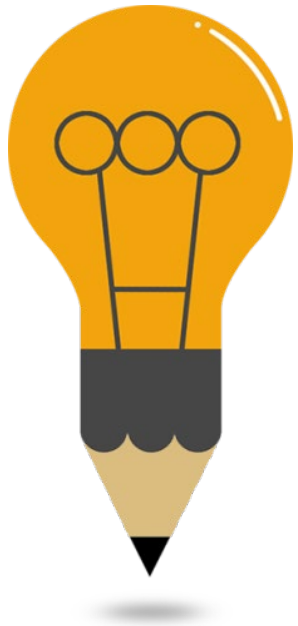
STRUKTUR DATA

Pertemuan 9

(Ratih Ngestrini)

Agenda Pertemuan



1. Tree (M-way, B-tree, AVL)

Multi-way (M-way) Search Tree

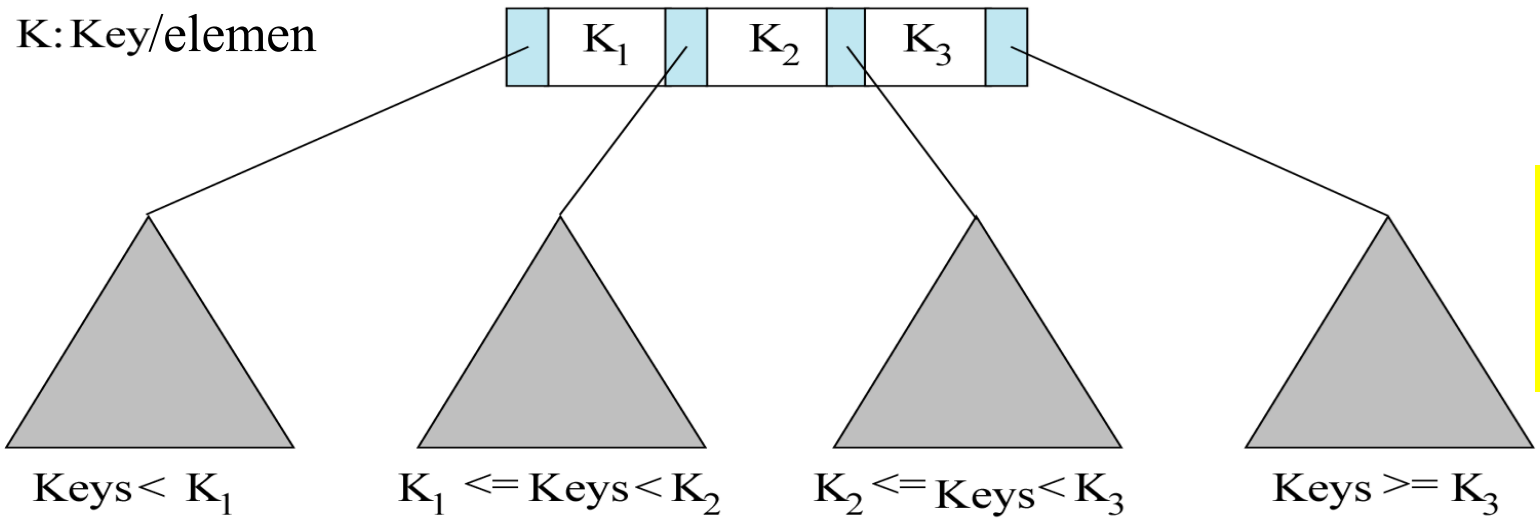
- Kita sudah mempelajari:
  - **Tree** secara general, satu node bisa memiliki lebih dari satu node anak (*multiple child node*)
  - **Search Tree**: ada hubungan antar elemennya (subtree kiri lebih kecil dan subtree kanan lebih besar). Strukturnya *binary (two-way)* atau *non binary (multi-way)*
  - **Binary Search Tree**: satu node maksimum memiliki dua node anak

- **Multi-way (M-way) Search Tree:** setiap node dapat memiliki lebih dari dua node anak dan antar elemennya mempunyai hubungan yang spesifik

Multi-way (M-way) Search Tree

Multi-way tree of order *M* (atau ***M*-way tree**) mempunyai karakteristik:

- Setiap node pada tree bisa berisi lebih dari satu elemen
- ***M*** adalah ***order*** dari tree tersebut
- Dapat memiliki ***M*** node anak (lebih dari dua)
- Jika setiap node dapat memiliki ***M*** node anak, maka node tersebut berisi paling banyak (***M - 1***) elemen
- Posisi elemen – elemen tersebut mengikuti konsep search tree (node kiri lebih kecil, node kanan lebih besar)

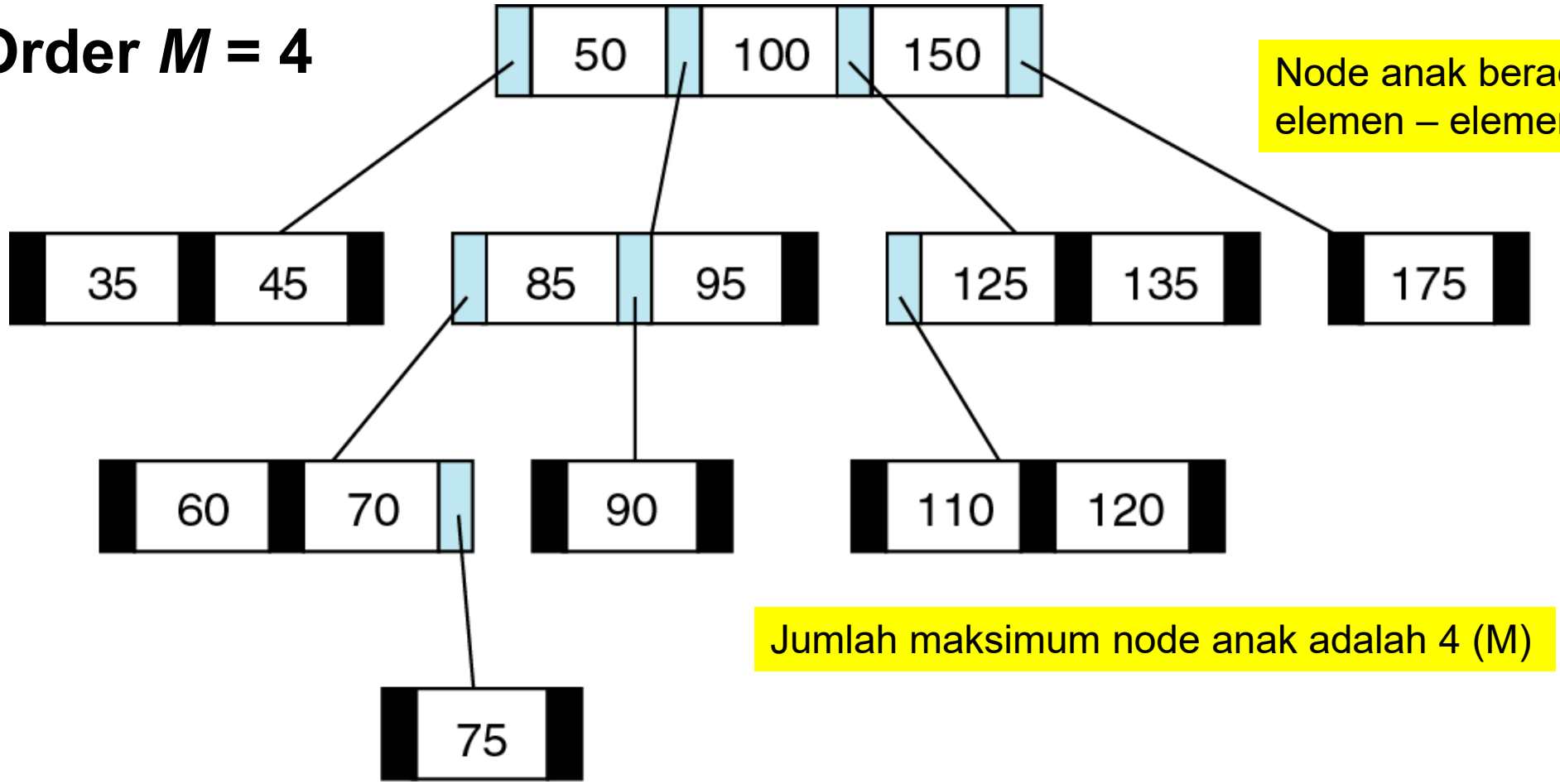


Gambar disamping asumsinya ada data yang sama di tree, tapi umunya tidak ada data yang sama di BST

M-way Tree (Contoh)

Elemen di dalam node maksimum adalah 3 (M-1)

Order *M* = 4

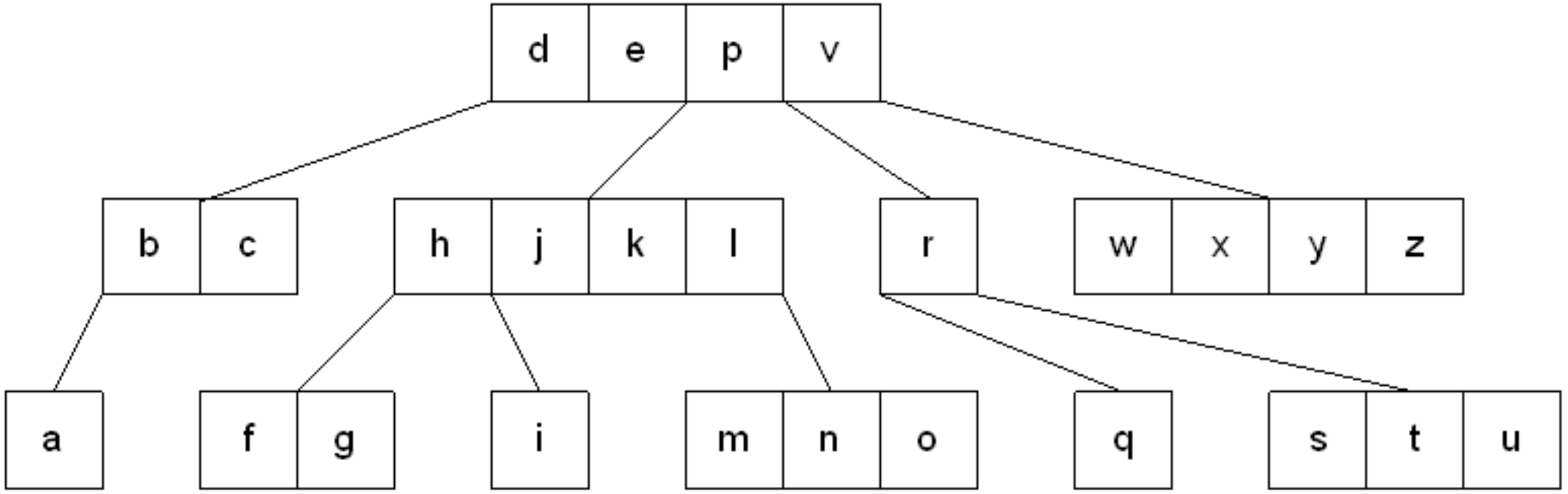


Node anak berada di antara elemen – elemen *parent*-nya

Jumlah maksimum node anak adalah 4 (M)

