



Bogor Agricultural University (IPB)

Searching & Serving the Best

<http://ipb.ac.id>

# Struktur Data (KOM20H)

---

Pertemuan 8 Struktur Data Graph



# Outline

---

- Representasi Graph dalam struktur data
- Graph Traversal
- Non-weighted shortest path
- Cycle Detection
- Topological sort



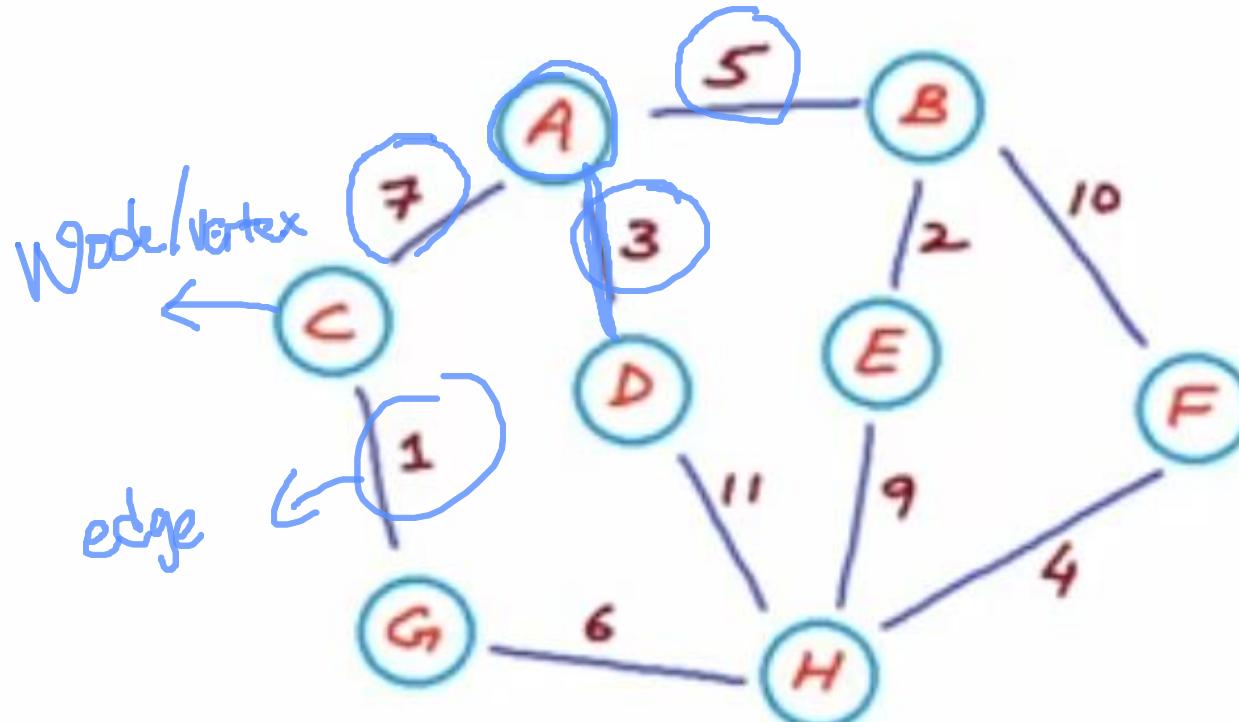


# Representasi Graph dalam struktur data

---

- Bagaimana merepresentasikan sebuah *graph dalam struktur data*?
- Bagaimana Representasi komponen verteks dan edges dalam Graph di sebuah Struktur Data
- 3 Metode umum representasi Graph dalam Struktur Data:
  - *Edge List*
  - *Adjacency Matrix*
  - *Adjacency List*

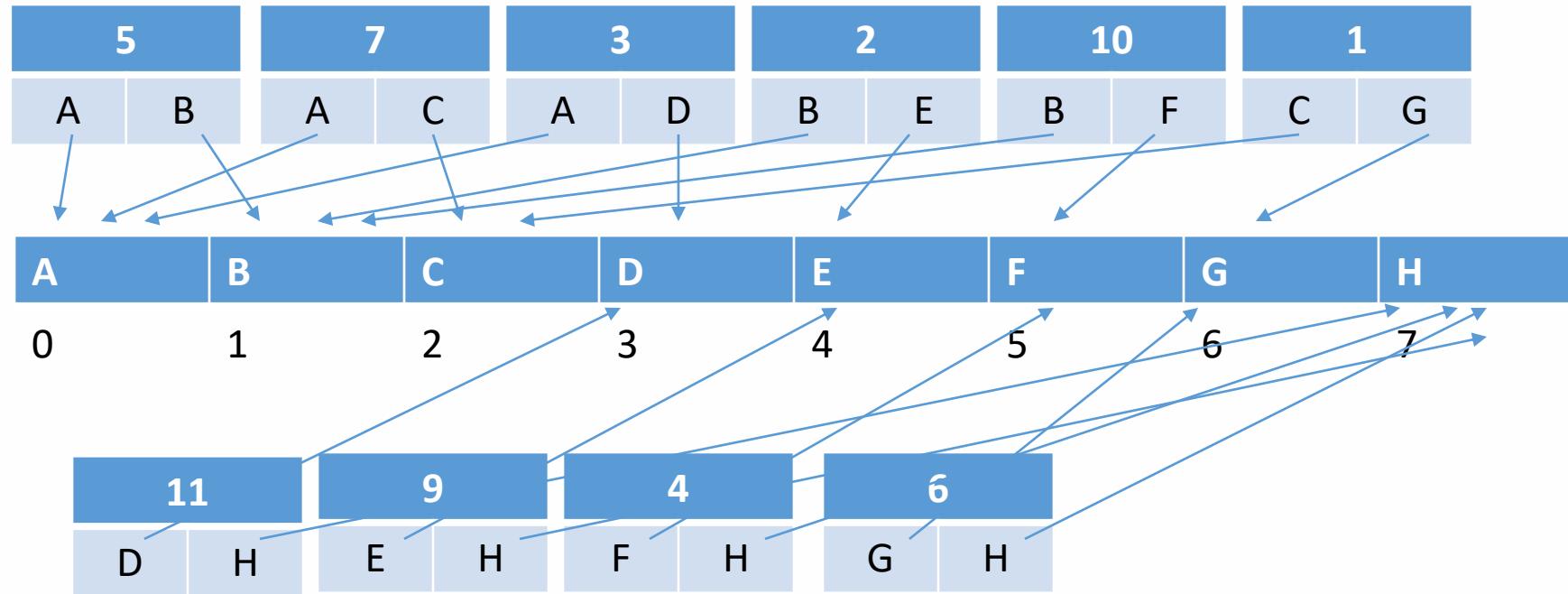
# Contoh Graph



a weighted graph



# Representasi Edge List



```
Struct Edge {  
    char *startVertex;  
    char *endVertex;  
    int weight;  
};
```



# Edge List

---

- Struktur ***edge list*** hanya menyimpan simpul (***vertices***) dan sisi (***edge***) dalam sebuah list yang tidak terurut.
- Pada tiap sisi disimpan informasi simpul yang terhubung oleh sisi tersebut
- Mudah diimplementasikan
- Tidak efisien dalam keperluan mencari *edge* bila diketahui *vertex*-nya



# Representasi Adjacency Matrix

0	A						
1	B	5	3	10	11	4	
2	C	0 1	0 3	1 5	3 7	5 7	
3	D	7	2	1	9	6	
4	E	0 2	1 4	2 6	4 7	6 7	
5	F						
6	G						
7	H						



# Representasi Adjacency Matrix

		5	3
0	A	0   1	0   3
1	B	7	2
2	C	0   2	1   4
3	D	10	11
4	E	1   5	3   7
5	F	1	9
6	G	2   6	4   7
7	H	4	6
		5   7	6   7

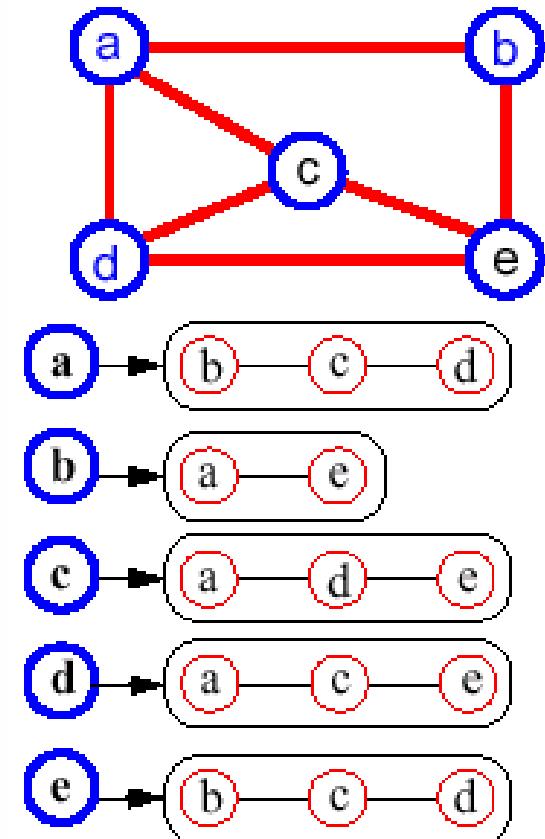
0	1	2	3	4	5	6	7								
0	0   1   1   1   0   0   0   0	1	0   0   0   0   1   1   0   0	2	1   0   0   0   0   0   1   0	3	1   0   0   0   0   0   0   1	4	0   1   0   0   0   0   0   1	5	0   1   0   0   0   0   0   1	6	0   0   1   0   0   0   0   1	7	0   0   0   1   1   1   1   0
1															
2															
3															
4															
5															
6															
7															

Adjacency Matrix A

$$A_{ij} = \begin{cases} 1, & \text{if } \exists \text{ edge from } i \text{ to } j \\ 0, & \text{otherwise} \end{cases}$$

# Adjacency List (Tradisional)

- Adjacency list dari sebuah *vertex v* adalah sekumpulan *vertex* yang terhubung dengan *v*
- Merepresentasikan graph, dengan menyimpan daftar *adjacency lists* dari seluruh *vertex*.
- Struktur adjacency list dapat digabungkan dengan struktur edge list.





# Representasi Adjacency List

	0	1	2	3	4	5	6	7
0	0	1	1	1	0	0	0	0
1	1	0	0	0	1	1	0	0
2	1	0	0	0	0	0	1	0
3	1	0	0	0	0	0	0	1
4	0	1	0	0	0	0	0	1
5	0	1	0	0	0	0	0	1
6	0	0	1	0	0	0	0	1
7	0	0	0	1	1	1	1	0

Adjacency Matrix A

0	A	1	2	3
1	B	0	4	5
2	C	0	6	
3	D	0	7	
4	E	1	7	
5	F	1	7	
6	G	2	7	
7	H	3	4	5
				6



# Kuis 1

---

Manakah pernyataan yang **benar**? (mungkin lebih dari satu)

- A. *Adjacency list* dapat menggunakan struktur representasi edge list
- B. Kebutuhan memori semua representasi matrix sama jika diterapkan pada graph yang sama
- C. Pada graph *undirected*, representasi adjacency matrix akan menghasilkan matrix yang simetris (artinya, isi pada baris ke-i kolom ke-j selalu sama dengan baris ke-j kolom ke-i)
- D. *Adjacency list* adalah representasi yang paling baik untuk semua kondisi dan tujuan



# Graph traversal

---

- Basic Idea
- **Depth** First Search
- **Breadth** First Search



# Graph traversal- Basic Idea

---

- Mengunjungi setiap vertex dari sebuah graph dalam cara yang sistematik.
- Ide dasar dari sebuah graph traversal adalah menandai (*marking*) setiap vertex yang telah kita kunjungi dan menjelajahi vertex yang belum kita kunjungi.



# Depth-First Search (DFS)

## Ide dasar:

- Proses penelusuran dilakukan ke dalam terlebih dahulu, sebelum melebar dari setiap *vertex*  $v$ , kita pilih satu *vertex*  $w$  yang *adjacent* dengan  $v$ , tetapi yang belum pernah dikunjungi sebelumnya, kemudian secara rekursif kita lakukan hal yang sama pada  $w$ .
- Kondisi berhenti diakukan jika semua *vertex* telah dikunjungi

## DFS:

- Ke dalam dulu, baru melebar
- Menggunakan pemanggilan secara rekursif
- Urutan pengunjungan **tidak** ditentukan oleh jarak dari titik asal



# Depth-First Search (DFS)

DFS( $G$ )

```
1  for each vertex  $u \in G.V$ 
2       $u.color = \text{WHITE}$ 
3       $u.\pi = \text{NIL}$ 
4       $time = 0$ 
5  for each vertex  $u \in G.V$ 
6      if  $u.color == \text{WHITE}$ 
7          DFS-VISIT( $G, u$ )
```

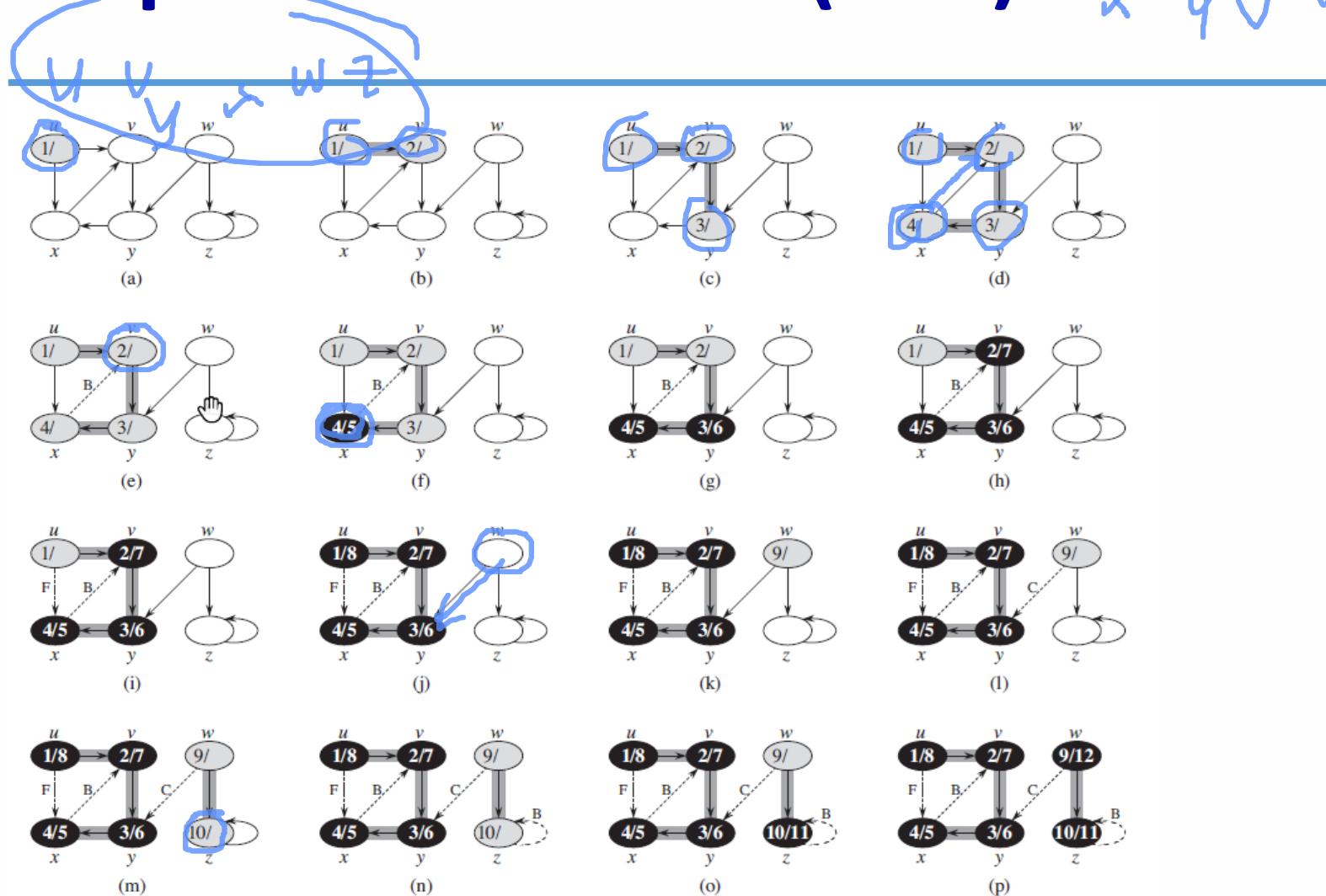
DFS-VISIT( $G, u$ )

```
1   $time = time + 1$            // white vertex  $u$  has just been discovered
2   $u.d = time$ 
3   $u.color = \text{GRAY}$ 
4  for each  $v \in G.Adj[u]$      // explore edge  $(u, v)$ 
5      if  $v.color == \text{WHITE}$ 
6           $v.\pi = u$ 
7          DFS-VISIT( $G, v$ )
8   $u.color = \text{BLACK}$            // blacken  $u$ ; it is finished
9   $time = time + 1$ 
10  $u.f = time$ 
```



# Depth-First Search (DFS)

x y g y



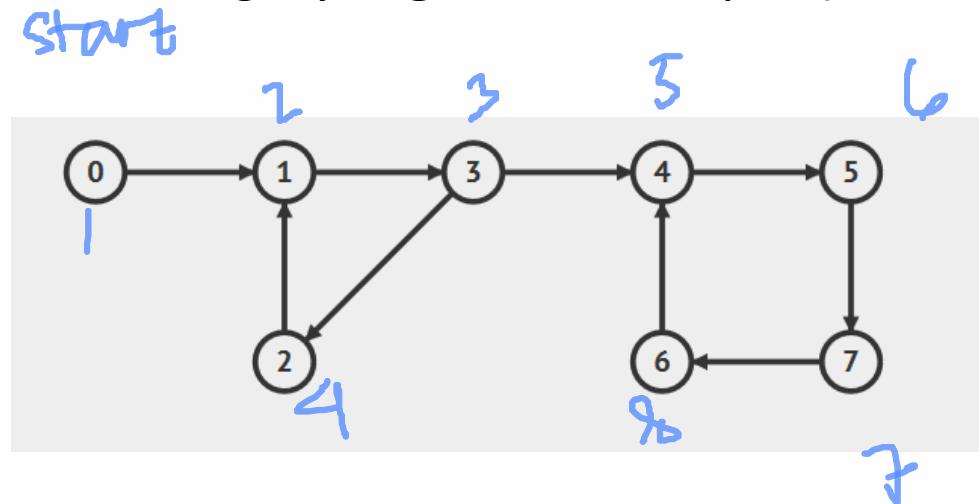


# Kuis 2

D, 1, 3, 2, 4, 5, 7, 6

Manakah dari pilihan-pilihan berikut yang *mungkin* menjadi urutan pengunjungan dengan DFS pada graph berikut (mungkin lebih dari satu jawaban, tergantung titik awal dan pilihan jika ada lebih dari dua *edge* yang bias ditempuh):

- A. 0, 1, 3, 2, 4, 5, 7, 6
- B. 4, 5, 7, 6, 0, 1, 3, 2
- C. 0, 1, 3, 4, 2, 5, 7, 6
- D. 3, 2, 1, 4, 5, 7, 6, 0
- E. 3, 2, 4, 1, 5, 7, 6, 0





# Breadth-First Search (BFS)

## Ide dasar:

- Melebar terlebih dahulu, sebelum ke dalam
- Dari setiap *vertex*  $v$ , kita harus kunjungi secara berturut-turut semua *vertex* yang *adjacent* dengan  $v$ , tetapi yang belum pernah dikunjungi sebelumnya
- Setelah itu, pilih satu *vertex*  $w$  yang *adjacent* dengan  $v$  dan lakukan hal yang sama
- Berhenti jika semua *vertex* telah dikunjungi

## BFS:

- Melebar dulu, baru mendalam
- Menggunakan Queue untuk menentukan urutan kunjungan
- Urutan pengunjungan ditentukan oleh **jarak langkah** dari sumber (*titik asal*)



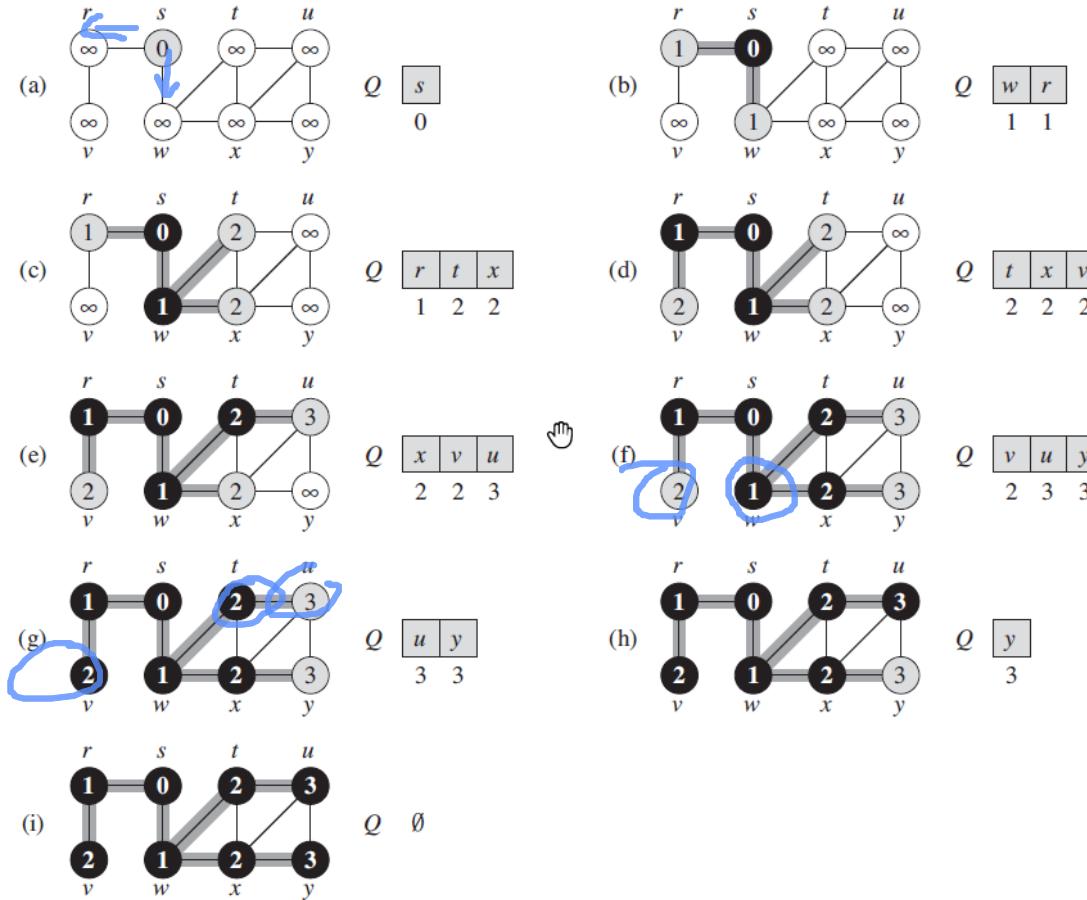
# Breadth-First Search (BFS)

---

BFS( $G, s$ )

```
1  for each vertex  $u \in G.V - \{s\}$ 
2       $u.color = \text{WHITE}$ 
3       $u.d = \infty$ 
4       $u.\pi = \text{NIL}$ 
5   $s.color = \text{GRAY}$ 
6   $s.d = 0$ 
7   $s.\pi = \text{NIL}$ 
8   $Q = \emptyset$ 
9  ENQUEUE( $Q, s$ )
10 while  $Q \neq \emptyset$ 
11      $u = \text{DEQUEUE}(Q)$ 
12     for each  $v \in G.Adj[u]$ 
13         if  $v.color == \text{WHITE}$ 
14              $v.color = \text{GRAY}$ 
15              $v.d = u.d + 1$ 
16              $v.\pi = u$ 
17             ENQUEUE( $Q, v$ )
18      $u.color = \text{BLACK}$ 
```

# Breadth-First Search (BFS)



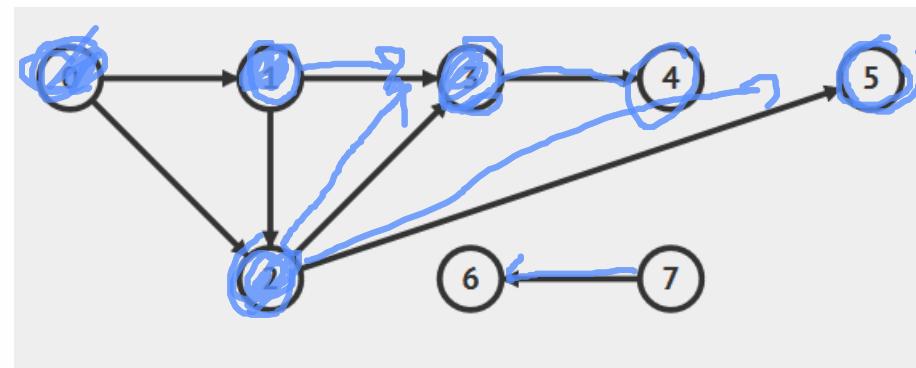


# Kuis 3

Manakah dari pilihan-pilihan berikut yang *mungkin* menjadi urutan pengunjungan dengan BFS pada graph berikut (mungkin lebih dari satu jawaban, tergantung titik awal dan pilihan jika ada lebih dari dua *edge* yang bias ditempuh):

- A. 0, 1, 2, 3, 4, 5, 6, 7
- B. 0, 2, 1, 3, 4, 5, 7, 6
- C. 0, 2, 1, 3, 5, 4, 7, 6
- D. 7, 6, 2, 5, 3, 4, 0, 1
- E. 7, 6, 2, 3, 4, 5, 0, 1

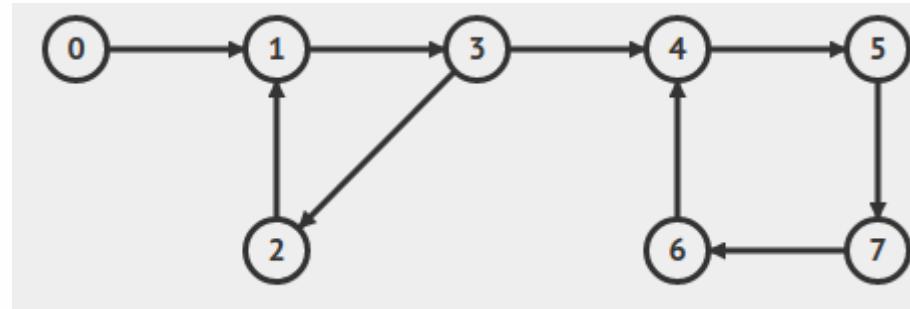
0 1 2 3 5 4 7 6



0 2 1 3 5 4 7 6

# Non-weighted shortest path

- Diberikan graph berikut, berapakah **jarak** (jumlah langkah **path** terpendek) dari simpul 0 ke simpul 6?
  - Jawab : 6 : 0 ↗ 1 ↗ 3 ↗ 4 ↗ 5 ↗ 7 ↗ 6
  - Ingat, **tidak ada** sisi dari 4 ke 6!





# Non-weighted shortest path

- BFS
  - Urutan pengunjungan ditentukan oleh **jarak langkah** dari sumber (*titik asal*)
- Modifikasi sederhana pada algoritme BFS:
  - Inisialisasi jarak semua simpul = **Inf** (tak-hingga)
  - Jarak **titik asal** terhadap dirinya sendiri = **0**
  - Jarak setiap **simpul** yang akan dikunjungi dari sebuah **simpul v** yang lain = jarak simpul **v** + 1
  - Secara rekursif dapat dibuktikan bahwa hasil jarak yang dihasilkan untuk setiap simpul adalah yang terkecil
  - Jika jarak suatu simpul di akhir = **Inf**, berarti simpul tersebut tidak dapat dicapai (**unreachable**) dari titik asal

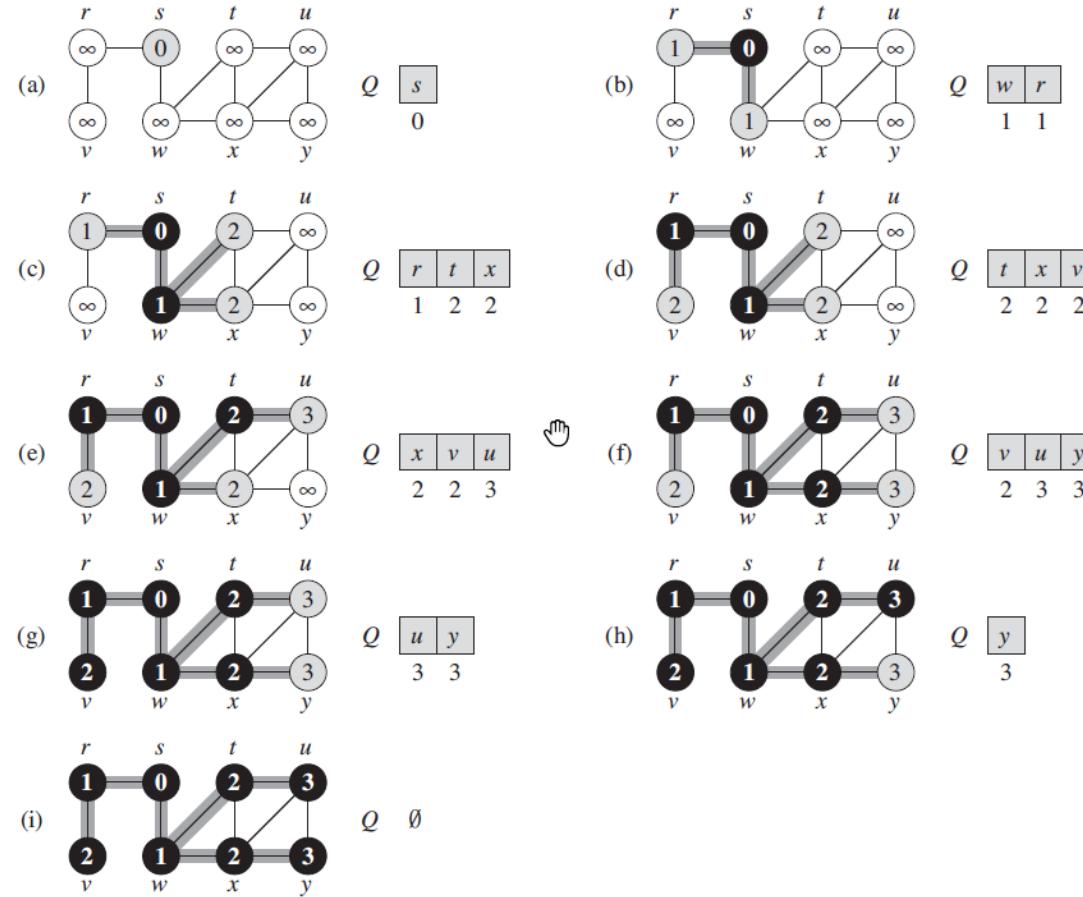


# Non-weighted shortest path

BFS( $G, s$ )

```
1  for each vertex  $u \in G.V - \{s\}$ 
2       $u.color = \text{WHITE}$ 
3       $u.d = \infty$ 
4       $u.\pi = \text{NIL}$ 
5   $s.color = \text{GRAY}$ 
6   $s.d = 0$ 
7   $s.\pi = \text{NIL}$ 
8   $Q = \emptyset$ 
9  ENQUEUE( $Q, s$ )
10 while  $Q \neq \emptyset$ 
11      $u = \text{DEQUEUE}(Q)$ 
12     for each  $v \in G.Adj[u]$ 
13         if  $v.color == \text{WHITE}$ 
14              $v.color = \text{GRAY}$ 
15              $v.d = u.d + 1$ 
16              $v.\pi = u$ 
17             ENQUEUE( $Q, v$ )
18      $u.color = \text{BLACK}$ 
```

# Non-weighted shortest path





# Kuis 4

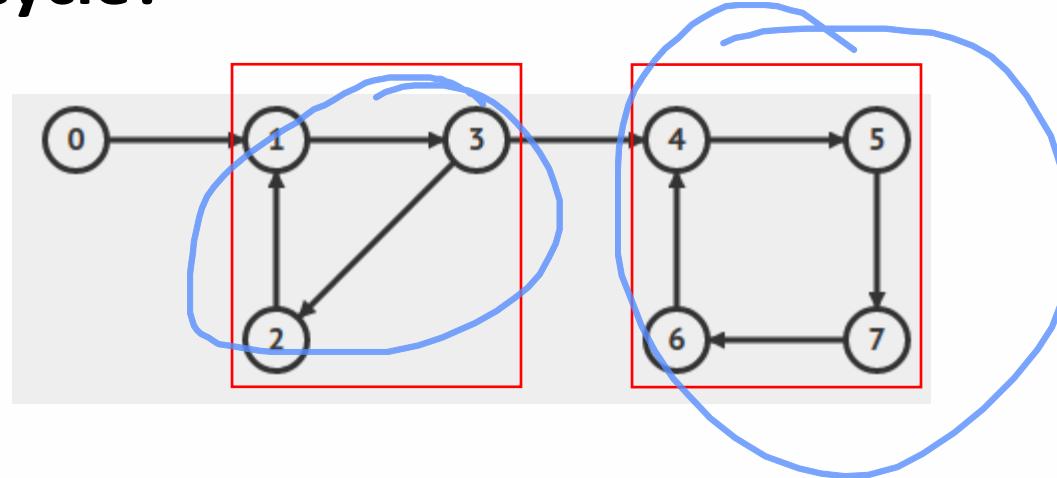
---

Manakah di antara pernyataan-pernyataan di bawah ini yang **benar** (mungkin lebih dari satu jawaban)?

- A. DFS juga dapat digunakan untuk menghitung **non-weighted shortest path** pada graph, jika dilakukan modifikasi yang sama seperti pada BFS
- B. Setelah inisialisasi, jarak suatu simpul hanya akan di-update maksimal **sekali** selama jalannya algoritme BFS
- C. BFS dapat juga diterapkan pada **undirected graph** untuk menghitung **non-weighted shortest path**
- D. Jika ada beberapa sisi/edge yang **masuk** ke sebuah simpul, maka simpul tersebut akan **di-update** jaraknya lebih dari sekali
- E. Simpul yang tidak dapat dicapai dari **titik asal** tidak akan pernah diketahui jaraknya dari titik asal

# Cycle Detection

- Apa itu **cycle**?



- Mungkin terjadi pada graph **berarah** dan **tidak berarah**
  - Cycle pada graph berarah juga memiliki **arah**
  - Cycle pada graph tidak berarah juga **tidak** memiliki **arah**



# Cycle Detection

- Mendeteksi adanya **cycle** hanya dibutuhkan informasi simpul-simpul mana saja yang **telah dikunjungi**
- Pada BFS dan DFS, ada informasi **COLOR** pada setiap simpul
  - WHITE = **belum pernah** dikunjungi sama sekali
- Kedua algoritme (BFS & DFS) dapat digunakan untuk melakukan deteksi **cycle**
  - Jika pada suatu simpul **v** yang **sedang dikunjungi**, kita menemukan simpul lain **w** yang **adjacent** terhadap **v** dan warnanya **bukan** WHITE, berarti terdapat **cycle** yang melibatkan **v** dan **w**
  - **DFS** cenderung **lebih cepat** dalam mendeteksi sebuah **cycle**

# Cycle Detection

## DFS

**DFS-VISIT**( $G, u$ )

```

1  time = time + 1           // white vertex  $u$  has just been discovered
2   $u.d$  = time
3   $u.color$  = GRAY
4  for each  $v \in G.Adj[u]$     // explore edge  $(u, v)$ 
5      if  $v.color == \text{WHITE}$ 
6           $v.\pi = u$ 
7          DFS-VISIT( $G, v$ )

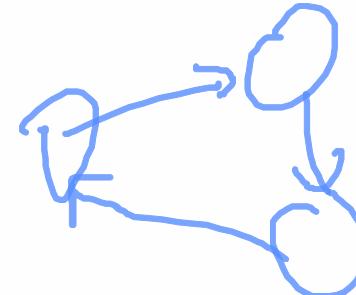
```

## BFS

```

for each  $v \in G.Adj[u]$ 
    if  $v.color == \text{WHITE}$       ←
         $v.color$  = GRAY
         $v.d$  =  $u.d + 1$ 
         $v.\pi = u$ 
        ENQUEUE( $Q, v$ )
        ...
    
```

**if  $v.color \neq \text{WHITE}$**   
**output ('Cycle detected!')**





# Kuis 5

---

Pilih **semua** pernyataan yang **salah** (mungkin lebih dari satu):

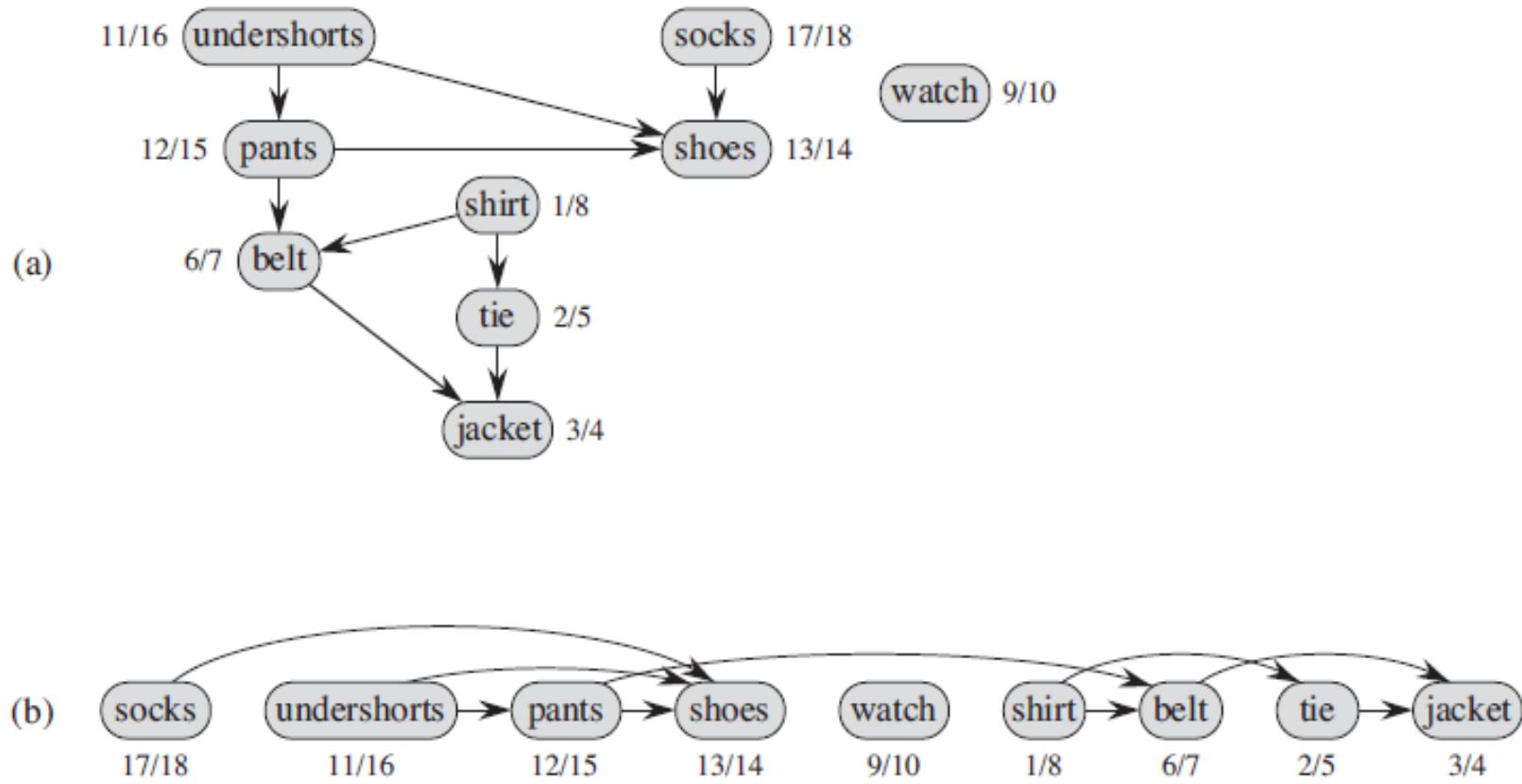
- A. Cycle detection adalah permasalahan yang dapat diterapkan pada graph **berarah** maupun **tidak berarah**
- B. Cycle detection dapat dilakukan mulai dari **titik asal** manapun pada graph
- C. Satu buah simpul dapat menjadi bagian dari **lebih** dari **satu** buah **cycle**
- D. Untuk graph pada hal. 10, jika kita lakukan deteksi cycle menggunakan BFS/DFS, dimulai dari simpul **0**, maka **cycle** di sebelah kanan (4 ? 5 ? 7 ? 6) **tidak mungkin** terdeteksi terlebih dahulu sebelum cycle di sebelah kiri (1 ? 2 ? 3) terdeteksi
- E. Cycle hanya terdeteksi jika ada **simpul** yang telah berwarna **BLACK** adjacent terhadap sebuah simpul yang sedang dikunjungi



# Topological sort

- Topological sort adalah pengurutan berdasarkan **ketergantungan antar-obyek**
- Banyak terjadi di dunia nyata:
  - urutan pengambilan mata kuliah sedemikian rupa sehingga setiap mata kuliah diambil **setelah semua prasyarat-nya** diambil
  - urutan *compiler* meng-*compile* sekumpulan *file* kode sumber berdasarkan **ketergantungan antar file (#include)**
- Hanya dapat dilakukan pada **Directed Acyclic Graph (DAG)**

# Topological sort





# Topological sort

- DFS
  - Ke dalam dulu, baru melebar ? semua ketergantungan yang berasal dari satu obyek akan terdeteksi
  - Menggunakan pemanggilan secara rekursif ? Ketergantungan juga bersifat **rekursif / transitif** (jika x bergantung pada y dan y bergantung pada z, maka x juga bergantung pada z)
- DFS lebih sesuai untuk tujuan topological sort
- BFS tidak memiliki sifat-sifat yang tepat untuk melakukan **topological sort**



# Topological sort

---

Modifikasi terhadap DFS untuk **topological sort**:

- Buat sebuah variabel pencatat waktu/langkah, diinisialisasi dengan nilai **0**
- Untuk setiap simpul **u**, kita akan catat waktu **discover** (kapan ditemukannya = warnanya berubah menjadi GRAY) simpul **u**, dan waktu **finish** (kapan **u** selesai dikunjungi = warnanya berubah menjadi BLACK)
- Setiap kali kita panggil DFS\_Visit pada sebuah simpul **u**, kita tambahkan variabel waktu, dan kita catat waktu discovery-nya
- Setelah semua vertex yang adjacent terhadap **u** selesai dikunjungi, kita tambahkan nilai variabel waktu dan catat **waktu finish** dari **u**.
- Urutkan semua **simpul** berdasarkan **waktu finish** secara **menurun**



# Topological sort

DFS( $G$ )

```
1 for each vertex  $u \in G.V$ 
2      $u.color = \text{WHITE}$ 
3      $u.\pi = \text{NIL}$ 
4     time = 0
5     for each vertex  $u \in G.V$ 
6         if  $u.color == \text{WHITE}$ 
7             DFS-VISIT( $G, u$ )
```

Inisialisasi variabel waktu secara global

DFS-VISIT( $G, u$ )

```
1     time = time + 1
2      $u.d = \text{time}$ 
3      $u.color = \text{GRAY}$ 
4     for each  $v \in G.Adj[u]$ 
5         if  $v.color == \text{WHITE}$ 
6              $v.\pi = u$ 
7             DFS-VISIT( $G, v$ )
8      $u.color = \text{BLACK}$ 
9     time = time + 1
10     $u.f = \text{time}$ 
```

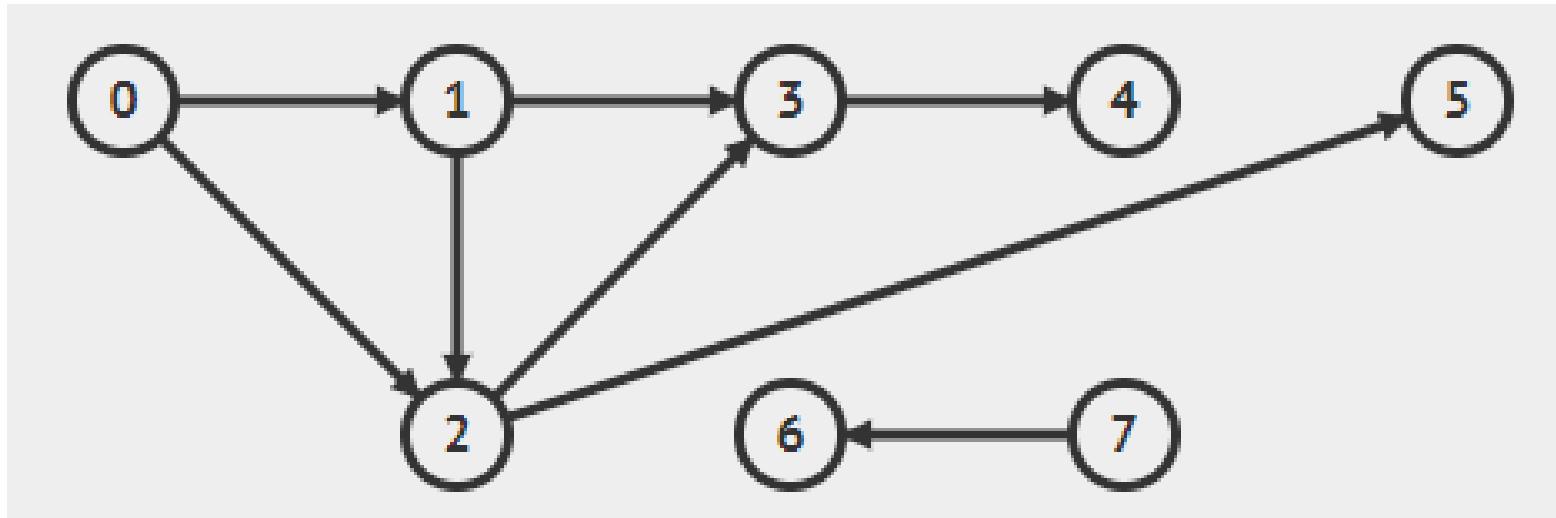
Pertambahan waktu dan pencatatan waktu *discovery* untuk simpul  $u$

// explore edge  $(u, v)$

Pertambahan waktu dan pencatatan waktu *finish* untuk simpul  $u$

# Topological sort

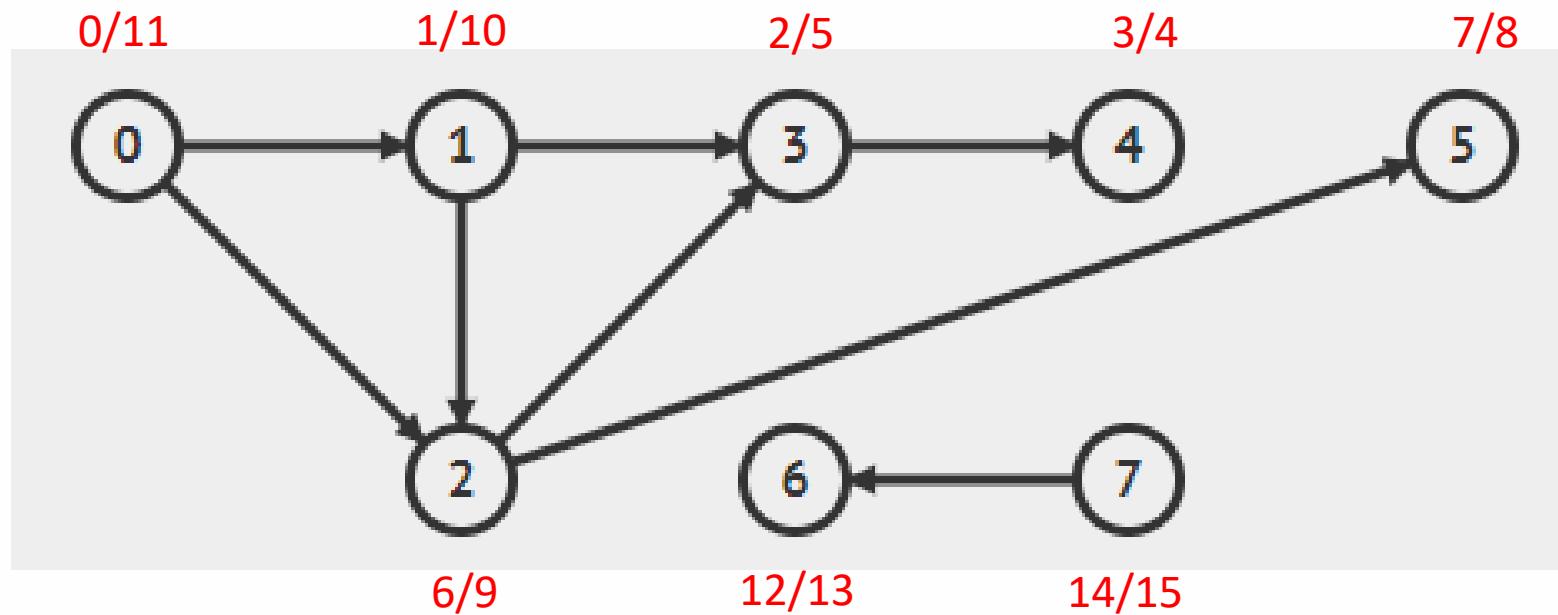
- Lakukan DFS mulai dari 0 : 0 ↗ 1 ↗ 3 ↗ 4 ↗ 2 ↗ 5 ↗ 6 ↗  
7



- Catat waktu **discovery** dan waktu **finish** dari setiap simpul

# Topological sort

- Lakukan DFS mulai dari 0 : 0 ? 1 ? 3 ? 4 ? 2 ? 5 ? 6 ? 7



- Topological sort : 7, 6, 0, 1, 2, 5, 3, 4



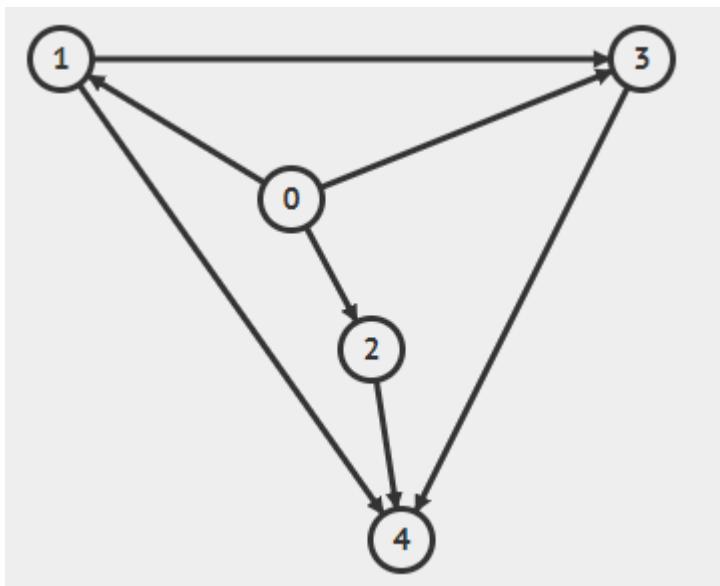
# Topological sort

- Topological sort dapat digabungkan dengan cycle detection
  - Input: graph berarah **G**
  - Output:
    - Ada cycle/tidak
    - Jika tidak ada cycle, topological sort dari graph **G**

```
if Detect_Cycle(G) == true
    output ('Cycle detected')
else
    output (Topological_sort(G))
```

# Kuis 6

- Pada DAG berikut, jika dijalankan DFS dengan urutan pengunjungan: 1 ? 3 ? 4? 0? 2, maka **waktu finish** untuk simpul 0 adalah:



- A. 4
- B. 5
- C. 6
- D. 7
- E. 8



Bogor Agricultural University (IPB)

Searching & Serving the Best

<http://ipb.ac.id>

# Struktur Data (KOM20H)

---

Pertemuan 9 Struktur Data Graph



# Outline

---

- Topological sort
- Implementasi Graf
  - Problem MST (Algoritme Prim & Kruskal)
  - Shortest Path (Algoritme Djikstra)
  - Problem Google Page Rank

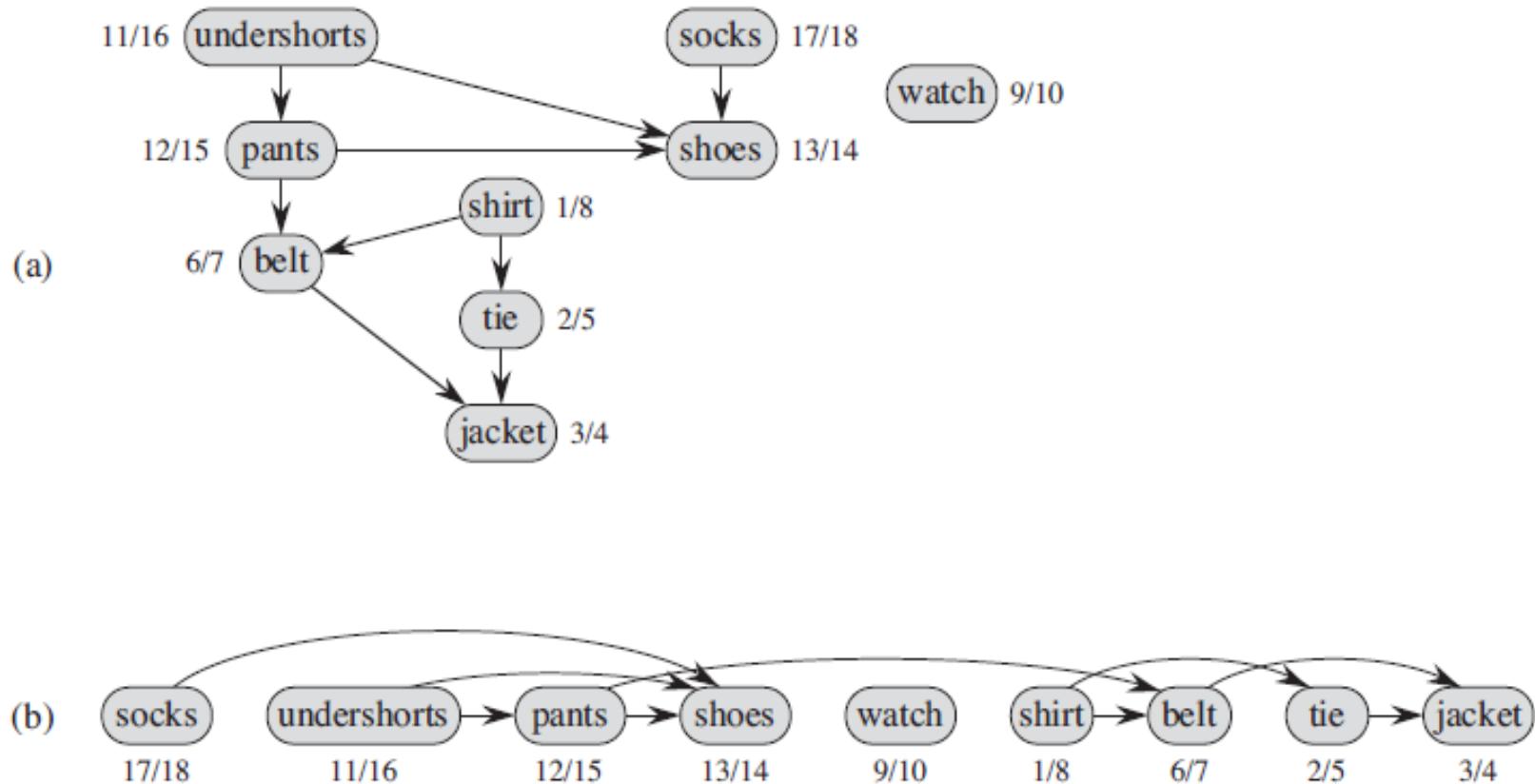


# Topological sort

---

- Topological sort adalah pengurutan berdasarkan **ketergantungan antar-obyek**
- Banyak terjadi di dunia nyata:
  - urutan pengambilan mata kuliah sedemikian rupa sehingga setiap mata kuliah diambil **setelah semua prasyarat-nya** diambil
  - urutan *compiler* meng-*compile* sekumpulan *file* kode sumber berdasarkan **ketergantungan antar file (#include)**
- Hanya dapat dilakukan pada **Directed Acyclic Graph (DAG)**

# Topological sort





# Topological sort

- DFS
  - Ke dalam dulu, baru melebar → semua ketergantungan yang berasal dari satu obyek akan terdeteksi
  - Menggunakan pemanggilan secara rekursif → Ketergantungan juga bersifat **rekursif / transitif** (jika x bergantung pada y dan y bergantung pada z, maka x juga bergantung pada z)
- DFS lebih sesuai untuk tujuan topological sort
- BFS tidak memiliki sifat-sifat yang tepat untuk melakukan **topological sort**



# Topological sort

---

Modifikasi terhadap DFS untuk **topological sort**:

- Buat sebuah variabel pencatat waktu/langkah, diinisialisasi dengan nilai **0**
- Untuk setiap simpul **u**, kita akan catat waktu **discover** (kapan ditemukannya = warnanya berubah menjadi GRAY) simpul **u**, dan waktu **finish** (kapan **u** selesai dikunjungi = warnanya berubah menjadi BLACK)
- Setiap kali kita panggil **DFS\_Visit** pada sebuah simpul **u**, kita tambahkan variabel waktu, dan kita catat waktu **discover**-nya
- Setelah semua vertex yang adjacent terhadap **u** selesai dikunjungi, kita tambahkan nilai variabel waktu dan catat **waktu finish** dari **u**.
- Urutkan semua **simpul** berdasarkan **waktu finish** secara **menurun**



