UFRN - UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

IMD - INSTITUTO METRÓPOLE DIGITAL

CURSO DE TECNOLOGIA DA INFORMAÇÃO

PROJETO 01 – IMPLEMENTAÇÃO DA ÁRVORE BINÁRIA DE BUSCA (ABB)

Gabriel Costa Lima Dantas

Hugo Vinicius Da Silva Figueirêdo

Tales Vinicius De Medeiros Alves

Outubro 2022

Natal/RN

UFRN - UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

IMD - INSTITUTO METRÓPOLE DIGITAL

CURSO DE TECNOLOGIA DA INFORMAÇÃO

PROJETO 01 – IMPLEMENTAÇÃO DA ÁRVORE BINÁRIA DE BUSCA (ABB)

Relatório técnico referente ao primeiro projeto da disciplina Estrutura de Dados Básicas II, implementação de uma árvore binária de busca (ABB).

Docente: Prof. Sidemar Fideles Cezario

Outubro 2022

Natal/RN

LISTA DE FIGURAS

SUMÁRIO

1 - Introdução

Aplicações diversas necessitam buscar um determinado valor em um conjunto de dados armazenados, essa busca precisa ser feita da maneira mais eficiente possível. Nessa perspectiva, as árvores binárias oportunizam buscas mais eficientes. Dado isso, a implementação das árvores binárias de busca(ABB) apresenta uma relação de ordem entre os nós, sendo definida pela chave (raiz).

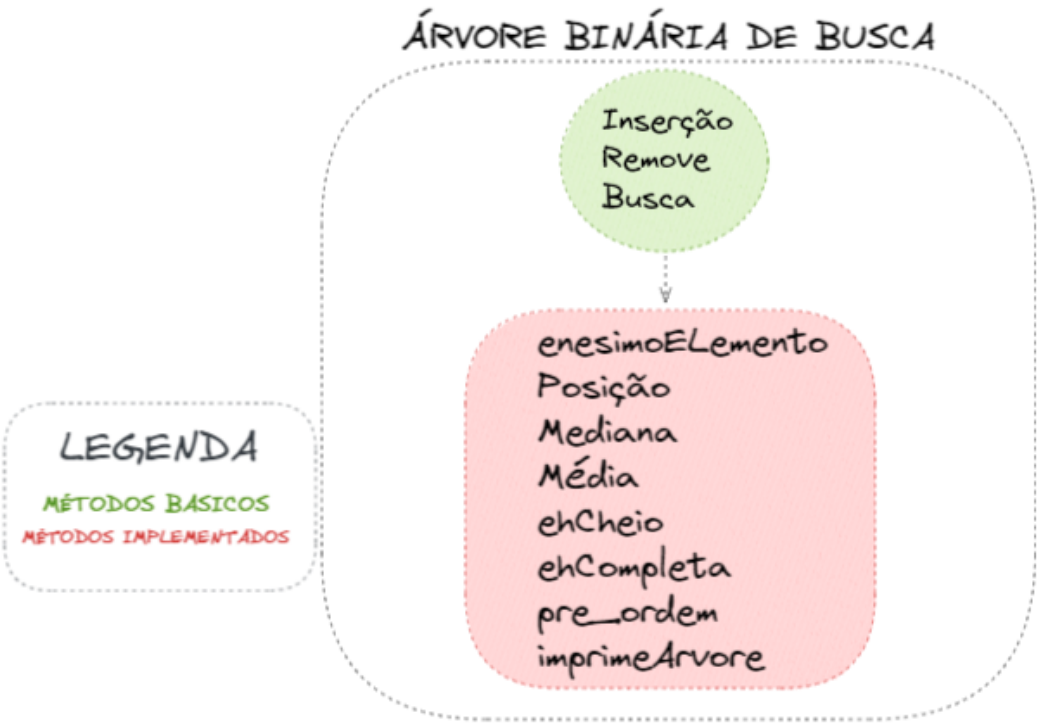
Uma ABB apresenta em sua estrutura uma ordem de informações que precisam ser seguidas: Chaves da subárvore esquerda são menores do que a raiz, chaves da subárvore da direita são maiores do que a raiz. Além disso, as subárvores da direita e esquerda são consideradas ABB, já que formam outro conjunto de elementos. Adicionalmente, um percurso *em-ordem* nessa árvore resulta na sequência de valores em ordem crescente.

