UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
INSTITUTO METRÓPOLE DIGITAL
CAMPUS NATAL CENTRAL

Rubens Matheus Venancio Melo Oliveira
Wisla Alves Argolo

IMPLEMENTAÇÃO DE ÁRVORE BINÁRIA DE BUSCA

Natal - RN
2023

Rubens Matheus Venancio Melo Oliveira
Wisla Alves Argolo

IMPLEMENTAÇÃO DE ÁRVORE BINÁRIA DE BUSCA

Relatório técnico apresentado como avaliação da disciplina Estruturas de Dados Básicas I ministrada pelo professor Sidemar Fideles Cezario para o curso de Bacharelado em Tecnologia da Informação do Instituto Metropóle Digital da Universidade Federal do Rio Grande do Norte - Campus Natal Central.

Natal - RN
2023

Sumário

Lista de Algoritmos

Lista de Tabelas

1 INTRODUÇÃO

Uma árvore pode ser definida como um agrupamento finito de entidades chamadas nós, arranjados de maneira que, com a exceção do elemento inicial chamado raiz, cada um está vinculado a um único nó superior, ou pai, e pode ter vários nós inferiores, ou filhos. Em particular, uma árvore de busca binária restringe cada nó a ter, no máximo, dois filhos e para qualquer nó dado, todos os nós na sua subárvore esquerda contêm valores menores do que o próprio nó, e todos na subárvore direita contêm valores maiores (SZWARCFITER; MARKENZON, 2015; DROZDEK, 2013). Essa configuração torna as árvores de busca binária bastante utilizadas para operações de busca e, por isso, podem ser frequentemente adotadas em uma vasta gama de aplicações, como sistemas de gerenciamento de arquivos e de bancos de dados (GOODRICH; TAMASSIA; MOUNT, 2011).

Nesse sentido, a análise da eficiência das operações de uma árvore de busca binária tem um papel fundamental na área da computação. Isso pode ser feito através de métodos empíricos, que consistem em executar o algoritmo e medir o tempo para entradas de tamanhos distintos, ou por meio de análise matemática para compreender o desempenho do algoritmo de maneira independente das especificidades do hardware ou do software empregado e considerando conjuntos de dados de grande escala (SZWARCFITER; MARKENZON, 2015; SEDGEWICK, 1998).

Pensando nisso, este relatório propõe a implementação de uma árvore binária de busca e a análise assintótica de suas operações. O objetivo desse processo consiste em examinar o comportamento da estrutura de dados em questão a fim de auxiliar o programador no planejamento futuro e aplicação da estrutura quando for vantajoso.

2 PROCEDIMENTO EXPERIMENTAL

2.1 Materiais

Esta subseção apresenta e descreve os softwares necessários para a realização deste trabalho.

2.1.1 Ferramentas de programação

A árvore binária de busca foi implementada na linguagem Java, com as especificações presentes na Tabela 1.

Tabela 1: Versão do Java

Versão	**Tipo de Suporte**	**Ambiente de Execução**
17.0.8	LTS	Java(TM) SE Runtime Environment *Build:* 17.0.8+9-LTS-211

Fonte: Elaborado pelo autor (2023).

Adicionalmente, tal código foi modificado utilizando a versão 4.28.0 do ambiente de desenvolvimento Eclipse.

2.2 Algoritmos

Nesta subseção, os pseudocódigos dos algoritmos utilizados na implementação da árvore binária de busca.

2.2.1 Árvore Binária de Busca

A árvore binária de busca é uma estrutura formada por nós. Nesse sentido, implementamos a classe "No"para representá-los e cada nó possui: i) o atributo valor do nó, ii) tamanhoEsquerda e tamanhoDireita que armazenam o número de nós nas subárvores esquerda e direitaos, respectivamente, iii) somaSubArvore para obter a soma dos valores de todos os nós na subárvore que apresenta o nó em questão como raiz, iv) esquerda e direita, as quais são referências aos nós filhos à esquerda e à direita, respectivamente e v) altura referente a distância deste nó até o descendente mais distante.

Além disso, a altura é inicializada com o valor 1, dado que, por definição, toda folha tem altura 1.

