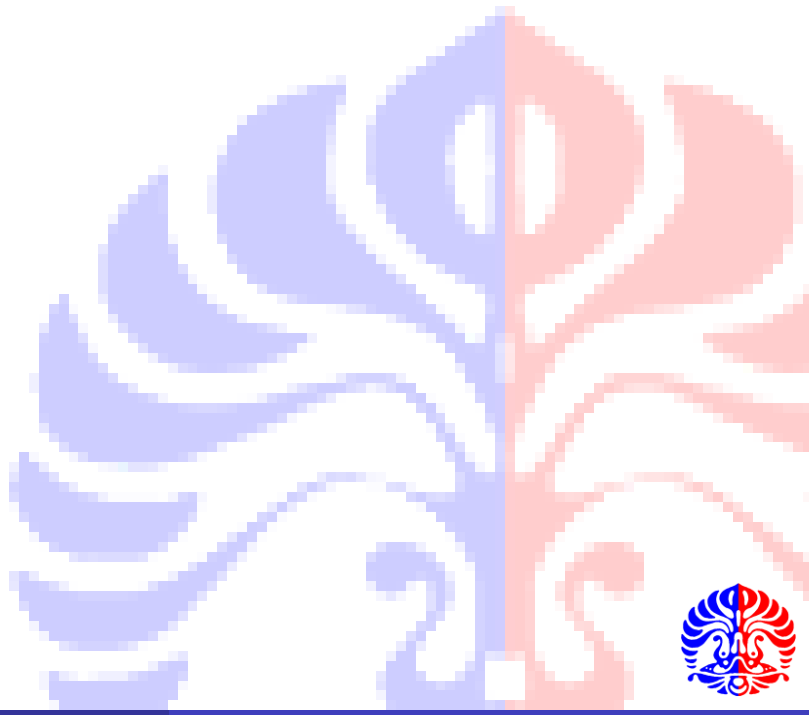
IKI10400 • Struktur Data & Algoritma: *AVL Tree*

Fakultas Ilmu Komputer • Universitas Indonesia

Slide acknowledgments:
Suryana Setiawan, Ade Azurat, Denny, Ruli Manurung, Tisha Melia, Bayu Distiawan

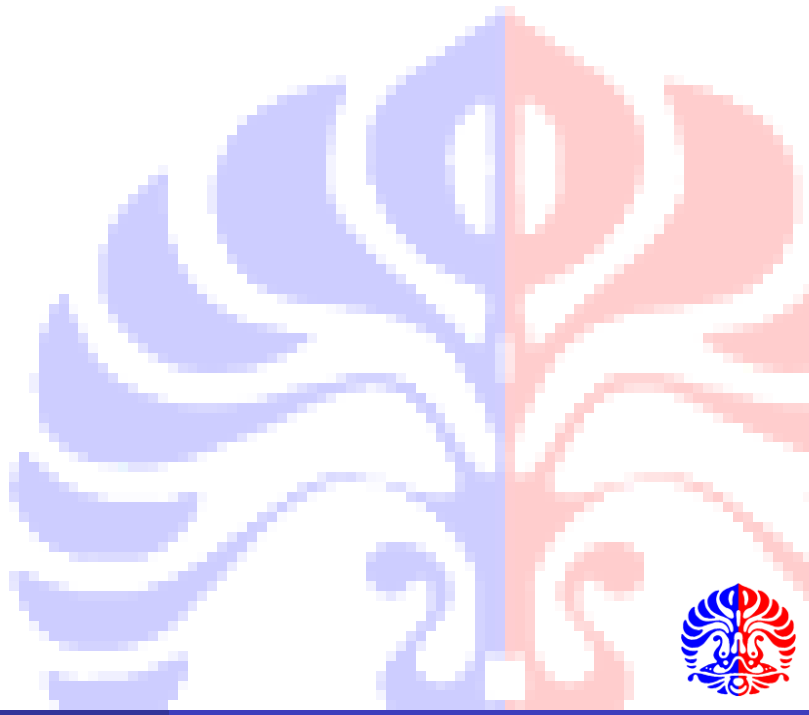
Tujuan

- Memahami variant dari Binary Search Tree yang balanced



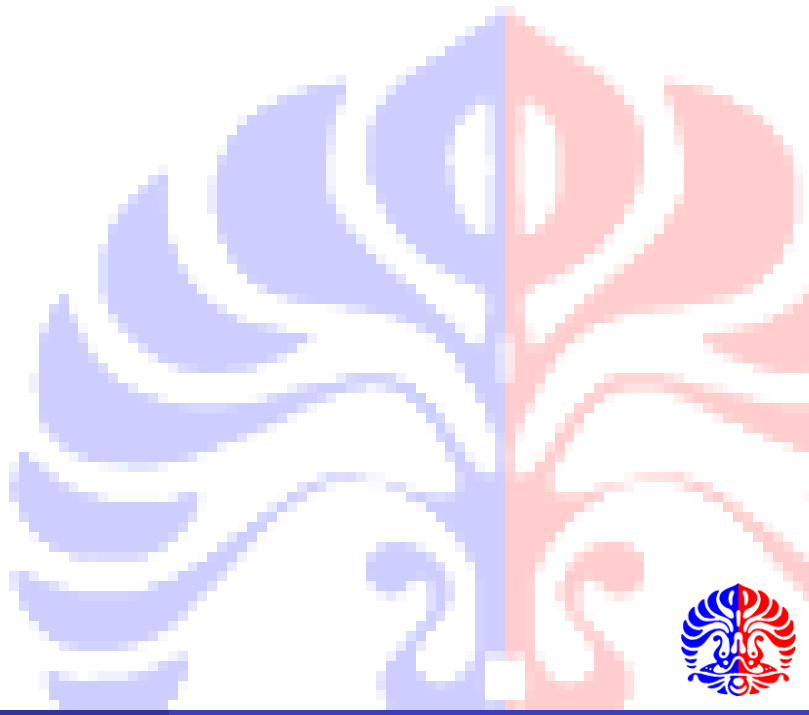
Outline

- AVL Tree
  - Definition
  - Property
  - Operations



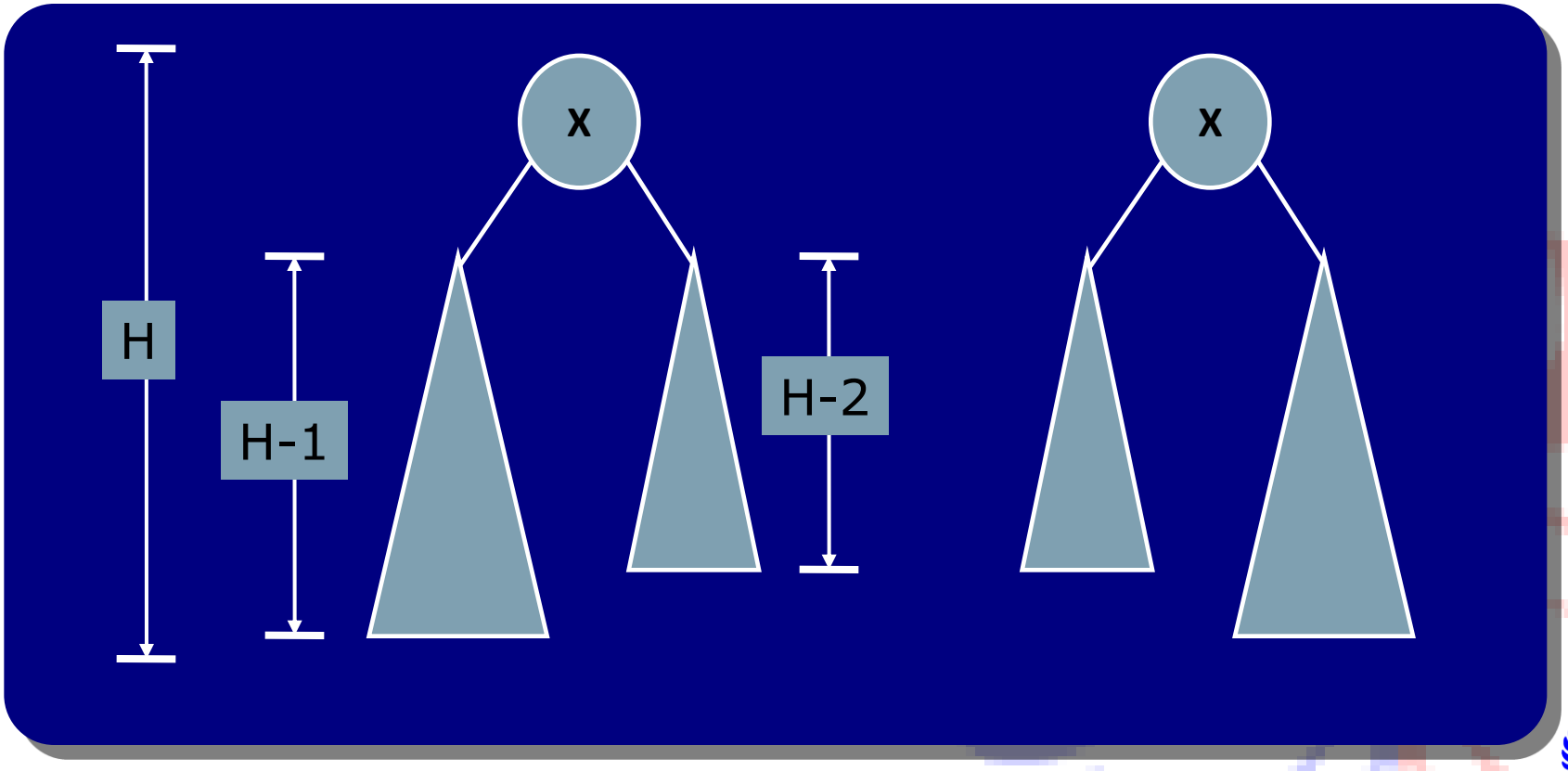
Motivasi

- Binary Search Tree yang tidak *balance* dapat membuat seluruh operasi memiliki kompleksitas running time O(n) pada kondisi worst case.

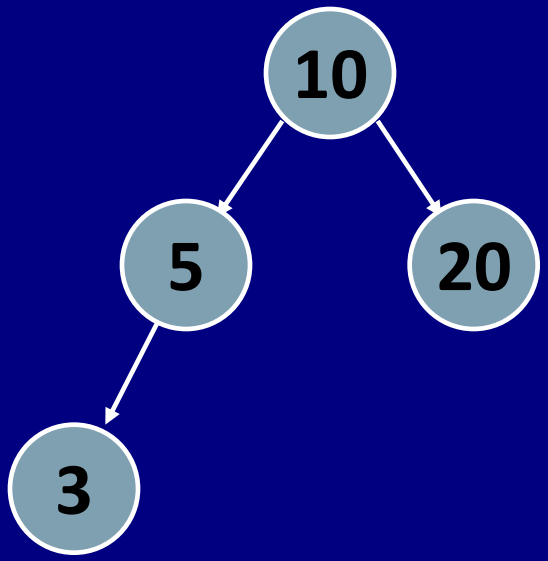


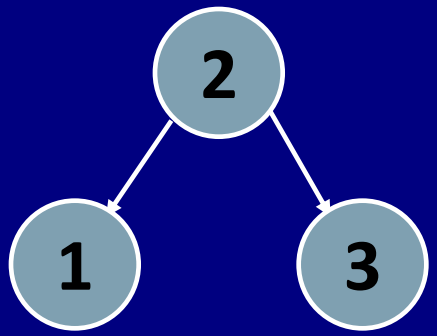
AVL Trees

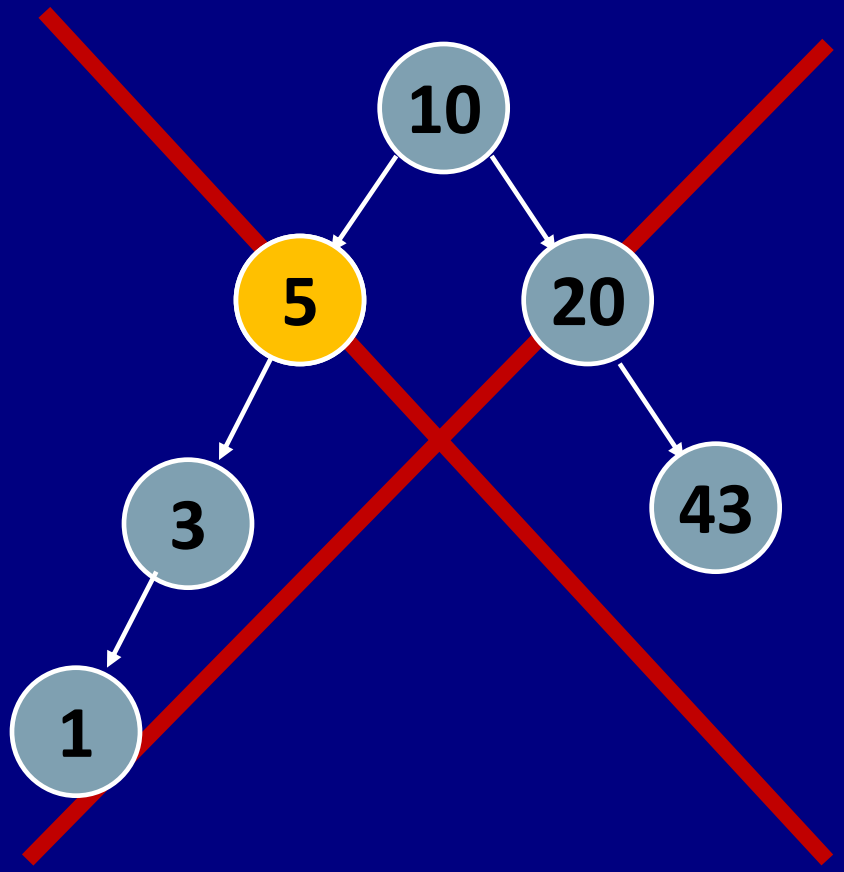
- Untuk setiap node dalam tree, ketinggian subtree di anak kiri dan subtree di anak kanan hanya berbeda maksimum 1.



