Relatório Projeto 2 V1.1 AED 2023/2024

Nome: Alexandre José Martins Rodrigues Nº Estudante: 2022249408
PL (inscrição):1 Email: uc2022249408@dei.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)*

Tree Type	Input Type	Insertion Time	Elements
ArvoreBinaria	A	16,37915087	20000
ArvoreBinaria	A	69,76183581	40000
ArvoreBinaria	A	162,645498	60000
ArvoreBinaria	A	305,9623463	80000
ArvoreBinaria	A	532,2308443	100000
BinariaPesquisa	A	12,63412833	20000
BinariaPesquisa	A	51,9954412	40000
BinariaPesquisa	A	119,2262578	60000
BinariaPesquisa	A	217,699084	80000
BinariaPesquisa	A	355,7075953	100000

Tree Type	Input Type	Insertion Time	Elements
ArvoreBinaria	C	33,69759607	20000
ArvoreBinaria	C	148,9436057	40000
ArvoreBinaria	C	380,1752338	60000
ArvoreBinaria	C	765,9026606	80000
ArvoreBinaria	C	1279,143912	100000
BinariaPesquisa	C	0,136099339	20000
BinariaPesquisa	C	0,109448433	40000
BinariaPesquisa	C	0,250670433	60000
BinariaPesquisa	C	0,298692465	80000
BinariaPesquisa	C	0,465849638	100000

Tree Type	Input Type	Insertion Time	Elements
ArvoreBinaria	B	15,93637443	20000
ArvoreBinaria	B	67,15658665	40000
ArvoreBinaria	B	158,0285039	60000
ArvoreBinaria	B	305,911917	80000
ArvoreBinaria	B	526,9225268	100000
BinariaPesquisa	B	10,06412578	20000
BinariaPesquisa	B	40,90051436	40000
BinariaPesquisa	B	96,47853208	60000
BinariaPesquisa	B	175,1312664	80000
BinariaPesquisa	B	292,2264066	100000

Tree Type	Input Type	Insertion Time	Elements
ArvoreBinaria	D	33,32440138	20000
ArvoreBinaria	D	142,0148232	40000
ArvoreBinaria	D	358,5128868	60000
ArvoreBinaria	D	702,3314989	80000
ArvoreBinaria	D	1163,02902	100000
BinariaPesquisa	D	0,053388357	20000
BinariaPesquisa	D	0,178390741	40000
BinariaPesquisa	D	0,250874519	60000
BinariaPesquisa	D	0,319177866	80000
BinariaPesquisa	D	0,46206975	100000

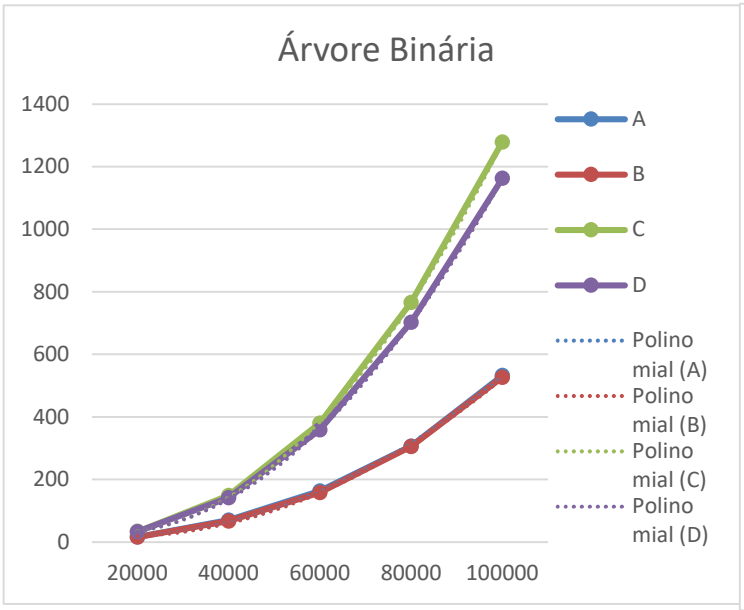
Tree Type	Input Type	Insertion Time	Rotation Count	Elements
ArvoreAVL	A	2,88760829	119449	200000
ArvoreAVL	A	6,260924339	239303	400000
ArvoreAVL	A	9,872602701	358617	600000
ArvoreAVL	A	13,58260584	478049	800000
ArvoreAVL	A	17,57118535	597268	1000000
RBTree	A	0,984103918	119435	200000
RBTree	A	2,157346964	239288	400000
RBTree	A	3,496940374	358602	600000
RBTree	A	4,6153543	478033	800000
RBTree	A	6,173992395	597252	1000000