# Topological sort

DFS( $G$ )

```
1 for each vertex  $u \in G.V$ 
2      $u.color = \text{WHITE}$ 
3      $u.\pi = \text{NIL}$ 
4     time = 0
5     for each vertex  $u \in G.V$ 
6         if  $u.color == \text{WHITE}$ 
7             DFS-VISIT( $G, u$ )
```

Inisialisasi variabel waktu secara global

DFS-VISIT( $G, u$ )

```
1     time = time + 1
2      $u.d = \text{time}$ 
3      $u.color = \text{GRAY}$ 
4     for each  $v \in G.Adj[u]$ 
5         if  $v.color == \text{WHITE}$ 
6              $v.\pi = u$ 
7             DFS-VISIT( $G, v$ )
8      $u.color = \text{BLACK}$ 
9     time = time + 1
10     $u.f = \text{time}$ 
```

Pertambahan waktu dan pencatatan waktu *discovery* untuk simpul  $u$

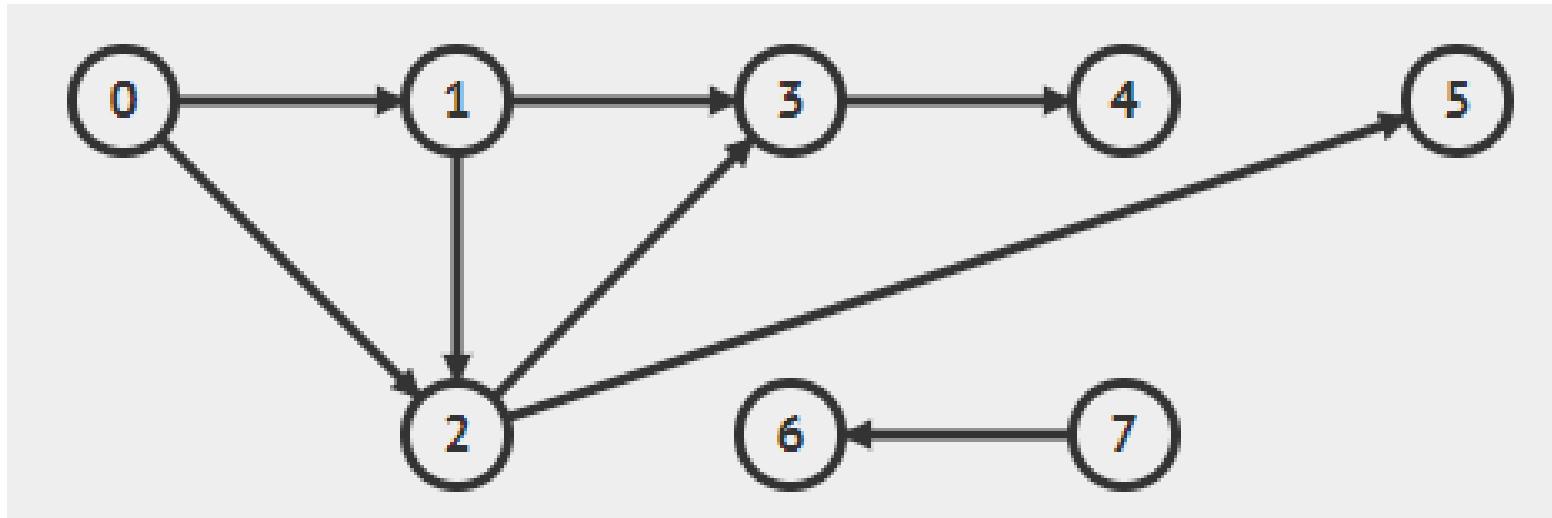
// explore edge  $(u, v)$

Pertambahan waktu dan pencatatan waktu *finish* untuk simpul  $u$



# Topological sort

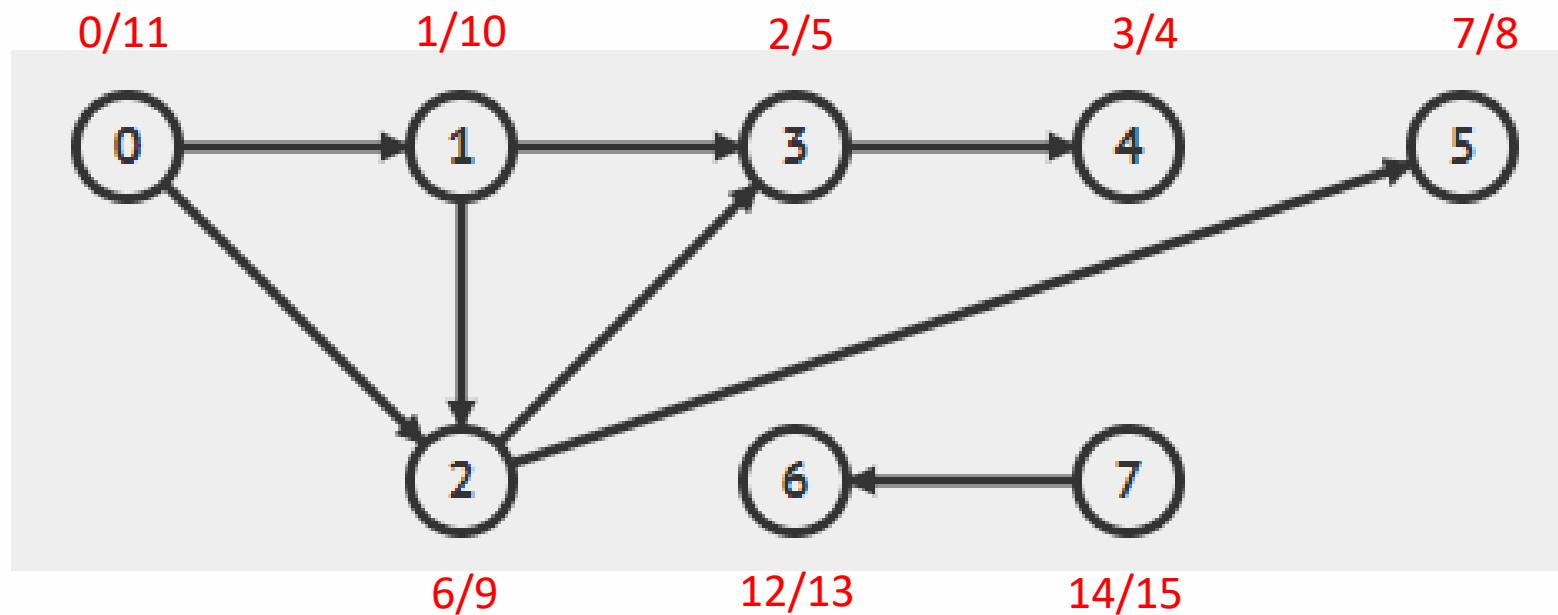
- Lakukan DFS mulai dari  $0 : 0 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 2 \rightarrow 5 \rightarrow 6 \rightarrow 7$



- Catat waktu **discovery** dan waktu **finish** dari setiap simpul

# Topological sort

- Lakukan DFS mulai dari  $0 : 0 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 2 \rightarrow 5 \rightarrow 6 \rightarrow 7$



- Topological sort : 7, 6, 0, 1, 2, 5, 3, 4



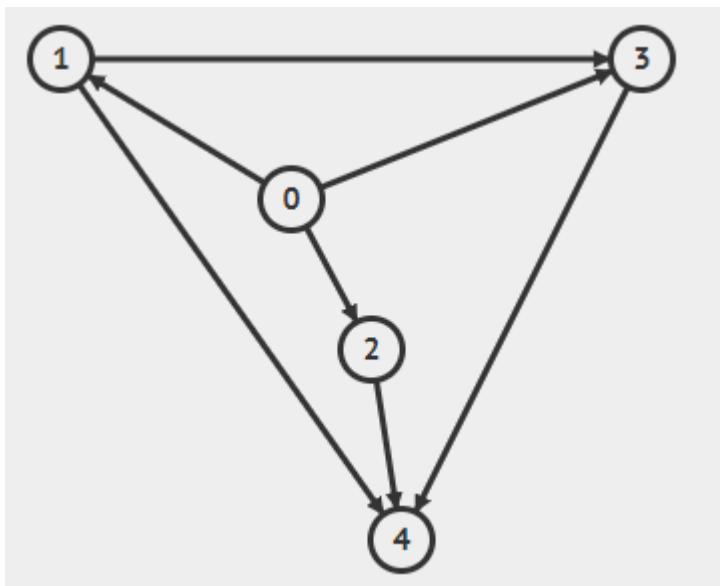
# Topological sort

- Topological sort dapat digabungkan dengan cycle detection
  - Input: graph berarah **G**
  - Output:
    - Ada cycle/tidak
    - Jika tidak ada cycle, topological sort dari graph **G**

```
if Detect_Cycle(G) == true
    output ('Cycle detected')
else
    output (Topological_sort(G))
```

# Kuis 6

- Pada DAG berikut, jika dijalankan DFS dengan urutan pengunjungan:  $1 \rightarrow 3 \rightarrow 4 \rightarrow 0 \rightarrow 2$ , maka **waktu finish** untuk simpul **0** adalah:



- A. 4
- B. 5
- C. 6
- D. 7
- E. 8



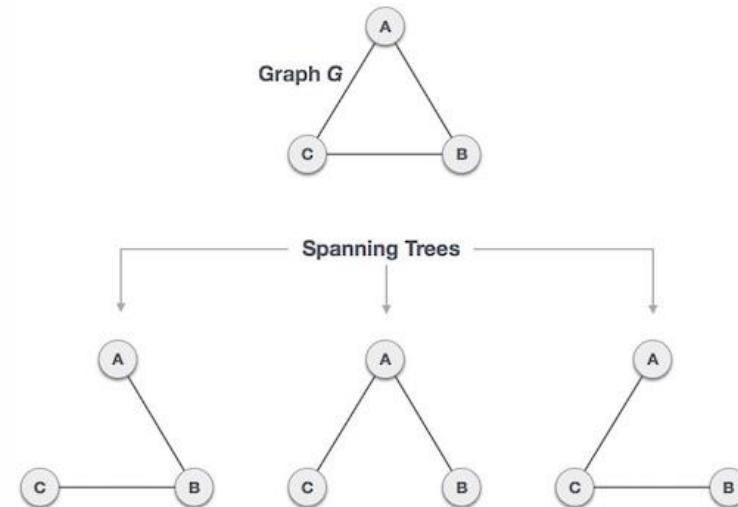
# Minimum Spanning Tree

---

- Spanning Tree dari sebuah graph **undirected & connected** adalah pohon/tree yang dibangun dari sebuah graph, dengan cara **menghapus** 0 atau lebih edge/sisi, sedemikian rupa sehingga:
  - Terbentuk tree (= tidak ada **cycle**)
  - Semua simpul/vertex tetap terhubung
- Satu graph mungkin memiliki banyak spanning tree
- Banyak aplikasi dalam dunia nyata:
  - Rute jalur kereta api
  - Jaringan listrik dan telepon
  - Sirkuit komponen digital
  - Distributed/parallel computing (**Spanning Tree Protocol**)

# Minimum Spanning Tree

- Contoh:
  - Diberikan graph G yang merupakan graph lengkap dengan 3 vertex sebagai berikut:

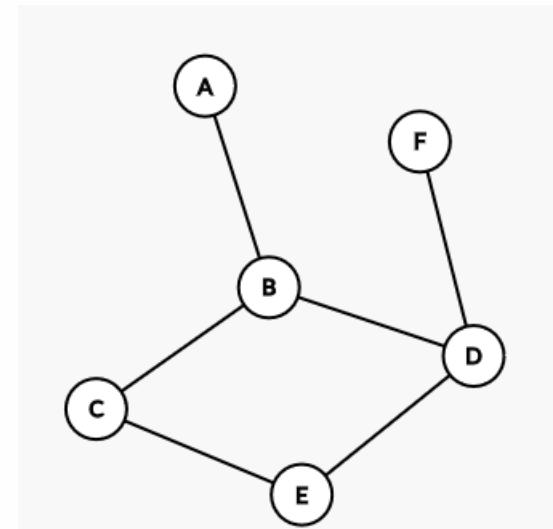


- Kita dapat membuat 3 buah spanning tree darinya



# Kuis 7

- Tentukan ada berapa buah spanning tree yang dapat dibuat untuk graph berikut:
  - A. 2
  - B. 3
  - C. 4
  - D. 5
  - E. 6





# Kuis 8

---

- Untuk sebuah graph terhubung  $G$  dengan  $n$  buah vertex, jumlah **edge** yang harus ada pada sebuah spanning tree dari  $G$  adalah:
  - $n$
  - $n - 1$
  - $n + 1$
  - $n - 2$
  - $n + 2$



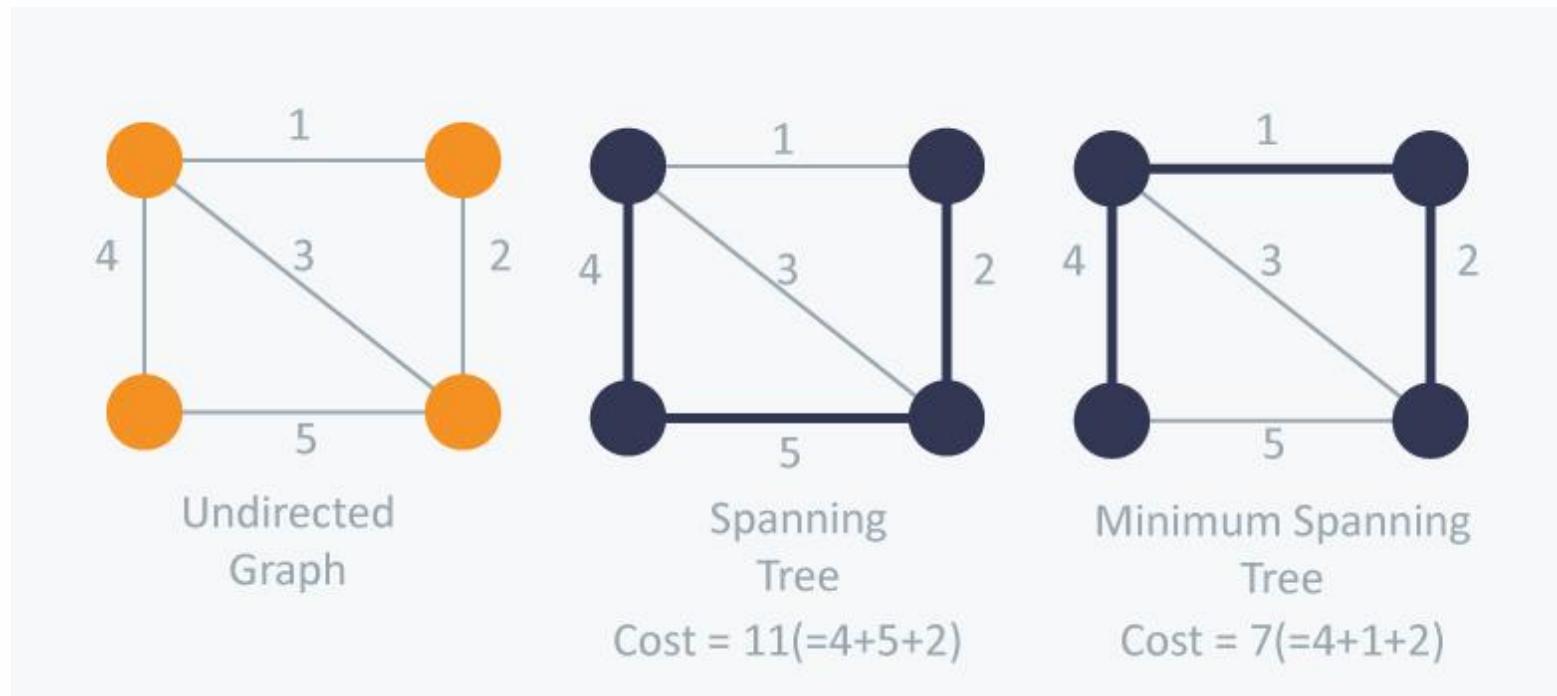
# Minimum Spanning Tree

---

- Diberikan graph yang:
  - Terhubung (connected)
  - Tidak berarah (undirected)
  - Berbobot (weighted)
- permasalahan Minimum Spanning Tree (MST) adalah permasalahan mencari spanning tree sedemikian rupa sehingga **total bobot** pada spanning tree adalah yang paling **kecil**
- Mungkin ada lebih dari satu MST, tetapi **total bobot-nya harus sama**

# Minimum Spanning Tree

- Contoh:





# Minimum Spanning Tree

---

- Algoritme klasik untuk menghitung MST:
  - Kruskal
  - Prim
- Didasarkan pada prinsip **greedy**:
  - Pada setiap langkah, ambil pilihan yang **terbaik** saat itu
  - Pada MST, berarti: pilih **edge** dengan bobot terkecil
  - Pastikan bahwa **cycle** tidak akan terbentuk



# Minimum Spanning Tree

---

- Kruskal
  - Diberikan input Graph  $G = \langle V, E \rangle$
  - Inisialisasi MST =  $\langle V, \{\} \rangle$
  - Lakukan, selama belum terbentuk spanning tree MST:
    - Pilih edge  $e$  terkecil di  $E$  yang belum terambil di MST, sedemikian hingga **tidak** terbentuk cycle pada MST
    - Tambahkan  $e$  pada MST
- Hasil antara (**intermediate result**) dari Kruskal adalah sebuah **forest** (**kumpulan tree terputus**)



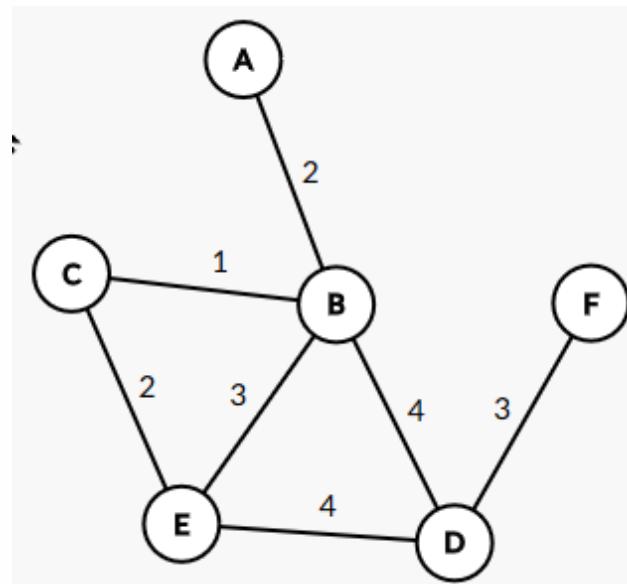
# Minimum Spanning Tree

---

- Prim
  - Diberikan input Graph  $G = \langle V, E \rangle$
  - Inisialisasi  $MST = \langle \{x\}, \{\} \rangle$ ,  $Rest = \langle V - \{x\}, E \rangle$ , dimana  $x$  adalah sebarang vertex pada  $V$
  - Lakukan, selama belum terbentuk spanning tree MST:
    - Pilih edge  $e = \langle v, w \rangle$  terkecil di  $E$  sedemikian rupa sehingga  $v \in Vertex(MST)$  dan  $w \in Vertex(Rest)$
    - Tambahkan  $e$  dan  $w$  pada  $MST$ , dan keluarkan  $e$  dan  $w$  dari  $Rest$
- Hasil antara (**intermediate result**) dari Prim adalah sebuah **tree**

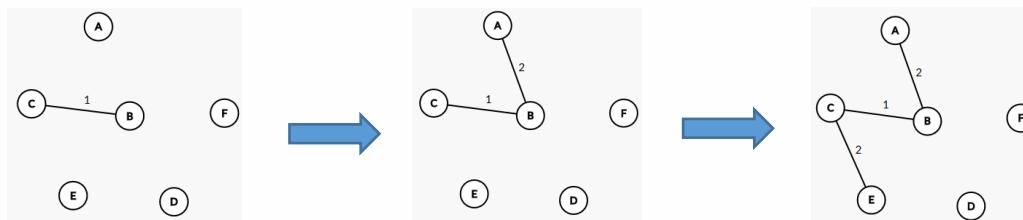
# Minimum Spanning Tree

- Contoh graph input  $G = \langle \{A, B, C, D, E, F\}, \text{Edges} \rangle$  dengan himpunan edge berbobot sebagai berikut:



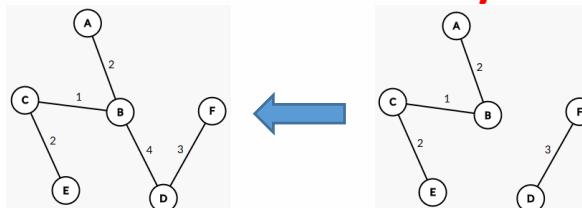
# Minimum Spanning Tree

## Kruskal



Urutkan edge berdasarkan bobotnya  
kemudian ambil 1-per-1, asalkan tidak terbentuk **cycle**

1. Pilih BC = 1
2. Pilih AB = 2
3. Pilih CE = 2
4. Tidak mungkin memilih BE karena akan menimbulkan **cycle**
5. Pilih DF → tercipta sebuah **forest** karena ada 2 tree terpisah
6. Pilih BD

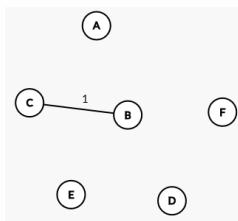


B • C • 1  
A • B • 2  
C • E • 2  
B • E • 3  
D • F • 3  
B • D • 4  
D • E • 4

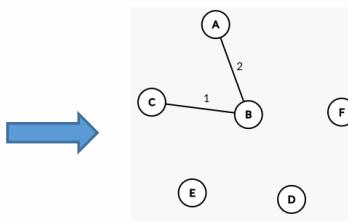
Urutan edge berdasarkan bobot

# Minimum Spanning Tree

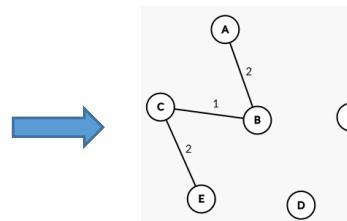
## Prim



MST = {B, C}, Rest = {A, D, E, F}



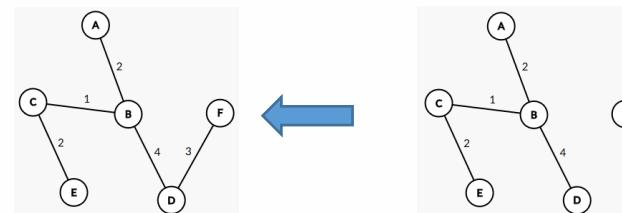
MST = {A, B, C}, Rest = {D, E, F}



MST = {A, B, C, E}, Rest = {D, F}

Urutkan edge berdasarkan bobotnya  
kemudian ambil 1-per-1, dimana satu vertex **harus** ada di MST  
dan satunya lagi ada di Rest

1. Pilih BC = 1
2. Pilih AB = 2
3. Pilih CE = 2
4. Pilih BD = 4
5. Pilih DF = 3



B . C . 1  
A . B . 2  
C . E . 2  
B . E . 3  
D . F . 3  
B . D . 4  
D . E . 4

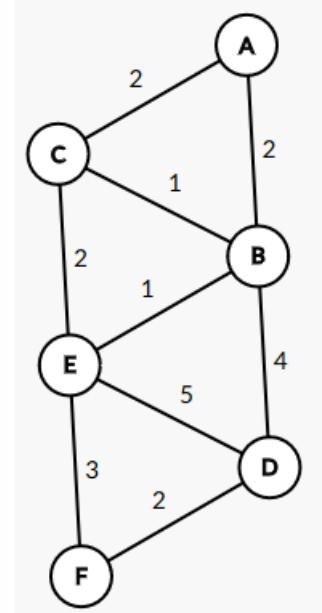
Urutan edge berdasarkan bobot

Pada setiap langkah, hanya ada 1 tree yang terbentuk



# Kuis 9

- Berapakah total bobot pada MST untuk graph di bawah ini?
  - 7
  - 8
  - 9
  - 10
  - 11





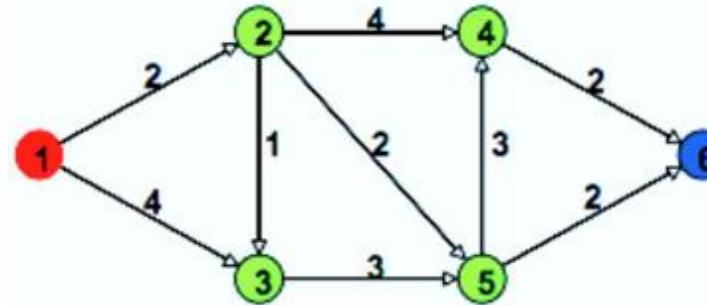
# Shortest Path

---

- Diberikan sebuah graph (**berarah/tidak berarah**) dan **berbobot** G, serta sebuah **vertex** sumber/titik asal s pada G:
  - Tentukan path dengan **total bobot** terkecil dari s ke **seluruh** vertex lainnya pada G
- Mirip dengan **unweighted shortest path** yang kita pelajari pada pertemuan sebelumnya
  - Perbedaan ada pada **bobot**, algoritme BFS tidak mempertimbangkan bobot dan hanya memperhitungkan jumlah **langkah** pada jalur terpendek
  - Jalur dengan total **bobot** terkecil bisa jadi **tidak** memiliki jumlah langkah terkecil (dan sebaliknya)

# Shortest Path

- Contoh:
  - Berapakah jarak terpendek dari vertex **1** ke vertex **6**?



- Jalur dengan total jarak terpendek =  $1 \rightarrow 2 \rightarrow 5 \rightarrow 6$  dengan total jarak =  $2 + 2 + 2 = 6$



# Kuis 10

---

- Mengapa algoritme BFS yang digunakan pada pertemuan sebelumnya **tidak** dapat digunakan untuk menyelesaikan permasalahan **shortest path** secara umum? Pilih **semua** jawaban yang benar!
  - A. Algoritme BFS tidak bersifat **rekursif** sehingga tidak sesuai untuk permasalahan **shortest path**
  - B. Jalur dengan jumlah langkah terkecil belum tentu memiliki **total jarak** terkecil
  - C. Jalur dengan total jarak terkecil belum tentu memiliki jumlah langkah terkecil
  - D. BFS hanya cocok diterapkan pada graph **tidak berarah**, sedangkan shortest path mungkin dilakukan pada graph berarah juga
  - E. Dua buah jalur mungkin memiliki jumlah langkah yang sama, tetapi total bobot-nya mungkin berbeda



# Algoritme Dijkstra

- Algoritme **Dijkstra** adalah satu algoritme untuk menyelesaikan permasalahan **shortest path** untuk graph dimana semua bobotnya **non-negatif**.
- Ide dasar:
  - **Prinsip Optimalitas Bellman:** Jika R adalah sebuah vertex yang ada pada jalur terpendek antara vertex P dan Q, maka jalur dari P ke R juga adalah jalur terpendek.



# Algoritme Dijkstra

- Notasi

- $v.d$  = estimasi terkini jarak terpendek dari vertex  $v$  ke titik asal  $s$
- $v.\pi$  = vertex **predecessor** dari  $v$ , yaitu vertex terakhir sebelum  $v$  pada jalur terpendek dari  $s$  ke  $v$ .

- Notasi

- $v.d$  = estimasi terkini jarak terpendek dari vertex  $v$  ke titik asal  $s$
- $v.\pi$  = vertex **predecessor** dari  $v$ , yaitu vertex terakhir sebelum  $v$  pada jalur terpendek dari  $s$  ke  $v$ .

- Contoh: pada jalur **1→2→5→6**, predecessor dari **6** adalah **5**, sedangkan predecessor dari **5** adalah **2**

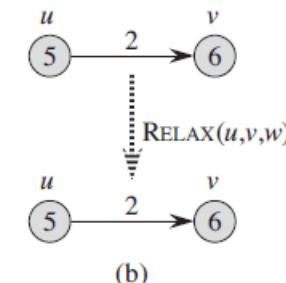
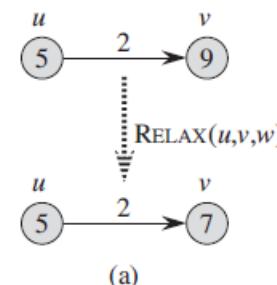
- Contoh: pada jalur **1→2→5→6**, predecessor dari **6** adalah **5**, sedangkan predecessor dari **5** adalah **2**

# Algoritme Dijkstra

- Sub-prosedur utama dari algoritme Dijkstra adalah proses **relaksasi** (Relax):

$\text{RELAX}(u, v, w)$

- 1 **if**  $v.d > u.d + w(u, v)$
- 2      $v.d = u.d + w(u, v)$
- 3      $v.\pi = u$



- Jika pada suatu vertex  $u$  yang sedang kita proses, kita dapatkan sebuah vertex  $v$  yang terhubung terhadap vertex  $u$  dengan bobot  $w$ , dan jika estimasi jarak untuk  $v$  saat ini **lebih besar** dari estimasi jarak untuk  $u$  **ditambah** dengan  $w$ , maka kita harus **update** estimasi jarak untuk  $v$  menjadi estimasi jarak  $u$  ditambah  $w$ , dan set vertex predecessor dari  $v$  menjadi  $u$ .



# Algoritme Dijkstra

- Inisialisasi di awal algoritme Dijkstra

INITIALIZE-SINGLE-SOURCE( $G, s$ )

1 for each vertex  $v \in G.V$

2      $v.d = \infty$

3      $v.\pi = \text{NIL}$

4      $s.d = 0$

Semua vertex jaraknya di-inisialisasi sebagai Infinity (tak-hingga)

Dan semua predecessor-nya kosong

Jarak dari  $s$  ke dirinya sendiri = 0

- Prosedur utama algoritme Dijkstra

DIJKSTRA( $G, w, s$ )

1 INITIALIZE-SINGLE-SOURCE( $G, s$ )

2  $S = \emptyset$

3  $Q = G.V$

Buat priority queue dari semua vertex berdasarkan estimasi jaraknya

4 while  $Q \neq \emptyset$

5      $u = \text{EXTRACT-MIN}(Q)$

Ambil vertex dengan estimasi jarak terkecil saat ini

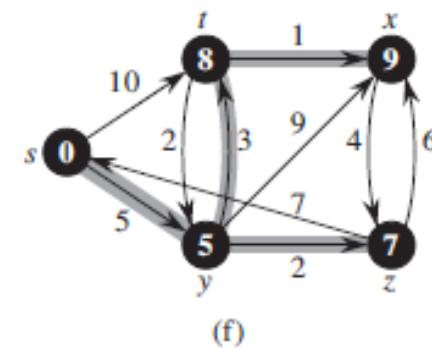
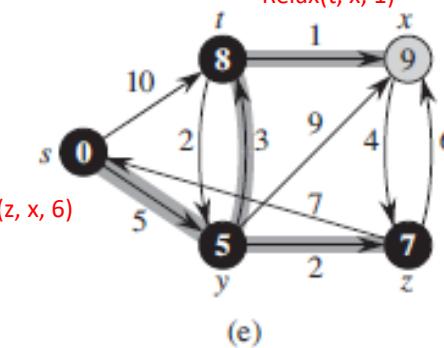
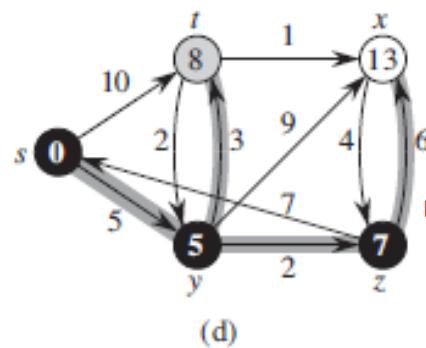
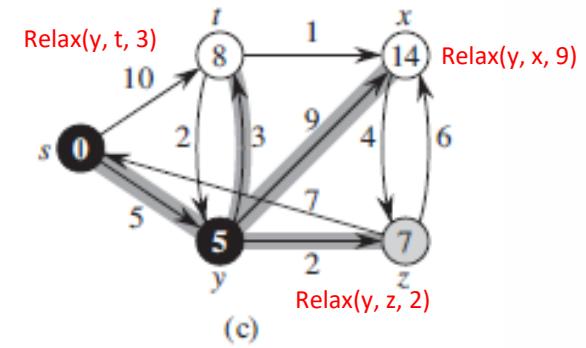
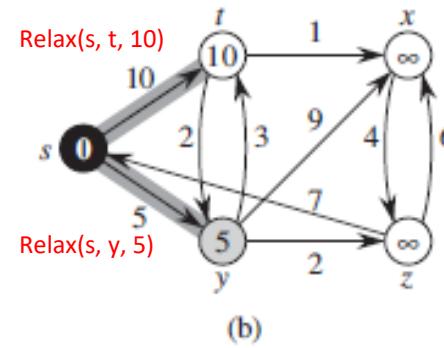
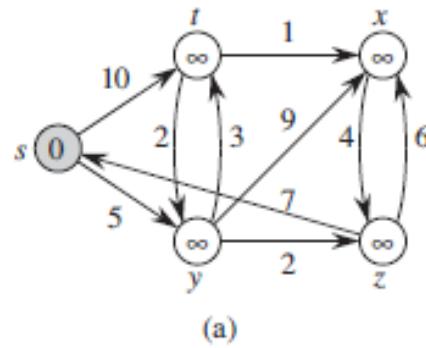
6      $S = S \cup \{u\}$

7     for each vertex  $v \in G.Adj[u]$

8          $\text{RELAX}(u, v, w)$

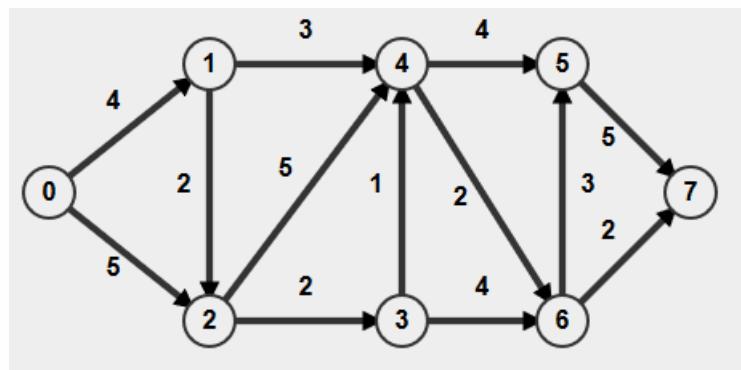
Lakukan relaksasi antara  $u$  dan  $v$

# Algoritme Dijkstra



# Kuis 11

- Tentukan jarak terpendek dari vertex **0** ke vertex **7**, dan apa vertex **predecessor**-nya!



- A. Total jarak = 16, predecessor = **5**
- B. Total jarak = 11, predecessor = **6**
- C. Total jarak = 11, predecessor = **4**
- D. Total jarak = 13, predecessor = **6**
- E. Total jarak = 13, predecessor = **2**

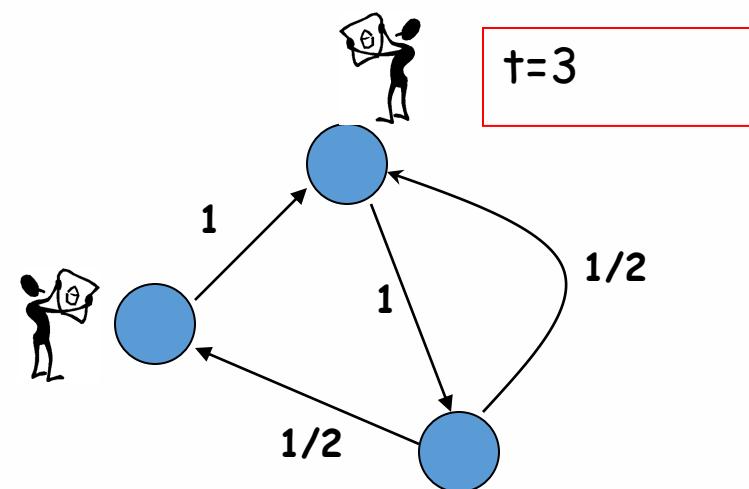
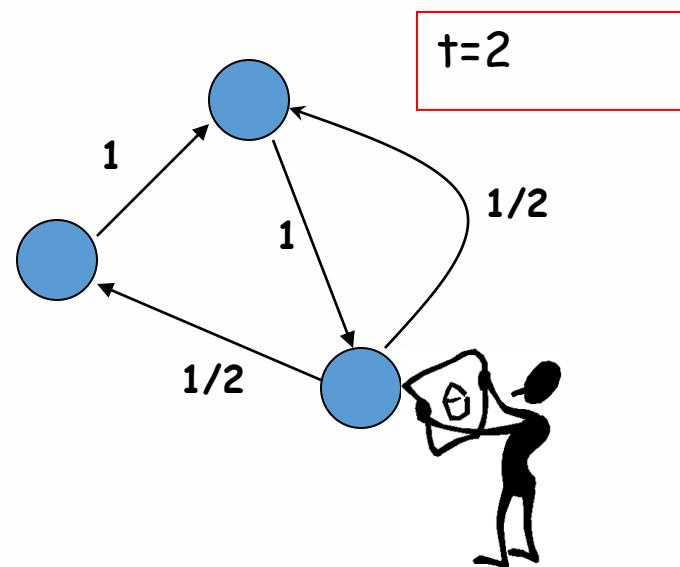
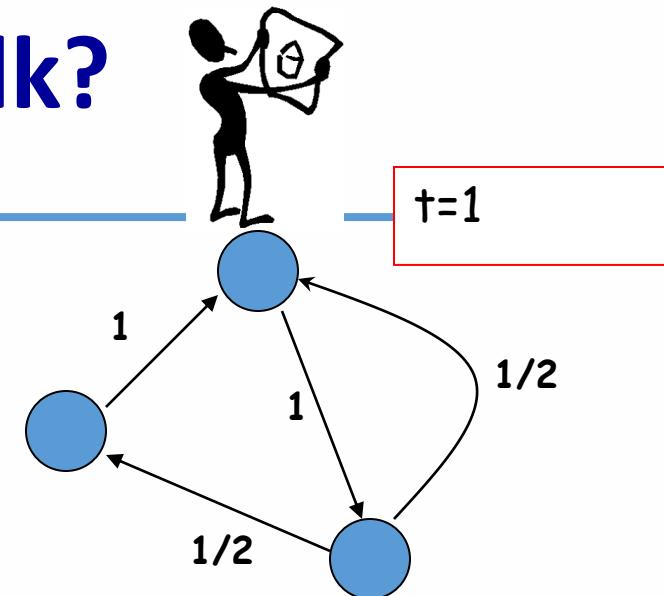
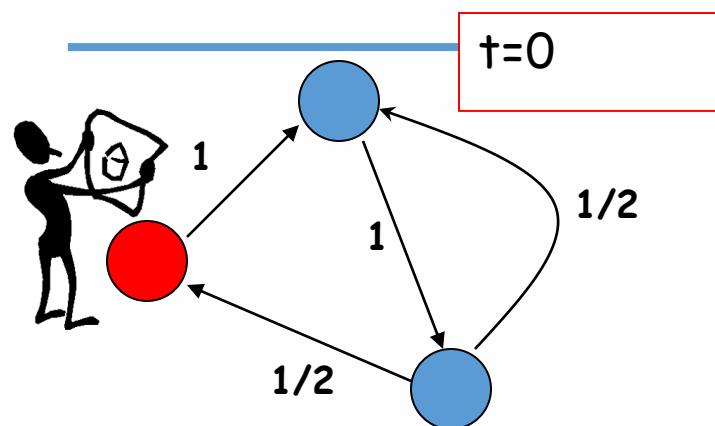


# Random Walk

---

- Kita telah mengenal DFS dan BFS
- Bagaimana jika traversal dilakukan secara acak?
  - Verteks berikutnya dipilih secara acak dengan semua tetangga yang dapat dikunjungi memiliki peluang yang sama
- Disebut *random walk*

# What is a random walk?





# Penerapan Random Walk

---

- Rekomendasi “who to follow” di Twitter
- Estimasi ukuran web
- Basis PageRank



# Problem Page Rank

- Google is a search engine owned by Google, Inc. whose mission statement is to "organize the world's information and make it universally accessible and useful". The largest search engine on the web, Google receives over 200 million queries each day through its various services.
- The heart of Google's searching software is PageRank™, a system for ranking web pages developed by Larry Page and Sergey Brin at Stanford University
- Essentially, Google interprets a link from page A to page B as a vote, by page A, for page B.
- **BUT** these votes doesn't weigh the same, because Google also analyzes the page that casts the vote



Domain Analysis For:

ipb.ac.id

Date: April 9 2023

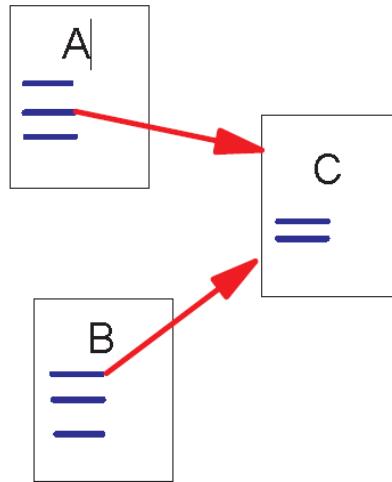
**Google PageRank: 7/10**  
**cPR Score: 7.8/10**

Domain Authority: 82  
Page Authority: 63  
Trust Flow: 26  
Trust Metric: 26  
Domain Validity: Found  
Global Rank: N/A  
Alexa USA Rank: N/A  
Alexa Reach Rank: N/A  
Spam Score: 1 / 18  
External Backlinks: 3,780,831  
Referring Domains: 15,579  
EDU Backlinks: 1,122,448  
EDU  
Domains: 607  
GOV Backlinks: 538  
GOV Domains: 32  
PR Quality: **Very Strong**  
Domain Age: NA  
HTTP Response Codes: 200  
Canonical  
URL: ipb.ac.id/  
Root IP: 3.1.214.219  
Title: NA  
Topic: World/Bahasa Indonesia  
Topic Value: 22  
Indexed URLs: 2,454,452  
Crawled Flag:  
False  
Google Directory listed: YES



# Link Structure of the Web

- 150 million web pages → 1.7 billion links



Backlinks and Forward links:

- A and B are C's backlinks
- C is A and B's forward link

Intuitively, a webpage is important if it has a lot of backlinks.

What if a webpage has only one link off [www.yahoo.com](http://www.yahoo.com)?



# A Simple Version of PageRank

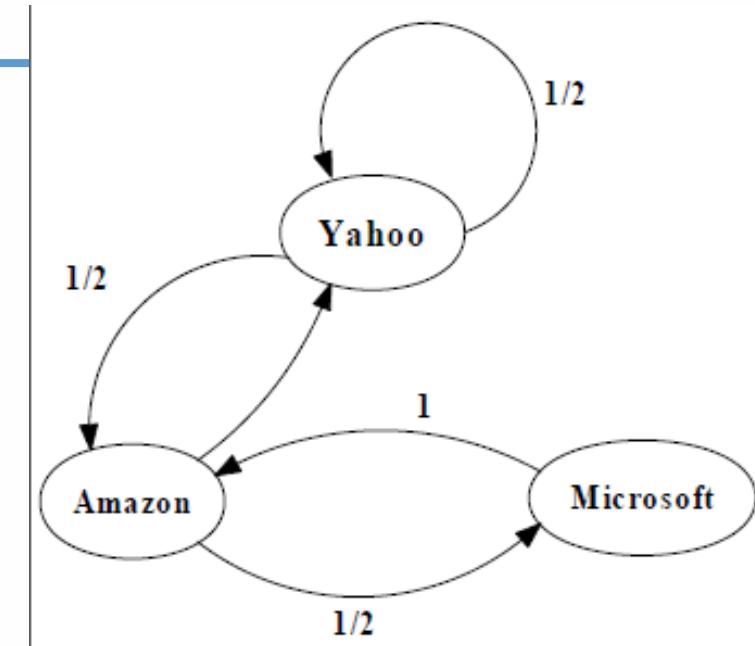
---

$$R(u) = c \sum_{v \in B_u} \frac{R(v)}{N_v}$$

- $u$ : a web page
- $B_u$ : the set of  $u$ 's backlinks
- $N_v$ : the number of forward links of page  $v$
- $c$ : the normalization factor to make  $\|R\|_{L1} = 1$   
 $(\|R\|_{L1} = |R_1 + \dots + R_n|)$



# An example of Simplified PageRank



$$M = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 1 \\ 0 & 1/2 & 0 \end{bmatrix}$$

$$\begin{bmatrix} \text{yahoo} \\ \text{Amazon} \\ \text{Microsoft} \end{bmatrix} = \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}$$

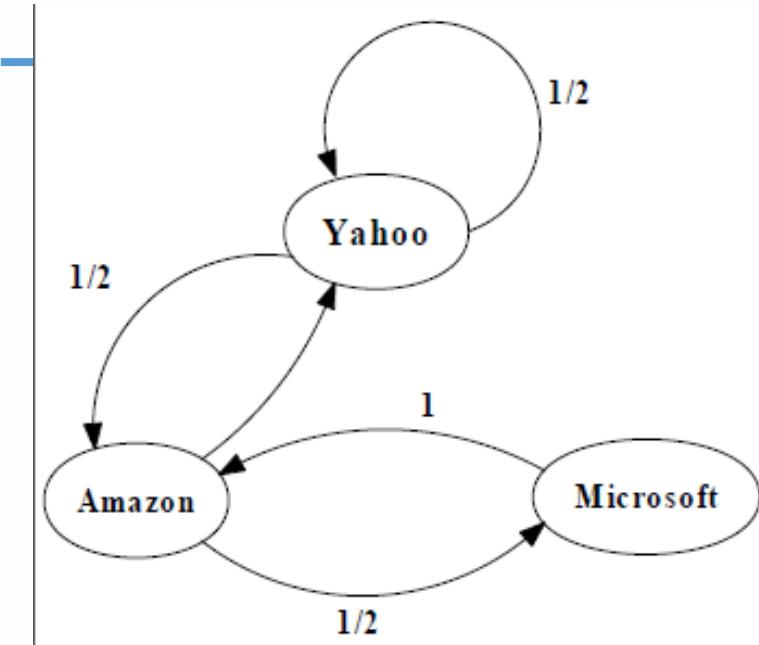
$$\begin{bmatrix} 1/3 \\ 1/2 \\ 1/6 \end{bmatrix} = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 1 \\ 0 & 1/2 & 0 \end{bmatrix} \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}$$

PageRank Calculation: first iteration

Searching & Serving the Best



# An example of Simplified PageRank



$$M = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 1 \\ 0 & 1/2 & 0 \end{bmatrix}$$

$$\begin{bmatrix} \text{yahoo} \\ \text{Amazon} \\ \text{Microsoft} \end{bmatrix} = \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}$$

$$\begin{bmatrix} 5/12 \\ 1/3 \\ 1/4 \end{bmatrix} = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 1 \\ 0 & 1/2 & 0 \end{bmatrix} \begin{bmatrix} 1/3 \\ 1/2 \\ 1/6 \end{bmatrix}$$

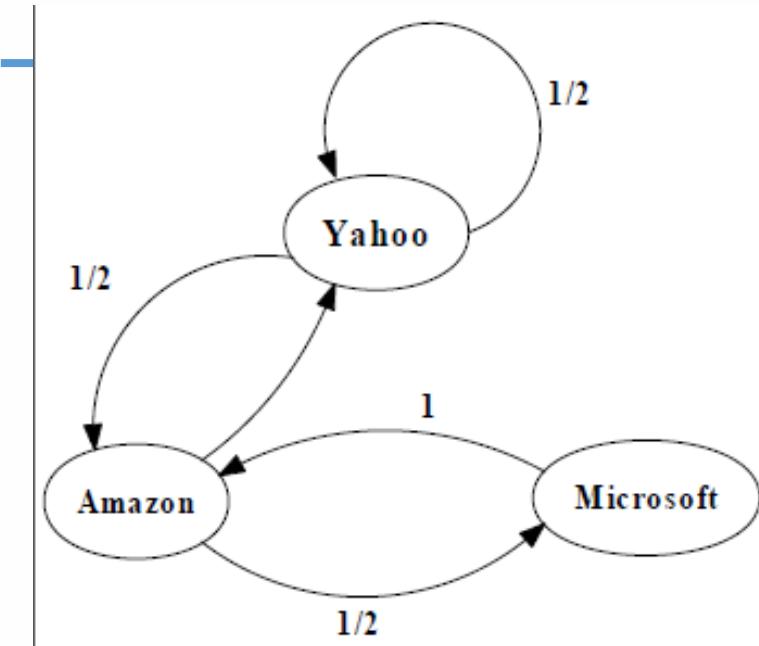
PageRank Calculation: second iteration

Searching & Serving the Best

<http://ipb.ac.id>



# An example of Simplified PageRank



$$M = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 1 \\ 0 & 1/2 & 0 \end{bmatrix}$$

$$\begin{bmatrix} \text{yahoo} \\ \text{Amazon} \\ \text{Microsoft} \end{bmatrix} = \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}$$

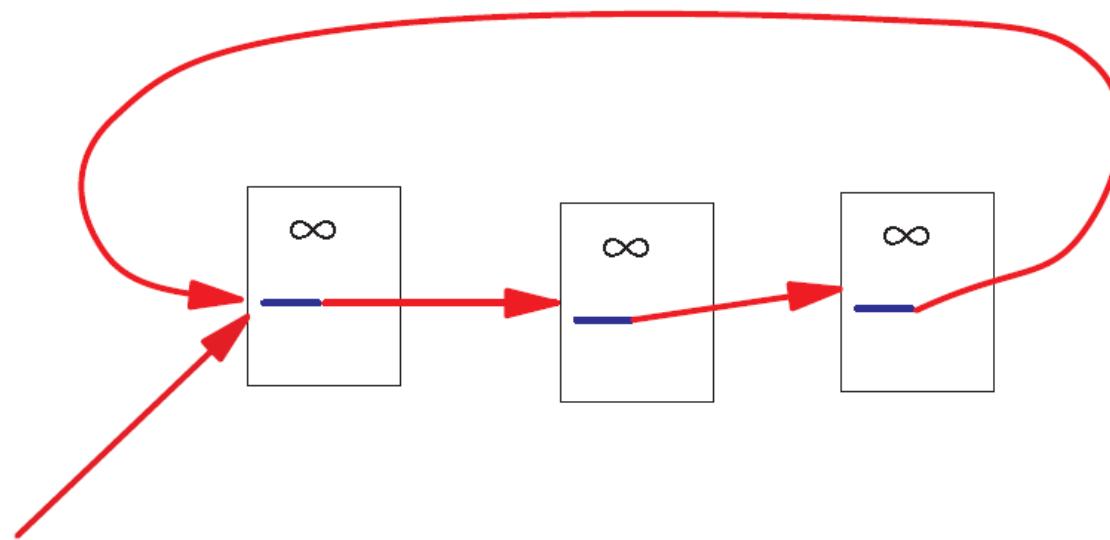
$$\begin{bmatrix} 3/8 \\ 11/24 \\ 1/6 \end{bmatrix} \begin{bmatrix} 5/12 \\ 17/48 \\ 11/48 \end{bmatrix} \dots \begin{bmatrix} 2/5 \\ 2/5 \\ 1/5 \end{bmatrix}$$

Convergence after some iterations

Searching & Serving the Best

# A Problem with Simplified PageRank

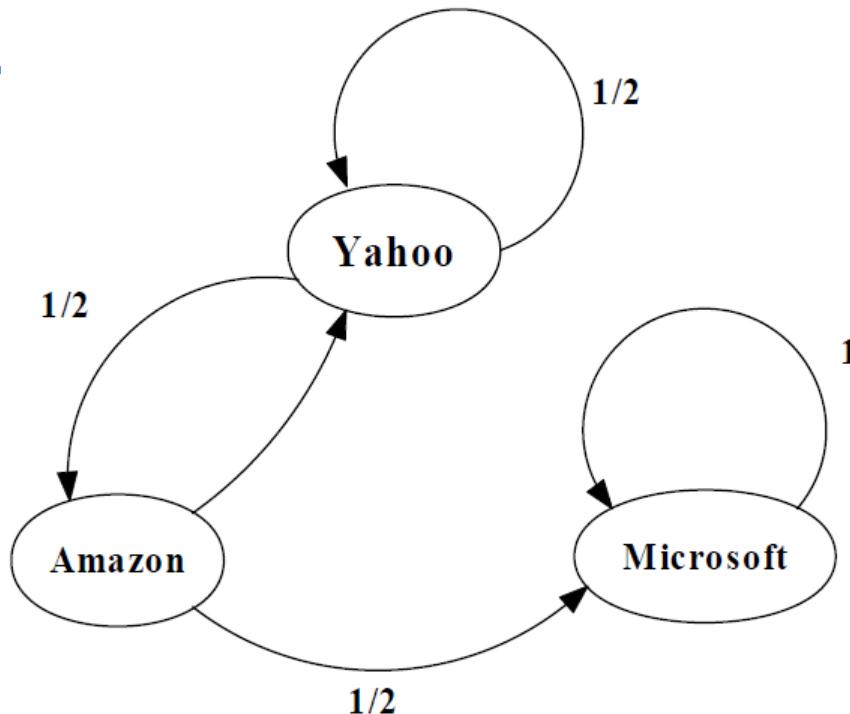
A loop:



During each iteration, the loop accumulates rank but never distributes rank to other pages!



# An example of the Problem



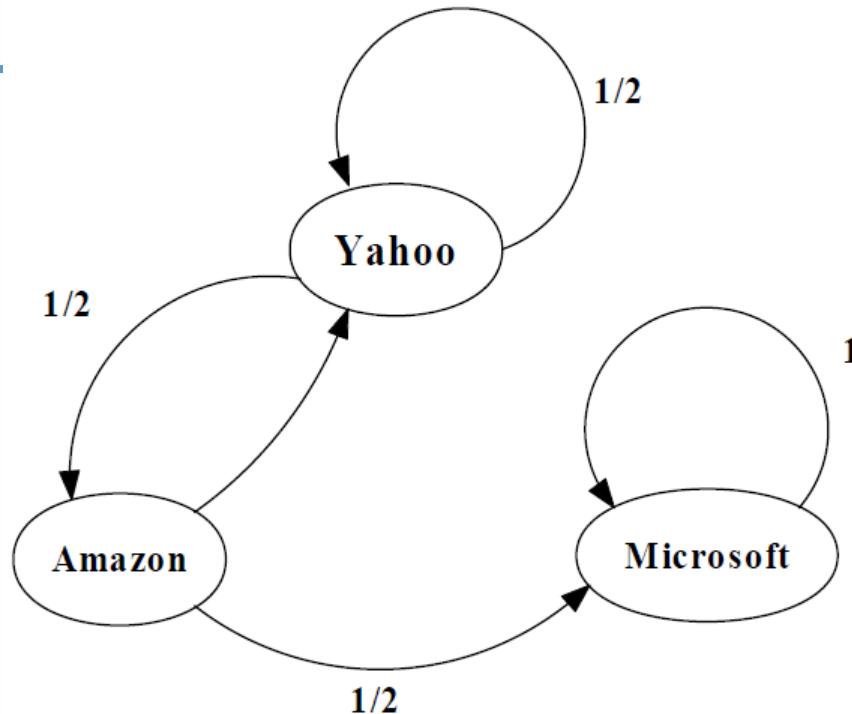
$$M = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 0 \\ 0 & 1/2 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \text{yahoo} \\ \text{Amazon} \\ \text{Microsoft} \end{bmatrix} = \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}$$

$$\begin{bmatrix} 1/3 \\ 1/6 \\ 1/2 \end{bmatrix} = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 0 \\ 0 & 1/2 & 1 \end{bmatrix} \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}$$



# An example of the Problem



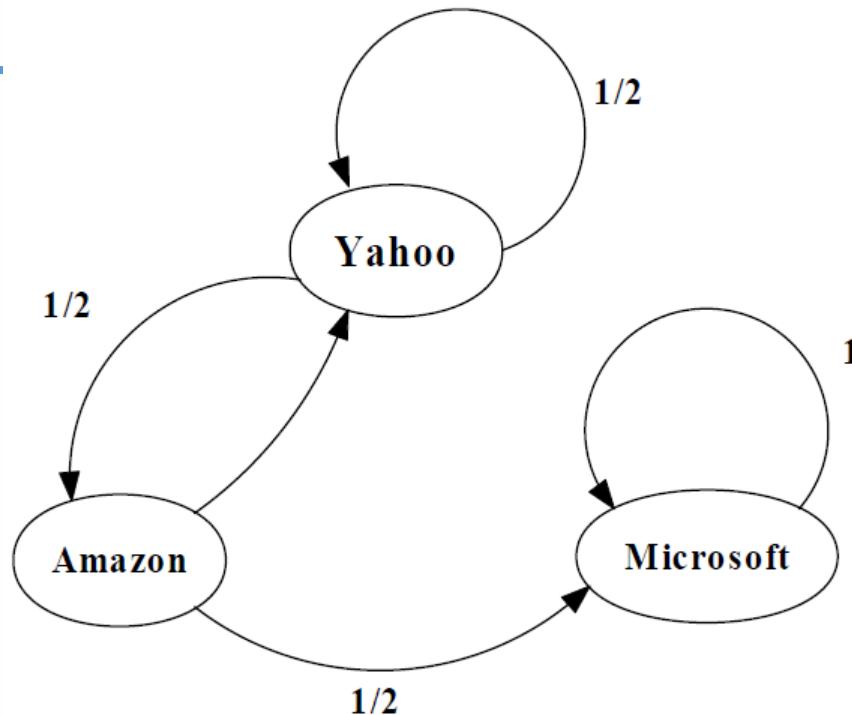
$$M = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 0 \\ 0 & 1/2 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \text{yahoo} \\ \text{Amazon} \\ \text{Microsoft} \end{bmatrix} = \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}$$

$$\begin{bmatrix} 1/4 \\ 1/6 \\ 7/12 \end{bmatrix} = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 0 \\ 0 & 1/2 & 1 \end{bmatrix} \begin{bmatrix} 1/3 \\ 1/6 \\ 1/2 \end{bmatrix}$$



# An example of the Problem



$$M = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 0 \\ 0 & 1/2 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \text{yahoo} \\ \text{Amazon} \\ \text{Microsoft} \end{bmatrix} = \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}$$

$$\begin{bmatrix} 5/24 \\ 1/8 \\ 2/3 \end{bmatrix} \begin{bmatrix} 1/6 \\ 5/48 \\ 35/48 \end{bmatrix} \dots \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$



# Random Walks in Graphs

- The Random Surfer Model
  - The simplified model: the standing probability distribution of a random walk on the graph of the web. simply keeps clicking successive links at random
- The Modified Model
  - The modified model: the “random surfer” simply keeps clicking successive links at random, but periodically “gets bored” and jumps to a random page based on the distribution of E



# Modified Version of PageRank

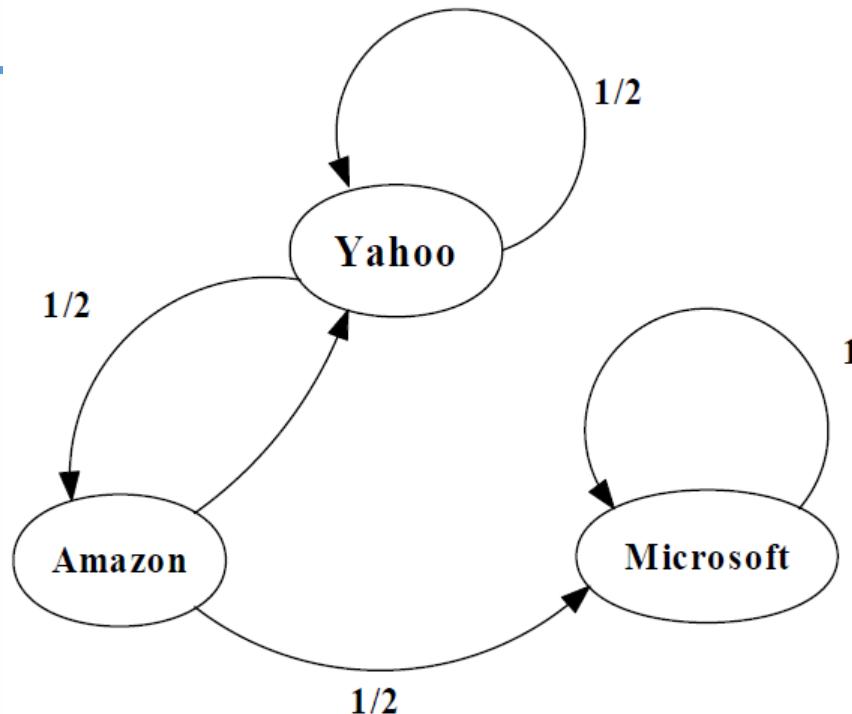
---

$$R'(u) = \textcolor{brown}{C_1} \sum_{v \in B_u} \frac{R'(v)}{N_v} + \textcolor{brown}{C_2} E(u)$$

E(u): a distribution of ranks of web pages that “users” jump to when they “gets bored” after successive links at random.



