# DFS LEAF SUM

```
graph = {
    'A': ['B', 'C'],
    'B': ['D','E'],
    'C': [],
    'D': [],
    'E': ['F'],
    'F': []
}
graph_val = {
    'A': 10,
    'B': 12,
    'C': 32,
    'D': 4,
    'E': -6,
    'F': -5,
}
visited = set()
leaf = set()

def dfs(visited, graph, node):
    if node not in visited:
        visited.add(node)
        if len(graph[node]) == 0:
            leaf.add(node)
        for neighbour in graph[node]:
            dfs(visited,graph,neighbour)
def leaf_sum(leaf):
    jmlh=8
    for leaf_node in leaf:
        leaf_val = graph_val[leaf_node]
        jmlh = jmlh + leaf_val

    return jmlh

dfs(visited,graph, 'A')
leaf_sum(leaf)
```

Program di atas merupakan implementasi dari algoritma Depth-First Search (DFS) pada sebuah graf yang direpresentasikan dalam bentuk adjacency list. Adjacency list merupakan matriks simetri dan menjadi representasi matriks yang menyatakan hubungan antar titik dalam suatu graf, bernilai 1 jika terdapat dua titik yang saling terhubung, dan bernilai 0 jika kedua titik tidak terhubung. Program ini juga menghitung jumlah dari nilai-nilai pada simpul daun (leaf) dalam graf.

Langkah-langkah yang dilakukan oleh program ini:

1. Program dimulai dengan mendefinisikan graf dalam bentuk dictionary graph, yang menyimpan informasi mengenai simpul-simpul dan hubungan antar simpul dalam graf tersebut.

```
graph = {
    'A': ['B', 'C'],
    'B': ['D','E'],
    'C': [],
    'D': [],
    'E': ['F'],
    'F': []
}
```

2. Selanjutnya, terdapat dictionary graph_val yang menyimpan nilai-nilai yang terkait dengan setiap simpul dalam graf.

```
graph_val = {
    'A': 10,
    'B': 12,
    'C': 32,
    'D': 4,
    'E': -6,
    'F': -5,
}
```

3. Kemudian, dibuat function visited yang akan digunakan untuk melacak simpul-simpul yang telah dikunjungi selama algoritma DFS berjalan.

```
visited = set()
```

4. Selanjutnya, dibuat juga function leaf yang akan digunakan untuk menyimpan simpul-simpul daun dalam graf.

```
leaf = set()
```

5. Program mendefinisikan fungsi dfs yang mengimplementasikan algoritma DFS. Fungsi ini menerima argumen visited untuk melacak simpul-simpul yang telah dikunjungi, graph untuk menyimpan informasi tentang hubungan antar simpul dalam graf, dan node yang merupakan simpul saat ini yang sedang dikunjungi.

```
def dfs(visited, graph, node):
```

6. Di dalam fungsi dfs, pertama-tama diperiksa apakah simpul saat ini belum pernah dikunjungi sebelumnya. Jika belum, simpul tersebut ditambahkan ke function visited.

```
if node not in visited:
        visited.add(node)
```

7. Selanjutnya, program memeriksa apakah simpul tersebut merupakan simpul daun dengan memeriksa apakah panjang daftar tetangga (neighbours) dari simpul tersebut adalah 0. Jika iya, simpul tersebut ditambahkan ke function leaf.

```
if len(graph[node]) == 0:
            leaf.add(node)
```

8. Setelah itu, program melakukan looping ke fungsi dfs untuk setiap tetangga simpul saat ini.

```
for neighbour in graph[node]:
    dfs(visited,graph,neighbour)
```

9. Program kemudian mendefinisikan fungsi leaf_sum yang menghitung jumlah nilai-nilai pada simpul-simpul daun dalam graf. Fungsi ini menerima argumen leaf yang merupakan himpunan simpul-simpul daun.

```
def leaf_sum(leaf):
```

10. Di dalam fungsi leaf_sum, variabel jmlh diinisialisasi dengan nilai 8. Kemudian, program melakukan iterasi melalui setiap simpul daun dalam himpunan leaf.

```
jmlh=8
```

```
    for leaf_node in leaf:
```

11. Pada setiap iterasi, program mengambil nilai dari dictionary graph_val berdasarkan simpul daun saat ini, dan menambahkannya ke variabel jmlh.

```
leaf_val = graph_val[leaf_node]
        jmlh = jmlh + leaf_val
```