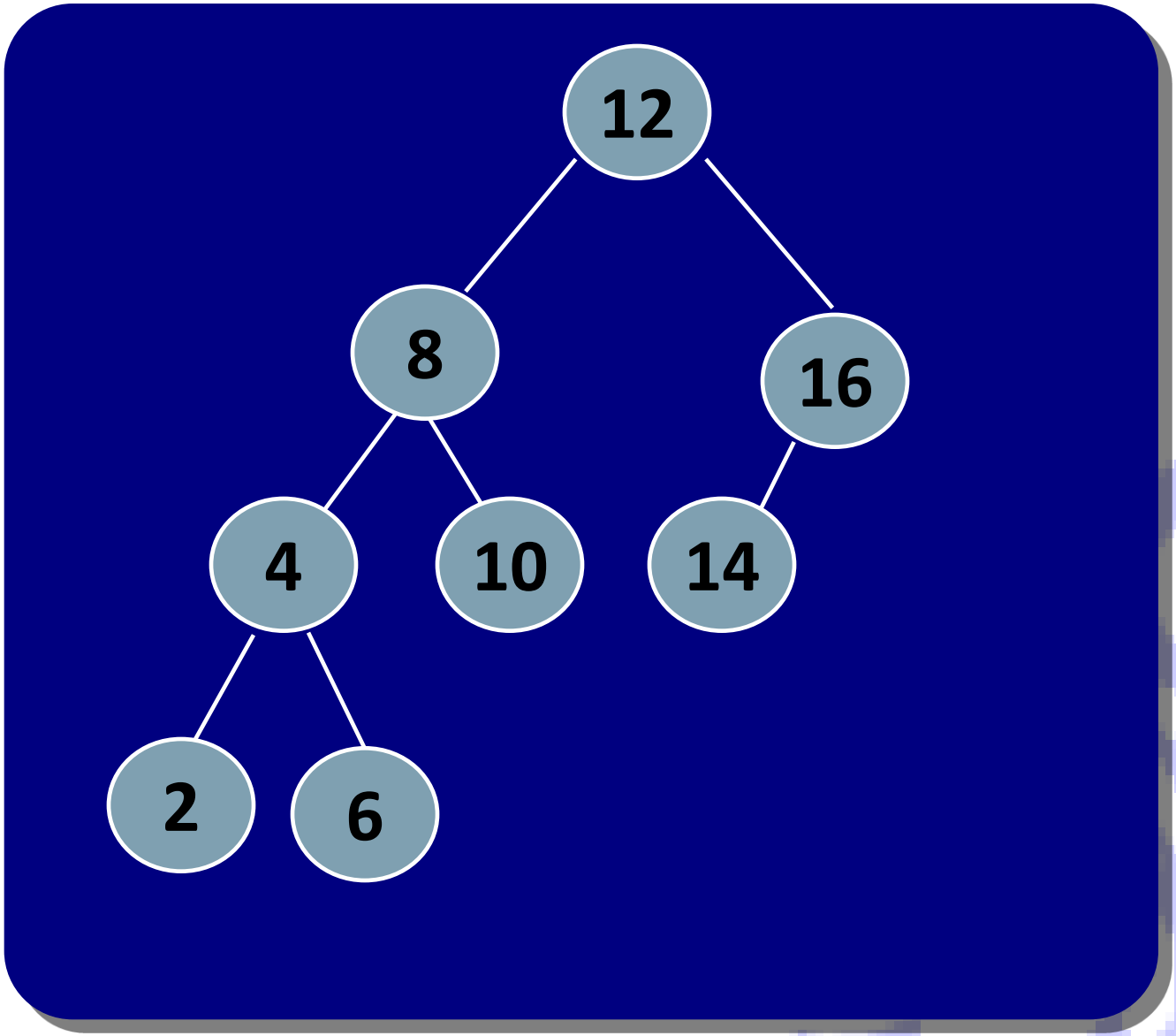
AVL Trees





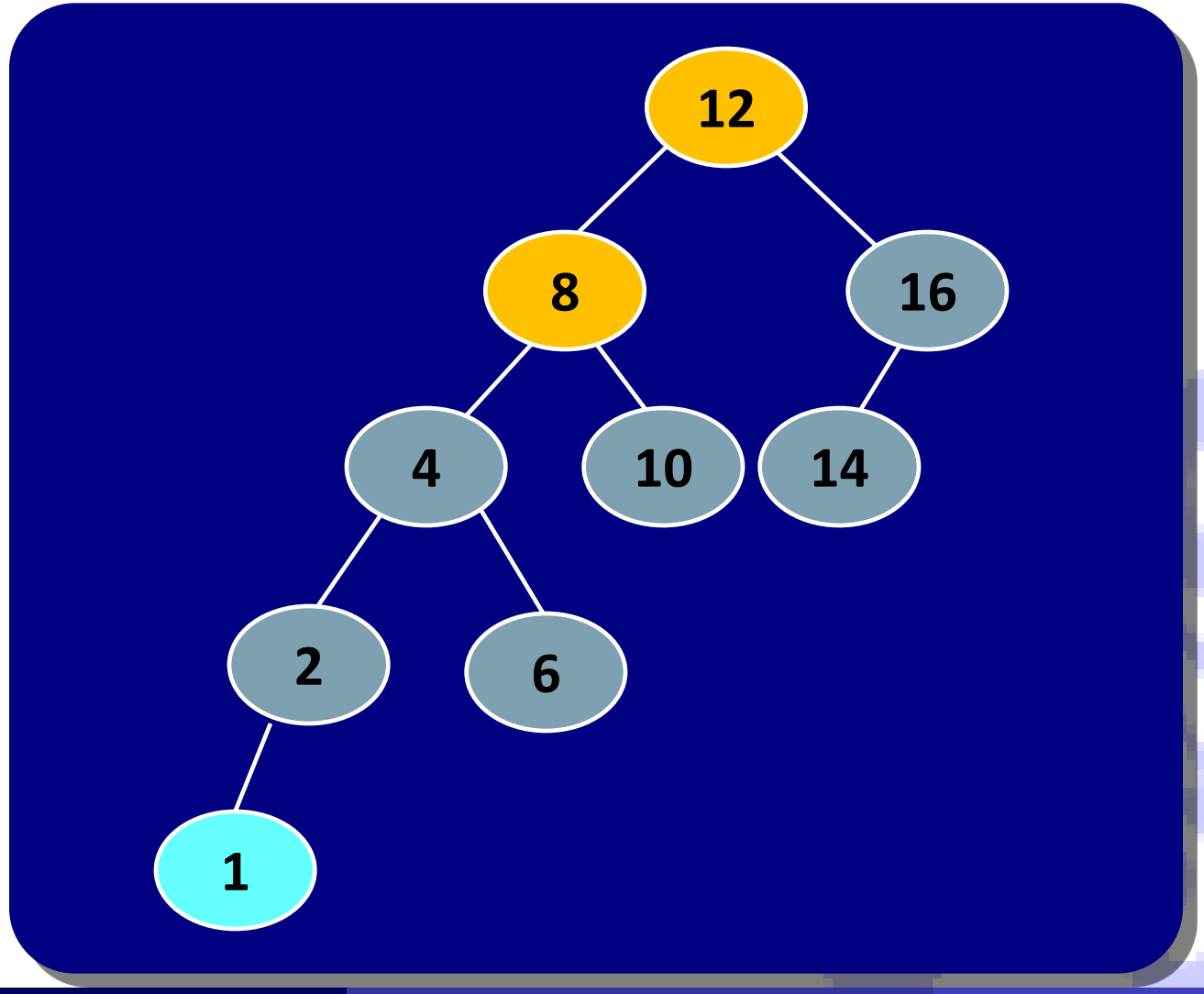


AVL Trees



Insertion pada AVL Tree

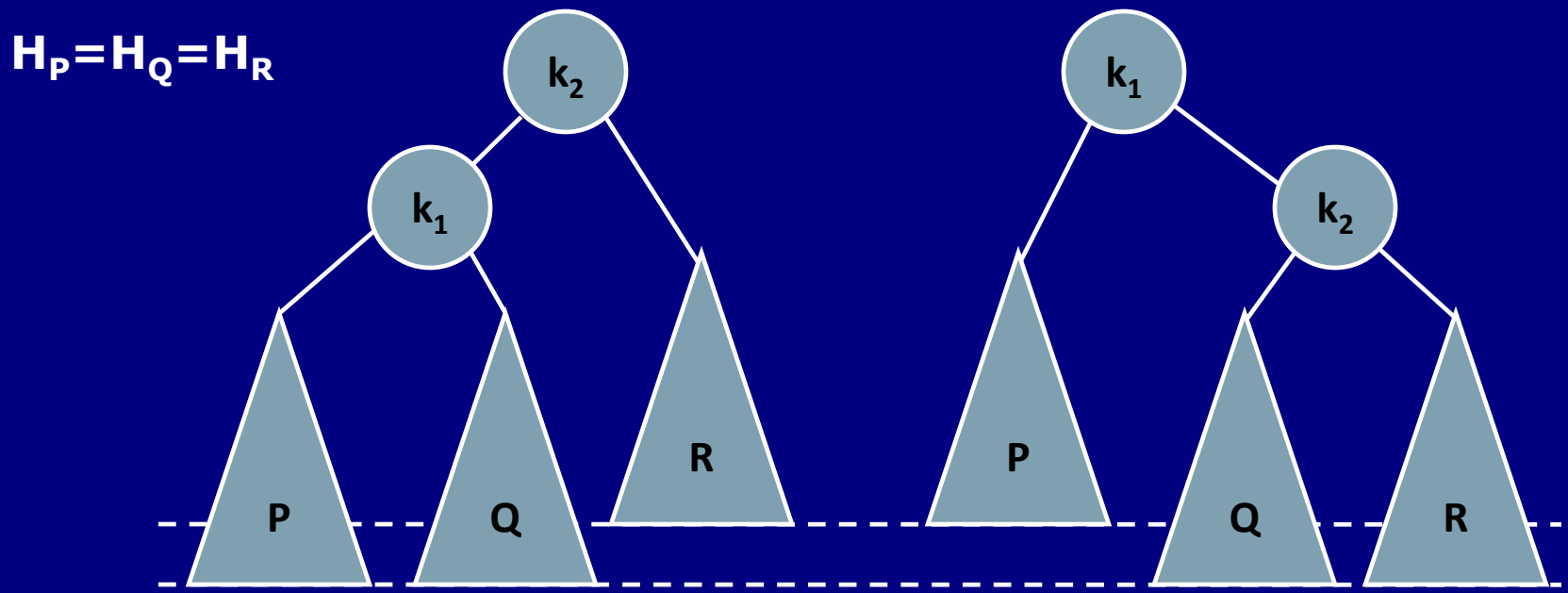
- Setelah *insert* 1



Insertion pada AVL Tree

- Untuk menjamin kondisi *balance* pada AVL tree, setelah penambahan sebuah node. jalur dari node baru tersebut hingga root di simpan dan di periksa kondisi *balance* pada tiap node-nya.
- Jika setelah penambahan, kondisi *balance* tidak terpenuhi pada node tertentu, maka lakukan salah satu rotasi berikut:
  - *Single rotation*
  - *Double rotation*

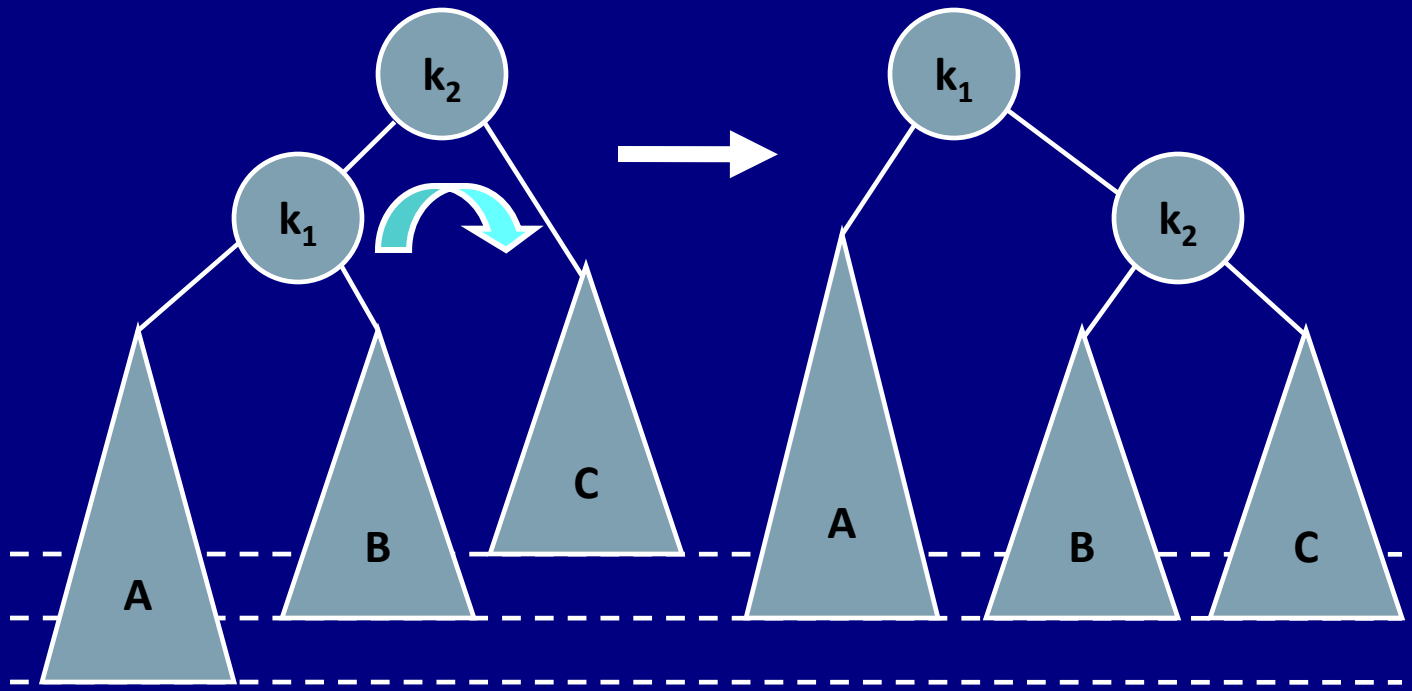
Kondisi tidak *balance*



- Sebuah penambahan pada subtree:
  - P (outside) - case 1
  - Q (inside) - case 2
- Sebuah penambahan pada subtree:
  - Q (inside) - case 3
  - R (outside) - case 4

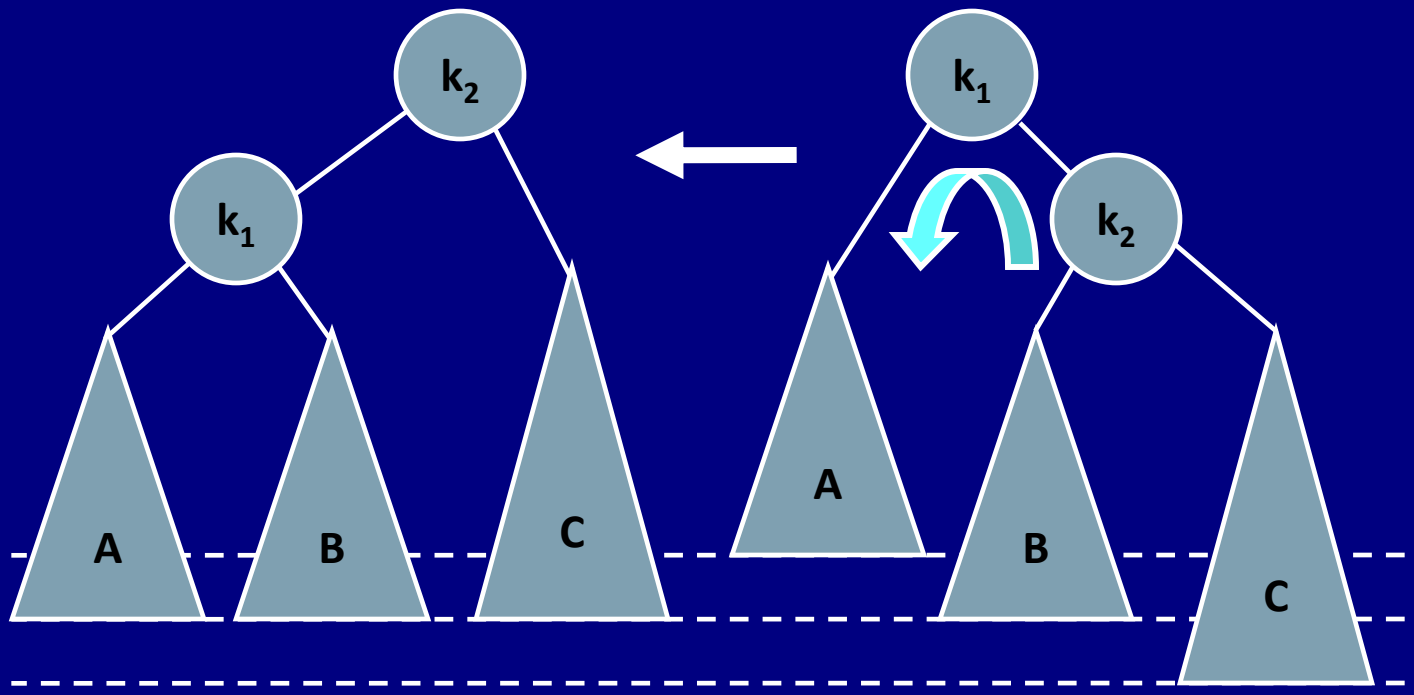
Single Rotation (case 1)