Algoritmo 1: Classe No

```
1  Classe No
2      Atributos
3          valor : inteiro
4          tamanhoEsquerda : inteiro
5          tamanhoDireita : inteiro
6          somaSubArvore : inteiro
7          esquerda : No
8          direita: No
9          altura: inteiro
10     Construtor No(valor)
11         this.valor ← valor
12         this.altura ← 1
13         this.somaSubArvore ← valor
```

A partir disso, a classe que representa a árvore de busca binária foi implementada com um nó raiz e métodos avançados e auxiliares além dos básicos de inserção, remoção e busca.

Algoritmo 2: Classe ArvoreBinariaBusca

```
1  Classe ArvoreBinariaBusca
2      Atributos
3          raiz: No
4      Métodos
5          busca(valor)
6          buscaRecursiva(no, valor)
7          inserir(valor)
8          inserirRecursivo(no, valor)
9          remover(valor)
10         removerRecursivo(no, pai, valor)
11         encontraAntecessor(no)
12         atualizaSomaSubArvore(no)
13         somaSubarvore(no)
14         atualizaAltura(no)
15         altura(no)
16         imprimeArvore(s)
17         imprimeFormato1(no, espaco, quantidadeTravessoes)
18         imprimeFormato2(no)
19         media(valor)
20         enesimoElemento(n)
21         enesimoElementoRecursivo(no, n)
22         posicao(valor)
23         posicaoRecursiva(no, valor)
24         tamanhoDaSubArvore(no)
25         mediana()
26         ehCheia()
27         ehCompleta()
28         preOrdem()
29         preOrdemRecursivo(no, resultado)
```

2.2.2 Busca

Este algoritmo recebe o valor a ser procurado como parâmetro e começa a busca pela raiz da árvore, comparando o valor buscado com o valor do nó atual. Se o valor é menor que o do nó atual, o valor procurado deve estar na subárvore à esquerda e se for maior, na subárvore à direita. O processo é repetido recursivamente até encontrar o nó com o valor e retorná-lo ou chegar a um nó nulo, o que indica que o valor buscado não está na árvore.

Algoritmo 3: Algoritmo de Busca

Entrada: Um valor a ser buscado

Saída: O nó contendo o valor ou *null* se o valor não for encontrado

```
1 Método busca(valor)
2     retorna buscaRecursiva(raiz, valor)
3 Método buscaRecursiva(no, valor)
4     se no = null ou node.valor = valor então
5         retorna no
6     se valor < no.valor então
7         retorna buscaRecursiva(no.esquerda, valor)
8     senão
9         retorna buscaRecursiva(no.direita, valor)
```

2.2.3 Atualização dos nós

Para manter as informações dos nós atualizadas ao inserir ou remover um novo nó, implementamos algoritmos auxiliares.

O Algoritmo 4 é responsável por atualizar a soma dos valores da subárvore de um determinado nó somando o valor do nó atual com as somas das subárvores esquerda e direita.

Algoritmo 4: Algoritmo de Atualização da Soma da Subárvore de um nó

Entrada: Um nó da árvore

Saída: Nó com soma da subárvore atualizada (sem retorno explícito)

```
1 Método atualizaSomaSubArvore(no)
2     se no ≠ null então
3         no.somaSubArvore ← somaSubArvore(no.direita) +
4                            somaSubArvore(no.esquerda) +
5                            no.valor
6 Método somaSubArvore(no)
7     se no = null então
8         retorna 0
9     retorna no.somaSubArvore
```

Já no Algoritmo 5, a altura do nó em questão é corrigida por meio da altura dos nós a esquerda e direita.