# An example of Modified PageRank



$$M = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 0 \\ 0 & 1/2 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \text{yahoo} \\ \text{Amazon} \\ \text{Microsoft} \end{bmatrix} = \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}$$

$$C_1 = 0.8 \quad C_2 = 0.2$$

$$\begin{bmatrix} 0.333 \\ 0.333 \\ 0.333 \end{bmatrix} \begin{bmatrix} 0.333 \\ 0.200 \\ 0.467 \end{bmatrix} \begin{bmatrix} 0.280 \\ 0.200 \\ 0.520 \end{bmatrix} \begin{bmatrix} 0.259 \\ 0.179 \\ 0.563 \end{bmatrix} \dots \begin{bmatrix} 7/33 \\ 5/33 \\ 21/33 \end{bmatrix}$$



Terima kasih

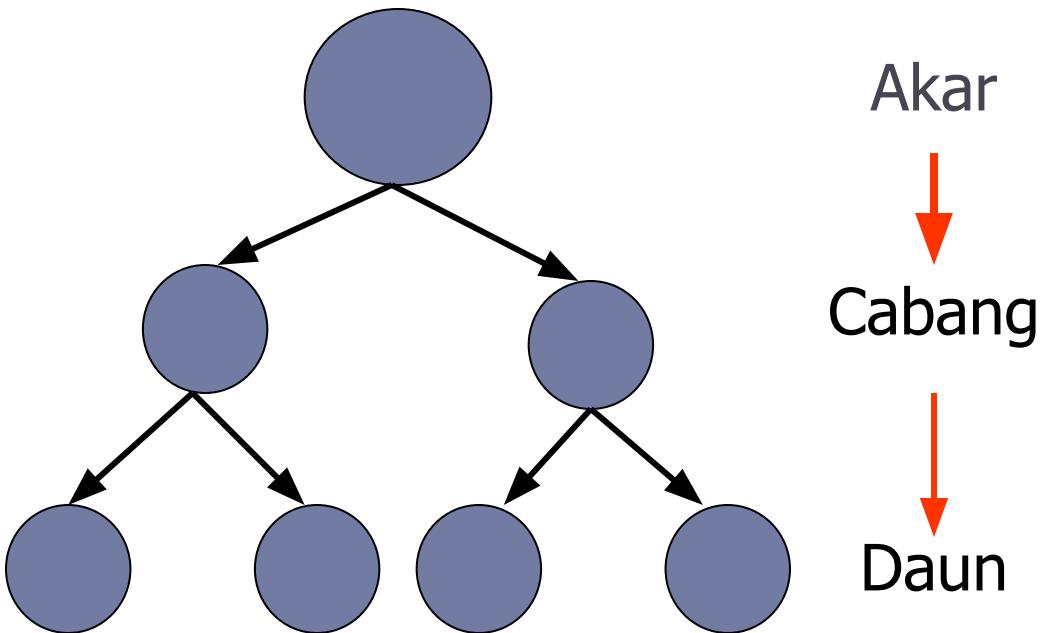
# **Struktur Data berhirarki**

## **KOM20H**

**Oleh: Tim Pengajar Struktur Data**

# Pendahuluan

Representasi struktur data berhirarkhi

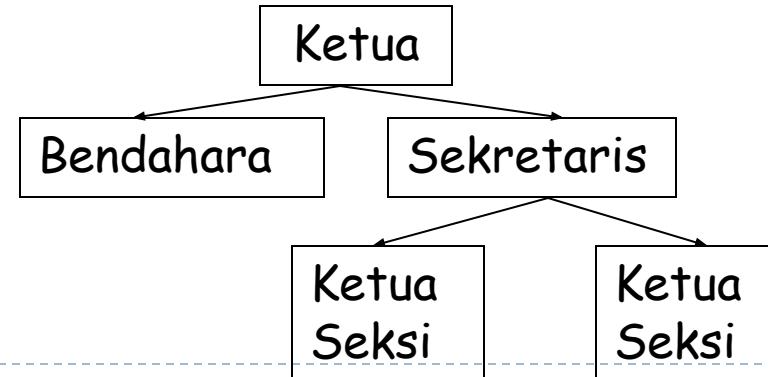


**Pohon Biner**

□ Mengapa dibutuhkan pengolahan data dengan struktur berhirarki?

# Pendahuluan

- Ada problem yang secara alamiah memang berhierarchy: silsilah, kepengurusan organisasi, taksonomi, dsb
- Pengorganisasian informasi atau data : data / informasi global dibagi-bagi berdasarkan tingkatannya
- Memudahkan pengambilan dan penyusunan informasi (proses retrievedata) :
  - secara linier □ ketua, bendahara, sekretaris1, ketua seksi, anggota seksi, dll.....
  - secara hirarkhi



# Pendahuluan

---

## Penggunaan Tree dalam Struktur Hirarki

- **Struktur Tetap**

Sejak awal sampai akhir program tetap (hampir tidak mengalami perubahan susunan).

Contoh : Pohon Silsilah

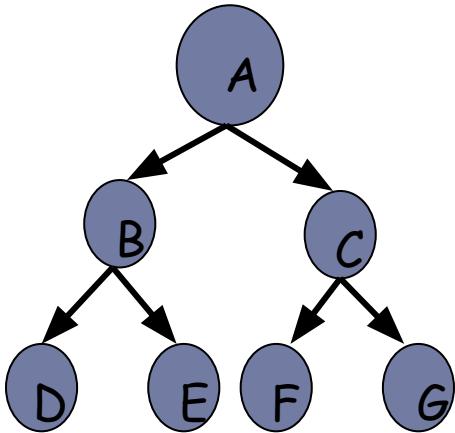
- **Struktur Dinamis**

Senantiasa berubah dari awal sampai akhir program (perlu dilengkapi dengan fasilitas untuk menangani perubahan susunan)



# Review

---



## Terminologi:

### Tree

Sejumlah node yang berhubungan secara hirarkis dimana node-node pada hirarki lebih rendah merupakan cabang dari node dengan hirarki yang lebih tinggi

### Root

Node dari suatu tree yang memiliki hirarki paling tinggi

### Leaf

Node yang tidak mempunyai cabang

### Inner

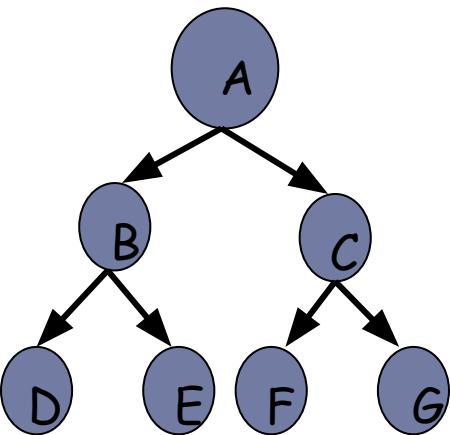
Node yang bukan leaf

### Edge

Penghubung antara dua buah node



# Review



## Node Predecessor

Node yang berada di atas node tertentu

## Node Successor

Node yang berada di bawah node tertentu

## Ancestor/nenek moyang

Seluruh node yang terletak sebelum node tertentu pada jalur yang sama

## Descendant/Keturunan

Seluruh node yang terletak sesudah node tertentu pada jalur yang sama

## Parent (ayah)

Node pada hirarki langsung diatas node tersebut  
(predecessor satu level)

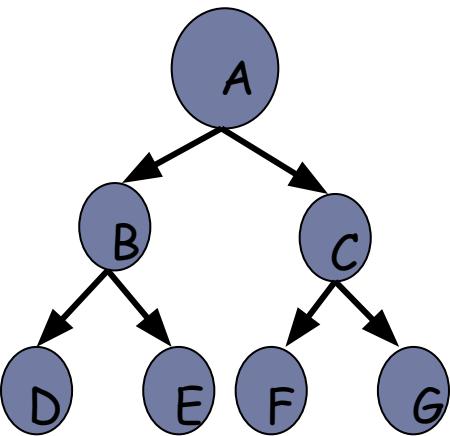
## Child

Node pada hirarki langsung dibawah node tersebut  
(successor satu level)



# Review

---



## Sibling

Node-node yang satu parent, sehingga berlevel sama.  
Dikenal juga istilah sibling yang tepat di kanan dan  
sibling yang tepat di kiri

## Subtree

Bagian dari tree berupa suatu node beserta  
descendantnya

## Size

Banyaknya node dalam suatu tree

## Level

Tingkatan suatu node. Root level 0, node dibawahnya  
level 1 dan seterusnya.

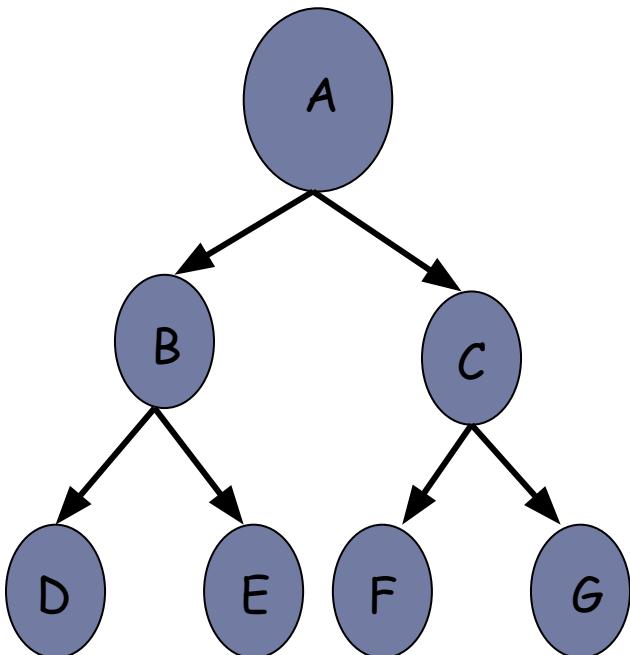
## Height

Ketinggian suatu tree dihitung dari bawah. Node leaf  
memiliki height 0



# Review

---



Size:

Root :

Leaf:

Ancestor (F):

Descendent (C) :

Parent (D) :

Child (A) :

Sibling (F) :

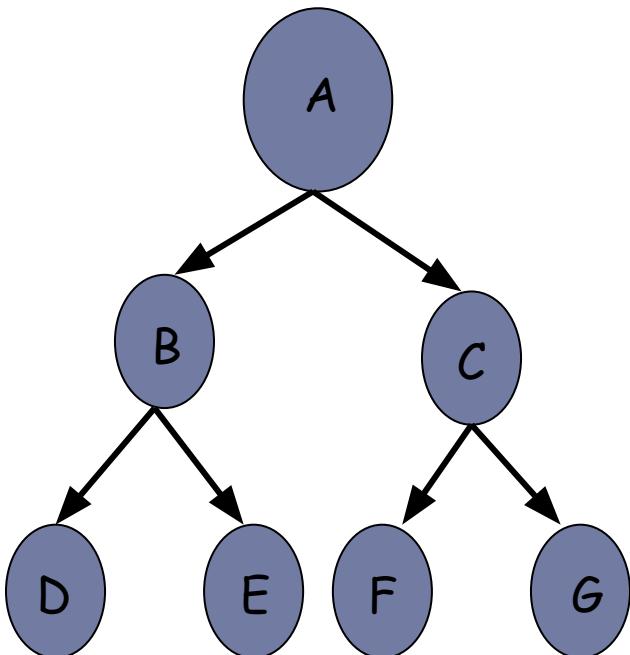
Depth (F) :

Heigth(A):



# Review

---



Size: 7

Root : A

Leaf: D, E, F, G

Ancestor (F): C, dan A

Descendent (C) : F, dan G

Parent (D) : B

Child (A) : B, dan C

Sibling (F) : G

Depth (F) : 2

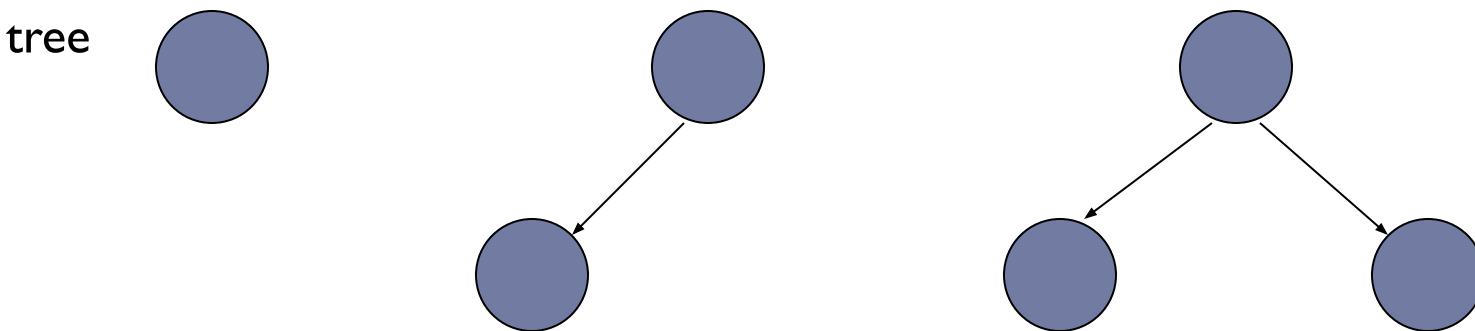
Heigth(A): 2



# Binary Tree

---

- Tree dengan cabang tidak lebih dari dua (hanya 0, 1, atau 2).
- Tree dengan seluruh innernya paling banyak berderajat 2.
- Cabang-cabang binary tree dibedakan menjadi cabang kiri dan cabang kanan.
- Jika hanya dijumpai satu cabang, harus ditentukan, cabang kiri atau kanan.
- Complete binary tree, full binary tree, perfect binary tree, skewed binary tree

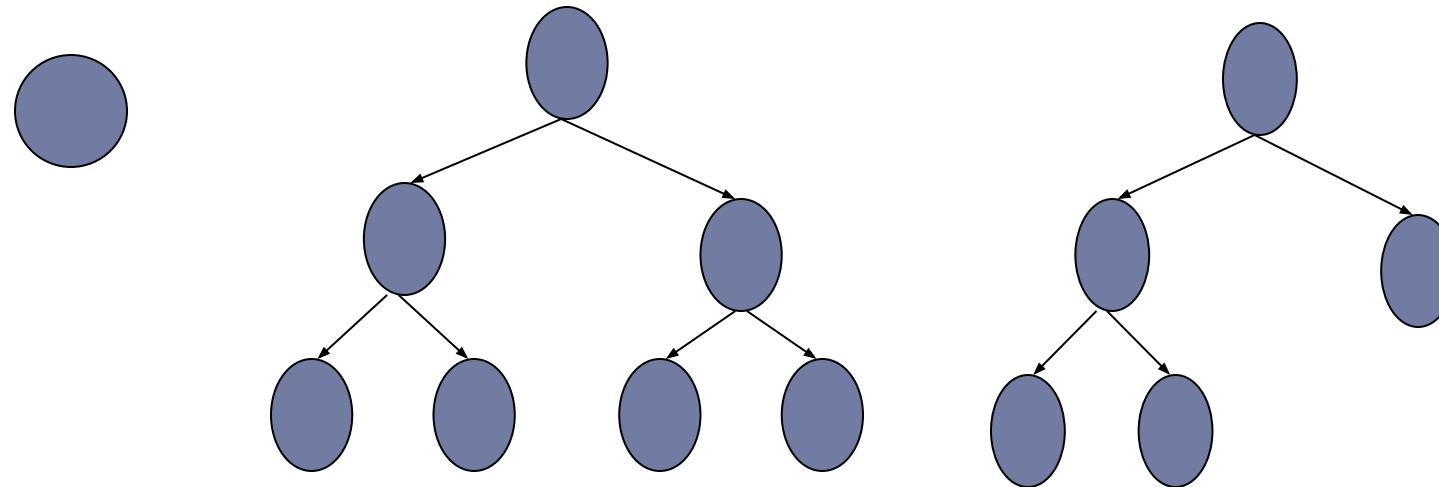


# Full Binary Tree

---

Binary tree yang :

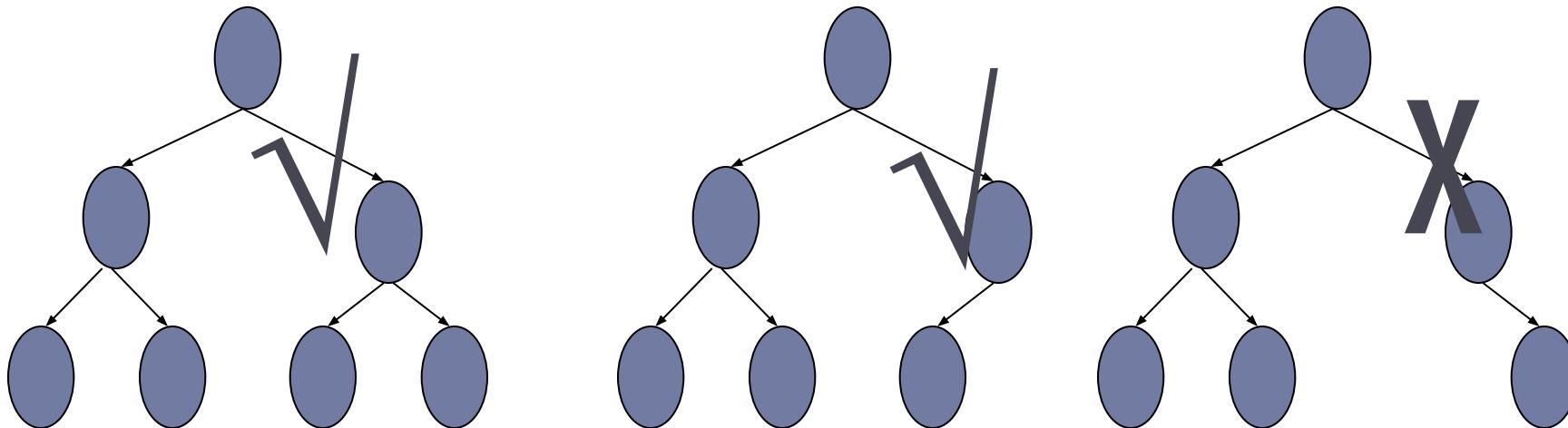
Setiap node memiliki tepat 0 atau 2 child



# Complete Binary Tree

Binary tree yang :

1. Setiap leaf memiliki depth n or n-1
2. Setiap leaf pada depth n / pada level terendah merapat ke kiri

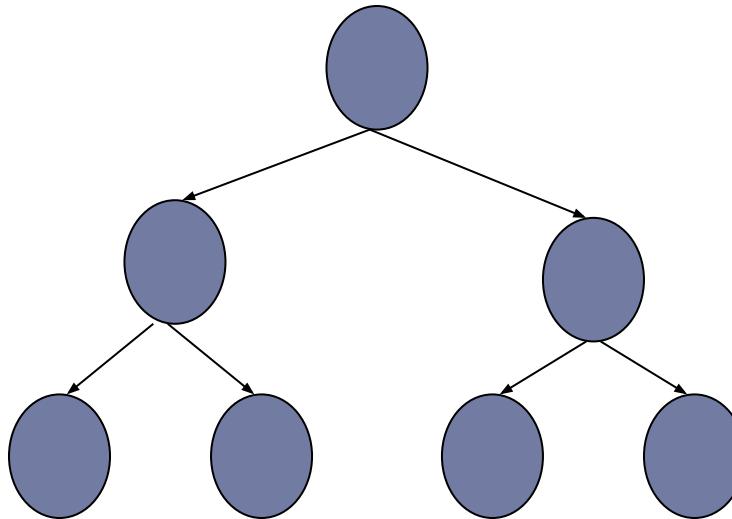


# Perfect Binary Tree

---

Binary tree yang :

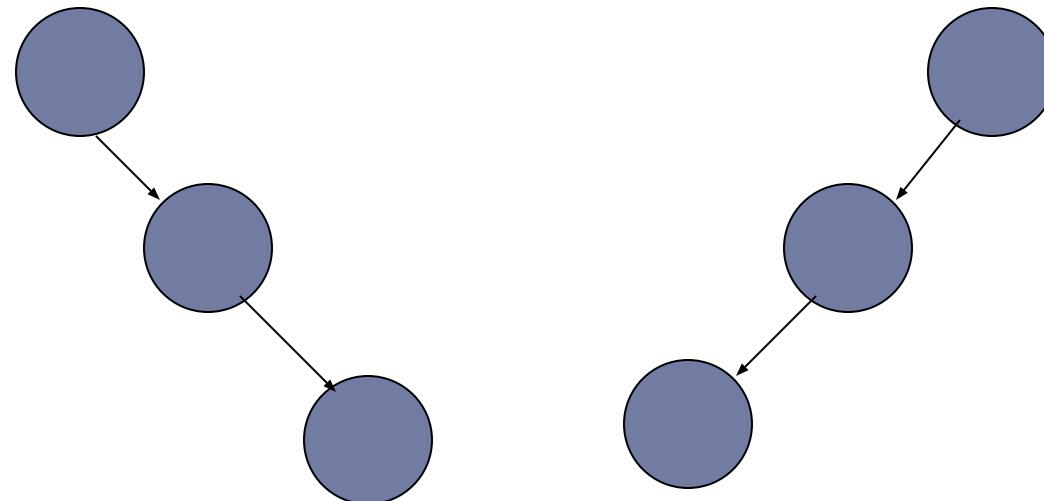
1. Setiap inner memiliki tepat dua child
2. Setiap leaf terletak pada depth yang sama



# Skewed Binary Tree

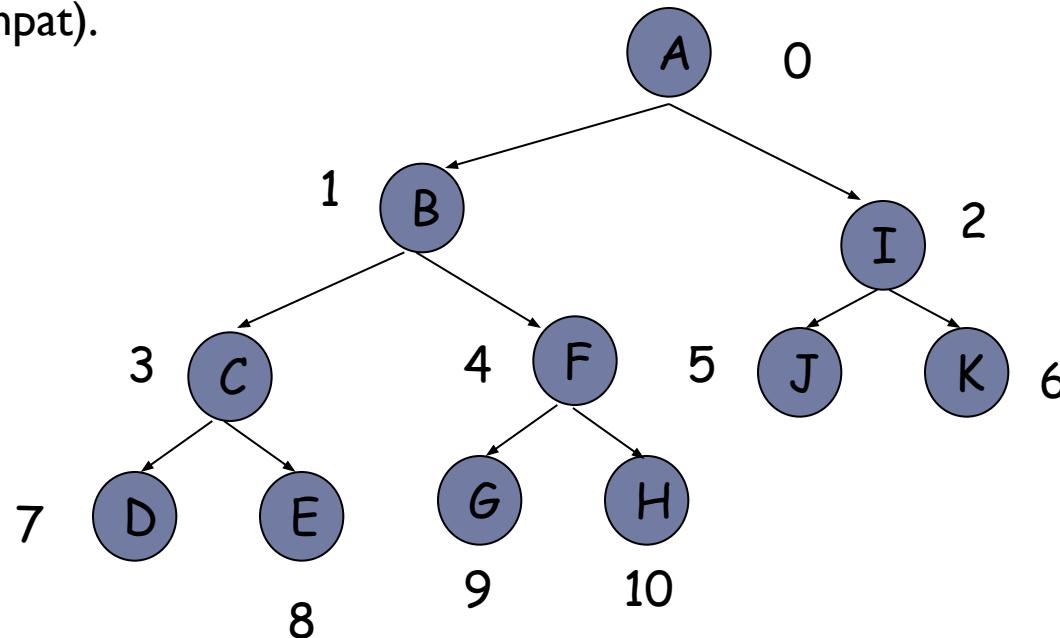
---

Binary tree yang semua nodenya  
kecuali leaf hanya memiliki satu child



# Representasi dengan Array

- CBT dan PBT dapat diimplementasikan dalam array
- Penomoran pada node-nodenya bersifat terurut dan tidak ada yang kosong (melompat).



Dalam Array

A	B	I	C	F	J	K	D	E	G	H
0	1	2	3	4	5	6	7	8	9	10



# Representasi dengan Array

---

**Contoh beberapa hubungan struktural representasi array :**

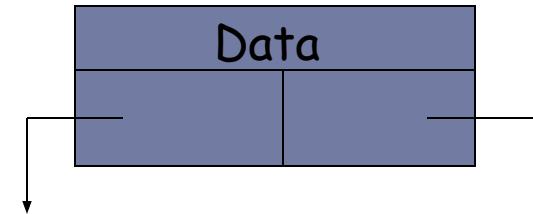
- Mendapatkan anak dari suatu node :
  1. Anak kiri dari node  $i \square 2i + 1$
  2. Anak kanan dari node  $i \square 2i + 2$
- Mendapatkan ayah dari suatu node :
  1. Ayah dari node  $i \square [(i-1)/2]$
- Sibling dari node ke  $i$  :
  1. Jika  $i$  genap, siblingnya adalah node  $i-1$
  2. Jika  $i$  genap, siblingnya adalah node  $i+1$



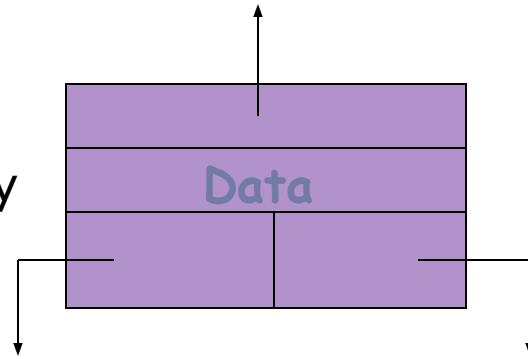
# Representasi Dengan Pointer

- Menggunakan Link list (Reference)

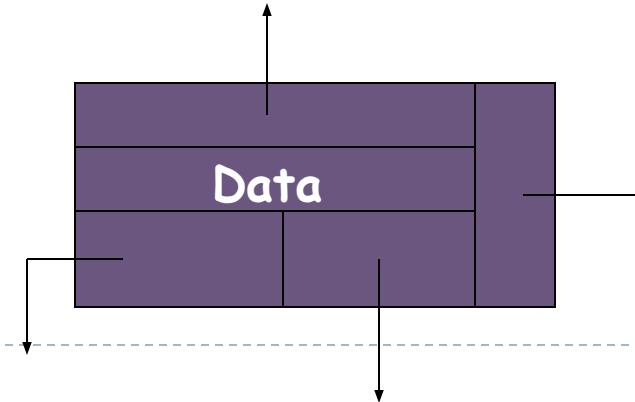
Bentuk paling sederhana (one-way)



Bentuk 2-way



Bentuk yang lebih lengkap



# Traversal / Kunjungan

---

Traversal : Mengunjungi setiap node pada tree

Ada tiga cara mengunjungi node-node pada suatu tree :

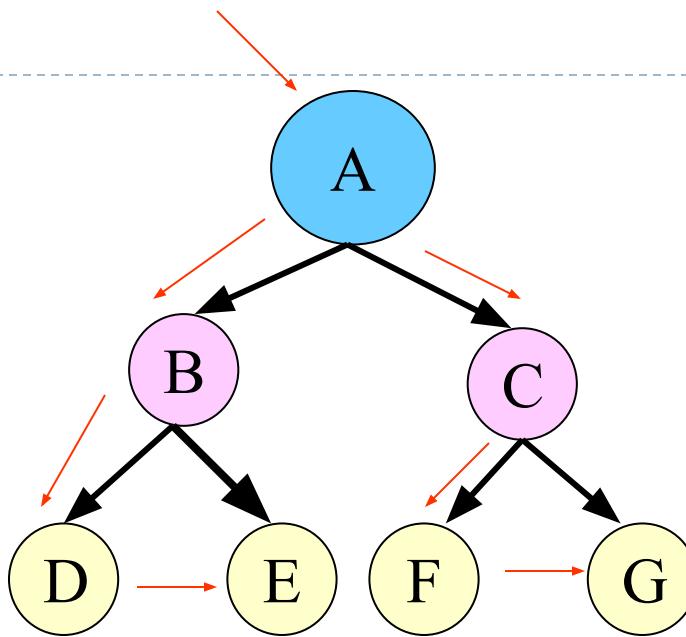
**Preorder : parent, kiri, kanan**

**Inorder : kiri, parent, kanan**

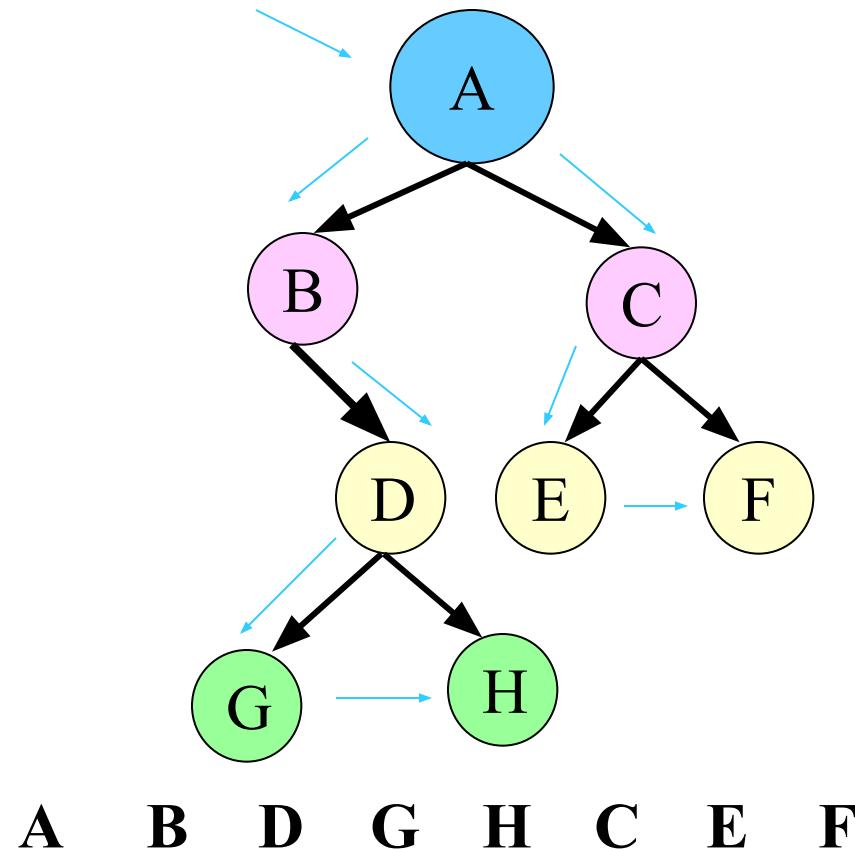
**Postorder : kiri, kanan, Parent**



## Preorder : parent, kiri, kanan

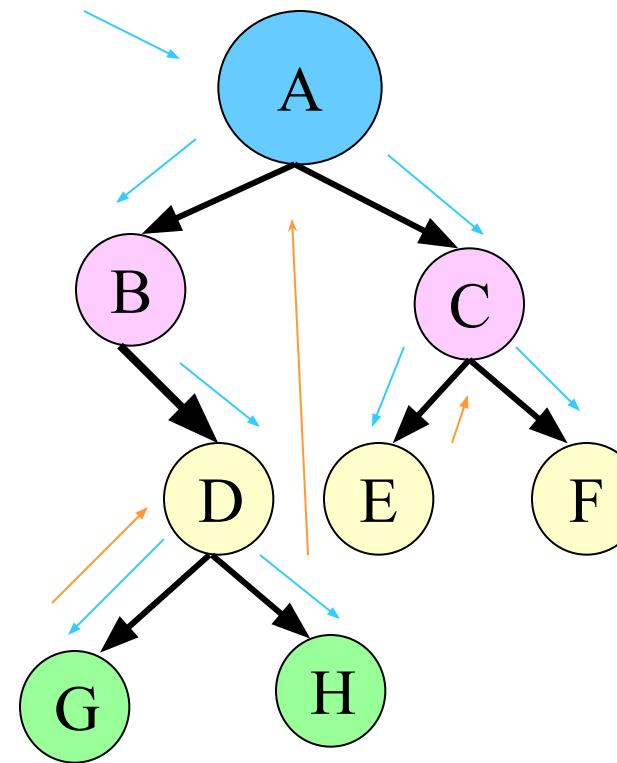


A   B   D   E   C   F   G



A   B   D   G   H   C   E   F

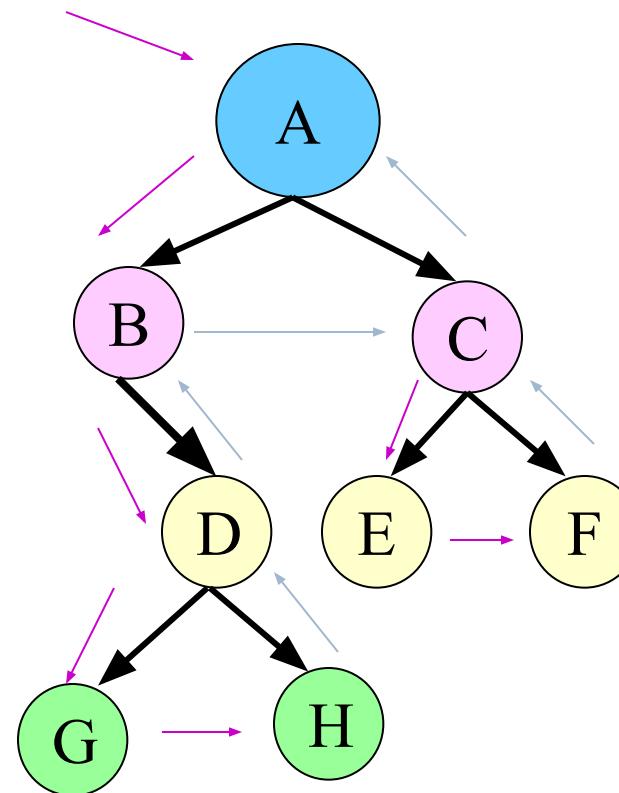
## Inorder : kiri, parent, kanan



B    G    D    H    A    E    C    F

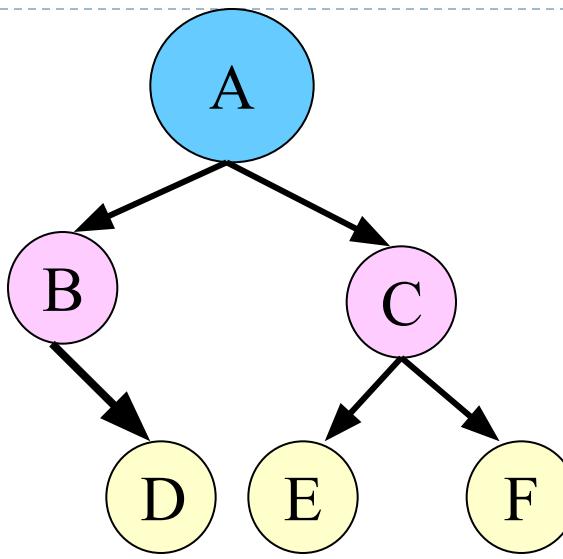


## Postorder : kiri, kanan, parent



G   H   D   B   E   F   C   A

## Try This One ! – Quiz di LMS



**Pre-Order: ?**

**In-Order: ?**

**Post-Order: ?**



# HEAP TREE

---

**Suatu tree dikatakan Heaptree** Jika tree tersebut **binary, complete** dan seluruh nodenya **heapordered**.

**Suatu node dikatakan heapordered jika :**

Nilai pada node tersebut lebih besar sama dengan atau lebih kecil sama dengan nilai child dibawahnya.

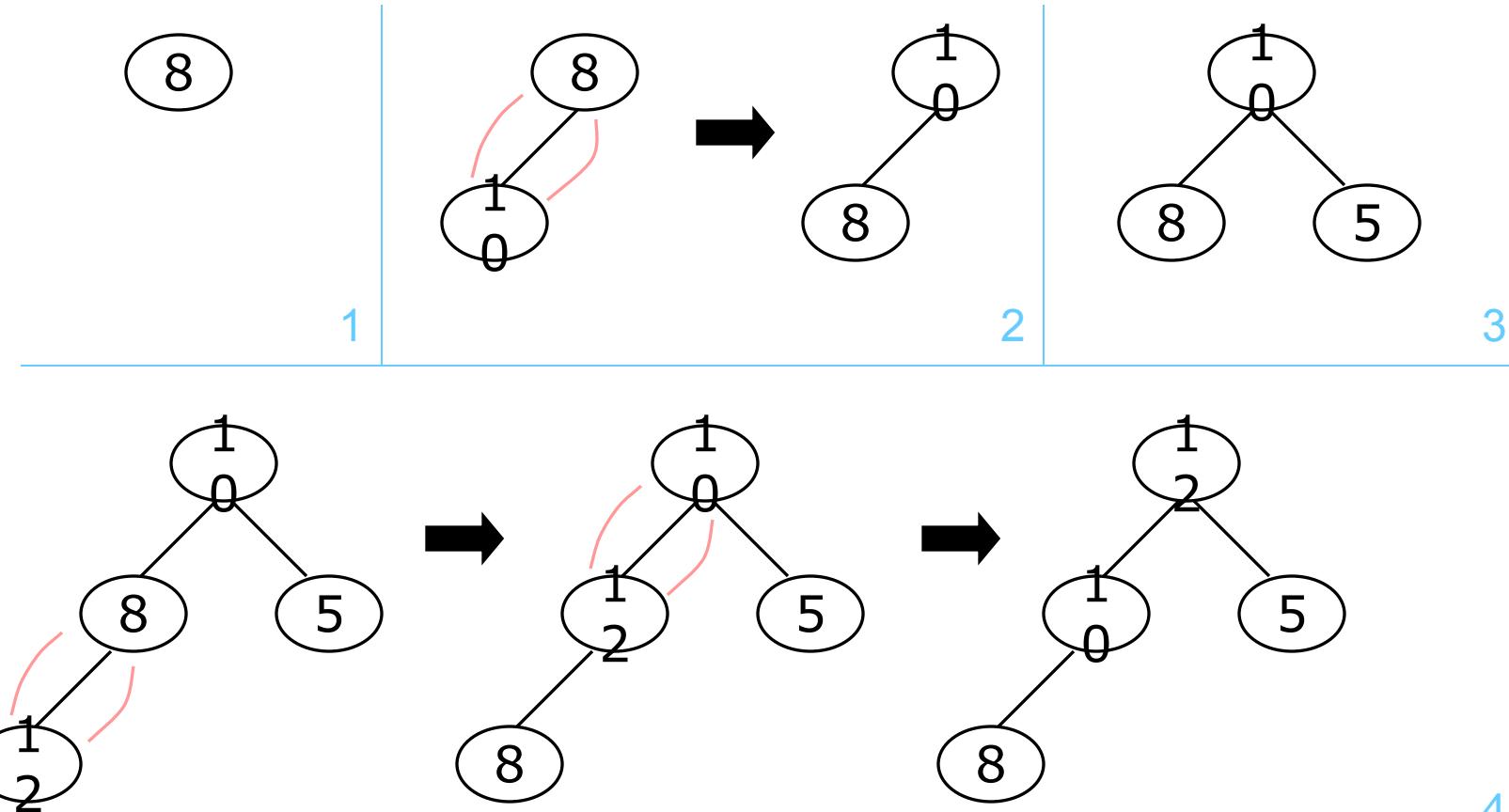
Lebih kecil sama dengan dan lebih besar sama dengan tergantung dari definisi prioritas. (yang kecil atau yang besar yang diprioritaskan).

**Binary:** Setiap node paling banyak memiliki dua child.

**Complete:** Setiap leaf memiliki depth n or n-1, dan setiap leaf pada depth n / pada level terendah merapat ke kiri. Tidak ada penomoran yang melompat.

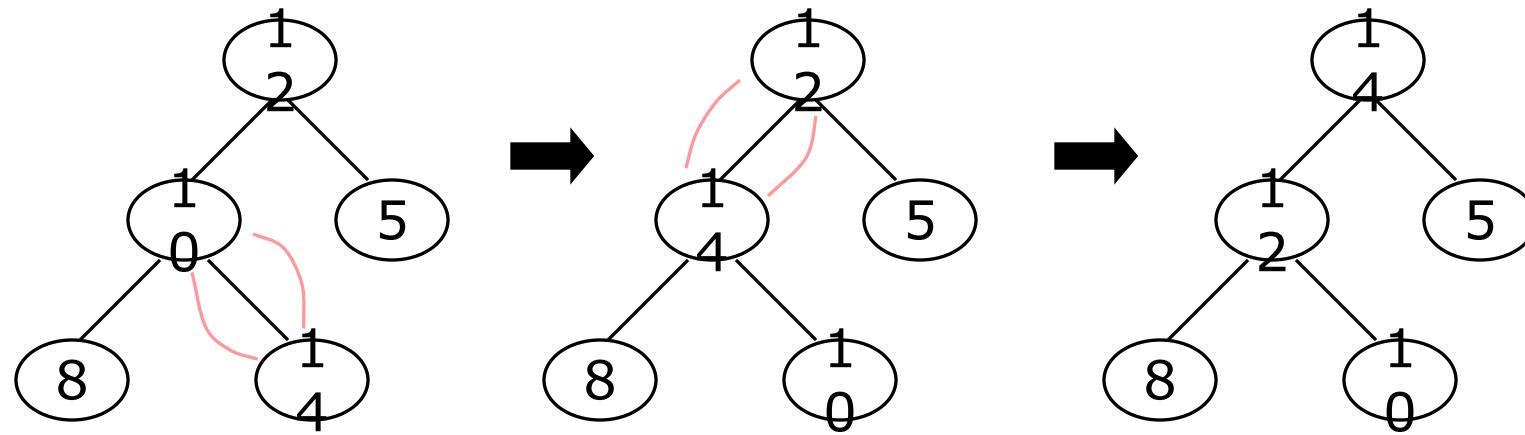
**Heapordered:** Setiap node terorganisasi berdasarkan suatu ketentuan prioritas





Konstruksi Heap





- Node bernilai 8 tidak terpengaruh karena parentnya lebih besar bukan lebih kecil
- Node bernilai 5 tidak terpengaruh karena parentnya lebih besar bukan lebih kecil
- Node bernilai 8 tidak terpengaruh karena parentnya lebih besar bukan lebih kecil



# Latihan

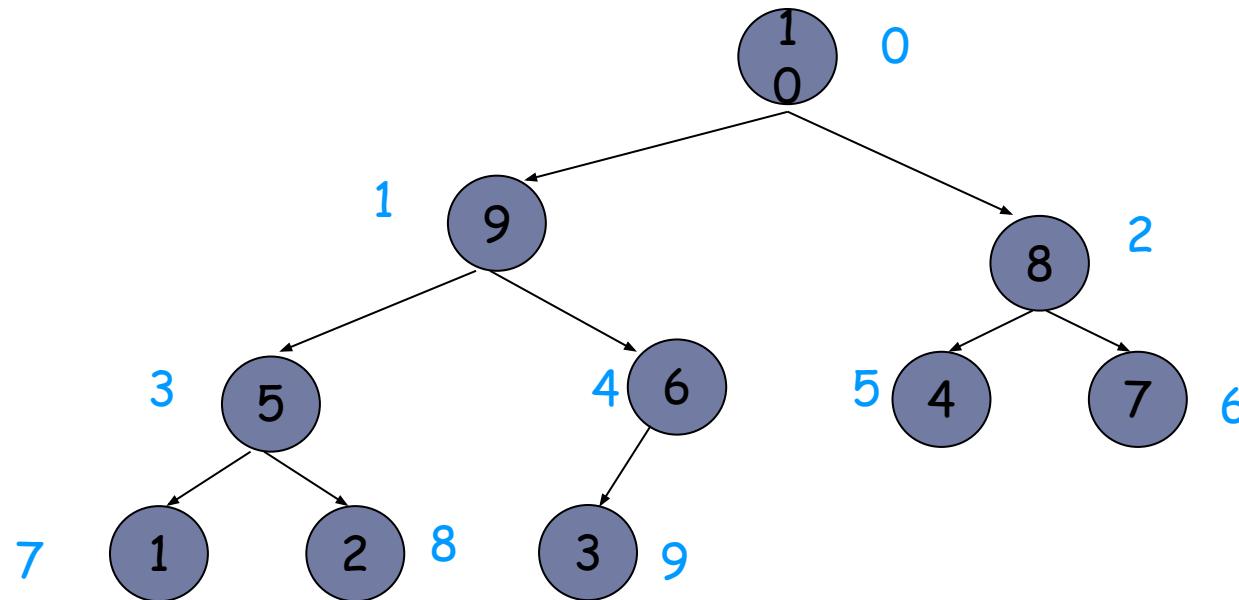
---

**Bentuklah sebuah heap tree dari masukan berikut : 2, 4, 5, 7, 3, 10, 8, 1, 9, 6 dengan prioritas pada nilai yang lebih besar !**

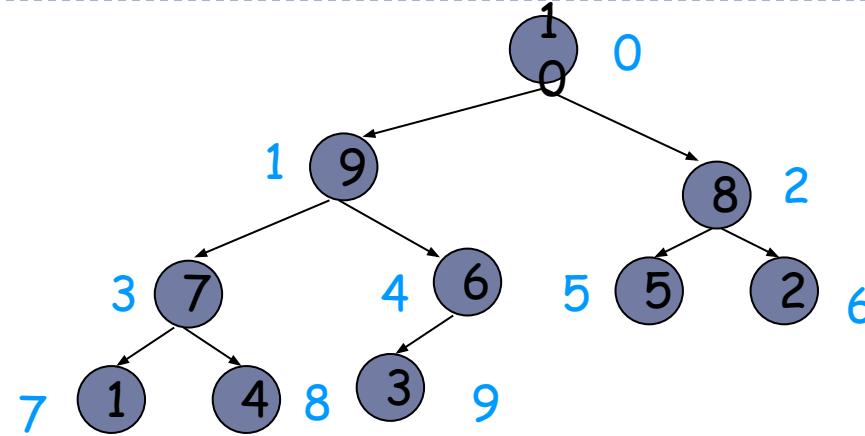


# Latihan

Bentuklah sebuah heap tree dari masukan berikut : 2, 4, 5, 7, 3, 10, 8, 1, 9, 6 dengan prioritas pada nilai yang lebih besar !



# Operasi Heaptree



## a. Operasi Penghapusan

1. Penghapusan dilakukan pada root.
2. Move node dengan indeks terbesar ke root.
3. Reheapify dari atas.

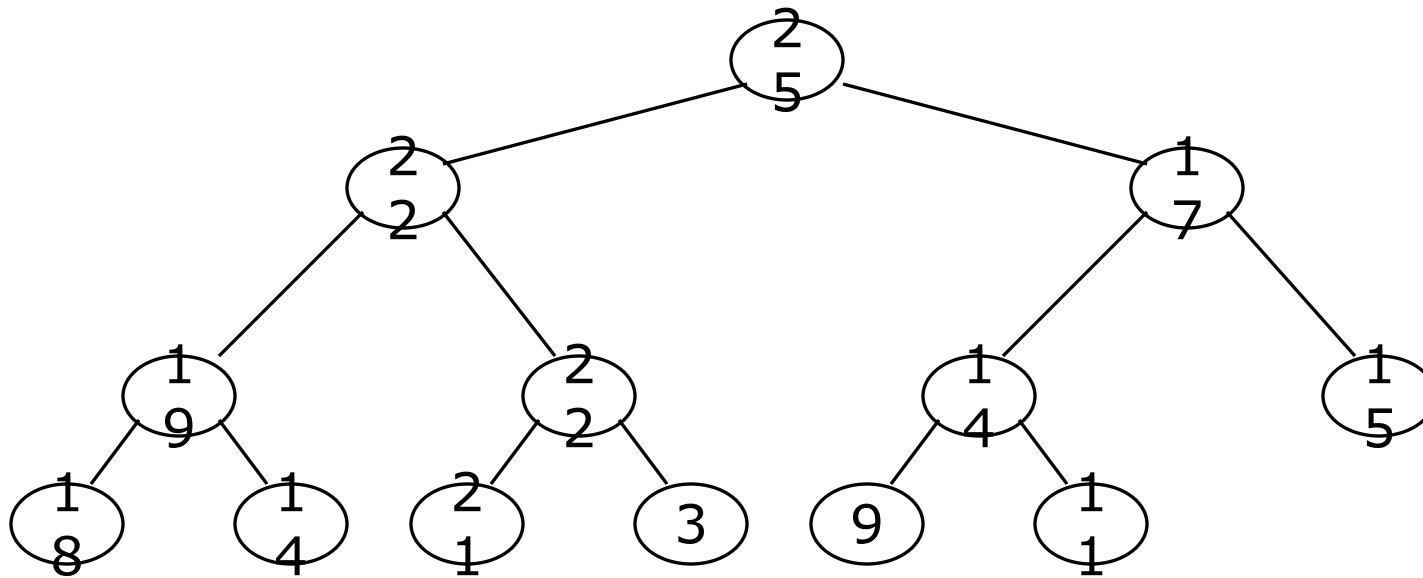
## b. Operasi Penyisipan

1. Penyisipan dilakukan pada indeks terbesar.
2. Reheapify dari bawah.



# Contoh Heap

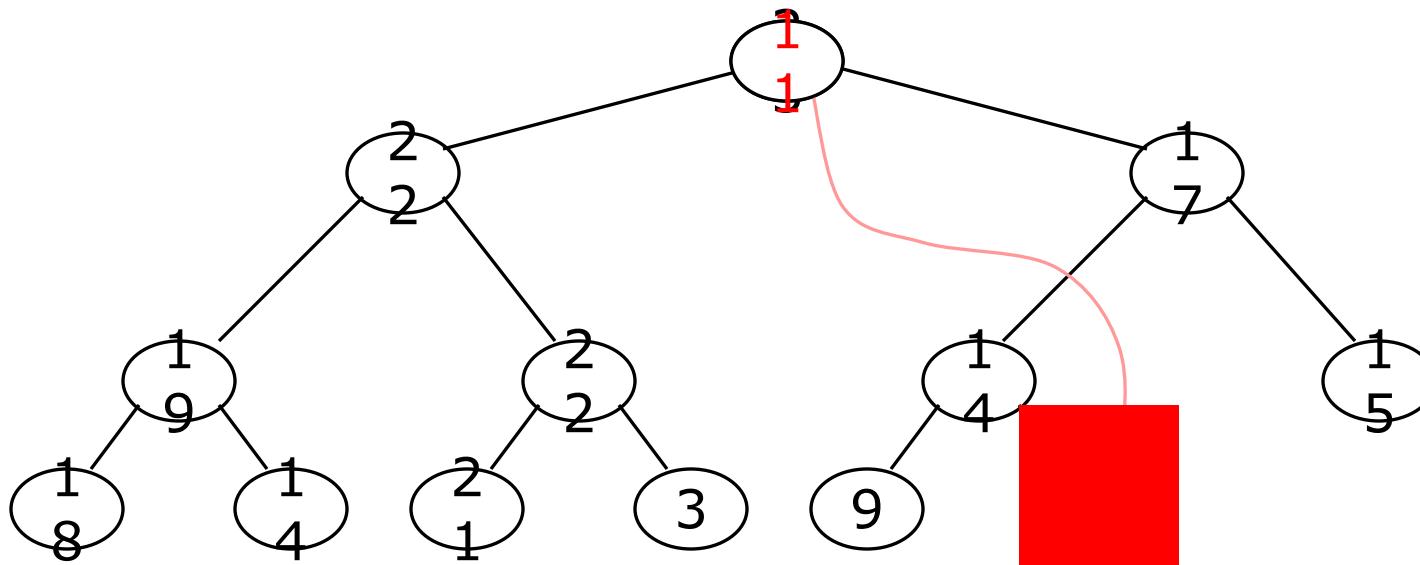
- Berikut Contoh Heaptree



- Lakukan operasi penghapusan dan penambahan pada Heap Tree



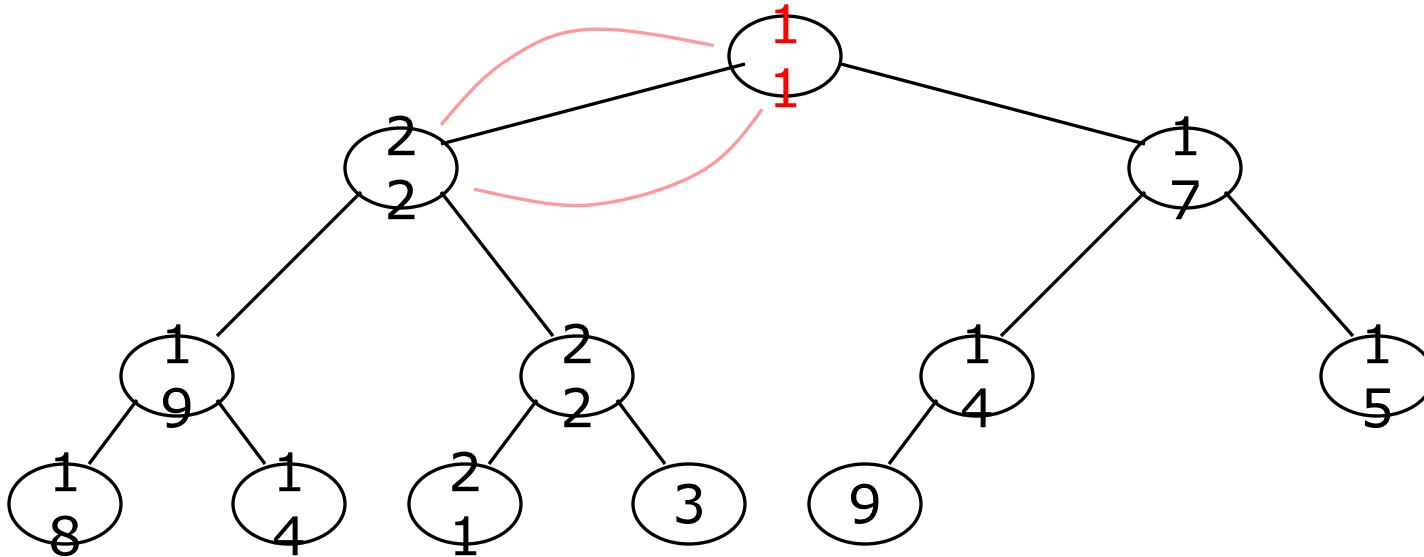
# Penghapusan (Deletion)



- Hilangkan leaf pada index terbesar (level terdalam) dan pindahkan nilai leaf tersebut menjadi root baru
- Lakukan Re-Heapfy



