Disciplina *Algoritmos e Estrutura de Dados – 2 [Semestre Letivo: 2022/2]*	
Nomes dos(as) acadêmico(as) 1 – Arthur Monteiro 2 – Mateus Kalleb Cintra Bastos 3 – Pedro Manuel Rodrigues Lima de Moura	**Números de Matrícula** 1 – 201904192 2 – 202103758 3 – 202107854
Turma: INF0287A	**Professor(a):** *Wanderley de Souza Alencar*

TEMA: ÁRVORES AVL

I – INTRODUÇÃO

Nascido em 1922, Georgy Maximovich Adelson-Velsky foi um matemático e cientista da computação Russo-israelita, famoso por ser um dos pioneiros do “xadrez digital”, mas foi em conjunto com outro matemático Soviético, Evgenii Landis, que sua maior contribuição viria a surgir.

Com o avanço da tecnologia durante a Guerra Fria no século XX, e consequentemente o desenvolvimento da computação, a necessidade em se organizar informações de forma eficiente e com complexidade de tempo satisfatória para as máquinas da época foi de extrema importância.

Pensando nisso, Adelson-Velsky e Landis, desenvolveram um algoritmo com o objetivo balancear uma árvore binária de busca sem qualquer intervenção humana durante o processo. Em 1962, os matemáticos publicaram o artigo “ОДИН АЛГОРИТМ ОРГАНИЗАЦИИ ИНФОРМАЦИИ”, Um Algoritmo para a Organização da Informação, onde descreviam em 5 páginas, o funcionamento de um algoritmo nunca antes visto, um tipo de árvore binária de busca que balanceava a si mesma, nomeada pelas iniciais de seus nome, AVL.

Com complexidade de tempo ***O***(log(n)), as árvores AVL foram primordiais para o surgimento de outras árvores ainda mais avançadas. Nos dias de hoje, apesar de não serem tão comumente utilizadas, as AVL são usadas para a organização de conjuntos na memória e dicionários. Além de aplicações de banco de dados em que o número de inserções e exclusões é menor, mas há frequentes pesquisas de dados necessários.

Nesse Trabalho e Tutorial, será descrito o funcionamento e implementação deste tipo de árvore, para que você também possa aprender e executar o algoritmo em sua própria máquina.

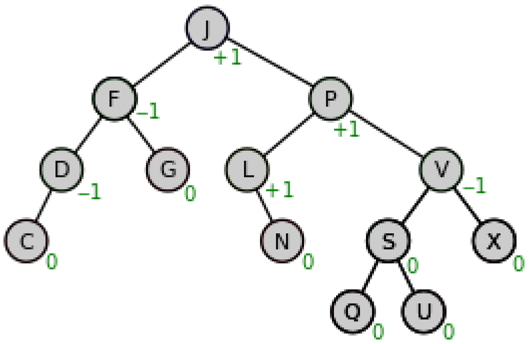
II – AS ÁRVORES AVL: DESCRIÇÃO

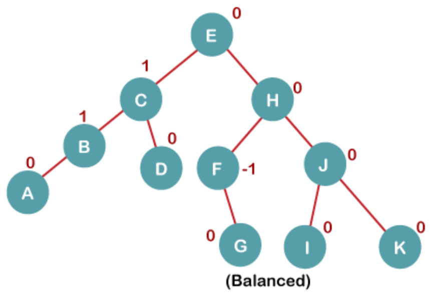
Pela sua definição informal, Uma árvore AVL é uma Árvore Binária de Busca na qual as alturas das subárvores da esquerda e da direita da raiz diferem por no máximo 1, e as subárvores da esquerda e da direita são também árvores AVL.

Além disso, é importante ressaltar, que a regra de balanceamento das Árvores AVL, atuam diretamente sobre sua altura, diferente de outros tipos de árvores, como a Red-Black cuja regra de balanceamento trata a altura apenas indiretamente. Dessa forma, o rebalanceamento da árvore (após inserções ou remoções) é feito através de operações locais na árvore. Nessa seção, serão descritos os principais conceitos e operações necessários para o funcionamento da AVL.

Um dos principais conceitos relacionados ao tipo de árvore descrito é o fator de balanceamento. Isto é, a diferença entre a altura da subárvore da direita e a altura da subárvore da esquerda ($-1 \leq (hR - hL) \leq 1$). Para uma árvore AVL, os fatores de balanceamento devem ser -1, 0 ou +1.

Veja Exemplos:





Outro conceito é o Limite de altura-h, os criadores, Adelson-Velsky e Landis, demonstraram que $\log 2(n + 1) \leq h \leq 1{,}44 \cdot \log 2(n + 2) - 0{,}328$ ou seja, o tempo de busca no pior caso é 44% pior do que em árvores já perfeitamente balanceadas, onde a busca é ***O***(log2n). Porém, em 1998, Donald Knuth, considerado o pai da análise de algoritmos, demonstrou de forma empírica que, na média, o tempo de busca está muito mais próximo do melhor caso do que do pior caso, para n de valor alto: $h = \log 2n + 0{,}25$. O que apenas prova a eficiência e genialidade do algoritmo AVL.

Quantos as operações, a **Inserção** e a **Remoção** são as que melhor demonstram o funcionamento da AVL.

A ideia geral da operação de ***inserção*** é de manter o fator de balanceamento para cada nó, fazendo uma rotação para reequilibrar a árvore quando o fator de balanceamento é alterado para ±2 em uma inserção.

