Relatório Projeto 2 V1.1 AED 2023/2024

Nome: Rodrigo José Morenito Borges Nº Estudante: 2022244993
PL (inscrição):7 Email:uc2022244993@student.uc.pt

IMPORTANTE:

- Os textos das conclusões devem ser manuscritos... o texto deve obedecer a este requisito para não ser penalizado.
- Texto para além das linhas reservadas, ou que não seja legível para um leitor comum, não será tido em conta.
- O relatório deve ser submetido num único PDF que deve incluir os anexos. A não observância deste formato é penalizada.

1. Planeamento

	Semana 1	Semana 2	Semana 3	Semana 4	Semana 5
Árvore Binária	X				
Árvore Binária Pesquisa		X			
Árvore AVL			X		
Árvore VP				X	
Finalização Relatório					X

2. Recolha de Resultados *(tabelas)*

Árvore binária:

Tree Type	Input Type	Insertion Time	Rotation Count	Elements
ArvoreBinaria	A	17,75475621		20000
ArvoreBinaria	A	79,09228778		40000
ArvoreBinaria	A	191,2497244		60000
ArvoreBinaria	A	379,7358871		80000
ArvoreBinaria	A	635,5164452		100000
ArvoreBinaria	B	16,72241426		20000
ArvoreBinaria	B	74,96011925		40000
ArvoreBinaria	B	193,1576424		60000
ArvoreBinaria	B	382,9081168		80000
ArvoreBinaria	B	640,8979406		100000
ArvoreBinaria	C	39,35269356		20000
ArvoreBinaria	C	195,1997771		40000
ArvoreBinaria	C	498,9410331		60000
ArvoreBinaria	C	939,8188901		80000
ArvoreBinaria	C	1503,464293		100000
ArvoreBinaria	D	38,85361433		20000
ArvoreBinaria	D	195,5665503		40000
ArvoreBinaria	D	480,108191		60000
ArvoreBinaria	D	909,0522506		80000
ArvoreBinaria	D	1515,930277		100000

Árvore binária pesquisa:

Tree Type	Input Type	Insertion Time	Rotation Count	Elements
BinariaPesquisa	A	10,65028405		20000
BinariaPesquisa	A	44,82563114		40000
BinariaPesquisa	A	102,4107065		60000
BinariaPesquisa	A	200,3756433		80000
BinariaPesquisa	A	338,9497194		100000
BinariaPesquisa	B	7,896276236		20000
BinariaPesquisa	B	32,16408372		40000
BinariaPesquisa	B	84,28624201		60000
BinariaPesquisa	B	172,9366741		80000
BinariaPesquisa	B	287,4074163		100000
BinariaPesquisa	C	0,203155994		20000
BinariaPesquisa	C	0,112511873		40000
BinariaPesquisa	C	0,318911552		60000
BinariaPesquisa	C	0,229348183		80000
BinariaPesquisa	C	0,450210094		100000
BinariaPesquisa	D	0,049037218		20000
BinariaPesquisa	D	0,107997656		40000
BinariaPesquisa	D	0,310964108		60000
BinariaPesquisa	D	0,265002966		80000
BinariaPesquisa	D	0,320006132		100000

Árvore AVL:

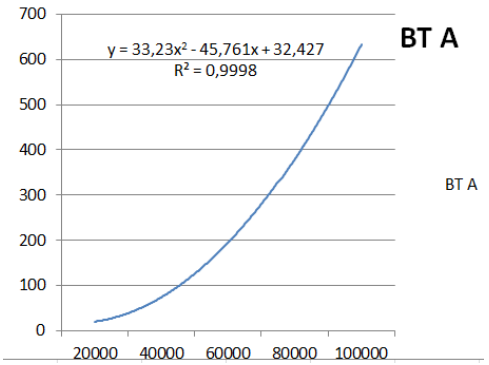
Tree Type	Input Type	Insertion Time	Rotation Count	Elements
ArvoreAVL	A	3,045026779	114812	200000
ArvoreAVL	A	5,584629059	229634	400000
ArvoreAVL	A	8,995342255	344636	600000
ArvoreAVL	A	13,06384587	459663	800000
ArvoreAVL	A	15,29864907	574761	1000000
ArvoreAVL	B	2,530788898	114818	200000
ArvoreAVL	B	6,28945446	229613	400000
ArvoreAVL	B	9,335362196	344647	600000
ArvoreAVL	B	11,68477988	459419	800000
ArvoreAVL	B	14,70384049	574501	1000000
ArvoreAVL	C	3,200802326	124598	200000
ArvoreAVL	C	6,995465517	223176	400000
ArvoreAVL	C	10,97605371	302724	600000
ArvoreAVL	C	15,09341359	366271	800000
ArvoreAVL	C	21,46306396	417565	1000000
ArvoreAVL	D	2,827914953	120905	200000
ArvoreAVL	D	6,3187747	210924	400000
ArvoreAVL	D	9,761363745	278814	600000
ArvoreAVL	D	13,07999682	330320	800000
ArvoreAVL	D	16,09253979	368532	1000000

