# Árvores binárias de busca

Estruturas de Dados

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

# Encadeamento e busca binária?

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?

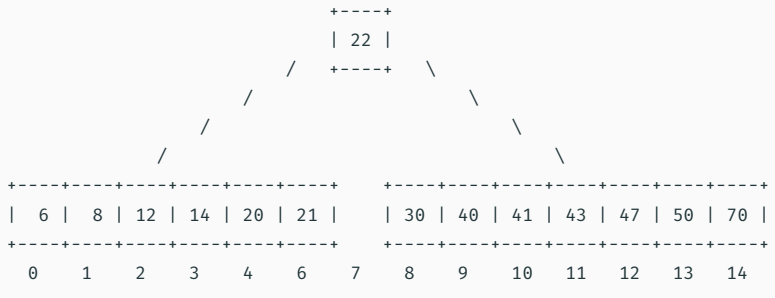
Porque iríamos querer fazer isso? Em um arranjo é possível fazer busca binária eficiente, mas a inserção e remoção tem complexidade de tempo $O(n)$.

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

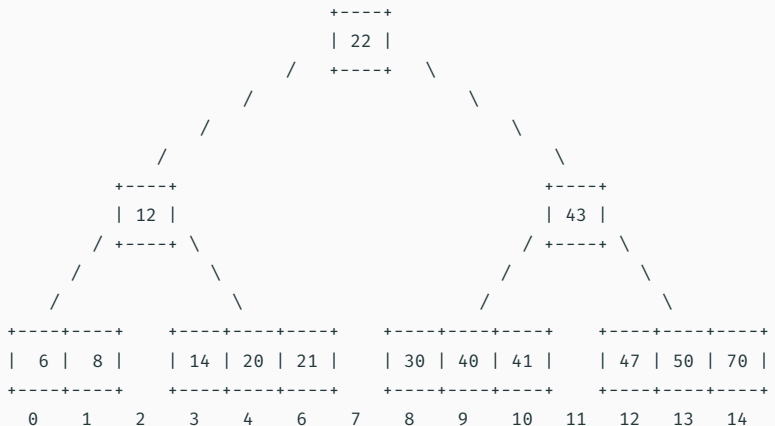
Vamos analisar uma sequência ordenada de elementos e tentar criar um encadeamento que permita a realização de uma busca binária.

# Encadeamento e busca binária?

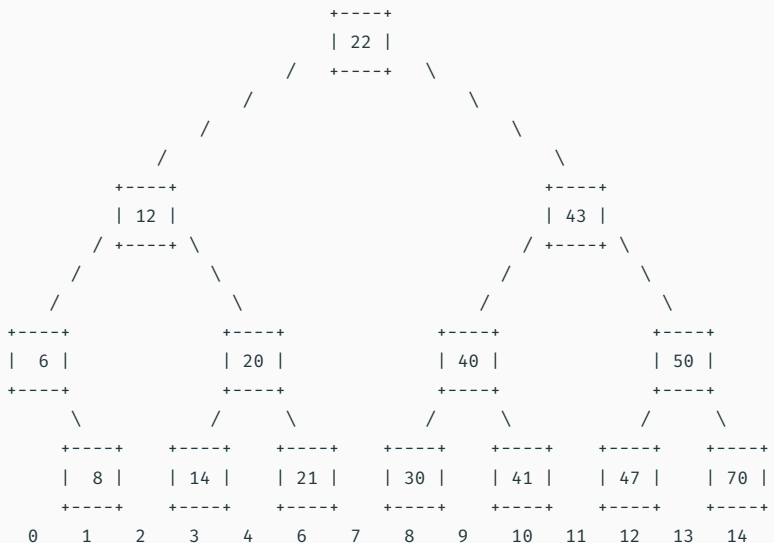
# Encadeamento e busca binária?



# Encadeamento e busca binária?



# Encadeamento e busca binária?



Essa tipo de estrutura é chamada de árvore binária, especificamente, uma **árvore binária de busca**.

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

Como podemos definir uma árvore binária?

# Definição de árvore

Uma **árvore binária** é:

- Vazia; ou
- Um nó com uma chave e uma **árvore binária** a esquerda e uma **árvore binária** a direita.

