Python Help

PUBLICADO POR

VALDIR STUMM JR
POSTADO NO

19 DE JANEIRO DE 2015
PUBLICADO EM

ALGORITMOS, ESTRUTURAS
COMENTÁRIOS

10 COMENTÁRIOS

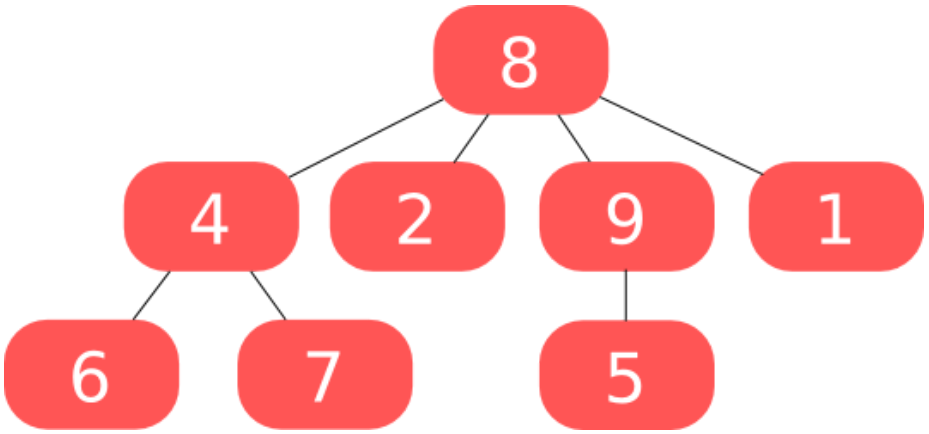
Árvore Binária de Busca em Python

Onde quer que você esteja lendo este texto, é bem provável que existam algumas **árvores** instanciadas na memória do seu computador. **Árvores** são estruturas de dados muito versáteis que são utilizadas na solução de uma enorme gama de problemas, como na otimização de consultas e na indexação de bancos de dados, na geração de códigos para compressão de dados, na análise sintática de código por compiladores, na representação da estrutura de diretórios em um sistema de arquivos, etc. O conceito será explorado neste post, onde veremos a implementação de uma das variações dessa estrutura, a **Árvore Binária de Busca**.

As Árvores

As árvores são estruturas de dados hierárquicas nas quais os dados são armazenados por **nós**, sendo o primeiro destes chamado de **nó raiz**. Cada nó de uma árvore possui um **nó pai** (exceto o nó raiz) e possui **nós filhos** (exceto os *nós folha*). Veja na figura

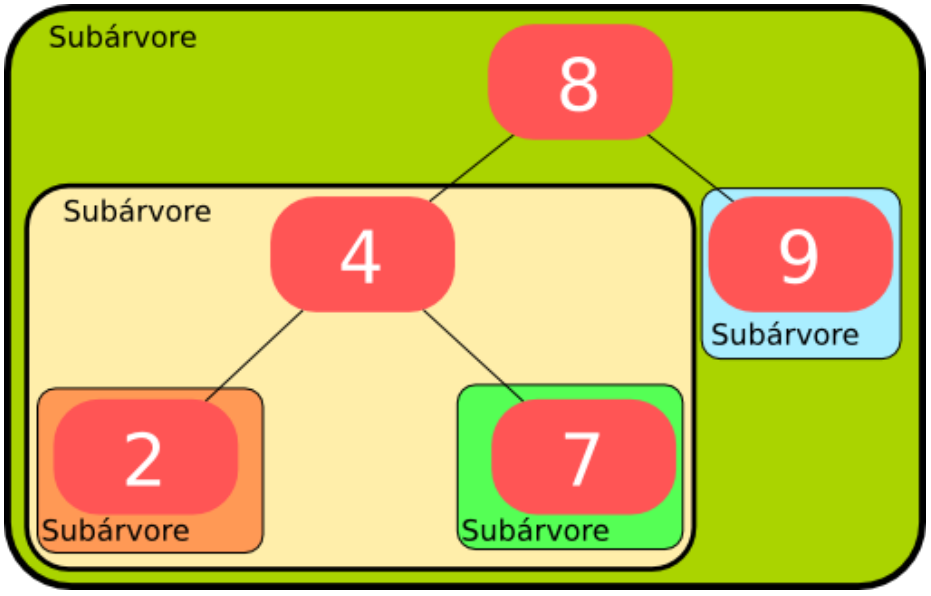
abaixo um exemplo de **árvore** que guarda números inteiros em seus nós.



(h◎ps://pythonhelp.files.wordpress.com/2015/01/image06.png)

Na figura acima, o **nó raiz** é o 8, e ele tem como filhos os nós de chave 4, 2, 9 e 1. O nó 4 é pai dos nós 6 e 7. Estes dois, assim como os nós de chave 5, 2 e 1, são chamados de **folhas** por não terem filhos (ou seja, seus filhos são nulos).

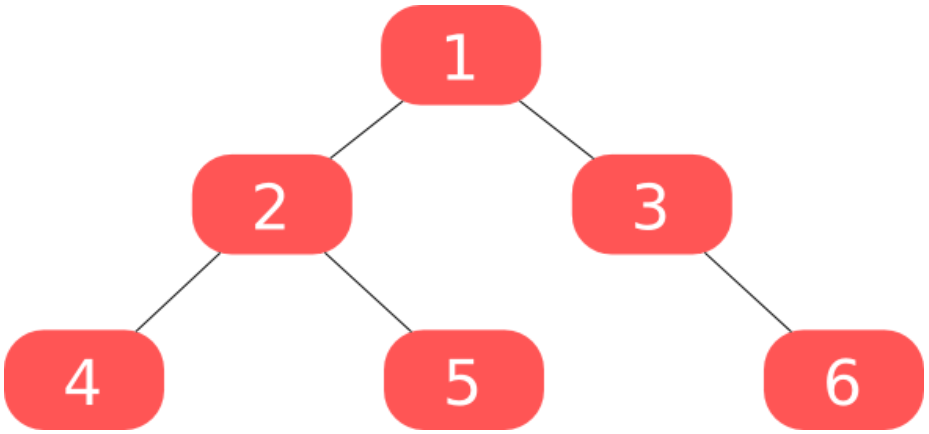
Uma árvore pode ser composta por uma ou mais **subárvores**. A figura abaixo destaca as subárvores que compõem uma outra árvore.