Algoritmo 5: Algoritmo de Atualização da altura de um nó

Entrada: Um nó da árvore

Saída: Nó com altura atualizada (sem retorno explícito)

```
1 Método atualizaAltura(no)
2     se no ≠ null então
3         no.altura ← maximo(altura(no.esquerda), altura(no.direita)) + 1
4 Método altura(no)
5     se no = null então
6         retorna 0
7     retorna no.altura
```

2.2.4 Inserir

Na inserção, se a árvore está vazia, a raiz da árvore recebe o valor a ser inserido. Caso contrário, o valor a ser inserido é comparado com o valor do nó atual recursivamente. Se for menor, o algoritmo segue para a subárvore esquerda e caso seja maior, segue para a direita. Esse processo é repetido tem inicio na raiz e ocorre até encontrar o local correto na árvore para o novo valor, mas caso o valor já exista, nada ocorre para evitar duplicatas.

Ademais, durante a inserção, o algoritmo atualiza o tamanho das subárvores esquerda e direita, a soma dos valores da subárvore, e a altura do nó para que a árvore mantenha suas propriedades.

Algoritmo 6: Algoritmo de Inserção

Entrada: Um valor a ser inserido

Saída: Verdadeiro se o valor foi inserido com sucesso, falso se não

```
1  Método inserir(valor)
2      se raiz = null então
3          raiz ← novo No(valor)
4          retorna verdadeiro
5      retorna inserirRecursivo(raiz, valor)
6  Método inserirRecursivo(no, valor)
7      se valor < no.getValor() então
8          se no.esquerda = null então
9              no.esquerda ← novo No(valor)
10         senão se !insertRecursive(no.esquerda, valor) então
11             retorna falso
12         no.tamanhoEsquerda ← no.tamanhoEsquerda + 1
13     senão se valor > no.valor então
14         se no.direita = null então
15             no.direita ← novo No(valor)
16         senão se !insertRecursive(no.direita, valor) então
17             retorna falso
18         no.tamanhoDireita ← no.tamanhoDireita + 1
19     senão
20         retorna falso
21     atualizaSomaSubArvore(no)
22     atualizaAltura(no)
23     retorna verdadeiro
```

2.2.5 Remover

Neste projeto, a solução para a remoção realiza uma busca recursiva para encontrar o nó com o valor recebido como parâmetro. Caso o valor seja menor ou maior que o valor do nó atual, a busca continua na subárvore esquerda ou direita, respectivamente. Esse procedimento inicia na raiz e é repetido até que um nó nulo seja encontrado, indicando a ausência do valor na árvore, ou até que se encontre o nó com o valor em questão. Ao encontrá-lo, existem os seguintes cenários de remoção: i) quando o nó tem dois filhos, o seu antecessor é encontrado, de modo que o valor deste nó é substituído pelo do antecessor e o antecessor é removido, ii) se o nó tem apenas um filho, ele é substituído por esse filho,

e iii) se o nó é uma folha, ele é removido e a referência do nó pai é atualizada para nulo.

Neste projeto, os casos ii) e iii) foram tratados em conjunto. Além disso, para encontrar o sucessor do nó especificado, foi realizada uma busca pelo nó mais à esquerda na subárvore direita desse nó.

Após uma remoção bem sucedida, os nós da árvore tem seus campos atualizados por métodos auxiliares.

Algoritmo 7: Algoritmo de Remoção

Entrada: Um valor a ser removido

Saída: Verdadeiro se o nó com o valor foi removido, falso caso contrário

```
1  Método remover(valor)
2      retorna removerRecursivo(raiz, null, valor)
3  Método removerRecursivo(no, pai, valor)
4      se no = null então
5          retorna falso
6      se valor < no.valor então
7          se !removerRecursivo(no.esquerda, no, valor) então
8              retorna falso
9          no.tamanhoEsquerda ← no.tamanhoEsquerda − 1
10     senão se valor > no.valor então
11         se !removerRecursivo(no.direita, no, valor) então
12             retorna falso
13         no.tamanhoDireita ← no.tamanhoDireita − 1
14     senão
15         se no.direita ≠ null  e  no.esquerda ≠ null então
16             antecessor ← encontraAntecessor(no.esquerda)
17             no.valor ← antecessor.valor
18             removerRecursivo(no.esquerda, no, antecessor.valor)
19             no.tamanhoEsquerda ← no.tamanhoEsquerda − 1
20             atualizaSomaSubArvore(no)
21             atualizaAltura(no)
22         senão
23             noFilho ← no.direita
24             se no.esquerda ≠ null então
25                 noFilho ← no.esquerda
26             se pai = null então
27                 raiz ← noFilho
28             senão se pai.esquerda = no então
29                 pai.esquerda ← noFilho
30             senão se pai.direita = no então
31                 pai.direita ← noFilho
32         retorna verdadeiro
33     atualizaSomaSubArvore(no)
34     atualizaAltura(no)
35     retorna verdadeiro
```

