Busca e árvores

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Os TAD's Pilha, Fila e FilaDupla, permitem o armazenamento e recuperação de itens independente do conteúdo.

O TAD Lista tem apenas uma operação que é dependente do conteúdo: `remove_item`.

Vamos estudar um TAD em que a maioria das operações depende do conteúdo dos itens armazenados.

Dicionário

Um **dicionário**, também chamado de arranjo associativo, é um tipo abstrato de dados que representa uma coleção de associações chave-valor, onde cada chave é única.

As operações comuns em um dicionário são a associação de uma chave com um valor, a consulta do valor associado com uma chave e a exclusão de uma chave e o valor associado.

Dicionário

```python
class Dicionario:
    '''Uma coleção de associações chave-valor, onde
    cada chave é única'''

    def num_itens(self) -> int:
        '''Devolve a quantidade de chaves no dicionário.'''

    def associa(self, chave: str, valor: int):
        '''Associa a *chave* com o *valor* no dicionário.
        Se *chave* já está associada com um valor, ele
        é sustituído por *valor*.'''

    def get(self, chave: str) -> int | None:
        ''' Devolve o valor associado com *chave* no dicio-
        nário ou None se a chave não está no dicionário.'''

    def remove(self, chave: str):
        ''' Remove a *chave* e o valor associado com ela do
        dicionário. Não faz nada se a *chave* não está no
        dicionário.'''
```

```
>>> d = Dicionario()
>>> d.num_itens()
0
>>> d.associa('Jorge', 25)
>>> d.associa('Bia', 40)
>>> d.num_itens()
2
>>> d.get('Jorge')
25
>>> d.get('Bia')
40
>>> d.get('Andre') is None
True
>>> d.associa('Bia', 50)
>>> d.get('Bia')
50
>>> d.remove('Jorge')
>>> d.get('Jorge') is None
True
>>> d.remove('Ana')
```

Dicionário - Implementação com arranjo

Como podemos implementar o TAD Dicionário utilizando arranjo?

- Armazenamos um par chave-valor em cada posição do arranjo.
- Busca: busca por todos os itens, se a chave está presente, devolve o valor associado, senão devolve None.
- Associação: *busca* por todos os itens, se a chave está presente, atualiza o valor, senão adiciona a nova associação chave-valor no final.
- Remoção: *busca* por todos os itens, se a chave está presente, troca pelo último item e remove o último.

```python
@dataclass class Item:
    chave: str
    valor: int

class Dicionario:
    itens: list[Item]

    def __init__(self) -> None:
        self.itens = []

    def num_itens(self) -> int:
        return len(self.itens)

    def __busca(self, chave: str) -> int | None:
        '''Devolve a posição da *chave* ou
        None se a *chave* não está presente.'''
        for i in range(len(self.itens)):
            if self.itens[i].chave == chave:
                return i
        return None
```

Dicionário - Implementação com arranjo

```
class Dicionario:
    def associa(self, chave: str, valor: int):
        i = self.__busca(chave)
        if i is not None:
            self.itens[i].valor = valor
        else:
            self.itens.append(Item(chave, valor))

    def get(self, chave: str) -> int | None:
        # Operador walrus para simplificar :=
        if (i := self.__busca(chave)) is not None:
            return self.itens[i].valor
        else:
            return None

    def remove(self, chave: str):
        if (i := self.__busca(chave)) is not None:
            self.itens[i], self.itens[-1] = \
                self.itens[-1], self.itens[i]
            self.itens.pop()
```

Qual a complexidade de tempo das operações?

Todas tem tempo de execução $O(n)$ pois requerem uma busca que pode analisar todos os itens.

Será que podemos fazer melhor usando encadeamento linear?

Não... A busca ainda precisaria analisar todos os elementos.

Podemos fazer melhor? As operações dependem do conteúdo do item mas não estamos usando o conteúdo para organizar os itens.

Como organizar uma coleção de cartas Pokémon de maneira que seja possível encontrar uma carta rapidamente, isso é, sem precisar olhar todas elas?

Se as cartas estiverem em ordem alfabética, pode usar o seguinte método:

- Dividimos o monte mais ou menos no meio e olhamos para a carta que está na metade, se é a carta que estamos procurando, ótimo, terminamos! Senão repetimos o processo para a primeira metade, se a carta que estamos procurando vem antes em ordem alfabética, ou para a segunda metade – sem a carta que já vimos – se a carta que estamos procurando vem depois. Se o monte que estamos procurando está vazio, então a carta não está presente.

Este algoritmo é chamado de **busca binária**.

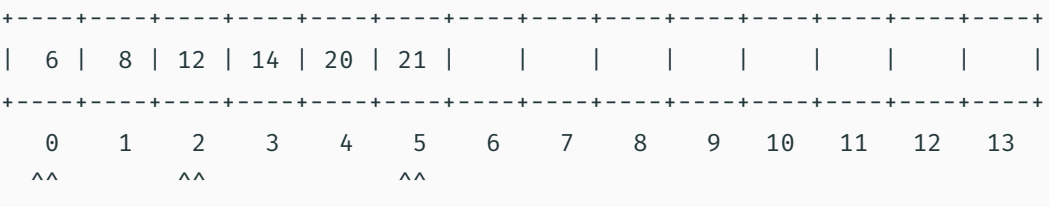
Exemplo

Busca pelo 20. `ini` e `fim` são o início do intervalo e `m = (ini + fim) // 2` é o "meio".

```
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
|  6 |  8 | 12 | 14 | 20 | 21 | 22 | 30 | 40 | 41 | 43 | 47 | 50 | 70 |
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
   0    1    2    3    4    5    6    7    8    9   10   11   12   13
  ^^                             ^^                             ^^
 ini                             m                            fim
```

