UFMS DIGITAL

Estrutura de Dados

Prof. Dr. Gedson Faria

Prof.ª Dr.ª Graziela Santos de Araújo

Prof. Dr. Jonathan de Andrade Silva

UFMS DIGITAL

Módulo 2 - Árvore Binária de Busca

Unidade 1 - Conceitos, algoritmo de inserção e algoritmo de busca

UFMS DIGITAL

Conceitos, algoritmo de inserção e algoritmo de busca

PARTE 1

Revisão

- Nós vimos no módulo anterior o desafio de, por exemplo, buscar um elemento em um estrutura de dados linear do tipo Lista (fila ou pilha);
- Encontramos um estratégia com Tabela de Dispersão. Porém, não é eficiente quando temos prioridades na busca. Em seguida, vimos então Heaps;

Revisão

- Nas Heaps vimos uma estratégia que de certa maneira envolve ordenar os elementos, o que ajuda bastante no processo de busca com prioridades. Porém, é eficiente para buscar apenas os elementos de maior prioridade (Max-Heap ou Min-Heap);
- Nessa ideia de realizar a busca com elementos em ordem, vamos conhecer uma outra estratégia de pesquisa denominada pesquisa binária.

Pesquisa Linear

- Considere um vetor V de valores ordenados V= [1,3,5,7,9,11,13] e desejamos buscar o valor 15, que não existe em V.
  - Na busca convencional perguntamos de i=0 até i=6:
    - V[i] == 15? (7 passos);
    - Custo da busca O(N).

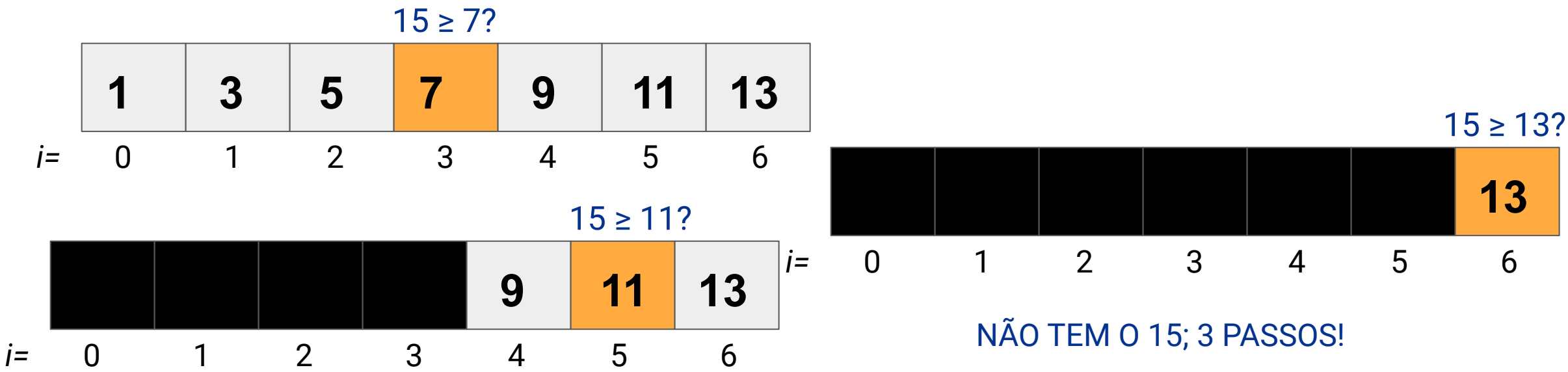
v	1	3	5	7	9	11	13
i=	0	1	2	3	4	5	6

Pesquisa Binária

- Essa estratégia assume que o conjunto de dados deve estar ordenado (crescente ou decrescente);
- Inicia a busca pela posição central do conjunto de dados;
  - Se não for o elemento a ser buscado, divide o conjunto em duas metades, continuando a busca na metade onde o elemento pode estar;
  - Esse processo é repetido até encontrar o elemento desejado ou chegar ao fim do conjunto de dados.

Pesquisa Binária

- Considere um vetor V de valores ordenados V= [1,3,5,7,9,11,13] e desejamos buscar o valor 15, que não existe em V.

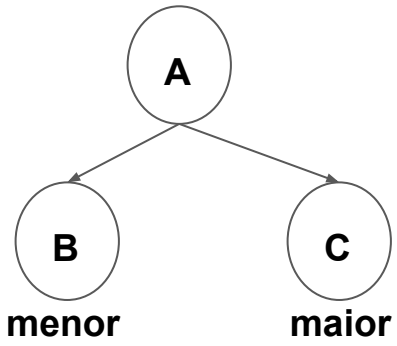


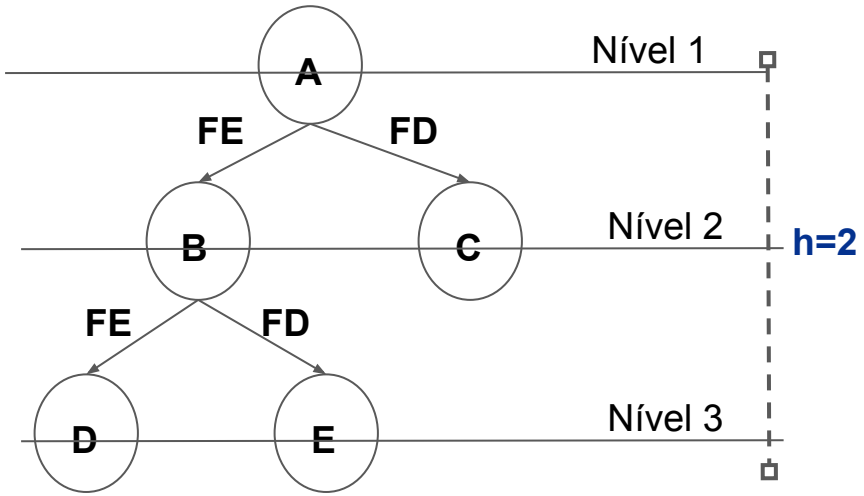
Pesquisa Binária

- Problema: 
  - Manter o vetor ordenado!
  - Se aplicar a ordenação em cada operação de inserção e remoção o custo fica alto!
    - Custo da inserção/remoção + custo da ordenação!
- Solução: 
  - Árvore de Busca Binária!

Árvore de Busca Binária

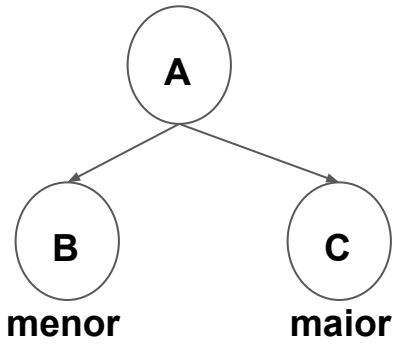
- Como vimos no módulo anterior sobre Heap, na árvore binária temos:
  - Cada nó possui até 2 filhos (FE e FD);
  - A altura (h) da árvore é igual ao número de níveis -1;
- Propriedade:
  - B < A < C
  - B é o Menor
  - C é o Maior

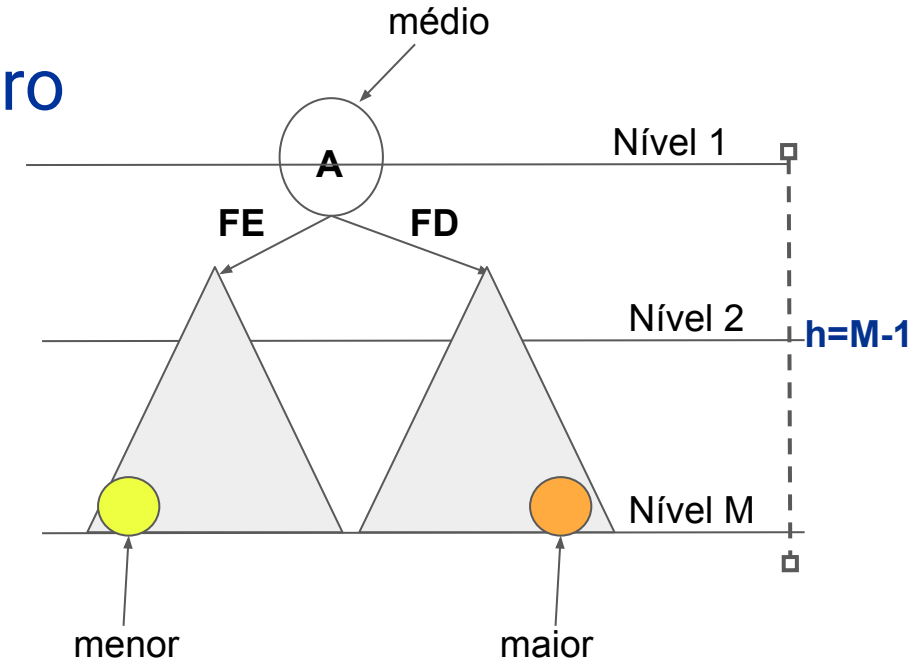




Árvore de Busca Binária

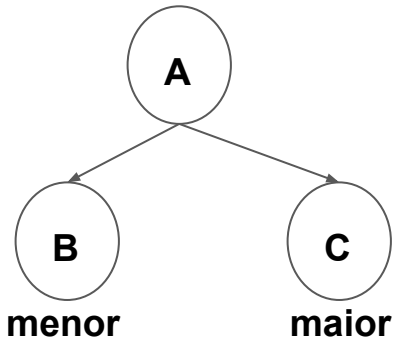
- Como vimos no módulo anterior sobre Heap, na árvore binária temos:
  - Cada nó possui até 2 filhos (FE e FD);
  - A altura (h) da árvore é igual ao número de níveis - 1;
- Propriedade:
  - B < A < C
  - B é o Menor
  - C é o Maior

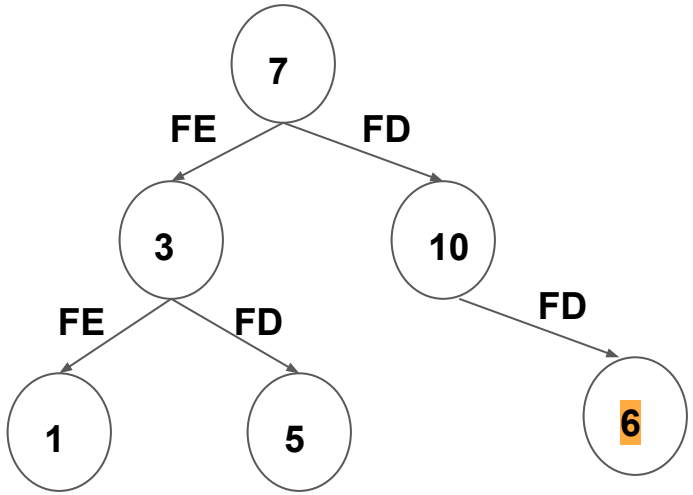




Árvore de Busca Binária

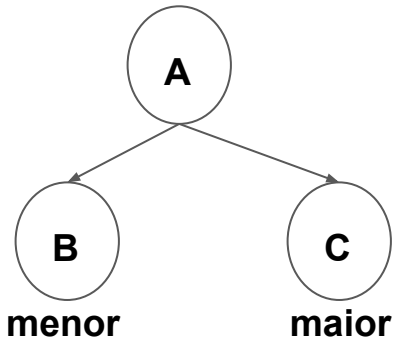
- A árvore ao lado é binária?
  - Não!
- Propriedade:
  - B < A < C
  - B é o Menor
  - C é o Maior

