UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
INSTITUTO METRÓPOLE DIGITAL
BACHARELADO EM TECNOLOGIA DA INFORMAÇÃO

RELATÓRIO DE IMPLEMENTAÇÃO DE UMA ÁRVORE BINÁRIA DE BUSCA

Aluno: Caio Vitor Dantas Machado
Professor: Sidemar Fideles Cezario

Natal, RN, 13 de Novembro de 2023.

Sumário

Introdução

O trabalho desenvolvido e descrito neste relatório consiste na implementação de uma árvore binária de busca. Tendo como funções padrões a inserção, remoção e busca de nós. Além dessas operações básicas, também foram implementados os seguintes métodos.

1. int **enesimoElemento** (int n): retorna o n-ésimo elemento (contando a partir de 1) do percurso em ordem (ordem simétrica) da ABB.

2. int **posicao** (int x): retorna a posição ocupada pelo elemento x em um percurso em ordem simétrica na ABB (contando a partir de 1).

3. int **mediana** (): retorna o elemento que contém a mediana da ABB. Se a ABB possuir um número par de elementos, retorne o menor dentre os dois elementos medianos.

4. double **media** (int x): retorna a média aritmética dos nós da árvore que x é a raiz.

5. boolean **ehCheia** (): retorna verdadeiro se a ABB for uma árvore binária cheia e falso, caso contrário.

6. boolean **ehCompleta** (): retorna verdadeiro se a ABB for uma árvore binária completa.

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

8. void **imprimeArvore** (int s): se “s” igual a 1, o método imprime a árvore no formato 1, “s” igual a 2, imprime no formato 2.

Métodos implementados

Método de Inserção

```java
public void insert(int value) {
        if (!search(value)) {
            insertRecursive(root, value);
            System.out.println(value + " inserido");
        } else {
            System.out.println(value + " já está na árvore, não pode ser inserido");
        }
    }

    private void insertRecursive(Node current, int value) {
        if (current == null) {
            root = new Node(value);
        } else if (value < current.getValue()) {
            if (current.getLeft() == null) {
                current.setLeft(new Node(value));
                current.setSizeLeft(1);
                return;
            }
            insertRecursive(current.getLeft(), value);
            current.setSizeLeft(current.getSizeLeft() + 1); // Atualiza o tamanho da subárvore à
esquerda
        } else if (value > current.getValue()) {
            if (current.getRight() == null) {
                current.setRight(new Node(value));
                current.setSizeRight(1);
                return;
            }
            insertRecursive(current.getRight(), value);
            current.setSizeRight(current.getSizeRight() + 1); // Atualiza o tamanho da subárvore à
direita
        }
    }
```

A função insert é a função pública, a qual terá a checagem se o valor já se encontra presente na árvore ou não, caso já esteja presente, ele avisa ao usuário por meio de um print no console. Caso não esteja, ele chama a função insertRecursive que irá de fato fazer a inserção na árvore.

Essencialmente, a função insertRecursive percorre a árvore recursivamente, escolhendo a direção com base nos valores dos nós, até encontrar um local adequado para inserir o novo valor na árvore binária de busca. O tamanho das sub árvores à esquerda e à direita de cada nó é atualizado durante o processo, fornecendo informações sobre a estrutura da árvore.

Complexidade Assintótica:
Se a árvore for balanceada, a complexidade será O(log n). Se a árvore não for balanceada e se tornar linear, a complexidade pode ser O(n).