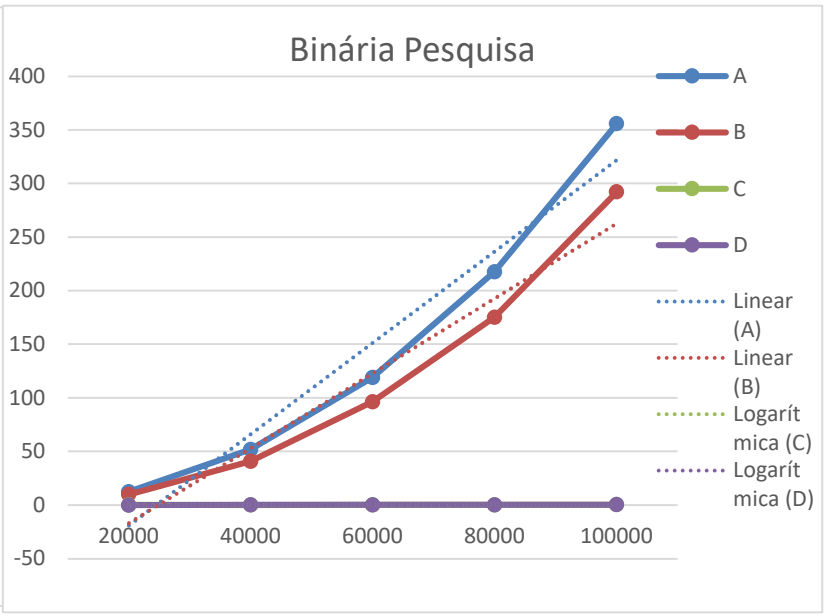
Tree Type	Input Type	Insertion Time	Rotation Count	Elements
ArvoreAVL	B	2,563715219	119103	200000
ArvoreAVL	B	6,051169634	238323	400000
ArvoreAVL	B	9,463010311	357994	600000
ArvoreAVL	B	13,02992964	477401	800000
ArvoreAVL	B	16,48031211	596863	1000000
RBTree	B	1,370542049	119089	200000
RBTree	B	1,8234303	238308	400000
RBTree	B	2,975420475	357979	600000
RBTree	B	3,967280865	477385	800000
RBTree	B	4,970087528	596847	1000000

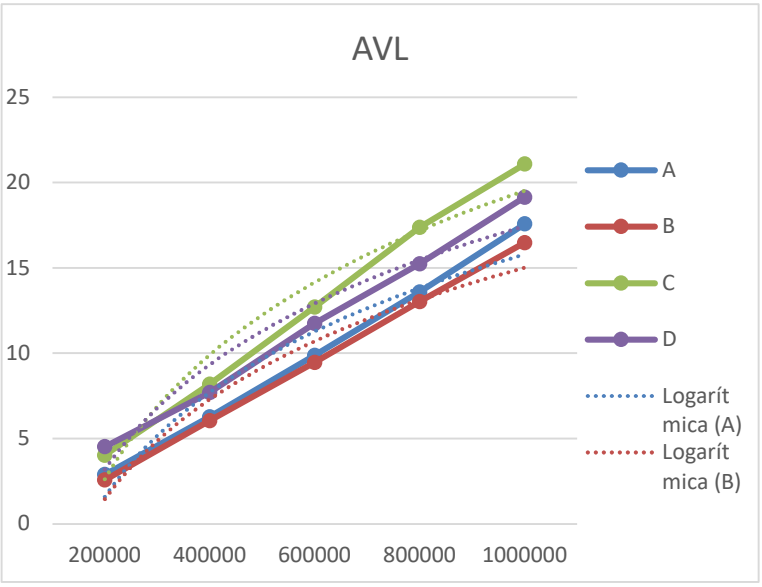
Tree Type	Input Type	Insertion Time	Rotation Count	Elements
ArvoreAVL	C	4,026592016	124351	200000
ArvoreAVL	C	8,189340115	222985	400000
ArvoreAVL	C	12,7108562	301636	600000
ArvoreAVL	C	17,37592387	365334	800000
ArvoreAVL	C	21,08476257	416617	1000000
RBTree	C	1,242748737	103986	200000
RBTree	C	2,469424963	185849	400000
RBTree	C	3,892627716	251846	600000
RBTree	C	5,326854706	304815	800000
RBTree	C	6,639460087	347261	1000000

