UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
Ian Gabriel Silva Dias
Emanuel Kywal Pinto Cabral Filho

# Análise de complexidade assintótica - Árvore Binária de Busca

Docente: Sidemar Fideles Cezario

NATAL - RN | 13 de Novembro de 2023

# Contents

# 1 Introdução

Neste trabalho desenvolvemos uma Árvore Binária de Busca (ABB) que realiza operações básicas de inserção, remoção e busca de elementos de chave inteira. Esta árvore não permite elementos repetidos. Também foram implementados métodos de impressão e funções auxiliares, como por exemplo: encontrar a mediana dos elementos, calcular a média de todos eles, dizer se a árvore é completa ou não, entre outros.

# 2 Metodologia

Para este trabalho utilizamos a linguagem Java. Foram criadas duas classes, a classe principal `BinarySearchTree` que representa a ABB e a classe `Node` que representa o nó da árvore. Na classe nó estamos armazenando o valor que ele representa (inteiro), o nó que é seu filho esquerdo, e outros atributos auxiliares.

Nesta seção detalharemos como foram feitos cada um dos métodos especificados, começando pelos atributos auxiliares da classe Node, apresentados os objetivos que basearam esse design de projeto, assim como seus impactos na implementação da ABB. Consecutivamente, abordaremos as operações básicas e apenas no final examinaremos os demais métodos utilitários.

## 2.1 Atributos auxiliares

Inicialmente, detalhando os conceitos que fundamentaram a ABB e seus métodos, temos que, além dos atributos essenciais, os atributos `rightSize`, `leftSize`, `height` e `subTreeSum` foram adicionados a classe `Node` com o objetivo de reduzir a complexidade de alguns dos métodos utilitários. Dessa forma, temos os seguintes papéis para cada atributo:

- `rightSize` e `leftSize`:

  Fazem referência à quantidade de nós das subárvores esquerda e direita do nó em questão. Utilizados em múltiplos métodos;

- `height`:

  A altura do nó em questão; por definição é a quantidade de nós até o descendente mais distante, sendo a altura mínima 1. Utilizado para diminuir a complexidade dos métodos `ehCheia()` e `ehCompleta()` para $O(1)$;

- `subTreeSum`:

  Soma do valores da subárvore do nó em questão. Utilizado para reduzir a complexidade do método `media()` para $O(1)$

Além disso, é importante reiterar que esses atributos são constantemente atualizados pelas operações básicas de inserção e remoção, o que será detalhado mais a frente.

## 2.2 Operações básicas

Este tópico detalha os métodos de busca, inserção e remoção da ABB, além dos métodos de atualização dos atributos auxiliares. Desse modo, é responsável por apresentar os algoritmos mais fundamentais, responsáveis por organizar e restruturar a ABB continuamente.

### 2.2.1 Busca

A operação de busca recebe um valor e um nó, que por padrão é a raíz. Caso o valor recebido é o valor do nó recebido, então encontramos o elemento. Caso contrário, comparamos o valor recebido com o valor do nó para decidir se vamos para a subárvore esquerda ou direita, conforme código abaixo.

Para medir a complexidade do algoritmo, podemos notar que a cada chamada recursiva da operação, a altura do nó recebido reduz em pelo menos 1. O algoritmo para, no pior caso, quando encontra uma folha. Portanto a complexidade é $O(h)$, onde $h$ é a altura da árvore. No caso de uma árvore zigue-zague a altura é $O(n)$ onde $n$ é a quantidade de nós na árvore.

```java
private Node auxFind(Node curr, int key) {
    if(curr.getValue() == key) {
        return curr;
    }

    if(curr.getLeft() != null && key < curr.getValue()) {
        return auxFind(curr.getLeft(), key);
    }

    if(curr.getRight() != null && key > curr.getValue()) {
        return auxFind(curr.getRight(), key);
    }

    return null;
}
```

Figure 1: Código de busca em ABB

### 2.2.2 Remoção

O método recebe três argumentos: O nó raiz da subárvore a ser considerada, o pai deste nó, e a chave para ser removida. Inicialmente os dois primeiros valores são `raiz` e `null`.

