TUGAS MANDIRI
ALGORITHMA AVL TREE
PERANCANGAN DAN ANALISIS ALGORITMA
INF1.62.4001

DOSEN PENGAMPU:

Randi Proska Sandra, M.Sc

OLEH:

Nama: Ranny Erzitha

NIM: 21343013

INFORMATIKA

PROGRAM STUDI INFORMATIKA

DEPARTERMEN TEKNIK ELETRONIKA

FAKULTAS TEKNI

UNIVERSITAS NEGERI PADANG

2023

Algorithma AVL Tree

A. AVL Tree

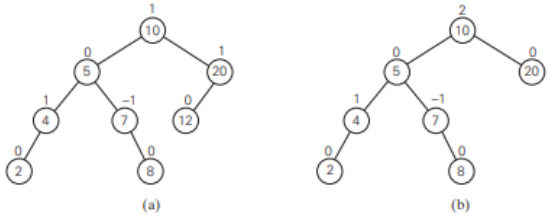
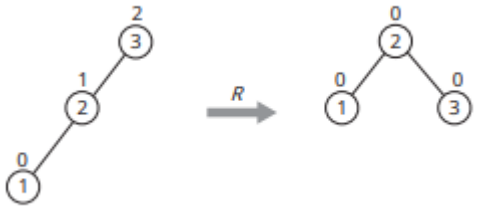


FIGURE 6.2 (a) AVL tree. (b) Binary search tree that is not an AVL tree. The numbers above the nodes indicate the nodes' balance factors.

Avl Tree adalah pohon pencarian biner dimana *balance factor d*ari setiap simpul, dijadikan sebagai pembeda antara ketinggian dari subpohon kiri dan kanan, yaitu) atau +1 atau -1 (Tinggi dari pohon yang bernilai 0 didefenisikan sebagai -1. Faktor keseimbangan ini juga dapat digunakan sebagai pembeda antara jumlah level dengan ketinggian subpohon kiri dan kanan node).

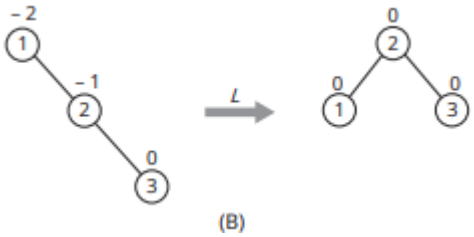
Dalam penambahan atau penyisipan akar pohon yang baru, diperlukan penyeimbangan akar pohon tersebut dengan **rotasi.** Rotasi adalah transformasi lokal dari subpohon yang berakar pada simpul yang keseimbangannya menjadi +2 atau −2. Jika ada beberapa simpul seperti itu, kami memutar pohon yang berakar pada simpul yang tidak seimbang yang paling dekat dengan daun yang baru disisipkan. Terdapat 4 macam rotasi, diantaranya:

- **Single Right Rotation/ R-Rotasi (**1 Rotasi ke Kanan**)**



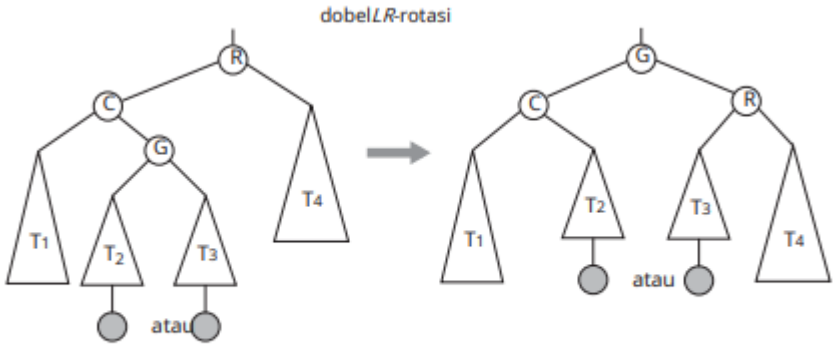
rotasi ini dilakukan setelah kunci baru dimasukkan ke dalam subpohon kiri dari anak kiri dari pohon yang akarnya memiliki keseimbangan +1 sebelum penyisipan.

