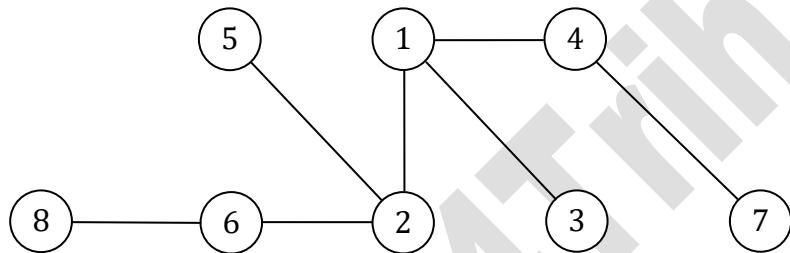


# Chapter 14

## Tree algorithms

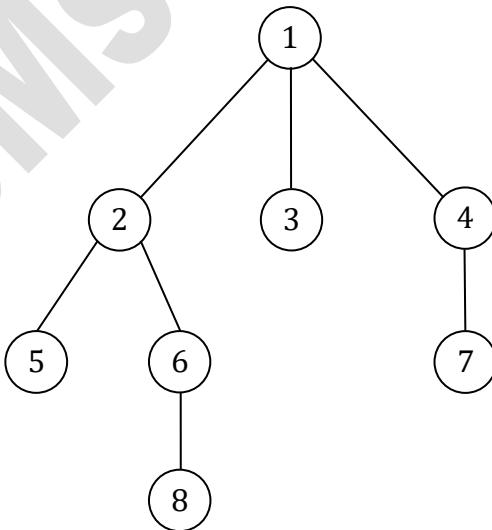
Sebuah **tree** adalah graf terhubung dan asiklik yang terdiri dari  $n$  node dan  $n - 1$  edge. Menghapus sembarang *edge* dari sebuah *tree* akan membaginya menjadi dua komponen, dan menambahkan sembarang *edge* ke dalam *tree* akan menciptakan sebuah *cycle*. Selain itu, selalu ada jalur unik antara sembarang dua *node* dalam sebuah *tree*.

Sebagai contoh, *tree* berikut terdiri dari 8 *node* dan 7 *edge*:



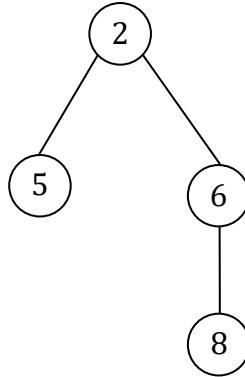
**Leaves** dari sebuah *tree* adalah *node* dengan *degree* 1, yaitu *node* yang hanya memiliki satu *neighbor*. Sebagai contoh, *leaves* dari *tree* di atas adalah *node* 3, 5, 7, dan 8.

Dalam sebuah **rooted tree**, salah satu *node* ditetapkan sebagai *root* dari *tree*, dan semua *node* lainnya ditempatkan di bawah *root*. Sebagai contoh, dalam *tree* berikut, *node* 1 adalah *root node*.



Dalam sebuah *rooted tree*, **children** dari sebuah *node* adalah *neighbor* yang berada di bawahnya, sedangkan *parent* dari sebuah *node* adalah *neighbor* yang berada di atasnya. Setiap *node* memiliki tepat satu *parent*, kecuali *root* yang tidak memiliki *parent*. Sebagai contoh, dalam *tree* di atas, *children* dari *node* 2 adalah *node* 5 dan 6, sedangkan *parent*-nya adalah *node* 1.

Struktur dari sebuah *rooted tree* bersifat rekursif: setiap *node* dalam *tree* bertindak sebagai *root* dari sebuah *subtree* yang mencakup *node* itu sendiri dan semua *node* yang berada dalam *subtree* dari *children*-nya. Sebagai contoh, dalam *tree* di atas, *subtree* dari *node* 2 terdiri dari *node* 2, 5, 6, dan 8.



