ESTRUTURA DE DADOS

Rafael Albuquerque

SAGAH
SOLUÇÕES EDUCACIONAIS INTEGRADAS

Pesquisa binária

Objetivos de aprendizagem

Ao final deste texto, você deve apresentar os seguintes aprendizados:

- Reconhecer a construção de uma árvore binária de busca.
- Definir a busca binária recursiva.
- Aplicar a busca iterativa em uma árvore binária.

Introdução

Costumeiramente, as árvores, como estrutura de dados, são muito utilizadas para a divisão de dados hierárquicos. As árvores, de modo básico, são estruturas formadas por um conjunto de elementos denominados, muitas vezes, como nós. Deste modo, o nível 0 é constituído pelo elemento raiz da árvore, estando no topo dela. A partir desse nó raiz, surgem as ramificações, que são conhecidas como subárvores.

A estrutura de árvore, por sua versatilidade, pode ser aplicada na solução de um leque de problemas, como visto em otimizações de consultas, como a indexação de bancos de dados, na fase de análise sintática de código por compilador de uma linguagem de programação, em estruturas de diretórios em sistemas de arquivos, e, claro, em sistema de buscas, pelo seu tempo de processamento em O(log(n)) para casos extremos de processamento.

Neste capítulo, veremos como criar uma estrutura de árvore em Python 3.X para a construção de uma árvore binária de busca e efetuar as duas formas de busca em uma árvore, que correspondem às formas recursiva e iterativa. Com a organização em árvores, é possível reduzir o espaço de busca, tornando a tarefa de procura mais intuitiva computacionalmente.

1 Construção de uma árvore binária de busca

A busca binária, naturalmente, está restrita à estrutura na qual cada nó armazenado nela suporta no máximo dois outros nós. Logo, conforme Piva Junior *et al.* (2014), as árvores binárias são caracterizadas como um conjunto finito vazio (ou não) de elementos. Portanto, sua representatividade é definida com a utilização de um nó raiz ligado às suas subárvores esquerda e direita. A Figura 1 demonstra a sua hierarquia, assim como suas subárvores esquerda e direita, em tons de cores laranja e azul, respectivamente.

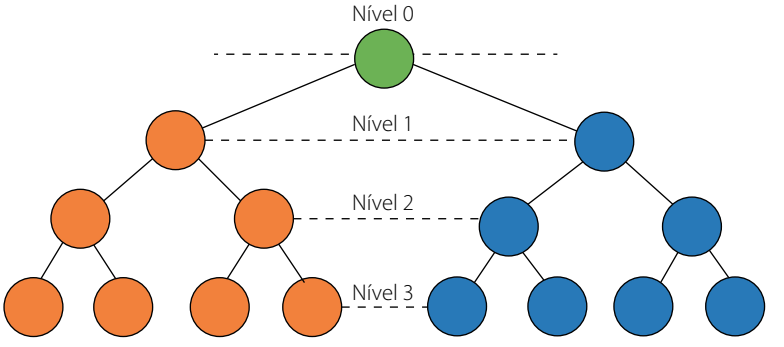

Figura 1. Árvore estritamente binária, com o nó raiz preenchido na cor verde; subárvore esquerda em tom alaranjado e subárvore direita em tom azulado.

Na árvore demonstrada na Figura 1, é possível observar os elementos raiz, subárvore esquerda e subárvore direita em destaque. É possível observar, também, que cada nó filho é um nó raiz da subárvore gerada a partir dele. Conforme Celes, Cerqueira e Rangel (2004), em uma árvore binária, cada nó pode ter: zero, um ou dois filhos. De forma recursiva, uma árvore binária pode ser definida como:

- uma árvore vazia;
- um nó raiz tendo duas subárvores, identificadas como subárvore esquerda e subárvore direita.

Conforme destaca Celes, Cerqueira e Rangel (2004), uma propriedade fundamental de todas as árvores binárias é que só existe um caminho, que parte da raiz para qualquer outro nó. Esse fundamento é significativo, pois é possível determinar a altura de uma árvore levando-se em consideração o comprimento da maior distância entre o nó raiz até uma das folhas.