(h◎ps://pythonhelp.files.wordpress.com/2015/01/image08.png)

Árvores Binárias

Uma **árvores binária** é um tipo especial de árvore, em que cada nó pode ter **no máximo 2 filhos**, um à esquerda e um à direita do nó.



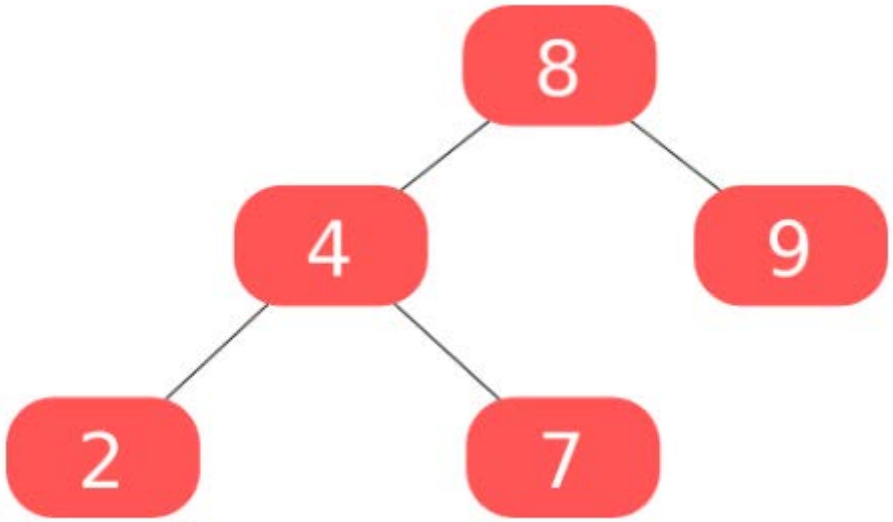
(h◎ps://pythonhelp.files.wordpress.com/2015/01/image10.png)

A árvore da figura acima é, de fato, uma árvore binária, pois nenhum nó possui mais do que dois filhos. Observe que um dos nós possui somente um filho e outros nós nem sequer possuem filhos, o que também está de acordo com as propriedades de uma árvore binária.

Árvores Binárias de Busca

As **Árvores Binárias de Busca** são árvores binárias com a seguinte propriedade: todos os nós pertencentes à subárvore esquerda de qualquer nó possuem chave menor que a chave do mesmo, e em que os nós da subárvore à sua direita possuem chave maior que a chave do nó em questão. **Essa propriedade deve ser válida para todas as subárvores**, possibilitando a realização de buscas mais eficientes, pois podemos comparar a chave procurada com a chave de um nó e decidir se devemos continuar a busca somente na subárvore à esquerda ou à direita do nó, reduzindo assim a quantidade de nós a serem visitados na busca.

Vamos agora observar a árvore da figura abaixo e verificar se a propriedade acima é realmente válida para todos os nós dela.



(h◎ps://pythonhelp.files.wordpress.com/2015/01/image14.png)

Todos os elementos contidos na subárvore à esquerda do nó 8 (nós 2, 4, e 7) possuem chaves menores que ele e todos os elementos da subárvore à direita dele (nó 9) são maiores. A subárvore iniciada pelo nó de chave 4 também respeita tais propriedades pois os elementos à esquerda dele são menores (no caso o nó de chave 2) e os que estão à direita são maiores (o nó 7). Desse modo, podemos afirmar que a árvore da figura acima é uma árvore binária de busca.

Implementação de uma Árvore Binária de Busca

Uma árvore nada mais é do que um conjunto de **nós**, e cada nó é um objeto com uma chave, um valor e uma referência aos seus dois filhos (esquerdo e direito). A chave serve para identificar o nó e o valor armazena os dados que o nó representa. Por exemplo, em um sistema de arquivos que utiliza uma árvore para representação da hierarquia de diretórios, a **chave** do nó poderia ser o nome do arquivo e o **valor** poderia ser uma referência ao conteúdo do arquivo em si.

Em Python, podemos definir um **nó** (BSTNode – de *Binary Search Tree Node*) de árvore da seguinte forma:

```
1  class BSTNode(object):
2      def __init__(self, key, value=None, left=None, right=None):
3          self.key = key
4          self.value = value
5          self.left = left
6          self.right = right
```

Os campos `left` e `right` são as referências a outros nós, o campo `key` guarda a chave utilizada para identificar o nó e `value` representa o valor que desejamos armazenar nele.

A construção de um **nó**, com chave 42, sem valor armazenado e sem filhos *pode ser feita* da seguinte forma:

```
1 | root = BSTNode(42)
```

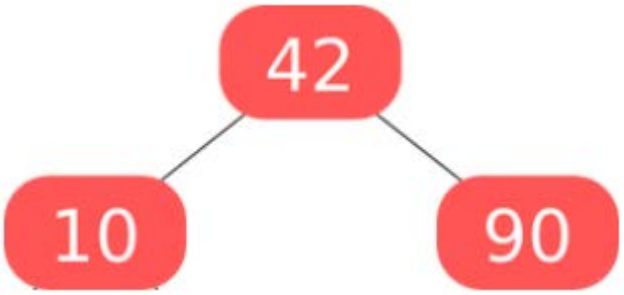
Esse código cria a seguinte árvore:

(h◎ps://pythonhelp.files.wordpress.com/2015/01/image15.png)

Se quisermos adicionar filhos ao nó 42, podemos fazer:

```
1 | root.left = BSTNode(10)
2 | root.right = BSTNode(90)
```

O que faz com que a árvore fique:

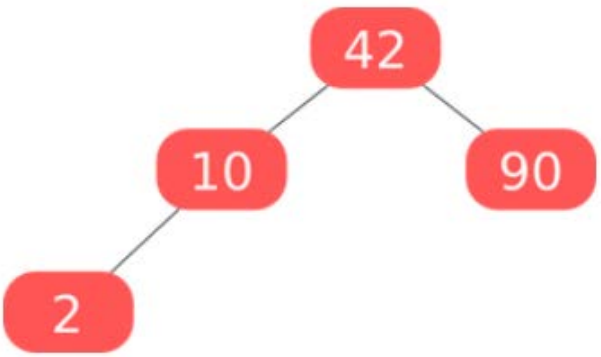


(h◎ps://pythonhelp.files.wordpress.com/2015/01/image05.png)

Se quisermos adicionar um filho esquerdo ao nó de valor 10, recém criado, podemos fazer:

```
1 | root.left.left = BSTNode(2)
```

O encadeamento feito acima gera a seguinte árvore:



Embora funcione, nossa implementação de árvore não possui uma boa **interface de programação**, pois é necessário fazer os encadeamentos entre nós de forma manual.

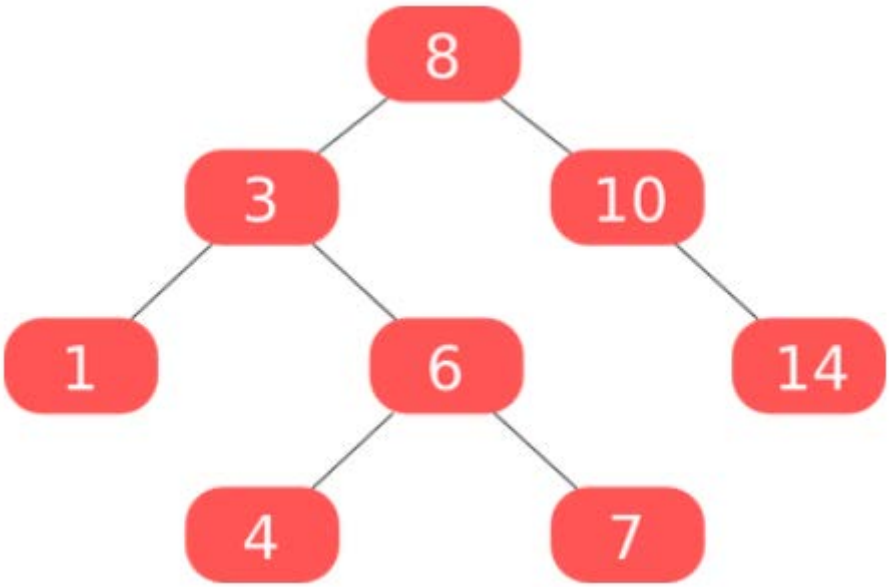
Operações em uma Árvore Binária de Busca

Para que seja útil, a interface de programação de nossa árvore binária de busca deve fornecer algumas operações básicas, como: busca por chave, inserção de um elemento na árvore, remoção de um elemento e travessia.

A seguir veremos a implementação dessas operações, definindo uma interface melhor para nossa classe.

Busca em uma Árvore Binária de Busca

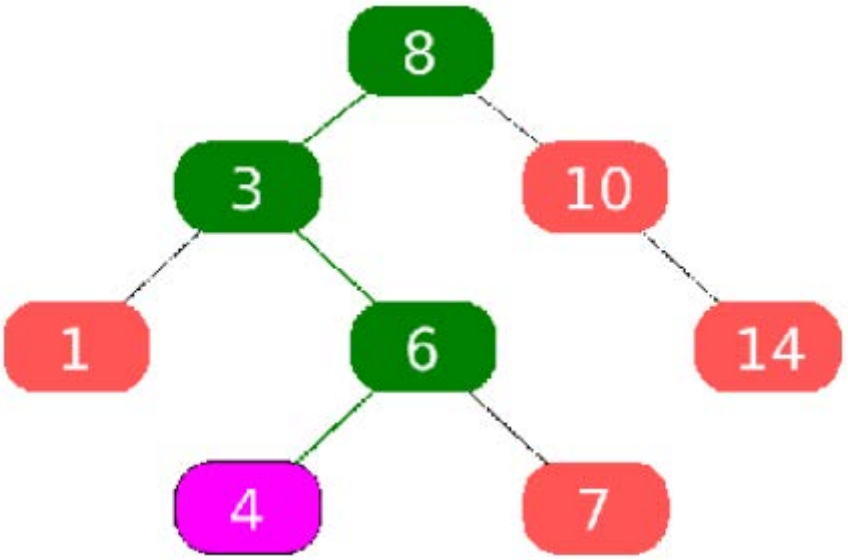
As Árvores Binárias de Busca são assim chamadas porque suas estruturas permitem a realização de buscas de forma eficiente. Vamos entender o porquê disso observando a árvore da figura abaixo:



(https://pythonhelp.files.wordpress.com/2015/01/image03.png)

Se estivermos procurando pelo nó de chave 10, tudo o que precisamos é de duas comparações: uma, na qual verificamos que a chave procurada é maior que a chave do nó raiz, o que nos indica que devemos continuar a busca na subárvore à direita, e outra, na qual encontramos o elemento no próximo nó.

Agora vamos simular a busca pelo nó de chave 4: começamos comparando o 4 com a chave da raiz. Como 4 é menor que a chave da raiz, devemos focar nossa busca na subárvore à esquerda (cuja raiz é o nó de chave 3). Como 4 é maior que a chave da raiz dessa subárvore, devemos nos direcionar para a subárvore à direita do nó de valor 3. A chave que estamos procurando (4) é menor que a chave contido na raiz dessa subárvore (6). Assim sendo, devemos seguir buscando pelo nosso elemento na subárvore à esquerda, cuja raiz possui chave 4. A animação abaixo demonstra as comparações realizadas nessa busca.