$H_A = H_B + 1$
$H_B = H_C$



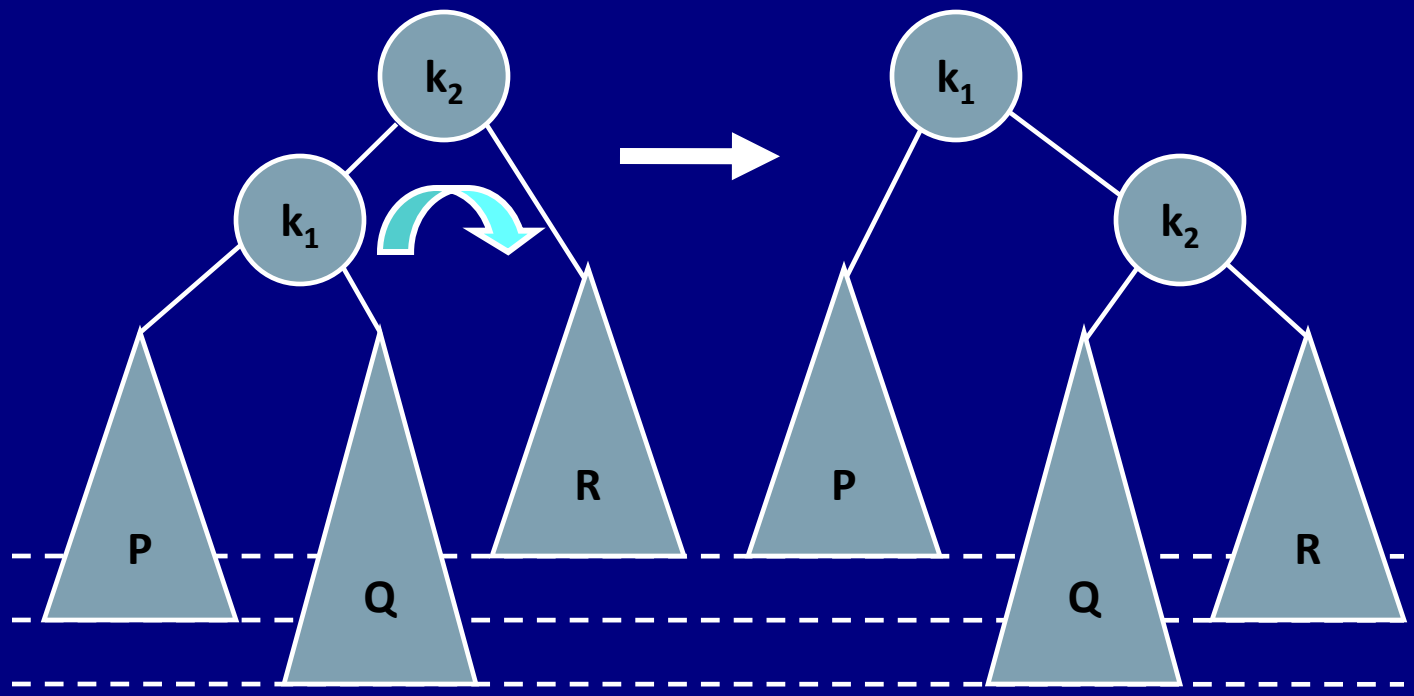
Single Rotation (case 4)

$H_A = H_B$
$H_C = H_B + 1$



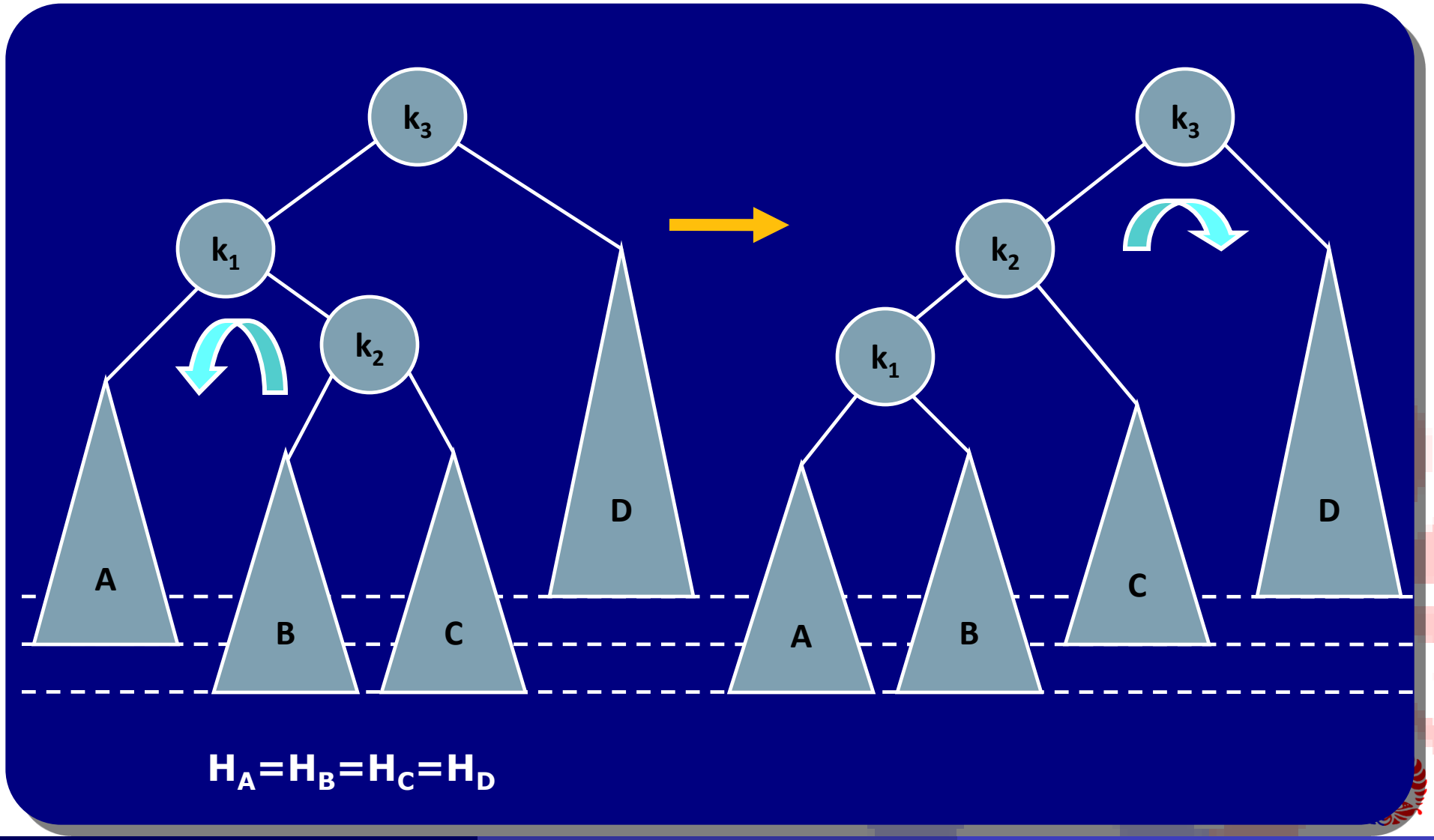
Keterbatasan Single Rotation

- Single rotation tidak bisa digunakan untuk kasus 2 dan 3 (*inside case*)

