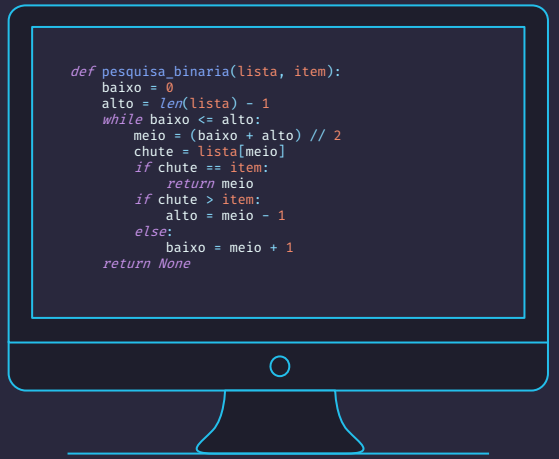
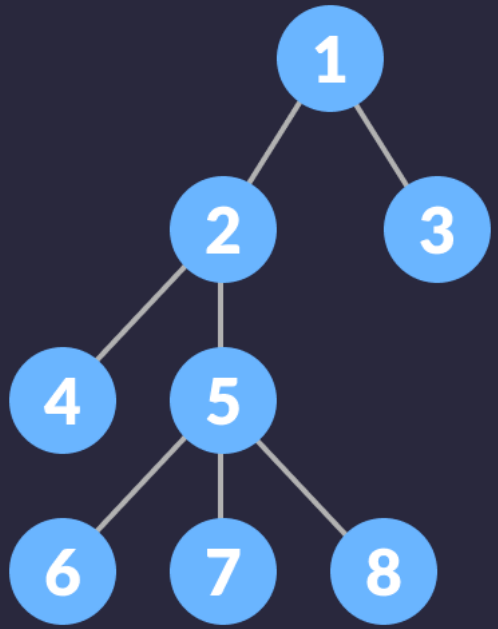
Árvores

Professor Roberto Cândido



Árvores

Árvores são estruturas de dados hierárquicas. Basicamente, árvores são formadas por um conjunto de elementos, os quais chamamos nodos (ou vértices) conectados de forma específica por um conjunto de arestas. Um dos nodos, que dizemos estar no nível 0, é a raiz da árvore, e está no topo da hierarquia. A raiz está conectada a outros nodos, que estão no nível 1, que por sua vez estão conectados a outros nodos, no nível 2, e assim por diante.

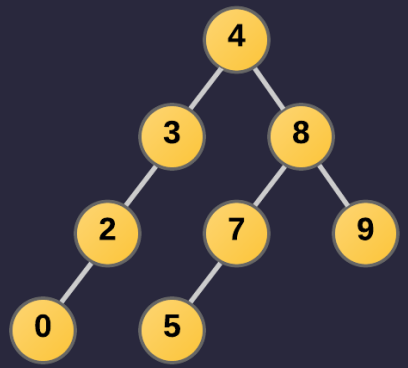

1
2
3
4
5
6
7
8

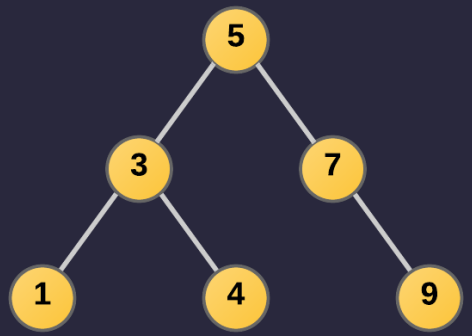
As conexões entre os nodos de uma árvore seguem uma nomenclatura genealógica. Um nodo em um dado nível está conectado a seus filhos (no nível abaixo) e a seu pai (no nível acima). A raiz da árvore, que está no nível 0, possui filhos mas não possui pai.