12. Terakhir, program membalikkan nilai jmlh sebagai hasil dari fungsi leaf_sum.

```
return jmlh
```

13. Setelah itu, program memanggil fungsi dfs dengan menginisialisasi graf awal, dan kemudian memanggil fungsi leaf_sum untuk menghitung jumlah nilai-nilai simpul daun dalam graf tersebut.

```
dfs(visited,graph, 'A')
leaf_sum(leaf)
```

14. Hasil dari fungsi leaf_sum kemudian dikembalikan dan dicetak sebagai output.

Jadi, program di atas secara berulang atau rekursif menjelajahi graf menggunakan algoritma DFS, dan kemudian menghitung jumlah nilai-nilai pada simpul-simpul daun dalam graf tersebut.

# CONTOH ROOTING TREE

```
class Node:
    def __init__(self,data):
        self.data = data
        self.left = None
        self.right = None
def maxDepth(node):
    if node is None:
        return -1;
    else:
        lDepth = maxDepth(node.left)
        rDepth = maxDepth(node.right)

        if(lDepth > rDepth):
            return lDepth +1
        else:
            return rDepth +1
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)

print("Height of tree is %d" %(maxDepth(root)))
```

Program di atas merupakan implementasi untuk menghitung tinggi maksimum (depth) dari sebuah pohon biner menggunakan pendekatan rekursif. Langkah-langkah yang berjalan pada program diatas adalah:

1. Program dimulai dengan mendefinisikan kelas **Node**, yang merupakan simpul dalam pohon biner. Setiap simpul memiliki atribut **data** untuk menyimpan data pada simpul tersebut, serta atribut **left** dan **right** untuk menunjukkan anak kiri dan anak kanan dari simpul tersebut. Metode **__init__** digunakan untuk menginisialisasi atribut-atribut ini.

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

2. Selanjutnya, program mendefinisikan fungsi **maxDepth** yang akan mengembalikan tinggi maksimum (depth) dari pohon biner. Fungsi ini menerima argumen **node**, yang merupakan simpul saat ini yang sedang dievaluasi.

```
def maxDepth(node):
```

3. Dalam fungsi **maxDepth**, pertama-tama dilakukan pemeriksaan apakah simpul saat ini (**node**) bernilai **None**. Jika iya, maka simpul tersebut tidak ada (kosong) dan dikembalikan nilai **-1** sebagai tinggi maksimum.

```
if node is None:
        return -1;
```

4. Jika simpul tidak kosong, program melakukan pemanggilan rekursif ke fungsi **maxDepth** untuk child kiri dan child kanan dari simpul tersebut. Nilai kembalian dari pemanggilan rekursif ini akan menjadi tinggi maksimum dari anak kiri (**lDepth**) dan anak kanan (**rDepth**).

```
else:
        lDepth = maxDepth(node.left)
        rDepth = maxDepth(node.right)
```

5. Selanjutnya, program membandingkan nilai **lDepth** dan **rDepth**. Jika **lDepth** lebih besar dari **rDepth**, maka tinggi maksimum dari simpul saat ini adalah **lDepth + 1** (karena simpul saat ini ditambahkan). Jika tidak, tinggi maksimum adalah **rDepth + 1**.

```
if(lDepth > rDepth):
            return lDepth +1
else:
            return rDepth +1
```

6. Akhirnya, program membuat objek **root** sebagai akar dari pohon biner dan menginisialisasi simpul-simpul anak kiri dan anak kanannya.

```
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
```

7. Kemudian, program mencetak pesan "Height of tree is x" di mana **x** adalah hasil dari pemanggilan fungsi **maxDepth** dengan akar pohon (**root**) sebagai argumen.

```
print("Height of tree is %d" %(maxDepth(root)))
```

Dengan demikian, program ini akan menghitung dan mencetak tinggi maksimum dari pohon biner yang telah ditentukan.