Implementação da busca

O código abaixo implementa o algoritmo de busca descrito acima para a classe `BSTNode`, através do método `get`:

```
1  class BSTNode(object):
2      def __init__(self, key, value=None, left=None, right=None):
3          self.key = key
4          self.value = value
5          self.left = left
6          self.right = right
7
8      def get(self, key):
9          if key < self.key:
10             return self.left.get(key) if self.left else None
11         elif key > self.key:
12             return self.right.get(key) if self.right else None
13         else:
14             return self
```

Observe que este é um método recursivo (h◎ps://pt.wikipedia.org/wiki/Recursividade_%28ci%C3%AAncia_da_computa%C3%A7%C3%A3o%29), o que é condizente com a estrutura da árvore, que também é uma estrutura recursiva, com um nó sendo definido com base nele próprio. A função tem como **condição de parada** o nó ser nulo (`None`). Quando isso acontece, significa que chegamos ao fim de um galho da árvore sem ter encontrado a chave, isto é, a chave não existe na árvore.

Para realizar uma busca pela chave 4, devemos fazer o seguinte (onde `tree` é uma referência ao nó raiz da árvore):

```
1  tree = BSTNode(8)
2  ...
3  found = tree.get(4)
4  if found:
5      print(found)
```

O método `get` apresentado acima poderia ser refatorado, evitando a duplicação de código:

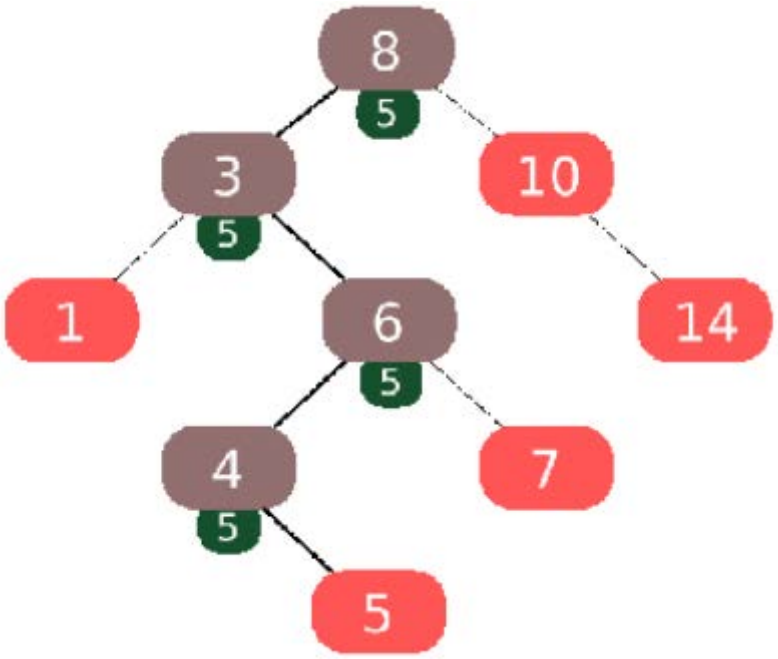
```
1  def get(self, key):
2      """Retorna uma referência ao nó de chave key
3      """
4      if self.key == key:
5          return self
6      node = self.left if key < self.key else self.right
7      if node is not None:
8          return node.get(key)
```

Inserção em uma Árvore Binária de Busca

Uma inserção em uma árvore binária de busca deve respeitar a propriedade fundamental dessa estrutura, mantendo menores à esquerda e maiores à direita. Para que isso seja possível, é interessante que a interface de programação da nossa árvore ofereça um método que faça a inserção de um elemento garantindo tal propriedade.

O algoritmo para a inserção funciona de forma semelhante à busca. Vamos descendo na árvore com o objetivo de encontrar o local certo onde o elemento deve ser inserido, verificando sempre se devemos continuar o percurso na subárvore à esquerda ou à direita do nó. Diferentemente da busca, na inserção nossa travessia termina ao encontrarmos um **nó folha**, no qual o elemento a ser inserido é adicionado como filho — à esquerda, se o elemento a ser adicionado for menor que o nó, ou à direita, caso contrário.

A animação abaixo ilustra o processo de inserção de um elemento:



(https://pythonhelp.files.wordpress.com/2015/01/image11.gif)

Implementação da inserção

Assim como a busca, o método para inserção também pode ser implementado de forma recursiva. Veja o código abaixo:

```
1  class BSTNode(object):
2      def __init__(self, key, value=None, left=None, right=None):
3          self.key = key
4          self.value = value
5          self.left = left
6          self.right = right
7
8      def add(self, node):
9          if node.value < self.value:
10             if self.left is None:
11                 self.left = node
12             else:
13                 self.left.add(node)
14         else:
15             if self.right is None:
16                 self.right = node
17             else:
18                 self.right.add(node)
```

Como podemos ver no método `add`, a inserção percorre a árvore até encontrar uma folha onde o novo nó pode ser inserido. Isso ocorre quando, no percurso da árvore, encontramos um nó que não possui um filho do lado esquerdo (quando o valor que estivermos inserindo for menor que o nó) ou do lado direito (quando o valor do nó que estivermos inserindo for maior que o valor do nó).

Novamente, para eliminar um pouco a repetição de código, o método `add` poderia ser refatorado para:

```
1  def add(self, key):
2      """Adiciona elemento à subárvore
3      """
4      side = 'left' if key < self.key else 'right'
5      node = getattr(self, side)
6      if node is None:
7          setattr(self, side, BSTNode(key))
8      else:
9          node.add(key)
```

