UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
INSTITUTO METRÓPOLE DIGITAL
CAMPUS NATAL CENTRAL
CURSO DE BACHARELADO EM TECNOLOGIA DA INFORMAÇÃO

RELATÓRIO DO PROJETO DE ESTRUTURAS DE DADOS BÁSICAS II

Esther Maria Da Silveira Wanderley
Thuanny Carvalho Rolim de Albuquerque
Pedro Costa Aragão

Natal-RN
Novembro, 2022

UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
INSTITUTO METRÓPOLE DIGITAL
CAMPUS NATAL CENTRAL
CURSO DE BACHARELADO EM TECNOLOGIA DA INFORMAÇÃO

RELATÓRIO DO PROJETO DE ESTRUTURAS DE DADOS BÁSICAS II

Esther Maria Da Silveira Wanderley
Thuanny Carvalho Rolim de Albuquerque
Pedro Costa Aragão

Relatório sobre o projeto de implementação de uma árvore binária de busca.

Natal-RN
Novembro, 2022

Sumário

1 INTRODUÇÃO 4

2 METODOLOGIA 5
2.1 Classes 5
2.1.1 ABB 5
2.1.1.1 Atributos da classe 5
2.1.2 Gerenciador de Testes 5
2.1.3 Main 5
2.2 Métodos da ABB 5
2.2.1 Métodos públicos 6
2.2.1.1 Construtores 6
2.2.1.2 Inserção 6
2.2.1.3 Remover 6
2.2.1.4 Buscar 7
2.2.1.5 Enésimo elemento 7
2.2.1.6 Posição 7
2.2.1.7 Mediana 8
2.2.1.8 Média 8
2.2.1.9 Árvore cheia 8
2.2.1.10 Árvore completa (nenhum parâmetro) 8
2.2.1.11 Pré-ordem 9
2.2.1.12 Impressor de árvore 9
2.2.2 Métodos privados 9
2.2.2.1 Impressor tipo 1 9
2.2.2.2 String de árvore tipo 2 9
2.2.2.3 Maior altura 9
2.2.2.4 Soma 10
2.2.2.5 Encontrar enésimo elemento 10
2.2.2.6 Busca de nó 10
2.2.2.7 Retorno de nó 10
2.2.2.8 Menor da Subárvore 11
2.2.2.9 Número de Descendentes 11
2.2.2.10 Atualizador de alturas 11
2.2.2.11 Árvore completa (1 parâmetro) 11

3 DESENVOLVIMENTO E RESULTADOS 13

3.1 Enésimo Elemento **13**
3.1.1 Algoritmo 13
3.1.2 Complexidade 13
3.2 Posição **14**
3.2.1 Algoritmo 14
3.2.2 Complexidade 14
3.3 Mediana **14**
3.3.1 Algoritmo 14
3.3.2 Complexidade 14
3.4 Média **15**
3.4.1 Algoritmo 15
3.4.2 Complexidade 15
3.5 É Cheia **15**
3.5.1 Algoritmo 15
3.5.2 Complexidade 16
3.6 É Completa **16**
3.6.1 Algoritmo 16
3.6.2 Complexidade 16
3.7 Pré-Ordem **17**
3.7.1 Algoritmo 17
3.7.2 Complexidade 17
3.8 Impressão de Árvore **18**
3.8.1 Algoritmo 18
3.8.2 Complexidade 18
3.9 Remover **19**
3.9.1 Complexidade 20
3.10 Inserir **22**
3.10.1 Algoritmo 22
3.10.2 Complexidade 23

4 DISCUSSÃO **24**
4.1 Conclusões **24**

1 Introdução

Neste relatório, iremos tratar da implementação de uma árvore binária de busca. Esta árvore possui os seguintes métodos públicos:

- **enesimoElemento()**

  Retorna o n-ésimo elemento (contando a partir de 1) do percurso em ordem simétrica da ABB.