Algoritmicamente isso se traduz em:

1. Inicialmente, inserir-se o nó como em qualquer ABB, o que pode desbalancear a árvore;
2. Ao inserir um nó, deve-se alterar apropriadamente os fatores de balanceamento de seus nós ascendentes;

Dessa forma, o algoritmo realiza o rebalanceamento se:

- Encontrar um nó cujo fator de balanceamento era ±1 e muda para ±2, faz-se a correção do balanceamento em torno desse nó e o algoritmo termina.

- Os fatores de balanceamento de todos os ascendentes (inclusive a raiz) eram 0 inicialmente, eles são corrigidos para −1 ou +1, não gerando violações e, portanto, terminando o algoritmo.

No geral existem 4 possíveis casos quando um nó é inserido na subárvore:

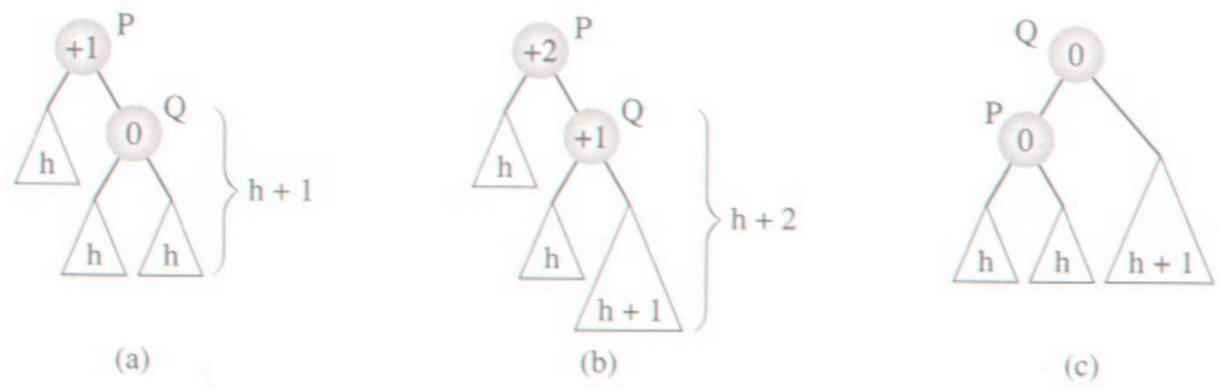
1. subárvore direita do filho à direita;
2. subárvore esquerda do filho à direita;
3. subárvore direita do filho à esquerda;
4. subárvore esquerda do filho à esquerda.

Observando-se que a Raiz da subárvore é o primeiro nó ascendente cujo fator de balanceamento se torne −2 ou +2 e os casos 3 e 4 são respectivamente simétricos, aos casos 2 e 1.

Caso 1: Subárvore direita do filho à direita.

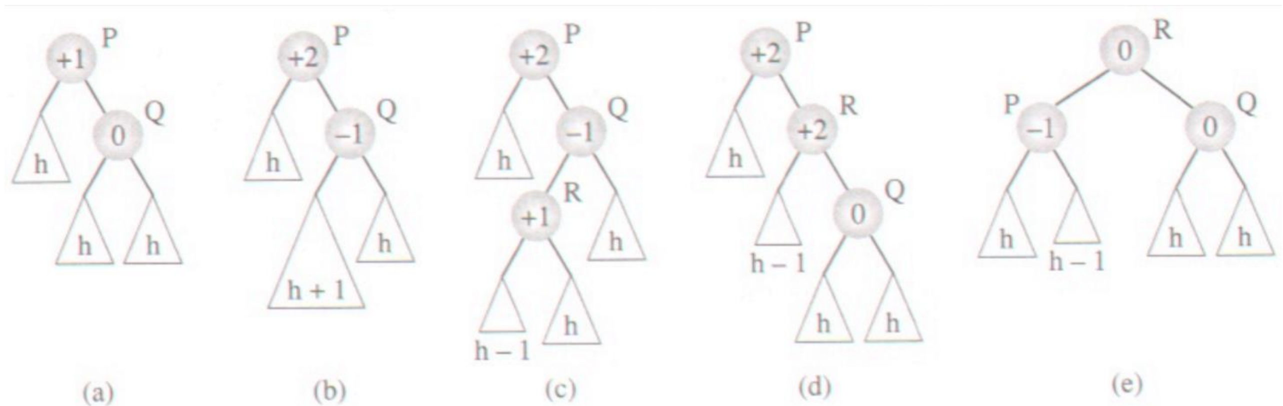
a) Árvore inicial;

b) Um nó é inserido na subárvore direita de Q, que é o filho da direita do nó P, que, por sua vez é o primeiro ascendente com fator de balanceamento ±2;
c) Árvore resultante da rotação para a esquerda em torno de P.



Caso 2: Subárvore esquerda do filho à direita.

a) Árvore inicial;
b) Um nó é inserido na subárvore esquerda de Q, que é o filho da direita de P, que, por sua vez é o primeiro ascendente com fator de balanceamento ±2;
c) Detalhamento da árvore resultante da inserção;
d) Rotação para a direita de R em torno de Q;
e) Rotação para a esquerda de R em torno de P.



Caso 3: Subárvore direita do filho à esquerda.

- Simétrico ao "Caso 2"

Caso 4: Subárvore esquerda do filho à esquerda

- Simétrico ao "Caso 1"

Agora, como exercício, tente identificar as operações e rotações realizadas na imagem a baixo:

Obs: Caso não esteja vendo o GIF, apenas ignore.