Note que esta definição de árvore não impõe nenhuma restrição sobre as chaves.

Para podemos usar uma árvore binária para fazer uma busca binária, vamos precisar adicionar restrições sobre as chaves.

Mas antes, vamos ver alguns definições e exemplos.

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*.


Quem são os filhos do nó 4? O nós 8 e 6.

Quem é o pai do nó 7? O nó 8.

Quem são os descendentes do nó 8? Os nós 2, 7 e 1.

Quem são os ancestrais do no 5? Os nós 6 e 4.

# Árvores binárias em Python

Como representar uma árvore binária?

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

Arvore = No | None
```

Como criar a seguinte árvore?


```
>>> t1 = No(None, 7, No(None, 1, None))
>>> t1
No(esq=None, chave=7, dir=No(esq=None, chave=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 a chave 1 a partir de `t4`?

```
>>> t4.esq.dir.chave
```

Como adicionar uma nova subárvore (chave 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
```

Como projetar funções que processam árvores?

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

Arvore = No | None
```

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

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

Como o modelo guia a implementação da função?

O modelo indica que, para processarmos um árvore, temos que ter pelo menos dois casos, uma para a árvore vazia, e outro para a árvore não vazia.

Além disso, no caso de árvore não vazia, o modelo sugere chamar a função recursivamente para as árvores a esquerda e a direita. (Por que?)

O nosso trabalho é determinar como combinar a chave do nó raiz com as respostas das chamadas recursivas para obter a resposta da função.

Nos exemplos a seguir, partimos do modelo e fazemos a implementação de algumas funções.

Tente completar as funções antes de ver as repostas.

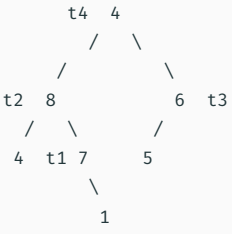
# Número de folhas

O **grau** de um nó é o número de filhos do nó.

Um **nó folha** é aquele que tem grau 0.

Um **nó interno** é aquele que não é folha.

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.
    >>> num_folhas(t2)
    2
    >>> num_folhas(t3)
    1
    >>> num_folhas(t4)
    3
    '''
    if t is None:
        return ...
    else:
        return self.chave ... \
               num_folhas(t.esq) ... \
               num_folhas(t.dir)
```

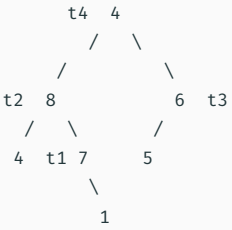
# Número de folhas

O **grau** de um nó é o número de filhos do nó.

Um **nó folha** é aquele que tem grau 0.

Um **nó interno** é aquele que não é folha.

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.
    >>> 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)
```

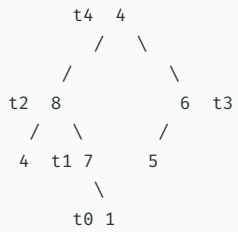
# Nível e altura

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




Em relação a `t4`, qual é o nível de:

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

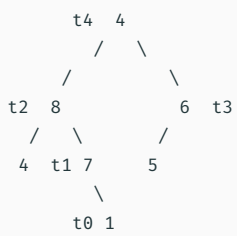
Qual é a altura da árvore:

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

Qual é a altura da árvore vazia? -1 (convenção).

# Altura

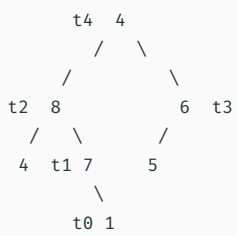
Projete uma função que determine a altura de uma árvore.




```
def altura(t: Arvore) -> int:
    '''
    Devolve a altura da árvore *t*, isto é, o
    comprimeiro do caminho mais longo da raíz
    até um no folha. Devolve -1 se *t* é None.
    >>> altura(None)
    -1
    >>> altura(t1)
    1
    >>> altura(t4)
    3
    '''
    if t is None:
        return ...
    else:
        return t.chave ... \
               altura(t.esq)
               altura(t.dir)
```