## Tree Traversal

Algoritma *graph traversal* umum dapat digunakan untuk menelusuri *node* dalam sebuah *tree*. Namun, *traversal* dalam *tree* lebih mudah diimplementasikan dibandingkan *graph* umum, karena tidak ada *cycle* dalam *tree* dan tidak mungkin mencapai sebuah *node* dari banyak arah.

Cara umum untuk menelusuri sebuah *tree* adalah dengan memulai *depth-first search (DFS)* dari sembarang *node*. Fungsi rekursif berikut dapat digunakan:

```
void dfs(int s, int e) {  
    // proses node s  
    for (auto u : adj[s]) {  
        if (u != e) dfs(u, s);  
    }  
}
```

Fungsi ini memiliki dua parameter: *node* saat ini *s* dan *node* sebelumnya *e*. Tujuan dari parameter *e* adalah untuk memastikan bahwa pencarian hanya bergerak ke *node* yang belum dikunjungi.

Pemanggilan fungsi berikut akan memulai pencarian dari *node* *x*:

```
dfs(x, 0);
```

Pada pemanggilan pertama, *e* = 0, karena tidak ada *node* sebelumnya, sehingga diperbolehkan untuk bergerak ke arah mana pun dalam *tree*.

## Dynamic Programming

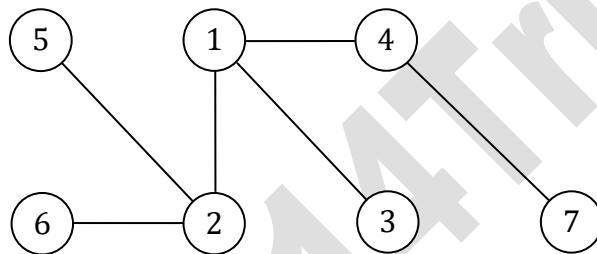
*Dynamic programming* dapat digunakan untuk menghitung beberapa informasi selama *tree traversal*. Dengan menggunakan *dynamic programming*, kita dapat, misalnya, menghitung dalam  $O(n)$  waktu jumlah *node* dalam *subtree* dari setiap *node* dalam sebuah *rooted tree* atau panjang *path* terpanjang dari sebuah *node* ke sebuah *leaf*.

Sebagai contoh, kita akan menghitung nilai `count[s]` untuk setiap `node s`: jumlah `node` dalam `subtree`-nya. *Sebuah subtree* mencakup `node` itu sendiri dan semua `node` dalam `subtree` dari `children`-nya, sehingga jumlah `node` dapat dihitung secara rekursif menggunakan kode berikut:

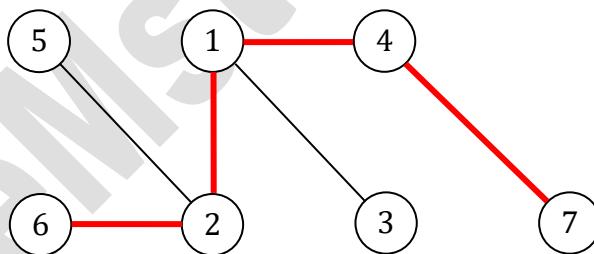
```
void dfs(int s, int e) {
    count[s] = 1;
    for (auto u : adj[s]) {
        if (u == e) continue;
        dfs(u, s);
        count[s] += count[u];
    }
}
```

## Diameter

**Diameter** dari sebuah *tree* adalah panjang maksimum dari sebuah *path* antara dua *node*. Sebagai contoh, pertimbangkan *tree* berikut:



Diameter dari *tree* ini adalah 4, yang sesuai dengan *path* berikut:



Perlu dicatat bahwa mungkin ada beberapa *path* dengan panjang maksimum. Dalam *path* di atas, kita bisa mengganti *node* 6 dengan *node* 5 untuk mendapatkan *path* lain dengan panjang 4.