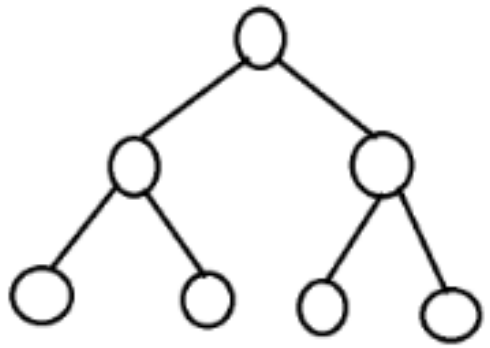
M-way Tree (Contoh)

M = 5

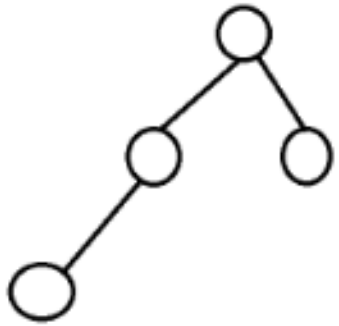


Fulll, Complete, dan *Balanced Binary Tree*

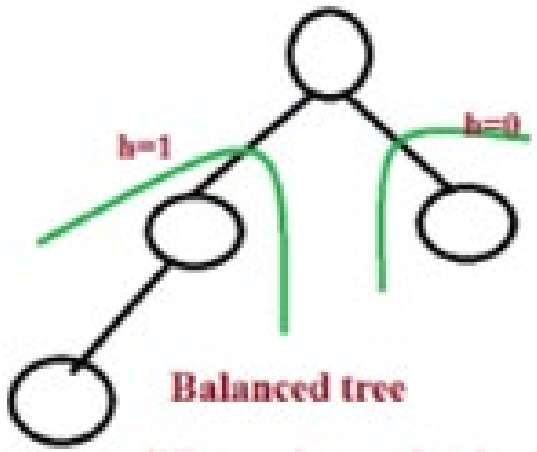
- **Full**: Jika setiap node mempunyai 2 node anak (sebuah binary tree dikatakan ***full*** jika semua node kecuali *leaf node* memiliki 2 node anak)
- **Complete**: Jika di setiap level "full" = berisi **2^n** node (n = level), kecuali level terakhir (jika terisi, maka left dulu yang terisi)
- **Balanced**: Jika tinggi/kedalaman subtree kiri dan kanan dari setiap node tidak lebih dari 1

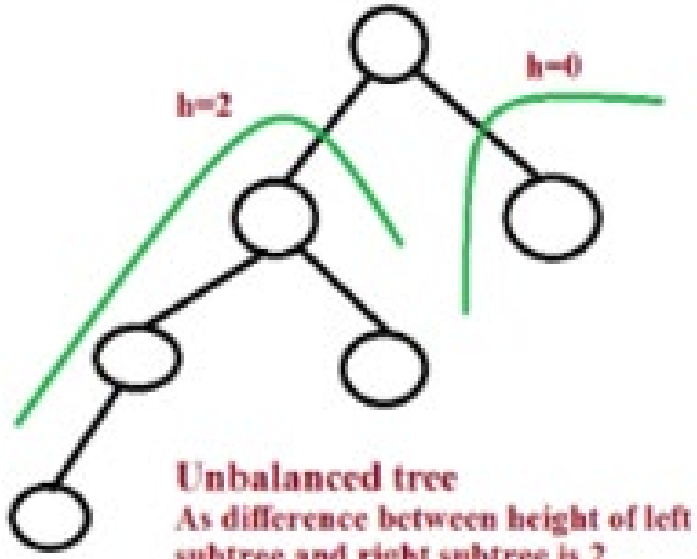


Full



Complete

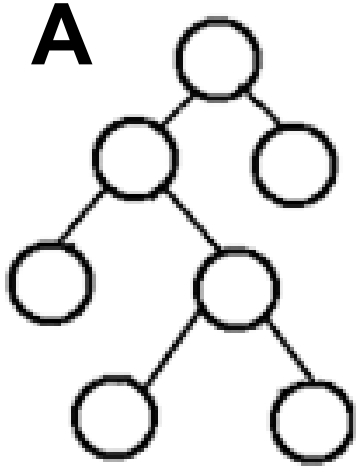




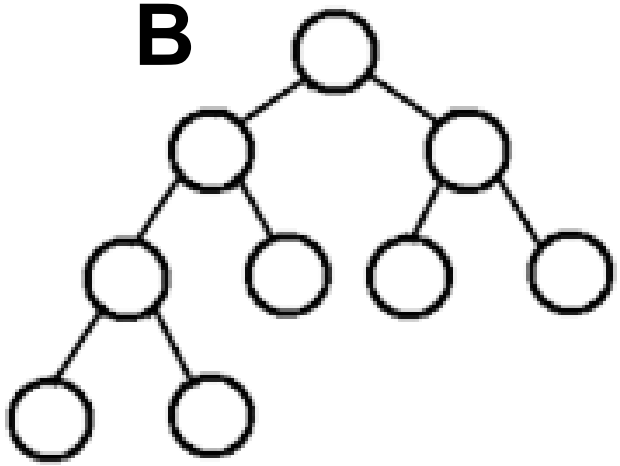
Balanced vs Unbalanced

Fulll, Complete, dan *Balanced*

A

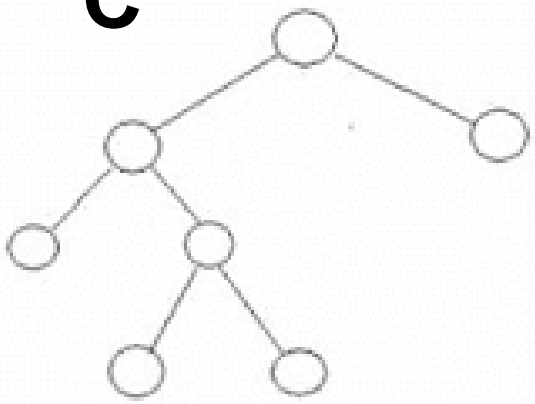


B

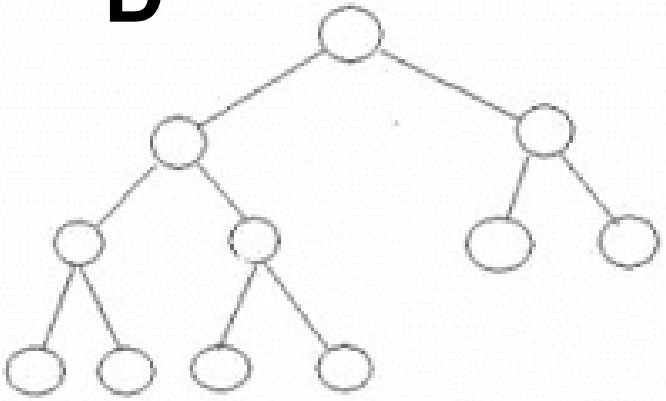


Bagaimana untuk pohon A, B, C, D, E?

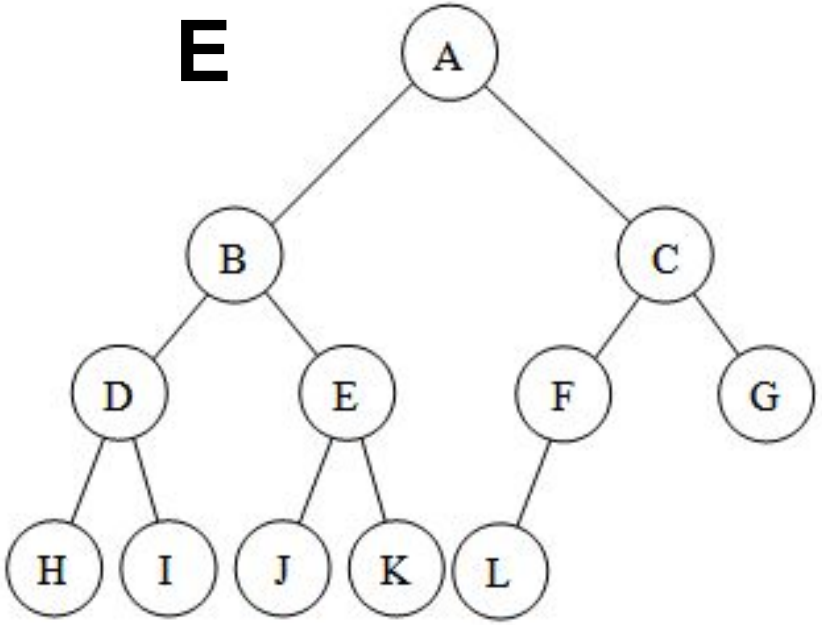
C



D



E



B-tree

B-tree merupakan ***Height Balanced Multi-way Search Tree*** (memiliki sifat – sifat multi-way search tree)

B-tree memiliki karakteristik:

- Setiap node memiliki maksimum *M* node anak → sifat *m-way search tree*
- Minimum node anak untuk: - Root → 2

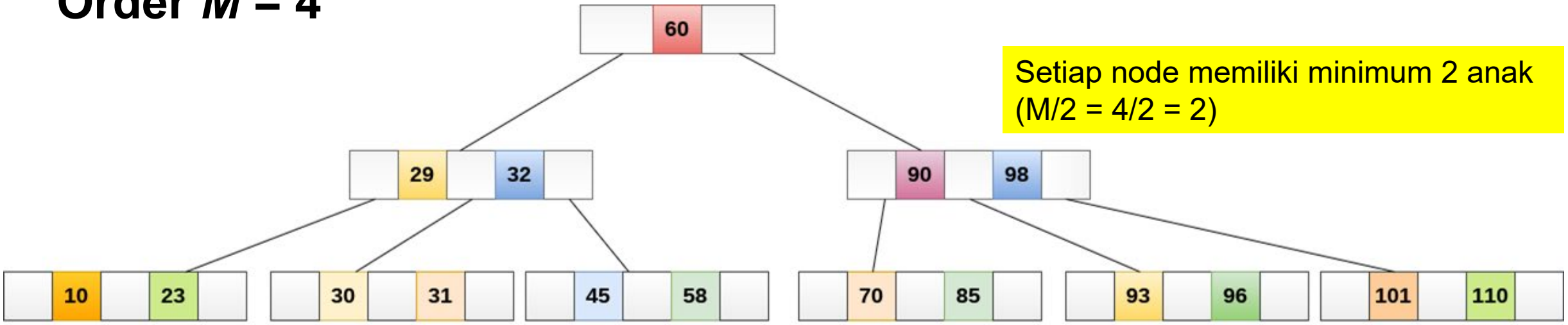
  - Internal node → $\left\lceil \frac{M}{2} \right\rceil$ (dibulatkan ke atas)
- Setiap node memiliki maksimum (M-1) elemen → sifat *m-way search tree*
- Minimum elemen pada: - Root → 1