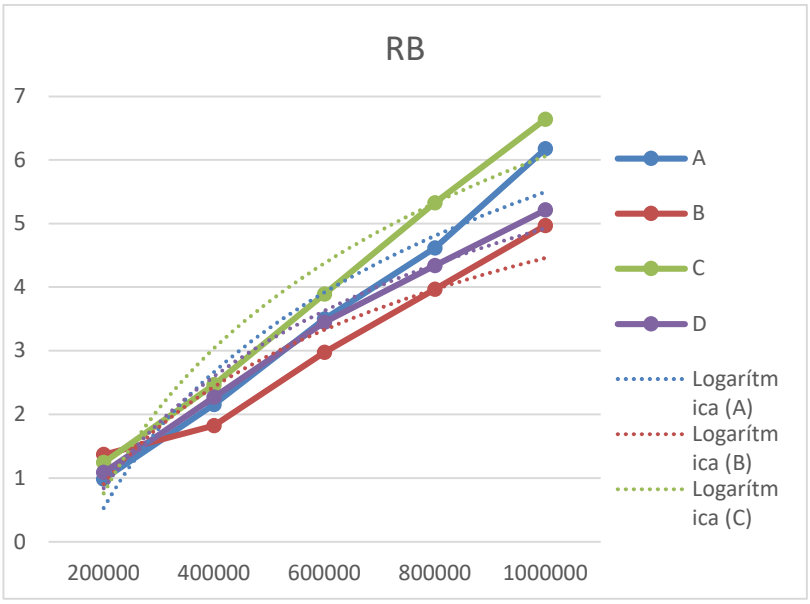
Tree Type	Input Type	Insertion Time	Rotation Count	Elements
ArvoreAVL	D	4,519394159	115166	200000
ArvoreAVL	D	7,700379372	192912	400000
ArvoreAVL	D	11,75221682	243359	600000
ArvoreAVL	D	15,23528171	273272	800000
ArvoreAVL	D	19,13880086	285527	1000000
RBTree	D	1,088003159	96221	200000
RBTree	D	2,271678209	161227	400000
RBTree	D	3,450167418	203097	600000
RBTree	D	4,33974719	228049	800000
RBTree	D	5,214368105	238318	1000000

