TRABALHO #1 – ESTRUTURAS DE DADOS II

EVANDRO E. S. RUIZ

As árvores de busca são tipos abstratos de dados (TAD) essenciais em Computação. Neste tópico temos as árvores 2-3-4, um tipo de árvore binária de busca (Abb) balanceada que são um tipo restrito de árvores **B**. Essa última é o tipo mais adequado de TAD para gerenciamento de dados em memória externa. Como um meio didático de fixar os conceitos sobre árvores **B**, solicita-se::

Enunciado

Desenvolver códigos em Python para implementar as árvores 2-3-4. Para este TAD, implemente códigos para inserção, remoção e visualização de dados unitários.

A implementação deste TAD será avaliado da seguinte forma:

- Dois tipos de percurso em árvore e uma forma de visualização de modo que seja possível avaliar as inserções e remoções neste TAD (peso=20%);
- Um algoritmo para inserção de um dado na árvore 2-3-4 (peso=20%);
- Um algoritmo remover um dado neste mesmo TAD (peso=50%), e;
- Uma função que demostre os algoritmos de inserção e remoção deste TAD (peso=10%).

É fornecido um código fonte inicial de árvore 2-3-4, o qual, a definição de `__init__` deve ser mantida. A árvore 2-3-4 deve ser implementada de modo que a capacidade do nó seja regulada internamente pelas funções de inserção e remoção, ou por funções auxiliares. A ideia é buscar uma parametrização para que seja possível escalar a implementação para tipos maiores de nós, ou seja, buscando implementar árvores **B** ainda maiores que a 2-3-4.

1. Entrega

A entrega dos trabalho deve ocorrer em duas mídias:

(1) Os códigos finais devem ter uma implementação possível de ser verificada on-line, por exemplo, Google Collab, e, se possível, devem ser compartilhadas com `evandro@usp.br` de modo que o mesmo possa executar e testar os códigos;
(2) Os códigos, juntamente com uma demonstração de sua execução, devem ser adicionados, no formato PDF, na plataforma `edisciplinas.usp.br`;
(3) A entrega deve ocorrer impreterivelmente até o dia 29 de outubro do corrente ano às 23h59.

Critérios de avaliação

(1) Completude e estruturação do código (80%) e;
(2) Didática e comentários ao código (20%).

Observações

- Os trabalhos são individuais, mas admite-se a possibilidade de ser produzido por até dois autores, no máximo;
- Caso o trabalho tenha dois autores, há a necessidade de explicar e detalhar qual a contribuição de cada um no trabalho, e;

Código fonte inicial

Árvore 2-3-4

Implementação baseada no código do acadêmico Vinicius Fernandez Baca dos Santos.
Modelo a seguir pelos estudantes caso não tenhamos uma estrutura mais didática.

```
class Node:

    def __init__(self, keys, par = None):
        self.keys = list([keys])
        self.parent = par
        self.child = list()

    def __lt__(self, node):
        return self.keys[0] < node.keys[0]

    def _isLeaf(self):
        return len(self.child) == 0

    """
    Função auxiliar para adicionar um novo noh ao original. Controla
    o split e dimensãoo de um noh ja existente
    """

    def _insertIntoNode(self, new_node):
        for child in new_node.child:
            child.parent = self
        self.keys.extend(new_node.keys)
        self.keys.sort()
        self.child.extend(new_node.child)
        if len(self.child) > 1:
            self.child.sort()
        if len(self.keys) > 3:
            self._split()

    """
    Para inserção de um dado na árvore.
    Procura onde inserir um noh na árvore e o insere recursivamente.
    Esta, é a função interna, usada apenas pelo TAD.
    """

    def _insert(self, new_node):
        # Caso 1 - Insercao em uma folha
        if self._isLeaf():
            self._insertIntoNode(new_node)

        # Caso 2 - Nao é uma folha - Encontra o lugar para inserir e o insere
        # com recursão
        elif new_node.keys[0] > self.keys[-1]:
            self.child[-1]._insert(new_node)
        else:
            for i in range(0, len(self.keys)):
                if new_node.keys[0] < self.keys[i]:
                    self.child[i]._insert(new_node)
                    break

    # Funcao "split". É usada quando um noh sofre overflow.
    def _split(self):
        left_child = Node(self.keys[0], self)
        right_child = Node(self.keys[2], self)
        right_child.keys.append(self.keys[3])

        if self.child:
            self.child[0].parent = left_child
            self.child[1].parent = left_child
            self.child[2].parent = right_child
            self.child[3].parent = right_child
            self.child[4].parent = right_child

            left_child.child = [self.child[0], self.child[1]]
            right_child.child = [self.child[2], self.child[3],self.child[4]]

        self.child = [left_child]
        self.child.append(right_child)
        self.keys = [self.keys[1]]

        # Adiciona o novo noh (resultado do split) ao seu node pai
        if self.parent:
            if self in self.parent.child:
                self.parent.child.remove(self)
            self.parent._insertIntoNode(self)
        else:
            left_child.parent = self
```

```
            right_child.parent = self


    def _preorder(self):
        print (self.keys)
        for child in self.child:
            child._preorder()
#end of Node class
```

```
class Tree234:

    def __init__(self):
        self.root = None

    def insert(self, elem):
        print("Inserindo: " + str(elem))
        if self.root is None:
            self.root = Node(elem)
        else:
            self.root._insert(Node(elem))
            while self.root.parent:
                self.root = self.root.parent
        return True


    def preorder(self):
        print('\n Impressao em pre-ordem\n')
        self.root._preorder()


    def visualize(self):
        print('\n Estrutura de arvore (visual em largura)')
        this_level = [self.root]

        while this_level:
            next_level = list()
            print('\n')

            for n in this_level:
                print (str(n.keys), end = ' ')

                for child in n.child:
                    next_level.append(child)
                this_level = next_level
#end of Tree class
```

```
import random
```

```
def main():

  tree = Tree234()
  numbers=random.sample(range(100), 25)

  for x in numbers:
      tree.insert(x)
  tree.visualize()

  tree.preorder()


main()
```

```
Inserindo: 33
Inserindo: 6
Inserindo: 42
Inserindo: 18
Inserindo: 31
Inserindo: 10
Inserindo: 90
Inserindo: 27
Inserindo: 0
Inserindo: 51
Inserindo: 67
Inserindo: 39
Inserindo: 58
Inserindo: 94
Inserindo: 70
Inserindo: 69
Inserindo: 28
Inserindo: 93
Inserindo: 12
Inserindo: 43
Inserindo: 88
Inserindo: 5
Inserindo: 15
```

```
Inserindo: 71
Inserindo: 97

 Estrutura de arvore (visual em largura)


[33, 67]

[6, 18] [51] [70, 90]

[0, 5] [10, 12, 15] [27, 28, 31] [39, 42, 43] [58] [69] [71, 88] [93, 94, 97]
 Impressao em pre-ordem

[33, 67]
[6, 18]
[0, 5]
[10, 12, 15]
[27, 28, 31]
[51]
[39, 42, 43]
[58]
[70, 90]
[69]
[71, 88]
[93, 94, 97]
```

Bom trabalho!

Departamento de Computação e Matemática, Faculdade de Filosofia, Ciências e Letras de Ribeirão Preto, Universidade de São Paulo – USP
E-mail address: evandro@usp.br