Professor
Edson Pedro FERLIN

Estruturas de Dados

Árvores Binárias de Busca

Prof. Edson Pedro Ferlin

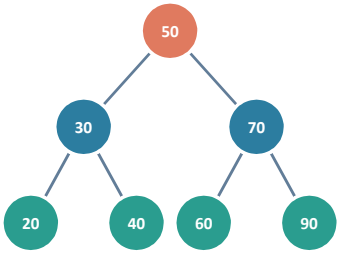
Professor
Edson Pedro FERLIN

Estruturas de Dados

AULA 4 · ÁRVORES BINÁRIAS DE BUSCA

Árvores Binárias de Busca

Conceitos • Inserção • Busca • Percursos • Complexidade



BST – Binary Search Tree

Por que utilizar uma BST?

Imagine localizar um aluno entre **1 milhão de registros.** Qual abordagem é mais eficiente?

Pesquisa sequencial

Percorre a lista item por item, do início ao fim.

até 1.000.000

comparações no pior caso

Árvore organizada (BST)

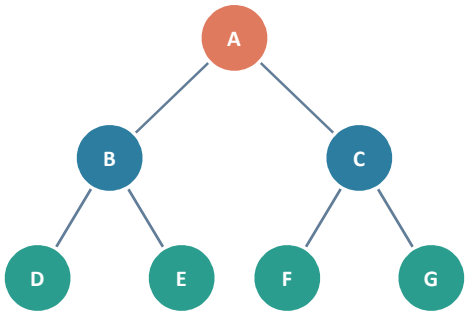
A cada comparação, descartamos metade dos dados.

≈ 20

comparações quando balanceada

O que é uma árvore binária?

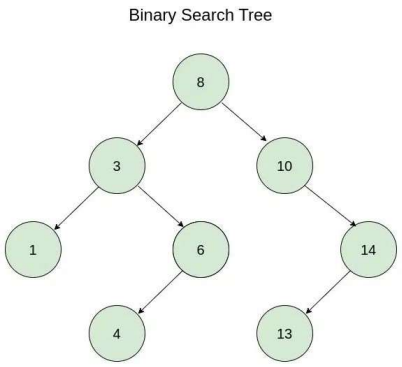
- Um nó raiz no topo da estrutura
- No máximo dois filhos por nó
- Um filho à esquerda
- Um filho à direita



Cada nó pode dar origem a duas subárvores.

Árvore Binária de Busca

- É uma árvore binária em que cada um de seus nós:
  - Todos os valores da subárvore da esquerda são menores
  - Todos os valores da subárvore da direita são maiores
- É eficiente para buscar valores, já que a busca em uma árvore binária genérica exige percorrer até encontrar o nó buscado - O(n)



Definição

- A árvore binária T se denomina binária de busca se:
  - T possui n nós. Cada nó v corresponde a uma chave distinta sj ∈ S e possui como rótulo o valor rt(v) = sj.
  - Seja um nó v de T. Seja também v1, pertencente à subárvore esquerda de v. Então rt(v1) < rt(v).
  - Analogamente, se v2 pertence à subárvore direita de v, rt(v2) >rt(v).

Algoritmos de Busca

- Passo 1: Comparar o valor da raiz com o valor buscado
- Passo 2: Se for maior, repetir o algoritmo para a subárvore da esquerda
- Passo 3: Se for menor, repetir o algoritmo para a subárvore da direita

```
procedimento busca-arvore(x, pt, f)
    se pt = λ então f := 0
    senão se x = pt ↑. chave então f := 1
        senão se x < pt ↑. chave então
            se pt ↑. esq = λ então f := 2
            senão pt := pt ↑.esq
                busca-arvore(x, pt, f)
            senão se pt ↑. dir = λ então f := 3
                senão pt := pt ↑.dir
                    busca-arvore(x, pt, f)
pt := ptraiz; busca-arvore(x, pt, f)
```