- **posicao()**

  Retorna a posição ocupada pelo elemento x em um percurso em ordem simétrica na ABB (contando a partir de 1).

- **mediana()**

  Retorna o elemento que é a mediana da ABB; caso haja dois elementos centrais, o retorno será o menor deles.

- **media()**

  Retorna a média aritmética dos nós desta árvore.

- **ehCheia()**

  Retorna verdadeiro se a ABB for uma árvore binária cheia e falso caso contrário.

- **ehCompleta()**

  Retorna verdadeiro se a ABB for uma árvore binária completa e falso caso contrário.

- **pre_ordem()**

  Retorna uma String que contém a sequência de visitação (percorrimento) da ABB em pré-ordem.

- **imprimeArvore()**

  Imprime a árvore de duas maneiras diferentes de acordo com o parâmetro passado.

Dessa forma, iremos mostrar o raciocínio utilizado na implementação de cada um dos métodos e a análise de complexidade de cada um deles, além de seus pseudo-códigos.

Toda a implementação deste trabalho foi realizada em JAVA.

2 Metodologia

2.1 Classes

2.1.1 ABB

Esta é a classe com a árvore binária de busca e todos os métodos e atributos que achamos adequados.

2.1.1.1 Atributos da classe

- Altura

  A altura (nível) daquele nó, começando em 0 na raiz.

- Subárvore Esquerda

  Uma ABB com os nós à esquerda deste.

- Subárvore Direita

  Uma ABB com os nós à direita deste.

- Valor

  O valor guardado naquele nó.

2.1.2 Gerenciador de Testes

Cria a árvore a partir da leitura de um arquivo e lê as operações a serem realizadas de outro. Executa cada comando como requisitado.

2.1.3 Main

Cria um objeto do gerenciador de recursos, responsável por rodar os testes pedidos. Inicializa a ABB e realiza as operações, ambos com entradas vindas da leitura de arquivos.

2.2 Métodos da ABB

Nesta seção, descreveremos cada método utilizado na classe da Árvore Binária de Busca (ABB) seguindo o seguinte formato:

Parâmetros: Cada parâmetro do método e sua função.

Retorno: O retorno daquela função e seu significado.

Breve descrição do funcionamento: Como aqueles parâmetros, combinados às informações da classe, geraram o retorno esperado.

2.2.1 Métodos públicos

Os métodos públicos são aqueles solicitados diretamente na descrição do trabalho e outros métodos mais gerais que podem se fazer necessários, como inserção, remoção e busca.

2.2.1.1 Construtores

Para esta classe, fizemos três construtores:

- Construtor default (sem parâmetros)

  Este construtor inicializa o nó com altura -1 (ou seja, o nó não guarda informação nenhuma) e suas subárvores são setadas como nulas.

- Construtor com um valor

  Este construtor inicializa o primeiro nó com o valor pedido e inicializa a altura como 0 e suas subárvores como nulas.

- Construtor com n valores

  Este construtor inicializa o primeiro nó com o valor pedido e inicializa a altura como 0 e suas subárvores como nulas. Depois disso, utiliza o método de inserção para criar novos nós com os outros valores e inseri-los na árvore.

2.2.1.2 Inserção

Parâmetros: Um inteiro
Este inteiro é o valor a ser inserido na árvore.
Retorno: Nenhum
Breve descrição do funcionamento:
Percorre a árvore procurando onde inserir o elemento. Caso encontre um nó vazio (sinalizado pela altura sendo -1), este receberá o valor. Caso contrário, ele buscará na árvore até achar seu respectivo lugar ou até achar o próprio valor na árvore de acordo com as definições da ABB. Caso já esteja, ele imprime que o número já está na árvore. Caso contrário, cria um novo nó no lugar certo da ABB com aquele valor e suas subárvores vazias.