- **Single Left Rotation/L-Rotasi (**1 Rotasi ke Kiri**)**



L-rotasi, adalah bayangan cermin dari single tersebutR -rotasi. Ini dilakukan setelah kunci baru dimasukkan ke subpohon kanan dari anak kanan dari pohon yang akarnya memiliki keseimbangan −1 sebelum penyisipan.

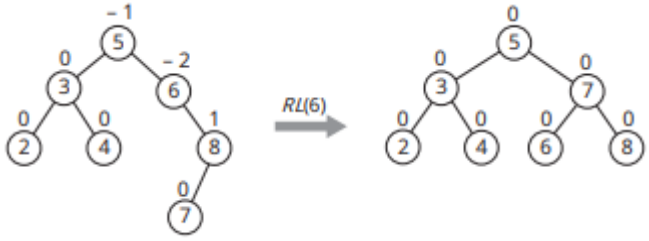
- **Doble Left-Right Rotation/LR-Rotation (**Rotasi Ganda Kiri-Kanan**)**



GAMBAR 6.5 Bentuk umum dari ganda ***LR***-rotasi di pohon AVL. Node yang diarsir adalah yang terakhir dimasukkan. Itu bisa di subpohon kiri atau di subpohon kanan dari cucu akar.

LR-Rotation adalah kombinasi dari dua putaran, L-rotasi subpohon kiri akar R dilakukan lebih dahulu diikuti oleh R-rotasi pohon baru berakar pada R. Ini dilakukan setelah kunci baru dimasukkan ke subpohon kanan dari anak kiri pohon yang akarnya memiliki saldo +1 sebelum penyisipan

- **Doble Right-Left Rotation (**Rotasi Ganda Kanan-Kiri**)**



RL-Rotation adalah bayangan cermin dari ganda LR -rotasi

Rotasi bukanlah transformasi sepele, meskipun dapat dilakukan dalam waktu yang konstan. Seperti pohon pencarian lainnya, karakteristik kritis dari algorithma AVL Tree ini adalah tinggi pohon, pencarian di pohon AVL membutuhkan, rata-rata, jumlah perbandingan yang hampir sama dengan pencarian di array yang diurutkan dengan pencarian biner.

Pengoperasian penghapusan kunci dalam AVL Tree jauh lebih sulit daripada penyisipan, tetapi untungnya ternyata berada di kelas efisiensi yang sama dengan penyisipan, yaitu logaritmik. Namun, karakteristik efisiensi yang paling mengesankan dari algoritma ini adalah nilainya. Kelemahan dari pohon AVL adalah rotasi yang sering dan kebutuhan untuk menjaga keseimbangan node-nodenya.