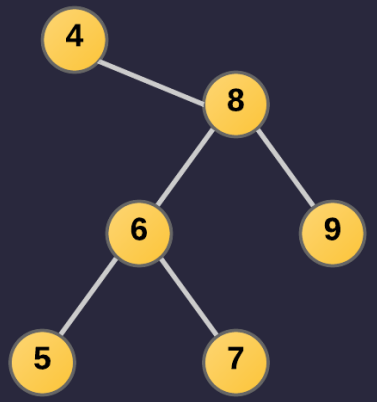
Árvores podem ser desenhadas de muitas formas, mas a convenção em Computação é desenhá-las com a raiz no topo, apesar de isso ser um pouco contraintuitivo de acordo com nossa noção de árvore do cotidiano.

Árvores Binárias

Árvores binárias são árvores nas quais cada nodo pode ter no máximo dois filhos, conforme mostrado na figura abaixo.


4
3
8
2
7
9
0
5


5
3
7
1
4
9

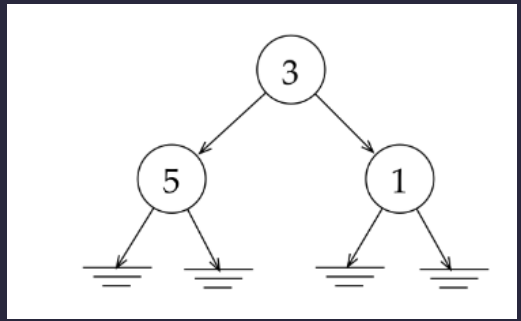

4
8
6
9
5
7

Representação de Árvores Binárias

Na prática, os nodos de uma árvore binária possuem um valor (chamado de chave) e dois apontadores, um para o filho da esquerda e outro para o filho da direita. Esses apontadores representam as ligações (arestas) de uma árvore. Veja abaixo uma implementação de árvore binária.

```python
class NodeTree:
    def __init__(self, data=None, left=None, right=None):
        self.data = data
        self.left = left
        self.right = right
```

```
root = NodeTree(3)
root.left = NodeTree(5)
root.right = NodeTree(1)
print_tree(root)
```



Função auxiliar Para Printar a árvore

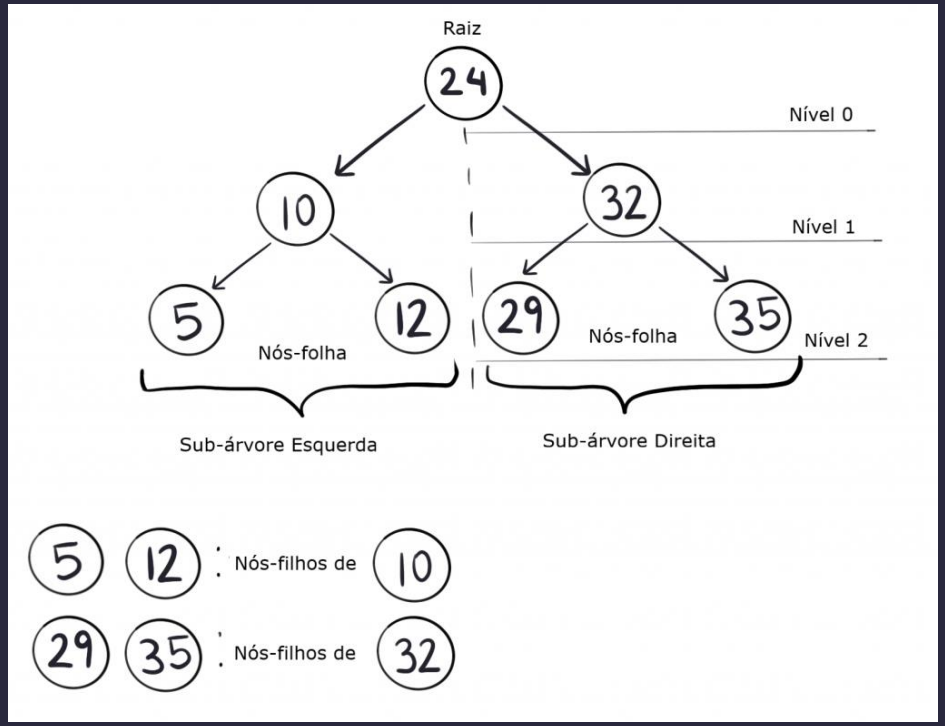
```
def print_tree(root):
    def height(root):
        return 1 + max(height(root.left), height(root.right)) if root else -1

    nlevels = height(root)
    width = pow(2, nlevels + 1)

    q = [(root, 0, width, 'c')]
    levels = []

    while (q):
        node, level, x, align = q.pop(0)
        if node:
            if len(levels) <= level:
                levels.append([])

            levels[level].append([node, level, x, align])
            seg = width // (pow(2, level + 1))
            q.append((node.left, level + 1, x - seg, 'l'))
            q.append((node.right, level + 1, x + seg, 'r'))

    for i, l in enumerate(levels):
        pre = 0
        preline = 0
        linestr = ''
        pstr = ''
        seg = width // (pow(2, i + 1))
        for n in l:
            valstr = str(n[0].data)
            if n[3] == 'r':
                linestr += ' ' * (n[2] - preline - 1 - seg - seg // 2) + '¯' * (seg + seg // 2) + '\\'
                preline = n[2]
            if n[3] == 'l':
                linestr += ' ' * (n[2] - preline - 1) + '/' + '¯' * (seg + seg // 2)
                preline = n[2] + seg + seg // 2
            pstr += ' ' * (n[2] - pre - len(valstr)) + valstr
            pre = n[2]
        print(linestr)
        print(pstr)
```