Para a realização da remoção, o método vai fazendo a busca na árvore de forma parecida com a busca padrão, até o valor da raíz ser igual ao da chave. Quando isso acontecer, há três situações possíveis:

- O elemento é uma folha

  Neste caso, podemos eliminar o nó diretamente. Atualizamos o valor do nó filho do pai para null. Comparando o valor da chave com o do pai, podemos detectar se o este é o filho esquerdo ou direito e assim realizar a remoção corretamente, conforme código 2.

- O elemento possui uma subárvore vazia

  Também podemos remover esté nó diretamente. Basta fazer com que o pai "pule" este nó e assim podemos ignorá-lo.

  O código em 3 mostra este caso quando o nó tem apenas a subárvore direita. O código é similar para quando o nó tem apenas a subárvore esquerda.

- O elemento possui as duas subárvores.

A estratégia é encontrar o antecessor do nó na ordem simétrica. Isto é, encontrar o nó mais a direita da subárvore esquerda. Após achar este nó, substituimos seu valor no nó atual, e agora queremos remover este novo nó da subárvore esquerda. Este nó é garantido satizfazer um dos dois casos anteriores.

Podemos ver isto no código 4. Nele é chamado uma função auxiliar `getRightmost` (Código 5) que pega o nó mais a direita da subárvore esquerda.

Sobre a complexidade do método, podemos fazer uma observação similar ao que foi feito no método de busca e concluir que a complexidade do algoritmo é $O(h)$. Pois no pior caso o nó mais profundo possível será removido e todo o percusso até ele terá de ser feito.

```
1  if(curr.getLeft() == null && curr.getRight() == null) {
2      // Caso 1 : eh folha
3      if(parent == null) {
4          root = null;
5      } else {
6          if(parent.getValue() > key) {
7              parent.setLeft(null); // Era filho esq.
8          } else {
9              parent.setRight(null); // Era filho dir.
10         }
11     }
12 }
```

Figure 2: Remoção em folha

```java
if (curr.getLeft() == null) {
    // Caso 2 : Somente possui a subarvore direita
    if(parent == null) {
        root = root.getRight();
    } else {
        if(parent.getValue() > key) {
            parent.setLeft(curr.getRight());
        } else {
            parent.setRight(curr.getRight());
        }
    }
}
```

Figure 3: Remoção em nó com a subárvore esquerda vazia.

```java
private boolean auxRemove(Node curr, Node parent, int key) {
    // [...]
    // Caso 4 : Possui as duas subarvores
    int newValue = getLeftmost(curr.getRight());
    curr.setValue(newValue);
    auxRemove(curr.getRight(), curr, newValue);
    // [...]
}
```

Figure 4: Remoção em nó com ambas as subárvores preenchidas.

```java
private int getRightmost(Node n) {
    if(n.getRight() == null) return n.getValue();
    return getRightmost(n.getRight());
}
```

Figure 5: Pega nó mais a direita da subárvore.

### 2.2.3 Inserção

O método responsável pela inserção recebe um número inteiro e retorna um booleano indicado se a inserção foi bem sucedida ou não. Inicialmente este método checa se a árvore está vazia, ou seja, se sua raiz é nula, sendo nula ele a inicializará com o inteiro recebido como argumento (Figura 6). Caso contrário, chamará um método auxiliar responsável por percorrer a árvore recursivamente, enquanto procura pelo lugar adequado para realizar a inserção.

```java
public boolean insert(int value) {
  if(root == null) {
    root = new Node();
    root.setValue(value);
    return true;
  } else {
    return auxInsert(root, value);
  }
}
```

Figure 6: Trecho responsável por verificação de inserção na árvore vazia.

O método auxiliar do procedimento de inserção recebe um nó e o valor inteiro a ser inserido, também retornando um booleano. A partir disso, o inteiro é comparado com os valores da árvore e pode ser analisado a partir de 3 casos diferentes (Código 7).