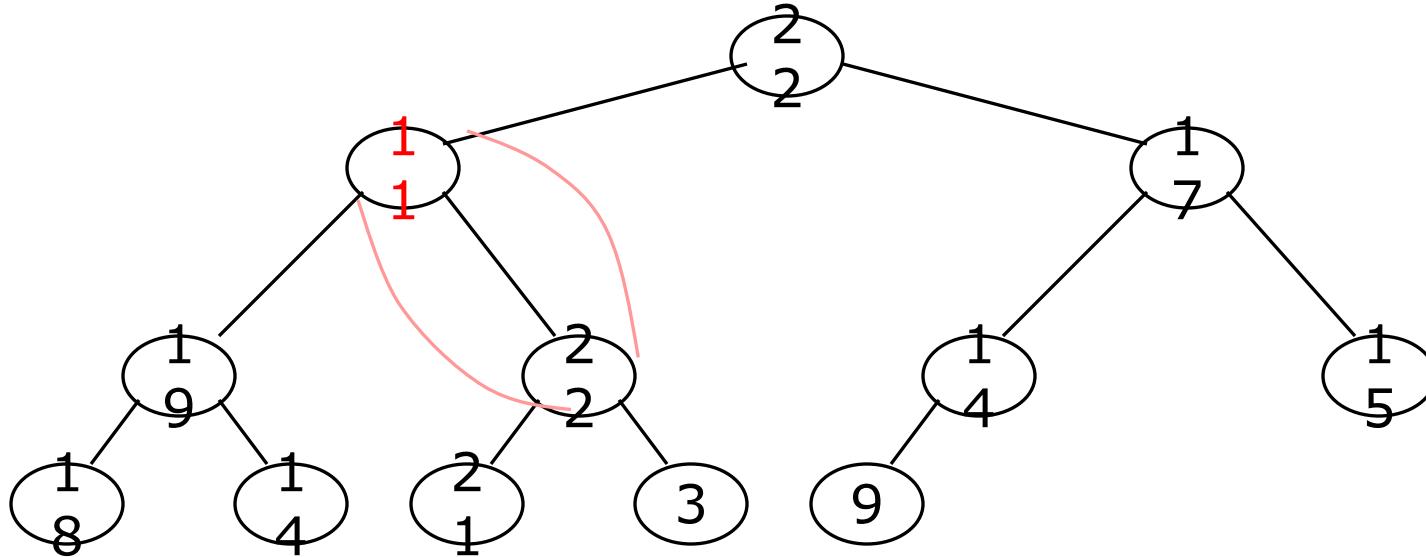
# Penghapusan (Re-Heapfy)



- Reheapfy dilakukan dengan memilih children dengan nilai terbesar yaitu 22, salah satu children saja nantinya strukturnya yang akan berubah (**lack of Heap Property**)

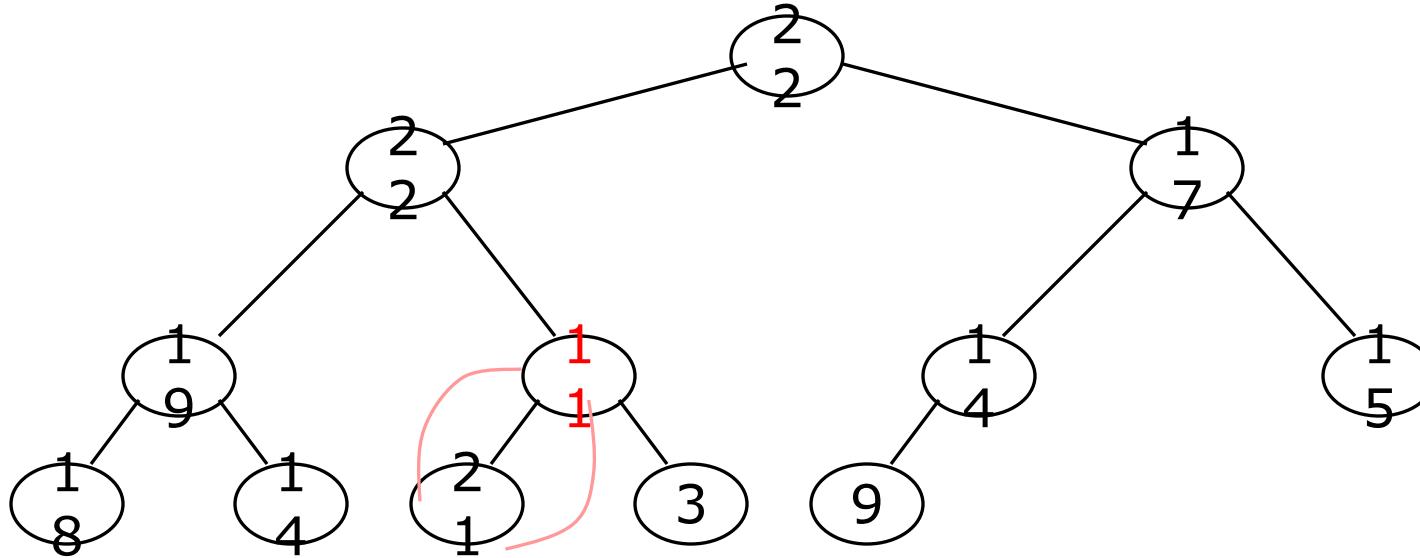


# Penghapusan (Re-Heapfy)



- Reheapfy dilakukan kembali karena di node tersebut unordered. dengan memilih children dengan nilai terbesar yaitu 22, maka strukturnya berganti

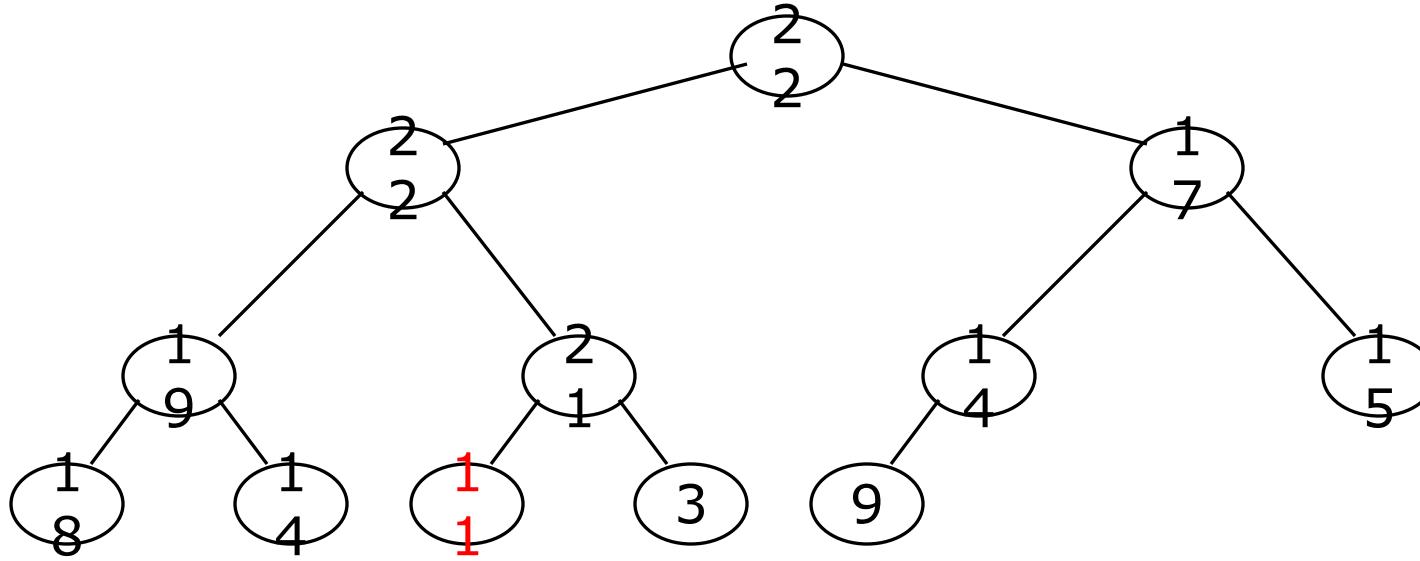
# Penghapusan (Re-Heapfy)



- Reheapfy dilakukan dengan memilih children dengan nilai terbesar yaitu 21, dan proses Heapfy selesai



# Penghapusan (Re-Heapfy)

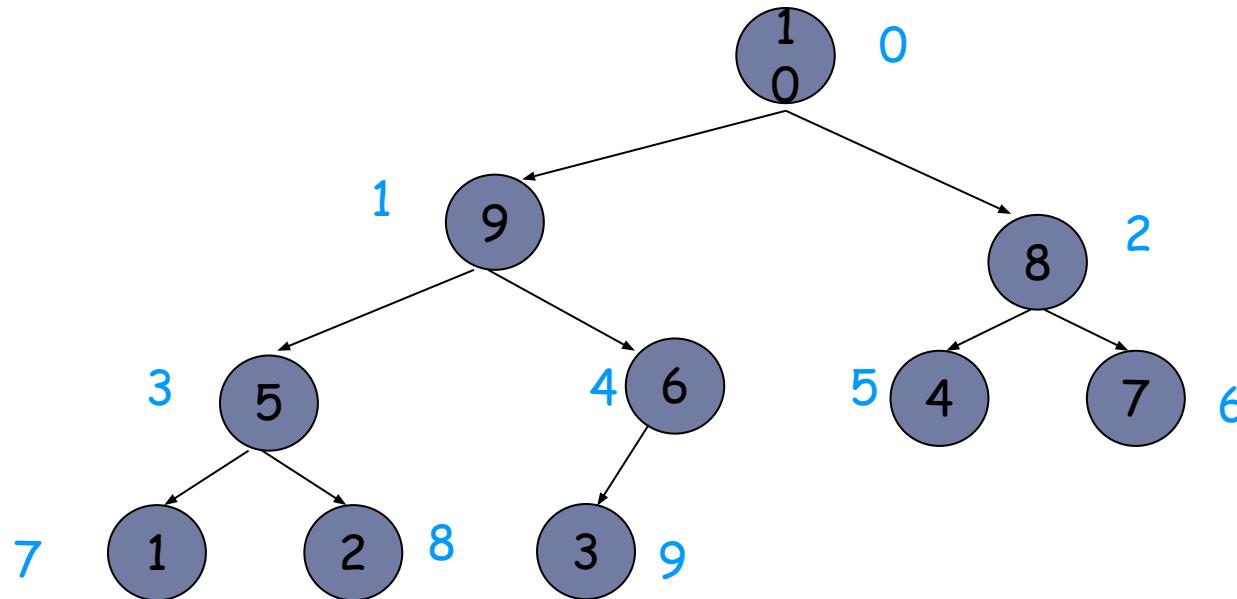


- Metode penghapusan di HeapTree

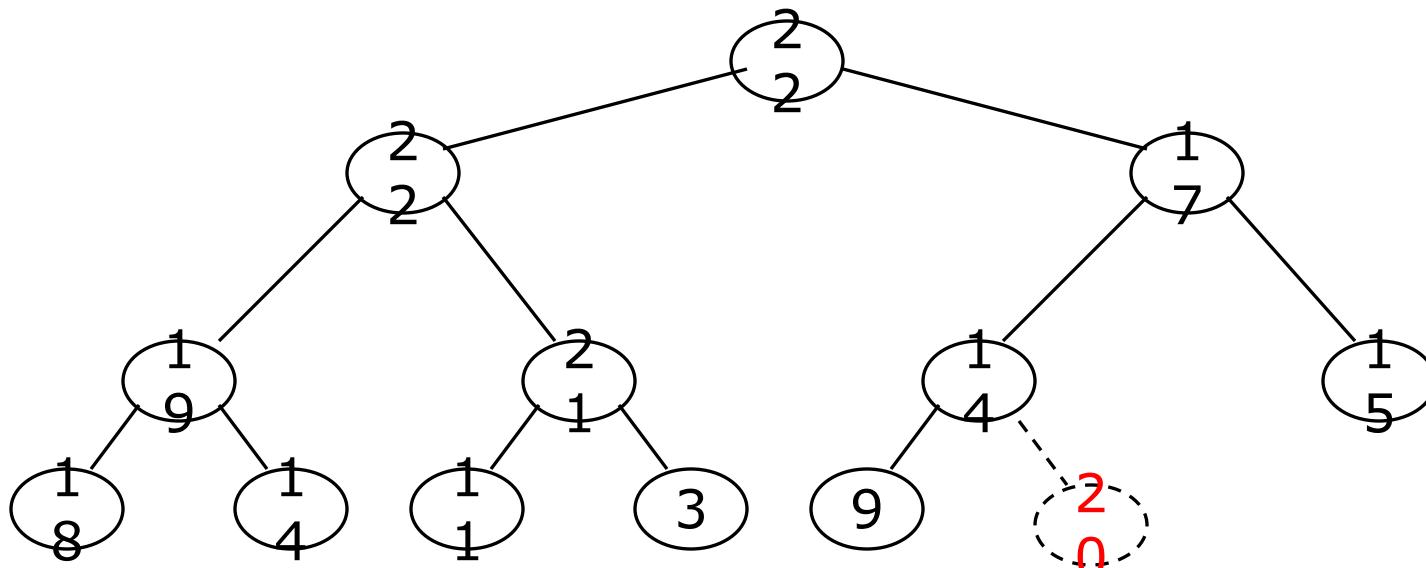


# Latihan

Lakukan penghapusan HeapTree berikut:



# Penambahan (Insertion)

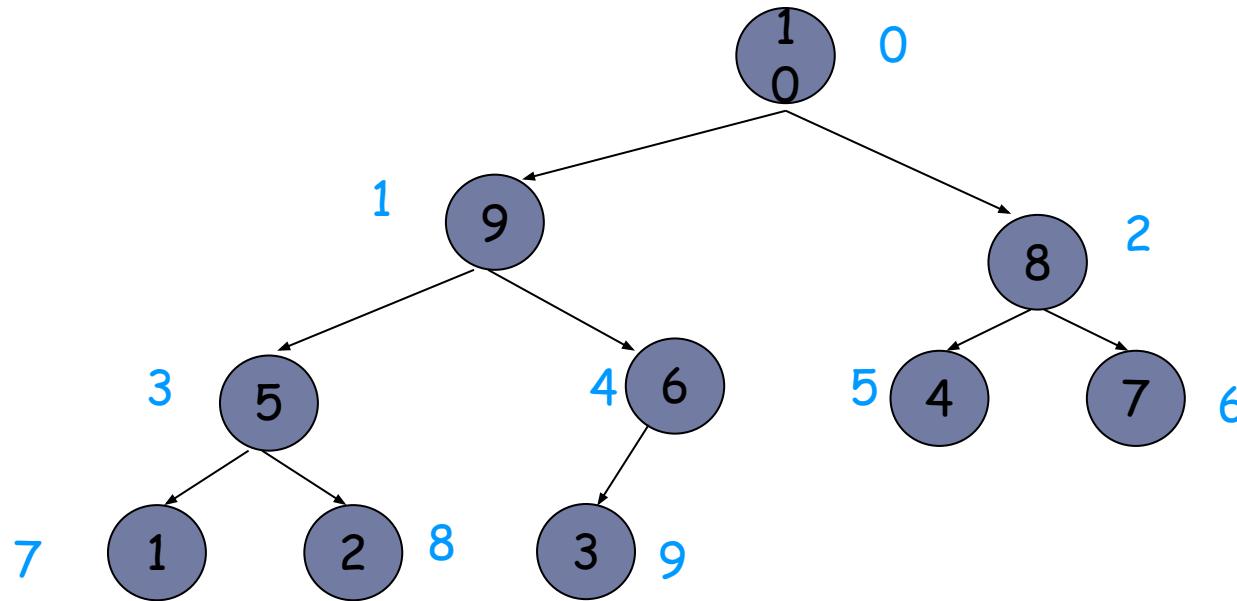


- Ingin disisipkan sebuah node dengan nilai 20
- Maka penyisipan dilakukan pada level terdalam (indeks terbesar)
- Lakukan Re-Heapfy



# Latihan

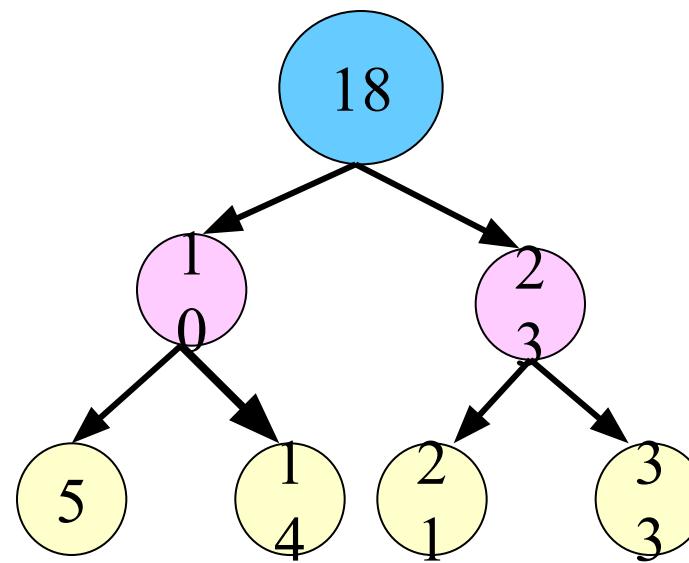
Lakukan Penambahan node pada HeapTree berikut,  
dengan nilai node yang ingin di tambahkan adalah 11:



# Binary Search Tree

Binary Search Tree adalah binary tree dimana node-node dari tree berisi nilai yang sudah terurut sebagai berikut :

**Key node subtree kiri < key dari root < key node subtree kanan**



# Binary Search Tree

---

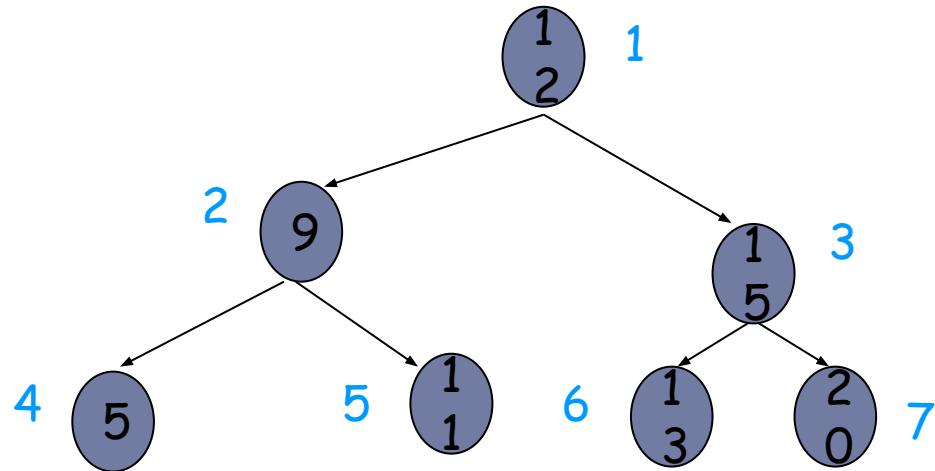
Binary Search Tree adalah binary tree dimana node-node dari tree berisi nilai yang sudah terurut sebagai berikut :

**Key node subtree kiri < key dari root < key node subtree kanan**



# Binary Search Tree Insertion

**Key node subtree kiri < key dari root < key node subtree kanan**



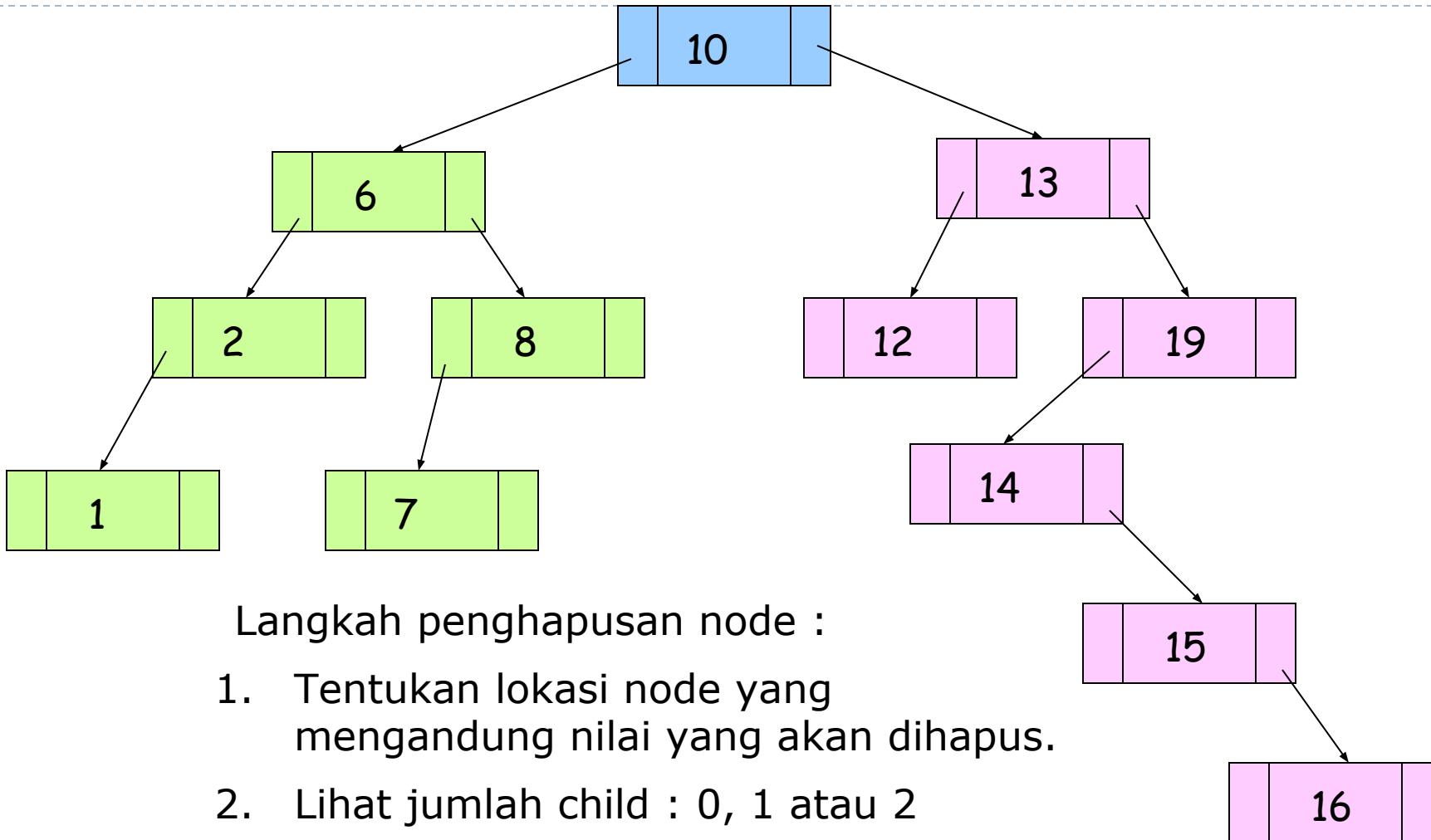
Mulai dari root lakukan perbandingan terhadap root untuk menuju subtree kiri atau subtree kanan

Lakukan insertion untuk nilai berikut ke dalam BST yang masih kosong masukan berikut :

10, 16, 14, 7, 6, 17, 21 !



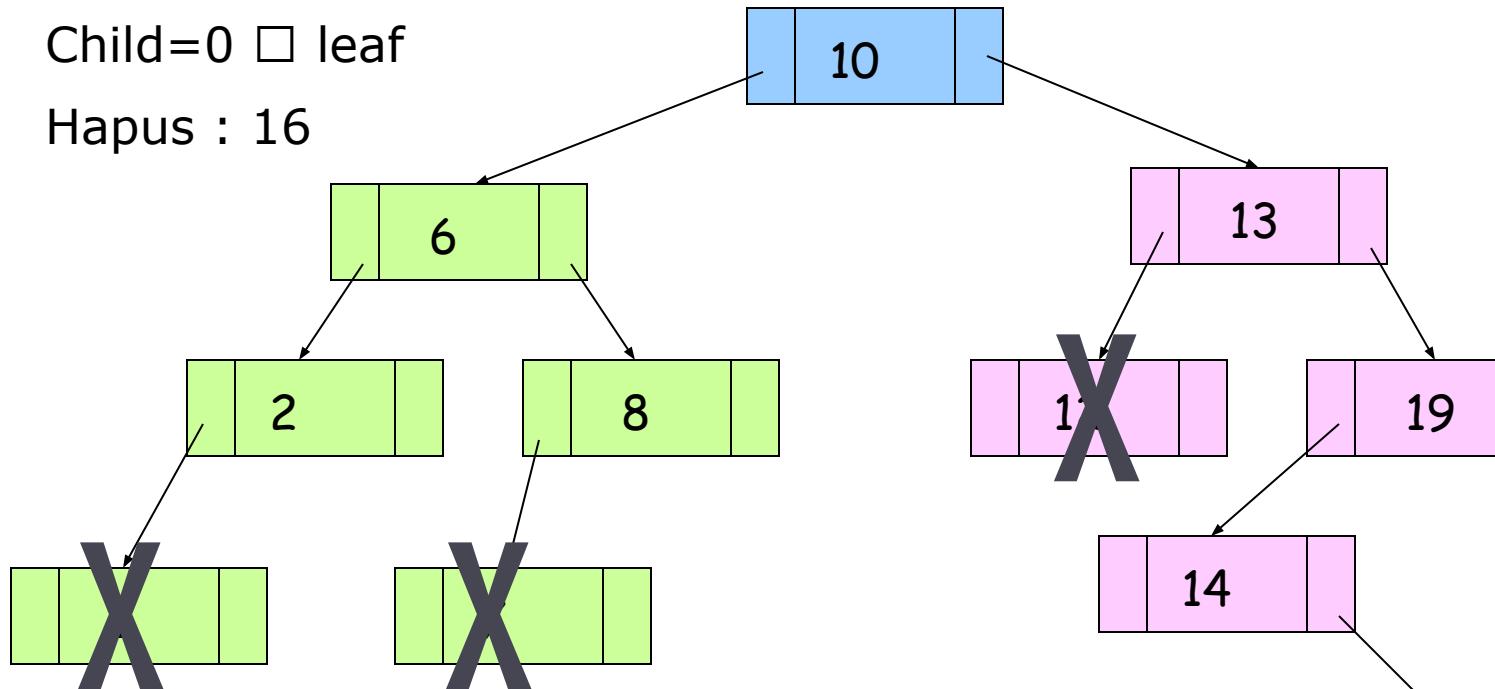
# Binary Search Tree Deletion



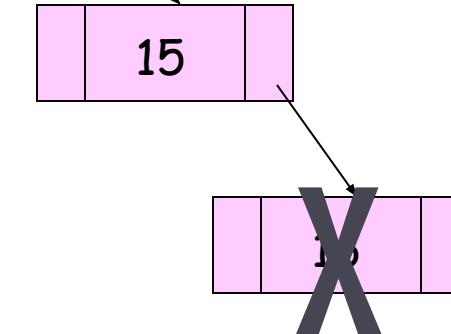
# Binary Search Tree Deletion

Child=0 □ leaf

Hapus : 16

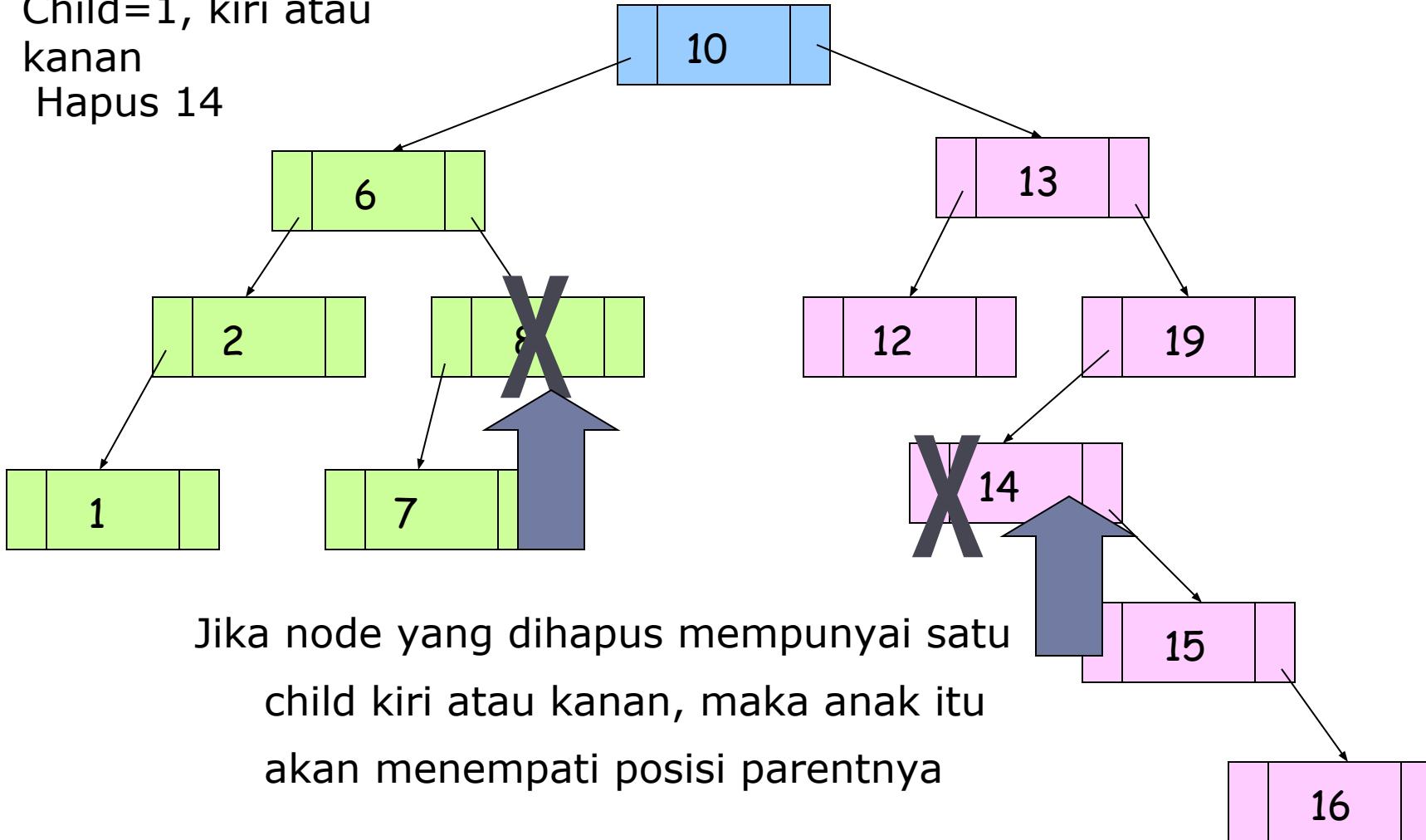


Jika node yang dihapus adalah leaf,  
maka langsung hapus

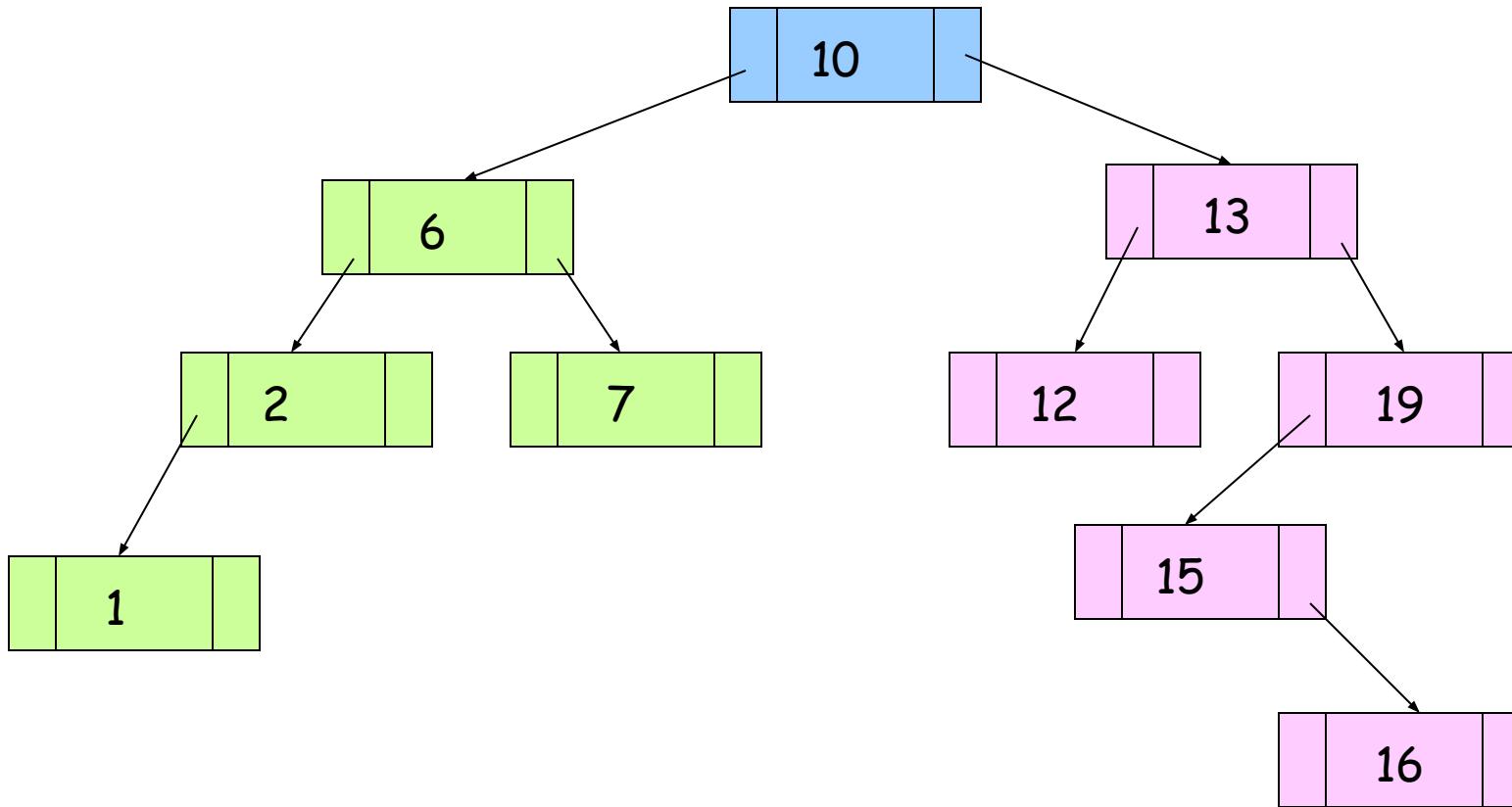


# Binary Search Tree Deletion

Child=1, kiri atau  
kanan  
Hapus 14



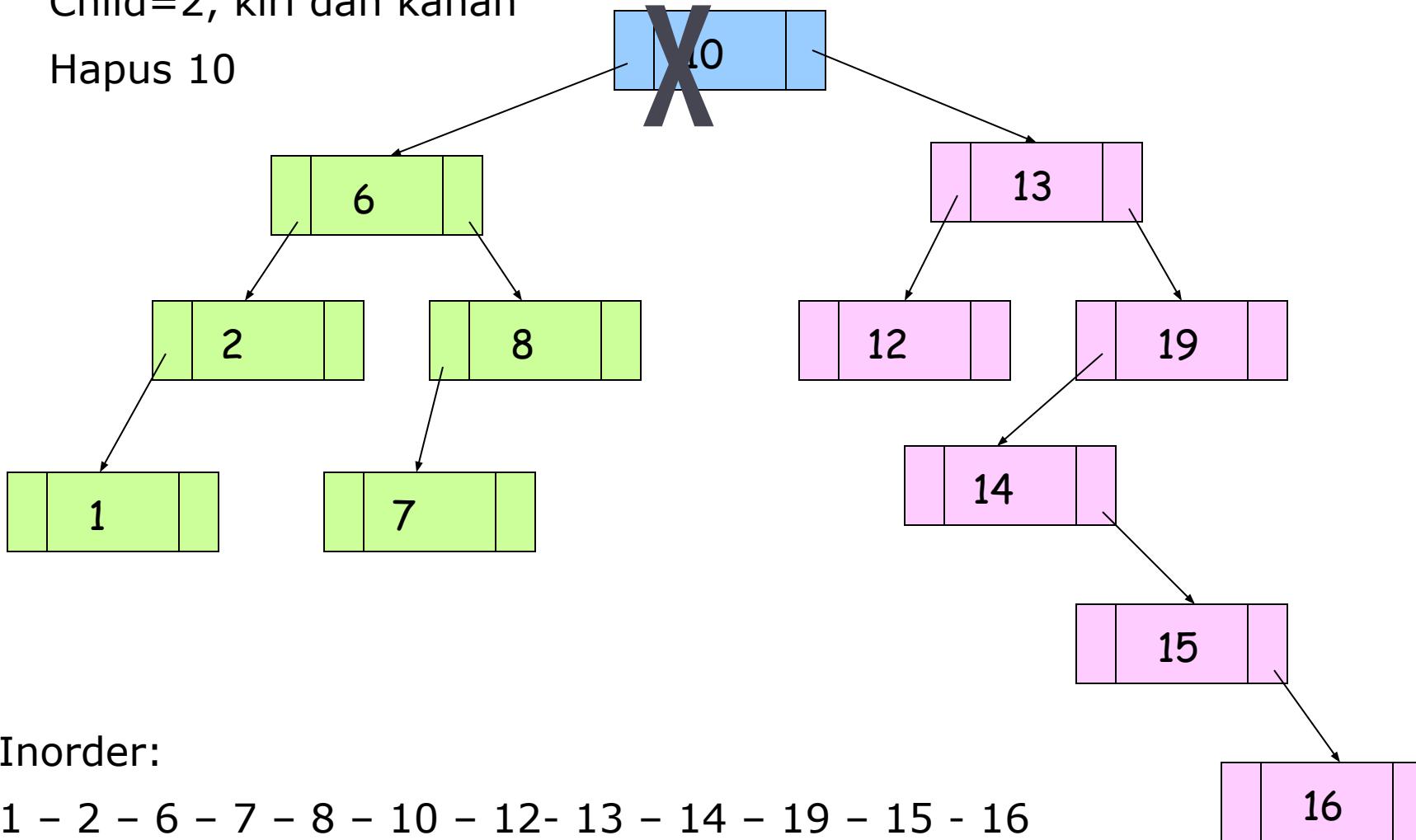
# Binary Search Tree Deletion



# Binary Search Tree Deletion

Child=2, kiri dan kanan

Hapus 10



Inorder:

1 - 2 - 6 - 7 - 8 - 10 - 12 - 13 - 14 - 15 - 16



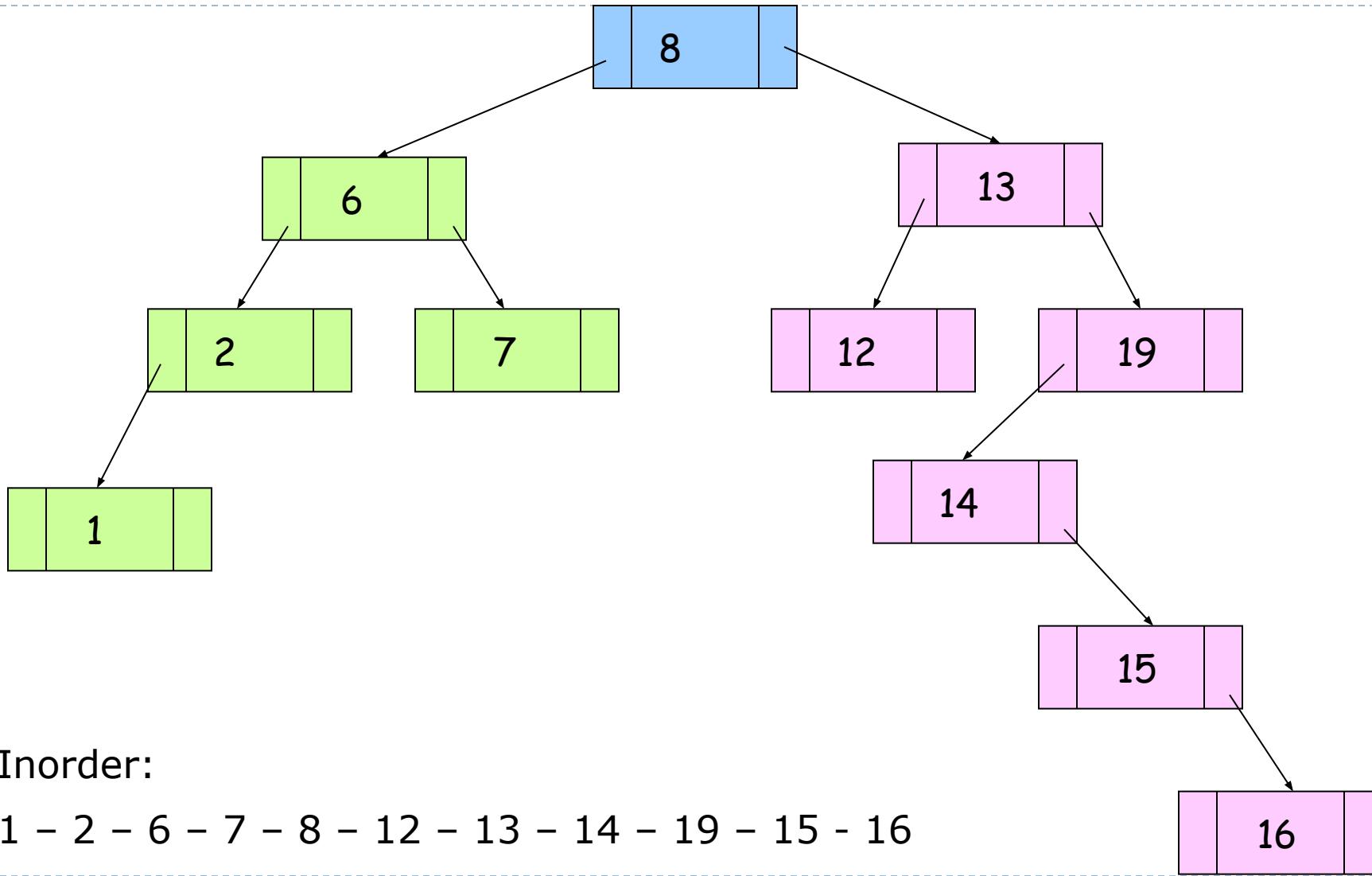
# Binary Search Tree Deletion

---

- Jika node yang dihapus punya dua child, gantikan posisinya dengan node terbesar yang ada pada subtree kiri, atau node terkecil pada subtree kanan.
- Bagaimana mencari elemen terbesar dan terkecil pada subtree?
  - Telusuri root pada subtree kiri dan ambil child di kanan sampai ditemukannya elemen terbesar
  - Telusuri root pada subtree kanan dan ambil child di kiri sampai ditemukannya elemen terkecil

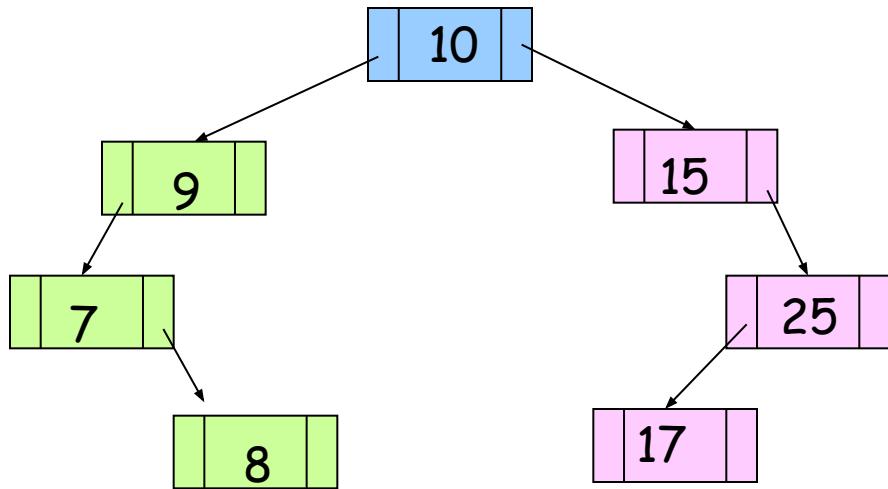


# Binary Search Tree Deletion



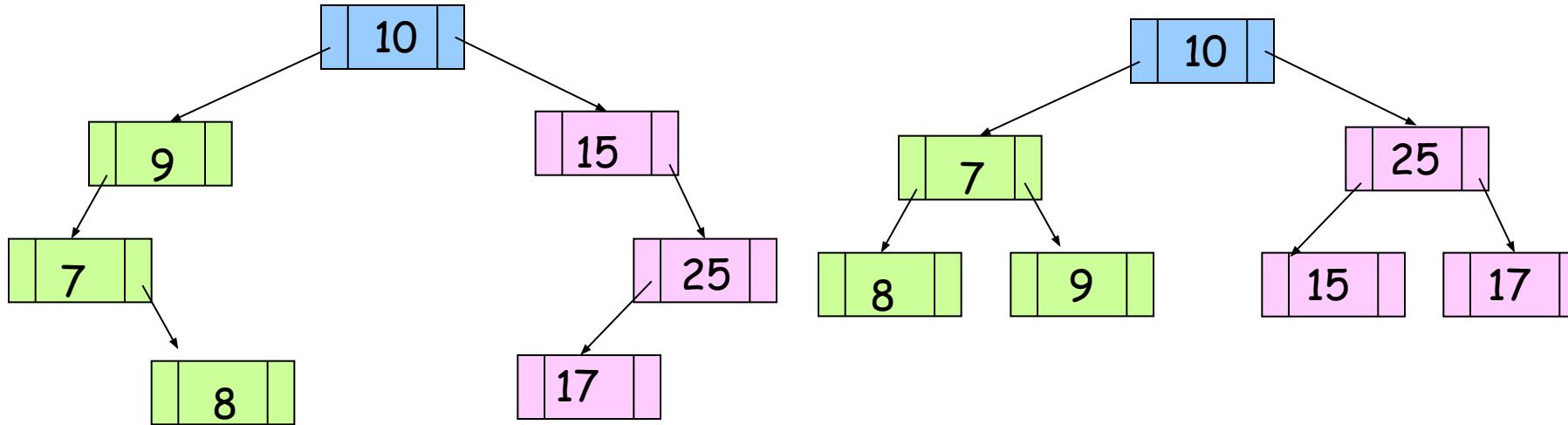
# Latihan

Bentuklah Binary Tree dengan Input : 10, 15, 25, 17, 9, 7, 8



# Kekurangan

Input : 10, 15, 25, 17, 9, 7, 8



- Bentuk binary tree yang paling ideal adalah perfect binary tree□  
minimal complete binary tree
- Pada BST terdapat kemungkinan ada node yang terletak jauh dari root
- Waktu yang dibutuhkan untuk searching menjadi lebih banyak



---

Wassalamu'alaikum ... Terima Kasih

---



# **Struktur Data Berhierarki Bag II**

## **KOM20H**

**Oleh: Tim Pengajar Struktur Data**

# Penghapusan (delete) BST

---

Operasi delete akan dikategorisasikan kepada 3 kasus berdasar lokasi node yang dihapus:

- Untuk node tanpa cabang (leaf) maka node tersebut langsung dihapus
- Jika node memiliki 1 anak maka jadikan anak yang satu-satunya tersebut sebagai pengganti node yang dihapus
- Jika node memiliki 2 anak maka sebagai pengganti node yang dihapus pilih satu dari dua opsi (predesessor inorder atau sucessor inorder)



# Penghapusan (delete) BST

---

## (Pred & Suks) Inorder:

- Disebut sebagai **Predecessor Inorder** yaitu node yang dikunjungi tepat sebelum node terkait dengan penelusuran inorder
- Disebut sebagai **Succesor Inorder** yaitu node yang dikunjungi tepat setelah node terkait dengan penelusuran inorder



# Penghapusan (delete) BST

(Pred & Sucs)

**Inorder:**

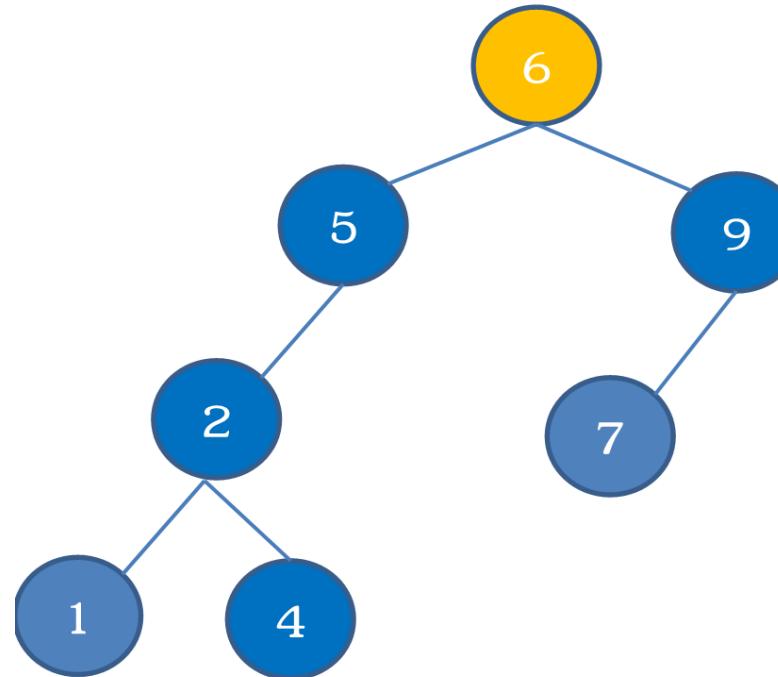
Perhatikan BST berikut:

Carilah Inorder dari BST tersebut

Tentukan:

PredInorder(6)

SuscInorder(6)



# Penghapusan (delete) BST

(Pred & Sucs)

**Inorder:**

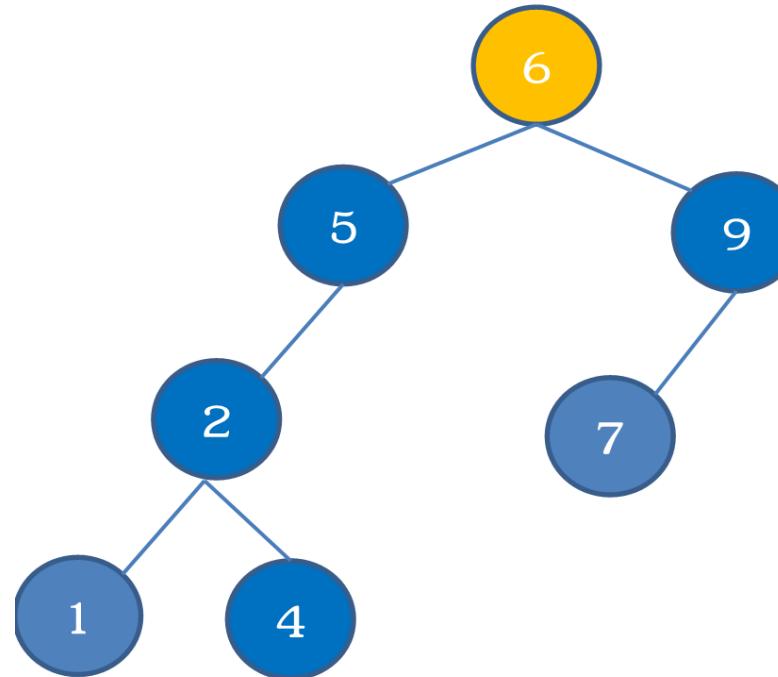
Perhatikan BST berikut:

Carilah Inorder dari BST tersebut  
1-2-4-5-6-7-9

Tentukan:

$\text{PredInorder}(6) = 1-2-4-\underline{5}-6-7-9$

$\text{SuscInorder}(6) = 1-2-4-5-6-\underline{7}-9$



# Syarat Operasi BST

---

- Apapun operasinya pada BST, syarat mutlaknya adalah sifat BST harus tetap terjaga
- Jika sifat BST menjadi gugur maka operasi yang dilakukan salah
- Kompleksitas operasi

Penelusuran, Insert, Edit  $\square$   $O(\lg n)$



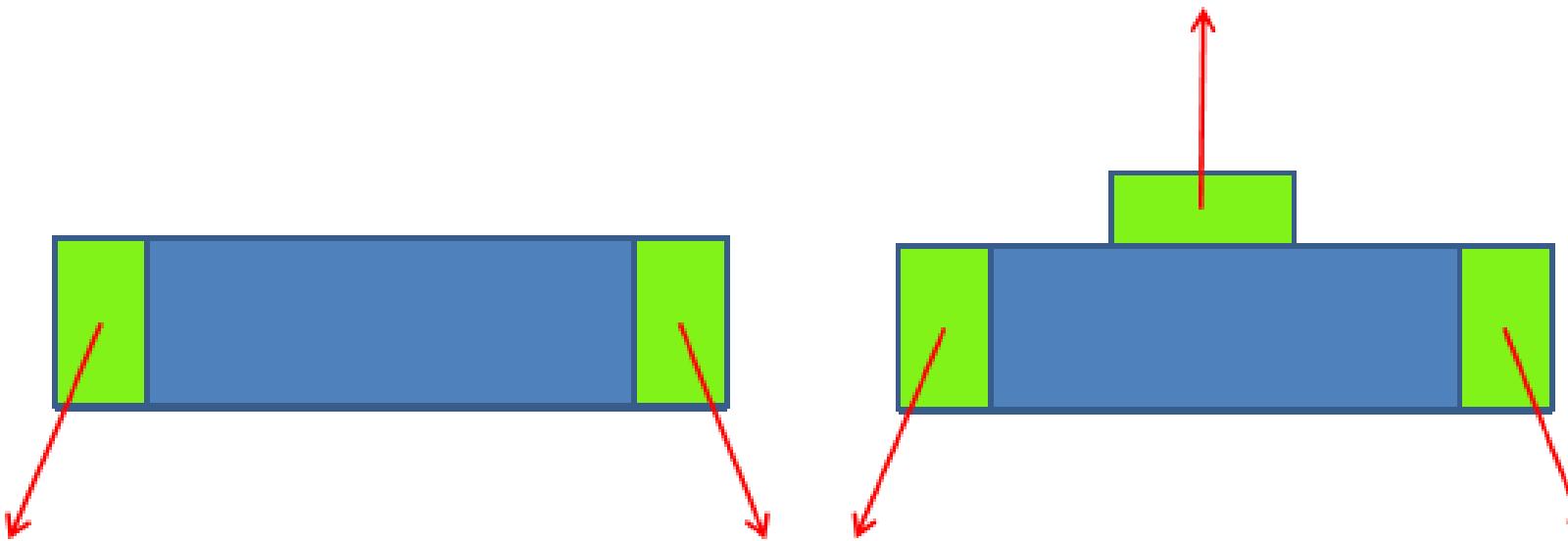
---

## **II. Implementasi BST**



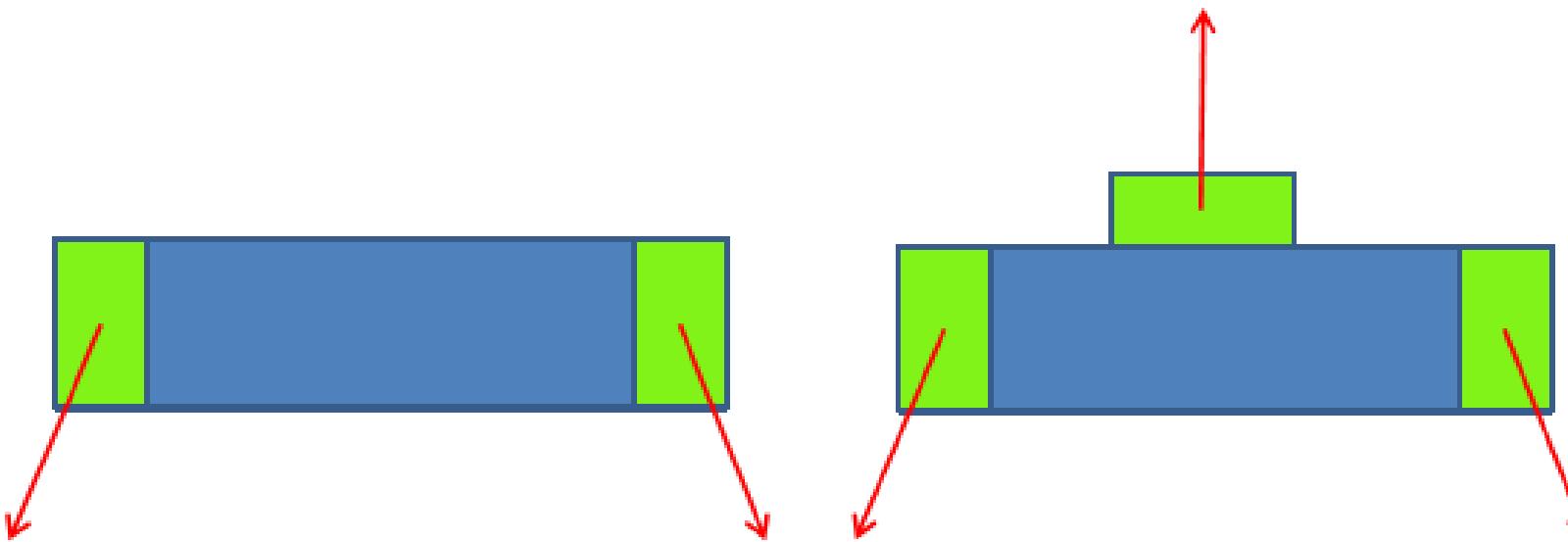
# BST Implementation

- Bentuk BST tidak selalu CBT sehingga representasinya tidak dapat digunakan array
- Salah satu solusinya gunakan node dengan pointer (bisa dua ataupun tiga)



