# Relatório Entrega 1 Árvore Binária

**Emanuel Hoffmann, Gabriel Davel, Guilherme Firme, Ruan Rody e Victor Emerson**

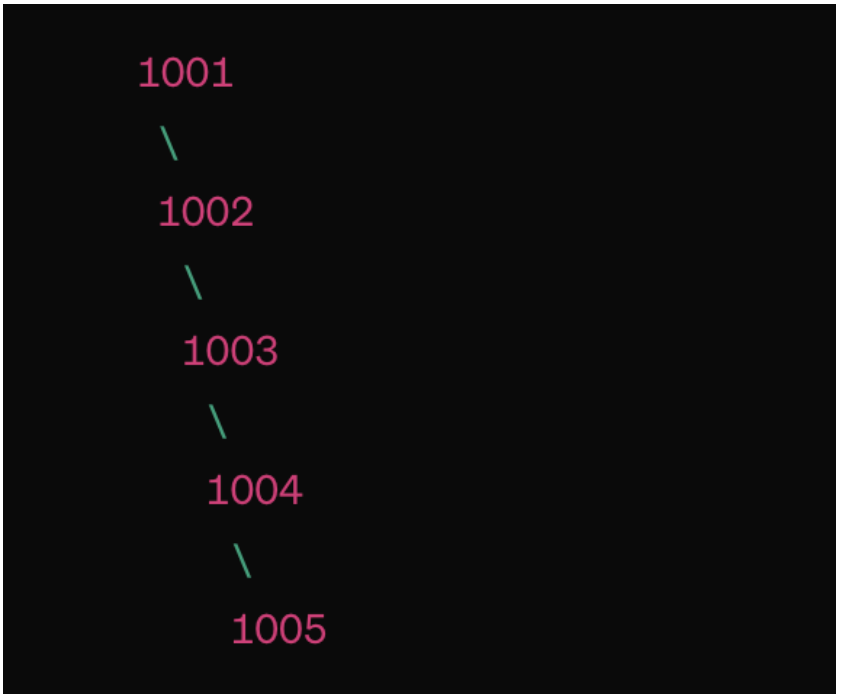
1. **Descreva a topologia das árvores criadas utilizando o método "geraArvoreDegenerada" do gerador de árvores. Utilize um exemplo e desenhe a árvore do exemplo para facilitar a visualização.**

   **R:** Uma árvore degenerada é uma árvore em que cada pai tem apenas um filho. Isso resulta em uma árvore que é essencialmente uma lista ligada. Se o método adicionar adiciona cada novo aluno sempre como um filho direito ou esquerdo do nó mais recentemente adicionado, então a árvore será uma sequência linear de nós.

   **Exemplo:** Suponhamos que matriculaBase seja 1000 e os nomes gerados sejam simplesmente "Aluno 1", "Aluno 2", etc. Para n = 5, o método adicionaria alunos com as seguintes matrículas e nomes:

   Aluno(matricula=1001, nome="Aluno 1")
   Aluno(matricula=1002, nome="Aluno 2")
   Aluno(matricula=1003, nome="Aluno 3")
   Aluno(matricula=1004, nome="Aluno 4")
   Aluno(matricula=1005, nome="Aluno 5")

   Se assumirmos que todos os alunos são adicionados como filhos direitos do último nó adicionado, a árvore teria a seguinte forma:



2. **Considere a árvore de 100 elementos criada utilizando o método "geraArvoreDegenerada" no AppRelatorioArvoreBinaria. No pior caso, quantos nós serão percorridos em uma busca, utilizando o método pesquisar(T valor), nesta árvore? E na árvore de 200 elementos? E na árvore de 1000 elementos? Como você chegou a essas conclusões?**

   **R:** Nesse tipo de árvore, cada novo elemento adicionado se torna um filho do elemento anteriormente adicionado, seguindo sempre o mesmo lado (direito ou esquerdo).

   **Análise do Pior Caso de Busca**
   No método de busca “pesquisar(T valor)” em uma árvore degenerada, o pior caso acontece quando o valor a ser buscado é o que está no nó mais distante da raiz ou quando o valor não está presente na árvore, obrigando a busca a percorrer todos os nós até confirmar a ausência do valor.

   **Número de nós percorridos no pior caso:**
   1. Árvore de 100 elementos: No pior caso, a busca percorreria todos os 100 nós para encontrar o valor ou concluir que ele não está presente.
   2. Árvore de 200 elementos: Similarmente, todos os 200 nós seriam percorridos no pior caso.
   3. Árvore de 1000 elementos: No pior caso, a busca percorreria todos os 1000 nós.