3.Visualização de Resultados *(gráficos)*



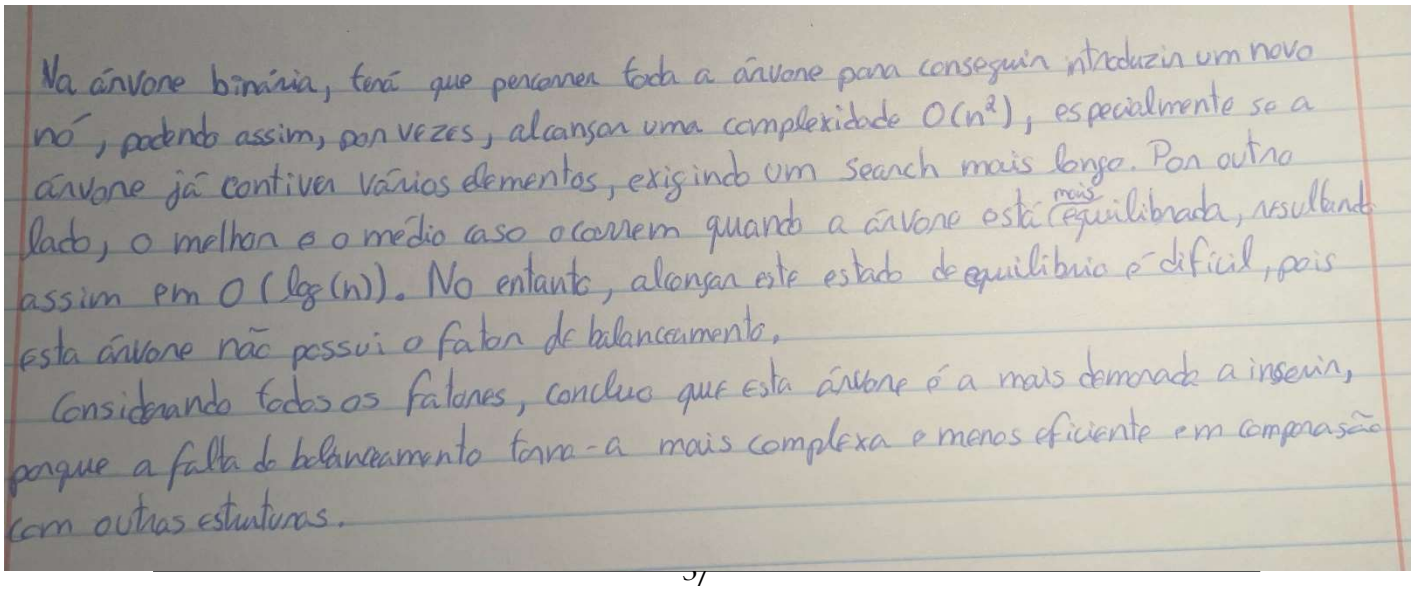






4.Conclusões *(as linhas desenhadas representam a extensão <u>máxima</u> de texto manuscrito)*