- chave inteira $>$ valor do nó atual:

  Caso não exista um nó à direita, um novo nó será criado com a chave e o método retornará `true`. Caso já exista um nó à direita, o método auxilar será chamado com este nó como parâmetro;

- chave inteira $<$ valor do nó atual:

  Caso não exista um nó à esquerda, um novo nó será criado com a chave e o método retornará `true`. Caso já exista um nó à esquerda, o método auxilar será chamado com este nó como parâmetro;

- chave inteira $=$ valor do nó atual:

  A inserção não pode acontecer, já que já existe um nó com essa chave na árvore. O método retorna `false`;

Portanto, tendo esse processo de comparação em mente, temos que no pior caso será necessário executar $h$ passos, onde $h$ é a altura da árvore, já que a chave inteira será comparada com 1 nó de cada nível, percorrendo todo o caminho da raiz até a folha mais profunda. Dessa forma, temos que a complexidade desse algoritmo pertence ao conjunto $O(h)$ e que de forma similar ao algorimos de busca e remoção, quando utilizadas árvores zigue-zague, a complexidade se torna $O(n)$, sendo n a quantiade de nós da árvore.

```java
private boolean auxInsert(Node curr, int value) {
    try {
      Node rightNode = curr.getRight();
      Node leftNode = curr.getLeft();

      if(value > curr.getValue()) {
        if(rightNode == null) {
          Node newRightNode = new Node();
          newRightNode.setValue(value);
          curr.setRight(newRightNode);
          return true;
        } else {
          return auxInsert(rightNode, value);
        }
      }
      else if(value < curr.getValue()) {
        if(leftNode == null) {
          Node newLeftNode = new Node();
          newLeftNode.setValue(value);
          curr.setLeft(newLeftNode);
          return true;
        } else {
          return auxInsert(leftNode, value);
        }
      }
      else return false;
    } finally {
      updateHeight(curr);
      updateSubtreeSize(curr);
      updateSubtreeSum(curr);
    }
  }
```

Figure 7: Trecho responsável por determinar a posição correta para a inserção.

### 2.2.4 Atualização dos atributos

Utilizados ao final dos métodos inserir e remover, assegurados de executar mesmo após lançamento de exceções por meio da estrutura de tratamento de exceções `try-finally`, os métodos de atualização dos atributos são responsáveis por continuamente adequarem esses dados à realidade

da árvore em questão.

```
} finally {
      updateHeight(curr);
      updateSubtreeSize(curr);
      updateSubtreeSum(curr);
    }
```

Figure 8: Trecho responsável por atualizar os atributos.

A atualização dos atributos auxiliares baseia-se em um comportamento indutivo, estabelecendo um caso base e adequando os atributos do nó em questão a partir das informações disponibilizadas pelos seus filhos. Vejamos em detalhe o algoritmo de cada atributo:

- `updateHeight(curr)` - código 9:

  Estabelecendo como caso base que toda folha tem altura 1, checamos 3 casos diferentes: (i) O nó atual possui apenas filho esquerdo; nesse caso a altura é a altura desse filho esquerdo mais 1; (ii) O nó atual possui apenas filho direito; nesse caso a altura é a altura desse filho direito mais 1; (iii) O nó atual possui filho direito e esquerdo; a altura se define como a maior altura dentre as alturas dos filhos, mais 1.

```
private void updateHeight(Node curr) {
    if(curr.getRight() == null && curr.getLeft() != null) {
      curr.setHeight(curr.getLeft().getHeight() + 1);
    } else if (curr.getLeft() == null && curr.getRight() != null){
      curr.setHeight(curr.getRight().getHeight() + 1);
    } else if (curr.getLeft() != null && curr.getRight() != null) {
      curr.setHeight(
          1 + Math.max(
              curr.getRight().getHeight(),
              curr.getLeft().getHeight()
          )
      );
    }
  }
```

Figure 9: Trecho responsável por atualizar a altura do nó.

- `updateSubTreeSize(curr)` - código 10:

Na atualização do tamanho da subárvore temos como caso base que um nó vazio/nulo deve possuir tamanho 0. A partir disso, analisamos separadamente as subárvores esquerda e direita, definindo seus tamanhos a partir da soma dos tamanhos das suas respectivas subárvores mais 1.

```
1  private void updateSubtreeSize(Node curr) {
2      if(curr.getRight() == null) {
3        curr.setRightSize(0);
4      } else {
5        Node right = curr.getRight();
6        curr.setRightSize(right.getLeftSize() + right.getRightSize() + 1);
7      }
8
9      if(curr.getLeft() == null) {
10       curr.setLeftSize(0);
11     } else {
12       Node left = curr.getLeft();
13       curr.setLeftSize(left.getLeftSize() + left.getRightSize() + 1);
14     }
15   }
```

Figure 10: Trecho responsável por atualizar o tamanho da subárvore do nó em questão.

