


<b>Nama : Mutiara Novianti Rambe</b> <b>NIM : 064002300029</b>	 <b>Algoritma dan Pemrograman Dasar</b>	<b>Modul 10</b> <b>Nama Dosen:</b> 1. Abdul Rochman 2. Anung B. Ariwibowo
<b>Hari/Tanggal:</b> <b>Rabu/08 Mei 2024</b>		<b>Nama Aslab:</b> 1. Nathanael W. (064002100020) 2. Adrian Alfajri (064002200009)

## MODUL 10 : AVL TREE

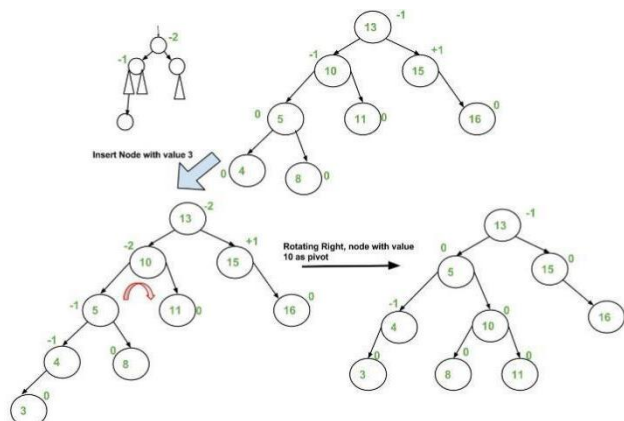
**Deskripsi Modul :** Memahami dan menerapkan ilmu struktur data dan algoritma untuk menyelesaikan masalah yang disajikan dengan menggunakan program berbasis bahasa Python.

No.	Elemen Kompetensi	Indikator Kinerja	Halaman
1.	Mampu memahami dan mengimplementasikan AVL Tree pada Python	Membuat dan memahami sebuah program yang menerapkan AVL Tree.	

### TEORI SINGKAT

AVL Tree adalah Binary Search Tree yang memiliki perbedaan tinggi / level antara sub-tree kanan dan kirinya maksimal 1. AVL Tree digunakan untuk menyeimbangkan Binary Search Tree. Dengan menggunakan AVL Tree waktu pencarian dan bentuk tree yang disederhanakan. AVL Tree dapat direpresentasikan dengan menggunakan array maupun linked list.

Pengurutan node secara manual.



## DAFTAR PERTANYAAN

1. Jelaskan perbedaan antara AVL Tree dengan Binary Search Tree!
2. Jelaskan peraturan pengurutan dalam AVL Tree!

## JAWABAN

### JAWAB DI SINI!!!

1. BST: Tidak memiliki jaminan keseimbangan, sehingga pohon dapat menjadi tidak seimbang dan menyebabkan waktu pencarian yang lama.

AVL Tree: Selalu menjaga keseimbangan dengan memastikan perbedaan ketinggian antara sub-pohon kiri dan kanan tidak lebih dari 1. Hal ini dicapai dengan operasi rotasi ketika pohon tidak seimbang setelah penyisipan atau penghapusan.

2. Peraturan pengurutan dalam AVL Tree memastikan bahwa pohon selalu dalam keadaan seimbang, sehingga waktu operasi untuk pencarian, penyisipan, dan penghapusan data selalu efisien ( $O(\log n)$ ). Aturan ini diimplementasikan dengan operasi rotasi yang menjaga perbedaan ketinggian antara sub-pohon kiri dan kanan pada setiap simpul tidak lebih dari 1.

## LAB SETUP

Hal yang harus disiapkan dan dilakukan oleh praktikan untuk menjalankan praktikum modul ini, antara lain:

1. Menyiapkan IDE untuk membangun program python (Spyder, Sublime, VSCode, dll);
2. Python sudah terinstal dan dapat berjalan dengan baik di laptop masing-masing;
3. Menyimpan semua dokumentasi hasil praktikum pada laporan yang sudah disediakan.