2.2.1.3 Remover

Parâmetros: Um inteiro
Este é o valor que devemos remover da árvore.
Retorno: Nenhum

Breve descrição do funcionamento: Este método trata cada caso de remoção de uma ABB. Caso seja um nó-folha, só o remove a partir de seu pai. Caso seja um nó com exatamente uma subárvore não-vazia, esta subárvore assumirá a posição do nó removido. Por fim, caso ambas as subárvores sejam não-vazias, utiliza o método menorDaSubarvore (descrição nos métodos privados) para encontrar o menor nó da subárvore direita, trocar os dois valores e reduzir este caso a um dos dois casos de remoção anteriores. Quando necessário atualizar as alturas dos descendentes de um nó removido, é chamado o método privado updateHeight.

2.2.1.4 Buscar

Parâmetros: Um inteiro
Este inteiro é o valor a ser buscado na árvore.
Retorno: Um booleano
Este valor é verdadeiro se o valor for achado e falso caso contrário.
Breve descrição do funcionamento: Este método utiliza o método privado retornarNo. Caso o nó retornado não seja nulo, imprime que a chave foi encontrada e retorna verdadeiro. Caso contrário, imprime que a chave não foi encontrada e retorna falso.

2.2.1.5 Enésimo elemento

Parâmetros: Um inteiro

A posição em ordem simétrica cujo elemento queremos encontrar.
Retorno: Um inteiro

O elemento na enésima posição em ordem simétrica.
Breve descrição do funcionamento: Faz uso de algumas arrays: uma com o contador (para a posição atual), uma com cada elemento na enésima posição e a última com booleanos para caso encontremos ou não. Estes são passados para o método privado encontrarEnesimoElemento. O elemento guardado na primeira posição do vetor do elemento na enésima posição é o elemento encontrado, e ele é impresso e retornado.

2.2.1.6 Posição

Parâmetros: Um inteiro
Este é o elemento cuja posição na árvore devemos encontrar.
Retorno: Um inteiro
Esta é a posição ocupada pelo elemento recebido.
Breve descrição do funcionamento: Este faz uso de dois "ponteiros": um booleano para caso o elemento tenha sido encontrado ou não (falso por default) e um inteiro que conta em que

elemento estamos. Estes são passados para o método privado encontrarPosicao. O número no contador é impresso e retornado.

2.2.1.7 Mediana

Parâmetros: Nenhum

Retorno: Nenhum

Breve descrição do funcionamento: Imprime o elemento do meio da árvore (o número em ordem simétrica na posição do número de descendentes mais um, totalizando o número de nós da árvore, dividido por dois, uma vez que a ordem simétrica de uma ABB é ordenada).

2.2.1.8 Média

Parâmetros: Nenhum

Retorno: Um número em ponto flutuante de precisão dupla

Este é o valor da média aritmética dos nós da árvore.

Breve descrição do funcionamento: Este faz uso de dois "ponteiros": um com a soma dos nós até então e outro que conta o número de nós. Estes são enviados para o método privado sum. A divisão do conteúdo do ponteiro com a soma pelo do ponteiro com a contagem gerará a média aritmética, que será impressa e retornada.

2.2.1.9 Árvore cheia

Parâmetros: Nenhum

Retorno: Um booleano

Retorna verdadeiro caso a árvore seja cheia, falso caso contrário.

Breve descrição do funcionamento: Este método utiliza uma propriedade das árvores binárias cheias: ABB cheia $\iff n = 2^h - 1$. Ele utiliza os métodos privados numeroDescendentes para descobrir quantos nós tem na árvore e o maiorAltura para descobrir a altura da árvore. Utilizando a bi-implicação acima, descobrimos se a ABB é cheia fazendo a comparação do número de nós com a altura. O retorno é descrito acima.

2.2.1.10 Árvore completa (nenhum parâmetro)

Parâmetros: Nenhum

Retorno: Um booleano

Retorna verdadeiro caso a árvore seja completa, falso caso contrário.

Breve descrição do funcionamento: Primeiro, ele calcula a altura da árvore e o usa em uma sobrecarga privada deste método que recebe a altura da árvore como parâmetro e retorna se ela é ou não completa.