3. **A que conclusão você chega sobre a ordem de complexidade de buscas utilizando o método pesquisar(T valor) em árvores criadas utilizando o método "geraArvoreDegenerada" do gerador de árvores. Explique.**

   **R:** A conclusão vem da natureza da árvore degenerada, que é essencialmente linear. Cada busca começa na raiz e prossegue de nó em nó até encontrar o valor buscado ou chegar ao final da árvore sem encontrá-lo. Em uma árvore balanceada, a busca seria muito mais rápida devido à divisão dos nós em ambos os lados da árvore, mas em uma árvore degenerada, cada busca é linear como numa lista ligada, resultando em uma complexidade de busca de O(n), onde (n) é o número de nós na árvore.

   Essa análise é importante para compreender as limitações do uso de árvores degeneradas em aplicações onde buscas frequentes são necessárias, pois sua eficiência é significativamente menor comparada às árvores balanceadas ou outras estruturas de dados mais apropriadas para buscas rápidas.

4. **Qual a ordem de complexidade do método "geraArvoreDegenerada"? Explique fazendo referência ao código do método. Inclua no relatório uma imagem (print) do código do método e referencie os trechos de código em sua resposta.**

   **R:**

```java
//---Este é o método citado na questão 4 do primeiro relatório
    public void geraArvoreDegenerada(int n, IArvoreBinaria<Aluno> arv){
        //inicio matricula com o valor da constante matriculaBase
        int i,matricula= matriculaBase;
        String nome;
        for(i=1;i<=n;i++){
            //Cada vez que entra a matrícula é incrementada em 1.
            matricula++;
            nome = geraNomeCompleto();
            //Aqui crio um aluno com os dados gerados e o adiciono na árvore.
            arv.adicionar(new Aluno(matricula,nome));
        }
    }
```

1. **Inicialização:** As variáveis “i” e “matricula” são inicializadas uma vez.
2. **Loop For:** O loop executa n vezes, onde n é o número de elementos que serão adicionados à árvore.
3. **Incremento da Matrícula:** O incremento da matrícula é uma operação constante, O(1), que ocorre em cada iteração do loop.
4. **Geração do Nome:** A função geraNomeCompleto() é chamada em cada iteração. A complexidade dessa função não é especificada, mas assumiremos que seja constante para esta análise, O(1).
5. **Criação e Adição de Alunos:** O custo de criar um novo objeto Aluno é constante. Cada novo aluno é adicionado de forma que a árvore se torne uma lista linear, tornando a adição uma operação O(n) no pior caso, pois cada novo nó é adicionado após todos os nós existentes.

A complexidade total do método geraArvoreDegenerada para n elementos é, portanto, O(n**2), pois a soma dos primeiros n inteiros é O(n**2). Isso indica que a complexidade do método é quadrática em relação ao número de elementos inseridos, o que o torna ineficiente para grandes valores de n se a árvore não for mantida balanceada.

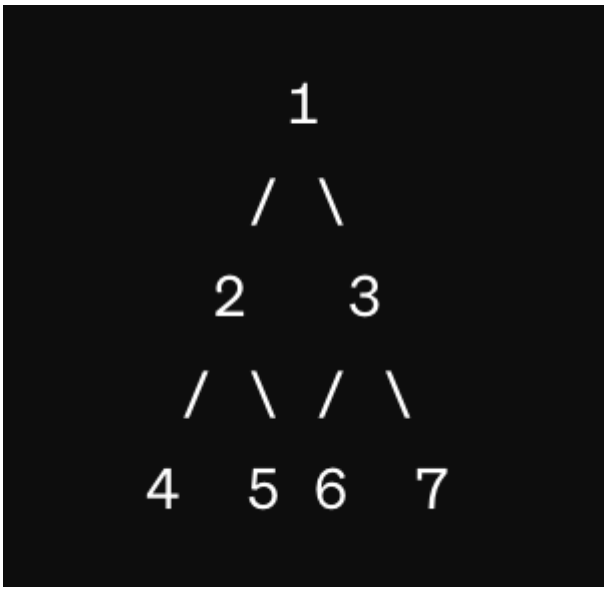
5. **Descreva a topologia das árvores criadas utilizando o método "geraArvorePerfeitamenteBalanceada" do gerador de árvores. Utilize um exemplo e desenha a árvore do exemplo para facilitar a visualização.**

**R:** Uma árvore binária perfeitamente balanceada é uma árvore binária em que cada nível, exceto possivelmente o último, é completamente preenchido, e todos os nós estão o mais à esquerda possível. Isso significa que: Cada nível k da árvore (exceto o último nível) contém 2**k nós. O último nível da árvore tem seus elementos o mais à esquerda possível.

