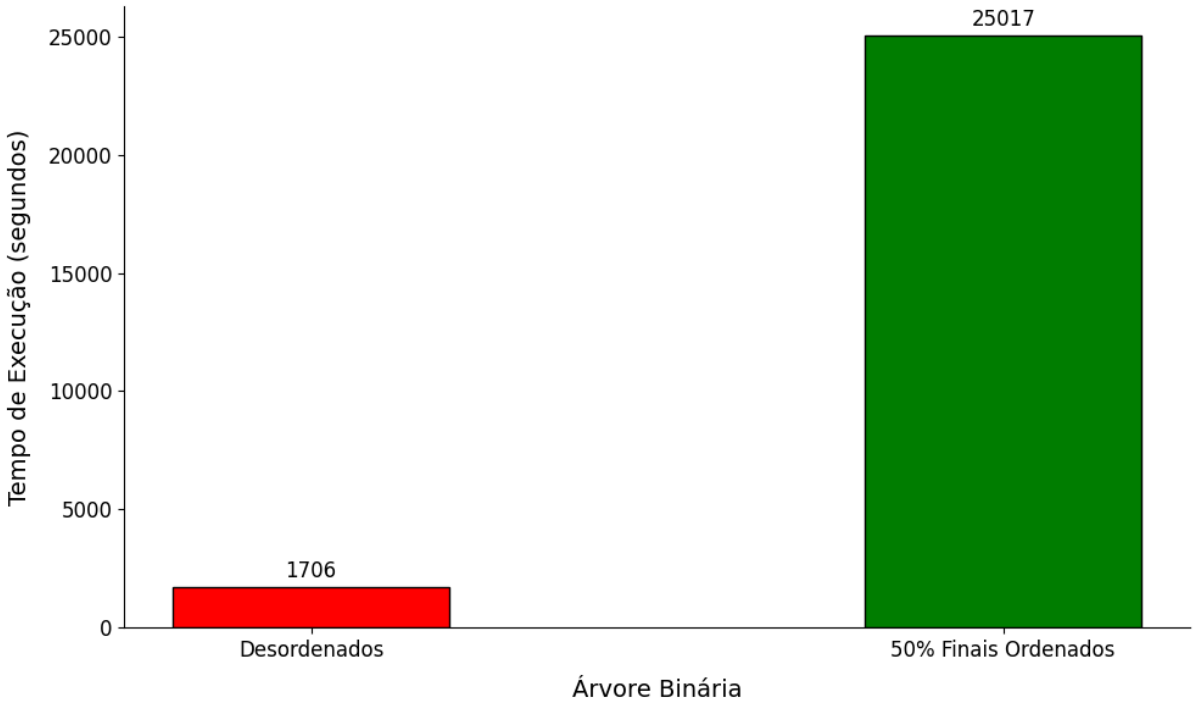
Comparando os resultados da base desordenada com a base parcialmente ordenada no gráfico 2, confirmamos a análise do gráfico 1. Por conta das menores alturas de árvores terem sido obtidas na base parcialmente ordenada, o tempo para inserir os valores consequentemente será menor do que nas árvores que apresentaram as maiores alturas. Em especial, podemos perceber que com a entrada de 3000000 dados na base desordenada, a altura cresce bastante por conta de uma escolha ineficiente para a raiz da árvore.

Quadro 3 - Tempo de execução de busca de 1 milhão de elementos em uma Árvore Binária de Busca de 3 milhões de elementos.

Tipo de Entrada	Total de Números Encontrados	Tempo De Execução
Desordenados	725	1706
Ordenados	stackoverflow	stackoverflow
Ordenados 50%	625	25017

Fonte: Elaboração própria com os dados analisados. (2024)

Gráfico 3 - Tempo de execução de busca de 1 milhão de elementos em uma Árvore Binária de Busca de 3 milhões de elementos.



Fonte: Elaboração própria com os dados analisados. (2024)

Pelo quadro 3, não é possível analisar a busca na árvore com os dados ordenados, pela limitação da pilha de chamadas.

Pelo gráfico 3, percebemos que o tempo de execução foi maior na base que apresentou a menor altura(base parcialmente ordenada), o que foge do que era esperado. Por conta da busca por um valor chave de um nó, depender totalmente da altura da árvore, a busca deveria ter sido mais eficiente na base parcialmente

ordenada, o que não aconteceu. O que pode ter acontecido, foi a escolha dos valores entre 0 e 500000 não ter favorecido a busca na árvore de menor altura. Considerando que nenhum elemento buscado foi encontrado na base parcialmente ordenada, cada busca teve que percorrer todos os níveis das árvores, e na base desordenada a maioria dos valores foram encontrados, esse resultado pode ser justificado. Ainda assim, mesmo considerando esse caso particular, os tempos de execução não deveriam ter sido tão discrepantes.

ÁRVORE AVL

- **Conceito**

Uma Árvore AVL proporciona uma estrutura que combina eficiência com balanceamento automático para operações de busca, inserção e remoção. A principal vantagem da Árvore AVL está em sua capacidade de manter o balanceamento dos nós, garantindo que a diferença de altura entre as subárvores de qualquer nó nunca seja maior que 1. Esse balanceamento assegura que a profundidade da árvore permaneça próxima de *O(log n)*, o que é essencial para que as operações sejam realizadas de maneira eficiente, mesmo em grandes volumes de dados.

A propriedade de balanceamento da Árvore AVL é mantida por meio de rotações, que são aplicadas automaticamente sempre que uma operação de inserção ou remoção cria um desequilíbrio. Essas rotações podem ser simples (esquerda ou direita) ou duplas (esquerda-direita ou direita-esquerda) permitindo que a árvore se reorganize, mantendo a ordem e garantindo que o tempo de acesso aos elementos seja otimizado.

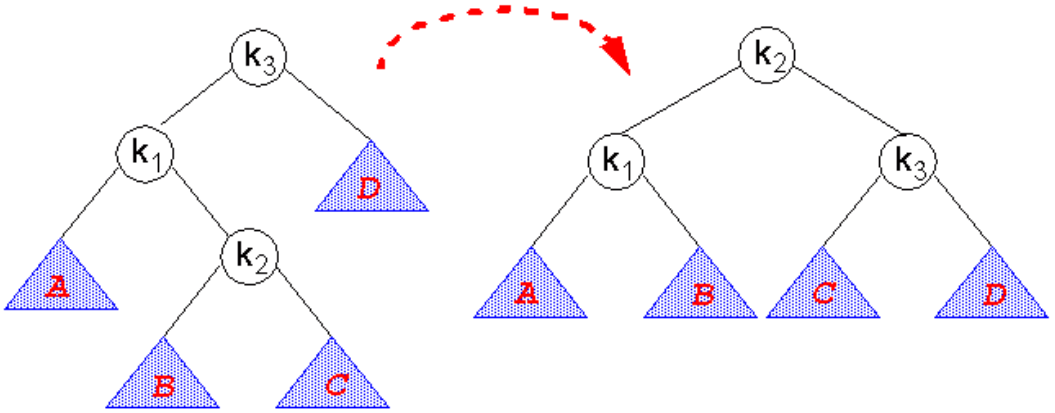


Figura 4 - Exemplo de uma árvore AVL. Fonte (https://www.inf.ufsc.br/~aldo.vw/estruturas/estru8-3.html, 2024)

- **Estrutura do Nó**

A estrutura da árvore AVL é semelhante a de outras árvores binárias de busca. Cada nó à esquerda possui um valor menor e cada nó à direita possui um valor maior do que o valor do nó atual. Essa propriedade facilita operações de busca e ordenação, permitindo que dados sejam percorridos de forma eficiente. Dessa maneira, a estrutura do nó da árvore AVL também permanece igual a árvore binária de busca.

- **Funcionamento**

A Árvore AVL funciona seguindo as regras de uma árvore binária de busca, onde a inserção de um novo nó posiciona valores menores à esquerda e valores maiores à direita. Após a inserção, o fator de balanceamento de cada nó ao longo do caminho até a raiz é atualizado, e, se algum nó ficar desbalanceado (com fator maior que 1 ou menor que -1), são aplicadas rotações para restaurar o equilíbrio. Na remoção de um nó, a árvore recalcula o fator de balanceamento dos nós ao longo do caminho até a raiz, e, caso haja desbalanceamento, rotações são realizadas para manter a árvore balanceada. As rotações podem ser simples ou duplas: a rotação **simples à direita** corrige desbalanceamentos à esquerda, enquanto a rotação **simples à esquerda** corrige desbalanceamentos à direita. As rotações duplas, como a **esquerda-direita e a direita-esquerda**, corrigem casos onde a subárvore esquerda ou direita do nó está desbalanceada na direção oposta. Após qualquer rotação, as subárvores são reorganizadas para manter a propriedade de busca e o fator de balanceamento dos nós é atualizado, fazendo com que a árvore retome seu estado balanceado.

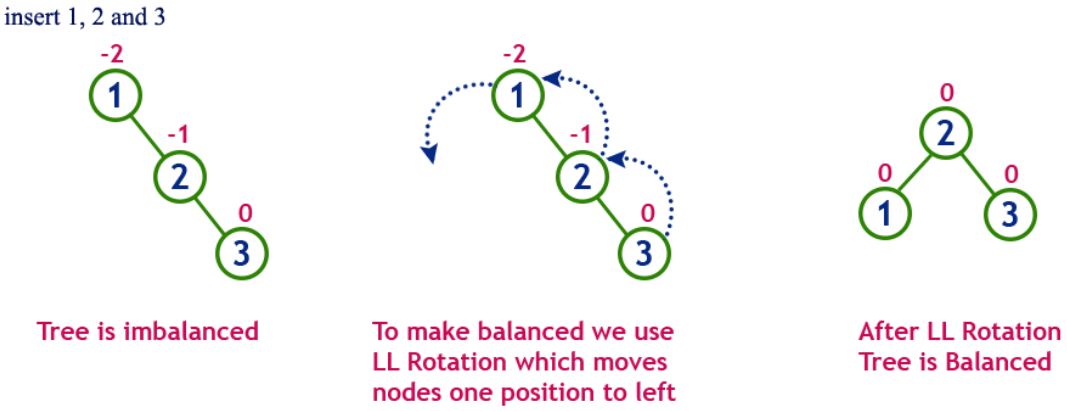


Figura 5- Exemplo de rotação simples à esquerda na árvore AVL. Fonte (http://www.btechsmartclass.com/data_structures/avl-trees.html, 2024)