Algoritmo 8: Encontrar Antecessor

Entrada: Um nó da árvore
Saída : O antecessor do nó fornecido

```
1 Método encontrarAntecessor(no)
2     enquanto no.direita ≠ null faça
3         no ← no.direita
4     retorna no
```

2.2.6 Imprimir a Árvore

Este método é utilizado para imprimir a árvore binária de busca em um formato específico de acordo com o parâmetro passado ao método (Algoritmo 9).

Algoritmo 9: Impressão de Árvore Binária de Busca

Entrada: Um inteiro s indicando o formato de impressão
Saída : Impressão da árvore no formato especificado

```
1 Método imprimirArvore(s)
2     se s = 1 então
3         imprimirFormato1(raiz, , altura(raiz) × 8)
4     senão
5         imprimirFormato2(raiz)
```

Um dos formatos refere-se ao diagrama de barras em que cada barra corresponde a um nó da árvore e nós de mesmo nível possuem barras de mesmo tamanho. Dessa forma, a barra maior no topo indica a raiz, seguida por barras menores que representam os filhos, netos, e assim por diante.

Para imprimir a árvore nessa organização, conforme observado no Algoritmo 10, consideramos que o tamanho da barra - quantidade de travessões - da raíz é oito vezes a altura da árvore, sem indentação. Nos nós subsequentes, o número de travessões reduz em oito - tamanho ocupado por um *tab* - a cada nível, e a indentação aumenta conforme descemos na hierarquia da árvore.

Algoritmo 10: Impressão de Árvore Binária de Busca no Formato 1

Entrada: O nó a ser impresso, a identação e quantidade de travessões necessários para imprimir

Saída : Impressão da árvore no formato de diagrama de barras

```
1 Método imprimirFormato1(no, espaco, quantidadeTravessoes)
2     se no ≠ null então
3         quantidadeDigitos ← tamanho(no.valor)
4         travessoes ← repetir('−', quantidadeTravessoes − quantidadeDigitos)
5         imprimir(espaco + no.valor + travessoes)
6         quantidadeTravessoes ← quantidadeTravessoes − 8
7         imprimirFormato1(no.esquerda, espaco + \t , quantidadeTravessoes)
8         imprimirFormato1(no.direita, espaco + \t , quantidadeTravessoes)
```

Já no segundo formato a árvore é impressa na representação de parênteses aninhados, na qual cada árvore está contida entre parênteses. Para isso, o Algoritmo 11 imprime um parêntese de abertura para o nó atual e o seu valor. O processo é repetido para os filhos recursivamente e após isso, ocorre a impressão de um parêntese fechado.

Algoritmo 11: Impressão de Árvore Binária de Busca no Formato 2

Entrada: Um nó a ser impresso

Saída : Impressão da árvore no formato de parênteses aninhados

```
1 Método imprimirFormato2(no)
2     se no ≠ null então
3         imprimir('(' + no.valor)
4         se no.esquerda ≠ null então
5             imprimirFormato2(no.esquerda)
6         se no.direita ≠ null então
7             imprimirFormato2(no.direita)
8         imprimir(')');
```

2.2.7 Média

Este algoritmo recebe um valor x como parâmetro e retorna a média dos valores nós da árvore em que x é a raiz, ou -1 caso um nó com valor x não exista. Para tanto, ele chama o método *buscar* e encontra o nó que tem x como valor, de modo a obter a *somaSubArvore* e quantidade de nós na subárvore desse nó, permitindo o cálculo da média.

