IFES SERRA

BACHARELADO DE SISTEMA DE INFORMAÇÃO

FELIPE BECALLI TRINDADE
JOÃO VICTOR FERRAREIS RIBEIRO
PERSEU FERNANDES MACHADO DE OLIVEIRA
RENZO FRAGA LOUREIRO MARINHO

RELATÓRIO SOBRE A IMPLEMENTAÇÃO DA ÁRVORE BINÁRIA DE BUSCA

IFES
SERRA
2023

SUMÁRIO

1. INTRODUÇÃO

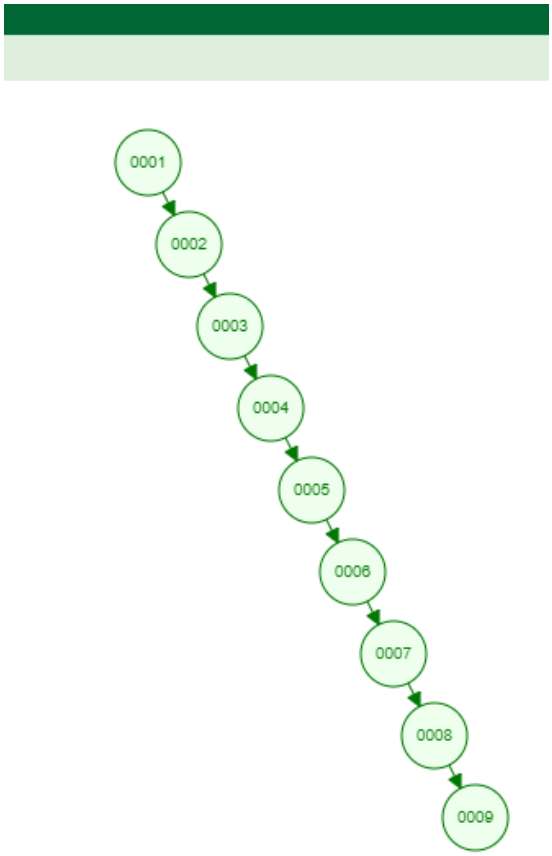
Este relatório tem como objetivo analisar a topologia de árvores binárias criadas utilizando dois métodos distintos: "geraArvoreDegenerada" e "geraArvorePerfeitamenteBalanceada". Além disso, será discutida a complexidade das operações de busca nessas árvores. Para isso, utilizamos o aplicativo "AppRelatorioArvoreBinaria" que gera árvores com diferentes topologias e tamanhos.

Avaliaremos o resultado da execução do programa AppRelatorioArvoreBinaria após baixar o código-fonte e ajustar os imports para utilizar a biblioteca de árvores binárias.

2. DESENVOLVIMENTO

Com isso, perguntas foram levantadas para serem respondidas após a implementação do código funcional, a execução do AppRelatorioArvoreBinaria e os códigos fontes das classes.

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.



A árvore degenerada gera seus alunos com matrículas sequenciais e nomes aleatórios. Logo, durante a inserção dos alunos na árvore pela matrícula, os alunos são alocados sempre à direita de seu pai.

Aqui podemos ver que conforme os alunos forem sendo inseridos, eles serão alocados à direita de seus pais atribuídos pelas matrículas.

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

Considerando o caso base para o nó raiz ao qual já temos acesso inicialmente, não teríamos percorrido nenhum nó. A partir de então, para o próximo nó será necessário avançar um nó e assim por diante até chegar no último nó, portanto, para se chegar ao centésimo nó (100º) será necessário apenas 99 verificações já que a raiz já é nosso ponto inicial. Para as seguintes árvores o processo é o mesmo, logo são percorridos 200 - 1 = 199 nós e 1000 - 1 = 999 nós.

3. **Qual a ordem de complexidade de buscas em árvores criadas utilizando o método "geraArvoreDegenerada" do gerador de árvores. Explique.**

O pior caso da árvore degenerada será uma sequência de N nós tal qual uma lista encadeada. Por causa disso, a ordem de complexidade aumenta linearmente conforme a entrada é aumentada, o que caracteriza O(n).

