ESTRUTURA DE DADOS

Clicéres Mack Dal Bianco

SAGAH
SOLUÇÕES EDUCACIONAIS INTEGRADAS

Árvores: estruturas hierárquicas

Objetivos de aprendizagem

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

- Identificar as características de um TAD do tipo árvore em Python.
- Descrever o armazenamento hierárquico dos elementos em Python.
- Especificar as aplicações que utilizam representações hierárquicas de dados.

Introdução

Uma árvore é considerada uma estrutura de dados não linear, um formato que possibilita organizar os dados e suas ligações de maneira mais produtiva do que o simples antes e depois permitido nas estruturas lineares, como as listas com base em vetores ou em encadeamentos.

As árvores fornecem uma organização natural dos dados, principalmente quando as informações estão dispostas em forma de pirâmide, como em um organograma empresarial, que apresenta uma distribuição hierárquica, com os líderes no topo e os demais funcionários na base.

Diante disso, as árvores tornaram-se estruturas onipresentes em uma ampla faixa de aplicações voltadas para os sistemas de arquivos, interfaces gráficas, bancos de dados e inteligência artificial. Neste capítulo, você aprofundará seus conhecimentos sobre as estruturas de dados do tipo árvores.

Características de um TAD do tipo árvore

As estruturas do tipo árvore recebem esse nome porque fazem analogia às árvores biológicas, mas de cabeça para baixo, conforme mostra a Figura 1a. Serão utilizadas terminologias como raiz e folha para identificar os nós, como o nó raiz, referente à informação que está no topo. De maneira intuitiva,

o exemplo da Figura 1b apresenta uma estrutura para categorização de veículos, em que o nó veículo é a raiz. É interessante observar que a raiz é um nó peculiar, que indica o ponto de partida para as operações de busca, inserção e remoção, não possuindo um nó antecessor.

Um tipo especial de árvores são as binárias, onde cada nó pode ter, no máximo, dois nós diretamente ligados a ele. O conjunto de nós que formam uma árvore pode ser particionado em três subconjuntos distintos, que são: o subconjunto formado somente pelo nó raiz; o subconjunto formado pelos nós da esquerda; e o subconjunto dos nós da direita. Essa divisão pode ser visualizada na Figura 1c.

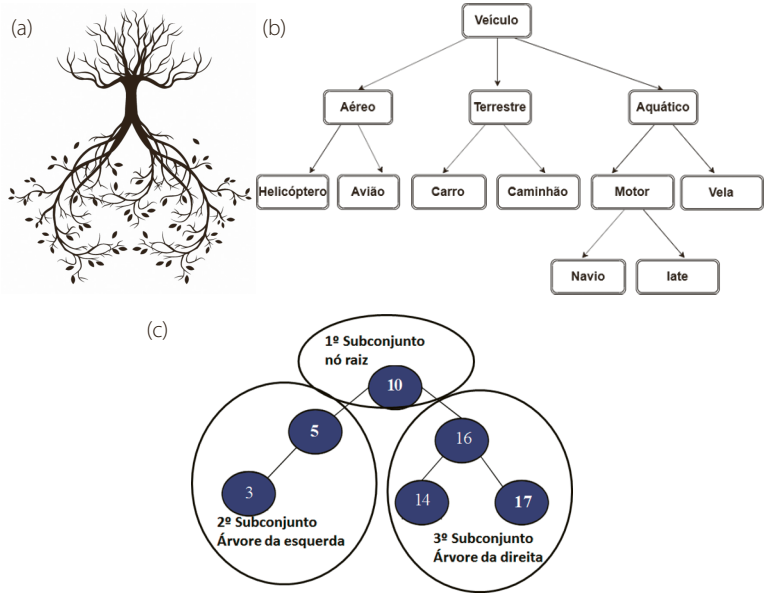


Figura 1. (a) Árvore invertida que demonstra a analogia das estruturas hierárquicas. (b) Aplicação de árvore para categorizar veículos. (c) Exemplo de árvore binária e seus subconjuntos.
Fonte: (a) DMCA/PngLot.com. (b) Edelweiss e Galante (2009, p. 169). (c) Adaptada de Edelweiss e Galante (2009).

Na representação de uma árvore binária, cada nó contém um campo de informação (pode ser do tipo inteiro, *float* ou *string*) e os ponteiros para os filhos. Para a organização geral de uma árvore, utiliza-se um Tipo Abstrato de Dados (TAD), ou seja, um modelo estruturado capaz de descrever um ambiente com número finito de objetos, em que os objetos são associados por meio de determinados relacionamentos. Criação de nós, inserção, remoção e busca de informação são exemplos de operações básicas fornecidas em um TAD do tipo árvore.

Segundo Goodrich e Tamassia (2013), um TAD é uma forma de definir um tipo de dado que, nesse caso, seria o nó, juntamente com as suas operações que manipulam esse tipo de dado. A implementação das operações não precisa ser descrita. Assim, para descrever uma estrutura de dados abstrata, deve-se:

- fornecer o relacionamento entre os objetos que estão sendo armazenados;
- especificar as operações que poderão ser utilizadas para representar relacionamentos de cenários hierárquicos.