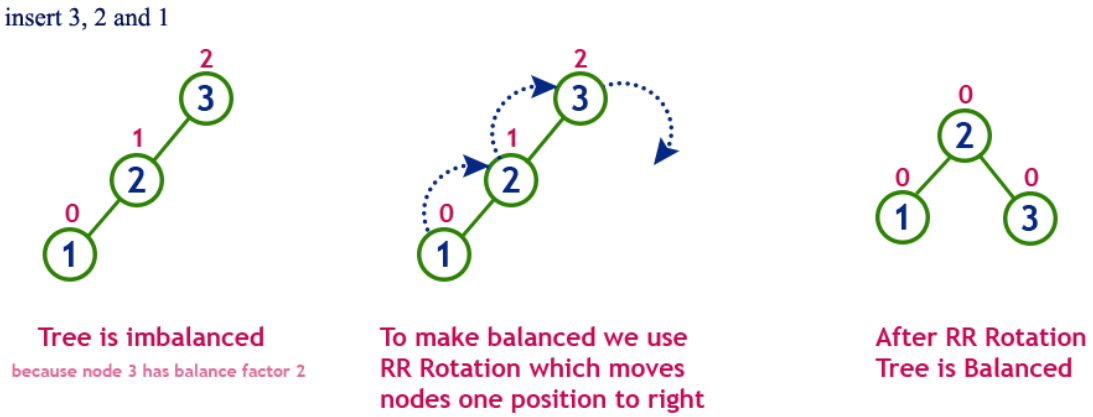


Figura 6 - Exemplo de rotação simples à direita na árvore AVL. Fonte (http://www.btechsmartclass.com/data_structures/avl-trees.html, 2024)

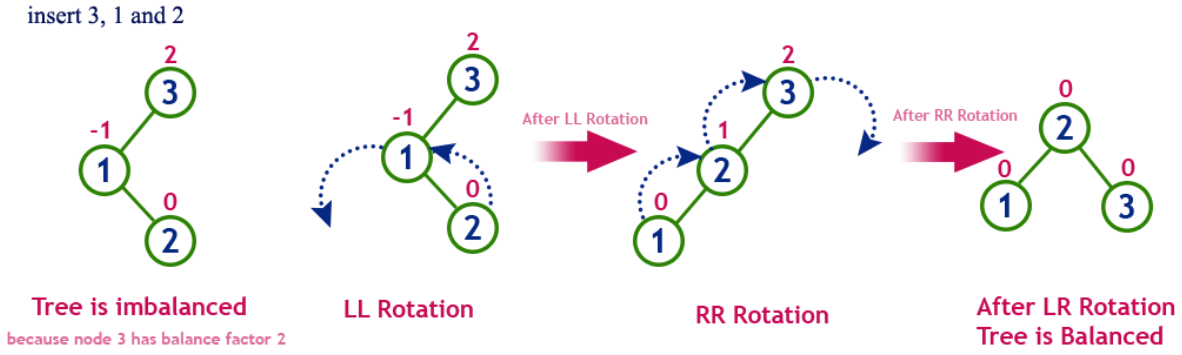


Figura 7 - Exemplo de rotação dupla esquerda-direita na árvore AVL. Fonte (http://www.btechsmartclass.com/data_structures/avl-trees.html, 2024)

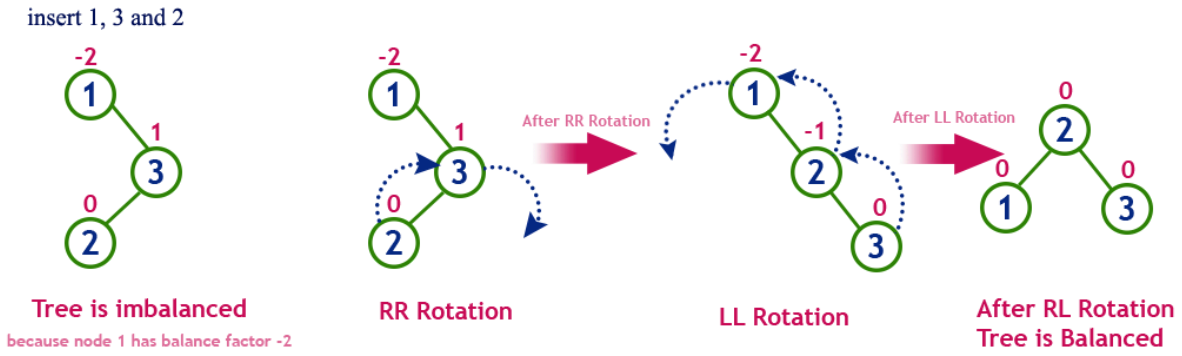


Figura 8 - Exemplo de rotação dupla direita-esquerda na árvore AVL. Fonte (http://www.btechsmartclass.com/data_structures/avl-trees.html, 2024)

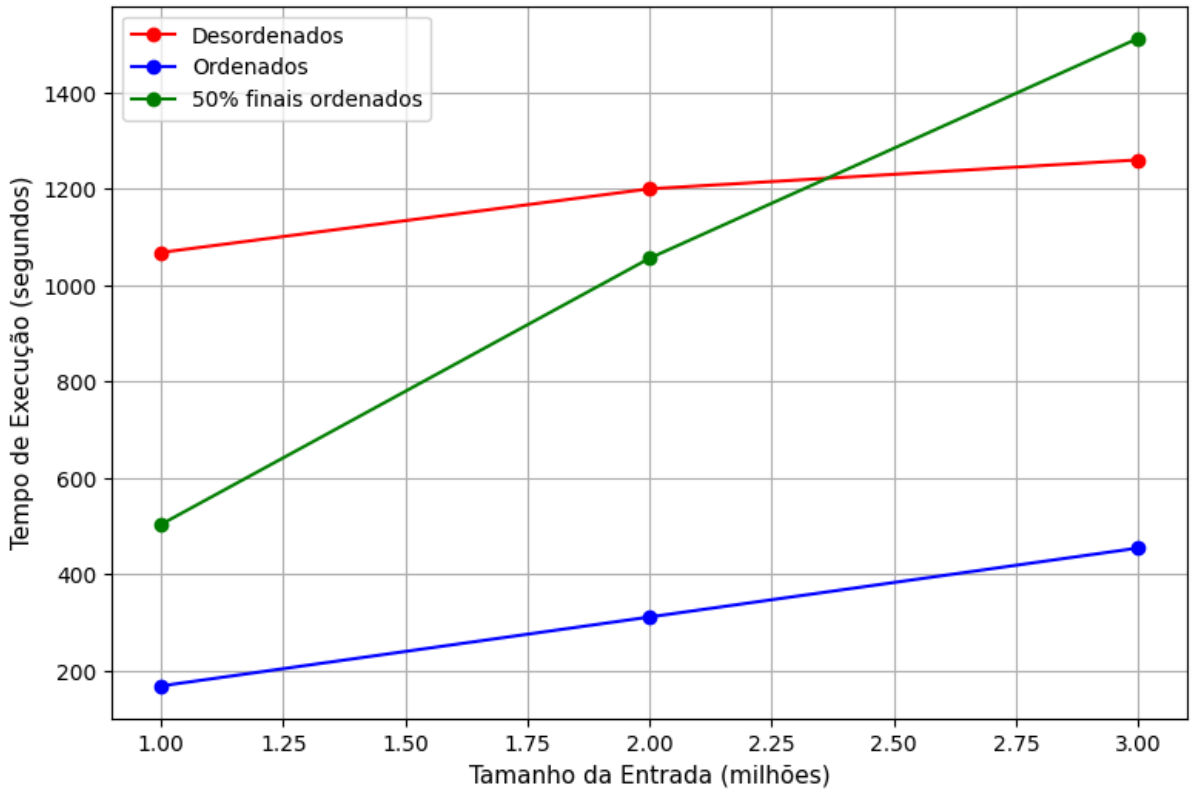
- **Análise**

Quadro 4 - Tempo de execução em milissegundos da inserção de elementos na árvore AVL.

Tamanho da Entrada	Desordenados	Ordenados	50% finais ordenados
1 milhão	1068	169	504
2 milhões	1200	312	1056
3 milhões	1260	455	1512

Fonte: Elaboração própria com os dados analisados. (2024)

Gráfico 4 - Tempo de execução em milissegundos da inserção de elementos na Árvore AVL.



Fonte: Elaboração própria com os dados analisados. (2024)

Conforme mostra o gráfico 4, os maiores tempos de execução ocorreram na base de dados desordenada e parcialmente ordenada, o que era esperado, pois esses conjuntos exigem mais operações de balanceamento, incluindo mais rotações para manter a estrutura AVL. Da mesma forma, o menor tempo de inserção foi obtido com dados totalmente ordenados, onde o padrão de inserção facilita o balanceamento, sendo necessário apenas rotações simples para balancear a árvore.

Um ponto interessante é que, no conjunto parcialmente ordenado (com os 50% finais ordenados), o tempo de inserção foi maior na base de dados de 3 milhões, até mesmo superando o dos dados desordenados. Isso ocorre porque, à medida que a árvore cresce, o número de subárvores e a quantidade de rotações necessárias aumentam. Com o crescimento da estrutura, o custo para manter a árvore balanceada também se eleva, o que explica o aumento no tempo de inserção. Quando a segunda metade dos dados começa a ser inserida de forma