# Altura

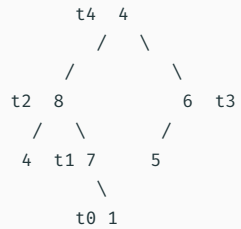
Projete uma função que determine a altura de uma árvore.




```
def altura(t: Arvore) -> int:
    '''
    Devolve a altura da árvore *t*, isto é, o
    comprimeiro do caminho mais longo da raíz
    até um no folha. Devolve -1 se *t* é None.
    >>> altura(None)
    -1
    >>> altura(t1)
    1
    >>> altura(t4)
    3
    '''
    if t is None:
        return -1
    else:
        return 1 + \
               max(altura(t.esq), altura(t.dir))
```

# Nível

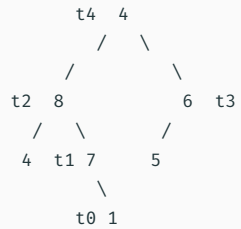
Projete uma função que encontre todos as chaves em um determinado nível de uma árvore.




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

# Nível

Projete uma função que encontre todos as chaves em um determinado nível de uma árvore.




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

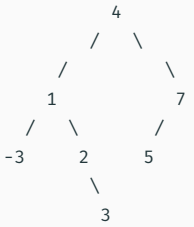
- Todos os elementos da subárvore a esquerda de $t$ são menores que $t.chave$;
- Todos os elementos da subárvore a direita de $t$ são maiores que $t.chave$;
- As subárvores a esquerda e a direta de $t$ são **árvores binárias de busca**.

# Busca em árvore binária de busca

Busca *v* em uma ABB *t*:

- Se *t* é vazia, *v* não está na árvore;
- Se *v* é igual a *t.chave*, *v* está na árvore;
- Senão, se *v* é menor que *t.chave*, continuamos a busca na subárvore a esquerda;
- Senão (*v* é maior que *t.chave*), continuamos a busca na subárvore a direita.

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



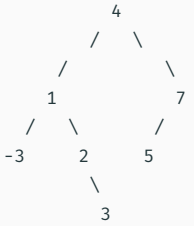

```
def busca(t: Arvore, chave: int) -> bool:
    '''
    Devolve True se *chave* está em *t*,
    False caso contrário.
    >>> busca(None, 10)
    False
    >>> busca(t, 2)
    True
    >>> busca(t, 6)
    False
    '''
    if t is None:
        return ... chave
    else:
        return chave ... \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

# Busca em árvore binária de busca

Busca *v* em uma ABB *t*:

- Se *t* é vazia, *v* não está na árvore;
- Se *v* é igual a *t.chave*, *v* está na árvore;
- Senão, se *v* é menor que *t.chave*, continuamos a busca na subárvore a esquerda;
- Senão (*v* é maior que *t.chave*), continuamos a busca na subárvore a direita.

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



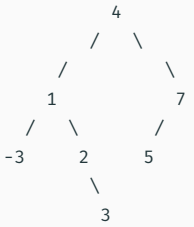

```python
def busca(t: Arvore, chave: int) -> bool:
    r'''
    Devolve True se *chave* está em *t*,
    False caso contrário.
    >>> busca(None, 10)
    False
    >>> busca(t, 2)
    True
    >>> busca(t, 6)
    False
    '''
    if t is None:
        return False
    elif chave == t.chave:
     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

# Busca em árvore binária de busca

Busca *v* em uma ABB *t*:

- Se *t* é vazia, *v* não está na árvore;
- Se *v* é igual a *t.chave*, *v* está na árvore;
- Senão, se *v* é menor que *t.chave*, continuamos a busca na subárvore a esquerda;
- Senão (*v* é maior que *t.chave*), continuamos a busca na subárvore a direita.