2.2.1.11 Pré-ordem

Parâmetros: Nenhum

Retorno: Uma string

Esta string contém cada elemento da árvore em pré-ordem.

Breve descrição do funcionamento: Este método implementa a versão iterativa da pré-ordem, imprimindo cada elemento e colocando-os na string em pré-ordem.

2.2.1.12 Impressor de árvore

Parâmetros: Um inteiro

Um "seletor" de tipo de impressão.

Retorno: Nenhum

Breve descrição do funcionamento:

Caso o seletor seja 1, imprime do jeito 1 utilizando o método privado imprimeArvore1; caso seja 2, imprime a string retornada pelo método privado stringArvore2, que gera a string no formato a ser impresso.

2.2.2 Métodos privados

Os métodos privados são métodos utilizados para auxiliar na execução dos métodos públicos, requisitados no trabalho

2.2.2.1 Impressor tipo 1

Parâmetros: Nenhum

Retorno: Nenhum

Breve descrição do funcionamento: Imprime todos os nós válidos no formato especificado, utilizando a altura como base para o número de espaços/traços antes e depois do número. É uma função recursiva que imprime primeiro os nós à esquerda e depois os à direita (pré-ordem).

2.2.2.2 String de árvore tipo 2

Parâmetros: Nenhum

Retorno: Uma string

A string contém os elementos como especificado na descrição.

Breve descrição do funcionamento: Para cada elemento adicionado na string, é colocado um parêntese abrindo e o valor deste nó, seguido dos filhos à esquerda e os filhos à direita (pré-ordem) para só então fechar o parêntese.

2.2.2.3 Maior altura

Parâmetros: Uma ABB

A ABB de que queremos a maior altura possível

Retorno: Um inteiro

A altura da árvore

Breve descrição do funcionamento: Guarda os maiores valores de altura existentes: seja o deste nó ou de seus possíveis filhos até que não haja mais filhos. Então, a maior altura é retornada, sendo esta a altura da árvore.

2.2.2.4 Soma

Parâmetros: Dois ponteiros e uma ABB

O primeiro ponteiro guardará a soma dos valores, o segundo a contagem de nós e a ABB será o nó em que estamos procurando.

Retorno: Nenhum

Breve descrição do funcionamento: Incrementa o conteúdo do ponteiro de soma com o valor daquele nó e incrementa em um o ponteiro contador para cada nó. Depois, segue a pesquisa na subárvore esquerda e na subárvore direita deste nó até que todos tenham sido percorridos.

2.2.2.5 Encontrar enésimo elemento

Parâmetros: Um inteiro, dois ponteiros para inteiro, um ponteiro para booleano e uma ABB.

O inteiro é a posição que estamos procurando, um ponteiro para inteiro é um contador de posições, o outro é o enésimo elemento, o ponteiro para booleano guarda se encontramos ou não o enésimo elemento.

Retorno: Nenhum

Breve descrição do funcionamento: Se ainda não tivermos encontrado o valor, procuramos na subárvore esquerda e vamos incrementando o contador. Se chegarmos na posição n, o booleano passa a ser verdadeiro e o ponteiro com o enésimo elemento guarda seu valor. Se não chegarmos, seguimos a busca recursiva na subárvore direita.

2.2.2.6 Busca de nó

Parâmetros: Um inteiro e uma ABB

O inteiro é o valor que estamos procurando e a ABB é o nó onde estará o valor se encontrado.

Retorno: Nenhum

Breve descrição do funcionamento: Procura na ordem da árvore pelo valor requisitado. Quando encontrado, este valor é guardado na ABB recebida.

2.2.2.7 Retorno de nó

Parâmetros: Um inteiro

O valor que devemos buscar.

Retorno: Uma ABB

Esta é a ABB cujo valor é aquele que procuramos.

Breve descrição do funcionamento: Cria uma ABB que guardará o valor requisitado. Se o valor é encontrado, esse nó é retornado; caso contrário, é retornado nulo.