Já na Operação de ***Remoção*** a ideia geral divide-se em:

- Remove-se o nó como em qualquer Árvore Binária de Busca.

- Os fatores de balanceamento devem ser atualizados, desde o nó ascendente do nó removido até a raiz;

- Para cada nó nesse caminho cujo fator de balanceamento se tornar ±2, realizar uma rotação simples ou dupla (dependendo do caso);

- Já diferente da inserção, o rebalanceamento não para no primeiro nó com fator de balanceamento ±2, ou seja, precisa subir até a raiz. Ou seja: ***O***(log2n) rotações;

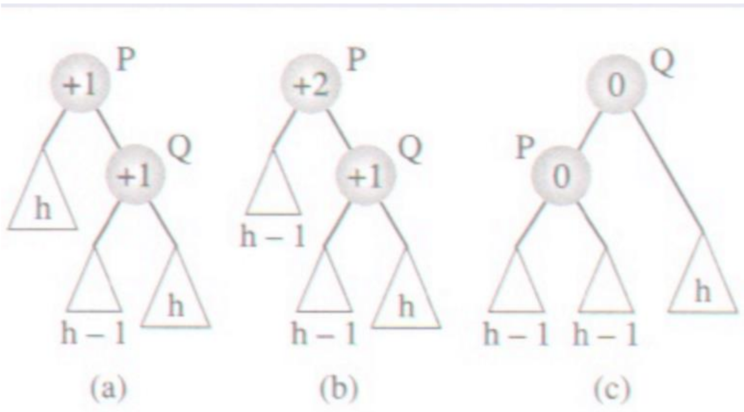
- O algoritmo é organizado em torno de 8 casos, com 4 deles sendo simétricos.

Agora, considere P sendo o próximo nó ascendente cujo fator de balanceamento supera ±1.

Caracterização do Caso 1:

- O nó foi removido da subárvore esquerda de P, e ele tinha fator de balanceamento +1;
- A raiz da subárvore direita de P (ou seja, Q) tem fator de balanceamento +1.

Ação a ser tomada: Rotação (para a esquerda) de Q em torno de P.

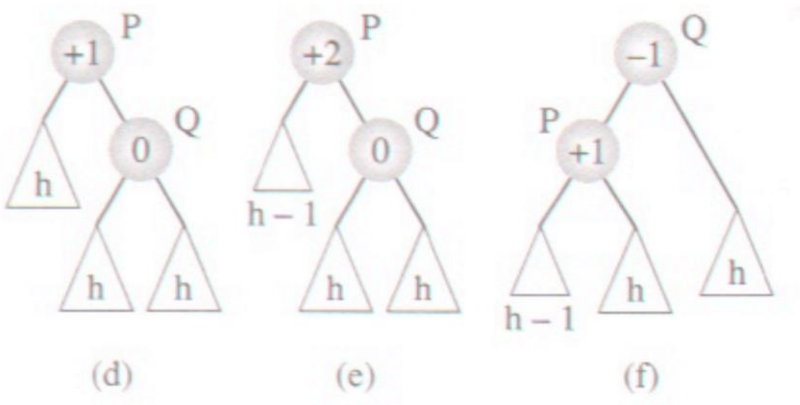


Caracterização do Caso 2: Semelhante ao Caso 1, porém, Q tem fator de balanceamento inicial igual a 0.

- O nó foi removido da subárvore esquerda de P, e ele tinha fator de balanceamento +1;
- A raiz da subárvore direita de P (ou seja, Q) tem fator de balanceamento +1.

Ação a ser tomada: Rotação (para a esquerda) de Q em torno de P (como no “Caso 1“).

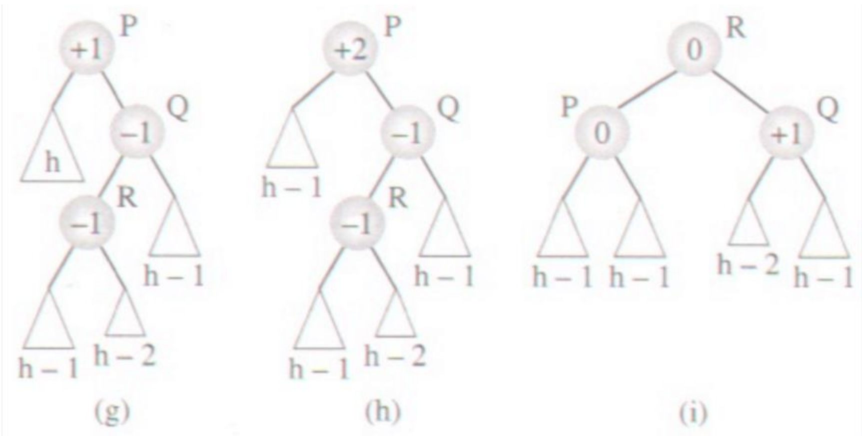
Assim, os casos 1 e 2 podem ser implementados conjuntamente, bastando distinguir se o fator de balanceamento de Q é +1 ou 0.



Caracterização do Caso 3: Q tem fator de balanceamento igual a -1 e a subárvore de Q com raiz em R tem fator de balanceamento igual a -1.

Ação a ser tomada: Rotação Dupla.

