# Frameworks em Python para árvores de busca balanceada

## Módulos Python para árvores balanceadas

A linguagem Python está emergindo como uma das linguagens de crescimento mais rápido no mercado. É o maior concorrente do *JavaScript*, especialmente por causa da onda crescente de IA (Inteligência Artificial). O Python foi desenvolvido por Guido van Rossum em 1991 e possui características vantajosas, como:

**Simplicidade**

**Versatilidade**

**Recursos de portabilidade**

Os frameworks são coleções de módulos de softwares e ferramentas que ajudam o programador no momento de construir seus projetos. Eles proporcionam uma ajuda fundamental aos profissionais, porque contêm embasamentos teóricos e práticos que otimizam o tempo, evitam erros comuns e repetitivos e, portanto, deixam o processo mais fluido e simplificado.

Existem diversos tipos de framework em Python, que veremos a seguir, mas as outras linguagens de programação também contam com seus próprios frameworks. Essa não é uma ferramenta exclusiva do Python.

Alguns módulos em Python para árvores balanceadas ainda são pouco explorados. Vejamos alguns exemplos:

**pyavl**
Trata-se de um pacote em C composto de um conjunto independente de rotinas dedicadas à manipulação de árvores AVL (arquivos avl.c, avl.h) e de um módulo de extensão para Python que se baseia nele (arquivo avlmodule.c) para fornecer objetos do tipo 'avl_tree' em Python, que pode se comportar como contêineres classificados ou listas sequenciais. Apesar da existência desse pacote de módulos, ele ainda é pouco explorado, existindo poucos exemplos na literatura de sua implementação.

**Rbtree**
Trata-se, sob a licença GPL 3, de um pacote composto de conjunto de rotinas para manutenção de árvores balanceadas rubro-negras. Ainda é um pacote com poucos módulos desenvolvidos e que carece de atualização (a última é de 2008).

# Módulo bisect

A pesquisa binária é uma técnica usada para pesquisar elementos em uma lista classificada. O **módulo bisect** fornece suporte para manter uma lista em ordem de classificação sem ter que classificar a lista após cada inserção. Para listas grandes de itens com operações de comparação custosas computacionalmente, isso pode ser uma melhoria em relação à abordagem mais comum na inserção dos dados.

É possível usar o módulo bisect para simular o uso da classe **TreeSet**, disponível em Java, inclusive evitando elementos duplicados nas árvores.

A classe TreeSet classifica os elementos em ordem crescente. É uma coleção para armazenar um conjunto de elementos únicos (objetos) de acordo com sua ordem natural. Ele cria uma coleção classificada que usa uma estrutura de árvore para o armazenamento de elementos ou objetos. Ou seja, os elementos são mantidos em ordem ascendente no conjunto de árvores.

Para usar os recursos do bisect, é necessário instalar seus pacotes. Use o **pip install bisect**.

No **algoritmo** a seguir, utilizamos os recursos do bisect para implementar as operações básicas de uma árvore binária:

```
import bisect

class Arvore(object):
  def __init__(self, elemento):
    self.arvore = []
    self.addElementos (elemento)


  #Adicionar muitos elementos
  def addElementos(self, elemento):
    for i in elemento:
      if i in self: continue
      self.addElementos(i)

  #Adicionar um(1) elemento
  def addElemento(self,elemento):
    if elemento not in self:
      bisect.insort(self.arvore, elemento)
```

```
#Remove um elemento
def removeElemento(self, elemento):
  try:
    self.arvore.remove(elemento)
  except:
    return False
  return True
```

Ainda dentro da **classe Arvore**, podemos inserir um *iterator* de objetos (__iter()__). Você pode inserir outros métodos interessantes para complementar a sua **classe Arvore**.

```
def __iter__(self):
  for element in self.arvore:
    yield element

def __str__(self):
  return str(self.arvore)
```

Para fazer uso das funcionalidades da nossa **classe Arvore**, no script principal invocamos os métodos implementados dentro da classe.