Método de Remoção

```java
public void remove(int value) {
        if (search(value)) {
            root = removeRecursive(root, value);
            System.out.println(value + " removido");
        } else {
            System.out.println(value + " não está na árvore, não pode ser removido");
        }
    }

    private Node removeRecursive(Node current, int value) {
        if (current == null) {
            return current;
        }

        if (value < current.getValue()) {
            current.setLeft(removeRecursive(current.getLeft(), value));
            current.setSizeLeft(current.getLeft() != null ? current.getLeft().getSizeLeft() +
current.getLeft().getSizeRight() + 1 : 0);
        } else if (value > current.getValue()) {
            current.setRight(removeRecursive(current.getRight(), value));
            current.setSizeRight(current.getRight() != null ? current.getRight().getSizeLeft() +
current.getRight().getSizeRight() + 1 : 0);
        } else {
            if (current.getLeft() == null) {
                return current.getRight();
            } else if (current.getRight() == null) {
                return current.getLeft();
            }

            current.setValue(findMinValue(current.getRight()));
            current.setRight(removeRecursive(current.getRight(), current.getValue()));
            current.setSizeLeft(current.getLeft() != null ? current.getLeft().getSizeLeft() +
current.getLeft().getSizeRight() + 1 : 0);
            current.setSizeRight(current.getRight() != null ? current.getRight().getSizeLeft() +
current.getRight().getSizeRight() + 1 : 0);
        }

        return current;
    }
```

```java
private int findMinValue(Node root) {
        int minValue = root.getValue();
        //Percorre a subárvore até o nó folha mais a esquerda, encontrando o nó com menor
valor da subárvore
        while (root.getLeft() != null) {
            minValue = root.getLeft().getValue();
            root = root.getLeft();
        }
        return minValue;
    }
```

A função remove() é a função pública, a qual terá a checagem se o valor se encontra presente na árvore ou não, caso não esteja presente, ele avisa ao usuário por meio de um print no console. Caso esteja, ele chama a função removeRecursive() que irá de fato fazer a remoção na árvore.

A função removeRecursive recebe como entrada um nó current e um valor value. Primeiramente, ela verifica se o nó atual é nulo; nesse caso, retorna o próprio nó atual. Se o valor value for menor que o valor do nó atual, a função chama a si mesma recursivamente no filho esquerdo do nó atual. Caso contrário, se o valor value for maior que o valor do nó atual, a função chama a si mesma recursivamente no filho direito do nó atual.

Se o valor value for igual ao valor do nó atual, a função trata três casos:

1. Se o nó atual não tiver filho à esquerda, a função retorna o filho direito do nó atual.
2. Se o nó atual não tiver filho à direita, a função retorna o filho esquerdo do nó atual.
3. Se o nó atual tiver tanto filho à esquerda quanto à direita, a função encontra o valor mínimo na subárvore à direita (o sucessor in-order) utilizando a função findMinValue, substitui o valor do nó atual por esse valor mínimo e chama a si mesma recursivamente no filho direito do nó atual para remover o valor mínimo da subárvore à direita.

Por fim, a função atualiza o tamanho das sub árvores à esquerda e à direita do nó atual com base na estrutura atualizada da árvore. Em seguida, retorna o nó atual atualizado.

Complexidade Assintótica:
Se a árvore for balanceada, a complexidade será O(log n). Se a árvore não for balanceada e se tornar linear, a complexidade será O(n).

Método de busca

```java
public boolean search(int value) {
        return searchRecursive(root, value);
    }

    private boolean searchRecursive(Node current, int value) {
        if (current == null) {
            return false;
        }

        if (current.getValue() == value) {
            return true;
        }

        //se o value for menor do que o valor do nó atual, vai para a subárvore à
esquerda.
        if (value < current.getValue()) {
            return searchRecursive(current.getLeft(), value);
        }
        //se o value for maior do que o valor do nó atual, vai para a subárvore à direita.
        else {
            return searchRecursive(current.getRight(), value);
        }
    }
```

A função search serve para chamar a função searchRecursive, a função search recebe apenas um valor inteiro que será o valor a ser pesquisado na árvore e passado como parâmetro para a função de busca recursiva.

A função searchRecursive busca recursivamente o valor especificado na árvore binária de busca. A função começa no nó raiz e, se o nó for null, retorna false. Se o valor especificado for igual ao valor do nó atual, a função retorna true. Caso contrário, a função verifica se o valor especificado é menor ou maior do que o valor do nó atual. Se o valor especificado for menor, a função chama-se recursivamente na subárvore à esquerda. Se o valor especificado for maior, a função chama recursivamente a si mesma na subárvore à direita.

