Árvores AVL

Estruturas de Dados

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Informalmente, uma árvore é **balanceada** (na altura), quando a diferença das alturas das suas subárvores é "pequena" e as subárvores são **balanceadas**. Ou ainda, uma árvore que tem altura $O(\lg n)$.

Uma **árvore binária de busca auto balanceada** é aquela que se mantém balanceada após cada modificação.

Existem diversos tipos de ABB auto balanceadas, entre elas: AVL, rubro-negra e treap.

A árvore AVL (nomeada a partir do nome dos criadores - **A**delson-**V**elsky and **L**andis) foi a primeira árvore auto balanceada a ser criada (1962).

Uma **árvore AVL**, é uma ABB de busca, que quando não é vazia, tem uma raiz *t* e:

- A diferença absoluta da altura das subárvores a direita e a esquerda de *t* é no máximo 1;
- As subárvores a esquerda e direita de *t* são **AVL**.

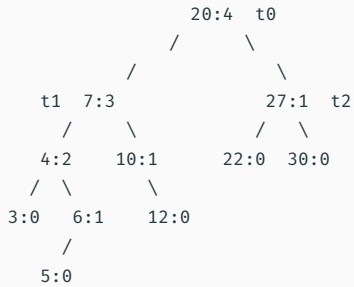
Para representar uma AVL, é preciso adicionar um atributo `altura` na classe `No`.

```python
@dataclass
class No:
    esq: Arvore
    val: int
    dir: Arvore
    altura: int

Arvore = No | None

def altura(t: Arvore) -> Int:
    '''
    Devolve a altura da árvore *t*.
    Devolve -1 se *t* é None.
    '''
    if t is None:
        return -1
    else:
        return t.altura
```

Considere as seguintes árvore, onde cada nó é representado pelo seu valor e altura:



A árvore t1 é AVL? Sim.

A árvore t2 é AVL? Sim.

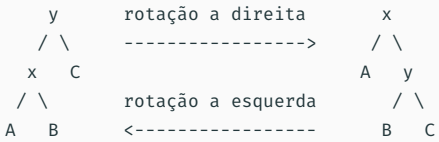
A árvore t0 é AVL? Não.

Quando um nó é inserido ou removido e a regra de balanceamento é violada, é preciso ajustar a árvore para restabelecer o balanceamento (**rebalancear**), o que é feito através de operações de rotações.

Uma **rotação** é uma operação que muda localmente a estrutura de uma ABB, mas mantém a propriedade de busca. No contexto de árvore AVL, a operação de rotação também deve ajustar o atributo altura dos nós envolvidos na rotação.

```
def atualiza_altura(no: No):
    '''
    Atualiza a altura do *no*.
    Requer que a altura de *no.esq* e *no.dir* esteja corretas.
    '''
    no.altura = 1 + max(altura(no.esq), altura(no.dir))
```

Na figura abaixo, x e y representam valores armazenados nos nós e A, B e C representam subárvores.



Note que $A < x < B < y < C$ nas duas figuras. Ou seja, essas rotações não alteram a propriedade de ABB.

Veja uma animação da rotação e outras informações na página Tree rotation.

Rotação a esquerda

Projete uma função para fazer a rotação a esquerda de uma árvore não vazia com raiz `r`.

```
def rotaciona_esq(r: No) -> No:
    r'''
    Rotaciona a árvore com raiz *r*
    conforme o seguinte esquema:

         r              x
        / \            / \
       A   x    ->    r   C
          / \        / \
         B   C      A   B

    E devolve como nova raiz o nó que estava
    em *r.dir* quando a função foi chamada.

    Requer que *r.dir* não seja None.
    '''
```

```
def rotaciona_esq(r: No) -> No:
    assert r.dir is not None
    x = r.dir
    r.dir = x.esq
    x.esq = r
    atualiza_altura(r)
    atualiza_altura(x)
    return x
```

Exercício: projete a função para fazer a rotação a direita.

Exemplo de inserção em árvore AVL

Crie uma árvore AVL inserindo os seguintes itens na ordem que eles aparecem: 20, 10, 5, 30, 40, 25, 8, 2, 6, 9, 12, 14.

Feito e discutido em sala.

Você pode conferir o resultado usando este simulador.

Agora que vimos o funcionamento das operações de rotação, vamos sistematizar a forma que o rebalanceamento é feito a partir dessas operações.

Quando uma árvore AVL com raiz *r* tem a subárvore a esquerda ou a direita alterada, é necessário verificar se a propriedade de balanceamento foi violada. Como fazer essa verificação?

```
abs(altura(r.esq) - altura(r.dir)) == 2

# Desbalanceamento a esquerda
altura(r.esq) > altura(r.dir) + 1

# Desbalanceamento a direita
altura(r.dir) > altura(r.esq) + 1
```

Se existe violação, é necessário rebalancear a árvore usando rotações.

Note que o rebalanceamento não é feito em uma árvore qualquer, mas sim em uma árvore AVL que acabou de ficar desbalanceada devido a inserção ou remoção de um elemento.

Se a subárvore a esquerda tem altura maior que a subárvore a direita, então fazemos o rebalanceamento a esquerda, senão fazemos o rebalanceamento a direita.

Como o rebalanceamento a esquerda afeta as alturas das subárvores?

- Aumenta a altura da árvore a direita
- Diminui a altura da árvore a esquerda

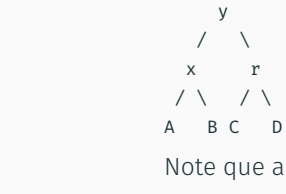
Como o rebalanceamento a direita afeta as alturas das subárvores?