O processo de criação de uma árvore binária de busca é bem simples, com a seguinte propriedade: a subárvore esquerda é composta por todos os elementos menores ou iguais que a raiz da árvore, enquanto os nós da subárvore à sua direita têm chave maior que a chave do nó em questão. Dessa forma, vamos construindo a árvore. Observe a Figura 2 e pense: caso fosse adicionado mais um elemento de valor 45, em qual subárvore seria alocado?

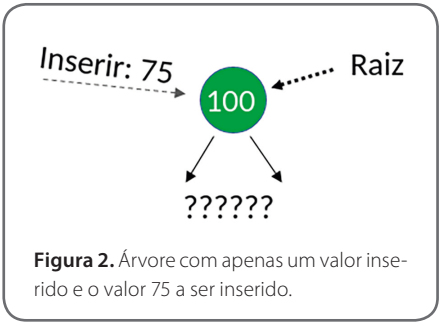

Figura 2. Árvore com apenas um valor inserido e o valor 75 a ser inserido.

Seguindo as regras supracitadas, teremos que adicionar o seu valor à subárvore esquerda, para o caso de termos que adicionar o valor 115 à árvore, obtendo o seguinte resultado apresentado na Figura 3.

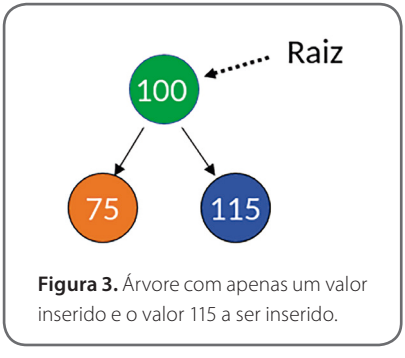

Figura 3. Árvore com apenas um valor inserido e o valor 115 a ser inserido.

Uma vez que criamos o entendimento básico de como preencher uma árvore, veremos, a seguir, como podemos implementar, de forma a automatizar os procedimentos de inserção de novos elementos, de maneira dinâmica, ou seja, alocando espaço em tempo de execução.

Implementação de uma árvore

Para implementarmos nossa árvore, vamos usar a linguagem de programação Python 3.X com o auxílio da ferramenta *on-line* Repl, o que nos fornecerá menos linhas de códigos, porém, ao invés de usar *struct*, comumente utilizado em linguagens como C/C++, usaremos o conceito básico de classe para adicionarmos um novo elemento. Logo, cada nó será constituído conforme apresentado na Figura 4.

```
class No(object):
  def __init__(self, chave):
    self.chave = chave
    self.esq = None
    self.dir = None
```

Figura 4. Definição de cada nó constituinte da árvore.

Para o controle de objetos do tipo `No`, ao ponto de deixar a sua criação de forma automática, vamos adicionar mais uma classe, chamada `Arvore`, na qual deixaremos de forma explícita a sua criação, com um atributo denominado raiz. Este elemento ajudará no controle de inserção dos valores à árvore. Veja a Figura 5, na qual apresentamos a sua implementação, seguida das funções **inserir** e **imprimir**.

