

INE5609 03238A - Estruturas de Dados

Daniel Castagna Santos (20100838)

Paula Zomignani Oliveira (20103838)

## TRABALHO 3

### IMPLEMENTAÇÃO DE UMA ÁRVORE AVL

#### OBJETIVO

Implementar uma árvore AVL, descrever os métodos de inclusão e exclusão, e indicar as partes do código fazem ações no método de balanceamento (rotações simples e duplas).

#### ESTRUTURA DO ALGORITMO AVL IMPLEMENTADO

- Classe **Node**: cria os elementos que serão inseridos na árvore AVL.
- Classe **AVLTree**: estrutura da árvore que linka os elementos. Possui os métodos:
  - ***insert***: método que faz a inserção de um novo nó na árvore.
  - ***\_inspect\_insertion***: avalia a necessidade do (re)balanceamento da árvore AVL após cada inserção.
  - ***\_rebalance\_node***: orquestra o (re)balanceamento da árvore (caso necessário) após uma inserção.
  - ***\_left\_rotate*** e ***\_right\_rotate***: funções que manipulam propriamente os nós da árvore para fazer o (re)balanceamento.
  - ***delete\_node***: deleta o nó da árvore dado um valor.
  - ***\_inspect\_deletion***: avalia a necessidade do (re)balanceamento da árvore AVL após cada deleção.
  - ***find***: encontra um nó na árvore dado um valor.

## MÉTODOS DE INSERÇÃO DE ELEMENTO

A seguir são descritas as etapas dos métodos de inclusão de um novo elemento no código implementado.

1. É chamada uma função *insert*, que recebe o valor da chave do elemento a ser inserido;
  - a. Essa função faz uma primeira verificação: caso a árvore não possua uma raiz definida (logo, não possui nenhum elemento), o elemento é criado e setado como raiz da árvore;
  - b. Caso a árvore já possua um elemento setado como sua raiz, é chamado o método interno *\_insert*;

```
def insert(self, value:int):  
    if self.root == None:  
        self.root = Node(value)  
    else:  
        self._insert(value, self.root)
```

2. O método *\_insert* funcionará de forma recursiva. Ele recebe o valor do elemento que será inserido e o Nó atual. O percorrimto da árvore em busca do local onde o elemento será inserido começa pela raiz;
3. A função *\_insert* irá trabalhar com 3 casos:
  - a. Se o valor da chave do elemento a ser inserido é maior que o valor da chave do nó atual;
  - b. Se o valor da chave do elemento a ser inserido é menor do que o nó atual;
  - c. Caso nenhuma das duas opções se concretizar, então o valor da chave do elemento a ser inserido é igual ao do nó atual (nesse caso o elemento não será inserido);
4. Após ser definido se a chave do elemento é maior ou menor do que a do nó atual, checka-se se o nó atual possui filhos:

5. Caso o nó atual não possua um filho à esquerda (*left\_child*) e o elemento for menor do que o nó atual: o elemento é adicionado como filho à esquerda do nó atual, o nó atual é setado como pai (*parent*) do elemento inserido e encerra-se o processo de inserção;
  - a. O mesmo acontece caso o nó atual não possua um filho à direita (*right\_child*) e o elemento a ser inserido for maior do que o nó atual;
6. Caso o cenário do elemento 5 não se concretize, a função `_insert` é chamada novamente, dessa vez passando o respectivo filho do nó atual como *cur\_node* (current node, que será o nó com o qual se farão as comparações do elemento a ser inserido);
7. O processo é repetido até que se caia na situação do item 5;
8. Após o elemento ser inserido na árvore é chamado o método `_inspect_insertion`, que irá avaliar o balanceamento da árvore e acionar as rotações caso seja necessário.

```
def _insert(self, value:int, cur_node:Node):  
    if value < cur_node.value:  
        if cur_node.left_child == None:  
            cur_node.left_child = Node(value)  
            cur_node.left_child.parent = cur_node # set parent  
            self._inspect_insertion(cur_node.left_child)  
        else:  
            self._insert(value, cur_node.left_child)  
    elif value > cur_node.value:  
        if cur_node.right_child == None:  
            cur_node.right_child = Node(value)  
            cur_node.right_child.parent = cur_node # set parent  
            self._inspect_insertion(cur_node.right_child)  
        else:  
            self._insert(value, cur_node.right_child)  
    else:  
        print("Value already in tree!")
```