Os relacionamentos são as próprias ligações dos nós em determinada árvore. No exemplo da Figura 1C, o nó 5 está ligado com o nó 3 (seu descendente). Um ponto importante dessa relação é que o nó descendente só pode ser acessado pelos seus nós ancestrais. Árvores binárias são, em geral, implementadas como uma estrutura dinâmica, ou seja, usa-se alocação de memória dinamicamente por meio de ponteiros, e o TAD possibilitará os encadeamentos. Os encadeamentos, no caso de um nó, representarão os relacionamentos ou os seus filhos.

Observe o código a seguir, em que o nó de uma árvore é composto por três informações, o `label`, que pode ser um nome ou número, o apontador esquerdo e o apontador direito, que indicam os relacionamentos, para a esquerda (`left`) e para a direita (`right`). A classe nó é inicializada com o atributo `label`, e os apontadores para esquerda e direta do nó são inicializados como vazios.

```
Class No:
    def __init__(self, label):
        self.label = label
        self.left = None
        self.right = None
```

De acordo com Goodrich, Tamassia e Goldwasser (2013), em Python, todos os tipos de dados (como `int`, `float` e `str`) são classes, então, definir uma classe é criar um novo tipo de dado ou um TAD, cujo comportamento lógico

é dado por um conjunto de operações. A seguir, observa-se a classe de uma árvore. O código apresenta os protótipos das principais operações. Todos os métodos comporão o TAD `BinaryTree`. A raiz (`root`) é inicializada como vazia. A operação inserção (`insert`) recebe como parâmetro o dado a ser armazenado e, nessa operação, localiza-se a posição que a informação deve ser inserida seguindo os critérios de armazenamento de uma árvore binária.

```
class BinaryTree:

def _ _ init _ _ (self):
self.root = None

# insere informação
def insert(self, number):

# verifica se árvore está vazia
def empty(self):

# mostra em pré-ordem
def show _ pre(self, curr _ node):

 # mostra em-ordem
def show _ em(self, curr _ node):

# mostra em-pós-ordem
def show _ pos(self, curr _ node):

#remove nó
def remove(self, number):

# busca informação
def search(self, number):

#recebe o endereço do nó raiz
def getRoot(self):

#busca o maior valor
def bigger(self):
```

A árvore pode ser percorrida de três formas, em pré-ordem, em ordem e em pós-ordem. Os protótipos desses percursos são `show_pre`, `show_em` e `show_pos`. Já a operação de remoção (*remove*) recebe a informação a ser removida, busca a sua localização e, após a efetivação da remoção, os encadeamentos serão atualizados. Observe que, nesse TAD, foram incluídas duas operações adicionais: `getRoot` e `bigger`. A primeira retorna o endereço de memória da raiz, e a segunda localiza o maior valor da árvore.

Já o protótipo de busca (`search`) recebe como parâmetro o dado a ser localizado que, nesse caso, é um número (`number`). A busca inicializa verificando se a árvore não está vazia. Para evitar qualquer alteração indesejada na raiz, seu endereço é passado para uma estrutura auxiliar (`curr_node`). Caso a árvore não esteja vazia, um conjunto de instruções é repetido enquanto não localizar o número ou uma subárvore vazia.

Segundo Baka (2017), a busca é como um processo de decisão que realiza perguntas a fim de identificar qual direção prosseguir. Inicia-se perguntando se o valor que está no nó (`curr_node.getLabel()`) é igual ao número. Se for igual, a busca termina com sucesso. Caso contrário, uma segunda pergunta é feita,

- o número é menor que o valor do nó?

Se a resposta for sim, a busca continuará na subárvore à esquerda, se a resposta for "maior que", a busca continuará na subárvore à direita. Por fim, se chegarmos a uma subárvore vazia, a pesquisa termina sem êxito.

```
def search(self, number):
    # verifica se a árvore está vazia
    if self.empty():
        print('Arvore está vazia!')
    else:

        # árvore não está vazia, busca número
        curr _ node = self.root

            while True:
                if curr _ node.getLabel() == number:
                    print('O dado foi encontrado na árvore ')
                    break;

                # verifica se a busca continua para a esq. ou
a dir.
```

```
            if number< curr _ node.getLabel():

                curr _ node = curr _ node.getLeft() #continua
a esquerda

            else:
                curr _ node = curr _ node.getRight() # continua
a direita
```

Fique atento

Os TADs das árvores encapsulam as operações básicas como inserção, busca, remoção e percursos. Além das operações básicas, você pode incluir outros métodos, como os baseados nos relacionamentos dos nós, por exemplo, encontrar o pai de determinado nó, retornar aos nós antecessores e, ainda, localizar o maior valor e retornar à altura da árvore.

Armazenamento hierárquico dos elementos

As propriedades das árvores impactam no armazenamento das informações. As principais propriedades que envolvem uma árvore são grau, nível, altura, nós folhas, nó pai, nó filho e os critérios de armazenamento. O grau de um nó representa o seu número de subárvores. Por sua vez, o grau de uma árvore corresponde ao número máximo de graus de todos os seus nós. Uma árvore binária tem grau máximo igual a 2. A Figura 2a traz os nós e seus respectivos graus.