Árvore Vermelha e Preta:

Tree Type	Input Type	Insertion Time	Rotation Count	Elements
RBTree	A	1,070888996	115032	200000
RBTree	A	2,022932529	229639	400000
RBTree	A	3,291566849	344474	600000
RBTree	A	4,721413851	459038	800000
RBTree	A	5,685123205	573541	1000000
RBTree	B	1,181544304	114951	200000
RBTree	B	1,846201897	229997	400000
RBTree	B	2,596480131	344724	600000
RBTree	B	3,841418743	459756	800000
RBTree	B	4,910843134	574699	1000000
RBTree	C	1,36613059	103936	200000
RBTree	C	2,593099356	186823	400000
RBTree	C	3,685707569	252670	600000
RBTree	C	5,237735748	306307	800000
RBTree	C	6,072700739	349471	1000000
RBTree	D	0,758996487	100934	200000
RBTree	D	1,59905076	176023	400000
RBTree	D	2,52888155	233033	600000
RBTree	D	3,005260229	275630	800000
RBTree	D	3,758115053	307862	1000000

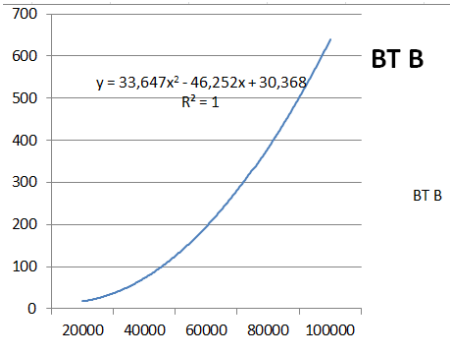
3. Visualização de Resultados *(gráficos) - Tempo em função do número de elementos em cada árvore e conjunto*

Árvore Binária - Tarefa 1:



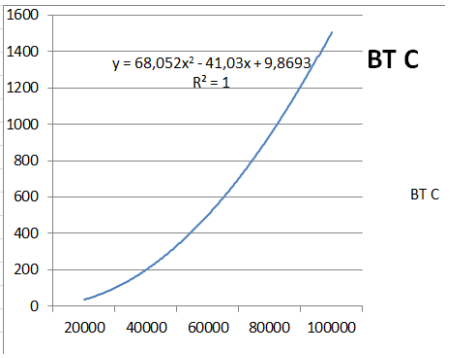
Complexidade/Regressão: O(n) linear

Conjunto B:



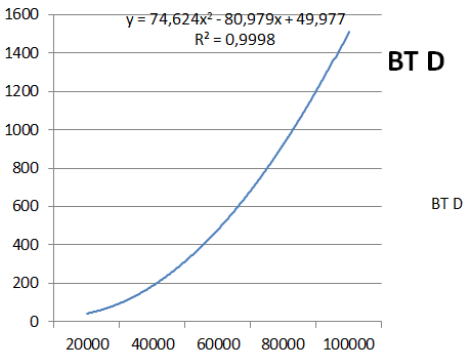
Complexidade/Regressão: O(n) linear

Conjunto C:



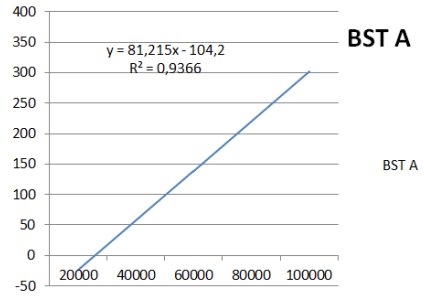
Complexidade/Regressão: O(n) linear

Conjunto D:



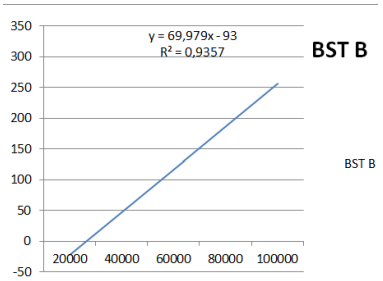
Complexidade/Regressão: O(n) linear

Árvore Binária Pesquisa - Tarefa 2 :



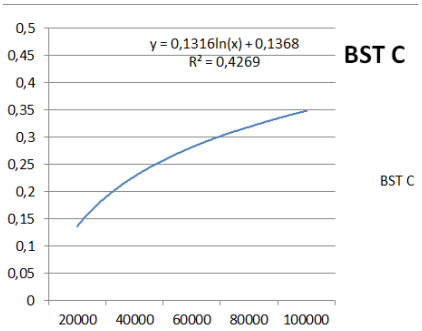
Complexidade/Regressão: O(n) linear

Conjunto B:



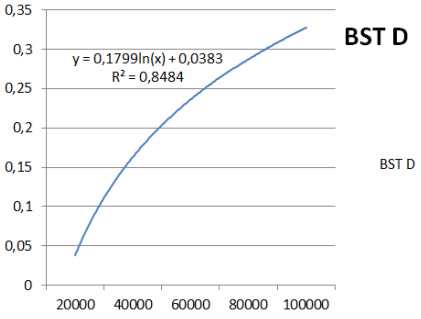
Complexidade/Regressão: O(n) linear

Conjunto C:



Complexidade/Regressão: Log(n)

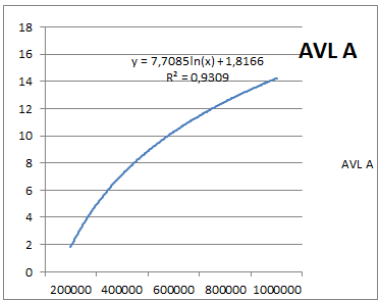
Coniunto D:



Complexidade/Regressão: Log(n)

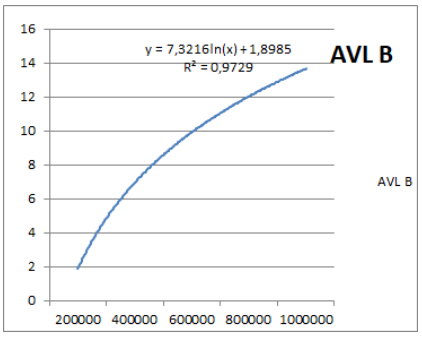
Árvore AVL - Tarefa 3:

Conjunto A:



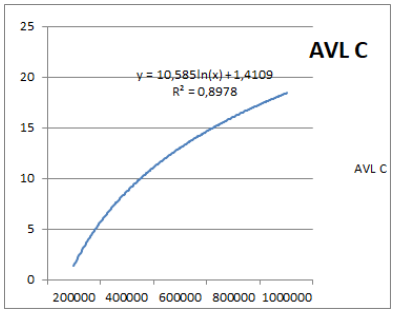
Complexidade/Regressão: Log(n)

Conjunto B:



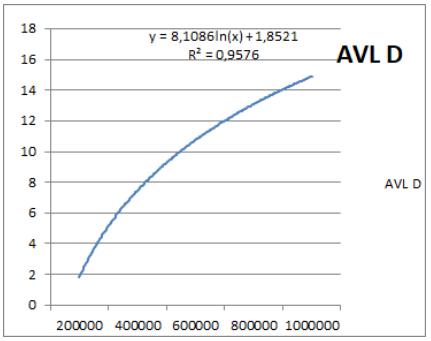
Complexidade/Regressão: Log(n)

Conjunto C:



Complexidade/Regressão: Log(n)