No algoritmo a seguir, temos implementadas as operações de busca, inserção e remoção em uma árvore binária:

1. Na operação de busca, usamos o operador in do Python, que verifica se o elemento está na árvore.
2. Na operação de inserção, usamos os métodos **addElemento()** e **addElementos()** para inserir um (1) ou mais elementos, respectivamente, na árvore.
    1. **addElemento(var)**: inserindo um (1) elemento (parâmetro).
    2. **addElementos()**: inserindo vários elementos (enviando uma lista de elementos por parâmetro).
3. Na operação de remoção, usamos o método **removeElementos()** para remover algum item (enviado por parâmetro) da árvore.

```
4.
5. if __name__ == '__main__':
6.     arvore = Arvore([12, 7, 7, 1, 3, 10])
7.
8.     print("árvore:", arvore)
9.
10.    print("Tem 7 na árvore?", 7 in arvore)
11.
12.    arvore.addElemento(4)
13.    print("Adicionando 4: ", arvore)
14.
15.    arvore.removeElemento(3)
16.    print("Removendo 3: ", arvore)
17.
18.    arvore.removeElemento(7)
```

## Módulo sbbst

Uma árvore binária de busca autobalanceada é uma estrutura de dados avançada, que otimiza os tempos de inserção, deleção e busca. Um módulo Python que implementa e executa essas atividades eficientemente é a **sbbst** (do inglês *self-balancing binary search tree*).

*Use* ***!pip install*** **sbbst** para instalar o módulo.

O módulo **sbbst** possui espaço **O**(**n**) na memória, em que **n** equivale à quantidade de nós de uma árvore. Os tempos de complexidade respectivos e funções são:

| Tempo de complexidade | Funções de classe | Ação |
|---|---|---|
| **O**(1) | sbbst.getSize() | Tamanho da árvore |
| **O**(1) | sbbst.getHeightTree() | Altura da árvore |
| **O**(log **n**) | sbbst.search(x) | Busca na árvore |

| Tempo de complexidade | Funções de classe | Ação |
|---|---|---|
| **O**(log **n**) | sbbst.insert(x) | Inserção na árvore |
| **O**(log **n**) | sbbst.delete(x) | Deletar x da árvore |
| **O**(log **n**) | sbbst.getMinVal() | Valor mínimo |
| **O**(log **n**) | sbbst.getMaxVal() | Valor máximo |
| **O**(**K**+log **n**) | sbbst.kthsmallest(k) | k-ésimo valor mínimo |
| **O**(**K**+log **n**) | sbbst.kthlargest(k) | k-ésimo valor máximo |
| **O**(**n**) | str(sbbst) | Visualizar árvore |

**Tabela 2** - Método de iteração e método de modo de exibição.
Adaptada da documentação contida em Pypi.org (2022).

Exemplo de implementação com sbbst

A seguir, teremos um exemplo de implementação de árvore AVL usando o módulo **sbbst**. No algoritmo a seguir, na linha 1, invocamos o import from sbbst import sbbst. Veja:

```
1  from sbbst import sbbst
2
3  tree = sbbst()
4  nums = [131, 4, 134, 135, 180, 1, 3, 21, 14, 142, 80, 146]
5  for num in nums:
       tree. insert (num)
6
```

```
7  print("Número de elementos:",tree.getSize())
8  print("altura:",tree.getHeightTree())
9  print("Min valor:",tree.getMinVal())
10 print("Max valor:",tree.getMaxVal())
11 print("3º menor valor:",tree.kthsmallest(3))
12 print("2º maior valor:",tree.kthlargest(2))
13 print("Pre-Orden:", tree.preOrder())
```

Na linha 3 até a linha 6 do algoritmo anterior, temos a instanciação do objeto tree, uma lista de elementos e a inserção desses elementos, usando um loop. Nas próximas linhas, temos o exemplo de uso das funções. Confira!

Linha 8: quantidade de elementos da árvores.

Linha 9: altura da árvore.

Linhas 10 e 11: valor mínimo e valor máximo, respectivamente.

Linhas 12 e 13: valor mínimo e máximo na **k**-ésima posição.

Linhas 14, 15 e 16: percurso pré-ordem, in-ordem e pós-ordem.

Linhas 17 e 18: deleção de elemento na árvore.

Linha 19: inserção de elemento na ár