Algoritmo 12: Média na Subárvore

Entrada: Um inteiro x

Saída : A média dos valores nós da árvore em que x é a raiz.

```
1  Método media(x)
2      no ← buscar(x);
3      se no = null então
4          retorna -1;
5      totalNo ← no.tamanhoEsquerda + no.tamanhoDireita + 1;
6      retorna no.somaSubArvore / totalNo;
```

2.2.8 Enésimo Elemento

Esta solução busca o enésimo elemento de acordo com o percurso simétrico a partir de n passado como parâmetro. Esse processo inicia na raiz e ocorre comparando n com a quantidade de nós na subárvore esquerda do nó atual. Se n for menor ou igual ao total de nós na subárvore esquerda, a busca pelo enésimo elemento prossegue nesta subárvore, ou se n for igual ao tamanho da subárvore esquerda mais um - incluindo o nó corrente -, o nó atual é enésimo elemento e o algoritmo encerra. Caso contrário, o enésimo elemento está na subárvore direita e o valor de n é ajustado para a pesquisa.

Algoritmo 13: Encontrar o Enésimo Elemento na Árvore

Entrada: Um inteiro n

Saída : O valor do enésimo elemento na árvore

```
1  Método enesimoElemento(n)
2      retorna enesimoElementoRecursivo(raiz, n);
3  Método enesimoElementoRecursivo(no, n)
4      se no = null então
5          retorna -1;
6      se tamanhoEsquerda(no) ≥ n então
7          retorna enesimoElementoRecursivo(no.esquerda, n);
8      se no.tamanhoEsquerda + 1 = n então
9          retorna no.valor;
10     senão
11         n ← n − no.tamanhoEsquerda − 1;
12         retorna enesimoElementoRecursivo(no.direita, n);
```

2.2.9 Tamanho da Sub Árvore

O método *tamanhoDaSubArvore* é responsável por calcular e retornar o tamanho - número de elementos - da subárvore com raiz no nó fornecido como parâmetro. Ele é usado em diversos métodos da classe para dar suporte a operações relacionadas à estrutura da árvore.

Algoritmo 14: Tamanho da Sub Árvore

Entrada: No riaz da sub árvore
Saída : Quantidade de elementos na sub árvore

```
1 Método tamanhoDaSubArvore(no)
2     se no = null então
3         retorna 0;
4     retorna no.tamanhoEsquerda + no.tamanhoDireita + 1;
```

2.2.10 Posição

O método *posicao* tem como propósito determinar a posição relativa de um valor específico na árvore em relação aos demais valores, seguindo uma travessia em ordem simétrica. O processo se inicia na raiz da árvore, onde o valor do nó atual é comparado com o valor procurado. Se o valor procurado for menor que o do nó atual, o método recursivamente chama *posicaoRecursivo* para a subárvore cuja raiz é o filho à esquerda do nó atual. Se o valor procurado for maior, a chamada recursiva ocorre na subárvore à direita. Essa recursão continua até que o valor desejado seja encontrado ou até que seja verificado que o valor não está presente na árvore.

Algoritmo 15: Encontrar a posição ocupada pelo elemento de valor n

Entrada: Um inteiro n
Saída : A posição ocupada pelo elemento de valor n

```
1  Método posicao(n)
2      retorna posicaoRecursivo(raiz, n);
3  Método posicaoRecursivo(no, n)
4      se no = null então
5          retorna -1
6      se no.valor > n então
7          posicaoEsquerda ← posicaoRecursivo(no.esquerda, n);
8          se posicaoEsquerda ≠ -1 então
9              retorna posicaoEsquerda;
10     senão se no.valor = n então
11         retorna tamanhoDaSubArvore(no.esquerda) + 1;
12     senão
13         posicaoDireita ← posicaoRecursivo(no.direita, n);
14         se posicaoDireita ≠ -1 então
15             retorna tamanhoDaSubArvore(no.esquerda) + 1 + posicaoDireita;
16     retorna -1
```

2.2.11 Mediana

Neste método calculamos a mediana da árvore e para isso utilizamos do método auxiliar *tamanhoDaSubArvore* para calcular a quantidade total de nós presentes na árvore e a partir daí checar se a quantidade é par ou ímpar. Caso possua um número par de elementos o menor dentre os dois elementos medianos é escolhido, caso seja ímpar simplesmente é o elemento do meio. Utilizamos o método *enesimoElemento* para encontrar o valor que está naquela posição e o retornamos

