Árvore Binária de Busca

Bruno Bragança, Lorenzo Simonassi, Robson Junior, Pedro Cravo, Henrique Miranda.

Universidade – Universidade Vila Velha (UVV)

Boa Vista - Vila Velha - Espírito Santo

Ciências da Computação
Universidade Vila Velha (UVV) – Vila Velha, ES – Brazil

Resumo. *Árvores binárias de pesquisa, são um clássico na ciência da computação. Elas organizam dados de um jeito que facilita encontrar, adicionar ou remover coisas. Neste trabalho, a gente resolveu implementar uma em Python para colocar em prática esses conceitos e ver como ela se comporta com operações básicas: inserir valores, excluir nós, alterar números e mostrar a estrutura inteira. De quebra, ainda calculamos a altura pra entender o tamanho dela.*

Nosso código traz um menu interativo para facilitar os testes, então quem usa pode mexer na árvore e ver os resultados na hora. Então, vamos explicar o que a gente fez, os desafios que apareceram e como a árvore funciona.

1. Introdução

As árvores binárias de busca são estruturas fundamentais na ciência da computação, utilizadas para organizar dados de forma eficiente, permitindo operações rápidas de busca, inserção e remoção. Seu funcionamento é baseado em um princípio simples: cada nó possui no máximo dois filhos, onde os valores menores ficam à esquerda e os maiores à direita. Esse conceito é amplamente aplicado em bancos de dados, compiladores e sistemas de indexação, demonstrando sua relevância prática.

Neste trabalho, implementamos uma árvore binária de busca em python, explorando suas principais operações. O código permite a inserção de novos valores, a remoção de nós específicos, a alteração de elementos e a exibição visual da estrutura. Além disso, incluímos uma função para calcular a altura da árvore, ajudando a compreender seu crescimento e eficiência. Para facilitar a interação com a BST, criamos um menu interativo, permitindo que o usuário teste as funcionalidades dinamicamente.

Ao longo deste artigo, discutiremos a implementação da árvore, os desafios encontrados e as decisões tomadas no desenvolvimento do código. Também analisaremos o impacto das operações na estrutura da BST e sua eficiência em diferentes cenários. Essa abordagem prática nos ajudará a entender melhor o funcionamento das árvores binárias de busca e sua importância na computação.

2. Desenvolvimento

Este código implementa uma árvore binária de pesquisa. O objetivo dessa estrutura de dados é organizar e armazenar elementos de forma eficiente, mantendo a propriedade que, para cada nó, o valor do nó à esquerda é menor que o valor do nó à direita. Isso facilita a busca, a inserção e a exclusão de elementos na árvore, tornando essas operações rápidas.

A classe principal do código é a Tree, que representa a árvore binária de pesquisa. A árvore possui uma raiz (root), que é o ponto de entrada, e diversos métodos que permitem operar sobre ela, como inserir, excluir, alterar, exibir a árvore e calcular a altura.

2.1. Estrutura de Dados

Cada nó da árvore é representado pela classe No, que contém três componentes essenciais:

- **item**: armazena o valor do nó.
- **dir**: aponta para o nó à direita.
- **esq**: aponta para o nó à esquerda.

Essa estrutura de nó permite que a árvore tenha a propriedade de ser binária, ou seja, cada nó pode ter no máximo dois filhos, um à esquerda e outro à direita.

2.2. Operações na Árvore

Inserção: O método inserir é responsável por adicionar um novo nó à árvore. Ele percorre a árvore, comparando o valor a ser inserido com os valores existentes nos nós, garantindo que o novo nó seja colocado na posição correta, sempre respeitando a ordem da árvore binária de pesquisa. Se o valor for menor que o nó atual, ele será inserido à esquerda, e se for maior, será inserido à direita. Caso o valor já exista na árvore, a inserção é ignorada, evitando duplicações.

Exclusão: A operação de exclusão de um nó envolve três casos:

1. Nó sem filhos (folha): Nesse caso, o nó é removido diretamente.
2. Nó com um filho: O nó é removido e seu único filho assume seu lugar na árvore.
3. Nó com dois filhos: O nó é substituído pelo menor valor da subárvore à direita, e o nó que contém o menor valor é então excluído.

Alteração: Para alterar o valor de um nó, o método alterar primeiro exclui o nó com o valor antigo e, em seguida, insere o novo valor no lugar adequado, mantendo a ordenação da árvore.

Altura: O método altura calcula a altura da árvore, que é definida como o número máximo de arestas que um nó precisa percorrer até atingir uma folha. A altura de uma árvore vazia é considerada -1, enquanto a altura de uma árvore com apenas um nó é `0`.