B. Pseudocode

```
Deklarasikan node AVL:
   - kiri: AVLNode
   - kanan: AVLNode
   - nilai: int
   - tinggi: int

Deklarasikan sebuah AVLTree:
   - root: AVLNode

Deklarasikan fungsi getHeight(node):
   - Jika node = null, kembalikan 0
   - Kembalikan node.tinggi

Deklarasikan fungsi getBalance(node):
   - Jika node = null, kembalikan 0
   - Kembalikan getHeight(node.kiri) - getHeight(node.kanan)

Deklarasikan fungsi rotateRight(node):
   - Simpan nodeKiri = node.kiri
   - Simpan nodeKanan = nodeKiri.kanan
   - nodeKiri.kanan = node
   - node.kiri = nodeKanan
   - Update tinggi node dan nodeKiri
   - Kembalikan nodeKiri

Deklarasikan fungsi rotateLeft(node):
   - Simpan nodeKanan = node.kanan
   - Simpan nodeKiri = nodeKanan.kiri
   - nodeKanan.kiri = node
   - node.kanan = nodeKiri
   - Update tinggi node dan nodeKanan
   - Kembalikan nodeKanan

Deklarasikan fungsi rebalance(node):
   - Update tinggi node
   - Jika getBalance(node) = -2:
   - Jika getBalance(node.kanan) = 1, rotasi kanan pada node.kanan
   - Kembalikan rotateLeft(node)

   - Jika getBalance(node) = 2:
   - Jika getBalance(node.kiri) = -1, rotasi kiri pada node.kiri
   - Kembalikan rotateRight(node)
   - Kembalikan node

Deklarasikan fungsi insert (node, nilai):
   - Jika node = null, kembalikan AVLNode(nilai)
   - Jika nilai < node.nilai, node.kiri = insert(node.kiri, nilai)
   - Jika nilai > node.nilai, node.kanan = insert(node.kanan,
     nilai)
   - Kembalikan rebalance(node)

Deklarasikan fungsi delete (node, nilai):
   - Jika node = null, kembalikan null
```

```
- Jika nilai < node.nilai, node.kiri =
  delete(node.kiri, nilai)
- Jika nilai > node.nilai, node.kanan =
  delete(node.kanan, nilai)
- Jika nilai = node.nilai:
    - Jika node.kiri = null, kembalikan node.kanan
    - Jika node.kanan = null, kembalikan node.kiri
    - Simpan nodeKananTerkecil = node.kanan
    - Selama nodeKananTerkecil.kiri! = null,
      nodeKananTerkecil = nodeKananTerkecil.kiri
- node.nilai = nodeKananTerkecil.nilai
- node.kanan = delete(node.kanan,
  nodeKananTerkecil.nilai)
- Kembalikan rebalance(node)
```

Penjelasan:

Dalam pseudocode tersebut, terdapat beberapa fungsi dan variabel yang dideklarasikan untuk melakukan operasi pada AVL Tree, antara lain:

1. AVL Node: merupakan struktur data simpul AVL Tree yang memiliki tiga atribut, yaitu kiri (left), kanan (right), nilai (value), dan tinggi (height). Atribut kiri dan kanan masing-masing merupakan pointer ke simpul anak kiri dan kanan dari simpul AVLNode tersebut, sedangkan atribut nilai dan tinggi adalah data yang menyimpan nilai dan tinggi simpul AVLNode tersebut.
2. AVL Tree: merupakan struktur data pohon AVL Tree yang memiliki satu atribut, yaitu root, yang merupakan pointer ke simpul akar (root node) dari pohon AVL Tree tersebut.
3. getHeight (node): adalah fungsi yang digunakan untuk menghitung tinggi (height) simpul AVLNode. Fungsi ini menerima parameter node yang merupakan simpul AVLNode, kemudian mengembalikan nilai tinggi simpul tersebut. Jika node yang diberikan bernilai null, maka fungsi ini akan mengembalikan nilai 0.
4. getBalance (node): adalah fungsi yang digunakan untuk menghitung keseimbangan (balance factor) simpul AVLNode. Fungsi ini menerima parameter node yang merupakan simpul AVLNode, kemudian mengembalikan nilai selisih antara tinggi anak kiri dan tinggi anak kanan dari simpul tersebut. Jika node yang diberikan bernilai null, maka fungsi ini akan mengembalikan nilai 0.
5. rotateRight (node): adalah fungsi yang digunakan untuk melakukan rotasi kanan (right rotation) pada simpul AVLNode. Fungsi ini menerima parameter node yang merupakan simpul AVLNode, kemudian mengembalikan simpul AVLNode baru yang telah dirotasi. Rotasi kanan pada simpul AVLNode dilakukan dengan menukar simpul kiri dengan simpul induk, kemudian memperbaharui tinggi simpul AVLNode yang terlibat dalam rotasi.
6. rotateLeft (node): adalah fungsi yang digunakan untuk melakukan rotasi kiri (left rotation) pada simpul AVLNode. Fungsi ini menerima parameter node yang merupakan simpul AVLNode, kemudian mengembalikan simpul AVLNode baru yang telah dirotasi. Rotasi kiri pada simpul AVLNode

dilakukan dengan menukar simpul kanan dengan simpul induk, kemudian memperbaharui tinggi simpul AVLNode yang terlibat dalam rotasi.