  - Node lain → $\left\lceil \frac{M}{2} \right\rceil - 1$
- Setiap leaf mempunyai ketinggian (*height/depth/level*) yang sama → pembentukannya: *bottom up* (buat leaf node dulu baru ke atas)

B-tree (Contoh)

Order *M* = 4

ROOT memiliki minimum 2 anak



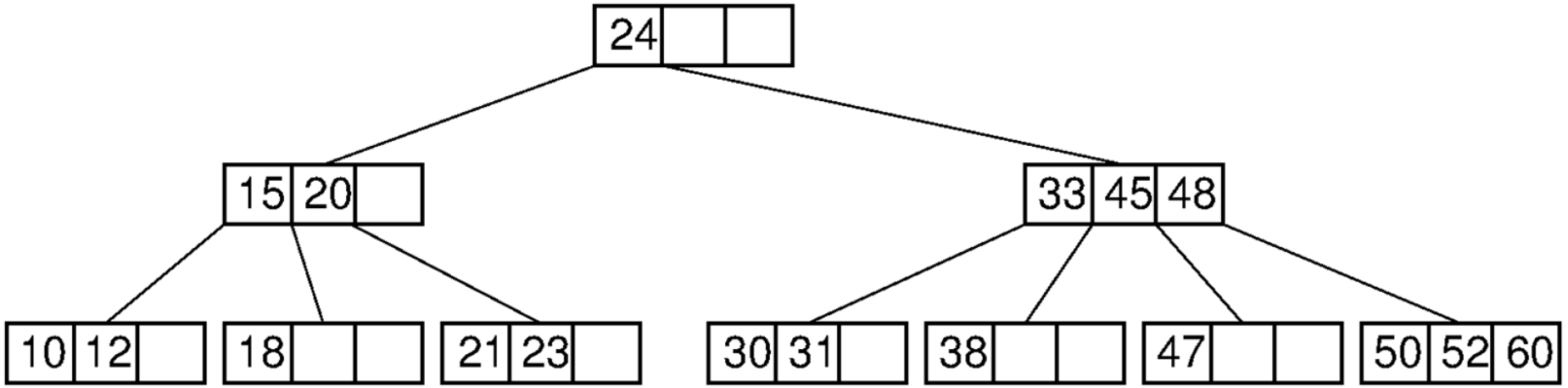
Setiap node memiliki minimum 2 anak
(M/2 = 4/2 = 2)

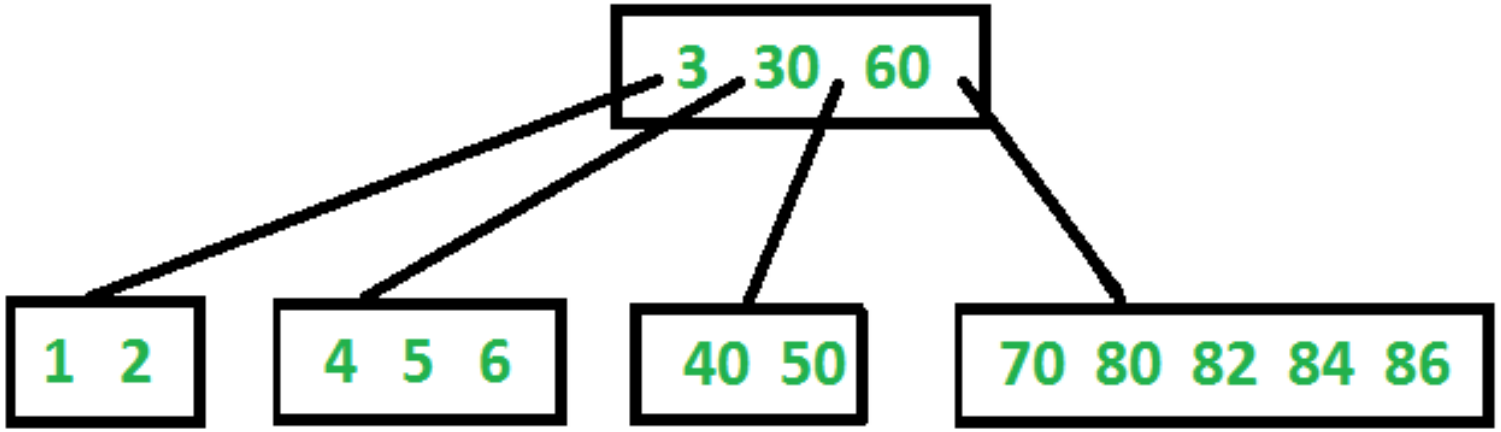
Setiap *leaf* mempunyai ketinggian (*height/depth*) yang sama

- Root node memiliki minimum 1 elemen, node lain juga minimum 1 elemen (M/2 -1) B-tree,
- Maksimum 3 elemen (M-1) sifat multi-way tree

B-tree (Contoh)

Order *M* = 4

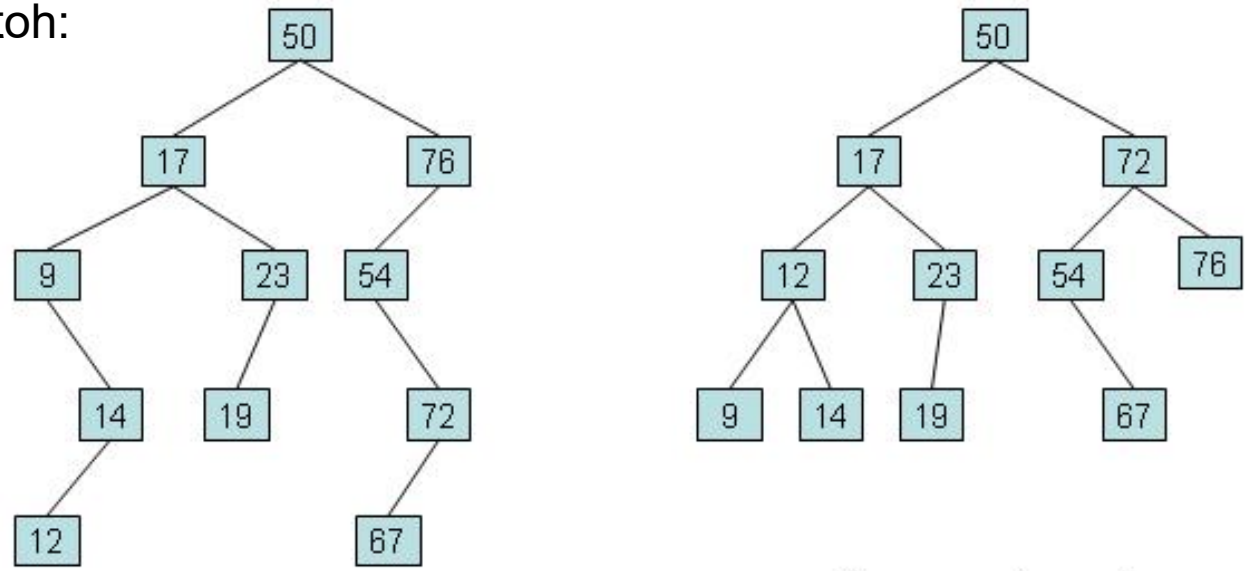




AVL Tree

- ***Height Balanced Binary Search Tree*** (penemu: **A**delson, **V**elski & **L**andis AVL Tree)
- Binary Search Tree digunakan dengan tujuan untuk mempercepat pencarian data
- Jika tidak *balanced*, maka waktu pencarian lebih lama (lihat fungsi search node di Binary Search Tree di slide sebelumnya bahwa proses pencarian akan iterasi dari root ke bawah sampai node ditemukan)

Contoh:



An unbalanced tree

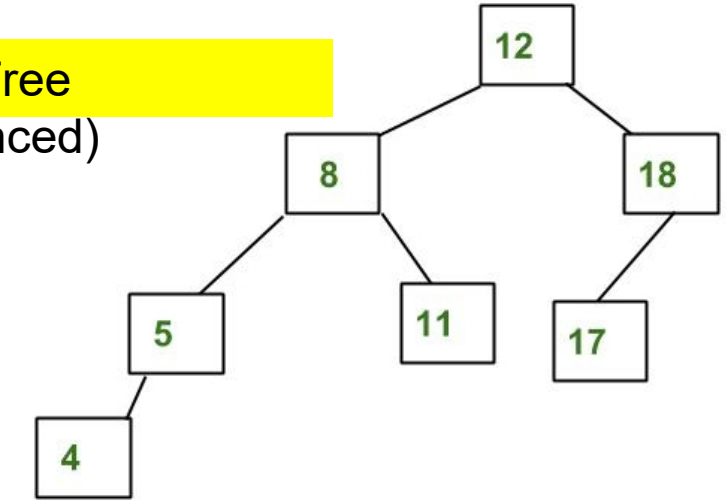
The same tree after being height-balanced

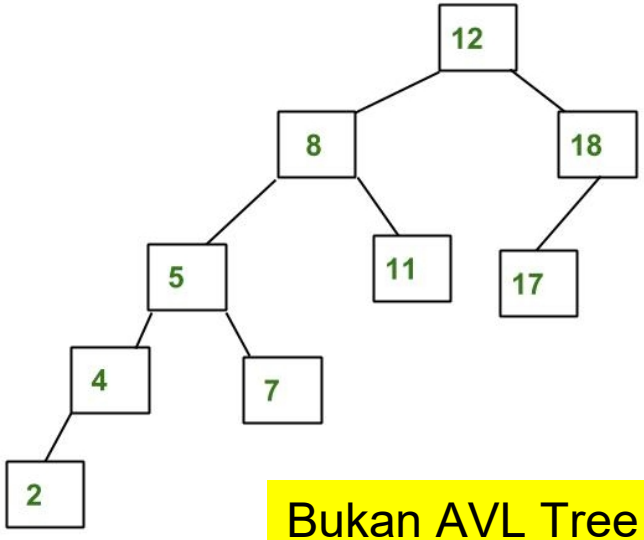
- BST dibangun berdasarkan urutan input data ke tree (akan membentuk BST yang berbeda – beda)
- Semakin pendek suatu tree, maka proses pencarian data/node akan lebih singkat