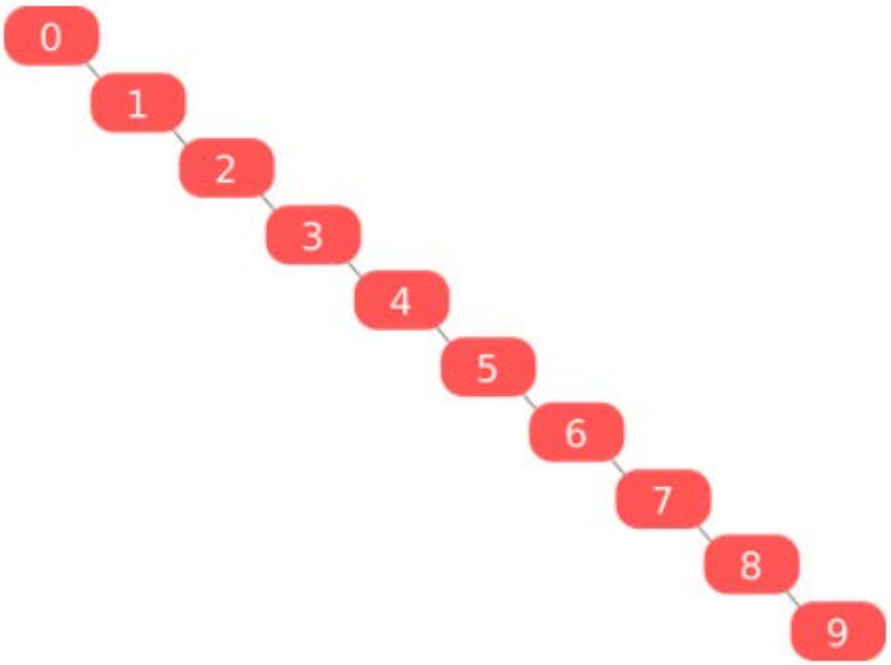
Porém, nosso algoritmo de inserção um problema: ele pode deixar desbalanceada a árvore após algumas inserções.

Balanceamento de árvore

Manter uma árvore bem organizadinha é um pouquinho mais complicado, pois é necessário que mantenhamos o **balanceamento da árvore**. Para entendermos melhor esse conceito, vamos ver um exemplo que ilustra o pior caso em uma árvore binária de busca não balanceada, que ocorre quando os elementos são inseridos de forma ordenada:

```
1  tree = BSTNode(0)
2  for i in range(1, 10):
3      tree.add(i)
```

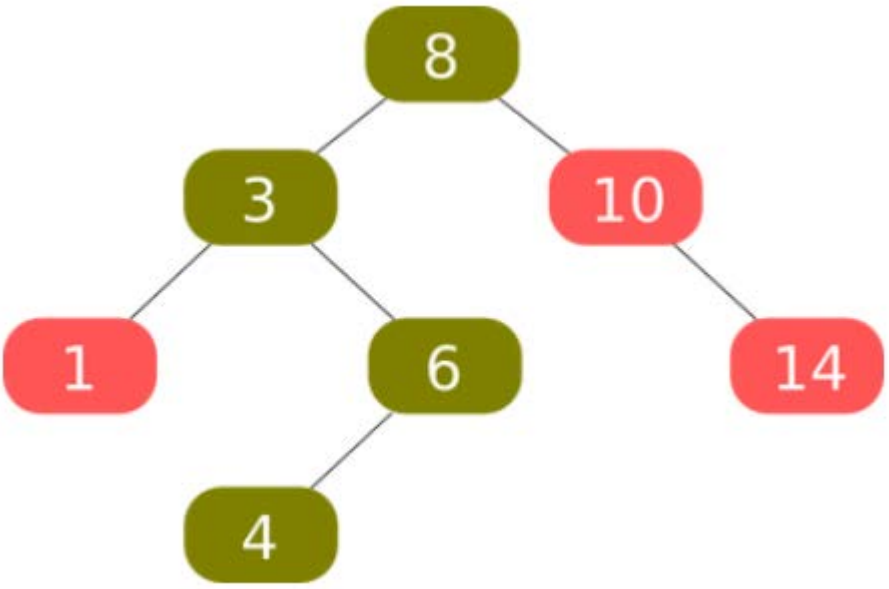
Veja a árvore resultante dessas inserções:



(h◉ps://pythonhelp.files.wordpress.com/2015/01/image09.png)

Esse layout é péssimo para a realização de uma busca, pois ela acaba se tornando linear, como se estivéssemos fazendo uma busca sequencial em uma lista encadeada.

Para evitar situações como esta, existem algoritmos que são usados durante a inserção de um elemento e que promovem uma reorganização dos nós da árvore para que seu layout fique mais **balanceado**. Para compreender melhor o conceito de balancemento de uma árvore, precisamos compreender antes o conceito de **altura** de uma árvore, que é definido pela quantidade de arestas no caminho mais longo entre a raiz e as folhas. A figura abaixo ilustra uma árvore de altura 3, que é a quantidade de arestas entre a raiz e a folha mais distante dela (de valor 4).



(https://pythonhelp.files.wordpress.com/2015/01/image01.png)

Uma **árvore balanceada** é uma árvore na qual a altura de uma subárvore não pode ser *muito maior* do que a altura da sua irmã.

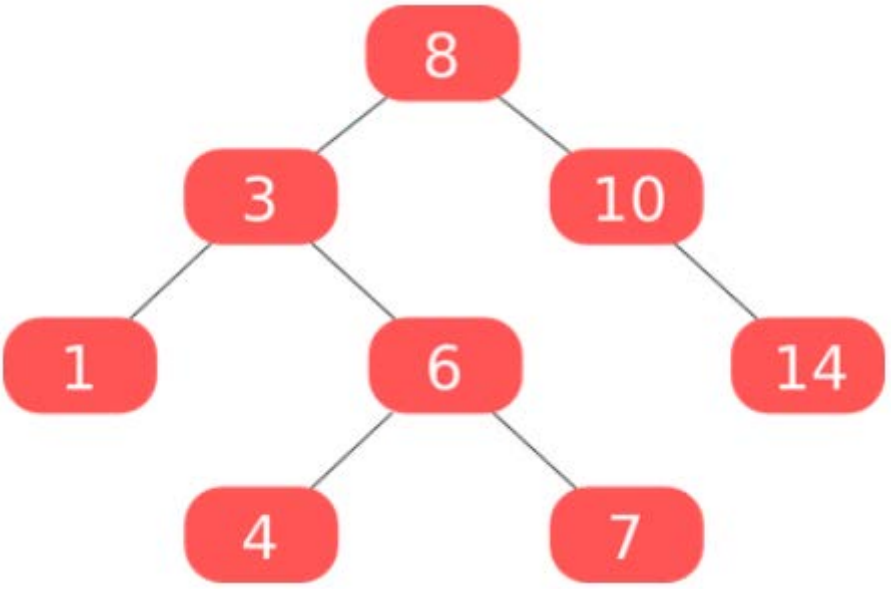
Para manter uma árvore balanceada, após cada inserção, devemos verificar se a árvore permanece balanceada e, em caso negativo, ela deve ser reorganizada, trocando os encadeamentos entre os nós. Isso pode ser um pouco custoso, mas compensa no momento de fazer a busca por algum elemento. Os tipos mais conhecidos de árvores binárias balanceadas são: as Árvores AVL (h p://pt.wikipedia.org/wiki/%C3%81rvore_AVL) e as Árvores Rubro-Negras (h p://pt.wikipedia.org/wiki/%C3%81rvore_rubro-negra). Mas esse assunto fica para um post futuro.