2.2.2.8 Menor da Subárvore

Parâmetros: Uma ABB

A ABB em que estamos procurando.

Retorno: Um inteiro

O menor valor presente naquela ABB.

Breve descrição do funcionamento: Enquanto houver nós à esquerda (valores menores), nós retornamos o retorno da mesma pesquisa na subárvore esquerda. Quando chegarmos ao último à esquerda, retornamos seu valor.

2.2.2.9 Número de Descendentes

Parâmetros: Uma ABB

O nó em que estamos procurando o número de descendentes.

Retorno: Um inteiro

O número de descendentes daquele nó.

Breve descrição do funcionamento: Seta, inicialmente, o número de descendentes para 0. Caso haja filhos à sua esquerda, soma ao número de descendentes deste nó o número dos descendentes de seu filho da esquerda mais um (representando o nó imediatamente à sua esquerda). Repete o processo para os filhos da direita. Caso não haja nenhum filho, retorna o 0 original do número de descendentes. Caso haja, vai retornando este valor.

2.2.2.10 Atualizador de alturas

Parâmetros: Um inteiro

A altura que deve ser colocada nos filhos a seguir

Retorno: Nenhum

Breve descrição do funcionamento: Atualiza a altura deste nó e, em seus filhos, atualiza com a nova altura deste nó mais um.

2.2.2.11 Árvore completa (1 parâmetro)

Parâmetros: Um inteiro

A altura da árvore.

Retorno: Um booleano

Se a árvore é completa, retorna verdadeiro; retorna falso caso contrário.

Breve descrição do funcionamento: Recebe a maior altura e compara, quando o nó tem ao menos uma subárvore vazia, se ele está no penúltimo ou último nível. Se não estiver, retorna falso. Se estiver, retorna verdadeiro.

3 Desenvolvimento e resultados

Agora, iremos analisar a complexidade de cada um dos métodos.

3.1 Enésimo Elemento

3.1.1 Algoritmo

enesimoElemento():

Algorithm 1: enesimoElemento

Input: um inteiro n.

Result: o enésimo elemento do percurso em ordem simétrica

$Integer[\,]count \leftarrow 0$;

$Integer[\,]elementoN \leftarrow 0$;

$encontrarEnesimoElemento(n, count, elementoN, encontrou, this)$;

$print(elementoN[0])$;

$return\ elementoN[0]$;

3.1.2 Complexidade

Analisando a complexidade, segue:

$$
\begin{aligned}
T(n) &= 1 + 1 + 6 + 2T(\frac{n}{2}) + 1 + 0 && \text{(Complexidade Local de encontrarEnesimoElemento)} \\
&= 3 + O(n) && \text{(Complexidade assintótica de encontrarEnesimoElemento)} \\
&\equiv O(n)
\end{aligned}
$$

3.2 Posição

3.2.1 Algoritmo

posicao():

Algorithm 2: posicao

Input: um inteiro x.

Result: a posição ocupada pelo elemento x em um percurso em ordem simétrica na ABB (contando a partir de 1).

$Integer[\,]count \leftarrow 0$;

$Boolean[\,]encontrou \leftarrow false$;

$encontrarPosicao(x, count, encontrou, this)$;

$print(count[0])$;

$return\ count[0]$;

3.2.2 Complexidade

Analisando a complexidade, segue:

$$\begin{aligned} T(n) &= 1 + 1 + O(n) + 1 \qquad \text{(Complexidade assintótica de encontrarPosicao)} \\ &= 3 + O(n) \\ &\equiv O(n) \end{aligned}$$

3.3 Mediana

3.3.1 Algoritmo

mediana():

Algorithm 3: mediana

Input: void

Result: o elemento que contém a mediana da ABB.

$print(enesimoElemento((numeroDescendentes(this) + 1)/2))$;

3.3.2 Complexidade

Analisando a complexidade:

$$\begin{aligned} T(n) &= 1 + O(n) \\ &\equiv O(n); \end{aligned}$$

3.4 Média

3.4.1 Algoritmo

media():

Algorithm 4: media

Input: um inteiro x.