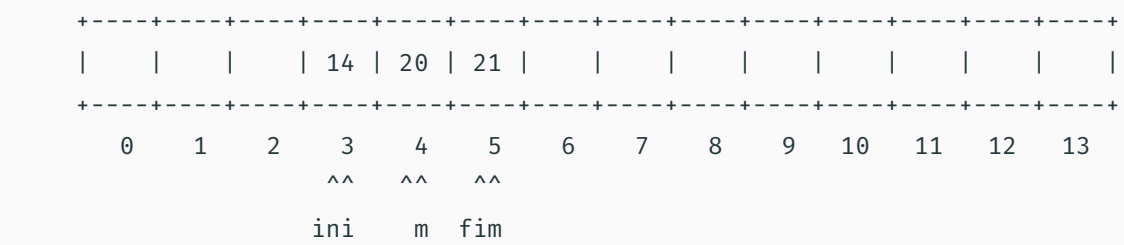
Exemplo - pesquisa pelo 20

Busca pelo 20. `ini` e `fim` são o início do intervalo e `m = (ini + fim) // 2` é o "meio".



Exemplo - pesquisa pelo 20

Busca pelo 20. `ini` e `fim` são o início do intervalo e `m = (ini + fim) // 2` é o "meio".



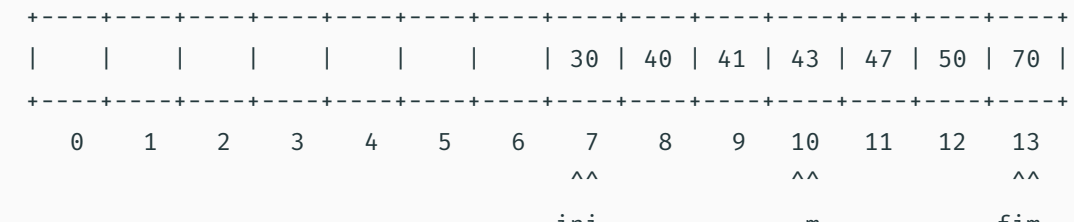
Exemplo - pesquisa pelo 42

Busca pelo 42. `ini` e `fim` são o início do intervalo e `m = (ini + fim) // 2` é o "meio".

```
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
|  6 |  8 | 12 | 14 | 20 | 21 | 22 | 30 | 40 | 41 | 43 | 47 | 50 | 70 |
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
   0    1    2    3    4    5    6    7    8    9   10   11   12   13
  ^^                            ^^                                ^^
 ini                            m                               fim
```

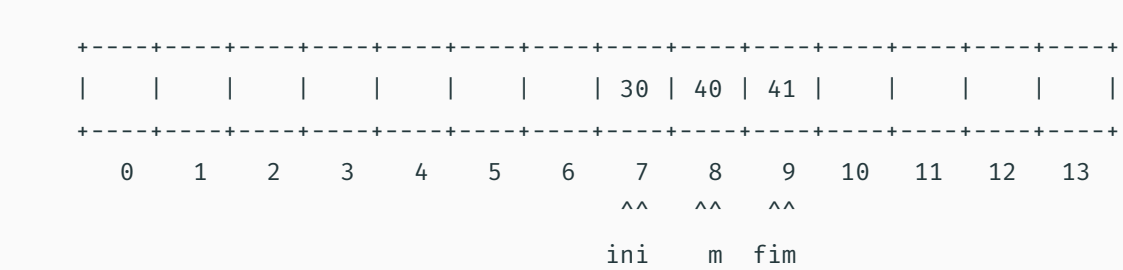
Exemplo - pesquisa pelo 42

Busca pelo 42. `ini` e `fim` são o início do intervalo e `m = (ini + fim) // 2` é o "meio".



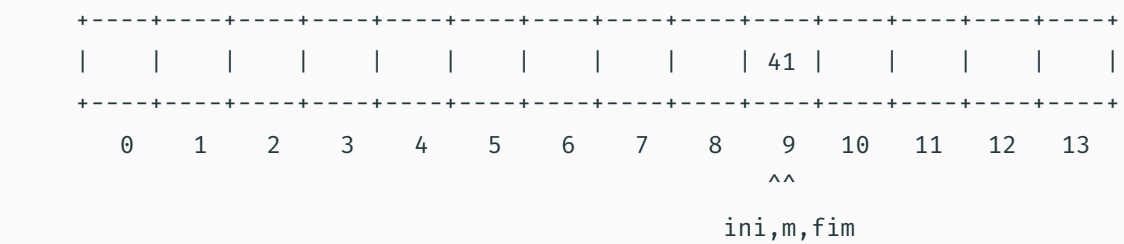
Exemplo - pesquisa pelo 42

Busca pelo 42. `ini` e `fim` são o início do intervalo e `m = (ini + fim) // 2` é o “meio”.



Exemplo - pesquisa pelo 42

Busca pelo 42. `ini` e `fim` são o início do intervalo e `m = (ini + fim) // 2` é o “meio”.



Exemplo - pesquisa pelo 42

Busca pelo 42. `ini` e `fim` são o início do intervalo e `m = (ini + fim) // 2` é o "meio".

Quantas comparações no máximo são feitas entre a chave e um valor do arranjo? Quantas divisões sucessivas por 2 são necessárias para que um valor n chegue em 1?