Algoritmo 16: Encontrar o elemento que contém a mediana da ABB

Saída : retorna o elemento que contém a mediana da ABB

```
1  Método mediana()
2      numeroDeElementos = tamanhoDaSubArvore(raiz);
3      se numeroDeElementos % 2 = 0 então
4          retorna enesimoElemento(numeroDeElementos/2);
5      senão
6          retorna enesimoElemento((numeroDeElementos+1)/2);
```

2.2.12 É Cheia

Este algoritmo verifica se a árvore é do tipo cheia e para isso utiliza-se do fato que uma ABB cheia possui uma quantidade de nós igual a $2^{altura-1}$. Para fazer essa verificação utilizamos de alguns cálculos simples e do método auxiliar *tamanhoDaSubArvore* passando como parâmetro a raiz da árvore.

Algoritmo 17: Verifica se a ABB é cheia ou não

Saída : Verdadeiro se a árvore é cheia e falso se não

```
1 Método ehCheia()
2     se raiz = null então
3         retorna falso;
4     resultado = 2^raiz.altura;
5     se resultado -1 = tamanhoDaSubArvore(raiz) então
6         retorna verdadeiro;
7     senão
8         retorna falso;
```

2.2.13 É completa

No algoritmo de *ehCompleta* usamos o seguinte lema: "seja T uma árvore binária completa com n nós e altura h. Então, $2^{h-1} \leq n \leq 2^h$–1". Com base nisso, para fazer a verificação utilizamos também o *tamanhoDaSubArvore* para saber a quantidade nós n na árvore.

Algoritmo 18: Verifica se a ABB é completa ou não

Saída : Verdadeiro se a árvore é completa e falso se não

```
1 Método ehCompleta()
2     se raiz = null então
3         retorna falso;
4     numeroDeNos = tamanhoDaSubArvore(raiz);
5     se 2^(raiz.altura-1) ≤ numeroDeNos e numeroDeNos ≤ 2^raiz.altura − 1 então
6         retorna verdadeiro;
7     senão
8         retorna falso;
```

2.2.14 Pré-Ordem

O algoritmo *preOrdem* tem como retorno uma *String* com os valores dos nós seguindo o padrão de percorrimento pré-ordem e para isso foi utilizado um método auxiliar *preOrdemRecursivo* que é responsável ir percorrendo a árvore e adicionando os valores a *String*. Além disso, foi passado como parâmetro para método auxiliar o nó raiz e a *String* para armazenamento.

Algoritmo 19: Sequência de Percorrimento em Pré-Ordem.

Saída : A sequência de percorrimento da ABB em pré-ordem

```
1 Método preOrdem()
2     preOrdemRecursivo(raiz, resultado);
3     retorna resultado;
4 Método preOrdemRecursivo(no, resultado)
5     se no ≠ null então
6         resultado.acrescenta(no.valor)
7         resultado.acrescenta("  ");
8         preOrdemRecursivo(no.esquerda, resultado);
9         preOrdemRecursivo(no.direita, resultado);
```

3 RESULTADOS E DISCUSSÕES

Nesta seção são apresentadas as análises assintóticas feitas sobre os algoritmos implementados de uma árvore binária de busca.

3.1 Busca

Neste projeto, a solução para o problema da busca, apresentada pelo Algoritmo 3, inicia o percurso a partir da raiz e desce através da árvore através das chamadas recursivas. Para cada uma dessas, o algoritmo realiza um número constante de comparações, de modo que a complexidade é determinada pelo número de chamadas.

Dessa forma, o melhor caso para este algoritmo ocorre quando o valor a ser procurando é encontrado na raiz da árvore. Nessas condições, o algoritmo realiza apenas uma quantidade fixa de comparações, resultando em uma complexidade de $O(1)$.