No modelo apresentado, operamos a ABB com operações básicas de: *buscar* nó com determinada chave, *inserir* novo nó, *remover* nó, além dessas operações, implementamos outras operações exigidas na elaboração do projeto: *enesimoElemento*, *posição*, *mediana*, *média*, *ehCheia*, *ehCompleta*, *pre-ordem* e *imprimeArvore*. Diante do apresentado, ao longo desse relatório explicaremos cada método implementado, com a abordagem de solução, e contendo análise de complexidade assintótica desses métodos.

2 - Arquitetura da solução

A arquitetura do projeto desenvolvido contém como elementos principais os métodos básicos de coleta dos dados, com eles complementaremos a implementação dos métodos de requisitados para a Árvore Binária de Busca, com isso, conseguimos entregar os resultados através dos arquivos inseridos para leitura.

Figura 1 – Arquitetura da solução



2.1 - Descrições dos métodos

- **inserção:** método básico da ABB, responsável pela inclusão de elementos que serviram como nós e formaram a estrutura da ABB. O primeiro número inserido na função é designado como raiz da árvore, posteriormente os números inseridos passaram por uma comparação com o nó pai, caso seja menor será inserido à esquerda, se não será inserido à direita. Ressalta-se, que o algoritmo de inserção não avaliza que a árvore derivada seja balanceada ou perfeitamente balanceada.
- **busca:** no algoritmo de busca, baseamos em verificar primeiramente a existência da árvore, caso seja vazia, retornamos e não encontramos o elemento. Se a chave da raiz for igual à chave, terminamos a busca pois encontramos com sucesso. Senão, repita o processo de busca para a sub-árvore esquerda, se a chave da raiz é maior que a chave procurada; caso seja menor, o processo será realizado na sub-árvore direita.
- **remoção**: método mais complicado no quesito de comparação em relação aos outros mecanismos básicos de uma ABB, teremos três casos considerados no algoritmo de remoção. Caso o nó seja folha pode ser retirado sem problema. Se o nó possui uma sub-árvore à esquerda ou à direita, o nó raiz da subárvore supramencionada pode substituir o nó eliminado. Se não, entramos no terceiro caso, onde o nó possui duas sub-árvores, para isso, o nó cuja chave seja a menor da sub-árvore direita pode substituir o nó eliminado; ou, podemos pegar o de maior valor da sub-árvore esquerda para substituí-lo.
- **enesimoElemento:** analisamos a sequência a ser percorrida como sendo a ordem simétrica, dado isso, retornaremos o n-ésimo elemento (contado a partir do primeiro) da ABB.
- **posição:** assim como o *enesimoElemento* utilizaremos a sequência de ordem simétrica, onde retornaremos a posição ocupada pelo elemento do parâmetro do método.
- **mediana:** retorna o elemento que contém a mediana da ABB. Caso a árvore tenha uma quantidade ímpar, retornaremos o valor central do conjunto. Se a quantidade for par, retorne o menor dentre os dois elementos medianos.
- **média:** retorna a média aritmética dos nós da árvore que o parâmetro é raiz. Diante disso, o método soma todos os elementos contidos no conjunto em que o parâmetro é a raiz e divide pela quantidade deles.
- **ehCheio:** como uma árvore binária cheia possui um número exato de nós para cada altura, o algoritmo compara se o número de nós da árvore é igual a: $2^h - 1$. A árvore então é cheia se for verdade e caso contrário a árvore é falsa.
- **ehCompleta:** o algoritmo analisa se os nós não folhas, com alturas maiores de 2 possuem ambos os filhos, para cada nó que não possuir ocorre um incremento em uma variável flag, se essa variável flag for maior que 0 então a árvore não é completa.

- **pre_ordem:** imprime uma String que contém a sequência de percorrimento da ABB em pré-ordem.
- **imprimeArvore:** um comparador de casos que determina como será impresso a árvore, tendo dois formatos cadastrados (se “s” igual a 1, o método imprime a árvore no formato 1, “s” igual a 2, imprime no formato 2).

3 - Análise assintótica

Explicaremos cada método implementado e seu respectivo custo de processamento, ressalta-se que cada método tem uma análise assintótica diferente, pois são códigos diferentes.