AVL Tree

- **AVL Tree** adalah Binary Search Tree yang memiliki perbedaan tinggi maksimal 1 antara subtree kiri dan subtree kanan
- AVL Tree muncul untuk menyeimbangkan Binary Search Tree (secara otomatis memastikan tree yang terbentuk selalu seimbang ***self balancing binary search tree***)
- Dengan AVL Tree, waktu pencarian suatu data dapat dipersingkat dan bentuk tree dapat disederhanakan

AVL Tree
(Balanced)



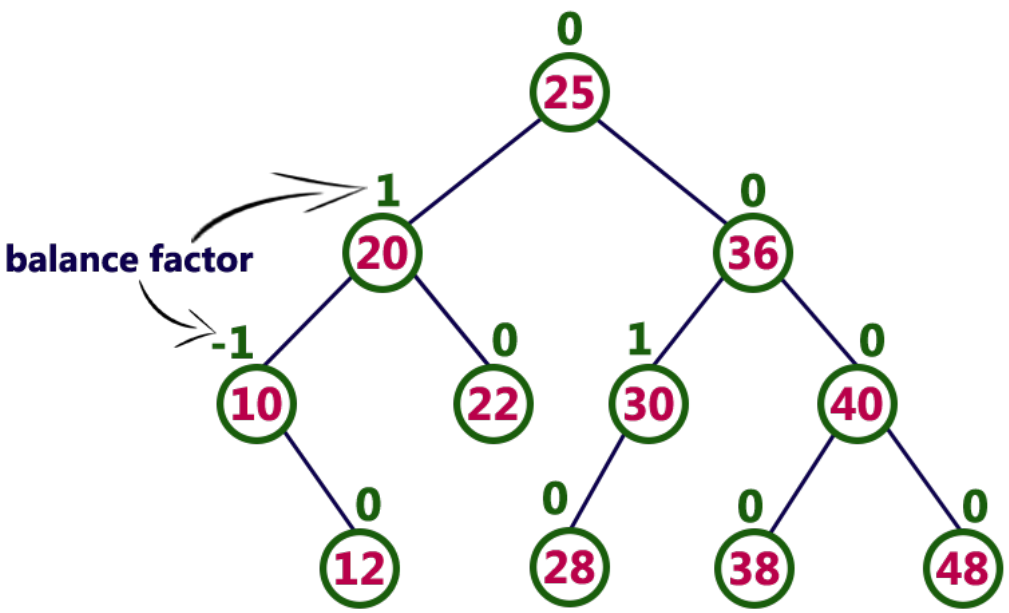


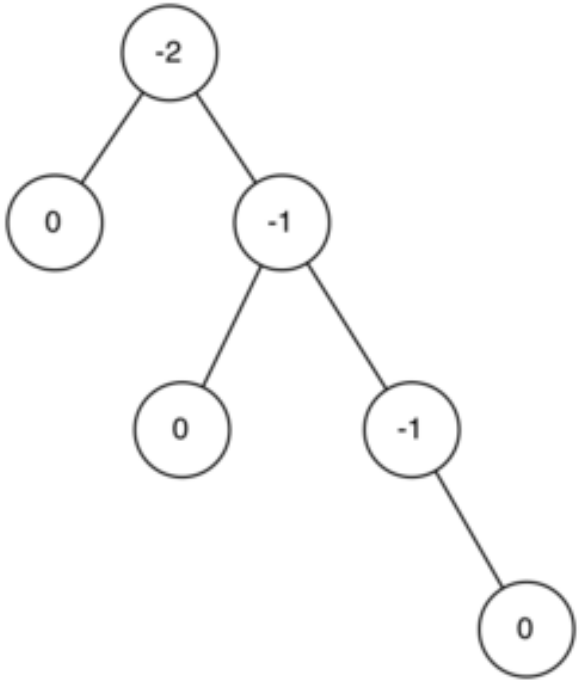
Bukan AVL Tree (Tidak Balanced)

AVL Tree

- Untuk memastikan tree selalu seimbang (*balanced*), setiap node memiliki perhitungan **Balance Factor**

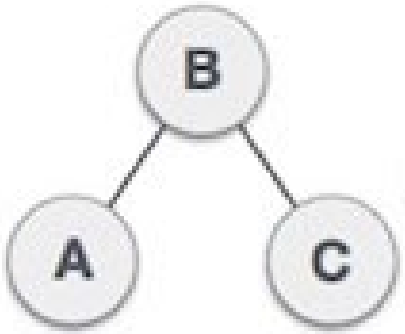
Balance Factor = tinggi (subtree kiri) − tinggi (subtree kanan)

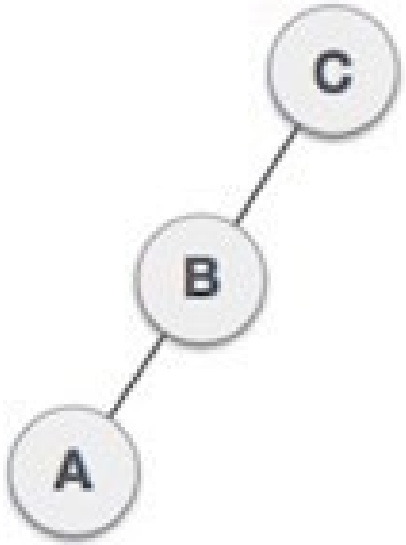


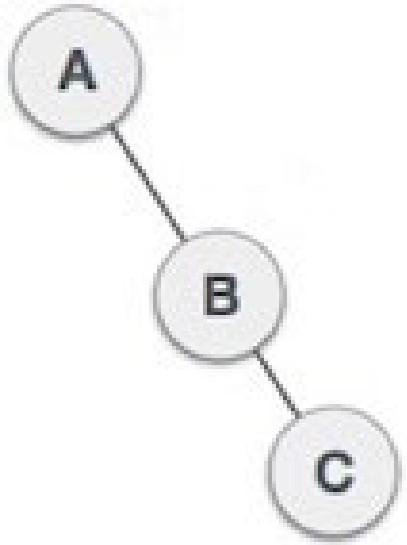


AVL Tree

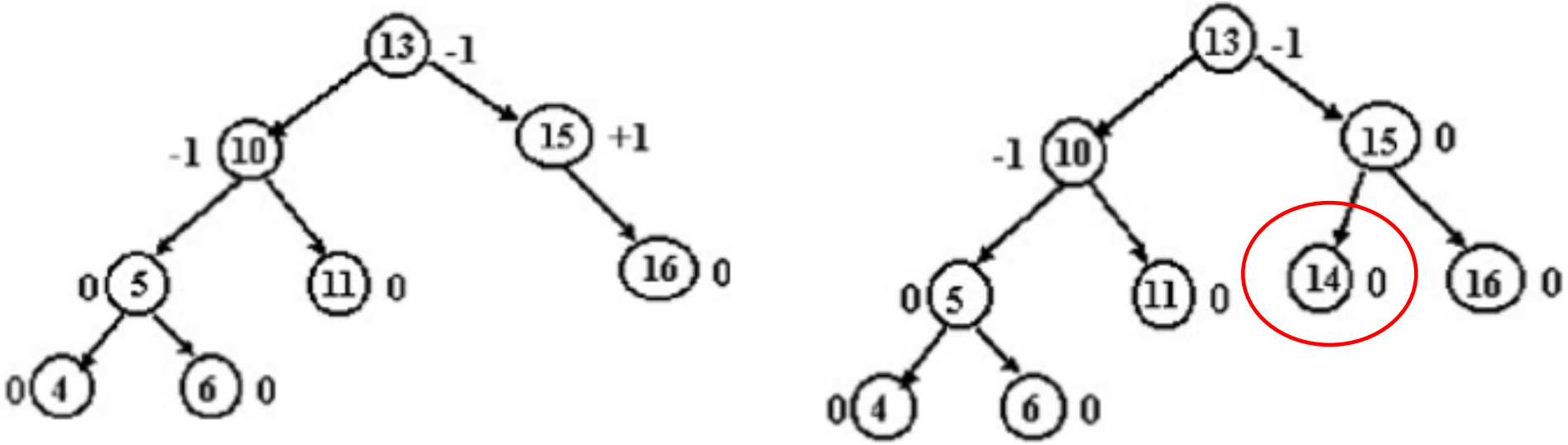
Mana yang merupakan AVL tree?