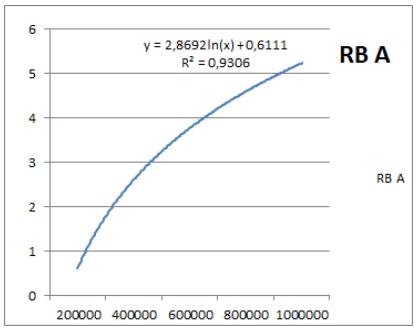
Conjunto D:



Complexidade/Regressão: Log(n)

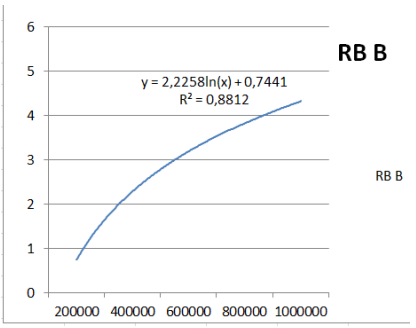
Árvore RB - Tarefa 4:

Conjunto A:



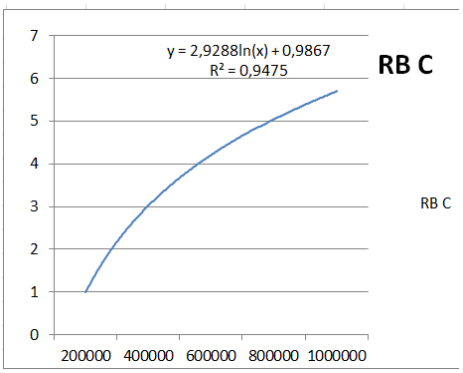
Complexidade/Regressão: Log(n)

Conjunto B:



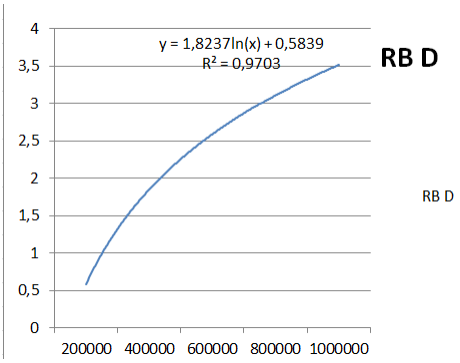
Complexidade/Regressão: Log(n)

Conjunto C:



Complexidade/Regressão: Log(n)

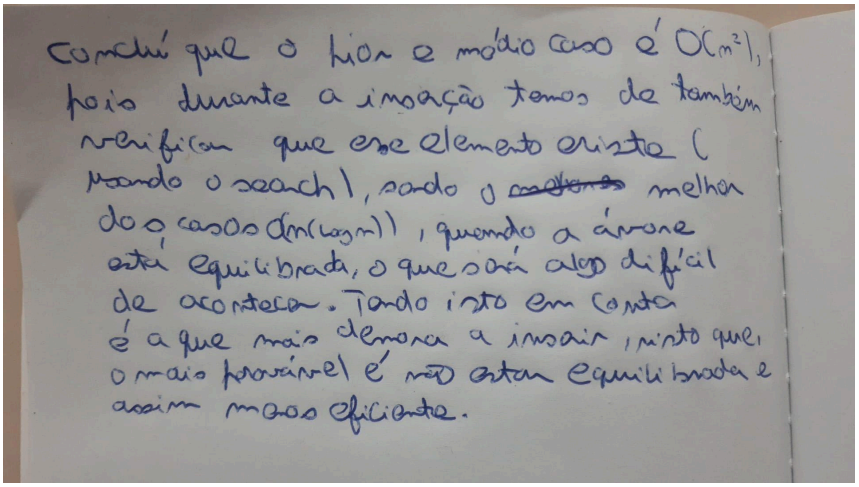
Conjunto D:



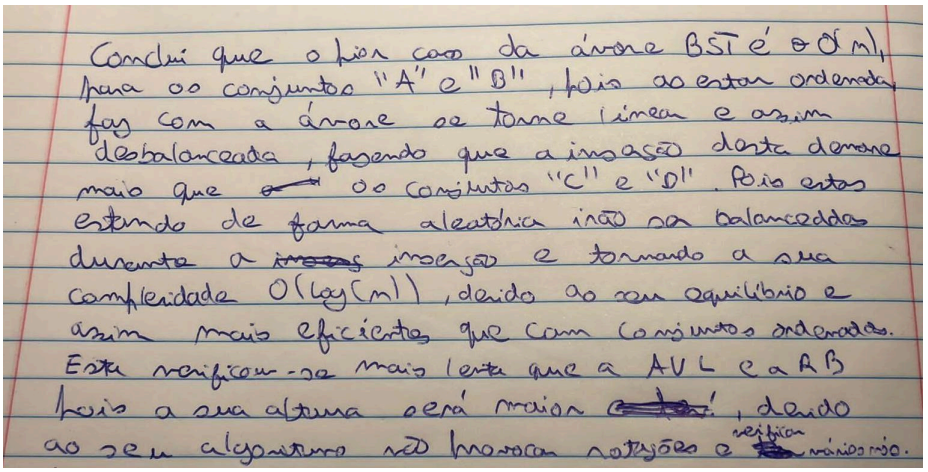
Complexidade/Regressão: Log(n)

4. Conclusões *(as linhas desenhadas representam a extensão <u>máxima</u> de texto manuscrito)*

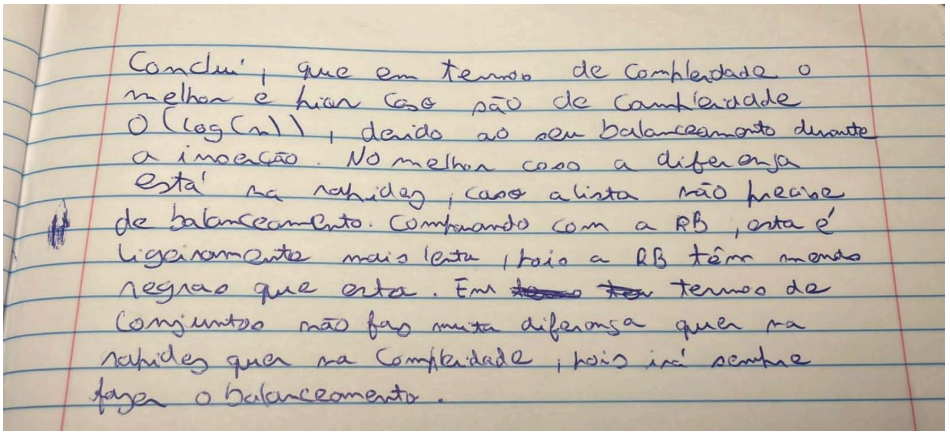
4.1 Tarefa 1


Concluí que o bior e médio caso é $O(m^2)$, pois durante a inserção temos de também verificar que esse elemento existe (usando o search), sendo o ~~melhor~~ melhor dos casos $O(n(\log m))$, quando a árvore está equilibrada, o que será algo difícil de acontecer. Tendo isto em conta é a que mais demora a inserir, visto que o mais provável é não estar equilibrada e assim mais eficiente.