Result: a média aritmética dos nós da árvore que x é raiz.

$Double[\,]sum \leftarrow 0.0$;

$Double[\,]count \leftarrow 0.0$;

$ABBabbEncontrada \leftarrow retornarNo(x)$;

$sum(sum, count, abbEncontrada)$;

$print(sum[0]/count[0])$;

$return\ sum[0]/count[0]$;

3.4.2 Complexidade

Analisando a complexidade, segue:

$$\begin{aligned} T(n) &= 1 + 1 + 1 + O(n) + O(n) + 1 \\ &\equiv O(n) \end{aligned}$$

3.5 É Cheia

3.5.1 Algoritmo

ehCheia():

Algorithm 5: ehCheia

Input: Um arranjo qualquer $A = \{A_0, A_1, ..., A_{n-1}\}$

Result: retorna verdadeiro se a ABB for uma árvore binária cheia e falso, caso contrário

if $this.numeroDescendentes(this) + 1 = 2^h + 1$ **then**

  $return\ true$;

end

if $this.numeroDescendentes(this) + 1 = 2^h + 1$ **then**

  $return\ true$

else

  $return\ false$

end

3.5.2 Complexidade

Analisando a complexidade, segue:

$$\begin{aligned} T(n) &= 1 + O(n) + 1 + O(n) + 1 \\ &\equiv O(n) \end{aligned}$$

3.6 É Completa

3.6.1 Algoritmo

ehCompleta():

Algorithm 6: ehCompleta

Input: void

Result: retorna verdadeiro se a ABB for uma árvore binária completa.

$int\ tamanho \leftarrow maiorAltura(this)$;

if $this.subarvoreEsquerda \neq \lambda$ **then**

  $return\ this.subarvoreEsquerda.ehCompleta(tamanho)$;

end

if $this.subarvoreDireita \neq \lambda$ **then**

  $return\ this.subarvoreDireita.ehCompleta(tamanho)$;

end

if $altura < tamanho$ **then**

  $return\ false$;

end

3.6.2 Complexidade

Analisando a complexidade, segue que:

$$\begin{aligned} T(n) &= 1 + O(n) + 1 + T(\frac{n}{2}) + 1 + T(\frac{n}{2}) + 1 \\ &= \left[2T(\frac{n}{2}) + 3\right] + O(n) \\ &= O(n) + O(n) \\ &\equiv O(n) \end{aligned}$$

3.7 Pré-Ordem

3.7.1 Algoritmo

pre_ordem():

Algorithm 7: pre_ordem

Input: void

Result: retorna uma String que contém a sequência de visitação (percorrimento) da ABB em pré-ordem.

```
Stack values;
String order ←" " ;
values.push(this) ;
while !values.empty() do
    ABB top = (ABB)values.pop() ;
    print(top.valor) ;
    if top.subarvoreDireita! = λ then
        values.push(top.subarvoreDireita) ;
    end
    if top.subarvoreEsquerda! = λ then
        values.push(top.subarvoreEsquerda) ;
    end
end
return order ;
```

3.7.2 Complexidade

Analisando a complexidade, segue que:

$$
\begin{aligned}
T(n) &= 1 + 1 + 1 + n \cdot (1 + 1 + 1 + 1 + 1) && \text{(No pior caso, o laço é verdadeiro} \\
&= 3 + n \cdot 5 && \text{para todos os elementos da árvore)} \\
&\equiv O(n)
\end{aligned}
$$

3.8 Impressão de Árvore

3.8.1 Algoritmo

imprimeArvore1():

Algorithm 8: imprimeArvore1()

Input: void

Result: imprime a árvore de duas maneiras diferentes.