- Aumenta a altura da árvore a esquerda
- Diminui a altura da árvore a direita

A forma que o rebalanceamento a esquerda de uma árvore AVL com raiz `r` é feito depende de qual das subárvores de `r.esq` tem maior altura.

Rebalanceamento a esquerda-esquerda

Esquerda-Esquerda – `altura(r.esq.esq) > altura(r.esq.dir)`



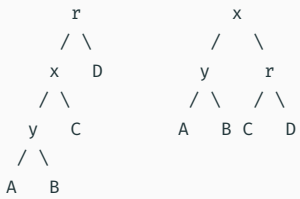
`h(r) > h(y) > h(x) > h(C) == h(D)`

O que é preciso para rebalancear a árvore?

```
return rotaciona_dir(r)
```


Note que a árvore tem uma nova raiz e que a altura da subárvore a esquerda diminui e a altura da subárvore a direita aumentou.

Esquerda-Direita – `altura(r.esq.esq) < altura(r.esq.dir)`



`h(r) > h(y) > h(x) > h(A) == h(D)`

O que é preciso para rebalancear a árvore?

```
# Transforma no caso esquerda-esquerda
r.esq = rotaciona_esq(r.esq)
return rotaciona_dir(r)
```

Note que a árvore fica com uma nova raiz e que a altura da subárvore a esquerda diminui e a altura da subárvore a direita aumenta.

Rebalanceamento a esquerda - código

Projete uma função que implemente o esquema de rebalanceamento a esquerda (e a correção da altura da árvore).

Rebalanceamento a esquerda - código

```
def rebalanceia_esq(r: No) -> No:
    '''
    Verifica o balanceamento de *r*, considerando o caso da subárvore a esquerda com maior altura,
    e faz o rebalanceamento e atualização das alturas se necessário. Devolve a raiz da árvore balanceada.
    '''

    assert r.esq is not None
    if altura(r.esq) - altura(r.dir) == 2:
        # r está desbalaceada
        if altura(r.esq.esq) > altura(r.esq.dir):
            # Caso Esquerda-Esquerda
            return rotaciona_dir(r)
        else:
            # Caso Esquerda-Direita
            assert altura(r.esq.dir) > altura(r.esq.esq)
            r.esq = rotaciona_esq(r.esq)
            return rotaciona_dir(r)
    else:
        # r está balaceada
        atualiza_altura(r)
        return r
```

Rebalanceamento a direita

Projete uma função que implemente o esquema de rebalanceamento a direita (e a correção da altura da árvore).

Fica como exercício.

Atualização da função de inserção

Atualize a função de inserção em ABB para árvores AVL.

```python
def insere(t: Arvore, val: int) -> No:
    if t is None:
        return No(None, val, None)
    else:
        if val < t.val:
            t.esq = insere(t.esq, val)
        elif val > t.val:
            t.dir = insere(t.dir, val)
        else: # val == t.val
            pass
        return t
```

```python
def insere(t: Arvore, val: int) -> No:
    if t is None:
        return No(None, val, None)
    else:
        if val < t.val:
            t.esq = insere(t.esq, val)
            t = rebalanceia_esq(t)
        elif val > t.val:
            t.dir = insere(t.dir, val)
            t = rebalanceia_dir(t)
        else: # val == t.val
            pass
        return t
```

Atualização da função de remoção

Atualize a função de remoção em ABB para árvores AVL.

Fica como exercício.

As funções `busca`, `insere` e `remove` definem a interface de uso da ABB e AVL.

Os exemplos servem tanto para mostrar para o usuário o uso da função e o seu comportamento. Os exemplos são bons testes iniciais para essas funções.

Já as funções de rotação e balanceamento são funções auxiliares, não fazem parte da interface para o usuário. Além disso, as funções são mais complicadas e interagem com outras funções. Os exemplos podem não ser suficientes para um bom teste.

Como proceder? Fazendo testes de propriedade.

Em um teste de propriedade executamos uma função e verificamos se a saída mantém alguma propriedade específica.

No caso de árvores AVL, podemos verificar se após cada inserção e remoção, a árvore continua sendo AVL.

Veja o código no arquivo `avl.py`.

Veja o arquivo `percursos.py`.

Implementação do TAD dicionário:

- Com arranjos e lista encadeada com busca linear, as operações de busca inserção e remoção tem tempo $O(n)$;
- Com arranjos ordenados e busca binária, a busca tem tempo $O(\lg n)$ e a inserção e remoção $O(n)$;
- Com ABB o tempo de busca, inserção e remoção é $O(h)$, onde h é a altura da árvore. No caso médio o tempo é de $O(\lg n)$ e no pior caso $O(n)$;
- Com árvore AVL o tempo de busca, inserção e remoção é $O(\lg n)$.

Podemos fazer melhor? Sim!

Quando usamos uma ABB ou AVL, precisamos manter os elementos “ordenados”, para podermos fazer uma busca binária.

A seguir vamos ver como fazer uma busca eficiente sem precisar manter os elementos ordenados.

Referências

Capítulo 10 - Árvores - Fundamentos de Python: Estruturas de dados. Kenneth A. Lambert. (Disponível na Minha Biblioteca na UEM).

Capítulo 12 - Árvores Binárias de Busca - Algoritmos: Teoria e Prática, 3a. edição, Cormen, T. at all.

Capítulo 6 - Binary Trees - Open Data Structures.