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.**

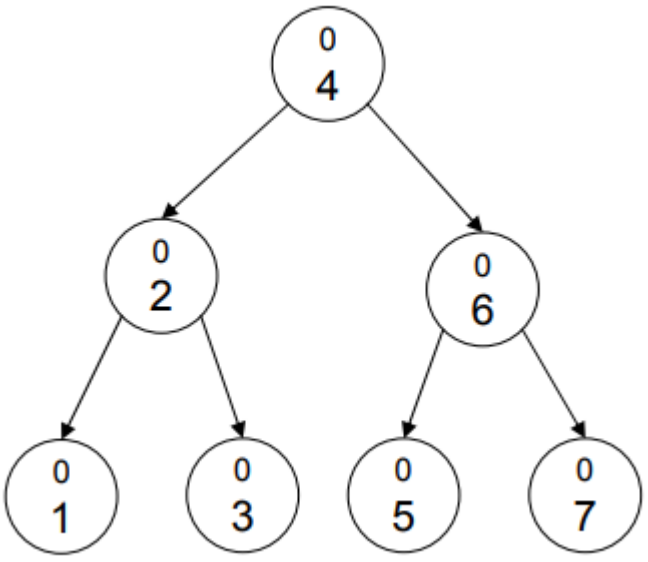
O(n), podemos evidenciar pelo código, que os alunos são inseridos na árvore em um loop for que vai de i1 até iN.

```
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));
    }
}
```

Consequentemente, todos os nós também serão inseridos pela matrícula de m1 até mN que é diretamente proporcional com i. Por consequência, a complexidade do método é O(n), indicando uma relação linear entre o número de elementos a serem inseridos e o tempo de execução do método.

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.

O(n), os elementos que serão inseridos na árvore já são definidos na chamada do método recursivo, através de valores mínimos e máximos que ajudam a definir o escopo dos possíveis valores na árvore e implementar a lógica de balanceamento.
A condição de parada da recursão é enquanto MIN for menor ou igual a MAX. Dentro dessa condição temos um int média que soma os dois e divide por 2 para se obter um valor médio que também soma na matriculaBase, mas o porquê disso?
Após fazer a inserção desse aluno à árvore, é chamado novamente o método com o valor min indo até média-1 e outro indo de média+1 até max.
Essa lógica irá garantir que os valores adicionados na árvore serão sempre balanceados garantindo que a diferença entre a altura das subárvores esquerda e direita seja no máximo 1.



Nesse exemplo, foi criada uma árvore balanceada (1,7).
O primeiro valor a ser adicionado na árvore é o 4 [(1+7) / 2], sendo a raiz.

O próximo elemento é 2[(1+3) / 2] à esquerda de 4, o próximo é 1[(1+1) / 2] à esquerda de 2. E assim sucessivamente 3 [(3+3) / 2] a direita de 2, 6[(5+7) / 2] à direita de 4, 5[(3+7) / 2] à esquerda de 6 e 7[(7+7) / 2] a direita.

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

Suponhamos que estamos buscando um elemento específico, como o valor 5, em uma árvore binária perfeitamente balanceada com os elementos de 1 a 7, conforme mostrado na imagem descrevendo a inserção dos nós. Vamos realizar a busca pelo valor 5:

Começamos na raiz da árvore, que é o número 4. Comparamos o valor que estamos buscando (5) com o valor na raiz (4). 5 é maior que 4, então sabemos que o elemento que estamos procurando deve estar à direita da raiz. Descemos para o próximo nível da árvore, que é o nó com valor 6. Novamente, comparamos o valor 5 com o valor no nó atual (6). 5 é menor que 6, então sabemos que o elemento que estamos procurando deve estar à esquerda deste nó. Descemos para o próximo nível da árvore, que é o nó com valor 5. Comparando novamente, vemos que o valor 5 é igual ao valor no nó atual (5).

Neste exemplo, a busca pelo valor 5 na árvore binária perfeitamente balanceada foi concluída em apenas 3 comparações, pois seguimos o caminho correto na árvore a cada comparação. O motivo pelo qual a busca em uma árvore binária perfeitamente balanceada é considerada O(log n) está relacionado ao fato de que a árvore está dividida de forma equilibrada em cada nível.