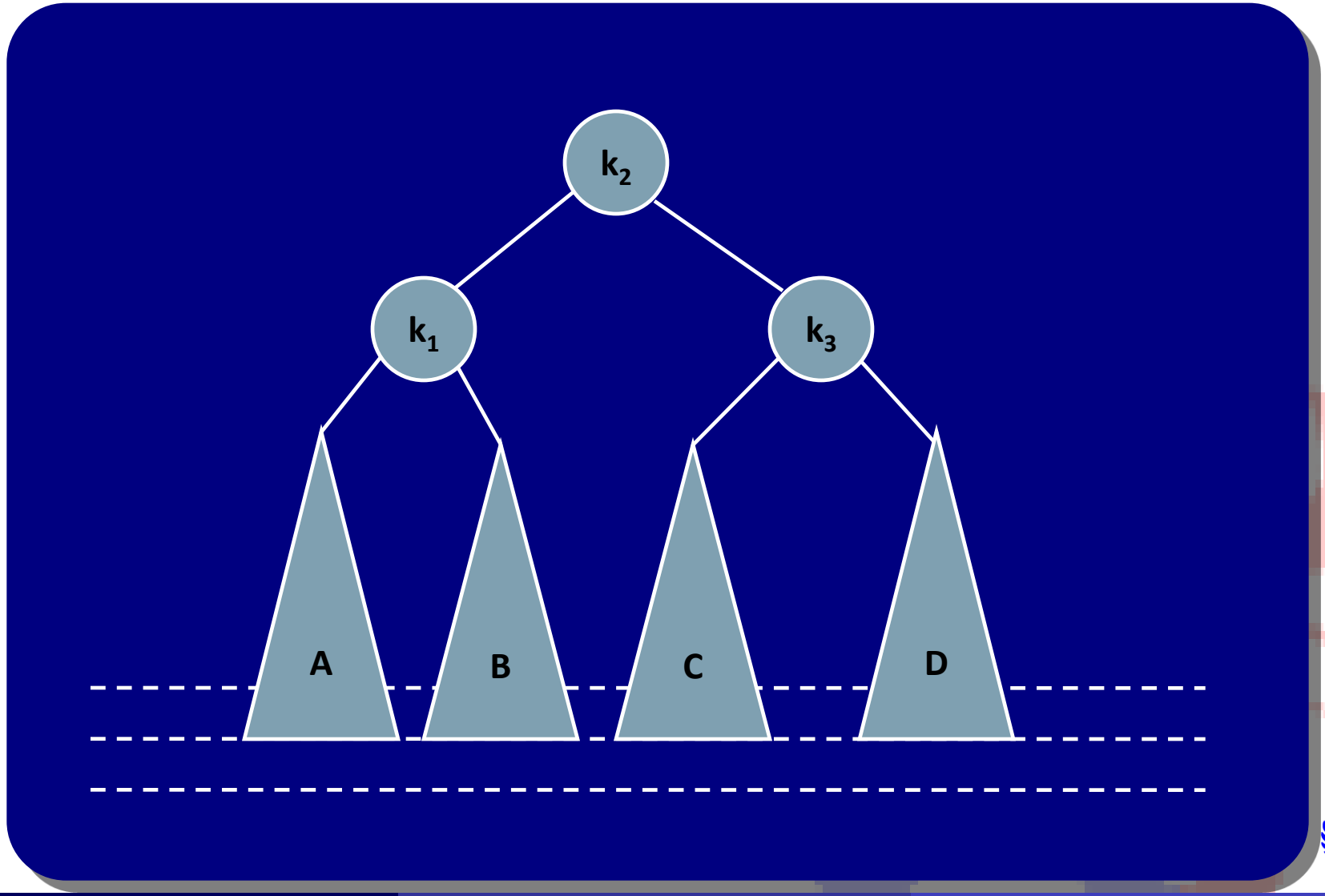


$H_Q = H_P + 1$
$H_P = H_R$

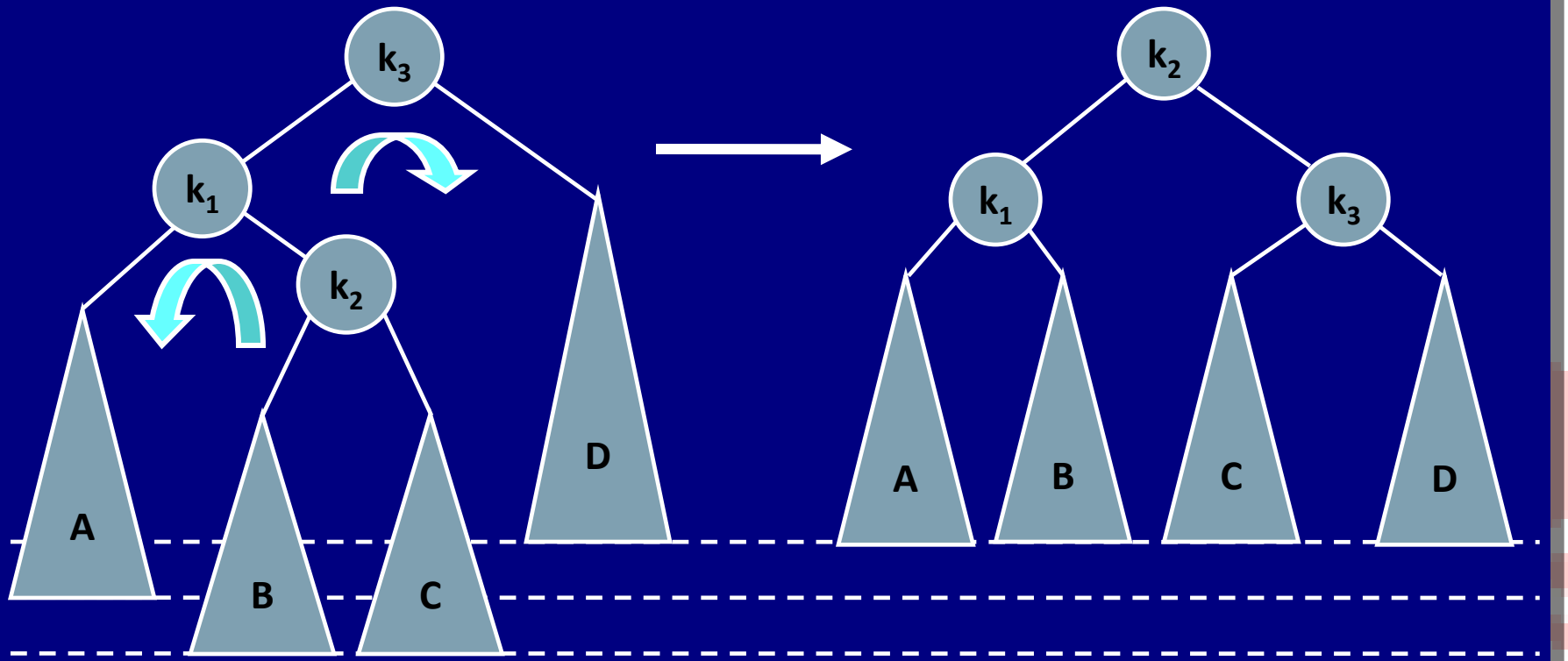
Double Rotation: Langkah



Double Rotation: Langkah

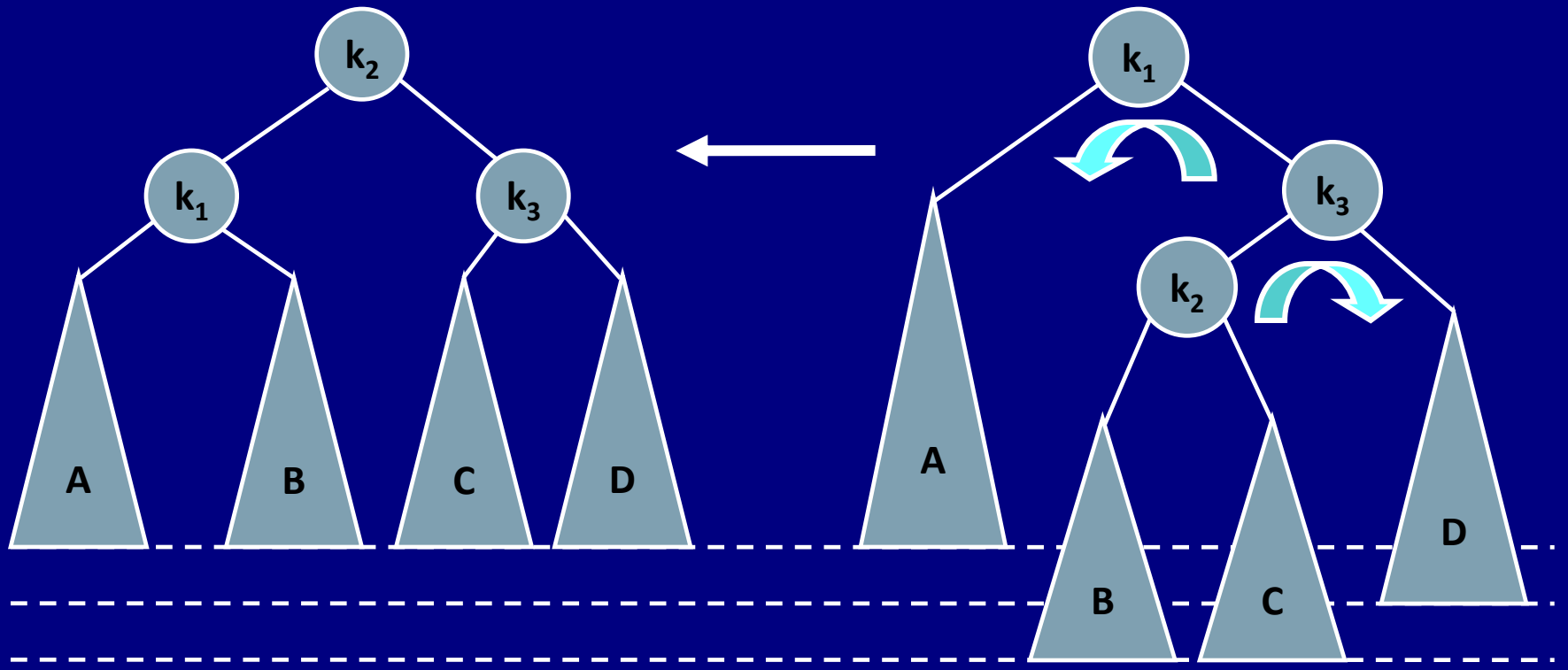


Double Rotation



$H_A = H_B = H_C = H_D$

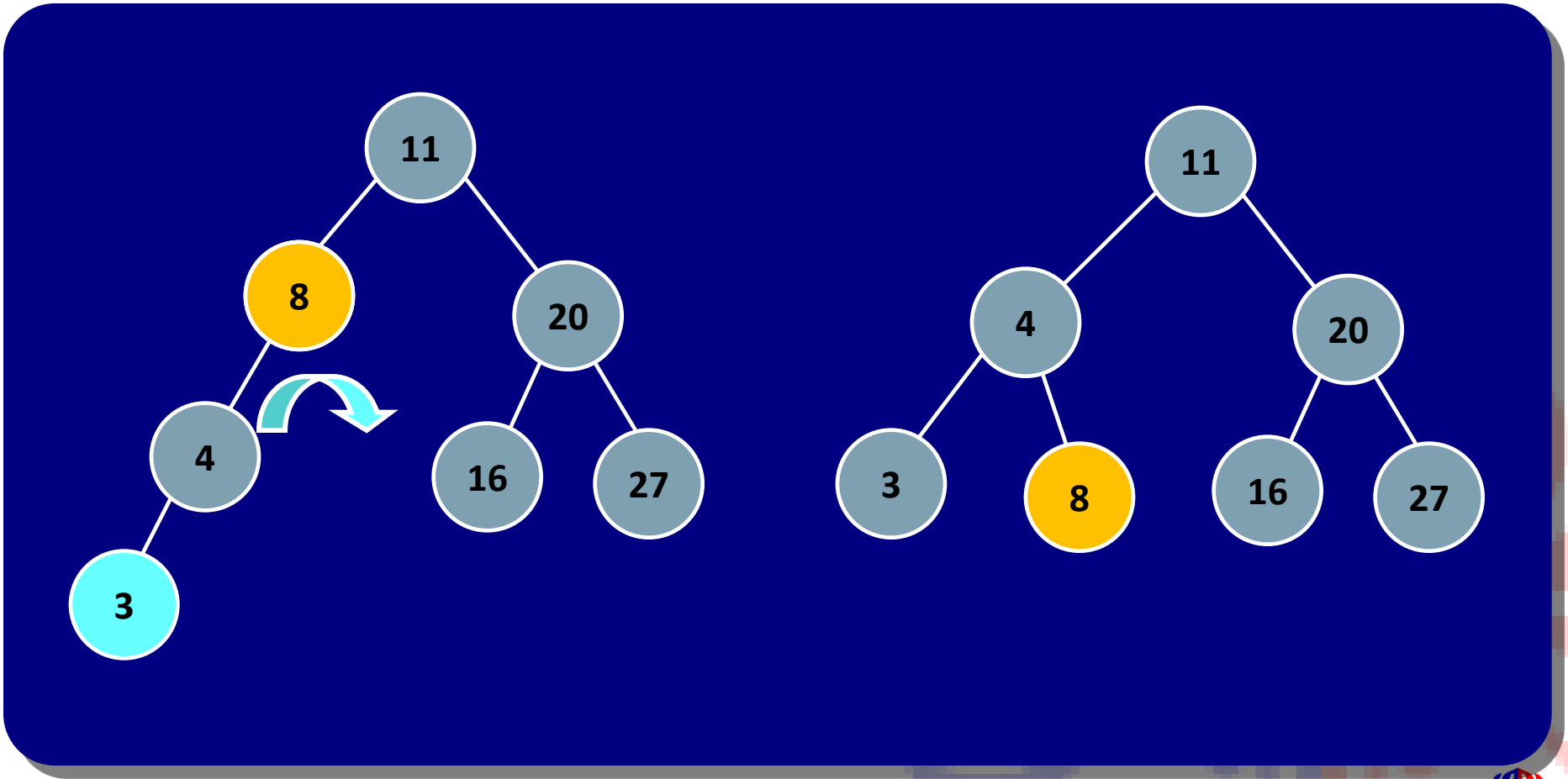
Double Rotation



$H_A = H_B = H_C = H_D$

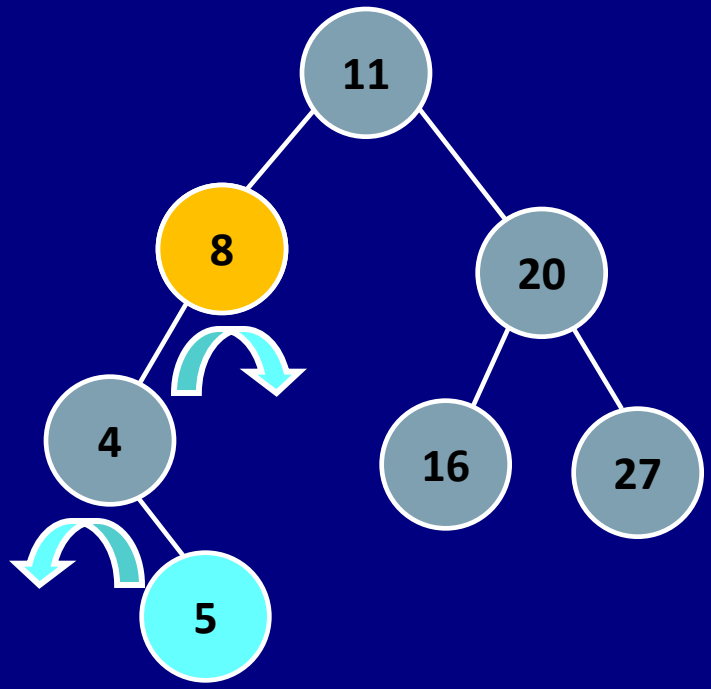
Contoh

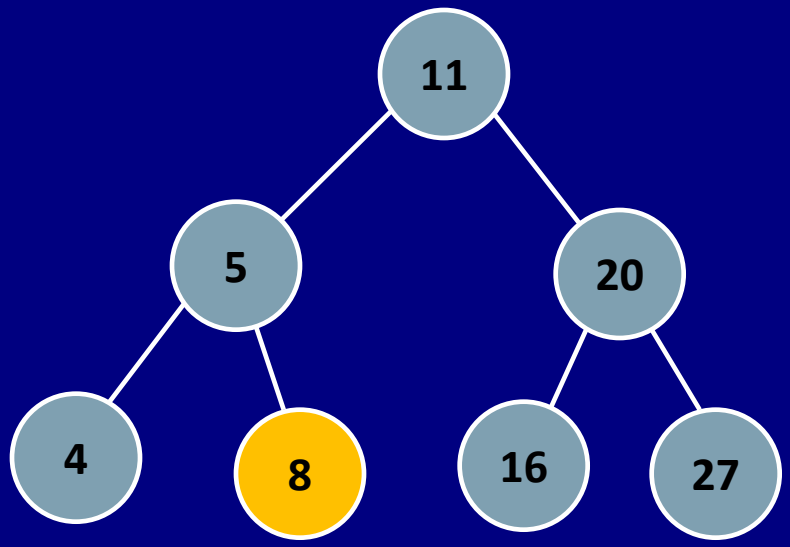
- penambahan 3 pada AVL tree



Contoh

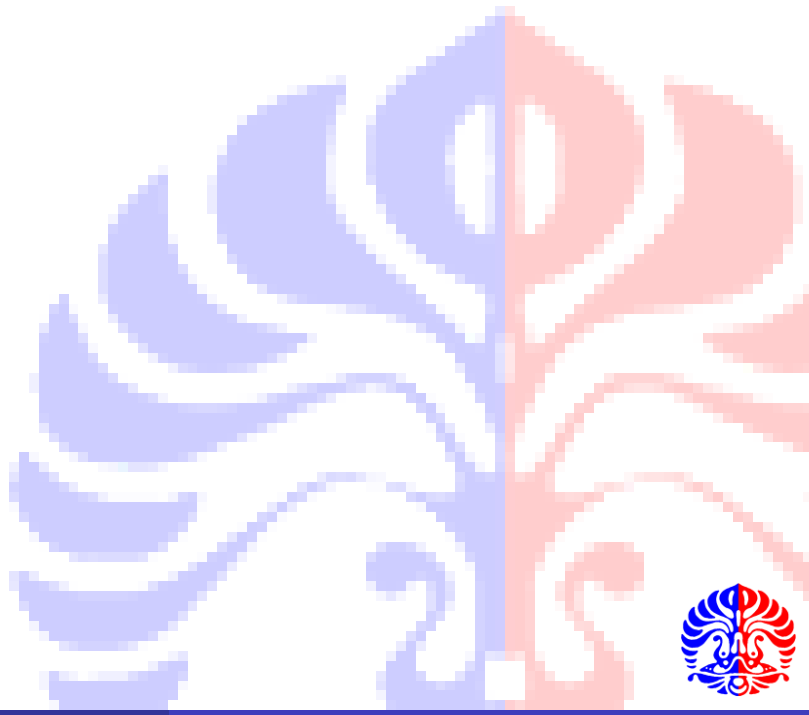
- penambahan 5 pada AVL tree





AVL Trees: Latihan

- Coba simulasikan penambahan pada sebuah AVL dengan urutan penambahan:
  - 10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55



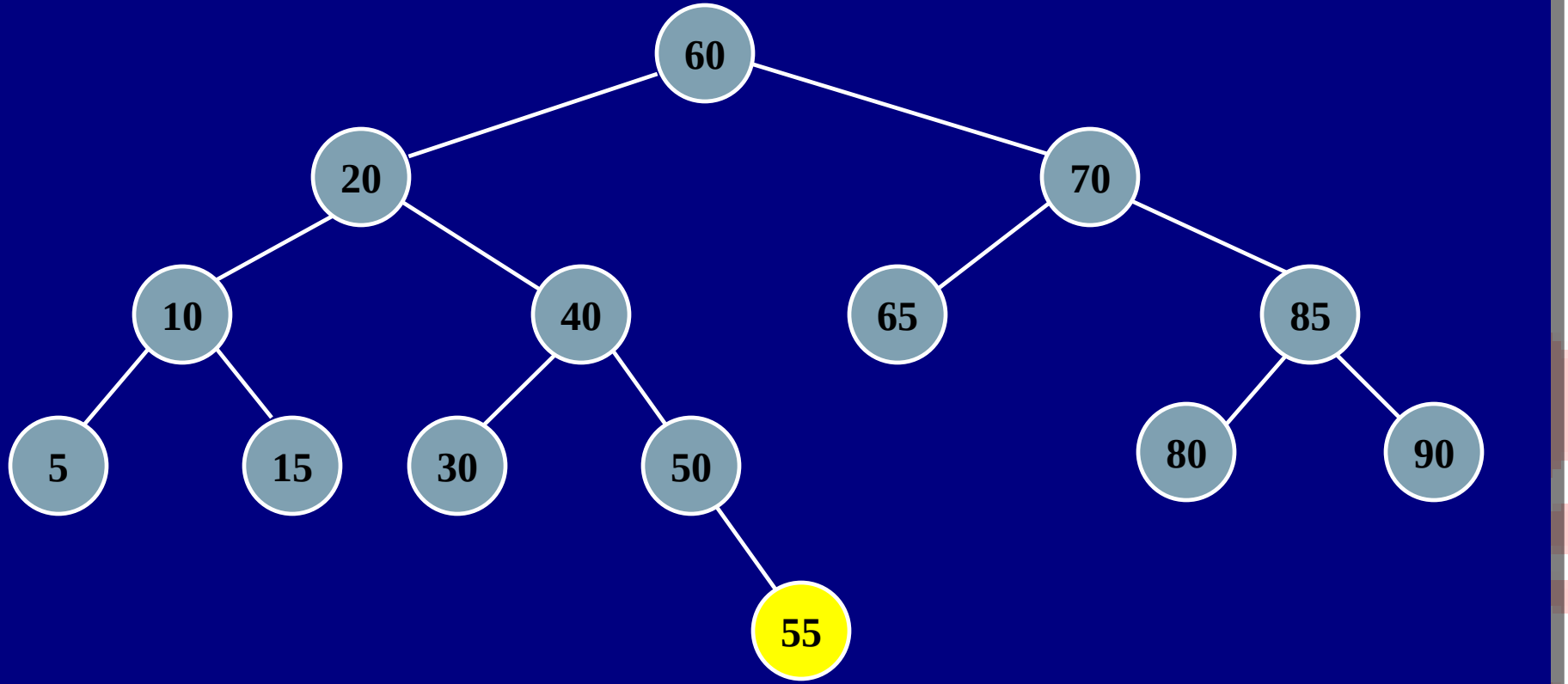
Operasi: Remove pada AVL Tree

1. Menghapus node pada AVL Tree sama dengan menghapus binary search tree procedure dengan perbedaan pada penanganan kondisi tidak *balance*.
2. Penanganan kondisi tidak balance pada operasi menghapus node AVL tree, serupa dengan pada operasi penambahan. Mulai dari node yang diproses (dihapus) periksa seluruh node pada jalur yang menuju root (termasuk root) untuk menentukan node tidak balance yang pertama
3. Terapkan ***single*** atau ***double rotation*** untuk menyeimbangkan *tree*.
4. Bila Tree masih belum balance, ulangi lagi dari langkah 2.