Complexidade Assintótica:
Nesse caso também ocorre o mesmo das duas funções anteriores. Se a árvore for balanceada, a complexidade será O(log n). Se a árvore não for balanceada e se tornar linear, a complexidade será O(n).

Método Enésimo Elemento

```java
public int enesimoElemento(int n) {
        //Checa se o valor enesimo passado está dentro do intervalo da árvore
        if (n < 1 || n > countNodes(root)) {
            System.out.println("Provide a number in the correct interval.");
            return -1;
        }
        return enesimoElementoRec(root, n);
    }


    //Função que encontra o elemento na n-ésima posição
    private int enesimoElementoRec(Node node, int n) {
        int nodesNaEsquerda = node.getSizeLeft() + 1; // Contagem de nós na subárvore
esquerda + o próprio nó

        if (n == nodesNaEsquerda) {
            return node.getValue(); // Encontramos o n-ésimo elemento
        } else if (n < nodesNaEsquerda) {
            return enesimoElementoRec(node.getLeft(), n); // O n-ésimo elemento está na
subárvore esquerda
        } else {
            return enesimoElementoRec(node.getRight(), n - nodesNaEsquerda); // O n-ésimo
elemento está na subárvore direita
        }
    }
```

A função enesimoElemento é uma função pública que recebe um valor inteiro, realiza a checagem se o n da entrada está entre o intervalo correto, que é entre 1 e a quantidade de nós na árvore. Caso não esteja, ele exibe uma mensagem de alerta na tela e encerra o andamento da função. Caso n de entrada esteja no intervalo correto, ele chama a função enesimoElementoRec passando o n e a raiz da árvore como parâmetro.

A função enesimoElementoRec recebe um nó node e um valor n como entrada. Ela primeiro calcula a quantidade de nós na subárvore esquerda do nó atual. Se o valor n for igual à quantidade de nós na subárvore esquerda, então o n-ésimo elemento é o próprio nó atual. Nesse caso, a função retorna o valor do nó atual.

Caso contrário, a função verifica se o valor n é menor ou maior do que a quantidade de nós na subárvore esquerda. Se o valor n for menor, então o n-ésimo elemento está na subárvore esquerda. Nesse caso, a função chama recursivamente a si mesma na subárvore esquerda com o valor n reduzido pela quantidade de nós na subárvore esquerda.

Se o valor n for maior, então o n-ésimo elemento está na subárvore direita. Nesse caso, a função chama recursivamente a si mesma na subárvore direita com o valor n original.

O processo continua até que um dos seguintes casos ocorra:

- O n-ésimo elemento é encontrado na árvore. Nesse caso, a função retorna o valor do n-ésimo elemento.
- O n-ésimo elemento não existe na árvore. Nesse caso, a função retorna -1.

Complexidade Assintótica:

Nesse caso também ocorre o mesmo visto anteriormente. Se a árvore for balanceada, a complexidade será O(log n). Se a árvore não for balanceada e se tornar linear, a complexidade será O(n).

Método Posição

```java
public int posicao(int x) {
        int position = posicaoRec(root, x);
        if(position == -1){
            System.out.println("Element not found");
            return position;
        }
        return position;
    }

    private int posicaoRec(Node node, int x) {
        if (node == null) {
            return -1; // O Elemento não foi encontrado na árvore
        }

        int posicaoEsquerda = posicaoRec(node.getLeft(), x); // Chamada recursiva para o nó à esquerda

        if (posicaoEsquerda != -1) {
            return posicaoEsquerda; // Retorna a posição da subárvore esquerda se o elemento for
encontrado
        }

        if (node.getValue() == x) {
            int posicaoSubarvoreEsquerda = node.getSizeLeft() + 1;
            return posicaoSubarvoreEsquerda; // Retorna a posição na raiz se o elemento for encontrado
        }

        int posicaoDireita = posicaoRec(node.getRight(), x);

        if (posicaoDireita != -1) {
            int posicaoSubarvoreEsquerda = node.getSizeLeft() + 1;
            return posicaoSubarvoreEsquerda + posicaoDireita; // Retorna a posição na subárvore direita
se o elemento for encontrado
        }

        return -1; // Retorna -1 se o elemento não for encontrado na árvore
    }
```

A função posicao é a função pública que irá receber um valor inteiro X, em seguida é feita atribuição à variável position da chamada da função posicaoRec, a qual recebe a raiz da árvore e o X como parâmetros. Após isso, é feita uma checagem se a variável position é igual a -1, se sim, ele imprime uma mensagem de erro na tela e retorna -1. Se não, ele retorna a posição do elemento X na árvore.