Inserção

- A inserção sempre ocorre em uma folha
- Passo 1:Executamos o algorítmo de busca pelo elemento a ser inserido até que a subárvore a ser buscada seja nula
- Passo 2: Inserir o elemento no lugar dessa árvore nula

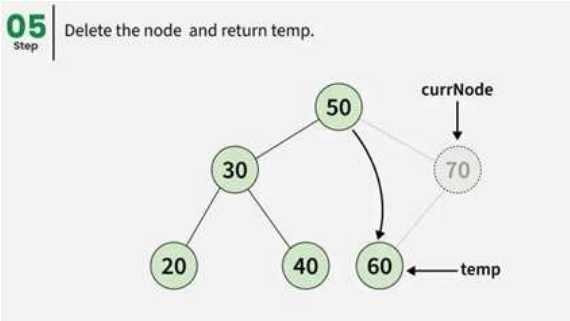
```
pt := ptraiz; busca-arvore(x, pt, f)
se f = 1 então
    "inserção inválida"
senão ocupar(pt1)
    pt1 ↑. chave := x; pt1 ↑. info := novo-valor
    pt1 ↑. esq := λ; pt1 ↑. dir := λ
    se f = 0 então ptraiz := pt1
    senão se f = 2 então
        pt ↑. esq := pt1
        senão pt ↑. dir := pt1
```

Remoção

- A remoção de 1 nó em uma árvore binária de busca é um pouco mais complexa
  - Isso ocorre porque temos que manter a estrutura de árvore binária de busca após a remoção
- Pode-se dividir esse problema em 3 casos:
  - Caso 1: O nó é uma folha
  - Caso 2: O nó possui 1 filho
  - Caso 3: O nó possui 2 filhos
- No caso 1, basta remover o nó em questão

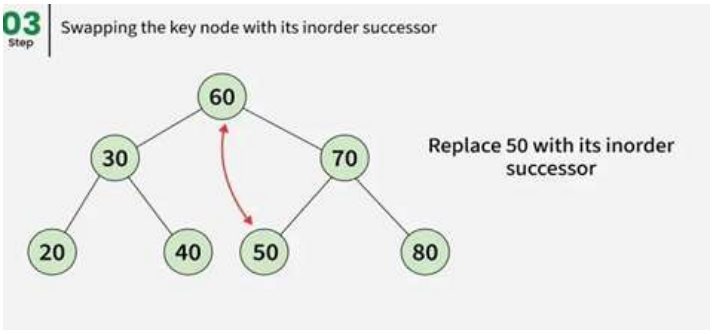
Remoção (Caso 2)

- Remove-se o nó a ser excluído e seu filho assume seu lugar na árvore



Remoção (Caso 3)

- Remove-se o nó a ser excluído e o seu sucessor/antecessor no percurso em ordem assume seu lugar na árvore



O que é uma BST?

Uma Árvore Binária de Busca é uma árvore binária que mantém uma regra de ordenação em todos os nós.

Subárvore esquerda	Subárvore direita
Todos os valores são **menores** que o nó atual.	Todos os valores são **maiores** que o nó atual.
`esquerda < nó`	`nó < direita`

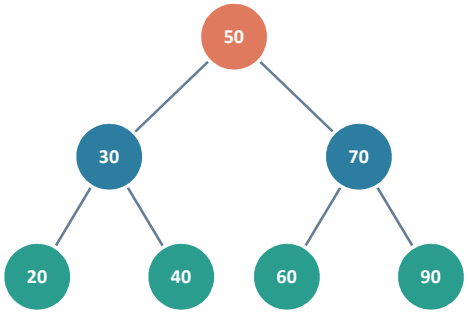
A regra fundamental

```
esquerda  <  X  <  direita
```

Para qualquer nó X, essa relação vale em toda a árvore — do topo até as folhas.