Este cálculo é útil para avaliar a eficiência de operações como busca e inserção, já que a altura determina o número de comparações necessárias.

Exibição da árvore: O método show_three exibe a árvore de maneira recursiva, mostrando a estrutura da árvore em diferentes níveis. Ele começa pela subárvore à direita e, em seguida, imprime o nó atual e a subárvore à esquerda. Isso resulta em uma visualização em formato de árvore invertida, o que facilita a visualização da estrutura da árvore.

2.3. Interação com o Usuário

O programa oferece uma interface simples, onde o usuário pode escolher entre várias operações para manipular a árvore. O usuário pode:

- Inserir novos valores na árvore.
- Exibir a estrutura atual da árvore junto com sua altura.
- Excluir um valor da árvore.
- Alterar um valor existente.
- Sair do programa.

Essas operações são realizadas por meio de um loop interativo que permite ao usuário interagir com a árvore de forma contínua até decidir sair do programa.

2.4. Considerações Finais

Este código é uma implementação clássica de uma árvore binária de pesquisa, cobrindo as operações essenciais para inserir, excluir, alterar, calcular a altura e exibir a árvore. Ele demonstra de forma clara como essas operações funcionam e como uma árvore binária pode ser manipulada de maneira eficiente. Com a árvore binária de pesquisa, podemos realizar buscas, inserções e exclusões em tempo médio proporcional à altura da árvore, o que garante um bom desempenho mesmo com grandes volumes de dados.

Trecho da implementação:

Imagem 1

```
class No:
    def __init__(self, key, dir=None, esq=None):
        self.item = key
        self.dir = dir
        self.esq = esq
```

Imagem 1. Mostrando a Classe No e um construtor

Imagem 2

```
class Tree:

    def __init__(self):
        self.root = None  # A raiz começa como None, representando uma árvore vazia.

    def inserir(self, v):
        novo = No(v)  # Cria um novo nó
        if self.root is None:  # Se a árvore estiver vazia
            self.root = novo
        else:
            atual = self.root
            while True:
                if v < atual.item:  # Se o valor for menor, ir para a esquerda
                    if atual.esq is None:
                        atual.esq = novo
                        return
                    else:
                        atual = atual.esq  # Continua para a esquerda
                elif v > atual.item:  # Se for maior ou igual, ir para a direita
                    if atual.dir is None:
                        atual.dir = novo
                        return
                    else:
                        atual = atual.dir  # Continua para a direita
                else:
                    return  # Não insere o valor se ele já existir
```

Imagem 2. Imagem mostrando a classe Tree e o início do método inserir

Imagem 3

```
def excluir(self, v):
    self.root = self._excluir(self.root, v)

def _excluir(self, raiz, v):
    # Caso base: se a árvore estiver vazia
    if raiz is None:
        return raiz

    # Se o valor a ser excluído for menor que o valor do nó atual
    if v < raiz.item:
        raiz.esq = self._excluir(raiz.esq, v)

    # Se o valor a ser excluído for maior que o valor do nó atual
    elif v > raiz.item:
        raiz.dir = self._excluir(raiz.dir, v)

    # Caso em que o valor a ser excluído é igual ao valor do nó atual
```

Imagem 3. Uma imagem mostrando a parte 1 do método excluir

Imagem 4

```
else:
    # Caso 1: O nó não tem filhos (é uma folha)
    if raiz.esq is None and raiz.dir is None:
        return None

    # Caso 2: O nó tem um filho à esquerda
    elif raiz.dir is None:
        return raiz.esq

    # Caso 3: O nó tem um filho à direita
    elif raiz.esq is None:
        return raiz.dir

    # Caso 4: O nó tem dois filhos
    else:
        # Encontra o nó com o menor valor na subárvore direita
        minimo = self._minimo(raiz.dir)
        raiz.item = minimo  # Substitui o valor do nó a ser excluído pelo mínimo da subárvore direita
        raiz.dir = self._excluir(raiz.dir, minimo)  # Exclui o nó mínimo da subárvore direita

return raiz
```

Imagem 4. Uma imagem mostrando a parte 2 do método excluir

Imagem 5

```python
def _minimo(self, raiz):
    atual = raiz
    while atual.esq is not None:
        atual = atual.esq
    return atual.item
```

Imagem 5. Uma imagem mostrando o método _minimo

Imagem 6

```python
def alterar(self, v_antigo, v_novo):
    # Exclui o nó com o valor antigo
    self.excluir(v_antigo)
    # Insere o novo valor na posição correta
    self.inserir(v_novo)
```

Imagem 6. Uma imagem mostrando o método alterar

3. Conclusões e resultados

3.1. Conclusão

A implementação da árvore binária de busca (BST) em Python permitiu uma análise detalhada das principais operações associadas a essa estrutura, incluindo inserção, remoção, alteração de valores e exibição da árvore. O desenvolvimento do código proporcionou um melhor entendimento sobre a organização hierárquica dos nós e sua influência na eficiência dessas operações, evidenciando a importância da BST na otimização do armazenamento e recuperação de dados.