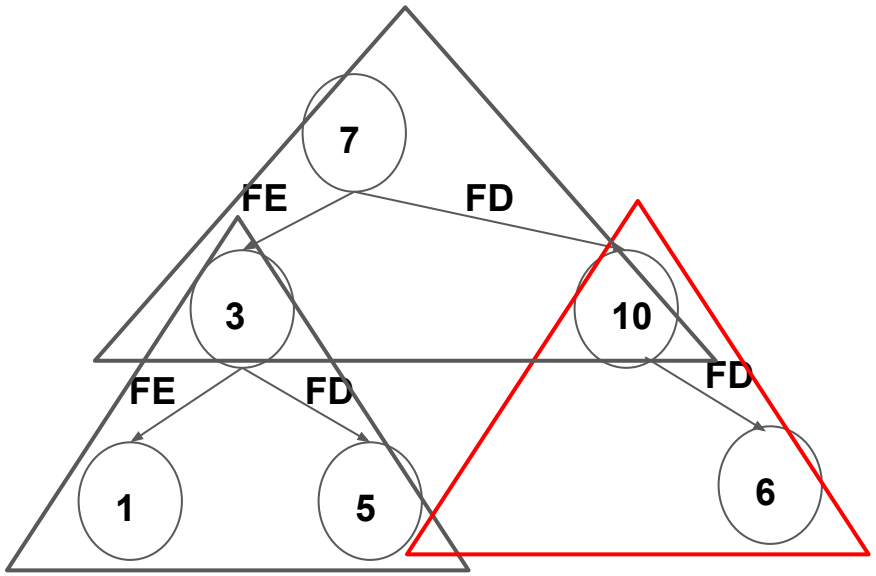




Árvore de Busca Binária

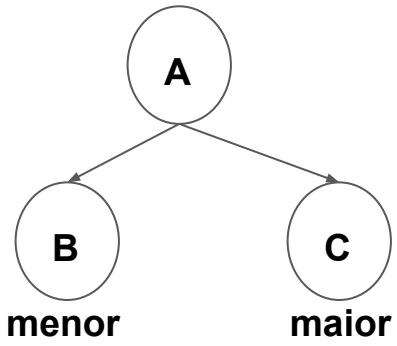
- A árvore ao lado é binária?
  - Não! Por quê?
    - 6 Viola a propriedade.
- Propriedade:
  - B < A < C
  - B é o Menor
  - C é o Maior

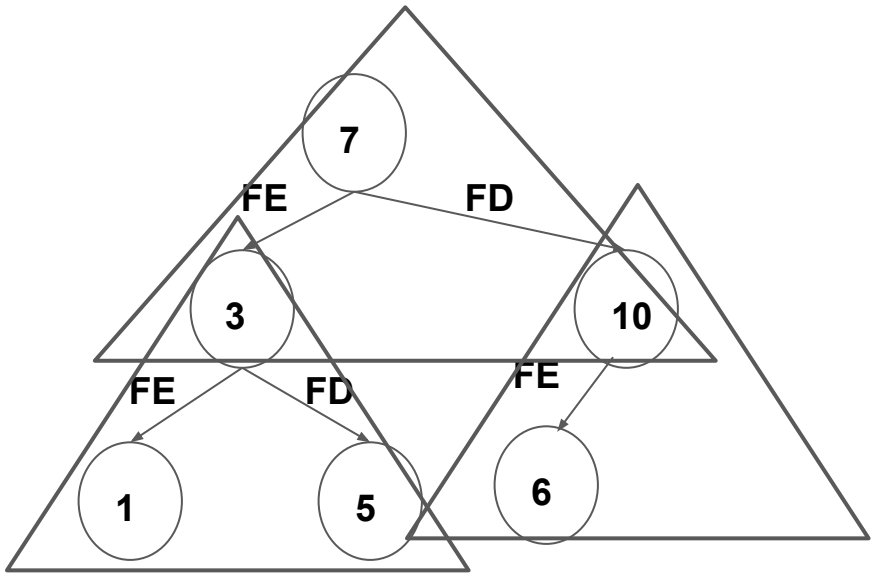




Árvore de Busca Binária

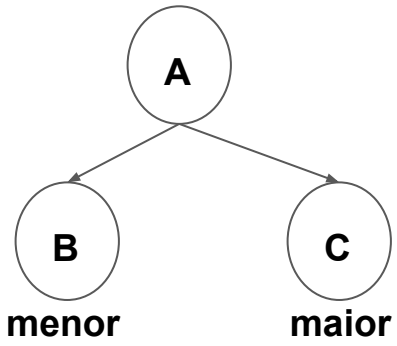
- A árvore ao lado é binária?
  - Agora sim!
- Propriedade:
  - B < A < C
  - B é o Menor
  - C é o Maior

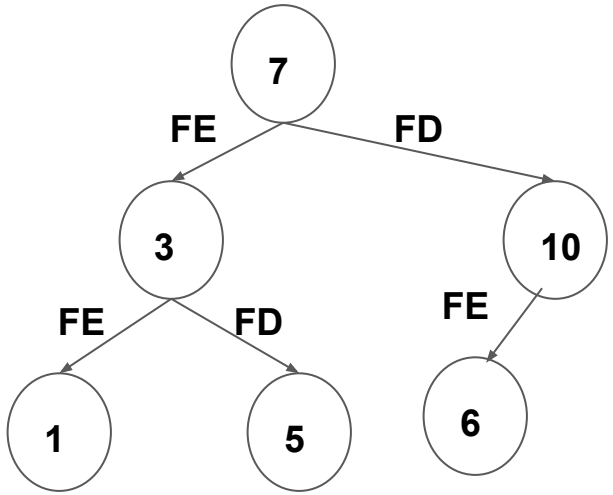




Árvore de Busca Binária

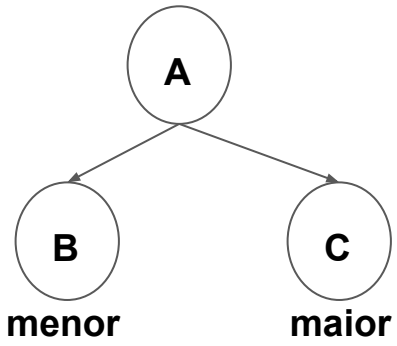
- A árvore ao lado é binária?
  - Agora sim!
- Propriedade:
  - B < A < C
  - B é o Menor
  - C é o Maior

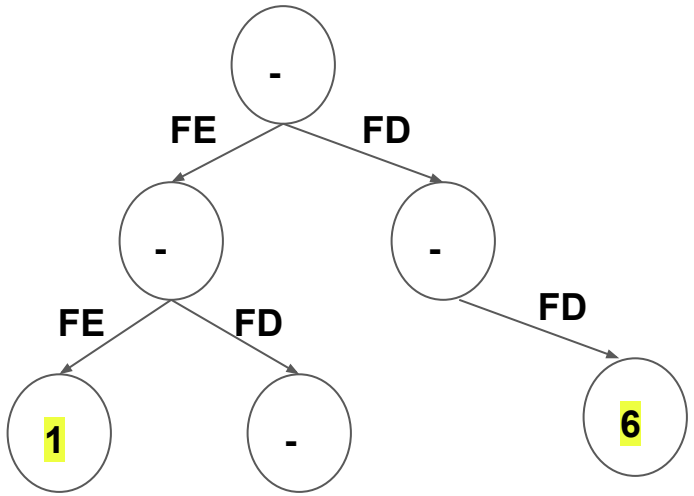




Árvore de Busca Binária

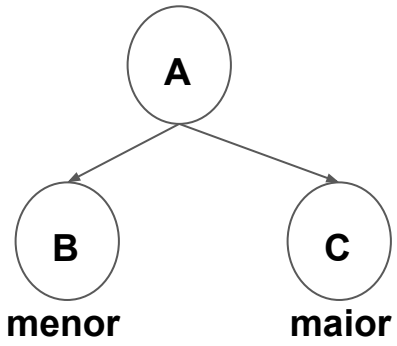
- Como podemos organizar os valores de 1 a 6 nessa árvore?
  - Onde colocar o menor e o maior valor?
- Propriedade:
  - B < A < C
  - B é o Menor
  - C é o Maior

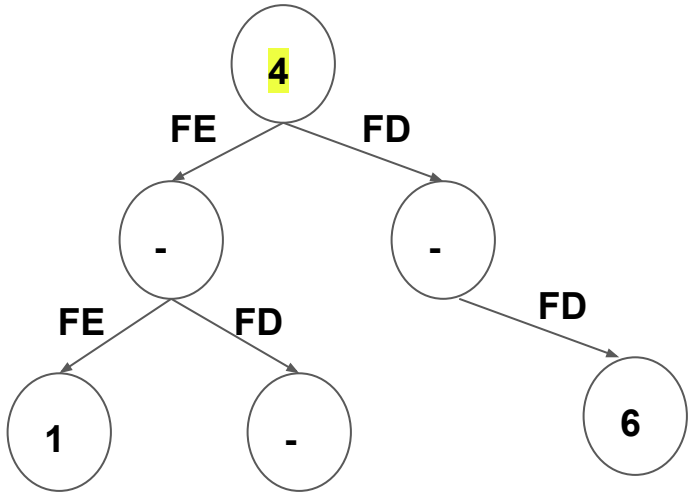




Árvore de Busca Binária

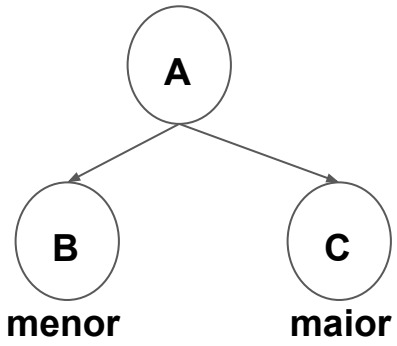
- Como podemos organizar os valores de 1 a 6 nessa árvore?
  - Qual valor vai na raiz?
- Propriedade:
  - B < A < C
  - B é o Menor
  - C é o Maior

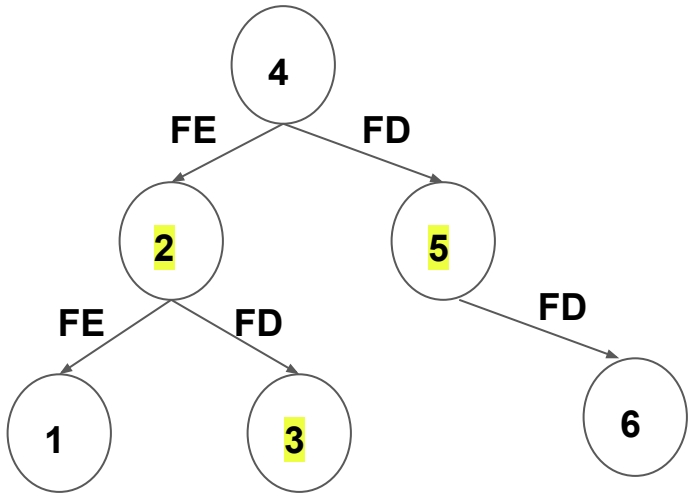




Árvore de Busca Binária

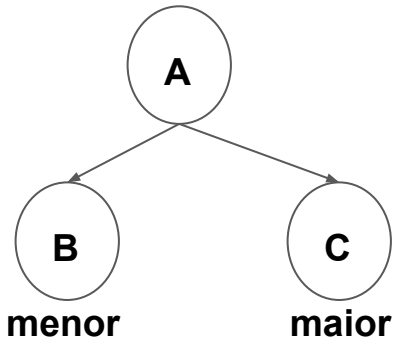
- Como podemos organizar os valores de 1 a 6 nessa árvore?
  - E os demais 2, 3 e 5?
- Propriedade:
  - B < A < C
  - B é o Menor
  - C é o Maior