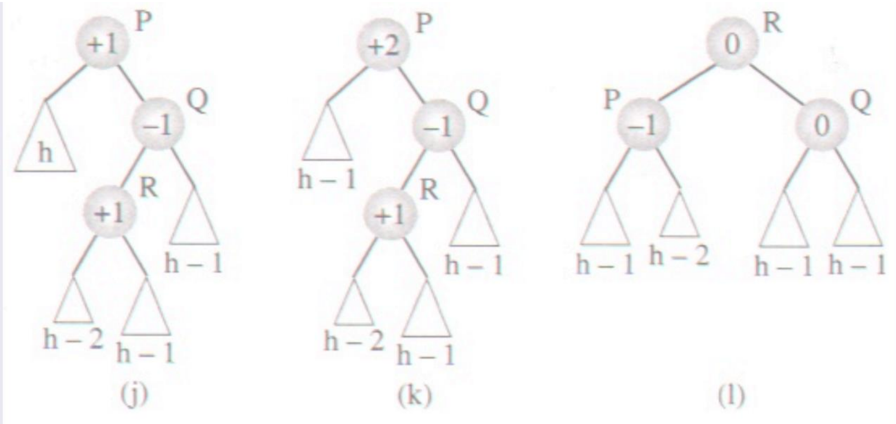
- Rotação (à direita) de R em torno de Q;
- Rotação (à esquerda) de R em torno de P.



Caracterização do Caso 4: como no Caso 3, mas o fator de balanceamento de R é igual a +1.

Ação a ser tomada: Rotação Dupla (como no “Caso 3”).

- Rotação (à direita) de R em torno de Q;
- Rotação (à esquerda) de R em torno de P.



Por Fim, vamos analisar as operações de Inserção de Remoção quanto a quantidade de nós visitados e rotações realizadas.

Quantidade de Nós visitados:

- Inserção e Remoção: busca pelo nó a ser removido (ou pelo local de inserção): ***O***(log2n) nós visitados

- Inserção: percurso ascendente até encontrar o nó P: ***O***(log2n) nós visitados (se não houver desbalanceamento, precisa ir até a raiz);

- Remoção: sempre sobe até a raiz: ***Θ***(log2n).

Quantidade de Rotações realizadas:

- Inserção: zero rotações, uma rotação simples, ou uma rotação dupla;

- Remoção: 1, 44 · log2(n + 2) rotações no pior caso. No caso médio: log2(n) + 0, 25 (ou seja, ***O***(log2n)).

Resultados empíricos indicam que:

- 78% das remoções não resultam em desbalanceamento;
- 53% das inserções não resultam em desbalanceamento.

Consequentemente, remoções mais demoradas são pouco frequentes, não comprometendo a eficiência de rebalanceamento de árvores AVL.

III – A REPRESENTAÇÃO COMPUTACIONAL

O código implementado, desenvolvido na linguagem Python, é formado por 3 arquivos principais:

1. Tree_Node.py
2. ABB_tree.py
3. AVL.py

O primeiro arquivo, realiza a implementação do nó da árvore, por meio da classe “Tree_Node”, que possui as seguintes funções/métodos:

1. def __init__(self, key):

   - Inicializa um nó com a key (dados) passada pelo usuário;
   - Possui as Informações utilizadas para “linkar” e “plotar” a árvore;
   - Possui o Fator de balanceamento do nó para a árvore AVL.

2. def __repr__(self) -> str:

   - Transforma o nó em uma string para ser imprimida ou exibida de alguma forma.

3. def is_leaf(self):

   - Função/método que retorna true se nó for folha, ou seja, se ambos “self.l_child” e “self.r_child” forem Nulos (None).

4. def is_left(self):

   - Função/método que retorna se o nó é filho da esquerda do pai dele.

5. def one_child(self):

   - Função que Retorna o filho único se o nó só tiver um filho;
   - Caso o nó seja folha, ou tenha dois filhos, retornará None.

6. def rec_search(self, node_key):

   - Função que faz a pesquisa pelo próprio nó recursivamente, pela sua chave (key).

O segundo arquivo, “ABB tree.py”, implementa uma Árvore Binária de Busca ainda sem as características do algoritmo AVL, porém, com todas as suas funções usuais e as operações complementares realizadas pela equipe para que haja uma melhor vizualização da árvore e seu funcionamento.

A Classe implementada “ABB_Tree”, utiliza “Tree_Node” e possui mais 7 funções, sendo elas:

1. def __init__(self):

   - Inicializa uma árvore com a raiz vazia.

2. def is_empty(self):

   - Função/Método que diz se a árvore está vazia ou não.

3. def incert_node(self, node_key):

   - Método utilizado para inserir um nó em uma árvore binária de busca;
   - Mantém a ordenação, porém não faz balanceamento, ou seja, pode desbalancear a árvore.

4. def delete_node(self, node_key):

   - Método utilizado para remover um nó, recebendo como parâmetro a chave do nó a ser removido;
   - Mantém a ordenação porém não faz o rebalanceamento.

5. def search(self, node_key):

- Método de busca binária de forma iterativa.

6. def rec_search(self, node_key):

    - Realiza a busca binária de forma recursiva;
    - Chama a busca recursiva no nó raiz da árvore.

7. def calculate_depth(self, node="Root"):

    - calcula a profundidade de uma árvore de forma recursiva.

Finalmente, o arquivo “AVL.py”, possui a implementação da árvore AVL em si. A classe “AVL_Tree” herda a classe “ABB_Tree”, já explicada, e adiciona as características específicas da AVL.