Exemplo

`30 < 50 < 70  →  20 < 30 < 40`



Árvore Binária × BST

Árvore binária	BST
Sem ordenação	Ordenada
Busca sequencial	Busca eficiente
Qualquer organização	Segue a regra de ordenação

A estrutura de um nó

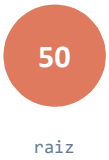
```
class No:
    def __init__(self, valor):
        self.valor = valor
        self.esquerda = None
        self.direita  = None
```

- valor — o dado armazenado no nó
- esquerda — ponteiro para o filho esquerdo
- direita — ponteiro para o filho direito

Criando a raiz

```
raiz = No(50)
```

A primeira inserção cria o nó raiz — o ponto de partida de toda a árvore.

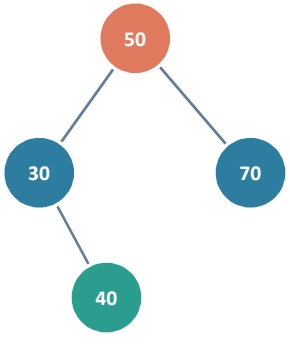


Inserindo valores na árvore

Como adicionar novos nós mantendo a regra de ordenação.

Onde inserir o valor 40?

1. **Começamos na raiz**
   Comparamos 40 com 50.
2. **40 < 50**
   Como é menor, seguimos para a esquerda.
3. **Chegamos ao nó 30**
   40 > 30 → vai para a direita de 30.



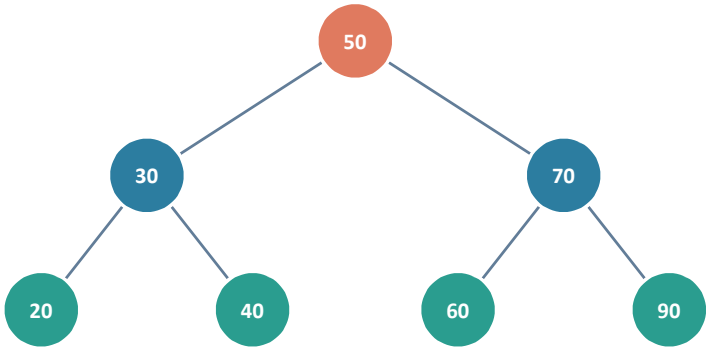
`40 entra aqui`

Construindo a árvore

Inserindo os valores nesta ordem:

A raiz é fixada em 50; cada valor seguinte desce pela árvore comparando-se aos nós até encontrar seu lugar.

A árvore construída



Árvore balanceada — cada nível dobra a capacidade de armazenamento.

A lógica da inserção

Se a árvore está vazia

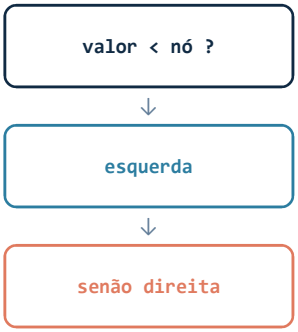
cria a raiz com o novo valor

Caso contrário, compare:

menor → segue para a esquerda

maior → segue para a direita

Repete até encontrar um espaço vazio.



Inserção recursiva

```
def inserir(no, valor):
    if no is None:
        return No(valor)
    if valor < no.valor:
        no.esquerda = inserir(no.esquerda, valor)
    else:
        no.direita  = inserir(no.direita, valor)
    return no
```

- Caso base: nó vazio → cria o nó
- Menor → recorre à esquerda
- Maior ou igual → recorre à direita
- Retorna o nó atualizado

Testando a inserção

```
raiz = None

for valor in [50, 30, 70, 20, 40, 60, 90]:
    raiz = inserir(raiz, valor)
```

Cada chamada devolve a árvore atualizada, que volta a ser atribuída a raiz.

Atividade – Árvore Binária

Insira os valores na ordem dada e desenhe a árvore resultante:

40 15 60 10 30 50 70 5 12

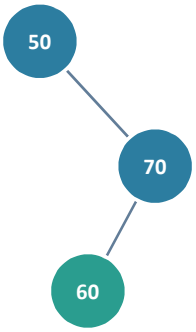
Dica o primeiro valor (40) é sempre a raiz. Compare cada valor seguinte e desça pela árvore até achar um espaço livre.