Árvores Binárias de Pesquisa

Árvores binárias de pesquisa (ou Binary Search Tree - BSTs, do Inglês) são árvores cujos nodos são organizados de acordo com algumas propriedades. Árvores binárias de pesquisa são árvores que obedecem às seguintes propriedades:

1 -> Dado um nodo qualquer da árvore, todos os nodos à esquerda dele são menores ou iguais a ele.

2 -> Dado um nodo qualquer da árvore, todos os nodos à direita dele são maiores ou iguais a ele.


Raiz
24
Nível 0
10
32
Nível 1
5
12
29
35
Nós-folha
Nós-folha
Nível 2
Sub-árvore Esquerda
Sub-árvore Direita
5
12
Nós-filhos de
10
29
35
Nós-filhos de
32

Caminhamentos em Árvore

Caminhamentos em árvore são formas de visitarmos todos os nodos de uma árvore em uma ordem pré-definida. Existem três tipos de caminhamentos básicos: pré-ordem, em ordem, e pós-ordem.

A única diferença é na ordem em que os nodos serão impressos.

Exemplo de Árvore Binária:

Pre-order (Root, Left, Right)

A B D E C F

In-order (Left, Root, Right)

D B E A C F

Post-order (Left, Right, Root)

D E B F C A

No caminhamento pré-ordem, visitamos o nodo corrente antes de visitarmos recursivamente os nodos da esquerda e direita.

```
def pre_order(root):
    if not root:
        return

    print(root.data, end=" ")

    pre_order(root.left)
    pre_order(root.right)
```

Em ordem (In-order)

No caminhamento em ordem, visitamos recursivamente o nodo da esquerda antes de visitar o nodo corrente e, em seguida, visitamos recursivamente o nodo da direita.

```
def in_order(root):
    if not root:
        return

    in_order(root.left)
    print(root.data, end=" ")
    in_order(root.right)
```

Pós-ordem (Post-order)

No caminhamento pós-ordem, visitamos recursivamente os nodos da esquerda e direita antes de visitar o nodo corrente.

```
def post_order(root):
    if not root:
        return

    post_order(root.left)
    post_order(root.right)
    print(root.data, end=" ")
```

Inserção em Árvores Binárias de Pesquisa

O maior desafio ao se construir uma função para inserir nodos em uma árvore binária de pesquisa é encontrar o ponto onde cada nodo deve ser inserido. Uma vez encontrado esse ponto, podemos simplesmente ajustar os apontadores esquerda ou direita para que o nodo seja inserido na árvore.