Supondo que n seja uma potência de dois, e sendo i a quantidade de divisões, temos

$$\frac{n}{2^i} = 1 \rightarrow n = 2^i$$

Aplicando $\log_2$ obtemos

$$\log_2 n = \log_2 2^i \rightarrow i = \lg n$$

Portanto, a complexidade de tempo da busca binária é $O(\lg n)$.

Como as complexidades de tempo da busca linear e binária se comparam?

n	Busca Linear	Busca binária
10^1	10	≈ 4
10^2	100	≈ 7
10^3	1.000	≈ 10
10^6	1.000.000	≈ 20
10^9	1.000.000.000	≈ 30

Existem várias formas de implementar a busca binária (veja a lista de exercícios!).

A seguir mostramos um implementação iterativa que devolve um índice onde a chave está na lista ou onde ela deveria estar. Isto é útil pois podemos usar esse índice para inserir a chave se ela não está presente.

Implementação da busca binária

```python
def busca_binaria(valores: list[int], chave: int) -> int:
    '''
    Se *chave* está presente em *valores*, devolve o
    índice i tal que *valores[i] == chave*. Senão devolve
    o índice i tal que a inserção de *chave* na posição
    *i* de *valores* mantém *valores* em ordem não
    decrescente.

    Requer que *valores* esteja em ordem não decrescente.

    Exemplos
    >>> busca_binaria([6, 8, 10, 12, 20], 7)
    1
    >>> busca_binaria([6, 8, 10, 12, 20], 20)
    4
    '''
```

```python
ini = 0
fim = len(valores) - 1
while ini <= fim:
    m = (ini + fim) // 2
    if chave == valores[m]:
        return m
    elif chave < valores[m]:
        fim = m - 1
    else: # chave > valores[m]
        ini = m + 1
return ini
```

O que é preciso para usar a busca binária na implementação do dicionário?

Manter as associações chave-valor ordenadas pela chave (a implementação fica como exercício).

Como isso afeta a complexidade de tempo de `associa` e `remove`? Não afeta! A complexidade continua sendo $O(n)$.

E a complexidade da busca? Passa a ser $O(\lg n)$.

De forma geral, a implementação usando arranjo ordenado e busca binária de dicionário é adequada? Se a quantidade de consultas for muito maior que a quantidade de alterações, então pode ser uma boa.

E a implementação usando arranjo com busca linear? Pode ser adequada se a quantidade de elementos for pequena.

Podemos melhor o tempo das operações de alteração? Sim! Mas antes precisamos revisar recursividade.

Podemos fazer uma busca binária em um encadeamento linear de forma eficiente? Não, pois não temos acesso em tempo constante ao elemento “do meio”.

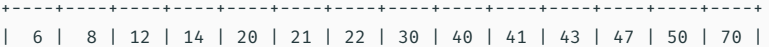
Podemos fazer uma busca binária em *algum tipo de encadeamento* de forma eficiente?

Porque iríamos querer fazer isso? Em um arranjo é possível fazer busca binária eficiente, mas a inserção e remoção tem complexidade de tempo $O(n)$.

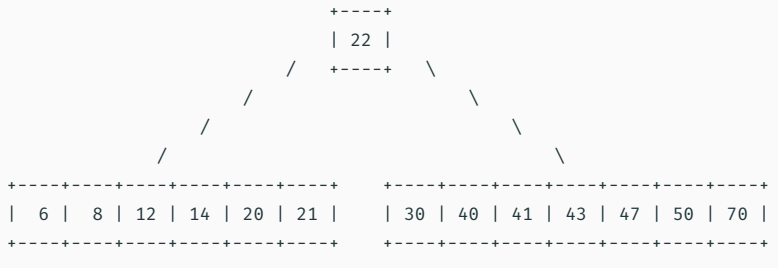
Se *conseguirmos* fazer uma busca binária eficiente em um encadeamento, *talvez* possamos fazer inserção e remoção de forma eficiente também!

Vamos analisar uma sequência ordenada de elementos e tentar criar um encadeamento que permita a realização de uma busca binária.

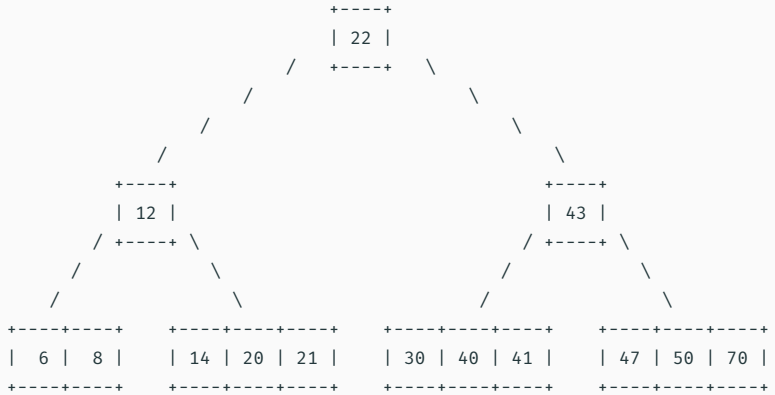
Encadeamento e busca binária?