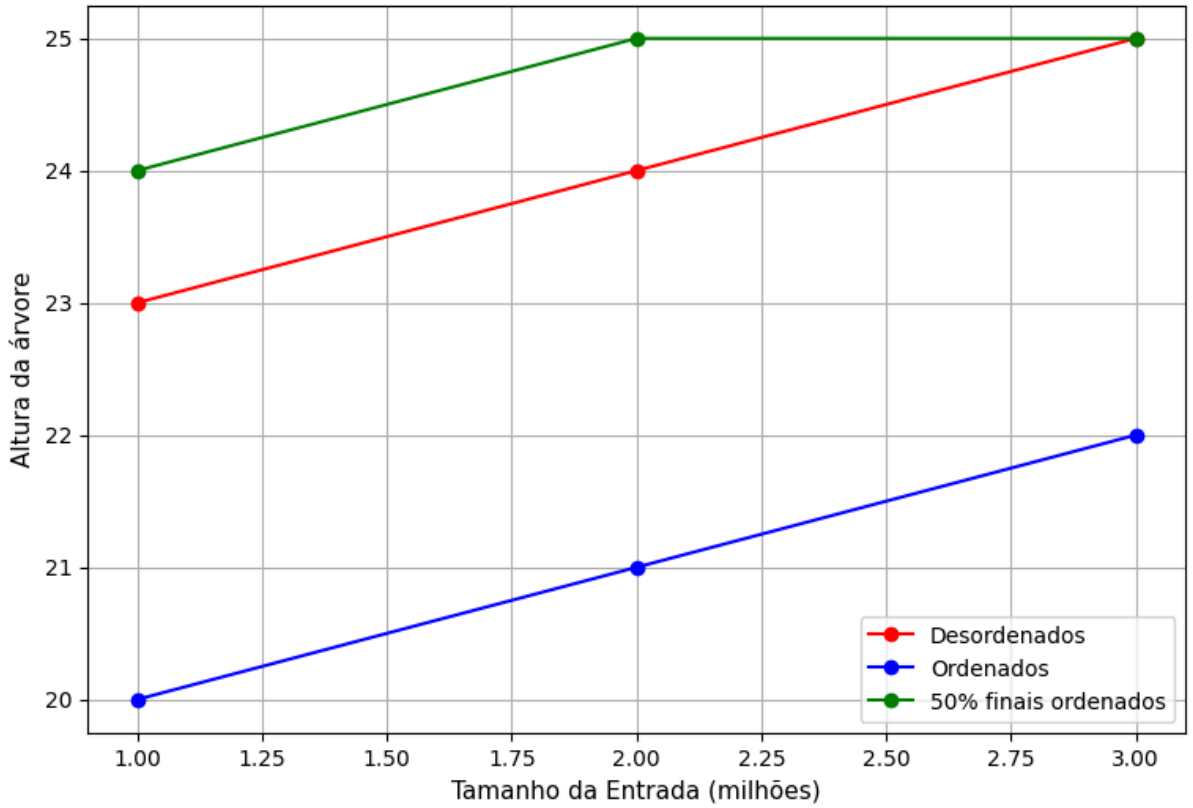
ordenada, a árvore já contém uma estrutura inicial com subárvores que precisam ser constantemente ajustados para acomodar o padrão parcialmente ordenado. Isso aumenta o número de rotações e o tempo de balanceamento, já que a árvore tenta ajustar o desequilíbrio introduzido pela primeira metade desordenada.

Quadro 5 - Altura da Árvore AVL de acordo com a entrada.

Tamanho da Entrada	Desordenados	Ordenados	50% finais ordenados
1 milhão	23	20	24
2 milhões	24	21	25
3 milhões	25	22	25

Fonte: Elaboração própria com os dados analisados. (2024)

Gráfico 5 - Altura da Árvore AVL de acordo com a entrada.



Fonte: Elaboração própria com os dados analisados. (2024)

De acordo com o gráfico X, observamos que a menor altura da árvore ocorre na base de dados com os dados previamente ordenados, conforme esperado. Por

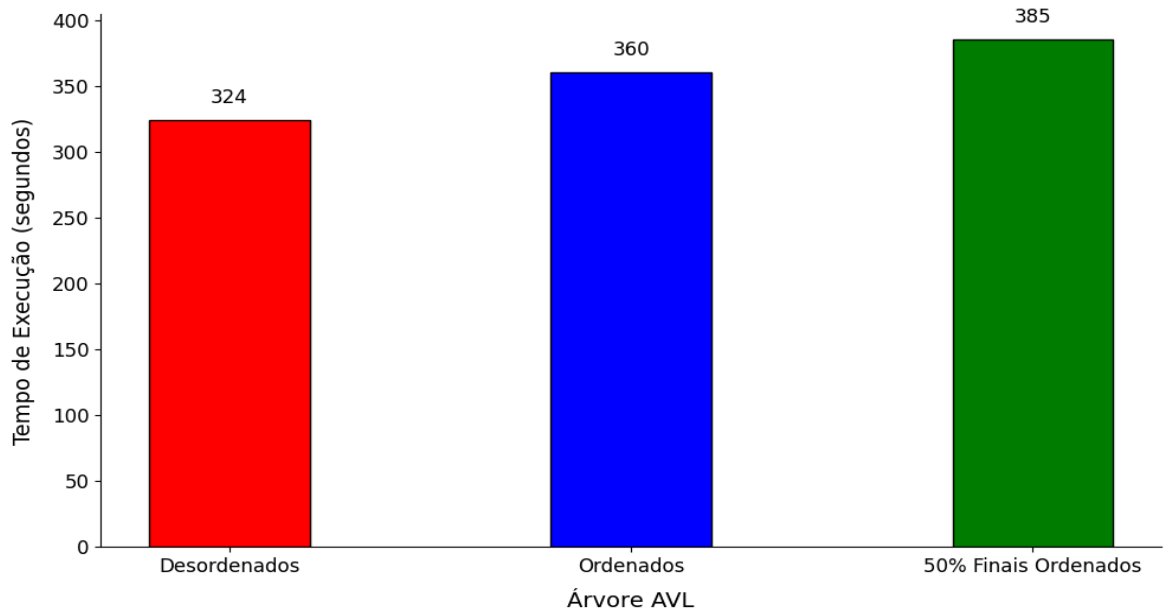
outro lado, a maior altura é observada na base de dados parcialmente ordenada, onde a primeira metade desordenada introduz desequilíbrios que a árvore precisa corrigir. Embora a segunda metade ordenada ajude a mitigar o impacto de novos desequilíbrios, o padrão misto ainda requer rotações adicionais, especialmente à medida que o número de elementos aumenta. Notavelmente, no maior conjunto de dados, com 3 milhões de elementos, houve um empate de altura entre os dados parcialmente ordenados e os totalmente desordenados. Apesar de tudo, as diferenças de altura não foram tão significativas, por conta da árvore ser auto balanceada, preservando uma altura média para a maioria dos casos.

Quadro 6 - Tempo de execução de busca de 1 milhão de elementos em uma Árvore AVL de 3 milhões de elementos.

Tipo de Entrada 3 Milhões	Total	Tempo De Execução
Desordenados	725	324
Ordenados	707	360
Ordenados 50%	625	385

Fonte: Elaboração própria com os dados analisados. (2024)

Gráfico 6 - Tempo de execução de busca de 1 milhão de elementos em uma Árvore AVL de 3 milhões de elementos.



Fonte: Elaboração própria com os dados analisados. (2024)

Como última análise da Árvore AVL nós temos o tempo de busca, onde quase tivemos os resultados esperados, que é o menor tempo para os dados ordenados. Porém, se observarmos no gráfico X, podemos ver que o menor tempo foi na estrutura que estava com os dados totalmente desordenados, mas como foi uma diferença extremamente mínima, podemos considerar sim que os resultados alcançaram a métrica almejada.

Para explicarmos essa pequena diferença, podemos levar em consideração que quando os dados são inseridos de forma desordenada em uma árvore AVL, a estrutura resultante pode ser mais equilibrada em termos de caminhos para busca, mesmo que a árvore em si tenha um número maior de rotações. Em certos casos, a distribuição dos elementos pode criar caminhos mais curtos para os elementos buscados, resultando em menos comparações durante a busca. Quando os dados são inseridos de forma desordenada em uma árvore AVL, a estrutura resultante pode ser mais equilibrada em termos de caminhos para busca, mesmo que a árvore em si tenha um número maior de rotações. Em certos casos, a distribuição dos elementos pode criar caminhos mais curtos para os elementos buscados, resultando em menos comparações durante a busca. Mas, mesmo assim, a diferença é extremamente mínima, sendo de apenas 36 milissegundos entre os dados desordenados e ordenados.

ÁRVORE SPLAY

- **Conceito**

Uma Árvore Splay desempenha um papel importante na ciência da computação, principalmente em contextos que a frequência de acesso a certos elementos não são uniformes. Ela oferece uma estrutura eficiente para operações de busca, inserção e remoção, destacando-se por trazer elementos recentemente acessados para o topo da árvore. Esse aspecto de splaying melhora o tempo de acesso a elementos que são frequentemente utilizados, tornando a árvore eficiente em cenários de acesso não uniformes.

A propriedade de uma Árvore Splay é que ela se reorganiza após cada operação de busca ou inserção, movendo o nó acessado para a raiz por meio de uma série de rotações (zig, zag, zig-zag ou zig-zig). Essa característica garante que operações subsequentes em elementos recentemente acessados sejam rápidas. A estrutura ainda mantém a propriedade de ordenação, no qual o valor de cada nó à esquerda é menor e o valor de cada nó à direita é maior do que o valor do nó atual.

As Árvores Splay foram projetadas para trabalhar com chaves únicas para simplificar as operações de busca e manipulação. Graças à sua propriedade de autoajuste, a Árvore Splay pode chegar a um desempenho de *O(log n)* para operações básicas, sendo muito útil em aplicações onde determinados elementos precisam ser acessados repetidamente.

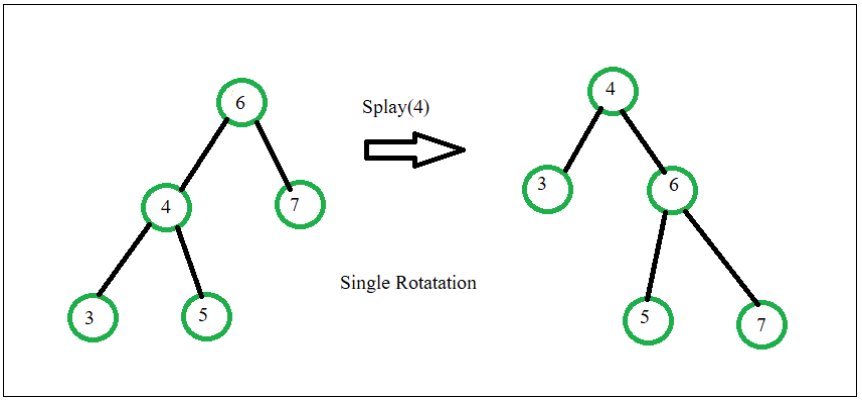


Figura 9 - Exemplo de uma Árvore Splay com rotação zig. Fonte (https://www.geeksforgeeks.org/introduction-to-splay-tree-data-structure/, 2024)

- **Estrutura do Nó**

A estrutura do nó da Árvore Splay é semelhante a uma árvore binária de busca (BST). Assim como nas outras árvores binárias de busca, na Árvore Splay cada nó possui um valor e, para cada nó, todos os valores à esquerda são menores e todos os valores à direita são maiores. No entanto, a diferença fundamental da Árvore Splay é o seu comportamento após operações de acesso.

- **Funcionamento**

A Árvore Splay funciona por meio de um processo chamado ***splaying***, que reorganiza a estrutura da árvore sempre que um nó é acessado. Quando você realiza uma operação de busca, inserção ou remoção, a árvore executa rotações, como **Zig** (move um nó que é filho direto da raiz para a posição de raiz, ajustando a árvore para que ele fique no topo), **Zag** (move um nó que é filho esquerda da raiz para a posição de raiz, ajustando a árvore para que ele fique no topo)**, Zig-Zig** (quando um nó e seu pai estão do mesmo lado (ambos à esquerda ou ambos à direita), a árvore faz duas rotações seguidas para levar o nó acessado para a raiz) e **Zig-Zag** (quando um nó e seu pai estão em lados opostos (um à esquerda e o outro à direita), a árvore faz duas rotações alternadas para levar o nó acessado para o topo) para mover o nó acessado para a raiz.