if $altura \neq -1$ **then**
  for $i \leftarrow 0$ *até* $altura \cdot 5$ **do**
    $print(' ')$;
  end
  $print(valor)$;
  for $i \leftarrow 0$ *até* $30 - altura \cdot 5$ **do**
    $print('-')$;
  end
  $print(' ')$;
end
if $this.subarvoreEsquerda \neq \lambda$ **then**
  $return\ this.subarvoreEsquerda.imprimeArvore1()$;
end
if $this.subarvoreDireita \neq \lambda$ **then**
  $return\ this.subarvoreDireita.imprimeArvore1()$;
end

3.8.2 Complexidade

Analisando sua complexidade, teremos:

$$
\begin{aligned}
T(n) &= 1 + 30 \cdot (1+0) + 1 + T\left(\frac{n}{2}\right) + 1 + T\left(\frac{n}{2}\right) \qquad \text{(Os laços somados terão peso 30)} \\
&= 3 + 30 + 2T\left(\frac{n}{2}\right) \\
&= 33 + 2T\left(\frac{n}{2}\right) \\
&= 2^i \cdot T\left(\frac{n}{2^i}\right) + \sum_{j=0}^{j=i} 33 \cdot 2^j \\
&= 2^{\log_2 n} \cdot T\left(\frac{n}{2^{\log_2 n}}\right) + 33 \cdot \sum_{j=0}^{j=\log_2 n} 2^j \\
&= n \cdot 33 + 33 \cdot 2 \cdot \frac{1 - 2^{\log_2 n}}{1-2} \\
&= 33n + 66 \cdot (-1 + n) \\
&= 33n + 66n - 66 \\
&= 99n - 66 \\
&\equiv O(n)
\end{aligned}
$$

3.9 Remover

Especificamente para este método, devido à sua extensão, ao invés do pseudocódigo, colocaremos aqui as imagens do código em si.

```
public void remove(int x){
    if(altura == -1 && subarvoreDireita == null && subarvoreEsquerda == null){ //negative height = empty tree
        System.out.println(x + " não está na árvore, não pode ser removido");
        return;
    } else if (altura == -1 && subarvoreEsquerda != null) {
        subarvoreEsquerda.remove(x);
    } else if (altura == -1 && subarvoreDireita != null) {
        subarvoreDireita.remove(x);
    }
    if(x<valor){
        if(subarvoreEsquerda != null){
            if(subarvoreEsquerda.valor == x) {
                if(subarvoreEsquerda.subarvoreEsquerda == null && subarvoreEsquerda.subarvoreDireita == null) subarvoreEsquerda = null;
                else if(subarvoreEsquerda.subarvoreEsquerda == null && subarvoreEsquerda.subarvoreDireita != null){
                    subarvoreEsquerda.subarvoreDireita.updateHeight(subarvoreEsquerda.altura);
                    subarvoreEsquerda = subarvoreEsquerda.subarvoreDireita;

                }
                else if(subarvoreEsquerda.subarvoreEsquerda != null && subarvoreEsquerda.subarvoreDireita == null){
                    subarvoreEsquerda.subarvoreEsquerda.updateHeight(subarvoreEsquerda.altura);
                    subarvoreEsquerda = subarvoreEsquerda.subarvoreEsquerda;
                }
                else{
                    int min = menorDaSubarvore(subarvoreEsquerda.subarvoreDireita);
                    subarvoreEsquerda.valor = min;
                    subarvoreEsquerda.subarvoreDireita.remove(min);
                }
            }
            else subarvoreEsquerda.remove(x);
        }
        else System.out.println(x + " não está na árvore, não pode ser removido");
    }
```

```
else if(x>valor){
    if(subarvoreDireita != null){
        if(subarvoreDireita.valor == x){
            if(subarvoreDireita.subarvoreEsquerda == null && subarvoreDireita.subarvoreDireita == null) subarvoreDireita = null;
            else if(subarvoreDireita.subarvoreEsquerda == null && subarvoreDireita.subarvoreDireita != null){
                subarvoreDireita.subarvoreDireita.updateHeight(subarvoreDireita.altura);
                subarvoreDireita = subarvoreDireita.subarvoreDireita;
            }
            else if(subarvoreDireita.subarvoreEsquerda != null && subarvoreDireita.subarvoreDireita == null){
                subarvoreDireita.subarvoreEsquerda.updateHeight(subarvoreDireita.altura);
                subarvoreDireita = subarvoreDireita.subarvoreEsquerda;
            }
            else{
                int min = menorDaSubarvore(subarvoreDireita.subarvoreDireita);
                subarvoreDireita.valor = min;
                subarvoreDireita.subarvoreDireita.remove(min);
            }
        }
        else subarvoreDireita.remove(x);
    }
    else System.out.println(x + " não está na árvore, não pode ser removido");
```

```
    } else {
        //delete root
        valor = 0;
        if(subarvoreEsquerda == null && subarvoreDireita == null) altura = -1;
        else if(subarvoreEsquerda == null && subarvoreDireita != null){
            subarvoreDireita.updateHeight( i: 0);
            altura = -1;
        }
        else if(subarvoreEsquerda != null && subarvoreDireita == null){
            subarvoreEsquerda.updateHeight( i: 0);
            altura = -1;
        }
        else{
            int min = menorDaSubarvore(subarvoreDireita);
            valor = min;
            subarvoreDireita.remove(min);
        }
    }
}
```