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




```python
def busca(t: Arvore, chave: int) -> bool:
    r'''
    Devolve True se *chave* 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 chave == r.chave:
            return True
        elif chave < r.chave:
            r = r.esq
        else:  # chave > r.chave
            r = r.dir
    return False
```

# Complexidade de tempo da busca em ABB

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

```python
def busca(t: Arvore, chave: int) -> bool:
    r = t
    while r is not None:
        if chave == r.chave:
            return True
        elif chave < r.chave:
            r = r.esq
        else:  # chave > r.chave
            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 superior de $h$? $n - 1$. Ocorre quando todos os nós da árvore, exceto as folhas, têm apenas um filho.

Qual é o limite inferior de $h$? $\lg(n)$. Ocorre quando todos os níveis da árvore estão cheios, exceto talvez, o último nível.

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 $n$ chaves aleatórias 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 uma nova chave, 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, chave: int) -> None:
    '''
    Insere *chave* em *t* mantendo as
    propriedades de ABB.
    Requer que *t* seja uma ABB.
    >>> r = None
    >>> insere(r, 10)
    >>> r
    No(esq=None, chave=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`.

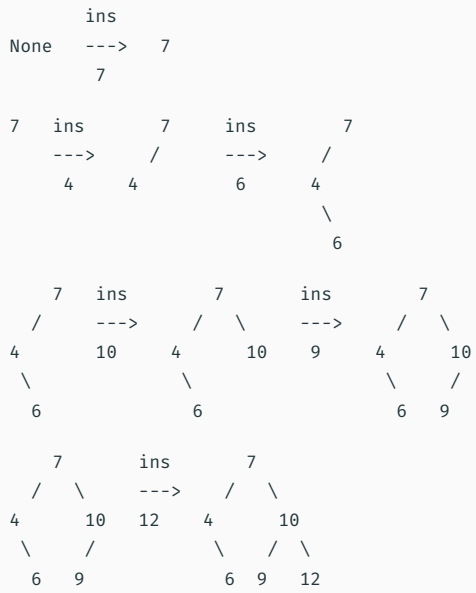
```
def insere(t: Arvore, chave: int) -> No:
    '''
    Devolve a raiz da ABB que é o resultado
    da inserção de *chave* em *t*.
    Se *chave* já está em *t*, devolve *t*.
    Requer que *t* seja uma ABB.

    Exemplo
    >>> r = None
    >>> r = insere(r, 10)
    >>> r
    No(esq=None, chave=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

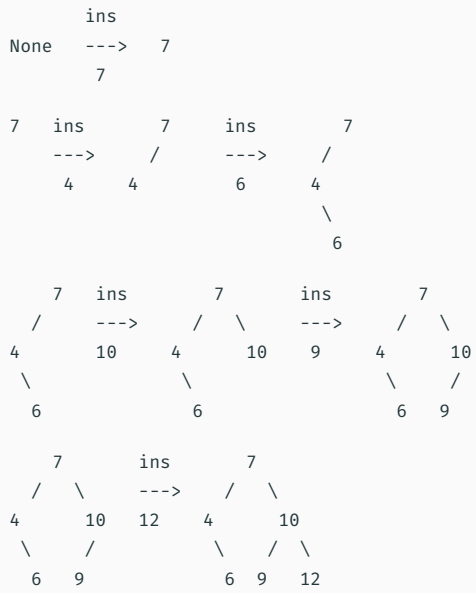



```python
def insere(t: Arvore, chave: int) -> No:
    '''
    Devolve a raiz da ABB que é o resultado
    da inserção de *chave* em *t*.
    Se *chave* já está em *t*, devolve *t*.
    Requer que *t* seja uma ABB.
    '''
    if t is None:
        return ... chave
    else:
        chave ...
      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

# Inserção em árvore binária de busca

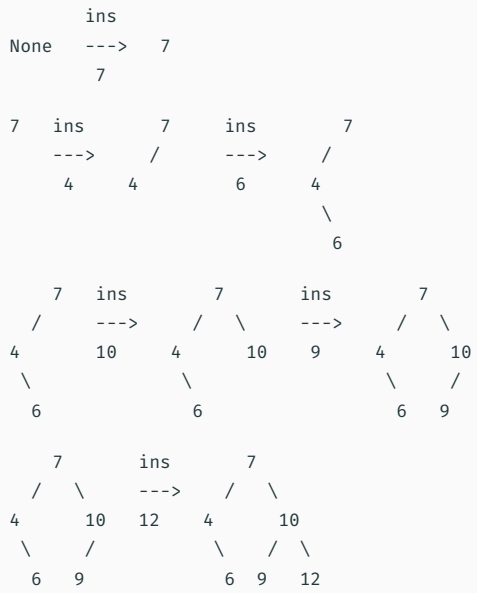

