Trabalho Final de Estrutura de Dados 2

Professor: Allan Rodrigo Leite
Alunos: Gustavo de Souza; Gustavo Piacentini da Silva; José Augusto Laube; Arthur Lodetti Gonçalves; Rafael Eduardo Gonçalves;

Objetivo principal do trabalho: analisar e comparar a complexidade algorítmica das operações de adição e remoção de nós, considerando o balanceamento, em árvores AVL, rubro-negra e B (com ordens 1, 5 e 10), utilizando conjuntos de chaves aleatórias para avaliar o esforço computacional de cada estrutura.

Trabalho disponível no GitHub em:
https://github.com/Groudon19/Trabalho-EDA2

Metodologia

Os integrantes do grupo se subdividiram e implementaram as árvores conforme instruído pelo professor e a implementação individual de cada árvore pode ser encontrada nas branchs do GitHub. Todos as implementações foram realizadas em C, com ressalva a plotagem dos gráficos feito utilizando a biblioteca `matplotlib` do Python, os testes com os resultados finais foram obtidos pela seguinte metodologia presente no código **main.c**:

1. Define o tamanho máximo dos conjuntos de dados (tam_max), o número de testes por tamanho (num_testes), e o intervalo entre os tamanhos de conjunto analisados (intervalo).
2. Cria dois vetores: um para os valores que serão adicionados à árvore e outro (idêntico) para os valores que serão removidos, ambos os vetores são passados por referência em cada função de adição e remoção das árvores e registram o esforço computacional.
3. Para cada tamanho de conjunto (de 100 a 10.000, em intervalos de 100), executar os experimentos 10 vezes.
4. Em cada experimento:
    a. Cria a árvore e adiciona os valores aleatórios.
    b. Remove os mesmos valores da árvore.
    c. Mede o esforço computacional das operações (armazenado em contadores).
5. Calcula a média do esforço computacional para cada tamanho de conjunto e estrutura de dados.
6. Grava os resultados em dois arquivos CSV: um para a operação de adição e outro para remoção.
7. Os arquivos gerados (`resultados_adicao.csv` e `resultados_remocao.csv`) contém os dados para análise e geração dos gráficos.

Vale ressaltar que o usuário se desejar pode alterar as variáveis no **main.c** para experimentar a implementação com diferentes parâmetros:

```
#define tam_max 10000 // numero maximo de elementos em cada arvore
#define num_testes 10 // numero de vezes que rodará cada conjunto
#define intervalo 100 // tamanho do intervalo para acrescimo da quantidade de elementos
```

Implementações

Árvore AVL**:**

A implementação da árvore AVL neste trabalho foi baseada na implementação do professor disponível parcialmente no Moodle da disciplina, logo, não há muitas ressalvas na implementação, somente a respeito da função remoção() que encontra-se inteiramente comentada no arquivo **avl.c** e seu objetivo é efetuar busca um nó que deseja ser removido e ao encontrá-lo terá dentre suas escolhas os seguintes possíveis casos:

- **Caso 1:** Nó é folha, apenas remove o nó e ajusta o ponteiro do pai.
- **Caso 2:** Nó possui um único filho, substitui o nó pelo seu único filho e ajusta o ponteiro do pai.
- **Caso 3:** Nó possui dois filhos, substitui o nó pelo seu **sucessor em ordem** (menor valor na subárvore direita) e remove o sucessor, que terá no máximo um filho.

Após a remoção é efetuado o balanceamento:

- Inicia a partir do pai do nó removido (ou da nova raiz, se aplicável).
- Atualize a altura de cada nó no caminho para cima.
- Calcula o **fator de balanceamento (FB)** e aplica rotações (simples ou duplas) quando necessário.
- Cada atualização de altura e cada rotação realizada é contabilizada.

Já os contadores para medir o esforço computacional da árvore AVL foram estrategicamente posicionados ao longo das funções da árvore, segue abaixo a localização dos contadores e a justificativa:

Posição 1: Colocado dentro dos laços de busca (adição e remoção) para cada comparação entre chaves ao navegar na árvore.

Motivo: Registra o número de passos necessários para localizar a posição correta de inserção ou o nó a ser removido, além de demonstrar a dependência da operação em relação à altura da árvore, evidenciando a complexidade O(log n).

Posição 2: Dentro das funções de atualização de altura durante o rebalanceamento.