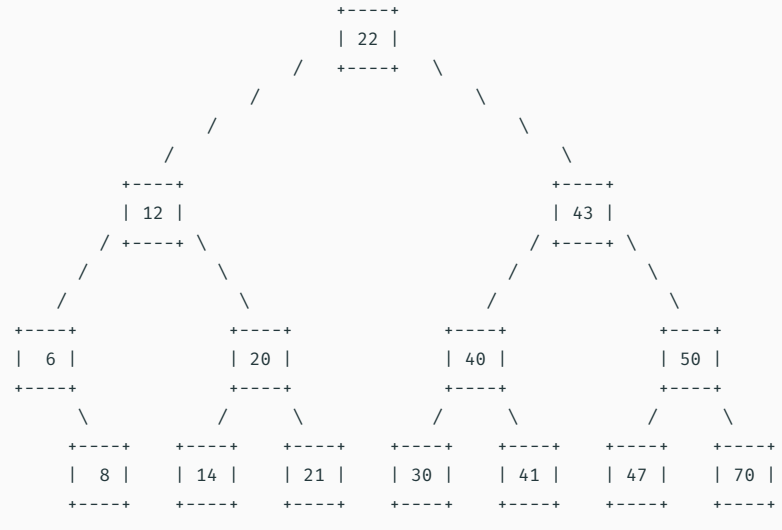
Encadeamento e busca binária?



Encadeamento e busca binária?



Encadeamento e busca binária?



Essa forma de representar uma coleção de valores é chamada de árvore, especificamente, uma árvore binária.

Vire de ponta cabeça para ver a árvore!!! As árvores em computação crescem para baixo!

Como podemos definir uma árvore binária?

Uma **árvore binária** é:

- Vazia (`None`); ou
- Um nó (`No`) com um valor e uma **árvore binária** a esquerda e uma **árvore binária** a direita.

Um nó é a **raiz** da árvore composta por ele e por suas subárvores.

Se *A* é o nó raiz de uma árvore e *B* é o nó raiz de uma das subárvores de *A*, então, *B* é **filho** de *A* e *A* é **pai** de *B*.

Um nó *A* é **ancestral** de um nó *B* se *A* é pai de *B* ou pai de algum ancestral de *B*. Se *A* é ancestral de *B*, então *B* é **descendente** de *A*.

Como representar uma árvore binária em Python?

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

Arvore = No | None
```

Como `Arvore` é um tipo com autorreferência, podemos derivar um modelo de função recursiva para processar uma árvore binária:

```
def fn_para_ab(t: Arvore) -> ...:
    if t is None:
        return ...
    else:
        return t.val ... \
               fn_para_ab(t.esq) ... \
               fn_para_ab(t.dir)
```

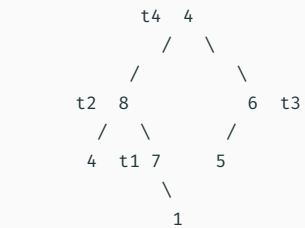
Note que da mesma forma que não podemos fazer uma busca binária em um arranjo qualquer, também não podemos fazer uma busca binária em uma árvore binária qualquer! É necessário adicionar restrições a árvore que veremos daqui a pouco.

Antes vamos ver alguns exemplos e definições.

Exemplo de criação de árvore

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

Arvore = No | None
```

Escreva o código Python para criar as seguintes árvores:




```
>>> t1 = No(None, 7, No(None, 1, None))
>>> t1
No(esq=None, val=7, dir=No(esq=None, val=1, dir=None))

>>> t2 = No(No(None, 4, None), 8, t1)
>>> t3 = No(No(None, 5, None), 6, None)
>>> t4 = No(t2, 4, t3)
```

Como acessar o valor 1 a partir de `t4`?

```
>>> t4.esq.dir.dir.val
```

Como adicionar uma nova subárvore (valor da raiz 4) a esquerda de `t3` a partir de `t4`?

```
>>> t4.dir.dir = No(None, 4, None)
```

Como remover a subárvore a direita de `t2` a partir de `t4`?

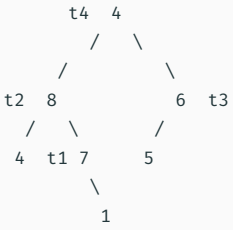
```
>>> t4.esq.dir = None
```

Número de folhas

O **grau** de um nó é a quantidade de subárvores do nó.

Um nó com grau zero é chamado de **nó folha**. Um nó que não é uma folha é chamado de **nó interno**.

Projete uma função que determine a quantidade de nós folhas de uma árvore.



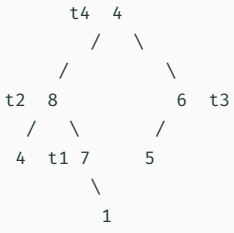

```python
def num_folhas(t: Arvore) -> int:
    '''
    Determina a quantidade de folhas em *t*.
    Uma folha é um nó sem nenhum filho.
    >>> # Criação das árvores e alguns exemplos omitidos
    >>> num_folhas(t2)
    2
    >>> num_folhas(t3)
    1
    >>> num_folhas(t4)
    3
    '''

    if t is None:
        return ...
    else:
        return self.val ... \
               num_folhas(t.esq) ... \
               num_folhas(t.dir)
```

Número de folhas

O **grau** de um nó é a quantidade de subárvores do nó.

Um nó com grau zero é chamado de **nó folha**. Um nó que não é uma folha é chamado de **nó interno**.

Projete uma função que determine a quantidade de nós folhas de uma árvore.




```
def num_folhas(t: Arvore) -> int:
    '''
    Determina a quantidade de folhas em *t*.
    Uma folha é um nó sem nenhum filho.
    >>> # Criação das árvores e alguns exemplos omitidos
    >>> num_folhas(t2)
    2
    >>> num_folhas(t3)
    1
    >>> num_folhas(t4)
    3
    '''

    if t is None:
        return 0
    else:
        if t.esq is None and t.dir is None:
            return 1
        else:
            return num_folhas(t.esq) + num_folhas(t.dir)
```

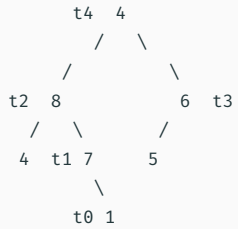
Nível e altura

O **nível** de um nó em uma árvore é:

- `0` se o nó é a raiz da árvore; ou
- O **nível** do pai mais 1 caso contrário

A **altura** (ou profundidade) de um nó é o máximo entre os níveis de todas as folhas da árvore com raiz nesse nó.