Buscando valores na árvore

Localizar um valor comparando-o aos nós, nível a nível.

Buscando o valor 60

1. Compara com 50 → 60 é maior, vai à direita
2. Compara com 70 → 60 é menor, vai à esquerda
3. ✓ Compara com 60 → encontrado!



`50 → 70 → 60`

A decisão em cada nó

`valor == nó ?`	`valor < nó ?`	`valor > nó ?`
↓	↓	↓
encontrou	vai à esquerda	vai à direita

Busca recursiva

```
def buscar(no, valor):
    if no is None:
        return False
    if valor == no.valor:
        return True
    if valor < no.valor:
        return buscar(no.esquerda, valor)
    return buscar(no.direita, valor)
```

- Nó vazio → não encontrado
- Igual → encontrado
- Menor → busca à esquerda
- Maior → busca à direita

Executando a busca

```
print(buscar(raiz, 60))
print(buscar(raiz, 15))
```

Saída

```
True
False
```

O valor 60 existe na árvore; 15 não foi inserido, logo a busca retorna False.

Busca iterativa

```
def buscar(no, valor):
    while no:
        if valor == no.valor:
            return True
        elif valor < no.valor:
            no = no.esquerda
        else:
            no = no.direita
    return False
```

- Mesma lógica, sem recursão
- Um laço percorre os nós
- Usa menos memória na pilha

Complexidade - Busca linear × BST

Busca linear

`20 → 30 → 40 → 50 → 60 → 70`

Verifica cada elemento até encontrar.

O(n)

Busca em BST

`50 → 70 → 60`

Descarta metade dos dados a cada passo.

O(log n)

Percorrendo a árvore

Três formas de visitar todos os nós de uma árvore.

Os três percursos

1

Pré-ordem

`raiz → esquerda → direita`

2

Em ordem

`esquerda → raiz → direita`

3

Pós-ordem

`esquerda → direita → raiz`

Em ordem produz valores ordenados

Numa BST, o percurso em ordem visita os valores em ordem crescente — uma de suas propriedades mais importantes.

20 < 30 < 40 < 50 < 60 < 70 < 90

Ordenação crescente garantida pela estrutura da BST.

Percurso Em ordem

```
def em_ordem(no):
    if no:
        em_ordem(no.esquerda)
        print(no.valor)
        em_ordem(no.direita)
```

- Visita toda a subárvore esquerda
- Imprime o valor do nó
- Visita toda a subárvore direita

Atividade – Percurso em Árvore

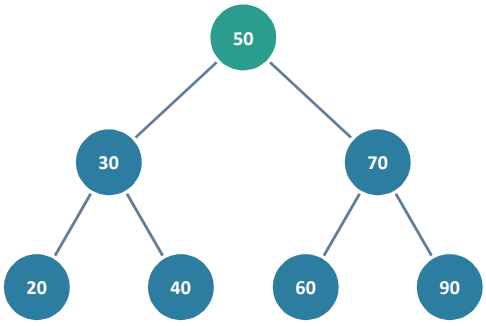
Considerando a árvore construída (raiz 50), qual a saída de cada percurso?

Pré-ordem	Em ordem	Pós-ordem
? ? ? ? ?	? ? ? ? ?	? ? ? ? ?

Desempenho da BST

Quão rápida é a árvore? Depende do seu formato.

Árvore balanceada



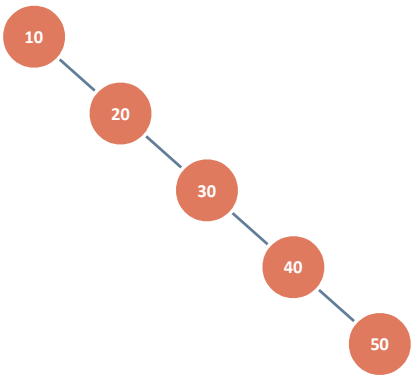
Altura

$\approx \log_2(n)$

Os nós se distribuem de forma equilibrada, então a altura cresce lentamente.