3.9.1 Complexidade

Analisando a complexidade, teremos:

$$\begin{aligned} T(n) &= 3 + 6 + 1 + T\left(\frac{n}{2}\right) \\ &= 10 + T\left(\frac{n}{2}\right) \\ &\equiv O(n) \end{aligned}$$

3.10 Inserir

3.10.1 Algoritmo

Algorithm 9: Insert

Input: um inteiro x

Result: void

if $altura = -1$ *and* $subarvoreDireita = \lambda$ *and* $subarvoreEsquerda = \lambda$) **then**
    $this.valor \leftarrow x$;
    $altura \leftarrow 0$;
    $return$;
else
    if $altura = -1$ *and* $subarvoreEsquerda \neq \lambda$ **then**
        $subarvoreEsquerda.insert(x)$;
    end
    if $altura = -1$ *and* $subarvoreDireita \neq \lambda$ **then**
        $subarvoreDireita.insert(x)$;
    end
end
if $x < valor$ **then**
    if $subarvoreEsquerda = \lambda$ **then**
        $subarvoreEsquerda \leftarrow ABB(x)$;
        $subarvoreEsquerda.altura \leftarrow this.altura + 1$;
        $print(x + \text{“}adicionado\text{”})$;
    else
        $subarvoreEsquerda.insert(x)$;
    end
else
    if $x > valor$ **then**
        if $subarvoreDireita = \lambda$ **then**
            $subarvoreDireita \leftarrow ABB(x)$;
            $subarvoreDireita.altura \leftarrow this.altura + 1$;
            $print(x + \text{“}adicionado\text{”})$;
        else
            $subarvoreDireita.insert(x)$;
        end
    else
        $print(x + \text{“}jestnarvore, nopodeserinserido\text{”})$;
    end
end

3.10.2 Complexidade

Sua complexidade será dada por:

$$\begin{aligned} T(n) &= 3 + 2 + T\left(\frac{n}{2}\right) \\ &= 5 + T\left(\frac{n}{2}\right) \\ &\equiv O(n) \end{aligned}$$

4 Discussão

4.1 Conclusões

Portanto, podemos ver que todos os métodos foram implementados, além de alguns outros auxiliares que não foram explicitamente pedidos mas foram úteis na elaboração dos requisitados. Como explicitado no desenvolvimento, todos os métodos pedidos têm complexidade assintótica linear, uma vez que a ABB feita não necessariamente será balanceada. Além disso, as duas classes auxiliares foram utilizadas para se testar os arquivos requeridos pelo professor e chamar o método correto a cada comando.