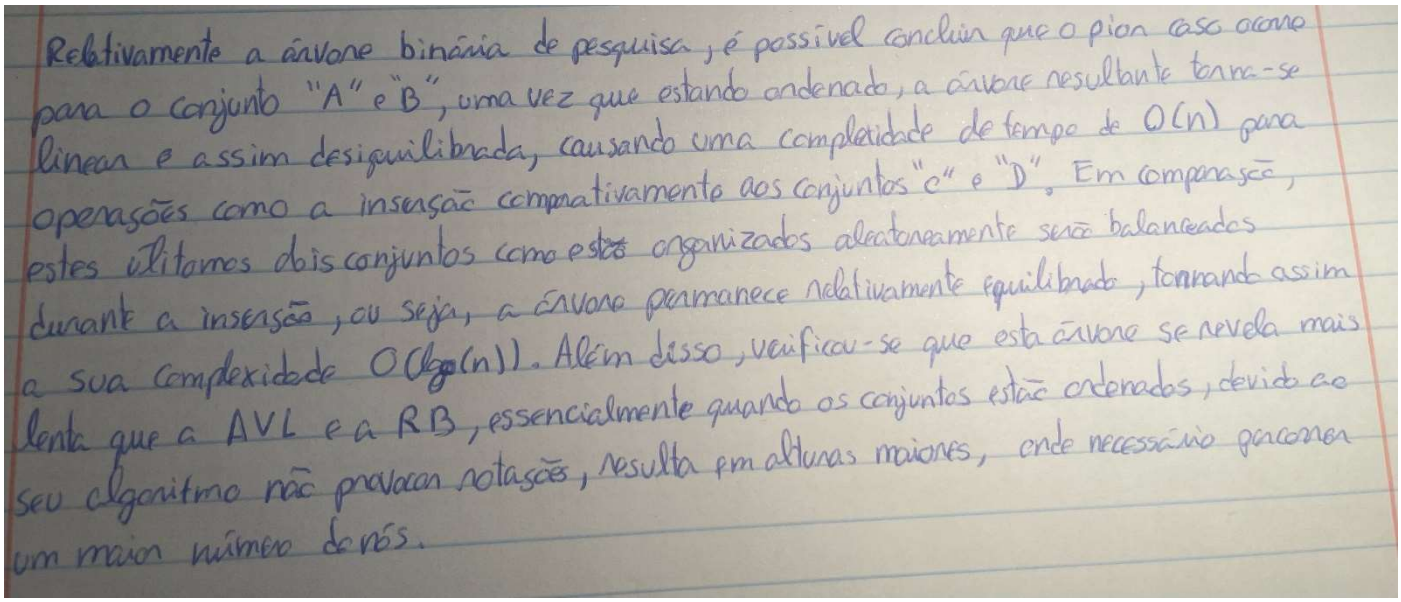
4.1Tarefa 1



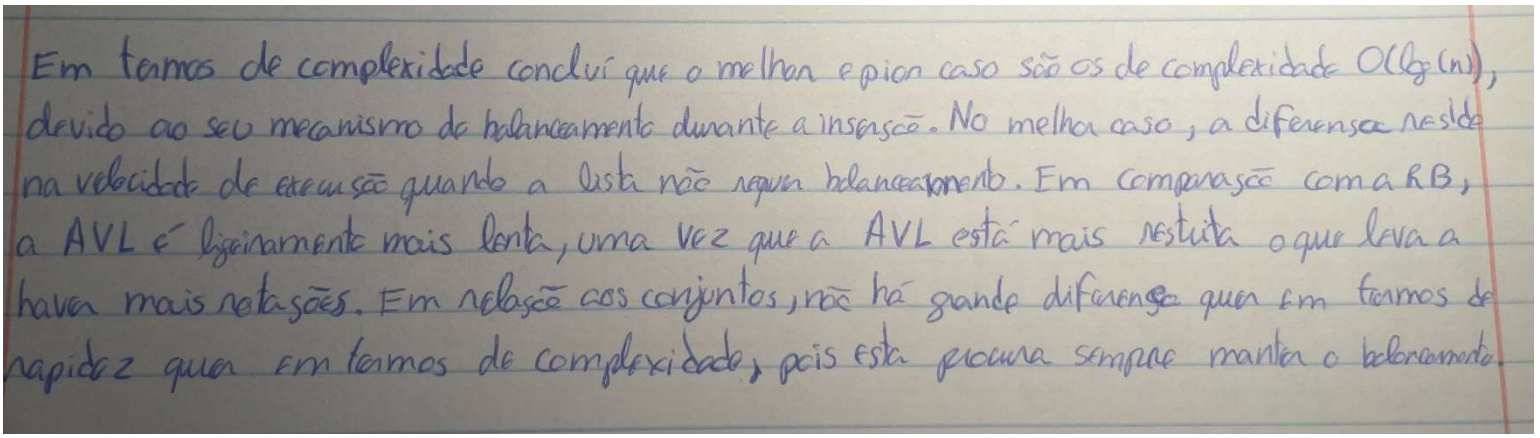
Na árvore binária, terá que percorrer toda a árvore para conseguir introduzir um novo nó, podendo assim, por vezes, alcançar uma complexidade $O(n^2)$, especialmente se a árvore já contiver vários elementos, exigindo um search mais longa. Por outro lado, o melhor e o médio caso ocorrem quando a árvore está (mais) equilibrada, resultando assim em $O(\log(n))$. No entanto, alcançar este estado de equilíbrio é difícil, pois esta árvore não possui o fator de balanceamento.

Considerando todos os fatores, concluo que esta árvore é a mais demorada a inserir, porque a falta de balanceamento torna-a mais complexa e menos eficiente em comparação com outras estruturas.