Para encontrar o ponto de inserção de um nodo em uma árvore binária de pesquisa, precisamos observar as propriedades dessas árvores: dado um nodo qualquer, nodos menores do que ele são inseridos à sua esquerda, e nodos maiores do que ele são inseridos à sua direita. Vejamos como transformar essas ideias em código.

```python
def insert(root, node):
    if root is None:
        root = node

    elif root.data < node.data:
        if root.right is None:
            root.right = node
        else:
            insert(root.right, node)

    else:
        if root.left is None:
            root.left = node
        else:
            insert(root.left, node)
```

```
tree = NodeTree(40)

for data in [20, 60, 50, 70, 10, 30]:
    node = NodeTree(data)
    insert(tree, node)

print_tree(tree)
```

Busca em Árvores Binárias de Pesquisa

Assim como a partir das propriedades de árvores binárias de pesquisa fomos capazes de criar um algoritmo para inserir nodos nessas árvores, faremos também para procurar nodos nelas. O algoritmo de busca em árvores binárias de pesquisa pode ser dividido em três casos:

```python
def search(root, data):
    if root is None or root.data == data:
        return root

    if data < root.data:
        return search(root.left, data)
    else:
        return search(root.right, data)
```

1 -> A chave procurada está na raiz da árvore. Nesse caso, simplesmente retornamos a raiz da árvore como resultado da busca.

2 -> A chave procurada é menor que a chave do nodo raiz. Nesse caso, precisamos procurar pela chave somente na sub-árvore esquerda.

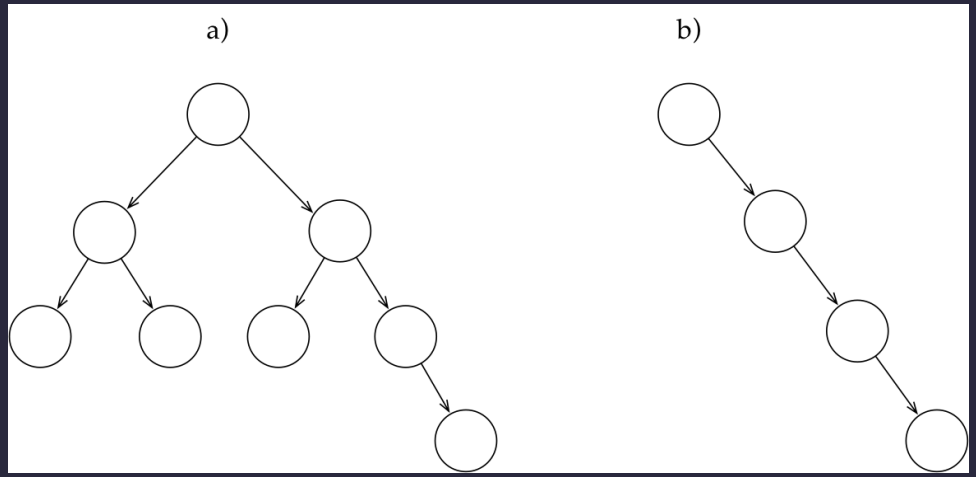
3 -> A chave procurada é maior que a chave do nodo raiz. Nesse caso, precisamos procurar pela chave somente na sub-árvore direita.

```
tree = NodeTree(40)

for data in [20, 60, 50, 70, 10, 30]:
    node = NodeTree(data)
    insert(tree, node)


node = search(tree, 50)
print(node.data)
```

Árvore Binária Balanceada

Uma árvore binária é balanceada se a diferença da profundidade de duas folhas quaisquer é no máximo 1. A profundidade de um nodo é o número de níveis da raiz até aquele nodo.