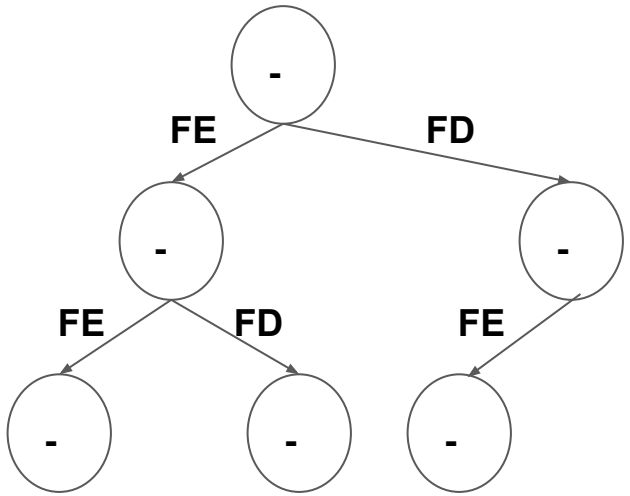




Árvore de Busca Binária

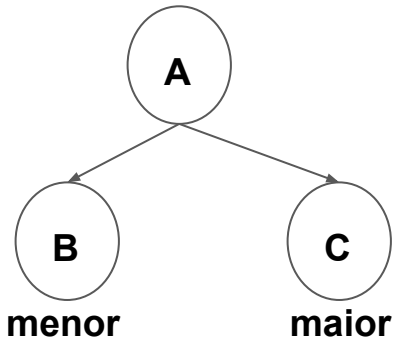
- Como podemos organizar os valores de 1 a 6 nessa árvore?
  - E se a árvore fosse assim ->
- Propriedade:
  - B < A < C
  - B é o Menor
  - C é o Maior

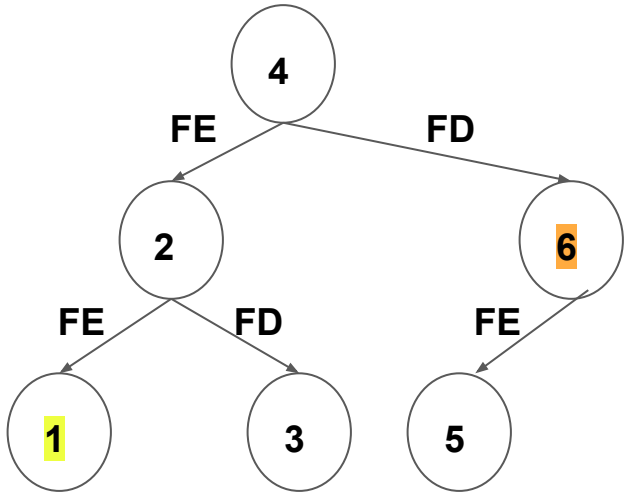




Árvore de Busca Binária

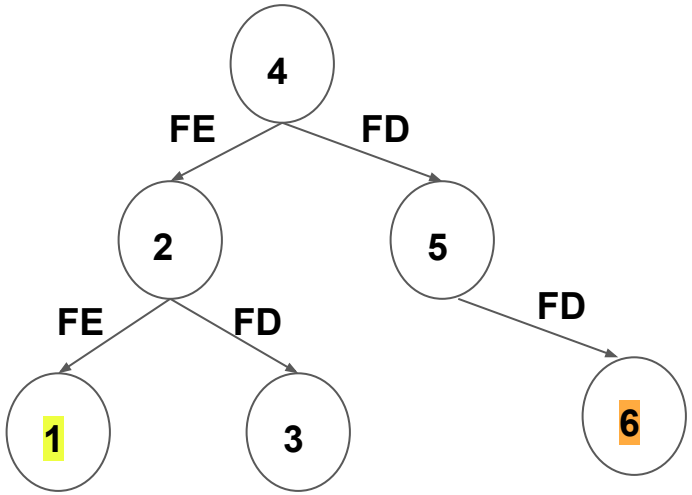
- Como podemos organizar os valores de 1 a 6 nessa árvore?
  - Teríamos ->
- Propriedade:
  - B < A < C
  - B é o Menor
  - C é o Maior

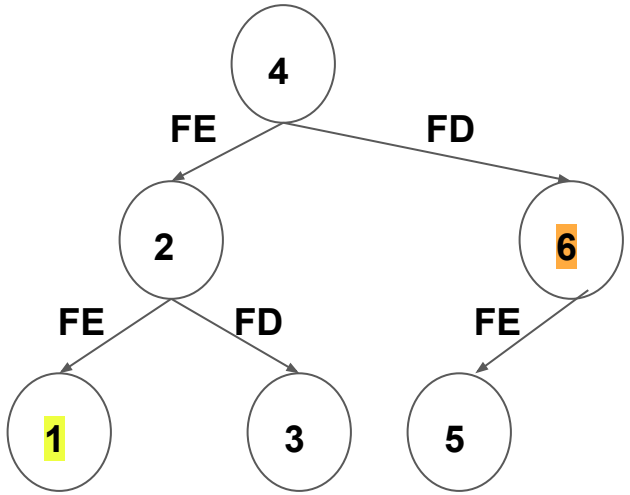




Árvore de Busca Binária

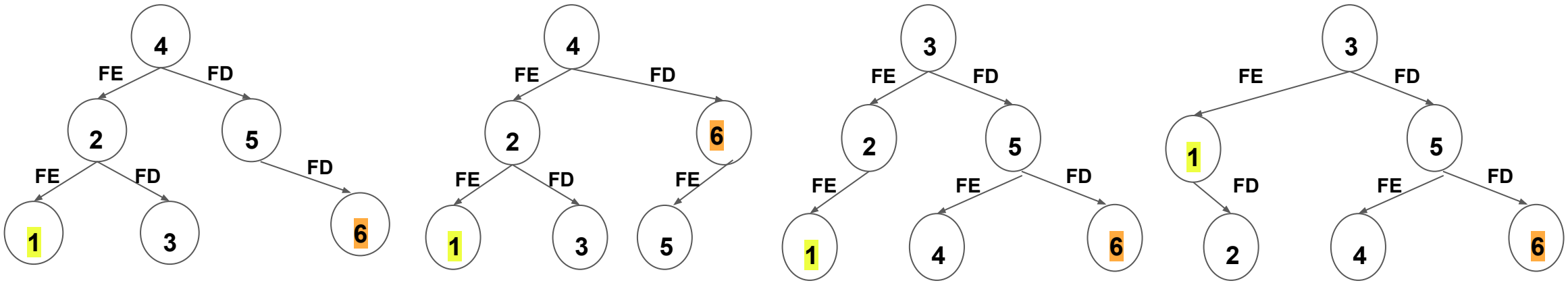
- Comparando as duas árvores binárias, temos:





Árvore de Busca Binária

- Poderíamos ter outras configurações de árvore binária para os mesmos valores;
  - Depende da ordem de inserção dos elementos.



Árvore de Busca Binária

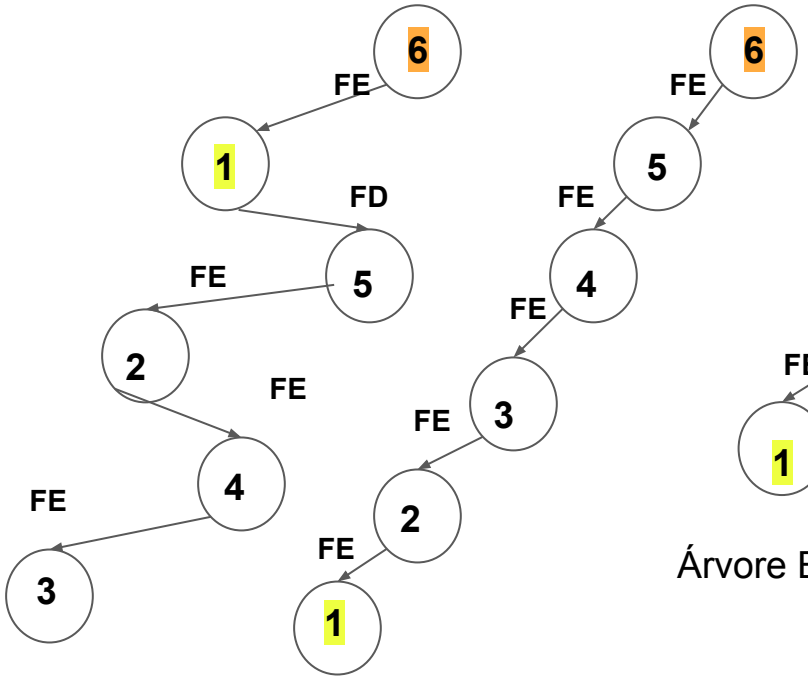
- Tipos de árvores binárias:
  - **Degenerada**: cada nó possui exatamente 1 filho (n° de níveis = n° de nós);
  - **Quase-Completa**: cada nível, exceto o último está completamente preenchido;

Árvore de Busca Binária

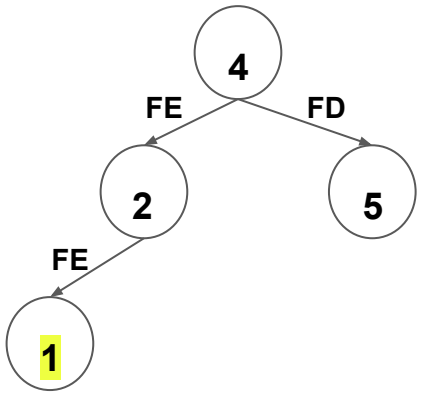
- Tipos de árvores binárias:
  - **Completa**: todos os níveis estão completamente preenchidos;
  - **Cheia** (completa): todos os nós, exceto os do último nível possuem exatamente 2 subárvores.
  - **Estritamente** Binária: todo nó tem 0 ou 2 filhos.