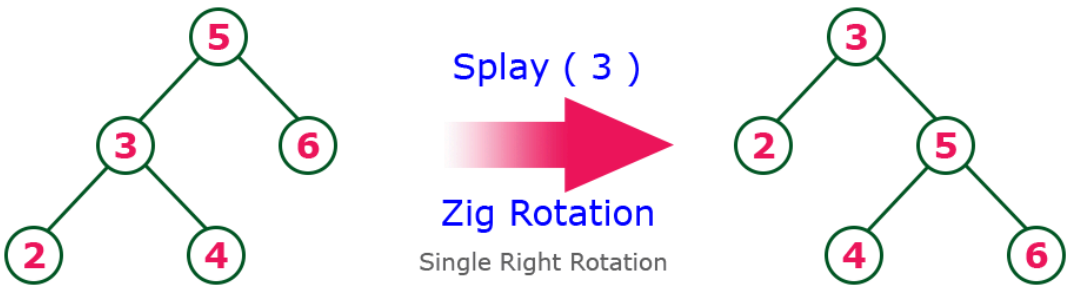


Figura 10 - Exemplo de uma Árvore Splay com rotação zig. Fonte (http://www.btechsmartclass.com/data_structures/splay-trees.html, 2024)

Figura 11 - Exemplo de uma Árvore Splay com rotação zag. Fonte (http://www.btechsmartclass.com/data_structures/splay-trees.html, 2024)

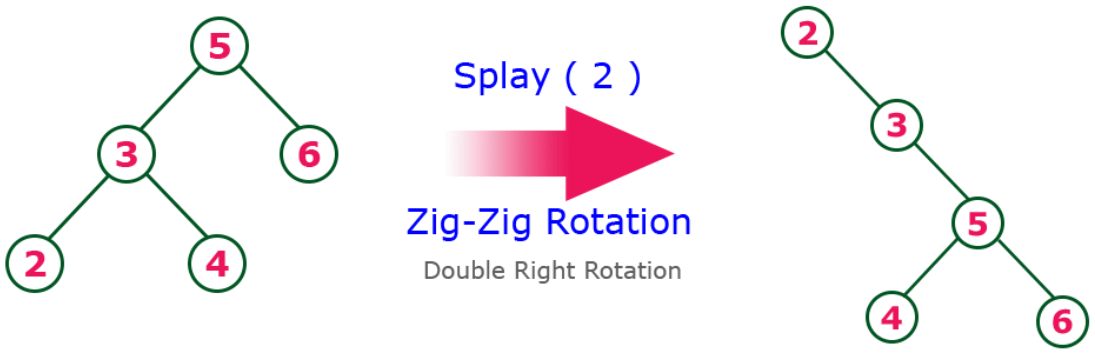


Figura 12 - Exemplo de uma Árvore Splay com rotação zig-zig. Fonte (http://www.btechsmartclass.com/data_structures/splay-trees.html, 2024)

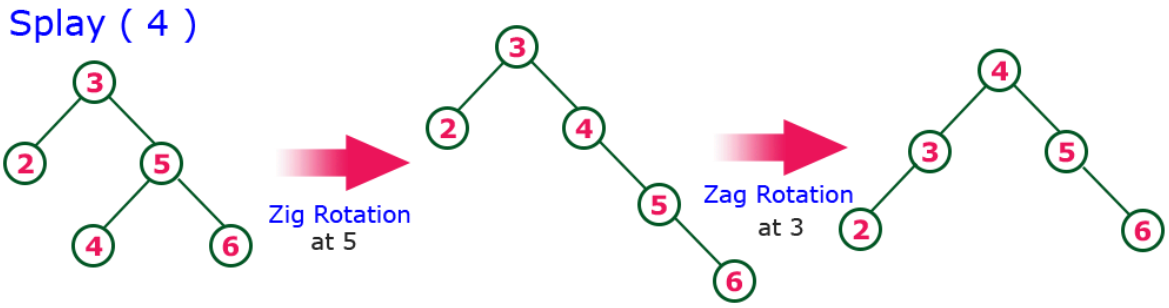


Figura 13 - Exemplo de uma Árvore Splay com rotação zig-zag. Fonte (http://www.btechsmartclass.com/data_structures/splay-trees.html, 2024)

Esse ajuste da árvore significa que os elementos que foram recentemente acessados ficam mais próximos da raiz, tornando as operações subsequentes mais rápidas. Embora algumas operações possam ser lentas em situações específicas, o tempo médio das operações se torna mais eficiente ao longo do uso contínuo, já que a árvore se adapta ao padrão de acesso. Essa característica a torna

especialmente útil em cenários onde certos dados são acessados com mais frequência.

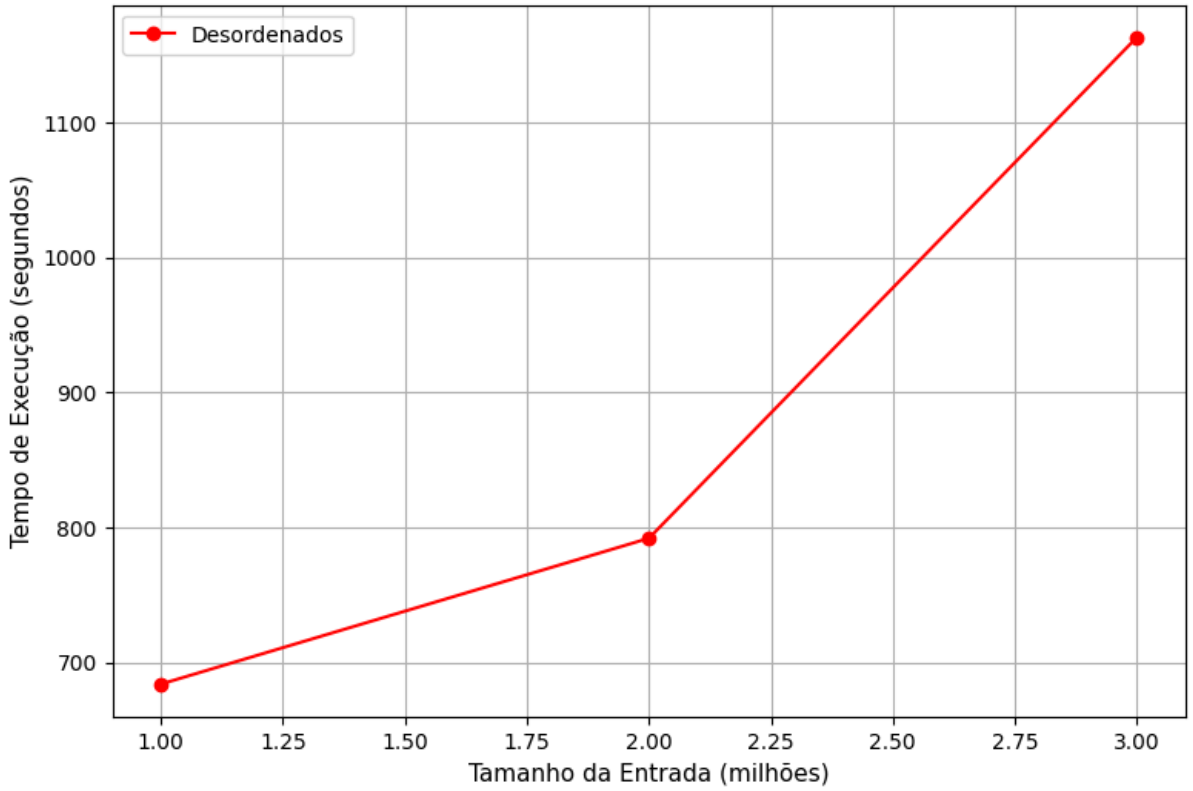
- **Análise**

Quadro 7 - Tempo de execução em milissegundos da inserção de elementos na Árvore Splay.

Tamanho da Entrada	Desordenados	Ordenados	50% finais ordenados
1 milhão	684	stackoverflow	stackoverflow
2 milhões	792	stackoverflow	stackoverflow
3 milhões	1163	stackoverflow	stackoverflow

Fonte: Elaboração própria com os dados analisados. (2024)

Gráfico 7 - Tempo de execução em milissegundos da inserção de elementos na Árvore Splay.



Fonte: Elaboração própria com os dados analisados. (2024)

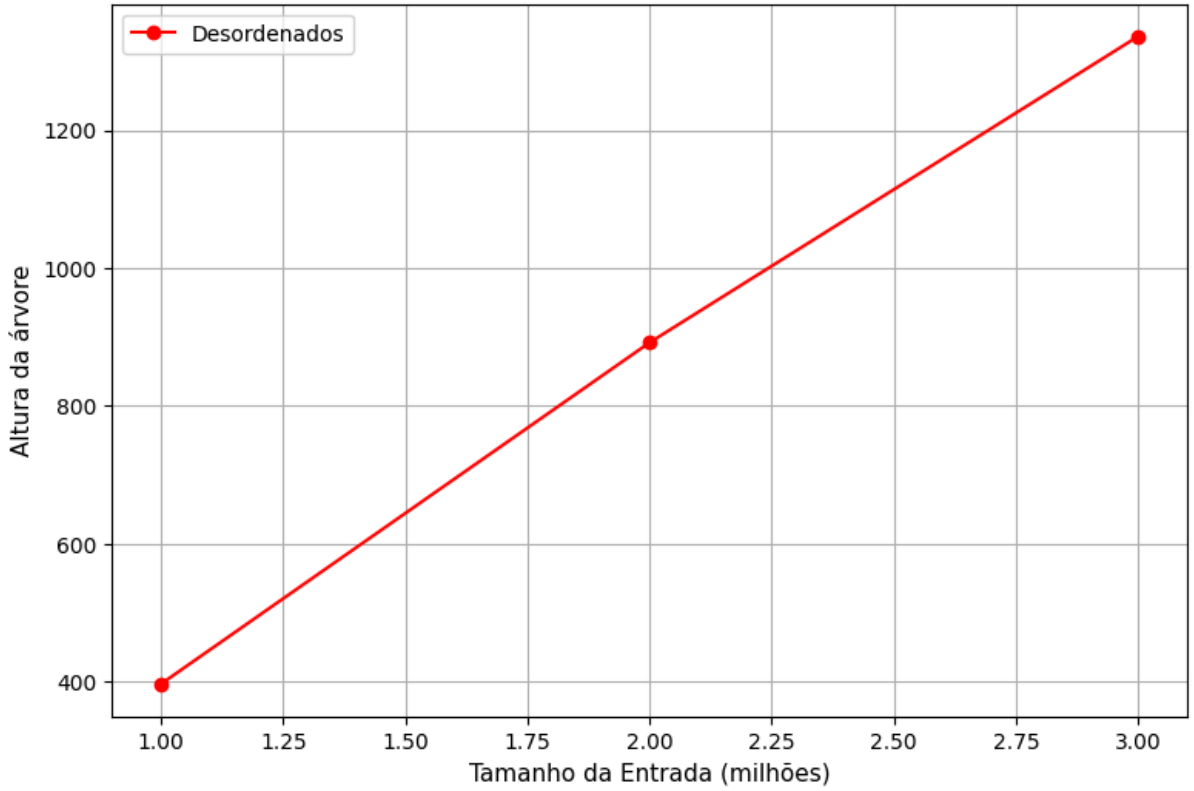
Pelo quadro 7 e gráfico 7, nas execuções das massas de testes ordenados e desordenados houve stack overflow pelo fato da função ser recursiva. Quando inserimos um elemento na árvore, checamos se ele é maior ou menor que as raízes de todas sub-árvores até que seja encontrado o lugar adequado para esse elemento (sem contar as rotações feitas para colocar o nó inserido na raiz).