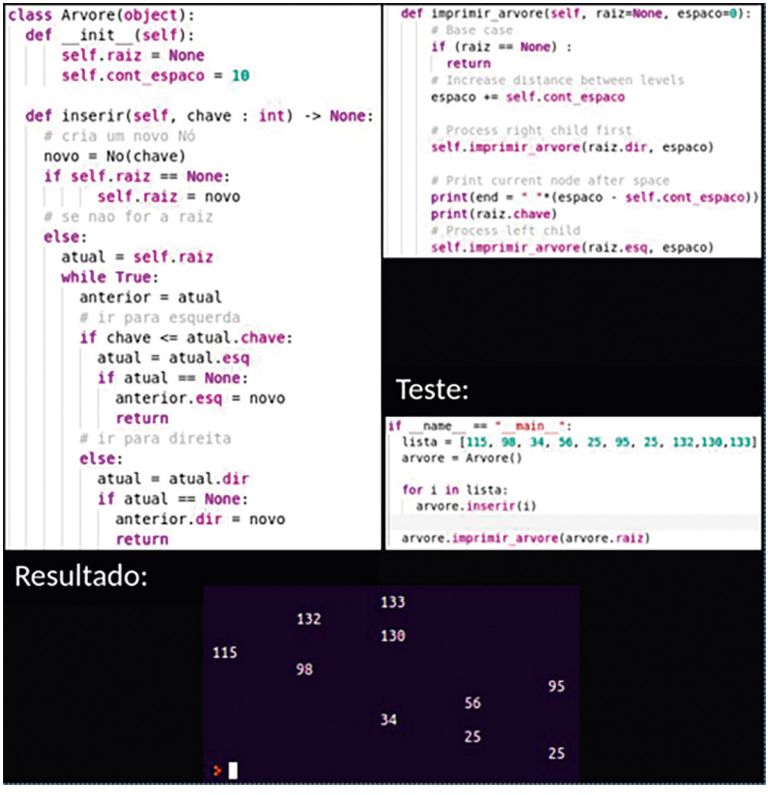

Figura 5. Criação da árvore, inserção de novos nós e impressão da árvore.

A implementação apresentada na Figura 5 é de viés simples, na qual apenas criamos uma classe que contém um atributo raiz para controle do primeiro elemento que será adicionado à arvore, assim como o atributo que auxilia na impressão da árvore da esquerda para direita, como resultado dos elementos adicionados. Além disso, como optamos por automatizar a posição de cada elemento adicionado, perceba que, de forma iterativa, a função de inserir está fazendo o comparativo da chave com o valor raiz de cada subárvore até encontrar uma posição na árvore para o seu valor. Com base nessa implementação, daremos continuidade para as abordagens sobre busca do elemento em função recursiva e de modo iterativo.

Fique atento

As árvores também podem ser implementadas sob listas fixas ou estáticas, que ocorrem por meio da distribuição dos nós da árvore ao longo de um vetor (*array*). Essa distribuição ocorre da seguinte forma: o nó raiz será o primeiro elemento do vetor, e para cada nó em determinada posição **i** do vetor, seu filho esquerdo ficará na posição 2*i + 1, e seu filho direito ficará na posição 2*i + 2. Atente-se para quando for implementá-las nesse estilo, para a quantidade de memória que será necessária para colocá-las em operação.

Link

Confira a ferramenta Repl no *link* a seguir.

https://qrgo.page.link/zU2Zv

2 Busca binária recursiva

As árvores binárias de busca recebem esse nome justamente pelas suas estruturas, que permitem a realização da busca de forma eficiente, sendo fácil observar o seu funcionamento. Veja a sequência apresentada na Figura 6 para a busca da chave número 91.

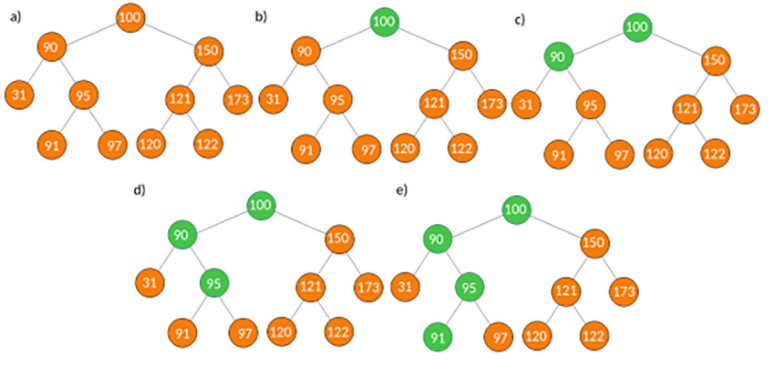

Figura 6. Busca da chave 91.

