Relatório do Trabalho Árvores AVL, Rubro-Negra e B

Alunos: Vitor Cardoso Burgarelli, Karla Bertol, Mateus Pereira Nepomuceno de Carvalho, João Paulo Ersching, Matheus Bertin

1. Visão Geral da Implementação

O experimento foi desenvolvido na linguagem C, estruturado de forma modular para garantir o isolamento lógico de cada estrutura de dados (Árvore AVL, Árvore Rubro-Negra e Árvore B). O arquivo principal trab_eda.c atua como orquestrador dos testes, enquanto os arquivos avl.c, rubro_negra.c e b.c contém as implementações específicas das estruturas.
O fluxo de execução do experimento consiste em:

1. Variar o tamanho da entrada (N) de 1 a 10.000 chaves.
2. Para cada tamanho N, executar 10 amostras independentes (validação estatística).
3. Em cada amostra, gerar um vetor de inteiros aleatórios e medir o esforço de inserção e remoção de todos os elementos.
4. Consolidar a média aritmética dos custos e exportar para um arquivo results.csv.

2. Definição da Métrica de Esforço Computacional

A fundamentação dos resultados apresentados nos gráficos baseia-se em uma métrica de **Custo Acumulado de Operações Elementares**. Ao contrário de uma medição de tempo de CPU (que pode sofrer interferência do sistema operacional), optou-se por instrumentar o código com contadores lógicos (comparacoes).
Esta métrica foi escolhida para capturar os dois aspectos fundamentais da complexidade em árvores balanceadas:

1. **Custo de Navegação:** O esforço para percorrer a árvore (proporcional à altura).
2. **Custo de Estruturação:** O esforço para manter a árvore balanceada após modificações.

A seguir, detalha-se como este contador é incrementado em cada estrutura:

2.1. Árvore AVL (avl.c)

Na AVL, o contador arv->comparacoes representa a soma de comparações de chaves e verificações de balanceamento.

- **Navegação:** Cada vez que o algoritmo compara a chave de busca com o valor de um

nó para decidir descer para a esquerda ou direita, o contador é incrementado.
- **Balanceamento:** O cálculo do Fator de Balanceamento (FB) e a decisão de qual rotação aplicar (simples ou dupla) incrementam o contador.
- **Rotações:** As funções rsd (rotação simples direita) e rse (rotação simples esquerda) incrementam o custo, representando o overhead de realocação de ponteiros.

Justificativa: A AVL é conhecida por ser rígidamente balanceada. Espera-se que este contador mostre um custo de inserção mais elevado (devido às frequentes verificações de altura e rotações) em troca de um custo de busca/remoção otimizado pela menor altura da árvore.

2.2. Árvore Rubro-Negra (rubro_negra.c)

Na Rubro-Negra, o contador reflete a navegação e as correções de cor/estrutura.

- **Navegação:** Incrementado durante a descida na árvore (busca da posição de inserção/remoção).
- **Rebalanceamento (Fix-up):** Loops while que verificam violações das propriedades rubro-negras (ex: pai vermelho) incrementam o contador a cada iteração.
- **Casos de Recolorir vs. Rotacionar:** As verificações condicionais que decidem entre apenas trocar a cor dos nós (recoloração) ou realizar rotações físicas contribuem para o custo total.

Justificativa: A métrica captura a natureza da Rubro-Negra de realizar menos rotações físicas que a AVL, muitas vezes resolvendo conflitos apenas com recoloração, o que deve resultar em um esforço de inserção comparativamente menor em cenários intensos.

2.3. Árvore B (b.c)

Na Árvore B, a métrica foi adaptada para refletir o modelo de custo de árvores multivias (geralmente focadas em acesso a páginas/nós). O contador arvore->comparacoes prioriza **visitas a nós** e **operações estruturais** (split/merge) sobre comparações individuais dentro do vetor de chaves.

- **Visita a Nó (insereNaoCheioB, removerDeNo):** Cada vez que o algoritmo "entra" em um nó (página) para processá-lo, o custo é incrementado. Isso simula o acesso ao bloco de memória/disco.
- **Split (Divisão de Nó):** A função splitB incrementa o contador, penalizando a operação custosa de criar um novo nó e redistribuir chaves quando a capacidade máxima é excedida.
- **Merge e Empréstimo (Remoção):** As funções mergeB (fusão de nós), borrowFromPrev e borrowFromNext incrementam o custo, representando a complexidade de reorganizar chaves entre irmãos ou fundir páginas.