# BST Implementation

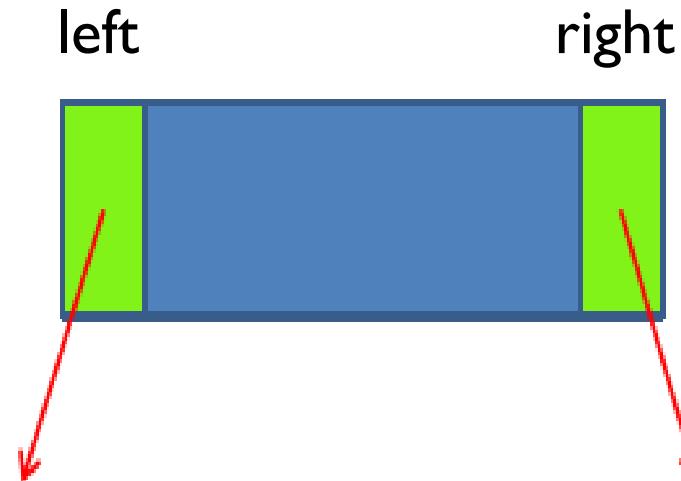
- ☐ Jika digunakan node dengan 2 pointer (left & right)
- ☐ Jika digunakan node dengan 3 pointer (left, right, & parent)



# BST Implementation

- Buatlah deklarasi node dalam tipe struct yang memiliki 2 pointer untuk cabang kanan dan cabang kiri:

```
struct BstNode
{
    BstNode* left;
    BstNode* right;
    int data;
};
```

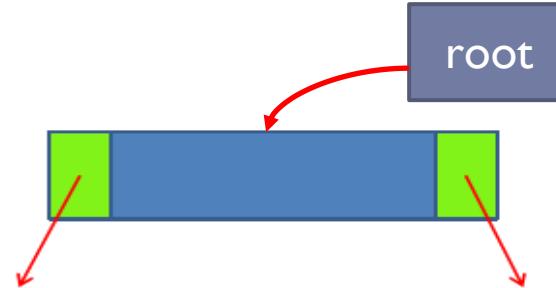


# BST Implementation

- Buatlah deklarasi node dalam tipe struct yang memiliki 2 pointer untuk cabang kanan dan cabang kiri:

```
struct BstNode
{
    BstNode* left;
    BstNode* right;
    int data;
} BstNode;

int main () {
BstNode* root; //pointer to root node
}
```

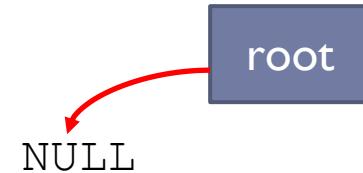


# BST Implementation

- Buatlah deklarasi node dalam tipe struct yang memiliki 2 pointer untuk cabang kanan dan cabang kiri:

```
struct BstNode
{
    BstNode* left;
    BstNode* right;
    int data;
} BstNode;

int main () {
    BstNode* root = NULL //pointer to root node diset sebagai Empty tree
}
```



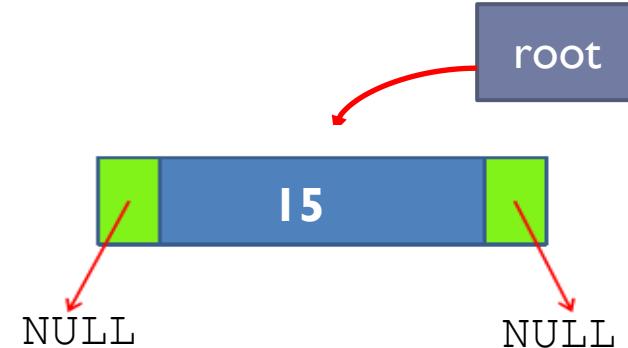
# BST Implementation

- Buatlah deklarasi node dalam tipe struct yang memiliki 2 pointer untuk cabang kanan dan cabang kiri:

```
struct BstNode
{
    BstNode* left;
    BstNode* right;
    int data;
}BstNode;

void insert(BstNode* root, int data);

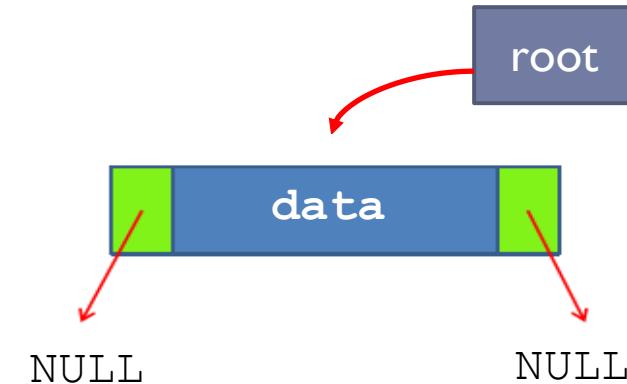
int main () {
    BstNode* root; //pointer to root node
    root = NULL; //buat empty node
    insert(root, 15)
}
```



# BST Implementation

## □ Insertion

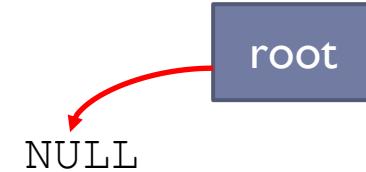
```
void BstNode* BuatNodeBaru(int data) {  
    BstNode* node_baru = new BstNode();  
    node_baru -> data = data;  
    node_baru -> left = NULL;  
    node_baru -> right = NULL;  
    return node_baru  
}
```



# BST Implementation

## □ Insertion

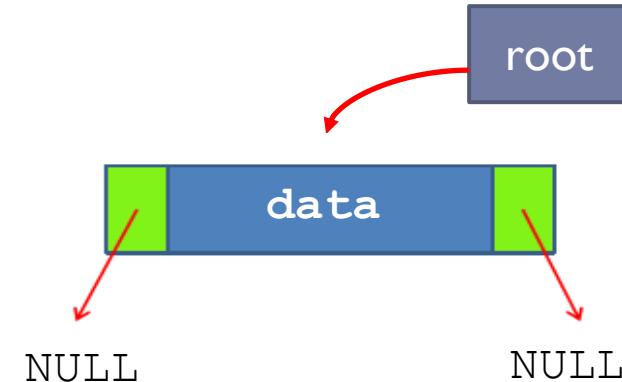
```
void insert(BstNode *root, int data) {  
    if (root==NULL) {  
        root=BuatNodeBaru(data);  
    }  
}
```



# BST Implementation

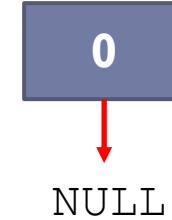
## □ Insertion

```
void insert(BstNode *root, int data) {  
    if (root==NULL) {  
        root=BuatNodeBaru(data);  
    }  
    else if(data<= root -> data) {  
        root -> left = insert(root->left,data);  
    }  
    else {  
        root -> right = insert(root->right,data);  
    }  
}
```



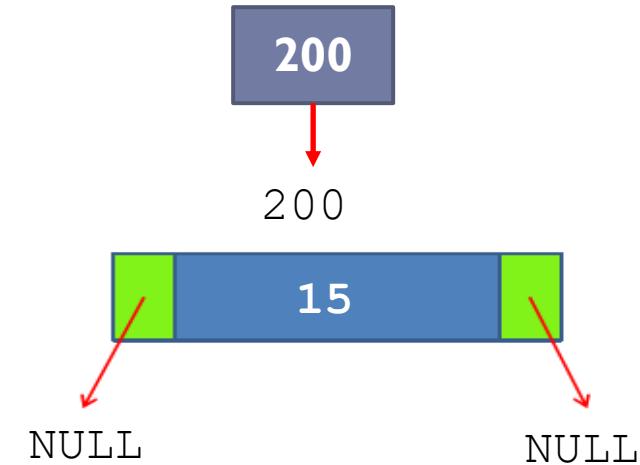
# BST Implementation

```
Int main (){  
BstNode* root; //pointer to root node  
root = NULL; //buat empty node  
}
```



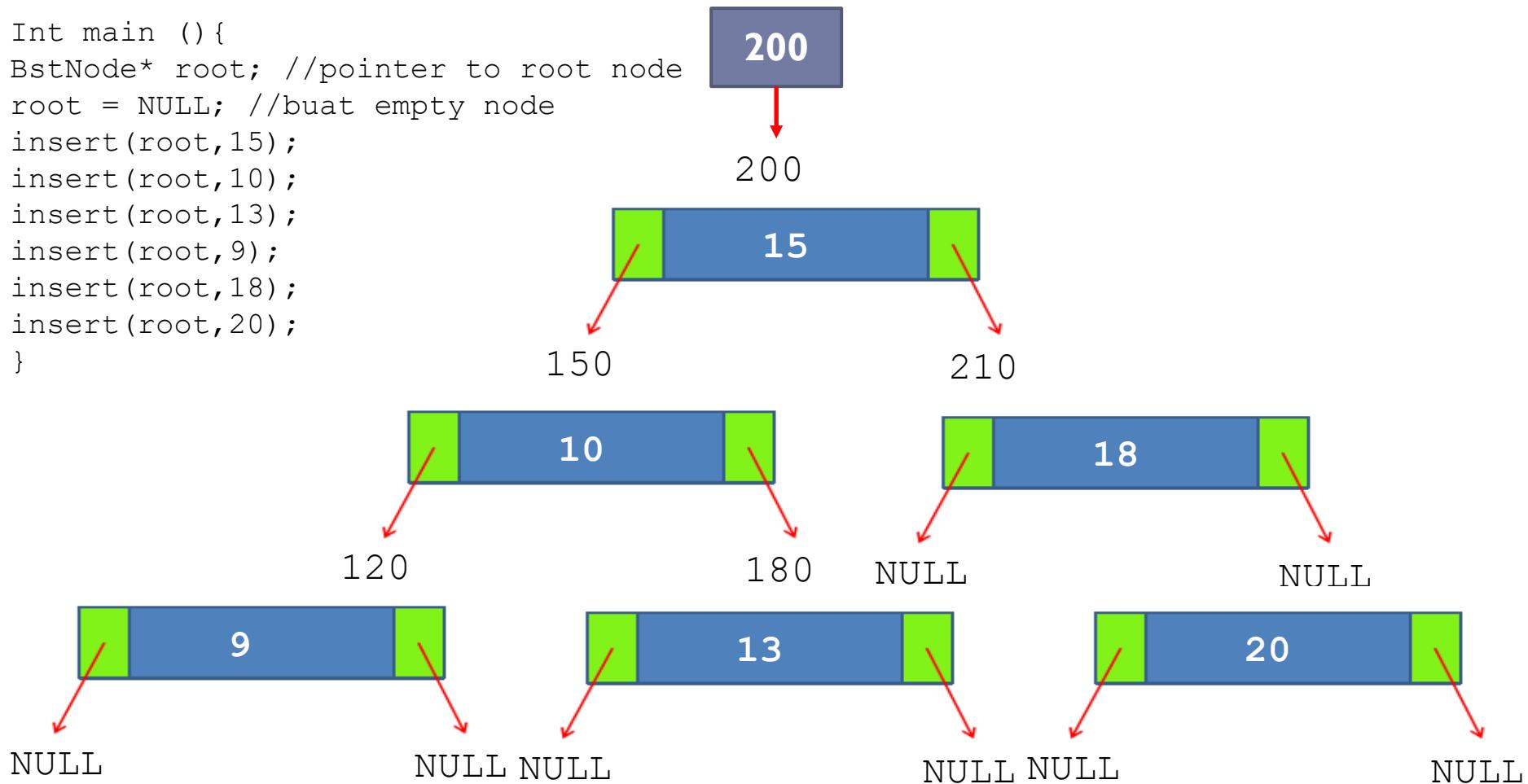
# BST Implementation

```
Int main (){  
BstNode* root; //pointer to root node  
root = NULL; //buat empty node  
insert(root,15);  
}
```



# BST Implementation

```
Int main (){  
BstNode* root; //pointer to root node  
root = NULL; //buat empty node  
insert(root,15);  
insert(root,10);  
insert(root,13);  
insert(root,9);  
insert(root,18);  
insert(root,20);  
}
```



## Pemanfaatan BST

---

- Teknik penyimpanan indeks data bagi repository (pada teknologi awal)
- Mungkin akan bertanya2 bahwa sebelumnya pernah mendengar teknik binary search pada array dan apa bedanya dengan BST ini?
  - BST ini pada kasus insert value baru  $O(\lg n)$  pada insertion biasa di array cari posisi mungkin bisa  $O(\lg n)$  namun setelahnya harus dilanjutkan dengan pergeseran nilai2 di kanan



---

# **AVL - Tree**

---



# AVL-Tree

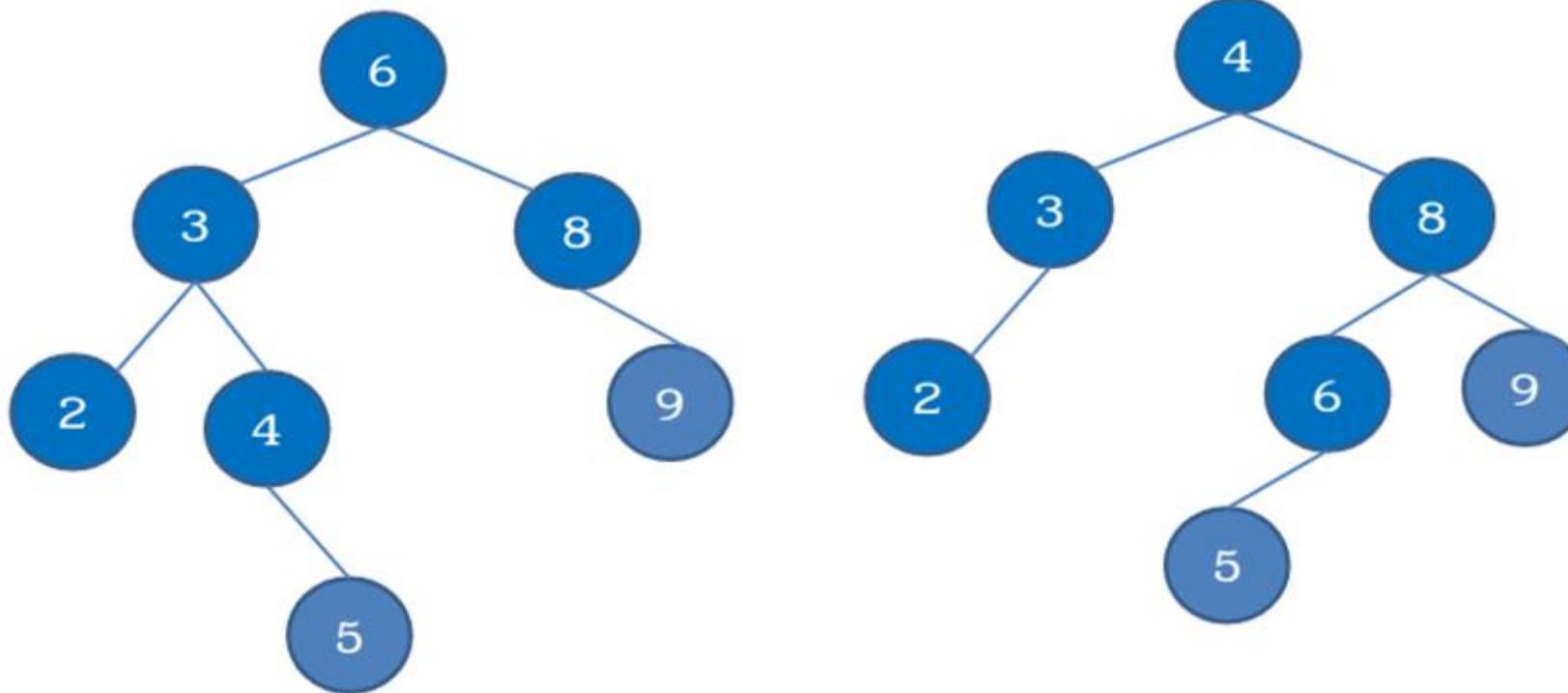
---

- AVL-Tree merupakan salah satu solusi bagi BST untuk mengkondisikan bahwa BST tetap memiliki tree seimbang
- AVL : Adelson – Velskii – Landis
- AVL Tree adalah BST dengan properti keseimbangan antar subtree
- Beda ketinggian untuk setiap subtree kiri dan subtree kanan dari setiap node (pada BST) maksimal adalah 1



# Contoh

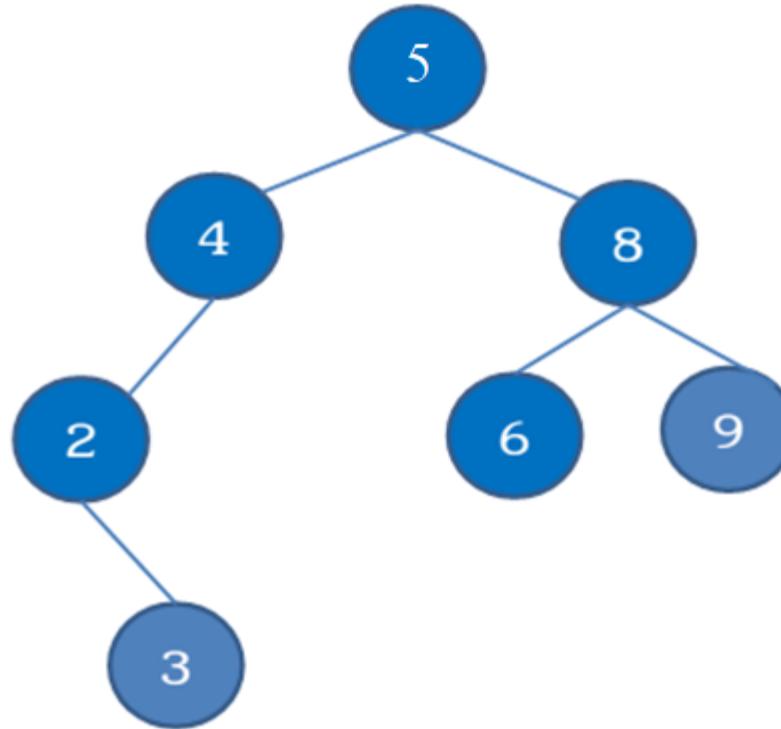
- Apakah BST berikut merupakan AVL?



# Contoh

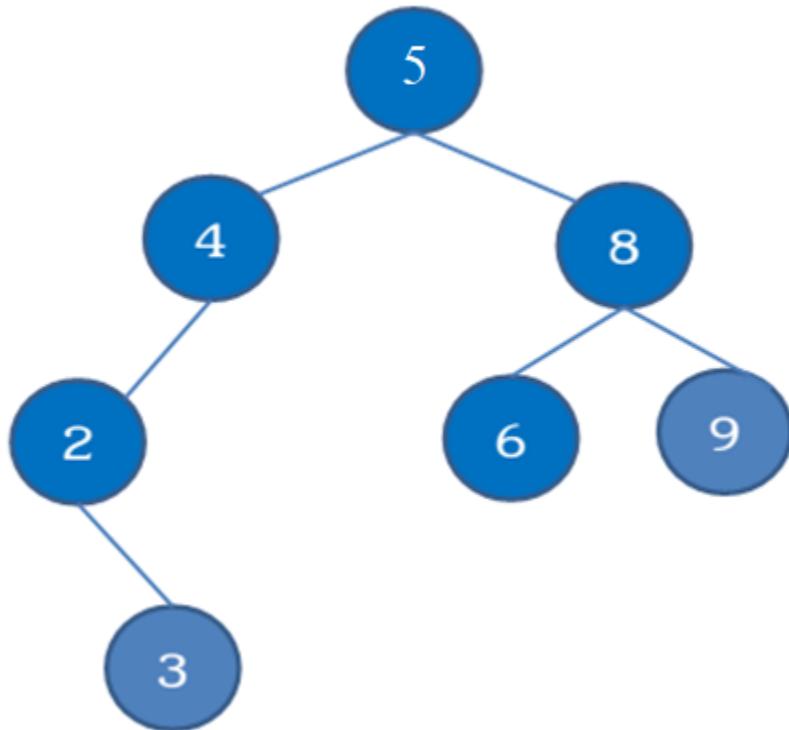
---

Apakah BST berikut AVL?



# Jumlah Node vs Ketinggian

Apakah BST berikut AVL?

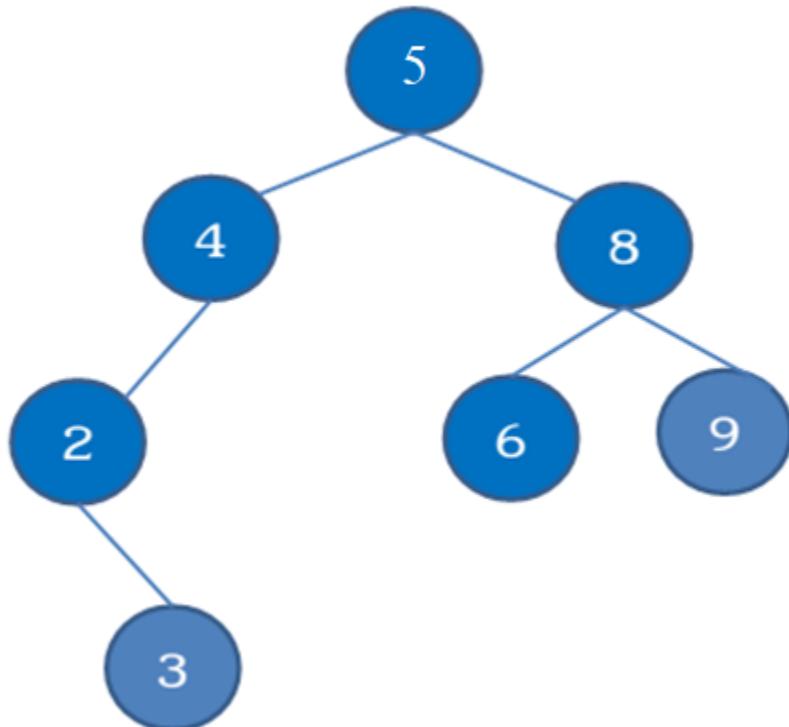


- Kondisi ideal adalah dengan banyak node besar ketinggian tetap kecil, contoh: CBT
- Bentuk AVL tidak harus CBT, lalu berapakah ketinggian maksimal AVL pada sejumlah (sebanyak) node tertentu?



# Jumlah Node vs Ketinggian

Apakah BST berikut AVL?

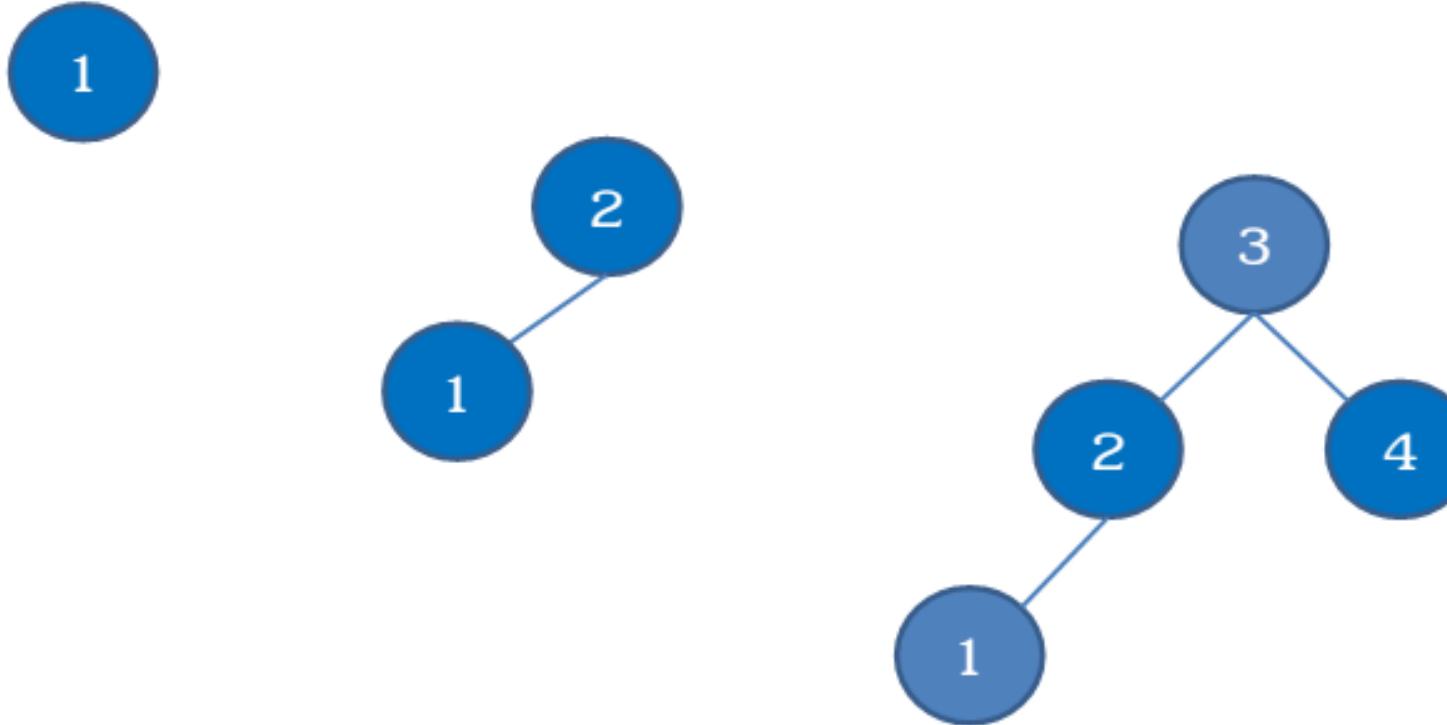


- Apakah kondisi yang tidak ideal pada AVL-Tree tetap dapat menghasilkan kondisi efisien?
- Efisien pada Tree  $\square$  penelusurannya adalah  $O(\log n)$



# Jumlah Node vs Ketinggian

---



Ketinggian  $h=0$  min 1 node ( $N$ )

$h=1$  min  $N=2$ , dan ketika  $h=2$  min 4 node



## Operasi pada AVL-Tree

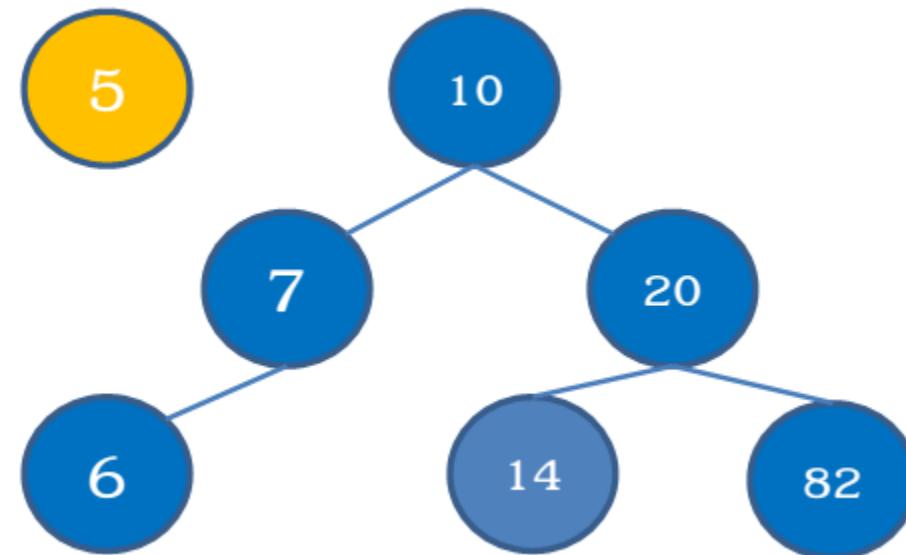
---

- Seperti halnya struktur data lain operasi dasar AVL yang utama meliputi: Search, Insert, Update, dan Delete
- Search pada AVL sama seperti BST
- Berikut akan dipaparkan operasi insert, delete, dan update



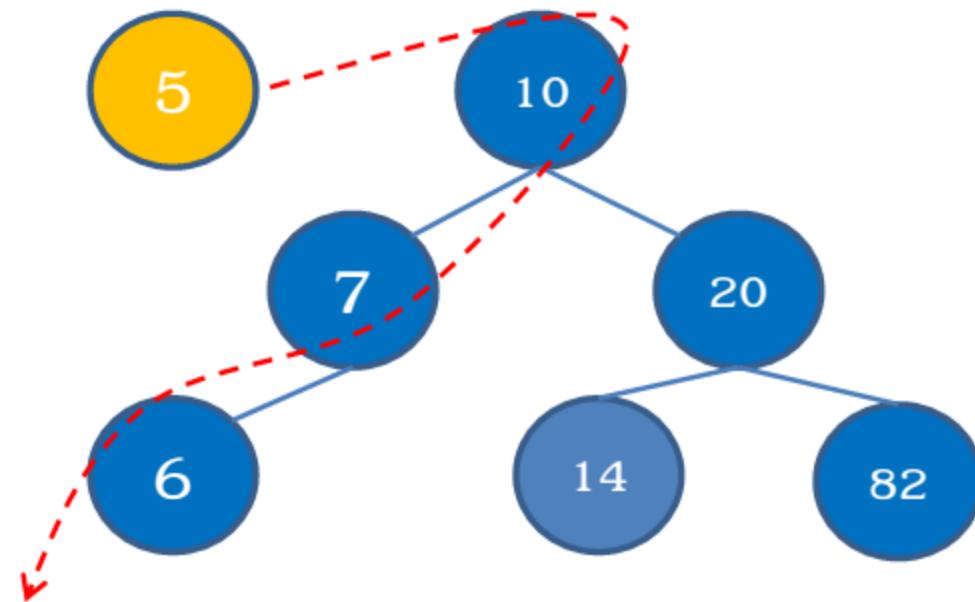
# Insert

- Bagaimana seandainya akan dimasukkan node baru bernilai **5** pada AVL berikut?



# Insert

- Bagaimana seandainya akan dimasukkan node baru bernilai **5** pada AVL berikut?

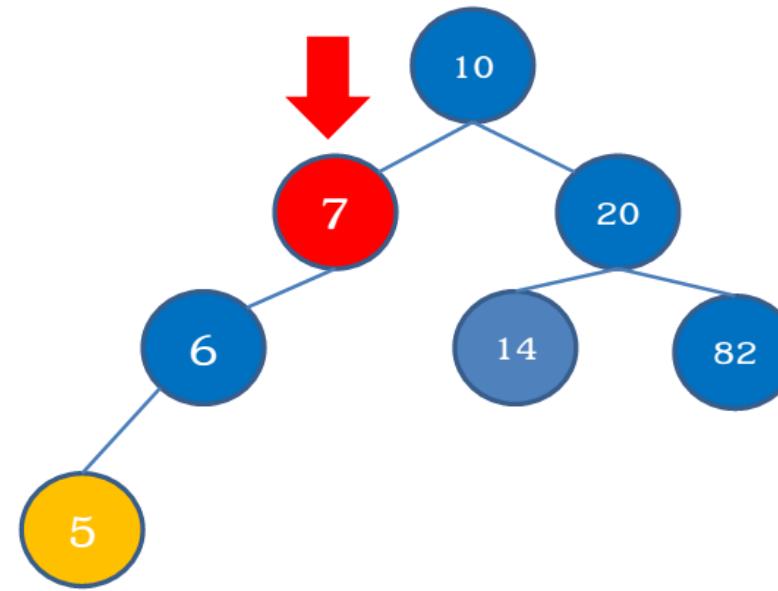


- Apa yang terjadi?



# Insert

- Pada saat node baru bernilai **5** dimasukkan pada AVL tersebut akan mengganggu keseimbangan pada node **7**

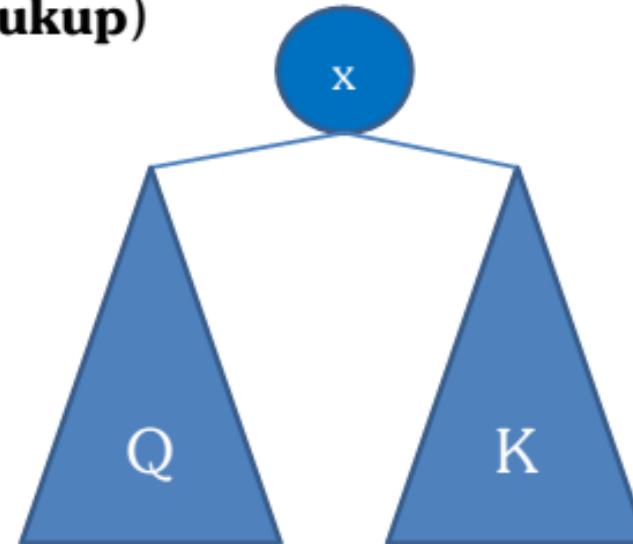


- Karena tidak memenuhi properti AVL (BST menjadi tidak seimbang maka kondisi tersebut harus ditangani/Kondisi kritis!)

# Kondisi Kritis?

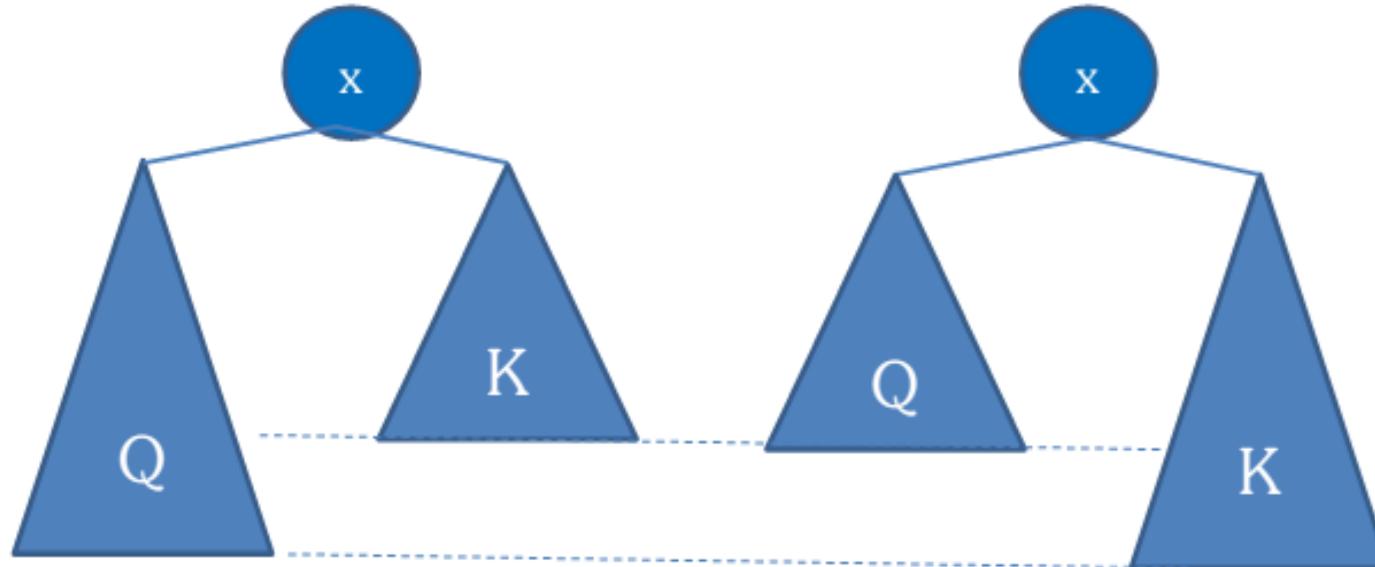
---

- Pada sebuah AVL, andaikan segitiga adalah sebuah subtree dari AVL, maka pada kondisi ini posisi x bukan merupakan kondisi kritis
- Masuknya sebuah node apapun dijamin tidak akan mengganggu keseimbangan di x (**insert BST cukup**)
- Q dan K memiliki tinggi sama



# Kondisi Kritis?

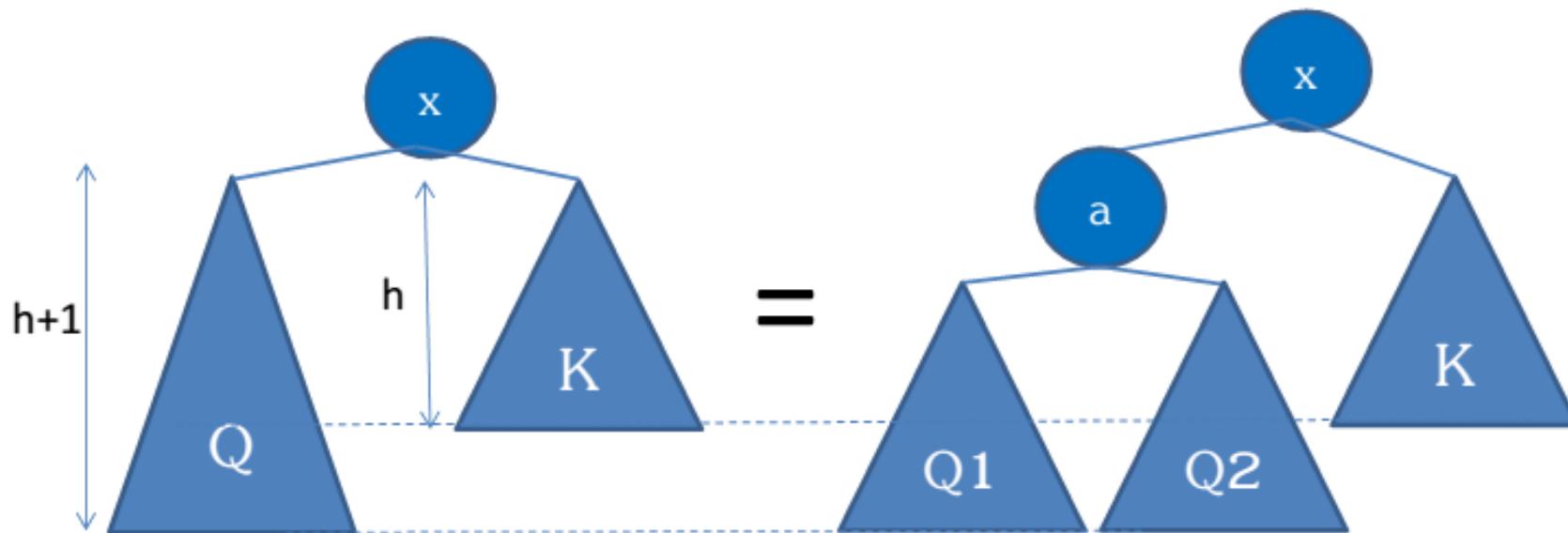
- Berbeda halnya jika Q dan K tidak sama tinggi (beda maksimal 1, kondisi awal masih AVL), maka x adalah posisi dengan kondisi kritis
  - Pada gambar kiri akan beresiko jika node baru masuk ke Q dan menambah tinggi Q
  - Pada gambar kanan akan beresiko jika node baru masuk ke K dan menambah tinggi K



# Kondisi Kritis?

## Kondisi Kritis (Q panjang)

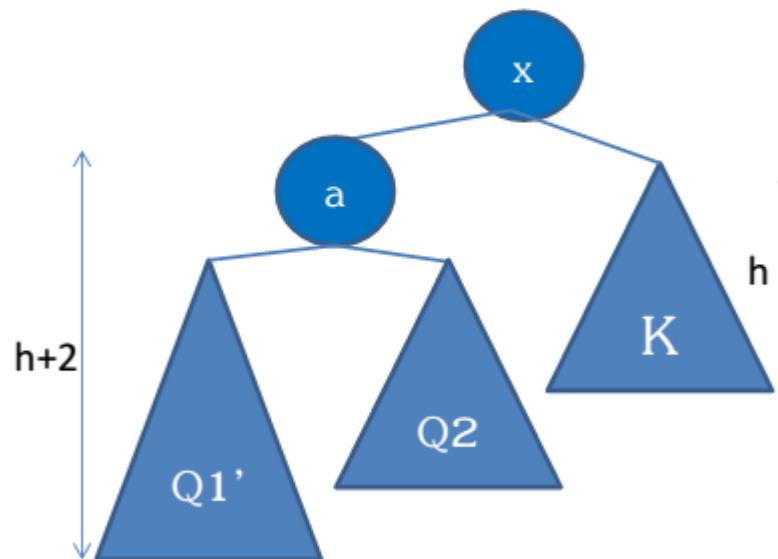
- Fokus pada kondisi kritis pada gambar kiri (slide sblm.)



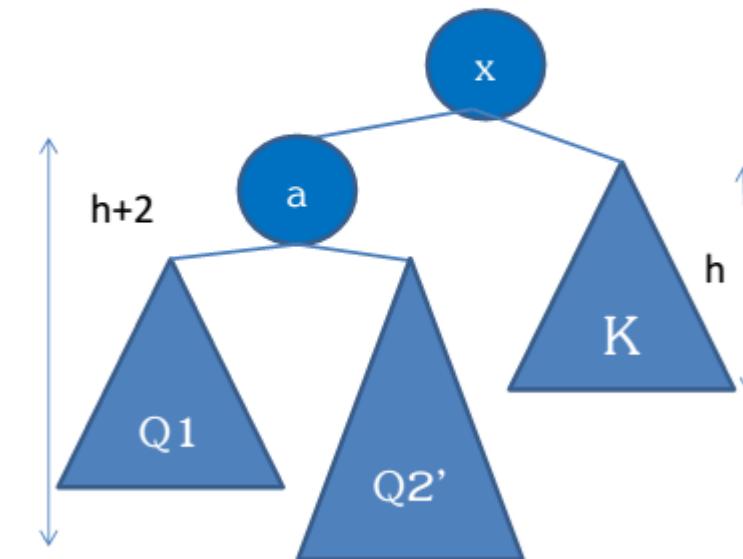
# Kondisi Kritis?

## Kondisi Kritis (Q panjang)

- Keseimbangan terganggu ketika node baru masuk ke kiri x (dalam/luar)



**Kasus 1: Kiri luar**



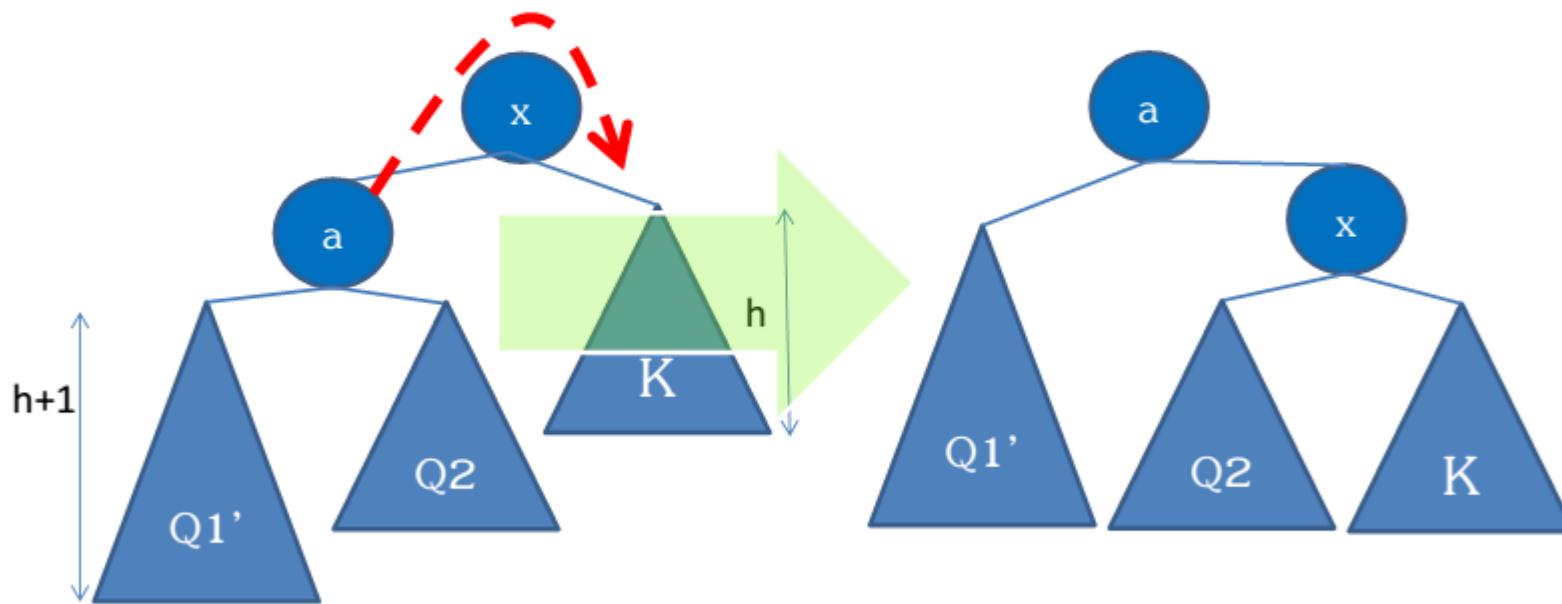
**Kasus 2: Kiri dalam**



# Kondisi Kritis?

## Solusi Kasus 1

- Gunakan single rotasi ke kanan di x



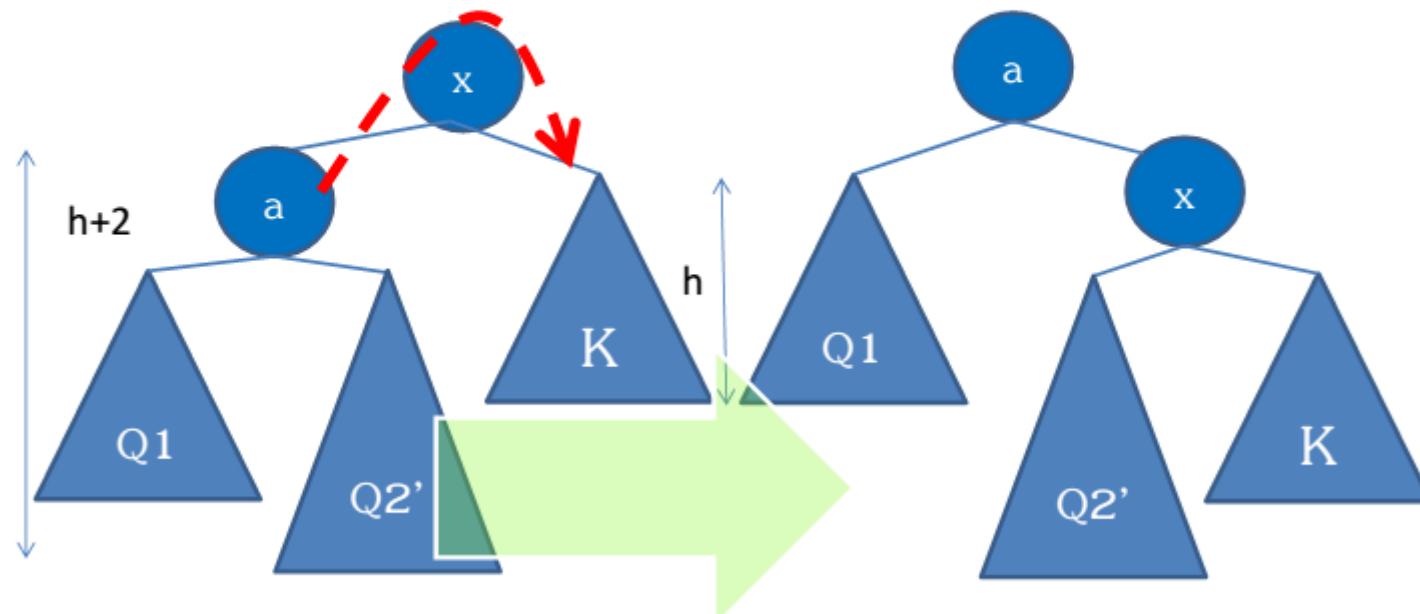
**Kasus 1:** Kiri luar



# Kondisi Kritis?

## Kasus 2

Jika diterapkan single rotasi ke kanan di x



**Kasus 2: Kiri dalam**

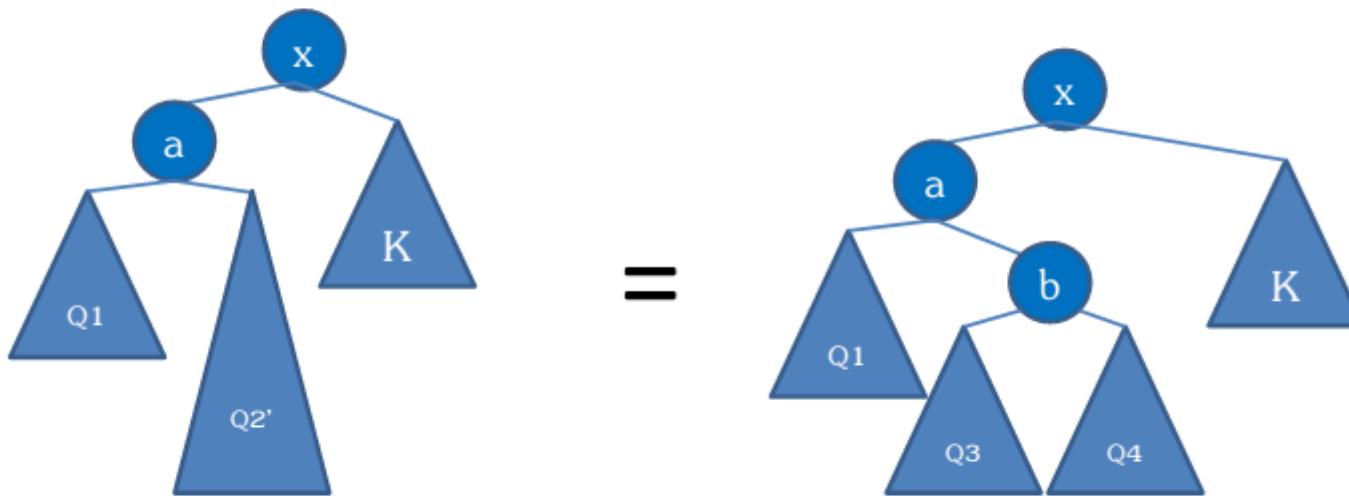
**Tetap tidak seimbang**



# Kondisi Kritis?

## Solusi Kasus 2

- Jika node  $Q_2'$  lebih dijabarkan



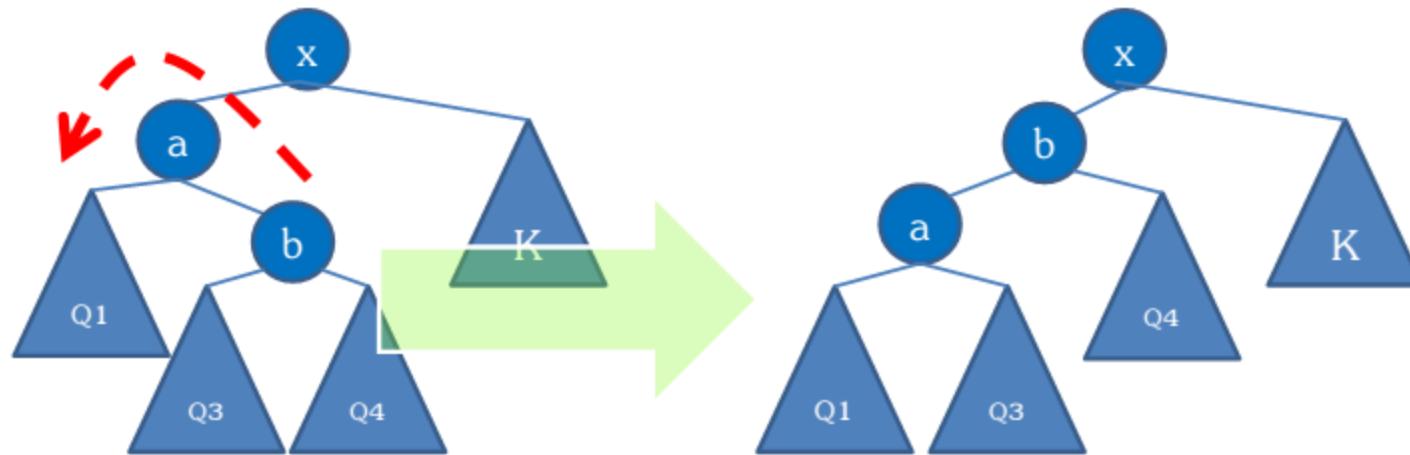
**Kasus 2: Kiri dalam**



# Kondisi Kritis?

## Solusi Kasus 2

- Awali dengan sekali rotasi ke kiri di root subtree kiri x (yaitu a)



**Kasus 2: Kiri dalam**

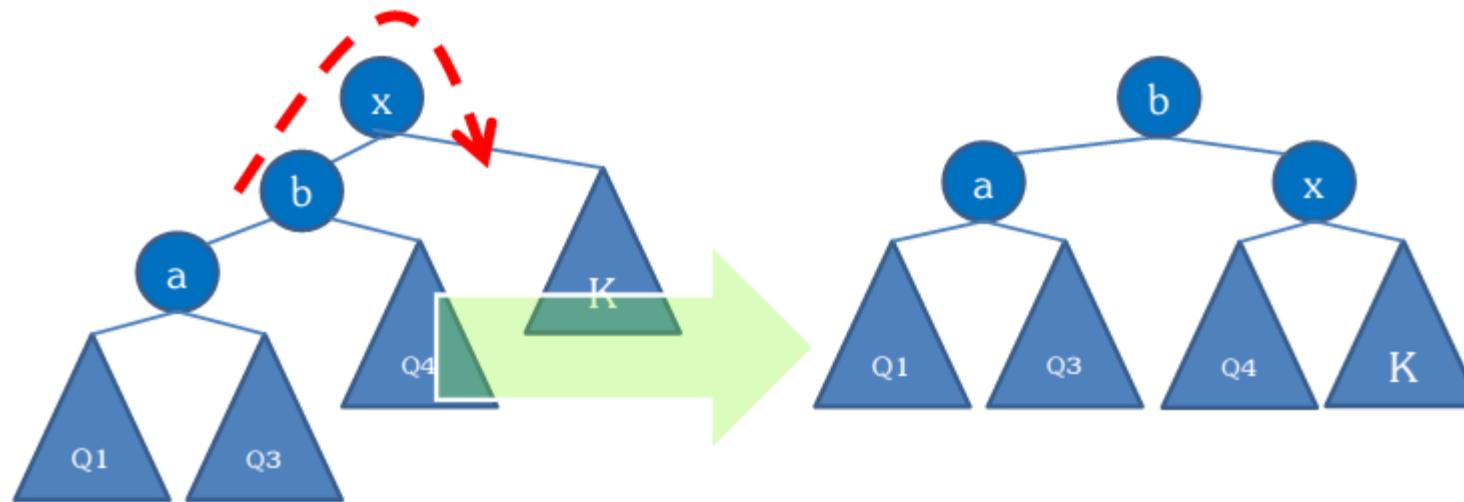
**Menjadi kasus kiri luar**



# Kondisi Kritis?

## Solusi Kasus 2

- Lanjutkan dengan sekali rotasi ke kanan di x



**Sudah seimbang**

**Kasus 2: Kiri dalam**

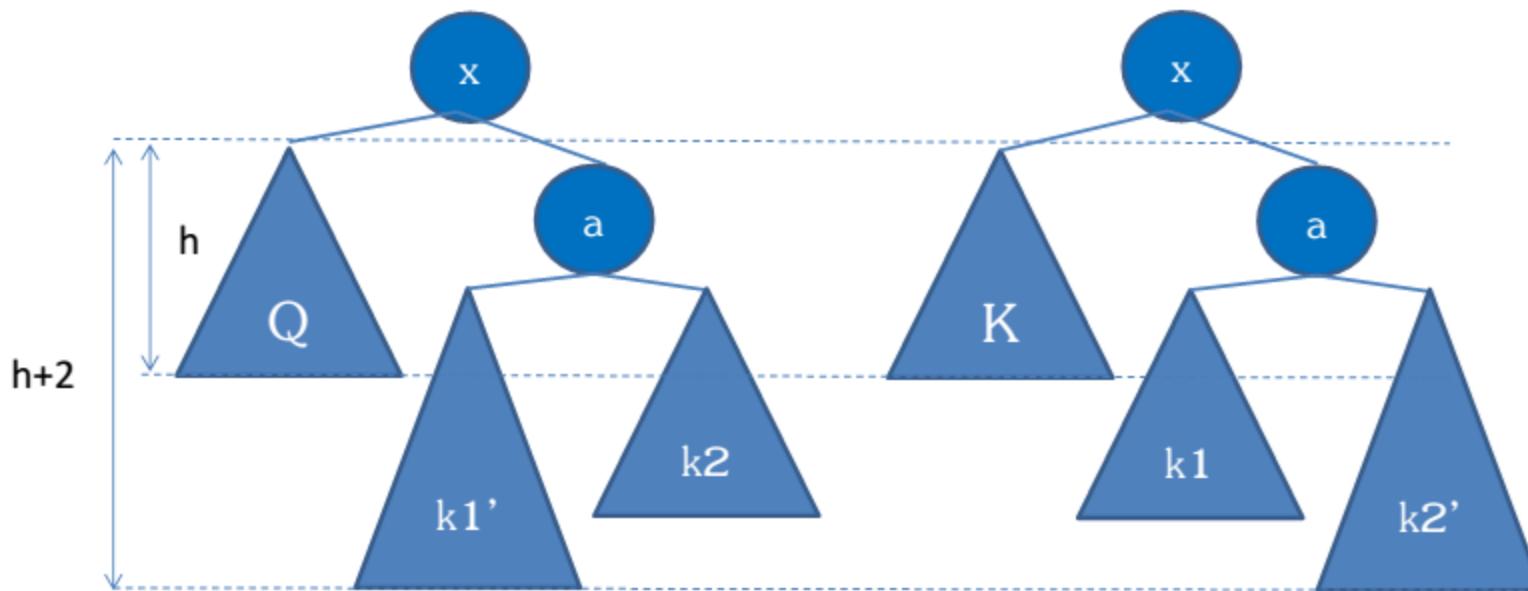
**Hasil rotasi**



# Kondisi Kritis?

## Kasus 3 & 4

- Keseimbangan terganggu ketika node baru masuk ke kanan x (dalam/luar)