Esta estrutura garante que a árvore é tão compacta quanto possível, minimizando a altura da árvore e maximizando a eficiência das operações de busca, inserção e remoção, que podem ser realizadas em tempo logarítmico, ou seja, O(log n), onde n é o número de elementos na árvore.

**Exemplo:**
Vamos considerar um exemplo onde o método geraArvorePerfeitamenteBalanceada cria uma árvore com 7 elementos. Os elementos serão adicionados de modo que a árvore permaneça perfeitamente balanceada.

Estrutura da árvore com 7 elementos:




6. **Considere a árvore de 100 elementos criada utilizando o método "geraArvorePerfeitamenteBalanceada" no AppRelatorioArvoreBinaria. No pior caso, quantos nós serão percorridos em uma busca, utilizando o método pesquisar(T valor), nesta árvore? E na árvore de 200 elementos? E na árvore de 1000? Como você chegou a essas conclusões?**

**R:** Em uma árvore perfeitamente balanceada, o pior caso para uma busca ocorre quando o valor buscado está nos níveis mais baixos da árvore ou não está presente, obrigando a busca a percorrer da raiz até um nó folha. Assim, o número de nós percorridos no pior caso é igual à altura da árvore.

**Exemplos:**

Árvore de 100 elementos:
Altura h ≈ log2(100+1) ≈ 6.66
Arredondando para cima, temos aproximadamente 7 nós percorridos no pior caso.

Árvore de 200 elementos:
Altura h ≈ log2(200+1) ≈ 7.65
Arredondando para cima, temos aproximadamente 8 nós percorridos no pior caso.

Árvore de 1000 elementos:
Altura h ≈ log2(1000+1) ≈ 9.97
Arredondando para cima, temos aproximadamente 10 nós percorridos no pior caso.

**Conclusão:** A eficiência das buscas em uma árvore binária perfeitamente balanceada é muito alta, pois a altura da árvore cresce apenas logaritmicamente com o número de elementos. Assim, mesmo para grandes quantidades de dados, o número de comparações necessárias em uma busca no pior caso permanece bastante gerenciável. A ordem de complexidade para a busca no pior caso em tais árvores é O(logn), onde n é o número de nós.

7. **A que conclusão você chega sobre a ordem de complexidade de buscas, utilizando o método pesquisar(T valor), em árvores criadas utilizando o método "geraArvorePerfeitamenteBalanceada" do gerador de árvores. Explique.**

   **R:** A ordem de complexidade das buscas utilizando o método pesquisar(T valor) em árvores criadas pelo método geraArvorePerfeitamenteBalanceada é O(logn). Isso ocorre porque a árvore é mantida perfeitamente balanceada, garantindo que o caminho mais longo que pode ser necessário percorrer em qualquer busca é proporcional ao logaritmo do número de elementos na árvore.

8. **Qual a ordem de complexidade do método "geraArvorePerfeitamenteBalanceada"? Explique fazendo referência ao código do método. Inclua no relatório uma imagem (print) do código do método e referencie os trechos de código em sua resposta.**

   **R: Detalhamento do Código:**

```java
//----Este é o método citado na questão 8 do primeiro relatório
public void geraArvorePerfeitamenteBalanceada(int min, int max, IArvoreBinaria<Aluno> arv){
    //Se o valor da menor matrícula for menor ou igual ou maior valor é sinal que ainda preciso inserir elementos na árvore
    //Senão essa recursão acabou...
    if (min <= max){
        //Calculo a matrícula média desta geração e insiro um aluno com essa matrícula na árvore
        int media = (min+max)/2;
        int matricula = matriculaBase+media;
        String nome = geraNomeCompleto();
        arv.adicionar(new Aluno(matricula,nome));
        //Chamo recursivamente para continuar inserindo os elementos com matrículas menores que a média
        geraArvorePerfeitamenteBalanceada(min,media-1,arv);
        //Chamo recursivamente para continuar inserindo os elementos com matrículas maiores que a média
        geraArvorePerfeitamenteBalanceada(media+1,max,arv);
    }
}
```

**Condição de Parada:** A verificação if (min <= max) é uma condição de parada para a recursão. Esta checagem é O(1) pois envolve apenas uma comparação de valores.

**Cálculo da Média e Criação de Alunos:** int media = (min + max) / 2; calcula o índice médio para a matrícula, o que é uma operação de tempo constante O(1). A criação do objeto Aluno e a chamada ao método adicionar são ações significativas. A complexidade de adicionar pode variar dependendo da implementação da árvore, mas aqui assumiremos que é eficiente, tendo uma complexidade aproximada de O(logn) porque a árvore é mantida balanceada.