Remoção de um elemento

A remoção de um elemento é um pouco mais complicada do que a inserção e busca de um elemento. Existem 3 situações diferentes e que requerem diferentes abordagens para a remoção de um elemento:

1. o nó a ser removido é um nó folha
2. o nó a ser removido possui somente um filho
3. o nó a ser removido possui dois filhos

Considere a árvore da imagem abaixo:



(https://pythonhelp.files.wordpress.com/2015/01/image03.png)

Vamos analisar cada um dos casos acima.

Remoção de um nó folha

Imagine que desejamos remover o nó 4 da árvore acima. Para isso, basta fazer com que o campo `left` do nó 6 passe a apontar para `None`, e o *coletor de lixo* elimina o nó da memória pra gente em algum momento.

Remoção de um nó que possui um filho

Agora, desejamos remover o nó 10. Para isso, temos que fazer com que o nó pai do nó 10 (8) passe a apontar para o único filho de 10 (14).

Remoção de um nó que possui dois filhos

Este é o caso mais complicadinho. Imagine que queremos remover o nó 3, que possui como filhos a subárvore que tem como raiz o nó 1 e a subárvore do nó 6. Para remover o nó 3, é preciso que algum dos outros nós assuma o papel de raiz da subárvore. O melhor candidato para assumir esse posto é o nó cuja chave é mais próxima da chave do nó a ser removido.

Uma forma prática de encontramos tal valor é procurar o **menor valor contido na subárvore à direita do nó a ser removido**, isto é, o nó mais à esquerda da subárvore à direita. Na árvore do exemplo, esse nó é o nó de chave 4.

Implementação

O código abaixo implementa a remoção de um elemento. O método `remove` primeiramente encontra o nó a ser removido — é isso que as chamadas recursivas fazem — para depois fazer a remoção do elemento no código dentro do `else`. O método `_min` retorna o nó que contém o menor elemento em uma subárvore, isto é, o elemento mais à esquerda na subárvore em questão. Já o método `_remove_min` retira da subárvore o menor elemento, sendo usado para remover de sua posição o elemento que será utilizado como substituto ao elemento a ser removido, no caso deste possuir dois filhos.

```
1  class BSTNode(object):
2
3      def __init__(self, key, value=None, left=None, right=None):
4          self.key = key
5          self.value = value
6          self.left = left
7          self.right = right
8
9      def remove(self, key):
10         if key < self.key:
11             self.left = self.left.remove(key)
12         elif key > self.key:
13             self.right = self.right.remove(key)
14         else:
15             # encontramos o elemento, então vamos removê-lo!
16             if self.right is None:
17                 return self.left
18             if self.left is None:
19                 return self.right
20             #ao invés de remover o nó, copiamos os valores do nó
21             tmp = self.right._min()
22             self.key, self.value = tmp.key, tmp.value
23             self.right._remove_min()
24         return self
25
26     def _min(self):
27         """Retorna o menor elemento da subárvore que tem self com
28         """
29         if self.left is None:
30             return self
31         else:
32             return self.left._min()
33
34     def _remove_min(self):
35         """Remove o menor elemento da subárvore que tem self como
36         """
37         if self.left is None:  # encontrou o min, daí pode rearr
38             return self.right
39         self.left = self.left._removeMin()
40         return self
```

Os dois primeiros `if`s dentro do `else` (linhas 16 e 18) tratam o caso em que o nó a ser removido não possui filhos ou possui somente um filho. Observe que se o nó não possuir filho à direita, o filho à esquerda é retornado ao chamador, que é o próprio método `remove` na linha 11 ou 13.

Da linha 21 em diante tratamos o caso em que o nó a ser removido possui os dois filhos. A linha 21 obtém o elemento que irá substituir o elemento a ser removido. A linha seguinte copia os valores do nó substituto para o nó a ser "removido" (repare que

acabamos não removendo o nó, mas sim copiando os valores do nó substituto para o seu lugar). Depois disso, removemos o elemento substituto de sua posição original, chamando o método `_remove_min`.

Caso queira entender melhor o algoritmo para remoção de um elemento, leia mais na seção sobre Árvores Binárias de Busca do material disponível na web (http://algs4.cs.princeton.edu/32bst/) para o livro Algorithms, 4th Edition (http://algs4.cs.princeton.edu/home/).

Travessia em uma Árvore Binária

A última operação que vamos ver neste post é a travessia, que é útil em diversas situações, como para fazer a impressão da árvore, a geração de uma representação gráfica da mesma, ou então a aplicação de determinada transformação sobre todos os nós.