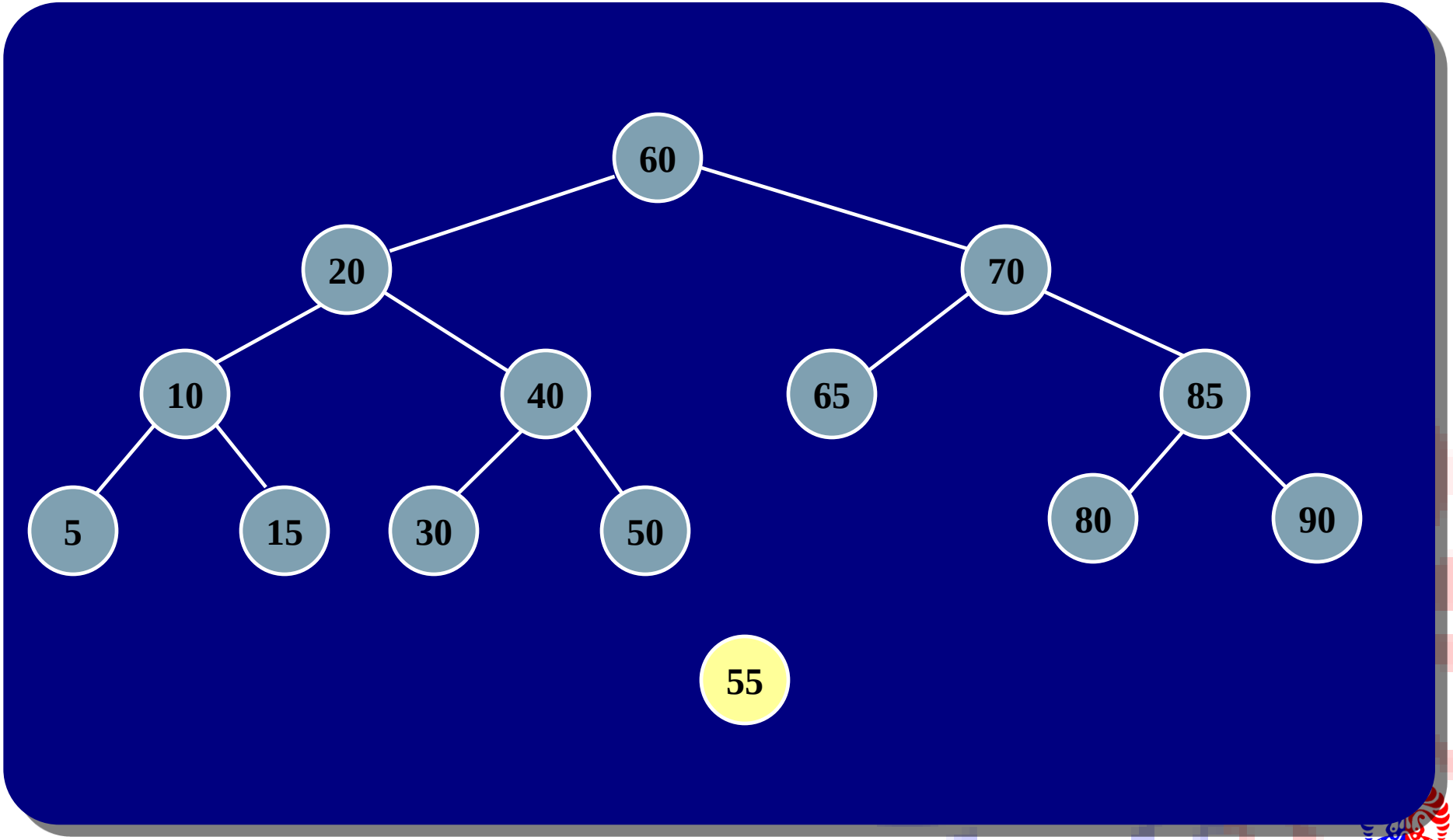
Menghapus node X pada AVL Trees

- Deletion:
  - Kasus 1: jika X adalah leaf, delete X
  - Kasus 2: jika X punya 1 child, X digantikan oleh child tsb.
  - Kasus 3: jika X punya 2 child, ganti X secara rekursif dengan predecessor-nya secara inorder
- Rebalancing

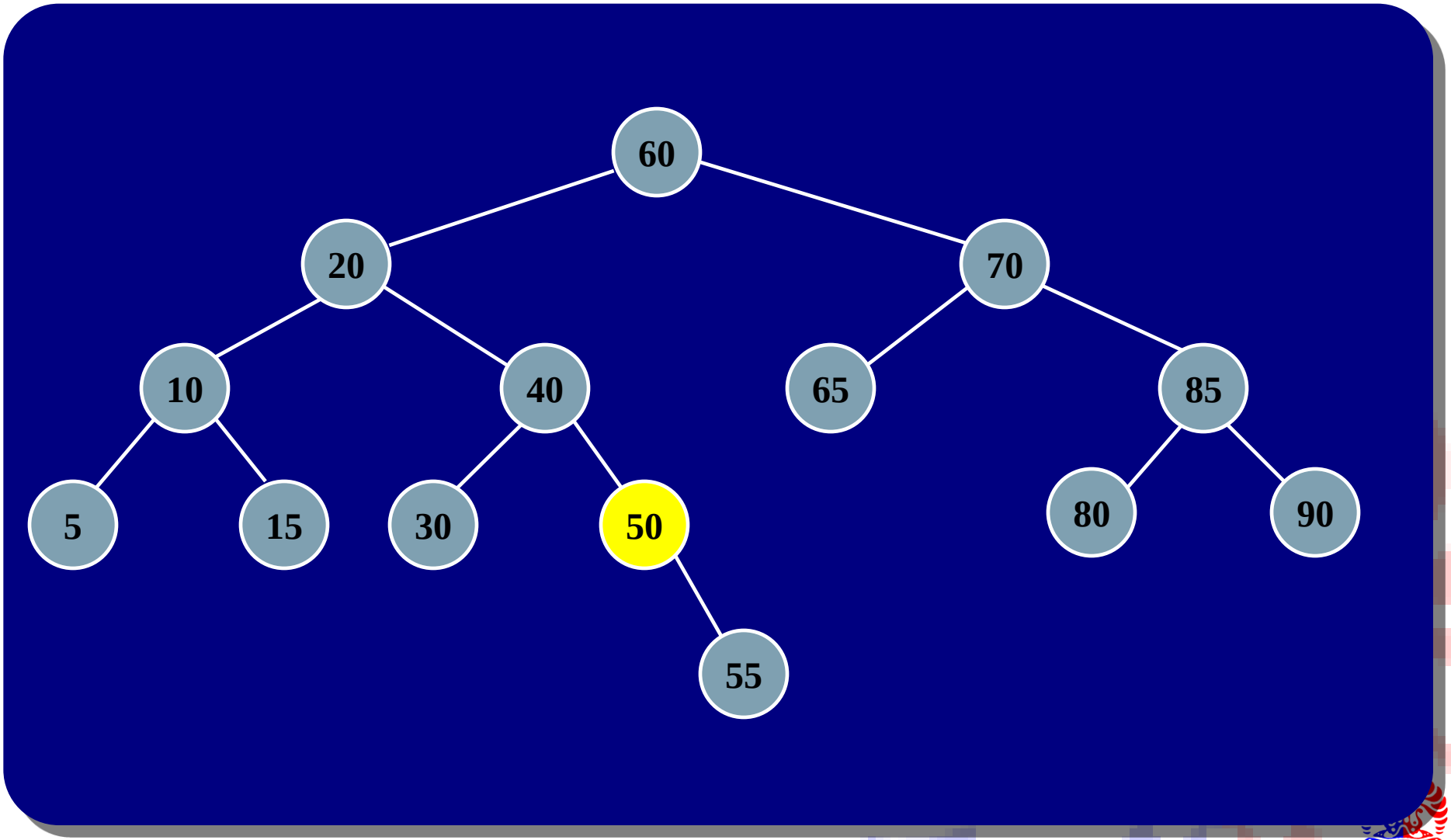
Delete 55 (Kasus 1)



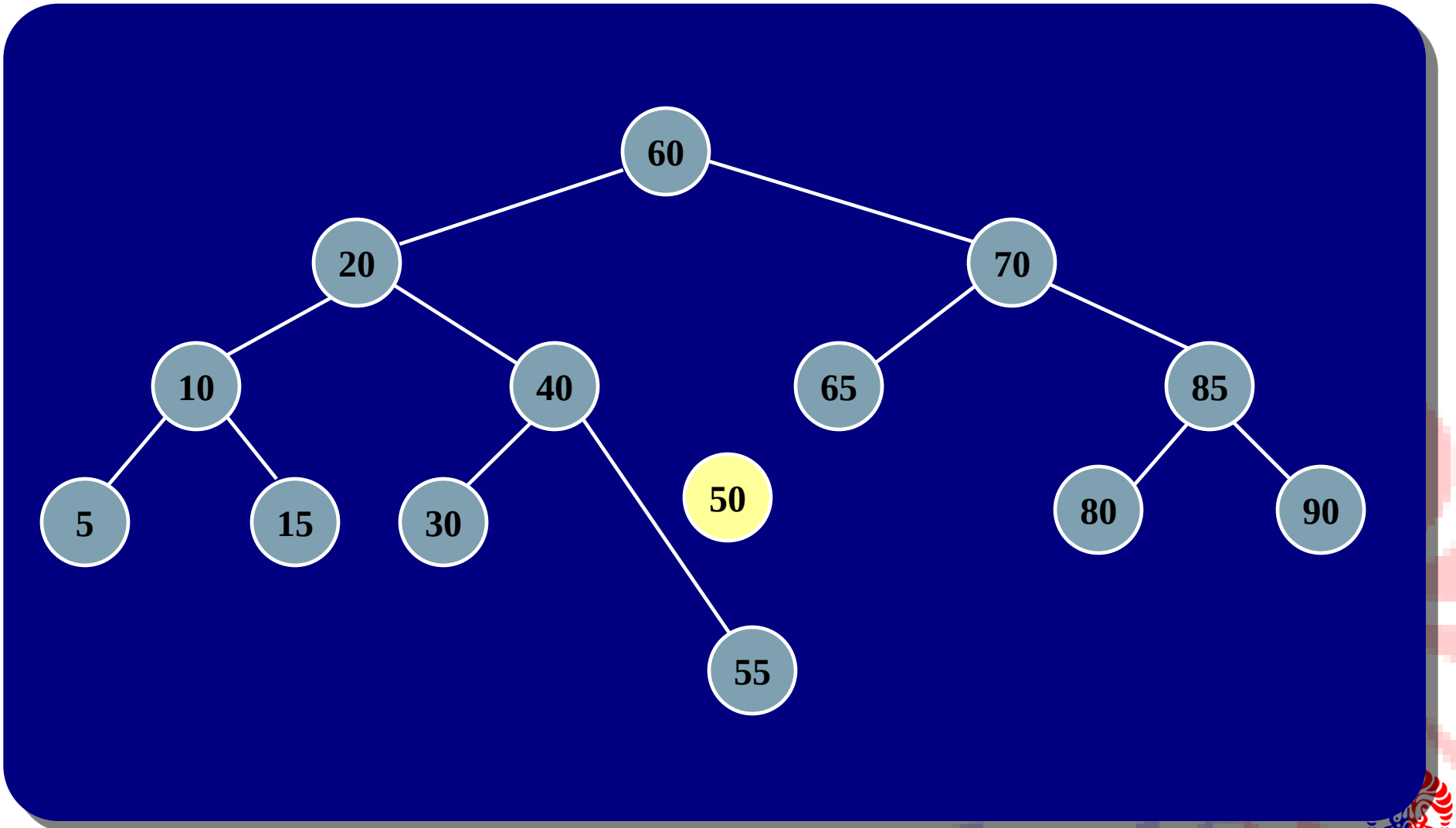
Delete 55 (Kasus 1)