- `updateSubTreeSum(curr)` - código 11:

  Seu caso base está no fato de que toda folha já possui a soma de sua subárvore calculada, isto é, seu próprio valor. Assim, e de modo similar a altura, temos a definição da soma da subárvore a partir de 3 casos: (i) O nó atual possui apenas filho direito; nesse caso a soma da subárvore se dá pela soma da subárvore do filho direito mais o valor do próprio nó; (ii) O nó atual possui apenas filho esquerdo; nesse caso a soma da subárvore se dá pela soma da subárvore do filho esquerdo mais o valor do próprio nó; (iii) O nó atual possui filho direito e esquerdo; desse modo a soma da subárvore define-se como a soma das somas das subárvores esquerda e direta mais 1.

```
1  private void updateSubtreeSum(Node curr) {
2      if(curr.getLeft() == null && curr.getRight() != null) {
3        curr.setSubtreeSum(curr.getRight().getSubtreeSum() + curr.getValue()
       );
4      }
5      else if(curr.getRight() == null && curr.getLeft() != null) {
6        curr.setSubtreeSum(curr.getLeft().getSubtreeSum() + curr.getValue())
       ;
7      } else if(curr.getLeft() != null && curr.getRight() != null) {
8        curr.setSubtreeSum(curr.getLeft().getSubtreeSum() + curr.getRight().
       getSubtreeSum() + curr.getValue());
9      }
10   }
```

Figure 11: Trecho responsável por atualizar a soma dos valores da subárvore do nó em questão.

## 2.3 Métodos utilitários

### 2.3.1 n-ésimo elemento

Este método recebe um número $n$ e retorna o $n$-ésimo elemento da árvore considerando a ordem simétrica. Se assume que o $n$ recebido está sempre $1$ e o tamanho da árvore.

Para resolver este problema, adicionamos dois atributos na classe `Node` para dizer quantos nós existem nas subárvores esquerda e direita. Estes atributos são inicializados como 0 e atualizados corretamente na inserção e na remoção. Com estes atributos conseguimos identificar se o $n$-ésimo nó está a esquerda ou a direita do nó atual.

O método é chamado inicialmente a partir da raíz. Seja $e$ o tamanho da subárvore esquerda do nó atual. Se o $n$ recebido é igual $e + 1$, então o nó atual é a resposta. Se for menor, então procuramos o $n$-ésimo nó na subárvore esquerda. Caso contrário procuramos o $(n - e - 1)$-ésimo nó na subárvore direita, conforme código 12.

Da mesma forma que os métodos anteriores, podemos concluir que a complexidade deste algoritmo é $O(h)$.

```
1  private int auxEnesimo(Node n, int pos) {
2      if(n.getLeft() == null && n.getRight() == null) {
3          return n.getValue();
4      }
5
6      if(pos == n.getLeftSize() + 1) {
7          return n.getValue();
8      }
9
10     if(pos <= n.getLeftSize()) {
11         return auxEnesimo(n.getLeft(), pos);
12     }
13
14     int newPos = pos - (n.getLeftSize() + 1);
15     return auxEnesimo(n.getRight(), newPos);
16 }
```

Figure 12: Enésimo elemento da árvore.

### 2.3.2 Posição de um elemento

Dada uma chave $x$, este método retorna a posição de $x$ na árvore considerando a ordem simétrica. Caso $x$ não exista, a função retorna -1.

Para resolver este problema utilizamos uma função auxiliar que recebe um nó raíz e retorna a resposta para o problema apenas considerando a subárvore enraizada por este nó. Esta função também recebe a chave a ser buscada e um acumulador que nos diz quantos valores vem antes de $x$ até então. Para cada chamada, há as seguintes possibilidades:

- O valor da raíz é a chave recebida.

  A posição da raíz é $e + 1 + acc$ onde $e$ é o tamanho da sua subárvore esquerda e $acc$ é o acumulador recebido pela função.

- O valor da raíz é maior que a chave recebida

  Resolvemos o problema para a subárvore esquerda. Note que nenhum novo nó ficou antes de $x$ na ordem simétrica ainda.

- O valor da raíz é menor que a chave recebida

  Resolvemos o problema para a subárvore direita. $e + 1$ novos nós ficaram antes de x na ordem simétrica. Portanto, o acumulador aumenta na mesma quantidade.