Árvore degenerada

Inserindo valores já ordenados — 10, 20, 30, 40, 50 — a árvore vira uma lista.

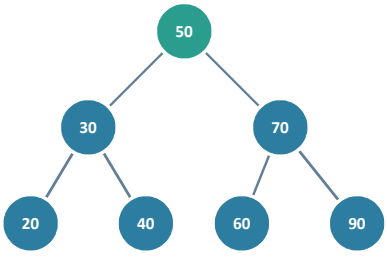


Vira uma lista encadeada

A altura passa a ser n, e a busca degrada para O(n).

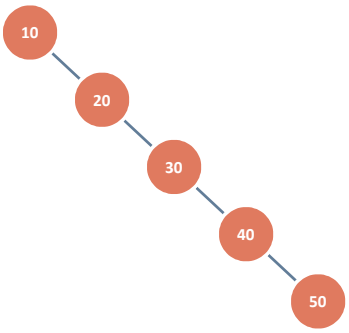
Complexidade - Balanceada × degenerada

Balanceada



O(log n)

Degenerada



O(n)

Complexidade das operações

Operação	Melhor caso	Pior caso
Busca	O(log n)	O(n)
Inserção	O(log n)	O(n)
Percurso	O(n)	O(n)

Complexidade - BST × outras estruturas

Estrutura	Busca
Lista	O(n)
BST balanceada	O(log n)
Vetor ordenado	O(log n)
Tabela hash	O(1) médio

Atividade – Árvore Binária de Busca

Insira, na ordem, os valores abaixo e desenhe a BST:

45 15 79 10 20 60 90 12 18 22

Depois, realize as buscas:

`buscar 20` `buscar 90` `buscar 50`

Dica: 50 não está na árvore — o que a busca deve retornar?

Implemente as funções:

Contar nós

```
def contar(no):
    # sua solução
    ...
```

Quantos nós a árvore possui?

Calcular altura

```
def altura(no):
    # sua solução
    ...
```

Qual o caminho mais longo até uma folha?

Monte a classe:

```
class BST:
    def inserir(self, valor): ...
    def buscar(self, valor):  ...
    def imprimir(self):       ...
```

- inserir() — adiciona um valor
- buscar() — verifica se existe
- imprimir() — mostra em ordem

Utilização

- Sistemas de arquivos
- Índices de banco de dados
- Compiladores
- Dicionários
- Sistemas de busca
- Tabelas ordenadas

Exercícios

1) Implemente os métodos de busca em pré-ordem e pós-ordem na classe de árvore binária de busca.

2) Implemente o método de busca em uma árvore binária de busca.

3) Implemente os métodos de inclusão e exclusão na árvore binária de busca.

4) Explique porque se uma árvore binária de busca tiver seus nós inseridos na seguinte sequência (1, 4, 5, 7, 8, 10, 12) a sua busca se torna "ineficiente" (equivalente a de uma estrutura linear).

5) Desenhar uma árvore binária de busca T para S={2, 5, 3, 7, 8, 10, 6} com as seguintes propriedades:
- T possui altura máxima
- T possui altura

Problema

Desenvolva uma classe BST completa em Python com os métodos abaixo:

```
class BST:
    inserir()       buscar()
    em_ordem()      pre_ordem()
    pos_ordem()     altura()
    contar_nos()    contar_folhas()
```

Entrada

```
dados = [50, 30, 70, 20, 40,
         60, 90, 10, 25, 35, 45]
```

- Construir a BST automaticamente a partir dos dados
- Exibir os três percursos (pré, em e pós-ordem)
- Informar a altura da árvore e o total de nós e folhas
- Permitir buscas por diferentes valores

Contato

eferlin@live.com

(BLOG) professorferlin.blogspot.com

(SITE) professorferlin.com.br

(YOUTUBE) ProfEdsonPedroFerlin