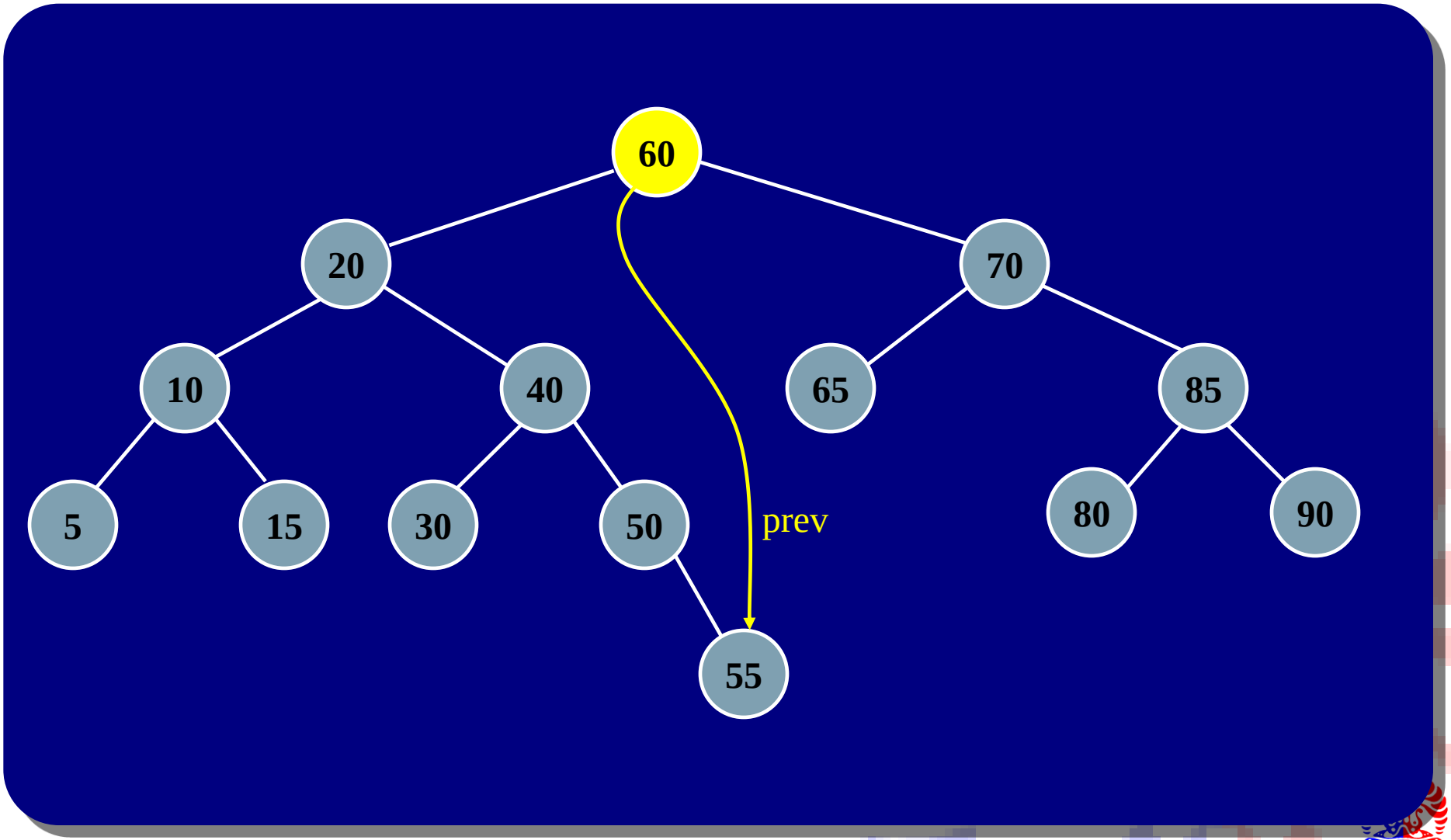
Delete 50 (Kasus 2)



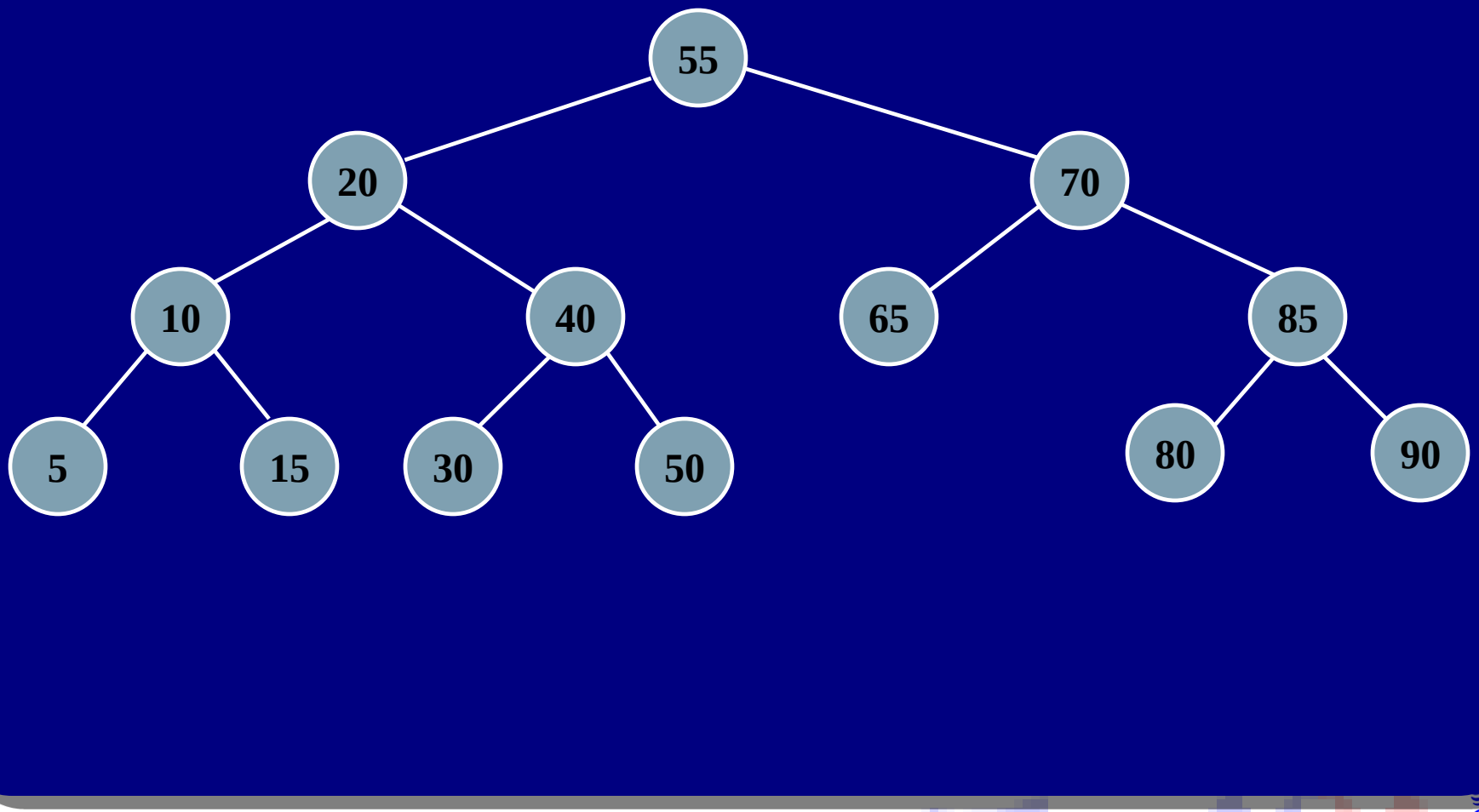
Delete 50 (Kasus 2)



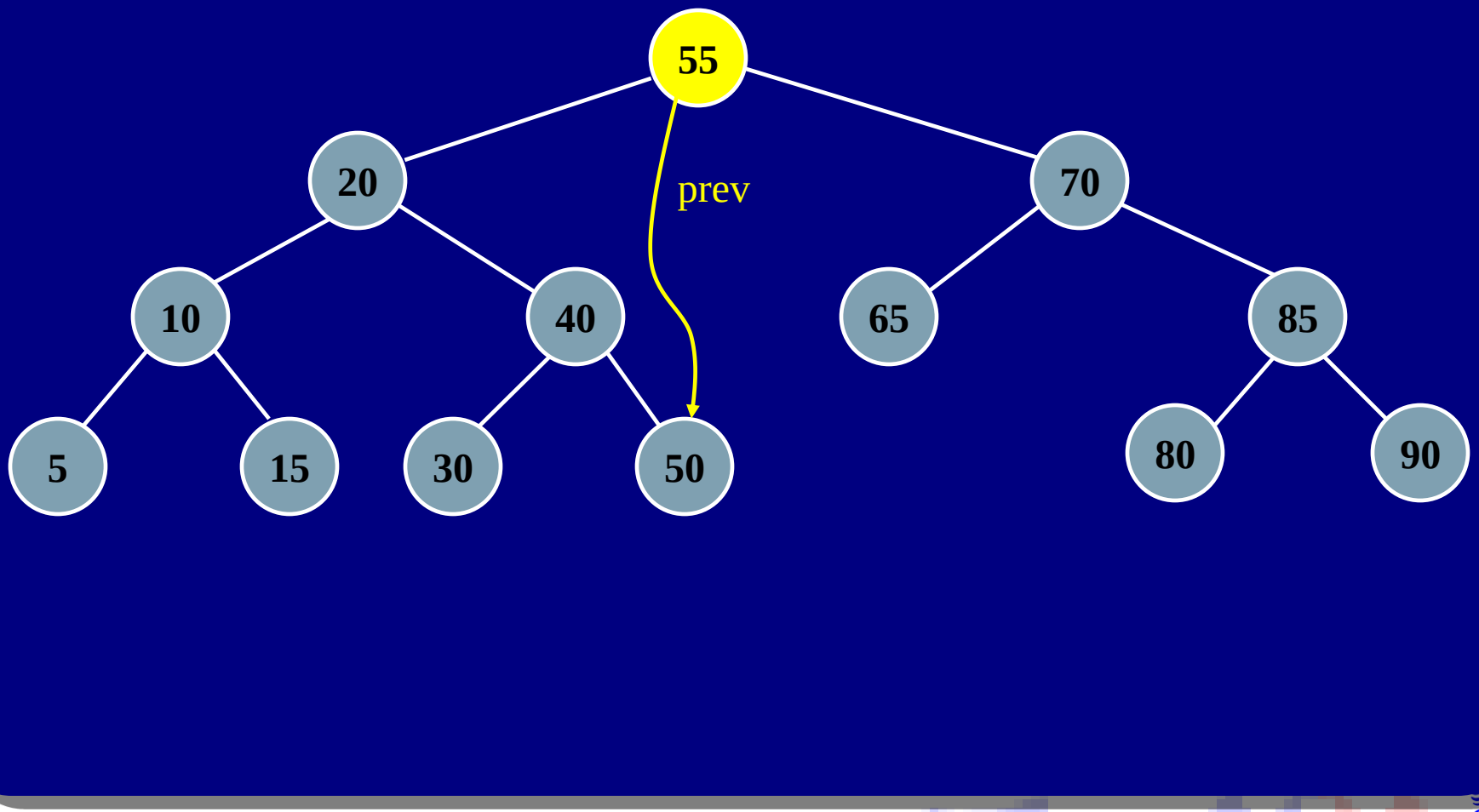
Delete 60 (Kasus 3)



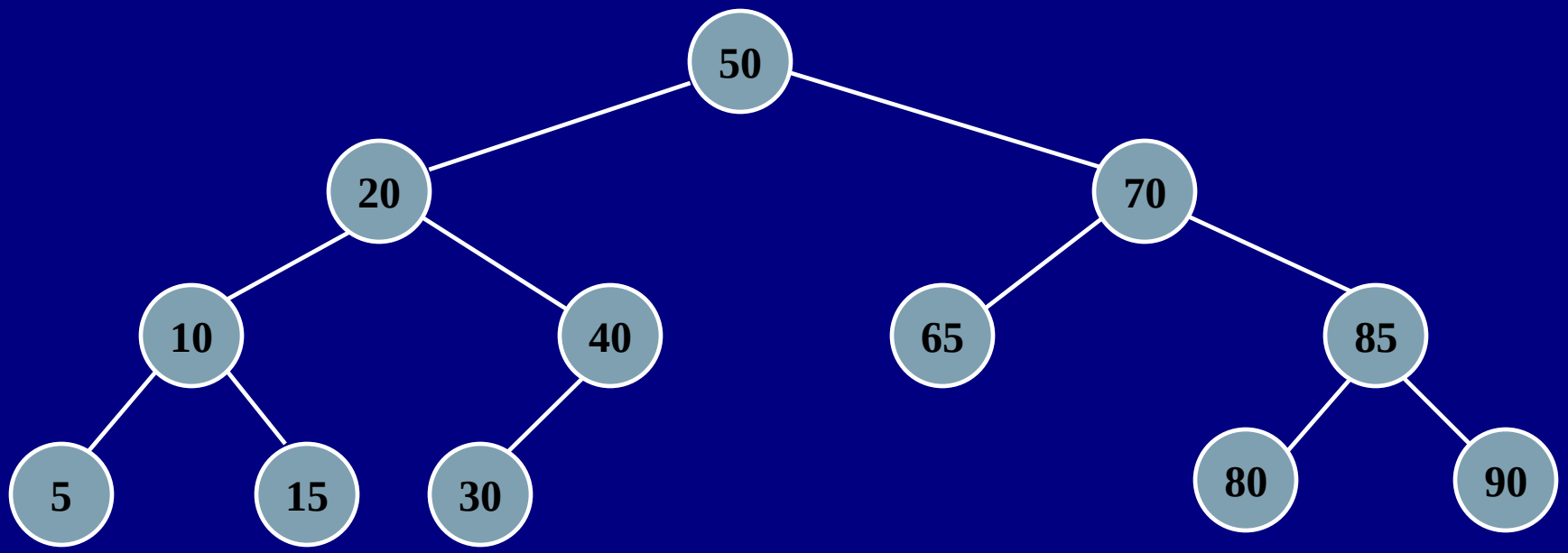
Delete 60 (Kasus 3)



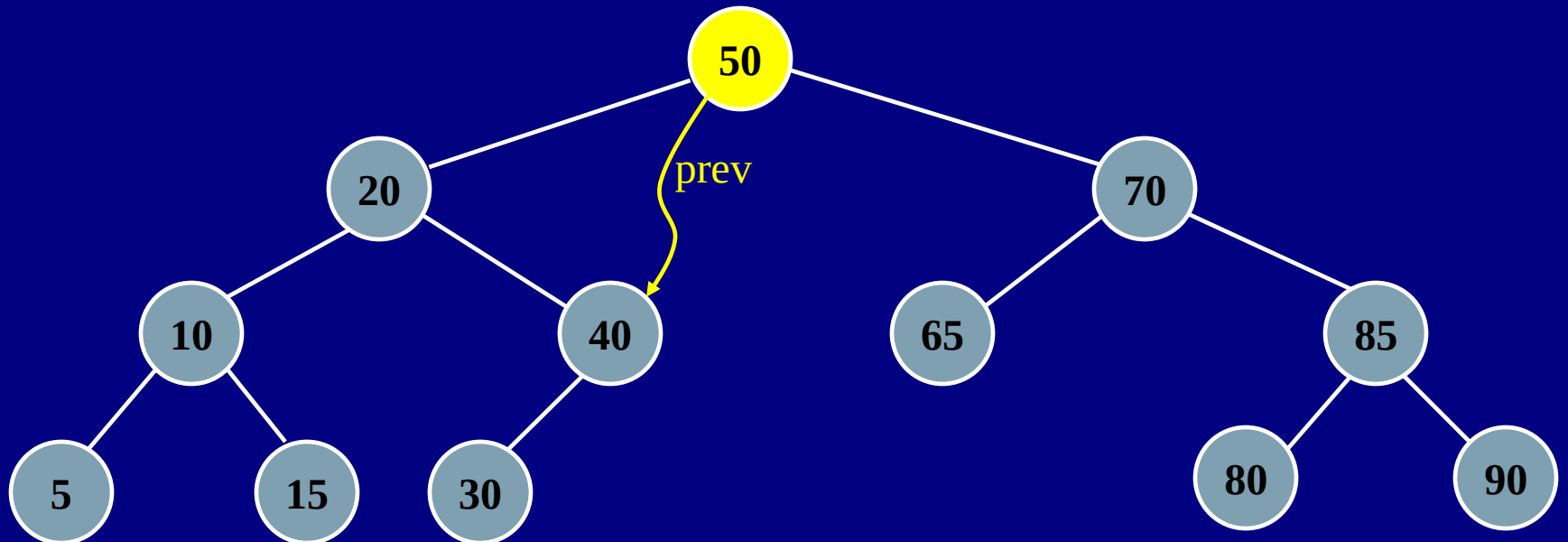
Delete 55 (Kasus 3)