- O valor é maior/menor mas a subárvore correspondente é vazia.

  O valor não existe na árvore e podemos retornar -1.

O código 13 descreve o processo acima. Novamente, pelo mesmo motivo das seções anteriores, temos que a complexidade é $O(h)$.

```
1  private int auxPosicao(Node curr, int key, int acc) {
2      if(curr.getValue() == key) {
3          return curr.getLeftSize() + acc + 1;
4      }
5
6      if(key < curr.getValue() && curr.getLeft() != null) {
7          return auxPosicao(curr.getLeft(), key, acc);
8      }
9
10     if(key > curr.getValue() && curr.getRight() != null) {
11         return auxPosicao(curr.getRight(), key, acc + curr.getLeftSize() +
       1);
12     }
13
14     return -1;
15 }
```

Figure 13: Posicao do elemento da árvore.

### 2.3.3 Mediana

Este método retorna a mediana dos elementos da árvore. Para isto pegamos o $n/2$-ésimo elemento considerando a ordem simétrica quando o quantidade de nós for par e o $(n/2+1)$-ésimo quando for ímpar. Fizemos isso apenas reutilizando a função que já foi descrita na seção 2.3.1. Trivialmente a complexidade da mediana será igual a deste algoritmo, logo, é $O(h)$.

```
1  public int mediana() {
2      int size = getSize();
3      if(size % 2 == 0) {
4          return enesimoElemento(size / 2);
5      }
6
7      return enesimoElemento(size / 2 + 1);
8  }
```

Figure 14: Mediana dos nós.

### 2.3.4 Média

Este método recebe um argumento $x$ e retorna a média aritimética dos valores dos nós da subárvore em que $x$ é raíz. Esta função lança a exceção `NullPointerException` quando o valor não é encontrado.

O valor de $x$ é buscado na árvore utilizando a operação básica de busca. Uma vez encontrado o nó, basta usar os atributos auxiliares de `subtreeSum`, `rightSize` e `leftSize` para encontrar a soma dos nós e a quantidade em $O(1)$. Observe o código 15 para mais detalhes.

Como o cálculo da média em si é constante, a complexidade assintótica é dependente apenas da operação de busca. Conforme dito na seção 2.2.1, a complexidade desta operação é $O(h)$. Portanto esta também é a complexidade do método da média.

```
1  public double media(int rootValue) throws NullPointerException {
2      double sum = 0.0;
3      int nodeAmount = 0;
4      Node fakeRoot = find(rootValue);
5
6      if(fakeRoot != null) {
7          sum = fakeRoot.getSubtreeSum();
8          nodeAmount = fakeRoot.getLeftSize() + fakeRoot.getRightSize() + 1;
9      } else {
10         throw new NullPointerException("Valor inserido nao pertence a
       arvore!");
11     }
12
13     return sum/nodeAmount;
14 }
```

Figure 15: Média dos nós.

### 2.3.5 É cheia

O método `ehCheia` retorna um booleano que diz se a árvore é cheia ou não. Se uma árvore é cheia, então a quantidade de nós é

$$\sum_{i=0}^{h-1} 2^i = 2^h - 1$$

Onde $h$ é a altura da árvore. Portanto, basta verificar se a quantidade total de nós é igual a este valor.

A complexidade deste método (Código 16) é $O(1)$ pois basta checar uma igualdade. Para exponenciar os números, utilizamos o método `Math.pow` que também faz esta operação em $O(1)$.

```
1 public boolean ehCheia() {
2     int expectatedAmountNodes = (int) Math.pow(2.0, (double) root.
    getHeight()) - 1;
3     int realAmountNodes = getSize();
4
5     return expectatedAmountNodes == realAmountNodes;
6 }
```

Figure 16: Código que verifica se a árvore é cheia.

### 2.3.6 É completa

O método `ehCompleta` verifica se a árvore é completa ou não. Uma propriedade conhecida é que, seja uma árvore binária completa com $n$ nós e altura $h$ então:

$$2^{h-1} \leq n \leq 2^h - 1$$

Portanto, basta checar se a desigualdade é satizfeita. Isso foi implementado em $O(1)$. O código 17 descreve o algoritmo.

```
1 public boolean ehCompleta() {
2     int lowerBound = (int) Math.pow(2.0, (double) root.getHeight() - 1);
3     int upperBound = (int) Math.pow(2.0, (double) root.getHeight()) - 1;
4     int realAmountNodes = getSize();
5
6     return realAmountNodes >= lowerBound && realAmountNodes <= upperBound;
7 }
```

Figure 17: Código que verifica se a árvore é completa.