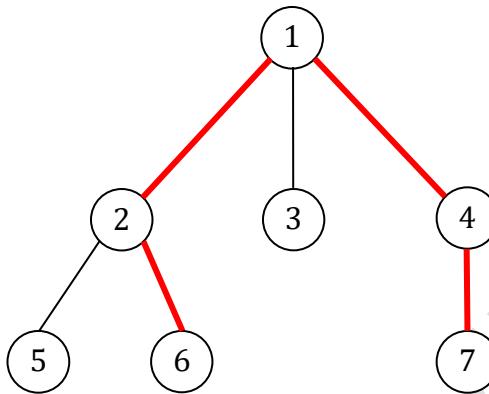
Selanjutnya, kita akan membahas dua algoritma  $O(n)$  untuk menghitung *diameter* dari sebuah *tree*. Algoritma pertama berbasis *dynamic programming*, sedangkan algoritma kedua menggunakan dua kali *depth-first search (DFS)*.

## Algorithm 1

Pendekatan umum untuk menyelesaikan banyak masalah *tree* adalah dengan terlebih dahulu meng-*root* *tree* secara arbitrer. Setelah itu, kita dapat mencoba menyelesaikan masalah secara terpisah untuk setiap *subtree*. Algoritma pertama untuk menghitung *diameter* berbasis pada ide ini.

Sebuah pengamatan penting adalah bahwa setiap *path* dalam *rooted tree* memiliki titik tertinggi: *node* tertinggi yang termasuk dalam *path* tersebut. Dengan demikian, kita dapat menghitung untuk setiap *node* panjang *path* terpanjang yang memiliki titik tertinggi pada *node* tersebut. Salah satu dari *path* tersebut akan sesuai dengan *diameter* dari *tree*.

Sebagai contoh, dalam *tree* berikut, *node* 1 adalah titik tertinggi pada *path* yang sesuai dengan *diameter*:



Kita menghitung dua nilai untuk setiap *node*  $x$ :

- $\text{toLeaf}(x)$ : panjang maksimum dari sebuah *path* dari  $x$  ke *leaf* mana pun.
- $\text{maxLength}(x)$ : panjang maksimum dari sebuah *path* yang memiliki titik tertinggi di  $x$ .

Sebagai contoh, dalam *tree* di atas:

- $\text{toLeaf}(1) = 2$ , karena terdapat *path*  $1 \rightarrow 2 \rightarrow 6$ .
- $\text{maxLength}(1) = 4$ , karena terdapat *path*  $6 \rightarrow 2 \rightarrow 1 \rightarrow 4 \rightarrow 7$ . Dalam kasus ini,  $\text{maxLength}(1)$  sama dengan *diameter* dari *tree*.

*Dynamic programming* dapat digunakan untuk menghitung nilai-nilai tersebut untuk semua *node* dalam  $O(n)$  waktu.

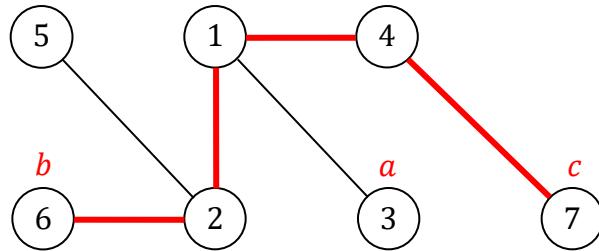
1. Untuk menghitung  $\text{toLeaf}(x)$ , kita melewati semua *children* dari  $x$ , memilih *child*  $c$  dengan nilai  $\text{toLeaf}(c)$  maksimum, lalu menambahkan 1 ke nilai tersebut.
2. Untuk menghitung  $\text{maxLength}(x)$ , kita memilih dua *children* yang berbeda,  $a$  dan  $b$ , sehingga jumlah  $\text{toLeaf}(a) + \text{toLeaf}(b)$  maksimum, lalu menambahkan 2 ke jumlah tersebut.