Quadro 8 - Altura da Árvore Splay de acordo com a entrada.

Tamanho da Entrada	Desordenados	Ordenados	50% finais ordenados
1 milhão	397	stackoverflow	stackoverflow
2 milhões	1449	stackoverflow	stackoverflow
3 milhões	2202	stackoverflow	stackoverflow

Fonte: Elaboração própria com os dados analisados. (2024)

Gráfico 8 - Altura da Árvore Splay de acordo com a entrada.



Fonte: Elaboração própria com os dados analisados. (2024)

A altura dessa árvore não é a mínima possível, já que a preocupação não é balancear a árvore, mas sim posicionar o último nó acessado na raiz, independente se a árvore ficará desbalanceada.

Quadro 9 - Tempo de execução de busca de 1 milhão de elementos em uma Árvore Splay de 3 milhões de elementos.

Tipo de Entrada 3 Milhões	**Total**	**Tempo De Execução**
Desordenados	725	1479
Ordenados	stackoverflow	stackoverflow
Ordenados 50%	stackoverflow	stackoverflow

Fonte: Elaboração própria com os dados analisados. (2024)

Como mencionamos anteriormente, nessa estrutura tivemos stackoverflow, então não há como comparar resultados de busca com outros conjuntos de dados.

ÁRVORE RUBRO-NEGRA

- **Conceito**

A árvore rubro-negra é uma estrutura de dados do tipo árvore binária de busca auto-balanceada que garante operações eficientes de busca, inserção e remoção. Seu principal diferencial é manter o balanceamento da árvore por meio de regras de coloração dos nós, que podem ser vermelhos ou pretos. Essas regras asseguram que o caminho da raiz até qualquer folha tenha um número aproximadamente igual de nós pretos, evitando que a árvore se degrade em uma estrutura desbalanceada, como uma lista encadeada.

As regras básicas de uma árvore rubro-negra são:

1. Cada nó é vermelho ou preto.
2. A raiz da árvore é sempre preta.
3. Nós vermelhos não podem ter filhos vermelhos (ou seja, não pode haver dois nós vermelhos consecutivos).
4. Todo caminho da raiz até uma folha ou nó nulo deve ter o mesmo número de nós pretos.
5. Nós nulos (nós externos) são considerados pretos.

Quando uma operação de inserção ou remoção é realizada e quebra uma dessas regras, a árvore é reajustada por meio de colorações e rotações (simples ou duplas) para restaurar as propriedades de balanceamento. Esse balanceamento dinâmico garante que a altura da árvore seja limitada a O(logn), permitindo que as operações de busca, inserção e remoção mantenham essa eficiência mesmo em grandes conjuntos de dados.

A árvore rubro-negra é amplamente usada em implementações de estruturas de dados fundamentais, como dicionários e conjuntos ordenados, devido ao seu desempenho consistente.

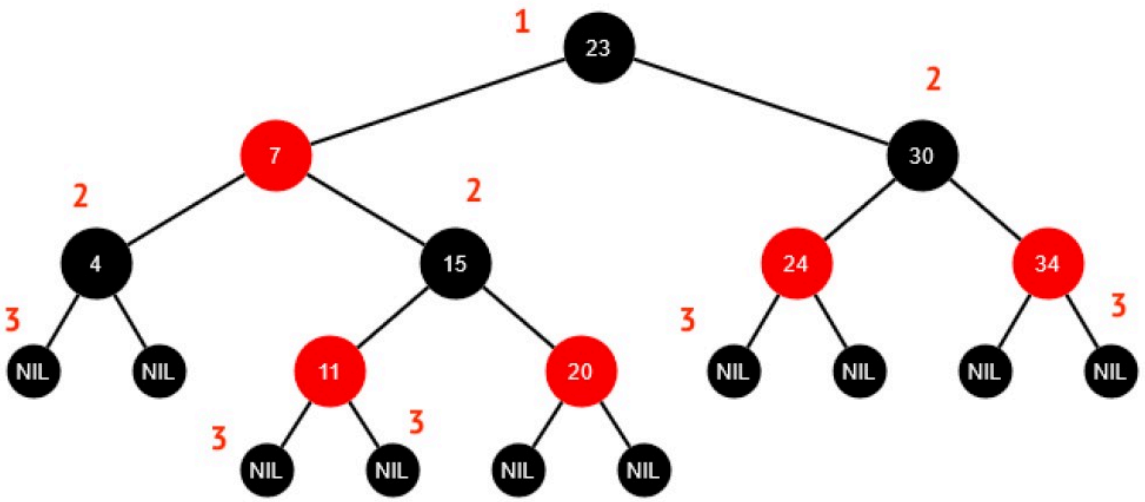


Figura 14 - Exemplo de árvore rubro-negra. Fonte (https://javarush.com/pt/groups/posts/pt.4165.rvore-rubro-negra-propriedades-princpios-de-organizao-mecanismo-de-insero, 2024)

- **Estrutura do Nó**

A estrutura de um nó em uma árvore rubro-negra contém alguns componentes essenciais. Primeiro, há o valor ou chave, que armazena o valor do nó e é utilizado para manter a ordem na árvore, similar ao que ocorre em uma árvore binária de busca. Em seguida, existe um campo em booleano que indica a cor do nó, que pode ser vermelho ou preto, sendo essa cor fundamental para as propriedades de balanceamento da árvore. Além disso, cada nó possui um ponteiro que aponta para o filho à esquerda e outro que aponta para o filho à direita. Opcionalmente, pode haver um ponteiro para o nó pai, o que facilita as operações de inserção e remoção, permitindo um acesso direto ao nó pai.

```
class Node {
    int key;
    Node left, right, parent;
    boolean isRed;
```

Figura 15 - Exemplo da estrutura de um nó de uma árvore rubro-negra. **Fonte:** código utilizado para este relatório, 2024.

- **Funcionamento**

O funcionamento da árvore rubro-negra é orientado por suas propriedades de balanceamento e regras que garantem eficiência nas operações. Ao inserir um novo nó, ele é adicionado como um nó vermelho, seguindo as regras de uma árvore binária de busca, onde valores menores vão para a esquerda e maiores para a direita. Após a inserção, é necessário verificar se as propriedades da árvore foram mantidas.

As principais regras incluem que cada nó é vermelho ou preto, a raiz é sempre preta, não pode haver dois nós vermelhos consecutivos, e todo caminho da raiz até as folhas deve conter o mesmo número de nós pretos. Caso alguma dessas propriedades seja violada, são aplicadas operações de recoloração e rotações.

A recoloração envolve mudar a cor dos nós para restaurar o equilíbrio. Se um nó vermelho tiver um pai vermelho, e o tio (irmão do pai) também for vermelho, ambos são coloridos para preto, e o avô se torna vermelho. Isso pode ocorrer repetidamente até que a árvore esteja balanceada.

As rotações são usadas quando o tio é preto ou nulo. A rotação à direita corrige um desbalanceamento à esquerda, enquanto a rotação à esquerda corrige um desbalanceamento à direita. Casos de rotações dupla (esquerda-direita ou direita-esquerda) também são utilizados quando um nó é inserido em posições específicas.

Na remoção, o processo segue regras semelhantes às da árvore binária de busca, mas requer atenção especial se o nó removido for preto, pois isso pode causar desbalanceamento. Assim, a árvore rubro-negra mantém sua altura próxima de $O(\log n)$, garantindo que as operações de busca, inserção e remoção sejam eficientes.

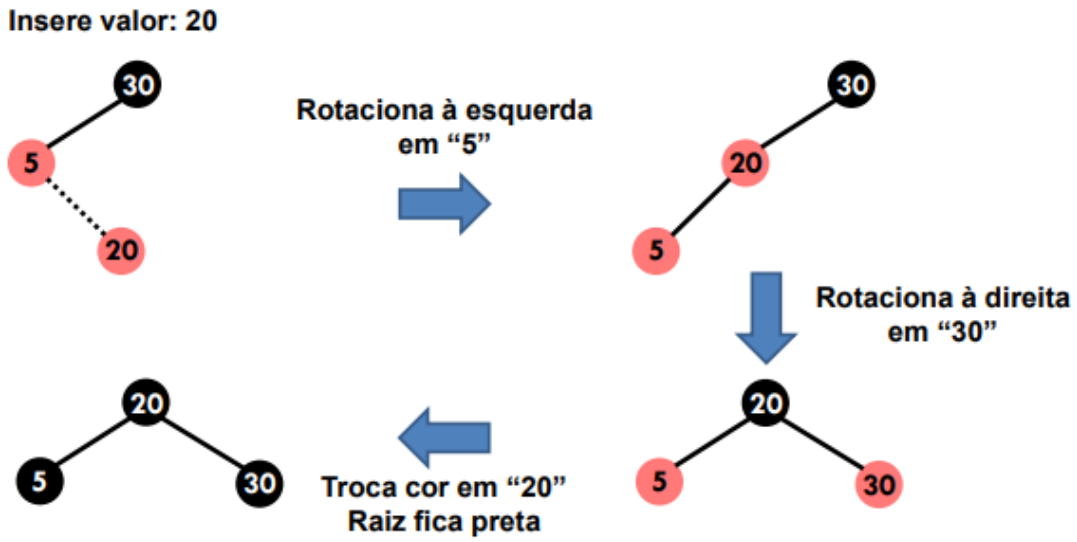


Figura 16 - Exemplo de inserção de elemento na árvore rubro-negra. Fonte (https://www.facom.ufu.br/~backes/gsi011/Aula12-ArvoreRB.pdf, 2024)