### 2.3.7 Pré-ordem

O método `preOrdem` imprime a árvore em pré-ordem. O algoritmo de pré-ordem é bastante conhecido. O passo a passo é

1. Imprime o nó atual
2. Imprime a subárvore esquerda em pre-ordem
3. Imprime a subárvore direita em pré-ordem

A implementação em Java que foi utilizada para descrever este algoritmo se encontra no código 18. A complexidade deste método é $O(n)$ pois todos os nós são impressos apenas uma vez.

```
1 public void preOrder(Node n) {
2     System.out.print(n.getValue() + " ");
3     if(n.getLeft() != null) preOrder(n.getLeft());
4     if(n.getRight() != null) preOrder(n.getRight());
5 }
```

Figure 18: Impressão da árvore em pré-ordem.

### 2.3.8 Imprimir árvore

O método `imprimirArvore` recebe um inteiro que só pode ter valor 1 ou 2. Ao receber o valor 1 ele imprime a árvore no formato descrito na imagem 19. Ao receber o valor 2 o método imprime a árvore no formato descrito na imagem 20.

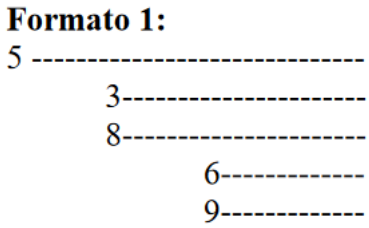


Figure 19: Formato 1 de impressão

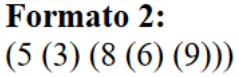


Figure 20: Formato 2 de impressão

Para a impressão do formato 1, foi construida uma função auxiliar que recebe o nó raíz da subárvore atual e acumuladores que refletem a profundidade atual do nó. O algoritmo imprime a

linha correspondente ao nó, indentando de acordo com a profundidade. Após isso, ele imprime as subárvores esquerda e direita pelo formato 1, chamando o método recursivamente. Este processo é resumido no código 21.

A impressão do formato 2 (Código 22) é similar ao algoritmo do percurso em pré-ordem. A diferença é que abrimos parênteses na primeira linha e fechamos na última, quando a função está saíndo da recursão.

Ambos os formatos realizam o percurso na árvore em pré-ordem. Conforme demonstrado na seção 2.3.7, a complexidade de ambos os métodos de impressão é $O(n)$.

```
1  private void printIndentFormat(Node curr, String indent, int depth) {
2      if(curr.getValue() < 10) {
3          System.out.print(indent + "0" + curr.getValue());
4      }
5      else System.out.print(indent + curr.getValue());
6
7      for(int i = 0; i < depth; i++) System.out.print("-");
8      System.out.println("");
9
10     if(curr.getLeft() != null) printIndentFormat(curr.getLeft(), indent +
       "    ", depth - 3);
11     if(curr.getRight() != null) printIndentFormat(curr.getRight(), indent
       + "    ", depth - 3);
12 }
```

Figure 21: Implementação do formato 1 de impressão

```
1  private void printPrecedenceFormat(Node curr) {
2      System.out.print(" (" + curr.getValue());
3
4      if(curr.getLeft() != null) printPrecedenceFormat(curr.getLeft());
5      if(curr.getRight() != null) printPrecedenceFormat(curr.getRight());
6
7      System.out.print(")");
8  }
```

Figure 22: Implementação do formato 2 de impressão

# 3 Conclusão

Com a realização deste trabalho foram adquiridos muitos conhecimentos sobre o funcionamento e a implementação de ABB's. Foram enfrentadas dificuldades em relação à implementação da remoção que não é nada trivial. Também houve dificuldades em relação à entrada/saída envolvendo leitura de arquivos em Java, que não tínhamos experiência.

Apesar das dificuldades, todos os métodos foram feitos. Além disso, diversas revisões foram feitas no código buscando otimizar os métodos. Portanto, acreditamos que os métodos estão bem otimizados, possuindo a menor complexidade possível.