## MÉTODOS DE BALANCEAMENTO

### Inspeção de Elemento Inserido (*\_inspect\_insertion*):

1. O fluxo de balanceamento da árvore AVL inicia com o método *\_inspect\_insertion*, que irá avaliar o balanceamento da árvore após a inserção de uma nova folha seguindo o sentido folha -> raiz;
  - a. O Método recebe o Nó atual (que foi recém inserido na árvore) e o “path” (caminho), uma lista vazia que será incrementada com o caminho que foi percorrido para inserir o nó;
2. Avalia-se a altura dos dois filhos do pai (*parent*) do nó atual (ou seja: a altura do nó atual e de seu irmão. Caso não exista irmão, considera-se altura 0). Essas alturas são obtidas pela função *\_get\_height*;
3. Avalia-se se a diferença da altura dos ramos da esquerda e da direita está no intervalo  $[-1, 1]$ ;
4. Caso seja verificado que a árvore não está balanceada, serão trabalhados 3 nós no balanceamento:
  - a. *node\_z*: nó não balanceado (é o pai do nó atual)
  - b. *node\_y* e *node\_x*: nó atual e nó encontrado na estrutura path
5. O elemento não balanceado é fixado no início da lista (*path*), e os 3 elementos são passados para a função *\_rebalance\_node*. Essa função irá de fato gerenciar o balanceamento da árvore;
6. Caso seja verificado que a árvore está balanceada, calcula-se uma nova altura para o pai:  $1 + \text{altura do nó atual}$ ;
7. Chama-se a função *\_inspect\_insertion* novamente atualizando seus valores. Isso será feito de forma recursiva até que se caia no cenário 5 ou que chegue em um elemento que não possui pai (ou seja, a raiz). Nos dois casos a função é encerrada.

```
def _inspect_insertion(self, cur_node:Node, node_path = []):

    if cur_node.parent == None:
        return

    # lista que acumula o caminho dos nodos da folha até a raiz
    # [raiz, ... z, x, y, ...folha]
    node_path = [cur_node] + node_path

    left_height = self._get_height(cur_node.parent.left_child)
    right_height = self._get_height(cur_node.parent.right_child)

    # se o absoluto da diferenca entre a arvore a esquerda e a direita do no atual for maior que 1
    # é preciso rebalancear
    if abs(left_height - right_height) > 1:
        node_path = [cur_node.parent] + node_path
        self._rebalance_node(
            node_z=node_path[0],
            node_y=node_path[1],
            node_x=node_path[2],
        )
        return





    # se não for preciso rebalancear, então vamos atualizar a altura do no pai
    new_parent_height = 1 + cur_node.height

    if new_parent_height > cur_node.parent.height:
        cur_node.parent.height = new_parent_height

    # vamos continuar esse processo recursivamente até a raiz
    self._inspect_insertion(cur_node.parent, node_path)
```

### Rebalanceamento da Árvore (*\_rebalance\_node*):

1. A função *\_rebalance\_node*, chamada dentro do método *\_inspect\_insertion*, será dividida nas seguintes possibilidades:

| <u>LEFT LEFT</u>  | <u>LEFT RIGHT</u>   | <u>RIGHT RIGHT</u>  | <u>RIGHT LEFT</u>   |
|---|---|---|---|
|  |  |  |  |
| Right Rotation  | Left Rotation,<br>Right Rotation  | Left Rotation   | Right Rotation,<br>Left Rotation  |

2. Cada um dos casos exibidos serão corrigidos com uma rotação única ou dupla, de acordo com a tabela;
3. São chamadas as funções `_left_rotate` e `_right_rotate` de acordo com a necessidade;
4. Caso nenhum dos casos exibidos acima seja identificado é gerada uma exceção.

```
def _rebalance_node(self, node_z:Node, node_y:Node, node_x:Node):  
  
    # LEFT LEFT CASE: se o filho esquerdo de Z for Y, e o filho esquerdo de Y for X,  
    # então faz uma rotacao direita em Z  
    if node_z.left_child == node_y and node_y.left_child == node_x:  
        self._right_rotate(node_z)  
  
    # LEFT RIGHT CASE: se o filho esquerdo de Z for Y, e o filho direito de Y for X,  
    # então faz uma rotacao esquerda em Y e rotação direita em Z  
    elif node_z.left_child == node_y and node_y.right_child == node_x:  
        self._left_rotate(node_y)  
        self._right_rotate(node_z)  
  
    # RIGHT RIGHT CASE: se o filho direito de Z for Y, e o filho direito de Y for X,  
    # então faz uma rotacao esquerda em Z  
    elif node_z.right_child == node_y and node_y.right_child == node_x:  
        self._left_rotate(node_z)  
  
    # RIGHT LEFT CASE: se o filho direito de Z for Y, e o filho esquerdo de Y for X,  
    # então faz uma rotacao direita em Y e rotação esquerda em Z  
    elif node_z.right_child == node_y and node_y.left_child == node_x:  
        self._right_rotate(node_y)  
        self._left_rotate(node_z)  
  
    else:  
        raise Exception('_rebalance_node: z, y, x node configuration not recognized!')
```