A função posicaoRec recebe um nó node e um valor x como entrada. Ela primeiro verifica se o nó node é null, em que caso ela retorna -1. Se o valor x for igual ao valor do nó atual, a função retorna a posição do nó atual, que é igual à quantidade de nós na subárvore esquerda do nó atual.

Caso contrário, a função verifica se o valor x é menor ou maior do que o valor do nó atual. Se o valor x for menor, então o elemento está na subárvore esquerda. Nesse caso, a função chama-se recursivamente na subárvore esquerda.

Se o valor x for maior, então o elemento está na subárvore direita. Nesse caso, a função chama-se recursivamente na subárvore direita.

O processo continua até que um dos seguintes casos ocorra:

- O elemento é encontrado na árvore. Nesse caso, a função retorna a posição do elemento.
- O elemento não existe na árvore. Nesse caso, a função retorna -1.

Complexidade Assintótica:

Nesse caso, a função é O(n), isso ocorre porque a função examina cada nó da árvore apenas uma vez, independentemente do tamanho da árvore. Em média, a função fará O(n/2) chamadas recursivas. Isso ocorre porque, na média, a função encontrará o elemento especificado na metade da árvore. E no seu pior caso, a função terá que percorrer a árvore inteira e o elemento não estar presente na árvore.

Método Mediana

```java
public int mediana() {
        int totalNodes = countNodes(root);

        // Verifica se o número total de nós é par ou ímpar
        if (totalNodes % 2 == 1) {
            // Se for ímpar, a mediana está em um único nó
            int medianPosition = (totalNodes + 1) / 2;
            return enesimoElementoRec(root, medianPosition);
        } else {
            // Se for par, a mediana é a média dos dois elementos do meio
            int medianPosition1 = totalNodes / 2;
            int medianPosition2 = totalNodes / 2 + 1;
            int medianValue1 = enesimoElementoRec(root, medianPosition1);
            int medianValue2 = enesimoElementoRec(root, medianPosition2);
            return Math.min(medianValue1, medianValue2);
        }
    }
```

A função mediana não possui parâmetros de entrada. Ela primeiro calcula o número total de nós na árvore usando a função countNodes. Se o número total de nós for ímpar, então a mediana está em um único nó. A função chama a função enesimoElementoRec para encontrar o valor do nó na posição mediana.

Se o número total de nós for par, então a mediana é o menor dos dois elementos do meio. A função chama a função enesimoElementoRec para encontrar os valores dos dois elementos na posição mediana. A função retorna o menor desses dois valores.

Complexidade Assintótica:

Nesse caso, a complexidade assintótica do método mediana é dominada pela complexidade assintótica de enesimoElementoRec, que é utilizado para encontrar os elementos que estão em uma posição específica. Ou seja, caso a árvore seja balanceada, será O(log n). E caso não esteja balanceada, será O(n).

Método Média

```java
public double media(int x) {
        Node rootNode = searchNode(x);

        if (rootNode != null) {
            int[] result = new int[2];  // Índice 0: soma, Índice 1: número total de elementos
            countElements(rootNode, result, 1);
            result[1] = 0;
            countElements(rootNode, result, 0);

            if (result[1] != 0) {
                return (double) result[0] / result[1];
            } else {
                System.out.println("ERROR: No elements found.");
                return -1;
            }
        } else {
            System.out.println("ERROR: Root invalid.");
            return -1;
        }
    }
```

A função media recebe um inteiro x como entrada, que representa o valor do nó raiz. Ela primeiro chama a função searchNode para encontrar o nó com o valor x. Se o nó for encontrado, ela o armazena na variável rootNode.