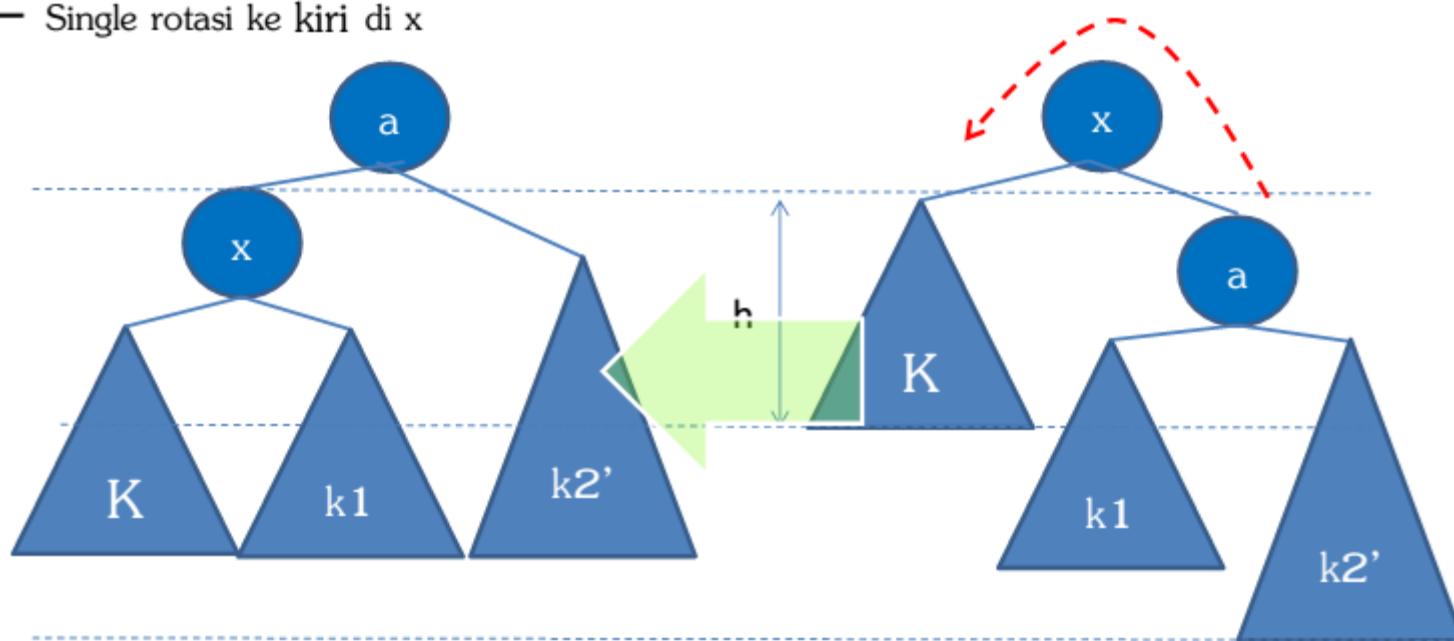
**Kasus 3: Kanan dalam**

**Kasus 4: kanan luar**

# Kondisi Kritis?

## Solusi Kasus 4

- Single rotasi ke kiri di x



**Tersolusikan**

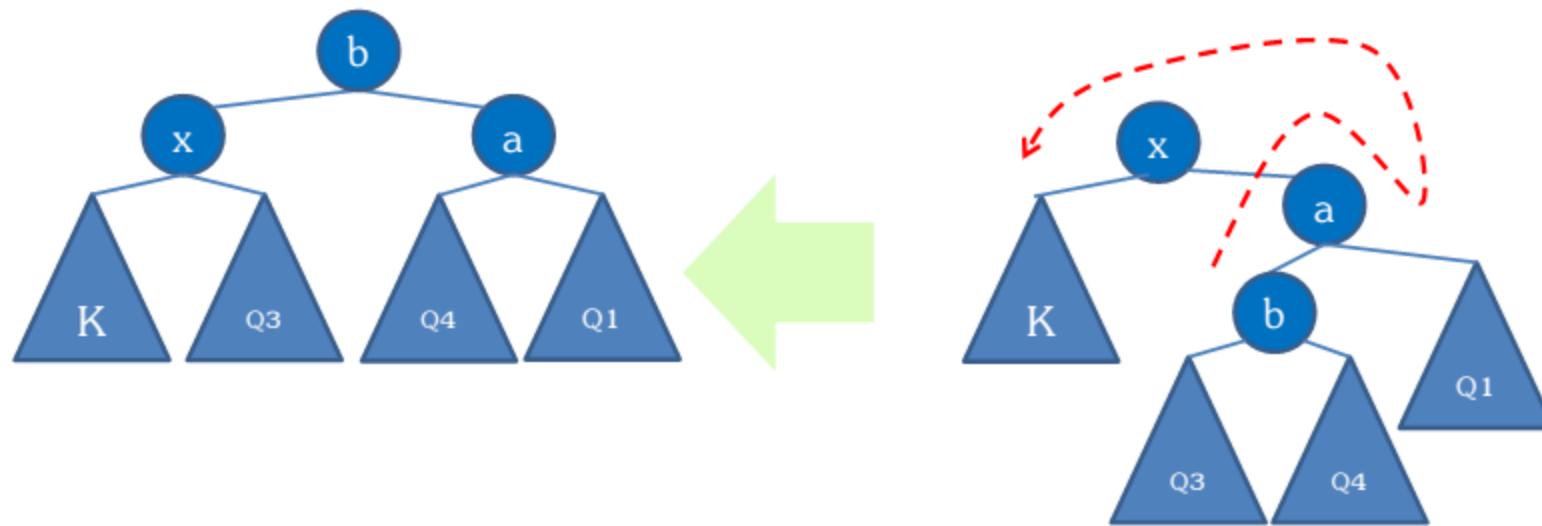
**Kasus 4: kanan luar**



# Kondisi Kritis?

## Solusi Kasus 3

- Gunakan double rotasi



**Kasus 3: Kanan dalam**

# Latihan

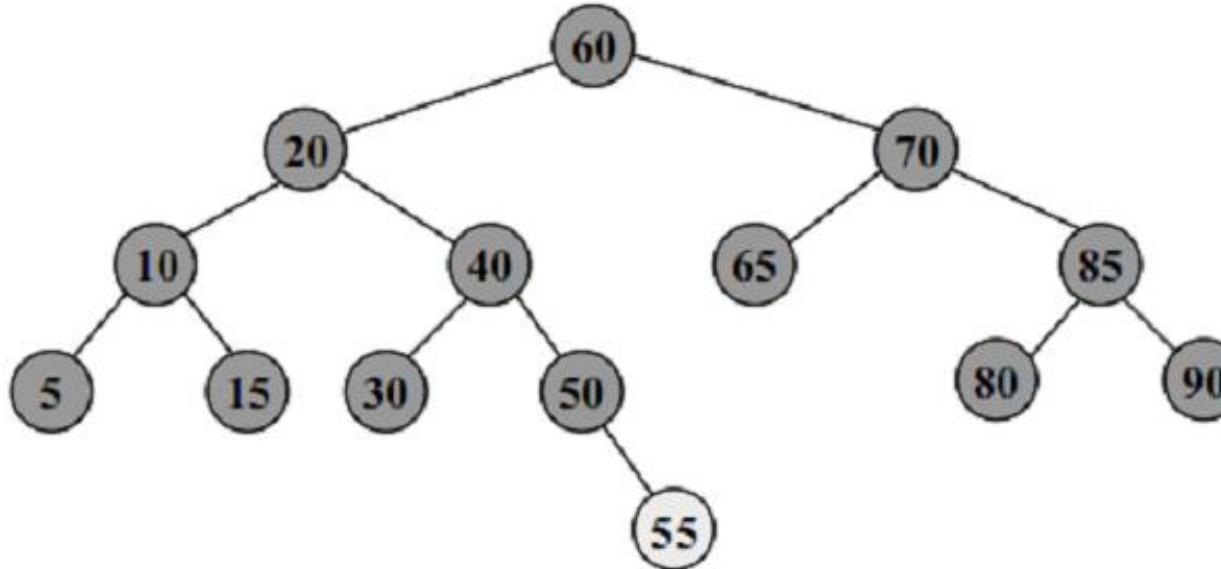
---

- **Masukan node-node berikut ini secara terurut: 10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55**
  - **Dengan metode BST**
  - **Dengan metode AVL-Tree**



# Latihan

- Masukan node-node berikut ini secara terurut: **10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55**
  - Dengan metode BST
  - Dengan metode AVL-Tree



# Operasi Penghapusan AVL-Tree

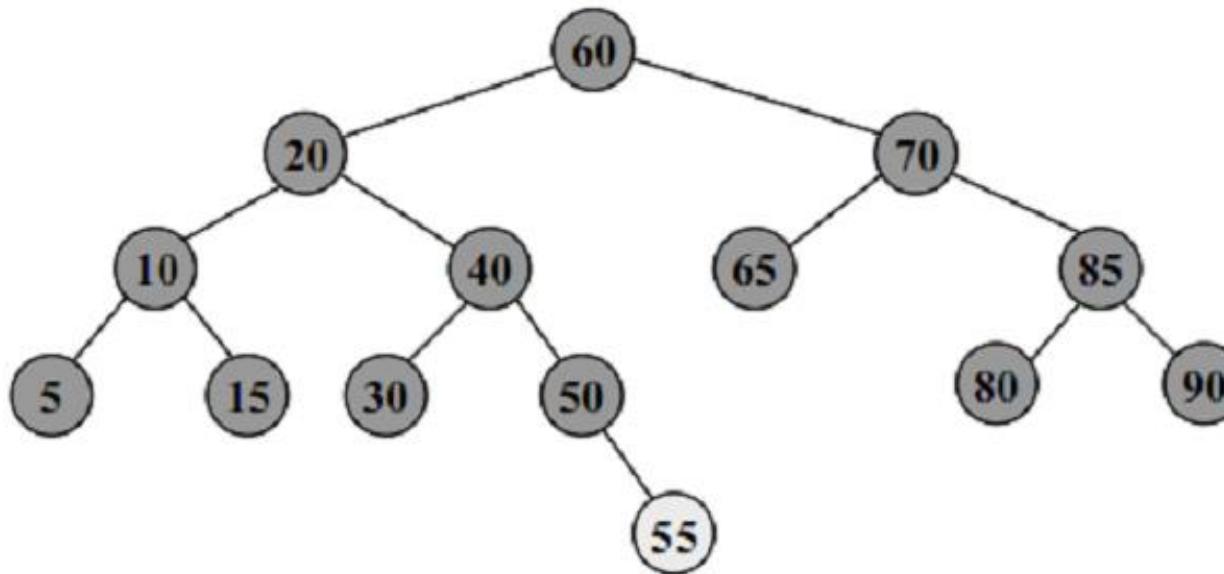
---

- Pada dasarnya sama seperti BST, tetapi ditambah dengan aksi rebalancing sesuai kasusnya
- Lokasi node yang berpotensi terganggu adalah titik ancestor bagi TKP



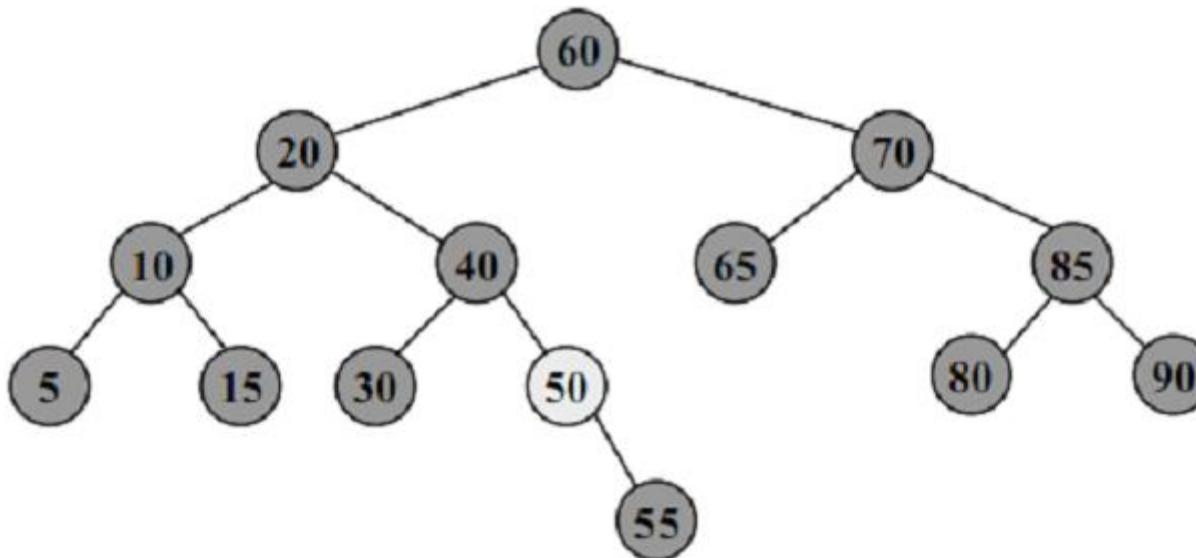
# Operasi Penghapusan AVL-Tree

- Bagaimana jika Node 55 dihapus pada AVL- Tree berikut:



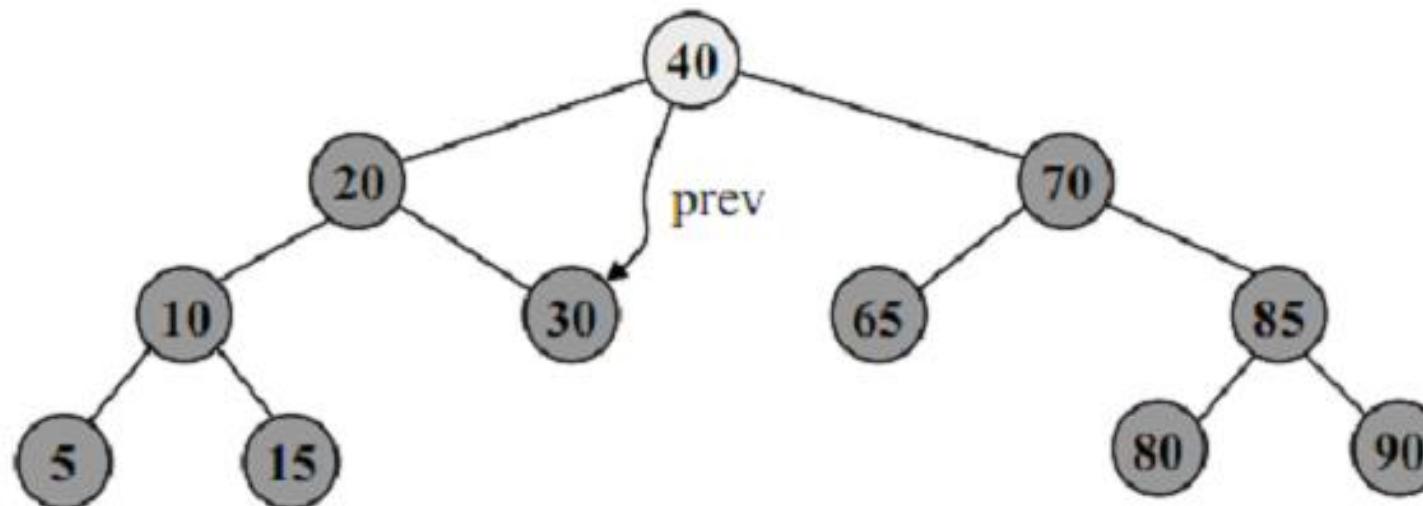
# Operasi Penghapusan AVL-Tree

- Bagaimana jika Node 50 dihapus pada AVL-Tree berikut:



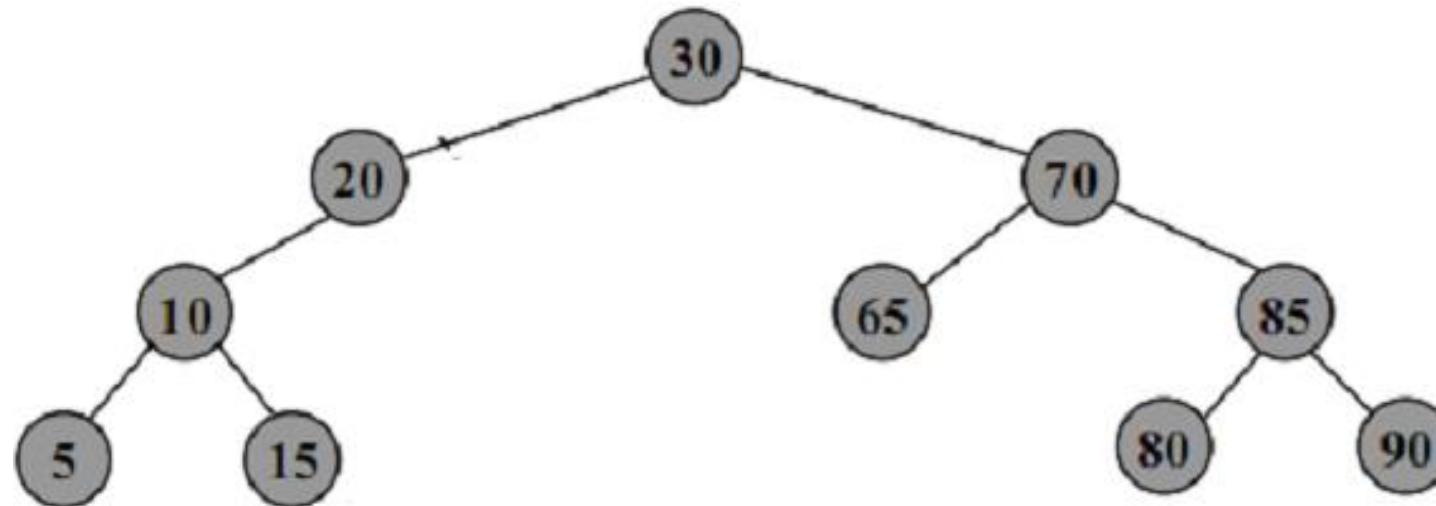
# Operasi Penghapusan AVL-Tree

- Bagaimana jika Node 40 dihapus pada AVL-Tree berikut:



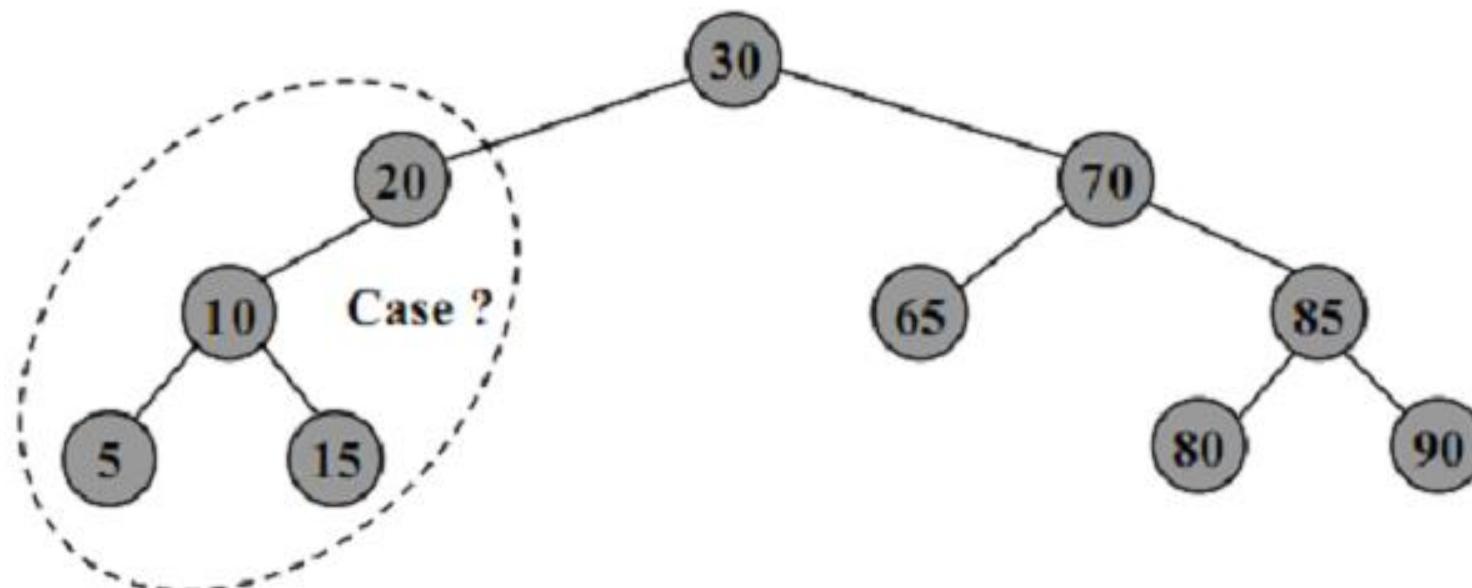
# Operasi Penghapusan AVL-Tree

- PredIn (40): 30 akan menggantikan node yang dihapus. Namun, apakah sudah sesuai?



# Operasi Penghapusan AVL-Tree

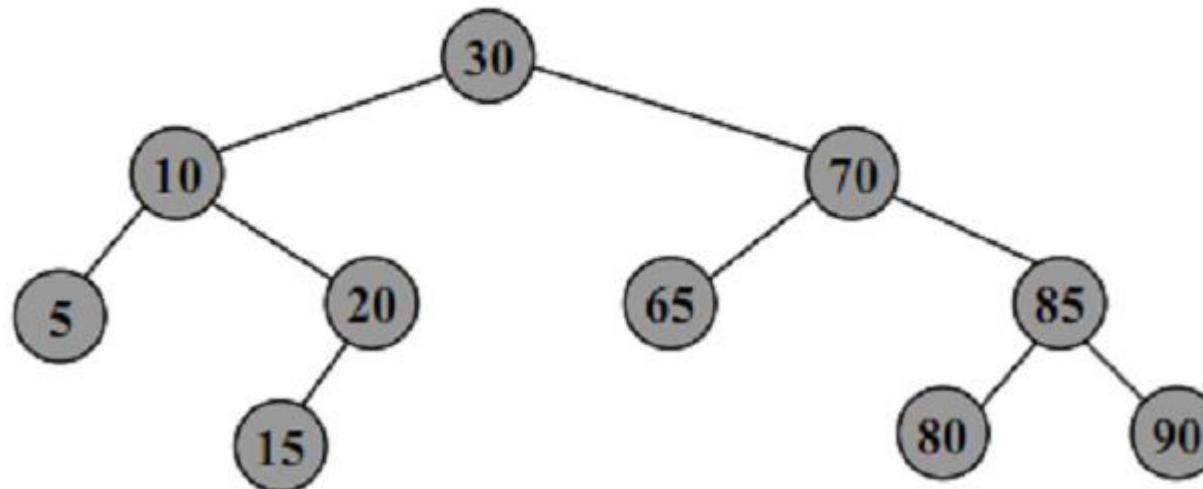
- PredIn (40): 30 akan menggantikan node yang dihapus. Namun, apakah sudah sesuai?



# Operasi Penghapusan AVL-Tree

- PredIn (40): 30 akan menggantikan node yang dihapus. Namun, apakah sudah sesuai?

Kasus 1 lebih diutamakan (lebih sederhana)



# Operasi Update

---

Jika ada aksi node x ingin diganti dengan node y, lakukan secara terurut:

- Hapus node x
- Insert node y



---

# **Implementasi AVL-Tree**

---



---

Wassalamu'alaikum ... Terima Kasih

---



# **Struktur Data berhirarki Bag III**

**Oleh: Muhammad Asyhar Agmalaro**

---

# B-Tree

---



# Pendahuluan

---

- AVL Tree menjamin tree selalu berada dalam keadaan (minimal) complete binary tree atau almost balanced
- Namun untuk jumlah node yang besar, perbedaan level pada AVL-tree tetap masih dikatakan belum optimum untuk pencarian dalam tree
- Cara/metode untuk menjaga keseimbangan (Re-balance) tetap dianggap memiliki high cost dalam kompleksitas



# Pendahuluan

---

- B-Tree dengan ide dasar 2,3 Tree mencoba menjaga keseimbangan Tree tanpa harus ada proses re-balance
- mengizinkan **jumlah key yang dapat ditampung oleh sebuah node bervariasi**, tak hanya satu key



# Pendahuluan

---

- Struktur berindeks untuk dataset yang berukuran besar tidak dapat langsung disimpan pada main memory
- Menyimpan data tersebut dalam disk harus membutuhkan beberapa pendekatan berbeda untuk mencapai efisiensi



## Pendahuluan

---

- Misal Disk/Cakram berputar 3600 rpm, 1 revolusi selama 1/60 detik atau 1,67 ms
- 1 cakram, dengan waktu yang sama dapat mengakses 200000 instruksi
- Misal diasumsikan menggunakan struktur AVL untuk menyimpan 20 jt records



# Pendahuluan

---

- Maka hasilnya berupa bynary tree yang sangat dalam dengan bnyak akses yang berbeda dari cakram
- $Time = \log_2 n \rightarrow \log_2 20\,000\,000$
- $Time \approx \log_2 2^{24} \approx 24 \text{ ms}$
- $Time \approx 0.2 \text{ Scnd}$



## Pendahuluan

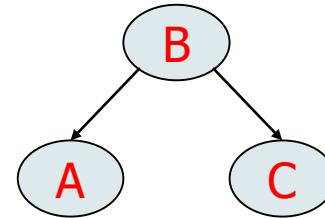
---

- Efisiensi dengan jumlah tampung node, tidak hanya 1 key.
- Setiap node bukan leaf diizinkan bisa diisi oleh nilai sejumlah satu, atau dua key, dan akan memperoleh 2 atau 3 descendant
- Tree tersebut dinamakan 2,3 tree

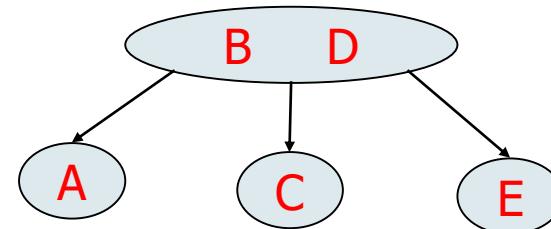


# 2,3 Tree Rule of placement

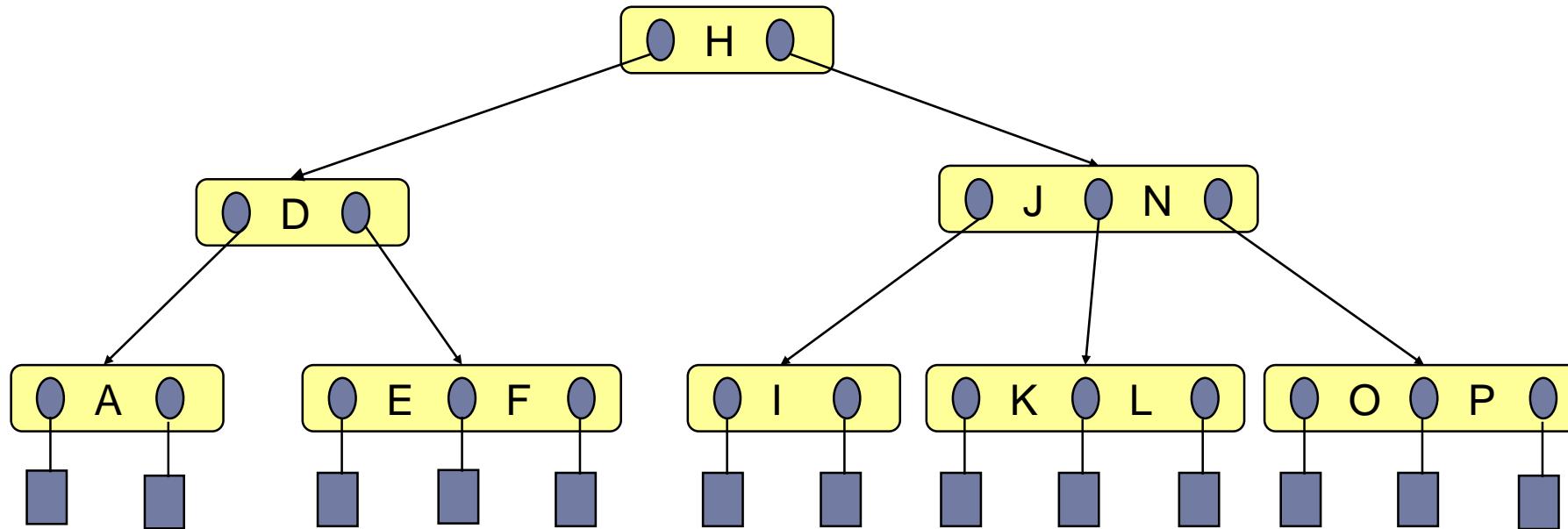
- ▶ Degree 2 (node berderajat 2)
  - ▶ key subtree kiri < keyroot <keysubtree kanan
  - ▶ A < B < C



- ▶ Degree 3 (node berderajat 3)
  - ▶ key subtree kiri < key root1 < key subtree tengah < key root2 < key subtree kanan
  - ▶ A < B < C < D < E

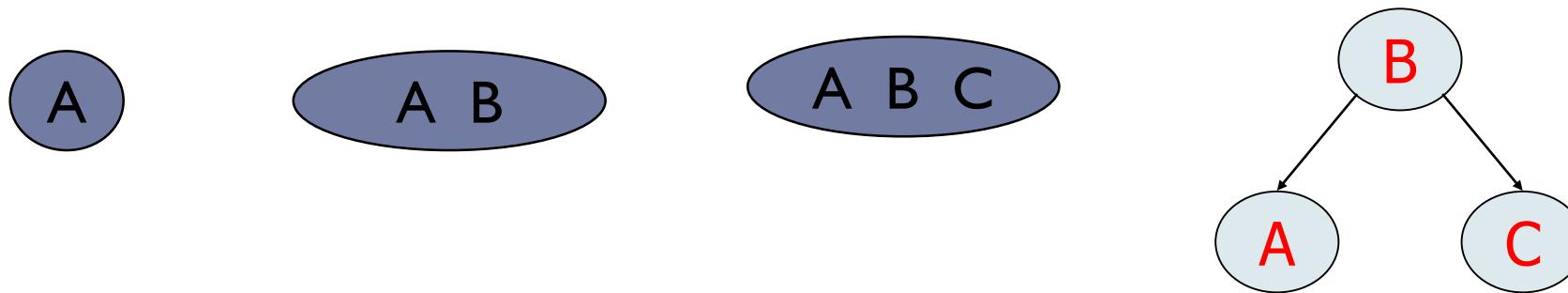


## 2,3 tree example



# 2,3 Tree Insertion

- Insert pada node dengan 1 elemen, langsung masuk
- Insert pada node dengan 2 elemen, urutkan search key, dan key yang di tengah naik menjadi parent.



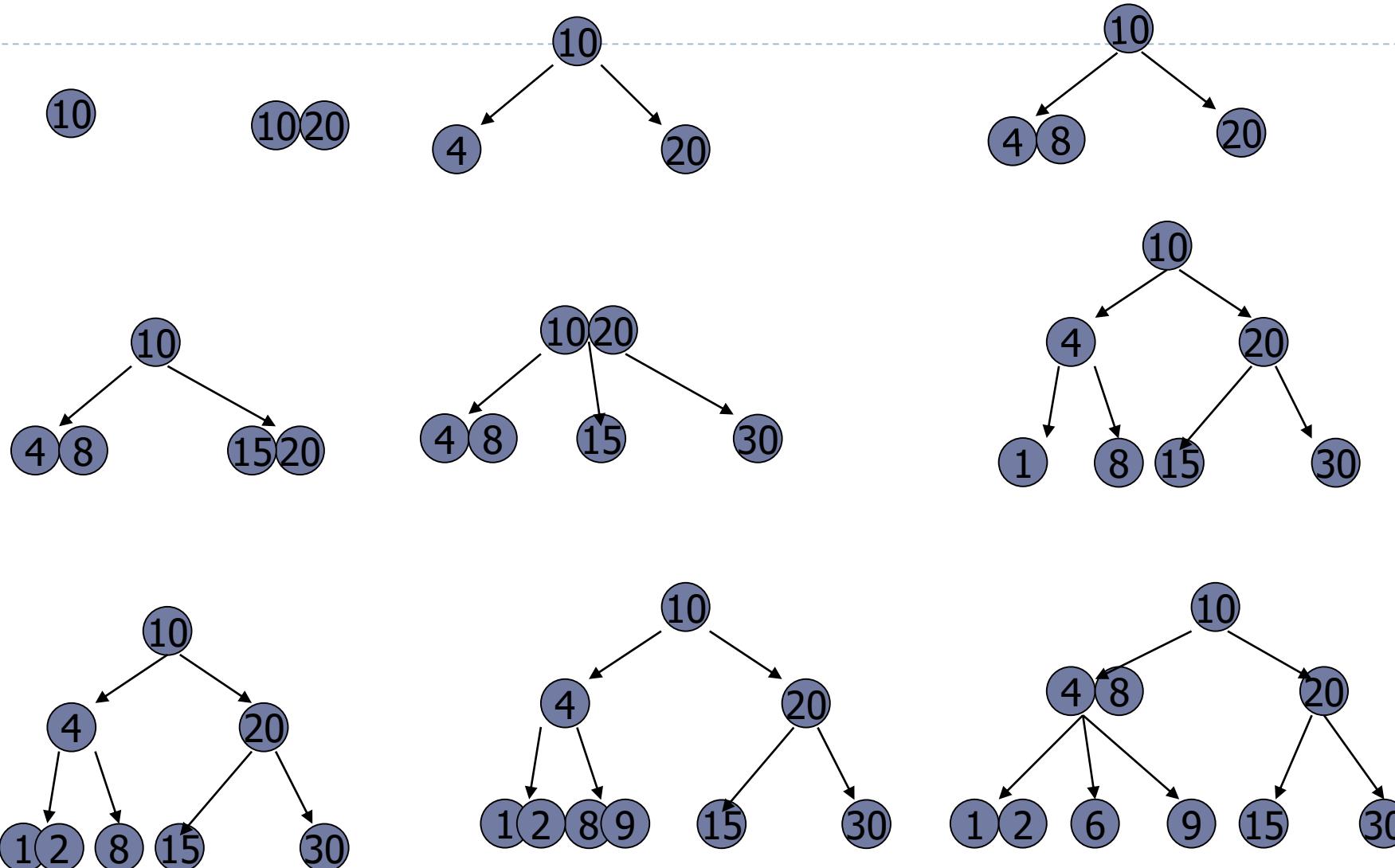
# Latihan I

---

- Masukkan ke dalam sebuah 2,3 tree masukan berikut:
- 10, 20, 4, 8, 15, 30, 1, 2, 9, 6



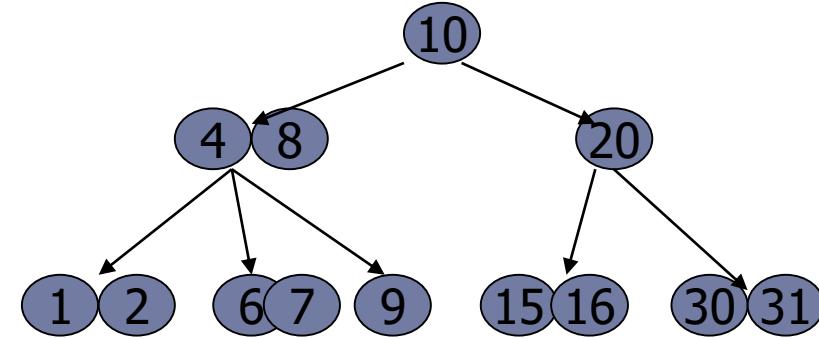
10, 20, 4, 8, 15, 30, 1, 2, 9, 6



▶ Masukkan 16

▶ Masukkan 31

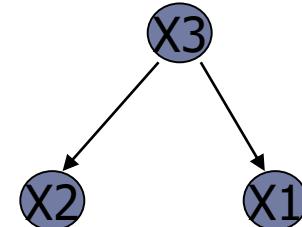
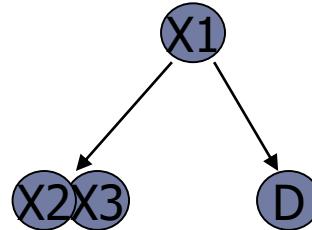
▶ Masukkan 7



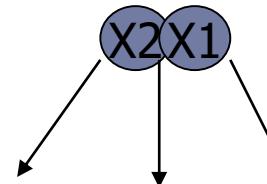
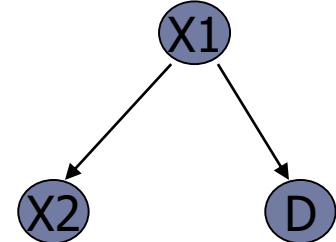
Example

# 2,3 tree deletion

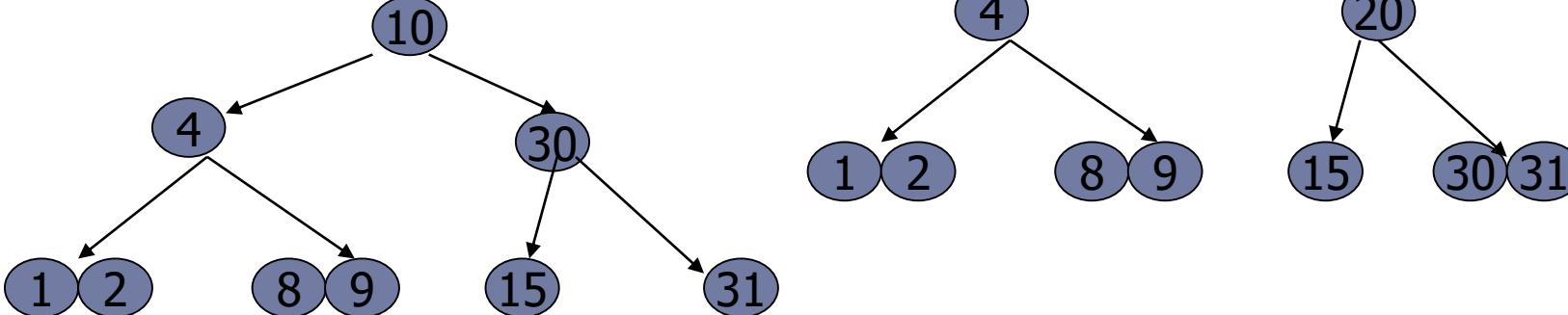
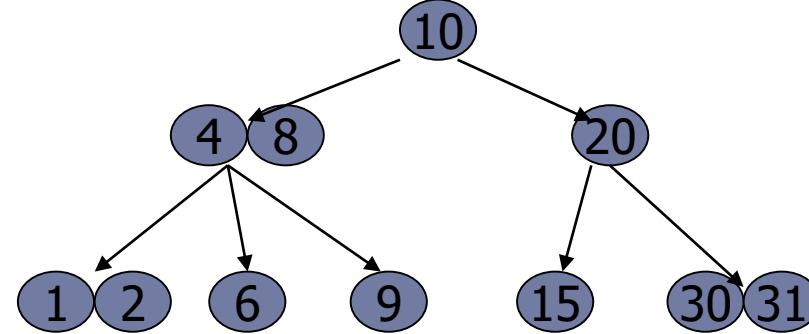
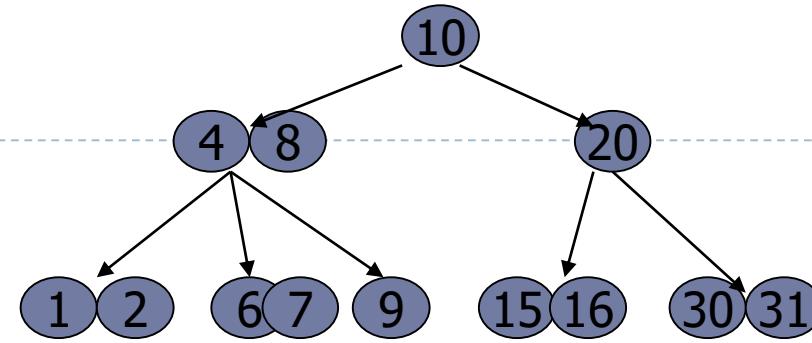
- ▶ Jika pada leaf dengan 2 elemen langsung hapus
- ▶ Jika pada leaf dengan 1 elemen lihat sibling
  - ▶ Jika sibling ada dua elemen:



- ▶ Jika sibling berderajat hanya 1 elemen:

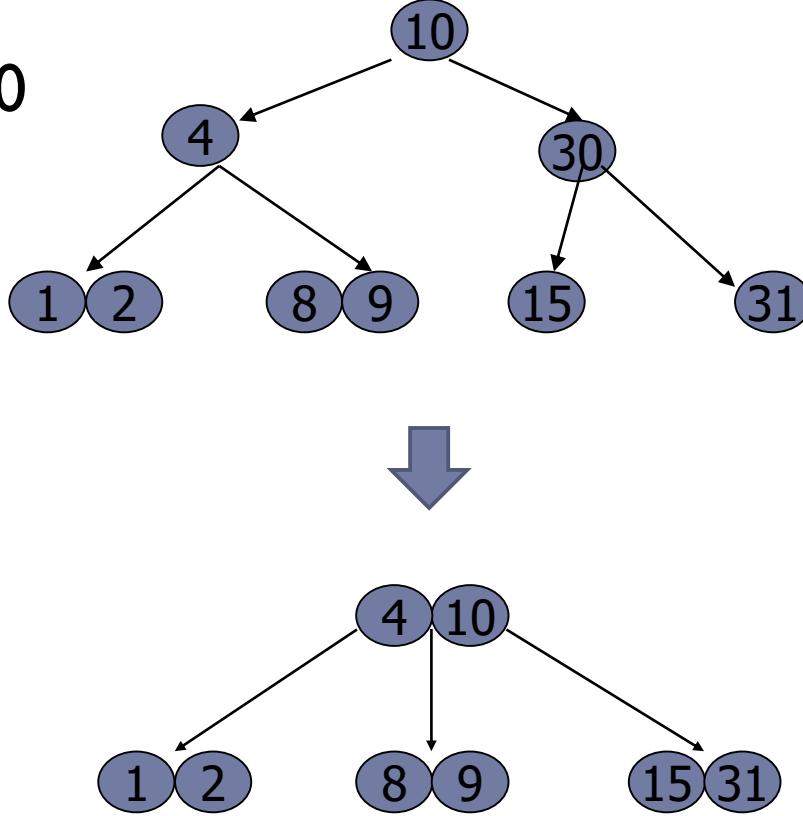


- ▶ Delete 7,16
- ▶ Delete 6
- ▶ Delete 20



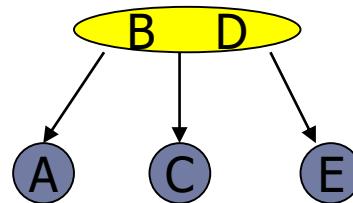
# Latihan II

- Delete 30



# B-Tree

- B-Tree, merupakan generalisasi dari a,b-Tree (contoh kasus  $a=2, b=3$ ) dengan B merupakan jumlah ordo



2,3 tree adalah juga a,b tree dengan ordo atau  $m=3$

- Semua leaf berada pada level yang sama
- Semua node internal kecuali root mempunyai paling banyak  $m$  anak, atau sedikitnya  $\lceil m/2 \rceil$  anak

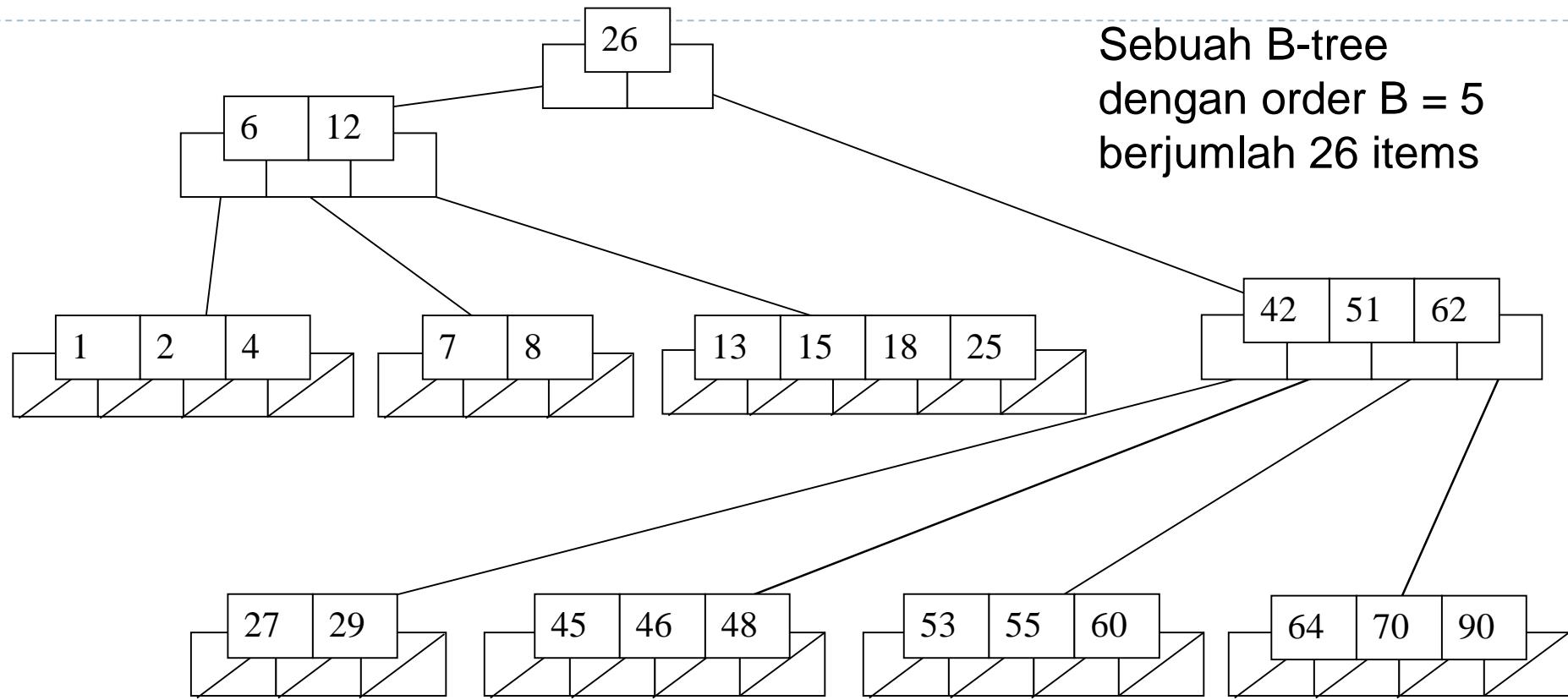


## B-Tree

---

- Setiap leaf berada pada level yang sama
- Root memiliki paling sedikit 1 key dan paling banyak  $m-1$  key (memiliki 2 sampai  $m$  subtree)  
CONTOH : root pada 2,3 tree memiliki paling sedikit 1 key paling banyak 2 key(memiliki 2 sampai 3 subtree) →  
 $m=3$
- M adalah branching factor dari a,b tree  
( $m=3$  untuk 2,3 tree)





Sebuah B-tree  
dengan order B = 5  
berjumlah 26 items

*Note that all the leaves are at the same level*

An example B-Tree



# Terminologi

---

- Key

Masukan pada node-node, baik root maupun children dan leaf.

- Branching factor

Faktor percabangan node

- Tree Kurus

Jika banyaknya cabang dari suatu node  $\lceil m/2 \rceil$

- Tree Gemuk

Jika banyaknya cabang dari suatu node m

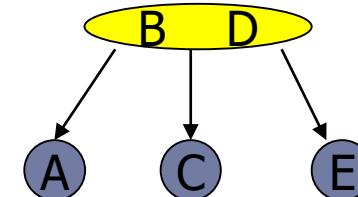


- Semua node internal kecuali root mempunyai paling banyak m cabang
- Misal : Jika  $m = 3$ , maka jumlah cabang paling banyak adalah 3

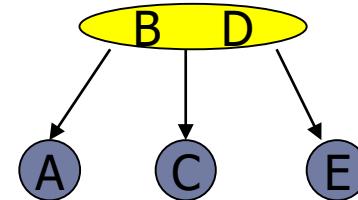
Bagaimana jika  $m=5$  ?

- ▶ Semua node internal kecuali root mempunyai sedikitnya  $\lceil m/2 \rceil$  cabang.
- Misal : Jika  $m = 3$ , maka jumlah cabang paling sedikit adalah  $\lceil m/2 \rceil = 2$

Bagaimana jika  $m=5$  ?



- Untuk masing-masing node memiliki key sedikitnya  $\lceil m/2 \rceil - 1$
- Misal : Jika  $m = 3$ , tiap node memiliki key sedikitnya  $3/2 - 1 = 1$   
Bagaimana jika  $m=5$  ?
- Untuk masing-masing node memiliki key maksimum  $m - 1$
- Misal : Jika  $m = 3$ , tiap node memiliki key maksimum  $3 - 1 = 2$   
Bagaimana jika  $m=5$  ?
- Root mempunyai paling banyak  $m$  cabang, tapi diperbolehkan sedikitnya mempunyai 2 cabang atau 0 cabang jika tree terdiri dari dari root saja.



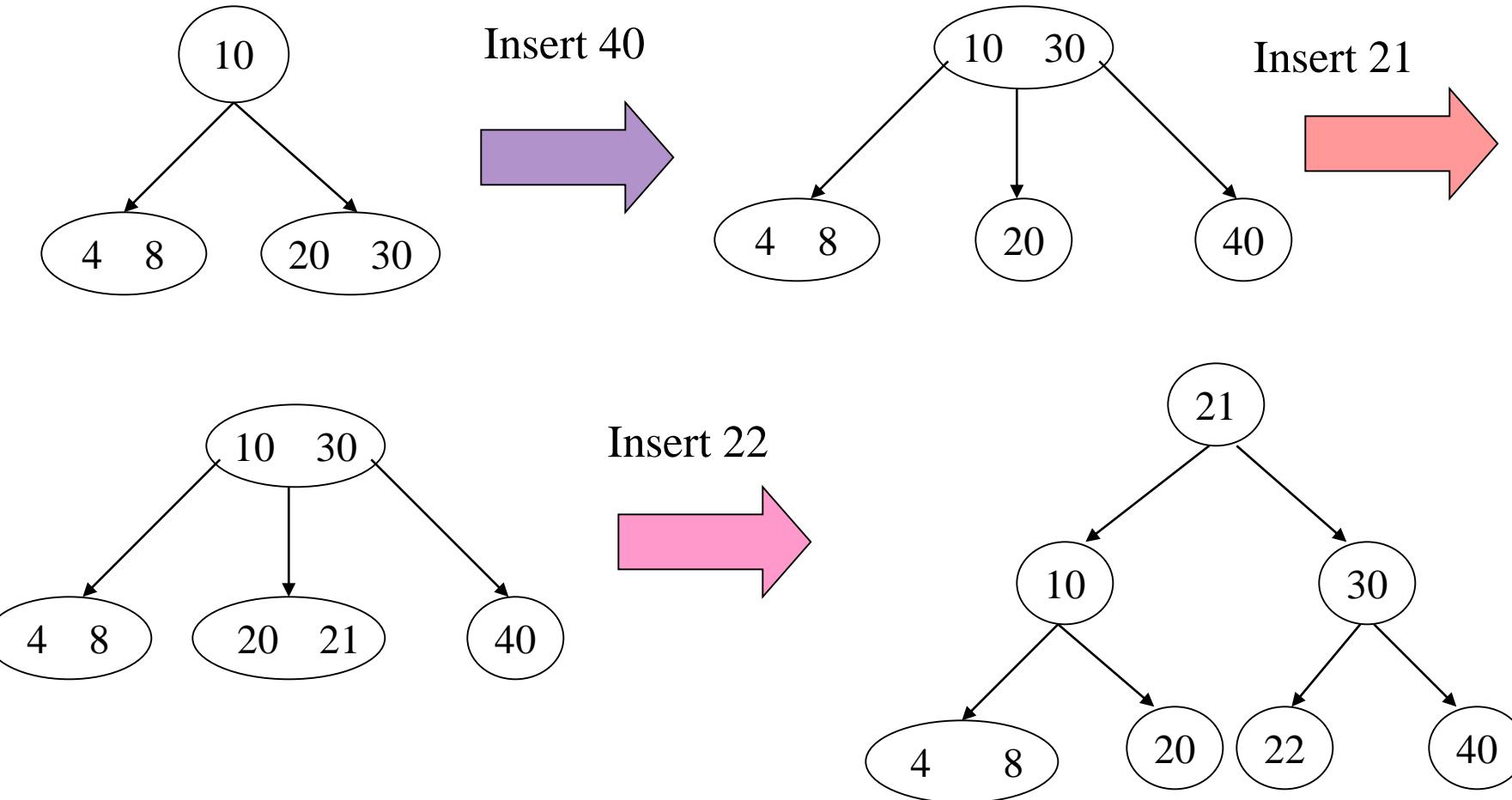
# Proses pada B-Tree

---

- *Inserting atau penyisipan*
  1. Key baru tersebut ditambahkan pada “leaf node” yang sesuai.
  2. Jika node sebelumnya tidak penuh , maka proses penyisipan dapat diselesaikan.
  3. Jika setelah ditambahkan nodenya menjadi penuh ( $\text{key} \geq m$ ), maka node tersebut pecah menjadi dua node pada level yang sama , kemudian median key disisipkan pada parent node.
  4. Jika parent menjadi penuh juga, ulangi langkah diatas untuk parent.