### Rotação de nós da Árvore (`_left_rotate` e `_right_rotate`):

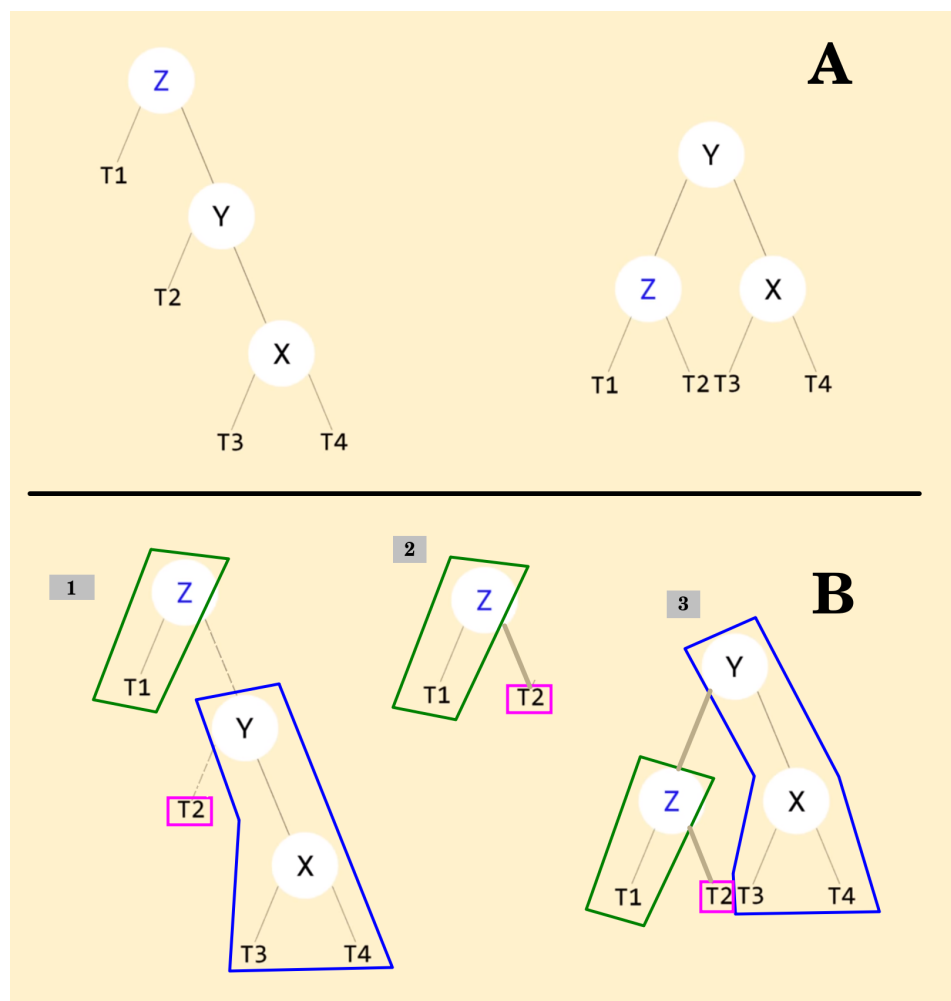
A seguir descrevemos o algoritmo de rotação para a esquerda `_left_rotate`. O método recebe um nó na posição referência Z, e faz a rotação conforme apresentado na imagem 'Rotação para a esquerda'. A rotação para a direita `_right_rotate`, segue a mesma lógica porém no sentido contrário (ou seja, a rotação para direita é um espelhamento da rotação para a esquerda).

Os passos da rotação para esquerda são os seguintes:

1. Guardar o pai de Z.
2. Atribui ao nó Y o filho direito de Z.
3. Atribui ao nó T2 o filho esquerdo do novo nó Y (que é o filho esquerdo do filho direito de Z).
4. Fazer o filho esquerdo de Y receber o nó Z.

5. O novo pai de Z vira Y.
6. O filho direito de Z recebe T2.
7. Caso o T2 não seja nulo, seu pai é Z.
8. O novo pai de Y recebe o antigo pai de Z.
9. Se o pai de Y for Nulo então ele é a raiz da árvore.
10. Se não, se por acaso o filho esquerdo do pai de Y for Z, ajustar para Y.
11. Se não, o filho direito do pai de Y deve ser Y

### Rotação para a esquerda



A imagem X ilustra na parte A como deve estar a estrutura da árvore antes da rotação à esquerda e como ela fica depois da rotação à esquerda. Já a parte B da imagem destaca os passos e as mudanças necessárias para alcançar a nova configuração.