## ROOTING TREE 1

```
class Node:
    def __init__(self,data):
        self.data = data
        self.left = None
        self.center = None
        self.right = None
def maxDepth(node):
    if node is None:
        return -1;
    else:
        lDepth = maxDepth(node.left)
        cDepth = maxDepth(node.center)
        rDepth = maxDepth(node.right)

        if(lDepth > cDepth):
            return lDepth +1
        elif (cDepth > rDepth):
            return cDepth +1
        else:
            return rDepth +1

root = Node(0)
root.left = Node(2)
root.center = Node(1)
root.right = Node(5)
root.left.center = Node(3)
root.right.left = Node(4)
root.right.right = Node(6)
```

```
print("Height of tree is %d" %(maxDepth(root)))
```

Program diatas adalah program yang sama dengan contoh dari rooting tree, hanya terdapat perbedaan pada jumlah node di level pertama. Yaitu pada contoh rooting tree level pertama hanya memiliki 2 node, sedangkan pada soal rooting tree 1 graph pada level pertama memiliki 3 node yaitu left, center, dan right

Untuk algoritma yang dijalankan juga memiliki perbedaan, yaitu :

1. Program dimulai dengan mendefinisikan kelas **Node**, yang merupakan simpul dalam pohon ternary. Setiap simpul memiliki atribut **data** untuk menyimpan data pada simpul tersebut, serta atribut **left**, **center**, dan **right** untuk menunjukkan tiga anak dari simpul tersebut. Metode **__init__** digunakan untuk menginisialisasi atribut-atribut ini.

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

2. Jika simpul tidak kosong, program melakukan pemanggilan rekursif ke fungsi maxDepth untuk anak kiri, anak tengah, dan anak kanan dari simpul tersebut. Nilai kembalian dari pemanggilan rekursif ini akan menjadi tinggi maksimum dari anak kiri (lDepth), anak tengah (cDepth), dan anak kanan (rDepth).

```
lDepth = maxDepth(node.left)
        cDepth = maxDepth(node.center)
        rDepth = maxDepth(node.right)
```

3. Selanjutnya, program membandingkan nilai lDepth, cDepth, dan rDepth. Jika lDepth lebih besar dari cDepth dan rDepth, maka tinggi maksimum dari simpul saat ini adalah lDepth + 1. Jika cDepth lebih besar dari rDepth, maka tinggi maksimum adalah cDepth + 1. Jika tidak, tinggi maksimum adalah rDepth + 1.

```
if(lDepth > cDepth):
            return lDepth +1
        elif (cDepth > rDepth):
            return cDepth +1
        else:
            return rDepth +1
```

# ROOTING TREE 2

```
class Node:
    def __init__(self,data):
        self.data = data
        self.left = None
        self.center = None
        self.right = None
def maxDepth(node):
    if node is None:
        return -1;
    else:
        lDepth = maxDepth(node.left)
        cDepth = maxDepth(node.center)
        rDepth = maxDepth(node.right)

        if(lDepth > cDepth or lDepth > rDepth):
            return lDepth +1
        elif (cDepth > lDepth or cDepth > rDepth):
            return cDepth +1
        else:
            return rDepth +1

root = Node(5)
root.left = Node(0)
root.center = Node(4)
root.right = Node(6)
root.left.left = Node(1)
root.left.right = Node(2)
root.left.right.right = Node(3)
root.left.right.left = Node(3)


print("Height of tree is %d" %(maxDepth(root)))
```

Perbedaan antara Rooting Tree 1 dengan Rooting Tree 2 adalah pada value node serta logic pada percabangannya, penjelasannya sebagai berikut :

```
if(lDepth > cDepth or lDepth > rDepth):
            return lDepth +1
        elif (cDepth > lDepth or cDepth > rDepth):
            return cDepth +1
        else:
            return rDepth +1
```

1. Program mengandung struktur kontrol **if-elif-else** yang digunakan untuk membandingkan tinggi maksimum (depth) dari tiga anak simpul saat ini: lDepth, cDepth, dan rDepth.
2. Pada kondisi pertama **if**, program memeriksa apakah lDepth lebih besar dari cDepth atau lDepth lebih besar dari rDepth. Jika salah satu kondisi tersebut benar, maka tinggi maksimum dari simpul saat ini adalah lDepth + 1.
3. Jika kondisi pertama tidak terpenuhi, maka program melanjutkan ke kondisi **elif**. Pada kondisi ini, program memeriksa apakah cDepth lebih besar dari lDepth atau cDepth lebih besar dari rDepth. Jika salah satu kondisi tersebut benar, maka tinggi maksimum dari simpul saat ini adalah cDepth + 1.
4. Jika kedua kondisi sebelumnya tidak terpenuhi, program akan menjalankan blok kode dalam **else**. Ini berarti bahwa tinggi maksimum dari simpul saat ini adalah rDepth + 1.
5. Setelah itu, program mengembalikan tinggi maksimum yang telah ditentukan.