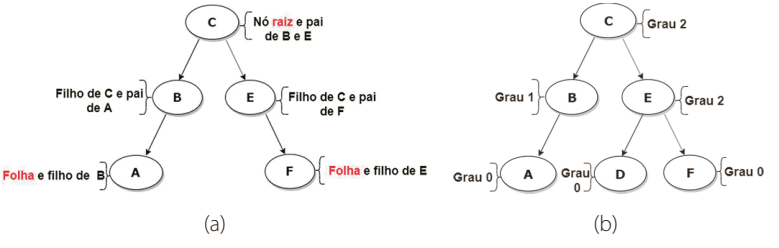


Figura 2. (a) Graus dos nós de uma árvore. (b) Nó raiz e nós folhas.

O nó denominado pai refere-se ao nó que está no topo e com ligação direta a outro nó. Logo, o nó filho é o nó abaixo e, finalmente, o nó folha é o nó que não possui filhos (Figura 2b). Diante disso, pode-se definir os níveis de uma árvore. O nível é a distância do nó raiz, então, o nível do nó raiz é zero. Dessa forma, é possível definir que a altura é o nível mais distante da raiz, conforme poder ser observado na Figura 3.

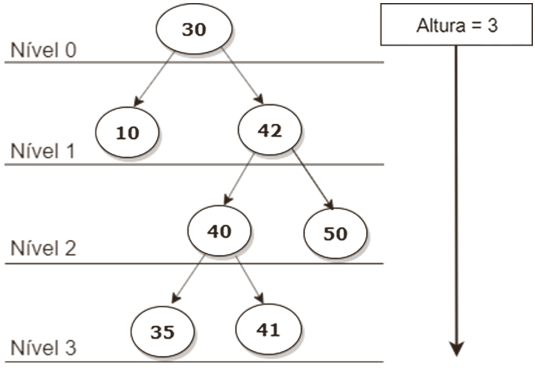


Figura 3. Níveis e altura de uma árvore.
Fonte: Adaptada de Ascencio e Araujo (2011).

As propriedades a seguir são importantes para definir os relacionamentos durante o armazenamento. Segundo Cormen *et al*. (2012), o armazenamento deve respeitar os seguintes critérios:

- todos os nós de uma subárvore da direita são maiores do que o nó raiz;
- todos os nós de uma subárvore da esquerda são menores do que o nó raiz;
- cada subárvore também é conhecida como árvore.

Armazenamento em uma árvore

O processo de armazenamento de informações em uma estrutura do tipo árvore é similar ao método de busca em que, inicialmente, é verificado se a árvore está vazia e, em caso afirmativo, o nó é definido como raiz. Caso contrário, compara-se o elemento com a raiz: se for menor, segue para a subárvore da esquerda e, se for maior, segue para a subárvore da direita. Esse processo é realizado até encontrar uma posição livre.

Veja o exemplo para o armazenamento dos valores 37, 20, 80 apresentado na Figura 4. Considere que a árvore está vazia, então, na primeira inserção, o nó 37 será alocado como raiz. O próximo número que será armazenado é o 20 e, dessa vez, a raiz não está vazia. Então, é testado se esse número é menor do que o valor que está na raiz. Em caso afirmativo, busca-se uma posição livre à esquerda. Nesse exemplo, como a subárvore à esquerda do nó 37 está vazia, encontrou-se a posição que referenciará o nó 20. Agora, será a vez de armazenar o número 80, que é maior do que 37, e esse nó está com a sua subárvore da direita vazia, então, esse é o endereço que referenciará o nó 80.

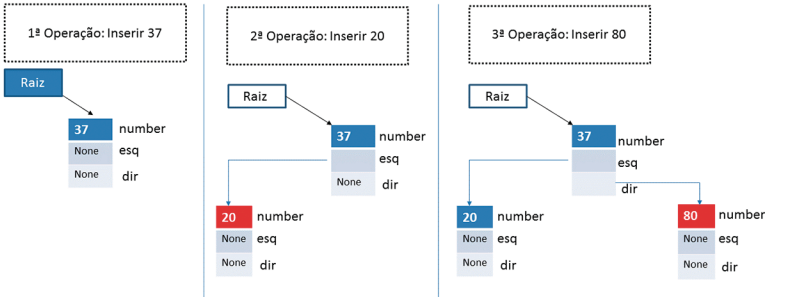


Figura 4. Exemplo de inserção em uma árvore.

Fonte: Adaptada de Ascencio e Araujo (2011).

Caso fosse necessário inserir o valor 72 na árvore anterior, a inserção seguiria a mesma lógica detalhada antes. A árvore será acessada pela raiz, verifica-se a direção a seguir, como o número (72) é maior do que o atual valor da raiz (37), e a posição da direita está alocada, o número será comparado com 80 e alocado à esquerda deste, conforme mostra a Figura 5. A cada inserção os encadeamentos são atualizados.