Árvore de Busca Binária

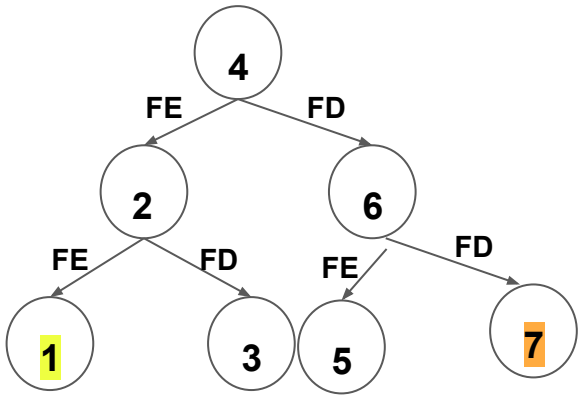
- Tipos de árvores binárias:



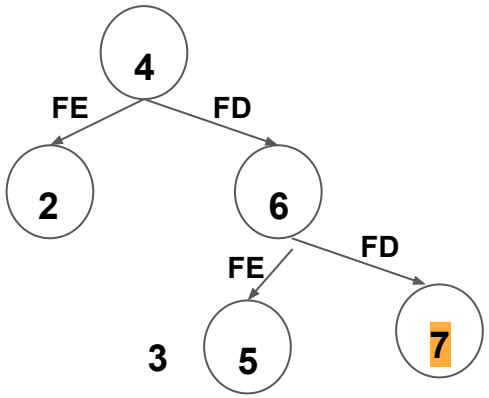
Árvore Binária Degenerada



Árvore Binária Quase-Completa



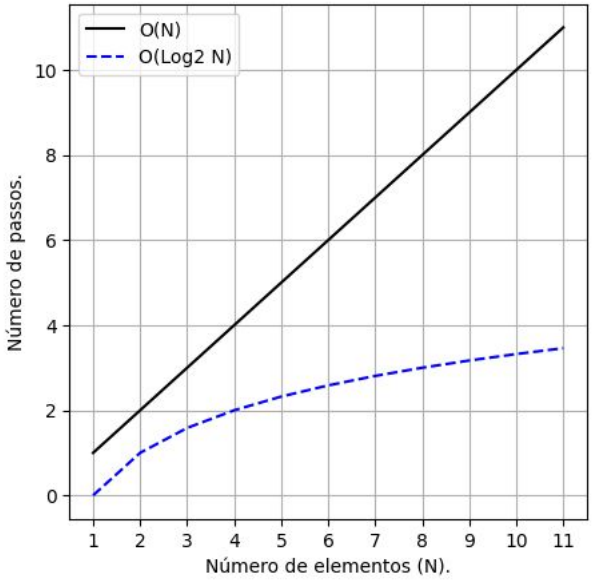
Árvore Binária Completa/Cheia



Árvore Estritamente Binária

Árvore de Busca Binária

- Nessa estrutura de dados gostaríamos de evitar obter árvores binárias degeneradas.
- Caso tenhamos árvores quase-completas ou completas, teríamos um ótimo desempenho na inserção, remoção e busca.
  - De O(N) para O($\log_2 N$)!



Árvore Binária (Inserção)

- Em geral, consiste na operação de incluir elementos nas sub-árvores esquerda ou direita da raiz;
- Pode modificar a altura da árvore dependendo de onde for o local apropriado de inserção;
  - Pode resultar nos tipos de árvores comentados anteriormente, por exemplo, árvores degeneradas.
- Deve garantir a propriedade da árvore de busca binária.

Árvore Binária (Inserção)

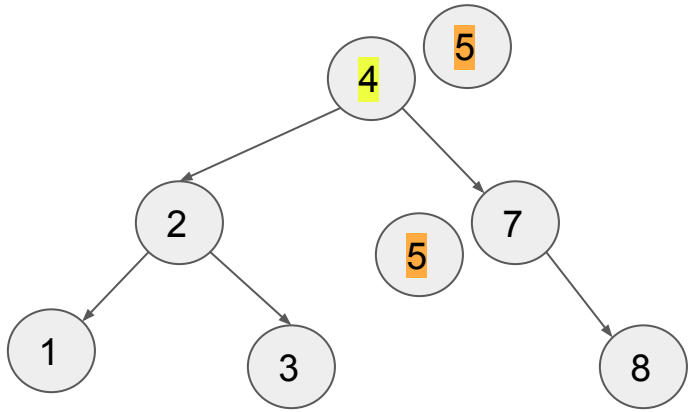
- Todo o processo de caminhada na árvore inicia-se na raiz;
- Precisamos encontrar em qual local na árvore podemos incluir o novo nó;
- Esse processo de inclusão sempre produzirá um nó folha;
- Esse novo nó se tornará um filho esquerdo (FE) ou filho direito (FD) do seu ancestral.
  - Temos que encontrar o seu ancestral.

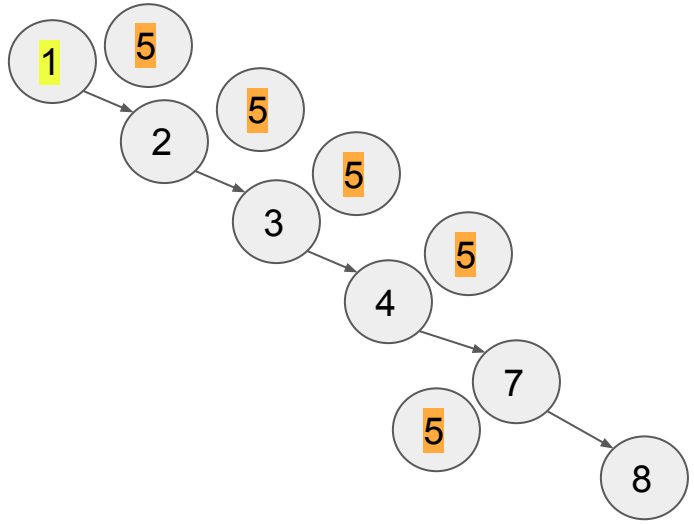
Árvore Binária (Inserção)

- Vamos visualizar o processo de inserção no VISUALGO;
  - Vamos criar uma árvore vazia (“Create” -> “Empty”)
  - Aplicar a função de inserção para os valores (“Insert(v)”):
    - A=[1,2,3,4,7,8]
    - B=[4,2,7,1,3,8]
  - Qual dessas árvores é uma árvore binária degenerada (A ou B)? A. Observe a organização dos valores.

Árvore Binária (Inserção)

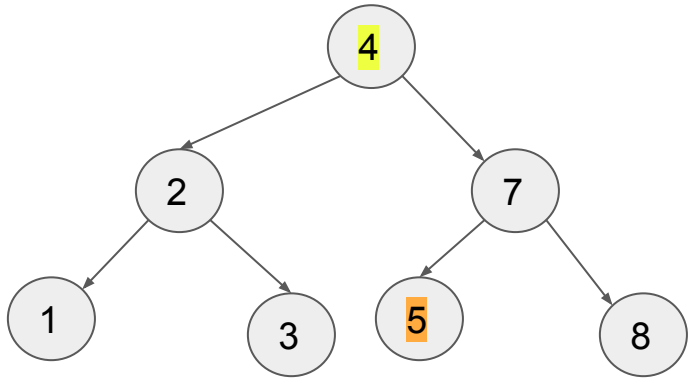
- Inserir o valor 5 nas árvores abaixo:

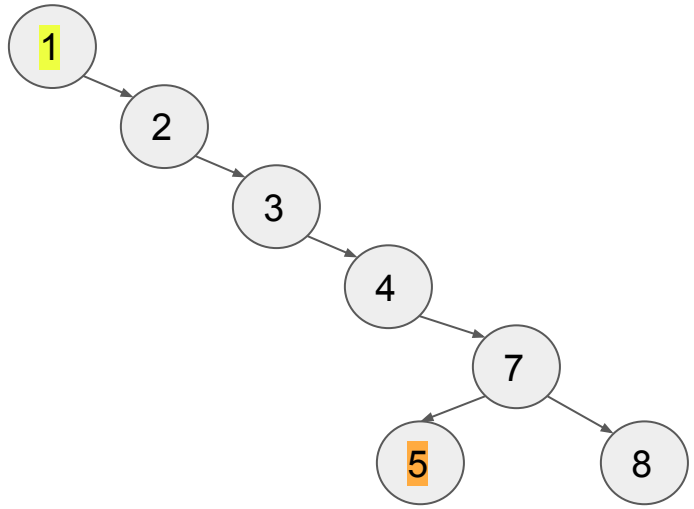




Árvore Binária (Inserção)

- Inserir o valor 5 nas árvores abaixo:





Árvore Binária (Inserção)

- Precisamos saber quando inserir na raiz ou buscar nas subárvores da esquerda ou direita o local de inserção.
  - Inserir na raiz é quando a árvore está vazia;
    - Qual a configuração de árvore vazia? Raiz sem nó.
    - Fazer a raiz ser esse novo nó.
  - Se a árvore tem raiz, então devemos buscar o local de inserção...

Árvore Binária (Inserção)

- Se a árvore tem raiz então devemos buscar o local de inserção.
  - Percorrer/buscar desde a raiz e perguntar se o novo nó deve estar na subárvore da esquerda ou da direita;
    - novo < nó atual: ir para FE;
    - novo > nó atual: ir para FD;
    - novo = nó atual: não inserir.
      - Podemos gerar uma árvore degenerada.

Árvore Binária (Busca)

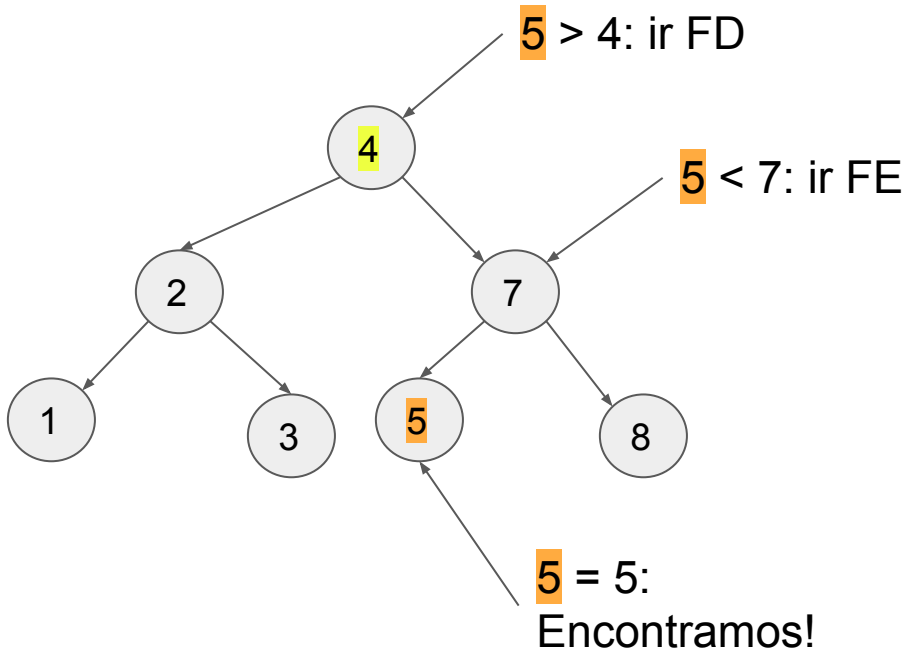
- Caminhar na árvore desde a raiz para encontrar um elemento ou encontrar uma posição de inserção para um novo elemento;
  - Na busca por um elemento vamos realizar a busca para encontrar um valor;
  - Na busca por uma posição de inserção vamos realizar a busca para encontrar um “ramo” sem filho (FE=vazio ou FD=vazio).

Árvore Binária (Busca)

- Vamos assumir que a estrutura de dados que representa o nosso nó tem 3 campos:
  - chave: contendo o valor;
  - FE: contendo referência ao filho esquerdo;
  - FD: contendo referência ao filho direito.