7. Rebalance (node): adalah fungsi yang digunakan untuk melakukan rekonsiliasi keseimbangan pohon AVL Tree setelah terjadi perubahan pada simpul. Fungsi ini menerima parameter node yang merupakan simpul AVLNode, kemudian mengembalikan simpul AVLNode baru yang telah seimbang. Pada dasarnya, fungsi ini memeriksa keseimbangan simpul AVLNode dan melakukan rotasi pada simpul yang tidak seimbang sesuai dengan aturan

C. Program AVL Tree dengan Python

```python
import random
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None


class Pohon:
    def __init__(self):
        self.node = None
        self.height = -1
        self.balance = 0

    def get_height(self):
        if self.node:
            return self.node.height
        else:
            return 0

    def insert(self, key):
        tree = self.node
        new_node = Node(key)

        if tree is None:
            self.node = new_node
            self.node.left = Pohon()
            self.node.right = Pohon()

        elif key < tree.key:
            self.node.left.insert(key)

        elif key > tree.key:
            self.node.right.insert(key)

        self.re_balance_tree()

    def re_balance_tree(self):
        self.update_heights(False)
        self.update_balances(False)
        while self.balance < -1 or self.balance > 1:
            if self.balance > 1:
                if self.node.left.balance < 0:
                    self.node.left.rotate_left()
                    self.update_heights()
                    self.update_balances()
                self.rotate_right()
                self.update_heights()
                self.update_balances()
```

```
        if self.balance < -1:
            if self.node.right.balance > 0:
                self.node.right.rotate_right()
                self.update_heights()
                self.update_balances()
            self.rotate_left()
            self.update_heights()
            self.update_balances()

def rotate_right(self):
    root = self.node
    left_child = self.node.left.node
    right_child = left_child.right.node

    self.node = left_child
    left_child.right.node = root
    root.left.node = right_child

def rotate_left(self):
    root = self.node
    right_child = self.node.right.node
    left_child = right_child.left.node

    self.node = right_child
    right_child.left.node = root
    root.right.node = left_child

def update_heights(self, recurse=True):
    if not self.node is None:
        if recurse:
            if self.node.left is not None:
                self.node.left.update_heights()
            if self.node.right is not None:
                self.node.right.update_heights()

        self.height = max(self.node.left.height,
                          self.node.right.height) + 1
    else:
        self.height = -1

def update_balances(self, recurse=True):
    if not self.node is None:
        if recurse:
            if self.node.left is not None:
                self.node.left.update_balances()
            if self.node.right is not None:
                self.node.right.update_balances()
```

```
            self.balance = self.node.left.height - self.node.right.height
        else:
            self.balance = 0

    def check_balanced(self):
        if self is None or self.node is None:
            return True

        self.update_heights()
        self.update_balances()
        return ((abs(
            self.balance) < 2) and self.node.left.check_balanced() and
                self.node.right.check_balanced())

    def tree_in_order_traversal(self):
        if self.node is None:
            return []
        nodes_list = []
        l = self.node.left.tree_in_order_traversal()
        for i in l:
            nodes_list.append(i)

        nodes_list.append(self.node.key)

        l = self.node.right.tree_in_order_traversal()
        for i in l:
            nodes_list.append(i)
        return nodes_list

    def logical_successor(self, node):
        '''
        Find the smallese valued node in RIGHT child
        '''
        node = node.right.node
        if node != None: # jika node tidak None

            while node.left != None:
                print("LS: traversing: " + str(node.key))
                if node.left.node == None:
                    return node
                else:
                    node = node.left.node
        return node

    def print_tree_as_tree_shape(self, node=None, level=0):
        if not node:
            node = self.node
```

```
        if node.right.node:
            self.print_tree_as_tree_shape(node.right.node, level + 1)
            print(('\t' * level), (' / '))
        print(('\t' * level), node.key)

        if node.left.node:
            print(('\t' * level), (' \\ '))
            self.print_tree_as_tree_shape(node.left.node, level + 1)

    def delete(self, key = 0):
        key = int(key)
        # mencoba menghapus node yang di pilih
        if self.node != None:
            if int(self.node.key) == int(key):
                print("Deleting ... " + str(key))
                if self.node.left.node == None and self.node.right.node == None:
                    self.node = None # leaves can be killed at will

                elif self.node.left.node == None:
                    self.node = self.node.right.node
                elif self.node.right.node == None:
                    self.node = self.node.left.node
                else:
                    replacement = self.logical_successor(self.node)
                    if replacement != None: # sanity check
                        print("Found replacement for " + str(key) + " -> " + str(replacement.key))
                        self.node.key = replacement.key
                        self.node.right.delete(replacement.key)

                self.re_balance_tree()
                return
            elif int(key) < int(self.node.key):
                self.node.left.delete(key)
            elif int(key) > int(self.node.key):
                self.node.right.delete(key)

            self.re_balance_tree()
        else:
            return


def create_random_node_list(n=10):

    random_node_list = random.sample(range(1, 100), n)
    print("Input :", random_node_list, "\n")
    return random_node_list


def create_avl_tree(node_list):
```

```
def create_avl_tree(node_list):

    tree = Pohon()
    for node in node_list:
        tree.insert(node)
    return tree

#if __name__ == "__main__":


loop = True
pilihan = 0
tree = Pohon()
avl = tree

while loop == True:
    print("Pilih Menu Untuk Manipulasikan AVL Tree")
    print("1.Tambah data pada avl tree")
    print("2.Hapus data pada avl tree")
    print("3.Random data")
    print("4.keluar")

    pilihan = int(input("Pilih : "))
    if(pilihan == 1):
        v_input = input("Masukan Nilai Value (pisahkan dengan koma) : ")
        vals = v_input.split(',')
        for val in vals:
            avl.insert(val)

        avl.print_tree_as_tree_shape()
    elif (pilihan == 2):

        print(avl.tree_in_order_traversal())
        k = input("Masukan nilai yang akan di hapus : ")
        avl.delete(int(k))
        avl.print_tree_as_tree_shape()

    elif (pilihan == 3):
            avl = create_avl_tree(create_random_node_list(8))
            avl.print_tree_as_tree_shape()
            print('\n')
            print(avl.tree_in_order_traversal())
    elif (pilihan == 4):
        loop = False
```