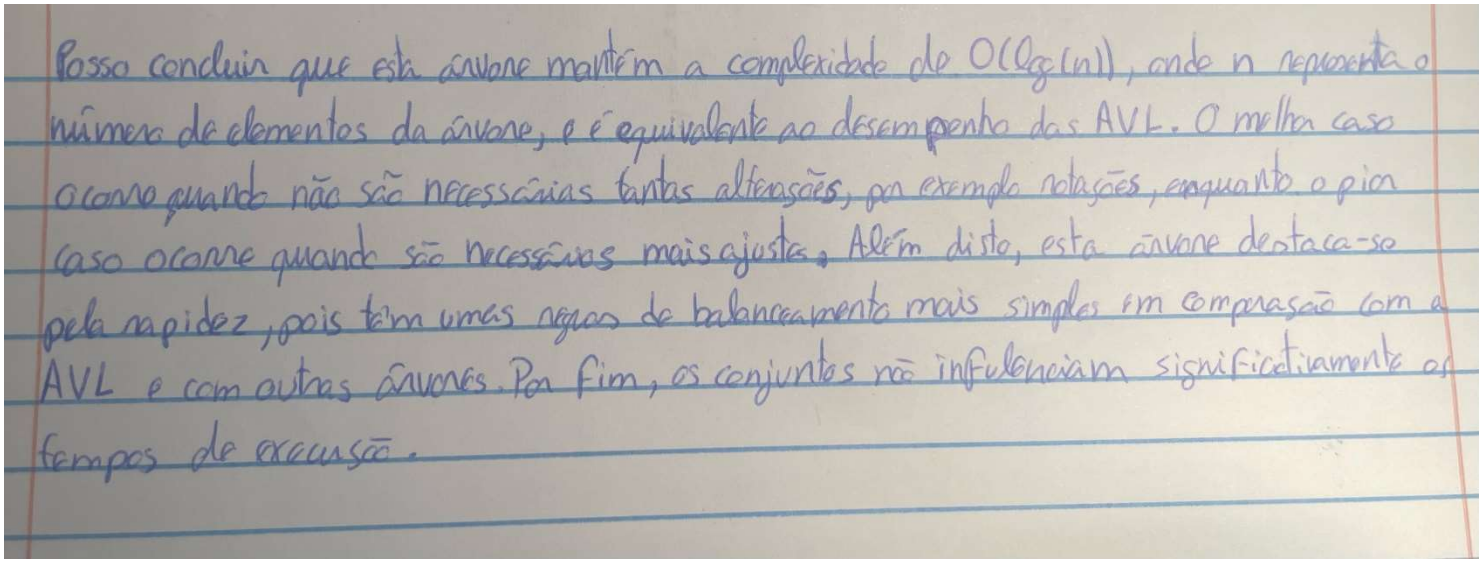
4.2. Tarefa 2


Relativamente a árvore binária de pesquisa, é possível concluir que o pior caso ocorre para o conjunto "A" e "B", uma vez que estando ordenado, a árvore resultante torna-se linear e assim desiquilibrada, causando uma complexidade de tempo de O(n) para operações como a inserção comparativamente aos conjuntos "C" e "D". Em comparação, estes últimos dois conjuntos como estão organizados aleatoriamente serão balanceados durante a inserção, ou seja, a árvore permanece relativamente equilibrada, tornando assim a sua complexidade O(log(n)). Além disso, verifica-se que esta árvore se revela mais lenta que a AVL e a RB, essencialmente quando os conjuntos estão ordenados, devido ao seu algoritmo não provocar rotações, resulta em alturas maiores, onde necessário percorrer um maior número de nós.

4.3 Tarefa 3


Em termos de complexidade concluí que o melhor e pior caso são os de complexidade O(log(n)), devido ao seu mecanismo de balanceamento durante a inserção. No melhor caso, a diferença reside na velocidade de execução quando a lista não requer balanceamento. Em comparação com a RB, a AVL é ligeiramente mais lenta, uma vez que a AVL está mais restrita o que leva a haver mais rotações. Em relação aos conjuntos, não há grande diferença quer em termos de rapidez quer em termos de complexidade, pois esta procura sempre manter o balanceamento.