Árvore Binária (Busca)

- Buscar o valor 5
  - no = buscar(raiz,5)



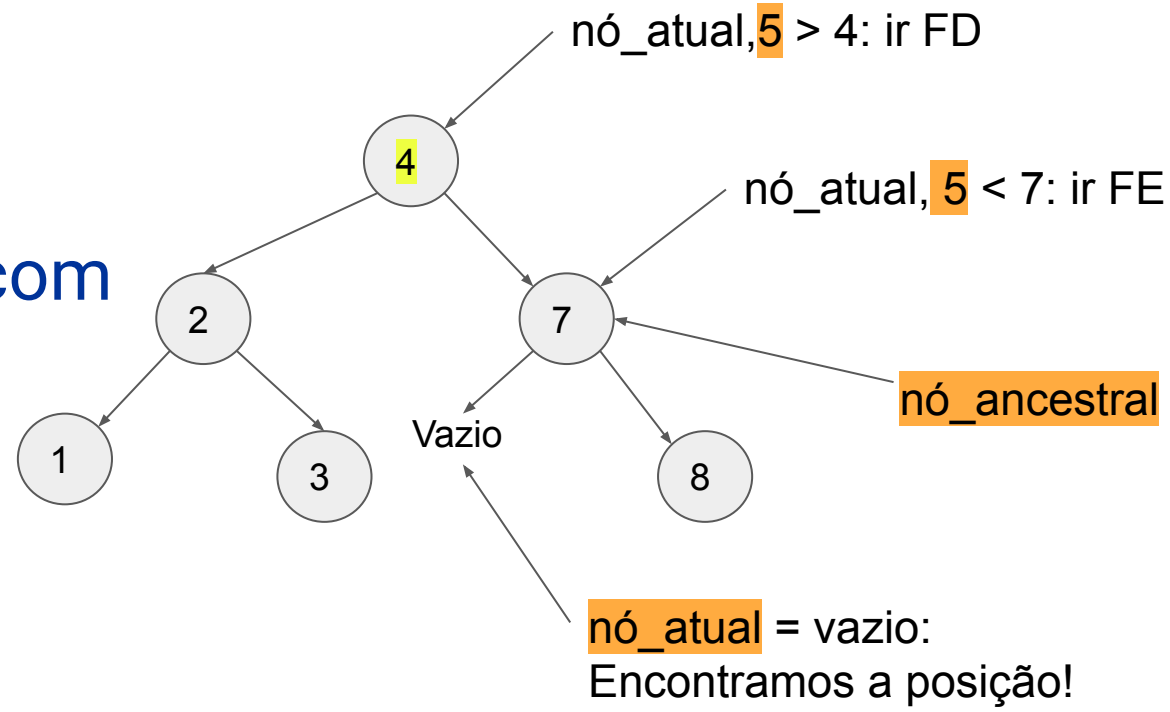
- Buscar(raiz,valor):
  - no_atual = raiz
  - **Enquanto** no_atual ≠ vazio **e** no_atual.chave ≠ valor **faça**
    - **Se** valor < no_atual.chave **Então**
      - no_atual = no_atual.FE
    - **Senão**
      - no_atual = no_atual.FD
  - **retorna** no_atual

Árvore Binária (Inserção)

- Agora que temos o algoritmo da busca, podemos criar o algoritmo de inserção.
  - Porém, temos que ajustar a busca para obter também o ancestral;
  - Lembrando: Se a árvore tem raiz, então devemos buscar o local de inserção.

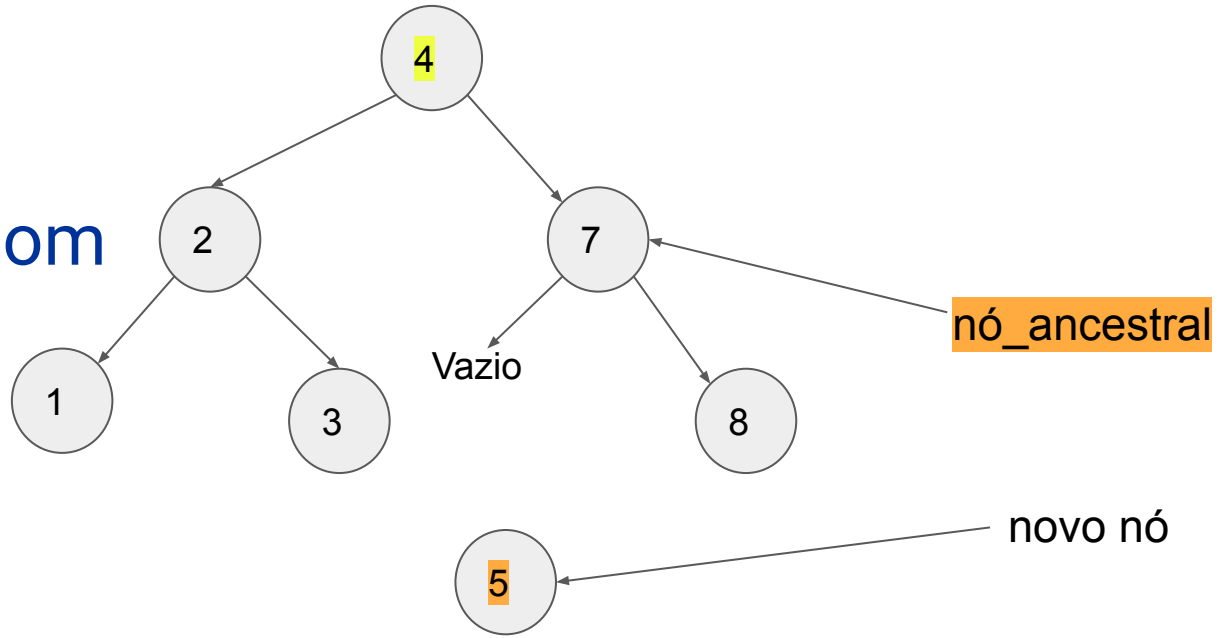
Árvore Binária (Inserção)

- Inserir o valor 5
  - Buscar o ancestral
    - anc = buscar(raiz,5)
  - Conectar esse ancestral com o novo nó
    - anc.FE = novo; ou
    - anc.FD = novo



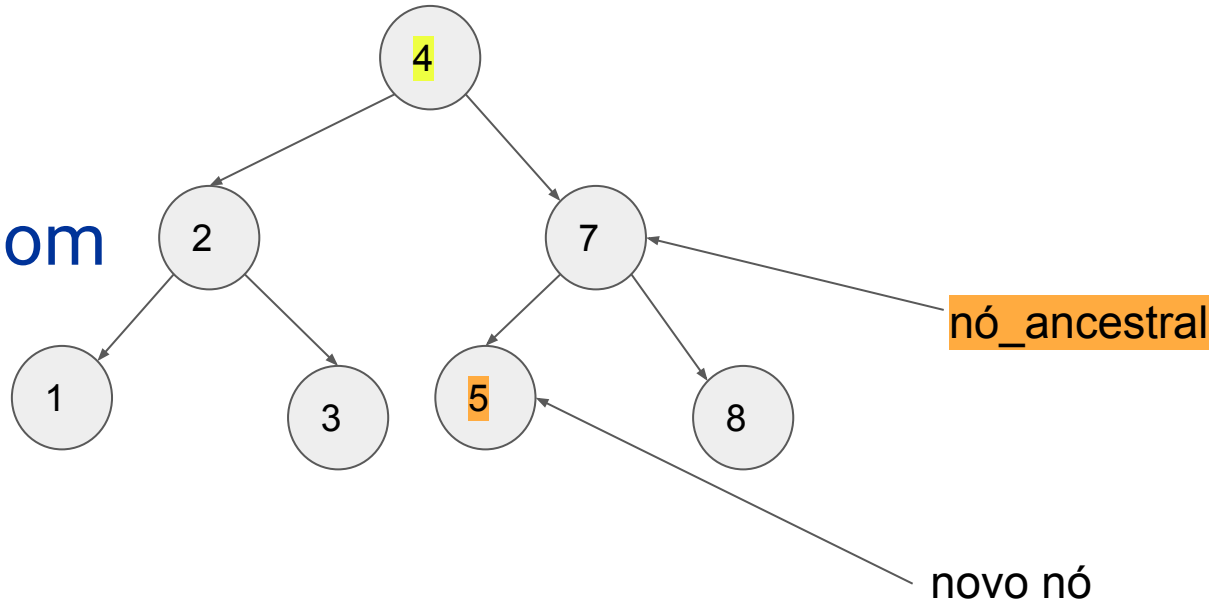
Árvore Binária (Inserção)

- Inserir o valor 5
  - Buscar o ancestral
    - anc = buscar (raiz,5)
  - Conectar esse ancestral com o novo nó
    - anc.FE = novo; ou
    - anc.FD = novo



Árvore Binária (Inserção)

- Inserir o valor 5
  - Buscar o ancestral
    - anc = buscar(raiz,5)
  - Conectar esse ancestral com o novo nó
    - anc.FE = novo; ou
    - anc.FD = novo

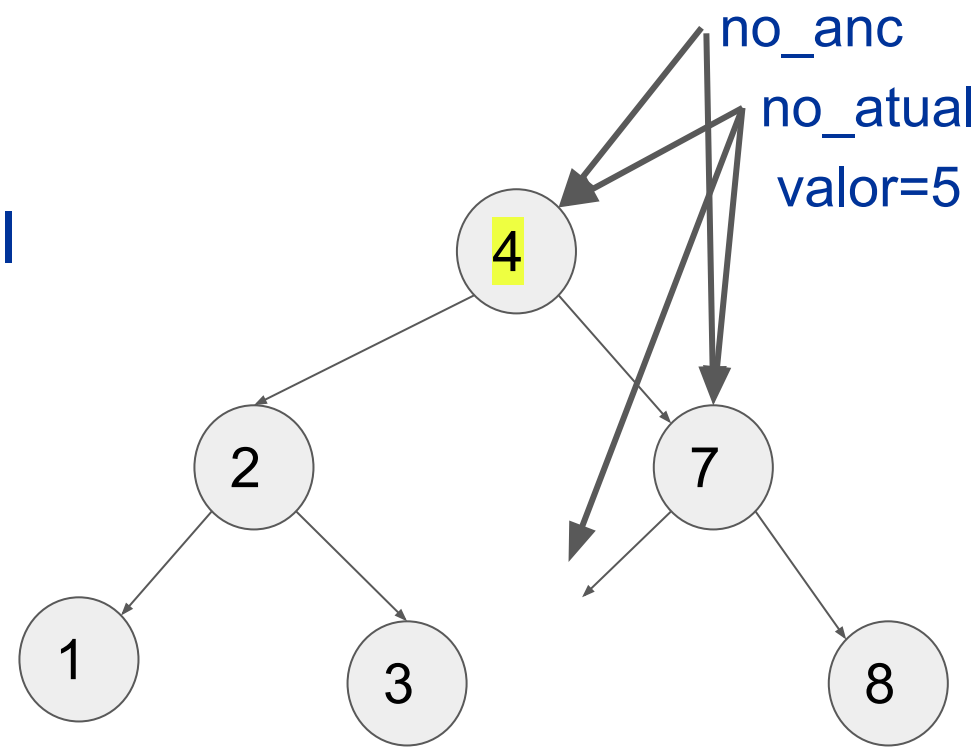


Árvore Binária (Inserção)

- Agora que ajustamos a busca para ter acesso ao nó ancestral, podemos definir o algoritmo de inserção.