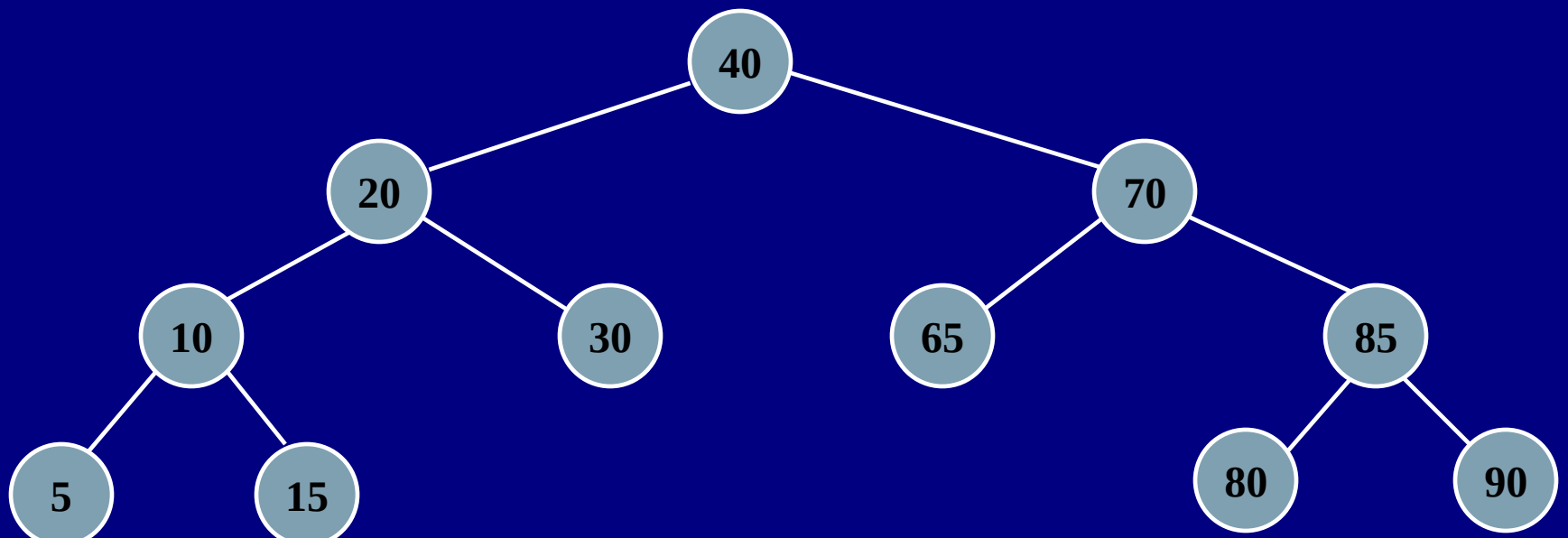
Delete 55 (Kasus 3)



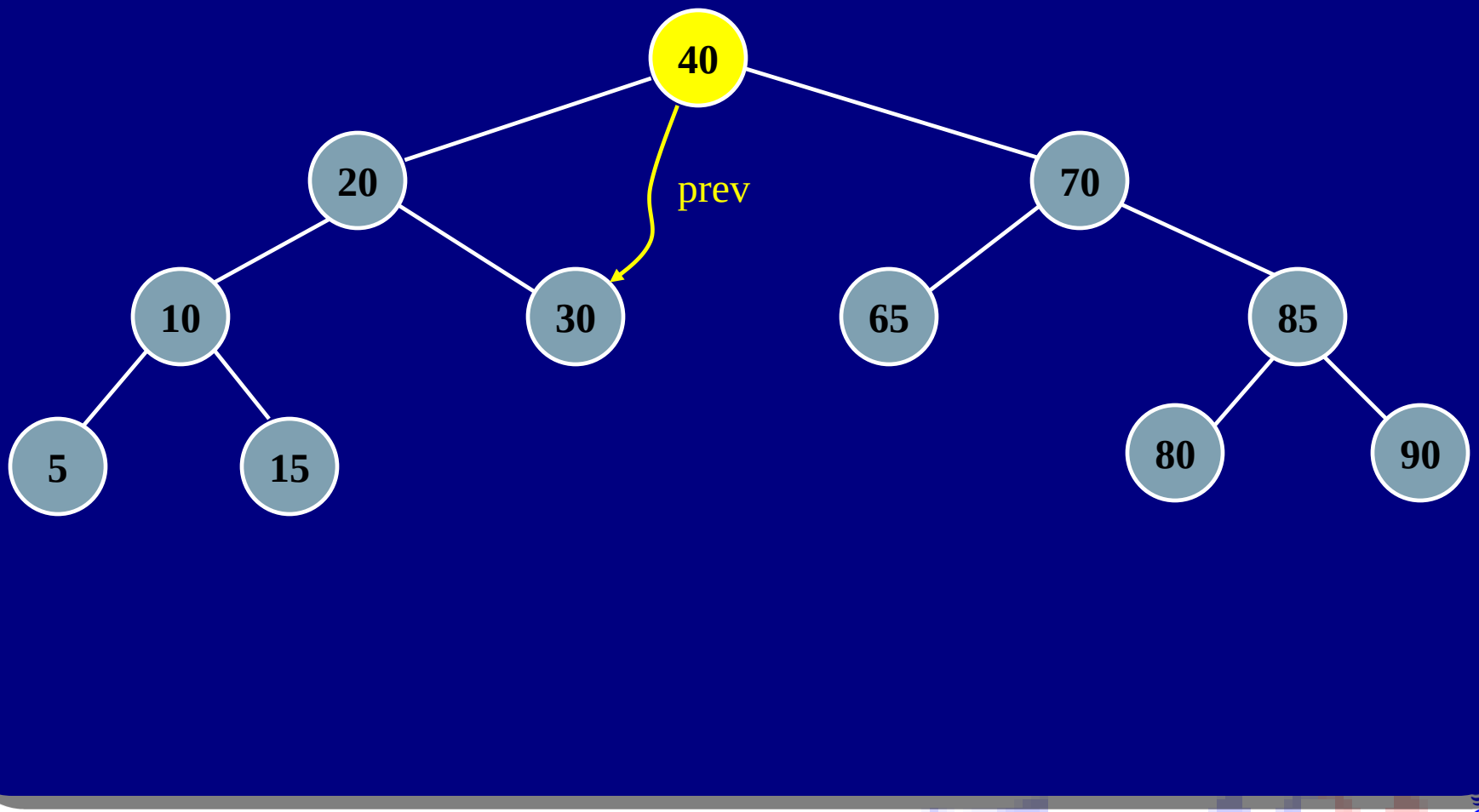
Delete 50 (Kasus 3)



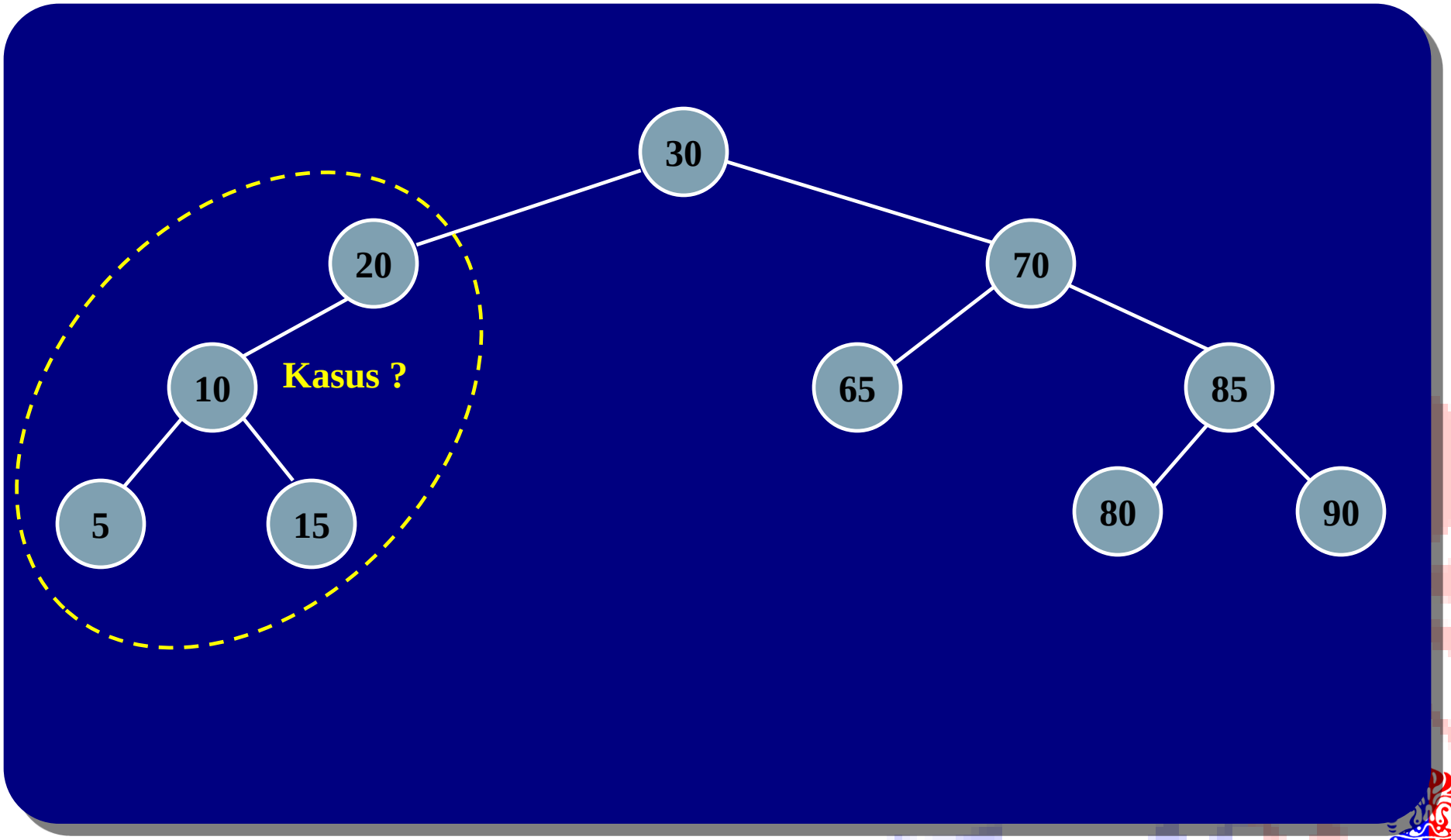
Delete 50 (Kasus 3)



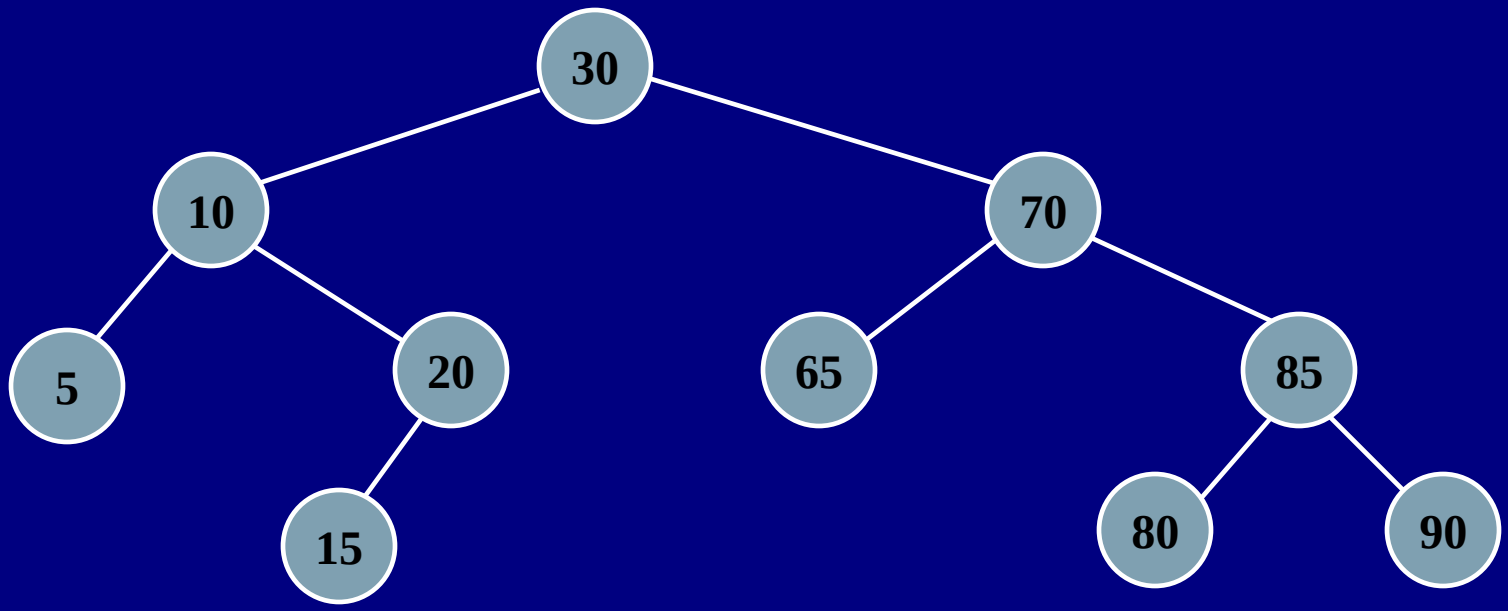
Delete 40 (Kasus 3)



Delete 40 : Rebalancing



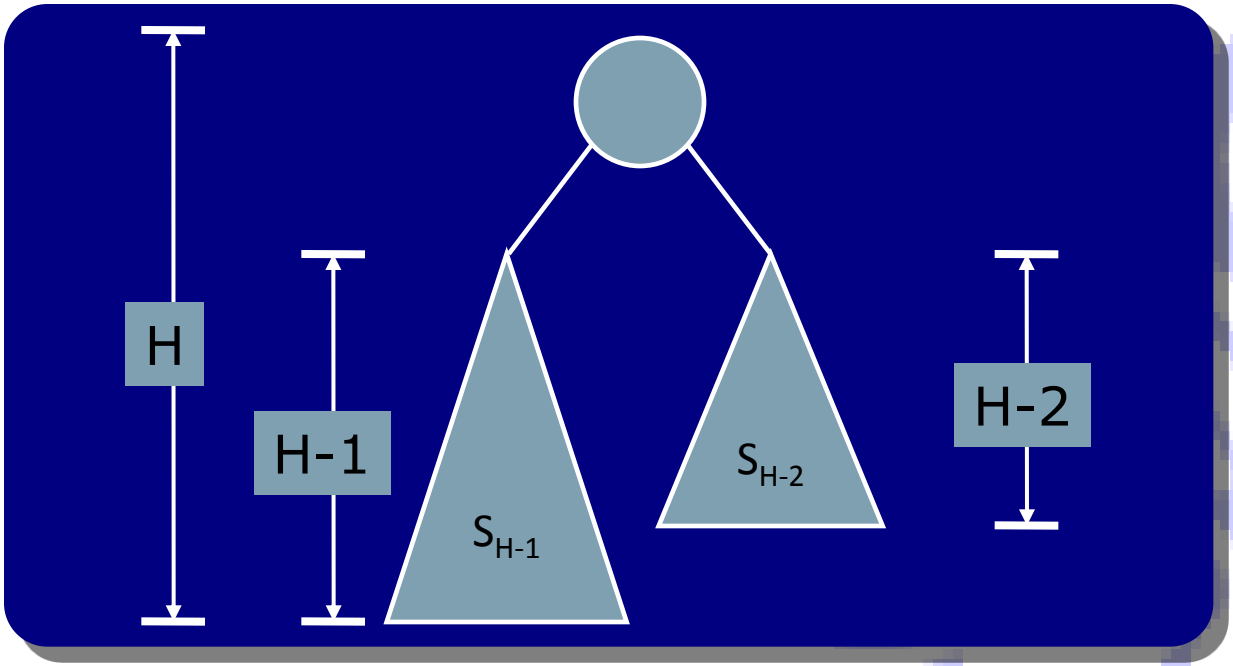
Delete 40: setelah *rebalancing*



Single rotation is preferred!