Já o pior caso acontece na situação em que o valor buscado não corresponde a nenhum nó da árvore ou está em uma folha que se encontra na maior distância em relação a raiz. Assim, no pior caso a quantidade de chamadas recursivas depende da altura da árvore. Dessa forma, para uma o pior caso do algoritmo é $O(h)$ em que h é a altura da árvore. Vale ressaltar que para árvores completas, essa altura é dada por $h = \lfloor \log n \rfloor + 1$ e para árvores degeneradas, $h = n$, sendo n a quantidade de nós da árvore (SZWARCFITER; MARKENZON, 2015).

3.2 Inserir

O melhor caso da inserção ocorre quando a posição correta para o novo nó é sendo o filho da raiz. Nesse caso, são realizadas algumas comparações e uma atribuição do novo nó (Algoritmo 6), de forma que a complexidade de melhor caso é dada por $O(1)$.

Para cada chamada recursiva do método, um número constante de operações é realizada, incluindo comparações entre o valor a ser inserido e o valor do nó atual analisado, e a chamada de métodos para a atualização dos campos do nós, que também realizam uma quantidade fixa de operações, como pode ser observado no Algoritmo 4 e Algoritmo 5. Nesse sentido, o pior caso para o algoritmo acontece quando o local para inserir o novo valor corresponde ao filho da folha mais distante da raiz da árvore e é preciso percorrer o caminho de tamanho da altura da árvore, de modo que a complexidade no pior caso é $O(h)$ em que h representa a altura da árvore.

3.3 Remover

No melhor caso, o valor a ser removido é filho da raiz e não possui nenhum filho. Nessa situação, o nó é encontrado após um número constante de comparações entre os

valores e a complexidade de melhor caso é $O(1)$.

Quanto ao pior caso, este ocorre quando o nó a ser removido é uma folha localizada no nível mais profundo da árvore. Nessas situações, o algoritmo percorre um caminho de tamanho correspondente a altura da árvore, de modo similar a inserção e a complexidade de pior caso é $O(h)$ considerando h a altura da árvore binária de busca.

3.4 Imprimir

De acordo com a formatação desejada, o método *imprimirArvore* pode chamar o método *imprimirFormato1* ou *imprimirFormato2*.

Ambos os algoritmos visitam e imprimem todos os nós da árvore de forma recursiva independente do caso. No caso de *imprimirFormato2*, para cada chamada recursiva, ele realiza uma quantidade constante de operações para formatar a saída. Portanto, sendo n a quantidade de nós da árvore, tanto no melhor quanto no pior caso, possui complexidade $O(n)$.

Por outro lado, em *imprimirFormato1*, cada chamada recursiva inclui o cálculo da quantidade de dígitos do valor do nó atual (Algoritmo 10). Considerando que d representa a quantidade de dígitos do valor e n a quantidade de nós da árvore, a complexidade em ambos os cenários, melhor e pior, é de $O(n \times d)$.

3.5 Média

O algoritmo em questão realiza uma chamada ao método *buscar* para localizar o nó a partir do qual a média será calculada. Posteriormente, é realizada uma comparação, uma atribuição e uma divisão, ou seja, uma quantidade constante de operações.

Dessa forma, a complexidade do método *média* é determinada pelo *buscar*, que, conforme visto anteriormente, possui complexidade $O(1)$ no melhor caso e $O(h)$ - em que h corresponde a altura da árvore - no pior caso.

3.6 Énesimo Elemento

O método *enesimoElemento* utiliza apenas o método *enesimoElementoRecursivo* e, portanto, a complexidade deste depende do segundo método.

O melhor caso ocorre quando o enésimo valor é a raiz da árvore, de modo que o *enesimoElementoRecursivo* é chamado somente uma vez e realiza uma quantidade constante de operações. Logo, a complexidade de melhor caso é $O(1)$,

O pior caso ocorre quando o enésimo elemento é uma folha que está localizada na maior profundidade em relação à raiz da árvore. Como resultado, o algoritmo teria uma quantidade de chamadas recursivas equivalente a altura h da árvore e a complexidade de pior caso seria $O(h)$. Nesse sentido, é importante destacar que quando a árvore está

balanceada, a sua altura é dada por $h = \lfloor \log n \rfloor + 1$ e para árvores degeneradas em que cada nó possui exatamente um filho, $h = n$, sendo n a quantidade de nós da árvore.