A implementação fica assim:

```
def _left_rotate(self, node_z:Node):
    # 1) Guarda o pai de Z
    z_parent = node_z.parent

    # 2) Atribui ao no Y o filho direito de Z
    node_y = node_z.right_child

    # 3) Atribui ao no T2 o filho esquerdo de Y
    node_t2 = node_y.left_child

    # 4) Faz filho esquerdo de Y receber o no Z
    node_y.left_child = node_z

    # 5) O novo pai de Z vira Y
    node_z.parent = node_y

    # 6) O filho direito de Z recebe a arvore 2
    node_z.right_child = node_t2

    # 7) Caso o a arvore 2 não seja nula, seu pai é Z
    if node_t2 != None:
        node_t2.parent = node_z

    # 8) O novo pai de Y recebe o antigo pai de Z
    node_y.parent = z_parent

    # 9) Se o pai de Y for Nulo então ele é a raiz da arvore
    if node_y.parent == None:
        self.root = node_y

    # 10) Se não, se por acaso o filho esquerdo do pai de Y for Z, ajustar para Y
    elif node_y.parent.left_child == node_z:
        node_y.parent.left_child = node_y

    # 11) Se não, o filho direito do pai de Y deve ser Y
    else:
        node_y.parent.right_child = node_y
```



## MÉTODOS DE DELEÇÃO DE ELEMENTO:

1. O método `_delete_node` se divide em 3 casos: deleção de uma folha, de um nó com um filho ou de um nó com dois filhos;
2. O fluxo de deleção inicia com a função `delete_node`, que busca o elemento desejado e, caso encontrado, chama a função `_delete_node` o passando como parâmetro

```
# funcao publica que encontra o No desejado (pelo metodo find)
# e chama a funcao privada _delete_node passando o valor encontrado
def delete_node(self,value):
    return self._delete_node(self.find(value))
```

3. A função `_delete_node` checa se o nó a ser deletado existe (caso não exista, a função é encerrada);
4. Existe uma função dentro do método `_delete_node` utilizada para retornar o número de

```
# retorna o numero de filhos de um No especifico (0, 1 ou 2)
def num_children(node:Node) -> int:
    num_children = 0
    if node.left_child != None:
        num_children += 1
    if node.right_child != None:
        num_children += 1
    return num_children
```

filhos de um determinado nó, possibilitando a execução dos métodos de deleção;

5. Caso o elemento a ser deletado não possua filhos (ou seja, caso seja uma folha) a deleção é muito simples: é checado se a folha é um filho à esquerda ou à direita de seu pai e seta-se a referência desse filho (nó deletado) para `None`. Caso a folha não possua pai (ou seja, caso seja a raiz) a referência da raiz (*root*) da árvore passa a ser `None`;
6. No segundo caso (nó com um filho), primeiro verifica-se se o filho está à esquerda (*left\_child*) ou à direita (*right\_child*) do nó e coloca-se o filho em uma variável genérica (*child*). Verifica-se se o nó possui raiz ou não (ou seja, se possui pai);
  - a. Caso possua, o pai do nó a ser deletado passará a apontar para o filho do nó a ser deletado como seu filho (ou seja, seu neto atual passará a ser seu filho);
  - b. Caso contrário, seta-se o filho do nó a ser deletado como raiz da árvore;
  - c. Nos dois casos, altera-se a referência de *parent* (pai) da *child* (filho do nó a ser deletado) para o pai de seu pai, completando a deleção.

7. No terceiro caso (Nó possui dois filhos), primeiramente é verificado o valor do sucessor do nó que será removido - esse valor é copiado para o nó que desejamos remover. Agora que o valor do sucessor já foi salvo em outro local, deleta-se o sucessor (através da função *delete*). Isso é feito de forma recursiva, até chegar a um nó com 0 ou 1 filho - então são utilizados os métodos descritos nos tópicos 5 ou 6 e o nó desejado é deletado.

```
# CASO 1: No sem filhos (folha)
if node_children == 0:

    if node_parent != None:
        # remove a referencia do No deletado No pai
        if node_parent.left_child == node:
            node_parent.left_child = None
        else:
            node_parent.right_child = None
    else:
        # se No nao tiver pai ele eh a raiz
        # seta a raiz para None
        self.root = None

# CASO 2: No com 1 filho
if node_children == 1:

    # encontra o filho do No (verifica se eh left_child ou right_child)
    if node.left_child != None:
        child = node.left_child
    else:
        child = node.right_child

    if node_parent != None:
        # passa o filho (child) do No deletado para o avo
        if node_parent.left_child == node:
            node_parent.left_child = child
        else:
            node_parent.right_child = child
    else:
        self.root = child

    # altera o pai da child para seu avo
    child.parent = node_parent

# CASO 3: No com dois filhos
if node_children == 2:

    # pega o valor que vai suceder o No que sera "removido"
    successor = min_value_node(node.right_child)

    # copia o valor do sucessor para o No que desejamos "remover"
    node.value = successor.value

    # deleta o sucessor agora que o seu valor esta copiado no No "removido"
    # o processo é recursivo
    self._delete_node(successor)

return
```