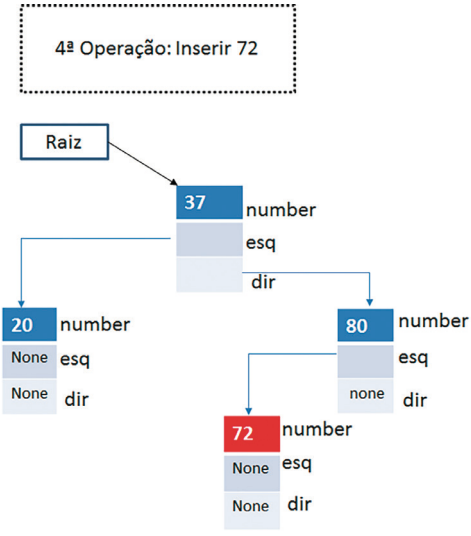


Figura 5. Operação de inserção do número 72.
Fonte: Adaptada de Ascencio e Araujo (2011).

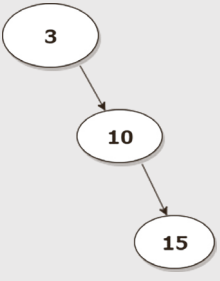
A implementação da inserção é similar à da busca. O método recebe o número, que é inicializado como um nó. Após, é necessário verificar se a árvore está vazia, caso contrário, entra-se no laço de repetição. Os primeiros testes no laço de repetição determinam se a tentativa de inserção ocorrerá na subárvore da esquerda ou da direita. Esses testes são feitos até encontrar um endereço livre, ou seja, um pai que referenciará o nó.

```
def insert(self, number):
        # cria um novo nó
        node = Node(number)
        # verifica se a árvore está vazia
        if self.empty():
            self.root = node
        else:
            # árvore não vazia,
            curr _ node = self.root
            dad _ node = None
            while True:
                if curr _ node != None:
```

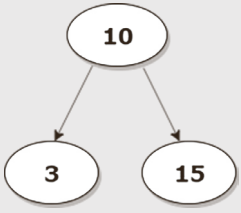
```
                # verifica se vai para esquerda ou direita
                if node.getLabel() < curr _ node.getLabel():
                    curr _ node = curr _ node.getLeft() # pros-
segue a esquerda
                else:
                    curr _ node = curr _ node.getRight() # pros-
segue a direita
            else:
                # se curr _ node é None, então encontrou onde
inserir
                if node.getLabel() < dad _ node.getLabel():
                    dad _ node.setLeft(node) #insere a esquerda
do nó pai
                else:
                    dad _ node.setRight(node) #insere a direita
do nó pai
                break # sai do loop
```

Fique atento

Nesse tipo de árvore, a ordem de inserção das informações determinará os encadeamentos, por exemplo, a inserção da sequência de valores 3,10,15, resultará na árvore a seguir:



Agora, se você inserir os valores em outra ordem, os relacionamentos se alteram. Para a inserção na sequência 10,3,15, o resultado é apresentado a seguir:



Observe que o primeiro valor será a raiz, e os demais relacionamentos são definidos respeitando as propriedades, maiores valores à direita e menores valores à esquerda.

Remoção em uma árvore

O TAD da árvore binária também possibilita a remoção de nós. Para a remoção de um nó da árvore, existem três situações a considerar: se o nó não tem filhos, ou seja, é um nó folha, ele poderá ser removido sem ajustes posteriores na árvore; se o nó possui apenas um filho, então esse filho passará a ocupar a posição do nó removido; e se o nó possui dois filhos, então encontra-se o sucessor desse nó, que deve estar na subárvore à direita do nó a ser removido. O caso mais simples é a remoção de um nó folha, bastando alterar o ponteiro do pai para nulo. A Figura 6 demonstra a remoção do nó 7 e a árvore final após a remoção.

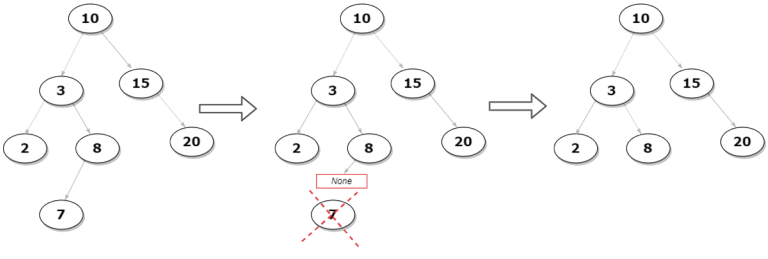


Figura 6. Operação de remoção do nó 7.

Fonte: Adaptada de Tenenbaum, Langsam e Augenstein (2010).

O segundo caso trata da remoção de um nó que possui um filho (à esquerda ou à direita). A Figura 7 demonstra a remoção do nó 8, que possui um filho à esquerda. É possível perceber que um dos ponteiros do pai deve ser atualizado, no exemplo, o ponteiro à direita do pai (o nó 3) recebeu o nó 7 (filho do nó removido).