**Chamadas Recursivas:** O método é chamado recursivamente duas vezes para cada nó: uma para a subárvore esquerda e outra para a subárvore direita, dividindo o problema em metades. Cada chamada recursiva processa metade do intervalo da chamada anterior.

**Complexidade Computacional:** O método divide o conjunto de dados em duas metades a cada chamada recursiva, o que sugere um padrão de divisão e conquista típico de algoritmos com complexidade logarítmica. No entanto, cada nível da árvore envolve uma operação de inserção para cada nó, que é O(logn) para cada inserção devido ao balanceamento da árvore.

O número total de chamadas recursivas está relacionado à altura da árvore, que é O(logn). Em cada nível de recursão, um número total de nós proporcional ao nível é processado. Cada inserção é O(logn), e há “n” inserções no total ao longo de todas as chamadas recursivas.

9. **Há um trecho do AppRelatorioArvoreBinaria em que instanciamos uma árvore indexada por matrícula, utilizando um ComparadorAlunoPorMatricula. Mas, na hora de fazer a busca, como queremos buscar por nome, utilizamos o método pesquisar(T valor, Comparator comparador) passando como argumento para a função um ComparadorAlunoPorNome. Quantos nós serão percorridos nesta busca, no pior caso? Neste caso, qual a ordem de complexidade da busca? Explique.**

**R:** Quando você tem uma árvore binária indexada por um critério (neste caso, matrícula dos alunos) e depois realiza uma busca utilizando um critério diferente (nome do aluno, por exemplo), a eficiência da árvore em termos de operações de busca pode ser significativamente afetada.

**Análise do Processo de Busca:** A árvore foi originalmente construída usando um ComparadorAlunoPorMatricula, o que significa que a árvore está organizada e otimizada para buscas baseadas na matrícula. Isso permite que operações de busca usando matrículas sejam eficientes, geralmente com complexidade O(\log n) em uma árvore balanceada.

No entanto, ao utilizar um ComparadorAlunoPorNome para buscar um aluno pelo nome, a estrutura otimizada da árvore (baseada em matrícula) não oferece nenhum benefício para a ordenação ou localização por nome. Assim, mesmo que você utilize o método pesquisar com um comparador diferente, o processo de busca não pode aproveitar a estrutura de árvore para efetuar uma busca rápida.

**Cenário de Pior Caso:** No pior caso, ao realizar uma busca por nome em uma árvore organizada por matrícula, todos os nós da árvore podem precisar ser verificados para encontrar o aluno desejado ou para concluir que ele não está presente na árvore. Isso ocorre porque a estrutura da árvore, que é baseada nas matrículas, não tem correlação com a ordem dos nomes dos alunos.

**Quantidade de Nós Percorridos:** Em uma árvore com \( n \) elementos, no pior caso, todos os “n” nós seriam percorridos para completar a busca.

**Ordem de Complexidade:** A ordem de complexidade dessa busca, portanto, seria O(n). Em outras palavras, a complexidade é linear com relação ao número de elementos na árvore, porque cada elemento pode potencialmente precisar ser examinado.

**Conclusão:** Utilizar uma árvore binária indexada por um atributo e buscar por outro atributo que não corresponde à estrutura de indexação original resulta em uma busca ineficiente, com a complexidade sendo linear. Este é um bom exemplo de como a escolha de uma estrutura de dados adequada e alinhada com as operações que serão mais frequentemente realizadas é crucial para o desempenho do sistema. Em casos onde múltiplos critérios de busca são comuns, pode ser mais eficaz considerar estruturas de dados alternativas ou manter múltiplas estruturas de dados para cada critério de busca.

10. **Na última parte do AppRelatorioArvoreBinaria geramos uma árvore balanceada de 50000 elementos e imprimimos sua altura e depois geramos uma árvore degenerada de 50000 elementos e imprimimos sua altura. Acontece algum erro? Em que momento? Por quê?**

**R:** Acontece o erro StackOverflowError no momento do cálculo da altura. O método altura é recursivo e calcula a altura da árvore verificando a maior altura entre as sub árvores esquerda e direita de cada nó, adicionando um a cada nível recursivo. Isso é uma operação eficiente para árvores balanceadas, mas para uma árvore degenerada com 50.000 nós, isso pode causar um estouro de pilha (StackOverflowError) devido à profundidade excessiva da recursão. Calcular a altura de uma árvore degenerada de 50.000 elementos usando um método recursivo é impraticável.

**Repositório GitHub :** https://github.com/Vitchao/ArvoreBinaria