## Algorithm 2

Cara lain yang efisien untuk menghitung *diameter* dari sebuah *tree* adalah dengan menggunakan dua kali *depth-first search (DFS)*.

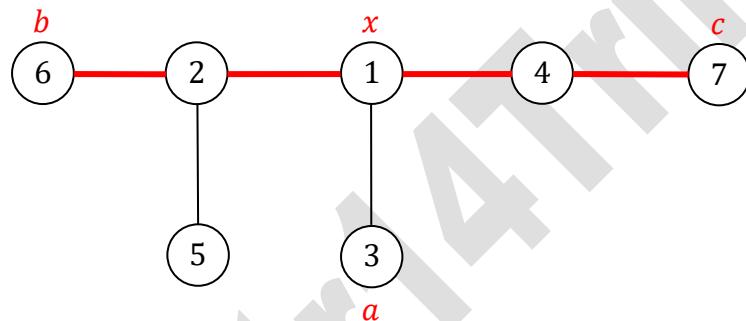
1. Pilih sembarang *node*  $a$  dalam *tree* dan temukan *node* terjauh  $b$  dari  $a$ .
2. Kemudian, temukan *node* terjauh  $c$  dari  $b$ .
3. *Diameter* dari *tree* adalah jarak antara  $b$  dan  $c$ .

Dalam *graph* berikut,  $a$ ,  $b$ , dan  $c$  bisa berupa:



Metode ini memang elegan, tetapi mengapa bisa bekerja?

Kita dapat memahami prinsipnya dengan menggambar ulang *tree* sedemikian rupa sehingga *path* yang sesuai dengan *diameter* berada dalam posisi horizontal, sementara semua *node* lainnya menggantung darinya.



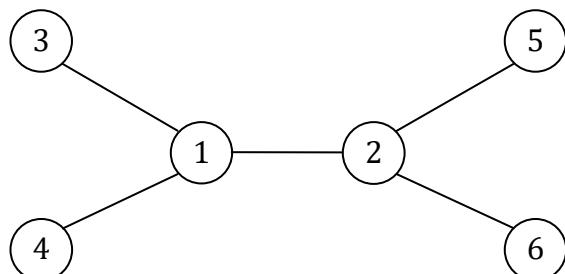
*Node x* menunjukkan titik di mana *path* dari *node a* bergabung dengan *path* yang sesuai dengan *diameter*.

*Node* terjauh dari *a* adalah *b*, *c*, atau *node* lain yang setidaknya sejauh *x*. Dengan demikian, *node* ini selalu menjadi pilihan yang valid sebagai ujung dari sebuah *path* yang sesuai dengan *diameter*.

## All Longest Paths

Masalah berikutnya adalah menghitung untuk setiap *node* dalam *tree* panjang maksimum dari sebuah *path* yang dimulai dari *node* tersebut. Masalah ini dapat dianggap sebagai generalisasi dari masalah *diameter tree*, karena nilai maksimum dari panjang-panjang tersebut akan sama dengan *diameter* dari *tree*. Masalah ini juga dapat diselesaikan dalam  $O(n)$  waktu.

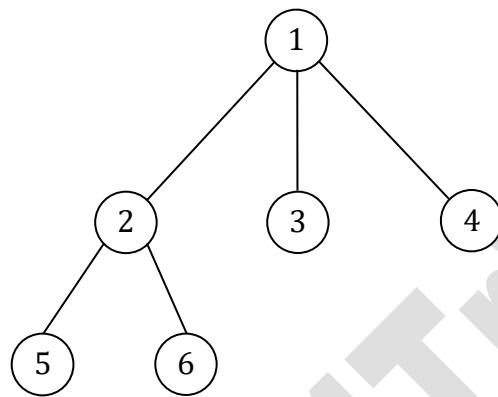
Sebagai contoh, pertimbangkan *tree* berikut:



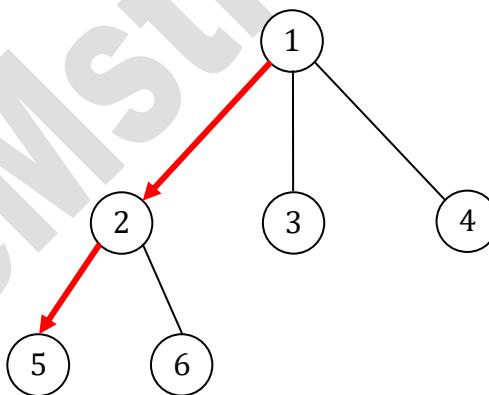
Misalkan  $\text{maxLength}(x)$  menyatakan panjang maksimum dari sebuah *path* yang dimulai dari *node*  $x$ . Sebagai contoh, dalam *tree* di atas,  $\text{maxLength}(4) = 3$ , karena terdapat *path*  $4 \rightarrow 1 \rightarrow 2 \rightarrow 6$ . Berikut adalah tabel lengkap dari nilai-nilai tersebut:

Node $x$	1	2	3	4	5	6
$\text{maxLength}(x)$	2	2	3	3	3	3

Dalam masalah ini juga, langkah awal yang baik untuk menyelesaiakannya adalah dengan meng-root *tree* secara arbitrer:

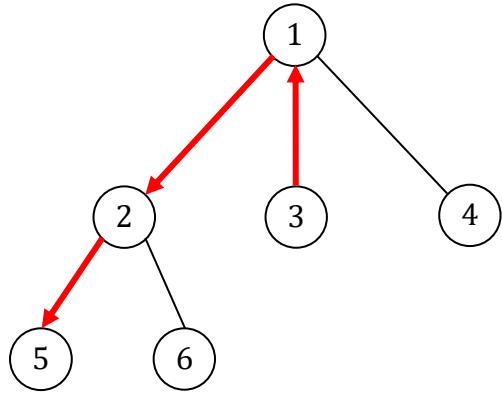


Bagian pertama dari masalah ini adalah menghitung untuk setiap *node*  $x$  panjang maksimum dari sebuah *path* yang melewati salah satu *child* dari  $x$ . Sebagai contoh, *path* terpanjang dari *node* 1 melewati *child*-nya, yaitu *node* 2:



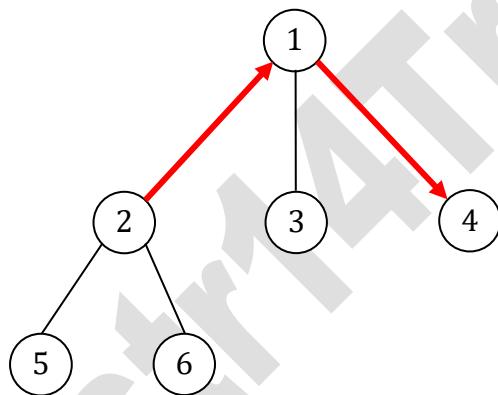
Bagian ini mudah diselesaikan dalam  $O(n)$  waktu, karena kita dapat menggunakan *dynamic programming* seperti yang telah kita lakukan sebelumnya.

Selanjutnya, bagian kedua dari masalah ini adalah menghitung untuk setiap *node*  $x$  panjang maksimum dari sebuah *path* yang melewati *parent*-nya  $p$ . Sebagai contoh, *path* terpanjang dari *node* 3 melewati *parent*-nya, yaitu *node* 1:



Sekilas, tampaknya kita harus memilih *path* terpanjang dari  $p$ . Namun, pendekatan ini tidak selalu berhasil, karena *path* terpanjang dari  $p$  mungkin saja melewati  $x$ .

Berikut adalah contoh dari situasi tersebut:



Namun, kita masih dapat menyelesaikan bagian kedua dalam  $O(n)$  waktu dengan menyimpan dua nilai maksimum untuk setiap  $node x$ :

- $\maxLength_1(x)$ : panjang maksimum dari sebuah *path* yang dimulai dari  $x$ .
  - $\maxLength_2(x)$ : panjang maksimum dari sebuah *path* yang dimulai dari  $x$ , tetapi dalam arah berbeda dari *path* pertama.