Motivo: Cada atualização de altura de um nó é uma operação essencial para manter as propriedades da AVL. E mede diretamente o impacto do rebalanceamento na complexidade da operação.

Posição 3: Em cada chamada para rotações (simples ou duplas).

Motivo: Rotações são operações custosas e fundamentais para corrigir desequilíbrios na árvore. Tanto rotações simples como duplas foram consideradas como esforço 1.

Posição 4: Nos pontos de decisão que verificam o tipo de nó a ser removido (folha, um filho, dois filhos).

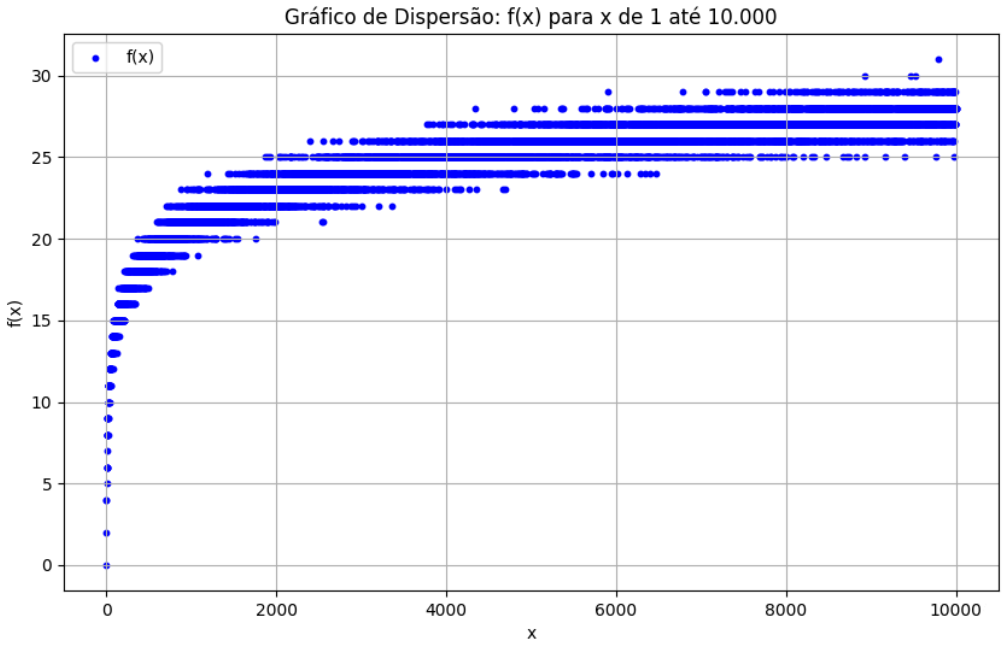
Motivo: Cada decisão exige uma análise das conexões do nó, sendo parte do custo total da operação de remoção e permite identificar quais casos têm maior impacto na complexidade.

Posição 5: Após ajustes de ponteiros dos nós filhos e pais.

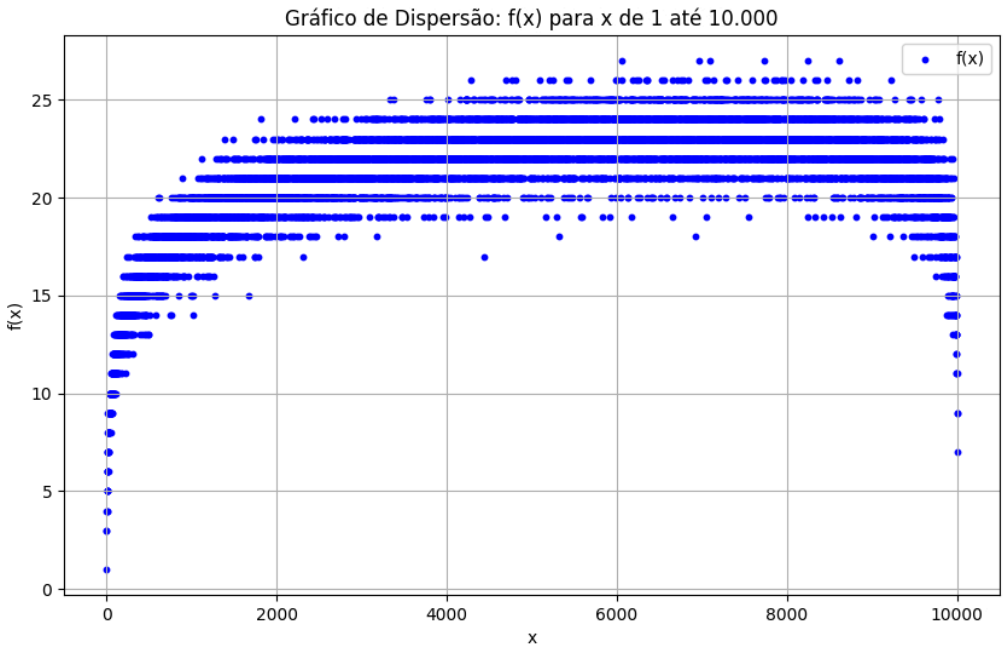
Motivo: Ajustar os ponteiros é uma operação constante, mas relevante para o custo total da operação e ajuda a medir o impacto acumulado dessas operações, especialmente em árvores maiores.