## Misalkan untuk B-tree ordo=3



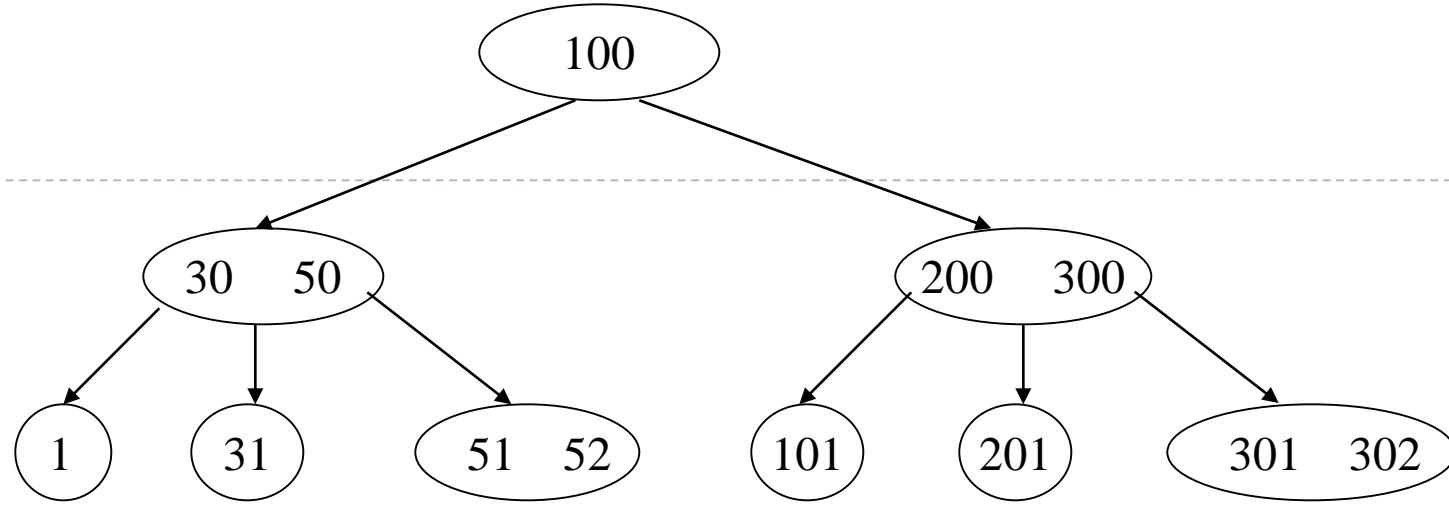
# Proses pada B-Tree

---

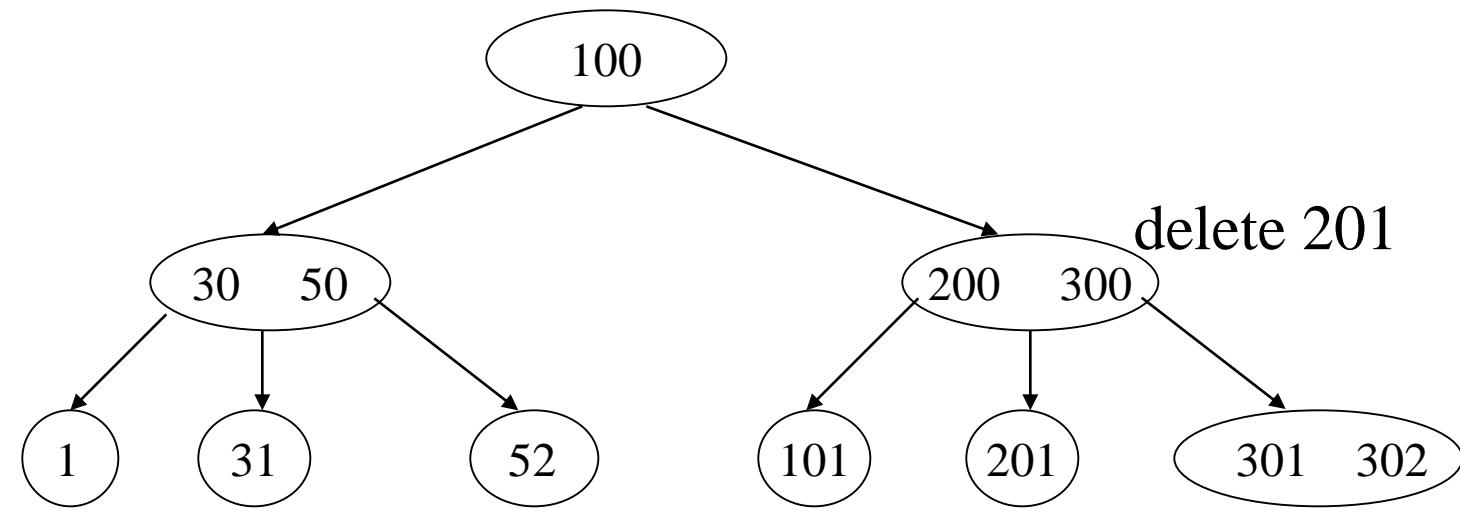
## □ *Deletion*

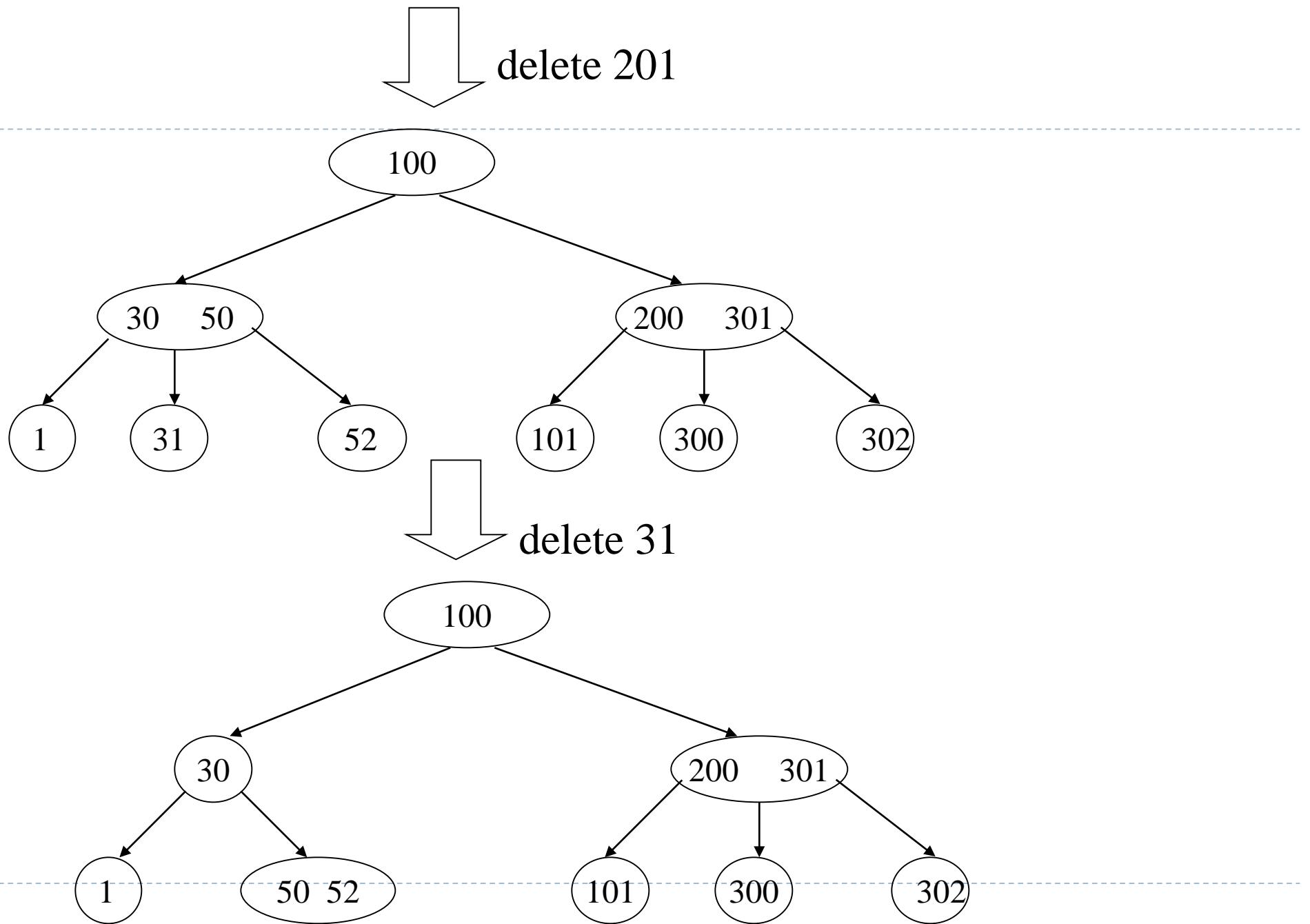
1. Jika entri yang akan dihapus tidak berada pada sebuah leaf, maka cari successor atau predecessornya untuk menggantikan tempat (seperti dalam BST).
2. Jika berisi key sebanyak  $\lceil m/2 \rceil - 1$ , maka satu di antaranya dapat dihapus secara langsung.
3. Jika berisi kurang dari jumlah minimum entri (jumlah key  $< \lceil m/2 \rceil - 1$ ) maka pinjam dari sibling kiri atau sibling kanan, yang mempunyai  $key > \lceil m/2 \rceil - 1$ .
4. Jika kedua sibling hanya memiliki jumlah key minimum, maka ambil sebuah key dari parent, kemudian gabungkan dengan salah satu sibling menjadi node baru.
5. Jika parent jadi kekurangan key, ulangi langkah diatas.





delete 51





# Latihan III

---

- ▶ Buat Struktur B-Tree dengan ordo  $m=4$
- ▶ Setiap node memiliki pointer paling banyak 4 dan 3 keys, dan sedikitnya 2 pointers dan 1 key.
- ▶ Lakukan proses berikut ini:
  - ▶ Konstruksi + Insert : 5, 3, 21, 9, 1, 13, 2, 7, 10, 12, 4, 8
  - ▶ Delete : 2, 21, 10, 3, 4



# Latihan III

---

Insert 5, 3, 21

\* 5 \*

a

\* 3 \* 5 \*

a

\* 3 \* 5 \* 21 \*

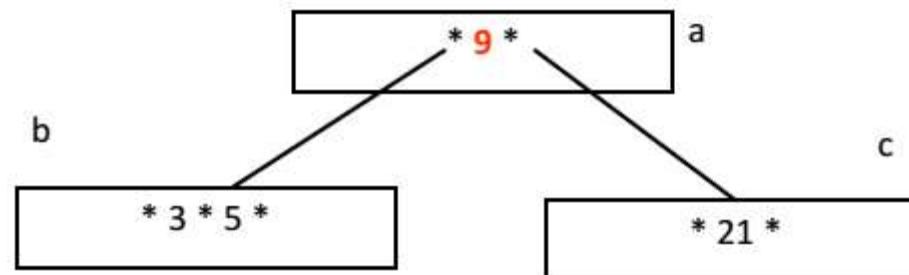
a



# Latihan III

---

## Insert 9



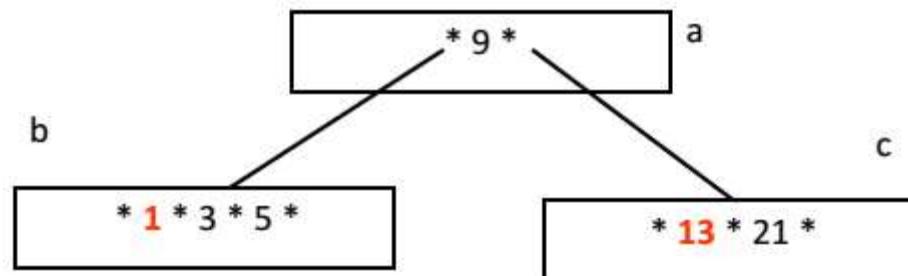
Node a splits creating 2 children: b and c



# Latihan III

---

Insert 1, 13



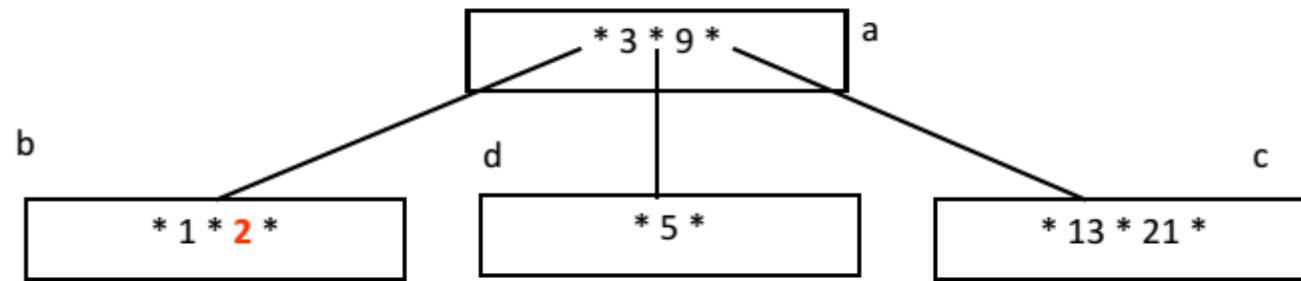
Nodes b and c have room to insert more elements



# Latihan III

---

## Insert 2



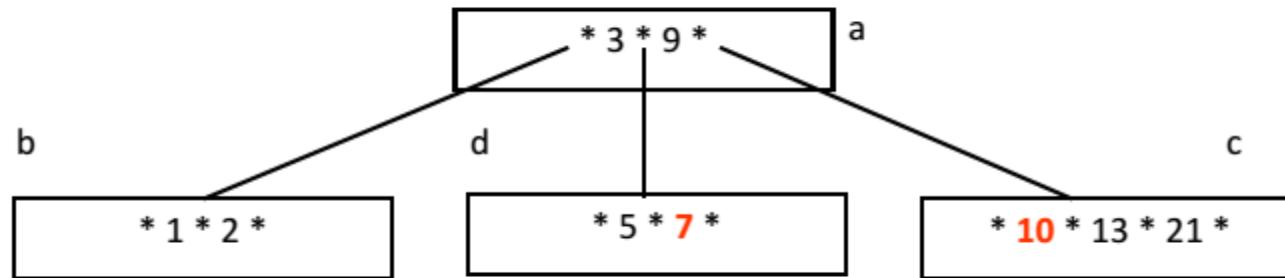
Node b has no more room, so it splits creating node d.



# Latihan III

---

Insert 7, 10



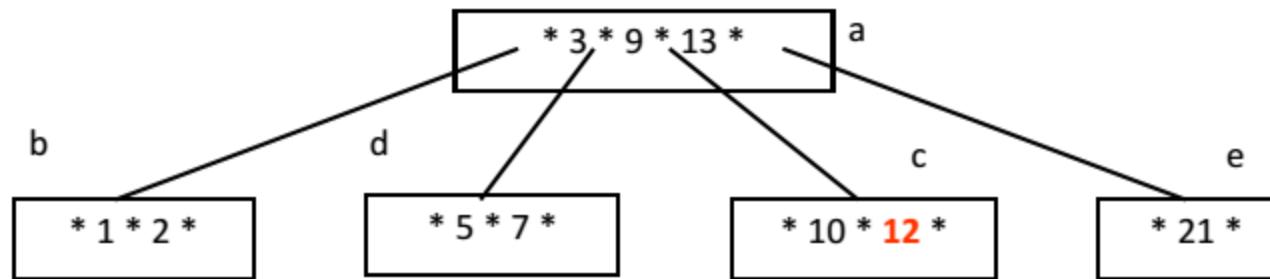
Nodes d and c have room to add more elements



# Latihan III

---

## Insert 12



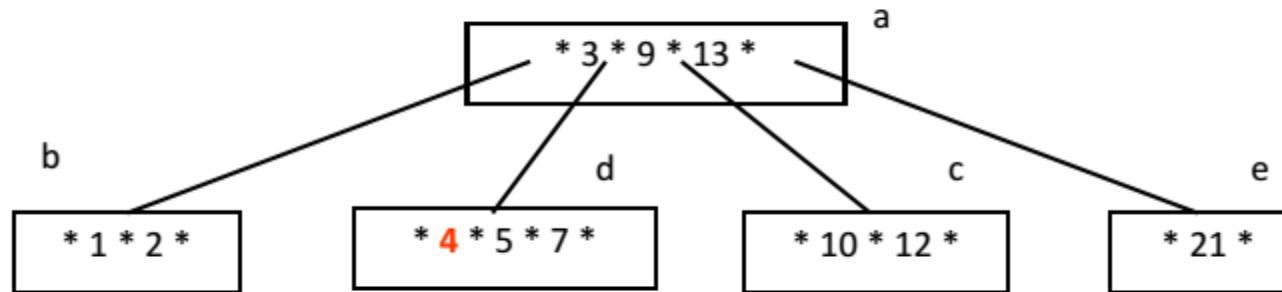
Nodes c must split into nodes c and e



# Latihan III

---

## Insert 4

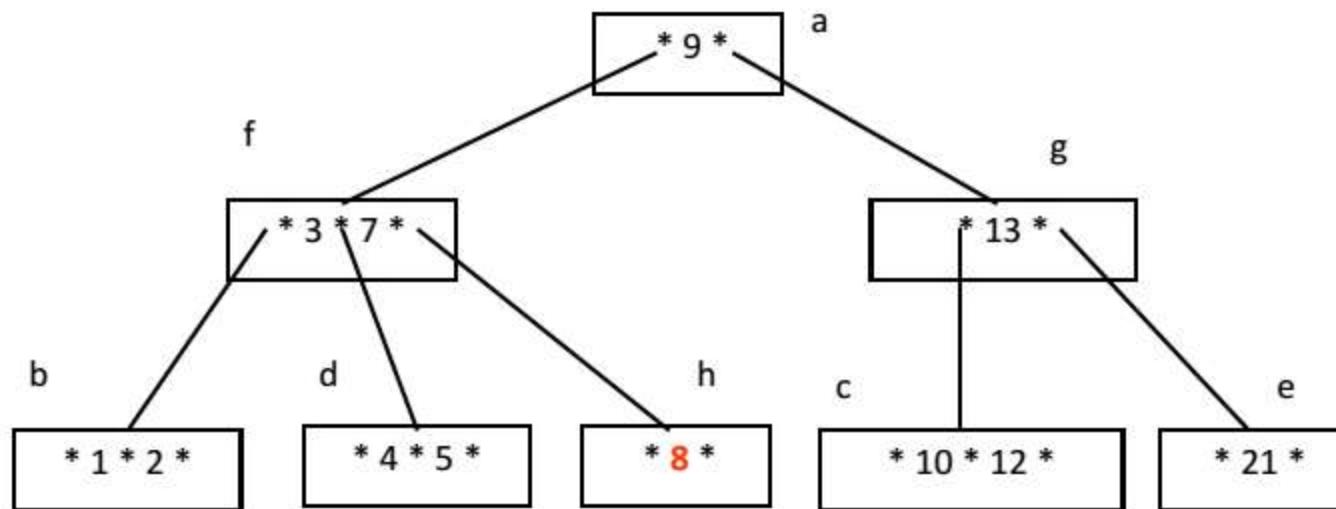


Node d has room for another element



# Latihan III

## Insert 8

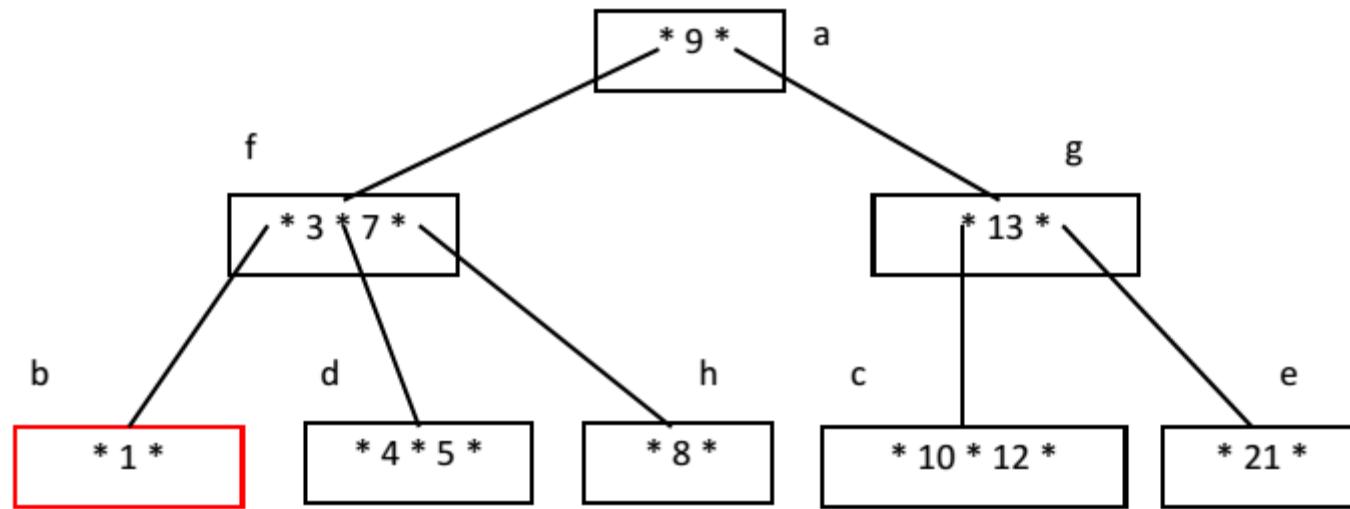


Node d must split into 2 nodes. This causes node a to split into 2 nodes and the tree grows a level.



# Latihan III

## Delete 2

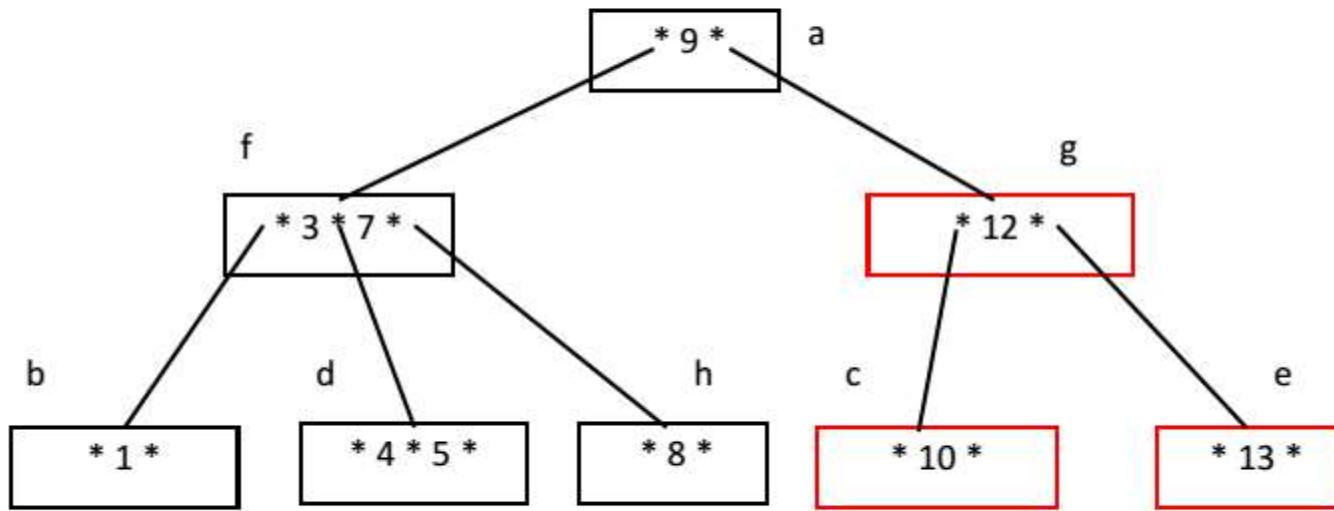


Node b can loose an element without underflow.



# Latihan III

## Delete 21

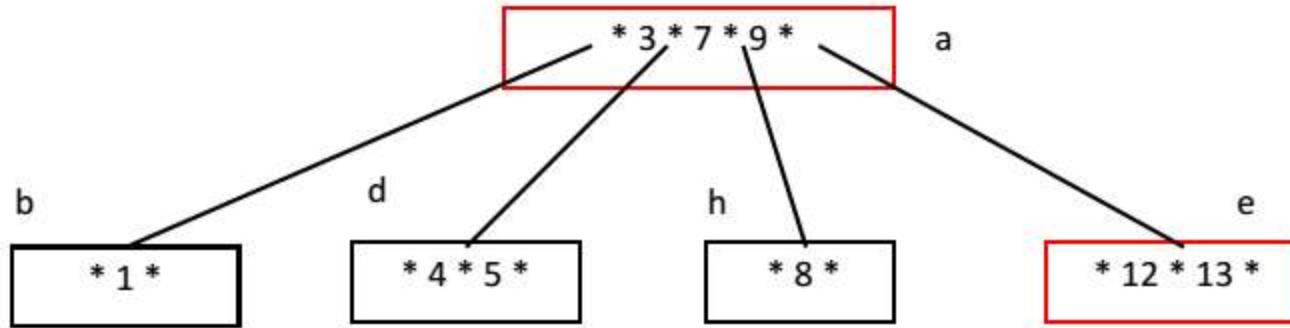


Deleting 21 causes node e to underflow, so elements are redistributed between nodes c, g, and e



# Latihan III

## Delete 10

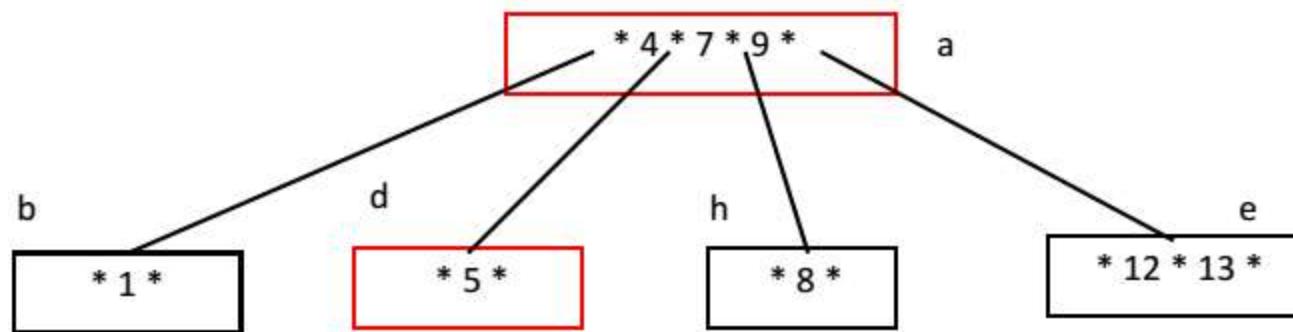


Deleting 10 causes node c to underflow. This causes the parent, node g to recombine with nodes f and a. This causes the tree to shrink one level.



# Latihan III

## Delete 3

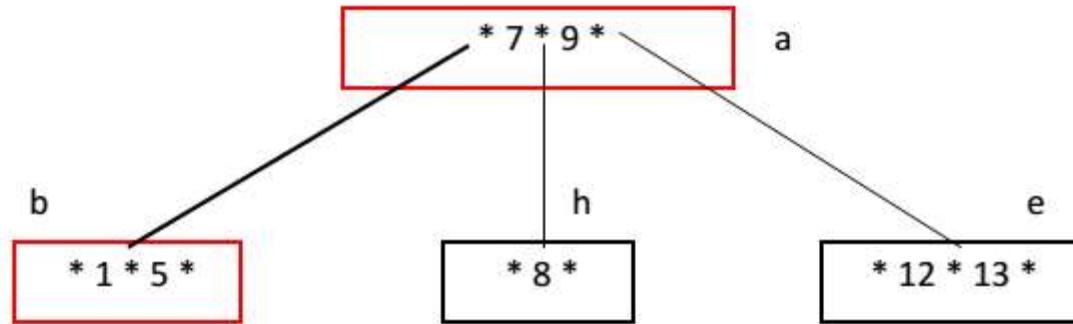


Because 3 is a pointer to nodes below it, deleting 3 requires keys to be redistributed between nodes a and d.



# Latihan III

## Delete 4



Deleting 4 requires a redistribution of the keys in the subtrees of 4; however, nodes b and d do not have enough keys to redistribute without causing an underflow. Thus, nodes b and d must be combined.



---

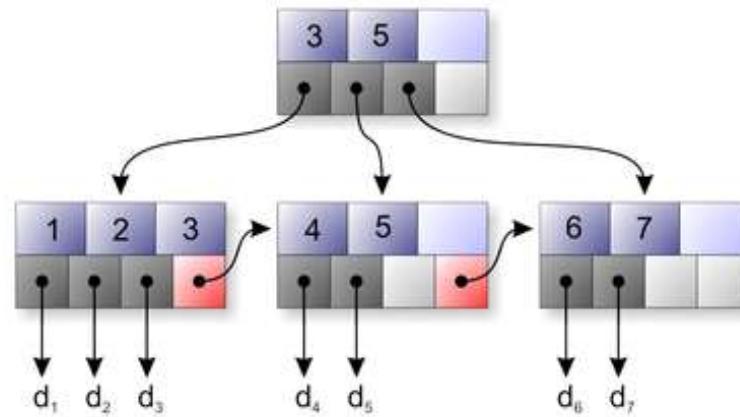
# R-Tree

---



# Pendahuluan

- ▶ B-Tree adalah struktur data untuk menyimpan data terutut dengan kompleksitas waktu yang lebih baik dibandingkan dengan metode BST dimana pada B-Tree dapat menyimpan lebih dari 1 key pada suatu node



- ▶ B-Tree sangat baik dan efisien untuk penyimpanan dengan tipe data tertentu (biasanya numerik), namun tidak dapat menimpan data dalam bentuk geometric/data dengan multi dimensi (cth: data spasial, lokasi, citra)

# Pendahuluan

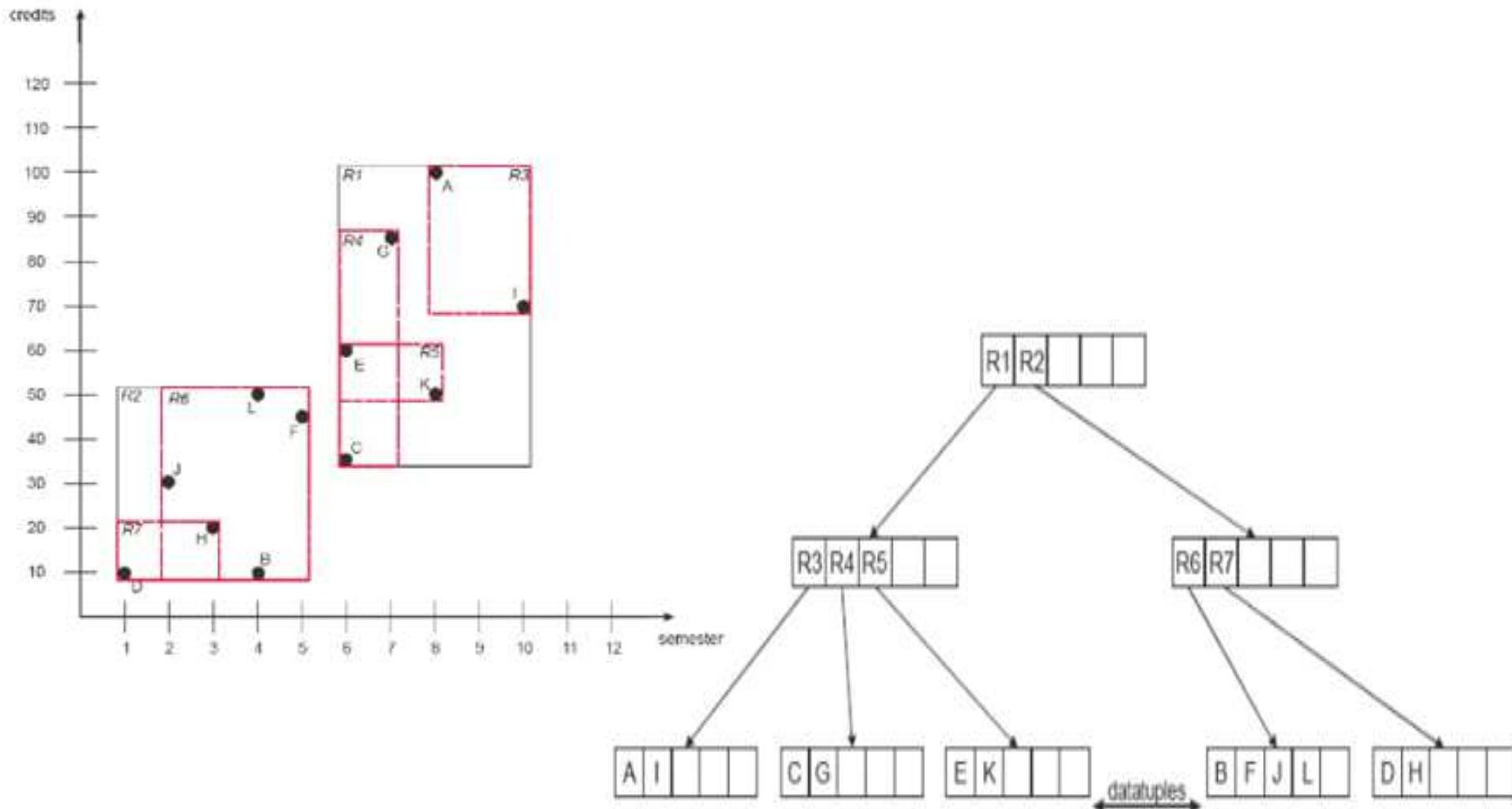
---

- ▶ R-Tree adalah struktur data advance dengan karakteristik seperti B-Tree namun dapat mengorganisir data berdimensi 2 atau 3 dengan mengoperasikan data tersebut dengan sebuah “Minimum *Bounding Box*”
- ▶ Setiap Node membatasi setiap node *child*
- ▶ Sebuah Node dapat memiliki banyak objek/nilai di dalamnya
- ▶ Node Leaf menunjuk kepada nilai objek actual (nilai spasial)
- ▶ Kedaaman atau tinggi dari Tree selalu seimbang (mengikuti karakteristik B-Tree)



# Contoh R-Tree

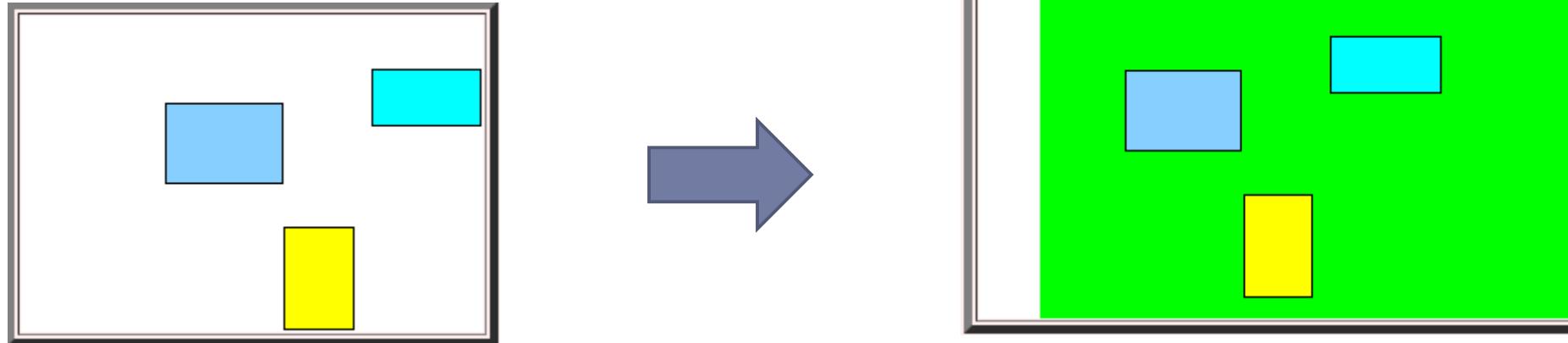
name	semester	credits
A	8	100
B	4	10
C	6	35
D	1	10
E	6	40
F	5	45
G	7	85
H	3	20
I	10	70
J	2	30
K	8	50
L	4	50



# Terminologi

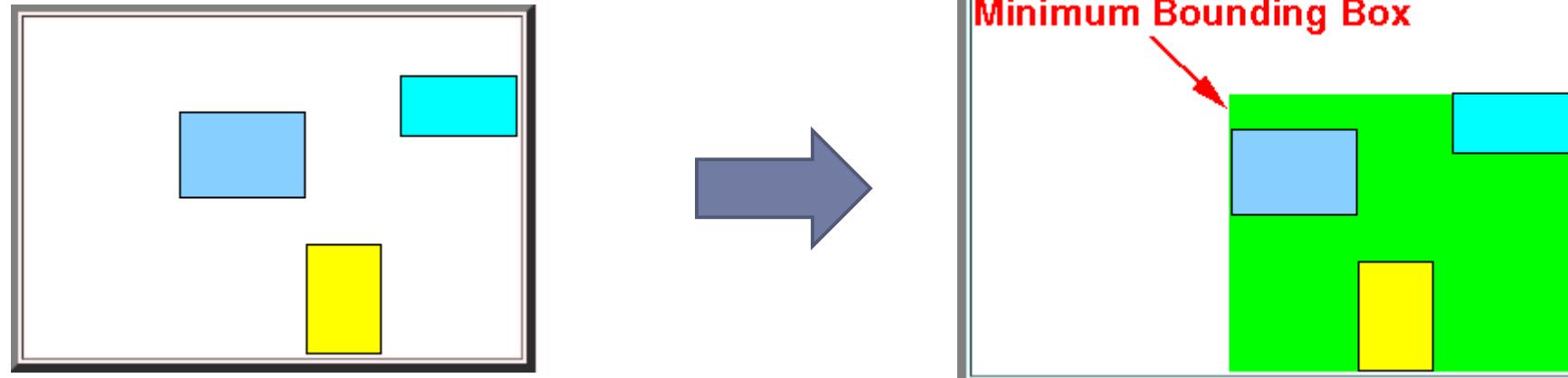
---

- ▶ Bounding Box: Sebuah 4 persegi yang berisikan sebuah grup dari Object/nilai



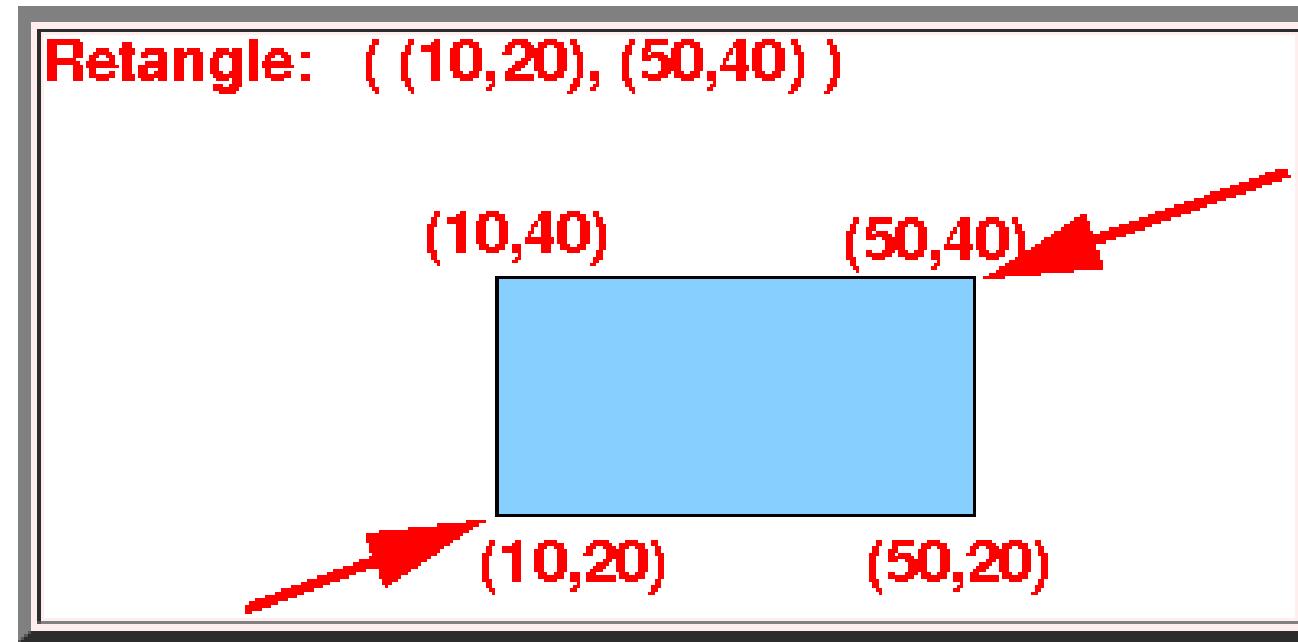
## Terminologi

- ▶ Minimum Bounding Box: **4 persegi yang paling kecil** yang dapat menyimpan grup dari Object/nilai

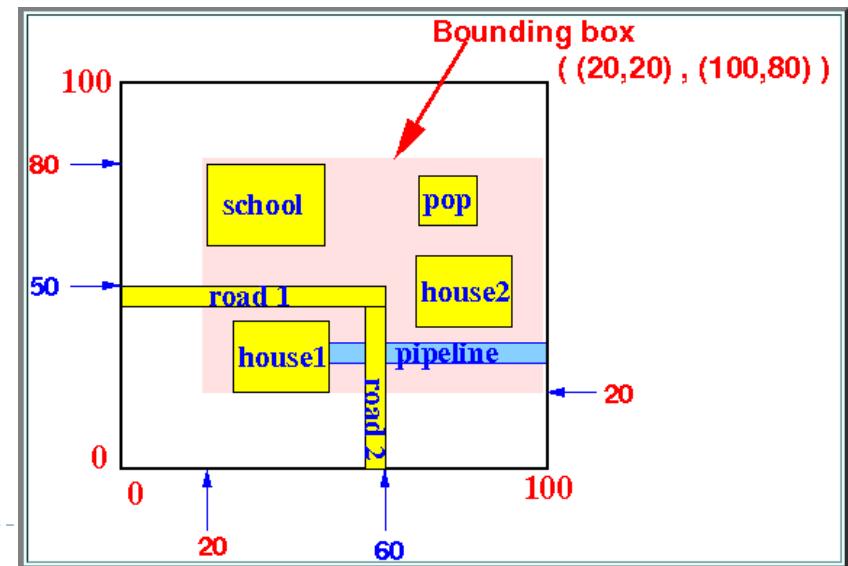
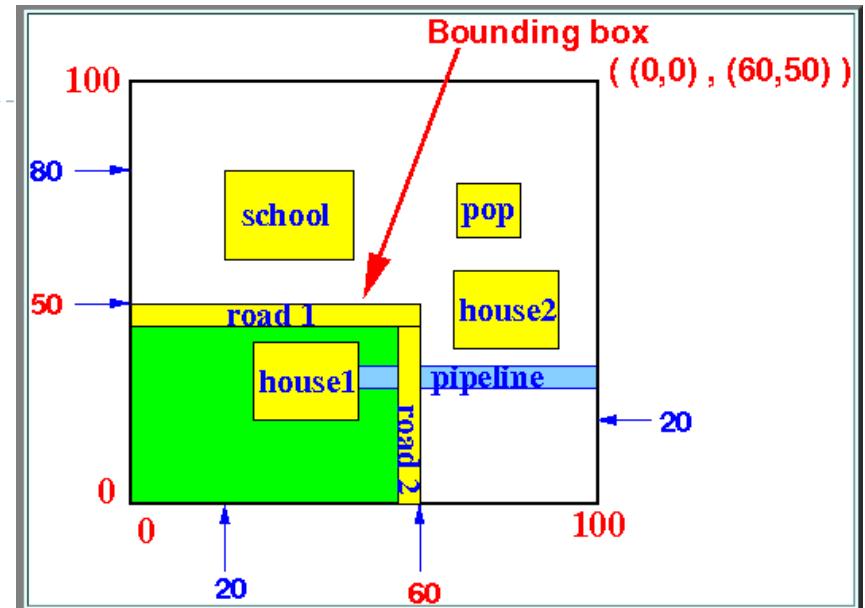
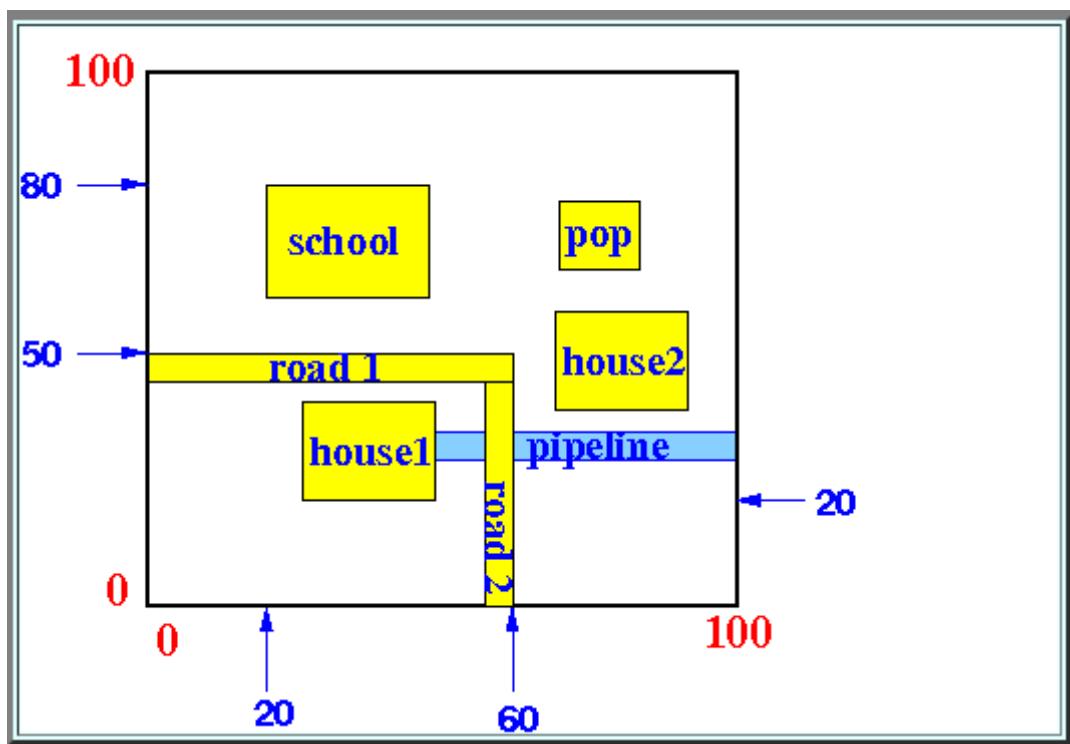


## Terminologi

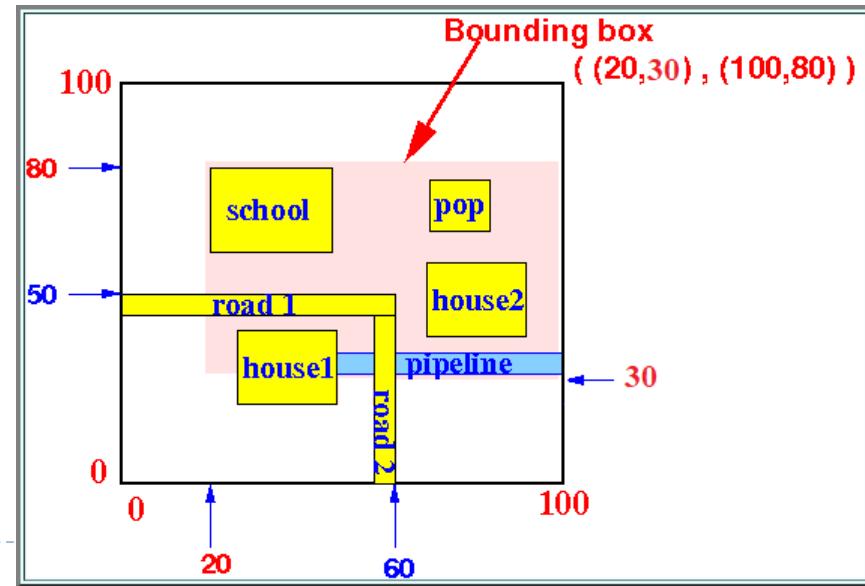
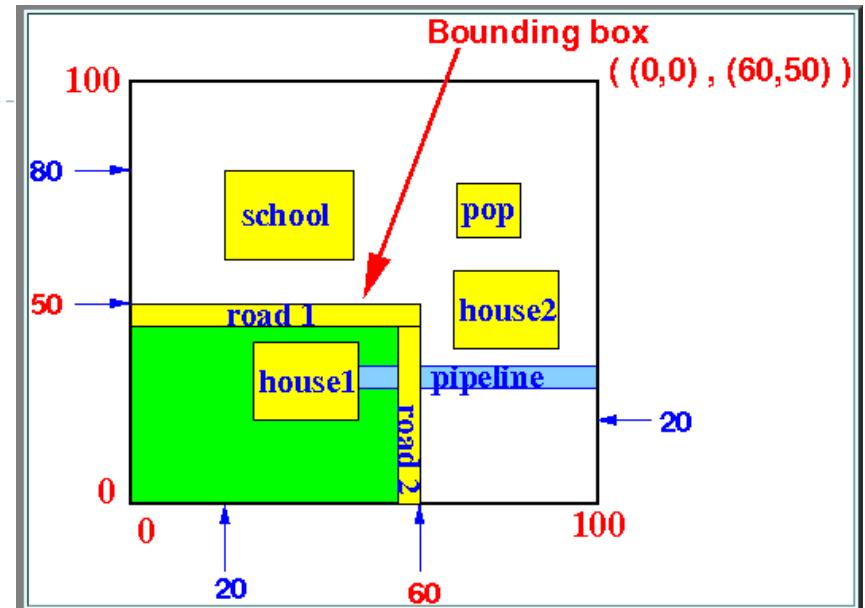
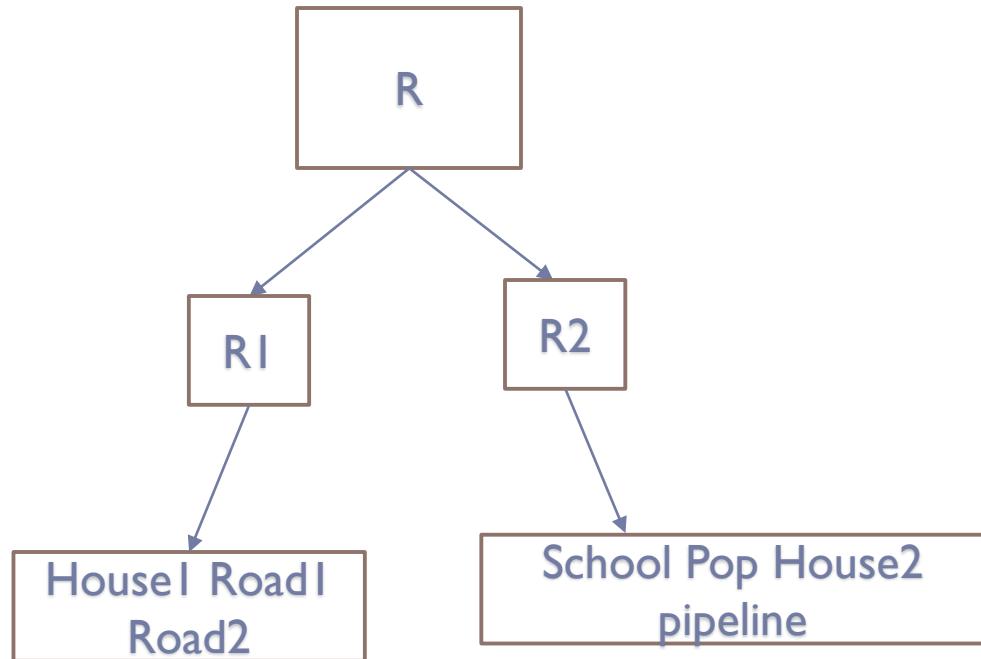
- ▶ Sebuah rectangle(4 persegi) dapat direpresentasikan nilai/object spasial pada pojok kiri bawah dan pojok kanan atas, atau pojok kiri atas dan pojok kanan bawah.



# Ilustrasi

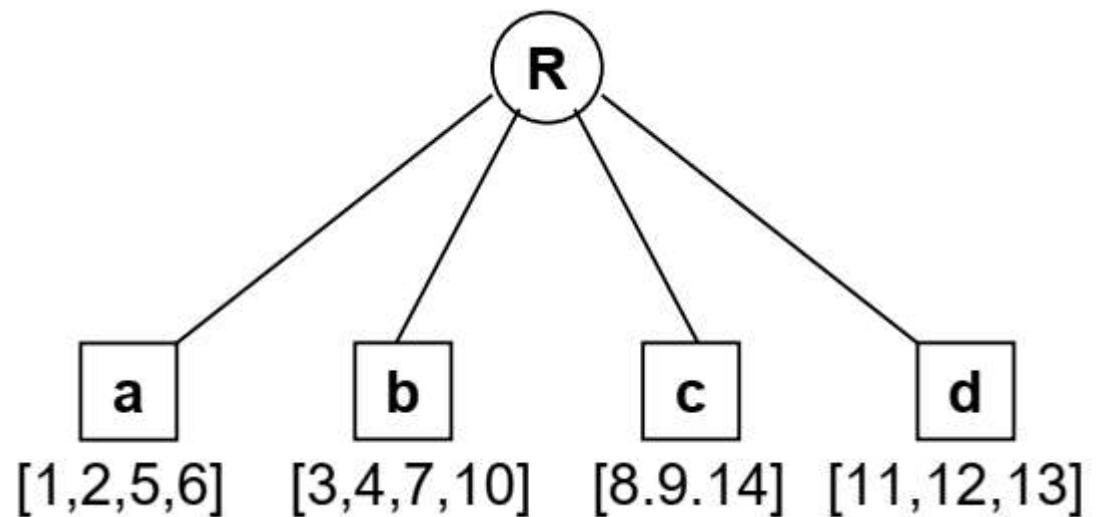
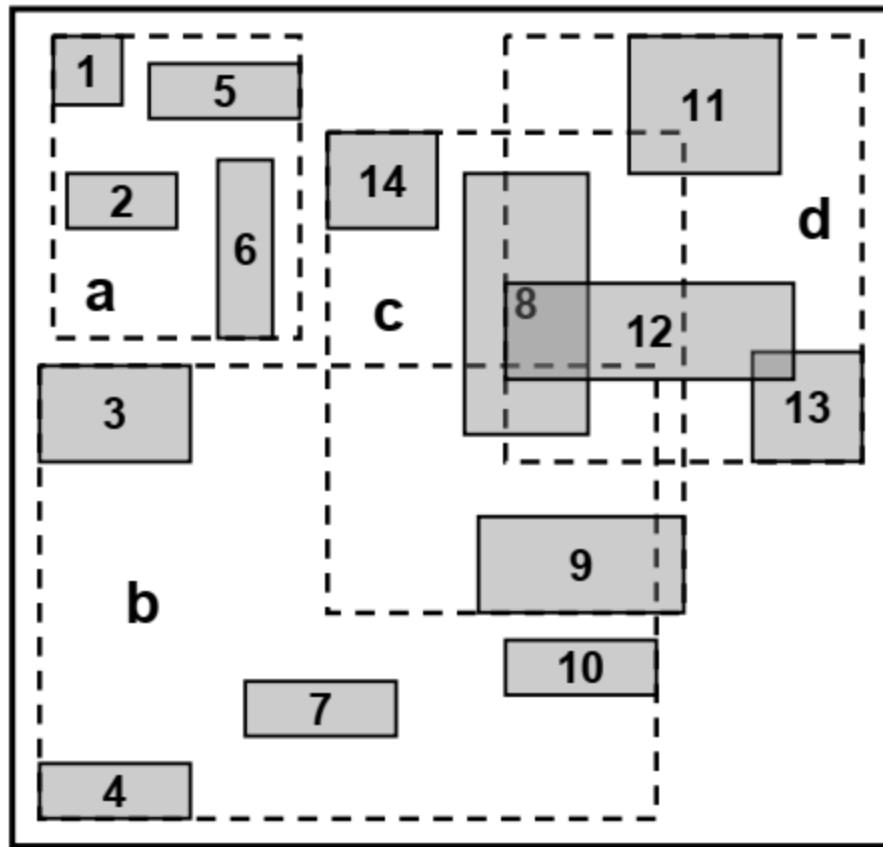


# Ilustrasi



## Contoh 1

- Tentukan R-Tree dari data multi dimensi berikut ini dengan M=4



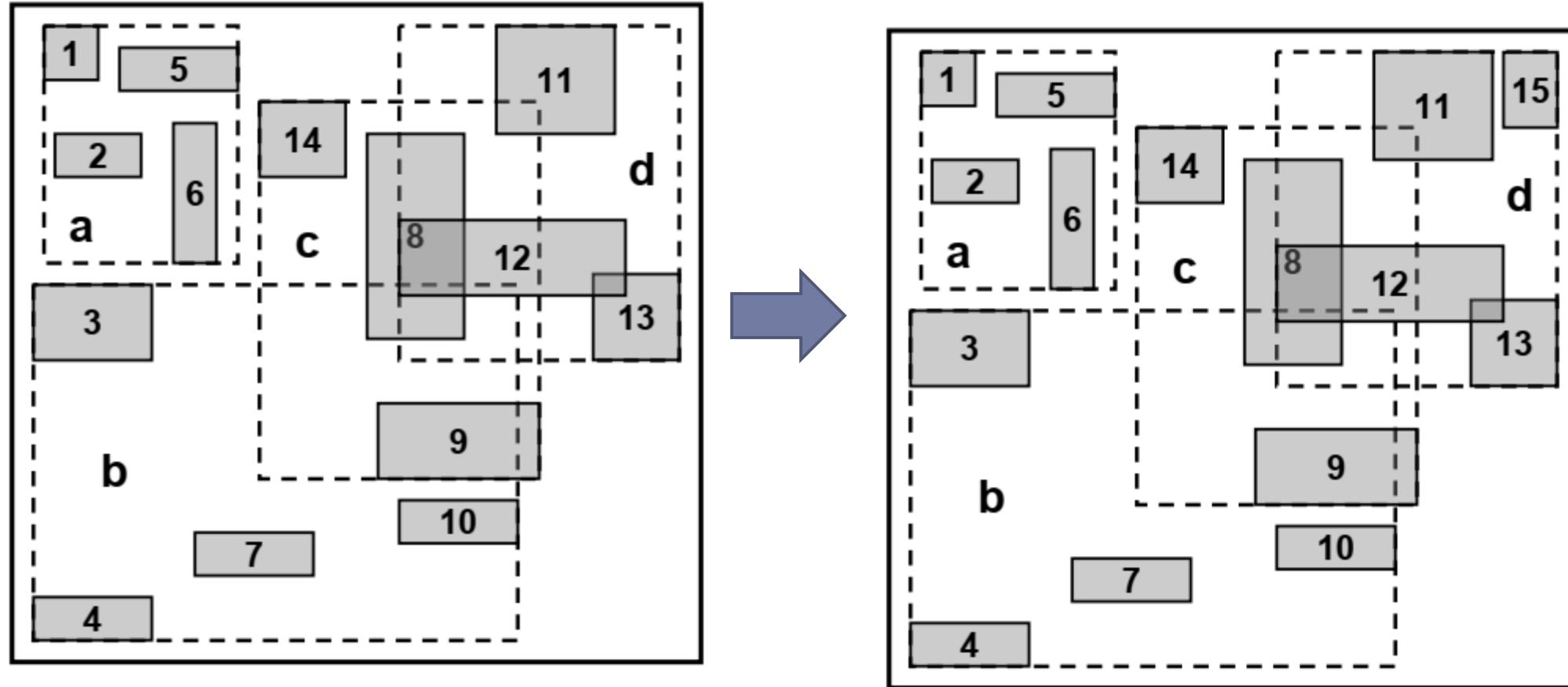
## Operasi pada R-Tree

---

- ▶ Searching: Lihat semua node yang *intersect*, kemudian kemudian lakukan pencarian ke dalam node
- ▶ Insertion: Sama seperti B-Tree tetapkan lokasi object ke dalam node, jika node penuh, maka split node
- ▶ Deletion: Sama seperti B-Tree, Jika saat delete node menjadi penuh, lakukan operasi penyeimbangan