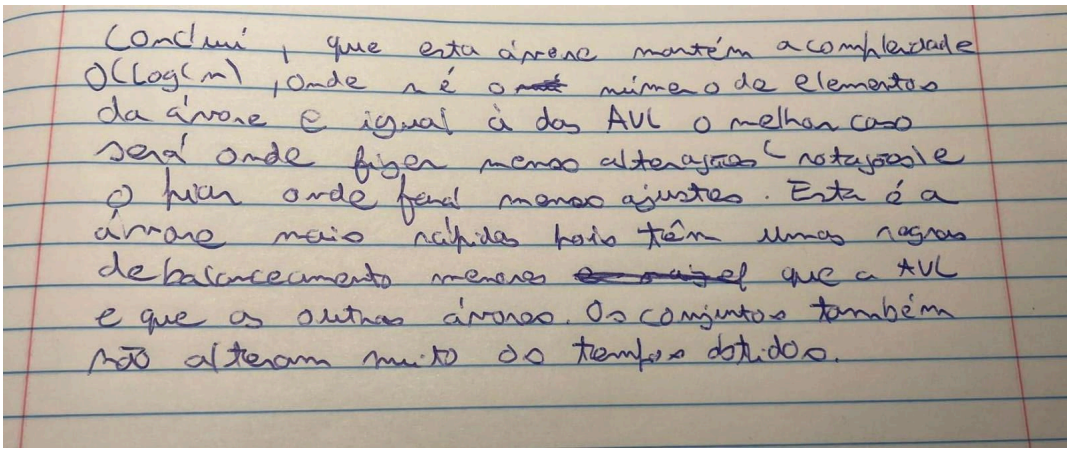
4.2 Tarefa 2


Concluí que o bior caso da árvore BST é $O(m)$, para os conjuntos "A" e "B", pois ao estar ordenada, faz com a árvore se torne linear e assim desbalanceada, fazendo que a inserção desta demore mais que ~~o~~ os conjuntos "C" e "D". Pois estes estando de forma aleatória irão ser balanceadas durante a ~~inserção~~ inserção e tornando a sua complexidade $O(\log(m))$, devido ao seu equilíbrio e assim mais eficientes que com conjuntos ordenados. Esta verificou-se mais lenta que a AVL e a RB pois a sua altura será maior ~~que~~, devido ao seu algoritmo não honrar notações e verifica mínimo.

4.3 Tarefa 3


Concluí, que em termos de complexidade o melhor e bior caso são de complexidade $O(\log(n))$, devido ao seu balanceamento durante a inserção. No melhor caso a diferença está na rapidez, caso a lista não precise de balanceamento. Comparando com a RB, esta é ligeiramente mais lenta, pois a RB têm menos regras que esta. Em ~~termos~~ termos de conjuntos não faz muita diferença quer na rapidez quer na complexidade, pois irá sempre fazer o balanceamento.

4.4 Tarefa 4


Concluí, que esta árvore mantém a complexidade O(Log(n), onde n é o ~~mat~~ número de elementos da árvore e igual à das AVL o melhor caso será onde fizer menos alterações (~~rotações~~) e o pior onde fará menos ajustes. Esta é a árvore mais rápidas pois têm umas regras de balanceamento menores ~~e mais~~ que a AVL e que as outras árvores. Os conjuntos também não alteram muito os tempos obtidos.