As três principais estratégias de travessia de uma árvore são:

- pré-ordem
- ordem simétrica
- pós-ordem

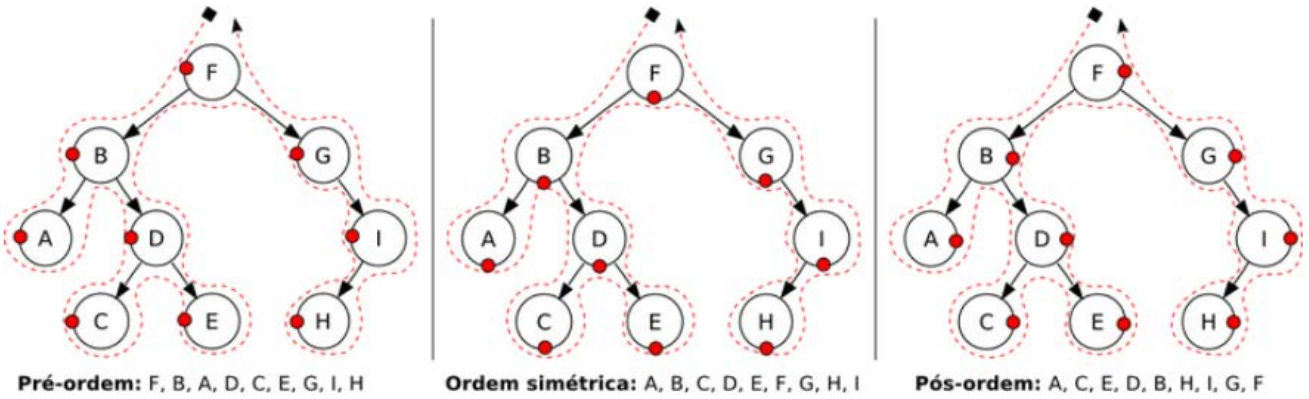
A seguir, temos um método que implementa as três possíveis estratégias para visitar todos os nós da árvore.

```
 1  def traverse(self, visit, order='pre'):
 2      """Percorre a árvore na ordem fornecida como parâmetro (pre,
 3         visitando os nós com a função visit() recebida como parâme
 4      """
 5      if order == 'pre':
 6          visit(self)
 7      if self.left is not None:
 8          self.left.traverse(visit, order)
 9      if order == 'in':
10          visit(self)
11      if self.right is not None:
12          self.right.traverse(visit, order)
13      if order == 'post':
14          visit(self)
```

Perceba que o parâmetro `visit` representa uma função que será aplicada a cada elemento da árvore. Se quiséssemos **imprimir** a árvore em **ordem simétrica**, bastaria fazermos:

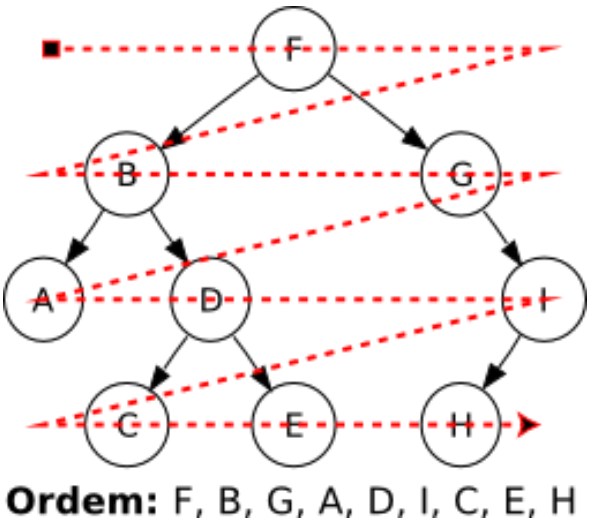
```
1 | tree.traverse(print, 'in')
```

As figuras abaixo — copiadas e adaptadas da Wikimedia (*ei, a licença permite!*) — ilustram os três tipos de travessia acima implementados:



(https://pythonhelp.files.wordpress.com/2015/01/image12.png)

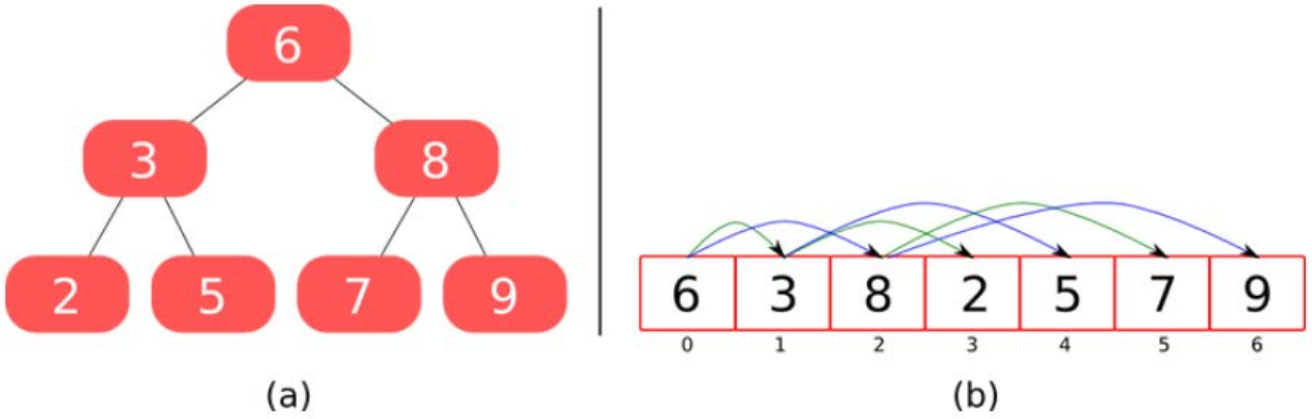
Essas três estratégias seguem a abordagem de travessia em profundidade (https://pt.wikipedia.org/wiki/Busca_em_profundidade), avançando sempre até o final de um galho da árvore e voltando para percorrer os outros galhos. Além dessa abordagem, existe também a travessia em largura (http://pt.wikipedia.org/wiki/Busca_em_largura), na qual os nós são percorridos nível por nível. A figura abaixo — *thanks again, Wikimedia!* — ilustra a travessia em largura.