Output

```
Pilih Menu Untuk Manipulasikan AVL Tree
1.Tambah data pada avl tree
2.Hapus data pada avl tree
3.Random data
4.keluar
Pilih : 1
Masukan Nilai Value (pisahkan dengan koma) : 20,30,10,50
                50
          /
         30
  /
 20
  \
         10
Pilih Menu Untuk Manipulasikan AVL Tree
1.Tambah data pada avl tree
2.Hapus data pada avl tree
3.Random data
4.keluar
Pilih : 2
['10', '20', '30', '50']
Masukan nilai yang akan di hapus : 10
Deleting ... 10
         50
  /
 30
  \
         20
Pilih Menu Untuk Manipulasikan AVL Tree
1.Tambah data pada avl tree
```

```
Pilih Menu Untuk Manipulasikan AVL Tree
1.Tambah data pada avl tree
2.Hapus data pada avl tree
          /
         40
          \
                  29
                   \
                           14


[14, 29, 40, 49, 65, 77, 80, 89]
Pilih Menu Untuk Manipulasikan AVL Tree
1.Tambah data pada avl tree
2.Hapus data pada avl tree
3.Random data
4.keluar
Pilih : 4
PS C:\Users\ACER>
```

Penjelasan AVL Tree

Program di atas merupakan implementasi dari struktur data AVL Tree dalam bahasa pemrograman Python. AVL Tree adalah sebuah pohon biner yang diatur agar keseimbangannya dipertahankan, sehingga operasi insert, delete, dan search dapat dilakukan pada waktu O(log n).