## ELEMEN KOMPETENSI I

**Deskripsi** : Mampu membuat program tentang AVL tree sesuai perintah yang ada.

**Kompetensi Dasar** : Membuat program yang mengimplementasikan AVL tree

### LATIHAN 1

1. Buatlah sebuah program yang mengimplementasikan AVL tree berdasarkan referensi yang diberikan, kemudian berikan detail langkah demi langkah balancing tree melalui print output pada program tersebut. Di mana angka pertama yang akan di input ke dalam AVL Tree adalah digit ke dua dan tiga pada nim (misal: 064, maka yang diinput adalah 64), dan digit terakhir adalah dua digit akhir nim. (Jumlah node dalam tree minimal 6).
2. Setiap program wajib menampilkan nama dan nim di bagian atas program.

```

C:\Users\PC\Documents\Tugas Univ\Algoritma & Phyton\Algo>avl.py
==R----16
==  L----5
==  |    L----4
==  R----22
==      L----20
==      R----64
postOrder
[64, 5, 16, 22, 4, 20]
preOrder:
16 ,5 ,4 ,22 ,20 ,64 ,
cari angka? :64
64 ditemukan.

```

## Source Code

```

#Mutiara Novianti Rambe
#064002300029
import sys

class Node:
    def __init__(self, data):
        self.data = data
        self.parent = None
        self.left = None
        self.right = None
        self.bf = 0

class AVLTree:

    def __init__(self):
        self.root = None

    def __printHelper(self, currPtr, indent, last):
        # Print the tree structure on the screen
        if currPtr != None:
            sys.stdout.write(indent)
            if last:
                sys.stdout.write("R----")
                indent += "  "
            else:
                sys.stdout.write("L----")
                indent += "|  "
            print(currPtr.data)

```

```

        self.__printHelper(currPtr.left, indent, False)
        self.__printHelper(currPtr.right, indent, True)

def __searchTreeHelper(self, node, key):
    if node == None or key == node.data:
        return node

    if key < node.data:
        return self.__searchTreeHelper(node.left, key)
    return self.__searchTreeHelper(node.right, key)

def __deleteNodeHelper(self, node, key):
    if node == None:
        return node
    elif key < node.data:
        node.left = self.__deleteNodeHelper(node.left, key)
    elif key > node.data:
        node.right = self.__deleteNodeHelper(node.right, key)
    else:
        if node.left == None and node.right == None:
            node = None

        # Case 2: Node has only one child
        elif node.left == None:
            temp = node
            node = node.right

        elif node.right == None:
            temp = node
            node = node.left

        # Case 3: Has both children
        else:
            temp = self.minimum(node.right)
            node.data = temp.data
            node.right = self.__deleteNodeHelper(node.right, temp.data)

        # Update the balance factor of the node (**correction**)
        self.__updateBalance(node)

    return node

# Update the balance factor of the node (**correction**)
def __updateBalance(self, node):
    if node.bf > 1 or node.bf < -1:
        self.__rebalance(node)

```

```
        return

    if node.parent != None:
        if node == node.parent.left:
            node.parent.bf -= 1

        if node == node.parent.right:
            node.parent.bf += 1

        if node.parent.bf != 0:
            self.__updateBalance(node.parent)

# Rebalance the tree
def __rebalance(self, node):
    if node.bf > 0:
        if node.right.bf < 0:
            self.rightRotate(node.right)
            self.leftRotate(node)
        else:
            self.leftRotate(node)
    elif node.bf < 0:
        if node.left.bf > 0:
            self.leftRotate(node.left)
            self.rightRotate(node)
        else:
            self.rightRotate(node)

def __preOrderHelper(self, node, result):
    if node != None:
        result.append(node.data)
        self.__preOrderHelper(node.left, result)
        self.__preOrderHelper(node.right, result)

def __postOrderHelper(self, node, result):
    if node is not None:
        self.__postOrderHelper(node.left, result)
        self.__postOrderHelper(node.right, result)
        result.append(node.data)

# Pre-Order traversal
# Node->Left Subtree->Right Subtree
def preorder(self):
    result = []
    self.__preOrderHelper(self.root, result)
    return result
```

```
# Post-Order traversal
# Left Subtree -> Right Subtree -> Node
def postorder(self):
    result = []
    self.__postOrderHelper(self.root, result)
    return result

def insert(self, data):
    if not self.root:
        self.root = Node(data)
    else:
        self.__insertHelper(self.root, data)

def __insertHelper(self, node, data):
    if data < node.data:
        if node.left is None:
            node.left = Node(data)
            node.left.parent = node
            self.__updateBalance(node.left)
        else:
            self.__insertHelper(node.left, data)
    elif data > node.data:
        if node.right is None:
            node.right = Node(data)
            node.right.parent = node
            self.__updateBalance(node.right)
        else:
            self.__insertHelper(node.right, data)

def delete(self, data):
    self.root = self.__deleteNodeHelper(self.root, data)

def minimum(self, node):
    while node.left != None:
        node = node.left
    return node

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

    y.right = z
    z.left = T3

    if T3 is not None:
        T3.parent = z
```