A classe implementa mais 9 métodos/funções, essenciais para o algoritmo de Adelson-Velsky e Landis, são elas:

1. def __init__(self):

    - Inicializa a árvore normalmente, com a raiz nula, Chamando o método da classe pai (ABB_Tree).

2. def get_balancing_factor(self, node):

    - Método que calcula o fator de balanceamento;
    - Primeiro, Calcula-se a profundidade da árvore à esquerda e à direita;
    - Em seguida, Subtrai-se a direita menos a da esquerda.

3. def update_balancing_factor(self):

    - Método que percorre toda a árvore depth-first, atualizando os fatores de balanceamento de cada nó;
    - Esse método é utilizado após o rebalanceamento para atualizar os fatores de balanceamento.

4. def left_rotate(self, node):

    - Método que implementa a rotação à esquerda em torno do nó passado como parâmetro.

5. def right_rotate(self, node):

    - Implementa a rotação à direita em torno do nó passado como parâmetro;
    - Simétrico ao método left_rotate.

6. def rebalance(self, node, bf):

    - Faz o rebalanceamento da árvore, recebendo um nó e seu fator de rebalanceamento como parâmetros;
    - implementa os algoritmos descritos como rebalanceamento após a inserção.

7. def busca_desbalanceamento(self, node):

    - Método que percorre o caminho ascendente partindo de um nó, e efetua os rebalanceamentos necessário.

8. def incert_node(self, node_key):

- Método que insere um novo nó, como em uma ABB, depois, caso haja necessidade, é realizado o balanceamento.

9. def delete_node(self, node_key):

   - Método que deleta um nó, como em uma ABB, depois, caso haja necessidade, é realizado o balanceamento.

IV – A IMPLEMENTAÇÃO

Como já explicado na seção anterior, o código, possui 3 arquivos principais, nos quais as operações mais importantes foram implementadas, estes podem ser acessados pelo endereço https://github.com/rodrigues-pedro/avl-tree.

Nessa seção serão apresentadas e exemplificadas as principais funções do programa: Criação da árvore (vazia), inserção de um nó, remoção de um nó, consulta de um nó (a partir do valor de sua chave primária) e destruição da árvore.

1. Criação da árvore (vazia)

```
class Tree_Node:

    def __init__(self, key):
        # Inicializa um nó com a key passada pelo usuário
        self.key = key

        # Informações utilizadas para linkar a árvore
        self.parent = None
        self.l_child = None
        self.r_child = None

        # Informações utilizadas para plotar a árvore
        self.x = None
        self.y = None

        # Apenas para AVL
        # Fator de Balanceamento para a AVL
        # No caso da ABB, essa informação não é utilizada
        self.bf = None
```

A árvore é formada por nós e cada nó é inicializado por uma key (chave) passada pelo usuário, o nó possui os dados mostrados no código acima.

```python
class ABB_Tree:
    """
    Árvore de Busca Binária
    """
    def __init__(self):
        # Inicializa uma árvore com a raiz vazia
        self.root = None

```

A Árvore Binária de Busca, utiliza os métodos do nó e possui o método que inicializa uma árvore com raiz vazia.

```python
class AVL_Tree(ABB_Tree):
    """
    Classe que vai herdar a Árvore de Busca Binária e adicionar as caracteristicas especificas da Árvore AVL
    """

    def __init__(self):
        """
        Inicializa a árvore normalmente, com a raiz nula
        Chamando o método da classe pai
        """
        super(AVL_Tree, self).__init__()

```

Por fim, a classe “AVL_Tree”, herda a ABB e inicializa a árvore com características AVL chamando o método da classe pai (ABB_Tree).

2. Inserção de um nó:

```python
    def incert_node(self, node_key):
        """
        Método utilizado para inserir um nó em uma árvore binária de busca
        Mantém a ordenação, porém não faz balanceamento, ou seja, pode desbalancear a árvore
        """
        # inicializa um novo nó com a chave recebida por parâmetro
        new_node = Tree_Node(node_key)

        if self.is_empty():
            # se a árvore for vazia:
            # coloca o novo nó como raiz da árvore
            self.root = new_node
        else:
            # se a árvore não for vazia:
            tmp = self.root
            # iremos percorrer a árvore, partindo da raiz
            while True:
                # caso a chave já exista, levantaremos uma exceção
                if node_key == tmp.key:
                    raise Exception("Essa chave já existe. Tente com outra chave!")
                # caso a chave seja menor que a chave do nó atual:
                elif node_key < tmp.key:
                    # se não houver filho para a esquerda
                    if tmp.l_child is None:
                        # adiciona-se o novo nó à esquerda do nó atual
                        new_node.parent = tmp
                        tmp.l_child = new_node
                        break
                    # se houver, passaremos a olhar para o filho da esquerda
                    tmp = tmp.l_child
                # caso a chave seja maior que a chave do nó atual:
                else:
                    # se não houver filho para a direita
                    if tmp.r_child is None:
                        # adiciona-se o novo nó à direita do nó atual
                        new_node.parent = tmp
                        tmp.r_child = new_node
                        break
                    # se houver, passaremos a olhar para o filho da esquerda
                    tmp = tmp.r_child
```

Este método é utilizado para inserir um nó em uma ABB. Mantém a ordenação, porém não realiza o balanceamento, ou seja, pode desbalancear a árvore.