Na figura abaixo, a árvore a) é balanceada, e a árvore b) não é balanceada.



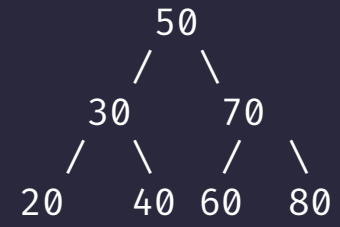
Se uma árvore é balanceada, tanto no caso da inserção quando no caso da busca, a cada chamada recursiva do algoritmo, descartamos metade da árvore original. Portanto, a complexidade assintótica desses dois procedimentos é logarítmica no tamanho (número de nodos) da árvore.

Na verdade, muitos procedimentos que operam sobre árvores binárias de pesquisa funcionam com base nessa mesma ideia de eliminar metade da árvore a cada etapa do procedimento. Isso ocorre por causa da natureza recursiva das árvores binárias de pesquisa e pela forma como os nodos são inseridos nelas.

Entretanto, se a árvore não for balanceada, não descartaremos metade da árvore original a cada chamada recursiva. Em casos como o da árvore não balanceada mostrada acima, a complexidade dos procedimentos de inserção e busca será linear no tamanho da árvore, pois, na prática, a árvore mostrada funciona como se fosse uma lista encadeada.

Exercício 1: Crie duas funções que percorram a árvore binária e retornem o menor (find_min) e o maior (find_max) elementos na árvore.

Ex: dada a seguinte BST:



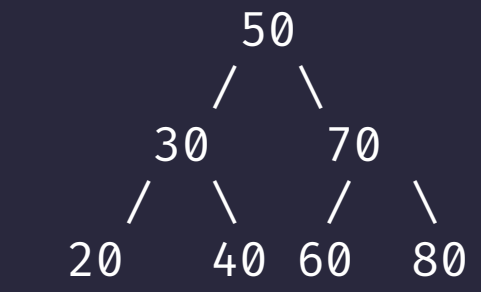
Retornar 20 como menor e 80 como maior elemento.

Exercício 2: Implementando o Percurso In-Order em uma Árvore Binária de Busca (BST) e Armazenando em uma Lista

Descrição

Neste exercício, você deve implementar a função in_order_traversal, que percorre uma Árvore Binária de Busca (BST) e armazena os elementos em uma lista, garantindo que os valores sejam armazenados em ordem crescente.

Exemplo Prático
Dada a seguinte BST:



A saída deve ser: [20, 30, 40, 50, 60, 70, 80]

Exercício 3:Seleção do k-ésimo Menor Elemento em uma Árvore Binária de Busca (BST)

Descrição:
Você deve implementar um programa que insere uma sequência de números em uma Árvore Binária de Busca (BST) e, em seguida, permite ao usuário consultar o k-ésimo menor elemento da árvore.

Regras:
Leia N números inteiros do usuário.
Insira esses números em uma Árvore Binária de Busca (BST).
O usuário deverá fornecer um valor k ($1 \leq k \leq N$).
O programa deverá retornar o k-ésimo menor elemento da árvore.

Desafio: Implementando um Sistema de Autocompletar

Objetivo

Implementar um sistema de autocompletar utilizando uma Árvore Binária de Busca (BST).

O programa deve sugerir palavras com base em um prefixo digitado pelo usuário.

Estrutura do Código
Complete a função autocomplete(self, prefix, result), que:

Busca na árvore todas as palavras que começam com o prefixo fornecido pelo usuário.
Adiciona essas palavras à lista result.
Retorna sugestões de palavras ordenadas.

Exercícios

https://chatgpt.com/c/67b921dd-8abc-800b-9403-622119c66a14