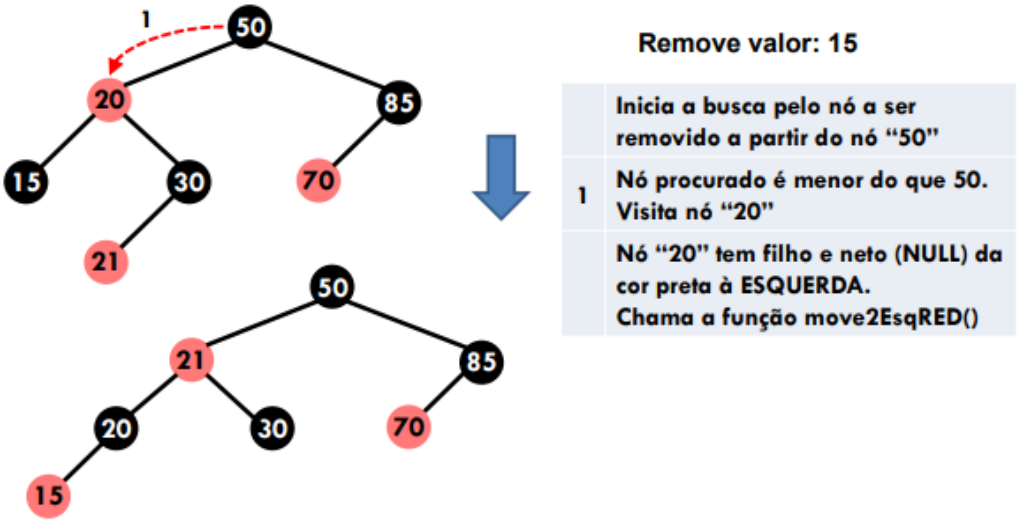


Figura 17 - Exemplo de remoção de elementos na árvore rubro-negra parte 1. Fonte (https://www.facom.ufu.br/~backes/gsi011/Aula12-ArvoreRB.pdf, 2024)

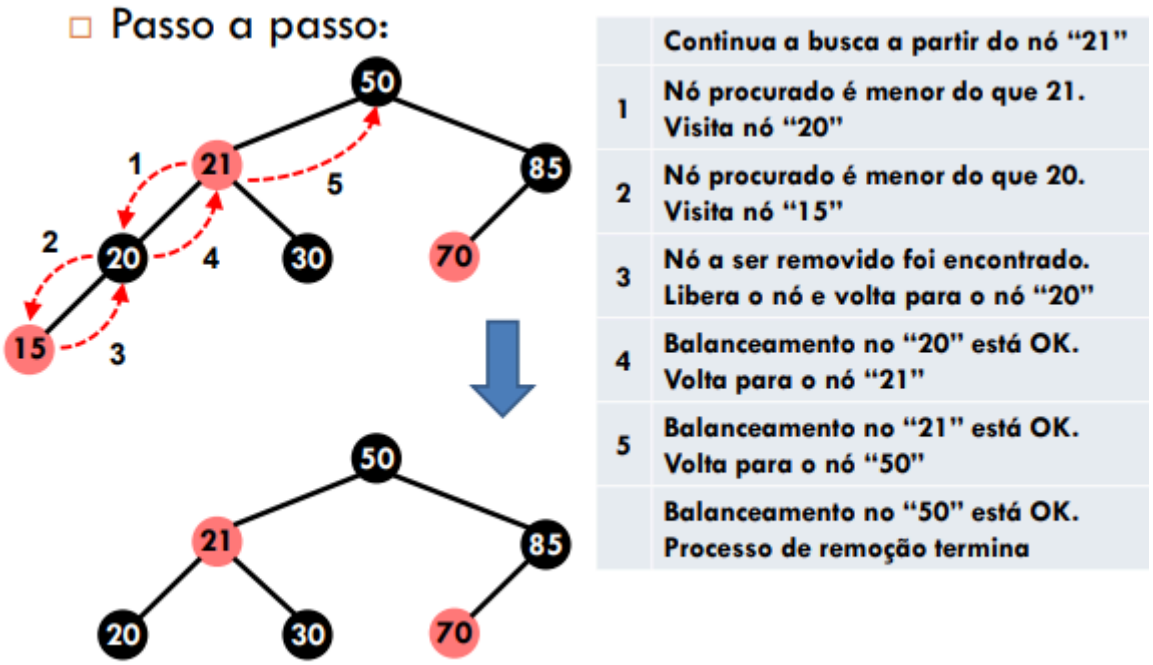


Figura 18 - Exemplo de remoção de elementos na árvore rubro-negra parte 2. Fonte (https://www.facom.ufu.br/~backes/gsi011/Aula12-ArvoreRB.pdf, 2024)

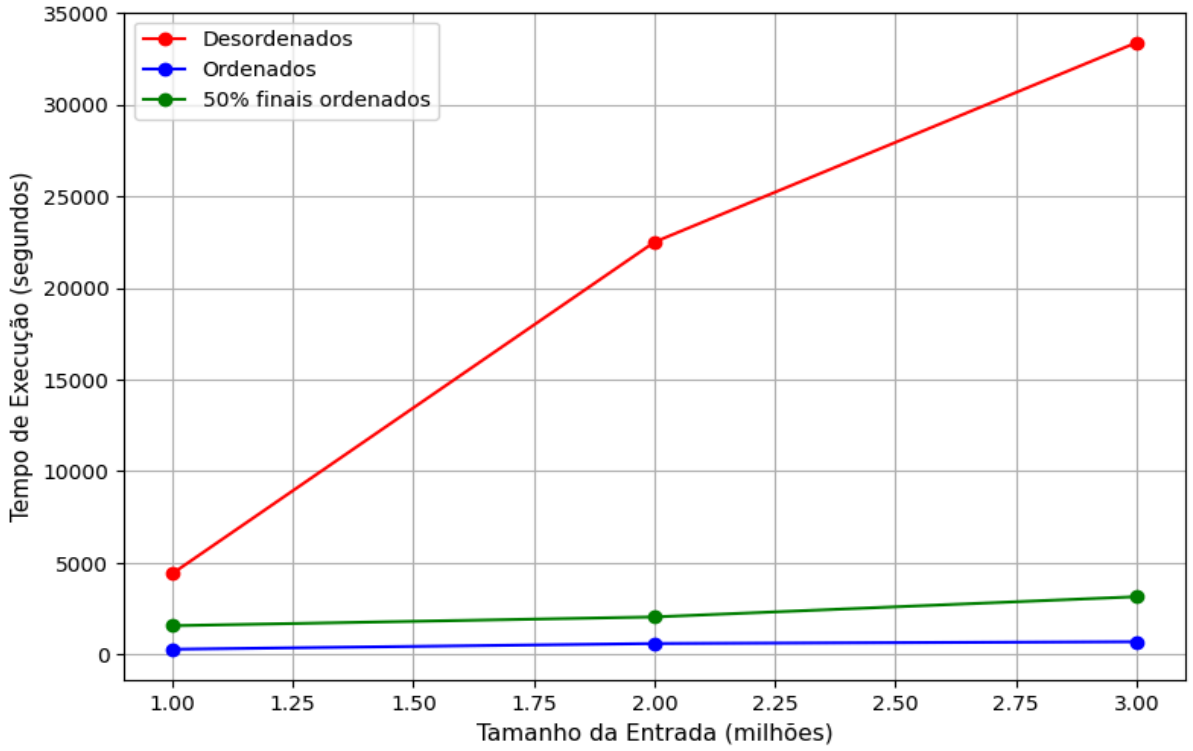
- **Análise**

Quadro 10 - Tempo de execução em milissegundos da inserção de elementos na Árvore Rubro.

Tamanho da Entrada	Desordenados	Ordenados	50% finais ordenados
1 milhão	4403	276	1561
2 milhões	22499	589	2040
3 milhões	33389	685	3145

Fonte: Elaboração própria com os dados analisados. (2024)

Gráfico 9 - Tempo de execução em milissegundos da inserção de elementos na Árvore Rubro.



Fonte: Elaboração própria com os dados analisados. (2024)

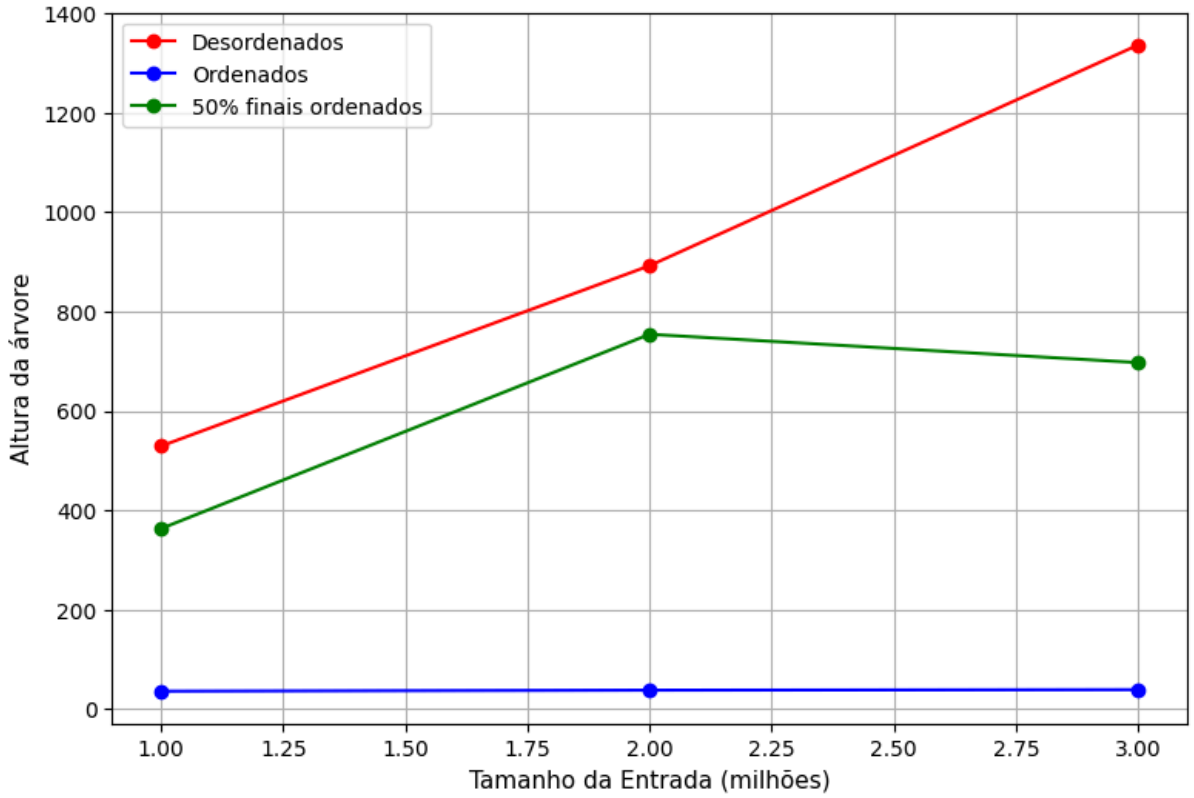
De acordo com o gráfico 9, é possível verificar que tivemos um tempo maior de inserção de elementos na base de dados desordenada, pois a estrutura teve que fazer mais operações de balanceamento nos níveis da árvore, já que os dados se encontram de forma desordenada, fora de um padrão. Esse resultado foi o esperado: obtivemos o melhor tempo de inserção nos dados totalmente ordenados, onde não foram necessárias tantas operações de balanceamento, pois os dados eram inseridos de acordo com um padrão.