Inicializa um novo nó com a chave recebida por parâmetro, se a árvore for vazia, coloca-se o novo nó como raiz da árvore. Se a árvore não for vazia, a árvore é percorrida partindo da raiz, caso a chave seja menor que a do nó atual e não houver filho para a esquerda então adiciona-se o novo nó à esquerda do nó atual.

Agora, caso a chave seja menor que a do nó atual e houver filho para a esquerda, então passaremos a olhar para este.

Outrora, caso a chave seja maior que a chave do nó atual:

- Se não houver filho para a direita, adiciona-se o novo nó à direita do nó atual.
- Se não houver, passaremos a olhar para o filho da esquerda.

```python
    def incert_node(self, node_key):
        """
        Método que insere um novo nó
        Primeiro inserimos o nó como em uma ABB
        Depois fazemos o rebalanceamento caso haja tal necessidade
        """
        # Inserimos o nó como em uma ABB
        super(AVL_Tree, self).incert_node(node_key)

        # Busca o desbalanceamento a partir do nó adicionado
        node = self.search(node_key)
        self.busca_desbalanceamento(node)
```

Já no algoritmo AVL, o método insere o nó como em uma Árvore binária de Busca e busca o desbalanceamento a partir do nó adicionado.

2.1. Busca Desbalanceamento

```python
    def busca_desbalanceamento(self, node):
        """
        Método que percorre o caminho ascendente partindo de um nó
        E efetua os rebalanceamentos nescessários
        """
        # Percorremos o caminho ascendente, partindo do nó recebido como parâmetro
        # buscando uma necessidade de rebalanceamento
        while node is not None:
            bf = self.get_balancing_factor(node)

            if bf > 1 or bf < -1:
                # Caso encontra-se um nó com |fator de balanceamento| > +-1
                # é feito o rebalanceamento em torno desse nó
                self.rebalance(node, bf)

            node = node.parent

        # Atualizamos todos o fatores de balanceamento
        self.update_balancing_factor()
```

Esse método percorre o caminho ascendente partindo de um nó e efetua os balanceamentos necessários.

Para isso, o método também utiliza de 2 outros métodos distintos: “rebalance” e “get_balancing_factor”.

2.2. Obter fator de balanceamento (get_balancing_factor)

```python
    def get_balancing_factor(self, node):
        """
        Método que calcula o fator de balanceamento
        Calcula-se a profundidade da árvore à esquerda e à direita
        Subtrai-se a direita menos a da esquerda
        """
        h_r = self.calculate_depth(node.r_child)
        h_l = self.calculate_depth(node.l_child)
```

O método, calcula o fator de balanceamento calculando também a profundidade da árvore à esquerda e à direita, e subtraindo as profundidades posteriormente.

2.2.1. Calcular Profundidade (calculate_depth)

```python
    def calculate_depth(self, node="Root"):
        """
        Método que calcula a profundidade de uma árvore recursivamente.

        :Params:
        node:
            str:       Root (default) parte da raiz
            Tree_Node: Partiremos a partir do nó que foi passado como parâmetro
            None:      Consideraremos uma árvore vazia, profundidade zero
        """
        # recebendo a string "Root" como parâmetro, buscaremos a raiz como ponto de partida
        if node == "Root":
            node = self.root

        if node is None:
            # se o nó for nulo, retorna-se zero
            return 0
        elif node.is_leaf():
            # se o nó for folha, retorna-se 1
            # Critério de parada da recursão
            return 1
        else:
            # caso o nó não seja folha, nem nulo
            # Calcula a profundidade das subárvores à esquerda e à direita
            l_depth = self.calculate_depth(node.l_child)
            r_depth = self.calculate_depth(node.r_child)

            # retorna a profundidade máxima + 1
            return max(r_depth, l_depth) + 1
```

Calcula-se a profundidade de uma árvore recursivamente (Código Comentado).

2.3. Rebalanceamento

```python
    def rebalance(self, node, bf):
        """
        Faz o rebalanceamento da árvore
        recebendo um nó e seu fator de rebalanceamento como parâmetros

        Ele implementa os algoritmos descritos como rebalanceamento após a incerção
        Pois, tendo uma árvore desbalanceada, sabendo que iremos atualizar todos os fatores de balanceamento,
        Esse procedimento é o suficiente para rebalancear qualquer árvore,
        desde que o caminho ascendente inteiro seja percorrido
        """
        if bf > 0:
            # Sub-árvore da direita maior
            bf_child = self.get_balancing_factor(node.r_child)
            if bf_child > 0:
                # Caso 1: Subárvore direita do filho à direita
                self.left_rotate(node)
            else:
                # Caso 2: Subárvore esquerda do filho à direita
                self.right_rotate(node.r_child)
                self.left_rotate(node)
        else:
            # Sub-árvore da esquerda maior
            bf_child = self.get_balancing_factor(node.l_child)
            if bf_child > 0:
                # Caso 3: Subárvore direita do filho à esquerda
                # Simétrico Caso 2
                self.left_rotate(node.l_child)
                self.right_rotate(node)
            else:
                # Caso 4: Subárvore esquerda do filho à esquerda
                # simétrico Caso 1
                self.right_rotate(node)
```

Realiza o balanceamento da árvore recebendo um nó e seu fator de rebalanceamento (obtido por “get_balancing_factor “) como parâmetros.