abaixo plotamos dois gráficos, de adição e remoção da árvore AVL respectivamentes, representando o custo para adicionar/remover cada vértice na árvore, no primeiro sendo que x = 1 representa o custo médio para adicionar/remover um nó em uma árvore vazia, e x = 10000 o custo médio para adicionar/remover um nó em uma árvore de tamanho 9999, cara calcular esses custos foi rodado o programa 10 vezes e tirado uma média para adição/remoção de cada x.

Adição:



Remoção:



Árvore Rubro-Negra:

A implementação da Árvore Rubro Negra foi feita com base na que foi disponibilizada pelo professor no moodle, criamos as funções necessárias para fazer a remoção de um elemento da árvore, para criar essas funções nos baseamos no livro “Algoritmos: teoria e prática” do Cormen, que trás em pseudo-código explicações sobre as funções para fazer a remoção em uma árvore Rubro-Negra, encontra-se no arquivo rubNeg.c. A remoção é sempre feita respeitando alguns critérios da árvore Rubro Negra sendo eles:

- Cada nó é vermelho ou preto.
- A raiz é sempre preta.
- Nós vermelhos não podem ter filhos vermelhos (propriedade da "proibição de vermelhos consecutivos").
- Todo caminho de um nó até uma folha nula contém o mesmo número de nós pretos (propriedade do "balanceamento preto").

Nó a ser removido é uma folha:

- Se o nó é **preto**, isso pode causar um desequilíbrio no número de nós pretos nos caminhos.
- Se o nó é **vermelho**, ele pode ser simplesmente removido, pois isso não afeta o número de nós pretos.

Nó a ser removido tem apenas um filho:

- Substituí-lo pelo seu único filho.
- Se o nó removido for preto, o filho deve ser ajustado (recolorido) para preservar o balanceamento preto.

Nó a ser removido tem dois filhos:

- Substituí-lo pelo seu sucessor in-ordem (menor nó maior que o nó a ser removido) ou antecessor in-ordem.
- O sucessor/antecessor será removido recursivamente. Isso reduz o problema para os dois cenários acima (folha ou nó com um filho).

Recoloração:

- Ajustar as cores dos nós afetados. Isso é usado para corrigir violações da propriedade do "balanceamento preto" ou evitar "nós vermelhos consecutivos".

Rotações:

- Realizar rotações simples ou duplas (esquerda ou direita) para reestruturar a árvore e manter o balanceamento.

Já os contadores para analisar o esforço computacional da árvore Rubro Negra foram colocados em posições em que fique justo a comparação entre as árvores, parecido com o que foi feito na árvore AVL :

Posição 1: Durante a adição

Motivo: Quando um nó é adicionado à esquerda ou à direita de um nó existente, o contador é atualizado. Isso mede o esforço necessário para localizar a posição correta para inserir o novo nó, garantindo que a árvore continue ordenada

Posição 2: Rotações

Motivo: As rotações são operações de balanceamento, essenciais para manter as propriedades da Árvore Vermelho-Preto (ou seja, evitar longas cadeias lineares que tornariam a árvore ineficiente). O contador é atualizado durante essas rotações para registrar o custo de reorganizar os nós e corrigir o balanceamento