(https://pythonhelp.files.wordpress.com/2015/01/image00.png)

Alternativas de Implementação

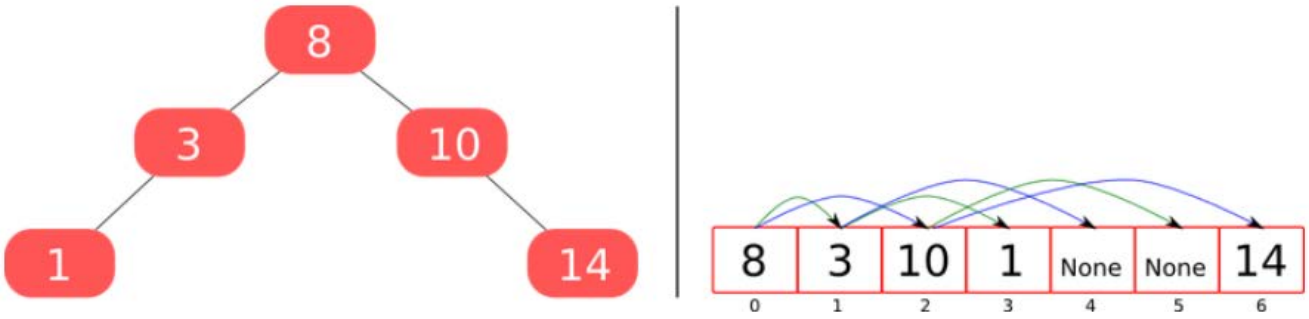
A estrutura de dados Árvore Binária de Busca pode também ser representada através de um simples array. A árvore do lado esquerdo da figura abaixo poderia ser representada pelo array ilustrado no lado direito da mesma figura.



(https://pythonhelp.files.wordpress.com/2015/01/image02.png)

Observe que a raiz é representada na posição `0` do array. Para acessar o filho à esquerda de qualquer elemento, basta acessar a posição `2*i+1` do array, sendo `i` a posição do elemento em questão. Para acessar o filho à direita de um elemento, basta acessar a posição `2*i+2`. Já o nó pai de um elemento `i` é encontrado na posição calculada através da divisão inteira `(i-1)/2`. Na figura acima, representamos os filhos à esquerda com uma seta verde e os filhos à direita com uma seta azul.

A desvantagem dessa abordagem está no espaço necessário para representar árvores binárias incompletas, como a árvore da figura abaixo, em que é necessário representar os nós não existentes também. No exemplo abaixo, uma árvore de 5 elementos precisou de um array de tamanho 7 para representá-la.



(https://pythonhelp.files.wordpress.com/2015/01/image13.png)

Mais sobre árvores

Isso não é tudo sobre árvores binárias de busca. Você pode estudar mais sobre essas estruturas lendo o artigo da Wikipedia sobre o assunto, (https://pt.wikipedia.org/wiki/%C3%81rvore_bin%C3%A1ria_de_busca) interagindo com a visualização do visualgo.net (http://visualgo.net/bst.html), ou lendo algum dos

livros clássicos de algoritmos e estruturas de dados. Além disso, você pode se interessar por outros tipos de árvores, como as listadas no artigo da wikipedia (https://pt.wikipedia.org/wiki/%C3%81rvore_%28estrutura_de_dados%29).

Conhecer e saber implementar uma árvore poderá facilitar a solução de diversos problemas que sem elas seriam bem mais complicados de resolver.

O código completo deste post pode ser visualizado aqui:
https://gist.github.com/stummjr/cd9974b513419f0554c5
(https://gist.github.com/stummjr/cd9974b513419f0554c5)

Obrigado ao **Elias** pela refatoração!

- algoritmos (https://pythonhelp.wordpress.com/tag/algoritmos/)
- árvores (https://pythonhelp.wordpress.com/tag/arvores/)
- árvores binárias de busca (https://pythonhelp.wordpress.com/tag/arvores-binarias-de-busca/)
- busca (https://pythonhelp.wordpress.com/tag/busca/)
- estruturas de dados (https://pythonhelp.wordpress.com/tag/estruturas-de-dados/)
- python (https://pythonhelp.wordpress.com/tag/python/)

10 comentários sobre “Árvore Binária de Busca em Python”

1. Eric Hideki disse:
   20 de janeiro de 2015 às 5:05 pm
   Como sempre, excelentes post.

   Continue assim!

   Abraço!

   Responder

   Valdir Stumm Jr disse:
   20 de janeiro de 2015 às 5:45 pm
   Valeu, Eric!

2. Responder

   Elton Morais (@eltonscm) disse:
   30 de janeiro de 2015 às 12:25 am

Eu aprendi a parte teórica sobre árvores na faculdade, mas nunca implementei na prática. Vou testar utilizando esse excelente post!

Abraço!

Responder

3. Pingback: Árvore Binária de Busca em Python - Peguei do
4. Pedro disse:
10 de junho de 2015 às 10:23 am
Um pouco de dúvida apenas no mapeamento no Array, se tivesse um código exemplo para esta forma de implementação ajudaria muito mais.
Excelente Post.

5. Responder
willamesoares disse:
22 de novembro de 2015 às 2:10 pm
Estou referenciando sua implementação em um simples projeto para a matéria de Pesquisa e Ordenação.
Ótimo post.
Vlw!

6. Responder
Adaut_o (@AdauT_0) disse:
7 de fevereiro de 2016 às 4:03 pm
Tem um pequeno bug no seu código, no tutorial do método de remoção, na linha 23, você tem que atribuir o retorno da chamada “self.right._remove_min()” à self.right …ficaria assim: “self.right = self.right._remove_min()”. Sem isso, a remoção do mínimo vai funcionar para quase todos os casos que ela deve cobrir, mas alguns ainda vão falhar. Se eu estiver errado, sitam-se à vontade para me corrigir.

Responder

7. duducosmos disse:
24 de abril de 2017 às 5:17 pm
Vou usar esse material na minha aula, deixo no slides a referencia desse site. Você autoriza?
Notei que o método add tem um problema, a versão correta está abaixo.

```
def add(self, key):
side = ‘left’ if key < self.key else 'right'
node = getattr(self, side)
if node is None:
setattr(self, side, BSTNode(key))
else:
node.add(key)
```

Responder

Valdir Stumm Jr disse:
24 de abril de 2017 às 7:52 pm
Opa, podes usar sim. 🙂
Sobre o erro, não entendi qual pois o método que você colou aqui parece idêntico ao do post. Ou estou enganado?

Responder

duducosmos disse:
24 de abril de 2017 às 9:39 pm
Eu que me confundi, troquei tá OK. Aí