Os experimentos realizados demonstraram que a BST é altamente eficiente para operações de busca, inserção e remoção, desde que a árvore permaneça balanceada. Em sua forma ideal, a busca ocorre com complexidade O(logn)O(\log n)O(logn), tornando-a significativamente mais rápida do que métodos lineares. No entanto, observou-se que inserções desordenadas podem resultar em árvores degeneradas, cuja estrutura se assemelha a uma lista encadeada, impactando negativamente o desempenho e aumentando a complexidade para O(n)O(n)O(n). Esse fenômeno destaca a necessidade de estratégias de balanceamento para evitar degradação na eficiência da árvore.

Além disso, a implementação do cálculo da altura da árvore demonstrou ser um recurso valioso para avaliar sua eficiência estrutural. A exibição visual da árvore, por meio do

método de impressão hierárquica, facilitou a compreensão da disposição dos elementos, permitindo uma análise intuitiva do seu comportamento. Concluímos que a BST se mostrou uma estrutura poderosa para armazenar e manipular dados de forma ordenada, sendo essencial para aplicações que exigem acesso rápido a informações. No entanto, para um desempenho otimizado em cenários reais, seria interessante a aplicação de algoritmos de balanceamento, como árvores AVL ou Red-Black, garantindo maior eficiência nas operações em grandes volumes de dados.

3.2. Resultados de códigos

Imagem 7

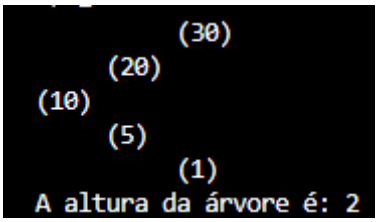

Imagem 7. Uma imagem mostrando uma árvore junto a sua altura

Imagem 8

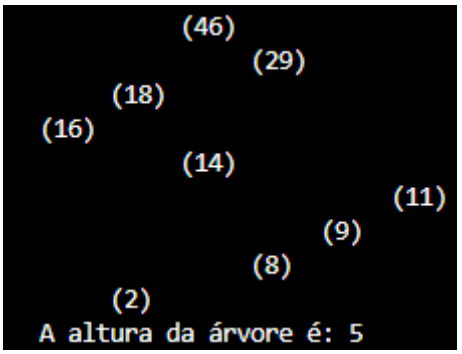

Imagem 8. Uma imagem mostrando uma árvore maior junto a sua altura

4. Tabela de complexidade de tempo

A Árvore Binária de Busca (BST – *Binary Search Tree*) é uma estrutura de dados amplamente utilizada para organizar e buscar informações de maneira eficiente. Sua principal característica é a organização dos nós de forma que, para qualquer nó da árvore, todos os elementos à esquerda são menores e todos os elementos à direita são maiores.

A eficiência das operações realizadas em uma BST depende diretamente do seu balanceamento. Quando a árvore está equilibrada, as operações apresentam um desempenho eficiente de O(log n). No entanto, em casos onde a árvore se torna degenerada (isto é, semelhante a uma lista encadeada), a complexidade pode aumentar para O(n).

Abaixo, detalhamos a complexidade de tempo para as principais operações realizadas em uma BST:

Tabela 1

Operação	Melhor Caso (O)	Caso Médio (Θ	Pior Caso (O)
Inserção	O(log n)	Θ(log n)	O(n)
Busca	O(log n)	Θ(log n)	O(n)
Exclusão	O(log n)	Θ(log n)	O(n)
Mínimo/Máximo	O(log n)	Θ(log n)	O(n)
Altura	O(log n)	Θ(log n)	O(n)
Travessia (InOrder, PreOrder, PostOrder)	O(n)	Θ(n	O(n)

Tabela 1. Uma tabela sobre complexidade de tempo

5. bibliografia

https://pythonhelp.wordpress.com/2015/01/19/arvore-binaria-de-busca-em-python/

https://gist.github.com/divanibarbosa/a8662693e44ab9ee0d0e8c2d74808929

https://www.freecodecamp.org/portuguese/news/arvores-binarias-de-busca-bst-explicada-com-exemplos/