Justificativa: Para as variações de ordem (1, 5, 10), esta métrica evidenciará que árvores de ordem maior (B10) realizam menos splits e possuem menor altura (menos visitas a nós), resultando em um custo acumulado menor em comparação à B de ordem 1 (que se comporta de forma análoga a uma árvore 2-3-4 ou similar, com mais splits).

3. Metodologia do Experimento (Main)

O arquivo trab_eda.c implementa o rigor estatístico exigido:

1. **Geração de Dados (rand):** Utiliza-se a função rand() inicializada com srand(time(NULL)) para garantir que as chaves sejam aleatórias, simulando o **caso médio**. Isso evita vícios de inserção ordenada (pior caso para BSTs simples, embora as árvores balanceadas tratem isso).
2. **Amostragem (Média de 10):**
    - Para cada N, o código executa o loop k=0 até AMOSTRAS (10).
    - A cada iteração, uma *nova* árvore é criada e destruída.
    - Os custos são somados nas variáveis total_ins e total_rem.
    - A média final é calculada como: Total de Operações / Amostras x N.
    - **Nota Importante:** A divisão por N ao final (/ n) normaliza o resultado, apresentando o **custo médio por chave inserida/removida**, e não o custo total do lote.
3. **Parâmetros da Árvore B:**
    - O código instancia a Árvore B com três ordens distintas explicitamente: criaArvoreB(1), criaArvoreB(5) e criaArvoreB(10).
    - Lembrando que na implementação, a capacidade máxima de chaves em um nó é 2 x ordem. Portanto, B1 suporta até 2 chaves, B5 até 10 chaves, e B10 até 20 chaves por nó.
    -

4. Conclusão sobre a Fundamentação

Os gráficos gerados a partir deste código representarão o comportamento assintótico das operações. O eixo Y (Esforço Computacional) não é unitário (segundos), mas adimensional (quantidade de operações elementares). Deste modo, o modo foi aplicado pensando na independência do hardware para a geração dos resultados, tendo em vista que a análise se torna determinística e depende apenas da lógica da estrutura de dados.

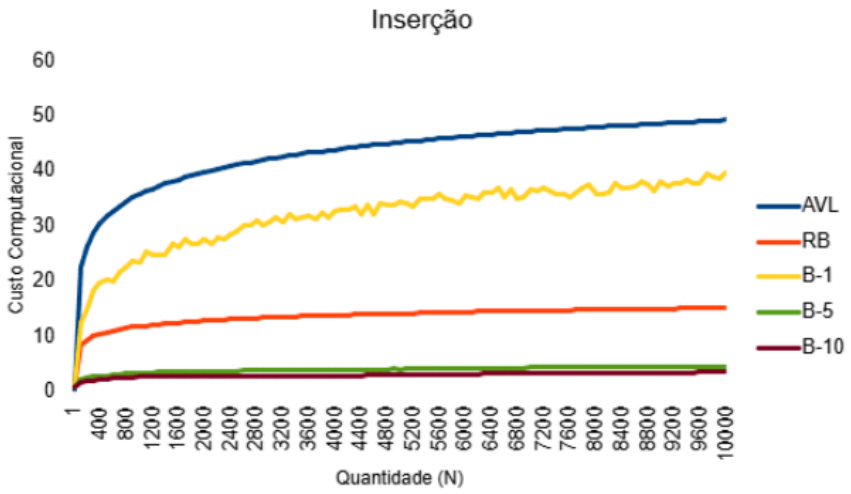
Esta abordagem é preferível em análises de algoritmos pois é determinística e independente do hardware onde o código é executado, dependendo apenas da lógica da estrutura de dados.

5. Análise dos Gráficos de Esforço Computacional

5.1. Desempenho na operação de inserção

O **Gráfico de Inserção** demonstra o esforço computacional médio para adicionar uma chave no conjunto, já incluindo o custo do balanceamento. O esforço é medido como o número de comparações de chaves, normalizado pelo tamanho do conjunto e pelo número de amostras.