De de outra forma, é o comprimento do caminho mais longo deste o nó até uma folha.



Em relação a árvore com raiz `t4`, qual é o nível de:

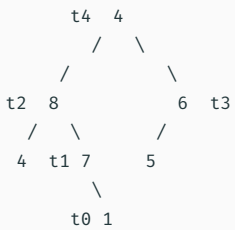
`t4`? 0. `t2`? 1. `t3`? 1. `t1`? 2. `t0`? 3.

Qual é a altura da árvore:

`t0`? 0. `t1`? 1. `t2`? 2. `t3`? 1. `t4`? 3.

Nível

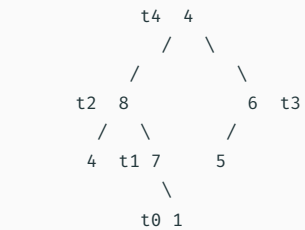
Projete uma função que encontre todos os valores em um determinado nível de uma árvore.




```python
def valores_nivel(t: Arvore, n: int) -> list[int]:
    '''
    Devolve os nós que estão no nível *n* de *t*.
    >>> valores_nivel(None, 0)
    []
    >>> valores_nivel(t4, 0)
    [4]
    >>> valores_nivel(t4, 2)
    [4, 7, 5]
    >>> valores_nivel(t4, 3)
    [1]
    '''

    if t is None:
        return ... n
    else:
        return n ... \
               t.val ... \
               valores_nivel(t.esq, ...) ... \
               valores_nivel(t.dir, ...) ...
```

Nível

Projete uma função que encontre todos os valores em um determinado nível de uma árvore.




```
def valores_nivel(t: Arvore, n: int) -> list[int]:
    '''
    Devolve os nós que estão no nível *n* de *t*.
    >>> valores_nivel(None, 0)
    []
    >>> valores_nivel(t4, 0)
    [4]
    >>> valores_nivel(t4, 2)
    [4, 7, 5]
    >>> valores_nivel(t4, 3)
    [1]
    '''
    if t is None:
        return []
    else:
        if n == 0:
            return [t.val]
        else:
            return valores_nivel(t.esq, n - 1) + \
                   valores_nivel(t.dir, n - 1)
```

O que é preciso para podemos fazer uma busca binária em um árvore binária? Que ela seja de busca!

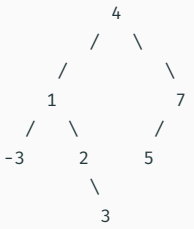
Uma **árvore binária de busca** (ABB) é uma árvore binária, que quando não é vazia, tem uma raiz t e:

- Todos os elementos da subárvore a esquerda de t são menores que o valor armazenado em t;
- Todos os elementos da subárvore a direita de t são maiores que o valor armazenado em t;
- As subárvores a esquerda e a direta de t são **árvores binárias de busca**.

Busca em árvore binária de busca

Dessa forma, quando estamos procurando um valor *v* e *v* é menor que o valor na raiz, continuamos a busca na subárvore a esquerda, se *v* é maior que o valor da raiz, continuamos a busca na subárvore a direita.

Implemente o algoritmo de busca para uma árvore binária de busca.

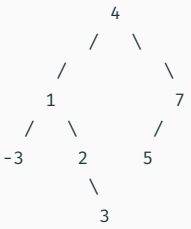



```
def busca(t: Arvore, val: int) -> bool:
    r'''
    Devolve True se *val* está em *t*,
    False caso contrário.
    >>> busca(None, 10)
    False
    >>> busca(t, 2)
    True
    >>> busca(t, 6)
    False
    '''
    if t is None:
        return ... val
    else:
        return val ... \
               t.val ... \
               busca(t.esq, val) ... \
               busca(t.dir, val) ...
```

Busca em árvore binária de busca

Dessa forma, quando estamos procurando um valor *v* e *v* é menor que o valor na raiz, continuamos a busca na subárvore a esquerda, se *v* é maior que o valor da raiz, continuamos a busca na subárvore a direita.

Implemente o algoritmo de busca para uma árvore binária de busca.




```python
def busca(t: Arvore, val: int) -> bool:
    r'''
    Devolve True se *val* está em *t*,
    False caso contrário.
    >>> busca(None, 10)
    False
    >>> busca(t, 2)
    True
    >>> busca(t, 6)
    False
    '''
    if t is None:
        return False
    elif val == t.val:
        return True
    elif val < t.val:
        return busca(t.esq, val)
    else:  # val > t.val
        return busca(t.dir, val)
```

Busca em árvore binária de busca

Dessa forma, quando estamos procurando um valor *v* e *v* é menor que o valor na raiz, continuamos a busca na subárvore a esquerda, se *v* é maior que o valor da raiz, continuamos a busca na subárvore a direita.