3.7 Posição

A complexidade do método *posicao* utiliza apenas o método *posicaoRecursivo* e, portanto, a complexidade deste depende desse último método.

O melhor caso ocorre quando temos que o valor procurado está localizado na raiz da árvore e o algoritmo realiza algumas operações de comparação e de aritmética, ou seja, a complexidade nesse caso é $O(1)$.

No pior caso temos que o valor desejado está localizado mais profundamente na árvore, de forma que a complexidade é influenciada pela altura h da árvore e a complexidade é $O(h)$. Isso acorre pois a cada chamada recursiva o método *posicaoRecursivo* desce um nível na árvore.

3.8 Mediana

A complexidade do método *mediana* é dominada principalmente pelo método *enesimoElemento*, pois as outras operações são constantes. Portanto a complexidade no melhor caso é $O(1)$ e no pior caso é $O(h)$ com h sendo a altura da árvore.

3.9 É Cheia

O método *ehCheia* envolve algumas operações aritméticas, comparações e a chamada do método *tamanhoDaSubArvore* que é constante, pois só faz operações aritméticas. Portanto podemos dizer que a complexidade do método é constante em todos os casos, ou seja $O(1)$.

3.10 É Completa

A complexidade do método *ehCompleta* é constante $O(1)$ em qualquer cenário, isto é, tanto no melhor quanto no pior caso. Isso porque as operações envolvidadas para verificar se é a ABB é completa ou não foram todas contantes.

3.11 Pré-Ordem

Em qualquer cenário, a complexidade do algoritmo *preOrdem* é $O(n)$, sendo n a quantidade de nós na árvore. Essa complexidade se deve a utilização do método *preOrdemRecursivo* que é responsável por passar por todos os nós da árvore para poder adicionar seus valores a *String* de saída.

4 CONCLUSÃO

A análise assintótica é um método bastante utilizado quando é necessário verificar a eficiência de algoritmos para que seja possível compará-los e aplicá-los de modo adequado na resolução de problemas. Com base nisso, uma árvore binária de busca estendida foi implementada, com métodos tradicionais, como buscar, inserir e remover, bem como métodos para retornar o n-ésimo elemento, retornar a posição ocupada em ordem simétrica por um elemento dado, calcular a mediana, calcular a média, verificar se a árvore é cheia ou completa, imprimir em a árvore em pré-ordem e em diferentes formatações. A implementação desses algoritmos foi feita de modo a tentar reduzir seu tempo de execução a partir da adição de novas informações aos nós da árvore.

Nesse contexto, foi realizada a análise assintótica de todos os algoritmos apresentados e os resultados obtidos indicaram que alguns algoritmos tiveram sua complexidade melhorada, incluindo o algoritmo *ehCheia* e *ehCompleta* que passaram a ter complexidade $O(1)$ para o melhor e pior caso devido a inserção do campo altura nos nós. Além disso, a implementação de outros algoritmos foi facilitada pelos atributos *tamanhoEsquerda* e *tamanhoDireita*, como foi o caso do *enesimoElemento*.

À vista disso, o desenvolvimento e resultados do trabalho atenderam os objetivos propostos, dado que proporcionaram uma compreensão mais aprofundada sobre o funcionamento de uma árvore de busca binária e do seu camportamento para tamanhos de entrada consideravelmente grandes.

Referências

DROZDEK, A. **Data Structures and Algorithms in C++**. [S.l.]: Cengage Learning, 2013.

GOODRICH, M. T.; TAMASSIA, R.; MOUNT, D. M. **Data Structures and Algorithms in C++**. [S.l.]: ohn Wiley Sons, Inc, 2011.

SEDGEWICK, R. **Algorithms in C**. New York: Addison-Wesley Publishing Company, 1998. 657 p.

SZWARCFITER, J. L. S.; MARKENZON, L. **Estruturas de Dados e Seus Algoritmos**. Rio de Janeiro: LTC — Livros Técnicos e Científicos Editora Ltda, 2015. 236 p.