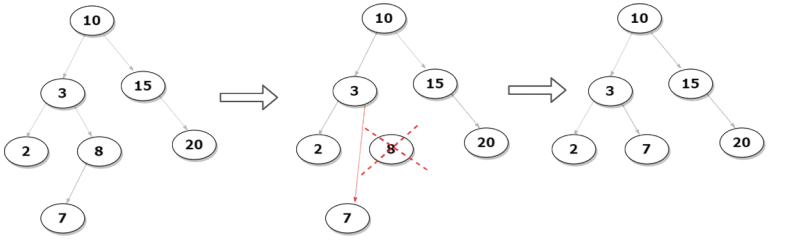


Figura 7. Operação de remoção do nó 8.
Fonte: Adaptada de Tenenbaum, Langsam e Augenstein (2010).

Considerando o terceiro caso, a remoção trata um nó que possui filhos à esquerda e à direita. A Figura 8 apresenta a remoção do nó 3. Esse nó possui duas subárvores que devem ser referenciadas por outro nó, ou seja, o nó sucessor. O sucessor será o nó que está mais à esquerda (ou de menor valor) na subárvore à direita do nó que será removido. A subárvore à direita do nó 3 apresenta o nó 7 mais à esquerda. O nó sucessor apontará para os filhos do nó 3.

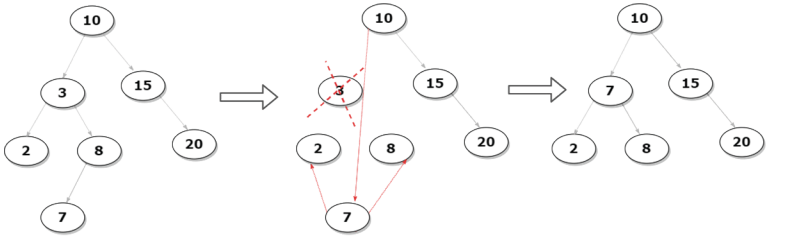


Figura 8. Remoção do nó 3 com atualização dos ponteiros do nó sucessor (nó 7). O ponteiro do nó pai também é atualizado com o endereço do nó sucessor.
Fonte: Adaptada de Tenenbaum, Langsam e Augenstein (2010).

A classe usada para a remoção recebe como parâmetro um número a ser removido. O algoritmo inicia verificando se a árvore não está vazia. A árvore não estando vazia, entra-se no laço de repetição para localizar a posição do número. Compara-se o valor do nó atual (`curr_node`) com o número e, se for menor, a busca continua à esquerda. Ao localizar a posição, o nó atual é atualizado com o endereço dessa posição (`curr_node.getLeft()`). Caso contrário, a busca continua à direita. Nesse caso, o nó atual recebe o endereço da subárvore à direita (`curr_node.getRight()`).

```
def remover(self, number):
    if self.root == None:
       return False # se arvore vazia
    curr _ node = self.root
    dad = self.root
    child _ left = True
    # Busca a posição do número
     while curr _ node.getLabel() != number: # enquanto não
encontrou
       dad = curr _ node
     if number < curr _ node.getLabel(): # caminha para esquerda
          curr _ node = curr _ node.getLeft()
          child _ left = True # é filho a esquerda? sim
       else: # caminha para direita
          curr _ node = curr _ node. getRight ()
          child _ left = False # é filho a esquerda? NAO
       if curr _ node == None:
         return False # encontrou uma folha, então sai do laço
```

- # se chegou aqui, significa que encontrou o número (`number`);
- # "`curr_node`": contém a referência ao nó a ser eliminado;
- # "`dad`": contém a referência para o pai do nó a ser eliminado;
- # "`child_left`": é verdadeira se o nó atual é filho à esquerda do pai.

Em relação ao nó a ser removido, a variável *booleana* (`child_left`) será verdadeira, caso esse nó seja filho à esquerda, ou falsa, se o nó for filho à direita. Essa variável será utilizada para atualizar o endereço do nó pai. Após localizar a posição do nó, a próxima etapa do método de remoção será identificar qual dos três casos enquadra o nó a ser removido e processar as atualizações dos ponteiros.

Supondo que o número a ser removido seja encontrado na árvore, os próximos testes condicionais identificarão qual é a situação que condiz com esse nó. A seguir, veremos como pode ser implementado cada um dos casos.

Primeiro caso

Vamos iniciar com o caso mais simples, em que o nó a ser removido é uma folha. Observe o código a seguir, a primeira condição retorna se esse nó é a raiz e, em caso afirmativo, o endereço da raiz recebe nulo. Caso o nó a ser removido seja um filho à esquerda (`child_left`), então o ponteiro à esquerda do pai recebe nulo (`None`). Não sendo um filho à esquerda, então significa que ele está à direita, e o ponteiro do pai à direita recebe nulo.

```
 # Caso 1;
 # Se não possui nenhum filho (é uma folha), elimine-o
    if curr _ node.getLeft() == None and curr _ node.getRight()
== None:
       if curr _ node == self.root:
          self.root = None # se raiz
       else:
          if child _ left:
            dad.left = None # se for filho a esquerda do pai
          else:
            dad.right = None # se for filho a direita do pai
```

Segundo caso

No segundo caso, o nó a ser removido tem um filho à esquerda ou à direita. Inicialmente, é avaliado se o filho está **à esquerda** (`curr_node.Left`). Se for o caso, é necessário guardar o endereço desse filho, ou seja, o nó que ficará órfão. Depois, testa-se o nó a ser removido que está alocado à esquerda do pai (`child_Left`), então, o ponteiro à esquerda do nó pai (`dad.Left`) receberá o endereço do nó órfão.