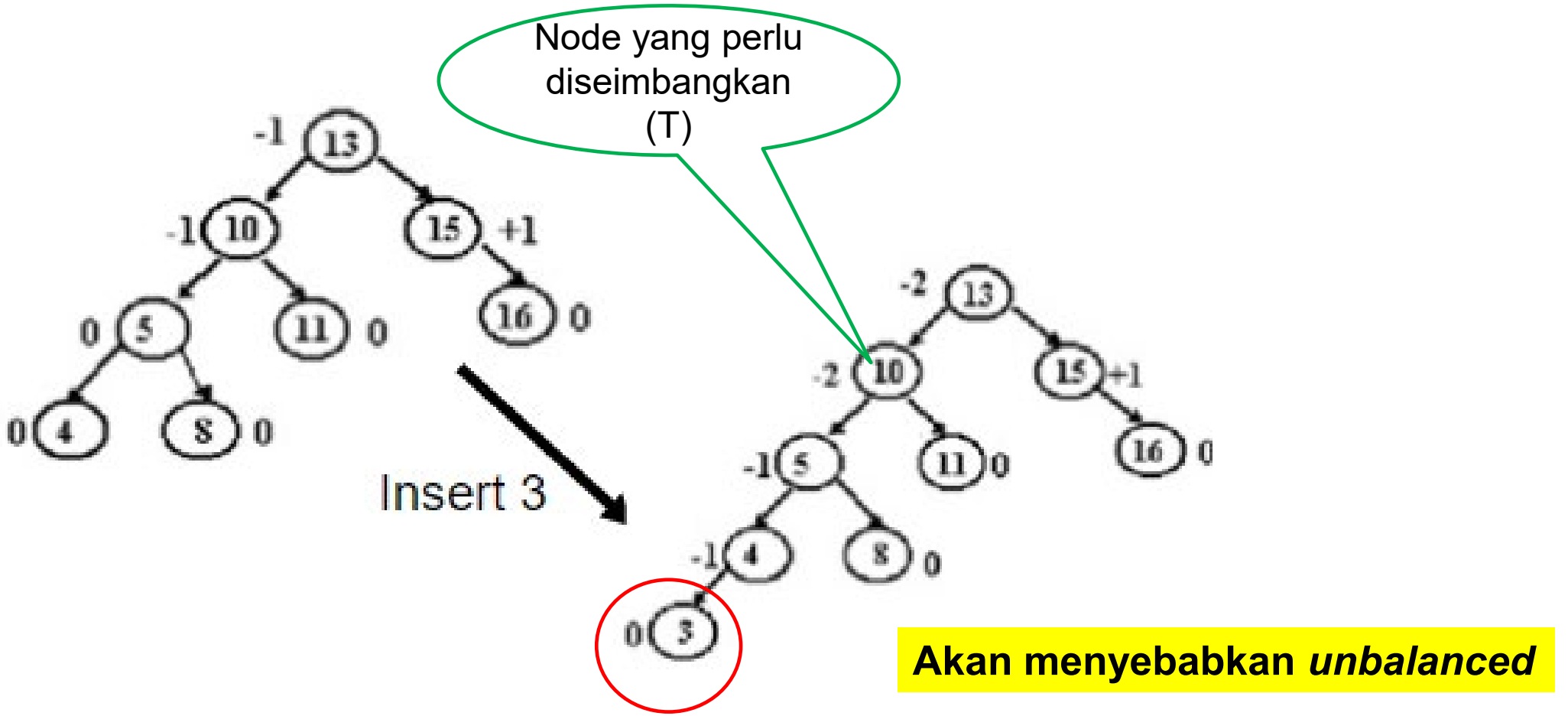


Operasi AVL Tree: Insert



Tidak menyebabkan *unbalanced*

Operasi AVL Tree: Insert

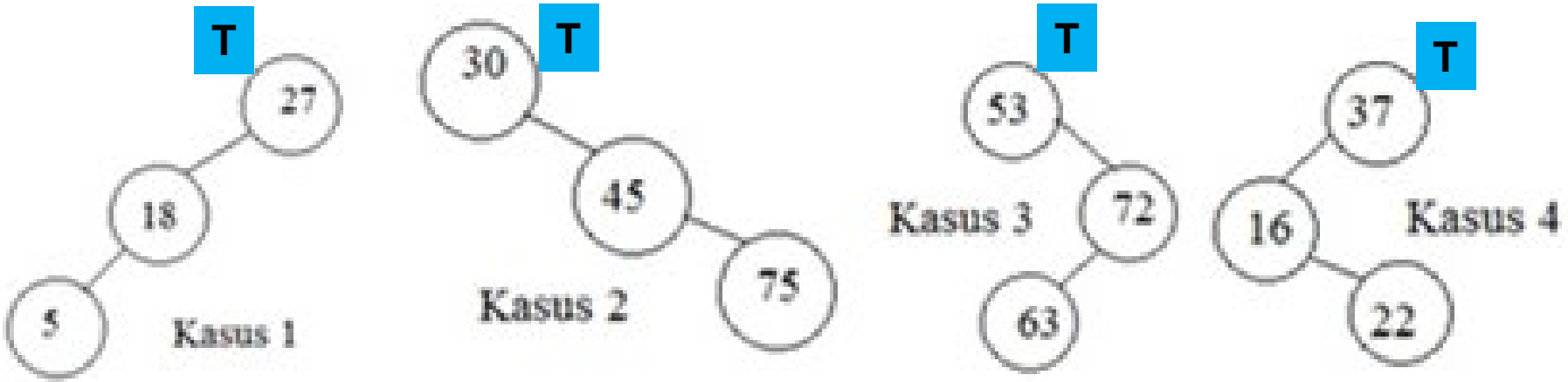


Akan menyebabkan *unbalanced*

Operasi AVL Tree: Insert

Ada 4 kasus yang biasanya terjadi setelah operasi ***insert*** dilakukan: (T adalah node yang harus diseimbangkan)

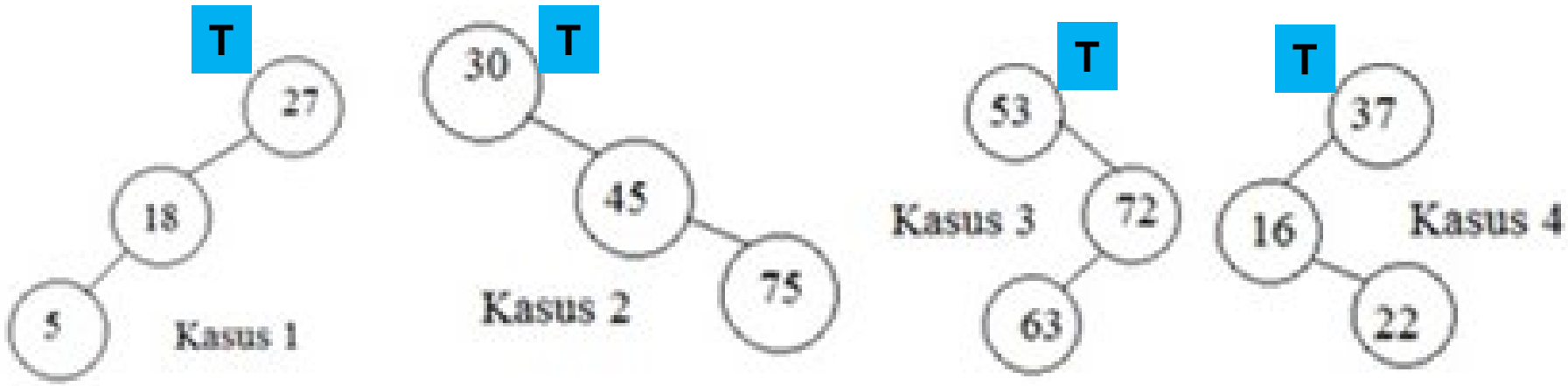
- Kasus 1 : node terbawah terletak pada subtree kiri dari anak kiri T (***left-left***)
- Kasus 2 : node terbawah terletak pada subtree kanan dari anak kanan T (***right-right***)
- Kasus 3 : node terbawah terletak pada subtree kanan dari anak kiri T (***right-left***)
- Kasus 4 : node terbawah terletak pada subtree kiri dari anak kanan T (***left-right***)



Operasi AVL Tree: Insert

Ke-4 kasus tersebut dapat diselesaikan dengan melakukan rotasi:

- Kasus 1 dan 2 dengan **single rotation**
- Kasus 3 dan 4 dengan **double rotation**