Árvore Binária (Inserção)

- Buscar (raiz,valor):
  - **no_anc**=vazio
  - no_atual = raiz
  - **Enquanto** no_atual ≠ vazio **e** no_atual ≠ valor **faça**
    - **no_anc** = no_atual
    - **Se** valor < no_atual.chave **Então**
      - no_atual = no_atual.FE
    - **Senão**
      - no_atual = no_atual.FD
  - **retornar** no_atual, **no_anc**



Árvore Binária (Inserção)

Após a adaptação do algoritmo de busca, temos:

- Buscar o ancestral:
  - no, anc = buscar(raiz,valor)
- Conectar esse ancestral com o novo nó:
  - **Se** valor < anc.valor **Então**
    - anc.FE = novo
  - **Senão Se** valor > anc.valor **Então**
    - anc.FD = novo

Árvore Binária (Inserção)

inserir(valor):

- novo = criaNo(valor)
- no, anc = buscar(raiz,valor)
- **Se** no = vazio **Então**
  - **Se** valor < anc.valor **Então**
    - anc.FE = novo
  - **Senão Se** valor > anc.valor **Então**
    - anc.FD = novo

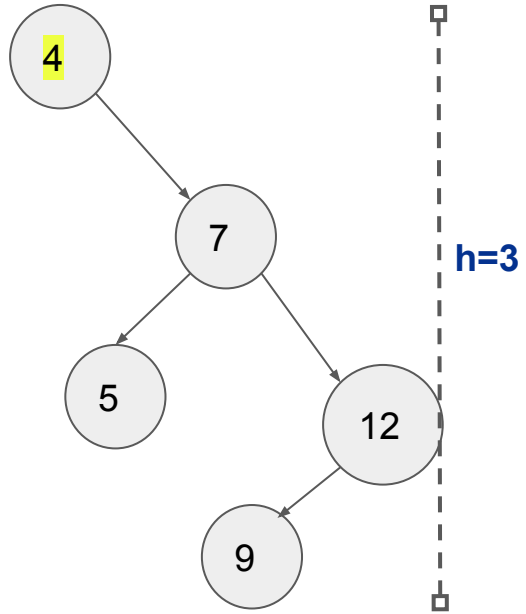
- Como devem ficar os campos do novo nó após “CriaNo”?
  - campo chave = valor;
  - filhos FE e FD = vazio.

Árvore Binária (Análise)

- Conforme observamos, a inserção depende da busca de um local para inserir um novo nó.
- O custo da inserção depende do número de passos dessa busca:
  - Caminhar em toda a altura da árvore;
  - Complexidade de custo no pior caso O(N), se a árvore for degenerada.
    - Se a árvore for completa ou quase-completa o custo fica em $O(\log_2 N)$.

Árvore Binária (Análise)

- Qual o melhor local de inserção de um novo nó?
  - Filho esquerdo de 4.
- Qual o pior local de inserção de um novo nó?
  - Filho (FE ou FD) do nó 9.
- Qual a altura da árvore?
  - altura (h=3).



UFMS DIGITAL

Implementação em Python

PARTE II

Árvore Binária (Estrutura de Dados)

- Podemos implementar a nossa árvore com 2 tipos de estrutura de dados:
  - Vetor/Array, como fizemos na Heap.
    - Acesso direto aos nós da árvore;
    - Porém, o tamanho da árvore fica fixo.
  - Utilizando Listas Ligadas.
    - Não temos acesso direto aos nós;
    - Porém, o tamanho da árvore é dinâmico.

Árvore Binária (Estrutura de Dados)

- Vamos considerar então as Listas Ligadas.
  - Queremos ter flexibilidade para inserção e remoção;
- Vamos considerar que cada nó tenha:
  - campo *chave* = valor;
  - campo *FE* = referência ao filho esquerdo;
  - campo *FD* = referência ao filho direito.

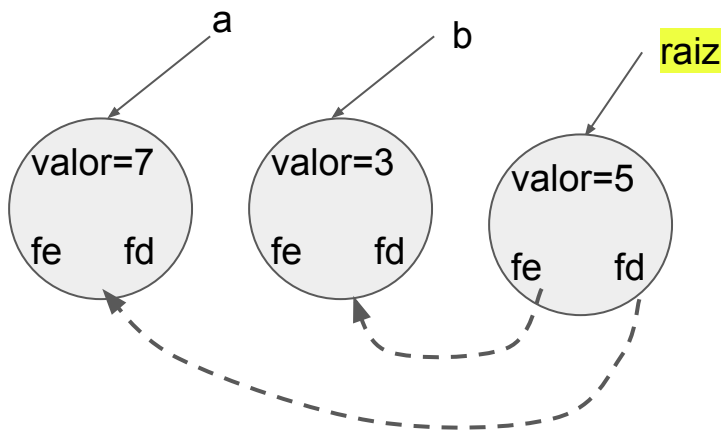
Árvore Binária (Estrutura de Dados)

- Vamos considerar também que a estrutura de dados Árvore Binária de Busca(ABB), pode ter:
  - campo *raiz* = referência ao primeiro nó da árvore;
  - campo *h* = indicando a altura da árvore.
  - funções de manutenção:
    - Inserção, remoção, busca, altura,...etc.

Árvore Binária (Estrutura de Dados)

- Como conectar os nós 7 e 5 na raiz?
  - raiz.fe = b;
  - raiz.fd = a.

```
1  class no:
2      def __init__(self,valor):
3          self.chave=valor #Campo chave
4          self.fe=None #Campo FE inicialmente vazio(None)
5          self.fd=None #Campo FD inicialmente vazio(None)
6
7  #Criando 3 nós raiz com valor 5, a com valor 7 e b com valor 3
8  a = no(7)
9  raiz = no(5)
10 b = no(3)
11 #Como conectar os nós a e b na raiz?
```

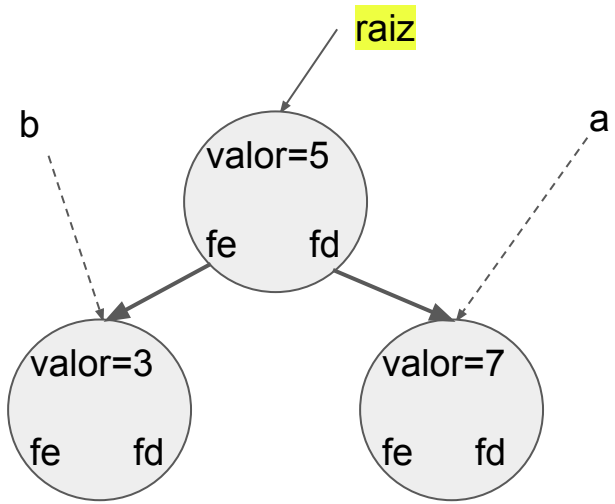


Árvore Binária (Estrutura de Dados)

- Como conectar os nós 7 e 5 na raiz?
  - raiz.fe = b;
  - raiz.fd = a.

```
1  class no:
2      def __init__(self,valor):
3          self.chave=valor #Campo chave
4          self.fe=None #Campo FE inicialmente vazio(None)
5          self.fd=None #Campo FD inicialmente vazio(None)
6
7  #Criando 3 nós raiz com valor 5, a com valor 7 e b com valor 3
8  a = no(7)
9  raiz = no(5)
10 b = no(3)
11 #Como conectar os nós a e b na raiz?
```

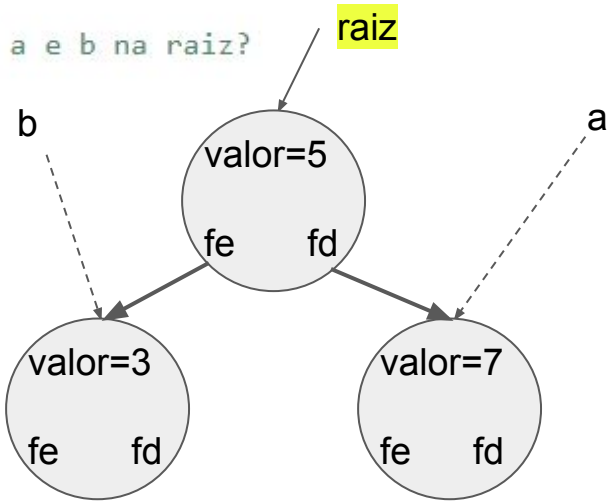
Código



Árvore Binária (Estrutura de Dados)

- E se fizermos uma função chamada conectar para fazer essa ligação, como seria?
  - lembre que temos que saber quais nós devem ir para esquerda ou direita da raiz.

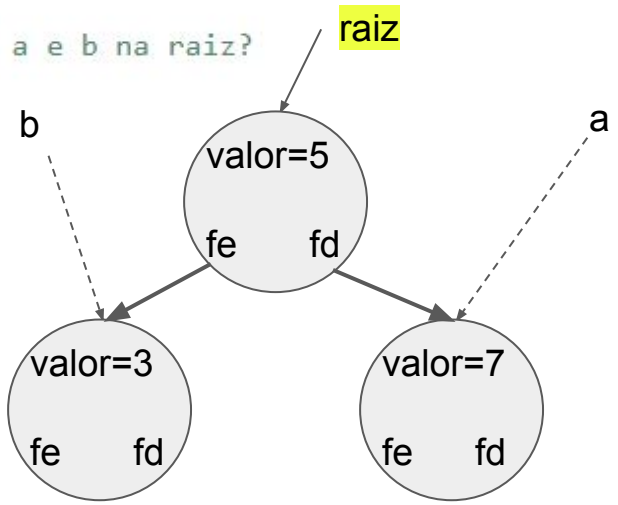
```python
class no:
    def __init__(self,valor):
        self.chave=valor #Campo chave
        self.fe=None #Campo FE inicialmente vazio(None)
        self.fd=None #Campo FD inicialmente vazio(None)

#função para conectar a raiz com um nó
def conectar(raiz, no):

#Criando 3 nós raiz com valor 5, a com valor 7 e b com valor 3
b = no(3)
raiz = no(5)
a = no(7)
#Como conectar os nós a e b na raiz?
conectar(raiz,a)
conectar(raiz,b)
```



Árvore Binária (Estrutura de Dados)

- Se o nó tiver valor menor que a raiz, devemos conectar o nó com fe da raiz;
- Se o nó tiver valor maior que a raiz, devemos conectar o nó com fd da raiz;

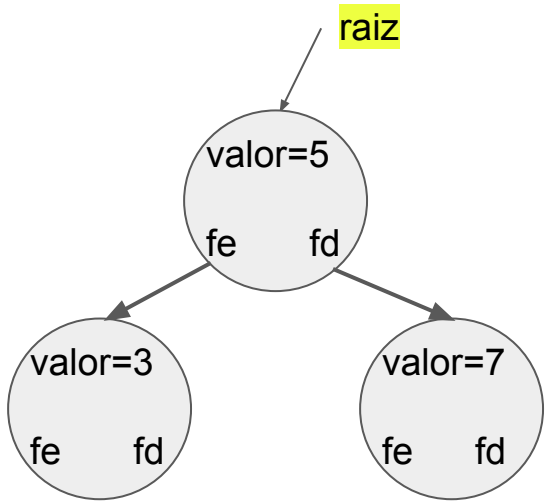
```python
class no:
    def __init__(self,valor):
        self.chave=valor #Campo chave
        self.fe=None #Campo FE inicialmente vazio(None)
        self.fd=None #Campo FD inicialmente vazio(None)

#função para conectar a raiz com um nó
def conectar(raiz, no):

#Criando 3 nós raiz com valor 5, a com valor 7 e b com valor 3
b = no(3)
raiz = no(5)
a = no(7)
#Como conectar os nós a e b na raiz?
conectar(raiz,a)
conectar(raiz,b)
```



Árvore Binária (Estrutura de Dados)

- Se o nó tiver valor menor que a raiz, devemos conectar o nó com fe da raiz;
- Se o nó tiver valor maior que a raiz, devemos conectar o nó com fd da raiz;