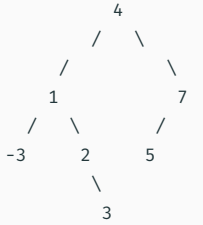
Implemente o algoritmo de busca para uma árvore binária de busca.


```python
def busca(t: Arvore, val: int) -> bool:
    r'''
    Devolve True se *val* está em *t*,
    False caso contrário.
    >>> busca(None, 10)
    False
    >>> busca(t, 2)
    True
    >>> busca(t, 6)
    False
    '''
    r = t
    while r is not None:
        if val == r.val:
            return True
        elif val < r.val:
            r = r.esq
        else:  # val > r.val
            r = r.dir
    return False
```

Complexidade de tempo da busca em ABB

```python
def busca(t: Arvore, val: int) -> bool:
    if t is None:
        return False
    elif val == t.val:
        return True
    elif val < t.val:
        return busca(t.esq, val)
    else:  # val > t.val
        return busca(t.dir, val)
```

```python
def busca(t: Arvore, val: int) -> bool:
    r = t
    while r is not None:
        if val == r.val:
            return True
        elif val < r.val:
            r = r.esq
        else:  # val > r.val
            r = r.dir
    return False
```



Qual é a complexidade de tempo do algoritmo de busca em árvore binária de busca? $O(h)$, onde h é a altura da árvore.

Qual é a relação entre a quantidade n de elementos da árvore e h? Qual é o limite inferior de h? $\lg(n)$. Qual é o limite superior de h? $n - 1$.

O que podemos concluir sobre isso? Para que a busca em uma ABB seja eficiente, precisamos manter a altura da árvore perto do valor mínimo.

Fato: uma ABB criada com n valores aleatórios tem altura média de $1.39 \lg n$.

Então, se as chaves usadas nas inserções e remoções tem uma distribuição aleatória, a ABB resultante tem uma altura pequena.

Como manter a altura pequena em uma árvore para qualquer distribuição de chaves? Veremos daqui a pouco.

Agora vamos ver como inserir e remover valores de uma ABB sem se preocupar com a altura.

Projete uma função que insira um novo valor, se ainda não estiver presente, em uma árvore binária de busca.

Quais são os tipos dos parâmetros da função? `Arvore` e `int`.

Quais deve ser o tipo de saída da função? `None`?

```
def insere(t: Arvore, val: int) -> None:
    '''
    Insere *val* em *t* mantendo as
    propriedades de ABB.
    Requer que *t* seja uma ABB.
    >>> r = None
    >>> insere(r, 10)
    >>> r
    No(esq=None, val=10, dir=None)
    '''
```

É possível implementar a função para que o exemplo funcione? Não!

Dentro da função é preciso fazer `t` referenciar um novo nó, mas quando fazemos isso, `r` permanece inalterado...

Como resolver essa questão? Alterando o tipo de retorno para `No` e atribuindo o retorno para `r`.

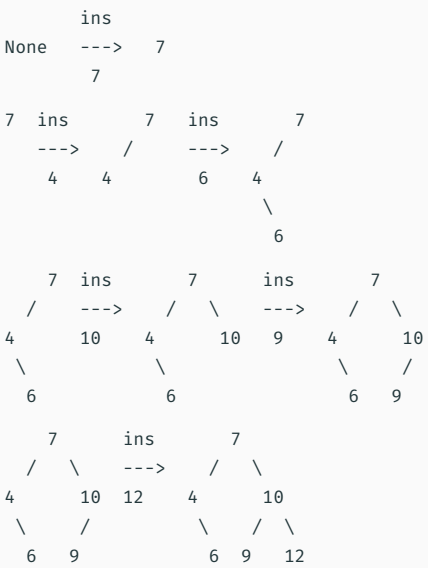
```
def insere(t: Arvore, val: int) -> No:
    '''
    Devolve a raiz da ABB que é o resultado
    da inserção de *val* em *t*.
    Se *val* já está em *t*, devolve *t*.
    Requer que *t* seja uma ABB.

    Exemplo
    >>> r = None
    >>> r = insere(None, 10)
    >>> r
    No(esq=None, val=10, dir=None)
    '''
```

Como proceder com a implementação?

Partindo do modelo!

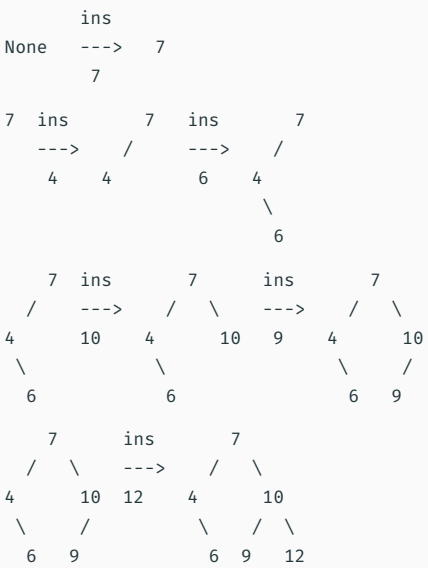
Mas temos que lembrar que quando chamamos `insere` é preciso armazenar o resultado no lugar da raiz que foi chamada como parâmetro.