Caso o filho, ou seja, o nó órfão, esteja **à direita** (`curr_node.Right`), o próximo teste será para verificar se o nó a ser removido é um filho da esquerda ou (`dad. Right`) da direita. Se for da direita, o ponteiro à direita do nó pai referenciará o nó órfão. Se for da esquerda, o endereço à esquerda do pai (`dad.Left`) referenciará o nó órfão.

```
 # Caso 2
# Se o nó a ser removido possui um filho a esquerda
    if curr _ node.Left != None:
          if child _ Left:
             dad.Left = curr _ node.Left # se for filho a es-
querda do pai
          else:
            dad.Right = curr _ node.Left # se for filho a di-
reita do pai
    # Se o nó a ser removido possui um filho a direita
    elif curr _ node.Right != None:
          if child _ Left:
            dad.Left = curr _ node.Right # se for filho a es-
querda do pai
          else:
              dad.Right = curr _ node.Right # se for filho a
direita do pai
```

Terceiro caso

O último caso remove o nó que possui subárvore à esquerda e subárvore à direita. Nessa situação, o primeiro procedimento é encontrar o nó sucessor, ou seja, o nó que ocupará a posição do nó que será removido. O sucessor deve ser o menor valor da subárvore à direita. Se o nó a ser removido é um filho à esquerda (`child_Left`), então o endereço à esquerda do pai apontará para o sucessor. Se o nó a ser removido é um filho à direita, o endereço à direita do pai apontará para o sucessor. Por fim, o ponteiro à esquerda do nó sucessor é atualizado com o filho à esquerda do nó a ser removido (`successor.Left = curr_node.Left`). Um exemplo é o caso do nó 7 apontando para o nó 2, conforme mostra a Figura 8.

```
 # Caso 3
# Se possui mais de um filho, se for um avô ou outro grau
maior de parentesco
       successor = self.noSuccessor(curr _ node)
        # Usando sucessor que seria o Nó mais à esquerda da
subárvore a direita
          if child _ Left:
          dad.Left = sucessor # se for filho a esquerda do pai
          else:
          dad.Right = sucessor # se for filho a direita do pai
      successor.Left = curr _ node.Left # acertando o ponteiro
a esquerda do sucessor
                   # agora que ele assumiu a posição correta
na árvore
    return True
```

Além de retornar o nó com menor valor da subárvore à direita, o método que retornará o nó sucessor (`NoSuccessor`) atualizará os ponteiros dessa subárvore. O método a seguir recebe como parâmetro o nó a ser removido (`noRemove`). São inicializados outros dois nós, o `dadSuccessor` e o `successor`. Ao finalizar o laço de repetição (`while`), o nó sucessor (`successor`) será o nó mais à esquerda da subárvore à direita, o `dadSuccessor` será o pai do nó sucessor, e o `noRemove` será o nó que deverá ser excluído.

```
# Método Sucessor
  def noSuccessor(self, noRemove): # O parâmetro é a referência
para o Nó que deseja-se remover
     dadSuccessor = noRemove
     successor = noRemove
      curr _ node = noRemove.Right # vai para a subárvore a
direita
      while curr _ node!= None: # enquanto não chegar no Nó
mais à esquerda
       dadSuccessor = sucessor
       successor = curr _ node
      curr _ node = curr _ node.Left # caminha para a esquerda
     if successor != no _ remove.Right: # se sucessor não é o
filho a direita do Nó a ser eliminado
```

```
      dadSuccessor.Left= successor.Right # pai herda os filhos
do sucessor
       successor.Right = no _ remove.Right # guardando a refe-
rencia a direita do sucessor
     return successor
```

Link

A ferramenta Visualgo realiza a inserção de maneira interativa em uma árvore e pode ser acessada no endereço a seguir.

https://visualgo.net

Aplicações que utilizam árvores

Em diversas situações, são necessárias estruturas mais complexas do que as puramente sequenciais. As árvores são largamente empregadas nas mais diversas áreas da computação como banco de dados, sistemas operacionais, compiladores, redes de computadores, entre outras. A seguir, estão descritos alguns exemplos de aplicações.

Árvore genealógica

Imagine uma árvore genealógica que fornece os relacionamentos entre gerações: avós, pais, filhos, irmãos. Essa organização é usada pelos profissionais de Direito, pois dela decorrem os deveres e os direitos dos familiares. De modo geral, organiza-se árvores genealógicas hierarquicamente. No exemplo da Figura 9, o nó raiz está identificado como "eu". Uma operação necessária seria localizar o grau de parentesco dos nós. Nessa situação, os níveis da árvore são um termo importante, pois, a partir do nível, é possível criar um método para retornar o grau de parentesco entre nós. Nesse caso, quanto maior o nível, mais distante é o parentesco.