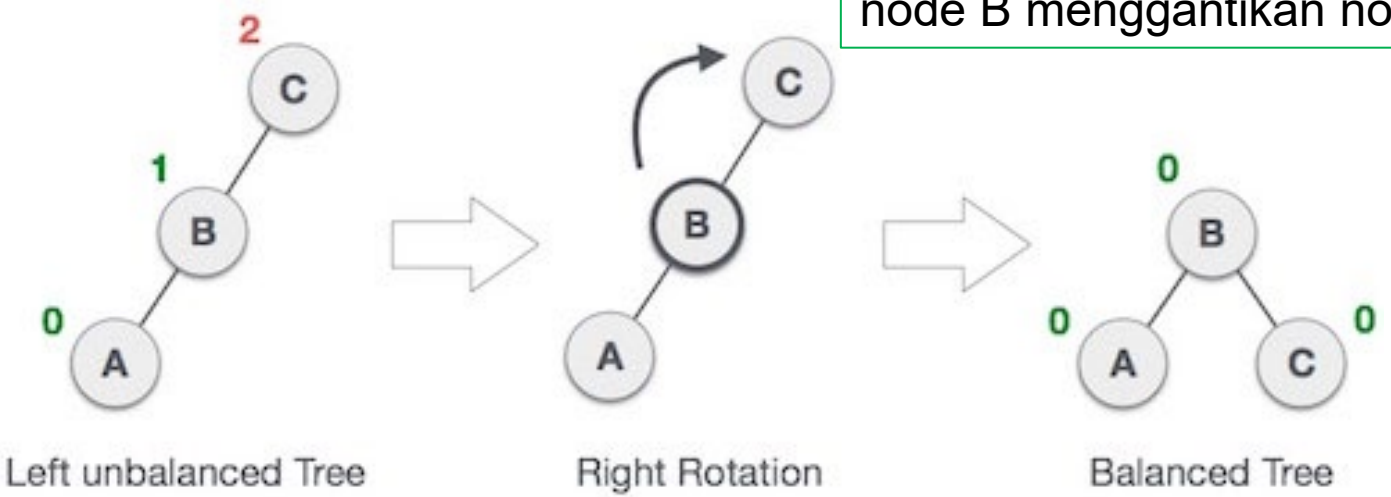


Left-left **Right-right** **Right-left** **Left-right**

Single Rotation

Kasus 1

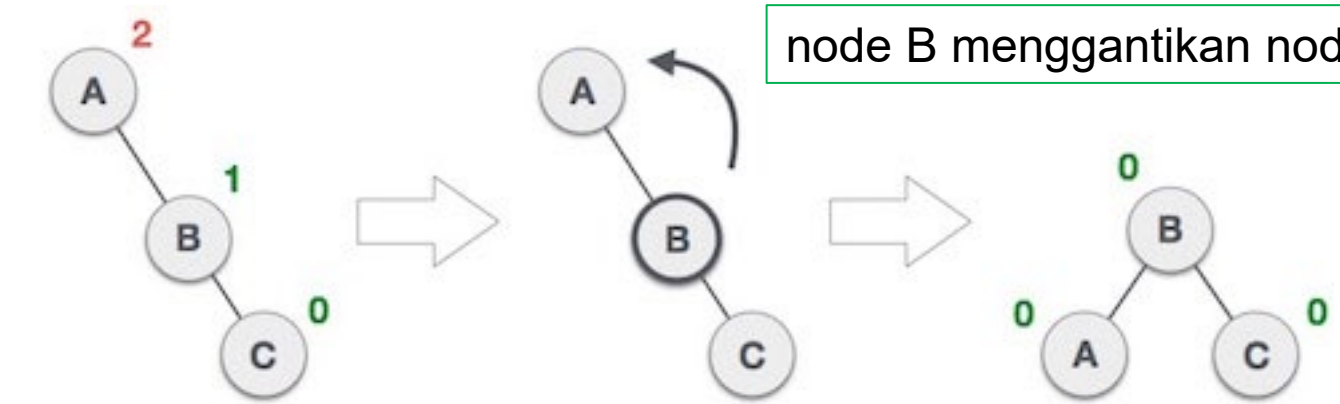
node B menggantikan node C



Left unbalanced Tree | Right Rotation | Balanced Tree

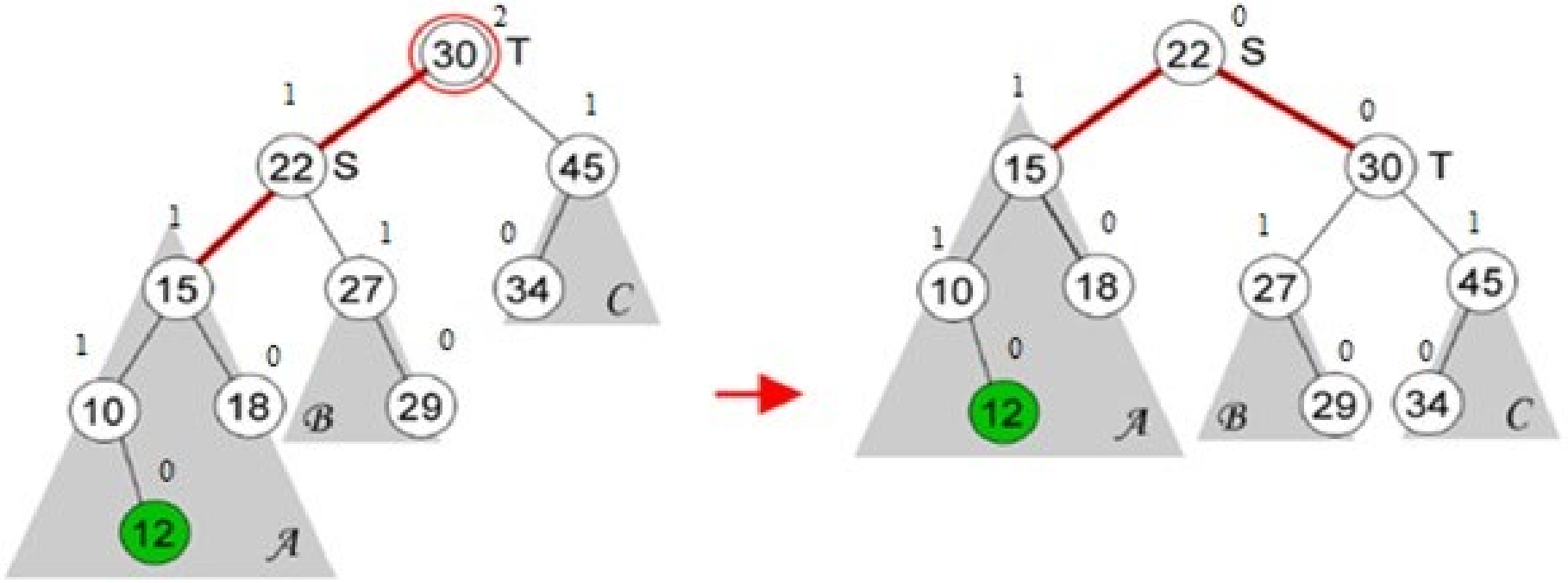
Kasus 2

node B menggantikan node A



Right unbalanced tree | Left Rotation | Balanced

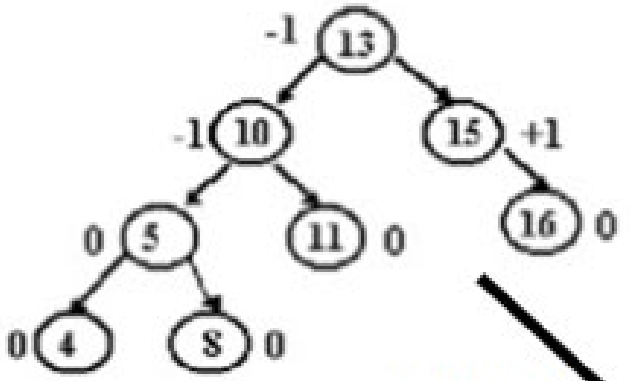
Single Rotation (Contoh)



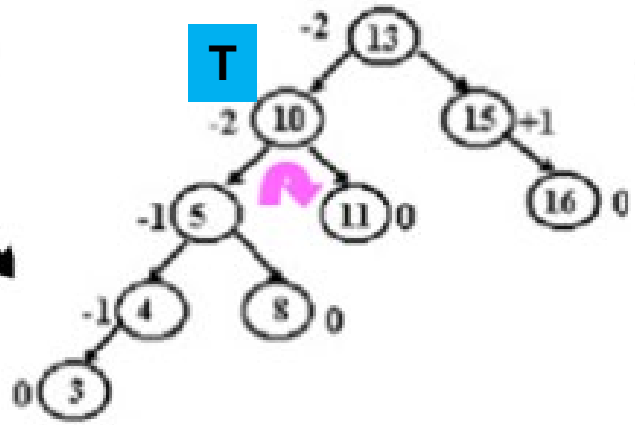
Node T adalah 30, lakukan rotasi kanan dengan node anak (kiri/kanan) menggantikan node T

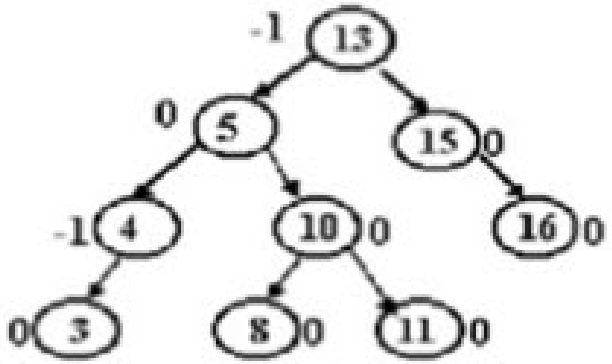
Jadikan subtree B sebagai anak kiri node T

Single Rotation (Contoh)



Insert 3

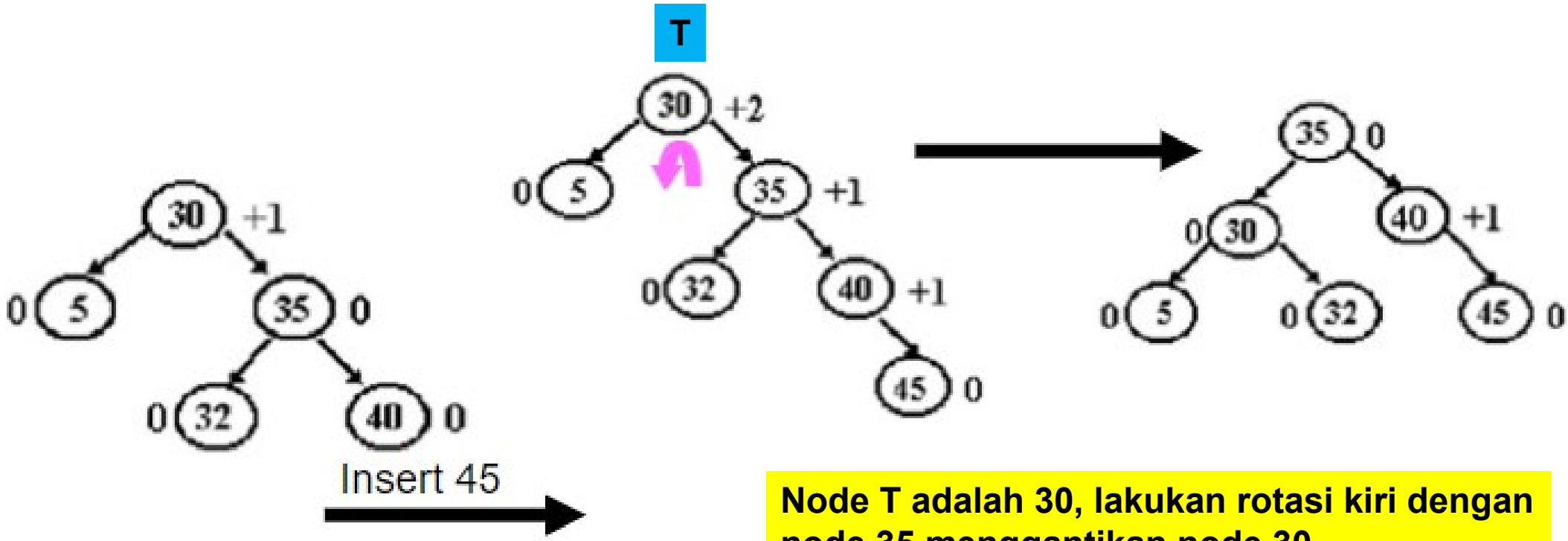




Node T adalah 10, lakukan rotasi kanan dengan node 5 menggantikan node 10

Node 8 menjadi anak kiri node T (10)

Single Rotation (Contoh)

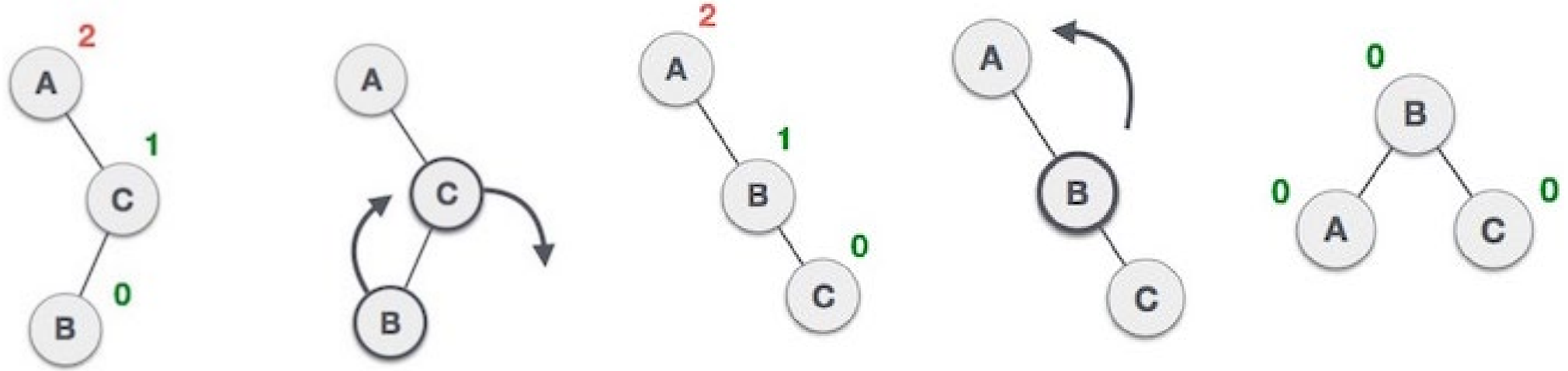


Node T adalah 30, lakukan rotasi kiri dengan node 35 menggantikan node 30

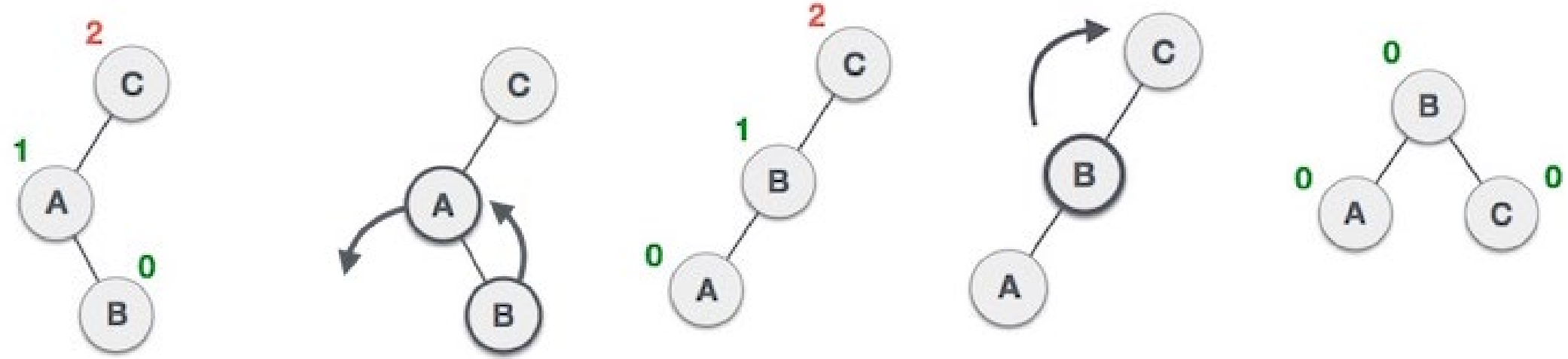
Node 32 menjadi anak kanan node T (30)