Se o rootNode não for null, ela inicializa um array result com dois elementos: result[0] armazenará a soma de todos os elementos na árvore, e result[1] armazenará o número total de elementos na árvore.

Em seguida, ela chama a função countElements duas vezes, passando o rootNode, o array result e uma flag 0 ou 1. A flag 0 indica que ela deve contar o número total de elementos, e a flag 1 indica que ela deve somar todos os elementos na árvore.

Se o número total de elementos (result[1]) não for zero, ela calcula a média dividindo a soma de todos os elementos (result[0]) pelo número total de elementos (result[1]). Em seguida, ela retorna a média como um double.

Se o número total de elementos for zero, ela imprime uma mensagem de erro e retorna -1.

A função media assume que a árvore binária de busca é válida e que a função searchNode encontra corretamente o nó com o valor x.

Complexidade Assintótica:
A complexidade assintótica da função media é dominada pela complexidade de searchNode, que em seu pior caso, é O(n), pois precisa percorrer a árvore toda procurando o nó.

Método EhCheia

```java
public void ehCheia() {
         if(ehCheiaRec(root)) {
             System.out.println("A árvore é cheia");
         }
         else {
             System.out.println("A árvore não é cheia");
         }
    }

    private boolean ehCheiaRec(Node node) {
        // Se a árvore está vazia, ela é considerada cheia
        if (node == null) {
            return true;
        }

        // Verifica se o nó tem ambos os filhos ou nenhum
        if ((node.getLeft() == null && node.getRight() != null) || (node.getLeft() != null &&
node.getRight() == null)) {
            return false;
        }

        // Chama a função recursivamente para os filhos
        return ehCheiaRec(node.getLeft()) && ehCheiaRec(node.getRight());
    }
```

A função ehCheia é apenas uma função pública para ser chamada e dentro de si tem uma chamada para ehCheiaRec passando como parâmetro a raiz da árvore. E imprimindo a mensagem de acordo com o resultado da função ehCheiaRec.

A função ehCheiaRec verifica se uma árvore binária de busca é cheia. A função recebe um nó como entrada. Se o nó for null, a árvore está vazia e, portanto, é considerada cheia.

Se o nó não for null, a função verifica se o nó tem ambos os filhos ou nenhum. Se o nó tiver apenas um filho, a árvore não é cheia.

Caso contrário, a função chama a si mesma recursivamente para os filhos do nó. Se ambos os filhos forem cheios, então o nó também é cheio.

Complexidade Assintótica:
A complexidade assintótica de ehCheia é O(n) já que ela irá percorrer todos os nós da árvore para saber se ela é cheia ou não.

Método ehCompleta

```java
public void ehCompleta() {
        int altura = alturaArvore(root);
        if(verificaCompleta(root, 0, altura)) {
            System.out.println("A árvore é completa");
        }
        else {
            System.out.println("A árvore não é completa");
        }
    }

    // Função auxiliar para verificar se a árvore é completa
    private boolean verificaCompleta(Node node, int nivelAtual, int alturaTotal) {
        // Verifica se a árvore está vazia (considerada completa)
        if (node == null) {
            return true;
        }

        // Se um nó tem uma subárvore vazia, ela deve estar no último ou penúltimo nível
        if ((node.getLeft() == null && node.getRight() != null) || (node.getLeft() != null &&
node.getRight() == null)) {
            return nivelAtual == alturaTotal - 1 || nivelAtual == alturaTotal - 2;
        }

        // Chama recursivamente para os filhos
        return verificaCompleta(node.getLeft(), nivelAtual + 1, alturaTotal)
                && verificaCompleta(node.getRight(), nivelAtual + 1, alturaTotal);
    }
```

A função ehCompleta é uma função pública que não recebe parâmetros, apenas chama a função verificaCompleta e exibe a mensagem de acordo com o tipo do retorno.