Ele implementa os algoritmos descritos como rebalanceamento após a inserção pois, tendo uma árvore desbalanceada, sabendo que iremos atualizar todos os fatores de balanceamento, esse procedimento é o suficiente para rebalancear qualquer árvore, desde que o caminho ascendente inteiro seja percorrido.

Este método/função, também utiliza 2 outros métodos essenciais para o funcionamento do algoritmo AVL: “right_rotate” e “left_rotate”.

2.3.1. Rotação à Direita (right_rotate)

```python
    def right_rotate(self, node):
        """
        Método que implementa a rotação à direita em torno do nó passado como parâmetro
        Esse parâmetro pode receber tanto um int, nesse caso buscaremos o nó usando a pesquisa iterativa
        ou o objeto Tree_Node em si, dessa forma pulariamos esse primeiro passo

        Simétrico ao método left_rotate
        """
        if isinstance(node, int):
            node = self.search(node)

        node_l = node.l_child

        node.l_child = node_l.r_child
        if node_l.r_child is not None:
            node_l.r_child.parent = node

        node_l.parent = node.parent
        if node.parent is None:
            self.root = node_l
        elif node == node.parent.l_child:
            node.parent.l_child = node_l
        else:
            node.parent.r_child = node_l

        node_l.r_child = node
        node.parent = node_l
```

O método, implementa a rotação à direita em torno do nó passado como parâmetro. Esse parâmetro pode receber tanto um int, nesse caso buscaremos o nó usando a pesquisa iterativa ou o objeto Tree_Node em si, dessa forma pularíamos esse primeiro passo.

2.3.2. Rotação à Esquerda (left_rotate)

```python
def left_rotate(self, node):
    """
    Método que implementa a rotação à esquerda em torno do nó passado como parâmetro
    Esse parâmetro pode receber tanto um int, nesse caso buscaremos o nó usando a pesquisa iterativa
    ou o objeto Tree_Node em si, dessa forma pulariamos esse primeiro passo
    """
    if isinstance(node, int):
        node = self.search(node)

    node_r = node.r_child

    node.r_child = node_r.l_child
    if node_r.l_child is not None:
        node_r.l_child.parent = node

    node_r.parent = node.parent
    if node.parent is None:
        self.root = node_r
    elif node == node.parent.l_child:
        node.parent.l_child = node_r
    else:
        node.parent.r_child = node_r

    node_r.l_child = node
    node.parent = node_r
```

Simétrico ao método “right_rotate”, este também, implementa a rotação à esquerda em torno do nó passado como parâmetro. Esse parâmetro pode receber tanto um int, nesse caso buscaremos o nó usando a pesquisa iterativa ou o objeto Tree_Node em si, dessa forma pularíamos esse primeiro passo.

2.4. Atualizar Fator de Balanceamento (update_balancing_factor)

```python
def update_balancing_factor(self):
    """
    Método que percorre toda a árvore depth-first
    atualizando os fatores de balanceamento de cada nó

    Esse método é utilizado após o rebalanceamento para atualizar os fatores de balanceamento
    """
    fila = []
    tmp = self.root

    # Loop para percorrer a árvore depth first
    while tmp is not None:
        # atualiza o fator de balanceamento de cada nó, ao percorrer a árvore
        tmp.bf = self.get_balancing_factor(tmp)

        if tmp.l_child is not None:
            fila.append(tmp.l_child)
        if tmp.r_child is not None:
            fila.append(tmp.r_child)

        if len(fila) == 0: break
        tmp = fila.pop(0)
```

Por fim, o método percorre toda a árvore depth-first atualizando os fatores de balanceamento de cada nó, apenas utilizado após o rebalanceamento para atualizar os fatores de balanceamento.

3. Remoção de um nó

```
    def delete_node(self, node_key):
        """
        Método utilizado para remover um nó, recebendo como parâmetro a chave do nó a ser removido
        Mantém a ordenação porém não faz o rebalanceamento.
        """
        # busca o nó a ser removido, utilizando o método de pesquisa
        try:
            to_delete = self.search(node_key)
        except Exception as E:
            raise E

        # busca o nó pai do nó a ser deletado
        parent = to_delete.parent
        # guardamos se o nó a ser deletado é filho da esquerda ou não
        lc = to_delete.is_left()

        # Caso 1: Nó folha
        # { p -> d } passa a ser { p -> None }
        if to_delete.is_leaf():
            # Nó pai do nó a ser deletado para de apontar para esse nó
            if lc:
                parent.l_child = None
            else:
                parent.r_child = None

        else:
            # busca o filho único caso seja único
            oc = to_delete.one_child()

            # Caso 2: Nó com uma subárvore
            # { p -> d -> oc } passa a ser { p -> oc }
            if oc is not None:
                # Nó pai do nó a ser deletado passa a apontar para o filho único do nó a ser deletado
                oc.parent = parent
                if lc:
                    parent.l_child = oc
                else:
                    parent.r_child = oc

            # Caso 3: Nó com duas subárvores
            # { p -> d -> [f_e, f_d] } passa a ser { p -> f_d -> f_e }
            else:
                # Aponta o pai do nó a ser deletado para o filho da direita do nó a ser deletado
                tmp = to_delete.r_child
                tmp.parent = parent
                if lc:
                    parent.l_child = tmp
```

```python
            else:
                parent.r_child = tmp

            # busca um nó vazio à esquerda do filho da direita do nó a ser deletado
            while tmp.l_child is not None:
                tmp = tmp.l_child

            # Aponta o filho da esquerda do nó à ser deletado como
            # filho da esquerda do nó encontrado no loop acima
            tmp.l_child = to_delete.l_child
            to_delete.l_child.parent = tmp
```