Double Rotation

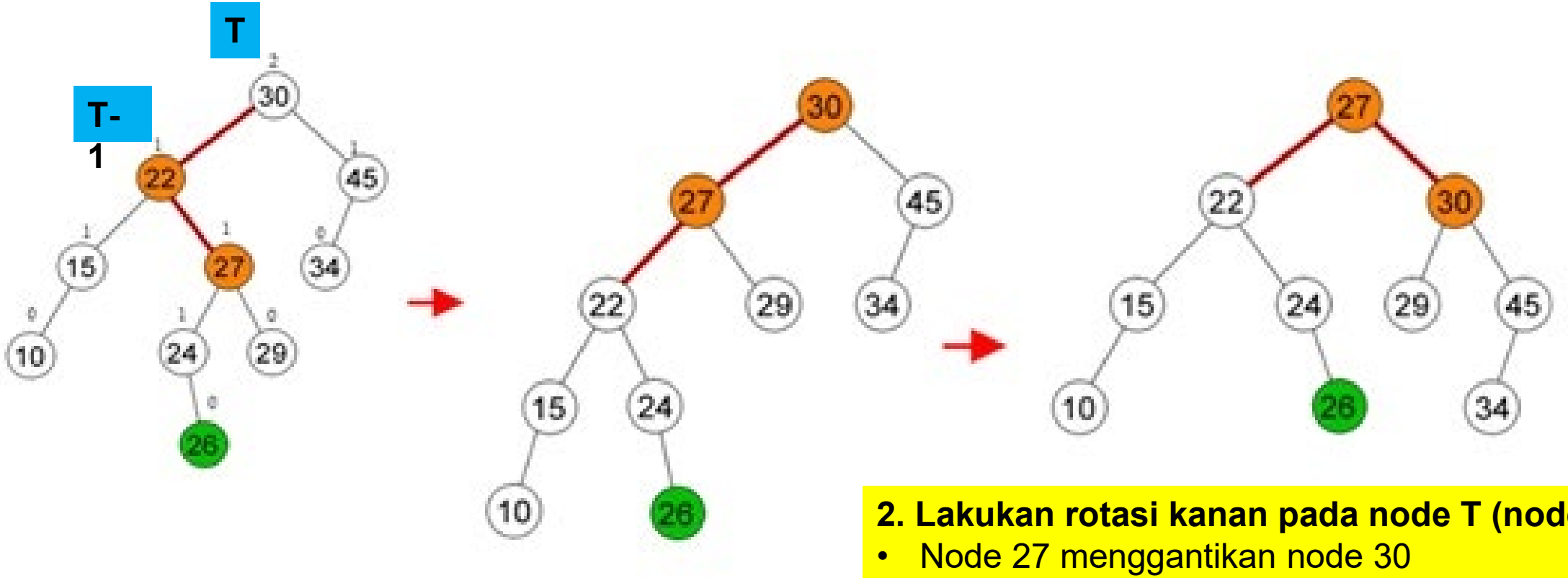
Kasus 3



Kasus 4



Double Rotation (Contoh)



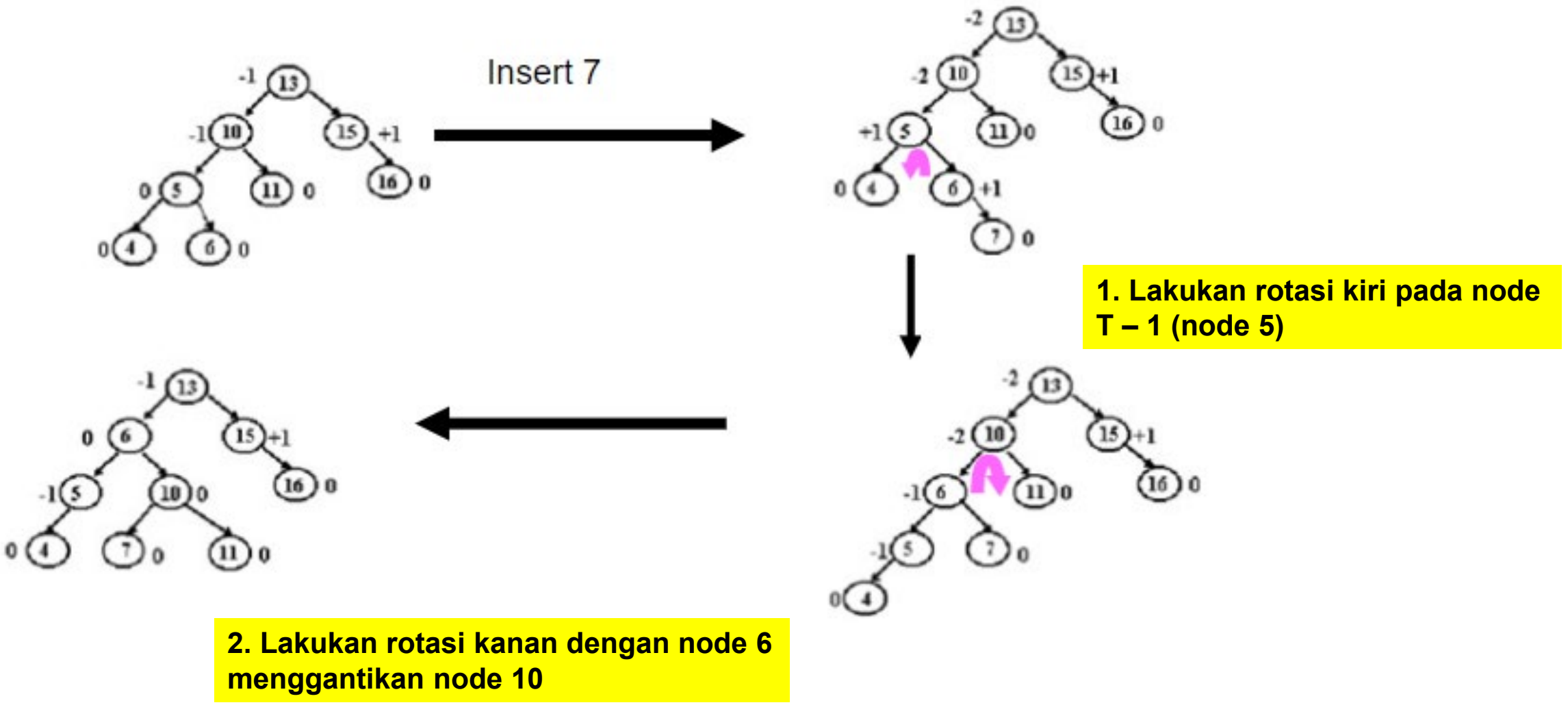
1. Lakukan rotasi kiri pada node T – 1 (node 22)

- Node 27 menggantikan node 22
- Subtree (24,26) menjadi anak kanan node 22

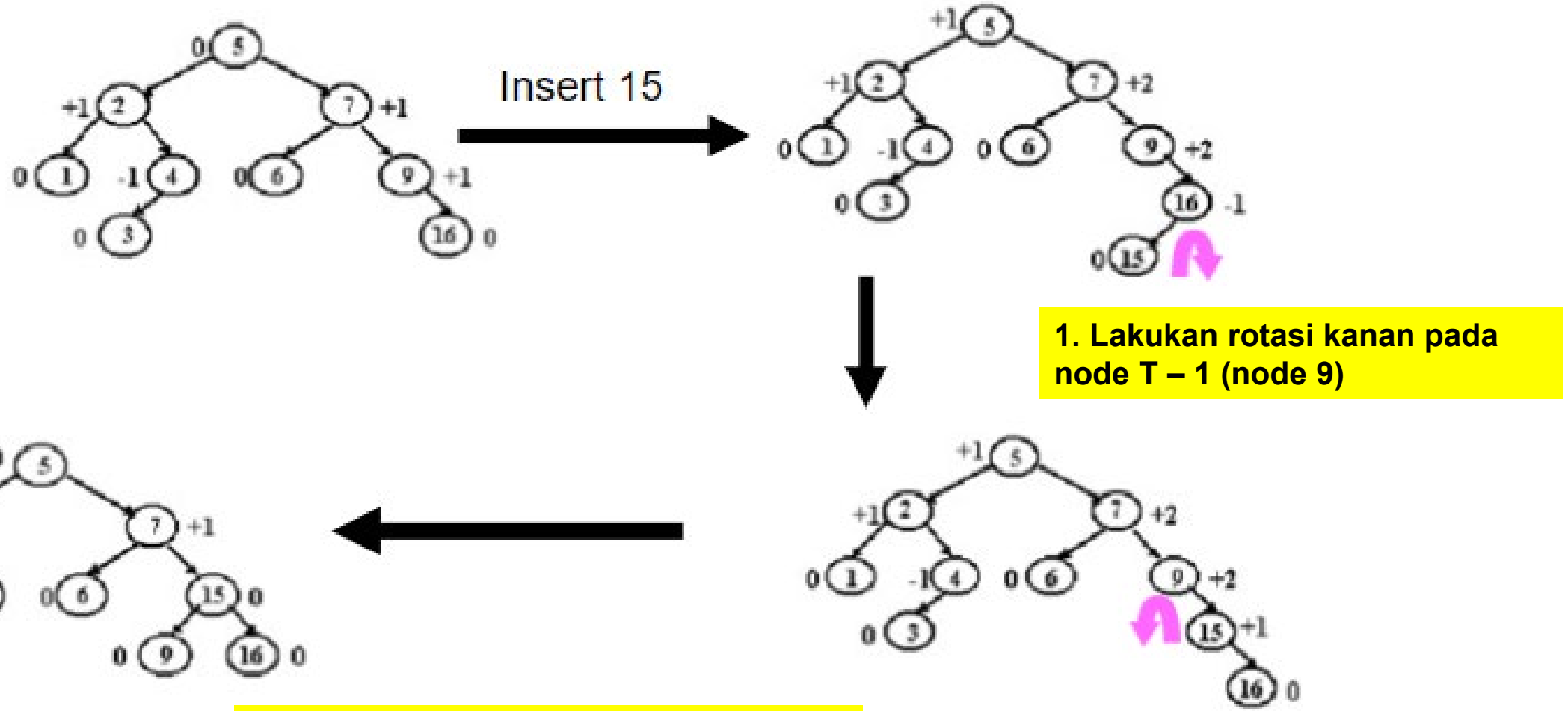
2. Lakukan rotasi kanan pada node T (node 30)