Jumlah node minimum pada AVL Tree

- Sebuah AVL Tree dengan tinggi H memiliki paling tidak sebanyak F_{H+3}-1 nodes, dimana F_i elemen ke-i dari deret bilangan fibonacci.
- $S_0 = 1$
- $S_1 = 2$
- $S_H = S_{H-1} + S_{H-2} + 1$



Jumlah node minimum pada AVL Tree

- Sebuah AVL Tree dengan tinggi H ->min memiliki F_{H+3}-1 nodes, dimana F_i elemen ke-i dari deret bilangan fibonacci: 1,1,2,3,5,8,13,21,…

$$F_i = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^i - \left(\frac{1-\sqrt{5}}{2}\right)^i\right]$$

$$misal: \ \vartheta = \left(\frac{1+\sqrt{5}}{2}\right)$$

$$maka: F_i \approx \frac{\vartheta^i}{\sqrt{5}}$$

- Sebuah AVL Tree dengan tinggi H ->min memiliki $\frac{\vartheta^{H+3}}{\sqrt{5}} - 1$ nodes

Tinggi AVL Tree

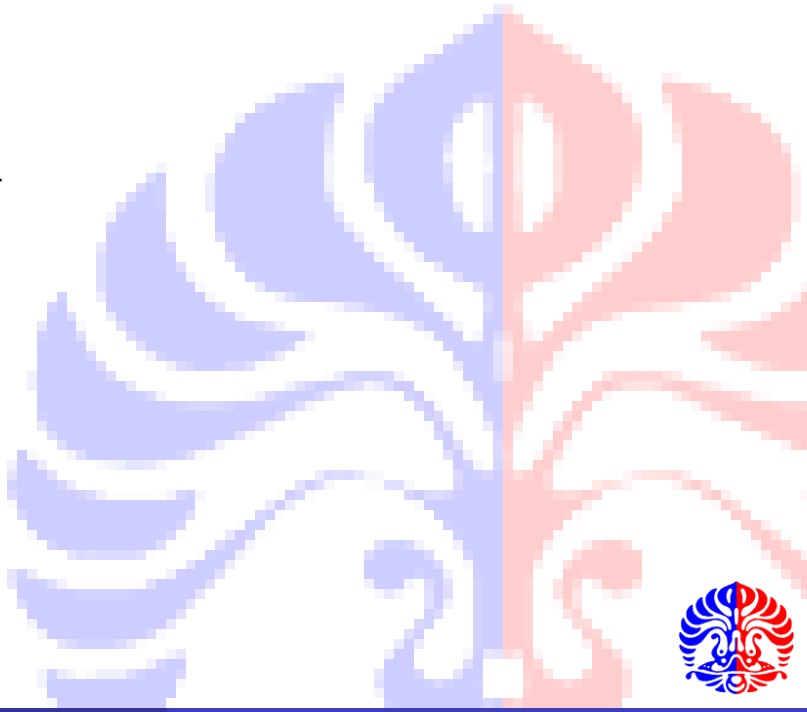
$$F_{H+3} = \frac{\vartheta^{H+3}}{\sqrt{5}} = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{H+3}}{\sqrt{5}}$$

$$\log F_{H+3} = \log \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{H+3}}{\sqrt{5}}$$

$$\log F_{H+3} = (H+3)\left(-\log\left(\frac{2}{1+\sqrt{5}}\right)\right) - \frac{\log(5)}{2}$$

$$(H+3) = \frac{\log F_{H+3} - \frac{\log(5)}{2}}{\left(-\log\left(\frac{2}{1+\sqrt{5}}\right)\right)}$$

$$H < 1.44 \log(N+2) - 1.328$$



AVL Tree: analisa (2)

- Tinggi sebuah AVL tree merupakan fungsi logaritmik dari jumlah seluruh node. (ukuran AVL Tree)
- Oleh karena itu, seluruh operations pada AVL trees juga logaritmik

Implementasi AVL Tree

- Beberapa method sama atau serupa dengan Binary Search Tree.
- Perbedaan utama terdapat pada tambahan proses balancing dengan *single* dan *double rotation.*
- Perlu tidak nya dilakukan balancing perlu diperiksa setiap kali melakukan insert dan remove.
- Kita akan pelajari lebih dalam bagaimana implementasi method **`insert`** pada AVL Tree.

Ide

- Setiap kali melakukan insert, perlu mencek pada node yang dilewati apakah node tersebut masih balance atau tidak.
- Proses *insertion* adalah *top-down*, dari *root* ke *leaf*.
- Proses pengecekan balancing adalah *bottom-up*, dari *leaf* ke *root*.

Algoritma insertion

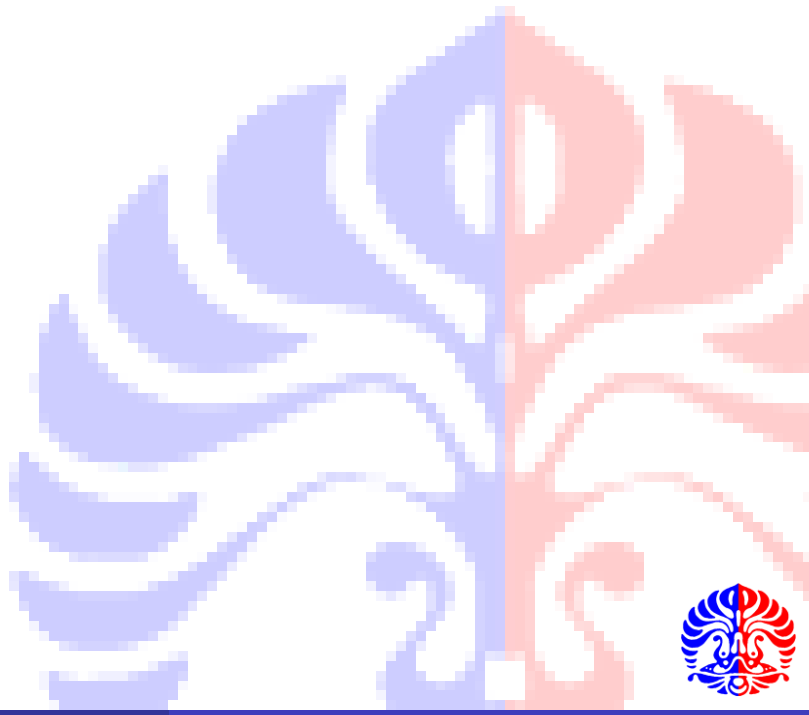
1. Letakkan node baru pada posisi yang sesuai sebagaimana pada Binary Search Tree. Proses pencarian posisi dapat dilakukan secara rekursif.
2. Ketika kembali dari pemanggilan rekursif, lakukan pengecekan apakah tiap node yang dilewati dari leaf hingga kembali ke root, apakah masih *balance* atau tidak.
3. Bila seluruh node yang dilewati hingga kembali ke root masih balance. Proses selesai.

Algoritma insertion (lanj.)

1. Untuk setiap node yang tidak balance lakukan balancing.
   a. Bila insertion terjadi pada “outside” lakukan single rotation
   b. Bila insertion terjadi pada “inside” lakukan double rotation.
2. Lakukan pengecekan dan balancing hingga *root*.

Diskusi?

- Bagaimana menentukan insertion terjadi pada bagian “inside” atau “outside” ?



Pseudo code

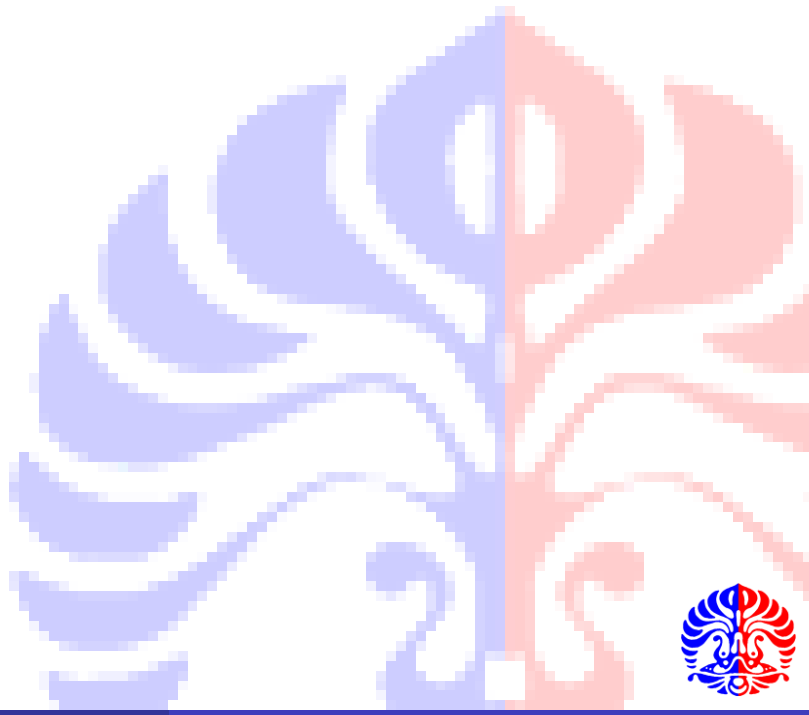
```
public static <A extends Comparable<A>> AvlNode<A>
insert( A x, AvlNode<A> t ){

    if( t == null )
        t = new AvlNode<A>( x, null, null );
    else if( x.compareTo( t.element ) < 0 )
    {
        t.left = insert( x, t.left );
        if(Math.abs(height( t.left ) - height( t.right )) == 2 )
            if( x.compareTo( t.left.element ) < 0 )
                t = rotateWithLeftChild( t );
            else
                t = doubleWithLeftChild( t );
    }
    else if( x.compareTo( t.element ) > 0 )
    {
       // simetris dengan program diatas
    }

    return t;
}
```

Diskusi

- Apakah implementasi tersebut sudah efisien?
  - Perhatikan pemanggilan method **`height`** !



Pseudo code

```
public static <A extends Comparable<A>> AvlNode<A>
insert( A x, AvlNode<A> t ){

    if( t == null )
        t = new AvlNode<A>( x, null, null );
    else if( x.compareTo( t.element ) < 0 )
    {
        t.left = insert( x, t.left );
        if(Math.abs(height( t.left ) - height( t.right )) == 2 )
            if( x.compareTo( t.left.element ) < 0 )
                t = rotateWithLeftChild( t );
            else
                t = doubleWithLeftChild( t );
    }
    else if( x.compareTo( t.element ) > 0 )
    {
       // simetris dengan program diatas
    }

    return t;
}
```

Pseudo code AvlNode

```
class AvlNode<A extends Comparable<A>> extends BinaryNode<A>{
        // Constructors
    AvlNode( A theElement )     {
        this( theElement, null, null );
    }

    AvlNode( A theElement, AvlNode<A> lt, AvlNode<A> rt ) {
        element  = theElement;
        left     = lt;
        right    = rt;
        height   = 0;
    }

    public int height(){
        return t.height;
    }

        // Friendly data; accessible by other package routines
    A           element;      // The data in the node
    AvlNode<A> left;          // Left child
    AvlNode<A> right;         // Right child
    int         height;       // Height
}
```

Pseudo code

```
public static <A extends Comparable<A>> AvlNode<A>
insert( A x, AvlNode<A> t ){
    if( t == null )
        t = new AvlNode<A>( x, null, null );
    else if( x.compareTo( t.element ) < 0 )
    {
        t.left = insert( x, t.left );
        if(t.left.height - t.right.height) == 2 )
            if( x.compareTo( t.left.element ) < 0 )
                t = singleRotateWithLeftChild( t );
            else
                t = doubleRotateWithLeftChild( t );
    }
    else if( x.compareTo( t.element ) > 0 )
    {
       // simetris dengan program diatas
    }

    t.height = Math.max(t.left.height, t.right.height) + 1;
    return t;
}
```

Diskusi

- Apakah ada cara lain?
  - Hanya menyimpan nilai perbandingan saja.
    - Nilai -1, menyatakan sub tree kiri lebih tinggi 1 dari sub tree kanan.
    - Nilai +1, menyatakan sub tree kanan lebih tinggi 1 dari sub tree kiri
    - Nilai 0, menyatakan tinggi sub tree kiri = tinggi sub tree kanan
  - Kapan dilakukan rotasi?
    - Bila harus diletakkan ke kiri dan node tersebut sudah bernilai -1 maka dinyatakan tidak balance
    - Berlaku simetris

Pseudocode: Single rotasi

```
static <A extends Comparable<A>> AvlNode<A>
singleRotateWithLeftChild( AvlNode<A> k2 )
{
    AvlNode<A> k1 = k2.left;
    k2.left = k1.right;
    k1.right = k2;
    // update tinggi kedua node.
    return k1;
}
```

Pseudocode: Double rotasi

```
static <A extends Comparable<A>> AvlNode<A>
doubleRotateWithLeftChild( AvlNode<A> k3 )
{
    k3.left = singleRotateWithRightChild( k3.left );
    return singleRotateWithLeftChild( k3 );
}
```

AVL Trees: Latihan

- Coba simulasikan urutan proses pada sebuah AVL Tree berikut ini
  - insert 10, 85, 15, 70, 20, 60, 30,
  - delete 15, 10,
  - insert 50, 65, 80,
  - delete 20, 60,
  - insert 90, 40, 5, 55
  - delete 70
- Gambarkan kondisi akhir dari AVL Tree tersebut.

Rangkuman

- Mencari elemen, menambahkan, dan menghapus, seluruhnya memiliki kompleksitas running time O(log n) pada kondisi worst case
- Insert operation: top-down insertion dan bottom up balancing