Dalam program ini, terdapat dua class yaitu Node dan Pohon. Class Node merepresentasikan setiap simpul pada AVL Tree, yang memiliki atribut kunci (key) dan dua referensi ke simpul anak kiri (left) dan simpul anak kanan (right).

Class Pohon merepresentasikan AVL Tree secara keseluruhan, yang memiliki beberapa atribut yaitu node (akar AVL Tree), height (tinggi AVL Tree), dan balance (keseimbangan AVL Tree). Pada class ini juga terdapat beberapa method untuk melakukan operasi pada AVL Tree seperti insert, delete, rotate_left, rotate_right, update_heights, update_balances, check_balanced, dan tree_in_order_traversal.

D. Analisis Waktu Algorithma AVL Tree

1. Analisis waktu algoritma AVL Tree berdasarkan operasi/instruksi yang dieksekusi:

Waktu algoritma AVL Tree tergantung pada operasi yang dilakukan pada pohon AVL.

- Insert: Pada saat menyisipkan sebuah node, waktu yang dibutuhkan sebanding dengan ketinggian (height) dari pohon. Dalam kasus terburuk, waktu yang dibutuhkan adalah O(log n), di mana n adalah jumlah node dalam pohon AVL. Namun, jika pohon AVL tidak seimbang, maka waktu yang dibutuhkan untuk melakukan rotasi untuk menjaga keseimbangan pohon bisa memakan waktu lebih lama.
- Delete: Pada saat menghapus sebuah node, waktu yang dibutuhkan juga sebanding dengan ketinggian pohon. Dalam kasus terburuk, waktu yang dibutuhkan adalah O(log n). Namun, jika pohon AVL tidak seimbang, maka waktu yang dibutuhkan untuk melakukan rotasi untuk menjaga keseimbangan pohon bisa memakan waktu lebih lama.
- Search: Pada saat mencari sebuah node, waktu yang dibutuhkan juga sebanding dengan ketinggian pohon. Dalam kasus terburuk, waktu yang dibutuhkan adalah O(log n).

Dalam keseluruhan, waktu yang dibutuhkan untuk operasi AVL Tree memiliki kompleksitas O(log n), di mana n adalah jumlah node dalam pohon AVL. Namun, jika pohon AVL tidak seimbang, waktu yang dibutuhkan untuk melakukan rotasi untuk menjaga keseimbangan pohon bisa memakan waktu lebih lama. Oleh karena itu, sangat penting untuk menjaga keseimbangan pohon AVL secara terus-menerus untuk memastikan bahwa waktu operasi tetap efisien.

2. Analisis waktu algoritma AVL Tree berdasarkan jumlah operasi abstrak:

Waktu algoritma AVL Tree dapat dianalisis berdasarkan jumlah operasi abstrak yang dilakukan pada setiap operasi dasar. Berikut adalah analisis waktu algoritma AVL Tree berdasarkan jumlah operasi abstrak:

1. Insert

- Pada setiap operasi **insert**, algoritma AVL Tree melakukan operasi abstrak yang setara dengan traversal tree dengan waktu O(log n).
- Selain itu, pada setiap operasi **insert**, algoritma AVL Tree juga melakukan operasi rotasi untuk memperbaiki keseimbangan tree dengan waktu O(1) untuk setiap rotasi.
- Oleh karena itu, waktu total operasi **insert** pada algoritma AVL Tree adalah O(log n).