O número de vezes que você pode dividir pela metade o espaço de busca (ou seja, o número de níveis que você pode descer na árvore) é governado pelo logaritmo na

base 2 do número de elementos na árvore. Isso ocorre porque, para reduzir pela metade continuamente um número n até que ele seja igual a 1, você precisa fazer aproximadamente $\log_2(n)$ divisões. Portanto, a altura máxima da árvore (o número máximo de comparações necessárias) é $\log_2(n)$.
Logo para uma busca em uma árvore para 100 elementos, o pior caso será para log 2(100) que é 7, para 200, log 2(200) que é 8 e para 1000, log 2(1000) que é 10.
Essas conclusões são derivadas diretamente da complexidade de busca em uma árvore binária perfeitamente balanceada, que é O(log n). À medida que o número de elementos aumenta, o número máximo de comparações necessárias para encontrar um elemento desejado aumenta de forma logarítmica, tornando a busca eficiente mesmo em árvores grandes. Portanto, o número de nós percorridos no pior caso em uma busca é logarítmico em relação ao número de elementos na árvore.

7. Qual a ordem de complexidade de buscas em árvores criadas utilizando o método "geraArvorePerfeitamenteBalanceada" do gerador de árvores. Explique.

Na pergunta anterior vimos como é feita a busca dentro de uma árvore perfeitamente balanceada, assim, a complexidade de busca em uma árvore binária perfeitamente balanceada é O(log n), onde n é o número de elementos na árvore. O logaritmo na base 2 é usado porque, em cada nível da árvore, estamos dividindo o espaço de busca por dois, e o logaritmo na base 2 representa quantas vezes podemos fazer essa divisão até atingirmos o elemento desejado.

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.**

```java
//---Este é o método citado na questão 8 do primeiro relatório
    Felipe Becalli T
    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 matriculas menores que a média
            geraArvorePerfeitamenteBalanceada(min, max: media-1,arv);
            //Chamo recursivamente para continuar inserindo os elementos com matriculas maiores que a média
            geraArvorePerfeitamenteBalanceada( min: media+1,max,arv);
        }
    }
```

A complexidade deste método é O(n), onde n é o número de elementos a serem inseridos na árvore. Isso ocorre porque, em cada chamada recursiva, um único elemento N é inserido na árvore, e o método é chamado uma vez para cada elemento, resultando em um tempo de execução linear em relação ao número de elementos.

9. **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? Porque?**

Quanto à implementação da árvore binária de 50.000 elementos, verificamos que a abordagem não recursiva foi adotada com sucesso para evitar possíveis erros de StackOverflow que poderiam ocorrer na implementação recursiva. Nossa implementação não apresentou erros durante a execução, mesmo ao lidar com uma quantidade considerável de elementos, como os 50.000 mencionados.

É importante destacar que a abordagem funcional escolhida se mostrou eficaz para gerar árvores binárias de grande porte, uma vez que não enfrentamos problemas de estouro de pilha. A implementação não recursiva permitiu que criássemos e manipulássemos árvores com eficiência, garantindo a estabilidade do programa.

Essa decisão reforça a importância de escolher a abordagem correta ao lidar com estruturas de dados como árvores binárias, especialmente em cenários com grandes volumes de dados, onde o gerenciamento de recursos, como a memória da pilha, é crítico para o desempenho e a estabilidade do sistema. Portanto, a escolha da abordagem não recursiva foi essencial para evitar possíveis problemas e garantir a execução bem-sucedida do programa.

3. CONCLUSÃO

Este relatório demonstrou como a topologia de árvores binárias influencia a complexidade das operações de busca. Árvores degeneradas resultam em busca linear (O(n)), enquanto árvores perfeitamente balanceadas proporcionam busca eficiente (O(log n)). Além disso, os métodos de geração de árvores também possuem complexidades distintas, sendo O(n) para ambos.

A escolha da topologia da árvore é crucial para otimizar operações de busca e deve ser considerada de acordo com as necessidades do sistema. Árvores balanceadas são ideais para operações rápidas, enquanto árvores degeneradas devem ser evitadas em cenários onde a eficiência nas buscas é crítica.

LINK PARA O GITHUB DO PROJETO: Kenko2002/BibArvores: Trabalho da disciplina de Tópicos Avançados de Programação onde é feita a implementação de uma biblioteca de Árvores para Java. (github.com)