- Node 27 menggantikan node 30
- Node 29 menjadi anak kiri node 30

Double Rotation (Contoh)

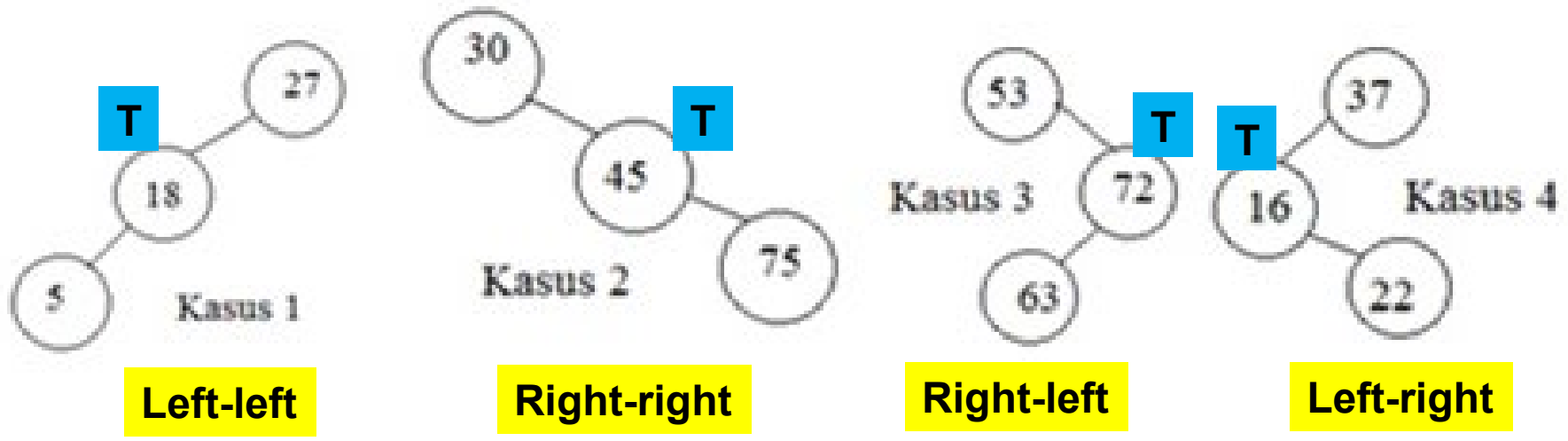


Double Rotation (Contoh)

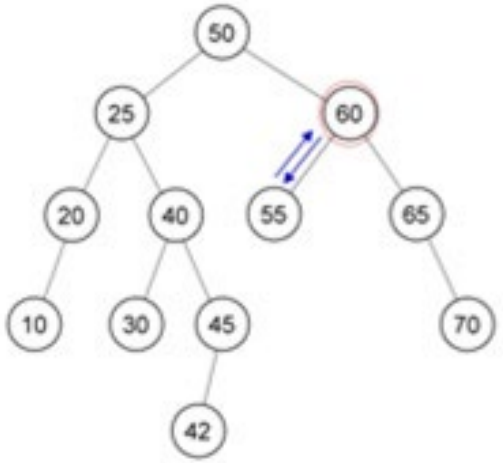


Operasi AVL Tree: Delete

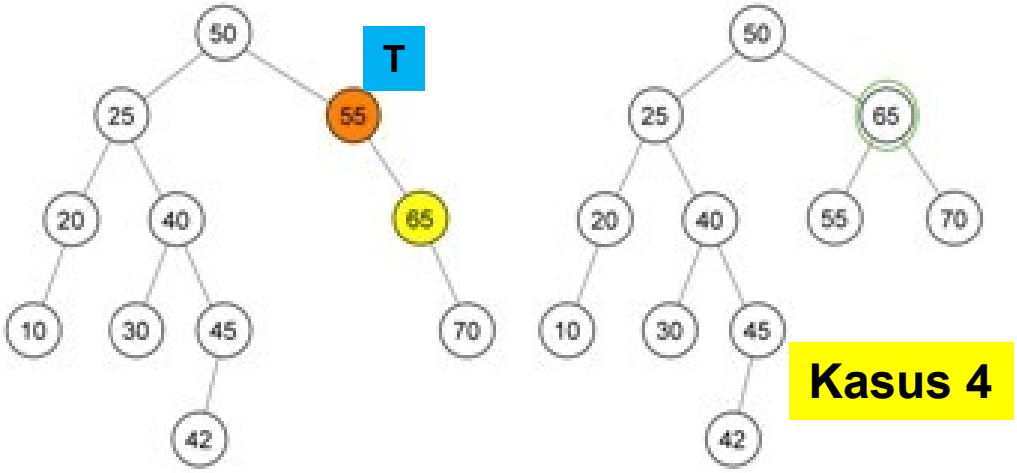
- Jika node yang akan dihapus berada pada posisi *leaf* atau node terbawah, maka dapat langsung di hapus
- Jika node yang akan dihapus memiliki anak, maka proses penghapusannya harus di cek kembali untuk menyeimbangkan Binary Search Tree dengan perbedaan tinggi / level maksimal 1
  - Diseimbangkan dengan melakukan rotasi, sama seperti waktu ***insert***
  - Kasus 1 dan 2 dengan **single rotation**
  - Kasus 3 dan 4 dengan **double rotation**



Operasi AVL Tree: Delete (Contoh)



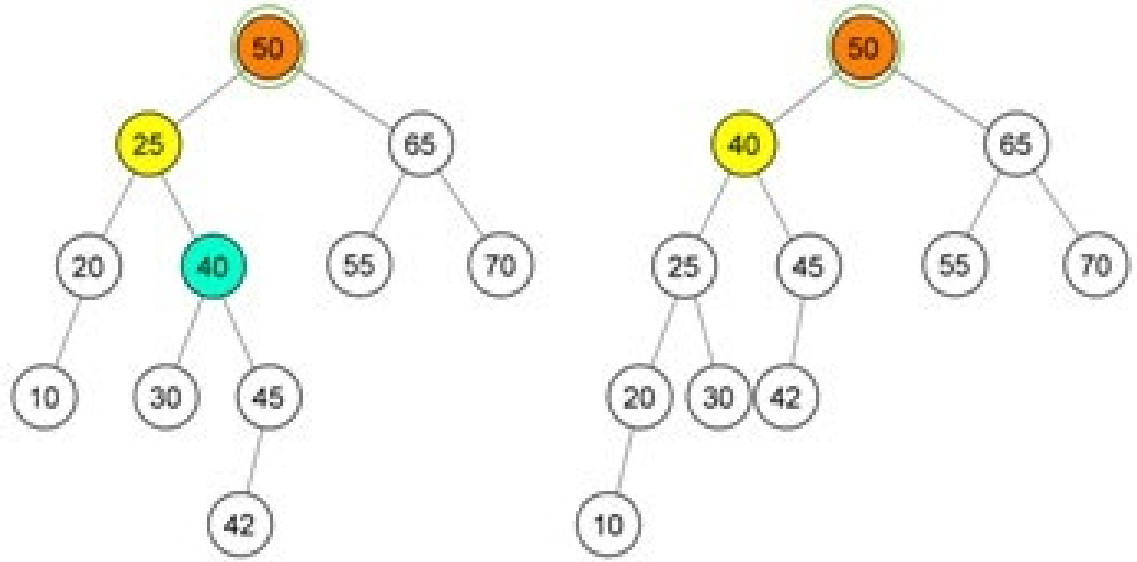
Kasus 2

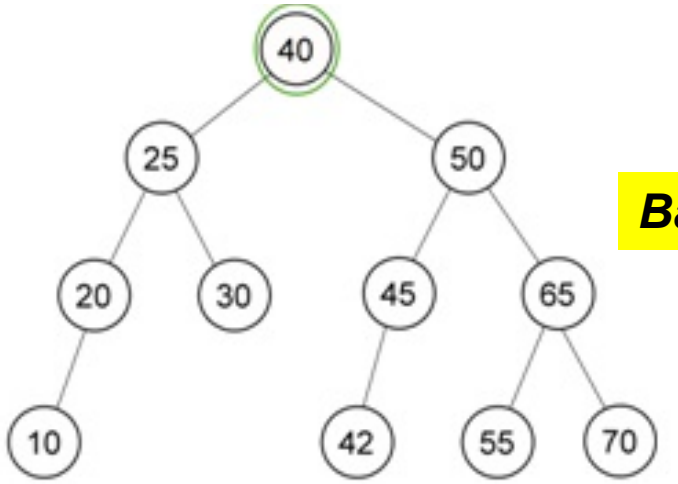


Kasus 4

- Jika kita menghapus **node 60** di AVL tree sebelah kiri, maka **node 55** akan menggantikan **node 60** (**Ingat**: cari maksimum di subtree kiri atau minimum di subtree kanan untuk menggantikan node yang dihapus)
- Tree menjadi tidak *balanced* (kasus 2) setelah proses penghapusan dan harus diseimbangkan
- Lakukan single rotation (rotasi kiri) pada **node T = 55** □ ternyata masih tidak seimbang (kasus 4)

Operasi AVL Tree: Delete





Balanced

- Diperlukan double rotation untuk kasus 4 (**rotasi kiri** pada node 25 dan **rotasi kanan** pada node 50) untuk menyeimbangkan AVL tree

Implementasi program AVL tree

- int getBalanceFactor(struct Node *N)
- struct Node* rightRotate(struct Node *y)
- struct Node *leftRotate(struct Node *x)
- struct Node* insert(struct Node* node, int new_data)
- struct Node* deleteNode(struct Node* root, int deleted_data)

```
struct Node
{
    int data;
    struct Node *left;
    struct Node *right;
    int height;
};
```

Latihan

- Gambarkan proses pembentukan AVL Tree dengan memasukan elemen – elemen dengan urutan sebagai berikut: 10, 20, 15, 25, 30, 16, 18, 19 dimulai dari Tree kosong!
- Gambarkan proses penghapusan node 30 dari AVL Tree yang terbentuk!

TERIMA KASIH