Quadro 11 - Altura da Árvore Rubro de acordo com a entrada.

Tamanho da Entrada	Desordenados	Ordenados	50% finais ordenados
1 milhão	529	36	363
2 milhões	892	38	754
3 milhões	1336	39	697

Fonte: Elaboração própria com os dados analisados. (2024)

Gráfico 10 - Altura da Árvore Rubro de acordo com a entrada.



Fonte: Elaboração própria com os dados analisados. (2024)

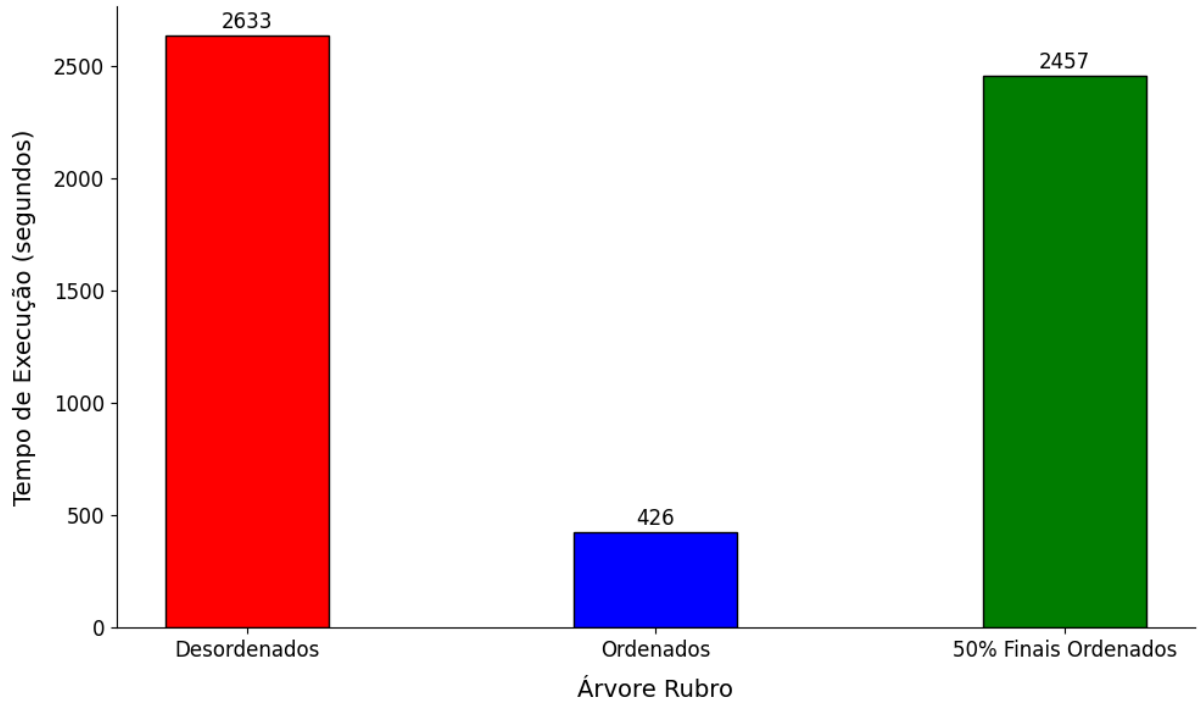
Em relação a altura da árvore rubro-negra, também obtivemos os resultados esperados, como podemos observar no gráfico 12, com o melhor resultado sendo onde os dados já estavam ordenados, gerando um número bem próximo a O(log n), pois a árvore manteve um padrão de inserção. Já no pior caso, tivemos como maior altura os dados desordenados, pois embora a estrutura faça diversas operações de balanceamento contínuo, a árvore tende a ter uma altura maior nos dados fora de ordem, já que os ajustes realizados em cada inserção são locais (afetando apenas subárvores próximas ao nó inserido).

Quadro 12 - Tempo de execução de busca de 1 milhão de elementos em uma Árvore Rubro de 3 milhões de elementos.

Tipo de Entrada 3 Milhões	Total	Tempo De Execução
Desordenados	725	324
Ordenados	707	360
Ordenados 50%	625	385

Fonte: Elaboração própria com os dados analisados. (2024)

Gráfico 11 - Tempo de execução de busca de 1 milhão de elementos em uma Árvore Rubro de 3 milhões de elementos.



Fonte: Elaboração própria com os dados analisados. (2024)

Em uma árvore balanceada, o tempo de busca é mais eficiente, pois o algoritmo consegue direcionar a busca de acordo com a posição relativa dos dados

na estrutura. Como o conjunto de dados ordenados resultou na menor altura, conforme explicado anteriormente, ele apresentou o menor tempo de busca, como podemos verificar no gráfico 13. O mesmo ocorre com os dados desordenados, que tiveram o maior tempo de busca devido à maior altura da árvore.

COMPARAÇÃO DE ROTAÇÕES

- **Árvore AVL**

Quadro 13 - Número de rotações de uma árvore AVL em um conjunto de dados desordenados.

	Esquerda	**Direta**
1 milhão	653066	520113
2 milhões	1281566	1110539
3 milhões	1898661	1697531

Fonte: Elaboração própria com os dados analisados. (2024)

Quadro 14 - Número de rotações de uma árvore AVL em um conjunto de dados ordenados.

	Esquerda	**Direta**
1 milhão	999980	0
2 milhões	1999979	0
3 milhões	2999978	0

Fonte: Elaboração própria com os dados analisados. (2024)

Quadro 15 - Número de rotações de uma árvore AVL em um conjunto de dados com os 50% finais desordenados.

	Esquerda	**Direta**
1 milhão	668848	319256
2 milhões	1345692	651154
3 milhões	2019728	2019728

Fonte: Elaboração própria com os dados analisados. (2024)

Quadro 16 - Número de rotações de uma árvore rubro-negra em um conjunto de dados desordenados.

	Esquerda	**Direta**
1 milhão	622034	512684
2 milhões	1211363	1073870
3 milhões	1784312	1643883

Fonte: Elaboração própria com os dados analisados. (2024)

Quadro 17 - Número de rotações de uma árvore rubro-negra em um conjunto de dados ordenados.

	Esquerda	**Direta**
1 milhão	999963	0
2 milhões	1999961	0
3 milhões	2999960	0

Fonte: Elaboração própria com os dados analisados. (2024)

Quadro 18 - Número de rotações de uma árvore rubro-negra em um conjunto de dados com os 50% finais desordenados.

	Esquerda	**Direta**
1 milhão	742371	372366
2 milhões	1492566	756682
3 milhões	2241558	1092639

Fonte: Elaboração própria com os dados analisados. (2024)

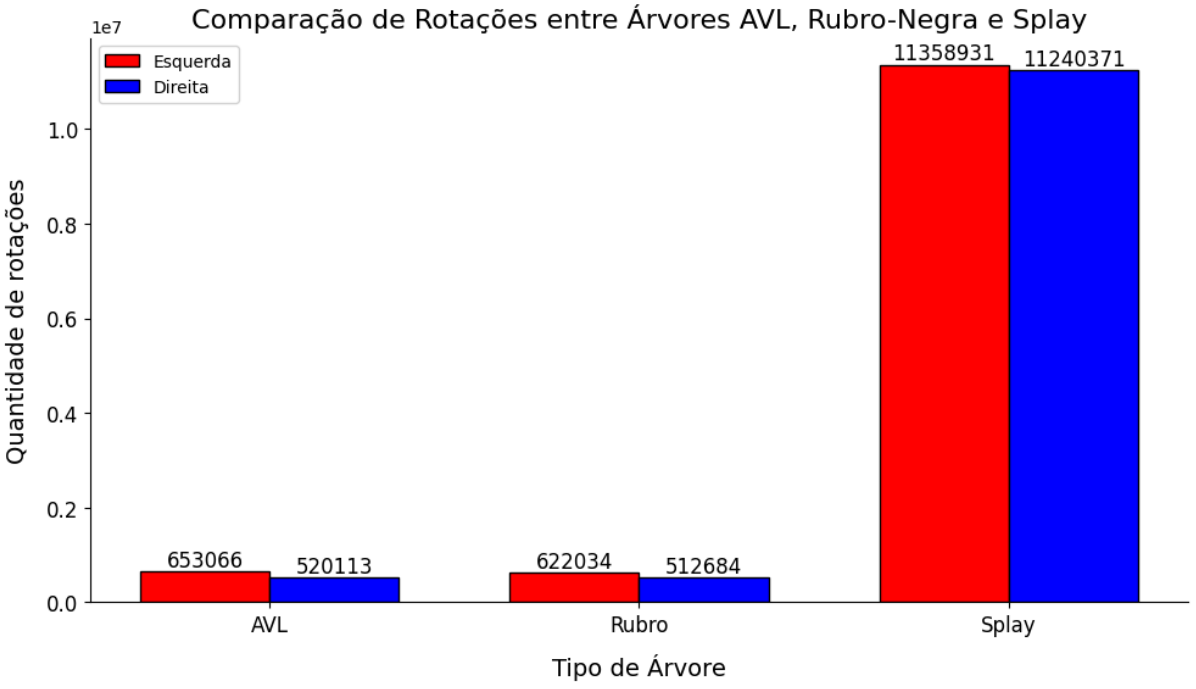
- **Árvore Splay**

Quadro 19 - Número de rotações de uma árvore splay em um conjunto de dados desordenados.