Perceba que a chave 91 está no que chamamos de folha da árvore, e esta folha está localizada no último nível da árvore. Para a simulação indicada, a chave 91 precisou ser comparada com o primeiro elemento, cujo valor é 100. Como 91 é menor que 100, o caminho a ser perseguido pertence à subárvore esquerda. Em seguida, apesar de estar na subárvore esquerda, a chave de número 91 é maior que o valor raiz 90, indicando que a próxima subárvore a ser visitada será à direita. Todavia, na próxima subárvore de raiz 95, a chave 91 é menor, indicando que terá que continuar a percorrer o caminho pela subárvore à esquerda. Por fim, a chave 91 é identificada, o que encerra a busca.

Dessa forma, fica fácil identificar que, a partir das propriedades de árvores binárias de pesquisa, com a função de inserir, podemos desenvolver a função de busca das chaves contidas. Para isso, o algoritmo de busca em árvores binárias pode ser dividido em três casos:

- Em que a chave procurada está contida na raiz. Logo, basta retornar a raiz da árvore como resultado da busca.
- Em que a chave é menor ou igual à chave do nó raiz. Caso isso ocorra, será preciso passar a subárvore esquerda (de modo recursivo ou iterativo) para continuar a procura.
- Em que a chave procurada é maior que a do nó raiz. Neste último caso, de modo análogo ao modo anterior, basta que a procura continue pela subárvore à direita.

Implementação da recursão

Conforme definimos as propriedades básicas, podemos implementar a função de busca de modo recursivo, seguindo os passos descritos anteriormente. Observe a Figura 7.

Definição de função

```
def busca_recursiva(self, chave):
    return self._busca_recursiva(self.raiz, chave)

def _busca_recursiva(self, no: No, chave:int):
    if no != None:
        print("Visitou chave: {}".format(no.chave))
        if no.chave == chave:
            return True
        elif chave <= no.chave:
            return self._busca_recursiva(no.esq, chave)
        elif chave > no.chave:
            return self._busca_recursiva(no.dir, chave)
    return False
```

Execução da função

```
if __name__ == "__main__":
    lista = [115, 98, 34, 56, 25, 95, 25, 132,130,133]
    arvore = Arvore()
    buscar = 56
    for i in lista:
        arvore.inserir(i)

    print("Impressão da árvore: \n")
    arvore.imprimir_arvore(arvore.raiz)

    print("\nTrajeto de Busca da chave ---> {} - Modo Recursivo\n".format(buscar))
    procura = "Sucesso" if arvore.busca_recursiva(buscar) else "Falha"
    print("\nResultado de busca: {0}".format(procura))
```

Resultados:

```
Impressão da árvore:

            133
        132
            130
115
        98
                            95
                    56
            34
                    25
                            25
```

```
Trajeto de Busca da chave ---> 56 - Modo Recursivo

Visitou chave: 115
Visitou chave: 98
Visitou chave: 34
Visitou chave: 56

Resultado de busca: Sucesso
>
```

Figura 7. Apresentação da função de busca em modo recursivo, seguida da execução do teste e seus resultados.

Observe que esse é um método recursivo, no qual corresponde à estrutura da árvore, que também é uma estrutura recursiva, de modo geral, tendo um nó definido como base nele próprio, o qual definimos com a nomenclatura *raiz*. A função aqui definida tem como condição de parada o valor ter sido encontrado. Caso contrário, são realizadas novas chamadas recursivas, para, no fim, retornar `True` como elemento encontrado na árvore e `False` como resposta padrão. Quando isso acontece, significa que chegamos ao fim de um galho da árvore sem ter encontrado a chave, logo, podemos compreender como a não existência da chave na árvore.