Foi realizada a análise assintótica nas figuras a seguir utilizando o pior caso, uma árvore zigue-zague, a qual possui altura e número de elementos igual a n.

- **Inserção:**

Figura 2 - Análise Assintótica Inserção.

```
CLASSE ÁRVORE:
public void inserir(int val) {--------------------------> O(N)
    if (raiz == null) {-------------------------------------> 1
        raiz = new No(val);-------------------------------> 1
    }
    else {
        raiz.inserir(val);--------------------------------> T(n)
    }
    attElementos();-----------------------------------------> AUX(n)
}

CLASSE NÓ:
public void inserir(int val) {--------------------------> O(N)
    if (val == this.data) {---------------------------------> 1
        System.out.println( val +" já está na árvore, não pode ser inserido");
    }
    else if (val < this.data) {-----------------------------> 1
        if (this.filhoEsq == null) {-----------------------> 1
            this.filhoEsq = new No(val);-----------------> 1
            System.out.println(val + " Adicionado!");
        }
        else {
            this.filhoEsq.inserir(val);-----------------> T(n-1)
        }
    }
    else if (val > this.data) {-----------------------------> 1
        if (this.filhoDir == null) {-----------------------> 1
            this.filhoDir = new No(val);-----------------> 1
            System.out.println(val + " Adicionado!");
        }
        else {
            this.filhoDir.inserir(val);-----------------> T(n-1)
        }
    }
 }
```

- **Busca:**

Figura 3 - Análise Assintótica Busca.

```
CLASSE ÁRVORE:

private void buscar(int n) {----------------------> O(N)
    if(raiz.buscar(n) == true) {-----------------> T(n)
        System.out.println("Chave encontrada");
    }
    else {
       System.out.println("Chave não encontrada");
    }
}
 CLASSE NÓ:
 public boolean buscar(int n) {-----------------> O(N)
     boolean flag = false; ----------------------> 1
     if(n == this.data) {  ----------------------> 1
         flag = true; -------------------------------> 1
     }
     else if(n < this.data) {
         if(this.filhoEsq ≠ null) {-------------> 1
             flag = this.filhoEsq.buscar(n); -> T(n-1)
         }
     }
     else {
         if(this.filhoDir ≠ null) {-------------> 1
             flag = this.filhoDir.buscar(n); -> T(n-1)
         }
     }
     return flag;
 }
```

- **Remoção**:

Figura 4 - Análise Assintótica Remoção.

```
private boolean remover(int data) { ----------------------> O(N)
    if (raiz == null) { ------------------------------------> 1
        return false;
    }
    No atual = raiz; ---------------------------------------> 1
    No pai = raiz; -----------------------------------------> 1
    boolean filhoEsq = true; -------------------------------> 1
    while (atual.data ≠ data) { ----------------------------> N
        pai = atual; -------------------------------------> 1
        if(data < atual.data ) { -------------------------> 1
            atual = atual.filhoEsq; ----------------------> 1
            filhoEsq = true; -----------------------------> 1
        }
        else {
            atual = atual.filhoDir; ----------------------> 1
            filhoEsq = false; ----------------------------> 1
        }
    if (atual == null) { -----------------------------------> 1
        System.out.println(data+" não está na árvore, não pode ser removido!");
        return false;
    }
}
```

- **enesimoElemento:**

Figura 5 - Análise Assintótica Enésimo Elemento.

```
CLASSE NÓ
public int enesimoElemento (int n) {----------------------> O(N)
    if(n == this.quantEsq + 1) {------------------------------> 1
        return this.data;
    }
    else if(n < this.quantEsq + 1) {-------------------------> 1
        return this.filhoEsq.enesimoElemento(n); ----> T(n-1)
    }
    else {
        return this.filhoDir.enesimoElemento(
        n - (this.quantEsq + 1)); ------------------------------> T(n-1)
    }
}
```