```python
def insere(t: Arvore, chave: int) -> No:
    '''
    Devolve a raiz da ABB que é o resultado
    da inserção de *chave* em *t*.
    Se *chave* já está em *t*, devolve *t*.
    Requer que *t* seja uma ABB.
    '''
    if t is None:
        return ... chave
    else:
        chave ...
      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

# Inserção em árvore binária de busca




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

# Inserção em árvore binária de busca




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

# Inserção em árvore binária de busca

```
def insere(t: Arvore, chave: int) -> No:
    '''
    Devolve a raiz da ABB que é o resultado
    da inserção de *chave* em *t*.
    Se *chave* já está em *t*, devolve *t*.
    Requer que *t* seja uma ABB.
    '''
    if t is None:
        return No(None, chave, None)
    else:
        if chave < t.chave:
            t.esq = insere(t.esq, chave)
        elif chave > t.chave:
            t.dir = insere(t.dir, chave)
        else: # chave == t.chave
            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 uma chave, se estiver presente, de uma árvore binária de busca.

```
def remove(t: Arvore, chave: int) -> Arvore:
    '''
    Devolve a raiz da ABB que é o resultado
    da remoção de *chave* de *t*.
    Se *chave* 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?

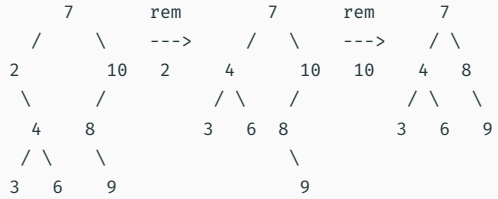
Partindo do modelo!

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

# Remoção em árvore binária de busca

Remoção de folha: retorna `None`.

Remoção de nó sem subárvore a esq ou dir



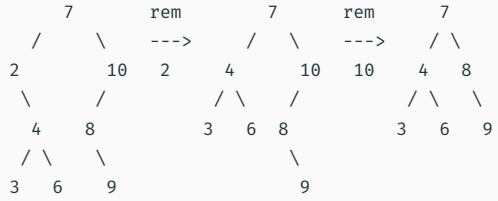

Remoção de nó com subárvore a esq e a dir


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

# Remoção em árvore binária de busca

Remoção de folha: retorna `None`.

Remoção de nó sem subárvore a esq ou dir



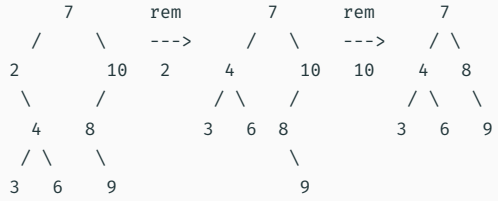

Remoção de nó com subárvore a esq e a dir


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

# Remoção em árvore binária de busca

Remoção de folha: retorna None.

Remoção de nó sem subárvore a esq ou dir



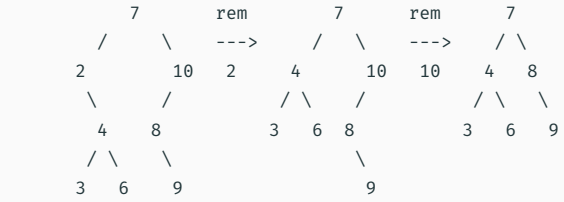

Remoção de nó com subárvore a esq e a dir


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



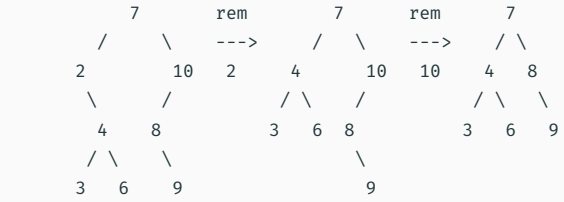

Remoção de nó com subárvore a esq e a dir


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

# Remoção em árvore binária de busca

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

# 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.