Inserção em árvore binária de busca



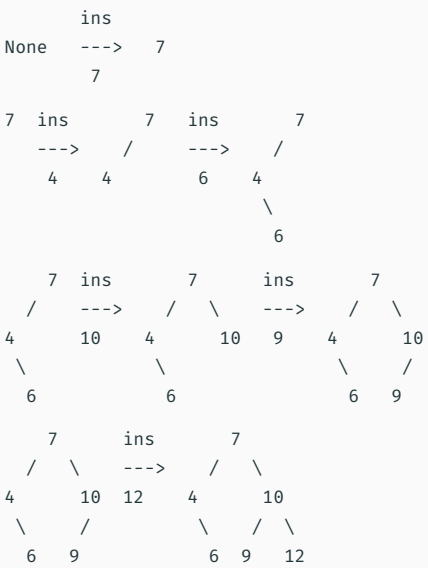

```python
def insere(t: Arvore, val: int) -> No:
    '''
    Devolve a raiz da ABB que é o resultado
    da inserção de *val* em *t*.
    Se *val* já está em *t*, devolve *t*.
    Requer que *t* seja uma ABB.
    '''
    if t is None:
        return ... val
    else:
        val ...
        t.val ...
        insere(t.esq, val) ...
        insere(t.dir, val) ...
        return ...
```

Inserção em árvore binária de busca



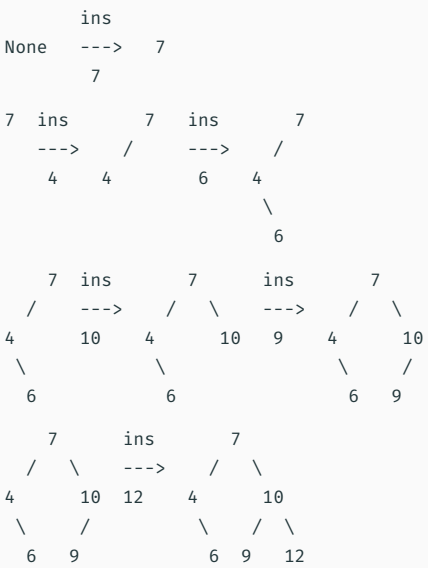

```python
def insere(t: Arvore, val: int) -> No:
    '''
    Devolve a raiz da ABB que é o resultado
    da inserção de *val* em *t*.
    Se *val* já está em *t*, devolve *t*.
    Requer que *t* seja uma ABB.
    '''
    if t is None:
        return ... val
    else:
        val ...
        t.val ...
        t.esq = insere(t.esq, val) ...
        t.dir = insere(t.dir, val) ...
        return ...
```

Inserção em árvore binária de busca




```
def insere(t: Arvore, val: int) -> No:
    '''
    Devolve a raiz da ABB que é o resultado
    da inserção de *val* em *t*.
    Se *val* já está em *t*, devolve *t*.
    Requer que *t* seja uma ABB.
    '''
    if t is None:
        return No(None, val, None)
    else:
        val ...
        t.val ...
        t.esq = insere(t.esq, val) ...
        t.dir = insere(t.dir, val) ...
        return ...
```

Inserção em árvore binária de busca




```python
def insere(t: Arvore, val: int) -> No:
    '''
    Devolve a raiz da ABB que é o resultado
    da inserção de *val* em *t*.
    Se *val* já está em *t*, devolve *t*.
    Requer que *t* seja uma ABB.
    '''
    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
```

Inserção em árvore binária de busca

```
def insere(t: Arvore, val: int) -> No:
    '''
    Devolve a raiz da ABB que é o resultado
    da inserção de *val* em *t*.
    Se *val* já está em *t*, devolve *t*.
    Requer que *t* seja uma ABB.
    '''
    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
```

Qual é a complexidade de tempo da inserção?

$O(h)$.

$O(1)$ operações para cada nó analisado. No pior caso todos os nós de um caminho de tamanho máximo são analisados.

Projete uma função que remova um valor, se estiver presente, de uma árvore binária de busca.

```
def remove(t: Arvore, val: int) -> Arvore:
    '''
    Devolve a raiz da ABB que é o resultado
    da remoção de *val* de *t*.
    Se *val* não está em *t*, devolve *t*.
    Requer que *t* seja uma ABB.

    Exemplo
    >>> r = No(None, 10, None)
    >>> r = remove(r, 10)
    >>> r is None
    True
    '''
```

Como proceder com a implementação?

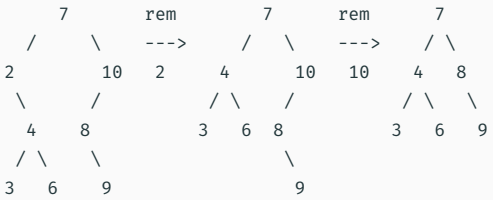
Partindo do modelo!

Mas temos lembrar que quando chamamos `remove` é preciso armazenar o resultado no lugar da raiz que foi chamada como parâmetro.

Remoção em árvore binária de busca

Remoção de folha: retorna None.

Remoção de nó sem subárvore a esq ou dir



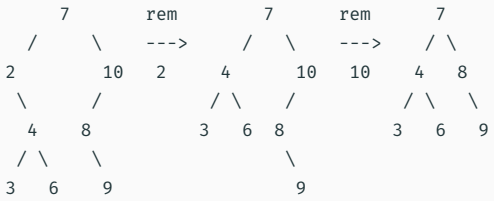
Remoção de nó com subárvore a esq e a dir


```
def remove(t: Arvore, val: int) -> Arvore:
    '''
    Devolve a raiz da ABB que é o resultado
    da remoção de *val* de *t*.
    Se *val* não está em *t*, devolve *t*.
    Requer que *t* seja uma ABB.
    '''
    if t is None:
        return ... val
    else:
        val ...
        t.val ...
        remove(t.esq, val) ...
        remove(t.dir, val) ...
        return ...
```

Remoção em árvore binária de busca

Remoção de folha: retorna None.