Para auxiliar no entendimento realizado nesse tipo de busca, foram utilizados os valores pelos quais a função de busca visitou até o elemento que foi identificado. Podemos executar também vários casos de testes, conforme apresentado na Figura 8, em que, para cada elemento a ser procurado na árvore, são apresentadas as chaves visitadas.

```
if __name__ == "__main__":
    lista = [115, 98, 34, 56, 25, 95, 25, 132,130,133, 89, 21, 10]
    testes = [1, 5, 56, 78, 132, 149, 25]

    arvore = Arvore()
    buscar = 56
    for i in lista:
        arvore.inserir(i)

    print("Impressão da árvore: \n")
    arvore.imprimir_arvore(arvore.raiz)

    # checa se esses elementos perteencem à arvore
    for teste in testes:
        print("\nChaves percorridas: {0}.".format(teste))
        procura = "Sucesso" if arvore.busca_recursiva(teste) else "Falha"
        print("Resultado de busca: {0}".format(procura))
```

```
Impressão da árvore:

          133
     132
          130
115
     98
                              95
                                   89
                    56
          34
                    25
                              25
                                   21
                                        10
```

```
Chaves percorridas: 1.
Visitou chave: 115
Visitou chave: 98
Visitou chave: 34
Visitou chave: 25
Visitou chave: 25
Visitou chave: 21
Visitou chave: 10
Resultado de busca: Falha

Chaves percorridas: 5.
Visitou chave: 115
Visitou chave: 98
Visitou chave: 34
Visitou chave: 25
Visitou chave: 25
Visitou chave: 21
Visitou chave: 10
Resultado de busca: Falha

Chaves percorridas: 56.
Visitou chave: 115
Visitou chave: 98
Visitou chave: 34
Visitou chave: 56
Resultado de busca: Sucesso

Chaves percorridas: 78.
Visitou chave: 115
Visitou chave: 98
Visitou chave: 34
Visitou chave: 56
Visitou chave: 95
Visitou chave: 89
Resultado de busca: Falha

Chaves percorridas: 132.
Visitou chave: 115
Visitou chave: 132
Resultado de busca: Sucesso

Chaves percorridas: 149.
Visitou chave: 115
Visitou chave: 132
Visitou chave: 133
Resultado de busca: Falha

Chaves percorridas: 25.
Visitou chave: 115
Visitou chave: 98
Visitou chave: 34
Visitou chave: 25
Resultado de busca: Sucesso
```

Figura 8. Execução de testes com retorno de sucesso e falha.

Saiba mais

Conforme Piva Junior *et al.* (2014), as operações básicas de uma árvore binária correspondem ao tempo proporcional à sua altura, no qual depende da quantidade N de chaves, e de sua ordem de inserção da árvore. Logo, o tempo de resposta das operações básicas dependerá da quantidade e da distribuição das chaves pelas subárvores com diferentes alturas.

No pior caso, em termos de altura da árvore binária, teríamos as inserções das chaves de forma que a altura da árvore fosse N-1, portanto, tendo seu tempo de execução das operações básicas em O(N) — tempo linear.

Em uma situação ideal de distribuição de N chaves pelas subárvores (árvore binária completa), no entanto, as operações básicas, para o pior caso, seriam executadas em um tempo O(log2 N).

3 Busca iterativa em uma árvore binária

Diferentemente da busca recursiva, a busca iterativa leva em consideração as ramificações de maneira explícita de cada nó para percorrer os seus nodos. Logo, levando em consideração que não há uma chamada interna para a mesma função, é entendível que precisaremos percorrer as suas ramificações de modo mais explícito. A forma como a árvore se apresenta não muda, e, portanto, podemos considerar a árvore apresentada na Figura 6 para essa forma de busca.

A Figura 9 apresenta a abordagem iterativa imprimindo o caminho por onde tem visitado. O `no_atual` é o elemento encarregado por percorrer as ramificações da árvore, o que possibilita acessar os seus níveis.

```
def busca_iterativa(self, chave : int):
  return self._busca_iterativa(self.raiz, chave)

def _busca_iterativa(self, no : No, chave : int):
  if no != None:
    no_atual = no
    while no_atual != None:
      print("Visitou chave: {}".format(no_atual.chave))
      if no_atual.chave == chave:
        return True
      elif chave <= no_atual.chave:
        no_atual = no_atual.esq
      else:
        no_atual = no_atual.dir
```

Figura 9. Função iterativa da árvore.