O método é utilizado para remover um nó, recebendo como parâmetro a chave do nó a ser removido, mantém a ordenação porém não faz o rebalanceamento.

```python
    def delete_node(self, node_key):
        """
        Método que deleta um nó
        Primeiro deletamos o nó como em uma ABB
        Depois fazemos o rebalanceamento caso haja tal necessidade
        """
        # Buscamos o nó pai do nó que vai ser deletado
        node = self.search(node_key)
        node = node.parent

        # deletamos o nó como em uma ABB
        super(AVL_Tree, self).delete_node(node_key)

        # Busca o desbalanceamento a partir do nó pai do nó que foi deletado
        self.busca_desbalanceamento(node)
```

Já no algoritmo AVL, o método “delete_node”, primeiro deleta um nó como em uma ABB e depois realiza o balanceamento caso necessário, utilizando o método “busca_desbalanceamento” já explicado.

4. Busca de um Nó

```
    def search(self, node_key):
        """
        Método para a busca iterativa
        """
        # partimos da raiz da árvore
        tmp = self.root
        while tmp is not None:
            # Se a chave buscada for igual a chave do nó
            # Retorna-se o própio nó
            if tmp.key == node_key:
                return tmp

            # Se a chave do nó for maior que chave buscada
            # Iremos olhar para o nó da esquerda
            elif tmp.key > node_key:
                tmp = tmp.l_child

            # Se a chave do nó for menor que chave buscada
            # Iremos olhar para o nó da direita
            else:
                tmp = tmp.r_child

        # Se chegarmos em uma folha sem encontrar a chave
        # Iremos erguer uma excessão
        raise Exception("Chave não encontrada!")

    def rec_search(self, node_key):
        """
        Busca recursiva
        Chama a busca recursiva no nó raiz da árvore
        """
        return self.root.rec_search(node_key)
```

A operação de busca de um nó, foi implementada tanto iterativamente quanto recursivamente.

O método “search”, parte da raiz da árvore, se a chave buscada for igual a chave do nó, retorna-se o próprio nó. Se a chave do nó for maior que a chave buscada olhamos para o nó da esquerda. Se a chave do nó for menor que a chave buscada iremos olhar para o nó da direita e se chegarmos em uma folha sem encontrar a chave informada, erguemos uma exceção.

Já o método “rec_search” que realiza a busca recursiva, chama a busca recursiva no nó da raíz.

5. Destruição da Árvore

```python
# Operação escolhida é feita no nó de acordo com o que o usuário solicitou
with cp:
    try:
        if operacao == "Inserir Nó":
            st.session_state.tree.incert_node(key)

        elif operacao == "Remover Nó":
            st.session_state.tree.delete_node(key)

        elif operacao ==  "Buscar um Nó":
            node = st.session_state.tree.search(key)
            st.write(node)

        elif operacao ==  "Deletar a Árvore":
            del st.session_state.tree
            st.session_state.tree = AVL_Tree()

        st.plotly_chart(st.session_state.tree.plot_tree())
    except Exception as E:
        st.write(E)
```

No arquivo app.py o usuário pode escolher a operação a ser realizada, uma delas seria Deletar a Árvore.

```python
        elif operacao ==  "Deletar a Árvore":
            del st.session_state.tree
            st.session_state.tree = AVL_Tree()
```

V – PROCESSO DE INSTALAÇÃO DO PROGRAMA DE COMPUTADOR

O código completo pode ser acessado pelo endereço https://github.com/rodrigues-pedro/avl-tree

Essa aplicação foi feita utilizando Python = 3.9.12 e, dessa forma, recomendo que você utilize a mesma versão ao testa-lo. Caso não saiba instalar o Python, esse link pode ser útil.

Tendo o python instalado, é possivel criar um ambiente virtual para a execução desse aplicativo, utilizando o comando:

```
python -m venv venv
```

Isso sendo feito, é necessário ativar o ambiente virtual e instalar as outras dependências:

```
.\venv\Scripts\activate  # Windows

source venv\bin\activate # Linux

pip install -r /requirements.txt
```

Após a execução desses comandos, podemos rodar a aplicação com o comando:

```
streamlit run app.py
```

Alternativamente, essa aplicação pode ser acessada pelo link https://aed2-avl-tree.streamlit.app/

VII – BIBLIOGRAFIA

AVL tree. **Wikipedia,** 2021. Disponível em: https://en.wikipedia.org/wiki/AVL_tree. Acesso em: 24 nov. 2022.

Georgy Adelson-Velsky. **Chess Programming Wiki,** 2018. Disponível em: https://www.chessprogramming.org/Georgy_Adelson-Velsky. Acesso em: 24 nov. 2022.

AVL Tree And Heap Data Structure In C++. **Software testing help,** 2022. Disponível em: https://www.softwaretestinghelp.com/avl-trees-and-heap-data-structure-in-cpp/. Acesso em: 25 nov. 2022.

G. M. Adel'son-Vel'skii, E. M. Landis, “An algorithm for organization of information”, *Dokl. Akad. Nauk SSSR* , **146** :2 (1962).

CORMEN, Thomas. **Introduction to Algorithms**. 3. ed. Mit: The Mit Press, 2009. (Capítulos 12 e 13).