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

z.parent = y
z.bf = z.bf + 1 - min(y.bf, 0)
y.bf = y.bf + 1 + max(z.bf, 0)

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

    y.left = z
    z.right = T2

    if T2 is not None:
        T2.parent = z

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

    z.parent = y
    z.bf = z.bf - 1 - max(y.bf, 0)
    y.bf = y.bf - 1 + min(z.bf, 0)

def main():
    avl = AVLTree()
    digits = [64, 2, 15, 9, 8, 29] # NIM: 064002300029

    for digit in digits:
        avl.insert(digit)

    avl._AVLTree__printHelper(avl.root, "", True) # Accessing private method using name mangling

    print("\nPostOrder")
```

```
print(avl.postorder())

print("PreOrder")
print(avl.preorder())

if __name__ == "__main__":
    main()
```

Screenshot

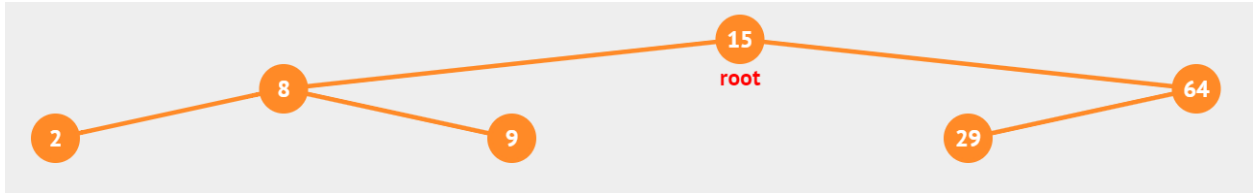
```
R-----15
  L-----8
  |   L-----2
  |   R-----9
  R-----64
    L-----29

PostOrder
[2, 9, 8, 29, 64, 15]
PreOrder
[15, 8, 2, 9, 64, 29]
```



## LATIHAN 2

1. Gambarlah hasil AVL Tree yang terbentuk menggunakan tools shape yang ada di ms. word. Di mana angka pertama yang akan di input ke dalam AVL Tree adalah digit ke dua dan tiga pada nim (misal: 064, maka yang diinput adalah 64), dan digit terakhir adalah dua digit akhir nim. (Jumlah node dalam tree minimal 6).



## KESIMPULAN

Saya mengetahui bahwa AVL Tree adalah varian dari Binary Search Tree (BST) yang mempertahankan keseimbangan tinggi antara sub-tree kiri dan kanan, dengan perbedaan maksimal satu tingkat. dan melalui praktikum ini saya belajar membuat program tentang AVL Tree.

## CEKLIST

1. Memahami dan mengimplementasikan AVL tree pada Python (✓)

## REFERENSI

<https://www.geeksforgeeks.org/avl-tree-set-1-insertion/>  
<https://algorithmtutor.com/Data-Structures/Tree/AVL-Trees/>  
<https://www.programiz.com/dsa/avl-tree>  
<https://pythonwife.com/avl-tree-in-python/>  
<https://www.javatpoint.com/avl-tree>