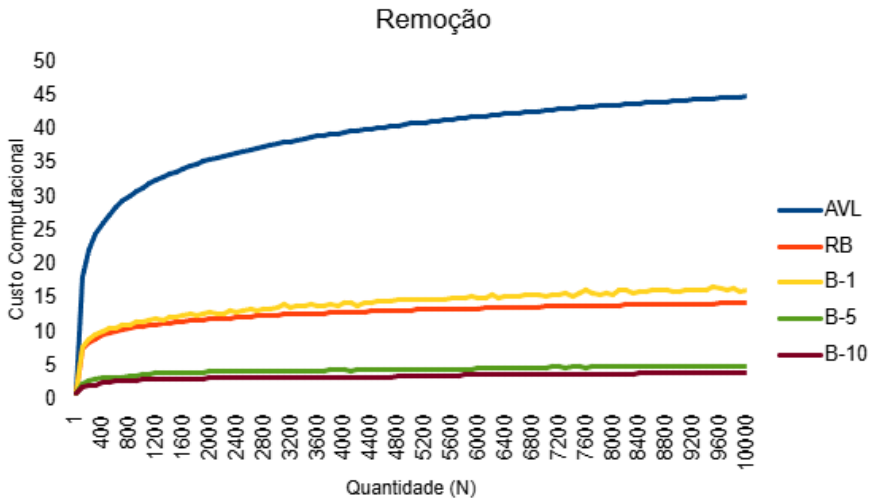
- **Árvore AVL:**
  - Apresenta o **maior custo computacional**, com a curva ascendente alcançando aproximadamente **50 comparações** médias por chave quando N=10.000.
  - Este custo elevado é uma consequência direta do seu **balanceamento mais rigoroso** (Fator de Balanceamento $\in \{-1, 0, 1\}$), que exige mais rotações e verificações de altura a cada inserção.
- **Árvore Rubro-Negra (RB):**
  - Exibe um **custo intermediário**, significativamente inferior ao da AVL, estabilizando em torno de **15 comparações** médias para N =10.000.
  - A sua eficiência é atribuída ao seu mecanismo de balanceamento por cores, que é **menos estrito**, resultando em menos operações de reestruturação por chave inserida.
- **Árvore B (Ordem *t*):**
  - As Árvores B demonstram, em geral, a maior eficiência de custo em termos de comparações, devido à sua baixa altura ($O(\log_t N)$).
  - **B-1 (*t*=1):** Embora seja mais eficiente que AVL, o seu custo é alto dentro das B-Trees (cerca de **40** em N=10.000), pois a baixa ordem aumenta o número de nós visitados, assemelhando-se a uma árvore binária balanceada em blocos.
  - **B-5 (*t*=5) e B-10 (*t*=10):** Apresentam o **menor custo total**, com as curvas praticamente planas e muito baixas (abaixo de **5 comparações** médias). Este desempenho é a principal vantagem da estrutura: a **alta ordem** minimiza drasticamente a altura da árvore e, consequentemente, o número de **visitas aos nós** (custo principal contabilizado) necessário para localizar o ponto de inserção e executar o *Split* (divisão de nó).



5.2 Desempenho na Operação de Remoção (Custo Médio por Chave)

O **Gráfico de Remoção** reflete um padrão de esforço computacional análogo ao da inserção, indicando que a complexidade de manutenção das propriedades de balanceamento após a remoção é consistente com a complexidade de inserção para cada estrutura.

- **Árvore AVL:**
  - Mantém o **custo mais alto**, estabilizando em cerca de **45 comparações** médias por chave removida.
  - O esforço é elevado devido à necessidade de garantir a estrita propriedade de balanceamento após a substituição da chave removida, o que pode desencadear uma série de rotações ao longo do caminho até a raiz.
- **Árvore Rubro-Negra (RB):**
  - Exibe um **custo intermediário**, com cerca de **14 comparações** médias em N=10.000.
  - Assim como na inserção, seu algoritmo de balanceamento de remoção (com transplant e balancear_remocao) é mais eficiente em termos de comparações do que o da AVL.
- **Árvore B (Ordem *t*):**
  - As Árvores B continuam a ser as mais eficientes, com as ordens t=5 e t=10 apresentando o **menor custo**.
  - A **baixa altura** da árvore de alta ordem compensa a complexidade das operações de balanceamento na remoção (como *Fusão* e *Empréstimo* entre nós irmãos), mantendo o número de **visitas aos nós** minimizado.
  - O desempenho superior, com um custo médio abaixo de **5 comparações**, reforça a eficácia da Árvore B para operações em ambientes onde a minimização de acessos a blocos (ou visitas a nós, neste contexto) é crítica.


Remoção
Custo Computacional
50
45
40
35
30
25
20
15
10
5
0
1
400
800
1200
1600
2000
2400
2800
3200
3600
4000
4400
4800
5200
5600
6000
6400
6800
7200
7600
8000
8400
8800
9200
9600
10000
Quantidade (N)
AVL
RB
B-1
B-5
B-10