Posição 3: Durante a remoção

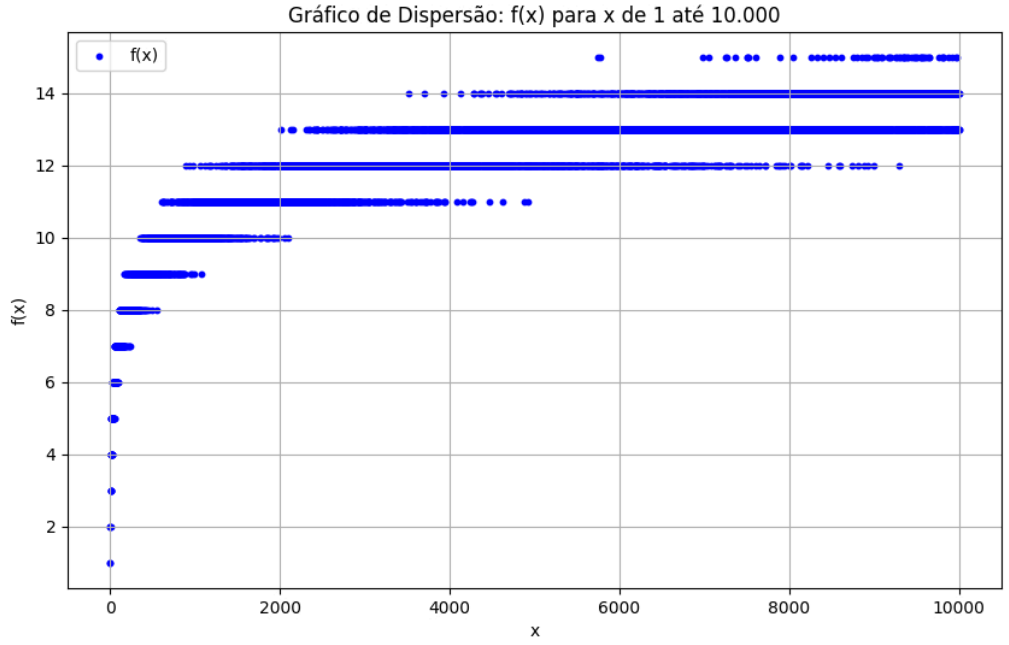
Motivo: Comparações durante a busca de um nó: Quando um nó é removido, o contador é incrementado para cada comparação realizada durante a pesquisa do nó a ser removido.

Durante a remoção, é comum substituir um nó pelo seu sucessor ou por outro nó para manter a estrutura correta da árvore. O contador é usado aqui para registrar essas operações de substituição, pois cada uma envolve reorganizar ponteiros de forma a manter a árvore válida

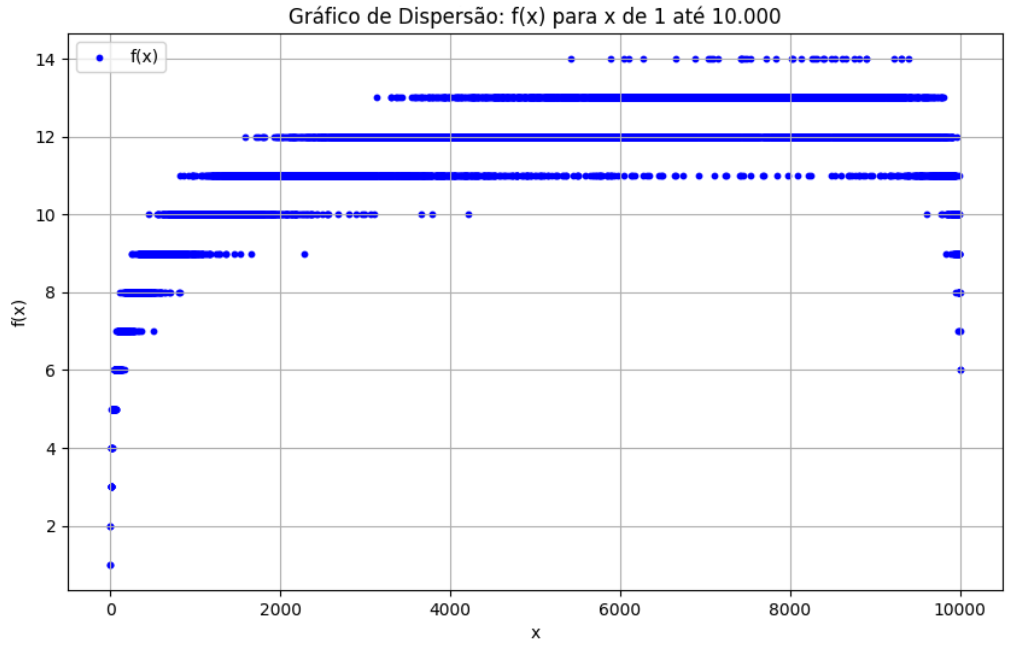
Para corrigir possíveis violações das propriedades da Árvore Vermelho-Preto após uma remoção, podem ser necessárias rotações. O contador mede essas rotações adicionais.