Anexo B - Código de Autor

//indicar as linhas de código de que é autor//

```
import random
import time
import pandas as pd
#criar os inputs
def criar_input(n,c):
    lista=[]
    i = 0
    if(c=="D"):
        while i <n:
            numero=random.randint(1,n)
            aleatorio=random.randint(1,10)
            if aleatorio!=1 and i!=n-1:
                lista.append(numero)
                i+=1
            lista.append(numero)
            i+=1
    else:
        while i <n:
            numero=random.randint(1,n)
            aleatorio=random.randint(1,10)
            if aleatorio==1 and i!=n-1:
                lista.append(numero)
                i+=1
            lista.append(numero)
            i+=1
    if(c == "A"):
        lista.sort()
    elif(c == "B"):
        lista.sort(reverse=True)
    elif(c=="C"or c=="D"):
```

```
        random.shuffle(lista)

    return lista

#arvore binaria
class NodeBinaria:
    def __init__(self, element):
        self.element = element
        self.left = None
        self.right = None
    def search(self,element):
        if self.left != None:
          if self.left.search(element):
                return True
        if self.element == element:
            return True
        if self.right != None:
            if self.right.search(element):
                return True
        return False
class ArvoreBinaria:
    def __init__(self, element):
       self.raiz=NodeBinaria(element)
    def insert(self,element):
        current = self.raiz
        if current.search(element) == False:
            while current:
                if current.left is None:
                    current.left = NodeBinaria(element)
                    return
                elif current.right is None:
                    current.right = NodeBinaria(element)
                    return
                else:
                    numeroextra = random.randint(0,100)
                    if numeroextra < 50:
                        current=current.left
                    else:
                        current=current.right



#arvore binaria de pesquisa
class BinariaPesquisa:
    def __init__(self, element):
```

```
        self.raiz=NodeBinaria(element)
    def insert(self, element):
        current = self.raiz
        while current:
            if element < current.element:
                if current.left is None:
                    current.left = NodeBinaria(element)
                    return
                else:
                    current = current.left
            elif element > current.element:
                if current.right is None:
                    current.right = NodeBinaria(element)
                    return
                else:
                    current = current.right
            else:
                return

#arvore AVL
class NodeAVL:
    def __init__(self, element):
        self.element = element
        self.left = None
        self.right = None
        self.height = 1

class ArvoreAVL:
    def __init__(self, element):
        self.raiz = NodeAVL(element)
        self.rotat_count = 0

    def insert(self, element):
        self.raiz = self._insert(element, self.raiz)

    def _insert(self, element, node):
        if node is None:
            return NodeAVL(element)

        if element < node.element:
            node.left = self._insert(element, node.left)
        elif element > node.element:
            node.right = self._insert(element, node.right)
        else:
            return node
```

```
        node.height = 1 + max(self._height(node.left),
self._height(node.right))

        balance = self._get_balance(node)

        if balance > 1 and element < node.left.element:
            self.rotat_count += 1
            return self._right_rotate(node)

        if balance < -1 and element > node.right.element:
            self.rotat_count += 1
            return self._left_rotate(node)

        if balance > 1 and element > node.left.element:
            self.rotat_count += 2
            node.left = self._left_rotate(node.left)
            return self._right_rotate(node)

        if balance < -1 and element < node.right.element:
            self.rotat_count += 2
            node.right = self._right_rotate(node.right)
            return self._left_rotate(node)

        return node

    def _height(self, node):
        if node is None:
            return 0
        return node.height

    def _get_balance(self, node):
        if node is None:
            return 0
        return self._height(node.left) - self._height(node.right)

    def _left_rotate(self, z):
        y = z.right
        T2 = y.left

        y.left = z
        z.right = T2

        z.height = 1 + max(self._height(z.left),
self._height(z.right))
```

```
        y.height = 1 + max(self._height(y.left),
self._height(y.right))

        return y

    def _right_rotate(self, z):
        y = z.left
        T3 = y.right

        y.right = z
        z.left = T3

        z.height = 1 + max(self._height(z.left),
self._height(z.right))
        y.height = 1 + max(self._height(y.left),
self._height(y.right))

        return y
class RBTree:
    def __init__(self):
        self.root = None
        self.rotat_count=0

    def insert(self, val):
        new_node = RBNode(val)
        new_node.red = True  # Começar o nó como vermelho
        parent = None  # Inicializa o pai como nulo
        current = self.root  # Começa a busca a partir da raiz

        # Encontra o local correto para inserir o novo nó
        while current is not None:
            parent = current
            if new_node.val < current.val:
                current = current.left
            elif new_node.val > current.val:
                current = current.right
            else:
                return

        # Define o pai do novo nó e faz a inserção
        new_node.parent = parent
        if parent is None:
            self.root = new_node
        elif new_node.val < parent.val:
            parent.left = new_node
```

```
        else:
            parent.right = new_node

        self.fix_insert(new_node)# verificar as regras da arvore
vermelha e preta

    def fix_insert(self, new_node):
        while new_node != self.root and new_node.parent.red:  #
Enquanto o pai do novo nó for vermelho
            if new_node.parent == new_node.parent.parent.right:  # Se
o pai do novo nó for filho direito
                uncle = new_node.parent.parent.left  # Tio é o filho
esquerdo do avô
                if uncle and uncle.red:  # Caso 1: Tio é vermelho
                    uncle.red = False
                    new_node.parent.red = False
                    new_node.parent.parent.red = True
                    new_node = new_node.parent.parent
                else:  # Caso 2: Tio é preto
                    if new_node == new_node.parent.left:  # Se o novo
nó for filho esquerdo
                        new_node = new_node.parent
                        self.rotate_right(new_node)
                        self.rotat_count+=1
                    new_node.parent.red = False
                    new_node.parent.parent.red = True
                    self.rotat_count+=1
                    self.rotate_left(new_node.parent.parent)
            else:  # Se o pai do novo nó for filho esquerdo
                uncle = new_node.parent.parent.right  # Tio é o filho
direito do avô
                if uncle and uncle.red:  # Caso 3: Tio é vermelho
                    uncle.red = False
                    new_node.parent.red = False
                    new_node.parent.parent.red = True
                    new_node = new_node.parent.parent
                else:  # Caso 4: Tio é preto
                    if new_node == new_node.parent.right:  # Se o
novo nó for filho direito
                        new_node = new_node.parent
                        self.rotate_left(new_node)
                        self.rotat_count+=1
                    new_node.parent.red = False
                    new_node.parent.parent.red = True
                    self.rotat_count+=1
```

```
                    self.rotate_right(new_node.parent.parent)
        self.root.red = False  # Raiz tem de ser preta
for input_type in input_types:
    for elements in [20000, 40000, 60000, 80000, 100000]:

        lista = listas[input_type]

        # Árvore Binária
        tree = ArvoreBinaria(lista[0])
        start_time = time.time()
        for i in range(1, elements):
            tree.insert(lista[i])
        end_time = time.time()
        tempo = end_time - start_time

        data["Tree Type"].append("ArvoreBinaria")
        data["Input Type"].append(input_type)
        data["Elements"].append(elements)
        data["Insertion Time"].append(tempo)
        data["Rotation Count"].append(None)
        print("feito - Árvore Binária")

        # Árvore Binária de Pesquisa
        tree = BinariaPesquisa(lista[0])
        start_time = time.time()
        for i in range(1, elements):
            tree.insert(lista[i])
        end_time = time.time()
        tempo = end_time - start_time

        data["Tree Type"].append("BinariaPesquisa")
        data["Input Type"].append(input_type)
        data["Elements"].append(elements)
        data["Insertion Time"].append(tempo)
        data["Rotation Count"].append(None)
        print("feito - Árvore Binária de Pesquisa")


# Para as duas últimas árvores
for input_type in input_types:
    for elements in [200000,400000,600000,800000,1000000]:
        lista = listas[input_type]

        # Árvore AVL
        tree = ArvoreAVL(lista[0])
```

```
        start_time = time.time()
        for i in range(1, elements):
            tree.insert(lista[i])
        end_time = time.time()
        tempo = end_time - start_time

        data["Tree Type"].append("ArvoreAVL")
        data["Input Type"].append(input_type)
        data["Elements"].append(elements)
        data["Insertion Time"].append(tempo)
        data["Rotation Count"].append(tree.rotat_count)
        print("feito - Árvore AVL")


        # Árvore RB
        tree = RBTree()
        start_time = time.time()
        for i in range(0, elements):
            tree.insert(lista[i])

        end_time = time.time()
        tempo = end_time - start_time

        data["Tree Type"].append("RBTree")
        data["Input Type"].append(input_type)
        data["Elements"].append(elements)
        data["Insertion Time"].append(tempo)
        data["Rotation Count"].append(tree.rotat_count)
        print("feito - Árvore RB")

df = pd.DataFrame(data)
df.to_excel("tree_performance_rb.xlsx", index=False)
```