4.4. Tarefa


Posso concluir que esta árvore mantém a complexidade de O(log(n)), onde n representa o número de elementos da árvore, e é equivalente ao desempenho das AVL. O melhor caso ocorre quando não são necessárias tantas alterações, por exemplo rotações, enquanto o pior caso ocorre quando são necessárias mais ajustes. Além disto, esta árvore destaca-se pela rapidez, pois têm umas regras de balanceamento mais simples em comparação com a AVL e com outras árvores. Por fim, os conjuntos não influenciam significativamente os tempos de execução.

Anexo B - Código de Autor

```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    def search(self, data):
        if self.left is not None:
            if self.left.search(data):
                return True
        if self.data == data:
            return True
        if self.right is not None:
            if self.right.search(data):
                return True
        return False

class ArvoreBinaria:
    def __init__(self, data):
        self.raiz = Node(data)

    def insert(self, data):
        atual = self.raiz
        if not atual.search(data):
            while atual:
                if atual.left is None:
                    atual.left = Node(data)
                    return
                elif atual.right is None:
                    atual.right = Node(data)
                    return
                else:
                    random_choice = random.randint(0, 1)
                    if random_choice == 0:
                        atual = atual.left
                    else:
                        atual = atual.right

    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.data)
            self._print_tree_recursive(node.left, level + 1)

# Definição da árvore binária de pesquisa
class BinariaPesquisa:
    def __init__(self, data):
        self.raiz=Node(data)
    def insert(self, data):
        atual = self.raiz
        while atual:
            if data < atual.data:
                if atual.left is None:
                    atual.left = Node(data)
                    return
                else:
                    atual = atual.left
            elif data > atual.data:
                if atual.right is None:
                    atual.right = Node(data)
                    return
                else:
                    atual = atual.right
            else:
                return
    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)

class NodeAVL:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
        self.height = 1

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

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

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

```
        if data < node.data:
            node.left = self._insert(data, node.left)
        elif data > node.data:
            node.right = self._insert(data, 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 data < node.left.data:
            self.rotat_count += 1
            return self._right_rotate(node)

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

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

        if balance < -1 and data < node.right.data:
            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 RBNode:
    def __init__(self, key):
        self.red = False
        self.parent = None
        self.key = key
        self.left = None
        self.right = None

class RBTree:
    def __init__(self):
        self.raiz = None
        self.rotat_count=0

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

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

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

```
        else:
            parent.right = new_node

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


    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.raiz = 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.raiz = y
        elif x is x.parent.right:
            x.parent.right = y
        else:
            x.parent.left = y
        y.right = x
        x.parent = y
```

Anexo C - Referências

```
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
```

chatgpt

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

arvores = {
    "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}

# Para as duas primeiras árvores
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
        tree_data["Tree Type"].append("ArvoreBinaria")
        tree_data["Input Type"].append(input_type)
        tree_data["Elements"].append(elements)
        tree_data["Insertion Time"].append(tempo)
        tree_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

        tree_data["Tree Type"].append("BinariaPesquisa")
        tree_data["Input Type"].append(input_type)
        tree_data["Elements"].append(elements)
        tree_data["Insertion Time"].append(tempo)
        tree_data["Rotation Count"].append(None)
        print("feito - Binária pesquisa2")


# 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

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

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

# Criando DataFrame e salvando em um arquivo Excel
df = pd.DataFrame(tree_data)
df.to_excel("Totalv2.xlsx", index=False)
```

chatgpt

class ArvoreAVL:

```
    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.data)
            self._print_tree_recursive(node.left, level + 1)
```

chatgpt

class RBTree:

```
    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.key, "R" if node.red else "B")
            self._print_tree_recursive(node.left, level + 1)
```

chatgpt

```
    def fix_insert(self, new_node):
```

```
        while new_node != self.raiz and new_node.parent.red:  # Enquanto o
pai do novo nó for vermelho
            if new_node.parent == new_node.parent.parent.right:  # Se 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.raiz.red = False  # Raiz tem de ser preta
```

chatgpt