A função verificaCompleta recebe três parâmetros: node, nivelAtual, alturaTotal. Se a árvore não estiver vazia, a função verifica se o nó atual tem uma subárvore vazia. Se o nó atual tiver uma subárvore vazia, essa subárvore deve estar no último ou penúltimo nível da árvore.

Caso contrário, a função chama a si mesma recursivamente para os filhos do nó atual. A função verifica se os filhos do nó atual também são completos.

Complexidade Assintótica:
A complexidade de verificaCompleta é O(n), já que ela terá que percorrer todos os nós da árvore.

Método PreOrder

```java
public void preOrder(){
        preOrdem(root);
    }

    public void preOrdem(Node node){
        System.out.print(node.getValue() + " ");
        if(node.getLeft() != null) preOrdem(node.getLeft());
        if(node.getRight() != null) preOrdem(node.getRight());
    }
```

A função preOrder é uma função pública que apenas chama a função preOrdem, passando a raiz da árvore como parâmetro.

A função preOrdem realiza o percurso em pré ordem na árvore, imprimindo os elementos da mesma nessa ordem específica.

Complexidade Assintótica:
A complexidade assintótica de pré ordem é O(n), já que a função irá percorrer toda a árvore imprimindo seus elementos na ordem específica.

Método imprimeArvore

```java
public void imprimeArvore(int s) {
        if (s == 1) {
            imprimeArvoreFormato1(root, 1);
        } else if (s == 2) {
            imprimeArvoreFormato2(root);
        }
    }

    private void imprimeArvoreFormato1(Node node, int depth) {
        if (node != null) {
            String nodeValue = String.valueOf(node.getValue());
            String prefix = "\t".repeat(depth);

            System.out.println(prefix + nodeValue + "-".repeat(depth * 7));
// Ajuste o valor 5 para controlar o espaçamento entre os traços

            imprimeArvoreFormato1(node.getLeft(), depth + 1);
            imprimeArvoreFormato1(node.getRight(), depth + 1);
        }
    }
```

```java
private void imprimeArvoreFormato2(Node node) {
        if (node == null) {
            return;
        }

        System.out.print("(" + node.getValue());

        // Imprime nó à esquerda
        imprimeArvoreFormato2(node.getLeft());

        // Imprime nó à direita
        imprimeArvoreFormato2(node.getRight());

        System.out.print(")");
    }
```

A função imprimeArvore é uma função pública que recebe como parâmetro um número inteiro, e caso o número seja igual a 1 ou 2, chamará a função respectiva de impressão atrelada a esse número.

A função imprimeArvore1 imprime a árvore de forma formatada, com cada nó recuado por um caractere de tabulação. A profundidade do recuo é igual ao nível do nó na árvore. Por

exemplo, o nó raiz será recuado por 0 tabulações, os filhos do nó raiz serão recuados por 1 tabulação, e assim por diante.

A função imprimeArvore2 imprime a árvore em um formato entre parênteses. Cada nó é encerrado entre parênteses, e os filhos esquerdo e direito de um nó são impressos dentro dos parênteses.

Complexidade Assintótica:
A complexidade assintótica de imprimeArvore1 é O(n * d), onde n é o número de nós na árvore e d é a profundidade máxima da árvore. Isso ocorre porque a função chama a si mesma recursivamente para cada nó da árvore, e a profundidade da recursão é igual à profundidade do nó na árvore.

O pior caso ocorre quando a árvore é uma árvore completa. Nesse caso, a profundidade máxima da árvore é log n, e a complexidade de tempo é O(n * log n).

A complexidade assintótica de imprimeArvore2 é O(n), onde n é o número de nós na árvore. Isso ocorre porque a função chama a si mesma recursivamente para cada nó da árvore, mas a profundidade da recursão é sempre 1.

Conclusão

Por fim, é possível perceber que a árvore está, finalmente, implementada. Além de suas operações básicas, a mesma possui os métodos adicionais requisitados no documento de definição deste presente trabalho. E a mesma sendo uma versão 100% funcional do que foi solicitado.