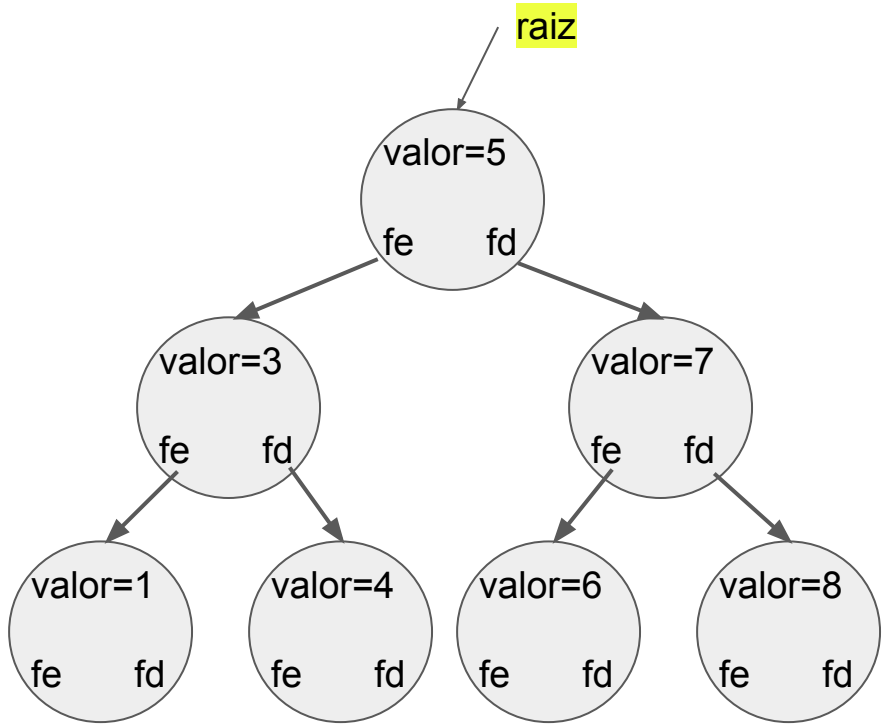
```
7  #função para conectar a raiz com um nó
8  def conectar(raiz, no):
9   if no.chave < raiz.chave:
10       raiz.fe=no
11  elif no.chave > raiz.chave:
12       raiz.fd=no
13
14 #Criando 3 nós raiz com valor 5, a com valor 7 e b com valor 3
15 raiz = no(5)
16 #Como conectar os nós a e b na raiz?
17 conectar(raiz,no(3))
18 conectar(raiz,no(7))
```



Código

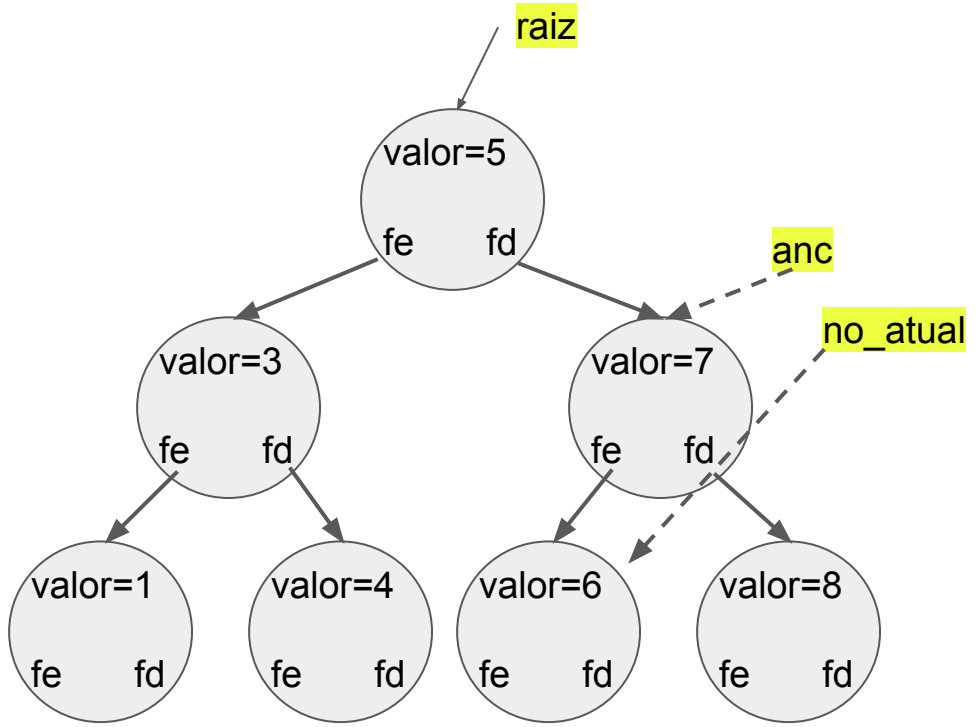
Árvore Binária (Busca)

- Vamos agora começar pela nossa função de busca.
  - Percorrer desde a raiz e encontrar um valor ou local de inserção.



Árvore Binária (Busca)

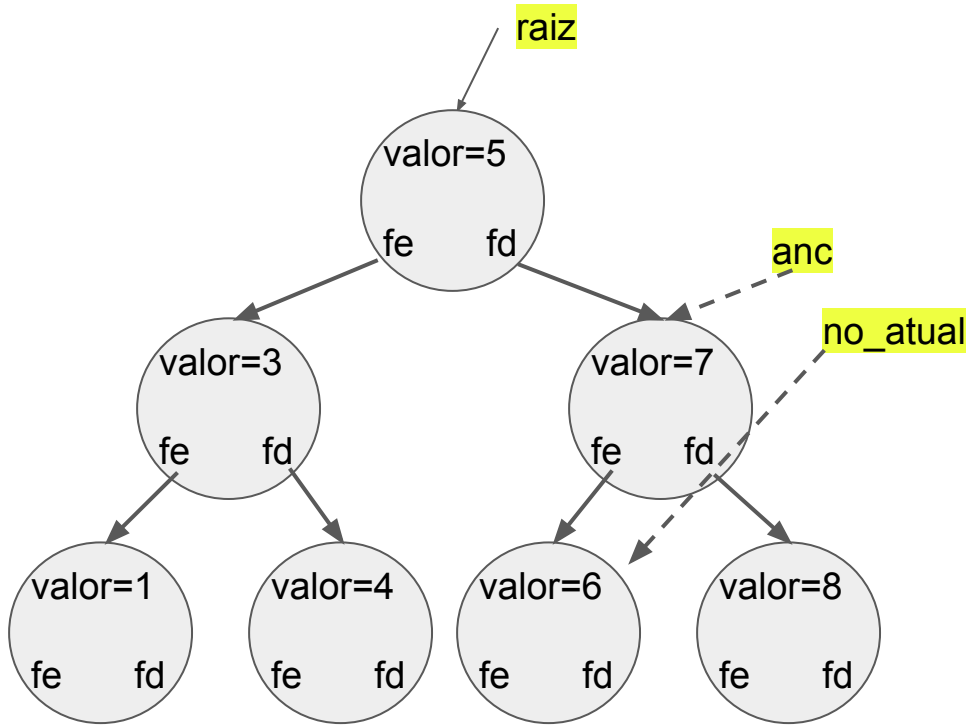
- Iniciar pela raiz e percorrer a árvore com duas variáveis:
  - anc: ancestral (pai).
  - no_atual: nó a ser buscado.
- Por exemplo, buscar(raiz,6)
  - buscar o nó com valor 6.
  - anc = no(7) e no_atual=no(6)



Árvore Binária (Busca)

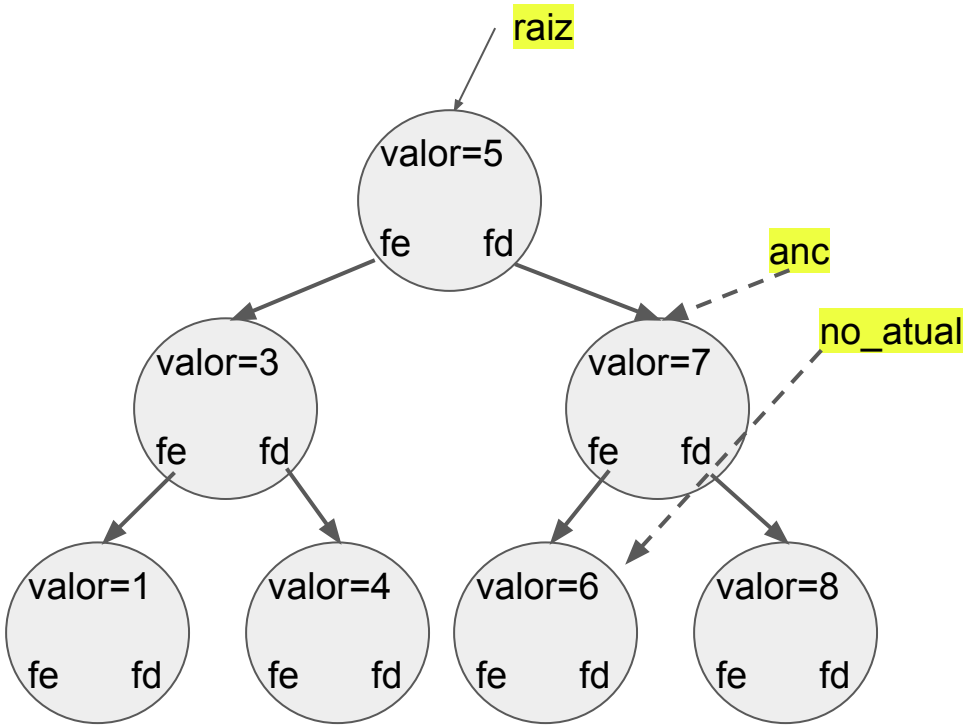
- Vamos criar um exemplo dessa árvore e em seguida testar nossa busca.

Código



Árvore Binária (Busca)

- buscar(raiz,valor):
  - **no_anc**=vazio
  - no_atual = raiz
  - **Enquanto** no_atual ≠ vazio **e** no_atual ≠ valor **faça**
    - **no_anc** = no_atual
    - **Se** valor < no_atual.chave **Então**
      - no_atual = no_atual.FE
    - **Senão**
      - no_atual = no_atual.FD
  - **retornar** no_atual, **no_anc**



Árvore Binária (Busca)

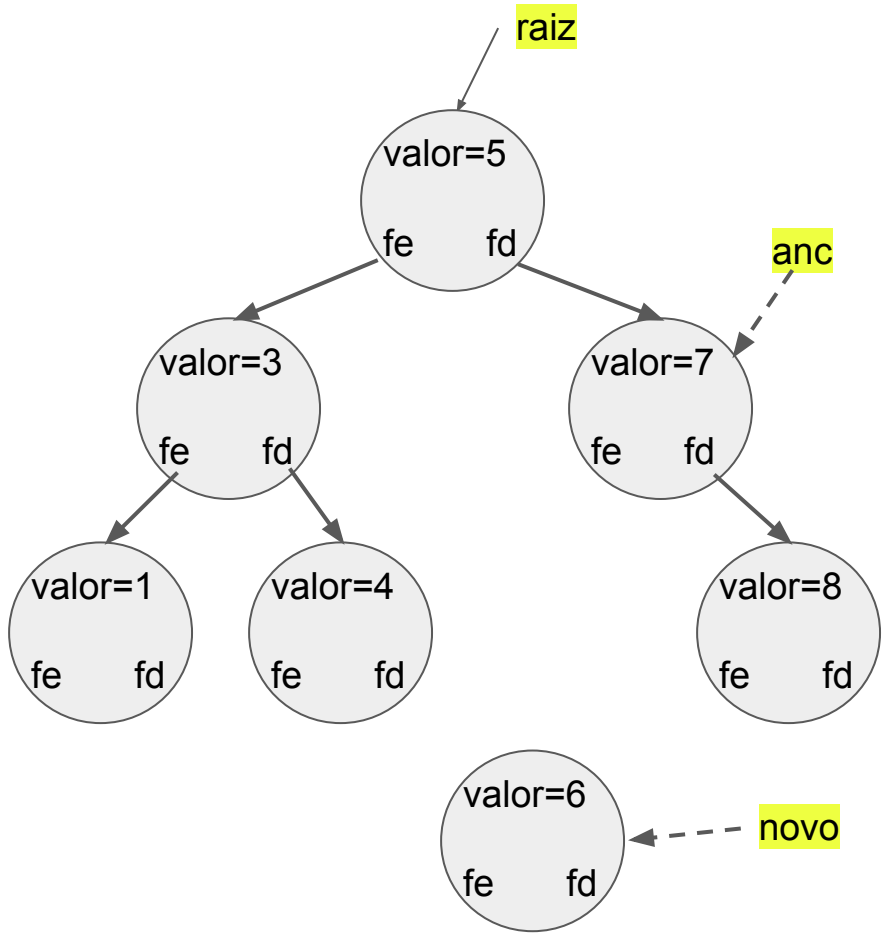
- buscar(raiz,valor):
  - **no_anc**=vazio
  - no_atual = raiz
  - **Enquanto** no_atual ≠ vazio **e** no_atual ≠ valor **faça**
    - **no_anc** = no_atual
    - **Se** valor < no_atual.chave **Então**
      - no_atual = no_atual.FE
    - **Senão**
      - no_atual = no_atual.FD
  - **retornar** no_atual, **no_anc**

```
def busca(raiz,valor):
    no_anc=None
    no_atual=raiz
    while no_atual!=None and no_atual.chave!=valor:
        no_anc=no_atual
        if valor <no_atual.chave:
            no_atual=no_atual.fe
        else:
            no_atual=no_atual.fd
    return no_atual,no_anc

r=criaExemplo()
n,anc=busca(r,6)
```

Implemente a busca e aplique na árvore do exemplo (criaExemplo()).