	Esquerda	**Direta**
1 milhão	11358931	11240371
2 milhões	20903750	20699850
3 milhões	31437816	31184799

Fonte: Elaboração própria com os dados analisados. (2024)

Gráfico 12 - Comparação da quantidade de rotações da Árvore AVL, Rubro-Negra e Splay em um conjunto de 1 milhão de dados desordenados, ordenados e com os 50% finais ordenados



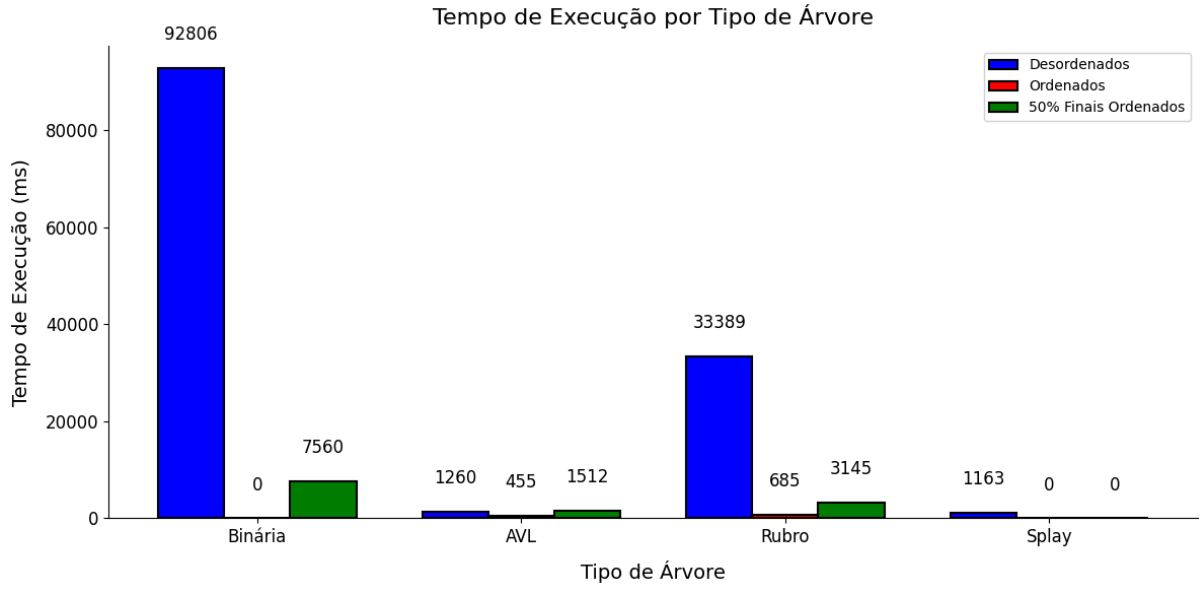
Fonte: Elaboração própria com os dados analisados. (2024)

Sobre a quantidade de rotações, a que apresentou um maior número em relação às demais foi a Splay, o que é totalmente esperado, pois essa estrutura faz

vários movimentos de ajustes toda vez que um elemento é acessado, fazendo com que a árvore passe por várias alterações.

CONSIDERAÇÕES GERAIS

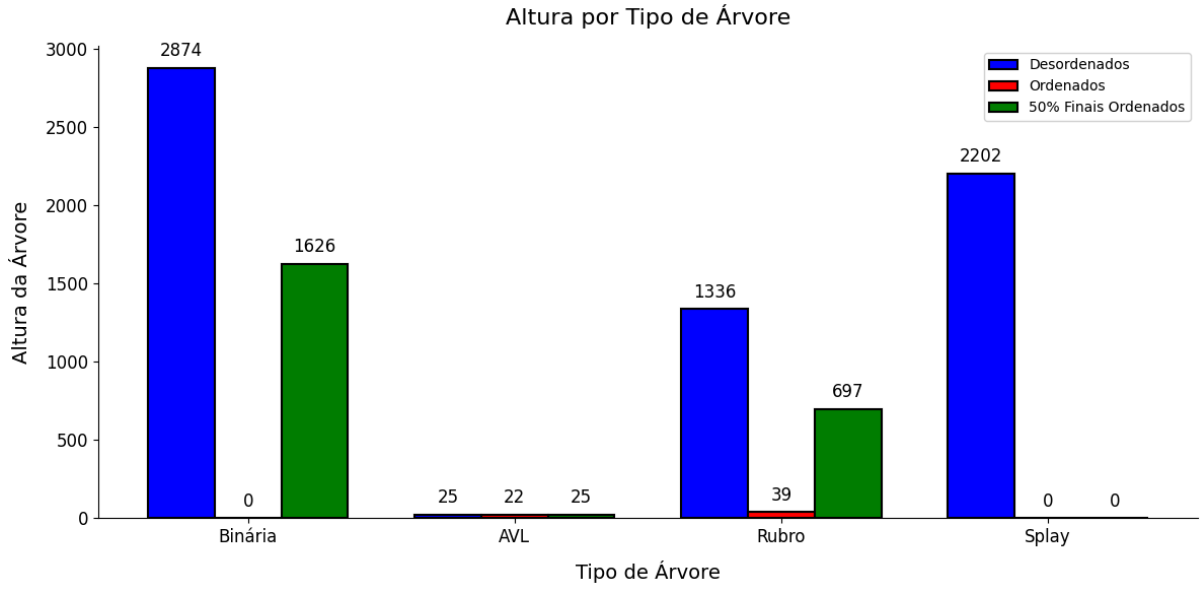
Gráfico 13 - Comparação do tempo de inserção de elementos em uma Árvore Binária de Busca, AVL, Rubro-Negra e Splay em um conjunto de 3 milhões de dados desordenados, ordenados e com os 50% finais ordenados



Fonte: Elaboração própria com os dados analisados. (2024)

Comparando os tempos de execução de inserção de elementos da árvore splay e da árvore AVL (as que tiveram menor tempo como podemos ver no gráfico 13), percebemos que o da Splay é menor. Porém, a árvore AVL não sofreu de stackoverflow como mencionado nos tópicos anteriores, o que a deixa como melhor estrutura na inserção de elementos, Isso se dá pois a árvore AVL tem um balanceamento mais simples, poucos nós são rotacionados a cada inserção, em comparação com a árvore Splay.

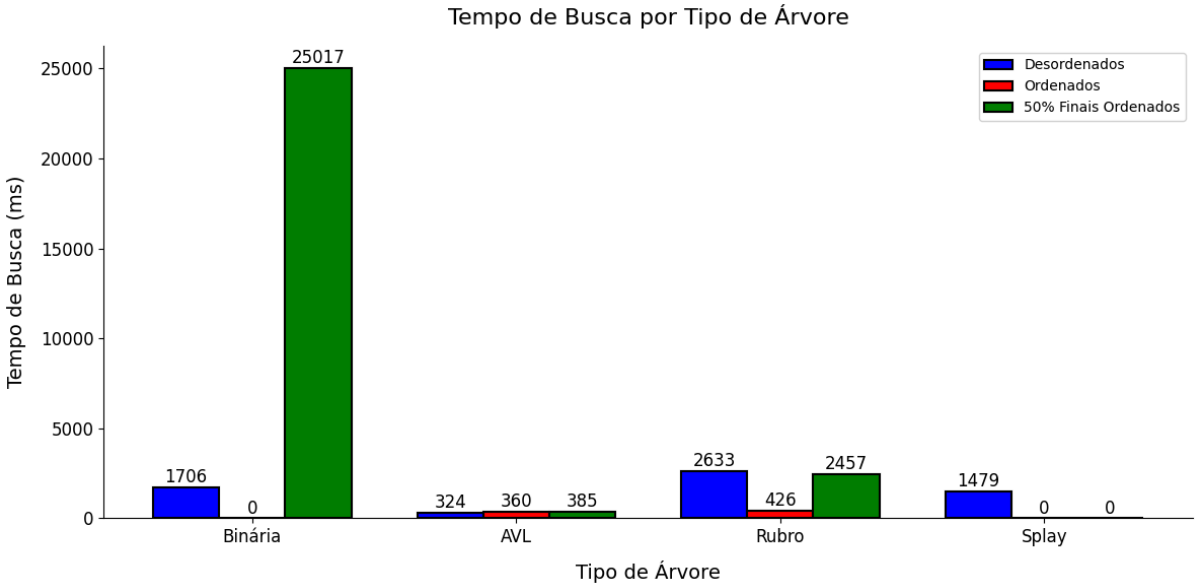
Gráfico 14 - Comparação da altura de uma Árvore Binária de Busca, AVL, Rubro-Negra e Splay em um conjunto de 3 milhões de dados desordenados, ordenados e com os 50% finais ordenados.



Fonte: Elaboração própria com os dados analisados. (2024)

Em relação a altura das árvores, a que obteve menor valor foi a AVL, como podemos observar no gráfico 14. Isso ocorre pois essa estrutura tende a ficar mais balanceada, devido ao seu método de inserção de rotações simples e duplas.

Gráfico 15 - Comparação do tempo de busca de 1 milhão de elementos em uma Árvore Binária de Busca, AVL, Rubro-Negra e Splay em um conjunto de 3 milhões de dados desordenados, ordenados e com os 50% finais ordenados.



Fonte: Elaboração própria com os dados analisados. (2024)

Por fim, o tempo de busca se mostrou mais eficiente naquelas estruturas em que as alturas foram inferiores, ou seja, mais uma vez a Árvore AVL se destaca, como podemos ver no gráfico 15.

REFERÊNCIAS

---

1. Repositório com o código. GitHub - sonallycecilia/projeto3-EDA. Disponível em: <https://github.com/sonallycecilia/projeto3-EDA>. Acesso em: 5 nov. 2024.

2. Árvores: estrutura de dados - Algol.dev - com ilustrações e animações. Disponível em: <https://algol.dev/arvores-estrutura-de-dados/>.

3. DOS, C. tipo abstrato de dados. Disponível em: <https://pt.wikipedia.org/wiki/%C3%81rvore_(estrutura_de_dados)>.

4. Árvores de Busca Semibalanceadas. Disponível em: <https://www.inf.ufsc.br/~aldo.vw/estruturas/estru8-3.html>.

5. Data Structures Tutorials - AVL Tree | Examples | Balance Factor. Disponível em: <http://www.btechsmartclass.com/data_structures/avl-trees.html>.

6. Introduction to Splay tree data structure. Disponível em: <https://www.geeksforgeeks.org/introduction-to-splay-tree-data-structure/>.

7. Data Structures Tutorials - Splay Tree with an example. Disponível em: <http://www.btechsmartclass.com/data_structures/splay-trees.html>.

8. REALYTE. Árvore rubro-negra. Propriedades, princípios de organização, mecanismo de inserção. Disponível em: <https://javarush.com/pt/groups/posts/pt.4165.rvore-rubro-negra-propriedades-princpios-de-organizao-mecanismo-de-insero>.

9. BACKES, A. ÁRVORE RUBRO-NEGRA Árvore rubro-negra 2. [s.l: s.n.]. Disponível em: <https://www.facom.ufu.br/~backes/gsi011/Aula12-ArvoreRB.pdf>.