Remoção de nó sem subárvore a esq ou dir



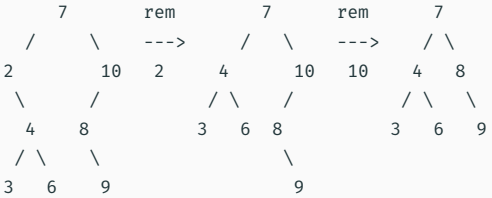
Remoção de nó com subárvore a esq e a dir


```python
def remove(t: Arvore, val: int) -> Arvore:
    '''
    Devolve a raiz da ABB que é o resultado
    da remoção de *val* de *t*.
    Se *val* não está em *t*, devolve *t*.
    Requer que *t* seja uma ABB.
    '''
    if t is None:
        return None
    else:
        val ...
        t.val ...
        t.esq = remove(t.esq, val) ...
        t.dir = remove(t.dir, val) ...
        return ...
```

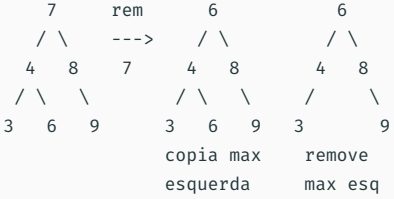
Remoção em árvore binária de busca

Remoção de folha: retorna None.

Remoção de nó sem subárvore a esq ou dir



Remoção de nó com subárvore a esq e a dir




```python
def remove(t: Arvore, val: int) -> Arvore:
    '''
    Devolve a raiz da ABB que é o resultado
    da remoção de *val* de *t*.
    Se *val* não está em *t*, devolve *t*.
    Requer que *t* seja uma ABB.
    '''
    if t is None:
        return None
    elif val < t.val:
        t.esq = remove(t.esq, val)
        return t
    elif val > t.val:
        t.dir = remove(t.dir, val)
        return t
    else: # val == t.val
        val, t.val, t.esq, t.dir
        ... = remove(t.esq, ...) ...
        ... = remove(t.dir, ...) ...
        return ...
```

Remoção em árvore binária de busca

Remoção de folha: retorna None.

Remoção de nó sem subárvore a esq ou dir

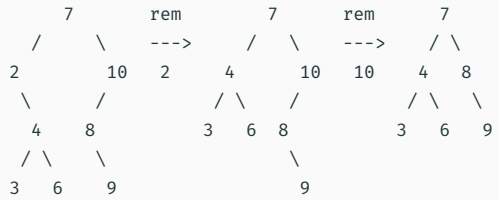
Remoção de nó com subárvore a esq e a dir


```python
def remove(t: Arvore, val: int) -> Arvore:
    if t is None:
        return None
    elif val < t.val:
        t.esq = remove(t.esq, val)
        return t
    elif val > t.val:
        t.dir = remove(t.dir, val)
        return t
    else: # val == t.val
        if t.esq is None:
            return t.dir
        elif t.dir is None:
            return t.esq
        else:
            val, t.val, t.esq, t.dir ...
            ... = remove(t.esq, ...) ...
            ... = remove(t.dir, ...) ...
            return ...
```

Remoção em árvore binária de busca

Remoção de folha: retorna None.

Remoção de nó sem subárvore a esq ou dir



Remoção de nó com subárvore a esq e a dir


```
def remove(t: Arvore, val: int) -> Arvore:
    if t is None:
        return None
    elif val < t.val:
        t.esq = remove(t.esq, val)
        return t
    elif val > t.val:
        t.dir = remove(t.dir, val)
        return t
    else: # val == t.val
        if t.esq is None:
            return t.dir
        elif t.dir is None:
            return t.esq
        else:
            m = maximo(t.esq)
            t.val = m
            t.esq = remove(t.esq, m)
            return t
```

```
def remove(t: Arvore, val: int) -> Arvore:
    if t is None:
        return None
    elif val < t.val:
        t.esq = remove(t.esq, val)
        return t
    elif val > t.val:
        t.dir = remove(t.dir, val)
        return t
    else: # val == t.val
        if t.esq is None:
            return t.dir
        elif t.dir is None:
            return t.esq
        else:
            m = maximo(t.esq)
            t.val = m
            t.esq = remove(t.esq, m)
            return t
```

Qual é a complexidade de tempo da remoção?

$O(h)$.

$O(1)$ operações para cada nó analisado. No pior caso, todos os nós de um caminho de tamanho máximo são analisados.

A complexidade de tempo das operações de busca, inserção e remoção em uma ABB tem tempo de execução $O(h)$.

Como vimos anteriormente, se as chaves usadas nas inserções e remoções têm distribuição aleatória, então a altura média da ABB é $O(\lg n)$.

Como garantir que a altura seja $O(\lg n)$ para uma distribuição qualquer de chaves?

Mantendo a árvore balanceada.

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:

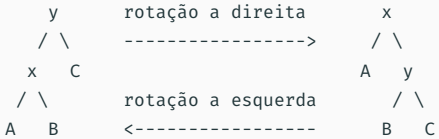
- A diferença absoluta da altura das subárvores a direita e a esquerda de *t* é no máximo 1;
- As subárvores a esquerda e direita de *t* são **AVL**.

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

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.

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.

Após essa experiência prática, vamos sistematizar a forma que o rebalanceamento é feito!

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.

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

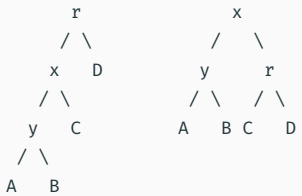
`h(r) > h(y) > h(x) > h(C) == h(D)`

O que é preciso para rebalancear a árvore?

```
return rotaciona_dir(r)
```


Note que 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`.

Discutido em sala.

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 $O(\lg n)$;
- 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.