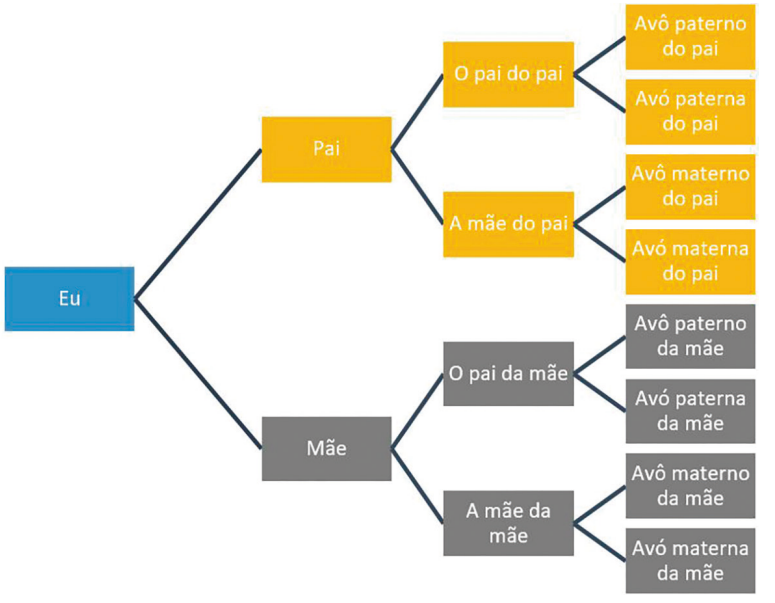


Figura 9. Exemplo de árvore genealógica organizada hierarquicamente.

Estrutura organizacional de uma empresa

A Figura 10 apresenta uma estrutura organizacional de uma empresa. A partir de determinado gerente, seria fácil listar todos os operadores (Op.) que são subordinados a ele. Também é possível verificar os gerentes que estão diretamente relacionados à determinado diretor. A subárvore da esquerda pode ser categorizada com os funcionários responsáveis pelas vendas, e a subárvore da direita, com os funcionários responsáveis pelo *marketing*, agilizando, assim, a localização das equipes.

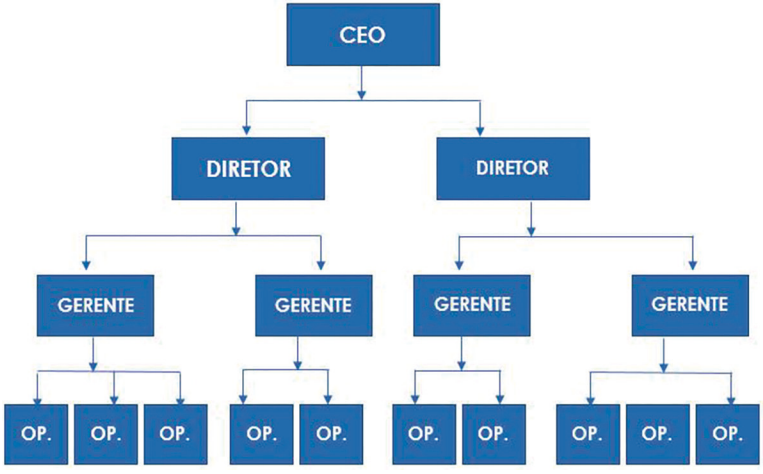


Figura 10. Estrutura simples de uma organização empresarial.
Fonte: Veyrat (2017, documento *on-line*).

Documento HTML

A linguagem HTML é a estrutura básica das páginas *web*, e essa linguagem exige que algumas *tags* básicas estejam presentes, como `html`, `head`, `body`, dentro da estrutura `head`. Por exemplo, existem comandos para a formatação e inserção de conteúdo, e o programador deve respeitar essa organização. Desse modo, é possível analisar se as *tags* do código estão relacionadas corretamente. A Figura 11 apresenta alguns elementos da linguagem HTML.

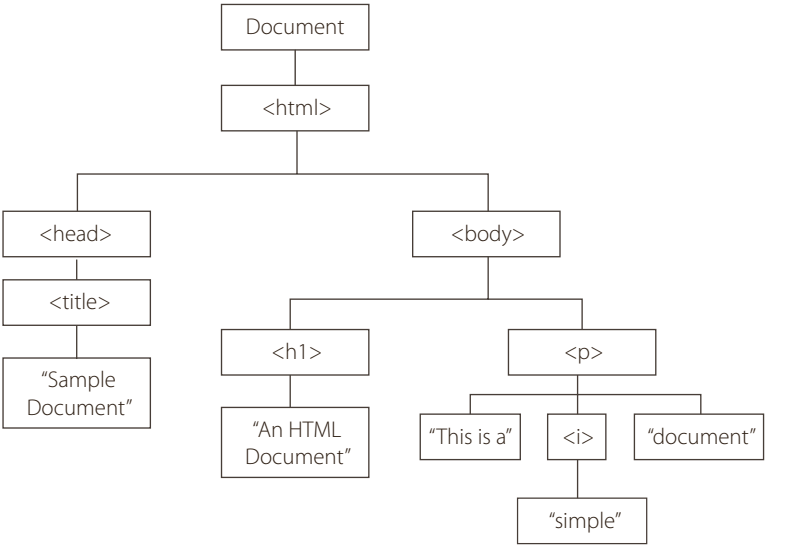


Figura 11. Modelo de documento HTML.
Fonte: Adaptada de Devfuria (2012).

