**SOAL:**

1. Buatlah fungsi untuk menghapus suatu node pada Tree!
2. Buatlah program lengkap untuk memanipulasi dan mensimulasikan tree dengan berbasis menu!

**JAWABAN**:

1. Funsgi untuk menghapus suatu node pada Tree

void hapus(btree \*\*node, int data) {

if (\*node == NULL) {

return;

}

if(data == (\*node)->data) { //data yang ingin dihapus ada di root

if((\*node)->kanan == NULL) {

\*node = (\*node)->kiri;

}

else if((\*node)->kiri == NULL) {

\*node = (\*node)->kanan;

}

else {

btree \*\*successor = getSuccessor(&(\*node)->kiri);

(\*node)->data = (\*successor)->data;

hapus(successor, (\*successor)->data);

}

}

else if(data < (\*node)->data) {

hapus(&(\*node)->kiri, data);

}

else {

hapus(&(\*node)->kanan, data);

}

}

1. Program memanipulasi data pada tree

import random

print('========================')

print('Nama : Febro Herdyanto')

print('NIM : 312010043')

print('Kelas : TI.20.B.1')

print('========================')

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):

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 Tree")

print("1.Tambah data pada tree")

print("2.Hapus data pada 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

Screenshot :

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Graphical user interface, text

Description automatically generated

Graphical user interface, application

Description automatically generated