Sebagai contoh, dalam *graph* di atas:

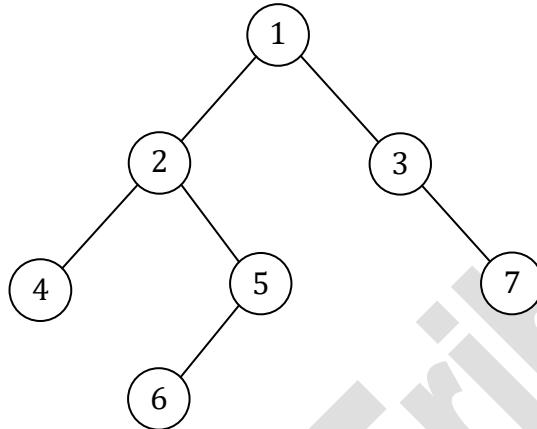
- $\maxLength_1(1) = 2$ , menggunakan path  $1 \rightarrow 2 \rightarrow 5$ .
  - $\maxLength_2(1) = 1$ , menggunakan path  $1 \rightarrow 3$ .

Terakhir, jika  $path$  yang sesuai dengan  $maxLength_1(p)$  melewati  $x$ , maka panjang maksimum dihitung sebagai  $maxLength_2(p) + 1$ . Jika tidak, panjang maksimum adalah  $maxLength_1(p) + 1$ .

## Binary Trees

Sebuah **binary tree** adalah *rooted tree* di mana setiap *node* memiliki *subtree* kiri dan kanan. Sebuah *subtree* dari suatu *node* dapat kosong. Oleh karena itu, setiap *node* dalam *binary tree* memiliki nol, satu, atau dua *children*.

Sebagai contoh, berikut ini adalah sebuah *binary tree*:



### Traversal dalam Binary Tree

Node-node dalam *binary tree* memiliki tiga cara penelusuran alami yang sesuai dengan berbagai metode untuk menelusuri *tree* secara rekursif:

- **Pre-order:** Proses *root* terlebih dahulu, lalu telusuri *subtree* kiri, kemudian *subtree* kanan.
- **In-order:** Telusuri *subtree* kiri terlebih dahulu, lalu proses *root*, kemudian telusuri *subtree* kanan.
- **Post-order:** Telusuri *subtree* kiri terlebih dahulu, lalu *subtree* kanan, kemudian proses *root*.

Untuk *binary tree* di atas, hasil penelusuran adalah:

- **Pre-order:** [1, 2, 4, 5, 6, 3, 7]
- **In-order:** [4, 2, 6, 5, 1, 3, 7]
- **Post-order:** [4, 6, 5, 2, 7, 3, 1]

### Rekonstruksi Binary Tree

Jika kita mengetahui **pre-order** dan **in-order** dari sebuah *tree*, kita dapat merekonstruksi struktur *tree* tersebut secara eksak. Sebagai contoh, *binary tree* di atas adalah satu-satunya kemungkinan yang sesuai dengan:

- **Pre-order:** [1, 2, 4, 5, 6, 3, 7]
- **In-order:** [4, 2, 6, 5, 1, 3, 7]

Dengan cara yang sama, kombinasi **post-order** dan **in-order** juga dapat menentukan struktur sebuah *tree*.

Namun, situasinya berbeda jika kita hanya mengetahui **pre-order** dan **post-order** dari sebuah *tree*. Dalam kasus ini, mungkin ada lebih dari satu *tree* yang cocok dengan urutan yang diberikan.

Sebagai contoh, dalam kedua *binary tree* berikut:



Urutan **pre-order** adalah [1,2] dan **post-order** adalah [2,1], tetapi struktur *tree* yang dihasilkan bisa berbeda.