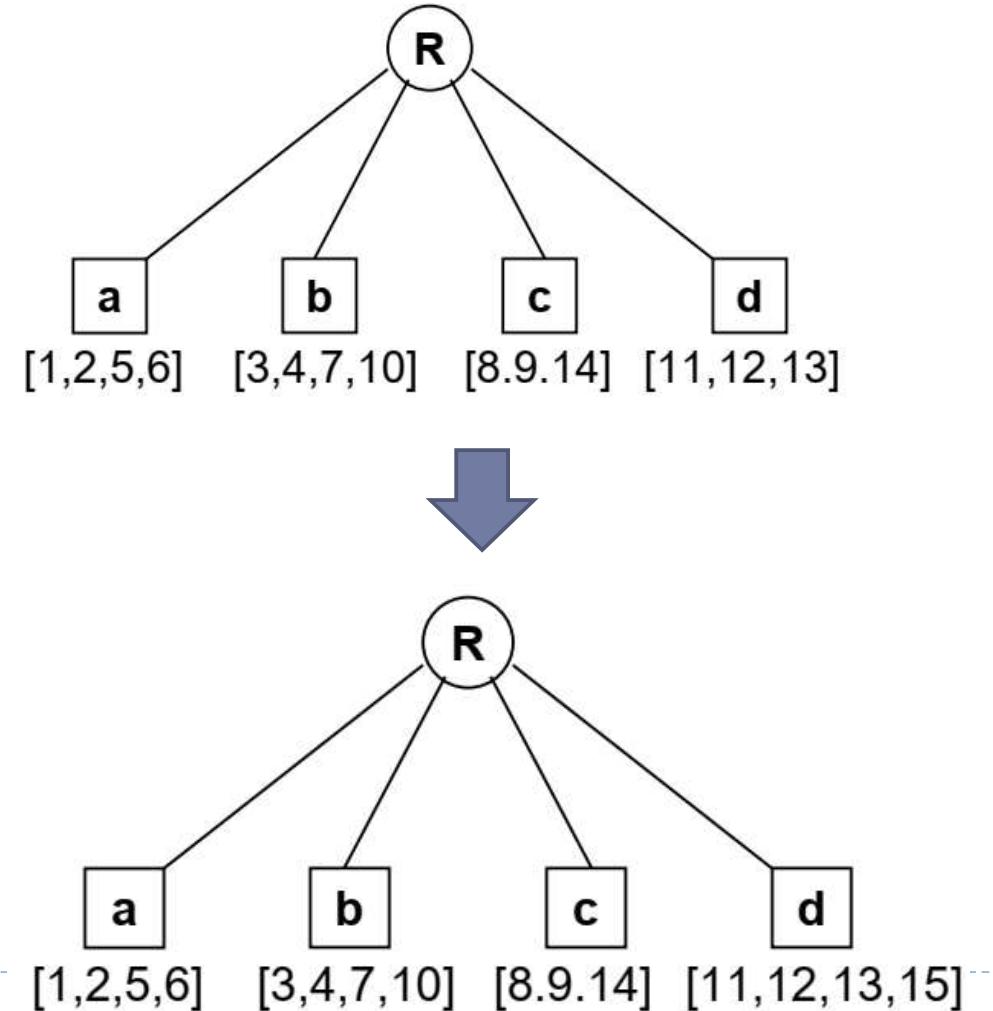
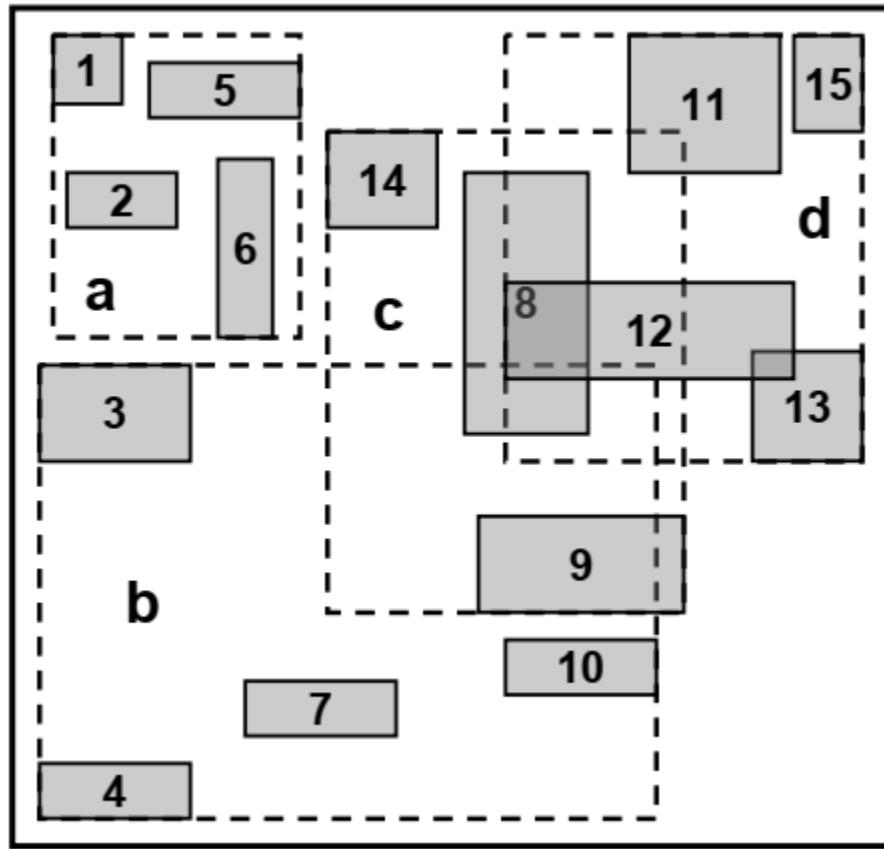
## Contoh 2

- Tentukan lokasi object baru (dengan nilai 15) berikut ini dengan  $M=4$



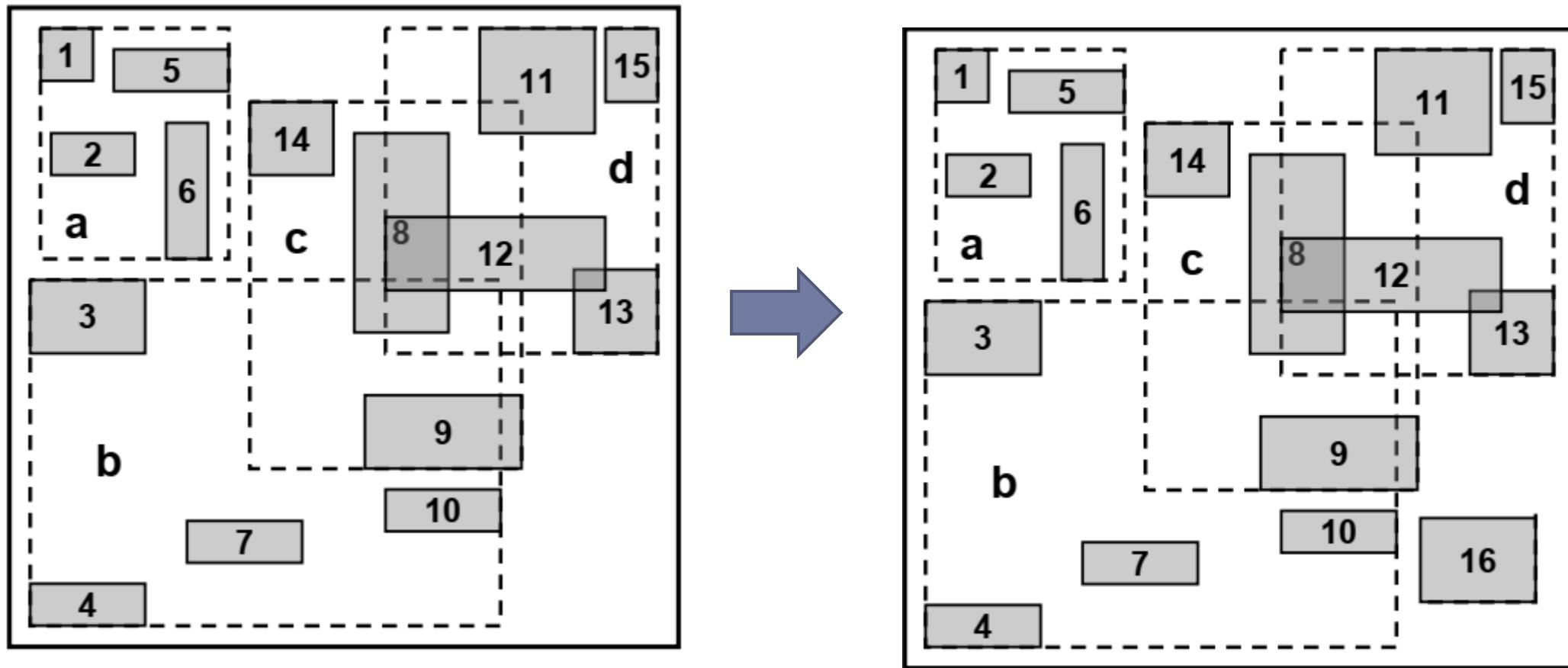
## Contoh 2

- Tentukan lokasi object baru (dengan nilai 15) berikut ini dengan  $M=4$



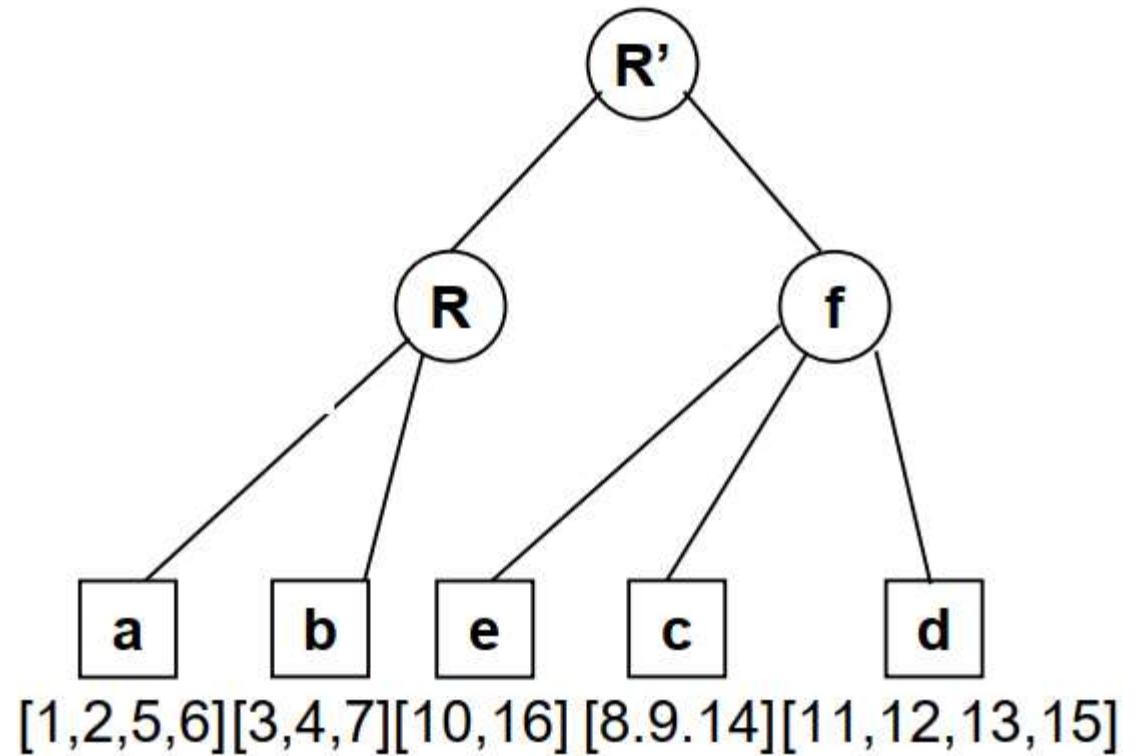
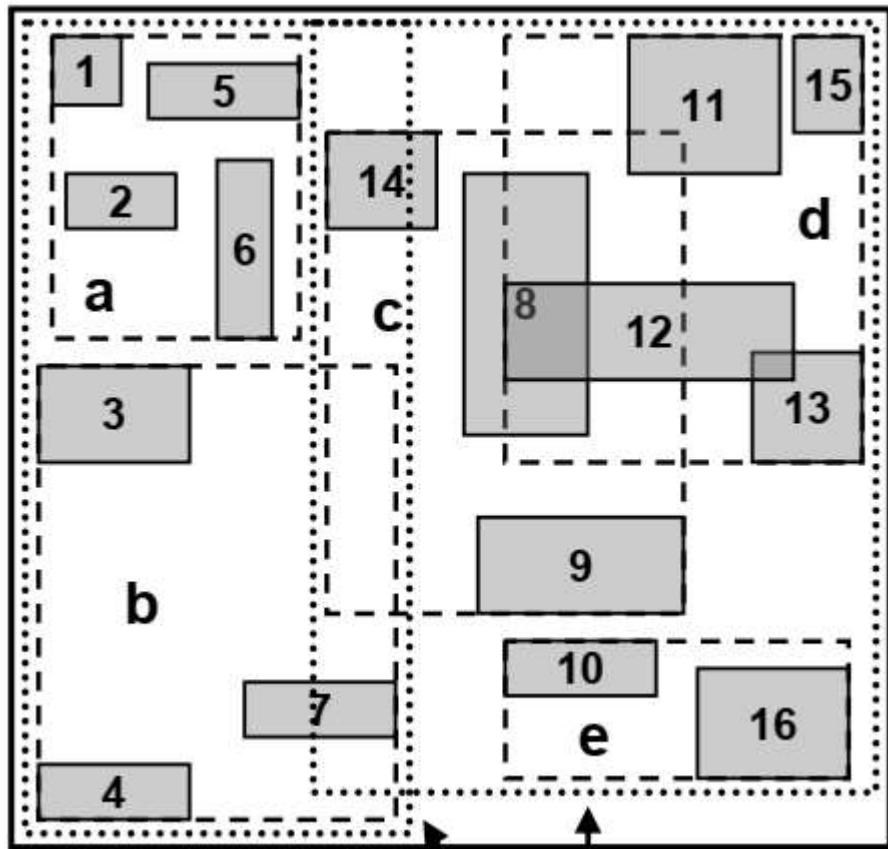
## Contoh 2

- Tentukan lokasi object baru (dengan nilai 16) berikut ini dengan  $M=4$



## Contoh 2

- Tentukan lokasi object baru (dengan nilai 16) berikut ini dengan  $M=4$



# Pengodean Data

FQJIXDYM EBSLJBWXDU NL  
GFBVWLCTFP0IZQA YWKA I  
MYVLOYFJRCVUNIJP NJKI  
WZUXQURAXIOMVMVOFTDC  
VYCDBYCJKMOPXEFRSPC0B  
KBJIMUKIVAGVGRQTEZK  
ZKYBSECNIMDGOMFVETOE  
CI PUYKFIXOCTFZCKJEAR  
YKRVECGIOCRLXCLKLCTR D  
QLGZRWF PF0E IYFVRMZHX  
RPZYDUIVTEAXLJWSIRUG  
JLA VMPLOTYCKIBQYWYPK  
BPFRDJTVAQIFSTZVF MJC  
SYECVINGFBRN YUCBSNTD  
CFIBRMSZJEDXRWT KADFE



# Encoding

---

- Sebuah pesan terdiri atas “karakter” yang harus direpresentasikan (di-*encode*) sebagai rangkaian bit
- Kode tsb perlu distandardkan agar pesan disajikan dengan semestinya
- Beberapa kode di sistem komputer:
  - ASCII
    - 7 bit per karakter: 128 *code point*
    - Dikembangkan ke 8 bit per karakter: Windows-1252, ISO-8859-1, dll
  - Unicode
    - *Internationalization (i18n) & localization (l10n)*

# ASCII

B <sub>7</sub> B <sub>6</sub> B <sub>5</sub>					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
					0	1	2	3	4	5	6	7
					NUL	DLE	SP	0	@	P	`	p
b <sub>4</sub>	b <sub>3</sub>	b <sub>2</sub>	b <sub>1</sub>		SOH	DC1	!	I	A	Q	a	q
0	0	0	1	1	STX	DC2	"	2	B	R	b	r
0	0	1	0	2	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	\$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	l
1	1	0	1	13	CR	GS	-	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	—	o	DEL

# 8-bit extensions

- Windows-1252
- ISO-8859-1

N\_U\_L\_S\_O\_H\_S\_T\_X\_E\_T\_X\_E\_O\_T\_E\_N\_Q\_A\_C\_K\_B\_E\_L\_B\_S\_H\_T\_L\_F\_V\_T\_F\_C\_R\_S\_S\_I\_D\_L\_E\_D\_C\_1\_D\_C\_2\_D\_C\_3\_D\_C\_4\_N\_A\_K\_S\_Y\_N\_E\_T\_B\_C\_A\_N\_E\_M\_S\_U\_B\_E\_S\_C\_F\_S\_G\_S\_R\_S\_U\_S

!"#\$%&' ( )\*+, - ./0123456789: ;<=>?  
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^\_`  
abcdefghijklmnopqrstuvwxyz{| }~<sub>D\_E\_L</sub>  
€・, f„…†‡^‰Š<Œ・Ž・‘’”•—~™š>œ・žŸ  
¡¢£¤¥¡§”©¤«¬¬®¬°±²³’µ¶·, ¹º»¹¹³²¼¿  
ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÔÖ×ØÙÚÛÜÝÞß  
àáâãäåæçèéêëìíîïðñòóôôö÷øùúûüýþÿ

N\_U\_L\_S\_O\_H\_S\_T\_X\_E\_T\_X\_E\_O\_T\_E\_N\_Q\_A\_C\_K\_B\_E\_L\_B\_S\_H\_T\_L\_F\_V\_T\_F\_C\_R\_S\_S\_I\_D\_L\_E\_D\_C\_1\_D\_C\_2\_D\_C\_3\_D\_C\_4\_N\_A\_K\_S\_Y\_N\_E\_T\_B\_C\_A\_N\_E\_M\_S\_U\_B\_E\_S\_C\_F\_S\_G\_S\_R\_S\_U\_S

!"#\$%&' ( )\*+, - ./0123456789: ;<=>?  
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^\_`  
abcdefghijklmnopqrstuvwxyz{| }~<sub>D\_E\_L</sub>  
.....

¡¢£¤¥¡§”©¤«¬¬®¬°±²³’µ¶·, ¹º»¹¹³²¼¿  
ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÔÖ×ØÙÚÛÜÝÞß  
àáâãäåæçèéêëìíîïðñòóôôö÷øùúûüýþÿ



Code range (hexadecimal)	UTF-8	UTF-16	UTF-32
000000 – 00007F	1		
000080 – 00009F			
0000A0 – 0003FF	2	2	
000400 – 0007FF		2	
000800 – 003FFF			4
004000 – 00FFFF	3		
010000 – 03FFFF			
040000 – 10FFFF	4	4	

Code point ↔ UTF-8 conversion					
First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
U+0000	U+007F	0xxxxxxxx			
U+0080	U+07FF	110xxxxx	10xxxxxx		
U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
U+10000	[b]U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

# The Unicode Standard

- 1 112 064 *code points*
- Beberapa pilihan *encoding* untuk merepresentasikan sebuah *code point*:
  - UTF-8: 1-4 byte
  - UTF-16: 2 atau 4 byte
  - UTF-32: 4 byte
- Termasuk emoji
  - 🌿 (U+1F64F)



# Apa makna ?

---

- Unicode memberikan *code point*, bukan *glyph* (simbol/gambar), untuk tiap karakter
  - Deskripsi abstrak
- Visualisasi diserahkan ke perangkat lunak yang mengimplementasikan
- Bisa menimbulkan interpretasi yang beragam
- U+1F64F = *PERSON WITH FOLDED HANDS*
  - *Praying? High-five? Thank you? Please?*

# *Printable vs Non-Printable (Text vs Binary)*

- Printable character adalah karakter yang ditujukan untuk ditulis/dicetak/ditampilkan untuk dibaca manusia
  - Sebuah *text file* berisikan hanya *printable characters*
  - Fail yang juga bisa berisikan *non-printable characters* disebut *binary file*
  - Contoh: di Notepad kita bekerja dengan *text file*, sedangkan hasil kompilasi merupakan *binary file*
  - *Binary file* lebih cocok untuk dilihat menggunakan *hex editor*

Address	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	Dump
00000000	4d	5a	90	00	03	00	00	00	04	00	00	00	ff	ff	00	00	MZ.....ÿ..
00000010	b8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00	.....@.....
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....ñ..
00000040	0e	1f	ba	0e	00	b4	09	cd	21	b8	01	4c	cd	21	54	68	..°...!.f!.,LÍ!Th
00000050	69	73	20	70	72	6f	67	72	61	6d	20	63	61	6e	6e	6f	is program canno
00000060	74	20	62	65	20	72	75	6e	20	69	6e	20	44	4f	53	20	t be run in DOS
00000070	6d	6f	64	65	2e	0d	0d	0a	24	00	00	00	00	00	00	00	mode....\$.....
00000080	c0	5a	eb	a9	84	3b	85	fa	84	3b	85	fa	84	3b	85	fa	ÀZëÓ;..û.;..û.;..û
00000090	8d	43	16	fa	86	3a	85	fa	90	50	81	fb	9b	3b	85	fa	.C.út;..û.P.û;..û
000000a0	90	50	86	fb	87	3b	85	fa	84	3b	84	fa	37	32	85	fa	.Ptút;..û.;..û72..û
000000b0	90	50	84	fb	9b	3b	85	fa	90	50	88	fb	f7	3a	85	fa	.P.ûú;..û.P.ûú;..û

# Representasi Newline

---

- *Newline* direpresentasikan dengan cara berbeda di aneka platform
- *File Transfer Protocol* (FTP) menyediakan tipe data ASCII (TYPE A) untuk melakukan konversi otomatis
  - Mode ini akan merusak *binary file*

Operating system	Character encoding	Abbreviation	hex value	dec value	Escape sequence
Unix and Unix-like systems (Linux, macOS, FreeBSD, AIX, Xenix, etc.), Multics, BeOS, Amiga, RISC OS, and others <sup>[5]</sup>	ASCII	LF	0A	10	\n
Microsoft Windows, DOS (MS-DOS, PC DOS, etc.), Atari TOS, DEC TOPS-10, RT-11, CP/M, MP/M, OS/2, Symbian OS, Palm OS, Amstrad CPC, and most other early non-Unix and non-IBM operating systems		CR LF	0D 0A	13 10	\r\n
Commodore 8-bit machines (C64, C128), Acorn BBC, ZX Spectrum, TRS-80, Apple II series, Oberon, the classic Mac OS, MIT Lisp Machine and OS-9		CR	0D	13	\r
QNX pre-POSIX implementation (version < 4)		RS	1E	30	\036
Acorn BBC <sup>[6]</sup> and RISC OS spooled text output <sup>[7]</sup>		LF CR	0A 0D	10 13	\n\r
Atari 8-bit machines	ATASCII		9B	155	
IBM mainframe systems, including z/OS (OS/390) and IBM i (OS/400)	EBCDIC	NL	15	21	\025
ZX80 and ZX81 (Home computers from Sinclair Research Ltd)	used a specific non-ASCII character set	NEWLINE	76	118	

# BASE64 *Encoding*

- Protokol seperti *Simple Mail Transfer Protocol* (SMTP) yang digunakan untuk pengiriman email dibuat berbasis teks dengan ASCII
- Bagaimana *binary file* bisa dikirimkan via email?
- BASE64 merupakan skema *binary-to-text encoding* yang merepresentasikan data 24 bit sebagai empat *printable characters* tertentu
  - 64 karakter untuk data: A-Z, a-z, 0-9, +, /
  - 1 karakter untuk *padding*: =
- Bukan enkripsi, bukan steganografi

# Alfabet Base64

<i>Index</i>	<i>Binary</i>	<i>Char</i>									
0	000000	A	16	010000	Q	32	100000	g	48	110000	w
1	000001	B	17	010001	R	33	100001	h	49	110001	x
2	000010	C	18	010010	S	34	100010	i	50	110010	y
3	000011	D	19	010011	T	35	100011	j	51	110011	z
4	000100	E	20	010100	U	36	100100	k	52	110100	0
5	000101	F	21	010101	V	37	100101	l	53	110101	1
6	000110	G	22	010110	W	38	100110	m	54	110110	2
7	000111	H	23	010111	X	39	100111	n	55	110111	3
8	001000	I	24	011000	Y	40	101000	o	56	111000	4
9	001001	J	25	011001	Z	41	101001	p	57	111001	5
10	001010	K	26	011010	a	42	101010	q	58	111010	6
11	001011	L	27	011011	b	43	101011	r	59	111011	7
12	001100	M	28	011100	c	44	101100	s	60	111100	8
13	001101	N	29	011101	d	45	101101	t	61	111101	9
14	001110	O	30	011110	e	46	101110	u	62	111110	+
15	001111	P	31	011111	f	47	101111	v	63	111111	/

# *Encoding dan Decoding dengan BASE64*

---

- Teks asli: **Budi**
  - Kode karakter desimal: 66 117 100 105
  - Dalam biner 8-bit: 01000010 01110101 01100100 01101001
  - Kelompokkan per 6 bit: 010000 100111 010101 100100 011010 01
  - Genapkan 6 bit: 010000 100111 010101 100100 011010 01**0000**
  - Padankan ke karakter Base64: **QnVkaQ**
  - *Padding* agar kelipatan 4 karakter: **QnVkaQ==**
- *Decoding* dilakukan dengan membuang *padding* lalu mengubah karakter Base64 ke biner lalu mengambil karakter per 8 bit

# Penentuan Panjang Kode

- Misalkan kita memiliki pesan yang hanya menggunakan simbol **A, B, C, D, dan E**
  - Berapa bit yang diperlukan untuk *encoding*?
    - Setidaknya 3 bit (Kenapa?)
  - Berapa bit yang diperlukan untuk pesan berikut: **DDDAAAACCAAAABEEEE?**
    - Setidaknya  $16 * 3 = 48$  bit
- Dengan skema *encoding* yang lazim digunakan, satu karakter minimal 8 bit sehingga pesan tersebut memerlukan 128 bit
- Makin panjang, makin besar ruang penyimpanan yang dibutuhkan

# *Run Length Encoding (RLE)*

---

- Metode kompresi *lossless* yang merepresentasikan sekuens data sebagai data dan ulangannya
- Sesuai untuk data yang mengandung banyak perulangan misalnya citra yang terdiri atas blok-blok warna yang sama
- Contoh:
  - Teks asli: **DDDAAAACCAAAABEEEE**
  - RLE: **3D3A2C3A1B4E**
- Beberapa format yang menggunakan konsep RLE: TGA, PCX
- Apa kekurangannya?



# Kekurangan Kode dengan Panjang Tetap

---

- Pemborosan ruang karena banyak fail yang hanya menggunakan sebagian karakter yang ada
  - Terlebih jika menggunakan Unicode
  - Frekuensi kemunculan huruf tidak seragam
  - Bandingkan frekuensi huruf ‘q’ dan huruf ‘a’
- **Solusi potensial:** gunakan kode dengan panjang bervariasi sesuai frekuensi penggunaan
  - Karakter yang lebih sering digunakan menggunakan kode yang lebih singkat

# *Variable-length codes*

- Karakter yang lebih sering digunakan menggunakan kode yang lebih singkat agar secara rata-rata, panjang pesan menjadi lebih singkat
- **Masalah potensial:** bagaimana cara memisahkan karakter?
  - Tidak masalah ketika panjang kode sudah tertentu

A = 00

B = 01

C = 10

D = 11

0010110111001111111111

A C D B A D D D D D

# *Prefix property*

- Sebuah pengodean memiliki ***prefix property*** jika tidak ada kode karakter yang merupakan prefix untuk karakter lain
- Contoh:

<b>Simbol</b>	<b>Kode</b>
P	000
Q	11
R	01
S	001
T	10

01001101100010

R S T Q P T

- 000 bukan prefiks 11, 01, 001, atau 10
- 11 bukan prefiks 000, 01, 001, atau 10 dst

Contoh kode  
tanpa *prefix*  
*property*

---

- Pengodean berikut **tidak** memiliki *prefix property*
- Pola **1110** bisa didekodenkan menjadi **QQQP**, **QTP**, **QQS**, atau **TS**

Simbol	Kode
P	0
Q	1
R	01
S	10
T	11

# Masalah

- Rancang sebuah pengodean untuk pesan **DDDAAAACCAAAABEEEE** dengan karakter *variable-length prefix-free*
- Solusi potensial:
  - **A** = 6, **B** = 1, **C** = 2, **D** = 3, **E** = 4
  - representasikan **A** dengan satu bit mis. 0
    - Berarti kode lainnya tidak boleh dimulai dengan 0
  - representasikan **E** dengan dua bit 10
    - Berarti kode lainnya tidak boleh dimulai dengan 0 atau 10
  - representasikan **D** dengan 110
  - representasikan **C** dengan 1110
  - representasikan **B** dengan 11110

# Alternatif 1

**DDDAAAACCAAAABEEEE**

Simbol	Kode
A	0
B	11110
C	1110
D	110
E	10

**110110110000111011100001111010101010**

36 bit

## Alternatif 2

**DDDAAAACCAAAABEEEE**

Simbol	Kode
A	01
B	000
C	001
D	10
E	11

**10101001010100100101010100011111111**

35 bit

# Gunakan kode yang mana?

- Apakah ada mekanisme baku agar pengodean efisien?

- Ada!



# Huffman coding tree

---

- *Binary tree*
  - *leaf*: sebuah simbol (karakter)
  - Label *edge* ke *left child*: 0
  - Label *edge* ke *right child*: 1
- Kode tiap simbol diperoleh dengan mengikuti *path* dari *root* ke *leaf*
- *Prefix property*
  - Suatu *leaf* tidak bisa muncul di *path* ke *leaf* lain

# Membangun *Huffman tree*



Hitung frekuensi tiap simbol dalam pesan



Mulai dengan *forest* dengan *single node trees*

Masing-masing node berisikan simbol dan frekuensi



Lakukan secara rekursif

Pilih dua *tree* dengan frekuensi terkecil di *root*  
Hasilkan *tree* baru dengan dua *tree* tsb dan simpan jumlah frekuensi di *root*



Selesai ketika tersisa satu *tree*

Merupakan *Huffman coding tree*

# Contoh

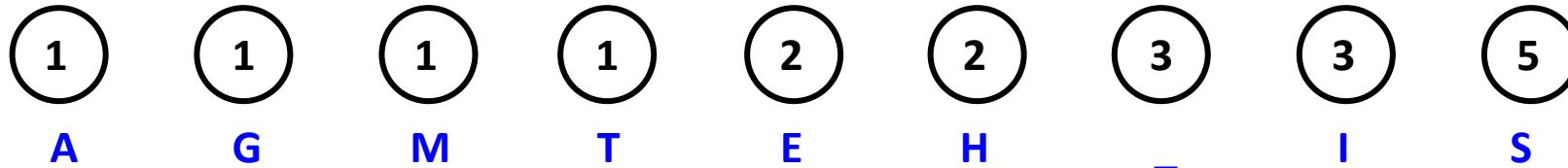
- Bangun *Huffman coding tree* untuk pesan berikut:

*This is his message*

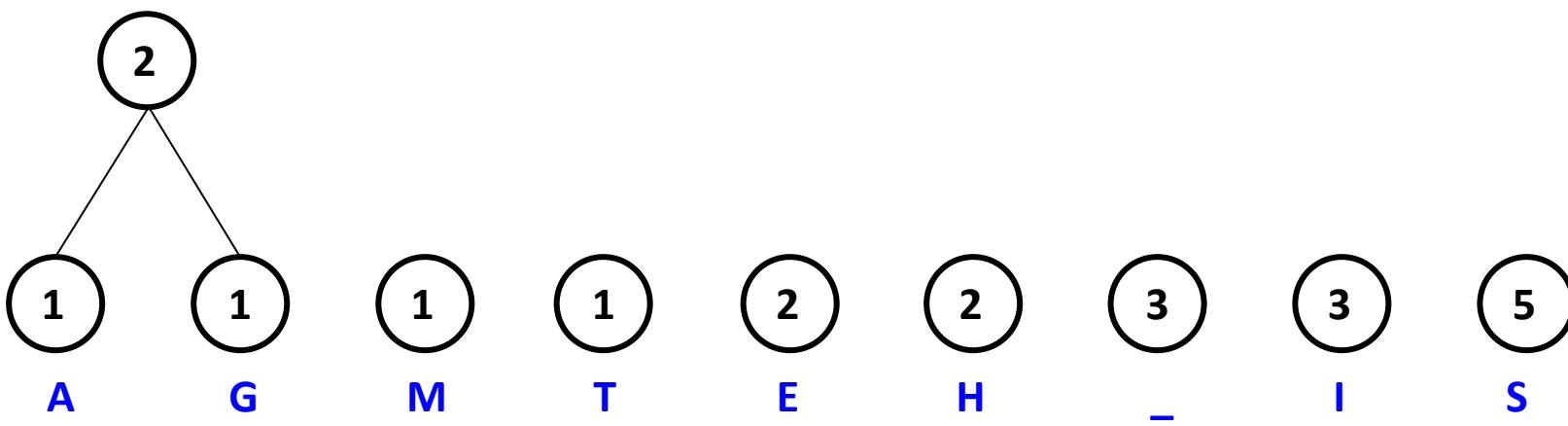
- Frekuensi karakter

A	G	M	T	E	H	-	I	S
1	1	1	1	2	2	3	3	5

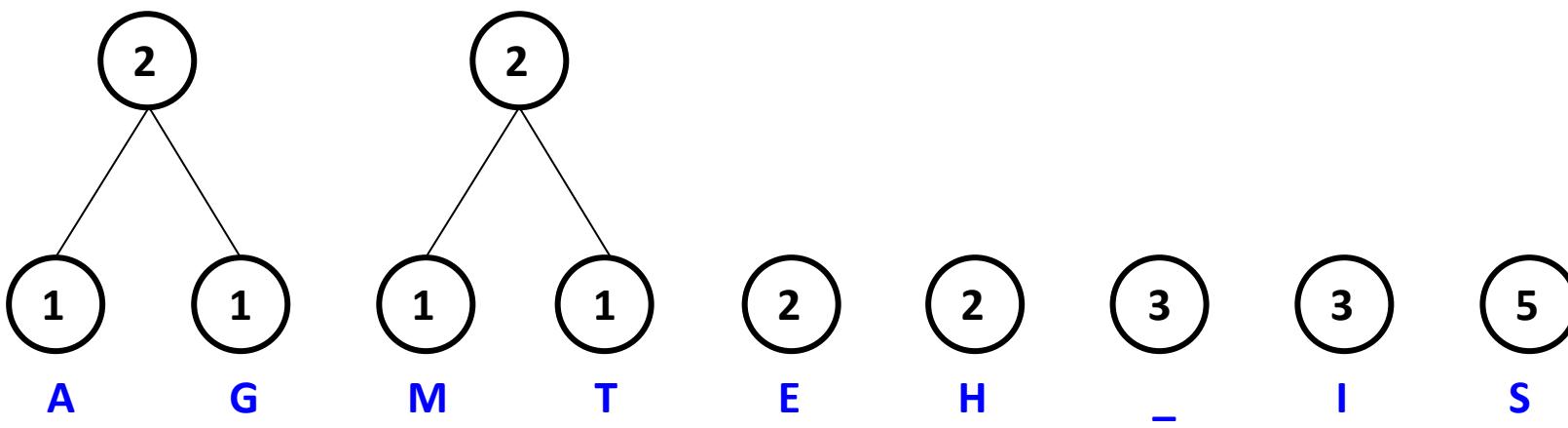
- Mulai dengan *single node trees*



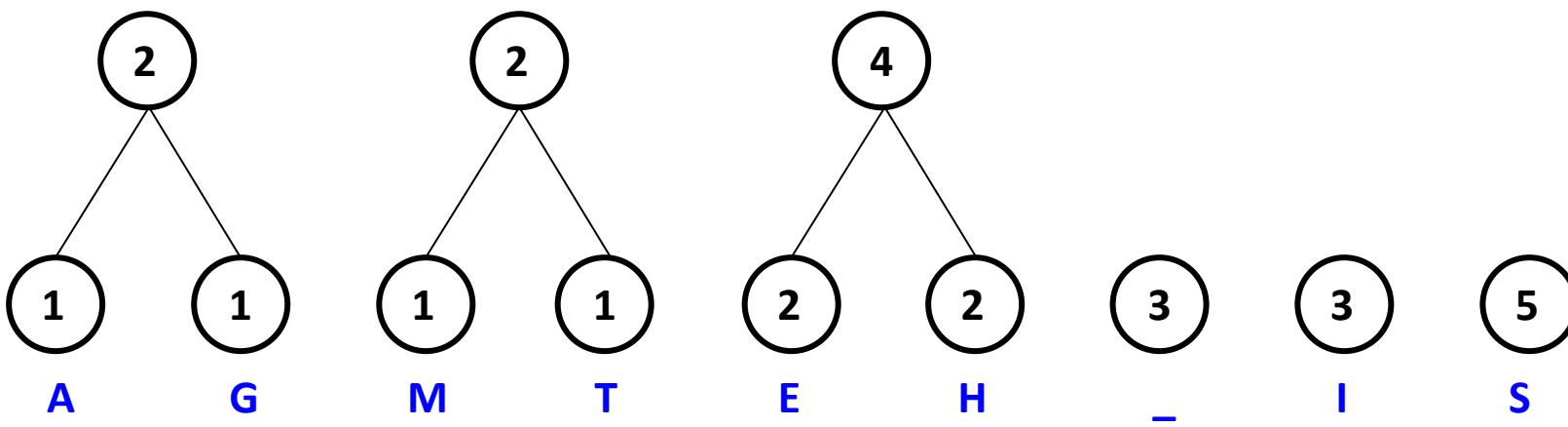
# Langkah 1



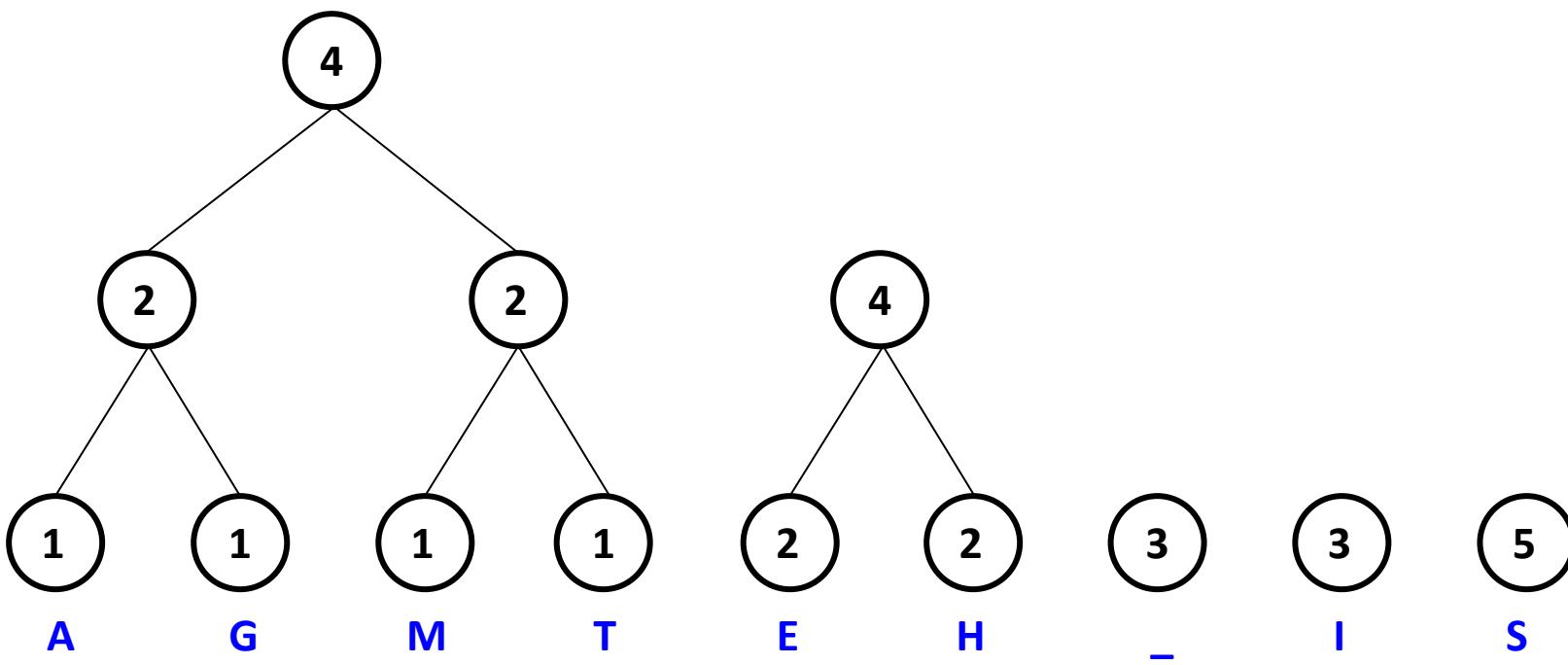
# Langkah 2



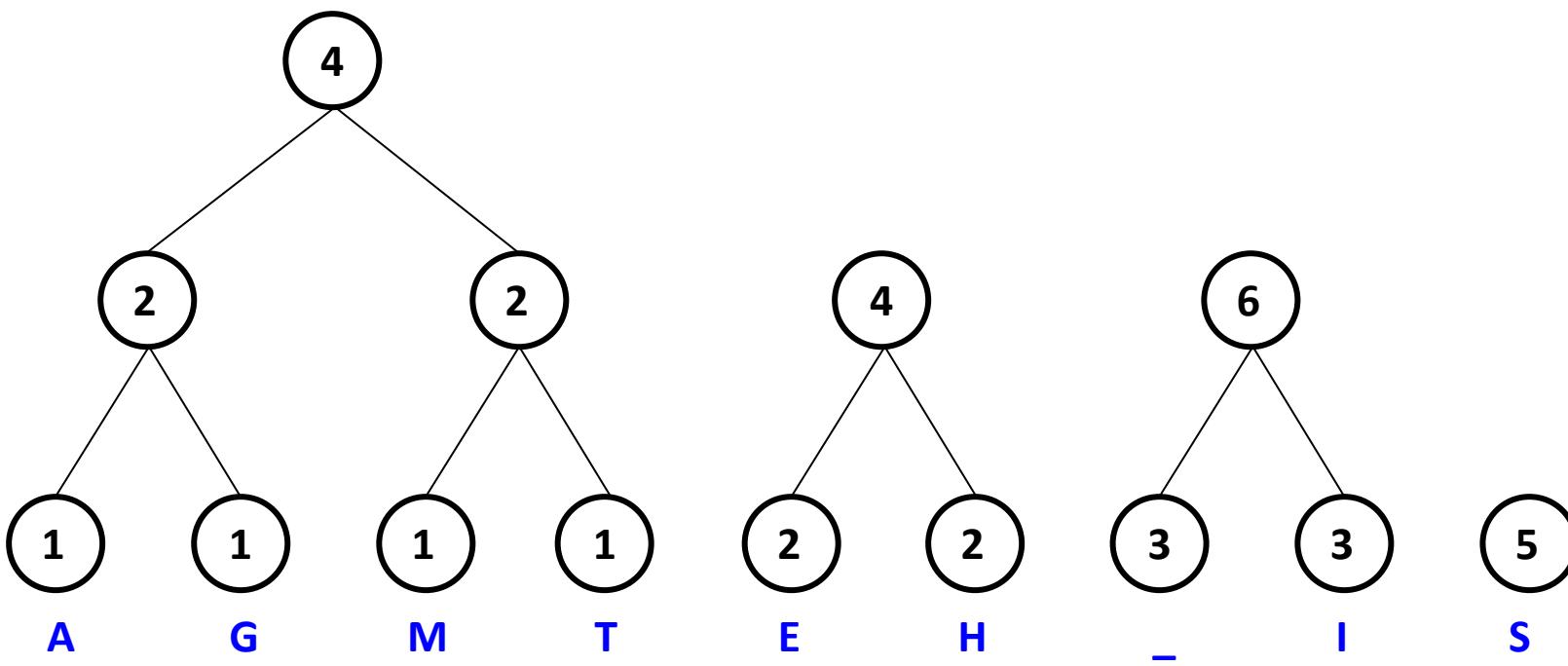
# Langkah 3



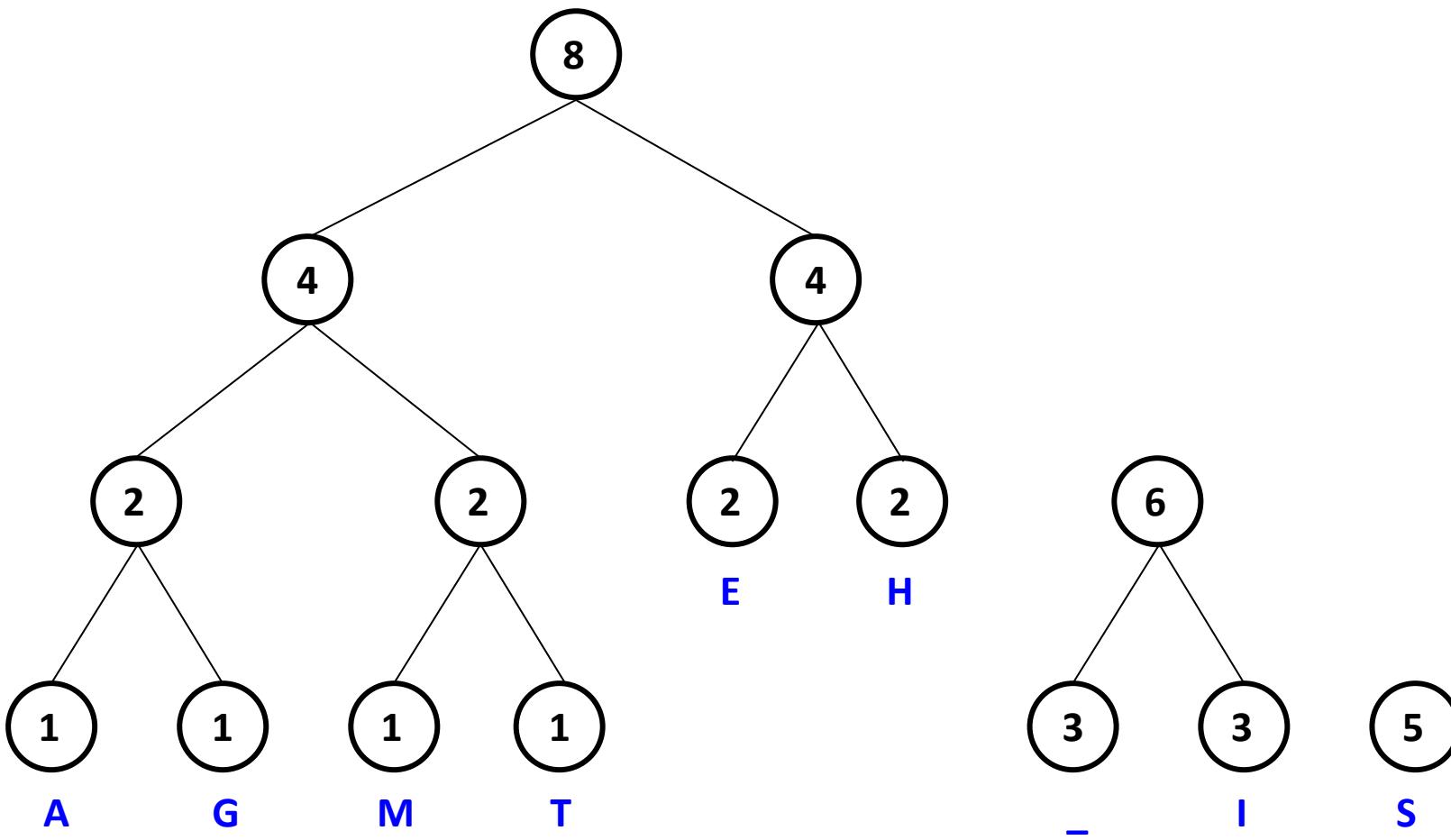
# Langkah 4



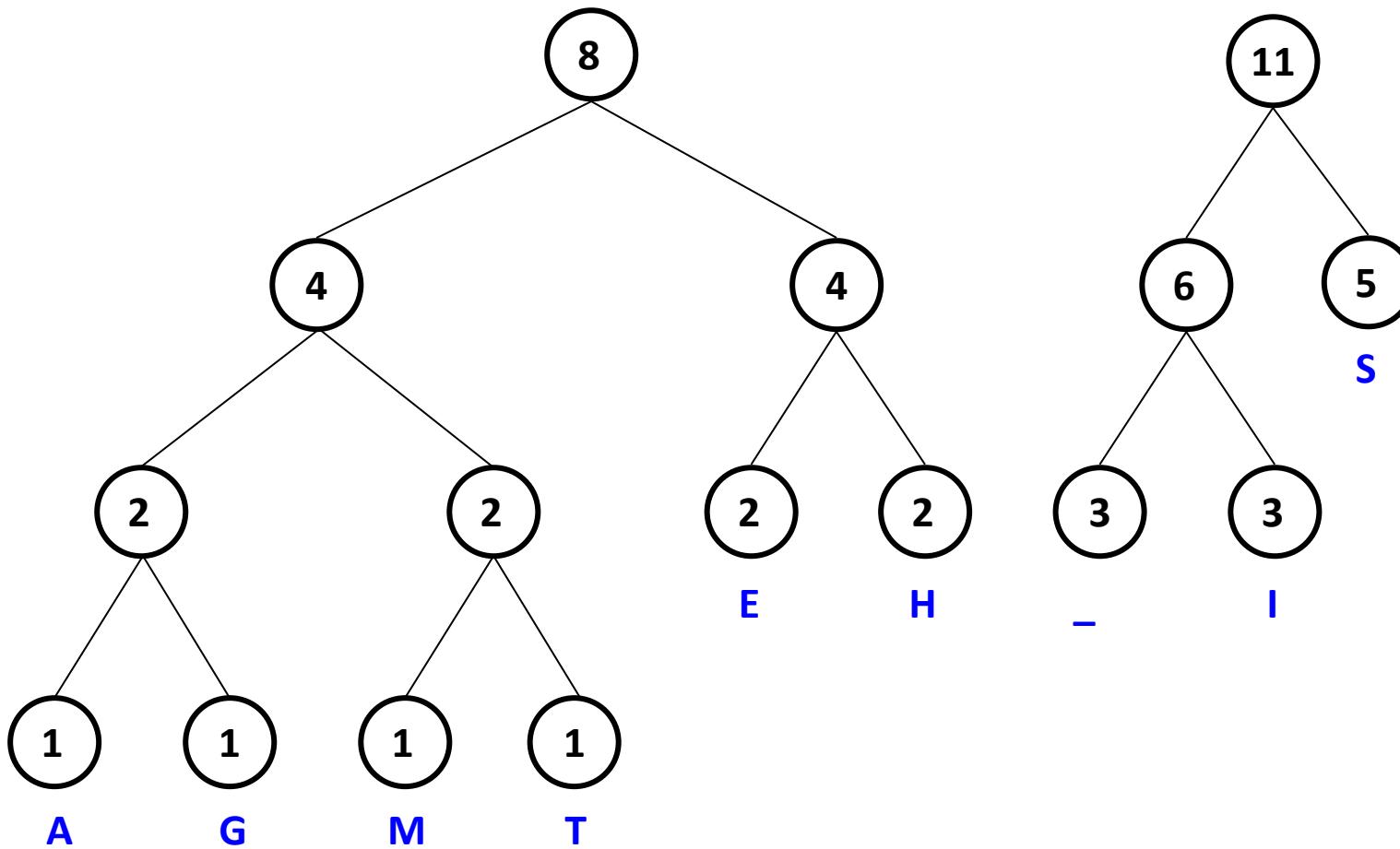
# Langkah 5



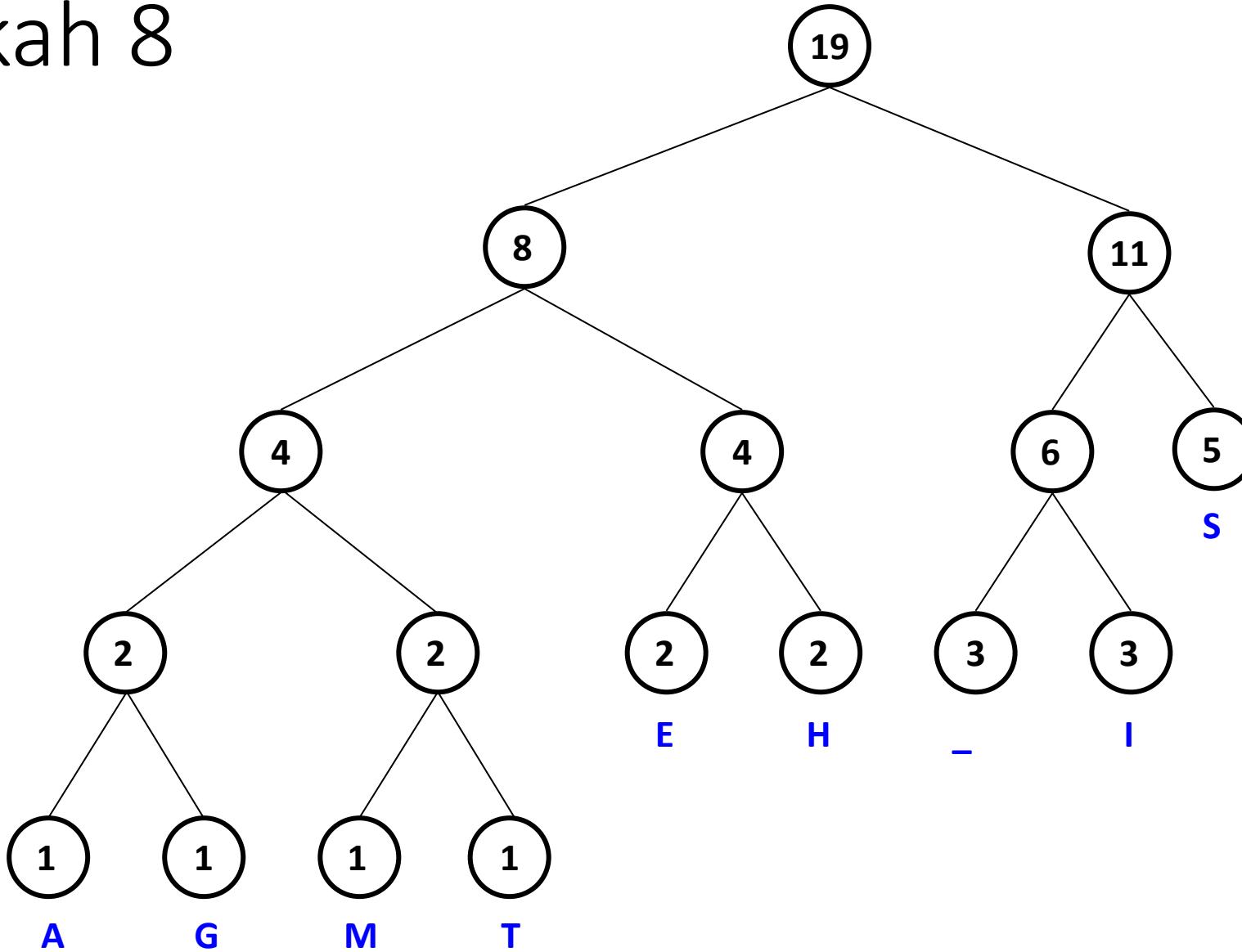
# Langkah 6



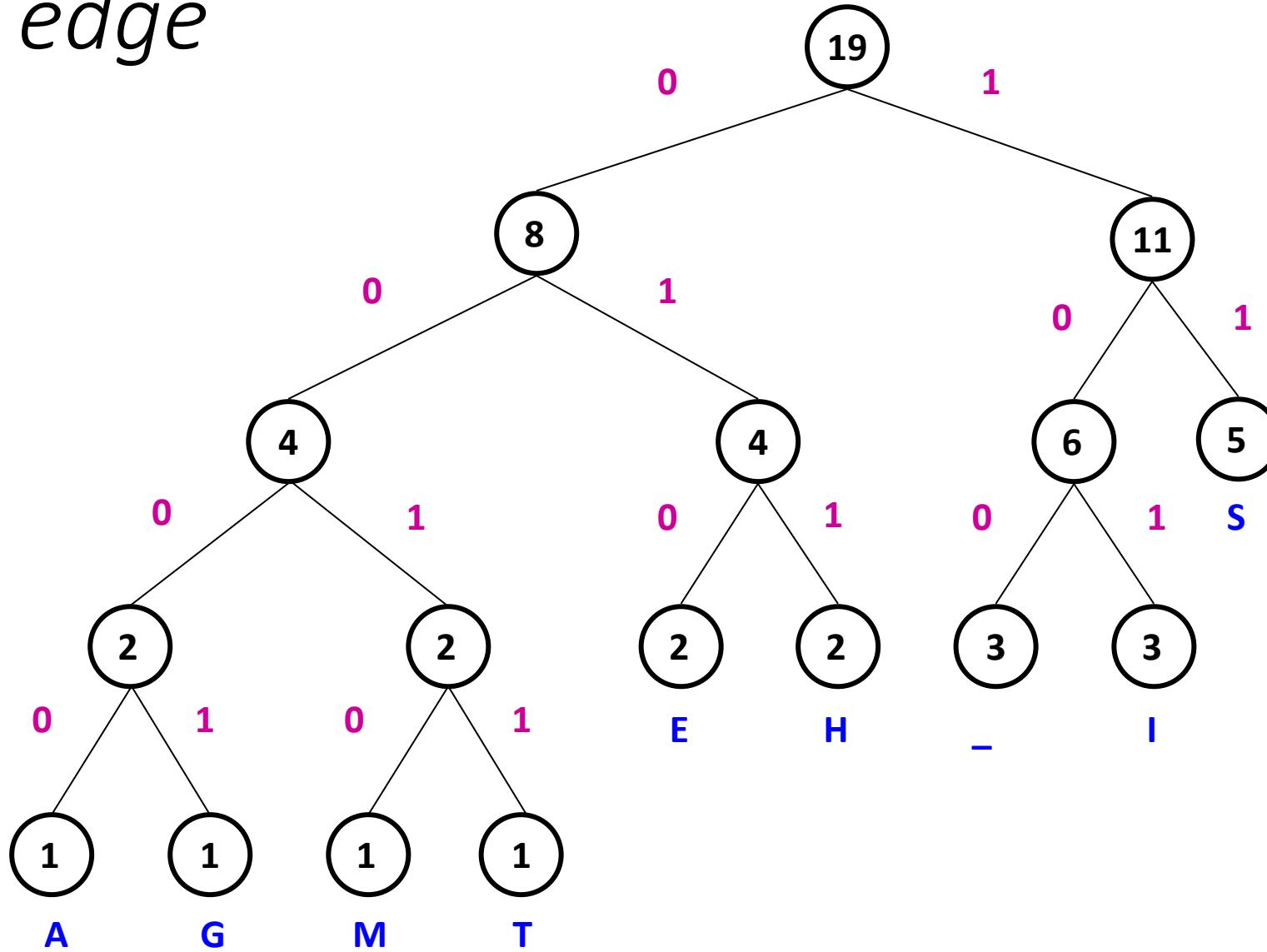
# Langkah 7



# Langkah 8



*Label edge*



# *Huffman code & hasil encoding*

*This is his message*

S	11
E	010
H	011
-	100
I	101
A	0000
G	0001
M	0010
T	0011

00110111011110010111100011101111000010010111100000001010



Terima kasih!

---