Para ver melhor o comportamento da árvore Rubro Negra, assim como na AVL, também plotamos dois gráficos que representam o custo computacional para a inserção e remoção de cada vértice da árvore, sendo mais fácil de observar o comportamento log(n) dessa forma.

Adição:



Remoção:



Árvore B:

A implementação da árvore B neste trabalho foi baseada nos códigos de adição disponibilizados pelo professor no moodle. Criamos as funções necessárias para fazer a remoção de um elemento da árvore, para criar essas funções nos baseamos no livro “Algoritmos: teoria e prática” do Cormen, que trás em pseudo-código explicações sobre as funções para fazer a remoção em uma árvore B, encontra-se no arquivo arvB.c. A remoção é sempre feita pelos seguintes casos:

- **Caso 1: Chega em um nó folha.**
  - remove a chave caso for encontrado no nó.
- **Caso 2: Chega em um nó que contém a chave que deve ser removida.**
  a) total de chaves do nó que precede a chave desejada possui t (ordem da árvore) chaves;
  - achamos o predecessor na subárvore com raiz nesse nó.

  b) nó que precede a chave possui t-1 chaves e o posterior possui t chaves;

  - achamos o sucessor na subárvore com raiz no nó posterior.

  c) ambos os nós possuem t-1 chaves;

  - fundimos os dois nós e eliminamos a chave do nó após a fusão.

- **Caso 3: Chega em um nó interno que não contém a chave**.

a) se o nó que precede a chave tem t-1 chaves, mas tem um irmão com t chaves;

- movemos uma chave do irmão imediato para o nível acima até o nó e move o ponteiro apropriado do filho do irmão.

b) o nó que precede e os irmãos possuem t-1 chaves;

- fundimos o nó com um dos irmãos.

Já os contadores para analisar o esforço computacional da árvore B foram colocados em posições em que fique justo a comparação entre as árvores.

Posição 1: Adição de Chaves:

Motivo: Durante a inserção de uma chave em um nó, o contador é utilizado para monitorar a quantidade de operações realizadas. Isso inclui dividir nós e redistribuir chaves, caso necessário

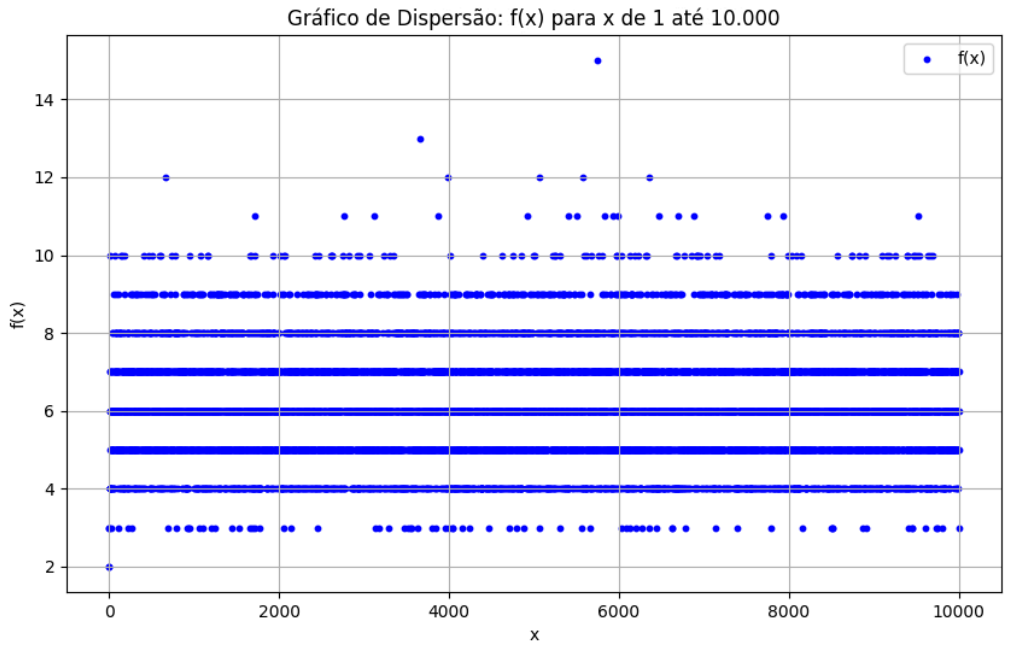
Posição 2: Remoção de Chaves:

Motivo: O contador mede operações como fusão de nós e redistribuição de chaves entre nós irmãos durante a remoção de uma chave, para assegurar que a árvore mantenha suas propriedades

OBSERVAÇÕES: na árvore B, a parte da remoção não está totalmente eficaz, por isso pegamos certos intervalos e fizemos uma projeção para remoção. Em alguns testes, com o tamanho de 10000, obtivemos os seguintes resultados (em média): **ordem 1**: add: 260k rem: 70k, **ordem 5:** add: 160k rem: 84k, **ordem 10:** add: 177k rem: 92k. Esses dados são quando tentamos remover todos os elementos da árvore, que em alguns casos era possível remover todos, e em alguns sobraram elementos na árvore. O gráfico não consegue ser plotado para remoção pois está com segmentation fault, o que indica algum acesso de memória inválido na parte de remoção.

Para a adição da árvore B podemos plotar um gráfico que mostra o custo computacional para adicional cada vértice, e em comparação com as outras árvores podemos observar que ela tem um custo bem menor, porém mais variado, sendo mais difícil de observar o comportamento log(n) por esse gráfico, contudo observamos um custo computacional bem menor.

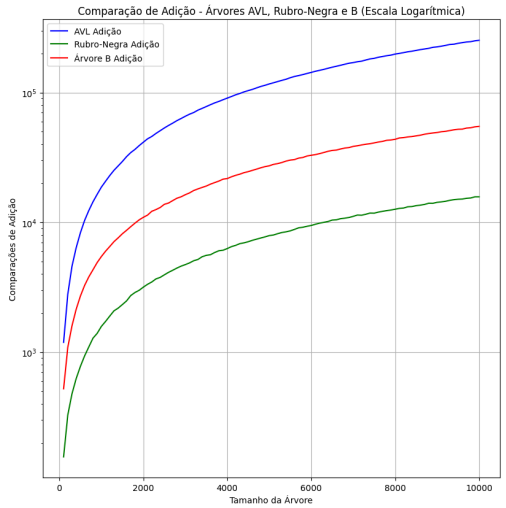
Adição:

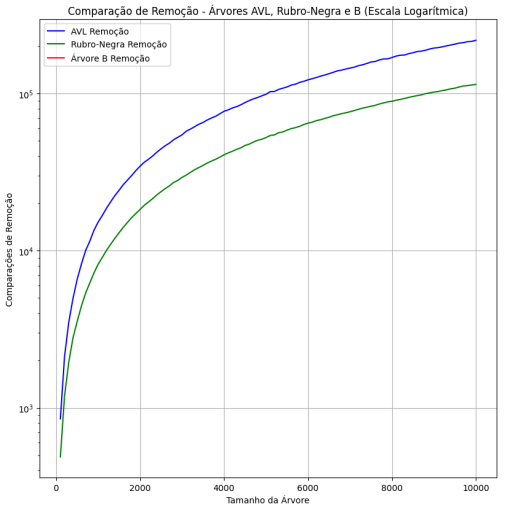


Resultados

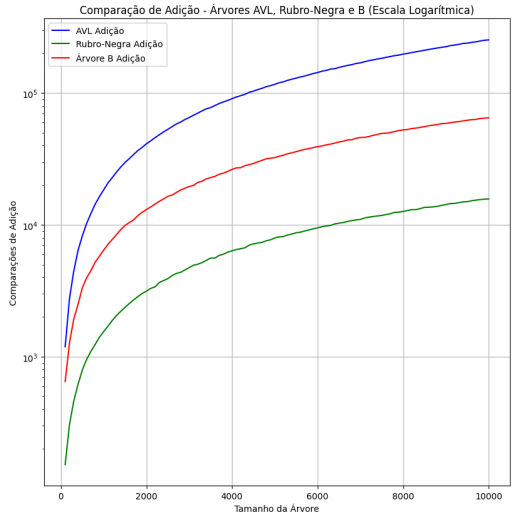
Ao rodar o python que lê o csv com o valor das iterações totais, plotamos os gráficos em escala logarítmica para melhor visualização da complexidade, como foi dito no segmento da implementação da árvore B anteriormente, devido a problemas de segment fault em remoções aleatórias como tentamos testar infelizmente não conseguimos plotar o gráfico da remoção com as árvores B, porém obtivemos os seguintes resultados:

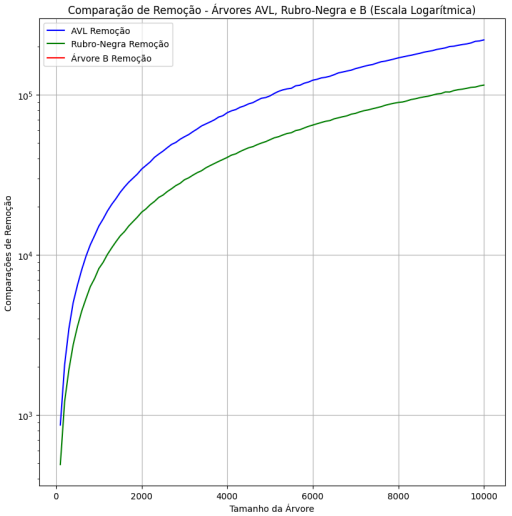
Para arvore B com ordem 1:



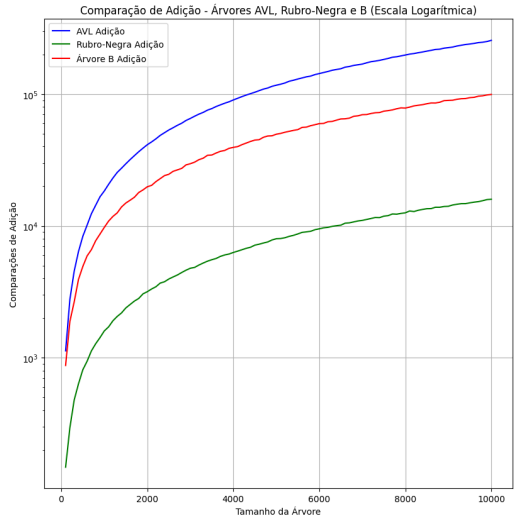


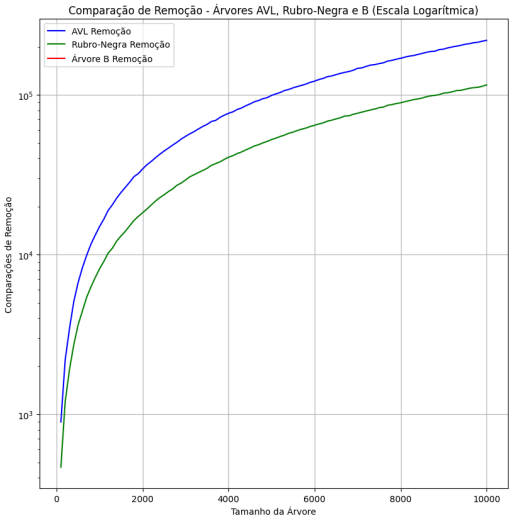
Para arvore B com ordem 5:





Para arvore B com ordem 10:





Análise e Conclusão

Portanto, o nosso trabalho comparou a complexidade das operações de adição e remoção em Árvores B, AVL e Rubro-Negra. Embora a remoção na Árvore B tenha sido implementada e funciona em casos isolados, não fomos capazes de medir um elevado número de elementos sem segment fault, porém os resultados obtidos nas operações de adição e nas análises das árvores AVL e Rubro-Negra foram satisfatórios. Elas apresentaram desempenho similar, com complexidade O(log n) para adição e remoção. A principal diferença entre elas está na abordagem de balanceamento, sendo as AVL mais rigorosas em suas rotações, enquanto as Rubro-Negras utilizam um balanceamento mais simples. A Árvore B, embora não tenha tido sua remoção analisada, mostrou um bom desempenho em adição, também com complexidade O(log n), sendo especialmente eficiente em cenários de grandes volumes de dados. Apesar da limitação na implementação da remoção da Árvore B, os resultados indicam que, em geral, todas as estruturas oferecem boas soluções para manipulação de dados, com a escolha da árvore dependendo das necessidades específicas da aplicação.