Anexo C - Referências

stackoverflow:

```
def print_tree(self):
        self._print_tree_recursive(self.raiz, 0)


    def _print_tree_recursive(self, node, level):
        if node is not None:
            self._print_tree_recursive(node.right, level + 1)
            print("    " * level, node.element)
            self._print_tree_recursive(node.left, level + 1)
def rotate_left(self, x):
        y = x.right  # Define y como o filho direito de x
        x.right = y.left
        if y.left is not None:
            y.left.parent = x


        y.parent = x.parent
        if x.parent is None:
            self.root = y
        elif x is x.parent.left:
            x.parent.left = y
        else:
            x.parent.right = y
        y.left = x
        x.parent = y


    def rotate_right(self, x):
        y = x.left
        x.left = y.right
        if y.right is not None:
```

```
            y.right.parent = x


        y.parent = x.parent
        if x.parent is None:
            self.root = y
        elif x is x.parent.right:
            x.parent.right = y
        else:
            x.parent.left = y
        y.right = x
        x.parent = y
```

chatgpt:

```
data = {
    "Tree Type": [],
    "Input Type": [],
    "Insertion Time": [],
    "Rotation Count": [],
    "Elements":[]
}


trees = {
    "Binary Tree": ArvoreBinaria,
```

```
    "BST": BinariaPesquisa,

    "AVL": ArvoreAVL,

    "Red-Black Tree": RBTree

}




input_types = ["A","B","C", "D"]

listas = {input_type: criar_input(1000000, input_type) for
input_type in input_types}
```