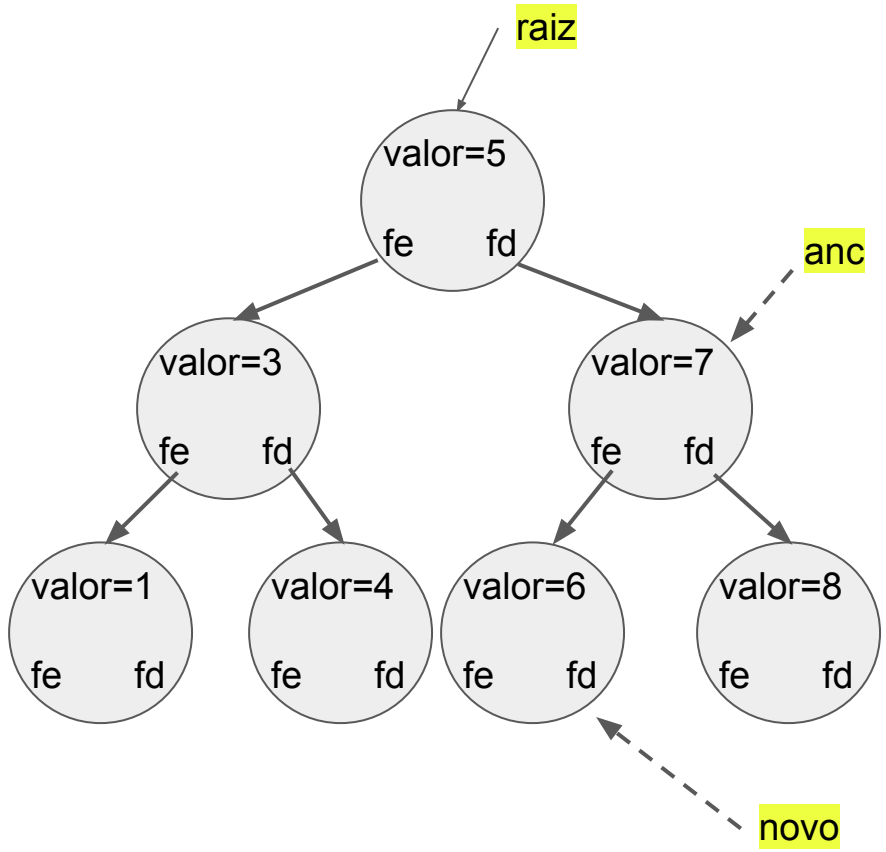
Árvore Binária (Inserção)

- Juntando todas as funções...
  - Vamos inserir o valor 6.
- **def** inserir(raiz,valor):
  - novo = no(valor)
  - **if** raiz==None:
    - raiz=novo
  - **else:**
    - _ , anc = buscar(raiz,valor)
    - conectar(anc,novo)
- inserir(raiz,6)



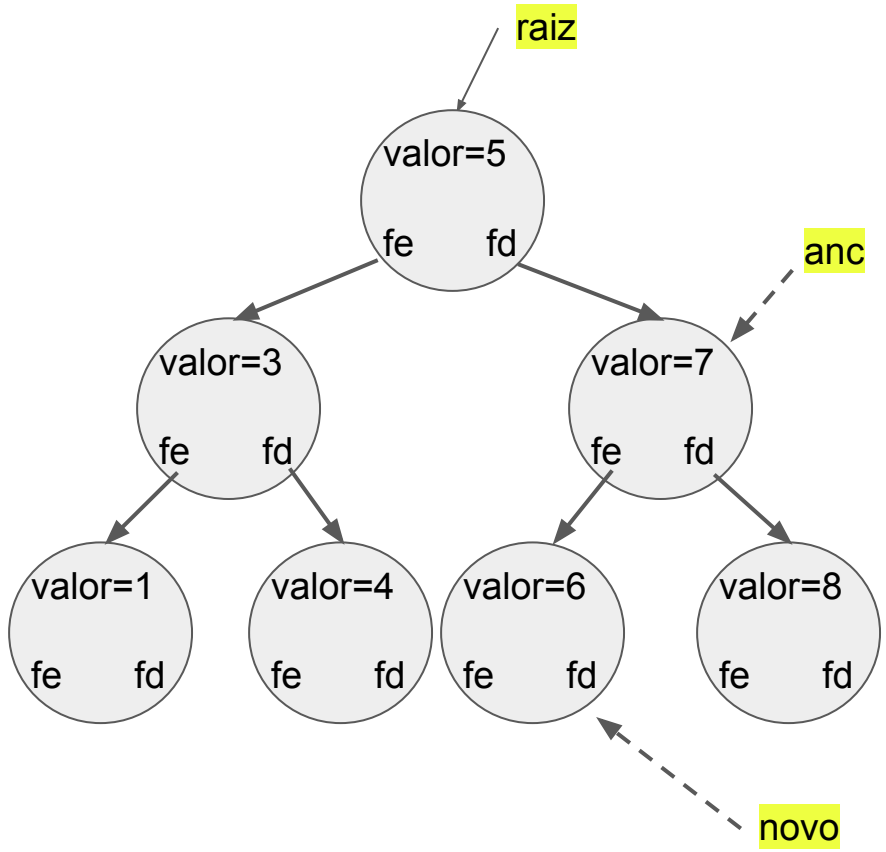
Árvore Binária (Inserção)

- Juntando todas as funções...
  - Vamos inserir o valor 6.
- **def** inserir(raiz,valor):
  - novo = no(valor)
  - **if** raiz==None:
    - raiz=novo
  - **else:**
    - _ , anc = buscar(raiz,valor)
    - conectar(anc,novo)
- inserir(raiz,6)



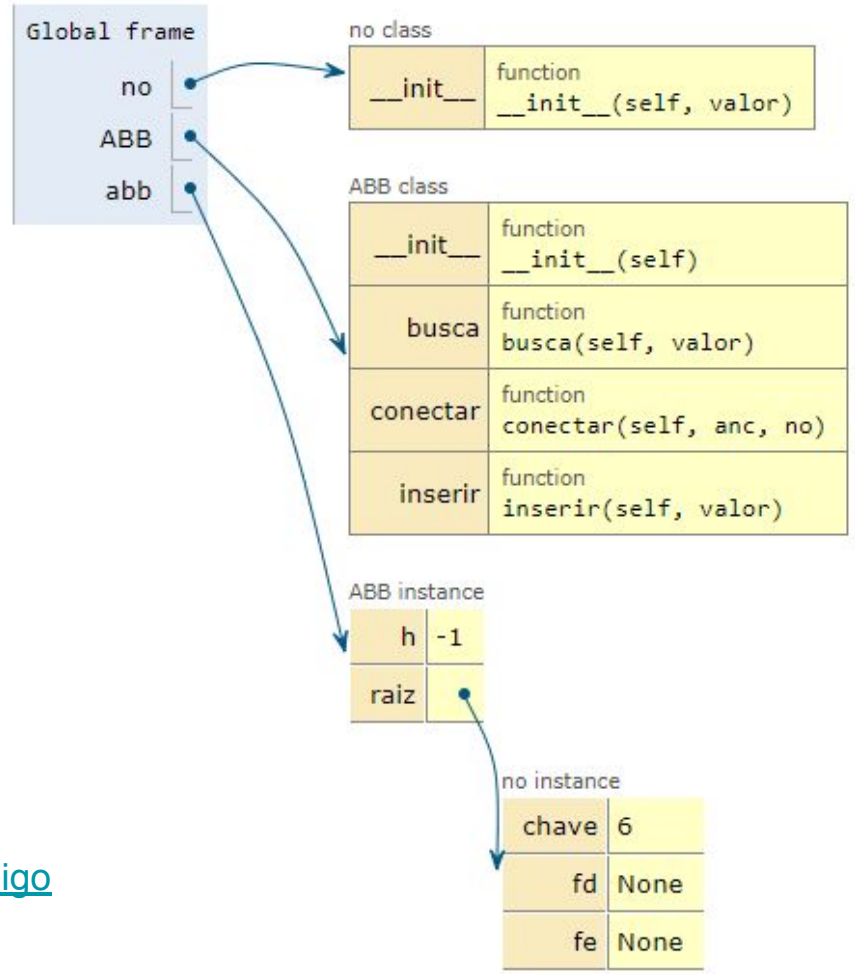
Árvore Binária (Classe)

- Agora podemos criar a nossa classe ABB que conterá todas as funções implementadas.
  - conectar(raiz,no)
  - buscar(raiz,valor)
  - inserir(raiz,valor)



Árvore Binária (Classe)

```python
class no:
    def __init__(self,valor):
        self.chave=valor #Campo chave
        self.fe=None #Campo FE inicialmente vazio(None)
        self.fd=None #Campo FD inicialmente vazio(None)

class ABB:
    def __init__(self):
        self.raiz=None
        self.h=-1 #Altura da árvore vazia

    def buscar(self,valor): #Não precisamos do parâmetro raiz
        no_anc=None
        no_atual=self.raiz
        #....Código...
        return no_atual,no_anc

    def conectar(self, anc, no):
        #....Código...
        pass

    def inserir(self,valor): #Não precisamos do parâmetro raiz
        novo = no(valor)
        if self.raiz==None:
            self.raiz = novo
        else:
            _,anc = self.busca(valor)
            self.conectar(anc,novo)

abb = ABB()
abb.inserir(6)
```



Código

Referências

CORMEN, Thomas. **Algoritmos: teoria e prática**. Rio de Janeiro: GEN LTC, 2013. ISBN 9788595158092. Disponível na Biblioteca Digital da UFMS.

SZWARCFITER, Jayme Luiz; MARKENZON, Lilian. **Estruturas de dados e seus algoritmos**. 3. ed. Rio de Janeiro, RJ: LTC, 2010. ISBN 9788521629955. Disponível na Biblioteca Digital da UFMS.

Licenciamento