2. Delete

- Pada setiap operasi **delete**, algoritma AVL Tree juga melakukan traversal tree dengan waktu O(log n).
- Selain itu, pada setiap operasi **delete**, algoritma AVL Tree juga melakukan operasi rotasi untuk memperbaiki keseimbangan tree dengan waktu O(1) untuk setiap rotasi.
- Oleh karena itu, waktu total operasi **delete** pada algoritma AVL Tree adalah O(log n).

3. Search

- Pada setiap operasi **search**, algoritma AVL Tree juga melakukan traversal tree dengan waktu O(log n).
- Oleh karena itu, waktu total operasi **search** pada algoritma AVL Tree adalah O(log n).

Dengan demikian, waktu algoritma AVL Tree dapat dinyatakan sebagai O(log n) untuk setiap operasi dasarnya. Namun, perlu dicatat bahwa analisis ini hanya berdasarkan pada jumlah operasi abstrak yang dilakukan dan dapat berbeda dengan waktu sebenarnya yang tergantung pada implementasi dan lingkungan eksekusi.

3. Analisis waktu algoritma AVL Tree pada pendekatan best-case, worst-case, dan average-case:

Sebuah struktur data tree yang diimplementasikan dengan cara menyeimbangkan node secara otomatis sehingga ketinggian subtree kanan dan kiri selalu berbeda tidak lebih dari 1. Waktu eksekusi dari sebuah operasi pada AVL tree tergantung pada banyak faktor seperti jumlah node, tinggi tree, dan struktur tree. Oleh karena itu, analisis waktu algoritma AVL tree dibagi menjadi tiga kasus, yaitu best-case, worst-case, dan average-case.

Best-case: Kasus terbaik terjadi ketika tree sudah seimbang atau minimal terpaut dalam tinggi antara subtree kiri dan kanan tidak lebih dari 1. Pada kasus ini, waktu yang dibutuhkan untuk operasi tree tergantung pada operasi yang dilakukan. Dalam kasus ini, operasi seperti pencarian, penyisipan, dan penghapusan membutuhkan waktu O(log n), di mana n adalah jumlah node dalam tree. Dalam kasus terbaik, tidak perlu dilakukan rotasi, sehingga operasi memiliki waktu eksekusi yang cepat.

Worst-case: Kasus terburuk terjadi ketika tree sangat tidak seimbang dan ketinggian tree mencapai n. Dalam kasus ini, operasi seperti pencarian, penyisipan, dan penghapusan membutuhkan waktu

O(n), di mana n adalah jumlah node dalam tree. Dalam kasus terburuk, diperlukan banyak rotasi untuk menyeimbangkan tree, sehingga operasi memiliki waktu eksekusi yang lambat.

Average-case: Kasus rata-rata terjadi ketika tree relatif seimbang dan ketinggiannya adalah log n. Pada kasus ini, operasi seperti pencarian, penyisipan, dan penghapusan membutuhkan waktu O(log n), di mana n adalah jumlah node dalam tree. Dalam kasus rata-rata, jumlah rotasi yang dibutuhkan untuk menyeimbangkan tree lebih sedikit daripada pada kasus terburuk, sehingga operasi memiliki waktu eksekusi yang lebih cepat daripada pada kasus terburuk.

Dalam keseluruhan, AVL tree memberikan waktu eksekusi yang sangat baik dalam kasus rata-rata dan terbaik, tetapi dapat menjadi lambat dalam kasus terburuk. Oleh karena itu, AVL tree sangat cocok untuk digunakan dalam aplikasi di mana operasi pencarian dan penyisipan sering dilakukan, seperti pada aplikasi database dan aplikasi pencarian teks.

E. Referensi

- https://yoi.home.blog/2020/01/11/membuat-program-sederhana-avl-tree-di-python/
- https://skillplus.web.id/binary-search-tree-menggunakan-python/

F. Github

Ranny-erzitha/Tugas-Analisis-Perancangan-Algoritma (github.com)