Testando a busca iterativa

Por fim, vamos rodar a mesma bateria de testes com o modo iterativo, para que possamos avaliar os seus resultados, podendo comparar com os resultados obtidos pelo modo recursivo. A Figura 10 apresenta os resultados.

Execução dos testes:

```
if __name__ == "__main__":
    lista = [115, 98, 34, 56, 25, 95, 25, 132,130,133, 89, 21, 10]
    testes = [1, 5, 56, 78, 132, 149, 25]

    arvore = Arvore()
    buscar = 56
    for i in lista:
        arvore.inserir(i)

    print("Impressão da árvore: \n")
    arvore.imprimir_arvore(arvore.raiz)

    # checa se esses elementos perteencem à arvore
    for teste in testes:
        print("\nChaves percorridas: {0}.".format(teste))
        procura = "Sucesso" if arvore.busca_iterativa(teste) else "Falha"
        print("Resultado de busca: {0}".format(procura))
```

Resultados:

```
Impressão da árvore:

                133
        132
                130
115
        98
                                95
                                        89
                        56
                34
                        25
                                25
                                        21
                                                10

Chaves percorridas: 1.
Visitou chave: 115
Visitou chave: 98
Visitou chave: 34
Visitou chave: 25
Visitou chave: 25
Visitou chave: 21
Visitou chave: 10
Resultado de busca: Falha

Chaves percorridas: 5.
Visitou chave: 115
Visitou chave: 98
Visitou chave: 34
Visitou chave: 25
Visitou chave: 25
Visitou chave: 21
Visitou chave: 10
Resultado de busca: Falha

Chaves percorridas: 56.
Visitou chave: 115
Visitou chave: 98
Visitou chave: 34
Visitou chave: 56
Resultado de busca: Sucesso

Chaves percorridas: 78.
Visitou chave: 115
Visitou chave: 98
Visitou chave: 34
Visitou chave: 56
Visitou chave: 95
Visitou chave: 89

Chaves percorridas: 132.
Visitou chave: 115
Visitou chave: 132
Resultado de busca: Sucesso

Chaves percorridas: 149.
Visitou chave: 115
Visitou chave: 132
Visitou chave: 133
Resultado de busca: Falha

Chaves percorridas: 25.
Visitou chave: 115
Visitou chave: 98
```

Figura 10. Teste da função de busca por iteração.

Segundo Tenenbaum, Langsam e Augenstein (1995), as rotinas de percurso se originam diretamente das definições dos métodos de percurso. Logo, essas definições são em termos de filhos da esquerda e da direita de um nó e não se referem ao pai de um nó. Por esse motivo, ambas as rotinas, recursivas e iterativas, não exigem um campo denominado como pai nem se beneficiam desse campo, mesmo se ele estiver presente.

Referências

CELES, W.; CERQUEIRA, R.; RANGEL, J. L. *Introdução e estrutura de dados*: com técnicas de programação em C. 4. ed. Rio de Janeiro: Campus, 2004.

PIVA JUNIOR, D. *et al. Estrutura de dados e técnicas de programação*. Rio de Janeiro: Elsevier, 2014.

TENENBAUM, A. M.; LANGSAM, Y.; AUGENSTEIN, M. J. *Estruturas de dados em C*. São Paulo: MAKRON Books, 1995.

Fique atento

Os *links* para *sites* da *web* fornecidos neste capítulo foram todos testados, e seu funcionamento foi comprovado no momento da publicação do material. No entanto, a rede é extremamente dinâmica; suas páginas estão constantemente mudando de local e conteúdo. Assim, os editores declaram não ter qualquer responsabilidade sobre qualidade, precisão ou integralidade das informações referidas em tais *links*.

Encerra aqui o trecho do livro disponibilizado para esta Unidade de Aprendizagem. Na Biblioteca Virtual da Instituição, você encontra a obra na íntegra.

Conteúdo:

SAGAH
SOLUÇÕES EDUCACIONAIS INTEGRADAS