- **Posição:**

Figura 6 - Análise Assintótica Posição.

```
CLASSE ÁRVORE:
private void posicao(int n) { ----------------------------------------> O(N)
    if(raiz.posicao(n) ≤ 0) { ----------------------------------------> 1
        System.out.println("O elemento não faz parte da arvore.");
    }
    else {
        System.out.println("O elemento "
            + n + " esta na " + raiz.posicao(n)+"ª posicao!");
    }
}

CLASSE NÓ:
public int posicao(int n) { ---------------------------------> O(N)
    if(n == this.data) { ----------------------------------------> 1
        return this.quantEsq +1;
    }
    else if(n < this.data) { ------------------------------------> 1
        if(this.filhoEsq ≠ null) { ----------------------------> 1
            return this.filhoEsq.posicao(n); ----------------> T(n-1)
        }
    }
    else{
        if(this.filhoDir ≠ null) {
            return this.quantEsq + 1 +
               this.filhoDir.posicao(n); ----------------------> T(n-1)
        }
    }
    return -1;
}
```

- **Mediana:**

Figura 7 - Análise Assintótica Mediana.

```
CLASSE NÓ:
public int mediana() { ----------------------------------------------> O(N)
    int tamanho = this.quantEsq + this.quantDir + 1; ----> 1
    if(tamanho % 2 == 1){ -----------------------------------------------> 1
        return this.enesimoElemento(tamanho/2 + 1);     ----> T(N/2)
    }
    else {
        return this.enesimoElemento(tamanho/2); -----------> T(N/2)
    }
}
```

- **Média:**

Figura 8 - Análise Assintótica Média.

```
CLASSE NÓ:
public double media(int x) {                                                    -----------------------------------------------------------> O(N)
    double soma = soma(buscarNo(x));                                            -----------------------------------------------------------> O(N)
    double quantNo = buscarNo(x).quantEsq + buscarNo(x).quantDir + 1;           ------> O(N)
    double media = soma/quantNo;                                                -----------------------------------------------------------> 1
    return media;
}
```

- **ehCheio:**

Figura 9 - Análise Assintótica ehCheia.

```
  private void ehCheia() {------> O(1)
      int nosArvore = (raiz.quantEsq + raiz.quantDir + 1); ----> 1
      int totalNos = (int)(Math.pow(2, raiz.altura) - 1); ----> 1
      if(nosArvore == totalNos) { ----> 1
          System.out.println("A árvore é cheia!");
      }
      else
          System.out.println("A árvore não é cheia!");
      }
```

- **ehCompleta:**

Figura 10 - Análise Assintótica ehCompleta.

```
CLASSE ÁRVORE:
private int completa(No atual, int flag) {                                      -----------------------------------> O(N)
    if(atual.altura > 2) {                                                      -----------------------------------------------------------> 1
        if(atual.filhoEsq == null || atual.filhoDir == null) {                  ----> 1
            flag += 1;                                                          -----------------------------------------------------------> 1
        }
        if(atual.filhoEsq ≠ null) {                                             -----------------------------------------------------------> 1
            if(atual.filhoEsq.altura > 1)                                       -----------------------------------------------------------> 1
            flag = completa(atual.filhoEsq, flag);                              ------------------------> T(N-1)
        }
        if(atual.filhoDir ≠ null) {                                             -----------------------------------------------------------> 1
            if(atual.filhoEsq.altura > 1)
                flag = completa(atual.filhoDir, flag);                          ---------------------> T(N-1)
            }
        }
    return flag;
}
```

No caso do método completo o esperado seria a árvore zigue-zague ser em fato o melhor caso. Pois, logo na raiz já existe o caso de não possuir filho, porém devido a implementação só parar no elemento filho mais a esquerda com altura menor ou igual a dois o transforma no pior caso já que precisa rodar n-2 vezes.

- **pre_ordem:**

Figura 11 - Análise Assintótica pre_ordem.

CLASSE NÓ:

```
public void preordem() {                                  -----------------------------> O(N)
    if (this ≠ null) {                                    -----------------------------> 1
        System.out.print(this.data + " ");
        if (this.filhoEsq ≠ null)                         ---------------> 1
        this.filhoEsq.preordem();                         -----------------> T(n-1)
        if (this.filhoDir ≠ null)
            this.filhoDir.preordem();                     ---------> T(n-1)
    }
}
```

4 - Conclusão

Visto isso, esse trabalho foi importante para termos uma visão mais prática sobre árvores e seus métodos, e suas vantagens e diferenças se comparado com listas. Também foi importante para aumentar o conhecimento em Java, como aprender sobre leitura de arquivos.