Cada nível da árvore corresponde a um nível de aninhamento interno nas *tags*. A primeira *tag* é `<html>`, todas as demais estão contidas nessa *tag*, e essa mesma propriedade de aninhamento hierárquico vale para as demais *tags*. Obviamente, existem outras *tags*, como para inserção de tabelas `<table>`, inclusão de *links* `<a link>` e marcadores `<li>`. A intenção é demonstrar a aplicabilidade usando algumas *tags* como exemplo.

Organização de diretórios

Outro exemplo prático de uso é a organização de um diretório de determinado sistema operacional. Seja ele *desktop* ou *mobile*, os arquivos estão organizados de forma hierárquica, conforme demonstra a Figura 12. Perceba que é fácil determinar o caminho de um arquivo ou até mesmo verificar rapidamente a existência de um arquivo.

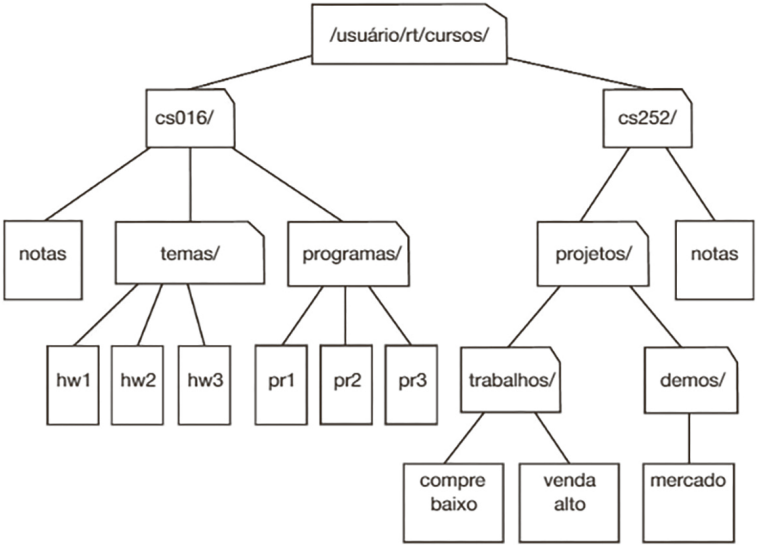


Figura 12. Organização de diretório em sistema operacional.
Fonte: Goodrich e Tamassia (2013, p. 286).

Este capítulo demonstrou o emprego de árvores em casos práticos, mas os exemplos não foram esgotados. Ao realizar uma busca sobre uso de árvores na área específica de computação gráfica, exemplos dessa estrutura aplicada a jogos, para determinar os próximos movimentos dos personagens, serão encontrados. Ao pesquisar pelo uso de árvores na área de mineração de dados, aplicações para identificar quando uma transação bancária pode ser uma fraude serão encontradas, entre outras. Assim, percebe-se a abrangência desse tema e a importância de saber codificar essa estrutura para a projeção de soluções, principalmente, na representação de situações multi ou bidirecionais, facilitando a própria implementação e agilizando a consulta de informações.

Referências

ASCENCIO, A. F. G.; ARAUJO, G. S. *Estrutura de dados*: algoritmos, análise de complexidade e implementações em Java e C/C++. São Paulo: Person, 2011.

BAKA, B. *Python data structures and algorithms*. Birmingham, UK: Packt, 2017.

CORMEN, T. H. *et al. Algoritmos*: teoria e prática. 3. ed. Rio de Janeiro: Elsevier, 2012.

DEVFURIA. *DOM — Document Object Model*: O que você precisa saber sobre o DOM (Document Object Model). 2012. Disponível em: http://www.devfuria.com.br/javascript/dom/. Acesso em: 21 dez. 2019.

EDELWEISS N.; GALANTE, R. *Estrutura de dados*. Porto Alegre: Bookman, 2009.

GOODRICH, M. T.; TAMASSIA, R. *Estruturas de dados e algoritmos em Java*. 5. ed. Porto Alegre: Bookman, 2013.

GOODRICH, M. T.; TAMASSIA, R.; GOLDWASSER M. H. *Data structures and algorithms in Python*. Hoboken, NJ: Wiley, 2013.

TENENBAUM A. M.; LANGSAM Y., AUGENSTEIN M. J. *Estruturas de dados usando C*. São Paulo: Person, 2010.

VEYRAT, P. *Exemplo de estrutura organizacional de uma empresa*: qual escolher? 24 maio 2017. Disponível em: https://www.heflo.com/pt-br/rh/exemplo-de-estrutura-organizacional-de-uma-empresa/. Acesso em: 21 dez. 2019.

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